4diac Libraries: FB & Adapter Reference (4.1)

Franz Höpfinger

Contents

1 4diac Libraries: FB & Adapter Reference 5

2 📚 Libraries 7

2.1 External Libraries 7
2.1.1 BlinkMarine 7
2.1.2 DataPanel 13
2.1.3 Radio 66
2.1.4 MyLib (test_AX) 71
2.1.5 MyLib_AX_Option 579
2.1.6 MyLib (test_B) 587
2.1.7 MyLib_OPC 750
2.1.8 OSCAT 754
2.1.9 Option 820
2.1.10 SafeArithmetic 849
2.1.11 adapter 859
2.2 Status Overview 3172
2.3 Status Overview 3174
2.4 Comparison with Similar Components 3190
2.5 Comparison with Similar Components 3192
2.6 Comparison with Similar Components 3560
2.7 Comparison with Similar Components 3578
2.8 Comparison with Similar Modules 3581
2.9 Comparison with Similar Components 4285
2.10 Comparison with Similar Components 4290
2.10.1 iec61131-3-bool 4417
2.10.2 isobus 4520
2.10.3 logiBUS 4972
2.11 Standard Libraries 5655
2.11.1 convert 5655
2.11.2 core 5714
2.11.3 events 5715
2.11.4 iec61131-3 5839
2.11.5 🔌 io 6330
2.11.6 net 6397
2.11.7 powerlink 6526
2.11.8 reconfiguration 6541
2.11.9 rtevents 6556
2.11.10 signal processing 6579
2.11.11 system 6589
2.11.12 utils 6592
2.12 ⚖️ Comparison with other standards 6674
2.12.1 IEC 61131-3 (Old Standard) 6674

3 static 6679
3.1 Fonts 6679
3.1.1 TGL Normschrift – _std Font Variants 6679

1. 4diac Libraries: FB & Adapter Reference

This wiki (**4.1**) holds the auto-generated Function Block and Adapter reference for Eclipse 4diac (standard and external libraries), split out from the main wiki Visual Programming Languages: Low-Code (4).

👉 Go to the library overview

- 📄 Download PDF Handbook

2. 📚 Libraries

2.1 External Libraries

2.1.1 BlinkMarine

2.1.1.1 🔌 io

📺 Video

- Automation of the Future
- The Great Migration
- Digital Decisions
- Ideal Diodes
- The Dual Wi-Fi Solution

🎧 Podcast

- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Target Groups, Structure, and Unique Selling Proposition
- Taming the Software Dragon: Industrial Automation and the Future of Production
- Eclipse 4diac: Innovation Engine Research vs. User Needs – What Really Drives Development?
- Taming the Software Dragon: Why IEC 61499 Could Revolutionize Industry

---

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Diode & Semiconductor Basics on ms-muc-docs.de
- 🌐 Smart Switch vs. Relay on ms-muc-docs.de

DI

📺 Video

- Decoding the datasheet
- The Great Migration
- The Art of Soldering
- The World of Standardization
- Digital Decisions

🎧 Podcast

- 4diac IDE: Your “Hello World” of Automation – The Blinking Tutorial
- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- 4diac IDE: How the IEC 61499 Standard is Shaping Industrial Automation Revolutionized
- 4diac presentation: Target groups, structure, and unique selling proposition Sharpening
- Taming the software dragon: Industrial automation and the future of production

---

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

PKP-2200-LI

PKP_2200



• – ◦ • – ◦ • – ◦ •

Introduction

PKP_2200 is a global constant for the Blink Marine PowerKey interface. This constant serves as the default value for configuring digital input functionalities in the Blink Marine system.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

No adapters available

Functionality

PKP_2200 defines a global constant of type USINT (Unsigned Short Integer) with the hexadecimal value 16#21 (decimal 33). This constant is used as the default value for the PowerKey configuration.

Technical Specifications

- **Data Type**: USINT (8-bit unsigned integer)
- **Initial Value**: USINT#16#21 (hexadecimal 21, decimal 33)
- **Packet Mapping**: BlinkMarine::io::DI

State Overview

As a global constant, PKP_2200 has no state transitions and retains its fixed value throughout its runtime.

Application Scenarios

- Default configuration for Blink Marine PowerKey interfaces
- Use as a reference value in digital input function blocks
- Configuration of I/O parameters in marine control systems

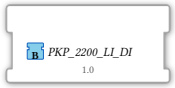
⚖️ Comparison with Similar Blocks

Unlike function blocks with dynamic behavior, PKP_2200 is a static constant without any behavioral logic. It serves solely to provide a fixed configuration value.

Conclusion

PKP_2200 represents an essential building block for configuring digital input interfaces in the BlinkMarine system. Its use as a global constant enables centralized and consistent configuration across different function blocks.

PKP_2200_LI_DI



• – ◦ • – ◦ • – ◦ •

Introduction

The PKP_2200_LI_DI is a global constant definition for Blink Marine PowerKey Digital Inputs. This constant defines the pin assignments for the digital inputs of a PowerKey system and provides predefined configurations for various keys as well as an invalid state.

Interface Structure

Event Inputs

No event inputs available - this is a global constant definition

Event Outputs

No event outputs available - this is a global constant definition

Data Inputs

No data inputs available - this is a global constant definition

Data Outputs

No data outputs available - this is a global constant definition

Adapters

No adapters available - this is a global constant definition

Functionality

The PKP_2200_LI_DI defines global constants of type `PKP_2200_LI_DI_S`, which specify the pin numbers for digital inputs of a Blink Marine PowerKey system. Each constant represents a specific key configuration with associated hardware pins.

- **Data Type**: All constants use the structure type `PKP_2200_LI_DI_S`
- **Pin Assignments**:
- DigitalInput_Key_1: Pin 1
- DigitalInput_Key_2: Pin 2
- DigitalInput_Key_3: Pin 3
- DigitalInput_Key_4: Pin 4
- Invalid: Pin 255 (invalid state)
- **Package Membership**: BlinkMarine::io::DI

State Overview

No state machine present - this is a static constant definition

Application Scenarios

- Configuration of digital inputs in marine control systems
- Hardware abstraction for PowerKey key assignments
- Simplification of pin mapping in IEC 61499 applications
- Error handling through defined invalid state

⚖️ Comparison with similar components

Unlike Function blocks with active logic are simply constant definitions that do not perform any active processing. They serve solely for the centralized management of hardware configurations.

Conclusion

The PKP_2200_LI_DI provides a standardized and maintainable solution for the pin configuration of digital inputs in Blink Marine PowerKey systems. Centralized definition of hardware assignments ensures consistency across different applications and improves code reusability.

PKP_2200_LI_DI_S

Blink Marine PowerKey DI Struct

Structure

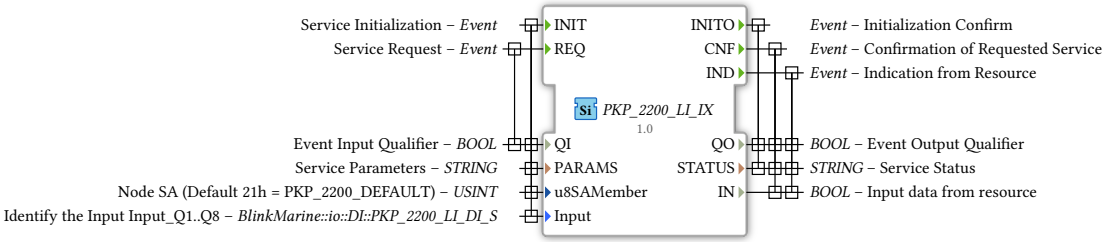
Name	Type	Initial Value
Pin	USINT	255

PKP_2200_LI_IX

• – ◦ • – ◦ • – ◦ •

Introduction

The PKP_2200_LI_IX is a service interface function block for Boolean input data. This block serves as an interface for digital inputs and enables communication with corresponding hardware resources.



Interface Structure

Event Inputs

- **INIT**: Service initialization with parameters QI, PARAMS, u8SAMember, and Input
- **REQ**: Service request with parameter QI

Event Outputs

- **INITO**: Initialization acknowledgment with parameters QO and STATUS
- **CNF**: Acknowledgement of the requested service operation with parameters QO, STATUS, and IN
- **IND**: Indication from the resource with parameters QO, STATUS, and IN

Data Inputs

- **QI**: Event input qualifier (BOOL)
- **PARAMS**: Service parameter (STRING)
- **u8SAMember**: Node SA (USINT) - Default value: PKP_2200::PKP_2200_DEFAULT (21h)
- **Input**: Identifies input Input_Q1..Q8 (Type: BlinkMarine::io::DI::PKP_2200_LI_DI_S) - Initial value: Invalid

Data Outputs

- **QO**: Event output qualifier (BOOL)
- **STATUS**: Service status (STRING)
- **IN**: Input data from the resource (BOOL)

Adapter

No adapter interfaces are available.

Functionality

This function block initializes and manages communication with digital input resources. The INIT event initializes the service with the appropriate parameters. The REQ event allows service operations to be requested. The INITO, CNF, and IND output events return acknowledgments and status information.

Technical Features

- Uses specific PKP_2200 communication parameters
- Supports up to 8 digital inputs (Q1-Q8)
- Includes default values for SA members (21h = PKP_2200_DEFAULT)
- Implements invalid initialization for inputs

State Transitions

The function block goes through the following states:

1. **Initialization Phase**: INIT event processed, INITO response generated
2. **Service Operation**: REQ event processed, CNF response generated
3. **Resource Indication**: Automatic IND events upon state changes

Application Scenarios

- Connecting digital input sensors in automation systems
- Integration into marine applications (BlinkMarine environment)
- Communication with PKP_2200-compatible hardware
- Monitoring of switching states and signal levels

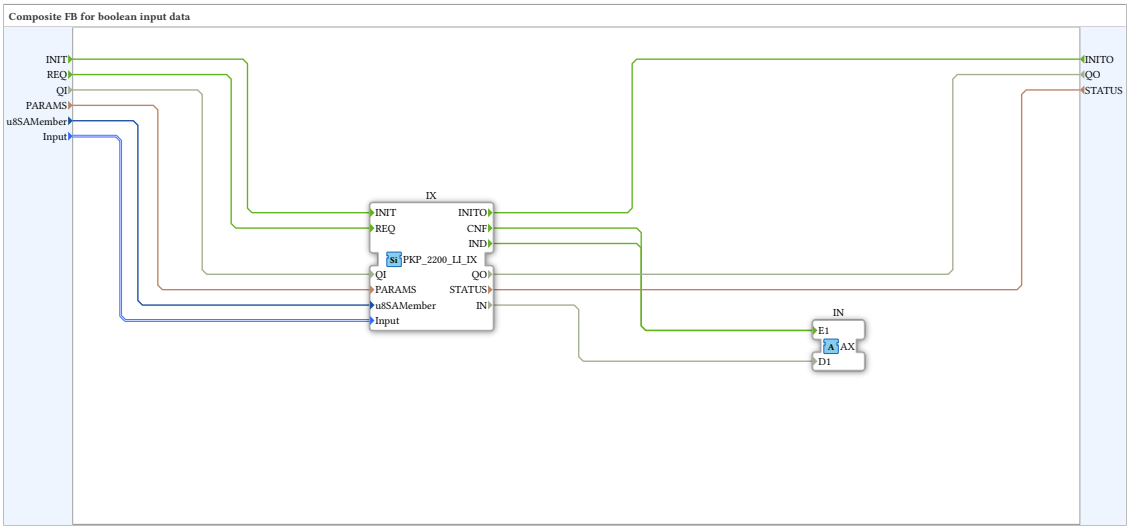
⚖️ Comparison with Similar Function Blocks

Compared to simple The PKP_2200_LI_IX offers enhanced service functionalities for digital input modules, including status feedback and parameterization options. Its specific PKP_2200 integration distinguishes it from generic I/O modules.

Conclusion

The PKP_2200_LI_IX is a specialized service interface module for the reliable connection of digital inputs in industrial automation systems, particularly in marine environments. Its enhanced functionality, including status feedback and parameterization options, makes it a robust solution for demanding I/O applications.

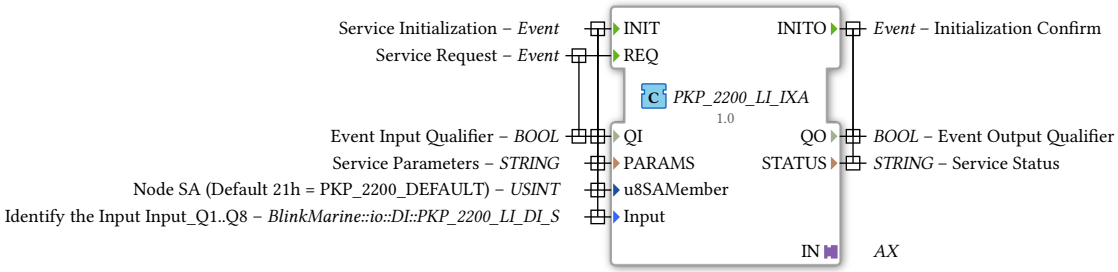
PKP_2200_LI_IXA



• — ○ • — ○ • — ○ •

Introduction

The PKP_2200_LI_IXA is a composite function block for processing Boolean input data. It serves as an interface for digital inputs and provides standardized initialization and query functions for input signals.



Interface Structure

Event Inputs

- **INIT**: Service Initialization (with QI, PARAMS, u8SAMember, Input)
- **REQ**: Service Request (with QI)

Event Outputs

- **INITO**: Initialization Acknowledgement (with QO, STATUS)

Data Inputs

- **QI**: Event Input Qualifier (BOOL)
- **PARAMS**: Service Parameters (STRING)
- **u8SAMember**: Node SA (USINT) - Default Value: PKP_2200::PKP_2200_DEFAULT
- **Input**: Input Identifier (BlinkMarine::io::DI::PKP_2200_LI_DI_S) - Default Value: Invalid

Data Outputs

- **QO**: Event Output Qualifier (BOOL)
- **STATUS**: Service Status (STRING)

Adapters

- **IN**: Unidirectional Adapter of Type AX

Functionality

The function block initializes itself via the INIT event and confirms successful initialization with INITO. Service requests can be made via the REQ event. The input data is processed via the IN adapter, and the results are provided via the corresponding outputs.

Technical Features

- Uses standardized initialization procedures according to EInit type
- Supports configuration via SA member (default: 21h)
- Provides error handling via STATUS output
- Implements qualified event processing

State Overview

The function block goes through the following states:

1. **Not Initialized**: Before INIT event
2. **Initialization**: During INIT processing
3. **Ready**: After successful initialization (INITO)
4. **Service Request**: During REQ processing

Application Scenarios

- Digital input signal processing in automation systems
- I/O communication with PKP-2200-compatible devices
- Integration into distributed control systems
- Boolean signal processing with quality monitoring

⚖️ Comparison with Similar Blocks

Compared to simple digital input blocks, PKP_2200_LI_IXA offers extended functionality through:

- Parameterizable SA Member Configuration
- Detailed status feedback
- Standardized initialization procedures
- Adapter-based interface

Conclusion

The PKP_2200_LI_IXA is a robust and flexible function block for processing digital input signals with advanced configuration and diagnostic capabilities. Its standardized interface enables easy integration into complex automation systems.

2.1.2 DataPanel

2.1.2.1 Status

DataPanel_AMP_FEEDBACK_SAE_J1939

Amperage Feedback of 1 DataPanel Node

Interface

Event inputs

Name	Comment	With
INIT	Service Initialization	QI, u8SAMember

Event Outputs

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

Input Vars

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

Output Vars

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status
Current_1A	UINT	Current in mA or 0.1A depending on module
Current_1B	UINT	
Current_2A	UINT	
Current_2B	UINT	
Current_3A	UINT	
Current_3B	UINT	
Current_4A	UINT	

Current_4B	UINT	
Current_5A	UINT	
Current_5B	UINT	
Current_6A	UINT	
Current_6B	UINT	
Current_7A	UINT	
Current_7B	UINT	
Current_8A	UINT	
Current_8B	UINT	

DataPanel_FREQ_SAE_J1939
Frequency/Counter Inputs 7A-8A Status

Interface

Event inputs

Name	Comment	With
INIT	Service Initialization	QI, u8SAMember

Event Outputs

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

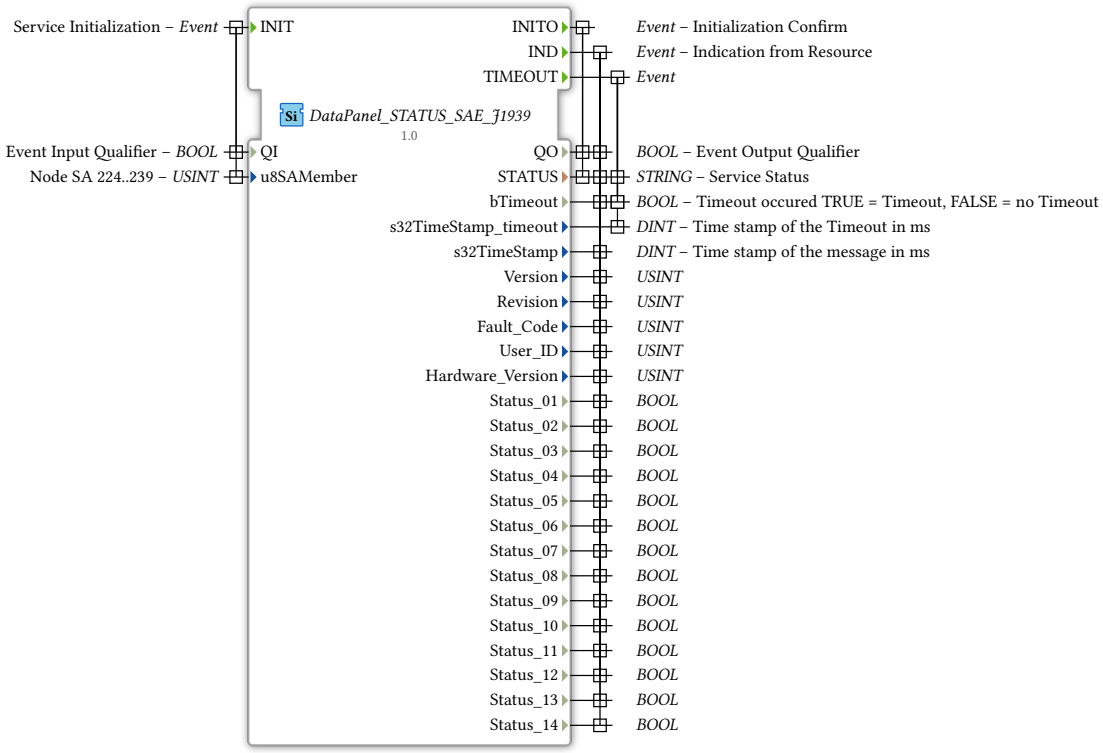
Input Vars

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

Output Vars

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status
Hertz_Count_7A	UINT	Frequency (Hz) or Count for Input 7A
DutyCycle_SP_7A	UINT	Duty Cycle or Setpoint for Input 7A
Hertz_Count_8A	UINT	Frequency (Hz) or Count for Input 8A
DutyCycle_SP_8A	UINT	Duty Cycle or Setpoint for Input 8A

DataPanel_STATUS_SAE_J1939



Status Output of 1 DataPanel Node

Interface

Event inputs

Name	Comment	With
INIT	Service Initialization	QI, u8SAMember

Event Outputs

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

Input Vars

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

Output Vars

Name	Type	Initial Value	Comment

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

2.1.2.2 🔌 io

📺 Video

- Automation of the Future
- The Great Migration
- Digital Decisions
- Ideal Diodes
- The Dual Wi-Fi Solution

🎧 Podcast

- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Target Groups, Structure, and Unique Selling Proposition
- Taming the Software Dragon: Industrial Automation and the Future of Production
- Eclipse 4diac: Innovation Engine Research vs. User Needs – What Really Drives Development?
- Taming the Software Dragon: Why IEC 61499 Could Revolutionize Industry

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Diode & Semiconductor Basics on ms-muc-docs.de
- 🌐 Smart Switch vs. Relay on ms-muc-docs.de

Modules with low IO density

DQ

DataPanel_LO_DO



• – ◦ • – ◦ • – ◦ •

Introduction

DataPanel_LO_DO is a global constant definition for low I/O density DataPanel modules for digital outputs (DO). The constant defines a structured mapping of pin numbers to specific digital output channels and provides an invalid state for error handling.

Interface Structure

Event Inputs

Not present - This is a global constant definition

Event Outputs

Not present - This is a global constant definition

Data Inputs

Not present - This is a global constant definition

Data Outputs

Not present - This is a global constant definition

Adapters

Not present - This is a global constant definition

Functionality

The global constant DataPanel_LO_DO defines a collection of structured constants of type `DataPanel_LO_DO_S`, representing digital output channels of a low-I/O-density DataPanel module. Each constant contains a pin assignment for specific output channels.

- **Data Type**: Uses the structured data type `DataPanel_LO_DO_S`
- **Pin Mapping**: Defines 8 digital output channels (1A-4B) with consecutive pin numbers 1-8
- **Error Handling**: Includes a special `Invalid` constant with pin value 255 for invalid states
- **Modular Structure**: Channel naming follows the pattern “DigitalOutput_XY” for clear identification

State Overview

Not applicable - This is a static constant definition without state machines

Application Scenarios

- Configuration of DataPanel modules with low I/O density
- Simplified pin mapping in control programs
- Type-safe use of digital output channels
- Error handling through defined invalid states

⚖️ Comparison with Similar Components

Compared to other I/O constant definitions, this offers DataPanel_LO_DO:

- Specific adaptation for modules with low I/O density
- Structured data types for improved type safety
- Clear naming convention for better readability
- Integrated error constant for more robust programming

Conclusion

DataPanel_LO_DO provides an efficient and type-safe solution for configuring digital output channels in DataPanel modules with low I/O density. Its structured definition and clear pin assignment enable maintenance-friendly and fault-resistant programming of control applications.

DataPanel_LO_DO_S

DataPanel Modules with low IO density DO Struct

Structure

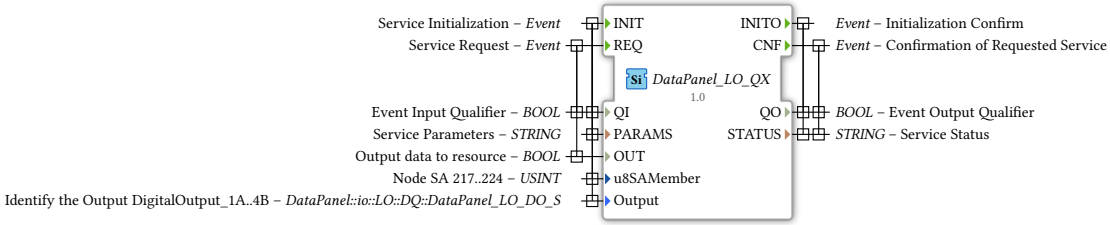
Name	Type	Initial Value
Pin	USINT	255

DataPanel_LO_QX

• – ◦ • – ◦ • – ◦ •

Introduction

The DataPanel_LO_QX is an output service interface function block for Boolean output data, specifically designed for communication with digital outputs in automation systems. This block enables the safe and controlled output of binary signals to connected hardware resources.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **OUT** (BOOL): Output Data for Resource
- **u8SAMember** (USINT): Node SA 217..224 (Initial Value: LO::LO_00)
- **Output** (DataPanel_LO_DO_S): Identifies Digital Output 1A..4B (Initial Value: Invalid)

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service status information

Adapter

No adapter interfaces are present.

Functionality

The function block operates according to a typical service interface pattern. During initialization (INIT event), the service parameters and hardware configuration are passed. The actual data output occurs via the REQ event, with the Boolean value at the OUT input being output to the configured hardware resource. Each operation is acknowledged by corresponding confirmation events (INITO, CNF).

Technical Features

- Supports specific hardware addressing via u8SAMember (SA nodes 217-224)
- Flexible output configuration via output parameters (DigitalOutput_1A to 4B)
- Initialization with default values for quick deployment
- String-based status feedback for detailed fault diagnosis

State Overview

The function block has two main states:

1. **Not Initialized**: Waiting for an INIT event
2. **Initialized**: Ready for REQ operations

After successful initialization, output requests can be processed continuously.

Application Scenarios

- Control of digital outputs in process automation
- Connection of binary signals to field devices
- Integration into distributed control systems
- Safety-critical switching operations

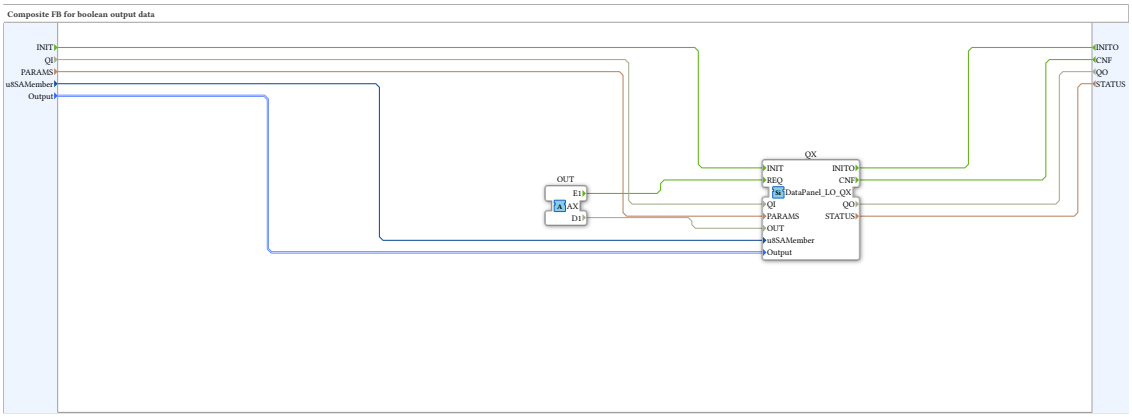
⚖️ Comparison with similar modules

Compared to simple digital output modules, DataPanel_LO_QX offers extended functionalities such as configurable hardware addressing, detailed status feedback, and a structured initialization procedure. Its specific SA member addressing makes it particularly suitable for complex distributed systems.

Conclusion

The DataPanel_LO_QX is a robust and flexible module for controlling digital outputs in industrial automation systems. Its structured interface and extensive configuration options make it particularly suitable for demanding applications with specific hardware requirements.

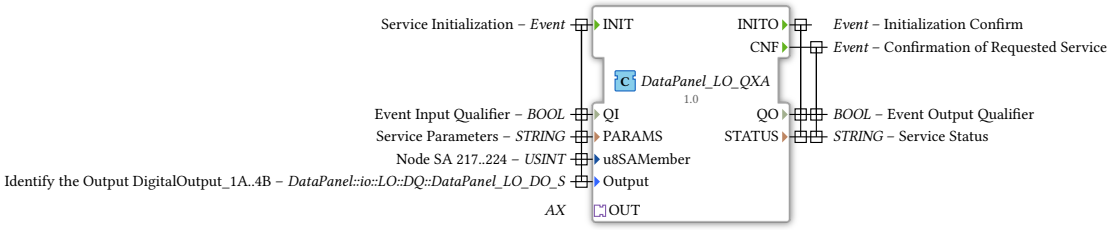
DataPanel_LO_QXA



• – ○ • – ○ • – ○ •

Introduction

The DataPanel_LO_QXA is a composite function block for outputting Boolean data in automation systems. It serves as an interface for digital output functions and enables the control of outputs via defined service parameters.



Interface Structure

Event Inputs

- **INIT**: Service initialization event with associated data QI, PARAMS, u8SAMember, and Output

Event Outputs

- **INITO**: Initialization acknowledgment with QO and STATUS
- **CNF**: Acknowledgement of the requested service with QO and STATUS

Data Inputs

- **QI** (BOOL): Event input qualifier
- **PARAMS** (STRING): Service parameter
- **u8SAMember** (USINT): Node SA 217..224 with initial value LO::LO_00
- **Output** (DataPanel_LO_DO_S): Identifies the output DigitalOutput_1A..4B with initial value Invalid

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service status

Adapter

- **OUT**: Unidirectional AX-type adapter for output control

Functionality

The function block initializes itself via the INIT event and returns the initialization status via INITO. The actual output control is handled by the OUT adapter, which is connected to the internal QX block. The QX block processes the input data and returns the status of the service request via CNF.

Technical Features

- Uses specific SA member addressing (217-224)
- Supports various digital outputs (1A-4B)
- Implements service parameter passing via STRING
- Includes integrated error handling with invalid initial value

State Overview

1. **Not Initialized**: Before INIT event
2. **Initialization in progress**: During INIT processing
3. **Ready**: After successful INITO confirmation
4. **Active**: During service requests via adapter OUT

Application Scenarios

- Control of digital outputs in automation systems
- Integration into process control systems with SA addressing
- Connection to higher-level control systems via service parameters
- Boolean signal output in industrial applications

⚖️ Comparison with Similar Blocks

Compared to simple digital output blocks, DataPanel_LO_QXA offers extended functionality through:

- Parameterizable Service Interface
- SA Member-Based Addressing
- Integrated Status Feedback
- Adapter-Based Extensibility

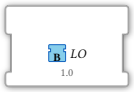
Conclusion

The DataPanel_LO_QXA is a powerful, integrated function block for the professional control of digital outputs in industrial automation systems. Its modular structure and configurable interface make it particularly suitable for complex applications with flexible configuration requirements.

const

LO

🎧 Podcast



- 4diac IDE: Your “Hello World” of Automation – The Blinking Tutorial Local
- Eclipse 4diac and FORTE: Unlocking Next-Gen Industrial Automation with Open-Source Power
- Field Devices: Eyes, Ears, and Hands of Industrial Automation – Why This Invisible Technology Is So Crucial
- Future Automation Unlocked: Powering “Plug & Produce” and Protecting Your Privacy
- Industrial Revolution Reloaded: Unpacking Plug & Produce, Data Privacy, and ETFA 2025

📺 Video

- The Catalog of 1863
- LogiBUS Project Update

Introduction

LO is a global constant definition for modules with low I/O density. These constants define specific address ranges for I/O modules and are used in the 4diac IDE to configure control systems.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

No adapters available

Functionality

LO consists of global constants defined as static values. The constants represent address ranges for low-density I/O modules and are integrated into the application at compile time.

Technical Features

- All constants are of type USINT (Unsigned Short Integer)
- Value range: 217 to 224
- Static initialization at compile time
- Constants are immutable during runtime

State Overview

No state transitions exist, as these are static constants

Application Scenarios

- Addressing low-density I/O modules
- Configuring hardware interfaces
- Use in conjunction with specific I/O drivers
- Simplifying hardware configuration through centralized constant definition

⚖️ Comparison with Similar Blocks

Compared to other global constant blocks, LO provides specific address ranges for low-density I/O modules, while other blocks may define different address ranges or constant types.

Conclusion

LO constants offer a structured and centralized way to define hardware addresses for low-density I/O modules. Using these global constants improves the maintainability and readability of control programs.

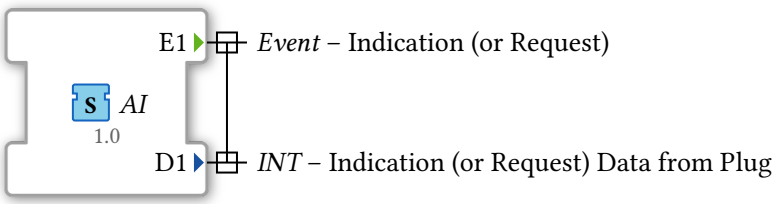
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Modules with medium IO density

AI



📺 Video

- AI: The Unknown Story
- The Silent Failure

🎧 Podcast

- From Clicking Relays to Intelligent Code: How Software Revolutionized Industrial Control
- ISOBUS Containers: Dynamic Control Panels for Clear Visibility and Greater Efficiency
- Analysis of the Amendment to the Master Craftsman Examination Regulations in the Agricultural and Construction Machinery Mechatronics Trade: A Detailed Comparison of the 2024 and 2001 Regulations
- The Relay in Detail: Switching Amplifiers, Protection, and the Secrets of A1/A2, 85/86, and the Hysteresis
- The fatal dilemma of the relay: Why silver contacts fail with small signals and gold melts under load – The burn-off effect explained

---

•

DataPanel_MI_AI

DataPanel Modules with medium IO density AI constants

Global Constants

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



DataPanel_MI_AI_S

DataPanel Modules with medium IO density AI Struct

Structure

Name	Type	Initial Value
Pin	USINT	255

DataPanel_MI_IW

Input service interface function block for analog input data

Interface

Event inputs

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

Event Outputs

Name	Comment	With

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

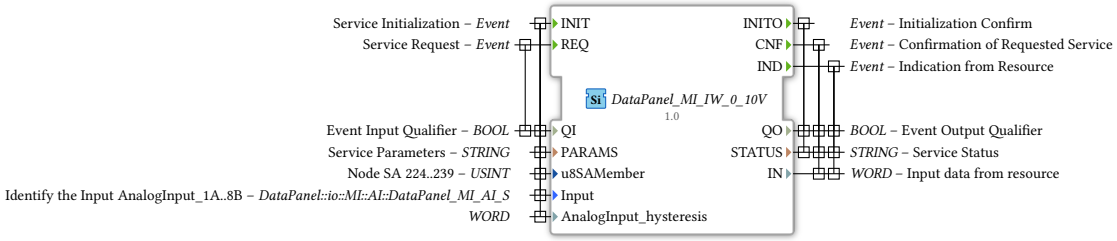
Input Vars

Name	Type	Initial Value	Comment
QI	BOOL		Event Input Qualifier
PARAMS	STRING		Service Parameters
u8SAMember	USINT	MI::MI_00	Node SA 224..239
Input	DataPanel::io::MI::AI::DataPanel_MI_AI_S	AI_1	Identify the Input AnalogInput_1A..8B

Output Vars

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

DataPanel_MI_IW_0_10V



• – ○ • – ○ • – ○ •

Introduction

The **DataPanel_MI_IW_0_10V** function block is a service-oriented interface module (SIFB) for acquiring an analog 0-10V input signal. It is designed as part of the **HR Agrartechnik DataPanel MI** system and enables parameterizable initialization as well as cyclical or event-driven retrieval of measured values.

Interface Structure

Event Inputs

Event	Type	Data Carried	Description
`INIT`	`EInit`	`QI`, `PARAMS`, `u8SAMember`, `Input`, `AnalogInput_hysteresis`	Service Initialization: Hardware Connection Configuration
`REQ`	`Event`	`QI`	Service Request: Triggering a Measurement Data Query

Event Outputs

Event	Type	Accompanying Data	Description
`INITO`	`EInit`	`QO`, `STATUS`	Initialization Confirmation

CNF	Event	QO, STATUS, IN	Measurement Query Confirmation
IND	Event	QO, STATUS, IN	Asynchronous display (e.g., spontaneous measurement update from the resource)

Data Inputs

Name	Type	Initial Value	Description
QI	BOOL	–	Input Qualifier (controls execution)
PARAMS	STRING	–	Service Parameter (e.g., communication configuration)
u8SAMember	USINT	MI::MI_00	Node SA Address (valid range 224…239)
Input	DataPanel::io::MI::AI::DataPanel_MI_AI_Slid	AI_1	Analog Input Selection (e.g., AnalogInput_1A … AnalogInput_8B)
AnalogInput_hysteresis	WORD	–	Hysteresis Value for Signal Smoothing

Data Outputs

Name	Type	Description
QO	BOOL	Output Qualifier (indicates valid processing)
STATUS`	STRING	Service Status (Error/Success Message)
IN`	WORD	Digitized Analog Value (0…10V, raw WORD value)

Adapter

No adapters available.

Functionality

1. **Initialization (INIT)**

The function block is configured with the parameters PARAMS, the node address u8SAMember, the selected input Input, and the hysteresis AnalogInput_hysteresis. Upon successful initialization, the output INITO is sent with QO = TRUE and a positive STATUS.

1. **Measurement Query (REQ)**

A new measurement is requested by creating an event at REQ. The result is available at output IN as soon as event CNF is triggered. Qualifiers QO and STATUS indicate the validity of the value.

1. **Asynchronous Display (IND)**

The function block can also receive an event IND from the hardware without an explicit request (e.g., during a spontaneous value change or a periodic update). In this case, IN, QO, and STATUS are also updated.

The digitized analog value is provided as a 16-bit word (WORD) on output IN. The scaling (e.g., 0V ⟶ 0, 10V ⟶ 65535) depends on the connected hardware and must be interpreted in the application.

Technical Features

- **HR Agricultural Technology Specific**: This function block is designed for the DataPanel MI family and uses predefined constants from the packages DataPanel::io::MI::const::MI and DataPanel::io::MI::AI::DataPanel_MI_AI.
- **Hysteresis**: The parameter AnalogInput_hysteresis allows you to configure noise reduction or debouncing of the signal.

- **Addressing**: The bus node is selected via `u8SAMember` (USINT, range 224–239); the default is `MI::MI_00`.
- **Compiler Integration**: The block is integrated into the 4diac IDE ecosystem as a SIFB and has a unique type hash (`eclipse4diac::core::TypeHash`).

State Overview

Since this is a service-oriented block, the internal state machine is controlled by the following events:

- **IDLE** – After successful initialization (or after a reset), the block waits for `REQ` or an external `IND`.
- **BUSY** – After a `REQ`, the hardware query is performed; no further `REQ` is possible during this time. Upon completion, `CNF` is sent, and the state returns to IDLE.
- **ERROR** – If an error occurs during initialization or measurement, `STATUS` is set accordingly, and `QO = FALSE` is signaled.

A detailed state machine (ECC) is not present in the XML model; however, the described behavior is typical for service-oriented SIFBs.

Application Scenarios

- **Agricultural Technology**: Acquisition of sensor signals (e.g., pressure, temperature, fill level) via 0-10V interfaces on HR data panels.
- **Industrial Control**: Integration of analog measured values into a PLC environment via 4diac applications.
- **Configurable Measuring Points**: Multiple channels of an MI module can be queried using different `Input` values.

Comparison with Similar Function Blocks

Function Block	Voltage Range	Platform	Features
`DataPanel_MI_IW_0_10V`	0-10V	HR DataPanel MI	Hysteresis, SA Addressing
`DataPanel_MI_IW_4_20mA`	4-20mA	HR DataPanel MI	Analog Current Inputs
`GenericAnalogInput`	Variable	Standard IEC 61131	General Input, No Hysteresis

This function block is specifically optimized for DataPanel-MI hardware and offers tight coupling to device-specific parameters.

Conclusion

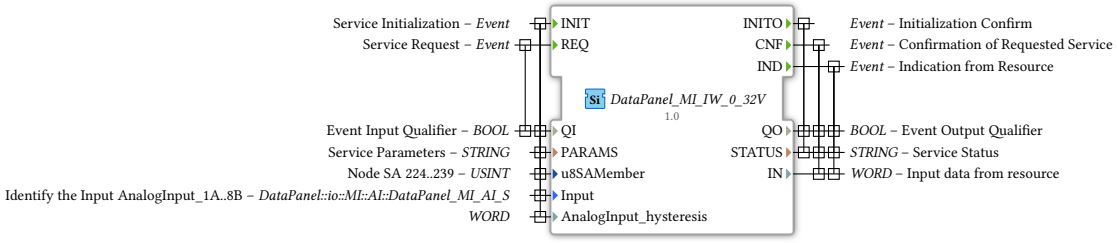
The **DataPanel_MI_IW_0_10V** function block provides a robust and flexibly configurable interface for acquiring analog 0-10V signals in the HR Agrartechnik DataPanel system. Its channel configuration, addressing, and hysteresis capabilities make it suitable for precise measurement tasks in agricultural automation.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

DataPanel_MI_IW_0_32V



• – ○ • – ○ • – ○ •

Introduction

The function block **DataPanel_MI_IW_0_32V** is a Service Interface Function Block (SIFB) according to IEC 61499. It is used to read an analog voltage signal in the range of 0–32V via a DataPanel MI module. The block handles the initialization of the communication channel, the configuration of an analog input (e.g., AnalogInput_1A to 8B), and the cyclical querying or reception of measured values. The output is as a **WORD** value; the status is signaled via **STATUS** and the **CNF** or **IND** event.

Interface Structure

Event Inputs

Event	Comment	With
INIT	Service Initialization	QI, PARAMS, u8SAMember, Input, AnalogInput_hysteresis
REQ	Service Request (Measurement Retrieval)	QI

Event Outputs

Event	Comment	With
INITO	Initialization Acknowledgement	QO, STATUS
CNF	Service Request Acknowledgement	QO, STATUS, IN
IND	Indication of a New Measured Value from the Resource	QO, STATUS, IN

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	–	Event input qualifier
PARAMS	STRING	–	Service parameters (e.g., addresses, baud rate)
u8SAMember	USINT	`MI::MI_00`	Node SA address (224..239)
Input	DataPanel::io::MI::AI::DataPanel_MI_AI_In	`Invalid`	Analog input selection (AnalogInput_1A ... 8B)
AnalogInput_hysteresis	WORD	–	Analog value hysteresis (configuration)

Data Outputs

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status (e.g., error/success messages)
IN	WORD	Measured Analog Value (0..32V scaled as WORD)

Adapters

No adapters available.

Functionality

The function block is started by the **INIT** event. The passed parameters (node address, analog input selection, and hysteresis) are then passed to the hardware. After successful initialization, **INITO** is triggered with `QO = TRUE`.

A **REQ** event triggers the query of the current analog value. The measured value is provided as a **WORD** via the **IN** output and acknowledged via the **CNF** event. The resource can also spontaneously

send a new value, which is then signaled via the **IND** event – also with the current **IN** and **STATUS** values.

STATUS returns a valid value for `"OK"`; in case of errors (e.g., communication error, invalid channel), an error message is issued and `QO` is set to `FALSE`.

Technical Specifications

- **Voltage Range:** 0V to 32V, represented as a WORD – scaling is hardware-dependent.
- **Hysteresis Configuration:** The parameter `AnalogInput_hysteresis` allows setting a threshold to suppress noise or small voltage fluctuations.
- **Initialization:** The input `u8SAMember` must contain a valid node address (224–239); `Input` selects the specific analog channel. The default settings are `MI::MI_00` (address of the first MI module) and `Invalid`, meaning a valid channel must be set before the first INIT.
- **Service Interface Block:** The FB communicates directly with the hardware and is not modeled by an ECC state machine – event control is event-driven, following the SIFB pattern.

State Overview

Since this is a Service Interface FB, the internal logic is handled by the hardware. From the perspective of the IEC 61499 model, the following typical phases emerge:

- **IDLE:** After system startup, before an INIT has been triggered.
- **INITIATE:** During INIT processing (hardware parameterization).
- **OPERATIONAL:** After successful INIT – REQ and IND events are processed.
- **ERROR:** In case of faulty initialization or loss of communication – STATUS contains an error message.

Application Scenarios

- **Voltage Monitoring:** Acquisition of sensor signals (e.g., pressure sensors, level sensors) with an output signal of 0…10V or 0…32V.
- **Analog Value Acquisition in Agricultural Technology:** Integration into a DataPanel control system for recording analog values from various machine components.
- **Hysteresis-controlled threshold evaluation:** Direct use of the input value in downstream logic that reacts to thresholds being exceeded or fallen below.

Comparison with similar components

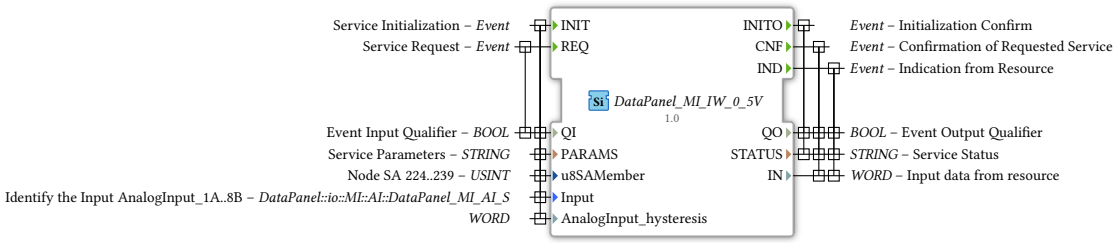
Unlike generic analog input components (e.g., `AI_BASIC`), the **DataPanel_MI_IW_0_32V** is specifically optimized for DataPanel MI hardware. Typical differences:

- **Parameterization:** Specific node address (`u8SAMember`) and channel selection (`Input`) instead of generic configuration strings.
- **Hysteresis:** Explicit input for hysteresis – with standard components, this is often controlled via the communication driver parameterization.
- **Voltage Range:** Fixed at 0–32V; other modules may offer adjustable measurement range.

Conclusion

The **DataPanel_MI_IW_0_32V** is a practical, custom-designed module for the DataPanel platform, enabling the acquisition of analog signals in the 0–32V range. Its clear interface (initialization, retrieval, spontaneous indication) and integrated hysteresis make it ideal for use in agricultural control systems and other industrial applications with DataPanel components. Easy operation via the INIT and REQ events, along with standardized status feedback, allows for rapid integration into IEC 61499 projects.

DataPanel_MI_IW_0_5V



• – ○ • – ○ • – ○ •

Introduction

The function block **DataPanel_MI_IW_0_5V** is a service interface function block (SIFB) for acquiring analog input signals in the voltage range **0 … 5 V**. It provides the interface to an analog input channel of the DataPanel-MI-IW series and enables initialization, cyclic querying, and asynchronous reception of measured values via a bus system. The block is implemented as an IEC 61499-compliant FB and uses system-specific data types from the package `DataPanel::io::MI::AI`.

Interface Structure

Event Inputs

Event	Type	Description	With Variables
`INIT`	EInit	Initializes the service connection	`QI`, `PARAMS`, `u8SAMember`, `Input`, `AnalogInput_hysteresis`
`REQ`	Event	Requests a current measurement value	`QI`

Event Outputs

Event	Type	Description	With Variables
`INITO`	EInit	Initialization Confirmation	`QO`, `STATUS`
`CNF`	Event	Confirmation of a Completed Request	`QO`, `STATUS`, `IN`
`IND`	Event	Asynchronous Indication of a Measured Value from the Resource	`QO`, `STATUS`, `IN`

Data Inputs

Variable	Type	Description	Initial Value
`QI`	BOOL	Qualifier for Event Inputs	–
`PARAMS`	STRING	Service Parameter (e.g., Communication Address)	–
`u8SAMember`	USINT	Node SA Address (224 … 239)	`MI::MI_00`
`Input`	DataPanel::io::MI::AI::DataPanel_MI_AI_Selection	Selection of the analog input channel (e.g., AnalogInput_1A … 8B)	`Invalid`
`AnalogInput_hysteresis`	WORD	Hysteresis value for signal evaluation	–

Data Outputs

Variable	Type	Description
`QO`	BOOL	Output qualifier (indicates valid status)
`STATUS`	STRING	Service status message (e.g., error text)
`IN`	WORD	Measured analog value (raw value) from the resource

Adapter

No adapter interfaces are available.

Functionality

The function block operates according to the **service interface principle**: It communicates with an underlying hardware resource (e.g., a bus system or I/O module) via events.

1. **Initialization (`INIT`)**

A `INIT` event (with `QI` set) initiates the connection. The parameters `PARAMS`, `u8SAMember`, `Input`, and `AnalogInput_hysteresis` define the target address and channel configuration. After successful initialization, `INITO` is triggered, and `QO` is set to `TRUE` and `STATUS` to the corresponding text.

1. **Measurement Request (`REQ`)**

Once the function block (FB) is initialized, a current measurement value can be requested via `REQ`. The function block (FB) sends the request to the resource and signals upon receiving the result with `CNF`. The raw value is provided in the output variable `IN` (type `WORD`).

1. **Asynchronous Indication (`IND`)**

The resource can independently report changes in measured values or alarms. These are reported via the `IND` event with the corresponding value in `IN`.

In case of a negative value (`QI`) or an error, `QO` is set to `FALSE` and `STATUS` to an error description.

Technical Features

- **System-Specific Types**

The function block uses the user-defined structures `DataPanel_MI_AI_S` (channel selection) and the constants `MI::MI_00` (address specification). These are defined in the package `DataPanel::io::MI::AI`.

- **Hysteresis**

A hysteresis value can be specified via the input `AnalogInput_hysteresis` as a `WORD` value to suppress the noise of the analog signal.

- **Channel Addressing**

The analog input channel is selected via the `Input` parameter. Valid values are, for example, `AnalogInput_1A` through `AnalogInput_8B`; the initial value `Invalid` must be replaced with a valid channel before first use.

State Overview

Since the XML definition does not contain an ECC (Execution Control Chart), the state logic is derived from the typical behavior of a SIFB (Single-Integrated Function Block). An abstract state machine can be described as follows:

State	Description	Event	Action
IDLE	Waiting for Initialization	`INIT` (QI=TRUE)	Starting connection
INIT	Initialization in progress	–	Waiting for resource confirmation
READY	Ready for requests	`INITO`	Set QO=TRUE
BUSY	Measurement request in progress	`REQ`	Sending request to resource
DONE	Response received	`CNF`	Loading `IN` and setting QO=TRUE
ERROR	Error status	–	Set QO=FALSE, STATUS=Error text

Asynchronous `IND` events can occur in the **READY** or **BUSY** states and result in the immediate availability of the value.

Application Scenarios

- **Agricultural Sensors** (e.g., level sensors, pressure sensors, temperature sensors with 0-5V output)
- **Data Acquisition in Stationary Motors** or **Vehicle Control Units** of the DataPanel family
- **Multi-Channel Analog Value Acquisition** through parallel instances of the FB with different `Input` and `u8SAMember` values

Comparison with Similar Function Blocks

Feature	DataPanel_MI_IW_0_5V	Generic Analog Input (e.g., IEC 61499 standard)
Voltage Range	0 – 5V	Usually configurable (0-10V, 4-20mA, etc.)
Channel Selection	Specific type `DataPanel_MI_AI_S`	Usually `INT` or `STRING` parameter
Hysteresis	Separate Variable (`WORD`)	Often not included

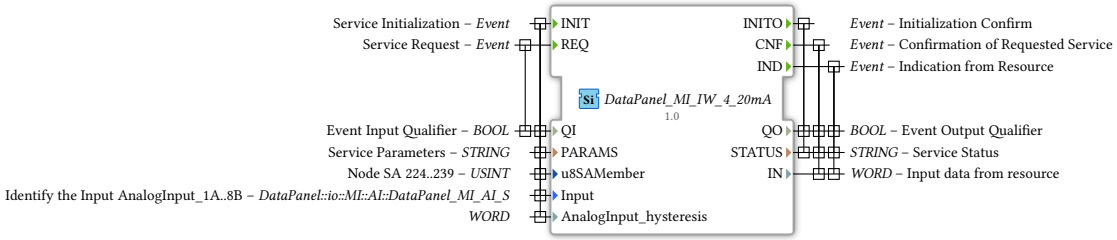
Bus connection	Proprietary (DataPanel-MI-IW)	Abstract communication interface

The function block is highly tailored to the DataPanel hardware and therefore offers less flexibility than a generic analog input, but provides a direct, optimized connection.

Conclusion

The **DataPanel_MI_IW_0_5V** is a specialized service interface function block for acquiring 0–5V analog signals in the DataPanel environment. It encapsulates the complex bus communication and offers a simple, event-driven interface to the application logic. Thanks to its integrated hysteresis and clear channel addressing, it is particularly suitable for robust measurement tasks in agricultural automation.

DataPanel_MI_IW_4_20mA



• – ○ • – ○ • – ○ •

Introduction

The function block **DataPanel_MI_IW_4_20mA** is a service interface function block (SIFB) for acquiring analog input data in the 4-20mA range. It serves as an interface between the automation system and a hardware-level analog input module. The block handles the initialization of communication (e.g., bus connection), the cyclic request for measured values, and the provision of the measured raw data via output events.

• – ○ • – ○ • – ○ •

Interface Structure

Event Inputs

Event	Type	Description	Associated Variables
INIT	EInit	Service Block Initialization	QI, PARAMS, u8SAMember, Input, AnalogInput_hysteresis
REQ	Event	Request for a Measured Value Update	QI

Event Outputs

Event	Type	Description	Carried Variables
INITO	EInit	Confirmation of Successful Initialization	QO, STATUS
CNF	Event	Confirmation of a Requested Measurement	QO, STATUS, IN
IND	Event	Asynchronous Display of an Incoming Measured Value	QO, STATUS, IN

Data Inputs

Variable	Type	Description	Initial Value
QI	BOOL	Qualifier for the Event Input	–
PARAMS	STRING	Parameter for Service Initialization	–

u8SAMember	USINT	Node Address (224...239) of the Slave Device	MI::MI_00 (224)
Input	*DataPanel::io::MI::AI::DataPanel_MI_AI_S*	Analog input channel identification (e.g., AnalogInput_1A..8B)	Invalid
AnalogInput_hysteresis	WORD	Hysteresis value for signal smoothing	–

Data Outputs

Variable	Type	Description
QO	BOOL	Qualifier for event output
STATUS	STRING	Status message (e.g., error, initialization)
IN	WORD	Raw value read from analog input

Adapter

No adapters defined.

• – ◦ • – ◦ • – ◦ •

Functionality

This function block implements asynchronous communication with a slave device (e.g., an analog input module with a 4-20mA interface) via a proprietary bus.

1. **Initialization** (INIT):

The parameters PARAMS (e.g., baud rate, protocol settings), the node address (u8SAMember), and the specific analog input channel (Input) are set. Hysteresis (AnalogInput_hysteresis) can be specified to stabilize the raw value. After a successful connection, INITO is sent along with QO = TRUE.

1. **Measurement Request** (REQ):

This function block requests the current measured value of the configured channel. The response is delivered asynchronously via output CNF (upon successful request) or, if necessary, via IND (in case of spontaneous value changes or cyclical messages from the slave). The read value appears in the output variable IN as a 16-bit raw value.

1. **Asynchronous Indication** (IND):

If the slave device sends data independently (e.g., upon exceeding a threshold), IND is triggered. This allows even unsolicited measurement values to be acquired.

The outputs QO and STATUS provide information about the success of the operations (e.g., initialization error, communication error).

• – ◦ • – ◦ • – ◦ •

Technical Specifications

- **Target Platform**: The module is designed for the "DataPanel" system of HR Agrartechnik GmbH (version 1.0, year 2026).
- **Node Address Range**: The slave addresses u8SAMember are limited to the range 224–239; The initial value MI::MI_00 corresponds to the smallest address (224).
- **Input Channel Identification**: The data type DataPanel_MI_AI_S defines possible channels (AnalogInput_1A … 8B). The initial value Invalid indicates that no channel is selected before the first initialization.
- **Hysteresis**: The AnalogInput_hysteresis value is passed as a 16-bit word and acts as a digital filter to prevent noise or oscillation.
- **Type Hash**: An attribute eclipse4diac::core::TypeHash is included but initialized with an empty string – it can be set later at runtime.

• – ◦ • – ◦ • – ◦ •

State Overview

Since this is a service interface function block (FB), its behavior is determined by the event logic and the underlying communication protocol. An internal state machine is not represented in the XML definition; however, the following phases typically exist:

- **OFF / UNINITIALIZED**: Before the first `INIT` event. No communication is active.
- **INIT** Phase: After receiving `INIT`, until `INITO` is sent. Slave configuration.
- **IDLE** (ready): After successful initialization. The FB can accept `REQ` events.
- **BUSY** (Request in progress): From `REQ` until the response arrives (followed by `CNF`).
- **INDICATION** state: `IND` is triggered when data arrives asynchronously, after which the function block returns to the IDLE state.

Error states (e.g., communication failure) are reported via `STATUS`.

• – ◦ • – ◦ • – ◦ •

Application Scenarios

- **4-20 mA Sensor Connection**: A pressure sensor, level sensor, or temperature sensor with a 4-20 mA output is connected to the DataPanel system. The function block reads the measured value cyclically or on demand.
- **Multi-channel acquisition**: Via different instances of the building block (with different `InputWith` parameters, multiple analog channels can be operated in parallel.
- **Data Logger System**: Combined with a function block for data recording, the `IND` output can be used to log status changes or alarms.
- **Agricultural Control**: In agricultural technology (HR Agrartechnik GmbH), this allows for the recording of fluid levels, flow rates, or axle loads, for example.

• – ◦ • – ◦ • – ◦ •

Comparison with Similar Function Blocks

•

Compared to a general analog input function block (e.g., a function block for standardized fieldbuses like PROFIBUS or IO-Link), this block is distinguished by the following features:

- **Specific Hardware Assignment**: It is directly tailored to the "DataPanel::io::MI::AI" protocol.
- **Node Addressing**: The limitation to nodes 224–239 and the predefined constants from `MI::MI_xx` simplify configuration in fixed networks.
- **Event Model**: `IND` alongside `CNF` allows for both synchronous and asynchronous data acquisition – more flexible than a pure polling model.
- **Event Model**: `IND` alongside `CNF` enables both synchronous and asynchronous data acquisition – more flexible than a pure polling model.
- **Node Addressing**: - **Hysteresis Parameter**: Provides simple debouncing at the function block level, a feature not found in many generic function blocks.

• – ◦ • – ◦ • – ◦ •

Conclusion

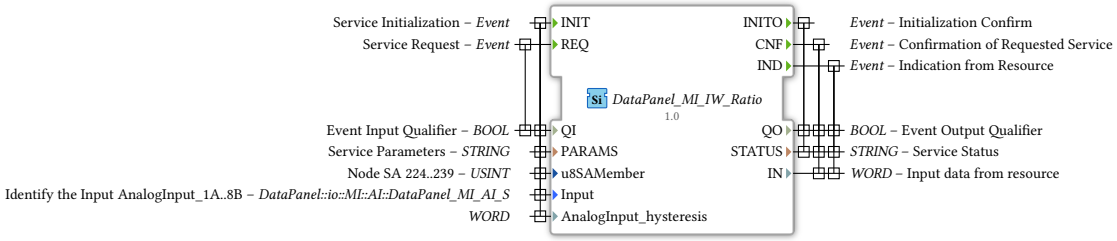
The **DataPanel_MI_IW_4_20mA** function block offers a robust and compact solution for acquiring 4-20mA signals in a proprietary DataPanel system. The clear separation of initialization, request, and spontaneous indication, along with the integrated hysteresis, makes it suitable for practical use in agricultural technology. Its configurable node address and channel allow for flexible deployment and integration into existing automation networks.

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

DataPanel_MI_IW_Ratio



• — ○ • — ○ • — ○ •

Introduction

The function block **DataPanel_MI_IW_Ratio** is a service interface function block (SIFB) for acquiring analog input data with ratiometric conversion. It provides the interface to a sensor connected to a node (SA 224..239) whose measured value is output as a 16-bit word (WORD). This block is part of the library `DataPanel::io::MI::AI` and is typically used in automation technology, particularly in agricultural engineering.

Interface Structure

Event Inputs

Event	Type	With Variables	Comment
INIT	EInit	QI, PARAMS, u8SAMember, Input, AnalogInput_hysteresis	Service Initialization
REQ	Event	QI	Service Request

Event Outputs

Event	Type	With Variables	Comment
INITO	EInit	QO, STATUS	Initialization Confirmation
CNF	Event	QO, STATUS, IN	Confirmation of Requested Service
IND	Event	QO, STATUS, IN	Indication from Resource

Data Inputs

Variable	Type	Initial Value	Comment
QI	BOOL	–	Event Input Qualifier
PARAMS	STRING	–	Service Parameters
u8SAMember	USINT	MI::MI_00	Node SA 224..239
Input	DataPanel::io::MI::AI::DataPanel_MI_AI_S	Invalid	Identify the Input AnalogInput_1A..8B
AnalogInput_hysteresis	WORD	–	(no value specified)

Data Outputs

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

Adapter

No adapters available.

Functionality

The function block operates as a state-controlled service interface block:

1. **Initialization (INIT)**:

The function block is configured via the INIT event. The inputs `PARAMS`, `u8SAMember`, `Input`, and `AnalogInput_hysteresis` define the bus/node address, the analog channel, and the hysteresis. The `QI` input must be set to TRUE to start initialization. After successful configuration, the INITO event is output with `QO = TRUE` and `STATUS = "OK"` (or similar).

1. **Measurement Request (REQ)**:

The REQ event requests a new measurement from the sensor. The function block evaluates the hardware and returns the result via the CNF event at output `IN`. Here too, `QO` indicates the validity of the value.

1. **Indication (IND)**:

If the hardware supports asynchronous events (e.g., cyclic updates), the IND event is used to report the current measurement value without prompting.

The measurement value is output as a 16-bit word (WORD) in ratiometric format. This means that the digital value is directly proportional to the ratio of the measured voltage to the reference voltage.

Technical Features

- **Ratiometric Measurement**: The function block is specifically designed for ratiometric sensors where the output is proportional to the supply voltage. This minimizes measurement errors caused by voltage fluctuations.
- **Configuration via constants**: The initial value of `u8SAMember` (`MI::MI_00`) and the data type `DataPanel_MI_AI_S` are derived from imported libraries (`DataPanel::io::MI::const::MI` and `DataPanel::io::MI::AI::DataPanel_MI_AI`). The valid address range for the node is between 224 and 239 (Node SA 224..239).
- **Hysteresis**: Threshold hysteresis behavior can be implemented using `AnalogInput_hysteresis`; the exact effect depends on the underlying driver.
- **TypeHash attribute**: The attribute `eclipse4diac::core::TypeHash` is used for unique type identification in the runtime system.

State Overview

Since the function block is implemented as a service interface function block, it has an internal state machine. The typical states are:

- **IDLE**: Waiting for an INIT event.
- **INIT**: Initialization is being executed (parameterization, hardware access).
- **READY**: Initialization is complete, waiting for REQ or IND.
- **REQ**: Processing a measurement request.
- **IND**: Asynchronous indication is being processed.

In case of an error, `QO = FALSE` is set and `STATUS` contains a corresponding error message.

Application Scenarios

- **Agricultural Machinery**: Acquisition of sensor values (e.g., fill levels, pressure, position) via the ratiometric input of a data panel.
- **Industrial Automation**: Connection of analog sensors with voltage output (0…5V, 0…10V) that operate ratiometrically.
- **Early Prototypes**: The module can be directly integrated into a controller from the 4diac IDE and tested with any parameters.

Comparison with Similar Modules

Module	Feature
`DataPanel_MI_AI`	Standard analog input without explicit ratiometric design.
`DataPanel_MI_IW_Voltage`	Voltage measurement with absolute values (e.g., mV).
DataPanel_MI_IW_Ratio	Specifically optimized for ratiometric sensors.

The module described here differs primarily in its use of the ratiometric measurement principle, which is employed in many modern sensors (e.g., Hall effect sensors, potentiometers).

Conclusion

The `DataPanel_MI_IW_Ratio` is a specialized service interface function block for the reliable acquisition of ratiometric analog signals. With its clearly defined interface (INIT/REQ/IND) and configurable parameters, it is ideally suited for use in agricultural and automation applications. Integration into the 4diac IDE is achieved via the `DataPanel::io::MI::AI` library.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

DI

📺 Video

- Decoding the datasheet
- The Great Migration
- The Art of Soldering
- The World of Standardization
- Digital Decisions

🎧 Podcast

- 4diac IDE: Your “Hello World” of Automation – The Blinking Tutorial
- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- 4diac IDE: How the IEC 61499 Standard is Shaping Industrial Automation Revolutionized
- 4diac presentation: Target groups, structure, and unique selling proposition Sharpening
- Taming the software dragon: Industrial automation and the future of production

---

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

DataPanel_MI_DI



• – ◦ • – ◦ • – ◦ •

Introduction

DataPanel_MI_DI is a global constant definition for medium I/O density DataPanel modules for digital inputs (DI). This constant defines the pin assignments for various digital input channels of a DataPanel system.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

No adapters available

Functionality

DataPanel_MI_DI is a global constant definition that provides fixed values for the pin assignments of the digital inputs. The constant defines structured data of type `DataPanel_MI_DI_S` with PIN numbers for various digital input channels.

Technical Features

- Defines pin assignments for 8 digital input channels (5A, 5B, 6A, 6B, 7A, 7B, 8A, 8B)
- Pin numbers range from 9 to 16
- Includes a special "Invalid" constant with a pin value of 255 for error handling
- Uses the structured data type `DataPanel_MI_DI_S`
- Organized in the package `DataPanel::io::MI::DI`

State Overview

No state transitions are present - this is a static constant definition

Application Scenarios

- Configuration of DataPanel systems with medium I/O density
- Hardware abstraction for digital inputs in control systems
- Simplification of pin assignment in IEC 61499-based applications
- Reusable constant definition for multiple projects

⚖️ Comparison with Similar Components

Compared to other constant definitions, DataPanel_MI_DI offers:

- Specific pin assignments for DataPanel hardware
- Structured data types for improved type safety
- Predefined invalid constant for robustness

Conclusion

DataPanel_MI_DI provides a standardized and type-safe way to configure digital input pins in DataPanel systems with medium I/O density. The use of structured constants increases maintainability and reduces configuration errors in automation projects.

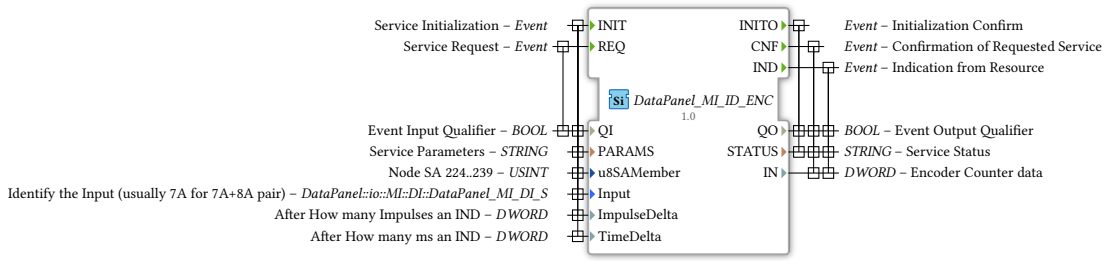
DataPanel_MI_DI_S

DataPanel Modules with medium IO density DI Struct

Structure

Name	Type	Initial Value
Pin	USINT	255

DataPanel_MI_ID_ENC



• – ○ • – ○ • – ○ •

Introduction

The **DataPanel_MI_ID_ENC** function block is an input service interface function block for acquiring encoder pulse data. It is designed to process a 7A+8A encoder pair and outputs the current counter reading and status information. Initialization is performed via parameters such as the node address (SA member), the input configuration, and thresholds for pulse- and time-based event triggering.

Interface Structure

Event Inputs

Event	Description	Accompanying Data
INIT	Service Initialization	QI, PARAMS, u8SAMember, Input, ImpulseDelta, TimeDelta
REQ	Service Request	QI

Event Outputs

Event	Description	Accompanying Data
INITO	Initialization Acknowledgement	QO, STATUS
CNF	Requested Action Acknowledgement	QO, STATUS, IN
IND	Asynchronous Event Display (Pulse or Timeout)	QO, STATUS, IN

Data Inputs

Name	Type	Initial Value	Description
QI	BOOL	–	Event Input Qualifier
PARAMS	STRING	–	Service Parameter
u8SAMember	USINT	MI::MI_00	Node Address (range 224..239)
Input	DataPanel::io::MI::DI::DataPanel_MI_DI_S	Invalid	Input Identifier (usually 7A for a 7A+8A pair)
PulseDelta	DWORD	–	Number of pulses after which an IND is triggered
TimeDelta	DWORD	–	Time in milliseconds after which an IND is triggered

Data Outputs

Name	Type	Description
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status
IN	DWORD	Current Encoder Counter Value

Adapter

None.

Functionality

1. **Initialization (INIT → INITO)**

The function block is configured with the parameters `PARAMS`, the node address `u8SAMember`, the input specification `Input`, and the threshold values `ImpulseDelta` and `TimeDelta`. After successful initialization, the event `INITO` is acknowledged with `QO` and `STATUS`.

1. **Request (REQ → CNF)**

The function block `REQ` triggers a targeted query of the current counter reading. The block responds with `CNF` and provides the current encoder value via `IN`, along with status information.

1. **Asynchronous Events (IND)**

Regardless of an explicit request, `IND` is triggered as soon as either the number of encoder pulses specified in `ImpulseDelta` has been exceeded or the time interval defined in `TimeDelta` has expired. This enables event-driven processing without continuous polling.

The output `IN` contains the current 32-bit counter value of the encoder for each event (CNF and IND).

Technical Features

- **User-Defined Types**: The input `Input` is based on the data type `DataPanel_MI_DI_S`, which expects a specific input configuration (e.g., 7A). The constant value `Invalid` serves as a placeholder for undefined inputs.
- **Configurable Event Triggering**: By combining `ImpulseDelta` and `TimeDelta`, the system can trigger an IND either after a specific number of pulses, after a time interval, or after the first event occurring (AND operation).
- **Node Addressing**: The physical node in the bus system is selected via `u8SAMember` (value range 224–239). The initial value `MI::MI_00` refers to a constant in the module `MI`.

State Overview

State	Description
IDLE	Waiting for INIT or REQ
INIT	Initialization in progress, parameters are being applied
ACTIVE	Initialization complete, ready for REQ and IND
ERROR	Error state (e.g., failed initialization)

The actual state machine is not explicitly represented in the code shown; the states depicted are derived from the typical behavior of service interface function blocks.

Application Scenarios

- **Agricultural Machinery Control**: Acquisition of rotational speeds on drive shafts via incremental encoders (7A+8A pair) for monitoring and controlling work processes.
- **Position Acquisition**: Use as a pulse counter for position measurement, e.g., for drive systems or actuators.
- **Event-Driven Data Acquisition**: Configurable IND thresholds prevent peak loads in data traffic and optimize logging tasks.

Comparison with Similar Components

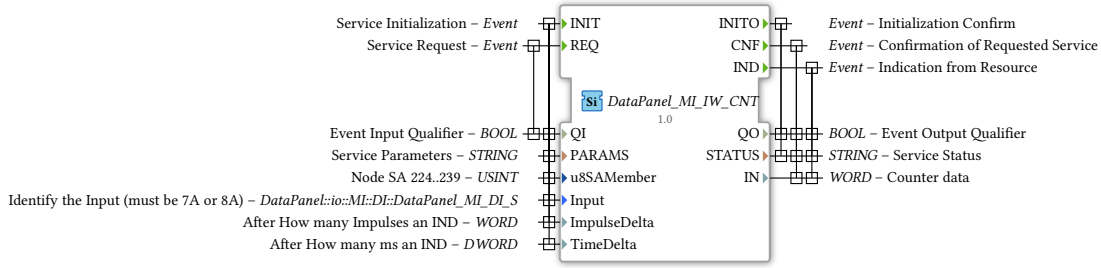
Compared to simple encoder counters, the DataPanel_MI_ID_ENC offers:

- **Flexible Trigger Criteria**: Instead of relying solely on polling, pulse or time thresholds can be set.
- **Structured Initialization**: The use of a user-defined type (`DataPanel_MI_DI_S`) enables clear mapping to specific hardware inputs.
- **Event-based output**: The separation of `CNF` (synchronous with the request) and `IND` (asynchronous) allows for decoupled processing at higher control levels.

Conclusion

The **DataPanel_MI_ID_ENC** function block is a powerful service interface module for encoder data acquisition in industrial control systems. Its configurable thresholds and support for both synchronous and asynchronous events make it particularly suitable for real-time applications in agricultural engineering. The clear interface definition and the use of specific data types enable easy integration into existing automation environments.

DataPanel_MI_IW_CNT



• – ○ • – ○ • – ○ •

The **DataPanel_MI_IW_CNT** is a service interface function block from the DataPanel family of HR Agrartechnik GmbH. It is used to acquire **pulse counter data** via the dedicated hardware inputs 7A and 8A of the underlying bus system. The block encapsulates the initialization, cyclic querying, and event-driven output of the counter values. Typical applications include agricultural machinery where pulse generators (e.g., speed, flow, or position sensors) need to be evaluated. The function block is designed for the IEC 61499-compliant 4diac IDE and uses the service interface patterns defined therein.

Event	Type	Description
INIT	EInit	Initializes the channel (sets hardware parameters)
REQ	Event	Requests a current counter value
Event	Type	Description
---	---	----------
INITO	EInit	Acknowledges successful initialization
CNF	Event	Confirmation of a requested REQ operation
IND	Event	Asynchronous indication (triggered by pulse or time interval)
Name	Type	Description
------	-----	----------
QI	BOOL	Qualifies the INIT/REQ event
PARAMS	STRING	Service parameter (vendor-dependent configuration)
u8SAMember	USINT	Node SA (224..239); Default = `MI::MI_00`
Input	DataPanel_MI_DI_S	Input identification (must be “7A” or “8A”); Default = `Invalid`
ImpulseDelta	WORD	Pulse threshold for asynchronous IND triggering (number of pulses)
TimeDelta	DWORD	Time threshold for asynchronous IND triggering (in ms)
Name	Type	Description
----	----	----------
QO	BOOL	Qualifies the event outputs
STATUS	STRING	Status message (e.g., “OK” or error code)
IN	WORD	Current 16-bit counter value (pulse counter)

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters
No adapter interfaces defined.

Functionality
The **DataPanel_MI_IW_CNT** operates as a service interface function block and communicates directly with the hardware interface of the DataPanel system. After startup, the channel must be initialized via the **INIT** event. The parameters (SA address, input number, pulse and time configuration) are passed during this process. The function block attempts to allocate and provide the corresponding hardware channel.

After successful initialization (signaled by **INITO**), the counter value can be queried via the **REQ** event. The response is provided via **CNF**: `IN` contains the current counter value, and `STATUS` contains the operating status.

Additionally, the function block can asynchronously output an **IND** indication when either the set number of pulses (`ImpulseDelta`) has been reached or the time interval (`TimeDelta`) has elapsed. This allows for event-driven processing without cyclical queries.

Error handling is provided via the `STATUS` output and the `QO` flag. If an error occurs (e.g., invalid input configuration, hardware unreachable), `QO` is set to `FALSE`, and a corresponding error message is output in `STATUS`.

- **Input Identification:** The parameter `Input` must be set to a valid DataPanel input type (7A or 8A). The default value `Invalid` prevents incorrect initialization.
- **Node SA (`u8SAMember`):** Defines the slave address of the DataPanel participant. Valid values are 224 to 239. The predefined value `MI::MI_00` represents the first slave.
- **Asynchronous Indication:** The function block can independently generate **IND** events using `ImpulseDelta` and `TimeDelta`. Both thresholds can be active independently. If either value exceeds the configured threshold, an IND event is triggered. This reduces the bus load compared to cyclic polling.
- **Data Format:** The counter value `IN` is a 16-bit word and allows values from 0 to 65535. The counter is reset to 0 upon overflow.
- **Copyright & Version:** This function block is published under version 1.0 for the year 2026 by HR Agrartechnik GmbH.

The FB (Function Block) cycles through classic service interface states:

State	Description
IDLE	Waiting for INIT or REQ. Hardware is not yet assigned.
INIT	INIT received – Parameterization and hardware reservation are in progress.
ACTIVE	Initialization successful – Channel is ready for operation.
REQUEST	REQ received – Querying the current counter value.
INDICATE	Asynchronous condition met – Sending an IND event.
ERROR	Error occurred (e.g., incorrect parameter, hardware error).

After an error, only a new INIT can return the FB to the IDLE state.

1. **Speed Measurement:** A radar sensor or magnetic pulse generator delivers rectangular pulses. The module counts the pulses and outputs the value as CNF or IND. `TimeDelta` can be used for time-based speed calculation.
2. **Flow Measurement:** Pulse flow meters (e.g., Hall effect) are used in agricultural irrigation systems. `ImpulseDelta`Generates an event after each defined quantity of liquid.

3. **Position Detection:** Distances are measured using an incremental encoder. By combining pulse and time monitoring, both position and speed can be determined.

- **DataPanel_MI_DI:** A purely digital input function block (status 0/1) without a counting function. The IW_CNT extends it with pulse counting and asynchronous event triggering.
- **DataPanel_MI_AI:** An analog input function block for voltage or current signals. In contrast, the IW_CNT processes only pulses (discrete) and not a continuous value.
- **CTUD** (IEC 61499 standard counter): This standard function block can count forwards/backwards but does not require low-level initialization. The DataPanel_MI_IW_CNT is specifically connected to the DataPanel hardware and offers the IND function, which is often missing in standard function blocks.

``## Conclusion
The **DataPanel_MI_IW_CNT** is a powerful service interface function block for the flexible acquisition of pulse counter data in agricultural control systems. Its integrated asynchronous indication based on pulse or time thresholds reduces response time and relieves the load on the control system. Thanks to its clearly structured interface and specific configuration options (SA address, input selection), it can be seamlessly integrated into the DataPanel ecosystem. It thus represents a reliable component for applications that require precise pulse evaluation at the system level.

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

Technical Features

State Overview

Application Scenarios

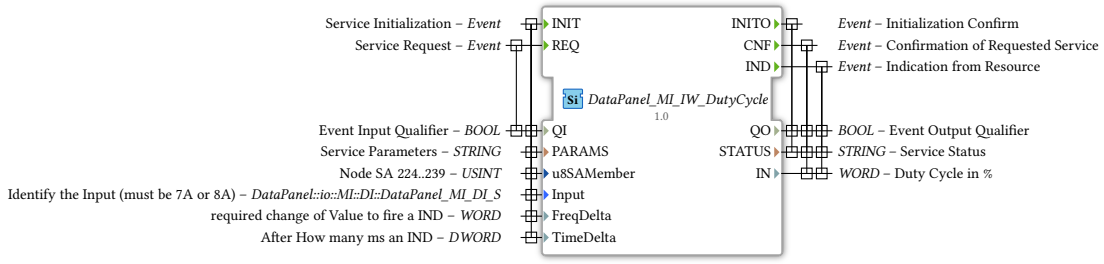
Comparison with Similar Function Blocks

Conclusion

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

DataPanel_MI_IW_DutyCycle

No image available.



• — ○ • — ○ • — ○ •

Introduction

The function block `DataPanel_MI_IW_DutyCycle` is a service interface function block (FB) for acquiring and processing frequency input signals (type 7A/8A) in the hardware system. It calculates the duty cycle from the incoming frequency signal and outputs it as a percentage. The block supports initial parameterization as well as event-driven outputs upon significant value changes or time-controlled queries.

Interface Structure

Event Inputs

Event	Type	With Variables	Comment
`INIT`	EInit	QI, PARAMS, u8SAMember, Input, FreqDelta, TimeDelta	Service Initialization

REQ	Event	QI	Service Request

Event Outputs

Event	Type	With Variables	Comment
INITO	EInit	QO, STATUS	Initialization Acknowledgement
CNF	Event	QO, STATUS, IN	Requested Action Acknowledgement
IND	Event	QO, STATUS, IN	Resource indication (upon value change/timer)

Data Inputs

Variable	Type	Comment
QI	BOOL	Event input qualifier
PARAMS	STRING	Service parameter (e.g., configuration string)
u8SAMember	USINT	Node address (SA 224..239, default: MI::MI_00)
Input	DataPanel_MI_DI_S	Input identification (must be 7A or 8A, initial value: Invalid)
FreqDelta	WORD	Required value change in %, to trigger an IND
TimeDelta	DWORD	Time interval in ms after which an IND is triggered

Data Outputs

Variable	Type	Comment
QO	BOOL	Event output qualifier
STATUS	STRING	Service status (failure/success message)
IN	WORD	Measured duty cycle in percent (0..100)

Adapter

No adapters defined.

Functionality

The function block initializes itself via the event INIT. The following parameters are passed: QI (activation), PARAMS (general configuration), the node address u8SAMember, the physical input (Input), the change delta FreqDelta, and the time interval TimeDelta. After successful initialization, INITO is output with valid QO and STATUS.

A REQ event triggers an immediate query of the current duty cycle. The result is output via CNF, with the current value displayed in IN.

The function block continuously monitors the input. If the measured duty cycle changes by more than the amount specified in FreqDelta, a IND event is sent with the new value. Additionally, after TimeDelta milliseconds (provided no change has occurred in the meantime), a IND event is also triggered to ensure regular updates.

Technical Specifications

- The function block is designed for connection to 7A or 8A type frequency inputs. The input is identified via the structure DataPanel_MI_DI_S – set to Invalid by default until a valid initialization occurs.
- The node address (SA) is limited to the range 224–239. The default is MI::MI_00.
- The parameters FreqDelta and TimeDelta allow for flexible configuration of the notification behavior: either change-only triggering, time-only triggering, or a combination of both.
- The status QO indicates the validity of the output data; STATUS contains detailed error or success messages.

- – Compiler imports reference the packages `DataPanel::io::MI::const::MI` and `DataPanel::io::MI::DI::DataPanel_MI_DI::Invalid`, as well as `eclipse4diac::core::TypeHash`.

State Overview

An explicit state machine is not defined in the XML. However, the following sequence can be derived from the behavior of the events:

1. **Initialization State** – After `INIT`, the module is configured and transitions to the operating state. Errors result in `INITO` with a negative `QO`.
2. **Operating State** – `REQ` is followed by `CNF` with the current duty cycle. When changes occur or the timer expires, `IND` is sent.
3. **Error State** – In case of invalid parameters or hardware errors, `STATUS` is set accordingly, and `QO` is changed to `FALSE`.

Application Scenarios

- **Speed Measurement in Agricultural Machinery** – Monitoring of frequency signals (e.g., from speed sensors) and output of the duty cycle as an indicator of engine speed or conveying rate.
- **Pulse Width Modulation (PWM) Analysis** – Measurement of the duty cycle of a PWM signal for controlling actuators or for providing feedback on the fill level.
- **Vibration/Frequency Monitoring** – Acquisition of periodic signals with configurable sensitivity (FreqDelta) and update rate (TimeDelta).

Comparison with Similar Components

- **`DataPanel_MI_DI`**: A digital input component without frequency or duty cycle calculation. Provides only binary states.
- **`DataPanel_MI_AI`**: Analog input component for voltage or current signals; not optimized for frequency signals.
- **`DataPanel_MI_IW_Frequency`**: Measures the absolute frequency, not the duty cycle. `DataPanel_MI_IW_DutyCycle` adds the percentage on/off time to this.

Conclusion

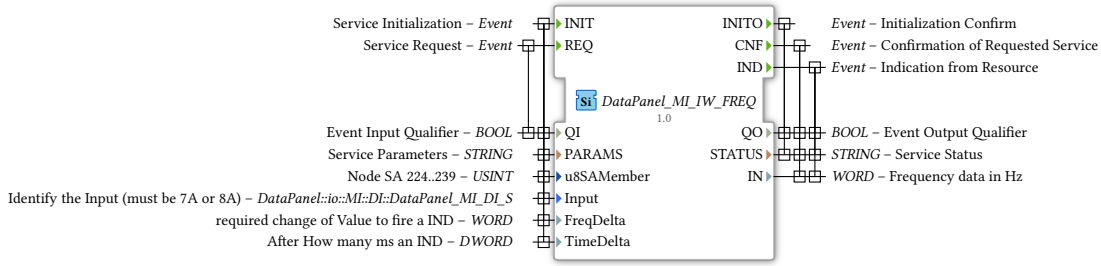
The `DataPanel_MI_IW_DutyCycle` function block offers a robust and configurable interface for acquiring duty cycles from frequency inputs in the 7A/8A range. The parameters `FreqDelta` and `TimeDelta` allow the reporting behavior to be flexibly adapted to the application. It is particularly suitable for use in agricultural technology and industrial control systems where reliable duty cycle monitoring is required.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 The PWM Signal & Infographic on ms-muc-docs.de

]

DataPanel_MI_IW_FREQ



No image available.

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **DataPanel_MI_IW_FREQ** is a service interface function block (SIFB) that encapsulates access to a frequency input of a data panel (type 7A/8A). It is used for initialization, cyclic or event-driven querying, and asynchronous notification of frequency changes. The FB is part of a modular control environment for agricultural machinery (MI – machine interface).

Interface Structure

Event Inputs

Event	Description	With
INIT	Service Initialization	`QI`, `PARAMS`, `u8SAMember`, `Input`, `FreqDelta`, `TimeDelta`
REQ	Service Request (Read Current Frequency)	`QI`

Event Outputs

Event	Description	With
INITO	Initialization Confirmation	`QO`, `STATUS`
CNF	Confirmation of a requested query	`QO`, `STATUS`, `IN`
IND	Asynchronous indication (frequency change or time elapsed)	`QO`, `STATUS`, `IN`

Data Inputs

Name	Type	Initial Value	Description
`QI`	BOOL	–	Event input qualifier (controls processing)
`PARAMS`	STRING	–	Service parameter (device-specific configuration)
`u8SAMember`	USINT	`MI::MI_00`	Node address (SA) of the data collection module (value range 224…239)
`Input`	`DataPanel::io::MI::DI::DataPanel_MI_DI_Enum`	`Invalid`	Identifies the physical input (must be `7A` or `8A`)
`FreqDelta`	WORD	–	Frequency change threshold [Hz] that triggers a `IND`
`TimeDelta`	DWORD	–	Time interval [ms] after which a `IND` is sent (even without changes)

Data Outputs

Name	Type	Description
`QO`	BOOL	Event output qualifier (indicates successful execution)
`STATUS`	STRING	Service status (e.g., error messages or confirmation texts)
`IN`	WORD	Current frequency in Hz

Adapters

None available.

Functionality

1. **Initialization** – The event `INIT` establishes the connection to the configured frequency input. The parameters `PARAMS`, `u8SAMember`, `Input`, `FreqDelta`, and `TimeDelta` are adopted. After successful initialization, the output event `INITO` is triggered with `QO = TRUE` and a positive `STATUS`.
2. **Query (REQ/CNF)** – An event `REQ` requests the current frequency value. The function block (FB) sends the request to the data panel and, upon response, returns the event `CNF` with the current value in `IN`. `QO` indicates whether the query was successful.
3. **Asynchronous Indication (IND)** – The FB continuously monitors the frequency input. An event `IND` is triggered when:

- the frequency value changes by at least `FreqDelta` [Hz], or
- the time interval specified in `TimeDelta` [ms] has elapsed since the last event `IND`.

This enables both threshold-based and time-based updates.

The events `INIT` and `REQ` are only executed if the corresponding qualifier `QI` has the value `TRUE`. The output qualifier `QO` signals successful execution.

Technical Features

- **Type Dependency**: The function block uses special types from the package `DataPanel::io::MI::DI`, which enable type-safe configuration of the input (e.g., `DataPanel_MI_DI_S`) as well as constants (`MI::MI_00`).
- **Input Initialization**: The data input `Input` is set to `Invalid` by default. Before the first `INIT`, this must be set to a valid value (`7A` or `8A`), otherwise initialization will fail.
- **Frequency Display**: The measured frequency is output as a 16-bit value (`WORD`) in Hz.
- **Event-Driven Retrieval**: The `INIT` service must be successfully executed once before using `REQ` or receiving `IND`.
- **Attribute Hash**: A `eclipse4diac::core::TypeHash` attribute exists but is not set – this is used for runtime optimization in 4diac.

State Overview

The function block (FB) internally cycles through the following logical states:

State	Description
IDLE	Waiting for `INIT` – no connection
INIT	Establish connection to input and apply parameters
RUN	Ready for operation – waiting for `REQ` or sending `IND` upon change/expiration
ERROR	Error state (e.g., incorrect `Input`, communication error) – `STATUS` contains an error message

A transition to the error state occurs when initialization fails. The only way to return from the error state is by sending `INIT` again.

Application Scenarios

- **Speed measurement** on an agricultural machine: A frequency sensor (e.g., a magnetic transducer) provides pulses that are recorded via the data panel. Using `FreqDelta = 5 Hz` and `TimeDelta = 100 ms`, the controller receives current values both during significant speed changes and at regular intervals.
- **Monitoring a constant frequency**: With a low change threshold (`FreqDelta = 1 Hz`) and a short time interval (`TimeDelta = 50 ms`), the function block (FB) is suitable for real-time monitoring of critical processes.
- **Redundant frequency acquisition**: Two parallel instances of the FB on different inputs (7A and 8A) allow for plausibility checks of the measured values.

Comparison with similar function blocks

Function block	Type	Special feature
`DataPanel_MI_IW_FREQ`	Frequency input (SIFB)	Event-driven, asynchronous IND, configurable thresholds and time values

`DataPanel_MI_DI`	Digital input (SIFB)	Binary states only, no frequency-dependent triggers
Generic `SIFB`with INIT/REQ/IND	General	No built-in frequency functions, must be developed in-house

The **DataPanel_MI_IW_FREQ** is specifically optimized for processing frequency signals, while other modules provide either only discrete states or generic interfaces.

Conclusion

The **DataPanel_MI_IW_FREQ** is a powerful and flexible service interface module for acquiring frequency data via a data panel (Type 7A/8A). Its combination of threshold-based and time-based indication makes it suitable for both simple measurement tasks and safety-critical monitoring. The type-safe configuration and clear event interface facilitate integration into agricultural automation solutions.

The **DataPanel_MI_IW_FREQ** is a powerful and flexible service interface module for acquiring frequency data via a data panel (Type 7A/8A). —

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

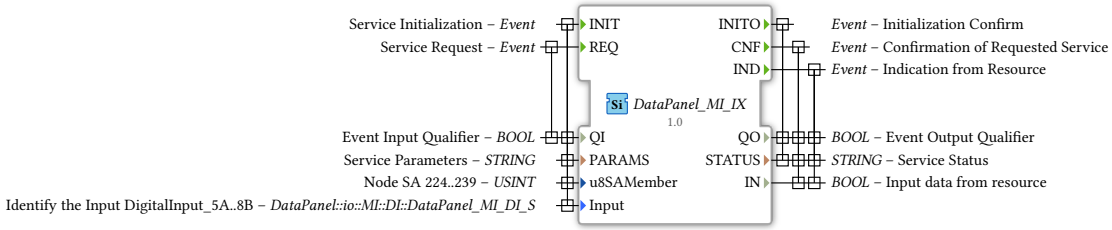
]

DataPanel_MI_IX

• – ◦ • – ◦ • – ◦ •

Introduction

The DataPanel_MI_IX is a service interface function block for Boolean input data, specifically designed for communication with digital inputs in automation systems. This block enables the initialization and querying of digital input signals via standardized service interfaces.



Interface Structure

Event Inputs

- **INIT**: Service initialization with parameters QI, PARAMS, u8SAMember, and Input
- **REQ**: Service request with parameter QI

Event Outputs

- **INITO**: Initialization acknowledgment with parameters QO and STATUS
- **CNF**: Acknowledgement of the requested service operation with parameters QO, STATUS, and IN
- **IND**: Indication from the resource with parameters QO, STATUS, and IN

Data Inputs

- **QI**: BOOL - Event input qualifier
- **PARAMS**: STRING - Service parameter
- **u8SAMember**: USINT - Node SA 224..239 (Initial value: MI::MI_00)
- **Input**: DataPanel_MI_DI_S - Identifies digital input 5A..8B (Initial value: DataPanel_MI_DI::Invalid)

Data Outputs

- **QO**: BOOL - Event output qualifier
- **STATUS**: STRING - Service status
- **IN**: BOOL - Input data from the resource

Adapters

No adapter interfaces available.

Functionality

The function block initializes communication with a specific digital input via the INIT event. After successful initialization, the input state can be queried via the REQ event. The block returns the current state of the digital input and status information via the CNF and IND output events.

Technical Features

- Supports specific SA member addressing in the range 224-239
- Uses structured input identification via the DataPanel_MI_DI_S data type
- Provides both request-based (REQ/CNF) and indication-based (IND) communication
- Initialization with configurable service parameters

State Overview

The function block goes through the following states:

1. **Not Initialized**: Before INIT call
2. **Initialized**: After successful INIT/INITO sequence
3. **Ready**: Can process REQ/CNF cycles
4. **Active**: Processes incoming IND events from the resource

Application Scenarios

- Monitoring of digital inputs in control cabinets
- Integration into PLC controllers for binary signal acquisition
- Industrial automation with specific SA addressing requirements
- Plants with distributed I/O systems

⚖️ Comparison with Similar Blocks

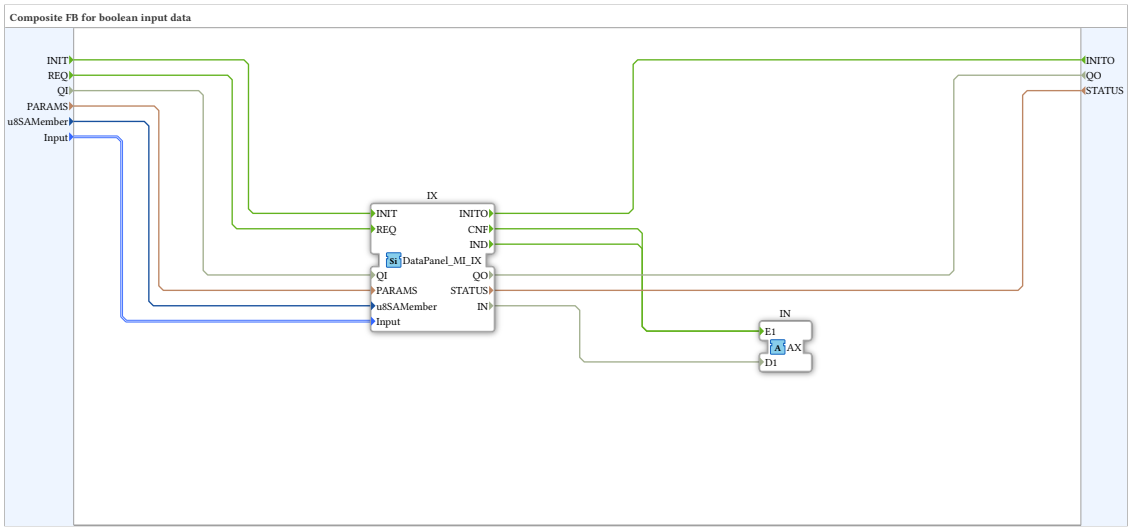
Compared to generic digital input blocks, DataPanel_MI_IX offers:

- Specific addressing for DataPanel systems
- Extended initialization options with parameter strings
- Additional IND functionality for event-based communication
- Structured input identification

Conclusion

DataPanel_MI_IX is a specialized function block for the reliable acquisition of digital input signals in DataPanel environments. Its flexible parameterization and dual operating modes (demand-driven and event-driven) make it ideally suited for demanding automation applications with specific addressing requirements.

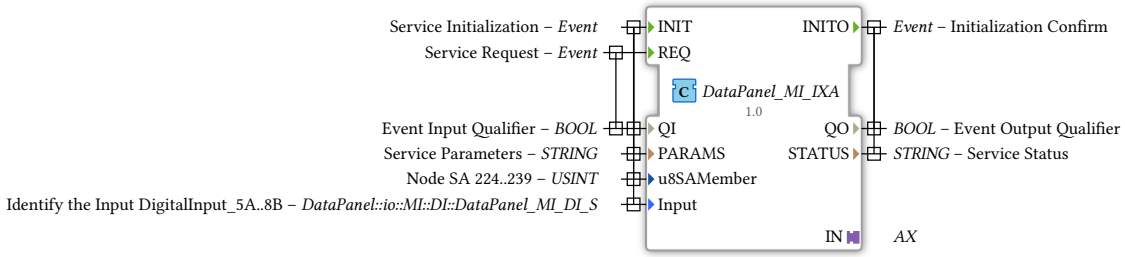
DataPanel_MI_IXA



• – ○ • – ○ • – ○ •

Introduction

The DataPanel_MI_IXA is a composite function block for processing Boolean input data. It serves as an interface for digital inputs and enables the initialization and querying of input data via standardized service interfaces.



Interface Structure

Event Inputs

- **INIT**: Service initialization with the associated data QI, PARAMS, u8SAMember, and Input
- **REQ**: Service request with the associated data QI

Event Outputs

- **INITO**: Initialization acknowledgment with the associated data QO and STATUS

Data Inputs

- **QI**: Event input qualifier (BOOL)
- **PARAMS**: Service parameters (STRING)
- **u8SAMember**: Node SA 224..239 (USINT) with initial value MI::MI_00
- **Input**: Identification of the digital input DigitalInput_5A..8B with initial value Invalid

Data Outputs

- **QO**: Event output qualifier (BOOL)
- **STATUS**: Service status (STRING)

Adapter

- **IN**: Unidirectional adapter of type AX for data communication

Functionality

This function block acts as a wrapper for the internal DataPanel_MI_IX block and forwards all events and data accordingly. During INIT initialization, all parameters are passed to the internal block, and confirmation is provided via INITO. REQ requests are forwarded directly to the internal block.

Technical Features

- Uses special types from the DataPanel::io::MI::DI namespace
- Supports SA members in the range 224-239
- Provides initialization with default values for u8SAMember and Input
- Implements error handling via the Invalid initial value

State Overview

The block has an initialized state and an operating state. After successful INIT initialization, the block is operational and can process REQ requests.

Application Scenarios

- Connecting digital inputs in automation systems
- Integration into DataPanel I/O structures
- Use in control systems with SA bus communication
- Abstraction of hardware inputs in standardized interfaces

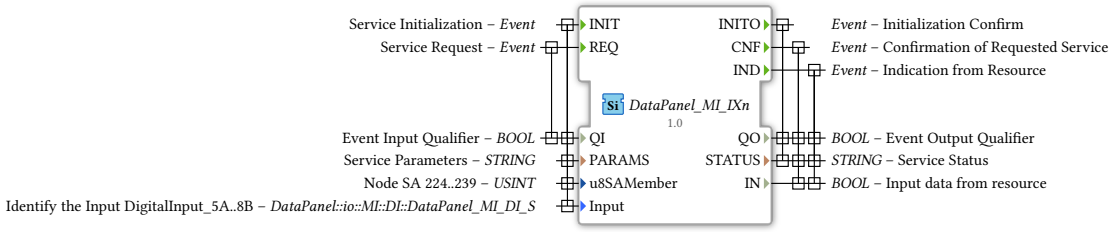
⚖️ Comparison with similar modules

Compared to simple digital input modules, DataPanel_MI_IXA offers extended initialization options and parameterization via the PARAMS input. SA member numbering enables specific addressing in larger systems.

Conclusion

The DataPanel_MI_IXA is a robust and flexible function block for integrating digital inputs into complex automation systems. Its structured interface and extended parameterization options make it particularly suitable for demanding applications in industrial environments.

DataPanel_MI_IXn



• – ○ • – ○ • – ○ •

Introduction

The DataPanel_MI_IXn function block is a service-oriented input block for digital binary inputs (ground switching / NPN). It is used to read a single Boolean input signal from a DataPanel system via the MI (Module Interface) protocol. The block is typically used in the 4diac IDE to connect field devices such as switches or sensors.

Interface Structure

Event Inputs

- **INIT** (EInit): Initializes the block. The parameters QI, PARAMS, u8SAMember, and Input are passed.
- **REQ** (Event): Requests the current input value to be read. Qualified by QI.

Event Outputs

- **INITO** (EInit): Confirms successful initialization. Returns QO and STATUS.
- **CNF** (Event): Acknowledges a completed request (REQ). Contains QO, STATUS, and the read value IN.

- **IND** (Event): Indicates an asynchronous change in the input signal from the resource driver. Contains QO, STATUS, and IN.

Data Inputs

- **QI** (BOOL): Qualifier for the event inputs; typically set to TRUE to enable processing.
- **PARAMS** (STRING): Service-specific parameters for configuration (e.g., addressing or protocol settings).
- **u8SAMember** (USINT): Node SA (Subscriber Address) of the device in the range 224…239. Initial value: MI::MI_00.
- **Input** (DataPanel::io::MI::DI::DataPanel_MI_DI_S): Identification of the digital input (e.g., DigitalInput_5A…8B). Initial value: Invalid.

Data Outputs

- **QO** (BOOL): Output qualifier; indicates whether the output data is valid.
- **STATUS** (STRING): Service status message (e.g., error or success message).
- **IN** (BOOL): The binary input value read from the connected sensor or switch.

Adapters

No adapters defined.

Functionality

The function block communicates with a DataPanel module via the MI protocol. After startup, the function block is initialized with INIT, during which the communication parameters (PARAMS, u8SAMember, Input) are configured. Upon successful initialization, INITO is triggered. The current state of the configured digital input can then be queried using REQ. The response is provided via CNF with the value IN. Additionally, the function block can receive asynchronous events (IND) when the input state changes without an explicit request (e.g., due to hardware interrupts). The QO output indicates the validity of the data, and STATUS provides diagnostic information.

Technical Features

- The function block is implemented as a Service Interface Function Block (SIFB) and expects a low-level hardware implementation in the 4diac runtime.
- The input configuration uses a user-defined structure type `DataPanel_MI_DI_S`, which enables precise channel assignment.
- The parameter `u8SAMember` is limited to values 224–239; this corresponds to typical SA addresses for MI-based subsystems.
- Initialization can only be performed with valid parameters. Initialization fails if the input is invalid (e.g., Invalid).

State Overview

An explicit state machine is not defined in the XML. Typically, the function block goes through the following states:

1. **IDLE** – After startup, waits for INIT.
2. **INIT** – Initialization is performed, configuring the hardware channel.
3. **OPERATE** – Ready for REQ and receiving IND events.
4. **ERROR** – In case of failed initialization or communication errors.

Alternative states may vary depending on the communication driver implementation.

Application Scenarios

- **Field Device Connection**: A binary proximity sensor (npn) is connected to a DataPanel input. The function block reads the switching state and makes it available in the control application.
- **Switch Monitoring**: A manually operated switch (ground switching) is monitored via the function block; Changes are reported asynchronously via IND.
- **Diagnostics**: Errors such as wire breaks or address conflicts can be detected via STATUS.

Comparison with similar function blocks

Similar function blocks such as `DataPanel_MI_DI_Xn` (for other input groups) or `DataPanel_MI_DO_Xn` (outputs) share the same interface but differ in the number of channels or the data direction. This function block focuses on a single digital input and is suitable for precise monitoring.

Conclusion

The `DataPanel_MI_IXn` function block provides a standardized interface for reading a binary input in DataPanel systems via the MI protocol. Its service-oriented structure (INIT/REQ/IND) allows for flexible integration into event-driven automation solutions. Configuration via dedicated parameters ensures simple and robust connectivity.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

DQ

DataPanel_MI_DO



• – ◦ • – ◦ • – ◦ •

Introduction

DataPanel_MI_DO is a global constant definition for medium I/O density DataPanel modules for digital outputs. This constant defines a structured mapping of pins for various digital output channels and power supply ports.

Interface Structure

Event Inputs

Not present - This is a global constant definition

Event Outputs

Not present - This is a global constant definition

Data Inputs

Not present - This is a global constant definition

Data Outputs

Not present - This is a global constant definition

Adapters

Not present - This is a global constant definition

Functionality

The global constant DataPanel_MI_DO defines fixed pin assignments for a medium I/O density DataPanel module. Each constant represents a specific output channel or power supply port and assigns it a fixed pin value. The structure is based on the data type `DataPanel_MI_DO_S` with a pin array.

Technical Features

- Defines 16 digital output channels (1A-8B) with pin assignments 1-16
- Includes 4 power supply ports (Input_Power_Port_5-8) with pin assignments 17-20
- Provides an invalid constant with a pin value of 255 for error handling
- Structured organization in pairs (A/B) for improved clarity
- Fixed pin assignments guarantee consistent hardware control

State Overview

Not applicable - These are static constant definitions

Application Scenarios

- Hardware configuration of DataPanel modules with medium I/O density
- Simplifying pin mapping in control programs
- Reusable constant definitions in larger automation projects
- Hardware-abstract programming using symbolic pin names

⚖️ Comparison with Similar Components

Compared to individual pin definitions, this global Constants:

- Centralized management of all pin assignments
- Typed access via structured data types
- Improved maintainability and code readability
- Consistent naming conventions across the entire project

Conclusion

DataPanel_MI_DO provides an efficient and structured way to manage pin assignments for DataPanel modules with medium digital output density. The use of global constants improves code maintainability and allows for centralized management of hardware changes.

DataPanel_MI_DO_S

DataPanel Modules with medium IO density DO Struct

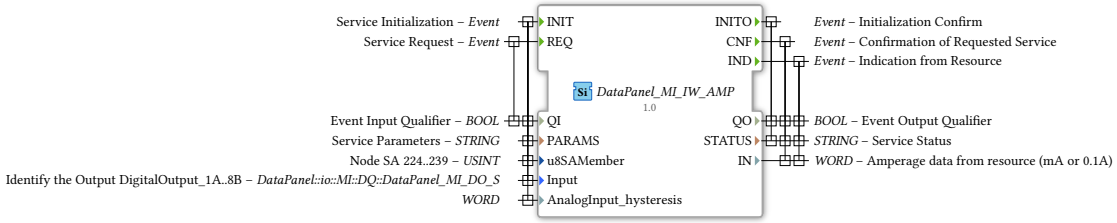
Structure

Name	Type	Initial Value
Pin	USINT	255

🛠️ Zugehörige Übungen

- Funk_IX_TO_DataPanel_MI_QX
- Funk_IXA_TO_DataPanel_MI_QXA

DataPanel_MI_IW_AMP



• — ○ • — ○ • — ○ •

Introduction

The DataPanel_MI_IW_AMP function block is an input interface function block (Service Interface Function Block) for acquiring the current feedback (amperage) of a single channel. It serves to integrate current measurements into the 4diac IDE and enables communication with a higher-level controller or a data panel. The block initializes the hardware-level communication and returns the current measurement value as a WORD upon each request.

Interface Structure

Event Inputs

- `INIT (EInit)` – Service initialization. Triggered to configure the function block and establish the connection to the hardware. This event transfers the parameters QI, PARAMS, u8SAMember, Input, and AnalogInput_hysteresis.

- `REQ (Event)` – Service request. Triggers a new measurement query. The function block responds with the CNF event and provides the current amperage value on the IN output.

Event Outputs

- `INITO (EInit)` – Initialization confirmation. Sent after successful initialization. Outputs the QO and STATUS status.
- `CNF (Event)` – Service request confirmation. Sent in response to REQ and contains the current measurement value IN as well as the status information QO and STATUS.
- `IND (Event)` – Resource indication. Triggered by asynchronous events from the hardware (e.g., spontaneous value changes). Also provides the current measurement value IN.

Data Inputs

- `QI (BOOL)` – Input Qualifier. Controls whether the FB is active (TRUE) or not (FALSE).
- `PARAMS (STRING)` – Service Parameter. Contains configuration data for initialization, e.g., bus interface or addressing.
- `u8SAMember (USINT)` – Node SA (Source Address) in the range 224..239. Defines the unique address of the connected device. Initial value: `MI::MI_00`.
- `Input (DataPanel_MI_DO_S)` – Identifies the digital output whose current is to be measured (e.g., DigitalOutput_1A..8B). Initial value: `Invalid`.
- `AnalogInput_hysteresis (WORD)` – Hysteresis for the analog input. Can be used to prevent edge detection during small fluctuations.

Data Outputs

- `QO (BOOL)` – Output qualifier. Indicates whether the function block is operating correctly (TRUE) or if an error has occurred (FALSE).
- `STATUS (STRING)` – Service status. Contains error messages or information about the operating status.
- `IN (WORD)` – Measured current value in units of mA or 0.1A (depending on the configuration). Provides the current amperage value of the selected channel.

Adapters

No adapters defined.

Functionality

The function block acts as a service interface between the application and the hardware level. When `INIT` is triggered, the configuration parameters (address, channel assignment, hysteresis) are passed to the hardware. After successful initialization, `INITO` is sent. A `REQ` event triggers an immediate query of the current measurement value, which is provided on the output `IN` and acknowledged with `CNF`. Additionally, the hardware can generate a `IND` event upon changes to asynchronously inform the application.

`IND`` The values of QIandQOcontrol the activity state: The function block (FB) can only deliver data ifQI=TRUE and QO=TRUE. In case of an error, QOis set to FALSE and an error message is stored inSTATUS`.

Technical Features

- The FB is designed as a generic service interface block and requires a hardware-dependent implementation of the service functions.
- Addressing via `u8SAMember` follows a predefined set of constants (e.g., `MI::MI_00` … `MI::MI_15`), which allows for easy configuration of multiple channels.
- The output type `DataPanel_MI_DO_S` is used to select a specific digital output; the function block will only operate correctly if this value is valid (not `Invalid`).
- Hysteresis (`AnalogInput_hysteresis`) can be used to suppress measurement noise and avoid unnecessary events.
- The amperage output is a WORD, depending on the configuration, in mA (e.g., 0–65535 mA) or in 0.1 A increments.

State Overview

The function block typically goes through the following states:

1. **IDLE** – Idle state after startup, waiting for INIT.
2. **INIT** – Initialization is performed; after success, transition to OPERATIONAL, then INITO.

3. **OPERATIONAL** – Ready for operation; REQ events are processed, measured values are delivered via CNF; asynchronous IND is possible.
4. **ERROR** – Error state (e.g., communication error, invalid parameters); QO = FALSE, STATUS contains error text; reset only by re-INIT.

A detailed state machine is embedded in the service implementation.

Application Scenarios

- **Current monitoring of individual outputs** in a machine controller, e.g., for detecting load changes or defects.
- **Integration of analog input modules** of a data panel to measure the current consumption per channel.
- **Parameterization via INIT** enables flexible use in various hardware configurations.

Comparison with Similar Function Blocks

Compared to a purely digital input function block, this block offers analog measured value feedback (amperage), which provides more information about the load state. Similar modules for voltage or temperature measurement use a comparable interface scheme (INIT/REQ/IND) with other physical units. The advantage of `DataPanel_MI_IW_AMP` lies in its specific optimization for current measurement and its integration into the DataPanel family.

Conclusion

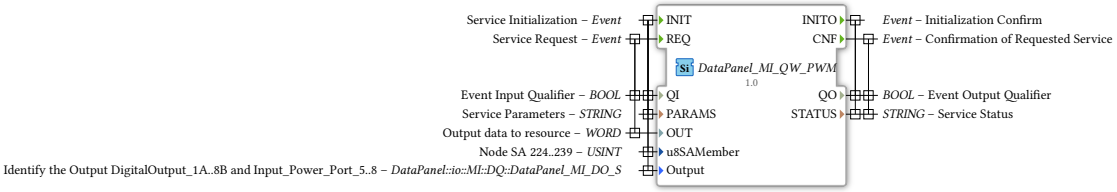
The `DataPanel_MI_IW_AMP` function block provides a standardized and flexible interface for acquiring current values in automation systems. Its clearly defined events and parameters allow for easy integration into 4diac projects and offer both synchronous and asynchronous notifications of measured value changes. Detailed configuration via address, channel, and hysteresis enables adaptable use in diverse hardware environments.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

DataPanel_MI_QW_PWM



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

Interface

Event inputs

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

Event Outputs

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

Input Vars

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

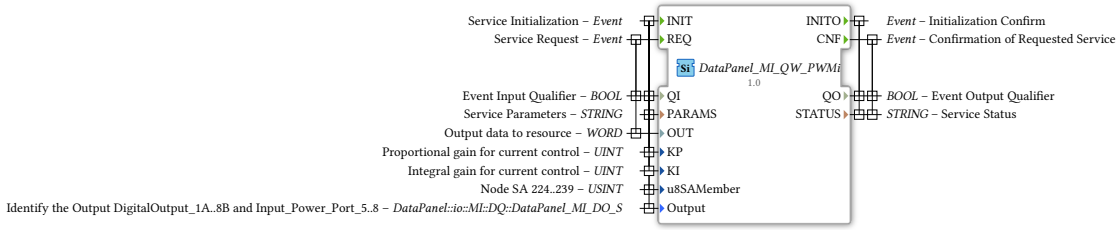
Output Vars

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

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 The PWM signal & infographic on ms-muc-docs.de

DataPanel_MI_QW_PWMi



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

Interface

Event inputs

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

Event Outputs

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

Input Vars

Name	Type	Initial Value	Comment
QI	BOOL		Event Input Qualifier
PARAMS	STRING		Service Parameters

OUT	WORD		Output data to resource
CP	UINT	70	Proportional gain for current control
AI	UINT	30	Integral gain for current control
u8SAMember	USINT	MI::MI_00	Node SA 224..239
Output	DataPanel::io::MI::DQ::DataPanel_MI_DO_S	Invalid_S	Identify the Output DigitalOutput_1A..8B and Input_Power_Port_5..8

Output Vars

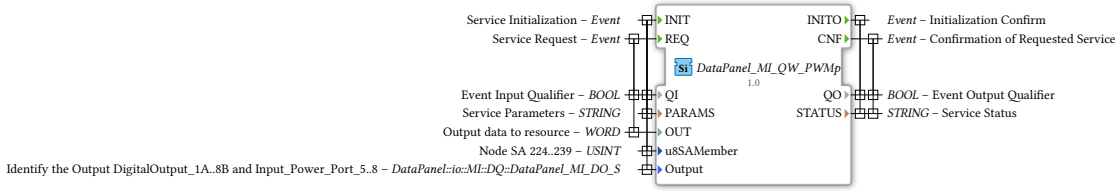
Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 The PWM signal & infographic on ms-muc-docs.de

]

DataPanel_MI_QW_PWMp



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

Interface

Event inputs

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

Event Outputs

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

Input Vars

Name	Type	Initial Value	Comment
QI	BOOL		Event Input Qualifier
PARAMS	STRING		Service Parameters
OUT	WORD		Output data to resource

u8SAMember	USINT	MI::MI_00	Node SA 224..239
Output	DataPanel::io::MI::DQ::DataPanel_MI_DO_S	Invalid	Identify the Output DigitalOutput_1A..8B and Input_Power_Port_5..8

Output Vars

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

🌐 Related topic subpages on ms-muc-docs.de

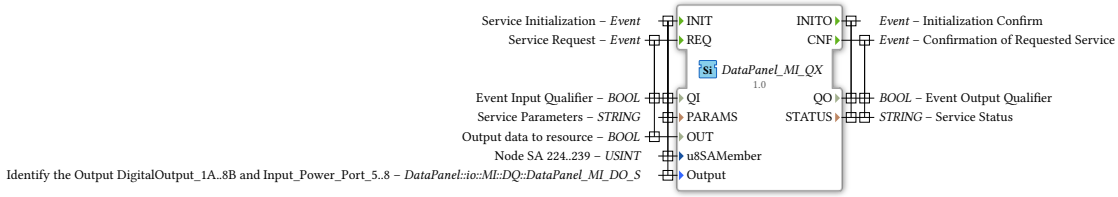
- 🌐 The PWM signal & infographic on ms-muc-docs.de

DataPanel_MI_QX

• – ◦ • – ◦ • – ◦ •

Introduction

The DataPanel_MI_QX is a Service Interface Function Block for outputting Boolean data to a resource. This block serves as an interface between the control logic and the physical outputs of a DataPanel MI system and enables the configuration and control of digital outputs.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **OUT** (BOOL): Output Data for Resource
- **u8SAMember** (USINT): Node SA 224..239, Initial Value: MI::MI_00
- **Output** (DataPanel_MI_DO_S): Identifies the output (DigitalOutput_1A..8B and Input_Power_Port_5..8), Initial Value: Invalid

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status

Adapters

No adapter interfaces are available.

Functionality

The DataPanel_MI_QX function block manages communication with the digital outputs of a DataPanel MI system. During initialization (INIT event), the service parameters are configured and the hardware interface is set up. The REQ event triggers the output of Boolean data to the specified resource. The function block acknowledges both initialization and service operations via the corresponding output events.

Technical Features

- Supports the configuration of SA nodes in the 224-239 range
- Enables the identification of specific digital outputs (1A to 8B) and power ports (5-8)
- Uses a special data type (DataPanel_MI_DO_S) for output identification
- Implements robust error handling through STATUS output

State Overview

The Function Block goes through the following states:

1. **Not Initialized**: Waiting for an INIT event
2. **Initialized**: Ready for service requests
3. **Service Execution**: Processing REQ events
4. **Acknowledgement**: Sending CNF/INITO events with status information

Application Scenarios

- Control of digital outputs in industrial automation systems
- Connection of DataPanel MI hardware components
- Implementation of secure output protocols
- Integration into larger control systems with multiple output channels

⚖️ Comparison with Similar Function Blocks

Compared to simple digital output blocks, DataPanel_MI_QX offers advanced functionalities such as:

- Configurable SA node configuration
- Specific hardware identification for DataPanel MI systems
- Detailed status feedback
- Flexible output configuration via the output parameter

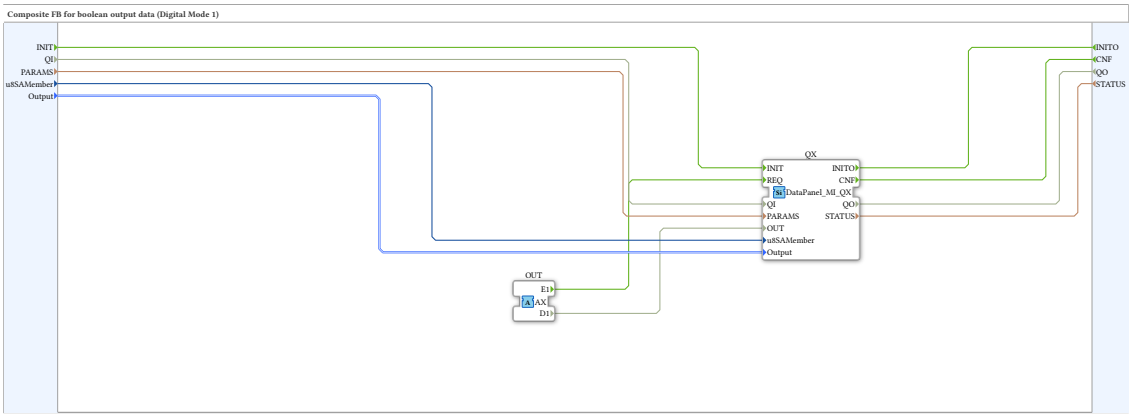
🛠️ Related Exercises

- Exercise_003b2_sub
- Exercise_010a4

Conclusion

The DataPanel_MI_QX Function Block represents a powerful and flexible solution for controlling digital outputs in DataPanel MI systems. Its extensive parameterization and detailed status feedback make it particularly suitable for complex automation applications that require a reliable and configurable output interface.

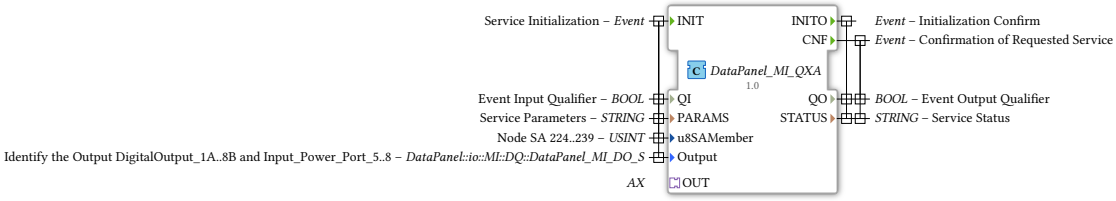
DataPanel_MI_QXA



• – ○ • – ○ • – ○ •

Introduction

The DataPanel_MI_QXA is a composite function block for controlling Boolean output data in automation systems. This block serves as an interface for digital outputs and enables the configuration and control of output channels via standardized service parameters.



Interface Structure

Event Inputs

- **INIT**: Service initialization event with associated data QI, PARAMS, u8SAMember, and Output

Event Outputs

- **INITO**: Initialization acknowledgment with QO and STATUS
- **CNF**: Acknowledgement of the requested service with QO and STATUS

Data Inputs

- **QI** (BOOL): Event input qualifier
- **PARAMS** (STRING): Service parameter
- **u8SAMember** (USINT): Node SA address in the range 224-239, initialized with MI::MI_00
- **Output** (DataPanel_MI_DO_S): Output identification (DigitalOutput_1A..8B and Input_Power_Port_5..8), initialized with Invalid

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status

Adapters

- **OUT**: Unidirectional AX-type adapter for output control

Functionality

The DataPanel_MI_QXA composite function block encapsulates the functionality of the DataPanel_MI_QX block and extends it with an adapter interface. Upon initialization, the configuration parameters are passed to the internal QX block. The OUT adapter enables external control of the output functions via standardized interfaces.

Technical Features

- Supports SA addresses in the range 224-239
- Uses specific output identification for digital outputs and power ports
- Integrated error handling through invalid initialization
- Unidirectional adapter architecture for clean interface separation

State Overview

The function block goes through the following states:

1. **Initialization Phase**: The INIT event processes configuration parameters
2. **Ready State**: Waits for control requests via adapter OUT
3. **Active Operation**: Processes output commands and generates CNF acknowledgments

Application Scenarios

- Control of digital outputs in industrial automation
- Connection of power port controllers
- Integration into modular control systems with SA addressing

- Use in distributed automation systems

⚖️ Comparison with Similar Function Blocks

Compared to simple digital output function blocks, DataPanel_MI_QXA offers:

- Extended parameterization options via STRING Parameters
- SA Addressing Support for Network Integration
- Composite Architecture with Adapter Interface
- Comprehensive Status Feedback

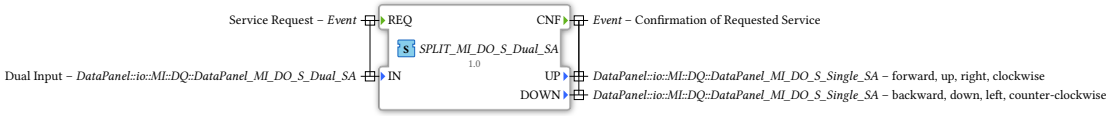
🛠️ Related Exercises

- Exercise_003b2_sub_AX
- Exercise_010a4_AX

Conclusion

The DataPanel_MI_QXA is a powerful composite function block for controlling Boolean output data in complex automation environments. Its modular architecture and standardized interfaces make it ideal for use in distributed control systems with high demands on flexibility and maintainability.

SPLIT_MI_DO_S_Dual_SA



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SPLIT_MI_DO_S_Dual_SA** is used to split a single data panel input of type `DataPanel_MI_DO_S_Dual_SA` into two separate outputs of type `DataPanel_MI_DO_S_Single_SA`. The drive information contained in the dual structure for forward/reverse movement (or up/down, right/left, clockwise/counterclockwise) is distributed to two independent outputs.

Interface Structure

Event Inputs

Event	Description
REQ	Service Request: The block processes the current value of the data input IN.

Event Outputs

Event	Description
CNF	Confirmation of processing completed. Sent after the algorithm has finished executing.

Data Inputs

Name	Type	Description
IN	`DataPanel::io::MI::DQ::DataPanel_MI_DO_S_Dual_SA`	Dual input carrying both forward (UP) and backward (DOWN) information.

Data Outputs

Name	Type	Description
UP	`DataPanel::io::MI::DQ::DataPanel_MI_DO_S_Single_SA`	Output for forward, upward, clockwise, or counterclockwise rotation.

DOWN	DataPanel::io::MI::DQ::DataPanel_MI_DO_S_Single_SA	Output for reverse, downward, counterclockwise, or counterclockwise rotation.

Adapter

None.

Functionality

The function block is activated by the **REQ** event. The associated algorithm performs the following assignments:

- The common control value `u8SAMember` of the dual structure is passed unchanged to both outputs (`UP.u8SAMember` and `DOWN.u8SAMember`).
- The direction information is separated: Output `UP.Port` receives the value of field `IN.Up`, and output `DOWN.Port` receives the value of field `IN.Down`.
- After the algorithm executes, the acknowledgment event **CNF** is sent.

Technical Features

- The function block is implemented as a simple function block (SimpleFB), meaning the logic is executed within a single algorithm without a state machine.
- The data types used are from the `DataPanel::io::MI::DQ` package. They are declared as implicit imports in the function block.
- The output data is valid at the time of the **CNF** event.

State Overview

The function block has only a single EC state, **REQ**, in which the algorithm is executed and the output event **CNF** is subsequently set. There are no other states or branches.

Application Scenarios

- Splitting a combined drive signal for a bidirectional motor into two separate control lines (e.g., for direction control).
- Separating forward and reverse signals in a machine control system when downstream function blocks expect separate inputs for each direction of movement.
- Preparing data for two independent actuators that operate in the same mode (e.g., power level) but control different directions.

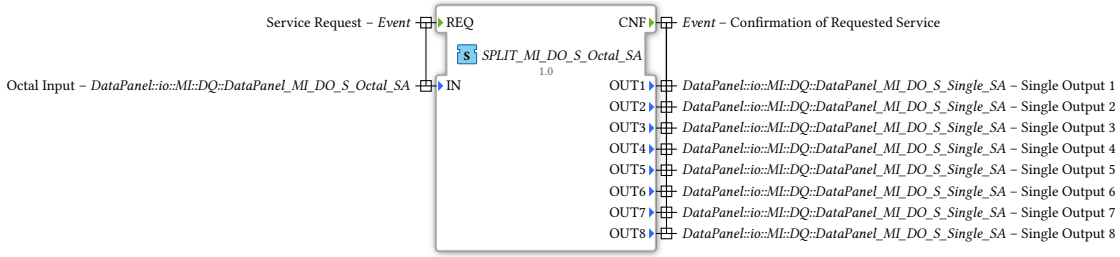
Comparison with Similar Function Blocks

A possible alternative approach would be to use a general-purpose splitter function block that splits arbitrary compound data types. **SPLIT_MI_DO_S_Dual_SA** is specifically tailored to the aforementioned data types and directly handles the correct assignment of the direction fields, reducing the potential for errors and improving the readability of the control code.

Conclusion

SPLIT_MI_DO_S_Dual_SA offers a simple and reliable way to split a dual data panel control command into two separate outputs for opposite directions of movement. Thanks to its clearly defined interface and direct processing from an event call, it is particularly well-suited for use in modular control systems where a clean separation of the direction signals is required.

SPLIT_MI_DO_S_Octal_SA



• — ○ • — ○ • — ○ •

The function block **SPLIT_MI_DO_S_Octal_SA** is used to split a single data value of type DataPanel_MI_DO_S_Octal_SA into eight separate outputs of type DataPanel_MI_DO_S_Single_SA. The common portion (u8SAMember) is retained unchanged, and the respective port information from the octal input is distributed to the individual outputs.

Event	Comment	
REQ	Service Request; Processing starts as soon as the event occurs. The user must first validate the data input IN.	
Event	Comment	
-----	-----	
CNF	Confirmation of completed processing. Indicates that all eight outputs have been updated.	
Name	Type	Comment
--	----------	----------
IN	DataPanel_MI_DO_S_Octal_SA	Octal input value (Port 1...8 and common u8SAMember)
Name	Type	Comment
--	----------	----------
OUT1	DataPanel_MI_DO_S_Single_SA	Single Output 1
OUT2	DataPanel_MI_DO_S_Single_SA	Single Output 2
OUT3	DataPanel_MI_DO_S_Single_SA	Single Output 3
OUT4	DataPanel_MI_DO_S_Single_SA	Single Output 4
OUT5	DataPanel_MI_DO_S_Single_SA	Single Output 5
OUT6	DataPanel_MI_DO_S_Single_SA	Single Output 6
OUT7	DataPanel_MI_DO_S_Single_SA	Single Output 7
OUT8	DataPanel_MI_DO_S_Single_SA	Single Output 8

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters available.

Functionality

This function block implements a simple, event-driven process:

1. An incoming REQ event triggers the execution of the algorithm REQ.

No adapters available.

This function block implements a simple, event-driven sequence:

1. Each output also receives the corresponding port value from the octal input:

- `OUT1.Port := IN.Port1`
- `OUT2.Port := IN.Port2`
- ...
- `OUT8.Port := IN.Port8`

1. Upon completion, the event `CNF` is sent, indicating that all eight outputs are valid.

- The function block is implemented as a **SimpleFB** (simple function block) and operates without a state machine with multiple states – processing occurs in a single step.
- There is no concurrency or time-critical processes; the partitioning occurs synchronously with the `REQ` event.
- The types `DataPanel_MI_DO_S_Octal_SA` and `DataPanel_MI_DO_S_Single_SA` must be defined in the same data namespace (here, `DataPanel::io::MI::DQ`).

The function block has exactly one state, `REQ`:

- **Entry** via the `REQ` event.
- **Action**: Execution of the algorithm `REQ`.
- **Output**: Sending the `CNF` event (all outputs are carried out simultaneously).

There are no other states that indicate delay or error handling.

- Splitting an octal data bus (e.g., a module with eight identical digital outputs) into individual logical channels.
- Interface between an octal sensor/actuator panel and a controller that processes individual outputs separately.
- Preprocessing in automation technology when a data packet contains multiple ports that need to be forwarded separately.

Comparable components exist for other data structures (e.g., `SPLIT_MI_DO_S_Octal_SA` for specific panel types). The basic mechanism—splitting a composite data type into its components—is universal but differs depending on the data field name and type.

The `SPLIT_MI_DO_S_Octal_SA` is a compact and clearly structured function block for decomposing an octal data packet into eight individual signals. It facilitates the modularization of automation software by dividing complex data types into manageable units and directly inheriting the port assignment from the source structure.

Technical Features

State Overview

Application Scenarios

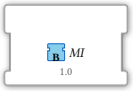
Comparison with Similar Function Blocks

Conclusion

const

MI

🎧 Podcast



- Eclipse 4diac 3.0: ST Interpreter, FBE, and 7200 Commits – The Turbo for Distributed Automation
- Industry 4.0: The Magic of Self-Configuring Systems with fortiss, IEC 61499, and Eclipse 4diac
- From Pyramid to Plug-and-Play: The Rise of Self-Configurable Industrial Automation
- The E_CTU in IEC 61499: Event-Driven Counting and Why the Minimalist is Convincing in Mechanical Engineering
- The E_PERMIT Block: The “Gatekeeper” for Events in IEC 61499 Systems Decoded ()

📺 Video

- The Great Migration ()

Introduction

The MI Global Constants represent a collection of constants for medium I/O density modules. These constant definitions are used in the 4diac IDE to provide consistent and reusable addressing information for I/O modules.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

No adapters available

Functionality

MI Global Constants are purely constant declarations without any active functionality. The constants define address ranges for 16 different I/O modules (MI_00 to MI_15), each with a unique numeric value in the range of 224 to 239.

Technical Features

- All constants are of type USINT (Unsigned Short Integer)
- The values form a continuous range from 224 to 239
- Constants are declared as global and constant
- Package assignment: DataPanel::io::MI::const

State Overview

No state transitions are present, as these are static constant definitions

Application Scenarios

- Addressing I/O modules in control applications
- Reuse in various function blocks
- Centralized management of module addresses
- Simplified project configuration

⚖️ Comparison with Similar Blocks

Compared to other constant definitions, MI provides specific addresses for modules with medium I/O density. Other constant sets might cover different address ranges or module types.

Conclusion

The MI Global Constants offer a convenient and centralized way to manage address information for I/O modules. Using these constants improves the maintainability and readability of control programs, as module address changes can be made centrally.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter Block on ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.1.3 Radio

2.1.3.1 🎧 Podcast

- DIN EN 61499-1 Demystified: Function Blocks, Object Orientation and Distributed Systems
- DIN EN 61499-1: Building Blocks for Industry 4.0 – How Function Blocks Revolutionize Automation
- DIN EN 61499-1: Function Blocks and the Paradigm Shift in Industrial Software Development
- DIN EN 61499: Mastering Modular and Event-Based Industrial Controls with Function Blocks – The ESR Switch in Focus
- Event FBs according to IEC 61499: Lego Bricks of Automation – How Event-Based Function Blocks Work

2.1.3.2 🔌 io

📺 Video

- Automation of the Future
- The Great Migration
- Digital Decisions
- Ideal Diodes
- The Dual Wi-Fi Solution

🎧 Podcast

- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Target Groups, Structure, and Unique Selling Proposition
- Taming the Software Dragon: Industrial Automation and the Future of Production
- Eclipse 4diac: Innovation Engine Research vs. User Needs – What Really Drives Development?
- Taming the Software Dragon: Why IEC 61499 Could Revolutionize Industry

---

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Diode & Semiconductor Basics on ms-muc-docs.de
- 🌐 Smart Switch vs. Relay on ms-muc-docs.de

DI

📺 Video

- Decoding the datasheet
- The Great Migration
- The Art of Soldering
- The World of Standardization
- Digital Decisions

🎧 Podcast

- 4diac IDE: Your “Hello World” of Automation – The Blinking Tutorial
- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- 4diac IDE: How the IEC 61499 Standard is Shaping Industrial Automation Revolutionized
- 4diac presentation: Target groups, structure, and unique selling proposition Sharpening
- Taming the software dragon: Industrial automation and the future of production

---

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Radio_DI



• – ◦ • – ◦ • – ◦ •

Introduction

Radio_DI is a GlobalConstants package for the 4diac IDE that defines constants for digital inputs in radio systems. The package provides predefined pin configurations for various digital input signals that can be used in control applications.

Interface Structure

Because Radio_DI is a GlobalConstants package, it does not have traditional function block interfaces. Instead, it provides global constants.

Event Inputs

Not available - GlobalConstants package

Event Outputs

Not available - GlobalConstants package

Data Inputs

Not available - GlobalConstants package

Data Outputs

Not available - GlobalConstants package

Adapters

Not available - GlobalConstants package

Functionality

The Funk_DI package defines a collection of global constants of type `Funk_DI_S`, representing digital input pins. Each constant contains a pin number that identifies the physical or logical input. The constants can be used in other function blocks to reference specific digital inputs.

Technical Features

- **Pin Assignments**: The pin numbers follow a specific scheme:
- Standard Inputs: Pins 1-15
- Special Functions: Pin 0 (STOP), Pin 25 (START)
- System Functions: Pin −1 (STATUS), Pin −6 (COUNTER)
- Invalid Value: Pin 127 (Invalid)
- **Data Type**: All constants use the structured data type `Funk_DI_S`
- **Package Structure**: Organized in the namespace `Funk::io::DI`

State Overview

Since this is a constant package, there are no state transitions. All values are hard-coded at compile time and are unchangeable.

Application Scenarios

- **Digital Input Configuration**: Simplified configuration of digital inputs in control applications
- **Standardization**: Ensuring consistent pin assignments across different projects
- **Maintainability**: Centralized management of all digital input configurations
- **Wireless Systems**: Specifically designed for applications in wireless control systems

⚖️ Comparison with Similar Components

Compared to other digital input constant packages, Funk_DI offers:

- Specific pin assignments for wireless systems
- Predefined constants for special functions (START, STOP, STATUS, COUNTER)
- Structured organization within the I/O area of the Funk namespace

Conclusion

The Funk_DI GlobalConstants package provides a convenient collection of predefined constants for digital inputs in wireless systems. Its standardized pin assignments and clear naming conventions enable efficient and easy-to-maintain configuration of control applications. The special functions such as START, STOP, and STATUS make it particularly suitable for industrial control applications with wireless components.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Funk_DI_S

Funk_DI Struct

Structure

Name	Type	Initial Value
Pin	SINT	127

🛠️ Zugehörige Übungen

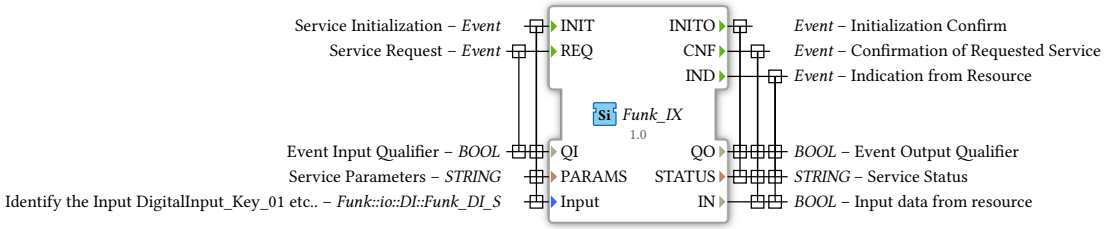
- Funk_IX_TO_DataPanel_MI_QX
- Funk_IXA_TO_DataPanel_MI_QXA

Funk_IX

• – ◦ • – ◦ • – ◦ •

Introduction

The Funk_IX is a Service Interface Function Block for Boolean input data. It serves as an interface for communication with digital inputs in distributed automation systems and enables the initialization, querying, and asynchronous notification of input signals.



Interface Structure

Event Inputs

- **INIT**: Service Initialization
- Linked to: QI, PARAMS, Input
- **REQ**: Service Request
- Linked to: QI

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Service
- Linked to: QO, STATUS, IN
- **IND**: Indication from Resource
- Linked to: QO, STATUS, IN

Data Inputs

- **QI**: Event Input Qualifier (BOOL) - Qualifier for event inputs
- **PARAMS**: Service Parameters (STRING) - Service parameters
- **Input**: Identifies the input (Funk::io::DI::Funk_DI_S) - Identifies the digital input (e.g., DigitalInput_Key_01)

Data Outputs

- **QO**: Event Output Qualifier (BOOL) - Qualifier for event outputs
- **STATUS**: Service Status (STRING) - Service status
- **IN**: Input data from resource (BOOL) - Input data from the resource

Adapter

No adapter interfaces are available.

Functionality

The Funk_IX Function Block implements a Service Interface Pattern for digital inputs. During initialization (INIT event), the service parameters are configured and the specific digital input is identified. Input values can be queried via the REQ event, resulting in a CNF response with the current input value. Additionally, the block supports asynchronous IND events, which are triggered by changes in the input signal's state.

Technical Features

- Uses the special data type `Funk::io::DI::Funk_DI_S` to identify digital inputs
- Initial value for input is set to "Invalid"
- Supports both synchronous queries (REQ/CNF) and asynchronous notifications (IND)
- Includes comprehensive status feedback via the STATUS output

State Overview

The Function Block goes through the following states:

1. **Not Initialized**: Before INIT event
2. **Initialized**: After successful INIT/INITO sequence
3. **Query Ready**: Ready for REQ/CNF cycles
4. **Notification Ready**: Ready for asynchronous IND events

Application Scenarios

- Monitoring of digital sensors and switches
- Integration into distributed control systems
- Connection of field devices in industrial automation
- Condition monitoring of machines and systems

⚖️ Comparison with Similar Function Blocks

Compared to Funk_IX offers simple digital input blocks with enhanced service functionality, including parameterization, status feedback, and asynchronous notifications. Its unique input identification enables precise addressing in complex systems.

🛠️ Related Exercises

- Exercise_003b2_sub

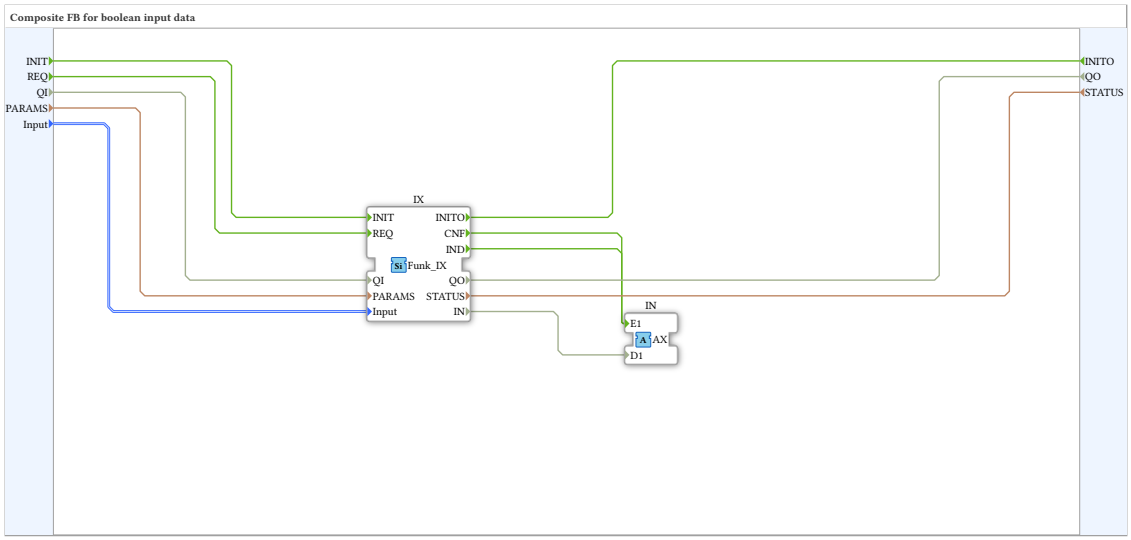
Conclusion

The Funk_IX Function Block provides a powerful and flexible solution for integrating digital inputs into 4diac-based automation systems. Its service-oriented architecture, with initialization, query, and notification functions, makes it particularly suitable for demanding industrial applications.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

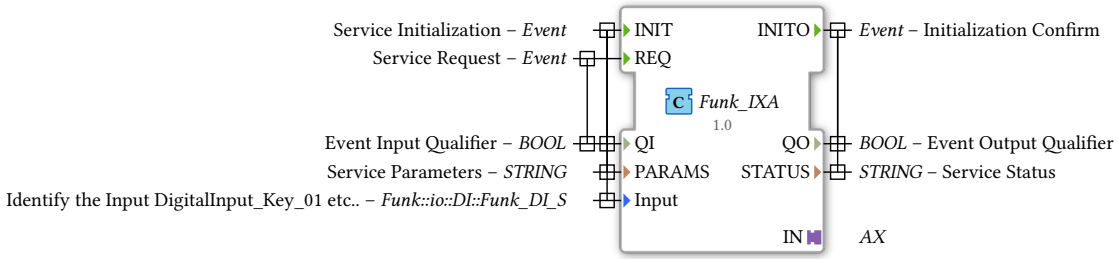
Funk_IXA



• – ○ • – ○ • – ○ •

Introduction

The **Funk_IXA** function block is a composite function block for processing Boolean input data. It serves as an interface for digital inputs and enables the initialization and querying of input signals via a standardized interface.



Interface Structure

Event Inputs

- **INIT**: Service Initialization (with QI, PARAMS, Input)
- **REQ**: Service Request (with QI)

Event Outputs

- **INITO**: Initialization Acknowledgement (with QO, STATUS)

Data Inputs

- **QI**: Event Input Qualifier (BOOL)
- **PARAMS**: Service Parameters (STRING)
- **Input**: Digital Input Identification (Funk_DI_S, Initial Value: Invalid)

Data Outputs

- **QO**: Event Output Qualifier (BOOL)
- **STATUS**: Service Status (STRING)

Adapters

- **IN**: Unidirectional Adapter Type AX

Functionality

The The Funk_IXA block acts as a wrapper for the underlying Funk_IX block. During INIT initialization, the parameters QI, PARAMS, and Input are passed to the internal IX block. The REQ request triggers a query of the current input state. The IN adapter enables communication with external systems.

Funk_IXA acts as a wrapper for the underlying Funk_IX block.

Technical Features

- Uses the specific data type Funk_DI_S to identify digital inputs
- Implements invalid initialization for the input parameter
- Supports string-based parameter passing for flexible configuration
- Provides status feedback via the STATUS output

State Overview

The block goes through the following states:

1. **Not Initialized**: Before INIT call
2. **Initialized**: After successful INIT processing
3. **Ready**: Can process REQ requests
4. **Active**: During REQ processing

Application Scenarios

- Connecting digital inputs in automation systems
- Integration into distributed control systems
- Use in agricultural machinery controls
- Interface for sensor data acquisition

⚖️ Comparison with Similar Blocks

Compared to simple digital input blocks, Funk_IXA offers:

- Extended parameterization options
- Status feedback
- Adapter interface for extended communication
- Typed Input Identification

🛠️ Related Exercises

- Exercise_003b2_sub_AX

Conclusion

The Funk_IXA function block represents a robust and flexible solution for processing digital input signals. Its composite structure and the use of standardized interfaces make it particularly suitable for complex automation applications that require reliable data acquisition and status monitoring.

2.1.4 MyLib (test_AX)

2.1.4.1 sys

🎧 Podcast

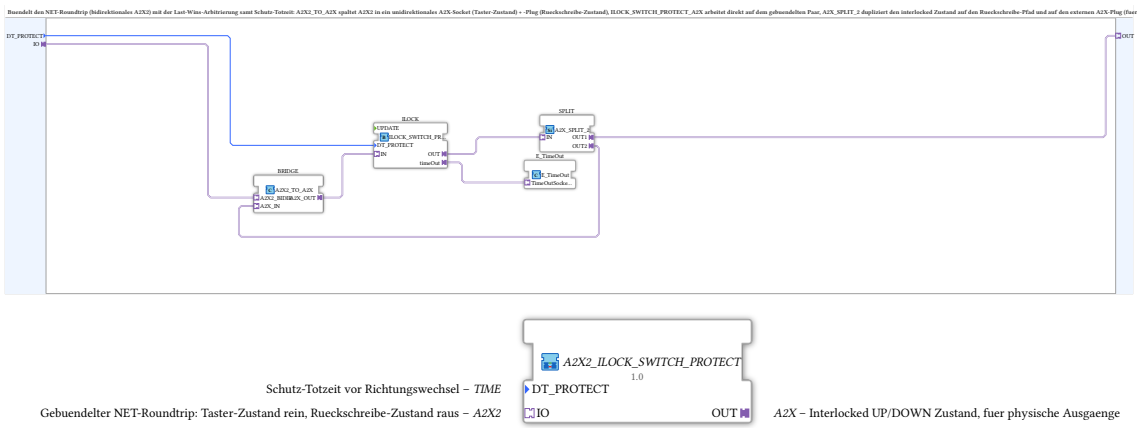
- Eclipse 4diac: Your Bike for Industry 4.0 – Mastering Software Overhead and Controlling Distributed Systems
- IEC 61499: The Future of Industrial Automation and Distributed Systems
- Industry 4.0: The Magic of Self-Configuring Systems with fortiss, IEC 61499, and Eclipse 4diac
- Eclipse 4diac: Revolutionizing Industrial Control with Open-Source Cyber-Physical Systems
- [The E_PERMIT Block: The “Gatekeeper” for Events in IEC 61499 Systems] decrypted] (https://podcasters.spotify.com/pod/show/iec-61499-grundkurs-de/episodes/Der-E_PERMIT-Baustein-Der-Trsteher-fr-Ereignisse-in-IEC-61499-Systemen-entschlsselt-e3681m5)

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

A2X2_ILOCK_SWITCH_PROTECT



• — ○ • — ○ • — ○ •

Introduction

The **A2X2_ILOCK_SWITCH_PROTECT** subapplication bundles a bidirectional A2X2 network round-trip with a last-wins arbitration mechanism including a protective dead time. It combines three logical functions in a single composite component: a bidirectional-to-unidirectional adapter bridge, an interlock switch protector operating on the paired adapter channels, and a splitter that duplicates the interlocked state both to the write-back return path and to an external plug for physical output driving.

The subapplication follows a “SUB style” architecture where the bridge, interlock, and splitter reside inside the composite (subapplication), rather than being placed in the device resource. This encapsulation hides the internal complexity and makes the component reusable across different hardware configurations.

Interface Structure

The subapplication interface consists of one data input, one bidirectional socket adapter, and one unidirectional plug adapter. No explicit event inputs or event outputs are exposed at the composite boundary; all event flow is carried implicitly through the adapter connections.

Event Inputs

None. All event handling is encapsulated within the adapter-based communication flow.

Event Outputs

None. Results propagate through the adapter channels.

Data Inputs

Name	Type	Initial Value	Comment
`DT_PROTECT`	`TIME`	`T#50ms`	Protective dead time before direction change is permitted.

`DT_PROTECT` defines the minimum time period that must elapse before a direction reversal is accepted, suppressing rapid toggling and mechanical bounce effects.

Data Outputs

None.

Adapters

Direction	Name	Type	Comment
Socket	`IO`	`adapter::types::bidirectional::A2X2`	Bundled network round-trip: button state enters, write-back state exits.

Plug	OUT	adapter::types::unidirectional::A2X	Interlocked UP/DOWN state, intended for physical output drivers.

- **IO** is the bidirectional A2X2 socket receiving the raw operator button states (up/down) from the network and sending back the acknowledged/interlocked state.
- **OUT** is a unidirectional A2X plug that provides the interlocked UP/DOWN output signals to be connected to actual output hardware (e.g., relays, contactors, motor drivers).

Functionality

The subapplication implements a complete interlock-protected switching chain for two-directional actuators (e.g., a lift, a slide, a directional valve). The logic flow is as follows:

1. **Bridge stage** – The internally instantiated A2X2_TO_A2X function block splits the bidirectional A2X2 socket into two unidirectional A2X channels: one acting as the input (button state reception) and one as the output (write-back transmission).
2. **Interlock stage** – The ILOCK_SWITCH_PROTECT_A2X block receives the raw input from the bridge and applies a **last-wins arbitration** algorithm. When a direction change is requested, the protection logic enforces a dead time (DT_PROTECT) during which the new direction cannot be activated. This prevents simultaneous activation of both directions and guards against rapid switching.
3. **Time-out stage** – An internal E_TimeOut event block is triggered by the interlock's timeout mechanism. It provides the temporal reference signal that controls when the protection window expires and the new direction is accepted.
4. **Split stage** – The A2X_SPLIT_2 block takes the interlocked output from the interlock stage and duplicates it:
   - The first copy (OUT1) is routed to the subapplication's external OUT plug for driving physical outputs.
   - The second copy (OUT2) is fed back into the bridge's input channel (A2X_IN), which propagates the state back through the IO socket as the acknowledged write-back signal.

This closed-loop structure ensures that the state acknowledged to the network always matches the state physically commanded at the output, while the interlock prevents hazardous direction conflicts.

Technical Features

- **Bidirectional round-trip support**: Uses the A2X2 bidirection adapter, carrying both the incoming operator request and the outgoing acknowledged state on a single combined channel.
- **Last-wins arbitration**: Implements priority handling where the most recent button command takes precedence, with a time-based guard against rapid reversals.
- **Configurable protective dead time**: The DT_PROTECT input allows the user to tune the minimum time between direction changes (default 50 ms) per application requirements.
- **Integrated time-out generation**: The internal E_TimeOut block functions as the watchdog/timing engine, eliminating the need for external timing resources.
- **Internal state duplication**: The splitter stage generates two synchronized output copies—one for the external physical world, one for the network write-back—guaranteeing consistency.
- **Composite encapsulation**: All three functional stages (bridge, interlock, splitter) are contained within the subapplication, simplifying resource integration and promoting reuse.
- **No explicit event interface**: The subapplication relies entirely on adapter-based event propagation, reducing interface complexity at the composite level.

State Overview

Although the subapplication does not expose an explicit state machine, the internal interlock behavior implies the following logical states:

State	Condition	Output Behavior
Idle	No button pressed, or both buttons released	Both UP and DOWN outputs are inactive (off).
UP Active	UP button pressed, no active DOWN conflict	UP output active; DOWN output inactive.

DOWN Active	DOWN button pressed, no active UP conflict	DOWN output active; UP output inactive.
Blocked / Switching	A direction change was requested during the dead time (`DT_PROTECT` still running)	The previous direction is maintained until the dead time expires; the new direction is then applied (last-wins).
Timeout Expired	`E_TimeOut` fires after the configured delay	The pending direction change is committed; the output state transitions accordingly.

The `E_TimeOut` block acts as a flip-flop reset mechansim: it triggers the transition from "Blocked" to the new active direction once the protection window has elapsed.

Application Scenarios

- **Lift / elevator control**: Interlocking the UP and DOWN commands of a hoist drive to prevent simultaneous activation, with a dead time to suppress bounce from mechanical limit switches.
- **Directional valve control** in hydraulic or pneumatic systems: Ensures that a spool valve is not commanded to two opposite positions at once.
- **Electric actuator control** (e.g., sliding gates, blinds, awnings): Protects the drive motor from rapid reversals that could cause overheating or mechanical wear.
- **Remote push-button panels**: Used as a central safety proxy between a network-based control panel (A2X2) and the local power electronics.
- **Machine safety circuits**: Provides a configurable switching delay between directions where a mandatory cool-down or mechanical settling period is required.

Comparison with Similar Blocks

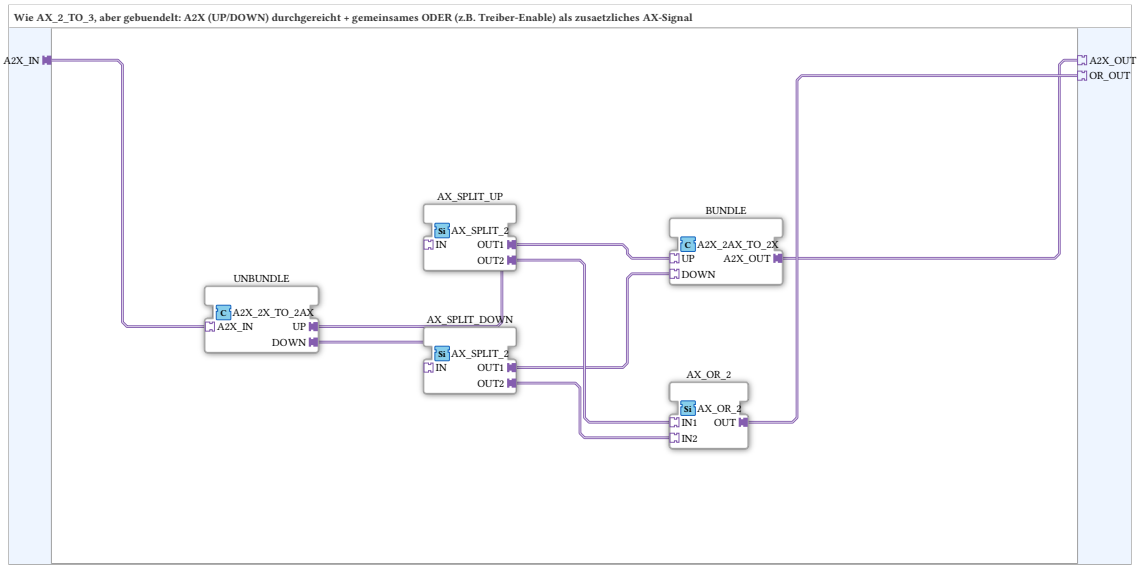
Feature	A2X2_ILOCK_SWITCH_PROTECT	ILOCK_SWITCH_PROTECT_A2X (standalone)	Simple interlock without dead time
Adapter interface	Bidirectional A2X2 + unidirectional A2X	Already operates on a bundled pair (A2X)	Typically discrete Boolean inputs
Dead-time protection	Yes, configurable via `DT_PROTECT`	Yes (if configured directly)	No
Network round-trip support	Yes, internal bridge handles A2X2 splitting	Requires external bridging	Not applicable
Output duplication	Internal splitter provides output + write-back	Output only, write-back must be added externally	Output only
Integration complexity	Low (single composite component)	Medium (requires external bridging and splitting)	High (requires external logic)
Encapsulation level	Complete (bridge + interlock + splitter)	Partial	None

The key advantage of this composite over using `ILOCK_SWITCH_PROTECT_A2X` alone is the integrated handling of the bidirectional A2X2 adapter channel and the automatic duplication of the result for both physical output and write-back feedback. This reduces wiring errors and external block count.

Conclusion

The **A2X2_ILOCK_SWITCH_PROTECT** subapplication provides a complete, encapsulated solution for interlocked, direction-protected switch control over a bidirectional A2X2 network connection. By combining the adapter conversion, last-wins arbitration with protective dead time, and feedback duplication into a single composite, it simplifies system integration, reduces external dependencies, and ensures consistent state propagation between the network and the physical output stage. Its configurable `DT_PROTECT` parameter makes it adaptable to a wide range of machinery and automation applications where safe directional switching is critical.

A2X_TO_A2X_AX

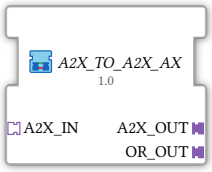


• – ○ • – ○ • – ○ •

Introduction

`A2X_TO_A2X_AX` is the bundled counterpart to `AX_2_TO_3`: Instead of two separate `AX` sockets/plugs (`UP_IN`/`DOWN_IN`, `UP_OUT`/`DOWN_OUT`), this block receives UP/DOWN signals bundled together as a single `A2X` signal, passes it through unchanged, and additionally provides a single `AX` signal that is the OR of both directions (e.g., for a common driver enable).

Gebuendeltes UP/DOWN Eingangssignal – *A2X*



A2X – Gebuendeltes UP/DOWN Ausgangssignal (unveraendert durchgereicht)
AX – ODER aus UP und DOWN

Function Blocks (FBs) Used

Sub-Blocks: A2X_TO_A2X_AX

- **Type**: SubAppType
- **Internal FBs Used**:
  - **UNBUNDLE**: `adapter::conversion::unidirectional::A2X_2X_TO_2AX` — splits the incoming `A2X_IN` into two separate `AX` signals (UP, DOWN).
  - **AX_SPLIT_UP** / **AX_SPLIT_DOWN**: each `adapter::events::unidirectional::AX_SPLIT_2` — each branch into a return path (back to the bundled output signal) and a branch for OR operation.
  - **BUNDLE**: `adapter::conversion::unidirectional::A2X_2AX_TO_2X` — re-bundles UP/DOWN into a single `A2X_OUT`.
  - **AX_OR_2**: `adapter::booleanOperators::AX_OR_2` — returns on `OR_OUT` whether the actuator is currently moving in either direction.
- **Functionality**: `A2X_IN` is unbundled, each direction is split (once back into the bundle, once into the OR operation), UP/DOWN are bundled again (`A2X_OUT`), and the OR operation of both directions is additionally located on `OR_OUT`.

Program Flow and Connections

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

Technical Features

- **Unchanged passthrough despite bundling**: Externally, `A2X` remains bundled; internally, UP/DOWN is passed through unchanged, as with `AX_2_TO_3`.
- **One split per direction**: Necessary because each of the two unbundled signals must go to two destinations—back into the bundling and into the OR gate.

Application Scenarios

- Circuits where UP/DOWN are already implemented as a single `A2X` signal (e.g., direction on a double-action valve) and should not be split into two separate `AX` signals.

Comparison with Similar Components

If UP/DOWN are already implemented individually as `AX`, `AX_2_TO_3` should be used instead—functionally identical (pass-through plus OR as a third signal), but with an unbundled interface.

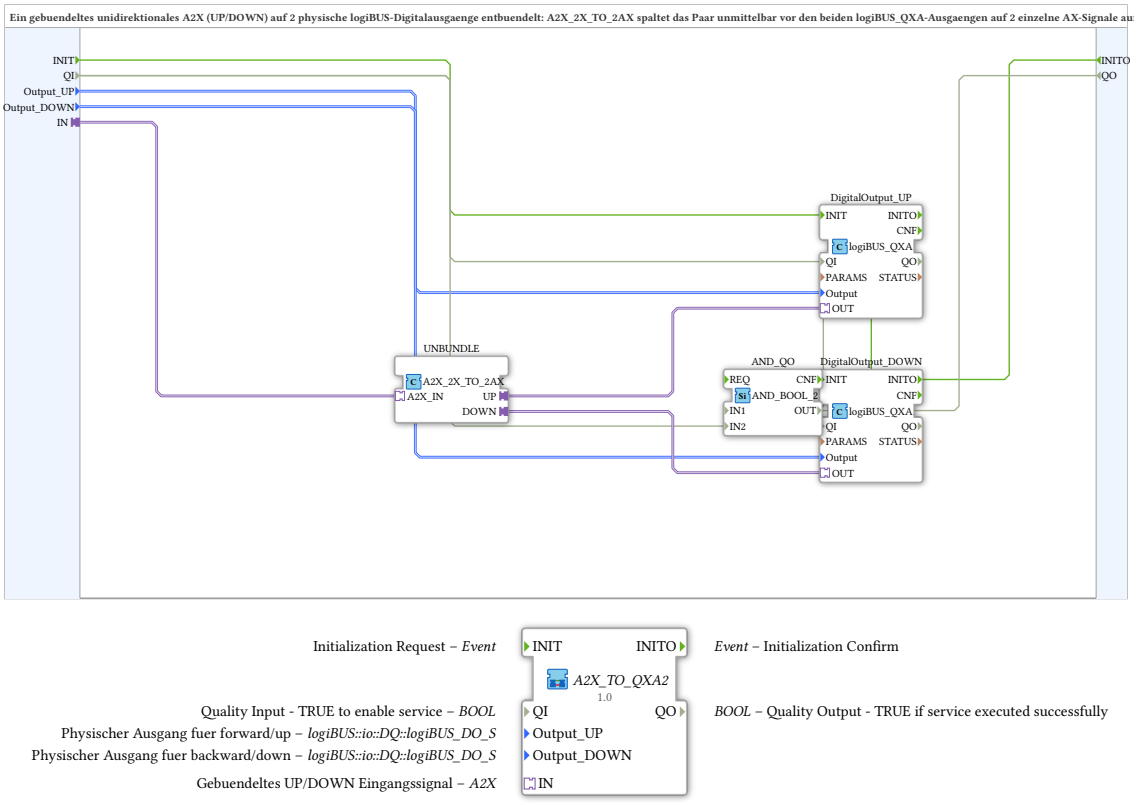
Summary

`A2X_TO_A2X_AX` passes a bundled `A2X` signal through unchanged and adds an additional `AX` OR signal—the bundled counterpart to `AX_2_TO_3`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

A2X_TO_QXA2



• – ○ • – ○ • – ○ •

Introduction

The **A2X_TO_QXA2** subapplication (composite type) acts as a bundling/unbundling adapter between a single bidirectional A2X signal (UP/DOWN) and two physical logicBUS digital outputs. It takes a bundled A2X adapter input and splits it into two separate AX signals, which are then routed to two independent logicBUS QXA digital output function blocks. The unbundling logic is encapsulated within the composite itself, so the device resource remains simple and does not need to handle the separation. This design

follows a “sub style” approach: the composite contains the unbundling and output mapping, making it reusable and modular.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirm

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	TRUE	Quality Input - TRUE to enable service.
Output_UP	logiBUS::io::DQ::logiBUS_DO_S	Invalid	Physical output for forward/up direction.
Output_DOWN	logiBUS::io::DQ::logiBUS_DO_S	Invalid	Physical output for backward/ down direction.

These data inputs are directly connected to the Output parameter of the internal logicBUS QXA function blocks.

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully.

Adapters

Direction	Name	Type	Comment
Socket	IN	adapter::types::unidirectional::A2X	Bundled UP/DOWN input signal.

Functionality

The subapplication receives a bundled A2X adapter signal (IN). Internally, this signal is fed into an instance of the A2X_2X_TO_2AX conversion adapter (named UNBUNDLE), which splits the bundled A2X into two separate AX signals: UP and DOWN. These two signals are then individually connected to the OUT adapter ports of two logiBUS_QXA function blocks (DigitalOutput_UP and DigitalOutput_DOWN). Each of these QXA blocks drives a physical logicBUS digital output, whose configuration is provided via the subapplication’s data inputs Output_UP and Output_DOWN.

The internal data connections directly assign the subapplication data inputs to the respective QXA block’s Output parameter, while the adapter connections carry the signal flow from the input through the unbundler and to the two QXA outputs.

Initialization is passed through the event chain INIT → DigitalOutput_UP.INIT, DigitalOutput_UP.INITO → DigitalOutput_DOWN.INIT, DigitalOutput_DOWN.INITO → INITO. QI is forwarded to the QI inputs of both QXA blocks. The QO outputs of both QXA blocks are combined by the AND_QO block (type iec61131::booleanOperators::AND_BOOL_2) and output as QO; AND_QO is triggered via REQ by DigitalOutput_DOWN.INITO.

Technical Features

- **Composite encapsulation**: All unbundling and output-driving logic is contained within the subapplication, simplifying the upper-level resource design.
- **Modularity**: The subapplication can be reused wherever a single A2X input must control two separate physical outputs (e.g., a forward/reverse motor drive).
- **Initialization events only**: Apart from `INIT`/`INITO` there are no events; signal processing operates purely on data and adapter connections, which keeps the interface simple and suitable for cyclic or continuous processing.
- **Aggregated status**: `QO` is the AND (`AND_QO`) of the `QO` outputs of `DigitalOutput_UP` and `DigitalOutput_DOWN`.
- **Parameterization**: The QXA function blocks receive their `QI` input from the subapplication's `QI` input (initial value `TRUE`).
- **Type compatibility**: Uses standard 4diac adapter types (`adapter::types::unidirectional::A2X` and `adapter::conversion::unidirectional::A2X_2X_TO_2AX`) and logicBUS I/O types.

State Overview

The subapplication itself does not maintain any internal state. The state behavior is entirely determined by the underlying QXA output blocks and the conversion adapter. No explicit state machine is defined in this composite.

Application Scenarios

- **Motor control**: When a single control signal (e.g., from a PLC) provides a direction command (UP/DOWN), the subapplication distributes the command to separate output channels for forward and reverse operation.
- **Actuator redundancy**: In systems where two independent physical outputs are required for the same logical signal (e.g., for safety or redundancy), this block provides a clean separation.
- **Interface adaptation**: When a device has a bundled A2X interface but the physical actuators require individual AX signals, this subapplication bridges the gap without additional glue logic.

Comparison with Similar Blocks

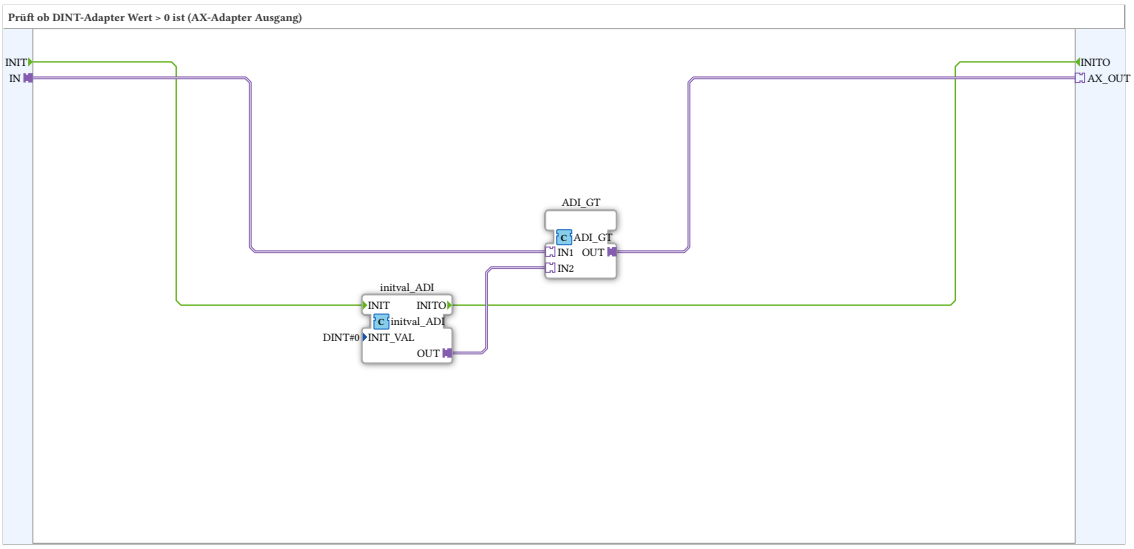
Compared to a direct wiring of the A2X signal to two QXA outputs, this subapplication encapsulates the unbundling step. Other possible approaches:

- **Direct use of `A2X_2X_TO_2AX`**: This would require the developer to manually instantiate the conversion adapter and connect the outputs to QXA blocks in the resource. The `A2X_TO_QXA2` subapplication simplifies this by pre-configuring the connections and parameter values.
- **Similar subapplications**: If other subapplications exist for different unbundling ratios (e.g., A2X to two separate AX outputs), they would differ in the number of outputs and the internal mapping. This specific variant is optimized for two independent physical outputs.

Conclusion

The `A2X_TO_QXA2` subapplication provides a clear, reusable solution for splitting a bundled A2X signal into two physical logicBUS digital outputs. Its composite nature keeps the device resource clean and promotes system modularity. With only initialization events and straightforward data/adapter connections, it integrates easily into control applications that require directional output separation.

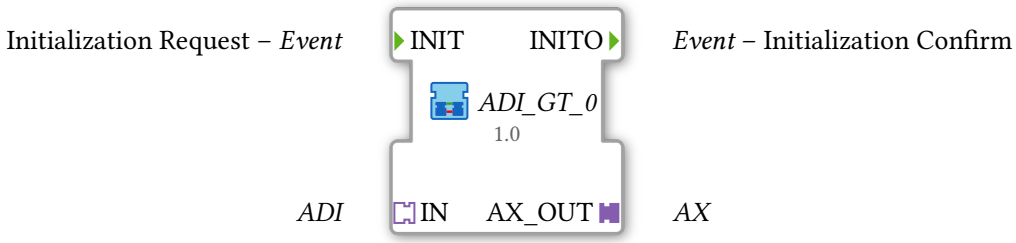
ADI_GT_0



• – ◦ • – ◦ • – ◦ •

Introduction

`ADI_GT_0` checks whether a DINT adapter value is greater than 0 and outputs the result as a boolean AX adapter signal. A typical input is a signed 32-bit value. The block is built identically for every numeric type (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Interface

- **Event input**: `INIT` (initialization)
- **Event output**: `INITO` (initialization finished, comes from the internal `initval_ADI`)
- **Socket**: `IN` (`ADI`) — the value compared against 0
- **Plug**: `AX_OUT` (`AX`) — TRUE when `IN > 0`

Function Blocks (FBs) Used

Sub-blocks: ADI_GT_0

- **Type**: SubAppType
- **Internal FBs used**:
  - **initval_ADI**: `adapter::types::unidirectional::ADI::initval::initval_ADI` — supplies the fixed comparison constant `DINT#0` as an adapter value (`INIT_VAL=DINT#0`). An adapter socket cannot take the constant as a parameter, so it is fed in through this block.
  - **ADI_GT**: `adapter::iec61131::comparison::ADI_GT` — adapter-native comparison `IN1 > IN2`.
- **Functionality**: `IN` ⟶ `ADI_GT.IN1`; constant 0 ⟶ `ADI_GT.IN2`; result ⟶ `AX_OUT`.

Program Flow and Connections

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

Application Scenarios

- Converting a signed 32-bit value into a simple boolean adapter signal for VT status displays or enable logic.

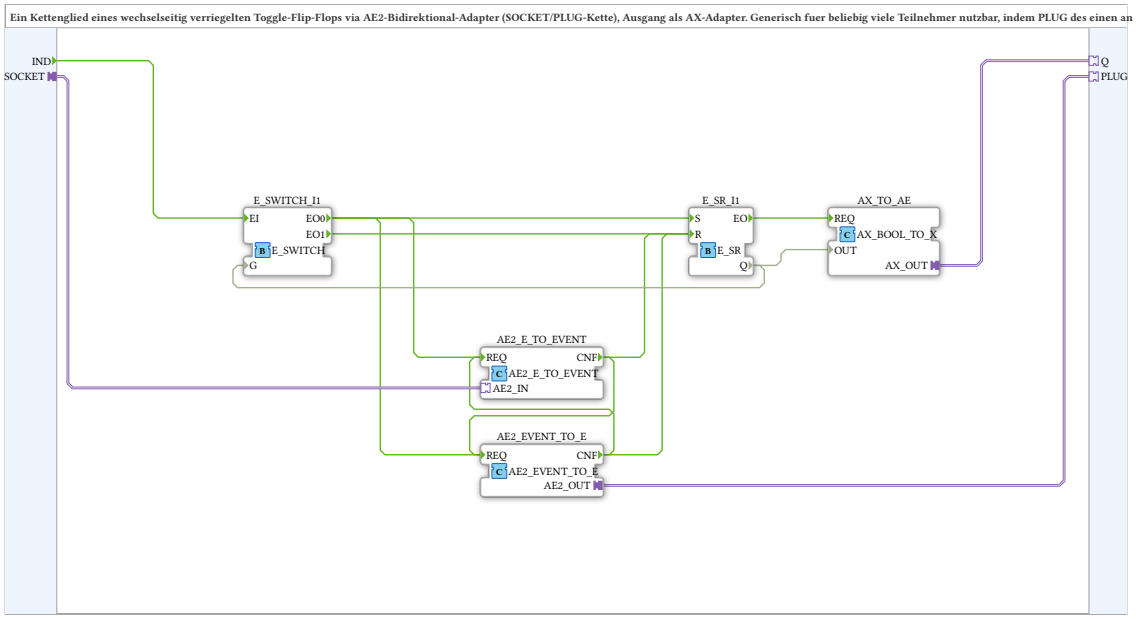
Summary

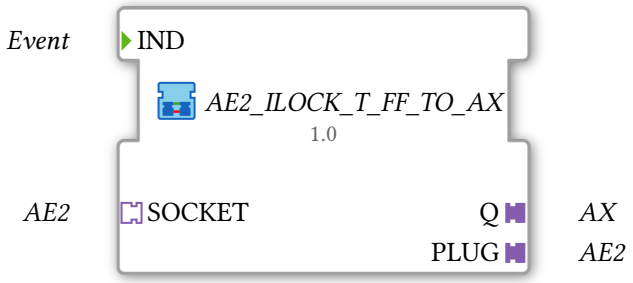
`ADI_GT_0` is a compact wrapper around the adapter-native comparison `ADI_GT` — converts “DINT > 0” directly into an AX adapter signal, so the caller does not have to create and wire the zero constant itself.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AE2_ILOCK_T_FF_TO_AX





• – ○ • – ○ • – ○ •

Introduction

`AE2_ILOCK_T_FF_TO_AX` is a reusable subapplication that implements a single link of a **mutually interlocked toggle flip-flop chain**. It is designed to be used in a network where multiple instances are connected in series via standard bidirectional AE2 adapters. The subapp guarantees that at any time only one output in the entire chain is active. A click (event) on the local input toggles the state of that link, while simultaneously forcing all other links to become inactive through a propagating reset signal sent along the chain.

The block is generic and can be used for any number of participants. It provides an AX adapter as the output interface, making it compatible with unidirectional adapter consumers.

Interface Structure

Event Inputs

Name	Type	Description
IND	Event	Toggle trigger. When a rising edge occurs, the internal state changes (set if currently inactive, reset if currently active), and the reset signal is sent to neighboring links.

Event Outputs

The subapplication has **no dedicated event output** interfaces. All communication with other links is performed via the AE2 adapters.

**Data Inputs

There are **no direct data inputs** exposed to the user. The internal logic relies solely on the event input and the data carried by the AE2 adapters.

**Data Outputs

There are **no direct data outputs**. The result is available via the Q adapter (AX type), which carries a boolean value.

Adapters

Name	Type	Direction	Purpose
SOCKET	AE2 (bidirectional)	Socket (input side)	Receives reset signals from the previous link in the chain.
PLUG	AE2 (bidirectional)	Plug (output side)	Sends reset signals to the next link in the chain.
Q	AX (unidirectional)	Plug (output side)	Provides the current state as a boolean value (true = active, false = inactive).

Functionality

The subapp implements a **toggle flip-flop with mutual exclusion**. Its internal network consists of:

- An E_SR set/reset flip-flop storing the current state.
- An E_SWITCH that routes the incoming IND event based on the current state.
- Two bidirectional adapter-conversion blocks (AE2_EVENT_TO_E and AE2_E_TO_EVENT) that translate between regular events and AE2 adapter events.
- A unidirectional converter (AX_BOOL_TO_X) that maps the internal boolean state to the output adapter Q.

Operation:

1. **Toggle action:** When an IND event arrives, E_SWITCH checks the current state Q.
   - If Q = false (inactive), the event is routed to the S (set) input of E_SR, causing the state to become active. Simultaneously, an event is sent to the AE2_EVENT_TO_E block, which emits a reset signal on the PLUG adapter.
   - If Q = true (active), the event is routed to the R (reset) input of E_SR, causing the state to become inactive. No reset signal is propagated in this case.
2. **Interlock propagation:** When a reset signal is received from the previous link via the SOCKET adapter, the AE2_E_TO_EVENT block converts it to a regular event. This event resets the local E_SR (forcing Q to false) and also sends a new event to AE2_EVENT_TO_E, which forwards the reset signal to the next link via PLUG. Thus the reset propagates down the chain, ensuring all other links become inactive.
3. **Output:** The internal state Q is directly fed to the AX_BOOL_TO_X converter, which produces the boolean value on the Q adapter.

The combination of the toggle action and the reset propagation guarantees that after any IND event, exactly one link in the chain is active – the link that received the event (if it was inactive) – and all others are reset.

Technical Features

- **Modularity:** The subapp is designed as a self-contained chain element. Multiple instances can be connected by linking `PLUG` of one to `SOCKET` of the next.
- **Synchronisation:** The AE2 adapter conversions handle event propagation with minimal delay, and the internal SR flip-flop ensures state stability.
- **Scalability:** The chain length is unlimited; the propagation delay grows linearly with the number of elements, but the interlock logic remains correct.
- **Output compatibility:** The `AX` output allows direct connection to unidirectional consumers (e.g., indicators, control words) without additional conversion.
- **Event-driven:** All operations are triggered by events; no cyclic polling is required.

State Overview

The subapp has two distinct states:

State	`Q` value	Description
Inactive	`false`	The link is not active. An incoming `IND` event will toggle it to active and propagate a reset signal.
Active	`true`	The link is the only active one in the chain. An incoming `IND` event toggles it back to inactive.

Transitions:

- **Inactive → Active:** Triggered by `IND` when `Q = false`.
- **Active → Inactive:** Triggered by `IND` when `Q = true`, or by a reset signal received from the chain (via `SOCKET`).

Internal reset signals do not change the state if the link is already inactive; they are simply forwarded.

Application Scenarios

- **Exclusive mode selection:** A set of mutually exclusive operating modes in a machine. Each mode is represented by one subapp instance; clicking a button selects the corresponding mode and deactivates all others.
- **Priority-free resource allocation:** A group of clients that must share a single resource. The first client that requests the resource gets it, and all others are automatically denied.
- **Single-active workflow steps:** A sequence of steps in a production system where only one step may run at a time, with manual triggering to advance.

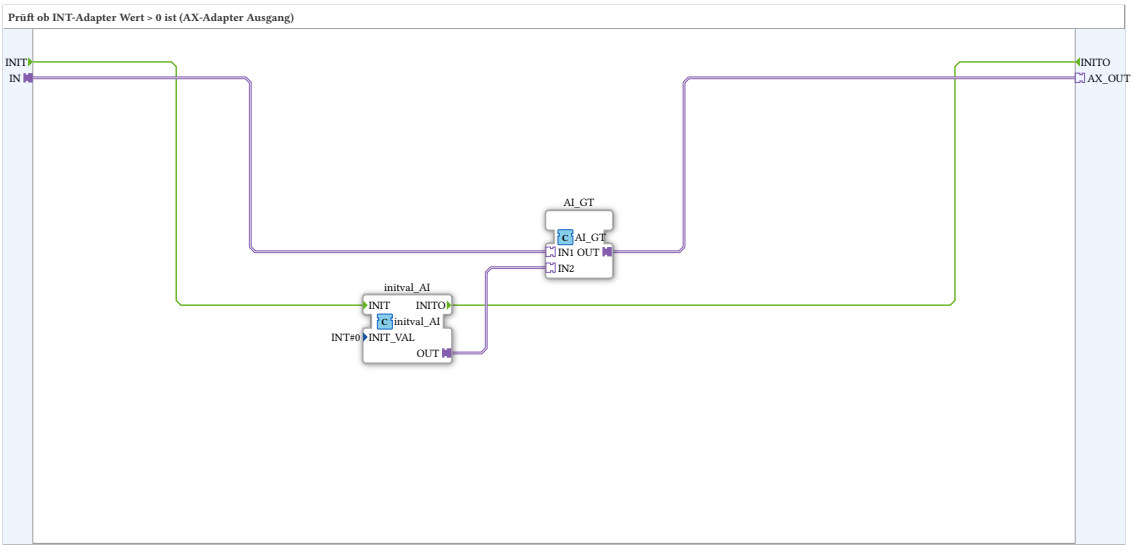
Comparison with Similar Blocks

- **Standard Toggle Flip-Flop (`E_T`):** A simple toggle changes state on every event but has no interlock capability. Multiple toggles can be active simultaneously.
- **Independent SR Latches:** Without a chain, each latch would need explicit external logic to enforce mutual exclusion. This subapp encapsulates the necessary interlock in a reusable and scalable manner.
- **Centralised Arbiter:** A central arbiter could be used, but it would require additional wiring and a different communication pattern. `AE2_ILOCK_T_FF_TO_AX` provides a decentralised solution with a simple daisy-chain topology.

Conclusion

`AE2_ILOCK_T_FF_TO_AX` is a compact and reusable subapplication that provides toggling behaviour with guaranteed mutual exclusion. Its chain-based design automatically propagates resets, making it suitable for any number of participants without additional logic. The use of standard AE2 adapters ensures easy integration into IEC 61499 systems, and the AX output simplifies connection to ordinary boolean consumers. This block is ideal for scenarios where a single active state among many must be maintained with minimal wiring and high reliability.

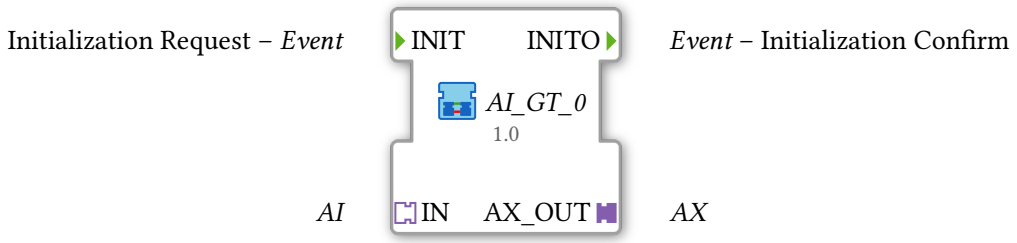
AI_GT_0



• – ○ • – ○ • – ○ •

Introduction

`AI_GT_0` checks whether a INT adapter value is greater than 0 and outputs the result as a boolean AX adapter signal. A typical input is a signed 16-bit value. The block is built identically for every numeric type (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Interface

- **Event input**: `INIT` (initialization)
- **Event output**: `INITO` (initialization finished, comes from the internal `initval_AI`)
- **Socket**: `IN` (`AI`) — the value compared against 0
- **Plug**: `AX_OUT` (`AX`) — TRUE when `IN > 0`

Function Blocks (FBs) Used

Sub-blocks: AI_GT_0

- **Type**: SubAppType
- **Internal FBs used**:
  - **initval_AI**: `adapter::types::unidirectional::AI::initval::initval_AI` — supplies the fixed comparison constant `INT#0` as an adapter value (`INIT_VAL=INT#0`). An adapter socket cannot take the constant as a parameter, so it is fed in through this block.
  - **AI_GT**: `adapter::iec61131::comparison::AI_GT` — adapter-native comparison `IN1 > IN2`.
- **Functionality**: `IN` ⟶ `AI_GT.IN1`; constant 0 ⟶ `AI_GT.IN2`; result ⟶ `AX_OUT`.

Program Flow and Connections

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

Application Scenarios

- Converting a signed 16-bit value into a simple boolean adapter signal for VT status displays or enable logic.

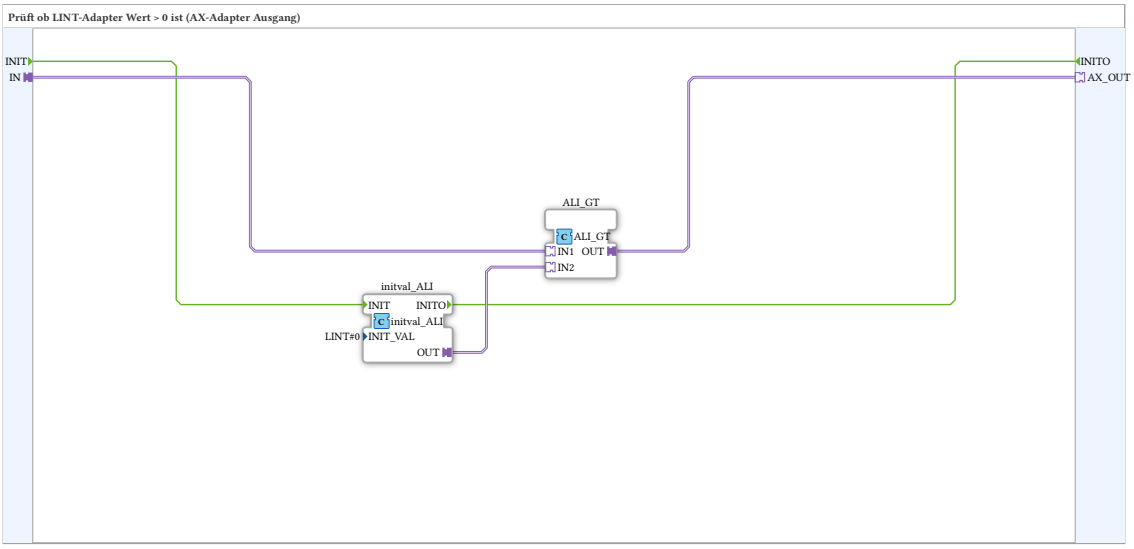
Summary

`AI_GT_0` is a compact wrapper around the adapter-native comparison `AI_GT` — converts "INT > 0" directly into an AX adapter signal, so the caller does not have to create and wire the zero constant itself.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

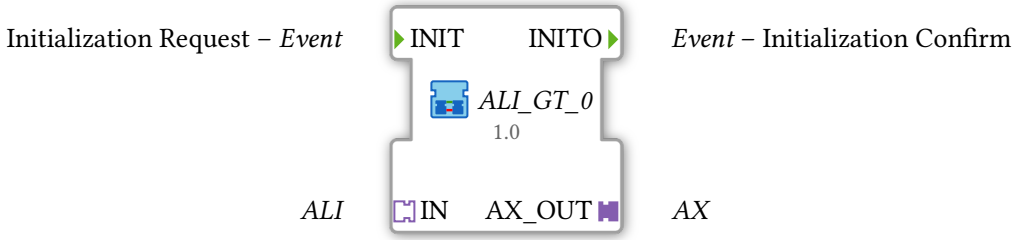
ALI_GT_0



• – ○ • – ○ • – ○ •

Introduction

`ALI_GT_0` checks whether a LINT adapter value is greater than 0 and outputs the result as a boolean AX adapter signal. A typical input is a signed 64-bit value. The block is built identically for every numeric type (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Interface

- **Event input**: `INIT` (initialization)
- **Event output**: `INITO` (initialization finished, comes from the internal `initval_ALI`)
- **Socket**: `IN` (`ALI`) — the value compared against 0
- **Plug**: `AX_OUT` (`AX`) — TRUE when `IN > 0`

Function Blocks (FBs) Used

Sub-blocks: ALI_GT_0

- **Type**: SubAppType
- **Internal FBs used**:

  - **initval_ALI**: `adapter::types::unidirectional::ALI::initval::initval_ALI` — supplies the fixed comparison constant `LINT#0` as an adapter value (`INIT_VAL=LINT#0`). An adapter socket cannot take the constant as a parameter, so it is fed in through this block.
  - **ALI_GT**: `adapter::iec61131::comparison::ALI_GT` — adapter-native comparison `IN1 > IN2`.
- **Functionality**: `IN` ⟶ `ALI_GT.IN1`; constant 0 ⟶ `ALI_GT.IN2`; result ⟶ `AX_OUT`.

Program Flow and Connections

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

Application Scenarios

- Converting a signed 64-bit value into a simple boolean adapter signal for VT status displays or enable logic.

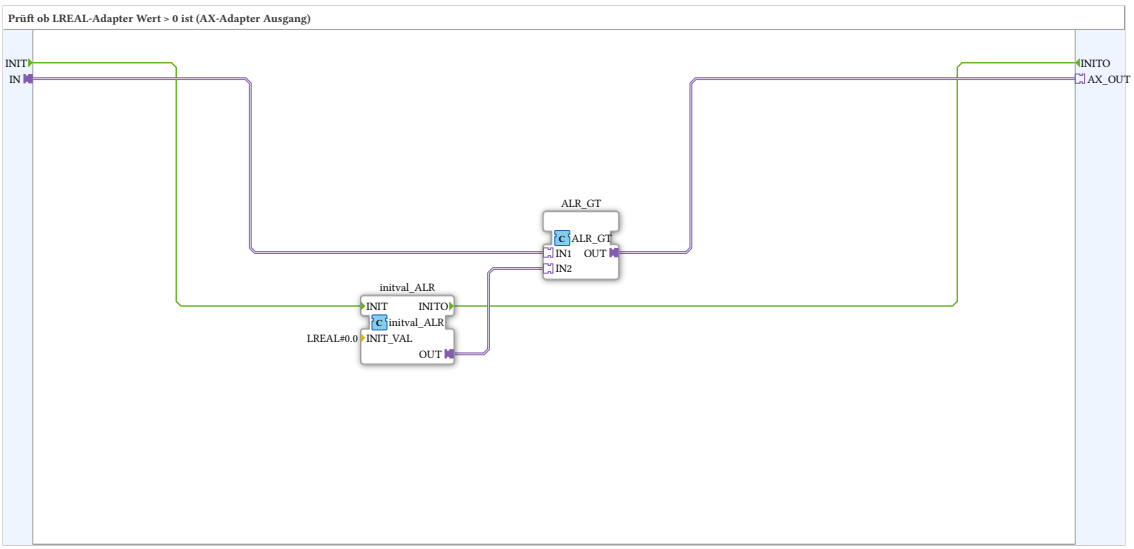
Summary

`ALI_GT_0` is a compact wrapper around the adapter-native comparison `ALI_GT` — converts "LINT > 0" directly into an AX adapter signal, so the caller does not have to create and wire the zero constant itself.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

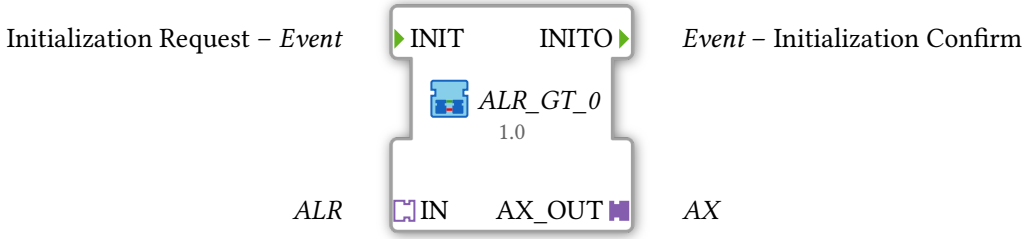
ALR_GT_0



• – ○ • – ○ • – ○ •

Introduction

`ALR_GT_0` checks whether a LREAL adapter value is greater than 0 and outputs the result as a boolean AX adapter signal. A typical input is a floating-point value (LREAL). The block is built identically for every numeric type (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Interface

- **Event input**: `INIT` (initialization)
- **Event output**: `INITO` (initialization finished, comes from the internal `initval_ALR`)
- **Socket**: `IN` (`ALR`) — the value compared against 0
- **Plug**: `AX_OUT` (`AX`) — TRUE when `IN > 0`

Function Blocks (FBs) Used

Sub-blocks: ALR_GT_0

- **Type**: SubAppType
- **Internal FBs used**:
  - **initval_ALR**: `adapter::types::unidirectional::ALR::initval::initval_ALR` — supplies the fixed comparison constant `LREAL#0.0` as an adapter value (`INIT_VAL=LREAL#0.0`). An adapter socket cannot take the constant as a parameter, so it is fed in through this block.
  - **ALR_GT**: `adapter::iec61131::comparison::ALR_GT` — adapter-native comparison `IN1 > IN2`.
- **Functionality**: `IN` ⟶ `ALR_GT.IN1`; constant 0 ⟶ `ALR_GT.IN2`; result ⟶ `AX_OUT`.

Program Flow and Connections

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

Application Scenarios

- Converting a floating-point value (LREAL) into a simple boolean adapter signal for VT status displays or enable logic.

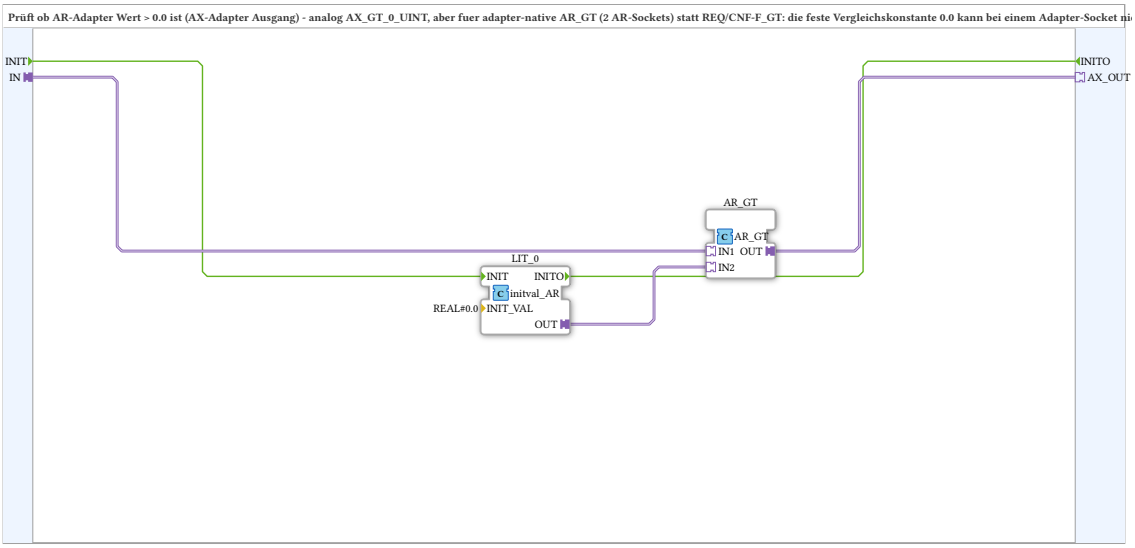
Summary

`ALR_GT_0` is a compact wrapper around the adapter-native comparison `ALR_GT` — converts "LREAL > 0" directly into an AX adapter signal, so the caller does not have to create and wire the zero constant itself.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

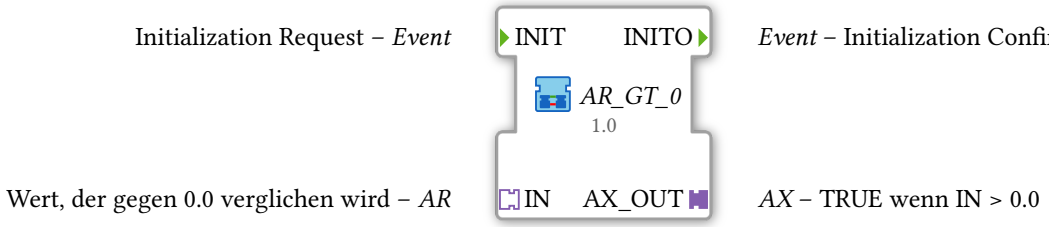
AR_GT_0



• – ◦ • – ◦ • – ◦ •

Introduction

AR_GT_0 checks whether a REAL adapter value is greater than 0 and outputs the result as a boolean AX adapter signal. A typical input is a floating-point value (REAL), e.g. a speed. The block is built identically for every numeric type (AS, AI, ADI, ALI, AUS, AUI, AUDI, AULI, AR, ALR).



Interface

- **Event input**: INIT (initialization)
- **Event output**: INITO (initialization finished, comes from the internal LIT_0)
- **Socket**: IN (AR) — the value compared against 0
- **Plug**: AX_OUT (AX) — TRUE when IN > 0

Function Blocks (FBs) Used

Sub-blocks: AR_GT_0

- **Type**: SubAppType
- **Internal FBs used**:
  - **LIT_0**: adapter::types::unidirectional::AR::initval::initval_AR — supplies the fixed comparison constant 0.0 as an adapter value (INIT_VAL=0.0). An adapter socket cannot take the constant as a parameter, so it is fed in through this block.
  - **AR_GT**: adapter::iec61131::comparison::AR_GT — adapter-native comparison IN1 > IN2.
- **Functionality**: IN ⟶ AR_GT.IN1; constant 0 ⟶ AR_GT.IN2; result ⟶ AX_OUT.

Program Flow and Connections

1. INIT ⟶ LIT_0.INIT; LIT_0.INITO ⟶ INITO.
2. IN (adapter) ⟶ AR_GT.IN1.
3. LIT_0.OUT (constant 0.0) ⟶ AR_GT.IN2.
4. AR_GT.OUT ⟶ AX_OUT (adapter).

Application Scenarios

- Converting a floating-point value (REAL), e.g. a speed into a simple boolean adapter signal for VT status displays or enable logic.

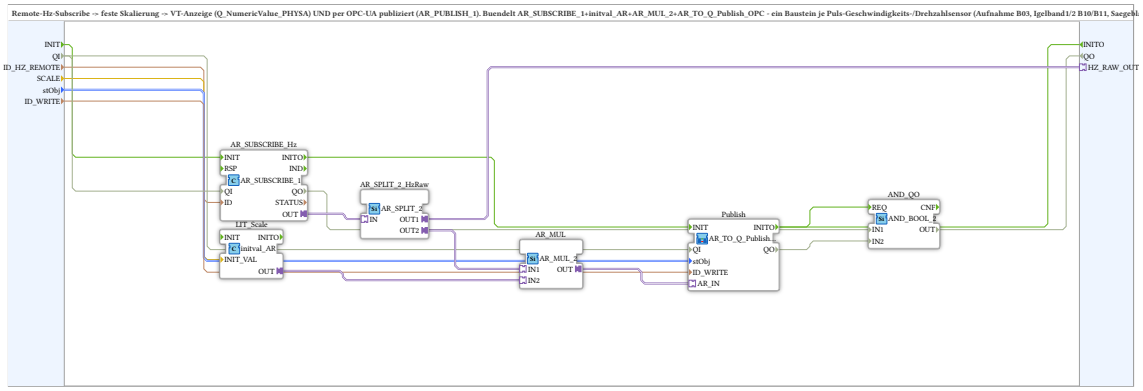
Summary

AR_GT_0 is a compact wrapper around the adapter-native comparison AR_GT — converts "REAL > 0" directly into an AX adapter signal, so the caller does not have to create and wire the zero constant itself.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

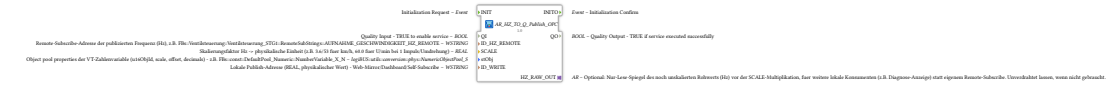
AR_HZ_TO_Q_Publish_OPC



• – ○ • – ○ • – ○ •

Introduction

`AR_HZ_TO_Q_Publish_OPC` bundles the chain **remote subscribe of a frequency (Hz) → fixed scaling → VT display (`Q_NumericValue_PHYSA`) and OPC-UA publish** into one reusable SubApp. It combines `AR_SUBSCRIBE_1`, `initval_AR`, `AR_MUL_2` and `AR_TO_Q_Publish_OPC` - one block per pulse speed/rotation sensor (pickup B03, hedgehog belt 1/2 B10/B11, saw blade B08), only the parameters differ. It was extracted from `Inputs_STG1.SUB`, where four nearly identical chains existed.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the initialization chain (`AR_SUBSCRIBE_Hz` → `Publish`)

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `Publish.INITO` after the chain has completed

Data Inputs

- **QI** (`BOOL`): quality input, service enable (initial value `TRUE`), passed on to `AR_SUBSCRIBE_Hz.QI` and `Publish.QI`
- **ID_HZ_REMOTE** (`WSTRING`): Remote subscribe address of the published frequency (Hz), e.g. `FBs::Ventilsteuerung::Venti`
- **SCALE** (`REAL`): Scale factor Hz → physical unit (e.g. `3.6/53` for km/h, `60.0` for rpm at 1 pulse/revolution)
- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): Object pool properties of the VT numeric variable (`u16ObjId`, `scale`, `offset`, `decimals`), e.g. `FBs::const::DefaultPool_Numeric::NumberVariable_X_N`
- **ID_WRITE** (`WSTRING`): Local publish address (REAL, physical value) for web mirror/dashboard/self-subscribe

Data Outputs

- **QO** (`BOOL`): quality output, `AND` of the `QO` outputs of `AR_SUBSCRIBE_Hz` and `Publish` (via `AND_QO`)

Adapters

- **HZ_RAW_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AR` - optional read-only mirror of the still unscaled raw value (Hz) before the `SCALE` multiplication. Leave unconnected if not needed.

Function Blocks (FBs) Used

- **AR_SUBSCRIBE_Hz**: `adapter::net::AR_SUBSCRIBE_1` (`QI` comes from the SubApp interface), subscribes to the remote address `ID_HZ_REMOTE`
- **LIT_Scale**: `adapter::types::unidirectional::AR::initval::initval_AR`, provides `SCALE` as an AR value

- **AR_SPLIT_2_HzRaw**: `adapter::events::unidirectional::AR_SPLIT_2`, branches the raw value
- **AR_MUL**: `adapter::iec61131::arithmetic::AR_MUL_2`, multiplies the raw value by `SCALE`
- **Publish**: SubApp `MyLib::sys::AR_TO_Q_Publish_OPC`, shows the value on the VT (`stObj`) and publishes it via OPC-UA (`ID_WRITE`)
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2`, combines the `QO` outputs of `AR_SUBSCRIBE_Hz` and `Publish` into the SubApp output `QO`

Program Flow and Connections

1. `ID_HZ_REMOTE` ⟶ `AR_SUBSCRIBE_Hz.ID`; the output `AR_SUBSCRIBE_Hz.OUT` leads to `AR_SPLIT_2_HzRaw.IN`.
2. `AR_SPLIT_2_HzRaw.OUT1` ⟶ plug `HZ_RAW_OUT` (unscaled raw value, optional).
3. `AR_SPLIT_2_HzRaw.OUT2` ⟶ `AR_MUL.IN1`; `SCALE` ⟶ `LIT_Scale.INIT_VAL`, `LIT_Scale.OUT` ⟶ `AR_MUL.IN2`.
4. `AR_MUL.OUT` ⟶ `Publish.AR_IN`.
5. `stObj` ⟶ `Publish.stObj`, `ID_WRITE` ⟶ `Publish.ID_WRITE`.
6. Initialization: `INIT` ⟶ `AR_SUBSCRIBE_Hz.INIT`; `AR_SUBSCRIBE_Hz.INITO` ⟶ `Publish.INIT`; `Publish.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
7. Quality: `QI` ⟶ `AR_SUBSCRIBE_Hz.QI` and `Publish.QI`; `AR_SUBSCRIBE_Hz.QO` ⟶ `AND_QO.IN1`, `Publish.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Technical Details

- **`HZ_RAW_OUT` (version 1.1)**: The optional plug mirrors the raw value (Hz) for a diagnostic display without needing a second, independent remote subscribe of the same node (FORTE reports `Cannot add another one` on a duplicate self-subscribe).
- **INIT/INITO and QI/QO**: initialization is passed through the chain `INIT` ⟶ `AR_SUBSCRIBE_Hz` ⟶ `Publish` ⟶ `INITO`; `QO` is the `AND` of the `QO` outputs of both blocks.
- Pure wiring of existing blocks; apart from the `AND_QO` combination, no logic of its own.

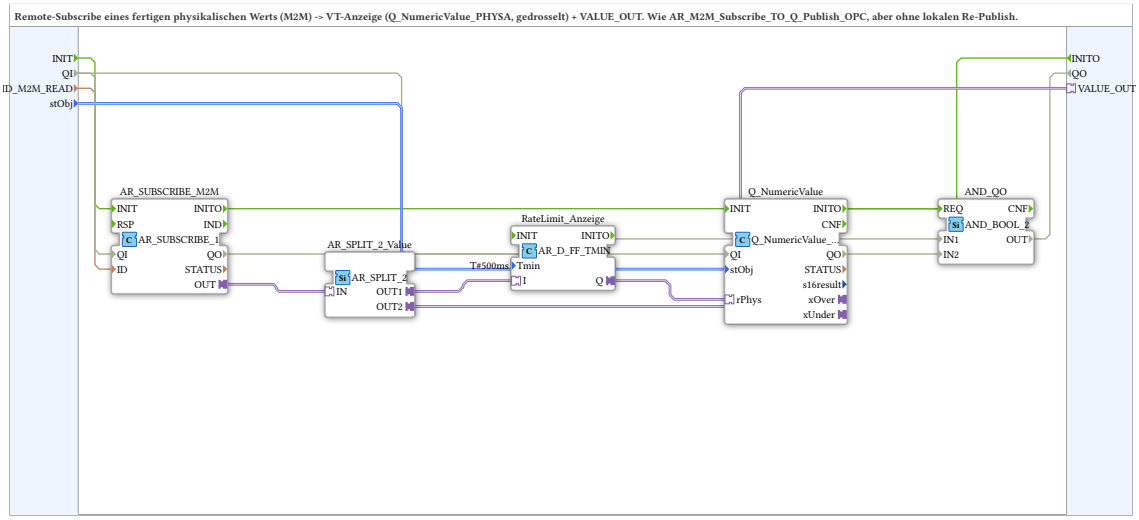
Application Scenarios

- Displaying and publishing speed or rotation from a pulse sensor frequency value (e.g. pickup, hedgehog belts, saw blade)

Summary

`AR_HZ_TO_Q_Publish_OPC` replaces the fourfold duplicated chain of subscribe, scaling and publish with a single parameterizable block.

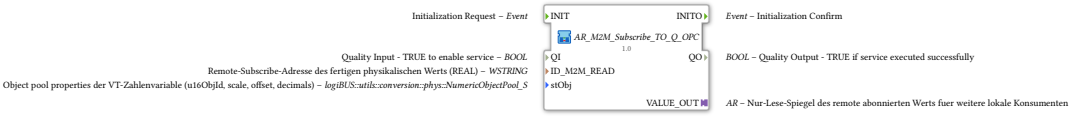
AR_M2M_Subscribe_TO_Q_OPC



• – ○ • – ○ • – ○ •

Introduction

`AR_M2M_Subscribe_TO_Q_OPC` subscribes remotely (M2M) to a finished physical value (`REAL`) and shows it on the VT (`Q_NumericValue_PHYSA`, throttled to at most one update per 500 ms). The value is also available as a read-only mirror on the plug `VALUE_OUT` for further local consumers. The block corresponds to `AR_M2M_Subscribe_TO_Q_Publish_OPC`, but without local re-publish (no `ID_WRITE`) - the lean library variant.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the chain `AR_SUBSCRIBE_M2M` ⟶ `Q_NumericValue`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `Q_NumericValue.INITO`

Data Inputs

Variable	Type	Default	Comment
`QI`	BOOL	TRUE	Quality input: enables the service; passed on to `AR_SUBSCRIBE_M2M` and `Q_NumericValue`.
`ID_M2M_READ`	WSTRING		Remote subscribe address of the finished physical value (`REAL`).
`stObj`	`logiBUS::utils::conversion::phys::NumericObjectPool_S`	`(u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)`	Object pool properties of the VT number variable (`u16ObjId`, scale, offset, decimals).

Data Outputs

Variable	Type	Default	Comment
`QO`	BOOL		Quality output: AND of `AR_SUBSCRIBE_M2M.QO` and `Q_NumericValue.QO`.

Adapters

Adapter	Type	Direction	Comment
`VALUE_OUT`	`adapter::types::unidirectional::AR`	Plug	Read-only mirror of the remotely subscribed value for further local consumers (unthrottled).

Function Blocks (FBs) Used

- **AR_SUBSCRIBE_M2M** (`adapter::net::AR_SUBSCRIBE_1`): remote subscribe of the value
- **AR_SPLIT_2_Value** (`adapter::events::unidirectional::AR_SPLIT_2`): splits the value into the display branch and `VALUE_OUT`
- **RateLimit_Anzeige** (`adapter::events::unidirectional::AR_D_FF_TMIN`, `Tmin = T#500ms`): throttles only the VT display to at most one update per `Tmin`
- **Q_NumericValue** (`isobus::UT::Q::Q_NumericValue_PHYSA`): writes the value to the VT number variable
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): combines the two `QO` outputs

Program Flow and Connections

1. Initialization: `INIT` -> `AR_SUBSCRIBE_M2M.INIT`; `AR_SUBSCRIBE_M2M.INITO` -> `Q_NumericValue.INIT`; `Q_NumericValue.INITO` -> `INITO` and -> `AND_QO.REQ`.
2. Data: `ID_M2M_READ` -> `AR_SUBSCRIBE_M2M.ID`; `stObj` -> `Q_NumericValue.stObj`.
3. Quality: `QI` -> `AR_SUBSCRIBE_M2M.QI` and `Q_NumericValue.QI`; `AR_SUBSCRIBE_M2M.QO` -> `AND_QO.IN1`; `Q_NumericValue.QO` -> `AND_QO.IN2`; `AND_QO.OUT` -> `QO`.
4. Value path: `AR_SUBSCRIBE_M2M.OUT` -> `AR_SPLIT_2_Value.IN`.
5. Display branch: `AR_SPLIT_2_Value.OUT1` -> `RateLimit_Anzeige.I`; `RateLimit_Anzeige.Q` -> `Q_NumericValue.rPhys`.
6. Mirror branch: `AR_SPLIT_2_Value.OUT2` -> `VALUE_OUT`.

Technical Features

- The throttling (`T#500ms`) only acts on the display branch; `VALUE_OUT` remains unthrottled as in the original.
- Compared with `AR_M2M_Subscribe_TO_Q_Publish_OPC` the local publish and `ID_WRITE` are omitted.

Application Scenarios

- Displaying an already physically scaled value delivered by another station on the VT when no local re-publish is needed.
- Passing the subscribed value on via `VALUE_OUT` to further local consumers.

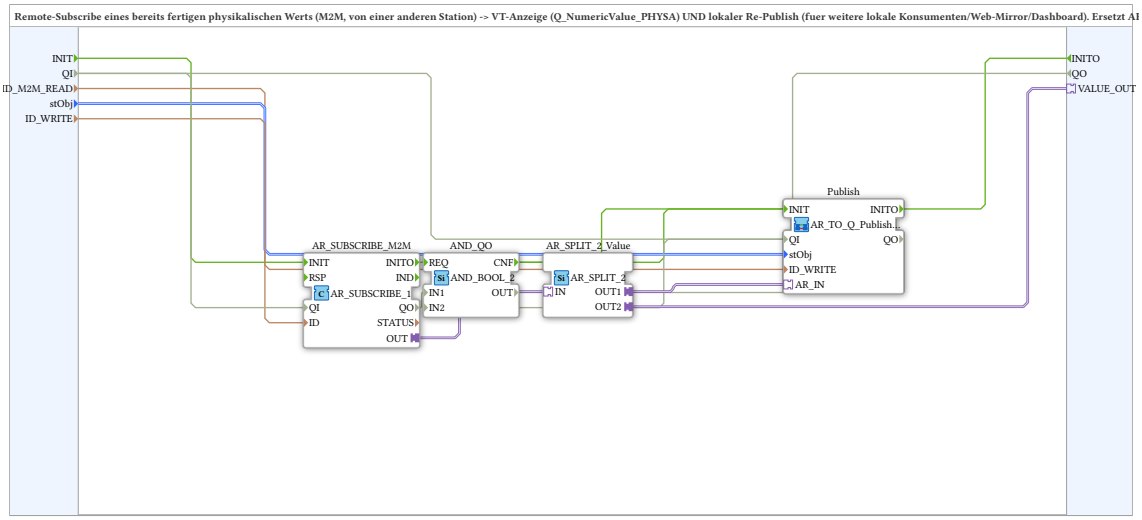
Summary

`AR_M2M_Subscribe_TO_Q_OPC` subscribes to a finished physical value remotely, shows it throttled on the VT and additionally provides it via `VALUE_OUT` - without local re-publish.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AR_M2M_Subscribe_TO_Q_Publish_OPC



• – ○ • – ○ • – ○ •

Introduction

`AR_M2M_Subscribe_TO_Q_Publish_OPC` subscribes to an already converted physical value (REAL) from another station via OPC-UA (M2M), shows it on the VT (`Q_NumericValue_PHYSA`) and republishes it locally so that further local consumers, web mirror or dashboard can read it. In addition, the value is available at the plug `VALUE_OUT` for local adapter consumers. The block is the lean successor of `AR_HZ_TO_Q_Publish_OPC` without its own scaling, because the Hz-to-unit conversion now runs on the sensor station (see `logiBUS_PI_Speed_Publish_OPC`).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; passed on to `AR_SUBSCRIBE_M2M.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `Publish.INITO`

Data Inputs

- **QI** (BOOL, initial value `TRUE`): quality input, enables the service; passed on to `AR_SUBSCRIBE_M2M.QI` and `Publish.QI`
- **ID_M2M_READ** (WSTRING): remote subscribe address of the finished physical value (REAL), e.g. `FBs::Ventilsteuerung::`
- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): object pool properties of the VT number variable (`u16ObjId`, scale, offset, decimals), e.g. `FBs::const::DefaultPool_Numeric::NumberVariable_X_N`
- **ID_WRITE** (WSTRING): local publish address (REAL, physical value) for web mirror, dashboard and self-subscribe of further local consumers

Data Outputs

- **QO** (BOOL): quality output, AND combination (`AND_QO`) of the `QO` outputs of `AR_SUBSCRIBE_M2M` and `Publish`

Adapters

- **VALUE_OUT** (plug, `adapter::types::unidirectional::AR`): read-only mirror of the remotely subscribed value for further local consumers; avoids a duplicate OPC-UA self-subscribe of the same remote node on this station

Functionality

The network consists of four parts:

- `AR_SUBSCRIBE_M2M` (`adapter::net::AR_SUBSCRIBE_1`, `QI` comes from the input `QI`): subscribes to the address `ID_M2M_READ` (`ID_M2M_READ` ⟶ `AR_SUBSCRIBE_M2M.ID`) and outputs the value as an AR adapter.
- `AR_SPLIT_2_Value` (`adapter::events::unidirectional::AR_SPLIT_2`): distributes the subscribed value to two destinations (`AR_SUBSCRIBE_M2M.OUT` ⟶ `AR_SPLIT_2_Value.IN`).
- `Publish` (`MyLib::sys::AR_TO_Q_Publish_OPC`): displays the value on the VT and publishes it locally (`AR_SPLIT_2_Value.OUT1` ⟶ `Publish.AR_IN`, `stObj` ⟶ `Publish.stObj`, `ID_WRITE` ⟶ `Publish.ID_WRITE`).
- `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): combines `AR_SUBSCRIBE_M2M.QO` and `Publish.QO` into the output `QO`.

`AR_SPLIT_2_Value.OUT2` is routed to the plug `VALUE_OUT`.

Initialization and quality: `INIT` ⟶ `AR_SUBSCRIBE_M2M.INIT`, `AR_SUBSCRIBE_M2M.INITO` ⟶ `Publish.INIT`, `Publish.INITO` ⟶ `INITO` and `AND_QO.REQ`. `QI` is passed on to `AR_SUBSCRIBE_M2M.QI` and `Publish.QI`; `AR_SUBSCRIBE_M2M.QO` ⟶ `AND_QO.IN1`, `Publish.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`.

Technical Details

- No more `SCALE`/`AR_MUL_2`: the subscribed value is already in the physical unit
- The `AR_SPLIT_2` in front of `Publish.AR_IN` was inserted instead of wiring `AR_SUBSCRIBE_M2M.OUT` directly, so that an adapter is never branched without a split
- `VALUE_OUT` supplies the subscribed value to consumers such as automatic filling or filling belt detection without creating a second subscribe on the same remote node

Application Scenarios

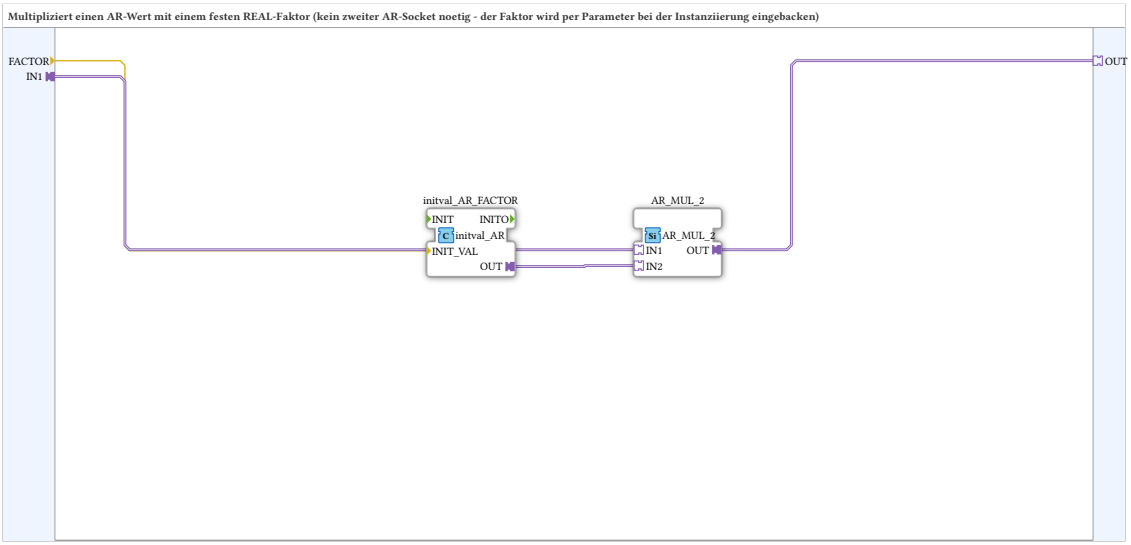
- Displaying a value measured by another station (e.g. speed) on the VT while making it available locally to dashboard and web client

- Passing the remote value on to local control or automation blocks via `VALUE_OUT`

Conclusion

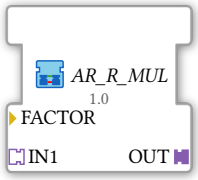
The block bundles remote subscribe, VT display and local republish of a finished physical value into one SubApp and additionally provides the value as an AR plug.

AR_R_MUL



Fester Multiplikator, ueber Parameter bei der Instanziierung gesetzt – *REAL*

Zu skalierender Wert – *AR*



AR – IN1 x FACTOR

• – ◦ • – ◦ • – ◦ •

Introduction

The **AR_R_MUL** function block is a subapplication designed to multiply an AR (Analog/Real) value by a fixed REAL factor. Unlike a generic multiplier that requires two AR inputs, this block uses a constant factor that is set once during instantiation via the `FACTOR` parameter. This simplifies usage in scenarios where the multiplication factor is known ahead of time and does not need to be changed dynamically.

Interface Structure

The subapplication provides the following interface elements:

Event Inputs

None.

Event Outputs

None.

Data Inputs

Name	Type	Initial Value	Description
`FACTOR`	REAL	`REAL#1.0`	Fixed multiplier applied to the input AR value. This value is set as a parameter when the subapplication is instantiated.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket	IN1	adapter::types::unidirectional::AR	AR value to be scaled.
Plug	OUT	adapter::types::unidirectional::AR	Result of IN1 × FACTOR.

Functionality

The subapplication multiplies the AR value received at the IN1 socket by the constant FACTOR and provides the result at the OUT plug. Internally, the FACTOR data input is converted into a constant AR value using an initval_AR_FACTOR function block. This constant AR value is then used as the second operand in an AR_MUL_2 multiplication block, together with the incoming IN1 AR value. The product is forwarded to the output.

Since the factor is embedded as a parameter, the subapplication behaves like a fixed-gain amplifier for AR signals.

Technical Features

- **Parameter-driven constant**: The multiplication factor is defined once at design time via the FACTOR input, eliminating the need for a second dynamic AR source.
- **Reusability**: The subapplication encapsulates the initialization and multiplication logic, making the overall function easy to instantiate and maintain.
- **Type uniformity**: Both operands are AR type, ensuring compatibility with standard AR signal processing chains.
- **No event involvement**: The block operates purely on data and adapter connections; no event inputs or outputs are required.

State Overview

The AR_R_MUL subapplication does not maintain any internal state. It processes data continuously: as soon as an AR value arrives at IN1, it is multiplied by the constant factor and the result is made available at OUT. The only relevant state is the fixed FACTOR value, which is set during instantiation and remains unchanged during runtime.

Application Scenarios

- **Sensor scaling**: Convert a raw analog measurement (e.g., 0–10 V) to an engineering unit by applying a fixed scale factor.
- **Unit conversion**: Multiply AR values by a constant to switch between units (e.g., meters to feet).
- **Signal conditioning**: Apply a constant gain to an AR signal for amplification or attenuation.
- **Static calibration**: Incorporate a fixed offset or gain correction in an existing AR processing chain.

Comparison with Similar Blocks

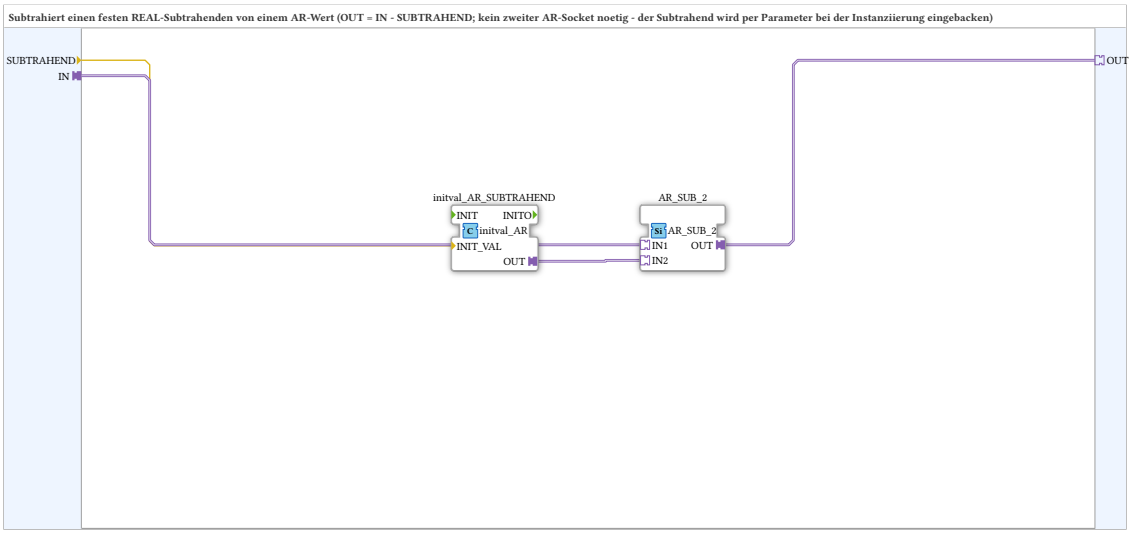
Block	Description	Key Difference
AR_R_MUL	Multiplies an AR value by a fixed REAL factor.	Only one AR input required; factor is static.
AR_MUL_2	Multiplies two AR values dynamically.	Requires two AR inputs; factor can vary at runtime.
AR_ADD_2	Adds two AR values.	Performs addition instead of multiplication.

The main advantage of AR_R_MUL over AR_MUL_2 is its simplicity when the multiplier is a constant – it saves the effort of creating and maintaining a separate constant source, and reduces wiring complexity.

Conclusion

AR_R_MUL provides a clean and efficient way to apply a fixed scaling factor to an AR signal. By embedding the factor as a parameter in the instantiation, it simplifies the design of control and measurement applications where a constant gain is required. Its straightforward interface and lack of event handling make it easy to integrate into larger 4diac systems.

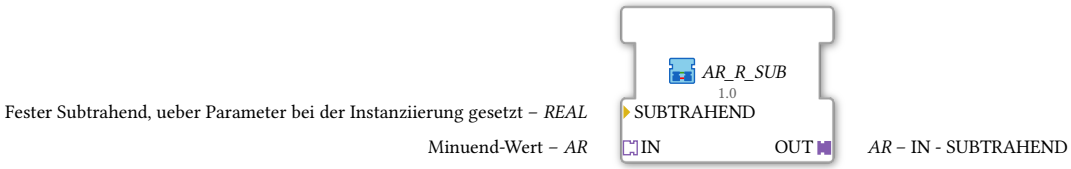
AR_R_SUB



• – ◦ • – ◦ • – ◦ •

Introduction

`AR_R_SUB` is a SubApp that subtracts a fixed `REAL` subtrahend from an `AR` value (`OUT = IN - SUBTRAHEND`). The subtrahend is baked in as a parameter at instantiation, so no second `AR` socket has to be wired. It is the subtraction counterpart of `AR_R_MUL`.



Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

- **SUBTRAHEND** (`REAL`): Fixed subtrahend, set as a parameter at instantiation. Initial value `REAL#0.0`.

Data Outputs

None.

Adapters

- **IN** (socket, `adapter::types::unidirectional::AR`): minuend value
- **OUT** (plug, `adapter::types::unidirectional::AR`): `IN - SUBTRAHEND`

Function Blocks (FBs) Used

- **`initval_AR_SUBTRAHEND`** (`adapter::types::unidirectional::AR::initval::initval_AR`): creates a constant `AR` value from `SUBTRAHEND`
- **`AR_SUB_2`** (`adapter::iec61131::arithmetic::AR_SUB_2`): subtracts two `AR` values

Functionality

- `SUBTRAHEND` ⟶ `initval_AR_SUBTRAHEND.INIT_VAL`
- `IN` ⟶ `AR_SUB_2.IN1` (minuend)
- `initval_AR_SUBTRAHEND.OUT` ⟶ `AR_SUB_2.IN2` (subtrahend)

- `AR_SUB_2.OUT` ⟶ `OUT`

The block has no events of its own; the calculation follows the data flow of the `AR` adapter.

Technical Details

- No second `AR` socket needed, the subtrahend is baked into the instance
- No change of the subtrahend at runtime is intended
- Pure wiring of existing blocks, no logic of its own

Application Scenarios

- Subtracting a constant offset from a scaled or physical `AR` value

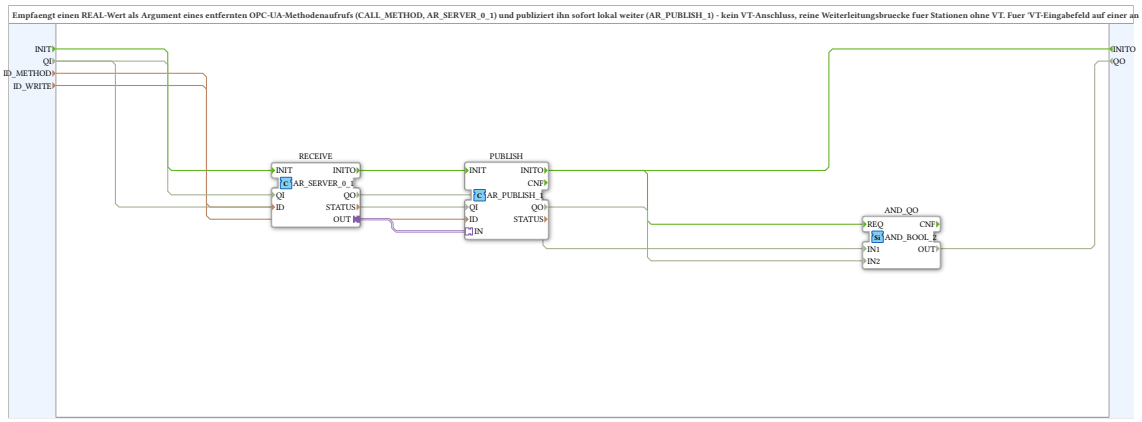
Summary

`AR_R_SUB` subtracts a fixed, parameterised `REAL` value from an `AR` signal.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

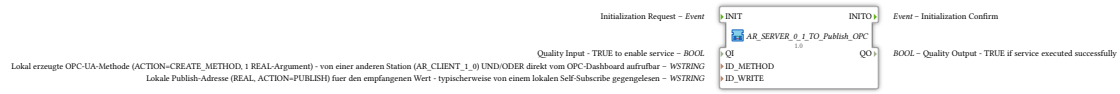
AR_SERVER_0_1_TO_Publish_OPC



• — ○ • — ○ • — ○ •

Introduction

`AR_SERVER_0_1_TO_Publish_OPC` receives a `REAL` value as the argument of a remote OPC-UA method call (`AR_SERVER_0_1`) and immediately publishes it locally (`AR_PUBLISH_1`). The block has no VT connection and is a pure forwarding bridge for stations without a VT: a VT input field on another station or a web dashboard calls the same method - and only on real input. A typical example is pressure reference values (B20).



Function Blocks (FBs) Used

Sub-blocks: AR_SERVER_0_1_TO_Publish_OPC

- **Type**: SubAppType
- **Internal FBs used**:
  - **RECEIVE**: `adapter::net::AR_SERVER_0_1` (`QI` from outside) - creates the OPC-UA method and provides the received `REAL` argument on the adapter `OUT`
  - **PUBLISH**: `adapter::net::AR_PUBLISH_1` (`QI` from outside) - publishes the value under the local address
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` - combines the `QO` of `RECEIVE` and `PUBLISH`
- **Functionality**: The value from the method call is passed directly to the publisher without intermediate logic.

Interface Structure

Event Inputs

- **INIT** (EInit): initialization request; goes to RECEIVE.INIT

Event Outputs

- **INITO** (EInit): initialization confirm; comes from PUBLISH.INITO

Data Inputs

- **QI** (BOOL, default TRUE): Quality Input - TRUE enables the service; goes to RECEIVE.QI and PUBLISH.QI
- **ID_METHOD** (WSTRING): Locally created OPC-UA method (ACTION=CREATE_METHOD, 1 REAL argument) - callable from another station (AR_CLIENT_1_0) and/or directly from the OPC dashboard
- **ID_WRITE** (WSTRING): Local publish address (REAL, ACTION=PUBLISH) for the received value - typically read back by a local self-subscribe

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if the service was executed successfully (AND of RECEIVE.QO and PUBLISH.QO)

Program Flow and Connections

- ID_METHOD ⟶ RECEIVE.ID
- ID_WRITE ⟶ PUBLISH.ID
- INIT ⟶ RECEIVE.INIT
- RECEIVE.INITO ⟶ PUBLISH.INIT (the publisher is initialised as soon as the method has been created)
- PUBLISH.INITO ⟶ INITO and ⟶ AND_QO.REQ
- QI ⟶ RECEIVE.QI and PUBLISH.QI
- RECEIVE.QO ⟶ AND_QO.IN1, PUBLISH.QO ⟶ AND_QO.IN2, AND_QO.OUT ⟶ QO
- RECEIVE.OUT ⟶ PUBLISH.IN (adapter connection of the REAL value)

Technical Details

- No VT connection: intended for sensor stations without their own VT (e.g. STG4) that should nevertheless accept editable reference values from another station (VT) or from the dashboard
- The method is called only on real input, there is no cyclic synchronisation
- Version 1.0 (2026-09-23): bundles AR_SERVER_0_1 (receive) and AR_PUBLISH_1 (local forwarding)

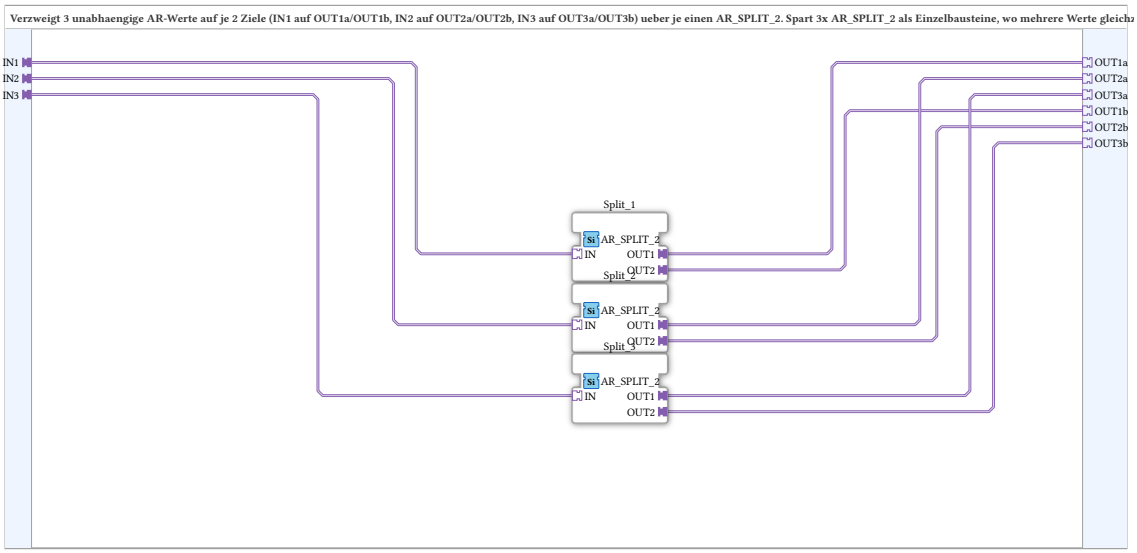
Application Scenarios

- Pressure reference values (B20) of a station without a VT, set by another station or by the web dashboard
- In general: accepting a REAL setpoint via an OPC-UA method call and providing it locally as a publish value

Summary

AR_SERVER_0_1_TO_Publish_OPC encapsulates AR_SERVER_0_1 and AR_PUBLISH_1 into a forwarding bridge that publishes a REAL value received via an OPC-UA method locally.

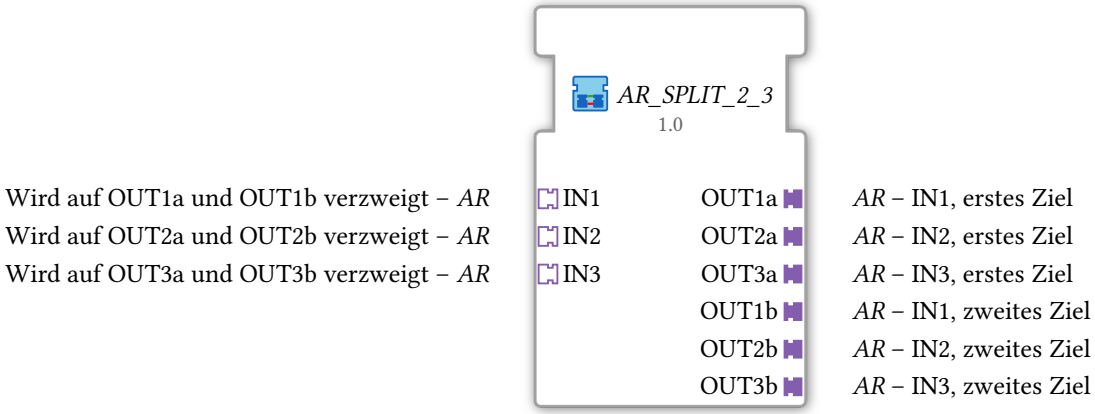
AR_SPLIT_2_3



• – ○ • – ○ • – ○ •

Introduction

`AR_SPLIT_2_3` branches **three independent AR values** onto two targets each (`IN1` ⟶ `OUT1a`/`OUT1b`, `IN2` ⟶ `OUT2a`/`OUT2b`, `IN3` ⟶ `OUT3a`/`OUT3b`) using one `AR_SPLIT_2` each. It saves three individual `AR_SPLIT_2` blocks wherever several values must be distributed simultaneously to two independent consumers (e.g. threshold/dead band/hysteresis to control logic and to position/display logic, as in the automatic filling).



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

All adapters are unidirectional of type `adapter::types::unidirectional::AR`.

- **IN1** (socket): branched to `OUT1a` and `OUT1b`
- **IN2** (socket): branched to `OUT2a` and `OUT2b`
- **IN3** (socket): branched to `OUT3a` and `OUT3b`
- **OUT1a**, **OUT2a**, **OUT3a** (plugs): first target of `IN1`/`IN2`/`IN3` respectively
- **OUT1b**, **OUT2b**, **OUT3b** (plugs): second target of `IN1`/`IN2`/`IN3` respectively

Function Blocks (FBs) Used

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

Program Flow and Connections

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

The three branches are completely independent of each other.

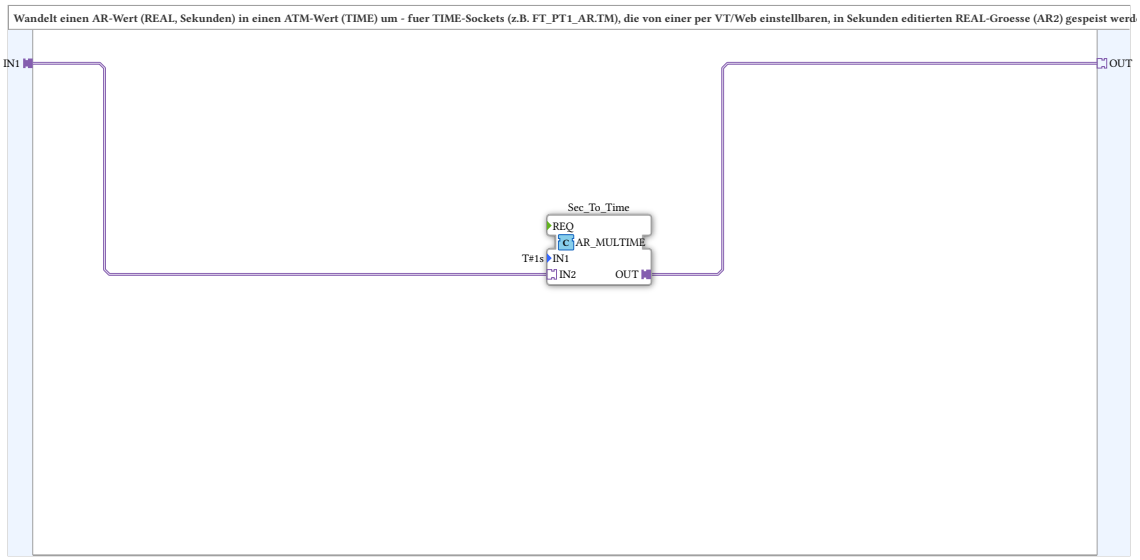
Application Scenarios

- Distributing several parameters (threshold, dead band, hysteresis) to two independent consumers at the same time

Summary

`AR_SPLIT_2_3` bundles three `AR_SPLIT_2` blocks in one SubApp and thus keeps the calling block's network tidy.

AR_S_TO_ATM

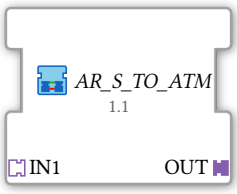


• – ○ • – ○ • – ○ •

Introduction

`AR_S_TO_ATM` converts an `AR` value (REAL, seconds) into an `ATM` value (TIME). It is needed for TIME sockets (e.g. `FT_PT1_AR.TM`) that are to be fed by a REAL quantity (AR2) that is editable via VT or web and edited in seconds.

Zeit in Sekunden (REAL) – *AR*



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

Interface Structure

Event Inputs
The function block has no direct event inputs.

Event Outputs
The function block has no direct event outputs.

Data Inputs
The function block has no direct data inputs.

Data Outputs
The function block has no direct data outputs.

Adapters
- **IN1** (socket, `adapter::types::unidirectional::AR`): time in seconds (REAL)
- **OUT** (plug, `adapter::types::unidirectional::ATM`): `IN1`, converted from seconds (REAL) to TIME

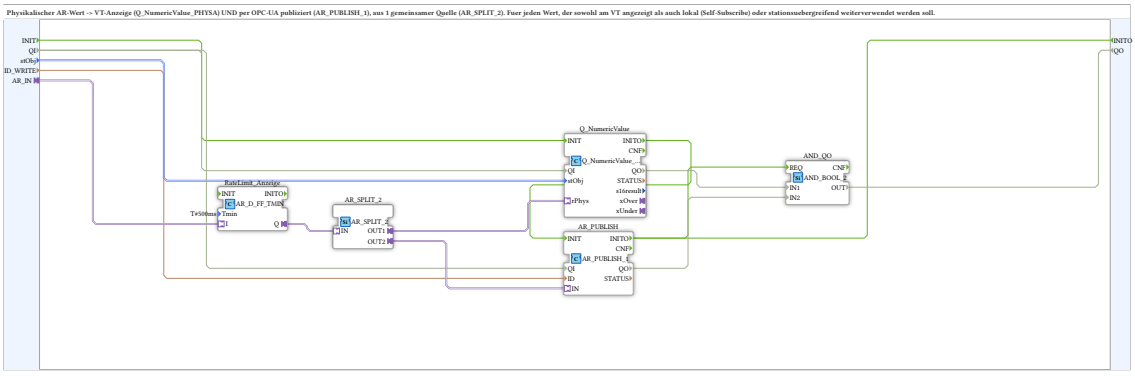
Functionality
Internally a single instance `Sec_To_Time` of type `adapter::iec61131::arithmetic::AR_MULTIME` is used with the parameter `IN1 = T#1s`. The socket `IN1` is routed to `Sec_To_Time.IN2` and the result `Sec_To_Time.OUT` to the plug `OUT`. Thus `OUT = T#1s * IN1`, i.e. true IEC 61131-3 TIME arithmetic: an input of `2.5` yields `T#2s500ms`.

Technical Details
- The earlier variant (`AR_R_MUL` × 1000 ⟶ `AR_TO_ADI` ⟶ `ADI_TO_ATM`) was replaced because it relied on a non-existent function `iec61131::conversion::F_DINT_TO_TIME`
- Stateless; events travel via the adapters (`E1` of `IN1` triggers the calculation)

Application Scenarios
- Adjustable PT1 filter time of the pulse frequency sensors (see `logiBUS_PI_Speed_Publish_OPC`): edited in seconds, routed as TIME to `FT_PT1_AR.TM`
- Any place where a quantity entered in seconds has to be passed to a TIME socket

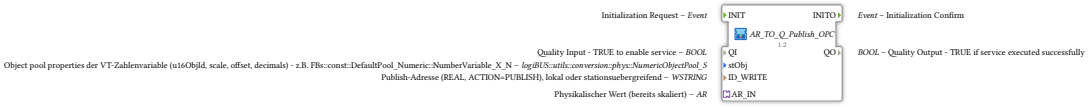
Conclusion
`AR_S_TO_ATM` is a small adapter SubApp that converts second values to TIME using `T#1s * seconds`.

AR_TO_Q_Publish_OPC



• – ○ • – ○ • – ○ •

Introduction
`AR_TO_Q_Publish_OPC` routes a physical `AR` value from **one common source** both to a VT numeric display (`Q_NumericValue_PHYSA`) and to an OPC-UA publish (`AR_PUBLISH_1`). It is intended for values that are shown on the VT and additionally reused locally (self-subscribe) or across stations. Extracted from `TECU_Speeds/TECU_RPTO`, where the combination `AR_SPLIT_2` + `Q_NumericValue_PHYSA` + `AR_PUBLISH_1` occurred frequently.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation request

Event Outputs

- **INITO** (`EInit`): initialisation confirmation

Data Inputs

- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): object pool properties of the VT number variable (`u16ObjId`, scale, offset, decimals), e.g. `FBs::const::DefaultPool_Numeric::NumberVariable_X_N`
- **ID_WRITE** (`WSTRING`): publish address (REAL, `ACTION=PUBLISH`), local or cross-station
- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE to enable the service (VT display and publish)

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully (AND of `Q_NumericValue.QO` and `AR_PUBLISH.QO`)

Adapters

- **AR_IN** (socket, `adapter::types::unidirectional::AR`): physical value (already scaled)

Function Blocks (FBs) Used

- **`RateLimit_Anzeige`** (`adapter::events::unidirectional::AR_D_FF_TMIN`, `Tmin = T#500ms`): limits the update rate
- **`AR_SPLIT_2`** (`adapter::events::unidirectional::AR_SPLIT_2`): distributes the value to two targets
- **`Q_NumericValue`** (`isobus::UT::Q::Q_NumericValue_PHYSA`): writes the physical value into the VT number variable
- **`AR_PUBLISH`** (`adapter::net::AR_PUBLISH_1`): publishes the value via OPC-UA
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): combines the `QO` outputs of `Q_NumericValue` and `AR_PUBLISH` into the `QO` of the SubApp

Functionality

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

Technical Details

- **Rate limiting before the split (intentional):** Since v1.2 `RateLimit_Anzeige` sits in front of `AR_SPLIT_2` and therefore throttles **both** branches (VT display and publish/self-subscribe consumers such as `CTRL_PID_AR.AR_ACT`) to at most one update per 500 ms. In v1.1 the throttle was only in the VT display branch. The motivation was sluggish behaviour with many quickly changing displays at the same time (suspected VT bus load).
- An adapter must not fan out to two targets, hence the explicit `AR_SPLIT_2`.

Application Scenarios

- Speed or other measured values that are shown on the VT and at the same time published locally or to other stations

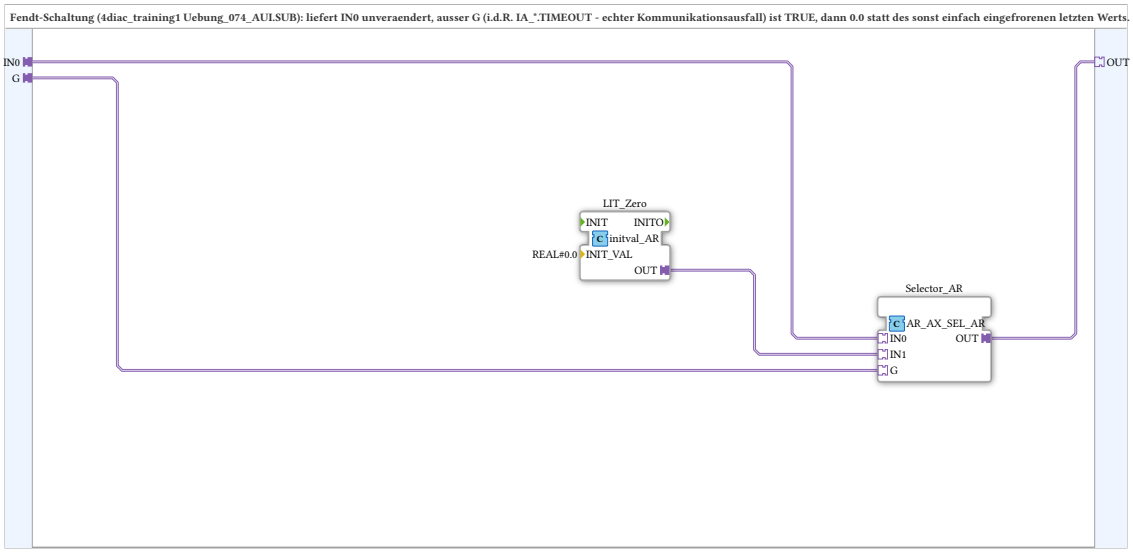
Summary

`AR_TO_Q_Publish_OPC` bundles rate limiting, splitting, VT display and OPC-UA publish of an `AR` value in one SubApp; `INIT`/`INITO` are chained through both services, `QO` is the AND of both services.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AR_Timeout_Zero

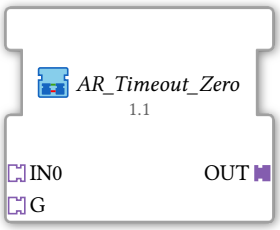


• – ○ • – ○ • – ○ •

Introduction

`AR_Timeout_Zero` passes the `REAL` value `IN0` through unchanged, unless the `AX` signal `G` (usually `IA_*.TIMEOUT`, a real communication failure) is `TRUE`: then `0.0` is output instead of the last value, which would otherwise simply stay frozen. The circuit originates from the Fendt exercise `Uebung_074_AUI`.

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



AR – IN0, oder 0.0 bei G=TRUE

Function Blocks (FBs) Used

Sub-blocks: AR_Timeout_Zero

- **Type**: SubAppType
- **Internal FBs used**:
  - **LIT_Zero**: `adapter::types::unidirectional::AR::initval::initval_AR` with `INIT_VAL=0.0` - provides the constant `0.0`
  - **Selector_AR**: `adapter::iec61131::selection::AR_AX_SEL_AR` - selects between `IN0` and `IN1` depending on `G`
- **Functionality**: With `G=FALSE` the selector outputs the live value `IN0`, with `G=TRUE` the constant `0.0`.

Interface Structure

Event Inputs / Event Outputs

None - the SubApp has no event ports of its own; events travel via the adapters.

Data Inputs / Data Outputs

No direct data ports.

Adapters

- **IN0** (socket): `adapter::types::unidirectional::AR` - live value, e.g. `FIELDBUS_*_SCALED.OUT`
- **G** (socket): `adapter::types::unidirectional::AX` - `TRUE` = timeout, outputs `0.0` instead of `IN0`
- **OUT** (plug): `adapter::types::unidirectional::AR` - `IN0`, or `0.0` if `G=TRUE`

Program Flow and Connections

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

Technical Details

- The block replaces a frequently recurring wiring (`AR_AX_SEL_AR` with `initval_AR=0`) that had previously been built individually in the TECU blocks (`TECU_RPTO`, `TECU_FPTO`, `TECU_COGSOG`, `TECU_GBSD`, `TECU_WBSD`, `TECU_VDS`, `TECU_FHS`, `TECU_RHS`, `TECU_MSS`)
- Version 1.1 (2026-09-16): moved to `MyLib_AX-1.0.0::sys`, as it is generic and not TECU-specific

Application Scenarios

- Forcing a measured value to zero on communication failure (timeout) instead of freezing the last value
- In general: substitute value `0.0` for an `AR` value, controlled by an `AX` signal

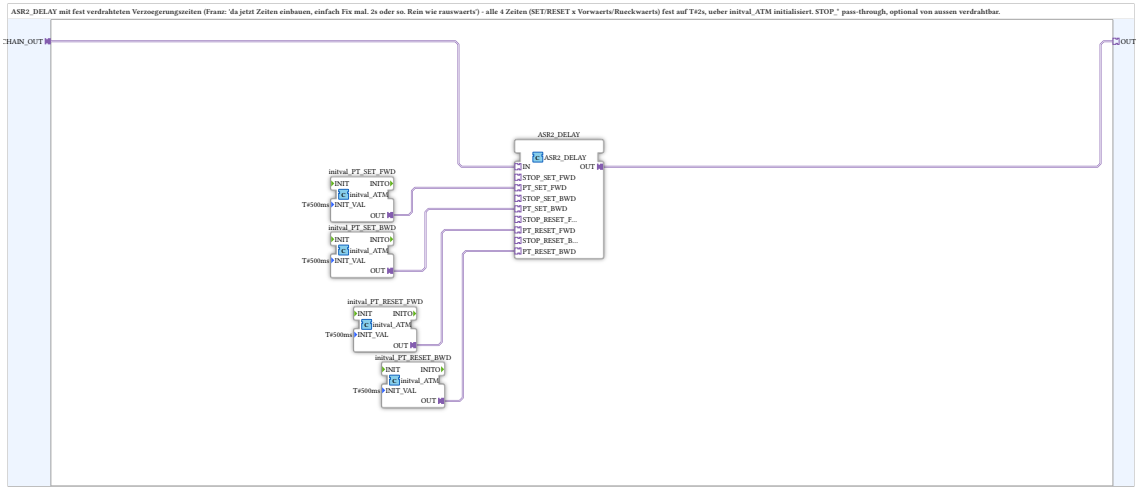
Summary

`AR_Timeout_Zero` encapsulates an `AR` selector with the constant `0.0` into a reusable block that sets the value to zero on timeout.

🛠️ Related Exercises

- Uebung_070d_AUI
- Uebung_079a_AUI

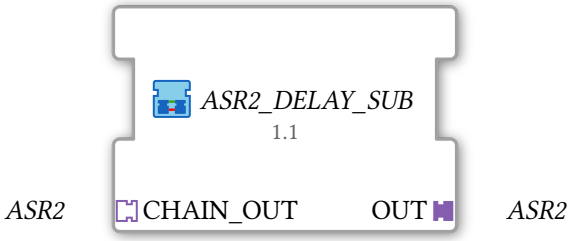
ASR2_DELAY_SUB



• — ○ • — ○ • — ○ •

Introduction

`ASR2_DELAY_SUB` is `ASR2_DELAY` with hard-wired delay times: the four times (set and reset, each forward and backward) are preset through `initval_ATM` instances so that the block can be inserted into an ASR2 chain as a delay element without any time wiring of its own.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **CHAIN_OUT** (socket, `adapter::types::bidirectional::ASR2`): input of the chain; routed to `ASR2_DELAY.IN`
- **OUT** (plug, `adapter::types::bidirectional::ASR2`): delayed chain; comes from `ASR2_DELAY.OUT`

Functionality

The network consists of one `ASR2_DELAY` instance (`adapter::events::bidirectional::ASR2_DELAY`) and four `initval_ATM` instances (`adapter::types::unidirectional::ATM::initval::initval_ATM`):

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

All four `initval_ATM` instances are parameterised with `INIT_VAL = T#500ms`.

Technical Details

- The times are placeholder fixed values without functional requirement and can be made configurable later if needed; the type comment mentions `T#2s` as an example, the instances are currently set to `T#500ms`
- The interface consists only of the input `CHAIN_OUT` and the output `OUT`; the `STOP_*` sockets of `ASR2_DELAY` remain unwired in this SubApp
- Stateless at SubApp level; the state (running timers) resides in `ASR2_DELAY`

Application Scenarios

- Time-staggered control of outputs in a set/reset chain (e.g. belt sequencing) when fixed delays are sufficient

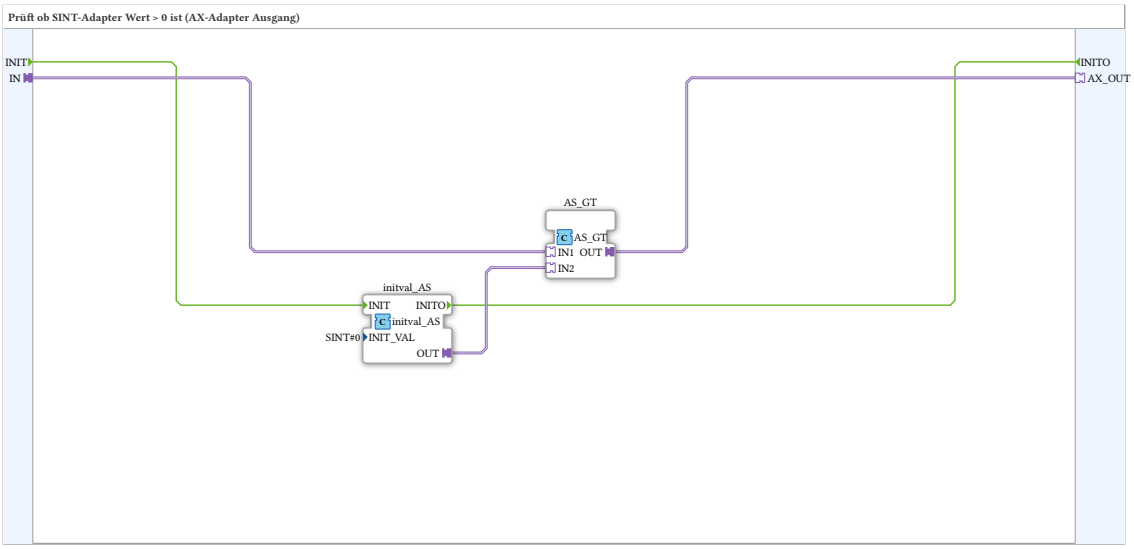
⚖️ Comparison with Similar Blocks

For freely adjustable times, `ASR2_DELAY` is wired directly with your own `ATM` sources; `ASR2_DELAY_SUB`, in contrast, encapsulates the fixed values. For a single event per direction see `AE2_DELAY`.

Conclusion

`ASR2_DELAY_SUB` is a plug-in delay element for ASR2 chains with preset times.

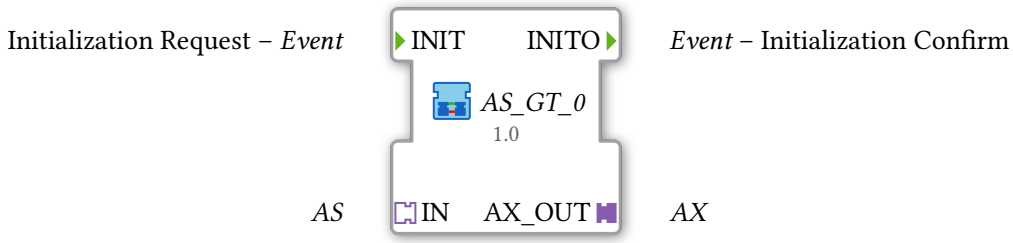
AS_GT_0



• – ○ • – ○ • – ○ •

Introduction

`AS_GT_0` checks whether a SINT adapter value is greater than 0 and outputs the result as a boolean AX adapter signal. A typical input is a signed 8-bit value. The block is built identically for every numeric type (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Interface

- **Event input**: `INIT` (initialization)
- **Event output**: `INITO` (initialization finished, comes from the internal `initval_AS`)
- **Socket**: `IN` (`AS`) — the value compared against 0
- **Plug**: `AX_OUT` (`AX`) — TRUE when `IN > 0`

Function Blocks (FBs) Used

Sub-blocks: AS_GT_0

- **Type**: SubAppType
- **Internal FBs used**:
  - **initval_AS**: `adapter::types::unidirectional::AS::initval::initval_AS` — supplies the fixed comparison constant `SINT#0` as an adapter value (`INIT_VAL=SINT#0`). An adapter socket cannot take the constant as a parameter, so it is fed in through this block.
  - **AS_GT**: `adapter::iec61131::comparison::AS_GT` — adapter-native comparison `IN1 > IN2`.
- **Functionality**: `IN` ⟶ `AS_GT.IN1`; constant 0 ⟶ `AS_GT.IN2`; result ⟶ `AX_OUT`.

Program Flow and Connections

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

Application Scenarios

- Converting a signed 8-bit value into a simple boolean adapter signal for VT status displays or enable logic.

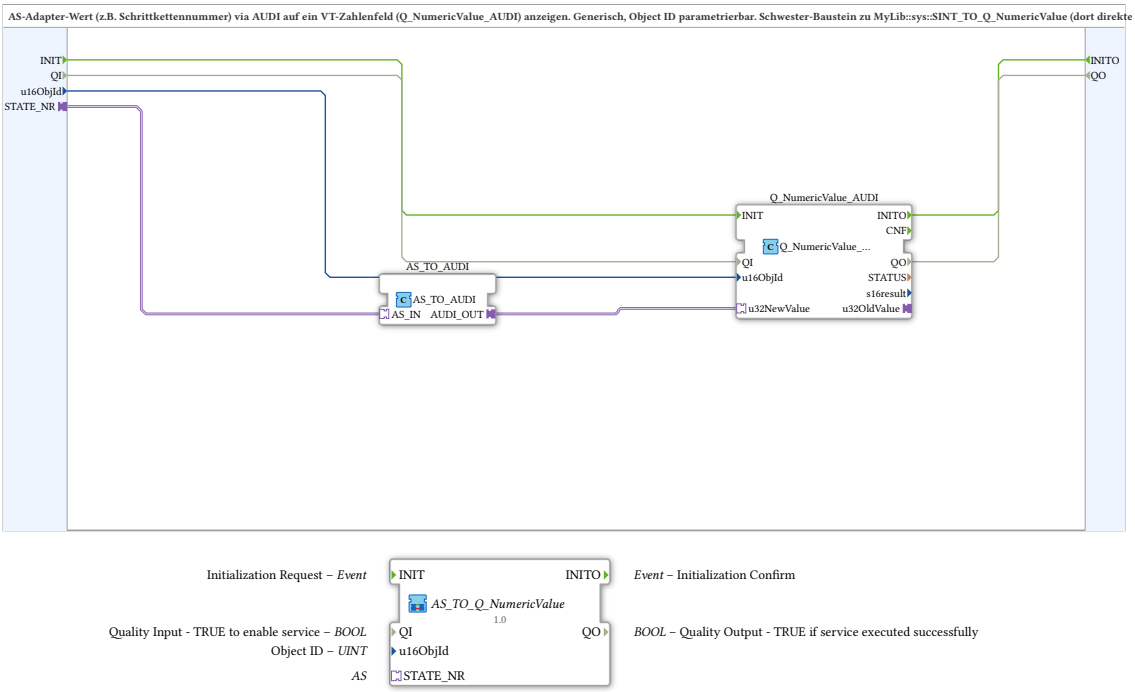
Summary

`AS_GT_0` is a compact wrapper around the adapter-native comparison `AS_GT` — converts "SINT > 0" directly into an AX adapter signal, so the caller does not have to create and wire the zero constant itself.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AS_TO_Q_NumericValue



• — ○ • — ○ • — ○ •

Introduction

`AS_TO_Q_NumericValue` is a reusable subapplication designed to display an AS adapter value (e.g., a step number from a sequencer) on a Visualization Terminal (VT) numeric field. It generically uses an Object ID to select the target field and leverages the `Q_NumericValue_AUDI` function block from the `isobus::UT::Q` library. The block acts as a cleaner interface by accepting an AS adapter input instead of a raw numeric signal, making it suitable for integration into Industrial Automation systems.

Interface Structure

The subapplication exposes a minimal interface: the initialization events `INIT`/`INITO`, the quality signals `QI`/`QO`, one further data input and one adapter socket.

Event Inputs

- **INIT** (`EInit`): initialization request - passed through to `Q_NumericValue_AUDI.INIT`.

Event Outputs

- **INITO** (`EInit`): initialization confirmation - comes from `Q_NumericValue_AUDI.INITO`.

Data Inputs

Name	Type	Initial Value	Comment

QI	BOOL	TRUE	Quality Input - TRUE to enable the service.
u16ObjId	UINT	ID_NULL	Object ID for the VT numeric field (target).

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if the service was executed successfully (from Q_NumericValue_AUDI.QO).

Adapters

Name	Type	Direction	Comment
STATE_NR	adapter::types::unidirectional::AS	Socket	AS adapter input carrying the value to display.

Functionality

The subapplication performs a unidirectional data conversion and transfer:

1. STATE_NR (AS adapter) carries the value (e.g., a step number) as an AS type.
2. The internal AS_TO_AUDI block converts the AS adapter value into an AUDI-compatible data stream (AUDI_OUT).
3. The converted AUDI value is then fed into Q_NumericValue_AUDI as u32NewValue.
4. Simultaneously, the u16ObjId input is passed directly to Q_NumericValue_AUDI to define which VT numeric field should be updated.

In addition, initialization and the quality signals are passed through: INIT → Q_NumericValue_AUDI.INIT, Q_NumericValue_AUDI.INITO → INITO, QI → Q_NumericValue_AUDI.QI and Q_NumericValue_AUDI.QO → QO.

In essence, the block bridges an AS signaling network with a VT display, making the object ID externally configurable without changing internal wiring.

Technical Features

- **Generic Object ID**: The u16ObjId input is fully parametrizable, allowing the same subapplication to control different VT fields without duplication.
- **Reusability**: Designed for reuse across multiple projects or instances; it was extracted from a larger exercise for modularity.
- **Leverages Standard Library Blocks**: Utilizes isobus::UT::Q::Q_NumericValue_AUDI and adapter::conversion::unidirectional::AS_TO_AUDI, ensuring compatibility with IEC 61499-2 and the 4diac IDE.
- **Simple Wiring**: Only one adapter input and one data input are required (QI defaults to TRUE, INIT is optional); no internal logic or state variables are exposed.
- **Quality Pattern**: QI (default TRUE) enables the service, QO reports the success of the inner Q_NumericValue_AUDI.

State Overview

This subapplication is **stateless** in its own right. It does not maintain any internal state variables or sequencing logic. All data flows are purely combinational from the adapter input and the object ID to the output of the internal Q_NumericValue_AUDI block. The functionality is deterministic and does not depend on execution order or history.

Application Scenarios

Typical use cases include:

- Displaying the current step number of a sequencer on an HMI or VT.
- Indicating the active recipe or status code from a machine as a numeric readout.
- Updating a numeric field on a VT based on a value from an AS-connected device, without requiring direct signal wiring.

- Serving as a drop-in replacement in systems that previously used a direct `SINT` input, now adding the flexibility of an AS adapter.

Comparison with Similar Blocks

The sister block `MyLib::sys::SINT_TO_Q_NumericValue` provides the same functionality but expects a direct `SINT` value as input instead of an AS adapter. The main differences:

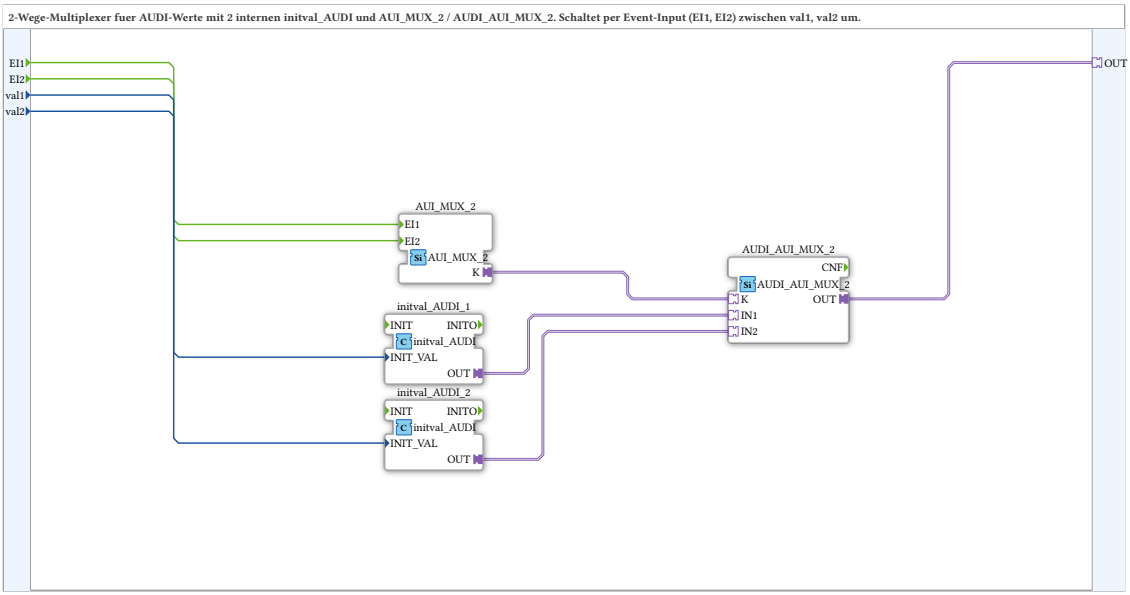
- **Input Type**: `SINT_TO_Q_NumericValue` takes a plain `SINT` data signal; `AS_TO_Q_NumericValue` requires an AS adapter.
- **Flexibility**: The AS adapter version is better suited for environments already using AS-communication patterns, while the SINT version may be simpler for purely data-oriented connections.
- **Wiring**: The AS variant adds an extra conversion layer (`AS_TO_AUDI`), which is transparent to the user and handled internally.

Choosing one over the other depends on the surrounding system architecture and the availability of AS adapter connections.

Conclusion

`AS_TO_Q_NumericValue` is a compact, reusable subapplication that effectively translates an AS adapter value into a numeric display on a VT, with an externally configurable object ID. Its stateless design and use of standard library blocks make it straightforward to integrate into IEC 61499-based projects. By reusing this block, developers can avoid duplicating conversion logic and ensure consistency across automation projects that require numeric visualization of AS signals.

AUDI_AUI_MUX_2_VAL





• – ○ • – ○ • – ○ •

Introduction

The **AUDI_AUI_MUX_2_VAL** is a composite application block (SubApp) that implements a two-channel multiplexer for AUDI data values. It selects between two initialization values (`val1` and `val2`) based on incoming event triggers (`EI1` and `EI2`). The selected value is provided as an AUDI adapter output. Inter-

nally, the block combines an event-driven selector (AUI_MUX_2) with a dedicated adapter multiplexer (AUDI_AUI_MUX_2) and two initial-value generators (initval_AUDI). This design allows clean orchestration of event-controlled value switching while maintaining the structure of AUDI adapters.

Interface Structure

The block is a SubApp with the following interface elements:

Event Inputs

Name	Type	Description
`EI1`	Event	Triggers selection of `val1`
`EI2`	Event	Triggers selection of `val2`

Event Outputs

The block has no explicit event outputs. Internal event handling is entirely contained within the subapplication.

Data Inputs

Name	Type	Description
`val1`	UDINT	First value to be output when `EI1` is triggered
`val2`	UDINT	Second value to be output when `EI2` is triggered

Data Outputs

The block does not provide direct data outputs. The result is delivered through the adapter output.

Adapters

Name	Type	Description
`OUT`	`adapter::types::unidirectional::AUDI`	Output adapter carrying the currently selected AUDI value

Functionality

The internal network consists of three functional parts:

1. **Event decoding** – The incoming events `EI1` and `EI2` are routed to the internal FB `AUI_MUX_2` (type `adapter::events::unidirectional::AUI_MUX_2`). This FB acts as an event selector, forwarding the corresponding selection signal to the adapter multiplexer.
2. **Value preparation** – The two input values `val1` and `val2` are fed to two instances of `initval_AUDI` (type `adapter::types::unidirectional::AUDI::initval::initval_AUDI`). These FBs convert the primitive UDINT data into AUDI adapter format, making them suitable for the adapter-based multiplexing stage.
3. **Adapter multiplexing** – The internal FB `AUDI_AUI_MUX_2` (type `adapter::selection::unidirectional::AUDI_AUI_MUX_2`) receives the selection control from `AUI_MUX_2` and the two prepared AUDI-formatted values from the `initval_AUDI` instances. Based on the control signal, it passes either `IN1` or `IN2` to its output `OUT`, which is then propagated to the block's external adapter output.

In summary, when `EI1` arrives, `val1` is output; when `EI2` arrives, `val2` is output. The output remains at the last selected value until the other event is received.

Technical Features

- **Modular composition** – Uses event-, data-, and adapter-oriented FBs in a clear separation of concerns.
- **Adapter-based data flow** – All AUDI data is handled through dedicated adapter types, ensuring type-safe connections.

- **Initial value handling** – The `initval_AUDI` FBs ensure that the UDINT input values are correctly wrapped into the AUDI adapter format before multiplexing.
- **Event-driven selection** – Selection is performed via event inputs, allowing precise timing control.
- **Unidirectional output** – Only one AUDI output adapter is provided; no feedback or bidirectional interfaces are used.

State Overview

The block does not maintain a persistent internal state machine; it is purely combinatorial with respect to data, while events drive the selection process. However, the output effectively "latches" the last selected value until an opposite event occurs. In that sense, the block can be considered to have two stable output states:

- **State A**: `val1` is active (after `EI1` has been received)
- **State B**: `val2` is active (after `EI2` has been received)

Since no internal memory elements are present, the state is determined solely by the most recent event that has been processed.

Application Scenarios

This block is particularly useful in automation and control systems where a single AUDI-type data stream must be switched between two different source values based on discrete events. Typical applications include:

- **Mode switching** – Selecting between different operating parameters (e.g., speed or temperature setpoints) based on operational mode events.
- **Redundant value selection** – Choosing between a primary and a backup measurement value depending on failure detection events.
- **Commanded swapping** – Alternating between two preset values (e.g., for test or calibration purposes) via digital triggers.

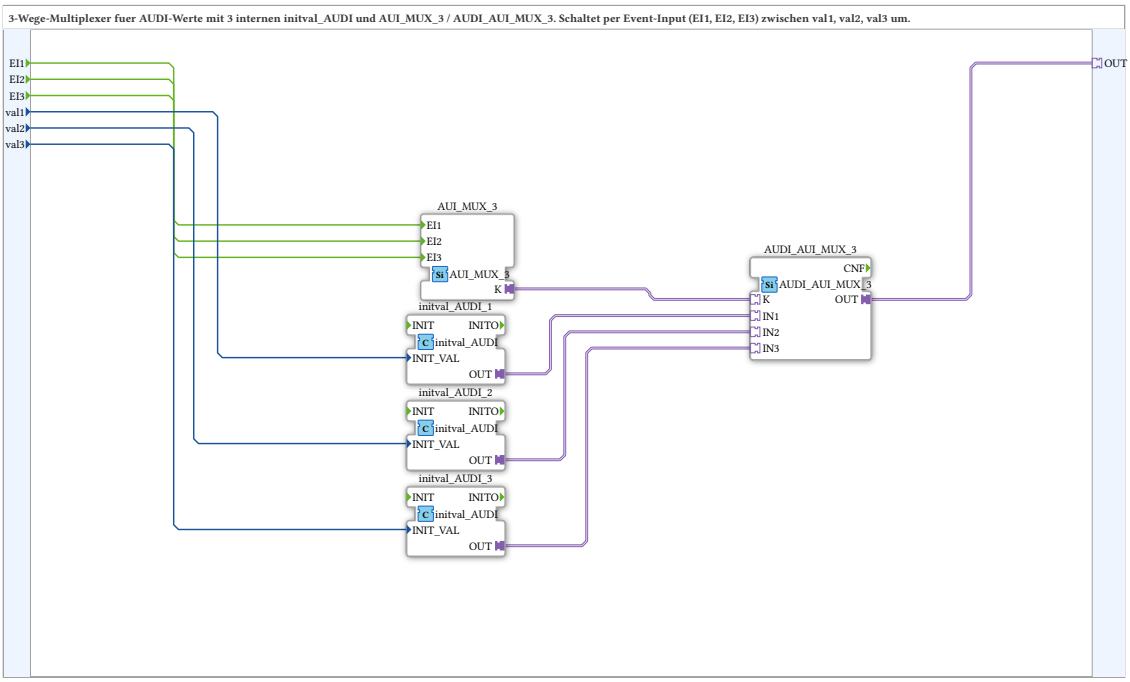
Comparison with Similar Blocks

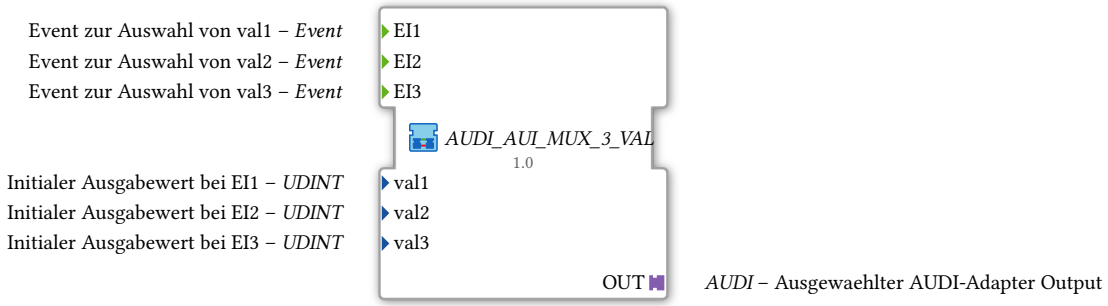
The `AUDI_AUI_MUX_2_VAL` is a specialized variant of a generic multiplexer. Unlike a plain data multiplexer (e.g., `MUX`), it works with AUDI adapter data and incorporates the necessary initial-value conversion. Compared to a three-channel version (`AUDI_AUI_MUX_3_VAL`), this block offers only two inputs, making it more compact for simple two-source scenarios. The internal separation into event selector and adapter selector allows reuse of the individual components in other contexts, but for an end-user the block behaves as a single, ready-to-use entity.

Conclusion

The `AUDI_AUI_MUX_2_VAL` subapplication provides a clean and efficient way to switch between two AUDI-formatted values based on event triggers. Its internal architecture leverages existing dedicated FBs, ensuring robustness and reusability. With its simple event-driven interface and adapter-based output, it integrates seamlessly into larger 4diac projects that require controlled data selection.

AUDI_AUI_MUX_3_VAL





• – ○ • – ○ • – ○ •

Introduction

The **AUDI_AUI_MUX_3_VAL** is a subapplication that implements a 3-way multiplexer for AUDI adapter values. It combines three internal `initval_AUDI` function blocks, an `AUI_MUX_3` event selection block, and an `AUDI_AUI_MUX_3` adapter multiplexer. Depending on which event input (`EI1`, `EI2`, or `EI3`) is triggered, one of the three UDINT input values (`val1`, `val2`, `val3`) is selected and made available as an AUDI adapter output. The subapp enables direct wiring of raw numeric values into the AUDI adapter domain without requiring pre-initialized adapter instances externally.

Interface Structure

Event Inputs

Event	Description
`EI1`	Event to select `val1` as the active output
`EI2`	Event to select `val2` as the active output
`EI3`	Event to select `val3` as the active output

Event Outputs

No event outputs are provided by this subapplication.

Data Inputs

Data	Type	Description
`val1`	UDINT	Initial output value when `EI1` is triggered
`val2`	UDINT	Initial output value when `EI2` is triggered
`val3`	UDINT	Initial output value when `EI3` is triggered

Data Outputs

No data outputs are provided by this subapplication.

Adapters

Adapter	Type	Description
`OUT`	`adapter::types::unidirectional::AUDI`	Selected AUDI adapter output carrying the multiplexed value

Functionality

The subapplication operates as a complete selection chain for AUDI values:

1. The three raw UDINT values `val1`, `val2`, and `val3` are passed to three internal `initval_AUDI` function blocks (`initval_AUDI_1`, `initval_AUDI_2`, `initval_AUDI_3`). Each block converts the corresponding UDINT value into an AUDI adapter initial-value stream.
2. The internal `AUI_MUX_3` block receives the event inputs `EI1`, `EI2`, and `EI3` and performs event-based selection logic.
3. The selection information is forwarded via the adapter connection `AUI_MUX_3.K` to the internal `AUDI_AUI_MUX_3` block.
4. The `AUDI_AUI_MUX_3` block multiplexes the three AUDI adapter inputs (`IN1`, `IN2`, `IN3`) originating from the `initval_AUDI` blocks and routes the chosen adapter stream to the output adapter `OUT`.

The entire chain is event-driven: only when an event is raised on `EI1`, `EI2`, or `EI3` does the output adapter reflect the corresponding input value.

Technical Features

- **Three-way selection** via dedicated event inputs (`EI1`, `EI2`, `EI3`), providing explicit and deterministic switching behavior.
- **UDINT input type** for `val1`, `val2`, and `val3`, allowing direct connection of unsigned double integer values from the application logic.
- **Adapter-based output** using the unidirectional `AUDI` adapter type, enabling seamless integration with other AUDI-compatible blocks.
- **Internal composition** combines `initval_AUDI` (value conversion), `AUI_MUX_3` (event selection), and `AUDI_AUI_MUX_3` (adapter multiplexing) into a single reusable subapplication.
- **No additional event or data outputs**, keeping the interface clean and focused on the multiplexing purpose.

State Overview

The subapplication does not maintain an explicit internal state machine. Selection behavior is purely event-driven: each trigger on `EI1`, `EI2`, or `EI3` immediately determines which of the three values is propagated to the output. Consequently, the active output corresponds to the most recently triggered event input, and no intermediate states are exposed to the user.

Application Scenarios

Typical use cases for the `AUDI_AUI_MUX_3_VAL` include:

- **Mode or channel selection** in automation processes where one of three preconfigured AUDI values must be chosen based on external events (e.g., operating mode switches, recipe selection).
- **Parameter switching** in control systems, enabling dynamic changes between different setpoints or configuration values that are supplied as raw UDINT values.

- **Redundant value handling** in safety-critical applications, where a fallback or alternative value can be selected via specific event triggers.
- **Integration into larger adapter-based networks**, providing a convenient bridge between raw numeric inputs and AUDI adapter streams without manual adapter initialization.

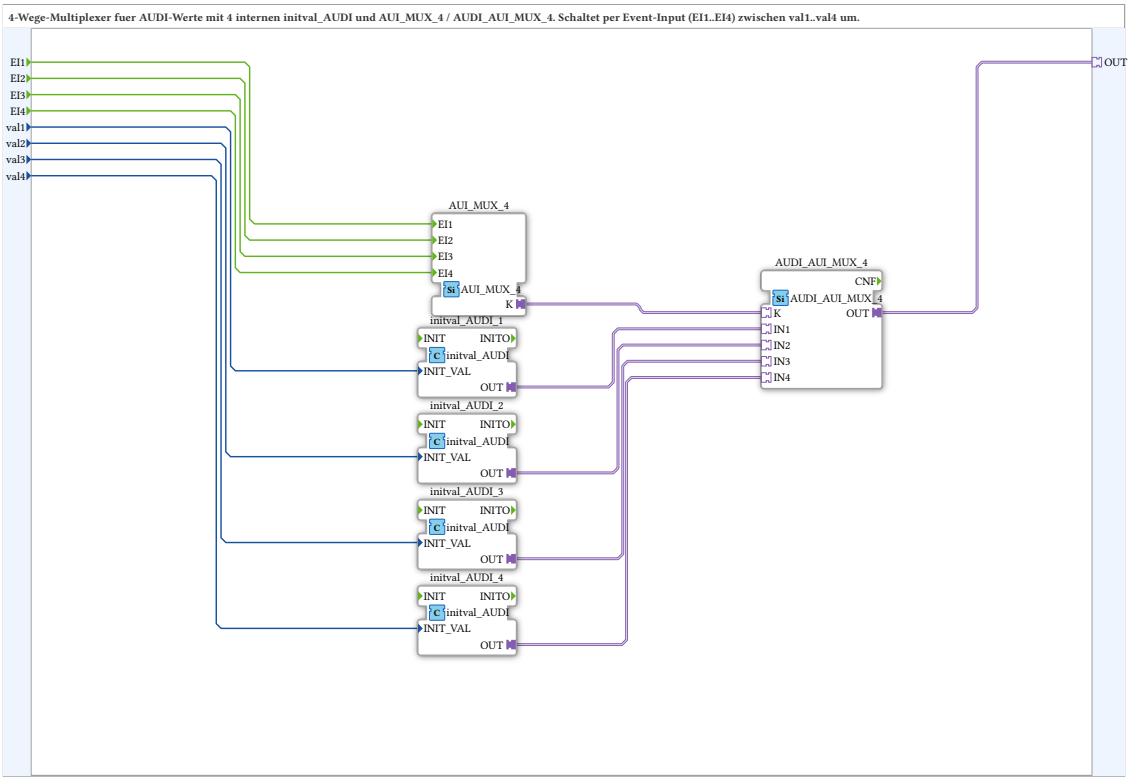
Comparison with Similar Blocks

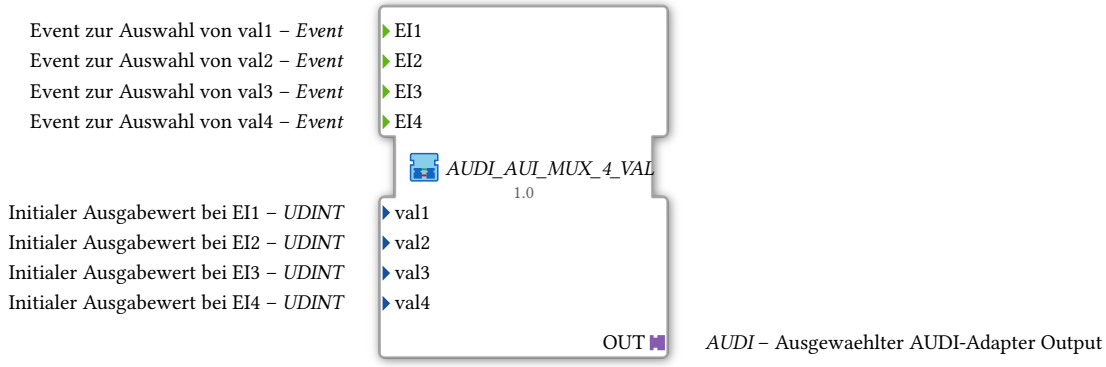
Compared to the standard `AUDI_AUI_MUX_3` block, the `AUDI_AUI_MUX_3_VAL` introduces an integrated initialization layer through the `initval_AUDI` blocks. This eliminates the need for the user to create and configure adapter instances beforehand; raw UDINT values can be wired directly. In contrast to a generic multiplexer without adapter support, this subapplication provides a complete, self-contained solution tailored to the AUDI adapter type, reducing external wiring complexity and improving reusability.

Conclusion

The `AUDI_AUI_MUX_3_VAL` subapplication offers a compact and practical solution for three-way multiplexing of AUDI adapter values driven by event inputs. By internally combining value initialization, event selection, and adapter multiplexing, it simplifies the integration of numeric data into AUDI-based communication chains. Its clear interface, event-driven behavior, and adapter-oriented design make it suitable for a wide range of automation and control scenarios where flexible value switching is required.

AUDI_AUI_MUX_4_VAL





• – ○ • – ○ • – ○ •

Introduction

AUDI_AUI_MUX_4_VAL is a composite subapplication that implements a 4-way multiplexer for AUDI adapter values. It selects between four internal AUDI values using the event inputs EI1 to EI4. Each data input val1 to val4 provides the initial value for one internal AUDI channel. The selected value is made available through a single AUDI adapter output plug.

Interface Structure

The subapplication has four event inputs, four data inputs, and one adapter output. It does not provide any event outputs or data outputs.

Event Inputs

Name	Type	Comment
EI1	Event	Event for selecting val1
EI2	Event	Event for selecting val2
EI3	Event	Event for selecting val3
EI4	Event	Event for selecting val4

Event Outputs

There are no event outputs.

Data Inputs

Name	Type	Comment
val1	UDINT	Initial output value for EI1
val2	UDINT	Initial output value for EI2
val3	UDINT	Initial output value for EI3
val4	UDINT	Initial output value for EI4

Data Outputs

There are no data outputs.

Adapters

Direction	Name	Type	Comment
Plug	OUT	adapter::types::unidirectional::AUDI	Selected AUDI adapter output

Functionality

AUDI_AUI_MUX_4_VAL combines four internal `initval_AUDI` instances with two adapter-based multiplexing elements. Each `initval_AUDI` instance receives one of the UDINT data inputs and converts it into an internal AUDI adapter value. The four resulting AUDI channels are connected to the selection block `AUDI_AUI_MUX_4`.

The event inputs EI1 to EI4 are forwarded to the internal event multiplexer `AUI_MUX_4`. Its control output `K` drives `AUDI_AUI_MUX_4` to select the corresponding AUDI input. When an event occurs on EIn, the AUDI value associated with valn is propagated to the `OUT` adapter plug.

This architecture allows the subapplication to switch between four different AUDI values purely by event control, without requiring additional data selection logic on the outside.

Technical Features

- 4-way event-driven selection of AUDI adapter values.
- Four independent UDINT input values.
- No event outputs or data outputs.
- Output provided as a unidirectional AUDI adapter plug.

- Built from standard adapter-based function blocks:
  - `AUI_MUX_4`
  - `AUDI_AUI_MUX_4`
  - Four `initval_AUDI` instances
- Uses an internal control connection `K` between the event multiplexer and the AUDI selection block.

State Overview

The subapplication is defined as a composite network and does not contain its own explicit ECC state machine. Its behavior is event-driven: the active AUDI output value depends on the most recently triggered event input. The internal blocks handle the selection and forwarding of the adapter values.

Application Scenarios

AUDI_AUI_MUX_4_VAL is suitable for applications that need to select one out of four AUDI adapter values at runtime. Typical use cases include:

- Switching between four preconfigured AUDI parameter sets.
- Event-based routing of AUDI data channels.
- Selecting initialization values for different operating modes.
- Multiplexing AUDI values in automation or control networks.

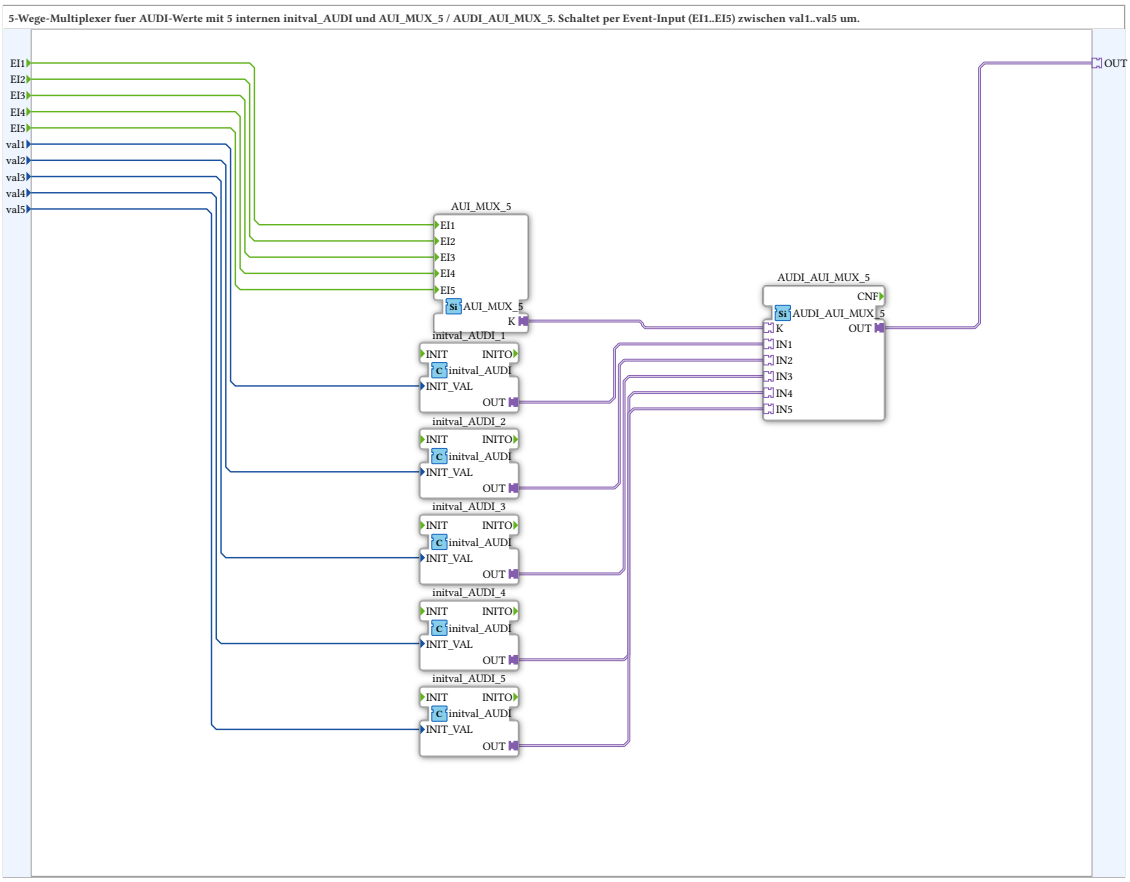
Comparison with Similar Blocks

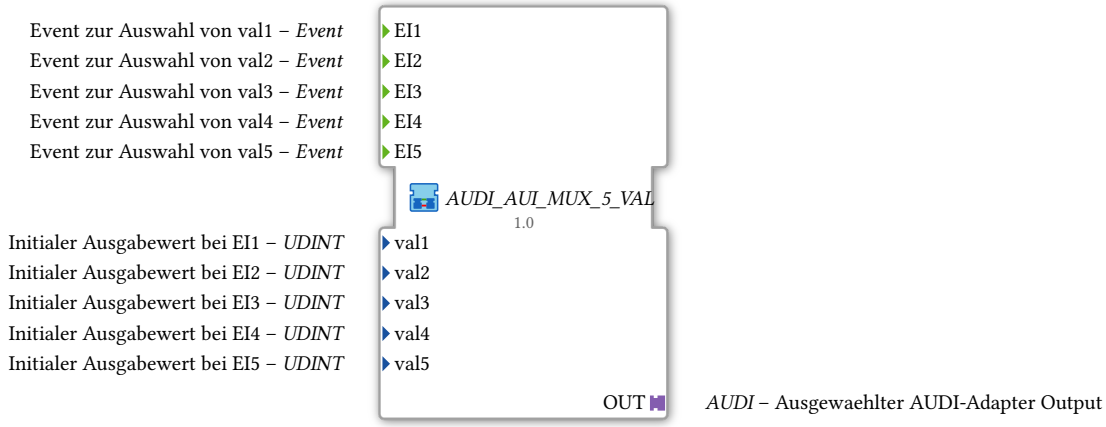
AUDI_AUI_MUX_4_VAL is the 4-way variant of the 3-way subapplication `AUDI_AUI_MUX_3_VAL`. Compared to that block, it provides one additional event input, one additional data input, and a fourth internal AUDI value channel. This makes it useful when exactly four alternatives must be handled. For applications with fewer inputs, the 3-way variant may be more compact; for larger selections, additional variants or cascaded multiplexers are required.

Conclusion

AUDI_AUI_MUX_4_VAL provides a clean and reusable way to multiplex four UDINT-based AUDI adapter values through a single output. Its event-driven interface is simple to integrate into 4diac applications, and the internal composition hides the adapter selection complexity from the surrounding logic.

AUDI_AUI_MUX_5_VAL





• – ○ • – ○ • – ○ •

Introduction

The **AUDI_AUI_MUX_5_VAL** is a composite subapplication that implements a 5-way multiplexer for AUDI values. It combines an event-based selection mechanism with five independent initial value generators, allowing the user to select one of five AUDI data streams via dedicated event inputs. The block is designed to be used in scenarios where multiple AUDI sources need to be switched to a single output line based on control events.

Interface Structure

Event Inputs

Event Name	Description

EI1	Event to select val1 as the active input
EI2	Event to select val2 as the active input
EI3	Event to select val3 as the active input
EI4	Event to select val4 as the active input
EI5	Event to select val5 as the active input

Event Outputs

None.

Data Inputs

Data Name	Type	Description
val1	UDINT	Initial value used when EI1 is triggered
val2	UDINT	Initial value used when EI2 is triggered
val3	UDINT	Initial value used when EI3 is triggered
val4	UDINT	Initial value used when EI4 is triggered
val5	UDINT	Initial value used when EI5 is triggered

Data Outputs

None.

Adapters

Plug Name	Type	Description
OUT	adapter::types::unidirectional::AUDI	Selected AUDI adapter output

Functionality

The subapplication is built around three functional blocks:

- **AUI_MUX_5** – receives the five event inputs and generates a selection signal (K) based on which event has occurred.
- **initval_AUDI_1 ... initval_AUDI_5** – each of these blocks holds one of the UDINT input values and converts it into an AUDI adapter output.
- **AUDI_AUI_MUX_5** – a dedicated AUDI multiplexer that takes the five AUDI inputs and the selection signal K, and routes the selected input to the output adapter.

When an event (EI1 ... EI5) is triggered, the corresponding `val1 … val5` value is propagated via the internal initval block to the multiplexer, and the output adapter `OUT` provides the selected AUDI data stream.

Technical Features

- **Composite subapplication** – encapsulates the entire logic in a single reusable block.
- **Five independent input channels** – each with its own event and data value.
- **UDINT data type** – supports 32-bit unsigned integer values for the initial value parameters.
- **AUDI adapter interface** – output is a standard unidirectional AUDI adapter, enabling easy integration with other AUDI-based components.
- **Event-driven selection** – the output is updated only when one of the selection events occurs.
- **No event or data outputs** – all outputs are carried via the adapter plug.

State Overview

The block does not have an explicit state machine, but its behavior can be described as:

- **Idle** – no event has been received; the output adapter holds the last selected value.
- **Selection** – upon receiving one of the five events, the corresponding input is latched and the output adapter is updated accordingly.

The internal AUI_MUX_5 and AUDI_AUI_MUX_5 ensure that exactly one input is routed to the output at any time.

Application Scenarios

- **Industrial control** – switching between different sensor values (e.g., temperature, pressure) based on operational modes.
- **Automation** – selecting different recipe data for a production line.
- **Test and measurement** – multiplexing multiple measurement channels into a single acquisition system.
- **Component integration** – providing a unified AUDI interface to a higher-level controller while allowing channel selection via events.

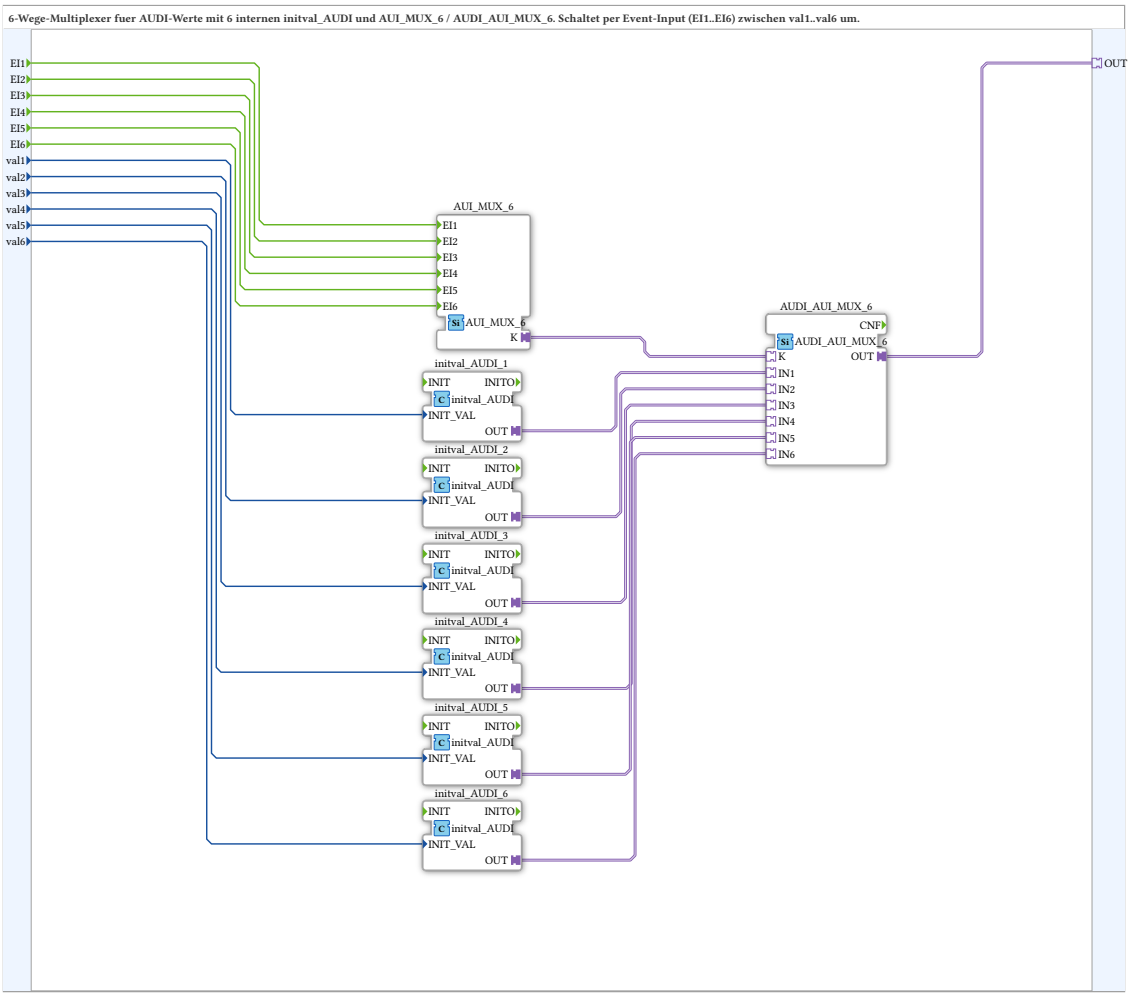
Comparison with Similar Blocks

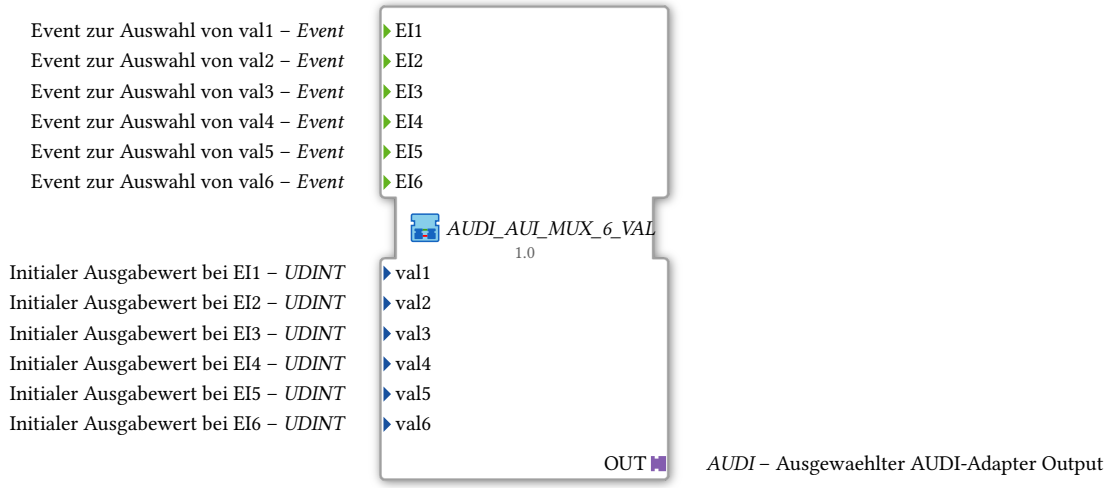
The block is a **5-input variant** of the `AUDI_AUI_MUX_3_VAL` subapplication. Compared to a basic `AUI_MUX_5` (which only provides the selection signal), this block adds the initial value generation and the AUDI multiplexing in a single package, making it more self-contained. It also differs from a simple `AUDI_MUX_5` (which works on pre-configured AUDI inputs) by taking raw UDINT values as input and automatically converting them to the AUDI format.

Conclusion

The `AUDI_AUI_MUX_5_VAL` subapplication provides a clean, event-driven solution for multiplexing up to five AUDI channels. Its internal structure isolates the selection and conversion logic, making it easy to integrate into larger systems. The use of standard AUDI adapter interfaces ensures compatibility with existing 4diac components and simplifies wiring.

AUDI_AUI_MUX_6_VAL





• – ◦ • – ◦ • – ◦ •

Introduction

The AUDI_AUI_MUX_6_VAL is a 6-way multiplexer subapplication designed for selecting AUDI values. It combines six internal initval_AUDI blocks with an AUI_MUX_6 event multiplexer and an AUDI_AUI_MUX_6 selection adapter. The subapp switches between six UDINT input values (val1 to val6) based on event inputs EI1 to EI6, providing the selected value as an AUDI adapter output.

Interface Structure

Event Inputs

Event	Type	Comment
EI1	Event	Event to select val1
EI2	Event	Event to select val2
EI3	Event	Event to select val3
EI4	Event	Event to select val4
EI5	Event	Event to select val5
EI6	Event	Event to select val6

Event Outputs

No event outputs are provided by this subapplication.

Data Inputs

Data	Type	Comment
val1	UDINT	Initial output value at EI1
val2	UDINT	Initial output value at EI2
val3	UDINT	Initial output value at EI3
val4	UDINT	Initial output value at EI4
val5	UDINT	Initial output value at EI5
val6	UDINT	Initial output value at EI6

Data Outputs

No data outputs are provided by this subapplication.

Adapters

Adapter	Type	Comment
OUT	adapter::types::unidirectional::AUDI	Selected AUDI adapter output

Functionality

The subapplication implements a 6-to-1 multiplexer for AUDI values. Internally, it contains:

- Six initval_AUDI instances (initval_AUDI_1 to initval_AUDI_6) that convert the UDINT input values (val1 to val6) into AUDI adapter format. Each initval block receives its corresponding value via the INIT_VAL data input.
- An AUI_MUX_6 event multiplexer block that receives the six event inputs (EI1 to EI6).
- An AUDI_AUI_MUX_6 selection adapter that combines the multiplexed event output (K) and the six initialized AUDI values (IN1 to IN6).

When an event arrives on one of the event inputs (EI1 to EI6), the AUI_MUX_6 block routes the corresponding selection signal to the AUDI_AUI_MUX_6 block. The AUDI_AUI_MUX_6 then forwards the corresponding initialized AUDI value to the OUT adapter.

Technical Features

- Six-channel event-driven selection
- UDINT input values converted to AUDI adapter values
- Unidirectional AUDI adapter output
- Clean separation of event selection and data routing
- Scalable design based on the 3-way variant

State Overview

The subapplication itself does not maintain an explicit state machine. The selection behavior is determined by the internal AUI_MUX_6 block, which reacts to the incoming events. After initialization, the output reflects the AUDI value associated with the most recently triggered event input.

Application Scenarios

- Selecting between multiple sensor values represented as AUDI data
- Switching between different preset AUDI configurations
- Multiplexing AUDI data streams in an IEC 61499-based control application

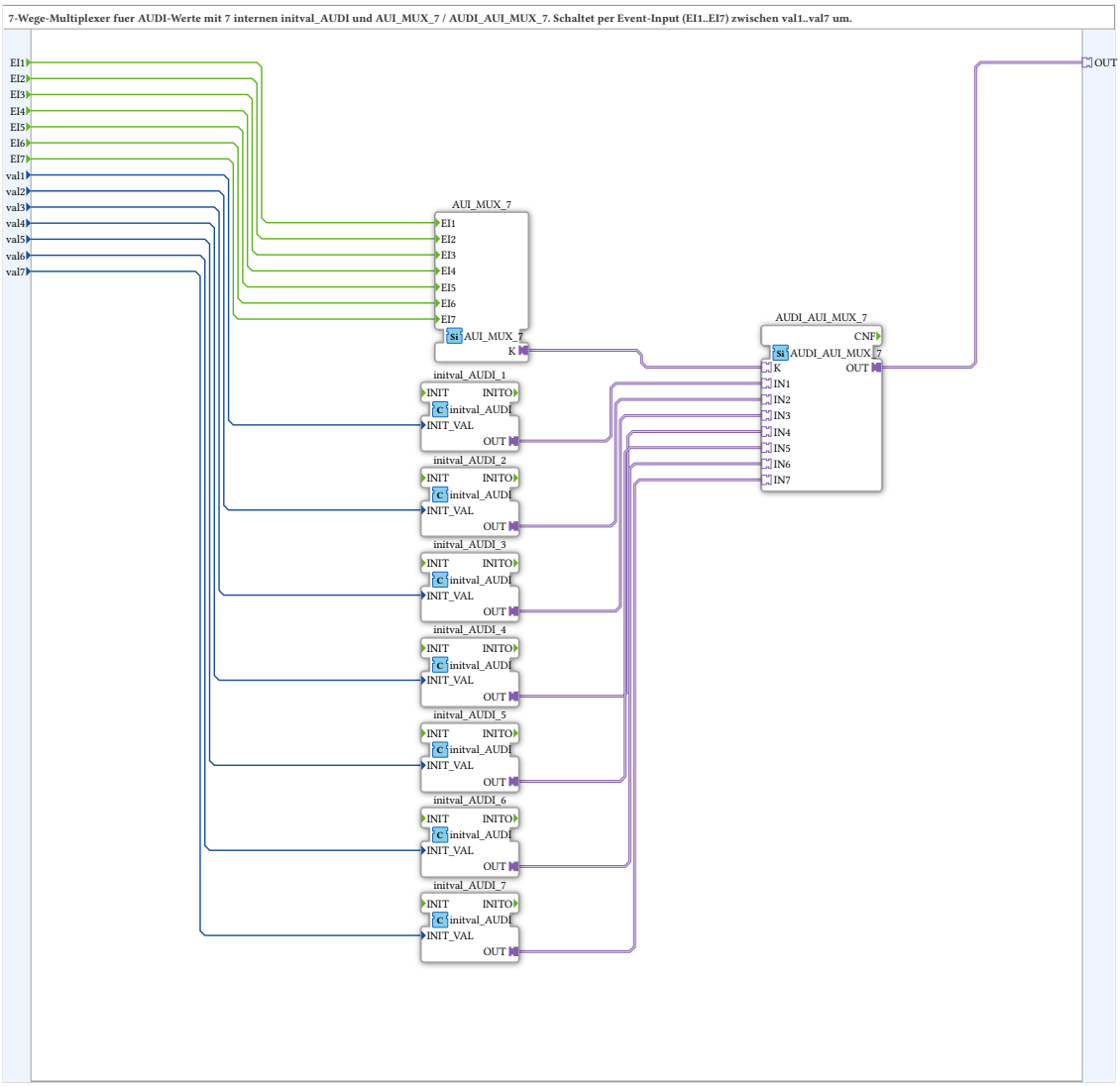
Comparison with Similar Blocks

This block is a 6-way extension of the 3-way variant (AUDI_AUI_MUX_3_VAL). Compared to its smaller counterpart, it provides two additional event inputs and data inputs (EI4/EI5/EI6 and val4/val5/val6), enabling the selection between six different AUDI values. The internal architecture follows the same pattern, using initval_AUDI blocks and a shared AUDI_AUI_MUX_6 selection adapter.

Conclusion

The AUDI_AUI_MUX_6_VAL subapplication provides a convenient, event-driven way to select between six UDINT values and output the corresponding AUDI adapter value. Its modular design makes it suitable for applications requiring multiple selectable AUDI data sources.

AUDI_AUI_MUX_7_VAL



Event zur Auswahl von val1 – *Event* EI1
Event zur Auswahl von val2 – *Event* EI2
Event zur Auswahl von val3 – *Event* EI3
Event zur Auswahl von val4 – *Event* EI4
Event zur Auswahl von val5 – *Event* EI5
Event zur Auswahl von val6 – *Event* EI6
Event zur Auswahl von val7 – *Event* EI7

AUDI_AUI_MUX_7_VAL
1.0

Initialer Ausgabewert bei EI1 – *UDINT* val1
Initialer Ausgabewert bei EI2 – *UDINT* val2
Initialer Ausgabewert bei EI3 – *UDINT* val3
Initialer Ausgabewert bei EI4 – *UDINT* val4
Initialer Ausgabewert bei EI5 – *UDINT* val5
Initialer Ausgabewert bei EI6 – *UDINT* val6
Initialer Ausgabewert bei EI7 – *UDINT* val7

OUT – *AUDI* – Ausgewaehlter AUDI-Adapter Output

• – ○ • – ○ • – ○ •

Introduction

The **AUDI_AUI_MUX_7_VAL** is a 7-way multiplexer subapplication designed to select one of seven AUDI values based on an event input. It extends the concept of a 3-channel variant to support seven independent input channels. The block integrates event-based selection with adapter-based data transfer, making it suitable for scenarios where a single AUDI output must be dynamically switched among multiple pre-configured values.

Interface Structure

Event Inputs

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

Event Outputs

None – the subapplication does not provide any event outputs.

Data Inputs

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

Data Outputs

None – all output data is provided through the adapter interface.

Adapters

Name	Type	Comment
OUT	adapter::types::unidirectional::AUDI	Ausgewählter AUDI-Adapter Output

Functionality

The subapplication operates as an event-driven selector. When one of the event inputs (EI1…EI7) receives a rising edge or a trigger, the corresponding data value (val1…val7) is made available at the output adapter **OUT**. Internally, the following components work together:

- Seven **initval_AUDI** function blocks are used to convert the UDINT input values into the AUDI adapter format. Each block receives one of the `val` inputs and provides an AUDI-typed output.

- The **AUI_MUX_7** event multiplexer forwards the incoming event from the selected input to a corresponding internal selection signal.
- The **AUDI_AUI_MUX_7** data multiplexer then routes the AUDI value from the selected channel to the output adapter.

The selection is purely event-based; no additional control input is required. The output continuously reflects the value of the most recently triggered input channel until another event changes it.

Technical Features

- **IEC 61499-compliant** subapplication structure.
- **Event-driven selection** – no polling or cycle-based updates.
- **UDINT data type** for the input values, providing 32-bit unsigned integer range.
- **Adapter-based output** (AUDI) enabling seamless integration with other components of an AUDI-based system.
- **Scalable design** – the internal architecture can be extended to support more channels if needed.
- **No explicit state variables** the behaviour is deterministic and purely reactive.

State Overview

The subapplication does not maintain an internal state machine. Its behaviour is purely combinatorial with respect to the last received event. The active channel is determined solely by the most recent occurrence of an event on EI1…EI7. Consequently, there is no hysteresis or memory beyond the current output value.

Application Scenarios

- **Audio/video switching** – selecting among up to seven different audio sources (e.g., in a multimedia system).
- **Industrial control** – choosing one of several sensor values (encoded as AUDI) for further processing.
- **Configuration management** – switching between preset parameter sets at runtime.
- **Redundancy handling** – selecting a backup value if the primary becomes unavailable.

Comparison with Similar Blocks

Compared to the -3-channel version (`AUDI_AUI_MUX_3_VAL`), this block:

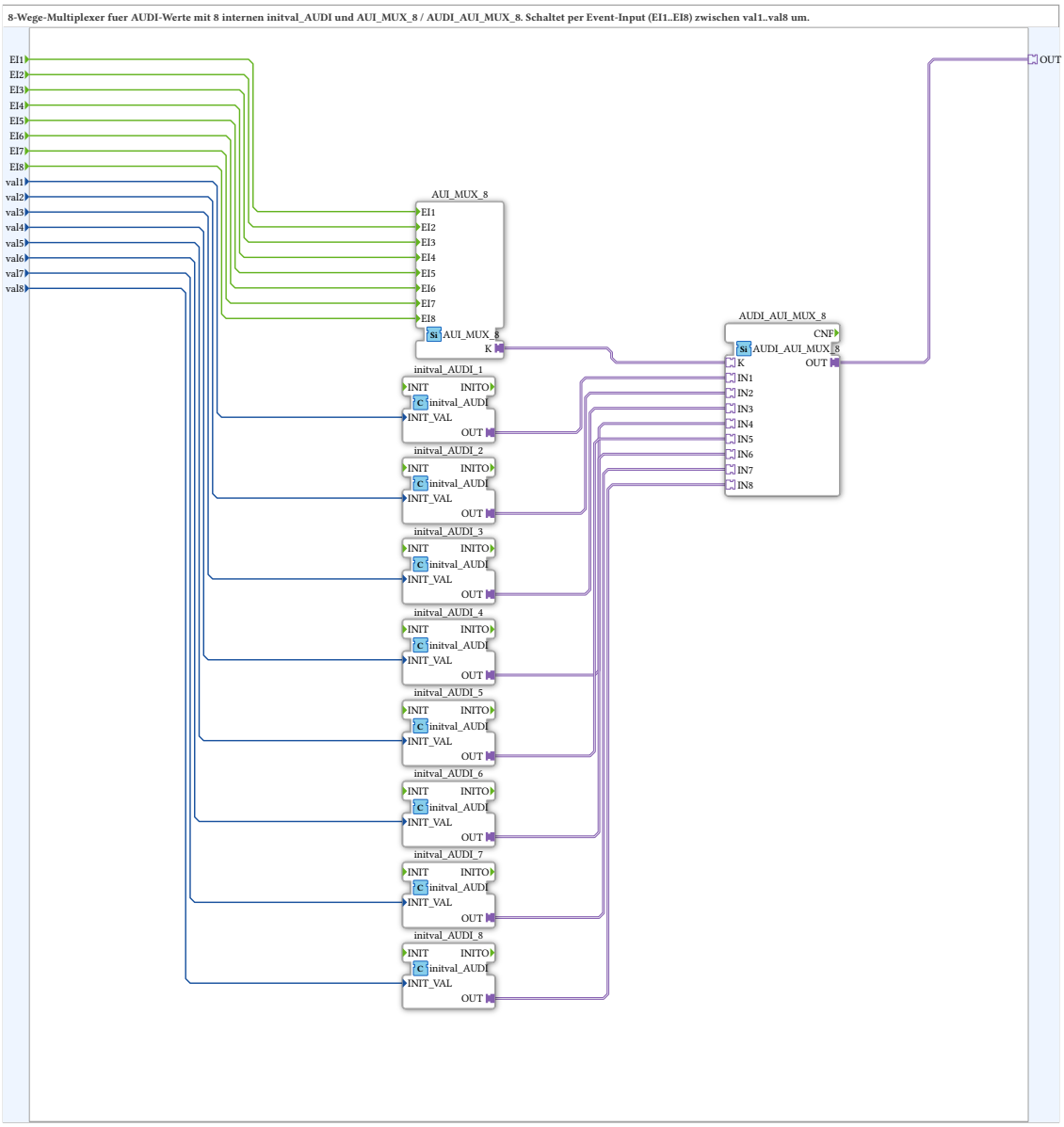
- Provides **seven input channels** instead of three.
- Has a **larger input data port set** (val1…val7).
- Uses a **seven-fold internal replication** of the initialization logic.
- Fulfils the same functional role but for applications requiring more selection options.

Other multiplexer blocks might use a numeric selector input (e.g., QW_INT) instead of multiple event inputs; this variant excels when each channel is triggered by a distinct event (e.g., individual buttons or sensor events).

Conclusion

The **AUDI_AUI_MUX_7_VAL** subapplication is a robust and flexible solution for event-driven selection among up to seven AUDI values. By combining standard IEC 61499 elements with a clear event-based interface, it offers a simple yet powerful mechanism to switch data sources in distributed automation and media systems. Its design ensures high reusability and easy integration into existing AUDI-based architectures.

AUDI_AUI_MUX_8_VAL





• – ◦ • – ◦ • – ◦ •

Introduction

AUDI_AUI_MUX_8_VAL is a composite subapplication that implements an 8-channel multiplexer for values of type `UDINT`. It provides eight event inputs (`EI1` … `EI8`) and eight corresponding data inputs (`val1` … `val8`). When an event is triggered, the associated data value is routed through an internal adapter network and made available at the single AUDI adapter output. The subapplication combines an event multiplexer (`AUI_MUX_8`), an adapter selection block (`AUDI_AUI_MUX_8`), and eight initializer blocks (`initval_AUDI_1` … `initval_AUDI_8`) to achieve a clean, event-driven selection of up to eight `UDINT` values.

Interface Structure

The subapplication exposes a simple, purpose-oriented interface: eight event inputs, eight data inputs, and one adapter output. There are no event outputs, data outputs, or additional adapters.

Event Inputs

Event Name	Description
`EI1`	Event to select `val1` as the output value.
`EI2`	Event to select `val2` as the output value.
`EI3`	Event to select `val3` as the output value.
`EI4`	Event to select `val4` as the output value.
`EI5`	Event to select `val5` as the output value.
`EI6`	Event to select `val6` as the output value.
`EI7`	Event to select `val7` as the output value.
`EI8`	Event to select `val8` as the output value.

Event Outputs

None.

Data Inputs

Data Name	Type	Description

val1	UDINT	Value to be output when EI1 is triggered.
val2	UDINT	Value to be output when EI2 is triggered.
val3	UDINT	Value to be output when EI3 is triggered.
val4	UDINT	Value to be output when EI4 is triggered.
val5	UDINT	Value to be output when EI5 is triggered.
val6	UDINT	Value to be output when EI6 is triggered.
val7	UDINT	Value to be output when EI7 is triggered.
val8	UDINT	Value to be output when EI8 is triggered.

Data Outputs

None.

Adapters

Adapter Name	Type	Description
OUT	adapter::types::unidirectional::AUDI	Adapter output carrying the selected AUDI value.

Functionality

The subapplication operates purely in an event-driven manner. Each of the eight event inputs is directly connected to the corresponding event input of the internal AUI_MUX_8 block. That block uses the active event to produce a selector index K (an integer value) on its output. Simultaneously, each val input is passed to a dedicated initval_AUDI block, which converts the raw UDINT value into an AUDI-typed adapter instance. The eight adapter outputs of the initializers are fed into the eight input channels of the AUDI_AUI_MUX_8 block. Using the selector index K provided by AUI_MUX_8, the selection block forwards exactly one of the eight incoming AUDI adapters to its output, which is then directly connected to the subapplication's OUT adapter.

In summary, triggering any event EI<n> causes the value val<n> to appear at the AUDI output, with no additional data processing. The internal structure ensures that the selection is synchronous with the triggering event and that only one value is active at a time.

Technical Features

- **8-Channel Selection**: Supports up to eight independent input values.
- **Pure Event-Driven Operation**: No continuous polling; each selection is activated by an explicit event.
- **Adapter-Based Output**: The result is provided as a typed AUDI adapter, allowing seamless integration with other AUDI-compatible function blocks.
- **Internal Value Conversion**: The initval_AUDI blocks encapsulate the conversion from UDINT to the AUDI adapter type, keeping the interface simple.
- **Modular Composition**: Uses separate blocks for event multiplexing and adapter selection, which improves reusability and maintainability.
- **No Internal State**: The subapplication is stateless; output selection is fully determined by the most recent event.

State Overview

The subapplication does not maintain any persistent internal state. Its behavior is purely combinational with respect to the triggering event:

- After a reset or at initial startup, no output is selected until one of the events EI1...EI8 is raised.
- When an event EI<n> occurs, the corresponding val<n> is immediately routed to OUT and remains there until another event is triggered.
- The output value is only updated on event edges; data input changes without an associated event do not alter the output.

Because no explicit state variables exist, the subapplication is inherently safe and deterministic.

Application Scenarios

This subapplication is particularly useful in control and automation systems where a single AUDI-typed signal must be selected from a set of predefined values based on asynchronous events. Typical use cases include:

- **Mode Selection**: Switching between different operational parameter sets in a machine controller.
- **Recipe Management**: Selecting a recipe (as a `UDINT` code) and forwarding it to a downstream AUDI-based processor.
- **Test Bench Configuration**: Choosing a test profile for a device under test.
- **Event-Driven Calibration**: Changing calibration constants on the fly by sending the appropriate event.

Because the interface is minimal (only events and `UDINT` inputs), it can be easily integrated into larger IEC 61499 applications.

Comparison with Similar Blocks

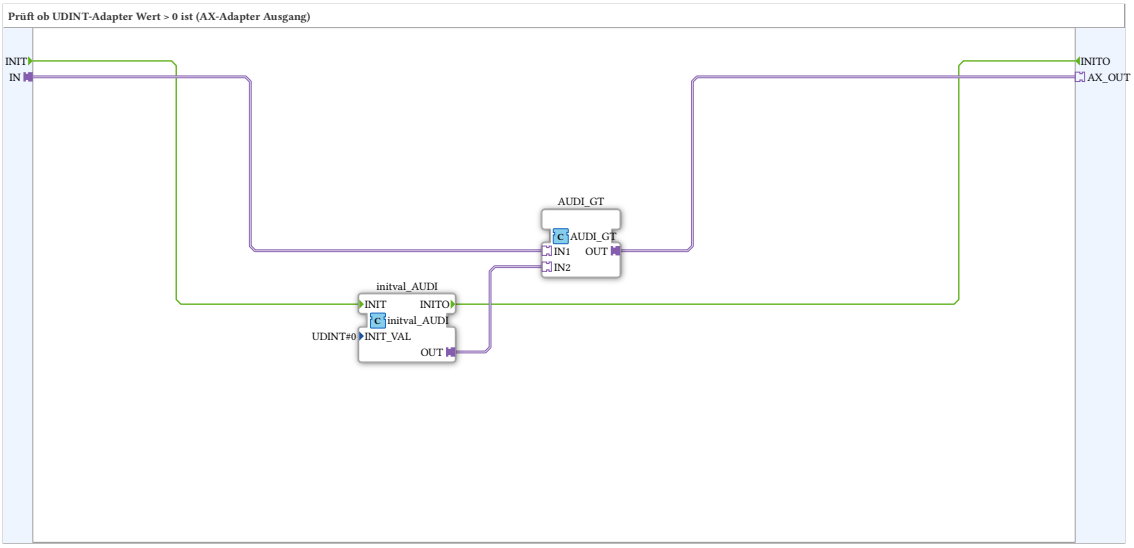
- **AUDI_AUI_MUX_3_VAL**: A three-channel version of the same subapplication. The 8-channel variant offers a larger selection count, making it suitable for applications with more options, but consumes more input events and data ports.
- **Standard MUX Function Blocks**: Typical multiplexers (e.g., based on a selector input) are not event-driven and often require a permanent selector value. This subapplication, in contrast, uses events to trigger the selection, which simplifies integration in event-oriented designs.
- **Simple Selector Adapters**: Some libraries provide direct adapter multiplexers without event synchronization. Here, the event input ensures that the output only changes at explicit trigger points, avoiding glitches or race conditions.

The key advantage of this subapplication is the combination of event-based triggering, typed AUDI adapter output, and the internal separation of event and adapter logic.

Conclusion

AUDI_AUI_MUX_8_VAL is a compact yet powerful subapplication that provides clean, event-driven multiplexing of up to eight `UDINT` values into a single AUDI adapter output. Its modular internal structure and stateless behaviour make it both reliable and easy to integrate into complex IEC 61499 systems. With a simple interface and clear semantics, it is an excellent choice for any application requiring event-synchronised signal selection in AUDI-based environments.

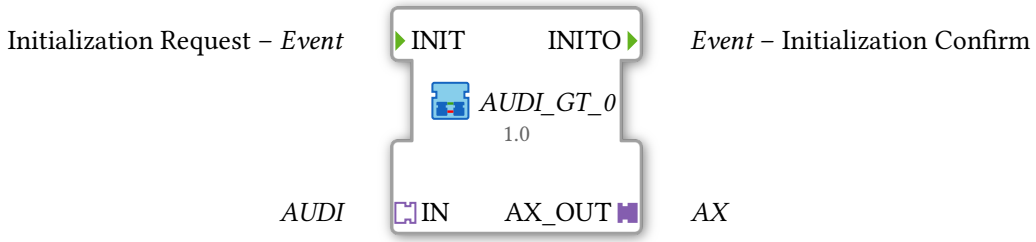
AUDI_GT_0



• – ○ • – ○ • – ○ •

Introduction

`AUDI_GT_0` checks whether a UDINT adapter value is greater than 0 and outputs the result as a boolean AX adapter signal. A typical input is an unsigned 32-bit counter value. The block is built identically for every numeric type (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Interface

- **Event input**: `INIT` (initialization)
- **Event output**: `INITO` (initialization finished, comes from the internal `initval_AUDI`)
- **Socket**: `IN` (`AUDI`) — the value compared against 0
- **Plug**: `AX_OUT` (`AX`) — TRUE when `IN > 0`

Function Blocks (FBs) Used

Sub-blocks: AUDI_GT_0

- **Type**: SubAppType
- **Internal FBs used**:
  - **initval_AUDI**: `adapter::types::unidirectional::AUDI::initval::initval_AUDI` — supplies the fixed comparison constant `UDINT#0` as an adapter value (`INIT_VAL=UDINT#0`). An adapter socket cannot take the constant as a parameter, so it is fed in through this block.
  - **AUDI_GT**: `adapter::iec61131::comparison::AUDI_GT` — adapter-native comparison `IN1 > IN2`.
- **Functionality**: `IN` ⟶ `AUDI_GT.IN1`; constant 0 ⟶ `AUDI_GT.IN2`; result ⟶ `AX_OUT`.

Program Flow and Connections

1. `INIT` ⟶ `initval_AUDI.INIT`; `initval_AUDI.INITO` ⟶ `INITO`.
2. `IN` (adapter) ⟶ `AUDI_GT.IN1`.
3. `initval_AUDI.OUT` (constant `UDINT#0`) ⟶ `AUDI_GT.IN2`.
4. `AUDI_GT.OUT` ⟶ `AX_OUT` (adapter).

Application Scenarios

- Converting an unsigned 32-bit counter value into a simple boolean adapter signal for VT status displays or enable logic.

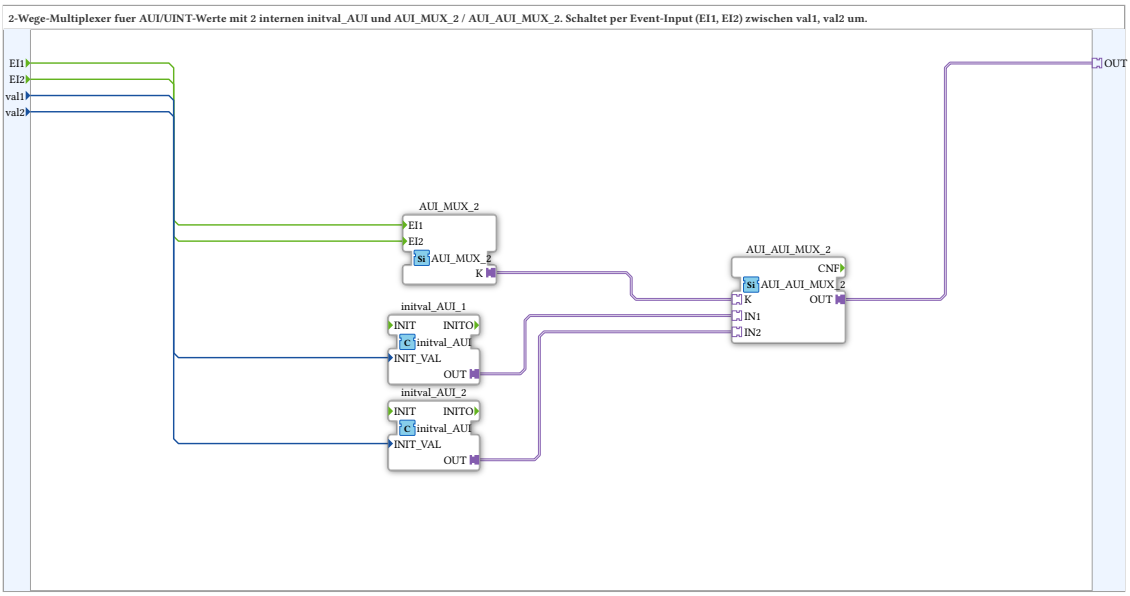
Summary

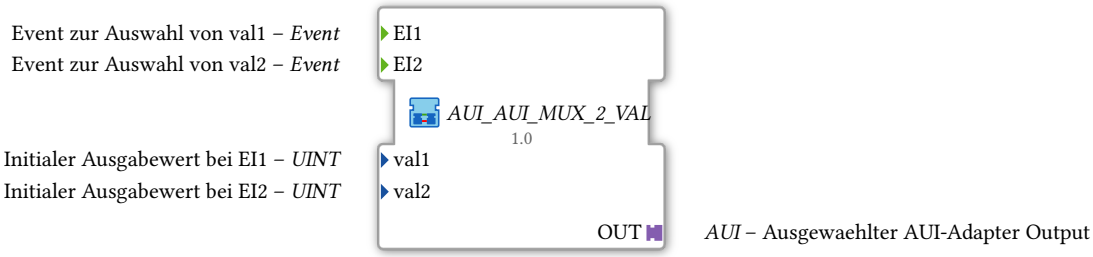
`AUDI_GT_0` is a compact wrapper around the adapter-native comparison `AUDI_GT` — converts "UDINT > 0" directly into an AX adapter signal, so the caller does not have to create and wire the zero constant itself.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AUI_AUI_MUX_2_VAL





• – ○ • – ○ • – ○ •

Introduction

The **AUI_AUI_MUX_2_VAL** subapplication implements a 2-way multiplexer for AUI adapter values. It accepts two `UINT` input values and two event inputs. Depending on which event is triggered, one of the two input values is converted into an AUI value and provided at the `OUT` adapter plug.

The internal structure combines:

- two `initval_AUI` instances to convert `UINT` values into AUI adapter outputs,
- an `AUI_MUX_2` block for event handling,
- an `AUI_AUI_MUX_2` block for selecting between two AUI input values.

This makes the subapplication a convenient, encapsulated solution for selecting between two AUI values based on events.

• – ○ • – ○ • – ○ •

Interface Structure

Event Inputs

Name	Type	Comment
`EI1`	`Event`	Event to select `val1`
`EI2`	`Event`	Event to select `val2`

Event Outputs

None.

Data Inputs

Name	Type	Comment
`val1`	`UINT`	Initial output value when `EI1` is triggered
`val2`	`UINT`	Initial output value when `EI2` is triggered

Data Outputs

None.

Adapters

Name	Type	Comment
`OUT`	`adapter::types::unidirectional::AUI`	Selected AUI adapter output

• – ◦ • – ◦ • – ◦ •

Functionality

The subapplication acts as a 2-input multiplexer for AUI adapter values:

1. The `UINT` values `val1` and `val2` are connected to two internal `initval_AUI` blocks. These blocks convert the `UINT` values into internal AUI adapter signals.
2. The event inputs `EI1` and `EI2` are connected to the internal `AUI_MUX_2` block. This block determines which input should be selected.
3. The selection information is forwarded to `AUI_AUI_MUX_2`, which receives the two prepared AUI values on its `IN1` and `IN2` adapter inputs.
4. The selected AUI value is routed to the external `OUT` adapter plug.

In practice:

- Triggering `EI1` routes `val1` to `OUT`.
- Triggering `EI2` routes `val2` to `OUT`.

• – ◦ • – ◦ • – ◦ •

Technical Features

- Implements a 2-way multiplexer for AUI adapter values.
- Accepts simple `UINT` values instead of requiring the caller to provide AUI adapter inputs.
- Uses internal `initval_AUI` blocks for `UINT`-to-AUI conversion and initialization.
- Uses `AUI_MUX_2` for event-controlled selection.
- Uses `AUI_AUI_MUX_2` for final adapter output selection.
- Provides a single unidirectional AUI adapter output.
- Encapsulates the complete multiplexing logic inside a reusable subapplication.

• – ◦ • – ◦ • – ◦ •

State Overview

The subapplication does not expose an explicit state machine, but its behavior can be described as a two-state selector:

Event	Selected Input	Output
`EI1`	`val1`	AUI representation of `val1` via `OUT`
`EI2`	`val2`	AUI representation of `val2` via `OUT`
None	Last selection remains active	Depends on the last received event

After one event has been processed, the selected AUI value remains available at `OUT` until the other event is triggered.

• – ◦ • – ◦ • – ◦ •

Application Scenarios

The AUI_AUI_MUX_2_VAL subapplication is useful in IEC 61499 applications where:

- Two predefined UINT values must be selected dynamically by events.
- AUI adapter connections are used for data exchange between function blocks.
- A compact and reusable multiplexer is needed for AUI-based signals.
- Two alternative configuration values must be switched, such as:
  - manual and automatic setpoints,
  - normal and backup values,
  - two operating modes with different parameters.

• – ◦ • – ◦ • – ◦ •

Comparison with Similar Blocks

Block / Subapplication	Data Type	Selection Mechanism	Additional Features
Standard MUX	Basic data types	Boolean or event selection	No AUI adapter support
AUI_MUX_2	AUI	Event inputs	Handles AUI event selection
AUI_AUI_MUX_2	AUI	Internal AUI selection	Requires already available AUI inputs
AUI_AUI_MUX_2_VAL	UINT inputs, AUI output	EI1 / EI2 events	Converts UINT values to AUI internally

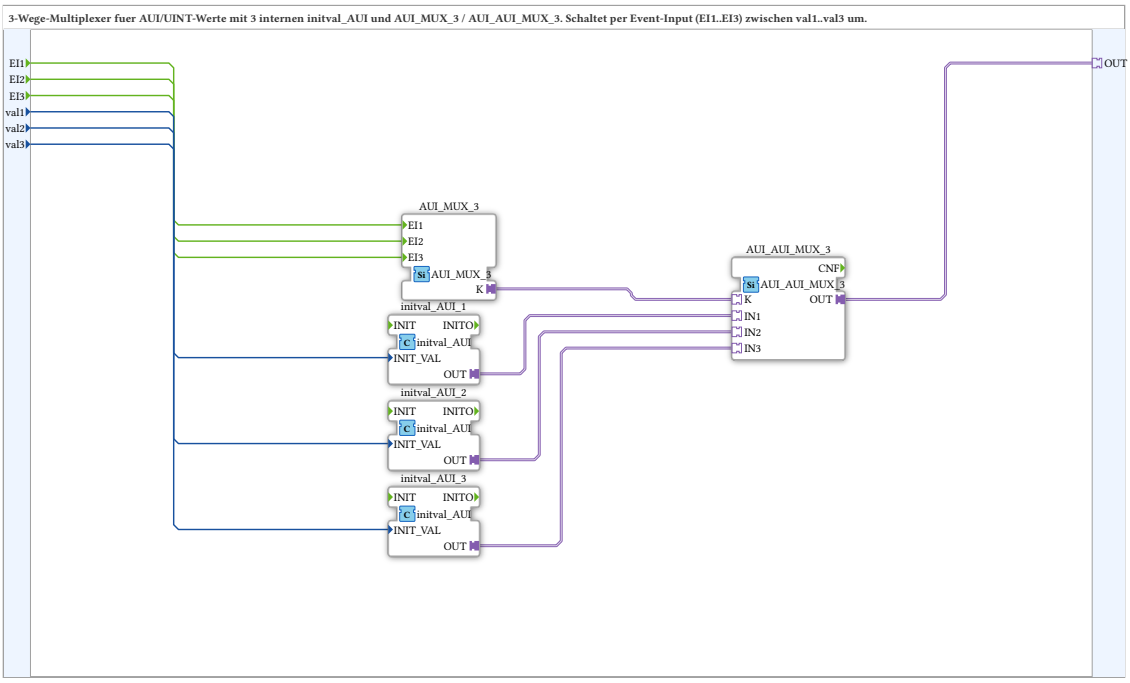
Compared to a direct AUI multiplexer, this subapplication adds the convenience of using plain UINT values while keeping the AUI adapter interface at the output.

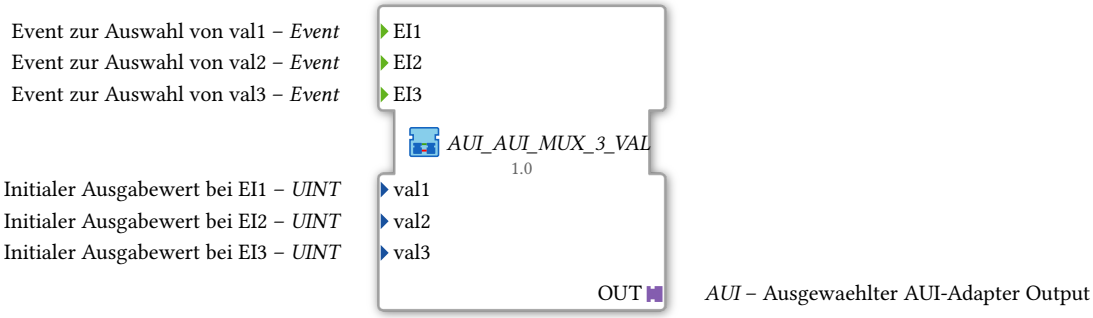
• – ◦ • – ◦ • – ◦ •

Conclusion

AUI_AUI_MUX_2_VAL is a compact and reusable subapplication for event-driven selection between two AUI values. It hides the internal AUI initialization and selection logic behind a simple interface consisting of two events, two UINT inputs, and one AUI adapter output. It is particularly suitable for applications that require fast switching between two predefined values within an AUI-based IEC 61499 environment.

AUI_AUI_MUX_3_VAL





• – ○ • – ○ • – ○ •

Introduction

The **AUI_AUI_MUX_3_VAL** is a composite subapplication that implements a 3-way multiplexer for AUI (Asynchronous Unified Interface) values. It allows selecting one of three AUI input channels via event triggers, while also providing the ability to set initial output values using standard UINT data inputs. The subapplication encapsulates the logic of a dedicated AUI multiplexer and three initial-value adapters, simplifying the integration of AUI-based selection logic into larger systems.

Interface Structure

Event Inputs

- **EI1** – Event to select the first input channel (val1).
- **EI2** – Event to select the second input channel (val2).
- **EI3** – Event to select the third input channel (val3).

Each event triggers the multiplexer to route the corresponding AUI value to the output.

Event Outputs

No explicit event outputs are defined. The subapplication operates purely on event-driven selection; the result is reflected on the AUI output adapter.

Data Inputs

- **val1** (UINT) – The initial value for the first AUI channel.

- **val2** (UINT) – The initial value for the second AUI channel.
- **val3** (UINT) – The initial value for the third AUI channel.

These values are converted to AUI format internally and serve as the default outputs for each corresponding selection event.

Data Outputs

No direct data outputs are exposed. The result is delivered via the AUI output adapter.

Adapters

- **OUT** (AUI, unidirectional) – The selected AUI output. This adapter carries the active channel's value after an event is received.

Functionality

The subapplication combines three main internal components:

1. **AUI_MUX_3** – An event-driven multiplexer control block that accepts the three event inputs (EI1, EI2, EI3) and generates a single control signal (K) indicating which channel is active.
2. **initval_AUI (×3)** – Three instances that accept a UINT initial value (`INIT_VAL`) and convert it into an AUI-compatible output (OUT). Each instance is fed with one of the `val1..val3` inputs.
3. **AUI_AUI_MUX_3** – The core AUI multiplexer that receives three AUI inputs (IN1, IN2, IN3) and the control signal (K). Depending on the control value, it selects one of the inputs and forwards it to the output (OUT).

The internal flow is as follows:

- The UINT values (`val1`, `val2`, `val3`) are passed to the respective `initval_AUI` blocks, which convert them into AUI signals.
- The event inputs (`EI1`, `EI2`, `EI3`) are fed into `AUI_MUX_3`, which produces a control signal based on which event was triggered.
- The control signal and the three AUI inputs are routed to `AUI_AUI_MUX_3`, which selects the appropriate channel and passes it to the external `OUT` adapter.

Thus, upon receiving an event, the subapplication immediately outputs the AUI representation of the corresponding `valN` value.

Technical Features

- **Modular design** – Reuses standard AUI and initval blocks, ensuring interoperability and ease of maintenance.
- **Event-driven selection** – No continuous polling; output changes only when one of the three events occurs.
- **Initial value support** – The UINT inputs allow the subapplication to be initialised with specific values that are converted and made available as AUI signals.
- **Unidirectional AUI output** – The OUT adapter is unidirectional, suitable for simple point-to-point data transfer.
- **Standardised interfaces** – All inputs and outputs adhere to the 61499-2 standard, making the block compatible with other IEC 61499-compliant tools.

State Overview

The subapplication does not maintain internal state beyond the current selection. It behaves as a combinational selector once an event is received:

- **Idle state** – No event has occurred; output holds its last value.
- **Selection states** – After EI1, EI2, or EI3, the output reflects the corresponding AUI value.

There is no explicit reset or default selection; the output remains on the last selected channel until a new event arrives.

Application Scenarios

- **Process control** – Selecting one of several sensor values (converted to AUI) based on external trigger events.
- **User-interface routing** – Choosing between different data streams (e.g., from different sources) based on operator inputs.

- **Initialization and failover** – Providing a pre-defined default value on each channel, with the ability to switch quickly when events occur.
- **System diagnostics** – Multiplexing diagnostic data from multiple subsystems into a single monitoring point.

Comparison with Similar Blocks

Compared to a simple multiplexer that directly takes UINT inputs and outputs a UINT, this block introduces AUI conversion and event-driven behaviour. It differs from a basic selector in that it:

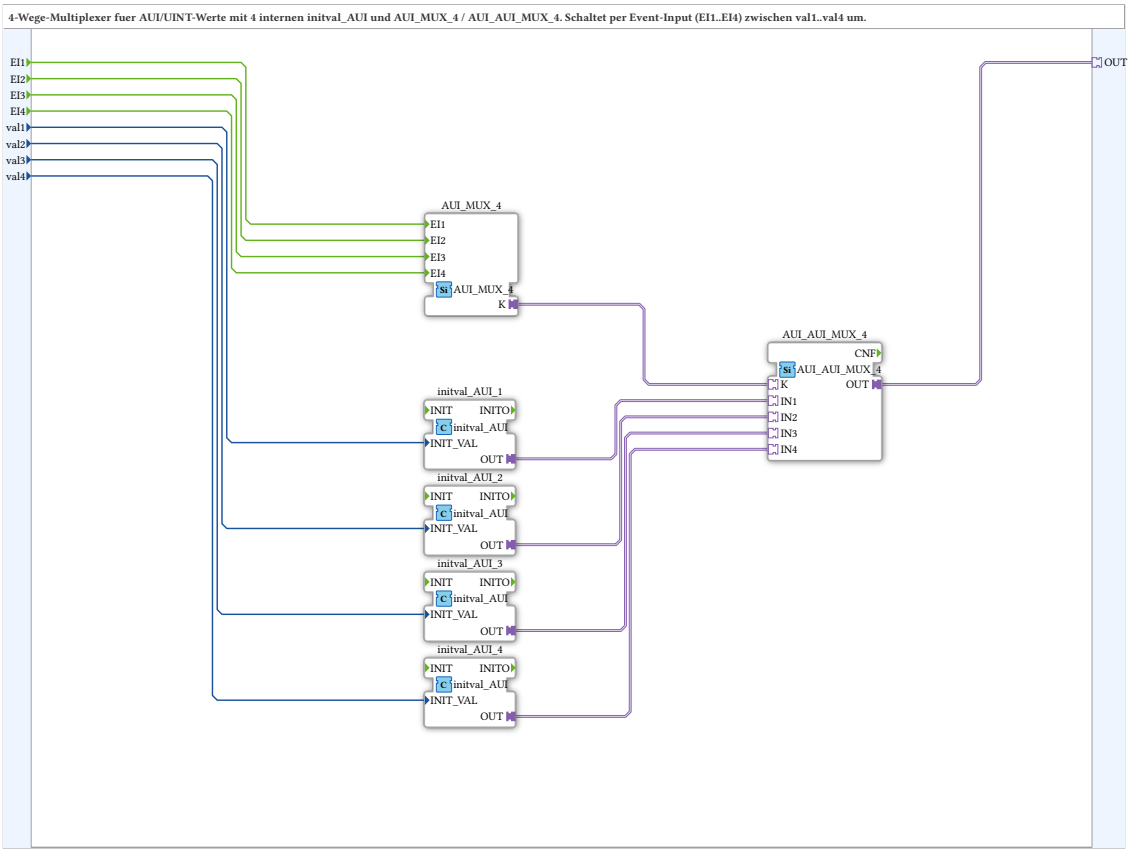
- Operates on AUI adapters rather than raw data types.
- Includes explicit initial value handling via `initval_AUI`, allowing each channel to have a predefined starting value.
- Is fully event-driven, whereas some multiplexers may use continuous data inputs or a selection input.

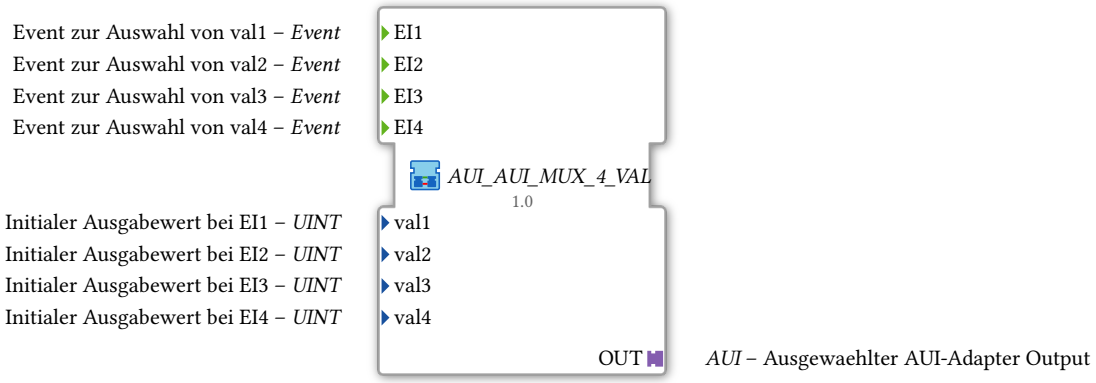
When compared to a standalone `AUI_MUX_3` block, this subapplication adds the UINT-to-AUI initialisation layer, making it easier to use in systems that work with standard integer values while still maintaining an AUI output.

Conclusion

The **AUI_AUI_MUX_3_VAL** subapplication provides a clean, event-driven solution for multiplexing AUI signals while supporting UINT-based initial values. Its modular internal structure leverages standard 4diac blocks, ensuring reliability and compatibility. With its simple interface and flexible configuration, it is well suited for a wide range of industrial automation and control applications where AUI communication and event-based selection are required.

AUI_AUI_MUX_4_VAL





• – ◦ • – ◦ • – ◦ •

Introduction

AUI_AUI_MUX_4_VAL is a composite subapplication that implements a 4-way multiplexer specifically for AUI (Application User Interface) adapter values of type UINT. It combines a set of internal initial value generators (`initval_AUI`) with an event-driven selection mechanism (`AUI_MUX_4`) and a final adapter switch (`AUI_AUI_MUX_4`). The subapplication is designed to select one of four UINT input values and route it to a single AUI output adapter based on which of the four event inputs is activated.

Interface Structure

Event Inputs

Event	Description
`EI1`	Event to select the value `val1`
`EI2`	Event to select the value `val2`
`EI3`	Event to select the value `val3`
`EI4`	Event to select the value `val4`

Event Outputs

None – the subapplication does not provide event outputs.

Data Inputs

Data Input	Type	Description
`val1`	UINT	UINT value to be output on `EI1`
`val2`	UINT	UINT value to be output on `EI2`
`val3`	UINT	UINT value to be output on `EI3`
`val4`	UINT	UINT value to be output on `EI4`

Data Outputs

None – all data is passed through the adapter interface.

Adapters

Adapter	Type	Description
`OUT`	`adapter::types::unidirectional::AUI`	Selected UINT value as an AUI adapter output

Functionality

The subapplication operates as follows:

1. The four UINT input values (`val1` ... `val4`) are fed into four internal `initval_AUI` blocks, which convert them into AUI adapter values (likely by wrapping the UINT into the AUI adapter structure).
2. The internal `AUI_MUX_4` block receives the event inputs (`EI1` ... `EI4`) and, based on the event that occurs, generates a selection signal (`K`) that indicates which input channel is to be chosen.
3. The `AUI_AUI_MUX_4` comparator/switch uses this selection signal to forward the corresponding AUI adapter value from the four internal `initval_AUI` blocks to the single output adapter `OUT`.

Thus, whenever an event `EIx` is triggered, the value `valx` becomes available on the output adapter `OUT`. Only one event is processed at a time; the last event received determines the active output.

Technical Features

- **Event-driven selection:** The output is only updated when one of the four event inputs is activated, avoiding continuous polling.
- **Internal initial value handling:** The `initval_AUI` blocks ensure that each input value is properly converted to the AUI adapter format before multiplexing.
- **Adapter-based output:** The result is provided as an AUI adapter, allowing seamless integration with other AUI-compatible components.
- **Modular structure:** The subapplication is built from reusable blocks (`AUI_MUX_4`, `AUI_AUI_MUX_4`, `initval_AUI`), promoting maintainability and reusability.

State Overview

The subapplication does not maintain an internal state machine in the traditional sense. Its behavior is purely combinational with respect to the most recent event: the output reflects the value corresponding to the last event that was received. There is no explicit state storage; the selection is performed asynchronously by the internal adapter switch.

Application Scenarios

- **Multi-source data selection:** When a system must choose between several UINT-based data streams (e.g., sensor readings, configuration values) and output the selected one via an AUI adapter.
- **Control scenario switching:** In an automation environment where different parameter sets (val1...val4) are applied based on discrete control events (EI1...EI4).
- **AUI integration:** For systems already using AUI adapters, this subapp provides a convenient way to multiplex UINT values without leaving the AUI paradigm.

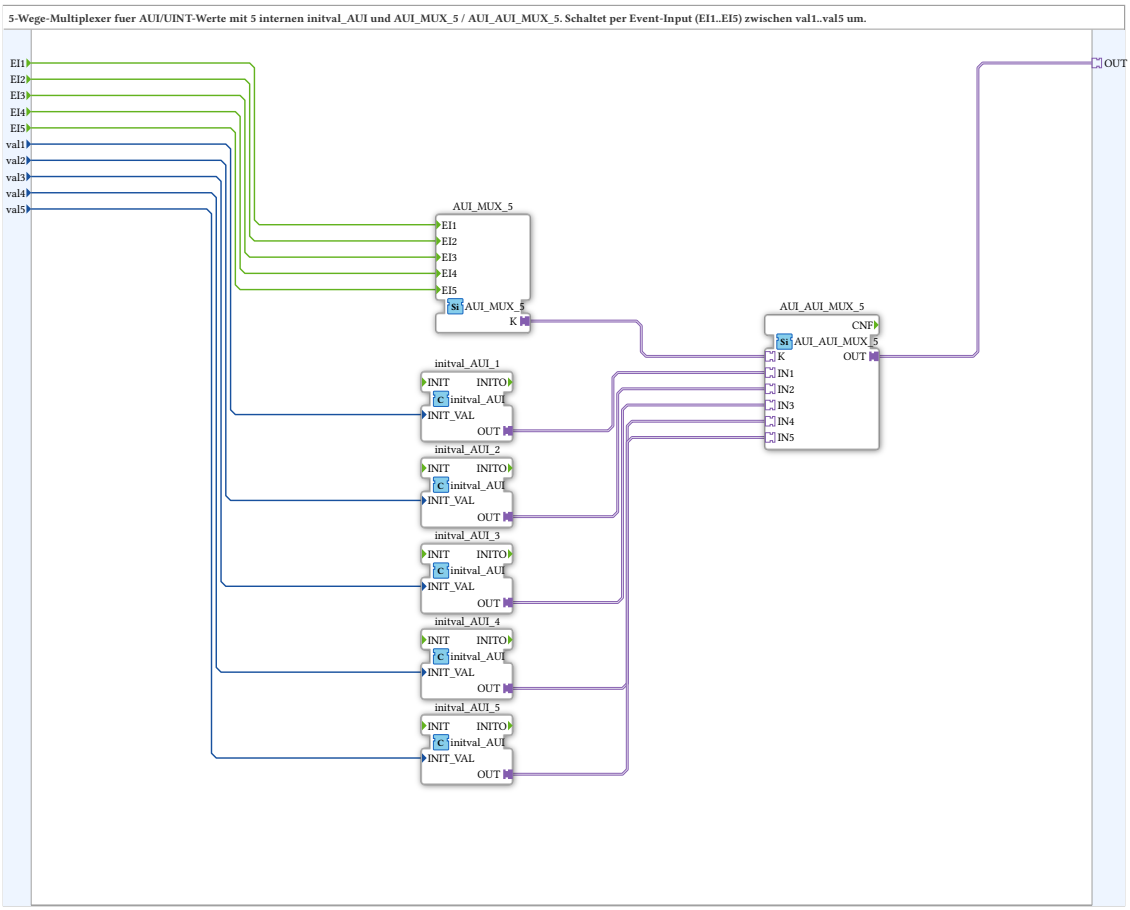
Comparison with Similar Blocks

- **AUI_AUI_MUX_4** (the internal selection block) itself is a plain adapter multiplexer that requires an additional selection signal. This subapplication **adds event inputs** and **automatic conversion from UINT to AUI**, simplifying the user interface.
- **Standard demultiplexer/multiplexer blocks** often work with plain data types and separate selection inputs; here the selection is embedded in the events, making the component self-contained and event-oriented.
- **Similar subapplications** may combine the same internal blocks but offer fewer inputs (e.g., 2-way), whereas this version supports four independent channels.

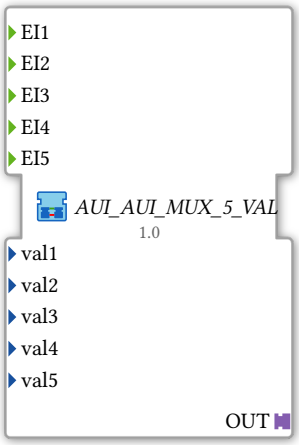
Conclusion

AUI_AUI_MUX_4_VAL provides a compact and efficient solution for selecting one of four UINT values and forwarding it as an AUI adapter output, driven entirely by events. Its internal structure reuses proven building blocks, ensuring reliability and ease of integration in IEC 61499 automation systems. The subapplication is particularly suited for scenarios where several value sources need to be switched dynamically without overcomplicating the application logic.

AUI_AUI_MUX_5_VAL



Event zur Auswahl von val1 – *Event*
Event zur Auswahl von val2 – *Event*
Event zur Auswahl von val3 – *Event*
Event zur Auswahl von val4 – *Event*
Event zur Auswahl von val5 – *Event*



Initialer Ausgabewert bei EI1 – *UINT*
Initialer Ausgabewert bei EI2 – *UINT*
Initialer Ausgabewert bei EI3 – *UINT*
Initialer Ausgabewert bei EI4 – *UINT*
Initialer Ausgabewert bei EI5 – *UINT*

AUI – Ausgewaehlter AUI-Adapter Output

• – ○ • – ○ • – ○ •

Introduction

The **AUI_AUI_MUX_5_VAL** is a composite subapplication (SubAppType) that implements a 5-way multiplexer for AUI (Application Interconnection Unit) adapter values with UINT-based initial configuration. It allows the selection of one out of five incoming AUI data streams based on a corresponding event trigger. Internally, the subapp combines three functional building blocks: an event-driven multiplexer (`AUI_MUX_5`), a data-path selection adapter (`AUI_AUI_MUX_5`), and five initialization adapters (`initval_AUI`) that convert UINT values into AUI adapter outputs. This structure provides a clean and reusable way to route AUI data in IEC 61499-based automation systems, especially in scenarios where a single output adapter must be switched between multiple sources.

Interface Structure

The subapplication exposes the following interface elements to the outside world:

Event Inputs

Name	Type	Description
EI1	Event	Event to select input value `val1`
EI2	Event	Event to select input value `val2`
EI3	Event	Event to select input value `val3`
EI4	Event	Event to select input value `val4`
EI5	Event	Event to select input value `val5`

Each event input corresponds to a dedicated selection action. Triggering one of these events causes the internal multiplexer to route the associated data value to the output.

Event Outputs

There are no event outputs in this subapplication. The selection process is purely input-driven; no completion or acknowledgment events are generated.

Data Inputs

Name	Type	Description
val1	UINT	Initial output value when EI1 is triggered
val2	UINT	Initial output value when EI2 is triggered
val3	UINT	Initial output value when EI3 is triggered
val4	UINT	Initial output value when EI4 is triggered
val5	UINT	Initial output value when EI5 is triggered

These UINT values are forwarded to the internal `initval_AUI` adapters, which convert them into AUI adapter data structures. The values are applied whenever the corresponding event input is activated.

Data Outputs

There are no direct data outputs in the interface list. The result of the multiplexing operation is delivered exclusively via the AUI adapter output.

Adapters

Name	Type	Description
OUT	`adapter::types::unidirectional::AUI`	Selected AUI adapter output containing the active data stream

The `OUT` adapter is a unidirectional AUI interface that carries the data of the currently selected input channel.

Functionality

The subapplication works as a data-selection switch with event-based control. Upon receiving an event at one of the five event inputs (`EI1` to `EI5`), the internal `AUI_MUX_5` block decodes the event and forwards a corresponding selection signal through its `K` adapter connection to the `AUI_AUI_MUX_5` block. Simultaneously, the UINT value associated with the triggered event (e.g., `val1` for `EI1`) is passed to the respective `initval_AUI` instance, which converts it into an AUI data structure. The `AUI_AUI_MUX_5` block then chooses the corresponding input (`IN1` to `IN5`) and routes its data to the `OUT` adapter.

The internal wiring is organized as follows:

- Each event input `EI1`–`EI5` is directly connected to the corresponding event input of the `AUI_MUX_5` block.
- Each UINT data input `val1`–`val5` feeds the `INIT_VAL` input of a dedicated `initval_AUI` instance.

- The five `initval_AUI` outputs (`OUT`) are connected to the `IN1`–`IN5` adapter ports of the `AUI_AUI_MUX_5` selection block.
- The selection event from `AUI_MUX_5` is transferred via its `K` adapter to the `K` port of `AUI_AUI_MUX_5`.
- The `OUT` port of the selection block is connected to the subapplication's external `OUT` adapter.

Thus, the overall behavior is a synchronous event-driven multiplexer: only one output is active at any time, and the active channel is determined exclusively by the most recent event input.

Technical Features

- **Composite subapplication**: Encapsulates a complete multiplexing logic into a single reusable component.
- **Event-driven selection**: Selection is controlled via five discrete event inputs, making the block suitable for cyclic or edge-triggered control schemas.
- **UINT-to-AUI conversion**: The internal `initval_AUI` adapters handle the conversion from simple UINT data to the compound AUI adapter format, simplifying the external interface.
- **Modular internal design**: The subapp is built from three different block types (`AUI_MUX_5`, `AUI_AUI_MUX_5`, and `initval_AUI`), each with a clear and focused responsibility, improving maintainability and testability.
- **Unidirectional AUI output**: The external adapter interface is unidirectional, which fits well in publish-subscribe or point-to-point data flow architectures.
- **No explicit state variables**: The subapp relies entirely on the internal blocks' behavior; no additional state management is required in the composite layer.

State Overview

The subapplication itself does not define an explicit state machine; its behavior is derived from the internal blocks. However, the effective operational states can be described as follows:

State	Trigger	Description
Idle	No event received	The output retains the last selected value; no switching occurs.
Channel 1 active	Event EI1	Input `val1` is converted and routed to the `OUT` adapter.
Channel 2 active	Event EI2	Input `val2` is converted and routed to the `OUT` adapter.
Channel 3 active	Event EI3	Input `val3` is converted and routed to the `OUT` adapter.
Channel 4 active	Event EI4	Input `val4` is converted and routed to the `OUT` adapter.
Channel 5 active	Event EI5	Input `val5` is converted and routed to the `OUT` adapter.

Each state transition is triggered by a rising edge of the corresponding event input. The output remains stable in the absence of new events.

Application Scenarios

- **Multi-source data routing**: In distributed automation systems where a single consumer must receive data from one out of several producers, this block can serve as a central selection switch.
- **Parameterizable initialization**: When different startup values are required for different operational modes, the UINT inputs allow runtime or configuration-time initialization of each channel.
- **Mode switching**: In machine control applications, the block can switch between different reference values (e.g., speed, position, temperature) based on operator events.
- **Test and simulation environments**: The subapp can be used to inject different test values into a processing chain by simply toggling the event inputs.

Comparison with Similar Blocks

Feature	AUI_AUI_MUX_5_VAL	Simple UINT MUX	Generic AUI MUX (without init)
Input data type	UINT (converted internally to AUI)	UINT only	AUI (requires pre-configured sources)
Output data type	AUI adapter	UINT	AUI adapter

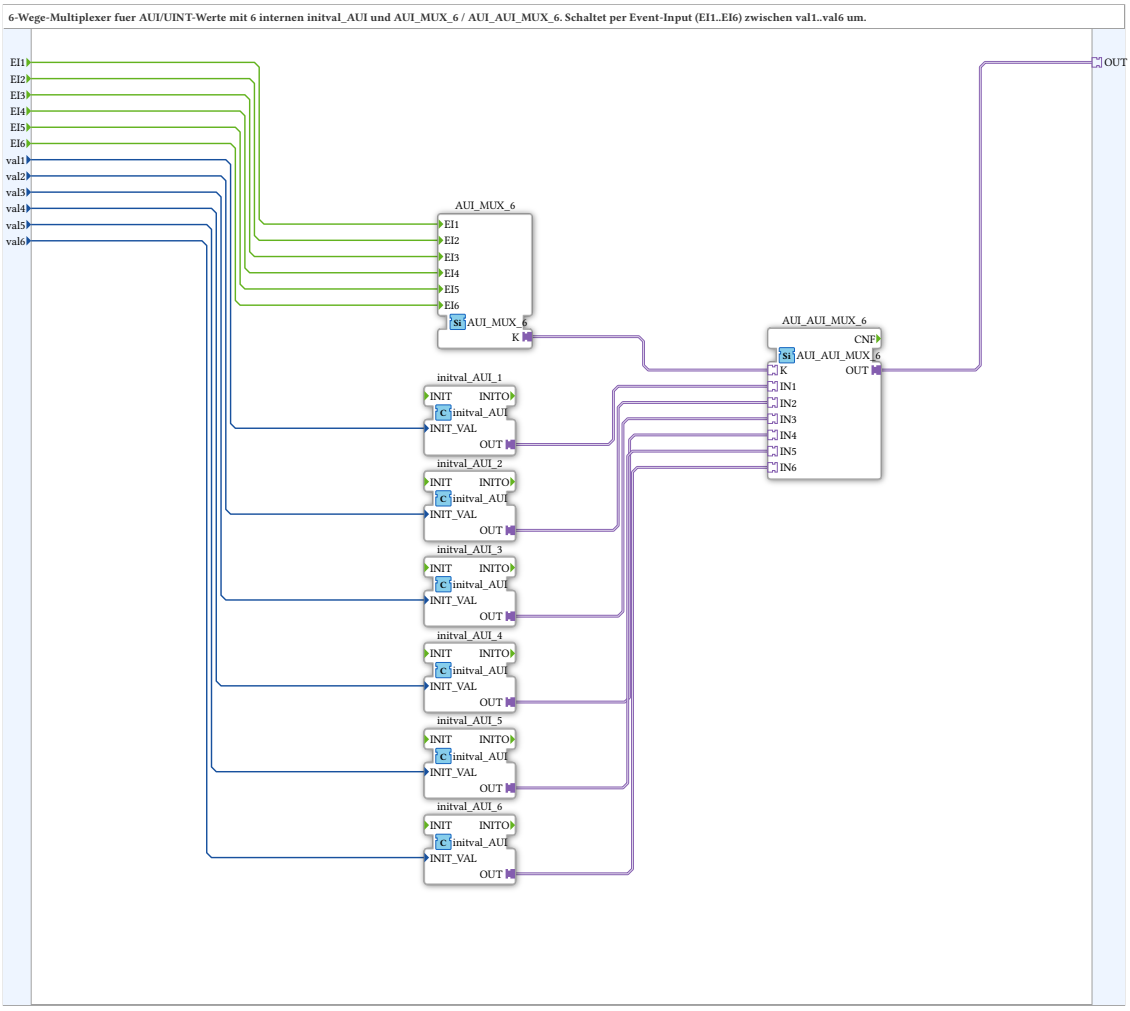
Event inputs	5 (EI1–EI5)	Typically 1 (with selector input)	5 (EI1–EI5)
Internal initialization	Yes, via `initval_AUI` blocks	No	No
Type conversion handling	Integrated, transparent to the user	Not applicable	Not applicable
Use case complexity	Medium – suitable for composite control	Low – simple data selection	Medium – requires external AUI sources

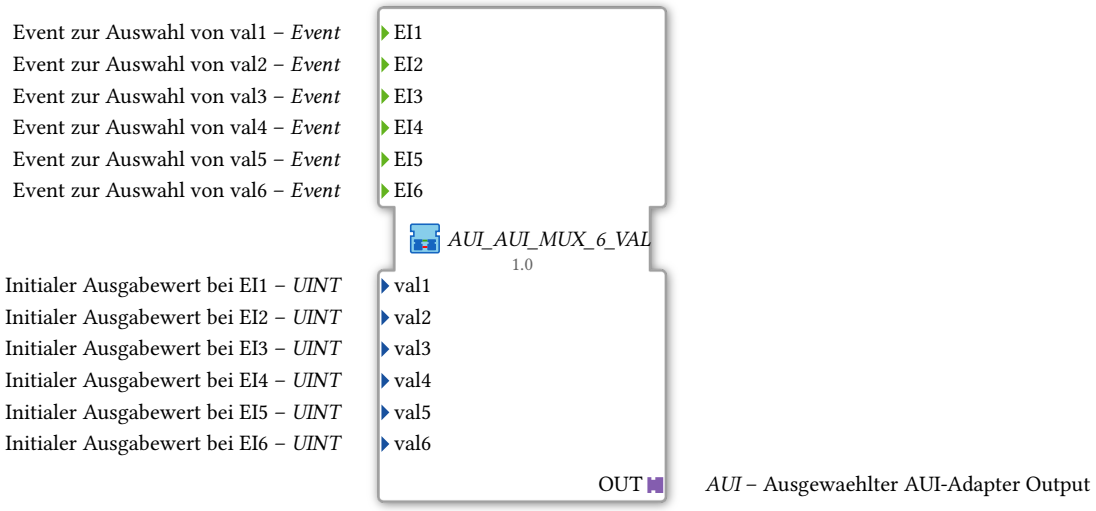
Compared to a plain UINT multiplexer, this subapp adds the advantage of AUI adapter compatibility. Compared to a generic AUI mux, it offers built-in UINT initialization, which reduces external wiring and makes it easier to configure values at design time or from HMI/PLC variables.

Conclusion

The `AUI_AUI_MUX_5_VAL` subapplication provides a robust and modular solution for selecting between five AUI data streams based on event triggers. Its internal architecture cleanly separates event decoding, data conversion, and data selection, resulting in a highly reusable component. The integration of UINT-based initialization values simplifies configuration and makes the block particularly useful in applications where parameterized startup values or dynamic channel switching are required. By maintaining a purely unidirectional external interface and relying on standard 4diac adapter patterns, the block fits seamlessly into IEC 61499-compliant engineering workflows and can be deployed across a wide range of automation scenarios.

AUI_AUI_MUX_6_VAL





• – ◦ • – ◦ • – ◦ •

Introduction

AUI_AUI_MUX_6_VAL is a composite subapplication that implements a 6-way multiplexer for UINT values, converting them into AUI adapter outputs. It combines six input events (EI1…EI6) with six data inputs (val1…val6) and a single AUI output adapter. Internally, it uses three functional blocks: an event-driven multiplexer (AUI_MUX_6), an AUI selection unit (AUI_AUI_MUX_6), and six instances of the initialiser block initval_AUI. This design allows the selection of one of six UINT values via event triggers and provides the selected value as an AUI adapter output.

Interface Structure

Event Inputs

Event	Type	Comment
EI1	Event	Event to select val1
EI2	Event	Event to select val2
EI3	Event	Event to select val3
EI4	Event	Event to select val4
EI5	Event	Event to select val5
EI6	Event	Event to select val6

Event Outputs

No event outputs are defined.

Data Inputs

Data Input	Type	Comment
val1	UINT	Initial output value when EI1 is triggered
val2	UINT	Initial output value when EI2 is triggered
val3	UINT	Initial output value when EI3 is triggered
val4	UINT	Initial output value when EI4 is triggered
val5	UINT	Initial output value when EI5 is triggered
val6	UINT	Initial output value when EI6 is triggered

Data Outputs

No direct data outputs are present; all output data is carried through the AUI adapter.

Adapters

Adapter	Type	Comment
OUT	AUI (adapter::types::unidirectional::AUI)	Selected AUI adapter output

Functionality

The subapplication operates as a 6-to-1 selector for AUI-encoded UINT values. When an event input (e.g., EI1) is activated, the corresponding UINT value (val1) is loaded into the respective internal initval_AUI block. These initval_AUI blocks convert the plain UINT data into an AUI adapter stream. The event multiplexer (AUI_MUX_6) forwards the triggering event to the selection block (AUI_AUI_MUX_6), which then switches the appropriate input adapter (from one of the six initval_AUI blocks) to the output adapter. Thus, the selected UINT value becomes available on the OUT adapter only after the corresponding event fires.

Technical Features

- **Modular design**: Combines dedicated multiplexer, selection, and initialisation blocks.
- **Adapter-based I/O**: All data exchange uses AUI type adapters, ensuring type safety and compatibility with other AUI-compatible blocks.
- **Event-driven selection**: Each input event directly controls the multiplexer, resulting in deterministic switching behaviour.
- **Parameterised initial values**: All six data inputs are UINT and are internally converted to AUI via initval_AUI blocks.
- **Scalable architecture**: The pattern can be extended to more channels by adding parallel blocks and events.

State Overview

The subapplication does not expose an explicit state machine. Its behaviour is purely combinational and event-driven: the state is implicitly defined by which event has been most recently received. Each event latches the corresponding UINT value into the AUI output. There is no internal persistent state beyond the current selection; however, the initval_AUI blocks hold the converted AUI values until overwritten by a new event.

Application Scenarios

- **Control systems**: Selecting between multiple preset speeds, pressures, or setpoints based on discrete control signals.
- **Data routing**: Forwarding one of several measured UINT values to a common monitoring adapter.
- **Configuration management**: Loading predefined parameter sets into an AUI-based process.
- **Test benches**: Simulating multi-channel sensor selection for validation purposes.

Comparison with Similar Blocks

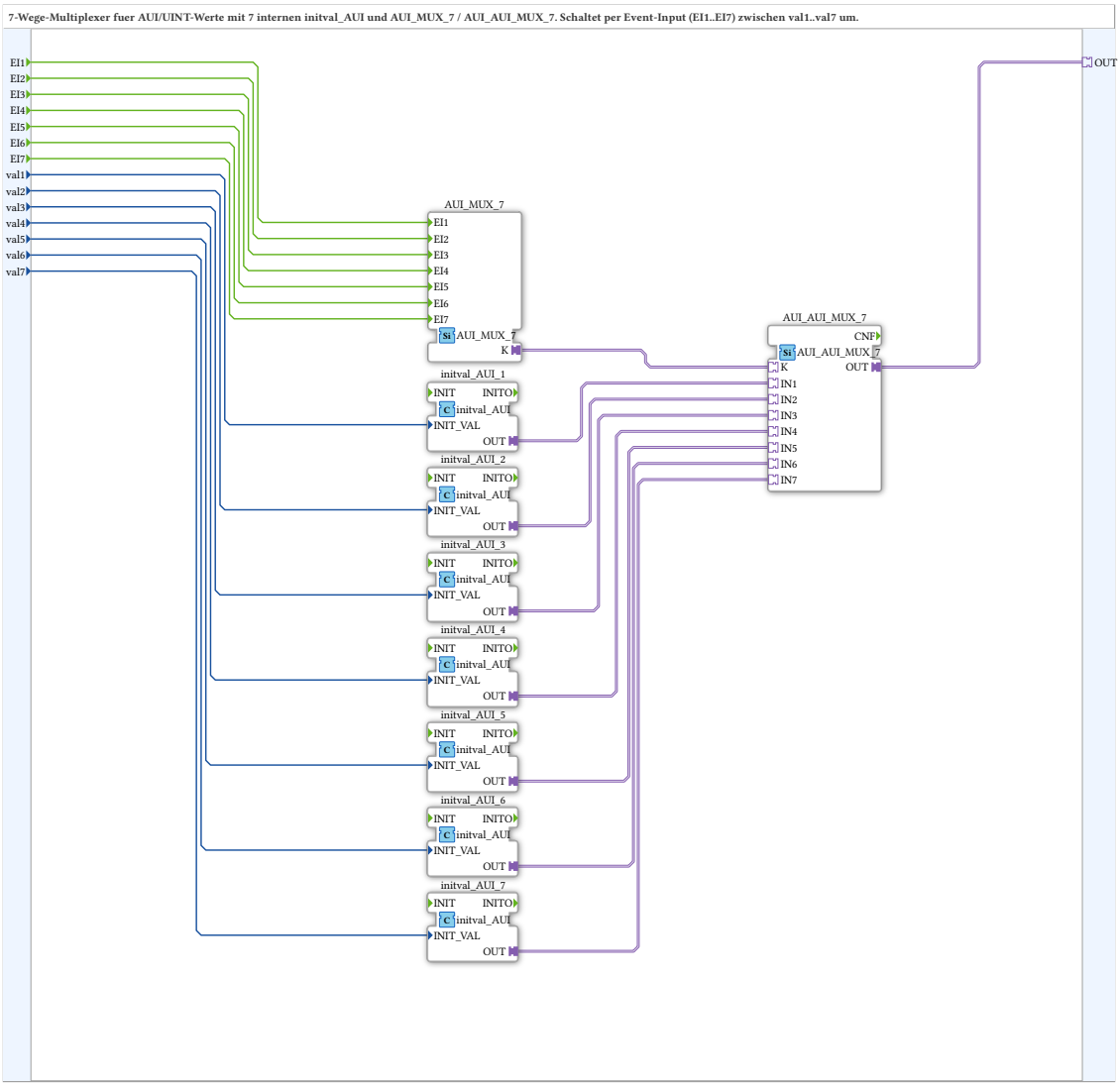
Feature	AUI_AUI_MUX_6_VAL	AUI_MUX_6 (standalone)	AUI_AUI_MUX_6 (standalone)
Input data type	UINT (converted to AUI internally)	UINT (via events)	AUI (direct adapters)
Output type	AUI adapter	AUI adapter	AUI adapter
Number of channels	6	6	6
Internal initialisers	Yes (initval_AUI blocks)	No	No
Event handling	Yes (EI1…EI6)	Yes (EI1…EI6)	(depends on implementation)

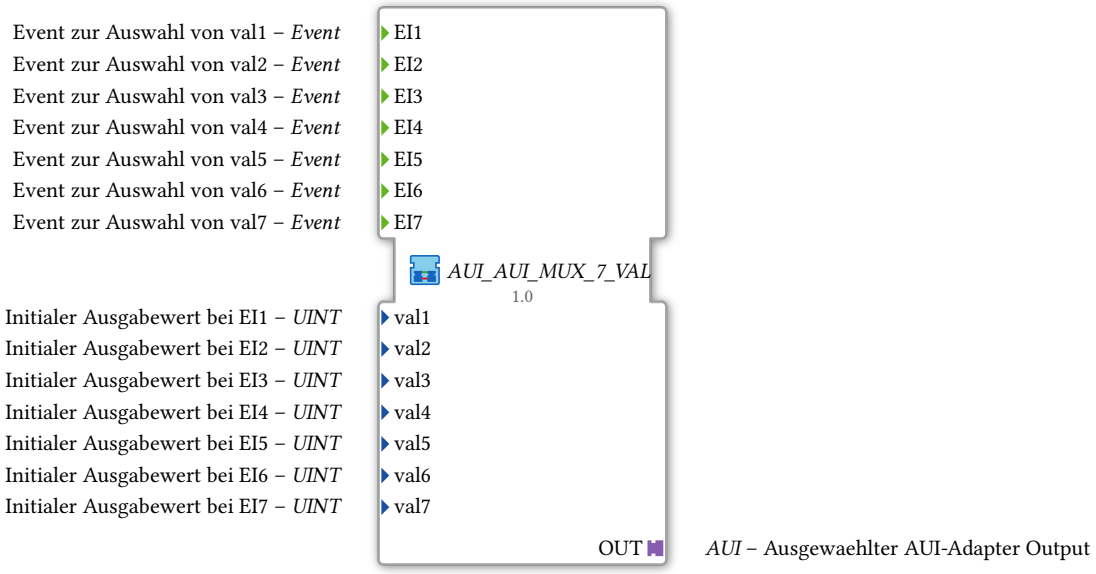
The combined subapplication integrates the initialisation and selection logic, simplifying external wiring and ensuring consistent AUI conversion for all input values.

Conclusion

AUI_AUI_MUX_6_VAL is a compact and reusable subapplication that provides a reliable 6-way AUI multiplexer with UINT inputs. By encapsulating the conversion and selection logic, it reduces complexity for application developers and promotes clear, event-driven control. Its design aligns with IEC 61499 standards and is well suited for industrial automation scenarios where multiple analogue or digital values must be selectively routed to a common adapter output.

AUI_AUI_MUX_7_VAL





• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_AUI_MUX_7_VAL** is a composite subapplication that implements a 7-way multiplexer for AUI (Adapter Unified Interface) values derived from UINT inputs. It selects one of seven UINT values (`val1` ... `val7`) based on a corresponding event input (`EI1` ... `EI7`) and provides the selected value as a unidirectional AUI adapter output (`OUT`). Internally, the subapplication uses pre-built function blocks to convert the UINT data to AUI format and to perform the actual selection, offering a ready-to-use, reusable component for scenarios requiring event-driven data selection in IEC 61499-based systems.

Interface Structure

The subapplication exposes a clean interface comprising event inputs, data inputs, and an adapter output. No event outputs or additional adapters are present.

Event Inputs

Name	Type	Comment
EI1	Event	Event to select `val1`
EI2	Event	Event to select `val2`
EI3	Event	Event to select `val3`
EI4	Event	Event to select `val4`
EI5	Event	Event to select `val5`
EI6	Event	Event to select `val6`
EI7	Event	Event to select `val7`

Event Outputs

None.

Data Inputs

Name	Type	Comment
val1	UINT	Value to be output when `EI1` occurs
val2	UINT	Value to be output when `EI2` occurs

val3	UINT	Value to be output when EI3 occurs
val4	UINT	Value to be output when EI4 occurs
val5	UINT	Value to be output when EI5 occurs
val6	UINT	Value to be output when EI6 occurs
val7	UINT	Value to be output when EI7 occurs

Data Outputs

None.

Adapters

Name	Type	Direction	Comment
OUT	adapter::types::unidirectional::AUI	Output (Plug)	Selected AUI adapter output

Functionality

The subapplication acts as an event-driven selector. When an event arrives on one of the event inputs (EI1 … EI7), the corresponding UINT data input (val1 … val7) is converted into an AUI adapter value and propagated to the output adapter OUT. The selection is entirely event-driven; no continuous data sampling occurs—the output is only updated upon the arrival of a triggering event.

Internally, the subapplication is composed of:

- Seven initval_AUI function blocks that convert each UINT input into an AUI value.
- An AUI_MUX_7 block that handles the event selection logic (likely routing the triggering event to the appropriate internal channel).
- An AUI_AUI_MUX_7 block that performs the final selection of the AUI values from the seven converted inputs and forwards the chosen one to the output adapter OUT.

The event input is mapped directly to the selection mechanism, ensuring that a single event selects exactly one data source. The combination of these internal blocks yields a robust, reusable multiplexer with a simple external interface.

Technical Features

- **Event-driven selection**: Only one event determines the active input; no polling or continuous evaluation.
- **UINT to AUI conversion**: Each UINT value is converted internally to an AUI adapter type, enabling seamless integration with AUI-based communication patterns.
- **7-channel multiplexing**: Supports up to seven distinct input values, each selectable via its own event.
- **Unidirectional AUI output**: The output is a unidirectional adapter, suitable for scenarios where data flows from the subapplication to a connected consumer.
- **Reuse of standard blocks**: The subapplication leverages existing, tested function blocks (initval_AUI, AUI_MUX_7, AUI_AUI_MUX_7) to reduce development and verification effort.
- **No hidden state**: The subapplication is stateless; the output depends solely on the most recent event and the current input values.

State Overview

The subapplication does not maintain any internal state. Its behaviour is purely combinatorial with respect to the event inputs—the output is updated synchronously when an event occurs. There is no startup, idle, or error state; the block is always ready to respond to events. This simplifies integration and testing, as no re-initialisation is required.

Application Scenarios

- **Multi-source data selection**: In systems where multiple data sources (e.g., sensor readings, configuration values) must be chosen on demand, this subapplication provides a clean event-driven interface.
- **AUI-based communication**: When the surrounding application uses AUI adapters for data exchange, this component bridges the gap between simple UINT values and the AUI data type.

- **Rapid prototyping**: Because the subapplication encapsulates the conversion and selection logic, it can be dropped into a 4diac-based design with minimal configuration.
- **Distributed control systems**: The event-driven nature aligns well with IEC 61499 event-based execution models, making the subapplication suitable for distributed automation and process control.

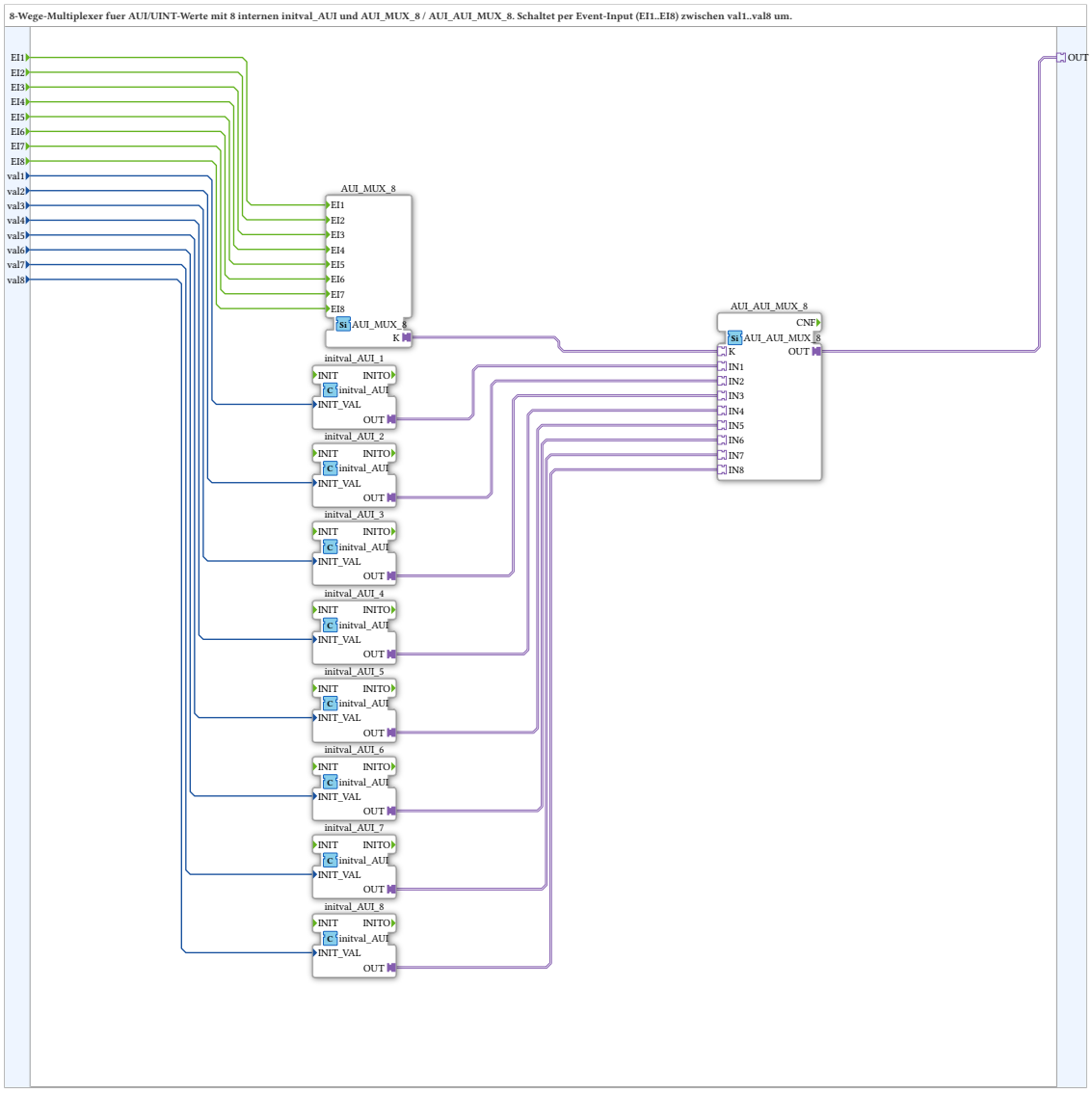
Comparison with Similar Blocks

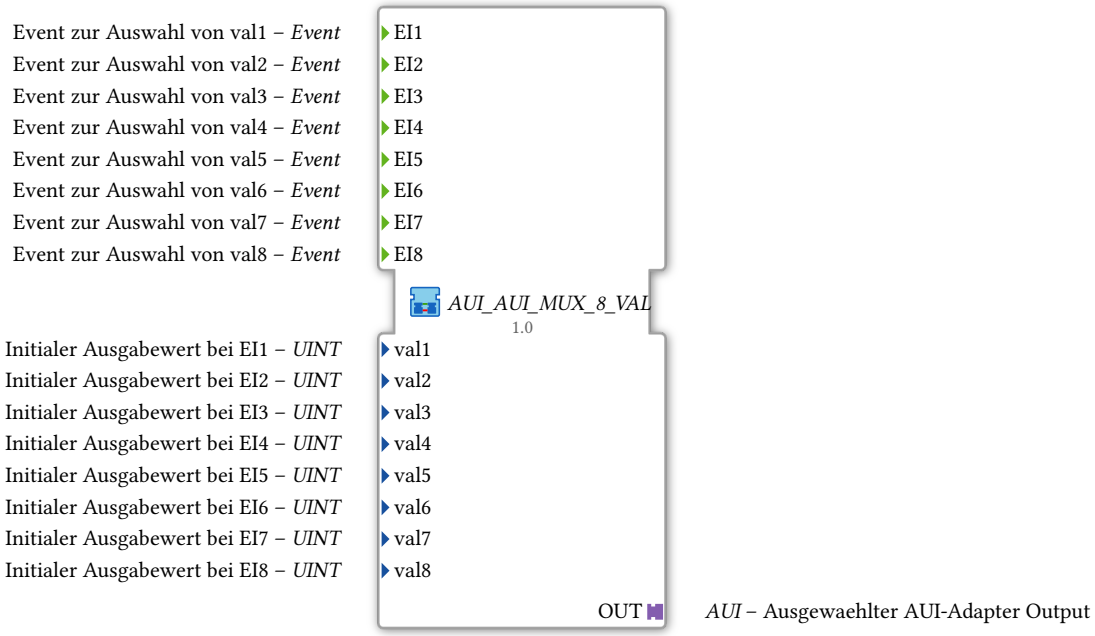
- **vs. Standard MUX blocks**: Typical multiplexer FBs in IEC 61499 often use a data input as the selector (e.g., a K input) and do not provide per-event selection. This subapplication uses individual event inputs, which is more natural for event-driven control flows.
- **vs. AUI-only multiplexers**: Some multiplexers operate entirely within the AUI domain, but they may not offer the UINT-to-AUI conversion. Here, the conversion is intrinsic, simplifying the user interface.
- **vs. Single-value selectors**: For fewer channels, a simpler selector might suffice, but the 7-channel configuration provides a scalable solution without needing multiple instances.

Conclusion

The **AUI_AUI_MUX_7_VAL** subapplication is a well-designed, reusable component for selecting one of seven UINT values based on event triggers and exposing the result as a unidirectional AUI adapter. Its event-driven nature, combined with built-in UINT-to-AUI conversion, makes it a practical building block for IEC 61499 applications needing flexible and reliable data multiplexing. With its clear interface and absence of internal state, it is easy to integrate, test, and maintain.

AUI_AUI_MUX_8_VAL





• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_AUI_MUX_8_VAL** is a composite subapplication that provides an 8-way multiplexer for **UINT** values, which are presented on an **AUI** adapter interface. It selects one of eight input values based on an incoming event and forwards the selected value as an AUI output. This block is especially useful when a control system needs to switch between different parameter sets or configuration values in a structured, event-driven manner.

Interface Structure

Event Inputs

Name	Type	Comment
EI1	Event	Event to select val1
EI2	Event	Event to select val2
EI3	Event	Event to select val3
EI4	Event	Event to select val4
EI5	Event	Event to select val5
EI6	Event	Event to select val6
EI7	Event	Event to select val7
EI8	Event	Event to select val8

Event Outputs

None.

Data Inputs

Name	Type	Comment
val1	UINT	Initial output value for EI1
val2	UINT	Initial output value for EI2

val3	UINT	Initial output value for EI3
val4	UINT	Initial output value for EI4
val5	UINT	Initial output value for EI5
val6	UINT	Initial output value for EI6
val7	UINT	Initial output value for EI7
val8	UINT	Initial output value for EI8

Data Outputs

None.

Adapters

Name	Type	Comment
OUT	adapter::types::unidirectional::AUI	Selected AUI adapter output

Functionality

The subapplication acts as a demultiplexer/selector for AUI-encapsulated values. Internally, it combines three functional elements:

1. **AUI_MUX_8** – an event-driven multiplexer block that receives the eight event inputs and converts them into a single selection signal (available on its `K` adapter output).
2. **initval_AUI_1 ... initval_AUI_8** – eight instances of an AUI initialisation block. Each block accepts a UINT value (`INIT_VAL`) and provides it as an AUI output.
3. **AUI_AUI_MUX_8** – a selection block that receives the selection signal `K` from the first block and the eight AUI inputs from the initval blocks. It forwards exactly one of these inputs to the subapplication's `OUT` adapter, depending on the active event.

When an event is received on `EIx` (x = 1 ... 8), the corresponding input value `valx` is propagated to the output adapter. The internal architecture ensures that only the selected value is presented on `OUT`, while all other values remain isolated.

Technical Features

- **8-channel selection** – up to eight independent UINT input values can be managed.
- **Event-driven switching** – selection occurs only upon an event, providing deterministic behaviour.
- **AUI adapter output** – the selected value is provided as an AUI type, allowing seamless integration with other adapter-based components in the 4diac environment.
- **Internal initialisation** – each input value is converted to an AUI representation using dedicated initval blocks, simplifying the connection to the selection logic.
- **Modular design** – built from reusable standard blocks, making the subapplication easy to maintain and extend.

State Overview

The subapplication does not maintain a persistent state. It is purely combinational with respect to the event inputs: after the reception of an event on `EIx`, the output reflects the value of `valx`. No internal state variables are used, and the block behaves identically for every execution cycle.

Application Scenarios

Typical use cases for the `AUI_AUI_MUX_8_VAL` include:

- **Parameter switching** – selecting between different configuration sets (e.g., different motor profiles, sensor calibration values) based on external control events.
- **Mode selection** – choosing between operating modes in a machine or process, where each mode has its own set of parameters.
- **Test and simulation** – dynamically feeding different test values into a component without reconfiguring the network.

- **Redundancy management** – switching between primary and backup values in case of failure detection.

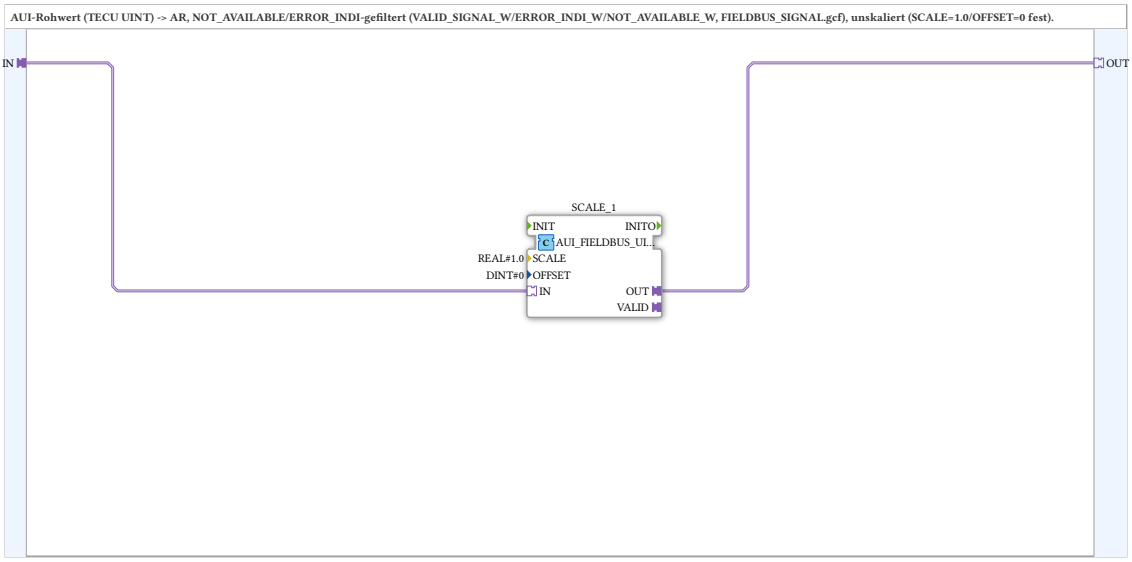
Comparison with Similar Blocks

Compared to simpler multiplexers that operate directly on UINT data, `AUI_AUI_MUX_8_VAL` provides an **AUI-typed output**, which is essential when the downstream components expect an adapter interface. Unlike a conventional event-driven multiplexer that may require additional conversion steps, this subapplication integrates the conversion and selection in one block. It also supports eight channels, while many standard multiplexers are limited to fewer inputs. However, it does not offer an explicit "no selection" or invalid event state – only the eight defined event inputs are accepted.

Conclusion

The `AUI_AUI_MUX_8_VAL` is a compact and reusable subapplication for event-based selection among eight UINT values, presented as an AUI output. Its internal structure leverages proven building blocks, resulting in a clean and deterministic behaviour. It is suitable for a wide range of industrial automation and control applications where multiple parameter sets or modes need to be switched dynamically.

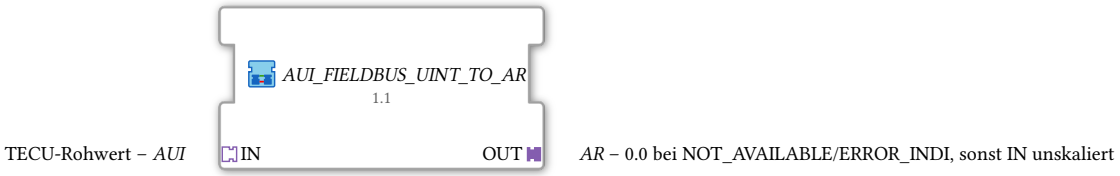
AUI_FIELDBUS_UINT_TO_AR



• – ○ • – ○ • – ○ •

Introduction

`AUI_FIELDBUS_UINT_TO_AR` converts an `AUI` raw value (TECU, `UINT`) into an `AR` value and filters out the fieldbus special values `NOT_AVAILABLE` and `ERROR_INDI` on the way. The conversion is unscaled (`SCALE = 1.0`, `OFFSET = 0` fixed). The block was extracted because the same parameterisation was previously set individually in `TECU_RPTO`, `TECU_FPTO`, `TECU_COGSOG`, `TECU_GBSD`, `TECU_WBSD`, `TECU_VDS`, `TECU_FHS` and `TECU_RHS`. It lives in `MyLib_AX::sys` because it is generic and not TECU-specific.



Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

- **IN** (socket, `adapter::types::unidirectional::AUI`): TECU raw value
- **OUT** (plug, `adapter::types::unidirectional::AR`): `0.0` on `NOT_AVAILABLE/ERROR_INDI`, otherwise `IN` unscaled

Function Blocks (FBs) Used

- **`SCALE_1`** (`logiBUS::signalprocessing::fieldbus::AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED`): parameters `SCALE = 1.0`, `OFFSET = 0`

Functionality

`IN` ⟶ `SCALE_1.IN` and `SCALE_1.OUT` ⟶ `OUT`. The filtering of the fieldbus special values (`VALID_SIGNAL_W`/`ERROR_INDI_W`/`NOT_AVAILABLE_W`, defined in `FIELDBUS_SIGNAL.gcf`) is done by the internal block; the SubApp only fixes scale and offset.

Technical Details

- No event or data ports of its own, adapters only
- Fixed scale factor 1.0 and offset 0 (generic, unscaled variant)

Application Scenarios

- Taking TECU raw values (e.g. speeds) from the fieldbus as an `AR` signal without scaling

Summary

`AUI_FIELDBUS_UINT_TO_AR` wraps `AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED` with `SCALE = 1.0`/`OFFSET = 0` as a reusable SubApp.

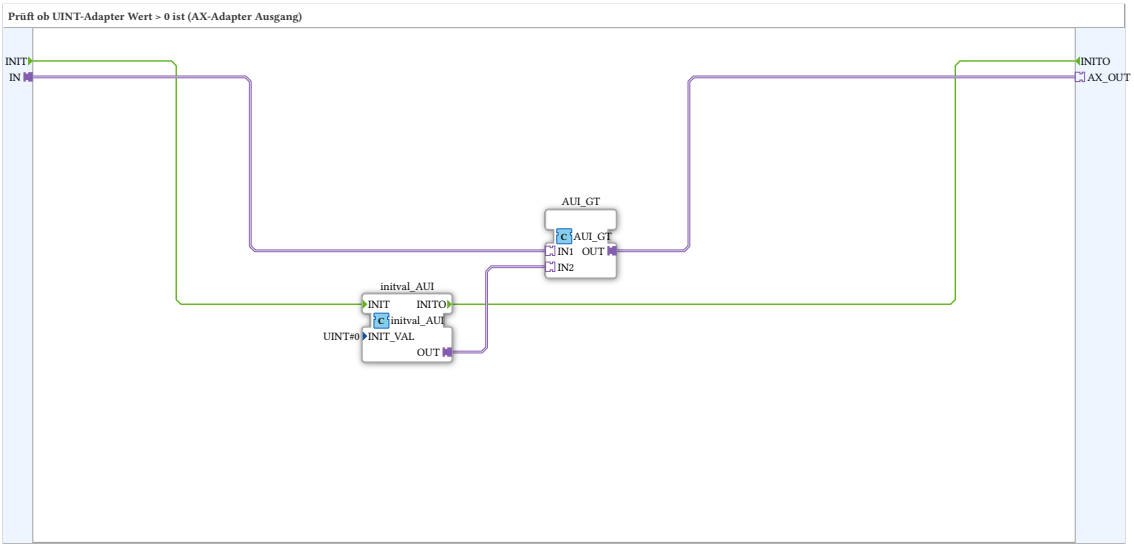
🛠️ Related Exercises

- Uebung_078_AUI

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

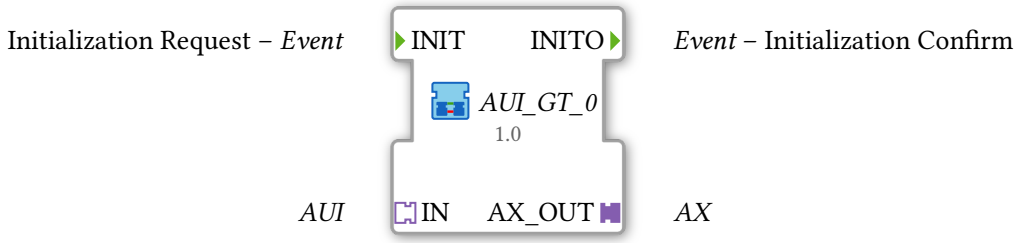
AUI_GT_0



• – ○ • – ○ • – ○ •

Introduction

`AUI_GT_0` checks whether a UINT adapter value is greater than 0 and outputs the result as a boolean AX adapter signal. A typical input is an object-ID or counter value (e.g. "how many channels are active"). The block is built identically for every numeric type (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Interface

- **Event input**: `INIT` (initialization)
- **Event output**: `INITO` (initialization finished, comes from the internal `initval_AUI`)
- **Socket**: `IN` (`AUI`) — the value compared against 0
- **Plug**: `AX_OUT` (`AX`) — TRUE when `IN > 0`

Function Blocks (FBs) Used

Sub-blocks: AUI_GT_0

- **Type**: SubAppType
- **Internal FBs used**:
  - **initval_AUI**: `adapter::types::unidirectional::AUI::initval::initval_AUI` — supplies the fixed comparison constant `UINT#0` as an adapter value (`INIT_VAL=UINT#0`). An adapter socket cannot take the constant as a parameter, so it is fed in through this block.
  - **AUI_GT**: `adapter::iec61131::comparison::AUI_GT` — adapter-native comparison `IN1 > IN2`.
- **Functionality**: `IN` ⟶ `AUI_GT.IN1`; constant 0 ⟶ `AUI_GT.IN2`; result ⟶ `AX_OUT`.

Program Flow and Connections

1. `INIT` ⟶ `initval_AUI.INIT`; `initval_AUI.INITO` ⟶ `INITO`.
2. `IN` (adapter) ⟶ `AUI_GT.IN1`.
3. `initval_AUI.OUT` (constant `UINT#0`) ⟶ `AUI_GT.IN2`.
4. `AUI_GT.OUT` ⟶ `AX_OUT` (adapter).

Application Scenarios

- Converting an object-ID or counter value (e.g. "how many channels are active") into a simple boolean adapter signal for VT status displays or enable logic.

Summary

`AUI_GT_0` is a compact wrapper around the adapter-native comparison `AUI_GT` — converts "UINT > 0" directly into an AX adapter signal, so the caller does not have to create and wire the zero constant itself.

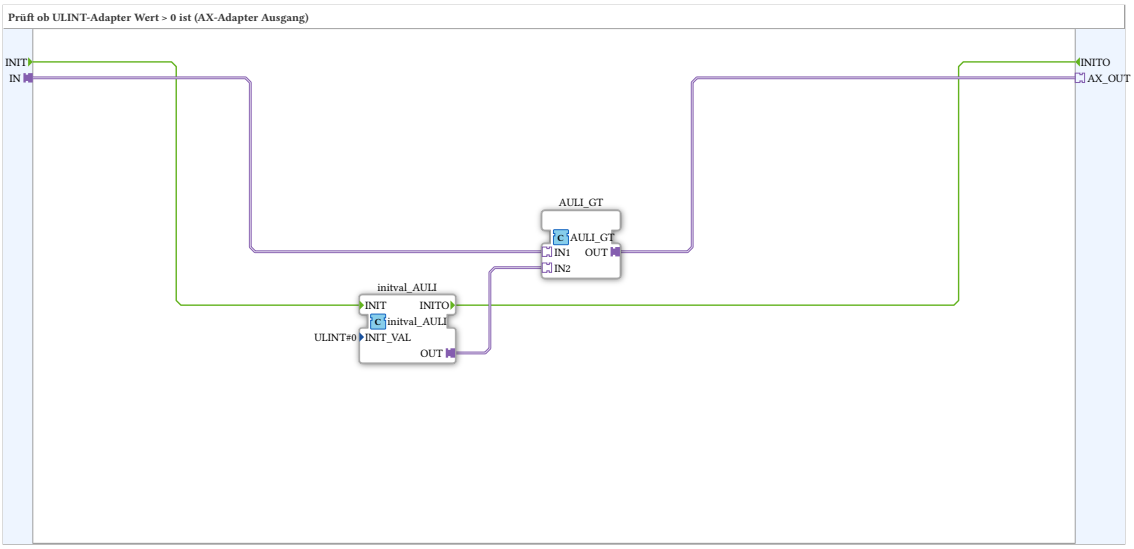
🛠️ Related Exercises

- Uebung_071_AUI
- Uebung_071a_AUI

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

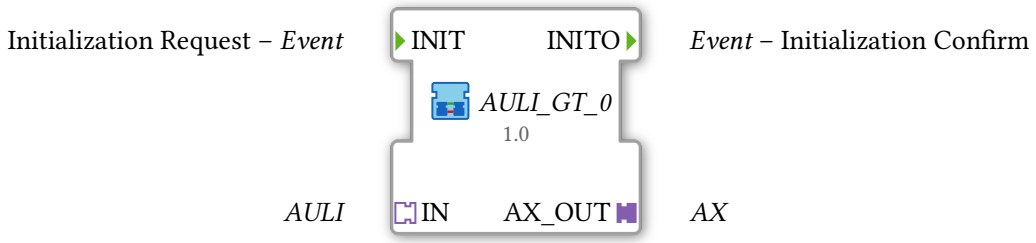
AULI_GT_0



• – ◦ • – ◦ • – ◦ •

Introduction

`AULI_GT_0` checks whether a ULINT adapter value is greater than 0 and outputs the result as a boolean AX adapter signal. A typical input is an unsigned 64-bit counter value. The block is built identically for every numeric type (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Interface

- **Event input**: `INIT` (initialization)
- **Event output**: `INITO` (initialization finished, comes from the internal `initval_AULI`)
- **Socket**: `IN` (`AULI`) — the value compared against 0
- **Plug**: `AX_OUT` (`AX`) — TRUE when `IN > 0`

Function Blocks (FBs) Used

Sub-blocks: AULI_GT_0

- **Type**: SubAppType
- **Internal FBs used**:
  - **initval_AULI**: `adapter::types::unidirectional::AULI::initval::initval_AULI` — supplies the fixed comparison constant `ULINT#0` as an adapter value (`INIT_VAL=ULINT#0`). An adapter socket cannot take the constant as a parameter, so it is fed in through this block.
  - **AULI_GT**: `adapter::iec61131::comparison::AULI_GT` — adapter-native comparison `IN1 > IN2`.
- **Functionality**: `IN` ⟶ `AULI_GT.IN1`; constant 0 ⟶ `AULI_GT.IN2`; result ⟶ `AX_OUT`.

Program Flow and Connections

1. `INIT` ⟶ `initval_AULI.INIT`; `initval_AULI.INITO` ⟶ `INITO`.
2. `IN` (adapter) ⟶ `AULI_GT.IN1`.
3. `initval_AULI.OUT` (constant `ULINT#0`) ⟶ `AULI_GT.IN2`.
4. `AULI_GT.OUT` ⟶ `AX_OUT` (adapter).

Application Scenarios

- Converting an unsigned 64-bit counter value into a simple boolean adapter signal for VT status displays or enable logic.

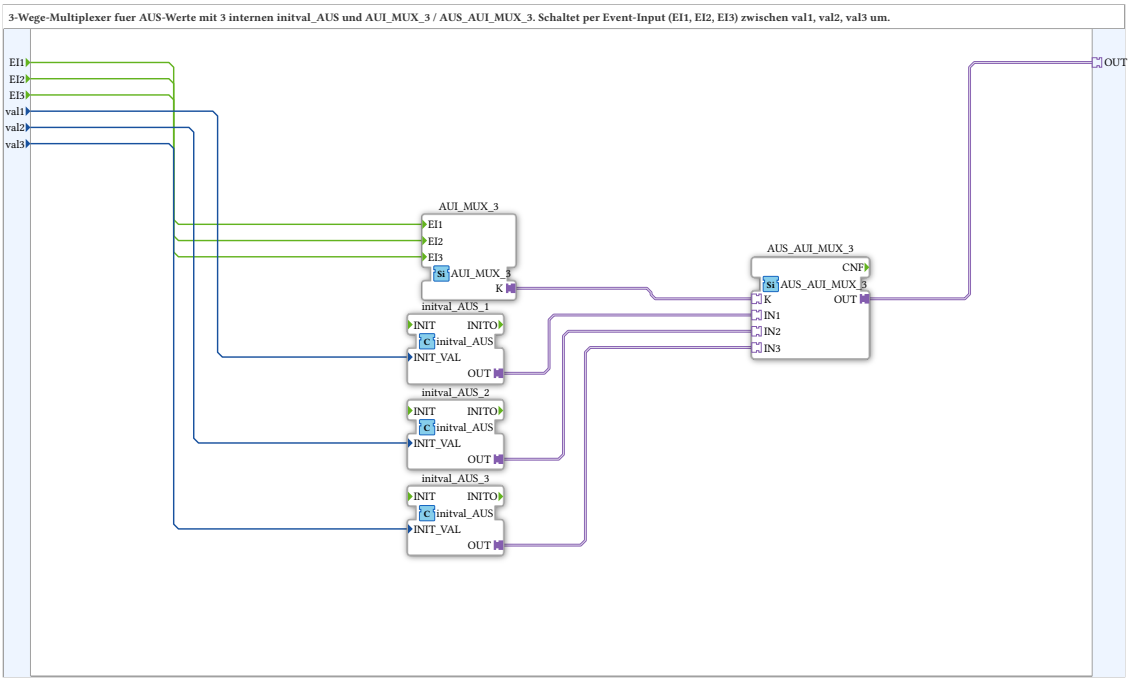
Summary

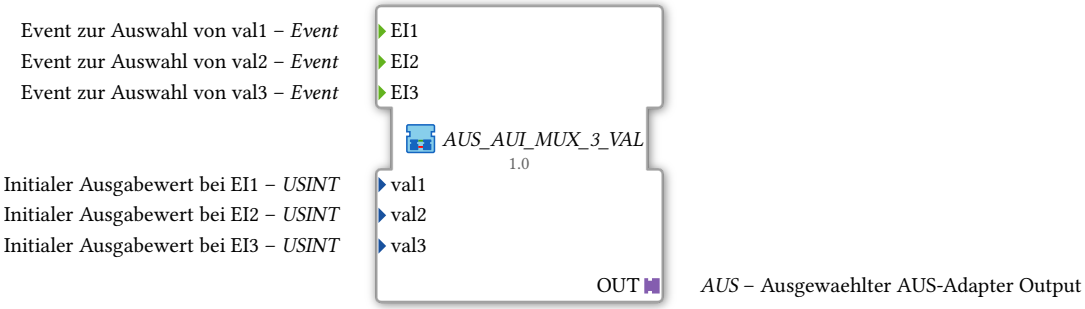
`AULI_GT_0` is a compact wrapper around the adapter-native comparison `AULI_GT` — converts "ULINT > 0" directly into an AX adapter signal, so the caller does not have to create and wire the zero constant itself.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AUS_AUI_MUX_3_VAL





• – ○ • – ○ • – ○ •

Introduction

`AUS_AUI_MUX_3_VAL` is a composite subapplication that implements a three-way multiplexer for AUS (Automation Unit Service) values. It provides three independent USINT input values (`val1`, `val2`, `val3`) and, based on which of the three event inputs (`EI1`, `EI2`, `EI3`) is triggered, routes the corresponding value to a unidirectional AUS adapter output (`OUT`).

Internally, the subapplication combines three initial-value adapters (`initval_AUS`) with an event selection block (`AUI_MUX_3`) and an adapter multiplexer (`AUS_AUI_MUX_3`). This allows the user to supply plain numeric values on the subapplication interface while internally converting them to the required AUS adapter representation.

Interface Structure

Event Inputs

Name	Type	Comment
EI1	Event	Event to select val1 as active output
EI2	Event	Event to select val2 as active output
EI3	Event	Event to select val3 as active output

Event Outputs

None. The subapplication does not provide any event outputs.

Data Inputs

Name	Type	Comment
val1	USINT	Initial output value used when EI1 is triggered
val2	USINT	Initial output value used when EI2 is triggered
val3	USINT	Initial output value used when EI3 is triggered

Data Outputs

None. The result of the multiplexing is delivered via the adapter plug OUT.

Adapters

Name	Direction	Type	Comment
OUT	Plug	adapter::types::unidirectional::AUS	Carries the currently selected AUS value

Functionality

The subapplication is composed of five internal function blocks that work together to perform the multiplexing task:

1. **initval_AUS_1, initval_AUS_2, initval_AUS_3** – Each of these blocks (type adapter::types::unidirectional::AUS::initval::initval_AUS) receives one of the USINT data inputs (val1, val2, val3) on its INIT_VAL input and exposes it as a unidirectional AUS adapter on its OUT output. This converts the plain numeric value into the AUS adapter form required by the downstream selection block.
2. **AUI_MUX_3** – An event multiplexer (type adapter::events::unidirectional::AUI_MUX_3) that receives the three event inputs. When an event is received, it sets the selection signal on its adapter output K to indicate which channel has been requested.
3. **AUS_AUI_MUX_3** – An adapter selection block (type adapter::selection::unidirectional::AUS_AUI_MUX_3) that takes the three prepared AUS adapter inputs (IN1, IN2, IN3) plus the selection signal K. Based on the value of K, it forwards the corresponding input to its OUT adapter.

Signal flow summary:

- EI1/EI2/EI3 ⟶ AUI_MUX_3.EI1/2/3 ⟶ AUI_MUX_3.K (selection)
- val1/val2/val3 ⟶ initval_AUS_1/2/3.INIT_VAL ⟶ initval_AUS_1/2/3.OUT ⟶ AUS_AUI_MUX_3.IN1/2/3
- AUS_AUI_MUX_3.OUT ⟶ subapplication plug OUT

When, for example, EI1 is triggered, AUI_MUX_3 encodes this on K, which instructs AUS_AUI_MUX_3 to pass the AUS value generated from val1 to the OUT adapter.

Technical Features

- **Three independent event-triggered channels** – Each input value is selected exclusively by its corresponding event.
- **USINT data type** – All three input values are unsigned short integers (0–255), suitable for small control values, addresses, or counters.
- **Adapter-based output** – The result is provided through a unidirectional AUS adapter, enabling direct connection to other AUS-compatible blocks in the application.
- **Internal value conversion** – The `initval_AUS` blocks transparently convert raw USINT values into the AUS adapter format, hiding internal representation details from the user.
- **Modular composition** – The subapplication reuses standard blocks (`AUI_MUX_3`, `AUS_AUI_MUX_3`, `initval_AUS`) and can be easily adapted or extended.
- **No data or event outputs** – Simplicity; the only external result is the selected AUS adapter value.
- **Deterministic selection** – The selection logic is purely event-driven; the most recently triggered event determines the active output.

State Overview

The subapplication does not contain an explicit state machine with multiple persistent states. Instead, it behaves as a **one-hot selection logic**:

- After receipt of `EI1`, the output presents the AUS value derived from `val1`.
- After receipt of `EI2`, the output presents the AUS value derived from `val2`.
- After receipt of `EI3`, the output presents the AUS value derived from `val3`.

The "current state" can be described as the identity of the last received event (or the corresponding selected channel). The output is updated immediately upon the event, without additional data processing steps. Because the selection is event-triggered, no polling or continuous evaluation of the data inputs is required.

Application Scenarios

- **Mode / profile selection** – Selecting among three pre-defined parameter sets (e.g., operating modes, speed profiles) via discrete event signals.
- **Input routing in process control** – Redirecting a control value from one of three sources to a common output depending on external triggers.
- **Value presets for automation units** – Supplying a fixed value to an AUS-consuming block (e.g., a setpoint or limit) and switching between presets on demand.
- **Testing and simulation** – Switching between different simulation values in a test environment without reconfiguration.
- **HMI or master control integration** – Where operator actions (buttons, commands) generate events that select the corresponding operation value.

Comparison with Similar Blocks

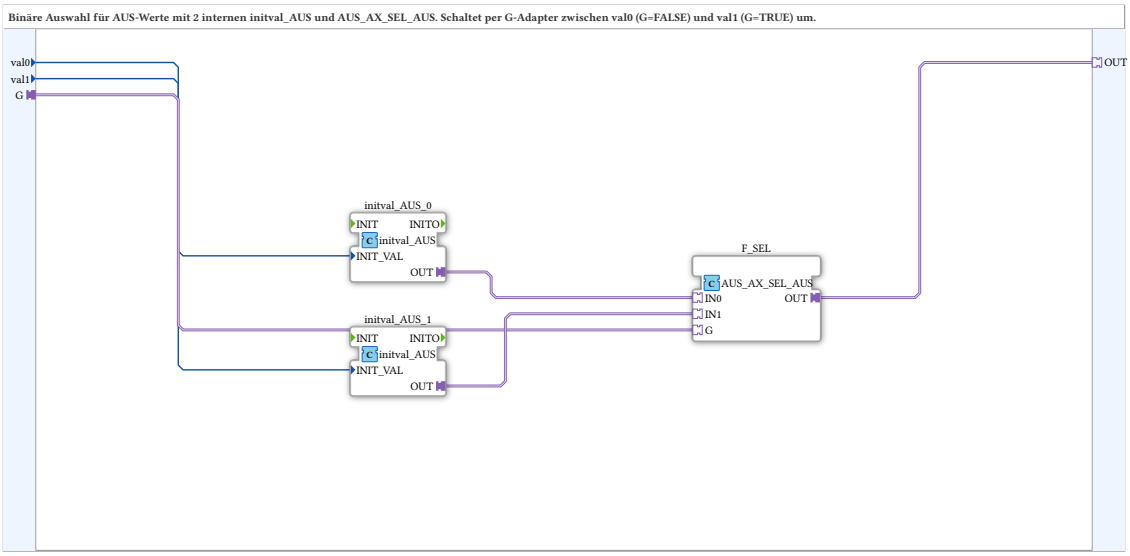
Feature	`AUS_AUI_MUX_3` (standalone)	`AUS_AUI_MUX_3_VAL` (this subapplication)
Input handling	Requires already-prepared AUS adapters on `IN1`, `IN2`, `IN3`	Accepts plain USINT values and converts them internally
Value provisioning	External blocks must supply AUS values	Self-contained; values are provided directly on the subapp interface
Number of external connections	More complex wiring (multiple AUS plugs)	Simplified – only three numeric inputs and one event per channel
Use of `initval_AUS`	Not included	Integrated internally
Ease of use	Lower – requires in-depth adapter knowledge	Higher – user only provides simple values and events

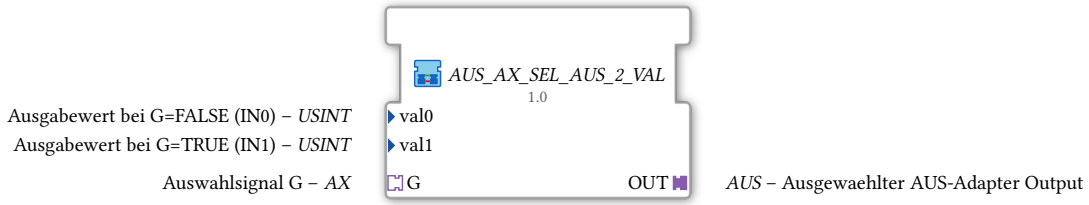
Compared to a generic IEC 61499 MUX function block, this subapplication adds the AUS **adapter conversion layer**, making it directly interoperable with AUS-based components while keeping the interface numeric. Unlike a binary-coded multiplexer, selection here is **event-driven** rather than address-driven.

Conclusion

`AUS_AUI_MUX_3_VAL` provides a clean, self-contained solution for selecting one of three AUS values via discrete events. By wrapping the `initval_AUS`, `AUI_MUX_3`, and `AUS_AUI_MUX_3` blocks into a single composite, it hides the internal adapter transformations and offers a simple numeric interface. This makes it an ideal building block for applications that need to switch between preset values in an event-oriented manner, while remaining compatible with the 4diac adapter ecosystem.

AUS_AX_SEL_AUS_2_VAL





• – ○ • – ○ • – ○ •

Introduction

The **AUS_AX_SEL_AUS_2_VAL** is a subapplication that implements a binary selection of two `AUS` (output) adapter values based on a select signal carried by an `AX` adapter. It encapsulates two internal `initval_AUS` blocks that convert raw `USINT` values into valid `AUS` adapter data, and a central `AUS_AX_SEL_AUS` selection block. Depending on the state of the select input `G`, the subapp forwards one of the two configured values (`val0` or `val1`) to its output adapter `OUT`.

Interface Structure

The subapplication provides no direct event inputs or outputs; all interaction takes place through data inputs and adapter connections.

Event Inputs

None.

Event Outputs

None.

Data Inputs

Name	Type	Comment
`val0`	`USINT`	Output value when `G = FALSE` (IN0)

val1	USINT	Output value when G = TRUE (IN1)

Data Outputs

None.

Adapters

Direction	Name	Type	Comment
Socket (Input)	G	adapter::types::unidirectional::AX	Selection signal G
Plug (Output)	OUT	adapter::types::unidirectional::AUS	Selected AUS adapter output

Functionality

The internal network consists of three function blocks and the associated connections:

1. **initval_AUS_0** – Takes the val0 data input and generates an AUS adapter value as its output.
2. **initval_AUS_1** – Takes the val1 data input and generates an AUS adapter value as its output.
3. **F_SEL** (type AUS_AX_SEL_AUS) – Receives the two AUS values on its IN0 and IN1 inputs, and the selection signal G (type AX) on its G input. It forwards the selected AUS value to its OUT output, which is connected to the subapplication's OUT plug.

The selection logic is straightforward: when G is FALSE, the value from val0 (via initval_AUS_0) is forwarded; when G is TRUE, the value from val1 (via initval_AUS_1) is forwarded.

Technical Features

- **Encapsulated initialization** – The raw USINT values are wrapped into AUS adapter objects by the two internal initval_AUS blocks, keeping the subapp interface clean and adapter-based.
- **Adapters only interface** – External communication is performed exclusively through unidirectional adapters (AX for selection, AUS for the output), making the block compatible with adapter-based system designs.
- **Combinational behavior** – The output follows the selection input without internal state memory; changes on val0, val1, or G are directly reflected.
- **No event dependencies** – The subapp works without event triggers, simplifying its integration in cyclic or continuous processing environments.

State Overview

This subapplication is purely combinational and does not maintain an internal state. The output is determined solely by the current values of the data inputs val0, val1, and the adapter input G. No initialization sequence, reset handling, or state transition logic is required.

Application Scenarios

- **Preset value switching** – Selecting between two predefined AUS output values (e.g., two different speed commands or configuration codes) based on a binary control signal.
- **Redundant parameter selection** – Switching between a primary and a fallback value in automation systems where an AUS adapter delivers the active parameter.
- **Adapter-based I/O routing** – In systems that standardize on unidirectional AUS/AX adapters, this subapp provides a reusable, interface-compatible selection element.
- **Configuration time setting** – Used to map static configuration values into adapter-formatted outputs, as the initval blocks convert constants or externally provided values.

Comparison with Similar Blocks

Feature	AUS_AX_SEL_AUS_2_VAL	Standard SEL / MUX (IEC 61131)
Interface type	Adapter-based (AX / AUS)	Direct data pins (e.g., USINT)
Value initialization	Implicit via internal initval_AUS blocks	Requires external value preparation
Output type	Adapter (AUS)	Raw data type

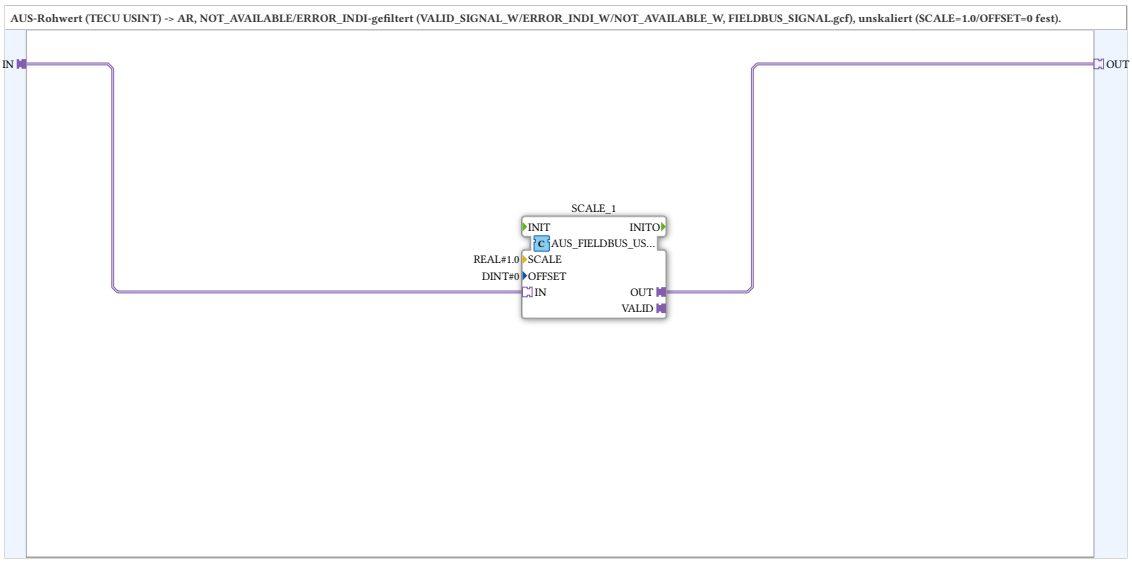
Use in adapter-heavy designs	Fully compatible	Requires additional conversion blocks

Unlike a plain SEL function block, this subapp encapsulates both the value initialization and the adapter connection logic, reducing the wiring effort and increasing reusability in adapter-oriented architectures.

Conclusion

AUS_AX_SEL_AUS_2_VAL provides a clean, adapter-based solution for selecting between two AUS values under the control of an AX selection signal. Its internal use of initval_AUS blocks hides the conversion complexity from the application programmer and ensures that the subapp can be dropped directly into 4diac projects relying on unidirectional adapters. With its combinational nature and simple interface, it is a practical building block for binary configuration and routing in IEC 61499-based automation systems.

AUS_FIELDBUS_USINT_TO_AR

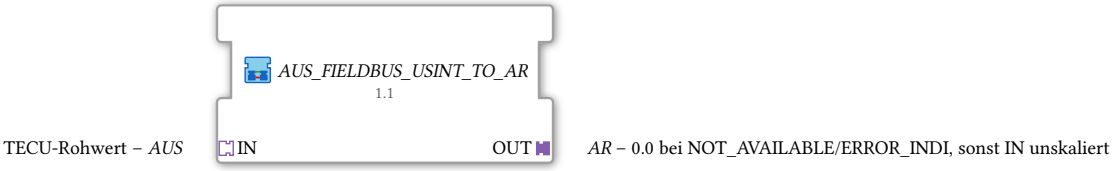


• – ○ • – ○ • – ○ •

Introduction

AUS_FIELDBUS_USINT_TO_AR converts an **AUS raw value** (TECU USINT) into an AR value. The values NOT_AVAILABLE/ERROR_INDI are filtered (VALID_SIGNAL_W/ERROR_INDI_W/NOT_AVAILABLE_W, FIELDBUS_SIGNAL.gcf); the scaling is fixed to SCALE=1.0 and OFFSET=0, so the value remains unscaled.

The SubApp was extracted from AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED because SCALE=1.0/OFFSET=0 had to be parameterized individually every time in TECU_FHS/TECU_RHS; it lives in MyLib_AX::sys because it is generic and not TECU-specific.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **IN** (socket): Unidirectional adapter of type `adapter::types::unidirectional::AUS` - TECU raw value
- **OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AR` - `0.0` on `NOT_AVAILABLE/ERROR_INDI`, otherwise `IN` unscaled

Function Blocks (FBs) Used

- **SCALE_1**: `logiBUS::signalprocessing::fieldbus::AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED`
  - Parameters: `SCALE=1.0`, `OFFSET=0`

Program Flow and Connections

- `IN` ⟶ `SCALE_1.IN`
- `SCALE_1.OUT` ⟶ `OUT`

The `VALID` plug of the inner block is not exposed.

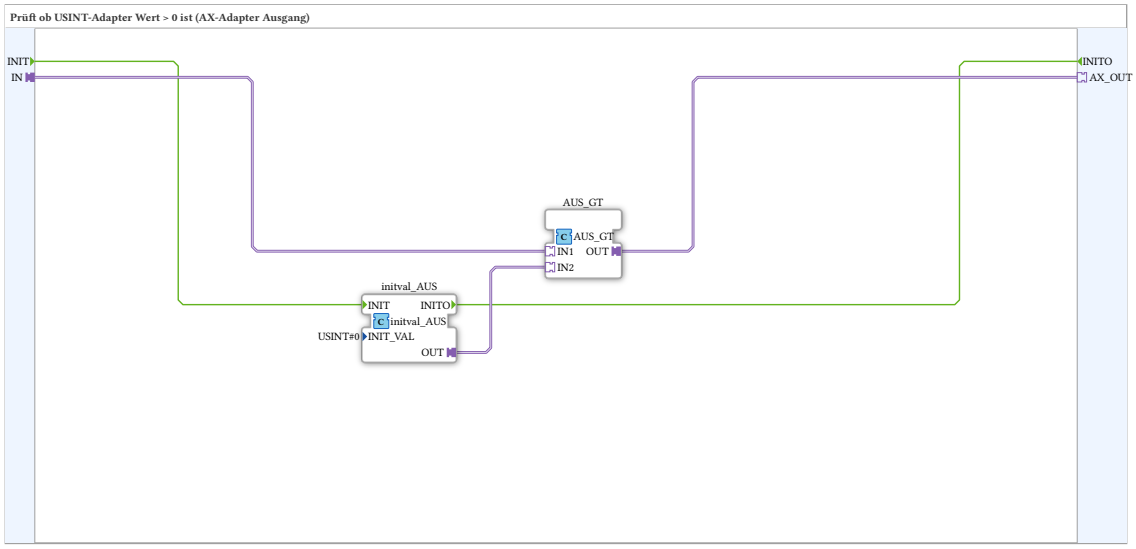
Application Scenarios

- Taking over TECU values (`USINT`) as a filtered, unscaled `AR` value

Summary

`AUS_FIELDBUS_USINT_TO_AR` is a small wrapper SubApp that provides `AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED` with the fixed parameters `SCALE=1.0`/`OFFSET=0`.

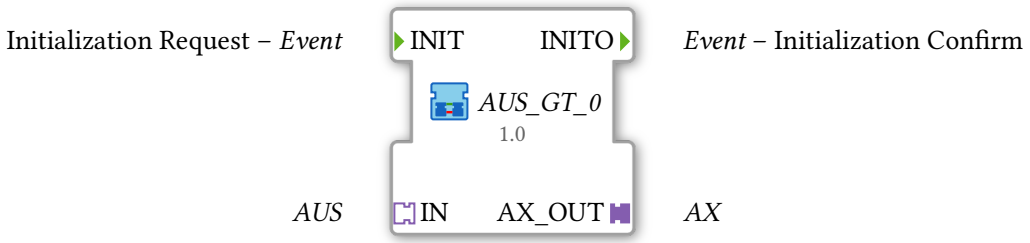
AUS_GT_0



• – ○ • – ○ • – ○ •

Introduction

`AUS_GT_0` checks whether a USINT adapter value is greater than 0 and outputs the result as a boolean AX adapter signal. A typical input is an unsigned 8-bit value. The block is built identically for every numeric type (`AS`, `AI`, `ADI`, `ALI`, `AUS`, `AUI`, `AUDI`, `AULI`, `AR`, `ALR`).



Interface

- **Event input**: `INIT` (initialization)
- **Event output**: `INITO` (initialization finished, comes from the internal `initval_AUS`)
- **Socket**: `IN` (`AUS`) — the value compared against 0
- **Plug**: `AX_OUT` (`AX`) — TRUE when `IN > 0`

Function Blocks (FBs) Used

Sub-blocks: AUS_GT_0

- **Type**: SubAppType
- **Internal FBs used**:
  - **initval_AUS**: `adapter::types::unidirectional::AUS::initval::initval_AUS` — supplies the fixed comparison constant `USINT#0` as an adapter value (`INIT_VAL=USINT#0`). An adapter socket cannot take the constant as a parameter, so it is fed in through this block.
  - **AUS_GT**: `adapter::iec61131::comparison::AUS_GT` — adapter-native comparison `IN1 > IN2`.
- **Functionality**: `IN` ⟶ `AUS_GT.IN1`; constant 0 ⟶ `AUS_GT.IN2`; result ⟶ `AX_OUT`.

Program Flow and Connections

1. `INIT` ⟶ `initval_AUS.INIT`; `initval_AUS.INITO` ⟶ `INITO`.
2. `IN` (adapter) ⟶ `AUS_GT.IN1`.
3. `initval_AUS.OUT` (constant `USINT#0`) ⟶ `AUS_GT.IN2`.
4. `AUS_GT.OUT` ⟶ `AX_OUT` (adapter).

Application Scenarios

- Converting an unsigned 8-bit value into a simple boolean adapter signal for VT status displays or enable logic.

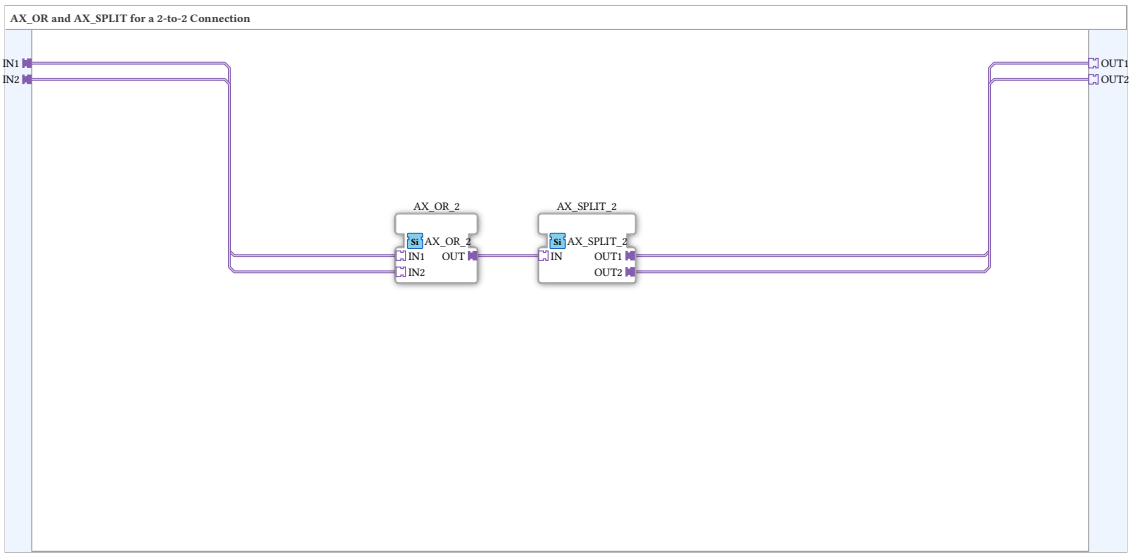
Summary

`AUS_GT_0` is a compact wrapper around the adapter-native comparison `AUS_GT` — converts “USINT > 0” directly into an AX adapter signal, so the caller does not have to create and wire the zero constant itself.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

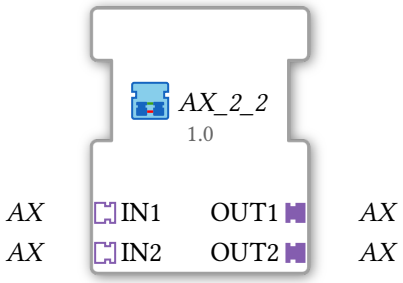
AX_2_2



• – ○ • – ○ • – ○ •

Introduction

`AX_2_2` ORs two AX adapter inputs (`IN1`, `IN2`) and distributes the result to two AX adapter outputs (`OUT1`, `OUT2`) — a "2-to-2" wiring block for cases where two independent boolean sources must be combined and the result forwarded to two independent destinations.



Function Blocks (FBs) Used

Sub-blocks: AX_2_2

- **Type**: SubAppType
- **Internal FBs used**:
  - **AX_OR_2**: `adapter::booleanOperators::AX_OR_2` — ORs two adapter inputs.
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — splits the OR result into two outputs.
- **Functionality**: `IN1 OR IN2` ⟶ `OUT1` and `OUT2` simultaneously.

Program Flow and Connections

1. `IN1` ⟶ `AX_OR_2.IN1`, `IN2` ⟶ `AX_OR_2.IN2`.
2. `AX_OR_2.OUT` ⟶ `AX_SPLIT_2.IN`.
3. `AX_SPLIT_2.OUT1` ⟶ `OUT1`, `AX_SPLIT_2.OUT2` ⟶ `OUT2`.

Application Scenarios

- Anywhere two independent boolean adapter sources need to be ORed and the combined result forwarded to two independent consumers (e.g. two VT status displays) without wiring a duplicate adapter connection.

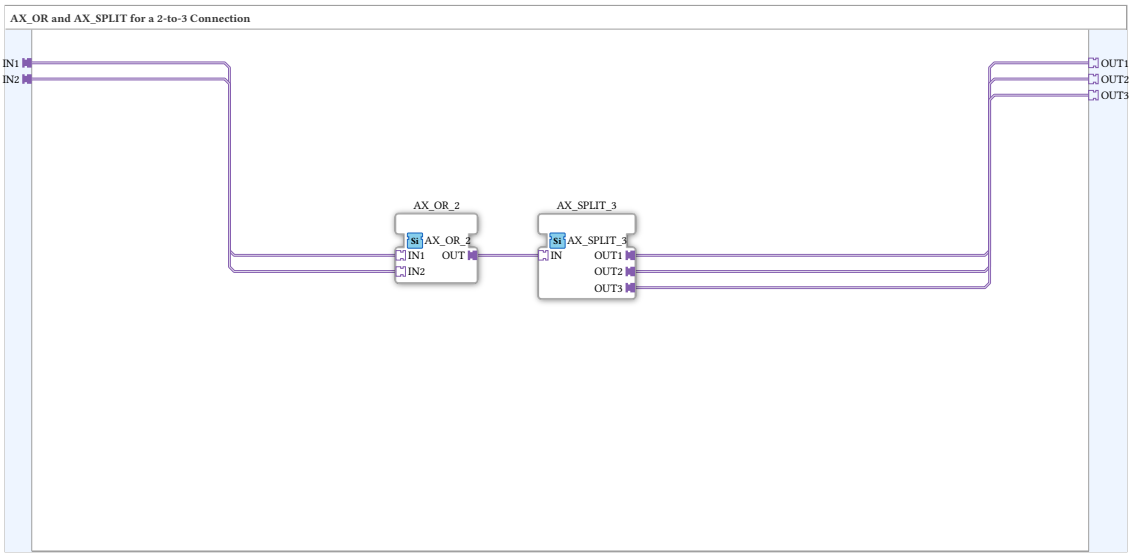
Summary

`AX_2_2` is a minimal block for "OR two sources, distribute to two destinations" — pure adapter wiring with no state logic of its own.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

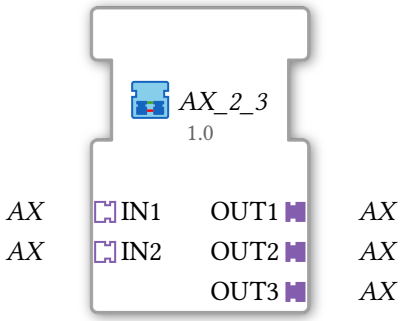
AX_2_3



• – ◦ • – ◦ • – ◦ •

Introduction

AX_2_3 combines AX_OR_2 and AX_SPLIT_3 into a single, complete module: Two AX sources (IN1, IN2) are ORed together to form a single signal and then distributed to three destinations (OUT1, OUT2, OUT3).



Function Blocks (FBs) Used

Sub-Blocks: AX_2_3

- **Type**: SubAppType
- **Internal FBs Used**:
  - **AX_OR_2**: adapter::booleanOperators::AX_OR_2 — ORs IN1 and IN2.
  - **AX_SPLIT_3**: adapter::events::unidirectional::AX_SPLIT_3 — distributes the OR result to three outputs.
- **How it works**: IN1/IN2 ⟶ AX_OR_2.OUT ⟶ AX_SPLIT_3.IN ⟶ OUT1/OUT2/OUT3.

Program Flow and Connections

1. IN1 ⟶ AX_OR_2.IN1; IN2 ⟶ AX_OR_2.IN2.
2. AX_OR_2.OUT ⟶ AX_SPLIT_3.IN.
3. AX_SPLIT_3.OUT1 ⟶ OUT1; AX_SPLIT_3.OUT2 ⟶ OUT2; AX_SPLIT_3.OUT3 ⟶ OUT3.

Application Scenarios

- Merging two AX sources (e.g., two command paths) whose combined output must be distributed to three independent destinations (e.g., hardware, VT display, OPC UA publishing).

Comparison with Similar Components

For other source/destination counts: `AX_3_2` (3⟶2), `AX_3_3` (3⟶3). Not to be confused with `AX_2_TO_3`: this one passes UP/DOWN through individually and only provides the OR as a third additional signal, instead of generically combining OR+SPLIT.

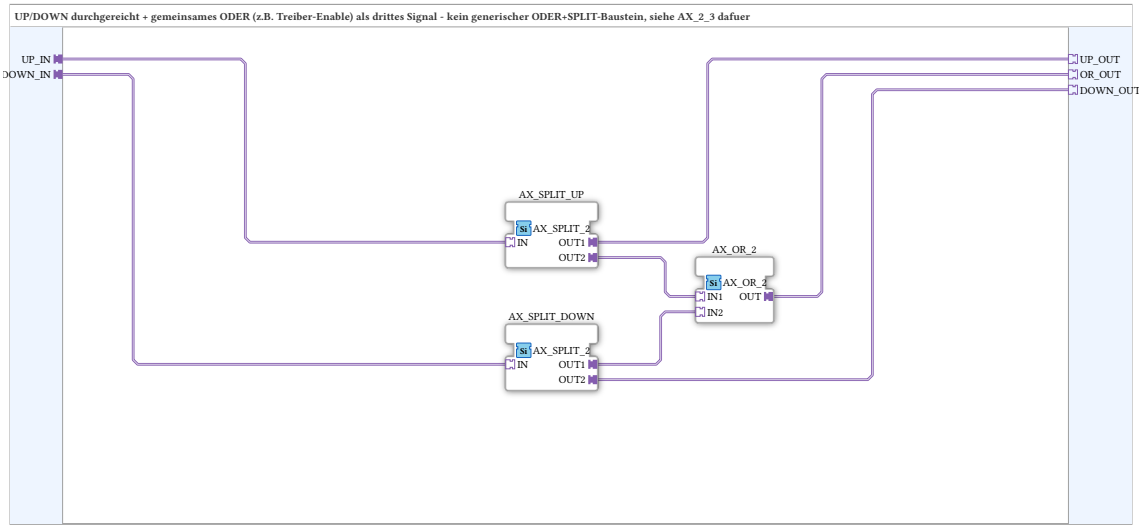
Summary

`AX_2_3` is purely a wiring aid that combines two `AX` sources via OR and distributes them to three destinations—without its own state logic.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

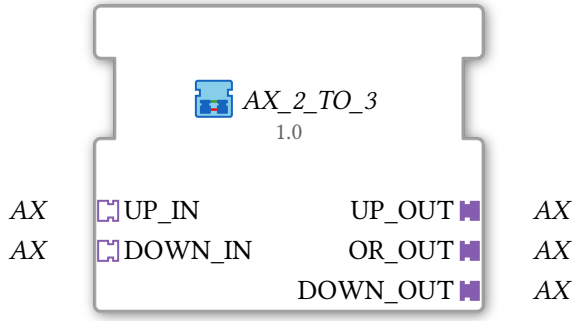
AX_2_TO_3



• – ○ • – ○ • – ○ •

Introduction

`AX_2_TO_3` takes two independent AX adapter signals (`UP_IN`, `DOWN_IN`) and provides three outputs: the two inputs passed through unchanged (`UP_OUT`, `DOWN_OUT`) plus their OR combination as a third signal (`OR_OUT`) — typically for "up"/"down" buttons where a combined "either direction active" signal is also needed.



Function Blocks (FBs) Used

Sub-blocks: AX_2_TO_3

- **Type**: SubAppType
- **Internal FBs used**:
  - **AX_SPLIT_UP** / **AX_SPLIT_DOWN**: each `adapter::events::unidirectional::AX_SPLIT_2` — split `UP_IN`/`DOWN_IN` each into a direct pass-through and a branch feeding the OR.
  - **AX_OR_2**: `adapter::booleanOperators::AX_OR_2` — combines both branches into `OR_OUT`.

- **Functionality**: `UP_IN` ⟶ `UP_OUT` (direct) and into `AX_OR_2`; `DOWN_IN` ⟶ `DOWN_OUT` (direct) and into `AX_OR_2`; `AX_OR_2.OUT` ⟶ `OR_OUT`.

Program Flow and Connections

1. `UP_IN` ⟶ `AX_SPLIT_UP.IN` ⟶ `AX_SPLIT_UP.OUT1` ⟶ `UP_OUT`, `AX_SPLIT_UP.OUT2` ⟶ `AX_OR_2.IN1`.
2. `DOWN_IN` ⟶ `AX_SPLIT_DOWN.IN` ⟶ `AX_SPLIT_DOWN.OUT2` ⟶ `DOWN_OUT`, `AX_SPLIT_DOWN.OUT1` ⟶ `AX_OR_2.IN2`.
3. `AX_OR_2.OUT` ⟶ `OR_OUT`.

Application Scenarios

- Ramp/ramp-button wiring (e.g. “up”/“down”) where, besides the individual direction signals, a combined “either button active” signal is also needed (e.g. to light up a status indicator only while actively operated).

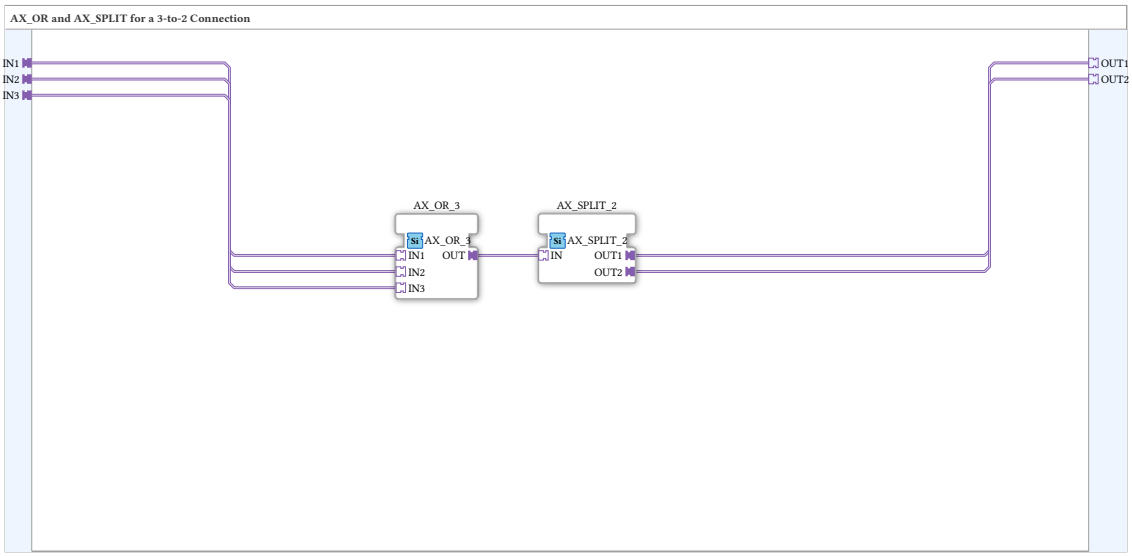
Summary

`AX_2_TO_3` passes two adapter signals through unchanged and adds their OR combination as a third signal — a pure wiring helper with no state logic of its own.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

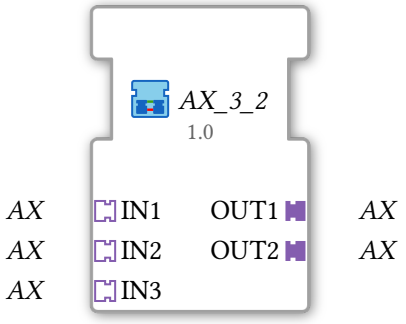
AX_3_2



• – ○ • – ○ • – ○ •

Introduction

`AX_3_2` combines `AX_OR_3` and `AX_SPLIT_2` into a single, complete module: Three `AX` sources (`IN1`, `IN2`, `IN3`) are ORed together to form a single signal and then distributed to two destinations (`OUT1`, `OUT2`).



Function Blocks (FBs) Used

Sub-Blocks: AX_3_2

- **Type**: SubAppType
- **Internal FBs Used**:
  - **AX_OR_3**: `adapter::booleanOperators::AX_OR_3` — ORs `IN1`/`IN2`/`IN3`.
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — splits the OR result to two outputs.
- **How it works**: `IN1`/`IN2`/`IN3` ⟶ `AX_OR_3.OUT` ⟶ `AX_SPLIT_2.IN` ⟶ `OUT1`/`OUT2`.

Program Flow and Connections

1. `IN1` ⟶ `AX_OR_3.IN1`; `IN2` ⟶ `AX_OR_3.IN2`; `IN3` ⟶ `AX_OR_3.IN3`.
2. `AX_OR_3.OUT` ⟶ `AX_SPLIT_2.IN`.
3. `AX_SPLIT_2.OUT1` ⟶ `OUT1`; `AX_SPLIT_2.OUT2` ⟶ `OUT2`.

Application Scenarios

- Merging three `AX` sources (e.g., three command paths) whose combined output must be distributed to two independent destinations.

Comparison with Similar Building Blocks

For other source/destination counts: `AX_2_3` (2⟶3), `AX_3_3` (3⟶3).

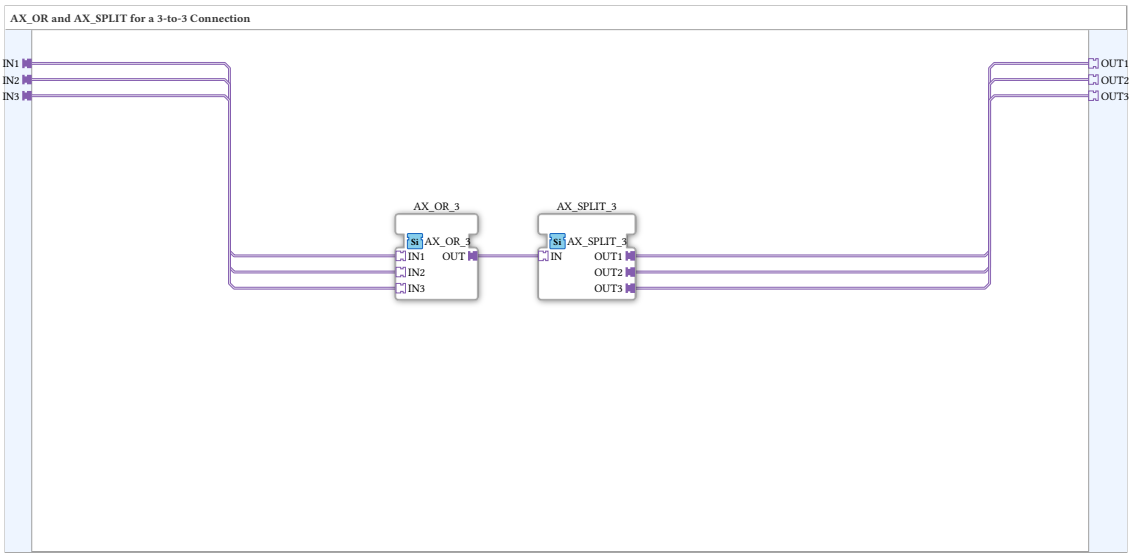
Summary

`AX_3_2` is a simple wiring aid that combines three `AX` sources using an OR operation and distributes them to two destinations—without its own state logic.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

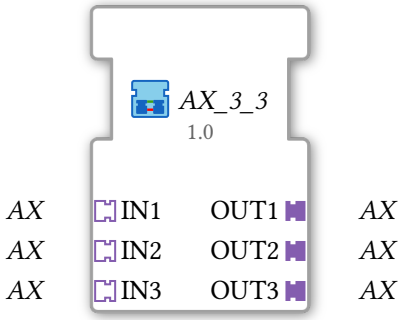
AX_3_3



• – ○ • – ○ • – ○ •

Introduction

`AX_3_3` combines `AX_OR_3` and `AX_SPLIT_3` into a single, complete module: Three `AX` sources (`IN1`, `IN2`, `IN3`) are ORed together to form a single signal and then distributed to three destinations (`OUT1`, `OUT2`, `OUT3`).



Function Blocks (FBs) Used

Sub-Blocks: AX_3_3

- **Type**: SubAppType
- **Internal FBs Used**:
  - **AX_OR_3**: `adapter::booleanOperators::AX_OR_3` — ORs `IN1`/`IN2`/`IN3`.
  - **AX_SPLIT_3**: `adapter::events::unidirectional::AX_SPLIT_3` — distributes the OR result to three outputs.
- **How it works**: `IN1`/`IN2`/`IN3` ⟶ `AX_OR_3.OUT` ⟶ `AX_SPLIT_3.IN` ⟶ `OUT1`/`OUT2`/`OUT3`.

Program Flow and Connections

1. `IN1` ⟶ `AX_OR_3.IN1`; `IN2` ⟶ `AX_OR_3.IN2`; `IN3` ⟶ `AX_OR_3.IN3`.
2. `AX_OR_3.OUT` ⟶ `AX_SPLIT_3.IN`.
3. `AX_SPLIT_3.OUT1` ⟶ `OUT1`; `AX_SPLIT_3.OUT2` ⟶ `OUT2`; `AX_SPLIT_3.OUT3` ⟶ `OUT3`.

Application Scenarios

- Merging three `AX` sources (e.g., three command paths) whose combined output must be distributed to three independent destinations.

Comparison with Similar Building Blocks

For other source/destination counts: `AX_2_3` (2⟶3), `AX_3_2` (3⟶2).

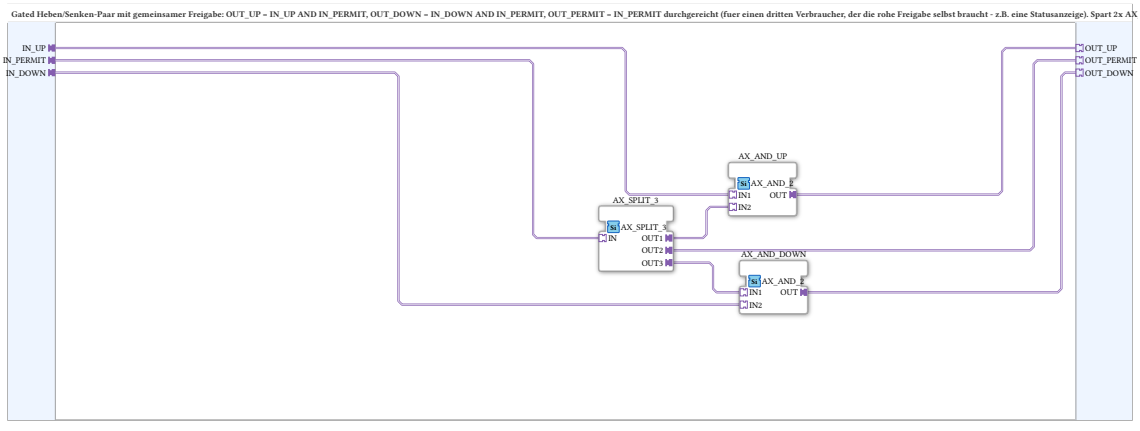
Summary

`AX_3_3` is a simple wiring aid that combines three `AX` sources using an OR operation and distributes them to three destinations—without its own state logic.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

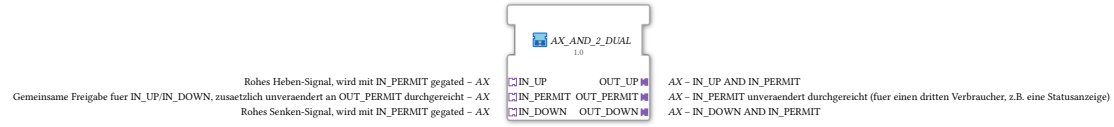
AX_AND_2_DUAL



• – ○ • – ○ • – ○ •

Introduction

`AX_AND_2_DUAL` is a gated raise/lower pair with a common permit: `OUT_UP = IN_UP AND IN_PERMIT` and `OUT_DOWN = IN_DOWN AND IN_PERMIT`. In addition, `IN_PERMIT` is passed through unchanged as `OUT_PERMIT`, e.g. for a third consumer such as a status display. The SubApp replaces two `AX_AND_2` plus one `AX_SPLIT_3` as individual blocks.



Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

All adapters are of type `adapter::types::unidirectional::AX`.

- **IN_UP** (socket): raw raise signal, gated with `IN_PERMIT`
- **IN_PERMIT** (socket): common permit for `IN_UP`/`IN_DOWN`, additionally passed through unchanged to `OUT_PERMIT`
- **IN_DOWN** (socket): raw lower signal, gated with `IN_PERMIT`
- **OUT_UP** (plug): `IN_UP AND IN_PERMIT`
- **OUT_PERMIT** (plug): `IN_PERMIT` unchanged (for a third consumer, e.g. a status display)
- **OUT_DOWN** (plug): `IN_DOWN AND IN_PERMIT`

Function Blocks (FBs) Used

- **`AX_SPLIT_3`** (`adapter::events::unidirectional::AX_SPLIT_3`): distributes `IN_PERMIT` to three targets
- **`AX_AND_UP`**, **`AX_AND_DOWN`** (each `adapter::booleanOperators::AX_AND_2`): AND combination

Functionality

- `IN_PERMIT` ⟶ `AX_SPLIT_3.IN`
- `AX_SPLIT_3.OUT1` ⟶ `AX_AND_UP.IN2`; `IN_UP` ⟶ `AX_AND_UP.IN1`; `AX_AND_UP.OUT` ⟶ `OUT_UP`
- `AX_SPLIT_3.OUT3` ⟶ `AX_AND_DOWN.IN1`; `IN_DOWN` ⟶ `AX_AND_DOWN.IN2`; `AX_AND_DOWN.OUT` ⟶ `OUT_DOWN`
- `AX_SPLIT_3.OUT2` ⟶ `OUT_PERMIT`

Technical Details

- Since an adapter must not branch to several targets, `AX_SPLIT_3` distributes the permit to both AND elements and the pass-through output
- No logic of its own beyond the standard blocks used

Application Scenarios

- Any raise/lower signal pair that needs the same permit (e.g. an interlock), where the permit should additionally be available for a display

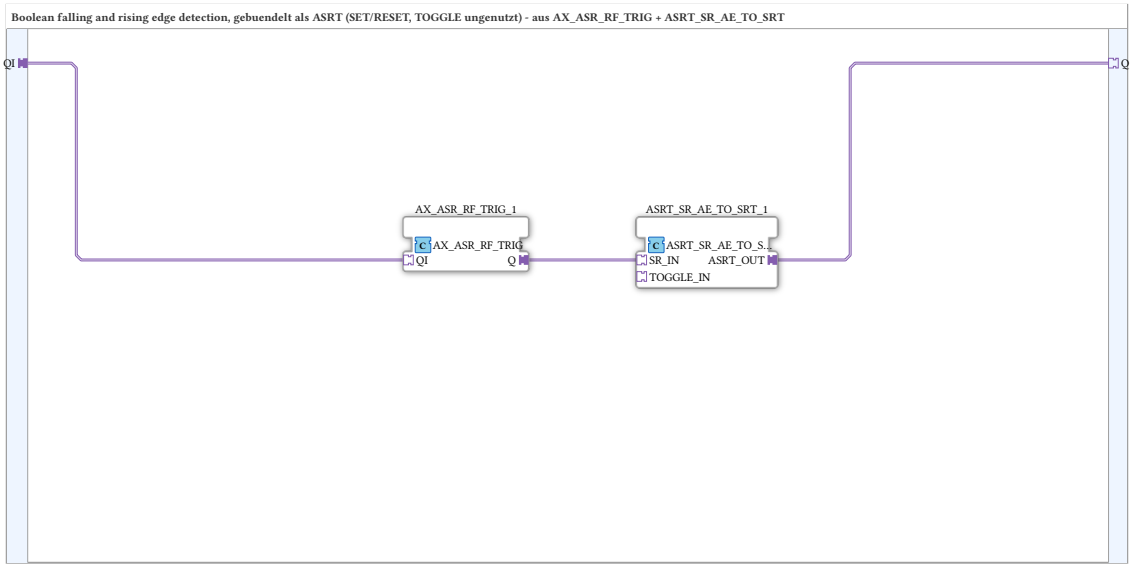
Summary

`AX_AND_2_DUAL` gates a raise/lower pair with a common permit and additionally passes the permit through.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AX_ASRT_RF_TRIG



• – ○ • – ○ • – ○ •

Introduction

`AX_ASRT_RF_TRIG` detects rising and falling edges of an `AX` signal and combines them as an `ASRT` adapter (Set/Reset, Toggle remains unused). Instead of a new low-level `FBType`, the function block is constructed as a composite of the existing `AX_ASR_RF_TRIG` (which returns an `ASR` from an edge) and `ASRT_SR_AE_TO_SRT` (which combines an `ASR` with an optional AE toggle event to create an `ASRT`).

value to check for a rising or falling edge – *AX*



ASRT – Set/Reset Output (Toggle wird nie gefeuert)

Function Blocks (FBs) Used

Sub-Blocks: AX_ASRT_RF_TRIG

- **Type**: SubAppType
- **Internal FBs Used**:
- **AX_ASR_RF_TRIG_1**: `adapter::events::unidirectional::AX_ASR_RF_TRIG` — rising edge at `QI` ⟶ SET, falling edge ⟶ RESET, bundled as `ASR`.
- **ASRT_SR_AE_TO_SRT_1**: `adapter::conversion::unidirectional::ASRT_SR_AE_TO_SRT` — takes `ASR` as `SR_IN` and bundles it to `ASRT_OUT`; The second socket (`TOGGLE_IN`, a single AE event) remains unwired and never fires, as a pure edge signal does not allow for the derivation of an independent third event.
- **Functionality**: `AX_ASR_RF_TRIG_1` generates a `ASR` from the edge of `QI`. This `SR_IN` is directly fed into `ASRT_SR_AE_TO_SRT_1`, whose `ASRT_OUT` then serves the plug `Q`.

Program Flow and Connections

1. `QI` (Socket) ⟶ `AX_ASR_RF_TRIG_1.QI`.
2. `AX_ASR_RF_TRIG_1.Q` ⟶ `ASRT_SR_AE_TO_SRT_1.SR_IN`.
3. `ASRT_SR_AE_TO_SRT_1.ASRT_OUT` ⟶ `Q` (Plug).

Technical Features

- **Composite instead of a new low-level component**: Reuses two existing components instead of implementing a new `FBType`.
- **TOGGLE socket remains unused**: `ASRT_SR_AE_TO_SRT_1.TOGGLE_IN` is never wired and therefore never fires—the component only provides SET/RESET, never a true toggle event.

Application Scenarios

- As one of several `ASRT` sources for a `ASRT_MERGE_2`, e.g., in `Uebung_232_AX`.

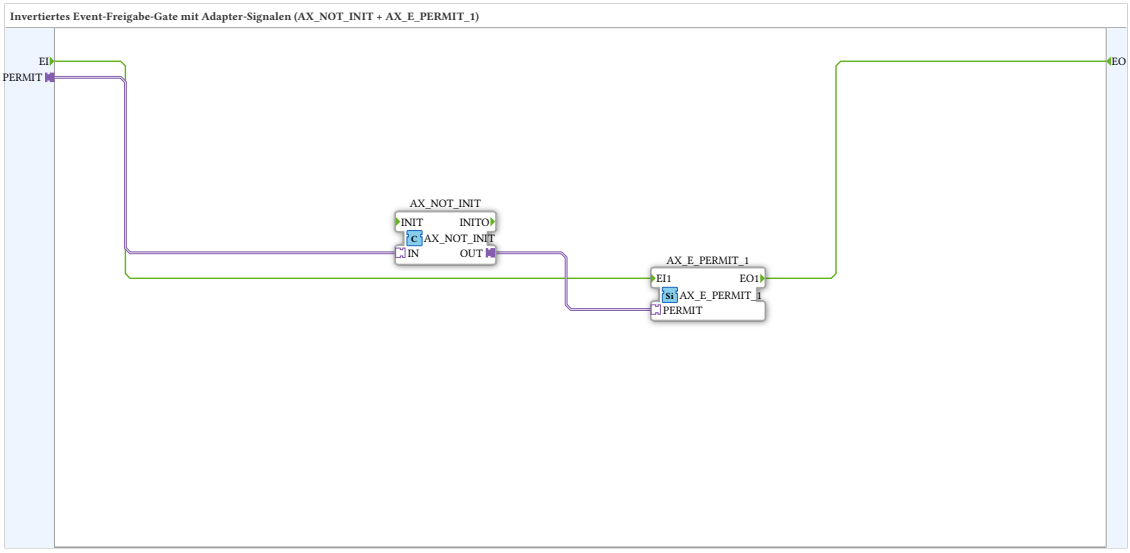
Summary

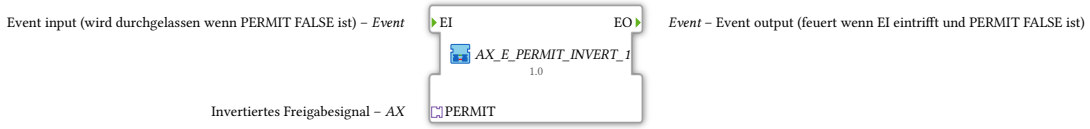
`AX_ASRT_RF_TRIG` delivers a `ASRT` adapter signal (Set/Reset only, Toggle unused) from a single edge detection and can therefore be directly connected to devices that expect a `ASRT` socket.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AX_E_PERMIT_INVERT_1





• – ○ • – ○ • – ○ •

Introduction

AX_E_PERMIT_INVERT_1 is a composite subapplication that implements an inverted event-permission gate. It combines the adapter-based blocks `AX_NOT_INIT` and `AX_E_PERMIT_1` to allow an event to pass from the event input `EI` to the event output `EO` only when the adapter signal `PERMIT` is **FALSE**.

This subapp is useful when an external permission condition is expressed through a unidirectional adapter and the desired logic is “pass the event when the permission is not active”.

Interface Structure

Event Inputs

Name	Type	Description
`EI`	Event	Event input. The event is forwarded to `EO` only when the `PERMIT` adapter value is FALSE.

Event Outputs

Name	Type	Description
EO	Event	Event output. Fires when an event arrives on EI and PERMIT is FALSE.

Data Inputs

There are no data inputs.

Data Outputs

There are no data outputs.

Adapters

Name	Type	Direction	Description
PERMIT	adapter::types::unidirectional::AX	Socket	Adapter signal used as the gate condition. Events are allowed to pass when this signal is FALSE.

Functionality

The internal network consists of two connected adapter-based function blocks:

1. AX_NOT_INIT
   Receives the PERMIT adapter signal and produces its logical inversion on OUT.
2. AX_E_PERMIT_1
   Receives the inverted signal on its PERMIT adapter input. It forwards events from its event input EI1 to its event output EO1 when the inverted signal is TRUE.

The event flow is:

- An event at EI is connected to AX_E_PERMIT_1.EI1.
- If the external PERMIT signal is FALSE, AX_NOT_INIT outputs TRUE.
- AX_E_PERMIT_1 therefore opens the gate and propagates the event to EO.
- If the external PERMIT signal is TRUE, the gate remains closed and the incoming event is not forwarded.

In other words, the subapp behaves as an event gate with inverted permission logic:

External PERMIT	Event EI	Event EO
FALSE	arrives	fires
TRUE	arrives	does not fire

Technical Features

- IEC 61499-2 compliant subapplication type.
- No data inputs or data outputs.
- One unidirectional adapter socket PERMIT.
- One event input and one event output.
- Internally composed of two reusable adapter-based function blocks.
- The event path and the adapter condition path are strictly separated.
- The inversion of the adapter signal is encapsulated inside the subapp, simplifying the external interface.

State Overview

The subapp does not expose an internal state machine. Its behavior is event-driven and depends on the current value of the PERMIT adapter at the moment an event arrives on EI.

There is no latching or retention of events. If the gate is closed when EI occurs, that event is discarded. If PERMIT changes afterward, the next event on EI is evaluated using the new adapter value.

Application Scenarios

- **Active-low permission logic**
  Useful when a system should process an event only while a permission signal is not active.

- **Adapter-based event filtering**
  Can be placed in an event chain to suppress events when an adapter condition is TRUE.
- **Polarity conversion in larger subapplications**
  Allows standard `AX_E_PERMIT_1` logic to be reused with an inverted enable signal without additional external wiring.
- **Safety and interlock patterns**
  Suitable for cases where an event must only be forwarded when a certain release condition is absent, such as “not in manual mode” or “no suppression active”.

Comparison with Similar Blocks

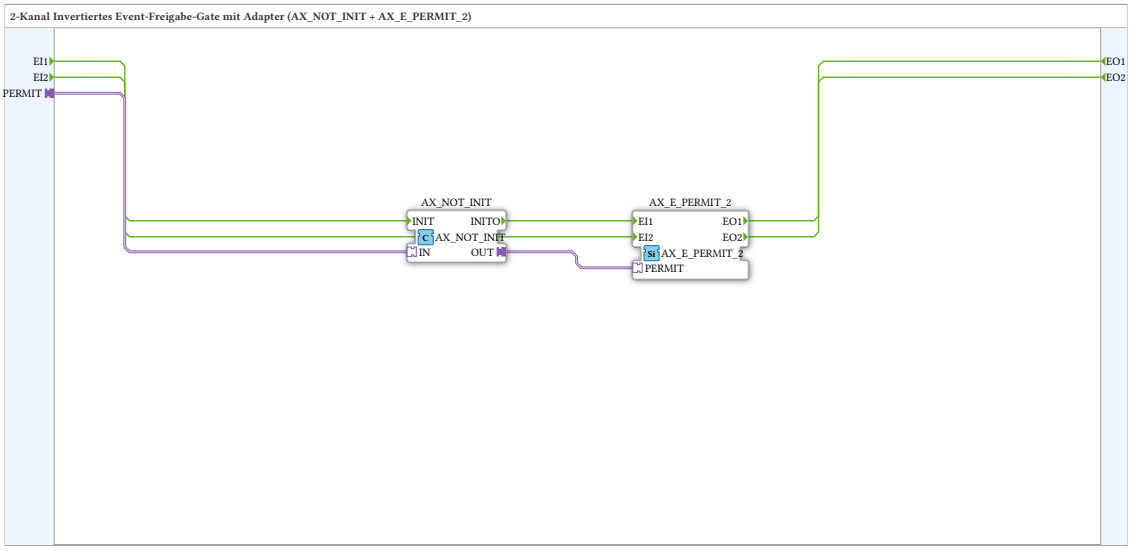
Block / Subapp	Gate Condition	Behavior
`AX_E_PERMIT_1`	Adapter signal TRUE	Forwards event when the adapter signal is TRUE.
`AX_E_PERMIT_INVERT_1`	External adapter signal FALSE	Forwards event when the external adapter signal is FALSE.
`E_PERMIT` (classic)	BOOL data input TRUE	Forwards event when the BOOL permission input is TRUE.

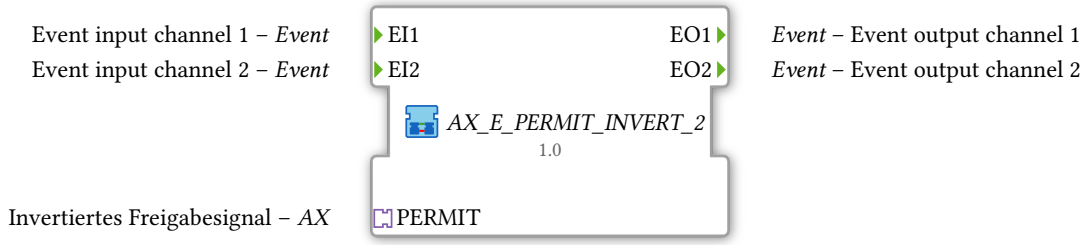
The main difference is that `AX_E_PERMIT_INVERT_1` encapsulates the inversion of an adapter signal. This makes it directly usable in adapter-based architectures where the permission condition is already provided as a unidirectional `AX` adapter.

Conclusion

AX_E_PERMIT_INVERT_1 provides a clean, reusable way to implement an inverted event-permission gate. By combining `AX_NOT_INIT` and `AX_E_PERMIT_1`, it converts a permit adapter signal into a blocking condition and only passes events when the permit is FALSE. This makes it a useful building block for adapter-oriented IEC 61499 applications that require active-low event gating.

AX_E_PERMIT_INVERT_2





• – ◦ • – ◦ • – ◦ •

Introduction

The **AX_E_PERMIT_INVERT_2** is a composite subapplication that provides a two-channel inverted event gating mechanism. It combines a boolean NOT operator (`AX_NOT_INIT`) with a two-channel event permit gate (`AX_E_PERMIT_2`) to create an event pass-through logic where events are only forwarded when the externally supplied permit signal is **inactive** (i.e., `FALSE`). This is particularly useful in safety-oriented control systems where events must be suppressed unless an explicit release condition is met.

Interface Structure

The subapplication exposes two event inputs, two event outputs, and one adapter socket. It contains no data inputs or data outputs.

Event Inputs

Name	Type	Description
`EI1`	Event	Event input channel 1
`EI2`	Event	Event input channel 2

Event Outputs

Name	Type	Description
`EO1`	Event	Event output channel 1
`EO2`	Event	Event output channel 2

Data Inputs

None.

Data Outputs

None.

Adapters

Name	Type	Description
`PERMIT`	`adapter::types::unidirectional::AX`	Inverted enable signal (socket)

Functionality

The subapplication implements a two-stage processing pipeline:

1. **Signal Inversion Stage**: The incoming `PERMIT` adapter signal is routed to the `AX_NOT_INIT` function block, which performs a logical NOT operation on the boolean value carried by the adapter.
2. **Event Gating Stage**: The inverted boolean result is forwarded to the `PERMIT` input of the `AX_E_PERMIT_2` function block. This block acts as an event gate that conditionally forwards incoming events based on the state of its permit input.

The event flow is straightforward:

- `EI1` ⟶ `AX_E_PERMIT_2.EI1` ⟶ `EO1`
- `EI2` ⟶ `AX_E_PERMIT_2.EI2` ⟶ `EO2`

Gating Behavior:

- When the external `PERMIT` signal is `FALSE`, the inverted signal becomes `TRUE`. The events arriving at `EI1`/`EI2` are passed through to `EO1`/`EO2`.
- When the external `PERMIT` signal is `TRUE`, the inverted signal becomes `FALSE`. The events arriving at `EI1`/`EI2` are **blocked** and not propagated to the outputs.

This inverted behavior distinguishes the block from a standard permit gate: events are allowed **by default** and are only suppressed when an explicit inhibit signal is asserted.

Technical Features

- **Two Identical Channels**: Both channels operate independently and identically, allowing simultaneous event processing on two separate paths.
- **Adapter-based Signal Interface**: The permit signal is exchanged via a unidirectional adapter type (`AX`), enabling seamless integration with other 4diac IDE components using the same adapter type.
- **Composite Design**: The subapplication is built from standard, reusable function blocks, making its internal logic transparent and maintainable.
- **No Internal State**: The absence of internal state variables makes the block deterministic and stateless with respect to event processing.
- **Eclipse Public License 2.0**: The implementation is distributed under EPL-2.0, allowing free use and modification.

State Overview

The subapplication does not maintain any internal state machine. Its behavior is purely combinational in the signal domain:

- The boolean inversion stage is static and immediate.
- The permit gate, `AX_E_PERMIT_2`, may maintain its own internal permit state, which is entirely determined by the inverted adapter signal.

Consequently, there are no distinct operational states to enumerate. The block behaves identically at all times, reacting to the current values of the permit signal and incoming events.

Application Scenarios

- **Safety Interlocks**: Suppress process events when a safety condition (e.g., emergency stop or guard open) is active, thereby preventing unintended operations.
- **Maintenance Mode**: Block event-triggered actions during maintenance windows by asserting the permit signal, while allowing normal operation when the signal is deasserted.
- **Inverted Permission Logic**: Implement logic where the absence of an explicit deny signal permits event processing, simplifying default-open policies.
- **Supervisory Control**: In multi-channel control systems, gate two independent event streams with a single shared inhibit signal.

Comparison with Similar Blocks

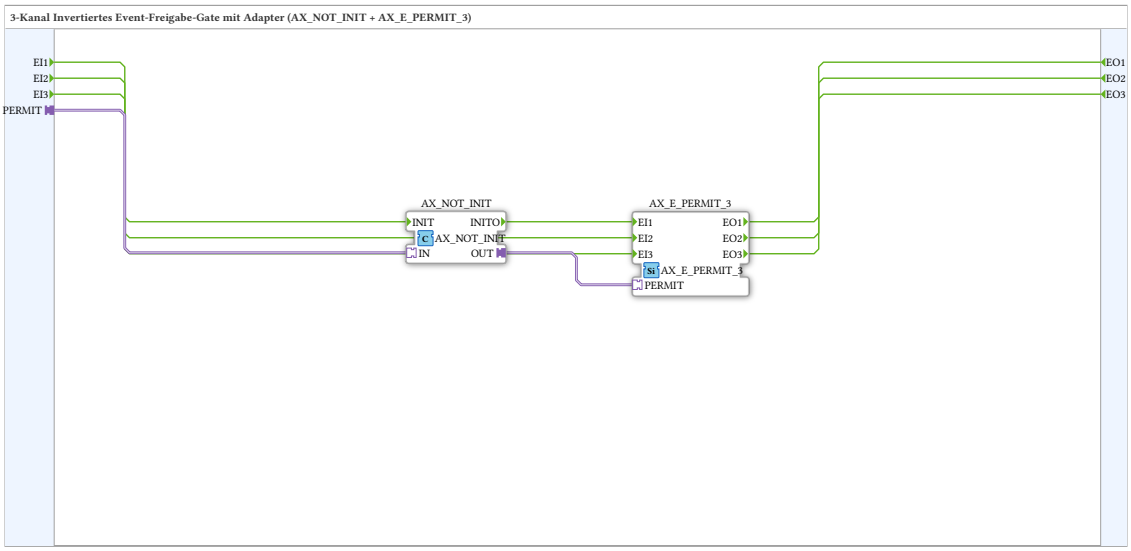
Feature	`AX_E_PERMIT_2`	`AX_E_PERMIT_INVERT_2`
Permit semantics	Events pass when permit is `TRUE`	Events pass when permit is `FALSE`
Number of channels	2	2
Inversion stage	None (direct permit)	Built-in `AX_NOT_INIT` inversion
Use case	Enable-based gating	Inhibit-based gating
Complexity	Lower	Slightly higher (added NOT operator)

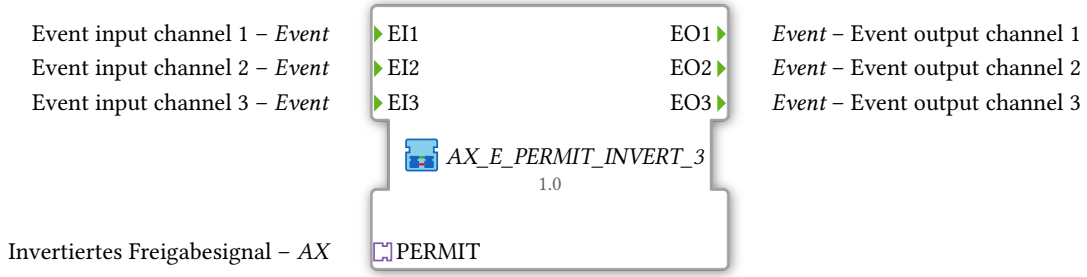
The inverted variant is functionally equivalent to inserting a NOT gate between the permit source and a standard permit block, but encapsulates this logic into a single reusable component.

Conclusion

The **AX_E_PERMIT_INVERT_2** subapplication provides a clean and reusable solution for two-channel event gating with inverted permit semantics. By integrating a boolean inversion stage with a standard event permit gate, it enables applications where events flow freely until an explicit inhibit signal is applied. Its adapter-based interface ensures compatibility with the broader 4diac ecosystem, while the composite structure keeps the logic transparent and easily maintainable. This block is particularly well suited for safety, maintenance, and supervisory control applications requiring default-open event behavior.

AX_E_PERMIT_INVERT_3





• – ○ • – ○ • – ○ •

Introduction

AX_E_PERMIT_INVERT_3 is a composite subapplication for 3-channel event gating with an inverted adapter-based enable signal. It combines an adapter inverter AX_NOT_INIT and a 3-channel event permit gate AX_E_PERMIT_3. The subapplication forwards events only when the external PERMIT signal is FALSE; when PERMIT is TRUE, all event channels are blocked.

Interface Structure

Event Inputs

Name	Type	Description
EI1	Event	Event input channel 1
EI2	Event	Event input channel 2
EI3	Event	Event input channel 3

Event Outputs

Name	Type	Description
EO1	Event	Event output channel 1
EO2	Event	Event output channel 2
EO3	Event	Event output channel 3

Data Inputs

This subapplication has no explicit data inputs.

Data Outputs

This subapplication has no explicit data outputs.

Adapters

Name	Type	Direction	Description
PERMIT	adapter::types::unidirectional::AX	Socket	Active-low permit signal. The boolean value carried by this adapter is inverted internally; events are passed only when this value is FALSE.

Functionality

The function is achieved by two internal components:

1. `AX_NOT_INIT` acts as a logical NOT operator on the adapter-based boolean signal connected to `PERMIT`.
2. `AX_E_PERMIT_3` is a three-channel event gate. It receives events from `EI1` … `EI3` and forwards them to `EO1` … `EO3` only when its own `PERMIT` input is active.

The external `PERMIT` adapter is connected to `AX_NOT_INIT.IN`. The inverted result is sent from `AX_NOT_INIT.OUT` to `AX_E_PERMIT_3.PERMIT`. Thus, the effective gate condition is the opposite of the external signal:

External PERMIT	Signal seen by gate	Event behavior
FALSE	TRUE	Events are forwarded
TRUE	FALSE	Events are blocked

Each event channel is independent: `EI1` is routed through the gate to `EO1`, `EI2` to `EO2`, and `EI3` to `EO3`. The same inverted permit value applies to all three channels.

Technical Features

- Three independent event input/output channels.
- Inverted enable behavior using an adapter-based boolean inverter.
- No explicit data inputs or outputs; the permission state is transported through the `PERMIT` adapter.
- One unidirectional adapter socket of type `adapter::types::unidirectional::AX`.
- Internal composition consists of `AX_NOT_INIT` and `AX_E_PERMIT_3`.
- The subapplication has no ECC or state machine; it is a purely event-routing composition.

State Overview

This subapplication does not define internal persisting states. The behavior is determined solely by the current boolean value carried by the `PERMIT` adapter. From an external perspective, the gate is either open or closed:

- **Open**: External `PERMIT` is `FALSE`, the gate receives an active permit, and arriving events are released on the matching event output.
- **Closed**: External `PERMIT` is `TRUE`, the gate receives an inactive permit, and arriving events are suppressed.

Because there is no internal state memory, the behavior is deterministic and independent of previous event occurrences.

Application Scenarios

Typical use cases include:

- **Active-low enable signals**: An application that should run while an enable signal is de-asserted.
- **Safety/inhibit gating**: A machine control event may be allowed only while an inhibit flag is `FALSE`.
- **Multi-channel event distribution**: Three related event paths, such as preparation, execution, and completion, can be opened or closed together using one common inverted permission signal.
- **Adapter-based software architectures**: Used in IEC 61499 systems where boolean status information is exchanged via unidirectional adapters rather than separate data connections.

Comparison with Similar Blocks

AX_E_PERMIT_3 is the direct non-inverted counterpart. It forwards events when its PERMIT adapter value is TRUE. AX_E_PERMIT_INVERT_3 inserts AX_NOT_INIT before the same gate, so the permission condition becomes inverted. In systems where the available inhibit signal is active-high, this avoids manually inverting the signal outside the subapplication.

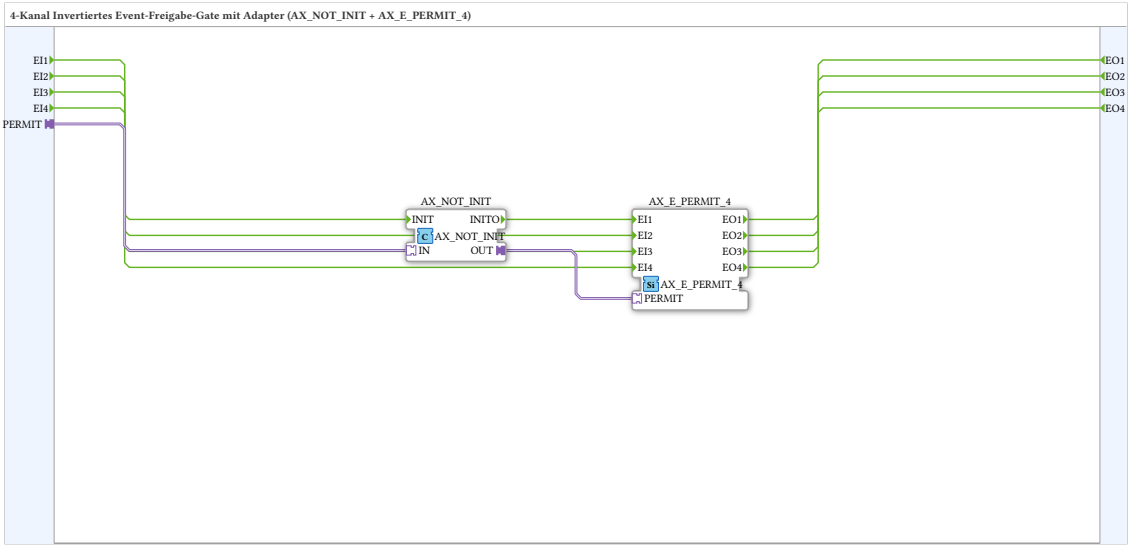
Block	Permits events when external PERMIT is
AX_E_PERMIT_3	TRUE
AX_E_PERMIT_INVERT_3	FALSE

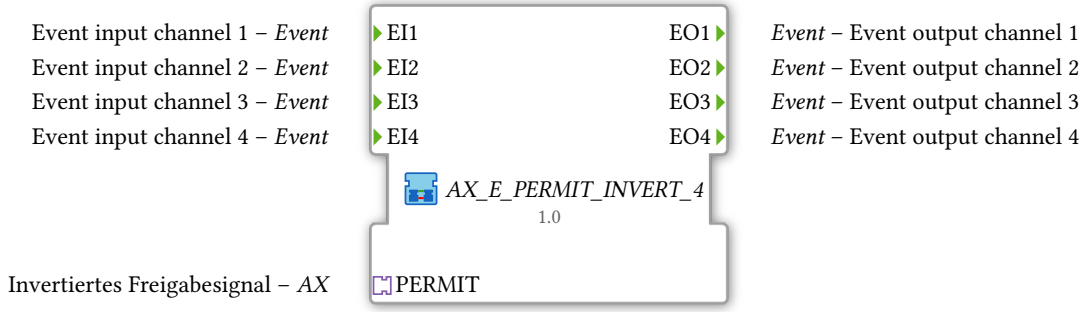
This makes the block suitable for applications that need an event gate with active-low enable logic while retaining the same channel structure as the standard AX_E_PERMIT_3.

Conclusion

AX_E_PERMIT_INVERT_3 is a compact, reusable subapplication for event gating with inverted adapter-based permission. By combining an adapter inverter with a 3-channel event gate, it provides a clean solution for applications that must pass events only while the permit signal is FALSE. Its simple internal structure and absence of data pins make it easy to integrate in event-driven IEC 61499 systems.

AX_E_PERMIT_INVERT_4





• – ○ • – ○ • – ○ •

Introduction

The **AX_E_PERMIT_INVERT_4** is a 4-channel inverted event permit gate implemented as a subapplication type. It combines the adapter logic block **AX_NOT_INIT** with the 4-channel event gate **AX_E_PERMIT_4**.

The subapplication receives four independent event inputs and forwards them to four event outputs only when the external adapter-based permit signal is inactive. The incoming PERMIT adapter is inverted

first, so the internal event gate sees the negated permission value. This provides a simple and reusable way to suppress or enable event streams using an active-low adapter signal.

Interface Structure

Event Inputs

Name	Type	Description
EI1	Event	Event input channel 1
EI2	Event	Event input channel 2
EI3	Event	Event input channel 3
EI4	Event	Event input channel 4

Event Outputs

Name	Type	Description
EO1	Event	Event output channel 1
EO2	Event	Event output channel 2
EO3	Event	Event output channel 3
EO4	Event	Event output channel 4

Data Inputs

None.

Data Outputs

None.

Adapters

Name	Type	Description
`PERMIT`	`adapter::types::unidirectional::AX`	Inverted enable signal for all event channels

Functionality

The subapplication uses two internal function blocks:

- **AX_NOT_INIT** inverts the adapter signal connected to its input.
- **AX_E_PERMIT_4** passes or blocks event inputs depending on the permit signal at its adapter socket.

The external `PERMIT` adapter is connected to `AX_NOT_INIT.IN`. The inverted adapter output `AX_NOT_INIT.OUT` is then connected to `AX_E_PERMIT_4.PERMIT`.

Each event input is directly wired to the corresponding event input of `AX_E_PERMIT_4`, and each internal event output is connected to the corresponding external event output:

- `EI1` ⟶ `AX_E_PERMIT_4.EI1` ⟶ `EO1`
- `EI2` ⟶ `AX_E_PERMIT_4.EI2` ⟶ `EO2`
- `EI3` ⟶ `AX_E_PERMIT_4.EI3` ⟶ `EO3`
- `EI4` ⟶ `AX_E_PERMIT_4.EI4` ⟶ `EO4`

As a result:

- If the external `PERMIT` signal is **false**, the internal permit becomes **true**, and incoming events are forwarded to the outputs.
- If the external `PERMIT` signal is **true**, the internal permit becomes **false**, and incoming events are blocked.

Technical Features

- Four independent event channels.

- Single common adapter socket for centralized event gating.
- Active-low permit behavior due to internal signal inversion.
- Implemented as a subapplication network using `AX_NOT_INIT` and `AX_E_PERMIT_4`.
- No data inputs or outputs are required.
- Designed for use in IEC 61499-compliant systems with unidirectional adapter types.
- Easy to reuse and integrate into larger 4diac applications.

State Overview

The subapplication itself does not contain an explicit state machine. Its behavior is determined by the combined logic of the internal function blocks.

The effective state depends on the current value of the `PERMIT` adapter signal:

External `PERMIT`	Internal permit	Event forwarding
`false`	`true`	Enabled
`true`	`false`	Disabled

The internal `AX_E_PERMIT_4` gate handles the actual per-channel event passing based on this internal permit state.

Application Scenarios

- **Active-low event enabling**: When the permit signal is used as a disable or inhibit flag, events are passed only while the flag is inactive.
- **Central gating of multiple event streams**: A single adapter signal can control four independent event paths at once.
- **Conditional event propagation**: Events can be forwarded or suppressed depending on external system states such as ready signals, safety conditions, or operating modes.
- **Composite subapplication design**: Useful as a reusable building block inside larger 4diac applications that require inverted permit logic.

Comparison with Similar Blocks

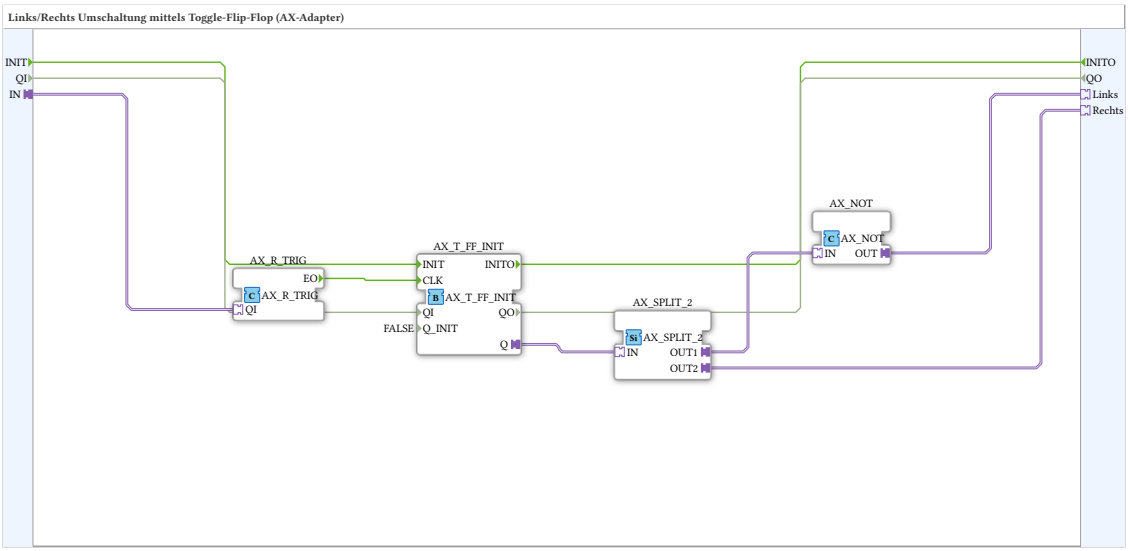
Block	Behavior
`AX_E_PERMIT_4`	Passes events while the permit adapter signal is true.
`AX_E_PERMIT_INVERT_4`	Passes events while the permit adapter signal is false.
`AX_NOT_INIT`	Inverts an adapter signal and is used here to negate the permit input before the event gate.

The main difference to the standard `AX_E_PERMIT_4` is the integrated inversion. This removes the need to manually insert an inverter block in the application network when an active-low permission is required.

Conclusion

The **AX_E_PERMIT_INVERT_4** subapplication provides a compact and reusable solution for 4-channel event gating with an inverted adapter-based permit signal. By combining `AX_NOT_INIT` and `AX_E_PERMIT_4`, it offers a clear, structured way to block or forward multiple event streams using a single active-low control signal. It is especially useful in IEC 61499 applications where conditional event propagation and centralized gating are required.

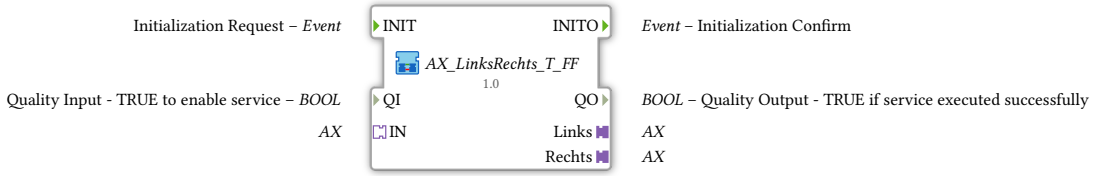
AX_LinksRechts_T_FF



• – ○ • – ○ • – ○ •

Introduction

`AX_LinksRechts_T_FF` ("left/right toggle flip-flop") converts a single button press (`IN`) via a toggle flip-flop into two complementary adapter outputs `Links`/`Rechts` (left/right) — each button press switches between "left active" and "right active".



Interface Structure

- **Event inputs**: `INIT` (initialization request)
- **Event outputs**: `INITO` (initialization confirm)
- **Data inputs**: `QI` (`BOOL`, default `TRUE`) — quality input
- **Data outputs**: `QO` (`BOOL`) — quality output
- **Adapters**: socket `IN`, plugs `Links` and `Rechts` (each `adapter::types::unidirectional::AX`)

Function Blocks (FBs) Used

Sub-blocks: AX_LinksRechts_T_FF

- **Type**: SubAppType
- **Internal FBs used**:
  - **AX_R_TRIG**: `adapter::events::unidirectional::AX_R_TRIG` — detects the button's rising edge.
  - **AX_T_FF_INIT**: `adapter::events::unidirectional::AX_T_FF_INIT` — toggle flip-flop with a defined initial state (`Q_INIT=FALSE`; `QI` comes from the wrapper input `QI`).
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — splits the flip-flop state.
  - **AX_NOT**: `adapter::booleanOperators::AX_NOT` — negates one of the two branches to produce the complementary signal.
- **Functionality**: Every rising edge on `IN` toggles `AX_T_FF_INIT.Q`. The direct state feeds `Rechts` (right), the negated state feeds `Links` (left) — so `Links` and `Rechts` are always exactly complementary (per the source comment: "AX_T_FF_INIT switches the Active output to FALSE, so LEFT is then TRUE").

Program Flow and Connections

1. `IN` ⟶ `AX_R_TRIG.QI`; `AX_R_TRIG.EO` ⟶ `AX_T_FF_INIT.CLK` (toggles on every rising edge).
2. `AX_T_FF_INIT.Q` ⟶ `AX_SPLIT_2.IN`.
3. `AX_SPLIT_2.OUT1` ⟶ `AX_NOT.IN` ⟶ `AX_NOT.OUT` ⟶ `Links`.

4. `AX_SPLIT_2.OUT2` ⟶ `Rechts`.
5. `INIT` ⟶ `AX_T_FF_INIT.INIT`; `AX_T_FF_INIT.INITO` ⟶ `INITO`.
6. `QI` ⟶ `AX_T_FF_INIT.QI`; `AX_T_FF_INIT.QO` ⟶ `QO`.

Application Scenarios

- Single-button switching between two mutually exclusive states (e.g. direction of travel, display toggling) without needing two separate buttons.

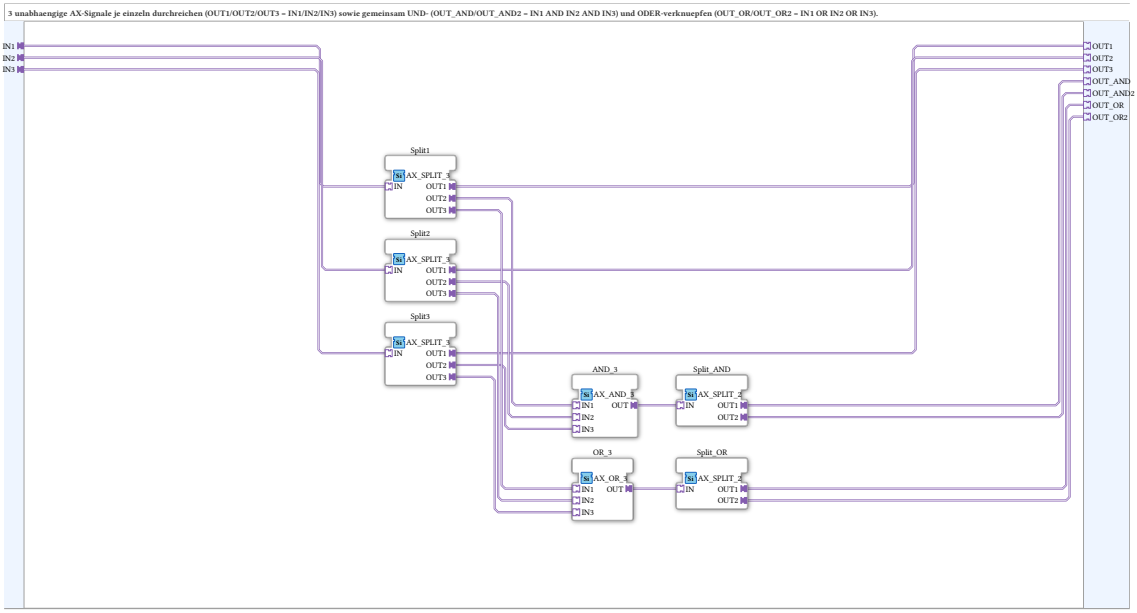
Summary

`AX_LinksRechts_T_FF` implements a classic single-button toggle between two complementary states, using a toggle flip-flop with a defined initial state and a negation for the opposite signal.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

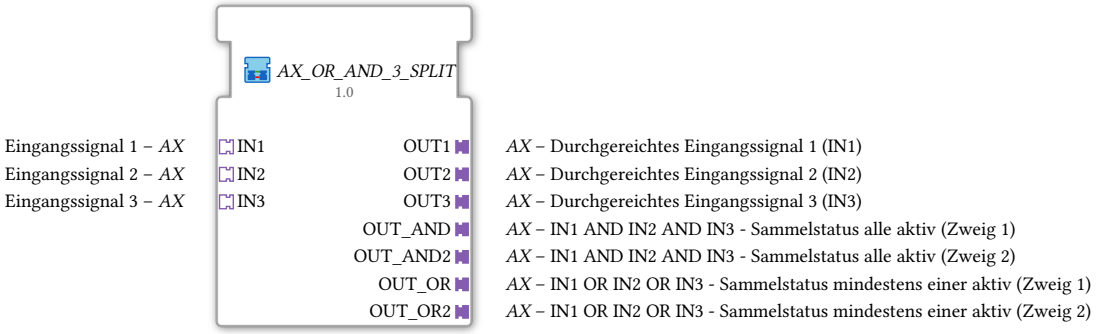
AX_OR_AND_3_SPLIT



• – ○ • – ○ • – ○ •

Introduction

`AX_OR_AND_3_SPLIT` passes **three independent AX signals** through individually (`OUT1`/`OUT2`/`OUT3` = `IN1`/`IN2`/`IN3`) and additionally combines them jointly with **AND** (`OUT_AND`/`OUT_AND2` = `IN1 AND IN2 AND IN3`) and **OR** (`OUT_OR`/`OUT_OR2` = `IN1 OR IN2 OR IN3`). The combined outputs are each provided twice (branch 1/branch 2).



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

All adapters are unidirectional of type `adapter::types::unidirectional::AX`.

- **IN1**, **IN2**, **IN3** (sockets): input signals 1 to 3
- **OUT1**, **OUT2**, **OUT3** (plugs): passed-through input signals 1 to 3
- **OUT_AND**, **OUT_AND2** (plugs): `IN1 AND IN2 AND IN3` - collective status "all active" (branch 1/ branch 2)
- **OUT_OR**, **OUT_OR2** (plugs): `IN1 OR IN2 OR IN3` - collective status "at least one active" (branch 1/ branch 2)

Function Blocks (FBs) Used

- **Split1**, **Split2**, **Split3**: each `adapter::events::unidirectional::AX_SPLIT_3` (branches each input signal onto three targets)
- **AND_3**: `adapter::booleanOperators::AX_AND_3`
- **OR_3**: `adapter::booleanOperators::AX_OR_3`
- **Split_AND**, **Split_OR**: each `adapter::events::unidirectional::AX_SPLIT_2` (duplicate the respective collective output)

Program Flow and Connections

- `INx` ⟶ `Splitx.IN`
- `Splitx.OUT1` ⟶ `OUTx` (pass-through)
- `Splitx.OUT2` ⟶ `AND_3.INx`
- `Splitx.OUT3` ⟶ `OR_3.INx`
- `AND_3.OUT` ⟶ `Split_AND.IN`; `Split_AND.OUT1` ⟶ `OUT_AND`, `Split_AND.OUT2` ⟶ `OUT_AND2`
- `OR_3.OUT` ⟶ `Split_OR.IN`; `Split_OR.OUT1` ⟶ `OUT_OR`, `Split_OR.OUT2` ⟶ `OUT_OR2`

(with x = 1, 2, 3)

Technical Details

- The AND and OR collective outputs are doubled (`AX_SPLIT_2`) so that two consumers each can be supplied.

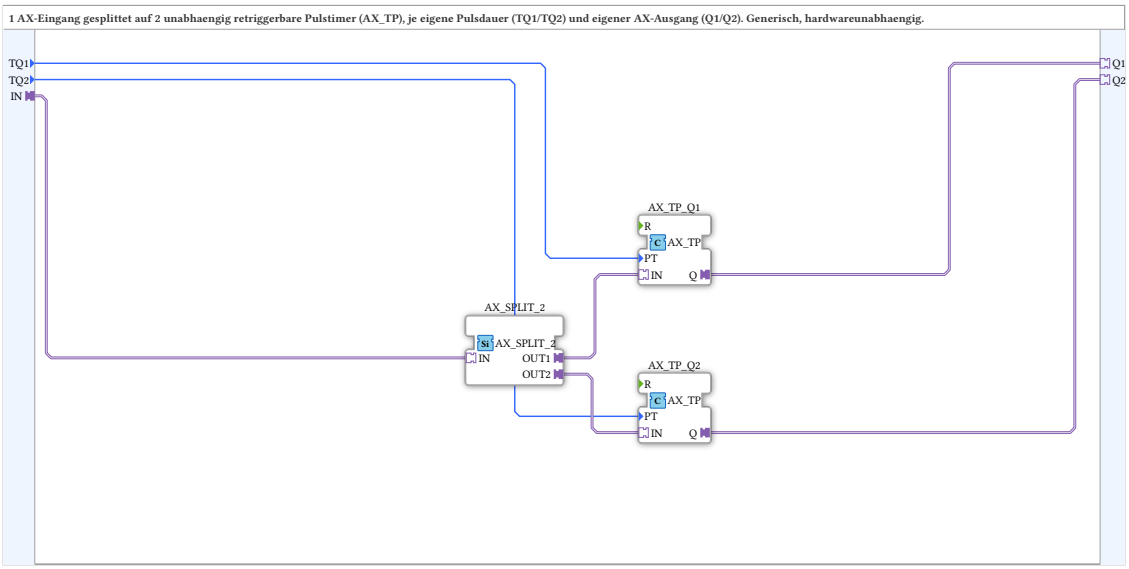
Application Scenarios

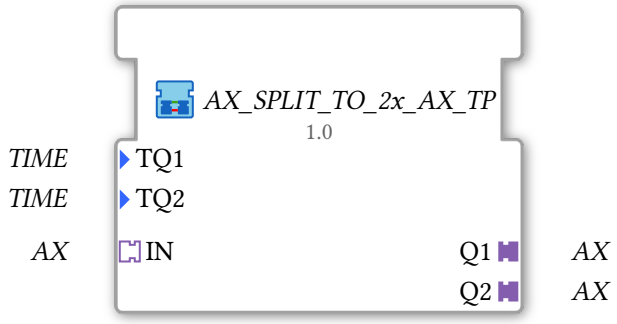
- Using several status signals (e.g. belts) individually and additionally forming a common status "all active" or "at least one active"

Summary

`AX_OR_AND_3_SPLIT` bundles pass-through, AND and OR combination of three AX signals in one SubApp.

AX_SPLIT_TO_2x_AX_TP





• – ○ • – ○ • – ○ •

Introduction

This subapplication provides a generic solution for splitting a single AX (axis) event input into two independent retriggerable pulse timers. Each timer has its own adjustable pulse duration and produces a separate AX output. The subapp is hardware-independent and can be reused in various control scenarios.

Interface Structure

Event Inputs

There are no direct event inputs. However, the adapter `IN` acts as an event-triggering input; events received on this adapter are processed internally.

Event Outputs

There are no direct event outputs. The generated pulse events are exposed via the adapter outputs `Q1` and `Q2`.

Data Inputs

Name	Type	Description
`TQ1`	`TIME`	Pulse duration for the first output (Q1).
`TQ2`	`TIME`	Pulse duration for the second output (Q2).

Data Outputs

None.

Adapters

Name	Type	Direction	Description
IN	adapter::types::unidirectional::AX	Input (Socket)	Receives the incoming AX event.
Q1	adapter::types::unidirectional::AX	Output (Plug)	Emits the pulse event for the first channel.
Q2	adapter::types::unidirectional::AX	Output (Plug)	Emits the pulse event for the second channel.

Functionality

The subapp uses an internal AX_SPLIT_2 instance to split the incoming AX event into two identical AX events. These events are then forwarded to two separate AX_TP timer instances (AX_TP_Q1 and AX_TP_Q2). Each timer is retriggerable: every new incoming event restarts the timing period. The pulse duration is independently set by TQ1 and TQ2. When a timing period elapses, the corresponding timer outputs an AX event on its Q adapter, which is then relayed to the respective subapp output (Q1 or Q2).

The split operation is transparent, meaning both channels receive the same trigger event. The timers operate independently, allowing different pulse widths and asynchronous behavior.

Technical Features

- **Generic and hardware-independent**: The subapp does not rely on any specific hardware platform.
- **Two independent retriggerable timers**: Each channel has its own timer and pulse duration.
- **Split of input event**: Uses the AX_SPLIT_2 block to duplicate the incoming event.
- **Easy configuration**: Pulse durations are set via data inputs.
- **Standard 61499 compliance**: Uses standard adapter types (unidirectional AX) and timers.

State Overview

The subapp does not maintain an explicit state machine; its behavior is determined by the internal AX_TP timers. The timers operate in a retriggerable manner: upon receiving an input event, they start (or restart) their timing cycle. When the timing period elapses, they emit an output event. While the timer is active, further input events reset the timing.

Application Scenarios

- **Control of two independent actuators** based on a single trigger signal.
- **Generation of two different pulse widths** for testing or calibration purposes.
- **Staggered pulse outputs** where each channel has its own response delay or duration.
- **Event distribution in automation systems** where a single sensor event must trigger multiple devices with different on-times.

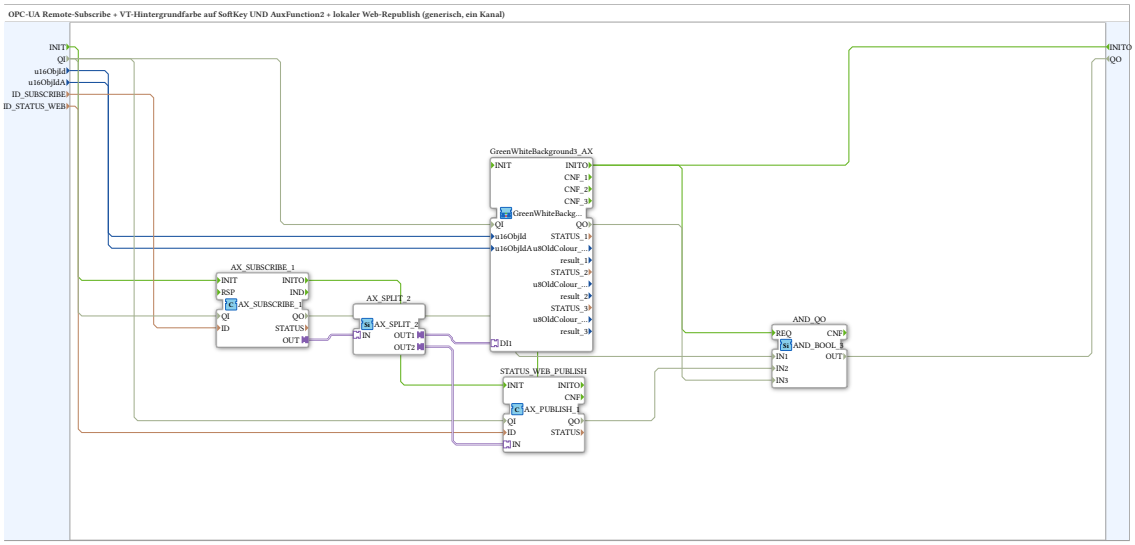
Comparison with Similar Blocks

Unlike a fixed pulse generator or a single retriggerable timer, this subapp provides dual-channel capability with independent timing settings. It can be seen as a combination of an event splitter and two pulse timers, offering flexibility not found in single-timer blocks. Compared to using two separate AX_TP instances manually, this subapp encapsulates the split and timer logic, reducing wiring complexity and improving readability.

Conclusion

AX_SPLIT_TO_2x_AX_TP is a compact, reusable component for scenarios requiring an AX trigger to be duplicated and delayed with independent pulse durations. Its clear interface, generic design, and straightforward configuration make it a valuable addition to any 4diac-based automation project.

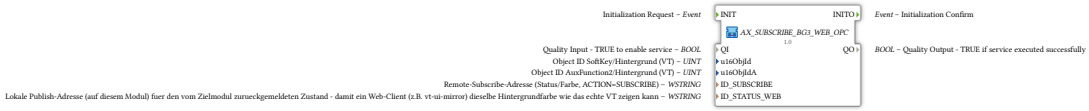
AX_SUBSCRIBE_BG3_WEB_OPC



• – ○ • – ○ • – ○ •

Introduction

`AX_SUBSCRIBE_BG3_WEB_OPC` is the status half of `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`, extracted as an independent component: a remote subscribe value is displayed as a background color on both a SoftKey AND an AuxFunction2 (`GreenWhiteBackground3_AX`) and is also republished locally via OPC UA so that a web client (e.g., vt-ui-mirror) on the same module can read the same status without needing its own connection to the target module.



Interface

- **Event input**: `INIT` (`EInit`, Initialization Request)
- **Event output**: `INITO` (`EInit`, Initialization Confirm)
- **Data inputs**: `QI` (`BOOL`, initial value `TRUE`, Quality Input), `u16ObjId`, `u16ObjIdA`, `ID_SUBSCRIBE`, `ID_STATUS_WEB`
- **Data output**: `QO` (`BOOL`, Quality Output)

Function Blocks (FBs) Used

Sub-Blocks: AX_SUBSCRIBE_BG3_WEB_OPC

- **Type**: SubAppType
- **Internal FBs Used**:
- **AX_SUBSCRIBE_1**: `adapter::net::AX_SUBSCRIBE_1` (`QI` from the block input `QI`) — Remote subscription of the status/color (`ID_SUBSCRIBE`).
- **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — Branches the received signal to the VT display and the local web republish.
- **GreenWhiteBackground3_AX** (SubApp, `MyLib::sys`): Synchronously colors both a SoftKey (`u16ObjId`) and an AuxFunction2 (`u16ObjIdA`); it has its own `INIT`/`INITO` and `QI`/`QO`.
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — ANDs the `QO` outputs of `AX_SUBSCRIBE_1`, `STATUS_WEB_PUBLISH` and `GreenWhiteBackground3_AX` into the block output `QO`.
- **STATUS_WEB_PUBLISH**: `adapter::net::AX_PUBLISH_1` (`QI` from the block input `QI`) — Republishes the received state to this module's local OPC UA server (`ID_STATUS_WEB`).
- **How it works**: The state received via remote subscribe is simultaneously passed to the VT background color and the local web republish server via `AX_SPLIT_2`.

Program Flow and Connections

1. u16ObjId → GreenWhiteBackground3_AX.u16ObjId; u16ObjIdA → GreenWhiteBackground3_AX.u16ObjIdA; ID_SUBSCRIBE → AX_SUBSCRIBE_1.ID; ID_STATUS_WEB → STATUS_WEB_PUBLISH.ID (Data connections, hidden).
2. AX_SUBSCRIBE_1.OUT → AX_SPLIT_2.IN → AX_SPLIT_2.OUT1 → GreenWhiteBackground3_AX.DI1 (VT display), AX_SPLIT_2.OUT2 → STATUS_WEB_PUBLISH.IN (Web Republish).
3. Initialization chain: INIT → AX_SUBSCRIBE_1.INIT, AX_SUBSCRIBE_1.INIT0 → STATUS_WEB_PUBLISH.INIT, STATUS_WEB_PUBLISH.INIT0 → GreenWhiteBackground3_AX.INIT, GreenWhiteBackground3_AX.INIT0 → INIT0; additionally GreenWhiteBackground3_AX.INIT0 → AND_Q0.REQ.
4. QI → AX_SUBSCRIBE_1.QI, STATUS_WEB_PUBLISH.QI and GreenWhiteBackground3_AX.QI.
5. AX_SUBSCRIBE_1.Q0 → AND_Q0.IN1, STATUS_WEB_PUBLISH.Q0 → AND_Q0.IN2, GreenWhiteBackground3_AX.Q0 → AND_Q0.IN3; AND_Q0.OUT → Q0.

Technical Features

- **Extension of AX_SUBSCRIBE_BG_OPC**: Compared to the simpler version (only Subscribe + background color), two things are added: GreenWhiteBackground3_AX instead of GreenWhiteBackground1_AX (colors SoftKey and Aux together) and local Web Republish via AX_SPLIT_2/STATUS_WEB_PUBLISH.
- **Decoupling from the target module**: A web client connected only to this module does not need to establish its own connection to the target module to see the same status as the actual VT.

Application Scenarios

- Status display for functions that need to be displayed synchronously on both a SoftKey and an AuxFunction2 (joystick assignment), plus web mirroring.
- For a SoftKey without an Aux, use AX_SUBSCRIBE_BG_OPC instead (no web republish).

Comparison with similar modules

For the complete, pre-wired combination with the command-line interface (SoftKey/Aux read + remote write), continue to use Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC, which includes this module as a status sub-module.

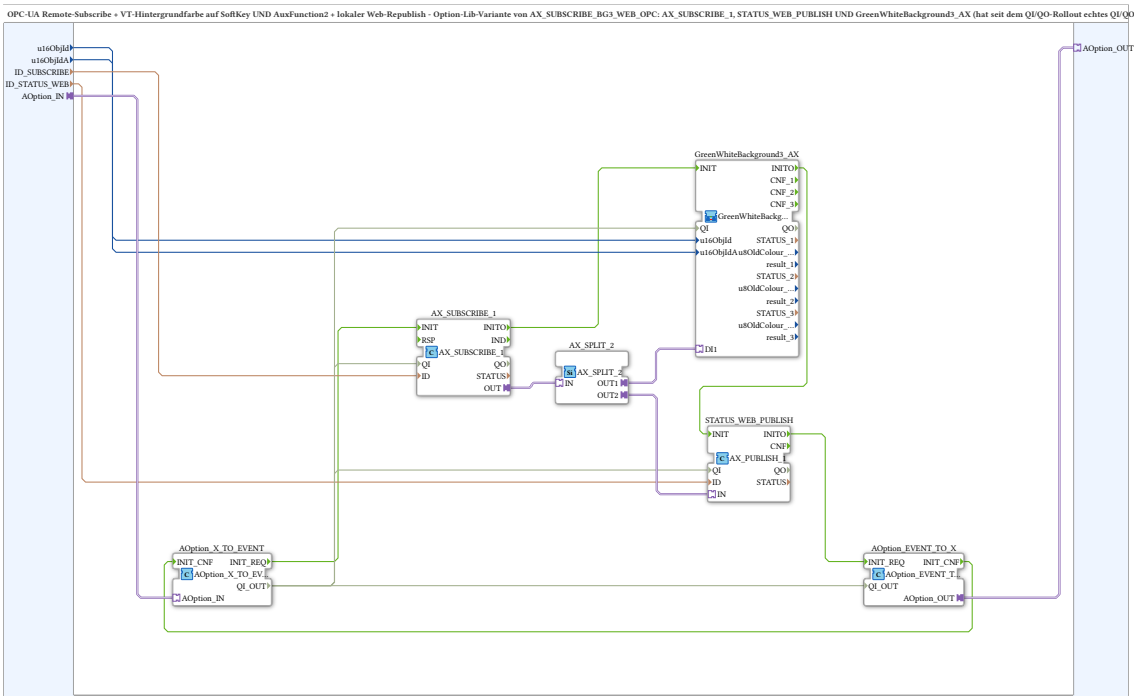
Summary

AX_SUBSCRIBE_BG3_WEB_OPC encapsulates the status page of a remote channel: reception, synchronous softkey/aux background color, and local web republish in a self-contained, reusable building block.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

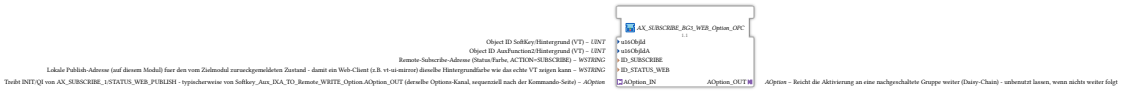
AX_SUBSCRIBE_BG3_WEB_Option_OPC



• — ○ • — ○ • — ○ •

Introduction

`AX_SUBSCRIBE_BG3_WEB_Option_OPC` is the option-lib variant of `AX_SUBSCRIBE_BG3_WEB_OPC`: OPC-UA remote subscribe with VT background colour on SoftKey AND AuxFunction2 plus local web republish. `AX_SUBSCRIBE_1`, `STATUS_WEB_PUBLISH` and `GreenWhiteBackground3_AX` (which has had a real `QI`/`QO` since the QI/QO rollout) are initialized and gated here by the adapter socket `Option::AOption`: all three sequentially in the order `AX_SUBSCRIBE_1` -> `GreenWhiteBackground3_AX` -> `STATUS_WEB_PUBLISH`. The plug `AOption_OUT` passes the activation on to a downstream group (daisy chain).



Interface Structure

Event Inputs

None (initialization runs via the adapter `AOption_IN`).

Event Outputs

None.

Data Inputs

Variable	Type	Default	Comment
`u16ObjId`	UINT	`ID_NULL`	Object ID SoftKey/background (VT).
`u16ObjIdA`	UINT	`ID_NULL`	Object ID AuxFunction2/background (VT).
`ID_SUBSCRIBE`	WSTRING		Remote subscribe address (status/colour, `ACTION=SUBSCRIBE`).
`ID_STATUS_WEB`	WSTRING		Local publish address for the state reported back by the target module, so that a web client (e.g. `vt-ui-mirror`) can show the same background colour as the real VT.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
AOption_IN	Option::AOption	Socket	Drives INIT/QI of AX_SUBSCRIBE_1, STATUS_WEB_PUBLISH and GreenWhiteBackground3_AX; typically fed by the AOption_OUT of Softkey_Aux_IXA_TO_Remote_WRITE_Option.
AOption_OUT	Option::AOption	Plug	Passes the activation on to a downstream group (daisy chain); leave unused if nothing follows.

Function Blocks (FBs) Used

Sub-blocks: AX_SUBSCRIBE_BG3_WEB_Option_OPC

- **Type**: SubAppType
- **Internal FBs used**:
  - **AOption_X_TO_EVENT** (Option::conversion::AOption_X_TO_EVENT): converts the socket AOption_IN into INIT_REQ/QI_OUT, see AOption_X_TO_EVENT.
  - **AX_SUBSCRIBE_1** (adapter::net::AX_SUBSCRIBE_1): OPC-UA subscribe on ID_SUBSCRIBE.
  - **AX_SPLIT_2** (adapter::events::unidirectional::AX_SPLIT_2): distributes the subscribe value to two branches.
  - **GreenWhiteBackground3_AX** (SubApp MyLib::sys::GreenWhiteBackground3_AX): sets the background colour on SoftKey and AuxFunction2; initialized and gated by the option socket sequentially, after AX_SUBSCRIBE_1 and before STATUS_WEB_PUBLISH.
  - **STATUS_WEB_PUBLISH** (adapter::net::AX_PUBLISH_1): local publish of the status on ID_STATUS_WEB.
  - **AOption_EVENT_TO_X** (Option::conversion::AOption_EVENT_TO_X): passes the activation on to AOption_OUT.

Program Flow and Connections

1. Option chain: AOption_X_TO_EVENT.INIT_REQ -> AX_SUBSCRIBE_1.INIT; AX_SUBSCRIBE_1.INITO -> GreenWhiteBackground3_AX.INIT; GreenWhiteBackground3_AX.INITO -> STATUS_WEB_PUBLISH.INIT; STATUS_WEB_PUBLISH.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF.
2. AOption_X_TO_EVENT.QI_OUT -> AX_SUBSCRIBE_1.QI, STATUS_WEB_PUBLISH.QI, GreenWhiteBackground3_AX.QI and AOption_EVENT_TO_X.QI_OUT.
3. Adapters: AOption_IN -> AOption_X_TO_EVENT.AOption_IN; AOption_EVENT_TO_X.AOption_OUT -> AOption_OUT.
4. Data: AX_SUBSCRIBE_1.OUT -> AX_SPLIT_2.IN; AX_SPLIT_2.OUT1 -> GreenWhiteBackground3_AX.DI1; AX_SPLIT_2.OUT2 -> STATUS_WEB_PUBLISH.IN.
5. Parameters: ID_SUBSCRIBE -> AX_SUBSCRIBE_1.ID; ID_STATUS_WEB -> STATUS_WEB_PUBLISH.ID; u16ObjId/u16ObjIdA -> GreenWhiteBackground3_AX.u16ObjId/u16ObjIdA.
6. The block has no event and no QI/QO interface; activation and acknowledgment run exclusively via the adapters AOption_IN/AOption_OUT.

Technical Features

- **All three blocks gated:** AX_SUBSCRIBE_1, STATUS_WEB_PUBLISH and GreenWhiteBackground3_AX hang on the option socket (version 1.1; GreenWhiteBackground3_AX has had a real QI/QO since the QI/QO rollout, the earlier assumption of no gating was outdated).
- **Sequential initialization:** The option chain initializes AX_SUBSCRIBE_1, GreenWhiteBackground3_AX and STATUS_WEB_PUBLISH one after the other; only after STATUS_WEB_PUBLISH.INITO does AOption_EVENT_TO_X.INIT_REQ fire.
- **Option initialization instead of INIT/QI:** The block has no INIT/INITO/QI/QO connections of its own; initialization runs via the INIT_REQ/INIT_CNF handshake of the option chain, and the acknowledgment returns via AOption_EVENT_TO_X.INIT_CNF to AOption_X_TO_EVENT.

- **Daisy chain:** `AOption_OUT` allows further instances or groups to be attached to the same option.
- The counterpart without option is `AX_SUBSCRIBE_BG3_WEB_OPC`.

Application Scenarios

- Status half of `Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC`: background colour on SoftKey and Aux from a remote subscribe, active only when the option is selected.

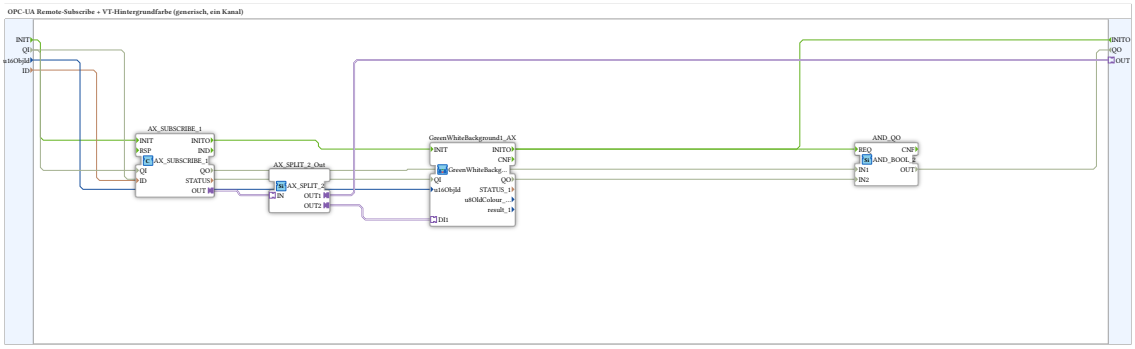
Summary

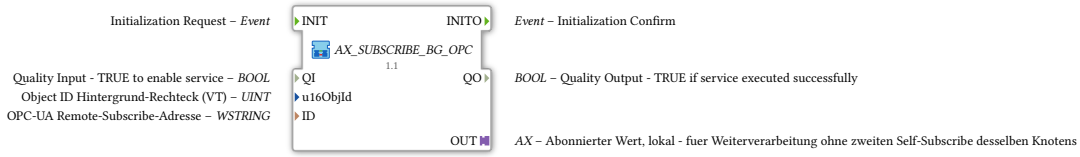
`AX_SUBSCRIBE_BG3_WEB_Option_OPC` combines remote subscribe, background colour on SoftKey and Aux and local web republish, and gates subscribe, publish and background colour via an `AOption` socket with forwarding through `AOption_OUT`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AX_SUBSCRIBE_BG_OPC





• — ○ • — ○ • — ○ •

Introduction

The **AX_SUBSCRIBE_BG_OPC** subapplication is a generic, single-channel solution that combines an OPC-UA remote subscription with a visualization (VT) background color change mechanism. It is designed for industrial HMI scenarios where a background rectangle of a VT screen needs to be dynamically recolored based on data received from an OPC-UA server. The subapp encapsulates the subscription logic and the background-color transformation into a single reusable component, hiding the internal complexity from the application engineer.

Interface Structure

The subapplication exposes the event input `INIT`, the event output `INITO`, the data inputs `QI`, `u16ObjId` and `ID`, and the data output `QO`. It has no adapter sockets at its external boundary. All internal communication is handled within the encapsulated network.

Event Inputs

- **INIT** (`EInit`): initialization request; starts the initialization chain `AX_SUBSCRIBE_1` → `GreenWhiteBackground1_AX`.

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `GreenWhiteBackground1_AX.INITO`.

Data Inputs

Name	Data Type	Description
QI	BOOL	Quality input, service enable (initial value TRUE).
u16ObjId	UINT	Object ID of the VT background rectangle. Default value is ID_NULL.
ID	WSTRING	OPC-UA remote-subscribe address (the server node or subscription identifier).

Data Outputs

Name	Data Type	Description
QO	BOOL	Quality output, AND of the QO outputs of AX_SUBSCRIBE_1 and GreenWhiteBackground1_AX.

The actual result is realized through the internal adapter connection to the background-color subapplication.

Adapters

No external adapters are exposed. Internally, the OPC-UA subscription function block AX_SUBSCRIBE_1 uses an adapter-type output (OUT) which is connected to the DI1 adapter socket of the embedded GreenWhiteBackground1_AX subapplication.

Functionality

The subapplication performs the following operations:

1. **Subscription Setup** – The internal function block AX_SUBSCRIBE_1 (type adapter::net::AX_SUBSCRIBE_1) is enabled through the subapplication input QI (wired to AX_SUBSCRIBE_1.QI; it is no longer a fixed QI=TRUE parameter) and is initialized via INIT → AX_SUBSCRIBE_1.INIT.
2. **Address Provision** – The external ID input (the OPC-UA subscribe address) is forwarded to the ID input of the AX_SUBSCRIBE_1 function block.
3. **Object Identification** – The external u16ObjId input is passed to the u16ObjId input of the embedded GreenWhiteBackground1_AX subapplication, identifying which VT background rectangle is to be affected.
4. **Color Change Propagation** – The adapter connection from AX_SUBSCRIBE_1.OUT to GreenWhiteBackground1_AX.DI1 transports the subscription result (e.g., a numeric value or status) to the background-color logic, which then alters the fill color of the specified VT rectangle accordingly.
5. **Initialization and Quality** – AX_SUBSCRIBE_1.INITO → GreenWhiteBackground1_AX.INIT; GreenWhiteBackground1_AX.INITO → INITO and → AND_QO.REQ. QI is also passed to GreenWhiteBackground1_AX.QI. AX_SUBSCRIBE_1.QO → AND_QO.IN1 and GreenWhiteBackground1_AX.QO → AND_QO.IN2; the instance AND_QO (iec61131::booleanOperators::AND_BOOL_2) drives QO.

The two data connections (ID → AX_SUBSCRIBE_1.ID and u16ObjId → GreenWhiteBackground1_AX.u16ObjId) are marked as invisible (Visible = false) in the diagram, reducing visual clutter in the engineering environment.

Technical Features

- **Generic Single-Channel Design** – The subapp is parameterized through its data inputs u16ObjId and ID (plus the enable QI), making it reusable for any OPC-UA node address and any VT rectangle object ID.
- **Embedded Subscription Logic** – The OPC-UA subscription FB is internally instantiated; its enable comes from QI and its initialization is sequenced by the subapp's own INIT/INITO chain.
- **Adapter-Based Coupling** – Communication between the subscription FB and the background-color FB is performed via an adapter connection (OUT to DI1), enabling type-safe links between the internal components.

- **Hidden Internal Wiring** – All internal connections are marked invisible to keep the subapp representation clean in the overall application diagram.
- **Default Object ID** – The `u16ObjId` input defaults to `ID_NULL`, allowing the subapp to be instantiated before a concrete object ID is assigned.

State Overview

Since `AX_SUBSCRIBE_BG_OPC` is a composite subapplication rather than a stateful function block, it does not define its own explicit state machine. However, the internal `AX_SUBSCRIBE_1` FB is expected to implement the standard OPC-UA subscription states:

- **Initialization** – With `QI = TRUE` and after `INIT`, the FB begins establishing the subscription to the given OPC-UA address.
- **Subscribed** – The subscription is active and data updates are propagated through the adapter output.
- **Error / Disconnected** – If the server is unreachable or the subscription fails, the FB signals an error condition (internally handled, not exposed externally).

The `GreenWhiteBackground1_AX` subapplication receives the subscription value via `DI1` and updates the background color based on the received data, typically switching between green (active/OK) and white (inactive/fault).

Application Scenarios

Typical use cases for this subapplication include:

- **Machine Status Indication** – In a machine HMI, a background rectangle is colored green when the OPC-UA node reports the machine as running, and white when it is stopped or in an error state.
- **Remote Variable Monitoring** – Monitoring a single OPC-UA variable (e.g., a production counter threshold) and visually reflecting its state on a VT screen.
- **Generic HMI Building Blocks** – Reuse in larger applications where multiple instances of the subapp are placed, each bound to a different OPC-UA address and a different VT rectangle, allowing rapid assembly of status displays.
- **SCADA-Style Dashboards** – Simple visual alarming where the background color of a label or area changes based on values read from a remote automation controller.

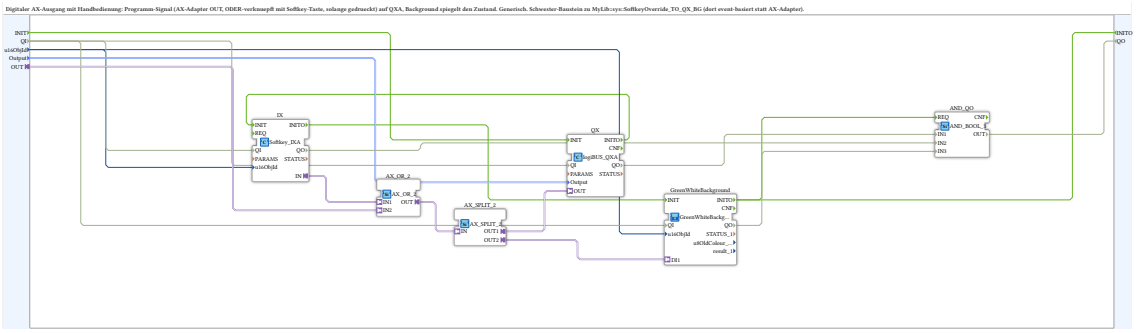
Comparison with Similar Blocks

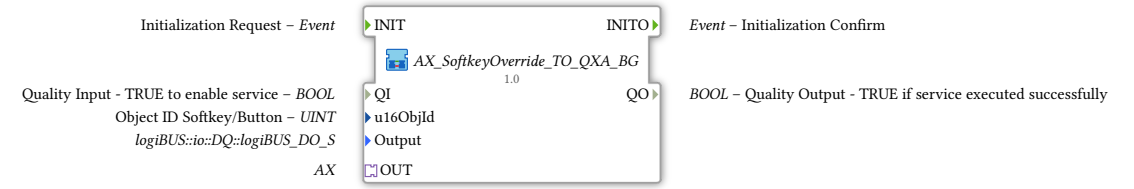
Feature	AX_SUBSCRIBE_BG_OPC	Direct FB Pairing (AX_SUBSCRIBE + GreenWhiteBackground)
Integration Effort	Low – only two inputs need configuring	Higher – requires manual wiring of adapter connections and parameters
Reusability	High – encapsulated as a self-contained unit	Medium – each instance must be reconnected individually
Interface Complexity	Small (`INIT`/`INITO`, `QI`/`QO`, 2 data inputs)	Multiple connections and event wiring
Error Handling	Handled internally by the subscribe FB	Depends on the immediate FB configuration
Visual Clarity	Clean – internal wiring hidden	Cluttered – all connections visible
Flexibility	Limited to one channel per instance	More flexible when individual FBs are used in complex networks

Conclusion

The **AX_SUBSCRIBE_BG_OPC** subapplication provides a compact, ready-to-use building block for OPC-UA-driven background color changes in visualization screens. By encapsulating the subscription logic and the color-change logic behind a simple interface (`INIT`/`INITO`, `QI`/`QO` and two data inputs), it significantly reduces engineering effort and improves the readability of application diagrams. Its generic design makes it suitable for a wide range of single-channel monitoring and status indication scenarios in industrial automation HMI environments.

AX_SoftkeyOverride_TO_QXA_BG





• – ○ • – ○ • – ○ •

Introduction

The **AX_SoftkeyOverride_TO_QXA_BG** is a generic subapplication that combines a program signal (via an AX adapter) with a manual softkey input to control a digital output (QXA). The effective output state is also mirrored to a background indicator for visual feedback. It is a sister block to `MyLib::sys::SoftkeyOverride_TO_QX_BG` but uses an adapter-based signal instead of an event-based trigger.

Interface Structure

Event Inputs

- `INIT` (EInit): initialization request; starts the init chain at `QX.INIT`.

Event Outputs

- `INITO` (EInit): initialization confirmation; comes from `GreenWhiteBackground.INITO`, the end of the init chain.

Data Inputs

- `QI` (BOOL) – Quality input, enables the service (initial value: `TRUE`); passed on to `QX.QI`, `IX.QI` and `GreenWhiteBackground.QI`.
- `u16ObjId` (UINT) – Object ID of the softkey used for manual override. Default value: `ID_NULL`.
- `Output` (`logiBUS::io::DQ::logiBUS_DO_S`) – Desired output value sent to the QXA block. Initial value: `logiBUS_DO::Invalid`.

Data Outputs

- `QO` (BOOL) – Quality output, AND combination (`AND_QO`, `AND_BOOL_3`) of the `QO` outputs of `QX`, `IX` and `GreenWhiteBackground`.

Adapters

- `OUT` (adapter::types::unidirectional::AX) – Receives the program signal that is logically OR-ed with the softkey state.

Functionality

The subapplication continuously processes the AX program signal from the `OUT` adapter and the softkey state from the internal `Softkey_IXA` block (identified by `u16ObjId`). These two signals are combined using an AX_OR_2 logic gate. The resulting boolean value is split via an AX_SPLIT_2 adapter:

- One branch (OUT1) is sent to the `QX` block (`logiBUS_QXA`) which drives the actual digital output.

- The second branch (OUT2) is connected to the `GreenWhiteBackground1_AX` subapplication to display the current output state.

Initialization and quality: `INIT` → `QX.INIT`, `QX.INITO` → `IX.INIT`, `IX.INITO` → `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` → `INITO` and `AND_QO.REQ`. `QI` is passed on to `QX`, `IX` and `GreenWhiteBackground`; `QX.QO`, `IX.QO` and `GreenWhiteBackground.QO` are combined in `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`) into the output `QO`.

The `Output` data input provides the value to be used by the QX block when the output is active.

Technical Features

- Integrates `Softkey_IXA` for reading the softkey state.
- Uses `AX_OR_2` to perform the logical OR operation between the program signal and the manual override.
- Employs `AX_SPLIT_2` to fan out the combined signal.
- Contains `GreenWhiteBackground1_AX` for visual state indication.
- Uses `logiBUS_QXA` as the digital output driver; its quality input `QI` (like those of `Softkey_IXA` and the background) comes from the block input `QI` instead of being fixed to `TRUE`.
- The init chain (`INIT` → `QX` → `IX` → `GreenWhiteBackground` → `INITO`) and the AND combination of the quality outputs (`QI`/`QO`) follow the standard interface pattern of the wrapper subapps.

State Overview

Program Signal (AX)	Softkey Pressed	Output (QXA)	Background
0	0	OFF	OFF
1	0	ON	ON
0	1	ON	ON
1	1	ON	ON

The output is ON whenever either the program signal or the manual softkey is active.

Application Scenarios

This subapplication is ideal for control systems requiring a manual override capability, e.g., a machine operator can force a digital output (like a valve, motor, or lamp) on or off using a softkey while automatic control is active. The background display helps operators quickly identify the current output state even in complex HMI environments.

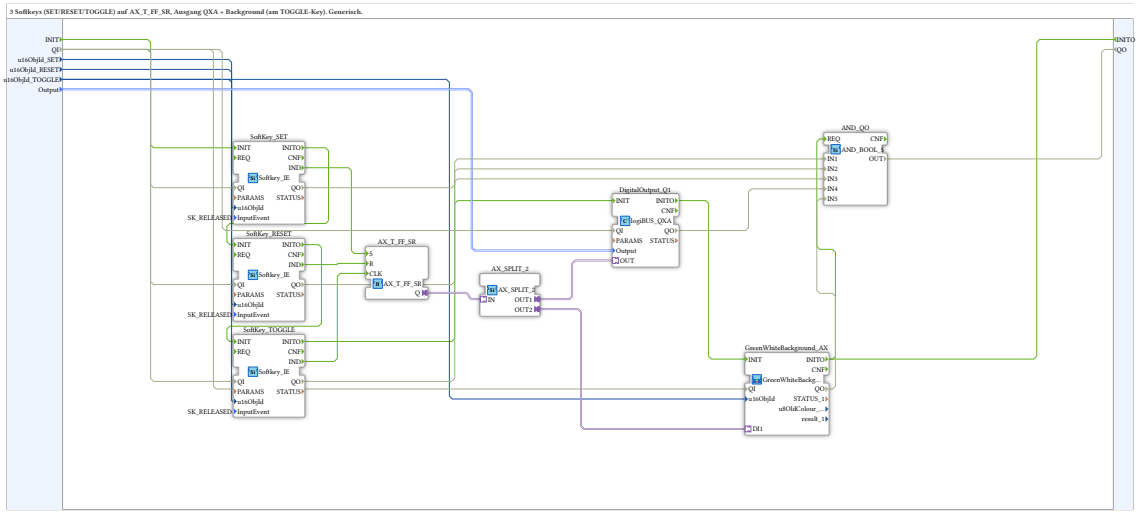
Comparison with Similar Blocks

- **SoftkeyOverride_TO_QX_BG**: This sister block achieves the same logical function but uses an event-based mechanism to trigger the override. It does not require an AX adapter and instead relies on event inputs and outputs.
- **AX_SoftkeyOverride_TO_QXA_BG**: The present block is adapter-centric, making it easier to integrate into systems where signals are communicated via unidirectional AX connections.

Conclusion

AX_SoftkeyOverride_TO_QXA_BG provides a clean, reusable solution for combining an automatic control signal with a manual softkey override to drive a digital output and its corresponding visual indicator. Its adapter-based interface simplifies integration into larger control networks and ensures consistent behavior across different applications.

AX_SoftkeySR_TO_QXA_BG





• — ○ • — ○ • — ○ •

Introduction

`AX_SoftkeySR_TO_QXA_BG` is a reusable 4diac subapplication that combines three softkeys with a SET/RESET/TOGGLE flip-flop function and a digital QXA output. It also integrates a green/white background indication that follows the same state, making it suitable for HMI or machine control applications where visual feedback is required.

The subapplication is generic: the softkey object IDs and the output selector are provided as configuration inputs. This allows the same subapplication to be reused in different ISOBUS or logiBUS-based control layouts without internal changes.

Interface Structure

The subapplication exposes the initialization events `INIT`/`INITO`, the quality ports `QI`/`QO` and the configuration data inputs. There are no adapter sockets/plugs at the subapplication boundary.

Event Inputs

- **INIT** (`EInit`): initialization request; starts the initialization chain of the inner blocks.

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `GreenWhiteBackground_AX.INITO`, the last link of the chain.

Data Inputs

Name	Type	Initial Value	Comment
`QI`	`BOOL`	`TRUE`	Quality input - TRUE enables the service; distributed to all inner blocks
`u16ObjId_SET`	`UINT`	`ID_NULL`	Object ID of the SET softkey
`u16ObjId_RESET`	`UINT`	`ID_NULL`	Object ID of the RESET softkey

u16ObjId_TOGGLE	UINT	ID_NULL	Object ID of the TOGGLE softkey, also used for the green/white background
Output	logiBUS::io::DQ::logiBUS_DQ_8	logiBUS_DO::Invalid	Identifies the output, Output_Q1..Q8

Data Outputs

- **QO** (BOOL): quality output - TRUE if the service was executed successfully (AND of the QO of all five inner instances).

Adapters

No adapter interface is exposed by the subapplication. All adapter connections are internal.

Functionality

The subapplication contains three Softkey_IE instances named SoftKey_SET, SoftKey_RESET, and SoftKey_TOGGLE. Each is configured with InputEvent = SK_RELEASED (its QI comes from the subapplication input QI), so a softkey action is triggered when the assigned softkey is released.

The internal event flow is:

- SoftKey_SET.IND is connected to AX_T_FF_SR.S
- SoftKey_RESET.IND is connected to AX_T_FF_SR.R
- SoftKey_TOGGLE.IND is connected to AX_T_FF_SR.CLK

This means:

- Pressing the configured SET softkey sets the flip-flop output Q to TRUE.
- Pressing the configured RESET softkey resets the flip-flop output Q to FALSE.
- Pressing the configured TOGGLE softkey toggles the flip-flop output Q.

The flip-flop output Q is then passed to AX_SPLIT_2, which duplicates the signal to two destinations:

1. DigitalOutput_Q1.OUT, a logiBUS_QXA digital output.
2. GreenWhiteBackground_AX.DI1, a subapplication of type MyLib::sys::GreenWhiteBackground1_AX.

The same u16ObjId_TOGGLE value is also connected to the background subapplication. Therefore, the TOGGLE softkey is used both to toggle the QXA output and to provide the associated visual background feedback.

Initialization and quality: INIT → SoftKey_SET.INIT, SoftKey_SET.INITO → SoftKey_RESET.INIT, SoftKey_RESET.INITO → SoftKey_TOGGLE.INIT, SoftKey_TOGGLE.INITO → DigitalOutput_Q1.INIT, DigitalOutput_Q1.INITO → GreenWhiteBackground_AX.INIT and GreenWhiteBackground_AX.INITO → INITO. QI is passed to SoftKey_SET, SoftKey_RESET, SoftKey_TOGGLE, DigitalOutput_Q1 and GreenWhiteBackground_AX. QO is the output of the instance AND_QO (iec61131::booleanOperators::AND_BOOL_5), which combines the QO of these five instances; its REQ is triggered by GreenWhiteBackground_AX.INITO.

The internal wiring is fully encapsulated. From the outside, the application only requires the three softkey object IDs and the desired QXA output selector.

Technical Features

- Encapsulates three softkey input FBs, one SET/RESET/TOGGLE flip-flop, one signal splitter, one QXA digital output, and one background feedback subapplication.
- Uses SK_RELEASED as the softkey activation behaviour.
- Generic object IDs are used with default value ID_NULL, so they can be assigned per integration.
- Generic output selector is used with default value logiBUS_DO::Invalid.
- QI is no longer hard-wired to TRUE inside; it is passed from the subapplication input QI (default TRUE) to all inner blocks, so the subapplication is ready to operate when integrated.
- Initialization runs sequentially through the chain from INIT to INITO; operation is otherwise controlled through softkey events internally mapped to the flip-flop logic.
- The same state is propagated to both the QXA digital output and the green/white background subapplication.

State Overview

The central state logic is performed by `AX_T_FF_SR`.

Condition	`Q` State	Effect
SET softkey released	`TRUE`	QXA output and background become active
RESET softkey released	`FALSE`	QXA output and background become inactive
TOGGLE softkey released	Toggles	QXA output and background change to the opposite state

All state changes occur on softkey release events, which prevents unintended multiple triggers during a single physical press.

Application Scenarios

- Machine HMI panels where three softkeys are assigned to SET, RESET, and TOGGLE functions.
- ISOBUS/logiBUS control tasks requiring a simple boolean QXA output.
- Applications where a green/white background indication should follow the state of the output.
- Reusable subapplication in larger control programs where multiple softkey-controlled outputs are needed.
- Generic integration into different machine layouts by configuring object IDs and output selectors externally.

Comparison with Similar Blocks

Compared to a manually wired network of `Softkey_IE` FBs and a flip-flop, this subapplication provides a clean and reusable interface. The internal event and adapter connections are already defined, which reduces integration effort.

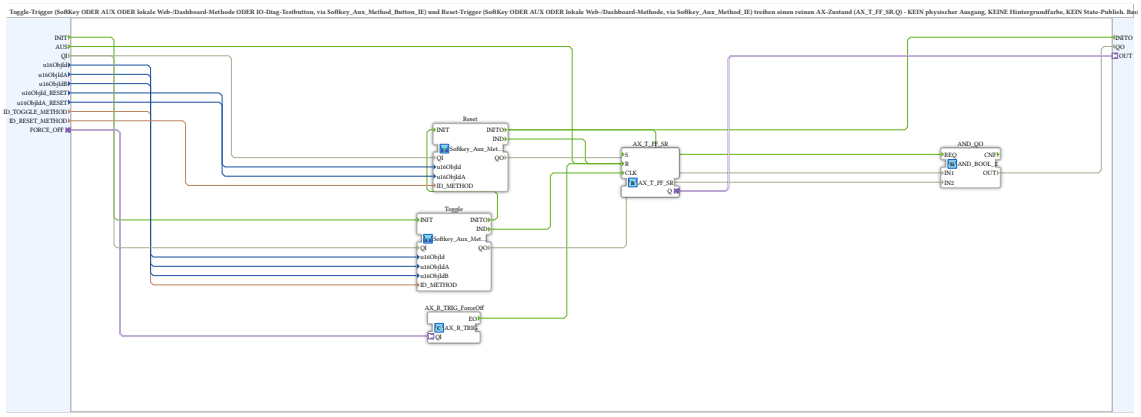
Compared to a simple SR flip-flop without a toggle function, `AX_SoftkeySR_TO_QXA_BG` adds a dedicated TOGGLE softkey input and therefore supports cyclic output switching from a single key.

Compared to fixed softkey implementations, this subapplication is generic. The object IDs and the QXA output can be configured, making it suitable for different HMI layouts without changing the internal logic.

Conclusion

`AX_SoftkeySR_TO_QXA_BG` is a compact, reusable subapplication for softkey-based SET/RESET/TOGGLE control of a QXA digital output. It combines the input handling, flip-flop state logic, output selection, and background feedback in a single encapsulated component. Its generic data interface makes it flexible for various machine control and HMI applications.

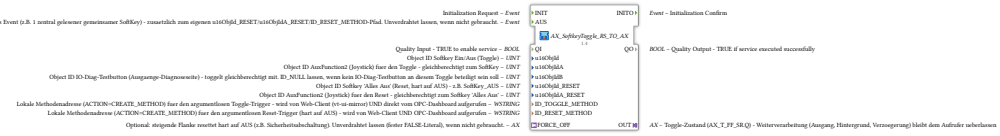
AX_SoftkeyToggle_RS_TO_AX



• – ○ • – ○ • – ○ •

Introduction

`AX_SoftkeyToggle_RS_TO_AX` generates a pure AX state from a toggle trigger and a reset trigger. The toggle trigger comes from SoftKey, AUX, a local web/dashboard method or an IO diagnostics test button (via `Softkey_Aux_Method_Button_IE`), the reset trigger from SoftKey, AUX or a local method (via `Softkey_Aux_Method_IE`). Both drive an `AX_T_FF_SR` whose state is present at the plug `OUT`. There is **no** physical output, **no** background colour and **no** state publish; these are left to the caller.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation request
- **AUS**: hard-resets the state to OFF from outside as an event (e.g. a centrally read common SoftKey 'All off'); in addition to the block's own reset path via `u16ObjId_RESET`/`u16ObjIdA_RESET`/`ID_RESET_METHOD`. Leave unwired if not needed.

Event Outputs

- **INITO** (`EInit`): initialisation confirmation

Data Inputs

- **QI** (BOOL): Quality Input, TRUE enables the service (default `TRUE`)
- **u16ObjId** (UINT): object ID of the SoftKey on/off (toggle), default `ID_NULL`
- **u16ObjIdA** (UINT): object ID of the AuxFunction2 (joystick) for the toggle, equal in rank to the SoftKey
- **u16ObjIdB** (UINT): object ID of the IO diagnostics test button (outputs diagnostics page), toggles with equal rank; leave `ID_NULL` if no such button is involved
- **u16ObjId_RESET** (UINT): object ID of the SoftKey 'All off' (reset, hard to OFF)
- **u16ObjIdA_RESET** (UINT): object ID of the AuxFunction2 (joystick) for the reset, equal in rank to the SoftKey 'All off'
- **ID_TOGGLE_METHOD** (WSTRING): local method address (`ACTION=CREATE_METHOD`) for the argument-less toggle trigger; called by the web client (vt-ui-mirror) and directly by the OPC dashboard
- **ID_RESET_METHOD** (WSTRING): local method address (`ACTION=CREATE_METHOD`) for the argument-less reset trigger (hard to OFF); called by the web client and the OPC dashboard

Data Outputs

- **QO** (BOOL): Quality Output, AND of `Toggle.QO` and `Reset.QO`

Adapters

- **OUT** (plug, `adapter::types::unidirectional::AX`): toggle state (`AX_T_FF_SR.Q`); further processing (output, background, delay) is left to the caller
- **FORCE_OFF** (socket, `adapter::types::unidirectional::AX`): optional; a rising edge hard-resets to OFF (e.g. safety shutdown). Leave unwired or at a fixed FALSE if not needed

Functionality

The network contains five parts:

- `Toggle` (`MyLib::sys::Softkey_Aux_Method_Button_IE`): reads `u16ObjId`, `u16ObjIdA`, `u16ObjIdB` and `ID_TOGGLE_METHOD` (to `Toggle.ID_METHOD`); its `IND` goes to `AX_T_FF_SR.CLK` and toggles the state.
- `Reset` (`MyLib::sys::Softkey_Aux_Method_IE`): reads `u16ObjId_RESET` (to `Reset.u16ObjId`), `u16ObjIdA_RESET` (to `Reset.u16ObjIdA`) and `ID_RESET_METHOD`; its `IND` goes to `AX_T_FF_SR.R`.
- `AX_T_FF_SR` (`adapter::events::unidirectional::AX_T_FF_SR`): toggle flip-flop with set/reset; `AX_T_FF_SR.Q` ⟶ `OUT`.
- `AX_R_TRIG_ForceOff` (`adapter::events::unidirectional::AX_R_TRIG`): detects the rising edge on `FORCE_OFF` (`FORCE_OFF` ⟶ `AX_R_TRIG_ForceOff.QI`); its `EO` also goes to `AX_T_FF_SR.R`.
- `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): combines `Toggle.QO` (`IN1`) and `Reset.QO` (`IN2`) into the `QO` of the SubApp.

The event `AUS` is a third, additive path to `AX_T_FF_SR.R`.

Initialisation and quality: `INIT` ⟶ `Toggle.INIT`; `Toggle.INITO` ⟶ `Reset.INIT`; `Reset.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`). `QI` is distributed to `Toggle.QI` and `Reset.QI`.

Technical Details

- Base block: for **one** output there is `MyLib::sys::AX_SoftkeyToggle_RS_TO_QXA_BG`, which wraps this block. For several time-staggered outputs (e.g. belt sequencing) this block is used directly and the `OUT` state is processed per channel with `AX_TON`/`AX_TOF`
- Three equal reset paths (VT/AUX/method, event `AUS`, edge on `FORCE_OFF`) all end at `AX_T_FF_SR.R`
- The reset input is called `AUS`, matching the central 'All off' softkey that several toggle instances distribute together

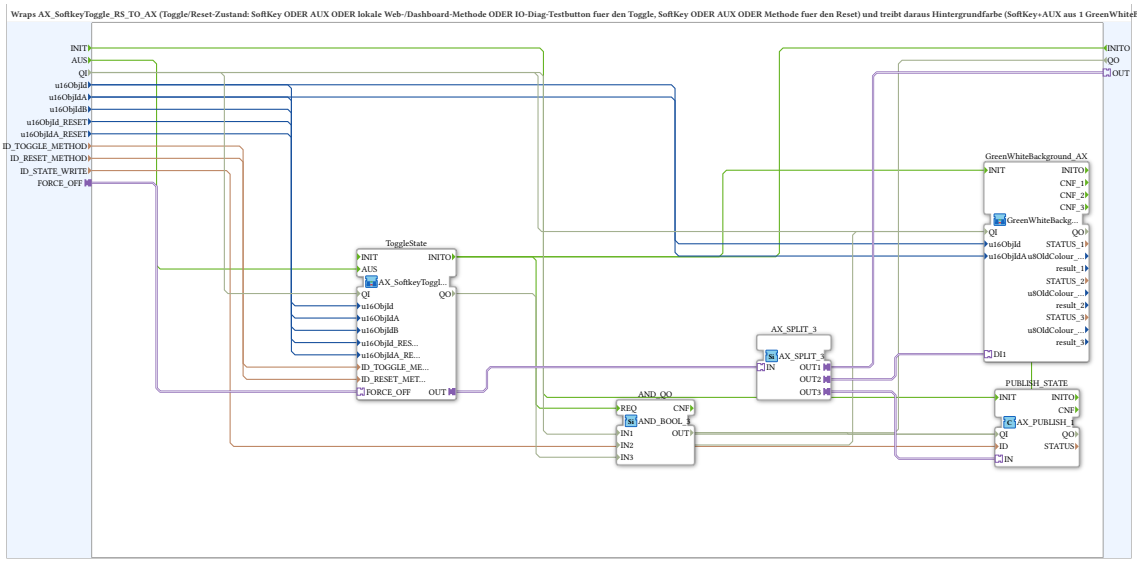
Application Scenarios

- On/off switch for drives operated equally by SoftKey, joystick assignment, web/dashboard and diagnostics button
- Central 'All off' for several toggle instances via the event `AUS`
- Safety shutdown via `FORCE_OFF` (e.g. PTO monitoring)

Conclusion

`AX_SoftkeyToggle_RS_TO_AX` supplies the pure toggle state with several equal operating and reset paths and leaves output, background colour and publish to the caller.

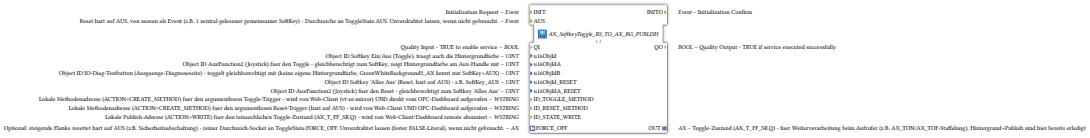
AX_SoftkeyToggle_RS_TO_AX_BG_PUBLISH



• — ○ • — ○ • — ○ •

Introduction

`AX_SoftkeyToggle_RS_TO_AX_BG_PUBLISH` wraps `AX_SoftkeyToggle_RS_TO_AX` (toggle/reset state) and drives the background colour and a local state publish from it, **without** a physical output. The toggle is triggered by SoftKey, AUX, local web/dashboard method or IO-diag test button; the reset by SoftKey, AUX or method. The background colour (SoftKey and AUX) comes from one shared `GreenWhiteBackground3_AX`. The block is the counterpart of `AX_SoftkeyToggle_RS_TO_QXA_BG`, which additionally drives a `logiBUS_QXA`. It is meant for cases in which the caller processes the toggle state further before it goes to one or more physical outputs (e.g. conveyor sequencing: key 5 toggles, only then `AX_TON`/`AX_TOF` per channel). The toggle state remains available as plug `OUT`.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request (chain: `PUBLISH_STATE` ⟶ `GreenWhiteBackground_AX` ⟶ `ToggleState`)
- **AUS**: hard reset to OFF, from outside as an event (e.g. one centrally read shared SoftKey). Pass-through to `ToggleState.AUS`. Leave unwired if not needed.

Event Outputs

- **INITO** (`EInit`): initialization confirm (from `ToggleState.INITO`)

Data Inputs

- **QI** (`BOOL`): quality input, TRUE enables the service (default `TRUE`)
- **u16ObjId** (`UINT`): object ID of the on/off softkey (toggle), also carries the background colour (default `ID_NULL`)
- **u16ObjIdA** (`UINT`): object ID of the AuxFunction2 (joystick) for the toggle, equal in rank to the SoftKey, also shows the background colour on the aux handle (default `ID_NULL`)
- **u16ObjIdB** (`UINT`): object ID of the IO-diag test button (outputs diagnostic page), toggles with equal rank; no background colour of its own because `GreenWhiteBackground3_AX` only knows SoftKey and AUX (default `ID_NULL`)
- **u16ObjId_RESET** (`UINT`): object ID of the "all off" softkey (reset, hard to OFF), e.g. `SoftKey_AUS` (default `ID_NULL`)
- **u16ObjIdA_RESET** (`UINT`): object ID of the AuxFunction2 (joystick) for the reset, equal in rank to the "all off" softkey (default `ID_NULL`)
- **ID_TOGGLE_METHOD** (`WSTRING`): local method address (`ACTION=CREATE_METHOD`) for the argument-less toggle trigger, called by the web client (vt-ui-mirror) and directly by the OPC dashboard
- **ID_RESET_METHOD** (`WSTRING`): local method address (`ACTION=CREATE_METHOD`) for the argument-less reset trigger (hard to OFF), called by the web client and the OPC dashboard
- **ID_STATE_WRITE** (`WSTRING`): local publish address (`ACTION=WRITE`) for the actual toggle state (`AX_T_FF_SR.Q`), subscribed remotely by web client/dashboard

Data Outputs

- **QO** (`BOOL`): quality output, AND combination (`AND_QO`) of the `QO` outputs of `PUBLISH_STATE`, `GreenWhiteBackground_AX` and `ToggleState`

Adapters

- **FORCE_OFF** (socket, `adapter::types::unidirectional::AX`): optional; a rising edge hard-resets to OFF (e.g. safety shutdown). Pure pass-through socket to `ToggleState.FORCE_OFF`. Leave unwired (fixed FALSE literal) if not needed.
- **OUT** (plug, `adapter::types::unidirectional::AX`): toggle state (`AX_T_FF_SR.Q`) for further processing by the caller; background and publish are already handled here

Function Blocks (FBs) Used

- **`ToggleState`** (SubApp `MyLib::sys::AX_SoftkeyToggle_RS_TO_AX`): generates the toggle/reset state
- **`AX_SPLIT_3`** (`adapter::events::unidirectional::AX_SPLIT_3`): distributes the state to three targets
- **`GreenWhiteBackground_AX`** (SubApp `MyLib::sys::GreenWhiteBackground3_AX`): green/white background colour
- **`PUBLISH_STATE`** (`adapter::net::AX_PUBLISH_1`): publishes the state (`QI` comes from the wrapper input `QI`)
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_3`): combines the `QO` outputs of the three inner blocks into `QO`

Functionality

- `AUS` ⟶ `ToggleState.AUS`; `FORCE_OFF` ⟶ `ToggleState.FORCE_OFF`
- `u16ObjId`, `u16ObjIdA`, `u16ObjIdB`, `u16ObjId_RESET`, `u16ObjIdA_RESET`, `ID_TOGGLE_METHOD`, `ID_RESET_METHOD` ⟶ the like-named inputs of `ToggleState`
- `ToggleState.OUT` ⟶ `AX_SPLIT_3.IN`
- `AX_SPLIT_3.OUT1` ⟶ `OUT`

- AX_SPLIT_3.OUT2 ⟶ GreenWhiteBackground_AX.DI1; u16ObjId, u16ObjIdA ⟶ GreenWhiteBackground_AX
- AX_SPLIT_3.OUT3 ⟶ PUBLISH_STATE.IN; ID_STATE_WRITE ⟶ PUBLISH_STATE.ID
- Initialization: INIT ⟶ PUBLISH_STATE.INIT; PUBLISH_STATE.INITO ⟶ GreenWhiteBackground_AX.INIT; GreenWhiteBackground_AX.INITO ⟶ ToggleState.INIT; ToggleState.INITO ⟶ INITO and ⟶ AND_QO.REQ
- Quality: QI ⟶ PUBLISH_STATE.QI, GreenWhiteBackground_AX.QI, ToggleState.QI; PUBLISH_STATE.QO ⟶ AND_QO.IN1, GreenWhiteBackground_AX.QO ⟶ AND_QO.IN2, ToggleState.QO ⟶ AND_QO.IN3; AND_QO.OUT ⟶ QO

Technical Details

- Extracted from Baender_EinAus (ToggleState/AX_SPLIT_3 part for background and publish, GreenWhiteBackground_AX, PUBLISH_STATE). The time-staggered outputs (TOF_IgelElevator/TON_Querband staggering) stay with the caller.
- Version history: FORCE_OFF (v1.1, for a higher-level safety shutdown), RESET pass-through (v1.2), input RESET renamed to AUS (v1.3).
- Imports isobus::UT::Q::const::IDs::ID_NULL as default for the object IDs.

Application Scenarios

- On/off toggle with colour indication and web publish whose state is processed further before the physical output (e.g. staggered conveyors)

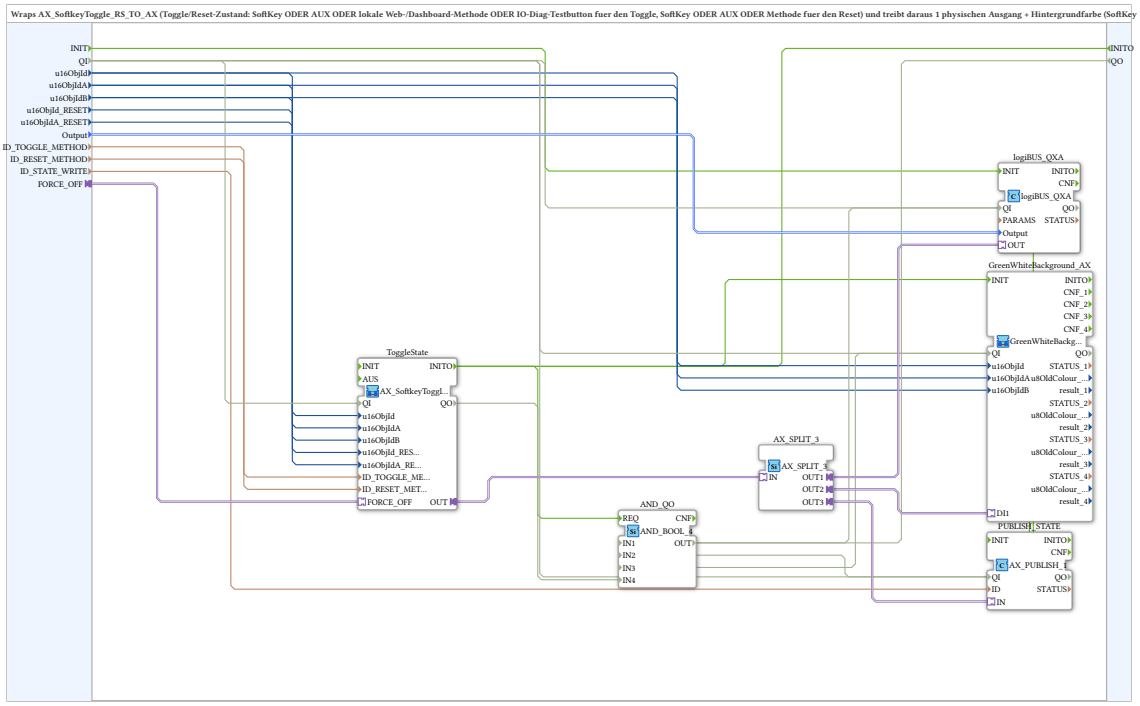
Summary

AX_SoftkeyToggle_RS_TO_AX_BG_PUBLISH provides a toggle/reset state with background colour and state publish and outputs the state as an AX plug for further processing.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AX_SoftkeyToggle_RS_TO_QXA_BG



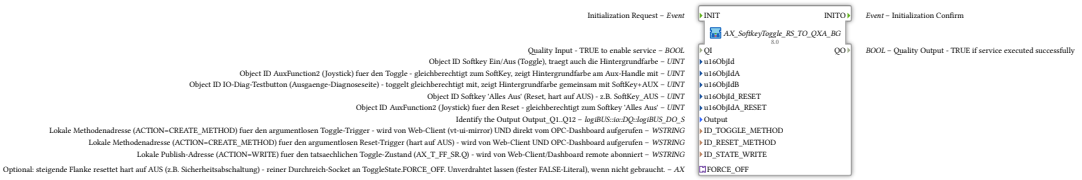
• – ○ • – ○ • – ○ •

Introduction

`AX_SoftkeyToggle_RS_TO_QXA_BG` wraps `AX_SoftkeyToggle_RS_TO_AX` (toggle/reset state) and uses it to drive **one physical output**, the **background colour** and a **local state publish**:

- Toggle: SoftKey **or** AUX **or** local web/dashboard method **or** IO-diag test button
- Reset: SoftKey **or** AUX **or** method
- Background colour: SoftKey, AUX handle and IO-diag test button together from one `GreenWhiteBackground4_AX`
- Publish: state for the web client/OPC dashboard background colour

Generic for any on/off output that must also be reset by a global “all off” key (e.g. saw blade drive).



Function Blocks (FBs) Used

Sub-blocks: AX_SoftkeyToggle_RS_TO_QXA_BG

- **Type**: SubAppType
- **Internal blocks used**:
  - **ToggleState**: SubApp `MyLib::sys::AX_SoftkeyToggle_RS_TO_AX` - toggle/reset state (incl. `FORCE_OFF`)
  - **AX_SPLIT_3**: `adapter::events::unidirectional::AX_SPLIT_3` - distributes the state to three consumers
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` (`QI` from the block input `QI`) - physical output
  - **GreenWhiteBackground_AX**: SubApp `MyLib::sys::GreenWhiteBackground4_AX` - background colour for SoftKey, AUX and test button
  - **PUBLISH_STATE**: `adapter::net::AX_PUBLISH_1` (`QI` from the block input `QI`) - local state publish
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_4` - ANDs the `QO` outputs of `logiBUS_QXA`, `PUBLISH_STATE`, `GreenWhiteBackground_AX` and `ToggleState` into the block output `QO`
- **Imports**: `ID_NULL`, `logiBUS_DO`, `logiBUS_DO_S`

Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization Request

Event Outputs

- **INITO** (`EInit`): Initialization Confirm

The signal processing (toggle/reset) still travels via the adapters.

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE to enable the service
- **u16ObjId** (`UINT`, default `ID_NULL`): Object ID of the on/off softkey (toggle), also carries the background colour
- **u16ObjIdA** (`UINT`, default `ID_NULL`): Object ID of the `AuxFunction2` (joystick) for the toggle - equal to the SoftKey, also shows the background colour on the aux handle
- **u16ObjIdB** (`UINT`, default `ID_NULL`): Object ID of the IO-diag test button (outputs diagnostic page) - toggles equally, shows the background colour together with SoftKey and AUX
- **u16ObjId_RESET** (`UINT`, default `ID_NULL`): Object ID of the “all off” softkey (reset, hard to OFF), e.g. `SoftKey_AUS`
- **u16ObjIdA_RESET** (`UINT`, default `ID_NULL`): Object ID of the `AuxFunction2` (joystick) for the reset - equal to the “all off” softkey
- **Output** (`logiBUS::io::DQ::logiBUS_DO_S`, default `logiBUS_DO::Invalid`): Output `Output_Q1..Q12`
- **ID_TOGGLE_METHOD** (`WSTRING`): Local method address (`ACTION=CREATE_METHOD`) for the argument-less toggle trigger - called by the web client (`vt-ui-mirror`) and directly by the OPC dashboard

- **ID_RESET_METHOD** (WSTRING): Local method address (ACTION=CREATE_METHOD) for the argument-less reset trigger (hard to OFF) - called by the web client and OPC dashboard
- **ID_STATE_WRITE** (WSTRING): Local publish address (ACTION=WRITE) for the actual toggle state (AX_T_FF_SR.Q) - subscribed remotely by the web client/dashboard

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if the service executed successfully (AND of the QO of the inner blocks)

Adapters

- **FORCE_OFF** (socket, adapter::types::unidirectional::AX): Optional: a rising edge hard-resets to OFF (e.g. safety shutdown) - pure passthrough socket to ToggleState.FORCE_OFF. Leave unwired (fixed FALSE literal) if not needed.

Program Flow and Connections

- u16ObjId, u16ObjIdA, u16ObjIdB, u16ObjId_RESET, u16ObjIdA_RESET, ID_TOGGLE_METHOD, ID_RESET_METHOD → like-named inputs of ToggleState
- FORCE_OFF → ToggleState.FORCE_OFF
- ToggleState.OUT → AX_SPLIT_3.IN
- AX_SPLIT_3.OUT1 → logiBUS_QXA.OUT (Output → logiBUS_QXA.Output)
- AX_SPLIT_3.OUT2 → GreenWhiteBackground_AX.DI1 (u16ObjId, u16ObjIdA, u16ObjIdB → like-named inputs)
- AX_SPLIT_3.OUT3 → PUBLISH_STATE.IN (ID_STATE_WRITE → PUBLISH_STATE.ID)
- Initialization chain: INIT → logiBUS_QXA.INIT, logiBUS_QXA.INITO → PUBLISH_STATE.INIT, PUBLISH_STATE.INITO → GreenWhiteBackground_AX.INIT, GreenWhiteBackground_AX.INITO → ToggleState.INIT, ToggleState.INITO → INITO; additionally ToggleState.INITO → AND_QO.REQ
- QI → logiBUS_QXA.QI, PUBLISH_STATE.QI, GreenWhiteBackground_AX.QI and ToggleState.QI
- logiBUS_QXA.QO → AND_QO.IN1, PUBLISH_STATE.QO → AND_QO.IN2, GreenWhiteBackground_AX.QO → AND_QO.IN3, ToggleState.QO → AND_QO.IN4; AND_QO.OUT → QO

Technical Details

- Since version 7.0 only a wrapper: toggle, reset and AX_T_FF_SR live in AX_SoftkeyToggle_RS_TO_AX, which is also used for multi-channel sequencing (several time-shifted outputs)
- Since version 6.0: one GreenWhiteBackground4_AX instead of two GreenWhiteBackground1_AX, so the joystick handle also receives feedback
- Version 8.0 (2026-09-15): FORCE_OFF socket added (purely additive; existing instances must wire it, e.g. to a fixed FALSE literal)
- History: 1.0 reset-set-toggle variant of AX_SoftkeyToggle_TO_QXA_BG; 2.0 IO-diag test button, two local RPC methods and state publish; 3.0 AUX added; 4.0/5.0 factored out into Softkey_Aux_Method_IE and Softkey_Aux_Method_Button_IE

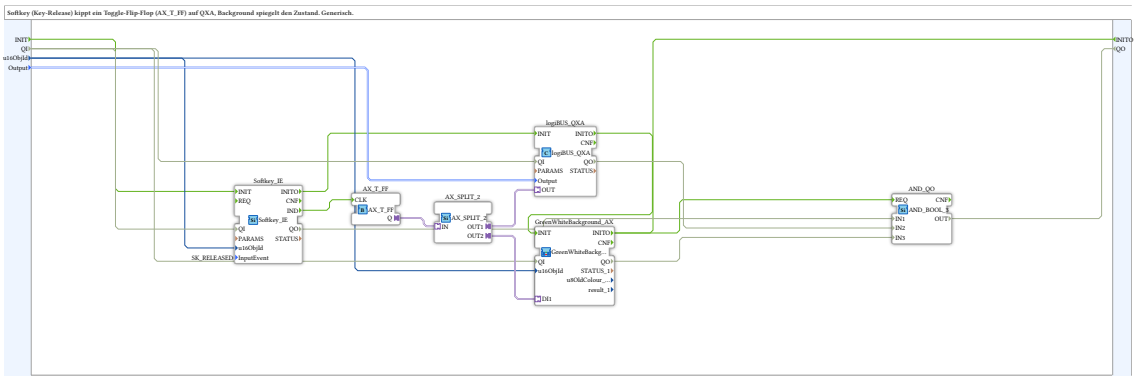
Application Scenarios

- On/off output operated via SoftKey, joystick, IO-diag test button and web/dashboard plus a global “all off” key (e.g. saw blade drive)

Summary

AX_SoftkeyToggle_RS_TO_QXA_BG connects the toggle/reset state from AX_SoftkeyToggle_RS_TO_AX with a physical output, background colour and local state publish.

AX_SoftkeyToggle_TO_QXA_BG





• – ○ • – ○ • – ○ •

Introduction

`AX_SoftkeyToggle_TO_QXA_BG` is a composite 4diac subapplication that implements a softkey release toggle for a QXA output and synchronizes an HMI background with the toggled state. It is generic: the softkey object ID and the QXA output channel are supplied as data inputs.

The internal network combines:

- `Softkey_IE` for detecting a configured softkey release event,
- `AX_T_FF` as a toggle flip-flop,
- `AX_SPLIT_2` to distribute the toggle state,
- `logiBUS_QXA` to drive the selected output,
- `GreenWhiteBackground_AX` to visualize the state in the background.

Interface Structure

The subapplication exposes the event input `INIT`, the event output `INITO`, the data inputs `QI`, `u16ObjId` and `Output`, and the data output `QO`. It has no public adapter endpoints.

Event Inputs

- **INIT** (`EInit`): initialization request; starts the chain `Softkey_IE` → `logiBUS_QXA` → `GreenWhiteBackground_AX`.

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `GreenWhiteBackground_AX.INITO`.

Data Inputs

Name	Type	Initial Value	Comment
`QI`	`BOOL`	`TRUE`	Quality input, service enable; passed on to `Softkey_IE`, `logiBUS_QXA` and `GreenWhiteBackground_AX`.
`u16ObjId`	`UINT`	`ID_NULL`	Object ID of the softkey to be used.
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	`logiBUS_DO::Invalid`	Identifies the QXA output channel (`Output_Q1` … `Output_Q8`).

Data Outputs

Name	Type	Comment
QO	BOOL	Quality output, AND of the QO outputs of Softkey_IE, logiBUS_QXA and GreenWhiteBackground_AX (via AND_QO).

Adapters

The subapplication does not expose any adapter sockets or plugs. Internally, unidirectional adapters are used for the toggle state flow:

- AX_T_FF.Q ⟶ AX_SPLIT_2.IN
- AX_SPLIT_2.OUT1 ⟶ logiBUS_QXA.OUT
- AX_SPLIT_2.OUT2 ⟶ GreenWhiteBackground_AX.DI1

Functionality

The subapplication is triggered by a softkey release. Softkey_IE is configured with InputEvent = SK_RELEASED (its QI comes from the subapplication input QI); it monitors the object ID given by u16ObjId. When the configured softkey is released, Softkey_IE emits the IND event. This event clocks the AX_T_FF toggle flip-flop. Each clock pulse changes the state of the toggle. The current toggle state is output on the adapter Q.

AX_SPLIT_2 forwards the same toggle state to two consumers:

1. logiBUS_QXA.OUT, which writes the state to the selected QXA output channel (Output).
2. GreenWhiteBackground_AX.DI1, which updates the background visualization according to the state.

Thus, a single key release toggles the actual output and keeps the HMI background in sync.

Initialization and quality: INIT ⟶ Softkey_IE.INIT, Softkey_IE.INITO ⟶ logiBUS_QXA.INIT, logiBUS_QXA.INITO ⟶ GreenWhiteBackground_AX.INIT, GreenWhiteBackground_AX.INITO ⟶ INITO and ⟶ AND_QO.REQ. QI goes to Softkey_IE.QI, logiBUS_QXA.QI and GreenWhiteBackground_AX.QI; their QO outputs are combined via AND_QO (iec61131::booleanOperators::AND_BOOL_3) into QO.

Technical Features

- Composite SubAppType reusable as a library element.
- Generic softkey identification via u16ObjId.
- Generic QXA output selection via Output.
- Internal event handling: the only external events are INIT/INITO; the clock events are handled internally.
- QI is no longer hard-coded to TRUE on Softkey_IE; it is passed through from the subapplication input QI (initial value TRUE).
- Key-release detection using SK_RELEASED, so the toggle reacts on release, not on press.
- AX_T_FF provides the bistable toggle memory.
- AX_SPLIT_2 provides a clean one-to-two adapter state distribution.
- logiBUS_QXA is enabled by the passed-through QI and drives the selected output.
- GreenWhiteBackground_AX receives the same state to show the HMI background.
- Internal data connections are marked invisible in the 4diac diagram for readability.

State Overview

The main state is the internal toggle state of AX_T_FF.

Condition	Event Path	Result
No softkey release	—	Toggle state remains unchanged.
Configured softkey released	Softkey_IE.IND ⟶ AX_T_FF.CLK	Toggle state changes from 0 to 1 or from 1 to 0.
Toggle state updated	AX_T_FF.Q ⟶ AX_SPLIT_2.IN	Both QXA output and background receive the new state.

Apart from INIT, the subapplication does not define an explicit reset input or initial state in the XML; the startup behavior depends on the default/reset behavior of AX_T_FF.

Application Scenarios

- Machine control panels where a softkey toggles a latching output.
- Operator buttons on HMI screens that must show the current state as a colored background.
- Generic output toggling for QXA modules, with flexible configuration of softkey and output channel.
- Reusable library subapplication for projects using `isobus` softkey handling and `logiBUS` digital outputs.

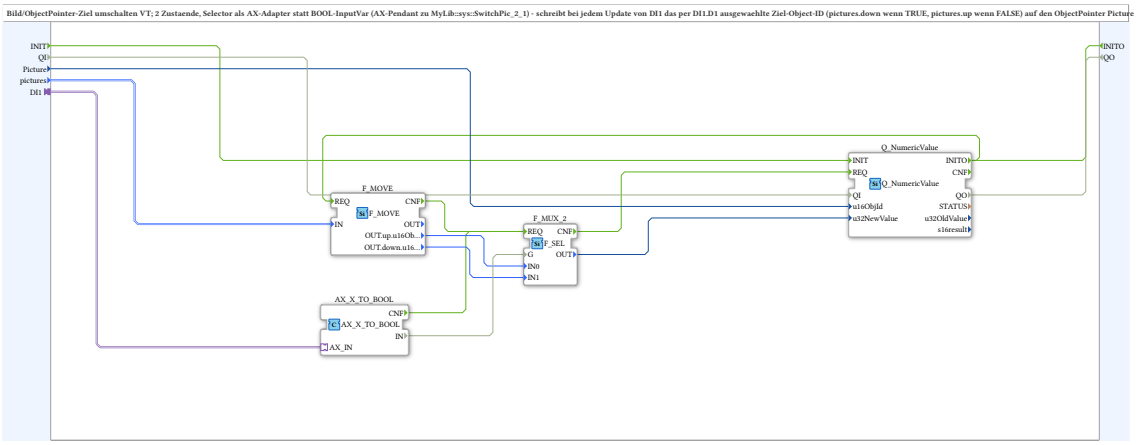
Comparison with Similar Blocks

- **Compared with a direct `Softkey_IE` → `logiBUS_QXA` connection:** The direct connection only follows the momentary softkey state. `AX_SoftkeyToggle_TO_QXA_BG` adds a toggle flip-flop, so the output remains latched after each release.
- **Compared with the same toggle without background:** This subapplication additionally drives `GreenWhiteBackground_AX`, so visualization and output always show the same state.
- **Compared with a fixed-softkey implementation:** The use of `u16ObjId` and `Output` makes the subapplication generic; no modification is needed when a different softkey or QXA channel is used.

Conclusion

`AX_SoftkeyToggle_TO_QXA_BG` is a compact and reusable 4diac subapplication that combines softkey release detection, a toggle flip-flop, QXA output control, and background visualization in one generic component. It reduces wiring effort and improves consistency between the physical/logical output state and the HMI background display.

AX_SwitchPic_2_1



• – ○ • – ○ • – ○ •

Introduction

`AX_SwitchPic_2_1` switches the target of a VT **ObjectPointer** (picture) between two states. The selector arrives as an **AX adapter** instead of a `BOOL` input variable (AX counterpart of `MyLib::sys::SwitchPic_2_1`): on every update of `DI1`, the target object ID selected via `DI1.D1` (`pictures.down` if `TRUE`, `pictures.up` if `FALSE`) is written to the ObjectPointer `Picture`. No external trigger is needed, since `DI1`'s own `E1/D1` feeds it directly. Initialization runs via `INIT/INITO` of the inner `Q_NumericValue` (quality pattern with `QI/QO`).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request - to `Q_NumericValue.INIT`; further events arrive via the adapter socket

Event Outputs

- **INITO** (`EInit`): initialization confirmation - from `Q_NumericValue.INITO`

Data Inputs

- **QI** (BOOL, default TRUE): Quality Input - TRUE to enable the service (to Q_NumericValue.QI)
- **Picture** (UINT): Object ID of the ObjectPointer to write into
- **pictures** (isobus::UT::Q::types::s2x1ObjectIDs): The two target object IDs (up, down); default: both ID_NULL

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if the service was executed successfully (from Q_NumericValue.QO)

Adapters

- **DI1** (socket): Unidirectional adapter of type adapter::types::unidirectional::AX - selector: down (pictures.down) if TRUE, up (pictures.up) if FALSE. Every update writes immediately.

Function Blocks (FBs) Used

- **F_MOVE**: iec61131::selection::F_MOVE (DataType=isobus::UT::Q::types::s2x1ObjectIDs) - copies pictures to OUT.up.u16ObjId/OUT.down.u16ObjId
- **AX_X_TO_BOOL**: adapter::conversion::unidirectional::AX_X_TO_BOOL - converts the AX selector into event CNF and value IN (BOOL)
- **F_MUX_2**: iec61131::selection::F_SEL - selects IN0 (up) for G=FALSE, IN1 (down) for G=TRUE
- **Q_NumericValue**: isobus::UT::Q::Q_NumericValue - writes the selected value (u32NewValue) to the VT object u16ObjId

Program Flow and Connections

1. INIT → Q_NumericValue.INIT; Q_NumericValue.INITO → INITO and → F_MOVE.REQ; F_MOVE.CNF → F_MUX_2.REQ (initialization of the target object IDs).
2. DI1 → AX_X_TO_BOOL.AX_IN; AX_X_TO_BOOL.CNF → F_MUX_2.REQ, AX_X_TO_BOOL.IN → F_MUX_2.G.
3. F_MOVE.OUT.up.u16ObjId → F_MUX_2.IN0; F_MOVE.OUT.down.u16ObjId → F_MUX_2.IN1.
4. F_MUX_2.CNF → Q_NumericValue.REQ; F_MUX_2.OUT → Q_NumericValue.u32NewValue; Picture → Q_NumericValue.u16ObjId.
5. QI → Q_NumericValue.QI; Q_NumericValue.QO → QO.

Technical Details

- Unlike SwitchPic_2_1 (there DI1: BOOL plus an external EI event), the AX variant needs no external trigger.
- The target ID_NULL can be used, for example, to switch an ObjectPointer "empty".

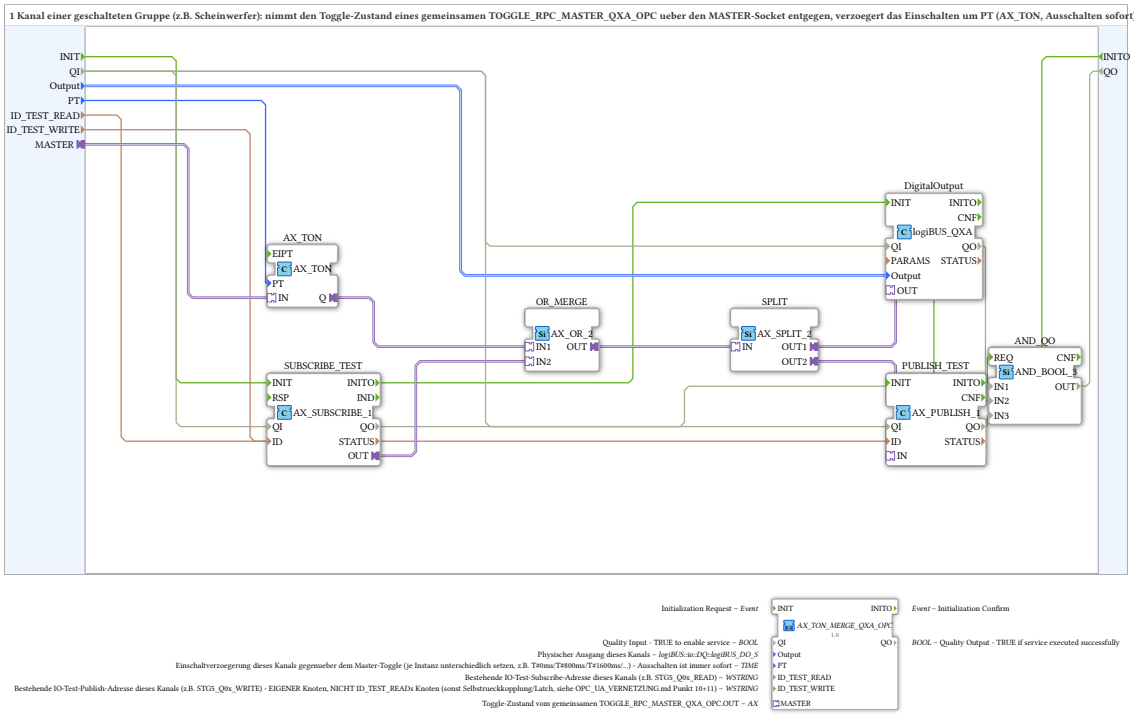
Application Scenarios

- Switching an ObjectPointer between e.g. Rectangle_Aktiv and ID_NULL depending on the operating state (e.g. display for elevator belt, hedgehog belt, filling belt)

Summary

AX_SwitchPic_2_1 writes one of two configurable object IDs to a VT ObjectPointer on every update of an AX selector.

AX_TON_MERGE_QXA_OPC



• — ○ • — ○ • — ○ •

Introduction

The `AX_TON_MERGE_QXA_OPC` function block (subapplication) implements a single channel of a switched group (e.g., a spotlight bank). It receives the toggle state from a common master via an adapter, delays the switch-on by a configurable time, and merges the result with the existing IO-test command for that channel. The combined signal drives the physical output and optionally publishes the state back for diagnostics. This design distributes capacitive inrush currents over time when multiple channels are switched together.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the init chain at `SUBSCRIBE_TEST.INIT`.

The actual signal processing remains purely data- and adapter-driven.

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `PUBLISH_TEST.INITO`, the end of the init chain.

Data Inputs

Name	Type	Description
`QI`	`BOOL`	Quality input, enables the service; initial value `TRUE`; passed on to `SUBSCRIBE_TEST`, `DigitalOutput` and `PUBLISH_TEST`.
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	Physical output for this channel (initial value: `logiBUS_DO::Invalid`).
`PT`	`TIME`	Switch-on delay relative to the master toggle. Set per instance (e.g., `T#0ms`, `T#800ms`, `T#1600ms`). Switch-off is always immediate.

ID_TEST_READ	WSTRING	Existing IO-test subscribe address for this channel (e.g., STG5_Q0x_READ).
ID_TEST_WRITE	WSTRING	Existing IO-test publish address for this channel (e.g., STG5_Q0x_WRITE). **Must be on its own OPC UA node** (not the same as ID_TEST_READ) to avoid feedback/latching issues.

Data Outputs

Name	Type	Description
QO	BOOL	Quality output, AND combination (AND_QO, AND_BOOL_3) of the QO outputs of SUBSCRIBE_TEST, DigitalOutput and PUBLISH_TEST.

Adapters

Socket	Type	Description
MASTER	adapter::types::unidirectional::AX	Receives the toggle state from a shared TOGGLE_RPC_MASTER_QXA_OPC block.

Functionality

1. The MASTER adapter supplies a toggle state (typically a boolean).
2. An internal AX_TON timer delays only the *activation* of this state by the configured PT. Deactivation (switch-off) propagates immediately.
3. The delayed signal is combined with the IO-test command via an AX_OR_2 gate. The IO-test command is obtained by subscribing to the OPC UA address ID_TEST_READ.
4. The merged output is then split (AX_SPLIT_2) into two paths:
   - **Path 1** drives the physical output via logiBUS_QXA.
   - **Path 2** publishes the state back to the OPC UA address ID_TEST_WRITE for monitoring or test feedback.

Initialization and quality: INIT → SUBSCRIBE_TEST.INIT, SUBSCRIBE_TEST.INITO → DigitalOutput.INIT, DigitalOutput.INITO → PUBLISH_TEST.INIT, PUBLISH_TEST.INITO → INITO and AND_QO.REQ. QI is passed on to SUBSCRIBE_TEST.QI, DigitalOutput.QI and PUBLISH_TEST.QI; their QO outputs are combined in AND_QO (iec61131::booleanOperators::AND_BOOL_3, IN1 to IN3) into the output QO.

The complete signal flow is internal and appears as a single logical unit to the application.

Technical Features

- **Configurable switch-on delay**: Each instance can have a distinct PT value, allowing a staggered startup of multiple channels.
- **Immediate switch-off**: No delay is applied to the falling edge, ensuring rapid deactivation.
- **Integration with existing IO-test infrastructure**: Reuses subscribe/publish addresses to merge manual test commands with the master toggle.
- **Decoupled OPC UA nodes**: Separate read and write addresses prevent self-feedback and unintended latching.
- **Physical output handling**: The block maps directly to a logiBUS_QXA digital output, abstracting low-level hardware access.

State Overview

This subapplication does not define an explicit state machine; its behavior is deterministic and solely based on the input data and the internal timer logic. The AX_TON component internally manages the delay timing, but no user-visible states are exposed.

Application Scenarios

Staggered activation of a spotlight bank

Consider a bank of six LED channels, each connected to the same master toggle via a shared

TOGGLE_RPC_MASTER_QXA_OPC. By assigning increasing PT values (e.g., 0 ms, 800 ms, 1600 ms, ...), the capacitive inrush currents of the LED drivers are spread over time instead of being switched simultaneously. This prevents overloads and reduces electrical stress.

Merging IO-test commands

When the IO-test system issues a command for a channel (via ID_TEST_READ), the AX_OR_2 gate combines it with the master-derived toggle. The physical output is driven by either source, and the resulting state is published back on ID_TEST_WRITE for verification.

Comparison with Similar Blocks

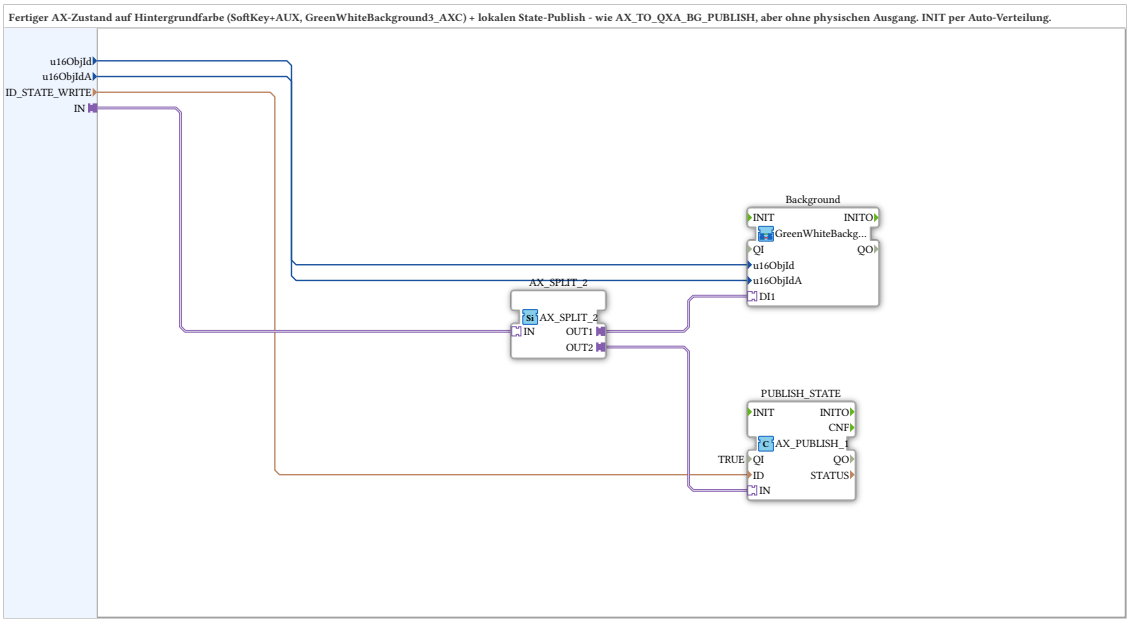
- **AX_TON_MERGE_QXA_OPC** (this block) – for channels that share a common master toggle and require a staggered switch-on. It adds a delay and merges with the IO-test input.
- **TOGGLE_RPC_MERGE_QXA_OPC** – for completely independent channels with their own soft-key, no shared master.
- **MERGE_SWITCH_1_QXA_OPC** – a simpler variant without the delayed-on feature, suitable when inrush spreading is not required.

Choose AX_TON_MERGE_QXA_OPC when you need to synchronize multiple channels from one master while avoiding simultaneous switching peaks.

Conclusion

AX_TON_MERGE_QXA_OPC provides a clean, reusable solution for controlling a switched group with controlled switch-on timing. Its integration with existing IO-test mechanisms and support for per-channel delays make it ideal for applications with capacitive loads and multiple outputs. The block encapsulates the required logic, simplifying application design and maintenance.

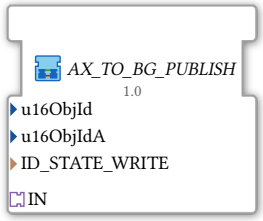
AX_TO_BG_PUBLISH



• – ○ • – ○ • – ○ •

Introduction

AX_TO_BG_PUBLISH brings a finished AX state to the background colour (SoftKey and AUX, GreenWhiteBackground3_AXC) and to a local state publish. It corresponds to AX_TO_QXA_BG_PUBLISH, but without a physical output. Initialization is done by automatic distribution (no event connections of its own). The block was extracted from Baender_EinAus.SUB (Background_Alle/AX_PUBLISH_Alle) as a reusable block.



Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

Variable	Type	Default	Comment
`u16ObjId`	UINT	`ID_NULL`	Object ID SoftKey; carries the background colour.
`u16ObjIdA`	UINT	`ID_NULL`	Object ID AuxFunction2 (joystick); also shows the background colour on the Aux handle.
`ID_STATE_WRITE`	WSTRING		Local publish address (`ACTION=WRITE`) for the state; subscribed remotely.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
`IN`	`adapter::types::unidirectional::AX`	Socket	Finished AX state; distributed to background and publish.

Function Blocks (FBs) Used

- **AX_SPLIT_2** (`adapter::events::unidirectional::AX_SPLIT_2`): adapter fan-out (strictly 1:1)
- **Background** (SubApp `MyLib::sys::GreenWhiteBackground3_AXC`): sets the background colour of SoftKey and Aux; see `GreenWhiteBackground3_AXC`
- **PUBLISH_STATE** (`adapter::net::AX_PUBLISH_1`, `QI = TRUE`): local state publish

Program Flow and Connections

1. `IN` -> `AX_SPLIT_2.IN`.
2. `AX_SPLIT_2.OUT1` -> `Background.DI1`; `AX_SPLIT_2.OUT2` -> `PUBLISH_STATE.IN`.
3. `u16ObjId` -> `Background.u16ObjId`; `u16ObjIdA` -> `Background.u16ObjIdA`; `ID_STATE_WRITE` -> `PUBLISH_STATE.ID`.
4. The block has no event and no `QI`/`QO` connections; `PUBLISH_STATE.QI` is fixed to `TRUE` via parameter.

Technical Features

- The `AX_SPLIT_2` is needed because the adapter fan-out is strictly 1:1.
- Initialization runs by automatic distribution instead of via a dedicated `INIT` input.
- Compared with `AX_TO_QXA_BG_PUBLISH` the physical output (`Output`) as well as `QI`/`QO` and `INIT`/`INITO` are missing.

Application Scenarios

- Pure display and publish channels without a hardware output of their own, e.g. identically structured channels with background colour on the SoftKey and Aux handle plus state publishing.

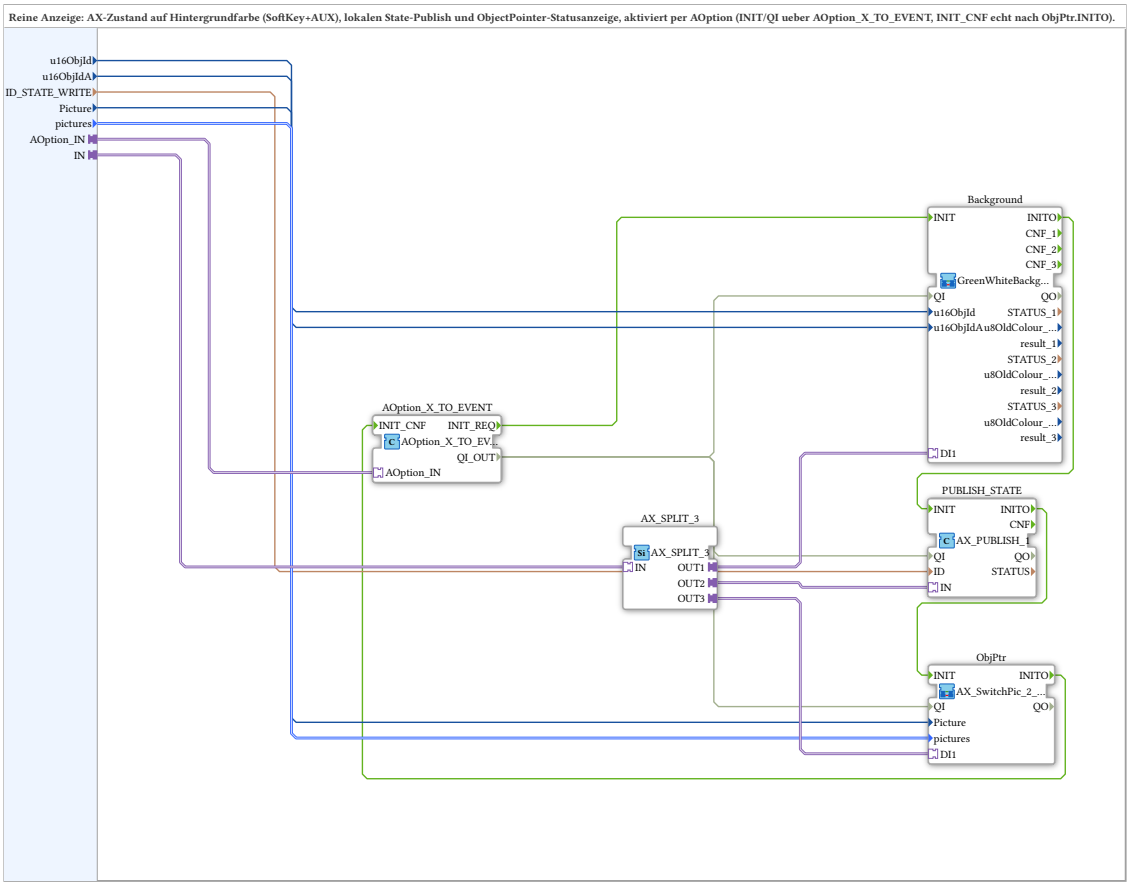
Summary

`AX_TO_BG_PUBLISH` distributes a finished AX state to the background colour of SoftKey and Aux and to a local state publish, without a physical output.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

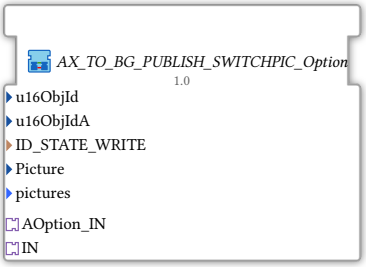
AX_TO_BG_PUBLISH_SWITCHPIC_Option



• – ○ • – ○ • – ○ •

Introduction

`AX_TO_BG_PUBLISH_SWITCHPIC_Option` is a pure display: it brings a finished AX state to the background colour (SoftKey and AUX), a local state publish and the ObjectPointer status display (`AX_SwitchPic_2_1`). It is activated via `AOption`: `INIT` and `QI` come through `AOption_X_TO_EVENT`, and `INIT_CNF` is returned for real after `ObjPtr.INITO`. It is the display part of `AX_TO_QXA_BG_PUBLISH_SWITCHPIC_OPT` as a block of its own and corresponds to `AX_TO_QXA_BG_PUBLISH_SWITCHPIC` without the physical output.



Object ID Softkey - traegt die Hintergrundfarbe – *UINT*
Object ID AuxFunction2 (Joystick) - zeigt Hintergrundfarbe am Aux-Handle mit – *UINT*
Lokale Publish-Adresse (ACTION=WRITE) fuer den Zustand - wird remote abonniert – *WSTRING*
Object ID des ObjectPointer, der den Status-Kaestchen-Inhalt traegt – *UINT*
die beiden Ziel-Object-IDs (up=aus, down=an) – *isobus::UT::Q::types::s2x1ObjectIDs*

Aktiviert/deaktiviert die Anzeige – *AOption*
Fertiger AX-Zustand fuer Hintergrund, Publish und ObjectPointer – *AX*

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs

Variable	Type	Default	Comment
u16ObjId	UINT	ID_NULL	Object ID SoftKey; carries the background colour.
u16ObjIdA	UINT	ID_NULL	Object ID AuxFunction2 (joystick); also shows the background colour on the Aux handle.
ID_STATE_WRITE	WSTRING		Local publish address (ACTION=WRITE) for the state; subscribed remotely.
Picture	UINT		Object ID of the ObjectPointer carrying the status box content.
pictureIDs	isobus::UT::Q::types::s2x1ObjectIDs	(up := (u16ObjId := ID_NULL), down := (u16ObjId := ID_NULL))	The two target object IDs (up = off, down = on).

Data Outputs
None.

Adapters

Adapter	Type	Direction	Comment
AOption_IN	Option::AOption	Socket	Enables/disables the display.
IN	adapter::types::unidirectional::AX	Socket	Finished AX state for background, publish and ObjectPointer.

Function Blocks (FBs) Used
- **AOption_X_TO_EVENT** (Option::conversion::AOption_X_TO_EVENT): converts the option into INIT_REQ/QI_OUT and accepts INIT_CNF
- **AX_SPLIT_3** (adapter::events::unidirectional::AX_SPLIT_3): adapter fan-out (strictly 1:1)
- **Background** (SubApp MyLib::sys::GreenWhiteBackground3_AX): background colour of SoftKey and Aux; see GreenWhiteBackground3_AX
- **PUBLISH_STATE** (adapter::net::AX_PUBLISH_1): local state publish
- **ObjPtr** (SubApp MyLib::sys::AX_SwitchPic_2_1): ObjectPointer status display; see AX_SwitchPic_2_1

Program Flow and Connections
1. Initialization chain: AOption_X_TO_EVENT.INIT_REQ -> Background.INIT; Background.INITO -> PUBLISH_STATE.INIT; PUBLISH_STATE.INITO -> ObjPtr.INIT; ObjPtr.INITO -> AOption_X_TO_EVENT.INIT_CNF.
2. Qualifier: AOption_X_TO_EVENT.QI_OUT -> Background.QI, PUBLISH_STATE.QI and ObjPtr.QI.
3. Option: AOption_IN -> AOption_X_TO_EVENT.AOption_IN.

4. AX state: `IN -> AX_SPLIT_3.IN`; `OUT1 -> Background.DI1`; `OUT2 -> PUBLISH_STATE.IN`; `OUT3 -> ObjPtr.DI1`.
5. Data: `u16ObjId -> Background.u16ObjId`; `u16ObjIdA -> Background.u16ObjIdA`; `ID_STATE_WRITE -> PUBLISH_STATE.ID`; `Picture -> ObjPtr.Picture`; `pictures -> ObjPtr.pictures`.
6. The block has no event or `QI`/`QO` connections of its own and no plug; activation and acknowledgment run via `AOption_IN`.

Technical Features

- The `AX_SPLIT_3` is needed because the adapter fan-out is strictly 1:1.
- `INIT_CNF` comes for real from the end of the chain (`ObjPtr.INITO`), not from a short-circuit connection.
- There is no `AOption_OUT` plug; the block therefore cannot be daisy-chained.

Application Scenarios

- Pure display side of a channel (background colour, state publish, status picture) that is active only when the option is selected, e.g. on an `AOption_OR_2` combination of several variants.

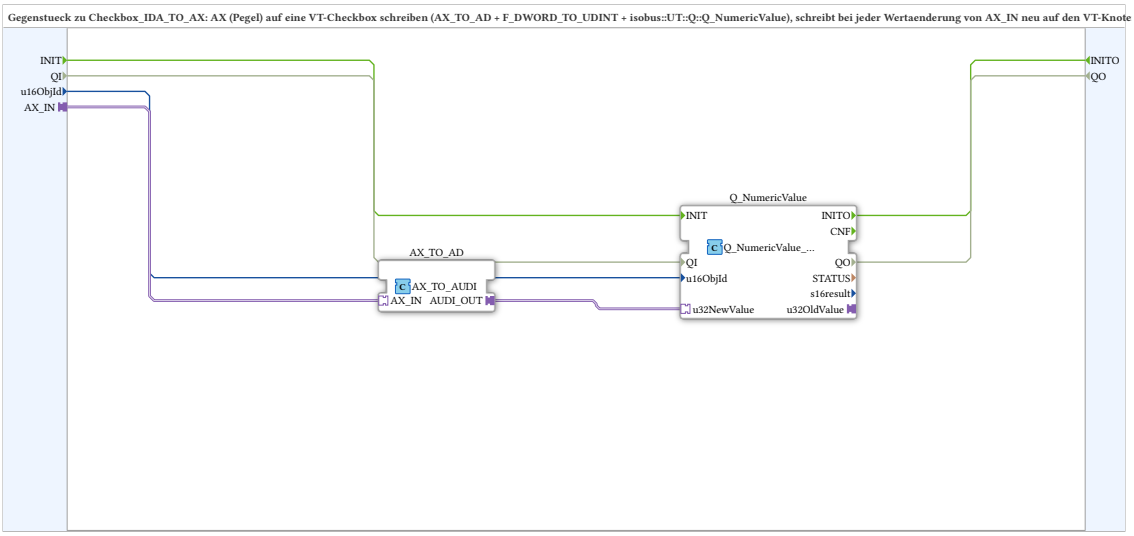
Summary

`AX_TO_BG_PUBLISH_SWITCHPIC_Option` shows a finished AX state as background colour, state publish and status picture and is activated via `AOption`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

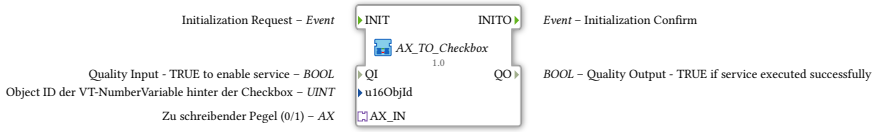
AX_TO_Checkbox



• – ○ • – ○ • – ○ •

Introduction

`AX_TO_Checkbox` is the counterpart of `Checkbox_IDA_TO_AX`: it writes a level (`AX_IN`, 0/1) to a VT checkbox (ECU → VT instead of VT → ECU). Internally, `AX_TO_AD` (`adapter::conversion::unidirectional::AX_TO_AUDI`) converts the level into a UDINT value, which `Q_NumericValue` (`isobus::UT::Q::Q_NumericValue_AUDI`) sends to the VT via `u32NewValue`.



Interface

Event ports

- **Input `INIT`** (`EInit`): initialization request - to `Q_NumericValue.INIT`.
- **Output `INITO`** (`EInit`): initialization confirmation - from `Q_NumericValue.INITO`.

Input variables

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable the service (to `Q_NumericValue.QI`).
- `u16ObjId` (UINT, default `ID_NULL`): ObjectID of the VT NumberVariable behind the checkbox.

Output variables

- `QO` (BOOL): Quality Output - TRUE if the service was executed successfully (from `Q_NumericValue.QO`).

Adapters

- **Socket `AX_IN`** (`adapter::types::unidirectional::AX`): level to be written (0/1).

Apart from `INIT`/`INITO`, the block has no event ports of its own; writing is triggered by the adapter `AX_IN`.

How it works

1. `AX_IN` is converted into a UDINT value via `AX_TO_AD`.
2. `Q_NumericValue` writes this value to the VT node under `u16ObjId`. According to the source comment, it is rewritten on every value change of `AX_IN`.
3. `INIT` ⟶ `Q_NumericValue.INIT`, `Q_NumericValue.INITO` ⟶ `INITO`, `QI` ⟶ `Q_NumericValue.QI`, `Q_NumericValue.QO` ⟶ `QO`.

Technical notes

- `u16ObjId` is the ObjectID of the **CNumberVariable** that the checkbox references as its value store - **not** the ObjectID of the checkbox (`CInputBoolean`) itself. The VT expects value changes under the NumberVariable ID (analogous to `Checkbox_IDA_TO_AX`).
- Instance uniqueness: `Q_NumericValue` allows only one instance per target ObjectID in the entire project.

Summary

Writes an AX level to a VT checkbox via its associated NumberVariable.

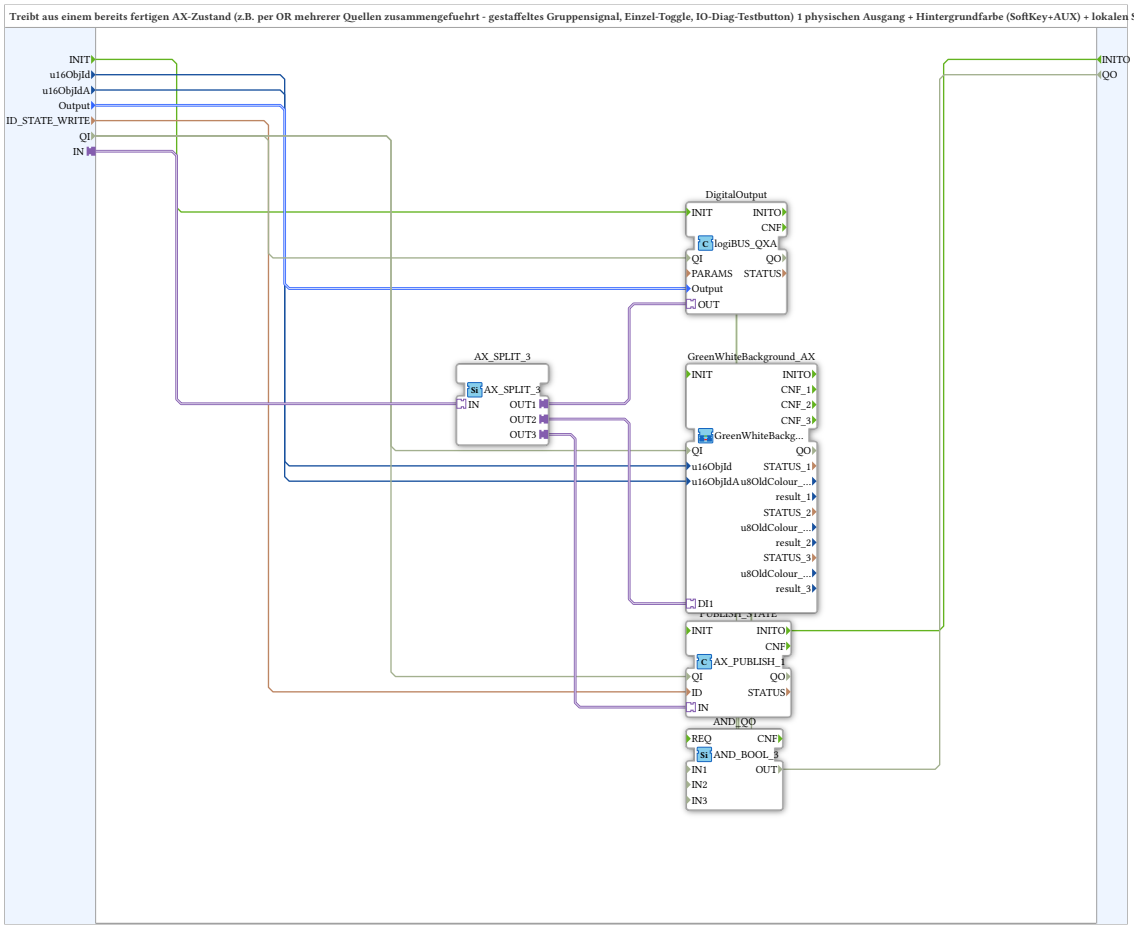
🛠️ Related Exercises

- Uebung_094b5_AX

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

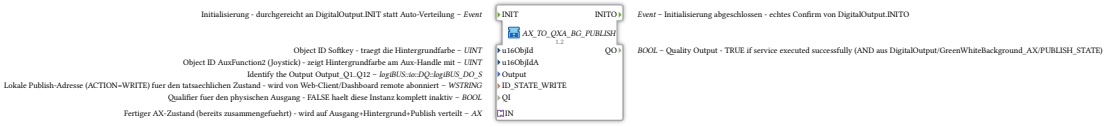
AX_TO_QXA_BG_PUBLISH



• — ○ • — ○ • — ○ •

Introduction

`AX_TO_QXA_BG_PUBLISH` drives, from an already finished AX state (e.g. merged by OR of several sources such as a staggered group signal, single toggle or IO diagnostics test button), **one** physical output, the background colour (SoftKey and AUX) and a local state publish for web client and OPC dashboard. It is the counterpart to `MyLib::sys::AX_SoftkeyToggle_RS_TO_QXA_BG`, which still generates the toggle/reset itself; here the AX state is already available.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation; passed on to `DigitalOutput.INIT` instead of automatic distribution

Event Outputs

- **INITO** (`EInit`): initialisation completed; real confirm from `PUBLISH_STATE.INITO` at the end of the chain `DigitalOutput` ⟶ `GreenWhiteBackground_AX` ⟶ `PUBLISH_STATE`

Data Inputs

- **QI** (BOOL): qualifier; goes to `DigitalOutput`, `GreenWhiteBackground_AX` and `PUBLISH_STATE`, FALSE keeps this instance completely inactive (default `TRUE`)
- **u16ObjId** (UINT): object ID of the SoftKey, carries the background colour (default `ID_NULL`)

- **u16ObjIdA** (UINT): object ID of the AuxFunction2 (joystick), also shows the background colour at the aux handle (default `ID_NULL`)
- **Output** (`logiBUS::io::DQ::logiBUS_DO_S`): selection of the output `Output_Q1`..`Output_Q12` (default `logiBUS_DO::Invalid`)
- **ID_STATE_WRITE** (WSTRING): local publish address (`ACTION=WRITE`) for the actual state; subscribed remotely by web client/dashboard

Data Outputs

- **QO** (BOOL): quality output - TRUE if the service was executed successfully (AND of `DigitalOutput`, `GreenWhiteBackground_AX` and `PUBLISH_STATE`)

Adapters

- **IN** (socket, `adapter::types::unidirectional::AX`): finished, already merged AX state; distributed to output, background and publish

Functionality

`IN` is routed to an `AX_SPLIT_3` (`adapter::events::unidirectional::AX_SPLIT_3`), which distributes the state to three destinations:

- `AX_SPLIT_3.OUT1` ⟶ `DigitalOutput.OUT`: `DigitalOutput` (`logiBUS::io::DQ::logiBUS_QXA`) switches the physical output; `Output` ⟶ `DigitalOutput.Output`, `QI` ⟶ `DigitalOutput.QI` and the event `INIT` ⟶ `DigitalOutput.INIT`.
- `AX_SPLIT_3.OUT2` ⟶ `GreenWhiteBackground_AX.DI1`: `GreenWhiteBackground_AX` (`MyLib::sys::GreenWhiteBackground3_AX`) colours the background green (TRUE) or white (FALSE); `u16ObjId` and `u16ObjIdA` go to its inputs of the same name, `QI` to `GreenWhiteBackground_AX.QI`.
- `AX_SPLIT_3.OUT3` ⟶ `PUBLISH_STATE.IN`: `PUBLISH_STATE` (`adapter::net::AX_PUBLISH_1`, `QI` from outside) publishes the state to the address `ID_STATE_WRITE` (`ID_STATE_WRITE` ⟶ `PUBLISH_STATE.ID`).

Initialisation and quality: `INIT` ⟶ `DigitalOutput.INIT`, `DigitalOutput.INITO` ⟶ `GreenWhiteBackground_AX.INIT`, `GreenWhiteBackground_AX.INITO` ⟶ `PUBLISH_STATE.INIT`, `PUBLISH_STATE.INITO` ⟶ `INITO`. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`) with `DigitalOutput.QO`, `GreenWhiteBackground_AX.QO` and `PUBLISH_STATE.QO` at `IN1`..`IN3`; its `REQ` is triggered by `PUBLISH_STATE.INITO`.

Technical Details

- Extracted from `Baender_EinAus` (wired three times identically there for hedgehog belt, elevator and cross belt)
- `ID_STATE_WRITE` is new because the web mirror needs its own state publish for the individual belt softkeys
- Version 1.1: `QI` and `INIT` are passed on to `DigitalOutput` (instead of a hard `TRUE` and automatic distribution), so that e.g. classic-vs-PVEA variants sharing pins can keep an instance inactive and the initialisation can be sequenced by the caller
- Version 1.2: new event output `INITO` (initialisation completed) and data output `QO` (AND of the inner `QO`); initialisation runs as a chain `DigitalOutput` ⟶ `GreenWhiteBackground_AX` ⟶ `PUBLISH_STATE`, `QI` is distributed to all three instances
- There are no own event or data ports for the state; everything runs via the AX adapter `IN`

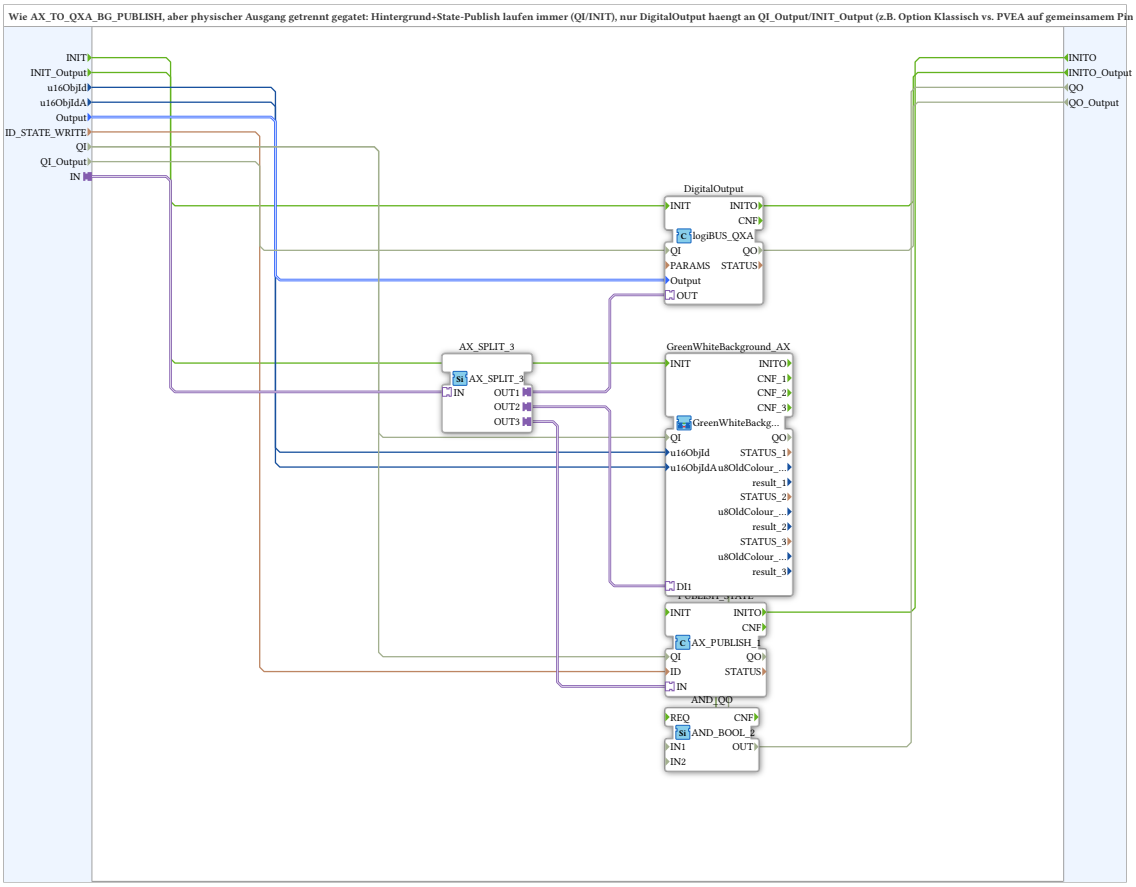
Application Scenarios

- Group signal or OR-merged state that is to switch an output, show the softkey colour and be published for the web
- Several time-staggered channels, each supplying its own state

Conclusion

`AX_TO_QXA_BG_PUBLISH` distributes a finished AX state to the physical output, background colour and state publish.

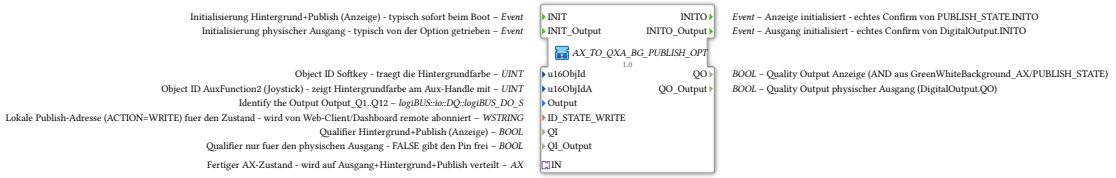
AX_TO_QXA_BG_PUBLISH_OPT



• — ○ • — ○ • — ○ •

Introduction

`AX_TO_QXA_BG_PUBLISH_OPT` corresponds to `AX_TO_QXA_BG_PUBLISH`, but gates the physical output separately: background and state publish always run via `QI/INIT`, only the `DigitalOutput` depends on `QI_Output/INIT_Output` (e.g. option “classic” versus “PVEA” on a shared pin). The block is a copy of `AX_TO_QXA_BG_PUBLISH` v1.2; the reason was that with the PVEA variant the SoftKey background and state publish stayed inactive together with the output.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization of background and publish (display); typically right at boot
- **INIT_Output** (`EInit`): initialization of the physical output; typically driven by the option

Event Outputs

- **INITO** (`EInit`): display initialized; real confirm from `PUBLISH_STATE.INITO`
- **INITO_Output** (`EInit`): output initialized; real confirm from `DigitalOutput.INITO`

Data Inputs

Variable	Type	Default	Comment

u16ObjId	UINT	ID_NULL	Object ID SoftKey; carries the background colour.
u16ObjIdA	UINT	ID_NULL	Object ID AuxFunction2 (joystick); also shows the background colour on the Aux handle.
Output	logiBUS::io::DQ::logiBUS_DO_1	logiBUS_DO::Invalid	Selection of the output Output_Q1..Output_Q12.
ID_STATE_WRITE	WSTRING		Local publish address (ACTION=WRITE) for the state; subscribed remotely by web client/dashboard.
QI	BOOL	TRUE	Qualifier for background and publish (display).
QI_Output	BOOL	TRUE	Qualifier for the physical output only; FALSE releases the pin.

Data Outputs

Variable	Type	Default	Comment
QO	BOOL		Quality output display (AND of GreenWhiteBackground_AX.QO and PUBLISH_STATE.QO).
QO_Output	BOOL		Quality output physical output (DigitalOutput.QO).

Adapters

Adapter	Type	Direction	Comment
IN	adapter::types::unidirectional::AX	Socket	Finished AX state; distributed to output, background and publish.

Function Blocks (FBs) Used

- **AX_SPLIT_3** (adapter::events::unidirectional::AX_SPLIT_3): adapter fan-out (strictly 1:1)
- **DigitalOutput** (logiBUS::io::DQ::logiBUS_QXA): physical output
- **GreenWhiteBackground_AX** (SubApp MyLib::sys::GreenWhiteBackground3_AX): background colour of SoftKey and Aux; see GreenWhiteBackground3_AX
- **PUBLISH_STATE** (adapter::net::AX_PUBLISH_1): local state publish
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): combines the QO of the display

Program Flow and Connections

1. AX state: IN -> AX_SPLIT_3.IN; OUT1 -> DigitalOutput.OUT; OUT2 -> GreenWhiteBackground_AX.DI1; OUT3 -> PUBLISH_STATE.IN.
2. Output initialization: INIT_Output -> DigitalOutput.INIT; DigitalOutput.INITO -> INITO_Output.
3. Display initialization: INIT -> GreenWhiteBackground_AX.INIT; GreenWhiteBackground_AX.INITO -> PUBLISH_STATE.INIT; PUBLISH_STATE.INITO -> INITO and -> AND_QO.REQ.
4. Qualifier: QI -> GreenWhiteBackground_AX.QI and PUBLISH_STATE.QI; QI_Output -> DigitalOutput.QI.
5. Quality: GreenWhiteBackground_AX.QO -> AND_QO.IN1; PUBLISH_STATE.QO -> AND_QO.IN2; AND_QO.OUT -> QO; DigitalOutput.QO -> QO_Output.
6. Data: u16ObjId, u16ObjIdA -> GreenWhiteBackground_AX; Output -> DigitalOutput.Output; ID_STATE_WRITE -> PUBLISH_STATE.ID.

Technical Features

- `QI`/`INIT` now apply only to background and publish; `QI_Output`/`INIT_Output`/`INITO_Output`/`QO_Output` belong exclusively to the `DigitalOutput`.
- The `AX_SPLIT_3` is needed because the adapter fan-out is strictly 1:1.

Application Scenarios

- Channels where the physical output can be switched off via an option (e.g. classic versus PVEA on a shared pin), while SoftKey background and state publish should always run.

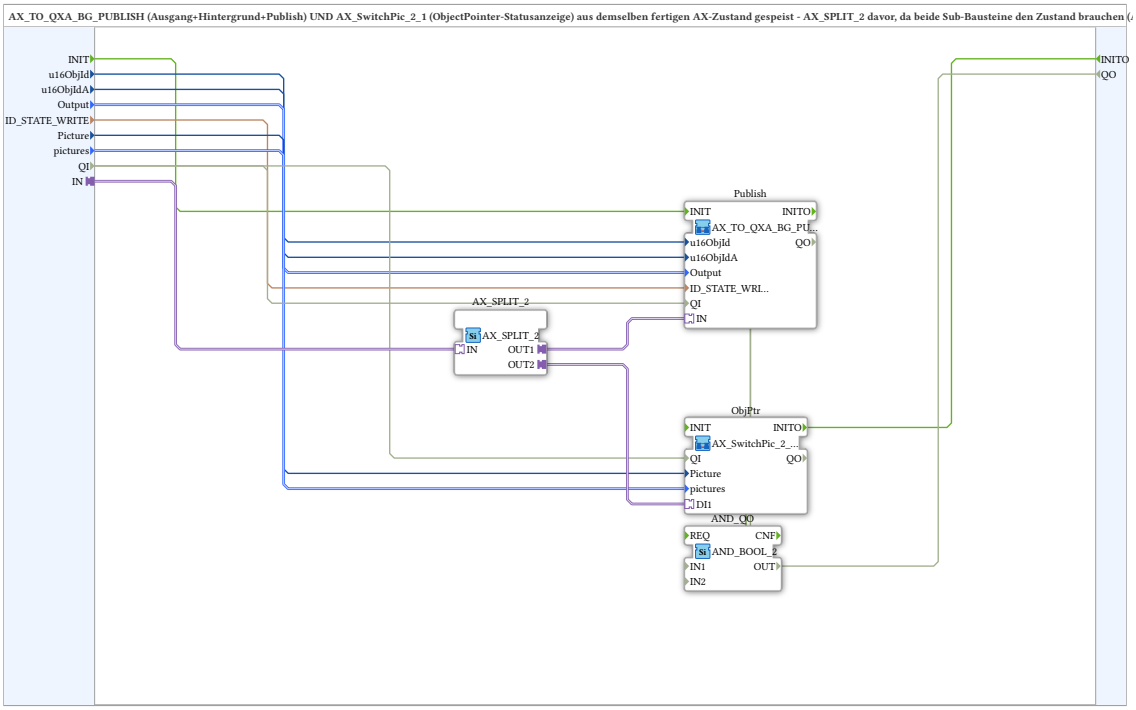
Summary

`AX_TO_QXA_BG_PUBLISH_OPT` distributes a finished AX state to output, background and publish, with the physical output enabled and initialized separately.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

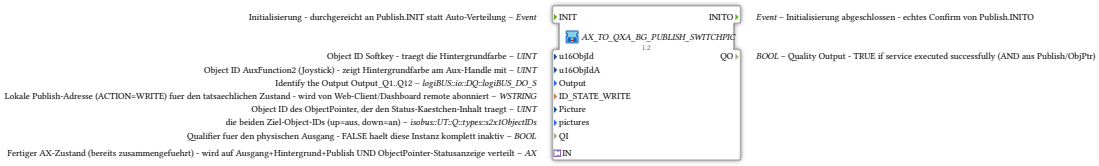
AX_TO_QXA_BG_PUBLISH_SWITCHPIC



• — ○ • — ○ • — ○ •

Introduction

`AX_TO_QXA_BG_PUBLISH_SWITCHPIC` feeds `AX_TO_QXA_BG_PUBLISH` (output + background + publish) **and** `AX_SwitchPic_2_1` (ObjectPointer status display) from the same finished AX state. An `AX_SPLIT_2` in front is required because both sub-blocks need the state (adapter fan-out rule). Extracted from `Baender_EinAus`, where this combination existed three times identically (Igel, Elevator, Querband).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation; passed on to `Publish.INIT` instead of automatic distribution

Event Outputs

- **INITO** (`EInit`): initialisation completed; real confirm from `ObjPtr.INITO` (end of the chain `Publish` → `ObjPtr`)

Data Inputs

- **QI** (`BOOL`): qualifier for the physical output; FALSE keeps this instance completely inactive (default `TRUE`)
- **u16ObjId** (`UINT`): object ID of the SoftKey, carries the background colour (default `ID_NULL`)
- **u16ObjIdA** (`UINT`): object ID of the AuxFunction2 (joystick), also shows the background colour on the aux handle (default `ID_NULL`)
- **Output** (`logiBUS::io::DQ::logiBUS_DO_S`): selection of the output `Output_Q1`..`Output_Q12` (default `logiBUS_DO::Invalid`)
- **ID_STATE_WRITE** (`WSTRING`): local publish address (`ACTION=WRITE`) for the actual state, subscribed remotely by web client/dashboard
- **Picture** (`UINT`): object ID of the ObjectPointer that carries the status box content
- **pictures** (`isobus::UT::Q::types::s2x1ObjectIDs`): the two target object IDs (`up` = off, `down` = on); default both `ID_NULL`

Data Outputs

- **QO** (`BOOL`): Quality Output, AND of `Publish.QO` and `ObjPtr.QO`

Adapters

- **IN** (socket, `adapter::types::unidirectional::AX`): finished, already merged AX state, distributed to output + background + publish and to the ObjectPointer status display

Function Blocks (FBs) Used

- **`AX_SPLIT_2`** (`adapter::events::unidirectional::AX_SPLIT_2`)
- **`Publish`** (SubApp `MyLib::sys::AX_TO_QXA_BG_PUBLISH`)
- **`ObjPtr`** (SubApp `MyLib::sys::AX_SwitchPic_2_1`)
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`)

Functionality

- `IN` → `AX_SPLIT_2.IN`
- `AX_SPLIT_2.OUT1` → `Publish.IN`; `u16ObjId`, `u16ObjIdA`, `Output`, `ID_STATE_WRITE`, `QI` → the like-named inputs of `Publish`
- `INIT` → `Publish.INIT`; `Publish.INITO` → `ObjPtr.INIT`; `ObjPtr.INITO` → `INITO` (and → `AND_QO.REQ`)
- `QI` → `ObjPtr.QI` (in addition to `Publish.QI`)
- `Publish.QO` → `AND_QO.IN1`; `ObjPtr.QO` → `AND_QO.IN2`; `AND_QO.OUT` → `QO`
- `AX_SPLIT_2.OUT2` → `ObjPtr.DI1`; `Picture` → `ObjPtr.Picture`, `pictures` → `ObjPtr.pictures`

Technical Details

- The `AX_SPLIT_2` is necessary because an adapter must not branch to two sub-blocks
- No logic of its own, just a combination of two existing SubApps
- Version 1.1: `QI` and `INIT` are passed on to `Publish` (`AX_TO_QXA_BG_PUBLISH` v1.1) (QI gate classic-vs-PVEA, initialisation sequenced by the caller)
- Version 1.2: new event output `INITO`, passed on from `Publish.INITO` via `ObjPtr`; additionally `QO` as AND of `Publish.QO` and `ObjPtr.QO`

Application Scenarios

- Several identically structured channels (e.g. conveyor belts) that each switch an output, show the softkey colour, publish, and additionally switch a status picture

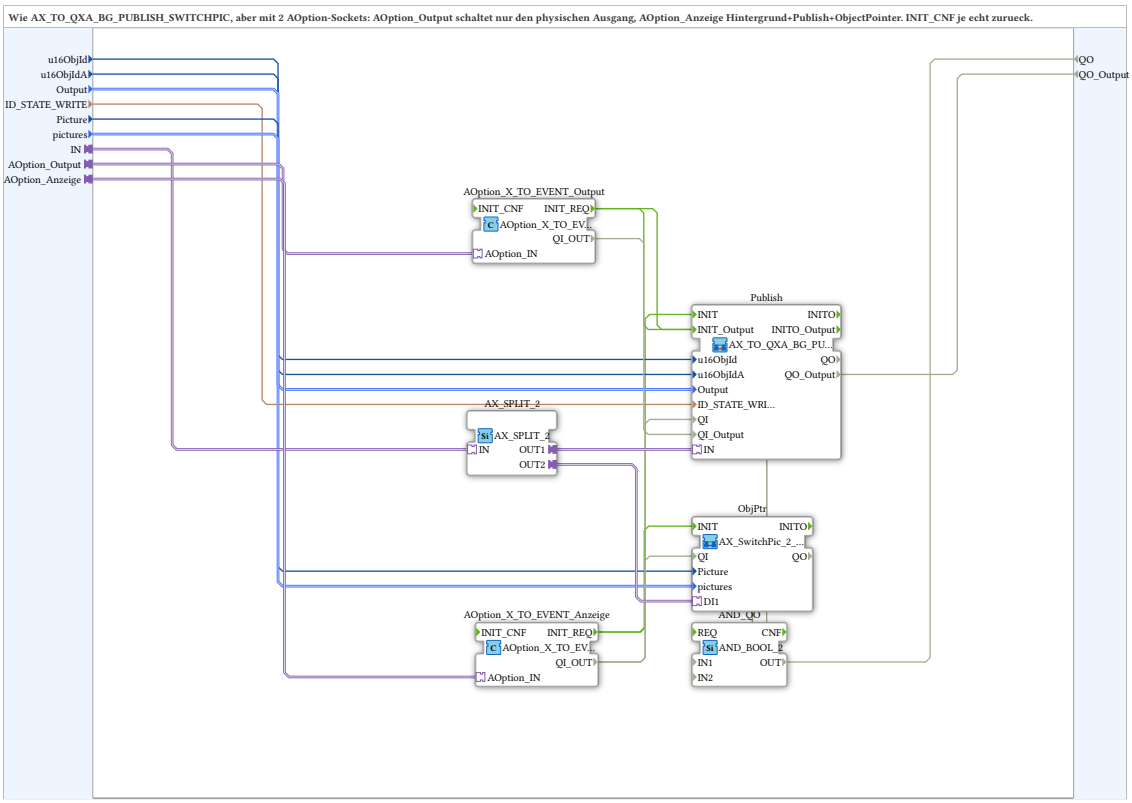
Summary

`AX_TO_QXA_BG_PUBLISH_SWITCHPIC` distributes a finished AX state to output/background/publish and an ObjectPointer status display.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

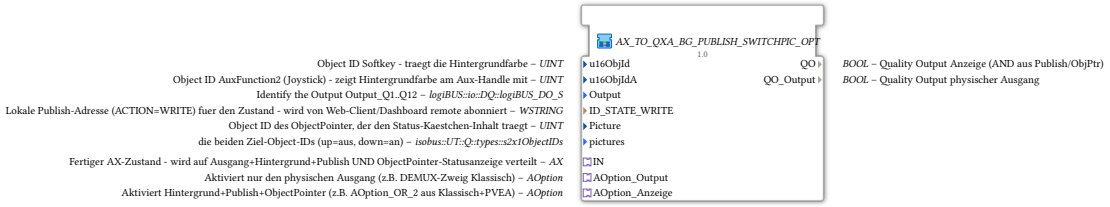
AX_TO_QXA_BG_PUBLISH_SWITCHPIC_OPT



• – ○ • – ○ • – ○ •

Introduction

`AX_TO_QXA_BG_PUBLISH_SWITCHPIC_OPT` corresponds to `AX_TO_QXA_BG_PUBLISH_SWITCHPIC`, but has two `AOption` sockets instead of `QI`/`INIT`: `AOption_Output` switches only the physical output, `AOption_Anzeige` switches background, publish and ObjectPointer. `INIT_CNF` is returned for real in each case. The block is a copy of `AX_TO_QXA_BG_PUBLISH_SWITCHPIC` v1.2; the reason was that with the PVEA variant the SoftKey background, ObjectPointer and state publish stayed inactive. The output part is also available on its own as `AX_TO_QXA_Option`, the display part as `AX_TO_BG_PUBLISH_SWITCHPIC_Option`.



Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

Variable	Type	Default	Comment
u16ObjId	UINT	ID_NULL	Object ID SoftKey; carries the background colour.
u16ObjIdA	UINT	ID_NULL	Object ID AuxFunction2 (joystick); also shows the background colour on the Aux handle.
Output	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Selection of the output Output_Q1..Output_Q12.
ID_STATE_WRITE	WSTRING		Local publish address (ACTION=WRITE) for the state; subscribed remotely by web client/ dashboard.
Picture	UINT		Object ID of the ObjectPointer carrying the status box content.
pictureIDs	isobus::UT::Q::types::s2x1ObjectIDs	(up := (u16ObjId := ID_NULL), down := (u16ObjId := ID_NULL))	The two target object IDs (up = off, down = on).

Data Outputs

Variable	Type	Default	Comment
QO	BOOL		Quality output display (AND of Publish.QO and ObjPtr.QO).
QO_Output	BOOL		Quality output physical output (Publish.QO_Output).

Adapters

Adapter	Type	Direction	Comment
IN	adapter::types::unidirectional::AX	Socket	Finished AX state; distributed to output, background, publish and ObjectPointer status display.
AOption_Output	Option::AOption	Socket	Enables only the physical output (e.g. DEMUX branch classic).
AOption_Anzeige	Option::AOption	Socket	Enables background, publish and ObjectPointer (e.g. AOption_OR_2 of classic and PVEA).

Function Blocks (FBs) Used

- **AX_SPLIT_2** (adapter::events::unidirectional::AX_SPLIT_2): adapter fan-out (strictly 1:1)
- **AOption_X_TO_EVENT_Output** (Option::conversion::AOption_X_TO_EVENT): option for the physical output
- **AOption_X_TO_EVENT_Anzeige** (Option::conversion::AOption_X_TO_EVENT): option for the display
- **Publish** (SubApp MyLib::sys::AX_TO_QXA_BG_PUBLISH_OPT): output, background and state publish; see AX_TO_QXA_BG_PUBLISH_OPT
- **ObjPtr** (SubApp MyLib::sys::AX_SwitchPic_2_1): ObjectPointer status display; see AX_SwitchPic_2_1
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): combines the QO of the display

Program Flow and Connections

1. AX state: `IN` -> `AX_SPLIT_2.IN`; `OUT1` -> `Publish.IN`; `OUT2` -> `ObjPtr.DI1`.
2. Options: `AOption_Output` -> `AOption_X_TO_EVENT_Output.AOption_IN`; `AOption_Anzeige` -> `AOption_X_TO_EVENT_Anzeige.AOption_IN`.
3. Display initialization: `AOption_X_TO_EVENT_Anzeige.INIT_REQ` -> `Publish.INIT`; `Publish.INITO` -> `ObjPtr.INIT`; `ObjPtr.INITO` -> `AOption_X_TO_EVENT_Anzeige.INIT_CNF` and -> `AND_QO.REQ`.
4. Output initialization: `AOption_X_TO_EVENT_Output.INIT_REQ` -> `Publish.INIT_Output`; `Publish.INITO_Output` -> `AOption_X_TO_EVENT_Output.INIT_CNF`.
5. Qualifier: `AOption_X_TO_EVENT_Anzeige.QI_OUT` -> `Publish.QI` and `ObjPtr.QI`; `AOption_X_TO_EVENT_Output.QI_OUT` -> `Publish.QI_Output`.
6. Quality: `Publish.QO` -> `AND_QO.IN1`; `ObjPtr.QO` -> `AND_QO.IN2`; `AND_QO.OUT` -> `QO`; `Publish.QO_Output` -> `QO_Output`.
7. Data: `u16ObjId`, `u16ObjIdA`, `Output`, `ID_STATE_WRITE` -> inputs of `Publish` with the same name; `Picture` -> `ObjPtr.Picture`; `pictures` -> `ObjPtr.pictures`.

Technical Features

- The block has no event inputs/outputs of its own; initialization runs via the two `AOption` sockets.
- Output and display can be enabled independently of each other.
- The `AX_SPLIT_2` is needed because the adapter fan-out is strictly 1:1.

Application Scenarios

- Several identically structured channels (e.g. conveyor belts) with output, SoftKey colour, publish and status picture, where the output should be active for one option only (e.g. classic), but the display for several options (e.g. classic and PVEA).

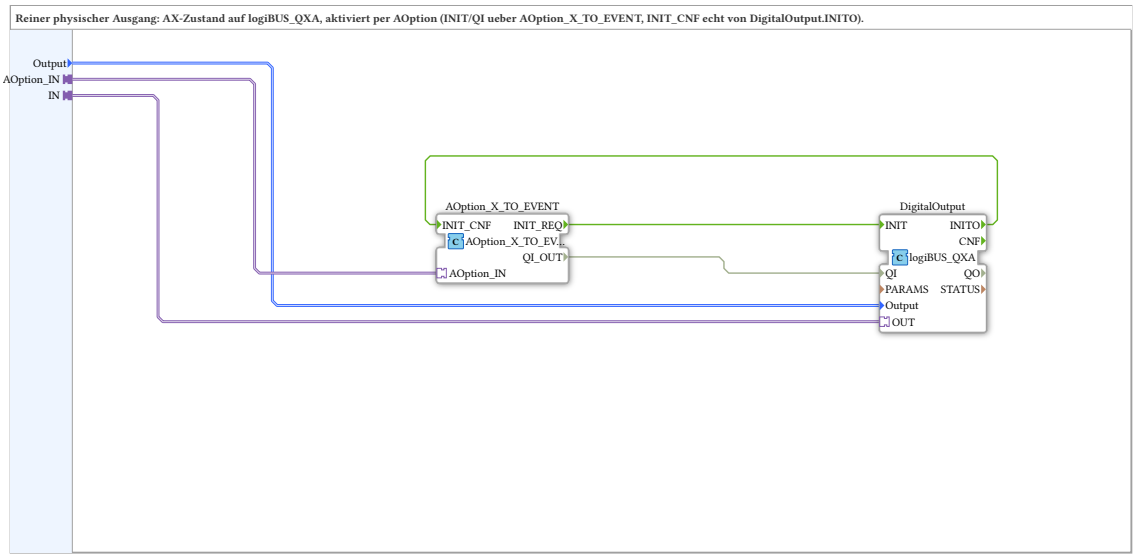
Summary

`AX_TO_QXA_BG_PUBLISH_SWITCHPIC_OPT` distributes a finished AX state to output, background, publish and ObjectPointer status display and enables output and display via two separate `AOption` sockets.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AX_TO_QXA_Option

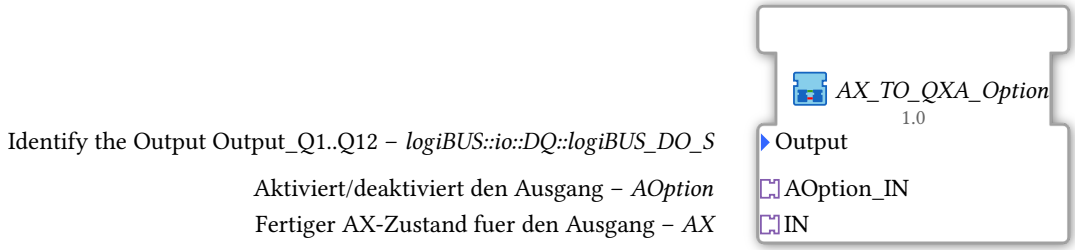


• – ○ • – ○ • – ○ •

Introduction

`AX_TO_QXA_Option` is a pure physical output: it brings a finished AX state to `logiBUS_QXA`. It is activated via `AOption` (`INIT/QI` through `AOption_X_TO_EVENT`, `INIT_CNF` for real from `DigitalOutput.INITO`). It is

the output part of AX_TO_QXA_BG_PUBLISH_SWITCHPIC_OPT as a block of its own, following the pattern of OptionalGroup_001.



Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

Variable	Type	Default	Comment
Output	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Selection of the output Output_Q1..Output_Q12.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
AOption_IN	Option::AOption	Socket	Enables/disables the output.
IN	adapter::types::unidirectional::AX	Socket	Finished AX state for the output.

Function Blocks (FBs) Used

- **AOption_X_TO_EVENT** (Option::conversion::AOption_X_TO_EVENT): converts the option into INIT_REQ/QI_OUT and accepts INIT_CNF
- **DigitalOutput** (logiBUS::io::DQ::logiBUS_QXA): physical output

Program Flow and Connections

1. Option: AOption_IN -> AOption_X_TO_EVENT.AOption_IN.
2. Initialization: AOption_X_TO_EVENT.INIT_REQ -> DigitalOutput.INIT; DigitalOutput.INITO -> AOption_X_TO_EVENT.INIT_CNF.
3. Qualifier: AOption_X_TO_EVENT.QI_OUT -> DigitalOutput.QI.
4. AX state: IN -> DigitalOutput.OUT.
5. Data: Output -> DigitalOutput.Output.
6. The block has no event or QI/QO connections of its own and no plug.

Technical Features

- No logic of its own: only the option connection and logiBUS_QXA.
- INIT_CNF comes for real from DigitalOutput.INITO.
- There is no AOption_OUT plug; the block therefore cannot be daisy-chained.

Application Scenarios

- Physical output that is to be switched only when the option (e.g. classic) is selected, separate from display and publish.

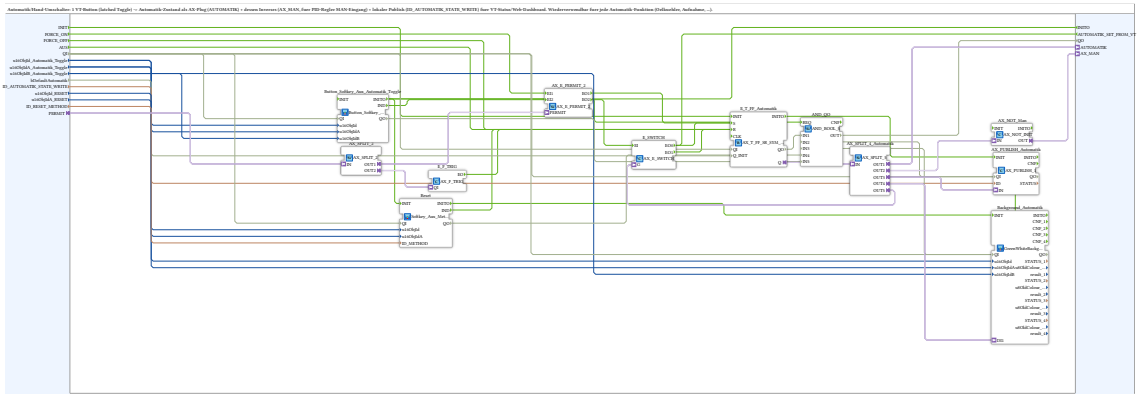
Summary

`AX_TO_QXA_Option` drives a physical output from a finished AX state and is activated via `AOption`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Automatik_Toggle_Button



• – ○ • – ○ • – ○ •

Introduction

`Automatik_Toggle_Button` is an **automatic/manual toggle**: a VT button (or softkey or AUX function) toggles the automatic state. The state is output as an AX plug (`AUTOMATIK`), as its inverse (`AX_MAN`, e.g. for the `MAN` input of a PID controller) and as a local publish (`ID_AUTOMATIK_STATE_WRITE`) for the VT status and web dashboard. The SubApp is reusable for any automatic function (oil cooler, pickup, ...) and was extracted from `Aufnahme_Automatik.SUB`.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request
- **FORCE_ON**: Switches the automatic hard ON (e.g. from a superordinate automatic or a method call). Runs via `AX_E_PERMIT_2` and is therefore `PERMIT`-filtered. Leave unconnected if not needed.
- **FORCE_OFF**: Switches the automatic hard OFF (e.g. by a collision protection). Leave unconnected if not needed.
- **AUS**: Hard reset to manual/OFF from outside as an event (e.g. a centrally read common softkey), in addition to the block's own `u16ObjId_RESET`/`u16ObjIdA_RESET`/`ID_RESET_METHOD` path. Leave unconnected if not needed.

Event Outputs

- **INITO** (`EInit`): initialization confirm
- **AUTOMATIK_SET_FROM_VT**: Fires only on a click (button/softkey/aux, `PERMIT`-filtered) that switches the automatic from OFF to ON. A click that switches the automatic off does not fire this event. It is a genuine display click event, not a state level or republish.

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, TRUE enables the service
- **u16ObjId_Automatik_Toggle** (`UINT`, default `ID_NULL`): Object ID of the VT softkey (automatic/manual toggle)
- **u16ObjIdA_Automatik_Toggle** (`UINT`, default `ID_NULL`): Object ID of the AuxFunction2 (joystick), equivalent to the VT button; optional

- **u16ObjIdB_Automatik_Toggle** (UINT): Object ID of the VT button (automatic/manual toggle)
- **bDefaultAutomatik** (BOOL, default FALSE): Automatic state at start
- **ID_AUTOMATIK_STATE_WRITE** (WSTRING): Local publish address (BOOL, ACTION=WRITE) for the automatic on/off state (status text on the VT, web client/dashboard)
- **u16ObjId_RESET** (UINT, default ID_NULL): Object ID of the softkey "All off" (reset, hard to manual/OFF), e.g. SoftKey_AUS
- **u16ObjIdA_RESET** (UINT, default ID_NULL): Object ID of the AuxFunction2 (joystick) for the reset
- **ID_RESET_METHOD** (WSTRING): Local method address (ACTION=CREATE_METHOD) for the argument-less reset trigger; called by the web client and the OPC dashboard

Data Outputs

- **QO** (BOOL): quality output, AND combination (AND_QO) of the QO outputs of E_T_FF_Automatik, AX_PUBLISH_Automatik, Background_Automatik, Reset and Button_Softkey_Aux_Automatik_Toggle

Adapters

- **PERMIT** (socket, adapter::types::unidirectional::AX): Enable. Gates the entire on/off; on the transition 1⟶0 a one-time RESET is triggered.
- **AUTOMATIK** (plug, AX): Automatic active (raw, e.g. for an enable AND combination)
- **AX_MAN** (plug, AX): NOT automatic active (for the MAN input of a PID controller)

Function Blocks (FBs) Used

- **Button_Softkey_Aux_Automatik_Toggle**: SubApp MyLib::sys::Button_Softkey_Aux_IE - merges button, softkey and aux into one IND
- **Reset**: SubApp MyLib::sys::Softkey_Aux_Method_IE - reset via softkey, aux or method call
- **E_T_FF_Automatik**: AX_T_FF_SR_SYM_INIT (QI from the wrapper input) - stores the automatic state, start value bDefaultAutomatik
- **AX_E_PERMIT_2**: AX_E_PERMIT_2 - lets FORCE_ON and the click through only when PERMIT=TRUE
- **AX_SPLIT_2**: distributes PERMIT to AX_E_PERMIT_2 and E_F_TRIG
- **E_F_TRIG**: AX_F_TRIG - falling edge on PERMIT
- **E_SWITCH**: AX_E_SWITCH - filters the click by the state before the click
- **AX_SPLIT_4_Automatik**: AX_SPLIT_5 - distributes the state to five targets
- **AX_NOT_Man**: AX_NOT_INIT - inverse for AX_MAN
- **AX_PUBLISH_Automatik**: AX_PUBLISH_1 (QI from the wrapper input) - publishes the state under ID_AUTOMATIK_STATE_WRITE
- **Background_Automatik**: GreenWhiteBackground4_AX - colors button and AUX green/white
- **AND_QO**: iec61131::booleanOperators::AND_BOOL_5 - combines the QO outputs of the five blocks E_T_FF_Automatik, AX_PUBLISH_Automatik, Background_Automatik, Reset and Button_Softkey_Aux_Automatik_Toggle into QO

Program Flow and Connections

1. Button_Softkey_Aux_Automatik_Toggle.IND ⟶ AX_E_PERMIT_2.EI2; FORCE_ON ⟶ AX_E_PERMIT_2.EI1.
2. AX_E_PERMIT_2.EO1 ⟶ E_T_FF_Automatik.S (switch-on via FORCE_ON).
3. AX_E_PERMIT_2.EO2 ⟶ E_SWITCH.EI; E_SWITCH.G receives the state Q before the click (AX_SPLIT_4_Automatik.OUT5).
   - E_SWITCH.EO0 (OFF ⟶ ON) ⟶ AUTOMATIK_SET_FROM_VT and E_T_FF_Automatik.S
   - E_SWITCH.EO1 (ON ⟶ OFF) ⟶ E_T_FF_Automatik.R
4. Reset.IND, AUS, FORCE_OFF and E_F_TRIG.EO ⟶ E_T_FF_Automatik.R.
5. PERMIT ⟶ AX_SPLIT_2 ⟶ AX_E_PERMIT_2.PERMIT and E_F_TRIG.QI.
6. E_T_FF_Automatik.Q ⟶ AX_SPLIT_4_Automatik: OUT1 ⟶ AUTOMATIK, OUT2 ⟶ AX_NOT_Man ⟶ AX_MAN, OUT3 ⟶ AX_PUBLISH_Automatik, OUT4 ⟶ Background_Automatik.DI1, OUT5 ⟶ E_SWITCH.G.
7. Initialization chain: INIT ⟶ E_T_FF_Automatik.INIT ⟶ AX_PUBLISH_Automatik.INIT ⟶ Background_Automatik.INIT ⟶ Reset.INIT ⟶ Button_Softkey_Aux_Automatik_Toggle.INIT; Button_Softkey_Aux_Automatik_Toggle.INITO ⟶ INITO and ⟶ AND_QO.REQ.
8. Quality: QI ⟶ .QI of E_T_FF_Automatik, AX_PUBLISH_Automatik, Background_Automatik, Reset and Button_Softkey_Aux_Automatik_Toggle; their QO ⟶ AND_QO.IN1 to AND_QO.IN5 (in that order); AND_QO.OUT ⟶ QO.

Technical Details

- **Button instead of softkey**: The VT button must be non-latching (`Latchable=0`); the feedback of the automatic display is provided by `Background_Automatik`.
- **PERMIT** gates the entire on/off and resets once on the transition 1⟶0 - it thereby acts like an `AX_PERMIT`.
- `FORCE_ON`/`FORCE_OFF`/`AUS` are genuine events; callers with level-based sources must provide the edge detection themselves.
- `AUTOMATIK_SET_FROM_VT` reacts only to the display click OFF ⟶ ON, not to `FORCE_ON` echoes.

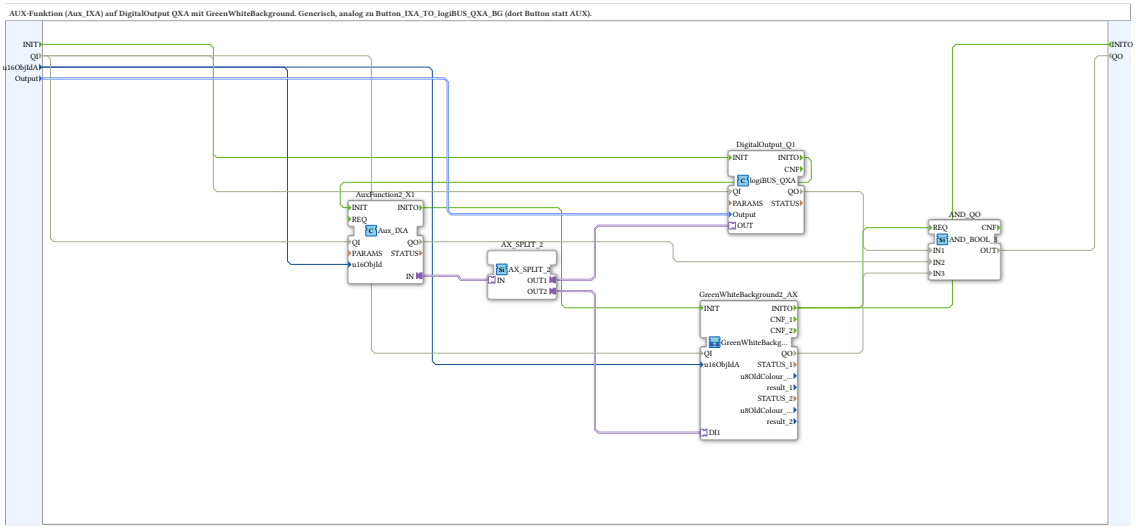
Application Scenarios

- Automatic/manual switching with status display for automatic functions (oil cooler, pickup, bunker harvest, ...)
- Manual mode signal for PID controllers (`AX_MAN`)

Summary

`Automatik_Toggle_Button` encapsulates operation, state storage, enable, reset and status output of an automatic function in one reusable SubApp.

AuxIXA_TO_logiBUS_QXA_BG





• – ○ • – ○ • – ○ •

Introduction

The `AuxIXA_TO_logiBUS_QXA_BG` subapplication is a generic, reusable composite function block that connects an AUX (auxiliary) input event from the ISOBUS UT system to a digital output on a logiBUS QXA module. It also incorporates a green/white background color control for visual indication. The subapp is designed to be instantiated wherever an AUX input needs to drive a digital output with color feedback, following the same pattern as the `Button_IXA_TO_logiBUS_QXA_BG` but using an AUX function block instead of a button.

Interface Structure

The subapplication exposes the initialization events `INIT`/`INITO`, the status pair `QI`/`QO` and data inputs; no adapters are visible at its boundary.

Event Inputs

- **INIT** (`EInit`): Initialization Request - starts the initialization chain of the internal blocks.

Event Outputs

- **INITO** (EInit): Initialization Confirm - emitted after the initialization chain has completed.

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	TRUE	Quality Input - TRUE to enable the service
u16ObjIdA	UINT	ID_NULL	Object ID of the AUX input
Output	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Selects the physical output (Q1..Q8)

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if the service executed successfully

QO is the AND (AND_QO, type iec61131::booleanOperators::AND_BOOL_3) of the QO outputs of DigitalOutput_Q1, AuxFunction2_X1 and GreenWhiteBackground2_AX.

Adapters

The subapplication does not expose any adapters; all internal adapter connections are encapsulated.

Functionality

The subapplication receives an AUX object ID (u16ObjIdA) and an output selection (Output). Internally, it performs the following steps:

1. The AuxFunction2_X1 block (type isobus::UT::io::Auxiliary::IN::Aux_IXA) receives QI from the subapplication's QI input and processes the AUX input based on the provided object ID. It generates an event on its IN adapter when the AUX input is activated.
2. The AX_SPLIT_2 block (type adapter::events::unidirectional::AX_SPLIT_2) splits the incoming event into two parallel event connections.
3. One event output (OUT1) goes to the DigitalOutput_Q1 block (type logiBUS::io::DQ::logiBUS_QXA), which drives the selected logiBUS digital output (based on the Output data input).
4. The other event output (OUT2) goes to the GreenWhiteBackground2_AX subapplication, which presumably handles the green/white background color logic for the display.
5. The u16ObjIdA is also connected to GreenWhiteBackground2_AX to keep the color logic synchronized with the AUX object.

Initialization and status: INIT → DigitalOutput_Q1.INIT, DigitalOutput_Q1.INITO → AuxFunction2_X1.INIT, AuxFunction2_X1.INITO → GreenWhiteBackground2_AX.INIT, GreenWhiteBackground2_AX.INITO → INITO (and additionally → AND_QO.REQ). QI is forwarded to DigitalOutput_Q1, AuxFunction2_X1 and GreenWhiteBackground2_AX; their QO outputs go to AND_QO.IN1..IN3, and AND_QO.OUT → QO.

The subapp thus provides a complete mapping from an AUX input to a physical digital output with visual feedback on the user interface.

Technical Features

- **Reusability**: The subapp is generic and can be used multiple times in a project, only needing the object ID and output selector to be configured.
- **Event-driven**: Uses adapter-based event propagation for efficient and clear signal flow.
- **Separation of concerns**: The AUX input handling, event splitting, and color logic are modular and can be replaced or extended individually.
- **Compatibility**: Internally uses standard ISOBUS and logiBUS function blocks, ensuring interoperability.

State Overview

Although the subapp does not define an explicit state machine, the internal blocks may have their own states. The AUX function block likely includes filtering and debouncing mechanisms, while the digital

output block handles energizing/de-energizing of the physical output. The background color subapplication manages visual states (green/white) based on the AUX activity.

Application Scenarios

- **Agricultural machinery**: Mapping AUX buttons to digital outputs (e.g., valve control, lighting) with simultaneous color indication on the in-cab display.
- **Generic ISOBUS UT implementations**: Wherever a simple AUX-to-output conversion with green/white feedback is required.
- **Rapid prototyping**: The subapp can be dropped into a network and configured without needing to redesign the logic.

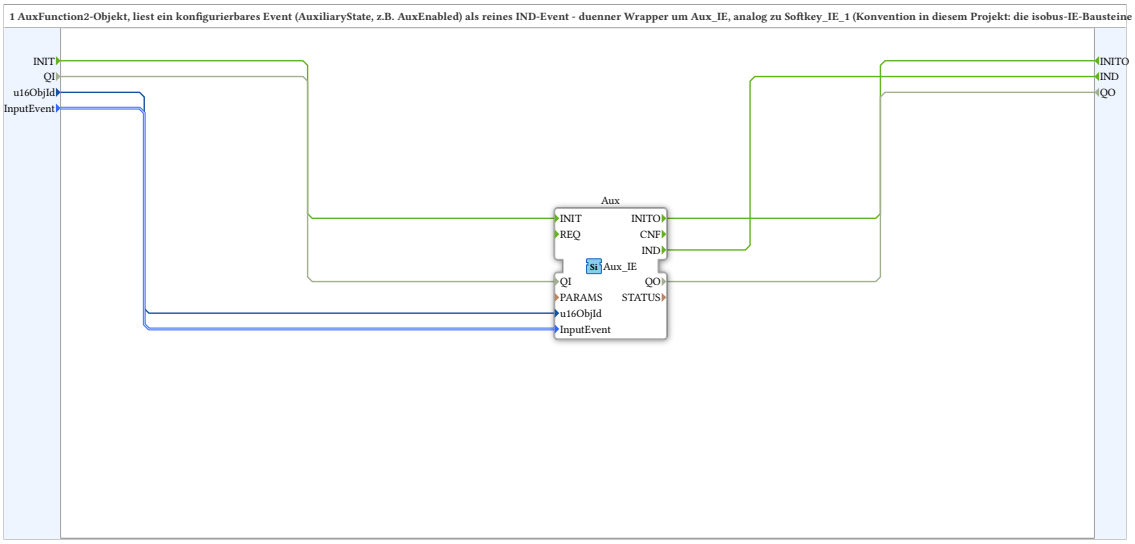
Comparison with Similar Blocks

- **Button_IXA_TO_logiBUS_QXA_BG**: Similar structure but uses a button input instead of an AUX object. This subapp replaces the button block with `Aux_IXA`.
- **Direct connection to logiBUS_DO**: A simpler approach would directly connect an event to the logiBUS output, but without color feedback and without the flexibility of an object ID mapping.

Conclusion

The `AuxIXA_TO_logiBUS_QXA_BG` subapplication provides a clean, reusable solution for integrating AUX inputs with digital outputs and visual feedback in ISOBUS-based systems. Its modular architecture makes it easy to maintain and adapt for various use cases, ensuring consistent behavior across different machines.

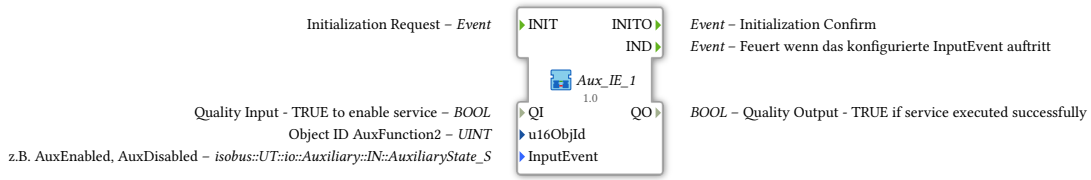
Aux_IE_1



• – ○ • – ○ • – ○ •

Introduction

`Aux_IE_1` reads an AuxFunction2 object and outputs a configurable event (`AuxiliaryState`, e.g. `AuxEnabled`) as a pure `IND` event. The block is a thin wrapper around `isobus::UT::io::Auxiliary::IN::Aux_IE`, analogous to `Softkey_IE_1` and `Button_IE_1`. Convention in this project: the isobus IE blocks are never used raw inline but always wrapped.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; passed on to `Aux.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `Aux.INITO`
- **IND**: fires when the configured `InputEvent` occurs

Data Inputs

- **QI** (BOOL): quality input, service enable (initial value `TRUE`), passed on to `Aux.QI`
- **u16ObjId** (UINT): object ID of the AuxFunction2 (default `ID_NULL`)
- **InputEvent** (`isobus::UT::io::Auxiliary::IN::AuxiliaryState_S`): e.g. `AuxEnabled`, `AuxDisabled` (default `Invalid`)

Data Outputs

- **QO** (BOOL): quality output, comes from `Aux.QO`

Adapters

No adapters are used.

Functionality

The network contains exactly one instance `Aux` of type `isobus::UT::io::Auxiliary::IN::Aux_IE`. `u16ObjId` and `InputEvent` are routed unchanged to the like-named inputs of the instance, and its event `IND` is output directly as the `IND` of the SubApp. In addition, `INIT` ⟶ `Aux.INIT`, `Aux.INITO` ⟶ `INITO`, `QI` ⟶ `Aux.QI` and `Aux.QO` ⟶ `QO` are passed through.

Technical Details

- Pure pass-through without logic or state of its own
- `QI` is no longer hard-wired to TRUE; it is passed through together with `INIT`/`INITO` and `QO` (the initial value of `QI` is `TRUE`)
- Because `InputEvent` is a parameter, the same SubApp can be used for different kinds of events (e.g. pressing or releasing)

Application Scenarios

- Reading a single VT operation as an event without using the raw IE block directly in the network

⚖️ Comparison with Similar Blocks

Equivalent wrappers are `Button_IE_1` and `Softkey_IE_1` they differ only in the VT object type and in the enumeration type of `InputEvent`.

Conclusion

`Aux_IE_1` is a minimal wrapper that provides a configurable AuxFunction2 event as `IND`.

Background Color Blocks: Shared Pattern

• – ◦ • – ◦ • – ◦ •

Introduction

`MyLib::sys` contains, in both training systems (`test_AX` and `test_B`), a large family of nearly identical blocks that **switch the VT background color of one or more objects based on a single boolean signal** — e.g. `GreenWhiteBackground1_AX`, `GreenRedBackground4_AXS`, `RedWhiteBackground2_AXC`. This page explains the shared pattern once in full; the individual block pages link back here and only state the concrete differences (color pair, object count, variant).

Naming Scheme

`<Color1><Color2>Background<N>[_aux][_AX][S][C]`

Component	Meaning

Color pair (`GreenWhite`, `GreenRed`, `GreenBlue`, `RedGreen`, `RedWhite`)	`TRUE` → first color, `FALSE` → second color (e.g. `GreenWhiteBackground`: TRUE→green, FALSE→white)
N (1–4)	Number of VT objects colored **simultaneously from the same selector bit** (not: number of independent channels)
`_aux` (N=1 only)	Uses `Q_BackgroundColourAux` instead of `Q_BackgroundColour` — targets an ISOBUS **Auxiliary Function** object instead of a regular softkey/button/data-mask object. For N≥2 variants, this role may already be baked into one specific position (see the individual page).
`_AX` (test_AX only)	The selector signal arrives via an `AX` adapter socket (`DI1`), not as a plain data input. In test_B this suffix is absent — there, `DI1` is a plain `BOOL` data input.
`S`	Object ID(s) are passed via the structured type `isobus::UT::Q::types::s1ObjectID` (`u16ObjIds`, unpacked internally via `F_MOVE`) instead of individual `UINT u16ObjId` inputs — a later unification of object-ID passing in the library.
`C` ("Compact")	A thin wrapper that just instantiates the base variant (`_AX` or `_AXS`) and doesn't expose the `CNF` event outputs/intermediate values (`STATUS_n`, `u8OldColour_n`, `result_n`) — for the common case where those diagnostic values aren't needed.

Examples: `GreenWhiteBackground1_AX` (base, 1 object, adapter), `GreenWhiteBackground1_AXC` (compact wrapper of it), `GreenWhiteBackground1_AXS` (with struct object ID), `GreenWhiteBackground1_AXSC` (struct ID + compact), `GreenWhiteBackground1_aux_AX` (like base, but for an AUX-function object).

Functionality (base variant, N=1)

1. The selector signal (`DI1` adapter for `_AX`, otherwise `DI1` data value) goes to `AX_SEL`/`F_SEL` (binary selection), parameterized with the two color constants (`IN0`=second color, `IN1`=first color, e.g. `IN0=COLOR_WHITE`, `IN1=COLOR_GREEN`).
2. `AX_SEL.CNF` triggers `Q_BackgroundColour.REQ` (or `Q_BackgroundColourAux.REQ` for `_aux`).
3. `Q_BackgroundColour` sets the background color of the VT object identified by `u16ObjId` and reports back `CNF` with `STATUS`, `u8OldColour` (previous color), and `s16result` (error code).

Functionality (N≥2)

With multiple objects (`GreenRedBackground4_AX` and similar), there is still **only one** selector (`DI1`/`AX_SEL`), whose output is distributed in parallel to several `Q_BackgroundColour_n` instances (numbered `_1`..`_N`) — each with its own `u16ObjId`/`u16ObjIdA`/`u16ObjIdB` (different object IDs, sometimes for different object roles like softkey/AUX/button) and its own `STATUS_n`/`u8OldColour_n`/`result_n`/`CNF_n` outputs. **Which position uses `Q_BackgroundColourAux` instead of `Q_BackgroundColour` varies per block** (not always the same position) — each block's individual page states the concrete mapping.

Technical Details

- **One selector, multiple destinations**: the point of N>1 is to apply the same boolean condition (e.g. "channel active") simultaneously to several VT representations of the same logical state (e.g. softkey background AND auxiliary-function background AND button background for the same function).
- **`_aux`/`Q_BackgroundColourAux`**: ISOBUS auxiliary function objects (controls freely assignable by the driver) require their own Q-block, since they use a different object ID range/VT message than regular softkeys/buttons.

- **`S` variant as a later unification**: identifiable by the later `VersionInfo` dates (2026 vs. 2022) — the `S` variants are a follow-up refactor that aligns object-ID passing with a later-established standard (`s1ObjectID`), without replacing the older base variants.

Family Overview

Color pair	test_AX (base/_aux, each also with S/SC/C)	test_B (base/_aux, each also with S/SC/C)
Green/White	1, 2, 3, 4	1, 2, 3, 4
Green/Red	1, 2, 3, 4	1, 2, 3, 4
Green/Blue	1	1
Red/Green	1, 4	1, 4
Red/White	1, 2, 3, 4	1, 2, 3, 4

In test_AX every combination additionally carries the `_AX` suffix; in test_B it's absent (selector is a `BOOL` data input instead of an adapter socket) — see MyLib (test_B) → sys.

Summary

The background color blocks are a single recurring idea (boolean signal → color selection → `Q_BackgroundColour`) offered in many small variants for different color pairs, object counts, and object-ID conventions. Understand one variant and you understand them all — the individual pages only state the concrete parameters.

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Background Color Blocks: Common Pattern

• – ◦ • – ◦ • – ◦ •

Introduction

In both training systems (`test_AX` and `test_B`), `MyLib::sys` contains a large family of nearly identical blocks that **toggle the VT background color of one or more objects based on a single Boolean signal** — e.g., `GreenWhiteBackground1_AX`, `GreenRedBackground4_AXS`, `RedWhiteBackground2_AXC`. This page explains the common pattern in full; the individual block pages refer back to this page and only mention the specific differences (color pair, number of objects, variant).

Naming Scheme

`<Farbe1><Farbe2>Background<N>[_aux][_AX][S][C]`

Component	Meaning
Color Pair (`GreenWhite`, `GreenRed`, `GreenBlue`, `RedGreen`, `RedWhite`)	`TRUE` → first color, `FALSE` → second color (e.g., `GreenWhiteBackground`: TRUE→Green, FALSE→White)
N (1–4)	Number of VT objects that are **simultaneously colored from the same selector bit** (not: number of independent channels)
`_aux` (only for N=1)	Uses `Q_BackgroundColourAux` instead of `Q_BackgroundColour` — targets an ISOBUS **Auxiliary Function** object instead of a normal softkey/button/data mask object. For N≥2 variants, this role may already be permanently integrated into one of the positions (see individual page).
`_AX` (only for test_AX)	The selector signal arrives via a `AX` adapter socket (`DI1`), not as a pure data input. This suffix is miss-

	ing in test_B — there, `DI1` is a simple `BOOL` data input.
`S`	Object ID(s) are passed via the structured type `isobus::UT::Q::types::s1ObjectID` (`u16ObjIds`, internally unpacked via `F_MOVE`) instead of via individual `UINT u16ObjId` inputs — a later standardization of object ID passing in the library.
`C` ("Compact")	A thin wrapper that instantiates only the base variant (`_AX` or `_AXS`) and does not expose the `CNF` event outputs/intermediate values (`STATUS_n`, `u8OldColour_n`, `result_n`) – for the typical case where these diagnostic values are not needed.

Examples: `GreenWhiteBackground1_AX` (Basic, 1 object, adapter), `GreenWhiteBackground1_AXC` (Compact wrapper of this), `GreenWhiteBackground1_AXS` (with struct object ID), `GreenWhiteBackground1_AXSC` (struct ID + compact), `GreenWhiteBackground1_aux_AX` (like Basic, but with an AUX function object).

Functionality (Basic Variant, N=1)

1. The selector signal (`DI1` adapter for `_AX`, otherwise `DI1` data value) goes to `AX_SEL`/`F_SEL` (binary selection), parameterized with the two color constants (`IN0` = second color, `IN1` = first color, e.g., `IN0=COLOR_WHITE`, `IN1=COLOR_GREEN`).
2. `AX_SEL.CNF`/`F_SEL.CNF` triggers `Q_BackgroundColour.REQ` (or `Q_BackgroundColourAux.REQ` for `_aux`).
3. `Q_BackgroundColour` sets the background color of the VT object identified by `u16ObjId` and returns `CNF` with `STATUS`, `u8OldColour` (previous color), and `s16result` (error code).

Functionality (N≥2)

For multiple objects (`GreenRedBackground4_AX` etc.), there is still only **one** selector (`DI1`/`AX_SEL`/`F_SEL`), whose output is distributed in parallel to multiple `Q_BackgroundColour_n` instances (numbered `_1`..`_N`) — each with its own `u16ObjId`/`u16ObjIdA`/`u16ObjIdB` (different object IDs, sometimes for different object roles such as softkey/AUX/button) and its own `STATUS_n`/`u8OldColour_n`/`result_n`/`CNF_n` outputs. **Which position uses `Q_BackgroundColourAux` instead of `Q_BackgroundColour` varies per block** (not always the same position) — the individual page for each block specifies the exact assignment.

Technical Features

- **One selector, multiple Objectives**: The purpose of N>1 is to apply the same Boolean condition (e.g., "Channel active") simultaneously to multiple VT representations of the same logical state (e.g., softkey background AND auxiliary function background AND button background of the same function).
- **`_aux`/`Q_BackgroundColourAux`**: ISOBUS auxiliary function objects (controls that can be freely assigned to the driver) require their own Q block because they use a different ObjectID range/VT message than normal softkeys/buttons.
- **`S` variant as a later unification**: Recognizable by the later `VersionInfo` dates (2026 vs. 2022) — the `S` variants are a follow-up refactoring that aligns object ID passing with a later-established standard (`s1ObjectID`), without replacing the older base variants.

Family Overview

Color Pair	test_AX (Base/_aux, also S/SC/C)	test_B (Base/_aux, also S/SC/C)
Green/White	1, 2, 3, 4	1, 2, 3, 4
Green/Red	1, 2, 3, 4	1, 2, 3, 4
Green/Blue	1	1
Red/Green	1, 4	1, 4
Red/White	1, 2, 3, 4	1, 2, 3, 4

In test_AX, each combination also carries the suffix `_AX`; in test_B, this suffix is omitted (selector as `BOOL` data input instead of adapter socket) — see MyLib (test_B) → sys.

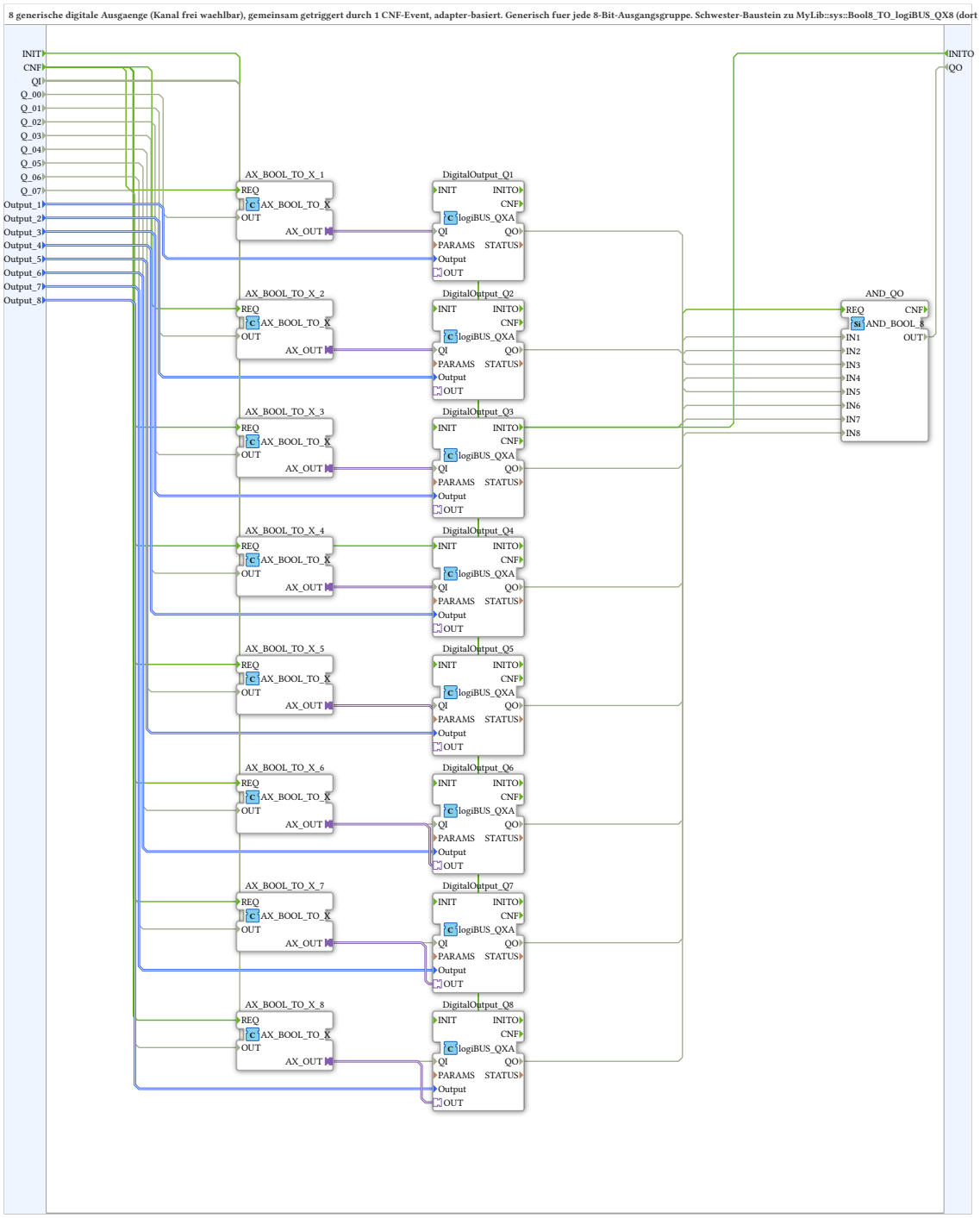
Summary

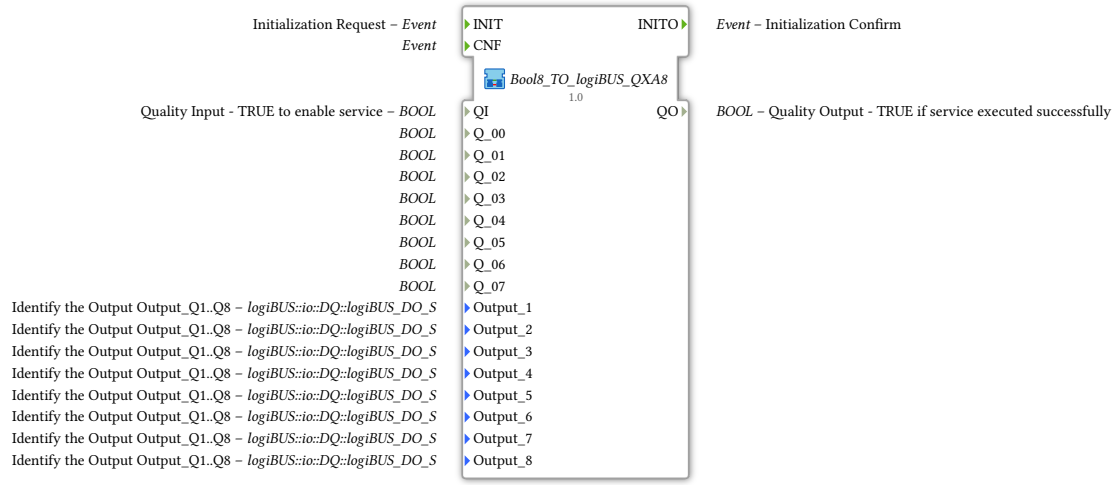
The background color blocks are based on a single, recurring concept (Boolean signal → color selection → `Q_BackgroundColour`) with many minor variations for different color pairs, object counts, and object ID conventions. Understanding one variation makes them all understandable—the individual pages simply list the specific parameters.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Bool8_TO_logiBUS_QXA8





• – ○ • – ○ • – ○ •

Introduction

The `Bool8_TO_logiBUS_QXA8` is a generic composite function block (subapplication) designed to convert eight Boolean input signals into eight digital output channels of a logiBUS system. It provides a flexible, adapter-based solution where each output channel can be freely assigned to any of the eight Boolean inputs via configuration parameters. All outputs are triggered simultaneously by a single event, ensuring synchronized updates across the entire group.

The block is particularly useful in control applications that require multiple digital outputs (e.g., relays, LEDs, solenoid valves) to be driven from a central logic controller, with the ability to remap outputs without changing the block's internal structure.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the init chain at `DigitalOutput_Q4.INIT`.
- **CNF** (Event): Common trigger event. When this event arrives, all eight Boolean inputs are read and their corresponding digital outputs are updated in the same processing cycle.

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `DigitalOutput_Q3.INITO`, the end of the init chain.

Data Inputs

- **QI** (BOOL, initial value `TRUE`): quality input, enables the service; passed on to all eight `DigitalOutput_Qx.QI`.
- **Q_00** … **Q_07** (BOOL): These eight Boolean inputs carry the logic values that are to be written to the selected output channels.
- **Output_1** … **Output_8** (logiBUS::io::DQ::logiBUS_DO_S): Channel selectors. Each parameter identifies the physical output channel (e.g., "Output_Q1" … "Output_Q8") to which the associated Boolean input (Q_00 ⟶ Output_1, Q_01 ⟶ Output_2, …) is routed. The value must be a valid logiBUS output identifier; setting it to `logiBUS_DO::Invalid` disables that output.

Data Outputs

- **QO** (BOOL): quality output, AND combination (`AND_QO`, `AND_BOOL_8`) of the `QO` outputs of all eight `DigitalOutput_Qx`.

Adapters

- None exposed at the interface level. Adapter connections are internal to the subapplication, linking the Boolean conversion blocks to the output driver blocks.

Functionality

The subapplication internally instantiates eight pairs of function blocks:

- **AX_BOOL_TO_X_1** ... **AX_BOOL_TO_X_8** (type: adapter::conversion::unidirectional::AX_BOOL_TO_X): These convert each Boolean input into an adapter-compatible signal (unidirectional data plug).
- **DigitalOutput_Q1** ... **DigitalOutput_Q8** (type: logiBUS::io::DQ::logiBUS_QXA): These are the actual digital output driver blocks that communicate with the logiBUS hardware.

On each `CNF` event, all eight conversion blocks receive the event and simultaneously fetch their corresponding Boolean input values. Each converted value is then passed via an adapter connection to its associated digital output block. The digital output block uses the `Output_n` parameter to determine which physical channel to write to (or to ignore the value if the output is disabled).

Initialization and quality: `INIT` is chained through all eight `logiBUS_QXA` instances (`INIT` → `DigitalOutput_Q4` → `DigitalOutput_Q7` → `DigitalOutput_Q1` → `DigitalOutput_Q5` → `DigitalOutput_Q2` → `DigitalOutput_Q6` → `DigitalOutput_Q8` → `DigitalOutput_Q3`, each `INITO` → next `INIT`); `DigitalOutput_Q3.INITO` leads to `INITO` and `AND_QO.REQ`. `QI` is distributed to all eight `DigitalOutput_Qx.QI`; their `QO` outputs go to `AND_QO.IN1` to `AND_QO.IN8` (`iec61131::booleanOperators::AND_BOOL_8`), whose `OUT` forms the output `QO`.

Since all conversion blocks are triggered by the same event, the entire group of outputs is updated in a single, atomic operation, ensuring temporal consistency among the eight channels.

Technical Features

- **8-channel design**: Supports eight independent digital outputs.
- **Free channel mapping**: Each Boolean input can be routed to any available output channel using the `Output_n` parameters, allowing flexible reconfiguration without changing wiring.
- **Single-event triggering**: A common `CNF` event triggers all outputs simultaneously, simplifying synchronization and reducing event overhead.
- **Adapter-based conversion**: Utilizes unidirectional adapter conversions, promoting modularity and compatibility with various logiBUS output types.
- **Generic and reusable**: The block is designed generically for any 8-bit output group; it can be instantiated multiple times for different hardware configurations.

State Overview

This subapplication is combinational in nature; it does not maintain any internal state. Each activation of the `CNF` event directly maps the current Boolean inputs to the configured physical outputs. There is no feedback or memory function, and the block's behavior is completely deterministic based on the input values and channel selection parameters.

Application Scenarios

Typical use cases include:

- **Industrial control panels**: Driving eight relays or contactors from a PLC's logic, with the ability to assign each Boolean signal to a specific relay.
- **Lighting or LED groups**: Controlling multiple lamps individually via a central control system, where each lamp can be remapped to a different Boolean input as needed.
- **Factory automation**: Setting up quick-change output configurations for different manufacturing cells without modifying the underlying control logic.
- **Prototyping and testing**: Allowing developers to experiment with different output assignments in a simulation environment before deployment.

Comparison with Similar Blocks

The `Bool8_TO_logiBUS_QXA8` is the sister block of `MyLib::sys::Bool8_TO_logiBUS_QX8`. The main difference lies in the type of digital output driver used:

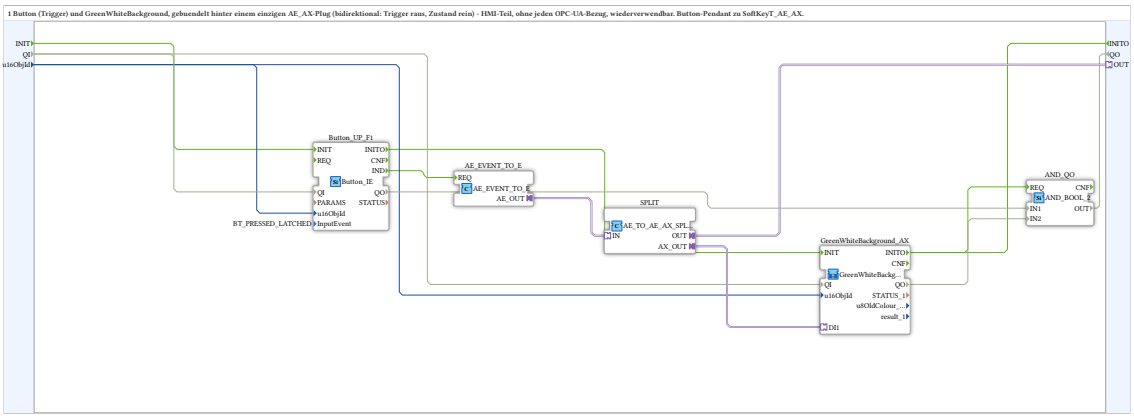
- **Bool8_TO_logiBUS_QX8** uses the `logiBUS_QX` block, which directly interfaces with logiBUS output modules without an adapter layer.
- **Bool8_TO_logiBUS_QXA8** uses the `logiBUS_QXA` block, which includes an adapter interface. This makes it more flexible when the hardware requires a specific adapter connection or when additional signal conditioning is needed.

Compared to other generic 8-output blocks, this subapplication offers the distinct advantage of per-output channel selection combined with a common trigger, whereas many simpler blocks either fix the output mapping or require separate events for each channel.

Conclusion

The `Bool8_TO_logiBUS_QXA8` subapplication provides a robust and flexible solution for driving eight digital outputs over a logiBUS network. Its design emphasizes modularity, ease of reconfiguration, and synchronized updates. By leveraging adapter-based conversions and a single trigger event, it simplifies complex output management tasks and can be seamlessly integrated into a wide range of industrial and automation projects. Its sister relationship with the `QXA`-free variant ensures that developers can choose the appropriate level of abstraction for their specific hardware requirements.

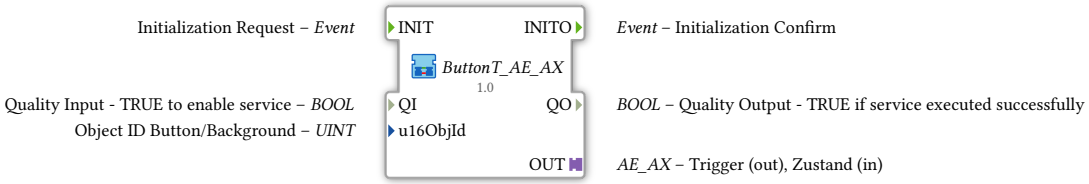
ButtonT_AE_AX



• – ○ • – ○ • – ○ •

Introduction

`ButtonT_AE_AX` bundles **one button (trigger)** and a `GreenWhiteBackground` behind a single bidirectional `AE_AX` plug (trigger out, state in). It is the pure HMI part without any OPC-UA reference and is reusable. The block is the button counterpart of `SoftKeyT_AE_AX`: `Softkey_IE` was replaced by `Button_IE`, `SK_RELEASED` by `BT_PRESSED_LATCHED`.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; goes to `Button_UP_F1.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `GreenWhiteBackground_AX.INITO`

Data Inputs

- **QI** (`BOOL`): quality input - TRUE enables the service (default `TRUE`); goes to `Button_UP_F1.QI` and `GreenWhiteBackground_AX.QI`
- **u16ObjId** (`UINT`): object ID of button/background (default `ID_NULL`)

Data Outputs

- **QO** (`BOOL`): quality output - TRUE if the service was executed successfully (AND of `Button_UP_F1.QO` and `GreenWhiteBackground_AX.QO`)

Adapters

- **OUT** (plug, `adapter::types::bidirectional::AE_AX`): trigger (out), state (in)

Function Blocks (FBs) Used

- **`Button_UP_F1`** (`isobus::UT::io::Button::Button_IE`): parameter `InputEvent` = `BT_PRESSED_LATCHED` (`QI` comes from outside)
- **`AE_EVENT_TO_E`** (`adapter::conversion::unidirectional::AE_EVENT_TO_E`): converts the button event into an `AE` adapter
- **`SPLIT`** (`adapter::events::bidirectional::AE_TO_AE_AX_SPLIT`): separates the bidirectional `AE_AX` plug into trigger direction and state direction
- **`GreenWhiteBackground_AX`** (SubApp `MyLib::sys::GreenWhiteBackground1_AX`): green/white background colour
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): combines the `QO` of `Button_UP_F1` and `GreenWhiteBackground_AX`

Functionality

- `Button_UP_F1.IND` ⟶ `AE_EVENT_TO_E.REQ`
- `AE_EVENT_TO_E.AE_OUT` ⟶ `SPLIT.IN`
- `SPLIT.OUT` ⟶ `OUT` (trigger to the outside, state from the outside)
- `SPLIT.AX_OUT` ⟶ `GreenWhiteBackground_AX.DI1` (the state colours the background)
- `u16ObjId` ⟶ `Button_UP_F1.u16ObjId` and `GreenWhiteBackground_AX.u16ObjId`
- Initialization: `INIT` ⟶ `Button_UP_F1.INIT`, `Button_UP_F1.INITO` ⟶ `GreenWhiteBackground_AX.INIT`, `GreenWhiteBackground_AX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`
- Quality: `QI` ⟶ `Button_UP_F1.QI` and `GreenWhiteBackground_AX.QI`; `Button_UP_F1.QO` ⟶ `AND_QO.IN1`, `GreenWhiteBackground_AX.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`

Technical Details

- The same object ID serves the button and the background colour
- `BT_PRESSED_LATCHED` instead of `SK_RELEASED` of the softkey counterpart
- No OPC-UA reference, so reusable in any scenario

Application Scenarios

- A button in the VT as a trigger whose state (e.g. from a toggle) is fed back as background colour, through a single `AE_AX` connection

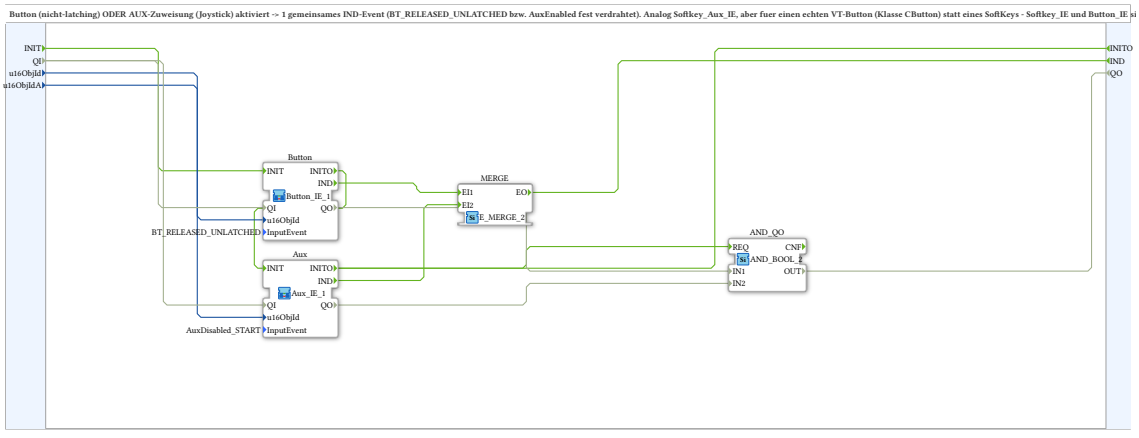
Summary

`ButtonT_AE_AX` bundles button trigger and state colour behind one `AE_AX` plug.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Button_Aux_IE

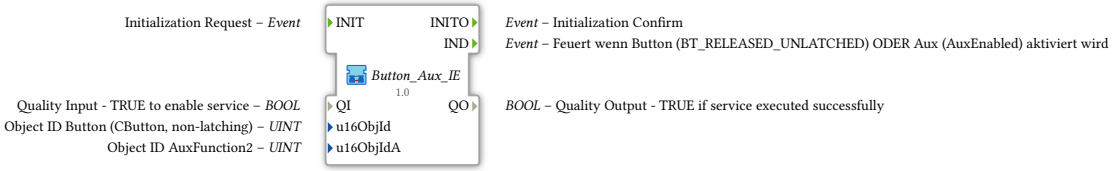


• — ○ • — ○ • — ○ •

Introduction

`Button_Aux_IE` combines a non-latching VT button **or** an AUX assignment (joystick) into one common `IND` event. The activation type is hard-wired (`BT_RELEASED_UNLATCHED` and `AuxEnabled` respectively). The block corresponds to `Softkey_Aux_IE`, but for a real VT button (class `CButton`) instead of a SoftKey: `Softkey_IE` and `Button_IE` are two separate FORTE service interface blocks for different VT activation messages; a button never delivers an `SK_PRESSED`.

The button **must** be non-latching (`Latchable=0` in the pool). A latching button can sensibly be read only via `Button_IXA` (continuous state/level), not via this event/`InputEvent` pattern.



Function Blocks (FBs) Used

Sub-blocks: Button_Aux_IE

- **Type**: SubAppType
- **Internal blocks used**:
  - **Button**: SubApp `Button_IE_1` with `InputEvent=BT_RELEASED_UNLATCHED`
  - **Aux**: SubApp `Aux_IE_1` with `InputEvent=AuxDisabled_START`
  - **MERGE**: `iec61499::events::E_MERGE_2` - merges both `IND` events
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` - combines `Button.QO` and `Aux.QO` into `QO`
- **Imports**: `BT_RELEASED_UNLATCHED` (`ButtonActivationCode`), `AuxDisabled_START` (`AuxiliaryState`)

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation request

Event Outputs

- **INITO** (`EInit`): initialisation confirmation
- **IND**: Fires when the button (`BT_RELEASED_UNLATCHED`) or the aux assignment (`AuxEnabled`) is activated

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **u16ObjId** (`UINT`): Object ID of the button (`CButton`, non-latching)
- **u16ObjIdA** (`UINT`): Object ID of the `AuxFunction2`

Data Outputs

- **QO** (`BOOL`): Quality Output - AND of `Button.QO` and `Aux.QO`

Program Flow and Connections

- `u16ObjId` ⟶ `Button.u16ObjId`
- `u16ObjIdA` ⟶ `Aux.u16ObjId`
- `Button.IND` ⟶ `MERGE.EI1`
- `Aux.IND` ⟶ `MERGE.EI2`
- `MERGE.EO` ⟶ `IND`
- `INIT` ⟶ `Button.INIT`; `Button.INITO` ⟶ `Aux.INIT`; `Aux.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`)
- `QI` ⟶ `Button.QI` and `QI` ⟶ `Aux.QI`
- `Button.QO` ⟶ `AND_QO.IN1`; `Aux.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technical Details

- Bugfix for `Automatik_Toggle_Button`: it previously used `Softkey_Aux_IE` with a button object ID, but `Softkey_IE` receives no button activation message (confirmed by the author of the FORTE implementation)
- The `E_MERGE_2` forwards the event whichever of the two sources fires
- Version 1.0 (2026-09-20)

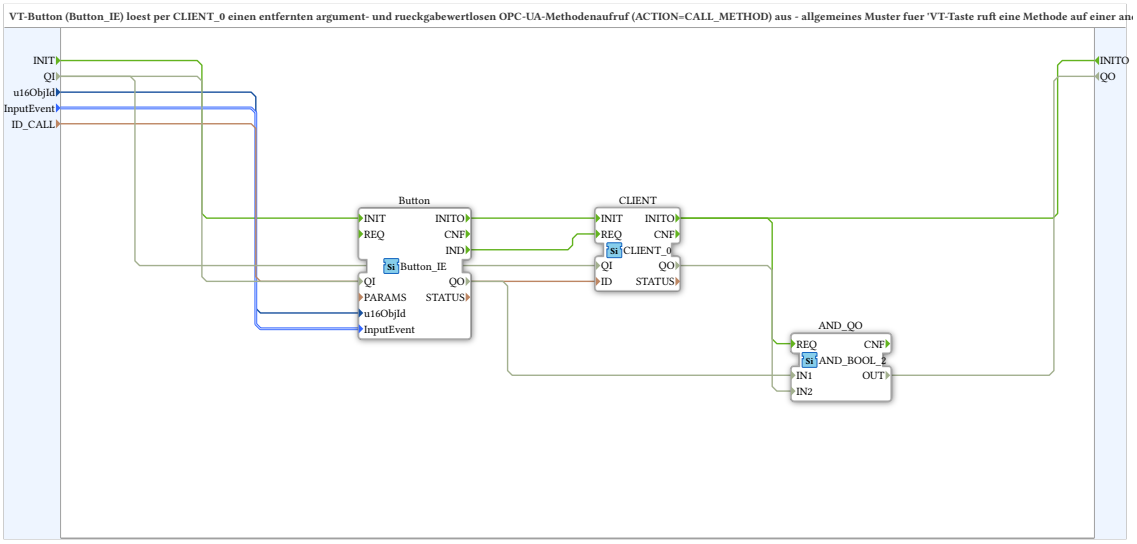
Application Scenarios

- A control element that is to be triggered either via a VT button or an AUX joystick input
- Toggle logic (e.g. `Automatik_Toggle_Button`) with two equivalent triggers

Summary

`Button_Aux_IE` combines `Button_IE_1` and `Aux_IE_1` via an `E_MERGE_2` into a single `IND` event for button and AUX.

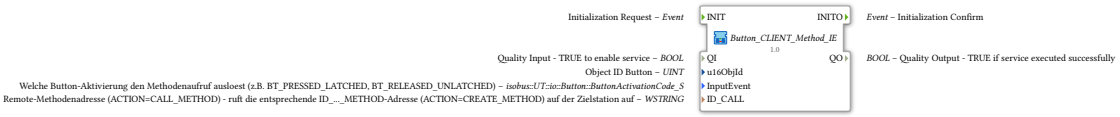
Button_CLIENT_Method_IE



• — ○ • — ○ • — ○ •

Introduction

`Button_CLIENT_Method_IE` uses a VT button (`Button_IE`) to trigger, via `CLIENT_0`, a remote **OPC-UA method call** without arguments and return value (`ACTION=CALL_METHOD`). It is the general pattern "VT key calls a method on another station" and was extracted from `logiBUS_AI_Calibrated_Subscribe_VT_OPC`, where it occurred twice identically. Unlike `Button_Method_IE` (button OR local method → common `IND`), this is the reverse "caller" case: button → remote method.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirm

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, TRUE enables the service
- **u16ObjId** (`UINT`): Object ID of the button
- **InputEvent** (`isobus::UT::io::Button::ButtonActivationCode_S`): Which button activation triggers the method call (e.g. `BT_PRESSED_LATCHED`, `BT_RELEASED_UNLATCHED`)
- **ID_CALL** (`WSTRING`): Remote method address (`ACTION=CALL_METHOD`); calls the corresponding `ID_..._METHOD` address (`ACTION=CREATE_METHOD`) on the target station

Data Outputs

- **QO** (`BOOL`): quality output, AND combination (`AND_QO`) of the `QO` outputs of `Button` and `CLIENT`

Adapters

This function block uses no adapters.

Function Blocks (FBs) Used

- **Button**: `isobus::UT::io::Button::Button_IE` (`QI` from the wrapper input)
- **CLIENT**: `iec61499::net::CLIENT_0` (`QI` from the wrapper input)
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` - combines `Button.QO` and `CLIENT.QO` into `QO`

Program Flow and Connections

- `u16ObjId` ⟶ `Button.u16ObjId`; `InputEvent` ⟶ `Button.InputEvent`
- `Button.IND` ⟶ `CLIENT.REQ`
- `ID_CALL` ⟶ `CLIENT.ID`
- Initialization: `INIT` ⟶ `Button.INIT`; `Button.INITO` ⟶ `CLIENT.INIT`; `CLIENT.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`
- Quality: `QI` ⟶ `Button.QI` and `CLIENT.QI`; `Button.QO` ⟶ `AND_QO.IN1`; `CLIENT.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

On every button activation matching `InputEvent`, the method call `CLIENT_0` is thus triggered.

Technical Details

- `InputEvent` is a parameter and not hard-coded to `BT_PRESSED_LATCHED`, so that other activation types (e.g. release) are reusable as well.
- The confirmation (`CNF`) of `CLIENT_0` is not exposed.

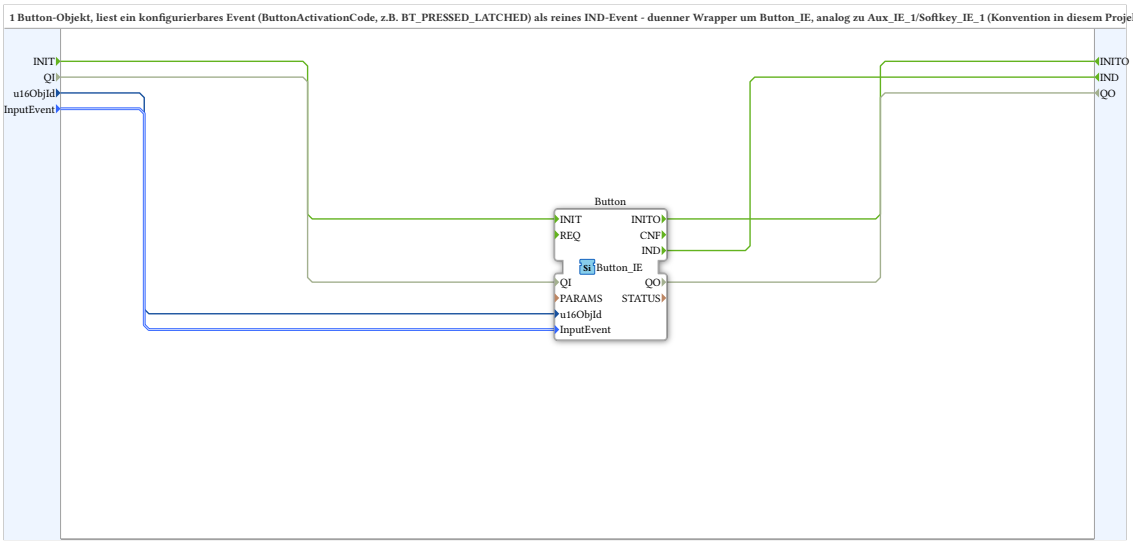
Application Scenarios

- VT key that calls a method on another station (e.g. calibration via OPC-UA)

Summary

`Button_CLIENT_Method_IE` is a small, generic pattern for calling a remote OPC-UA method via a VT key.

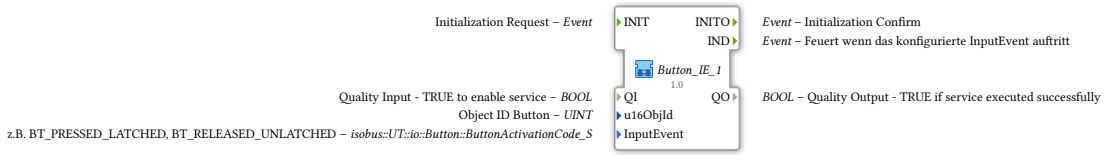
Button_IE_1



• – ◦ • – ◦ • – ◦ •

Introduction

`Button_IE_1` reads a button object and outputs a configurable event (`ButtonActivationCode`, e.g. `BT_PRESSED_LATCHED`) as a pure `IND` event. The block is a thin wrapper around `isobus::UT::io::Button::Button_IE`, analogous to `Aux_IE_1` and `Softkey_IE_1`. Convention in this project: the isobus IE blocks are never used raw inline but always wrapped.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization Request

Event Outputs

- **INITO** (`EInit`): Initialization Confirm
- **IND**: fires when the configured `InputEvent` occurs

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE to enable the service
- **u16ObjId** (UINT): object ID of the button (default `ID_NULL`)
- **InputEvent** (`isobus::UT::io::Button::ButtonActivationCode_S`): e.g. `BT_PRESSED_LATCHED`, `BT_RELEASED_UNLATCHED` (default `Invalid`)

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service executed successfully (passed through from `Button.QO`)

Adapters

No adapters are used.

Functionality

The network contains exactly one instance `Button` of type `isobus::UT::io::Button::Button_IE`. `u16ObjId`, `InputEvent` and `QI` are routed unchanged to the like-named inputs of the instance, `INIT` → `Button.INIT`; its events `IND` and `INITO` and its output `QO` are output directly as `IND`, `INITO` and `QO` of the SubApp.

Technical Details

- Pure pass-through without logic or state of its own
- `QI` is passed through from outside to the inner block (default `TRUE`)
- Because `InputEvent` is a parameter, the same SubApp can be used for different kinds of events (e.g. pressing or releasing)

Application Scenarios

- Reading a single VT operation as an event without using the raw IE block directly in the network

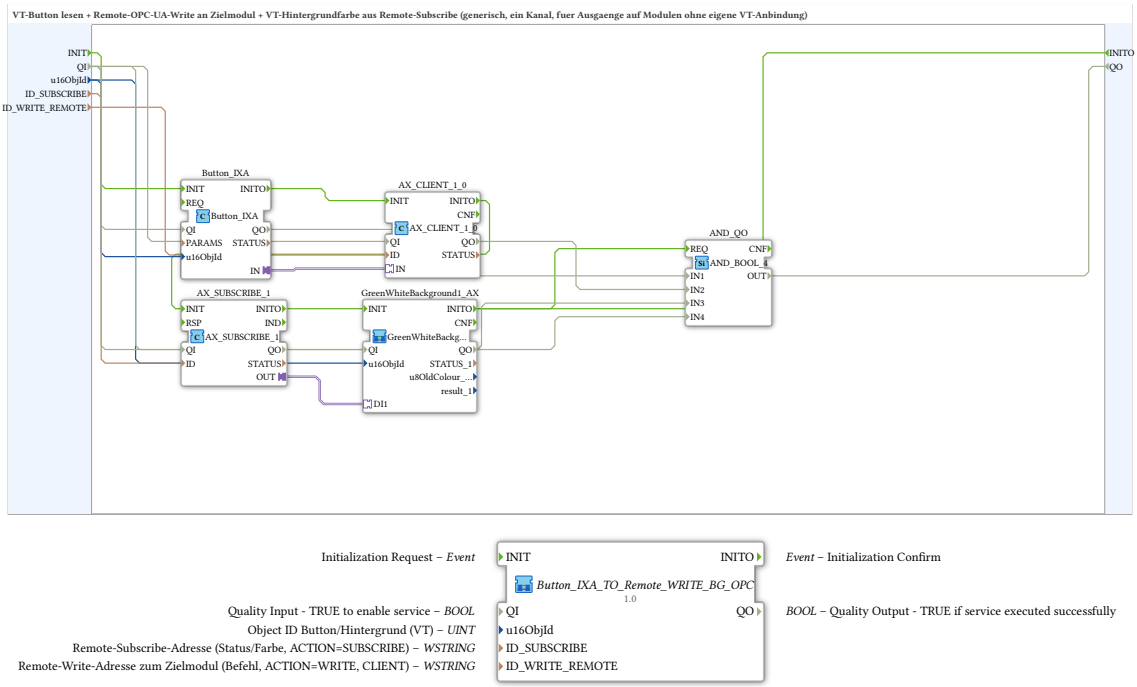
⚖️ Comparison with Similar Blocks

Equivalent wrappers are `Aux_IE_1` and `Softkey_IE_1` they differ only in the VT object type and in the enumeration type of `InputEvent`.

Conclusion

`Button_IE_1` is a minimal wrapper that provides a configurable button event as `IND`.

Button_IXA_TO_Remote_WRITE_BG_OPC



• – ○ • – ○ • – ○ •

Introduction

The **Button_IXA_TO_Remote_WRITE_BG_OPC** subapplication is a generic single-channel solution that combines three core functions:

- Reading a visual terminal (VT) button state via the `isobus::UT::io::Button::Button_IXA` function block.
- Writing the button state to a remote target module using an OPC-UA client adapter (`adapter::net::AX_CLIENT_1_0`).
- Updating a VT background colour based on a remote subscription (`adapter::net::AX_SUBSCRIBE_1`) that feeds a background-processing subapplication (`GreenWhiteBackground1_AX`).

It is intended for outputs on modules that do not have their own VT connection, allowing the VT button to both send commands and reflect remote status.

Interface Structure

The subapplication exposes the event input `INIT`, the event output `INITO`, the data inputs `QI`, `u16ObjId`, `ID_SUBSCRIBE` and `ID_WRITE_REMOTE`, and the data output `QO`; it has no external adapters. All internal communication is handled through data and adapter connections inside the subapplication.

Event Inputs

- `INIT` (EInit): initialization request; starts the chain `Button_IXA` ⟶ `AX_CLIENT_1_0` ⟶ `AX_SUBSCRIBE_1` ⟶ `GreenWhiteBackground1_AX`

Event Outputs

- `INITO` (EInit): initialization confirmation; comes from `GreenWhiteBackground1_AX.INITO`

Data Inputs

Name	Type	Initial Value	Comment
`QI`	BOOL	`TRUE`	Quality input, service enable
`u16ObjId`	UINT	`ID_NULL`	Object ID of the button and background (VT)

ID_SUBSCRIBE	WSTRING	–	Remote-subscribe address for status/colour (ACTION=SUBSCRIBE)
ID_WRITE_REMOTE	WSTRING	–	Remote-write address to the target module (ACTION=WRITE, CLIENT)

Data Outputs

Name	Type	Comment
QO	BOOL	Quality output, AND of the QO outputs of Button_IXA, AX_CLIENT_1_0, AX_SUBSCRIBE_1 and GreenWhiteBackground1_AX (via AND_QO)

Adapters

None at the subapplication interface level. Adapters are used internally for OPC-UA communication and subscription handling.

Functionality

The subapplication works as follows:

1. **Button Reading** – The Button_IXA block reads the state of a VT-associated button identified by u16ObjId. Its output is connected through an adapter to the OPC-UA client block.
2. **Remote Write** – The AX_CLIENT_1_0 block acts as an OPC-UA client. It receives the button state via an adapter connection from Button_IXA.IN and writes it to the remote address specified by ID_WRITE_REMOTE. This enables the button to control an output on a module without a direct VT link.
3. **Remote Subscribe** – The AX_SUBSCRIBE_1 block subscribes to a remote status/colour signal using the address given in ID_SUBSCRIBE. The subscribed data is passed via an adapter connection to the GreenWhiteBackground1_AX subapplication, which determines the appropriate VT background colour.
4. **Background Update** – The GreenWhiteBackground1_AX subapplication uses the received status (e.g., on/off, alarm) and the object ID to set the background colour of the same VT object that contains the button.

The data inputs u16ObjId, ID_SUBSCRIBE, ID_WRITE_REMOTE are distributed internally to the respective blocks, ensuring full decoupling between the button input, remote write, and remote subscribe paths.

1. **Initialization and Quality** – INIT → Button_IXA.INIT, Button_IXA.INITO → AX_CLIENT_1_0.INIT, AX_CLIENT_1_0.INITO → AX_SUBSCRIBE_1.INIT, AX_SUBSCRIBE_1.INITO → GreenWhiteBackground1_AX.INIT, GreenWhiteBackground1_AX.INITO → INITO and → AND_QO.REQ. QI goes to Button_IXA.QI, AX_CLIENT_1_0.QI, AX_SUBSCRIBE_1.QI and GreenWhiteBackground1_AX.QI; their four QO outputs feed AND_QO.IN1 to IN4 (iec61131::booleanOperators::AND_BOOL_4), and AND_QO.OUT drives QO.

Technical Features

- **Generic Design** – The subapplication is parameterised by three inputs, making it usable for any VT object, remote write target, and remote subscription source.
- **Integrated OPC-UA Client and Subscriber** – Employs AX_CLIENT_1_0 and AX_SUBSCRIBE_1 adapters, which provide standardised OPC-UA connectivity.
- **Built-in Button Handling** – Uses the Button_IXA block, which encapsulates VT button state detection.
- **Background Colour Logic** – Delegates colour computation to the reusable GreenWhiteBackground1_AX subapplication, promoting modularity.
- **Pass-through enable** – The internal function blocks are no longer fixed to QI = TRUE; they receive the subapplication input QI (initial value TRUE), and QO is the AND of all four internal QO outputs.
- **Input Types** – u16ObjId is a 16-bit unsigned integer, while the two address inputs are wide strings, accommodating full OPC-UA address strings.

State Overview

The subapplication does not maintain its own explicit state machine. Instead, its behaviour is driven by the states of the internal components:

- **Button State** – Determined by the Button_IXA block (pressed/released).

- **Remote Communication State** – The `AX_CLIENT_1_0` and `AX_SUBSCRIBE_1` blocks manage connection establishment, data transfer, and error handling.
- **Background Colour State** – Based on the subscribed value, the `GreenWhiteBackground1_AX` subapplication outputs a colour (e.g., green for running, white for idle, or red for alarm).

Because the logic is entirely distributed, the subapplication can be considered a stateless composition that reacts to its inputs and the current network conditions.

Application Scenarios

- **Remote Output Control without Local VT** – Use a VT pushbutton on a display panel to toggle an output on a remote I/O module that has no visual terminal of its own.
- **Status Indication via Colour** – Simultaneously, the same VT object's background changes colour (e.g., green = active, white = inactive) based on a subscription to the remote output's status.
- **Generic Single-Channel Expansion** – The subapplication can be duplicated for multiple channels, each with its own object ID, write address, and subscribe address, providing a scalable control panel.
- **Legacy Module Integration** – Modules that lack native VT binding can be controlled and monitored through OPC-UA while still being represented on a central VT screen.

Comparison with Similar Blocks

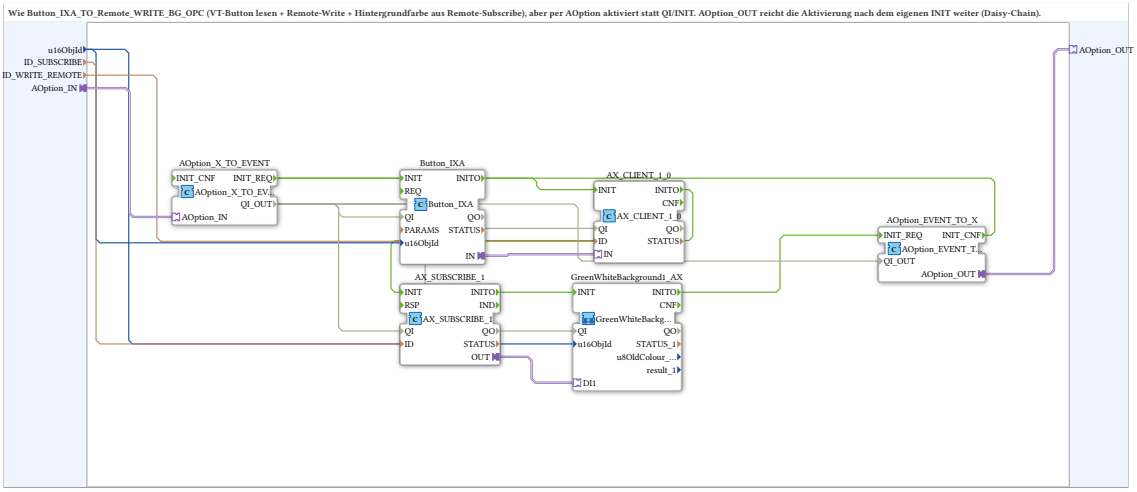
Compared to simpler button-to-remote-write blocks that only send the button state to a remote output, **Button_IXA_TO_Remote_WRITE_BG_OPC** adds the crucial background-colour subscription. This extra capability provides immediate visual feedback on the VT, eliminating the need for separate status polling.

In contrast to blocks that handle local output directly, this subapplication is specifically designed for remote destinations, offering OPC-UA connectivity out of the box. The inclusion of the `GreenWhiteBackground1_AX` subapplication also separates colour logic from communication logic, making the solution more maintainable than a monolithic block with hard-coded colour transitions.

Conclusion

The **Button_IXA_TO_Remote_WRITE_BG_OPC** subapplication provides a compact, reusable, and generic solution for controlling remote outputs via a VT button while visualising their status through a background colour. By combining an OPC-UA client for write operations, an OPC-UA subscriber for status updates, and a dedicated background-colour subapplication, it delivers a complete remote-control and status-indication package. Its simple interface (`INIT`/`INITO`, `QI`/`QO` and three data inputs) and clearly separated internal components make it easy to integrate into larger automation projects.

Button_IXA_TO_Remote_WRITE_BG_Option_OPC



• – ○ • – ○ • – ○ •

Introduction

`Button_IXA_TO_Remote_WRITE_BG_Option_OPC` reads a VT button, writes to a target module via remote OPC-UA write and sets the background colour from a remote subscribe - like `Button_IXA_TO_Remote_WRITE_BG_OPC`, but activated via `AOption` instead of `QI`/`INIT`. `AOption_OUT` passes

the activation on after its own `INIT` (daisy chain like `OptionalGroup_001a`). The block is a copy of `Button_IXA_TO_Remote_WRITE_BG_OPC` in which `INIT/INITO/QI/QO` were replaced by `AOption_IN` and `AOption_OUT`.



Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

Variable	Type	Default	Comment
`u16ObjId`	UINT	`ID_NULL`	Object ID button/background (VT).
`ID_SUBSCRIBE`	WSTRING		Remote subscribe address (status/colour, `ACTION=SUBSCRIBE`).
`ID_WRITE_REMOTE`	WSTRING		Remote write address to the target module (command, `ACTION=WRITE`, CLIENT).

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
`AOption_IN`	`Option::AOption`	Socket	Drives `INIT/QI`; connect to `AOption_AUI_DEMUX_N.OUTx` or the `AOption_OUT` of an upstream instance.
`AOption_OUT`	`Option::AOption`	Plug	Passes the activation on to the next link after its own INIT (daisy chain); must be connected, end of chain e.g. `DummyGroup`.

Function Blocks (FBs) Used

- **AOption_X_TO_EVENT** (`Option::conversion::AOption_X_TO_EVENT`): converts the option into `INIT_REQ/QI_OUT` and accepts `INIT_CNF`
- **Button_IXA** (`isobus::UT::io::Button::Button_IXA`): reads the VT button
- **AX_CLIENT_1_0** (`adapter::net::AX_CLIENT_1_0`): remote write to the target module
- **AX_SUBSCRIBE_1** (`adapter::net::AX_SUBSCRIBE_1`): remote subscribe of status/colour
- **GreenWhiteBackground1_AX** (SubApp `MyLib::sys::GreenWhiteBackground1_AX`): background colour; see `GreenWhiteBackground1_AX`
- **AOption_EVENT_TO_X** (`Option::conversion::AOption_EVENT_TO_X`): passes the activation on to `AOption_OUT`

Program Flow and Connections

1. Initialization chain: `AOption_X_TO_EVENT.INIT_REQ -> Button_IXA.INIT; Button_IXA.INITO -> AX_CLIENT_1_0.INIT; AX_CLIENT_1_0.INITO -> AX_SUBSCRIBE_1.INIT; AX_SUBSCRIBE_1.INITO -> GreenWhiteBackground1_AX.INIT; GreenWhiteBackground1_AX.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF.`
2. Qualifier: `AOption_X_TO_EVENT.QI_OUT -> Button_IXA.QI, AX_CLIENT_1_0.QI, AX_SUBSCRIBE_1.QI, GreenWhiteBackground1_AX.QI` and `AOption_EVENT_TO_X.QI_OUT`.

3. Option: `AOption_IN -> AOption_X_TO_EVENT.AOption_IN; AOption_EVENT_TO_X.AOption_OUT -> AOption_OUT`.
4. Command: `Button_IXA.IN -> AX_CLIENT_1_0.IN`.
5. Status: `AX_SUBSCRIBE_1.OUT -> GreenWhiteBackground1_AX.DI1`.
6. Data: `u16ObjId -> Button_IXA.u16ObjId` and `GreenWhiteBackground1_AX.u16ObjId`; `ID_WRITE_REMOTE -> AX_CLIENT_1_0.ID`; `ID_SUBSCRIBE -> AX_SUBSCRIBE_1.ID`.

Technical Features

- The block has no event or `QI`/`QO` connections of its own; activation and acknowledgment run exclusively via `AOption_IN`/`AOption_OUT`.
- `AOption_OUT` must be connected onward (e.g. `DummyGroup` as end of chain), otherwise no `INIT_CNF` is returned.

Application Scenarios

- Optional operator station with a VT button for a remote actuator including coloured status display, active only when the option is selected.
- Several instances sharing the same option, chained via `AOption_OUT`.

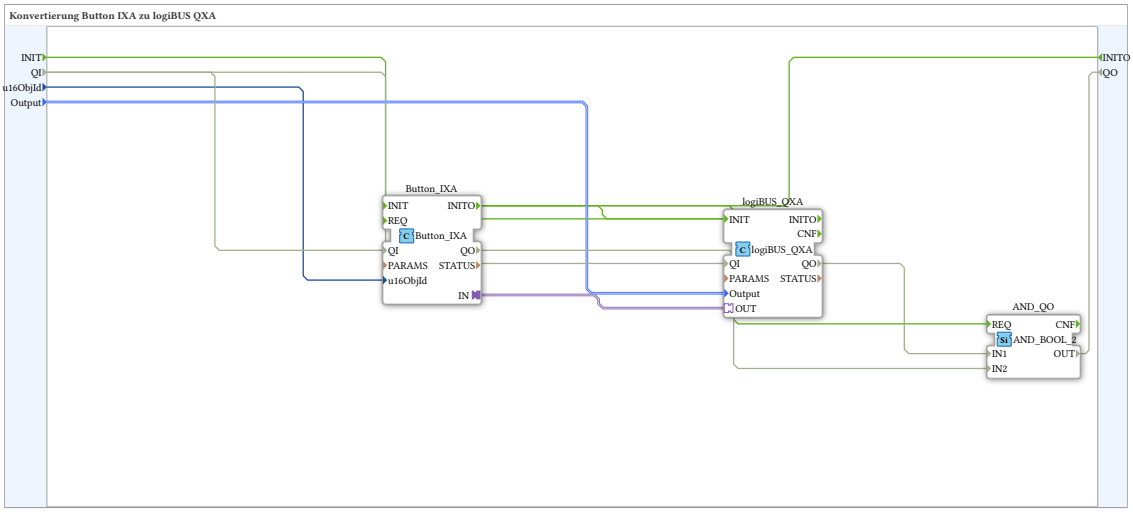
Summary

`Button_IXA_TO_Remote_WRITE_BG_Option_OPC` bundles VT button, remote write and background colour from remote subscribe behind an `AOption` socket and passes the activation on via `AOption_OUT`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

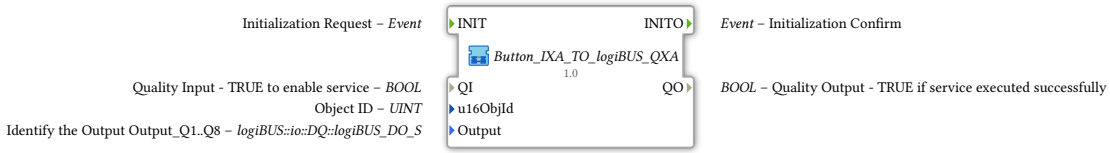
Button_IXA_TO_logiBUS_QXA



• — ○ • — ○ • — ○ •

Introduction

`Button_IXA_TO_logiBUS_QXA` connects a VT button (`Button_IXA`) directly to a physical digital output (`logiBUS_QXA`) — the simplest form of a VT-switchable output, with no status display and no OPC-UA. For the variant with a VT status color, see `Button_IXA_TO_logiBUS_QXA_BG`.



Function Blocks (FBs) Used

Sub-blocks: Button_IXA_TO_logiBUS_QXA

- **Type**: SubAppType
- **Internal FBs used**:
  - **Button_IXA**: `isobus::UT::io::Button::Button_IXA` — VT button adapter, `QI` comes from the input `QI`, `u16ObjId` identifies the VT button.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physical digital output, `QI` comes from the input `QI`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — combines `logiBUS_QXA.QO` and `Button_IXA.QO` into the output `QO`.
- **Functionality**: The button's adapter output is wired directly to the physical output's adapter input — no intermediate logic. In addition, the SubApp has the interface `INIT`/`INITO` (events) and `QI`/`QO` (BOOL): `INIT` starts the init chain, `QI` (initial value `TRUE`) is passed on to both FBs, `QO` is the AND combination of their `QO` outputs.

Program Flow and Connections

1. `u16ObjId` ⟶ `Button_IXA.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`.
2. `Button_IXA.IN` (adapter) ⟶ `logiBUS_QXA.OUT` (adapter) — direct pass-through.
3. `INIT` ⟶ `logiBUS_QXA.INIT`; `logiBUS_QXA.INITO` ⟶ `Button_IXA.INIT`; `Button_IXA.INITO` ⟶ `INITO` and `AND_QO.REQ`.
4. `QI` ⟶ `logiBUS_QXA.QI` and `Button_IXA.QI`.
5. `logiBUS_QXA.QO` ⟶ `AND_QO.IN1`; `Button_IXA.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Application Scenarios

- Minimal VT-button-to-output block for exercises that don't yet need a status display or remote control.

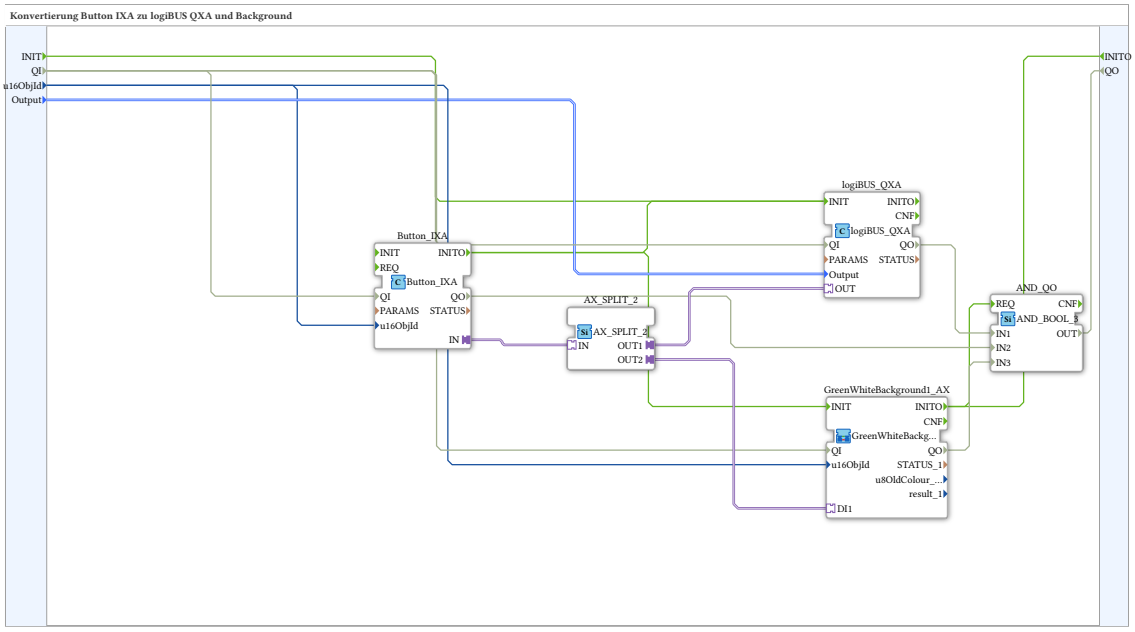
Summary

The simplest block variant in this family: a VT button wired directly to a physical output, with no additional function.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

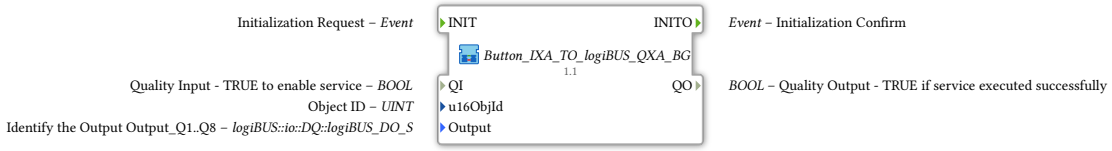
Button_IXA_TO_logiBUS_QXA_BG



• – ○ • – ○ • – ○ •

Introduction

`Button_IXA_TO_logiBUS_QXA_BG` extends `Button_IXA_TO_logiBUS_QXA` with a VT status color: the button state not only switches the physical output but also the button's own background color (green/white). The OPC-UA-capable follow-up is `Button_IXA_TO_logiBUS_QXA_BG_OPC`.



Function Blocks (FBs) Used

Sub-blocks: Button_IXA_TO_logiBUS_QXA_BG

- **Type**: SubAppType
- **Internal FBs used**:
  - **Button_IXA**: `isobus::UT::io::Button::Button_IXA` — VT button adapter.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physical digital output.
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — splits the button signal.
  - **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): sets the VT background color to match the state (see Background Color Blocks).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — combines the `QO` of `logiBUS_QXA`, `Button_IXA` and `GreenWhiteBackground1_AX`.
- **Functionality**: `Button_IXA.IN` is distributed via `AX_SPLIT_2` to both `logiBUS_QXA.OUT` (physical output) and `GreenWhiteBackground1_AX.DI1` (status color).

Program Flow and Connections

1. `u16ObjId` ⟶ `Button_IXA.u16ObjId` and `GreenWhiteBackground1_AX.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`.
2. `Button_IXA.IN` ⟶ `AX_SPLIT_2.IN` ⟶ `AX_SPLIT_2.OUT1` ⟶ `logiBUS_QXA.OUT`, `AX_SPLIT_2.OUT2` ⟶ `GreenWhiteBackground1_AX.DI1`.
3. Initialization: event input `INIT` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `Button_IXA.INIT`, `Button_IXA.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`, `GreenWhiteBackground1_AX.INITO` ⟶ event output `INITO` and ⟶ `AND_QO.REQ`.
4. Quality: data input `QI` (`BOOL`, default `TRUE`) ⟶ `logiBUS_QXA.QI`, `Button_IXA.QI` and `GreenWhiteBackground1_AX.QI`; `logiBUS_QXA.QO` ⟶ `AND_QO.IN1`, `Button_IXA.QO` ⟶ `AND_QO.IN2`, `GreenWhiteBackground1_AX.QO` ⟶ `AND_QO.IN3`, `AND_QO.OUT` ⟶ data output `QO`.

Application Scenarios

- VT button with direct visual feedback (status color) but no OPC-UA remote control.

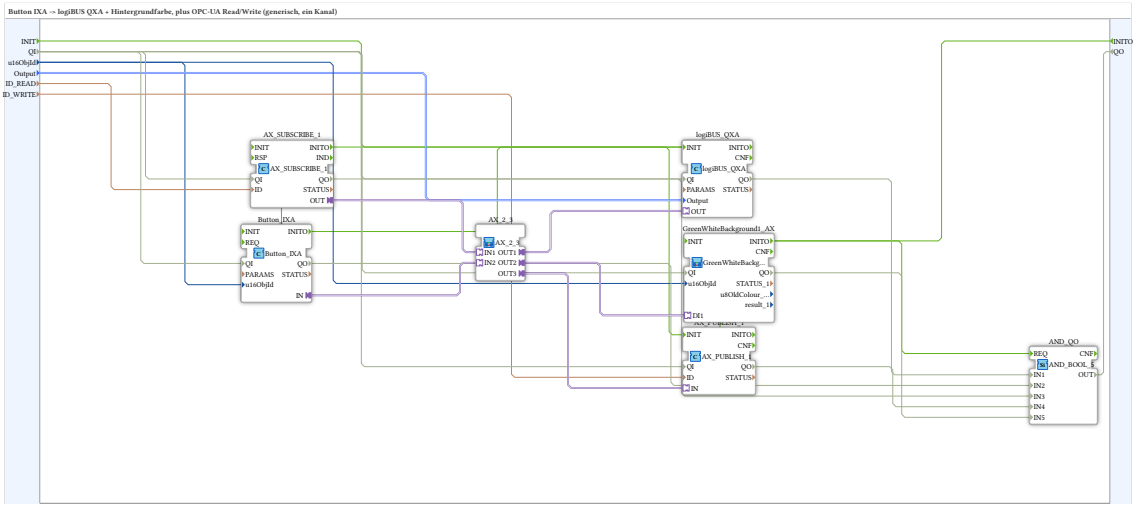
Summary

Adds a VT status color to the simple button-to-output wiring — the precursor to the OPC-UA-capable variant.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

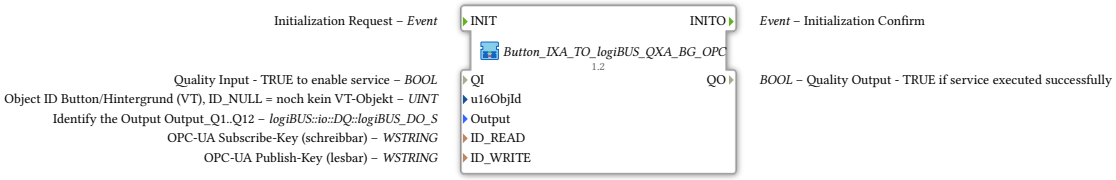
Button_IXA_TO_logiBUS_QXA_BG_OPC



• – ○ • – ○ • – ○ •

Introduction

`Button_IXA_TO_logiBUS_QXA_BG_OPC` is the reusable block for **a single digital output** that can be switched both via a VT button and via OPC-UA (web client) — including a VT status display. It is instantiated 12 times with different parameters in `InputOutputTesterButton_DIDO_OPC_UA`.



Function Blocks (FBs) Used

Sub-blocks: Button_IXA_TO_logiBUS_QXA_BG_OPC

- **Type**: SubAppType
- **Interface (additional)**: event input `INIT`, event output `INITO` (each `EInit`), data input `QI` (`BOOL`, default `TRUE`), data output `QO` (`BOOL`).
- **Internal FBs used**:
  - **Button_IXA**: `isobus::UT::io::Button::Button_IXA` — VT button, `u16ObjId` identifies the VT button. Since version 1.2 temporarily **non-latching** (momentary level, no click toggle any more).
  - **AX_2_3**: SubApp `AX_2_3` — combines `IN1` (OPC-UA subscribe) and `IN2` (VT button) with `AX_OR_2` into one common level and distributes it with `AX_SPLIT_3` to three destinations (`OUT1`..`OUT3`).
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physical digital output.
  - **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): VT background color matching the output state.
  - **AX_SUBSCRIBE_1** / **AX_PUBLISH_1**: `adapter::net::AX_SUBSCRIBE_1/AX_PUBLISH_1` — OPC-UA read/write access.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_5` — combines the `QO` outputs of `logiBUS_QXA`, `Button_IXA`, `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` and `GreenWhiteBackground1_AX` into the `QO` of the SubApp.
- **Functionality**: Two independent switching sources (VT button, OPC-UA write) are ORed in `AX_2_3`; the resulting level simultaneously feeds the physical output, the VT status color, and the OPC-UA echo.

Program Flow and Connections

1. **VT button path**: `Button_IXA.IN` (adapter) ⟶ `AX_2_3.IN2`.
2. **OPC-UA path**: `AX_SUBSCRIBE_1.OUT` (adapter, web write) ⟶ `AX_2_3.IN1`.

3. **Shared state**: `AX_2_3.OUT1` ⟶ `logiBUS_QXA.OUT` (physical output), `AX_2_3.OUT2` ⟶ `GreenWhiteBackground1_AX.DI1` (VT status color), `AX_2_3.OUT3` ⟶ `AX_PUBLISH_1.IN` (OPC-UA echo to the web client).
4. **Initialization chain**: `INIT` ⟶ `logiBUS_QXA.INIT` ⟶ `Button_IXA.INIT` ⟶ `AX_SUBSCRIBE_1.INIT`; `AX_SUBSCRIBE_1.INITO` ⟶ `AX_PUBLISH_1.INIT` (hidden connection) — ensures the publish adapter is only initialized after the subscribe adapter is ready; `AX_PUBLISH_1.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`; `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`).
5. **Parameters**: `u16ObjId` ⟶ `Button_IXA.u16ObjId` and `GreenWhiteBackground1_AX.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`; `ID_READ` ⟶ `AX_SUBSCRIBE_1.ID`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`.
6. **Quality**: `QI` ⟶ `.QI` of `logiBUS_QXA`, `Button_IXA`, `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` and `GreenWhiteBackground1_AX`; their `QO` outputs ⟶ `AND_QO.IN1..IN5`; `AND_QO.OUT` ⟶ `QO`. The instances are no longer hard-wired to `QI=TRUE` but are enabled via the `QI` of the SubApp.

Technical Details

- **Momentary level instead of click toggle**: Since version 1.2 a simple `AX_OR_2` (inside `AX_2_3`) replaces the former SR latch (`AX_ASR_RF_TRIG` × 2 + `ASR_MERGE_2` + `ASR_AX_SR`). VT button **and** remote command are therefore momentary (level); the output follows the OR of both sources. The variant with edge detection and latch is `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`.
- **Separate OPC-UA keys**: `ID_READ` (subscribe key, writable) and `ID_WRITE` (publish key, readable) are separate parameters; the echo path (`AX_PUBLISH_1`) writes the ORed level, `AX_SUBSCRIBE_1` only reads the web client's write access.
- **`AX_SUBSCRIBE_1`'s `RSP` event is a no-op**: verified in FORTE source (`CCommFB::sendData()`, `commfb.cpp`) — for the *Subscriber* service type, the `RSP` branch is a complete no-op; the `INIT`/`RSP` event pair only exists because `CCommFB` is the shared base class for all "responder"-style service FBs (e.g. `SERVER`, where `RSP` does trigger something real).
- **Boot order**: `AX_SUBSCRIBE_1.INITO` triggers `AX_PUBLISH_1.INIT` to avoid race conditions at deployment (compare the same pattern in the PWM example's `RampLimitFS_TO_logiBUS_QDA_PWM_OPC`).

Application Scenarios

- Any digital output that needs to be switchable in momentary mode both locally (VT button) and remotely (web/OPC-UA), with VT status color and OPC-UA echo.

Comparison with Similar Function Blocks

Compared to its input counterpart `logiBUS_IXA_BG_OPC`, this block adds the second write source (OPC-UA subscribe), which is ORed with the VT button via `AX_2_3`. The latching variant with edge detection and latch is `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`.

Summary

`Button_IXA_TO_logiBUS_QXA_BG_OPC` connects a digital output bidirectionally with OPC-UA: VT button and web command are ORed in `AX_2_3` and the level is distributed to hardware, VT display, and OPC-UA echo; `INIT`/`INITO` are chained through the services, `QO` is the AND of the services.

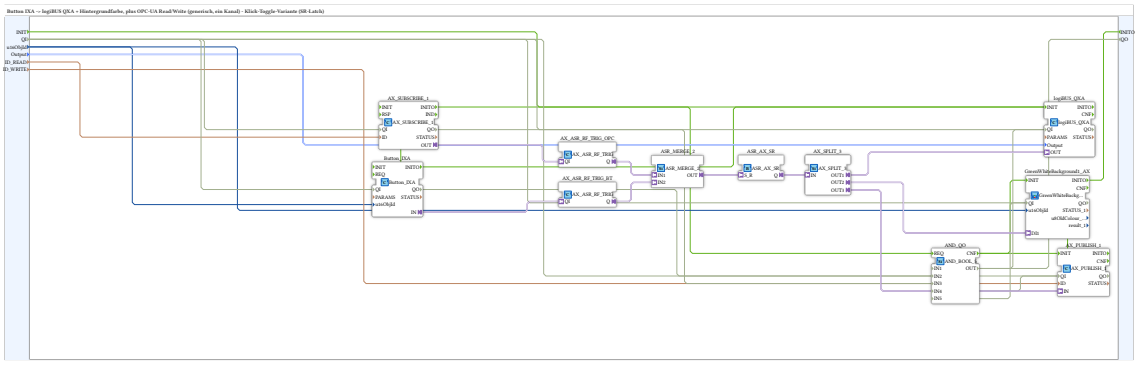
🛠️ Related Exercises

- InputOutputTesterButton_DIDO_OPC_UA

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

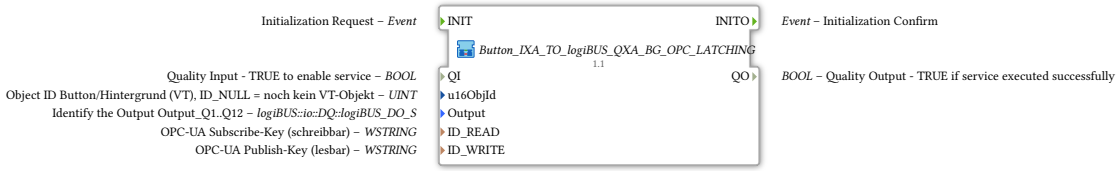
Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING



• — ○ • — ○ • — ○ •

Introduction

`Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING` replaces the simple OR gate (`AX_OR_2`) of `Button_IXA_TO_logiBUS_QXA_BG_OPC` with an edge-detection/merge/latch chain (`AX_ASR_RF_TRIG` × 2, `ASR_MERGE_2`, `ASR_AX_SR`). This is **not** a true click-toggle: `AX_ASR_RF_TRIG` maps the rising edge (press) to `SET` and the falling edge (release) to `RESET`, so with only one source active the block behaves exactly like the momentary `AX_OR_2` variant (ON only while held). The difference only shows up once the VT button and the OPC UA command overlap in time: unlike a true OR gate (which stays ON as long as either source is active), the last edge to arrive always wins here, regardless of which source it came from — e.g. if the OPC UA side sends a release while the VT button is still held, the output switches OFF immediately anyway. The block is retained for use cases that need exactly this last-wins behavior between two sources. Both function blocks share the same interface (`u16ObjId`/`Output`/`ID_READ`/`ID_WRITE`) and are interface-compatible — their switching behavior differs only in this overlap case.



Function Blocks (FBs) Used

Sub-Blocks: Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING

- **Type**: SubAppType
- **Interface (additional)**: event input `INIT`, event output `INITO` (each `EInit`), data input `QI` (`BOOL`, default `TRUE`), data output `QO` (`BOOL`).
- **Internal FBs Used**:
- **Button_IXA**: `isobus::UT::io::Button::Button_IXA` (`QI` from the wrapper input) — VT button, `u16ObjId` identifies the VT button.
- **AX_ASR_RF_TRIG_BT** / **AX_ASR_RF_TRIG_OPC**: each `adapter::events::unidirectional::AX_ASR_RF_TRIG` — convert the rising/falling edge of the VT button or the OPC UA subscribe value into a Set/Reset event pair (ASR).
- **ASR_MERGE_2**: `adapter::events::unidirectional::ASR_MERGE_2` — combines the two ASR sources (button, OPC UA).
- **ASR_AX_SR**: `adapter::events::unidirectional::ASR_AX_SR` — Set/Reset latch that creates the latched output state (last wins) from the combined ASR signal.
- **AX_SPLIT_3**: `adapter::events::unidirectional::AX_SPLIT_3` — distributes the latch output to three destinations.
- **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` (`QI` from the wrapper input) — Physical digital output.

- **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): VT background color matching the output state.
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_5` — combines the `QO` outputs of `logiBUS_QXA`, `Button_IXA`, `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` and `GreenWhiteBackground1_AX` into `QO`.
- **AX_SUBSCRIBE_1** / **AX_PUBLISH_1**: each `adapter::net::AX_SUBSCRIBE_1/AX_PUBLISH_1` (`QI` from the wrapper input) — OPC UA read/write access.

Functionality: Both switching sources (VT button, OPC UA command) are each converted into set/reset events via `AX_ASR_RF_TRIG`, combined via `ASR_MERGE_2`, and latched in `ASR_AX_SR` — the latch state simultaneously powers the hardware, the VT status color, and the OPC UA echo.

Program Flow and Connections

1. `Button_IXA.IN` (adapter) ⟶ `AX_ASR_RF_TRIG_BT.QI` ⟶ `AX_ASR_RF_TRIG_BT.Q` ⟶ `ASR_MERGE_2.IN2`.
2. `AX_SUBSCRIBE_1.OUT` ⟶ `AX_ASR_RF_TRIG_OPC.QI` ⟶ `AX_ASR_RF_TRIG_OPC.Q` ⟶ `ASR_MERGE_2.IN1`
3. `ASR_MERGE_2.OUT` ⟶ `ASR_AX_SR.S_R` (Set/Reset input of the latch).
4. `ASR_AX_SR.Q` ⟶ `AX_SPLIT_3.IN` ⟶ `AX_SPLIT_3.OUT1` ⟶ `logiBUS_QXA.OUT`, `AX_SPLIT_3.OUT2` ⟶ `GreenWhiteBackground1_AX.DI1`, `AX_SPLIT_3.OUT3` ⟶ `AX_PUBLISH_1.IN`.
5. Initialization: `AX_SUBSCRIBE_1.INITO` ⟶ `AX_PUBLISH_1.INIT` (hidden connection). The chain runs in from the outside: `INIT` ⟶ `logiBUS_QXA.INIT`; `logiBUS_QXA.INITO` ⟶ `Button_IXA.INIT`; `Button_IXA.INITO` ⟶ `AX_SUBSCRIBE_1.INIT`; `AX_PUBLISH_1.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`; `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
6. Parameters: `u16ObjId` ⟶ `Button_IXA.u16ObjId` and `GreenWhiteBackground1_AX.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`; `ID_READ` ⟶ `AX_SUBSCRIBE_1.ID`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`.
7. Quality: `QI` ⟶ `.QI` of `logiBUS_QXA`, `Button_IXA`, `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` and `GreenWhiteBackground1_AX`; their `QO` ⟶ `AND_QO.IN1` to `AND_QO.IN5` (in that order); `AND_QO.OUT` ⟶ `QO`.

Technical Features

- **Last-wins via ASR latch, not a real toggle**: Each actuation produces a set (press) or reset (release) event per source; `ASR_AX_SR` latches the combined result. With only one source active, the behavior is purely momentary (ON only while held) — identical to the `AX_OR_2` variant. The difference only shows up once both sources overlap: whichever edge (set or reset, from either source) arrives last determines the state, rather than a continuous OR.
- **Two independent last-wins sources**: The VT button and the OPC UA command each independently deliver set/reset events; `ASR_MERGE_2` combines both into a common latch input without prioritizing either source.
- **Interface-compatible external interface**: The interface is identical to `Button_IXA_TO_logiBUS_QXA_BG_OPC`, so both variants can be instantiated interchangeably. The switching behavior is only identical with a single active source — with overlapping use of both sources it differs (last-wins vs. a true OR gate).

Application Scenarios

- Outputs with two independent switching sources (e.g. a local VT button AND a remote OPC UA command) where whichever action arrives last (press or release, from either source) should win, instead of one permanently active source blocking the other — which is exactly what a true OR gate would do.

Comparison with Similar Modules

Compared to `Button_IXA_TO_logiBUS_QXA_BG_OPC` (currently: tactile `AX_OR_2`), this module replaces the simple OR operation with an edge-detection/merge/latch chain (`AX_ASR_RF_TRIG` × 2, `ASR_MERGE_2`, `ASR_AX_SR`). With only one source active the behavior is identical (momentary); the difference only shows up once both sources overlap (last-wins instead of OR).

Summary

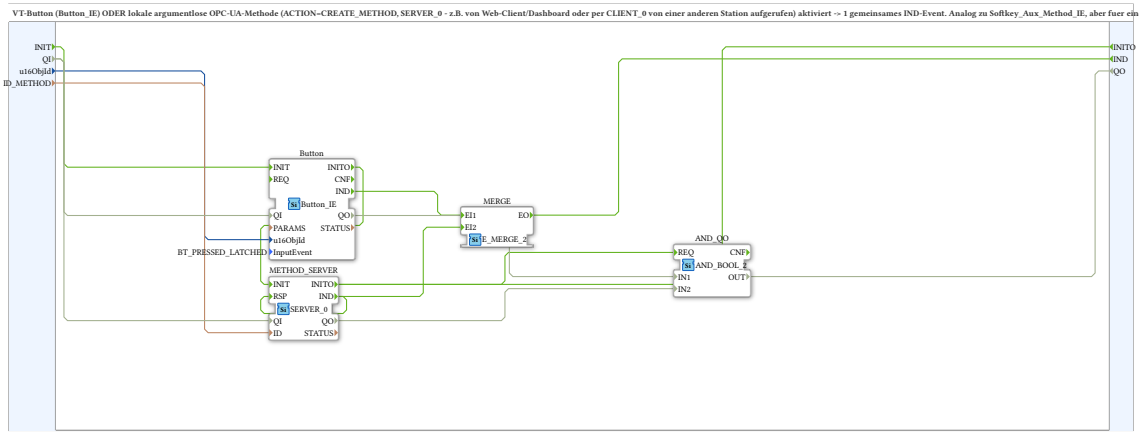
With only one source active, `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING` offers the same momentary basic functionality as `Button_IXA_TO_logiBUS_QXA_BG_OPC` — the difference is last-wins behavior for

overlapping VT/OPC UA commands, not a click-toggle. Both blocks are interface-compatible, but not behaviorally identical in every scenario.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

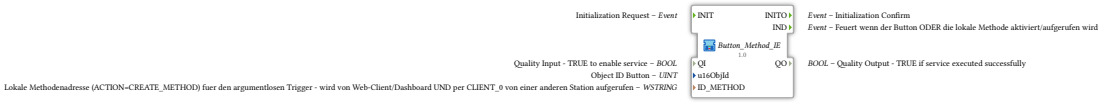
Button_Method_IE



• ─ ○ • ─ ○ • ─ ○ •

Introduction

`Button_Method_IE` raises a common `IND` event when either a **VT button** (`Button_IE`) is activated or a **local argument-less OPC-UA method** (`ACTION=CREATE_METHOD`, `SERVER_0`) is called, e.g. by the web client/dashboard or via `CLIENT_0` from another station. The block is analogous to `Softkey_Aux_Method_IE`, but for a regular VT button instead of SoftKey/AUX. It was extracted from `logiBUS_AI_Calibrate_IDA_OPC`, where the combination (CO/CS) occurred twice identically.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization Request

Event Outputs

- **INITO** (`EInit`): Initialization Confirm
- **IND**: fires when the button **or** the local method is activated/called

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE to enable the service
- **u16ObjId** (`UINT`): object ID of the button
- **ID_METHOD** (`WSTRING`): local method address (`ACTION=CREATE_METHOD`) for the argument-less trigger; called by the web client/dashboard and via `CLIENT_0` from another station

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service executed successfully (AND of `Button.QO` and `METHOD_SERVER.QO`)

Adapters

None.

Function Blocks (FBs) Used

- **`Button`** (`isobus::UT::io::Button::Button_IE`): parameter `InputEvent = BT_PRESSED_LATCHED`, `QI` from the block input
- **`METHOD_SERVER`** (`iec61499::net::SERVER_0`): `QI` from the block input, provides the local method
- **`MERGE`** (`iec61499::events::E_MERGE_2`): merges both triggers
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): ANDs `Button.QO` and `METHOD_SERVER.QO` into `QO`

Functionality

- `u16ObjId` ⟶ `Button.u16ObjId`; `ID_METHOD` ⟶ `METHOD_SERVER.ID`
- `Button.IND` ⟶ `MERGE.EI1`
- `METHOD_SERVER.IND` ⟶ `MERGE.EI2`
- `METHOD_SERVER.IND` ⟶ `METHOD_SERVER.RSP` (immediate response)
- `MERGE.EO` ⟶ `IND`
- Initialization chain: `INIT` ⟶ `Button.INIT`, `Button.INITO` ⟶ `METHOD_SERVER.INIT`, `METHOD_SERVER.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`
- `QI` ⟶ `Button.QI` and `METHOD_SERVER.QI`
- `Button.QO` ⟶ `AND_QO.IN1`, `METHOD_SERVER.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`

Technical Details

- **`RSP` must be wired back to `IND`.** The trigger is argument-less and needs no processing. Without the loop-back, FORTE's `onServerMethodCall` blocks the OPC-UA server thread for up to 4 s and meanwhile holds open62541's global `serviceMutex`, which every `AX_PUBLISH_1` on another FORTE thread also needs (see `Ventilsteuerung/OPC_UA_VERNETZUNG.md`, item 12).
- `BT_PRESSED_LATCHED` as the button event (latched press/release)

Application Scenarios

- A button that should additionally be triggered via web client, OPC dashboard or from another station (e.g. a calibration trigger)

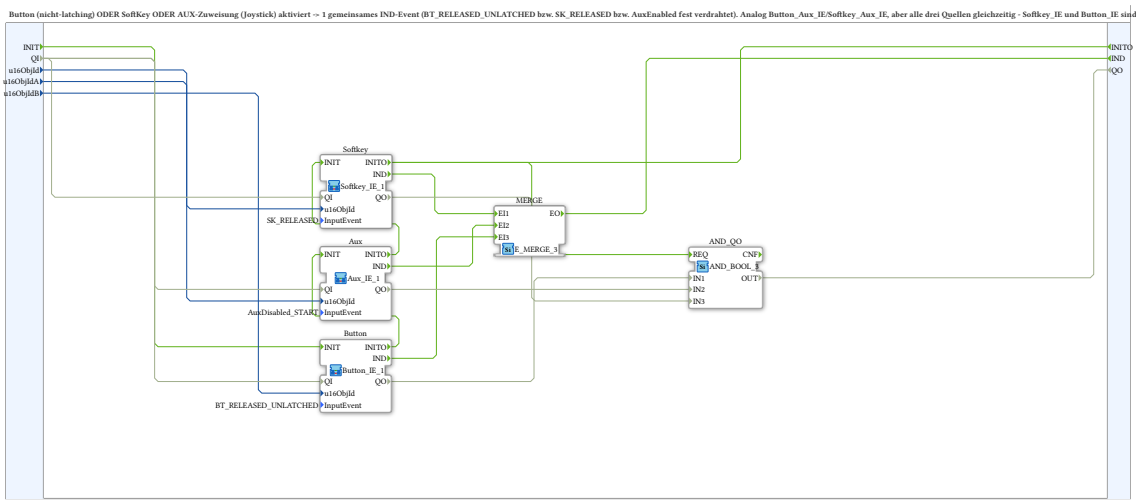
Summary

`Button_Method_IE` combines a VT button and a local OPC-UA method into one common `IND` event.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Button_Softkey_Aux_IE

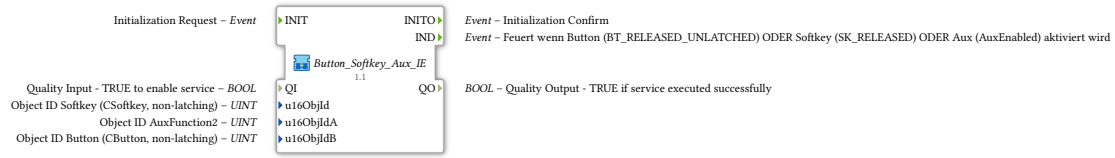


• – ○ • – ○ • – ○ •

Introduction

`Button_Softkey_Aux_IE` combines a non-latching VT button **or** a SoftKey **or** an AUX assignment (joystick) into one common `IND` event. The activation types are hard-wired (`BT_RELEASED_UNLATCHED`, `SK_RELEASED` and `AuxEnabled` respectively). The block corresponds to `Button_Aux_IE` and `Softkey_Aux_IE`, but serves all three sources at once. `Softkey_IE` and `Button_IE` are separate FORTE service interface blocks for different VT activation messages; a button never delivers an `SK_PRESSED`.

The button **must** be non-latching (`Latchable=0` in the pool); a latching button can sensibly be read only via `Button_IXA` (level).



Function Blocks (FBs) Used

Sub-blocks: Button_Softkey_Aux_IE

- **Type**: SubAppType
- **Internal blocks used**:
  - **Button**: SubApp `Button_IE_1` with `InputEvent=BT_RELEASED_UNLATCHED`
  - **Softkey**: SubApp `Softkey_IE_1` with `InputEvent=SK_RELEASED`
  - **Aux**: SubApp `Aux_IE_1` with `InputEvent=AuxDisabled_START`
  - **MERGE**: `iec61499::events::E_MERGE_3` - merges the three `IND` events
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` - combines the `QO` outputs of `Button`, `Aux` and `Softkey` into the SubApp output `QO`
- **Imports**: `BT_RELEASED_UNLATCHED`, `AuxDisabled_START`, `SK_RELEASED`

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the chain `Button` ⟶ `Aux` ⟶ `Softkey`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `Softkey.INITO`
- **IND**: Fires when a button (`BT_RELEASED_UNLATCHED`), softkey (`SK_RELEASED`) or aux (`AuxEnabled`) is activated

Data Inputs

- **QI** (`BOOL`): quality input, service enable (initial value `TRUE`); passed on to `Button.QI`, `Aux.QI` and `Softkey.QI`
- **u16ObjId** (`UINT`): Object ID of the softkey (`CSoftkey`, non-latching)
- **u16ObjIdA** (`UINT`): Object ID of the `AuxFunction2`
- **u16ObjIdB** (`UINT`): Object ID of the button (`CButton`, non-latching)

Data Outputs

- **QO** (`BOOL`): quality output, `AND` of the `QO` outputs of `Button`, `Aux` and `Softkey` (via `AND_QO`)

Program Flow and Connections

- `u16ObjId` ⟶ `Softkey.u16ObjId`; `Softkey.IND` ⟶ `MERGE.EI1`
- `u16ObjIdA` ⟶ `Aux.u16ObjId`; `Aux.IND` ⟶ `MERGE.EI2`
- `u16ObjIdB` ⟶ `Button.u16ObjId`; `Button.IND` ⟶ `MERGE.EI3`
- `MERGE.EO` ⟶ `IND`
- Initialization: `INIT` ⟶ `Button.INIT`; `Button.INITO` ⟶ `Aux.INIT`; `Aux.INITO` ⟶ `Softkey.INIT`; `Softkey.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`
- Quality: `QI` ⟶ `Button.QI`, `Aux.QI` and `Softkey.QI`; `Button.QO` ⟶ `AND_QO.IN1`, `Aux.QO` ⟶ `AND_QO.IN2`, `Softkey.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`

Technical Details

- Version 1.1 (2026-09-29): `packageName` typo fixed (`MyLib_AX::sys` → `MyLib::sys`), instance `Softkey_IE_1` renamed to `Softkey`, comment corrected (it described only 2 of 3 sources)
- Version 1.0 (2026-09-20): bugfix for `Automatik_Toggle_Button`, which previously used `Softkey_Aux_IE` with a button object ID

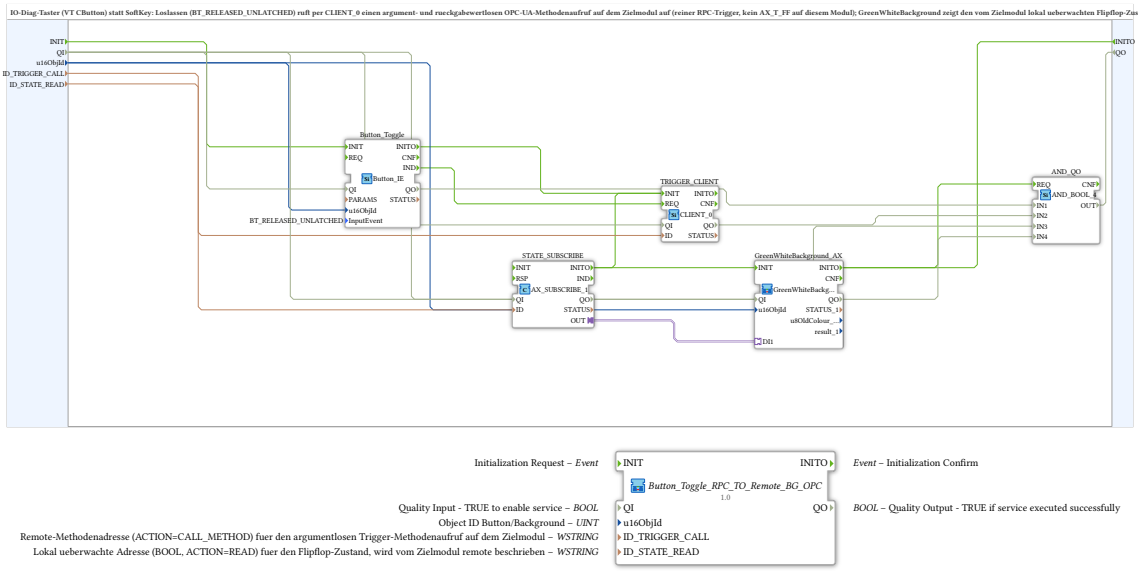
Application Scenarios

- A control element that can be triggered via a button, a SoftKey or an AUX input

Summary

`Button_Softkey_Aux_IE` bundles `Button_IE_1`, `Softkey_IE_1` and `Aux_IE_1` via an `E_MERGE_3` into a single `IND` event.

Button_Toggle_RPC_TO_Remote_BG_OPC



• — ○ • — ○ • — ○ •

Introduction

The **Button_Toggle_RPC_TO_Remote_BG_OPC** subapplication implements a remote toggle control for IO-Diag buttons on a machine operator panel. When the operator releases the button (activation code `BT_RELEASED_UNLATCHED`), the subapp initiates an argument-less and return-value-less OPC-UA method call on a remote target module via a `CLIENT_0` communication function block. The remote flip-flop state is continuously monitored through a subscription and visually represented by a GreenWhiteBackground indicator. The subapp is designed for IO-Diag DataMask buttons (e.g. VT CButton) and follows the same protocol as its softkey counterpart, differing only in the button input type.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the init chain at `Button_Toggle.INIT`.

The actual event handling is still performed internally by the embedded function blocks.

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `GreenWhiteBackground_AX.INITO`, the end of the init chain.

There are no other event outputs.

Data Inputs

Name	Type	Description

QI	BOOL	Quality input (initial value TRUE), enables the service; passed on to Button_Toggle, TRIGGER_CLIENT, STATE_SUBSCRIBE and GreenWhiteBackground_AX.
u16ObjId	UINT	Object ID for the button and the visual background element. Initial value is ID_NULL.
ID_TRIGGER_CALL	WSTRING	Remote method address (with ACTION=CALL_METHOD) used for the argument-less trigger method invocation on the target module.
ID_STATE_READ	WSTRING	Locally monitored address (BOOL, with ACTION=READ) for the flip-flop state, which is written remotely by the target module.

Data Outputs

- **QO** (BOOL): quality output, AND combination (AND_QO, AND_BOOL_4) of the QO outputs of Button_Toggle, TRIGGER_CLIENT, STATE_SUBSCRIBE and GreenWhiteBackground_AX.

Adapters

None exposed at the subapplication interface. Adapter connections are entirely internal.

Functionality

The subapplication provides a complete remote toggle workflow in a single reusable package:

1. **Button Input Handling** – The embedded Button_IE function block (type isobus::UT::io::Button::Button_IE) is configured with InputEvent=BT_RELEASED_UNLATCHED (its QI comes from the block input QI). It detects the physical button release on the IO-Diag mask and generates an event on its IND output.
2. **Remote Trigger Invocation** – The button release event (Button_Toggle.IND) is connected to the REQ input of the TRIGGER_CLIENT function block (type iec61499::net::CLIENT_0). TRIGGER_CLIENT uses the address provided via ID_TRIGGER_CALL to perform an OPC-UA CALL_METHOD operation on the remote target module. Since the call is argument-less and return-value-less, it acts purely as an RPC trigger – no local flip-flop (AX_T_FF) logic is executed on this module.
3. **State Subscription** – The STATE_SUBSCRIBE function block (type adapter::net::AX_SUBSCRIBE_1) continuously reads the remote boolean state from the address specified by ID_STATE_READ (with ACTION=READ). Its output adapter is connected to the DI1 input of the GreenWhiteBackground_AX subapplication.
4. **Visual Feedback** – The GreenWhiteBackground_AX subapplication (type MyLib::sys::GreenWhiteBackground1_AX) receives the remote flip-flop state and renders it as a green or white background on the associated IO-Diag element, giving the operator immediate visual confirmation of the remote toggle state.
5. **Initialization and Quality** – INIT → Button_Toggle.INIT, Button_Toggle.INITO → TRIGGER_CLIENT.INIT, TRIGGER_CLIENT.INITO → STATE_SUBSCRIBE.INIT, STATE_SUBSCRIBE.INITO → GreenWhiteBackground_AX.INIT, GreenWhiteBackground_AX.INITO → INITO and AND_QO.REQ. QI is passed on to all four blocks; their QO outputs (Button_Toggle, TRIGGER_CLIENT, STATE_SUBSCRIBE, GreenWhiteBackground_AX) go to AND_QO.IN1 to AND_QO.IN4 (iec61131::booleanOperators::AND_BOOL_4), whose OUT forms the output QO.

The u16ObjId input is distributed internally to both the button (Button_Toggle.u16ObjId) and the background (GreenWhiteBackground_AX.u16ObjId), ensuring that both the input element and the visual indicator reference the same object instance.

Technical Features

- **IEC 61499-2 compliant** subapplication type with a cleanly encapsulated networking layer.
- **OPC-UA remote method invocation** via iec61499::net::CLIENT_0 with CALL_METHOD action, supporting argument-less, return-value-less calls (pure RPC trigger).
- **Subscription-based state monitoring** using adapter::net::AX_SUBSCRIBE_1 for continuous polling of the remote boolean state.

- **Activation code configuration** – the button is preconfigured with `BT_RELEASED_UNLATCHED`, meaning the trigger is fired on the release transition (unlatched), preventing repeated triggers while the button is held.
- **Unified object identification** – a single `u16ObjId` parameter links both the button and the background visualization to the same target object.
- **Hidden internal connections** – all internal data connections are marked as non-visible, keeping the subapplication interface clean and focused on the three essential configuration inputs.
- **Standard library components** – uses standard 4diac function blocks and adapters, facilitating maintenance and reuse.

State Overview

The subapplication itself is stateless – it delegates all stateful behavior to its internal blocks. The logical state flow can be described as follows:

Stage	Component	Behavior
Idle	`Button_IE`	Waits for the physical button to be released; no event is generated while the button is pressed or held.
Trigger	`Button_Toggle.IND`	Fires on the release transition (`BT_RELEASED_UNLATCHED`). The event is forwarded to `TRIGGER_CLIENT.REQ`.
Remote Call	`TRIGGER_CLIENT`	Executes the OPC-UA `CALL_METHOD` on the target module using `ID_TRIGGER_CALL`. The remote module toggles its internal flip-flop state.
State Read	`STATE_SUBSCRIBE`	Continuously reads the remote boolean value from `ID_STATE_READ` and forwards it through the adapter to the background visualization.
Visualization	`GreenWhiteBackground_AX`	Displays the current remote state: green when active, white when inactive.

There is no internal state machine in the subapp; the behavior is fully event-driven and data-driven by the remote system.

Application Scenarios

- **IO-Diag control panels** – Used on data masks with physical buttons (VT CButton) to toggle remote functions such as record enable/release on camera modules (e.g. Q06/Q08 left/right recording toggle).
- **Remote machine control** – Where operator panel buttons need to trigger actions on distributed PLC or edge modules via OPC-UA without embedding complex logic in the panel itself.
- **Visual state feedback** – Ideal when the operator needs immediate confirmation of a remote toggle state through a colored background indicator (green/white) on the same IO-Diag element.
- **Reusable RPC trigger pattern** – Suitable for any scenario where a button press should issue a remote method call and the result should be visualized locally, following a standard 4diac pattern.

Comparison with Similar Blocks

The main counterpart is the **Softkey_Toggle_RPC_TO_Remote_BG_OPC** subapplication, which follows an identical protocol and internal architecture. The key differences are:

Aspect	Button_Toggle_RPC_TO_Remote_BG_OPC	Softkey_Toggle_RPC_TO_Remote_BG_OPC
Input element	`Button_IE` (IO-Diag button / VT CButton)	Softkey (`Softkey_IE`)
Activation code	`BT_RELEASED_UNLATCHED` (release-triggered)	Softkey-specific activation logic
Use case	Physical buttons on an IO-Diag Data-Mask	Softkeys on a SoftKeyMask
Protocol & RPC	Identical (CLIENT_0, CALL_METHOD)	Identical

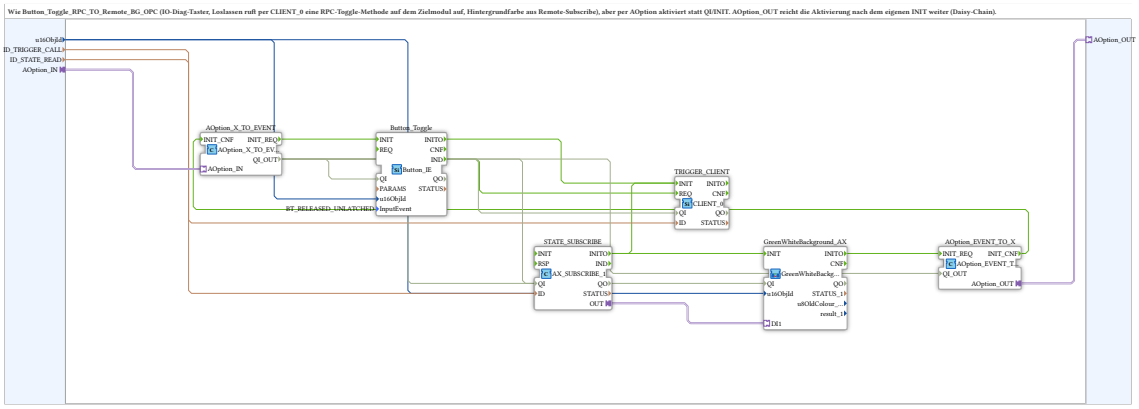
State visualization	Identical (GreenWhiteBackground)	Identical

Both variants share the same networking, subscription, and visualization mechanics, making them interchangeable from a protocol perspective – only the input front-end differs.

Conclusion

The **Button_Toggle_RPC_TO_Remote_BG_OPC** subapplication provides a robust, reusable solution for remote toggle control via OPC-UA RPC calls, triggered by physical IO-Diag buttons. It separates the button input handling from the remote communication and visualization logic, keeping the subapplication interface minimal (three configuration inputs plus `QI`) while encapsulating all complex networking inside. Its design follows the IEC 61499 standard and integrates seamlessly with standard 4diac library components, making it suitable for a wide range of industrial operator panel scenarios where a button press must toggle a remote state and provide immediate visual feedback.

Button_Toggle_RPC_TO_Remote_BG_Option_OPC



• – ○ • – ○ • – ○ •

Introduction

`Button_Toggle_RPC_TO_Remote_BG_Option_OPC` is an IO diagnostic button: on release it calls an RPC toggle method on the target module via `CLIENT_0`, and the background colour comes from a remote subscribe. The block corresponds to `Button_Toggle_RPC_TO_Remote_BG_OPC`, but is activated via `AOption` instead of `QI/INIT`. `AOption_OUT` passes the activation on after its own `INIT` (daisy chain like `OptionalGroup_001a`). It is a copy of `Button_Toggle_RPC_TO_Remote_BG_OPC` in which `INIT/INITO/QI/QO` were replaced by `AOption_IN` and `AOption_OUT`.



Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

Variable	Type	Default	Comment
`u16ObjId`	UINT	`ID_NULL`	Object ID button/background.
`ID_TRIGGER_CALL`	WSTRING		Remote method address (`ACTION=CALL_METHOD`) for the argument-less trigger method call on the target module.

ID_STATE_READ	WSTRING		Locally monitored address (BOOL, ACTION=READ) for the flip-flop state; written remotely by the target module.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
AOption_IN	Option::AOption	Socket	Drives INIT/QI; connect to AOption_AUI_DEMUX_N.OUTx or the AOption_OUT of an upstream instance.
AOption_OUT	Option::AOption	Plug	Passes the activation on to the next link after its own INIT (daisy chain); must be connected, end of chain e.g. DummyGroup.

Function Blocks (FBs) Used

- **AOption_X_TO_EVENT** (Option::conversion::AOption_X_TO_EVENT): converts the option into INIT_REQ/QI_OUT and accepts INIT_CNF
- **Button_Toggle** (isobus::UT::io::Button::Button_IE, InputEvent = BT_RELEASED_UNLATCHED): VT button, event on release
- **TRIGGER_CLIENT** (iec61499::net::CLIENT_0): calls the method on the target module
- **STATE_SUBSCRIBE** (adapter::net::AX_SUBSCRIBE_1): remote subscribe of the state
- **GreenWhiteBackground_AX** (SubApp MyLib::sys::GreenWhiteBackground1_AX): background colour; see GreenWhiteBackground1_AX
- **AOption_EVENT_TO_X** (Option::conversion::AOption_EVENT_TO_X): passes the activation on to AOption_OUT

Program Flow and Connections

1. Initialization chain: AOption_X_TO_EVENT.INIT_REQ -> Button_Toggle.INIT; Button_Toggle.INITO -> TRIGGER_CLIENT.INIT; TRIGGER_CLIENT.INITO -> STATE_SUBSCRIBE.INIT; STATE_SUBSCRIBE.INITO -> GreenWhiteBackground_AX.INIT; GreenWhiteBackground_AX.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF.
2. Method call: Button_Toggle.IND -> TRIGGER_CLIENT.REQ.
3. Qualifier: AOption_X_TO_EVENT.QI_OUT -> Button_Toggle.QI, TRIGGER_CLIENT.QI, STATE_SUBSCRIBE.QI, GreenWhiteBackground_AX.QI and AOption_EVENT_TO_X.QI_OUT.
4. Option: AOption_IN -> AOption_X_TO_EVENT.AOption_IN; AOption_EVENT_TO_X.AOption_OUT -> AOption_OUT.
5. Status: STATE_SUBSCRIBE.OUT -> GreenWhiteBackground_AX.DI1.
6. Data: u16ObjId -> Button_Toggle.u16ObjId and GreenWhiteBackground_AX.u16ObjId; ID_TRIGGER_CALL -> TRIGGER_CLIENT.ID; ID_STATE_READ -> STATE_SUBSCRIBE.ID.

Technical Features

- The block has no event or QI/QO connections of its own; activation and acknowledgment run exclusively via AOption_IN/AOption_OUT.
- AOption_OUT must be connected onward (e.g. DummyGroup as end of chain), otherwise no INIT_CNF is returned.
- The state is not computed here but delivered remotely by the target module.

Application Scenarios

- Optional diagnostic button that triggers an output toggle on a remote module and shows its state in colour, active only when the option is selected.
- Several instances sharing the same option, chained via AOption_OUT.

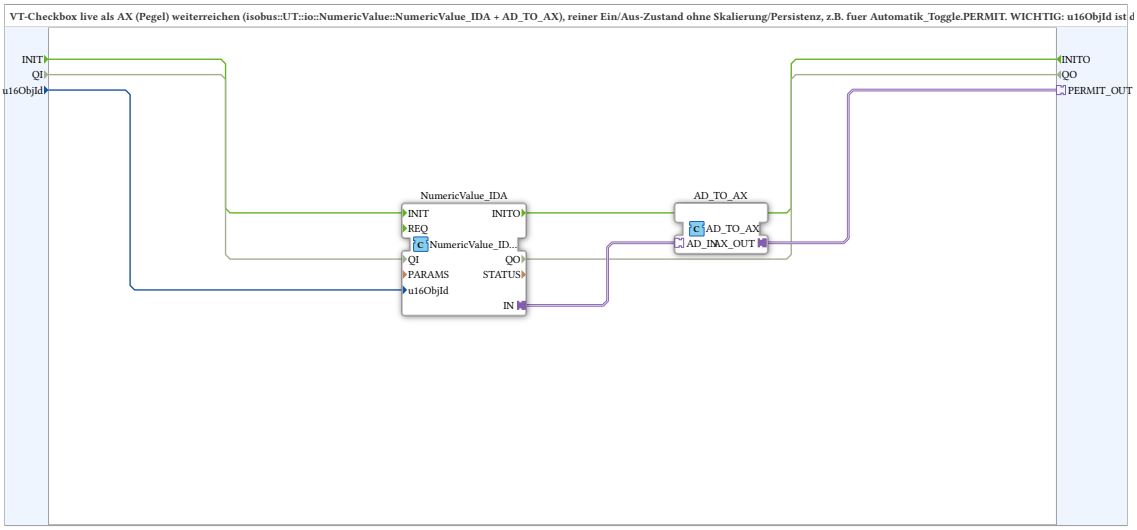
Summary

Button_Toggle_RPC_TO_Remote_BG_Option_OPC triggers a remote method call on release, shows the remotely delivered state as background colour and is activated and chained via AOption.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Checkbox_IDA_TO_AX

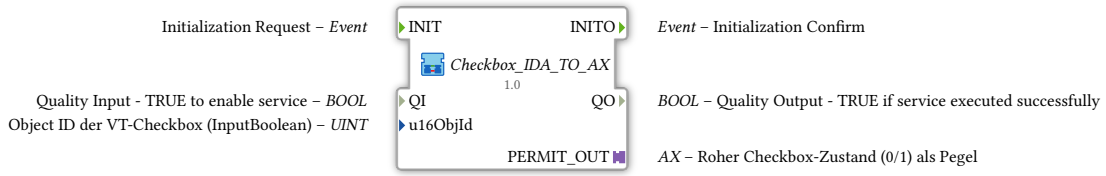


• — ○ • — ○ • — ○ •

Introduction

`Checkbox_IDA_TO_AX` passes the state of a **VT checkbox on live as an AX level** (`isobus::UT::io::NumericValue::NumericValue_IDA + AD_TO_AX`). It is a pure on/off state without scaling and persistence, e.g. for `Automatik_Toggle.PERMIT`.

Important: `u16ObjId` is the object ID of the **`CNumberVariable`** that the checkbox references as its value store (e.g. `NumberVariable_Bunker_Fuellstand_Hoehe_Automatik`) - **not** the object ID of the checkbox (`CInputBoolean`) itself. The VT reports value changes under the NumberVariable ID (verified live).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; goes to `NumericValue_IDA.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `NumericValue_IDA.INITO`

Data Inputs

- **QI** (`BOOL`): quality input - TRUE enables the service (default `TRUE`); goes to `NumericValue_IDA.QI`
- **u16ObjId** (`UINT`): object ID of the number variable that the VT checkbox (InputBoolean) uses as its value store (default `ID_NULL`)

Data Outputs

- **QO** (`BOOL`): quality output - TRUE if the service was executed successfully (comes from `NumericValue_IDA.QO`)

Adapters

- **PERMIT_OUT** (plug, `adapter::types::unidirectional::AX`): raw checkbox state (0/1) as a level

Function Blocks (FBs) Used

- **`NumericValue_IDA`** (`isobus::UT::io::NumericValue::NumericValue_IDA`): delivers the value of the number variable as an `AD` adapter
- **`AD_TO_AX`** (`adapter::conversion::unidirectional::AD_TO_AX`): converts the value to an `AX` level

Functionality

- `u16ObjId` ⟶ `NumericValue_IDA.u16ObjId`
- `NumericValue_IDA.IN` ⟶ `AD_TO_AX.AD_IN`
- `INIT` ⟶ `NumericValue_IDA.INIT`, `NumericValue_IDA.INITO` ⟶ `INITO`
- `QI` ⟶ `NumericValue_IDA.QI`, `NumericValue_IDA.QO` ⟶ `QO`
- `AD_TO_AX.AX_OUT` ⟶ `PERMIT_OUT`

Technical Details

- Level only; no scaling, no persistence
- Variant with initialisation: Checkbox_IDA_TO_AX_INIT_Force1

Application Scenarios

- Permit (`PERMIT`) of an automatic function via a checkbox in the VT, e.g. for `Automatik_Toggle`

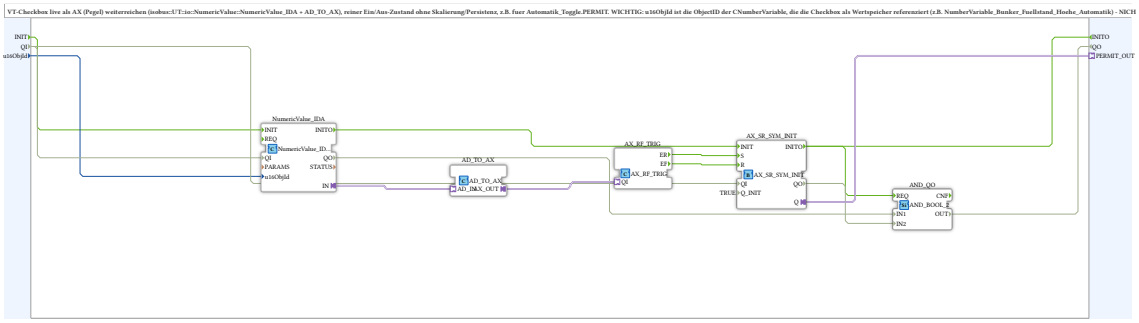
Summary

`Checkbox_IDA_TO_AX` turns a VT checkbox into an `AX` level for further logic.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Checkbox_IDA_TO_AX_INIT

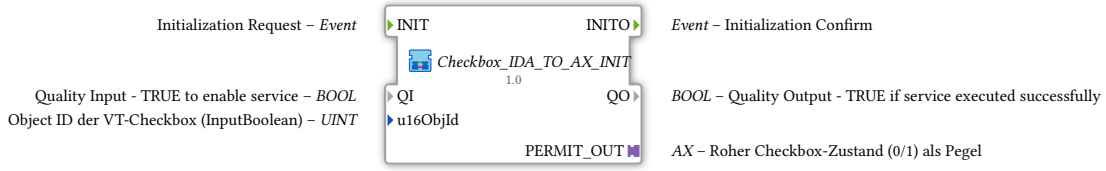


• — ○ • — ○ • — ○ •

Introduction

`Checkbox_IDA_TO_AX_INIT` passes the state of a VT checkbox on live as an `AX` level (`isobus::UT::io::NumericValue::NumericValue_IDA` + `AD_TO_AX`). It is a pure on/off state without scaling or persistence, e.g. for `Automatik_Toggle.PERMIT`. The state is held via an edge detector and an SR flip-flop with a defined initial value (`_INIT`).

Important: `u16ObjId` is the ObjectID of the `CNumberVariable` that the checkbox references as its value store (e.g. `NumberVariable_Bunker_Fuellstand_Hoehe_Automatik`) - **not** the ObjectID of the checkbox (`CInputBoolean`) itself. The VT reports value changes under the NumberVariable ID (verified live).



Function Blocks (FBs) Used

Sub-blocks: Checkbox_IDA_TO_AX_INIT

- **Type**: SubAppType

- **Internal FBs used**:
  - **NumericValue_IDA**: `isobus::UT::io::NumericValue::NumericValue_IDA` - reads the raw value by ObjectID
  - **AD_TO_AX**: `adapter::conversion::unidirectional::AD_TO_AX` - converts the `DWORD` value into an `AX` signal (value <> `0`)
  - **AX_RF_TRIG**: `adapter::events::unidirectional::AX_RF_TRIG` - detects rising (`ER`) and falling (`EF`) edge
  - **AX_SR_SYM_INIT**: `adapter::events::unidirectional::AX_SR_SYM_INIT` (`Q_INIT=TRUE`) - flip-flop with a defined initial value
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` - combines `NumericValue_IDA.QO` and `AX_SR_SYM_INIT.QO` into `QO`
- **Import**: `isobus::UT::Q::const::IDs::ID_NULL` (initial value of `u16ObjId`)

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation request

Event Outputs

- **INITO** (`EInit`): initialisation confirmation

The user events still travel via the adapters.

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **u16ObjId** (`UINT`, initial value `ID_NULL`): Object ID of the VT checkbox, i.e. of the NumberVariable it references

Data Outputs

- **QO** (`BOOL`): Quality Output - AND of `NumericValue_IDA.QO` and `AX_SR_SYM_INIT.QO`

Adapters

- **PERMIT_OUT** (plug): `adapter::types::unidirectional::AX` - raw checkbox state (0/1) as a level

Program Flow and Connections

- `u16ObjId` ⟶ `NumericValue_IDA.u16ObjId`
- `NumericValue_IDA.IN` ⟶ `AD_TO_AX.AD_IN`
- `AD_TO_AX.AX_OUT` ⟶ `AX_RF_TRIG.QI`
- `AX_RF_TRIG.ER` ⟶ `AX_SR_SYM_INIT.S` (checkbox set)
- `AX_RF_TRIG.EF` ⟶ `AX_SR_SYM_INIT.R` (checkbox cleared)
- `AX_SR_SYM_INIT.Q` ⟶ `PERMIT_OUT`
- `INIT` ⟶ `NumericValue_IDA.INIT`; `NumericValue_IDA.INITO` ⟶ `AX_SR_SYM_INIT.INIT`; `AX_SR_SYM_INIT.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`)
- `QI` ⟶ `NumericValue_IDA.QI` and `QI` ⟶ `AX_SR_SYM_INIT.QI`
- `NumericValue_IDA.QO` ⟶ `AND_QO.IN1`; `AX_SR_SYM_INIT.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technical Details

- Initial value: `AX_SR_SYM_INIT` is parameterised with `Q_INIT=TRUE`
- The checkbox is read raw by ObjectID; there is no scaling and no persistence
- Version 1.0 (2026-09-27): acts like an `AX_PERMIT` to gate the entire on/off of the filling automation

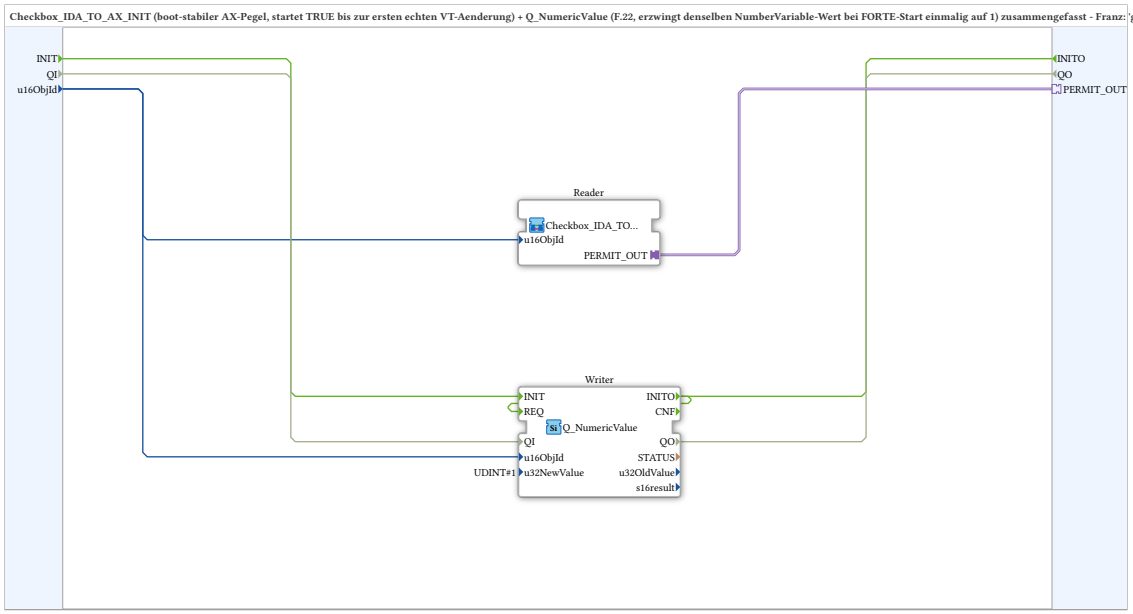
Application Scenarios

- Enabling (`PERMIT`) an automation via a VT checkbox, e.g. `Automatik_Toggle.PERMIT`

Summary

`Checkbox_IDA_TO_AX_INIT` reads a VT checkbox via its NumberVariable and provides the state as an `AX` level with a defined initial value.

Checkbox_IDA_TO_AX_INIT_Force1



• — ○ • — ○ • — ○ •

Introduction

`Checkbox_IDA_TO_AX_INIT_Force1` combines `Checkbox_IDA_TO_AX_INIT` (boot-stable AX level, starts `TRUE` until the first real VT change) and `Q_NumericValue` (forces the value 1 of the associated Number-Variable once at FORTE start). Both blocks read/write the same `u16ObjId`, hence they belong together; the `u16ObjId` is a parameter for reuse. The SubApp was extracted from `Aufnahme_Automatik.SUB` (`Permit_Automatik` plus `Q_NumericValue_Force1_Geschwindigkeitsregelung`).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request - to `Writer.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirmation - from `Writer.INITO`

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE to enable the service (to `Writer.QI`)
- **u16ObjId** (`UINT`, default `ID_NULL`): Object ID of the VT checkbox NumberVariable (the same object for reading and the Force1 write)

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully (from `Writer.QO`)

Adapters

- **PERMIT_OUT** (plug, `adapter::types::unidirectional::AX`): Boot-stable checkbox state (starts `TRUE`, afterwards follows the real VT checkbox)

Function Blocks (FBs) Used

- **Reader**: SubApp `MyLib_AX::sys::Checkbox_IDA_TO_AX_INIT` - provides the boot-stable checkbox state as AX
- **Writer**: `isobus::UT::Q::Q_NumericValue` with `u32NewValue=1` - fixed FB-side one-time writer (not the `InitialValue` property of the VT pool)

Program Flow and Connections

- `u16ObjId` ⟶ `Reader.u16ObjId` and `Writer.u16ObjId`
- `Writer.INITO` ⟶ `Writer.REQ`: the value 1 is written once at start.
- `INIT` ⟶ `Writer.INIT`; `Writer.INITO` ⟶ `INITO`
- `QI` ⟶ `Writer.QI`; `Writer.QO` ⟶ `QO`
- `Reader.PERMIT_OUT` ⟶ `PERMIT_OUT`

Technical Details

- Through the feedback `INITO` ⟶ `REQ`, the writer fires exactly once on initialization; afterwards the checkbox stays under the user's control.

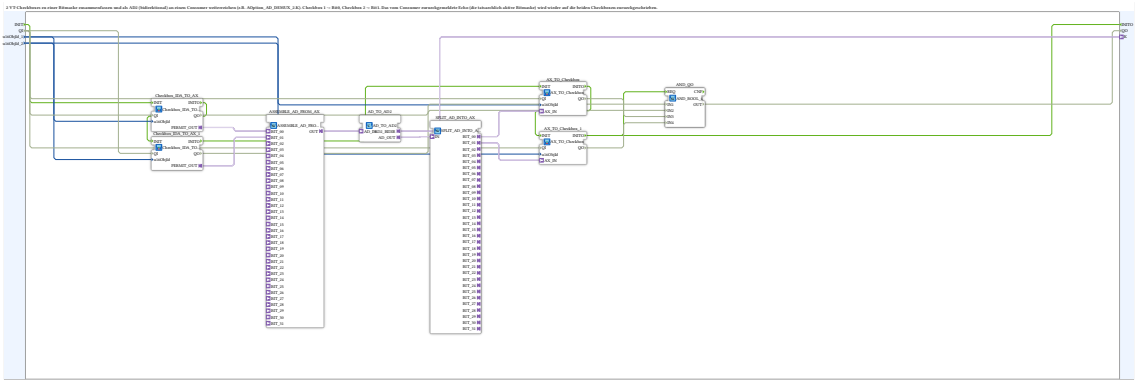
Application Scenarios

- Enable checkbox (e.g. for an automatic) that should be active at start and whose NumberVariable is set consistently to 1 at FORTE start

Summary

`Checkbox_IDA_TO_AX_INIT_Force1` provides a checkbox enable that is active at start and boot-stable as AX, and sets the associated VT object consistently to 1.

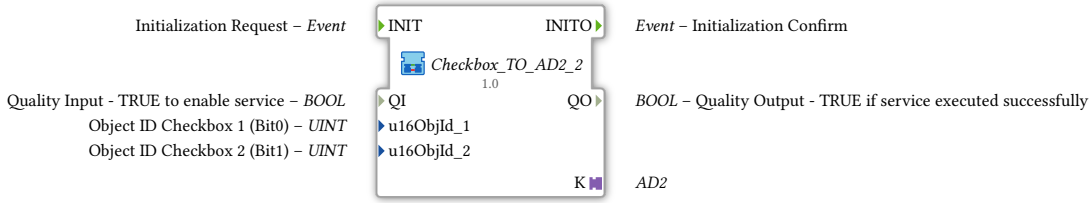
Checkbox_TO_AD2_2



• – ○ • – ○ • – ○ •

Introduction

`Checkbox_TO_AD2_2` combines two VT checkboxes into a bitmask and passes it bidirectionally as an AD2 adapter (`K`) to a consumer (e.g. `AOption_AD_DEMUX_2.K`). Checkbox 1 corresponds to bit 0, checkbox 2 to bit 1. The bitmask reported back by the consumer (the actually active one) is written back to the two checkboxes.



Interface

Event inputs

- `INIT` (`EInit`): initialization request.

Event outputs

- `INITO` (`EInit`): initialization confirm.

Input variables

- `QI` (BOOL, default `TRUE`): quality input, TRUE enables the service.

- `u16ObjId_1` (UINT, default `ID_NULL`): ObjectID of checkbox 1 (bit 0).
- `u16ObjId_2` (UINT, default `ID_NULL`): ObjectID of checkbox 2 (bit 1).

Output variables

- `QO` (BOOL): quality output, AND combination (`AND_QO`, `AND_BOOL_4`) of the `QO` outputs of the two `Checkbox_IDA_TO_AX` and the two `AX_TO_Checkbox` instances.

Adapters

- **Plug `K`** (`adapter::types::bidirectional::AD2`): bitmask to the consumer and echo back.

Initialization runs as a chain through the four sub-blocks (see below).

How it works

Forward direction (VT → consumer):

1. Two instances of `Checkbox_IDA_TO_AX` each read one checkbox and deliver an AX level (`PERMIT_OUT`).
2. `ASSEMBLE_AD_FROM_AX` assembles the levels into a DWORD (`BIT_00` and `BIT_01`).
3. `AD_TO_AD2` converts it into the bidirectional AD2 adapter, which is output on `K`.

Reverse direction (consumer → VT):

1. The echo reported back via `K` appears at `AD_TO_AD2.AD_OUT`.
2. `SPLIT_AD_INTO_AX` splits the DWORD into `BIT_00` and `BIT_01`.
3. Two instances of `AX_TO_Checkbox` write the bits to the respective checkbox.

Initialization and quality:

1. `INIT` → `Checkbox_IDA_TO_AX.INIT`; `Checkbox_IDA_TO_AX.INITO` → `Checkbox_IDA_TO_AX_1.INIT`; `Checkbox_IDA_TO_AX_1.INITO` → `AX_TO_Checkbox.INIT`; `AX_TO_Checkbox.INITO` → `AX_TO_Checkbox_1.INIT`; `AX_TO_Checkbox_1.INITO` → `INITO` and → `AND_QO.REQ`.
2. `QI` → `.QI` of all four sub-blocks; their `QO` → `AND_QO.IN1` to `AND_QO.IN4` (`Checkbox_IDA_TO_AX`, `Checkbox_IDA_TO_AX_1`, `AX_TO_Checkbox`, `AX_TO_Checkbox_1`); `AND_QO.OUT` → `QO`.

`u16ObjId_1` and `u16ObjId_2` each go to both the `Checkbox_IDA_TO_AX` and the `AX_TO_Checkbox` instance of the corresponding checkbox.

Technical notes

- Extracted from `Uebung_094b5_AX` for reuse; designed for N=2 bits/checkboxes, analogous to `AOption_AD_DEMUX_2`.
- Does not store anything itself; persistence is the consumer's responsibility.

Summary

Bundles two VT checkboxes as a bitmask onto a bidirectional AD2 plug and mirrors the consumer echo back to the checkboxes.

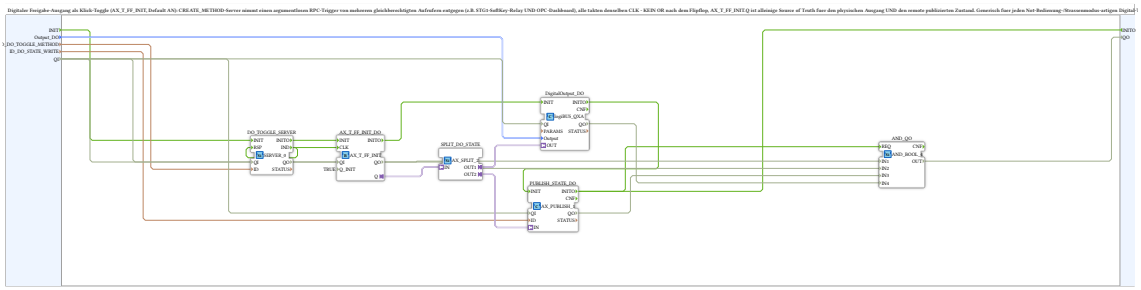
🛠️ Related Exercises

- Uebung_094b5b_AX

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

DO_TOGGLE_RPC_QXA_OPC





• — ○ • — ○ • — ○ •

Introduction

The **DO_TOGGLE_RPC_QXA_OPC** subapplication implements a digital enable output as a click-toggle (flip-flop) that can be triggered by multiple, independent, equal-right callers via an OPC UA RPC method. Typical callers are a local soft-key relay (STG1) and a remote OPC UA dashboard. Every trigger event toggles the internal state; the current state is the single source of truth for both the physical digital output and the remotely published status value. The design deliberately omits any OR-merge logic after the flip-flop, so that the flip-flop output alone determines the physical channel and the published state. This makes the block generic and reusable for any digital toggle output in emergency-operation or street-mode scenarios that require multi-access control.

Interface Structure

The subapplication exposes the initialisation events `INIT`/`INITO`, the input variables and the output `QO`; all other events, adapters, and data processing are encapsulated internally.

Event Inputs

- **INIT** (`EInit`): initialisation; sequenced by the caller instead of automatic distribution, so that `QI` is settled before the physical channel is bound; starts the initialisation chain `DO_TOGGLE_SERVER` ⟶ `AX_T_FF_INIT_DO` ⟶ `DigitalOutput_DO` ⟶ `PUBLISH_STATE_DO`

The toggle events themselves are still generated internally via the embedded OPC UA server (SERVER_0) and the internal event connections.

Event Outputs

- **INITO** (`EInit`): initialisation completed; genuine confirm from `PUBLISH_STATE_DO.INITO` (end of the initialisation chain)

**Data Inputs

Name	Type	Initial Value	Description
`QI`	`BOOL`	`TRUE`	Qualifier for the physical output; FALSE keeps the instance completely inactive (no hardware binding); passed on to `DO_TOGGLE_SERVER.QI`, `AX_T_FF_INIT_DO.QI`, `PUBLISH_STATE_DO.QI` and `DigitalOutput_DO.QI`.
`Output_DO`	`logiBUS::io::DQ::logiBUS_DO`	`logiBUS_DO::Invalid`	Configuration/description of the physical enable output channel. Drives the digital output as a click-toggle with default state **ON**.
`ID_DO_TOGGLE_METHOD`	`WSTRING`	–	OPC UA method node address (ACTION=CREATE_METHOD) for the argumentless toggle trigger. This address is shared by all callers; every caller

			clocks the same flip-flop.
ID_DO_STATE_WRITE	WSTRING	–	OPC UA variable node address (ACTION=WRITE) used to publish the actual enable state (AX_T_FF_INIT.Q). This address is subscribed by remote clients.

Data Outputs

Name	Type	Description
QO	BOOL	Quality Output - TRUE if the service executed successfully; AND (AND_QO, AND_BOOL_4) of the QO of DO_TOGGLE_SERVER, AX_T_FF_INIT_DO, PUBLISH_STATE_DO and DigitalOutput_DO.

Adapters

None exposed at the subapplication interface.

Functionality

The subapplication is built around five internal function blocks plus the AND block AND_QO:

1. **DO_TOGGLE_SERVER** (iec61499::net::SERVER_0) – an OPC UA server endpoint that listens for the method call addressed by ID_DO_TOGGLE_METHOD. The incoming request event (IND) is forwarded directly to the clock input of the toggle flip-flop and – without any processing – immediately back to the server (RSP).
2. **AX_T_FF_INIT_DO** (adapter::events::unidirectional::AX_T_FF_INIT) – a toggle flip-flop. Its parameter Q_INIT=TRUE sets the default enable state to **ON**. Each event arriving at its CLK input flips the state.
3. **SPLIT_DO_STATE** (adapter::events::unidirectional::AX_SPLIT_2) – fans out the flip-flop output to two identical copies.
4. **DigitalOutput_DO** (logiBUS::io::DQ::logiBUS_QXA) – the physical digital output driver; receives OUT1 from the split and applies it to the channel configured by Output_DO; QI comes from the subapplication interface; INIT comes from AX_T_FF_INIT_DO.INITO, and its INITO starts PUBLISH_STATE_DO.INIT.
5. **PUBLISH_STATE_DO** (adapter::net::AX_PUBLISH_1) – publishes OUT2 as the current enable state to the OPC UA variable addressed by ID_DO_STATE_WRITE; INIT comes from DigitalOutput_DO.INITO, and its INITO is output as INITO (last link of the initialisation chain).
6. **AND_QO** (iec61131::booleanOperators::AND_BOOL_4) – ANDs the QO outputs of DO_TOGGLE_SERVER, AX_T_FF_INIT_DO, PUBLISH_STATE_DO and DigitalOutput_DO into the block output QO; triggered by PUBLISH_STATE_DO.INITO (REQ).

Initialisation chain: INIT → DO_TOGGLE_SERVER.INIT, DO_TOGGLE_SERVER.INITO → AX_T_FF_INIT_DO.INIT, AX_T_FF_INIT_DO.INITO → DigitalOutput_DO.INIT, DigitalOutput_DO.INITO → PUBLISH_STATE_DO.INIT, PUBLISH_STATE_DO.INITO → INITO and → AND_QO.REQ.

The event chain for a single toggle request is:

```
DO_TOGGLE_SERVER.IND ──┬─► AX_T_FF_INIT_DO.CLK   (state flips)
                       └─► DO_TOGGLE_SERVER.RSP   (immediate acknowledgement)
```

After the flip, the new state propagates through SPLIT_DO_STATE to both the physical output and the remote publish. Because no intermediate OR logic exists, the flip-flop output is the **sole authoritative state**, and both consumers always see exactly the same value.

Technical Features

- **Single source of truth**: `AX_T_FF_INIT.Q` directly and exclusively drives both the physical output and the published OPC UA variable. There is no merging or arbitration logic on the data path.
- **Multi-caller capability**: Any number of OPC UA clients can invoke the same trigger method; they all clock the same flip-flop, so no data-level synchronization or merge block is required.
- **Argumentless remote trigger**: The RPC method takes no arguments. The server acknowledges immediately without further processing.
- **Race-free RSP handling**: The response event (`RSP`) is wired directly back to the indication event (`IND`) of the server FB. This is mandatory: if the response is not sent, FORTE's `onServerMethodCall` blocks the OPC UA server thread for up to 4 seconds (`scmMethodCallTimeoutInNanoSeconds`). During that time, open62541's global service mutex is held, which is also required by every `AX_PUBLISH_1` running on any other FORTE thread. The consequence would be that the physical output reacts immediately (no lock needed), but every OPC UA status message – not only for this block – would stall for up to 4 seconds.
- **Robust default state**: With `Q_INIT=TRUE`, the output is ON at initialization, which is the safe default for enable-type outputs.
- **Invalid channel protection**: The `Output_DO` input is initialized to `logiBUS_DO::Invalid`; the physical output driver guards against an unconfigured channel.

State Overview

The toggle state machine has two stable states, both reflected simultaneously on the physical output and the published variable:

State	Physical Output	Published Variable	Meaning
ON (initial)	Active (1)	TRUE	Enable active / output released
OFF	Inactive (0)	FALSE	Enable inhibited / output blocked

Each valid trigger event on the OPC UA method transitions between these two states. No internal intermediate states exist; the transition is atomic from the perspective of both consumers.

Application Scenarios

- **Emergency-operation enable switches** (Not-Bedienung): A physical soft-key relay on a machine panel and a remote OPC UA dashboard shall both be able to toggle the same safety-related enable output. Every operator action has equal priority; no single side is master.
- **Street-mode or mode-selection toggles**: A digital output that alternates between two operating modes (e.g., manual/semi-automatic, street/workshop) and must be operable from multiple control rooms.
- **Distributed HMI panels**: Multiple HMI clients subscribe to the published state variable, ensuring all screens display the identical, unmodified flip-flop state.
- **Generic reuse**: The block can be instantiated with different `Output_DO` channel configurations and OPC UA address strings for any equipment that requires a remote-toggleable, locally-published digital enable.

Comparison with Similar Blocks

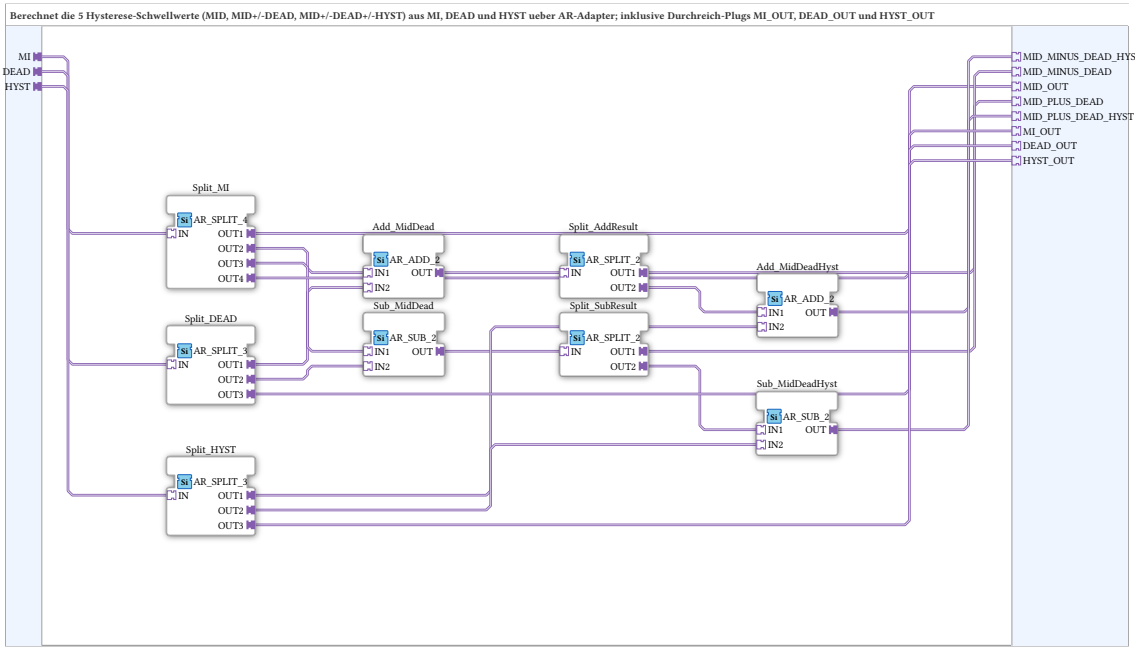
Feature	DO_TOGGLE_RPC_QXA_OPC (this block)	TOGGLE_RPC_MERGE_QXA_OPC (analogous block with merge)
Trigger handling	Single method, multiple callers all clock one flip-flop	Method triggers are merged via an OR gate before the flip-flop
Source of truth	Flip-flop output only	Merged combination of flip-flop output and trigger inputs
Data path after toggle	Straight split to output and publish	Additional OR logic on the data path
Complexity	Lower – fewer blocks, fewer connections	Higher – additional merge FB required
Use case	Truly independent callers, no data arbitration needed	Callers that may drive the output directly without toggling

The absence of the merge block in this subapplication eliminates a potential source of divergence between the physical output and the published state. It is the recommended variant whenever all callers are purely toggle-style (click on / click off) and none shall force an absolute output level.

Conclusion

DO_TOGGLE_RPC_QXA_OPC provides a clean, reliable, and reusable pattern for a remote-toggleable digital enable output that is simultaneously published to OPC UA clients. Its architecture guarantees that the locally measured physical state and the remotely observed state are always identical, because both derive from a single flip-flop with no additional logic in between. The direct RSP-to-IND wiring avoids OPC UA server thread blocking and ensures that remote publications continue to work fluently even under concurrent method calls. With its generic configuration interface (physical channel descriptor and two OPC UA address strings), the block can be dropped into any 4diac application that requires multi-operator click-toggling of a digital output.

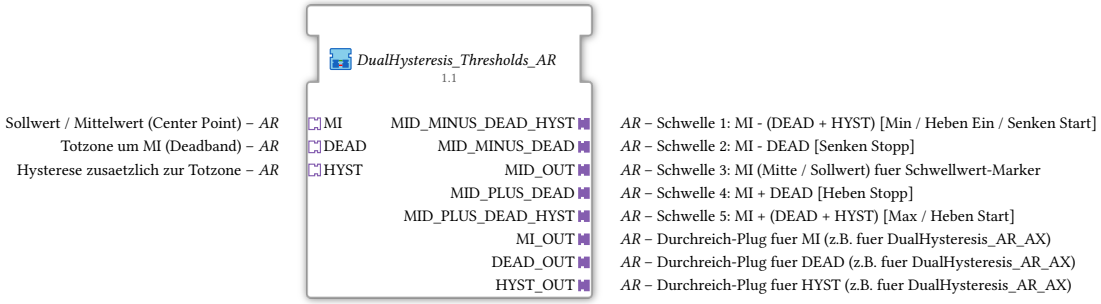
DualHysteresis_Thresholds_AR



• – ○ • – ○ • – ○ •

Introduction

`DualHysteresis_Thresholds_AR` calculates, from the set point `MI` (centre value), the dead band `DEAD` and the hysteresis `HYST`, the five thresholds of a two-way hysteresis (e.g. raise/lower) using AR adapters only. In addition, `MI`, `DEAD` and `HYST` are passed on through the pass-through plugs `MI_OUT`, `DEAD_OUT` and `HYST_OUT`.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **MI** (socket, AR): set point / centre value (centre point)
- **DEAD** (socket, AR): dead band around MI
- **HYST** (socket, AR): hysteresis in addition to the dead band
- **MID_MINUS_DEAD_HYST** (plug, AR): threshold 1 = MI - (DEAD + HYST) [min / raise on / lower start]
- **MID_MINUS_DEAD** (plug, AR): threshold 2 = MI - DEAD [lower stop]
- **MID_OUT** (plug, AR): threshold 3 = MI (centre / set point), e.g. for a threshold marker
- **MID_PLUS_DEAD** (plug, AR): threshold 4 = MI + DEAD [raise stop]
- **MID_PLUS_DEAD_HYST** (plug, AR): threshold 5 = MI + (DEAD + HYST) [max / raise start]
- **MI_OUT**, **DEAD_OUT**, **HYST_OUT** (plugs, AR): pass-through plugs for MI, DEAD and HYST (e.g. for DualHysteresis_AR_AX)

Functionality

Each input is first distributed to several consumers by a split block, because an adapter should never be branched without a split:

- MI ⟶ Split_MI (AR_SPLIT_4): OUT1 ⟶ MID_OUT, OUT2 ⟶ Add_MidDead.IN1, OUT3 ⟶ Sub_MidDead.IN1, OUT4 ⟶ MI_OUT.
- DEAD ⟶ Split_DEAD (AR_SPLIT_3): OUT1 ⟶ Add_MidDead.IN2, OUT2 ⟶ Sub_MidDead.IN2, OUT3 ⟶ DEAD_OUT.
- HYST ⟶ Split_HYST (AR_SPLIT_3): OUT1 ⟶ Add_MidDeadHyst.IN2, OUT2 ⟶ Sub_MidDeadHyst.IN2, OUT3 ⟶ HYST_OUT.

The calculation chain consists of AR_ADD_2 and AR_SUB_2 (adapter::iec61131::arithmetic):

- Add_MidDead = MI + DEAD ⟶ Split_AddResult (AR_SPLIT_2): OUT1 ⟶ MID_PLUS_DEAD, OUT2 ⟶ Add_MidDeadHyst.IN1.
- Sub_MidDead = MI - DEAD ⟶ Split_SubResult (AR_SPLIT_2): OUT1 ⟶ MID_MINUS_DEAD, OUT2 ⟶ Sub_MidDeadHyst.IN1.
- Add_MidDeadHyst = (MI + DEAD) + HYST ⟶ MID_PLUS_DEAD_HYST.
- Sub_MidDeadHyst = (MI - DEAD) - HYST ⟶ MID_MINUS_DEAD_HYST.

Technical Details

- Pure calculation/wiring block without state of its own; the outputs follow the inputs
- MI_OUT is separate from the threshold MID_OUT (version 1.1: Split_MI extended from AR_SPLIT_3 to AR_SPLIT_4) so that every plug has its own source
- The labels in square brackets (raise/lower) are the intended use of the thresholds in the caller logic

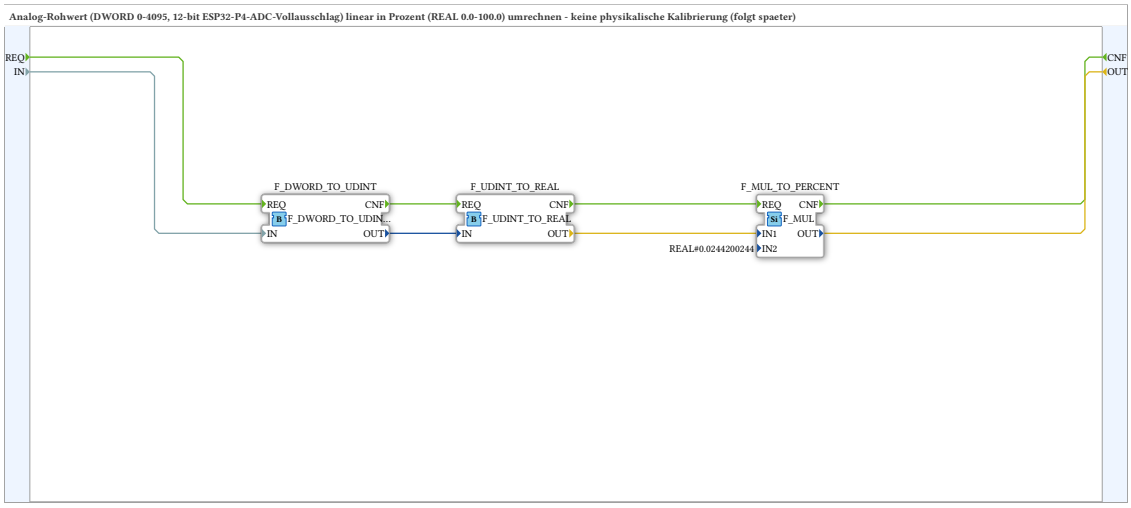
Application Scenarios

- Two-way control (raise/lower) with dead band and hysteresis, e.g. filling belt or slope compensation height control, including threshold markers on the VT

Conclusion

DualHysteresis_Thresholds_AR supplies, from MI, DEAD and HYST, the five thresholds MI ± DEAD and MI ± (DEAD + HYST) as well as MI itself using AR adapter arithmetic and passes the three input quantities on unchanged.

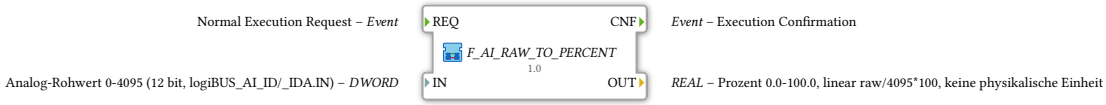
F_AI_RAW_TO_PERCENT



• – ○ • – ○ • – ○ •

Introduction

`F_AI_RAW_TO_PERCENT` converts the raw analog value from `logiBUS_AI_ID`/`logiBUS_AI_IDA` (DWORD, 0-4095, 12-bit ESP32-P4 ADC full scale) **linearly** into percent (REAL 0.0-100.0) — no physical calibration (planned as a later step). Data-based variant; the fully adapter-based alternative is `F_AI_RAW_TO_PERCENT_AD`.



Function Blocks (FBs) Used

Sub-blocks: F_AI_RAW_TO_PERCENT

- **Type**: SubAppType
- **Internal FBs used**:
  - **F_DWORD_TO_UDINT**: `iec61131::conversion::F_DWORD_TO_UDINT` — bit-reinterpretation DWORD⟶UDINT, valid here (same 32-bit unsigned-integer representation).
  - **F_UDINT_TO_REAL**: `iec61131::conversion::F_UDINT_TO_REAL` — a real numeric cast UDINT⟶REAL.
  - **F_MUL_TO_PERCENT**: `iec61131::arithmetic::F_MUL` — multiplication by `REAL#0.0244200244` (= 100/4095).
- **Functionality**: `IN` (raw value 0-4095) ⟶ UDINT ⟶ REAL ⟶ ×(100/4095) = percent.

Program Flow and Connections

1. `IN` ⟶ `F_DWORD_TO_UDINT.IN` ⟶ `F_UDINT_TO_REAL.IN` ⟶ `F_MUL_TO_PERCENT.IN1`.
2. `F_MUL_TO_PERCENT.IN2 = REAL#0.0244200244` (parameter, = 100/4095).
3. `F_MUL_TO_PERCENT.OUT` ⟶ `OUT` (percent 0.0-100.0).

Technical Details

- **Numerically correct chain**: DWORD⟶UDINT is a valid bit-reinterpretation (see Numeric vs. bitwise), UDINT⟶REAL is a real numeric cast — together they form a correct DWORD⟶REAL conversion, avoiding the bit-reinterpretation trap of `F_DWORD_TO_REAL`.
- **No physical calibration**: The factor 100/4095 only yields the linear percentage of ADC full scale, not a physical unit (volts, bar, etc.) — per the source comment, that's a later extension step.

Application Scenarios

- Displaying/processing a raw analog input value as a percentage, e.g. for an AI training example.

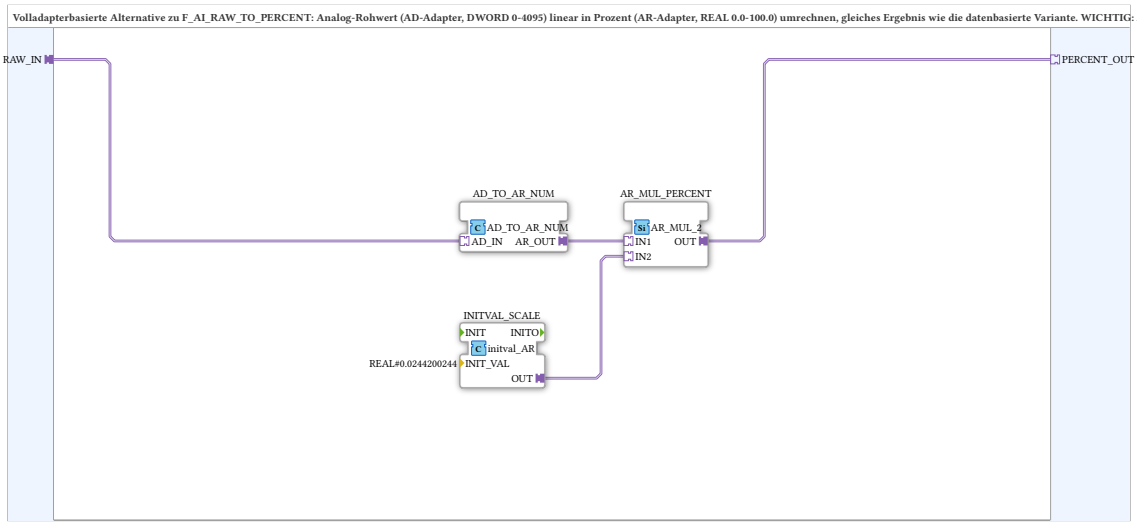
Summary

F_AI_RAW_TO_PERCENT demonstrates the numerically correct DWORD⟶REAL chain (via UDINT) on a concrete analog raw value example — data-based variant; see F_AI_RAW_TO_PERCENT_AD for the adapter-based alternative.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

F_AI_RAW_TO_PERCENT_AD

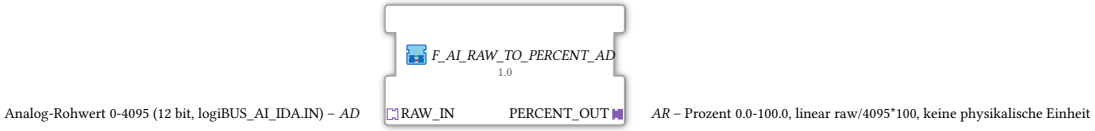


• – ○ • – ○ • – ○ •

Introduction

F_AI_RAW_TO_PERCENT_AD is the fully adapter-based alternative to F_AI_RAW_TO_PERCENT: the same raw analog value (0-4095) is converted linearly to percent via AD/AR adapters instead of regular data connections — identical result, different wiring style.

> ⚠️ **Important note from the block's own comment:** AD_TO_AR/F_DWORD_TO_REAL would be **wrong** here — that's a bit-reinterpretation (IEEE754 cast), not a numeric conversion (see forte_real.cpp, CIEC_REAL::setValue, case e_DWORD: setValueSimple). The correct path is the numeric conversion AD_TO_AR_NUM (DWORD⟶UDINT by bit-reinterpretation, valid here, same bit width/representation; then F_UDINT_TO_REAL, a real numeric conversion). Details in Numeric vs. bitwise: the FORTE conversion trap.



Function Blocks (FBs) Used

Sub-blocks: F_AI_RAW_TO_PERCENT_AD

- **Type**: SubAppType
- **Internal FBs used**:
  - **AD_TO_AR_NUM**: adapter::conversion::unidirectional::AD_TO_AR_NUM — numeric conversion DWORD adapter⟶REAL adapter via UDINT, see AD_TO_AR_NUM.
  - **INITVAL_SCALE**: adapter::types::unidirectional::AR::initval::initval_AR — supplies the constant REAL#0.0244200244 (= 100/4095) as an adapter.

  - **AR_MUL_PERCENT**: `adapter::iec61131::arithmetic::AR_MUL_2` — multiplies two REAL adapters.
- **Functionality**: `RAW_IN` (AD adapter, 0-4095) ⟶ `AD_TO_AR_NUM` ⟶ multiplied by the constant ⟶ `PERCENT_OUT` (AR adapter, 0.0-100.0).

Program Flow and Connections

1. `RAW_IN` ⟶ `AD_TO_AR_NUM.AD_IN`; `AD_TO_AR_NUM.AR_OUT` ⟶ `AR_MUL_PERCENT.IN1`.
2. `INITVAL_SCALE.OUT` (constant 0.0244200244) ⟶ `AR_MUL_PERCENT.IN2`.
3. `AR_MUL_PERCENT.OUT` ⟶ `PERCENT_OUT`.

Technical Details

- **Deliberate negative example in the comment**: the block's own comment explicitly documents why `AD_TO_AR` would be wrong here — a rare but valuable "anti-pattern" note right in the source.
- **Adapter constant instead of a parameter**: the scaling constant is supplied via `initval_AR` as an adapter rather than a classic FB parameter — consistent with the block's fully adapter-based design.

Application Scenarios

- Like `F_AI_RAW_TO_PERCENT`, but for SubApp networks built consistently on adapter connections instead of classic data connections.

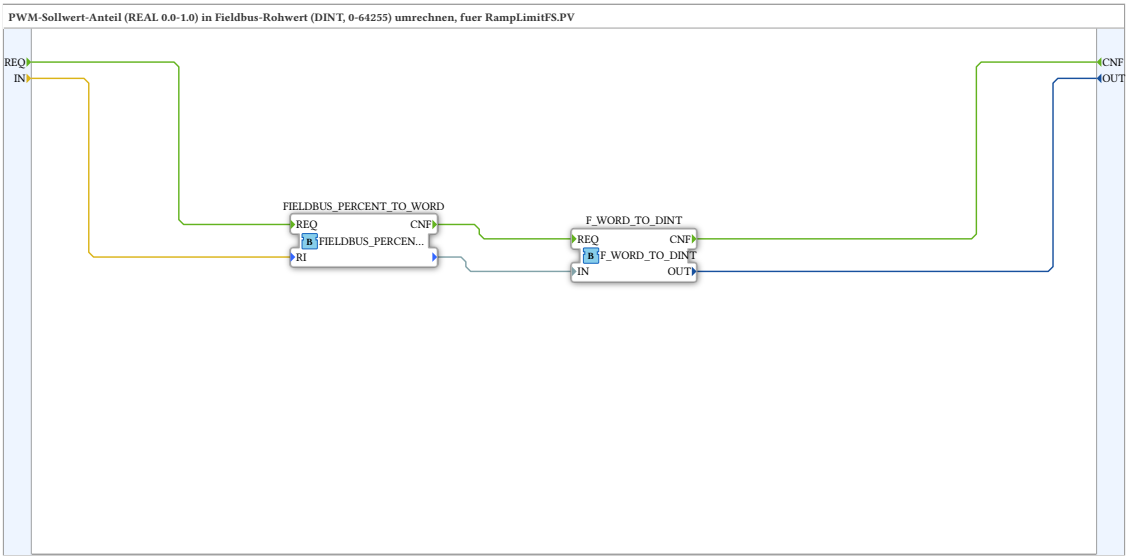
Summary

`F_AI_RAW_TO_PERCENT_AD` delivers the same numerically correct DWORD⟶REAL percent conversion as `F_AI_RAW_TO_PERCENT`, fully adapter-based — and explicitly documents the bit-reinterpretation trap a naive `AD_TO_AR` wiring would create.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

F_PWM_PERCENT_TO_RAW



• — ○ • — ○ • — ○ •

Introduction

`F_PWM_PERCENT_TO_RAW` converts a **PWM setpoint fraction (REAL 0.0–1.0)** into the **fieldbus raw value (DINT, 0–64255)** expected by `RampLimitFS.PV`. Despite its name, the block does not expect a percent value (0–100) but a fraction (0.0–1.0) — the percent-to-fraction conversion is handled upstream by `logiBUS::signalprocessing::fieldbus::F_PERCENT_TO_FRACTION`.



Function Blocks (FBs) Used

Sub-blocks: F_PWM_PERCENT_TO_RAW

- **Type**: SubAppType
- **Internal FBs used**:
  - **FIELDBUS_PERCENT_TO_WORD**: `eclipse4diac::signalprocessing::FIELDBUS_PERCENT_TO_WORD`
    - Data input: `RI` (fraction 0.0-1.0)
    - Event output: `CNF`
  - **F_WORD_TO_DINT**: `iec61131::conversion::F_WORD_TO_DINT`
    - Data input: `IN`, data output: `OUT`
- **Functionality**: The standard 4diac block `FIELDBUS_PERCENT_TO_WORD` already delivers the fieldbus raw value as a `WORD` (16-bit); `F_WORD_TO_DINT` then converts it to `DINT`, since that is the type `RampLimitFS.PV` expects.

Program Flow and Connections

1. `REQ` (SubApp event input) ⟶ `FIELDBUS_PERCENT_TO_WORD.REQ`
2. `IN` (fraction 0.0-1.0) ⟶ `FIELDBUS_PERCENT_TO_WORD.RI`
3. `FIELDBUS_PERCENT_TO_WORD.CNF` ⟶ `F_WORD_TO_DINT.REQ`, its data output ⟶ `F_WORD_TO_DINT.IN`
4. `F_WORD_TO_DINT.OUT` ⟶ `OUT` (fieldbus raw value 0-64255)
5. `F_WORD_TO_DINT.CNF` ⟶ `CNF` (SubApp event output)

Technical Details

- Uses the standard 4diac block `FIELDBUS_PERCENT_TO_WORD` (`eclipse4diac::signalprocessing`) instead of a custom formula — consistent with the SAE J1939/ISO 11783 convention `VALID_SIGNAL_W`.
- Expects a **fraction 0.0-1.0**, not percent 0-100 — easy to confuse with the similarly named `F_PERCENT_TO_FRACTION` (percent ⟶ fraction), which must be placed upstream.

Application Scenarios

- Any exercise that needs to map an analog 0-100 % setpoint onto a `RampLimitFS` or other fieldbus raw-value input (0-64255).

Summary

`F_PWM_PERCENT_TO_RAW` is a thin adapter around the standard block `FIELDBUS_PERCENT_TO_WORD` that brings the fraction-to-fieldbus-raw-value conversion to the `DINT` type expected by `RampLimitFS.PV`.

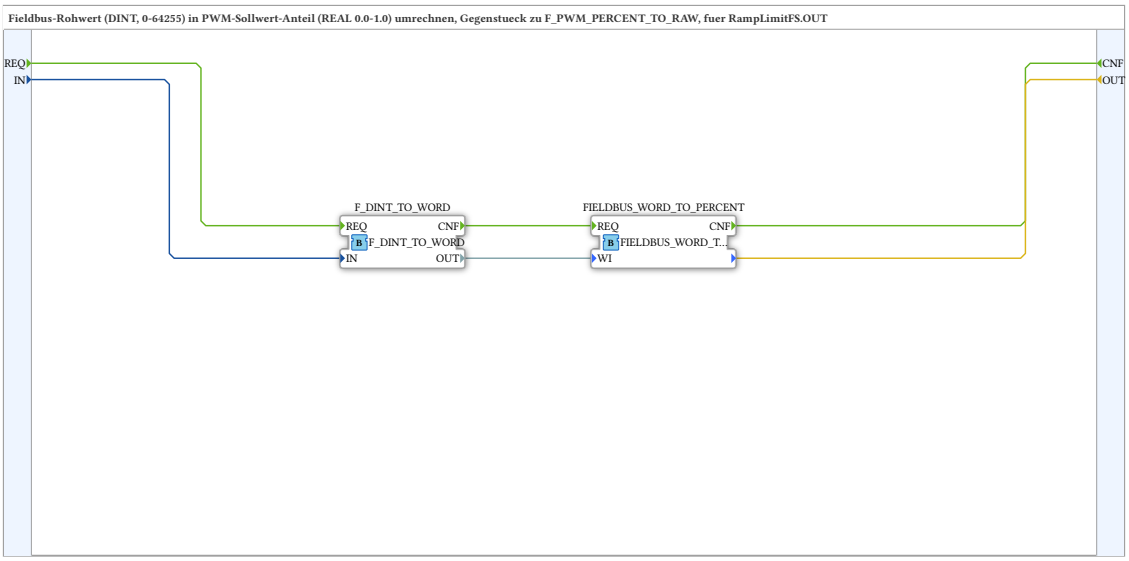
🛠️ Related Exercises

- RampLimitFS_TO_logiBUS_QDA_PWM_OPC
- F_PWM_RAW_TO_PERCENT (counterpart)
- InputOutputTesterButton_PWM_OPC_UA

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

F_PWM_RAW_TO_PERCENT



• – ○ • – ○ • – ○ •

Introduction

`F_PWM_RAW_TO_PERCENT` is the counterpart to `F_PWM_PERCENT_TO_RAW`: it converts the **fieldbus raw value (DINT, 0–64255)** from `RampLimitFS.OUT` back into a **fraction (REAL 0.0–1.0)**. Despite its name, the block does not produce a percent value — the fraction-to-percent conversion is handled downstream by `logiBUS::signalprocessing::fieldbus::F_FRACTION_TO_PERCENT`.



Function Blocks (FBs) Used

Sub-blocks: F_PWM_RAW_TO_PERCENT

- **Type**: SubAppType
- **Internal FBs used**:
  - **F_DINT_TO_WORD**: `iec61131::conversion::F_DINT_TO_WORD`
    - Data input: `IN`, data output: `OUT`
  - **FIELDBUS_WORD_TO_PERCENT**: `eclipse4diac::signalprocessing::FIELDBUS_WORD_TO_PERCENT`
    - Data input: `WI` (fieldbus raw value as `WORD`)
    - Event output: `CNF`
- **Functionality**: `RampLimitFS.OUT` delivers the raw value as `DINT`; `F_DINT_TO_WORD` converts it back to `WORD`, since the standard 4diac block `FIELDBUS_WORD_TO_PERCENT` expects that type and computes the fraction (0.0-1.0) from it.

Program Flow and Connections

1. `REQ` (SubApp event input) ⟶ `F_DINT_TO_WORD.REQ`
2. `IN` (fieldbus raw value 0-64255, DINT) ⟶ `F_DINT_TO_WORD.IN`
3. `F_DINT_TO_WORD.CNF` ⟶ `FIELDBUS_WORD_TO_PERCENT.REQ`, its data output ⟶ `FIELDBUS_WORD_TO_PERCENT.WI`
4. `FIELDBUS_WORD_TO_PERCENT` (data output) ⟶ `OUT` (fraction 0.0-1.0)
5. `FIELDBUS_WORD_TO_PERCENT.CNF` ⟶ `CNF` (SubApp event output)

Technical Details

- Uses the standard 4diac block `FIELDBUS_WORD_TO_PERCENT` (`eclipse4diac::signalprocessing`) — consistent with the SAE J1939/ISO 11783 convention `VALID_SIGNAL_W`.
- Delivers a **fraction 0.0-1.0**, not percent — for a percent display, the caller must still multiply by 100.0 or place `F_FRACTION_TO_PERCENT` downstream.

Application Scenarios

- Any exercise that needs to convert a `RampLimitFS` or other fieldbus raw value (0-64255) back into an analog 0-100 % setpoint for display or forwarding (e.g. OPC-UA publish).

Summary

`F_PWM_RAW_TO_PERCENT` is a thin adapter around the standard block `FIELDBUS_WORD_TO_PERCENT` that brings the fieldbus-raw-value-to-fraction conversion to the `DINT` type delivered by `RampLimitFS.OUT`.

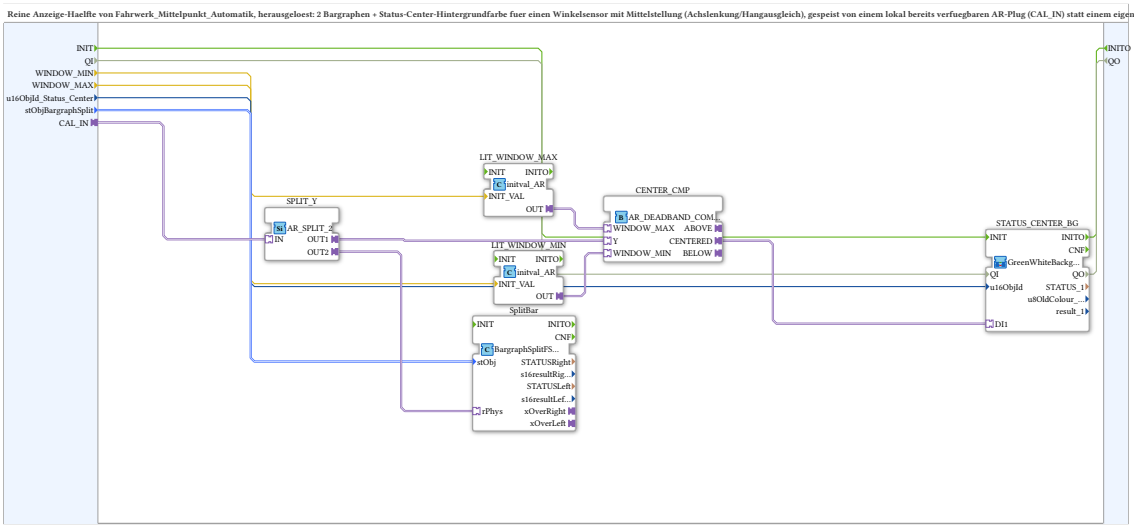
🛠️ Related Exercises

- RampLimitFS_TO_logiBUS_QDA_PWM_OPC
- F_PWM_PERCENT_TO_RAW (counterpart)
- InputOutputTesterButton_PWM_OPC_UA

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Fahrwerk_Mittelpunkt_Anzeige

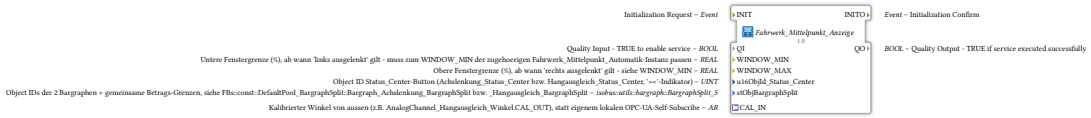


• — ○ • — ○ • — ○ •

Introduction

`Fahrwerk_Mittelpunkt_Anzeige` is the **pure display half** of `Fahrwerk_Mittelpunkt_Automatik`: two bar graphs plus a status-centre background colour for an angle sensor with a centre position (axle steering/ slope compensation). It is fed by an `AR` plug that is already available locally (`CAL_IN`) instead of by its own OPC-UA self-subscribe.

It is intended for cases where the actual automation (`AUTO_LINKS/AUTO_RECHTS` control) runs on a different station than the VT (e.g. slope compensation since the move to STG4), but a display is still needed there. No extra forwarding is necessary: the calibration channel consumer (`logiBUS_AI_Calibrated_Subscribe_VT_(3P_)OPC`) already has the value locally as the `CAL_OUT` plug, which is wired on directly here.



Function Blocks (FBs) Used

Sub-blocks: Fahrwerk_Mittelpunkt_Anzeige

- **Type**: SubAppType
- **Internal blocks used**:

- **SPLIT_Y**: `adapter::events::unidirectional::AR_SPLIT_2` - distributes the angle to two consumers
- **CENTER_CMP**: `adapter::Engineering::measurements::AR_DEADBAND_COMPARE` - window comparison
- **LIT_WINDOW_MIN**, **LIT_WINDOW_MAX**: `adapter::types::unidirectional::AR::initval::initval_AR` - convert the parameters `WINDOW_MIN`/`WINDOW_MAX` into `AR` signals
- **SplitBar**: `isobus::UT::Q::BargraphSplitFS_AR` - two bar graphs (left/right of centre)
- **STATUS_CENTER_BG**: SubApp `MyLib::sys::GreenWhiteBackground1_AX` - background colour of the `Status_Center` button

Interface Structure

Event Inputs / Event Outputs

- **INIT** (`EInit`, input): initialization request; passed on to `STATUS_CENTER_BG.INIT`
- **INITO** (`EInit`, output): initialization confirmation; comes from `STATUS_CENTER_BG.INITO`

The runtime events still travel via the adapter.

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, service enable; passed on to `STATUS_CENTER_BG.QI`
- **WINDOW_MIN** (`REAL`, default `-2.0`): Lower window limit (%) from which "deflected left" applies - must match the `WINDOW_MIN` of the associated `Fahrwerk_Mittelpunkt_Automatik` instance
- **WINDOW_MAX** (`REAL`, default `2.0`): Upper window limit (%) from which "deflected right" applies - see `WINDOW_MIN`
- **u16ObjId_Status_Center** (`UINT`): Object ID of the `Status_Center` button (`Achslenkung_Status_Center` or `Hangausgleich_Status_Center`, "><" indicator)
- **stObjBargraphSplit** (`isobus::utils::bargraph::BargraphSplit_S`): Object IDs of the two bar graphs plus common magnitude limits (see `FBs::const::DefaultPool_BargraphSplit::Bargraph_Achslenkung_BargraphSplit` or `_Hangausgleich_BargraphSplit`)

Data Outputs

- **QO** (`BOOL`): quality output, comes from `STATUS_CENTER_BG.QO`

Adapters

- **CAL_IN** (socket, `adapter::types::unidirectional::AR`): Calibrated angle from outside (e.g. `AnalogChannel_Hangausgleich_Winkel.CAL_OUT`)

Program Flow and Connections

- `CAL_IN` ⟶ `SPLIT_Y.IN`
- `SPLIT_Y.OUT1` ⟶ `CENTER_CMP.Y`; `SPLIT_Y.OUT2` ⟶ `SplitBar.rPhys`
- `WINDOW_MIN` ⟶ `LIT_WINDOW_MIN.INIT_VAL`; `LIT_WINDOW_MIN.OUT` ⟶ `CENTER_CMP.WINDOW_MIN`
- `WINDOW_MAX` ⟶ `LIT_WINDOW_MAX.INIT_VAL`; `LIT_WINDOW_MAX.OUT` ⟶ `CENTER_CMP.WINDOW_MAX`
- `CENTER_CMP.CENTERED` ⟶ `STATUS_CENTER_BG.DI1`
- `u16ObjId_Status_Center` ⟶ `STATUS_CENTER_BG.u16ObjId`
- `stObjBargraphSplit` ⟶ `SplitBar.stObj`
- Initialization: `INIT` ⟶ `STATUS_CENTER_BG.INIT`; `STATUS_CENTER_BG.INITO` ⟶ `INITO`
- Quality: `QI` ⟶ `STATUS_CENTER_BG.QI`; `STATUS_CENTER_BG.QO` ⟶ `QO`

Technical Details

- Factored out of `Fahrwerk_Mittelpunkt_Automatik` (where `SUBSCRIBE_Y`, `SPLIT_Y`, `CENTER_CMP`, `SplitBar` and `STATUS_CENTER_BG` already appeared in exactly the same way)
- The automation itself has run on STG4 (no VT) since v3.0 of the slope compensation (there `u16ObjId_Status_Center` and `stObjBargraphSplit` are deliberately `ID_NULL`); this SubApp runs on STG1 instead, fed by the `CAL_OUT` plug that is available locally anyway - no additional remote round trip
- Version 1.0 (2026-09-26)

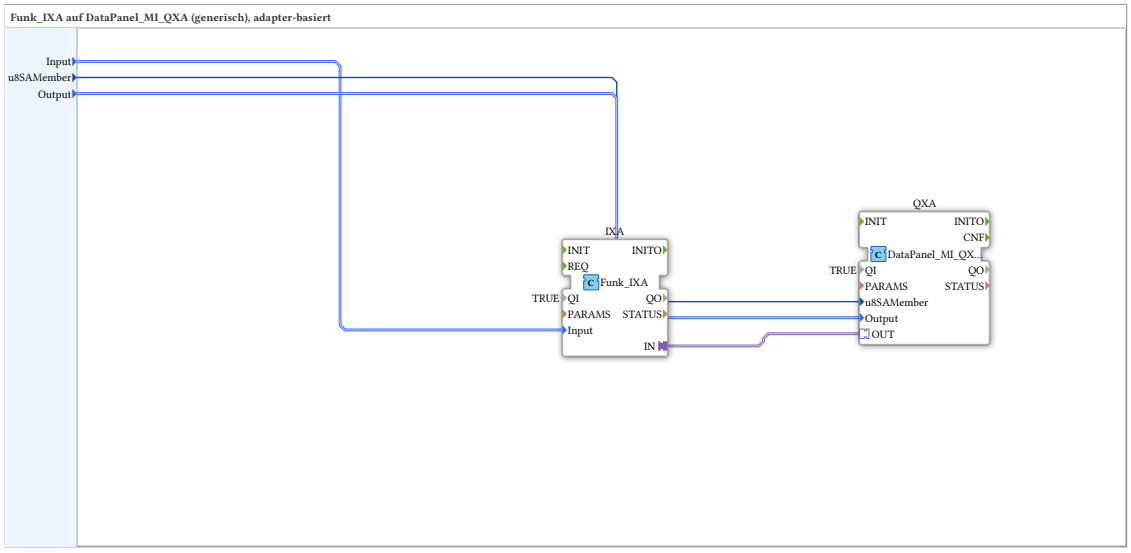
Application Scenarios

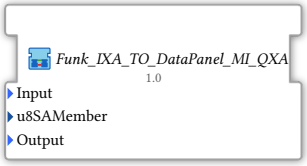
- Displaying axle steering or slope compensation (two bar graphs plus centre indicator) on a station whose control runs on another station

Summary

`Fahrwerk_Mittelpunkt_Anzeige` displays a calibrated angle as two bar graphs and as a status-centre background colour, without its own control and without an OPC-UA subscribe.

Funk_IXA_TO_DataPanel_MI_QXA





Identify the Input DigitalInput_Key_01 etc.. – *Funk::io::DI::Funk_DI_S*
Node SA 224..239 – *USINT*
Identify the Output DigitalOutput_1A..8B and Input_Power_Port_5..8 – *DataPanel::io::MI::DQ::DataPanel_MI_DO_S*

• – ○ • – ○ • – ○ •

Introduction

The **Funk_IXA_TO_DataPanel_MI_QXA** is a generic, adapter-based subapplication that bridges a Funk (wireless) digital input to a DataPanel MI QXA output. It encapsulates the necessary internal function blocks and the type-safe adapter connection between the wireless input receiver and the data panel output controller. The subapplication is designed for reuse: by changing only its data inputs, different input keys, output channels, and node addresses can be selected without modifying the internal structure.

Interface Structure

The subapplication exposes only data inputs. It has no event inputs, event outputs, data outputs, or externally visible adapters.

Event Inputs

None.

Event Outputs

None.

Data Inputs

Name	Type	Initial Value	Comment

Input	Funk::io::DI::Funk_DI_S	Funk_DI::Invalid	Identifies the input digital input key (e.g., DigitalInput_Key_01, etc.)
u8SAMember	USINT	MI::MI_00	Node SA address in the range 224..239
Output	DataPanel::io::MI::DQ::DataPanel_MI_DO_SInvalid		Identifies the output digital output (e.g., DigitalOutput_1A..8B and Input_Power_Port_5..8)

Data Outputs

None.

Adapters

The subapplication does not expose any adapters on its interface. Internally, one adapter connection is established between the two embedded function blocks:

- IXA.IN ⟶ QXA.OUT

This internal adapter connection carries the wireless input signal from the Funk receiver to the DataPanel output controller in a type-safe manner.

Functionality

The subapplication implements a direct signal path from a selected Funk input to a selected DataPanel output. The internal network consists of two function blocks:

1. **IXA** (type Funk::io::DI::Funk_IXA) – receives the configured Input selection and produces the adapter output signal IN.
2. **QXA** (type DataPanel::io::MI::DQ::DataPanel_MI_QXA) – receives the u8SAMember node address and the Output channel selection, and accepts the adapter input OUT.

The data flow is:

- Input ⟶ IXA.Input – defines which wireless input key is monitored.
- u8SAMember ⟶ QXA.u8SAMember – sets the node SA (substation address) in the 224..239 range.
- Output ⟶ QXA.Output – defines which DataPanel digital output (or power port) is actuated.
- IXA.IN ⟶ QXA.OUT – the adapter connection routes the signal from the wireless receiver to the output controller.

Both embedded FBs are enabled with their QI parameter set to TRUE. The PARAMS parameter of IXA is set to an empty string (with visibility disabled), keeping the internal configuration hidden.

Technical Features

- **Generic adapter-based design**: The subapplication can be instantiated multiple times for different input/output combinations without internal modifications.
- **Encapsulated logic**: All internal wiring and adapter connections are hidden, reducing the risk of incorrect manual wiring.
- **Type-safe adapter connection**: The internal IXA.IN ⟶ QXA.OUT connection ensures compatibility between the Funk input adapter and the DataPanel output adapter.
- **Configuration via data inputs**: The entire behavior is controlled by three data inputs (Input, Output, u8SAMember), which can be bound to constant values or runtime variables.
- **No event interface**: The subapplication operates purely on data, simplifying integration into larger IEC 61499 networks.
- **Enabled by default**: Both internal FBs have QI set to TRUE, meaning the subapplication is operational immediately after instantiation.

State Overview

The subapplication itself does not contain a state machine or event-triggered logic. Its behavior is purely combinatorial with respect to its data inputs: the selected input key, output channel, and node address are forwarded directly to the internal FBs. Any state handling is performed by the embedded function blocks

(`Funk_IXA` and `DataPanel_MI_QXA`), which manage their own internal states and diagnostics. Users of this subapplication do not need to manage any additional states.

Application Scenarios

- **Wireless key switching**: Connect a wireless digital input (e.g., a key switch) to a DataPanel digital output for remote or wireless activation of loads.
- **Generic I/O mapping**: Use the same subapplication to map multiple different Funk inputs to different DataPanel outputs by simply changing the `Input` and `Output` selection values.
- **Building automation**: Integrate wireless sensors or switches with central DataPanel output modules for flexible lighting, HVAC, or security control.
- **Modular control panels**: Simplify the creation of control-panel logic by reusing this subapplication for every radio-controlled output channel.

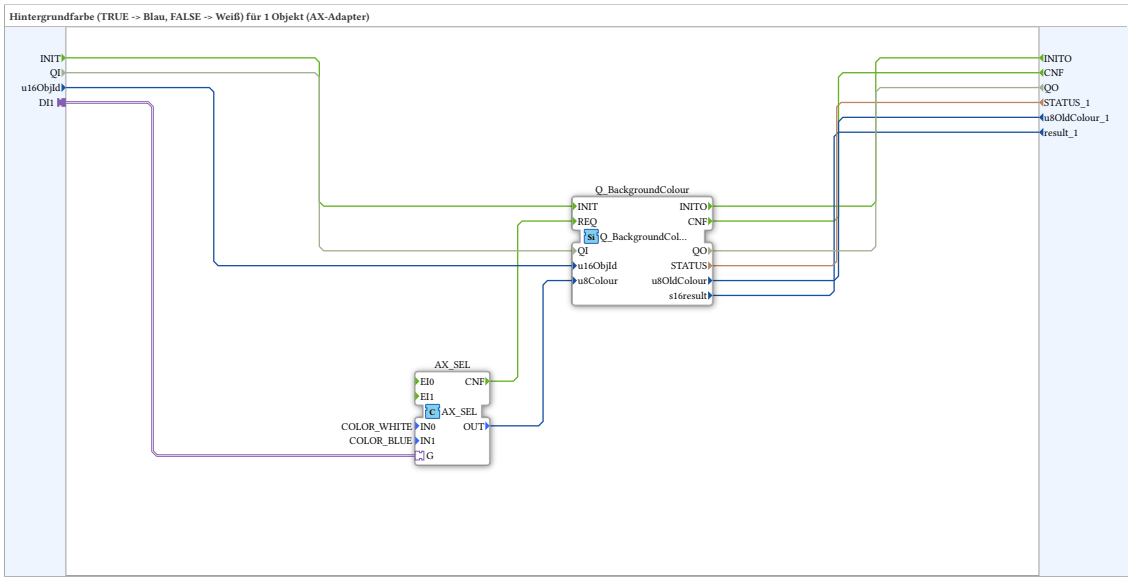
Comparison with Similar Blocks

- **vs. direct FB wiring**: Compared to manually wiring `Funk_IXA` and `DataPanel_MI_QXA` together, this subapplication encapsulates the adapter connection and exposes only the essential configuration parameters, significantly reducing wiring effort and error potential.
- **vs. an application-specific FB**: Unlike a dedicated, hard-coded function block, this subapplication is generic and can be instantiated with different parameter values, making it more flexible and reusable across projects.
- **vs. event-driven blocks**: This subapplication has no event interface, which makes it simpler to use, but it is not suitable for time-critical or sequence-dependent logic that requires explicit event synchronization.
- **vs. other mapping subapps**: This subapplication is specifically tailored for the Funk-to-DataPanel adapter pair; other mapping subapps for different device families would have a different internal structure and interface.

Conclusion

Funk_IXA_TO_DataPanel_MI_QXA offers a clean, reusable, adapter-based solution for bridging Funk wireless inputs to DataPanel MI QXA outputs. Its minimal interface – three data inputs – and fully encapsulated internal signal path make it easy to integrate and configure. The generic design ensures that the same subapplication can be leveraged across a wide range of wireless I/O mapping scenarios, improving consistency and maintainability in 4diac-based automation projects.

GreenBlueBackground1_AX



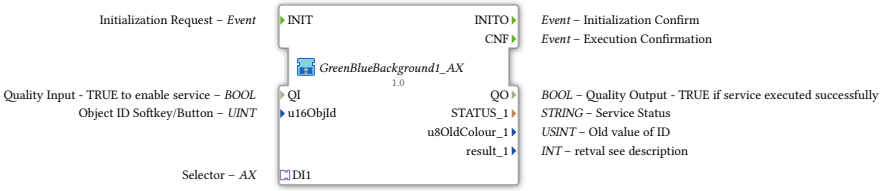
• – ○ • – ○ • – ○ •

Introduction

`GreenBlueBackground1_AX` switches the VT background color of one object based on a boolean selector signal: `TRUE` → **Blau**, `FALSE` → **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector → `AX_SEL/F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). They are passed straight through to `Q_BackgroundColour`: `INIT` → `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` → `INITO`, `QI` → `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` → `QO`.



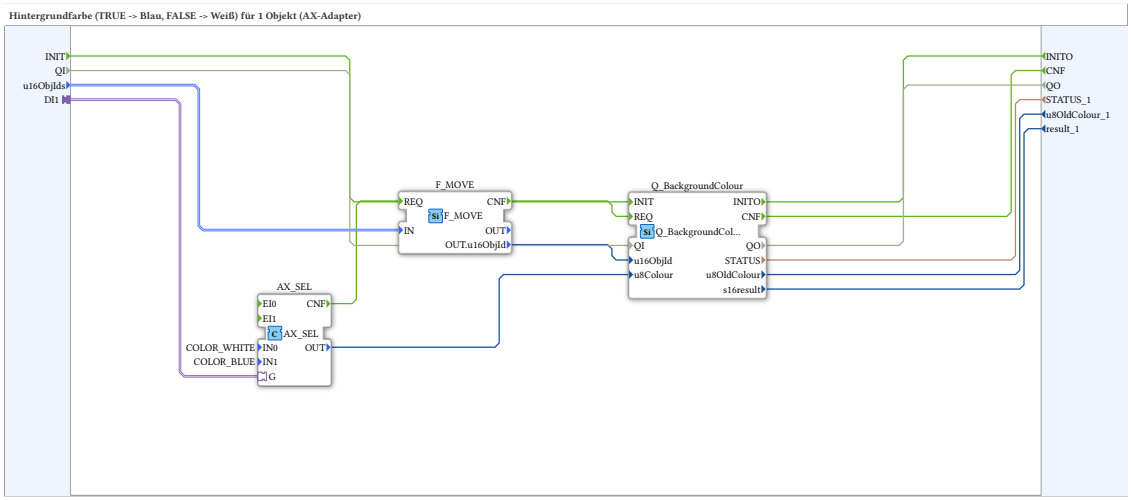
Summary

One of many variants in the background color block family: color pair Blau/Weiß, 1 object, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenBlueBackground1_AXS



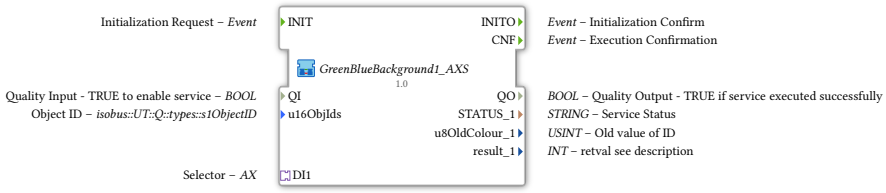
• – ○ • – ○ • – ○ •

Introduction

`GreenBlueBackground1_AXS` switches the VT background color of one object based on a boolean selector signal: `TRUE` → **Blau**, `FALSE` → **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector → `AX_SEL/F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). They are passed straight through to `Q_BackgroundColour`: `INIT` → `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` → `INITO`, `QI` → `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` → `QO`.



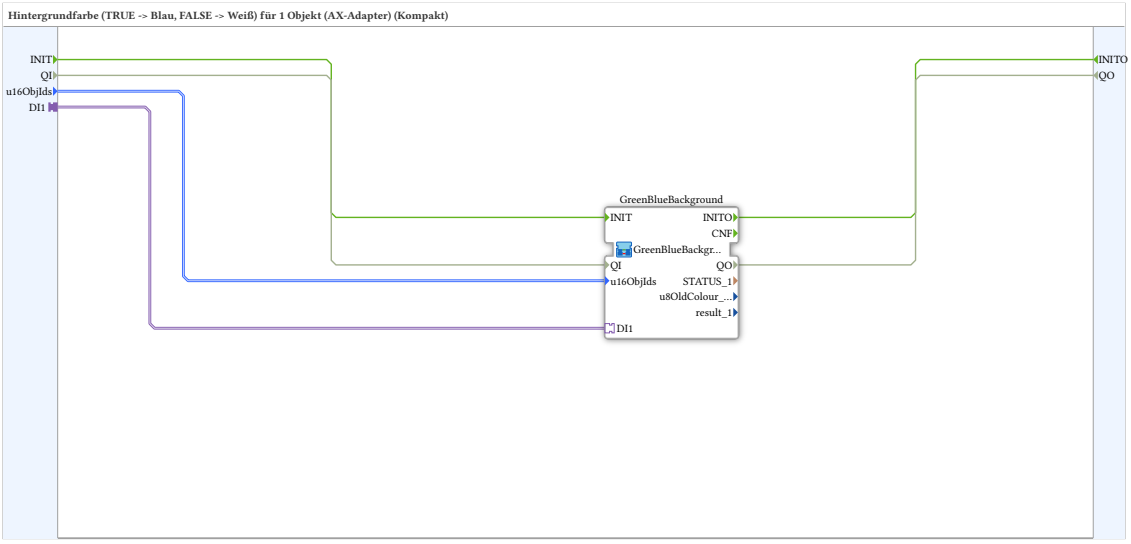
Summary

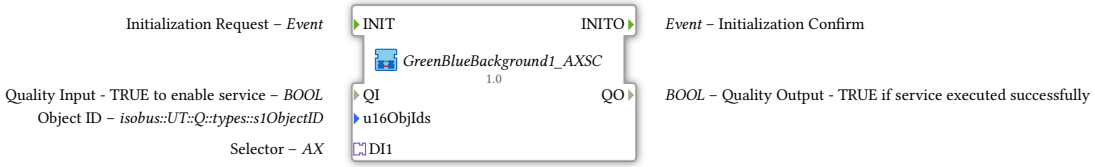
One of many variants in the background color block family: color pair Blau/Weiß, 1 object, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenBlueBackground1_AXSC





• – ○ • – ○ • – ○ •

Introduction

`GreenBlueBackground1_AXSC` is a **compact wrapper** around `GreenBlueBackground1_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

In addition, initialization and quality are passed through: `INIT` ⟶ `GreenBlueBackground.INIT`, `GreenBlueBackground.INITO` ⟶ `INITO`, `QI` (initial value `TRUE`) ⟶ `GreenBlueBackground.QI` and `GreenBlueBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

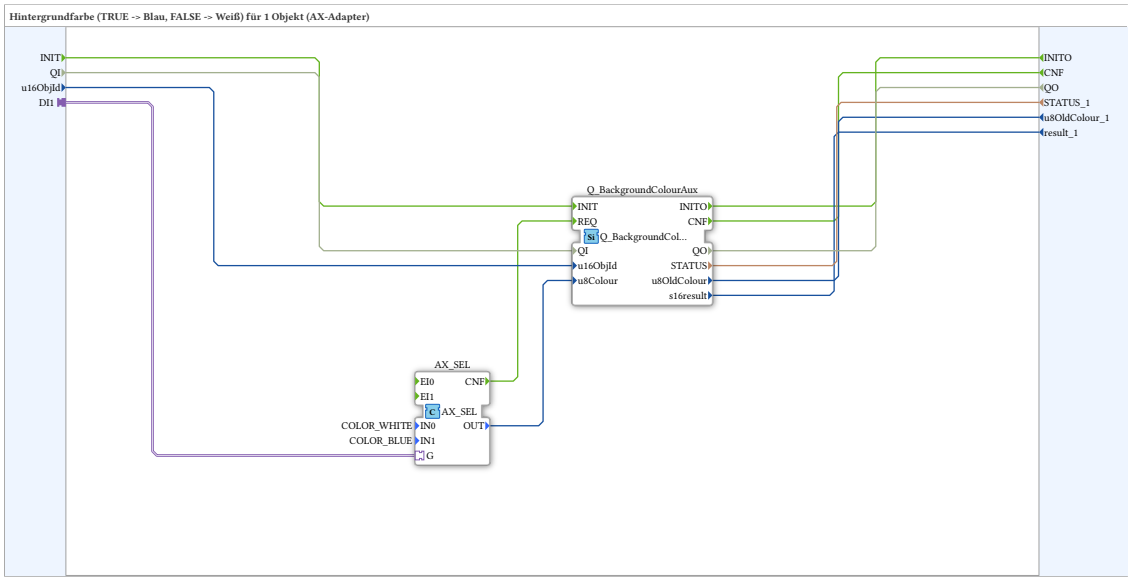
Summary

A pure pass-through to `GreenBlueBackground1_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenBlueBackground1_aux_AX



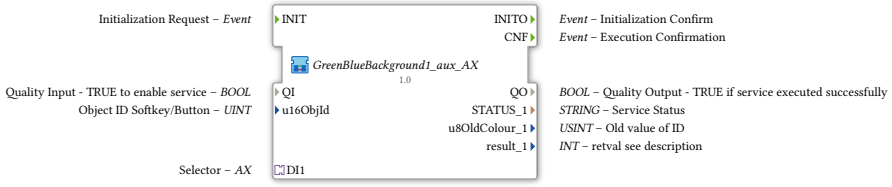
• — ○ • — ○ • — ○ •

Introduction

`GreenBlueBackground1_aux_AX` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). They are passed straight through to `Q_BackgroundColourAux`: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



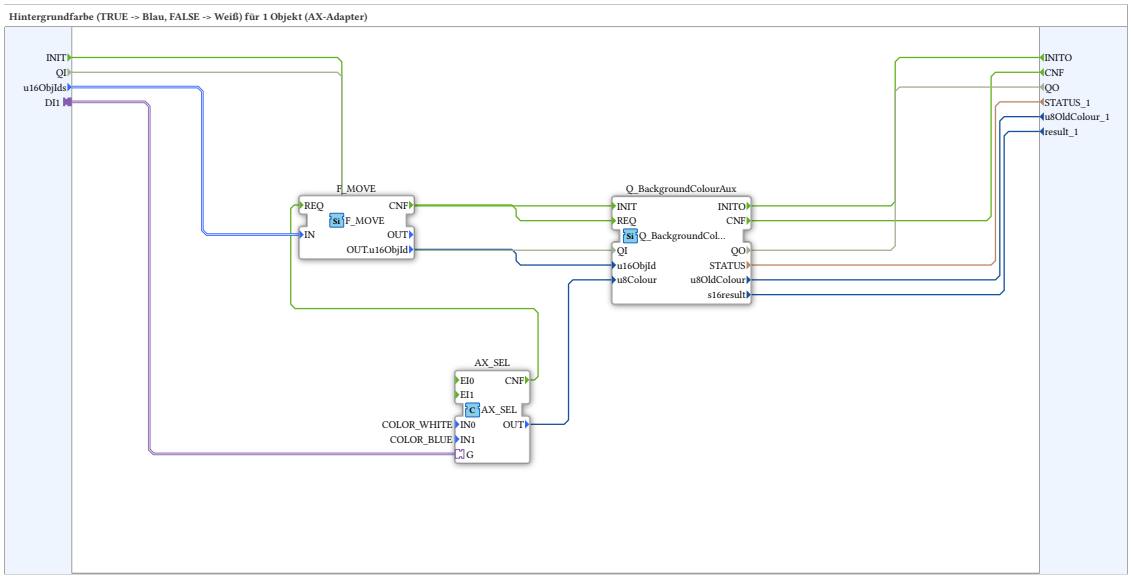
Summary

One of many variants in the background color block family: color pair Blau/Weiß, 1 object, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenBlueBackground1_aux_AXS



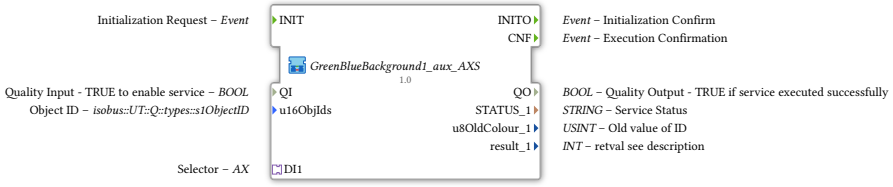
• — ○ • — ○ • — ○ •

Introduction

`GreenBlueBackground1_aux_AXS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, “Quality Input - TRUE to enable service”) and the data output `QO` (`BOOL`, “Quality Output - TRUE if service executed successfully”). They are passed straight through to `Q_BackgroundColourAux`: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



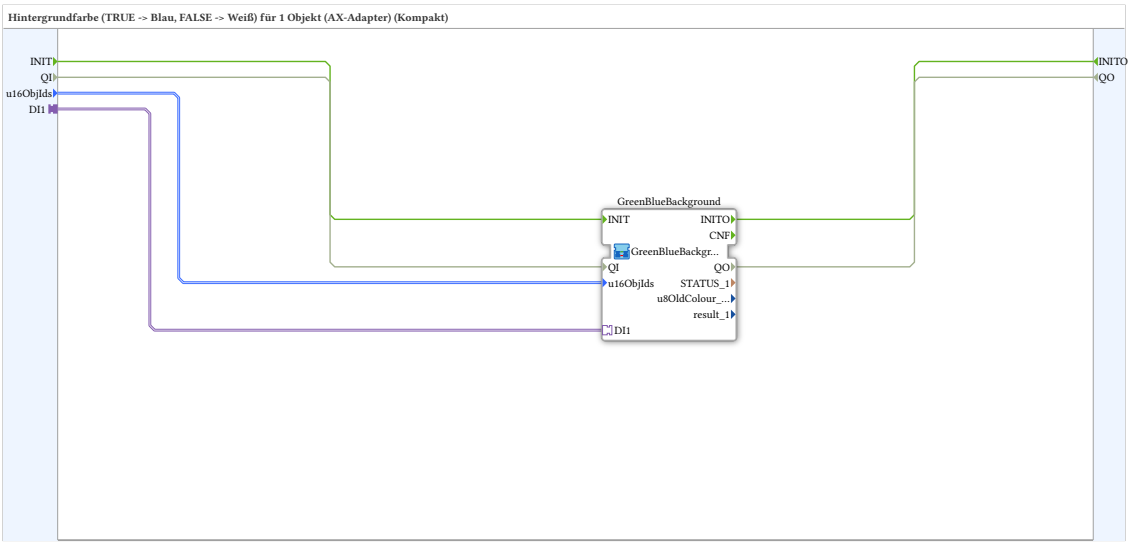
Summary

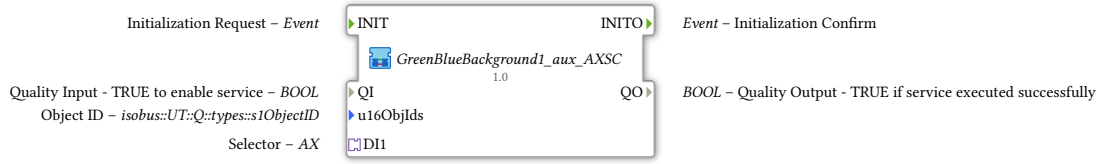
One of many variants in the background color block family: color pair Blau/Weiß, 1 object, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenBlueBackground1_aux_AXSC





• — ○ • — ○ • — ○ •

Introduction

`GreenBlueBackground1_aux_AXSC` is a **compact wrapper** around `GreenBlueBackground1_aux_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `GreenBlueBackground`: `INIT` ⟶ `GreenBlueBackground.INIT`, `GreenBlueBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenBlueBackground.QI`, `GreenBlueBackground.QO` ⟶ `QO`.

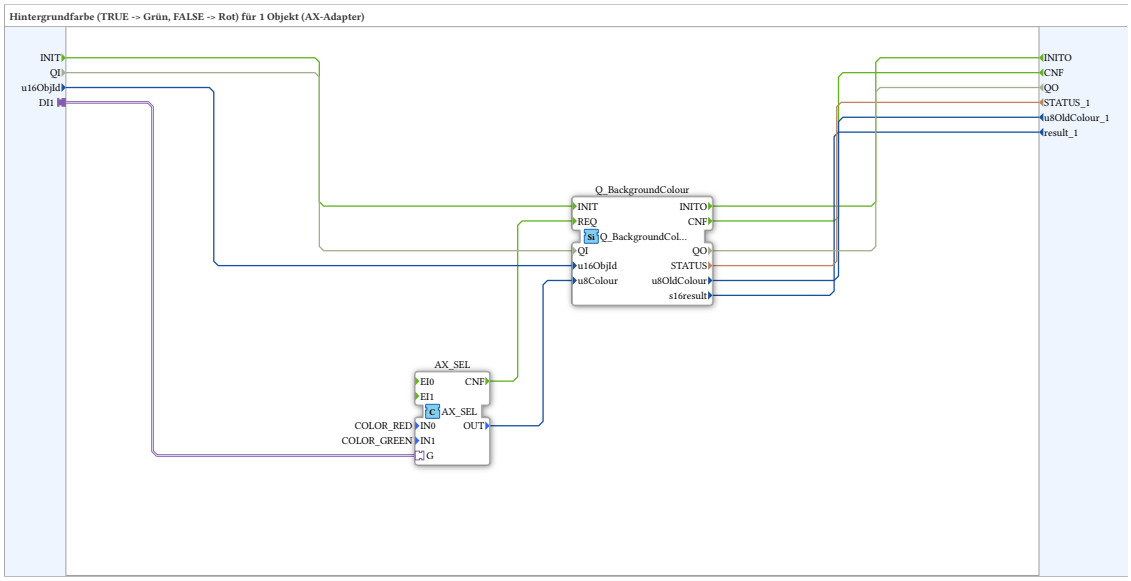
Summary

A pure pass-through to `GreenBlueBackground1_aux_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1_AX



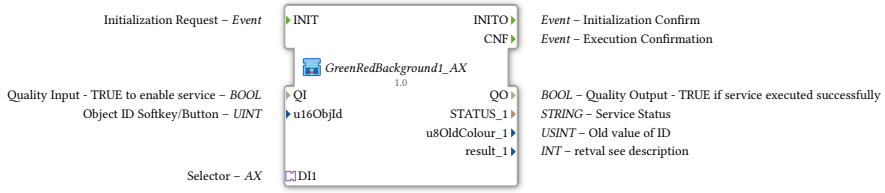
• — ○ • — ○ • — ○ •

Introduction

`GreenRedBackground1_AX` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). They are passed straight through to `Q_BackgroundColour`: `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` ⟶ `QO`.



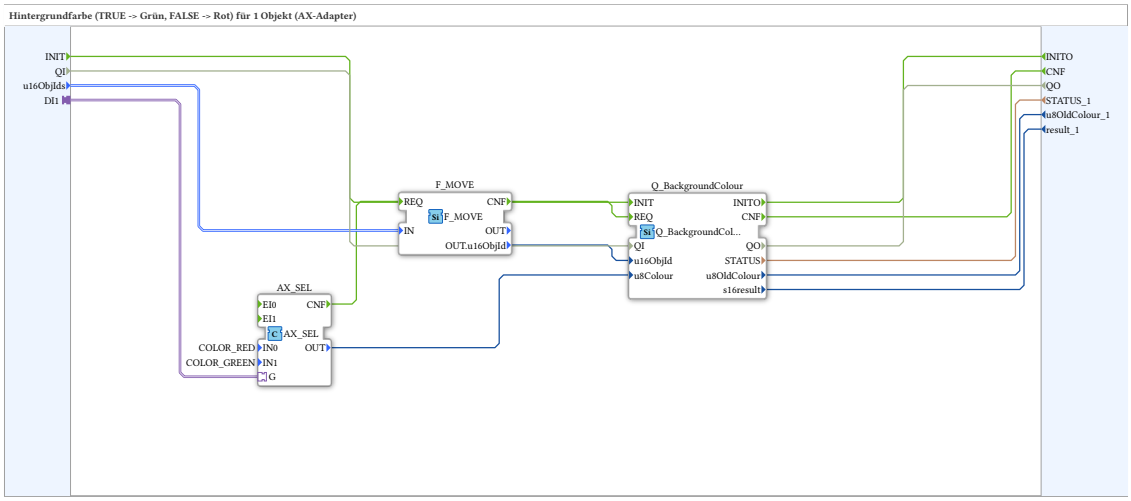
Summary

One of many variants in the background color block family: color pair Grün/Rot, 1 object, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1_AXS



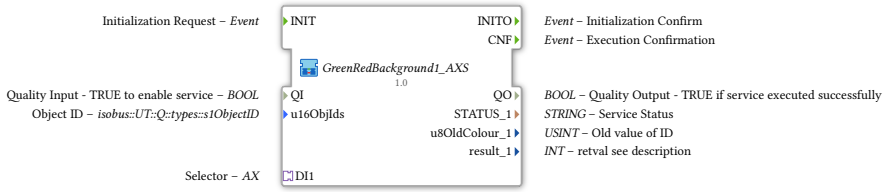
• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground1_AXS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). They are passed straight through to `Q_BackgroundColour`: `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` ⟶ `QO`.



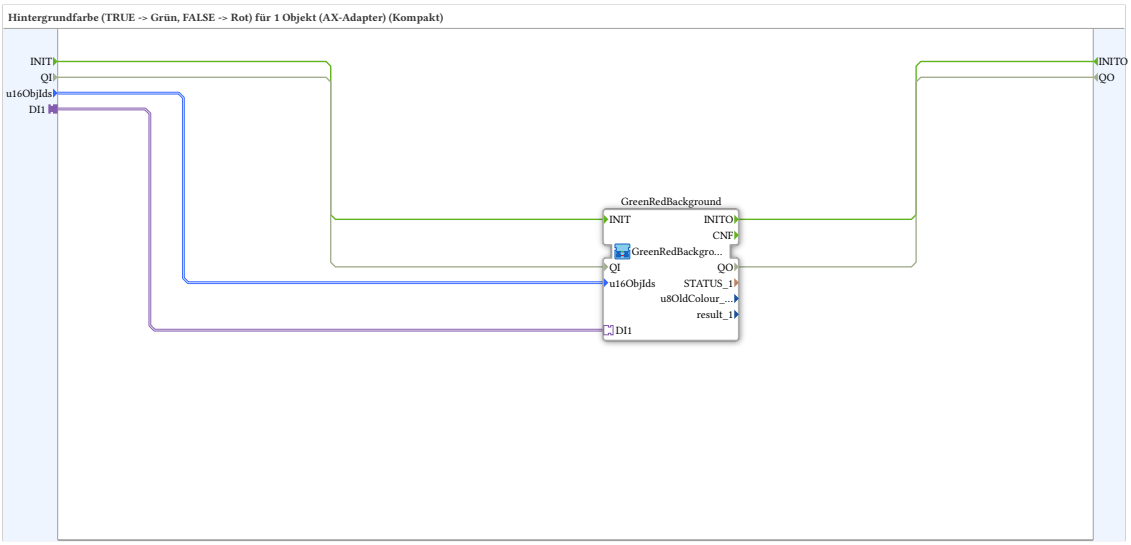
Summary

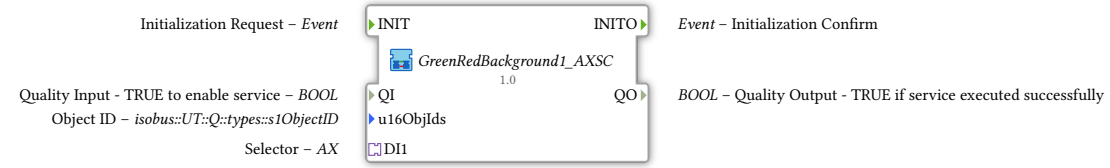
One of many variants in the background color block family: color pair Grün/Rot, 1 object, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1_AXSC





• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground1_AXSC` is a **compact wrapper** around `GreenRedBackground1_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`) are passed through: `INIT` ⟶ `GreenRedBackground.INIT`; `GreenRedBackground.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) ⟶ `GreenRedBackground.QI`; **data output `QO`** (`BOOL`) ⟵ `GreenRedBackground.QO`.

The inner SubApp instance `GreenRedBackground` is of type `GreenRedBackground1_AXS`.

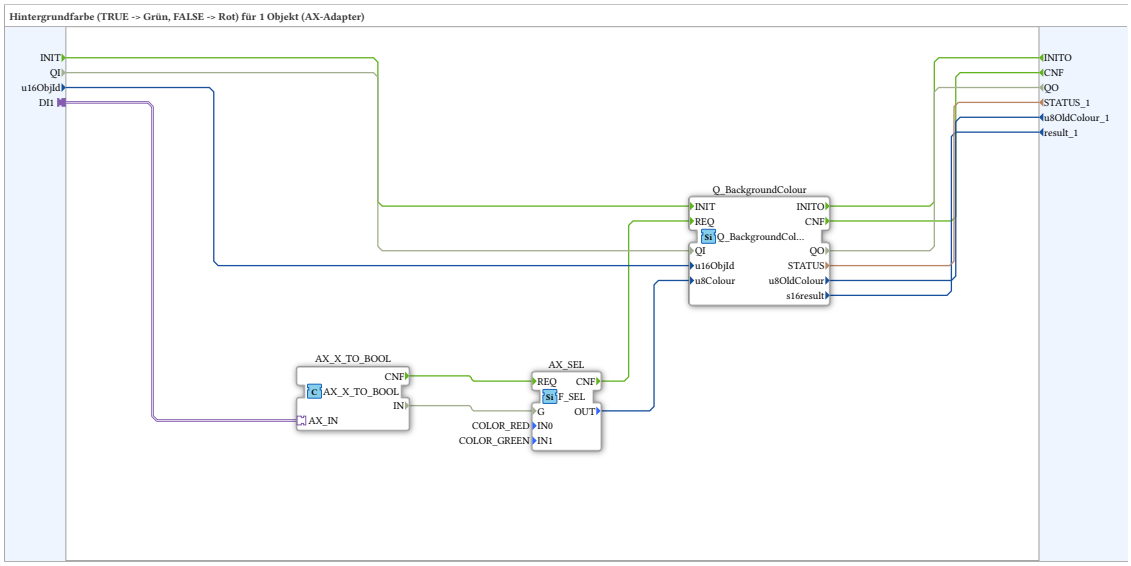
Summary

A pure pass-through to `GreenRedBackground1_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

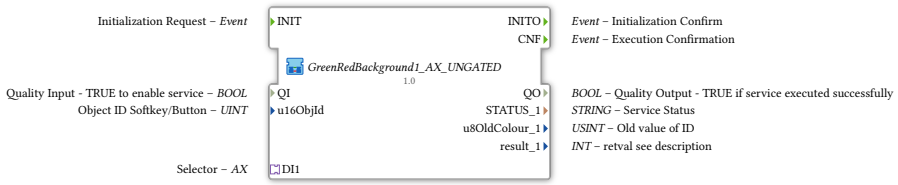
GreenRedBackground1_AX_UNGATED



• — ○ • — ○ • — ○ •

Introduction

`GreenRedBackground1_AX_UNGATED` sets the **background color** of a VT object via an AX adapter: `TRUE` ⟶ green, `FALSE` ⟶ red. "UNGATED" means that the selector is not gated by an additional enable signal - every update of `DI1` writes the color immediately.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request - to `Q_BackgroundColour.INIT`; further events arrive via the adapter socket

Event Outputs

- **INITO** (`EInit`): initialization confirmation (from `Q_BackgroundColour.INITO`)
- **CNF**: Execution confirmation (from `Q_BackgroundColour.CNF`)

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE to enable the service
- **u16ObjId** (`UINT`, default `ID_NULL`): Object ID of the softkey/button

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully (from `Q_BackgroundColour.QO`)
- **STATUS_1** (`STRING`): Service status
- **u8OldColour_1** (`USINT`): Old color value of the object
- **result_1** (`INT`): Return value (see `Q_BackgroundColour`)

Adapters

- **DI1** (socket, `adapter::types::unidirectional::AX`): Selector - `TRUE` for green, `FALSE` for red

Function Blocks (FBs) Used

- **AX_X_TO_BOOL**: `adapter::conversion::unidirectional::AX_X_TO_BOOL` - converts the AX selector into event `CNF` and value `IN` (`BOOL`)
- **AX_SEL**: `iec61131::selection::F_SEL` with `IN0 = COLOR_RED`, `IN1 = COLOR_GREEN` - selects the color
- **Q_BackgroundColour**: `isobus::UT::Q::Q_BackgroundColour` - writes the background color to the VT object

Program Flow and Connections

1. `DI1` ⟶ `AX_X_TO_BOOL.AX_IN`; `AX_X_TO_BOOL.IN` ⟶ `AX_SEL.G`; `AX_X_TO_BOOL.CNF` ⟶ `AX_SEL.REQ`.
2. `AX_SEL.OUT` ⟶ `Q_BackgroundColour.u8Colour`; `AX_SEL.CNF` ⟶ `Q_BackgroundColour.REQ`.
3. `u16ObjId` ⟶ `Q_BackgroundColour.u16ObjId`.
4. `Q_BackgroundColour.CNF` ⟶ `CNF`; `STATUS`, `u8OldColour`, `s16result` ⟶ `STATUS_1`, `u8OldColour_1`, `result_1`.
5. `INIT` ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ `INITO`; `QI` ⟶ `Q_BackgroundColour.QI`; `Q_BackgroundColour.QO` ⟶ `QO`.

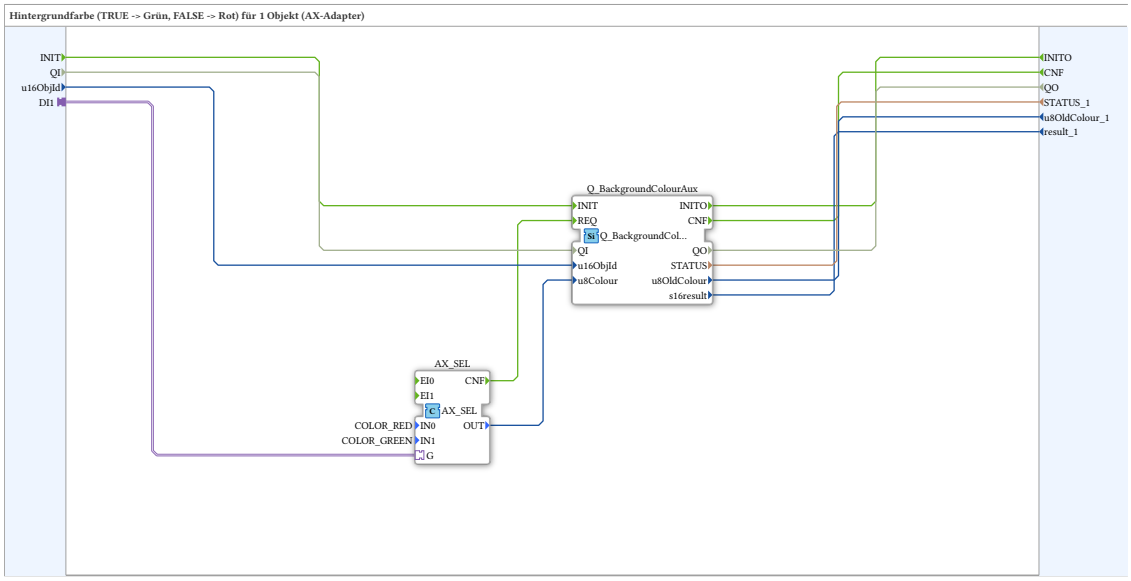
Application Scenarios

- Green/red status display for buttons or areas (e.g. operation/fault) without additional enable logic

Summary

`GreenRedBackground1_AX_UNGATED` immediately colors a VT object green or red depending on an AX signal.

GreenRedBackground1_aux_AX



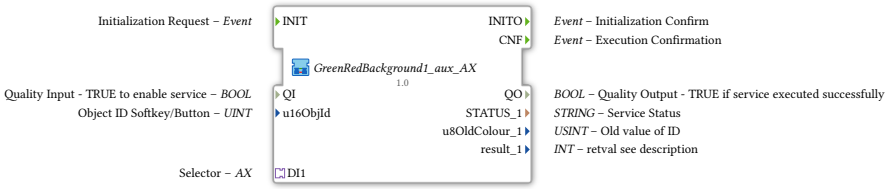
• — ○ • — ○ • — ○ •

Introduction

`GreenRedBackground1_aux_AX` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). They are passed straight through to `Q_BackgroundColourAux`: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



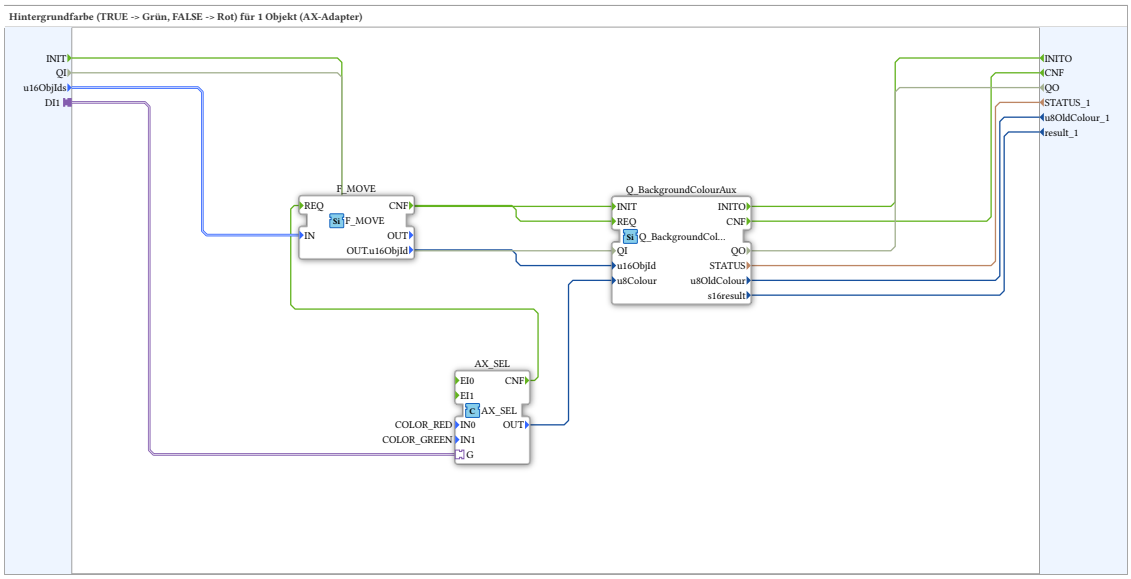
Summary

One of many variants in the background color block family: color pair Grün/Rot, 1 object, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1_aux_AXS



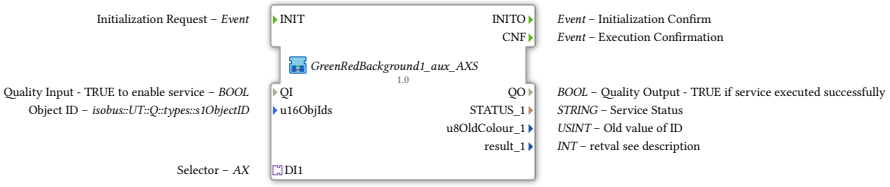
• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground1_aux_AXS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, “Quality Input - TRUE to enable service”) and the data output `QO` (`BOOL`, “Quality Output - TRUE if service executed successfully”). They are passed straight through to `Q_BackgroundColourAux`: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



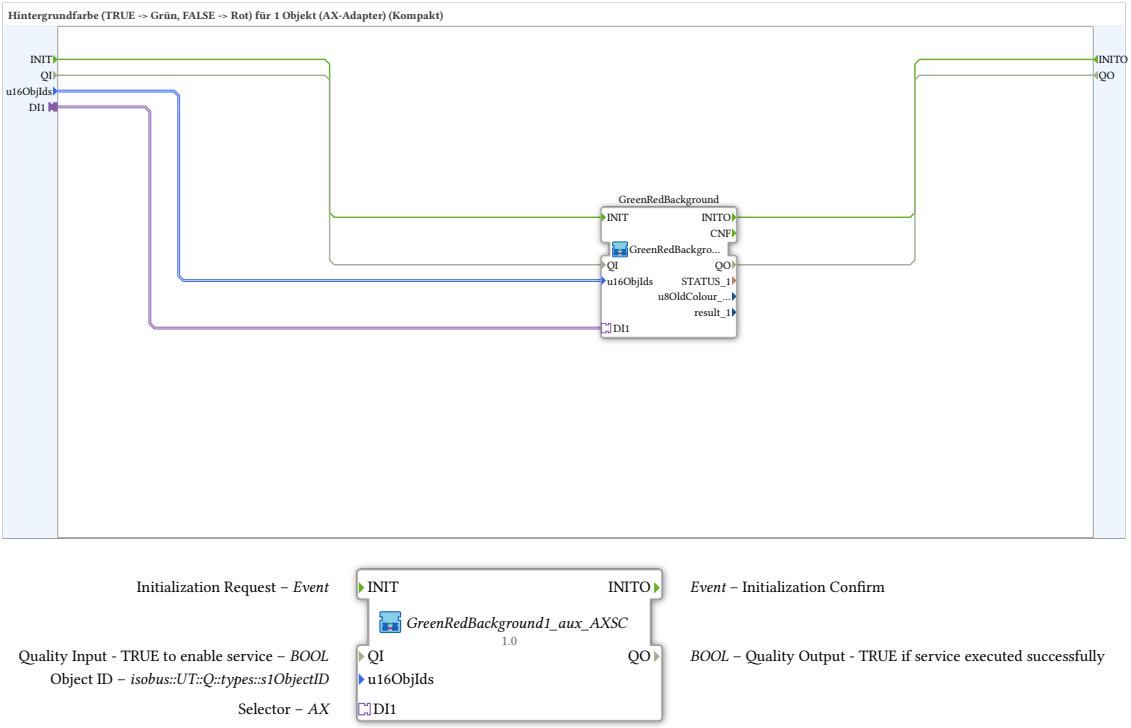
Summary

One of many variants in the background color block family: color pair Grün/Rot, 1 object, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1_aux_AXSC



• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground1_aux_AXSC` is a **compact wrapper** around `GreenRedBackground1_aux_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `GreenRedBackground`: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenRedBackground.QI`, `GreenRedBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

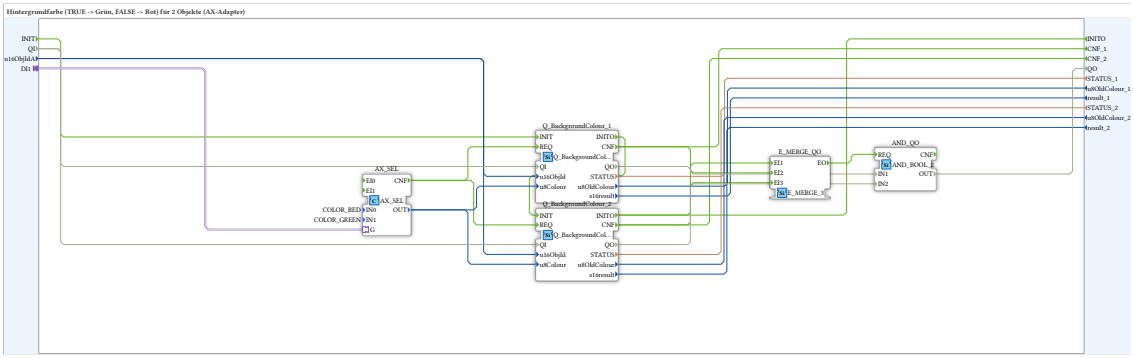
Summary

A pure pass-through to `GreenRedBackground1_aux_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground2_AX



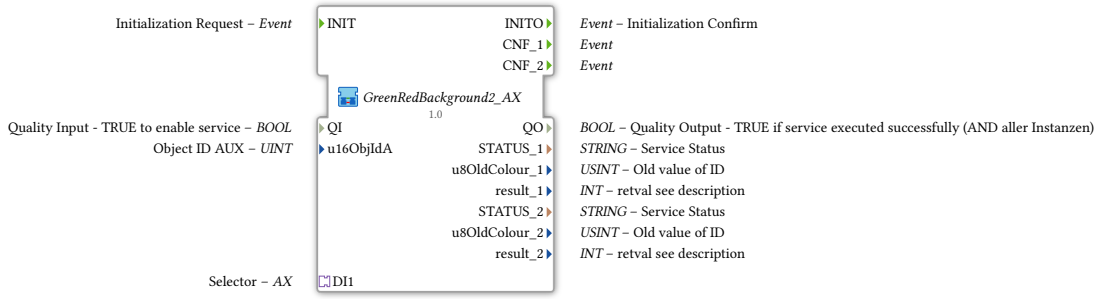
• — ○ • — ○ • — ○ •

Introduction

`GreenRedBackground2_AX` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via inputs `u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_BackgroundColour_1.QI` and `Q_BackgroundColour_2.QI`
- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_2`) of the `QO` outputs of `Q_BackgroundColour_1.QO` and `Q_BackgroundColour_2.QO`
- **Initialization chain**: `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `INITO`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_3`) merges `Q_BackgroundColour_2.INITO`, `Q_BackgroundColour_1.CNF` and `Q_BackgroundColour_2.CNF` and triggers `AND_QO.REQ`

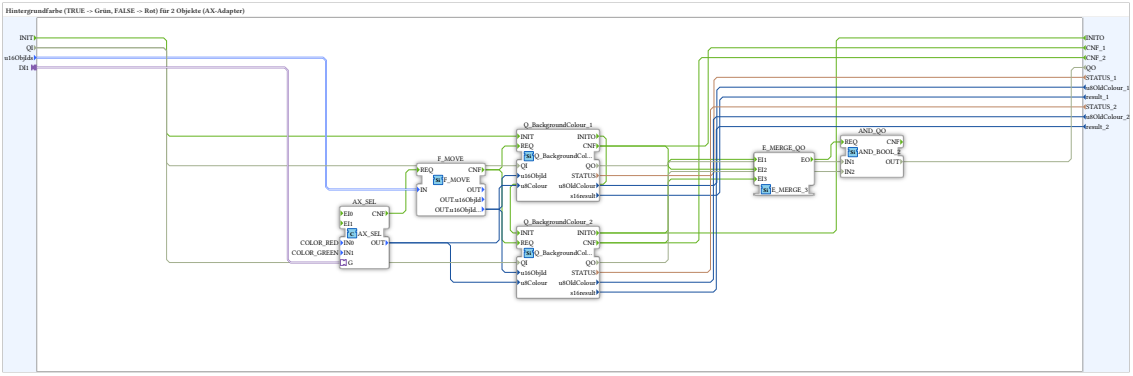
Summary

One of many variants in the background color block family: color pair Grün/Rot, 2 objects, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground2_AXS



• – ○ • – ○ • – ○ •

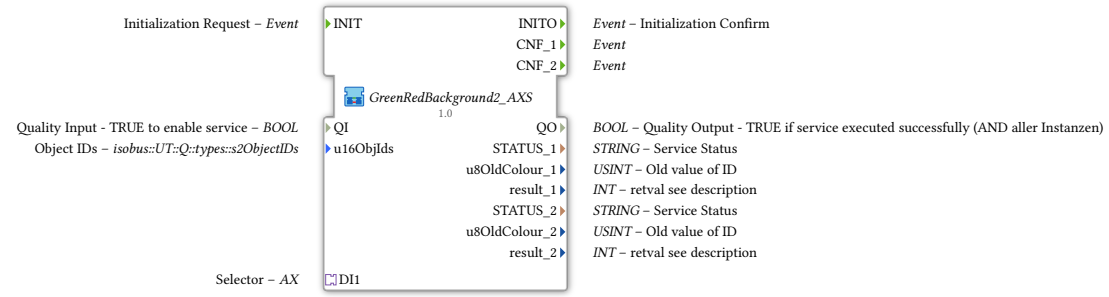
Introduction

`GreenRedBackground2_AXS` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s2ObjectIDs` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Initialization and quality: The SubApp has `INIT`/`INITO` and `QI`/`QO`. `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `INITO`. `QI` goes to both `Q_BackgroundColour` instances. `QO` is the `AND` (`AND_QO`, `AND_BOOL_2`) of the two `QO` outputs; `AND_QO.REQ` is triggered via `E_MERGE_QO` (`E_MERGE_3`: `Q_BackgroundColour_2.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`).



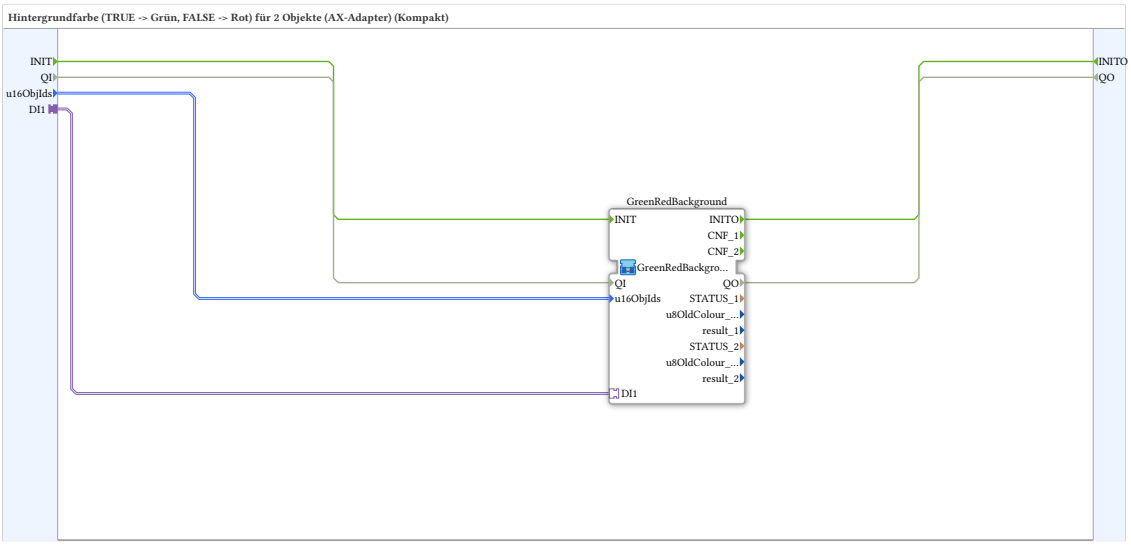
Summary

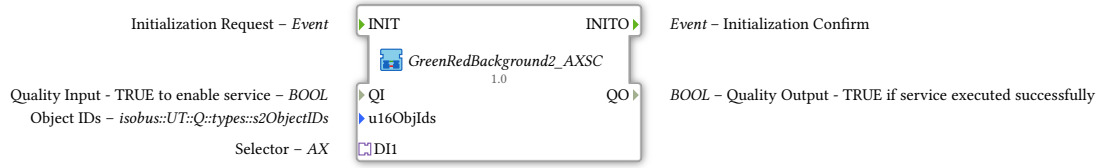
One of many variants in the background color block family: color pair Grün/Rot, 2 objects, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground2_AXSC





• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground2_AXSC` is a **compact wrapper** around `GreenRedBackground2_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

In addition, initialization and quality are passed through: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` (initial value `TRUE`) ⟶ `GreenRedBackground.QI` and `GreenRedBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

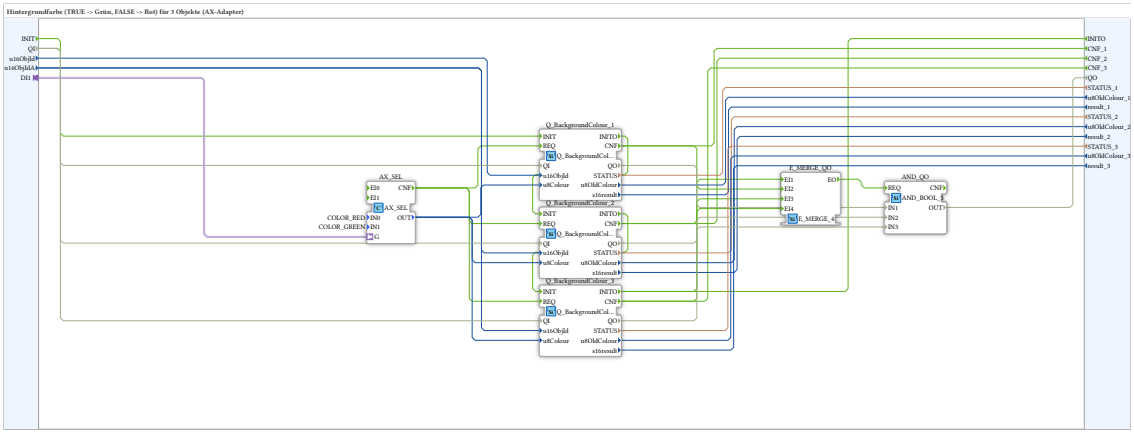
Summary

A pure pass-through to `GreenRedBackground2_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground3_AX



• — ○ • — ○ • — ○ •

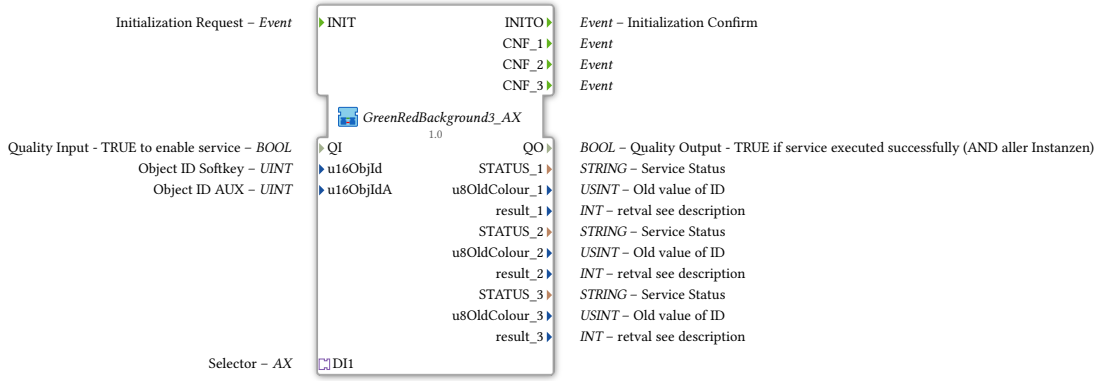
Introduction

`GreenRedBackground3_AX` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

In addition, the block has the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`, AND of all instances). `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` ⟶ `INITO` (so the three instances are initialized one after another). `QI` is passed to `QI` of all three instances. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`), which combines `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO` and `Q_BackgroundColour_3.QO`; its `REQ` is triggered by `E_MERGE_QO` (`iec61499::events::E_MERGE_4`), which merges `Q_BackgroundColour_3.INITO` and the `CNF` events of the three instances.



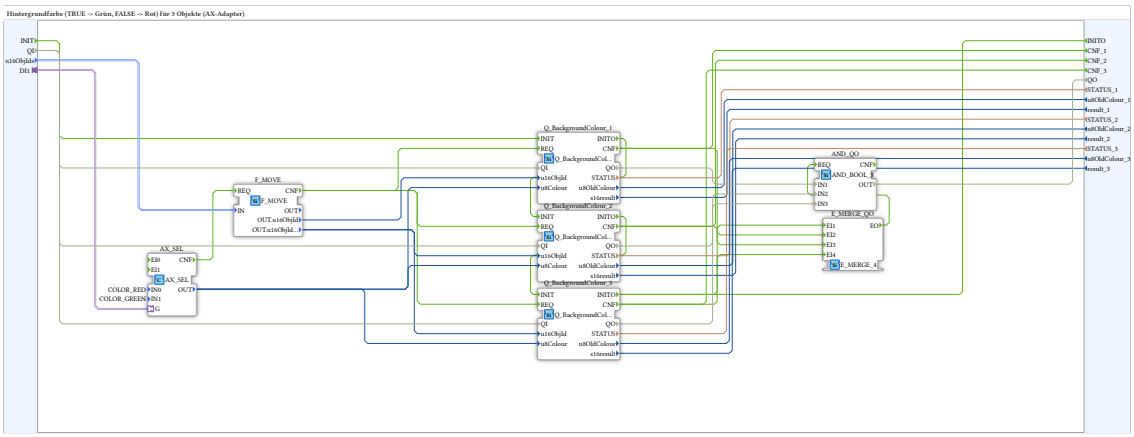
Summary

One of many variants in the background color block family: color pair Grün/Rot, 3 objects, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground3_AXS



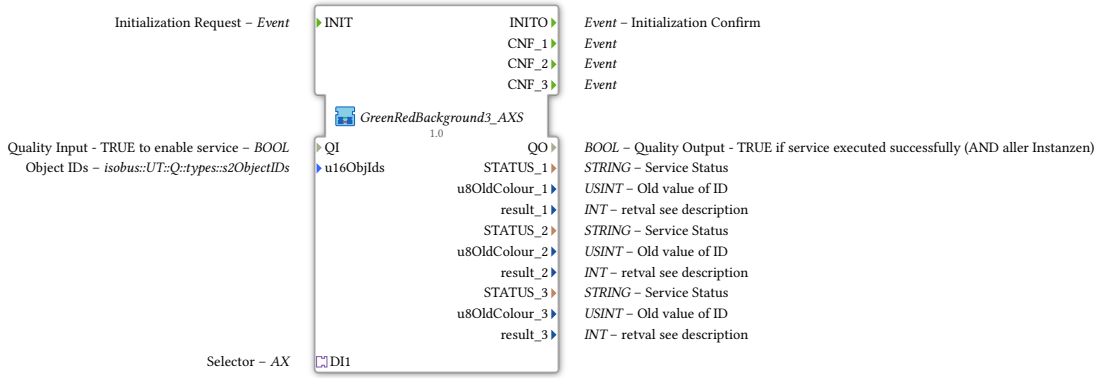
• — ○ • — ○ • — ○ •

Introduction

`GreenRedBackground3_AXS` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s2ObjectIDs` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.

- `E_MERGE_QO` (`iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_3.INITO` and the `CNF` events of all three instances (`Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`) and thereby triggers `AND_QO.REQ`.

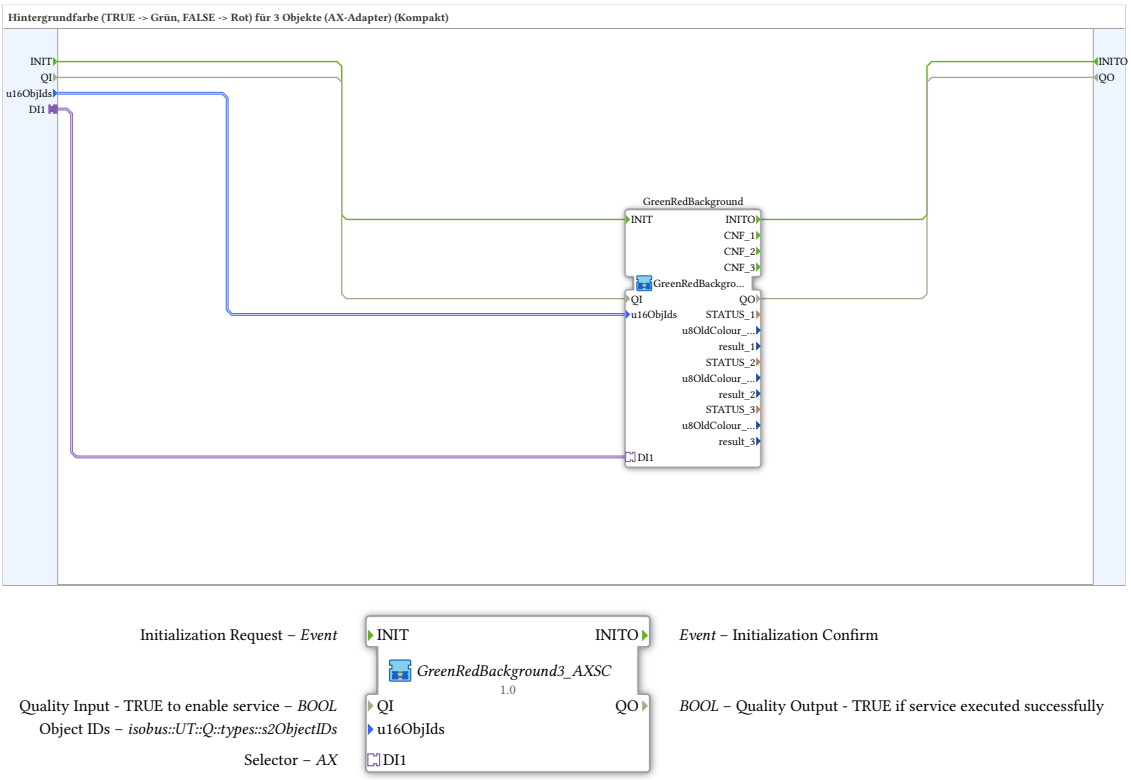
Summary

One of many variants in the background color block family: color pair Grün/Rot, 3 objects, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground3_AXSC



• — ○ • — ○ • — ○ •

Introduction

`GreenRedBackground3_AXSC` is a **compact wrapper** around `GreenRedBackground3_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `GreenRedBackground`: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenRedBackground.QI`, `GreenRedBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

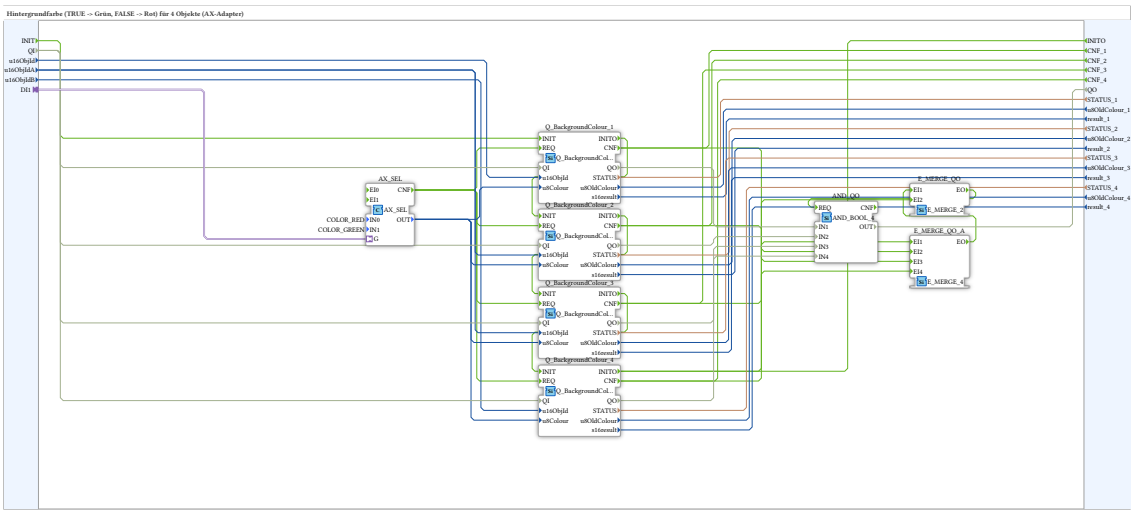
Summary

A pure pass-through to `GreenRedBackground3_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground4_AX



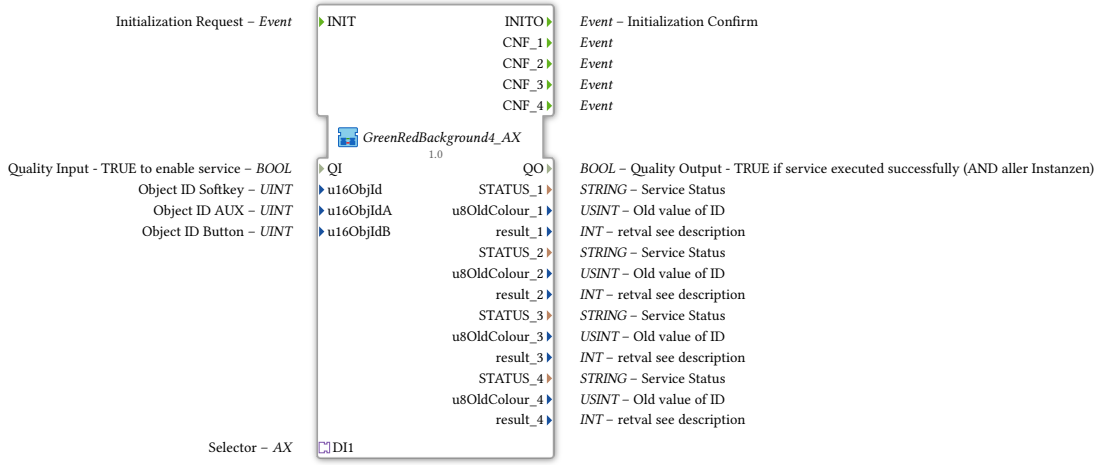
• — ○ • — ○ • — ○ •

Introduction

`GreenRedBackground4_AX` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA, u16ObjIdB`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI` and `Q_BackgroundColour_4.QI`

- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_4`) of the `QO` outputs of `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO` and `Q_BackgroundColour_4.QO`
- **Initialization chain**: `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`, `Q_BackgroundColour_4.INITO` → `INITO`
- `E_MERGE_QO_A` (type `iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_4.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` and `Q_BackgroundColour_3.CNF` and passes the result to `E_MERGE_QO.EI1`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_2`) merges `E_MERGE_QO_A.EO` and `Q_BackgroundColour_4.CNF` and triggers `AND_QO.REQ`

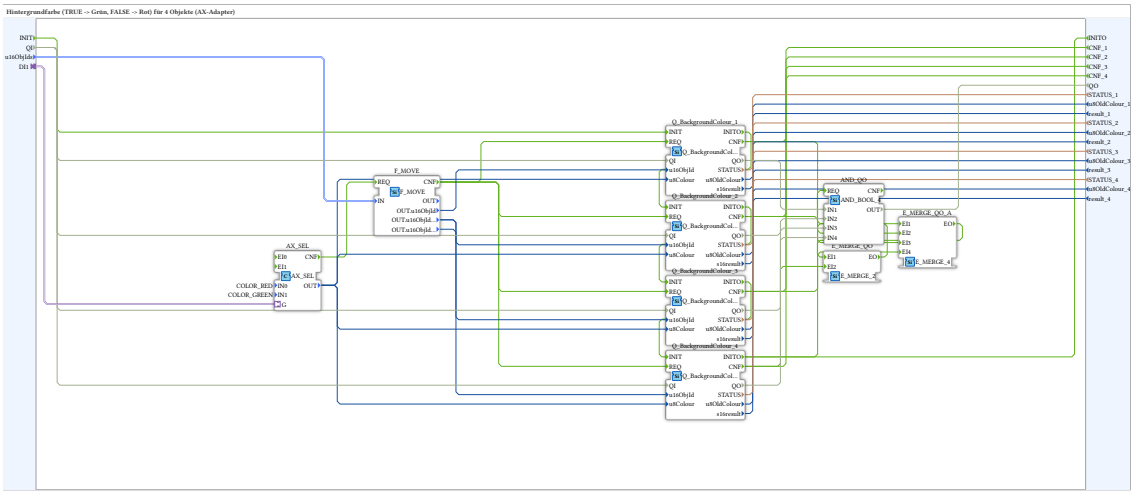
Summary

One of many variants in the background color block family: color pair Grün/Rot, 4 objects, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground4_AXS



• – ○ • – ○ • – ○ •

Introduction

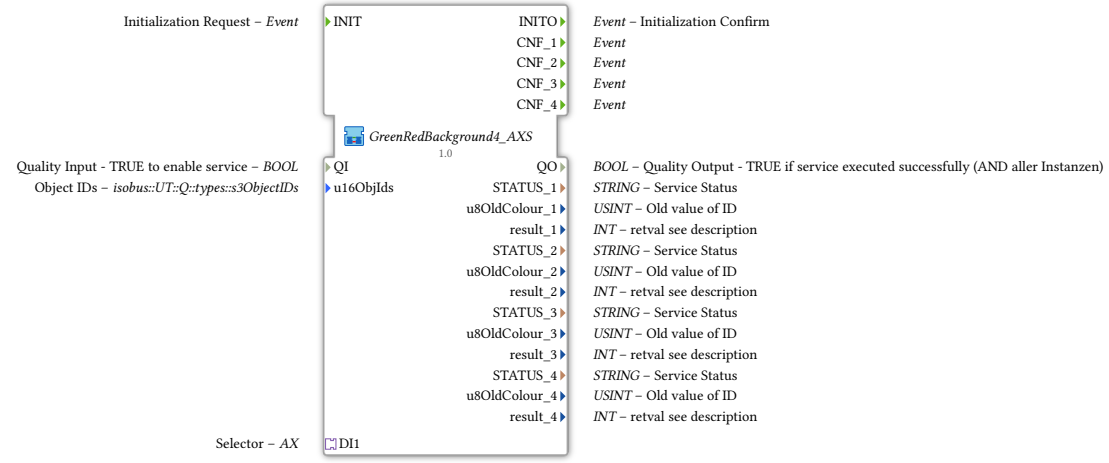
`GreenRedBackground4_AXS` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` → **Grün**, `FALSE` → **Rot**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s3ObjectIDs` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Initialization and quality: The SubApp has `INIT`/`INITO` and `QI`/`QO`. `INIT` → `Q_BackgroundColour_1.INIT` → `Q_BackgroundColour_2.INIT` → `Q_BackgroundColour_3.INIT` → `Q_BackgroundColour_4.INIT` (each via `INITO`), `Q_BackgroundColour_4.INITO` → `INITO`. `QI` goes to all four instances. `QO` is the `AND` (`AND_QO`, `AND_BOOL_4`) of the four `QO` outputs; `AND_QO.REQ` is triggered via the

cascade `E_MERGE_QO_A` (`E_MERGE_4`: `Q_BackgroundColour_4.INITO`, `Q_BackgroundColour_1.CNF`, `_2.CNF`, `_3.CNF`) and `E_MERGE_QO` (`E_MERGE_2`: output of `E_MERGE_QO_A` and `Q_BackgroundColour_4.CNF`).



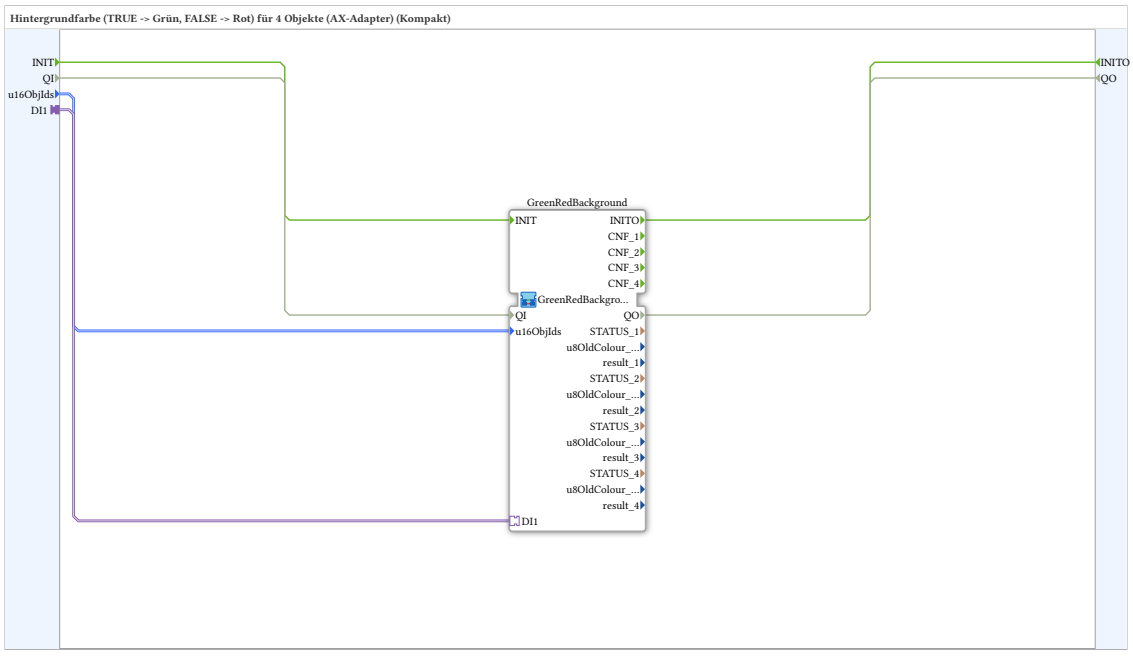
Summary

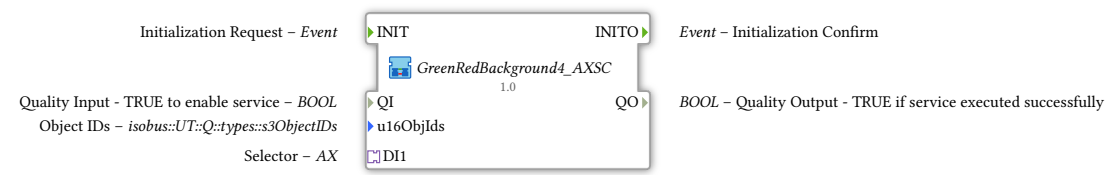
One of many variants in the background color block family: color pair Grün/Rot, 4 objects, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground4_AXSC





• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground4_AXSC` is a **compact wrapper** around `GreenRedBackground4_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

In addition, initialization and quality are passed through: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` (initial value `TRUE`) ⟶ `GreenRedBackground.QI` and `GreenRedBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

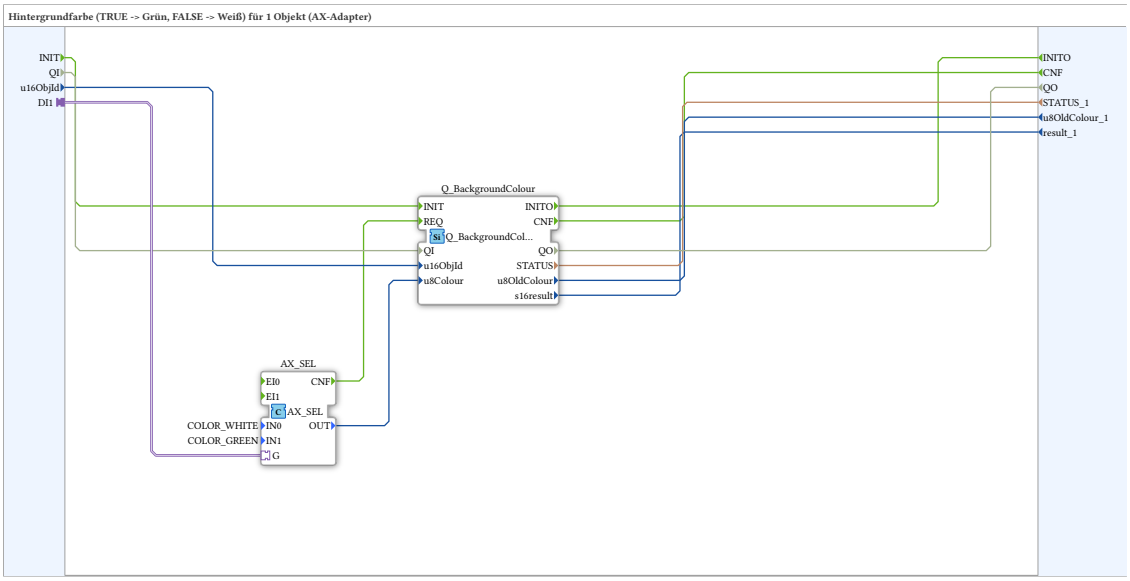
Summary

A pure pass-through to `GreenRedBackground4_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1_AX



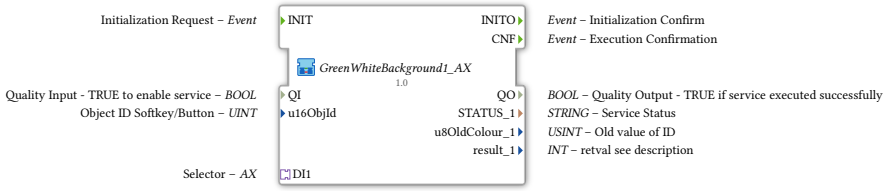
• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground1_AX` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). They are passed straight through to `Q_BackgroundColour`: `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` ⟶ `QO`.



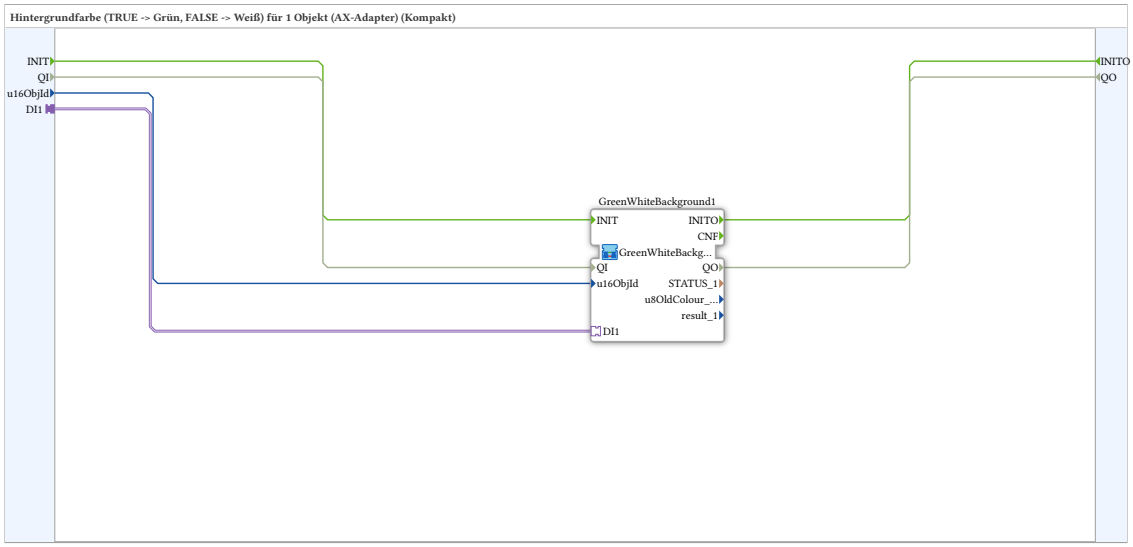
Summary

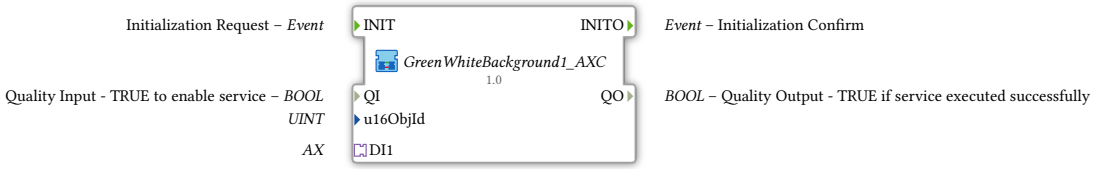
One of many variants in the background color block family: color pair Grün/Weiß, 1 object, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1_AXC





• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground1_AXC` is a **compact wrapper** around `GreenWhiteBackground1_AX` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `GreenWhiteBackground1`: `INIT` ⟶ `GreenWhiteBackground1.INIT`, `GreenWhiteBackground1.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground1.QI`, `GreenWhiteBackground1.QO` ⟶ `QO`.

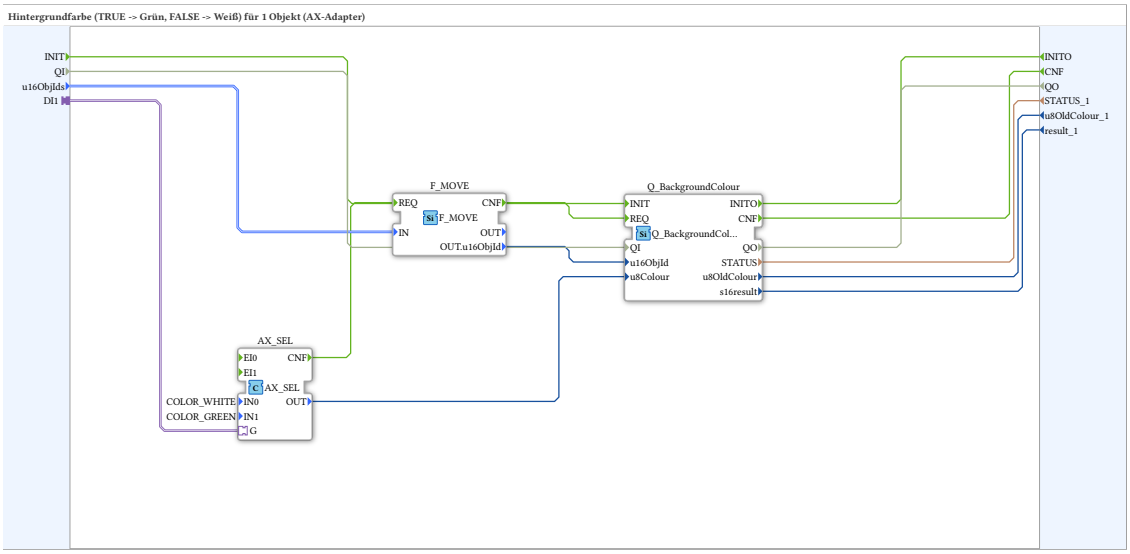
Summary

A pure pass-through to `GreenWhiteBackground1_AX` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1_AXS



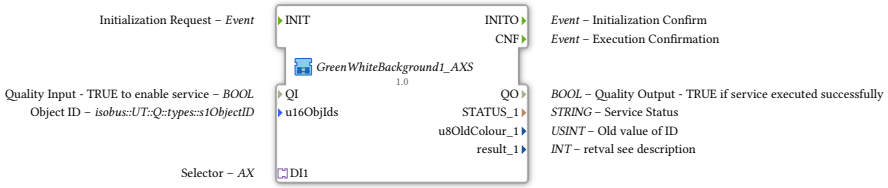
• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground1_AXS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). They are passed straight through to `Q_BackgroundColour`: `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColour.QI`, `Q_BackgroundColour.QO` ⟶ `QO`.



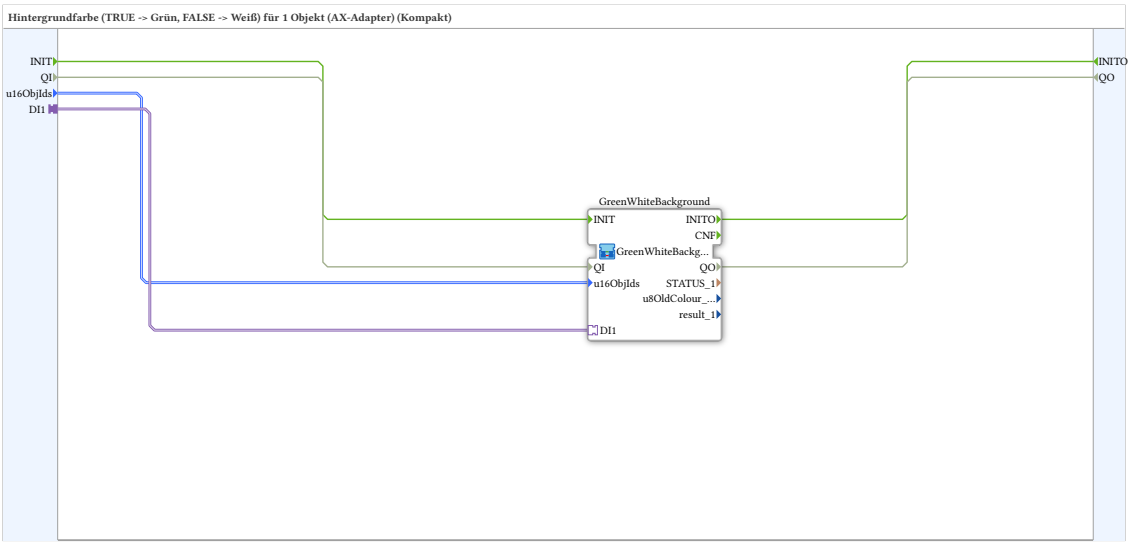
Summary

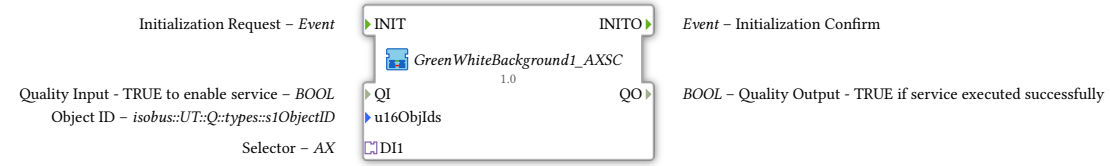
One of many variants in the background color block family: color pair Grün/Weiß, 1 object, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1_AXSC





• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground1_AXSC` is a **compact wrapper** around `GreenWhiteBackground1_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`) are passed through: `INIT` ⟶ `GreenWhiteBackground.INIT`; `GreenWhiteBackground.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) ⟶ `GreenWhiteBackground.QI`; **data output `QO`** (`BOOL`) ⟵ `GreenWhiteBackground.QO`.

The inner SubApp instance `GreenWhiteBackground` is of type `GreenWhiteBackground1_AXS`.

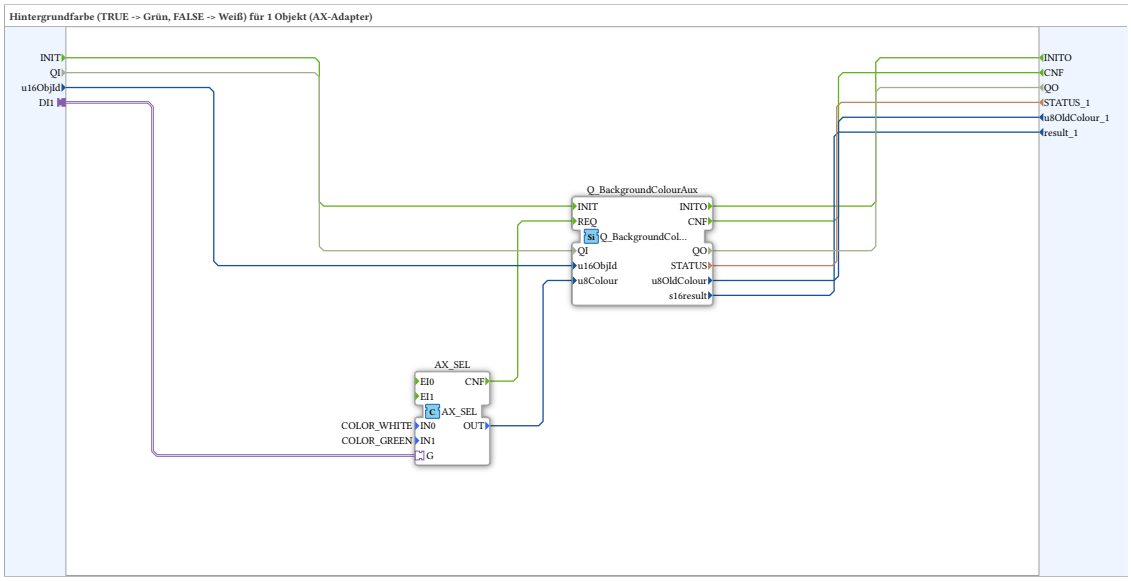
Summary

A pure pass-through to `GreenWhiteBackground1_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1_aux_AX



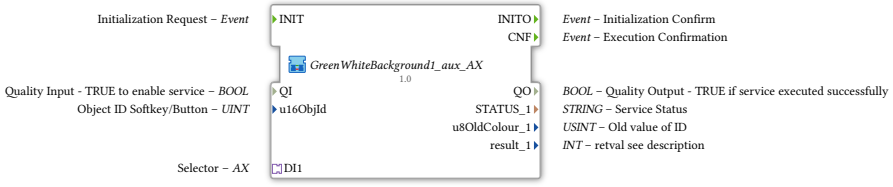
• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground1_aux_AX` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). They are passed straight through to `Q_BackgroundColourAux`: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



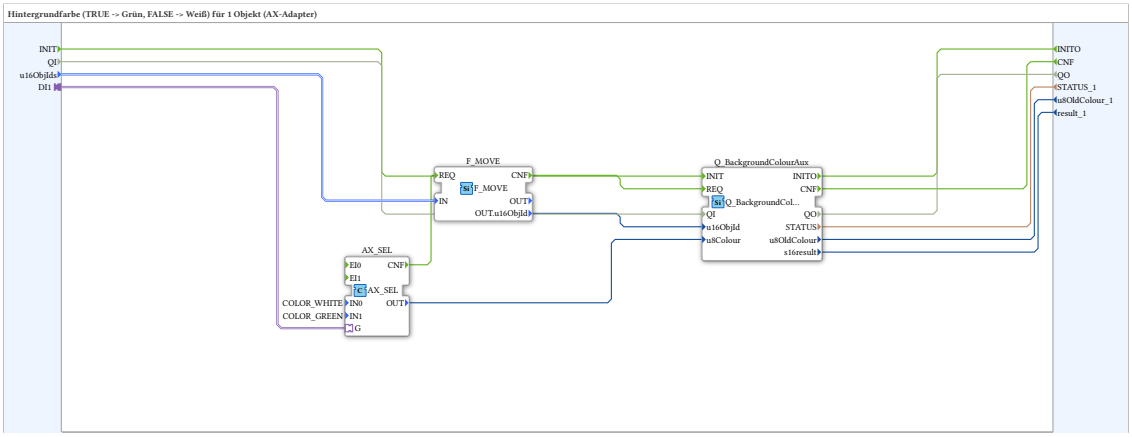
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 1 object, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1_aux_AXS



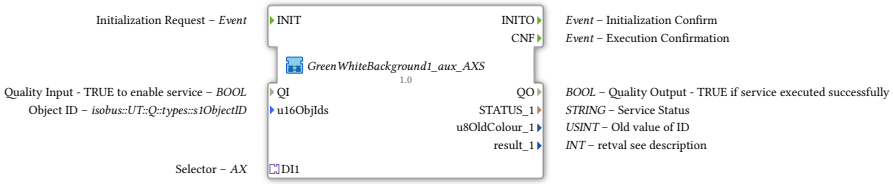
• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground1_aux_AXS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality pattern: the SubApp has the event input `INIT`, the event output `INITO`, the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). They are passed straight through to `Q_BackgroundColourAux`: `INIT` ⟶ `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` ⟶ `INITO`, `QI` ⟶ `Q_BackgroundColourAux.QI`, `Q_BackgroundColourAux.QO` ⟶ `QO`.



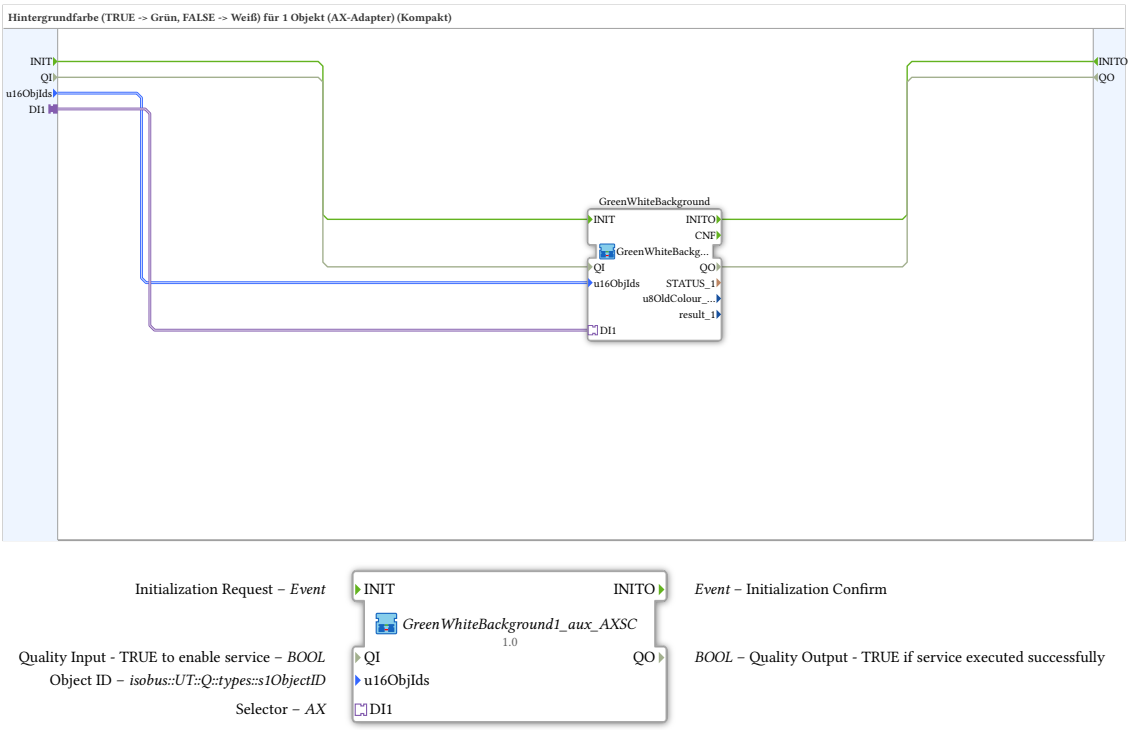
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 1 object, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1_aux_AXSC



• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground1_aux_AXSC` is a **compact wrapper** around `GreenWhiteBackground1_aux_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `GreenWhiteBackground`: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

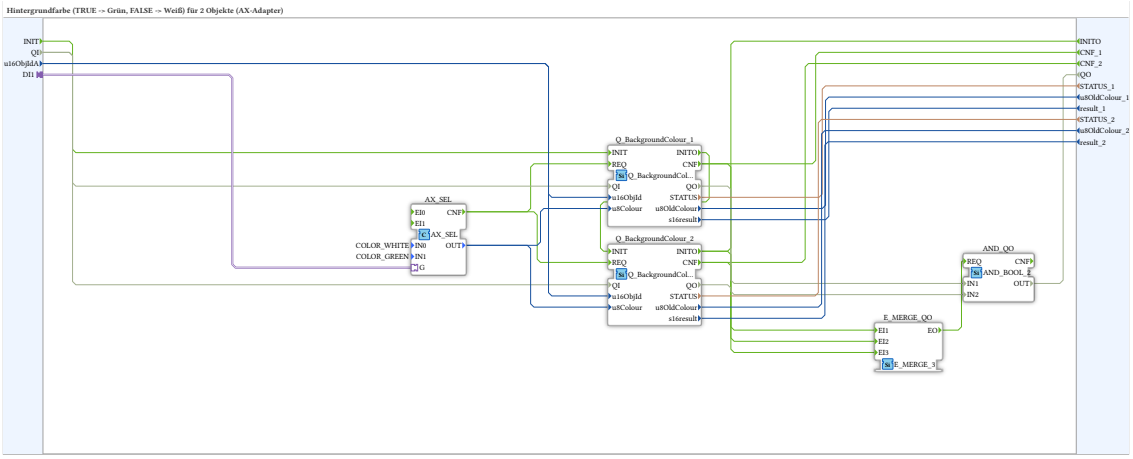
Summary

A pure pass-through to `GreenWhiteBackground1_aux_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground2_AX



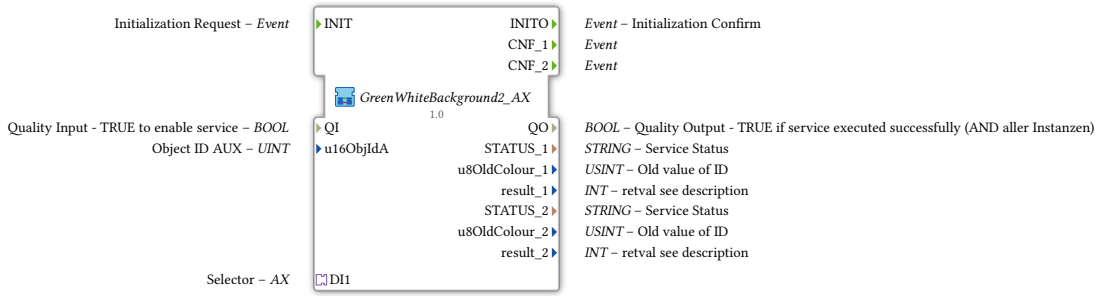
• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground2_AX` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` → **Grün**, `FALSE` → **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via inputs `u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_BackgroundColour_1.QI` and `Q_BackgroundColour_2.QI`
- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_2`) of the `QO` outputs of `Q_BackgroundColour_1.QO` and `Q_BackgroundColour_2.QO`
- **Initialization chain**: `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `INITO`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_3`) merges `Q_BackgroundColour_2.INITO`, `Q_BackgroundColour_1.CNF` and `Q_BackgroundColour_2.CNF` and triggers `AND_QO.REQ`

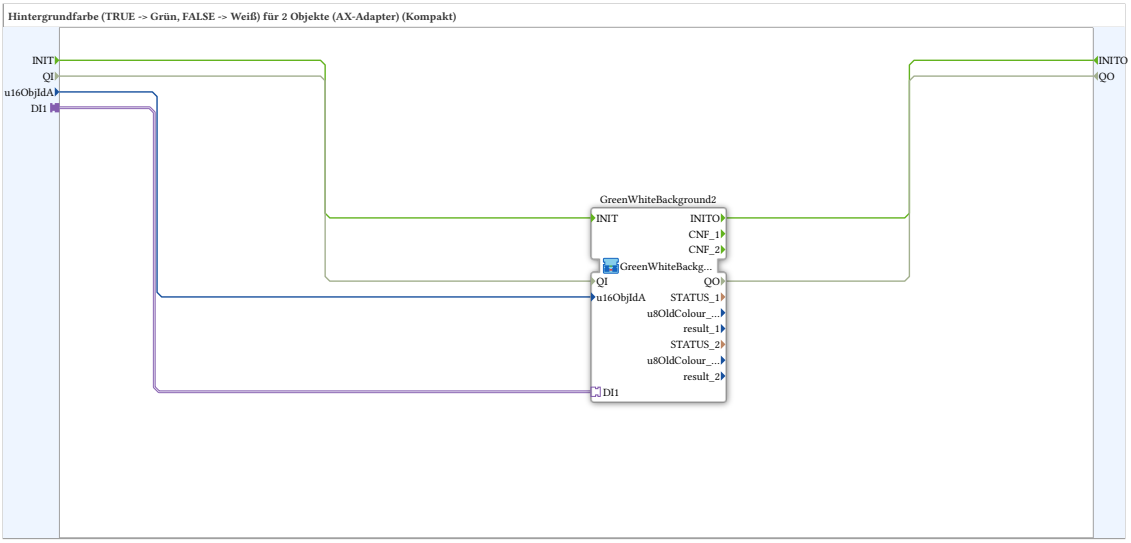
Summary

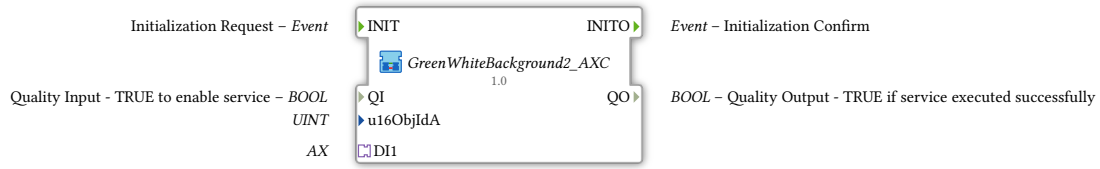
One of many variants in the background color block family: color pair Grün/Weiß, 2 objects, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground2_AXC





• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground2_AXC` is a **compact wrapper** around `GreenWhiteBackground2_AX` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `GreenWhiteBackground2.INIT`, `GreenWhiteBackground2.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground2.QI` (initial value `TRUE`) and `GreenWhiteBackground2.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

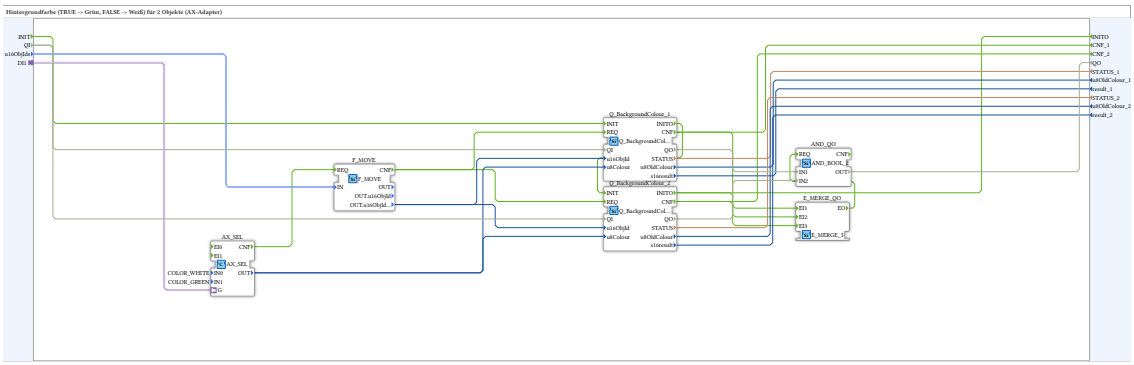
Summary

A pure pass-through to `GreenWhiteBackground2_AX` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground2_AXS



• – ○ • – ○ • – ○ •

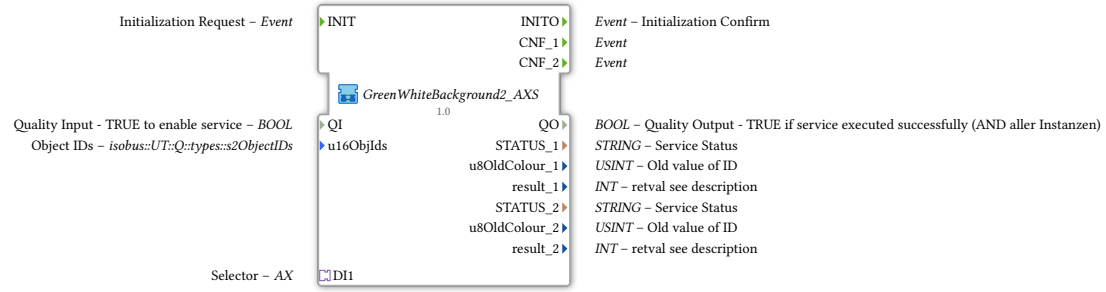
Introduction

`GreenWhiteBackground2_AXS` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` → **Grün**, `FALSE` → **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

Initialization and quality: In addition to the previous events, the block has the event input `INIT`, the event output `INITO`, as well as `QI` (BOOL, input, initial value `TRUE`) and `QO` (BOOL, output, AND of all instances). Init chain: `INIT` → `Q_BackgroundColour_1` → `Q_BackgroundColour_2` → `INITO` (each `INITO` → next `INIT`). `QI` is passed on to all `Q_BackgroundColour_n.QI`. `QO` is formed by `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_1.QO` → `AND_QO.IN1`, `Q_BackgroundColour_2.QO` → `AND_QO.IN2`, `AND_QO.OUT` → `QO`. `E_MERGE_QO` (`E_MERGE_3`) merges `Q_BackgroundColour_2.INITO` and the `CNF` outputs `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`; `E_MERGE_QO.EO` triggers `AND_QO.REQ`.

For the general pattern (selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



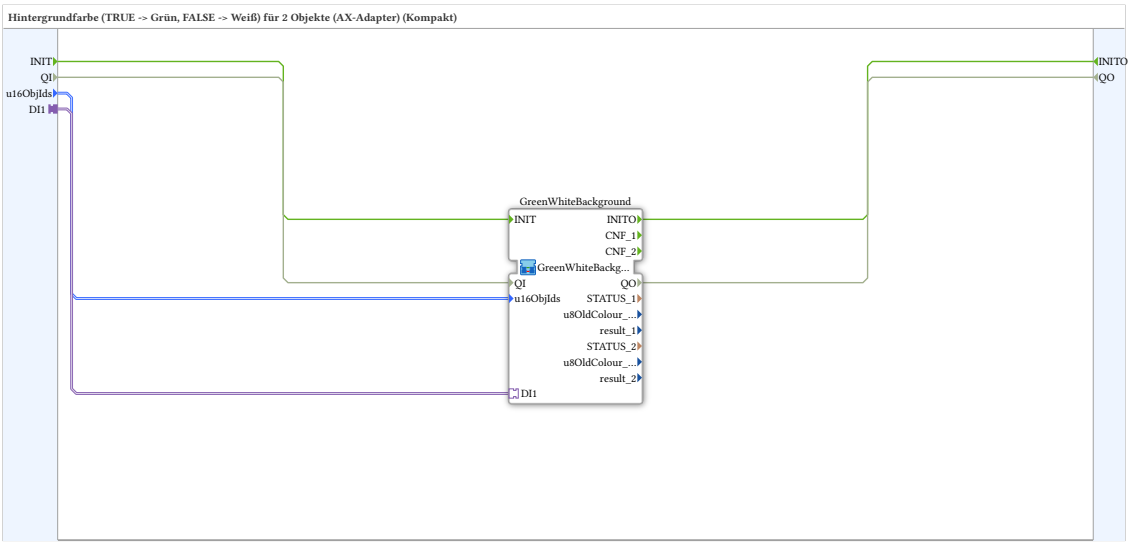
Summary

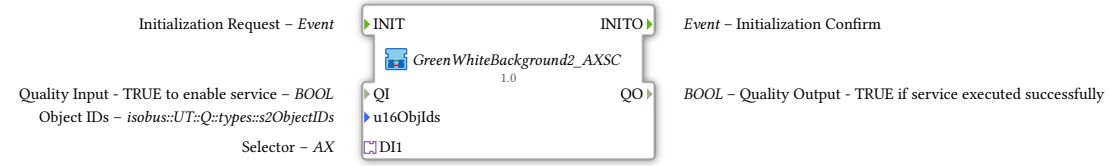
One of many variants in the background color block family: color pair Grün/Weiß, 2 objects, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground2_AXSC





• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground2_AXSC` is a **compact wrapper** around `GreenWhiteBackground2_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `GreenWhiteBackground`: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

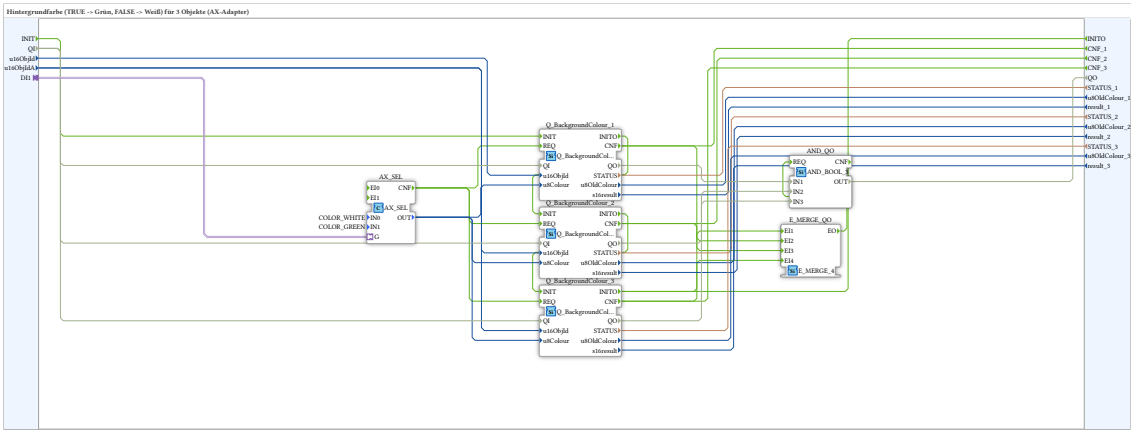
Summary

A pure pass-through to `GreenWhiteBackground2_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground3_AX



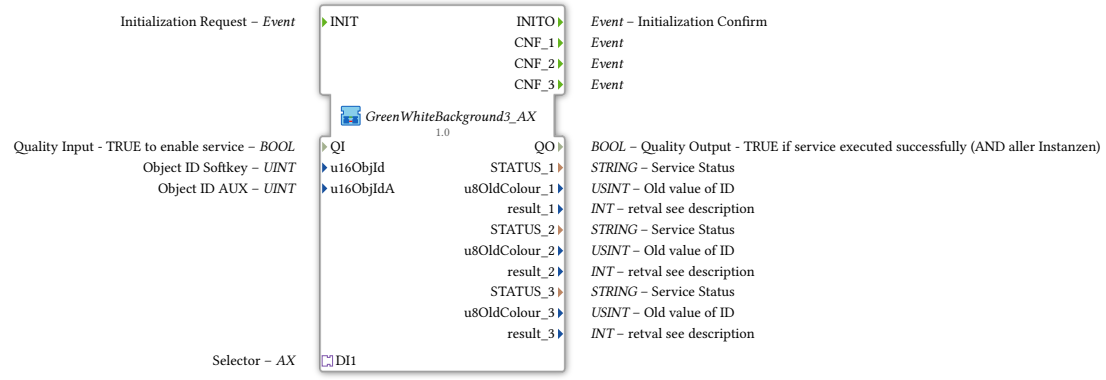
• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground3_AX` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_3.INITO` and the `CNF` events of all three instances (`Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`) and thereby triggers `AND_QO.REQ`.

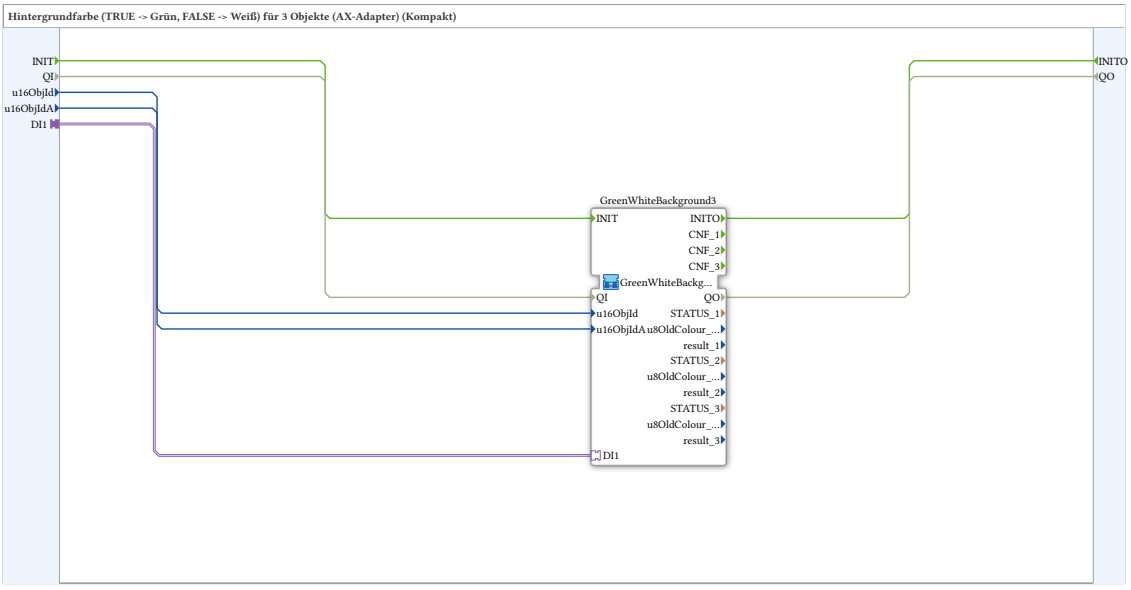
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 3 objects, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground3_AXC





• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground3_AXC` is a **compact wrapper** around `GreenWhiteBackground3_AX` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `GreenWhiteBackground3`: `INIT` ⟶ `GreenWhiteBackground3.INIT`, `GreenWhiteBackground3.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground3.QI`, `GreenWhiteBackground3.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

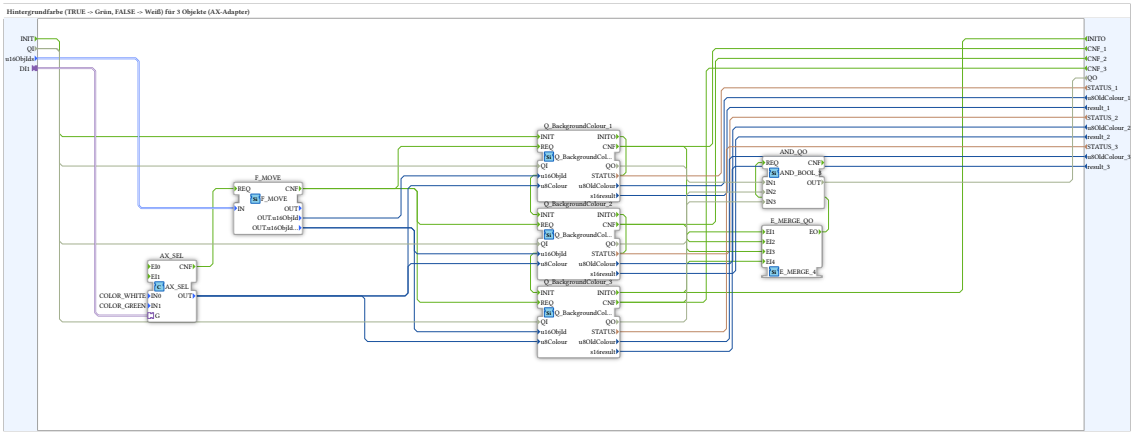
Summary

A pure pass-through to `GreenWhiteBackground3_AX` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground3_AXS



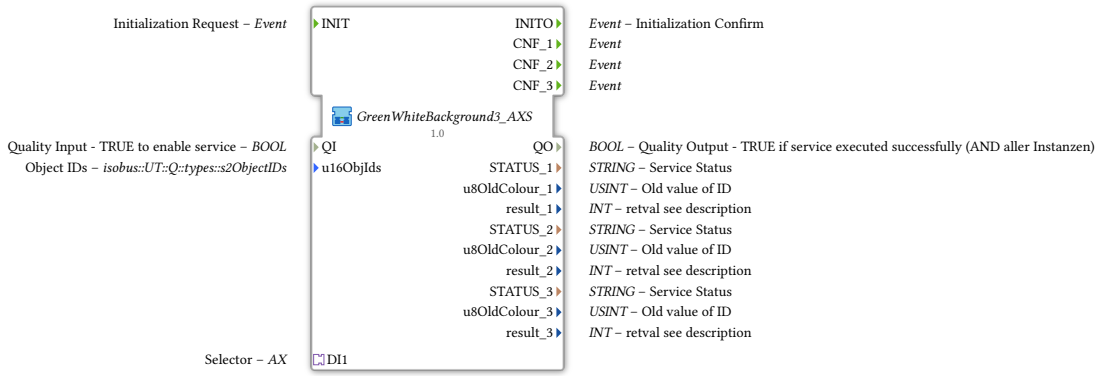
• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground3_AXS` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI` and `Q_BackgroundColour_3.QI`
- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_3`) of the `QO` outputs of `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO` and `Q_BackgroundColour_3.QO`
- **Initialization chain**: `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` ⟶ `INITO`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_3.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` and `Q_BackgroundColour_3.CNF` and triggers `AND_QO.REQ`

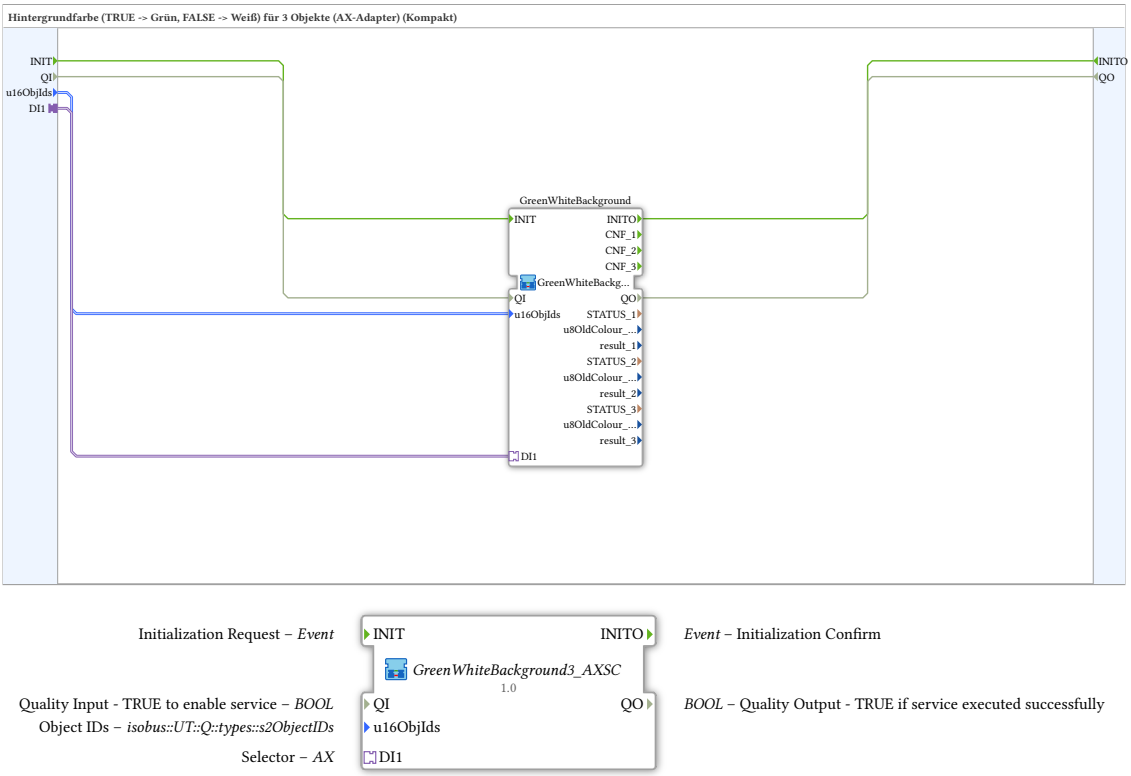
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 3 objects, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground3_AXSC



• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground3_AXSC` is a **compact wrapper** around `GreenWhiteBackground3_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI` (initial value `TRUE`) and `GreenWhiteBackground.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

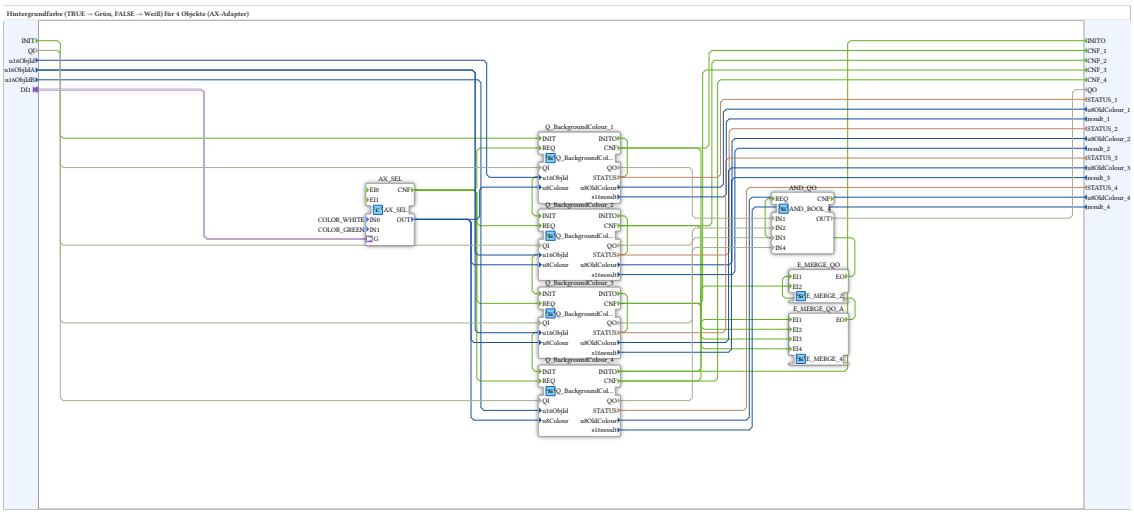
Summary

A pure pass-through to `GreenWhiteBackground3_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground4_AX



• – ○ • – ○ • – ○ •

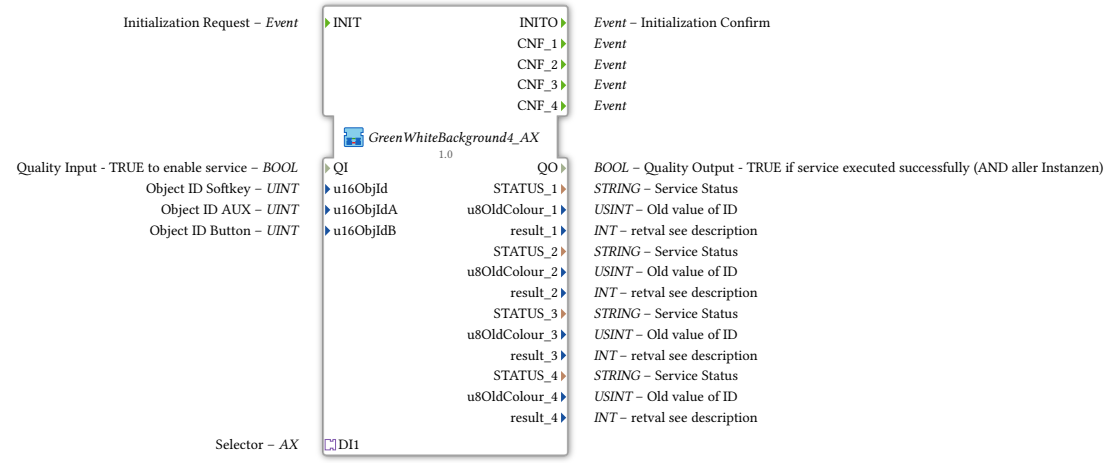
Introduction

`GreenWhiteBackground4_AX` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` → **Grün**, `FALSE` → **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA, u16ObjIdB`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`u16ObjIdB`	`Q_BackgroundColour` (regular object)

Initialization and quality: In addition to the previous events, the block has the event input `INIT`, the event output `INITO`, as well as `QI` (BOOL, input, initial value `TRUE`) and `QO` (BOOL, output, AND of all instances). Init chain: `INIT` → `Q_BackgroundColour_1` → `Q_BackgroundColour_2` → `Q_BackgroundColour_3` → `Q_BackgroundColour_4` → `INITO` (each `INITO` → next `INIT`). `QI` is passed on to all `Q_BackgroundColour_n.QI`. `QO` is formed by `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` → `AND_QO.IN1`, `Q_BackgroundColour_2.QO` → `AND_QO.IN2`, `Q_BackgroundColour_3.QO` → `AND_QO.IN3`, `Q_BackgroundColour_4.QO` → `AND_QO.IN4`, `AND_QO.OUT` → `QO`. `E_MERGE_QO_A` (`E_MERGE_4`) merges `Q_BackgroundColour_4.INITO` and the `CNF` outputs of `Q_BackgroundColour_1`, `Q_BackgroundColour_2` and `Q_BackgroundColour_3`, `E_MERGE_QO` (`E_MERGE_2`) adds `Q_BackgroundColour_4.CNF`; `E_MERGE_QO.EO` triggers `AND_QO.REQ`.

For the general pattern (selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



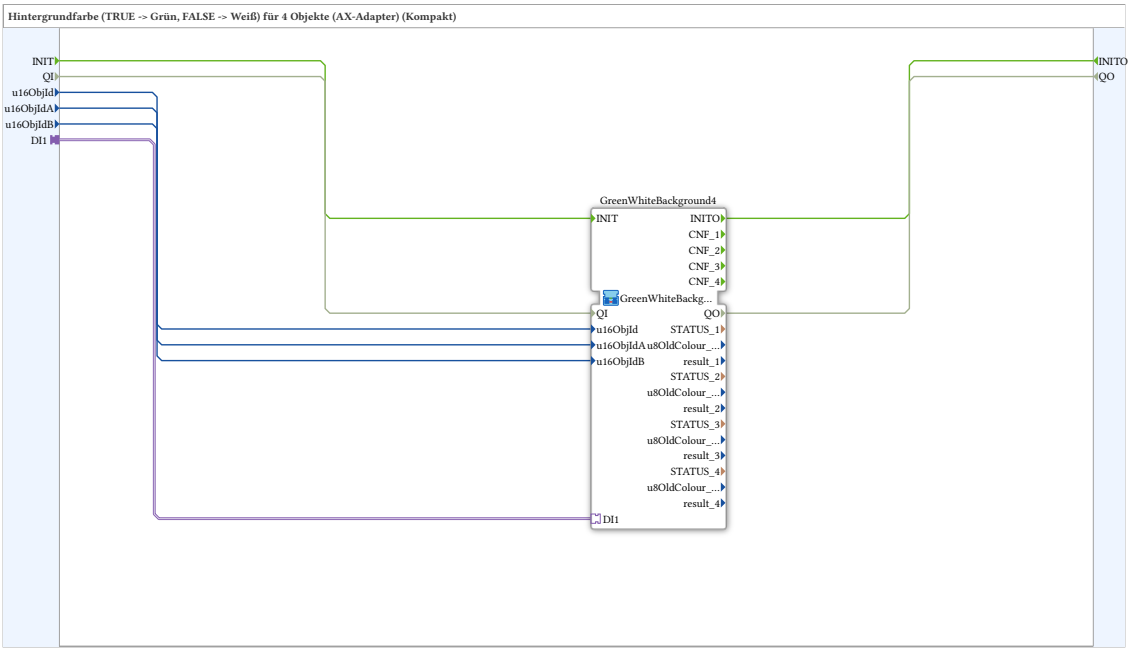
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 4 objects, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground4_AXC





• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground4_AXC` is a **compact wrapper** around `GreenWhiteBackground4_AX` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `GreenWhiteBackground4`: `INIT` → `GreenWhiteBackground4.INIT`, `GreenWhiteBackground4.INITO` → `INITO`, `QI` → `GreenWhiteBackground4.QI`, `GreenWhiteBackground4.QO` → `QO`.

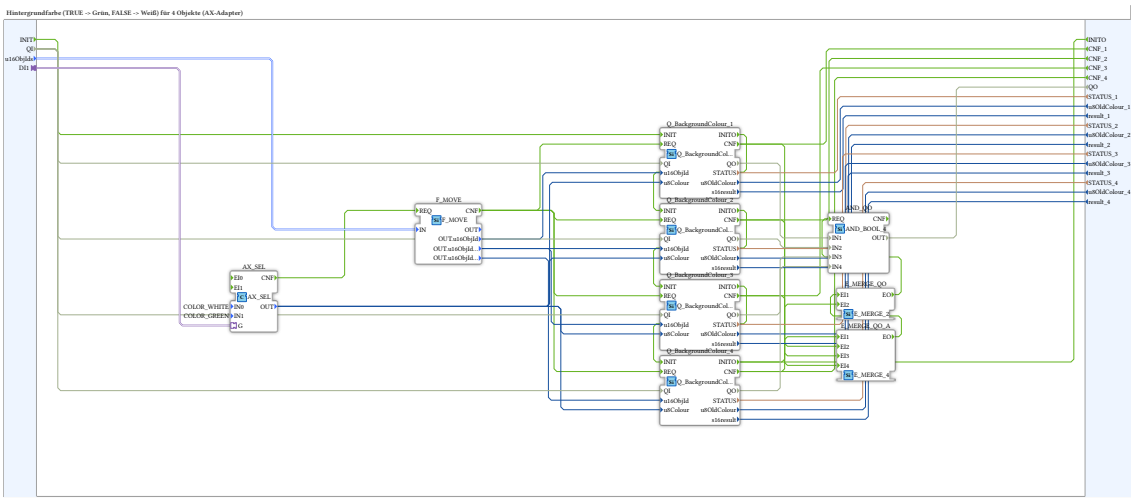
Summary

A pure pass-through to `GreenWhiteBackground4_AX` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground4_AXS



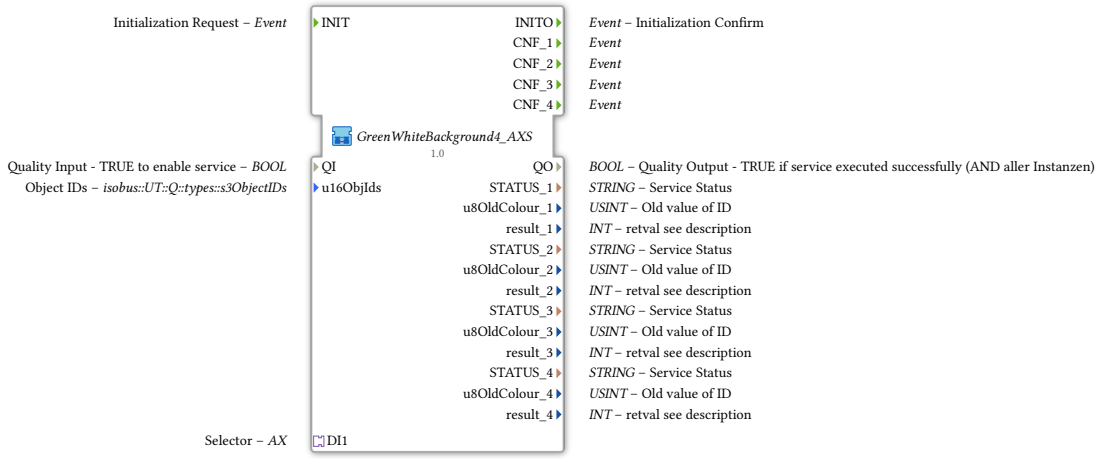
• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground4_AXS` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` → **Grün**, `FALSE` → **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s3ObjectIDs` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`, `Q_BackgroundColour_4`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO_A` (`iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_4.INITO` and the `CNF` events of instances 1 to 3; `E_MERGE_QO` (`iec61499::events::E_MERGE_2`) merges its output `E_MERGE_QO_A.EO` with `Q_BackgroundColour_4.CNF` and thereby triggers `AND_QO.REQ`.

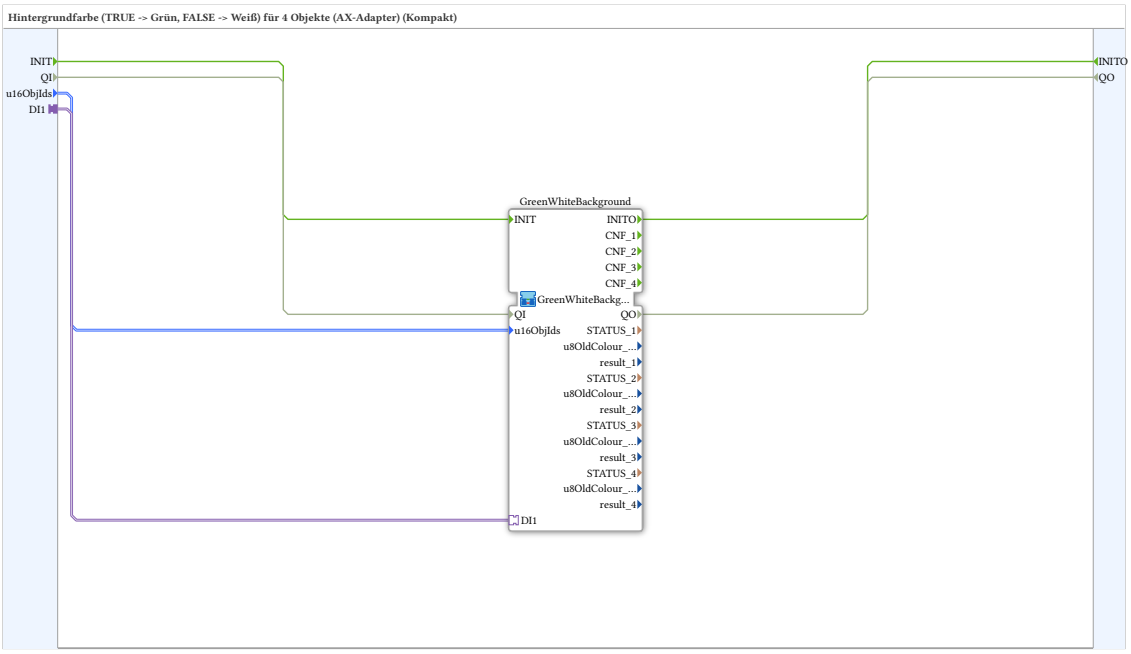
Summary

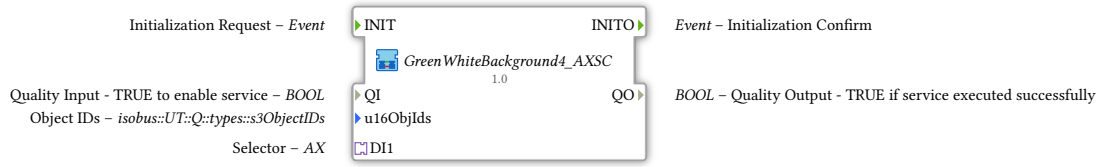
One of many variants in the background color block family: color pair Grün/Weiß, 4 objects, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground4_AXSC





• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground4_AXSC` is a **compact wrapper** around `GreenWhiteBackground4_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `GreenWhiteBackground`: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

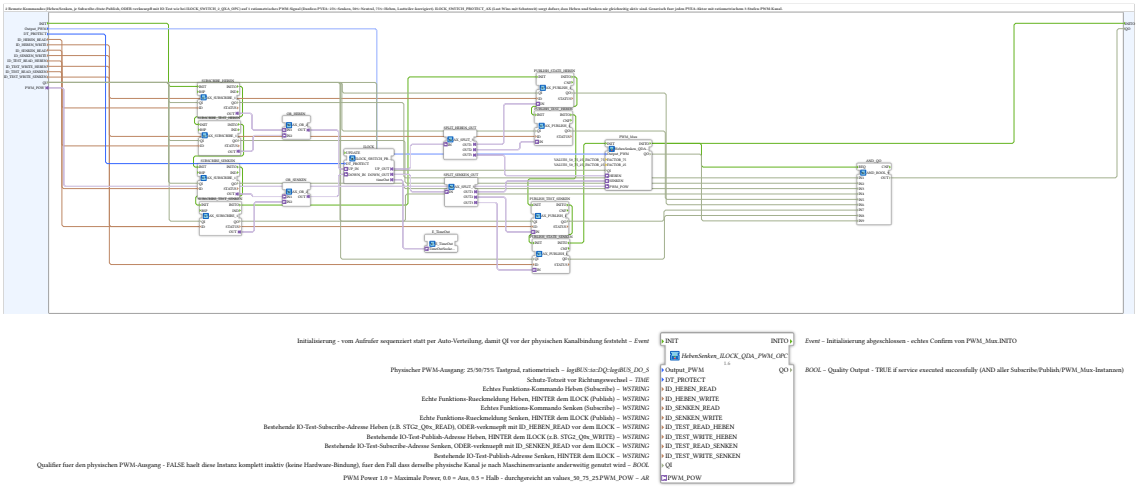
Summary

A pure pass-through to `GreenWhiteBackground4_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

HebenSenken_ILOCK_QDA_PWM_OPC



• – ○ • – ○ • – ○ •

Introduction

The **HebenSenken_ILOCK_QDA_PWM_OPC** subapplication combines two remote commands – **Raise** (Heben) and **Lower** (Senken) – with existing IO-test signals and generates a single ratiometric PWM output signal. The subapp is designed for Danfoss PVEA actuators with a three-stage PWM channel, where:

- **25%** duty cycle = Lower (Senken)
- **50%** duty cycle = Neutral
- **75%** duty cycle = Raise (Heben)

Each remote command path consists of a Subscribe adapter (for the actual control command) OR-combined with an existing IO-test Subscribe signal, followed by a Publish adapter for state feedback behind the interlock. An **ILOCK_SWITCH_PROTECT_AX** interlock block applies a *last-wins* strategy with a configurable protection time (`DT_PROTECT`) to guarantee that Raise and Lower are never active simultaneously. The selected duty cycle is then corrected by a load-divider compensation factor and scaled to a 13-bit PWM range before being forwarded to a physical PWM output.

The subapp is generic and can be reused with any PVEA actuator equipped with a ratiometric three-stage PWM channel.

Interface Structure

The subapplication exposes data inputs, the event input `INIT` and the event output `INITO`. It does not provide any adapter interfaces at its boundary; all other event and adapter communication occurs internally within the subapp network.

Event Inputs

- **INIT** (`EInit`): initialisation; sequenced by the caller instead of automatic distribution, so that `QI` is settled before the physical channel is bound; starts the initialisation chain `SUBSCRIBE_HEBEN` ⟶ `SUBSCRIBE_TEST_HEBEN` ⟶ `SUBSCRIBE_SENKEN` ⟶ `SUBSCRIBE_TEST_SENKEN` ⟶ `PUBLISH_STATE_HEBEN` ⟶ `PUBLISH_TEST_HEBEN` ⟶ `PUBLISH_TEST_SENKEN` ⟶ `PUBLISH_STATE_SENKEN` ⟶ `PWM_Mux` (`HebenSenken_QDA_PWM`)

All other triggering is handled internally via the subscribe/publish adapter pattern.

Event Outputs

- **INITO** (`EInit`): initialisation completed; genuine confirm from `PWM_Mux.INITO` (end of the initialisation chain)

The subapp provides no further event outputs.

Data Inputs

Name	Type	Description

QI	BOOL	Qualifier for the physical PWM output (default TRUE); FALSE keeps the instance completely inactive (no hardware binding) in case the same physical channel is used differently depending on the machine variant; passed on to all Subscribe/Publish blocks (SUBSCRIBE_*, PUBLISH_*) and to PWM_Mux.QI.
Output_PWM	logiBUS::io::DQ::logiBUS_DO_S	Physical PWM output: 25/50/75% duty cycle, ratiometric. Default: logiBUS_DO::Invalid
DT_PROTECT	TIME	Protection deadtime before direction changes. Default: T#300ms
ID_HEBEN_READ	WSTRING	Actual function command for Raise (Subscribe).
ID_HEBEN_WRITE	WSTRING	Actual function feedback for Raise, *behind* the ILOCK (Publish).
ID_SENKEN_READ	WSTRING	Actual function command for Lower (Subscribe).
ID_SENKEN_WRITE	WSTRING	Actual function feedback for Lower, *behind* the ILOCK (Publish).
ID_TEST_READ_HEBEN	WSTRING	Existing IO-test Subscribe address for Raise (e.g., STG2_Q0x_READ), OR-combined with ID_HEBEN_READ before the ILOCK.
ID_TEST_WRITE_HEBEN	WSTRING	Existing IO-test Publish address for Raise, *behind* the ILOCK (e.g., STG2_Q0x_WRITE).
ID_TEST_READ_SENKEN	WSTRING	Existing IO-test Subscribe address for Lower, OR-combined with ID_SENKEN_READ before the ILOCK.
ID_TEST_WRITE_SENKEN	WSTRING	Existing IO-test Publish address for Lower, *behind* the ILOCK.

Data Outputs

Name	Type	Description
QO	BOOL	Quality Output - TRUE if the service executed successfully; AND (AND_QO, type iec61131::booleanOperators::AND_BOOL_9) of the QO of all four SUBSCRIBE_*, all four PUBLISH_* and PWM_Mux; triggered by PWM_Mux.INITO (REQ).

Status and feedback information are published via the internal adapter network using the configured Publish addresses (ID_HEBEN_WRITE, ID_SENKEN_WRITE, ID_TEST_WRITE_HEBEN, ID_TEST_WRITE_SENKEN).

Adapters

None exposed at the interface level. All adapter communication is internal to the subapp network.

Functionality

The subapp processes two independent command sources and merges them into a single PWM output according to the following flow:

1. **Command acquisition**: The four Subscribe adapters (SUBSCRIBE_HEBEN, SUBSCRIBE_TEST_HEBEN, SUBSCRIBE_SENKEN, SUBSCRIBE_TEST_SENKEN) receive the actual function commands and the IO-test commands. Each pair is combined with an AX_OR_2 OR block, so either the real command or the IO-test command can activate the corresponding direction.
2. **Interlock protection**: The OR outputs (OR_HEBEN.OUT, OR_SENKEN.OUT) are fed into ILOCK_SWITCH_PROTECT_AX. This block implements a *last-wins* strategy with a user-configurable protection time (DT_PROTECT). It ensures that the Raise and Lower signals can never be active simultaneously. The internal timeOut event of the ILOCK is connected to an E_TimeOut FB, which provides the timing for the protection interval.
3. **Signal distribution**: The protected ILOCK outputs (UP_OUT, DOWN_OUT) are split via AX_SPLIT_3 blocks:
   - SPLIT_HEBEN_OUT: OUT1 publishes the state to PUBLISH_STATE_HEBEN, OUT2 publishes to PUBLISH_TEST_HEBEN, and OUT3 provides the Raise bit to the assembly block.
   - SPLIT_SENKEN_OUT: OUT1 provides the Lower bit to the assembly block, OUT2 publishes to PUBLISH_TEST_SENKEN, and OUT3 publishes to PUBLISH_STATE_SENKEN.
4. **Selection logic**: The two bits (BIT_00 for Raise, BIT_01 for Lower) are assembled by ASSEMBLE_AB_FROM_AX into a 2-bit pattern, which is converted to an unsigned integer value by AB_TO_AUI. This value is used as the selector input K of the AR_AUI_MUX_3 multiplexer. The MUX selects among three reference duty cycles provided by the values_50_75_25 subapp:
   - IN1 ⟶ 50% (Neutral)
   - IN2 ⟶ 75% (Raise)
   - IN3 ⟶ 25% (Lower)

   Since the ILOCK prevents both bits from being set simultaneously, a 3-way MUX is sufficient (the invalid 4th state cannot occur).
5. **Scaling and conversion**: The selected duty-cycle value passes through two AR_R_MUL multiplication stages:
   - **Correction factor** (FACTOR = REAL#1.17619, which equals 1/0.8502) compensates for the load-divider characteristics.
   - **PWM scale factor** (FACTOR = REAL#81.91, which equals 8191/100) maps the percentage value to the 13-bit PWM range (0…8191).

   The resulting analog value is converted by AR_TO_AD_NUM into a digital number, which drives the logiBUS_QDA_PWM block (DigitalOutput_PWM) that generates the physical PWM signal on the configured Output_PWM channel.
6. **Initialization and status**: QI is passed on to all Subscribe/Publish blocks and to PWM_Mux. INIT runs through the initialisation chain SUBSCRIBE_HEBEN ⟶ SUBSCRIBE_TEST_HEBEN ⟶ SUBSCRIBE_SENKEN ⟶ SUBSCRIBE_TEST_SENKEN ⟶ PUBLISH_STATE_HEBEN ⟶ PUBLISH_TEST_HEBEN ⟶ PUBLISH_TEST_SENKEN ⟶ PUBLISH_STATE_SENKEN ⟶ PWM_Mux (HebenSenken_QDA_PWM); PWM_Mux.INITO is output as INITO and also triggers AND_QO.REQ. QO is the AND (AND_QO) of the nine QO outputs.
7. **Feedback publishing**: The ILOCK-protected states are published back via four AX_PUBLISH_1 adapters (PUBLISH_STATE_HEBEN, PUBLISH_TEST_HEBEN, PUBLISH_STATE_SENKEN, PUBLISH_TEST_SENKEN) to the respective feedback addresses, ensuring that external systems only see the actual (interlock-protected) commands, not the raw inputs.

Technical Features

- **Interlock with protection time**: The ILOCK_SWITCH_PROTECT_AX block applies a *last-wins* arbitration and enforces a deadtime (DT_PROTECT, default 300 ms) between direction changes, preventing mechanical and hydraulic stress on the actuator.
- **OR-combined command sources**: Each direction can be triggered either by the actual function command or by an IO-test command, allowing test and operation modes without changing the network topology.
- **State feedback behind the ILOCK**: Published states reflect the *actual* actuator commands after interlock protection, so downstream monitoring always sees the real effective command.
- **Ratiometric PWM scaling**: The duty cycles are corrected for load-divider nonlinearity (factor 1.17619 = 1/0.8502) and scaled to a 13-bit PWM resolution (factor 81.91 = 8191/100).

- **Generic 3-stage PWM generation**: Works with any PVEA-type actuator requiring a ratiometric PWM channel with three defined levels.
- **Configurable via data inputs**: All addresses (`ID_*`) and timing parameters (`DT_PROTECT`) are exposed as data inputs, making the subapp parametrizable without internal modification.

State Overview

The subapp can be in one of the following operating states:

State	Condition	MUX Input	PWM Duty Cycle	Description
Neutral	Neither Raise nor Lower active	IN1 (50%)	50%	Actuator is idle; no direction command present.
Raise	Raise command active (after ILOCK)	IN2 (75%)	75%	Actuator moves upward (Heben).
Lower	Lower command active (after ILOCK)	IN3 (25%)	25%	Actuator moves downward (Senken).
Protection	Direction change during `DT_PROTECT`	Held at previous state	Held at previous level	ILOCK holds the last command until the protection time elapses; the new command takes effect afterwards.

The interlock ensures that the combination *Raise+Lower* (both bits set) can never occur; hence the 3-way MUX is sufficient.

Application Scenarios

- **PVEA actuator control**: Directly controlling Danfoss PVEA hydraulic actuators that accept a ratiometric PWM signal with three discrete duty-cycle levels for Raise/Neutral/Lower.
- **Remote HMI control with IO-test integration**: Industrial systems where a remote operator station (via OPC/network subscribes) and a local IO-test panel both need to control the same actuator, with the IO-test acting as an override or backup.
- **Direction-safe switching**: Applications where simultaneous Raise and Lower commands must be impossible due to safety requirements – e.g., crane or lifting systems, agricultural machinery, or mobile hydraulics.
- **Actuator state feedback**: Systems that require publication of the *actual* (protected) command state for visualization, logging, or supervisory control.

Comparison with Similar Blocks

The subapp follows the same pattern as `ILOCK_SWITCH_2_QXA_OPC` (which also OR-combines real commands with IO-test signals and applies `ILOCK_SWITCH_PROTECT_AX` interlock protection). However, the key difference is the output stage:

- **`ILOCK_SWITCH_2_QXA_OPC`** typically produces two discrete digital outputs (QXA) for Raise and Lower.
- **`HebenSenken_ILOCK_QDA_PWM_OPC`** merges the two command bits into a *single ratiometric PWM signal* using a 3-way MUX, compensation scaling, and a PWM output stage. This reduces the output pins from two digital channels to one analog PWM channel and is specifically tailored for PVEA actuators that accept proportional PWM control.

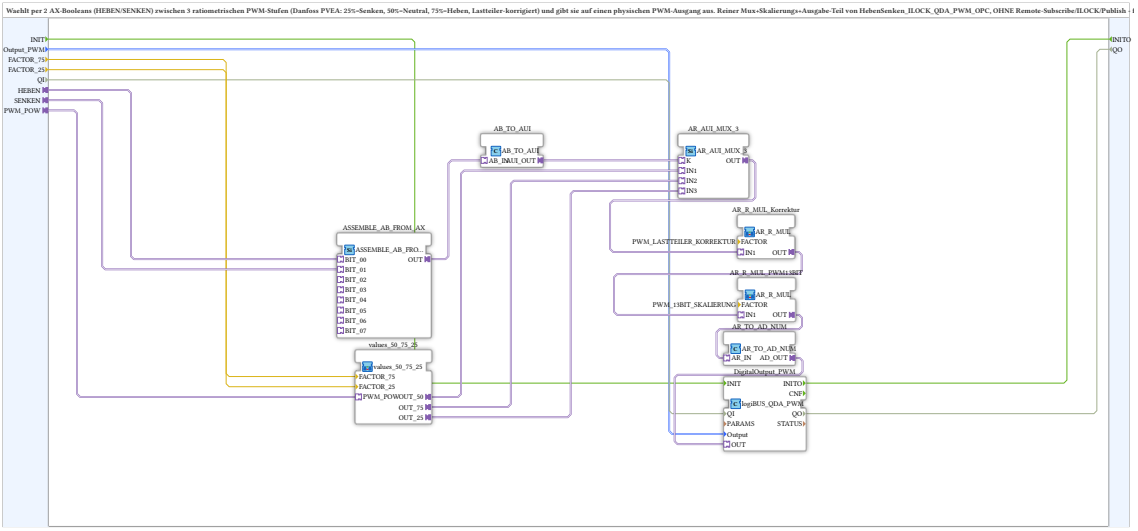
The interlock core remains identical, ensuring consistent protection behavior across both variants.

Conclusion

The `HebenSenken_ILOCK_QDA_PWM_OPC` subapplication provides a robust, generic solution for controlling three-stage PVEA actuators via a single ratiometric PWM signal. It integrates remote function commands and IO-test signals with interlock protection and configurable deadtime, guaranteeing that Raise and Lower can never be active simultaneously. The built-in load-divider correction and 13-bit PWM scaling ensure accurate and repeatable actuator positioning. With its fully parametrizable addresses and timing, the subapp can be reused across different machines and communication layouts without internal changes,

making it a valuable component for hydraulic actuation systems requiring safe and flexible command handling.

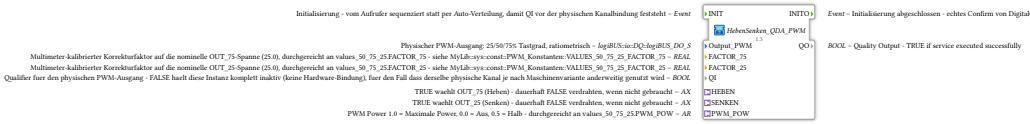
HebenSenken_QDA_PWM



• – ○ • – ○ • – ○ •

Introduction

`HebenSenken_QDA_PWM` selects, via two AX booleans (`HEBEN`/`SENKEN`), between three ratiometric PWM levels (Danfoss PVEA: 25 % = lower, 50 % = neutral, 75 % = raise, load-divider corrected) and outputs the selected level on a physical PWM output. It is the pure mux, scaling and output part of `HebenSenken_ILOCK_QDA_PWM_OPC`, without remote subscribe, ILOCK and publish. It suits callers that already supply `HEBEN`/`SENKEN` by other means (e.g. already gated/exclusive or using only one direction).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation; sequenced by the caller instead of automatic distribution, so that `QI` is settled before the physical channel is bound; passed on to `DigitalOutput_PWM.INIT`

Event Outputs

- **INITO** (`EInit`): initialisation complete; real confirm from `DigitalOutput_PWM.INITO`

Data Inputs

- **Output_PWM** (`logiBUS::io::DQ::logiBUS_DO_S`): physical PWM output (25/50/75 % duty cycle, ratiometric), default `logiBUS_DO::Invalid`
- **FACTOR_75** (REAL): multimeter-calibrated correction factor on the nominal `OUT_75` span (25.0), passed on to `values_50_75_25.FACTOR_75` (default `REAL#1.0`)
- **FACTOR_25** (REAL): multimeter-calibrated correction factor on the nominal `OUT_25` span (25.0), passed on to `values_50_75_25.FACTOR_25` (default `REAL#1.0`)
- **QI** (BOOL): qualifier for the physical PWM output (default TRUE); FALSE keeps the instance completely inactive (no hardware binding) in case the same physical channel is used differently depending on the machine variant

Data Outputs

- **QO** (BOOL): quality output, comes from `DigitalOutput_PWM.QO`

Adapters

- **HEBEN** (socket, `AX`): TRUE selects `OUT_75` (raise); wire permanently to FALSE if not needed
- **SENKEN** (socket, `AX`): TRUE selects `OUT_25` (lower); wire permanently to FALSE if not needed
- **PWM_POW** (socket, `AR`): PWM power, 1.0 = maximum power, 0.0 = off, 0.5 = half; passed on to `values_50_75_25.PWM_POW`

Functionality

The chain in the network:

1. `HEBEN` ⟶ `ASSEMBLE_AB_FROM_AX.BIT_00` and `SENKEN` ⟶ `ASSEMBLE_AB_FROM_AX.BIT_01`: both booleans are assembled into a byte (`HEBEN` is the least significant bit).
2. `AB_TO_AUI` converts the byte into an `AUI` value that goes to `AR_AUI_MUX_3` as the selection `K`: `K = 0` (both FALSE) selects `IN1` = `OUT_50` (neutral), `K = 1` (`HEBEN`) selects `IN2` = `OUT_75`, `K = 2` (`SENKEN`) selects `IN3` = `OUT_25`.
3. `values_50_75_25` supplies the three set values `OUT_50`, `OUT_75`, `OUT_25` from `PWM_POW` and `FACTOR_75`/`FACTOR_25`.
4. `AR_AUI_MUX_3.OUT` ⟶ `AR_R_MUL_Korrektur` (factor `PWM_LASTTEILER_KORREKTUR`) ⟶ `AR_R_MUL_PWM13BIT` (factor `PWM_13BIT_SKALIERUNG`) ⟶ `AR_TO_AD_NUM` ⟶ `DigitalOutput_PWM.OUT`.
5. `DigitalOutput_PWM` (`logiBUS::io::DQ::logiBUS_QDA_PWM`) outputs the value at `Output_PWM`; `QI` is routed directly to `DigitalOutput_PWM.QI` and the event `INIT` to `DigitalOutput_PWM.INIT`; `DigitalOutput_PWM.INITO` is output as `INITO` and `DigitalOutput_PWM.QO` as `QO`.

Technical Details

- There is **no** interlock: the caller must ensure that `HEBEN` and `SENKEN` are never TRUE at the same time, otherwise `AR_AUI_MUX_3` delivers undefined behaviour for `K = 3`
- The plug/variable names `_75`/`_25` are historical; the actually calibrated values at `PWM_POW = 1.0` are 90.0 and 15.0 (see `values_50_75_25`)
- The input `QI` (version 1.1) allows the instance to be kept completely inactive when the channel is used otherwise
- The event input `INIT` (version 1.2) replaces the automatic distribution of the initialisation: the caller triggers it itself (“init like Tecu”), so that `QI` is settled before the physical channel is bound
- The event output `INITO` and the data output `QO` (version 1.3) pass the real confirm and the quality status of `DigitalOutput_PWM` through, so the caller (e.g. `Aufnahme_Option.SUB` via the `Option::AOption` socket) can wait for a real `INIT_CNF` instead of a timing delay
- First caller: `Bunkerboden_PWM.SUB`, which uses only `SENKEN` (full throttle) and ties `HEBEN` permanently to FALSE

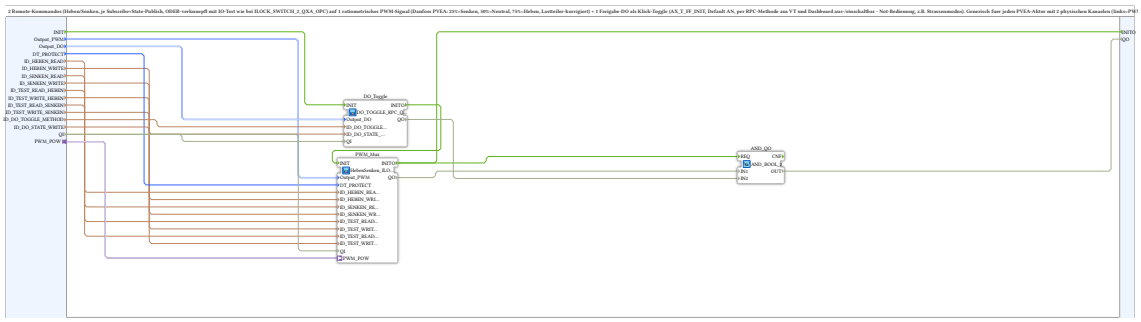
Application Scenarios

- Control of a proportional valve (Danfoss PVEA) with 25/50/75 % PWM for lower/neutral/raise
- Applications with already gated or only one-sidedly used direction signals

Conclusion

`HebenSenken_QDA_PWM` converts two direction booleans without interlock into one of the three ratiometric PWM levels and outputs it scaled at the physical PWM output.

Heben_Senken_TO_QDA_PWM_OPC





• — ○ • — ○ • — ○ •

Introduction

Heben_Senken_TO_QDA_PWM_OPC is a composite subapplication that integrates two independent subapplications to provide a complete control solution for a PVEA actuator with two physical channels. It combines a PWM generation module (HebenSenken_ILOCK_QDA_PWM_OPC) for lifting/lowering commands with a digital output toggle module (DO_TOGGLE_RPC_QXA_OPC) for an enable signal. The subapplication is designed for systems where the module itself has no local visualization terminal, allowing full operation via remote commands, I/O tests, and an RPC-triggered enable toggle.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation; sequenced by the caller instead of automatic distribution, so that `QI` is settled before the physical channel is bound; passed on to `DO_Toggle.INIT`; `DO_Toggle.INITO` starts `PWM_Mux.INIT` (sequential chain)

Event Outputs

- **INITO** (`EInit`): initialisation completed; sequential confirm: `DO_Toggle` first, then `PWM_Mux`; `INITO` comes from `PWM_Mux.INITO` (no rendezvous needed, both are plain physical output channels)

Data Inputs

Name	Type	Description
`QI`	`BOOL`	Qualifier for both physical outputs (PWM + enable DO), default `TRUE`; FALSE keeps the instance completely inactive (no hardware binding) in case the same channel is used differently depending on the machine variant.
`Output_PWM`	`logiBUS::io::DQ::logiBUS_DO_S`	Physical PWM output (left channel): 25% / 50% / 75% duty cycle, ratiometric.
`Output_DO`	`logiBUS::io::DQ::logiBUS_DO_S`	Physical enable output (right channel): click-toggle, default ON.
`DT_PROTECT`	`TIME`	Protection dead time before direction change, default `T#300ms`.
`ID_HEBEN_READ`	`WSTRING`	Actual function command “raise” (subscribe).
`ID_HEBEN_WRITE`	`WSTRING`	Actual function feedback “raise”, behind the ILOCK (publish).
`ID_SENKEN_READ`	`WSTRING`	Actual function command “lower” (subscribe).
`ID_SENKEN_WRITE`	`WSTRING`	Actual function feedback “lower”, behind the ILOCK (publish).
`ID_TEST_READ_HEBEN`	`WSTRING`	Existing IO-test subscribe address for raise (e.g., `STG2_Q0x_READ`), OR-com-

		bined with ID_HEBEN_READ before the ILOCK.
ID_TEST_WRITE_HEBEN	WSTRING	Existing IO-test publish address for raise, behind the ILOCK (e.g., STG2_Q0x_WRITE).
ID_TEST_READ_SENKEN	WSTRING	Existing IO-test subscribe address for lower, OR-combined with ID_SENKEN_READ before the ILOCK.
ID_TEST_WRITE_SENKEN	WSTRING	Existing IO-test publish address for lower, behind the ILOCK.
ID_DO_TOGGLE_METHOD	WSTRING	Local method address (ACTION=CREATE_METHOD) for the argument-less enable toggle trigger – called from STG1 (softkey relay) and directly from the OPC dashboard (CALL_METHOD).
ID_DO_STATE_WRITE	WSTRING	Local publish address (ACTION=WRITE) for the actual enable state (AX_T_FF_INIT.Q) – subscribed remotely from VT and dashboard.

Data Outputs

- **QO** (BOOL): quality output, AND combination (AND_Q0, AND_BOOL_2) of PWM_Mux.Q0 and DO_Toggle.Q0.

There are no further output variables; the physical outputs are controlled directly through the Output_PWM and Output_DO references.

Adapters

None.

Functionality

The subapplication collects remote raise/lower commands and OR-combines them with IO-test signals. The internal PWM subapplication (HebenSenken_ILOCK_QDA_PWM_OPC) generates a single ratiometric PWM signal with a duty cycle of 25% (lower), 50% (neutral), and 75% (raise), incorporating an interlock and a protection dead time. The second internal subapplication (DO_TOGGLE_RPC_QXA_OPC) manages a digital output that can be toggled via an RPC method, defaulting to ON. QI is routed jointly to PWM_Mux.QI and DO_Toggle.QI, and the event INIT to DO_Toggle.INIT; DO_Toggle.INITO triggers PWM_Mux.INIT, so both physical channels are kept inactive together or initialised one after the other by the caller. The confirmation is sequential (no E_REND_2 block): PWM_Mux.INITO → INITO and AND_Q0.REQ. AND_Q0 (iec61131::booleanOperators::AND_BOOL_2) combines PWM_Mux.Q0 (IN1) and DO_Toggle.Q0 (IN2) into the output Q0. The two modules operate independently but are combined here to provide a unified interface for a PVEA actuator with two physical channels.

Technical Features

- **Composite structure**: Encapsulates two purpose-built subapplications into a single logical block.
- **Remote operation**: Supports Subscribe/Publish for command and state feedback.
- **IO-test integration**: Test signals are OR-combined with actual commands, enabling commissioning and maintenance overrides.
- **PWM generation**: Includes a programmable protection dead time (DT_PROTECT) to prevent rapid direction changes.
- **Enable toggle**: A latchable output (default ON) controlled via an RPC method, suitable for emergency stop or street mode.
- **Generic applicability**: Works with any PVEA actuator having two physical channels (left = PWM, right = enable), independent of a local visualization terminal.

State Overview

The subapplication itself does not implement an external state machine; the internal subapplications are responsible for state handling. The PWM module likely has states for neutral, raise, lower, and protection delay. The DO toggle module uses a flip-flop (`AX_T_FF_INIT`) to maintain its ON/OFF state. The composite ensures that the physical outputs reflect the combined behavior.

Application Scenarios

- **Centralized control**: Operate a Danfoss PVEA actuator lifting/lowering from a main control station without module-local visualization.
- **Remote enable control**: Toggle the enable output from a dashboard or a VT using the RPC method, e.g., to activate a street mode or an emergency stop.
- **IO testing**: Use the integrated IO-test signals to validate PWM and enable outputs during commissioning.
- **Generic usage**: Apply to any system requiring a PWM drive plus a latching enable output, with configurable dead time and command IDs.

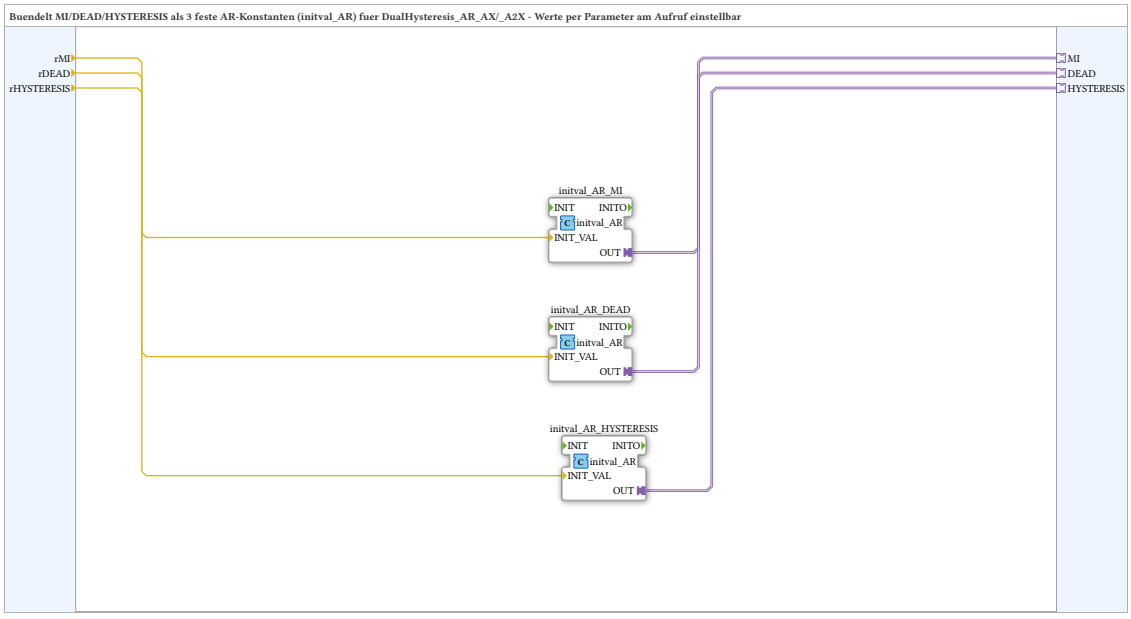
Comparison with Similar Blocks

- **Integrated vs. separate blocks**: This composite combines two essential functions (PWM control and enable toggle) in one interface, whereas separate FBs would require explicit wiring between two blocks.
- **Unified channel handling**: Directly maps the two physical channels (left PWM, right enable) to a single input set, simplifying application engineering.
- **RPC-based toggle**: Unlike simple DO control blocks, the enable output includes a latchable flip-flop triggered via an RPC method, offering more flexible remote operation.

Conclusion

Heben_Senken_TO_QDA_PWM_OPC is a composite subapplication that encapsulates the complexity of controlling a PVEA actuator with a PWM signal and a toggleable enable output. It is generic, configurable via data inputs, and provides a clean interface for modules without a local visualization, enabling remote operation, IO testing, and flexible enable control.

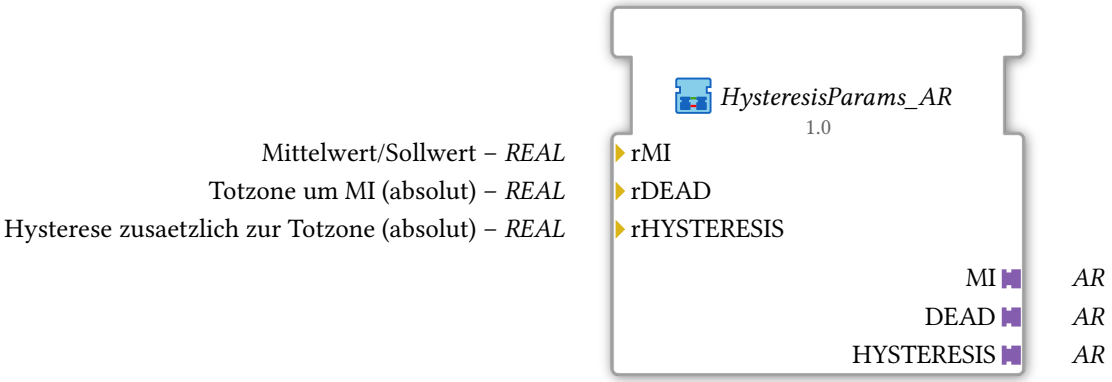
HysteresisParams_AR



• — ○ • — ○ • — ○ •

Introduction

`HysteresisParams_AR` bundles the three parameters `MI` (mean/target value), `DEAD` (dead zone), and `HYSTERESIS` (additional hysteresis) as fixed `AR` adapter constants for `DualHysteresis_AR_AX`/`DualHysteresis_AR_A2X`. Instead of repeating three `initval_AR` instances individually in each exercise/subapp, this building block combines them.



Function Blocks (FBs) Used

Sub-Blocks: HysteresisParams_AR

- **Type**: SubAppType
- **Internal FBs Used**:
- **initval_AR_MI** / **initval_AR_DEAD** / **initval_AR_HYSTERESIS**: each `adapter::types::unidirectional::AR::initval::initval_AR` — converts a `REAL` input value into a fixed `AR` adapter plug.
- **How it Works**: The three `InputVars`, `rMI`, `rDEAD`, and `rHYSTERESIS` are each converted into the corresponding `AR` plugs (`MI`, `DEAD`, and `HYSTERESIS`) via a `initval_AR` instance.

Program Flow and Connections

1. `rMI` ⟶ `initval_AR_MI.INIT_VAL`; `rDEAD` ⟶ `initval_AR_DEAD.INIT_VAL`; `rHYSTERESIS` ⟶ `initval_AR_HYSTERESIS.INIT_VAL` (Data connections).
2. `initval_AR_MI.OUT` ⟶ `MI`; `initval_AR_DEAD.OUT` ⟶ `DEAD`; `initval_AR_HYSTERESIS.OUT` ⟶ `HYSTERESIS` (Adapter connections, directly to the SubApp interface).

Technical Features

- **Configurable Default Values**: `rMI`/`rDEAD`/`rHYSTERESIS` have meaningful default values (`500.0`/`20.0`/`30.0`) that can be overridden via parameters when calling the function—similar to `u16ObjId` in the GreenWhiteBackground function blocks.
- **Reuse Instead of Repetition**: Replaces the manual duplication of three `initval_AR` instances, as was initially done in `Uebung_234_AX`/`Uebung_235_AX`.

Application Scenarios

- Parameterization of `DualHysteresis_AR_AX`/`DualHysteresis_AR_A2X` when setpoint, dead zone, and hysteresis are required as fixed constants that can be overridden during the call.
- Consistent parameterization of several comparable exercises (e.g., `Uebung_234_AX` for hardware and `Uebung_235_AX` for VT) that should remain directly comparable with identical values.

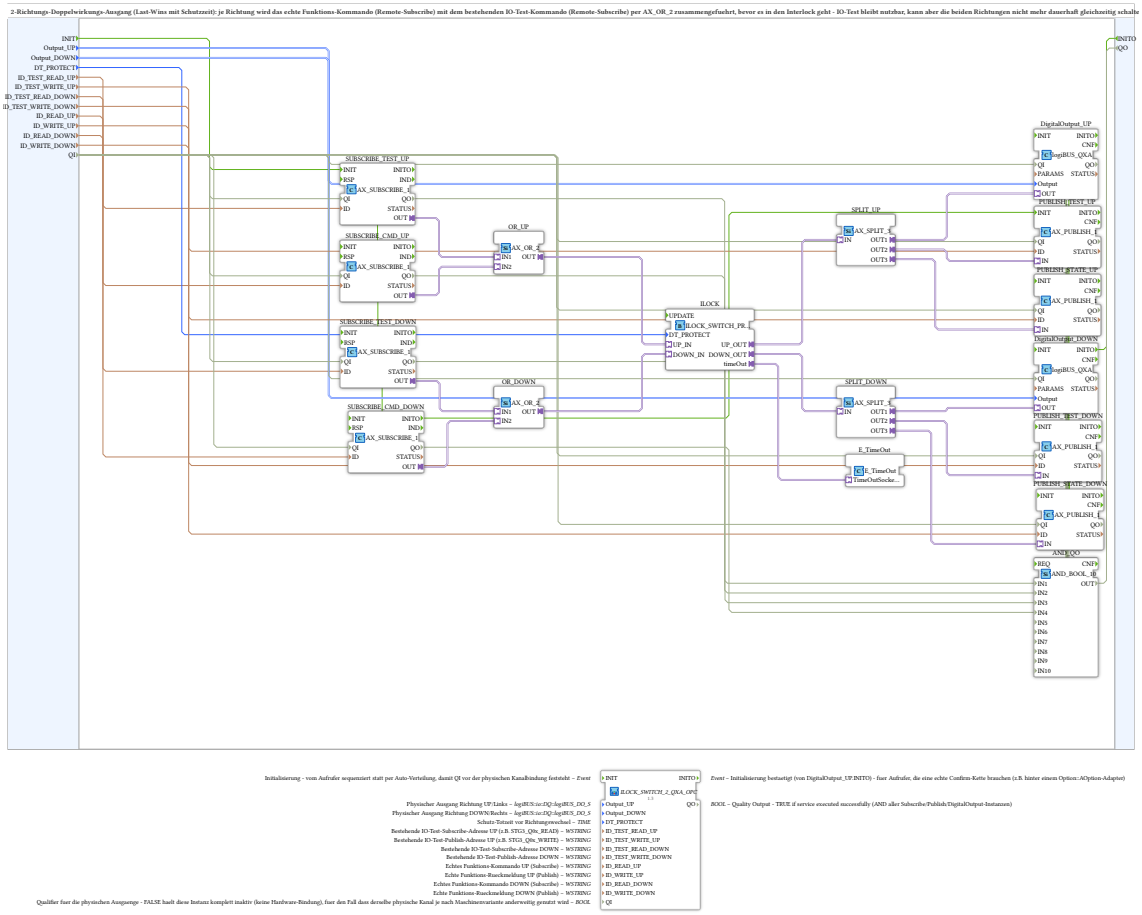
Summary

`HysteresisParams_AR` combines three fixed `AR` parameter constants for the hysteresis blocks into a single, parameterizable block.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

ILOCK_SWITCH_2_QXA_OPC



• – ○ • – ○ • – ○ •

Introduction

The ILOCK_SWITCH_2_QXA_OPC subapplication implements a 2-direction double-acting digital output with a “last-wins” interlock and configurable protection time. For each direction, the real function command (remote subscribe) is merged with the existing IO-test command (remote subscribe) using an AX_OR_2 block before entering the interlock logic. This keeps the IO-test fully usable while preventing both directions from being permanently switched simultaneously. The interlocked output drives the physical logiBUS channel and reports its actual state both to the existing IO-test monitoring and to the real function feedback (e.g., GreenWhiteBackground on a SoftKey). It is generic for any “digital DW” actuator operated from a module without its own visualization terminal.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation; sequenced by the caller instead of automatic distribution, so that `QI` is settled before the physical channel is bound; starts the initialisation chain through all inner instances (`SUBSCRIBE_TEST_UP` ⟶ `SUBSCRIBE_CMD_UP` ⟶ `SUBSCRIBE_TEST_DOWN` ⟶ `SUBSCRIBE_CMD_DOWN` ⟶ `PUBLISH_TEST_UP` ⟶ `PUBLISH_STATE_UP` ⟶ `PUBLISH_TEST_DOWN` ⟶ `PUBLISH_STATE_DOWN` ⟶ `DigitalOutput_UP` ⟶ `DigitalOutput_DOWN`); the physical outputs are therefore initialised last. The caller must trigger `INIT` itself (e.g. without delay for unchanged immediate behaviour, or delayed if an INI value has to be awaited).

Event Outputs

- **INITO** (`EInit`): initialisation confirmed (from `DigitalOutput_DOWN.INITO`, the end of the chain); for callers that need a real confirm chain (e.g. behind an `Option::AOption` adapter). Purely additive, harmless if left unwired.

Data Inputs

- **QI** (BOOL): Qualifier for the physical outputs; FALSE keeps the instance completely inactive (no hardware binding). Default `TRUE`. Distributed to all subscribe, publish and DigitalOutput instances.
- **Output_UP** (logiBUS::io::DQ::logiBUS_DO_S): Physical output channel for the UP/Left direction. Initial value: `logiBUS_DO::Invalid`.
- **Output_DOWN** (logiBUS::io::DQ::logiBUS_DO_S): Physical output channel for the DOWN/Right direction. Initial value: `logiBUS_DO::Invalid`.
- **DT_PROTECT** (TIME): Protection dead time before a direction change is allowed. Default: `T#300ms`.
- **ID_TEST_READ_UP** (WSTRING): Existing IO-test subscribe address for the UP direction (e.g., `STG3_Q0x_READ`).
- **ID_TEST_WRITE_UP** (WSTRING): Existing IO-test publish address for the UP direction (e.g., `STG3_Q0x_WRITE`).
- **ID_TEST_READ_DOWN** (WSTRING): Existing IO-test subscribe address for the DOWN direction.
- **ID_TEST_WRITE_DOWN** (WSTRING): Existing IO-test publish address for the DOWN direction.
- **ID_READ_UP** (WSTRING): Real function command subscribe address for the UP direction.
- **ID_WRITE_UP** (WSTRING): Real function feedback publish address for the UP direction.
- **ID_READ_DOWN** (WSTRING): Real function command subscribe address for the DOWN direction.
- **ID_WRITE_DOWN** (WSTRING): Real function feedback publish address for the DOWN direction.

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if the service was executed successfully (AND of all subscribe/publish/DigitalOutput instances).

Adapters

None.

Functionality

For each direction (UP and DOWN), the subapplication subscribes to two independent remote command sources via `AX_SUBSCRIBE_1` blocks:

1. The existing IO-test command (`ID_TEST_READ_*`).
2. The real function command (`ID_READ_*`).

Both signals are combined using an `AX_OR_2` block, so that either the IO-test command or the real function command can request the activation of that direction. The combined requests are then passed to an `ILOCK_SWITCH_PROTECT_AX` interlock block, which enforces a “last-wins” policy and a configurable protection time (`DT_PROTECT`). This ensures that both directions can never be active simultaneously and that a minimum dead time elapses before switching directions.

The interlocked output for each direction is distributed via an `AX_SPLIT_3` block into three paths:

- **Physical output**: Drives the logiBUS digital output channel (`DigitalOutput_UP` / `DigitalOutput_DOWN`).
- **IO-test feedback**: Publishes the actual state to the existing IO-test publish address (`ID_TEST_WRITE_*`).
- **Function feedback**: Publishes the actual state to the real function feedback address (`ID_WRITE_*`, e.g., for SoftKey background indication).

Initialisation and quality: `INIT` runs as a chain `SUBSCRIBE_TEST_UP` → `SUBSCRIBE_CMD_UP` → `SUBSCRIBE_TEST_DOWN` → `SUBSCRIBE_CMD_DOWN` → `PUBLISH_TEST_UP` → `PUBLISH_STATE_UP` → `PUBLISH_TEST_DOWN` → `PUBLISH_STATE_DOWN` → `DigitalOutput_UP` → `DigitalOutput_DOWN` (each `INITO` → next `INIT`); `DigitalOutput_DOWN.INITO` goes to `INITO` and to `AND_QO.REQ`. `QI` is distributed to all eight subscribe/publish instances and to both `DigitalOutput` instances. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_10`), which combines the `QO` of these ten instances.

The `E_TimeOut` block is connected to the interlock’s timeout event to trigger and manage the protection timer logic.

Technical Features

- Merging of IO-test and real function commands using `AX_OR_2` logic blocks.
- Last-wins interlocking via the `ILOCK_SWITCH_PROTECT_AX` block.
- Configurable protection dead time (`DT_PROTECT`), defaulting to 300 ms.
- Remote subscribe (`AX_SUBSCRIBE_1`) and publish (`AX_PUBLISH_1`) capabilities for integration into distributed control networks.

- Signal fan-out via AX_SPLIT_3 to drive the physical output, IO-test monitoring, and function feedback simultaneously.
- Fully generic design for any "digital DW" actuator, with all addresses parameterized via WSTRING inputs.
- Quality output QO as the AND of the ten inner QO outputs.
- Only INIT/INITO as events at the interface level; all other logic is driven internally by the subscribed data.

State Overview

The subapplication relies on the internal state machine of the ILOCK_SWITCH_PROTECT_AX interlock block. The relevant operating states are:

- **Idle**: No direction is requested; both outputs are inactive.
- **UP active**: The UP direction is interlocked and active; the DOWN direction is blocked.
- **DOWN active**: The DOWN direction is interlocked and active; the UP direction is blocked.
- **Switching with protection time**: After a direction change request, the protection timer (DT_PROTECT) runs before the new direction is applied, preventing rapid toggling.

Application Scenarios

- Controlling a double-acting digital actuator (e.g., a motorized valve or damper) from a control module without a local visualization terminal.
- Integrating existing IO-test functionality with real remote function commands, ensuring that both can drive the output while maintaining interlock safety.
- Providing feedback to both the legacy IO-test monitoring system and the operator interface (e.g., SoftKey background color) with the actual interlocked output state.
- Generic use in any "digital DW" actuator system where two directional commands must be merged and interlocked.

Comparison with Similar Blocks

- Compared to a simple logiBUS_QXA direct output, this subapplication adds interlock logic and protection timing.
- Compared to an interlock block that only accepts single-command inputs, this block explicitly merges remote subscribe/publish channels for both IO-test and real function commands.
- The use of AX_OR_2 before the interlock distinguishes it from pure interlock variants, enabling both test and functional commands to share the same interlock protection path.

Conclusion

ILOCK_SWITCH_2_QXA_OPC provides a robust and generic solution for driving a double-acting digital output with interlocking, protection time, and merged IO-test/function command handling. It preserves the existing IO-test infrastructure while guaranteeing safe operation and consistent feedback through both test and functional channels, making it suitable for a wide range of actuator control applications.

INI_IN_AND_STORE / NVS_IN_AND_STORE: Common Pattern

• – ◦ • – ◦ • – ◦ •

Introduction

MyLib::sys contains a family of function blocks that **persistently store** a value entered via VT and automatically reload it during initialization (INIT)—either in an INI file (INI_IN_AND_STORE_*) or in ESP32 flash memory (NVS, NVS_IN_AND_STORE_*). This page explains the common pattern.

Naming Scheme

{INI|NVS}_IN_AND_STORE_<Typ>, where <Typ> determines the data type: AIS (string adapter), AR (REAL adapter, with physical scaling NumericObjectPool_S), AUDI (UDINT adapter) in test_AX; UDINT (classic, without adapter) in test_B.

Functionality (test_AX, adapter-based)

1. A VT input field (StringValue_AIS/NumericValue_PHYSA/...) returns the value entered by the user as an adapter.

2. INI_<Typ>/NVS_<Typ> (eclipse4diac::storage::INI_* or logiBUS::storage::esp32_nvs::NVS_*) persistently stores this value under KEY/SECTION (INI only) and returns the last saved value on INIT (SETM=TRUE: saving active, DEFAULT_VALUE: initial value if nothing has been saved yet).
3. The (newly loaded or newly entered) value is distributed via <Typ>_SPLIT_2: once as a plug VALUEO to the outside (for further use in the calling network) and once to Q_StringValue_AIS/Q_NumericValue_PHYSA (written back to the VT display field so that the input field and display remain synchronized).

Functionality (test_B, classic)

The test_B variant (INI_IN_AND_STORE_UDINT/NVS_IN_AND_STORE_UDINT) is older and uses classic event/data connections instead of adapters: A NumericValue_ID input field returns a DWORD value, which is converted to UDINT via F_DWORD_TO_UDINT and passed to INI/NVS (generic, non-type-specific blocks). The stored value is sent out as VALUEO and back to Q_NumericValue for display.

Summary

Both approaches solve the same problem—"remembering a user-entered value across restarts"—using the same basic pattern (input field → memory function block with KEY/SECTION → branching to output + display writeback), only with different storage locations (INI file vs. NVS flash) and different wiring styles (adapter in test_AX, classic in test_B).

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_IN_AND_STORE / NVS_IN_AND_STORE: Shared Pattern

• – ◦ • – ◦ • – ◦ •

Introduction

MyLib::sys contains a family of blocks that **persistently store** a value entered via the VT and automatically reload it on deployment — either in an INI file (INI_IN_AND_STORE_*) or in ESP32 flash (NVS, NVS_IN_AND_STORE_*). This page explains the shared pattern.

Naming Scheme

{INI|NVS}_IN_AND_STORE_<type>, where <type> determines the data type: AIS (string adapter), AR (REAL adapter, with physical scaling NumericObjectPool_S), AUDI (UDINT adapter) in test_AX; UDINT (classic, no adapter) in test_B.

Functionality (test_AX, adapter-based)

1. A VT input field (StringValue_AIS/NumericValue_PHYSA/...) delivers the user-entered value as an adapter.
2. INI_<type>/
   NVS_<type> (eclipse4diac::storage::INI_* or logiBUS::storage::esp32_nvs::NVS_*) stores this value persistently under KEY/SECTION (INI only) and returns the last stored value on INIT (SETM=TRUE: storing active, DEFAULT_VALUE: initial value if nothing has been stored yet).
3. The (newly loaded or newly entered) value is split via <type>_SPLIT_2: once as plug VALUEO outward (for reuse in the calling network) and once to Q_StringValue_AIS/Q_NumericValue_PHYSA (writing back to the VT display field, so the input field and display stay in sync).

Functionality (test_B, classic)

The test_B variant (INI_IN_AND_STORE_UDINT/NVS_IN_AND_STORE_UDINT) is older and uses classic event/data connections instead of adapters: a NumericValue_ID input field delivers a DWORD value, converted to UDINT via F_DWORD_TO_UDINT and passed to INI/NVS (generic, non-type-specific blocks). The stored value goes out as VALUEO and back to Q_NumericValue for display.

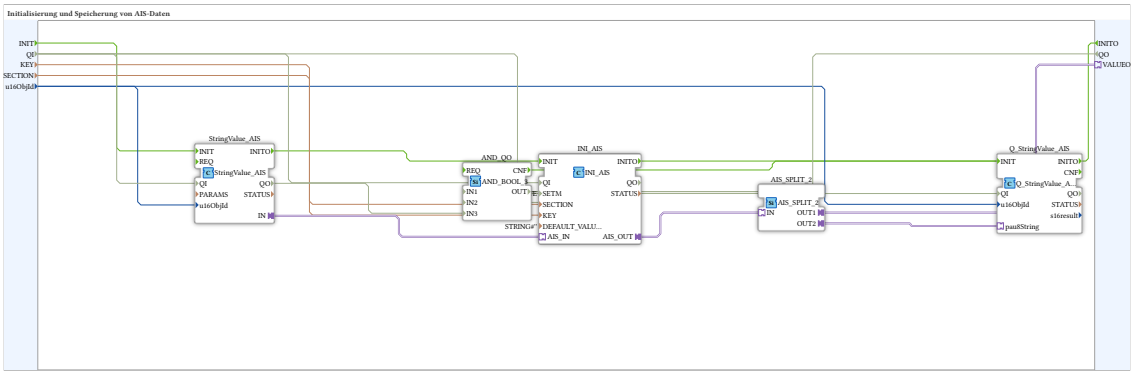
Summary

Both variants solve the same problem — "remember a user-entered value across restarts" — with the same basic pattern (input field → storage FB with KEY/SECTION → split into output + display write-back), differing only in storage location (INI file vs. NVS flash) and wiring style (adapters in test_AX, classic in test_B).

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_IN_AND_STORE_AIS



• – ○ • – ○ • – ○ •

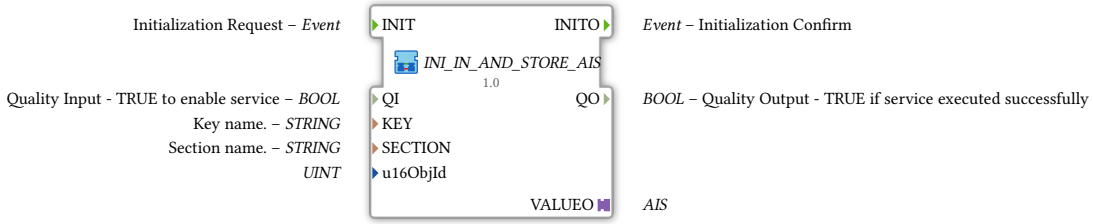
Introduction

`INI_IN_AND_STORE_AIS` persistently stores a string value entered via the VT (AIS adapter) in an INI file and automatically reloads it on deployment.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).

Interface (quality pattern): the data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully") are connected to all three internal service blocks (`StringValue_AIS`, `INI_AIS`, `Q_StringValue_AIS`): `QI` ⟶ `Q_StringValue_AIS.QI`, `INI_AIS.QI` and `StringValue_AIS.QI`; `QO` is the AND (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_3`) of `Q_StringValue_AIS.QO`, `INI_AIS.QO` and `StringValue_AIS.QO`.

In addition, the block has the events `INIT` (input, `EInit`) and `INITO` (output, `EInit`); initialization is chained: `INIT` ⟶ `StringValue_AIS.INIT`; `StringValue_AIS.INITO` ⟶ `INI_AIS.INIT`; `INI_AIS.INITO` ⟶ `Q_StringValue_AIS.INIT`; `Q_StringValue_AIS.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`).



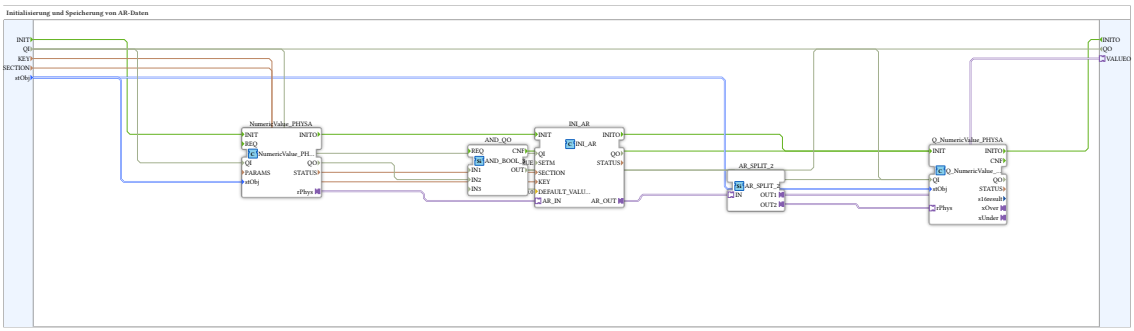
Summary

AIS (string) variant of the INI storage family.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_IN_AND_STORE_AR



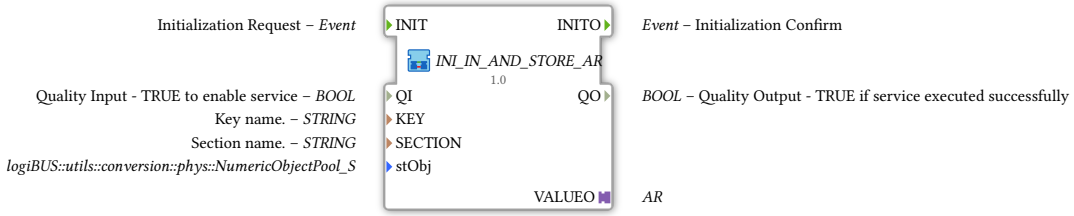
• – ○ • – ○ • – ○ •

Introduction

`INI_IN_AND_STORE_AR` persistently stores a physically scaled REAL value entered via the VT (AR adapter, `NumericValue_PHYSA`/`Q_NumericValue_PHYSA` with `NumericObjectPool_S` scaling: `r32Scale`, `i32Offset`, `u8Decimals`) in an INI file.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).

Interface (quality pattern): event input `INIT` and event output `INITO` (each `EInit`), data input `QI` (`BOOL`, default `TRUE`, "Quality Input - TRUE to enable service") and data output `QO` (`BOOL`, "Quality Output - TRUE if service executed successfully"). Initialization: `INIT` → `NumericValue_PHYSA.INIT`; `NumericValue_PHYSA.INITO` → `INI_AR.INIT`; `INI_AR.INITO` → `Q_NumericValue_PHYSA.INIT`; `Q_NumericValue_PHYSA.INITO` → `INITO` and → `AND_QO.REQ`. Quality: `QI` → `Q_NumericValue_PHYSA.QI`, `NumericValue_PHYSA.QI`, `INI_AR.QI`; `Q_NumericValue_PHYSA.QO`, `NumericValue_PHYSA.QO`, `INI_AR.QO` → `AND_QO.IN1` to `AND_QO.IN3` (in this order); `AND_QO.OUT` → `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_3`).



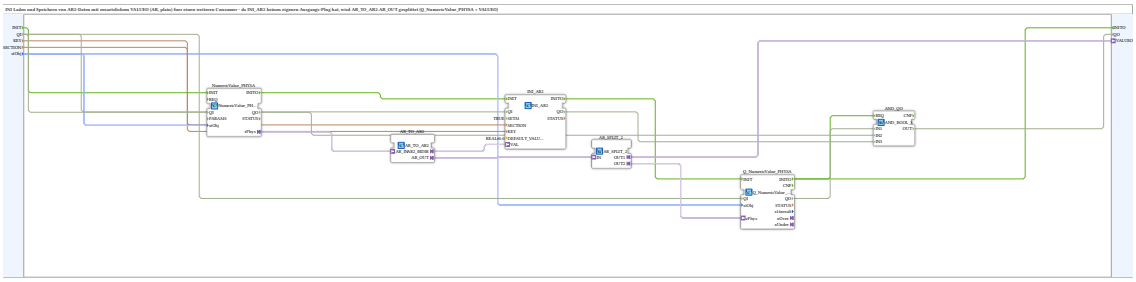
Summary

AR (physically scaled REAL) variant of the INI storage family.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_IN_AND_STORE_AR2



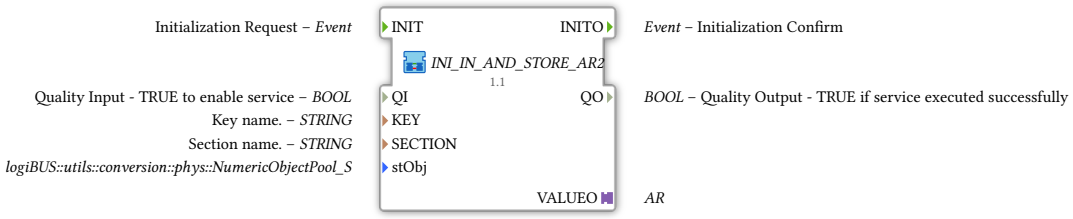
• – ○ • – ○ • – ○ •

Introduction

`INI_IN_AND_STORE_AR2` is the AR2-adapter variant of `INI_IN_AND_STORE_AR`: a physically scaled REAL value entered via the VT (`NumericValue_PHYSA`) is persistently stored in an INI file

(`eclipse4diac::storage::INI_AR2`, storage via the AR2 adapter socket instead of a plain value). Since `INI_AR2` has no output plug of its own, the `AR_TO_AR2` output is duplicated via `AR_SPLIT_2`: once back to the VT display (`Q_NumericValue_PHYSA`), once outward on `VALUEO` for a further consumer.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).



Technical notes

- `SETM=TRUE`: a live change during operation is confirmed/echoed immediately, not only at boot - important since the fresh value can be reused in the same run via `VALUEO`.
- `AR_TO_AR2` converts the plain-AR value from the VT input field into the bidirectional AR2 adapter that `INI_AR2` expects for storage.

Quality ports (QI/QO)

The block has the event input `INIT` (`EInit`, Initialization Request), the event output `INITO` (`EInit`, Initialization Confirm), the data input `QI` (`BOOL`, default `TRUE`, Quality Input - TRUE enables the service) and the data output `QO` (`BOOL`, Quality Output - TRUE if the service was executed successfully).

- Initialization chain: `INIT` ⟶ `NumericValue_PHYSA.INIT`, `NumericValue_PHYSA.INITO` ⟶ `INI_AR2.INIT`, `INI_AR2.INITO` ⟶ `Q_NumericValue_PHYSA.INIT`, `Q_NumericValue_PHYSA.INITO` ⟶ `INITO` (and additionally ⟶ `AND_QO.REQ`)
- `QI` is connected to `Q_NumericValue_PHYSA.QI`, `NumericValue_PHYSA.QI` and `INI_AR2.QI`
- `QO` is the AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_3`) of `Q_NumericValue_PHYSA.QO`, `NumericValue_PHYSA.QO` and `INI_AR2.QO`

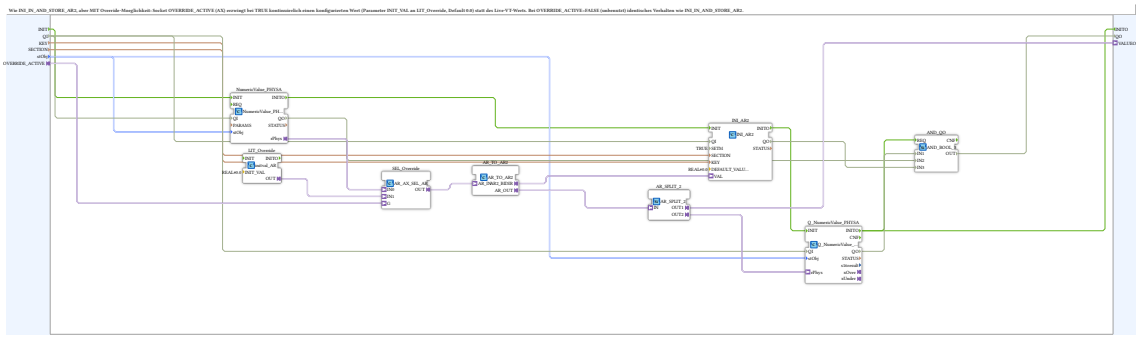
Summary

AR2-adapter variant of the INI storage family, with an additional `VALUEO` output for a further consumer.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_IN_AND_STORE_AR2_OVERRIDE



• – ○ • – ○ • – ○ •

Introduction

`INI_IN_AND_STORE_AR2_OVERRIDE` is like `INI_IN_AND_STORE_AR2`, but with an override option: while the `OVERRIDE_ACTIVE` socket (AX) is continuously TRUE, a configured value is forced instead of the live VT value (parameter `INIT_VAL` on `LIT_Override`, default `0.0`). With `OVERRIDE_ACTIVE`=FALSE (unused) the behavior is identical to `INI_IN_AND_STORE_AR2`.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).



Interface

Input variables

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE enables the service; passed through to `NumericValue_PHYSA`, `INI_AR2` and `Q_NumericValue_PHYSA`.
- `KEY` (STRING): key name in the INI file.
- `SECTION` (STRING): section name in the INI file.
- `stObj` (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): VT object with `u16ObjId`, `r32Scale`, `i32Offset`, `u8Decimals` (default `u16ObjId := ID_NULL`, `r32Scale := 1.0`, `i32Offset := 0`, `u8Decimals := 0`).

Output variables

- `QO` (BOOL): Quality Output - TRUE if the service was executed successfully; `AND` of the `QO` outputs of `Q_NumericValue_PHYSA`, `NumericValue_PHYSA` and `INI_AR2` (via `AND_QO`).

Adapters

- **Plug `VALUEO`** (`adapter::types::unidirectional::AR`): value for a further consumer.
- **Socket `OVERRIDE_ACTIVE`** (`adapter::types::unidirectional::AX`): when continuously TRUE, the override value is forced.

Events

- `INIT` (EInit, input): initialization request; starts the chain `NumericValue_PHYSA` → `INI_AR2` → `Q_NumericValue_PHYSA`.
- `INITO` (EInit, output): initialization confirmation; comes from `Q_NumericValue_PHYSA.INITO`.

How it works

1. `NumericValue_PHYSA` reads the VT input field as a physical value (`rPhys`).
2. `SEL_Override` (`AR_AX_SEL_AR`) selects: `IN0` = live VT value, `IN1` = override value from `LIT_Override` (`initval_AR`, `INIT_VAL=REAL#0.0`), selected via `G` = `OVERRIDE_ACTIVE`.
3. `AR_TO_AR2` converts into the bidirectional AR2 adapter for `INI_AR2` (`eclipse4diac::storage::INI_AR2`; `SETM`=TRUE, `DEFAULT_VALUE=REAL#0.0`; `KEY`/`SECTION` from outside).
4. `AR_TO_AR2.AR_OUT` is duplicated via `AR_SPLIT_2`: `OUT1` to `VALUEO`, `OUT2` to `Q_NumericValue_PHYSA` (VT display, `stObj`).

`QI` is passed through to `NumericValue_PHYSA.QI`, `INI_AR2.QI` and `Q_NumericValue_PHYSA.QI` (no fixed `QI`=TRUE any more). Initialization runs as the chain `INIT` → `NumericValue_PHYSA.INIT`, `NumericValue_PHYSA.INITO` → `INI_AR2.INIT`, `INI_AR2.INITO` → `Q_NumericValue_PHYSA.INIT`, `Q_NumericValue_PHYSA.INITO` → `INITO` and → `AND_QO.REQ`. `QO` is the `AND` (`AND_QO`, `AND_BOOL_3`) of the `QO` outputs of `Q_NumericValue_PHYSA`, `NumericValue_PHYSA` and `INI_AR2`.

Technical notes

- Unlike the AUDI2 variant (`INI_IN_AND_STORE_AUDI2_OVERRIDE`, enum/dropdown, 1=disabled), AR is a continuous physical value: the override value has no fixed meaning but is freely configurable via `INIT_VAL` on `LIT_Override`.
- `SETM`=TRUE: the boot value loads automatically (INIT → INITO → GET → GETO, independent of `SETM`); `SETM`=TRUE additionally ensures that a live change during operation is confirmed/echoed immediately (e.g. for further use via `VALUEO`).

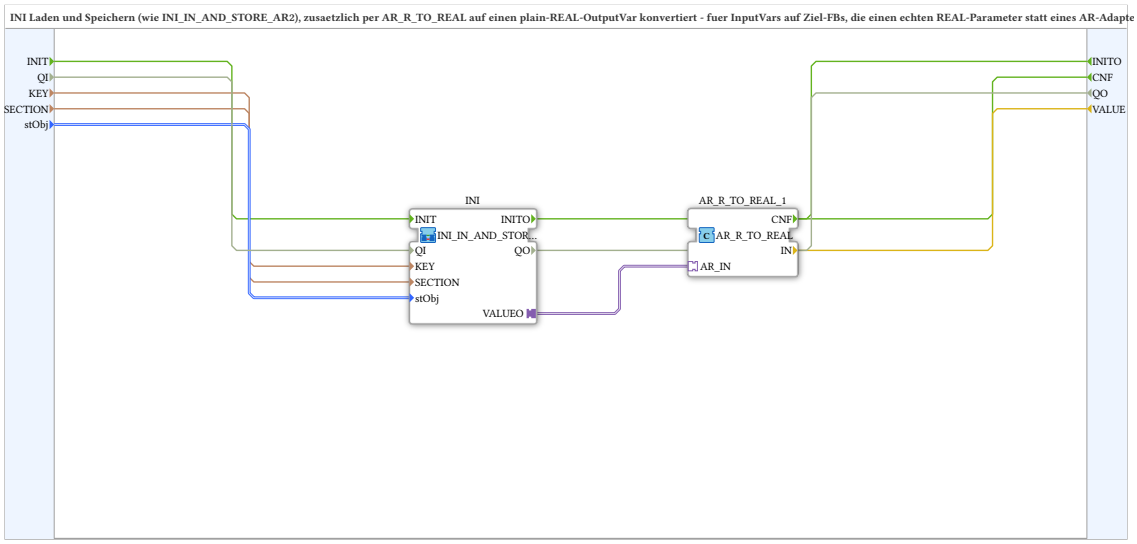
Summary

AR2 storage block with a switchable, configurable override value.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

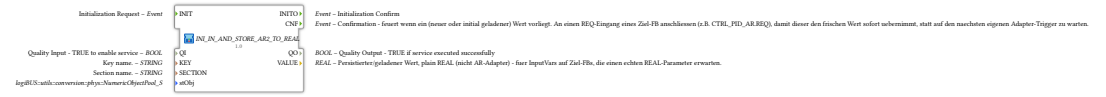
INI_IN_AND_STORE_AR2_TO_REAL



• – ○ • – ○ • – ○ •

Introduction

`INI_IN_AND_STORE_AR2_TO_REAL` loads and stores a value in the INI file (like `INI_IN_AND_STORE_AR2`) and additionally converts it to a plain REAL output variable via `AR_R_TO_REAL`. This is needed for input variables of target blocks that expect a real REAL parameter instead of an AR adapter socket (e.g. `CTRL_PID_AR.KP/TN/TV`).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; passed on to `INI.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `INI.INITO`
- **CNF** (confirmation): fires when a (new or initially loaded) value is available. Connect it to a `REQ` input of a target FB (e.g. `CTRL_PID_AR.REQ`) so that it takes over the fresh value immediately instead of waiting for its own next adapter trigger

Data Inputs

- **QI** (BOOL, initial value `TRUE`): quality input, enables the service; passed on to `INI.QI`
- **KEY** (STRING): key name in the INI file
- **SECTION** (STRING): section name in the INI file
- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): object pool properties of the VT number variable (default: `u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0`)

Data Outputs

- **QO** (BOOL): quality output; comes from `INI.QO`
- **VALUE** (REAL): persisted/loaded value as a plain REAL (no AR adapter)

Adapters

No adapters are used.

Functionality

The network contains two instances:

- `INI` (`MyLib::sys::INI_IN_AND_STORE_AR2`): `KEY` ⟶ `INI.KEY`, `SECTION` ⟶ `INI.SECTION`, `stObj` ⟶ `INI.stObj`. The value is available at `INI.VALUEO` (AR).
- `AR_R_TO_REAL_1` (`adapter::conversion::unidirectional::AR_R_TO_REAL`): `INI.VALUEO` ⟶ `AR_R_TO_REAL_1.AR_IN`; `AR_R_TO_REAL_1.IN` ⟶ `VALUE` and `AR_R_TO_REAL_1.CNF` ⟶ `CNF`.

Initialization and quality are passed straight through: `INIT` ⟶ `INI.INIT`, `INI.INITO` ⟶ `INITO`, `QI` ⟶ `INI.QI` and `INI.QO` ⟶ `QO`.

Technical Details

- Extracted from `Aufnahme_Automatik.SUB`, where this combination was wired three times identically (KP/TN/TV)
- `CNF` is generated only after the AR-to-REAL conversion and thus signals that `VALUE` is up to date

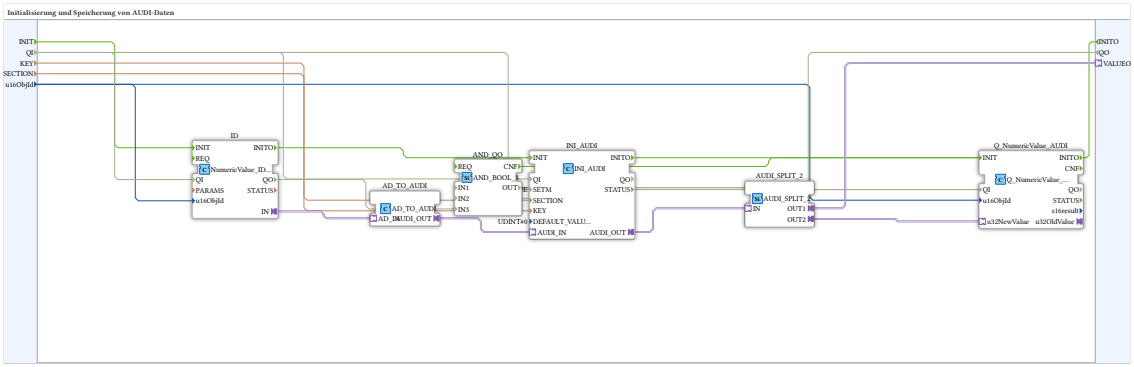
Application Scenarios

- Persisted controller parameters (e.g. KP/TN/TV of a PID controller) that are connected to REAL inputs of target FBs

Conclusion

`INI_IN_AND_STORE_AR2_TO_REAL` combines INI persistence with an AR-to-REAL conversion and delivers the value together with a `CNF` event at a plain REAL output variable.

INI_IN_AND_STORE_AUDI

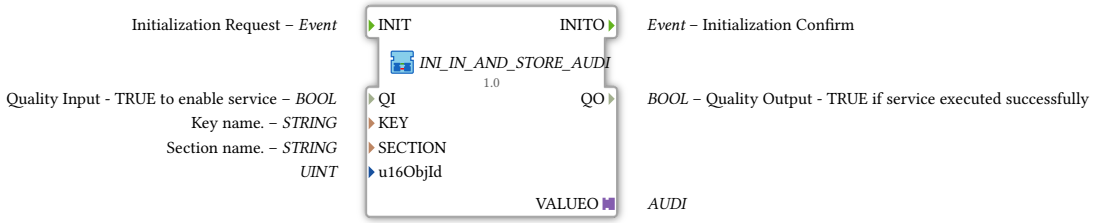


• – ○ • – ○ • – ○ •

Introduction

`INI_IN_AND_STORE_AUDI` persistently stores a UDINT value entered via the VT (AUDI adapter) in an INI file.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).



Event and quality ports (INIT/INITO, QI/QO)

The block has the event input `INIT` and the event output `INITO` (both `EInit`), the data input `QI` (`BOOL`, default `TRUE`, Quality Input - TRUE enables the service) and the data output `QO` (`BOOL`, Quality Output - TRUE if the service was executed successfully).

Initialization runs as a chain through the inner blocks: `INIT` ⟶ `ID.INIT`, `ID.INITO` ⟶ `INI_AUDI.INIT`, `INI_AUDI.INITO` ⟶ `Q_NumericValue_AUDI.INIT`, `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`). `QI` is passed to `ID`, `INI_AUDI` and `Q_NumericValue_AUDI`. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`), which combines `Q_NumericValue_AUDI.QO`, `INI_AUDI.QO` and `ID.QO`.

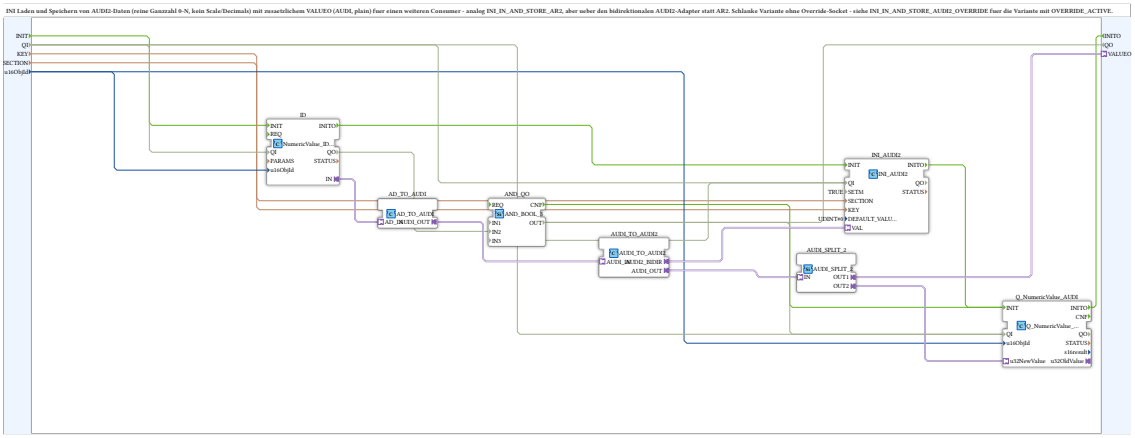
Summary

AUDI (UDINT) variant of the INI storage family.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_IN_AND_STORE_AUDI2



• — ○ • — ○ • — ○ •

Introduction

`INI_IN_AND_STORE_AUDI2` loads and stores `AUDI2` data (plain integer 0..N, without scale/decimals) persistently in an INI file and additionally provides an output `VALUEO` (`AUDI`, plain) for a further consumer. The block is analogous to `INI_IN_AND_STORE_AR2` but works via the bidirectional `AUDI2` adapter instead of `AR2`. The general pattern is described in INI_IN_AND_STORE / NVS_IN_AND_STORE a variant with `AUDI` is `INI_IN_AND_STORE_AUDI`.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirmation

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **KEY** (`STRING`): key name in the INI file
- **SECTION** (`STRING`): section name in the INI file
- **u16ObjId** (`UINT`): object ID of the VT number variable (default `ID_NULL`)

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully (AND of `Q_NumericValue_AUDI.QO`, `ID.QO` and `INI_AUDI2.QO`)

Adapters

- **VALUEO** (plug, `adapter::types::unidirectional::AUDI`): value for a further consumer

Function Blocks (FBs) Used

- **`ID`** (`isobus::UT::io::NumericValue::NumericValue_IDA`): delivers the VT value entered by the operator
- **`AD_TO_AUDI`** (`adapter::conversion::unidirectional::AD_TO_AUDI`): converts the `AD` value to `AUDI`
- **`AUDI_TO_AUDI2`** (`adapter::conversion::bidirectional::AUDI_TO_AUDI2`): converts `AUDI` into the bidirectional `AUDI2` adapter
- **`INI_AUDI2`** (`eclipse4diac::storage::INI_AUDI2`): `SETM = TRUE`, `DEFAULT_VALUE = UDINT#0`; loads and stores the value
- **`AUDI_SPLIT_2`** (`adapter::events::unidirectional::AUDI_SPLIT_2`): duplicates the value
- **`Q_NumericValue_AUDI`** (`isobus::UT::Q::Q_NumericValue_AUDI`): writes the value into the VT number variable
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_3`): combines the `QO` outputs of `Q_NumericValue_AUDI`, `ID` and `INI_AUDI2`

Functionality

- `ID.IN` ⟶ `AD_TO_AUDI.AD_IN` ⟶ `AUDI_TO_AUDI2.AUDI_IN`
- `AUDI_TO_AUDI2.AUDI2_BIDIR` ⟶ `INI_AUDI2.VAL` (load/store)
- `AUDI_TO_AUDI2.AUDI_OUT` ⟶ `AUDI_SPLIT_2.IN`
- `AUDI_SPLIT_2.OUT1` ⟶ `VALUEO`; `AUDI_SPLIT_2.OUT2` ⟶ `Q_NumericValue_AUDI.u32NewValue` (VT display)
- `KEY`, `SECTION` ⟶ `INI_AUDI2`; `u16ObjId` ⟶ `ID` and `Q_NumericValue_AUDI`
- `INIT` ⟶ `ID.INIT`; `ID.INITO` ⟶ `INI_AUDI2.INIT`; `INI_AUDI2.INITO` ⟶ `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`)
- `QI` ⟶ `Q_NumericValue_AUDI.QI`, `ID.QI` and `INI_AUDI2.QI`
- `Q_NumericValue_AUDI.QO` ⟶ `AND_QO.IN1`; `ID.QO` ⟶ `AND_QO.IN2`; `INI_AUDI2.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`

Technical Details

- **`SETM = TRUE`:** the boot value is loaded automatically anyway (`INIT` ⟶ `INITO` ⟶ `GET` ⟶ `GETO`, independent of `SETM`). `SETM = TRUE` ensures that a live change during operation is also confirmed/reported back immediately; important if the fresh value is to be used further in the same run (here via `VALUEO`).
- `INI_AUDI2` has no output plug of its own, so the output is duplicated via `AUDI_SPLIT_2`
- Plain integer (`UDINT`) without scaling

Application Scenarios

- A persistently stored integer setting (e.g. counter, selection index) that is entered on the VT and reused

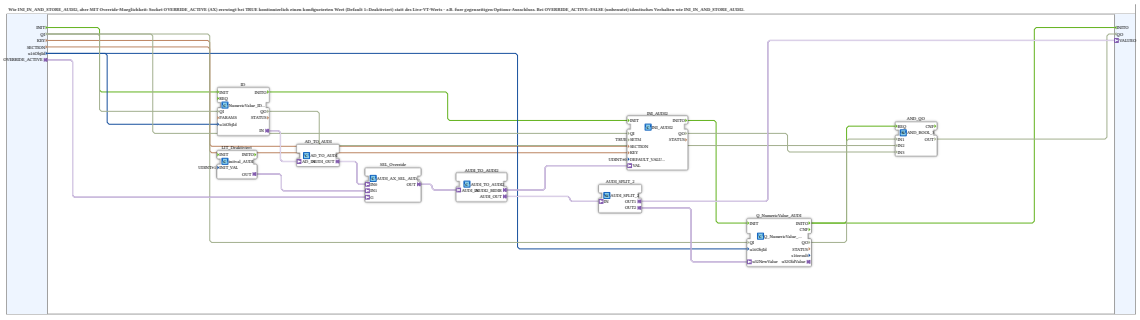
Summary

`INI_IN_AND_STORE_AUDI2` stores a VT integer value in an INI file (via `AUDI2`) and additionally outputs it as `AUDI`.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_IN_AND_STORE_AUDI2_OVERRIDE



• — ○ • — ○ • — ○ •

Introduction

`INI_IN_AND_STORE_AUDI2_OVERRIDE` is like `INI_IN_AND_STORE_AUDI2`, but with an override option: while the `OVERRIDE_ACTIVE` socket (AX) is continuously TRUE, the value 1 (disabled) is forced instead of the live VT value - e.g. for mutual option exclusion. With `OVERRIDE_ACTIVE`=FALSE (unused) the behavior is identical to `INI_IN_AND_STORE_AUDI2`. The variant without `OVERRIDE_ACTIVE` is available separately as `INI_IN_AND_STORE_AUDI2` (rename in version 1.2).

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).



Interface

Event inputs

- `INIT` (`EInit`): initialization request.

Event outputs

- `INITO` (`EInit`): initialization confirm.

Input variables

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE enables the service.
- `KEY` (STRING): key name in the INI file.
- `SECTION` (STRING): section name in the INI file.
- `u16ObjId` (UINT, default `ID_NULL`): ObjectID of the VT input field.

Output variables

- `QO` (BOOL): Quality Output - TRUE if the service was executed successfully (AND combination `AND_QO` of the `QO` outputs of the inner blocks).

Adapters

- **Plug `VALUEO`** (`adapter::types::unidirectional::AUDI`): value for a further consumer.
- **Socket `OVERRIDE_ACTIVE`** (`adapter::types::unidirectional::AX`): when continuously TRUE, VAL=1 (disabled) is forced.

The events `INIT`/`INITO` are passed as a chain through the inner blocks (see How it works).

How it works

1. `ID` (`NumericValue_IDA`) reads the VT input field; `AD_TO_AUDI` converts to AUDI.
2. `SEL_Override` (`AUDI_AX_SEL_AUDI`) selects: `IN0` = live VT value, `IN1` = override value from `LIT_Deaktiviert` (`initval_AUDI`, `INIT_VAL=UDINT#1`), selected via `G` = `OVERRIDE_ACTIVE`.
3. `AUDI_TO_AUDI2` converts into the bidirectional AUDI2 adapter for `INI_AUDI2` (`eclipse4diac::storage::INI_AUDI2`; `SETM`=TRUE, `DEFAULT_VALUE=UDINT#0`).
4. `AUDI_TO_AUDI2.AUDI_OUT` is duplicated via `AUDI_SPLIT_2`: `OUT1` to `VALUEO`, `OUT2` to `Q_NumericValue_AUDI` (`u32NewValue`, VT display).

Initialization: `INIT` → `ID.INIT`; `ID.INITO` → `INI_AUDI2.INIT`; `INI_AUDI2.INITO` → `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` → `INITO` and → `AND_QO.REQ`. Quality: `QI` → `Q_NumericValue_AUDI.QI`, `ID.QI`, `INI_AUDI2.QI`; `Q_NumericValue_AUDI.QO`, `ID.QO`, `INI_AUDI2.QO` → `AND_QO.IN1` to `AND_QO.IN3` (in this order); `AND_QO.OUT` → `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_3`).

Technical notes

- The override runs over the same `INI_AUDI2`/`Q_NumericValue_AUDI` path: according to its own documentation, `Q_NumericValue` allows only one instance per ObjectID, so a second writing instance is not possible.
- For pure enum/integer values (e.g. dropdowns 0-3), no decimal places.
- `SETM`=TRUE: the boot value loads automatically (INIT → INITO → GET → GETO); `SETM`=TRUE additionally ensures that a live change is confirmed/echoed immediately.

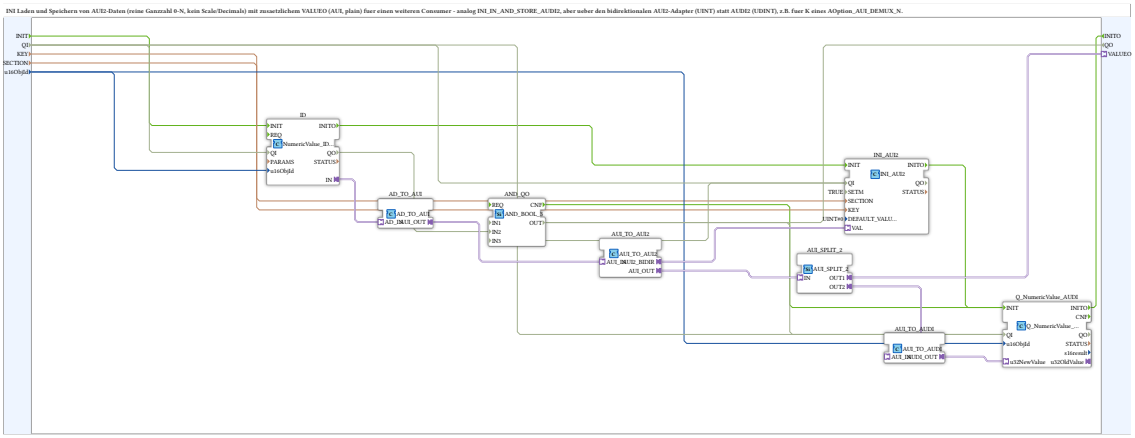
Summary

AUDI2 storage block with a switchable override to 1 (disabled), e.g. for mutual option exclusion.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_IN_AND_STORE_AUI2

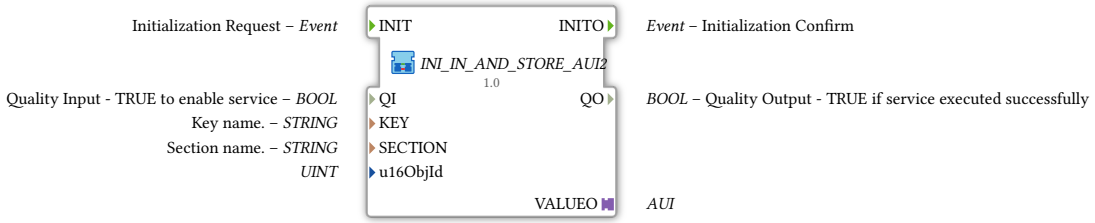


• – ○ • – ○ • – ○ •

Introduction

`INI_IN_AND_STORE_AUI2` loads and stores AUI2 data (pure integer 0-N, no scale/decimals) in an INI file and additionally provides `VALUEO` (AUI, plain) for a further consumer. It corresponds to `INI_IN_AND_STORE_AUDI2`, but uses the bidirectional AUI2 adapter (UINT) instead of AUDI2 (UDINT) - e.g. for `K` of an `AOption_AUI_DEMUX_N`.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).



Interface

Input variables

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE enables the service.
- `KEY` (STRING): key name in the INI file.

- `SECTION` (STRING): section name in the INI file.
- `u16ObjId` (UINT, default `ID_NULL`): ObjectID of the VT input field.

Output variables

- `QO` (BOOL): Quality Output - TRUE if the service was executed successfully (AND of `Q_NumericValue_AUDI.QO`, `ID.QO` and `INI_AUI2.QO`).

Adapters

- **Plug `VALUEO`** (`adapter::types::unidirectional::AUI`): value for a further consumer.

Events

- Input `INIT` (`EInit`): Initialization Request
- Output `INITO` (`EInit`): Initialization Confirm

How it works

1. `ID` (`NumericValue_IDA`) reads the VT input field; `AD_TO_AUI` converts to AUI.
2. `AUI_TO_AUI2` converts into the bidirectional AUI2 adapter for `INI_AUI2` (`eclipse4diac::storage::INI_AUI2`; `SETM`=TRUE, `DEFAULT_VALUE=UINT#0`).
3. `AUI_TO_AUI2.AUI_OUT` is duplicated via `AUI_SPLIT_2`: `OUT1` to `VALUEO`, `OUT2` via `AUI_TO_AUDI` to `Q_NumericValue_AUDI` (`u32NewValue`, VT display).

`QI` is passed through to `ID.QI`, `INI_AUI2.QI` and `Q_NumericValue_AUDI.QI`. The initialization chain is `INIT` ⟶ `ID.INIT`, `ID.INITO` ⟶ `INI_AUI2.INIT`, `INI_AUI2.INITO` ⟶ `Q_NumericValue_AUDI.INIT`, `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (and additionally ⟶ `AND_QO.REQ`). `QO` is the AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_3`) of `Q_NumericValue_AUDI.QO`, `ID.QO` and `INI_AUI2.QO`.

Technical notes

- Structurally identical to `INI_IN_AND_STORE_AUDI2`, only with AUI2 instead of AUDI2.
- `SETM`=TRUE: the boot value loads automatically (INIT ⟶ INITO ⟶ GET ⟶ GETO); `SETM`=TRUE additionally ensures that a live change is confirmed/echoed immediately (e.g. for further use via `VALUEO`).

Summary

AUI2 variant of the INI storage family for pure integers with an additional `VALUEO` output.

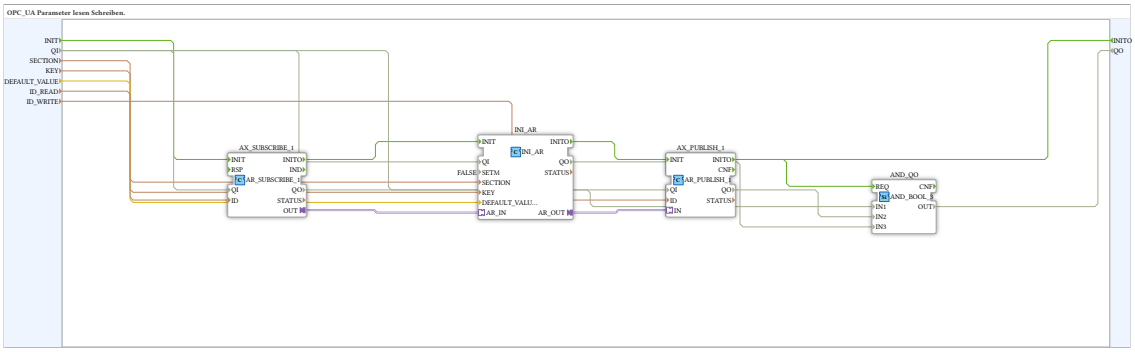
🛠️ Related Exercises

- Uebung_094b4_AX

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_OPC_PARAM

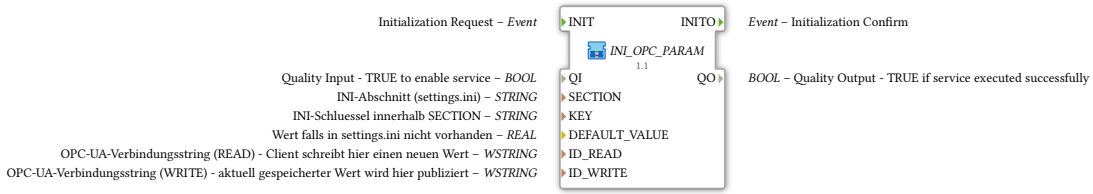


• – ○ • – ○ • – ○ •

Introduction

`INI_OPC_PARAM` reads and writes a REAL parameter via OPC UA and stores it persistently in the INI file (`settings.ini`). A client can set a new value using `ID_READ`; the currently stored value is published using

ID_WRITE. After a restart, the last stored value (or DEFAULT_VALUE, if none exists) is automatically loaded and published again.



Function Blocks (FBs) Used

Sub-Blocks: INI_OPC_PARAM

- **Type**: SubAppType
- **Internal FBs Used**:
- **AX_SUBSCRIBE_1**: `adapter::net::AR_SUBSCRIBE_1` — subscribes to new values from the OPC UA client (`ID_READ`).
- **INI_AR**: `eclipse4diac::storage::INI_AR` (`SETM=FALSE`) — persistently stores the value in the INI file under `SECTION`/`KEY`, with `DEFAULT_VALUE` as the initial value.
- **AX_PUBLISH_1**: `adapter::net::AR_PUBLISH_1` — publishes the current value via OPC UA (`ID_WRITE`).
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — combines the `QO` outputs of `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` and `INI_AR` into the SubApp output `QO`.
- **Interface**: event input `INIT`, event output `INITO`, data input `QI` (initial value `TRUE`, passed on to all three FBs) and data output `QO` (`AND` of the three `QO` outputs).
- **How it works**: A value received via OPC UA is forwarded to `INI_AR`, stored there, and the value (newly received or loaded at startup) is then published again via `AX_PUBLISH_1`.

Program Flow and Connections

1. **Initialization Chain**: `INIT` → `AX_SUBSCRIBE_1.INIT`; `AX_SUBSCRIBE_1.INITO` → `INI_AR.INIT`; `INI_AR.INITO` → `AX_PUBLISH_1.INIT`; `AX_PUBLISH_1.INITO` → `INITO` and → `AND_QO.REQ`.
2. **Quality**: `QI` → `AX_SUBSCRIBE_1.QI`, `AX_PUBLISH_1.QI` and `INI_AR.QI`; `AX_SUBSCRIBE_1.QO` → `AND_QO.IN1`, `AX_PUBLISH_1.QO` → `AND_QO.IN2`, `INI_AR.QO` → `AND_QO.IN3`; `AND_QO.OUT` → `QO`.
3. **Parameters**: `SECTION` → `INI_AR.SECTION`; `KEY` → `INI_AR.KEY`; `DEFAULT_VALUE` → `INI_AR.DEFAULT_VALUE`; `ID_READ` → `AX_SUBSCRIBE_1.ID`; `ID_WRITE` → `AX_PUBLISH_1.ID`.
4. **Adapter Chain**: `AX_SUBSCRIBE_1.OUT` → `INI_AR.AR_IN` (new value from the client) → `INI_AR.AR_OUT` → `AX_PUBLISH_1.IN` (saved/current value back to the client).

Technical Features

- **`SETM=FALSE`**: `INI_AR` only saves on an explicit write event via `AR_IN`, not on every cycle.
- **Sequential Initialization**: `AX_SUBSCRIBE_1.INITO` → `INI_AR.INIT` → `INI_AR.INITO` → `AX_PUBLISH_1.INIT` ensures that the stored value is loaded before being published.
- **No Dedicated Adapter Output**: Unlike `INI_OPC_PARAM_AR`, this block does not provide the stored value as a plug for other blocks in the calling network—it serves exclusively for OPC UA connectivity.

Application Scenarios

- A numerical parameter (e.g., setpoint) editable via OPC UA/Web client that must be retained across restarts but is not used further on the local network.

Comparison with Similar Function Blocks

`INI_OPC_PARAM_AR` extends this function block with an additional `AR` plug (`OUT`) for local use of the stored value. `INI_OPC_PARAM_ATM` additionally provides a `TIME` conversion, and `INI_OPC_PARAM_AX` is the BOOL variant with a `AX` output.

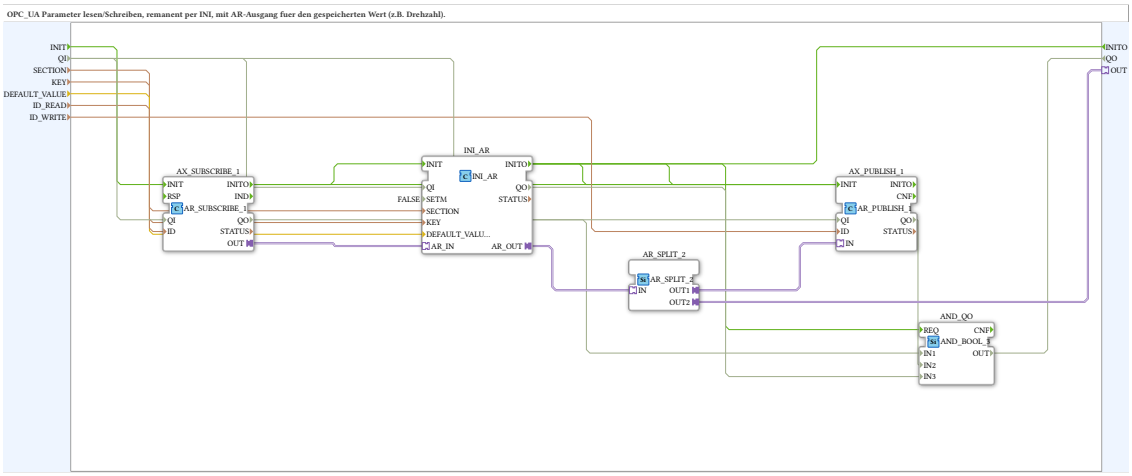
Summary

`INI_OPC_PARAM` combines OPC UA read/write access with persistent INI storage of a REAL parameter without making the value available locally as an adapter.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

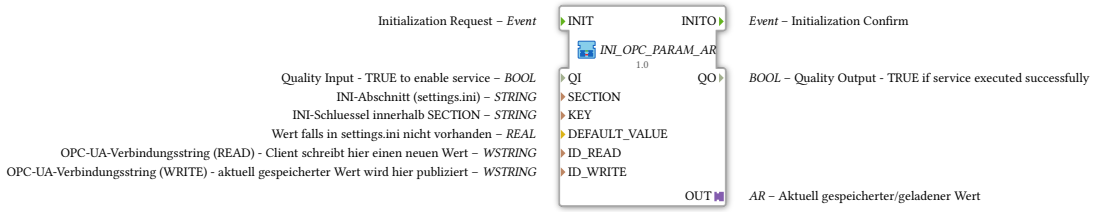
INI_OPC_PARAM_AR



• — ○ • — ○ • — ○ •

Introduction

`INI_OPC_PARAM_AR` extends `INI_OPC_PARAM` with an additional `AR` adapter output (`OUT`), which provides the currently stored/loaded value directly for further use in the calling network—e.g., for a locally adjustable rotational speed that should be editable via OPC UA and also used directly in the same application.



Function Blocks (FBs) Used

Sub-Blocks: INI_OPC_PARAM_AR

- **Type**: SubAppType
- **Internal FBs Used**:
- **AX_SUBSCRIBE_1**: `adapter::net::AR_SUBSCRIBE_1` (`QI` comes from the input `QI`) — subscribes to new values from the OPC UA client (`ID_READ`).
- **INI_AR**: `eclipse4diac::storage::INI_AR` (`QI` comes from the input `QI`, `SETM=FALSE`) — persistently stores the value in the INI file under `SECTION/KEY`, with `DEFAULT_VALUE` as the initial value.
- **AX_PUBLISH_1**: `adapter::net::AR_PUBLISH_1` (`QI` comes from the input `QI`) — publishes the current value via OPC UA (`ID_WRITE`).
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — combines the `QO` outputs of the three FBs into the output `QO`.
- **Functionality**: Identical to `INI_OPC_PARAM`, except that `INI_AR.AR_OUT` is directly routed to the plugin `OUT`. The SubApp interface additionally has the events `INIT`/`INITO` and the data `QI` (BOOL, initial value `TRUE`) and `QO` (BOOL).

Program Flow and Connections

1. **Initialization Chain**: `INIT` → `AX_SUBSCRIBE_1.INIT`; `AX_SUBSCRIBE_1.INITO` → `INI_AR.INIT` and → `AX_PUBLISH_1.INIT`; `AX_PUBLISH_1.INITO` → `INI_AR.INIT`; `INI_AR.INITO` → `AX_PUBLISH_1.INIT`, → `INITO` and → `AND_QO.REQ`.
2. **Parameters**: `SECTION` → `INI_AR.SECTION`; `KEY` → `INI_AR.KEY`; `DEFAULT_VALUE` → `INI_AR.DEFAULT_VALUE`; `ID_READ` → `AX_SUBSCRIBE_1.ID`; `ID_WRITE` → `AX_PUBLISH_1.ID`.
3. **Adapter Chain**: `AX_SUBSCRIBE_1.OUT` → `INI_AR.AR_IN` → `INI_AR.AR_OUT` → `AX_PUBLISH_1.IN` **and** → `OUT` (plug into the SubApp interface).
4. **Quality**: `QI` → `AX_SUBSCRIBE_1.QI`, `AX_PUBLISH_1.QI` and `INI_AR.QI`; `AX_SUBSCRIBE_1.QO` → `AND_QO.IN1`, `AX_PUBLISH_1.QO` → `AND_QO.IN2`, `INI_AR.QO` → `AND_QO.IN3`; `AND_QO.OUT` → `QO`.

Technical Features

- **One output, two destinations**: `INI_AR.AR_OUT` simultaneously feeds the OPC UA publish adapter and the external `OUT` plug—both receive the same value without any additional splitting, as these are two separate adapter connections from the same source.
- Otherwise identical to `INI_OPC_PARAM` (see there for details on `SETM`/initialization sequence).

Application Scenarios

- A REAL parameter (e.g., speed setpoint) that can be edited via OPC UA/web client and is stored persistently, and which must also be directly passed to other components in the same network.

Comparison with Similar Building Blocks

Compared to `INI_OPC_PARAM`, the only difference is the additional `AR` plugin, `OUT`. `INI_OPC_PARAM_ATM` additionally converts the value to `TIME`, and `INI_OPC_PARAM_AX` is the BOOL variant.

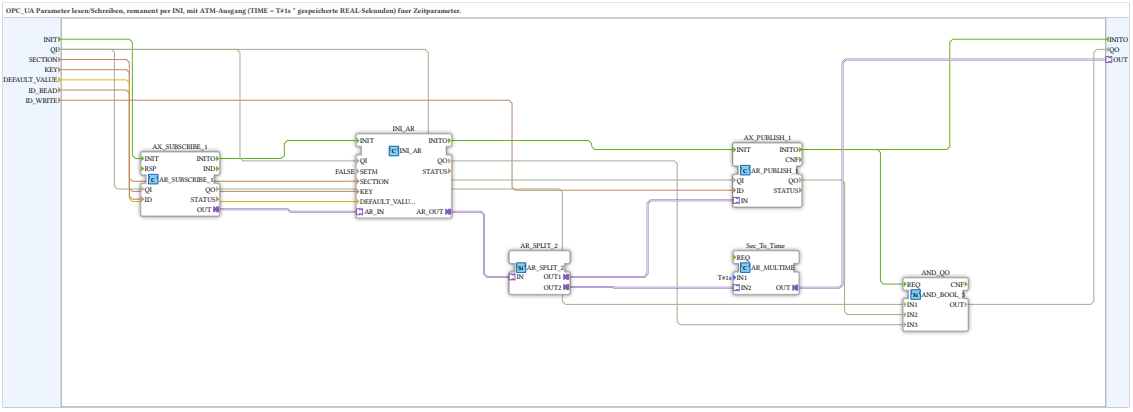
Summary

`INI_OPC_PARAM_AR` is `INI_OPC_PARAM` with an additional `AR` output for local reuse of the persistently stored parameter.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

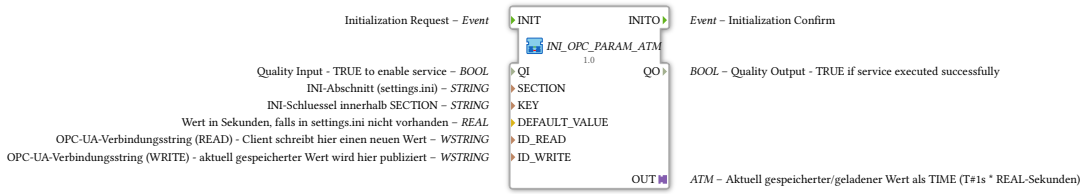
INI_OPC_PARAM_ATM



• – ○ • – ○ • – ○ •

Introduction

`INI_OPC_PARAM_ATM` is the time parameter variant of `INI_OPC_PARAM_AR`: The persistently stored REAL value is additionally converted into a `ATM` adapter (`TIME`) — `DEFAULT_VALUE` and the stored value are interpreted as seconds (`T#1s * Sekunden`).



Function Blocks (FBs) Used

Sub-Blocks: INI_OPC_PARAM_ATM

- **Type**: SubAppType
- **Internal FBs Used**:
- **AX_SUBSCRIBE_1**: `adapter::net::AR_SUBSCRIBE_1` (`QI` from outside) — subscribes to new values from the OPC UA client (`ID_READ`).
- **INI_AR**: `eclipse4diac::storage::INI_AR` (`SETM=FALSE`, `QI` from outside) — persistently stores the value in the INI file under `SECTION`/`KEY`, with `DEFAULT_VALUE` (in seconds) as the starting value.
- **AX_PUBLISH_1**: `adapter::net::AR_PUBLISH_1` (`QI` from outside) — publishes the current REAL value via OPC UA (`ID_WRITE`).
- **Sec_To_Time**: `adapter::iec61131::arithmetic::AR_MULTIME` (`IN1=T#1s`) — multiplies the stored REAL value (seconds) by `T#1s` to provide it as the value `TIME`.
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — combines the `QO` of `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` and `INI_AR` into the output `QO`.
- **How it works**: Like `INI_OPC_PARAM_AR`, except that `INI_AR.AR_OUT` is converted via `Sec_To_Time` (`AR_MULTIME`) to `TIME` and provided as the `ATM` plug `OUT`.

Program Flow and Connections

1. **Initialization Chain**: event input `INIT` ⟶ `AX_SUBSCRIBE_1.INIT`; `AX_SUBSCRIBE_1.INITO` ⟶ `INI_AR.INIT`; `INI_AR.INITO` ⟶ `AX_PUBLISH_1.INIT`; `AX_PUBLISH_1.INITO` ⟶ event output `INITO` and ⟶ `AND_QO.REQ`.
2. **Parameters**: `SECTION` ⟶ `INI_AR.SECTION`; `KEY` ⟶ `INI_AR.KEY`; `DEFAULT_VALUE` (seconds) ⟶ `INI_AR.DEFAULT_VALUE`; `ID_READ` ⟶ `AX_SUBSCRIBE_1.ID`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`.
3. **Adapter Chain**: `AX_SUBSCRIBE_1.OUT` ⟶ `INI_AR.AR_IN` ⟶ `INI_AR.AR_OUT` ⟶ `AX_PUBLISH_1.IN` (REAL, published unchanged via OPC UA) **and** ⟶ `Sec_To_Time.IN2` ⟶ `Sec_To_Time.OUT` ⟶ `OUT` (as `TIME`).
4. **Quality**: data input `QI` (`BOOL`, default `TRUE`) ⟶ `AX_SUBSCRIBE_1.QI`, `AX_PUBLISH_1.QI` and `INI_AR.QI`; their `QO` ⟶ `AND_QO.IN1`, `AND_QO.IN2`, `AND_QO.IN3`; `AND_QO.OUT` ⟶ data output `QO`.

Technical Features

- **`AR_MULTIME` with fixed `IN1=T#1s`**: The multiplication `T#1s * REAL-Sekunden` directly converts the stored numerical value into a `TIME` value, without requiring any conversion by the caller.
- **OPC UA Publish Remains REAL**: `ID_WRITE` continues to publish the original REAL value (seconds) — only the additional `OUT` plug delivers `TIME`.
- Otherwise identical to `INI_OPC_PARAM_AR`.

Application Scenarios

- Retentive time parameters (e.g., delay or timeout times in seconds) that should be editable as a simple number via OPC UA, but are used internally as a `TIME` value (e.g., for a `E_DELAY` or `timeOut` adapter).

Comparison with Similar Function Blocks

Compared to `INI_OPC_PARAM_AR`, only the additional `Sec_To_Time` conversion (`AR_MULTIME`) is added; the `OUT` plug is of type `ATM` instead of `AR`. `INI_OPC_PARAM_AX` is the BOOL variant.

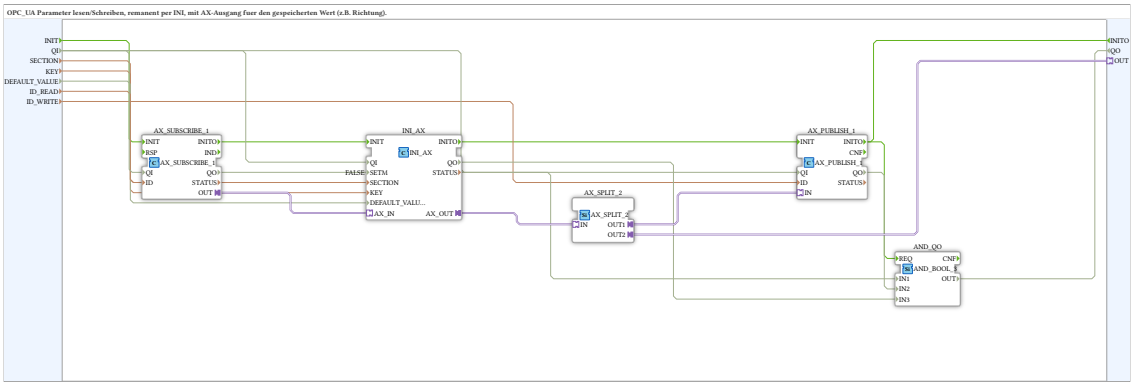
Summary

`INI_OPC_PARAM_ATM` provides a persistently stored, OPC UA editable time parameter as a ready-made `TIME` adapter, calculated from a real value stored in seconds.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

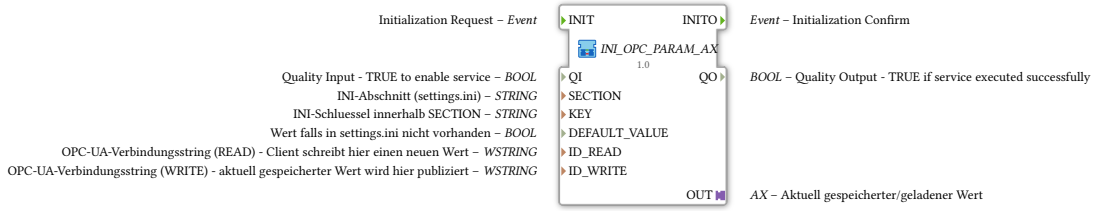
INI_OPC_PARAM_AX



• – ○ • – ○ • – ○ •

Introduction

`INI_OPC_PARAM_AX` is the BOOL variant of `INI_OPC_PARAM_AR`: Instead of a REAL value, a BOOL parameter (e.g., a direction or operating mode switch) is read/written via OPC UA, permanently stored in the INI file, and made available as the `AX` adapter for local use.



Function Blocks (FBs) Used

Sub-Blocks: INI_OPC_PARAM_AX

- **Type**: SubAppType
- **Internal FBs Used**:
- **AX_SUBSCRIBE_1**: `adapter::net::AX_SUBSCRIBE_1` — subscribes to new BOOL values from the OPC UA client (`ID_READ`).
- **INI_AX**: `eclipse4diac::storage::INI_AX` (`SETM=FALSE`) — persistently stores the BOOL value in the INI file under `SECTION`/`KEY`, with `DEFAULT_VALUE` as the initial value.
- **AX_PUBLISH_1**: `adapter::net::AX_PUBLISH_1` — publishes the current value via OPC UA (`ID_WRITE`).
- **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — branches the output of `INI_AX` to `AX_PUBLISH_1` and `OUT`.
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — combines the `QO` outputs of `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` and `INI_AX` into the `QO` of the SubApp.
- **Functionality**: Structurally identical to `INI_OPC_PARAM_AR`, but with `AX` adapters (`AX_SUBSCRIBE_1`/`INI_AX`/`AX_PUBLISH_1`) instead of `AR` adapters, matching the BOOL data type of `DEFAULT_VALUE`.

Program Flow and Connections

1. **Initialization Chain**: `INIT` ⟶ `AX_SUBSCRIBE_1.INIT`; `AX_SUBSCRIBE_1.INITO` ⟶ `INI_AX.INIT`; `INI_AX.INITO` ⟶ `AX_PUBLISH_1.INIT`; `AX_PUBLISH_1.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`).
2. **Parameters**: `SECTION` ⟶ `INI_AX.SECTION`; `KEY` ⟶ `INI_AX.KEY`; `DEFAULT_VALUE` (BOOL) ⟶ `INI_AX.DEFAULT_VALUE`; `ID_READ` ⟶ `AX_SUBSCRIBE_1.ID`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`.
3. **Quality**: `QI` ⟶ `.QI` of `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` and `INI_AX` (no longer hard-wired `QI=TRUE`); `AX_SUBSCRIBE_1.QO` ⟶ `AND_QO.IN1`, `AX_PUBLISH_1.QO` ⟶ `AND_QO.IN2`, `INI_AX.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.
4. **Adapter Chain**: `AX_SUBSCRIBE_1.OUT` ⟶ `INI_AX.AX_IN` ⟶ `INI_AX.AX_OUT` ⟶ `AX_SPLIT_2.IN`; `AX_SPLIT_2.OUT1` ⟶ `AX_PUBLISH_1.IN` **and** `AX_SPLIT_2.OUT2` ⟶ `OUT` (plug into the SubApp interface).

Technical Features

- **BOOL instead of REAL**: The only functional difference to `INI_OPC_PARAM_AR` is the data type (`DEFAULT_VALUE` is `BOOL` here) and the corresponding `AX` adapter types instead of `AR`.
- Otherwise identical to `INI_OPC_PARAM_AR` (initialization sequence, `SETM=FALSE`, one output with two destinations).

Application Scenarios

- Retentively stored BOOL parameters (e.g., direction of rotation, operating mode switch) that should be editable via OPC UA/Web client and additionally reused locally as a `AX` adapter.

Comparison with Similar Function Blocks

`INI_OPC_PARAM_AX` is to `INI_OPC_PARAM_AR` as `AX` is to `AR`: same structure, different data type. `INI_OPC_PARAM` and `INI_OPC_PARAM_ATM` are the REAL-based variants (without and with additional `TIME` conversion, respectively).

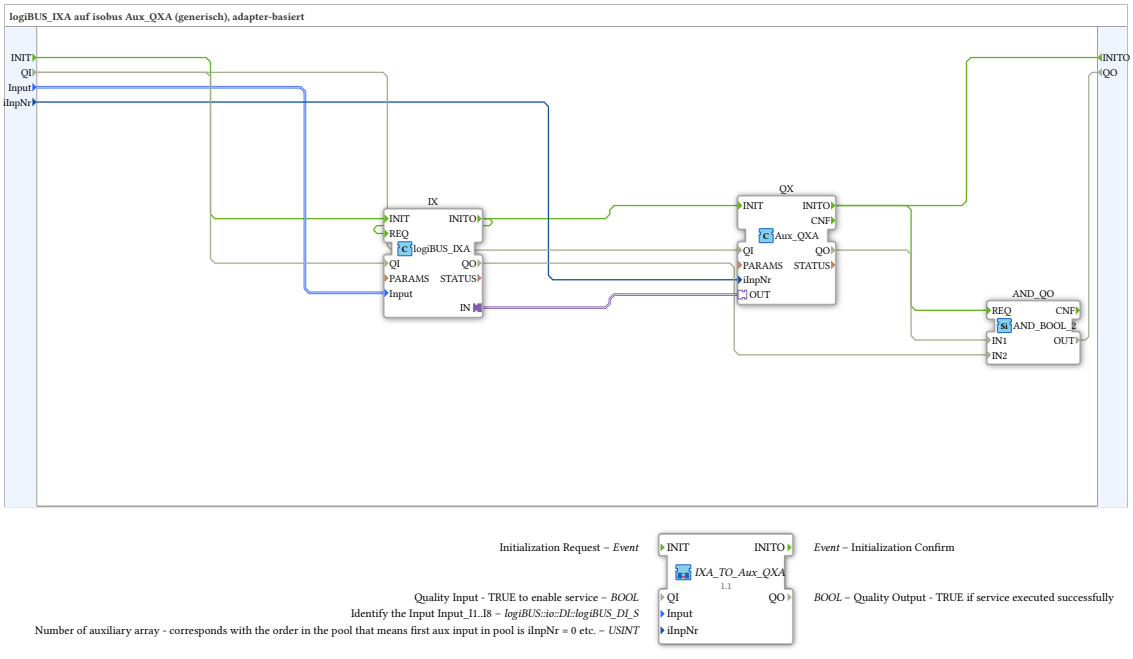
Summary

`INI_OPC_PARAM_AX` is the BOOL variant of the `INI_OPC_PARAM` family: OPC UA read/write access, persistent INI storage, and a local `AX` output for a Boolean parameter.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

IXA_TO_Aux_QXA



Introduction

The **IXA_TO_Aux_QXA** subapplication serves as a generic, adapter-based bridge between a logiBUS digital input channel and an ISOBUS auxiliary output (Aux_QXA). It is designed for reuse in scenarios where a physical input from a logiBUS I/O system must be mapped to a specific auxiliary output position on an ISOBUS network, commonly found in tractor-implement control systems.

By encapsulating the internal conversion logic (from the logiBUS domain to the ISOBUS domain), the subapp provides a clean, reusable interface with only two data inputs, hiding all protocol-specific details from the surrounding logic.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; passed on to `IX.INIT`. The data path itself is purely continuous (not event-triggered).

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `QX.INITO`.

Data Inputs

Name	Type	Initial Value	Description
`Input`	`logiBUS::io::DI::logiBUS_DI_S`	`Invalid`	Identifies the physical logiBUS digital input (e.g., `Input_I1` .. `Input_I8`) to be read.
`QI`	`BOOL`	`TRUE`	Quality input - TRUE enables the service.
`iInpNr`	`USINT`	`0`	Number of the auxiliary array entry; corresponds to the position in the pool. The first aux input in the pool is `iInpNr = 0`, the second is `1`, etc.

Data Outputs

- **QO** (`BOOL`): quality output - AND combination (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_2`) of `QX.QO` and `IX.QO`. The signal itself is propagated internally via adapter connections.

Adapters

The subapp does not expose adapters at its interface. Internally, an adapter connection links the logiBUS input block to the ISOBUS auxiliary output block.

Functionality

The subapp performs the following tasks:

1. Receives a digital input identifier (`Input`) of type `logiBUS_DI_S` and an auxiliary index (`iInpNr`).
2. Internally instantiates a `logiBUS_IXA` function block, which reads and processes the specified logiBUS digital input. This block is enabled via the subapp input `QI` (default `TRUE`), and its `PARAMS` parameter is deliberately left empty (visible attribute hidden).
3. The processed input signal is forwarded through an adapter connection (`IX.IN` ⟶ `QX.OUT`) to an instantiated `Aux_QXA` function block from the ISOBUS utility library.
4. The `Aux_QXA` block maps the incoming adapter data to the auxiliary output position specified by `iInpNr`, thereby activating the corresponding ISOBUS auxiliary function.

Initialization runs as a chain: `INIT` ⟶ `IX.INIT`; `IX.INITO` ⟶ `QX.INIT`; `QX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`. Quality: `QI` ⟶ `IX.QI` and `QX.QI`; `QX.QO` ⟶ `AND_QO.IN1`, `IX.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`. With `QI = TRUE` (default), the entire chain is active as soon as valid input data is available.

Technical Features

- **Adapter-based coupling**: The connection between the logiBUS input handling and the ISOBUS output handling is realized via a typed adapter connection, ensuring clean, protocol-compliant data transfer.

- **Generic input selection**: The `Input` parameter allows selecting any of the physical inputs `I1`..`I8`, making the subapp reusable for different wiring configurations without modification.
- **Pool-position mapping**: The `iInpNr` parameter maps the selected input to a dedicated position in the ISOBUS auxiliary output pool.
- **Hidden internal parameters**: The `PARAMS` parameter of the `logiBUS_IXA` block is hidden in the graphical representation (`Visible = false`), reducing clutter in the network editor.
- **Default-enabled operation**: Both internal function blocks receive `QI` from the subapp input `QI` (default `TRUE`), so they are enabled by default; there is no hard-wired `QI` parameter any more. Start-up sequencing is done via the `INIT`/`INITO` chain.
- **Standard-conformant types**: Uses types defined according to IEC 61499-2, with the overall subapp packaged for reuse in different projects.

State Overview

The subapp does not implement an explicit state machine. Its operational behavior is derived from the states of the internal function blocks:

- **Idle / invalid input**: If `Input` is set to `Invalid` (as per its initial value), the `logiBUS_IXA` block will not deliver valid data, and the auxiliary output remains inactive.
- **Active input**: When a valid input identifier (`Input_I1`..`Input_I8`) is applied, the `logiBUS_IXA` block processes the signal and forwards it via the adapter to `Aux_QXA`.
- **Enabled output**: With `QI = TRUE` (default) on `Aux_QXA` and a valid adapter data stream, the auxiliary output position (`iInpNr`) is continuously updated to reflect the state of the selected logiBUS input.

Users should ensure that the `Input` value matches an actual, configured logiBUS input; otherwise, the output remains inactive.

Application Scenarios

- **Tractor implement control**: Mapping physical push-buttons or switches on an implement (connected via logiBUS) to ISOBUS auxiliary outputs for functions such as raising/lowering implements, activating PTOs, or controlling hydraulic valves.
- **Reusable input-to-output bridge**: Since the subapp is generic, it can be instantiated multiple times in a project — once for each physical input that shall control a distinct auxiliary output.
- **Migration of legacy wiring**: When upgrading from direct-wired controls to ISOBUS-based control, this subapp allows reusing existing logiBUS input modules without changing the wiring.

Comparison with Similar Blocks

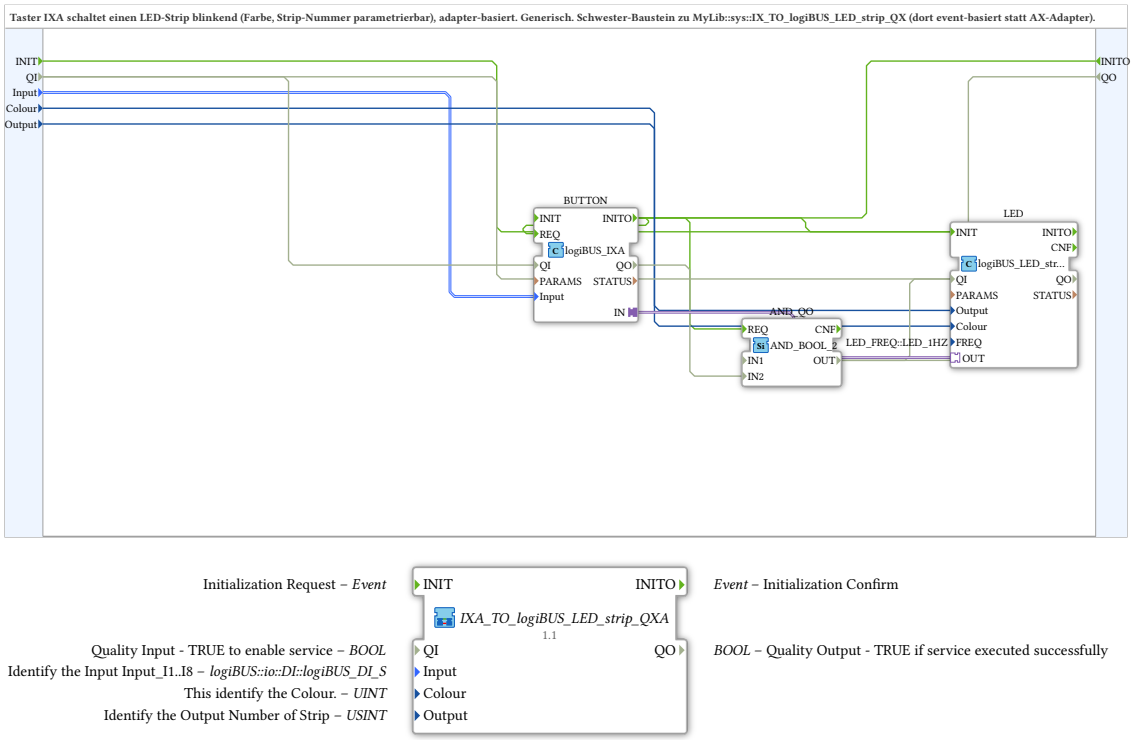
Compared to a direct hard-wired connection between a logiBUS input module and an ISOBUS auxiliary output, this subapp offers:

- **Protocol abstraction**: The user does not need to interact with raw adapter types or internal FB configurations.
- **Reusability**: Extracted from a larger exercise project (previously part of `Uebung_003c_sub_AX`), it can be dropped into any 4diac project with minimal effort.
- **Clarity**: The two-parameter interface (`Input`, `iInpNr`, plus the usual `QI`/`QO` and `INIT`/`INITO` ports) is far simpler than manually wiring internal FBs and checking adapter compatibility.
- **Limitation**: It provides a one-way signal path (input → output) without bidirectional feedback. Applications requiring status readback from the ISOBUS side would need a different, more elaborate subapp.

Conclusion

The **IXA_TO_Aux_QXA** subapplication is a compact, generic, and reusable solution for connecting logiBUS digital inputs to ISOBUS auxiliary outputs. By encapsulating the protocol adaptation in an adapter-based design, it simplifies project engineering, promotes reuse, and keeps the network representation clean. It is particularly suited for agricultural and off-highway applications built on the 4diac framework.

IXA_TO_logiBUS_LED_strip_QXA



• – ○ • – ○ • – ○ •

Introduction

The `IXA_TO_logiBUS_LED_strip_QXA` is a generic composite subapplication that integrates a logiBUS push button (via the `logiBUS_IXA` function block) with an LED strip controller (`logiBUS_LED_strip_QXA`). It provides a simple, adapter-based mechanism to control an LED strip's blinking state, color, and output line based on a digital input. The subapplication is designed to be reusable across different logiBUS configurations, allowing the mapping of any input to a specific LED strip with configurable parameters.

Interface Structure

This subapplication exposes the initialization events `INIT`/`INITO`, the status ports `QI`/`QO` and data inputs; it has no adapter interfaces at its boundary. All communication between internal blocks is handled internally.

Event Inputs

Name	Type	Comment
INIT	`EInit`	Initialization Request

Event Outputs

Name	Type	Comment
INITO	`EInit`	Initialization Confirm

Data Inputs

Name	Type	Initial Value	Comment
QI	`BOOL`	`TRUE`	Quality Input - TRUE enables the service.
Input	`logiBUS::io::DI::logiBUS_DI_S`	`Invalid`	Identify the Input Input_I1..I8

Colour	UINT	LED_COLOURS::LED_GREEN	This identify the Colour.
Output	USINT	(default 0)	Identify the Output Number of Strip

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if the service executed successfully (AND of LED.QO and BUTTON.QO).

Adapters

None (externally exposed).

Functionality

The subapplication routes its external data inputs to two internal function blocks:

- **BUTTON** (type logiBUS::io::DI::logiBUS_IXA): Receives the Input data; its QI comes from the subapplication's QI input.
- **LED** (type logiBUS::io::DO_LED::logiBUS_LED_strip_QXA): Receives Colour and Output data; FREQ=LED_FREQ::LED_1HZ is configured and QI comes from the subapplication's QI input.

Initialization runs as a chain INIT → LED.INIT, LED.INITO → BUTTON.INIT, BUTTON.INITO → INITO (and additionally → AND_QO.REQ). QI is forwarded to LED.QI and BUTTON.QI; LED.QO and BUTTON.QO are ANDed by the AND_QO block (type iec61131::booleanOperators::AND_BOOL_2) into QO.

An internal adapter connection links BUTTON.IN to LED.OUT, enabling the button to send state/event information directly to the LED controller via an adapter interface. This design decouples the event handling from the data path, allowing the LED strip to respond to button actions (e.g., toggling, blinking) while maintaining a simple parameterized configuration.

Technical Features

- **Adapter-based communication**: The internal connection via BUTTON.IN → LED.OUT uses an adapter, providing a flexible and type-safe link between the button and the LED controller.
- **Parameterization**: The Colour and Output data inputs allow runtime configuration of the LED color and the specific strip number.
- **Generic input mapping**: The Input data input accepts any logiBUS_DI_S value, making the subapplication adaptable to various button/input assignments.
- **Preconfigured internal blocks**: Both internal FBs receive QI from the subapplication's QI input (default TRUE), and the LED frequency is preset to 1 Hz, ensuring immediate operation upon deployment.

State Overview

The subapplication itself does not implement a dedicated state machine. Instead, it relies on the internal behavior of the logiBUS_IXA and logiBUS_LED_strip_QXA blocks, which manage their own operational states. The subapplication serves as a static wiring and parameterization layer, passing data and adapter signals to the appropriate endpoints.

Application Scenarios

- **Building automation**: Connect a physical button (e.g., a wall switch) to control a colored LED strip on a specific output line, with configurable color and blinking frequency.
- **Status indication**: Use a button to toggle the blinking of an LED strip to indicate system states (e.g., alarm, readiness) without requiring additional logic.
- **Generic I/O mapping**: Serve as a reusable component in larger logiBUS networks to map arbitrary inputs to LED outputs, reducing development effort for similar tasks.

Comparison with Similar Blocks

The subapplication is described as a "sister block" to MyLib::sys::IX_TO_logiBUS_LED_strip_QX. The primary difference is that the sister block uses an event-based interface, whereas this subapplication uses an adapter-based internal connection. This distinction affects integration patterns:

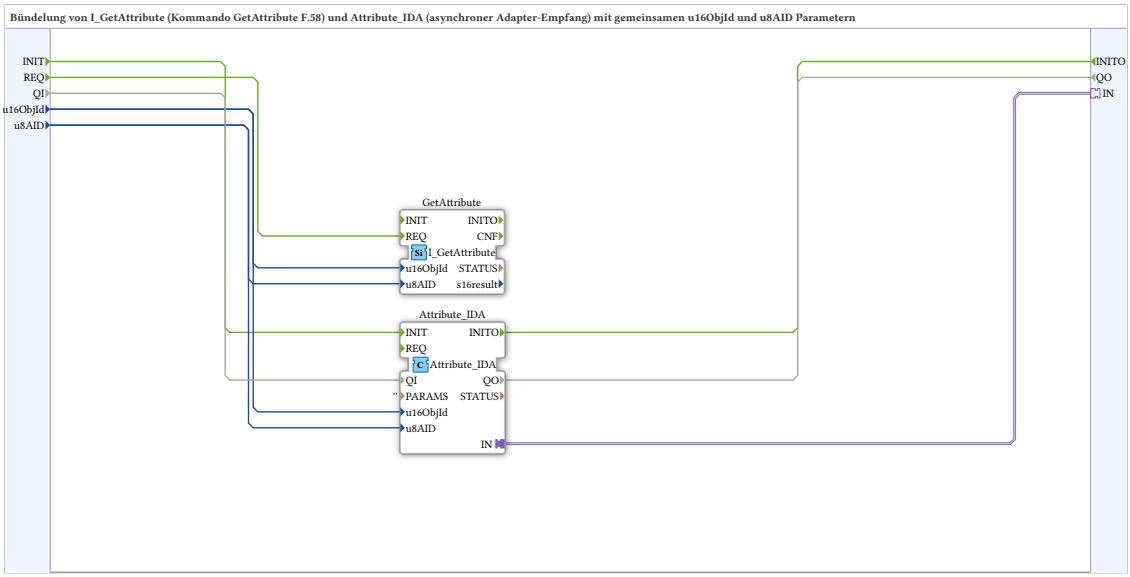
- **Adapter-based** (this block): More suitable for structured data exchange and when the button and LED controller need to share additional information beyond simple event triggers.
- **Event-based** (sister block): More lightweight for simple trigger scenarios, but may require additional event handling logic.

Both achieve similar functionality but cater to different architectural preferences.

Conclusion

`IXA_TO_logiBUS_LED_strip_QXA` provides a clean, parameterized solution for connecting a logiBUS input to a blinking LED strip. Its adapter-based design offers flexibility and easy integration into existing systems, while the explicit data inputs enable runtime customization of color and output line. As a generic, reusable component, it simplifies the development of building automation and indicator applications, making it a valuable asset in the MyLib::sys library.

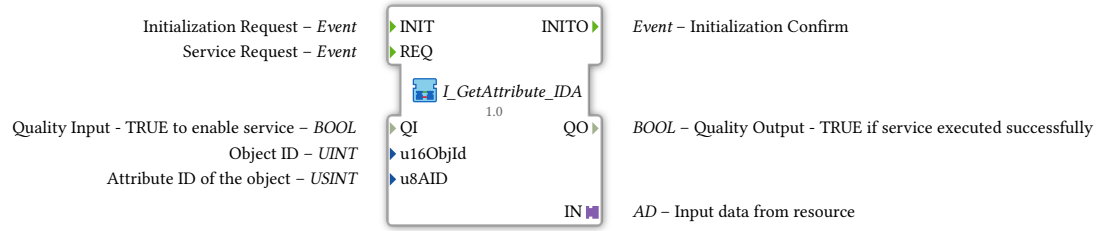
I_GetAttribute_IDA



• – ○ • – ○ • – ○ •

Introduction

`I_GetAttribute_IDA` bundles `I_GetAttribute` (command *GetAttribute*, F.58) and `Attribute_IDA` (asynchronous adapter reception) with the shared parameters `u16ObjId` and `u8AID`. With `REQ`, the attribute of a VT object is requested; the response arrives asynchronously via the adapter `IN`.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request (to `Attribute_IDA.INIT`)
- **REQ**: Service request (to `GetAttribute.REQ`)

Event Outputs

- **INITO** (`EInit`): initialization confirmation (from `Attribute_IDA.INITO`)

The response itself arrives via the adapter plug.

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, service enable (to `Attribute_IDA.QI`)
- **u16ObjId** (`UINT`, default `ID_NULL`): Object ID
- **u8AID** (`USINT`): Attribute ID of the object

Data Outputs

- **QO** (`BOOL`): quality output (from `Attribute_IDA.QO`)

Adapters

- **IN** (plug, `adapter::types::unidirectional::AD`): Input data (response) from the resource

Function Blocks (FBs) Used

- **GetAttribute**: `isobus::UT::I::I_GetAttribute` - sends the command *GetAttribute*
- **Attribute_IDA**: `isobus::UT::io::Attribute::Attribute_IDA` (`PARAMS=''`, `QI` comes from the SubApp interface) - receives the response asynchronously as an adapter

Program Flow and Connections

- `REQ` ⟶ `GetAttribute.REQ`
- `u16ObjId` ⟶ `GetAttribute.u16ObjId` and `Attribute_IDA.u16ObjId`
- `u8AID` ⟶ `GetAttribute.u8AID` and `Attribute_IDA.u8AID`
- `Attribute_IDA.IN` ⟶ `IN`
- `INIT` ⟶ `Attribute_IDA.INIT`; `Attribute_IDA.INITO` ⟶ `INITO`
- `QI` ⟶ `Attribute_IDA.QI`; `Attribute_IDA.QO` ⟶ `QO`

Technical Details

- The shared parameters ensure that request and reception refer to the same object and attribute.

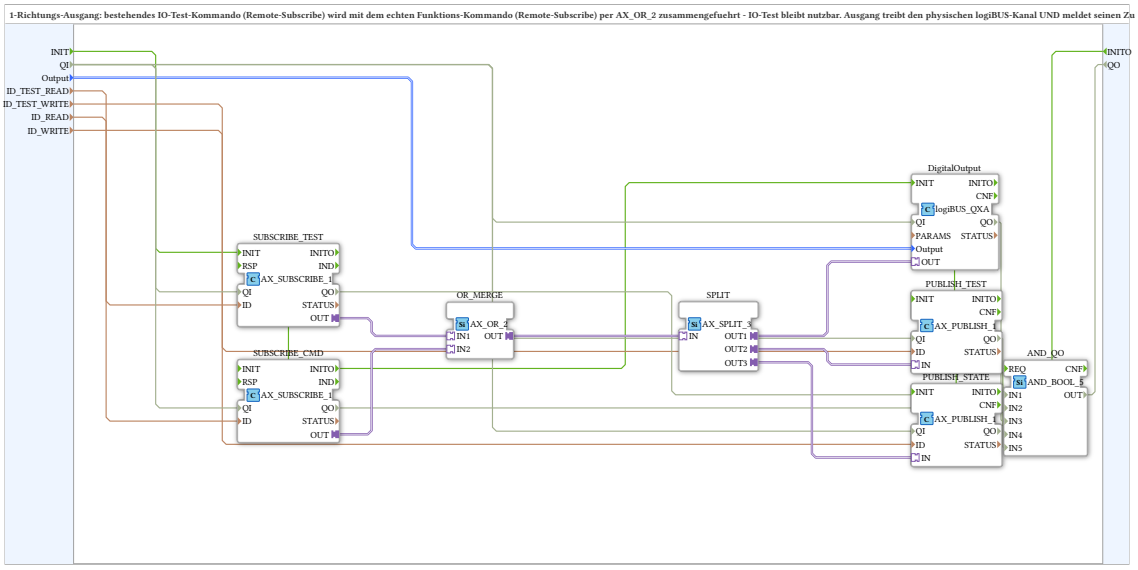
Application Scenarios

- Reading an attribute of a VT object (e.g. current value, color) and processing the response as an `AD` adapter value

Summary

`I_GetAttribute_IDA` combines the request and reception of a VT object attribute in one SubApp.

MERGE_SWITCH_1_QXA_OPC

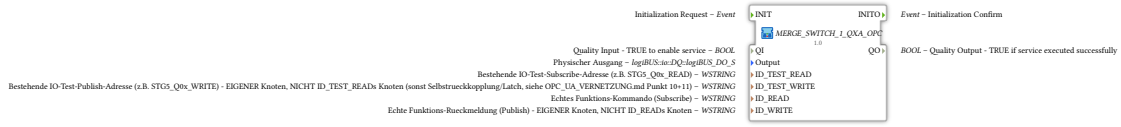


• – ○ • – ○ • – ○ •

Introduction

`MERGE_SWITCH_1_QXA_OPC` drives a single (non-double-acting) logiBUS output from two OPC UA sources: the existing I/O test command (remote subscribe) and the actual function command (also remote subscribe,

e.g., from a soft key/AUX via `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`). Both are combined using a level OR gate, so the existing I/O test remains usable while the actual function can simultaneously control the same output. For double-acting actuators (left/right, up/down), use `ILOCK_SWITCH_2_QXA_OPC` instead.



Function Blocks (FBs) Used

Sub-Blocks: MERGE_SWITCH_1_QXA_OPC

- **Type**: SubAppType
- **Internal FBs Used**:
- **SUBSCRIBE_TEST** / **SUBSCRIBE_CMD**: each `adapter::net::AX_SUBSCRIBE_1` (`QI` comes from the input `QI`) — subscribe to the existing I/O test command (`ID_TEST_READ`) or the actual function command (`ID_READ`), respectively.
- **OR_MERGE**: `adapter::booleanOperators::AX_OR_2` — combines both subscribe sources using a level OR operation.
- **SPLIT**: `adapter::events::unidirectional::AX_SPLIT_3` — distributes the combined signal to the physical output and two feedback channels.
- **DigitalOutput**: `logiBUS::io::DQ::logiBUS_QXA` (`QI` comes from the input `QI`) — physical digital output.
- **PUBLISH_TEST** / **PUBLISH_STATE**: each `adapter::net::AX_PUBLISH_1` (`QI` comes from the input `QI`) — report the resulting state to the existing I/O test monitoring (`ID_TEST_WRITE`) or to the actual function feedback (`ID_WRITE`, e.g., `GreenWhiteBackground` on the soft key).
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_5` — combines the `QO` outputs of the five service FBs into the output `QO`.
- **How it Works**: Two independent remote subscribe sources are combined via `AX_OR_2` and jointly drive the physical output as well as two separate feedback channels. The SubApp interface additionally has the events `INIT`/`INITO` and the data `QI` (BOOL, initial value `TRUE`) and `QO` (BOOL).

Program Flow and Connections

1. `ID_TEST_READ` ⟶ `SUBSCRIBE_TEST.ID`; `ID_READ` ⟶ `SUBSCRIBE_CMD.ID` (Data connections, hidden).
2. `SUBSCRIBE_TEST.OUT` ⟶ `OR_MERGE.IN1`; `SUBSCRIBE_CMD.OUT` ⟶ `OR_MERGE.IN2`.
3. `OR_MERGE.OUT` ⟶ `SPLIT.IN` ⟶ `SPLIT.OUT1` ⟶ `DigitalOutput.OUT` (physical output), `SPLIT.OUT2` ⟶ `PUBLISH_TEST.IN`, `SPLIT.OUT3` ⟶ `PUBLISH_STATE.IN`.
4. Parameters: `Output` ⟶ `DigitalOutput.Output`; `ID_TEST_WRITE` ⟶ `PUBLISH_TEST.ID`; `ID_WRITE` ⟶ `PUBLISH_STATE.ID`.
5. Initialization chain: `INIT` ⟶ `SUBSCRIBE_TEST.INIT` ⟶ `SUBSCRIBE_CMD.INIT` ⟶ `DigitalOutput.INIT` ⟶ `PUBLISH_TEST.INIT` ⟶ `PUBLISH_STATE.INIT` (each `INITO` ⟶ next `INIT`); `PUBLISH_STATE.INITO` ⟶ `INITO` and `AND_QO.REQ`.
6. Quality: `QI` ⟶ `SUBSCRIBE_TEST.QI`, `SUBSCRIBE_CMD.QI`, `DigitalOutput.QI`, `PUBLISH_TEST.QI` and `PUBLISH_STATE.QI`; their `QO` outputs ⟶ `AND_QO.IN1` to `AND_QO.IN5` (in this order); `AND_QO.OUT` ⟶ `QO`.

Technical Features

- **Two Separate Feedback Nodes**: The resulting state is reported back to two separate OPC UA nodes, independent of the respective command node (`ID_TEST_WRITE` for the existing I/O diagnostic display, `ID_WRITE` for the actual function feedback). Command and status nodes must never share the same OPC UA node—otherwise, the subscribe node will interpret its own publish echo as a new command, and the output will be stuck.
- **Level OR instead of Edge Logic**: Unlike latching button blocks (e.g., `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`), this block uses a simple, continuous OR (`AX_OR_2`)— suitable for a command that is held (not toggled).

- **Generic for Simple Outputs**: Applicable to any simple, non-double-acting output controlled by a module without its own VT.

Application Scenarios

- Physical outputs whose existing I/O test channel should remain usable even after the actual function is implemented, without the two command sources blocking each other.
- Simple, non-double-acting actuators (e.g., strobe light, lighting) that can be controlled by both the I/O test and an actual operating function (SoftKey/AUX).

Comparison with Similar Function Blocks

For double-acting actuators (left/right, up/down) where the two directions must be mutually exclusive, use `ILOCK_SWITCH_2_QXA_OPC` — `MERGE_SWITCH_1_QXA_OPC` does not have an interlock because it only controls a single output direction.

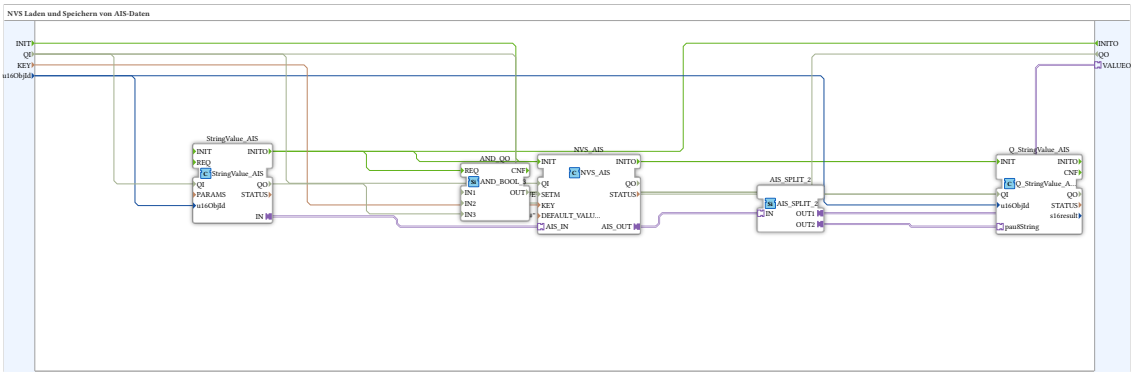
Summary

`MERGE_SWITCH_1_QXA_OPC` combines an I/O test command and a real function command into a single, simple output using a level OR logic gate and reports the status separately to both monitoring paths.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

NVS_IN_AND_STORE_AIS

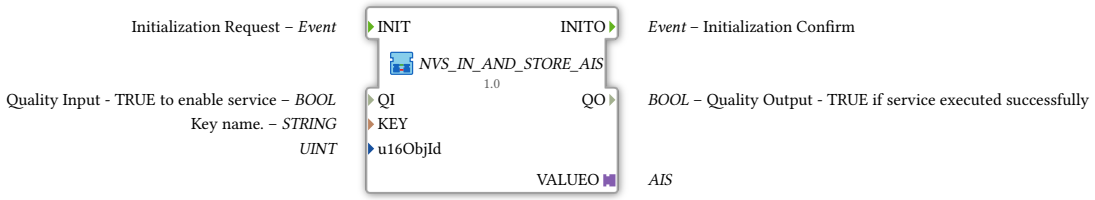


• — ○ • — ○ • — ○ •

Introduction

`NVS_IN_AND_STORE_AIS` persistently stores a string value entered via the VT (AIS adapter) in ESP32 flash (NVS) instead of an INI file — otherwise identical to `INI_IN_AND_STORE_AIS` (no `SECTION` parameter, since NVS is organized differently from INI files).

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).



Quality ports (QI/QO)

The block has the data input `QI` (`BOOL`, default `TRUE`, Quality Input - TRUE enables the service) and the data output `QO` (`BOOL`, Quality Output - TRUE if the service was executed successfully). `QI` is passed to `Q_StringValue_AIS`, `NVS_AIS` and `StringValue_AIS`. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`), which combines `Q_StringValue_AIS.QO`, `NVS_AIS.QO` and `StringValue_AIS.QO`.

In addition, the block has the events `INIT` (input, `EInit`, initialization request) and `INITO` (output, `EInit`, initialization confirmation): `INIT` ⟶ `Q_StringValue_AIS.INIT`, `Q_StringValue_AIS.INITO` ⟶ `NVS_AIS.INIT`, `NVS_AIS.INITO` ⟶ `StringValue_AIS.INIT`, `StringValue_AIS.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`).

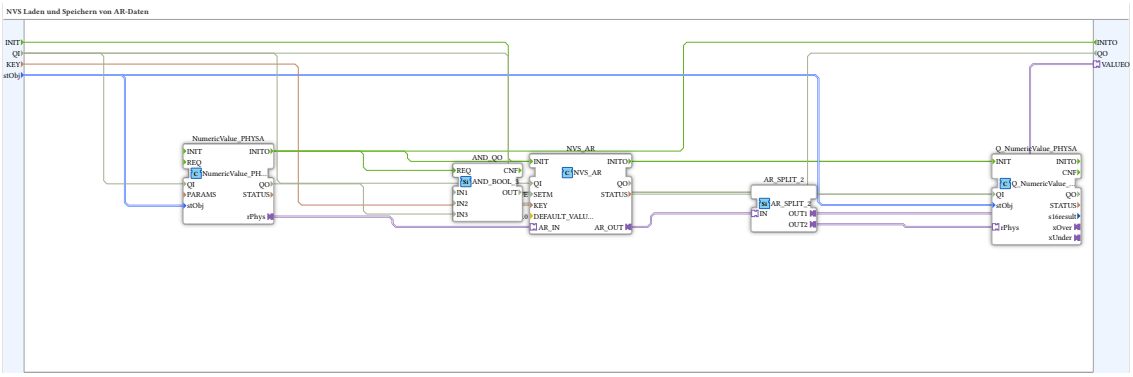
Summary

NVS counterpart to `INI_IN_AND_STORE_AIS`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

NVS_IN_AND_STORE_AR

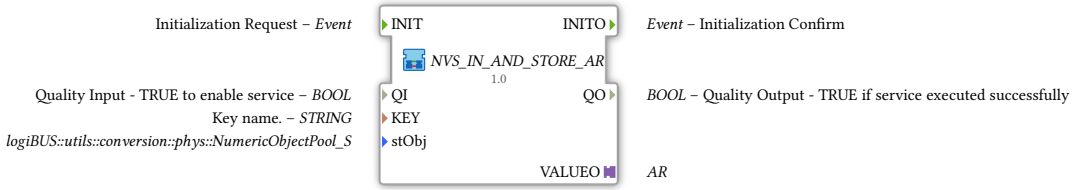


• – ○ • – ○ • – ○ •

Introduction

`NVS_IN_AND_STORE_AR` persistently stores a physically scaled REAL value entered via the VT (AR adapter) in ESP32 flash (NVS) — otherwise identical to `INI_IN_AND_STORE_AR`.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).



Quality ports (QI/QO)

The block has the data input `QI` (`BOOL`, default `TRUE`, Quality Input - TRUE enables the service) and the data output `QO` (`BOOL`, Quality Output - TRUE if the service was executed successfully). `QI` is connected to `Q_NumericValue_PHYSA.QI`, `NVS_AR.QI` and `NumericValue_PHYSA.QI`; `QO` is the AND (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_3`) of `Q_NumericValue_PHYSA.QO`, `NVS_AR.QO` and `NumericValue_PHYSA.QO`.

In addition, the block has the events `INIT` (input, `EInit`, initialization request) and `INITO` (output, `EInit`, initialization confirmation): initialization is chained: `INIT` ⟶ `Q_NumericValue_PHYSA.INIT`; `Q_NumericValue_PHYSA.INITO` ⟶ `NVS_AR.INIT`; `NVS_AR.INITO` ⟶ `NumericValue_PHYSA.INIT`; `NumericValue_PHYSA.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`).

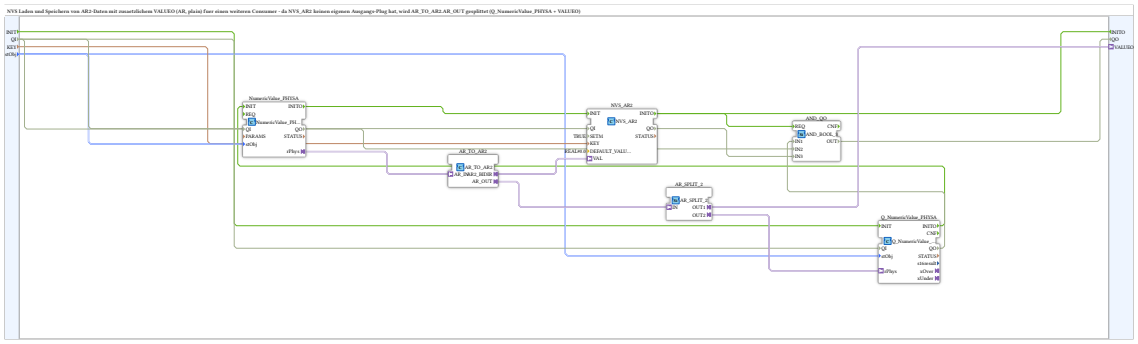
Summary

NVS counterpart to `INI_IN_AND_STORE_AR`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

NVS_IN_AND_STORE_AR2

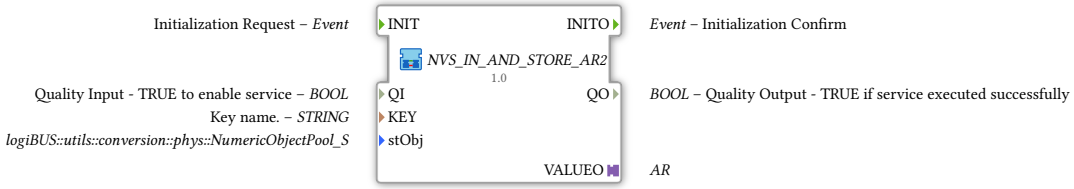


• – ○ • – ○ • – ○ •

Introduction

`NVS_IN_AND_STORE_AR2` is the AR2-adapter variant of `NVS_IN_AND_STORE_AR` (ESP32 flash storage instead of an INI file): a physically scaled REAL value entered via the VT is persistently stored in NVS flash via `logiBUS::storage::esp32_nvs::NVS_AR2`. Since `NVS_AR2` (unlike the INI variant) has no `SECTION` concept either, that parameter is absent compared to `INI_IN_AND_STORE_AR2`. As there, `AR_TO_AR2.AR_OUT` is duplicated via `AR_SPLIT_2`: VT display (`Q_NumericValue_PHYSA`) + `VALUEO` for a further consumer.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).



Technical notes

- `SETM=TRUE`, same reasoning as `INI_IN_AND_STORE_AR2`: live changes are confirmed immediately, not only at the next boot.
- No `SECTION` parameter, since NVS keys are flat (no INI-section concept).

Quality ports (QI/QO)

The block has the data input `QI` (`BOOL`, default `TRUE`, Quality Input - TRUE enables the service) and the data output `QO` (`BOOL`, Quality Output - TRUE if the service was executed successfully). `QI` is distributed to `Q_NumericValue_PHYSA`, `NumericValue_PHYSA` and `NVS_AR2`; their `QO` outputs are combined by `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`) into `QO`.

In addition, the block has the events `INIT` (input, `EInit`, initialization request) and `INITO` (output, `EInit`, initialization confirmation). Initialization: `INIT` ⟶ `Q_NumericValue_PHYSA.INIT`; `Q_NumericValue_PHYSA.INITO` ⟶ `NumericValue_PHYSA.INIT`; `NumericValue_PHYSA.INITO` ⟶ `NVS_AR2.INIT`; `NVS_AR2.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`. Quality: `QI` ⟶ `Q_NumericValue_PHYSA.QI`, `NumericValue_PHYSA.QI`, `NVS_AR2.QI`; `Q_NumericValue_PHYSA.QO`, `NumericValue_PHYSA.QO`, `NVS_AR2.QO` ⟶ `AND_QO.IN1` to `AND_QO.IN3` (in this order); `AND_QO.OUT` ⟶ `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_3`).

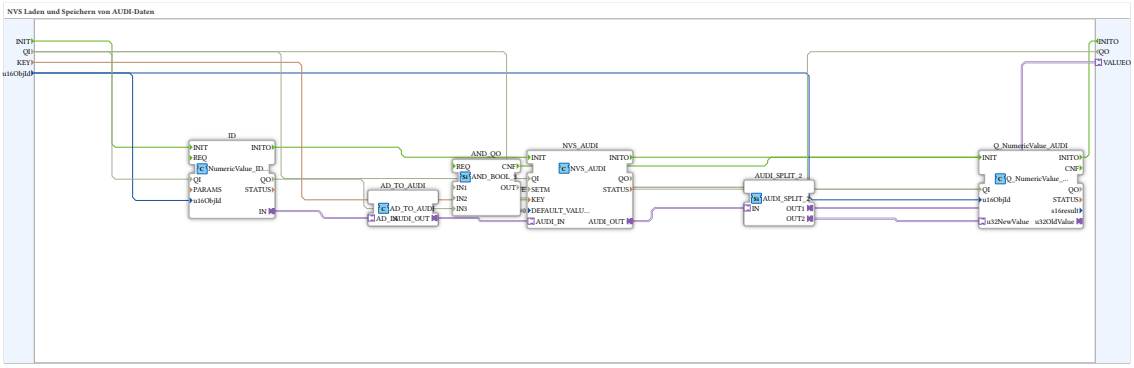
Summary

AR2-adapter variant of the NVS storage family (ESP32 flash instead of an INI file), with an additional `VALUEO` output.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

NVS_IN_AND_STORE_AUDI

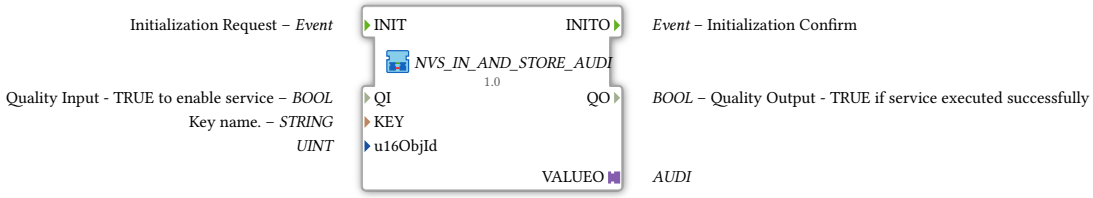


• – ○ • – ○ • – ○ •

Introduction

`NVS_IN_AND_STORE_AUDI` persistently stores a UDINT value entered via the VT (AUDI adapter) in ESP32 flash (NVS) — otherwise identical to `INI_IN_AND_STORE_AUDI`.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).



Quality ports (QI/QO)

The block has the event input `INIT` (`EInit`, Initialization Request), the event output `INITO` (`EInit`, Initialization Confirm), the data input `QI` (`BOOL`, default `TRUE`, Quality Input - TRUE enables the service) and the data output `QO` (`BOOL`, Quality Output - TRUE if the service was executed successfully).

- Initialization chain: `INIT` ⟶ `ID.INIT`, `ID.INITO` ⟶ `NVS_AUDI.INIT`, `NVS_AUDI.INITO` ⟶ `Q_NumericValue_AUDI.INIT`, `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (and additionally ⟶ `AND_QO.REQ`)
- `QI` is connected to `Q_NumericValue_AUDI.QI`, `NVS_AUDI.QI` and `ID.QI`
- `QO` is the AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_3`) of `Q_NumericValue_AUDI.QO`, `NVS_AUDI.QO` and `ID.QO`

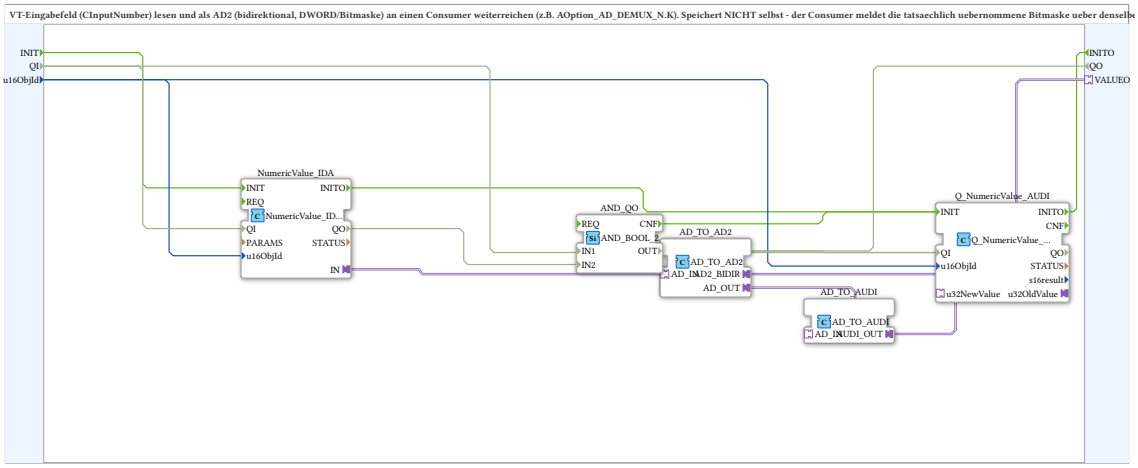
Summary

NVS counterpart to `INI_IN_AND_STORE_AUDI`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

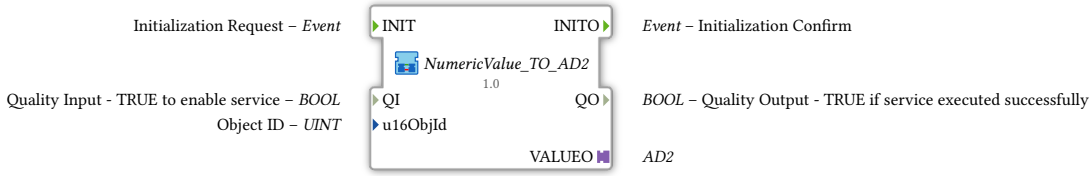
NumericValue_TO_AD2



• — ○ • — ○ • — ○ •

Introduction

`NumericValue_TO_AD2` reads a VT input field (CInputNumber) and passes the value bidirectionally as an AD2 adapter (DWORD/bitmask, `VALUEO`) to a consumer (e.g. `AOption_AD_DEMUX_N.K`). The block does **not** store anything itself: the consumer reports the bitmask it actually accepted back over the same AD2 plug, and the echo is written back to the VT field.



Interface

Input variables

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE enables the service; passed through to `NumericValue_IDA` and `Q_NumericValue_AUDI`.
- `u16ObjId` (UINT, default `ID_NULL`): ObjectID of the VT input field.

Output variables

- `QO` (BOOL): Quality Output - TRUE if the service was executed successfully; `AND` of the `QO` outputs of `Q_NumericValue_AUDI` and `NumericValue_IDA` (via `AND_QO`).

Adapters

- **Plug `VALUEO`** (`adapter::types::bidirectional::AD2`): bitmask to the consumer and echo back.

Events

- `INIT` (EInit, input): initialization request; starts the chain `NumericValue_IDA` ⟶ `Q_NumericValue_AUDI`.
- `INITO` (EInit, output): initialization confirmation; comes from `Q_NumericValue_AUDI.INITO`.

How it works

1. `NumericValue_IDA` reads the VT input field.
2. `AD_TO_AD2` converts the value into the bidirectional AD2 adapter, which is output on `VALUEO`.
3. The echo reported back by the consumer appears at `AD_TO_AD2.AD_OUT`, is converted to UDINT via `AD_TO_AUDI`, and written to the VT field under `u16ObjId` by `Q_NumericValue_AUDI` (`u32NewValue`).

`QI` is passed through to `NumericValue_IDA.QI` and `Q_NumericValue_AUDI.QI` (no fixed `QI`=TRUE any more). Initialization runs as the chain `INIT` ⟶ `NumericValue_IDA.INIT`, `NumericValue_IDA.INITO` ⟶ `Q_NumericValue_AUDI.INIT`, `Q_NumericValue_AUDI.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`. `QO` is the `AND` (`AND_QO`, `AND_BOOL_2`) of the `QO` outputs of `Q_NumericValue_AUDI` and `NumericValue_IDA`.

Technical notes

- DWORD/bitmask counterpart of `NumericValue_TO_AUI2` - for `AOption_AD_DEMUX_N` (multi-select) instead of `AOption_AUI_DEMUX_N` (single-select).
- No `AD_TO_AUI` intermediate step is needed, since AD2 is already a DWORD.
- Pure VT-to-AD2 bridge without a storage FB; compare `NumericValue_TO_AR2`.

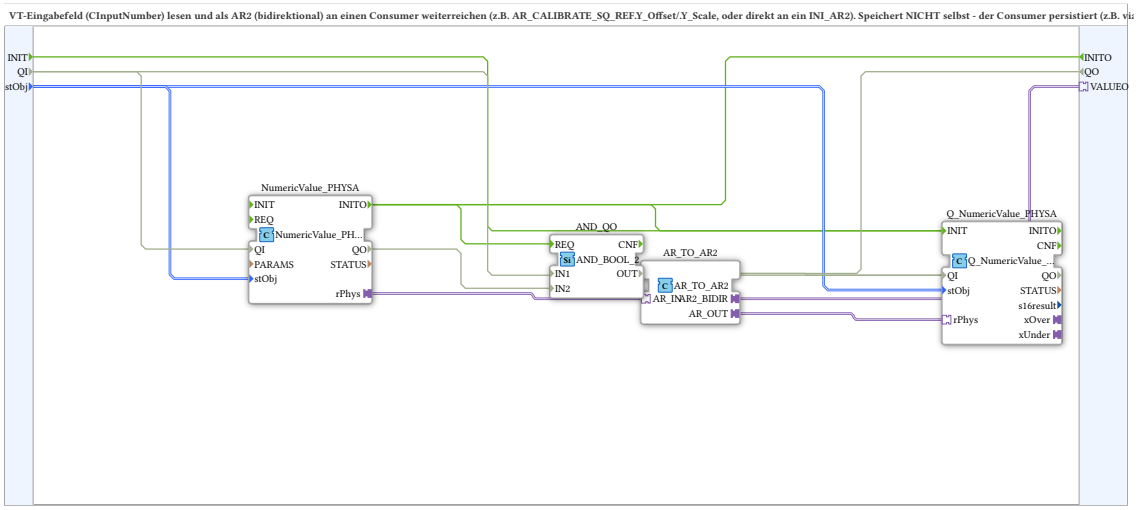
Summary

Bidirectional bridge from the VT input field to an AD2 plug (bitmask) without its own persistence.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

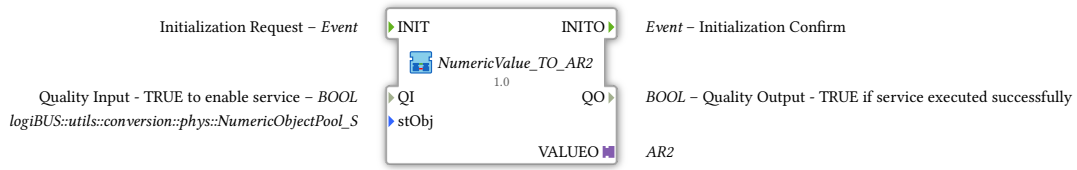
NumericValue_TO_AR2



• – ○ • – ○ • – ○ •

Introduction

`NumericValue_TO_AR2` reads a VT input field (`NumericValue_PHYSA`) and forwards the value bidirectionally as an AR2 adapter (`VALUEO`) to a consumer - unlike `INI_IN_AND_STORE_AR2`, this block itself stores **nothing**: persistence is the consumer's responsibility (e.g. `AR_CALIBRATE.Y_Offset/.Y_Scale`, or directly an `INI_AR2`). Whatever the consumer echoes back over the same AR2 plug is written back to the VT field - even at boot, so the input field and the actually active value stay in sync.



Technical notes

- Pure VT-to-AR2 bridge: no storage FB in the network, unlike the INI/NVS family.
- Historical note: per the source comment, renamed from `INI_IN_AND_STORE_AR2` when the block changed from a self-contained storage function to a pure bridge.

Quality ports (QI/QO)

The block has the data input `QI` (`BOOL`, default `TRUE`, Quality Input - TRUE enables the service) and the data output `QO` (`BOOL`, Quality Output - TRUE if the service was executed successfully). `QI` is passed on to `Q_NumericValue_PHYSA.QI` and `NumericValue_PHYSA.QI`; their `QO` outputs are combined in `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`, `Q_NumericValue_PHYSA.QO` ⟶ `IN1`, `NumericValue_PHYSA.QO` ⟶ `IN2`) into the output `QO`.

In addition, the block has the events `INIT` (input, `EInit`, initialization request) and `INITO` (output, `EInit`, initialization confirmation): Init chain: `INIT` ⟶ `Q_NumericValue_PHYSA.INIT`, `Q_NumericValue_PHYSA.INITO` ⟶ `NumericValue_PHYSA.INIT`, `NumericValue_PHYSA.INITO` ⟶ `INITO` and `AND_QO.REQ`.

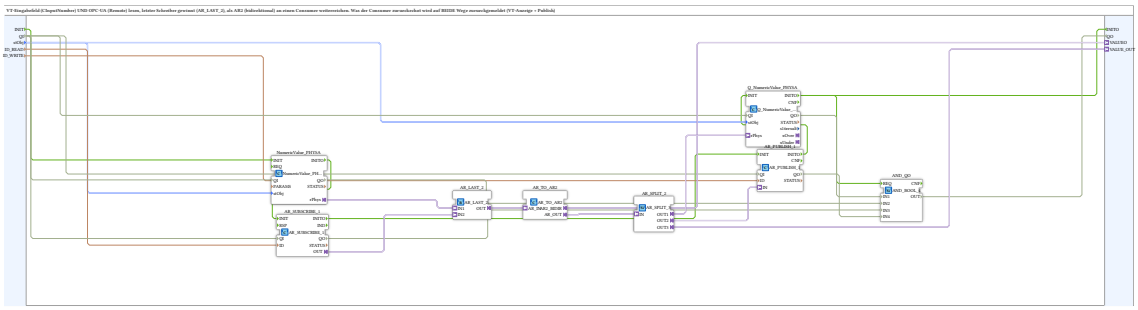
Summary

Bidirectional VT-input-field-to-AR2 bridge with no storage of its own - the consumer persists, this block only displays and forwards.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

NumericValue_TO_AR2_OPC



• – ○ • – ○ • – ○ •

Introduction

`NumericValue_TO_AR2_OPC` is the VT+OPC-UA sibling of `NumericValue_TO_AR2` (VT only) and `OPC_TO_AR2` (OPC only): both the VT input field and a remotely-written OPC-UA value (`AR_SUBSCRIBE_1`) feed `AR_LAST_2` - "last writer wins". The result goes to the consumer as an AR2 adapter (`VALUEO`); whatever the consumer echoes back is reported both ways: VT display (`Q_NumericValue_PHYSA`) and OPC-UA publish (`AR_PUBLISH_1`).



Function blocks used

- **NumericValue_PHYSA**: VT input field, local value.
- **AR_SUBSCRIBE_1**: subscribes to a remotely-written REAL value under `ID_READ`.
- **AR_LAST_2**: merges both sources, last writer wins.
- **AR_TO_AR2**: converts the merged plain-AR value into the bidirectional AR2 adapter for the consumer.
- **AR_SPLIT_2**: splits the consumer's echo to the VT display and the OPC publish.
- **AR_PUBLISH_1**: publishes the current/persisted value under `ID_WRITE` (triggered by `AR_SUBSCRIBE_1.INITO`, so the initial value is published immediately too).

Technical notes

- Merge strategy identical to the VT/web merge already proven in `RampLimitFS_TO_logiBUS_QDA_PWM_OPC.SUB` (per the source comment).

Event and quality ports (INIT/INITO, QI/QO)

The block has the data input `QI` (`BOOL`, default `TRUE`, Quality Input - TRUE enables the service) and the data output `QO` (`BOOL`, Quality Output - TRUE if the service was executed successfully). `QI` is passed to `NumericValue_PHYSA`, `AR_SUBSCRIBE_1`, `AR_PUBLISH_1` and `Q_NumericValue_PHYSA`. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`), which combines `Q_NumericValue_PHYSA.QO`, `NumericValue_PHYSA.QO`, `AR_SUBSCRIBE_1.QO` and `AR_PUBLISH_1.QO`.

In addition, the block has the event input `INIT` and the event output `INITO` (both `EInit`). Initialization runs as a chain: `INIT` ⟶ `NumericValue_PHYSA.INIT`, `NumericValue_PHYSA.INITO`

⟶ `AR_SUBSCRIBE_1.INIT`, `AR_SUBSCRIBE_1.INITO` ⟶ `AR_PUBLISH_1.INIT`, `AR_PUBLISH_1.INITO` ⟶ `Q_NumericValue_PHYSA.INIT`, `Q_NumericValue_PHYSA.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`).

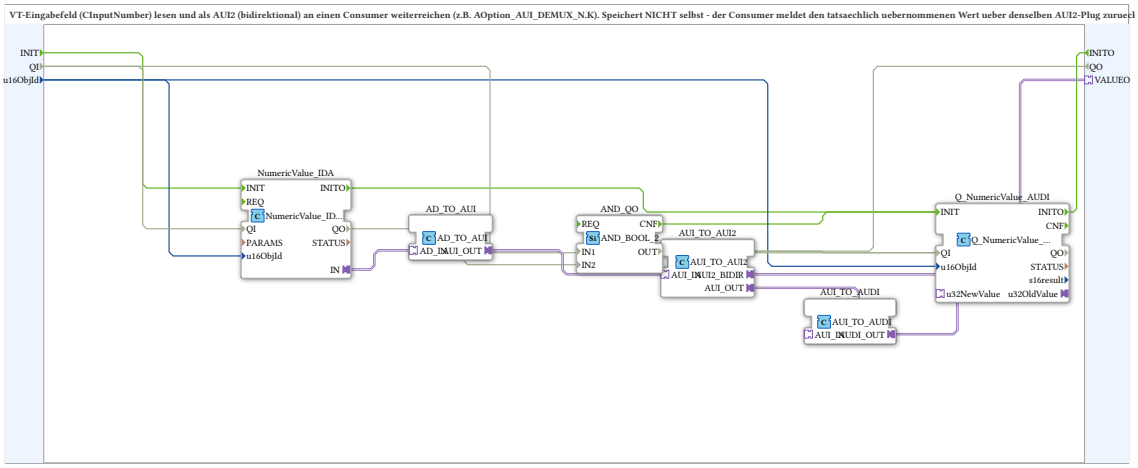
Summary

VT and OPC-UA input for the same AR2 consumer, with last-writer-wins merging and echo feedback on both paths.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

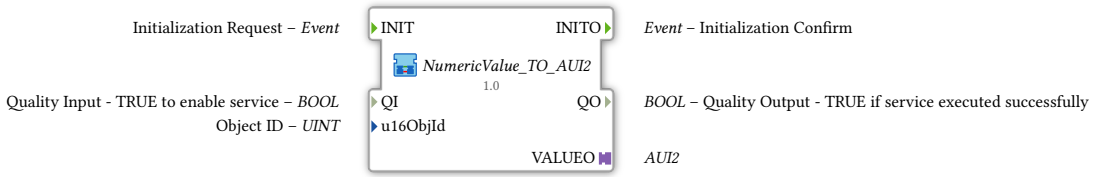
NumericValue_TO_AUI2



• – ○ • – ○ • – ○ •

Introduction

`NumericValue_TO_AUI2` reads a VT input field (CInputNumber) and passes the value bidirectionally as an AUI2 adapter (`VALUEO`) to a consumer (e.g. `AOption_AUI_DEMUX_N.K`). The block does **not** store anything itself: the consumer reports the value it actually accepted back over the same AUI2 plug, and the echo is written back to the VT field.



Interface

Input variables

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE enables the service.
- `u16ObjId` (UINT, default `ID_NULL`): ObjectID of the VT input field.

Output variables

- `QO` (BOOL): Quality Output - TRUE if the service was executed successfully (AND of `Q_NumericValue_AUDI.QO` and `NumericValue_IDA.QO`).

Adapters

- **Plug `VALUEO`** (`adapter::types::bidirectional::AUI2`): value to the consumer and echo back.

Events

- **INIT** (event input, `EInit`): initialization request
- **INITO** (event output, `EInit`): initialization confirmation

How it works

1. `NumericValue_IDA` reads the VT input field; `AD_TO_AUI` converts to AUI.
2. `AUI_TO_AUI2` converts into the bidirectional AUI2 adapter, which is output on `VALUEO`.
3. The echo reported back by the consumer appears at `AUI_TO_AUI2.AUI_OUT`, is converted to UDINT via `AUI_TO_AUDI`, and written to the VT field under `u16ObjId` by `Q_NumericValue_AUDI` (`u32NewValue`).

`QI` is distributed to `Q_NumericValue_AUDI.QI` and `NumericValue_IDA.QI`; `Q_NumericValue_AUDI.QO` and `NumericValue_IDA.QO` are combined into `QO` via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`).

Initialization: `INIT` ⟶ `NumericValue_IDA.INIT`; `NumericValue_IDA.INITO` ⟶ `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`).

Technical notes

- Extracted from `Uebung_094c_AX`, where this was duplicated inline per dropdown.
- Analogous to `NumericValue_TO_AR2`, but UINT/AUI2 instead of REAL/AR2 and a simple `u16ObjId` instead of `stObj` (no scale/offset/decimals needed for a pure index value).
- For a variant with INI persistence, see `NumericValue_TO_AUI2_AND_STORE`.

Summary

Bidirectional bridge from the VT input field to an AUI2 plug without its own persistence.

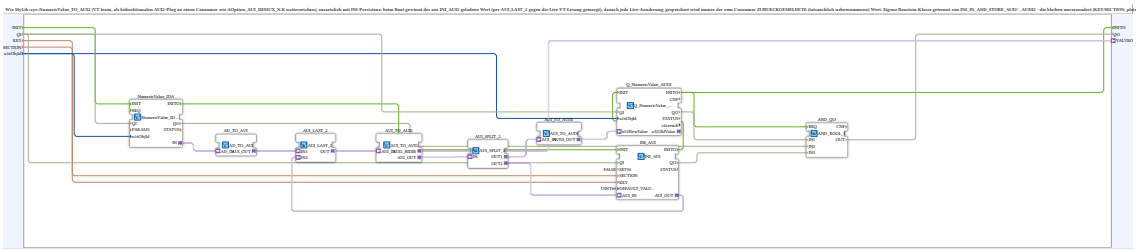
🛠️ Related Exercises

- Uebung_094b3_AX
- Uebung_094b4_AX
- Uebung_094c2_AX

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

NumericValue_TO_AUI2_AND_STORE

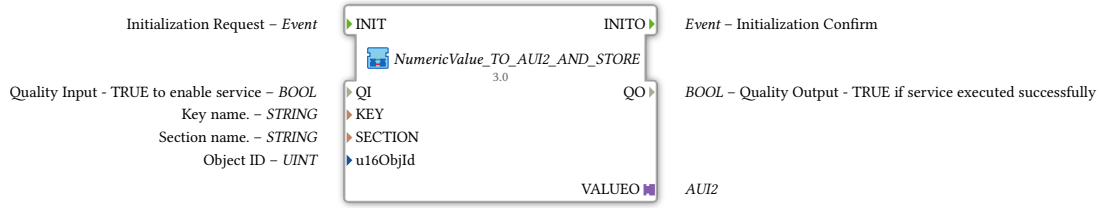


• — ○ • — ○ • — ○ •

Introduction

`NumericValue_TO_AUI2_AND_STORE` corresponds to `NumericValue_TO_AUI2` (read the VT, pass it as a bidirectional AUI2 plug `VALUEO` to a consumer such as `AOption_AUI_DEMUX_N.K`), extended with INI persistence: at boot the value loaded from the INI file wins (merged against the live VT reading via `AUI_LAST_2`), after that every live change. What is stored is always the value **reported back** by the consumer (the one actually accepted). The block is its own class, separate from `INI_IN_AND_STORE_AUI2`/`_AUDI2`, which remain unchanged (KEY/SECTION, plain `VALUEO`) because they are used in numerous other projects.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).



Interface

Event inputs

- `INIT` (`EInit`): initialization request.

Event outputs

- `INITO` (`EInit`): initialization confirm.

Input variables

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE enables the service.
- `KEY` (STRING): key name in the INI file.
- `SECTION` (STRING): section name in the INI file.
- `u16ObjId` (UINT, default `ID_NULL`): ObjectID of the VT input field.

Output variables

- `QO` (BOOL): Quality Output - TRUE if the service was executed successfully (AND combination `AND_QO` of the `QO` outputs of the inner blocks).

Adapters

- **Plug `VALUEO`** (`adapter::types::bidirectional::AUI2`): value to the consumer and echo back.

The events `INIT`/`INITO` are passed as a chain through the inner blocks (see How it works).

How it works

1. `NumericValue_IDA` reads the VT input field; `AD_TO_AUI` converts to AUI and feeds `AUI_LAST_2.IN1`.
2. `AUI_LAST_2` (last writer wins) merges: `IN1` = live VT change, `IN2` = INI value at boot. Its output goes to `AUI_TO_AUI2` (bridge to the external consumer), whose bidirectional output goes to `VALUEO`.
3. The result reported back by the consumer appears at `AUI_TO_AUI2.AUI_OUT` and is duplicated via `AUI_SPLIT_2`: `OUT1` via `AUI_TO_AUDI` to `Q_NumericValue_AUDI` (`u32NewValue`, VT display under `u16ObjId`), `OUT2` to `INI_AUI` (`eclipse4diac::storage::INI_AUI`, `AUI_IN`; `SETM=FALSE`, `DEFAULT_VALUE=UINT#0`).
4. `INI_AUI.AUI_OUT` feeds the loaded value back to `AUI_LAST_2.IN2`.

Initialization: `INIT` → `NumericValue_IDA.INIT`; `NumericValue_IDA.INITO` → `INI_AUI.INIT`; `INI_AUI.INITO` → `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` → `INITO` and → `AND_QO.REQ`. Quality: `QI` → `Q_NumericValue_AUDI.QI`, `NumericValue_IDA.QI`, `INI_AUI.QI`; `Q_NumericValue_AUDI.QO`, `NumericValue_IDA.QO`, `INI_AUI.QO` → `AND_QO.IN1` to `AND_QO.IN3` (in this order); `AND_QO.OUT` → `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_3`).

Technical notes

- `SETM=FALSE`: `INI_AUI` acknowledges automatically only at boot (INIT → INITO → GET → GETO), not on every later write. With `SETM=TRUE` every write would trigger a new echo that would arrive at `AUI_LAST_2` again and be sent towards `VALUEO` - an endless loop.
- The live VT reading goes directly to `VALUEO` (a single `AUI_TO_AUI2`); only the result reported back by the consumer is stored, via the split.
- The boot value wins once in `AUI_LAST_2` against the live VT reading.

Summary

Like `NumericValue_TO_AUI2`, with INI persistence of the consumer-confirmed value and a boot value merge.

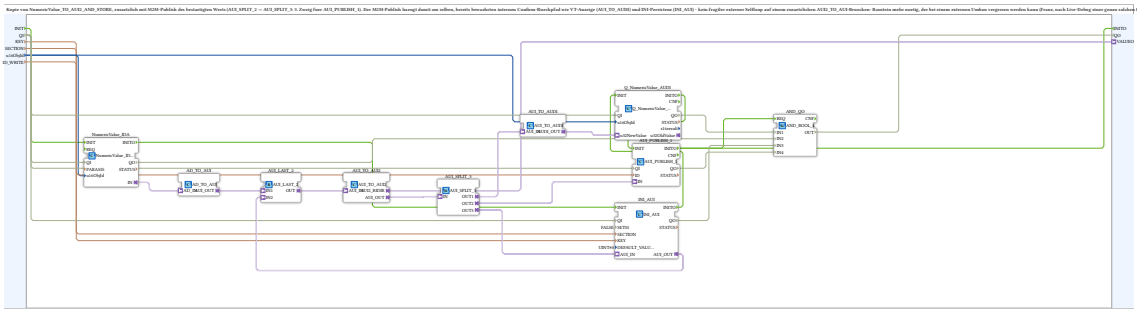
🛠️ Related Exercises

- Uebung_094b3b_AX
- Uebung_094b4b_AX
- Uebung_094c2b_AX

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

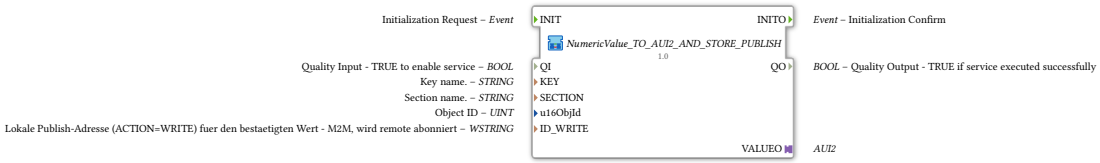
NumericValue_TO_AUI2_AND_STORE_PUBLISH



• — ○ • — ○ • — ○ •

Introduction

`NumericValue_TO_AUI2_AND_STORE_PUBLISH` is a copy of `NumericValue_TO_AUI2_AND_STORE` that additionally publishes the confirmed value via M2M publish: `AUI_SPLIT_2` was extended to `AUI_SPLIT_3`, and the third branch feeds `AUI_PUBLISH_1`. The M2M publish thus sits on the same internal confirm return path as the VT display (`AUI_TO_AUDI`) and the INI persistence (`INI_AUI`). An additional external self-loop via an `AUI2_TO_AUI` bridge block is no longer needed.



Interface

Input variables

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE enables the service.
- `KEY` (STRING): key name in the INI file.
- `SECTION` (STRING): section name in the INI file.
- `u16ObjId` (UINT, default `ID_NULL`): ObjectID of the VT input field.
- `ID_WRITE` (WSTRING): local publish address (`ACTION=WRITE`) for the confirmed value; M2M, subscribed remotely.

Output variables

- `QO` (BOOL): Quality Output - TRUE if all blocks involved succeeded (AND over `Q_NumericValue_AUDI`, `NumericValue_IDA`, `INI_AUI`, `AUI_PUBLISH_1`).

Adapters

- **Plug `VALUEO`** (`adapter::types::bidirectional::AUI2`): value to the consumer and echo back.

The block has the events `INIT` (input) and `INITO` (output).

Function Blocks (FBs) Used

- **NumericValue_IDA** (`isobus::UT::io::NumericValue::NumericValue_IDA`): reads the VT input field.
- **AD_TO_AUI** (`adapter::conversion::unidirectional::AD_TO_AUI`): converts the value to AUI.
- **AUI_LAST_2** (`adapter::events::unidirectional::AUI_LAST_2`): last writer wins; at boot the INI value wins (`IN2`), afterwards every live VT change (`IN1`).
- **AUI_TO_AUI2** (`adapter::conversion::bidirectional::AUI_TO_AUI2`): bridge to the external consumer (e.g. `AOption_AUI_DEMUX_N.K`).
- **AUI_SPLIT_3** (`adapter::events::unidirectional::AUI_SPLIT_3`): third branch `OUT3` for `INI_AUI`, `OUT2` for `AUI_PUBLISH_1`, `OUT1` for the VT display.
- **AUI_TO_AUDI** (`adapter::conversion::unidirectional::AUI_TO_AUDI`) and **Q_NumericValue_AUDI** (`isobus::UT::Q::Q_NumericValue_AUDI`): VT display under `u16ObjId`.
- **INI_AUI** (`eclipse4diac::storage::INI_AUI`): parameters `SETM=FALSE`, `DEFAULT_VALUE=UINT#0`.

- **AUI_PUBLISH_1** (`adapter::net::AUI_PUBLISH_1`): M2M publish of the confirmed value on `ID_WRITE`.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_4`): combines the four `QO` outputs.

How it works

1. `NumericValue_IDA` reads the VT input field; `AD_TO_AUI` converts to AUI and feeds `AUI_LAST_2.IN1`.
2. `AUI_LAST_2` goes to `AUI_TO_AUI2`, whose bidirectional output goes to `VALUEO`.
3. The result reported back by the consumer appears at `AUI_TO_AUI2.AUI_OUT` and is distributed by `AUI_SPLIT_3`: `OUT1 -> AUI_TO_AUDI -> Q_NumericValue_AUDI.u32NewValue` (VT display), `OUT2 -> AUI_PUBLISH_1.IN` (M2M publish), `OUT3 -> INI_AUI.AUI_IN` (persistence).
4. `INI_AUI.AUI_OUT` feeds the loaded value back to `AUI_LAST_2.IN2`.
5. Initialization chain: `INIT -> NumericValue_IDA -> INI_AUI -> Q_NumericValue_AUDI -> AUI_PUBLISH_1 -> INITO`. `AUI_PUBLISH_1.INITO` also triggers `AND_QO.REQ`, whose result goes to `QO`.
6. `QI` is distributed to `Q_NumericValue_AUDI`, `NumericValue_IDA`, `INI_AUI` and `AUI_PUBLISH_1`.

Technical notes

- `SETM`=FALSE: `INI_AUI` acknowledges automatically only at boot, not on every later write. With `SETM`=TRUE every write would trigger a new echo and create an endless loop (see also `NumericValue_TO_AUI2_AND_STORE`).
- The M2M publish deliberately sits in the already proven internal return path so that it cannot get lost during an external rework.
- The block is its own class; `NumericValue_TO_AUI2_AND_STORE` remains unchanged.

Application Scenarios

- Numeric VT input field with INI persistence where the value confirmed by the consumer is additionally to be published to other modules.

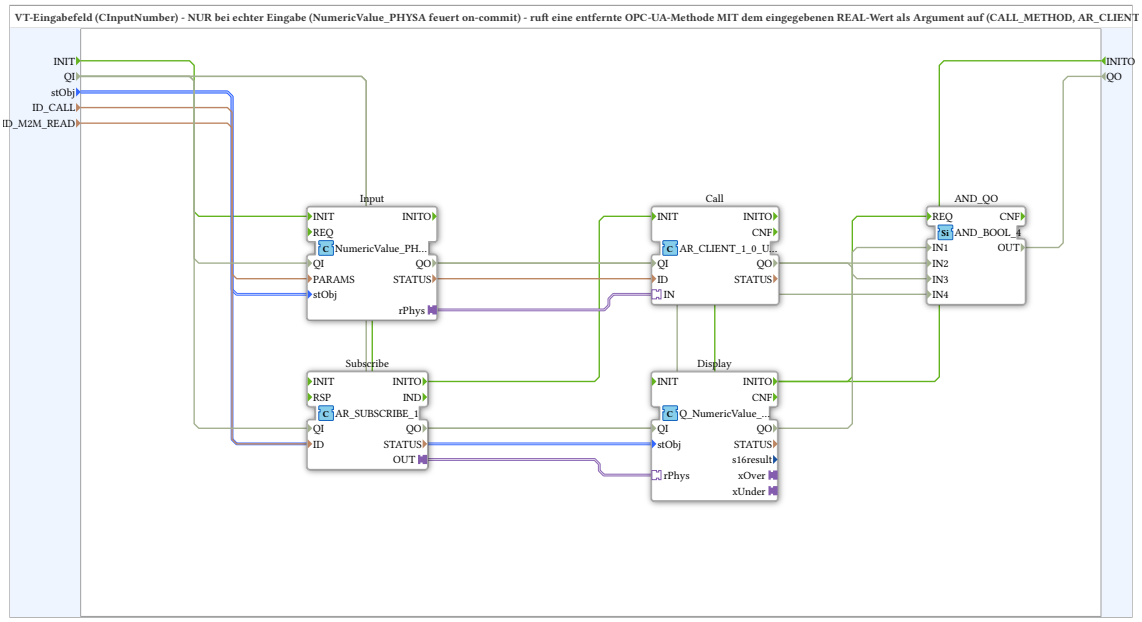
Summary

Like `NumericValue_TO_AUI2_AND_STORE`, with an additional M2M publish of the confirmed value on the third branch of `AUI_SPLIT_3`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

NumericValue_TO_CLIENT_1_0_OPC



• – ○ • – ○ • – ○ •

Introduction

`NumericValue_TO_CLIENT_1_0_OPC` connects a VT input field (`CInputNumber`) to a remote **OPC-UA method**: **only on a genuine input** (`NumericValue_PHYSA` fires on commit) is the method (`CALL_METHOD`, `AR_CLIENT_1_0`) called with the entered `REAL` value as argument. The return path - displaying the actually effective, confirmed value on the same VT field - runs separately and uncritically via a normal subscribe (`Q_NumericValue_PHYSA`). Hence there is no circular reference and no merge/last-writer-wins: on the receiver side there is only one caller per method, serialized by the method call itself.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization Request

Event Outputs

- **INITO** (`EInit`): Initialization Confirm

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): Object ID and scale/offset/decimals of the VT input field; used for input and output (the confirmed value is written back here). Default: `(u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 1)`
- **ID_CALL** (`WSTRING`): Remote method address (`ACTION=CALL_METHOD`); called only on a genuine VT input with the entered value as argument
- **ID_M2M_READ** (`WSTRING`): Remote subscribe address (`REAL`, `ACTION=SUBSCRIBE`); returns the actually effective value for display on the same VT field (read only)

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully (AND of `Display.QO`, `Input.QO`, `Call.QO` and `Subscribe.QO`)

Adapters

This function block has no adapter sockets/plugs at its interface; `AR` adapters are used internally.

Function Blocks (FBs) Used

- **Input**: `isobus::UT::io::NumericValue::NumericValue_PHYSA` (`QI` from the block input) - provides the entered physical value (`rPhys`) on commit
- **Call**: `adapter::net::AR_CLIENT_1_0_UNGATED` (`QI` from the block input) - calls the remote method with the value as argument
- **Subscribe**: `adapter::net::AR_SUBSCRIBE_1` (`QI` from the block input) - subscribes to the confirmed value
- **Display**: `isobus::UT::Q::Q_NumericValue_PHYSA` - writes the value to the VT field
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_4` - ANDs the `QO` outputs of `Display`, `Input`, `Call` and `Subscribe`

Program Flow and Connections

1. **Forward path**: `Input.rPhys` ⟶ `Call.IN` (input ⟶ method call).
2. **Return path**: `Subscribe.OUT` ⟶ `Display.rPhys` (confirmed value ⟶ VT display).
3. Initialization chain: `INIT` ⟶ `Input.INIT`, `Input.INITO` ⟶ `Subscribe.INIT`, `Subscribe.INITO` ⟶ `Call.INIT` (the client is initialized only after the subscribe has been initialized), `Call.INITO` ⟶ `Display.INIT`, `Display.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
4. `stObj` ⟶ `Input.stObj` and `Display.stObj`; `ID_CALL` ⟶ `Call.ID`; `ID_M2M_READ` ⟶ `Subscribe.ID`.
5. `QI` ⟶ `Display.QI`, `Input.QI`, `Call.QI` and `Subscribe.QI`; `Display.QO` ⟶ `AND_QO.IN1`, `Input.QO` ⟶ `AND_QO.IN2`, `Call.QO` ⟶ `AND_QO.IN3`, `Subscribe.QO` ⟶ `AND_QO.IN4`; `AND_QO.OUT` ⟶ `QO`

Technical Details

- Forward and return paths are separate, so no merge is needed.
- `AR_CLIENT_1_0_UNGATED` is the ungated variant of the client.

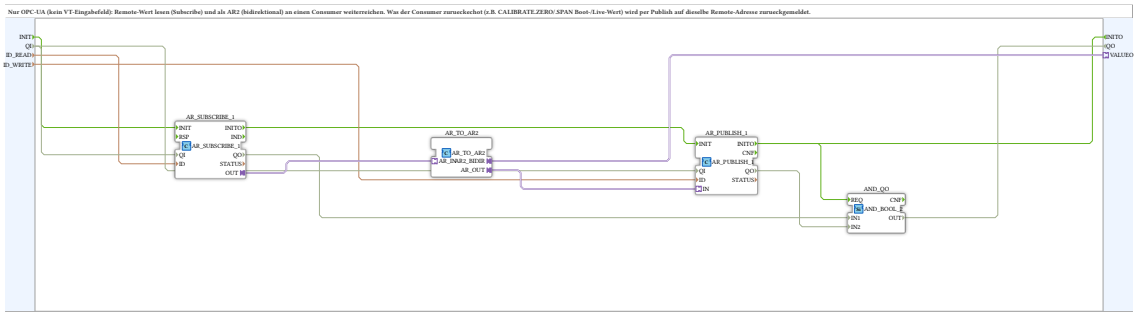
Application Scenarios

- Setpoint entry on the VT that takes effect on another station via an OPC-UA method, with feedback of the actually valid value

Summary

`NumericValue_TO_CLIENT_1_0_OPC` writes VT inputs to the target station via an OPC-UA method call and displays the confirmed value independently via a subscribe.

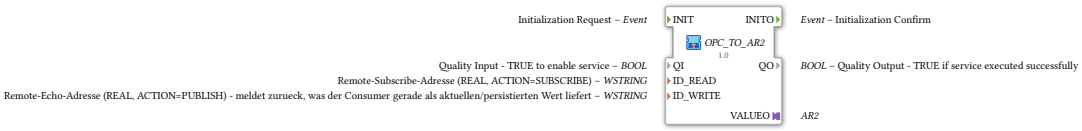
OPC_TO_AR2



• — ○ • — ○ • — ○ •

Introduction

`OPC_TO_AR2` is the pure OPC-UA sibling of `NumericValue_TO_AR2` - no VT input field involved. A remotely-written value is subscribed via `AR_SUBSCRIBE_1` and forwarded to a consumer as an AR2 adapter (`VALUEO`), e.g. `CALIBRATE.ZERO`/`.SPAN`. Whatever the consumer echoes back (current/persisted value) is reported back to the same remote address via `AR_PUBLISH_1`.



Interface and initialization

- Events: `INIT` (EInit, input) and `INITO` (EInit, output, comes from `AR_PUBLISH_1.INITO`).
- Data: `QI` (BOOL, default `TRUE`) as well as `ID_READ` and `ID_WRITE` (WSTRING) as inputs, `QO` (BOOL) as output; adapter plug `VALUEO` (AR2).
- Initialization: `INIT` ⟶ `AR_SUBSCRIBE_1.INIT`, `AR_SUBSCRIBE_1.INITO` ⟶ `AR_PUBLISH_1.INIT`, `AR_PUBLISH_1.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
- Quality: `QI` ⟶ `AR_SUBSCRIBE_1.QI` and `AR_PUBLISH_1.QI` (no fixed `QI=TRUE` any more); `AR_SUBSCRIBE_1.QO` ⟶ `AND_QO.IN1`, `AR_PUBLISH_1.QO` ⟶ `AND_QO.IN2` (`AND_BOOL_2`), `AND_QO.OUT` ⟶ `QO`.

Technical notes

- `AR_PUBLISH_1` is triggered by `AR_SUBSCRIBE_1.INITO` so the boot value is published immediately, not only on the first change.

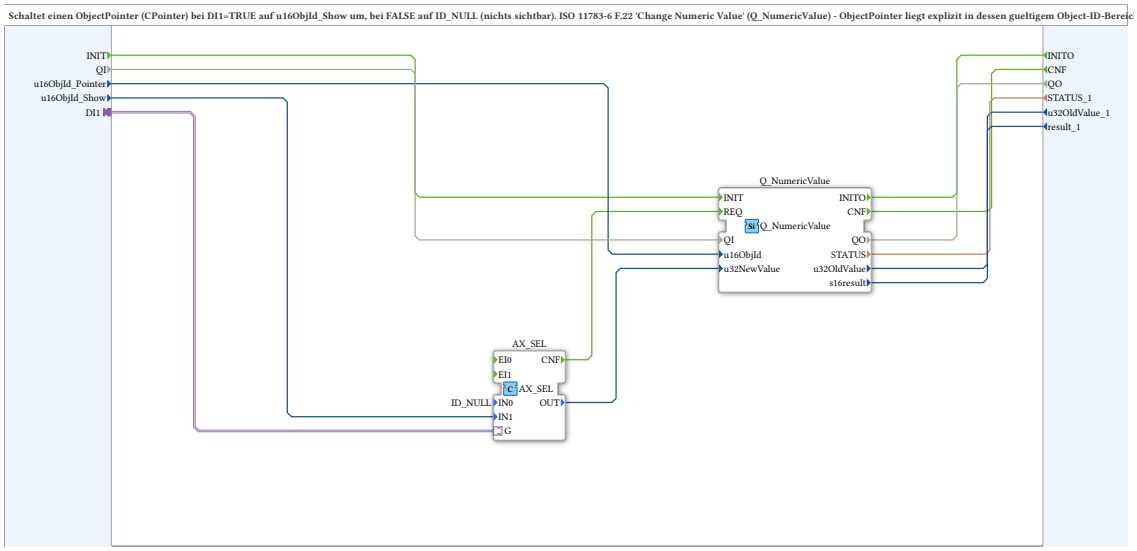
Summary

Purely remote (OPC-UA) driven AR2 bridge with no VT involvement at all - for consumers parameterized exclusively from a control room.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

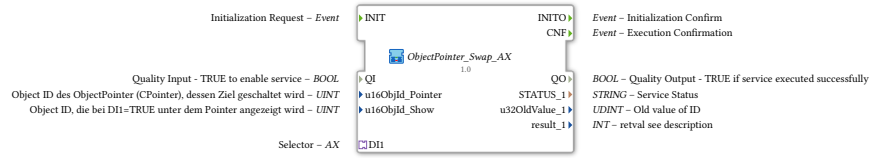
ObjectPointer_Swap_AX



• — ○ • — ○ • — ○ •

Introduction

`ObjectPointer_Swap_AX` switches an object pointer (CPointer) in the VT: with `DI1 = TRUE` it points to `u16ObjId_Show`, with FALSE to `ID_NULL` (nothing visible). The switching uses ISO 11783-6 F.22 “Change Numeric Value” (`Q_NumericValue`); the object pointer lies explicitly within its valid object ID range (27000-27999). F.38 “Change Attribute” (`Q_Attribute`) must not be used because `AID_OP_PTR.VALUE` is read-only there.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization request (to `Q_NumericValue.INIT`)

Event Outputs

- **INITO** (`EInit`): Initialization confirmation (from `Q_NumericValue.INITO`)
- **CNF**: execution confirmation, passed through from `Q_NumericValue.CNF`

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **u16ObjId_Pointer** (UINT): object ID of the object pointer (CPointer) whose target is switched (default `ID_NULL`)
- **u16ObjId_Show** (UINT): object ID that is shown under the pointer when `DI1 = TRUE` (default `ID_NULL`)

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully
- **STATUS_1** (STRING): service status of `Q_NumericValue`
- **u32OldValue_1** (UDINT): old value of the ID
- **result_1** (INT): return value (see `Q_NumericValue`)

Adapters

- **DI1** (socket, `adapter::types::unidirectional::AX`): selector

Functionality

The network consists of two blocks:

- `AX_SEL` (`adapter::iec61131::selection::AX_SEL`): binary selection with `IN0 = ID_NULL` (parameter), `IN1` = `u16ObjId_Show` and selector `G` = `DI1`. An output event is generated only when `G` changes.
- `Q_NumericValue` (`isobus::UT::Q::Q_NumericValue`): `AX_SEL.OUT` ⟶ `u32NewValue`, `u16ObjId_Pointer` ⟶ `u16ObjId`, triggered by `AX_SEL.CNF` ⟶ `Q_NumericValue.REQ`. `Q_NumericValue.CNF` ⟶ `CNF`; `STATUS`, `u32OldValue` and `s16result` are output as `STATUS_1`, `u32OldValue_1` and `result_1`.

When `DI1` changes, the new target value (`u16ObjId_Show` or `ID_NULL`) is thereby written as a numeric value to the object pointer.

In addition: `INIT` ⟶ `Q_NumericValue.INIT`; `Q_NumericValue.INITO` ⟶ `INITO`; `QI` ⟶ `Q_NumericValue.QI`; `Q_NumericValue.QO` ⟶ `QO`.

Technical Details

- Structure analogous to `GreenWhiteBackground1_AX` (AX_SEL wrapper around a `Q_` FB)
- `Q_NumericValue` (F.22) is the right way for object pointers; `Q_Attribute` (F.38) was discarded
- For the pattern see the exercise `Uebung_015` in the workspace `test_B`

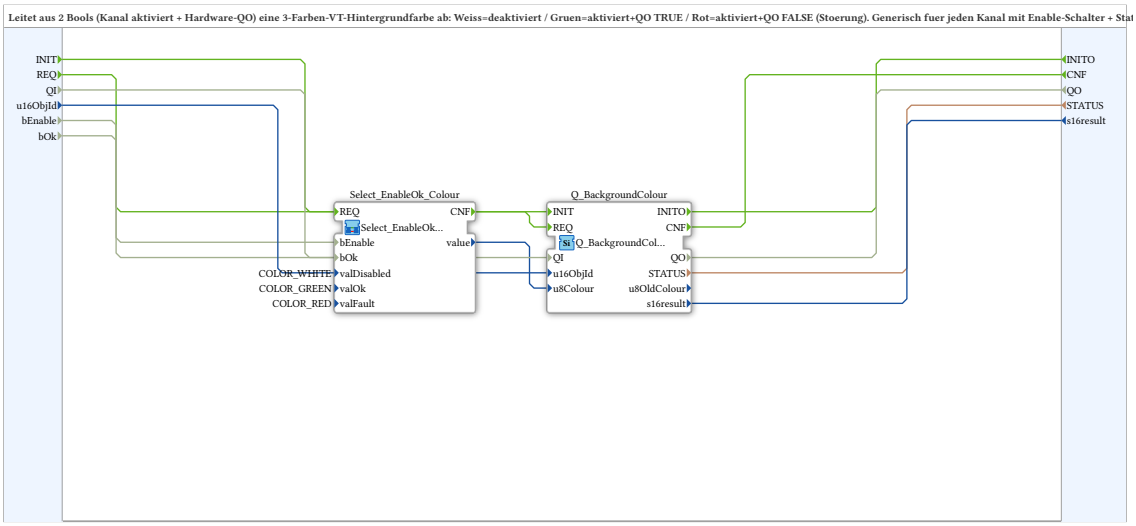
Application Scenarios

- Showing and hiding a VT object (e.g. warning symbol) at a fixed place via an AX state
- Switching between an empty state and a content via object pointer

Conclusion

`ObjectPointer_Swap_AX` translates an AX state into an object pointer switch between `u16ObjId_Show` and `ID_NULL` via `Q_NumericValue`.

Q_BackgroundColour_EnableOk_A



• – ○ • – ○ • – ○ •

Introduction

`Q_BackgroundColour_EnableOk_A` derives a 3-colour VT background colour from two Boolean values (channel enabled and hardware `QO`): **White** = disabled, **Green** = enabled and `QO` TRUE, **Red** = enabled and `QO` FALSE (fault). Generic for any channel with an enable switch and status LED (PWM, PI, ...).



Function Blocks (FBs) Used

Sub-blocks: Q_BackgroundColour_EnableOk_A

- **Type**: SubAppType

- **Internal blocks used**:
  - **Select_EnableOk_Colour**: SubApp `MyLib::sys::Select_EnableOk_a` with `valDisabled=COLOR_WHITE`, `valOk=COLOR_GREEN`, `valFault=COLOR_RED` - selects one of the three colours based on `bEnable` and `bOk`
  - **Q_BackgroundColour**: `isobus::UT::Q::Q_BackgroundColour` - writes the selected colour to the VT object `u16ObjId`
- **Imports**: `COLOR_GREEN`, `COLOR_RED`, `COLOR_WHITE`

Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization request (to `Q_BackgroundColour.INIT`)
- **REQ**: Re-evaluate - fired by the caller once per possible change of `bEnable` or `bOk` (switch toggled or fresh `QO` arrived); both values are supplied together

Event Outputs

- **INITO** (`EInit`): Initialization confirmation (from `Q_BackgroundColour.INITO`)
- **CNF**: Colour sent (passthrough of `Q_BackgroundColour.CNF`)

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **u16ObjId** (`UINT`): Object ID of the status display (`CButton`, background colour only, no text) - set once via parameter at instantiation
- **bEnable** (`BOOL`): Channel enabled (e.g. enable flip-flop `Q`)
- **bOk** (`BOOL`): Hardware confirms active (e.g. `logiBUS_..._IDA.QO`) - only relevant if `bEnable=TRUE`

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully
- **STATUS** (`STRING`): Service status (passthrough of `Q_BackgroundColour.STATUS`)
- **s16result** (`INT`): Return value (passthrough of `Q_BackgroundColour.s16result`)

Program Flow and Connections

- `REQ` ⟶ `Select_EnableOk_Colour.REQ`; `Select_EnableOk_Colour.CNF` ⟶ `Q_BackgroundColour.REQ`; `Q_BackgroundColour.CNF` ⟶ `CNF`
- `bEnable` ⟶ `Select_EnableOk_Colour.bEnable`; `bOk` ⟶ `Select_EnableOk_Colour.bOk`
- `Select_EnableOk_Colour.value` ⟶ `Q_BackgroundColour.u8Colour`
- `u16ObjId` ⟶ `Q_BackgroundColour.u16ObjId`
- `Q_BackgroundColour.STATUS` ⟶ `STATUS`; `Q_BackgroundColour.s16result` ⟶ `s16result`
- `INIT` ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ `INITO`
- `QI` ⟶ `Q_BackgroundColour.QI`; `Q_BackgroundColour.QO` ⟶ `QO`

Technical Details

- The pure "2 bools ⟶ 1 of 3 values" logic is factored out into `MyLib::sys::Select_EnableOk` (generic, not limited to colours); this block merely instantiates it with the three colours as parameters
- Variant with AX adapters: `Q_BackgroundColour_EnableOk_AX`
- Version 1.1 (2026-08-31)

State Overview

bEnable	bOk	Colour
FALSE	any	White (`COLOR_WHITE`)
TRUE	TRUE	Green (`COLOR_GREEN`)
TRUE	FALSE	Red (`COLOR_RED`)

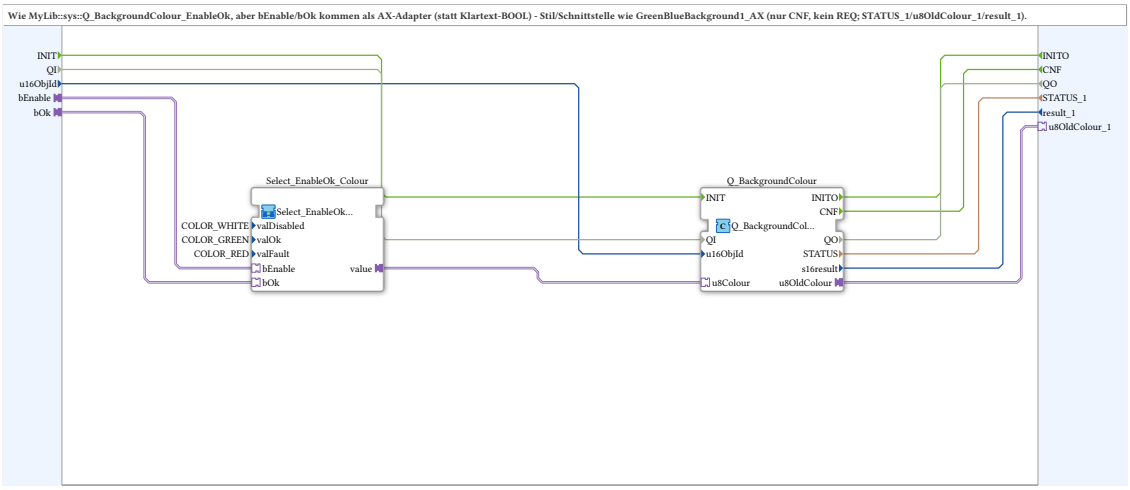
Application Scenarios

- Status display of a channel with an enable switch and hardware feedback (PWM, PI, ...)

Summary

`Q_BackgroundColour_EnableOk_A` sets the background colour of a VT object following the white/green/red scheme from `bEnable` and `bOk`.

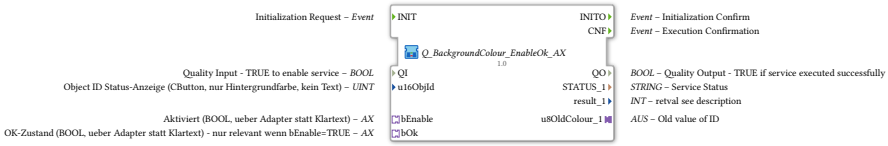
Q_BackgroundColour_EnableOk_AX



• – ○ • – ○ • – ○ •

Introduction

`Q_BackgroundColour_EnableOk_AX` corresponds to `Q_BackgroundColour_EnableOk`, but `bEnable`/`bOk` arrive as AX adapters (instead of plain BOOL) - style/interface matching the existing GreenBlueBackground1_AX/GreenRedBackground1_AX family: only `CNF` (no `REQ`), plus `INIT`/`INITO` and `QI`/`QO`, as well as `STATUS_1`/`u8OldColour_1`/`result_1` as named outputs.



Function blocks used

- **Select_EnableOk_Colour** (SubApp, type `MyLib::sys::Select_EnableOk_AX`): `valDisabled=COLOR_WHITE`, `valOk=COLOR_GREEN`, `valFault=COLOR_RED`.
- **Q_BackgroundColour** (`isobus::UT::Q::Q_BackgroundColour_AUS`): AUS-adapter variant, writes the color to `u16ObjId` and returns `u8OldColour_1` (previous color) as an AUS adapter.

Quality ports (QI/QO)

The block has the data input `QI` (`BOOL`, default `TRUE`, Quality Input - TRUE enables the service) and the data output `QO` (`BOOL`, Quality Output - TRUE if the service was executed successfully). `QI` is connected to `Q_BackgroundColour.QI`, and `Q_BackgroundColour.QO` to `QO`.

In addition, the block has the events `INIT` (input, `EInit`, initialization request) and `INITO` (output, `EInit`, initialization confirmation): `INIT` ⟶ `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` ⟶ `INITO`.

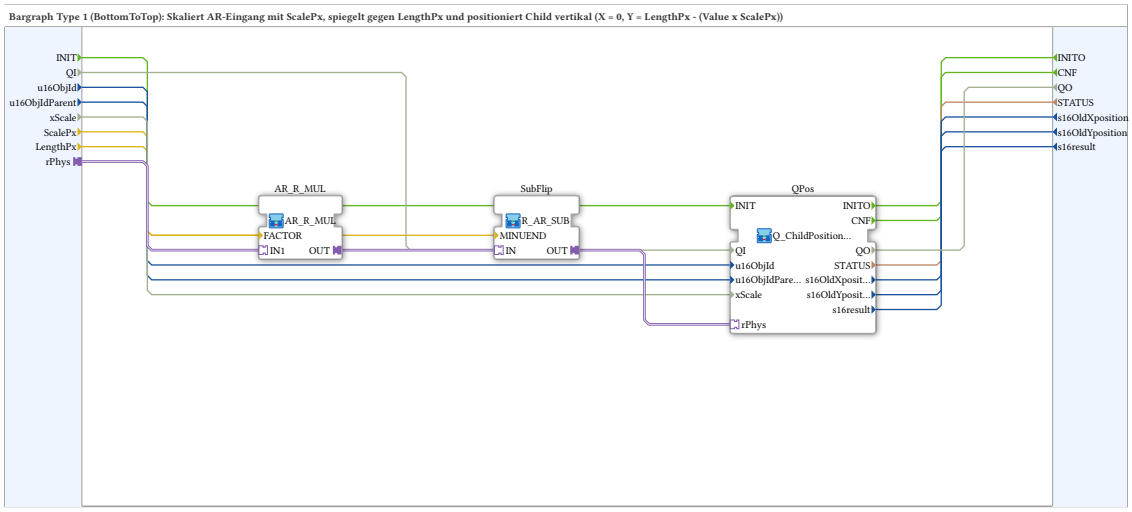
Summary

AX-adapter variant of `Q_BackgroundColour_EnableOk`, matching the existing GreenBlueBackground1_AX/GreenRedBackground1_AX family.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

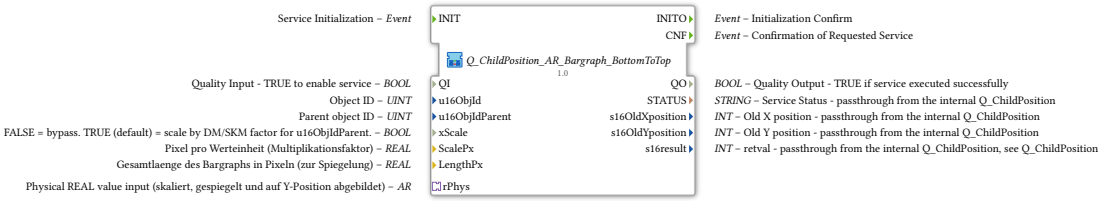
Q_ChildPosition_AR_Bargraph_BottomToTop



• — ○ • — ○ • — ○ •

Introduction

`Q_ChildPosition_AR_Bargraph_BottomToTop` is a **bar graph type 1 (bottom to top)**: the AR input is scaled with `ScalePx`, mirrored against `LengthPx` and output as the vertical position of the child object in the VT (`X = 0`, `Y = LengthPx - (value × ScalePx)`). A larger value thus moves the child upwards.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization (to `QPos.INIT`)

Event Outputs

- **INITO** (`EInit`): Initialization confirmation (from `QPos.INITO`)
- **CNF**: Confirmation of the requested service (from `QPos.CNF`)

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality input – TRUE enables the service (to `QPos.QI`)
- **u16ObjId** (`UINT`, default `ID_NULL`): Object ID of the child object
- **u16ObjIdParent** (`UINT`, default `ID_NULL`): Object ID of the parent object
- **xScale** (`BOOL`, default `TRUE`): `FALSE` = bypass; `TRUE` = scale by the DM/SKM factor for `u16ObjIdParent`
- **ScalePx** (`REAL`, default `1.0`): Pixels per value unit (multiplication factor)
- **LengthPx** (`REAL`, default `100.0`): Total length of the bar graph in pixels (for mirroring)

Data Outputs

- **QO** (`BOOL`): Quality output – TRUE if the service was executed successfully (from `QPos.QO`)
- **STATUS** (`STRING`): Service status (passthrough from the inner `Q_ChildPosition`)
- **s16OldXposition** (`INT`): Old X position (passthrough)
- **s16OldYposition** (`INT`): Old Y position (passthrough)
- **s16result** (`INT`): Return value (passthrough, see `Q_ChildPosition`)

Adapters

- **rPhys** (socket, `adapter::types::unidirectional::AR`): Physical `REAL` input value (scaled, mirrored and mapped to the Y position)

Function Blocks (FBs) Used

- **AR_R_MUL**: SubApp `MyLib::sys::AR_R_MUL` - multiplies the value by `ScalePx` (`FACTOR`)
- **SubFlip**: SubApp `MyLib::sys::R_AR_SUB` - subtracts from the `MINUEND` (`LengthPx`) and thereby mirrors
- **QPos**: SubApp `MyLib::sys::Q_ChildPosition_AR_Vertikal` - sets the vertical child position

Program Flow and Connections

1. `rPhys` ⟶ `AR_R_MUL.IN1`; `ScalePx` ⟶ `AR_R_MUL.FACTOR`.
2. `AR_R_MUL.OUT` ⟶ `SubFlip.IN`; `LengthPx` ⟶ `SubFlip.MINUEND`.
3. `SubFlip.OUT` ⟶ `QPos.rPhys`.
4. `INIT` ⟶ `QPos.INIT`; `QPos.INITO` ⟶ `INITO`; `QPos.CNF` ⟶ `CNF`; `QI` ⟶ `QPos.QI`; `QPos.QO` ⟶ `QO`.
5. `u16ObjId`, `u16ObjIdParent`, `xScale` ⟶ `QPos`; `STATUS`, `s16OldXposition`, `s16OldYposition`, `s16result` from `QPos` to the outside.

Technical Details

- The mirroring is necessary because the VT Y coordinate grows from top to bottom.
- `xScale` defaults to `TRUE` here (whereas to `FALSE` in `Q_ChildPosition_AR_Horizontal`).

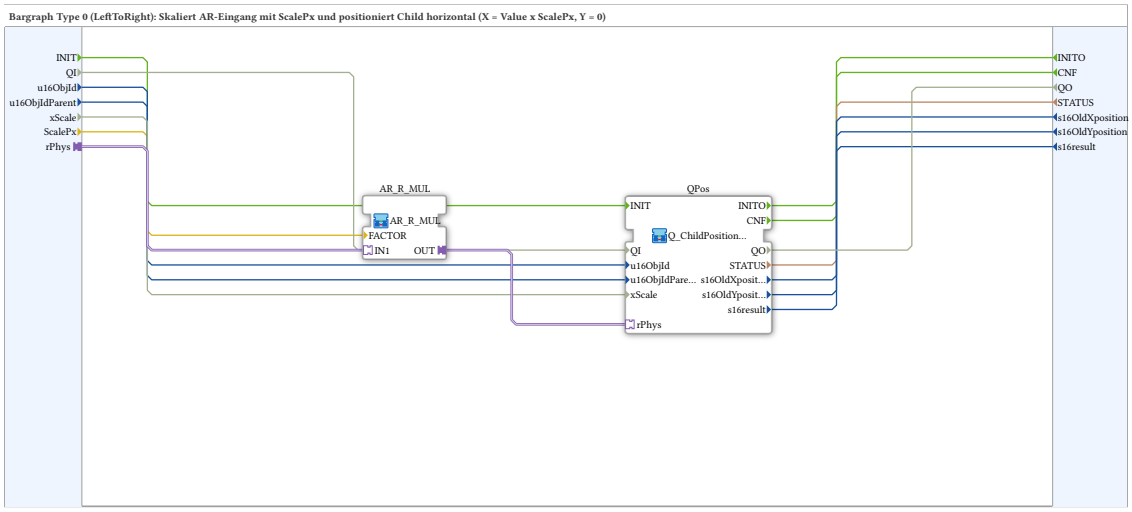
Application Scenarios

- Vertical bar/level display that grows from bottom to top

Summary

`Q_ChildPosition_AR_Bargraph_BottomToTop` maps an AR value to the vertical position of a VT child object via scaling and mirroring.

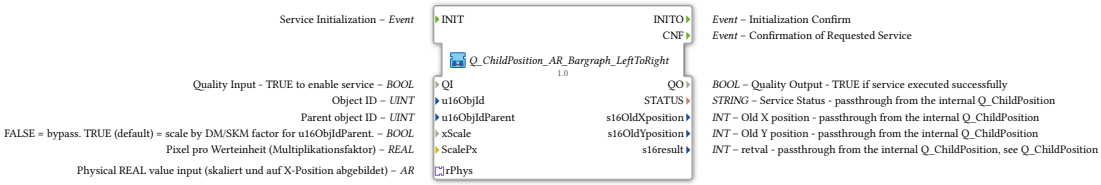
Q_ChildPosition_AR_Bargraph_LeftToRight



• — ○ • — ○ • — ○ •

Introduction

`Q_ChildPosition_AR_Bargraph_LeftToRight` implements a bargraph of type 0 (left to right): the AR input `rPhys` is scaled with `ScalePx` (pixels per value unit) and positions the child object horizontally (X = value × `ScalePx`, Y = 0).



Interface Structure

Event Inputs

- **INIT**: service initialisation (`EInit`)

Event Outputs

- **INITO**: initialisation confirmation (`EInit`)
- **CNF**: confirmation of the requested service

Data Inputs

- **QI** (BOOL): quality input - TRUE enables the service (default `TRUE`), goes to `QPos.QI`
- **u16ObjId** (UINT): object ID of the child object (default `ID_NULL`)
- **u16ObjIdParent** (UINT): object ID of the parent object (default `ID_NULL`)
- **xScale** (BOOL): FALSE = bypass; TRUE (default) = scale by the DM/SKM factor for `u16ObjIdParent`
- **ScalePx** (REAL): pixels per value unit (multiplication factor), default `REAL#1.0`

Data Outputs

- **QO** (BOOL): quality output - TRUE if the service was executed successfully, comes from `QPos.QO`
- **STATUS** (STRING): service status, passed through from the internal `Q_ChildPosition`
- **s16OldXposition** (INT): old X position, passed through
- **s16OldYposition** (INT): old Y position, passed through
- **s16result** (INT): return value, passed through (see `Q_ChildPosition`)

Adapters

- **rPhys** (socket, `adapter::types::unidirectional::AR`): physical REAL value (scaled and mapped to the X position)

Functionality

The network consists of two SubApps:

- `AR_R_MUL` (`MyLib::sys::AR_R_MUL`): `rPhys` ⟶ `AR_R_MUL.IN1`, `ScalePx` ⟶ `AR_R_MUL.FACTOR`; multiplies the AR value by the factor.
- `QPos` (`MyLib::sys::Q_ChildPosition_AR_Horizontal`): `AR_R_MUL.OUT` ⟶ `QPos.rPhys`; sets the X position of the child object via the AR adapter (Y fixed 0).

The events `INIT`, `INITO` and `CNF` as well as `QI`, `u16ObjId`, `u16ObjIdParent`, `xScale` and the outputs `QO` and the four status outputs are passed directly from/to `QPos`.

Technical Details

- Type 0 (LeftToRight) of the bargraph variants: the position grows with the value in X direction, Y stays 0
- `xScale` defaults to TRUE here (scaling by DM/SKM factor), in contrast to `Q_ChildPosition_AR_Vertikal` with default FALSE
- Stateless at SubApp level; all status outputs come from the inner `Q_ChildPosition`

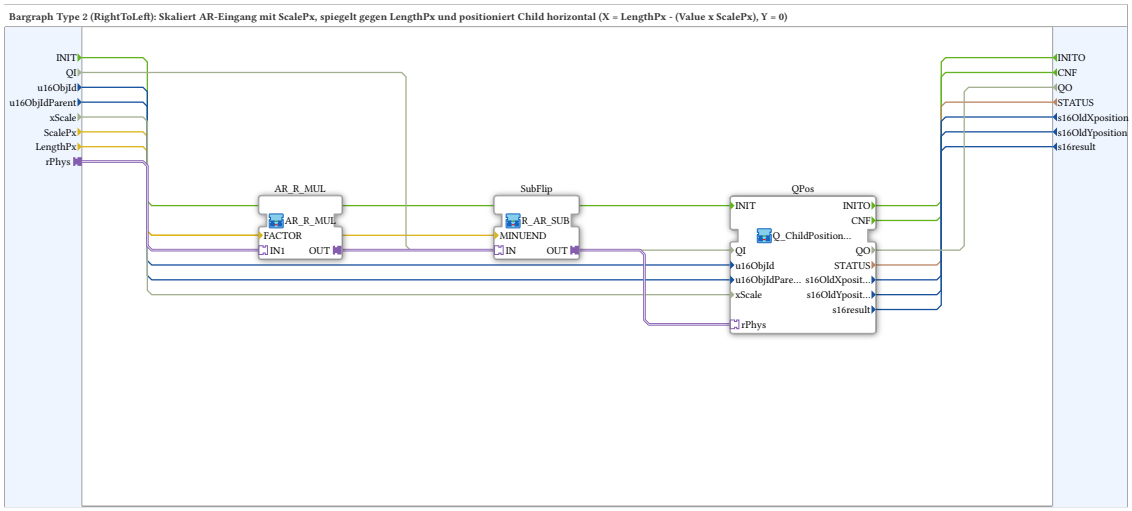
Application Scenarios

- Bar display in which a child object is shifted horizontally proportionally to a measured value

Conclusion

`Q_ChildPosition_AR_Bargraph_LeftToRight` scales an AR value with `ScalePx` and thereby sets the X position of a child object for a left-to-right bargraph.

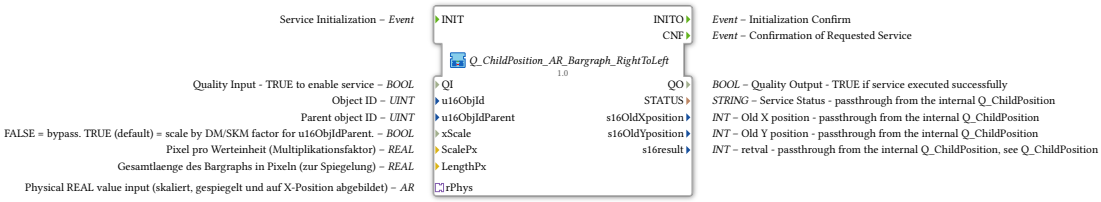
Q_ChildPosition_AR_Bargraph_RightToLeft



• — ○ • — ○ • — ○ •

Introduction

`Q_ChildPosition_AR_Bargraph_RightToLeft` is a bargraph of type 2 (right to left): the `AR` input is scaled with `ScalePx`, mirrored against `LengthPx` and set as the horizontal position of a child object in the VT (`X = LengthPx - (value × ScalePx)`, `Y = 0`). Counterpart of `Q_ChildPosition_AR_Bargraph_LeftToRight`.



Interface Structure

Event Inputs

- **INIT** (`EInit`): service initialisation (passed on to `QPos.INIT`)

Event Outputs

- **INITO** (`EInit`): initialisation confirm (from `QPos.INITO`)
- **CNF**: confirmation of the requested positioning (from `QPos.CNF`)

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service (passed on to `QPos.QI`)
- **u16ObjId** (`UINT`): object ID of the child object (default `ID_NULL`)
- **u16ObjIdParent** (`UINT`): object ID of the parent object (default `ID_NULL`)
- **xScale** (`BOOL`): `FALSE` = bypass; `TRUE` (default) = scale by the DM/SKM factor for `u16ObjIdParent`
- **ScalePx** (`REAL`): pixels per value unit (multiplication factor), default `REAL#1.0`
- **LengthPx** (`REAL`): total length of the bargraph in pixels (for mirroring), default `REAL#100.0`

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully (from `QPos.QO`)
- **STATUS** (`STRING`): service status, passed through from the internal `Q_ChildPosition`
- **s16OldXposition** (`INT`): old X position, passed through
- **s16OldYposition** (`INT`): old Y position, passed through
- **s16result** (`INT`): return value, passed through from the internal `Q_ChildPosition`

Adapters

- **rPhys** (socket, `adapter::types::unidirectional::AR`): physical `REAL` value; scaled, mirrored and mapped to the X position

Function Blocks (FBs) Used

- **`AR_R_MUL`** (SubApp `MyLib::sys::AR_R_MUL`, AR_R_MUL): multiplies the value by `ScalePx`
- **`SubFlip`** (SubApp `MyLib::sys::R_AR_SUB`): subtracts the scaled value from `LengthPx` (`MINUEND = LengthPx`)
- **`QPos`** (SubApp `MyLib::sys::Q_ChildPosition_AR_Horizontal`, Q_ChildPosition_AR_Horizontal): sets the horizontal position of the child object

Functionality

- `rPhys` ⟶ `AR_R_MUL.IN1`; `ScalePx` ⟶ `AR_R_MUL.FACTOR`
- `AR_R_MUL.OUT` ⟶ `SubFlip.IN`; `LengthPx` ⟶ `SubFlip.MINUEND`
- `SubFlip.OUT` ⟶ `QPos.rPhys`
- `INIT` ⟶ `QPos.INIT`; `QPos.INITO` ⟶ `INITO`; `QPos.CNF` ⟶ `CNF`
- `QI` ⟶ `QPos.QI`; `QPos.QO` ⟶ `QO`
- `u16ObjId`, `u16ObjIdParent`, `xScale` ⟶ the like-named inputs of `QPos`
- `QPos.STATUS`, `QPos.s16OldXposition`, `QPos.s16OldYposition`, `QPos.s16result` ⟶ the like-named outputs

Technical Details

- The mirroring (`LengthPx - value × ScalePx`) lets the bar grow from right to left
- All status and position outputs are pure pass-throughs of the internal `Q_ChildPosition`

Application Scenarios

- A horizontal bar display in the VT that grows from right to left (e.g. fill level, utilisation)

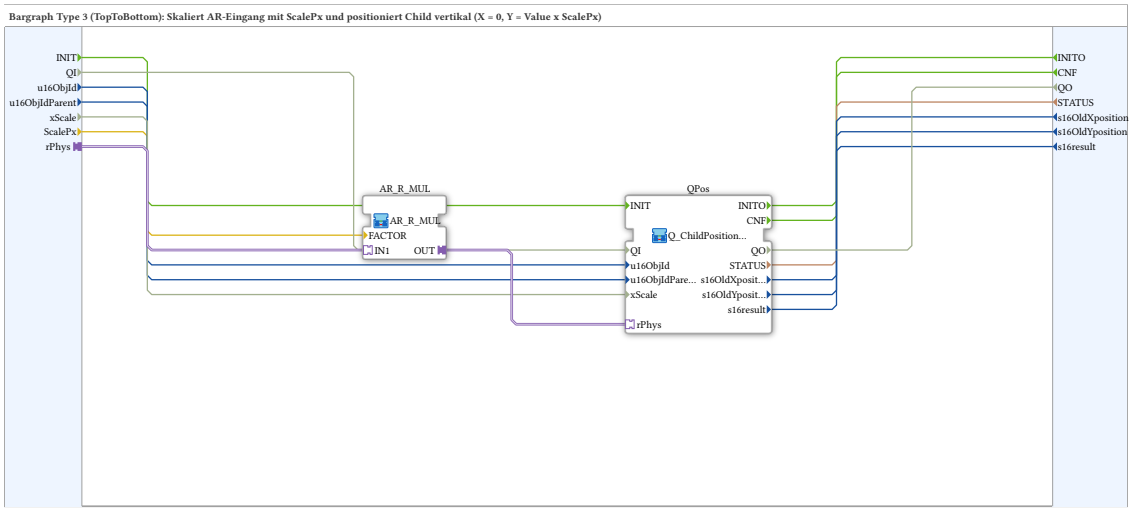
Summary

`Q_ChildPosition_AR_Bargraph_RightToLeft` maps an `AR` value, mirrored, to the X position of a VT child object.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

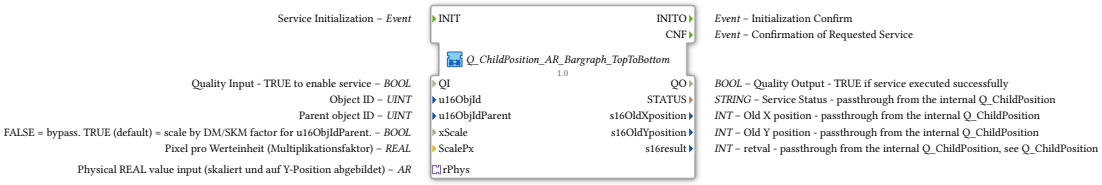
Q_ChildPosition_AR_Bargraph_TopToBottom



• – ○ • – ○ • – ○ •

Introduction

`Q_ChildPosition_AR_Bargraph_TopToBottom` is a **bar graph type 3 (top to bottom)**: the AR input is scaled with `ScalePx` and output as the vertical position of the child object in the VT (`X = 0`, `Y = value × ScalePx`). A larger value thus moves the child downwards. In contrast to `Q_ChildPosition_AR_Bargraph_BottomToTop`, no mirroring takes place.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization (to `QPos.INIT`)

Event Outputs

- **INITO** (`EInit`): Initialization confirmation (from `QPos.INITO`)
- **CNF**: Confirmation of the requested service (from `QPos.CNF`)

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, TRUE enables the service
- **u16ObjId** (`UINT`, default `ID_NULL`): Object ID of the child object
- **u16ObjIdParent** (`UINT`, default `ID_NULL`): Object ID of the parent object
- **xScale** (`BOOL`, default `TRUE`): `FALSE` = bypass; `TRUE` = scale by the DM/SKM factor for `u16ObjIdParent`
- **ScalePx** (`REAL`, default `1.0`): Pixels per value unit (multiplication factor)

Data Outputs

- **QO** (`BOOL`): quality output (from `QPos.QO`)
- **STATUS** (`STRING`): Service status (passthrough from the inner `Q_ChildPosition`)
- **s16OldXposition** (`INT`): Old X position (passthrough)
- **s16OldYposition** (`INT`): Old Y position (passthrough)
- **s16result** (`INT`): Return value (passthrough, see `Q_ChildPosition`)

Adapters

- **rPhys** (socket, `adapter::types::unidirectional::AR`): Physical `REAL` input value (scaled and mapped to the Y position)

Function Blocks (FBs) Used

- **AR_R_MUL**: SubApp `MyLib::sys::AR_R_MUL` - multiplies the value by `ScalePx` (`FACTOR`)
- **QPos**: SubApp `MyLib::sys::Q_ChildPosition_AR_Vertikal` - sets the vertical child position

Program Flow and Connections

1. `rPhys` ⟶ `AR_R_MUL.IN1`; `ScalePx` ⟶ `AR_R_MUL.FACTOR`.
2. `AR_R_MUL.OUT` ⟶ `QPos.rPhys`.
3. `INIT` ⟶ `QPos.INIT`; `QPos.INITO` ⟶ `INITO`; `QPos.CNF` ⟶ `CNF`; `QI` ⟶ `QPos.QI`; `QPos.QO` ⟶ `QO`.
4. `u16ObjId`, `u16ObjIdParent`, `xScale` ⟶ `QPos`; `STATUS`, `s16OldXposition`, `s16OldYposition`, `s16result` from `QPos` to the outside.

Technical Details

- No mirroring: since the VT Y coordinate grows from top to bottom, a larger value corresponds directly to a position further down.
- `xScale` defaults to `TRUE` here.
- Version 1.0 (2026-09-27): `AR_R_MUL` + `Q_ChildPosition_AR_Vertikal`

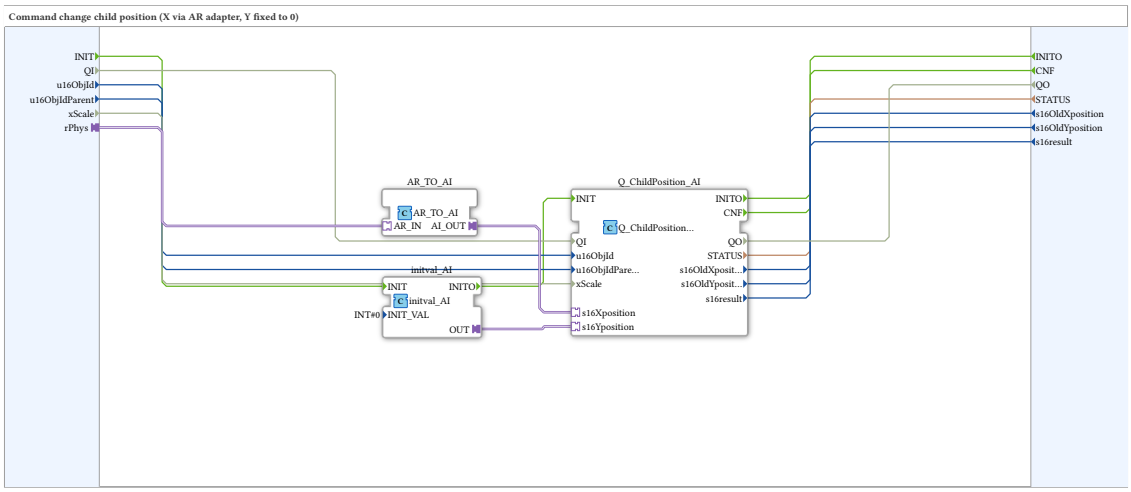
Application Scenarios

- Vertical bar/display that grows from top to bottom

Summary

`Q_ChildPosition_AR_Bargraph_TopToBottom` maps an AR value directly to the vertical position of a VT child object via scaling.

Q_ChildPosition_AR_Horizontal



• – ○ • – ○ • – ○ •

Introduction

`Q_ChildPosition_AR_Horizontal` changes the position of a VT child object (command *Change Child Position*): the **X position** arrives via an AR adapter, the **Y position is fixed at 0**. The `REAL` value is converted to `INT` and mapped to the X position.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization (to `initval_AI.INIT`)

Event Outputs

- **INITO** (`EInit`): Initialization confirmation (from `Q_ChildPosition_AI.INITO`)
- **CNF**: Confirmation of the requested service (from `Q_ChildPosition_AI.CNF`)

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **u16ObjId** (`UINT`, default `ID_NULL`): Object ID of the child object
- **u16ObjIdParent** (`UINT`, default `ID_NULL`): Object ID of the parent object
- **xScale** (`BOOL`, default `FALSE`): `FALSE` = bypass, X/Y position are sent unchanged; `TRUE` = scale by the DM/SKM factor for `u16ObjIdParent`

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully
- **STATUS** (`STRING`): Service status (passthrough from the inner `Q_ChildPosition`)
- **s16OldXposition** (`INT`): Old X position (passthrough)
- **s16OldYposition** (`INT`): Old Y position (passthrough)
- **s16result** (`INT`): Return value (passthrough, see `Q_ChildPosition`)

Adapters

- **rPhys** (socket, `adapter::types::unidirectional::AR`): Physical `REAL` input value (converted to `INT` and mapped to the X position)

Function Blocks (FBs) Used

- **AR_TO_AI**: `adapter::conversion::unidirectional::AR_TO_AI` - converts `REAL` to `INT`
- **initval_AI**: `adapter::types::unidirectional::AI::initval::initval_AI` (`INIT_VAL=0`) - provides the constant Y position 0
- **Q_ChildPosition_AI**: `isobus::UT::Q::Q_ChildPosition_AI` - sends the position change

Program Flow and Connections

1. `rPhys` ⟶ `AR_TO_AI.AR_IN`; `AR_TO_AI.AI_OUT` ⟶ `Q_ChildPosition_AI.s16Xposition`.
2. `initval_AI.OUT` ⟶ `Q_ChildPosition_AI.s16Yposition` (constant 0).
3. `INIT` ⟶ `initval_AI.INIT`; `initval_AI.INITO` ⟶ `Q_ChildPosition_AI.INIT`; `Q_ChildPosition_AI.INITO` ⟶ `INITO`; `Q_ChildPosition_AI.CNF` ⟶ `CNF`.
4. `u16ObjId`, `u16ObjIdParent`, `xScale` ⟶ `Q_ChildPosition_AI`; status outputs from there to the outside.
5. `QI` ⟶ `Q_ChildPosition_AI.QI`; `Q_ChildPosition_AI.QO` ⟶ `QO`

Technical Details

- Initialization runs in series: first `initval_AI`, then `Q_ChildPosition_AI`.
- Unlike `Q_ChildPosition_AR_Bargraph_BottomToTop` there is no scaling/mirroring; `xScale` defaults to `FALSE`.

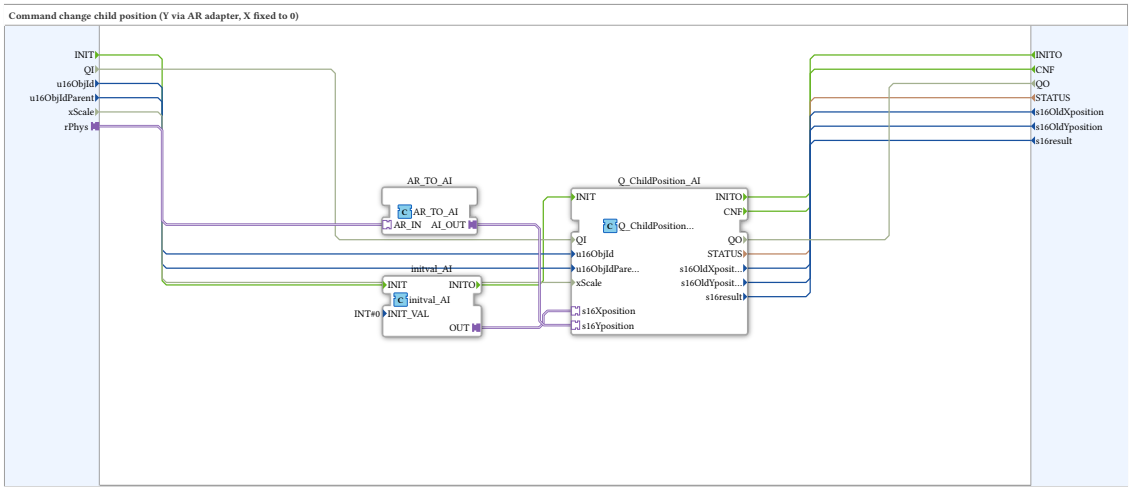
Application Scenarios

- Moving a VT object horizontally (e.g. pointer or bar) depending on a `REAL` value

Summary

`Q_ChildPosition_AR_Horizontal` moves a VT child object horizontally based on an AR value at a fixed Y position of 0.

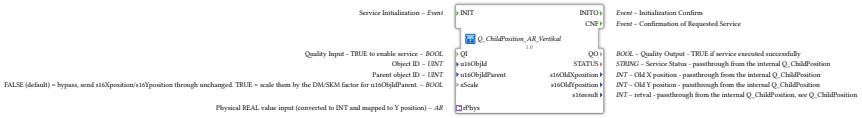
Q_ChildPosition_AR_Vertikal



• — ○ • — ○ • — ○ •

Introduction

`Q_ChildPosition_AR_Vertikal` sets the position of a child object in the VT: the Y position comes via an AR adapter (`rPhys`, converted to INT), the X position is fixed at 0.



Interface Structure

Event Inputs

- **INIT**: service initialisation (`EInit`)

Event Outputs

- **INITO**: initialisation confirmation (`EInit`)
- **CNF**: confirmation of the requested service

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **u16ObjId** (UINT): object ID of the child object (default `ID_NULL`)
- **u16ObjIdParent** (UINT): object ID of the parent object (default `ID_NULL`)
- **xScale** (BOOL): FALSE (default) = bypass, send X/Y position unchanged; TRUE = scale them by the DM/SKM factor for `u16ObjIdParent`

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully
- **STATUS** (STRING): service status, passed through from the internal `Q_ChildPosition`
- **s16OldXposition** (INT): old X position, passed through
- **s16OldYposition** (INT): old Y position, passed through
- **s16result** (INT): return value, passed through (see `Q_ChildPosition`)

Adapters

- **rPhys** (socket, `adapter::types::unidirectional::AR`): physical REAL value (converted to INT and mapped to the Y position)

Functionality

The network consists of three blocks:

- `AR_TO_AI` (`adapter::conversion::unidirectional::AR_TO_AI`): `rPhys` ⟶ `AR_TO_AI.AR_IN`; converts the REAL value into an INT adapter value.
- `initval_AI` (`adapter::types::unidirectional::AI::initval::initval_AI`, `INIT_VAL = 0`): supplies the fixed value 0 for the X position.
- `Q_ChildPosition_AI` (`isobus::UT::Q::Q_ChildPosition_AI`): `initval_AI.OUT` ⟶ `s16Xposition`, `AR_TO_AI.AI_OUT` ⟶ `s16Yposition`.

Initialisation: `INIT` ⟶ `initval_AI.INIT`, `initval_AI.INITO` ⟶ `Q_ChildPosition_AI.INIT`, `Q_ChildPosition_AI.INITO` ⟶ `INITO`. `Q_ChildPosition_AI.CNF` ⟶ `CNF`. `u16ObjId`, `u16ObjIdParent`, `xScale` and the four outputs go directly from/to `Q_ChildPosition_AI`.

`QI` is passed through to `Q_ChildPosition_AI.QI`, and `Q_ChildPosition_AI.QO` to `QO`.

Technical Details

- Initialisation runs in series: first `initval_AI`, then `Q_ChildPosition_AI`
- `xScale` defaults to FALSE (bypass) here; in contrast, `Q_ChildPosition_AR_Bargraph_LeftToRight` defaults to TRUE
- The AR value is not scaled: `rPhys` is only converted to INT and used directly as the Y position

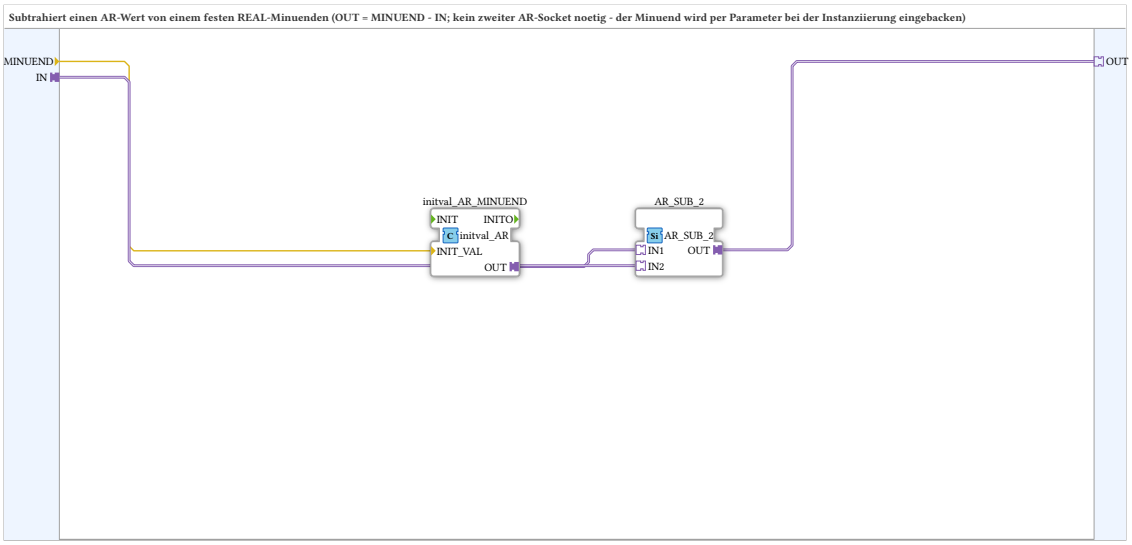
Application Scenarios

- Vertical shifting of a VT object (e.g. marker) proportionally to a measured value, X position fixed at the left

Conclusion

`Q_ChildPosition_AR_Vertikal` maps an AR value to the Y position of a child object via `AR_TO_AI` and holds the X position fixed at 0.

R_AR_SUB



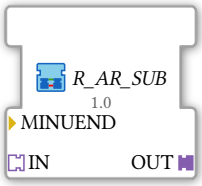
• – ◦ • – ◦ • – ◦ •

Introduction

`R_AR_SUB` subtracts an `AR` value from a fixed `REAL` minuend: `OUT = MINUEND - IN`. A second `AR` socket is not needed, since the minuend is baked in via a parameter at instantiation.

Fester Minuend, ueber Parameter bei der Instanziierung gesetzt – *REAL*

Zu subtrahierender Subtrahend – *AR*



AR – MINUEND - IN

Function Blocks (FBs) Used

Sub-blocks: R_AR_SUB

- **Type**: SubAppType
- **Internal FBs used**:
  - **initval_AR_MINUEND**: `adapter::types::unidirectional::AR::initval::initval_AR` - converts the parameter `MINUEND` into an `AR` signal
  - **AR_SUB_2**: `adapter::iec61131::arithmetic::AR_SUB_2` - forms the difference `IN1 - IN2`

Interface Structure

Event Inputs / Event Outputs

None - events travel via the adapters.

Data Inputs

- **MINUEND** (`REAL`, initial value `0.0`): Fixed minuend, set via parameter at instantiation

Data Outputs

None.

Adapters

- **IN** (socket): `adapter::types::unidirectional::AR` - subtrahend to be subtracted
- **OUT** (plug): `adapter::types::unidirectional::AR` - `MINUEND - IN`

Program Flow and Connections

- `MINUEND` ⟶ `initval_AR_MINUEND.INIT_VAL`
- `initval_AR_MINUEND.OUT` ⟶ `AR_SUB_2.IN1`

- `IN` ⟶ `AR_SUB_2.IN2`
- `AR_SUB_2.OUT` ⟶ `OUT`

Technical Details

- The minuend is fixed as a parameter; no second `AR` input is needed
- Version 1.0 (2026-09-27)

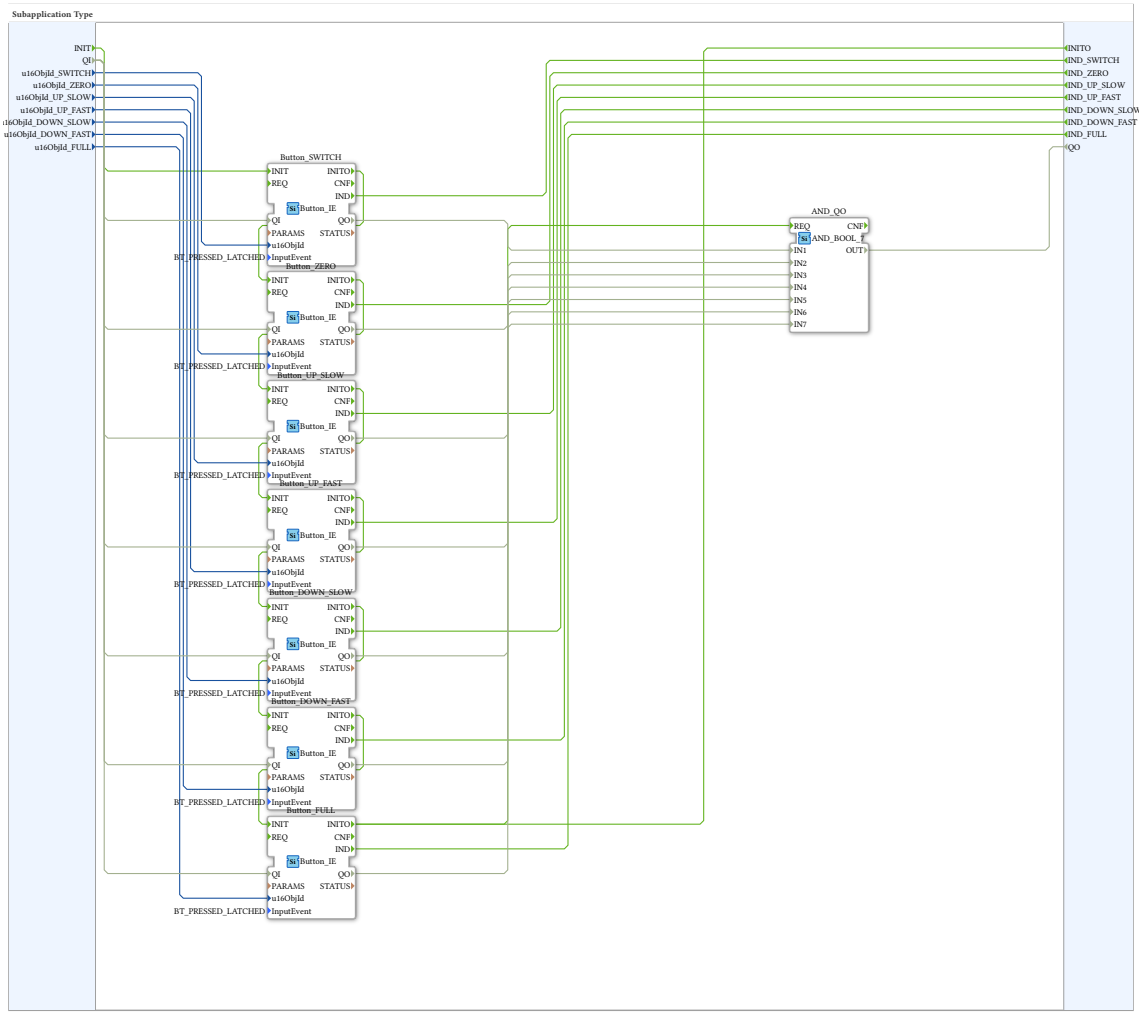
Application Scenarios

- Inversion relative to a fixed quantity, e.g. `100 - value`

Summary

`R_AR_SUB` provides `MINUEND - IN` for an `AR` value with a minuend fixed by parameter.

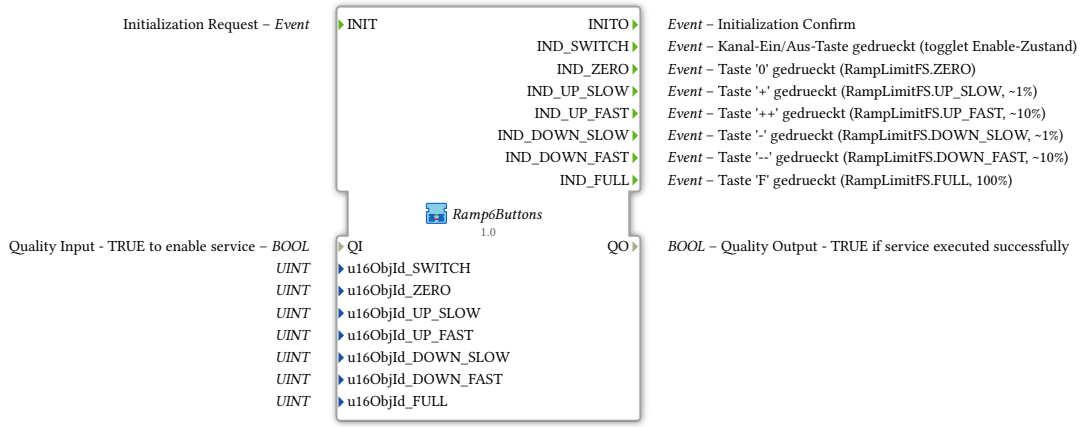
Ramp6Buttons



• — ○ • — ○ • — ○ •

Introduction

`Ramp6Buttons` encapsulates the **7 VT buttons of a PWM channel** — the 6 ramp buttons (`0 -- - + ++ F`) plus the channel enable/disable switch — as its own reusable SubApp. It was extracted from `RampLimitFS_TO_logiBUS_QDA_PWM_OPC` to declutter that block's network and make the button logic independently reusable.



Interface

- **Event input**: `INIT` (`EInit`, Initialization Request)
- **Event outputs**: `INITO` (`EInit`, Initialization Confirm), `IND_SWITCH`, `IND_ZERO`, `IND_UP_SLOW`, `IND_UP_FAST`, `IND_DOWN_SLOW`, `IND_DOWN_FAST`, `IND_FULL`
- **Data inputs**: `QI` (`BOOL`, default `TRUE`, Quality Input) as well as `u16ObjId_SWITCH`, `u16ObjId_ZERO`, `u16ObjId_UP_SLOW`, `u16ObjId_UP_FAST`, `u16ObjId_DOWN_SLOW`, `u16ObjId_DOWN_FAST`, `u16ObjId_FULL`
- **Data output**: `QO` (`BOOL`, Quality Output)

Function Blocks (FBs) Used

Sub-blocks: Ramp6Buttons

- **Type**: SubAppType
- **Internal FBs used**:
  - **Button_SWITCH**, **Button_ZERO**, **Button_UP_SLOW**, **Button_UP_FAST**, **Button_DOWN_SLOW**, **Button_DOWN_FAST**, **Button_FULL**: each `isobus::UT::io::Button::Button_IE`
    - Parameter: `InputEvent=BT_PRESSED_LATCHED` (a latched press/release event, no adapter bridging via `AX_RF_TRIG` needed)
    - Data inputs: `u16ObjId` (object ID of the respective VT button), `QI` (from the SubApp input `QI`)
    - Event output: `IND` (button pressed)
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_7` — ANDs the `QO` outputs of all seven `Button_*` instances into the SubApp output `QO`
- **Functionality**: Seven identical `Button_IE` instances, each bound to its own VT object ID, pass their `IND` event straight through as their own SubApp output event — a pure 1:1 pass-through with no additional logic. In addition, the initialization is chained through all seven instances and the status `QO` is aggregated.

Program Flow and Connections

The same pattern applies to all 7 buttons (`u16ObjId_<NAME>` ⟶ `Button_<NAME>.u16ObjId`, `Button_<NAME>.IND` ⟶ `IND_<NAME>`):

- `u16ObjId_SWITCH` ⟶ `Button_SWITCH.u16ObjId`; `Button_SWITCH.IND` ⟶ `IND_SWITCH` (toggles channel enable)
- `u16ObjId_ZERO` ⟶ `Button_ZERO.u16ObjId`; `Button_ZERO.IND` ⟶ `IND_ZERO` (`RampLimitFS.ZERO`)
- `u16ObjId_UP_SLOW` ⟶ `Button_UP_SLOW.u16ObjId`; `Button_UP_SLOW.IND` ⟶ `IND_UP_SLOW` (`RampLimitFS.UP_SLOW`, 1 %)
- `u16ObjId_UP_FAST` ⟶ `Button_UP_FAST.u16ObjId`; `Button_UP_FAST.IND` ⟶ `IND_UP_FAST` (`RampLimitFS.UP_FAST`, 10 %)
- `u16ObjId_DOWN_SLOW` ⟶ `Button_DOWN_SLOW.u16ObjId`; `Button_DOWN_SLOW.IND` ⟶ `IND_DOWN_SLOW` (`RampLimitFS.DOWN_SLOW`, 1 %)
- `u16ObjId_DOWN_FAST` ⟶ `Button_DOWN_FAST.u16ObjId`; `Button_DOWN_FAST.IND` ⟶ `IND_DOWN_FAST` (`RampLimitFS.DOWN_FAST`, 10 %)
- `u16ObjId_FULL` ⟶ `Button_FULL.u16ObjId`; `Button_FULL.IND` ⟶ `IND_FULL` (`RampLimitFS.FULL`, 100 %)

In addition:

- Initialization chain: `INIT` ⟶ `Button_SWITCH.INIT`, `Button_SWITCH.INITO` ⟶ `Button_ZERO.INIT`, `Button_ZERO.INITO` ⟶ `Button_UP_SLOW.INIT`, `Button_UP_SLOW.INITO` ⟶ `Button_UP_FAST.INIT`, `Button_UP_FAST.INITO` ⟶ `Button_DOWN_SLOW.INIT`, `Button_DOWN_SLOW.INITO` ⟶ `Button_DOWN_FAST.INIT`, `Button_DOWN_FAST.INITO` ⟶ `Button_FULL.INIT`, `Button_FULL.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`
- `QI` ⟶ `QI` of all seven `Button_*` instances
- `Button_SWITCH.QO`, `Button_ZERO.QO`, `Button_UP_SLOW.QO`, `Button_UP_FAST.QO`, `Button_DOWN_SLOW.QO`, `Button_DOWN_FAST.QO`, `Button_FULL.QO` ⟶ `AND_QO.IN1..IN7`; `AND_QO.OUT` ⟶ `QO`

The calling block wires `IND_ZERO`/`IND_UP_SLOW`/`IND_UP_FAST`/`IND_DOWN_SLOW`/`IND_DOWN_FAST`/`IND_FULL` directly to the like-named event inputs of `RampLimitFS`, and `IND_SWITCH` to the channel enable flip-flop.

Technical Details

- **`BT_PRESSED_LATCHED` instead of `Button_IXA+AX_RF_TRIG`**: a simpler pattern with no adapter bridging, since `Button_IE` outputs the press event directly as `IND`.
- **Event order aligned with `RampLimitFS`**: The output events (`SWITCH, ZERO, UP_SLOW, UP_FAST, DOWN_SLOW, DOWN_FAST, FULL`) deliberately follow the order in which `RampLimitFS` declares its own event inputs, to make wiring easier.

Application Scenarios

- Any exercise with a ramped setpoint (`RampLimitFS` or similar) that should be operated via 6 buttons plus one enable switch.

Summary

`Ramp6Buttons` bundles seven identical `Button_IE` instances into a single reusable SubApp, keeping the button wiring out of the actual PWM channel block.

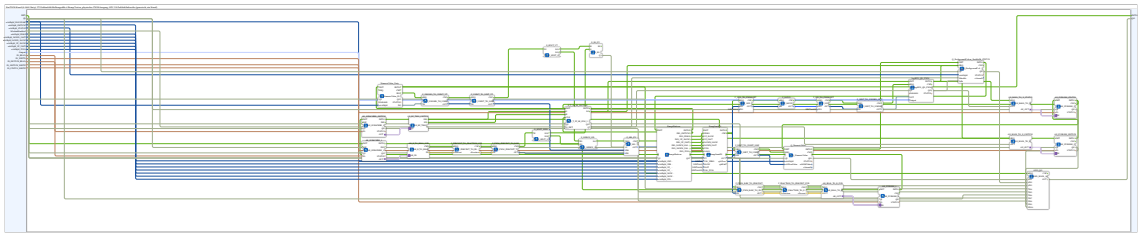
🛠️ Related Exercises

- RampLimitFS_TO_logiBUS_QDA_PWM_OPC
- InputOutputTesterButton_PWM_OPC_UA

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

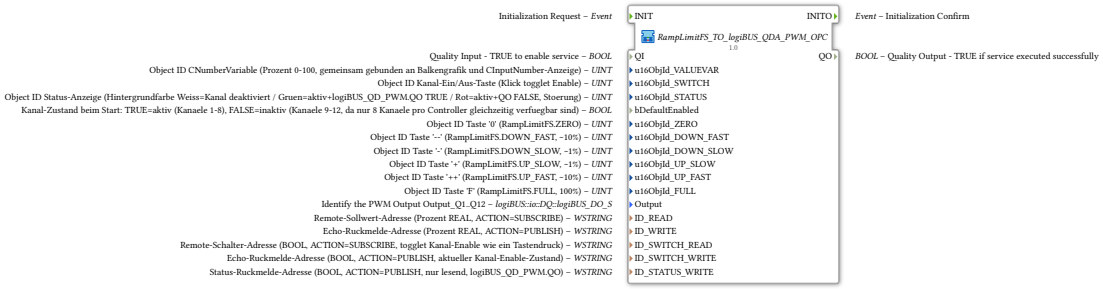
RampLimitFS_TO_logiBUS_QDA_PWM_OPC



• – ◦ • – ◦ • – ◦ •

Introduction

`RampLimitFS_TO_logiBUS_QDA_PWM_OPC` is the reusable block for **a single PWM output channel (0–100 % duty cycle)** with a VT number field, bar graph, 6 ramp buttons, channel enable/disable switch, 3-color status indicator, and bidirectional OPC-UA connectivity. It is instantiated 12× with different parameters in `InputOutputTesterButton_PWM_OPC_UA` and is the PWM counterpart of the simpler, purely digital `RampLimitFS_TO_logiBUS_QDA_OPC`.



Function Blocks (FBs) Used

Sub-blocks: RampLimitFS_TO_logiBUS_QDA_PWM_OPC

- **Type**: SubAppType
- **Internal FBs used**:
  - **RampLimitFS**: `eclipse4diac::signalprocessing::RampLimitFS`
    - Parameters: `VAL_ZERO=DINT#0`, `SLOW=DINT#643` (1 %), `FAST=DINT#6426` (10 %), `VAL_FULL=DINT#64255`
    - Data input: `PV` (setpoint mux output), event inputs: `ZERO/UP_SLOW/UP_FAST/DOWN_SLOW/DOWN_FAST/FULL/LOAD`
    - Data output: `OUT` (0–64255)
  - **Ramp6Buttons** (SubApp, `MyLib::sys`): encapsulates the 7 VT buttons (6 ramp buttons + channel switch), see its own documentation
  - **F_PWM_PERCENT_TO_RAW** / **F_PWM_RAW_TO_PERCENT** (SubApp, `MyLib::sys`): conversion between fraction 0.0-1.0 and fieldbus raw value, see F_PWM_PERCENT_TO_RAW / F_PWM_RAW_TO_PERCENT
  - **F_PERCENT_TO_FRACTION_SUB** / **F_FRACTION_TO_PERCENT_PUB**: `logiBUS::signalprocessing::fieldbus::F_PERCENT_TO_FRACTION/F_FRACTION_TO_PERCENT` — percent 0-100 ↔ fraction 0.0-1.0
  - **E_RS_PV** (`iec61499::events::E_RS`) + **F_SEL_PV** (`iec61131::selection::F_SEL`): two-source merge (VT number field and web setpoint) onto the single data input `RampLimitFS.PV`, since the 4diac IDE does not allow multiple connections into one data input
  - **E_T_FF_SR_SYM_INIT** (`E_T_FF_SR_SWITCH`): channel enable state, set/reset by the VT button (`CLK`, a real toggle) and by web writes (`S/R`, an edge change detected via `AX_RF_TRIG`, not a toggle)
  - **F_MUL_TO_PWM13BIT/F_DIV_TO_PWM13BIT** (`iec61131::arithmetic::F_MUL/F_DIV`): convert `RampLimitFS.OUT` (0-64255) via `×8191 ÷ 64255` into the 13-bit raw value (0-8191) for `logiBUS_QD_PWM.OUT`
  - **logiBUS_QD_PWM**: physical PWM output (the `Output` parameter identifies `Output_Q1..Q12`)
  - **F_SEL_OK_FAULT/F_SEL_STATUS** (`iec61131::selection::F_SEL`) + **Q_BackgroundColour**: 3-color status logic (white=disabled, green=active+`QO`=TRUE, red=active+`QO`=FALSE)
  - **AR_SUBSCRIBE_1/AR_PUBLISH_1**, **AX_SUBSCRIBE_1/AX_PUBLISH_1** (×2): OPC-UA adapters for setpoint (REAL), switch (BOOL), and status (BOOL)
  - **AND_QO** (`iec61131::booleanOperators::AND_BOOL_10`): combines ten `QO` sources into the SubApp output `QO`
- **Functionality**: Two independent setpoint sources — the VT number field/ramp buttons and a web percent value received via OPC-UA subscribe — are muxed onto the same `RampLimitFS` ramp block via `E_RS+F_SEL` (whichever source was most recently active wins). The ramp output (0-64255) feeds three destinations: the 13-bit raw value going to the physical `logiBUS_QD_PWM` output, a percent value going back to the VT number field/bar graph, and a percent value published via OPC-UA to the web client. In parallel, a VT button press or a web write toggles the channel enable state (`E_T_FF_SR_SWITCH`), which arms/disarms the physical output via `logiBUS_QD_PWM.QI` and feeds into the 3-color status display.

Program Flow and Connections

1. **VT setpoint path**: `NumericValue_Duty.IND` (number field changed) ⟶ `F_DWORD_TO_UDINT_VT` ⟶ `F_UDINT_TO_DINT_VT` ⟶ `E_SPLIT_VT` ⟶ on one branch `E_RS_PV.R` (reset: "web takes precedence until VT becomes active again" — see below), on the other `E_MERGE_SEL` ⟶ `F_SEL_PV.REQ`.
2. **Web setpoint path**: `AR_SUBSCRIBE_1` (OPC-UA subscribe, percent REAL) ⟶ `AR_R_TO_REAL_SUB` ⟶ `F_PERCENT_TO_FRACTION_SUB` (percent ⟶ fraction) ⟶ `F_PWM_PERCENT_TO_RAW` (fraction ⟶ fieldbus

raw value) → E_SPLIT_WEB → on one branch E_RS_PV.S (set: web source active), on the other E_MERGE_SEL → F_SEL_PV.REQ.

3. **Mux and ramp**: E_RS_PV.Q drives F_SEL_PV.G (0=VT value IN0, 1=web value IN1); F_SEL_PV.OUT loads RampLimitFS.PV via the LOAD event. Ramp6Buttons additionally supplies the 6 ramp events directly to RampLimitFS (ZERO/UP_SLOW/UP_FAST/DOWN_SLOW/DOWN_FAST/FULL).
4. **Physical output**: RampLimitFS.OUT (0-64255) → F_MUL_TO_PWM13BIT (×8191) → F_DIV_TO_PWM13BIT (÷64255) → F_DINT_TO_DWORD_OUT → logiBUS_QD_PWM.OUT.
5. **VT display**: RampLimitFS.OUT → F_DINT_TO_UDINT_DISP → Q_NumericValue.REQ (updates the number field + the bar graph sharing the same variable).
6. **OPC-UA publish**: RampLimitFS.OUT → F_PWM_RAW_TO_PERCENT (raw value → fraction) → F_FRACTION_TO_PERCENT_PUB (fraction → percent) → AR_REAL_TO_R_PUB → AR_PUBLISH_1.
7. **Channel switch**: Ramp6Buttons.IND_SWITCH (VT button) → E_T_FF_SR_SWITCH.CLK (toggles). AX_SUBSCRIBE_SWITCH (web write, BOOL) → AX_RF_TRIG_SWITCH detects a real edge change → ER→S / EF→R (sets instead of toggling, so two identical web writes cannot accidentally invert the state). bDefaultEnabled feeds E_T_FF_SR_SWITCH.Q_INIT for the initial state.
8. **Status chain**: E_T_FF_SR_SWITCH.Q → logiBUS_QD_PWM.QI (armed/disarmed) and → F_SEL_STATUS.G. logiBUS_QD_PWM.INITO/.QO feed F_SEL_OK_FAULT (red/green based on QO), whose result goes through F_SEL_STATUS (white if disabled) to Q_BackgroundColour_STATUS. In addition, AX_BOOL_TO_X_SWITCH/AX_PUBLISH_SWITCH and AX_BOOL_TO_X_STATUS/AX_PUBLISH_STATUS mirror the enable state and QO to the web client via OPC-UA publish.
9. **Boot sequence**: The SubApp has the event input INIT and the event output INITO (both new). INIT → AR_SUBSCRIBE_1.INIT; the five OPC-UA adapters are initialized in a strict chain (AR_SUBSCRIBE_1.INITO → AR_PUBLISH_1.INIT → AX_SUBSCRIBE_SWITCH.INIT → AX_PUBLISH_SWITCH.INIT → AX_PUBLISH_STATUS.INIT); only afterward does AX_PUBLISH_STATUS.INITO fire E_T_FF_SR_SWITCH.INIT, so the first published enable state is not dropped by an adapter that is not yet ready. Also starting from AX_PUBLISH_STATUS.INITO, the VT initialization chain runs Q_NumericValue.INIT → Q_BackgroundColour_EnableOk_STATUS.INIT → Ramp6Buttons.INIT → NumericValue_Duty.INIT; NumericValue_Duty.INITO → INITO and → AND_QO.REQ.
10. **QI/QO pass-through**: QI (BOOL, default TRUE) is distributed to ten instances: Q_NumericValue, NumericValue_Duty, AR_SUBSCRIBE_1, AR_PUBLISH_1, E_T_FF_SR_SWITCH, AX_SUBSCRIBE_SWITCH, AX_PUBLISH_SWITCH, AX_PUBLISH_STATUS, Q_BackgroundColour_EnableOk_STATUS and Ramp6Buttons (the internal FBs no longer carry a fixed QI=TRUE). Their QO outputs — Q_NumericValue (IN1), Q_BackgroundColour_EnableOk_STATUS (IN2), Ramp6Buttons (IN3), NumericValue_Duty (IN4), AR_SUBSCRIBE_1 (IN5), AR_PUBLISH_1 (IN6), E_T_FF_SR_SWITCH (IN7), AX_SUBSCRIBE_SWITCH (IN8), AX_PUBLISH_SWITCH (IN9), AX_PUBLISH_STATUS (IN10) — feed AND_QO (AND_BOOL_10), and AND_QO.OUT drives QO (BOOL, quality output). This QO is independent of the logiBUS_QD_PWM.QO used in the status chain.

Technical Details

- **Value range**: Consistently uses the SAE J1939/ISO 11783 fieldbus convention VALID_SIGNAL_W (0-64255) internally, not percent — avoids rounding errors and stays consistent with RampLimitFS.
- **13-bit PWM raw value**: logiBUS_QD_PWM.OUT was empirically confirmed to expect 0-8191 (not 0-64255) — hence the additional ×8191÷64255 conversion chain.
- **Multi-connection workaround**: E_RS+F_SEL instead of a (disallowed in the 4diac IDE) duplicate data connection into RampLimitFS.PV.
- **No PERMIT gating**: The channel switch wires QI directly as a data connection (the pattern from Uebung_094a), no E_PERMIT blocks.
- **Set/reset instead of toggle for web writes**: AX_RF_TRIG + E_T_FF_SR_SYM_INIT instead of a pure toggle flip-flop, so repeated writes with the same value do not randomly invert the state.

Application Scenarios

- Training example for analog (PWM) outputs with remote control via OPC-UA/web, alongside classic VT operation.
- Template for any further multi-channel analog output exercise with setpoint ramping and per-channel enable.

Comparison with Similar Blocks

Compared to the simpler, purely digital counterpart `RampLimitFS_TO_logiBUS_QDA_OPC`, `RampLimitFS_TO_logiBUS_QDA_PWM_OPC` differs in its analog setpoint (a ramp instead of a bit), the additional scaling chain (percent ↔ fraction ↔ fieldbus raw value ↔ 13-bit PWM), and the 3-color instead of 2-color status logic.

Summary

`RampLimitFS_TO_logiBUS_QDA_PWM_OPC` encapsulates a complete PWM channel — setpoint ramping, two-source mux between VT and web, scaling down to the physical 13-bit PWM output, a channel enable switch with robust set/reset behavior, and 3-color status feedback — in a single block that is reused 12 times.

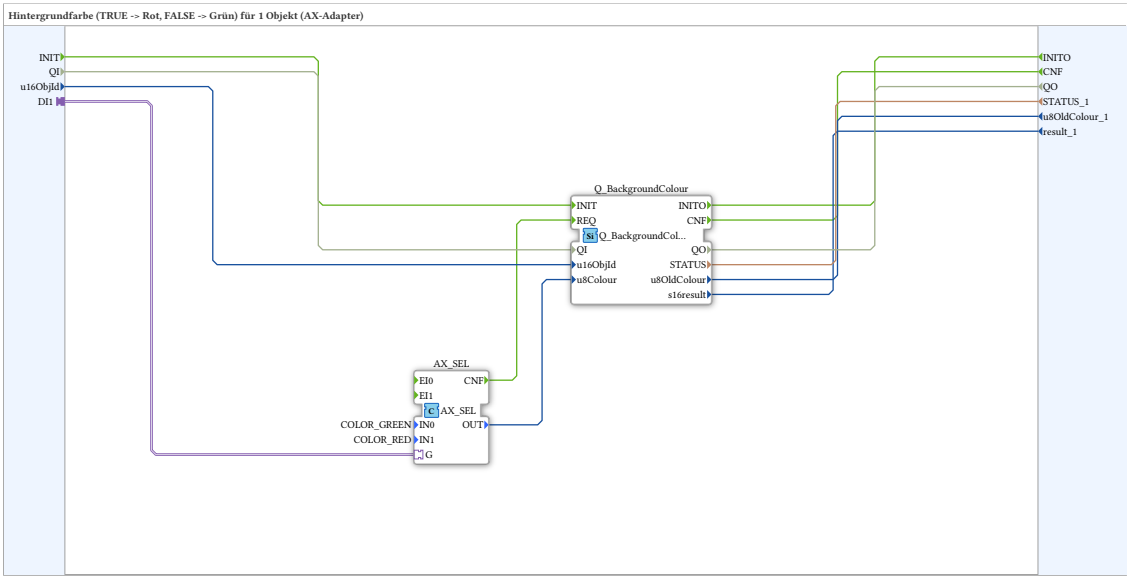
🛠️ Related Exercises

- InputOutputTesterButton_PWM_OPC_UA

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1_AX

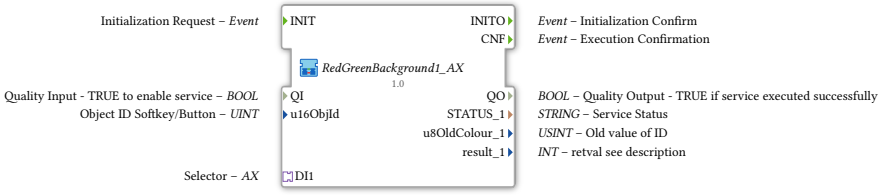


• – ○ • – ○ • – ○ •

Introduction

`RedGreenBackground1_AX` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (event output, `EInit`).

- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColour.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColour.QO`.

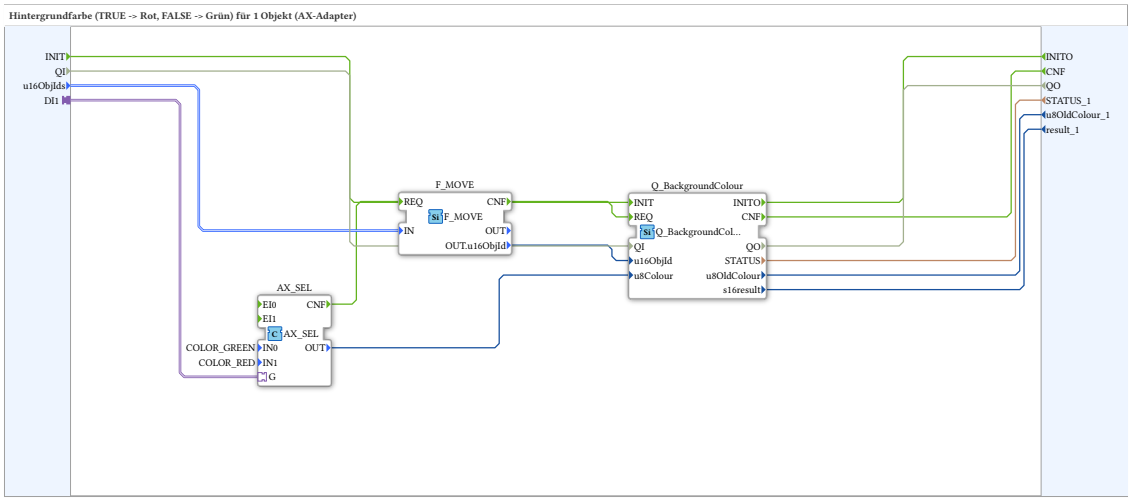
Summary

One of many variants in the background color block family: color pair Rot/Grün, 1 object, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1_AXS

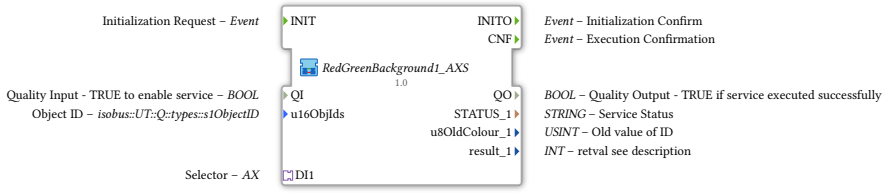


• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground1_AXS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColour.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColour.QO`.

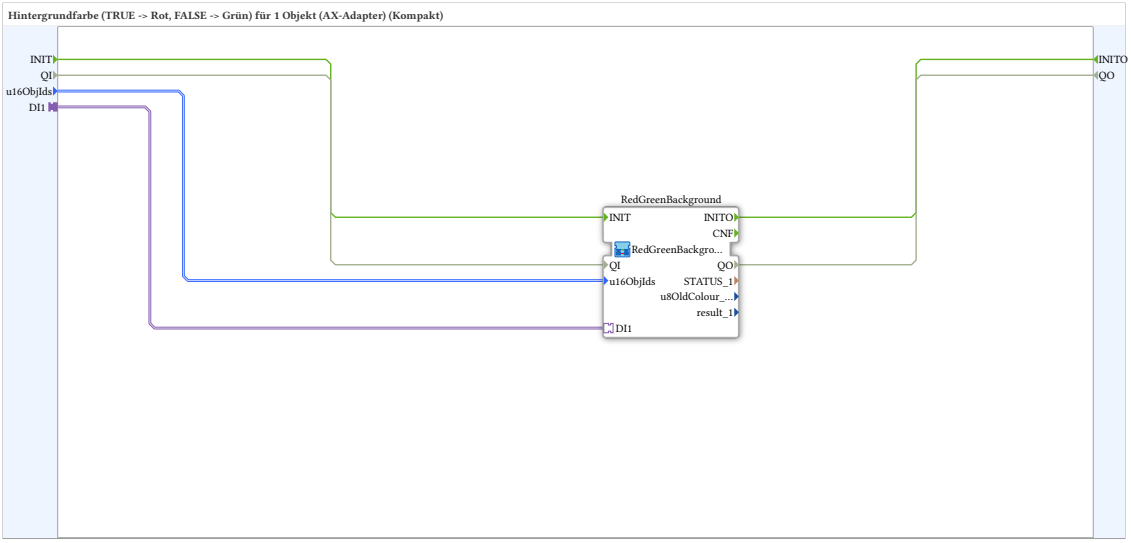
Summary

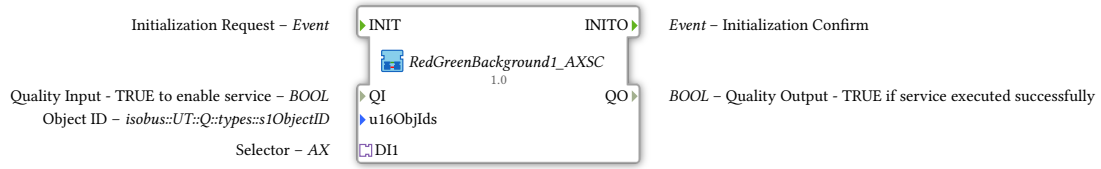
One of many variants in the background color block family: color pair Rot/Grün, 1 object, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1_AXSC





• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground1_AXSC` is a **compact wrapper** around `RedGreenBackground1_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

In addition, initialization and quality are passed through: `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`, `QI` (initial value `TRUE`) ⟶ `RedGreenBackground.QI` and `RedGreenBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

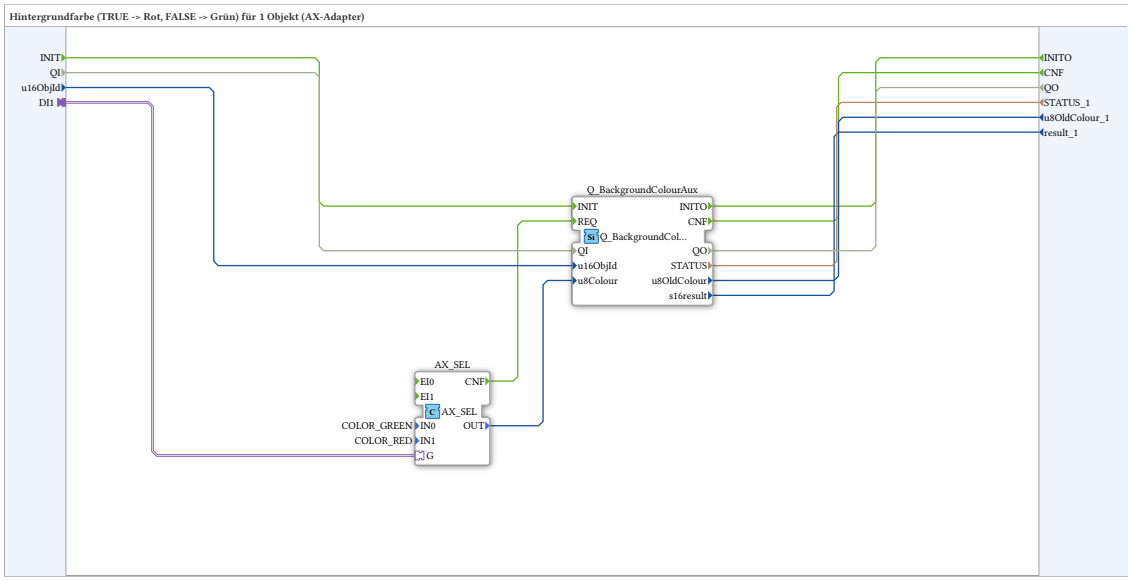
Summary

A pure pass-through to `RedGreenBackground1_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1_aux_AX

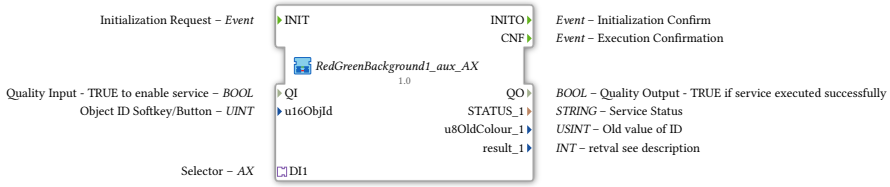


• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground1_aux_AX` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColourAux.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColourAux.QO`.

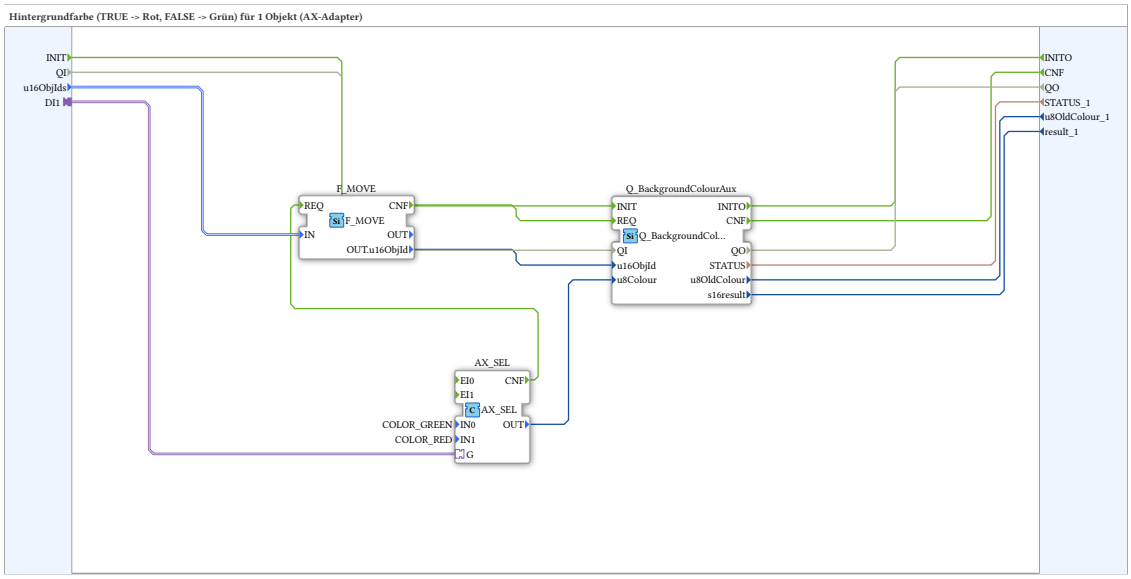
Summary

One of many variants in the background color block family: color pair Rot/Grün, 1 object, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1_aux_AXS

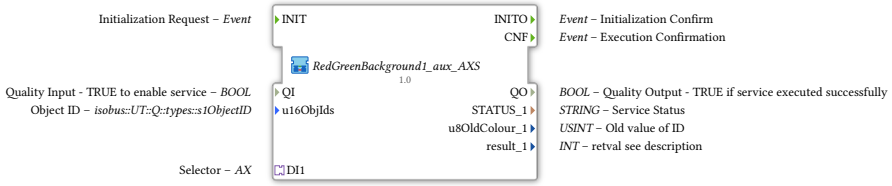


• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground1_aux_AXS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColourAux.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColourAux.QO`.

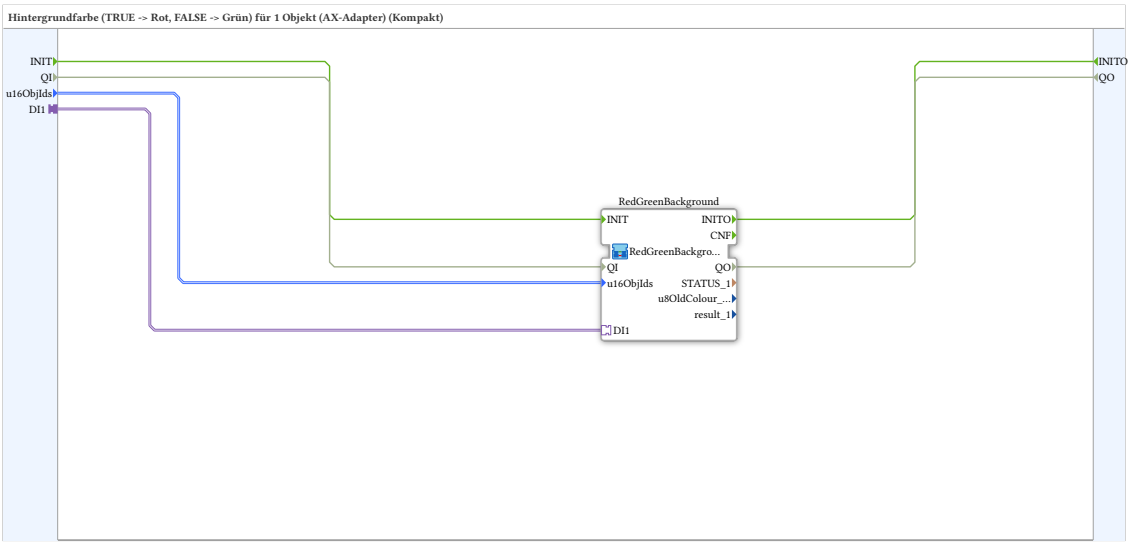
Summary

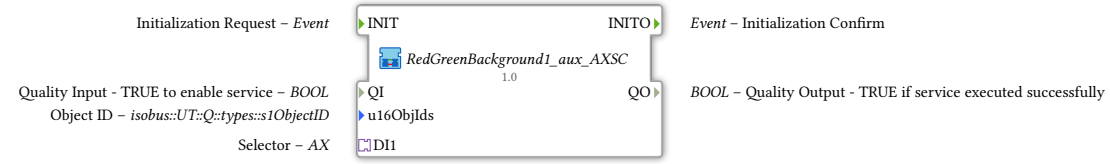
One of many variants in the background color block family: color pair Rot/Grün, 1 object, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1_aux_AXSC





• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground1_aux_AXSC` is a **compact wrapper** around `RedGreenBackground1_aux_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `RedGreenBackground`: `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground.QI`, `RedGreenBackground.QO` ⟶ `QO`.

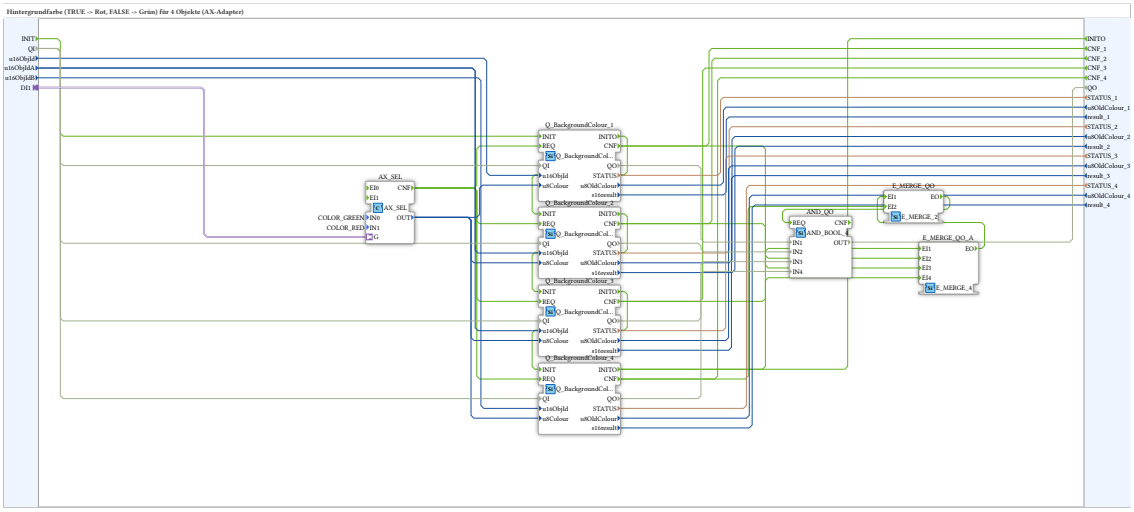
Summary

A pure pass-through to `RedGreenBackground1_aux_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground4_AX



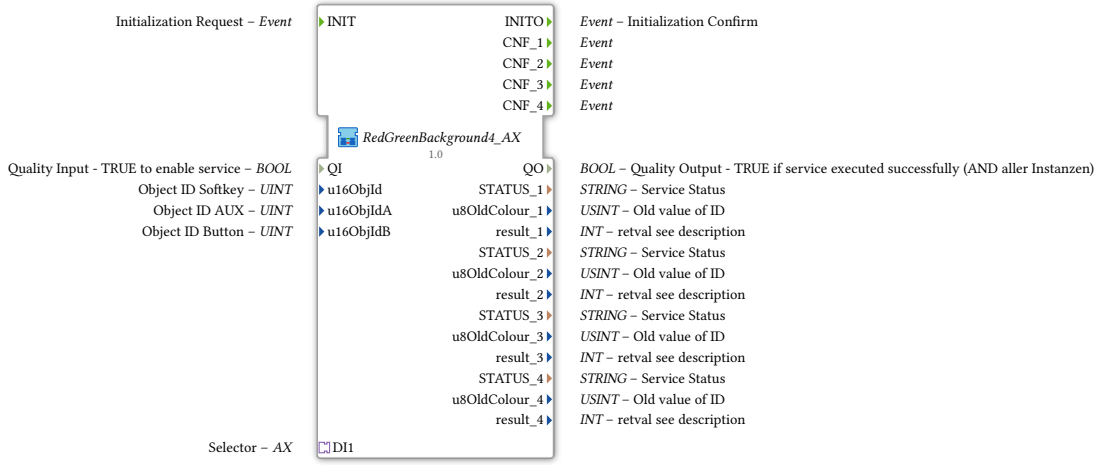
• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground4_AX` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA, u16ObjIdB`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` ⟶ `INITO`.

- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`, `Q_BackgroundColour_4`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO_A` (`iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_4.INITO` and the `CNF` events of instances 1 to 3; `E_MERGE_QO` (`iec61499::events::E_MERGE_2`) merges its output `E_MERGE_QO_A.EO` with `Q_BackgroundColour_4.CNF` and thereby triggers `AND_QO.REQ`.

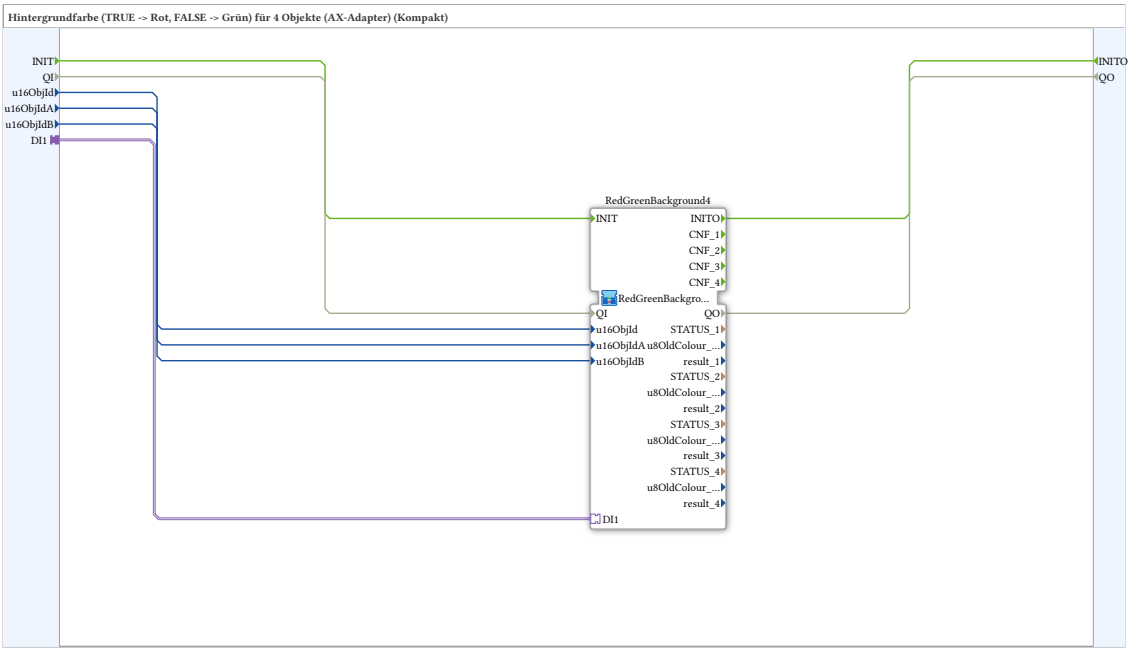
Summary

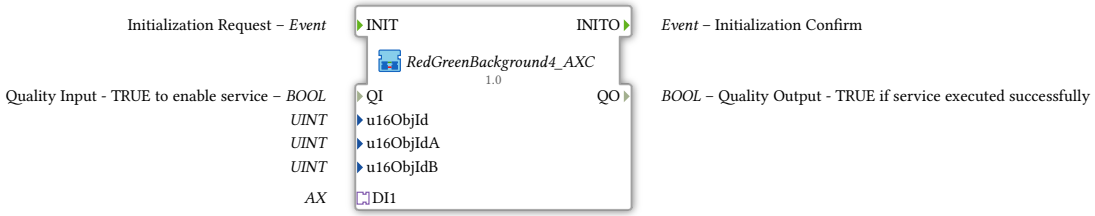
One of many variants in the background color block family: color pair Rot/Grün, 4 objects, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground4_AXC





• – ◦ • – ◦ • – ◦ •

Introduction

`RedGreenBackground4_AXC` is a **compact wrapper** around `RedGreenBackground4_AX` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `RedGreenBackground4`: `INIT` ⟶

`RedGreenBackground4.INIT`, `RedGreenBackground4.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground4.QI`, `RedGreenBackground4.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

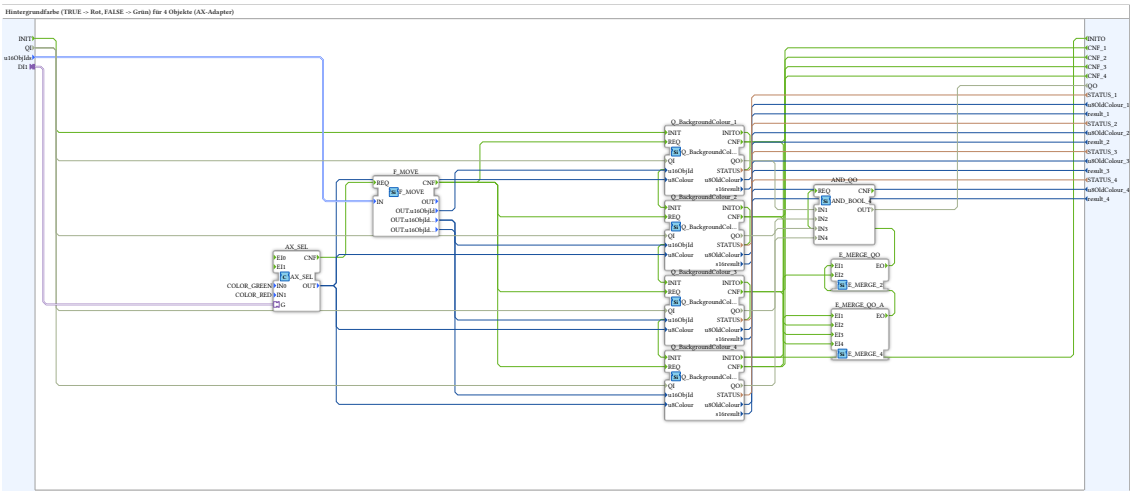
Summary

A pure pass-through to `RedGreenBackground4_AX` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground4_AXS



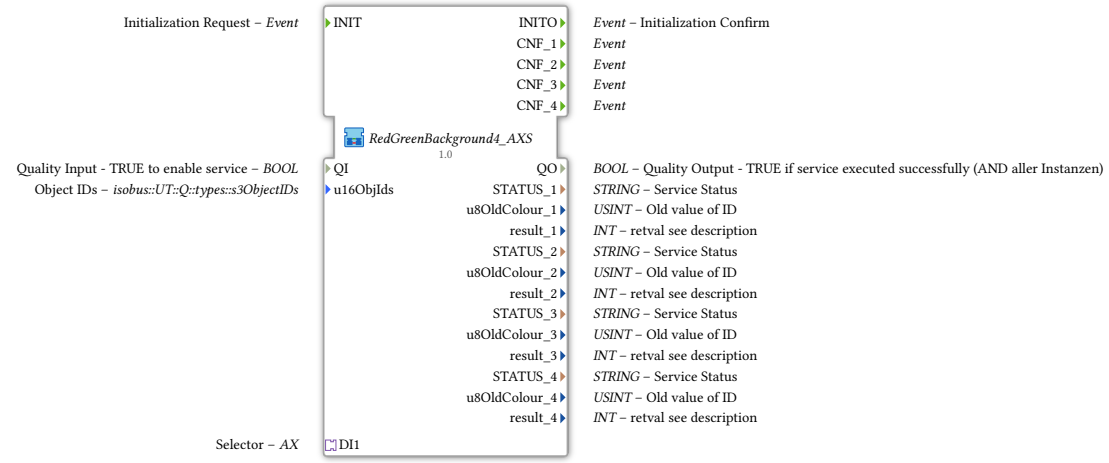
• – ○ • – ○ • – ○ •

Introduction

`RedGreenBackground4_AXS` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI` and `Q_BackgroundColour_4.QI`
- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_4`) of the `QO` outputs of `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO` and `Q_BackgroundColour_4.QO`
- **Initialization chain**: `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`, `Q_BackgroundColour_4.INITO` → `INITO`
- `E_MERGE_QO_A` (type `iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_4.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` and `Q_BackgroundColour_3.CNF` and passes the result to `E_MERGE_QO.EI1`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_2`) merges `E_MERGE_QO_A.EO` and `Q_BackgroundColour_4.CNF` and triggers `AND_QO.REQ`

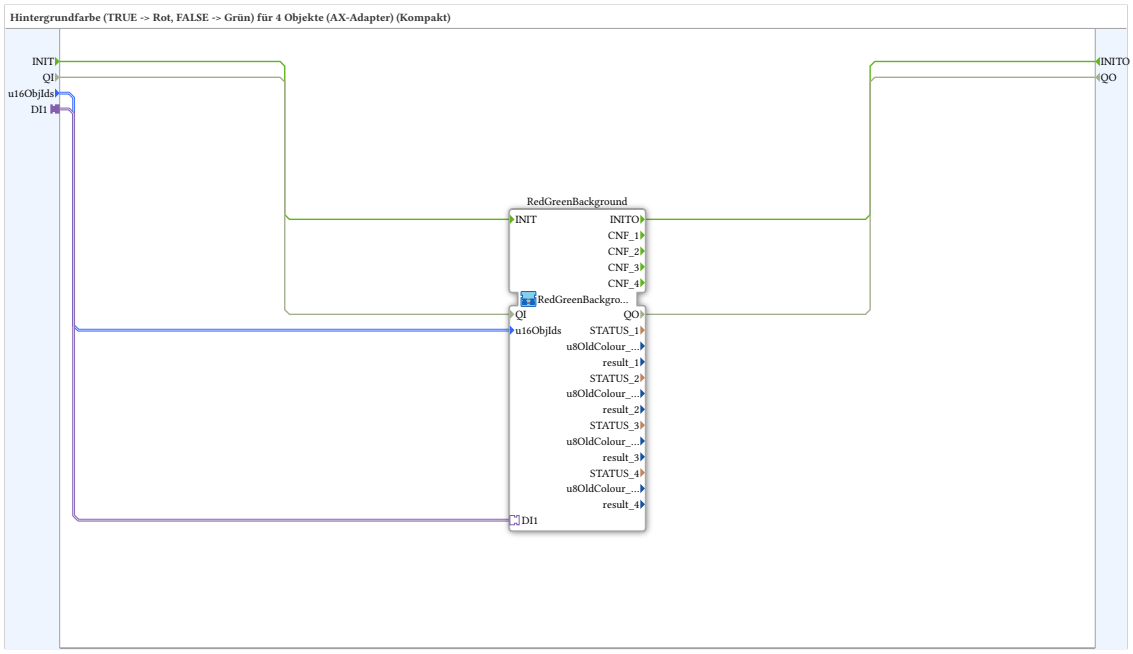
Summary

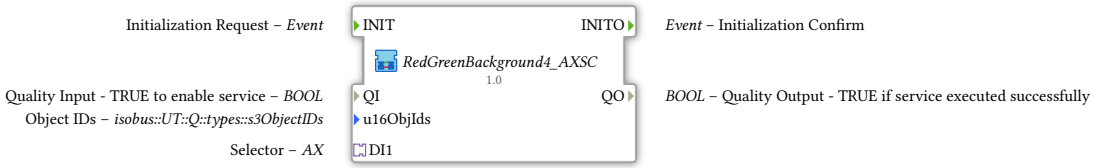
One of many variants in the background color block family: color pair Rot/Grün, 4 objects, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground4_AXSC





• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground4_AXSC` is a **compact wrapper** around `RedGreenBackground4_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground.QI` (initial value `TRUE`) and `RedGreenBackground.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

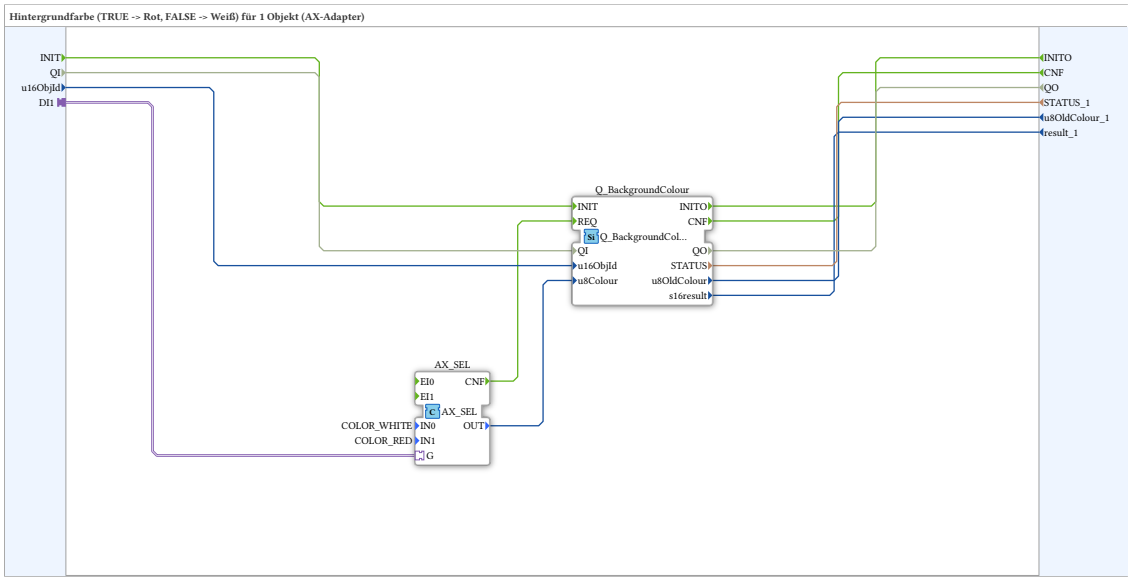
Summary

A pure pass-through to `RedGreenBackground4_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1_AX

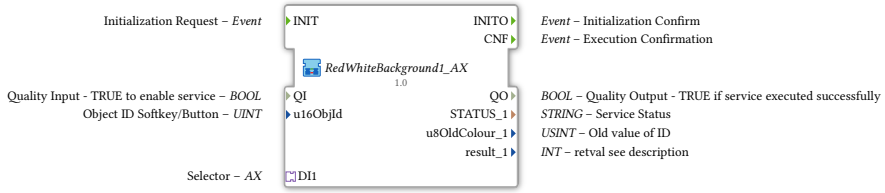


• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground1_AX` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColour.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColour.QO`.

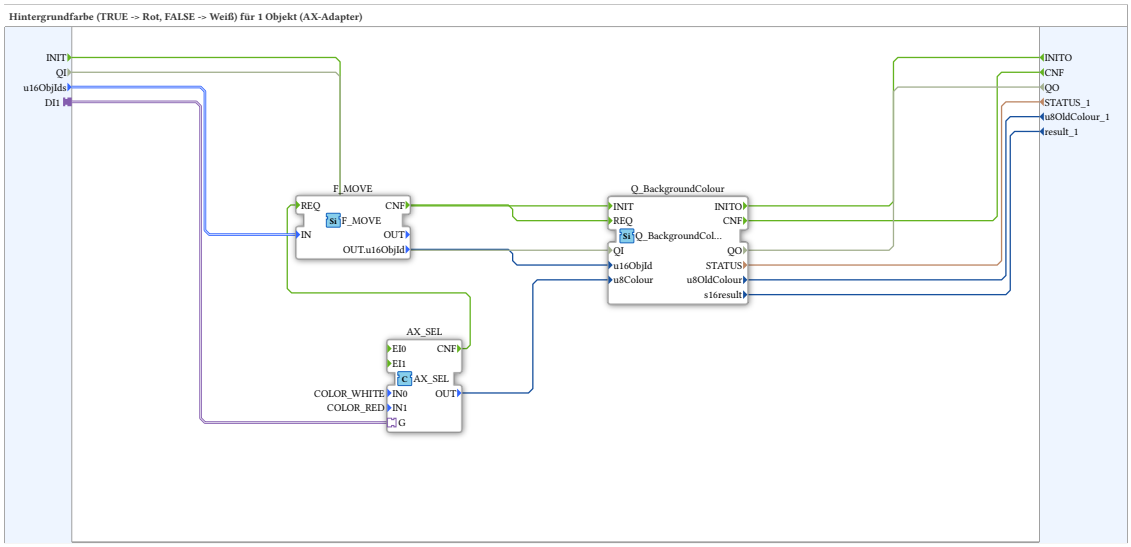
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 1 object, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1_AXS

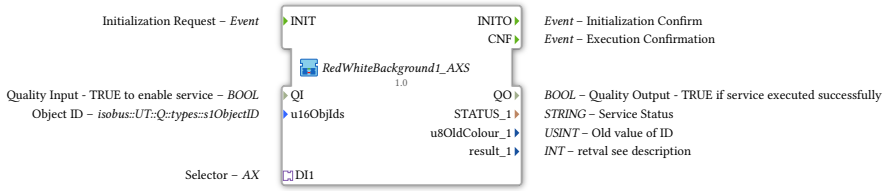


• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground1_AXS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColour.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColour.QO`.

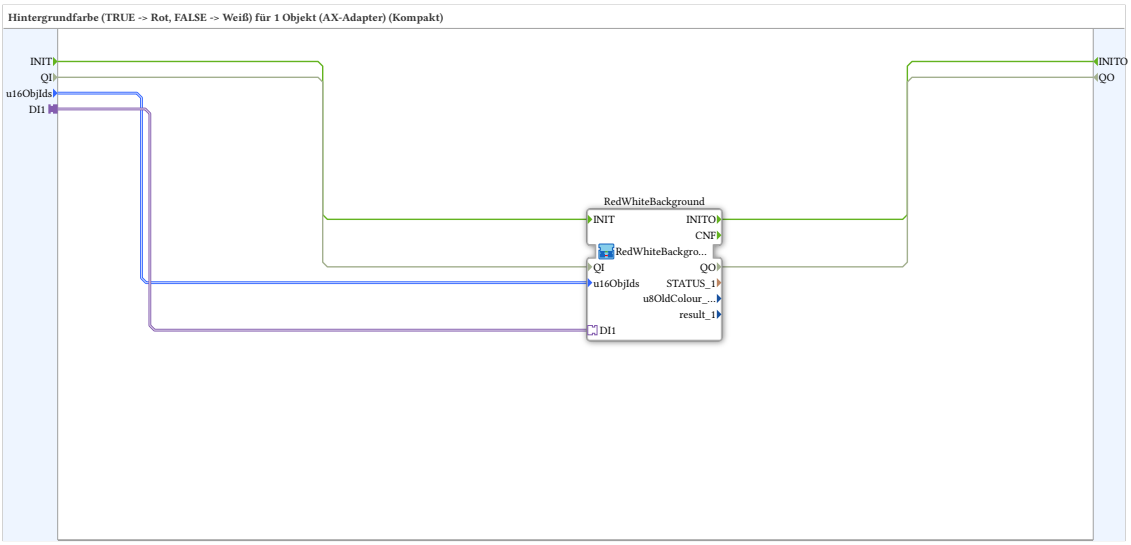
Summary

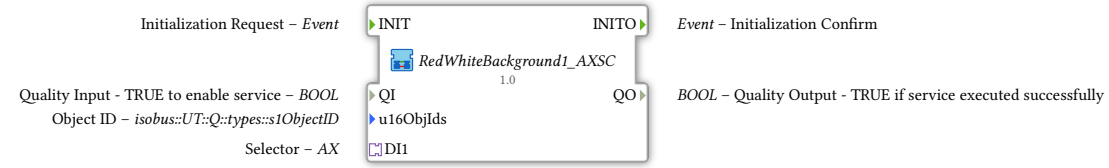
One of many variants in the background color block family: color pair Rot/Weiß, 1 object, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1_AXSC





• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground1_AXSC` is a **compact wrapper** around `RedWhiteBackground1_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

In addition, initialization and quality are passed through: `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` (initial value `TRUE`) ⟶ `RedWhiteBackground.QI` and `RedWhiteBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

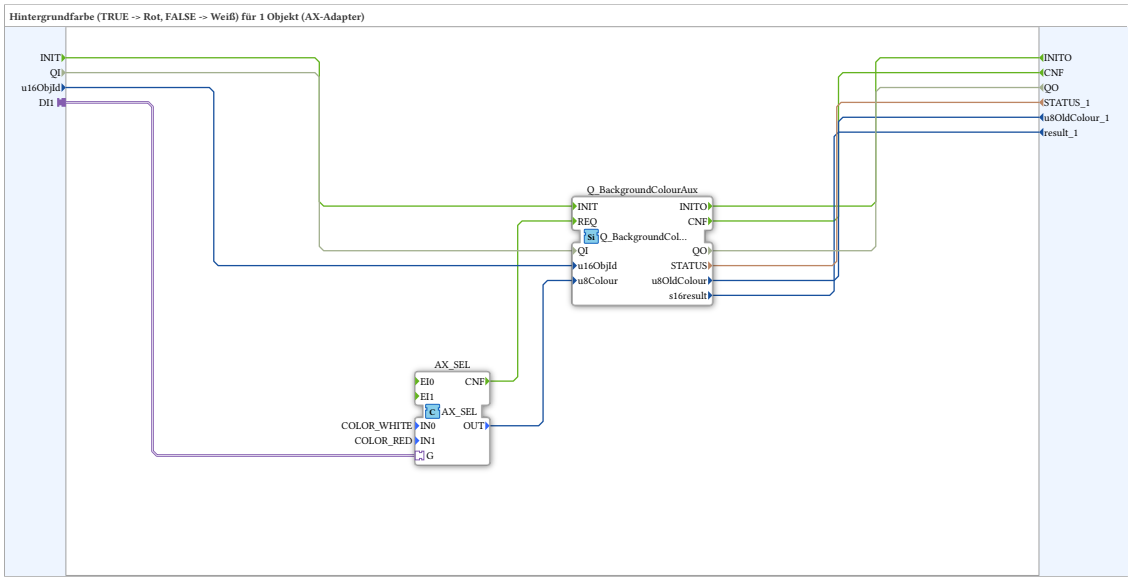
Summary

A pure pass-through to `RedWhiteBackground1_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1_aux_AX



• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground1_aux_AX` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColourAux.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColourAux.QO`.

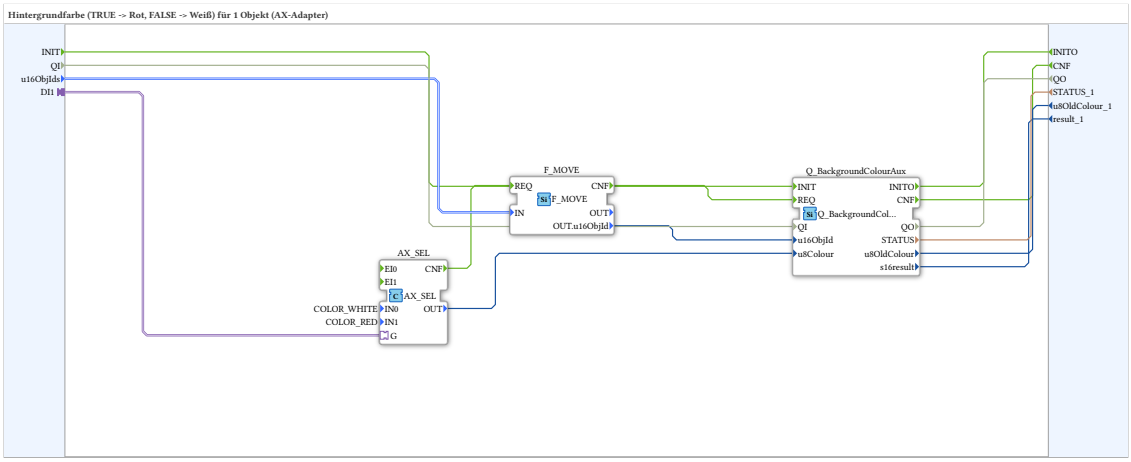
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 1 object, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1_aux_AXS

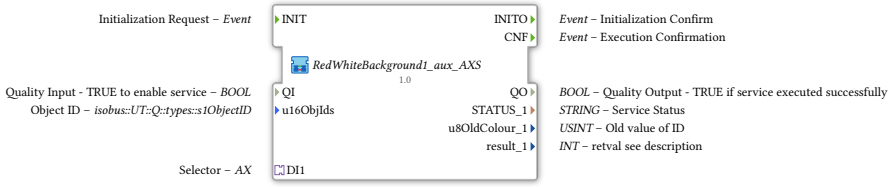


• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground1_aux_AXS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColourAux.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColourAux.QO`.

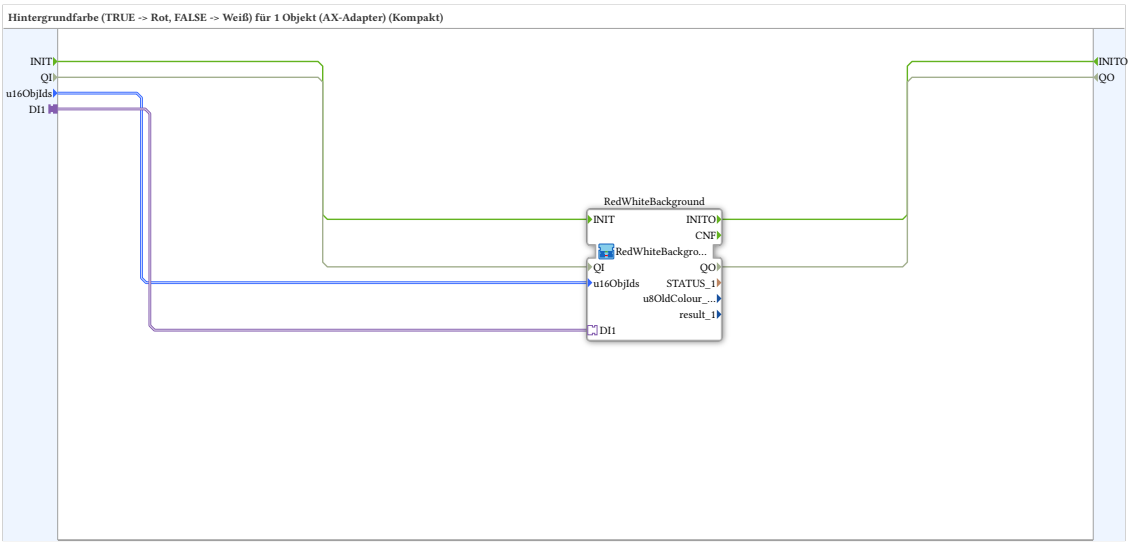
Summary

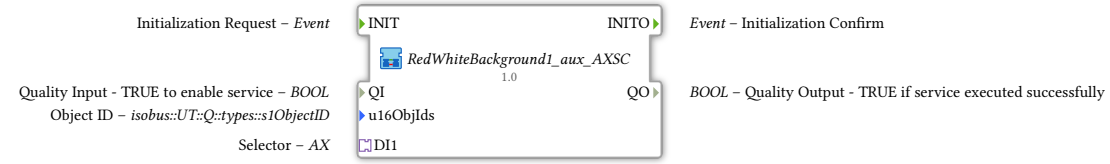
One of many variants in the background color block family: color pair Rot/Weiß, 1 object, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1_aux_AXSC





• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground1_aux_AXSC` is a **compact wrapper** around `RedWhiteBackground1_aux_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `RedWhiteBackground`: `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI`, `RedWhiteBackground.QO` ⟶ `QO`.

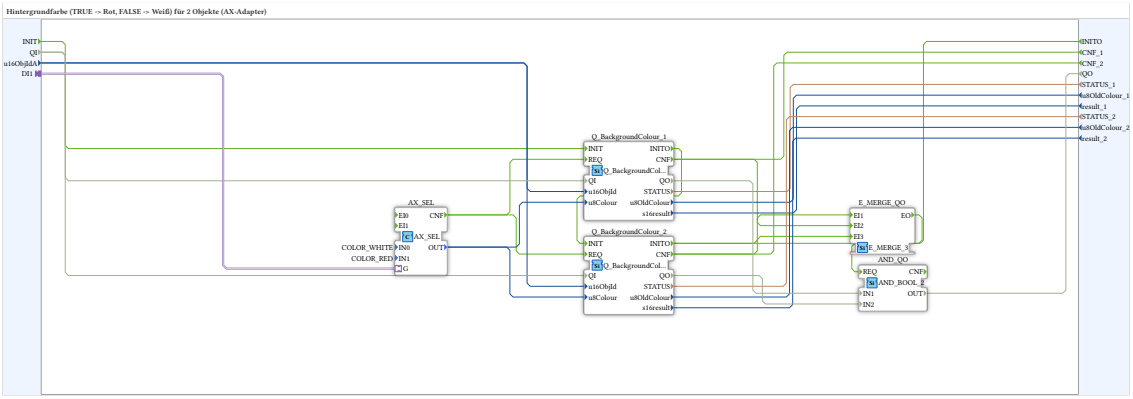
Summary

A pure pass-through to `RedWhiteBackground1_aux_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground2_AX



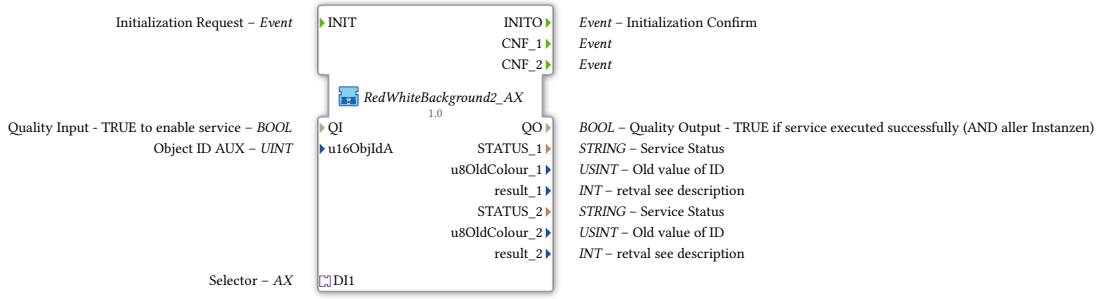
• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground2_AX` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via inputs `u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_3`) merges `Q_BackgroundColour_2.INITO` and the `CNF` events of both instances (`Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`) and thereby triggers `AND_QO.REQ`.

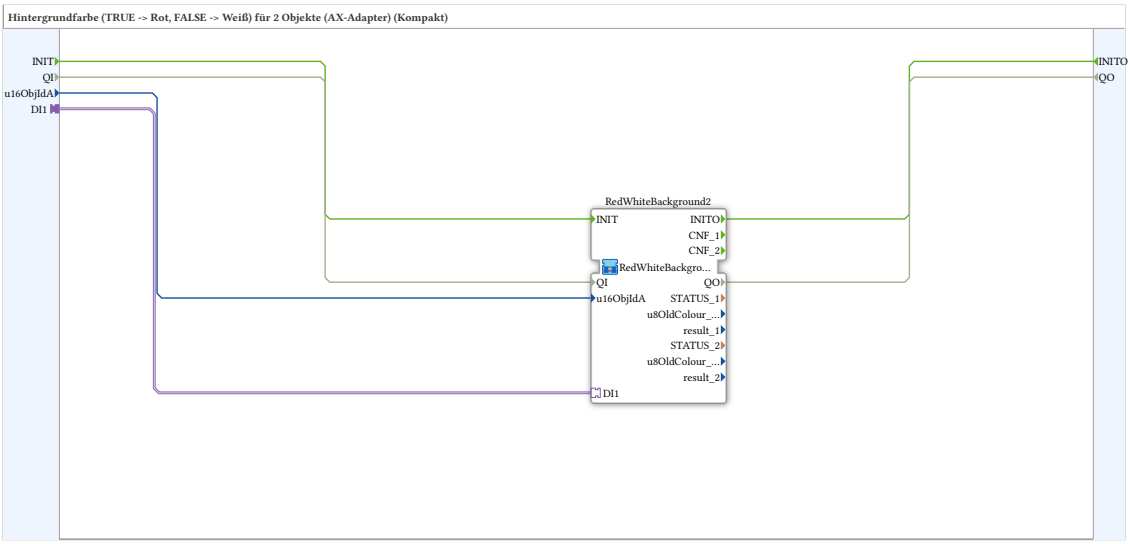
Summary

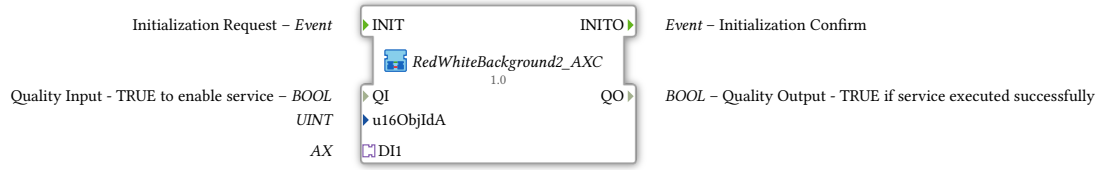
One of many variants in the background color block family: color pair Rot/Weiß, 2 objects, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground2_AXC





• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground2_AXC` is a **compact wrapper** around `RedWhiteBackground2_AX` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `RedWhiteBackground2`: `INIT` ⟶ `RedWhiteBackground2.INIT`, `RedWhiteBackground2.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground2.QI`, `RedWhiteBackground2.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

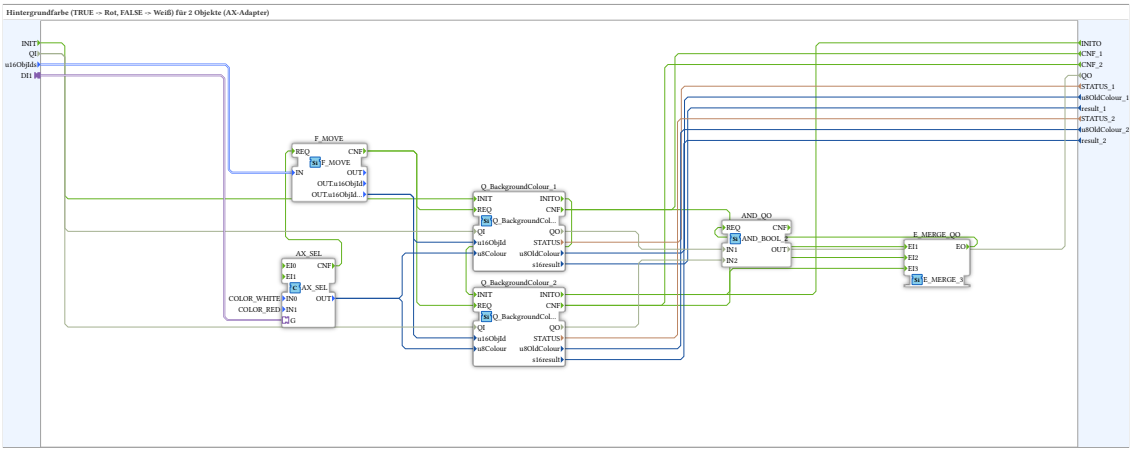
Summary

A pure pass-through to `RedWhiteBackground2_AX` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground2_AXS



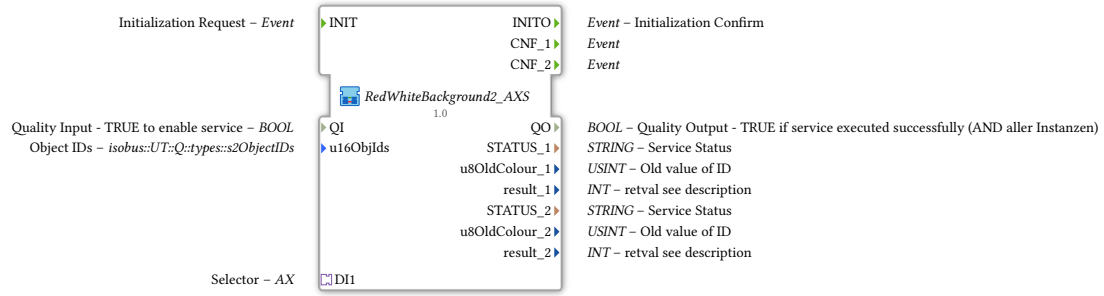
• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground2_AXS` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_BackgroundColour_1.QI` and `Q_BackgroundColour_2.QI`
- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_2`) of the `QO` outputs of `Q_BackgroundColour_1.QO` and `Q_BackgroundColour_2.QO`
- **Initialization chain**: `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `INITO`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_3`) merges `Q_BackgroundColour_2.INITO`, `Q_BackgroundColour_1.CNF` and `Q_BackgroundColour_2.CNF` and triggers `AND_QO.REQ`

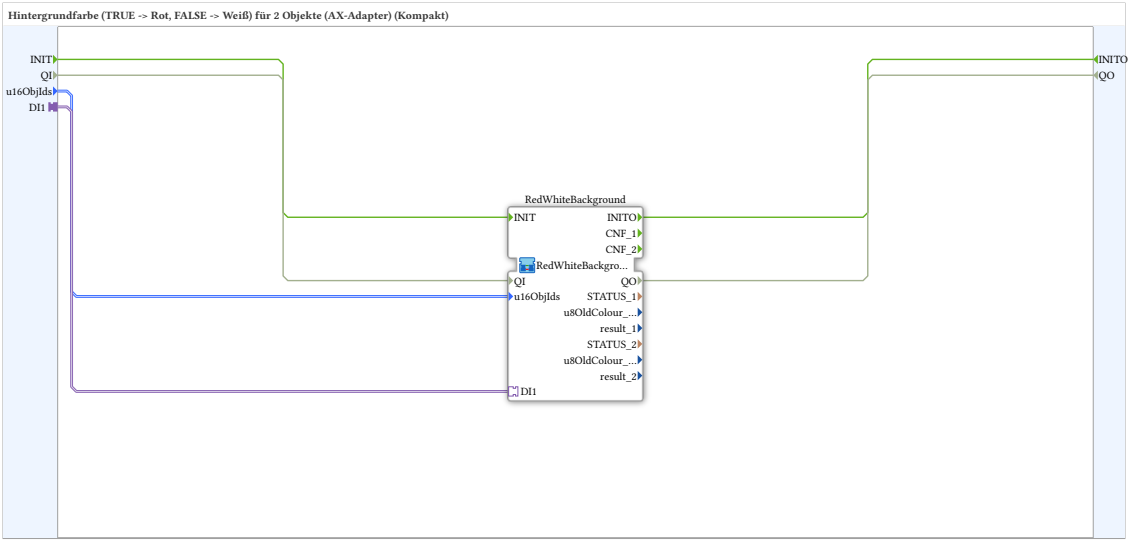
Summary

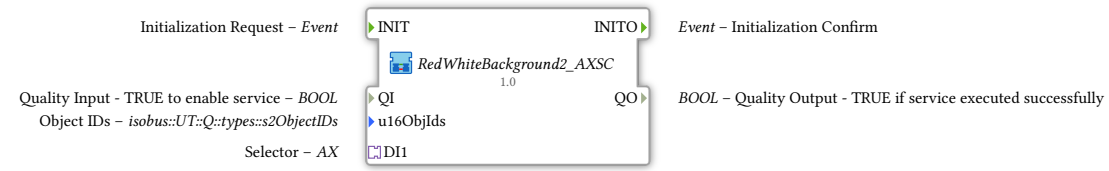
One of many variants in the background color block family: color pair Rot/Weiß, 2 objects, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground2_AXSC





• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground2_AXSC` is a **compact wrapper** around `RedWhiteBackground2_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI` (initial value `TRUE`) and `RedWhiteBackground.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

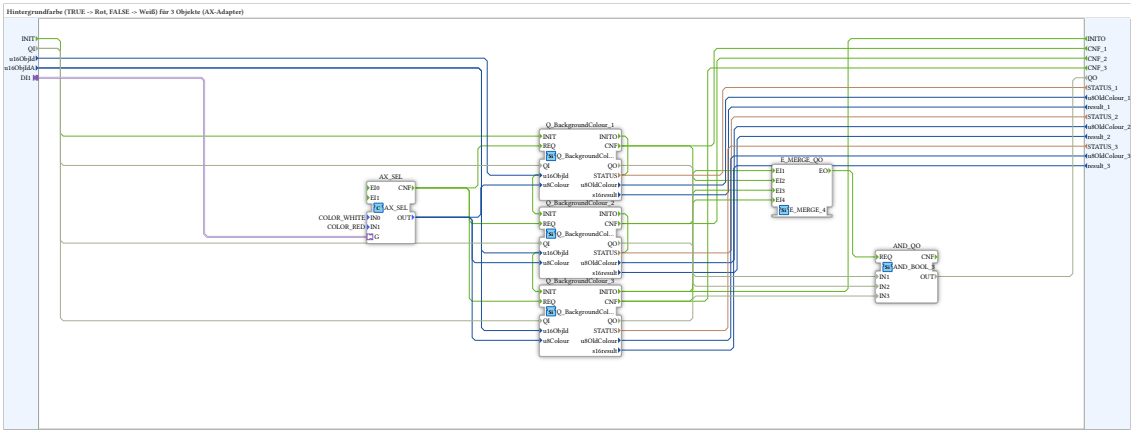
Summary

A pure pass-through to `RedWhiteBackground2_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground3_AX



• – ○ • – ○ • – ○ •

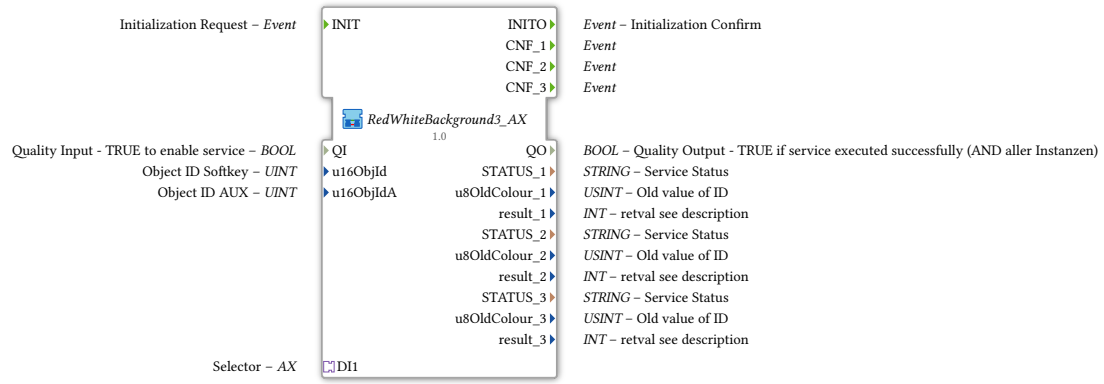
Introduction

`RedWhiteBackground3_AX` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` → **Rot**, `FALSE` → **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

Initialization and quality: In addition to the previous events, the block has the event input `INIT`, the event output `INITO`, as well as `QI` (BOOL, input, initial value `TRUE`) and `QO` (BOOL, output, AND of all instances). Init chain: `INIT` → `Q_BackgroundColour_1` → `Q_BackgroundColour_2` → `Q_BackgroundColour_3` → `INITO` (each `INITO` → next `INIT`). `QI` is passed on to all `Q_BackgroundColour_n.QI`. `QO` is formed by `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` → `AND_QO.IN1`, `Q_BackgroundColour_2.QO` → `AND_QO.IN2`, `Q_BackgroundColour_3.QO` → `AND_QO.IN3`, `AND_QO.OUT` → `QO`. `E_MERGE_QO` (`E_MERGE_4`) merges `Q_BackgroundColour_3.INITO` and the `CNF` outputs `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`; `E_MERGE_QO.EO` triggers `AND_QO.REQ`.

For the general pattern (selector → `AX_SEL/F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



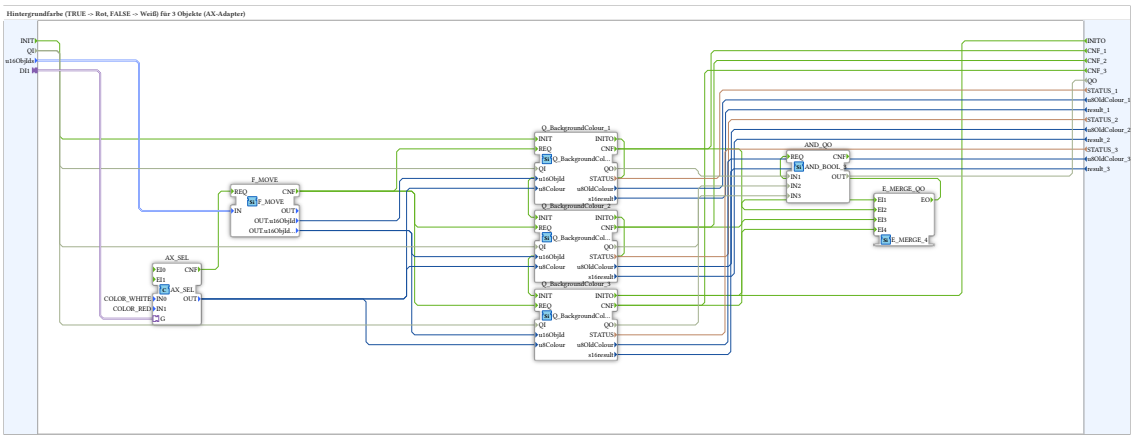
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 3 objects, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground3_AXS



• – ○ • – ○ • – ○ •

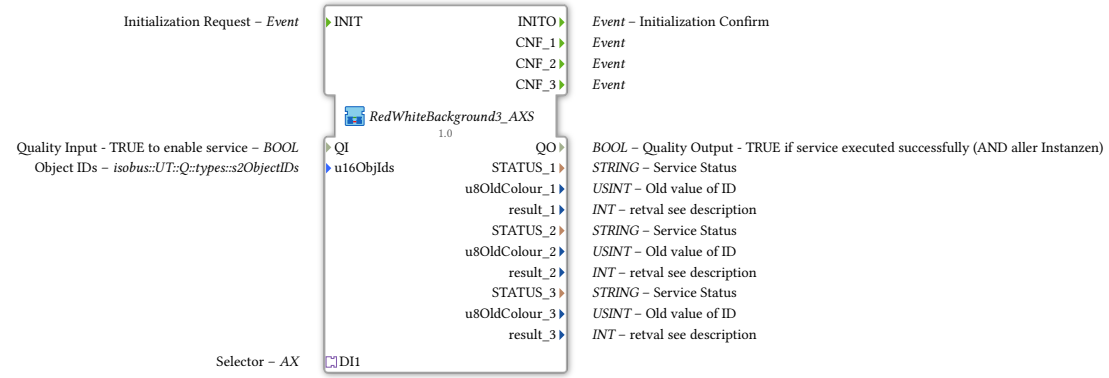
Introduction

`RedWhiteBackground3_AXS` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

In addition, the block has the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`, AND of all instances). `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` ⟶ `INITO` (so the three instances are initialized one after another). `QI` is passed to `QI` of all three instances. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`), which combines `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO` and `Q_BackgroundColour_3.QO`; its `REQ` is triggered by `E_MERGE_QO` (`iec61499::events::E_MERGE_4`), which merges `Q_BackgroundColour_3.INITO` and the `CNF` events of the three instances.



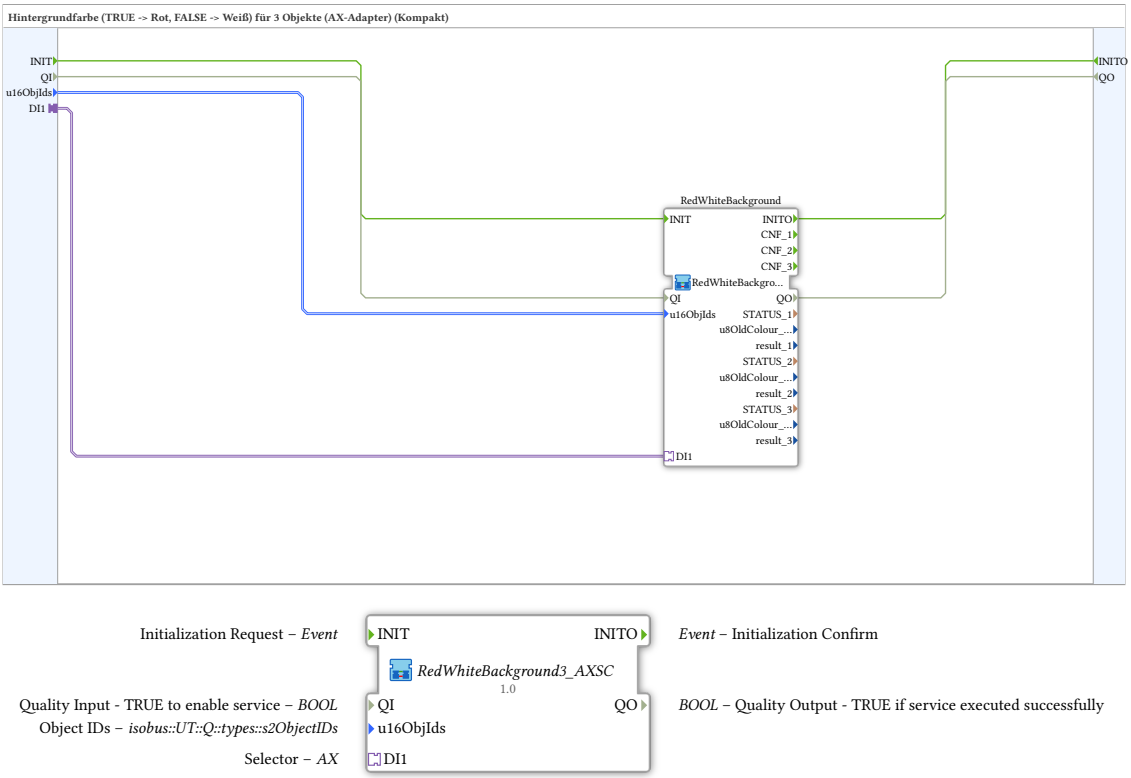
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 3 objects, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground3_AXSC



• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground3_AXSC` is a **compact wrapper** around `RedWhiteBackground3_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`) are passed through: `INIT` → `RedWhiteBackground.INIT`; `RedWhiteBackground.INITO` → `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) → `RedWhiteBackground.QI`; **data output `QO`** (`BOOL`) ← `RedWhiteBackground.QO`.

The inner SubApp instance `RedWhiteBackground` is of type `RedWhiteBackground3_AXS`.

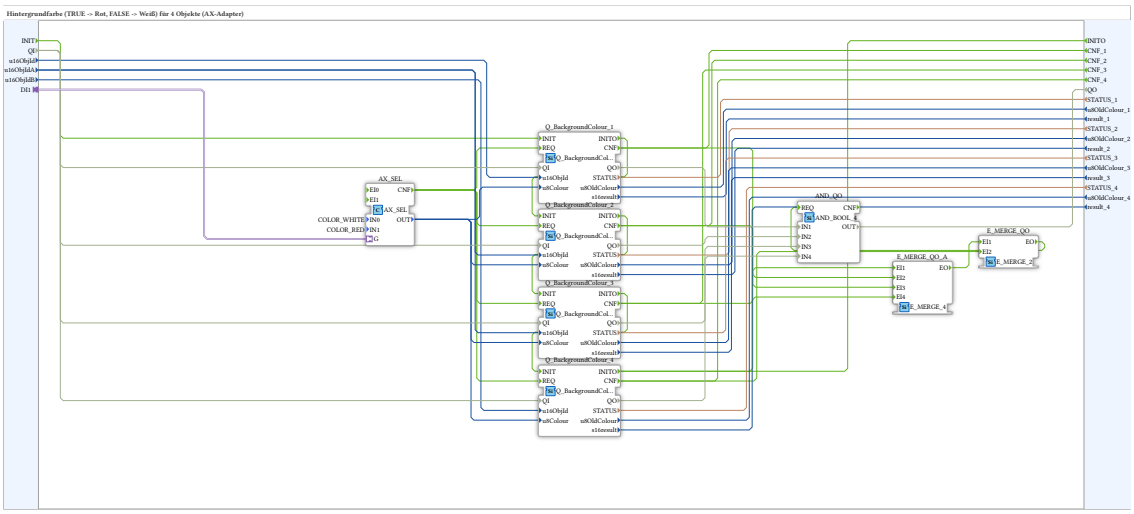
Summary

A pure pass-through to `RedWhiteBackground3_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground4_AX



• – ○ • – ○ • – ○ •

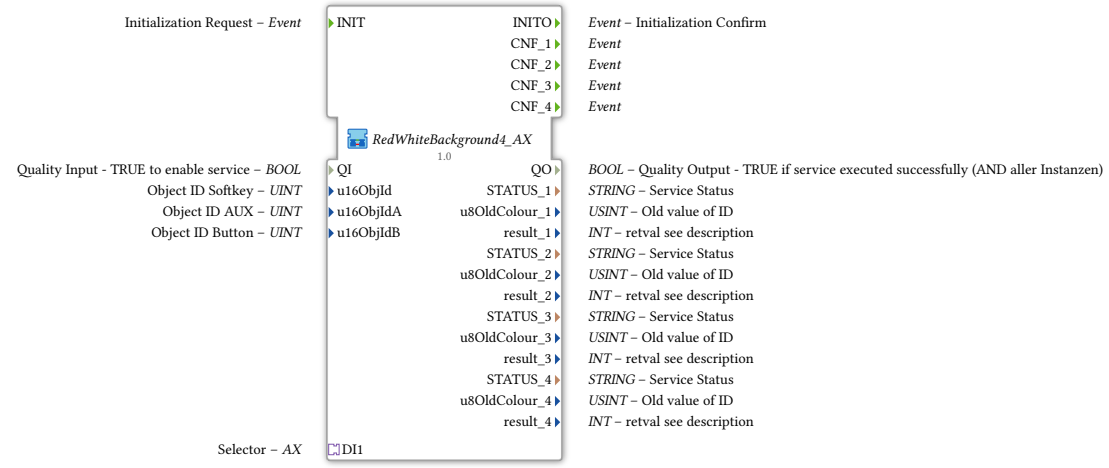
Introduction

`RedWhiteBackground4_AX` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` → **Rot**, `FALSE` → **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA, u16ObjIdB`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`u16ObjIdB`	`Q_BackgroundColour` (regular object)

Interface (quality pattern): event input `INIT` and event output `INITO` (each `EInit`), data input `QI` (`BOOL`, default `TRUE`) and data output `QO` (`BOOL`, AND of all instances). Initialization: `INIT` → `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` → `INITO`. Quality: `QI` → `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI`, `Q_BackgroundColour_4.QI`; `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO`, `Q_BackgroundColour_4.QO` → `AND_QO.IN1` to `AND_QO.IN4` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_4`); `AND_QO.OUT` → `QO`. The trigger `AND_QO.REQ` comes via two merge blocks: `E_MERGE_QO_A` (`E_MERGE_4`) merges `Q_BackgroundColour_4.INITO` and the `CNF` of `Q_BackgroundColour_1` to _3, and `E_MERGE_QO` (`E_MERGE_2`) merges its output with `Q_BackgroundColour_4.CNF`; `E_MERGE_QO.EO` → `AND_QO.REQ`.

For the general pattern (selector → `AX_SEL/F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



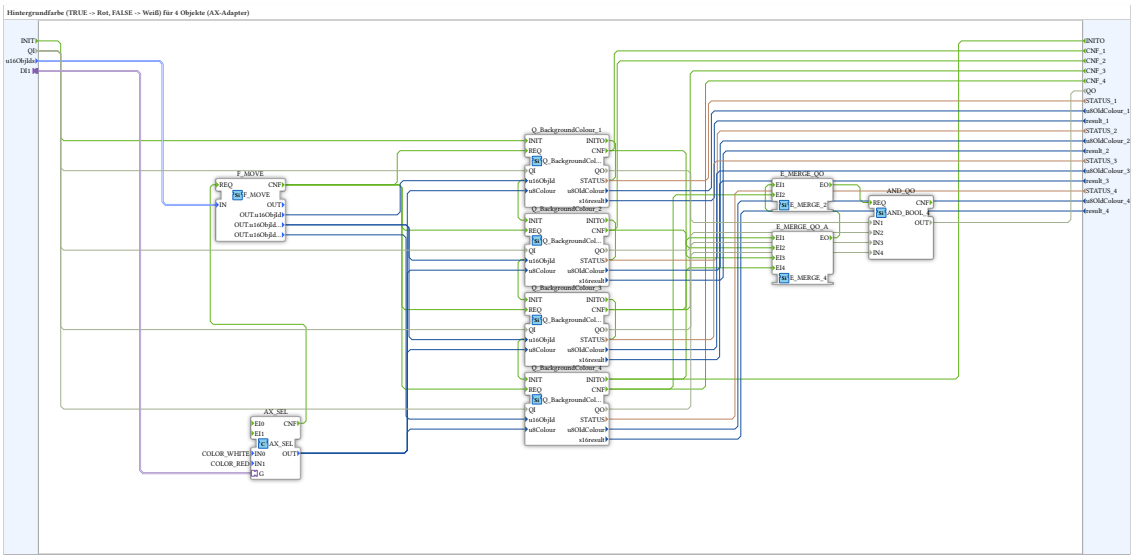
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 4 objects, adapter selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground4_AXS



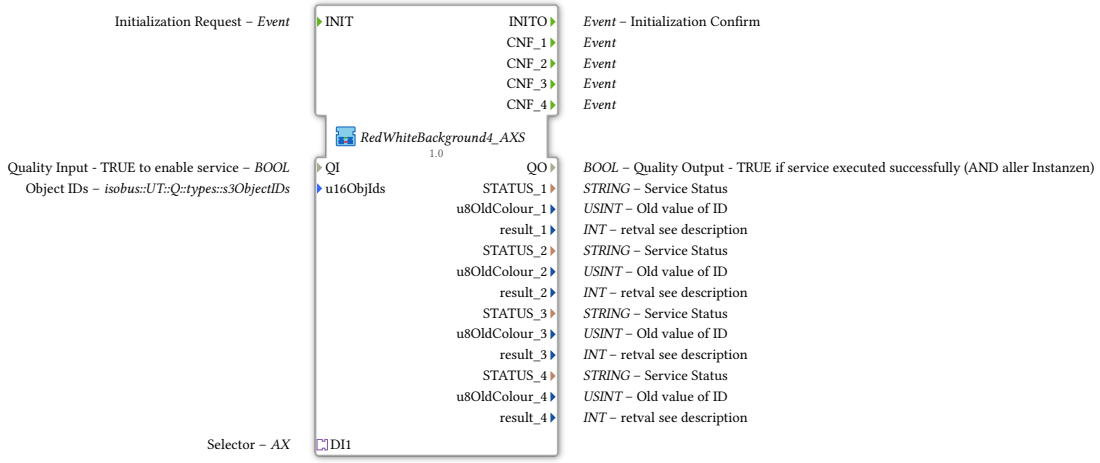
• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground4_AXS` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives via an `AX` adapter socket (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI` and `Q_BackgroundColour_4.QI`
- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_4`) of the `QO` outputs of `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO` and `Q_BackgroundColour_4.QO`
- **Initialization chain**: `INIT` ⟶ `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`, `Q_BackgroundColour_4.INITO` ⟶ `INITO`
- `E_MERGE_QO_A` (type `iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_4.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` and `Q_BackgroundColour_3.CNF` and passes the result to `E_MERGE_QO.EI1`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_2`) merges `E_MERGE_QO_A.EO` and `Q_BackgroundColour_4.CNF` and triggers `AND_QO.REQ`

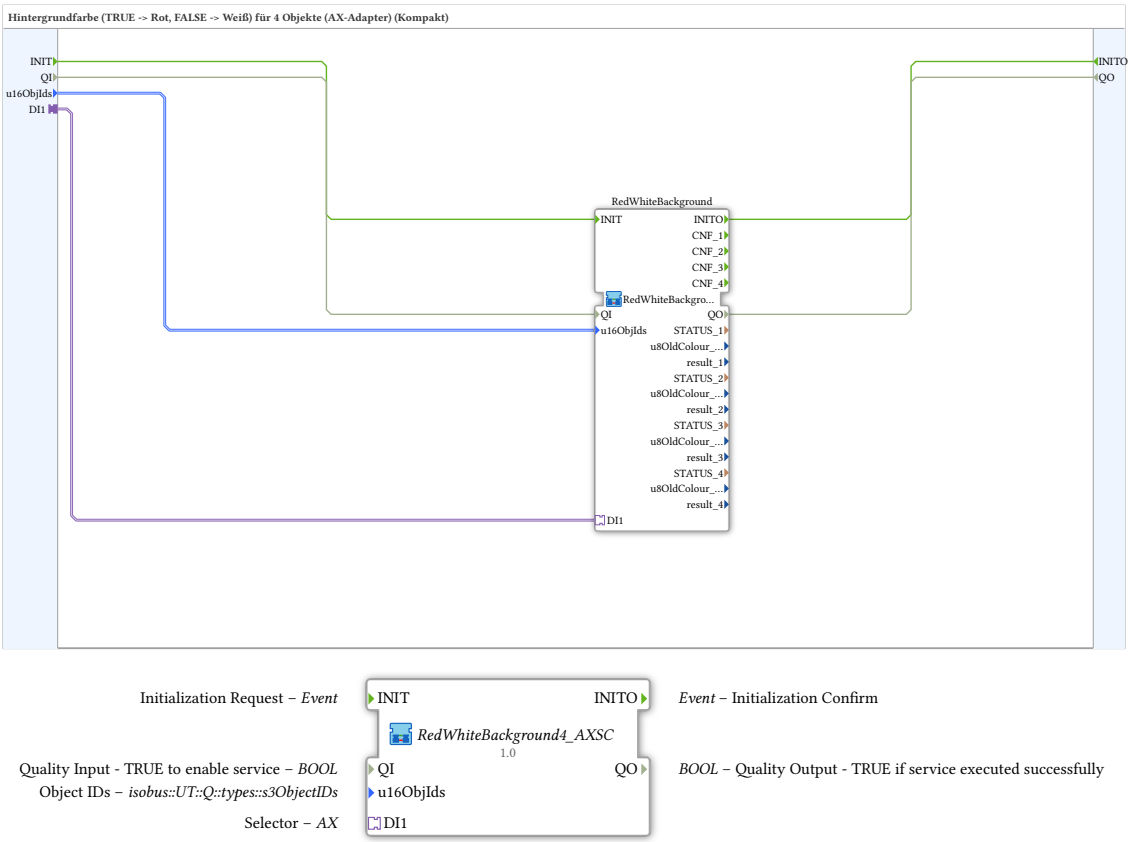
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 4 objects, adapter selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground4_AXSC



• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground4_AXSC` is a **compact wrapper** around `RedWhiteBackground4_AXS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI` (initial value `TRUE`) and `RedWhiteBackground.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

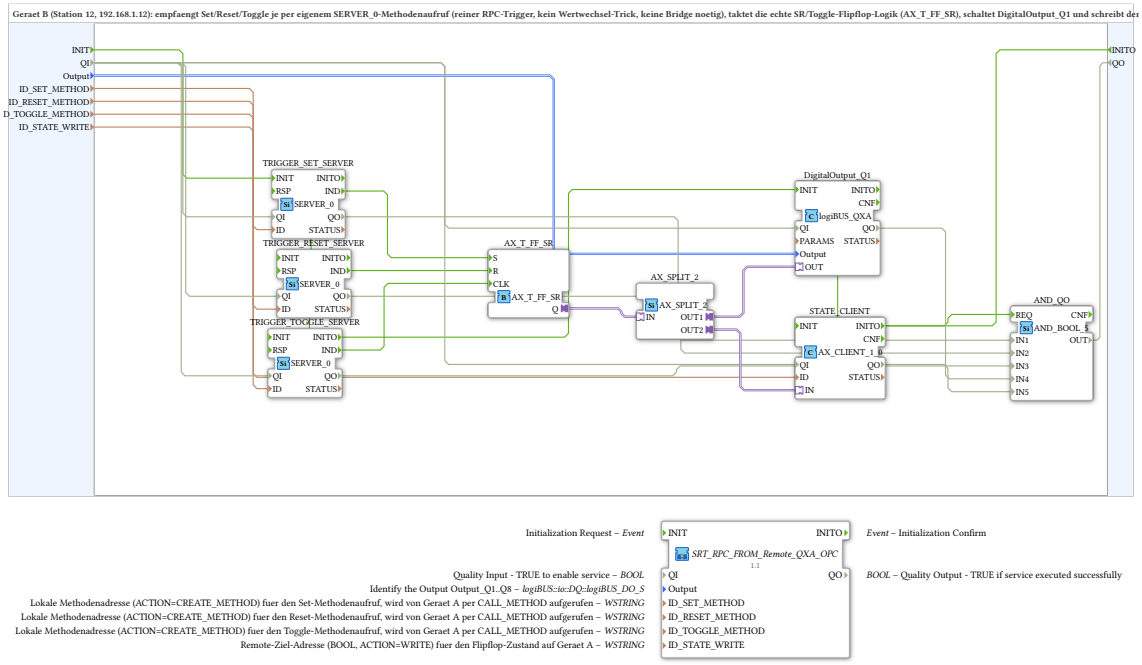
Summary

A pure pass-through to `RedWhiteBackground4_AXS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SRT_RPC_FROM_Remote_QXA_OPC



• — ○ • — ○ • — ○ •

Introduction

SRT_RPC_FROM_Remote_QXA_OPC is a composite subapplication designed for device B (Station 12, IP 192.168.1.12) that receives Set, Reset, and Toggle commands via three independent OPC-UA server method calls. Unlike approaches that rely on value-change tricks or bridging, this subapp uses pure RPC triggers — each command is delivered through its own `SERVER_0` method invocation initiated by device A.

The subapp contains the actual flip-flop logic based on the `AX_T_FF_SR` adapter, which combines SR and toggle behavior. The resulting state drives a physical digital output (`DigitalOutput_Q1`) and is actively written back to device A via an `AX_CLIENT_1_0` client, ensuring that the central controller always has the current output state. The entire protocol is encapsulated within the `MyLib::sys` composite, following a "SUB style" design where the resource of the device remains free of protocol-specific logic.

Interface Structure

The subapplication exposes the events `INIT`/`INITO`, the data `QI`/`QO` and the data inputs for output and addresses at its boundary; it has no external adapters.

Event Inputs

- **INIT** (`EInit`): initialization request; starts the init chain at `TRIGGER_SET_SERVER.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `STATE_CLIENT.INITO`, the end of the init chain

Data Inputs

Name	Type	Initial Value	Description
`QI`	`BOOL`	`TRUE`	Quality input; passed on to all five service FBs (three `SERVER_0`, `DigitalOutput_Q1`, `STATE_CLIENT`).
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	`logiBUS_DO::Invalid`	Identifies the target digital output (Output_Q1..Q8).

ID_SET_METHOD	WSTRING	—	Local method address (ACTION=CREATE_METHOD) for the Set method, invoked by device A via CALL_METHOD.
ID_RESET_METHOD	WSTRING	—	Local method address (ACTION=CREATE_METHOD) for the Reset method, invoked by device A via CALL_METHOD.
ID_TOGGLE_METHOD	WSTRING	—	Local method address (ACTION=CREATE_METHOD) for the Toggle method, invoked by device A via CALL_METHOD.
ID_STATE_WRITE	WSTRING	—	Remote target address (BOOL, ACTION=WRITE) for writing the flip-flop state back to device A.

Data Outputs

- **QO** (BOOL): quality output, AND combination (AND_QO, AND_BOOL_5) of the QO outputs of the three SERVER_0 instances, of DigitalOutput_Q1 and of STATE_CLIENT

Adapters

None at the subapplication interface. All adapter connections are internal to the network.

Functionality

The subapp implements a complete remote-controlled SR/T flip-flop with output feedback:

1. **Command reception**: Three SERVER_0 instances (TRIGGER_SET_SERVER, TRIGGER_RESET_SERVER, TRIGGER_TOGGLE_SERVER) each listen for a specific OPC-UA method call. Device A invokes these methods remotely to send Set, Reset, or Toggle commands.
2. **Flip-flop logic**: The incoming event from each server is forwarded to the AX_T_FF_SR block:
   - Set event (TRIGGER_SET_SERVER.IND) drives the S input ⟶ sets the output Q to TRUE.
   - Reset event (TRIGGER_RESET_SERVER.IND) drives the R input ⟶ sets the output Q to FALSE.
   - Toggle event (TRIGGER_TOGGLE_SERVER.IND) drives the CLK input ⟶ inverts the current Q state.
3. **Immediate response**: Each server's IND event is directly wired back to its own RSP output. This ensures the OPC-UA server thread is released instantly, preventing long blocking periods (a known issue fixed in version 1.1).
4. **Signal distribution**: The flip-flop output Q is fed into AX_SPLIT_2, which fans it out to two consumers:
   - **OUT1** ⟶ DigitalOutput_Q1 (logiBUS_QXA) — switches the physical digital output on the logiBUS device.
   - **OUT2** ⟶ STATE_CLIENT (AX_CLIENT_1_0) — sends the new state back to device A via an OPC-UA write operation at the address specified by ID_STATE_WRITE.
5. **Initialization and quality**: INIT is chained: INIT ⟶ TRIGGER_SET_SERVER ⟶ TRIGGER_RESET_SERVER ⟶ TRIGGER_TOGGLE_SERVER ⟶ DigitalOutput_Q1 ⟶ STATE_CLIENT (each INITO ⟶ next INIT); STATE_CLIENT.INITO ⟶ INITO and AND_QO.REQ. QI goes to all five FBs; their QO outputs go in this order to AND_QO.IN1 to AND_QO.IN5 (iec61131::booleanOperators::AND_BOOL_5), AND_QO.OUT ⟶ QO.
6. **Configuration inputs**: All method addresses and the output channel are configured via the subapp's data inputs, making the block reusable for different method IDs and output channels without modifying the internal structure.

Technical Features

- **RPC-based trigger**: Commands are delivered as explicit OPC-UA method calls, eliminating the need for value-change detection tricks or intermediate bridge FBs.
- **Tailored server instances**: Separate `SERVER_0` for Set, Reset, and Toggle allows independent method addresses and clear segregation of control semantics.
- **Non-blocking response handling**: The `IND → RSP` direct connection prevents the open62541 server thread from being held, avoiding global `serviceMutex` contention (critical fix in version 1.1 that improved project-wide OPC-UA responsiveness).
- **Combined SR + T flip-flop**: The `AX_T_FF_SR` adapter provides both deterministic Set/Reset behavior and toggle functionality in one compact element.
- **Active state feedback**: The current flip-flop state is actively written back to the controlling device, ensuring synchronized state awareness without poll requests.
- **Protocol encapsulation**: All communication logic resides within the `MyLib::sys` composite, keeping the device resource clean and the subapp portable across resources and devices.
- **Structured configuration**: All network addresses and output identifiers are externalized as WSTRING/ struct inputs, enabling easy adaptation to different OPC-UA topologies.

State Overview

The core state machine is implemented by the `AX_T_FF_SR` block, which operates as follows:

Input Event	Condition	Resulting State (Q)
Set (`S`)	Always	`TRUE`
Reset (`R`)	Always	`FALSE`
Toggle (`CLK`)	Rising edge	Invert current Q: `TRUE → FALSE` or `FALSE → TRUE`

- **Initial state**: `FALSE` (after startup or explicit Reset).
- **Stable states**: `TRUE` (output active) or `FALSE` (output inactive), with no undefined or metastable conditions.
- **Conflict handling**: If Set and Reset arrive simultaneously, priority behavior follows the `AX_T_FF_SR` adapter semantics (typically Reset dominates, depending on the underlying implementation).

The `Q` output is then split and routed to both the digital output and the remote feedback client, so the observed physical state always matches the state reported back to device A.

Application Scenarios

- **Remote output control**: A central controller (device A) remotely sets, resets, or toggles digital output Q1 on a field device (device B) over OPC-UA.
- **Distributed automation networks**: Coordinating outputs across multiple stations (e.g., Station 12 in a larger plant) where each station hosts its own OPC-UA server and receives commands on demand.
- **State-synchronized systems**: Scenarios where the controller must know the exact output state after each command — the active write-back via `CLIENT_1_0` guarantees this without extra polling.
- **Integration in machine safety or indicator logic**: Toggle functionality is useful for alternating signals (blinking lamps, periodic valve switching), while Set/Reset provides deterministic latching.

Comparison with Similar Blocks

Feature	SRT_RPC_FROM_Remote_QXA_OPC	Typical local I/O FB	Bridge-based remote FB
Command transport	OPC-UA method calls (RPC)	Local event connections	Value-change signals
State feedback	Active write-back via client	Direct wiring	Requires extra feedback FB
Server thread blocking	Avoided (IND→RSP direct)	N/A	Can block if RSP unconnected
Protocol location	Inside subapp (SUB style)	In resource	In resource

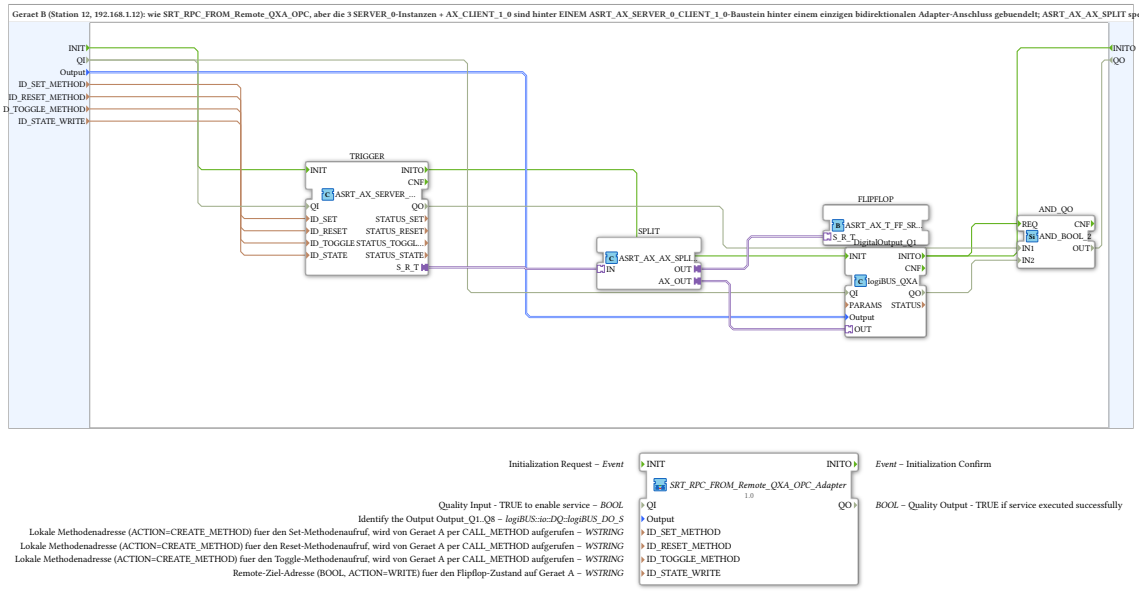
Reusability	High — copy subapp between devices	Low — tied to resource	Medium
Toggle capability	Built-in (`AX_T_FF_SR`)	Usually only Set/Reset	Depends on logic

Compared to a bridge-based solution that translates value changes into commands, this subapp provides a cleaner, semantically richer interface. Compared to placing the protocol in the device resource, the SUB style allows the same communication pattern to be reused across multiple devices without duplicating resource-level wiring.

Conclusion

SRT_RPC_FROM_Remote_QXA_OPC is a well-structured subapplication that encapsulates remote OPC-UA-based control of a digital output in a compact, reusable composite. Its three dedicated server instances allow precise Set, Reset, and Toggle semantics through pure RPC calls, while the direct `IND→RSP` wiring ensures responsive OPC-UA behavior. The integrated `AX_T_FF_SR` flip-flop provides reliable state management, and the active feedback via `CLIENT_1_0` keeps the controlling device synchronized. By keeping all protocol logic inside the subapp, the design enhances modularity, reusability, and maintainability across distributed IEC 61499 applications.

SRT_RPC_FROM_Remote_QXA_OPC_Adapter



• — ○ • — ○ • — ○ •

Introduction

The **SRT_RPC_FROM_Remote_QXA_OPC_Adapter** is a subapplication (SUB-style) that provides remote control of a digital output (QXA) via a single bidirectional adapter connection. It is intended for Device B (Station 12, 192.168.1.12) and is structurally an optimized variant of the SRT_RPC_FROM_Remote_QXA_OPC subapplication: instead of exposing multiple server/client instances separately, all protocol handling is bundled behind one `ASRT_AX_SERVER_0_CLIENT_1_0` block, and the incoming adapter stream is split to drive both the digital output and a Set/Reset/Toggle flip-flop logic. The communication protocol itself is embedded in the MyLib::sys composite type, not in the device resource.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; goes to `TRIGGER.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `DigitalOutput_Q1.INITO`

Data Inputs

Name	Type	Initial Value	Description
QI	BOOL	TRUE	Quality input - TRUE enables the service; goes to TRIGGER.QI and DigitalOutput_Q1.QI.
Output	logiBUS::io::DQ::logiBUS_DQ_8	logiBUS_DO::Invalid	Identifies which physical output (Q1..Q8) is to be controlled.
ID_SET_METHOD	WSTRING	–	Local method address (ACTION=CREATE_METHOD) used for the Set method call, invoked remotely by Device A via CALL_METHOD.
ID_RESET_METHOD	WSTRING	–	Local method address used for the Reset method call, invoked remotely by Device A via CALL_METHOD.
ID_TOGGLE_METHOD	WSTRING	–	Local method address used for the Toggle method call, invoked remotely by Device A via CALL_METHOD.
ID_STATE_WRITE	WSTRING	–	Remote target address (BOOL, ACTION=WRITE) for writing back the flip-flop state to Device A.

Data Outputs

- **QO** (BOOL): quality output - TRUE if the service was executed successfully (AND of TRIGGER.QO and DigitalOutput_Q1.QO)

Adapters

The subapplication exposes no interface-level adapter ports. All adapter communication is performed internally through the TRIGGER block, which bundles the server/client functionality. No external adapter is visible at the subapplication boundary.

Functionality

The subapplication processes an incoming bidirectional adapter stream that carries Set, Reset, and Toggle commands from a remote device (Device A). The internal flow is as follows:

1. **TRIGGER** (adapter::net::ASRT_AX_SERVER_0_CLIENT_1_0) – receives the remote command stream over its S_R_T adapter port. Its data inputs are bound to the subapplication's method address variables (ID_SET_METHOD, ID_RESET_METHOD, ID_TOGGLE_METHOD, ID_STATE_WRITE) to enable remote method invocation. QI is passed on from the subapplication input QI.
2. **SPLIT** (adapter::events::bidirectional::ASRT_AX_AX_SPLIT) – takes the incoming stream from TRIGGER.S_R_T and splits it into two parallel paths:
   - OUT → forwarded to the flip-flop logic
   - AX_OUT → forwarded to the digital output block

3. **FLIPFLOP** (`adapter::events::bidirectional::ASRT_AX_T_FF_SR_2`) – implements the Set/Reset/Toggle flip-flop behavior. It consumes the `OUT` path from the splitter and maintains the internal state. The current state is written back to Device A using the `ID_STATE_WRITE` remote address.
4. **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`) – receives the `AX_OUT` path from the splitter and drives the physical output selected by the `Output` data input. `QI` is passed on to it as well.

Initialization and quality: `INIT` → `TRIGGER.INIT`, `TRIGGER.INITO` → `DigitalOutput_Q1.INIT`, `DigitalOutput_Q1.INITO` → `INITO` and → `AND_QO.REQ`. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`), which combines `TRIGGER.QO` and `DigitalOutput_Q1.QO`.

The data flow ensures that the same incoming command stream simultaneously updates both the flip-flop state and the physical output, while maintaining a single bidirectional adapter connection to the remote controller.

Technical Features

- **Single bundled adapter connection**: All three server instances and one client instance are consolidated behind a single `ASRT_AX_SERVER_0_CLIENT_1_0` block, reducing connection complexity at the subapplication boundary.
- **Method-based remote control**: Set, Reset, and Toggle operations are exposed as local methods (CREATE_METHOD) and invoked remotely via CALL_METHOD, with addresses supplied through WSTRING inputs.
- **Bidirectional state synchronization**: The flip-flop state is written back to the remote device via a configurable target address (`ID_STATE_WRITE`), enabling closed-loop monitoring.
- **Signal splitting**: The `ASRT_AX_AX_SPLIT` block distributes the incoming adapter stream to two consumers without losing information.
- **SUB-style architecture**: Protocol logic is contained within the MyLib::sys composite type, keeping the device resource free from protocol-specific configuration.
- **Output selection**: The physical output (Q1..Q8) is selected at runtime via the `Output` data input.

State Overview

The internal state machine is governed by the `ASRT_AX_T_FF_SR_2` flip-flop, which combines Set (S), Reset (R), and Toggle (T) semantics:

- **Set**: Forces the flip-flop output to TRUE.
- **Reset**: Forces the flip-flop output to FALSE.
- **Toggle**: Inverts the current flip-flop output.

The flip-flop state is kept consistent with the physical output `DigitalOutput_Q1` because both are driven by the same split adapter stream. The current state is reported back to the remote device via the write address provided in `ID_STATE_WRITE`.

Application Scenarios

- **Remote I/O control over LAN**: Device A (a controller or HMI) issues Set/Reset/Toggle commands to Device B over an Ethernet link, controlling a discrete digital output on Device B.
- **Distributed automation cells**: Suitable for stations where a single bidirectional channel must carry both control commands and status feedback for a flip-flop-driven actuator.
- **PLC-to-PLC communication**: Can be used in multi-device production lines where a central PLC manages outputs on peripheral stations via structured method calls.
- **Reusable subapplication**: Because the protocol stack is embedded in a composite type, the subapplication can be instantiated multiple times within the same device or across devices without resource-level modifications.

Comparison with Similar Blocks

Compared to its predecessor **SRT_RPC_FROM_Remote_QXA_OPC**, this subapplication offers the following differences:

Aspect	SRT_RPC_FROM_Remote_QXA_OPC	SRT_RPC_FROM_Remote_QXA_OPC_Adapter
Adapter ports	Three separate SERVER_0 instances plus one AX_CLIENT_1_0	One bundled `ASRT_AX_SERVER_0_CLIENT_1_0` behind a single bidirectional port

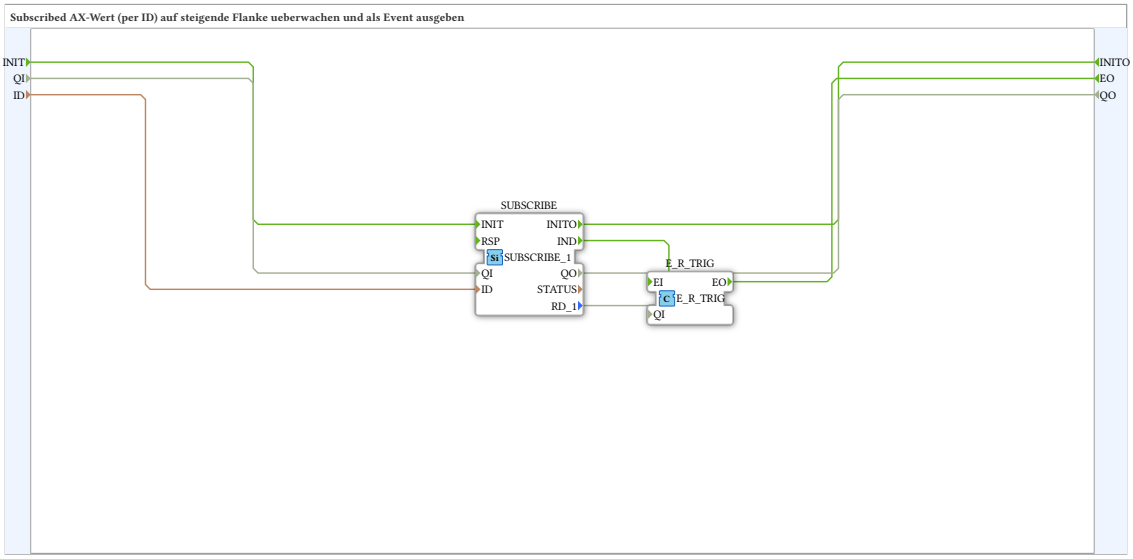
Signal distribution	Dedicated wiring per function	Centralized splitting via `ASRT_AX_AX_SPLIT`
Flip-flop integration	Separate logic paths	Shared adapter stream feeding both output and flip-flop
Resource footprint	Higher (more connections)	Lower (reduced connection count)

The bundled approach reduces wiring overhead and makes the subapplication easier to integrate into larger systems.

Conclusion

The **SRT_RPC_FROM_Remote_QXA_OPC_Adapter** subapplication provides a compact, protocol-encapsulated solution for remote digital output control with flip-flop semantics. By bundling server/client functionality behind a single adapter connection and using a splitter to feed both the output stage and the state logic, it achieves a clean and efficient design. Its method-based RPC architecture and state write-back capability make it well suited for distributed automation environments where centralized control of remote I/O is required.

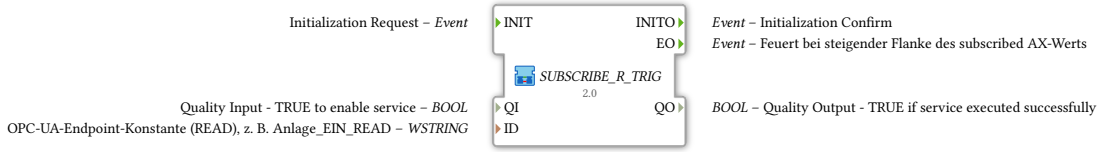
SUBSCRIBE_R_TRIG



• – ○ • – ○ • – ○ •

Introduction

`SUBSCRIBE_R_TRIG` monitors a `AX` value (BOOL) subscribed to via OPC UA for a rising edge and outputs it as a single event (`EO`). This allows a remotely set BOOL value to be used directly as a triggering event without having to process the level itself further.



Function Blocks (FBs) Used

Sub-Blocks: SUBSCRIBE_R_TRIG

- **Type**: SubAppType
- **Internal FBs Used**:
- **SUBSCRIBE**: `iec61499::net::SUBSCRIBE_1` — subscribes to the remote BOOL value (`ID`, e.g., `Anlage_EIN_READ`).

- **E_R_TRIG**: `iec61499::events::E_R_TRIG` — detects the rising edge of the subscribed value and fires `EO`.
- **How it Works**: Since version 2.0 the raw `SUBSCRIBE_1` and `E_R_TRIG` are wired directly (no duplicate edge memory any more): `SUBSCRIBE.IND` and `SUBSCRIBE.RD_1` go directly into `E_R_TRIG`, whose edge detection triggers the SubApp event `EO`.
- **Interface**: event input `INIT` (`EInit`), event outputs `INITO` (`EInit`) and `EO`; data inputs `QI` (`BOOL`, default `TRUE`) and `ID` (`WSTRING`); data output `QO` (`BOOL`).

Program Flow and Connections

1. `ID` ⟶ `SUBSCRIBE.ID` (Data connection, hidden).
2. `SUBSCRIBE.IND` ⟶ `E_R_TRIG.EI`; `SUBSCRIBE.RD_1` ⟶ `E_R_TRIG.QI`.
3. `E_R_TRIG.EO` ⟶ `EO` (SubApp event output).
4. `INIT` ⟶ `SUBSCRIBE.INIT`; `SUBSCRIBE.INITO` ⟶ `INITO`.
5. `QI` ⟶ `SUBSCRIBE.QI`; `SUBSCRIBE.QO` ⟶ `QO`.

Application Scenarios

- A remote OPC UA signal (e.g., “Plant ON”) should trigger a one-time event (e.g., start an initialization) instead of being processed as a continuous level.

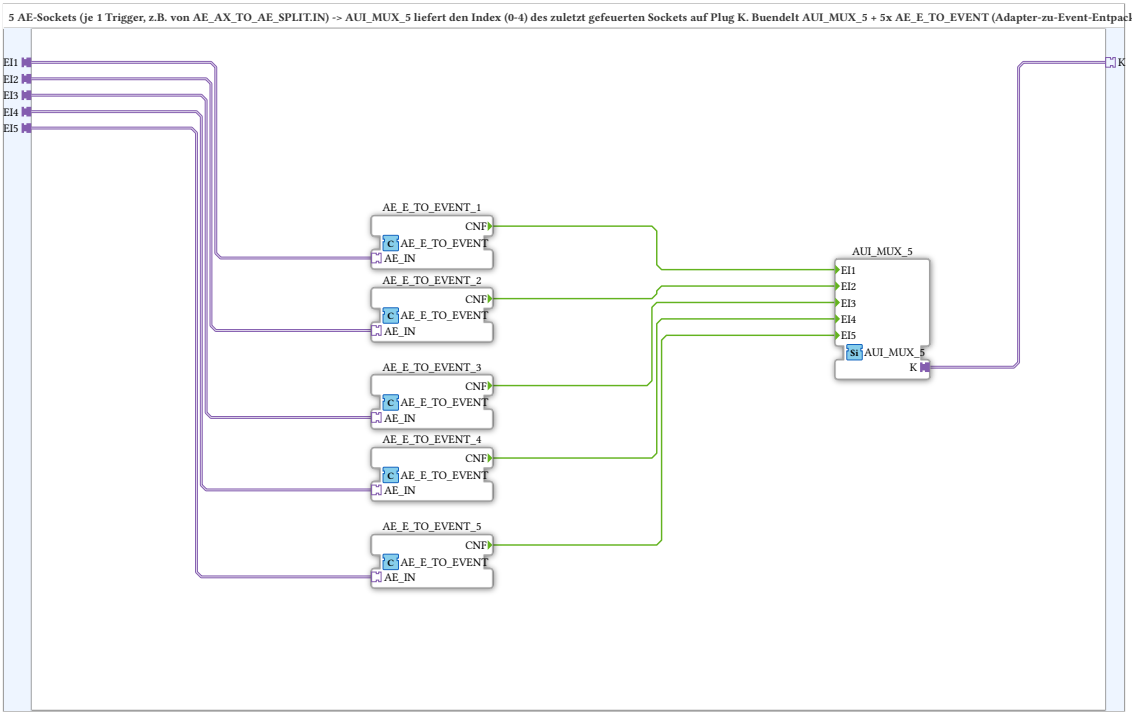
Summary

`SUBSCRIBE_R_TRIG` directly combines OPC UA Subscribe with edge detection and delivers a single event on the rising edge of the remote value.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

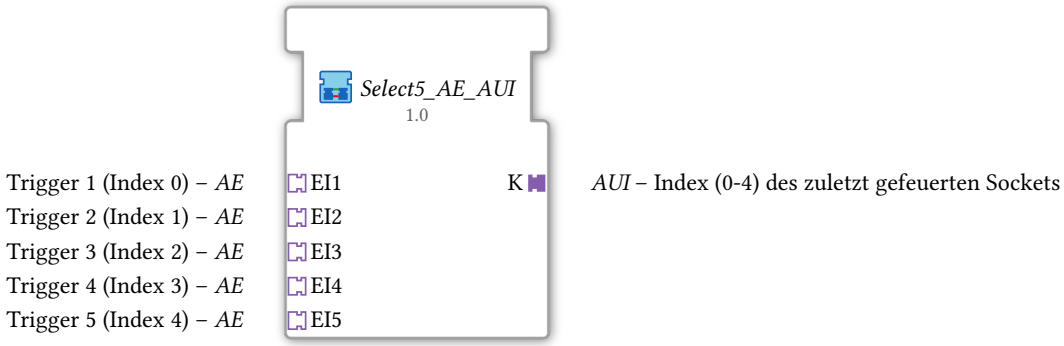
Select5_AE_AUI



• – ○ • – ○ • – ○ •

Introduction

`Select5_AE_AUI` accepts five `AE` sockets (one trigger each, e.g. from `AE_AX_TO_AE_SPLIT.IN`) and provides on plug `K` the index (0-4) of the socket that fired last. The block bundles `AUI_MUX_5` and five `AE_E_TO_EVENT` (adapter-to-event unpackers) into a reusable block.



Function Blocks (FBs) Used

Sub-blocks: Select5_AE_AUI

- **Type**: SubAppType
- **Internal FBs used**:
  - **AE_E_TO_EVENT_1** to **AE_E_TO_EVENT_5**: each `adapter::conversion::unidirectional::AE_E_TO_EVENT` - converts an `AE` adapter event into a normal event (`CNF`)
  - **AUI_MUX_5**: `adapter::events::unidirectional::AUI_MUX_5` - provides the index of the most recently active input on `K`

Interface Structure

Event Inputs / Event Outputs

None - events travel via the adapters.

Data Inputs / Data Outputs

No direct data ports.

Adapters

- **EI1** (socket, `adapter::types::unidirectional::AE`): Trigger 1 (index 0)
- **EI2** (socket, `AE`): Trigger 2 (index 1)
- **EI3** (socket, `AE`): Trigger 3 (index 2)
- **EI4** (socket, `AE`): Trigger 4 (index 3)
- **EI5** (socket, `AE`): Trigger 5 (index 4)
- **K** (plug, `adapter::types::unidirectional::AUI`): Index (0-4) of the socket that fired last

Program Flow and Connections

- `EI1`…`EI5` ⟶ `AE_E_TO_EVENT_1`…`AE_E_TO_EVENT_5` (`AE_IN`)
- `AE_E_TO_EVENT_n.CNF` ⟶ `AUI_MUX_5.EIn` (n = 1…5)
- `AUI_MUX_5.K` ⟶ `K`

Technical Details

- Index mapping: `EI1` ⟶ 0, `EI2` ⟶ 1, `EI3` ⟶ 2, `EI4` ⟶ 3, `EI5` ⟶ 4
- Version 1.0 (2026-09-12): encapsulated for the exclusive TECU speed source selection (5x `AE_AX_TO_AE_SPLIT.IN` as source)

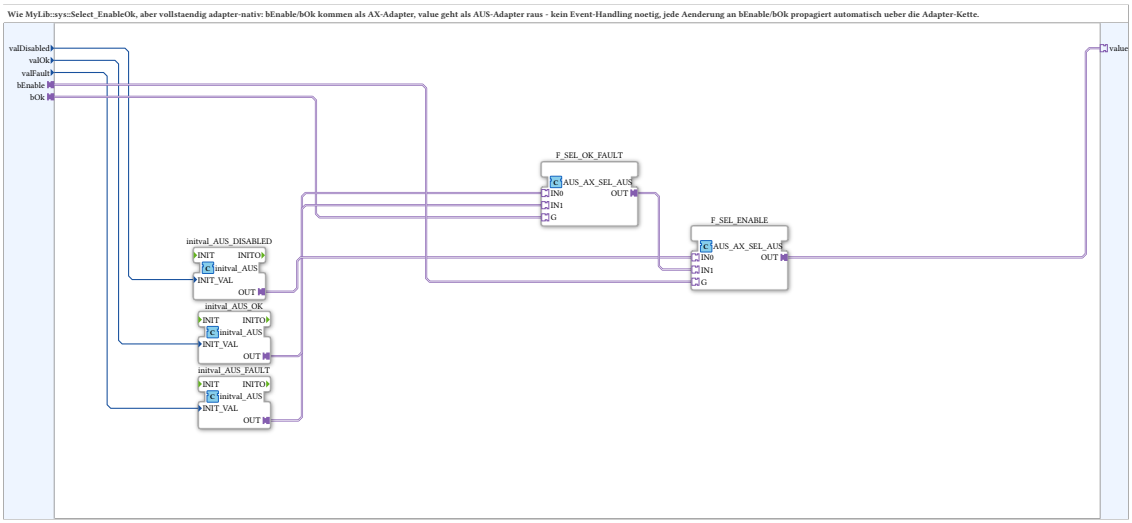
Application Scenarios

- Exclusive source selection: the most recently activated of five triggers determines the selected index (e.g. TECU speed source)

Summary

`Select5_AE_AUI` converts five `AE` triggers into the index of the most recently fired trigger as an `AUI` signal.

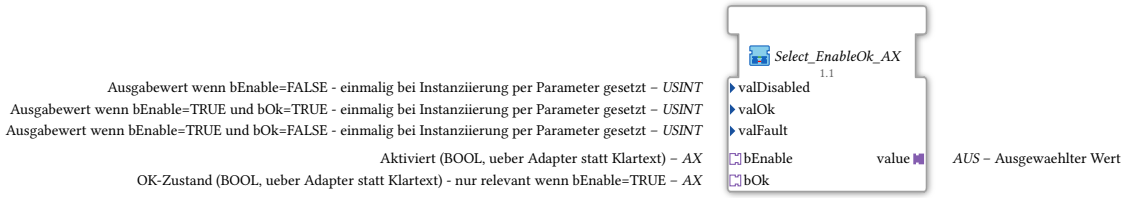
Select_EnableOk_AX



• – ○ • – ○ • – ○ •

Introduction

`Select_EnableOk_AX` is the adapter-native variant of `Select_EnableOk`: `bEnable`/`bOk` arrive as AX adapters (instead of plain BOOL), `value` leaves as an AUS adapter. No event handling needed - any change to `bEnable`/`bOk` propagates automatically through the adapter chain.



Function blocks used

- **initval_AUS_DISABLED** / **initval_AUS_OK** / **initval_AUS_FAULT** (`adapter::types::unidirectional::AUS::initval::initval_AUS`): feed the 3 parameter values (`valDisabled`/`valOk`/`valFault`) in as AUS adapters, once at instantiation.
- **F_SEL_OK_FAULT** (`adapter::iec61131::selection::AUS_AX_SEL_AUS`): selects between the fault/ok adapter based on `bOk` (adapter gate instead of an event selector).
- **F_SEL_ENABLE** (`adapter::iec61131::selection::AUS_AX_SEL_AUS`): selects between the disabled adapter/intermediate result based on `bEnable`.

Technical notes

- Two-stage like the event variant `Select_EnableOk`, but with no `REQ`/`CNF` at this level - the adapter connections themselves propagate every change.

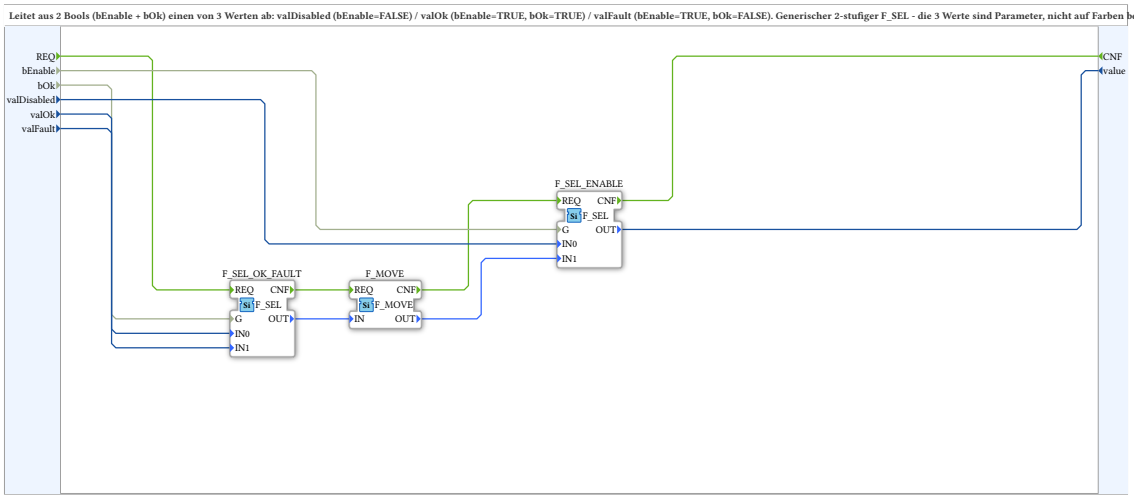
Summary

Fully adapter-native variant of `Select_EnableOk`, matching the GreenBlueBackground1_AX/GreenRedBackground1_AX family's style.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

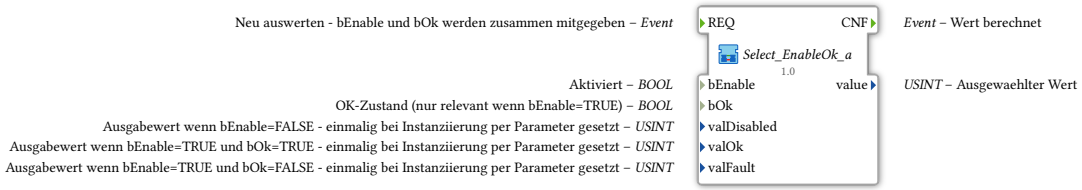
Select_EnableOk_a



• — ○ • — ○ • — ○ •

Introduction

`Select_EnableOk_a` derives one of three values from two `BOOL` values (`bEnable` and `bOk`): `valDisabled` (`bEnable = FALSE`), `valOk` (`bEnable = TRUE`, `bOk = TRUE`) or `valFault` (`bEnable = TRUE`, `bOk = FALSE`). It is a generic, two-stage `F_SEL`; the three values are parameters and not restricted to colors.



Interface Structure

Event Inputs

- **REQ**: Re-evaluate - `bEnable` and `bOk` are supplied together

Event Outputs

- **CNF**: Value calculated

Data Inputs

- **bEnable** (`BOOL`): Enabled
- **bOk** (`BOOL`): OK state (only relevant if `bEnable = TRUE`)
- **valDisabled** (`USINT`): Output value if `bEnable = FALSE`; set once at instantiation via parameter
- **valOk** (`USINT`): Output value if `bEnable = TRUE` and `bOk = TRUE`; set once at instantiation via parameter
- **valFault** (`USINT`): Output value if `bEnable = TRUE` and `bOk = FALSE`; set once at instantiation via parameter

Data Outputs

- **value** (`USINT`): Selected value

Adapters

This function block uses no adapters.

Function Blocks (FBs) Used

- **F_SEL_OK_FAULT**: `iec61131::selection::F_SEL` - selects with `G = bOk` between `IN0 = valFault` and `IN1 = valOk`
- **F_MOVE**: `iec61131::selection::F_MOVE` (`DataType=USINT`) - passes the result on

- **F_SEL_ENABLE**: `iec61131::selection::F_SEL` - selects with `G = bEnable` between `IN0 = valDisabled` and `IN1` = result of `F_MOVE`

Program Flow and Connections

1. `REQ` ⟶ `F_SEL_OK_FAULT.REQ` ⟶ `F_MOVE.REQ` ⟶ `F_SEL_ENABLE.REQ` ⟶ `CNF` (event chain).
2. `bOk` ⟶ `F_SEL_OK_FAULT.G`; `valFault` ⟶ `IN0`; `valOk` ⟶ `IN1`.
3. `F_SEL_OK_FAULT.OUT` ⟶ `F_MOVE.IN` ⟶ `F_SEL_ENABLE.IN1`.
4. `bEnable` ⟶ `F_SEL_ENABLE.G`; `valDisabled` ⟶ `F_SEL_ENABLE.IN0`.
5. `F_SEL_ENABLE.OUT` ⟶ `value`.

Technical Details

- The value table:

bEnable	bOk	value
FALSE	any	`valDisabled`
TRUE	TRUE	`valOk`
TRUE	FALSE	`valFault`

- The three values are `USINT` and can be e.g. color codes, but also any other state values.
- The AX variant of the block is `Select_EnableOk_AX`.

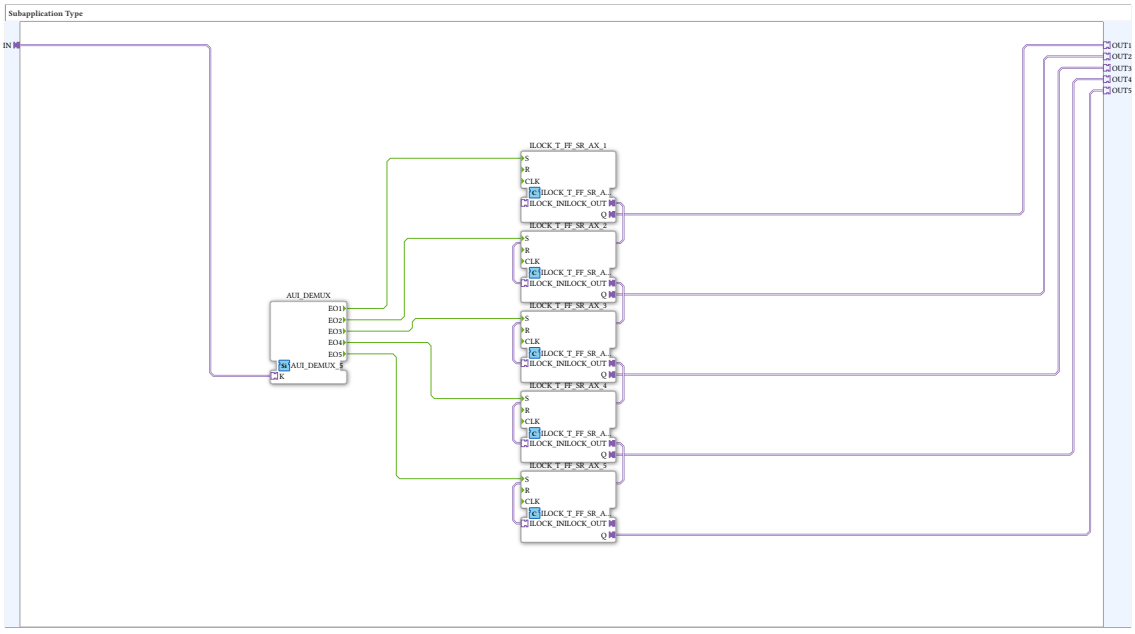
Application Scenarios

- Three-stage status display (off/ok/fault), e.g. color selection for a VT object

Summary

`Select_EnableOk_a` selects one of three parameterizable `USINT` values from enable and OK state.

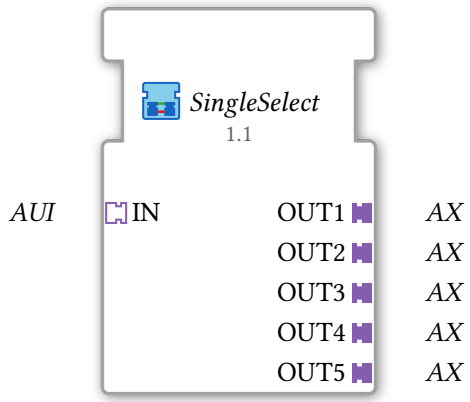
SingleSelect



• – ○ • – ○ • – ○ •

Introduction

`SingleSelect` is a single selection (radio button behaviour) with five outputs: a selection index at the AUI socket `IN` activates exactly one of the five AX outputs `OUT1`..`OUT5`; the others are reset in the process. The interlocking is performed by a chain of five `ILOCK_T_FF_SR_AX`.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **IN** (socket, `adapter::types::unidirectional::AUI`): selection index (UINT)
- **OUT1**..**OUT5** (plugs, `adapter::types::unidirectional::AX`): state of the five selection positions

Functionality

- `IN` ⟶ `AUI_DEMUX.K`: `AUI_DEMUX` (`adapter::events::unidirectional::AUI_DEMUX_5`) forwards the event depending on the index: `K = 0` ⟶ `EO1`, `K = 1` ⟶ `EO2`, `K = 2` ⟶ `EO3`, `K = 3` ⟶ `EO4`, `K = 4` ⟶ `EO5`.
- `AUI_DEMUX.EOn` ⟶ `ILOCK_T_FF_SR_AX_n.S` (n = 1..5): the selected flip-flop is set.
- `ILOCK_T_FF_SR_AX_n.Q` ⟶ `OUTn`: the state of each flip-flop appears at the associated output.
- `ILOCK_T_FF_SR_AX_n.ILOCK_OUT` ⟶ `ILOCK_T_FF_SR_AX_(n+1).ILOCK_IN` (n = 1..4): the five flip-flops are connected into a chain via their bidirectional `AE2` interlock adapters. When one element sets its output, it reports this through the chain and the other elements are reset (see `ILOCK_T_FF_SR_AX`).

Technical Details

- The block has no event or data ports; everything runs via adapters
- Version 1.1 tidied up the layout: `AUI_DEMUX_4` was renamed to `AUI_DEMUX` (the `_4` had no meaning)
- Valid indices are 0 to 4; the mapping between index and output is `K = n - 1` ↔ `OUTn`

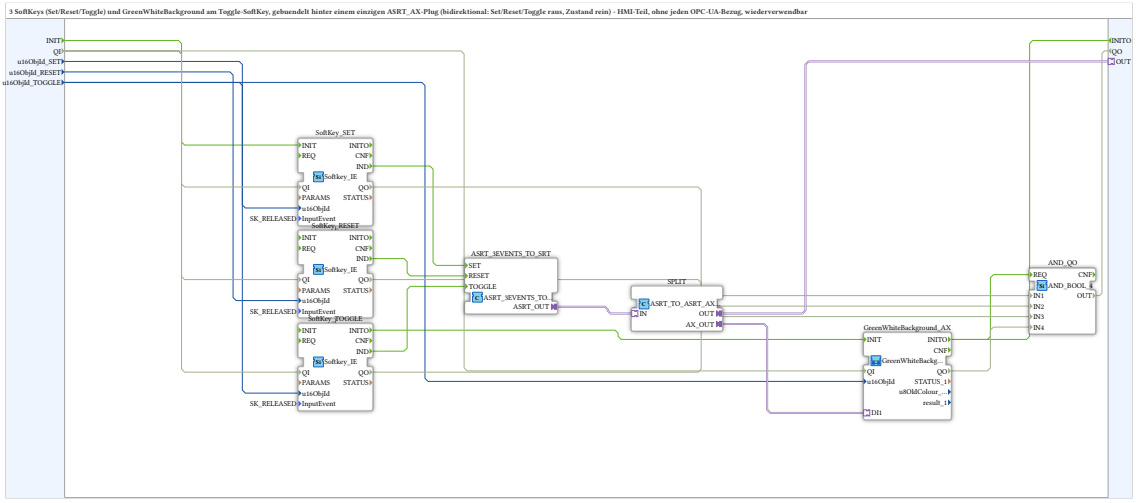
Application Scenarios

- Selection menus or operating mode selection in which only one of five options may be active at a time (e.g. via a VT selection list)

Conclusion

`SingleSelect` converts a selection index into a single selection with five AX outputs using a demultiplexer and an interlock chain of five `ILOCK_T_FF_SR_AX`.

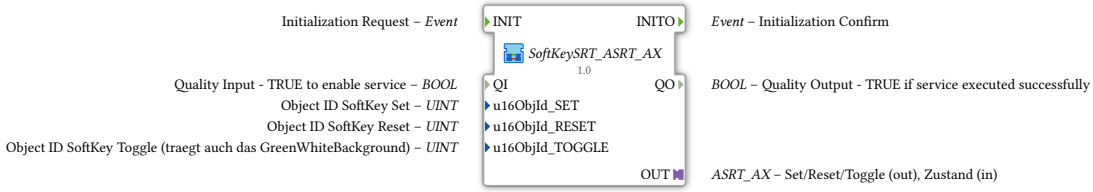
SoftKeySRT_ASRT_AX



• – ○ • – ○ • – ○ •

Introduction

`SoftKeySRT_ASRT_AX` bundles 3 softkeys (set/reset/toggle) and a `GreenWhiteBackground1_AX` (on the toggle softkey) behind a single `ASRT_AX` plug (bidirectional: set/reset/toggle go out, state comes in). A pure HMI block with no OPC-UA involvement - the SR+toggle counterpart of `SoftKeySR_ASR_AX`.



Function blocks used

- **SoftKey_SET / SoftKey_RESET / SoftKey_TOGGLE** (`isobus::UT::io::Softkey::Softkey_IE`, `InputEvent=SK_RELEASED`): the 3 physical softkeys.
- **ASRT_3EVENTS_TO_SRT** (`adapter::conversion::unidirectional::ASRT_3EVENTS_TO_SRT`): converts the 3 softkey events into an ASRT adapter event triple.
- **SPLIT** (`adapter::events::bidirectional::ASRT_TO_ASRT_AX_SPLIT`): splits the bidirectional `ASRT_AX` traffic to `OUT` and to the background element.
- **GreenWhiteBackground_AX** (SubApp, type `MyLib::sys::GreenWhiteBackground1_AX`): status color on the toggle softkey.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_4`): combines the `QO` outputs of the inner blocks into `QO`.

Initialization and Quality

Interface (quality pattern): event input `INIT` and event output `INITO` (each `EInit`), data input `QI` (`BOOL`, default `TRUE`) and data output `QO` (`BOOL`). Initialization: `INIT` ⟶ `SoftKey_SET.INIT`; `SoftKey_SET.INITO` ⟶ `SoftKey_RESET.INIT`; `SoftKey_RESET.INITO` ⟶ `SoftKey_TOGGLE.INIT`; `SoftKey_TOGGLE.INITO` ⟶ `GreenWhiteBackground_AX.INIT`; `GreenWhiteBackground_AX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`. Quality: `QI` ⟶ `SoftKey_SET.QI`, `SoftKey_RESET.QI`, `SoftKey_TOGGLE.QI`, `GreenWhiteBackground_AX.QI`; `SoftKey_SET.QO`, `SoftKey_RESET.QO`, `SoftKey_TOGGLE.QO`, `GreenWhiteBackground_AX.QO` ⟶ `AND_QO.IN1` to `AND_QO.IN4` (in this order); `AND_QO.OUT` ⟶ `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_4`).

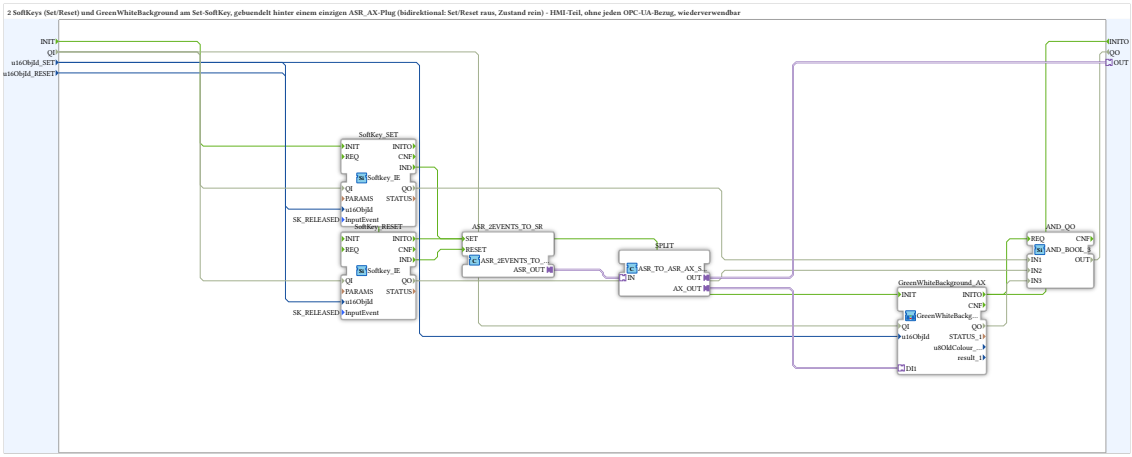
Summary

Reusable set/reset/toggle-with-status block, bundled behind an `ASRT_AX` adapter - the HMI counterpart of `Uebung_010e_PC_A_OPC_Adapter`/`Uebung_010e_PC_B_OPC_Adapter`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

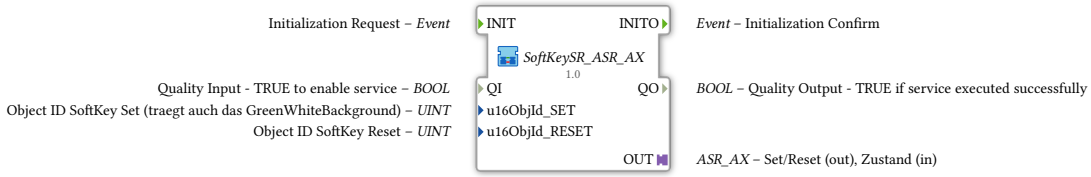
SoftKeySR_ASR_AX



• — ○ • — ○ • — ○ •

Introduction

`SoftKeySR_ASR_AX` bundles 2 softkeys (set/reset) and a `GreenWhiteBackground1_AX` (on the set softkey) behind a single `ASR_AX` plug (bidirectional: set/reset go out, state comes in). A pure HMI block with no OPC-UA involvement.



Interface

- **Event input**: `INIT` (`EInit`, Initialization Request)
- **Event output**: `INITO` (`EInit`, Initialization Confirm)
- **Data inputs**: `QI` (`BOOL`, default `TRUE`, Quality Input), `u16ObjId_SET`, `u16ObjId_RESET` (`UINT`, default `ID_NULL`)
- **Data output**: `QO` (`BOOL`, Quality Output)
- **Plug**: `OUT` (`adapter::types::bidirectional::ASR_AX`)

Function blocks used

- **SoftKey_SET / SoftKey_RESET** (`isobus::UT::io::Softkey::Softkey_IE`, `InputEvent=SK_RELEASED`): the two physical softkeys.
- **ASR_2EVENTS_TO_SR** (`adapter::conversion::unidirectional::ASR_2EVENTS_TO_SR`): converts the 2 softkey events into an ASR adapter event pair.
- **SPLIT** (`adapter::events::bidirectional::ASR_TO_ASR_AX_SPLIT`): splits the bidirectional `ASR_AX` traffic to `OUT` and to the background element.
- **GreenWhiteBackground_AX** (SubApp, type `MyLib::sys::GreenWhiteBackground1_AX`): status color on the set softkey.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_3`): ANDs the `QO` outputs of `SoftKey_SET`, `SoftKey_RESET` and `GreenWhiteBackground_AX` into the block output `QO`.

Initialization and status

- Initialization chain: `INIT` ⟶ `SoftKey_SET.INIT`, `SoftKey_SET.INITO` ⟶ `SoftKey_RESET.INIT`, `SoftKey_RESET.INITO` ⟶ `GreenWhiteBackground_AX.INIT`, `GreenWhiteBackground_AX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`
- `QI` ⟶ `SoftKey_SET.QI`, `SoftKey_RESET.QI` and `GreenWhiteBackground_AX.QI`

- `SoftKey_SET.QO` ⟶ `AND_QO.IN1`, `SoftKey_RESET.QO` ⟶ `AND_QO.IN2`, `GreenWhiteBackground_AX.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`

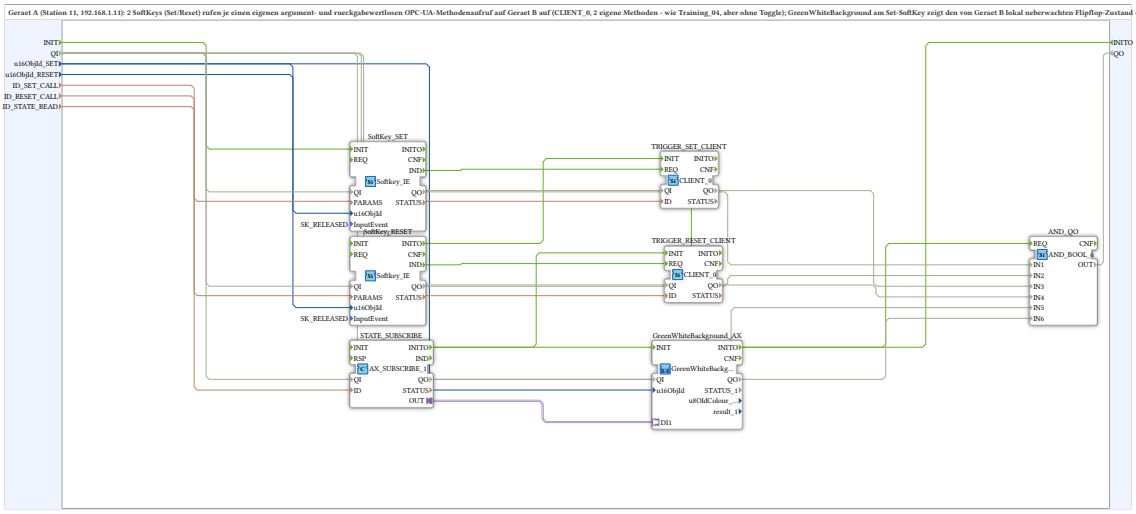
Summary

Reusable set/reset-with-status block, bundled behind an `ASR_AX` adapter - the HMI counterpart of `SoftKeySR_PC_A_OPC_Adapter`/`SoftKeySR_PC_B_OPC_Adapter`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

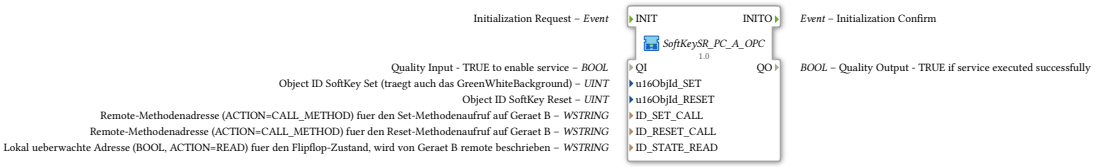
SoftKeySR_PC_A_OPC



• – ○ • – ○ • – ○ •

Introduction

`SoftKeySR_PC_A_OPC` is the device-A side (station 11, 192.168.1.11) of a PC-to-PC OPC-UA set/reset pattern: 2 softkeys (set/reset) each call their own argument- and return-value-less OPC-UA method on device B (`CLIENT_0`, 2 distinct methods - like Training_04, but without toggle). `GreenWhiteBackground1_AX` on the set softkey shows the flip-flop state locally monitored from device B. “SUB style”: the protocol lives in the `MyLib::sys` composite, not in the device’s resource - counterpart: `SoftKeySR_PC_B_OPC`.



Function blocks used

- **SoftKey_SET / SoftKey_RESET** (`isobus::UT::io::Softkey::Softkey_IE`): the two physical softkeys.
- **TRIGGER_SET_CLIENT / TRIGGER_RESET_CLIENT** (`iec61499::net::CLIENT_0`): each calls its own argument-less remote method on device B (`ID_SET_CALL`/`ID_RESET_CALL`).
- **STATE_SUBSCRIBE** (`adapter::net::AX_SUBSCRIBE_1`): subscribes to the flip-flop state under `ID_STATE_READ`.
- **GreenWhiteBackground_AX** (SubApp, type `MyLib::sys::GreenWhiteBackground1_AX`): displays the state at the set softkey.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_6`): combines the `QO` outputs of the six blocks above into the SubApp output `QO`.

Initialization and quality

- Events: `INIT` (EInit, input) and `INITO` (EInit, output, comes from `GreenWhiteBackground_AX.INITO`).

- Data input `QI` (BOOL, default `TRUE`) and data output `QO` (BOOL); the internal FBs no longer carry a fixed `QI=TRUE`.
- Initialization chain: `INIT` → `SoftKey_SET.INIT` → `SoftKey_RESET.INIT` → `TRIGGER_SET_CLIENT.INIT` → `TRIGGER_RESET_CLIENT.INIT` → `STATE_SUBSCRIBE.INIT` → `GreenWhiteBackground_AX.INIT` (each via `INITO`); `GreenWhiteBackground_AX.INITO` → `INITO` and → `AND_QO.REQ`.
- Quality: `QI` → `.QI` of all six blocks; their `QO` outputs → `AND_QO.IN1` to `IN6` (`SoftKey_SET`, `SoftKey_RESET`, `TRIGGER_SET_CLIENT`, `TRIGGER_RESET_CLIENT`, `STATE_SUBSCRIBE`, `GreenWhiteBackground_AX`); `AND_QO.OUT` → `QO`.

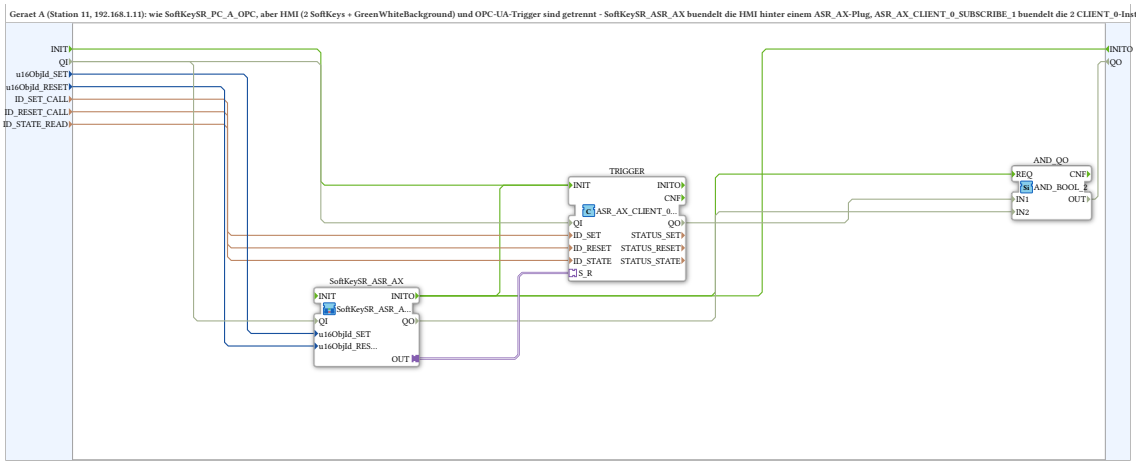
Summary

Device-A side of a set/reset RPC pattern with 2 distinct method calls (instead of one string parameter) and state feedback.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

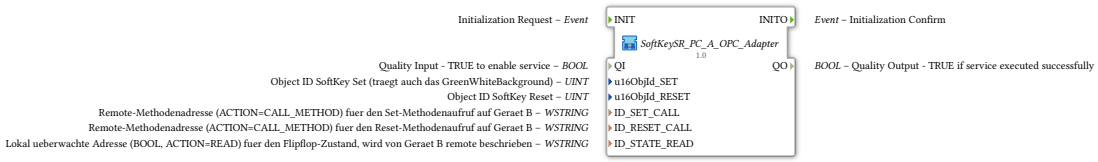
SoftKeySR_PC_A_OPC_Adapter



• — ○ • — ○ • — ○ •

Introduction

`SoftKeySR_PC_A_OPC_Adapter` is the adapter-bundled variant of `SoftKeySR_PC_A_OPC` (device A, station 11): HMI (2 softkeys + `GreenWhiteBackground`) and OPC-UA trigger are separated. `SoftKeySR_ASR_AX` bundles the HMI behind an `ASR_AX` plug; `ASR_AX_CLIENT_0_SUBSCRIBE_1` bundles the 2 `CLIENT_0` instances + `AX_SUBSCRIBE_1` behind an `ASR_AX` socket. Counterpart: `SoftKeySR_PC_B_OPC_Adapter`.



Function blocks used

- **SoftKeySR_ASR_AX** (SubApp, type `MyLib::sys::SoftKeySR_ASR_AX`): 2 softkeys + status display, bundled behind an `ASR_AX` plug.
- **TRIGGER** (`adapter::net::ASR_AX_CLIENT_0_SUBSCRIBE_1`): bundles 2 method calls (`ID_SET_CALL`/`ID_RESET_CALL`) and the state subscription (`ID_STATE_READ`) behind a single `ASR_AX` socket.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): combines `TRIGGER.QO` and `SoftKeySR_ASR_AX.QO` into the output `QO`.

Initialization and quality: The SubApp has the events `INIT`/`INITO` as well as `QI` (BOOL, initial value `TRUE`) and `QO` (BOOL). Init chain: `INIT` → `TRIGGER.INIT`, `TRIGGER.INITO` → `SoftKeySR_ASR_AX.INIT`, `SoftKeySR_ASR_AX.INITO` → `INITO` and `AND_QO.REQ`. `QI` is passed on to `TRIGGER.QI` and

`SoftKeySR_ASR_AX.QI`; `TRIGGER.QO` ⟶ `AND_QO.IN1`, `SoftKeySR_ASR_AX.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`.

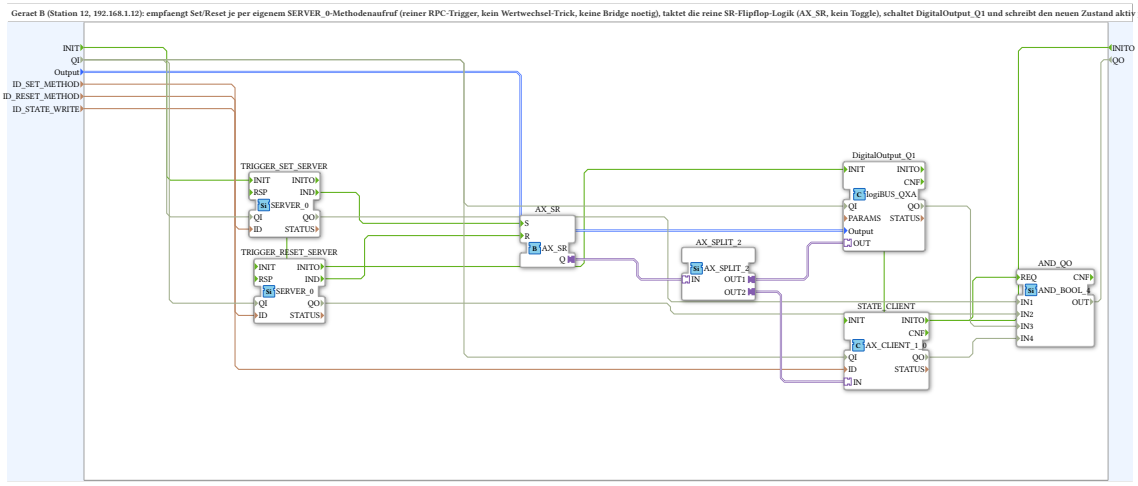
Summary

Adapter-bundled variant of `SoftKeySR_PC_A_OPC`: HMI and network protocol cleanly separated, reusable via `SoftKeySR_ASR_AX`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

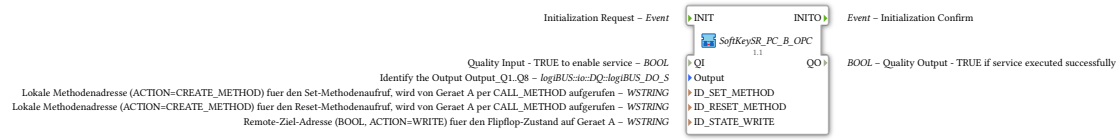
SoftKeySR_PC_B_OPC



• — ○ • — ○ • — ○ •

Introduction

`SoftKeySR_PC_B_OPC` is the device-B side (station 12, 192.168.1.12): receives set/reset each via its own `SERVER_0` method call (a pure RPC trigger, no value-change trick, no bridge needed), clocks the pure SR flip-flop logic (`AX_SR`, no toggle), drives `DigitalOutput_Q1`, and actively writes the new state back to device A via `AX_CLIENT_1_0`. Counterpart: `SoftKeySR_PC_A_OPC`.



Function blocks used

- **TRIGGER_SET_SERVER / TRIGGER_RESET_SERVER** (`iec61499::net::SERVER_0`): each receives its own method call (`ID_SET_METHOD`/`ID_RESET_METHOD`).
- **AX_SR** (`adapter::events::unidirectional::AX_SR`): pure set/reset flip-flop logic, no toggle.
- **AX_SPLIT_2** (`adapter::events::unidirectional::AX_SPLIT_2`): splits the state to the physical output + the echo.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physical output.
- **STATE_CLIENT** (`adapter::net::AX_CLIENT_1_0`): actively writes the state to device A under `ID_STATE_WRITE`.

Initialization and quality

The SubApp has the event input `INIT`, the event output `INITO` (both `EInit`), the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). Initialization runs as a chain: `INIT` ⟶ `TRIGGER_SET_SERVER.INIT`, `TRIGGER_SET_SERVER.INITO` ⟶ `TRIGGER_RESET_SERVER.INIT`, `TRIGGER_RESET_SERVER.INITO` ⟶ `DigitalOutput_Q1.INIT`, `DigitalOutput_Q1.INITO` ⟶ `STATE_CLIENT.INIT`, `STATE_CLIENT.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`). `QI` is passed to `TRIGGER_SET_SERVER`, `TRIGGER_RESET_SERVER`, `DigitalOutput_Q1` and `STATE_CLIENT`. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`), which combines the `QO` of these four instances.

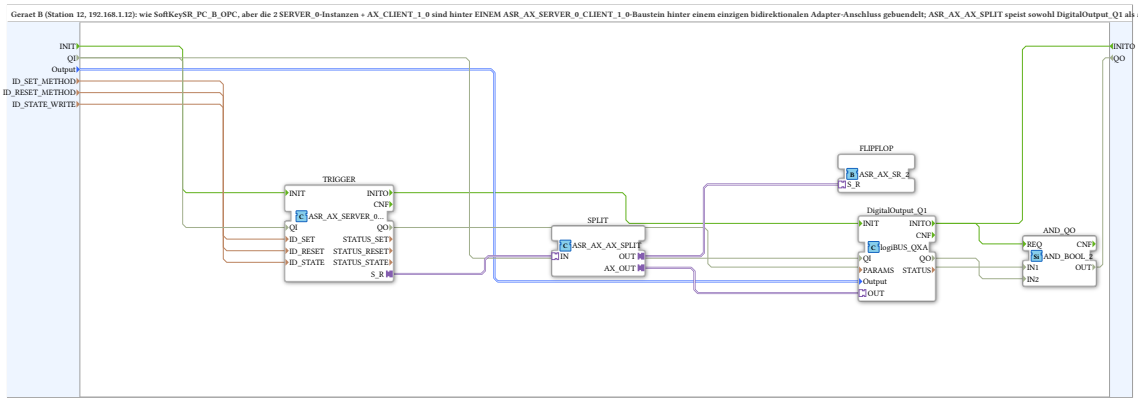
Summary

Device-B side: receives set/reset via RPC, holds the pure SR flip-flop logic, and actively reports the state back.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

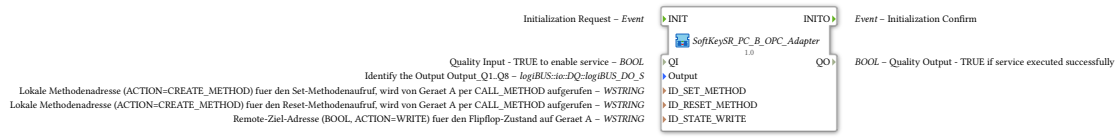
SoftKeySR_PC_B_OPC_Adapter



• – ○ • – ○ • – ○ •

Introduction

`SoftKeySR_PC_B_OPC_Adapter` is the adapter-bundled variant of `SoftKeySR_PC_B_OPC` (device B, station 12): the 2 `SERVER_0` instances + `AX_CLIENT_1_0` are bundled behind a SINGLE `ASR_AX_SERVER_0_CLIENT_1_0` block behind one bidirectional adapter connection; `ASR_AX_AX_SPLIT` feeds both `DigitalOutput_Q1` and the pure SR flip-flop logic (`ASR_AX_SR_2`, no toggle). Counterpart: `SoftKeySR_PC_A_OPC_Adapter`.



Function blocks used

- **TRIGGER** (`adapter::net::ASR_AX_SERVER_0_CLIENT_1_0`): bundles server reception (`ID_SET_METHOD`/`ID_RESET_METHOD`) and state feedback (`ID_STATE_WRITE`).
- **SPLIT** (`adapter::events::bidirectional::ASR_AX_AX_SPLIT`): splits to the physical output + the flip-flop logic.
- **FLIPFLOP** (`adapter::events::bidirectional::ASR_AX_SR_2`): pure SR flip-flop logic as an adapter block, no toggle.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physical output.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): combines `TRIGGER.QO` and `DigitalOutput_Q1.QO` into the `QO` of the SubApp.

Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): `INIT` ⟶ `TRIGGER.INIT`; `TRIGGER.INITO` ⟶ `DigitalOutput_Q1.INIT`; `DigitalOutput_Q1.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`).
- **Data input `QI`** (`BOOL`, default `TRUE`) ⟶ `TRIGGER.QI` and `DigitalOutput_Q1.QI`.
- **Data output `QO`** (`BOOL`): `TRIGGER.QO` ⟶ `AND_QO.IN1`; `DigitalOutput_Q1.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

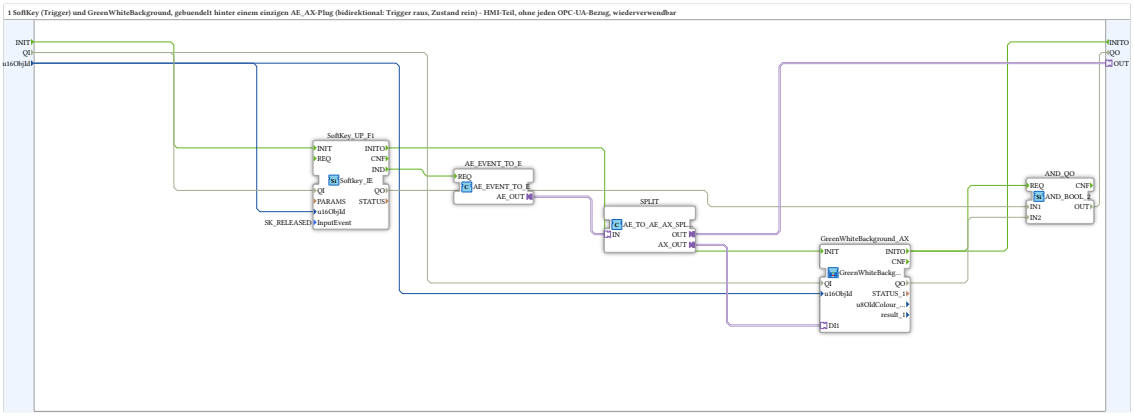
Summary

Adapter-bundled variant of `SoftKeySR_PC_B_OPC`: network protocol and SR flip-flop logic as reusable adapter blocks.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

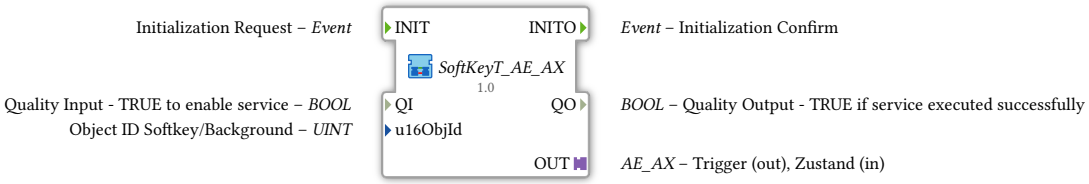
SoftKeyT_AE_AX



• – ○ • – ○ • – ○ •

Introduction

`SoftKeyT_AE_AX` bundles 1 SoftKey (trigger) and a `GreenWhiteBackground1_AX` behind a single `AE_AX` plug (bidirectional: trigger goes out, state comes in). A pure HMI block with no OPC-UA involvement - reusable in any network that needs a single trigger softkey with a state display.



Function blocks used

- **SoftKey_UP_F1** (`isobus::UT::io::Softkey::Softkey_IE`, `InputEvent=SK_RELEASED`): physical softkey.
- **AE_EVENT_TO_E** (`adapter::conversion::unidirectional::AE_EVENT_TO_E`): converts the softkey event into an AE adapter event.
- **GreenWhiteBackground_AX** (SubApp, type `MyLib::sys::GreenWhiteBackground1_AX`): status color on the same object.
- **SPLIT** (`adapter::events::bidirectional::AE_TO_AE_AX_SPLIT`): splits the bidirectional `AE_AX` traffic to the external plug `OUT` and internally to the background element.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): combines the `QO` outputs of the inner blocks into `QO`.

Program flow and connections

`SoftKey_UP_F1.IND -> AE_EVENT_TO_E.REQ -> SPLIT.IN -> OUT` (outward) and `SPLIT.AX_OUT -> GreenWhiteBackground_AX.DI1` (state display).

Initialization and Quality

Interface (quality pattern): event input `INIT` and event output `INITO` (each `EInit`), data input `QI` (`BOOL`, default `TRUE`) and data output `QO` (`BOOL`). Initialization: `INIT` ⟶ `SoftKey_UP_F1.INIT`; `SoftKey_UP_F1.INITO` ⟶ `GreenWhiteBackground_AX.INIT`; `GreenWhiteBackground_AX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`. Quality: `QI` ⟶ `SoftKey_UP_F1.QI`, `GreenWhiteBackground_AX.QI`; `SoftKey_UP_F1.QO`, `GreenWhiteBackground_AX.QO` ⟶ `AND_QO.IN1` to `AND_QO.IN2` (in this order); `AND_QO.OUT` ⟶ `QO` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_2`).

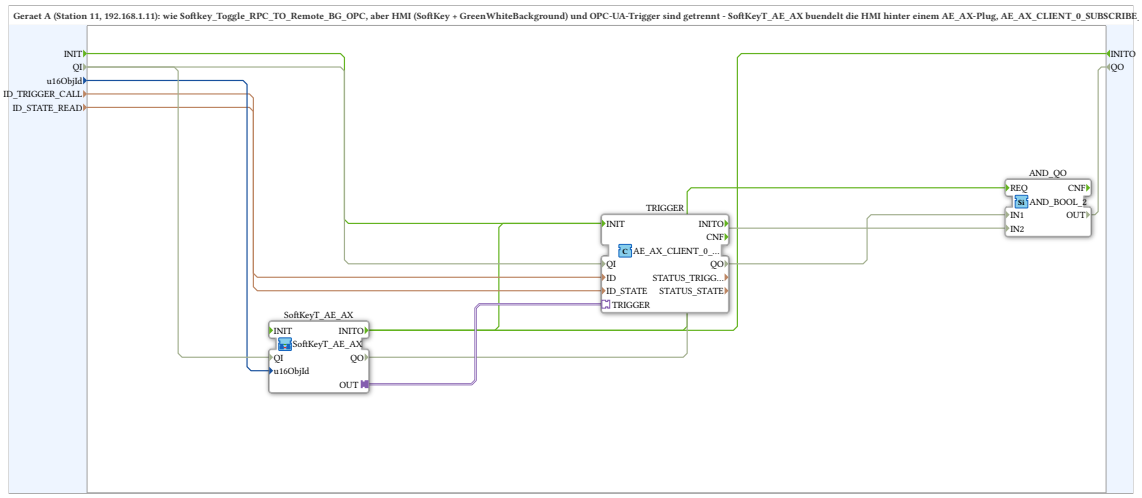
Summary

Reusable single-key trigger-with-status block, bundled behind an `AE_AX` adapter - the HMI counterpart of the PC_A/PC_B OPC network blocks like `SoftKeyT_PC_A_OPC_Adapter`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

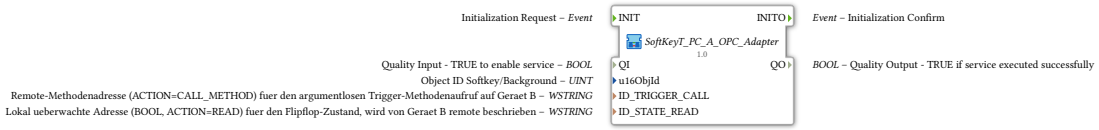
SoftKeyT_PC_A_OPC_Adapter



• – ○ • – ○ • – ○ •

Introduction

`SoftKeyT_PC_A_OPC_Adapter` is the adapter-bundled variant of `Uebung_010d_PC_A_OPC` (device A, station 11): HMI (softkey + `GreenWhiteBackground`) and OPC-UA trigger are separated. `SoftKeyT_AE_AX` bundles the HMI behind an `AE_AX` plug; `AE_AX_CLIENT_0_SUBSCRIBE_1` bundles `CLIENT_0 + AX_SUBSCRIBE_1` behind an `AE_AX` socket. "SUB style": the protocol still lives in the `MyLib::sys` composite. Counterpart: `SoftKeyT_PC_B_OPC_Adapter`.



Interface

- **Event input**: `INIT` (`EInit`, Initialization Request)
- **Event output**: `INITO` (`EInit`, Initialization Confirm)
- **Data inputs**: `QI` (`BOOL`, default `TRUE`, Quality Input), `u16ObjId`, `ID_TRIGGER_CALL`, `ID_STATE_READ`
- **Data output**: `QO` (`BOOL`, Quality Output)

Function blocks used

- **SoftKeyT_AE_AX** (SubApp, type `MyLib::sys::SoftKeyT_AE_AX`): softkey + status display, bundled behind an `AE_AX` plug.
- **TRIGGER** (`adapter::net::AE_AX_CLIENT_0_SUBSCRIBE_1`): bundles the method call (`ID_TRIGGER_CALL`) and state subscription (`ID_STATE_READ`) behind a single `AE_AX` socket.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): ANDs `TRIGGER.QO` and `SoftKeyT_AE_AX.QO` into the block output `QO`.

Program flow and connections

`SoftKeyT_AE_AX.OUT` -> `TRIGGER.TRIGGER` (bidirectional: trigger out, state back to the HMI SubApp).

Initialization chain: `INIT` -> `TRIGGER.INIT`, `TRIGGER.INITO` -> `SoftKeyT_AE_AX.INIT`, `SoftKeyT_AE_AX.INITO` -> `INITO` and -> `AND_QO.REQ`. `QI` -> `TRIGGER.QI` and `SoftKeyT_AE_AX.QI`; `TRIGGER.QO` -> `AND_QO.IN1`, `SoftKeyT_AE_AX.QO` -> `AND_QO.IN2`, `AND_QO.OUT` -> `QO`.

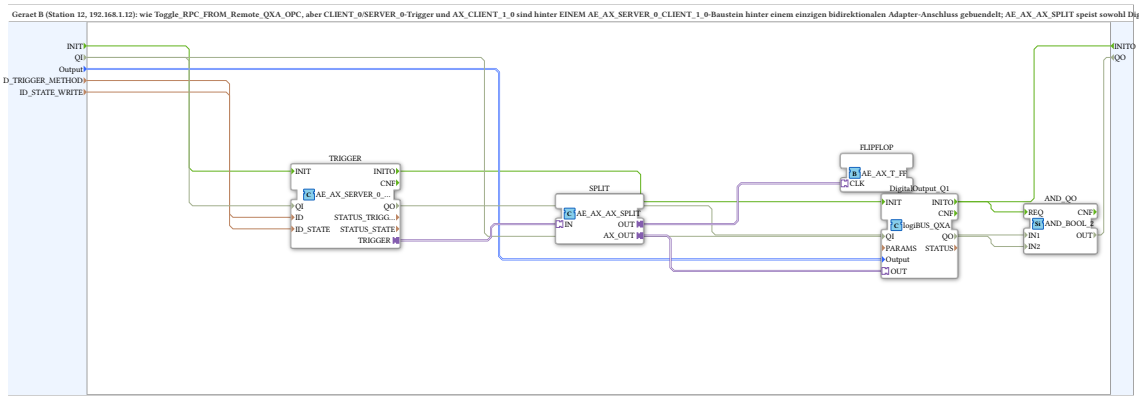
Summary

Adapter-bundled variant of `Uebung_010d_PC_A_OPC`: the same function, but HMI and network protocol are cleanly separated into 2 reusable blocks instead of being wired in a single composite.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

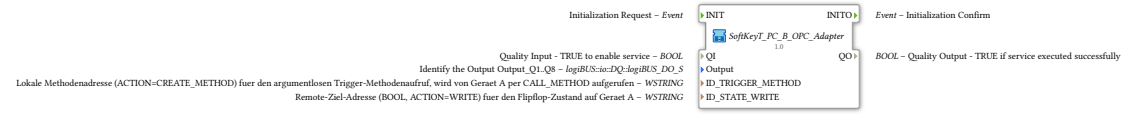
SoftKeyT_PC_B_OPC_Adapter



• – ○ • – ○ • – ○ •

Introduction

`SoftKeyT_PC_B_OPC_Adapter` is the adapter-bundled variant of `Uebung_010d_PC_B_OPC` (device B, station 12): the `CLIENT_0/SERVER_0` trigger and `AX_CLIENT_1_0` are bundled behind a SINGLE `AE_AX_SERVER_0_CLIENT_1_0` block behind one bidirectional adapter connection; `AE_AX_AX_SPLIT` feeds both `DigitalOutput_Q1` and the toggle flip-flop logic (`AE_AX_T_FF`). Counterpart: `SoftKeyT_PC_A_OPC_Adapter`.



Function blocks used

- **TRIGGER** (`adapter::net::AE_AX_SERVER_0_CLIENT_1_0`): bundles server reception (`ID_TRIGGER_METHOD`) and state feedback (`ID_STATE_WRITE`).
- **SPLIT** (`adapter::events::bidirectional::AE_AX_AX_SPLIT`): splits the trigger to the physical output + the flip-flop logic.
- **FLIPFLOP** (`adapter::events::bidirectional::AE_AX_T_FF`): toggle flip-flop logic as an adapter block.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physical output.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): combines `TRIGGER.QO` and `DigitalOutput_Q1.QO` into the SubApp output `QO`.

Initialization and quality

- Events: `INIT` (EInit, input) and `INITO` (EInit, output, comes from `DigitalOutput_Q1.INITO`).
- Data input `QI` (BOOL, default `TRUE`) and data output `QO` (BOOL); `TRIGGER` and `DigitalOutput_Q1` no longer carry a fixed `QI=TRUE`.
- Initialization: `INIT` ⟶ `TRIGGER.INIT`, `TRIGGER.INITO` ⟶ `DigitalOutput_Q1.INIT`, `DigitalOutput_Q1.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
- Quality: `QI` ⟶ `TRIGGER.QI` and `DigitalOutput_Q1.QI`; `TRIGGER.QO` ⟶ `AND_QO.IN1`, `DigitalOutput_Q1.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

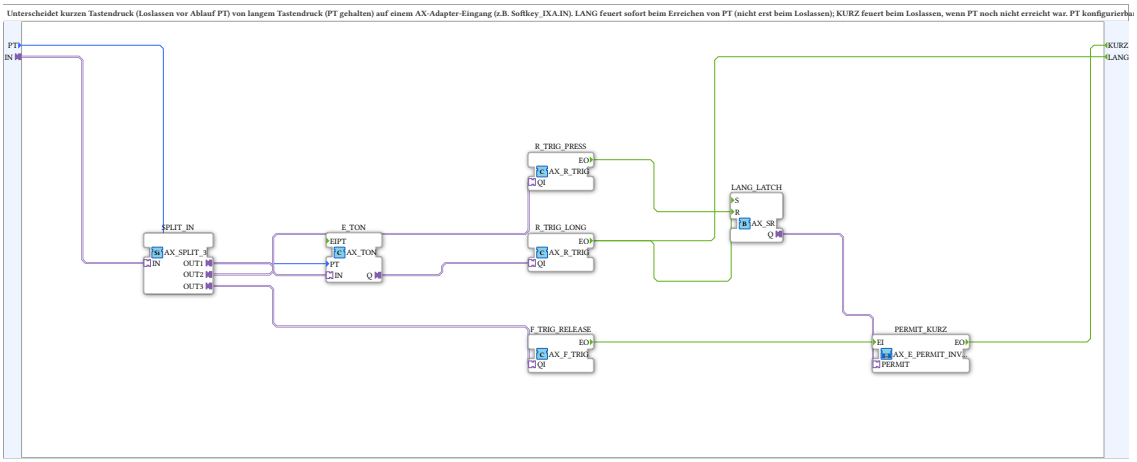
Summary

Adapter-bundled variant of `Uebung_010d_PC_B_OPC`: network protocol and flip-flop logic as reusable adapter blocks instead of individually wired FBs.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SoftKey_Kurz_Lang



• — ○ • — ○ • — ○ •

Introduction

`SoftKey_Kurz_Lang` distinguishes a **short** key press (released before `PT` elapses) from a **long** key press (`PT` held) on an `AX` adapter input (e.g. `Softkey_IXA.IN`). `LANG` fires immediately when `PT` is reached (not only on release); `KURZ` fires on release if `PT` had not yet been reached. `PT` is configurable (default 5 s).



Interface Structure

Event Inputs

None.

Event Outputs

- **KURZ**: short press confirmed (released before `PT` elapses)
- **LANG**: long press confirmed (`PT` reached, fires immediately, not only on release)

Data Inputs

- **PT** (`TIME`): hold time for “long”, default `T#5s`

Data Outputs

None.

Adapters

- **IN** (socket, `adapter::types::unidirectional::AX`): raw key state (e.g. `Softkey_IXA.IN`)

Function Blocks (FBs) Used

- **`SPLIT_IN`** (`adapter::events::unidirectional::AX_SPLIT_3`): distributes `IN` to three targets
- **`E_TON`** (`adapter::events::unidirectional::timers::AX_TON`): on-delay with `PT`
- **`R_TRIG_LONG`** (`adapter::events::unidirectional::AX_R_TRIG`): rising edge of `E_TON.Q`
- **`R_TRIG_PRESS`** (`adapter::events::unidirectional::AX_R_TRIG`): rising edge of `IN` (start of key press)
- **`LANG_LATCH`** (`adapter::events::unidirectional::AX_SR`): remembers that `LANG` has already fired in this cycle
- **`F_TRIG_RELEASE`** (`adapter::events::unidirectional::AX_F_TRIG`): falling edge of `IN` (release)

- **PERMIT_KURZ** (SubApp `MyLib::sys::AX_E_PERMIT_INVERT_1`, AX_E_PERMIT_INVERT_1): lets the release event pass only if the permit is FALSE

Functionality

- `IN` ⟶ `SPLIT_IN.IN`; `SPLIT_IN.OUT1` ⟶ `E_TON.IN`; `PT` ⟶ `E_TON.PT`
- `SPLIT_IN.OUT2` ⟶ `R_TRIG_PRESS.QI`; `SPLIT_IN.OUT3` ⟶ `F_TRIG_RELEASE.QI`
- `E_TON.Q` ⟶ `R_TRIG_LONG.QI`
- `R_TRIG_LONG.EO` ⟶ `LANG` and `LANG_LATCH.S`
- `R_TRIG_PRESS.EO` ⟶ `LANG_LATCH.R`
- `F_TRIG_RELEASE.EO` ⟶ `PERMIT_KURZ.EI`; `LANG_LATCH.Q` ⟶ `PERMIT_KURZ.PERMIT`; `PERMIT_KURZ.EO` ⟶ `KURZ`

`E_TON` runs as long as `IN` is held. The rising edge of `E_TON.Q` (the moment `PT` is reached while the key is still held) triggers `LANG` immediately and sets `LANG_LATCH`. On release, `F_TRIG_RELEASE` detects the falling edge; `PERMIT_KURZ` only lets it pass if `LANG_LATCH.Q` is FALSE, i.e. `LANG` has not yet fired in this press cycle. At the start of the next key press (`R_TRIG_PRESS`) `LANG_LATCH` is reset.

Technical Details

- **Bugfix compared to version 1.0:** the check must not be made directly on `E_TON.Q`. `E_TON.Q` is reset to FALSE by the same `IN` edge change (release) that also triggers `F_TRIG_RELEASE` (`AX_TON`: no `IN` ⟶ no `Q`). Since both targets are fed by the same `IN` event, the execution order between them is not guaranteed; `E_TON.Q` would practically always already be FALSE at the time of the check, and `KURZ` would additionally have fired even after `LANG` had already fired.
- `LANG_LATCH` is completely decoupled from the reset timing (set while holding, reset only at the next key press) and makes the check robust.
- `PT` is configurable as a parameter.

Application Scenarios

- Dual-function key: short press for one function, long press (e.g. 5 s) for another, together with `Softkey_Web_IXA` for keys that should also be operable via the web client

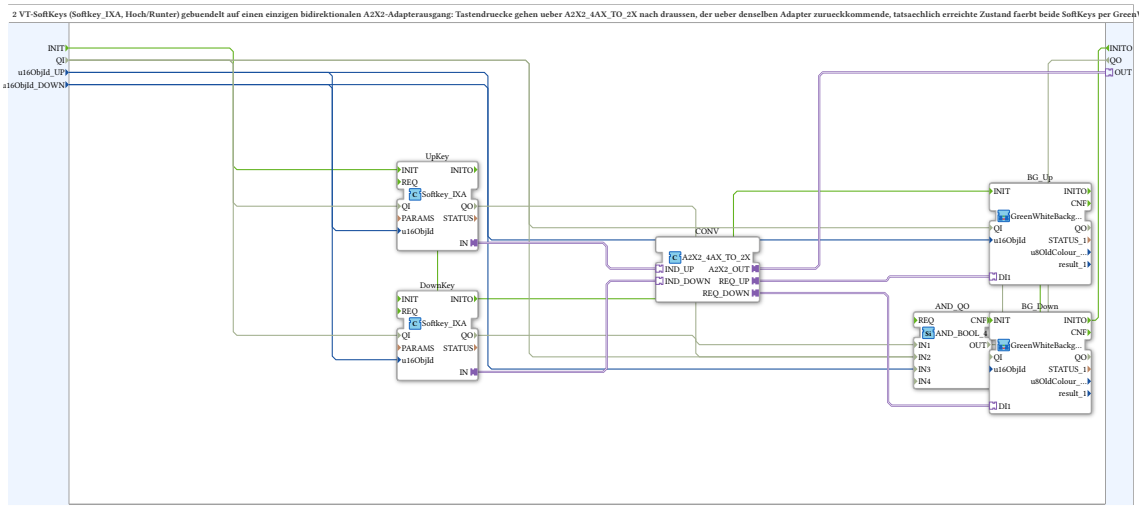
Summary

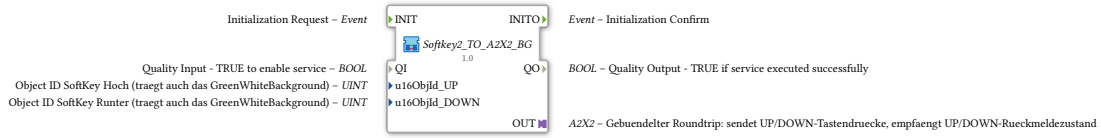
`SoftKey_Kurz_Lang` derives the separate events `KURZ` and `LANG` with a configurable hold time from an `AX` key state.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Softkey2_TO_A2X2_BG





• — ○ • — ○ • — ○ •

Introduction

`Softkey2_TO_A2X2_BG` is a composite subapplication designed to bundle two ISOagriLib softkey function blocks (one for "UP", one for "DOWN") into a single bidirectional A2X2 adapter interface. It also implements visual feedback for the softkeys by colouring them based on the actual state returned via the same adapter. This subapplication encapsulates the logic entirely within the composite, keeping the device resource simple and focused.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the init chain at `UpKey.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `BG_Down.INITO`, the end of the init chain

Data Inputs

- `QI` (BOOL, initial value `TRUE`): Quality input, enables the service; passed on to `UpKey`, `DownKey`, `BG_Up` and `BG_Down`.
- `u16ObjId_UP` (UINT): Object ID for the "UP" softkey. This ID is used both by the softkey logic and by the background colouring subapplication.
- `u16ObjId_DOWN` (UINT): Object ID for the "DOWN" softkey. This ID is used both by the softkey logic and by the background colouring subapplication.

Data Outputs

- `QO` (BOOL): Quality output, AND combination (`AND_QO`, `AND_BOOL_4`) of the `QO` outputs of `UpKey`, `DownKey`, `BG_Up` and `BG_Down`.

Adapters

- `OUT` (adapter::types::bidirectional::A2X2): A bidirectional adapter that bundles the sending of UP/DOWN key press events and the reception of the actual state feedback from the remote partner (e.g. a virtual terminal).

Functionality

The subapplication internally instantiates two `Softkey_IXA` function blocks (`UpKey` and `DownKey`) to handle the ISOagriLib softkey semantics. Each softkey receives its object ID from the corresponding data input. The key press events are forwarded via adapter connections to a conversion block (`A2X2_4AX_TO_2X`), which consolidates the two separate softkey streams into a single A2X2 adapter connection.

The same A2X2 adapter also carries the returned state (e.g. which softkey is actually pressed or released at the remote terminal). This state is fed back to two `GreenWhiteBackground1_AX` subapplications (`BG_Up` and `BG_Down`), each associated with one softkey. These subapplications colour the respective softkey background green (active) or white (inactive) based on the feedback received via the adapter.

Initialization and quality: `INIT` ⟶ `UpKey.INIT`, `UpKey.INITO` ⟶ `DownKey.INIT`, `DownKey.INITO` ⟶ `BG_Up.INIT`, `BG_Up.INITO` ⟶ `BG_Down.INIT`, `BG_Down.INITO` ⟶ `INITO` and `AND_QO.REQ`. `QI` is passed on to `UpKey`, `DownKey`, `BG_Up` and `BG_Down`; their `QO` outputs go in this order to `AND_QO.IN1` to `AND_QO.IN4` (`iec61131::booleanOperators::AND_BOOL_4`), `AND_QO.OUT` ⟶ `QO`.

Thus, the entire round trip — sending key presses and receiving state — is done through the single `OUT` adapter, and the visual status is updated accordingly.

Technical Features

- **Composite bundling**: Encapsulates two independent softkey channels into one adapter interface, reducing the external connection count.

- **ISOagriLib compliance**: Uses standard `Softkey_IXA` blocks, ensuring compatibility with ISO 11783 (ISOBUS) virtual terminal implementations.
- **Bidirectional communication**: The A2X2 adapter supports both sending and receiving, enabling state feedback.
- **Background colouring**: Uses `GreenWhiteBackground1_AX` subapplications to change softkey background based on the acknowledged state.
- **Parameterisable**: Object IDs are provided as inputs, allowing reuse for different softkey identifiers without internal modification.

State Overview

The subapplication does not maintain a finite state machine of its own. Instead, it relies on the internal `Softkey_IXA` blocks to handle the softkey state transitions (e.g. pressed, released). The `GreenWhiteBackground1_AX` subapplications react to the feedback from the adapter and colour the softkey background accordingly:

- **Green**: The softkey is currently recognised as active/pressed by the remote terminal.
- **White**: The softkey is inactive/released.

The state information flows continuously through the adapter, reflecting the latest acknowledged condition.

Application Scenarios

- **Agricultural machinery control**: A virtual terminal with multiple softkeys that must be grouped and communicated via a single A2X2 adapter for efficiency.
- **Remote terminal interaction**: When several softkeys need to be bundled into one adapter connection to reduce wiring or complexity in the device resource.
- **Status-aware softkeys**: Applications where the background colour of the softkey must match the actual operational state received from the remote partner.

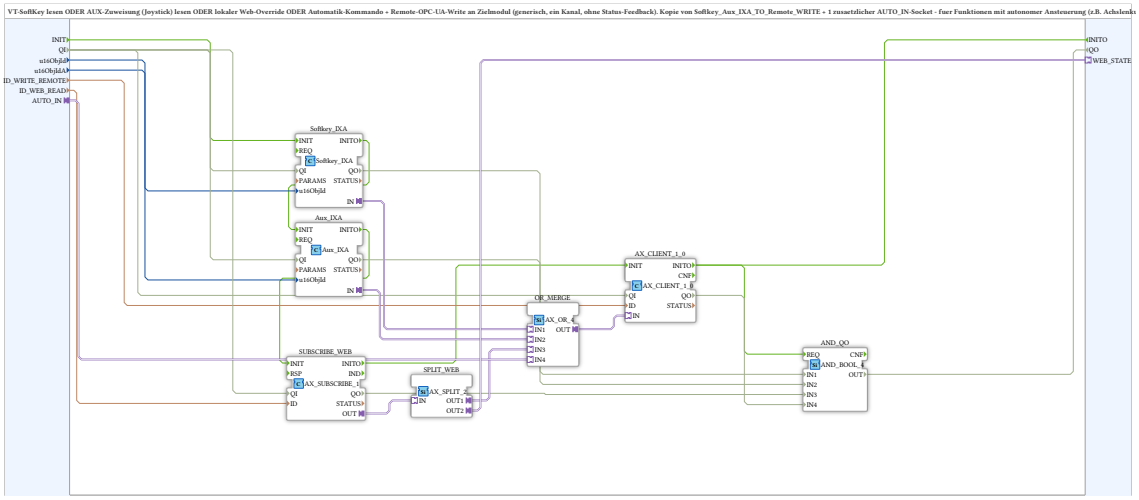
Comparison with Similar Blocks

- **Individual softkey adapters**: Each softkey would require its own adapter connection. `Softkey2_TO_A2X2_BG` consolidates two into one, reducing the number of connections and simplifying the resource design.
- **Non-feedback softkeys**: Some softkey blocks only send events without receiving state. This subapplication adds bidirectional capability, enabling accurate visual feedback.
- **Hard-coded softkeys**: In contrast to blocks that hard-code object IDs, this subapplication accepts them as inputs, making it flexible for reuse in different contexts.

Conclusion

`Softkey2_TO_A2X2_BG` provides a compact, reusable solution for bundling two ISOagriLib softkeys into a single bidirectional A2X2 adapter while incorporating state-based background colouring. Its design keeps the device resource clean and reduces the number of external connections, while fully supporting the required communication and visual feedback mechanisms. This subapplication is a valuable building block for ISOBUS-compliant virtual terminals that need efficient, status-aware softkey handling.

Softkey_Aux_Auto_IXA_TO_Remote_WRITE



• — ○ • — ○ • — ○ •

Introduction

`Softkey_Aux_Auto_IXA_TO_Remote_WRITE` reads a **VT SoftKey**, an **AUX assignment (joystick)**, a **local web override** or an **automation command** and writes the result via a remote OPC-UA write to a target module (generic, one channel, without status feedback). The block is a copy of `Softkey_Aux_IXA_TO_Remote_WRITE` with an additional socket `AUTO_IN` for functions with autonomous control (e.g. axle steering centre automation).

There is still exactly **one** publisher per channel on `ID_WRITE_REMOTE`, even though four instead of three sources are merged - no self-latch risk. For status feedback/background colour see the pre-wired combination `Softkey_Aux_Auto_IXA_TO_Remote_WRITE_BG_OPC`.



Function Blocks (FBs) Used

Sub-blocks: Softkey_Aux_Auto_IXA_TO_Remote_WRITE

- **Type**: SubAppType
- **Internal FBs used**:
  - **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA` - SoftKey level
  - **Aux_IXA**: `isobus::UT::io::Auxiliary::IN::Aux_IXA` - AUX level
  - **SUBSCRIBE_WEB**: `adapter::net::AX_SUBSCRIBE_1` - local subscribe for a web client
  - **SPLIT_WEB**: `adapter::events::unidirectional::AX_SPLIT_2` - distributes the web state to `OR_MERGE` and `WEB_STATE`
  - **OR_MERGE**: `adapter::booleanOperators::AX_OR_4` - OR combination of the four sources
  - **AX_CLIENT_1_0**: `adapter::net::AX_CLIENT_1_0` - remote write to the target module
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_4` - combines the `QO` of `Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB` and `AX_CLIENT_1_0`
- **Import**: `ID_NULL`

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; goes to `Softkey_IXA.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `AX_CLIENT_1_0.INITO`, the end of the chain

The switching events themselves still travel via the adapters.

Data Inputs

- **QI** (BOOL, default TRUE): Quality Input - TRUE enables the service; goes to Softkey_IXA, Aux_IXA, SUBSCRIBE_WEB and AX_CLIENT_1_0
- **u16ObjId** (UINT, default ID_NULL): Object ID of the SoftKey (VT)
- **u16ObjIdA** (UINT, default ID_NULL): Object ID of the AuxFunction2 (VT)
- **ID_WRITE_REMOTE** (WSTRING): Remote write address to the target module (command, ACTION=WRITE, CLIENT)
- **ID_WEB_READ** (WSTRING): Local subscribe address for a web client (e.g. vt-ui-mirror) on the same module, ORed with SoftKey and AUX

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if the service was executed successfully (AND of the QO of Softkey_IXA, Aux_IXA, SUBSCRIBE_WEB and AX_CLIENT_1_0)

Adapters

- **AUTO_IN** (socket, adapter::types::unidirectional::AX): Autonomous command (e.g. axle steering centre automation), ORed with SoftKey/Aux/web
- **WEB_STATE** (plug, adapter::types::unidirectional::AX): Web state (= SUBSCRIBE_WEB.OUT) passed on to the outside, so that other composite instances can use the same value via a data connection instead of subscribing locally again (only one local OPC-UA subscriber per node is allowed)

Program Flow and Connections

- u16ObjId ⟶ Softkey_IXA.u16ObjId; Softkey_IXA.IN ⟶ OR_MERGE.IN1
- u16ObjIdA ⟶ Aux_IXA.u16ObjId; Aux_IXA.IN ⟶ OR_MERGE.IN2
- ID_WEB_READ ⟶ SUBSCRIBE_WEB.ID; SUBSCRIBE_WEB.OUT ⟶ SPLIT_WEB.IN
- SPLIT_WEB.OUT1 ⟶ OR_MERGE.IN3; SPLIT_WEB.OUT2 ⟶ WEB_STATE
- AUTO_IN ⟶ OR_MERGE.IN4
- OR_MERGE.OUT ⟶ AX_CLIENT_1_0.IN; ID_WRITE_REMOTE ⟶ AX_CLIENT_1_0.ID
- Initialization: INIT ⟶ Softkey_IXA.INIT, Softkey_IXA.INITO ⟶ Aux_IXA.INIT, Aux_IXA.INITO ⟶ SUBSCRIBE_WEB.INIT, SUBSCRIBE_WEB.INITO ⟶ AX_CLIENT_1_0.INIT, AX_CLIENT_1_0.INITO ⟶ INITO and ⟶ AND_QO.REQ
- Quality: QI ⟶ Softkey_IXA.QI, Aux_IXA.QI, SUBSCRIBE_WEB.QI and AX_CLIENT_1_0.QI; their QO ⟶ AND_QO.IN1..IN4; AND_QO.OUT ⟶ QO

Technical Details

- Version 1.2 (2026-09-16): SPLIT_WEB (AX_SPLIT_2) inserted, since adapter plugs are strictly 1:1 (no fan-out as with data/event connections); two destinations on the same plug caused a FORTE boot error (INVALID_STATE on the second CREATE Connection)
- Version 1.1 (2026-09-16): plug WEB_STATE added; fixes the double self-subscribe on ID_WEB_READ (FORTE: “Cannot add another one”)
- Version 1.0 (2026-09-11): copy of Softkey_Aux_IXA_TO_Remote_WRITE with AX_OR_4 instead of AX_OR_3; use only for channels that need an automation, all other users remain on the original type
- No status feedback and no background colour in this block

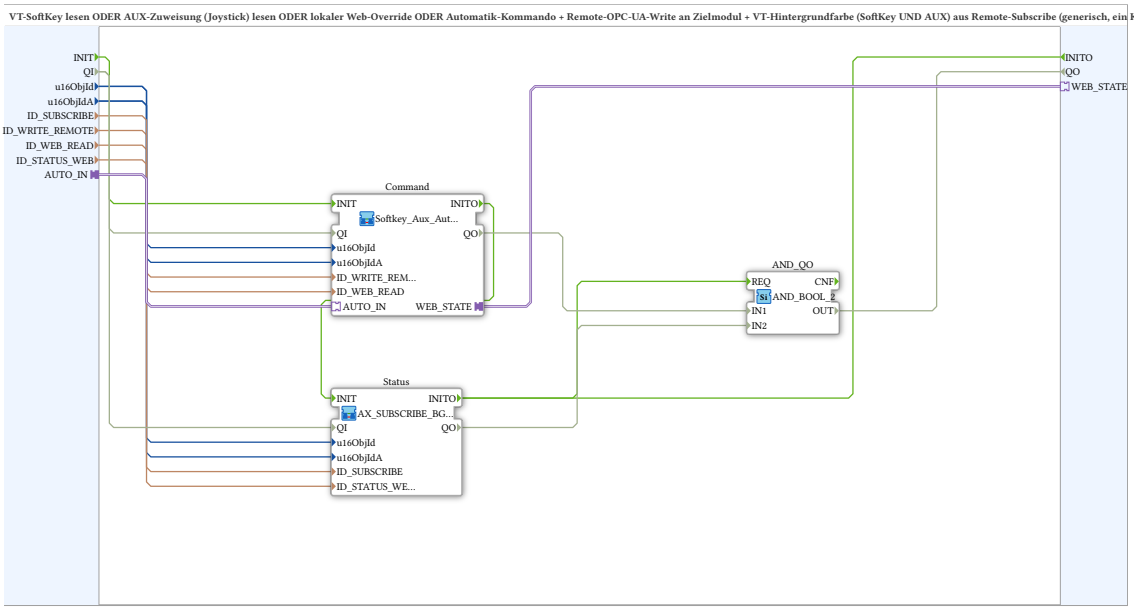
Application Scenarios

- Remote command that can be triggered simultaneously via SoftKey, joystick, web client and autonomous automation (e.g. axle steering centre automation)
- Source of WEB_STATE for Softkey_Aux_Web_IE (override detection)

Summary

Softkey_Aux_Auto_IXA_TO_Remote_WRITE ORs SoftKey, AUX, web override and automation and writes the result to the target module via exactly one publisher.

Softkey_Aux_Auto_IXA_TO_Remote_WRITE_BG_OPC



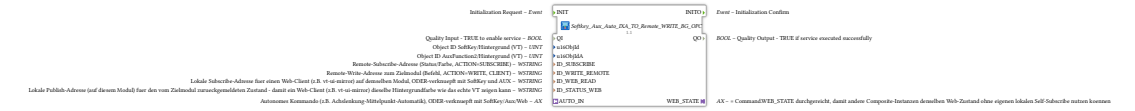
• – ○ • – ○ • – ○ •

Introduction

`Softkey_Aux_Auto_IXA_TO_Remote_WRITE_BG_OPC` is a thin wrapper (one channel) that places two Sub-Apps side by side:

- **Command** (`Softkey_Aux_Auto_IXA_TO_Remote_WRITE`): read VT softkey OR read AUX assignment (joystick) OR local web override OR automatic command → remote OPC-UA write to the target module.
- **Status** (`AX_SUBSCRIBE_BG3_WEB_OPC`): VT background color (softkey AND AUX) from a remote subscribe plus local web republish (unchanged).

The block is a copy of `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` that uses `Softkey_Aux_Auto_IXA_TO_Remote_WRITE` instead of `Softkey_Aux_IXA_TO_Remote_WRITE` as its command half. It is intended only for channels with an automatic requirement (currently: axle steering left/right, slope compensation left/right); for all others `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` remains responsible.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

The user events still arrive via the adapter socket.

Event Outputs

- **INITO** (`EInit`): initialization confirmation (from `Status.INITO`)

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **u16ObjId** (`UINT`): Object ID softkey/background (VT)
- **u16ObjIdA** (`UINT`): Object ID AuxFunction2/background (VT)
- **ID_SUBSCRIBE** (`WSTRING`): Remote subscribe address (status/color, `ACTION=SUBSCRIBE`)
- **ID_WRITE_REMOTE** (`WSTRING`): Remote write address to the target module (command, `ACTION=WRITE`, CLIENT)

- **ID_WEB_READ** (WSTRING): Local subscribe address for a web client (e.g. vt-ui-mirror) on the same module, ORed with softkey and AUX
- **ID_STATUS_WEB** (WSTRING): Local publish address (on this module) for the state reported back by the target module, so that a web client can show the same background color as the real VT

Data Outputs

- **QO** (BOOL): Quality Output - AND of Command.QO and Status.QO

Adapters

- **AUTO_IN** (socket, adapter::types::unidirectional::AX): Autonomous command (e.g. axle steering center automatic), ORed with softkey/aux/web
- **WEB_STATE** (plug, adapter::types::unidirectional::AX): Command.WEB_STATE passed through, so that other composite instances can use the same web state without their own local self-subscribe

Function Blocks (FBs) Used

- **Command**: SubApp MyLib::sys::Softkey_Aux_Auto_IXA_TO_Remote_WRITE
- **Status**: SubApp MyLib::sys::AX_SUBSCRIBE_BG3_WEB_OPC
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): combines Command.QO and Status.QO into QO

Program Flow and Connections

- u16ObjId, u16ObjIdA ⟶ Command and Status
- ID_WRITE_REMOTE, ID_WEB_READ ⟶ Command
- ID_SUBSCRIBE, ID_STATUS_WEB ⟶ Status
- AUTO_IN ⟶ Command.AUTO_IN
- Command.WEB_STATE ⟶ WEB_STATE
- INIT ⟶ Command.INIT; Command.INITO ⟶ Status.INIT; Status.INITO ⟶ INITO (and ⟶ AND_QO.REQ)
- QI ⟶ Command.QI and Status.QI
- Command.QO ⟶ AND_QO.IN1; Status.QO ⟶ AND_QO.IN2; AND_QO.OUT ⟶ QO

Command and Status are not directly connected to each other; the feedback runs via the remote subscribe.

Technical Details

- WEB_STATE (version 1.1 of Softkey_Aux_Auto_IXA_TO_Remote_WRITE) is a passed-through fan-out plug.
- Use only for channels with an automatic requirement.

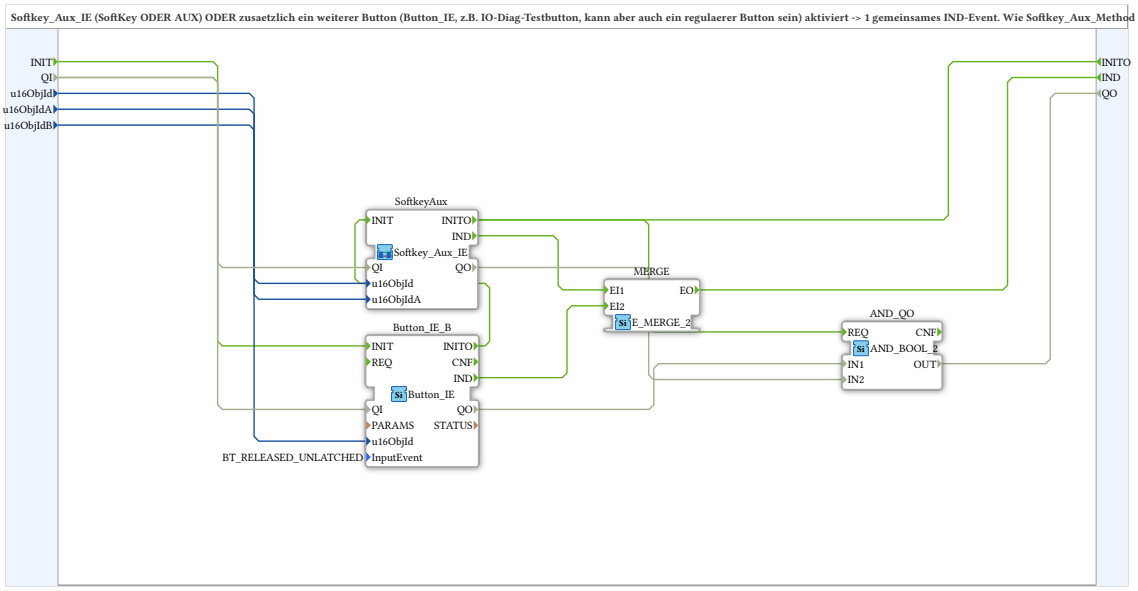
Application Scenarios

- VT key/AUX/web/automatic that writes a command via OPC-UA to a target module, with background color from the reported-back state (e.g. axle steering, slope compensation)

Summary

Softkey_Aux_Auto_IXA_TO_Remote_WRITE_BG_OPC combines command (incl. automatic) and status feedback of one channel in one SubApp.

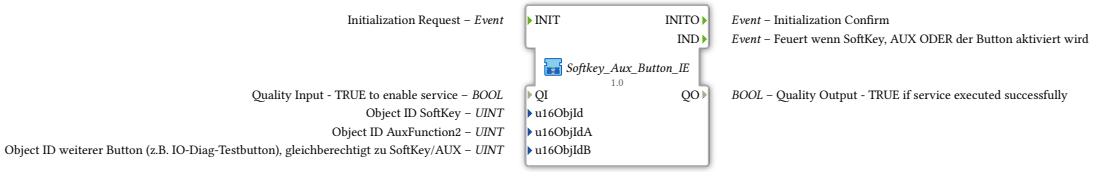
Softkey_Aux_Button_IE



• – ○ • – ○ • – ○ •

Introduction

`Softkey_Aux_Button_IE` combines `Softkey_Aux_IE` (SoftKey OR AUX) and an additional button (`Button_IE`, e.g. the IO diagnostics test button, but it can also be a regular button) into **one** common `IND` event. It corresponds to `Softkey_Aux_Method_Button_IE` but without the local OPC-UA method and is intended for cases that only need SoftKey, AUX and button with equal rank.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirm
- **IND**: fires when the SoftKey, AUX **or** the button is activated

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, TRUE enables the service
- **u16ObjId** (UINT): object ID of the SoftKey
- **u16ObjIdA** (UINT): object ID of the AuxFunction2
- **u16ObjIdB** (UINT): object ID of a further button (e.g. IO diagnostics test button), equal in rank to SoftKey/AUX

Data Outputs

- **QO** (`BOOL`): quality output, AND combination (`AND_QO`) of the `QO` outputs of `Button_IE_B` and `SoftkeyAux`

Adapters

No adapters are used.

Functionality

The network contains four blocks:

- SoftkeyAux (MyLib::sys::Softkey_Aux_IE): u16ObjId ⟶ SoftkeyAux.u16ObjId, u16ObjIdA ⟶ SoftkeyAux.u16ObjIdA; its IND goes to MERGE.EI1.
- Button_IE_B (isobus::UT::io::Button::Button_IE, InputEvent = BT_RELEASED_UNLATCHED): u16ObjIdB ⟶ Button_IE_B.u16ObjId; its IND goes to MERGE.EI2.
- MERGE (iec61499::events::E_MERGE_2): MERGE.EO ⟶ IND.
- AND_QO (iec61131::booleanOperators::AND_BOOL_2): combines Button_IE_B.QO (IN1) and SoftkeyAux.QO (IN2) into QO.

Initialization: INIT ⟶ Button_IE_B.INIT; Button_IE_B.INITO ⟶ SoftkeyAux.INIT; SoftkeyAux.INITO ⟶ INITO and ⟶ AND_QO.REQ. Quality: QI ⟶ Button_IE_B.QI and SoftkeyAux.QI; AND_QO.OUT ⟶ QO.

Technical Details

- The additional button triggers on release (BT_RELEASED_UNLATCHED)
- Extracted from Baender_Individual_Toggle.SUB, where the same combination (Softkey_Aux_IE + Button_IE + E_MERGE_2) was wired three times for hedgehog belt, elevator and cross belt
- Stateless

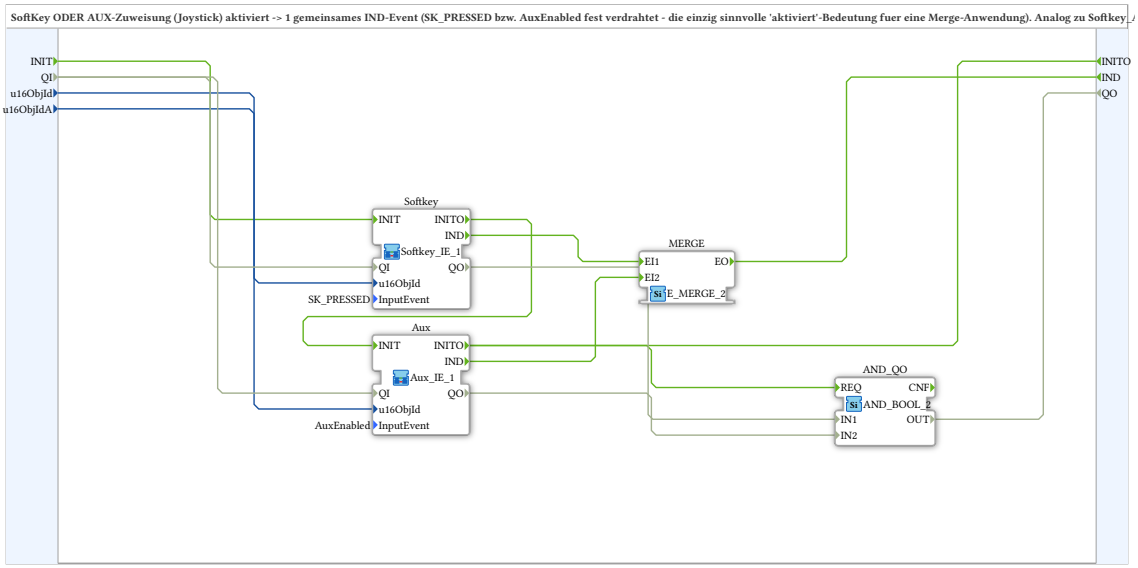
Application Scenarios

- An output that is to be triggered with equal rank by SoftKey, joystick assignment and an additional button (e.g. IO diagnostics test button)

Conclusion

Softkey_Aux_Button_IE merges SoftKey, AUX and a further button into a single IND event.

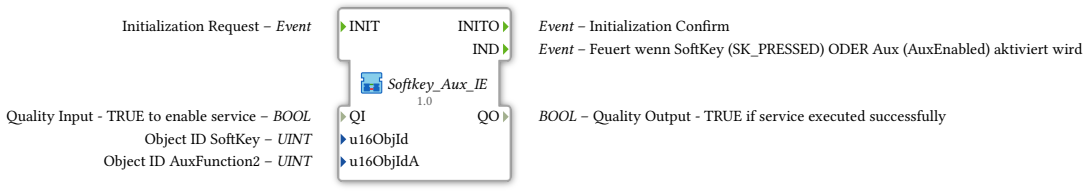
Softkey_Aux_IE



• – ○ • – ○ • – ○ •

Introduction

Softkey_Aux_IE raises a common IND event when a **SoftKey** OR an **AUX assignment (joystick)** is activated. SK_PRESSED and AuxEnabled are hard-wired, since this is the only sensible "activated" meaning for a merge application. The block is analogous to Softkey_Aux_IXA_TO_Remote_WRITE (continuous-state OR), but for the discrete event channel. It therefore does not collide with an already existing Softkey_IXA/Aux_IXA continuous-state subscription on the same object (two different subscription types).



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization Request

Event Outputs

- **INITO** (`EInit`): Initialization Confirm
- **IND**: fires when the SoftKey (`SK_PRESSED`) OR Aux (`AuxEnabled`) is activated

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE to enable the service
- **u16ObjId** (`UINT`): object ID of the SoftKey
- **u16ObjIdA** (`UINT`): object ID of the AuxFunction2

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service executed successfully (AND of `Softkey.QO` and `Aux.QO`)

Adapters

None.

Function Blocks (FBs) Used

- **`Softkey`** (SubApp `MyLib::sys::Softkey_IE_1`, Softkey_IE_1): parameter `InputEvent = SK_PRESSED`
- **`Aux`** (SubApp `MyLib::sys::Aux_IE_1`, Aux_IE_1): parameter `InputEvent = AuxEnabled`
- **`MERGE`** (`iec61499::events::E_MERGE_2`): merges both events
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): ANDs `Softkey.QO` and `Aux.QO` into `QO`

Functionality

- `u16ObjId` ⟶ `Softkey.u16ObjId`; `u16ObjIdA` ⟶ `Aux.u16ObjId`
- `Softkey.IND` ⟶ `MERGE.EI1`; `Aux.IND` ⟶ `MERGE.EI2`
- `MERGE.EO` ⟶ `IND`
- Initialization chain: `INIT` ⟶ `Softkey.INIT`, `Softkey.INITO` ⟶ `Aux.INIT`, `Aux.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`
- `QI` ⟶ `Softkey.QI` and `Aux.QI`
- `Softkey.QO` ⟶ `AND_QO.IN1`, `Aux.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`

Technical Details

- Imports `isobus::UT::io::Softkey::SoftKeyActivationCode::SK_PRESSED` and `isobus::UT::io::Auxiliary::IN::AuxiliaryState::AuxEnabled`
- Works via the discrete event channel, not via the continuous state

Application Scenarios

- One event from SoftKey or joystick assignment, e.g. for toggle or trigger functions

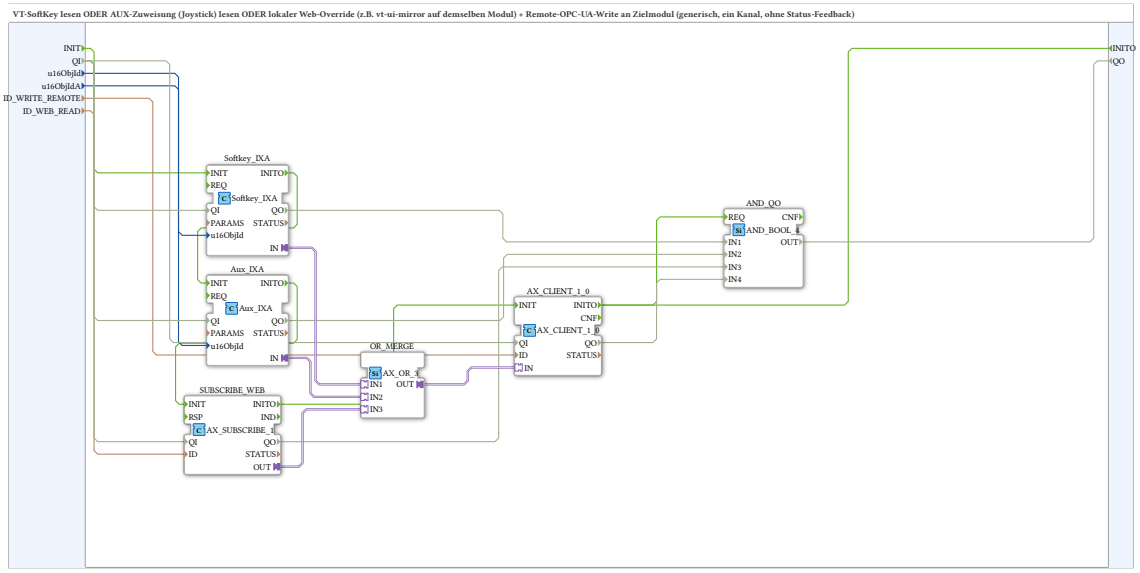
Summary

`Softkey_Aux_IE` combines SoftKey and AUX activation into one common `IND` event.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

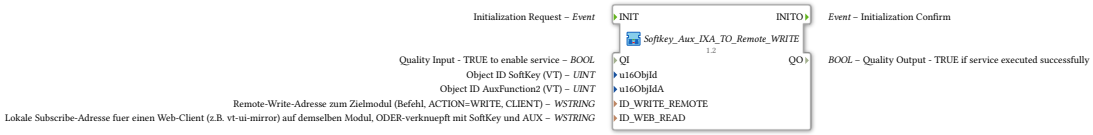
Softkey_Aux_IXA_TO_Remote_WRITE



• – ○ • – ○ • – ○ •

Introduction

`Softkey_Aux_IXA_TO_Remote_WRITE` is the command half of `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`, extracted as an independent component: VT SoftKey, AUX assignment (joystick), and a local web override (e.g., vt-ui-mirror on the same module) are combined using an OR operator and written to a target module via remote write—without status feedback.



Function Blocks (FBs) Used

Sub-Blocks: Softkey_Aux_IXA_TO_Remote_WRITE

- **Type**: SubAppType
- **Internal FBs Used**:
- **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA` — VT SoftKey Adapter (`u16ObjId`).
- **Aux_IXA**: `isobus::UT::io::Auxiliary::IN::Aux_IXA` — AUX Assignment/Joystick Adapter (`u16ObjIdA`).
- **SUBSCRIBE_WEB**: `adapter::net::AX_SUBSCRIBE_1` — Local web subscribe (`ID_WEB_READ`), e.g., from a vt-ui-mirror on the same module.
- **OR_MERGE**: `adapter::booleanOperators::AX_OR_3` — Combines SoftKey, Aux, and Web Override using OR.
- **AX_CLIENT_1_0**: `adapter::net::AX_CLIENT_1_0` — Writes the combined signal to the target module using remote write (`ID_WRITE_REMOTE`).
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_4` — Combines the `QO` outputs of `Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB` and `AX_CLIENT_1_0` into the SubApp output `QO`.
- **Interface**: event input `INIT`, event output `INITO`, data input `QI` (default `TRUE`, passed on to the four FBs), data output `QO`.
- **How it Works**: Three independent command sources (SoftKey, Aux, local web override) are combined using `AX_OR_3` and transmitted to the target module as a single remote write command.

Program Flow and Connections

1. `u16ObjId` ⟶ `Softkey_IXA.u16ObjId`; `u16ObjIdA` ⟶ `Aux_IXA.u16ObjId`; `ID_WEB_READ` ⟶ `SUBSCRIBE_WEB.ID`; `ID_WRITE_REMOTE` ⟶ `AX_CLIENT_1_0.ID` (Data connections, hidden).
2. `Softkey_IXA.IN` ⟶ `OR_MERGE.IN1`; `Aux_IXA.IN` ⟶ `OR_MERGE.IN2`; `SUBSCRIBE_WEB.OUT` ⟶ `OR_MERGE.IN3`.
3. `OR_MERGE.OUT` ⟶ `AX_CLIENT_1_0.IN`.
4. Initialization: `INIT` ⟶ `Softkey_IXA.INIT`; `Softkey_IXA.INITO` ⟶ `Aux_IXA.INIT`; `Aux_IXA.INITO` ⟶ `SUBSCRIBE_WEB.INIT`; `SUBSCRIBE_WEB.INITO` ⟶ `AX_CLIENT_1_0.INIT`; `AX_CLIENT_1_0.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
5. Quality: `QI` ⟶ `Softkey_IXA.QI`, `Aux_IXA.QI`, `SUBSCRIBE_WEB.QI` and `AX_CLIENT_1_0.QI`; their `QO` outputs ⟶ `AND_QO.IN1` to `IN4`; `AND_QO.OUT` ⟶ `QO`.

Technical Features

- **Three Equal Command Sources**: SoftKey, AUX Assignment, and Local Web Override function equally via the OR operator—any of the three sources can trigger the command.
- **No Status Feedback**: Unlike `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`, this block does not contain any feedback/background color; For this purpose, `AX_SUBSCRIBE_BG3_WEB_OPC` should be placed alongside it with the same values as `u16ObjId`/`u16ObjIdA`. Important for this manual pairing: `ID_WEB_READ` (here) and `ID_STATUS_WEB` (in `AX_SUBSCRIBE_BG3_WEB_OPC`) must be different local OPC UA nodes, otherwise the same feedback-loop risk arises as described for `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`. `ID_WRITE_REMOTE` (here) and `ID_SUBSCRIBE` (there), by contrast, both target the remote module and may reference the same remote node.

Application Scenarios

- Remote control of a target module via soft key, AUX joystick assignment, or a local web client, without requiring a status display on the same module.

Comparison with Similar Modules

For the complete, pre-wired combination of both sides (command + status), `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` should still be used, which contains this module as a command sub-module.

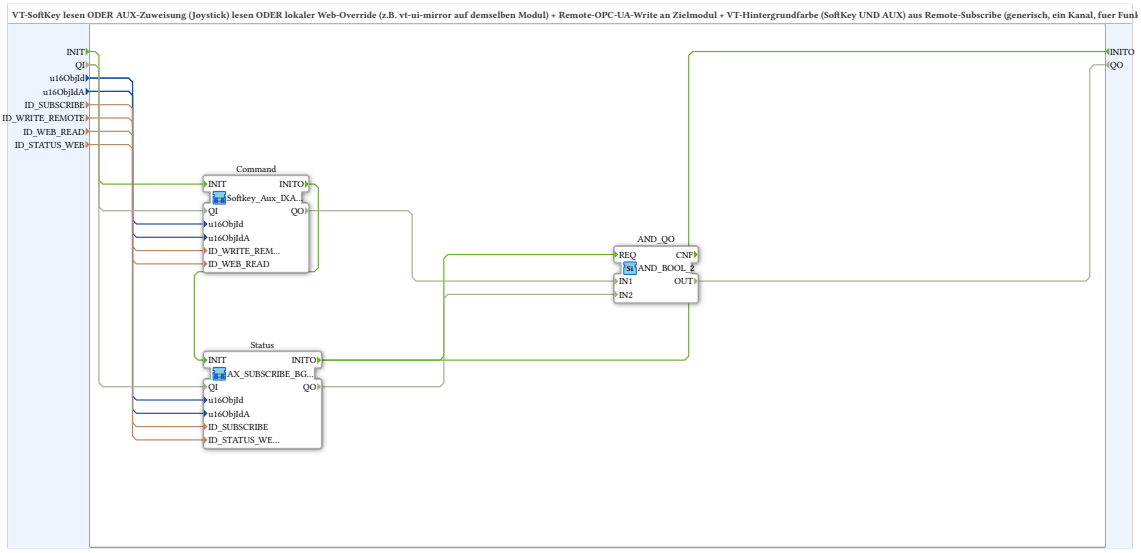
Summary

`Softkey_Aux_IXA_TO_Remote_WRITE` encapsulates the pure command side of a remote channel: three OR-linked sources (soft key, aux, local web override), written to a target module via remote write.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC

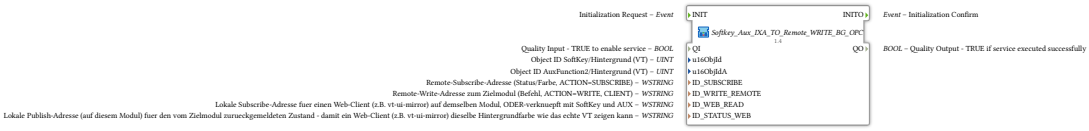


• — ○ • — ○ • — ○ •

Introduction

`Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` reads a VT softkey OR an AUX assignment (joystick) OR a local web override and writes the result to a target module via remote OPC UA write. Additionally, the VT background color (softkey AND AUX) is set using a remote subscribe from the target module. This is intended for functions without a local physical output, where the control interface is located on one module (e.g., STG1), but the actuator is on a different module without its own VT connection.

`Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC` reads a VT softkey OR an AUX assignment (joystick) OR a local web override and writes the result to a target module via remote OPC UA write. Since version 1.2, the function block is a thin-walled wrapper that places two independently reusable sub-blocks side by side: `Softkey_Aux_IXA_TO_Remote_WRITE` (command) and `AX_SUBSCRIBE_BG3_WEB_OPC` (status). The external interface initially remained unchanged; the wrapper has since gained the events `INIT`/`INITO` as well as `QI` (BOOL, initial value `TRUE`) and `QO` (BOOL).



Function Blocks (FBs) Used

Sub-blocks: Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC

- **Type**: SubAppType
- **Internal FBs Used**:
- **Command** (SubApp, `MyLib::sys::Softkey_Aux_IXA_TO_Remote_WRITE`): SoftKey OR Aux OR local web override ⟶ Remote write to the target module.
- **Status** (SubApp, `MyLib::sys::AX_SUBSCRIBE_BG3_WEB_OPC`): Remote Subscribe ⟶ Background color on SoftKey AND Aux + local web republish.
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_2`): combines `Command.QO` and `Status.QO` into the output `QO`.
- **Functionality**: Both sub-components run independently as far as data is concerned (only the init chain and the quality combination connect them); they only share the same `u16ObjId`/`u16ObjIdA` values so that the command and status display operate on the same VT object.

Program Flow and Connections

1. `u16ObjId` ⟶ `Command.u16ObjId` and `Status.u16ObjId`; `u16ObjIdA` ⟶ `Command.u16ObjIdA` and `Status.u16ObjIdA` (data connections, hidden).

2. ID_WRITE_REMOTE ⟶ Command.ID_WRITE_REMOTE; ID_WEB_READ ⟶ Command.ID_WEB_READ.
3. ID_SUBSCRIBE ⟶ Status.ID_SUBSCRIBE; ID_STATUS_WEB ⟶ Status.ID_STATUS_WEB.
4. Apart from the init chain (item 5), there is no direct SubApp connection between Command and Status —both sub-apps are completely independent and communicate only indirectly via the target module (commands write to it, status updates are subscribed to from it). Locally, their paths can still become connected if ID_WEB_READ and ID_STATUS_WEB are accidentally configured to alias the same OPC UA node (see the feedback-loop note below).
5. Initialization: INIT ⟶ Command.INIT; Command.INITO ⟶ Status.INIT; Status.INITO ⟶ INITO and AND_QO.REQ.
6. Quality: QI ⟶ Command.QI and Status.QI; Command.QO ⟶ AND_QO.IN1; Status.QO ⟶ AND_QO.IN2; AND_QO.OUT ⟶ QO.

Technical Features

- **Pure wrapper since version 1.2**: The splitting into Command/Status does not change the behavior, but allows the independent reuse of both halves (e.g., AX_SUBSCRIBE_BG3_WEB_OPC solely for status displays).
- **No local physical output**: The actuator is located on a separate module; this block only maps operation and status feedback to the operator module.
- **Separate local nodes for web override and web status**: ID_WRITE_REMOTE (write) and ID_SUBSCRIBE (read) both point at the remote target module and may legitimately reference the same remote node there — write and subscribe are different OPC UA operations, so this poses no feedback risk on this module. The actual feedback-loop risk lies with the two *local* nodes on this module: ID_WEB_READ (read by Command via AX_SUBSCRIBE_1 and merged into the command OR) and ID_STATUS_WEB (written by Status via AX_PUBLISH_1 with the state echoed back from the target module) must be different local OPC UA nodes — otherwise Command would read the state republished by Status as if it were a manual web override and write it straight back to the target module as a new command.

Application Scenarios

- Control stations without their own physical output, where both the SoftKey and AUX joystick assignment are intended to control a remote actuator, and the control screen must additionally display its status using color.

Comparison with Similar Modules

If only a SoftKey without AUX is required, use Softkey_IXA_TO_Remote_WRITE_BG_OPC. For command or status alone (without the other half), the sub-modules Softkey_Aux_IXA_TO_Remote_WRITE and AX_SUBSCRIBE_BG3_WEB_OPC, respectively, can be used directly.

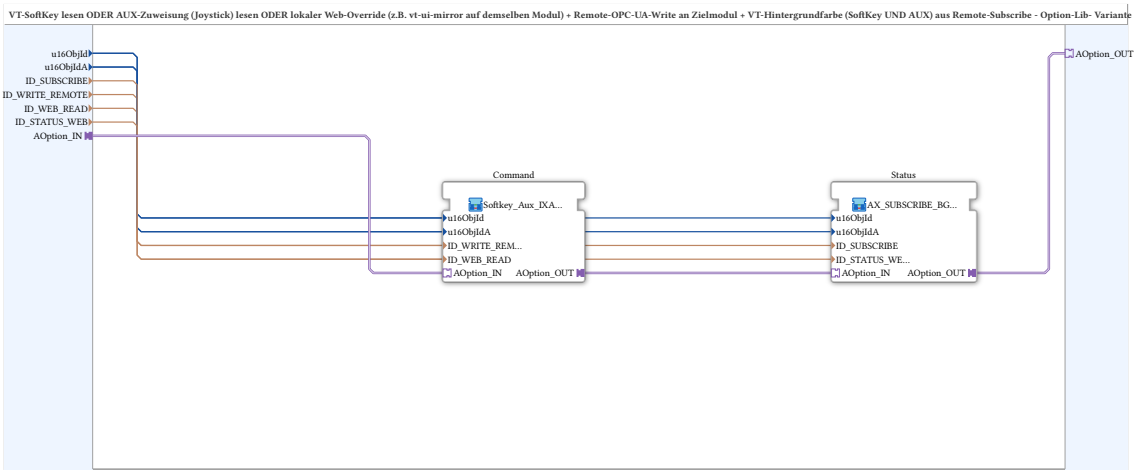
Summary

Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC combines the command and status pages of a remote channel with softkey and auxiliary controls behind a stable, well-known interface—since version 1.2, it's a pure wrapper around two independent components.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC



• — ○ • — ○ • — ○ •

Introduction

`Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC` is the option-lib variant of `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`: read a VT SoftKey OR an AUX assignment (joystick) OR a local web override, remote OPC-UA write to a target module, and VT background colour (SoftKey and AUX) from a remote subscribe. If the option is not selected, both the command side and the status side are inactive (real `INIT_REQ`/`INIT_CNF`, no VT object binding). The block is a thin wrapper around two sub-blocks that are chained sequentially via the `AOption` adapter.



Interface Structure

Event Inputs

None (initialization runs via the adapter `AOption_IN`).

Event Outputs

None.

Data Inputs

Variable	Type	Default	Comment
`u16ObjId`	UINT		Object ID SoftKey/background (VT).
`u16ObjIdA`	UINT		Object ID AuxFunction2/background (VT).
`ID_SUBSCRIBE`	WSTRING		Remote subscribe address (status/colour, `ACTION=SUBSCRIBE`).
`ID_WRITE_REMOTE`	WSTRING		Remote write address to the target module (command, `ACTION=WRITE`, CLIENT).
`ID_WEB_READ`	WSTRING		Local subscribe address for a web client on the same module, ORed with SoftKey and AUX.
`ID_STATUS_WEB`	WSTRING		Local publish address for the state reported back by the target module.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
AOption_IN	Option::AOption	Socket	Drives INIT/QI of Command AND Status sequentially; connect e.g. to AOption_AUI_DEMUX_N.OUTx or the AOption_OUT of an upstream instance.
AOption_OUT	Option::AOption	Plug	Passes the activation on to a downstream instance (daisy chain, e.g. a second instance sharing the same option); leave unused if nothing follows.

Function Blocks (FBs) Used

Sub-blocks: Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC

- **Type**: SubAppType
- **Internal FBs used**:
  - **Command** (SubApp, MyLib::sys::Softkey_Aux_IXA_TO_Remote_WRITE_Option): SoftKey OR Aux OR web override -> remote write; see Softkey_Aux_IXA_TO_Remote_WRITE_Option.
  - **Status** (SubApp, MyLib::sys::AX_SUBSCRIBE_BG3_WEB_Option_OPC): remote subscribe -> background colour on SoftKey and Aux plus local web republish; see AX_SUBSCRIBE_BG3_WEB_Option_OPC.

Program Flow and Connections

1. Adapter chain: AOption_IN -> Command.AOption_IN; Command.AOption_OUT -> Status.AOption_IN; Status.AOption_OUT -> AOption_OUT. The option thus activates the command side first, then the status side, then a downstream instance.
2. u16ObjId -> Command.u16ObjId and Status.u16ObjId; u16ObjIdA -> Command.u16ObjIdA and Status.u16ObjIdA.
3. ID_WRITE_REMOTE -> Command.ID_WRITE_REMOTE; ID_WEB_READ -> Command.ID_WEB_READ; ID_SUBSCRIBE -> Status.ID_SUBSCRIBE; ID_STATUS_WEB -> Status.ID_STATUS_WEB.
4. The block has no event or QI/QO connections of its own; activation and acknowledgment run exclusively via the adapters AOption_IN/AOption_OUT.

Technical Features

- **Command and Status are both option-gated:** All blocks with a real QI can be switched off via the AOption socket.
- **Sequential chain instead of socket split:** Command.AOption_OUT -> Status.AOption_IN (strictly 1:1), analogous to the daisy-chain style of OptionalGroup_001a. The block's own AOption_OUT passes the chain on to further instances sharing the same option.
- As in the original: ID_WEB_READ (from Command) and ID_STATUS_WEB (from Status) must point to different local OPC-UA nodes, otherwise feedback arises (see Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC).

Application Scenarios

- Optional operator station with SoftKey and Aux assignment for a remote actuator including coloured status display, active only when the option is selected.
- Several instances sharing the same option, chained via AOption_OUT.

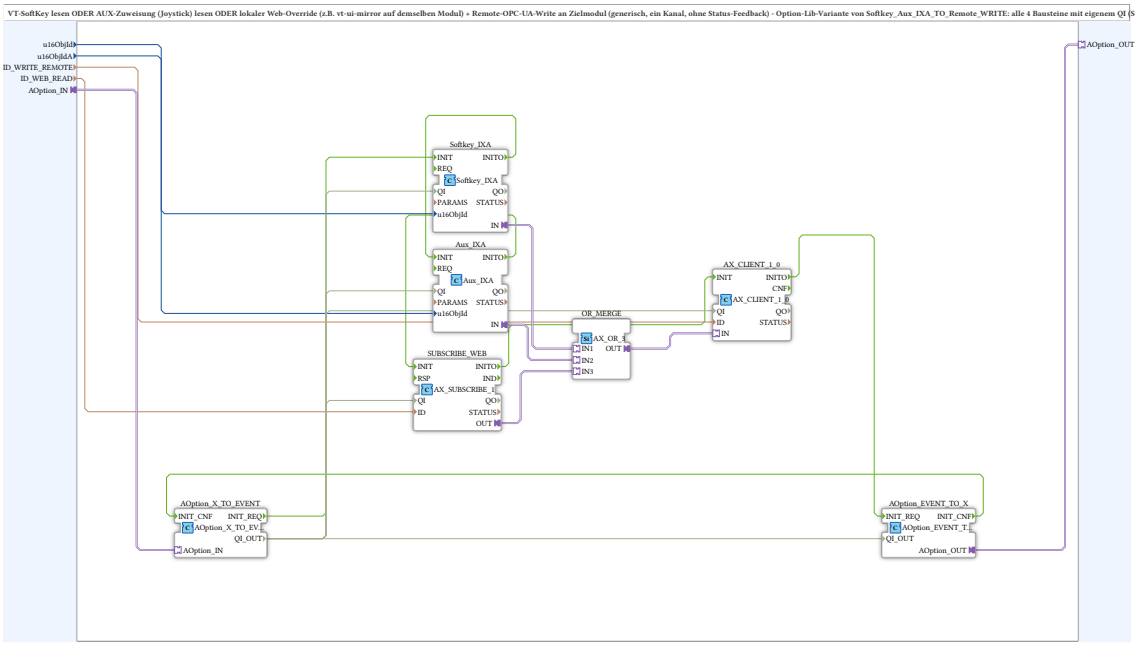
Summary

Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC bundles the command and status sides of a remote channel behind a shared AOption socket and passes the activation on via AOption_OUT.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Softkey_Aux_IXA_TO_Remote_WRITE_Option



• — ○ • — ○ • — ○ •

Introduction

`Softkey_Aux_IXA_TO_Remote_WRITE_Option` reads a VT SoftKey OR an AUX assignment (joystick) OR a local web override (e.g. `vt-ui-mirror` on the same module) and writes the result via remote OPC-UA write to a target module (generic, one channel, without status feedback). It is the option-lib variant of `Softkey_Aux_IXA_TO_Remote_WRITE`: all four blocks with their own `QI` (`Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB`, `AX_CLIENT_1_0`) are initialized and gated sequentially by a single `Option::AOption` socket. The plug `AOption_OUT` passes the activation on to a downstream group (daisy chain).



Interface Structure

Event Inputs / Outputs

None (initialization runs via the adapter `AOption_IN`).

Data Inputs

Variable	Type	Default	Comment
`u16ObjId`	UINT	`ID_NULL`	Object ID SoftKey (VT).
`u16ObjIdA`	UINT	`ID_NULL`	Object ID AuxFunction2 (VT).
`ID_WRITE_REMOTE`	WSTRING		Remote write address to the target module (command, `ACTION=WRITE`, CLIENT).
`ID_WEB_READ`	WSTRING		Local subscribe address for a web client (e.g. `vt-ui-mirror`) on the same module, ORed with SoftKey and AUX.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
AOption_IN	Option::AOption	Socket	Drives INIT/QI of Softkey_IXA, Aux_IXA, SUBSCRIBE_WEB and AX_CLIENT_1_0; connect e.g. to AOption_AUI_DEMUX_N.OUTx.
AOption_OUT	Option::AOption	Plug	Passes the activation on to a downstream group (daisy chain); leave unused if nothing follows.

Function Blocks (FBs) Used

Sub-blocks: Softkey_Aux_IXA_TO_Remote_WRITE_Option

- **Type**: SubAppType
- **Internal FBs used**:
  - **AOption_X_TO_EVENT** (Option::conversion::AOption_X_TO_EVENT): converts the socket AOption_IN into INIT_REQ/QI_OUT, see AOption_X_TO_EVENT.
  - **Softkey_IXA** (isobus::UT::io::Softkey::Softkey_IXA): reads the SoftKey (u16ObjId).
  - **Aux_IXA** (isobus::UT::io::Auxiliary::IN::Aux_IXA): reads the AUX assignment (u16ObjIdA).
  - **SUBSCRIBE_WEB** (adapter::net::AX_SUBSCRIBE_1): reads the local web override (ID_WEB_READ).
  - **OR_MERGE** (adapter::booleanOperators::AX_OR_3): OR of the three sources.
  - **AX_CLIENT_1_0** (adapter::net::AX_CLIENT_1_0): remote write to ID_WRITE_REMOTE.
  - **AOption_EVENT_TO_X** (Option::conversion::AOption_EVENT_TO_X): passes the activation on to AOption_OUT.

Program Flow and Connections

1. Sequential initialization: AOption_X_TO_EVENT.INIT_REQ -> Softkey_IXA.INIT; Softkey_IXA.INITO -> Aux_IXA.INIT; Aux_IXA.INITO -> SUBSCRIBE_WEB.INIT; SUBSCRIBE_WEB.INITO -> AX_CLIENT_1_0.INIT; AX_CLIENT_1_0.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF.
2. AOption_X_TO_EVENT.QI_OUT -> QI of Softkey_IXA, Aux_IXA, SUBSCRIBE_WEB, AX_CLIENT_1_0 and AOption_EVENT_TO_X.QI_OUT.
3. Data: Softkey_IXA.IN -> OR_MERGE.IN1; Aux_IXA.IN -> OR_MERGE.IN2; SUBSCRIBE_WEB.OUT -> OR_MERGE.IN3; OR_MERGE.OUT -> AX_CLIENT_1_0.IN.
4. Parameters: u16ObjId -> Softkey_IXA.u16ObjId; u16ObjIdA -> Aux_IXA.u16ObjId; ID_WEB_READ -> SUBSCRIBE_WEB.ID; ID_WRITE_REMOTE -> AX_CLIENT_1_0.ID.
5. Adapters: AOption_IN -> AOption_X_TO_EVENT.AOption_IN; AOption_EVENT_TO_X.AOption_OUT -> AOption_OUT.

Technical Features

- **Real INIT_REQ/INIT_CNF instead of just a QI bool:** If the option is not selected, all four blocks stay inactive.
- **Sequential instead of parallel:** Initialization runs as a daisy chain (Softkey_IXA -> Aux_IXA -> SUBSCRIBE_WEB -> AX_CLIENT_1_0); AOption_OUT allows further instances sharing the same option to be attached (e.g. a second SoftKey instance) or, as in Softkey_Aux_IXA_TO_Remote_WRITE_BG_Option_OPC, the status half.
- The counterpart without option is Softkey_Aux_IXA_TO_Remote_WRITE.

Application Scenarios

- Optional operating function that reads SoftKey, Aux and web override only when the option is selected and writes to a target module via remote write.
- Several SoftKeys sharing the same option, chained via AOption_OUT.

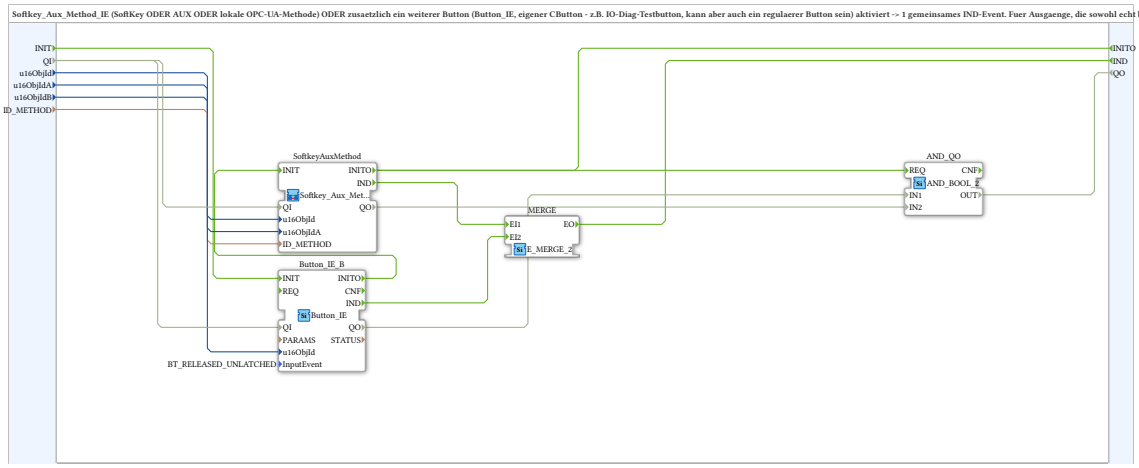
Summary

Softkey_Aux_IXA_TO_Remote_WRITE_Option is the option-gated variant of the remote-write command with SoftKey, Aux and web override; its four QI blocks are activated via an AOption socket and it passes the activation on via AOption_OUT.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Softkey_Aux_Method_Button_IE

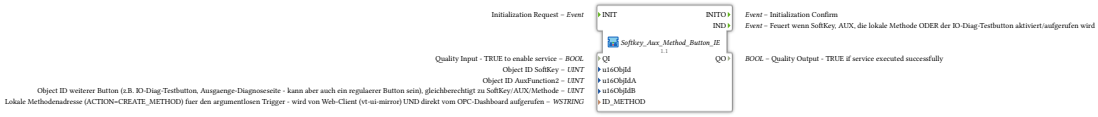


• – ○ • – ○ • – ○ •

Introduction

`Softkey_Aux_Method_Button_IE` extends `Softkey_Aux_Method_IE` (SoftKey OR AUX OR local OPC-UA method) by **one further button** (`Button_IE`, its own `CButton` - e.g. an IO-diag test button, but it can also be a regular button). If any of these sources is activated, a common `IND` event is produced.

The block serves outputs that are both operated for real (SoftKey/AUX/web/dashboard) and triggered equally via an additional button. The button background is fed separately by the caller from the resulting state; `u16ObjIdB` therefore remains available to the caller.



Function Blocks (FBs) Used

Sub-blocks: Softkey_Aux_Method_Button_IE

- **Type**: SubAppType
- **Internal blocks used**:
  - **SoftkeyAuxMethod**: SubApp `MyLib::sys::Softkey_Aux_Method_IE` - SoftKey, AUX or local method
  - **Button_IE_B**: `isobus::UT::io::Button::Button_IE` (`InputEvent=BT_RELEASED_UNLATCHED`) - further button
  - **MERGE**: `iec61499::events::E_MERGE_2` - merges both `IND` events
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` - combines the `QO` of `Button_IE_B` and `SoftkeyAuxMethod`
- **Imports**: `ID_NULL`, `BT_RELEASED_UNLATCHED`

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; goes to `Button_IE_B.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `SoftkeyAuxMethod.INITO`
- **IND**: Fires when the SoftKey, AUX, the local method or the additional button is activated/called

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service; goes to `Button_IE_B` and `SoftkeyAuxMethod`
- **u16ObjId** (`UINT`): Object ID of the SoftKey
- **u16ObjIdA** (`UINT`): Object ID of the `AuxFunction2`
- **u16ObjIdB** (`UINT`, default `ID_NULL`): Object ID of a further button (e.g. IO-diag test button on the outputs diagnostic page; can also be a regular button), equal to SoftKey/AUX/method
- **ID_METHOD** (`WSTRING`): Local method address (`ACTION=CREATE_METHOD`) for the argument-less trigger - called by the web client (`vt-ui-mirror`) and directly by the OPC dashboard

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully (AND of the `QO` of `Button_IE_B` and `SoftkeyAuxMethod`)

Program Flow and Connections

- `u16ObjId`, `u16ObjIdA`, `ID_METHOD` ⟶ `SoftkeyAuxMethod`
- `u16ObjIdB` ⟶ `Button_IE_B.u16ObjId`
- `SoftkeyAuxMethod.IND` ⟶ `MERGE.EI1`; `Button_IE_B.IND` ⟶ `MERGE.EI2`
- `MERGE.EO` ⟶ `IND`
- Initialization: `INIT` ⟶ `Button_IE_B.INIT`, `Button_IE_B.INITO` ⟶ `SoftkeyAuxMethod.INIT`, `SoftkeyAuxMethod.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`
- Quality: `QI` ⟶ `Button_IE_B.QI` and `SoftkeyAuxMethod.QI`; `Button_IE_B.QO` ⟶ `AND_QO.IN1`, `SoftkeyAuxMethod.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technical Details

- Version 1.1 (2026-09-15): `u16ObjId_TEST` renamed to `u16ObjIdB` (`Button_IE_TEST` ⟶ `Button_IE_B`), since the additional button is not always an IO-diag test button
- Used, among others, by `AX_SoftkeyToggle_RS_TO_AX` for the toggle branch

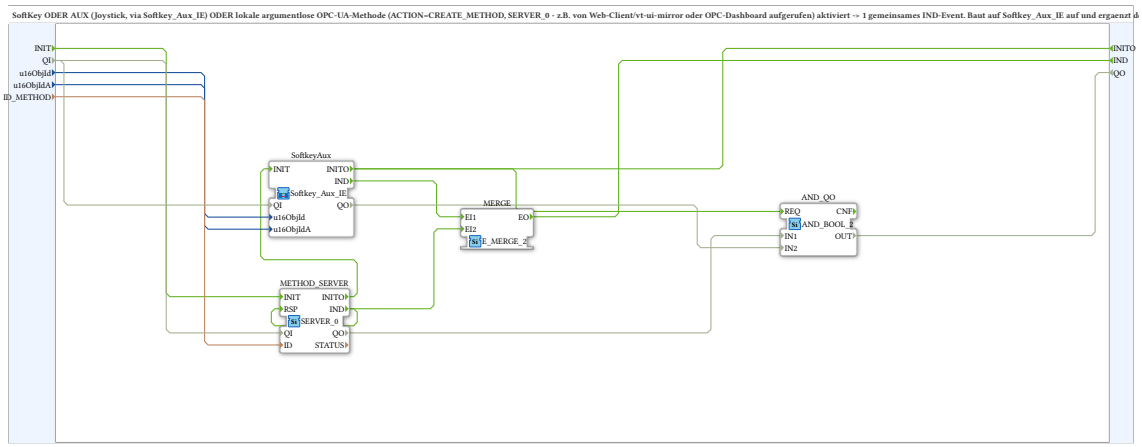
Application Scenarios

- Toggle output that is to be operated equally via SoftKey, joystick, web/dashboard and an additional diagnostic button

Summary

`Softkey_Aux_Method_Button_IE` supplements `Softkey_Aux_Method_IE` with one more button and provides a common `IND` event.

Softkey_Aux_Method_IE

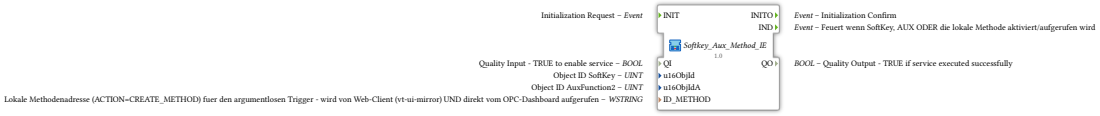


• – ○ • – ○ • – ○ •

Introduction

`Softkey_Aux_Method_IE` merges three triggers into **one common `IND` event**: a VT softkey, an AUX function (joystick, via `Softkey_Aux_IE`) or a **local argument-less OPC-UA method** (`ACTION=CREATE_METHOD`, `SERVER_0`) that is called e.g. by the web client (`vt-ui-mirror`) or by

the OPC dashboard. The block builds on `Softkey_Aux_IE` and adds the method channel. Unlike `Softkey_Aux_Web_IE` (web via `SUBSCRIBE_R_TRIG` on a written variable), the web/dashboard trigger here is a genuine RPC method call without a shared variable.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirmation
- **IND**: Fires when the softkey, AUX or the local method is activated/called

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **u16ObjId** (`UINT`): Object ID of the softkey
- **u16ObjIdA** (`UINT`): Object ID of the AuxFunction2
- **ID_METHOD** (`WSTRING`): Local method address (`ACTION=CREATE_METHOD`) for the argument-less trigger; called by the web client (`vt-ui-mirror`) and directly by the OPC dashboard

Data Outputs

- **QO** (`BOOL`): Quality Output - AND of `METHOD_SERVER.QO` and `SoftkeyAux.QO`

Adapters

This function block uses no adapters.

Function Blocks (FBs) Used

- **SoftkeyAux**: SubApp `MyLib::sys::Softkey_Aux_IE` - softkey or AUX
- **METHOD_SERVER**: `iec61499::net::SERVER_0` - local method
- **MERGE**: `iec61499::events::E_MERGE_2` - merges the two triggers
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` - combines `METHOD_SERVER.QO` and `SoftkeyAux.QO` into `QO`

Program Flow and Connections

- `SoftkeyAux.IND` ⟶ `MERGE.EI1`
- `METHOD_SERVER.IND` ⟶ `MERGE.EI2`
- `METHOD_SERVER.IND` ⟶ `METHOD_SERVER.RSP` (feedback wiring)
- `MERGE.EO` ⟶ `IND`
- `u16ObjId`, `u16ObjIdA` ⟶ `SoftkeyAux`; `ID_METHOD` ⟶ `METHOD_SERVER.ID`
- `INIT` ⟶ `METHOD_SERVER.INIT`; `METHOD_SERVER.INITO` ⟶ `SoftkeyAux.INIT`; `SoftkeyAux.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`)
- `QI` ⟶ `METHOD_SERVER.QI` and `SoftkeyAux.QI`
- `METHOD_SERVER.QO` ⟶ `AND_QO.IN1`; `SoftkeyAux.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technical Details

- **`RSP` must be wired back to `IND`.** The trigger is argument-less and needs no processing; without this connection, FORTE's `onServerMethodCall` blocks the OPC-UA server thread for up to 4 s while holding open62541's global `serviceMutex`, which every `AX_PUBLISH_1` on another FORTE thread also needs (see `Ventilsteuerung/OPC_UA_VERNETZUNG.md`, item 12).

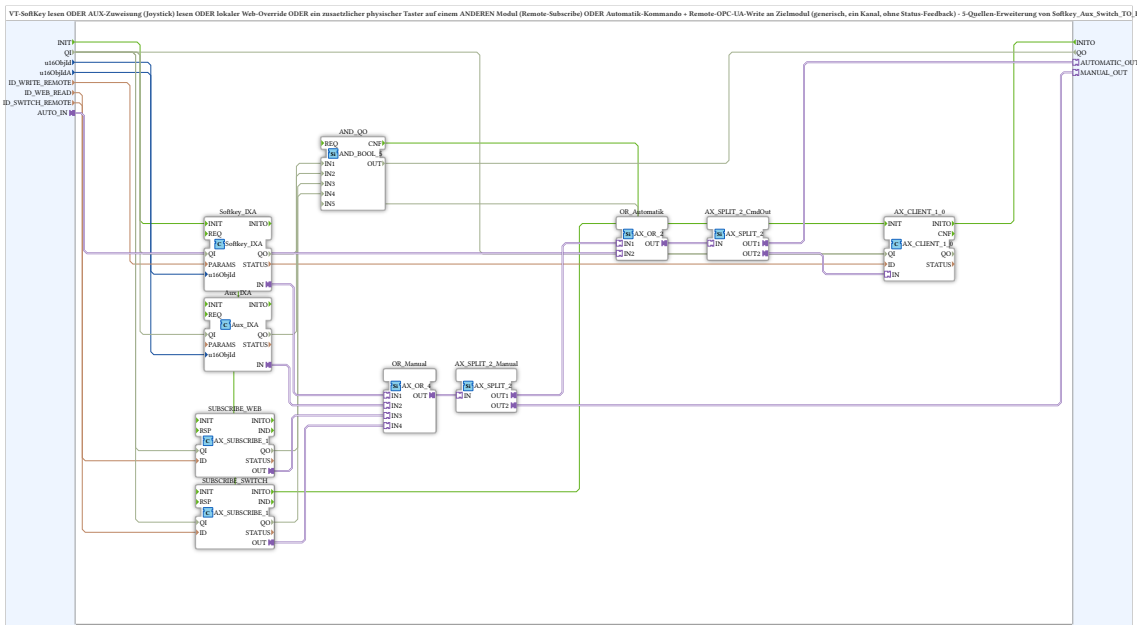
Application Scenarios

- Key, joystick assignment and web/dashboard call that trigger the same action (e.g. "All off")

Summary

`Softkey_Aux_Method_IE` combines softkey, AUX and an OPC-UA method call into one common `IND` event.

Softkey_Aux_Switch_Auto_TO_Remote_WRITE



• – ○ • – ○ • – ○ •

Introduction

`Softkey_Aux_Switch_Auto_TO_Remote_WRITE` combines, by OR, five sources of a command: the VT SoftKey, the AUX assignment (joystick), a local web override, an additional physical button on another module (remote subscribe) and an autonomous automation command (`AUTO_IN`). The result is written to a target module via remote OPC-UA write (generic, one channel, without status feedback). The block is the 5-source extension of `Softkey_Aux_Switch_TO_Remote_WRITE` by the additional socket `AUTO_IN` (e.g. automatic filling hysteresis driving the same filling belt raise line as SoftKey/AUX/web/physical button).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirm

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, TRUE enables the service
- **u16ObjId** (UINT): object ID of the SoftKey (VT), default `ID_NULL`
- **u16ObjIdA** (UINT): object ID of the AuxFunction2 (VT), default `ID_NULL`
- **ID_WRITE_REMOTE** (WSTRING): remote write address to the target module (command, `ACTION=WRITE`, CLIENT)
- **ID_WEB_READ** (WSTRING): local subscribe address for a web client (e.g. vt-ui-mirror) on the same module, ORed with SoftKey and AUX
- **ID_SWITCH_REMOTE** (WSTRING): remote subscribe address of an additional physical button on another module, ORed with SoftKey/AUX/web/automation

Data Outputs

- **QO** (`BOOL`): quality output, AND combination (`AND_QO`, `AND_BOOL_5`) of the `QO` outputs of `Softkey_IXA`, `Aux_IXA`, `SUBSCRIBE_WEB`, `SUBSCRIBE_SWITCH` and `AX_CLIENT_1_0`

Adapters

- **AUTO_IN** (socket, AX): autonomous command (e.g. automatic filling hysteresis), ORed with SoftKey/AUX/web/physical button
- **AUTOMATIC_OUT** (plug, AX): optional; locally merged raw command (SoftKey OR AUX OR web OR switch OR automation) before the remote write. Leave unwired if not needed
- **MANUAL_OUT** (plug, AX): optional; only the manual sources (SoftKey OR AUX OR web OR switch), **without** AUTO_IN. This allows, for example, an automation to be switched off when the operator intervenes manually, without the automation switching itself off through its own command. Leave unwired if not needed

Functionality

The network works in two OR stages:

1. **Manual sources**: Softkey_IXA (isobus::UT::io::Softkey::Softkey_IXA, u16ObjId) → OR_Manual.IN1, Aux_IXA (isobus::UT::io::Auxiliary::IN::Aux_IXA, u16ObjIdA) → OR_Manual.IN2, SUBSCRIBE_WEB (AX_SUBSCRIBE_1, address ID_WEB_READ) → OR_Manual.IN3, SUBSCRIBE_SWITCH (AX_SUBSCRIBE_1, address ID_SWITCH_REMOTE) → OR_Manual.IN4. OR_Manual is an AX_OR_4.
2. OR_Manual.OUT → AX_SPLIT_2_Manual: OUT1 → OR_Automatik.IN1, OUT2 → MANUAL_OUT.
3. **Automation**: AUTO_IN → OR_Automatik.IN2. OR_Automatik is an AX_OR_2.
4. OR_Automatik.OUT → AX_SPLIT_2_CmdOut: OUT1 → AUTOMATIC_OUT, OUT2 → AX_CLIENT_1_0.IN.
5. AX_CLIENT_1_0 (adapter::net::AX_CLIENT_1_0) writes the command via OPC-UA to the address ID_WRITE_REMOTE.
6. AND_QO (iec61131::booleanOperators::AND_BOOL_5) combines the QO outputs of Softkey_IXA (IN1), Aux_IXA (IN2), SUBSCRIBE_WEB (IN3), SUBSCRIBE_SWITCH (IN4) and AX_CLIENT_1_0 (IN5) into QO.
7. Initialization: INIT → Softkey_IXA.INIT → Aux_IXA.INIT → SUBSCRIBE_WEB.INIT → SUBSCRIBE_SWITCH.INIT → AX_CLIENT_1_0.INIT; AX_CLIENT_1_0.INITO → INITO and → AND_QO.REQ. QI is distributed to all five blocks.

Technical Details

- The former AX_OR_5 was split into OR_Manual (AX_OR_4) and OR_Automatik (AX_OR_2) so that MANUAL_OUT is available without the automation part; the overall behaviour for AUTOMATIC_OUT and the remote write remained unchanged
- MANUAL_OUT prevents self-shutdown: if the automation used its own command for switching off, it would switch itself off through it
- Every adapter is branched only through AX_SPLIT_2

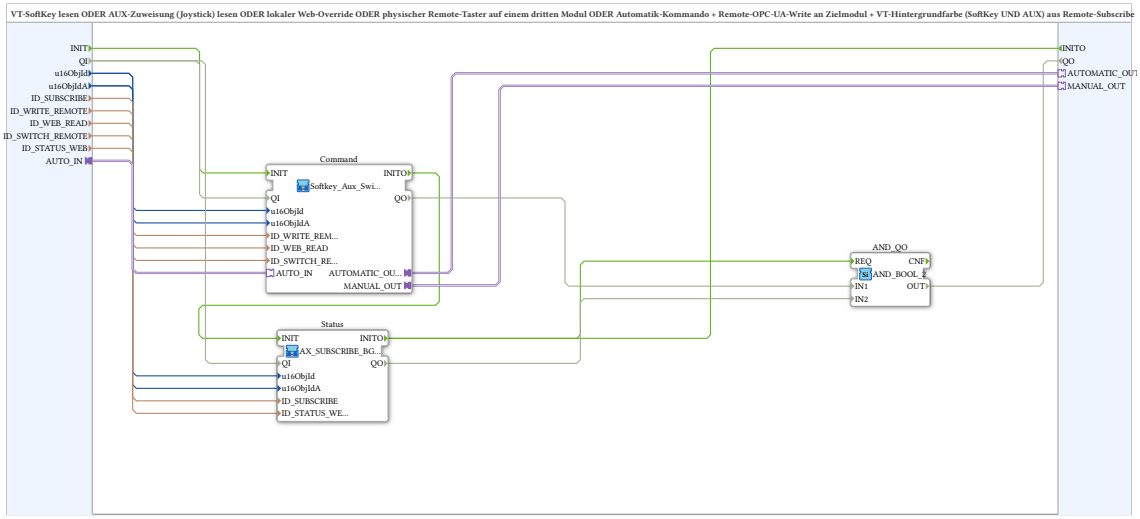
Application Scenarios

- Filling belt raise: the command can come from the operator (SoftKey/AUX/web/physical button S08) or from the automatic filling hysteresis and is written to the target module
- Automation that switches off as soon as the operator intervenes manually (via MANUAL_OUT)

Conclusion

Softkey_Aux_Switch_Auto_TO_Remote_WRITE merges five command sources by OR, writes the result to a target module via OPC-UA and optionally provides two intermediate states via AUTOMATIC_OUT and MANUAL_OUT.

Softkey_Aux_Switch_Auto_TO_Remote_WRITE_BG_OPC



• – ○ • – ○ • – ○ •

Introduction

`Softkey_Aux_Switch_Auto_TO_Remote_WRITE_BG_OPC` combines, for a command to a target module: reading the VT **SoftKey** OR reading the **AUX assignment** (joystick) OR a local **web override** OR a physical **remote switch** on a third module OR an **automatic command**, plus a **remote OPC-UA write** to the target module and the **VT background colour** (SoftKey and AUX) from a remote subscribe. It is the 5-source variant of `Softkey_Aux_Switch_TO_Remote_WRITE_BG_OPC` with an additional `AUTO_IN` socket, e.g. filling conveyor raise: SoftKey/AUX/web/physical switch S08 AND automatic filling hysteresis.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization Request

Event Outputs

- **INITO** (`EInit`): Initialization Confirm

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE to enable the service
- **u16ObjId** (`UINT`): object ID of SoftKey/background (VT)
- **u16ObjIdA** (`UINT`): object ID of AuxFunction2/background (VT)
- **ID_SUBSCRIBE** (`WSTRING`): remote subscribe address (status/colour, `ACTION=SUBSCRIBE`)
- **ID_WRITE_REMOTE** (`WSTRING`): remote write address to the target module (command, `ACTION=WRITE`, CLIENT)
- **ID_WEB_READ** (`WSTRING`): local subscribe address for a web client (e.g. vt-ui-mirror) on the same module, OR-ed with SoftKey and AUX
- **ID_SWITCH_REMOTE** (`WSTRING`): remote subscribe address of an additional physical switch on another module, OR-ed with SoftKey/AUX/web/automatic
- **ID_STATUS_WEB** (`WSTRING`): local publish address (on this module) for the state reported back by the target module, so that a web client can show the same background colour as the real VT

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service executed successfully (AND of `Command.QO` and `Status.QO`)

Adapters

All adapters are of type `adapter::types::unidirectional::AX`.

- **AUTO_IN** (socket): autonomous command (e.g. automatic filling hysteresis), OR-ed with SoftKey/Aux/web/physical switch
- **AUTOMATIC_OUT** (plug): optional, locally merged raw command, passed through from `Command.AUTOMATIC_OUT`; leave unwired if not needed
- **MANUAL_OUT** (plug): optional, only the manual sources (SoftKey/AUX/web/switch) without the automatic part, passed through from `Command.MANUAL_OUT`; leave unwired if not needed

Function Blocks (FBs) Used

- **`Command`** (SubApp `MyLib::sys::Softkey_Aux_Switch_Auto_TO_Remote_WRITE`, Softkey_Aux_Switch_Auto_TO_Remote_WRITE): forms the command from all sources and writes it remotely
- **`Status`** (SubApp `MyLib::sys::AX_SUBSCRIBE_BG3_WEB_OPC`, AX_SUBSCRIBE_BG3_WEB_OPC): subscribes to the state, colours the background and publishes it for the web client
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): ANDs `Command.QO` and `Status.QO` into `QO`

Functionality

- `u16ObjId`, `u16ObjIdA`, `ID_WRITE_REMOTE`, `ID_WEB_READ`, `ID_SWITCH_REMOTE` ⟶ the like-named inputs of `Command`
- `AUTO_IN` ⟶ `Command.AUTO_IN`
- `Command.AUTOMATIC_OUT` ⟶ `AUTOMATIC_OUT`; `Command.MANUAL_OUT` ⟶ `MANUAL_OUT`
- `u16ObjId`, `u16ObjIdA`, `ID_SUBSCRIBE`, `ID_STATUS_WEB` ⟶ the like-named inputs of `Status`
- Initialization chain: `INIT` ⟶ `Command.INIT`, `Command.INITO` ⟶ `Status.INIT`, `Status.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`
- `QI` ⟶ `Command.QI` and `Status.QI`
- `Command.QO` ⟶ `AND_QO.IN1`, `Status.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`

Technical Details

- Command path (`Command`) and status path (`Status`) are separate: the command goes to the target module via remote write, the background colour comes from the state reported back by the target module
- Optional outputs `AUTOMATIC_OUT`/`MANUAL_OUT` allow local reuse of the merged command or of the manual part only

Application Scenarios

- A mixed manual/automatic operating point, e.g. filling conveyor raise (SoftKey, AUX, web, physical switch S08 and automatic filling)

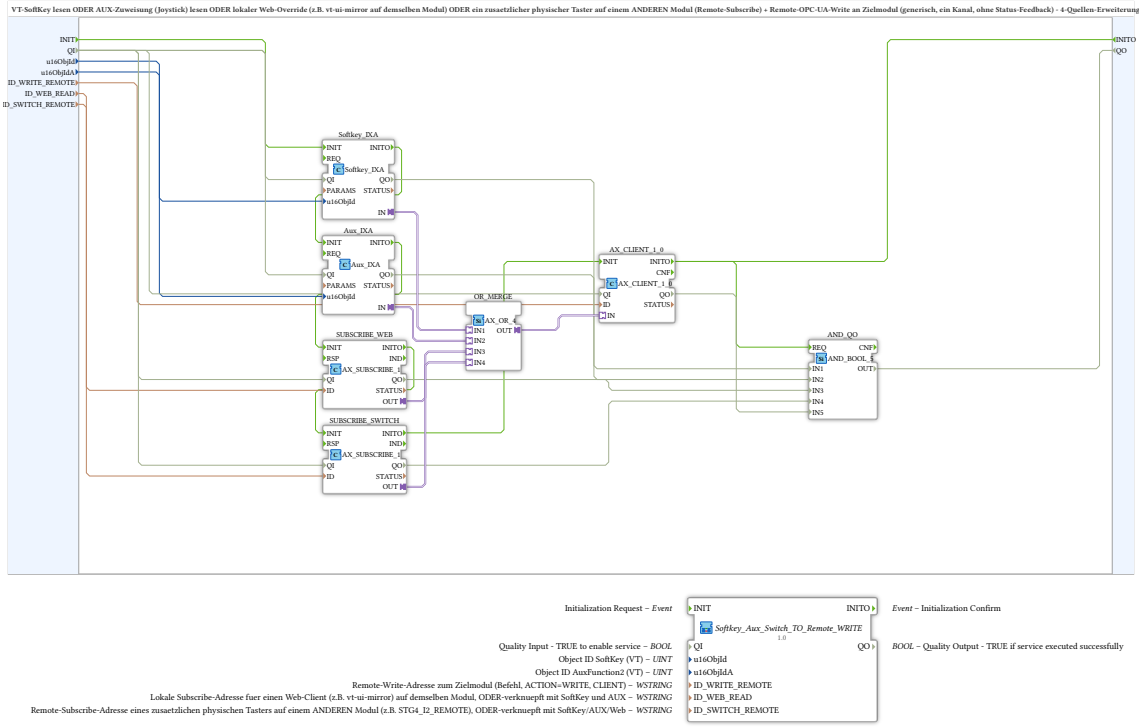
Summary

The SubApp bundles five command sources, a remote write to the target module and the reported VT background colour.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Softkey_Aux_Switch_TO_Remote_WRITE



• — ○ • — ○ • — ○ •

Introduction

The `Softkey_Aux_Switch_TO_Remote_WRITE` subapplication is a 4-source extension of the `Softkey_Aux_IXA_TO_Remote_WRITE` function block. It merges four independent command sources into a single OPC-UA write request that is sent to a remote target module. The four sources are:

1. **SoftKey** – an ISO-bus VT SoftKey (identified by `u16ObjId`).
2. **Auxiliary function** – an ISO-bus Auxiliary Function 2 (identified by `u16ObjIdA`).
3. **Local Web override** – a subscription to a web client (e.g., `vt-ui-mirror`) on the same module (`ID_WEB_READ`).
4. **Remote physical switch** – an additional physical button on a different module, obtained via a remote subscription (`ID_SWITCH_REMOTE`).

All four sources are combined using an OR-logic into a single command, which is then written to the target module via a generic one-channel OPC-UA client (`AX_CLIENT_1_0`) that uses the address provided in `ID_WRITE_REMOTE`. The subapplication is designed for scenarios such as cutting-height adjustment, where a SoftKey on STG1, a physical button on STG4, and a web override must all trigger the same actuator on STG2 – without relying on multiple independent writers that could cause race conditions or last-write-wins errors.

This block does **not** provide status feedback or background colour highlighting; for that purpose, a separate block (e.g., `AX_SUBSCRIBE_BG3_WEB_OPC`) can be placed alongside, reusing the same object IDs.

Interface Structure

The subapplication has the event input `INIT`, the event output `INITO`, six data inputs (`QI` plus the five described below) and the data output `QO`. It exposes no adapters.

Event Inputs

- **INIT** (`EInit`): initialization request; starts the chain `Softkey_IXA` ⟶ `Aux_IXA` ⟶ `SUBSCRIBE_WEB` ⟶ `SUBSCRIBE_SWITCH` ⟶ `AX_CLIENT_1_0`.

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `AX_CLIENT_1_0.INITO`.

Data Inputs

Name	Type	Initial Value	Description
QI	BOOL	TRUE	Quality input, service enable; passed on to all five internal FBs
u16ObjId	UINT	ID_NULL	Object ID of the SoftKey (VT)
u16ObjIdA	UINT	ID_NULL	Object ID of the AuxFunction2 (VT)
ID_WRITE_REMOTE	WSTRING	–	OPC-UA address of the remote target module for the write command (ACTION=WRITE, CLIENT)
ID_WEB_READ	WSTRING	–	Local subscribe address for a web client (e.g., vt-ui-mirror) on the same module, OR-combined with SoftKey and AUX
ID_SWITCH_REMOTE	WSTRING	–	Remote subscribe address of an additional physical button on a different module (e.g., STG4_I2_REMOTE), OR-combined with the other sources

Data Outputs

Name	Type	Description
QO	BOOL	Quality output, AND of the QO outputs of Softkey_IXA, Aux_IXA, SUBSCRIBE_WEB, SUBSCRIBE_SWITCH and AX_CLIENT_1_0 (via AND_QO, AND_BOOL_5)

Adapters

None (all adapter connections are internal to the subapplication).

Functionality

The subapplication implements a logical OR of four boolean command sources, followed by a single OPC-UA write to a remote target. Internally, it uses the following function blocks:

- Softkey_IXA – reads the SoftKey state via an ISO-bus adapter and outputs a boolean.
- Aux_IXA – reads the Auxiliary Function 2 state and outputs a boolean.
- SUBSCRIBE_WEB – subscribes to the ID_WEB_READ address to receive a boolean from a web client.
- SUBSCRIBE_SWITCH – subscribes to the ID_SWITCH_REMOTE address to receive a boolean from a remote physical button.
- OR_MERGE (type AX_OR_4) – combines the four boolean inputs into a single output.
- AX_CLIENT_1_0 – writes the merged boolean to the remote target using the ID_WRITE_REMOTE address.

The data flow is as follows:

1. The four sources are read continuously via their respective adapters/subscriptions.
2. The four boolean signals are fed into AX_OR_4, which produces a TRUE if any of the four is active.
3. The OR result is passed to AX_CLIENT_1_0, which sends an OPC-UA write request to the configured remote address.

Initialization and quality: INIT → Softkey_IXA.INIT, Softkey_IXA.INITO → Aux_IXA.INIT, Aux_IXA.INITO → SUBSCRIBE_WEB.INIT, SUBSCRIBE_WEB.INITO → SUBSCRIBE_SWITCH.INIT, SUBSCRIBE_SWITCH.INITO → AX_CLIENT_1_0.INIT, AX_CLIENT_1_0.INITO → INITO and → AND_QO.REQ. QI goes to all five FBs; their QO outputs feed AND_QO.IN1 to IN5, and AND_QO.OUT drives QO.

This design ensures that all commands from different inputs are combined into a single write operation, avoiding multiple writers that might conflict with each other. The subapplication is intended for use cases where a command must be triggered from multiple physical or logical locations but only one data point is written to the target.

Technical Features

- **Generic OPC-UA Client** – Uses a single AX_CLIENT_1_0 adapter to write a boolean value to a remote node. The node address is provided via ID_WRITE_REMOTE.

- **Four-input OR Combination** – Internal `AX_OR_4` merges SoftKey, AuxFunction, Web override, and remote switch signals.
- **Remote Subscription** – Uses `AX_SUBSCRIBE_1` adapters for both the web override and the remote switch, enabling cross-module communication.
- **No Status Feedback** – The subapplication only sends commands; it does not receive or process status information from the target. For feedback, a separate subscription block must be used.
- **Configurable Object IDs** – Both VT object IDs (`u16ObjId` and `u16ObjIdA`) are passed through to the internal SoftKey and Aux blocks, allowing flexible mapping to ISO-bus objects.
- **Initialisation** – The internal FB instances are no longer fixed to `QI = TRUE`; they receive the subapplication input `QI` (initial value `TRUE`) and are initialised sequentially via `INIT`/`INITO`.

State Overview

The subapplication does not implement an explicit state machine. Its behaviour is purely combinatorial: the logic continuously evaluates the four input sources and writes the OR result to the remote target. No internal states are stored, and no sequence or timing logic is applied. This makes it suitable for simple command-forwarding tasks where immediate response is required.

Application Scenarios

- **Cutting-Height Adjustment** – A SoftKey on module STG1, a physical push-button S06/S07 on module STG4, and a web override (via `vt-ui-mirror`) on STG1 all control an actuator on STG2. The subapplication combines these inputs and sends a single write command to STG2.
- **Multi-Location Emergency Stop** – When a stop command can be issued from a panel, a remote button, or a web interface, this block forwards the first activated command to a safety-relevant controller.
- **Generic Command Forwarding** – Any scenario where a boolean command must be generated from multiple sources and sent to a remote node with exactly one OPC-UA write per transition.

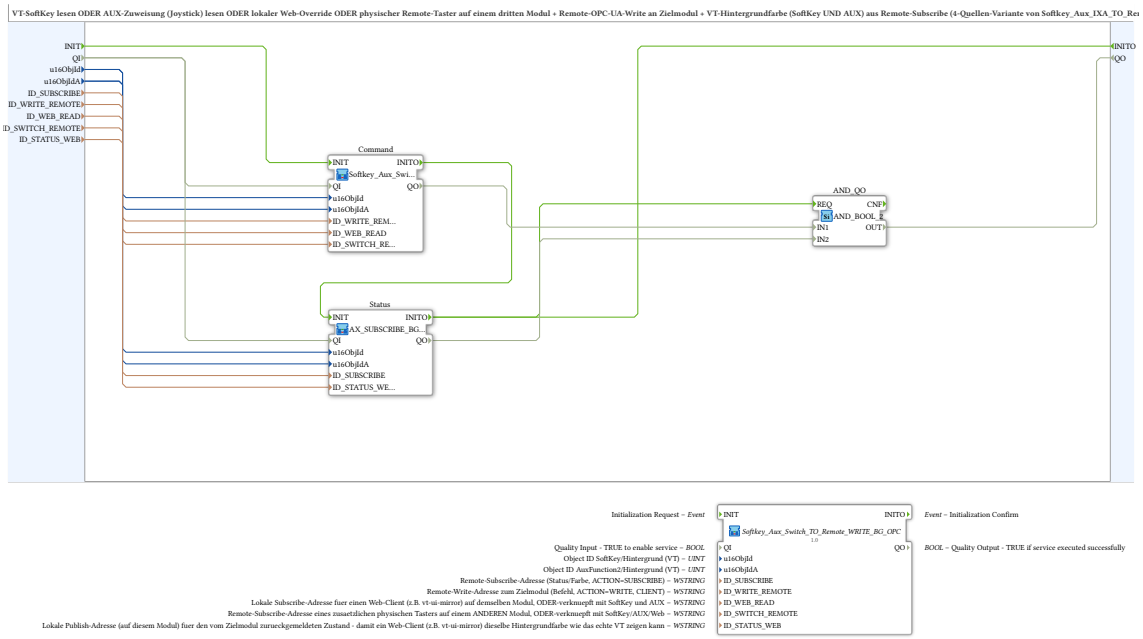
Comparison with Similar Blocks

- **vs. `Softkey_Aux_IXA_TO_Remote_WRITE`** – The present block adds a fourth input (`ID_SWITCH_REMOTE`), allowing a physical button on a third module to be included. The internal logic is identical except for the extended OR gate (4 inputs instead of 3).
- **vs. a network of separate `AX_CLIENT_1_0` blocks** – Using multiple writers to the same target would create race conditions and last-write-wins issues. This subapplication aggregates all sources into a single client, guaranteeing deterministic behaviour.
- **vs. blocks with status feedback** – This block is deliberately simple and does not provide feedback; a companion block (e.g., `AX_SUBSCRIBE_BG3_WEB_OPC`) must be used if status highlighting is required.

Conclusion

`Softkey_Aux_Switch_TO_Remote_WRITE` provides a clean, robust solution for merging four command sources into one remote OPC-UA write. Its architecture avoids common pitfalls of distributed control systems, such as conflicting writes, and keeps the interface minimal – only the necessary object IDs and communication addresses are exposed. It is well-suited for applications where multiple physical or virtual controls must trigger a single actuator or function on a remote module, especially in agricultural and industrial machinery. The block is intentionally simple, offering no status feedback and requiring an external subscription block for full UI integration.

Softkey_Aux_Switch_TO_Remote_WRITE_BG_OPC



• — ○ • — ○ • — ○ •

Introduction

The **Softkey_Aux_Switch_TO_Remote_WRITE_BG_OPC** subapplication is a thin wrapper that combines two existing subapplications into a complete OPC UA remote control and status visualization solution. It reads up to four independent input sources — a VT softkey, an AUX joystick assignment, a local web override, and a physical remote button located on a third module — OR-combines them, and writes the resulting command to a target module via OPC UA. Simultaneously, it subscribes to the target module's status and updates the background color of both the softkey and AUX elements, while republishing the status locally for web clients.

This is the 4-source variant of `Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`, intended for functions that include an additional permanently installed button on a third module, such as blade adjustment (Schnittverstellung).

Interface Structure

The subapplication exposes the events `INIT`/`INITO`, the data `QI`/`QO` and seven further data inputs, but no adapters. Internally, it connects these inputs to two child subapplications: `Command` (type `MyLib::sys::Softkey_Aux_Switch_TO_Remote_WRITE`) and `Status` (type `MyLib::sys::AX_SUBSCRIBE_BG3_WEB_OPC`).

Event Inputs

- **INIT** (`EInit`): initialization request; starts the init chain at `Command.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `Status.INITO`, the end of the init chain

Data Inputs

Name	Type	Description
`QI`	BOOL	Quality input, initial value `TRUE`; passed on to `Command.QI` and `Status.QI`.
`u16ObjId`	UINT	Object ID of the SoftKey/background element on the VT.
`u16ObjIdA`	UINT	Object ID of the AuxFunction2/background element on the VT.

ID_SUBSCRIBE	WSTRING	Remote subscribe address for status/color data (ACTION=SUBSCRIBE).
ID_WRITE_REMOTE	WSTRING	Remote write address on the target module for command data (ACTION=WRITE, CLIENT).
ID_WEB_READ	WSTRING	Local subscribe address for a web client (e.g., vt-ui-mirror) on the same module; OR-combined with SoftKey and AUX.
ID_SWITCH_REMOTE	WSTRING	Remote subscribe address for an additional physical button on another module; OR-combined with SoftKey/AUX/Web.
ID_STATUS_WEB	WSTRING	Local publish address (on this module) for the state reported back from the target module, enabling a web client to display the same background color as the real VT.

Data Outputs

- **QO** (BOOL): quality output, AND combination (AND_QO, AND_BOOL_2) of Command.QO and Status.QO

Adapters

None.

Functionality

Initialization and quality: INIT → Command.INIT, Command.INITO → Status.INIT, Status.INITO → INITO and AND_QO.REQ. QI is passed on to Command.QI and Status.QI; Command.QO → AND_QO.IN1, Status.QO → AND_QO.IN2, AND_QO.OUT → QO.

The subapplication operates in two parallel paths (with respect to data; they are connected only by the init chain and the AND combination of the quality outputs):

1. **Command path (Write):** The internal Command subapplication reads the four input sources:
   - u16ObjId — SoftKey state at the VT.
   - u16ObjIdA — AUX joystick assignment state at the VT.
   - ID_WEB_READ — local web override.
   - ID_SWITCH_REMOTE — physical button on a third module.

   These sources are OR-combined (any active source triggers the command), and the result is written to the target module via OPC UA using the address from ID_WRITE_REMOTE.
2. **Status path (Read):** The internal Status subapplication subscribes to the target module's status using ID_SUBSCRIBE. It updates the background color for both u16ObjId (SoftKey) and u16ObjIdA (AUX), and republishes the status locally via ID_STATUS_WEB so web clients can mirror the same background color.

Technical Features

- **OPC UA Integration:** Uses ACTION=WRITE (CLIENT) for commands and ACTION=SUBSCRIBE for status updates.
- **OR Logic:** Four independent input sources are logically OR-combined, providing high flexibility in control and redundancy.
- **Status Feedback Loop:** The target module's reported state is subscribed to and used for visual feedback (background color) on both the VT and connected web clients.
- **Reusable Internal Blocks:** Composed of two well-established subapplications (Softkey_Aux_Switch_TO_Remote_WRITE and AX_SUBSCRIBE_BG3_WEB_OPC), ensuring reliability and maintainability.
- **Standard Compliance:** Implements IEC 61499-2 and is licensed under the Eclipse Public License 2.0.

State Overview

The subapplication does not define an explicit state machine at its own level. The internal Command subapplication handles the OR-combination of the input sources and the remote write operation, while the internal Status subapplication handles subscription and background-color update logic. Both operate continuously, driven by the OPC UA communication events.

Application Scenarios

Typical use cases include:

- **Machine tool control panels (e.g., blade adjustment):** A physical button on a third module, combined with VT softkeys, AUX joystick assignments, and web overrides, can trigger the same function on a target module.
- **Remote visualization:** Web clients (e.g., vt-ui-mirror) can display the same background color as the physical VT, providing consistent user feedback.
- **Multi-source control:** When a function must be controllable from multiple independent locations — HMI panel, joystick, web interface, and a remote hardware button.

Comparison with Similar Blocks

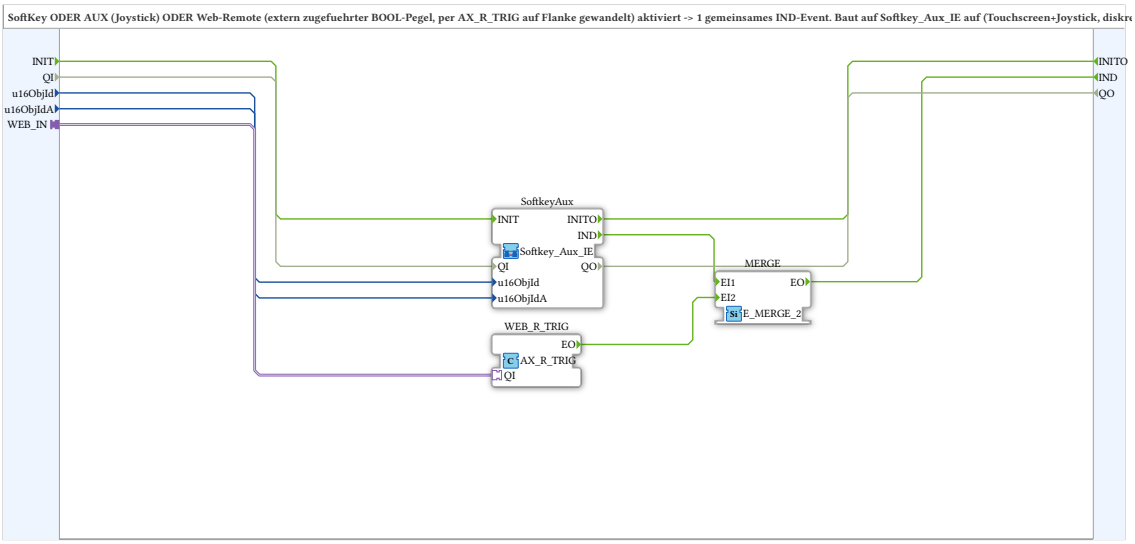
Feature	Softkey_Aux_Switch_TO_Remote_WRITE_BG_OPC (this block)	Softkey_Aux_TO_Remote_WRITE_BG_OPC (3-source variant)
Sources	4 (SoftKey, AUX, Web, Remote Switch)	3 (SoftKey, AUX, Web)
Remote hardware button on third module	Yes	No
Target module write	Yes	Yes
Status subscription / background color	Yes	Yes
Web republish	Yes	Yes

The primary difference is the additional `ID_SWITCH_REMOTE` input, which enables the integration of a physical button located on a separate module.

Conclusion

Softkey_Aux_Switch_TO_Remote_WRITE_BG_OPC is a flexible, reusable subapplication for OPC UA-based remote control with visual status feedback. By combining four independent input sources and providing both command write and status subscribe capabilities, it addresses complex control scenarios requiring input from multiple physical and virtual locations. For standard 3-source applications without an additional remote button, the 3-source variant remains the recommended choice.

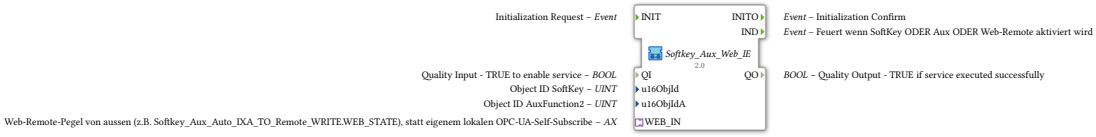
Softkey_Aux_Web_IE



• – ○ • – ○ • – ○ •

Introduction

`Softkey_Aux_Web_IE` raises a common `IND` event when **SoftKey OR AUX (joystick) OR web remote** is activated. The web channel is an externally supplied `BOOL` level (`AX`) that is converted into an edge via `AX_R_TRIG`. The block builds on `Softkey_Aux_IE` (touchscreen + joystick, discrete event channel) and additionally merges the web channel - a pattern for "activated from whatever source, one result event", e.g. for override detection.



Function Blocks (FBs) Used

Sub-blocks: Softkey_Aux_Web_IE

- **Type**: SubAppType
- **Internal blocks used**:
  - **SoftkeyAux**: SubApp `MyLib::sys::Softkey_Aux_IE` - SoftKey or AUX
  - **WEB_R_TRIG**: `adapter::events::unidirectional::AX_R_TRIG` - converts the web level into a rising edge (`EO`)
  - **MERGE**: `iec61499::events::E_MERGE_2` - merges both events

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; goes to `SoftkeyAux.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `SoftkeyAux.INITO`
- **IND**: Fires when SoftKey, Aux or web remote is activated

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service; goes to `SoftkeyAux.QI`
- **u16ObjId** (`UINT`): Object ID of the SoftKey
- **u16ObjIdA** (`UINT`): Object ID of the `AuxFunction2`

Data Outputs

- **QO** (`BOOL`): Quality Output - TRUE if the service was executed successfully; comes from `SoftkeyAux.QO`

Adapters

- **WEB_IN** (socket, `adapter::types::unidirectional::AX`): Web remote level from outside (e.g. `Softkey_Aux_Auto_IXA_TO_Remote_WRITE.WEB_STATE`) instead of its own local OPC-UA self-subscribe

Program Flow and Connections

- `u16ObjId` ⟶ `SoftkeyAux.u16ObjId`; `u16ObjIdA` ⟶ `SoftkeyAux.u16ObjIdA`
- `SoftkeyAux.IND` ⟶ `MERGE.EI1`
- `WEB_IN` ⟶ `WEB_R_TRIG.QI`; `WEB_R_TRIG.EO` ⟶ `MERGE.EI2`
- `MERGE.EO` ⟶ `IND`
- Initialization: `INIT` ⟶ `SoftkeyAux.INIT`; `SoftkeyAux.INITO` ⟶ `INITO`
- Quality: `QI` ⟶ `SoftkeyAux.QI`; `SoftkeyAux.QO` ⟶ `QO`

Technical Details

- Version 2.0 (2026-09-16): ID parameter and internal self-subscribe (`SUBSCRIBE_R_TRIG`) removed, replaced by the socket `WEB_IN` plus `AX_R_TRIG`. This fixes the double self-subscribe (FORTE: "Cannot add another one"), because the web node is already subscribed by the command side (`Softkey_Aux_Auto_IXA_TO_Remote_WRITE.WEB_STATE`)
- Only the rising edge of the web level is evaluated
- Version 1.0 (2026-09-11): initial implementation

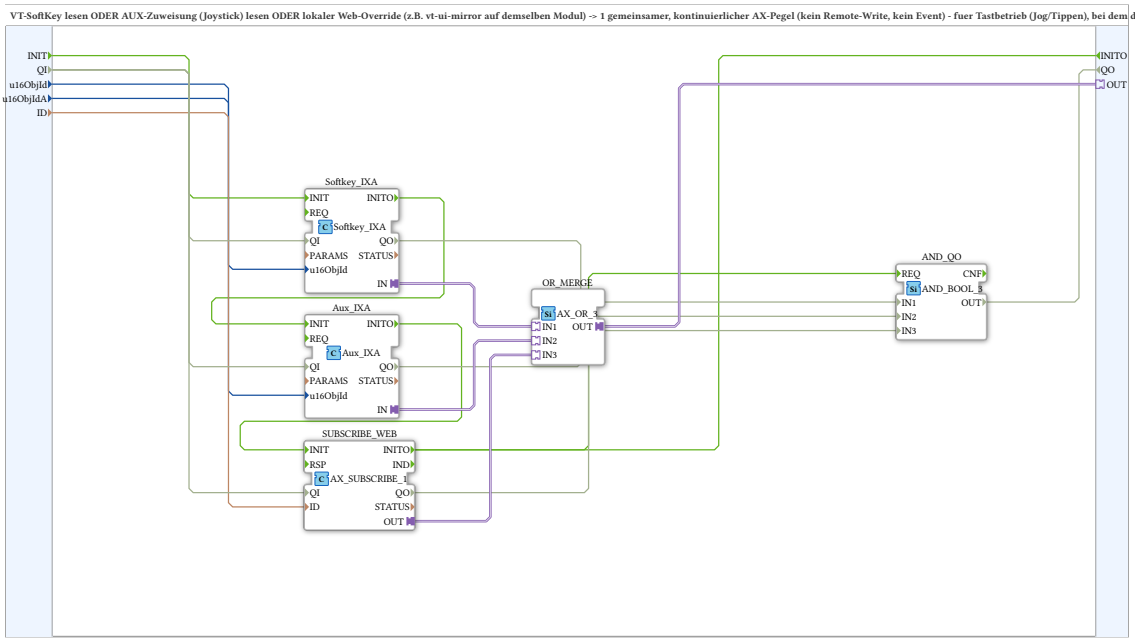
Application Scenarios

- Override detection: event as soon as the operator intervenes via SoftKey, joystick or web
- Interplay with `Softkey_Aux_Auto_IXA_TO_Remote_WRITE` (`WEB_STATE`)

Summary

`Softkey_Aux_Web_IE` bundles SoftKey, AUX and an externally supplied web level into a single `IND` event.

Softkey_Aux_Web_IXA



• — ○ • — ○ • — ○ •

Introduction

`Softkey_Aux_Web_IXA` merges a VT softkey, an AUX assignment (joystick) and a local web override (e.g. `vt-ui-mirror` on the same module) into **one common, continuous AX level** - without a remote write and without an event. It is intended for jog operation, where the level itself (not an edge) is to be processed further, e.g. as the `G` input of an `AX_AX_SEL_AR` or as `DI1` of a GreenWhiteBackground feedback. It corresponds to the command half of `Softkey_Aux_IXA_TO_Remote_WRITE`, but provides `OUT` instead of a remote write.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirmation

Apart from `INITO` there are no event outputs - user events run via the adapter plug.

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the service
- **u16ObjId** (`UINT`, default `ID_NULL`): Object ID of the softkey (VT)
- **u16ObjIdA** (`UINT`, default `ID_NULL`): Object ID of the AuxFunction2 (VT)

- **ID** (`WSTRING`): Local subscribe address for a web client (e.g. `vt-ui-mirror`) on the same module, ORed with softkey and AUX

Data Outputs

- **QO** (`BOOL`): Quality Output - AND of `Softkey_IXA.QO`, `Aux_IXA.QO` and `SUBSCRIBE_WEB.QO`

Adapters

- **OUT** (plug, `adapter::types::unidirectional::AX`): Softkey OR aux OR web override, continuous level (`TRUE` as long as held/active)

Function Blocks (FBs) Used

- **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA`
- **Aux_IXA**: `isobus::UT::io::Auxiliary::IN::Aux_IXA`
- **SUBSCRIBE_WEB**: `adapter::net::AX_SUBSCRIBE_1`
- **OR_MERGE**: `adapter::booleanOperators::AX_OR_3`
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3`

Program Flow and Connections

- `u16ObjId` ⟶ `Softkey_IXA.u16ObjId`; `u16ObjIdA` ⟶ `Aux_IXA.u16ObjId`; `ID` ⟶ `SUBSCRIBE_WEB.ID`
- `Softkey_IXA.IN` ⟶ `OR_MERGE.IN1`
- `Aux_IXA.IN` ⟶ `OR_MERGE.IN2`
- `SUBSCRIBE_WEB.OUT` ⟶ `OR_MERGE.IN3`
- `OR_MERGE.OUT` ⟶ `OUT`
- `INIT` ⟶ `Softkey_IXA.INIT`; `Softkey_IXA.INITO` ⟶ `Aux_IXA.INIT`; `Aux_IXA.INITO` ⟶ `SUBSCRIBE_WEB.INIT`; `SUBSCRIBE_WEB.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`)
- `QI` ⟶ `Softkey_IXA.QI`, `Aux_IXA.QI` and `SUBSCRIBE_WEB.QI`
- `Softkey_IXA.QO` ⟶ `AND_QO.IN1`; `Aux_IXA.QO` ⟶ `AND_QO.IN2`; `SUBSCRIBE_WEB.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`

Technical Details

- Continuous level instead of an edge: `OUT` stays `TRUE` as long as at least one source is active.
- No remote write and no user event output (only `INITO`).

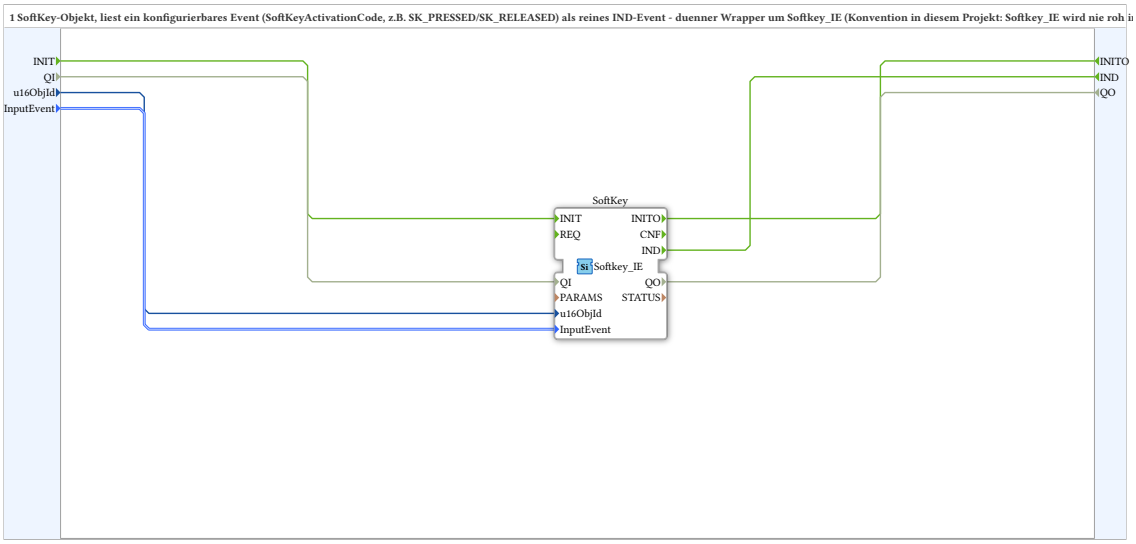
Application Scenarios

- Jog operation where the level is to go to a selector or a background color feedback

Summary

`Softkey_Aux_Web_IXA` ORs softkey, aux and web override into one AX level.

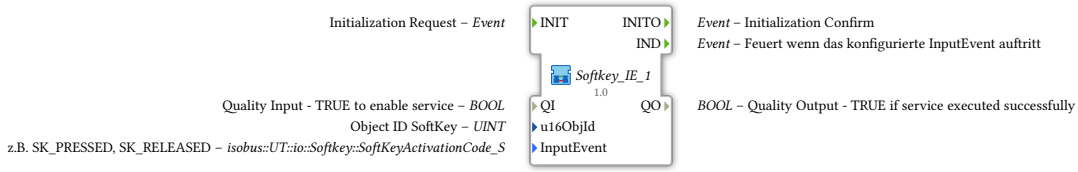
Softkey_IE_1



• – ◦ • – ◦ • – ◦ •

Introduction

`Softkey_IE_1` reads a SoftKey object and outputs a configurable event (`SoftKeyActivationCode`, e.g. `SK_PRESSED`/`SK_RELEASED`) as a pure `IND` event. The block is a thin wrapper around `isobus::UT::io::Softkey::Softkey_IE`, analogous to `Aux_IE_1` and `Button_IE_1`. Convention in this project: `Softkey_IE` is never used raw inline but always wrapped.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirm
- **IND**: fires when the configured `InputEvent` occurs

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, TRUE enables the service
- **u16ObjId** (UINT): object ID of the SoftKey (default `ID_NULL`)
- **InputEvent** (`isobus::UT::io::Softkey::SoftKeyActivationCode_S`): e.g. `SK_PRESSED`, `SK_RELEASED` (default `Invalid`)

Data Outputs

- **QO** (`BOOL`): quality output, taken directly from `SoftKey.QO`

Adapters

No adapters are used.

Functionality

The network contains exactly one instance `SoftKey` of type `isobus::UT::io::Softkey::Softkey_IE` (`QI` comes from the subapp input `QI`). `u16ObjId` and `InputEvent` are routed unchanged to the like-named inputs of the instance, and its event `IND` is output directly as the `IND` of the SubApp.

Technical Details

- Pure pass-through without logic or state of its own
- `QI` is exposed as an input (default `TRUE`) and `QO` comes directly from `SoftKey.QO`; left unwired, the block is therefore active
- Because `InputEvent` is a parameter, the same SubApp can be used for different kinds of events (e.g. pressing or releasing)

Application Scenarios

- Reading a single VT operation as an event without using the raw IE block directly in the network

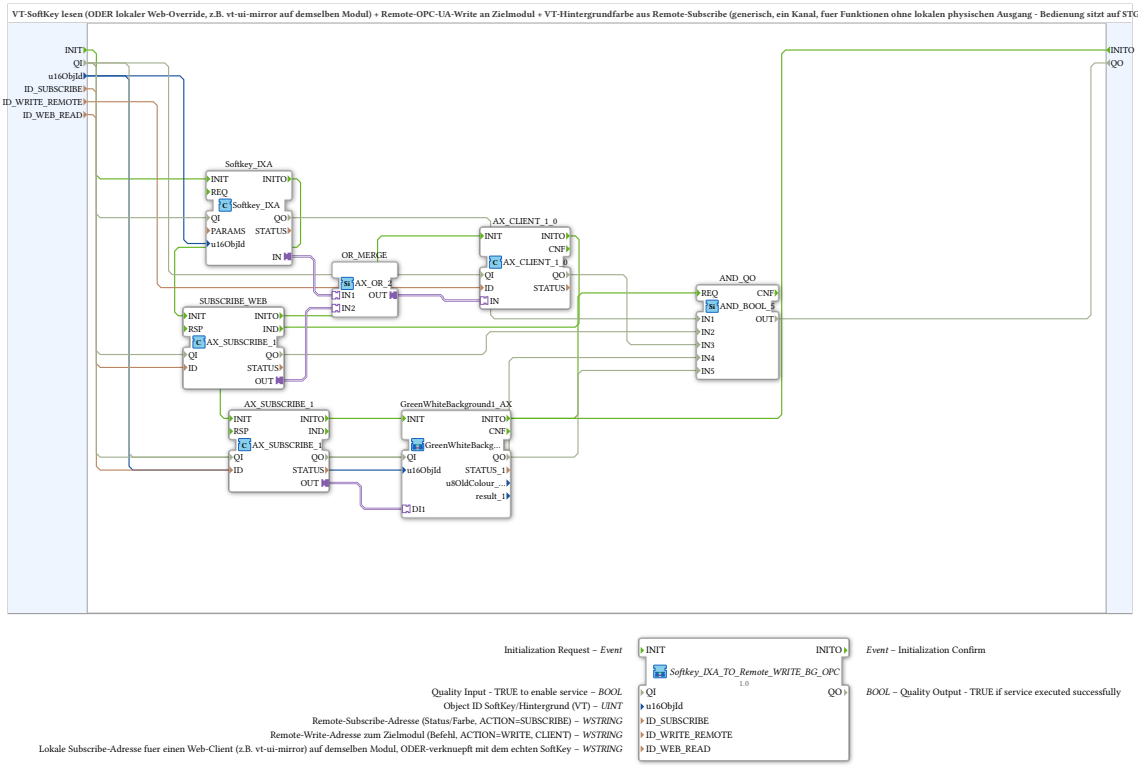
⚖️ Comparison with Similar Blocks

Equivalent wrappers are `Aux_IE_1` and `Button_IE_1` they differ only in the VT object type and in the enumeration type of `InputEvent`.

Conclusion

`Softkey_IE_1` is a minimal wrapper that provides a configurable SoftKey event as `IND`.

Softkey_IXA_TO_Remote_WRITE_BG_OPC



• — ○ • — ○ • — ○ •

Introduction

This subapplication is designed for scenarios where the operator interface is located on a VT (Virtual Terminal) device (STG1), but the actual actuator resides on a different module that does not have direct VT access. The subapplication reads a local softkey, optionally merges it with a web-based override, sends the resulting command to the remote module via OPC UA, and then receives the actual state back from the remote module to update the softkey background colour accordingly. This creates a closed-loop feedback mechanism that keeps the operator informed of the real actuator status.

Interface Structure

The subapplication exposes the initialization events `INIT`/`INITO`, the status ports `QI`/`QO` and data inputs. It has no adapter interfaces at the subapplication level.

Event Inputs

- **INIT** (`EInit`): Initialization Request

Event Outputs

- **INITO** (`EInit`): Initialization Confirm

Data Inputs

Name	Type	Initial Value	Comment
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE enables the service.
`u16ObjId`	`UINT`	`ID_NULL`	Object ID of the softkey / background (VT).
`ID_SUBSCRIBE`	`WSTRING`	–	Remote subscribe address (status/colour, `ACTION=SUBSCRIBE`).
`ID_WRITE_REMOTE`	`WSTRING`	–	Remote write address to the target module (command, `ACTION=WRITE`, `CLIENT`).

ID_WEB_READ	WSTRING	–	Local subscribe address for a web client (e.g. vt-ui-mirror) on the same module, OR-combined with the real softkey.

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if the service executed successfully (AND of Softkey_IXA, SUBSCRIBE_WEB, AX_CLIENT_1_0, AX_SUBSCRIBE_1 and GreenWhiteBackground1_AX).

Adapters

None.

Functionality

The internal logic of the subapplication consists of seven blocks:

1. Softkey_IXA – reads the local softkey state based on the configured u16ObjId.
2. SUBSCRIBE_WEB – subscribes to a local web-based input (e.g. a mirror of the VT display) using the ID_WEB_READ address.
3. OR_MERGE – an AX_OR_2 boolean operator that combines the softkey signal and the web override signal using a logical OR.
4. AX_CLIENT_1_0 – an OPC UA client that writes the merged signal to the remote device via the ID_WRITE_REMOTE endpoint.
5. AX_SUBSCRIBE_1 – an OPC UA subscriber that receives the actual state from the remote device via ID_SUBSCRIBE.
6. GreenWhiteBackground1_AX – a subapplication that issues a Q_BackgroundColour command to change the softkey background colour based on the received state (green for active, white for inactive).
7. AND_QO – an iec61131::booleanOperators::AND_BOOL_5 block that ANDs the QO outputs of Softkey_IXA, SUBSCRIBE_WEB, AX_CLIENT_1_0, AX_SUBSCRIBE_1 and GreenWhiteBackground1_AX into the subapplication output QO.

Initialization and status: INIT ⟶ Softkey_IXA.INIT, Softkey_IXA.INITO ⟶ SUBSCRIBE_WEB.INIT, SUBSCRIBE_WEB.INITO ⟶ AX_CLIENT_1_0.INIT, AX_CLIENT_1_0.INITO ⟶ AX_SUBSCRIBE_1.INIT, AX_SUBSCRIBE_1.INITO ⟶ GreenWhiteBackground1_AX.INIT, GreenWhiteBackground1_AX.INITO ⟶ INITO and ⟶ AND_QO.REQ. QI ⟶ Softkey_IXA.QI, SUBSCRIBE_WEB.QI, AX_CLIENT_1_0.QI, AX_SUBSCRIBE_1.QI and GreenWhiteBackground1_AX.QI; their QO outputs ⟶ AND_QO.IN1..IN5; AND_QO.OUT ⟶ QO.

The overall data flow is:

- The local softkey and the optional web override are OR-merged.
- The merged signal is written to the remote module via OPC UA.
- The remote module's actual state is read back via OPC UA subscription.
- The state is used to set the background colour of the softkey, giving immediate visual feedback.

Technical Features

- ✅ OPC UA client and subscriber functionality built in.
- ✅ Local softkey reading via Softkey_IXA.
- ✅ Optional integration with web-based UI mirroring (vt-ui-mirror).
- ✅ OR logic for merging multiple input sources.
- ✅ Remote state feedback with background colour update.
- ✅ Only the initialization handshake (INIT/INITO) and the aggregated status QO are exposed – the subapplication acts as a self-contained control unit.
- ✅ Configurable via the address/object-ID inputs plus QI.

State Overview

The subapplication does not contain an explicit state machine. It operates purely combinatorially – the current input values determine the output behaviour. The only internal states are those of the individual

function blocks (e.g. connection states of the OPC UA client and subscriber), but these are managed internally and are not exposed to the user. The dynamic behaviour is that of a closed-loop control: the displayed colour always reflects the most recent remote state.

Application Scenarios

- **Remote actuator control from a VT** – when the operator interface is on STG1 (VT), but the actuator is on another module without VT capabilities.
- **Softkey with web-override** – when operators may also use a local web-based mirror to trigger the same function.
- **Visual status feedback** – the softkey background changes to green/white to indicate the actual actuator state, not just the pressed command.
- **Generic single-channel usage** – suitable for any function that does not have a physical output on the VT itself but requires remote actuation and feedback.

Comparison with Similar Blocks

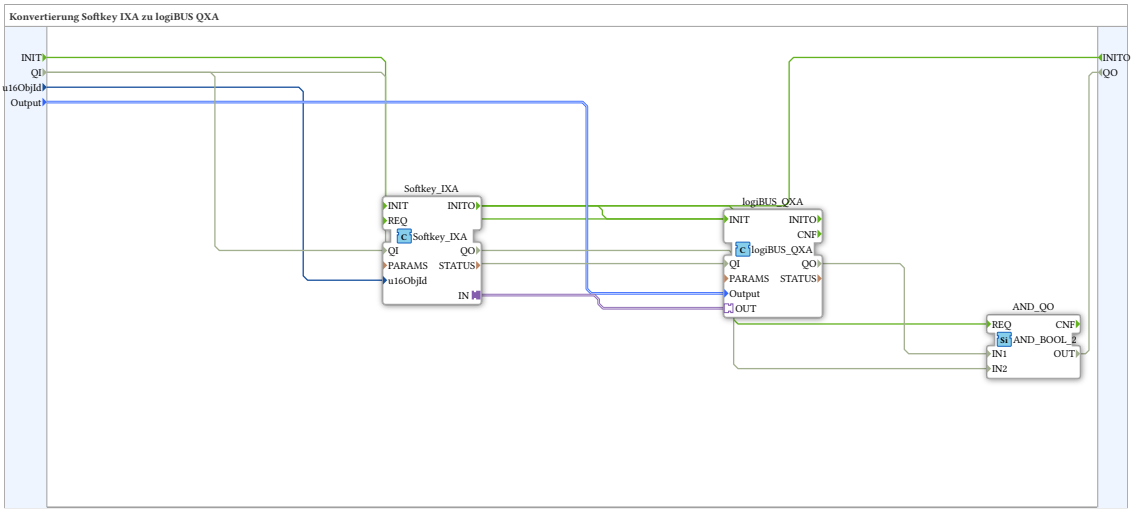
- **`Softkey_Aux_IXA_TO_Remote_WRITE_BG_OPC`** – extends this subapplication with a third input source (e.g. an auxiliary function object like a joystick) and uses `GreenWhiteBackground3_AX` instead of `1_AX` for a three-state background.
- **`Button_IXA_TO_Remote_WRITE_BG_OPC`** – identical functionality but for a VT button instead of a softkey.
- **`Softkey_IXA_TO_Remote_WRITE_BG_OPC`** – the single-channel version described here, optimised for functions with exactly one operator-initiated source (plus optional web override).

These variants share the same core architecture but differ in input multiplicity and background colour handling.

Conclusion

`Softkey_IXA_TO_Remote_WRITE_BG_OPC` provides a robust and reusable solution for bridging VT-based operator inputs to remote actuators over OPC UA. Its integration of local softkey reading, web override, OR logic, remote write, and remote state subscription ensures reliable operation with clear visual feedback. The simple interface and configurable addresses make it easy to adapt to various modules and use cases, while the underlying design supports extensibility to more complex scenarios.

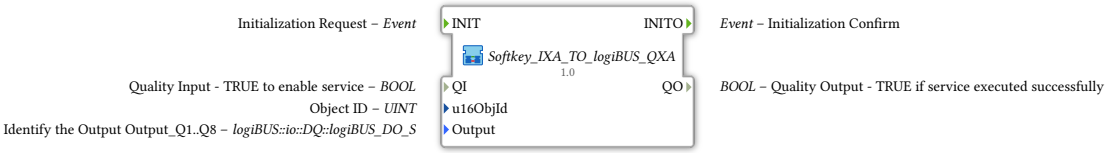
Softkey_IXA_TO_logiBUS_QXA



• – ○ • – ○ • – ○ •

Introduction

`Softkey_IXA_TO_logiBUS_QXA` connects a VT softkey (`Softkey_IXA`) directly to a physical digital output (`logiBUS_QXA`) — functionally identical to `Button_IXA_TO_logiBUS_QXA`, but for softkeys instead of buttons.



Function Blocks (FBs) Used

Sub-blocks: Softkey_IXA_TO_logiBUS_QXA

- **Type**: SubAppType
- **Internal FBs used**:
  - **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA` — VT softkey adapter.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physical digital output.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — combines `logiBUS_QXA.QO` and `Softkey_IXA.QO` into the SubApp output `QO`.
- **Interface**: event input `INIT`, event output `INITO`, data input `QI` (default `TRUE`, passed on to both FBs), data output `QO`.
- **Functionality**: Direct adapter connection `Softkey_IXA.IN` ⟶ `logiBUS_QXA.OUT`, no intermediate logic (apart from the `AND_QO` combination of the quality signals).

Program Flow and Connections

1. `u16ObjId` ⟶ `Softkey_IXA.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`.
2. `Softkey_IXA.IN` ⟶ `logiBUS_QXA.OUT`.
3. Initialization: `INIT` ⟶ `logiBUS_QXA.INIT`; `logiBUS_QXA.INITO` ⟶ `Softkey_IXA.INIT`; `Softkey_IXA.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
4. Quality: `QI` ⟶ `logiBUS_QXA.QI` and `Softkey_IXA.QI`; `logiBUS_QXA.QO` ⟶ `AND_QO.IN1`, `Softkey_IXA.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Application Scenarios

- Minimal softkey-to-output block with no status display or remote control.

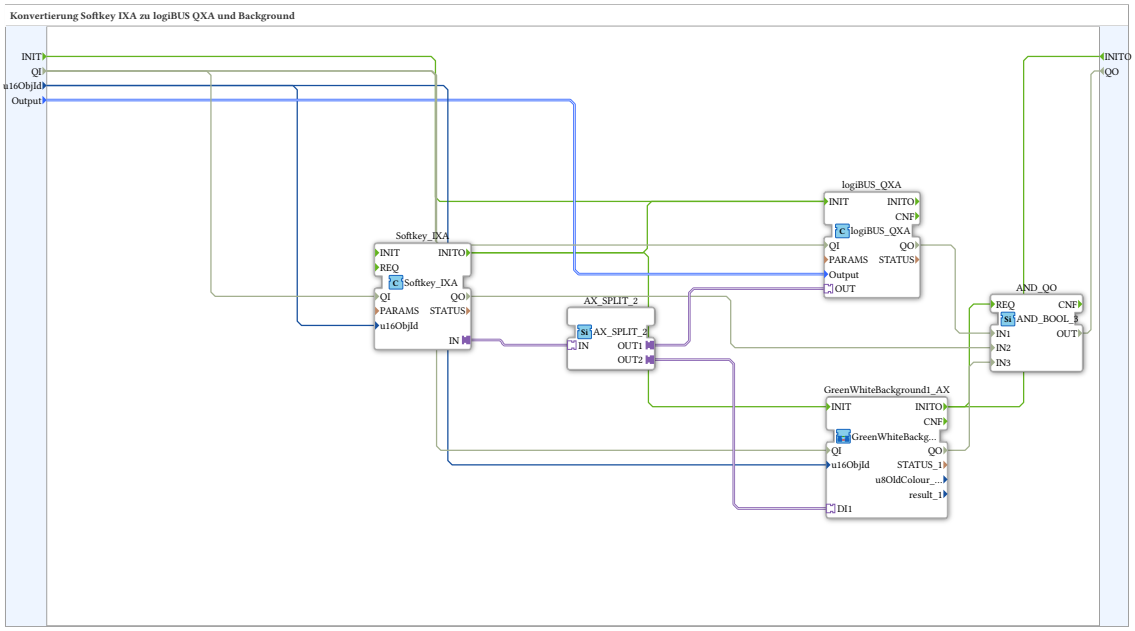
Summary

Softkey counterpart to `Button_IXA_TO_logiBUS_QXA` — identical pattern, different control type.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

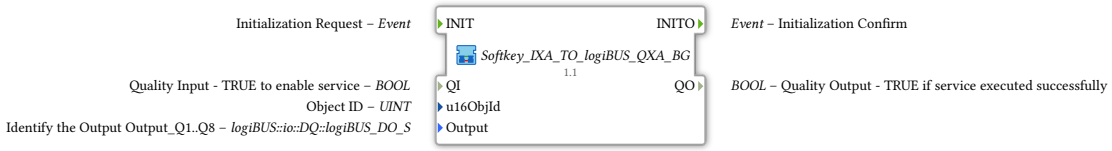
Softkey_IXA_TO_logiBUS_QXA_BG



• – ○ • – ○ • – ○ •

Introduction

`Softkey_IXA_TO_logiBUS_QXA_BG` extends `Softkey_IXA_TO_logiBUS_QXA` with a VT status color — the softkey counterpart to `Button_IXA_TO_logiBUS_QXA_BG`.



Function Blocks (FBs) Used

Sub-blocks: Softkey_IXA_TO_logiBUS_QXA_BG

- **Type**: SubAppType
- **Internal FBs used**:
  - **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA` — VT softkey adapter.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physical digital output.
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — splits the softkey signal.
  - **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): sets the VT status color (see Background Color Blocks).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — combines `logiBUS_QXA.QO`, `Softkey_IXA.QO` and `GreenWhiteBackground1_AX.QO` into the output `QO`.
- **Functionality**: `Softkey_IXA.IN` is distributed via `AX_SPLIT_2` to both `logiBUS_QXA.OUT` and `GreenWhiteBackground1_AX.DI1`. The SubApp interface additionally has the events `INIT`/`INITO` as well as `QI` (BOOL, initial value `TRUE`) and `QO` (BOOL).

Program Flow and Connections

1. `u16ObjId` ⟶ `Softkey_IXA.u16ObjId` and `GreenWhiteBackground1_AX.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`.
2. `Softkey_IXA.IN` ⟶ `AX_SPLIT_2.IN` ⟶ `AX_SPLIT_2.OUT1` ⟶ `logiBUS_QXA.OUT`, `AX_SPLIT_2.OUT2` ⟶ `GreenWhiteBackground1_AX.DI1`.
3. Initialization chain: `INIT` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `Softkey_IXA.INIT`, `Softkey_IXA.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`, `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` and `AND_QO.REQ`.

4. Quality: `QI` ⟶ `logiBUS_QXA.QI`, `Softkey_IXA.QI` and `GreenWhiteBackground1_AX.QI`; their `QO` outputs ⟶ `AND_QO.IN1` to `AND_QO.IN3` (in this order); `AND_QO.OUT` ⟶ `QO`.

Application Scenarios

- VT softkey with direct visual feedback, without OPC-UA.

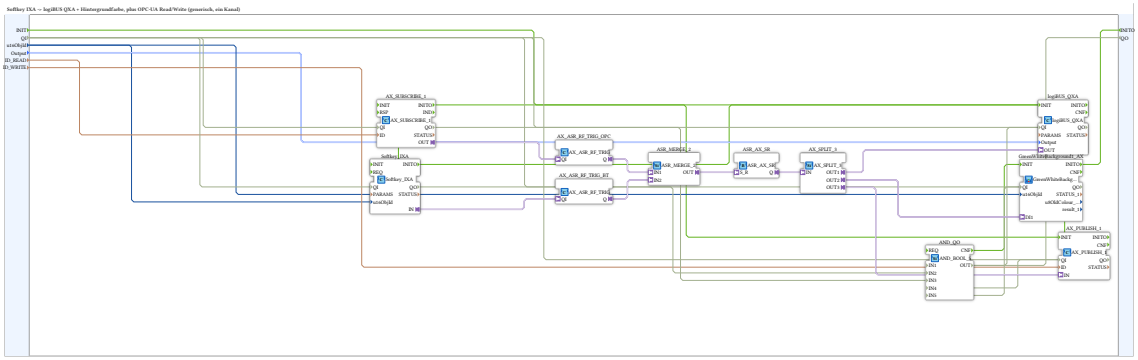
Summary

Softkey counterpart to `Button_IXA_TO_logiBUS_QXA_BG` — identical pattern, different control type.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

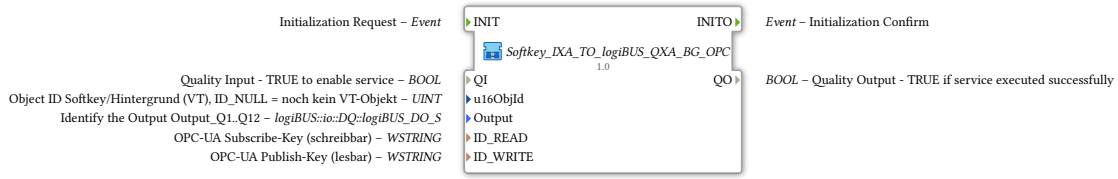
Softkey_IXA_TO_logiBUS_QXA_BG_OPC



• — ○ • — ○ • — ○ •

Introduction

`Softkey_IXA_TO_logiBUS_QXA_BG_OPC` is the SoftKey counterpart to `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`: A VT SoftKey and an OPC UA remote command are each converted into set/reset events via edge detection and combined in a common latch. This is **not** a true click-toggle: pressing triggers `SET`, releasing triggers `RESET` — with only one source active, that's purely momentary behavior. The difference from a simple OR gate only shows up once the SoftKey and the OPC UA command overlap in time: whichever edge arrives last always wins, regardless of source (last-wins), instead of the output staying ON as long as either source is active.



Function Blocks (FBs) Used

Sub-Blocks: Softkey_IXA_TO_logiBUS_QXA_BG_OPC

- **Type**: SubAppType
- **Internal FBs Used**:
- **Softkey_IXA**: `isobus::UT::io::Softkey::Softkey_IXA` — VT SoftKey, `u16ObjId` identifies the SoftKey.
- **AX_ASR_RF_TRIG_BT** / **AX_ASR_RF_TRIG_OPC**: each `adapter::events::unidirectional::AX_ASR_RF_TRIG` — convert the rising/falling edge of the SoftKey or the OPC UA Subscribe value into a Set/Reset event pair (ASR).
- **ASR_MERGE_2**: `adapter::events::unidirectional::ASR_MERGE_2` — combines the two ASR sources (SoftKey, OPC UA).

- **ASR_AX_SR**: `adapter::events::unidirectional::ASR_AX_SR` — Set/Reset Latch, creates the latched output state from the combined ASR signal.
- **AX_SPLIT_3**: `adapter::events::unidirectional::AX_SPLIT_3` — distributes the latch output to three destinations.
- **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physical digital output.
- **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): VT background color matching the output state.
- **AX_SUBSCRIBE_1** / **AX_PUBLISH_1**: each `adapter::net::AX_SUBSCRIBE_1/AX_PUBLISH_1` — OPC UA read/write access.
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_5` — combines the `QO` of `logiBUS_QXA`, `Softkey_IXA`, `AX_SUBSCRIBE_1`, `AX_PUBLISH_1` and `GreenWhiteBackground1_AX`.
- **Functionality**: Identical to the latching structure of `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`, but with `Softkey_IXA` instead of `Button_IXA` as the local input source.

Program Flow and Connections

1. `Softkey_IXA.IN` (adapter) ⟶ `AX_ASR_RF_TRIG_BT.QI` ⟶ `AX_ASR_RF_TRIG_BT.Q` ⟶ `ASR_MERGE_2.IN2`.
2. `AX_SUBSCRIBE_1.OUT` ⟶ `AX_ASR_RF_TRIG_OPC.QI` ⟶ `AX_ASR_RF_TRIG_OPC.Q` ⟶ `ASR_MERGE_2.IN1`.
3. `ASR_MERGE_2.OUT` ⟶ `ASR_AX_SR.S_R`.
4. `ASR_AX_SR.Q` ⟶ `AX_SPLIT_3.IN` ⟶ `AX_SPLIT_3.OUT1` ⟶ `logiBUS_QXA.OUT`, `AX_SPLIT_3.OUT2` ⟶ `GreenWhiteBackground1_AX.DI1`, `AX_SPLIT_3.OUT3` ⟶ `AX_PUBLISH_1.IN`.
5. Initialization: event input `INIT` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `Softkey_IXA.INIT`, `Softkey_IXA.INITO` ⟶ `AX_SUBSCRIBE_1.INIT`, `AX_SUBSCRIBE_1.INITO` ⟶ `AX_PUBLISH_1.INIT` (hidden connection), `AX_PUBLISH_1.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`, `GreenWhiteBackground1_AX.INITO` ⟶ event output `INITO` and ⟶ `AND_QO.REQ`.
6. Parameters: `u16ObjId` ⟶ `Softkey_IXA.u16ObjId` and `GreenWhiteBackground1_AX.u16ObjId`; `Output` ⟶ `logiBUS_QXA.Output`; `ID_READ` ⟶ `AX_SUBSCRIBE_1.ID`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`.
7. Quality: data input `QI` (`BOOL`, default `TRUE`) ⟶ `logiBUS_QXA.QI`, `Softkey_IXA.QI`, `AX_SUBSCRIBE_1.QI`, `AX_PUBLISH_1.QI` and `GreenWhiteBackground1_AX.QI`; their `QO` ⟶ `AND_QO.IN1`..`IN5`; `AND_QO.OUT` ⟶ data output `QO`.

Technical Features

- **Last-wins via ASR latch, not a real toggle**: As with `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`, the behavior is purely momentary with only one source active (ON only while held). The difference from a momentary OR gate only shows up once both sources overlap: whichever edge (set or reset, from either source) arrives last determines the state.
- **Two independent last-wins sources**: A soft key and an OPC UA command each independently deliver set/reset events; `ASR_MERGE_2` combines both into a common latch input without prioritizing either source.

Application Scenarios

- Digital outputs with two independent switching sources (soft key AND remote OPC UA command) where whichever action arrives last should win, instead of one permanently active source blocking the other, with VT status display and OPC UA feedback.

Comparison with Similar Modules

Structurally identical to `Button_IXA_TO_logiBUS_QXA_BG_OPC_LATCHING`, except that `Softkey_IXA` is used instead of `Button_IXA` as the local control source — including the same last-wins characteristic (not click-toggle). Compared to the simpler `Softkey_IXA_TO_logiBUS_QXA_BG` (only local softkey, no OPC UA connection, no latch logic), this module includes the complete edge detection/merge/latch chain as well as OPC UA subscribe/publish functionality.

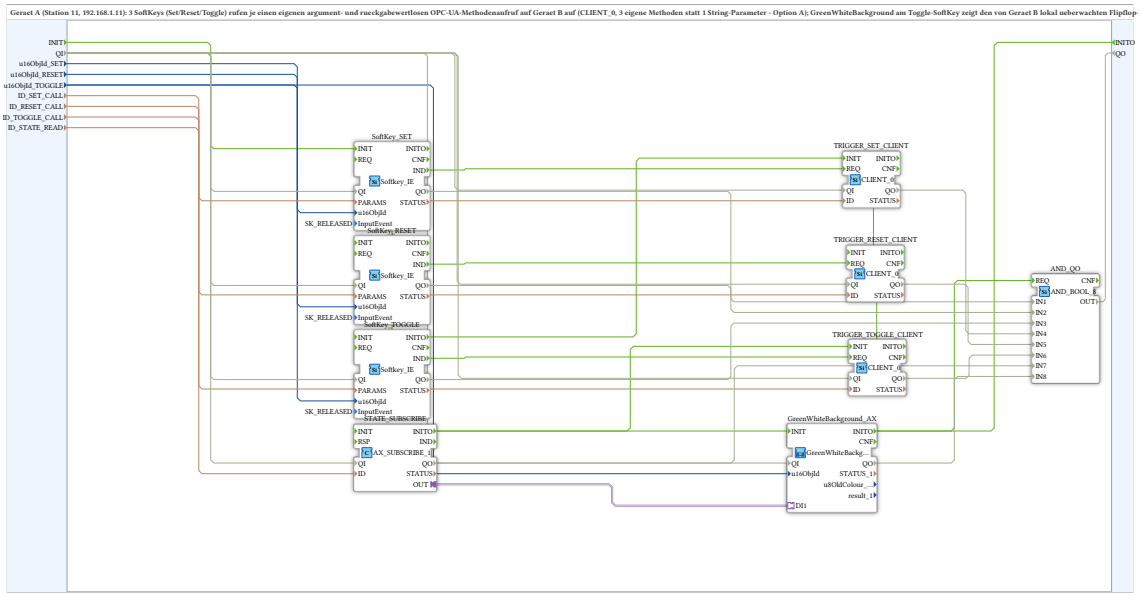
Summary

`Softkey_IXA_TO_logiBUS_QXA_BG_OPC` switches a digital logiBUS output via a soft key and/or a remote OPC UA command, including VT status color and OPC UA echo. With only one source active the behavior is purely momentary; when both sources overlap, last-wins applies instead of a true OR gate.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Softkey_SRT_RPC_TO_Remote_BG_OPC



• — ○ • — ○ • — ○ •

Introduction

The **Softkey_SRT_RPC_TO_Remote_BG_OPC** subapplication is a composite type designed for a two-device OPC-UA automation scenario. It runs on **Device A** (Station 11, 192.168.1.11) and controls three softkeys – **Set**, **Reset**, and **Toggle** – each of which triggers its own argument-free, return-value-free OPC-UA method call on a remote **Device B**. Instead of passing a string parameter to a single generic method, this design (Option A) uses three distinct remote method addresses, providing clearer semantics and easier maintenance. Additionally, a **GreenWhiteBackground** indication on the Toggle softkey reflects a flip-flop state that is locally monitored on Device B and made available to Device A via an OPC-UA variable subscription. The entire protocol logic is encapsulated inside the `MyLib::sys` composite, keeping the device resource simple and reusable.

Interface Structure

The subapplication exposes only data inputs at its interface. All event handling and adapter logic are internal to the composite.

Event Inputs

- **INIT** (`EInit`): initialization request. Apart from that, the subapplication does not expose any event inputs. Activation is performed internally by the softkey function blocks, which react to hardware events.

Event Outputs

- **INITO** (EInit): initialization confirmation (from GreenWhiteBackground_AX.INITO). No other events are propagated to the outside world.

Data Inputs

Name	Type	Description
QI	BOOL	Quality Input, TRUE enables the service (default TRUE).
u16ObjId_SET	UINT	Object ID of the **Set** softkey. Initial value: ID_NULL.
u16ObjId_RESET	UINT	Object ID of the **Reset** softkey. Initial value: ID_NULL.
u16ObjId_TOGGLE	UINT	Object ID of the **Toggle** softkey (also carries the GreenWhiteBackground indication). Initial value: ID_NULL.
ID_SET_CALL	WSTRING	Remote method address (ACTION=CALL_METHOD) for the **Set** method call on Device B.
ID_RESET_CALL	WSTRING	Remote method address (ACTION=CALL_METHOD) for the **Reset** method call on Device B.
ID_TOGGLE_CALL	WSTRING	Remote method address (ACTION=CALL_METHOD) for the **Toggle** method call on Device B.
ID_STATE_READ	WSTRING	Locally monitored OPC-UA address (BOOL, ACTION=READ) for the flip-flop state that is remotely written by Device B.

Data Outputs

- **QO** (BOOL): Quality Output, AND (AND_QO, iec61131::booleanOperators::AND_BOOL_8) of the QO outputs of all eight internal blocks (3 × Softkey_IE, 3 × CLIENT_0, STATE_SUBSCRIBE, GreenWhiteBackground_AX).

Adapters

None at the interface level. Internally, an AX_SUBSCRIBE_1 adapter is used to establish the subscription for the state variable.

Functionality

The subapplication implements the following end-to-end behavior:

1. **Softkey Detection** – Three isobus::UT::io::Softkey::Softkey_IE instances (SoftKey_SET, SoftKey_RESET, SoftKey_TOGGLE) are configured with InputEvent = SK_RELEASED (activation on key release); their QI comes from the QI input of the subapplication. Each instance is associated with a distinct Object ID supplied via the u16ObjId_* inputs.
2. **Remote Method Invocation** – When a softkey is pressed, the corresponding IND event is routed to the REQ input of a dedicated iec61499::net::CLIENT_0 function block (TRIGGER_SET_CLIENT, TRIGGER_RESET_CLIENT, TRIGGER_TOGGLE_CLIENT). Each client is enabled via the QI input of the subapplication and holds the remote method address in its ID input, which is wired from the corresponding ID_*_CALL data input.
3. **State Subscription** – An adapter::net::AX_SUBSCRIBE_1 block (STATE_SUBSCRIBE) continuously subscribes to the OPC-UA variable specified in ID_STATE_READ. Its output (an adapter interface) is connected to the DI1 adapter socket of a MyLib::sys::GreenWhiteBackground1_AX subapplication.
4. **Initialization and Quality** – INIT runs through a chain SoftKey_SET → SoftKey_RESET → SoftKey_TOGGLE → TRIGGER_SET_CLIENT → TRIGGER_RESET_CLIENT → TRIGGER_TOGGLE_CLIENT → STATE_SUBSCRIBE → GreenWhiteBackground_AX (each INITO → next INIT); GreenWhiteBackground_AX.INITO → INITO (and → AND_QO.REQ). QI is distributed to .QI of all eight blocks; their QO outputs feed AND_QO.IN1..IN8, and AND_QO.OUT → QO.
5. **Visual Indication** – The GreenWhiteBackground1_AX subapplication, fed with u16ObjId_TOGGLE and the subscribed state value, renders either a green or a white background on the Toggle softkey, reflecting the current flip-flop state maintained on Device B.

Technical Features

- **Fully composite design** – The protocol logic is contained within a `MyLib::sys` composite, keeping the device resource lean and making the subapplication portable across different resources or devices.
- **Dedicated OPC-UA method clients** – Three separate `CLIENT_0` instances, one per softkey, allow independent configuration and fault isolation for each remote call.
- **No parameter passing** – The remote methods are intentionally argument-free, simplifying the OPC-UA server implementation on Device B and avoiding string-parsing overhead.
- **Event-driven activation** – Softkey events are captured on release (`SK_RELEASED`), providing deterministic triggering.
- **Subscription-based state feedback** – The flip-flop state is obtained via OPC-UA subscription (not polling), ensuring timely and bandwidth-efficient updates.
- **Configurable identifiers** – Object IDs and remote addresses are injected at instantiation time, allowing the same subapplication to be reused with different hardware layouts or OPC-UA server configurations.
- **EPL-2.0 licensing** – The subapplication is published under the Eclipse Public License 2.0.

State Overview

The subapplication itself does not contain an explicit state machine. However, its constituent elements exhibit the following state-oriented behavior:

- **Softkey FBs** – Each `Softkey_IE` is in an *idle* state until a hardware-level key-release event occurs. Upon detection, the FB generates an `IND` event and returns to the idle state.
- **Client FBs** – Each `CLIENT_0` transitions through *request sent* → *response received* → *idle* for every triggered call. While `QI` is TRUE, the clients remain operationally ready.
- **State variable (remote)** – The flip-flop state on Device B is a single BOOL (binary) value. It transitions between `0` (white background) and `1` (green background) each time the Toggle method is invoked. The `STATE_SUBSCRIBE` adapter continuously tracks changes and forwards them to the background rendering component.

The overall system thus implements a classic **set / reset / toggle** control paradigm with visual feedback, where the state source of truth resides on Device B.

Application Scenarios

This subapplication is well suited for:

- **Industrial control panels** – Physical or virtual softkeys on a local operator panel (Device A) that remotely operate actuators, valves, or machine functions on a PLC or controller (Device B).
- **Multi-device OPC-UA architectures** – Where one device needs to invoke discrete, side-effect-free methods on another device, and the current state must be displayed locally.
- **Toggle-with-feedback user interfaces** – For example, enabling/disabling a production line parameter, where the background color of the key gives the operator an immediate visual confirmation of the actual remote state.
- **Modular and portable HMI components** – Because all protocol logic is embedded in the composite, the same subapplication can be dropped into different 4diac projects without modifying the resource.

Comparison with Similar Blocks

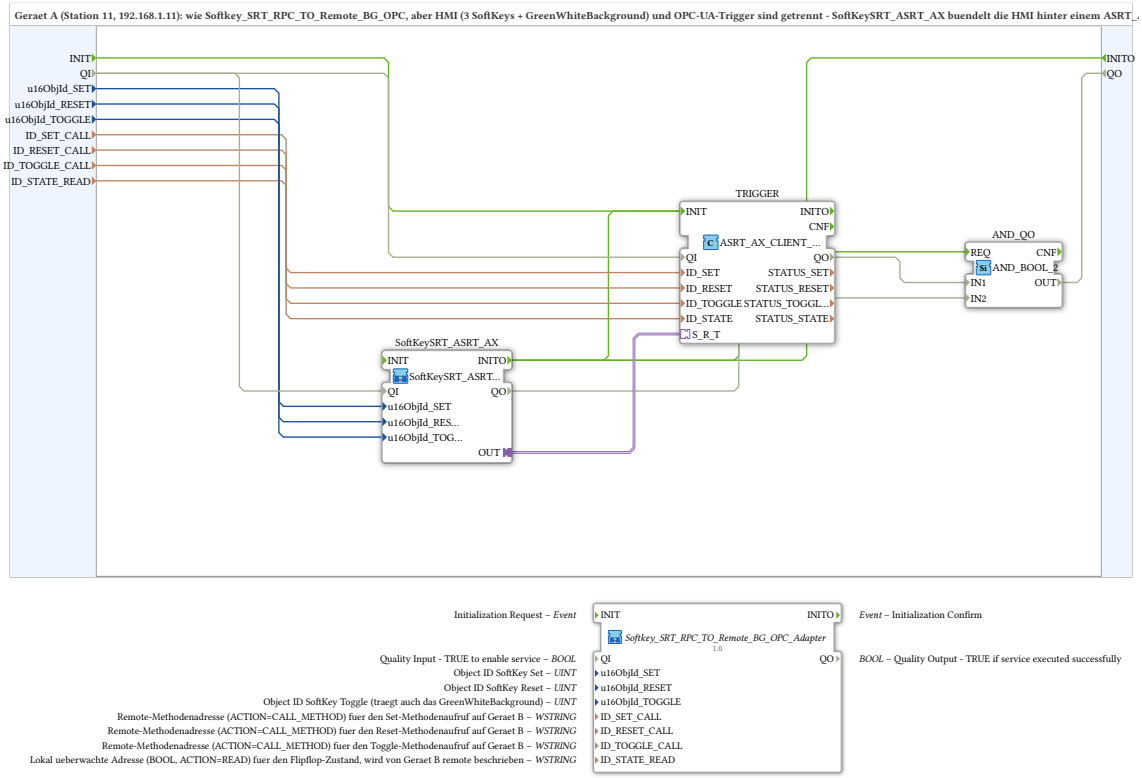
Feature / Aspect	**Softkey_SRT_RPC_TO_Remote_BG** (Option A)	Remote_Softkey_Single-Method-With-String-Parameter (Option B)	Generic OPC-UA Client FB (non-dedicated)
Remote method granularity	Three dedicated methods (Set / Reset / Toggle)	One method that accepts a string parameter to select the action	User must supply address and parameters per call
Parameter handling	No arguments; minimal server complexity	Server must parse the string, incurring overhead and error risk	Flexible but requires manual mapping
State feedback	Built-in subscription and GreenWhiteBackground visualization	Typically requires an additional subscription block	Not included – must be added externally

Error isolation	High – failure of one method call does not affect the others	Lower – a parsing error in the string affects all calls	Depends on configuration, often shared
Configuration effort	Moderate – three addresses must be provided	Low – only one address, but the string must be constructed	High – full OPC-UA node configuration required
Clarity / maintainability	Excellent – each call is explicit and self-documenting	Acceptable – but string contracts are error-prone	Depends on the engineer's discipline
Suitability for industrial HMI	Very high – designed for operator panels with visual feedback	Medium – more suited for ad-hoc remote calls	Medium – versatile but not tailored to softkey use cases

Conclusion

The **Softkey_SRT_RPC_TO_Remote_BG_OPC** subapplication provides a clean, reusable, and industrially oriented solution for controlling three softkey-driven remote method invocations over OPC-UA, combined with live state visualization. Its dedicated-method approach (Option A) eliminates string-parsing pitfalls and improves failure isolation, while the subscription-based GreenWhiteBackground feedback gives operators immediate and reliable confirmation of the remote flip-flop state. By encapsulating all protocol details within a `MyLib::sys` composite, the subapplication promotes modularity, portability, and maintainability across different 4diac projects. This makes it an excellent building block for modern HMI and SCADA-style applications built on the Eclipse 4diac framework.

Softkey_SRT_RPC_TO_Remote_BG_OPC_Adapter



• – ○ • – ○ • – ○ •

Introduction

The **Softkey_SRT_RPC_TO_Remote_BG_OPC_Adapter** is a composite subapplication designed for distributed automation scenarios, specifically for Device A (Station 11, IP 192.168.1.11). It realizes a remote

control architecture where three SoftKeys (Set, Reset, Toggle) and a GreenWhiteBackground indicator are operated via an HMI on Device A, while the actual flip-flop logic is executed on a remote Device B.

This subapplication separates the HMI interaction from the OPC-UA trigger mechanism: the `SoftKeySRT_ASRT_AX` subapplication bundles the HMI logic behind an ASRT_AX plug, while the `ASRT_AX_CLIENT_0_SUBSCRIBE_1` FB bundles three CLIENT_0 instances plus an AX_SUBSCRIBE_1 behind an ASRT_AX socket. The communication protocol is implemented inside the `MyLib::sys` composite, not in the device resource, following the "SUB style" architectural pattern.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirm

Data Inputs

Name	Type	Initial Value	Comment
`QI`	BOOL	`TRUE`	Quality input - TRUE enables the service.
`u16ObjId_SET`	UINT	`ID_NULL`	Object ID for the SoftKey Set function.
`u16ObjId_RESET`	UINT	`ID_NULL`	Object ID for the SoftKey Reset function.
`u16ObjId_TOGGLE`	UINT	`ID_NULL`	Object ID for the SoftKey Toggle function (also carries the GreenWhiteBackground state).
`ID_SET_CALL`	WSTRING	–	Remote method address (`ACTION=CALL_METHOD`) for the Set method call on Device B.
`ID_RESET_CALL`	WSTRING	–	Remote method address (`ACTION=CALL_METHOD`) for the Reset method call on Device B.
`ID_TOGGLE_CALL`	WSTRING	–	Remote method address (`ACTION=CALL_METHOD`) for the Toggle method call on Device B.
`ID_STATE_READ`	WSTRING	–	Locally monitored address (`BOOL`, `ACTION=READ`) for the flip-flop state, written remotely by Device B.

Data Outputs

- **QO** (`BOOL`): quality output, AND combination (`AND_QO`, `AND_BOOL_2`) of the `QO` outputs of `TRIGGER` and `SoftKeySRT_ASRT_AX`

Adapters

The subapplication contains two internal components connected via an adapter link:

- **SoftKeySRT_ASRT_AX** (type `MyLib::sys::SoftKeySRT_ASRT_AX`) – provides the HMI/SoftKey logic and exposes a plug.
- **TRIGGER** (type `adapter::net::ASRT_AX_CLIENT_0_SUBSCRIBE_1`) – provides the OPC-UA client/subscription functionality and exposes a socket.

The adapter connection is established between `SoftKeySRT_ASRT_AX.OUT` (plug) and `TRIGGER.S_R_T` (socket).

Functionality

The subapplication implements a remote SoftKey control loop with the following data flows:

1. **HMI Signal Acquisition** – The three SoftKey object IDs (`u16ObjId_SET`, `u16ObjId_RESET`, `u16ObjId_TOGGLE`) are forwarded to the `SoftKeySRT_ASRT_AX` subapplication, which interprets HMI input and maintains the GreenWhiteBackground state.

2. **Remote Method Invocation** – The remote method addresses (`ID_SET_CALL`, `ID_RESET_CALL`, `ID_TOGGLE_CALL`) are passed to the `TRIGGER` FB. This FB uses OPC-UA `CALL_METHOD` actions to invoke the corresponding methods on Device B.
3. **State Monitoring** – The `ID_STATE_READ` address is monitored via an OPC-UA `READ` action to obtain the flip-flop state, which is written remotely by Device B.
4. **Adapter Coupling** – The `SoftKeySRT_ASRT_AX` output is connected to the `TRIGGER` socket through an ASRT_AX adapter, decoupling the HMI logic from the OPC-UA transport details.

Technical Features

- **Modular architecture** – Separation of HMI handling (SoftKeySRT_ASRT_AX) and OPC-UA communication (TRIGGER) improves reusability and testability.
- **Protocol encapsulation** – The communication protocol resides within the `MyLib::sys` composite library, not in the device resource, enabling easier maintenance and versioning.
- **Parameterized addresses** – All remote end-points are configured via WSTRING inputs, allowing flexible deployment without modifying the internal network.
- **Initialization support** – Object IDs are initialized to `ID_NULL` (imported from `isobus::UT::Q::const::IDs`), ensuring a well-defined startup state.
- **Quality wiring** – `QI` (default `TRUE`) is passed to the `TRIGGER` FB and to `SoftKeySRT_ASRT_AX` (there is no hard-wired `QI` parameter any more); the `QO` outputs of both are combined by `AND_QO` (`AND_BOOL_2`) into `QO`. Initialization runs as `INIT` ⟶ `TRIGGER.INIT`; `TRIGGER.INITO` ⟶ `SoftKeySRT_ASRT_AX.INIT`; `SoftKeySRT_ASRT_AX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.

State Overview

The subapplication itself does not maintain an explicit state machine; the state logic is contained in the embedded `SoftKeySRT_ASRT_AX` subapplication. The overall system reflects three operational states:

- **Idle** – No SoftKey pressed, flip-flop state unknown/initial.
- **Set / Reset** – Corresponding remote method called on Device B, state updated via `ID_STATE_READ`.
- **Toggle** – Remote toggle method executed; the GreenWhiteBackground indicator reflects the new flip-flop state.

The flip-flop state is continuously readable via the locally monitored `ID_STATE_READ` address.

Application Scenarios

This subapplication is suited for:

- **Distributed HMI panels** – Controlling remote actuators or logic from a central operator station.
- **OPC-UA based machine control** – When the actual SoftKey logic resides on a different device than the HMI.
- **Reusable SoftKey libraries** – Where the same HMI pattern is applied to multiple remote targets by changing the object IDs and method addresses.
- **Complex automation cells** – Such as station-based production lines where station 11 (Device A) supervises station functions on Device B.

Comparison with Similar Blocks

Compared to the simpler `Softkey_SRT_RPC_TO_Remote_BG_OPC` (where HMI and OPC-UA trigger are combined), this adapter variant:

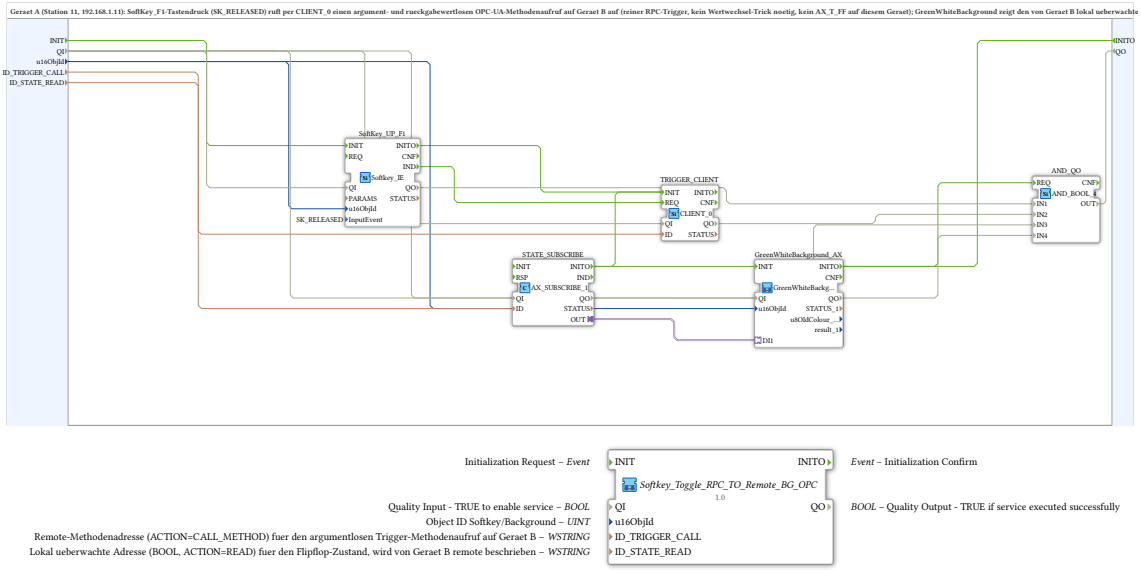
- **Separates concerns** – HMI processing and OPC-UA subscription are individually replaceable.
- **Reduces resource footprint** – Bundles three CLIENT_0 instances and one AX_SUBSCRIBE_1 into a single ASRT_AX socket, simplifying wiring.
- **Improves maintainability** – The protocol logic lives in the `MyLib::sys` composite, making it easier to update without touching device resources.
- **Enables plug-and-play** – The ASRT_AX adapter pattern allows rapid exchange of either the HMI or the communication layer.

Conclusion

The `Softkey_SRT_RPC_TO_Remote_BG_OPC_Adapter` is a well-structured subapplication that cleanly separates HMI SoftKey handling from OPC-UA based remote method invocation and state monitoring. By leveraging adapter-based design and encapsulating the protocol in a library composite, it provides a flexible, scalable, and maintainable solution for distributed automation scenarios. Its parameterized inter-

face makes it adaptable to various object IDs and remote addresses without code changes, significantly reducing engineering effort in multi-station systems.

Softkey_Toggle_RPC_TO_Remote_BG_OPC



• — ○ • — ○ • — ○ •

Introduction

Softkey_Toggle_RPC_TO_Remote_BG_OPC is a composite subapplication used on device A, e.g. Station 11 at IP address 192.168.1.11. It combines a softkey input, an OPC UA client request, and a state-monitoring adapter to provide a remote toggle control with visual feedback.

When the F1 softkey is released, represented by `SK_RELEASED`, the subapplication sends an argumentless and return-value-less OPC UA method call to a remote device B through a `CLIENT_0` function block. This is a pure remote procedure call trigger. No value-change trick and no additional flip-flop block is required on device A. The actual flip-flop state is monitored on device B and displayed locally by a `GreenWhiteBackground` composite, which shows either green or white depending on the remote state.

The protocol logic is encapsulated inside the `MyLib::sys` composite subapplication, not directly in the device resource.

Interface Structure

The subapplication exposes the initialization events `INIT`/`INITO`, the status ports `QI`/`QO` and data inputs. It has no externally visible adapter interfaces.

Event Inputs

Name	Type	Description
`INIT`	`EInit`	Initialization Request

All other event generation is handled internally by the softkey input block.

Event Outputs

Name	Type	Description
`INITO`	`EInit`	Initialization Confirm

Data Inputs

Name	Type	Initial Value	Description
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE enables the service.

u16ObjId	UINT	ID_NULL	Object ID used for the softkey and the background element.
ID_TRIGGER_CALL	WSTRING	–	Remote OPC UA method address with ACTION=CALL_METHOD. Used for the argument-less trigger method call on device B.
ID_STATE_READ	WSTRING	–	Locally monitored OPC UA address for the flip-flop state. The address is of type BOOL and uses ACTION=READ. It is written remotely by device B.

Data Outputs

Name	Type	Description
QO	BOOL	Quality Output - TRUE if the service executed successfully (AND of SoftKey_UP_F1, TRIGGER_CLIENT, STATE_SUBSCRIBE and GreenWhiteBackground_AX).

Adapters

None in the external interface.
Internally, the adapter output STATE_SUBSCRIBE.OUT is connected to the adapter input GreenWhiteBackground_AX.DI1.

Functionality

The subapplication performs two main tasks, triggering a remote OPC UA method call and visualizing the remote flip-flop state, plus the initialization and status wiring.

1. SoftKey_UP_F1, an instance of isobus::UT::io::Softkey::Softkey_IE, is configured with InputEvent = SK_RELEASED and receives QI from the subapplication's QI input.
2. When the F1 softkey is released, the internal event output IND of SoftKey_UP_F1 is emitted.
3. This event is connected to the REQ input of TRIGGER_CLIENT, an instance of iec61499::net::CLIENT_0.
4. TRIGGER_CLIENT uses the ID_TRIGGER_CALL address to execute the remote OPC UA method on device B. The method has no arguments and no return value, so a single event-triggered request is sufficient.
5. In parallel, STATE_SUBSCRIBE, an instance of adapter::net::AX_SUBSCRIBE_1, monitors the ID_STATE_READ address and provides the current flip-flop state through its adapter output OUT.
6. This adapter output is connected to GreenWhiteBackground_AX.DI1.
7. GreenWhiteBackground_AX uses u16ObjId to address the correct background element and displays the state as green or white.
8. Initialization and status: INIT → SoftKey_UP_F1.INIT, SoftKey_UP_F1.INITO → TRIGGER_CLIENT.INIT, TRIGGER_CLIENT.INITO → STATE_SUBSCRIBE.INIT, STATE_SUBSCRIBE.INITO → GreenWhiteBackground_AX.INIT, GreenWhiteBackground_AX.INITO → INITO and → AND_QO.REQ. QI is forwarded to SoftKey_UP_F1, TRIGGER_CLIENT, STATE_SUBSCRIBE and GreenWhiteBackground_AX; their QO outputs are ANDed by the AND_QO block (type iec61131::booleanOperators::AND_BOOL_4) into QO.

The result is a clean remote toggle operation: the user presses F1, device B toggles its internal flip-flop, and device A displays the new state.

Technical Features

- Composite subapplication type belonging to the MyLib::sys package.
- Uses isobus::UT::io::Softkey::Softkey_IE for softkey event detection.
- Uses iec61499::net::CLIENT_0 for the OPC UA remote method call.
- Uses adapter::net::AX_SUBSCRIBE_1 for adapter-based state subscription.
- Uses MyLib::sys::GreenWhiteBackground1_AX for green/white state visualization.
- Configuration is done through data inputs; apart from the optional INIT/INITO chain no external event wiring is necessary.
- No internal state machine is required.

- All internal blocks receive `QI` from the subapplication's `QI` input (default `TRUE`).
- The remote method call is a pure RPC trigger and does not require writing alternating values to a tag.
- The protocol and communication details are contained inside the composite subapplication, making it reusable across resources.

State Overview

The subapplication does not contain an explicit ECC state machine. Its behavior is event-driven and dataflow-oriented.

Conceptually, the following states can be identified:

State	Description
Waiting for F1 release	The softkey block is armed and waits for `SK_RELEASED`. The background displays the current remote state.
Issuing remote call	On `SK_RELEASED`, the event is forwarded to `CLIENT_0`, which sends the OPC UA method call to device B.
Monitoring and visualization	`STATE_SUBSCRIBE` reads the remote BOOL state and updates `GreenWhiteBackground_AX`.

No flip-flop state is stored locally on device A. The authoritative state remains on device B.

Application Scenarios

- **Remote operator panel**: Device A acts as an operator interface for device B. Pressing F1 triggers a remote action on device B.
- **RPC-based toggle**: Device B contains a flip-flop that toggles as side effect of an OPC UA method call. Device A only sends the trigger and does not need to simulate a value change.
- **Visual state feedback**: The green/white background gives the operator immediate feedback about the remote flip-flop state.
- **Reusable protocol encapsulation**: The complete OPC UA trigger and state-read logic is bundled in a library composite, so the resource on device A remains simple and maintainable.

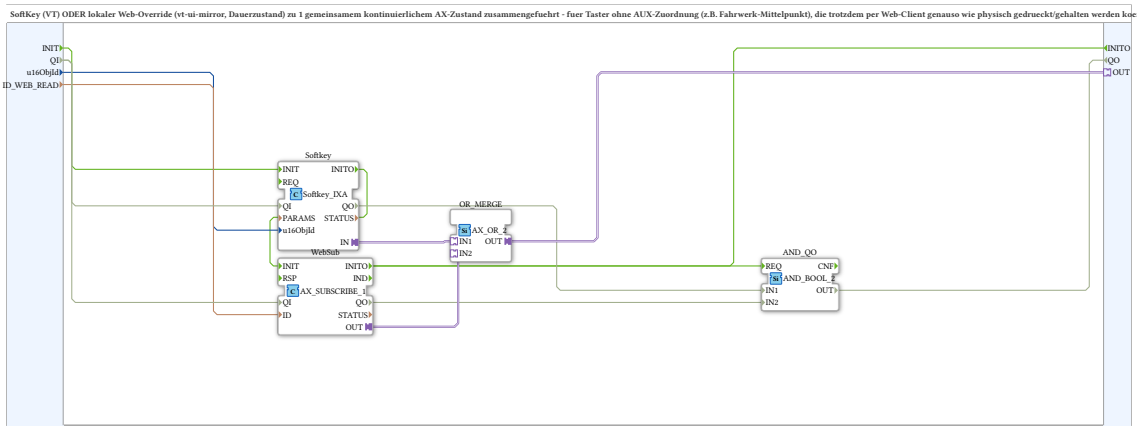
Comparison with Similar Blocks

Approach	Characteristics
Direct softkey to local output	Simple, but cannot trigger remote OPC UA methods.
Raw `CLIENT_0` in the resource	Requires manual event and data wiring in every resource. No integrated state visualization.
Value-change trigger with `AX_T_FF`	Requires an extra flip-flop or alternating write values on device A and can produce unnecessary network traffic.
This subapplication	Uses an explicit OPC UA `CALL_METHOD` trigger and an adapter-based state subscription. The complete protocol logic is encapsulated in the composite subapplication.

Conclusion

Softkey_Toggle_RPC_TO_Remote_BG_OPC provides a compact and reusable solution for remote OPC UA triggering with local visual feedback. By combining a softkey event source, a `CLIENT_0` RPC trigger, an adapter-based state subscription, and a green/white background display, it achieves a clean separation between HMI input, communication logic, and visualization. The design follows a "subapplication style" where the complexity of the remote protocol is hidden inside the composite, making it suitable for reusable library-based automation components.

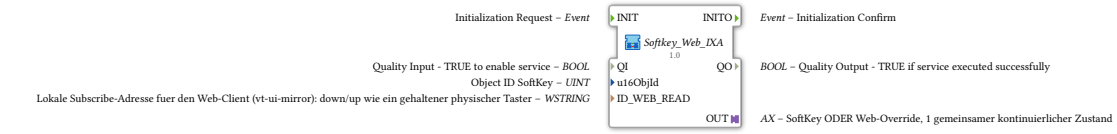
Softkey_Web_IXA



• — ○ • — ○ • — ○ •

Introduction

`Softkey_Web_IXA` merges a **SoftKey (VT)** OR a local **web override** (vt-ui-mirror, continuous state) into **one** common continuous `AX` state. It is intended for keys without an AUX assignment (e.g. undercarriage centre point) that should nevertheless be pressable or holdable via the web client just like physically. Analogous to `Softkey_Aux_IXA_TO_Remote_WRITE`, which additionally has AUX and remote write; here only the pure merge without AUX and without writing anywhere. The caller processes the merged state further itself (e.g. with `SoftKey_Kurz_Lang`).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the chain `Softkey` ⟶ `WebSub`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `WebSub.INITO`

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, service enable; passed on to `Softkey` and `WebSub`
- **u16ObjId** (`UINT`): object ID of the SoftKey
- **ID_WEB_READ** (`WSTRING`): local subscribe address for the web client (vt-ui-mirror): down/up like a held physical key

Data Outputs

- **QO** (`BOOL`): quality output, `AND` of the `QO` outputs of `Softkey` and `WebSub` (via `AND_QO`)

Adapters

- **OUT** (plug, `adapter::types::unidirectional::AX`): SoftKey OR web override as one common continuous state

Function Blocks (FBs) Used

- **`Softkey`** (`isobus::UT::io::Softkey::Softkey_IXA`): reads the SoftKey state
- **`WebSub`** (`adapter::net::AX_SUBSCRIBE_1`): subscribes to the web override
- **`OR_MERGE`** (`adapter::booleanOperators::AX_OR_2`): OR combination of both states
- **`AND_QO`** (`iec61131::booleanOperators::AND_BOOL_2`): combines `Softkey.QO` and `WebSub.QO` into the SubApp output `QO`

Functionality

- `u16ObjId` ⟶ `Softkey.u16ObjId`; `ID_WEB_READ` ⟶ `WebSub.ID`
- `Softkey.IN` ⟶ `OR_MERGE.IN1`; `WebSub.OUT` ⟶ `OR_MERGE.IN2`
- `OR_MERGE.OUT` ⟶ `OUT`
- Initialization: `INIT` ⟶ `Softkey.INIT`; `Softkey.INITO` ⟶ `WebSub.INIT`; `WebSub.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`
- Quality: `QI` ⟶ `Softkey.QI` and `WebSub.QI`; `Softkey.QO` ⟶ `AND_QO.IN1`, `WebSub.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`

Technical Details

- Continuous state (not an event): the web client can hold the key like a physical one
- No AUX, no remote write

Application Scenarios

- Keys without AUX assignment that should also be operable via the web client and whose state is processed further e.g. by `SoftKey_Kurz_Lang`

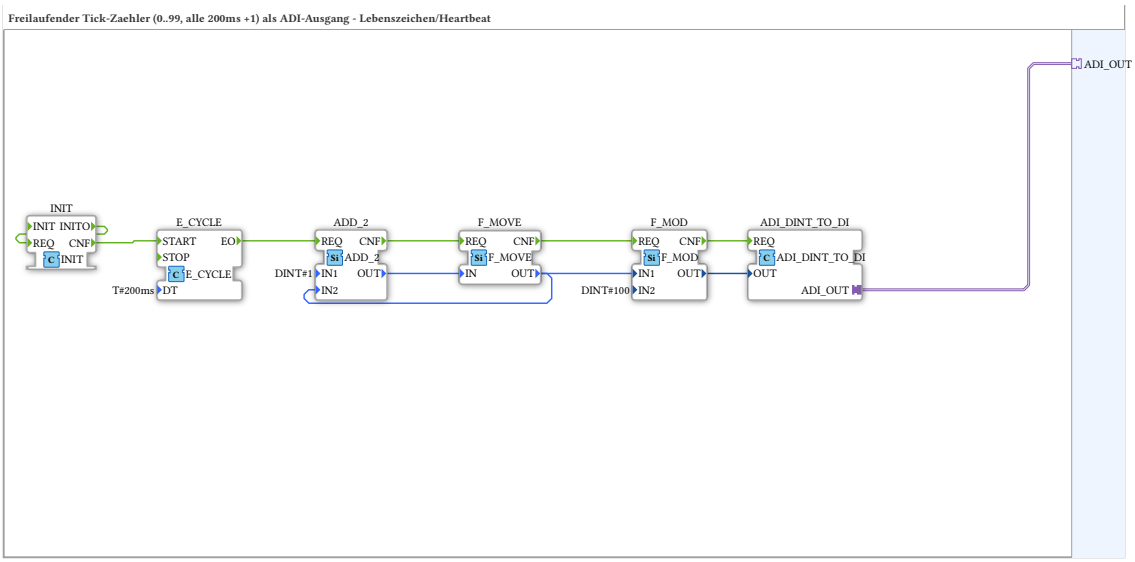
Summary

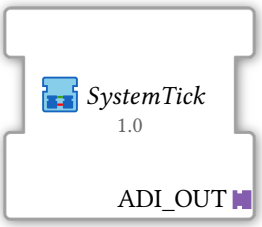
`Softkey_Web_IXA` ORs SoftKey and web override into one continuous `AX` state.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SystemTick





ADI – Aktueller Tick-Zaehlerstand (0..99), aendert sich alle 200ms

• – ○ • – ○ • – ○ •

Introduction

The **SystemTick** subapplication is a free-running tick counter designed to serve as a heartbeat indicator for industrial control systems. It independently increments a counter value every 200 milliseconds, cycling back to zero after reaching 100 (i.e., a full cycle lasts 20 seconds). The current counter value is provided via

an adapter output, making it easy to monitor the health of the system that hosts this subapplication. As long as the output value keeps changing, the system is alive — regardless of whether the value is displayed locally or distributed via OPC-UA to other modules.

Interface Structure

This subapplication provides a single adapter output and no direct event or data interfaces. All internal logic is encapsulated within the embedded function block network.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Name	Type	Description
`ADI_OUT`	`adapter::types::unidirectional::ADI`	Current tick counter value (0–99), updated every 200 ms.

Functionality

The subapplication implements a cyclic counter using the following internal function blocks:

- **INIT** – Performs initialization and starts the cycle generation.
- **E_CYCLE** – Generates a periodic event every 200 ms (configured via its `DT` parameter).
- **ADD_2** – Adds 1 to the current counter value on each cycle.
- **F_MOVE** – Stores the current counter value and forwards it to the arithmetic operations.
- **F_MOD** – Applies modulo 100 when returning the visible value to ensure the range 0–99.
- **ADI_DINT_TO_DI** – Converts the internal DINT counter value into the `ADI` adapter format for the `ADI_OUT` plug.

Working sequence:

1. The `INIT` function block starts `E_CYCLE` after initialization.
2. Each cycle, `E_CYCLE` triggers `ADD_2`, which increments the counter by 1.
3. The incremented value is stored in `F_MOVE` and also fed to `F_MOD` for modulo 100.
4. `F_MOD` outputs a value between 0 and 99 and passes it to `ADI_DINT_TO_DI`.
5. `ADI_DINT_TO_DI` packages the value and sends it via the `ADI_OUT` adapter output.

Technical Features

- **Periodicity:** 200 ms (configurable via the `DT` parameter of the `E_CYCLE` block).
- **Value range:** 0 to 99 (modulo 100).
- **Cycle duration:** 20 seconds for a complete wrap-around.
- **Output type:** Unidirectional `ADI` adapter, carrying one event and one 32-bit integer (DINT).
- **Initialization:** Automatic start via the `INIT` block; no external trigger required.
- **Encapsulation:** All logic is self-contained; no direct I/O or interface connections except the adapter output.

State Overview

The subapplication does not expose an explicit state machine to the outside. Internally, the counter value is held in the `F_MOVE`/`ADD_2` loop, which behaves as a simple infinite cycle:

1. **Startup:** `INIT` runs and starts `E_CYCLE`.
2. **Counting:** Each cycle increments the counter by 1.
3. **Wrap:** When the value reaches 100, `F_MOD` resets it to 0.

4. **Output:** The current value is continuously reflected on `ADI_OUT`.

Application Scenarios

- **Heartbeat monitoring:** Use the periodic updating counter as a "system alive" signal in distributed control systems.
- **Operational indication:** Display the tick value on a visualization (HMI) to show that the underlying logic is still executing.
- **OPC-UA distribution:** Feed the adapter output to an OPC-UA server (e.g., a gateway) to make the heartbeat available to remote clients.
- **Watchdog functionality:** Detect loss of signal when the output stops changing, indicating a malfunction or offline state.

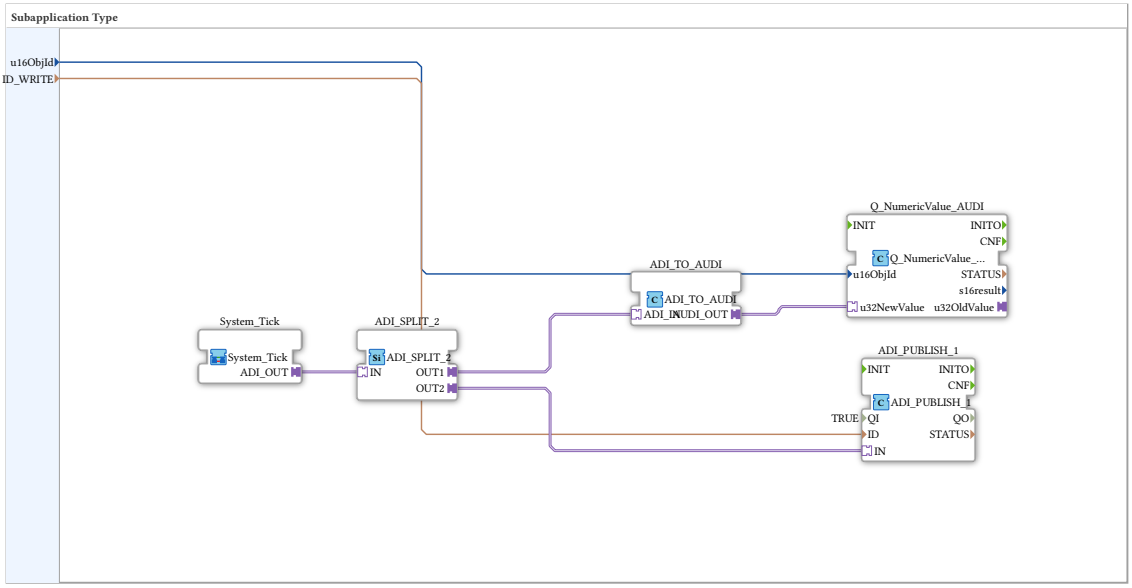
Comparison with Similar Blocks

- **SystemTick vs. simple E_CYCLE with counter:** Unlike a raw periodic event, SystemTick provides a standardized adapter output that encapsulates both an event and a value, making it easier to reuse and integrate with other adapters.
- **SystemTick vs. PLC timer blocks:** Traditional IEC 61131 timers require explicit reset and hold logic; SystemTick runs autonomously and includes modulo scaling.
- **SystemTick vs. heartbeat FBs with fixed periods:** The period is adjustable (via `DT`) and the value range can be easily modified by changing the modulo constant, offering flexibility.

Conclusion

The **SystemTick** subapplication is a compact, self-contained heartbeat generator well suited for monitoring the health of automation systems. Its simple interface (single adapter output) and clear internal structure make it easy to integrate into larger 4diac projects. The periodic, bounded counter provides a reliable liveliness signal that can be consumed locally or remotely, fulfilling a fundamental requirement in distributed control environments.

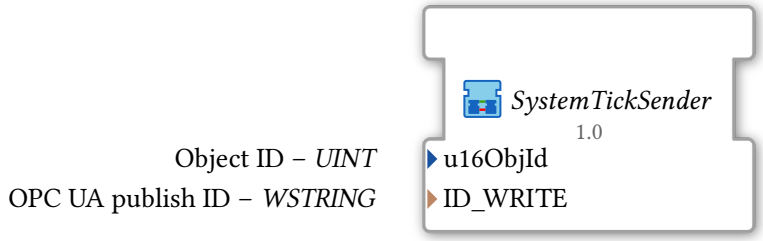
SystemTickSender



• — ○ • — ○ • — ○ •

Introduction

`SystemTickSender` provides a cyclically incrementing "heartbeat" for the VT and web client — a simple, immediately visible confirmation that the controller is running and processing events. The block is used in several training examples in this system, including `InputOutputTesterButton_DIDO_OPC_UA` and `InputOutputTesterButton_PWM_OPC_UA`.



Function Blocks (FBs) Used

Sub-blocks: SystemTickSender

- **Type**: SubAppType
- **Internal FBs used**:
  - **System_Tick** (SubApp, `MyLib::sys`): provides a cyclically incrementing value as an `ADI` adapter (DINT).
  - **ADI_SPLIT_2**: `adapter::events::unidirectional::ADI_SPLIT_2` — splits the counter adapter into two directions.
  - **ADI_TO_AUDI**: `adapter::conversion::unidirectional::ADI_TO_AUDI` — converts a DINT adapter to a UDINT adapter (numerically correct, see Numeric vs. bitwise: `ANY_INT`⟶`ANY_INT` is a safe cast).
  - **Q_NumericValue_AUDI**: `isobus::UT::Q::Q_NumericValue_AUDI` — writes the counter value into a VT number field.
  - **ADI_PUBLISH_1**: `adapter::net::ADI_PUBLISH_1` — OPC-UA publish of the raw DINT counter value to the web client.
- **Functionality**: The internally generated counter is used once for the VT display (via a UDINT intermediate step, since `Q_NumericValue_AUDI` expects that type) and once raw as DINT for OPC-UA publish.

Program Flow and Connections

1. `System_Tick.ADI_OUT` ⟶ `ADI_SPLIT_2.IN`.
2. `ADI_SPLIT_2.OUT1` ⟶ `ADI_TO_AUDI.ADI_IN` ⟶ `ADI_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue` (VT display).
3. `ADI_SPLIT_2.OUT2` ⟶ `ADI_PUBLISH_1.IN` (OPC-UA publish).
4. Parameters: `u16ObjId` ⟶ `Q_NumericValue_AUDI.u16ObjId`; `ID_WRITE` ⟶ `ADI_PUBLISH_1.ID`.

Technical Details

- **Two parallel uses of the same counter**: VT display (via `ADI_TO_AUDI`, since `Q_NumericValue_AUDI` expects a UDINT adapter) and OPC-UA publish (directly as DINT) run independently via `ADI_SPLIT_2`.
- **Numerically correct conversion**: `ADI_TO_AUDI` (DINT⟶UDINT) is an `ANY_INT`⟶`ANY_INT` cast, not a bit-reinterpretation trap like e.g. `AD_TO_AR`.

Application Scenarios

- "Heartbeat" display on the VT and web UI to show at a glance that the controller is active and hasn't crashed or frozen.

Summary

`SystemTickSender` is a simple but widely reused block across almost every training example in this system, distributing a cyclic counter value in parallel to a VT number field and an OPC-UA web client — a minimal but effective heartbeat for the controller.

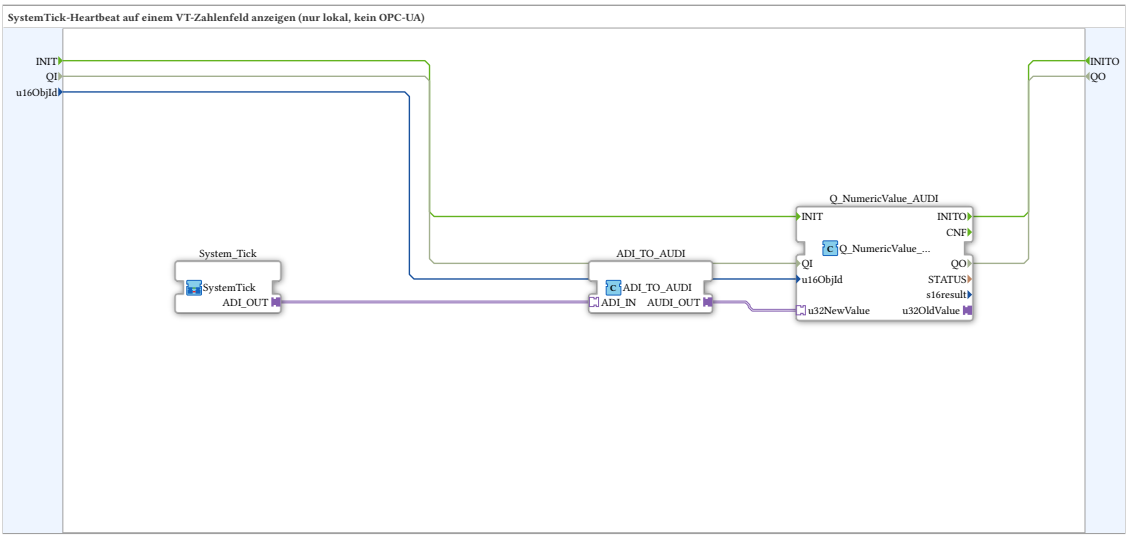
🛠️ Related Exercises

- InputOutputTesterButton_DIDO_OPC_UA
- InputOutputTesterButton_PWM_OPC_UA

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SystemTickSender_ISO



• — ○ • — ○ • — ○ •

Introduction

`SystemTickSender_ISO` displays the heartbeat locally on a VT numeric output field, without OPC UA. The running tick counter is written directly to a VT numeric field, so it's immediately clear on the screen whether the controller is still active (the value must change every 200 ms).



Function Blocks (FBs) Used

Sub-Blocks: SystemTickSender_ISO

- **Type**: SubAppType
- **Internal FBs Used**:
- **System_Tick** (SubApp, `MyLib::sys`): returns the counter reading as a `ADI` adapter (DINT).
- **ADI_TO_AUDI**: `adapter::conversion::unidirectional::ADI_TO_AUDI` — converts the DINT value to `AUDI` (UDINT) because `Q_NumericValue_AUDI` expects an unsigned value.
- **Q_NumericValue_AUDI**: `isobus::UT::Q::Q_NumericValue_AUDI` — VT command "Change numeric value" (Part 6 – F.22), writes the value to the numeric output field `u16ObjId`.
- **Functionality**: `System_Tick` returns the counter value, `ADI_TO_AUDI` converts it to the data type expected by `Q_NumericValue_AUDI`.

Program Flow and Connections

1. `u16ObjId` ⟶ `Q_NumericValue_AUDI.u16ObjId` (data connection, hidden).
2. `System_Tick.ADI_OUT` ⟶ `ADI_TO_AUDI.ADI_IN` ⟶ `ADI_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue`.
3. `INIT` ⟶ `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` ⟶ `INITO` (new event input `INIT` / event output `INITO`, both `EInit`).
4. `QI` (`BOOL`, default `TRUE`) ⟶ `Q_NumericValue_AUDI.QI`; `Q_NumericValue_AUDI.QO` ⟶ `QO` (`BOOL`, Quality Output - `TRUE` if the VT write succeeded).

Technical Features

- **Purely local, no OPC UA**: Unlike `SystemTickSender_ISO_OPC`, this module does not have a `ADI_PUBLISH_1` adapter.

Application Scenarios

- Modules with their own VT connection that only need to make the heartbeat visible locally, without remote monitoring.

Comparison with Similar Modules

For additional remote distribution, use `SystemTickSender_ISO_OPC` for a module without its own VT connection (OPC UA only), use `SystemTickSender_OPC`. Compared to `SystemTickSender` (VT + OPC UA, older example with `ADI_SPLIT_2`), this module is the pure VT-only variant without split/publish functionality.

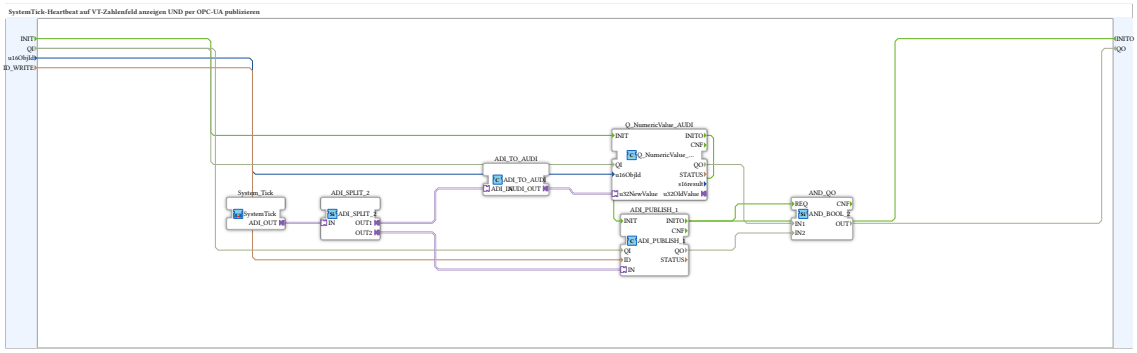
Summary

`SystemTickSender_ISO` displays the system tick heartbeat exclusively on a local VT numeric field—the most streamlined option for modules with their own VT connection without remote monitoring.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

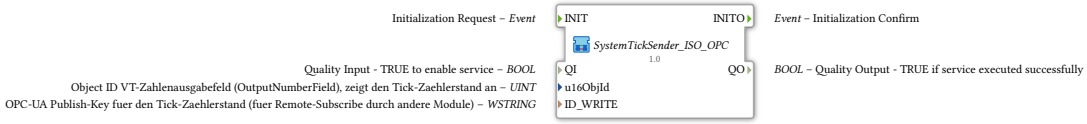
SystemTickSender_ISO_OPC



• – ○ • – ○ • – ○ •

Introduction

`SystemTickSender_ISO_OPC` combines `SystemTickSender_ISO` (display on a local VT number field) with `SystemTickSender_OPC` (remote publishing via OPC UA) so that the same status indicator is visible both on the local screen and can be monitored by other modules via remote subscribe.



Function Blocks (FBs) Used

Sub-Blocks: SystemTickSender_ISO_OPC

- **Type**: SubAppType
- **Internal FBs Used**:
- **System_Tick** (SubApp, `MyLib::sys`): Returns the counter reading as a `ADI` adapter (DINT).
- **ADI_SPLIT_2**: `adapter::events::unidirectional::ADI_SPLIT_2` — Fans `System_Tick.ADI_OUT` out to two destinations, as an adapter socket can only accept one source: one output feeds the VT display (via `ADI_TO_AUDI`), the other feeds `ADI_PUBLISH_1` for OPC UA publishing.
- **ADI_TO_AUDI**: `adapter::conversion::unidirectional::ADI_TO_AUDI` — Converts DINT to UDINT for VT display.
- **Q_NumericValue_AUDI**: `isobus::UT::Q::Q_NumericValue_AUDI` — displays the value locally on the VT (`u16ObjId`).

- **ADI_PUBLISH_1**: `adapter::net::ADI_PUBLISH_1` (`QI` comes from the input `QI`) — publishes the same counter value via OPC UA (`ID_WRITE`).
- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — combines `Q_NumericValue_AUDI.QO` and `ADI_PUBLISH_1.QO` into the output `QO`.
- **Functionality**: The counter value from `System_Tick` is branched via `ADI_SPLIT_2`: one branch goes to the VT display via `ADI_TO_AUDI`, as with `SystemTickSender_ISO`, the other goes directly to `ADI_PUBLISH_1` for OPC UA publishing — as with `SystemTickSender_OPC`.

Program Flow and Connections

1. `System_Tick.ADI_OUT` ⟶ `ADI_SPLIT_2.IN`.
2. `ADI_SPLIT_2.OUT1` ⟶ `ADI_TO_AUDI.ADI_IN` ⟶ `ADI_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_AUDI.u32NewValue` (VT display).
3. `ADI_SPLIT_2.OUT2` ⟶ `ADI_PUBLISH_1.IN` (OPC UA publish).
4. Parameters: `u16ObjId` ⟶ `Q_NumericValue_AUDI.u16ObjId`; `ID_WRITE` ⟶ `ADI_PUBLISH_1.ID`.
5. Initialization chain (event input `INIT` / event output `INITO`, both `EInit`): `INIT` ⟶ `Q_NumericValue_AUDI.INIT`; `Q_NumericValue_AUDI.INITO` ⟶ `ADI_PUBLISH_1.INIT`; `ADI_PUBLISH_1.INITO` ⟶ `INITO` and `AND_QO.REQ`.
6. Quality: `QI` (`BOOL`, default `TRUE`) ⟶ `Q_NumericValue_AUDI.QI` and `ADI_PUBLISH_1.QI`; `Q_NumericValue_AUDI.QO` ⟶ `AND_QO.IN1`; `ADI_PUBLISH_1.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO` (`BOOL`, Quality Output - `TRUE` if the VT write and the OPC UA publish succeeded).

Technical Features

- **One counter, two uses**: `ADI_SPLIT_2` allows the parallel use of the same `System_Tick` counter for VT display and OPC UA publishing without duplicating the counter.

Application Scenarios

- Modules with their own VT connection whose heartbeat should also be monitored by other modules via remote subscribe.

Comparison with similar function blocks

If only one of the two outputs is needed, the simpler `SystemTickSender_ISO` (VT only) or `SystemTickSender_OPC` (OPC UA only) should be used. This pattern (Split ⟶ VT Display + OPC UA Publish) is structurally equivalent to the older `SystemTickSender`.

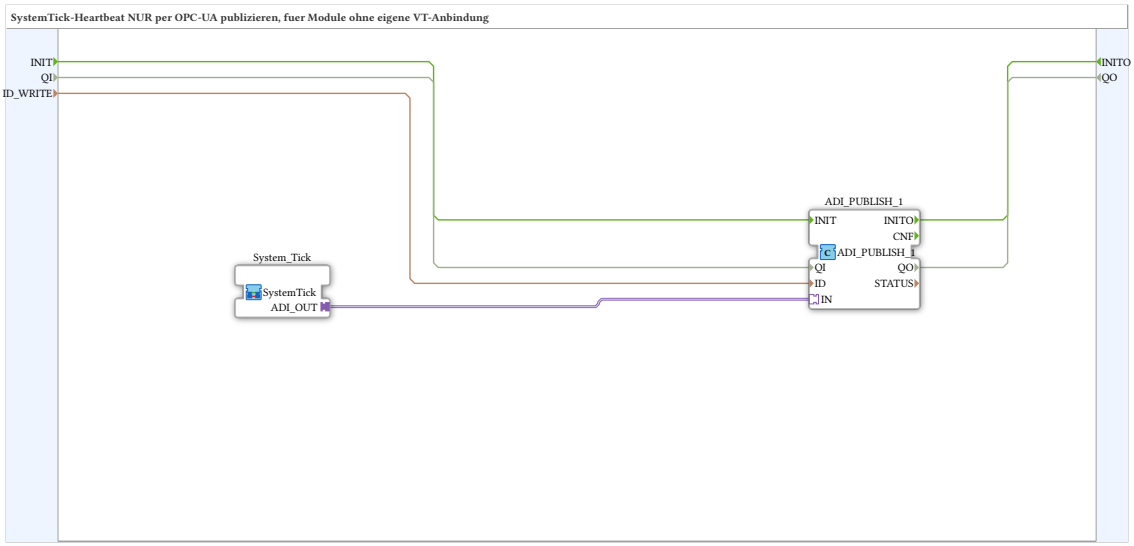
Summary

`SystemTickSender_ISO_OPC` displays the SystemTick Heartbeat simultaneously locally on the VT and remotely via OPC UA by combining the two leaner individual variants into a single component.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

SystemTickSender_OPC



• – ○ • – ○ • – ○ •

Introduction

`SystemTickSender_OPC` is the most streamlined SystemTick variant for modules without their own VT connection: The running tick counter from `System_Tick` is published directly via OPC UA, without a local VT display. Another module with VT can subscribe to and display this value via remote subscribe to monitor the module's activity.



Function Blocks (FBs) Used

Sub-Blocks: SystemTickSender_OPC

- **Type**: SubAppType
- **Internal FBs Used**:
- **System_Tick** (SubApp, `MyLib::sys`): returns the counter reading as a `ADI` adapter (DINT).
- **ADI_PUBLISH_1**: `adapter::net::ADI_PUBLISH_1` — publishes the counter reading directly via OPC UA (`ID_WRITE`).
- **Functionality**: The counter reading from `System_Tick` is passed directly to `ADI_PUBLISH_1` without conversion.

Program Flow and Connections

1. `ID_WRITE` ⟶ `ADI_PUBLISH_1.ID` (Data connection, hidden).
2. `System_Tick.ADI_OUT` ⟶ `ADI_PUBLISH_1.IN`.
3. Initialization: event input `INIT` ⟶ `ADI_PUBLISH_1.INIT`; `ADI_PUBLISH_1.INITO` ⟶ event output `INITO`.
4. Quality: data input `QI` (`BOOL`, default `TRUE`) ⟶ `ADI_PUBLISH_1.QI`; `ADI_PUBLISH_1.QO` ⟶ data output `QO`.

Technical Features

- **No type conversion required**: Since no VT display is involved, the `ADI_TO_AUDI` conversion is omitted —the raw DINT value is published directly.

Application Scenarios

- Modules without their own VT connection (e.g., pure I/O/field modules) whose liveness is to be monitored by another module with VT.

Comparison with Similar Modules

If a local VT display is also required, use `SystemTickSender_ISO_OPC` (VT + OPC UA) or `SystemTickSender_ISO` (VT only).

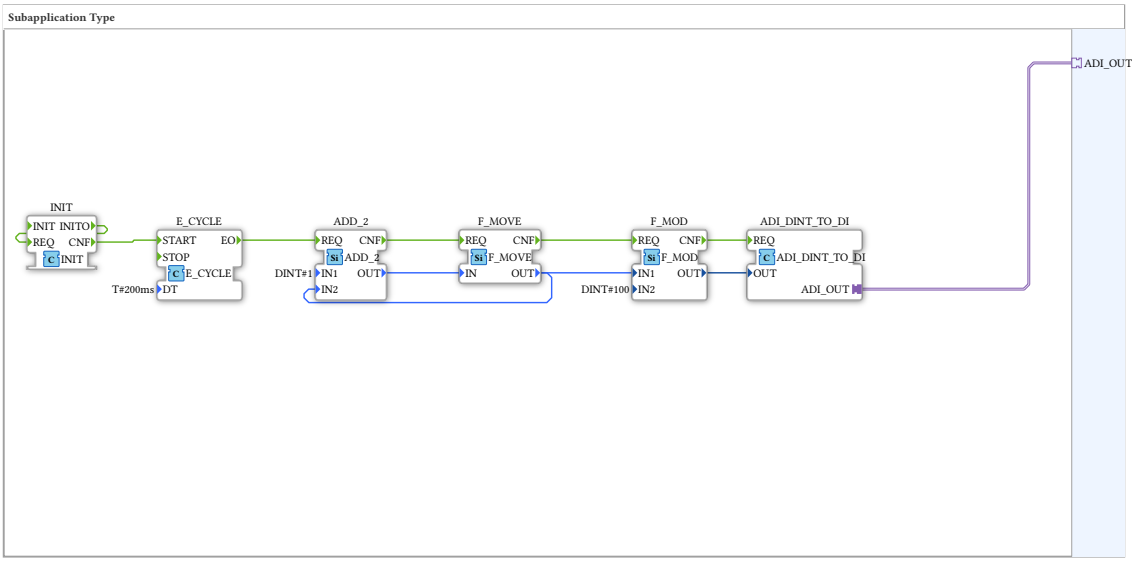
Summary

`SystemTickSender_OPC` publishes the SystemTick heartbeat exclusively via OPC UA — the minimal option for modules without their own VT connection.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

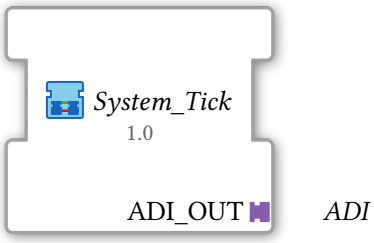
System_Tick



• – ○ • – ○ • – ○ •

Introduction

`System_Tick` generates a continuous, cyclically incrementing DINT counter (200 ms tick, value range 0..99, wrapping via modulo) and exposes it through an `ADI` adapter (DINT) for other blocks to consume — a simple heartbeat counter.



Function Blocks Used

Sub-blocks: System_Tick

- **Type**: SubAppType
- **Internal FBs used**:
  - **INIT**: `iec61131::booleanOperators::INIT` — fires once at startup (`INITO` fed back to `REQ`) and starts the cycle timer.

  - **E_CYCLE**: `iec61499::events::E_CYCLE` — cyclic event, `DT=T#200ms`, started via `INIT.CNF`.
  - **ADD_2**: `iec61131::arithmetic::ADD_2` — adds `IN1=DINT#1` to the current counter value (`IN2`, fed back from `F_MOVE.OUT`).
  - **F_MOVE**: `iec61131::selection::F_MOVE` (`DataType=DINT`) — holds/passes the current counter value, acting as the feedback loop's intermediate storage.
  - **F_MOD**: `iec61131::arithmetic::F_MOD` — modulo operation with `IN2=DINT#100`, limiting the value range to 0..99.
  - **ADI_DINT_TO_DI**: `adapter::conversion::unidirectional::ADI_DINT_TO_DI` — converts the DINT value onto the `ADI_OUT` adapter for external use.
- **Operation**: `INIT` starts the `E_CYCLE` timer; each cycle increments the counter via `ADD_2`/`F_MOVE` by 1, `F_MOD` keeps it within 0..99, and `ADI_DINT_TO_DI` places the result onto the output adapter.

Program Flow and Connections

1. `INIT.INITO` ⟶ `INIT.REQ` (self-trigger at startup); `INIT.CNF` ⟶ `E_CYCLE.START`.
2. `E_CYCLE.EO` ⟶ `ADD_2.REQ` (every 200 ms).
3. `F_MOVE.OUT` ⟶ `ADD_2.IN2` (feedback of the current value); `ADD_2.OUT` ⟶ `F_MOVE.IN`.
4. `ADD_2.CNF` ⟶ `F_MOVE.REQ` ⟶ `F_MOD.REQ` (event chain); `F_MOVE.OUT` ⟶ `F_MOD.IN1`.
5. `F_MOD.CNF` ⟶ `ADI_DINT_TO_DI.REQ`; `F_MOD.OUT` ⟶ `ADI_DINT_TO_DI.OUT`.
6. `ADI_DINT_TO_DI.ADI_OUT` (adapter) ⟶ `ADI_OUT` (SubApp interface).

Technical Details

- The counter runs purely internally within the block (no external data inputs) — it is an autonomous clock that triggers itself via `INIT`.
- The modulo bound `DINT#100` yields a value range of 0..99, not 1..100 as one might assume at first glance — the value implicitly starts at 0 after `INIT`, and `ADD_2` counts up by 1 each cycle from there.

Application Scenarios

- Heartbeat/liveness signal for diagnostic purposes, e.g. to show on the VT or via OPC-UA that the controller is actively cycling.
- Simple substitute for a one-second tick (5 cycles of 200 ms ≈ 1 s) for test purposes, without configuring a dedicated timer block.

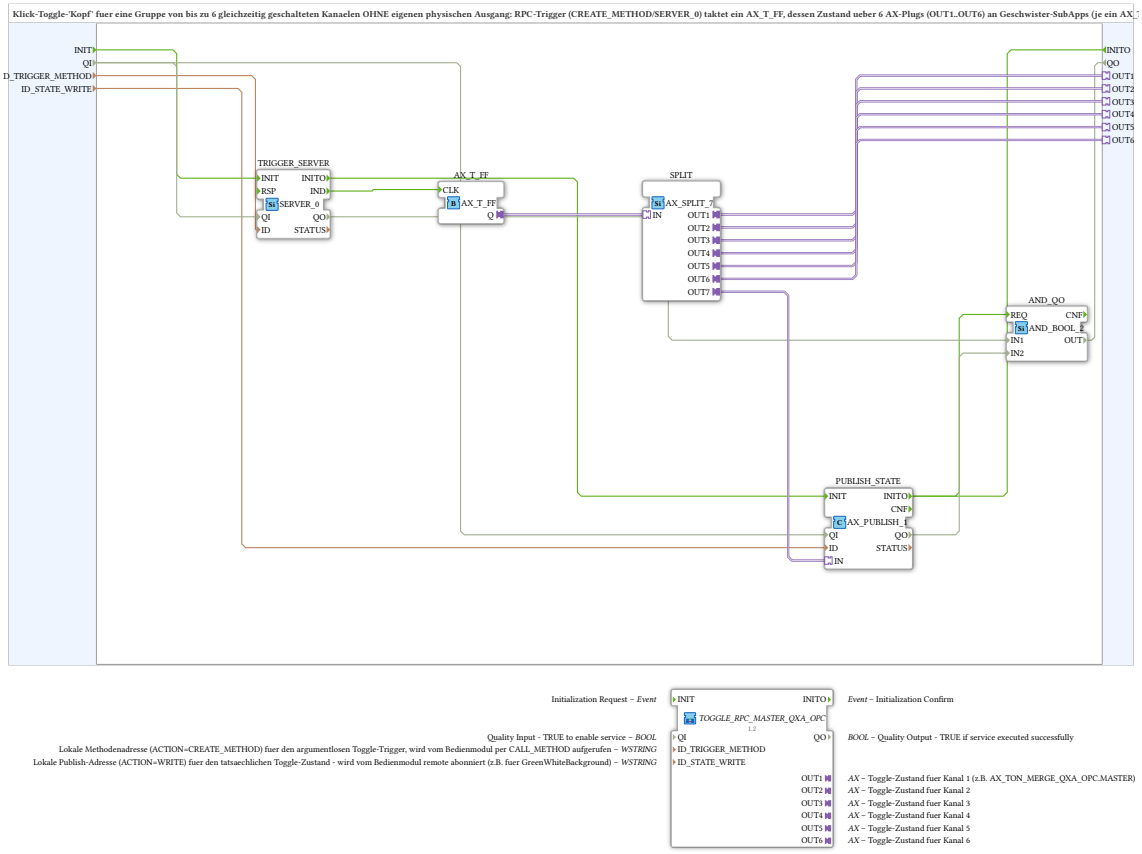
Summary

Autonomous, self-triggering 200 ms counter with modulo limiting, exposed via a DINT adapter — a simple heartbeat block.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

TOGGLE_RPC_MASTER_QXA_OPC



• — ○ • — ○ • — ○ •

Introduction

The `TOGGLE_RPC_MASTER_QXA_OPC` is a subapplication that implements a remote-controlled toggle "head" for a group of up to six simultaneously switched channels without having its own physical output. It is intended for OPC UA based operator panels: a remote client triggers a toggle operation by calling a local OPC UA method, and the resulting state is distributed through six adapter plugs to sibling subapplications. The same state is also published locally so that an operator panel can display it, for example as a background colour.

This subapplication is designed for multi-channel use cases where one softkey should toggle several outputs at the same time, but each physical channel may still have its own delayed start behaviour.

Interface Structure

The subapplication exposes the data input `QI`, two WSTRING data inputs, the data output `QO` and six unidirectional AX adapter plugs. Apart from `INIT`/`INITO` it has no event inputs or outputs and no sockets.

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirmation (from `PUBLISH_STATE.INITO`)

Data Inputs

Name	Type	Description
`QI`	BOOL	Quality Input, TRUE enables the service (default `TRUE`); distributed to `TRIGGER_SERVER.QI` and `PUBLISH_STATE.QI`.

ID_TRIGGER_METHOD	WSTRING	Local OPC UA method address (ACTION=CREATE_METHOD) used as the argumentless toggle trigger. It is called remotely by the operator panel via CALL_METHOD.
ID_STATE_WRITE	WSTRING	Local OPC UA publish address (ACTION=WRITE) for the actual toggle state. It can be subscribed by remote operator panels, e.g. for a GreenWhite background indication.

Data Outputs

- **QO** (BOOL): Quality Output, AND of TRIGGER_SERVER.QO and PUBLISH_STATE.QO (AND_QO, iec61131::booleanOperators::AND_BOOL_2)

Adapters

Name	Type	Direction / Kind	Description
OUT1	adapter::types::unidirectional::AX	Plug	Toggle state for channel 1, typically connected to an AX_TON_MERGE_QXA_OPC.MASTER.
OUT2	adapter::types::unidirectional::AX	Plug	Toggle state for channel 2.
OUT3	adapter::types::unidirectional::AX	Plug	Toggle state for channel 3.
OUT4	adapter::types::unidirectional::AX	Plug	Toggle state for channel 4.
OUT5	adapter::types::unidirectional::AX	Plug	Toggle state for channel 5.
OUT6	adapter::types::unidirectional::AX	Plug	Toggle state for channel 6.

Unused OUTn plugs can simply remain unconnected.

Functionality

The subapplication provides a click-toggle behaviour:

1. The internal OPC UA server FB TRIGGER_SERVER receives an incoming method call or event at its IND input.
2. The same event clocks the internal AX_T_FF flip-flop.
3. The flip-flop output Q toggles between FALSE and TRUE on every trigger.
4. The state is forwarded to the AX_SPLIT_7 block, which distributes it to seven branches.
5. Six of the branches are connected to the OUT1 .. OUT6 adapter plugs and can be wired to sibling subapplications.
6. The seventh branch is connected to PUBLISH_STATE, which publishes the state under the configured OPC UA address ID_STATE_WRITE.
7. TRIGGER_SERVER.IND is directly connected to TRIGGER_SERVER.RSP, ensuring that the OPC UA server response is sent immediately and that the server thread is not blocked.
8. Initialization and quality: INIT → TRIGGER_SERVER.INIT; TRIGGER_SERVER.INITO → PUBLISH_STATE.INIT; PUBLISH_STATE.INITO → INITO (and → AND_QO.REQ). QI → TRIGGER_SERVER.QI and PUBLISH_STATE.QI; TRIGGER_SERVER.QO → AND_QO.IN1, PUBLISH_STATE.QO → AND_QO.IN2, AND_QO.OUT → QO.

This allows the toggle state to be used both locally in the automation network and remotely for HMI display purposes.

Technical Features

- Uses iec61499::net::SERVER_0 as the OPC UA method and event server.
- Uses adapter::events::unidirectional::AX_T_FF as the central toggle flip-flop.
- Uses adapter::events::unidirectional::AX_SPLIT_7 to distribute one toggle state to seven outputs.
- Uses adapter::net::AX_PUBLISH_1 to publish the current toggle state to OPC UA.
- Uses iec61131::booleanOperators::AND_BOOL_2 (AND_QO) to combine the QO outputs of the server and the publisher.
- TRIGGER_SERVER.IND is wired directly to TRIGGER_SERVER.RSP to avoid blocking the OPC UA server thread.

- Supports up to six 1:1 adapter connections to sibling subapplications.
- The data input connections are marked as not visible in the internal network to keep the design compact.
- Licensed under the Eclipse Public License 2.0.

State Overview

The central state is the boolean output `Q` of the internal `AX_T_FF` block. It toggles on every rising edge of the trigger event from `TRIGGER_SERVER.IND`.

State	Meaning
`FALSE`	Toggle state is off. All connected `OUTn` adapter values and the published OPC UA state are `FALSE`.
`TRUE`	Toggle state is on. All connected `OUTn` adapter values and the published OPC UA state are `TRUE`.
Transition	A trigger event is received, `Q` is inverted, and the new state is distributed to all branches.

There is no additional internal state machine; the behaviour is that of a simple T flip-flop.

Application Scenarios

- **Lighting groups**: A bank of six spotlights can be switched with a single softkey while each channel starts with its own delay to avoid capacitive inrush current peaks.
- **Multi-channel remote control**: An operator panel calls one OPC UA method to toggle a group of drives, valves, or light channels.
- **HMI background indication**: The published state is subscribed by the operator panel and used for colour feedback, e.g. Green/White background.
- **Distributed systems**: The `OUTn` plugs can be connected to sibling subapplications such as `AX_TON_MERGE_QXA_OPC`, one per physical channel, allowing shared toggle control with individual channel timing.

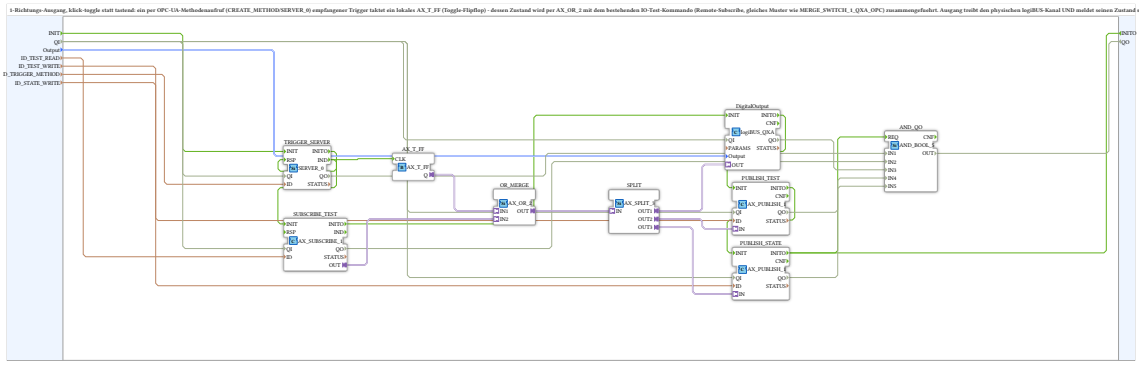
Comparison with Similar Blocks

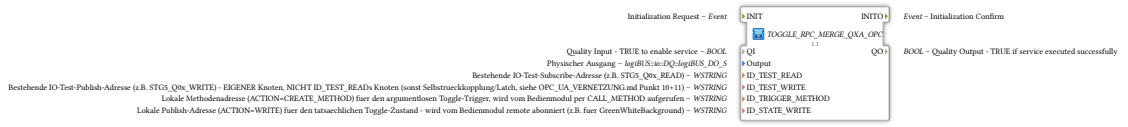
- `TOGGLE_RPC_MASTER_QXA_OPC` has no physical output of its own. Instead, it provides six adapter plugs to distribute the toggle state to multiple sibling subapplications.
- `TOGGLE_RPC_MERGE_QXA_OPC` is the single-output variant. It includes the physical output path and is recommended when only one output is required.
- Other softkey/RPC toggle blocks may support remote background toggling, but they do not provide the same multi-channel fan-out through six separate plugs.

Conclusion

`TOGGLE_RPC_MASTER_QXA_OPC` is a flexible solution for remote click-toggle control of up to six channels via OPC UA. It combines a method-triggered flip-flop, local adapter-based fan-out, and OPC UA state publication. The direct `IND` to `RSP` connection keeps the OPC UA server responsive, and unused outputs can be left unconnected, making the subapplication suitable for both small and medium-sized multi-channel HMI-controlled applications.

TOGGLE_RPC_MERGE_QXA_OPC





• — ○ • — ○ • — ○ •

Introduction

TOGGLE_RPC_MERGE_QXA_OPC is a composite subapplication designed for single-action (click-toggle) output control. It combines an RPC-based trigger with an existing IO-test command to drive a physical binary output. The RPC trigger is received via OPC-UA method calls (CREATE_METHOD/SERVER_0) and toggles a local flip-flop. The resulting state is merged with the standard IO-test signal and then split to provide both a physical output and two separate feedback channels – one for IO diagnostics and one for the actual functional state (e.g., for background colour indication on a softkey). The design ensures full consistency with the project's other similar blocks, avoiding self-latch by using distinct publish nodes for commands and feedback.

Interface Structure

The subapplication exposes data inputs plus the quality/initialization ports (`QI`, `QO`, `INIT`, `INITO`); no adapters are present at the subapp level. All other event and adapter connections are internal.

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirm

Data Inputs

Name	Type	Description
`QI`	`BOOL`	Quality input - TRUE enables the service (initial value: `TRUE`)
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	Physical output value to be applied (initialized to `logiBUS_DO::Invalid`). Internal connection to the physical output FB.
`ID_TEST_READ`	`WSTRING`	Subscription address for the existing IO-test command (e.g., `STG5_Q0x_READ`).
`ID_TEST_WRITE`	`WSTRING`	Publish address for the IO-test feedback. Must be a separate node from `ID_TEST_READ` to avoid self-feedback/latch.
`ID_TRIGGER_METHOD`	`WSTRING`	Local method address (ACTION=CREATE_METHOD) for the argumentless toggle trigger, invoked by the operation panel via CALL_METHOD.
`ID_STATE_WRITE`	`WSTRING`	Local publish address (ACTION=WRITE) for the actual toggle state – subscribed remotely by the operation panel (e.g., for GreenWhiteBackground).

Data Outputs

- **QO** (`BOOL`): quality output, AND combination (`AND_QO`, `AND_BOOL_5`) of the `QO` outputs of `TRIGGER_SERVER`, `SUBSCRIBE_TEST`, `DigitalOutput`, `PUBLISH_TEST` and `PUBLISH_STATE`. The actual results are still delivered through the internal OPC-UA publish blocks.

Adapters

None at the subapp interface. Internally, the following adapters/FBs are used:

- `TRIGGER_SERVER` (iec61499::net::SERVER_0) – receives the RPC method call.
- `AX_T_FF` (adapter::events::unidirectional::AX_T_FF) – toggle flip-flop.
- `SUBSCRIBE_TEST` (adapter::net::AX_SUBSCRIBE_1) – subscribes to the IO-test command.
- `OR_MERGE` (adapter::booleanOperators::AX_OR_2) – merges flip-flop state and IO-test signal.
- `SPLIT` (adapter::events::unidirectional::AX_SPLIT_3) – distributes the merged result.
- `DigitalOutput` (logiBUS::io::DQ::logiBUS_QXA) – physical output.
- `PUBLISH_TEST` (adapter::net::AX_PUBLISH_1) – publishes feedback to IO-test node.
- `PUBLISH_STATE` (adapter::net::AX_PUBLISH_1) – publishes the functional state to the dedicated feedback node.

Functionality

The subapplication implements a **click-toggle** behaviour:

- A single press of a softkey (or any RPC client) issues an **argument-free method call** on the `ID_TRIGGER_METHOD` node.
- `TRIGGER_SERVER` receives this call and its `IND` event toggles the `AX_T_FF` flip-flop (`CLK` input).
- The flip-flop output (`Q`) is combined via `OR_MERGE` with the signal coming from the IO-test subscription (`SUBSCRIBE_TEST`). This way the IO-test command can still force the output regardless of the toggle state.
- The combined result is passed to `SPLIT`, which feeds three destinations:
  1. `DigitalOutput` – drives the physical logiBUS channel.
  2. `PUBLISH_TEST` – writes to the IO-test feedback node (address from `ID_TEST_WRITE`).
  3. `PUBLISH_STATE` – writes to the functional feedback node (address from `ID_STATE_WRITE`).

Initialization and quality: `INIT` → `TRIGGER_SERVER.INIT` → `SUBSCRIBE_TEST.INIT` → `DigitalOutput.INIT` → `PUBLISH_TEST.INIT` → `PUBLISH_STATE.INIT`; `PUBLISH_STATE.INITO` → `INITO` and → `AND_QO.REQ`. `QI` is distributed to all five blocks (no hard-wired `QI` parameter any more); their `QO` outputs feed `AND_QO.IN1` to `AND_QO.IN5` in this order, `AND_QO.OUT` → `QO`.

The use of **separate write nodes** for feedback avoids the self-latch problem that would occur if the same OPC-UA node were used for both command and feedback. The toggle state is **not actively written back** to the panel via a client; instead, it is locally published and remotely subscribed – a pattern consistent with all other functions in the project.

Technical Features

- **RPC Trigger without Polling**: The trigger is delivered as a method call (`CREATE_METHOD/SERVER_0`), eliminating any value-change tricks or polling overhead.
- **Toggle Flip-Flop**: The `AX_T_FF` changes its output on each rising edge of the input event, providing a bistable state.
- **Merging with IO-Test**: The `AX_OR_2` ensures that the IO-test command can override the toggle state, keeping the diagnostic capability intact.
- **Deterministic Feedback**: Two separate publish adapters (`PUBLISH_TEST` and `PUBLISH_STATE`) guarantee that IO diagnostics and functional state are reported independently.
- **No Active Client-Write**: The feedback is published locally and subscribed remotely, consistent with the project's distributed architecture (e.g., softkey background updates).
- **Unique Publish Nodes**: `ID_TEST_WRITE` and `ID_STATE_WRITE` must be distinct from each other and from the corresponding command nodes to prevent feedback loops.

State Overview

The internal `AX_T_FF` flip-flop maintains a single Boolean state (`Q`). This state toggles on every `CLK` pulse, which is generated by the RPC method call. The flip-flop is **not** reset by any external signal; it only changes when a new trigger arrives. The output of the OR gate follows either the flip-flop state or the IO-test signal, whichever is currently TRUE. Since the IO-test signal is typically FALSE during normal operation, the output normally mirrors the flip-flop state.

Application Scenarios

- **Flash Light Control**: A single press turns the light ON, another press turns it OFF – no need to hold the button.
- **Any Toggle-Type Output**: Suitable for functions where a momentary switch action must produce a sustained state (e.g., latching relays, mode toggles).

- **Integration with IO-Testing**: The block preserves the ability to force the output via the IO-test system without interfering with the toggle logic.
- **Softkey Interaction**: The functional state is made available to the operation panel via a dedicated publish node, enabling dynamic visual feedback (e.g., green/white background) as seen in other project functions.

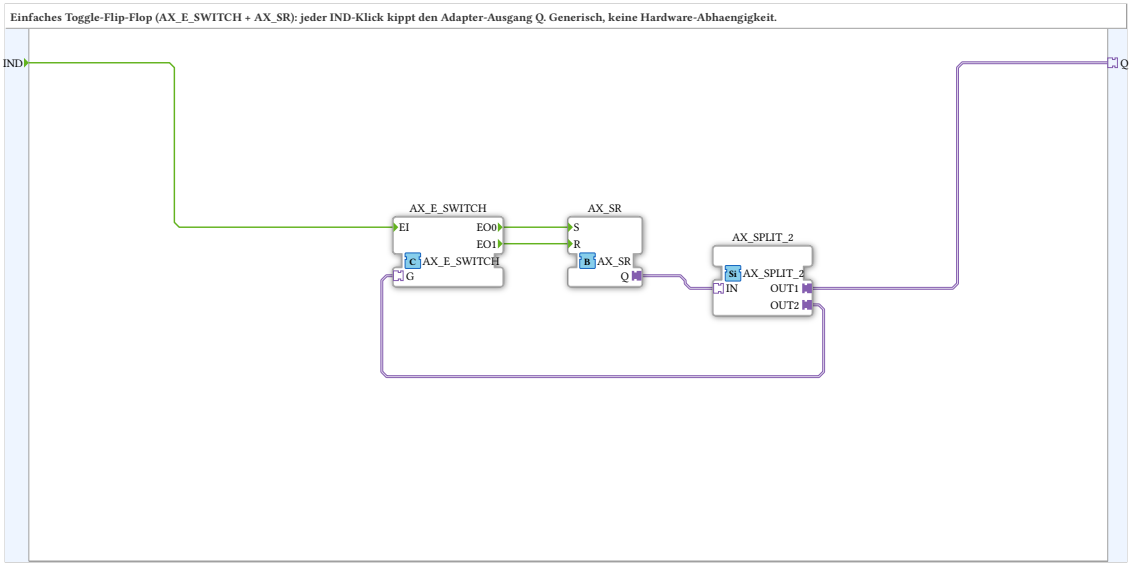
Comparison with Similar Blocks

- **MERGE_SWITCH_1_QXA_OPC**: Uses a level-based subscription (value change) as the command source instead of an RPC trigger. TOGGLE_RPC_MERGE_QXA_OPC additionally toggles the state via a flip-flop, making it a true click-toggle rather than a momentary switch.
- **ILOCK_SWITCH_2_QXA_OPC**: Also follows the pattern of separate feedback nodes, but its command source is different. TOGGLE_RPC_MERGE_QXA_OPC adds the toggle functionality while maintaining the same robust feedback architecture.
- **Toggle_RPC_FROM_Remote_QXA_OPC (original exercise)**: The original actively wrote back to the panel via a client. This block instead publishes locally and subscribes remotely, which is more consistent with the project's distributed design and avoids unnecessary OPC-UA client writes.

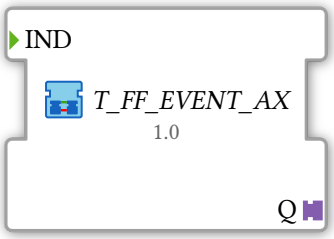
Conclusion

TOGGLE_RPC_MERGE_QXA_OPC provides a clean, reusable solution for click-toggle outputs in a distributed automation environment. By combining an RPC-based trigger with a local flip-flop, merging it with the IO-test command, and publishing separate feedback channels, it achieves reliable operation without feedback loops. Its design is fully aligned with the project's established patterns, making it ideal for functions that require a single press to toggle a binary state.

T_FF_EVENT_AX



Event Input (Clock / Click) – *Event*



AX – Adapter Output Q

• – ◦ • – ◦ • – ◦ •

Introduction

T_FF_EVENT_AX is a generic toggle flip-flop (T-FF) subapplication that implements a toggling latch behavior based on an event-driven input and an adapter-based output. Every incoming `IND` event toggles

the logical state of the output adapter Q. The subapplication is built entirely from standard event adapter blocks (AX_E_SWITCH, AX_SR, AX_SPLIT_2) and is completely hardware-independent.

Interface Structure

Event Inputs

Name	Type	Comment
IND	Event	Event input (Clock / Click) – triggers the toggling

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Name	Type	Direction	Comment
Q	adapter::types::unidirectional::AX	Output (Plug)	Adapter output holding the current toggled state

Functionality

The subapplication implements a classic toggle (T) flip-flop behavior entirely through event and adapter connections:

1. An IND event triggers the internal AX_E_SWITCH block.
2. AX_E_SWITCH evaluates its gating input G and activates exactly one of its two event outputs:
   - EO0 if the gate is inactive (state FALSE),
   - EO1 if the gate is active (state TRUE).
3. If EO0 fires, the internal AX_SR is **Set** (Q becomes TRUE).
   If EO1 fires, the internal AX_SR is **Reset** (Q becomes FALSE).
4. The current state from AX_SR.Q is passed to AX_SPLIT_2.
5. AX_SPLIT_2.OUT1 drives the external adapter output Q; AX_SPLIT_2.OUT2 is fed back to the gating input G of AX_E_SWITCH.
6. This feedback loop ensures that on the next IND event the gate has switched state, so the opposite branch is triggered – producing the alternating toggle effect.

Technical Features

- **Pure event-driven design**: no cyclic data scanning, no level-triggered polling.
- **Generic adapter interface**: no hardware-specific dependencies, fully reusable in distributed IEC 61499 systems.
- Built from three reusable adapter-pattern blocks:
  - AX_E_SWITCH – gated event router with two complementary outputs.
  - AX_SR – Set/Reset flip-flop that stores the state.
  - AX_SPLIT_2 – fan-out adapter splitting the state signal to the external plug and the internal feedback path.
- **Internal feedback loop** provides the memory/toggling effect without additional logic.
- Implemented as a subapplication, encapsulating the logic for easy reuse and readability.

State Overview

State	Description
Q = FALSE	Internal AX_SR is Reset. On the next IND event, EO0 fires and Q becomes TRUE.

Q = TRUE	Internal AX_SR is Set. On the next IND event, EO1 fires and Q becomes FALSE.

The system alternates between these two states on every IND event, regardless of the duration or timing of the input pulse.

Application Scenarios

- **Push-button toggles**: e.g., start/stop of a motor, light ON/OFF control in HMI panels.
- **Mode switching**: toggling between two operating modes in a control application.
- **Edge-triggered state toggling**: event-driven state changes in distributed automation systems.
- **Generic event-based logic**: wherever a single event should invert a Boolean/state output, without requiring a level-based data input.

Comparison with Similar Blocks

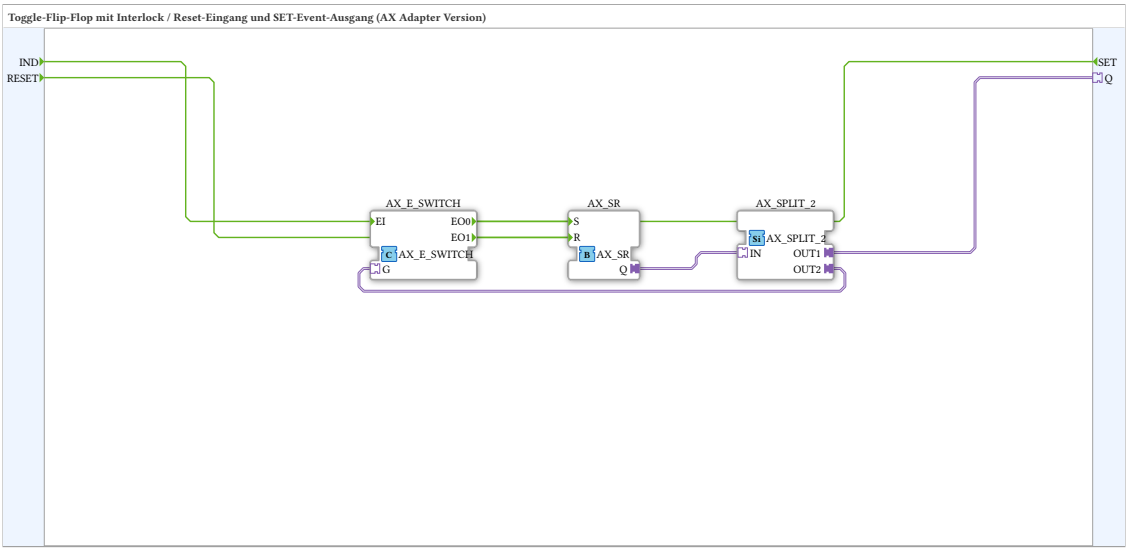
Block	Behavior
AX_SR	Level-triggered Set/Reset – the output depends on which input (S or R) was activated last, not on toggling.
D flip-flop	Copies the data input on a clock edge, does not toggle.
Hardware T-FF	Toggles on a clock edge – similar behavior, but uses binary level inputs/outputs.
T_FF_EVENT_AX	Event-driven toggle flip-flop with an AX adapter output – toggles on every IND event, hardware-independent.

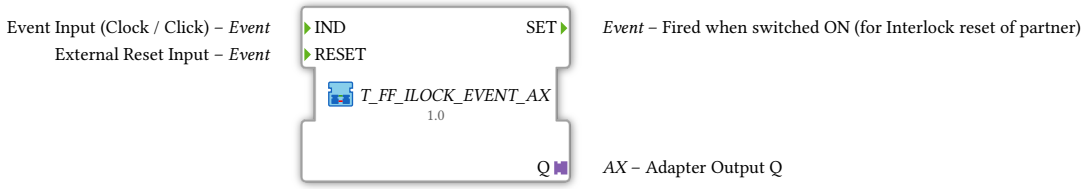
The main distinction of T_FF_EVENT_AX is its **event-driven nature** and **adapter-type interface**, making it ideal for IEC 61499 event-based processing rather than cyclic scanning.

Conclusion

T_FF_EVENT_AX is a compact, reusable, and generic subapplication implementing a toggle flip-flop with a purely event-driven design. Its adapter-based output and internal feedback loop make it a flexible building block for IEC 61499 applications where state toggling on a single event is required. Since it relies only on standard event adapters and no hardware-specific resources, it can be deployed across different platforms and runtime environments without modification.

T_FF_ILOCK_EVENT_AX





• – ◦ • – ◦ • – ◦ •

Introduction

T_FF_ILOCK_EVENT_AX is a subapplication-based function block that implements a **Toggle Flip-Flop with Interlock** functionality. It provides a clock/toggle input (`IND`), an unconditional reset input (`RESET`), a state output via an AX adapter (`Q`), and a dedicated `SET` event output that is used to coordinate interlocking with a partner block.

This component is realized entirely with event-driven adapter connections (unidirectional AX type), making it suitable for modular and distributed control applications where event-based data exchange between function blocks is preferred over cyclic data transmission. The block is primarily designed for scenarios where **mutual exclusion** between two devices must be guaranteed — for example, two actuators that may never be active at the same time.

Interface Structure

Event Inputs

Name	Description
`IND`	Event input acting as clock / toggle trigger. Each event toggles the state.
`RESET`	External reset input. Forces the output `Q` to inactive (OFF) unconditionally.

Event Outputs

Name	Description
`SET`	Fired when the block switches from OFF to ON. Used to reset the interlock partner.

Data Inputs

No data inputs are used. The block is purely event-driven.

Data Outputs

No data outputs are used. The current state is provided through the `Q` adapter plug.

Adapters

Name	Type	Description
`Q`	`adapter::types::unidirectional::AX`	Output adapter carrying the current state (ON / OFF) of the flip-flop.

Functionality

The subapplication contains three internal function blocks that are connected to realize the toggle-with-interlock behavior:

- **`AX_E_SWITCH`** — routes the incoming `IND` event depending on the gate signal `G`.
- **`AX_SR`** — a Set/Reset flip-flop that stores the actual state.
- **`AX_SPLIT_2`** — duplicates the state signal for the external output and the feedback path.

Internal event flow:

1. When an event arrives at `IND`, it is forwarded to the `EI` input of `AX_E_SWITCH`.
2. The `AX_E_SWITCH` block evaluates the gate adapter `G`, which receives the current state `Q` via the feedback loop:

- **If `Q` is inactive (OFF):** the event is routed to output `EO0`.
  - `EO0` triggers the **Set** input of `AX_SR`, driving the state to ON.
  - Simultaneously, the **`SET`** event output is fired, signalling the interlock partner that this block has been activated.
- **If `Q` is active (ON):** the event is routed to output `EO1`.
  - `EO1` triggers the **Reset** input of `AX_SR`, driving the state back to OFF.

3. The `AX_SR` block maintains the current state; its adapter output `Q` is fed into `AX_SPLIT_2`.
4. `AX_SPLIT_2` duplicates the state:
   - **`OUT1`** is connected to the external `Q` adapter plug of the subapplication.
   - **`OUT2`** is fed back to the gate `G` of `AX_E_SWITCH`, closing the control loop.

The external `RESET` event is connected directly to the **Reset** input of the `AX_SR` flip-flop, forcing the state to OFF at any time — regardless of the current state.

Technical Features

- **Pure event-driven design:** No data inputs or outputs; all state exchange uses AX adapter types.
- **Unidirectional AX adapters:** All internal and external adapter connections are of the unidirectional AX type, providing a lightweight and defined data flow direction.
- **State-dependent event routing:** The `AX_E_SWITCH` uses the feedback of the output state to decide whether an `IND` event turns the flip-flop ON or OFF.
- **Interlock support:** A dedicated `SET` event output is fired on every OFF→ON transition, allowing a partner block to be reset. This ensures that two mutually exclusive devices cannot be activated simultaneously.
- **Unconditional reset:** The `RESET` event bypasses any state evaluation and forces the output to OFF.
- **Low resource footprint:** The implementation uses only three internal function blocks and requires no internal variables or state machines from the user's perspective.

State Overview

The following table summarizes the behavior of the flip-flop:

Current State (`Q`)	Input Event	Next State (`Q`)	`SET` Output fired?
OFF	`IND`	ON	**Yes**
ON	`IND`	OFF	No
OFF or ON	`RESET`	OFF	No

This state diagram shows a classic toggle operation with the added interlock pulse on the rising edge (OFF→ON).

Application Scenarios

- **Interlocked actuators:** Two motors, valves, or conveyors sharing a common output station where only one may run at a time. Pressing the start button on one unit activates it and automatically resets the other via the `SET` event.
- **Mutual exclusion for control panels:** Two push-buttons for alternative operating modes; the active mode is toggled and the partner mode is locked out.
- **Alternating control with safety interlock:** In production lines where two stations work alternately, this block ensures that a handover of the active state occurs cleanly and without overlap.
- **Event-driven distributed systems:** Because the block uses AX adapters, it can be embedded in systems where separate applications communicate via adapter sockets and plugs.

Comparison with Similar Blocks

Block / Approach	Difference
Standard T_FF (toggle)	Provides only a simple toggle output without interlock coordination or reset event capability.
SR flip-flop	Is level/event driven by Set and Reset inputs but does not toggle on a single clock and cannot signal an interlock partner.

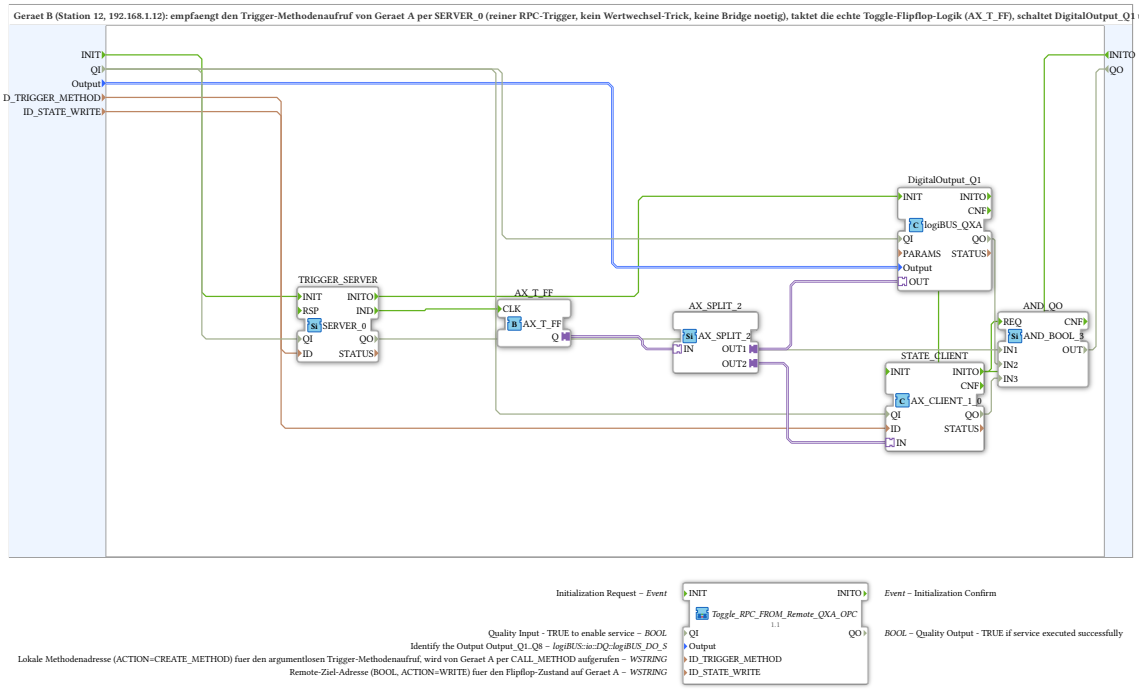
E_SWITCH based latch	A latch can hold a state but lacks the automatic toggle behavior and the SET synchronisation output.
Counter / step chain	Counters provide numeric step sequences but are not intended for simple two-state interlocking.

The key advantage of T_FF_ILOCK_EVENT_AX is the combination of toggle functionality, unconditional reset, and the interlock SET output in a single compact event-driven component.

Conclusion

T_FF_ILOCK_EVENT_AX is a compact, event-driven subapplication that implements a toggle flip-flop with interlock capabilities. Its internal architecture uses adapter-based event switching and a set/reset flip-flop with feedback, resulting in a reliable mutual-exclusion mechanism for control applications. The dedicated SET event output and the reset input make it especially suitable for systems where two or more participants must never be active at the same time. Because it is implemented entirely with AX adapters and events, it integrates cleanly into modern distributed control platforms such as Eclipse 4diac.

Toggle_RPC_FROM_Remote_QXA_OPC



• — ○ • — ○ • — ○ •

Introduction

Toggle_RPC_FROM_Remote_QXA_OPC is a composite SubApplication designed for a distributed OPC UA control scenario. It runs on Device B and waits for a pure RPC trigger method call coming from Device A. Each received trigger toggles a real flip-flop logic block, updates a local digital output, and actively writes the new state back to Device A. The communication protocol is encapsulated in the composite itself, so the device resource does not need any protocol-specific wiring.

Interface Structure

The SubApplication exposes the initialization events INIT/INITO, the status ports QI/QO and no adapter plugs/sockets. All other external information enters through three data inputs.

Event Inputs

- INIT (EInit): Initialization Request

Event Outputs

- INITO (EInit): Initialization Confirm

Data Inputs

Name	Type	Initial Value	Description
QI	BOOL	TRUE	Quality Input - TRUE enables the service.
Output	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Selects the digital output channel, e.g. Q1..Q8, that shall be toggled.
ID_TRIGGER_METHOD	WSTRING		Local method address for the argumentless RPC trigger. Device A calls this method using CALL_METHOD.
ID_STATE_WRITE	WSTRING		Remote address on Device A where the new flip-flop state is written back as a BOOL.

Data Outputs

- QO (BOOL): Quality Output - TRUE if the service executed successfully (AND of TRIGGER_SERVER.QO, DigitalOutput_Q1.QO and STATE_CLIENT.QO).

Adapters

No adapter sockets or plugs are exposed on the SubApplication boundary. Internally, adapter connections are used to forward the toggle event and state to the digital output and to the write-back client.

Functionality

The SubApplication implements a remote toggle request/response cycle:

1. Device A invokes the configured OPC UA method.
2. TRIGGER_SERVER, an instance of iec61499::net::SERVER_0, receives the request and raises its IND event.
3. The IND event directly clocks the toggle flip-flop AX_T_FF.
4. The same IND event is immediately connected back to TRIGGER_SERVER.RSP, releasing the OPC UA server thread and avoiding blocking delays.
5. The flip-flop output Q is fed into AX_SPLIT_2, which duplicates the adapter event/data stream.
6. The first split output drives DigitalOutput_Q1, a logiBUS::io::DQ::logiBUS_QXA instance, to set or clear the selected physical digital output.
7. The second split output feeds STATE_CLIENT, an adapter::net::AX_CLIENT_1_0 instance, which writes the new state back to Device A using the address given by ID_STATE_WRITE.
8. Initialization and status: INIT → TRIGGER_SERVER.INIT, TRIGGER_SERVER.INITO → DigitalOutput_Q1.INIT, DigitalOutput_Q1.INITO → STATE_CLIENT.INIT, STATE_CLIENT.INITO → INITO and → AND_QO.REQ. QI is forwarded to TRIGGER_SERVER, DigitalOutput_Q1 and STATE_CLIENT; their QO outputs go to AND_QO.IN1..IN3 (block AND_QO, type iec61131::booleanOperators::AND_BOOL_3), and AND_QO.OUT → QO.

The result is a complete remote toggle operation: Device A triggers, Device B toggles, Device B updates its output, and Device A receives the new state.

Technical Features

- Pure RPC trigger mechanism using SERVER_0; no value-change trick or intermediate bridge is required.
- TRIGGER_SERVER.IND is wired directly to TRIGGER_SERVER.RSP, preventing the OPC UA server thread from being held and avoiding project-wide communication delays.
- Real toggle logic is implemented with the AX_T_FF flip-flop adapter.
- AX_SPLIT_2 distributes the same adapter event/data stream to two consumers.

- Digital output handling is done with `logiBUS_QXA`, supporting channel selection via the `Output` data input.
- Active state write-back is performed by `AX_CLIENT_1_0`.
- The whole protocol stack lives inside the `MyLib::sys` composite, keeping the device resource clean.

State Overview

The SubApplication does not contain an explicit ECC state machine. Its stateful behavior is fully encapsulated in the internal `AX_T_FF` flip-flop:

- On each `CLK` event, the internal state toggles.
- The current state is available on the `Q` adapter output.
- The toggled state is then propagated to the digital output and to the write-back client.

Because the toggle logic is a true flip-flop, every remote method call changes the output state alternately between true and false.

Application Scenarios

This SubApplication is useful in distributed automation systems where one controller must remotely toggle an output on another controller and keep both devices synchronized. A typical scenario is:

- Device A acts as the master or HMI/SCADA gateway.
- Device B is a remote station, e.g. Station 12 at IP `192.168.1.12`.
- Device A sends an OPC UA method call to Device B.
- Device B toggles its local digital output.
- Device B writes the new output state back to Device A for consistent visualization or further logic.

This pattern is especially valuable when no continuous cyclic data exchange is required and a simple event-driven RPC is preferred.

Comparison with Similar Blocks

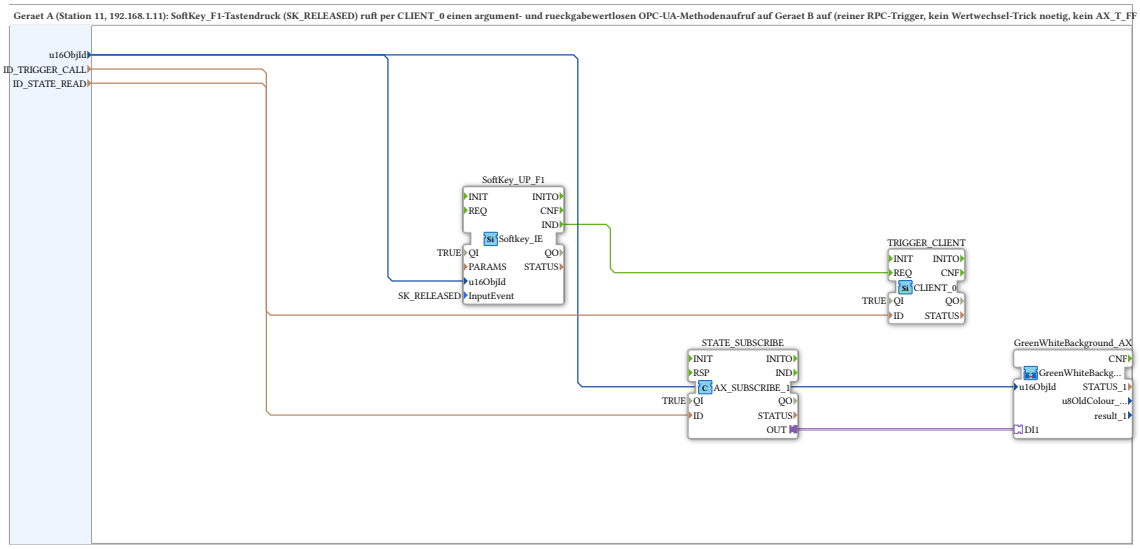
Compared to simpler or alternative implementations:

- **Plain `SERVER_0` without `RSP` wiring**: Can block the OPC UA server thread and delay unrelated OPC UA requests. This SubApplication avoids that by directly connecting `IND` to `RSP`.
- **Value-change trick**: Many implementations use a data value change instead of a real method call. This SubApplication uses a real RPC method trigger, which is cleaner and avoids unintended triggers.
- **Bridge-based communication**: Some architectures require a separate bridge block in the device resource. This SubApplication encapsulates the bridge/protocol logic inside the composite, so the resource remains simple and reusable.
- **FB with internal ECC**: A basic function block would implement the toggle in an ECC; this composite uses a dedicated `AX_T_FF` adapter, making the logic explicit and reusable.

Conclusion

`Toggle_RPC_FROM_Remote_QXA_OPC` provides a robust, self-contained way to implement remote toggling over OPC UA. It combines a responsive RPC server, true flip-flop behavior, digital output control, and active state write-back in one composite SubApplication. The direct `IND` to `RSP` wiring ensures that the OPC UA stack remains responsive, while the clean separation of protocol and resource logic makes the SubApplication easy to reuse and maintain.

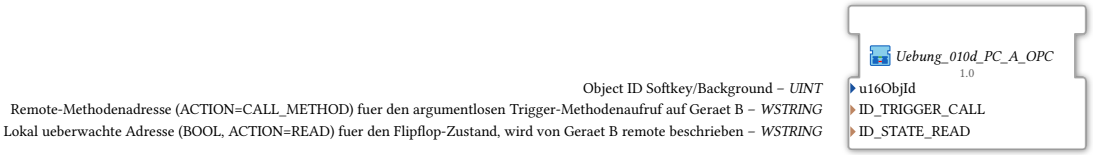
Uebung_010d_PC_A_OPC



• – ○ • – ○ • – ○ •

Introduction

`Uebung_010d_PC_A_OPC` is the device-A side (station 11, 192.168.1.11) of the PC-to-PC OPC-UA variant of exercise 010d (toggle flip-flop via softkey): a softkey press calls an argument- and return-value-less OPC-UA method on device B via `CLIENT_0` (a pure RPC trigger, no value-change trick needed, no toggle logic on this device). `GreenWhiteBackground1_AX` shows the flip-flop state locally monitored from device B. "SUB style": the protocol lives in the `MyLib::sys` composite, not in the device's resource - counterpart: `Uebung_010d_PC_B_OPC`.



Function blocks used

- **SoftKey_UP_F1** (`isobus::UT::io::Softkey::Softkey_IE`): physical softkey (F1), `InputEvent=SK_RELEASED`.
- **TRIGGER_CLIENT** (`iec61499::net::CLIENT_0`): calls the argument-less remote method under `ID_TRIGGER_CALL`.
- **STATE_SUBSCRIBE** (`adapter::net::AX_SUBSCRIBE_1`): subscribes to the flip-flop state written by device B, under `ID_STATE_READ`.
- **GreenWhiteBackground_AX** (SubApp, type `MyLib::sys::GreenWhiteBackground1_AX`): displays the subscribed state at the softkey.

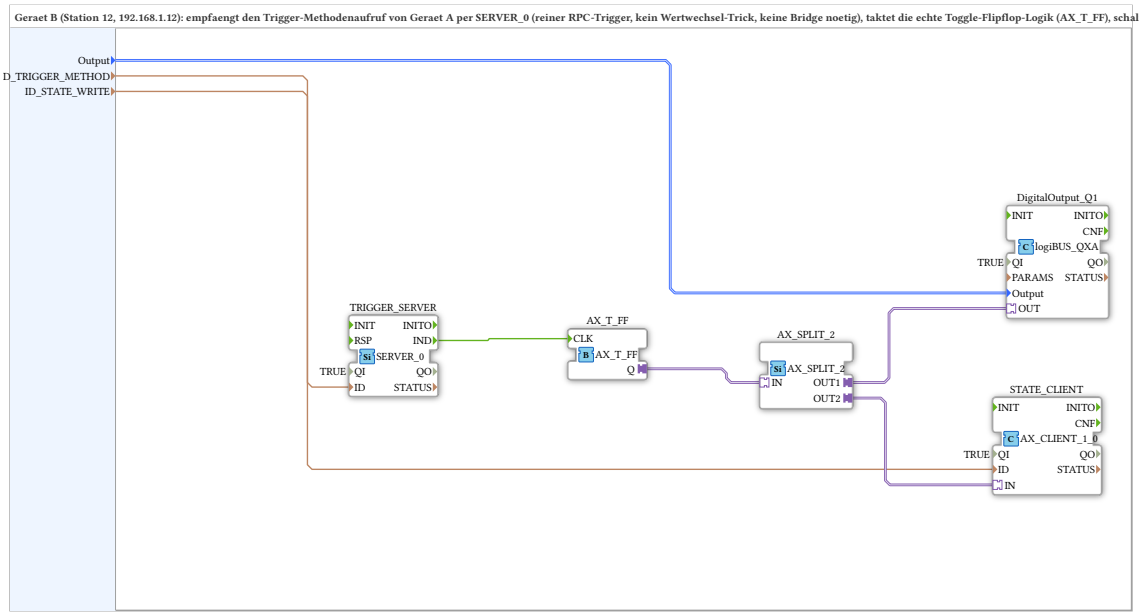
Summary

Device-A side of a PC-to-PC toggle trigger: the softkey calls a remote method, the actual flip-flop state is reported back from device B.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

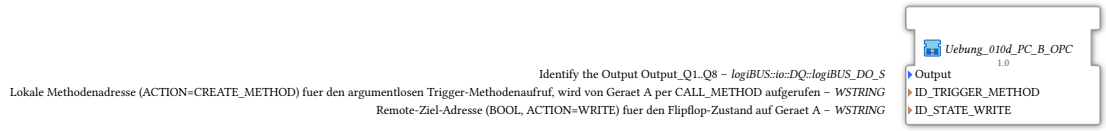
Uebung_010d_PC_B_OPC



• – ○ • – ○ • – ○ •

Introduction

`Uebung_010d_PC_B_OPC` is the device-B side (station 12, 192.168.1.12) of the PC-to-PC OPC-UA variant of exercise 010d: receives the trigger method call from device A via `SERVER_0` (a pure RPC trigger, no value-change trick, no bridge needed), clocks the actual toggle flip-flop logic (`AX_T_FF`), drives `DigitalOutput_Q1`, and actively writes the new state back to device A via `AX_CLIENT_1_0`. Counterpart: `Uebung_010d_PC_A_OPC`.



Function blocks used

- **TRIGGER_SERVER** (`iec61499::net::SERVER_0`): receives the method call under `ID_TRIGGER_METHOD`.
- **AX_T_FF** (`adapter::events::unidirectional::AX_T_FF`): toggle flip-flop, clocked by `TRIGGER_SERVER.IND`.
- **AX_SPLIT_2** (`adapter::events::unidirectional::AX_SPLIT_2`): splits the flip-flop state to the physical output + the echo.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physical output (`Output` selects `Output_Q1..Q8`).
- **STATE_CLIENT** (`adapter::net::AX_CLIENT_1_0`): actively writes the new state to device A under `ID_STATE_WRITE`.

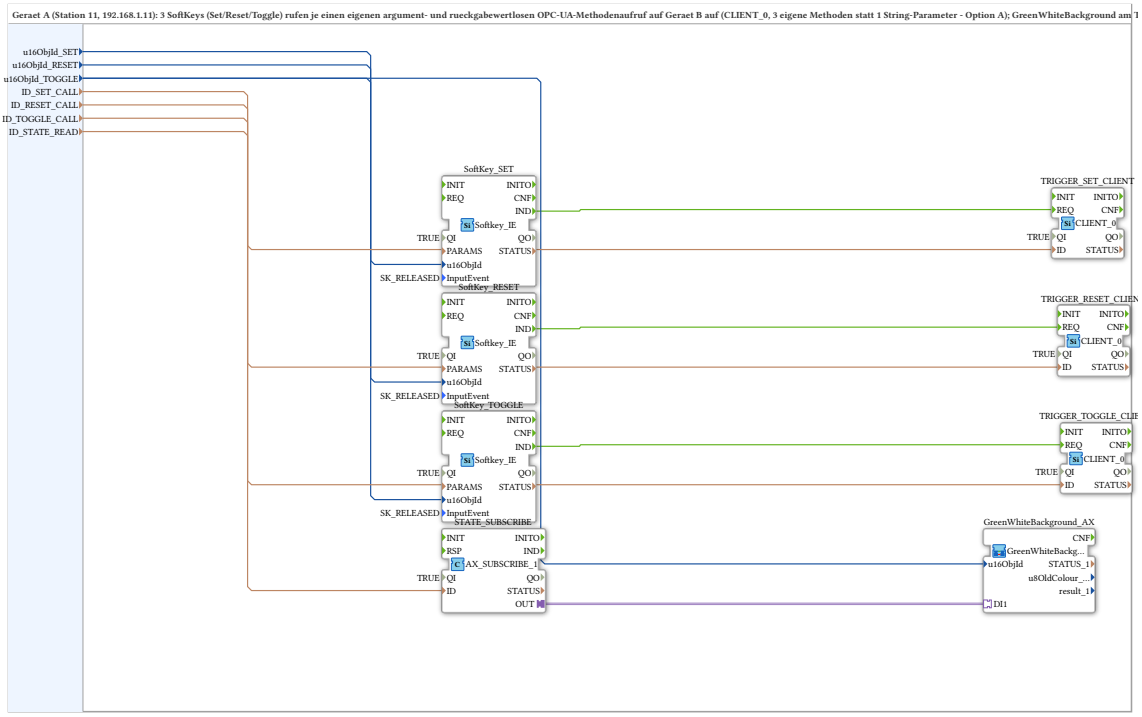
Summary

Device-B side: receives the trigger via RPC, holds the actual toggle logic, and actively reports the state back to device A.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

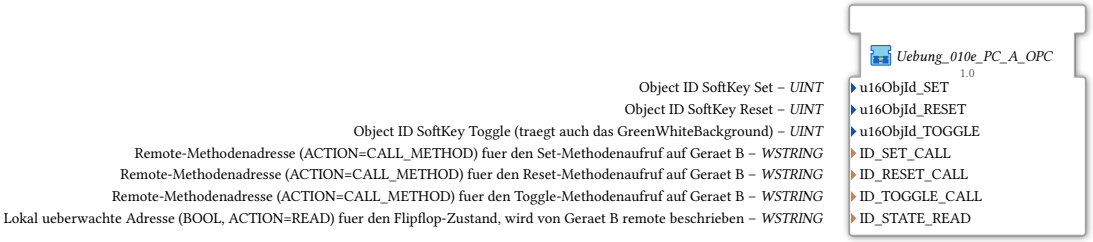
Uebung_010e_PC_A_OPC



• – ○ • – ○ • – ○ •

Introduction

`Uebung_010e_PC_A_OPC` is the device-A side (station 11, 192.168.1.11) of the PC-to-PC OPC-UA variant of exercise 010e (SR+toggle flip-flop via 3 softkeys): 3 softkeys (set/reset/toggle) each call their own argument- and return-value-less OPC-UA method on device B (`CLIENT_0`, 3 distinct methods instead of 1 string parameter - option A). `GreenWhiteBackground1_AX` on the toggle softkey shows the flip-flop state locally monitored from device B. Counterpart: `Uebung_010e_PC_B_OPC`.



Function blocks used

- **SoftKey_SET / SoftKey_RESET / SoftKey_TOGGLE** (`isobus::UT::io::Softkey::Softkey_IE`): the 3 physical softkeys.
- **TRIGGER_SET_CLIENT / TRIGGER_RESET_CLIENT / TRIGGER_TOGGLE_CLIENT** (`iec61499::net::CLIENT_0`): each its own remote method (`ID_SET_CALL`/`ID_RESET_CALL`/`ID_TOGGLE_CALL`).
- **STATE_SUBSCRIBE** (`adapter::net::AX_SUBSCRIBE_1`): subscribes to the flip-flop state under `ID_STATE_READ`.
- **GreenWhiteBackground_AX** (SubApp, type `MyLib::sys::GreenWhiteBackground1_AX`): displays the state at the toggle softkey.

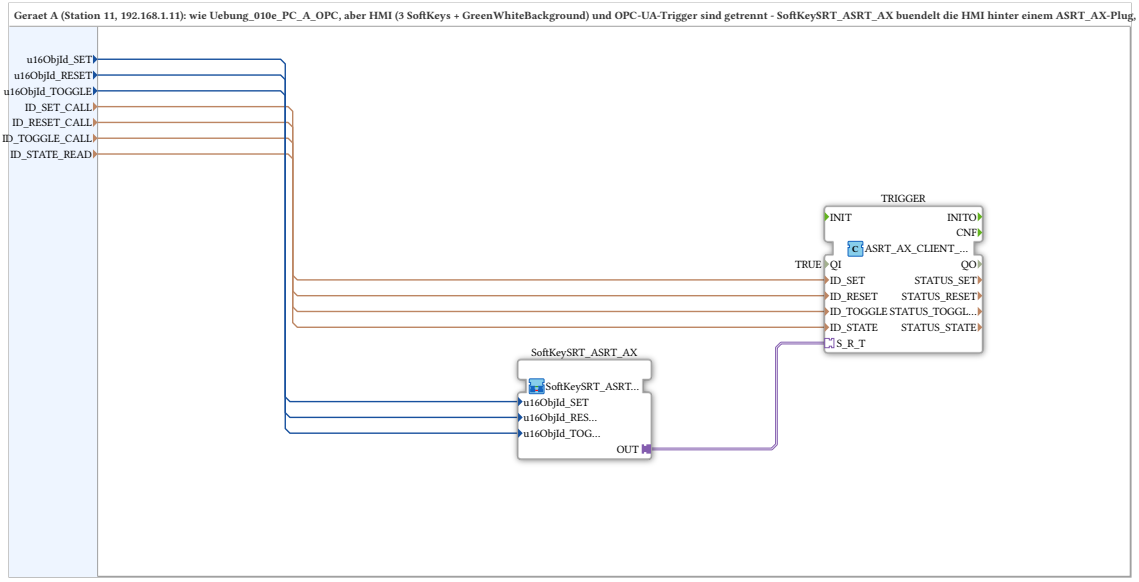
Summary

Device-A side of a set/reset/toggle RPC pattern with 3 distinct method calls and state feedback.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

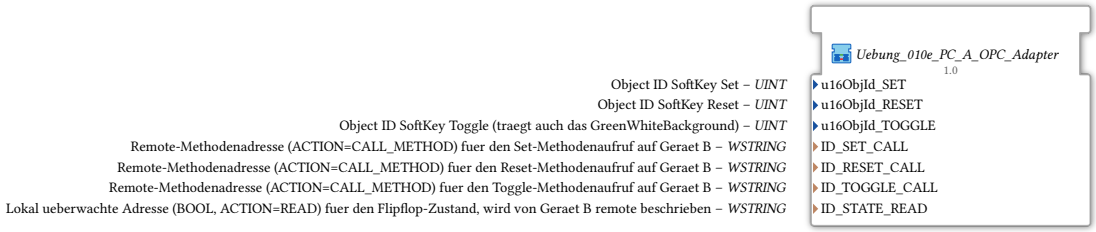
Uebung_010e_PC_A_OPC_Adapter



• — ○ • — ○ • — ○ •

Introduction

`Uebung_010e_PC_A_OPC_Adapter` is the adapter-bundled variant of `Uebung_010e_PC_A_OPC` (device A, station 11): HMI (3 softkeys + `GreenWhiteBackground`) and OPC-UA trigger are separated. `SoftKeySRT_ASRT_AX` bundles the HMI behind an `ASRT_AX` plug; `ASRT_AX_CLIENT_0_SUBSCRIBE_1` bundles the 3 `CLIENT_0` instances + `AX_SUBSCRIBE_1` behind an `ASRT_AX` socket. Counterpart: `Uebung_010e_PC_B_OPC_Adapter`.



Function blocks used

- **SoftKeySRT_ASRT_AX** (SubApp, type `MyLib::sys::SoftKeySRT_ASRT_AX`): 3 softkeys + status display, bundled behind an `ASRT_AX` plug.
- **TRIGGER** (`adapter::net::ASRT_AX_CLIENT_0_SUBSCRIBE_1`): bundles 3 method calls (`ID_SET_CALL`/`ID_RESET_CALL`/`ID_TOGGLE_CALL`) and the state subscription (`ID_STATE_READ`) behind a single `ASRT_AX` socket.

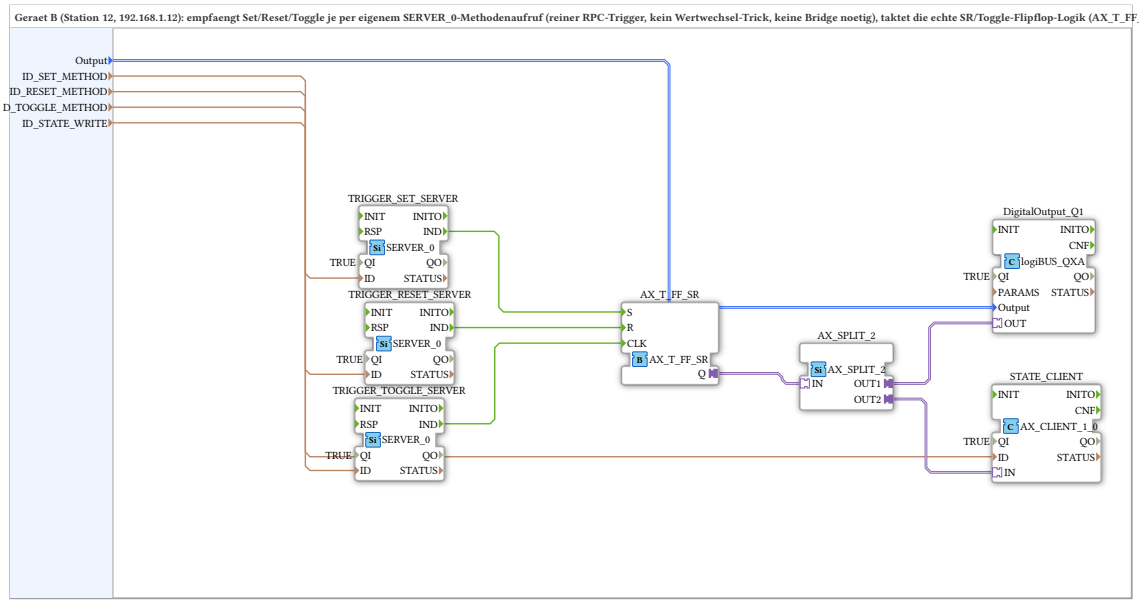
Summary

Adapter-bundled variant of `Uebung_010e_PC_A_OPC`: HMI and network protocol cleanly separated, reusable via `SoftKeySRT_ASRT_AX`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

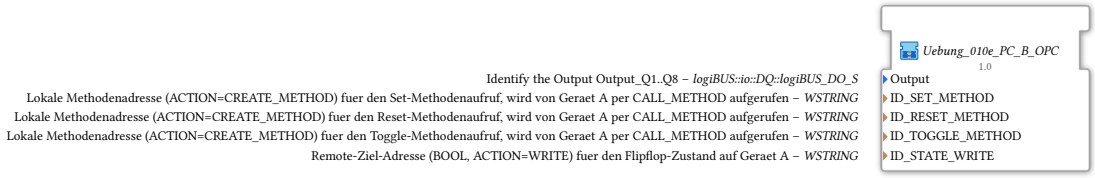
Uebung_010e_PC_B_OPC



• – ○ • – ○ • – ○ •

Introduction

`Uebung_010e_PC_B_OPC` is the device-B side (station 12, 192.168.1.12): receives set/reset/toggle each via its own `SERVER_0` method call (a pure RPC trigger, no value-change trick, no bridge needed), clocks the actual SR/toggle flip-flop logic (`AX_T_FF_SR`), drives `DigitalOutput_Q1`, and actively writes the new state back to device A via `AX_CLIENT_1_0`. Counterpart: `Uebung_010e_PC_A_OPC`.



Function blocks used

- **TRIGGER_SET_SERVER / TRIGGER_RESET_SERVER / TRIGGER_TOGGLE_SERVER** (`iec61499::net::SERVER_0`): each receives its own method call.
- **AX_T_FF_SR** (`adapter::events::unidirectional::AX_T_FF_SR`): combined set/reset/toggle flip-flop logic.
- **AX_SPLIT_2** (`adapter::events::unidirectional::AX_SPLIT_2`): splits the state to the physical output + the echo.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physical output.
- **STATE_CLIENT** (`adapter::net::AX_CLIENT_1_0`): actively writes the state to device A under `ID_STATE_WRITE`.

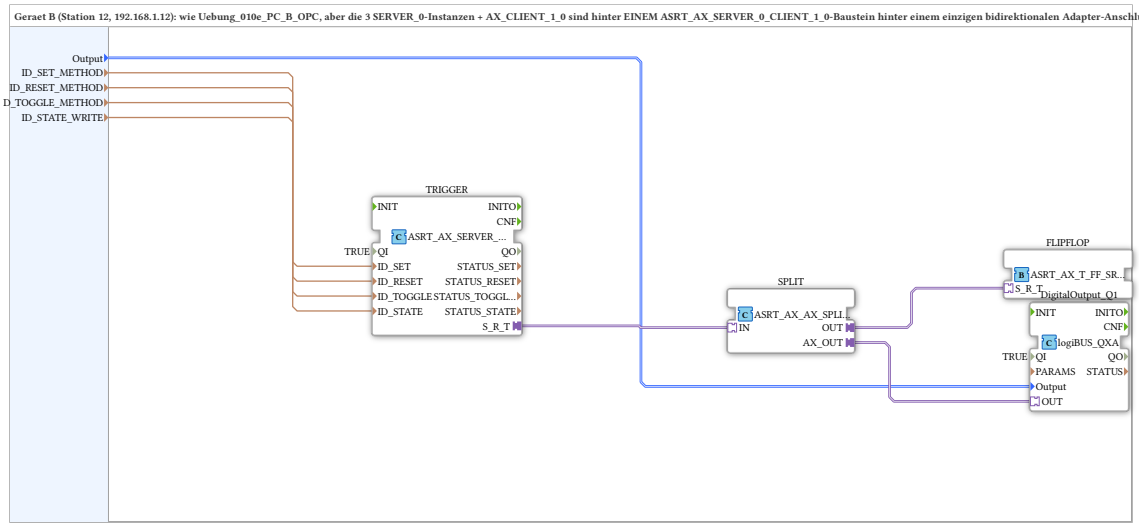
Summary

Device-B side: receives set/reset/toggle via RPC, holds the combined SR/toggle flip-flop logic, and actively reports the state back.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

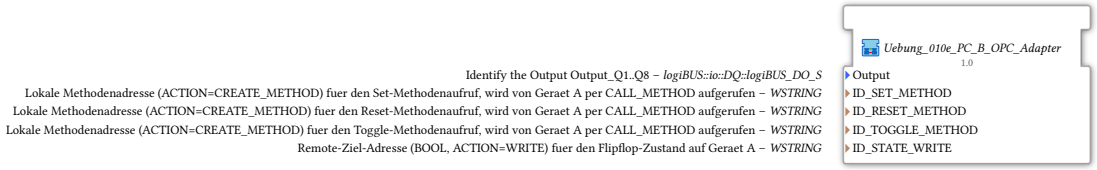
Uebung_010e_PC_B_OPC_Adapter



• — ○ • — ○ • — ○ •

Introduction

`Uebung_010e_PC_B_OPC_Adapter` is the adapter-bundled variant of `Uebung_010e_PC_B_OPC` (device B, station 12): the 3 `SERVER_0` instances + `AX_CLIENT_1_0` are bundled behind a SINGLE `ASRT_AX_SERVER_0_CLIENT_1_0` block behind one bidirectional adapter connection; `ASRT_AX_AX_SPLIT` feeds both `DigitalOutput_Q1` and the SR/toggle flip-flop logic (`ASRT_AX_T_FF_SR_2`). Counterpart: `Uebung_010e_PC_A_OPC_Adapter`.



Function blocks used

- **TRIGGER** (`adapter::net::ASRT_AX_SERVER_0_CLIENT_1_0`): bundles server reception (`ID_SET_METHOD`/`ID_RESET_METHOD`/`ID_TOGGLE_METHOD`) and state feedback (`ID_STATE_WRITE`).
- **SPLIT** (`adapter::events::bidirectional::ASRT_AX_AX_SPLIT`): splits to the physical output + the flip-flop logic.
- **FLIPFLOP** (`adapter::events::bidirectional::ASRT_AX_T_FF_SR_2`): combined SR/toggle flip-flop logic as an adapter block.
- **DigitalOutput_Q1** (`logiBUS::io::DQ::logiBUS_QXA`): physical output.

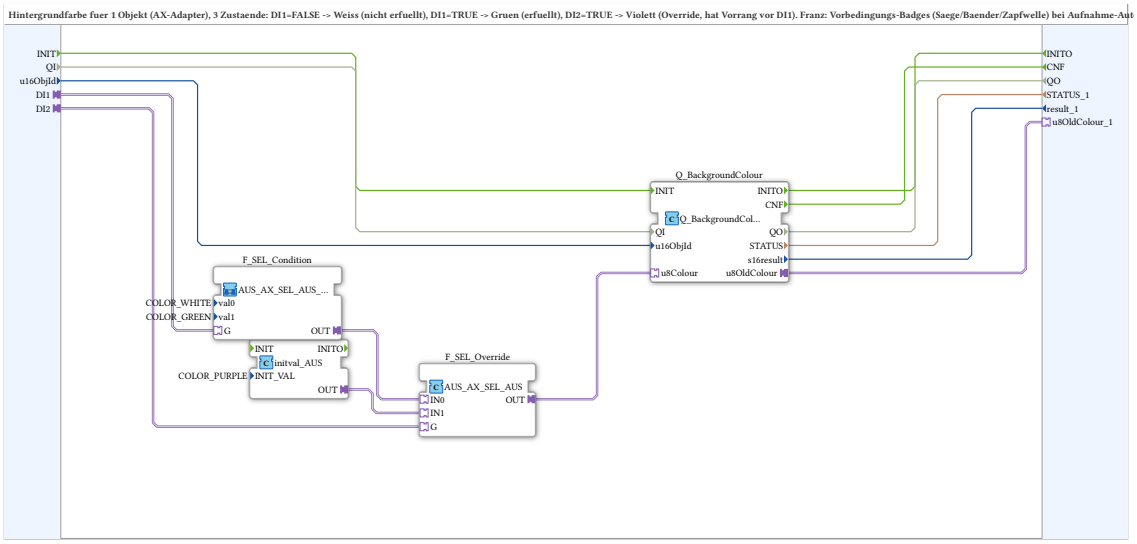
Summary

Adapter-bundled variant of `Uebung_010e_PC_B_OPC`: network protocol and SR/toggle flip-flop logic as reusable adapter blocks.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

WhiteGreenPurpleBackground1_AX



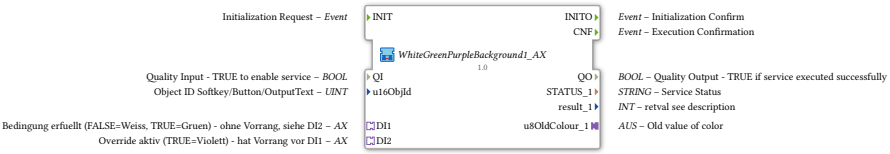
• – ○ • – ○ • – ○ •

Introduction

`WhiteGreenPurpleBackground1_AX` sets the background colour for **one object** (`AX` adapter) with three states:

- `DI1=FALSE` ⟶ **white** (not fulfilled)
- `DI1=TRUE` ⟶ **green** (fulfilled)
- `DI2=TRUE` ⟶ **purple** (override, takes precedence over `DI1`)

It is used, for example, for precondition badges (saw/belts/PTO) of the pick-up automation. The block is built analogously to `GreenWhiteBackground1_AX`.



Function Blocks (FBs) Used

Sub-blocks: WhiteGreenPurpleBackground1_AX

- **Type**: SubAppType
- **Internal blocks used**:
  - **F_SEL_Condition**: SubApp `MyLib::sys::AUS_AX_SEL_AUS_2_VAL` with `val0=COLOR_WHITE`, `val1=COLOR_GREEN` - selects white or green based on `DI1`
  - **F_SEL_Override**: `adapter::iec61131::selection::AUS_AX_SEL_AUS` - selects between the condition result (`IN0`) and purple (`IN1`) based on `DI2`
  - **initval_AUS**: `adapter::types::unidirectional::AUS::initval::initval_AUS` with `INIT_VAL=COLOR_PURPLE`
  - **Q_BackgroundColour**: `isobus::UT::Q::Q_BackgroundColour_AUS` - writes the colour to the VT object `u16ObjId`
- **Imports**: `ID_NULL`, `COLOR_GREEN`, `COLOR_WHITE`, `COLOR_PURPLE`

Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization request (⟶ `Q_BackgroundColour.INIT`)

Further events arrive via the adapters `DI1` and `DI2`.

Event Outputs

- **INITO** (EInit): Initialization confirmation (from Q_BackgroundColour.INITO)
- **CNF**: Execution confirmation (from Q_BackgroundColour.CNF)

Data Inputs

- **QI** (BOOL, default TRUE): Quality Input - TRUE enables the service (⟶ Q_BackgroundColour.QI)
- **u16ObjId** (UINT, default ID_NULL): Object ID of softkey/button/output text

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if the service executed successfully (from Q_BackgroundColour.QO)
- **STATUS_1** (STRING): Service status (from Q_BackgroundColour.STATUS)
- **result_1** (INT): Return value (from Q_BackgroundColour.s16result)

Adapters

- **DI1** (socket, adapter::types::unidirectional::AX): Condition fulfilled (FALSE = white, TRUE = green), without precedence, see DI2
- **DI2** (socket, adapter::types::unidirectional::AX): Override active (TRUE = purple), takes precedence over DI1
- **u8OldColour_1** (plug, adapter::types::unidirectional::AUS): Old colour value

Program Flow and Connections

- DI1 ⟶ F_SEL_Condition.G
- DI2 ⟶ F_SEL_Override.G
- F_SEL_Condition.OUT ⟶ F_SEL_Override.IN0; initval_AUS.OUT ⟶ F_SEL_Override.IN1
- F_SEL_Override.OUT ⟶ Q_BackgroundColour.u8Colour
- u16ObjId ⟶ Q_BackgroundColour.u16ObjId
- INIT ⟶ Q_BackgroundColour.INIT; Q_BackgroundColour.INITO ⟶ INITO
- QI ⟶ Q_BackgroundColour.QI; Q_BackgroundColour.QO ⟶ QO
- Q_BackgroundColour.u8OldColour ⟶ u8OldColour_1; .STATUS ⟶ STATUS_1; .s16result ⟶ result_1; .CNF ⟶ CNF

Technical Details

- Two nested selectors (not a single AX_SEL), since a single AX_SEL re-evaluates only when its **own** G changes and the second stage would otherwise not follow up on a change of DI1
- Both selectors are re-evaluated in series via a common trigger (E_MERGE_2 from DI1.E1/DI2.E1), so that a change of DI1 **or** DI2 always yields the complete, correct colour
- Version 1.0 (2026-10-01)

State Overview

DI1	DI2	Colour
FALSE	FALSE	White (COLOR_WHITE)
TRUE	FALSE	Green (COLOR_GREEN)
any	TRUE	Purple (COLOR_PURPLE)

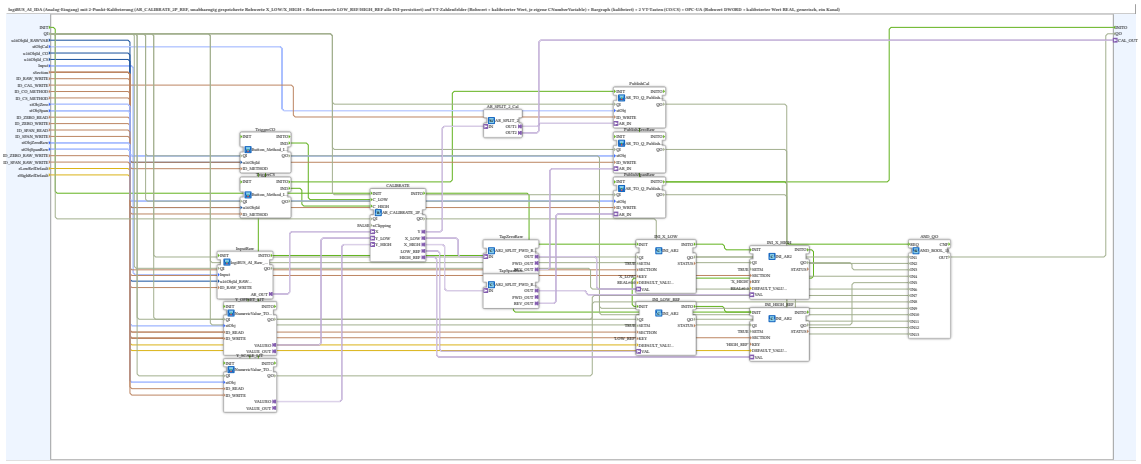
Application Scenarios

- Precondition badges (saw, belts, PTO) of a pick-up automation with override indication

Summary

WhiteGreenPurpleBackground1_AX colours a VT object white, green or purple, where the override DI2 takes precedence over the condition DI1.

logiBUS_AI_Calibrate_2P_IDA_OPC



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_AI_Calibrate_2P_IDA_OPC` connects a physical analog input (`logiBUS_AI_IDA`) to a full VT- and OPC-UA-backed 2-point calibration (`AR_CALIBRATE_2P_REF`) — used by the AI_Calibrate_2P training sample. The input's raw value is converted into a physically scaled value through the 2-point calibration chain; zero point (`Zero`) and span (`Span`) as well as their measured raw values (`ZeroRaw`/`SpanRaw`) are adjustable and readable via both VT and OPC-UA, and are persisted in an INI file. In partial calibration (only one point calibrated), a clean fallback logic immediately utilizes the available reference value.



Function blocks used

- **logiBUS_AI_IDA** (`logiBUS::io::AI::logiBUS_AI_IDA`): physical analog input, provides the raw value as an adapter.
- **AR_CALIBRATE_2P_REF** (adapter composite): 2-point calibration (zero/span) with raw value diagnostics (`ZeroRaw`/`SpanRaw`), partial calibration fallback logic, and configurable clipping (`xClipping`).
- **VT and OPC-UA bridges** (analogous to `NumericValue_TO_AR2_OPC`/`OPC_TO_AR2` and `INI_IN_AND_STORE_AR2`): expose `Zero`/`Span` and `ZeroRaw`/`SpanRaw` both via VT input fields/displays and via OPC-UA, and persist calibration values in an INI file.

Initialization and quality

- Events: `INIT` (EInit, input) and `INITO` (EInit, output, comes from `PublishSpanRaw.INITO`).
- Data input `QI` (BOOL, default `TRUE`) and data output `QO` (BOOL); the internal FBs (`CALIBRATE`, `INI_X_LOW`, `INI_X_HIGH`, `INI_LOW_REF`, `INI_HIGH_REF`) no longer carry a fixed `QI=TRUE`.
- Initialization chain (each `INITO` ⟶ `INIT` of the next): `INIT` ⟶ `CALIBRATE` ⟶ `INI_X_LOW` ⟶ `INI_X_HIGH` ⟶ `INI_LOW_REF` ⟶ `INI_HIGH_REF` ⟶ `InputRaw` ⟶ `Y_OFFSET_LIT` ⟶ `Y_SCALE_LIT` ⟶ `TriggerCO` ⟶ `TriggerCS` ⟶ `PublishCal` ⟶ `PublishZeroRaw` ⟶ `PublishSpanRaw`; `PublishSpanRaw.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
- Quality: `QI` is passed on to all 13 instances of this chain; their `QO` outputs feed `AND_QO.IN1` to `IN13` in exactly this order (a single `iec61131::booleanOperators::AND_BOOL_13`; the former cascade of `AND_QO_A` (`AND_BOOL_5`) and `AND_QO` (`AND_BOOL_9`) is replaced, version 1.12); `AND_QO.OUT` ⟶ `QO`.

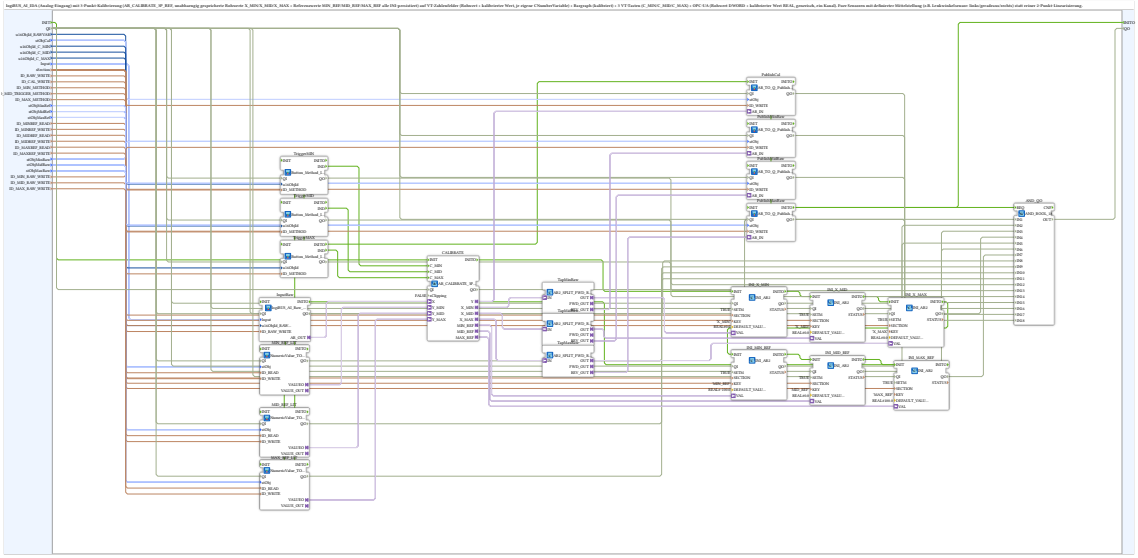
Summary

Complete 2-point calibration block for a physical analog input: raw-value acquisition, 2-point calibration with raw-value diagnostics, immediate partial calibration fallback, VT display/input, and OPC-UA connectivity in one composite.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

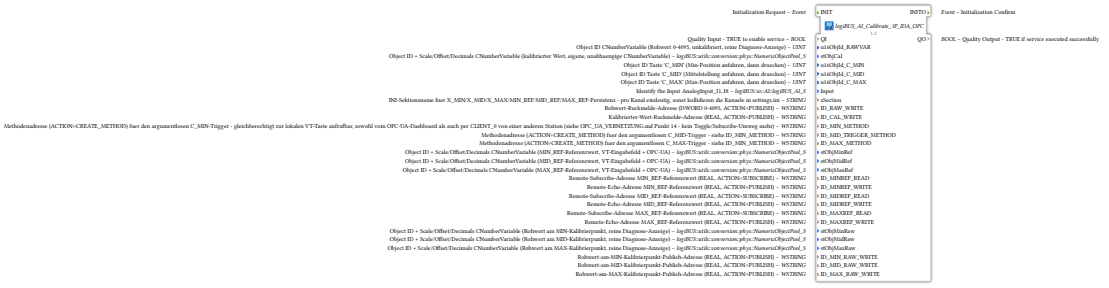
logiBUS_AI_Calibrate_3P_IDA_OPC



• — ○ • — ○ • — ○ •

Introduction

`logiBUS_AI_Calibrate_3P_IDA_OPC` reads a logiBUS analog input (`logiBUS_AI_IDA`) and calibrates it with a **3-point calibration** (`AR_CALIBRATE_3P_REF`): the three raw values `X_MIN/X_MID/X_MAX` and the three reference values `MIN_REF/MID_REF/MAX_REF` are stored independently of each other and all persisted in an INI file. The raw and the calibrated value are output to VT number fields (each its own `CNumberVariable`), three VT keys (`C_MIN/C_MID/C_MAX`) trigger the calibration, and the values are published via OPC-UA (raw value `DWORD`, calibrated value `REAL`; generic, one channel). The block is intended for sensors with a defined center position (e.g. steering angle sensor: left/straight/right) instead of a pure 2-point linearization.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the init chain at `CALIBRATE.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `PublishMaxRaw.INITO`, the end of the init chain

Data Inputs

General:

- **QI** (`BOOL`, default `TRUE`): Quality input, enables the service; passed on to all 18 service instances (see `AND_QO`)

Input and display:

- **Input** (logiBUS::io::AI::logiBUS_AI_S, default logiBUS_AI::Invalid): Identifies the analog input AnalogInput_I1..I8
- **u16ObjId_RAWVAR** (UINT, default ID_NULL): Object ID of the CNumberVariable for the uncalibrated raw value (0-4095, pure diagnostic display)
- **stObjCal** (NumericObjectPool_S): Object ID + scale/offset/decimals of the CNumberVariable for the calibrated value (own, independent variable)
- **sSection** (STRING): INI section name for the persistence of X_MIN/X_MID/X_MAX/MIN_REF/MID_REF/MAX_REF; unique per channel, otherwise the channels collide in settings.ini

VT keys and methods:

- **u16ObjId_C_MIN**, **u16ObjId_C_MID**, **u16ObjId_C_MAX** (UINT, default ID_NULL): Object IDs of the keys C_MIN (approach min position, then press), C_MID (center position) and C_MAX (max position)
- **ID_MIN_METHOD**, **ID_MID_TRIGGER_METHOD**, **ID_MAX_METHOD** (WSTRING): Method addresses (ACTION=CREATE_METHOD) for the argument-less triggers; equivalent to the local VT key, callable from the OPC-UA dashboard and via CLIENT_0 from another station

Reference values (VT input field + OPC-UA):

- **stObjMinRef**, **stObjMidRef**, **stObjMaxRef** (NumericObjectPool_S): Input fields for MIN_REF/MID_REF/MAX_REF
- **ID_MINREF_READ**, **ID_MIDREF_READ**, **ID_MAXREF_READ** (WSTRING): Remote subscribe addresses of the reference values (REAL, ACTION=SUBSCRIBE)
- **ID_MINREF_WRITE**, **ID_MIDREF_WRITE**, **ID_MAXREF_WRITE** (WSTRING): Remote echo addresses of the reference values (REAL, ACTION=PUBLISH)

Raw values at the calibration points (pure diagnostic display):

- **stObjMinRaw**, **stObjMidRaw**, **stObjMaxRaw** (NumericObjectPool_S): Display fields for the raw value at the MIN/MID/MAX calibration point
- **ID_MIN_RAW_WRITE**, **ID_MID_RAW_WRITE**, **ID_MAX_RAW_WRITE** (WSTRING): Publish addresses (REAL, ACTION=PUBLISH)

OPC-UA outputs:

- **ID_RAW_WRITE** (WSTRING): Raw value feedback address (DWORD 0-4095, ACTION=PUBLISH)
- **ID_CAL_WRITE** (WSTRING): Feedback address of the calibrated value (REAL, ACTION=PUBLISH)

Data Outputs

- **QO** (BOOL): Quality output, AND combination of all 18 QO sources in a single AND_QO block (iec61131::booleanOperators::AND_BOOL_18)

Adapters

This function block has no adapters at its interface; AR adapters are used internally.

Function Blocks (FBs) Used

- **InputRaw**: SubApp MyLib::sys::logiBUS_AI_Raw_VT_AR_OPC - reads the analog input, displays the raw value and publishes it
- **CALIBRATE**: adapter::Engineering::measurements::AR_CALIBRATE_3P_REF (xClipping=FALSE, QI comes from the input QI) - 3-point calibration
- **MIN_REF_LIT**, **MID_REF_LIT**, **MAX_REF_LIT**: SubApp MyLib::sys::NumericValue_TO_AR2_OPC - reference values from VT input field or OPC-UA
- **TriggerMIN**, **TriggerMID**, **TriggerMAX**: SubApp MyLib::sys::Button_Method_IE - key OR local method → IND
- **INI_X_MIN**, **INI_X_MID**, **INI_X_MAX**: eclipse4diac::storage::INI_AR2 (keys X_MIN/X_MID/X_MAX, default 0.0)
- **INI_MIN_REF**, **INI_MID_REF**, **INI_MAX_REF**: INI_AR2 (keys MIN_REF/MID_REF/MAX_REF, defaults -100.0/0.0/100.0)
- **TapMinRaw**, **TapMidRaw**, **TapMaxRaw**: adapter::events::bidirectional::AR2_SPLIT_FWD_REV
- **PublishCal**, **PublishMinRaw**, **PublishMidRaw**, **PublishMaxRaw**: SubApp MyLib::sys::AR_TO_Q_Publish_OPC - VT display and OPC-UA publish

- **AND_QO**: `iec61131::booleanOperators::AND_BOOL_18` - combines the `QO` outputs of all 18 service instances (`CALIBRATE`, 6× `INI_AR2`, `InputRaw`, 3× `*_REF_LIT`, 3× `Trigger*`, `PublishCal`, 3× `Publish*Raw`) into the output `QO` (formerly a cascade of several smaller AND blocks)

All `INI_AR2` use `SETM=TRUE`; their `QI` comes from the input `QI`; `sSection` is their common `SECTION`.

Program Flow and Connections

1. `InputRaw.AR_OUT` ⟶ `CALIBRATE.X` (raw value).
2. `TriggerMIN.IND/TriggerMID.IND/TriggerMAX.IND` ⟶ `CALIBRATE.C_MIN/C_MID/C_MAX`.
3. Reference values: `MIN_REF_LIT.VALUEO/MID_REF_LIT.VALUEO/MAX_REF_LIT.VALUEO` ⟶ `CALIBRATE.Y_MIN/Y_MID/Y_MAX`.
4. Stored raw values: `CALIBRATE.X_MIN/X_MID/X_MAX` ⟶ `TapMinRaw.IN/TapMidRaw.IN/TapMaxRaw.IN`; their `OUT` ⟶ `INI_X_MIN.VAL/INI_X_MID.VAL/INI_X_MAX.VAL`; `REV_OUT` ⟶ `PublishMinRaw/PublishMidRaw/PublishMaxRaw.AR_IN`.
5. Reference values (persistence): `CALIBRATE.MIN_REF/MID_REF/MAX_REF` ⟶ `INI_MIN_REF.VAL/INI_MID_REF.VAL/INI_MAX_REF.VAL`.
6. Calibrated value: `CALIBRATE.Y` ⟶ `PublishCal.AR_IN`.
7. Initialization chain: `INIT` ⟶ `CALIBRATE` ⟶ `INI_X_MIN` ⟶ `INI_X_MID` ⟶ `INI_X_MAX` ⟶ `INI_MIN_REF` ⟶ `INI_MID_REF` ⟶ `INI_MAX_REF` ⟶ `InputRaw` ⟶ `MIN_REF_LIT` ⟶ `MID_REF_LIT` ⟶ `MAX_REF_LIT` ⟶ `TriggerMIN` ⟶ `TriggerMID` ⟶ `TriggerMAX` ⟶ `PublishCal` ⟶ `PublishMinRaw` ⟶ `PublishMidRaw` ⟶ `PublishMaxRaw` (each `INITO` ⟶ next `INIT`); `PublishMaxRaw.INITO` ⟶ `INITO` and `AND_QO.REQ`.
8. Quality: `QI` ⟶ `.QI` of all 18 instances; `CALIBRATE.QO` ⟶ `AND_QO.IN1`; `INI_X_MIN/INI_X_MID/INI_X_MAX.QO` ⟶ `IN2/IN3/IN4`; `INI_MIN_REF/INI_MID_REF/INI_MAX_REF.QO` ⟶ `IN5/IN6/IN7`; `InputRaw.QO` ⟶ `IN8`; `MIN_REF_LIT/MID_REF_LIT/MAX_REF_LIT.QO` ⟶ `IN9/IN10/IN11`; `TriggerMIN/TriggerMID/TriggerMAX.QO` ⟶ `IN12/IN13/IN14`; `PublishCal.QO` ⟶ `IN15`; `PublishMinRaw/PublishMidRaw/PublishMaxRaw.QO` ⟶ `IN16/IN17/IN18`; `AND_QO.OUT` ⟶ `QO`.

Technical Details

- All six calibration corner values (three raw values, three reference values) are INI-persisted independently of each other via one `INI_AR2` each.
- `xClipping=FALSE`: the calibrated value is not clamped.
- `sSection` must be unique per channel.

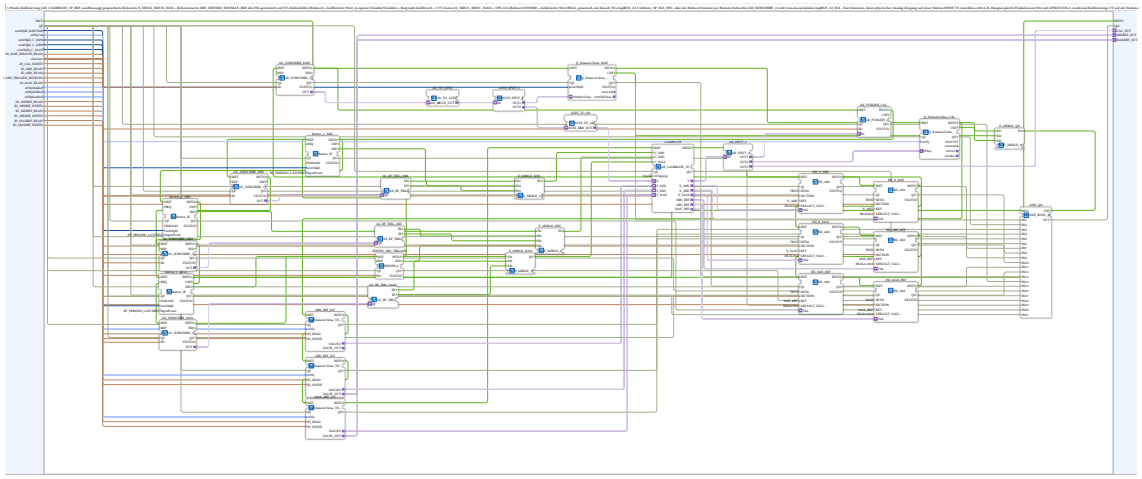
Application Scenarios

- Sensors with a defined center position, e.g. steering angle sensor (left/straight/right)
- Calibration via VT key, OPC-UA dashboard or remote call from another station

Summary

`logiBUS_AI_Calibrate_3P_IDA_OPC` bundles analog input, 3-point calibration, INI persistence, VT display and OPC-UA connection of one channel in one SubApp.

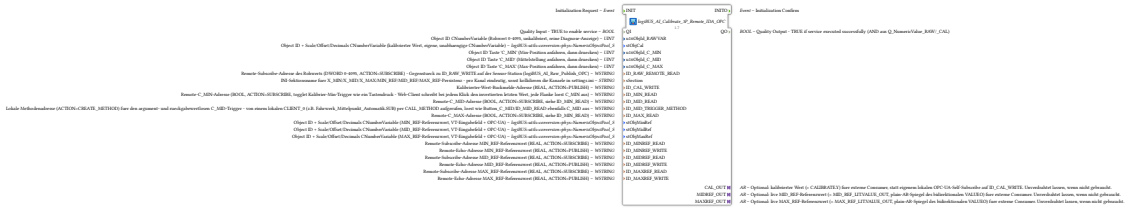
logiBUS_AI_Calibrate_3P_Remote_IDA_OPC



• – ◦ • – ◦ • – ◦ •

Introduction

`logiBUS_AI_Calibrate_3P_Remote_IDA_OPC` performs a **3-point calibration** (`AR_CALIBRATE_3P_REF`) for an analog raw value. The raw values `X_MIN`/`X_MID`/`X_MAX` and the reference values `MIN_REF`/`MID_REF`/`MAX_REF` are stored independently and all INI-persisted. The results appear on VT number fields (raw value and calibrated value, each with its own CNumberVariable), three VT buttons (`C_MIN`/`C_MID`/`C_MAX`) trigger the calibration points, and the raw value (DWORD) and the calibrated value (REAL) are provided generically for one channel via OPC-UA. In contrast to `logiBUS_AI_Calibrate_3P_IDA_OPC`, the raw value comes **via remote subscribe** (`AD_SUBSCRIBE_1`) instead of from a local `logiBUS_AI_IDA`; this allows sensors on a station without a VT connection to be calibrated (e.g. slope compensation angle sensor B14 on STG3) while calibration and VT run on the station with the VT connection. The raw value is published there by `logiBUS_AI_Raw_Publish_OPC`.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the initialization chain (first link `Q_NumericValue_RAW.INIT`)

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `Button_C_MAX.INITO`, the end of the chain

Data Inputs

- **QI** (BOOL): quality input - TRUE enables the service (default `TRUE`); distributed to all 21 inner instances
- **u16ObjId_RAWVAR** (UINT): object ID of the CNumberVariable for the raw value (0-4095, uncalibrated, pure diagnostic display)
- **stObjCal** (`NumericObjectPool_S`): object ID + scale/offset/decimals of the CNumberVariable for the calibrated value (own, independent variable)
- **u16ObjId_C_MIN**, **u16ObjId_C_MID**, **u16ObjId_C_MAX** (UINT): object IDs of the buttons `C_MIN`, `C_MID`, `C_MAX` (move to the position, then press)
- **ID_RAW_REMOTE_READ** (WSTRING): remote subscribe address of the raw value (DWORD 0-4095, `ACTION=SUBSCRIBE`); counterpart to `ID_RAW_WRITE` on the sensor station
- **sSection** (STRING): INI section name for the persistence of `X_MIN`/`X_MID`/`X_MAX`/`MIN_REF`/`MID_REF`/`MAX_REF`; unique per channel, otherwise the channels collide in `settings.ini`
- **ID_CAL_WRITE** (WSTRING): feedback address of the calibrated value (REAL, `ACTION=PUBLISH`)
- **ID_MIN_READ**, **ID_MID_READ**, **ID_MAX_READ** (WSTRING): remote addresses (BOOL, `ACTION=SUBSCRIBE`) for `C_MIN`/`C_MID`/`C_MAX`; every edge (the web client writes the inverted last value on each click) triggers the calibration point like a button press
- **ID_MID_TRIGGER_METHOD** (WSTRING): local method address (`ACTION=CREATE_METHOD`) for the argument-less and return-value-less `C_MID` trigger; called by a local `CLIENT_0` (e.g. `Fahrwerk_Mittelpunkt_Automatik.SUB`) via `CALL_METHOD` and triggers `C_MID` like `Button_C_MID`/`ID_MID_READ`
- **stObjMinRef**, **stObjMidRef**, **stObjMaxRef** (`NumericObjectPool_S`): object ID + scale/offset/decimals of the reference values `MIN_REF`/`MID_REF`/`MAX_REF` (VT input field + OPC-UA)
- **ID_MINREF_READ**, **ID_MIDREF_READ**, **ID_MAXREF_READ** (WSTRING): remote subscribe addresses of the reference values (REAL, `ACTION=SUBSCRIBE`)
- **ID_MINREF_WRITE**, **ID_MIDREF_WRITE**, **ID_MAXREF_WRITE** (WSTRING): remote echo addresses of the reference values (REAL, `ACTION=PUBLISH`)

Data Outputs

- **QO** (BOOL): quality output - TRUE if the service was executed successfully (AND of the `QO` of all 21 inner instances, see section Functionality)

Adapters

- **CAL_OUT** (plug, AR): optional; calibrated value (CALIBRATE.Y) for external consumers instead of an own local OPC-UA self-subscribe on ID_CAL_WRITE
- **MIDREF_OUT** (plug, AR): optional; live MID_REF reference value (MID_REF_LIT.VALUE_OUT, plain AR mirror of the bidirectional VALUEO) for external consumers
- **MAXREF_OUT** (plug, AR): optional; live MAX_REF reference value (MAX_REF_LIT.VALUE_OUT) for external consumers

Functionality

Raw value: AD_SUBSCRIBE_RAW (adapter::net::AD_SUBSCRIBE_1, QI from outside, address ID_RAW_REMOTE_READ) ⟶ AD_TO_AUDI ⟶ AUDI_SPLIT_2. OUT1 ⟶ Q_NumericValue_RAW.u32NewValue (Q_NumericValue_AUDI, raw value display on u16ObjId_RAWVAR); OUT2 ⟶ AUDI_TO_AR ⟶ CALIBRATE.X.

Calibration: CALIBRATE (adapter::Engineering::measurements::AR_CALIBRATE_3P_REF, xClipping = FALSE, QI from outside) maps the raw value X via the three points to Y. Y_MIN/Y_MID/Y_MAX come from MIN_REF_LIT/MID_REF_LIT/MAX_REF_LIT (each MyLib::sys::NumericValue_TO_AR2_OPC VT input field and OPC-UA, last writer wins).

Result: CALIBRATE.Y ⟶ AR_SPLIT_2 (AR_SPLIT_3): OUT1 ⟶ AR_PUBLISH_CAL (AR_PUBLISH_1, address ID_CAL_WRITE), OUT2 ⟶ Q_NumericValue_CAL.rPhys (Q_NumericValue_PHYSA, display with stObjCal), OUT3 ⟶ CAL_OUT. MID_REF_LIT.VALUE_OUT ⟶ MIDREF_OUT and MAX_REF_LIT.VALUE_OUT ⟶ MAXREF_OUT.

Triggers: every calibration point has its own event merge. C_MIN: Button_C_MIN.IND (Button_IE, BT_PRESSED_LATCHED), AX_RF_TRIG_MIN.ER and .EF (both edges of the remote subscribe AX_SUBSCRIBE_MIN, address ID_MIN_READ) ⟶ E_MERGE_MIN (E_MERGE_3) ⟶ CALIBRATE.C_MIN. C_MAX analogously via Button_C_MAX, AX_SUBSCRIBE_MAX, AX_RF_TRIG_MAX, E_MERGE_MAX (E_MERGE_3). C_MID additionally with SERVER_MID_TRIGGER (iec61499::net::SERVER_0, method ID_MID_TRIGGER_METHOD) ⟶ E_MERGE_MID (E_MERGE_4) ⟶ CALIBRATE.C_MID.

Persistence: CALIBRATE.X_MIN/X_MID/X_MAX and MIN_REF/MID_REF/MAX_REF each go to one INI_AR2 instance (INI_X_MIN, INI_X_MID, INI_X_MAX, INI_MIN_REF, INI_MID_REF, INI_MAX_REF; SETM = TRUE, SECTION from sSection, KEY = name). The defaults are X_MIN/X_MID/X_MAX = REAL#0.0 as well as MIN_REF = REAL#-100.0, MID_REF = REAL#50.0, MAX_REF = REAL#100.0.

Initialisation: INIT runs as one single chain (each INITO ⟶ next INIT) through Q_NumericValue_RAW ⟶ Q_NumericValue_CAL ⟶ MIN_REF_LIT ⟶ MID_REF_LIT ⟶ MAX_REF_LIT ⟶ CALIBRATE ⟶ INI_X_MIN ⟶ INI_X_MID ⟶ INI_X_MAX ⟶ INI_MIN_REF ⟶ INI_MID_REF ⟶ INI_MAX_REF ⟶ AD_SUBSCRIBE_RAW ⟶ AR_PUBLISH_CAL ⟶ AX_SUBSCRIBE_MIN ⟶ AX_SUBSCRIBE_MID ⟶ AX_SUBSCRIBE_MAX ⟶ SERVER_MID_TRIGGER ⟶ Button_C_MIN ⟶ Button_C_MID ⟶ Button_C_MAX; Button_C_MAX.INITO goes to the event output INITO and to AND_QO.REQ.

Quality: QI is passed to all 21 instances of the chain. QO is the output of the instance AND_QO (iec61131::booleanOperators::AND_BOOL_21), which combines the QO of these 21 instances in a single block (IN1..IN21: Q_NumericValue_RAW, Q_NumericValue_CAL, MIN_REF_LIT, MID_REF_LIT, MAX_REF_LIT, CALIBRATE, INI_X_MIN, INI_X_MID, INI_X_MAX, INI_MIN_REF, INI_MID_REF, INI_MAX_REF, AD_SUBSCRIBE_RAW, AR_PUBLISH_CAL, AX_SUBSCRIBE_MIN, AX_SUBSCRIBE_MID, AX_SUBSCRIBE_MAX, SERVER_MID_TRIGGER, Button_C_MIN, Button_C_MID, Button_C_MAX). In addition there is E_MERGE_QO (E_MERGE_3), which merges Q_NumericValue_CAL.INITO and the CNF events of Q_NumericValue_RAW and Q_NumericValue_CAL; its output EO triggers AND_QO.REQ, so QO is re-evaluated on every new Q_NumericValue_RAW/Q_NumericValue_CAL value (CNF) (up to v1.6 EO was not connected).

Technical Details

- The defaults of X_MID/X_MAX are deliberately 0.0: with 0.5/1.0 the calibrated flags xMidCal/xMaxCal of AR_CALIBRATE_3P_REF latched to TRUE at every restart although MID/MAX were never calibrated (fix in version 1.6)
- CALIBRATE has an INIT/QI/QO enable pattern; without QI = TRUE the block stays in state START and REQ never runs (version 1.5); here QI comes from the SubApp input QI (default TRUE) instead of a fixed parameter
- MIDREF_OUT/MAXREF_OUT are fed from VALUE_OUT, not from the bidirectional VALUEO: an AR_SPLIT_2 directly on an AR2 plug would be a type error (version 1.2)
- The optional plugs CAL_OUT/MIDREF_OUT/MAXREF_OUT allow external consumers to read the already calibrated value via adapter instead of subscribing to the same OPC-UA node locally again (FORTE: 'Cannot add another one'); leave unwired if not needed (e.g. slope compensation channel)

- The merge chains for `C_MIN`/`C_MID`/`C_MAX` are a single `E_MERGE_3`/`E_MERGE_4` each instead of several `E_MERGE_2` (OR is associative, functionally identical); `C_MIN`/`C_MID`/`C_MAX` of `CALIBRATE` are pure event inputs
- Remote triggers fire on **every** edge (rising and falling), since the web client writes the inverted last value on each click

Application Scenarios

- Calibration of an angle or position sensor (e.g. slope compensation) whose analog input sits on a different station than the VT
- Three-point calibration with min, centre and max position including adjustable reference values and persisted raw values

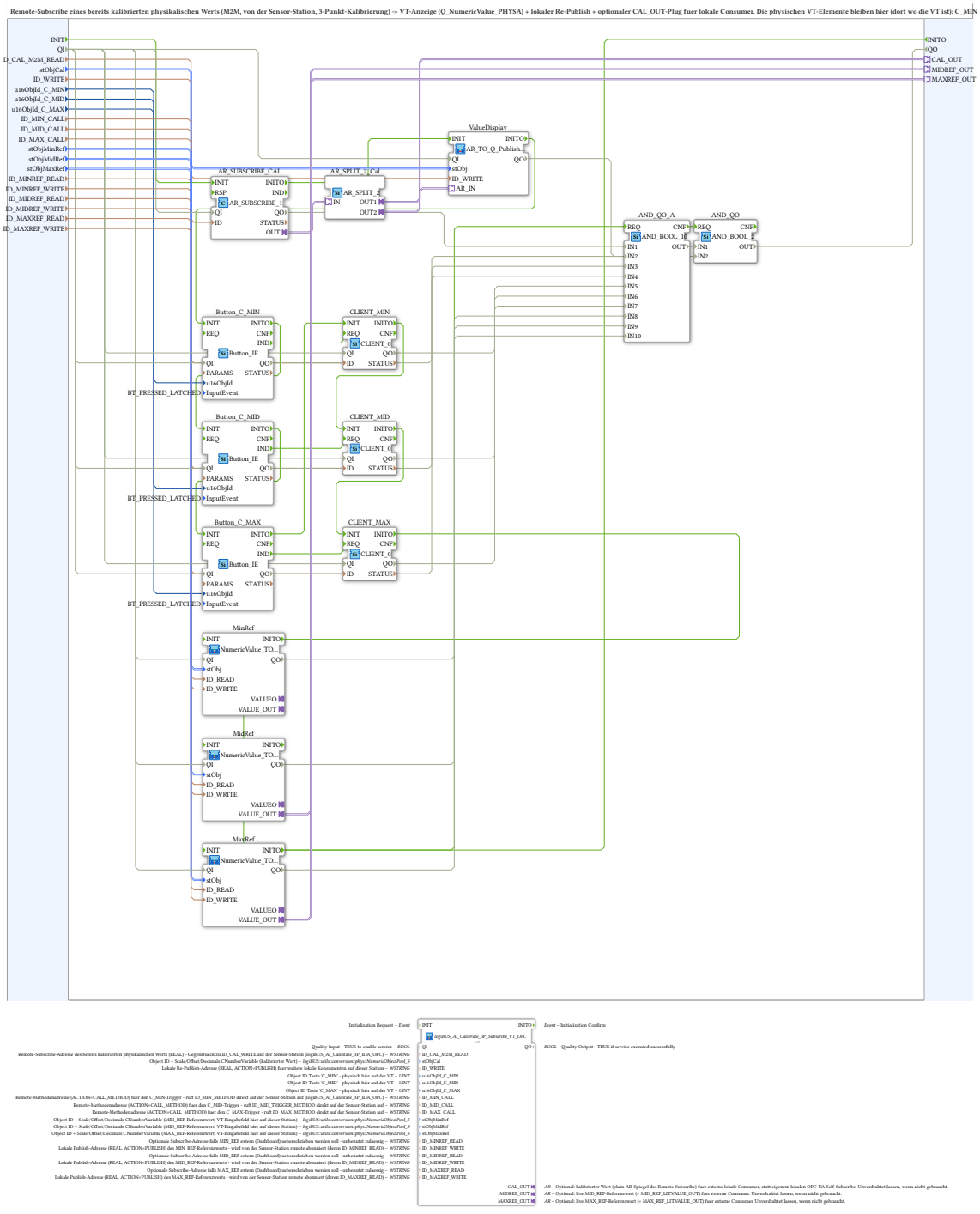
⚖️ Comparison with Similar Blocks

`logiBUS_AI_Calibrate_3P_IDA_OPC` reads the raw value locally from a `logiBUS_AI_IDA`; this remote variant replaces its `logiBUS_AI_IDA` + `AD_SPLIT_2` + `AD_PUBLISH_RAW` with an `AD_SUBSCRIBE_1`. The chain of calibration and VT connection is unchanged. The sensor side is `logiBUS_AI_Raw_Publish_OPC`.

Conclusion

`logiBUS_AI_Calibrate_3P_Remote_IDA_OPC` calibrates a remotely subscribed analog raw value via three points, persists all calibration data, shows raw and calibrated value on the VT and provides them via OPC-UA.

logiBUS_AI_Calibrate_3P_Subscribe_VT_OPC



• — ○ • — ○ • — ○ •

Introduction

`logiBUS_AI_Calibrate_3P_Subscribe_VT_OPC` is the VT-side counterpart to the sensor station `logiBUS_AI_Calibrate_3P_IDA_OPC`: the SubApp subscribes via OPC-UA (M2M) to the **already calibrated** physical value of the 3-point calibration, shows it in the VT, re-publishes it locally and optionally provides it on the plug `CAL_OUT` for local consumers. The physical VT elements stay here (where the VT is): the keys `C_MIN`/`C_MID`/`C_MAX` call the corresponding methods on the sensor station directly via `CLIENT_0` (a genuine event-to-event method call, no toggle/publish/subscribe detour). The three reference value input fields `MIN_REF`/`MID_REF`/`MAX_REF` (`NumericValue_TO_AR2_OPC`) are still published **locally** and subscribed remotely by the sensor station; `MID_REF` and `MAX_REF` are additionally re-

exposed as plain AR plugs (MIDREF_OUT/MAXREF_OUT). Used by the 3-point calibration training sample InputOutputTesterButton_AI_Calibrate_3P_OPC_UA.

Interface Structure

Event Inputs

- **INIT** (EInit): initialization request

Event Outputs

- **INITO** (EInit): initialization confirmation (from MaxRef.INITO)

Data Inputs

- **QI** (BOOL, default TRUE): Quality Input - TRUE enables the services
- **ID_CAL_M2M_READ** (WSTRING): remote subscribe address of the already calibrated physical value (REAL), counterpart of ID_CAL_WRITE on the sensor station
- **stObjCal** (NumericObjectPool_S): object ID + scale/offset/decimals of the CNumberVariable for the calibrated value
- **ID_WRITE** (WSTRING): local re-publish address (REAL, ACTION=PUBLISH) for further local consumers
- **u16ObjId_C_MIN**, **u16ObjId_C_MID**, **u16ObjId_C_MAX** (UINT, default ID_NULL): object IDs of the keys C_MIN/C_MID/C_MAX on the VT
- **ID_MIN_CALL**, **ID_MID_CALL**, **ID_MAX_CALL** (WSTRING): remote method addresses (ACTION=CALL_METHOD) for the three calibration triggers on the sensor station
- **stObjMinRef**, **stObjMidRef**, **stObjMaxRef** (NumericObjectPool_S): object ID + scale/offset/decimals of the reference value input fields on this station
- **ID_MINREF_READ**, **ID_MIDREF_READ**, **ID_MAXREF_READ** (WSTRING): optional subscribe addresses if the reference value is to be overridden externally (dashboard); may be left unused
- **ID_MINREF_WRITE**, **ID_MIDREF_WRITE**, **ID_MAXREF_WRITE** (WSTRING): local publish addresses (REAL, ACTION=PUBLISH) of the reference values, subscribed remotely by the sensor station

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if all services succeeded (AND of AND_QO_A.OUT and ValueDisplay.QO)

Adapters

- **CAL_OUT** (plug, adapter::types::unidirectional::AR): optional, calibrated value (plain AR mirror of the remote subscribe) for external local consumers; leave unwired if not needed
- **MIDREF_OUT** (plug, adapter::types::unidirectional::AR): optional, live MID_REF reference value
- **MAXREF_OUT** (plug, adapter::types::unidirectional::AR): optional, live MAX_REF reference value

Function blocks used

- **AR_SUBSCRIBE_CAL** (adapter::net::AR_SUBSCRIBE_1): subscribes to the calibrated value from the sensor station (ID_CAL_M2M_READ).
- **AR_SPLIT_2_Cal** (adapter::events::unidirectional::AR_SPLIT_2): distributes the value to ValueDisplay and CAL_OUT.
- **ValueDisplay** (SubApp AR_TO_Q_Publish_OPC): VT display and local re-publish (stObjCal, ID_WRITE).
- **Button_C_MIN**, **Button_C_MID**, **Button_C_MAX** (isobus::UT::io::Button::Button_IE, InputEvent = BT_PRESSED_LATCHED): read the three VT keys.
- **CLIENT_MIN**, **CLIENT_MID**, **CLIENT_MAX** (iec61499::net::CLIENT_0): call the methods on the sensor station (ID_MIN_CALL/ID_MID_CALL/ID_MAX_CALL).
- **MinRef**, **MidRef**, **MaxRef** (SubApp NumericValue_TO_AR2_OPC): VT input fields of the reference values, published locally.
- **AND_QO_A** (iec61131::booleanOperators::AND_BOOL_10) and **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): combine the QO outputs of all sub-blocks into the QO of the SubApp.

Functionality

- **Calibrated value:** AR_SUBSCRIBE_CAL.OUT → AR_SPLIT_2_Cal.IN; AR_SPLIT_2_Cal.OUT1 → ValueDisplay.AR_IN; AR_SPLIT_2_Cal.OUT2 → CAL_OUT.

- **Calibration keys:** `Button_C_MIN.IND` ⟶ `CLIENT_MIN.REQ`, `Button_C_MID.IND` ⟶ `CLIENT_MID.REQ`, `Button_C_MAX.IND` ⟶ `CLIENT_MAX.REQ`.
- **Reference values:** `MinRef/MidRef/MaxRef` receive `stObjMinRef/stObjMidRef/stObjMaxRef`, `ID_*REF_READ` and `ID_*REF_WRITE`; `MidRef.VALUE_OUT` ⟶ `MIDREF_OUT`, `MaxRef.VALUE_OUT` ⟶ `MAXREF_OUT`.
- **Initialization chain:** `INIT` ⟶ `AR_SUBSCRIBE_CAL` ⟶ `ValueDisplay` ⟶ `Button_C_MIN` ⟶ `Button_C_MID` ⟶ `Button_C_MAX` ⟶ `CLIENT_MIN` ⟶ `CLIENT_MID` ⟶ `CLIENT_MAX` ⟶ `MinRef` ⟶ `MidRef` ⟶ `MaxRef` (each `INITO` ⟶ next `INIT`); `MaxRef.INITO` ⟶ `INITO` and ⟶ `AND_QO_A.REQ`; `AND_QO_A.CNF` ⟶ `AND_QO.REQ`.
- **Quality:** `QI` ⟶ `.QI` of `AR_SUBSCRIBE_CAL`, `Button_C_MIN/_MID/_MAX`, `CLIENT_MIN/_MID/_MAX`, `MinRef`, `MidRef`, `MaxRef` and `ValueDisplay`. `AR_SUBSCRIBE_CAL.QO`, the three `Button_C_*.QO`, the three `CLIENT_*.QO` and `MinRef.QO/MidRef.QO/MaxRef.QO` ⟶ `AND_QO_A.IN1..IN10`; `AND_QO_A.OUT` ⟶ `AND_QO.IN1`, `ValueDisplay.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

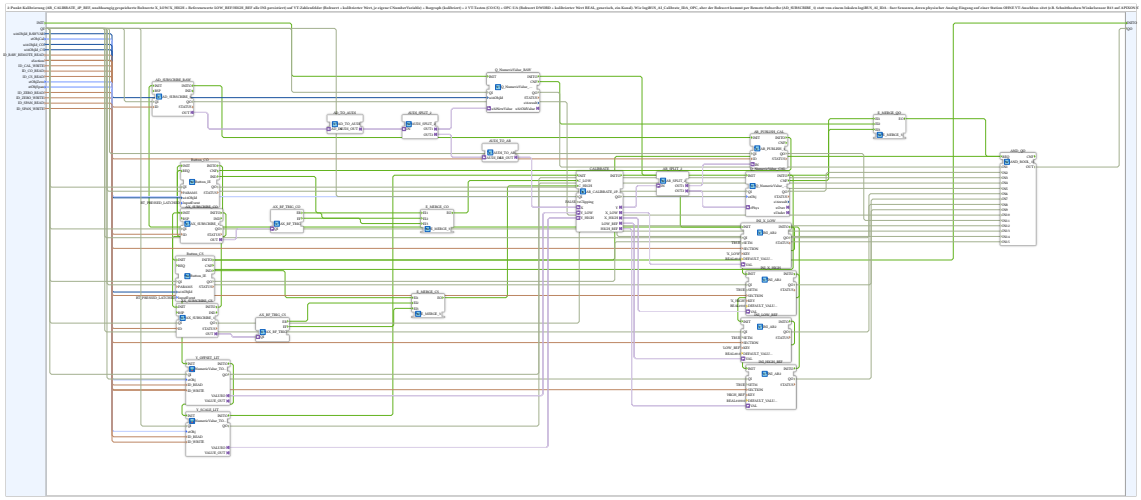
Summary

Remote display block for an already calibrated 3-point value: subscribe of the calibrated value, VT display with local re-publish, calibration keys as method calls to the sensor station, and locally published reference value input fields in one composite.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

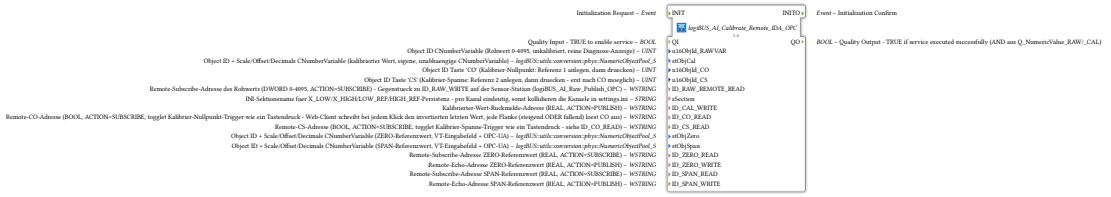
logiBUS_AI_Calibrate_Remote_IDA_OPC



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_AI_Calibrate_Remote_IDA_OPC` is a 2-point calibration (`AR_CALIBRATE_2P_REF`, independently stored raw values `X_LOW/X_HIGH` and reference values `LOW_REF/HIGH_REF`, all INI-persisted) with VT number fields (raw value and calibrated value, each its own `CNumberVariable`), two VT buttons (CO/CS) and OPC-UA (raw value `DWORD`, calibrated value `REAL`, generic, one channel). It corresponds to `logiBUS_AI_Calibrate_2P_IDA_OPC`, but the raw value arrives via **remote subscribe** (`AD_SUBSCRIBE_1`) instead of from a local `logiBUS_AI_IDA`. This is intended for sensors whose physical analog input sits on a station **without a VT connection** (e.g. cutting-height angle sensor B13 on APIXON STG3, while calibration and VT run on the station with the VT connection). The raw value is published there via `logiBUS_AI_Raw_Publish_OPC`.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirm

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, TRUE enables the service
- **u16ObjId_RAWVAR** (`UINT`): object ID of the `CNumberVariable` (raw value 0..4095, uncalibrated, pure diagnostic display), default `ID_NULL`
- **stObjCal** (`NumericObjectPool_S`): object ID plus scale/offset/decimals of the `CNumberVariable` for the calibrated value (its own, independent variable)
- **u16ObjId_CO** (`UINT`): object ID of button "CO" (calibration zero point: apply reference 1, then press)
- **u16ObjId_CS** (`UINT`): object ID of button "CS" (calibration span: apply reference 2, then press; only possible after CO)
- **ID_RAW_REMOTE_READ** (`WSTRING`): remote subscribe address of the raw value (`DWORD` 0..4095, `ACTION=SUBSCRIBE`), counterpart of `ID_RAW_WRITE` on the sensor station
- **sSection** (`STRING`): INI section name for the persistence of `X_LOW`/`X_HIGH`/`LOW_REF`/`HIGH_REF`; unique per channel, otherwise the channels collide in `settings.ini`
- **ID_CAL_WRITE** (`WSTRING`): feedback address of the calibrated value (`REAL`, `ACTION=PUBLISH`)
- **ID_CO_READ** (`WSTRING`): remote CO address (`BOOL`, `ACTION=SUBSCRIBE`): toggles the calibration zero point trigger like a button press; the web client writes the inverted last value on every click, every edge (rising or falling) triggers CO
- **ID_CS_READ** (`WSTRING`): remote CS address (`BOOL`, `ACTION=SUBSCRIBE`): toggles the calibration span trigger, see `ID_CO_READ`
- **stObjZero** (`NumericObjectPool_S`): `CNumberVariable` of the ZERO reference value (VT input field + OPC-UA)
- **stObjSpan** (`NumericObjectPool_S`): `CNumberVariable` of the SPAN reference value (VT input field + OPC-UA)
- **ID_ZERO_READ** (`WSTRING`): remote subscribe address ZERO reference value (`REAL`, `ACTION=SUBSCRIBE`)
- **ID_ZERO_WRITE** (`WSTRING`): remote echo address ZERO reference value (`REAL`, `ACTION=PUBLISH`)
- **ID_SPAN_READ** (`WSTRING`): remote subscribe address SPAN reference value (`REAL`, `ACTION=SUBSCRIBE`)
- **ID_SPAN_WRITE** (`WSTRING`): remote echo address SPAN reference value (`REAL`, `ACTION=PUBLISH`)

Data Outputs

- **QO** (`BOOL`): quality output, AND combination (`AND_QO`, a single `AND_BOOL_15`) of the `QO` outputs of all 15 inner blocks with `QO`: `Q_NumericValue_RAW`, `Q_NumericValue_CAL`, `Y_OFFSET_LIT`, `Y_SCALE_LIT`, `CALIBRATE`, `INI_X_LOW`, `INI_X_HIGH`, `INI_LOW_REF`, `INI_HIGH_REF`, `AD_SUBSCRIBE_RAW`, `AR_PUBLISH_CAL`, `AX_SUBSCRIBE_CO`, `AX_SUBSCRIBE_CS`, `Button_CO`, `Button_CS`

Adapters

None.

Function Blocks (FBs) Used

- **`AD_SUBSCRIBE_RAW`** (`adapter::net::AD_SUBSCRIBE_1`): subscribes to the remote raw value
- **`AD_TO_AUDI`**, **`AUDI_SPLIT_2`**, **`AUDI_TO_AR`**: conversion and splitting of the raw value
- **`Q_NumericValue_RAW`** (`isobus::UT::Q::Q_NumericValue_AUDI`): VT raw value display
- **`CALIBRATE`** (`adapter::Engineering::measurements::AR_CALIBRATE_2P_REF`, `xClipping = FALSE`): 2-point calibration
- **`Y_OFFSET_LIT`**, **`Y_SCALE_LIT`** (SubApp `MyLib::sys::NumericValue_TO_AR2_OPC`, NumericValue_TO_AR2_OPC): ZERO/SPAN reference values as `AR2`
- **`Button_CO`**, **`Button_CS`** (`isobus::UT::io::Button::Button_IE`, `InputEvent = BT_PRESSED_LATCHED`): VT buttons
- **`AX_SUBSCRIBE_CO`**, **`AX_SUBSCRIBE_CS`** (`adapter::net::AX_SUBSCRIBE_1`): remote triggers

- **AX_RF_TRIG_CO**, **AX_RF_TRIG_CS** (adapter::events::unidirectional::AX_RF_TRIG): detect rising and falling edge
- **E_MERGE_CO**, **E_MERGE_CS** (iec61499::events::E_MERGE_3): merge button, rising and falling edge
- **INI_X_LOW**, **INI_X_HIGH**, **INI_LOW_REF**, **INI_HIGH_REF** (eclipse4diac::storage::INI_AR2, SETM = TRUE): persistence; keys X_LOW/X_HIGH/LOW_REF/HIGH_REF, defaults 0.0/0.0/0.0/100.0
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_15, since v1.5): replaces the former cascade AND_QO_A (AND_BOOL_4) + AND_QO_B (AND_BOOL_10) + AND_QO (AND_BOOL_3) with a single block for all 15 QO sources
- **AR_SPLIT_2** (adapter::events::unidirectional::AR_SPLIT_2), **AR_PUBLISH_CAL** (adapter::net::AR_PUBLISH_1), **Q_NumericValue_CAL** (isobus::UT::Q::Q_NumericValue_PHYSA): output of the calibrated value

Functionality

- Raw value path: AD_SUBSCRIBE_RAW.OUT → AD_TO_AUDI.AD_IN → AUDI_SPLIT_2.IN; AUDI_SPLIT_2.OUT1 → Q_NumericValue_RAW.u32NewValue (VT display); AUDI_SPLIT_2.OUT2 → AUDI_TO_AR.AUDI_IN → CALIBRATE.X
- Persistence: CALIBRATE.X_LOW/X_HIGH/LOW_REF/HIGH_REF ↔ INI_X_LOW/INI_X_HIGH/INI_LOW_REF/INI_HIGH_REF.VAL; sSection → SECTION of all four
- References: Y_OFFSET_LIT.VALUEO → CALIBRATE.Y_LOW; Y_SCALE_LIT.VALUEO → CALIBRATE.Y_HIGH
- Calibrated value: CALIBRATE.Y → AR_SPLIT_2.IN; OUT1 → AR_PUBLISH_CAL.IN; OUT2 → Q_NumericValue_CAL.rPhys
- CO trigger: Button_CO.IND → E_MERGE_CO.EI1; AX_SUBSCRIBE_CO.OUT → AX_RF_TRIG_CO.QI; AX_RF_TRIG_CO.ER → E_MERGE_CO.EI2, AX_RF_TRIG_CO.EF → E_MERGE_CO.EI3; E_MERGE_CO.EO → CALIBRATE.C_LOW
- CS trigger: analogous with Button_CS, AX_SUBSCRIBE_CS, AX_RF_TRIG_CS, E_MERGE_CS → CALIBRATE.C_HIGH
- Initialisation chain: INIT → Q_NumericValue_RAW.INIT → Q_NumericValue_CAL.INIT → Y_OFFSET_LIT.INIT → Y_SCALE_LIT.INIT → CALIBRATE.INIT → INI_X_LOW.INIT → INI_X_HIGH.INIT → INI_LOW_REF.INIT → INI_HIGH_REF.INIT → AD_SUBSCRIBE_RAW.INIT → AR_PUBLISH_CAL.INIT → AX_SUBSCRIBE_CO.INIT → AX_SUBSCRIBE_CS.INIT → Button_CO.INIT → Button_CS.INIT; Button_CS.INITO → INITO and → AND_QO.REQ. In addition E_MERGE_QO (E_MERGE_3) merges Q_NumericValue_CAL.INITO and the CNF events of Q_NumericValue_RAW and Q_NumericValue_CAL; its output EO triggers AND_QO.REQ, so QO is re-evaluated on every new value (from v1.6; previously EO was not connected). QI is distributed to all inner blocks; their QO outputs feed AND_QO.IN1 to AND_QO.IN15 in the order of the chain, AND_QO.OUT → QO. (The former partial chain AD_SUBSCRIBE_RAW.INITO → AR_PUBLISH_CAL.INIT → AX_SUBSCRIBE_CO.INIT → AX_SUBSCRIBE_CS.INIT is part of this chain.)

Technical Details

- **Raw value via remote subscribe:** logiBUS_AI_IDA, AD_SPLIT_2 and AD_PUBLISH_RAW of the local variant are replaced by AD_SUBSCRIBE_1, the rest of the calibration and VT chain is unchanged.
- **CO/CS without AX sockets:** since v1.1 C_LOW/C_HIGH of AR_CALIBRATE_2P_REF are pure event inputs; the E_MERGE_3 blocks connect directly via event connections (v1.2: one E_MERGE_3 instead of two E_MERGE_2 in series, functionally identical).
- **QI on CALIBRATE:** since v1.3 AR_CALIBRATE_2P_REF has an INIT/QI/QO enable pattern; without QI = TRUE the block would stay in state START and REQ would never run. The QI of all inner blocks now comes from the subapp input QI (default TRUE) instead of fixed parameters.
- **Bugfix v1.4 (code review PR #260):** INI_X_HIGH.DEFAULT_VALUE was REAL#1.0 instead of 0.0; as a result the calibrated flag xHighCal latched TRUE immediately at every boot although HIGH was never calibrated. Affects cutting height, oil temperature and pressure.
- **Web trigger:** CO/CS are triggered by every edge (rising or falling), because the web client writes the inverted last value on every click.
- This type is being superseded by logiBUS_AI_Calibrated_Subscribe_VT_OPC (local analog processing); the adjustments keep the file compilable.

Application Scenarios

- Calibrating a sensor whose analog input sits on a station without a VT connection, with VT operation on another station

⚖️ Comparison with Similar Building Blocks

- `logiBUS_AI_Calibrate_2P_IDA_OPC`: raw value from a local `logiBUS_AI_IDA`
- `logiBUS_AI_Calibrate_3P_Remote_IDA_OPC`: three-point variant with remote raw value

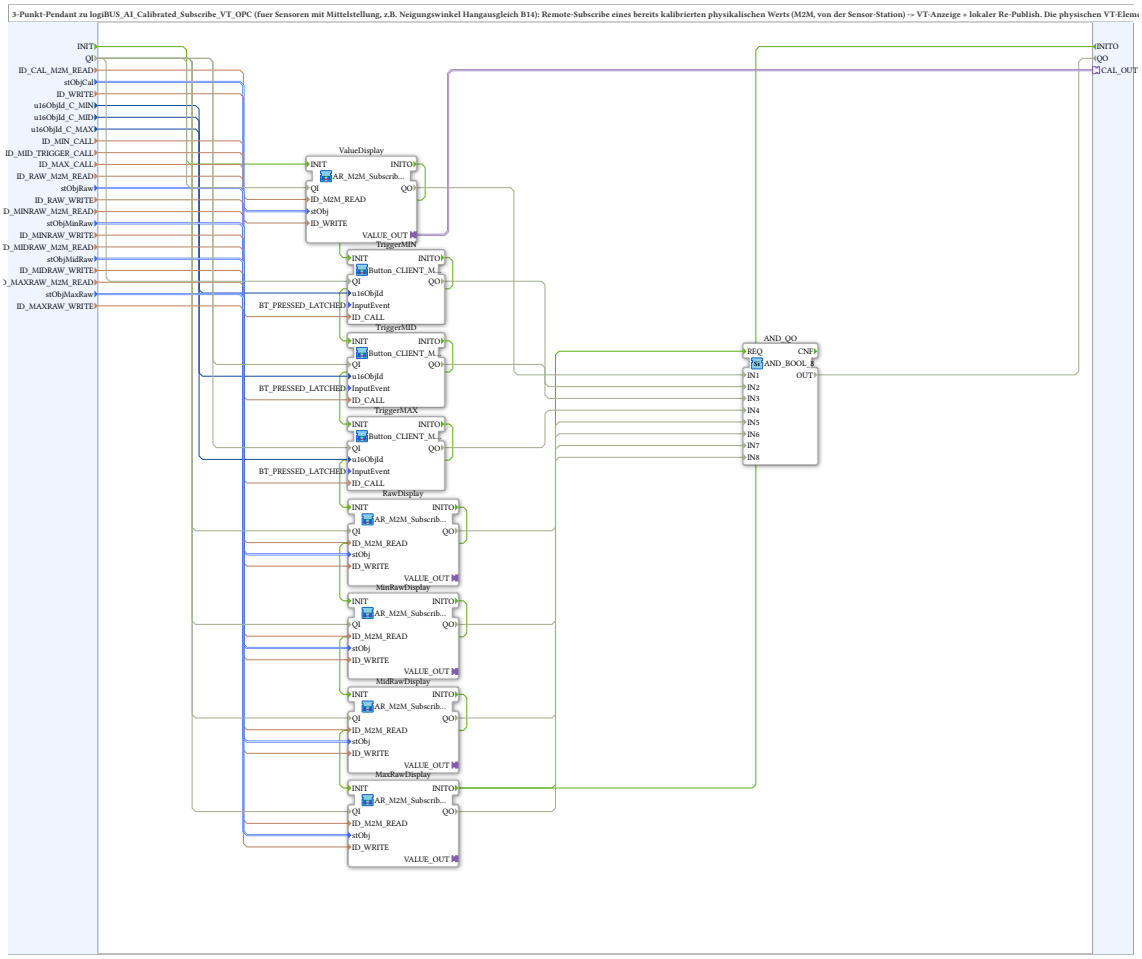
Summary

`logiBUS_AI_Calibrate_Remote_IDA_OPC` calibrates a remotely published analog raw value in two points, shows raw and calibrated value on the VT and publishes the calibrated value via OPC-UA.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

logiBUS_AI_Calibrated_Subscribe_VT_3P_OPC

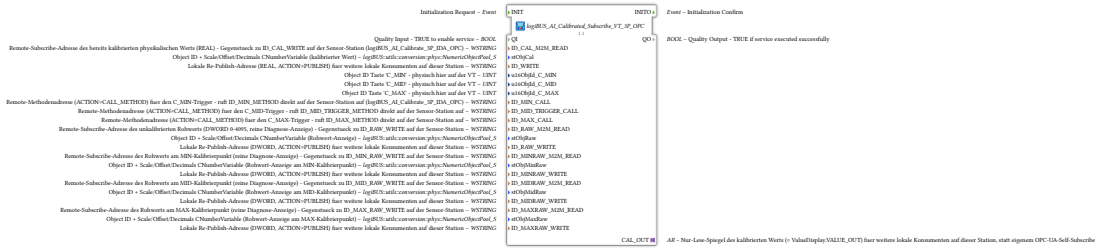


• — ○ • — ○ • — ○ •

Introduction

`logiBUS_AI_Calibrated_Subscribe_VT_3P_OPC` is the **3-point counterpart** of `logiBUS_AI_Calibrated_Subscribe_VT_OPC` for sensors with a centre position (e.g. slope compensation tilt angle B14). Via remote subscribe (M2M, from the sensor station) it subscribes to an **already calibrated physical value**, shows it on the VT and re-publishes it locally.

The physical VT elements stay where the VT is: the keys **C_MIN/C_MID/C_MAX** call the methods `ID_MIN_METHOD/ID_MID_TRIGGER_METHOD/ID_MAX_METHOD` directly on the sensor station via `CLIENT_0` (`logiBUS_AI_Calibrate_3P_IDA_OPC`) - a true event-to-event method call, no toggle/publish/subscribe detour (see `OPC_UA_VERNETZUNG.md`, item 14). The reference value input fields `MIN_REF/MID_REF/MAX_REF` deliberately stay outside this SubApp (separate `NumericValue_TO_CLIENT_1_0_OPC` instances per caller).



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization Request

Event Outputs

- **INITO** (`EInit`): Initialization Confirm

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE to enable the service

Calibrated value:

- **ID_CAL_M2M_READ** (`WSTRING`): Remote subscribe address of the already calibrated physical value (`REAL`); counterpart of `ID_CAL_WRITE` on the sensor station (`logiBUS_AI_Calibrate_3P_IDA_OPC`)
- **stObjCal** (`NumericObjectPool_S`, default `u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 1`): Object ID + scale/offset/decimals of the `CNumberVariable` (calibrated value)
- **ID_WRITE** (`WSTRING`): Local re-publish address (`REAL`, `ACTION=PUBLISH`) for further local consumers on this station

Keys C_MIN/C_MID/C_MAX:

- **u16ObjId_C_MIN**, **u16ObjId_C_MID**, **u16ObjId_C_MAX** (`UINT`, default `ID_NULL`): Object IDs of the keys “C_MIN”, “C_MID”, “C_MAX”, physically here on the VT
- **ID_MIN_CALL**, **ID_MID_TRIGGER_CALL**, **ID_MAX_CALL** (`WSTRING`): Remote method addresses (`ACTION=CALL_METHOD`); call `ID_MIN_METHOD`, `ID_MID_TRIGGER_METHOD` and `ID_MAX_METHOD` respectively directly on the sensor station

Raw value displays (pure diagnostics):

- **ID_RAW_M2M_READ** (`WSTRING`): Remote subscribe address of the uncalibrated raw value (`DWORD` 0-4095); counterpart of `ID_RAW_WRITE` on the sensor station
- **stObjRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Raw value display field
- **ID_RAW_WRITE** (`WSTRING`): Local re-publish address (`DWORD`, `ACTION=PUBLISH`)
- **ID_MINRAW_M2M_READ** (`WSTRING`): Remote subscribe address of the raw value at the MIN calibration point; counterpart of `ID_MIN_RAW_WRITE` on the sensor station
- **stObjMinRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Display field for the raw value at the MIN calibration point
- **ID_MINRAW_WRITE** (`WSTRING`): Local re-publish address (`DWORD`, `ACTION=PUBLISH`)
- **ID_MIDRAW_M2M_READ** (`WSTRING`): Remote subscribe address of the raw value at the MID calibration point; counterpart of `ID_MID_RAW_WRITE` on the sensor station
- **stObjMidRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Display field for the raw value at the MID calibration point
- **ID_MIDRAW_WRITE** (`WSTRING`): Local re-publish address (`DWORD`, `ACTION=PUBLISH`)
- **ID_MAXRAW_M2M_READ** (`WSTRING`): Remote subscribe address of the raw value at the MAX calibration point; counterpart of `ID_MAX_RAW_WRITE` on the sensor station
- **stObjMaxRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Display field for the raw value at the MAX calibration point
- **ID_MAXRAW_WRITE** (`WSTRING`): Local re-publish address (`DWORD`, `ACTION=PUBLISH`)

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if the service executed successfully (AND of the QO of ValueDisplay, TriggerMIN, TriggerMID, TriggerMAX, RawDisplay, MinRawDisplay, MidRawDisplay and MaxRawDisplay)

Adapters

- **CAL_OUT** (plug, adapter::types::unidirectional::AR): Read-only mirror of the calibrated value (ValueDisplay.VALUE_OUT) for further local consumers on this station (e.g. Fahrwerk_Mittelpunkt_Anzeige for bar graph/status centre), instead of its own OPC-UA self-subscribe

Function Blocks (FBs) Used

- **ValueDisplay**: SubApp MyLib::sys::AR_M2M_Subscribe_TO_Q_Publish_OPC - calibrated value
- **RawDisplay**, **MinRawDisplay**, **MidRawDisplay**, **MaxRawDisplay**: the same SubApp for the raw value and the raw values of the three calibration points
- **TriggerMIN**, **TriggerMID**, **TriggerMAX**: SubApp MyLib::sys::Button_CLIENT_Method_IE (InputEvent=BT_PRESSED_LATCHED) - key ⟶ remote method call (ID_CALL)
- **AND_QO**: iec61131::booleanOperators::AND_BOOL_8 - ANDs the QO outputs of the eight inner SubApps into the block output QO
- **Imports**: ID_NULL, BT_PRESSED_LATCHED, NumericObjectPool_S

Program Flow and Connections

- ID_CAL_M2M_READ, stObjCal, ID_WRITE ⟶ ValueDisplay (ID_M2M_READ, stObj, ID_WRITE); ValueDisplay.VALUE_OUT ⟶ CAL_OUT
- u16ObjId_C_MIN/ID_MIN_CALL ⟶ TriggerMIN (u16ObjId/ID_CALL); likewise C_MID ⟶ TriggerMID, C_MAX ⟶ TriggerMAX
- ID_RAW_M2M_READ, stObjRaw, ID_RAW_WRITE ⟶ RawDisplay
- ID_MINRAW_M2M_READ, stObjMinRaw, ID_MINRAW_WRITE ⟶ MinRawDisplay
- ID_MIDRAW_M2M_READ, stObjMidRaw, ID_MIDRAW_WRITE ⟶ MidRawDisplay
- ID_MAXRAW_M2M_READ, stObjMaxRaw, ID_MAXRAW_WRITE ⟶ MaxRawDisplay
- Initialization chain: INIT ⟶ ValueDisplay.INIT, ValueDisplay.INITO ⟶ TriggerMIN.INIT, TriggerMIN.INITO ⟶ TriggerMID.INIT, TriggerMID.INITO ⟶ TriggerMAX.INIT, TriggerMAX.INITO ⟶ RawDisplay.INIT, RawDisplay.INITO ⟶ MinRawDisplay.INIT, MinRawDisplay.INITO ⟶ MidRawDisplay.INIT, MidRawDisplay.INITO ⟶ MaxRawDisplay.INIT, MaxRawDisplay.INITO ⟶ INITO and ⟶ AND_QO.REQ
- QI ⟶ QI of all eight inner SubApps
- ValueDisplay.QO, TriggerMIN.QO, TriggerMID.QO, TriggerMAX.QO, RawDisplay.QO, MinRawDisplay.QO, MidRawDisplay.QO, MaxRawDisplay.QO ⟶ AND_QO.IN1..IN8; AND_QO.OUT ⟶ QO

Technical Details

- The calibration itself runs on the sensor station; this block contains no calibration logic
- Version 1.1 (2026-09-26): plug CAL_OUT added - mirrors the already remotely subscribed value for further consumers on STG1, no additional self-subscribe needed
- Version 1.0 (2026-09-26): 3-point counterpart of logiBUS_AI_Calibrated_Subscribe_VT_OPC v1.8, for slope compensation angle B14 (calibration moves from STG1 to STG3)

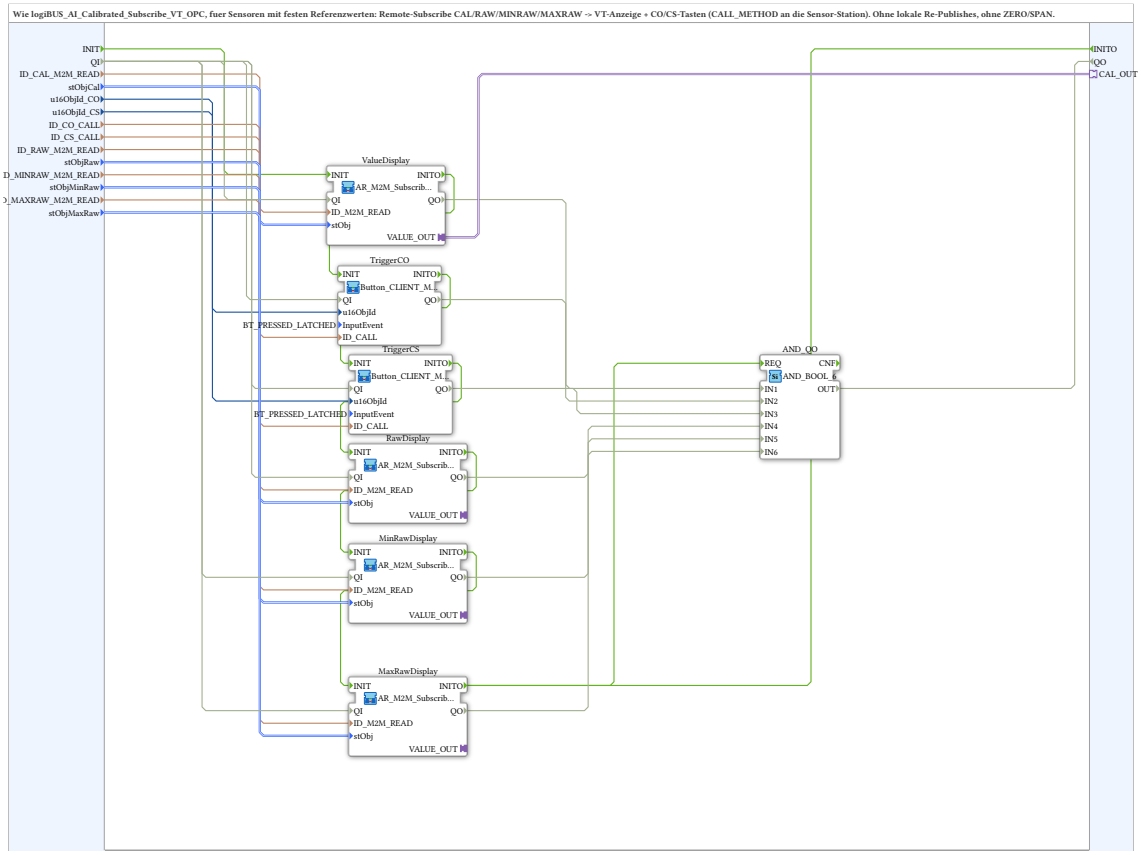
Application Scenarios

- VT station that displays the calibrated tilt angle (B14) of another station and triggers the 3-point calibration (MIN/MID/MAX) via keys

Summary

logiBUS_AI_Calibrated_Subscribe_VT_3P_OPC shows remotely calibrated values and raw values of the calibration points on the VT and forwards the keys C_MIN/C_MID/C_MAX to the sensor station via OPC-UA method calls.

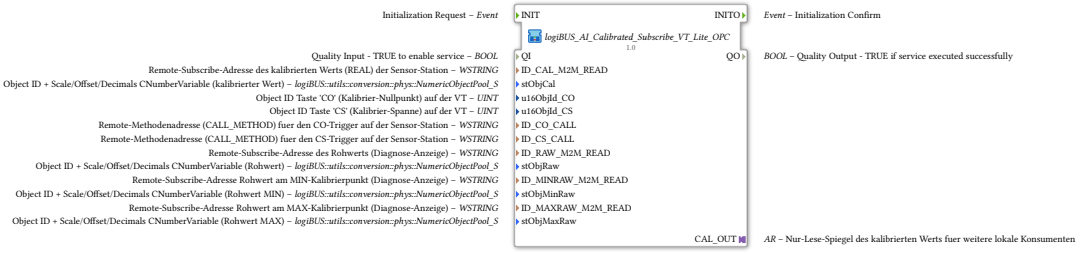
logiBUS_AI_Calibrated_Subscribe_VT_Lite_OPC



• — ○ • — ○ • — ○ •

Introduction

`logiBUS_AI_Calibrated_Subscribe_VT_Lite_OPC` is the lean variant of `logiBUS_AI_Calibrated_Subscribe_VT_OPC` for sensors with fixed reference values. The calibrated value, the raw value and the raw values at the MIN and MAX calibration points are subscribed remotely from the sensor station and shown on the VT; the keys **CO**/**CS** call the calibration methods on the sensor station via `CALL_METHOD`. Compared with the original there are no local re-publishes (no `ID_*_WRITE` mirrors) and no `ZERO`/`SPAN` reference value input fields (`ZeroRef`/`SpanRef` are omitted, the reference values are fixed on the sensor station). Instead of `AR_M2M_Subscribe_TO_Q_Publish_OPC`, `AR_M2M_Subscribe_TO_Q_OPC` is used four times.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the chain `ValueDisplay` ⟶ `TriggerCO` ⟶ `TriggerCS` ⟶ `RawDisplay` ⟶ `MinRawDisplay` ⟶ `MaxRawDisplay`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `MaxRawDisplay.INITO`

Data Inputs

Variable	Type	Default	Comment
QI	BOOL	TRUE	Quality input: enables the service; passed on to all six instances.
ID_CAL_M2M_READ	WSTRING		Remote subscribe address of the calibrated value (REAL) of the sensor station.
stObjIdCal	utils::conversion::phys::NumericObjectID	iQI := TRUE, bID_NULL := FALSE, r32Scale := 1.0, i32Offset := 0, u8Decimals := 1	Object ID plus scale/offset/decimals of the CNumberVariable (calibrated value).
u16ObjId_CO	UINT	ID_NULL	Object ID of the key “CO” (calibration zero point) on the VT.
u16ObjId_CS	UINT	ID_NULL	Object ID of the key “CS” (calibration span) on the VT.
ID_CO_CALL	WSTRING		Remote method address (CALL_METHOD) for the CO trigger on the sensor station.
ID_CS_CALL	WSTRING		Remote method address (CALL_METHOD) for the CS trigger on the sensor station.
ID_RAW_M2M_READ	WSTRING		Remote subscribe address of the raw value (diagnostic display).
stObjIdRaw	utils::conversion::phys::NumericObjectID	iQI := TRUE, bID_NULL := FALSE, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0	Object ID plus scale/offset/decimals of the CNumberVariable (raw value).
ID_MINRAW_M2M_READ	WSTRING		Remote subscribe address of the raw value at the MIN calibration point (diagnostic display).
stObjIdMinRaw	utils::conversion::phys::NumericObjectID	iQI := TRUE, bID_NULL := FALSE, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0	Object ID plus scale/offset/decimals of the CNumberVariable (raw value MIN).
ID_MAXRAW_M2M_READ	WSTRING		Remote subscribe address of the raw value at the MAX calibration point (diagnostic display).
stObjIdMaxRaw	utils::conversion::phys::NumericObjectID	iQI := TRUE, bID_NULL := FALSE, r32Scale :=	Object ID plus scale/offset/decimals of the

		`1.0, i32Offset := 0, u8Decimals := 0)`	`CNumberVariable` (raw value MAX).

Data Outputs

Variable	Type	Default	Comment
`QO`	BOOL		Quality output: `AND` of the `QO` outputs of all six instances (via `AND_QO`, `AND_BOOL_6`).

Adapters

Adapter	Type	Direction	Comment
`CAL_OUT`	`adapter::types::unidirectional::AR`	Plug	Read-only mirror of the calibrated value (`ValueDisplay.VALUE_OUT`) for further local consumers.

Function Blocks (FBs) Used

- **ValueDisplay** (SubApp `MyLib::sys::AR_M2M_Subscribe_TO_Q_OPC`): calibrated value
- **TriggerCO**, **TriggerCS** (SubApp `MyLib::sys::Button_CLIENT_Method_IE`, `InputEvent = BT_PRESSED_LATCHED`): keys CO and CS with method call
- **RawDisplay**, **MinRawDisplay**, **MaxRawDisplay** (SubApp `MyLib::sys::AR_M2M_Subscribe_TO_Q_OPC`): raw value and raw values of the MIN and MAX calibration points
- **AND_QO** (`iec61131::booleanOperators::AND_BOOL_6`): combines the six `QO` outputs

Program Flow and Connections

1. Value: `ID_CAL_M2M_READ -> ValueDisplay.ID_M2M_READ`; `stObjCal -> ValueDisplay.stObj`; `ValueDisplay.VALUE_OUT -> CAL_OUT`.
2. Keys: `u16ObjId_CO -> TriggerCO.u16ObjId`; `ID_CO_CALL -> TriggerCO.ID_CALL`; `u16ObjId_CS -> TriggerCS.u16ObjId`; `ID_CS_CALL -> TriggerCS.ID_CALL`.
3. Raw value: `ID_RAW_M2M_READ -> RawDisplay.ID_M2M_READ`; `stObjRaw -> RawDisplay.stObj`.
4. MIN raw value: `ID_MINRAW_M2M_READ -> MinRawDisplay.ID_M2M_READ`; `stObjMinRaw -> MinRawDisplay.stObj`.
5. MAX raw value: `ID_MAXRAW_M2M_READ -> MaxRawDisplay.ID_M2M_READ`; `stObjMaxRaw -> MaxRawDisplay.stObj`.
6. Initialization: `INIT -> ValueDisplay.INIT -> TriggerCO.INIT -> TriggerCS.INIT -> RawDisplay.INIT -> MinRawDisplay.INIT -> MaxRawDisplay.INIT` (each via `INITO`); `MaxRawDisplay.INITO -> INITO` and `-> AND_QO.REQ`.
7. Qualifier: `QI -> .QI` of the six instances.
8. Quality: `ValueDisplay.QO -> AND_QO.IN1`; `TriggerCO.QO -> IN2`; `TriggerCS.QO -> IN3`; `RawDisplay.QO -> IN4`; `MinRawDisplay.QO -> IN5`; `MaxRawDisplay.QO -> IN6`; `AND_QO.OUT -> QO`.

Technical Features

- Without local re-publishes: there are no `ID_*_WRITE` inputs.
- Without `ZERO`/`SPAN`: the reference values are fixed on the sensor station (e.g. filling belt angle).
- The calibration itself runs on the sensor station; this block contains no calibration logic.

Application Scenarios

- VT station that shows calibrated sensor values of another station and triggers their calibration by key, when the sensor has fixed reference values (e.g. filling belt angle) and no local re-publishes are needed.

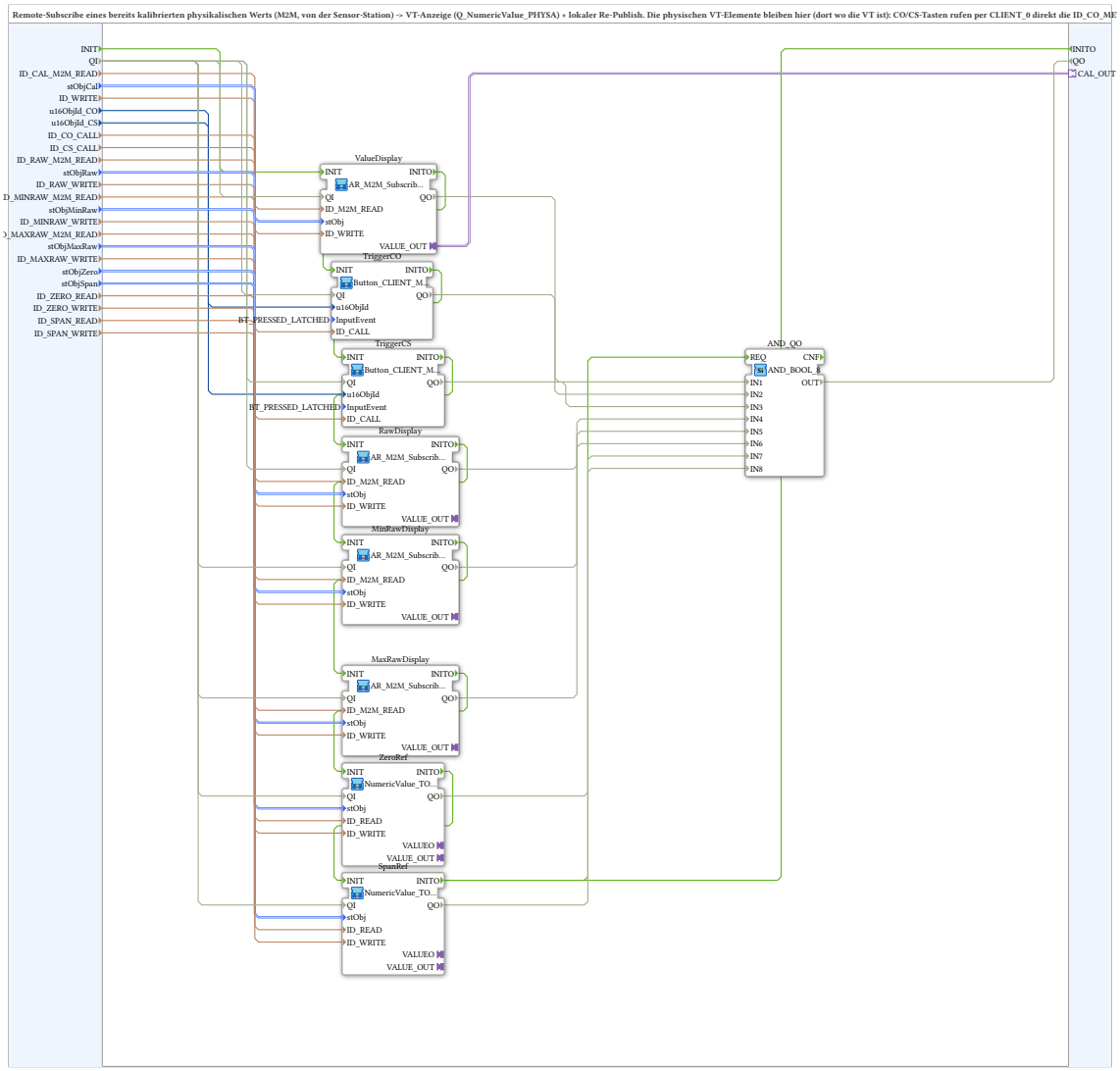
Summary

`logiBUS_AI_Calibrated_Subscribe_VT_Lite_OPC` shows remotely calibrated values and raw values on the VT and forwards the calibration keys to the sensor station via method call - without local re-publishes and without `ZERO`/`SPAN`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

logiBUS_AI_Calibrated_Subscribe_VT_OPC

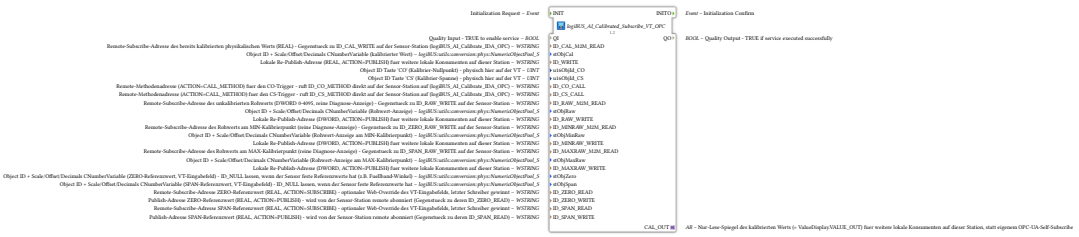


• — ○ • — ○ • — ○ •

Introduction

`logiBUS_AI_Calibrated_Subscribe_VT_OPC` subscribes via remote subscribe (M2M, from the sensor station) to an **already calibrated physical value** and shows it on the VT (`Q_NumericValue_PHYSA`) with a local re-publish. The physical VT elements stay where the VT is: the keys **CO**/**CS** call the methods `ID_CO_METHOD`/`ID_CS_METHOD` directly on the sensor station (`logiBUS_AI_Calibrate_IDA_OPC`) via `CLIENT_0` - a genuine event-to-event method call, no toggle/publish/subscribe detour (see `OPC_UA_VERNETZUNG.md`, item 14).

The block replaces `logiBUS_AI_Calibrate_Remote_IDA_OPC`: the calibration itself now runs on the sensor station, only the display and the forwarding of the VT controls remain here. The `ZERO`/`SPAN` reference value input fields (`stObjZero`/`stObjSpan`) are optional: leave `ID_NULL` if the sensor has fixed reference values (filling belt angle: 0/100 as a literal on the sensor station), pass real values through if the operator enters them (pressure: 0/250 bar, no calibrator always at hand).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; starts the chain `ValueDisplay` ⟶ `TriggerCO` ⟶ `TriggerCS` ⟶ `RawDisplay` ⟶ `MinRawDisplay` ⟶ `MaxRawDisplay` ⟶ `ZeroRef` ⟶ `SpanRef`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `SpanRef.INITO`

Data Inputs

Enable:

- **QI** (`BOOL`, default `TRUE`): quality input, service enable; passed on to all eight instances (`ValueDisplay`, `TriggerCO`, `TriggerCS`, `RawDisplay`, `MinRawDisplay`, `MaxRawDisplay`, `ZeroRef`, `SpanRef`)

Calibrated value:

- **ID_CAL_M2M_READ** (`WSTRING`): Remote subscribe address of the already calibrated physical value (`REAL`); counterpart of `ID_CAL_WRITE` on the sensor station
- **stObjCal** (`NumericObjectPool_S`): Object ID + scale/offset/decimals of the `CNumberVariable` (calibrated value)
- **ID_WRITE** (`WSTRING`): Local re-publish address (`REAL`, `ACTION=PUBLISH`) for further local consumers

Keys CO/CS:

- **u16ObjId_CO** (`UINT`, default `ID_NULL`): Object ID of the key “CO” (calibration zero point), physically here on the VT
- **u16ObjId_CS** (`UINT`, default `ID_NULL`): Object ID of the key “CS” (calibration span), physically here on the VT
- **ID_CO_CALL**, **ID_CS_CALL** (`WSTRING`): Remote method addresses (`ACTION=CALL_METHOD`); call `ID_CO_METHOD`/`ID_CS_METHOD` directly on the sensor station

Raw value displays (pure diagnostics):

- **ID_RAW_M2M_READ** (`WSTRING`): Remote subscribe address of the uncalibrated raw value (`DWORD` 0-4095); counterpart of `ID_RAW_WRITE` on the sensor station
- **stObjRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Display field raw value
- **ID_RAW_WRITE** (`WSTRING`): Local re-publish address (`DWORD`, `ACTION=PUBLISH`)
- **ID_MINRAW_M2M_READ** (`WSTRING`): Remote subscribe address of the raw value at the MIN calibration point; counterpart of `ID_ZERO_RAW_WRITE` on the sensor station
- **stObjMinRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Display field raw value at the MIN calibration point
- **ID_MINRAW_WRITE** (`WSTRING`): Local re-publish address (`DWORD`, `ACTION=PUBLISH`)
- **ID_MAXRAW_M2M_READ** (`WSTRING`): Remote subscribe address of the raw value at the MAX calibration point; counterpart of `ID_SPAN_RAW_WRITE` on the sensor station
- **stObjMaxRaw** (`NumericObjectPool_S`, `u8Decimals := 0`): Display field raw value at the MAX calibration point
- **ID_MAXRAW_WRITE** (`WSTRING`): Local re-publish address (`DWORD`, `ACTION=PUBLISH`)

Reference values (optional):

- **stObjZero**, **stObjSpan** (`NumericObjectPool_S`): VT input fields for the ZERO/SPAN reference value; leave `ID_NULL` if the sensor has fixed reference values
- **ID_ZERO_READ**, **ID_SPAN_READ** (`WSTRING`): Remote subscribe addresses (`REAL`, `ACTION=SUBSCRIBE`) - optional web override of the VT input field, last writer wins

- **ID_ZERO_WRITE**, **ID_SPAN_WRITE** (WSTRING): Publish addresses (REAL, ACTION=PUBLISH), subscribed remotely by the sensor station (counterpart of its ID_ZERO_READ/ID_SPAN_READ)

Data Outputs

- **QO** (BOOL): quality output, AND of the QO outputs of all eight instances (via AND_QO, AND_BOOL_8)

Adapters

- **CAL_OUT** (plug, adapter::types::unidirectional::AR): Read-only mirror of the calibrated value (ValueDisplay.VALUE_OUT) for further local consumers on this station, instead of its own OPC-UA self-subscribe

Function Blocks (FBs) Used

- **ValueDisplay**: SubApp MyLib::sys::AR_M2M_Subscribe_TO_Q_Publish_OPC - calibrated value
- **RawDisplay**, **MinRawDisplay**, **MaxRawDisplay**: same SubApp for the raw value and the raw values of the calibration points
- **TriggerCO**, **TriggerCS**: SubApp MyLib::sys::Button_CLIENT_Method_IE (InputEvent=BT_PRESSED_LATCHED)
- **ZeroRef**, **SpanRef**: SubApp MyLib::sys::NumericValue_TO_AR2_OPC - reference value input fields
- **AND_QO**: iec61131::booleanOperators::AND_BOOL_8 - combines the eight QO outputs into the SubApp output QO

Program Flow and Connections

- ID_CAL_M2M_READ, stObjCal, ID_WRITE → ValueDisplay; ValueDisplay.VALUE_OUT → CAL_OUT
- u16ObjId_CO/ID_CO_CALL → TriggerCO; u16ObjId_CS/ID_CS_CALL → TriggerCS
- ID_RAW_M2M_READ, stObjRaw, ID_RAW_WRITE → RawDisplay
- ID_MINRAW_M2M_READ, stObjMinRaw, ID_MINRAW_WRITE → MinRawDisplay
- ID_MAXRAW_M2M_READ, stObjMaxRaw, ID_MAXRAW_WRITE → MaxRawDisplay
- stObjZero, ID_ZERO_READ, ID_ZERO_WRITE → ZeroRef
- stObjSpan, ID_SPAN_READ, ID_SPAN_WRITE → SpanRef
- Initialization: INIT → ValueDisplay.INIT → TriggerCO.INIT → TriggerCS.INIT → RawDisplay.INIT → MinRawDisplay.INIT → MaxRawDisplay.INIT → ZeroRef.INIT → SpanRef.INIT (each via INITO); SpanRef.INITO → INITO and → AND_QO.REQ
- Quality: QI → .QI of the eight instances; ValueDisplay.QO → AND_QO.IN1, TriggerCO.QO → IN2, TriggerCS.QO → IN3, RawDisplay.QO → IN4, MinRawDisplay.QO → IN5, MaxRawDisplay.QO → IN6, ZeroRef.QO → IN7, SpanRef.QO → IN8; AND_QO.OUT → QO

Technical Details

- The button + CLIENT_0 call formerly occurred twice identically here (CO and CS) and was factored out into Button_CLIENT_Method_IE.
- The calibration itself runs on the sensor station; this block contains no calibration logic.

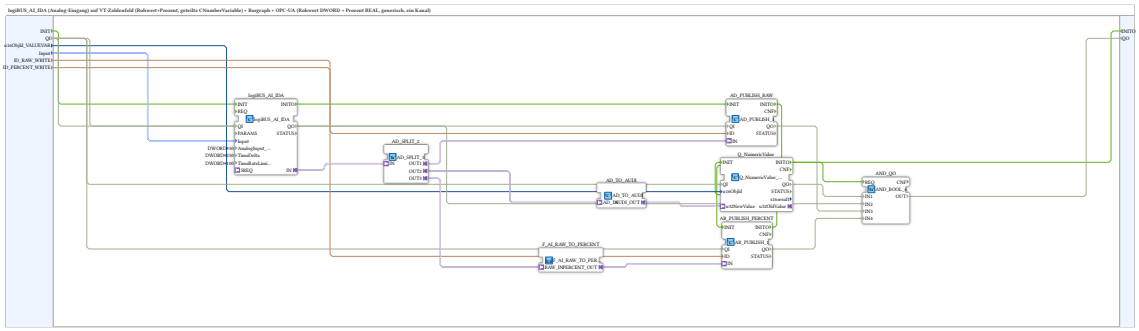
Application Scenarios

- VT station that displays calibrated sensor values of another station and triggers its calibration via key (e.g. filling belt angle, pressure)

Summary

logiBUS_AI_Calibrated_Subscribe_VT_OPC shows remotely calibrated values and raw values on the VT and forwards the calibration keys to the sensor station via OPC-UA method call.

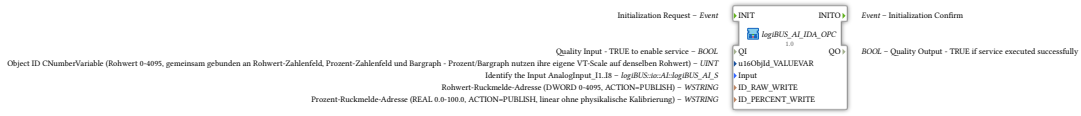
logiBUS_AI_IDA_OPC



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_AI_IDA_OPC` connects a physical analog input (`logiBUS_AI_IDA`, raw value 0-4095, see logiBUS_AI hardware documentation) simultaneously to a VT numeric field (raw value), a second VT numeric field/bargraph (percent), and OPC-UA: the raw value (DWORD) and a linearly converted percent value (REAL) are both published separately via OPC-UA. The percent conversion is done via `F_AI_RAW_TO_PERCENT_AD`.



Function Blocks Used

Sub-blocks: logiBUS_AI_IDA_OPC

- **Type**: SubAppType
- **Internal FBs used**:
  - **logiBUS_AI_IDA**: `logiBUS::io::AI::logiBUS_AI_IDA` — physical analog input, raw value 0-4095 (12-bit), `AnalogInput_hysteresis=DWORD#0`, `TimeDelta=DWORD#250`, `TimeRateLimit=DWORD#100`, `QI` comes from the input `QI`.
  - **AD_SPLIT_2** (type `AD_SPLIT_3`): `adapter::events::unidirectional::AD_SPLIT_3` — splits the DWORD adapter value into three independent outputs (`OUT1..OUT3`).
  - **AD_PUBLISH_RAW** (type `AD_PUBLISH_1`): `adapter::net::AD_PUBLISH_1` — publishes the raw value (DWORD) via OPC-UA, target address `ID_RAW_WRITE`, `QI` comes from the input `QI`.
  - **Q_NumericValue** (type `Q_NumericValue_AUDI`): `isobus::UT::Q::Q_NumericValue_AUDI` — writes the raw value into the VT numeric field `u16ObjId_VALUEVAR`.
  - **F_AI_RAW_TO_PERCENT** (SubApp, type `F_AI_RAW_TO_PERCENT_AD`): `MyLib::sys::F_AI_RAW_TO_PERCENT_AD` — linearly converts the raw value to percent (see F_AI_RAW_TO_PERCENT / F_AI_RAW_TO_PERCENT_AD).
  - **AR_PUBLISH_PERCENT** (type `AR_PUBLISH_1`): `adapter::net::AR_PUBLISH_1` — publishes the percent value (REAL) via OPC-UA, target address `ID_PERCENT_WRITE`, `QI` comes from the input `QI`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_4` — combines the `QO` outputs of `Q_NumericValue`, `logiBUS_AI_IDA`, `AD_PUBLISH_RAW` and `AR_PUBLISH_PERCENT` into the output `QO`.
  - **AD_TO_AUDI**: `adapter::conversion::unidirectional::AD_TO_AUDI` — converts the DWORD raw value to UDINT for the VT numeric field.
- **Operation**: The raw value is split three ways via `AD_SPLIT_2`: one branch goes directly to the raw-value OPC-UA publication, a second via `AD_TO_AUDI` to the VT raw-value numeric field, and a third via `F_AI_RAW_TO_PERCENT_AD` for percent conversion followed by OPC-UA publication.

Program Flow and Connections

1. `Input` ⟶ `logiBUS_AI_IDA.Input`; `u16ObjId_VALUEVAR` ⟶ `Q_NumericValue.u16ObjId`; `ID_RAW_WRITE` ⟶ `AD_PUBLISH_RAW.ID`; `ID_PERCENT_WRITE` ⟶ `AR_PUBLISH_PERCENT.ID`.
2. `logiBUS_AI_IDA.IN` (adapter) ⟶ `AD_SPLIT_2.IN` (adapter).
3. `AD_SPLIT_2.OUT1` ⟶ `AD_PUBLISH_RAW.IN` (raw-value OPC-UA branch).

4. `AD_SPLIT_2.OUT2` ⟶ `AD_TO_AUDI.AD_IN`; `AD_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue.u32NewValue` (VT raw-value branch).
5. `AD_SPLIT_2.OUT3` ⟶ `F_AI_RAW_TO_PERCENT.RAW_IN`; `F_AI_RAW_TO_PERCENT.PERCENT_OUT` ⟶ `AR_PUBLISH_PERCENT.IN` (percent OPC-UA branch).
6. `QI` (`BOOL`, default `TRUE`) ⟶ `Q_NumericValue.QI`, `logiBUS_AI_IDA.QI`, `AD_PUBLISH_RAW.QI` and `AR_PUBLISH_PERCENT.QI`; their `QO` outputs ⟶ `AND_QO.IN1` (`Q_NumericValue`), `IN2` (`logiBUS_AI_IDA`), `IN3` (`AD_PUBLISH_RAW`), `IN4` (`AR_PUBLISH_PERCENT`); `AND_QO.OUT` ⟶ `QO` (`BOOL`, Quality Output - `TRUE` if all four services succeeded).
7. Initialization chain (event input `INIT` / event output `INITO`, both `EInit`): `INIT` ⟶ `logiBUS_AI_IDA.INIT` ⟶ `AD_PUBLISH_RAW.INIT` ⟶ `AR_PUBLISH_PERCENT.INIT` ⟶ `Q_NumericValue.INIT` (each `INITO` ⟶ next `INIT`); `Q_NumericValue.INITO` ⟶ `INITO` and `AND_QO.REQ`.

Technical Details

- Raw value range 0-4095 (12-bit) — see the verified AI hardware documentation for details.
- The percent conversion is purely linear and uncalibrated (no physical scaling), as explicitly noted in the SubApp definition's comment.
- Three independent consumers (two VT displays + two OPC-UA publications) run in parallel from the same `AD_SPLIT_3` branch, so all values reflect the same acquisition cycle synchronously.

Application Scenarios

- An analog process value (e.g. level, pressure) that should be shown locally on the VT (raw and percent) and also read by an upstream SCADA system via OPC-UA in both representations (raw and percent).

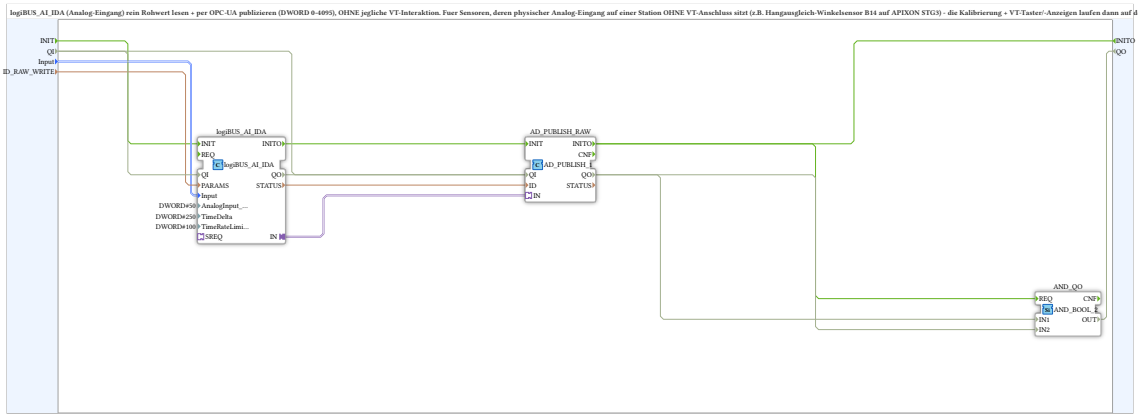
Summary

Full VT and OPC-UA integration of a physical analog input with parallel raw-value and percent representations, built from standard adapter blocks and the reusable percent conversion.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

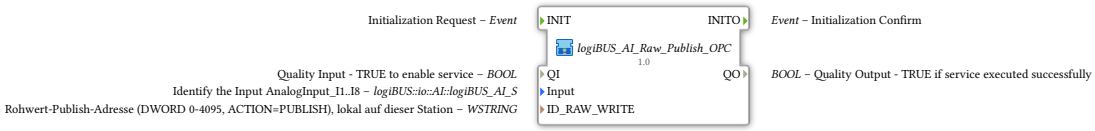
logiBUS_AI_Raw_Publish_OPC



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_AI_Raw_Publish_OPC` reads an analog input (`logiBUS_AI_IDA`) as a raw value (DWORD, 0-4095) and publishes it via OPC-UA, **without any VT interaction**. It is intended for sensors whose physical analog input sits on a station without a VT connection (e.g. slope compensation angle sensor B14 on APIXON STG3). Calibration as well as VT buttons and displays then run on the station with a VT connection and are fed by remote subscribe of this raw value (see `logiBUS_AI_Calibrate_3P_Remote_IDA_OPC`).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; goes to `logiBUS_AI_IDA.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `AD_PUBLISH_RAW.INITO`

Data Inputs

- **QI** (BOOL): quality input - TRUE enables the service (default `TRUE`); goes to `logiBUS_AI_IDA.QI` and `AD_PUBLISH_RAW.QI`
- **Input** (`logiBUS::io::AI::logiBUS_AI_S`): selection of the analog input `AnalogInput_I1..I8` (default `logiBUS_AI::Invalid`)
- **ID_RAW_WRITE** (WSTRING): publish address of the raw value (DWORD 0-4095, `ACTION=PUBLISH`), local on this station

Data Outputs

- **QO** (BOOL): quality output - TRUE if the service was executed successfully (AND of `logiBUS_AI_IDA.QO` and `AD_PUBLISH_RAW.QO`)

Adapters

No adapters are used.

Functionality

The network consists of three blocks (two function blocks plus the combination `AND_QO`):

- `logiBUS_AI_IDA` (`logiBUS::io::AI::logiBUS_AI_IDA`, `QI` from outside): reads the selected analog input (`Input` ⟶ `logiBUS_AI_IDA.Input`). Parameterised are `AnalogInput_hysteresis = DWORD#50`, `TimeDelta = DWORD#250` and `TimeRateLimit = DWORD#100`.
- `AD_PUBLISH_RAW` (`adapter::net::AD_PUBLISH_1`, `QI` from outside): `logiBUS_AI_IDA.IN` ⟶ `AD_PUBLISH_RAW.IN`; publishes the raw value to the address `ID_RAW_WRITE` (`ID_RAW_WRITE` ⟶ `AD_PUBLISH_RAW.ID`).
- `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): combines `logiBUS_AI_IDA.QO` (`IN1`) and `AD_PUBLISH_RAW.QO` (`IN2`) into the output `QO`; `REQ` comes from `AD_PUBLISH_RAW.INITO`.

Initialization: `INIT` ⟶ `logiBUS_AI_IDA.INIT`, `logiBUS_AI_IDA.INITO` ⟶ `AD_PUBLISH_RAW.INIT`, `AD_PUBLISH_RAW.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`).

Technical Details

- Raw value half of `logiBUS_AI_Calibrate_3P_IDA_OPC`: calibration does not take place here but on the station with a VT connection
- No further ports apart from `INIT`/`INITO`, `QI`/`QO` and the two input variables `Input` and `ID_RAW_WRITE`; no adapters
- The behaviour of the raw value input (hysteresis, `TimeDelta`, `TimeRateLimit`) is set through the parameters of `logiBUS_AI_IDA`

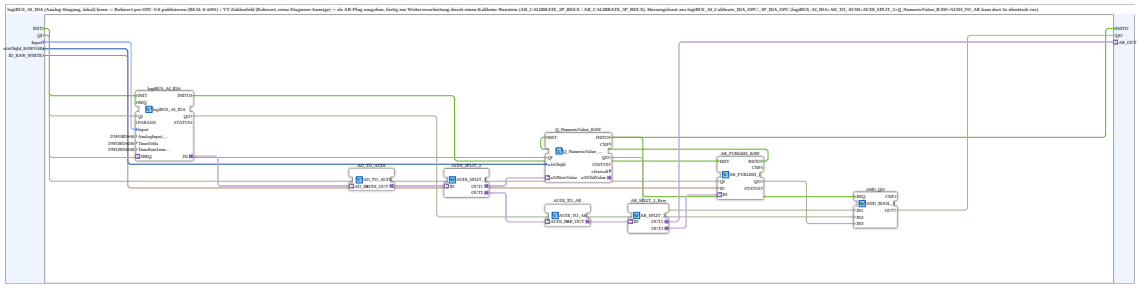
Application Scenarios

- Sensors on stations without a VT connection whose calibration and display run on another station

Conclusion

`logiBUS_AI_Raw_Publish_OPC` provides the raw value of an analog input via OPC-UA and is thus the sensor side of a cross-station calibration.

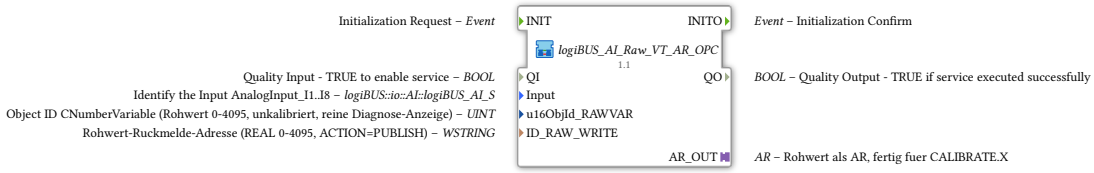
logiBUS_AI_Raw_VT_AR_OPC



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_AI_Raw_VT_AR_OPC` reads a local logiBUS analog input (`logiBUS_AI_IDA`), publishes the raw value via OPC-UA (`REAL`, 0..4095) and shows it in a VT number field (raw value, pure diagnostic display). In addition it is output as an `AR` plug, ready for further processing by a calibration block (`AR_CALIBRATE_2P_REF.X` or `AR_CALIBRATE_3P_REF.X`). The block was extracted from `logiBUS_AI_Calibrate_IDA_OPC` and `logiBUS_AI_Calibrate_3P_IDA_OPC`, where the combination `logiBUS_AI_IDA + AD_TO_AUDI + AUDI_SPLIT_2 + Q_NumericValue_RAW + AUDI_TO_AR` occurred twice identically. Related: `logiBUS_AI_Raw_Publish_OPC`.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirmation (from `Q_NumericValue_RAW.INITO`)

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - `TRUE` enables the services (⟶ `Q_NumericValue_RAW.QI`, `logiBUS_AI_IDA.QI` and `AR_PUBLISH_RAW.QI`)
- **Input** (`logiBUS::io::AI::logiBUS_AI_S`): selection of the input `AnalogInput_I1..I8` (default `logiBUS_AI::Invalid`)
- **u16ObjId_RAWVAR** (`UINT`): object ID of the `CNumberVariable` (raw value 0..4095, uncalibrated, pure diagnostic display)
- **ID_RAW_WRITE** (`WSTRING`): raw value feedback address (`REAL` 0..4095, `ACTION=PUBLISH`)

Data Outputs

- **QO** (`BOOL`): Quality Output - `TRUE` if all services succeeded (AND of `Q_NumericValue_RAW.QO`, `logiBUS_AI_IDA.QO` and `AR_PUBLISH_RAW.QO`)

Adapters

- **AR_OUT** (plug, `adapter::types::unidirectional::AR`): raw value as `AR`, ready for `CALIBRATE.X`

Function Blocks (FBs) Used

- **`logiBUS_AI_IDA`** (`logiBUS::io::AI::logiBUS_AI_IDA`): parameters `AnalogInput_hysteresis = DWORD#50`, `TimeDelta = DWORD#250`, `TimeRateLimit = DWORD#100`
- **`AD_TO_AUDI`** (`adapter::conversion::unidirectional::AD_TO_AUDI`)
- **`AUDI_SPLIT_2`** (`adapter::events::unidirectional::AUDI_SPLIT_2`)
- **`Q_NumericValue_RAW`** (`isobus::UT::Q::Q_NumericValue_AUDI`): VT raw value display
- **`AUDI_TO_AR`** (`adapter::conversion::unidirectional::AUDI_TO_AR`)

- **AR_SPLIT_2_Raw** (adapter::events::unidirectional::AR_SPLIT_2)
- **AR_PUBLISH_RAW** (adapter::net::AR_PUBLISH_1): OPC-UA publish
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_3): combines the QO outputs of Q_NumericValue_RAW, logiBUS_AI_IDA and AR_PUBLISH_RAW

Functionality

- Input ⟶ logiBUS_AI_IDA.Input
- logiBUS_AI_IDA.IN ⟶ AD_TO_AUDI.AD_IN ⟶ AUDI_SPLIT_2.IN
- AUDI_SPLIT_2.OUT1 ⟶ Q_NumericValue_RAW.u32NewValue (VT display); u16ObjId_RAWVAR ⟶ Q_NumericValue_RAW.u16ObjId
- INIT ⟶ logiBUS_AI_IDA.INIT; logiBUS_AI_IDA.INITO ⟶ AR_PUBLISH_RAW.INIT; AR_PUBLISH_RAW.INITO ⟶ Q_NumericValue_RAW.INIT; Q_NumericValue_RAW.INITO ⟶ INITO (and ⟶ AND_QO.REQ)
- QI ⟶ Q_NumericValue_RAW.QI, logiBUS_AI_IDA.QI and AR_PUBLISH_RAW.QI
- Q_NumericValue_RAW.QO ⟶ AND_QO.IN1; logiBUS_AI_IDA.QO ⟶ AND_QO.IN2; AR_PUBLISH_RAW.QO ⟶ AND_QO.IN3; AND_QO.OUT ⟶ QO
- AUDI_SPLIT_2.OUT2 ⟶ AUDI_TO_AR.AUDI_IN ⟶ AR_SPLIT_2_Raw.IN
- AR_SPLIT_2_Raw.OUT1 ⟶ AR_OUT
- AR_SPLIT_2_Raw.OUT2 ⟶ AR_PUBLISH_RAW.IN; ID_RAW_WRITE ⟶ AR_PUBLISH_RAW.ID

Technical Details

- The raw value is uncalibrated (0..4095); calibration takes place in the downstream block
- Two splitters (AUDI_SPLIT_2, AR_SPLIT_2_Raw), since an adapter must not branch to several targets
- Input parameters of logiBUS_AI_IDA: hysteresis 50, TimeDelta 250, TimeRateLimit 100

Application Scenarios

- Showing the raw value of an analog input, publishing it via OPC-UA and feeding it to a calibration block

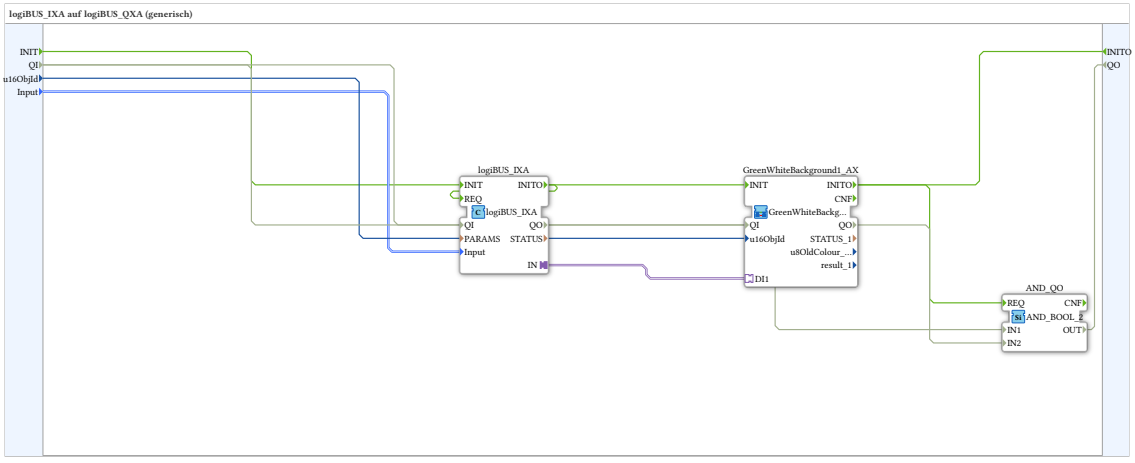
Summary

logiBUS_AI_Raw_VT_AR_OPC delivers the analog raw value simultaneously to the VT, OPC-UA and an AR output.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

logiBUS_IXA_BG

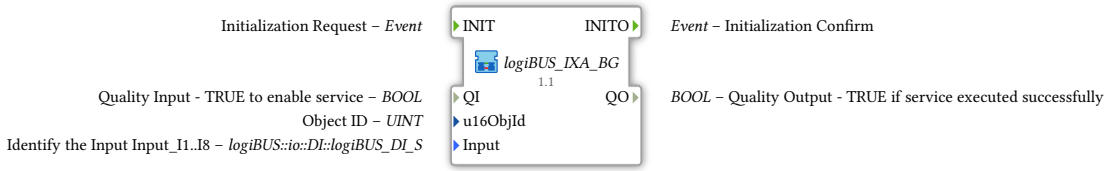


• – ○ • – ○ • – ○ •

Introduction

logiBUS_IXA_BG connects a physical digital input (logiBUS_IXA) to a VT status indication via the background-color family GreenWhiteBackground1_AX (see Background Color Blocks (shared pattern)) –

the current state of the input is made visible on the VT as a green/white background, with no OPC-UA connection. For the variant with additional OPC-UA feedback, see `logiBUS_IXA_BG_OPC`.



Function Blocks Used

Sub-blocks: logiBUS_IXA_BG

- **Type**: SubAppType
- **Interface (additional)**: event input `INIT`, event output `INITO` (each `EInit`), data input `QI` (`BOOL`, default `TRUE`), data output `QO` (`BOOL`).
- **Internal FBs used**:
  - **logiBUS_IXA**: `logiBUS::io::DI::logiBUS_IXA` — physical digital input, adapter output `IN`; `QI` comes from the subapp input `QI`.
  - **GreenWhiteBackground1_AX** (SubApp): `MyLib::sys::GreenWhiteBackground1_AX` — single-object background-color block from the Background Color family, drives the VT object background color from `DI1`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — combines the `QO` outputs of `logiBUS_IXA` and `GreenWhiteBackground1_AX` into `QO`.
- **Operation**: The physical input's adapter output is wired directly to the `DI1` adapter input of the background-color block, which handles the color switch of a VT object.

Program Flow and Connections

1. `Input` ⟶ `logiBUS_IXA.Input`; `u16ObjId` ⟶ `GreenWhiteBackground1_AX.u16ObjId`.
2. `logiBUS_IXA.IN` (adapter) ⟶ `GreenWhiteBackground1_AX.DI1` (adapter).
3. Initialization: `INIT` ⟶ `logiBUS_IXA.INIT`; `logiBUS_IXA.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`; `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
4. Quality: `QI` ⟶ `logiBUS_IXA.QI` and `GreenWhiteBackground1_AX.QI`; `logiBUS_IXA.QO` ⟶ `AND_QO.IN1`, `GreenWhiteBackground1_AX.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Application Scenarios

- Purely visual status indication of a physical digital input on the VT (e.g. limit switch state), where the value does not need to be reported externally via OPC-UA.

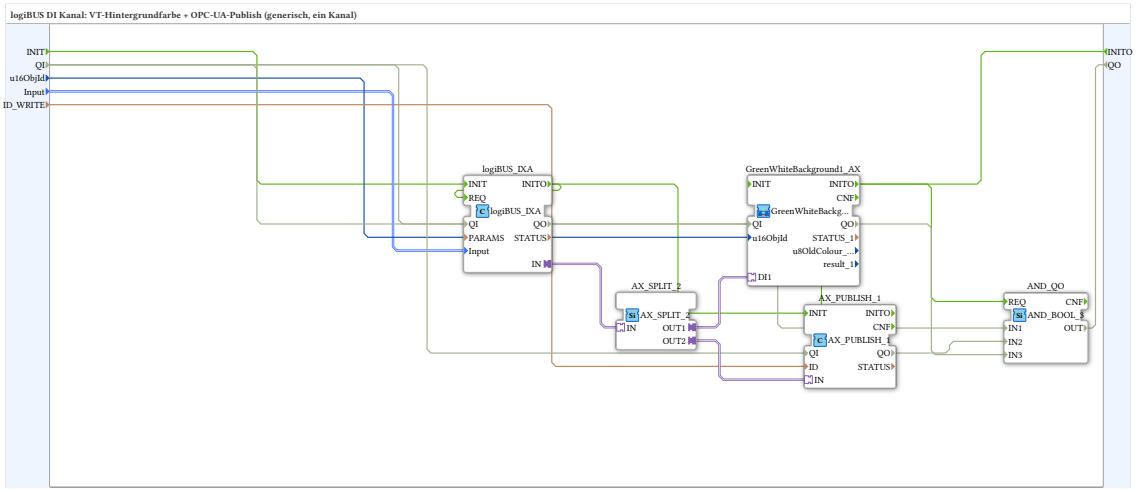
Summary

Combines a physical digital input with the standard background-color family into a simple VT status indication.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

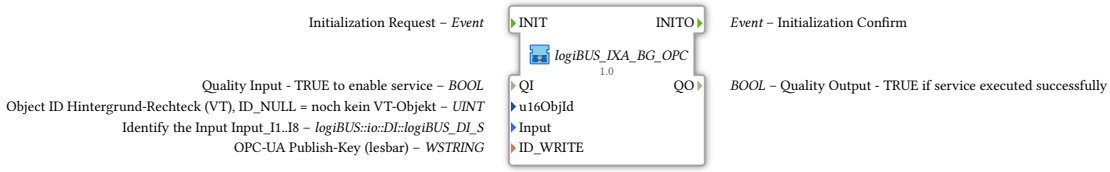
logiBUS_IXA_BG_OPC



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_IXA_BG_OPC` is the reusable block for **a single digital input** with VT status display (background color) and OPC-UA publish. It is instantiated 8 times with different parameters in `InputOutputTesterButton_DIDO_OPC_UA` and is reused unchanged for the 8 inputs in the PWM example `InputOutputTesterButton_PWM_OPC_UA`.



Interface

- **Event input**: `INIT` (`EInit`, Initialization Request)
- **Event output**: `INITO` (`EInit`, Initialization Confirm)
- **Data inputs**: `QI` (`BOOL`, default `TRUE`, Quality Input), `u16ObjId`, `Input`, `ID_WRITE`
- **Data output**: `QO` (`BOOL`, Quality Output)

Function Blocks (FBs) Used

Sub-blocks: logiBUS_IXA_BG_OPC

- **Type**: SubAppType
- **Internal FBs used**:
  - **logiBUS_IXA**: `logiBUS::io::DI::logiBUS_IXA` — adapter-based digital input; `QI` from the block input, `Input` identifies `Input_I1..I8`.
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` — splits the single adapter signal from `logiBUS_IXA.IN` into two independent destinations.
  - **GreenWhiteBackground1_AX** (SubApp, `MyLib::sys`): sets the VT background color (green/white) to match the input state.
  - **AX_PUBLISH_1**: `adapter::net::AX_PUBLISH_1` — OPC-UA publish adapter, `QI` from the block input.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — ANDs the `QO` outputs of `logiBUS_IXA`, `AX_PUBLISH_1` and `GreenWhiteBackground1_AX` into the block output `QO`.
- **Functionality**: Purely one-way data flow from the physical input to two display/reporting paths — no write-back possible, since a digital input cannot be set externally.

Program Flow and Connections

1. `Input` (parameter, identifies `Input_I1..I8`) ⟶ `logiBUS_IXA.Input` (data connection, hidden).
2. `u16ObjId` (VT object ID of the status display) ⟶ `GreenWhiteBackground1_AX.u16ObjId`.

3. `ID_WRITE` (OPC-UA address) ⟶ `AX_PUBLISH_1.ID`.
4. **Adapter chain**: `logiBUS_IXA.IN` ⟶ `AX_SPLIT_2.IN` ⟶ `AX_SPLIT_2.OUT1` ⟶ `GreenWhiteBackground1_AX.DI1` (VT display) and `AX_SPLIT_2.OUT2` ⟶ `AX_PUBLISH_1.IN` (OPC-UA publish).
5. **Initialization chain**: `INIT` ⟶ `logiBUS_IXA.INIT`, `logiBUS_IXA.INITO` ⟶ `AX_PUBLISH_1.INIT`, `AX_PUBLISH_1.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`, `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
6. `QI` ⟶ `logiBUS_IXA.QI`, `AX_PUBLISH_1.QI` and `GreenWhiteBackground1_AX.QI`.
7. `logiBUS_IXA.QO` ⟶ `AND_QO.IN1`, `AX_PUBLISH_1.QO` ⟶ `AND_QO.IN2`, `GreenWhiteBackground1_AX.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.

Technical Details

- **AX_SPLIT_2 instead of two data connections**: since an adapter output can only be connected directly to one place, `AX_SPLIT_2` splits the signal for the VT display and the OPC-UA publish.
- **Unidirectional**: unlike its output counterpart `Button_IXA_TO_logiBUS_QXA_BG_OPC`, there is no `AX_SUBSCRIBE_1` here — an input is never written from the web.

Application Scenarios

- Any exercise with digital inputs that should be visible both on the VT (status color) and via OPC-UA (web client).

Summary

`logiBUS_IXA_BG_OPC` encapsulates the standard combination “digital input + VT status color + OPC-UA publish” in a single, repeatedly reusable block.

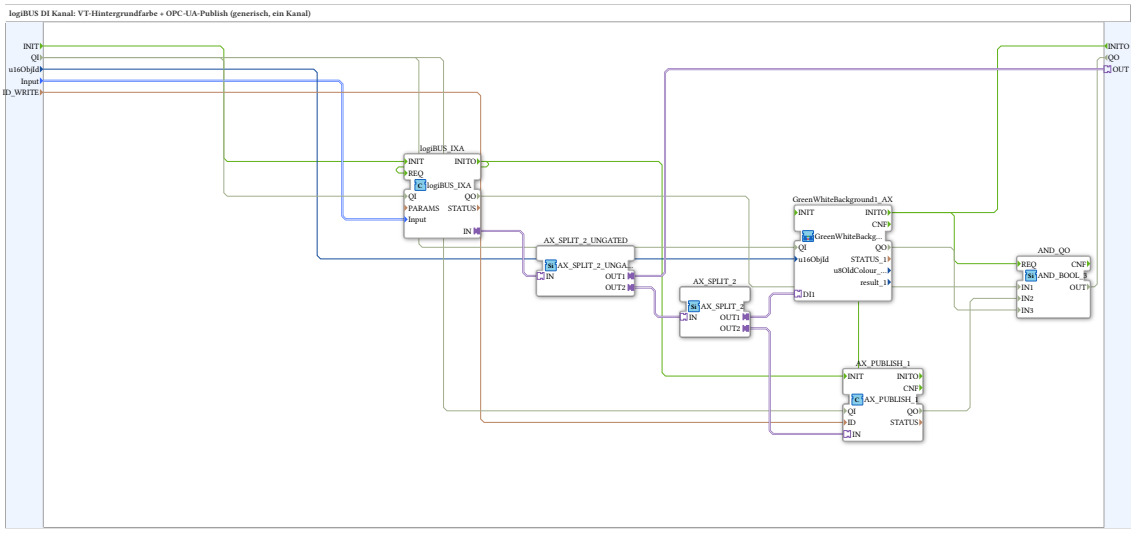
🛠️ Related Exercises

- InputOutputTesterButton_DIDO_OPC_UA
- InputOutputTesterButton_PWM_OPC_UA

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

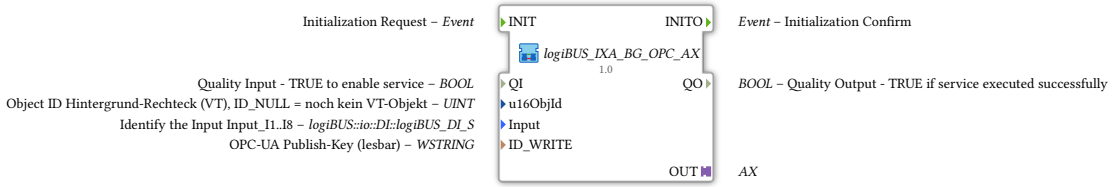
logiBUS_IXA_BG_OPC_AX



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_IXA_BG_OPC_AX` is a generic block for **one logiBUS DI channel** with VT background colour and OPC-UA publish. Compared with `logiBUS_IXA_BG_OPC`, it additionally passes the input state on as the `AX` plug `OUT`, so that other blocks can use the level.



Function Blocks (FBs) Used

Sub-blocks: logiBUS_IXA_BG_OPC_AX

- **Type**: SubAppType
- **Internal blocks used**:
  - **logiBUS_IXA**: `logiBUS::io::DI::logiBUS_IXA` - digital input; `Input` identifies `Input_I1..I8`
  - **AX_SPLIT_2_UNGATED**: `adapter::events::unidirectional::AX_SPLIT_2_UNGATED` - branches the input state into the plug `OUT` and into the second branch
  - **AX_SPLIT_2**: `adapter::events::unidirectional::AX_SPLIT_2` - branches the second branch into VT display and publish
  - **GreenWhiteBackground1_AX**: SubApp `MyLib::sys::GreenWhiteBackground1_AX` - sets the background colour (green/white)
  - **AX_PUBLISH_1**: `adapter::net::AX_PUBLISH_1` - OPC-UA publish
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` - combines the `QO` outputs of `logiBUS_IXA`, `AX_PUBLISH_1` and `GreenWhiteBackground1_AX` into the SubApp output `QO`
- **Imports**: `ID_NULL`, `logiBUS_DI`, `logiBUS_DI_S`

Interface Structure

Event Inputs / Event Outputs

- **INIT** (`EInit`, input): initialization request; passed on to `logiBUS_IXA.INIT`
- **INITO** (`EInit`, output): initialization confirmation; comes from `GreenWhiteBackground1_AX.INITO`

The runtime events still travel via the adapter.

Data Inputs

- **QI** (`BOOL`, default `TRUE`): quality input, service enable; passed on to `logiBUS_IXA`, `AX_PUBLISH_1` and `GreenWhiteBackground1_AX`
- **u16ObjId** (`UINT`, default `ID_NULL`): Object ID of the background rectangle (VT); `ID_NULL` = no VT object yet
- **Input** (`logiBUS::io::DI::logiBUS_DI_S`, default `logiBUS_DI::Invalid`): Identifies the input `Input_I1..I8`
- **ID_WRITE** (`WSTRING`): OPC-UA publish key (readable)

Data Outputs

- **QO** (`BOOL`): quality output, `AND` of the `QO` outputs of `logiBUS_IXA`, `AX_PUBLISH_1` and `GreenWhiteBackground1_AX` (via `AND_QO`)

Adapters

- **OUT** (plug): `adapter::types::unidirectional::AX` - input state as a level

Program Flow and Connections

- `Input` ⟶ `logiBUS_IXA.Input`; `logiBUS_IXA.INITO` ⟶ `logiBUS_IXA.REQ` (first read after initialisation)
- `logiBUS_IXA.IN` ⟶ `AX_SPLIT_2_UNGATED.IN`
- `AX_SPLIT_2_UNGATED.OUT1` ⟶ `OUT`
- `AX_SPLIT_2_UNGATED.OUT2` ⟶ `AX_SPLIT_2.IN`
- `AX_SPLIT_2.OUT1` ⟶ `GreenWhiteBackground1_AX.DI1`; `u16ObjId` ⟶ `GreenWhiteBackground1_AX.u16ObjId`
- `AX_SPLIT_2.OUT2` ⟶ `AX_PUBLISH_1.IN`; `ID_WRITE` ⟶ `AX_PUBLISH_1.ID`
- Initialization: `INIT` ⟶ `logiBUS_IXA.INIT`; `logiBUS_IXA.INITO` ⟶ `AX_PUBLISH_1.INIT` (in addition to `logiBUS_IXA.INITO` ⟶ `logiBUS_IXA.REQ`); `AX_PUBLISH_1.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`; `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`

- Quality: `QI` ⟶ `logiBUS_IXA.QI`, `AX_PUBLISH_1.QI` and `GreenWhiteBackground1_AX.QI`; their `QO` outputs ⟶ `AND_QO.IN1` (`logiBUS_IXA`), `IN2` (`AX_PUBLISH_1`), `IN3` (`GreenWhiteBackground1_AX`); `AND_QO.OUT` ⟶ `QO`

Technical Details

- Two splitters in series, because an adapter plug can be connected to only one destination and three destinations (`OUT`, VT display, publish) are served here
- One-way data flow: a digital input is not written from outside
- Version 1.0 (2023-10-23)

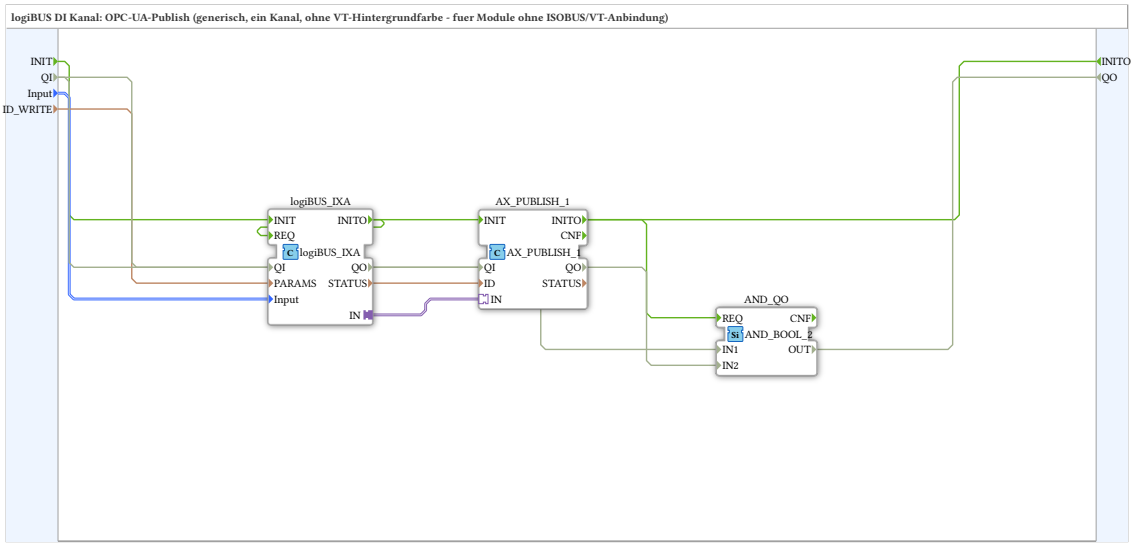
Application Scenarios

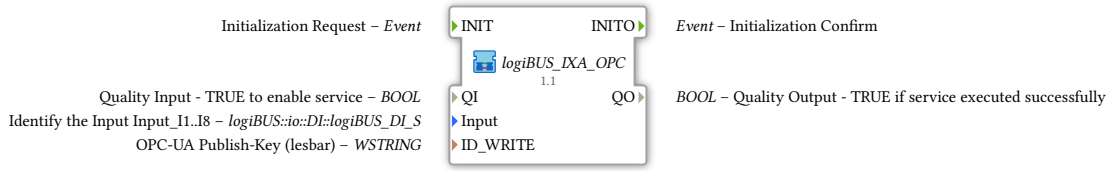
- Digital input with VT status colour and OPC-UA publish whose state should additionally be used by other blocks

Summary

`logiBUS_IXA_BG_OPC_AX` extends `logiBUS_IXA_BG_OPC` with a passed-through `AX` output `OUT` for the input state.

logiBUS_IXA_OPC





• – ○ • – ○ • – ○ •

Introduction

The logiBUS_IXA_OPC subapplication provides a generic single-channel logiBUS digital input (DI) monitoring solution with built-in OPC-UA publish capability. It is designed for modules that do not require ISOBUS/VT integration, as it omits VT background color handling. The subapp combines the logiBUS_IXA function block for digital input processing with the AX_PUBLISH_1 adapter for publishing input states to an OPC-UA server.

Interface Structure

The subapplication exposes the events `INIT`/`INITO`, the data `QI`/`QO` and two further data inputs. All data processing and publishing logic is encapsulated internally within the subapp network.

Event Inputs

- **INIT** (`EInit`): initialization request; starts the init chain at `logiBUS_IXA.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `AX_PUBLISH_1.INITO`, the end of the init chain

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	TRUE	Quality input; passed on to `logiBUS_IXA.QI` and `AX_PUBLISH_1.QI`
Input	logiBUS::io::DI::logiBUS_DI_S	logiBUS_DI::Invalid	Identifies the logiBUS digital input channel (Input_I1..I8)
ID_WRITE	WSTRING	—	OPC-UA publish key (readable)

Data Outputs

- **QO** (BOOL): quality output, AND combination (`AND_QO`, `AND_BOOL_2`) of `logiBUS_IXA.QO` and `AX_PUBLISH_1.QO`.

The digital input state itself is published externally via the OPC-UA mechanism rather than through output variables.

Adapters

No adapters are exposed on the subapplication interface. Internally, the adapter connection between logiBUS_IXA.IN and AX_PUBLISH_1.IN transfers the processed digital input data to the publisher adapter.

Functionality

The subapplication accepts a selected logiBUS digital input channel (specified via the Input variable of type logiBUS_DI_S, allowing selection from Input_I1 through Input_I8) and routes it to the internal logiBUS_IXA function block. This block processes the raw digital input data. The processed result is then passed via an internal adapter connection to the AX_PUBLISH_1 adapter, which publishes the data to an OPC-UA server. The ID_WRITE parameter supplies a readable publish key used by the OPC-UA publisher to identify the published data item.

Initialization and quality: `INIT` ⟶ `logiBUS_IXA.INIT`, `logiBUS_IXA.INITO` ⟶ `AX_PUBLISH_1.INIT`, `AX_PUBLISH_1.INITO` ⟶ `INITO` and `AND_QO.REQ`. `QI` is passed on to `logiBUS_IXA.QI` and `AX_PUBLISH_1.QI`; `logiBUS_IXA.QO` ⟶ `AND_QO.IN1`, `AX_PUBLISH_1.QO` ⟶ `AND_QO.IN2` (`iec61131::booleanOperators::AND_BOOL_2`), `AND_QO.OUT` ⟶ `QO`.

Technical Features

- Single-channel logiBUS digital input handling with support for Input_I1 to Input_I8.
- OPC-UA publish functionality through the internal AX_PUBLISH_1 adapter.
- Generic design for modules without ISOBUS/VT connectivity (no VT background color handling).
- Configurable OPC-UA publish key via the WSTRING ID_WRITE parameter.
- Initial input state set to logiBUS_DI::Invalid, ensuring undefined states are explicitly handled.
- The internal enable signals (QI) of the logiBUS_IXA and AX_PUBLISH_1 blocks come from the block input `QI` (initial value TRUE).

State Overview

The subapplication does not define explicit state machines of its own. Its behavior is determined by the internal logiBUS_IXA function block, which manages the digital input processing states. The initial state of the Input variable is logiBUS_DI::Invalid, representing an undefined or unselected input channel until a valid channel is configured.

Application Scenarios

- Industrial automation environments where logiBUS digital input states need to be monitored and forwarded to OPC-UA-based SCADA or MES systems.
- Modular systems that lack ISOBUS/VT connectivity but require OPC-UA data publishing for remote supervision.

- Generic single-channel DI monitoring applications where a readable OPC-UA key is needed for data identification.
- Retrofit or extension scenarios where existing logiBUS DI inputs are integrated into OPC-UA-based supervisory layers.

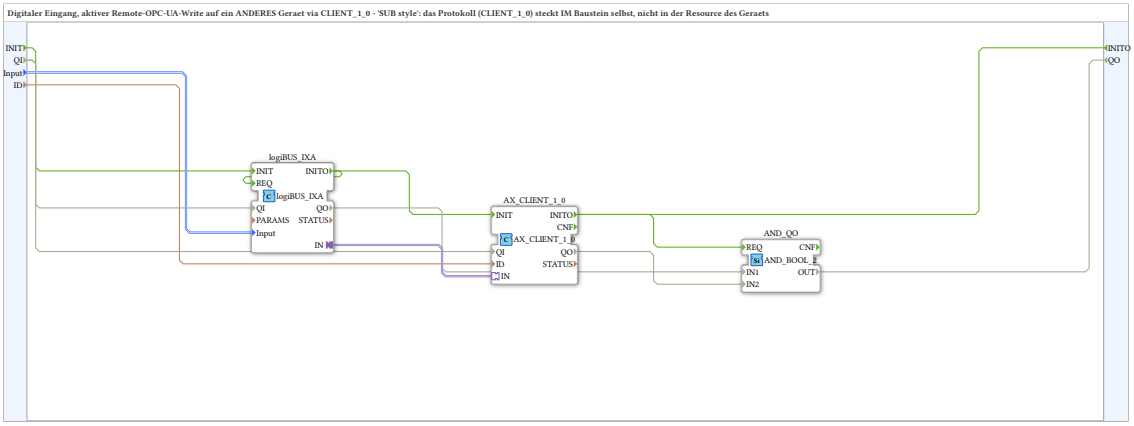
Comparison with Similar Blocks

Compared to other logiBUS DI subapplications, this block distinguishes itself through its OPC-UA publishing focus. While blocks like logiBUS_DI or logiBUS_DI_S provide raw digital input handling, logiBUS_IXA_OPC adds the AX_PUBLISH_1 adapter for direct OPC-UA publication. Unlike subapplications that include ISOBUS/VT integration, this block deliberately omits VT background color handling, making it lighter and more suitable for non-VT modules. It also differs from multi-channel variants by supporting a single channel, simplifying configuration for one-input scenarios.

Conclusion

The logiBUS_IXA_OPC subapplication offers a compact, generic solution for publishing a single logiBUS digital input channel to an OPC-UA server. By combining the logiBUS_IXA processing block with the AX_PUBLISH_1 publisher adapter, it delivers a complete publish path in a single subapp. Its lack of ISOBUS/VT dependencies and its configurable publish key make it well suited for straightforward automation scenarios where OPC-UA connectivity of DI states is required.

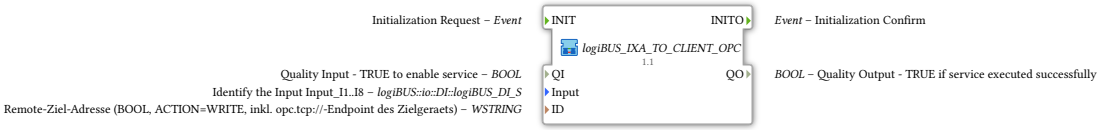
logiBUS_IXA_TO_CLIENT_OPC



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_IXA_TO_CLIENT_OPC` reads a digital input (`logiBUS_IXA`) and actively writes the value to ANOTHER device via OPC-UA using `AX_CLIENT_1_0` - "Krauternter style" (per the source comment): the protocol (`AX_CLIENT_1_0`) lives INSIDE the block itself, not in the device's resource, unlike the "SUB style" of the PC_A/PC_B blocks, where trigger/server instances are wired separately in the composite.



Function blocks used

- **logiBUS_IXA** (`logiBUS::io::DI::logiBUS_IXA`): physical digital input, identified via `Input_I1..I8`.
- **AX_CLIENT_1_0** (`adapter::net::AX_CLIENT_1_0`): actively writes the adapter value to the remote target address `ID` (including the target device's `opc.tcp://` endpoint).

Initialization and quality

The block has the event input `INIT`, the event output `INITO` (both `EInit`), the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). `INIT` ⟶ `logiBUS_IXA.INIT`, `logiBUS_IXA.INITO` ⟶ `AX_CLIENT_1_0.INIT` (and ⟶ `logiBUS_IXA.REQ`), `AX_CLIENT_1_0.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`. `QI` is passed to `logiBUS_IXA` and `AX_CLIENT_1_0`. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`), which combines `logiBUS_IXA.QO` and `AX_CLIENT_1_0.QO`.

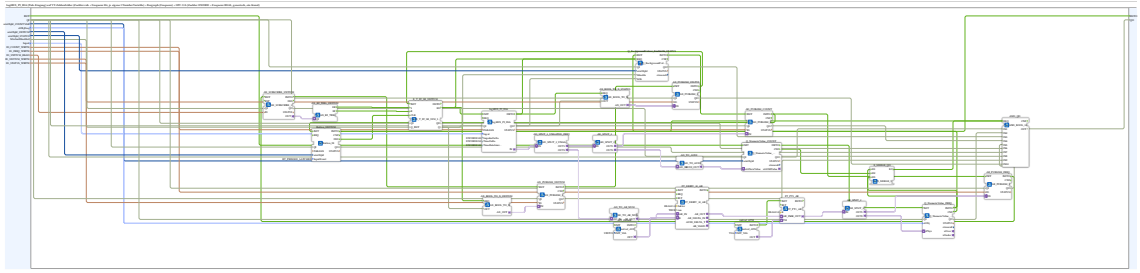
Summary

Single-block bridge from a physical digital input to an active remote write on another device - counterpart to `logiBUS_QXA_FROM_SUBSCRIBE_OPC` on the receiving side.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

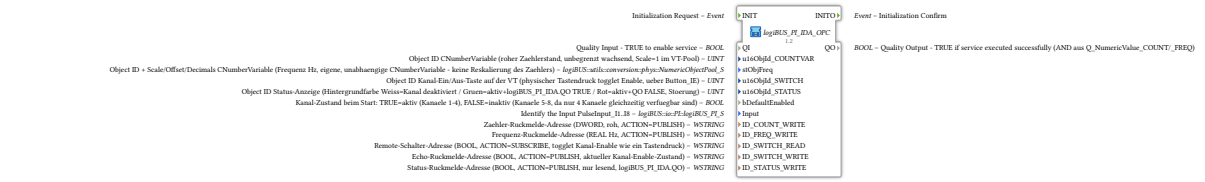
logiBUS_PI_IDA_OPC



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_PI_IDA_OPC` connects a physical pulse/counter input (`logiBUS_PI_IDA`, one channel) to two independent VT number fields (raw counter value and the frequency in Hz derived from it) and to OPC-UA: the raw counter value (DWORD) is published directly, the frequency is computed by time derivative (`FT_DERIV_10_AR`) from the counter value, smoothed with a PT1 element (`FT_PT1_AR`) and published separately (REAL, physically scaled via `NumericObjectPool_S`). In addition, the channel can be switched on and off (enable) via a VT key or an OPC-UA switch; a status display (background colour) and an OPC-UA status report disabled/active/fault.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirmation (from `Button_SWITCH.INITO`)

Data Inputs

- **QI** (`BOOL`, default `TRUE`): Quality Input - TRUE enables the services
- **u16ObjId_COUNTVAR** (`UINT`, default `ID_NULL`): object ID of the `CNumberVariable` for the raw counter value (ever-increasing, scale=1 in the VT pool)
- **stObjFreq** (`NumericObjectPool_S`): object ID + scale/offset/decimals of the `CNumberVariable` for the frequency in Hz (own, independent variable, no rescaling of the counter)
- **u16ObjId_SWITCH** (`UINT`, default `ID_NULL`): object ID of the channel on/off key on the VT (a key press toggles the enable, via `Button_IE`)
- **u16ObjId_STATUS** (`UINT`, default `ID_NULL`): object ID of the status display (background colour white = channel disabled, green = active and `logiBUS_PI_IDA.QO` TRUE, red = active and `QO` FALSE, fault)
- **bDefaultEnabled** (`BOOL`): channel state at start (TRUE = active, e.g. channels 1-4; FALSE = inactive, e.g. channels 5-8, since only 4 channels are available at the same time)
- **Input** (`logiBUS::io::PI::logiBUS_PI_S`): selection of the input `PulseInput_I1..I8` (default `logiBUS_PI::Invalid`)
- **ID_COUNT_WRITE** (`WSTRING`): counter feedback address (DWORD, raw, `ACTION=PUBLISH`)

- **ID_FREQ_WRITE** (WSTRING): frequency feedback address (REAL Hz, ACTION=PUBLISH)
- **ID_SWITCH_READ** (WSTRING): remote switch address (BOOL, ACTION=SUBSCRIBE, toggles the channel enable like a key press)
- **ID_SWITCH_WRITE** (WSTRING): echo feedback address (BOOL, ACTION=PUBLISH, current channel enable state)
- **ID_STATUS_WRITE** (WSTRING): status feedback address (BOOL, ACTION=PUBLISH, read-only, logiBUS_PI_IDA.QO)

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if the services were executed successfully (AND_QO, AND_BOOL_10: AND of Q_NumericValue_COUNT.QO, Q_NumericValue_FREQ.QO, Q_BackgroundColour_EnableOk_STATUS.QO, AD_PUBLISH_COUNT.QO, AR_PUBLISH_FREQ.QO, E_T_FF_SR_SWITCH.QO, Button_SWITCH.QO, AX_SUBSCRIBE_SWITCH.QO, AX_PUBLISH_SWITCH.QO and AX_PUBLISH_STATUS.QO)

Function Blocks Used

Sub-blocks: logiBUS_PI_IDA_OPC

- **Type**: SubAppType
- **Internal FBs used**:
  - **logiBUS_PI_IDA**: logiBUS::io::PI::logiBUS_PI_IDA — physical pulse input, ImpulseDelta=DWORD#100, TimeDelta=DWORD#250, TimeRateLimit=DWORD#100; its QI comes from the channel enable (E_T_FF_SR_SWITCH.Q).
  - **AD_SPLIT_2_UNGATED_FREQ**: adapter::events::unidirectional::AD_SPLIT_2_UNGATED — branches the DWORD counter value into the counter branch and the frequency branch.
  - **AD_SPLIT_2**: adapter::events::unidirectional::AD_SPLIT_2 — branches the counter value to the OPC-UA publish and the VT display.
  - **AD_PUBLISH_COUNT**: adapter::net::AD_PUBLISH_1 — publishes the raw counter value via OPC-UA, target address ID_COUNT_WRITE.
  - **AD_TO_AUDI**: adapter::conversion::unidirectional::AD_TO_AUDI — bit reinterpretation DWORD→UDINT for the VT counter field.
  - **Q_NumericValue_COUNT**: isobus::UT::Q::Q_NumericValue_AUDI — writes the raw counter value to u16ObjId_COUNTVAR.
  - **AD_TO_AR_NUM**: adapter::conversion::unidirectional::AD_TO_AR_NUM — numerically correct conversion DWORD counter value → REAL for the derivative.
  - **FT_DERIV_10_AR**: OSCAT_adapter::Control::FT_DERIV_10_AR (Faktor=1.0, run=TRUE) — time derivative, delivers the rate of change of the counter value = frequency in Hz; the parameter K is an AUI adapter socket fed by LIT_DerivK (adapter::types::unidirectional::AUI::initval::initval_AUI, INIT_VAL=9).
  - **FT_PT1_AR**: OSCAT_adapter::Control::FT_PT1_AR (K=1.0) — PT1 smoothing of the frequency; the time constant TM comes from initval_ATM (adapter::types::unidirectional::ATM::initval::initval_ATM, INIT_VAL=T#2s).
  - **AR_SPLIT_2**: adapter::events::unidirectional::AR_SPLIT_2 — branches the smoothed frequency value to two outputs.
  - **AR_PUBLISH_FREQ**: adapter::net::AR_PUBLISH_1 — publishes the frequency via OPC-UA, target address ID_FREQ_WRITE.
  - **Q_NumericValue_FREQ**: isobus::UT::Q::Q_NumericValue_PHYSA — writes the frequency, physically scaled (stObjFreq: r32Scale, i32Offset, u8Decimals), into the VT number field/bargraph.
  - **Button_SWITCH**: isobus::UT::io::Button::Button_IE (InputEvent=BT_PRESSED_LATCHED) — reads the channel on/off key (u16ObjId_SWITCH).
  - **AX_SUBSCRIBE_SWITCH** / **AX_RF_TRIG_SWITCH**: adapter::net::AX_SUBSCRIBE_1 and adapter::events::unidirectional::AX_RF_TRIG — remote switch (ID_SWITCH_READ) with edge detection (ER → set, EF → reset of the enable flip-flop).
  - **E_T_FF_SR_SWITCH**: iec61499::events::E_T_FF_SR_SYM_INIT — enable flip-flop (Q_INIT = bDefaultEnabled); Button_SWITCH.IND clocks CLK.
  - **AX_BOOL_TO_X_SWITCH** / **AX_PUBLISH_SWITCH**: adapter::conversion::unidirectional::AX_BOOL_TO_X and adapter::net::AX_PUBLISH_1 — publish the enable state (ID_SWITCH_WRITE).

  - **Q_BackgroundColour_EnableOk_STATUS**: SubApp `MyLib::sys::Q_BackgroundColour_EnableOk_A` — status background colour (`u16ObjId_STATUS`, `bEnable` = enable, `bOk` = `logiBUS_PI_IDA.QO`).
  - **AX_BOOL_TO_X_STATUS** / **AX_PUBLISH_STATUS**: `adapter::conversion::unidirectional::AX_BOOL_TO_X` and `adapter::net::AX_PUBLISH_1` — publish `logiBUS_PI_IDA.QO` (`ID_STATUS_WRITE`).
  - **E_MERGE_QO**: `iec61499::events::E_MERGE_3` — merges `Q_NumericValue_FREQ.INITO`, `Q_NumericValue_COUNT.CNF` and `Q_NumericValue_FREQ.CNF`; its output `EO` triggers `AND_QO.REQ`, so `QO` is re-evaluated on every new `Q_NumericValue_COUNT`/`Q_NumericValue_FREQ` value (`CNF`) (up to v1.1 `EO` was not connected and `QO` was only computed at start-up).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_10` — forms `QO` (see Data Outputs).
- **Functionality**: The raw counter value is branched: one branch goes directly to the OPC-UA publication, a second one via `AD_TO_AUDI` to the VT counter field, a third one via `AD_TO_AR_NUM`, `FT_DERIV_10_AR` and `FT_PT1_AR` to the frequency calculation, whose result is duplicated again (OPC-UA publication and VT display). The channel enable (key, remote switch) releases `logiBUS_PI_IDA`.

Program Flow and Connections

1. `Input` ⟶ `logiBUS_PI_IDA.Input`; `u16ObjId_COUNTVAR` ⟶ `Q_NumericValue_COUNT.u16ObjId`; `stObjFreq` ⟶ `Q_NumericValue_FREQ.stObj`; `ID_COUNT_WRITE` ⟶ `AD_PUBLISH_COUNT.ID`; `ID_FREQ_WRITE` ⟶ `AR_PUBLISH_FREQ.ID`.
2. `logiBUS_PI_IDA.IN` (adapter) ⟶ `AD_SPLIT_2_UNGATED_FREQ.IN`; `AD_SPLIT_2_UNGATED_FREQ.OUT1` ⟶ `AD_SPLIT_2.IN`; `AD_SPLIT_2_UNGATED_FREQ.OUT2` ⟶ `AD_TO_AR_NUM.AD_IN`.
3. `AD_SPLIT_2.OUT1` ⟶ `AD_PUBLISH_COUNT.IN` (counter OPC-UA branch).
4. `AD_SPLIT_2.OUT2` ⟶ `AD_TO_AUDI.AD_IN`; `AD_TO_AUDI.AUDI_OUT` ⟶ `Q_NumericValue_COUNT.u32NewValue` (VT counter branch).
5. `AD_TO_AR_NUM.AR_OUT` ⟶ `FT_DERIV_10_AR.AR_IN`; `LIT_DerivK.OUT` ⟶ `FT_DERIV_10_AR.K`; `FT_DERIV_10_AR.AR_OUT` ⟶ `FT_PT1_AR.AR_IN` (with `initval_ATM.OUT` ⟶ `FT_PT1_AR.TM`); `FT_PT1_AR.AR_OUT` ⟶ `AR_SPLIT_2.IN` (frequency calculation branch).
6. `AR_SPLIT_2.OUT1` ⟶ `AR_PUBLISH_FREQ.IN` (frequency OPC-UA branch).
7. `AR_SPLIT_2.OUT2` ⟶ `Q_NumericValue_FREQ.rPhys` (frequency VT branch).
8. **Channel enable**: `Button_SWITCH.IND` ⟶ `E_T_FF_SR_SWITCH.CLK`; `AX_SUBSCRIBE_SWITCH.OUT` ⟶ `AX_RF_TRIG_SWITCH.QI`, `AX_RF_TRIG_SWITCH.ER` ⟶ `E_T_FF_SR_SWITCH.S`, `AX_RF_TRIG_SWITCH.EF` ⟶ `E_T_FF_SR_SWITCH.R`; `bDefaultEnabled` ⟶ `E_T_FF_SR_SWITCH.Q_INIT`; `E_T_FF_SR_SWITCH.Q` ⟶ `logiBUS_PI_IDA.QI`, `Q_BackgroundColour_EnableOk_STATUS.bEnable` and `AX_BOOL_TO_X_SWITCH.OUT` (⟶ `AX_PUBLISH_SWITCH.IN`); `E_T_FF_SR_SWITCH.EO` ⟶ `logiBUS_PI_IDA.INIT`, `Q_BackgroundColour_EnableOk_STATUS.REQ` and `AX_BOOL_TO_X_SWITCH.REQ`.
9. **Status**: `logiBUS_PI_IDA.QO` ⟶ `Q_BackgroundColour_EnableOk_STATUS.bOk` and `AX_BOOL_TO_X_STATUS.OUT` (⟶ `AX_PUBLISH_STATUS.IN`); `logiBUS_PI_IDA.INITO` ⟶ `Q_BackgroundColour_EnableOk_STATUS.REQ` and `AX_BOOL_TO_X_STATUS.REQ`; `u16ObjId_STATUS` ⟶ `Q_BackgroundColour_EnableOk_STATUS.u16ObjId`.
10. **Initialization chain**: `INIT` ⟶ `Q_NumericValue_COUNT.INIT` ⟶ `Q_NumericValue_FREQ.INIT` ⟶ `Q_BackgroundColour_EnableOk_STATUS.INIT` ⟶ `AD_PUBLISH_COUNT.INIT` ⟶ `Button_SWITCH.INIT` ⟶ `INITO` (and ⟶ `AND_QO.REQ`; additionally `E_MERGE_QO.EO` ⟶ `AND_QO.REQ` triggers it). In parallel, from `AD_PUBLISH_COUNT.INITO` the chain `AR_PUBLISH_FREQ` ⟶ `AX_SUBSCRIBE_SWITCH` ⟶ `AX_PUBLISH_SWITCH` ⟶ `AX_PUBLISH_STATUS` ⟶ `E_T_FF_SR_SWITCH` runs (each `INITO` ⟶ next `INIT`); `initval_ATM.INITO` ⟶ `FT_PT1_AR.INIT`.
11. **Quality**: `QI` ⟶ `.QI` of `Q_NumericValue_COUNT`, `Q_NumericValue_FREQ`, `AD_PUBLISH_COUNT`, `AR_PUBLISH_FREQ`, `E_T_FF_SR_SWITCH`, `Button_SWITCH`, `AX_SUBSCRIBE_SWITCH`, `AX_PUBLISH_SWITCH`, `AX_PUBLISH_STATUS` and `Q_BackgroundColour_EnableOk_STATUS`; their `QO` outputs ⟶ `AND_QO.IN1`..`IN10`; `AND_QO.OUT` ⟶ `QO`.

Technical Details

- The frequency is not delivered by the hardware FB itself but computed purely from the time derivative (`FT_DERIV_10_AR`, `Faktor=1.0`, `K=9`) of the raw, monotonically increasing counter value and smoothed with `FT_PT1_AR` (`TM=T#2s`) — the counter itself is never reset.
- `K` of `FT_DERIV_10_AR` is an AUI adapter socket and no longer a parameter (version 1.1); the value 9 comes via `LIT_DerivK`.

- `AD_TO_AR_NUM` is deliberately used instead of `AD_TO_AR` to obtain a numerically correct DWORD→REAL conversion rather than a bit reinterpretation (the same pitfall described for `F_AI_RAW_TO_PERCENT_AD`).
- Counter value and frequency use independent VT numeric fields (`u16ObjId_COUNTVAR` and `stObjFreq`) with different scaling — the frequency display additionally uses `NumericObjectPool_S` for physical scaling/decimals, the counter does not.
- The `QI` inputs of the internal blocks are no longer hard-wired to `TRUE` but are enabled via the `QI` of the SubApp (exception: `logiBUS_PI_IDA.QI` is tied to the channel enable).

Application Scenarios

- A flow meter or speed sensor where both the raw (unbounded, ever-increasing) counter value for diagnostic purposes and a derived, immediately interpretable frequency (e.g. Hz, RPM) should be displayed and transmitted; the channel can be switched on and off via a VT key or OPC-UA.

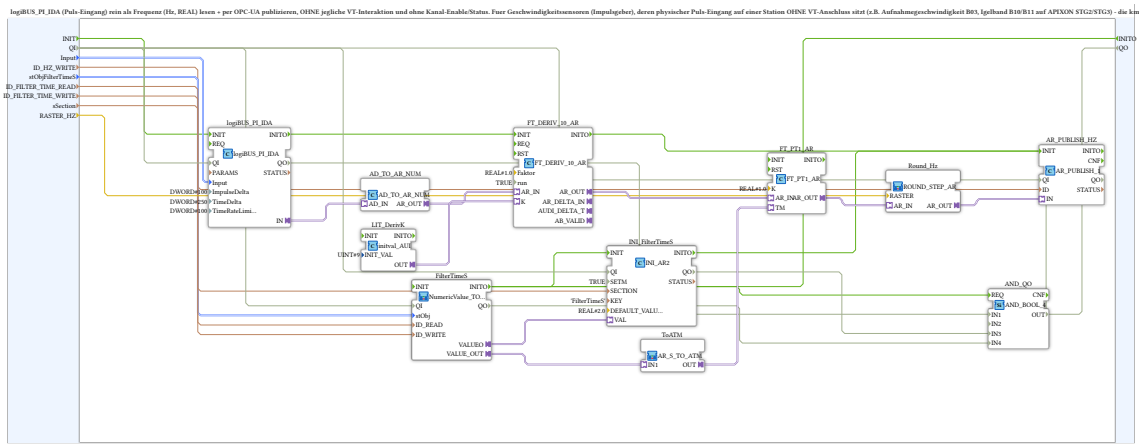
Summary

Complete VT and OPC-UA connection of a physical pulse input with a parallel raw counter value, a frequency computed from it by time derivative and PT1 smoothing, and a switchable channel enable with status display; `INIT`/`INITO` are chained through the services, `QO` is the AND of the services.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

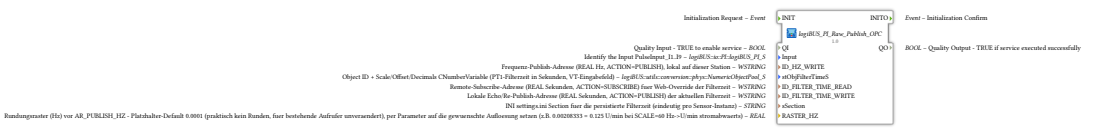
logiBUS_PI_Raw_Publish_OPC



• — ○ • — ○ • — ○ •

Introduction

`logiBUS_PI_Raw_Publish_OPC` reads a logiBUS pulse input (`logiBUS_PI_IDA`) purely as a **frequency (Hz, `REAL`)** and publishes it via OPC-UA - **without any VT interaction** and without channel enable/status. It is intended for speed sensors (pulse generators) whose physical pulse input sits on a station **without a VT connection** (e.g. pickup speed B03, hedgehog belt B10/B11 on APIXON STG2/STG3). The km/h conversion and VT display then run on the station with the VT connection, fed by a remote subscribe of this frequency (analogous to `logiBUS_AI_Raw_Publish_OPC`/`logiBUS_AI_Calibrate_Remote_IDA_OPC`).



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request

Event Outputs

- **INITO** (`EInit`): initialization confirm

Data Inputs

- **QI** (BOOL, default TRUE): quality input, TRUE enables the service
- **Input** (logiBUS::io::PI::logiBUS_PI_S, default logiBUS_PI::Invalid): Identifies the pulse input PulseInput_I1..I9
- **ID_HZ_WRITE** (WSTRING): Frequency publish address (REAL Hz, ACTION=PUBLISH), local on this station
- **stObjFilterTimeS** (NumericObjectPool_S): Object ID + scale/offset/decimals of the CNumberVariable for the PT1 filter time in seconds (VT input field)
- **ID_FILTER_TIME_READ** (WSTRING): Remote subscribe address (REAL seconds, ACTION=SUBSCRIBE) for the web override of the filter time
- **ID_FILTER_TIME_WRITE** (WSTRING): Local echo/re-publish address (REAL seconds, ACTION=PUBLISH) of the current filter time
- **sSection** (STRING): INI section in settings.ini for the persisted filter time (unique per sensor instance)
- **RASTER_HZ** (REAL, default 0.0001): Rounding grid (Hz) before AR_PUBLISH_HZ. The default corresponds to practically no rounding (unchanged for existing callers); set via parameter to the desired resolution (e.g. 0.00208333 = 0.125 rpm at SCALE=60 Hz ⟶ rpm downstream)

Data Outputs

- **QO** (BOOL): quality output, AND combination (AND_QO, AND_BOOL_4) of the QO outputs of logiBUS_PI_IDA, AR_PUBLISH_HZ, INI_FilterTimeS and FilterTimeS

Adapters

This function block has no adapters at its interface; AR/AD adapters are used internally.

Function Blocks (FBs) Used

- **logiBUS_PI_IDA**: logiBUS::io::PI::logiBUS_PI_IDA (ImpulseDelta=100, TimeDelta=250, TimeRateLimit=100) - pulse input as an AD adapter
- **AD_TO_AR_NUM**: adapter::conversion::unidirectional::AD_TO_AR_NUM - converts the count value to REAL
- **FT_DERIV_10_AR**: OSCAT_adapter::Control::FT_DERIV_10_AR (Faktor=1.0, run=TRUE) - derivative (pulses per time = frequency)
- **LIT_DerivK**: initval_AUI (INIT_VAL=9) - constant K of the derivative
- **FT_PT1_AR**: OSCAT_adapter::Control::FT_PT1_AR (K=1.0) - PT1 filter
- **Round_Hz**: OSCAT_adapter::Mathematical::ROUND_STEP_AR - rounds to the grid RASTER_HZ
- **AR_PUBLISH_HZ**: adapter::net::AR_PUBLISH_1 - publishes the frequency
- **FilterTimeS**: SubApp MyLib::sys::NumericValue_TO_AR2_OPC - filter time from VT input field or OPC-UA
- **INI_FilterTimeS**: eclipse4diac::storage::INI_AR2 (SETM=TRUE, KEY='FilterTimeS', default 2.0) - persists the filter time
- **ToATM**: SubApp MyLib::sys::AR_S_TO_ATM - converts seconds into an ATM time value
- **AND_QO**: iec61131::booleanOperators::AND_BOOL_4 - combines the QO outputs of logiBUS_PI_IDA, AR_PUBLISH_HZ, INI_FilterTimeS and FilterTimeS into QO

Program Flow and Connections

1. Input ⟶ logiBUS_PI_IDA.Input; logiBUS_PI_IDA.IN ⟶ AD_TO_AR_NUM.AD_IN.
2. AD_TO_AR_NUM.AR_OUT ⟶ FT_DERIV_10_AR.AR_IN; LIT_DerivK.OUT ⟶ FT_DERIV_10_AR.K.
3. FT_DERIV_10_AR.AR_OUT ⟶ FT_PT1_AR.AR_IN.
4. Filter time: FilterTimeS.VALUEO ⟶ INI_FilterTimeS.VAL (persistence); FilterTimeS.VALUE_OUT ⟶ ToATM.IN1; ToATM.OUT ⟶ FT_PT1_AR.TM.
5. FT_PT1_AR.AR_OUT ⟶ Round_Hz.AR_IN (RASTER_HZ ⟶ Round_Hz.RASTER) ⟶ AR_PUBLISH_HZ.IN.
6. ID_HZ_WRITE ⟶ AR_PUBLISH_HZ.ID.
7. Initialization: INIT ⟶ logiBUS_PI_IDA.INIT; logiBUS_PI_IDA.INITO ⟶ AR_PUBLISH_HZ.INIT; AR_PUBLISH_HZ.INITO ⟶ INI_FilterTimeS.INIT; INI_FilterTimeS.INITO ⟶ FilterTimeS.INIT; FilterTimeS.INITO ⟶ INITO and ⟶ AND_QO.REQ.
8. Quality: QI is distributed to logiBUS_PI_IDA, AR_PUBLISH_HZ, INI_FilterTimeS and FilterTimeS (no hard-wired QI parameter any more); their QO ⟶ AND_QO.IN1 to AND_QO.IN4 (in this order); AND_QO.OUT ⟶ QO.

Technical Details

- No VT interaction for the frequency itself, no channel enable and no status.

- The filter time is INI-persisted and can be changed via the VT input field or OPC-UA.
- `RASTER_HZ` can be used to limit the resolution in order to avoid unnecessary publish events.

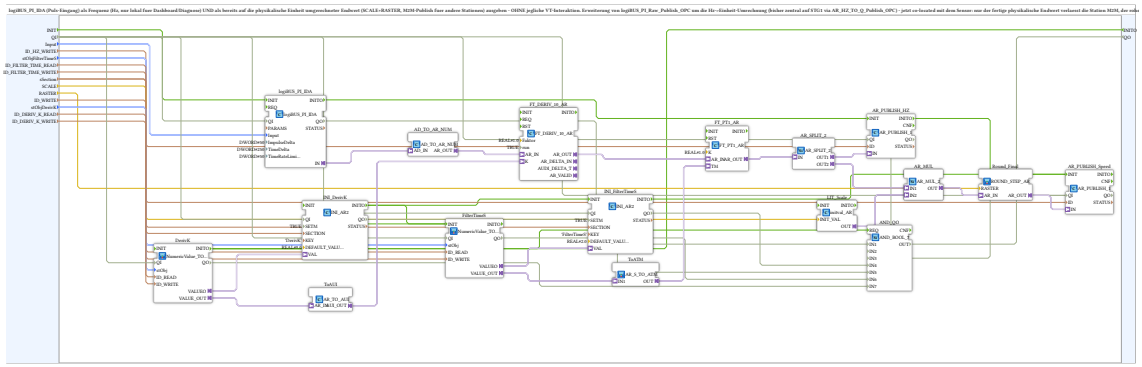
Application Scenarios

- Speed/rotation sensors on stations without a VT, whose frequency is transmitted via OPC-UA to the VT station (e.g. complemented by `AR_HZ_TO_Q_Publish_OPC` on the VT station)

Summary

`logiBUS_PI_Raw_Publish_OPC` produces a filtered, rounded frequency in Hz from a pulse input and publishes it via OPC-UA.

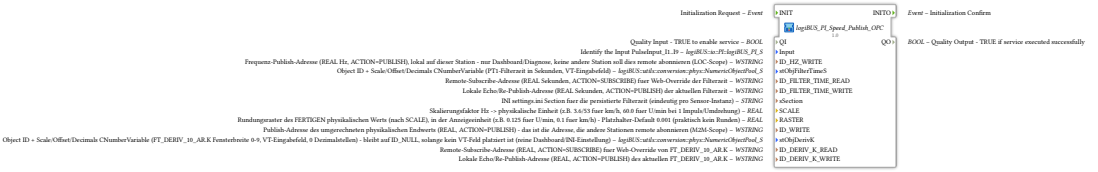
logiBUS_PI_Speed_Publish_OPC



• — ○ • — ○ • — ○ •

Introduction

`logiBUS_PI_Speed_Publish_OPC` outputs a pulse input (`logiBUS_PI_IDA`) in two ways: as a **frequency in Hz** (local only, for dashboard/diagnostics) and as a **final value already converted to the physical unit** (SCALE + RASTER, M2M publish for other stations). There is **no** VT interaction. The block extends `logiBUS_PI_Raw_Publish_OPC` with the Hz-to-unit conversion that used to run centrally on STG1 (`AR_HZ_TO_Q_Publish_OPC`) and now sits at the sensor: only the finished physical final value leaves the station via M2M, the raw Hz value stays local.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization Request

Event Outputs

- **INITO** (`EInit`): Initialization Confirm

Data Inputs

- **QI** (BOOL, default `TRUE`): Quality Input - TRUE to enable the service
- **Input** (`logiBUS::io::PI::logiBUS_PI_S`): selection of the pulse input `PulseInput_I1..I9` (default `logiBUS_PI::Invalid`)
- **ID_HZ_WRITE** (WSTRING): frequency publish address (REAL Hz, `ACTION=PUBLISH`), local; dashboard/diagnostics only, no other station should subscribe to it remotely (scope LOC)
- **stObjFilterTimeS** (`NumericObjectPool_S`): object ID + scale/offset/decimals of the CNumberVariable for the PT1 filter time in seconds (VT input field); default `u8Decimals := 1`
- **ID_FILTER_TIME_READ** (WSTRING): remote subscribe address (REAL seconds, `ACTION=SUBSCRIBE`) for a web override of the filter time

- **ID_FILTER_TIME_WRITE** (WSTRING): local echo/republish address (REAL seconds, ACTION=PUBLISH) of the current filter time
- **sSection** (STRING): INI section in settings.ini for the persisted values (filter time and DerivK), unique per sensor instance
- **SCALE** (REAL): scaling factor Hz → physical unit (e.g. 3.6/53 for km/h, 60.0 for rpm at 1 pulse/revolution)
- **RASTER** (REAL): rounding grid of the **finished** physical value (after SCALE) in the display unit (e.g. 0.125 for rpm, 0.1 for km/h); placeholder default 0.001 (practically no rounding)
- **ID_WRITE** (WSTRING): publish address of the converted physical final value (REAL, ACTION=PUBLISH); other stations subscribe to this address remotely (scope M2M)
- **stObjDerivK** (NumericObjectPool_S): object ID + scale/offset/decimals of the CNumberVariable for FT_DERIV_10_AR.K (window width 0-9, 0 decimals); stays at ID_NULL as long as no VT field is placed (dashboard/INI setting only)
- **ID_DERIV_K_READ** (WSTRING): remote subscribe address (REAL, ACTION=SUBSCRIBE) for a web override of FT_DERIV_10_AR.K
- **ID_DERIV_K_WRITE** (WSTRING): local echo/republish address (REAL, ACTION=PUBLISH) of the current FT_DERIV_10_AR.K

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if the service executed successfully (AND of the QO of logiBUS_PI_IDA, AR_PUBLISH_HZ, AR_PUBLISH_Speed, INI_FilterTimeS, INI_DerivK, FilterTimeS and DerivK; block AND_QO, type iec61131::booleanOperators::AND_BOOL_7)

Adapters

No adapters are used.

Functionality

Measurement chain (frequency): logiBUS_PI_IDA (logiBUS::io::PI::logiBUS_PI_IDA, QI from the block input, ImpulseDelta = DWORD#50, TimeDelta = DWORD#250, TimeRateLimit = DWORD#50) → AD_TO_AR_NUM → FT_DERIV_10_AR (OSCAT_adapter::Control, Faktor = 1.0, run = TRUE) → FT_PT1_AR (K = 1.0) → AR_SPLIT_2.

Two output paths from AR_SPLIT_2:

- OUT1 → AR_PUBLISH_HZ (AR_PUBLISH_1, QI from the block input): publishes the frequency in Hz to ID_HZ_WRITE (local, unrounded).
- OUT2 → AR_MUL.IN1 (AR_MUL_2), LIT_Scale.OUT → AR_MUL.IN2 (LIT_Scale is an initval_AR with INIT_VAL from SCALE) → Round_Final (OSCAT_adapter::Mathematical::ROUND_STEP_AR, RASTER) → AR_PUBLISH_Speed (AR_PUBLISH_1, QI from the block input): publishes the converted final value, rounded to RASTER, to ID_WRITE.

Adjustable PT1 filter time: FilterTimeS (MyLib::sys::NumericValue_TO_AR2_OPC stObjFilterTimeS, ID_FILTER_TIME_READ, ID_FILTER_TIME_WRITE) supplies the filter time in seconds. FilterTimeS.VALUEO ↔ INI_FilterTimeS.VAL (eclipse4diac::storage::INI_AR2, QI from the block input, SETM = TRUE, KEY = 'FilterTimeS', DEFAULT_VALUE = REAL#2.0, SECTION from sSection) persists it. FilterTimeS.VALUE_OUT → ToATM.IN1 (MyLib::sys::AR_S_TO_ATM), ToATM.OUT → FT_PT1_AR.TM.

Adjustable derivative window width: DerivK (NumericValue_TO_AR2_OPC; stObjDerivK, ID_DERIV_K_READ, ID_DERIV_K_WRITE) supplies K. DerivK.VALUEO ↔ INI_DerivK.VAL (INI_AR2, KEY = 'DerivK', DEFAULT_VALUE = REAL#9.0, SECTION from sSection) persists the value. DerivK.VALUE_OUT → ToAUI.AR_IN (AR_TO_AUI), ToAUI.AUI_OUT → FT_DERIV_10_AR.K.

Initialization and status: The initialization chain is INIT → logiBUS_PI_IDA.INIT, logiBUS_PI_IDA.INITO → AR_PUBLISH_HZ.INIT, AR_PUBLISH_HZ.INITO → AR_PUBLISH_Speed.INIT, AR_PUBLISH_Speed.INITO → INI_FilterTimeS.INIT, INI_FilterTimeS.INITO → INI_DerivK.INIT, INI_DerivK.INITO → FilterTimeS.INIT, FilterTimeS.INITO → DerivK.INIT, DerivK.INITO → INITO and → AND_QO.REQ. QI is forwarded to logiBUS_PI_IDA, AR_PUBLISH_HZ, AR_PUBLISH_Speed, INI_FilterTimeS, INI_DerivK, FilterTimeS and DerivK; their QO outputs → AND_QO.IN1..IN7, AND_QO.OUT → QO.

Technical Details

- The Hz value (ID_HZ_WRITE) is deliberately left unrounded and purely local (scope LOC); only ID_WRITE is intended for M2M subscribers

- RASTER rounds the **finished** physical value after the SCALE multiplication (no longer in Hz as RASTER_HZ used to do in logiBUS_PI_Raw_Publish_OPC), so intuitive values arise directly in the display unit
- Filter time (FilterTimeS) and window width (DerivK) are adjustable via VT field, web override and INI persistence; K of FT_DERIV_10_AR is itself an AUI adapter socket, so the earlier unpacking workaround is no longer needed
- The configuration applies automatically to all five pulse sensors that use this common block (pickup B03, hedgehog belt 1/2 B10/B11, PTO B21, saw blade B08)
- Both INI entries share the same sSection but have different KEYs (FilterTimeS, DerivK)

Application Scenarios

- Speed or rotational speed measurement (km/h, rpm) on stations without a VT connection with cross-station provision of the finished value

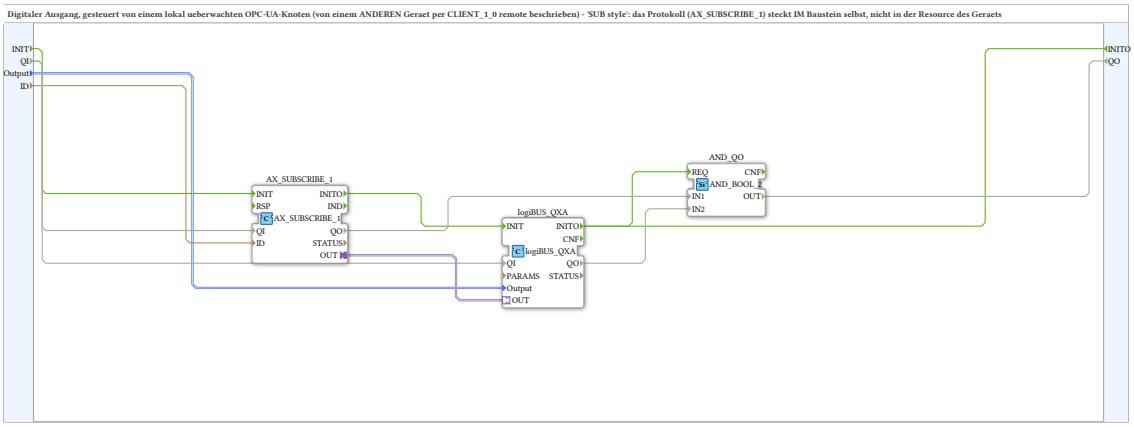
⚖️ Comparison with Similar Blocks

In contrast to logiBUS_PI_Raw_Publish_OPC, this block rounds in the display unit grid and additionally supplies the converted final value. The subscribing counterpart is AR_M2M_Subscribe_TO_Q_Publish_OPC.

Conclusion

logiBUS_PI_Speed_Publish_OPC covers the complete sensor side of a speed/rotational speed measurement: frequency determination with an adjustable filter, conversion and rounding to the display unit as well as local and M2M publication.

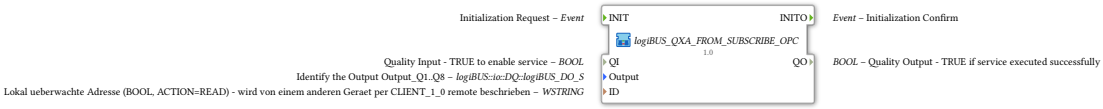
logiBUS_QXA_FROM_SUBSCRIBE_OPC



• — ○ • — ○ • — ○ •

Introduction

logiBUS_QXA_FROM_SUBSCRIBE_OPC drives a digital output (logiBUS_QXA) from a locally-subscribed OPC-UA node that is remotely written by ANOTHER device via AX_CLIENT_1_0 - "Krauternter style" (per the source comment): the protocol (AX_SUBSCRIBE_1) lives INSIDE the block itself, not in the device's resource.



Function blocks used

- **AX_SUBSCRIBE_1** (adapter::net::AX_SUBSCRIBE_1): subscribes to the locally-monitored address ID (BOOL, written remotely).
- **logiBUS_QXA** (logiBUS::io::DQ::logiBUS_QXA): physical digital output, identified via Output_Q1..Q8.
- **AND_QO** (iec61131::booleanOperators::AND_BOOL_2): combines AX_SUBSCRIBE_1.QO and logiBUS_QXA.QO into the SubApp output QO.

Initialization and quality

- Events: INIT (EInit, input) and INITO (EInit, output, comes from logiBUS_QXA.INITO).

- Data input `QI` (BOOL, default `TRUE`) and data output `QO` (BOOL); `AX_SUBSCRIBE_1` and `logiBUS_QXA` no longer carry a fixed `QI=TRUE`.
- Initialization: `INIT` ⟶ `AX_SUBSCRIBE_1.INIT`, `AX_SUBSCRIBE_1.INITO` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
- Quality: `QI` ⟶ `AX_SUBSCRIBE_1.QI` and `logiBUS_QXA.QI`; `AX_SUBSCRIBE_1.QO` ⟶ `AND_QO.IN1`, `logiBUS_QXA.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

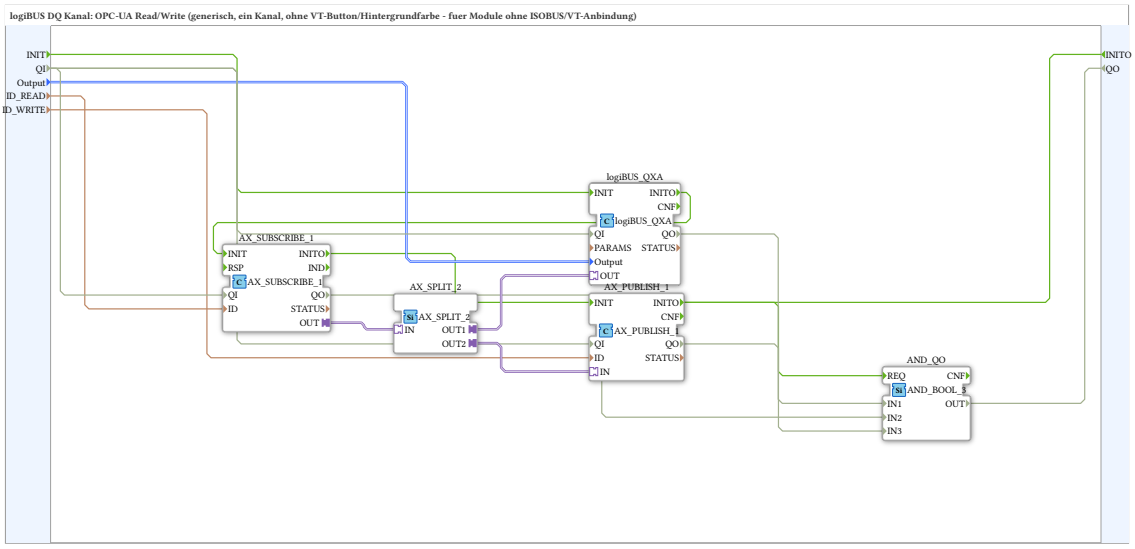
Summary

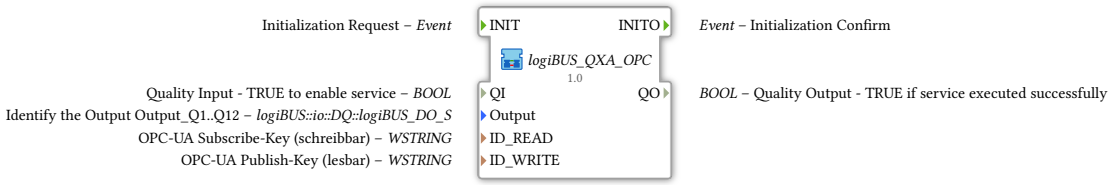
Single-block bridge from a locally-subscribed OPC-UA value to a physical digital output - counterpart to `logiBUS_IXA_TO_CLIENT_OPC` on the sending side.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

logiBUS_QXA_OPC





• — ○ • — ○ • — ○ •

Introduction

`logiBUS_QXA_OPC` is a composite subapplication designed to provide a generic single-channel OPC UA read/write access to a logiBUS digital output (DQ) module. It is specifically intended for modules that lack ISOBUS or Virtual Terminal (VT) support, offering a standard OPC UA interface for remote control and monitoring. The subapp internally couples a `logiBUS_QXA` FB with OPC UA subscribe and publish adapters, enabling bidirectional communication over the network.

Interface Structure

The subapp exposes the events `INIT`/`INITO`, the data `QI`/`QO` and three further data input variables. It has no adapter interfaces at its boundary; the actual communication is handled internally.

Event Inputs

- **INIT** (`EInit`): initialization request; starts the init chain at `logiBUS_QXA.INIT`

Event Outputs

- **INITO** (`EInit`): initialization confirmation; comes from `AX_PUBLISH_1.INITO`, the end of the init chain

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	TRUE	Quality input, enables the service; passed on to logiBUS_QXA, AX_SUBSCRIBE_1 and AX_PUBLISH_1.
Output	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Identifies the target output channel (e.g., Output_Q1..Q12).
ID_READ	WSTRING	–	OPC UA subscribe key (used to receive commands or status requests).
ID_WRITE	WSTRING	–	OPC UA publish key (used to send data back to the network).

Data Outputs

- **QO** (BOOL): quality output, AND combination (AND_QO, AND_BOOL_3) of the QO outputs of logiBUS_QXA, AX_SUBSCRIBE_1 and AX_PUBLISH_1

Adapters

None at the subapp level. Internally, the following adapter instances are utilised:

- AX_SUBSCRIBE_1 – subscribes to an OPC UA data source.
- AX_PUBLISH_1 – publishes data to an OPC UA target.
- AX_SPLIT_2 – splits the incoming event stream for parallel processing.

Functionality

The subapp acts as a bridge between an OPC UA client and a physical logiBUS digital output. The external inputs configure the behaviour:

- The Output input selects which specific DQ output channel is to be controlled (e.g., Q1…Q12).
- ID_READ defines the OPC UA subscribe key – incoming events on this topic are processed.
- ID_WRITE defines the OPC UA publish key – outgoing events are sent to this topic.

Internally, the OPC UA subscribe adapter receives events. These events are then split into two paths:

1. The first path goes directly to the logiBUS_QXA FB, which interprets the event as a control command and sets the selected output accordingly.
2. The second path is forwarded to the OPC UA publish adapter, allowing the same event to be echoed back or used for acknowledgment/status purposes.

Initialization and quality: INIT → logiBUS_QXA.INIT, logiBUS_QXA.INITO → AX_SUBSCRIBE_1.INIT, AX_SUBSCRIBE_1.INITO → AX_PUBLISH_1.INIT, AX_PUBLISH_1.INITO → INITO and AND_QO.REQ. QI is passed on to logiBUS_QXA.QI, AX_SUBSCRIBE_1.QI and AX_PUBLISH_1.QI; their QO outputs go in this order to AND_QO.IN1 to AND_QO.IN3 (iec61131::booleanOperators::AND_BOOL_3), AND_QO.OUT → QO.

The publish adapter can also transmit status information back to the OPC UA server, enabling read access to the current state of the output. The separation of read and write keys allows for fine-grained access control.

Technical Features

- **Single-channel configuration**: The subapp handles one DQ output at a time, selected via the Output input.
- **OPC UA integration**: Uses standard 4diac adapter types AX_SUBSCRIBE_1 and AX_PUBLISH_1 for seamless OPC UA connectivity.
- **Event splitting**: AX_SPLIT_2 ensures that an incoming event can be both processed locally and forwarded to the publishing side without loss.

- **Flexible addressing**: The keys `ID_READ` and `ID_WRITE` are runtime-writable, allowing dynamic re-binding to different OPC UA nodes.
- **No VT dependency**: Works with modules that do not implement ISOBUS VT, making it suitable for simpler or custom hardware.

State Overview

The subapp does not maintain an explicit state machine. The internal FBs (`logiBUS_QXA`, the adapters, and the splitter) operate asynchronously. However, the overall behaviour can be described by the following data flow:

- **Idle**: Waiting for incoming OPC UA events on the subscribe key.
- **Event received**: The subscribe adapter emits an event; the splitter forwards it in parallel to the `logiBUS_QXA` FB (which updates the output) and to the publish adapter (which may send an acknowledgment or status).
- **Publish operation**: The publish adapter sends data on the configured key, which can be used for read-back of the current output state or for broadcasting the received command.

Application Scenarios

- Remote control of logiBUS digital outputs via OPC UA in industrial automation environments.
- Integration of logiBUS modules into OPC UA-based supervisory control and data acquisition (SCADA) systems.
- Use cases where the module has no ISOBUS or VT interface, and a simple OPC UA channel is required.
- Generic deployment across multiple DQ channels by re-configuring the `Output` input per instance.

Comparison with Similar Blocks

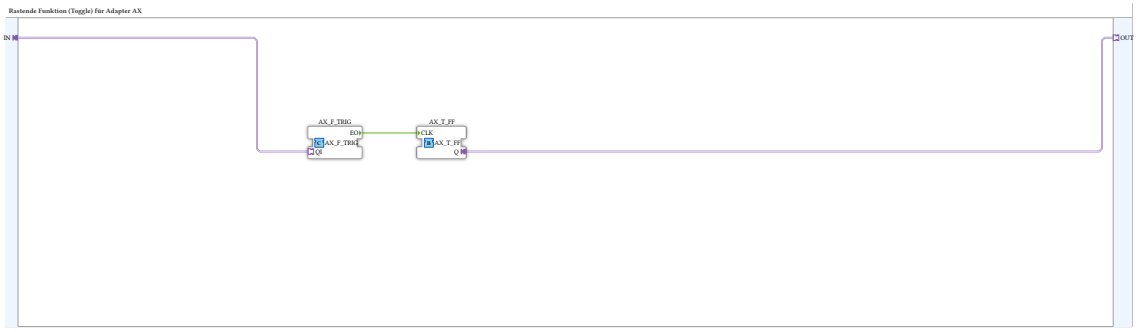
- **logiBUS_QXA** (the internal FB): This is the raw DQ control block. `logiBUS_QXA_OPC` extends it with OPC UA connectivity and a fixed single-channel configuration.
- **logiBUS_QXA_VT**: A variant that likely includes VT button and background color support – `logiBUS_QXA_OPC` omits these for modules without VT, reducing complexity.
- **Generic OPC UA I/O blocks**: Many OPC UA adapters exist, but this subapp encapsulates both subscribe and publish directions specifically for logiBUS DQ control, saving integration time.

Conclusion

`logiBUS_QXA_OPC` provides a compact, reusable solution for integrating logiBUS digital outputs with OPC UA. Its generic single-channel design makes it adaptable to various modules, while the separation of read and write keys offers flexible access control. The use of standard 4diac adapters ensures easy integration into existing projects. This subapp is ideal for environments where a lightweight, network-based control path is required.

Rastend





• – ◦ • – ◦ • – ◦ •

Introduction

This exercise implements a latching switching function that is triggered when a signal is released. The sub-application uses adapters for signal processing and event control to achieve reliable latching functionality.

Function Blocks Used (FBs)

Sub-modules: rastend

- **Type**: SubAppType
- **Internal FBs Used**:
  - **E_T_FF**: iec61499::events::E_T_FF
    - Event Input: CLK
    - Event Output: EO
    - Data Output: Q
  - **E_SWITCH**: iec61499::events::E_SWITCH
    - Event Input: EI
    - Event Output: EO0
    - Data Input: G
  - **AX_X_TO_BOOL**: adapter::conversion::unidirectional::AX_X_TO_BOOL
    - Adapter Input: AX_IN
    - Event Output: CNF
    - Data Input: IN
  - **AX_BOOL_TO_X**: adapter::conversion::unidirectional::AX_BOOL_TO_X
    - Event Input: REQ
    - Adapter Output: AX_OUT
    - Data Input: OUT
- **Functionality**: The subapplication converts incoming AX signals into Boolean values, processes them using a toggle flip-flop, and outputs the result back as an AX signal.

Program Flow and Connections

The program flow follows this signal processing chain:

1. **Input Signal Processing**: The incoming AX signal is passed through the IN adapter to AX_X_TO_BOOL.
2. **Signal Conversion**: AX_X_TO_BOOL converts the AX signal into a Boolean value.
3. **Switch Control**: The Boolean value controls E_SWITCH, which forwards events to the toggle flip-flop.
4. **Latching Function**: E_T_FF implements the actual latching function and stores the state.
5. **Output Signal Generation**: AX_BOOL_TO_X converts the Boolean output value back into an AX signal.

Connections in Detail:

- Event Connections: E_SWITCH.EO0 → E_T_FF.CLK, E_T_FF.EO → AX_BOOL_TO_X.REQ, AX_X_TO_BOOL.CNF → E_SWITCH.EI
- Data connections: E_T_FF.Q → AX_BOOL_TO_X.OUT, AX_X_TO_BOOL.IN → E_SWITCH.G
- Adapter connections: IN → AX_X_TO_BOOL.AX_IN, AX_BOOL_TO_X.AX_OUT → OUT

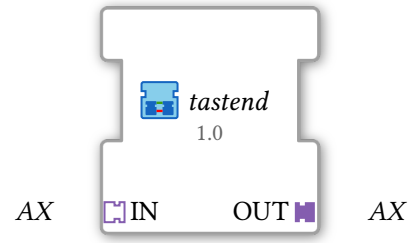
Summary

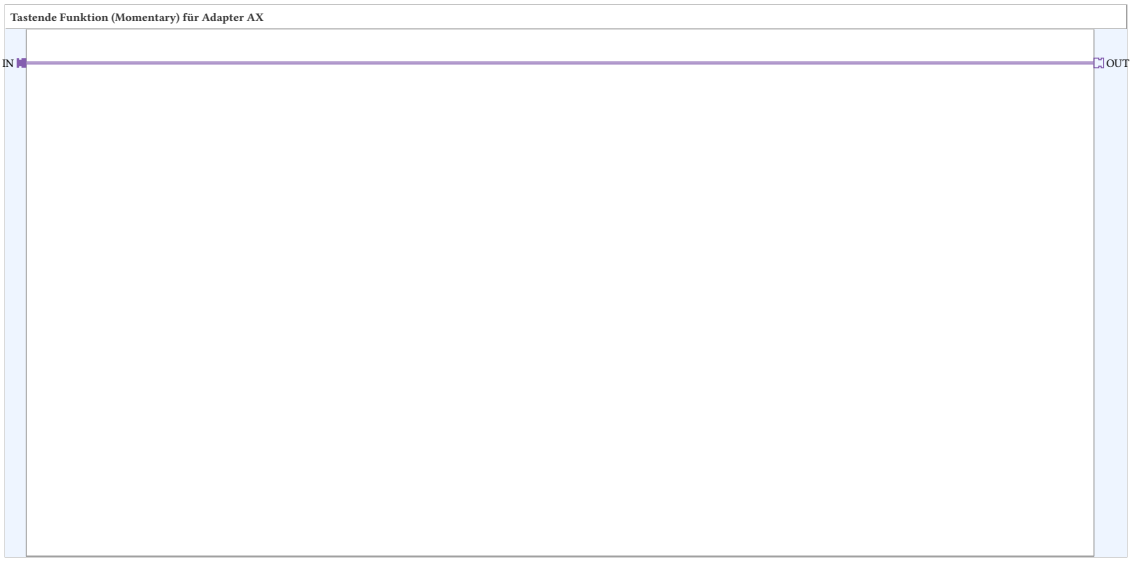
This exercise demonstrates the implementation of a latching switching function with adapters for signal conversion. It shows the use of toggle flip-flops for state storage and structured signal processing via various function blocks. The solution is particularly suitable for applications where a signal should remain active after being released.

🛠️ Related Exercises

- Uebung_103
- Uebung_103c
- Uebung_103c2

Touch-sensitive





• – ◦ • – ◦ • – ◦ •

Introduction

This exercise deals with creating and using a sub-application type in the 4diac IDE. The “touch-sensitive” function block serves as a sub-application and implements a unidirectional adapter connection between input and output adapters.

Function Blocks (FBs) Used

Sub-block: touch-sensitive

- **Type**: SubAppType
- **Internal FBs Used**: No internal FBs present
- **Functionality**: The “touch-sensitive” sub-block establishes an adapter connection between an input adapter (IN) and an output adapter (OUT). Both adapters use the type “adapter::types::unidirectional::AX” and enable unidirectional data transmission.

Program Flow and Connections

The “tactile” sub-module has the following interfaces:

- **Input Adapter**: IN (Type: adapter::types::unidirectional::AX)
- **Output Adapter**: OUT (Type: adapter::types::unidirectional::AX)

The connection is made directly from the input adapter IN to the output adapter OUT. This configuration allows for easy forwarding of signals or data through the sub-module.

Difficulty Level: Easy

Required Prior Knowledge: Basic knowledge of 4diac-IDE, understanding of adapter connections and sub-applications

Starting the Exercise: The sub-module can be integrated into larger applications and controlled via the defined adapter interfaces.

Summary

The “tactile” exercise demonstrates the creation of a simple sub-application type with adapter connections. It shows the basic structure of sub-modules in 4diac-IDE and their integration into larger control applications using standardized adapter interfaces.

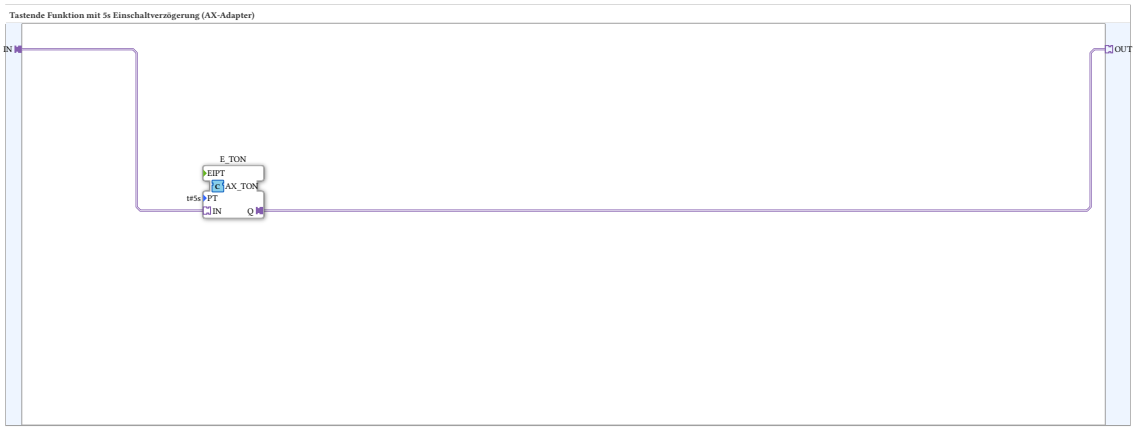
🛠️ Related exercises

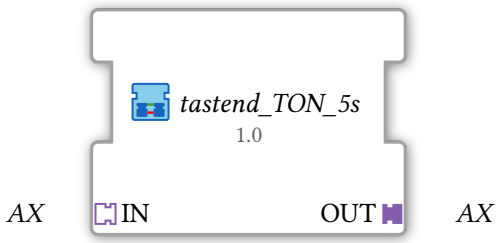
- Uebung_103
- Uebung_103c
- Uebung_103c2

🌐 Matching topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

tactile_TON_5s





• – ○ • – ○ • – ○ •

Introduction

This exercise demonstrates the use of a time-delay relay (timer on-delay) with a fixed time setting of 5 seconds. The sub-application serves as a reusable component for time-delayed signal processing.

Function Blocks Used (FBs)

Sub-Blocks: tastend_TON_5s

- **Type**: SubAppType
- **Internal FBs Used**:
- **AX_X_TO_BOOL**: adapter::conversion::unidirectional::AX_X_TO_BOOL
- Parameters: None
- Event Output/Input: CNF (Output), REQ (Input)
- Data Output/Input: IN (Input), OUT (Output)
- **E_TON**: iec61499::events::timers::E_TON
- Parameters: PT = t#5s
- Event Output/Input: CNF (Output), REQ (Input)
- Data Output/Input: IN (Input), Q (Output)
- **AX_BOOL_TO_X**: adapter::conversion::unidirectional::AX_BOOL_TO_X

- Parameters: None
- Event output/input: REQ (input), CNF (output)

Data output/input: OUT (input), AX_OUT (output)
Functionality:
The subapplication converts an incoming adapter signal (IN) into a Boolean value via AX_X_TO_BOOL, which is then passed to the E_TON timer. The timer waits 5 seconds before activating the output signal via Q. This signal is then converted back into an adapter signal (OUT) via AX_BOOL_TO_X.

Program Flow and Connections

Adapter Connections:

- Input: IN ⟶ AX_X_TO_BOOL.AX_IN
- Output: AX_BOOL_TO_X.AX_OUT ⟶ OUT

Event Connections:

- AX_X_TO_BOOL.CNF ⟶ E_TON.REQ
- E_TON.CNF ⟶ AX_BOOL_TO_X.REQ

Data Connections:

- AX_X_TO_BOOL.IN ⟶ E_TON.IN
- E_TON.Q ⟶ AX_BOOL_TO_X.OUT

Learning Objectives:

- Understanding of Timer Function Blocks (E_TON)
- Working with Adapter Conversion Blocks
- Building Reusable Subapplications
- Time-Controlled Signal Processing

Difficulty Level: Beginner
Required Prior Knowledge: Basic knowledge of the 4diac IDE, understanding of function blocks and Adapters
Start of the exercise: The sub-application can be integrated into larger applications where a 5-second delay is required.

Summary

The sub-application tastend_TON_5s provides a compact solution for time-delayed signal processing. Thanks to its fixed 5-second delay and the use of standardized adapter interfaces, it can be easily integrated into various control applications. Its clear structure, including signal conversion, timer functionality, and feedback, makes its operation transparent and comprehensible.

🛠️ Related Exercises

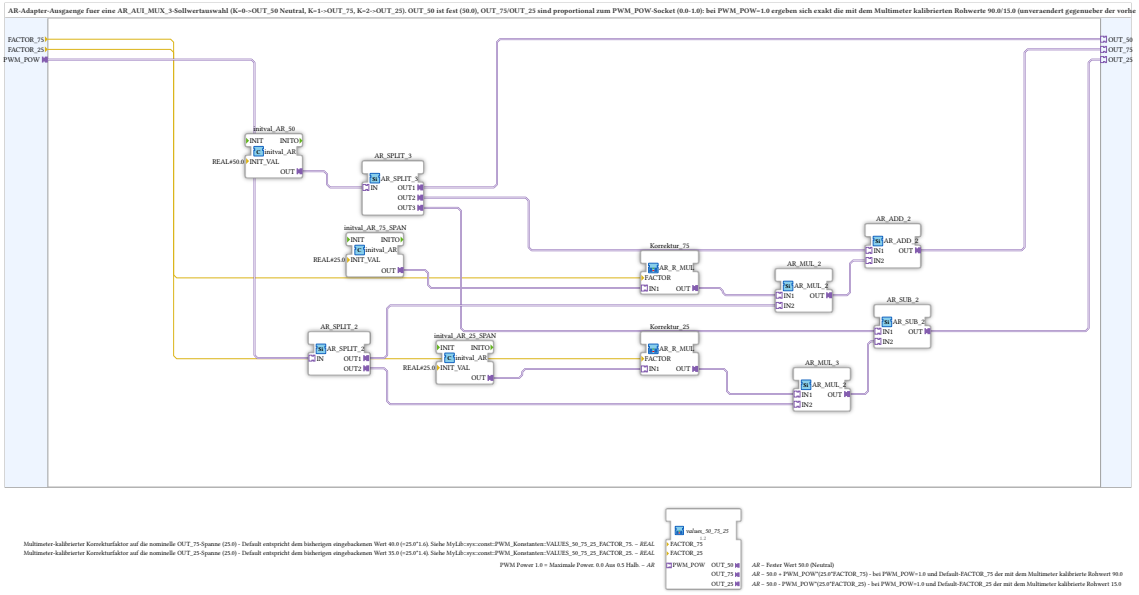
- Exercise_103
- Exercise_103c

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

values_50_75_25



• — ○ • — ○ • — ○ •

Introduction

The **values_50_75_25** subapplication provides three constant REAL values (50.0, 75.0, and 25.0) as outputs via unidirectional AR adapters. It is designed to supply fixed setpoints to a downstream selection logic, such as an `AR_AUI_MUX_3` block, where the selection index `K` maps to one of the three constant values. The subapplication encapsulates three independent constant sources, each implemented using an internal `initval_AR` function block.

Interface Structure

The subapplication exposes only adapter outputs; it has no event inputs, event outputs, data inputs, or data outputs.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter Name	Type	Description
OUT_50	unidirectional::AR	Fixed value 50.0
OUT_75	unidirectional::AR	Fixed value 75.0
OUT_25	unidirectional::AR	Fixed value 25.0

All adapters are of type `adapter::types::unidirectional::AR`, representing a unidirectional connection that carries a REAL value.

Functionality

The subapplication internally instantiates three `initval_AR` function blocks, each parameterized with a specific initial value:

- `initval_AR_50` with `INIT_VAL = 50.0`
- `initval_AR_75` with `INIT_VAL = 75.0`
- `initval_AR_25` with `INIT_VAL = 25.0`

Each block writes its constant to its `OUT` adapter, which is directly connected to the corresponding subapplication adapter output. Thus, the subapplication continually provides the three predefined constant values on its outputs.

Technical Features

- **Pure constant output:** No dynamic calculation or external inputs are involved; the values are fixed at design time.
- **Unidirectional adapters:** The AR adapter type is specifically designed for continuous value transfer from producer to consumer without handshaking.
- **Parameterization:** The internal `initval_AR` blocks are simple and robust, ensuring immediate availability of the output values.
- **Separation of concerns:** The subapplication isolates constant generation, making the source of different setpoints explicit and exchangeable.

State Overview

This subapplication is stateless. The internal function blocks are combinatorial in nature (or initial value based) and do not maintain any runtime state. Outputs are available as soon as the subapplication is activated.

Application Scenarios

- **Setpoint selection for automation tasks:** Used in conjunction with a multiplexer (e.g., `AR_AUI_MUX_3`) where different constant outputs (50, 75, 25) are selected via an index.
- **Test and simulation environments:** Provide fixed reference values for validating control algorithms or monitoring logic.
- **Reusable constant library:** The subapplication can be instantiated multiple times in a project to supply identical constant values without duplicating individual function blocks.

Comparison with Similar Blocks

- **vs. simple constant function block:** A single constant block provides only one value; this subapplication encapsulates three related constants with dedicated outputs, improving readability and maintainability.
- **vs. parameterized subapplication:** If the constants were dynamic, a subapplication would require data inputs. Here, because the values are fixed, the adapter-only interface is sufficient and simplifies external wiring.
- **vs. direct use of internal blocks:** By hiding the internal implementation, the subapplication offers a clean interface and makes the intended logic (three distinct setpoints) immediately obvious to other engineers.

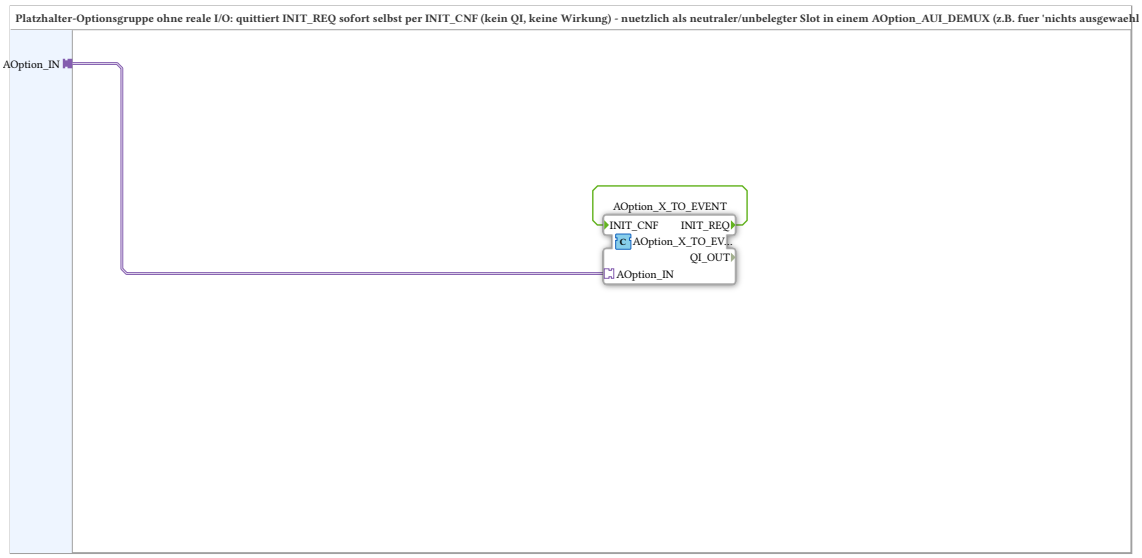
Conclusion

The `values_50_75_25` subapplication is a straightforward solution for providing three fixed REAL constants via unidirectional adapters. Its simplicity, clear naming, and encapsulation make it ideal for use in selector or multiplexer scenarios where predefined setpoints are required. With no dynamic parts, it is easy to integrate and guarantees consistent output values across the entire runtime.

2.1.5 MyLib_AX_Option

2.1.5.1 sys

DummyGroup



• – ◦ • – ◦ • – ◦ •

Introduction

`DummyGroup` is a placeholder option group without real I/O. It acknowledges `INIT_REQ` immediately by itself with `INIT_CNF` (no `QI` effect, no action). It serves as a neutral, unassigned slot in an `AOption_AUI_DEMUX`, e.g. for "nothing selected" at `K=1` when real options only start at `K=2`. This avoids an override literal for "which number means off".



Used Function Blocks (FBs)

Sub-blocks: DummyGroup

- **Type**: SubAppType
- **Interface**: socket `AOption_IN` of type `Option::AOption`.
- **Internal FBs used**:
  - **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`.

Program Flow and Connections

- `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`
- `AOption_X_TO_EVENT.INIT_REQ` -> `AOption_X_TO_EVENT.INIT_CNF` (self-acknowledgement)

`QI_OUT` remains unused.

Technical Features

- No I/O blocks, no effect on the application.

Application Scenarios

- Neutral slot in an option DEMUX, e.g. `AOption_AUI_DEMUX_n`.

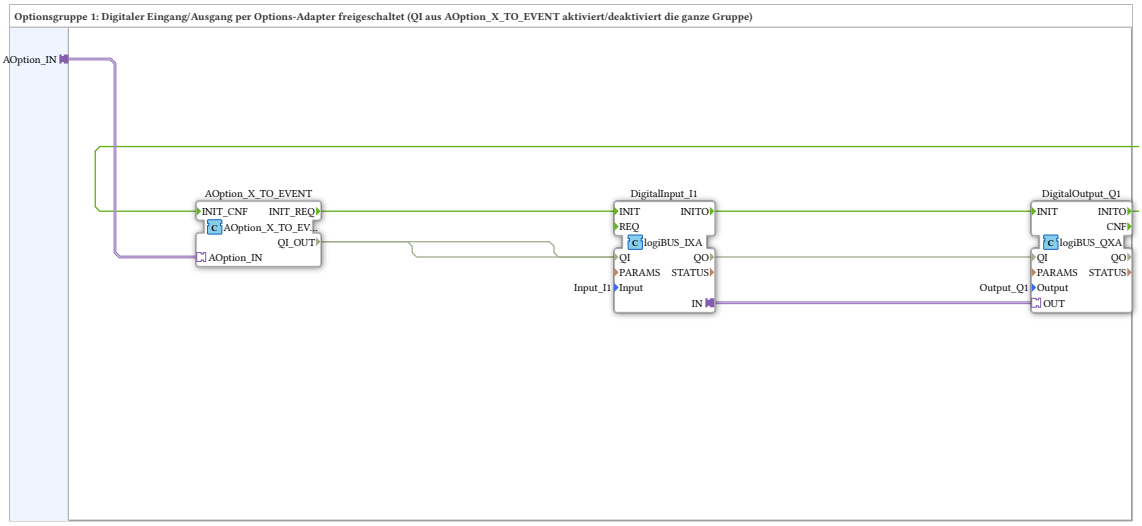
Summary

DummyGroup is an option group without effect that acknowledges requests immediately.

🛠️ Related Exercises

- Uebung_094b2_AX
- Uebung_094b3_AX
- Uebung_094b3b_AX

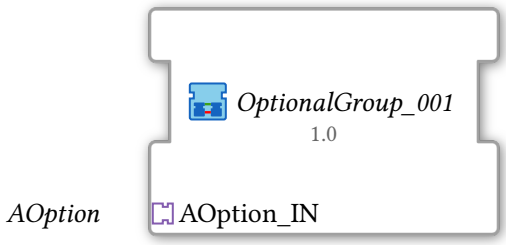
OptionalGroup_001



• – ◦ • – ◦ • – ◦ •

Introduction

OptionalGroup_001 is a SubApp (library MyLib_AX_Option): option group 1, whose digital input Input_I1 and digital output Output_Q1 (logiBUS) are enabled via the option adapter. The QI from AOption_X_TO_EVENT activates or deactivates the whole group.



Used Function Blocks (FBs)

Sub-blocks: OptionalGroup_001

- **Type**: SubAppType
- **Interface**: socket AOption_IN of type Option::AOption.
- **Internal FBs used**:
  - **AOption_X_TO_EVENT**: Option::conversion::AOption_X_TO_EVENT, converts the adapter into INIT_REQ/QI_OUT/INIT_CNF.
  - **DigitalInput_I1**: logiBUS::io::DI::logiBUS_IXA, parameter Input = Input_I1.
  - **DigitalOutput_Q1**: logiBUS::io::DQ::logiBUS_QXA, parameter Output = Output_Q1.

Program Flow and Connections

- AOption_IN -> AOption_X_TO_EVENT.AOption_IN
- Init chain: AOption_X_TO_EVENT.INIT_REQ -> DigitalInput_I1.INIT; DigitalInput_I1.INITO -> DigitalOutput_Q1.INIT; DigitalOutput_Q1.INITO -> AOption_X_TO_EVENT.INIT_CNF
- AOption_X_TO_EVENT.QI_OUT -> DigitalInput_I1.QI and DigitalOutput_Q1.QI

- Adapter connection `DigitalInput_I1.IN` -> `DigitalOutput_Q1.OUT`: the input is routed through to the output.

Technical Features

- The initialization (input, then output) is only acknowledged to the plug once both blocks are done.
- The group's `QI` is passed on to the `QI` of input and output.

Application Scenarios

- Optional digital input/output that is switched on or off by a selection block such as `AOption_AD_DEMUX_2`.

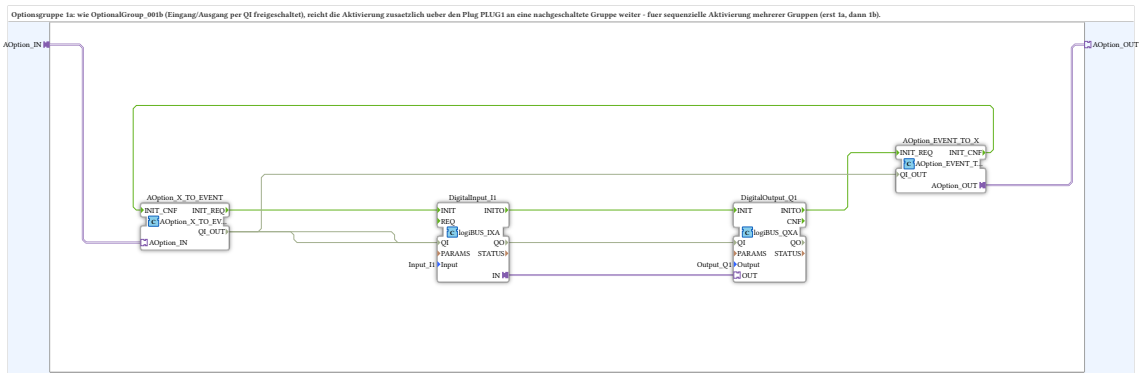
Summary

`OptionalGroup_001` bundles `Input_I1` and `Output_Q1` into an option group that can be switched on and off via `AOption`.

🛠️ Related Exercises

- Uebung_094b2_AX
- Uebung_094b3_AX
- Uebung_094b3b_AX
- Uebung_094b4_AX
- Uebung_094b4b_AX
- Uebung_094b5_AX
- Uebung_094b5b_AX
- Uebung_094b_AX
- Uebung_094c2_AX
- Uebung_094c2b_AX
- Uebung_094c_AX

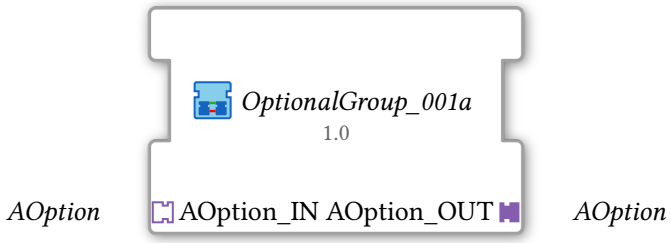
OptionalGroup_001a



• – ○ • – ○ • – ○ •

Introduction

`OptionalGroup_001a` is a SubApp (library `MyLib_AX_Option`): option group 1a. Like `OptionalGroup_001b`, its digital input `Input_I1` and digital output `Output_Q1` (logiBUS) are enabled via the option adapter. In addition, the group passes the activation on through the plug `AOption_OUT` to a downstream group - for sequential activation of several groups (first 1a, then 1b).



Used Function Blocks (FBs)

Sub-blocks: OptionalGroup_001a

- **Type**: SubAppType
- **Interface**: socket `AOption_IN` of type `Option::AOption` (adapter input) and plug `AOption_OUT` of type `Option::AOption` (adapter output).
- **Internal FBs used**:
  - **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`, converts the adapter at the socket into `INIT_REQ/QI_OUT/INIT_CNF`.
  - **DigitalInput_I1**: `logiBUS::io::DI::logiBUS_IXA`, parameter `Input = Input_I1`.
  - **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA`, parameter `Output = Output_Q1`.
  - **AOption_EVENT_TO_X**: `Option::conversion::AOption_EVENT_TO_X`, converts `INIT_REQ/QI_OUT` into the adapter at the plug `AOption_OUT` and reports `INIT_CNF` back.

Program Flow and Connections

- `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`
- Init chain: `AOption_X_TO_EVENT.INIT_REQ -> DigitalInput_I1.INIT; DigitalInput_I1.INITO -> DigitalOutput_Q1.INIT; DigitalOutput_Q1.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF`
- `AOption_X_TO_EVENT.QI_OUT -> DigitalInput_I1.QI, DigitalOutput_Q1.QI` and `AOption_EVENT_TO_X.QI_OUT`
- `AOption_EVENT_TO_X.AOption_OUT` -> `AOption_OUT`
- Adapter connection `DigitalInput_I1.IN` -> `DigitalOutput_Q1.OUT`: the input is routed through to the output.

Technical Features

- `AOption_OUT` is only triggered once `DigitalOutput_Q1` itself is initialized (`DigitalOutput_Q1.INITO -> AOption_EVENT_TO_X.INIT_REQ`).
- Its `INIT_CNF` is reported back to the own socket `AOption_IN` of the group. The calling DEMUX therefore only sees the group as done once the entire chain (including downstream groups at `AOption_OUT`) has acknowledged.
- The `QI` of the group is passed on to input, output and the plug `AOption_OUT`.

Application Scenarios

- First link of a daisy chain: `AOption_OUT` is connected to the socket of a downstream group such as `OptionalGroup_001b`, so both groups are activated one after the other (first a, then b).

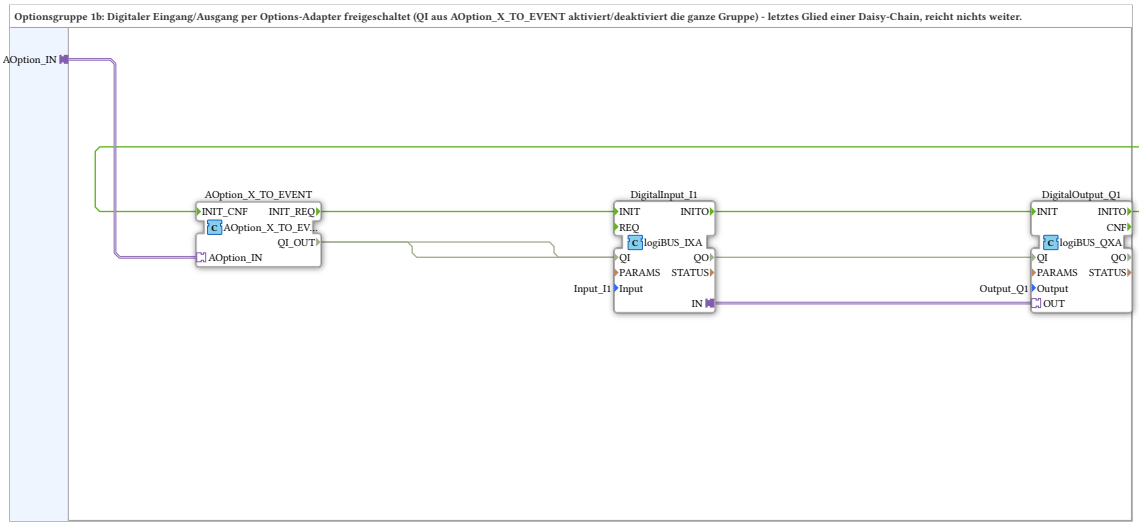
Summary

`OptionalGroup_001a` bundles `Input_I1` and `Output_Q1` into an option group switchable via `AOption` and passes the activation on to the next group via `AOption_OUT`.

🛠️ Related Exercises

- Uebung_094b3c_AX

OptionalGroup_001b



• – ○ • – ○ • – ○ •

Introduction

`OptionalGroup_001b` is a SubApp (library `MyLib_AX_Option`): option group 1b, whose digital input `Input_I1` and digital output `Output_Q1` (logiBUS) are enabled via the option adapter. The `QI` from `AOption_X_TO_EVENT` activates or deactivates the whole group. It is the last link of a daisy chain and passes nothing on.



Used Function Blocks (FBs)

Sub-blocks: OptionalGroup_001b

- **Type**: SubAppType
- **Interface**: socket `AOption_IN` of type `Option::AOption`.
- **Internal FBs used**:
  - **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`, converts the adapter into `INIT_REQ/QI_OUT/INIT_CNF`.
  - **DigitalInput_I1**: `logiBUS::io::DI::logiBUS_IXA`, parameter `Input = Input_I1`.
  - **DigitalOutput_Q1**: `logiBUS::io::DQ::logiBUS_QXA`, parameter `Output = Output_Q1`.

Program Flow and Connections

- `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`
- Init chain: `AOption_X_TO_EVENT.INIT_REQ` -> `DigitalInput_I1.INIT`; `DigitalInput_I1.INITO` -> `DigitalOutput_Q1.INIT`; `DigitalOutput_Q1.INITO` -> `AOption_X_TO_EVENT.INIT_CNF`
- `AOption_X_TO_EVENT.QI_OUT` -> `DigitalInput_I1.QI` and `DigitalOutput_Q1.QI`
- Adapter connection `DigitalInput_I1.IN` -> `DigitalOutput_Q1.OUT`: the input is routed through to the output.

Technical Features

- Identical in construction to `OptionalGroup_001a`, but without the plug `AOption_OUT`.
- The initialization (input, then output) is only acknowledged to the socket once both blocks are done.

Application Scenarios

- Typically attached to `AOption_OUT` of an upstream `OptionalGroup_001a`, so both groups are activated one after the other (first a, then b) before the original caller receives its acknowledgment.

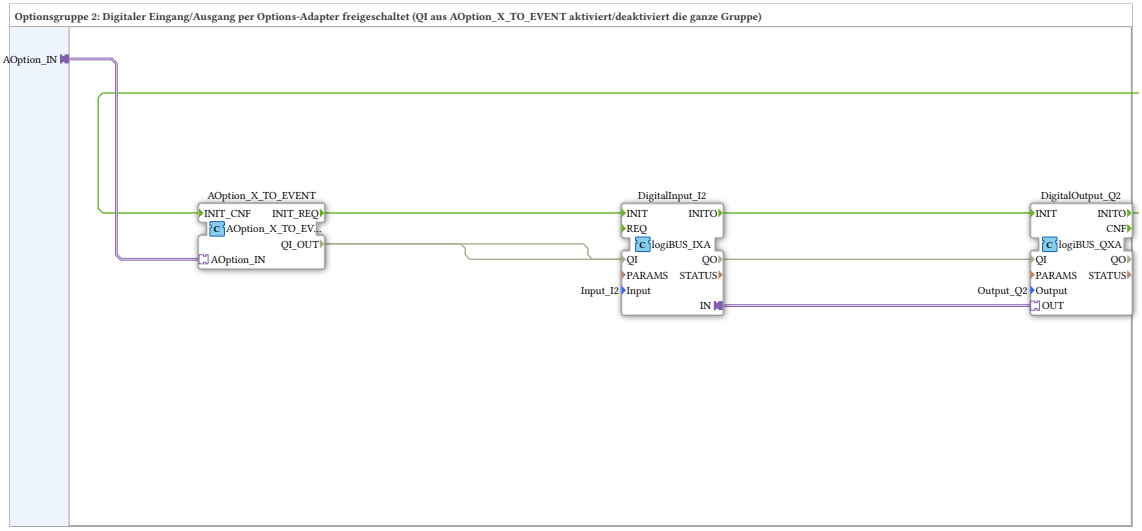
Summary

`OptionalGroup_001b` bundles `Input_I1` and `Output_Q1` into an option group switchable via `AOption` and forms the end of a daisy chain.

🛠️ Related Exercises

- Uebung_094b3c_AX

OptionalGroup_002



• – ◦ • – ◦ • – ◦ •

Introduction

`OptionalGroup_002` is a SubApp (library `MyLib_AX_Option`): option group 2, whose digital input `Input_I2` and digital output `Output_Q2` (logiBUS) are enabled via the option adapter. The `QI` from `AOption_X_TO_EVENT` activates or deactivates the whole group.



Used Function Blocks (FBs)

Sub-blocks: OptionalGroup_002

- **Type**: SubAppType
- **Interface**: socket `AOption_IN` of type `Option::AOption`.
- **Internal FBs used**:
  - **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`, converts the adapter into `INIT_REQ/QI_OUT/INIT_CNF`.
  - **DigitalInput_I2**: `logiBUS::io::DI::logiBUS_IXA`, parameter `Input = Input_I2`.
  - **DigitalOutput_Q2**: `logiBUS::io::DQ::logiBUS_QXA`, parameter `Output = Output_Q2`.

Program Flow and Connections

- `AOption_IN -> AOption_X_TO_EVENT.AOption_IN`

- Init chain: `AOption_X_TO_EVENT.INIT_REQ -> DigitalInput_I2.INIT; DigitalInput_I2.INITO -> DigitalOutput_Q2.INIT; DigitalOutput_Q2.INITO -> AOption_X_TO_EVENT.INIT_CNF`
- `AOption_X_TO_EVENT.QI_OUT -> DigitalInput_I2.QI` and `DigitalOutput_Q2.QI`
- Adapter connection `DigitalInput_I2.IN -> DigitalOutput_Q2.OUT`: the input is routed through to the output.

Technical Features

- The initialization (input, then output) is only acknowledged to the plug once both blocks are done.
- The group's `QI` is passed on to the `QI` of input and output.

Application Scenarios

- Optional digital input/output that is switched on or off by a selection block such as `AOption_AD_DEMUX_2`.

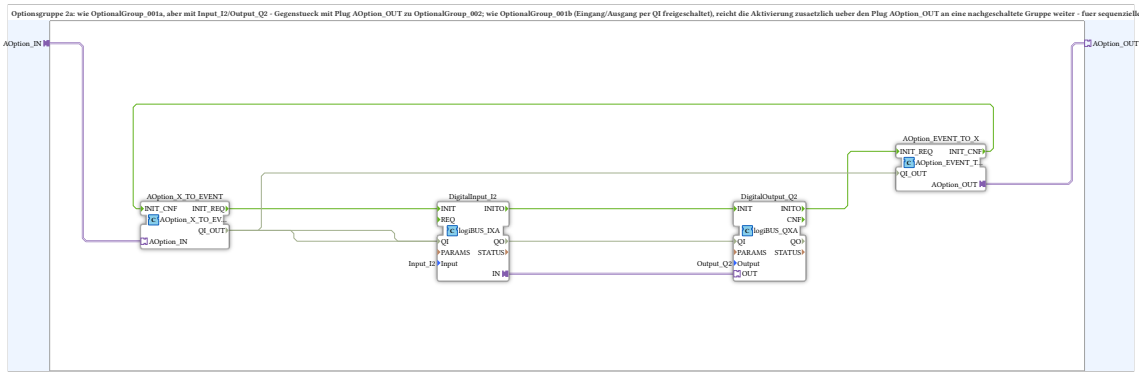
Summary

`OptionalGroup_002` bundles `Input_I2` and `Output_Q2` into an option group that can be switched on and off via `AOption`.

🛠️ Related Exercises

- Uebung_094b2_AX
- Uebung_094b3_AX
- Uebung_094b3b_AX
- Uebung_094b4_AX
- Uebung_094b4b_AX
- Uebung_094b5_AX
- Uebung_094b5b_AX
- Uebung_094b_AX
- Uebung_094c2_AX
- Uebung_094c2b_AX
- Uebung_094c_AX

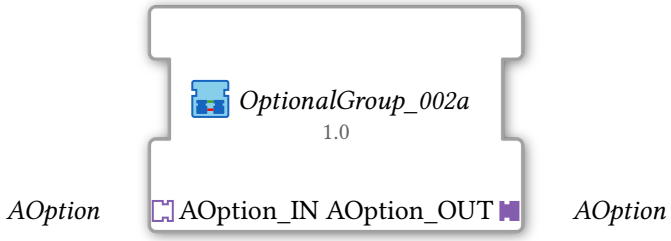
OptionalGroup_002a



• — ○ • — ○ • — ○ •

Introduction

`OptionalGroup_002a` is a SubApp (library `MyLib_AX_Option`): option group 2a. Like `OptionalGroup_001b`, its digital input `Input_I2` and digital output `Output_Q2` (logiBUS) are enabled via the option adapter. In addition, the group passes the activation on through the plug `AOption_OUT` to a downstream group - for sequential activation of several groups (first 2a, then the downstream group).



Used Function Blocks (FBs)

Sub-blocks: OptionalGroup_002a

- **Type**: SubAppType
- **Interface**: socket `AOption_IN` of type `Option::AOption` (adapter input) and plug `AOption_OUT` of type `Option::AOption` (adapter output).
- **Internal FBs used**:
  - **AOption_X_TO_EVENT**: `Option::conversion::AOption_X_TO_EVENT`, converts the adapter at the socket into `INIT_REQ/QI_OUT/INIT_CNF`.
  - **DigitalInput_I2**: `logiBUS::io::DI::logiBUS_IXA`, parameter `Input = Input_I2`.
  - **DigitalOutput_Q2**: `logiBUS::io::DQ::logiBUS_QXA`, parameter `Output = Output_Q2`.
  - **AOption_EVENT_TO_X**: `Option::conversion::AOption_EVENT_TO_X`, converts `INIT_REQ/QI_OUT` into the adapter at the plug `AOption_OUT` and reports `INIT_CNF` back.

Program Flow and Connections

- `AOption_IN` -> `AOption_X_TO_EVENT.AOption_IN`
- Init chain: `AOption_X_TO_EVENT.INIT_REQ -> DigitalInput_I2.INIT; DigitalInput_I2.INITO -> DigitalOutput_Q2.INIT; DigitalOutput_Q2.INITO -> AOption_EVENT_TO_X.INIT_REQ; AOption_EVENT_TO_X.INIT_CNF -> AOption_X_TO_EVENT.INIT_CNF`
- `AOption_X_TO_EVENT.QI_OUT -> DigitalInput_I2.QI, DigitalOutput_Q2.QI` and `AOption_EVENT_TO_X.QI_OUT`
- `AOption_EVENT_TO_X.AOption_OUT` -> `AOption_OUT`
- Adapter connection `DigitalInput_I2.IN` -> `DigitalOutput_Q2.OUT`: the input is routed through to the output.

Technical Features

- `AOption_OUT` is only triggered once `DigitalOutput_Q2` itself is initialized (`DigitalOutput_Q2.INITO -> AOption_EVENT_TO_X.INIT_REQ`).
- Its `INIT_CNF` is reported back to the own socket `AOption_IN` of the group. The calling DEMUX therefore only sees the group as done once the entire chain (including downstream groups at `AOption_OUT`) has acknowledged.
- The `QI` of the group is passed on to input, output and the plug `AOption_OUT`.

Application Scenarios

- First link of a daisy chain: `AOption_OUT` is connected to the socket of a downstream group such as `OptionalGroup_001b`, so both groups are activated one after the other (first a, then b).

Summary

`OptionalGroup_002a` bundles `Input_I2` and `Output_Q2` into an option group switchable via `AOption` and passes the activation on to the next group via `AOption_OUT`.

🛠️ Related Exercises

- Uebung_094b3f_AX

2.1.6 MyLib (test_B)

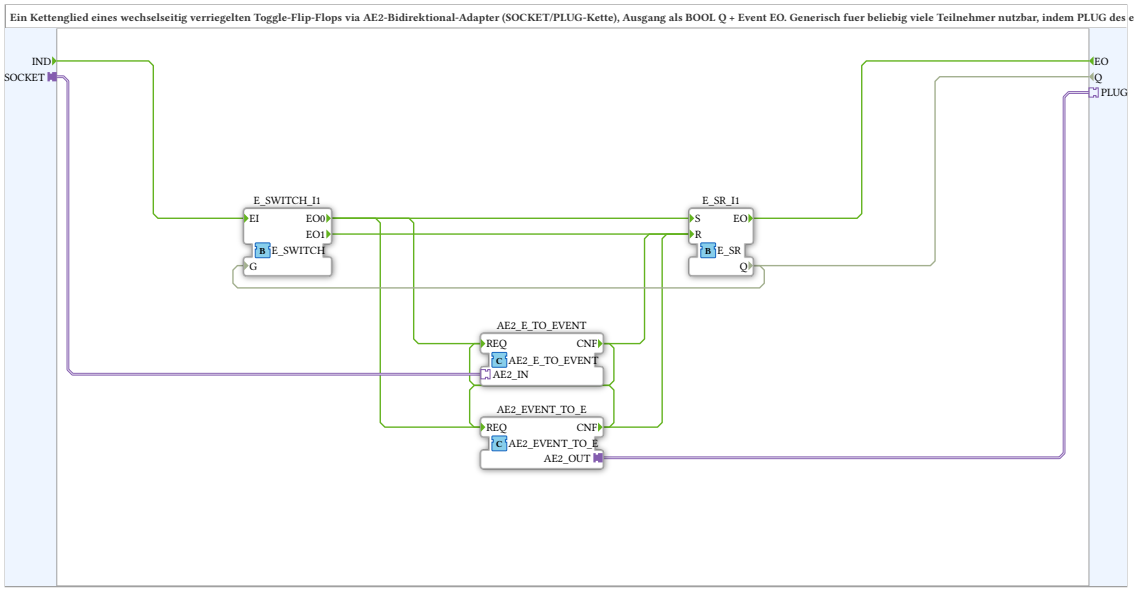
2.1.6.1 sys (test_B)

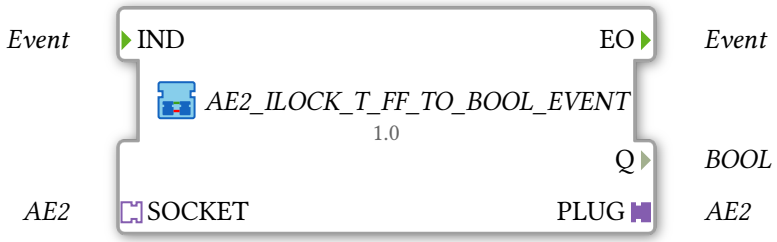
The `test_B` training system's own `MyLib::sys` block library — no name overlap with MyLib (test_AX). See in particular Background Color Blocks (shared pattern) for the large family of background-color blocks, present here without the `_AX` suffix (selector is a `BOOL` instead of an adapter).

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

AE2_ILOCK_T_FF_TO_BOOL_EVENT





• – ○ • – ○ • – ○ •

Introduction

`AE2_ILOCK_T_FF_TO_BOOL_EVENT` is a reusable IEC 61499 subapplication that implements one link of a mutually interlocked toggle flip-flop chain. It uses an AE2 bidirectional adapter interface with a `SOCKET`/`PLUG` chain topology and provides the current state as a Boolean output `Q` together with a confirmation event `EO`.

The block is designed as a chain element: several instances can be connected by linking the `PLUG` of one element to the `SOCKET` of the next element. This makes the interlock logic scalable for an arbitrary number of participants.

• – ○ • – ○ • – ○ •

Interface Structure

The subapplication has one event input, one event output, one Boolean data output, and two bidirectional adapter interfaces.

Event Inputs

Name	Type	Description
IND	Event	Trigger input used to toggle the local flip-flop state.

Event Outputs

Name	Type	Description
EO	Event	Emitted when the local E_SR flip-flop has been updated and Q has changed.

Data Inputs

None.

Data Outputs

Name	Type	Description
Q	BOOL	Current state of the local toggle flip-flop.

Adapters

Name	Type	Description
SOCKET	adapter::types::bidirectional::AE2	Bidirectional adapter connection to the previous element in the chain.
PLUG	adapter::types::bidirectional::AE2	Bidirectional adapter connection to the next element in the chain.

• – ◦ • – ◦ • – ◦ •

Functionality

The subapplication behaves as one interlock-capable toggle flip-flop inside a chain of identical elements.

When an event arrives at IND, the event is routed by the internal E_SWITCH according to the current value of Q:

- If Q = FALSE, the event is routed to the set branch of the internal E_SR. The internal state is set, so Q becomes TRUE.
- If Q = TRUE, the event is routed to the reset branch of the internal E_SR. The internal state is reset, so Q becomes FALSE.

In both cases, the internal E_SR issues its output event, which is propagated to the subapplication output EO.

The AE2 adapter part is responsible for the mutual interlock. The internal conversion FBs AE2_EVENT_TO_E and AE2_E_TO_EVENT exchange interlock events through SOCKET and PLUG. When a participant becomes active, the interlock event is sent into the chain. The event can be forwarded through neighboring elements, preventing conflicting active states in other participants.

The overall result is a decentralized, chainable interlocked toggle flip-flop that can be extended simply by adding more subapplication instances and connecting PLUG to SOCKET.

• – ◦ • – ◦ • – ◦ •

Technical Features

- Standard-compliant IEC 61499 subapplication type.
- Uses standard event FBs E_SR and E_SWITCH for state storage and event routing.
- Bidirectional AE2 adapter interface supports chain-based communication.
- No explicit data inputs are required; the only external data output is the Boolean state Q.
- The application logic is isolated from the adapter protocol by the conversion FBs AE2_EVENT_TO_E and AE2_E_TO_EVENT.

- The SOCKET/PLUG topology allows an arbitrary number of chain participants.
- A sister block AE2_ILOCK_T_FF_TO_AX exists with an AX adapter output instead of BOOL + Event.

• – ◦ • – ◦ • – ◦ •

State Overview

The internal behavior can be described by the two states of the stored flip-flop:

State	Meaning	Behavior on IND
Q = FALSE	Participant is inactive/reset.	The set branch is activated, Q becomes TRUE, and the interlock event is sent through the adapter chain.
Q = TRUE	Participant is active/set.	The reset branch is activated, Q becomes FALSE, and the confirmation event EO is emitted.

Additionally, interlock signals arriving from neighboring chain elements through the adapter interfaces can reset the local flip-flop, implementing the mutual exclusion between participants.

• – ◦ • – ◦ • – ◦ •

Application Scenarios

- Decentralized toggle systems in modular machines where only one station may be active at a time.
- Interlocked control chains built from identical reusable subapplication instances.
- Event-driven applications that need a Boolean state output combined with an event confirmation.
- Educational or experimental IEC 61499 systems demonstrating adapter-based chain communication.
- Architectures where a centralized interlock unit would require too much wiring or configuration.

• – ◦ • – ◦ • – ◦ •

Comparison with Similar Blocks

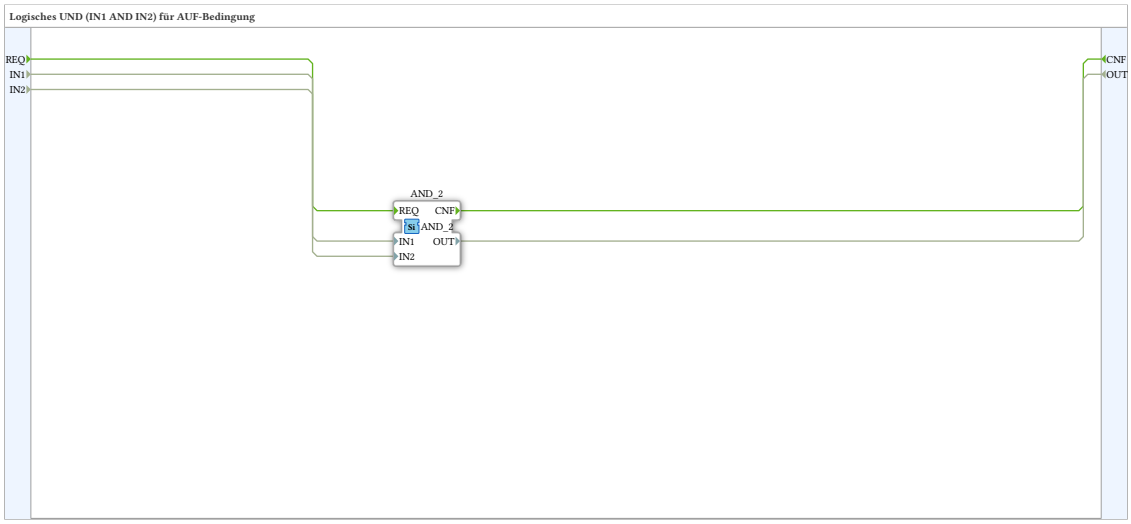
- AE2_ILOCK_T_FF_TO_BOOL_EVENT provides a Boolean output Q and a confirmation event EO, making it convenient for connection to classic IEC 61499 Boolean and event logic.
- The sister block AE2_ILOCK_T_FF_TO_AX provides the same interlocked toggle behavior but outputs an AX adapter instead of a Boolean/event interface. It is intended for systems where the next participant also expects an adapter-based interface.
- Compared with a simple E_SR or toggle flip-flop, this block adds a distributed interlock mechanism through the AE2 adapter chain. A standalone flip-flop cannot coordinate with other participants.
- Compared with a central interlock function block, this subapplication uses a neighbor-to-neighbor chain topology. This reduces central wiring and allows the interlock logic to scale more flexibly.

• – ◦ • – ◦ • – ◦ •

Conclusion

AE2_ILOCK_T_FF_TO_BOOL_EVENT is a compact and reusable IEC 61499 subapplication for building mutually interlocked toggle chains. It combines the simplicity of a Boolean output and event confirmation with the flexibility of a bidirectional AE2 adapter interface. Its SOCKET/PLUG design makes it particularly suitable for decentralized, scalable interlock applications.

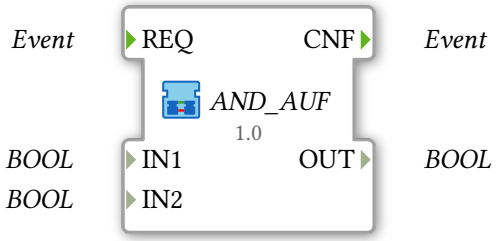
AND_AUF



• – ○ • – ○ • – ○ •

Introduction

`AND_AUF` computes the logical `IN1 AND IN2` and provides the enable condition for an "OPEN" movement (e.g. opening a valve/flap): both conditions must be true at the same time. For the complementary "CLOSE" condition, see `AND_ZU`.



Function Blocks Used

Sub-blocks: AND_AUF

- **Type**: SubAppType
- **Internal FBs used**:
  - **AND_2**: `iec61131::bitwiseOperators::AND_2` — bitwise AND, used here at BOOL level as a logical AND.
- **Operation**: `REQ` triggers `AND_2`; `IN1` and `IN2` are ANDed and returned as `OUT` and `CNF`.

Program Flow and Connections

1. `REQ` ⟶ `AND_2.REQ`.
2. `IN1` ⟶ `AND_2.IN1`; `IN2` ⟶ `AND_2.IN2`.
3. `AND_2.OUT` ⟶ `OUT`; `AND_2.CNF` ⟶ `CNF`.

Application Scenarios

- Enable logic for opening movements where two independent conditions (e.g. CLOSE limit switch not active AND enable button) must be true simultaneously.

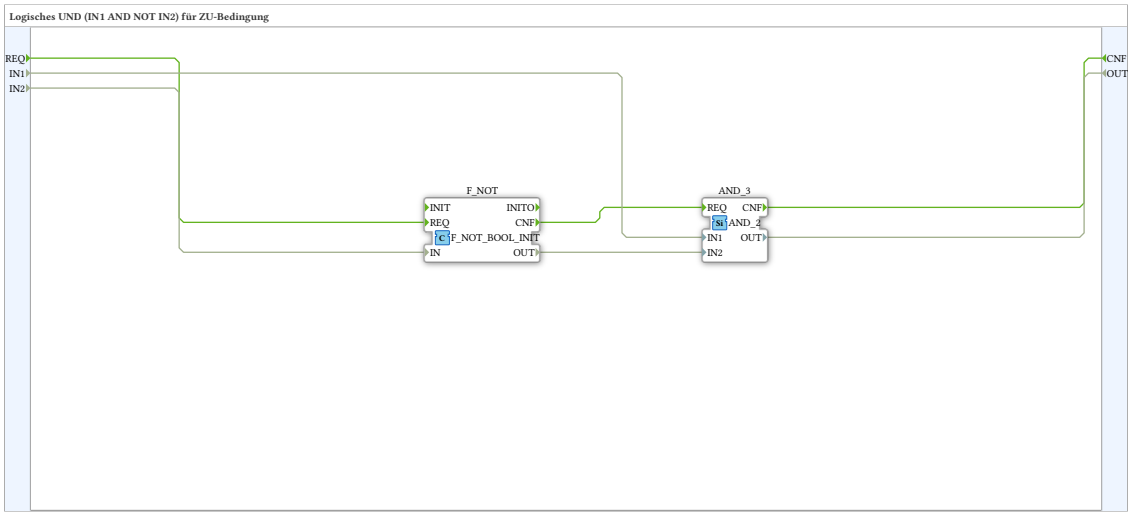
Summary

Simple, reusable AND block, meaningfully named for the OPEN condition of a valve/flap control.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

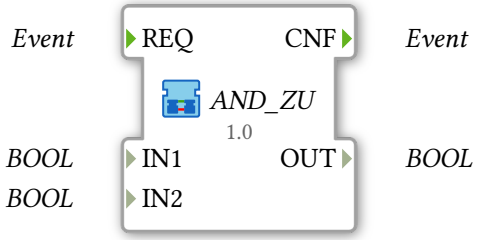
AND_ZU



• – ◦ • – ◦ • – ◦ •

Introduction

`AND_ZU` computes the logical `IN1 AND NOT IN2` and provides the enable condition for a "CLOSE" movement (e.g. closing a valve/flap): `IN1` must be true and `IN2` must be false. Complementary to `AND_AUF`.



Function Blocks Used

Sub-blocks: AND_ZU

- **Type**: SubAppType
- **Internal FBs used**:
  - **F_NOT** (`F_NOT`): `iec61131::booleanOperators::F_NOT_BOOL_INIT` — negates `IN2` before the AND is computed.
  - **AND_3** (type `AND_2`): `iec61131::bitwiseOperators::AND_2` — ANDs `IN1` with the negated `IN2`.
- **Operation**: `REQ` first triggers `F_NOT`, whose `CNF` in turn triggers `AND_3` — the negation happens before the AND, both in data flow and in event sequencing.

Program Flow and Connections

1. `REQ` ⟶ `F_NOT.REQ`; `IN2` ⟶ `F_NOT.IN`.
2. `F_NOT.CNF` ⟶ `AND_3.REQ` (event chain).
3. `IN1` ⟶ `AND_3.IN1`; `F_NOT.OUT` ⟶ `AND_3.IN2`.
4. `AND_3.OUT` ⟶ `OUT`; `AND_3.CNF` ⟶ `CNF`.

Application Scenarios

- Enable logic for closing movements where one condition must be true and a second must explicitly be *false* (e.g. enable button AND OPEN limit switch not active).

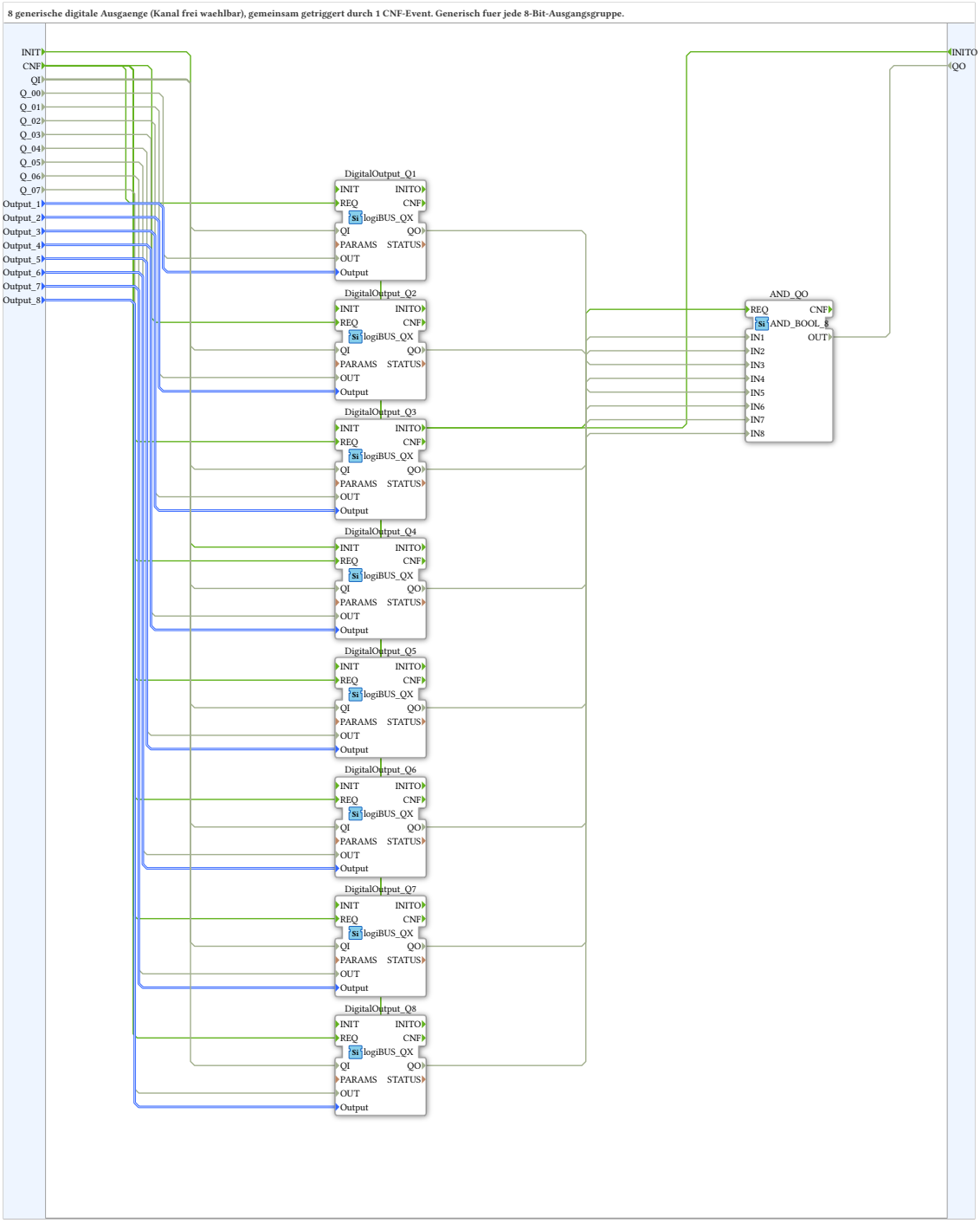
Summary

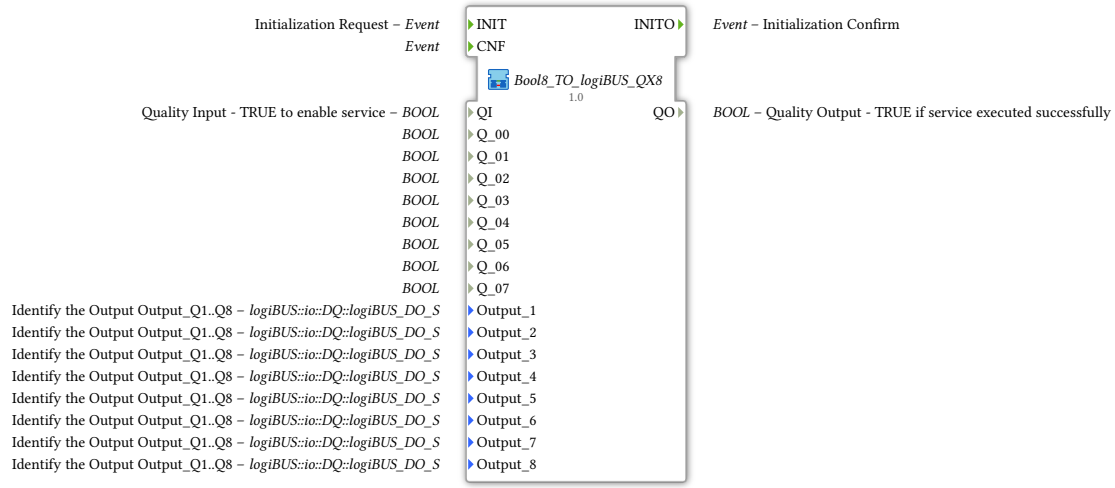
Extends `AND_AUF` with a negation of `IN2` — the matching counterpart logic for the CLOSE condition of a valve/flap control.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Bool8_TO_logiBUS_QX8





• – ○ • – ○ • – ○ •

Introduction

The Bool8_TO_logiBUS_QX8 subapplication provides a generic interface for controlling eight digital output channels on a logiBUS output module. It accepts eight Boolean values (Q_00 to Q_07) and eight channel selector parameters (Output_1 to Output_8), and forwards each Boolean value to its individually selected output channel. All eight channels are triggered simultaneously by a single confirmation (CNF) event, ensuring synchronized updates of the entire 8-bit output group. The subapplication is designed as a reusable building block for any 8-bit output group on logiBUS-based I/O systems.

Interface Structure

The interface consists of one event input, sixteen data inputs, and no event outputs, data outputs, or adapters.

Event Inputs

Event	Description
CNF	Common trigger (confirmation) event. When this event occurs, all eight internal output FBs are requested simultaneously to apply the currently present Q_00..Q_07 values to the configured output channels.
INIT	Initialization request; starts the initialization chain of the eight `logiBUS_QX` instances.

Event Outputs

Event	Description
INITO	Initialization confirm; comes from `DigitalOutput_Q3.INITO`, the end of the chain.

Data Inputs

Data Input	Type	Initial Value	Description
QI	BOOL	TRUE	Quality input - TRUE enables the service; distributed to all eight `logiBUS_QX` instances.
Q_00	BOOL	–	Logic value for channel 1 (bit position 0).
Q_01	BOOL	–	Logic value for channel 2 (bit position 1).

Q_02	BOOL	–	Logic value for channel 3 (bit position 2).
Q_03	BOOL	–	Logic value for channel 4 (bit position 3).
Q_04	BOOL	–	Logic value for channel 5 (bit position 4).
Q_05	BOOL	–	Logic value for channel 6 (bit position 5).
Q_06	BOOL	–	Logic value for channel 7 (bit position 6).
Q_07	BOOL	–	Logic value for channel 8 (bit position 7).
Output_1	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Channel identifier for the first output position.
Output_2	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Channel identifier for the second output position.
Output_3	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Channel identifier for the third output position.
Output_4	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Channel identifier for the fourth output position.
Output_5	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Channel identifier for the fifth output position.
Output_6	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Channel identifier for the sixth output position.
Output_7	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Channel identifier for the seventh output position.
Output_8	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Channel identifier for the eighth output position.

Data Outputs

Data Output	Type	Description
QO	BOOL	Quality output - TRUE if the service was executed successfully (AND of the `QO` of all eight `logiBUS_QX` instances).

Adapters

None.

Functionality

The subapplication encapsulates eight instances of the `logiBUS::io::DQ::logiBUS_QX` function block. Each instance corresponds to one bit position of an 8-bit output group:

- DigitalOutput_Q1 handles Q_00 and is configured by Output_1.
- DigitalOutput_Q2 handles Q_01 and is configured by Output_2.
- DigitalOutput_Q3 handles Q_02 and is configured by Output_3.
- DigitalOutput_Q4 handles Q_03 and is configured by Output_4.
- DigitalOutput_Q5 handles Q_04 and is configured by Output_5.
- DigitalOutput_Q6 handles Q_05 and is configured by Output_6.
- DigitalOutput_Q7 handles Q_06 and is configured by Output_7.
- DigitalOutput_Q8 handles Q_07 and is configured by Output_8.

All internal FBs receive QI from the subapplication's data input QI (default TRUE), so they are active and ready to process requests as long as QI is TRUE.

Initialization and quality: INIT runs as a chain DigitalOutput_Q4 → DigitalOutput_Q7 → DigitalOutput_Q1 → DigitalOutput_Q5 → DigitalOutput_Q2 → DigitalOutput_Q6 → DigitalOutput_Q8 → DigitalOutput_Q3 (each INITO → next INIT); DigitalOutput_Q3.INITO goes to INITO and to AND_QO.REQ. QO is the output of the instance AND_QO (iec61131::booleanOperators::AND_BOOL_8), which combines the QO of the eight instances.

When the CNF event is received, it is forwarded to the REQ input of every internal FB instance. Each instance then reads the current value of its associated Q_x input and writes it to the physical logiBUS output channel referenced by its Output_x parameter.

Because the event broadcast happens in parallel, all eight channels are updated at the same time, providing a consistent 8-bit output pattern on the bus.

Technical Features

- **Channel-individual mapping:** Each bit position (Q_00..Q_07) can be routed to any logiBUS digital output channel by configuring the corresponding Output_1..Output_8 parameter. This makes the subapplication independent of fixed hardware wiring.
- **Synchronized triggering:** A single distributed event (CNF) simultaneously activates all eight internal output blocks, guaranteeing that the complete output word is applied in one cycle.
- **Configurable channel identifiers:** The Output_x inputs are of type logiBUS_DO_S and default to logiBUS_DO::Invalid, forcing the user to explicitly select the desired output channels before operation.
- **Self-contained 8-bit group:** The subapplication encapsulates all control logic, simplifying the reuse of an 8-channel output group in different projects or contexts.
- **Minimal output interface:** Since the result is written directly to the logiBUS hardware, the only outputs are the quality output QO and the event INITO.

State Overview

The subapplication itself does not contain an execution control chart (ECC) and therefore has no internal state machine. Its behavior is strictly event-driven: the CNF event triggers the internal logiBUS_QX FBs, which may themselves implement their own internal states for handshaking with the logiBUS hardware. From the perspective of the surrounding application, the subapplication behaves deterministically: each CNF event causes a single write operation to the eight configured output channels.

Application Scenarios

- **Distributed output control:** Use the subapplication in a control system that needs to drive eight independent digital actuators (e.g., relays, valves, signal lamps) over a logiBUS fieldbus segment.
- **Standardized output modules:** When a machine or plant contains several identical 8-bit output groups, the same subapplication can be instantiated multiple times with different channel parameters, reducing engineering effort.
- **Flexible hardware mapping:** In commissioning or reconfiguration situations, the physical output channels can be reassigned by simply changing the Output_x parameters without modifying the logic structure.
- **Combined binary word output:** Use the Q_00..Q_07 inputs to transfer an 8-bit binary word (e.g., a value from a step sequencer or pattern generator) and write it to the bus in one atomic action.

Comparison with Similar Blocks

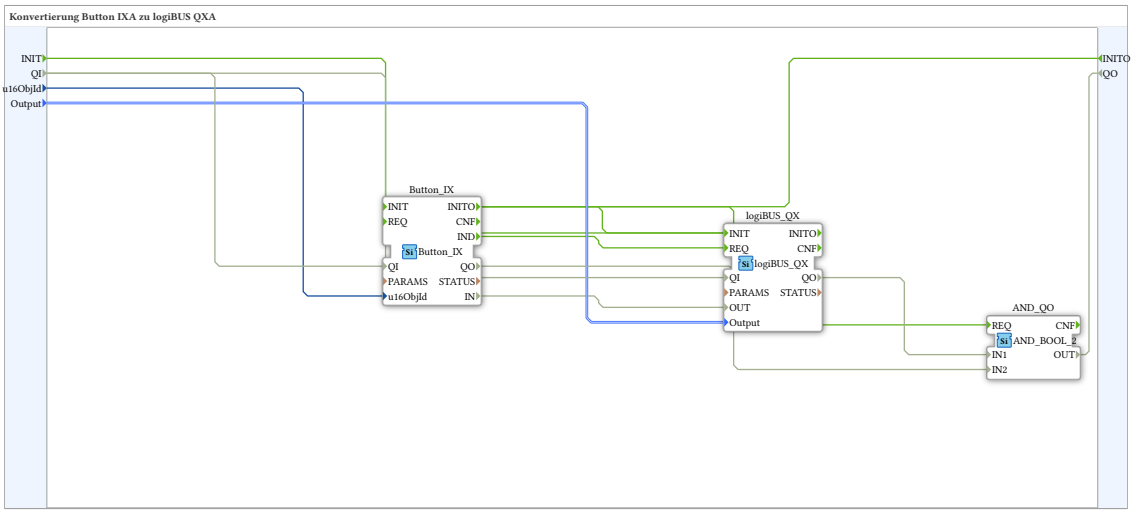
Feature	Bool8_TO_logiBUS_QX8	Individual logiBUS_QX FBs
Number of channels	8 (fixed group)	1 per FB
Channel selection	Parameterized per position (Output_1..Output_8)	Single Output parameter per FB
Synchronization	All 8 channels triggered by one CNF event	Requires individual REQ events per FB
Reusability	Encapsulated, ready-to-use 8-bit group	Requires wiring management by the user
Interface complexity	Reduced (only 9 inputs)	Higher when used in groups

The subapplication adds no functionality beyond the underlying FBs, but it significantly improves the usability, readability, and maintainability of 8-bit output groups.

Conclusion

Bool8_TO_logiBUS_QX8 is a practical, generic subapplication for controlling an 8-bit digital output group on a logiBUS system. By combining eight individually configurable output channels with a common trigger event, it offers synchronized, hardware-independent output control. Its parameterizable channel mapping and pre-integrated structure reduce engineering overhead and make it a valuable building block for modular automation projects.

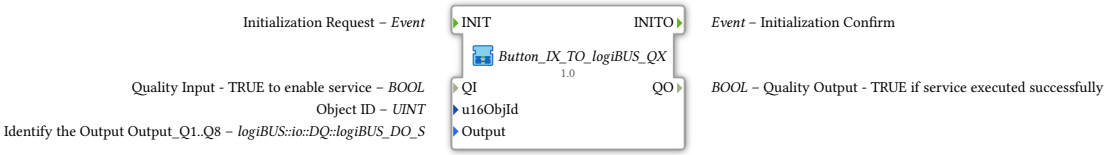
Button_IX_TO_logiBUS_QX



• — ○ • — ○ • — ○ •

Introduction

`Button_IX_TO_logiBUS_QX` is the test_B counterpart to `Button_IXA_TO_logiBUS_QXA`: a VT button (`Button_IX`) directly switches a physical digital output (`logiBUS_QX`) — here without adapters, using classic event/data connections.



Function Blocks (FBs) Used

Sub-blocks: Button_IX_TO_logiBUS_QX

- **Type**: SubAppType
- **Internal FBs used**:
  - **Button_IX**: `isobus::UT::io::Button::Button_IX` — VT button (non-adapter variant).
  - **logiBUS_QX**: `logiBUS::io::DQ::logiBUS_QX` — physical digital output (non-adapter variant).
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — combines `logiBUS_QX.QO` and `Button_IX.QO` into the `QO` of the SubApp.
- **Functionality**: `Button_IX.IND` triggers `logiBUS_QX.REQ`; the data value `Button_IX.IN` is wired directly to `logiBUS_QX.OUT`.

Program Flow and Connections

1. `u16ObjId` ⟶ `Button_IX.u16ObjId`; `Output` ⟶ `logiBUS_QX.Output`.
2. `Button_IX.IND` ⟶ `logiBUS_QX.REQ`; `Button_IX.IN` ⟶ `logiBUS_QX.OUT`.
3. **Initialization**: event input `INIT` (`EInit`) ⟶ `logiBUS_QX.INIT`; `logiBUS_QX.INITO` ⟶ `Button_IX.INIT`; `Button_IX.INITO` ⟶ event output `INITO` (`EInit`) and ⟶ `AND_QO.REQ`.

4. **Quality**: data input `QI` (`BOOL`, default `TRUE`) ⟶ `logiBUS_QX.QI` and `Button_IX.QI`; `logiBUS_QX.QO` ⟶ `AND_QO.IN1`; `Button_IX.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ data output `QO` (`BOOL`).

Application Scenarios

- Minimal VT-button-to-output block for test_B, with no status display.

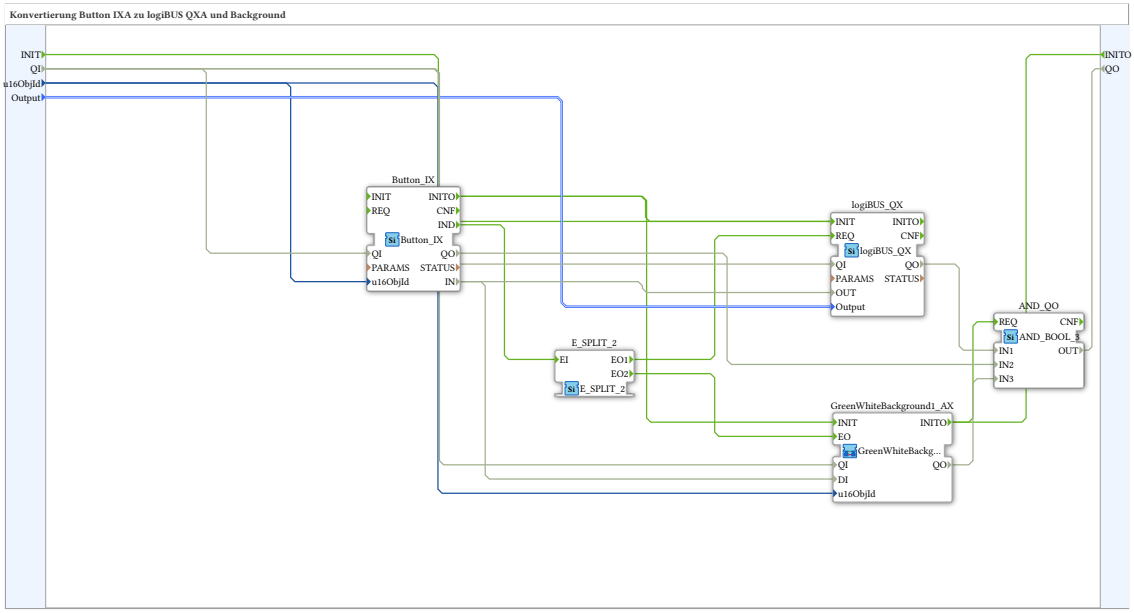
Summary

test_B counterpart to `Button_IXA_TO_logiBUS_QXA`: same function, classic event/data connections instead of adapters.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

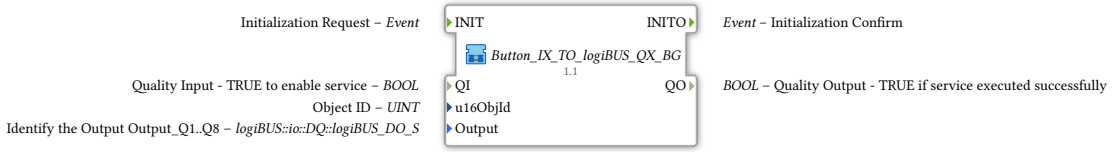
Button_IX_TO_logiBUS_QX_BG



• – ○ • – ○ • – ○ •

Introduction

`Button_IX_TO_logiBUS_QX_BG` extends `Button_IX_TO_logiBUS_QX` with a VT status color — the test_B counterpart to `Button_IXA_TO_logiBUS_QXA_BG`.



Function Blocks (FBs) Used

Sub-blocks: Button_IX_TO_logiBUS_QX_BG

- **Type**: SubAppType
- **Interface (additional)**: event input `INIT`, event output `INITO` (each `EInit`), data input `QI` (`BOOL`, default `TRUE`), data output `QO` (`BOOL`).
- **Internal FBs used**:
  - **Button_IX**: `isobus::UT::io::Button::Button_IX` — VT button.
  - **logiBUS_QX**: `logiBUS::io::DQ::logiBUS_QX` — physical digital output.
  - **E_SPLIT_2**: `iec61499::events::E_SPLIT_2` — splits the button event.

  - **GreenWhiteBackground1_AX** (SubApp, actually of type `MyLib::sys::GreenWhiteBackground1C` — a compact wrapper around the test_B base variant, see Background Color Blocks): sets the status color.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — combines the `QO` outputs of `logiBUS_QX`, `Button_IX` and `GreenWhiteBackground1_AX` into `QO`.
- **Functionality**: `Button_IX.IND` is distributed via `E_SPLIT_2` to `logiBUS_QX.REQ` and to the color block; the data value `Button_IX.IN` feeds both `logiBUS_QX.OUT` and `GreenWhiteBackground1_AX.DI`.

Program Flow and Connections

1. `u16ObjId` ⟶ `Button_IX.u16ObjId` and the color block instance's `.u16ObjId`; `Output` ⟶ `logiBUS_QX.Output`.
2. `Button_IX.IND` ⟶ `E_SPLIT_2.EI` ⟶ `E_SPLIT_2.EO1` ⟶ `logiBUS_QX.REQ`, `E_SPLIT_2.EO2` ⟶ color block `.EO`.
3. `Button_IX.IN` ⟶ `logiBUS_QX.OUT` and ⟶ color block `.DI`.
4. Initialization: `INIT` ⟶ `logiBUS_QX.INIT`; `logiBUS_QX.INITO` ⟶ `Button_IX.INIT`; `Button_IX.INITO` ⟶ `GreenWhiteBackground1_AX.INIT`; `GreenWhiteBackground1_AX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`.
5. Quality: `QI` ⟶ `.QI` of `logiBUS_QX`, `Button_IX` and `GreenWhiteBackground1_AX`; their `QO` ⟶ `AND_QO.IN1` to `AND_QO.IN3` (in this order); `AND_QO.OUT` ⟶ `QO`.

Application Scenarios

- VT button with visual status feedback for test_B, without OPC-UA.

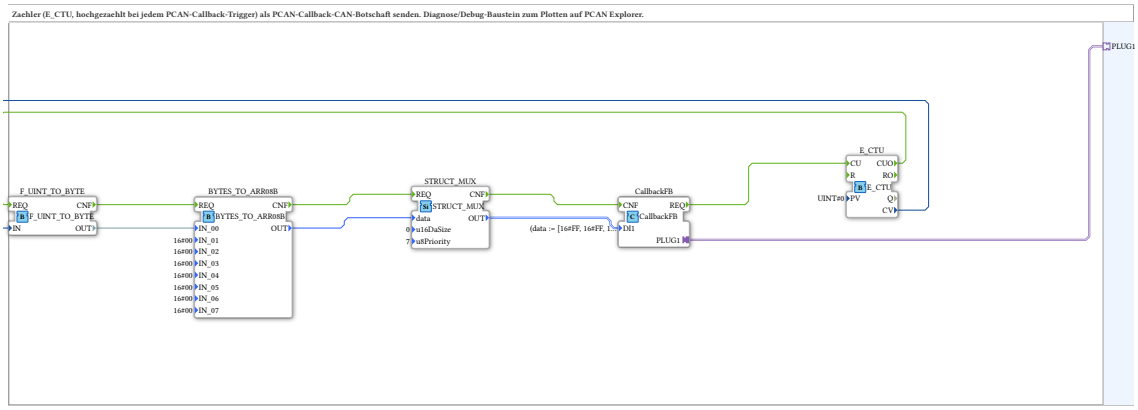
Summary

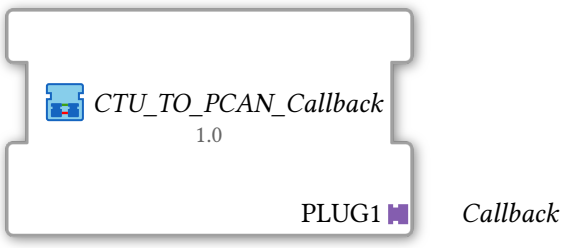
test_B counterpart to `Button_IXA_TO_logiBUS_QXA_BG`, using `E_SPLIT_2` instead of `AX_SPLIT_2` and the compact-wrapper variant of the background block.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

CTU_TO_PCAN_Callback





• – ○ • – ○ • – ○ •

Introduction

CTU_TO_PCAN_Callback is a diagnostic/debug subapplication designed to monitor PCAN callback triggers by incrementing a counter and transmitting the current counter value as a CAN message over the PCAN bus. Each time the connected callback adapter fires, the internal counter is incremented, its value is converted and packed into a CAN message structure, and the message is sent back through the callback interface. This block is intended for visualization and debugging purposes on tools like the PCAN Explorer.

Interface Structure

The subapplication exposes a single adapter plug as its primary interface. All data and event processing happen internally, triggered by the adapter's callback request.

Adapters

Adapter	Type	Description
PLUG1	`isobus::pgn::tx::Callback`	ISOBUS PGN transmission callback. Fires on incoming CAN callback requests and receives the generated CAN message for transmission.

Functionality

The subapplication implements an event-counting and CAN message generation pipeline. It operates entirely on event-driven logic triggered by the callback adapter.

Event & Data Flow

1. **Trigger**: The external callback adapter (connected to `PLUG1`) issues a request event via `CallbackFB.REQ`.
2. **Count**: The `E_CTU` event counter (with initial value 0) increments on the rising edge of the request and outputs the updated count value (`CV`).
3. **Convert**: The counter value is converted from `UINT` to a single `BYTE` by `F_UINT_TO_BYTE`.
4. **Assemble**: `BYTES_TO_ARR08B` places the converted byte into the first position (`IN_00`) of an 8-byte array. The remaining bytes (`IN_01` … `IN_07`) are initialized to `16#00`.
5. **Mux**: `STRUCT_MUX` packs the byte array into a structured `CAN_MSG` (data field, priority = 7, data size = 0) for ISOBUS framing.
6. **Transmit**: `CallbackFB` receives the structured message via its `DI1` input and forwards it as a PCAN message through the adapter (`PLUG1`).

Internal Block Diagram

```
CallbackFB.REQ → E_CTU.CU → E_CTU.CV → F_UINT_TO_BYTE.IN → BYTES_TO_ARR08B.IN_00
                                                                    |
                     STRUCT_MUX.OUT ← STRUCT_MUX.data ← BYTES_TO_ARR08B.OUT
                          |
                     CallbackFB.DI1 → transmitted via PLUG1
```

The event chain is: `CallbackFB.REQ` ⟶ `E_CTU.CU` ⟶ `E_CTU.CUO` ⟶ `F_UINT_TO_BYTE.REQ` ⟶ `F_UINT_TO_BYTE.CNF` ⟶ `BYTES_TO_ARR08B.REQ` ⟶ `BYTES_TO_ARR08B.CNF` ⟶ `STRUCT_MUX.REQ` ⟶ `STRUCT_MUX.CNF` ⟶ `CallbackFB.CNF`, providing a fully synchronous handshake from trigger to acknowledgement.

Technical Features

Feature	Details
Event Counter	`E_CTU` with preset value `0` (`PV = UINT#0`), counts rising edges of the callback request.
Data Conversion	`iec61131::conversion::F_UINT_TO_BYTE` – maps the 16-bit counter value to an 8-bit byte value.
Byte Array Assembly	`logiBUS::utils::conversion::arr::reversing::BYTES_TO_ARR08B` – constructs an 8-byte array; the counter byte occupies `IN_00`, remaining bytes are `0x00`.

CAN Message Packing	`eclipse4diac::convert::STRUCT_MUX` with structured type `isobus::pgn::CAN_MSG`; sets `u8Priority = 7` and `u16DaSize = 0`.
Callback Interface	`isobus::pgn::tx::Callback` – enables sending the generated CAN frame back to the ISOBUS transmission layer.
Flow Control	Full event chain with `REQ`/`CNF` handshakes ensures deterministic processing without race conditions.

State Overview

This subapplication does not define an explicit state machine. Instead, it relies on the built-in event flow of the participating function blocks:

- **Idle State**: No callback request pending; counter holds its last value.
- **Processing State**: Triggered by `CallbackFB.REQ`, the pipeline executes through the counter, converter, array assembler, and mux in a single synchronous pass.
- **Completion State**: `CallbackFB.CNF` acknowledges the processed trigger, and the counter value has been incremented.

The `E_CTU` block internally maintains the count value as its persistent state across cycles.

Application Scenarios

- **Debugging / Diagnostics**: Visualizing the number of ISOBUS callback events on the PCAN bus using PCAN Explorer or similar tools.
- **Protocol Monitoring**: Tracking the frequency of incoming PGN transmission callbacks at runtime.
- **System Health Checks**: Verifying that callback paths are active and working by observing monotonically increasing counter values.
- **Signal Generation for Testing**: Producing a repeating CAN frame whose data byte encodes the current trigger count, useful for instrumentation and test benches.

Comparison with Similar Blocks

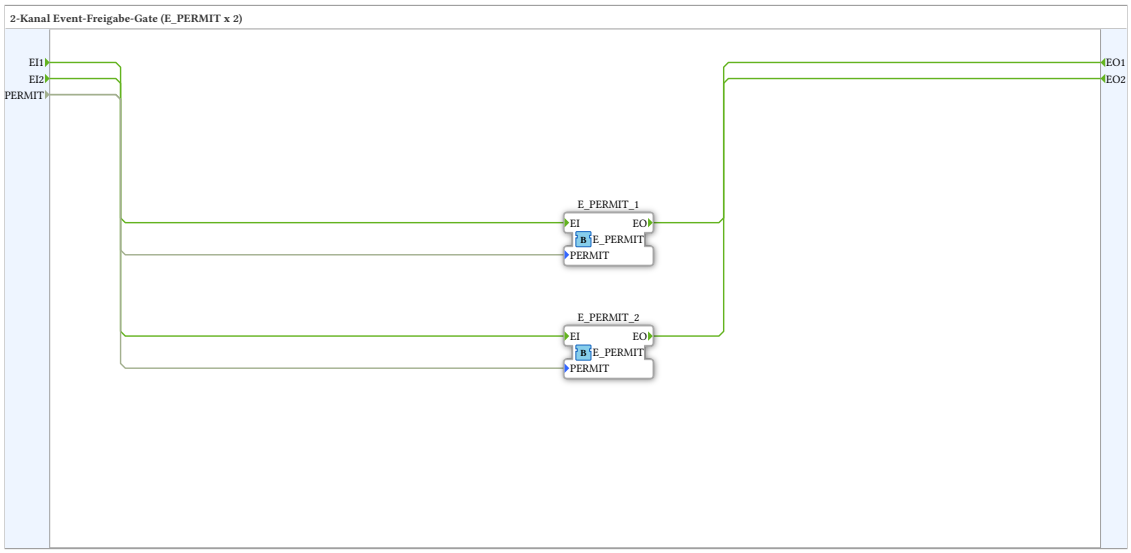
Block	Data Source	Transmission Type	Use Case
CTU_TO_PCAN_Callback	Internal event counter	ISOBUS PGN callback	Diagnostics of callback frequency on PCAN bus
CTU_TO_PCAN_Send (typical alternative)	External data input	Direct CAN send	Sending arbitrary data upon counter event
E_CTU with IO mapping	Counter output mapped directly to output	No CAN packaging	Simple CPU-local counting, no bus output

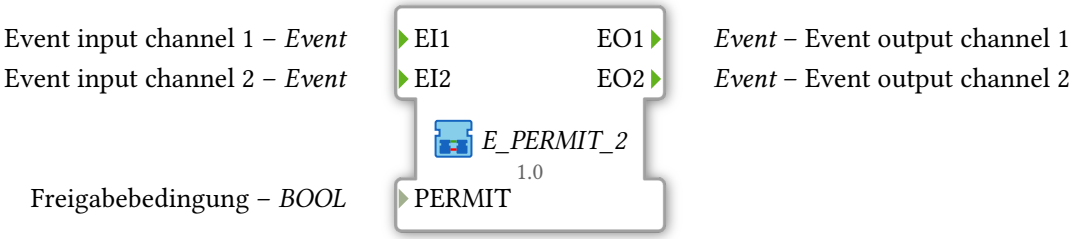
Unlike a plain counter or a direct data sender, this subapplication combines counting, byte conversion, CAN message packing, and callback-based transmission in one integrated and reusable unit. Its reverse byte order handling (`reversing::BYTES_TO_ARR08B`) accommodates endianness requirements of the ISOBUS protocol.

Conclusion

CTU_TO_PCAN_Callback is a compact, event-driven diagnostic tool that turns PCAN callback triggers into an easily observable CAN message containing an incrementing counter value. Its well-defined event flow, combined with standard IEC 61499 and ISOBUS building blocks, makes it a reusable and reliable component for debugging and monitoring ISOBUS communication on PCAN hardware. The consistent use of request/confirm handshakes guarantees deterministic behavior and simple integration into larger automation or vehicle electronics applications.

E_PERMIT_2





• – ○ • – ○ • – ○ •

Introduction

The **E_PERMIT_2** is a composite subapplication that provides a two-channel event permission gate. It encapsulates two independent instances of the standard IEC 61499 `E_PERMIT` function block, allowing two separate event streams to be selectively forwarded or blocked using a single common permission signal. This block is particularly useful in safety-oriented or access-controlled automation scenarios where multiple event paths must be gated simultaneously.

Interface Structure

Event Inputs

Name	Type	Description
EI1	Event	Event input channel 1
EI2	Event	Event input channel 2

Event Outputs

Name	Type	Description
EO1	Event	Event output channel 1
EO2	Event	Event output channel 2

Data Inputs

Name	Type	Description
PERMIT	BOOL	Common permission condition for both channels

Data Outputs

None.

Adapters

None.

Functionality

The E_PERMIT_2 subapplication implements two parallel event gate channels. Internally, it instantiates two iec61499::events::E_PERMIT function blocks:

- E_PERMIT_1 connects event input EI1 to event output EO1.
- E_PERMIT_2 connects event input EI2 to event output EO2.

Both internal blocks receive the same boolean PERMIT signal as their gating condition. The behavior of each channel follows the standard E_PERMIT semantics:

- When PERMIT = TRUE, an incoming event at EI1 (or EI2) is immediately forwarded to EO1 (or EO2).
- When PERMIT = FALSE, incoming events are suppressed and not propagated to the outputs.

The two channels operate completely independently in terms of event flow, but share a common permission signal to ensure coordinated gating behavior across both paths.

Technical Features

- **Dual redundant channel structure** – Two identical event gate paths in a single subapplication.
- **Common gating signal** – A single PERMIT data input controls both channels, simplifying external wiring.
- **Event-driven processing** – No cyclic polling; events are processed only when received on the event inputs.
- **Standard IEC 61499 compliance** – Built entirely from standard E_PERMIT function blocks.
- **Zero internal data output** – No data is produced by the subsystem; it purely routes or blocks event flows.

State Overview

Since the internal E_PERMIT blocks are event-triggered and stateless in terms of event propagation, the subapplication does not maintain a persistent internal state machine. The effective state of the gate is solely determined by the value of the PERMIT input:

State	Condition	Behavior
Permitted	PERMIT = TRUE	Events on EI1/EI2 pass to EO1/EO2
Blocked	PERMIT = FALSE	Events on EI1/EI2 are discarded

No further internal states or history are retained by the subapplication.

Application Scenarios

- **Machine safety interlocks** – Block multiple event-driven control commands (e.g., start, stop) when a safety condition is not met.
- **Production line supervision** – Gate two independent processing steps behind a single enable/disable master signal.
- **Conditional data acquisition** – Permit or suppress event notifications from two different sensors or sources based on a common readiness flag.
- **Redundant monitoring paths** – Use both channels to carry safety-relevant events while ensuring that both are disabled together during maintenance.

Comparison with Similar Blocks

Feature	Single E_PERMIT	E_PERMIT_2 (this subapp)
Number of channels	1	2
Gating inputs	1 (own PERMIT)	1 shared PERMIT for both channels
Event inputs	1 (EI)	2 (EI1, EI2)

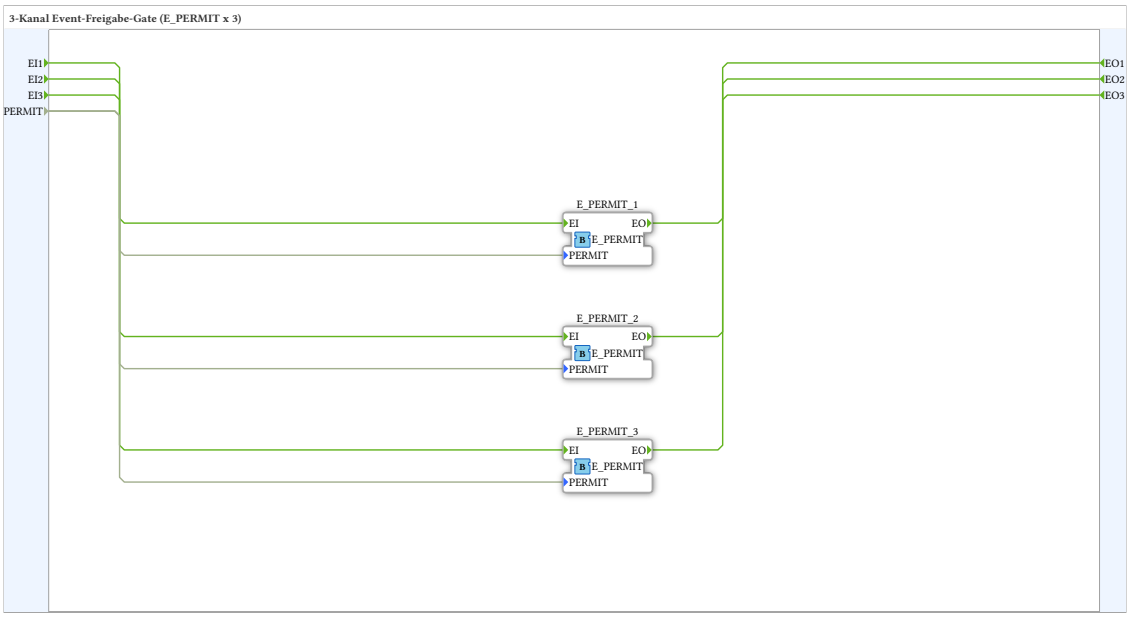
Event outputs	1 (EO)	2 (EO1, EO2)
Wiring complexity	Requires separate gating per channel	Reduced due to shared permission
Independence	Fully independent if used twice	Channels independent except for gate signal

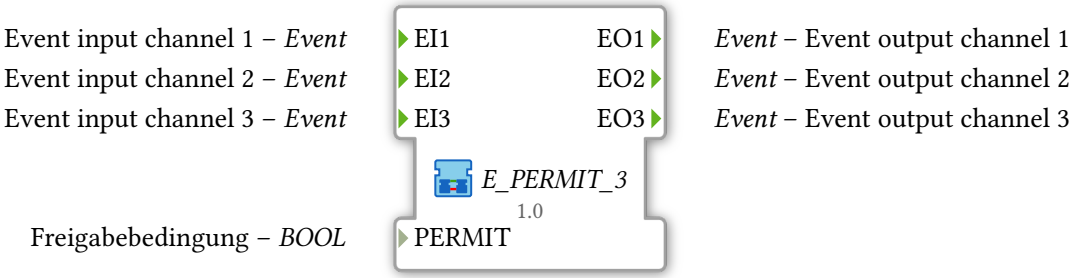
Compared to simply placing two E_PERMIT blocks manually, the E_PERMIT_2 subapplication offers a cleaner, reusable component with unified control logic, reduced wiring overhead, and a clearer interface for system integrators.

Conclusion

The **E_PERMIT_2** subapplication provides a compact and reusable solution for gating two independent event channels using a single permission input. By reusing the standardized E_PERMIT function block internally, it ensures predictable and IEC 61499-compliant behavior. Its dual-channel design makes it well suited for applications requiring coordinated event blocking across multiple paths, while keeping the external interface minimal and easy to integrate.

E_PERMIT_3





• – ○ • – ○ • – ○ •

Introduction

E_PERMIT_3 is a composite subapplication that implements a three-channel event gating mechanism. It combines three independent instances of the standard IEC 61499 E_PERMIT function block into a single reusable component. Each channel permits or blocks an incoming event stream based on a common Boolean permission signal. This subapp simplifies the design of systems requiring multiple synchronized event gates without duplicating logic.

Interface Structure

Event Inputs

- **EI1**: Event input for channel 1. When triggered, the event is forwarded to output **EO1** if the permission condition is TRUE.
- **EI2**: Event input for channel 2. When triggered, the event is forwarded to output **EO2** if the permission condition is TRUE.
- **EI3**: Event input for channel 3. When triggered, the event is forwarded to output **EO3** if the permission condition is TRUE.

Event Outputs

- **EO1**: Event output for channel 1. Corresponds to the gated event from **EI1**.
- **EO2**: Event output for channel 2. Corresponds to the gated event from **EI2**.
- **EO3**: Event output for channel 3. Corresponds to the gated event from **EI3**.

Data Inputs

- **PERMIT**: Boolean data input (type `BOOL`) that serves as the global gate condition for all three channels. If `PERMIT = TRUE`, incoming events are propagated to the respective outputs; if `FALSE`, events are discarded.

Data Outputs

No data outputs are provided by this subapplication.

Adapters

No adapters are used in this subapplication.

Functionality

The subapplication internally instantiates three identical E_PERMIT function blocks, each wired to one channel. The data input `PERMIT` is broadcast to all three blocks simultaneously. When an event arrives on any of the event inputs (`EI1`, `EI2`, or `EI3`), the corresponding internal E_PERMIT block evaluates the current value of `PERMIT`. If `PERMIT` is `TRUE`, the event is immediately re-emitted on the matching event output; otherwise, the event is suppressed. All three channels operate independently, and the behavior is identical to that of a single E_PERMIT block multiplied by three.

Technical Features

- **Three independent channels**: Each channel has its own event input and output, preventing cross-interference.
- **Shared permission signal**: A single `PERMIT` input controls all channels, simplifying coordination.
- **Standard-based implementation**: The subapplication is composed of standard IEC 61499 E_PERMIT function blocks, ensuring compatibility and predictable behavior.
- **Reusable and compact**: Encapsulates complex wiring into a single, clean interface.
- **No internal state**: The E_PERMIT block is combinational with respect to events; the subapplication does not introduce additional memory.

State Overview

The subapplication does not maintain its own state. Each internal E_PERMIT block is stateless (purely event-driven). The only relevant condition is the value of `PERMIT` at the moment an event is received. No state transitions occur within the subapplication itself.

Application Scenarios

- **Synchronized gating**: When multiple event sources must be enabled or disabled simultaneously, e.g., enabling several production machine signals based on a safety interlock.
- **System mode switching**: A central `PERMIT` can activate or deactivate three different event-driven actions (e.g., data logging, alarm processing, or control outputs) in one go.
- **Multi-channel permission checks**: In distributed control systems, three independent event paths can be selectively allowed based on a common authorization flag.
- **Prototyping and reuse**: The subapplication can be placed in a library and reused across projects where triple-channel event gating is needed.

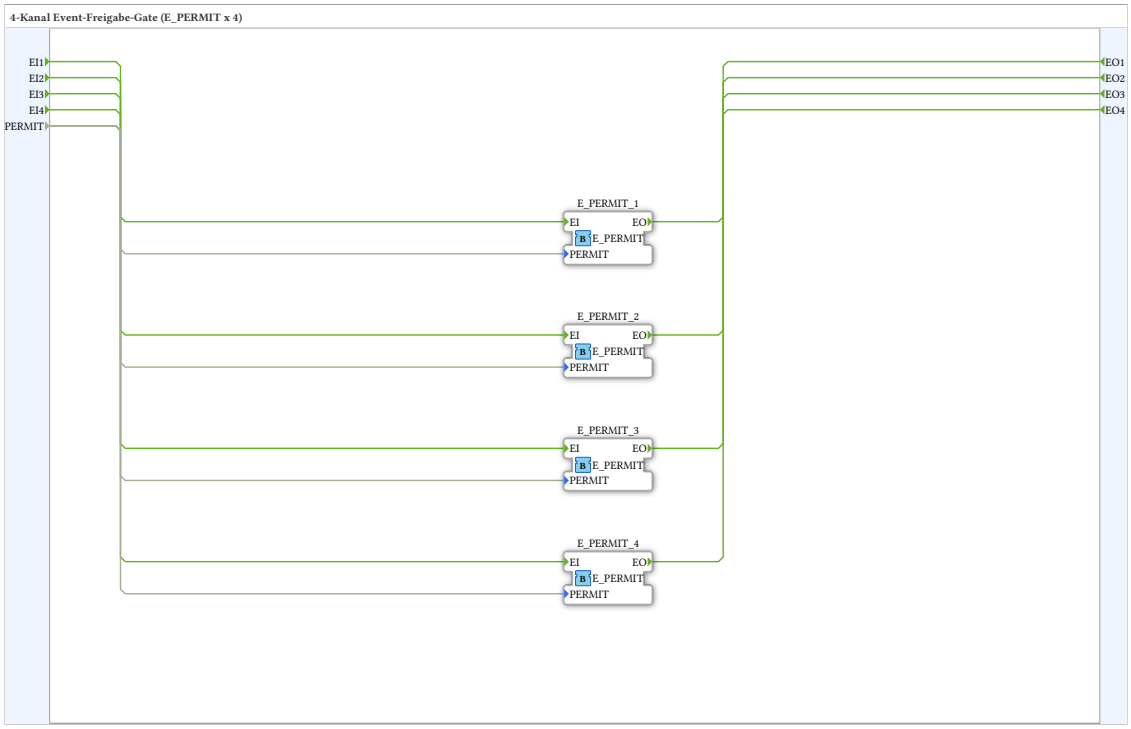
Comparison with Similar Blocks

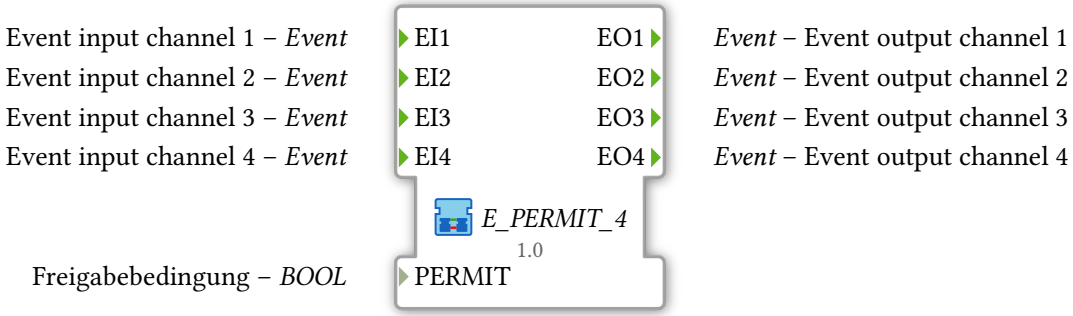
- **Single E_PERMIT**: Offers only one channel. E_PERMIT_3 is a direct triplication of the E_PERMIT functionality, providing greater convenience and reducing diagram clutter.
- **Multiple independent E_PERMIT blocks**: While functionally equivalent, wiring three separate blocks manually increases the risk of connection errors and makes the design less maintainable. E_PERMIT_3 centralizes the shared `PERMIT` signal.
- **E_PERMIT with fan-out**: Some designs might use one E_PERMIT and split its output; however, this does not provide per-channel event inputs. E_PERMIT_3 preserves distinct event inputs and outputs for each channel.

Conclusion

E_PERMIT_3 is a straightforward composite subapplication that encapsulates three instances of the standard E_PERMIT function block into a single, reusable unit. It offers a clean, three-channel event gating solution with a shared permission flag, making it ideal for scenarios where multiple event streams must be controlled together. Its simplicity, standard-based construction, and lack of internal state ensure reliable and predictable operation in a variety of automation and control applications.

E_PERMIT_4





• – ◦ • – ◦ • – ◦ •

Introduction

The **E_PERMIT_4** is a composite subapplication that bundles four independent event-permission gates (based on the standard IEC 61499 `E_PERMIT` function block) into a single reusable component. It provides

four event channels, each of which can selectively pass or block an incoming event depending on a common Boolean permit condition. This design simplifies the integration of multi-channel event gating logic in distributed control applications.

Interface Structure

Event Inputs

Name	Type	Description
EI1	Event	Event input channel 1
EI2	Event	Event input channel 2
EI3	Event	Event input channel 3
EI4	Event	Event input channel 4

Event Outputs

Name	Type	Description
EO1	Event	Event output channel 1
EO2	Event	Event output channel 2
EO3	Event	Event output channel 3
EO4	Event	Event output channel 4

Data Inputs

Name	Type	Description
PERMIT	BOOL	Permit condition shared by all four channels

Data Outputs

None.

Adapters

None.

Functionality

The subapplication contains four internal instances of the standard `E_PERMIT` function block, connected as follows:

- `EI1` ⟶ `E_PERMIT_1.EI` ⟶ `EO1`
- `EI2` ⟶ `E_PERMIT_2.EI` ⟶ `EO2`
- `EI3` ⟶ `E_PERMIT_3.EI` ⟶ `EO3`
- `EI4` ⟶ `E_PERMIT_4.EI` ⟶ `EO4`

The single Boolean input `PERMIT` is distributed to all four internal blocks.

The operating behavior for each channel is identical:

1. An event arrives at the corresponding event input (e.g., `EI1`).
2. The internal `E_PERMIT` block evaluates the shared `PERMIT` signal.
3. If `PERMIT` is **TRUE**, the event is immediately forwarded to the associated event output (e.g., `EO1`).
4. If `PERMIT` is **FALSE**, the event is discarded and no output event is generated.

The channels are fully independent in terms of event flow, meaning an event on one channel does not influence the others, apart from sharing the same permit value.

Technical Features

- **Four-channel event gating** in a single compact subapplication.

- **One shared permit input** (PERMIT) controlling all channels simultaneously, reducing wiring complexity in higher-level applications.
- **Based on standard IEC 61499** E_PERMIT blocks, ensuring predictable, platform-independent behavior.
- **No internal state retention** – the gate is purely event-driven; each event is processed on arrival.
- **Scalable concept** – the pattern can be extended to more channels if required.
- **Event propagation delay** is minimal, as each channel is a direct pass-through path when permitted.

State Overview

The E_PERMIT_4 subapplication does not maintain any persistent internal state. As it is composed of E_PERMIT blocks, each individual gate behaves in a stateless combinational manner with respect to events:

- When no event is present, the subapplication is idle.
- Upon an event input, the gate either forwards (permit = TRUE) or drops (permit = FALSE) the event.
- Changes to PERMIT take effect immediately for subsequent events.

There are no hidden latches, timers, or counters inside the component.

Application Scenarios

- **Multi-channel access control**: Enable or disable several independent event streams (e.g., sensor triggers) based on a single enabling condition.
- **Safe stop / emergency gating**: If a safety signal (PERMIT) becomes FALSE, all outgoing event triggers are suppressed.
- **Sequencing gates**: Conditionally forward start or stop commands to multiple machine axes simultaneously.
- **Test & simulation**: Provide a global switch to block all event-driven communication while simulating or debugging.
- **Batch processing**: Allow a set of parallel process steps only when a common prerequisite is fulfilled.

Comparison with Similar Blocks

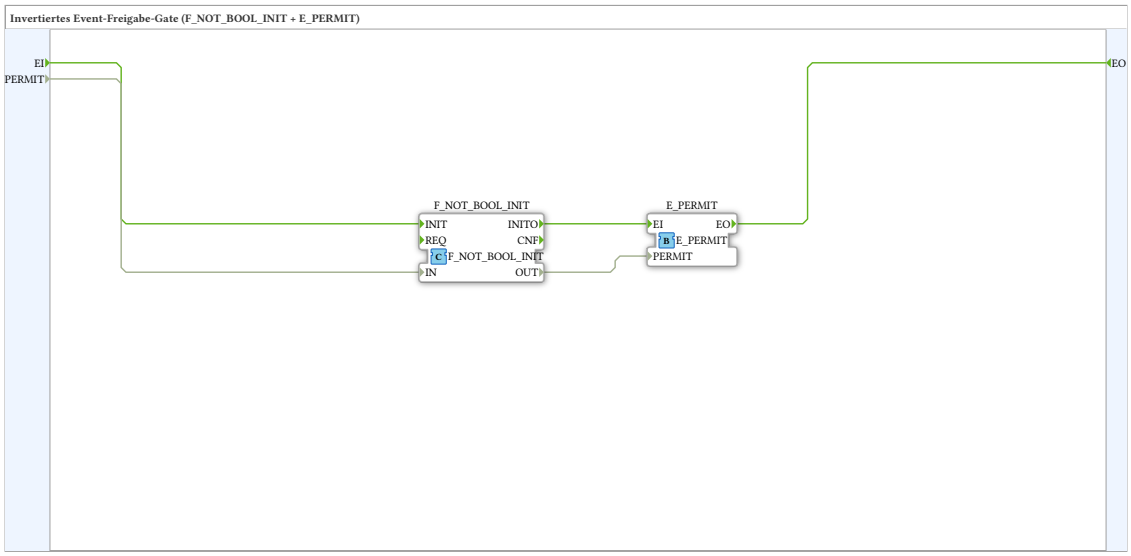
Feature	E_PERMIT_4	Single E_PERMIT	E_SWITCH (per MB)
Number of channels	4	1	1 (with 2 outputs)
Permit input	Single shared PERMIT	Dedicated PERMIT per instance	Not applicable
Event routing	Pass or block	Pass or block	Route to one of two outputs
Use case	Multi-channel gating	Single-channel gating	Branching / multiplexing events
Integration effort	Low (one component)	High for many channels	Medium for branching logic

Compared to placing four separate E_PERMIT blocks manually, E_PERMIT_4 offers a cleaner interface, consistent wiring, and a single control point for the permit condition. It does not provide the branching capability of an E_SWITCH, but it is purpose-built for independent channel blocking.

Conclusion

The E_PERMIT_4 subapplication is a practical and reusable building block for applications that require simultaneous enable/disable control over multiple event streams. By wrapping four standard E_PERMIT blocks behind a unified interface, it reduces engineering effort, improves readability of the application logic, and ensures consistent behavior across all channels. It is particularly well suited for access control, safety gating, and multi-axis coordination tasks within IEC 61499-based systems.

E_PERMIT_INVERT





• – ○ • – ○ • – ○ •

Introduction

E_PERMIT_INVERT is an event-gating subapplication that implements an inverted event permission gate. It combines an inversion function and an event-permit block so that an incoming event is only forwarded when the Boolean input `PERMIT` is `FALSE`. This is useful in control logic where a `TRUE` signal should suppress or inhibit an event, rather than enable it.

Interface Structure

Event Inputs

Name	Type	Description
`EI`	Event	Event input. An event arriving here is forwarded to `EO` when `PERMIT` is `FALSE`.

Event Outputs

Name	Type	Description
`EO`	Event	Event output. Fires when `EI` occurs and `PERMIT` is `FALSE`.

Data Inputs

Name	Type	Description
`PERMIT`	BOOL	Inverted enable condition. If `TRUE`, the event is blocked. If `FALSE`, the event is allowed to pass.

Data Outputs

None.

Adapters

None.

Functionality

The internal structure of E_PERMIT_INVERT consists of two standard blocks:

- F_NOT_BOOL_INIT
- E_PERMIT

The PERMIT input is connected to F_NOT_BOOL_INIT.IN. The inverted result is then connected to E_PERMIT.PERMIT. The incoming event EI is connected directly to E_PERMIT.EI, and the output event E_PERMIT.EO is forwarded as EO.

This means:

1. EI triggers the event path.
2. F_NOT_BOOL_INIT inverts the current value of PERMIT.
3. E_PERMIT receives the inverted permission value.
4. If the original PERMIT is FALSE, the inverted value is TRUE, so the event is forwarded to EO.
5. If the original PERMIT is TRUE, the inverted value is FALSE, so the event is blocked.

Technical Features

- Implements an inverted event permission gate.
- Reuses standard IEC 61499 event and IEC 61131 logic blocks.
- Encapsulates the wiring of an inverter and an event gate in one reusable subapplication.
- Provides a single Boolean control input and a single event input/output pair.
- Does not expose any data outputs.
- Behaves as a pure event gate: no output event is generated without an input event.
- Suitable for event-driven control applications.

State Overview

E_PERMIT_INVERT has no internal state machine and no persistent state variables. The output behavior depends only on the current value of PERMIT and the occurrence of an event at EI. The gate is combinational with respect to the Boolean condition: a change of PERMIT alone does not generate an output event, but it affects whether the next EI event is passed through.

Application Scenarios

- Active-low enable signals: an event is allowed when the enable signal is FALSE.
- Inhibit logic: a TRUE inhibit signal suppresses event-driven actions.
- Interlocking: events are blocked while a safety condition is active.
- Replacing a manual inverter followed by an E_PERMIT with a single reusable component.
- Cleaner design in event-based control systems where inverted permission semantics are required.

Comparison with Similar Blocks

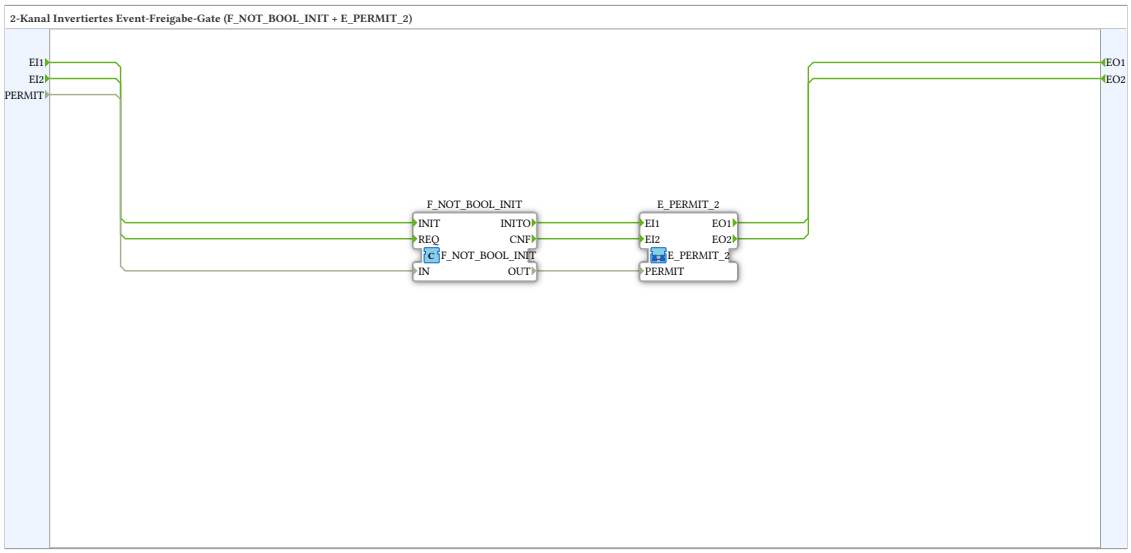
Block / Approach	Behavior
E_PERMIT	Passes an event when the permit input is TRUE.
E_PERMIT_INVERT	Passes an event when the permit input is FALSE.
Manual inverter + E_PERMIT	Equivalent logic, but requires additional wiring and is less reusable.
E_SWITCH	Routes events to different output paths, rather than simply suppressing or passing one event.

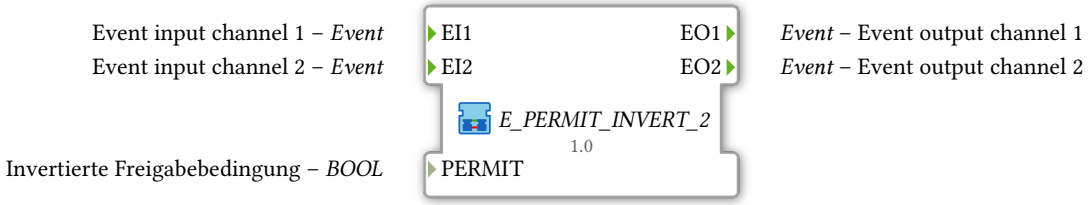
E_PERMIT_INVERT is therefore best understood as an “event gate with inverted permission logic”.

Conclusion

E_PERMIT_INVERT is a compact and reusable subapplication that provides inverted event gating logic. By internally combining F_NOT_BOOL_INIT and E_PERMIT, it gives designers a clear, maintainable way to permit event flow when a Boolean condition is FALSE. It is especially useful in event-driven control applications where inhibit or active-low enable signals are required.

E_PERMIT_INVERT_2





• – ○ • – ○ • – ○ •

Introduction

E_PERMIT_INVERT_2 is a two-channel inverted event permit gate. It is implemented as a subapplication that combines an IEC 61131 boolean inverter with a standard IEC 61499 E_PERMIT_2 event gate.

The subapplication accepts two independent event input channels and forwards them to two output channels only when an external BOOL permission signal is FALSE. This makes the block suitable for applications where an active-low permission or release signal must control event forwarding.

Interface Structure

Event Inputs

Name	Type	Description
EI1	Event	Event input channel 1. Forwarded to EO1 when the gate is open.
EI2	Event	Event input channel 2. Forwarded to EO2 when the gate is open.

Event Outputs

Name	Type	Description
EO1	Event	Event output channel 1. Emitted when EI1 is received and the gate is open.
EO2	Event	Event output channel 2. Emitted when EI2 is received and the gate is open.

Data Inputs

Name	Type	Description
PERMIT	BOOL	Inverted permission condition. FALSE enables event forwarding; TRUE blocks event forwarding.

Data Outputs

None.

Adapters

None.

Functionality

The subapplication uses an internal F_NOT_BOOL_INIT block to invert the external PERMIT signal. The resulting value is connected to the PERMIT input of an internal E_PERMIT_2 block.

The behavior is as follows:

- When the external PERMIT input is FALSE, the internal E_PERMIT_2 receives a TRUE permit signal. An event on EI1 is forwarded to EO1, and an event on EI2 is forwarded to EO2.
- When the external PERMIT input is TRUE, the internal E_PERMIT_2 receives a FALSE permit signal. Incoming events on EI1 and EI2 are blocked and no output events are generated.

Each event channel is independent. EI1 only controls EO1, and EI2 only controls EO2. Both channels share the same inverted permission condition.

Technical Features

- Reusable subapplication defined as an IEC 61499 SubAppType.
- Two event input channels and two event output channels.
- One common BOOL permission input.
- No data outputs and no adapters.
- Internally uses the standard function blocks F_NOT_BOOL_INIT and E_PERMIT_2.
- Provides active-low event gating behavior.
- Encapsulates the inverter plus event gate logic into a single application component.
- The gate does not queue or store events. Events arriving while the gate is closed are discarded.

State Overview

The subapplication does not define an internal state machine or sequential function chart. Its behavior is determined by the current value of the external PERMIT input.

External PERMIT	Internal E_PERMIT_2 Input	Result
FALSE	TRUE	Gate is open. EI1 triggers EO1, EI2 triggers EO2.
TRUE	FALSE	Gate is closed. EI1 and EI2 are blocked.

The behavior is combinational with respect to the permission signal. Changing PERMIT immediately changes whether new events are allowed through. No internal state is retained.

Application Scenarios

E_PERMIT_INVERT_2 is useful in control applications where the available permission signal has inverted logic. Typical scenarios include:

- Using a “fault active” or “blocked” signal as the event gate. Events are only passed when the fault signal is FALSE.
- Gating two independent event streams with a single active-low release condition.
- Integrating inverted enable logic into a reusable subapplication to avoid adding a separate inverter block in the calling application.
- Implementing safety-oriented event suppression where a TRUE signal must prevent event propagation.

Comparison with Similar Blocks

Block / Pattern	Behavior
E_PERMIT_2	Passes events when the PERMIT input is TRUE. Requires an external inverter for active-low permission signals.
E_PERMIT_INVERT_2	Passes events when the external PERMIT input is FALSE. The inversion is integrated internally.

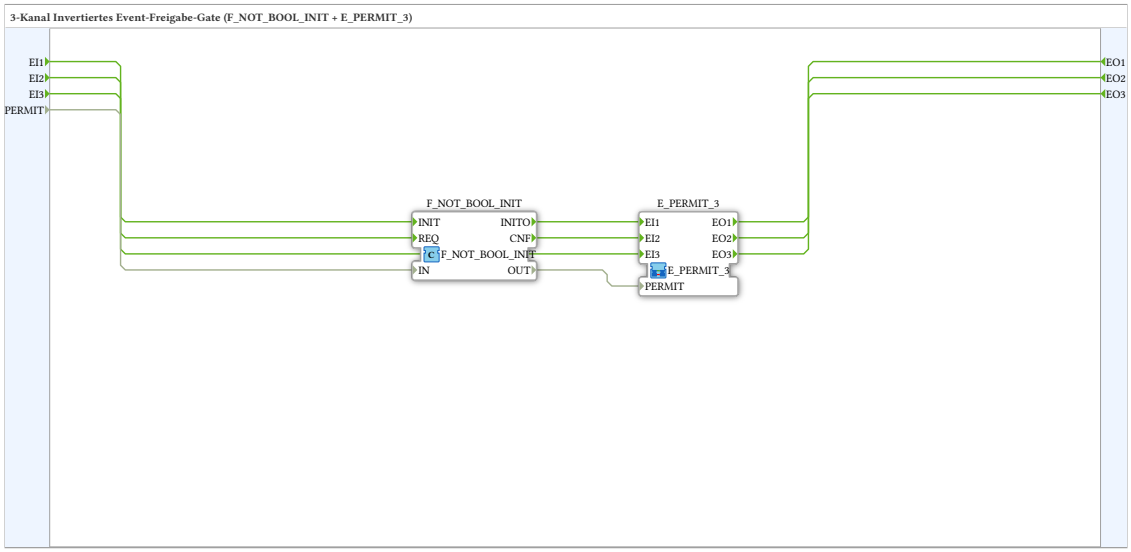
F_NOT_BOOL_INIT	Inverts a BOOL data signal only. It does not gate events.
E_PERMIT	A single-channel permit gate with similar high-active semantics.

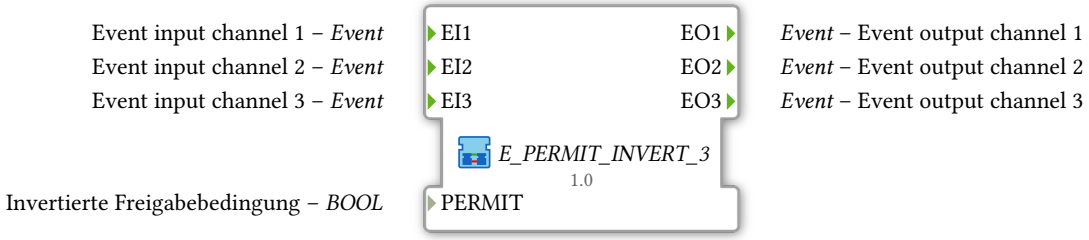
E_PERMIT_INVERT_2 is therefore a convenient alternative to manually wiring an inverter in front of an E_PERMIT_2 block.

Conclusion

E_PERMIT_INVERT_2 is a compact and reusable two-channel event gate for inverted permission logic. By combining a boolean inverter with an E_PERMIT_2 event gate, it provides clear active-low event forwarding behavior. It is especially useful when an application uses inhibit or fault signals to control whether events may pass. The subapplication reduces wiring complexity and improves readability in larger IEC 61499 systems.

E_PERMIT_INVERT_3





• – ○ • – ○ • – ○ •

Introduction

E_PERMIT_INVERT_3 is a composite subapplication that combines the IEC 61131 boolean inverter F_NOT_BOOL_INIT with the IEC 61499 event gate E_PERMIT_3. It provides three independent event channels that are only enabled when the external PERMIT signal is FALSE. The subapp therefore behaves as a 3-channel inverted event-enable gate: incoming events are forwarded to the outputs as long as the enable condition is de-asserted.

Interface Structure

Event Inputs

Name	Type	Comment
EI1	Event	Event input channel 1
EI2	Event	Event input channel 2

EI3	Event	Event input channel 3

Event Outputs

Name	Type	Comment
EO1	Event	Event output channel 1
EO2	Event	Event output channel 2
EO3	Event	Event output channel 3

Data Inputs

Name	Type	Comment
PERMIT	BOOL	Inverted enable condition (Freigabebedingung)

Data Outputs

None.

Adapters

None.

Functionality

The subapp implements a three-channel event gate with an inverted enable signal. The external PERMIT input is connected to the IN input of F_NOT_BOOL_INIT, whose output is the logical negation of PERMIT. This negated value is then fed to the PERMIT input of E_PERMIT_3.

Inside E_PERMIT_3, an event arriving on EI1, EI2, or EI3 is forwarded to the corresponding output EO1, EO2, or EO3 only while the PERMIT input of the gate is TRUE. Because the subapp inverts the external PERMIT signal before passing it to the gate, the effective forwarding condition for the external user is:

- PERMIT = FALSE ⟶ internal permit = TRUE ⟶ events are forwarded.
- PERMIT = TRUE ⟶ internal permit = FALSE ⟶ events are blocked.

No data values are consumed or produced by the subapp; it purely controls event propagation.

Technical Features

- Reusable composite design built from standard IEC 61131 and IEC 61499 function blocks.
- Three independent event channels controlled by a single shared enable signal.
- Complementary behavior compared to the plain E_PERMIT_3 block.
- F_NOT_BOOL_INIT provides a deterministic boolean negation with a defined initial output state, which avoids undefined behavior during cold start.
- Easy to integrate into larger 4diac-ide applications due to its simple interface.
- No data outputs or adapters reduce wiring complexity.

State Overview

E_PERMIT_INVERT_3 contains no explicit state machine. Its runtime behavior can be described in terms of the combined states of the internal blocks:

External PERMIT	F_NOT_BOOL_INIT.OUT	E_PERMIT_3 internal permit	Result
FALSE	TRUE	TRUE	Events pass through
TRUE	FALSE	FALSE	Events blocked

There is no event-driven state retention; each incoming event is either forwarded or discarded depending solely on the current value of PERMIT.

Application Scenarios

- **Safety and emergency-stop logic:** Event processing is permitted only while a safety signal is inactive, e.g. when a stop button is not pressed.

- **Interlock circuits:** Machine event handling is disabled while a guard door is open or a warning condition is active.
- **Supervisory enable/disable:** A central control signal blocks or releases event-driven activities for three parallel subsystems at once.
- **Fail-safe event gating:** In designs where events must be suppressed whenever an alarm flag is set, the inverted permit semantics fit naturally.

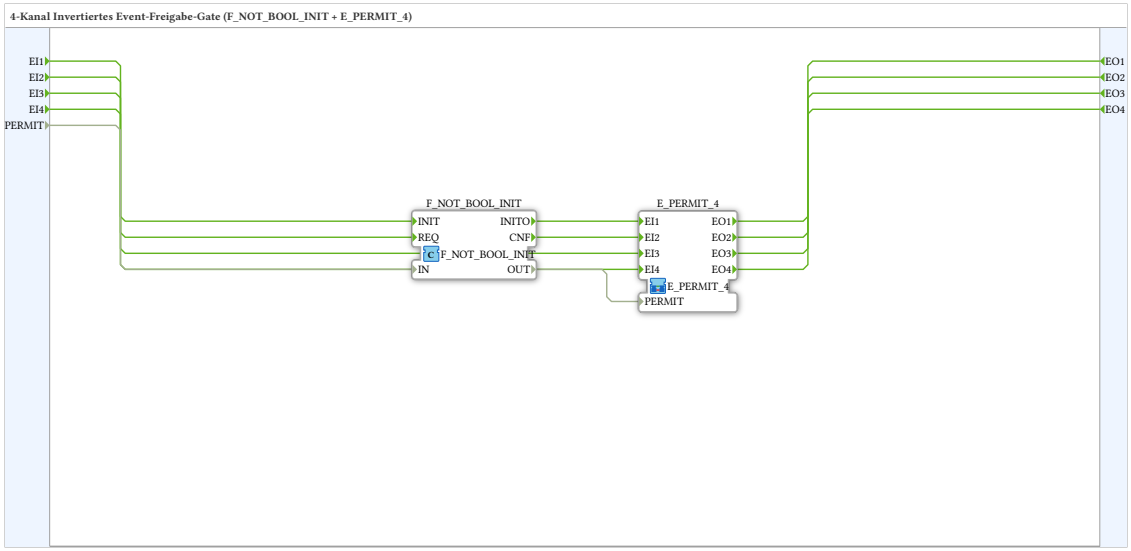
Comparison with Similar Blocks

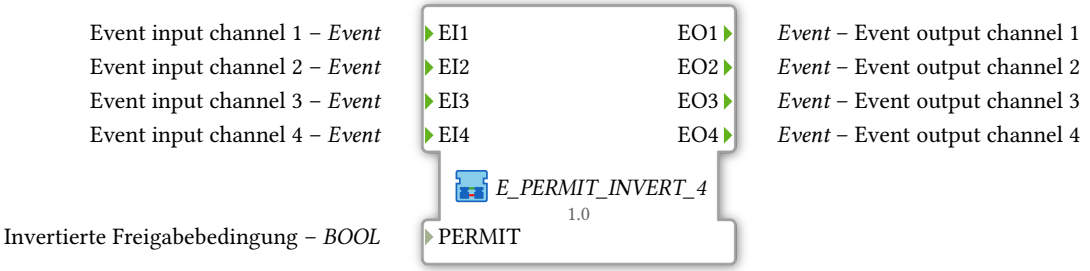
- **E_PERMIT_3:** The standard E_PERMIT_3 forwards events when its PERMIT input is TRUE. E_PERMIT_INVERT_3 behaves exactly opposite from the perspective of the external signal, making it suitable for active-low enable logic.
- **E_SWITCH / E_DEMUX:** These blocks route events to selected outputs based on a control input, whereas E_PERMIT_INVERT_3 only blocks or forwards events on the same channel.
- **F_NOT_BOOL_INIT:** A pure logic element without event gating semantics. E_PERMIT_INVERT_3 adds the event-gate behavior around it.

Conclusion

E_PERMIT_INVERT_3 is a compact and practical subapplication for inverted event gating. By internally combining a boolean inverter with a three-channel event permit gate, it offers a clearly named and reusable component for applications that require event flow to be enabled by an active-low signal. Its simple interface, deterministic initialization, and complementary behavior to E_PERMIT_3 make it a valuable building block in IEC 61499-based control systems.

E_PERMIT_INVERT_4





• – ○ • – ○ • – ○ •

Introduction

`E_PERMIT_INVERT_4` is a typed subapplication that implements a four-channel inverted event-permission gate. It combines an `F_NOT_BOOL_INIT` block with an `E_PERMIT_4` block. The external `PERMIT` input is

logically inverted before it is applied to the internal event gate. Therefore, an event is forwarded only when PERMIT is FALSE. When PERMIT is TRUE, arriving events are suppressed.

The four event channels operate independently from each other, but they all share the same permission condition.

Interface Structure

The subapplication has four event inputs, four event outputs, and one Boolean data input. There are no data outputs and no adapters.

Event Inputs

Name	Type	Description
EI1	Event	Event input channel 1.
EI2	Event	Event input channel 2.
EI3	Event	Event input channel 3.
EI4	Event	Event input channel 4.

Event Outputs

Name	Type	Description
EO1	Event	Event output channel 1.
EO2	Event	Event output channel 2.
EO3	Event	Event output channel 3.
EO4	Event	Event output channel 4.

Data Inputs

Name	Type	Description
PERMIT	BOOL	Inverted enable condition. FALSE opens the event gate; TRUE closes it.

Data Outputs

None.

Adapters

None.

Functionality

E_PERMIT_INVERT_4 performs the following internal operations:

1. The Boolean input PERMIT is negated by F_NOT_BOOL_INIT.
2. The negated value is connected to the PERMIT input of E_PERMIT_4.
3. Each event input EIx is connected to the corresponding event input of E_PERMIT_4.
4. The corresponding event output EOx is emitted only when the internal E_PERMIT_4.PERMIT value is TRUE.

The effective enable condition is:

enable = **NOT PERMIT**

For each channel x:

- When PERMIT = FALSE, enable = TRUE, and an event on EIx produces an event on EOx.
- When PERMIT = TRUE, enable = FALSE, and an event on EIx is suppressed.

The gate does not buffer events. If the gate is closed, the incoming event is discarded.

Technical Features

- Four-channel event gating in a single reusable subapplication.

- Active-low permission behavior due to the inverted PERMIT input.
- Shared gate condition for all four channels.
- Direct one-to-one event routing from each EIx to the corresponding EOx.
- No data outputs or additional data transformation.
- Built from standard blocks: E_PERMIT_4 and F_NOT_BOOL_INIT.
- No adapters required for integration.

State Overview

E_PERMIT_INVERT_4 has no complex internal state machine. The relevant state is determined by the value of PERMIT:

State	PERMIT	Internal E_PERMIT_4.PERMIT	Event behavior
Gate open	FALSE	TRUE	EIx ⟶ EOx
Gate closed	TRUE	FALSE	EIx is suppressed

A change of PERMIT takes effect for the next arriving event. The four event channels remain independent and do not synchronize with one another.

Application Scenarios

This subapplication is suitable whenever an event stream should be allowed while a signal is inactive. Typical examples include:

- Passing alarm or notification events only when the process is not in an error state.
- Blocking event outputs during maintenance or startup phases, where the control signal is TRUE.
- Implementing fail-safe enable logic in which FALSE means “normal operation” and TRUE means “inhibit”.
- Combining multiple permission-dependent event channels into one compact component.

Comparison with Similar Blocks

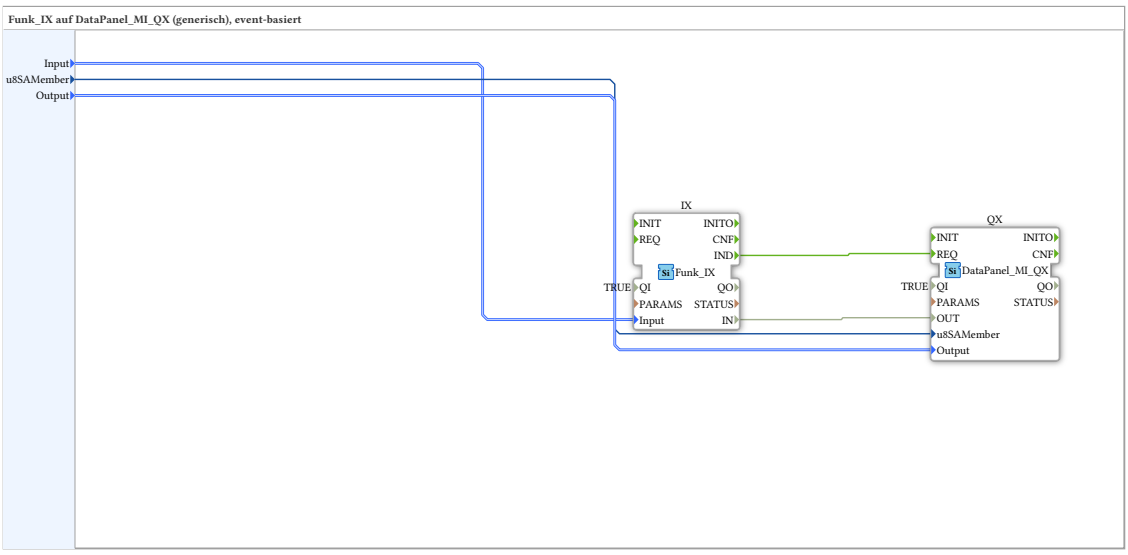
E_PERMIT_4 is a non-inverted event gate: events are forwarded when its PERMIT input is TRUE. E_PERMIT_INVERT_4 reverses this behavior, so events are forwarded when the external PERMIT is FALSE.

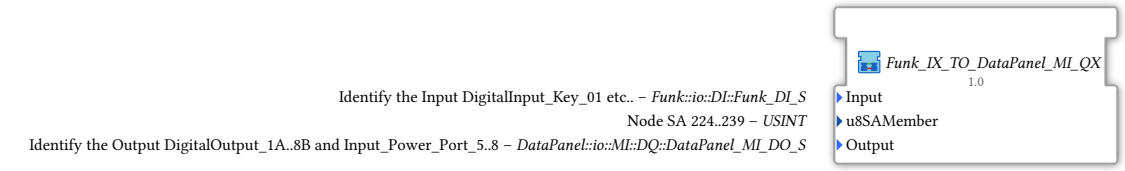
Compared to manually wiring an F_NOT_BOOL_INIT in front of an E_PERMIT_4, this subapplication reduces connection effort and avoids wiring mistakes. It is not a routing switch like E_SWITCH or E_SELECT; it simply forwards or suppresses an event on the same channel.

Conclusion

E_PERMIT_INVERT_4 is a compact and clear solution for four-channel event gating with an active-low enable condition. By combining a Boolean inverter with the standard E_PERMIT_4 block, it provides predictable event suppression and easy integration into IEC 61499 applications.

Funk_IX_TO_DataPanel_MI_QX





• – ◦ • – ◦ • – ◦ •

Introduction

The `Funk_IX_TO_DataPanel_MI_QX` is a generic, event-based subapplication that bridges a wireless input module (Funk IX) to a DataPanel output module (MI QX). It is designed to be reused in various contexts where a single digital input event (e.g., a key press) must be forwarded to a corresponding digital output on a DataPanel. The subapp encapsulates the wiring and synchronisation between these two function blocks, providing a clean and configurable interface.

Interface Structure

The subapplication exposes three data inputs, no event inputs or outputs, and no adapters. All communication with the outside world is performed via these data ports.

Event Inputs

None.

Event Outputs

None.

Data Inputs

Name	Type	Initial Value	Description
`Input`	`Funk::io::DI::Funk_DI_S`	`Funk_DI::Invalid`	Identifies the incoming digital input (e.g., `DigitalInput_Key_01`).
`u8SAMember`	`USINT`	`MI::MI_00`	Node SA address (range 224..239) used for the output module.
`Output`	`DataPanel::io::MI::DQ::DataPanel_MI_DO_S`	`Invalid`	Identifies the output channel on the DataPanel (e.g., `DigitalOutput_1A..8B` or `Input_Power_Port_5..8`).

Data Outputs

None.

Adapters

None.

Functionality

The subapplication connects an instance of the function block `Funk::io::DI::Funk_IX` to an instance of `DataPanel::io::MI::DQ::DataPanel_MI_QX`. The internal wiring is as follows:

- **Event connection**: The event output `IND` of the Funk block (IX) is connected to the event input `REQ` of the DataPanel block (QX). This ensures that every time a new value is received by the Funk input, the DataPanel output is triggered.
- **Data connection**: The data output `IN` of the Funk block is connected to the data input `OUT` of the DataPanel block. This transfers the actual value (e.g., a button state) from the wireless module to the output stage.

The three external data inputs of the subapplication are passed directly to the corresponding parameters of the internal blocks:

- `Input` is forwarded to `IX.Input`
- `u8SAMember` is forwarded to `QX.u8SAMember`
- `Output` is forwarded to `QX.Output`

Both internal blocks have their `QI` parameter set to `TRUE`, so they are permanently enabled. The `PARAMS` parameter of the Funk block is set to an empty string and is hidden in the configuration.

Technical Features

- **Generic configuration**: The subapplication can be reused for any combination of Funk digital input and DataPanel output by simply setting the three data inputs.
- **Event-driven operation**: The forwarding of an input event to the output is synchronous and triggered by the Funk block's `IND` event.
- **No internal state**: The subapplication itself does not contain any state variables; it only establishes the data and event flow between the two function blocks.
- **Compile-time optimised**: All connections are defined in the subapplication network, allowing the compiler to inline or optimise the data paths.

State Overview

The subapplication does not maintain any state of its own. The behaviour is fully determined by the states of the two internal function blocks:

- **Funk IX**: Waits for a new input event. On arrival, it updates its `IN` output and issues an `IND` event.
- **DataPanel QX**: Upon receiving the `REQ` event, it reads the `OUT` data input and applies the value to the configured output channel.

There is no explicit state machine in the subapplication; the overall system state is the combination of the states of the internal blocks.

Application Scenarios

This subapplication is particularly useful in scenarios where a wireless remote control or sensor needs to control a physical output on a DataPanel. Typical use cases include:

- Wireless keypad to activate a light or motor output.
- Remote button to trigger a digital output on a DataPanel in an industrial or building automation environment.
- Generic bridging of any `Funk_DI` input to a `DataPanel_MI_DO` output, with the flexibility to choose the exact input and output channel at runtime.

Because the configuration is done via data inputs, the same subapplication instance can be re-parameterised without changing the internal structure, making it ideal for reusable library components.

Comparison with Similar Blocks

The subapplication is tailored to the specific combination of Funk input and DataPanel output. Similar blocks might include:

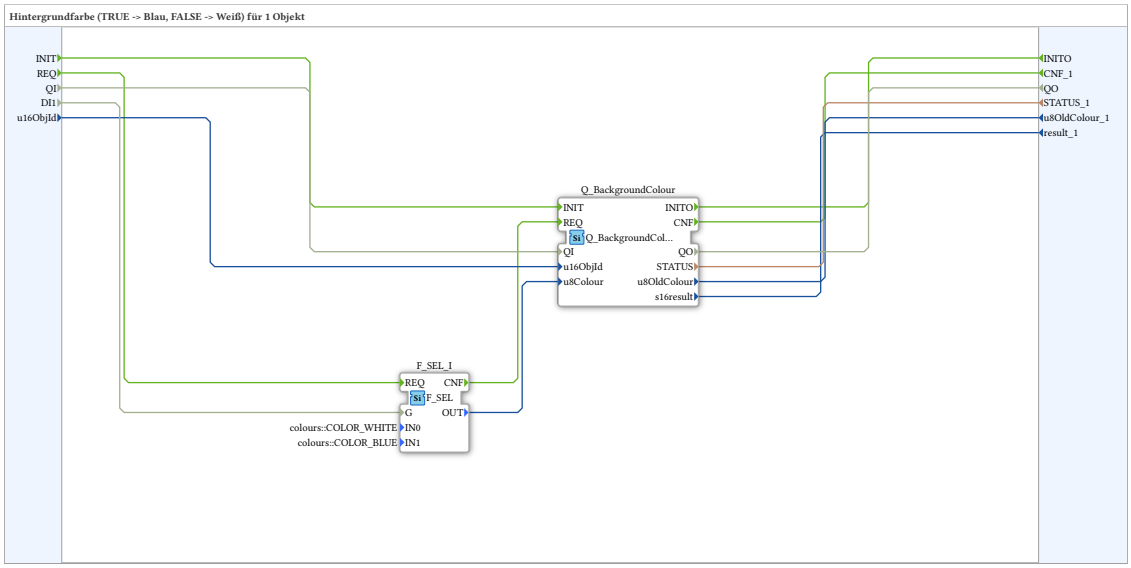
- **Direct Function Block Connection**: Instead of a wrapper subapplication, a designer could directly connect a `Funk_IX` to a `DataPanel_MI_QX` in the parent application. This would be more flexible but would expose internal details and require manual wiring for every new instance.
- **Other bridge blocks**: Other subapplications may exist that combine different input and output types (e.g., Modbus to DataPanel). However, this subapplication is specialised for the Funk wireless input family.

Compared to a direct connection, the subapplication offers the advantages of encapsulation, reusability, and a simplified interface. It also ensures that the correct event and data paths are always used, reducing the chance of wiring errors.

Conclusion

`Funk_IX_TO_DataPanel_MI_QX` is a compact and reusable subapplication that provides a clean bridge between a Funk digital input and a DataPanel digital output. Its event-based nature and configurable parameters make it a practical building block for automation projects that require wireless-to-output connectivity. The subapplication hides the complexity of the internal wiring and can be easily integrated into larger systems, promoting maintainability and consistency.

GreenBlueBackground1

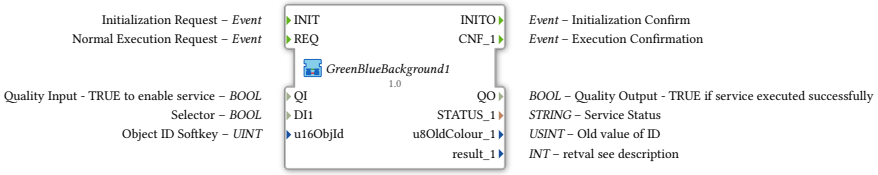


• — ○ • — ○ • — ○ •

Introduction

`GreenBlueBackground1` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColour.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColour.QO`.

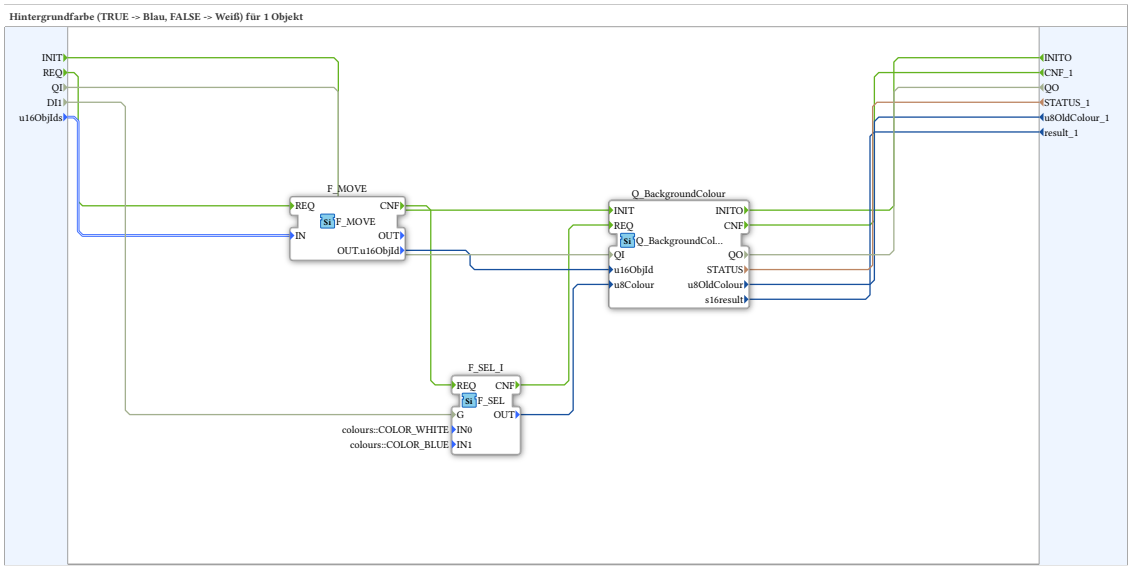
Summary

One of many variants in the background color block family: color pair Blau/Weiß, 1 object, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenBlueBackground1S

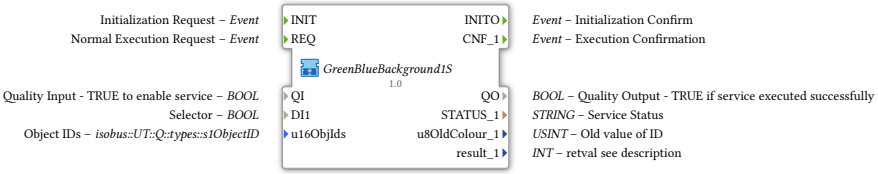


• — ○ • — ○ • — ○ •

Introduction

`GreenBlueBackground1S` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColour.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColour.QO`.

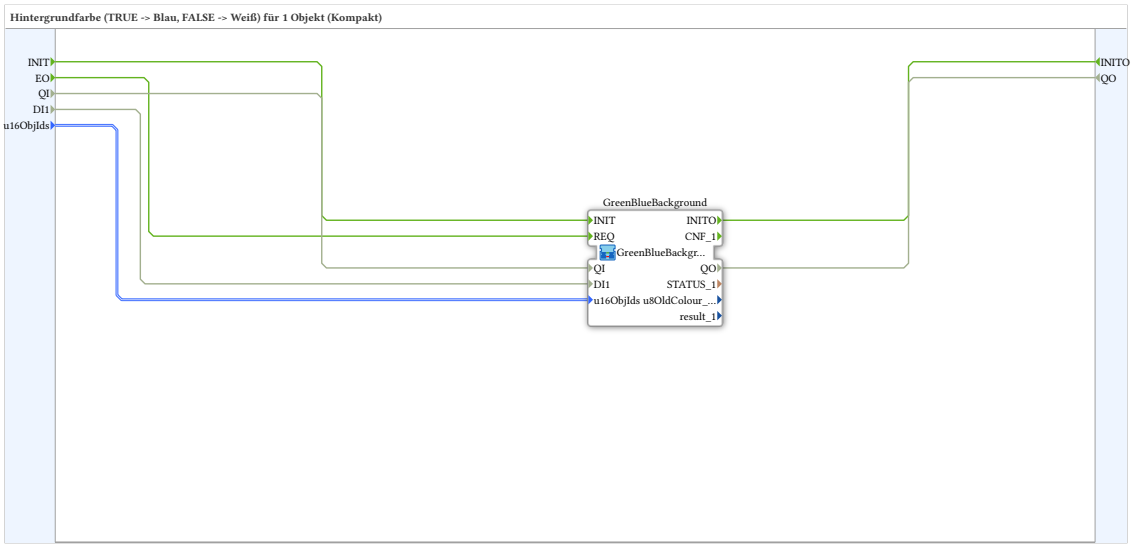
Summary

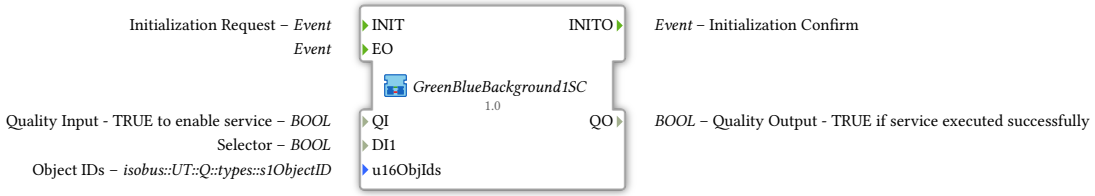
One of many variants in the background color block family: color pair Blau/Weiß, 1 object, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenBlueBackground1SC





• – ○ • – ○ • – ○ •

Introduction

`GreenBlueBackground1SC` is a **compact wrapper** around `GreenBlueBackground1S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `GreenBlueBackground.QI`
- **Data output** `QO` (`BOOL`, Quality Output): passed through from `GreenBlueBackground.QO`
- **Initialization chain**: `INIT` ⟶ `GreenBlueBackground.INIT`, `GreenBlueBackground.INITO` ⟶ `INITO`

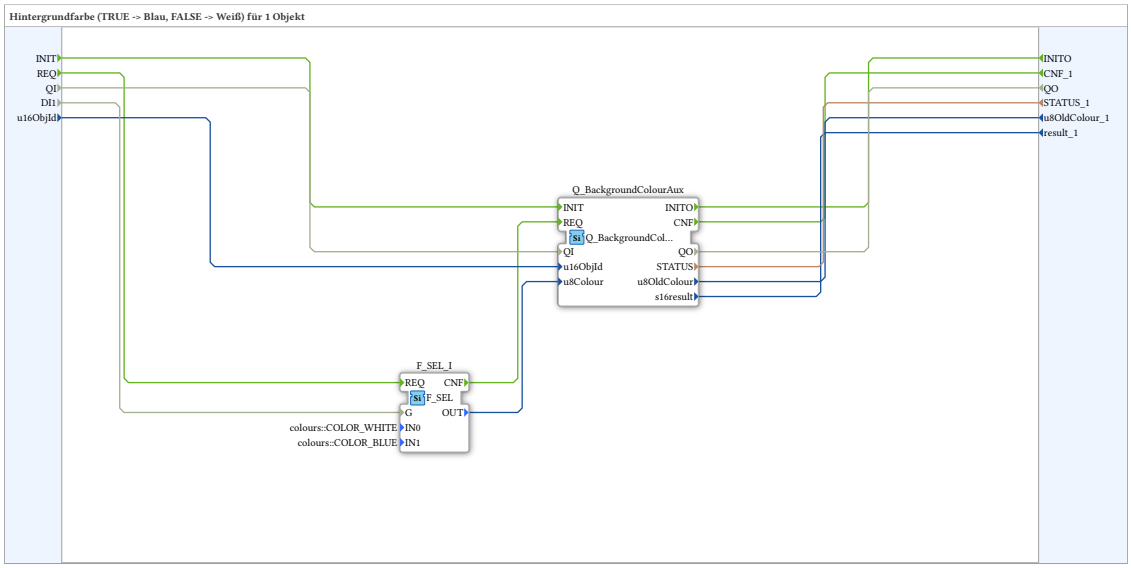
Summary

A pure pass-through to `GreenBlueBackground1S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenBlueBackground1_aux

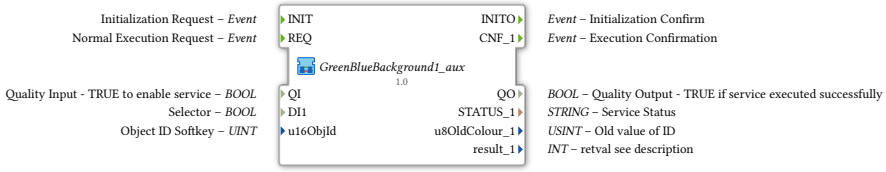


• — ○ • — ○ • — ○ •

Introduction

`GreenBlueBackground1_aux` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColourAux.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColourAux.QO`.

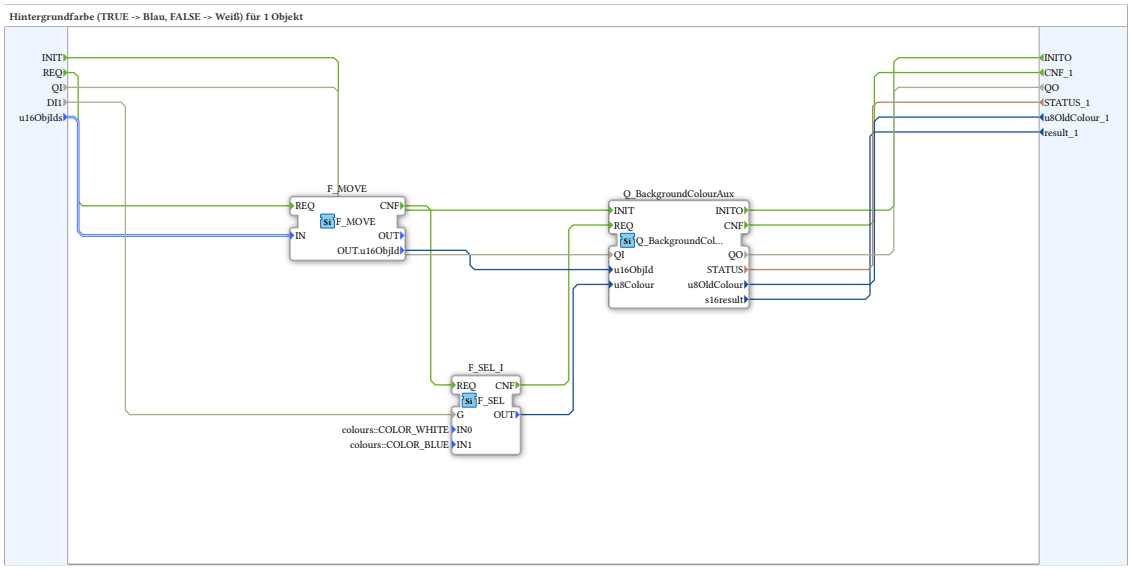
Summary

One of many variants in the background color block family: color pair Blau/Weiß, 1 object, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenBlueBackground1_auxS

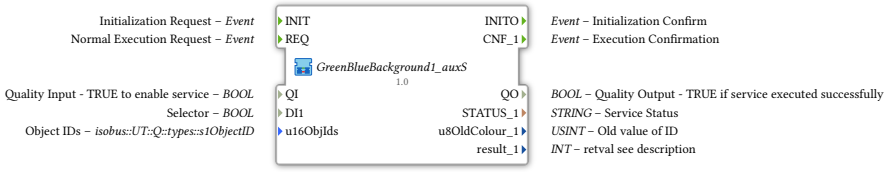


• – ○ • – ○ • – ○ •

Introduction

`GreenBlueBackground1_auxS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Blau**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColourAux.INIT`; `Q_BackgroundColourAux.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColourAux.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColourAux.QO`.

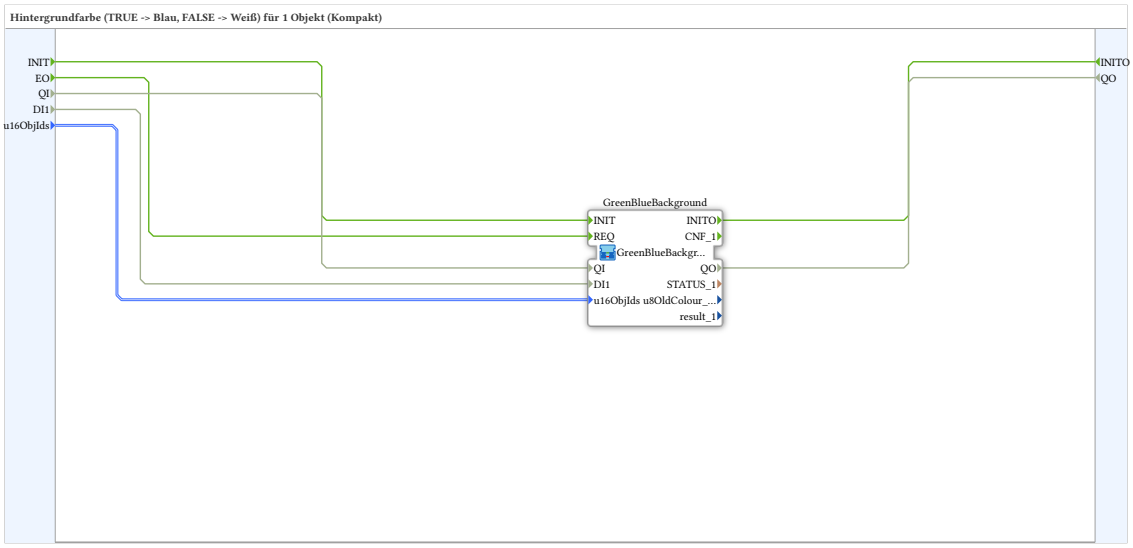
Summary

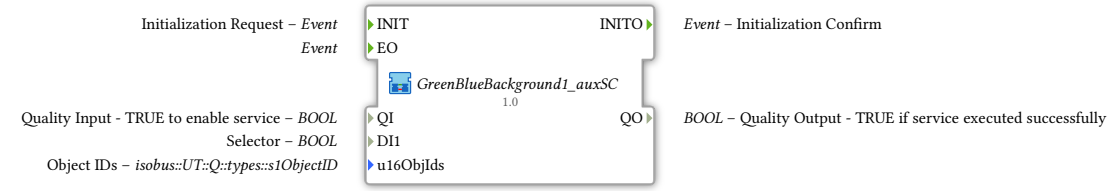
One of many variants in the background color block family: color pair Blau/Weiß, 1 object, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenBlueBackground1_auxSC





• — ○ • — ○ • — ○ •

Introduction

`GreenBlueBackground1_auxSC` is a **compact wrapper** around `GreenBlueBackground1_auxS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `GreenBlueBackground.INIT`, `GreenBlueBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenBlueBackground.QI` (initial value `TRUE`) and `GreenBlueBackground.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

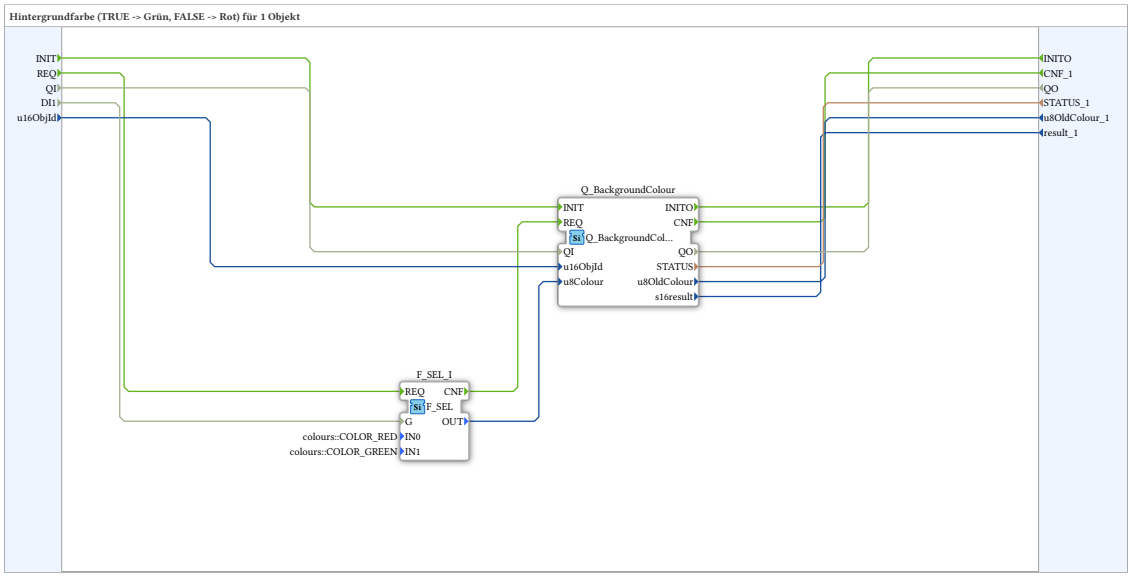
Summary

A pure pass-through to `GreenBlueBackground1_auxS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1

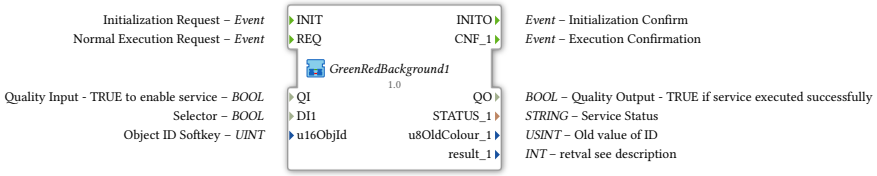


• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground1` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColour.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColour.QO`.

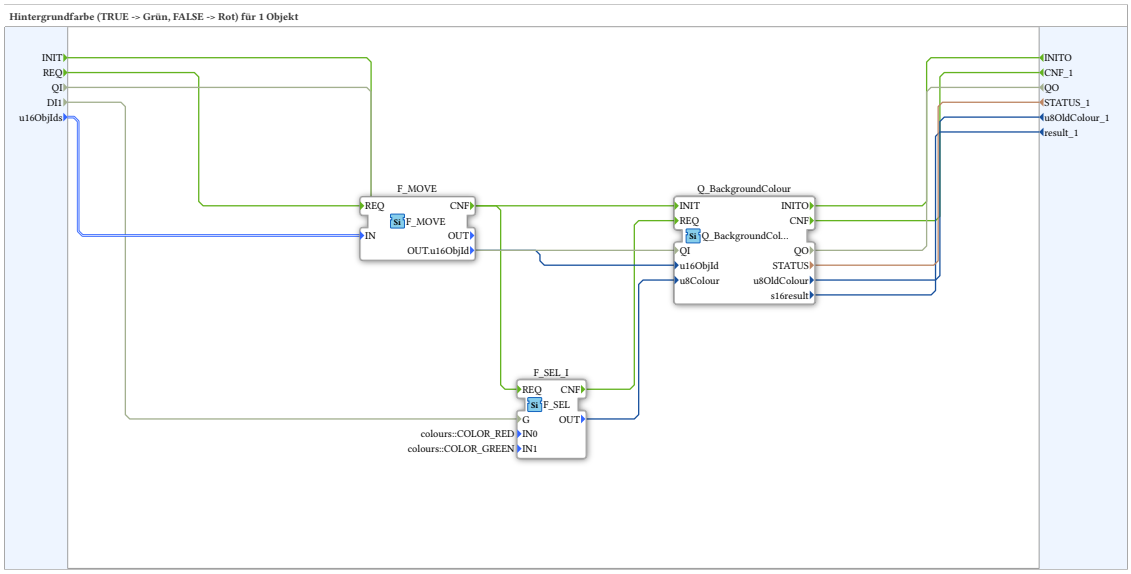
Summary

One of many variants in the background color block family: color pair Grün/Rot, 1 object, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1S

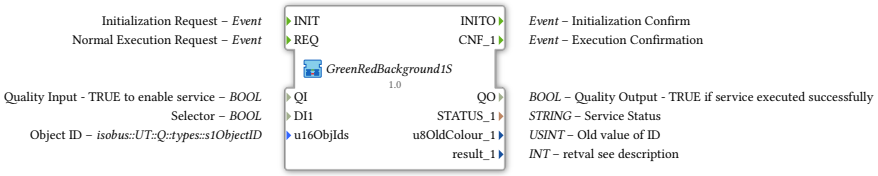


• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground1S` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Interface: Initialization and Quality

- **INIT** (event input, `EInit`) ⟶ `Q_BackgroundColour.INIT`; `Q_BackgroundColour.INITO` ⟶ **INITO** (event output, `EInit`).
- **QI** (data input, `BOOL`, default `TRUE`): Quality Input - `TRUE` enables the service; passed through to `Q_BackgroundColour.QI`.
- **QO** (data output, `BOOL`): Quality Output - `TRUE` if the service executed successfully; comes from `Q_BackgroundColour.QO`.

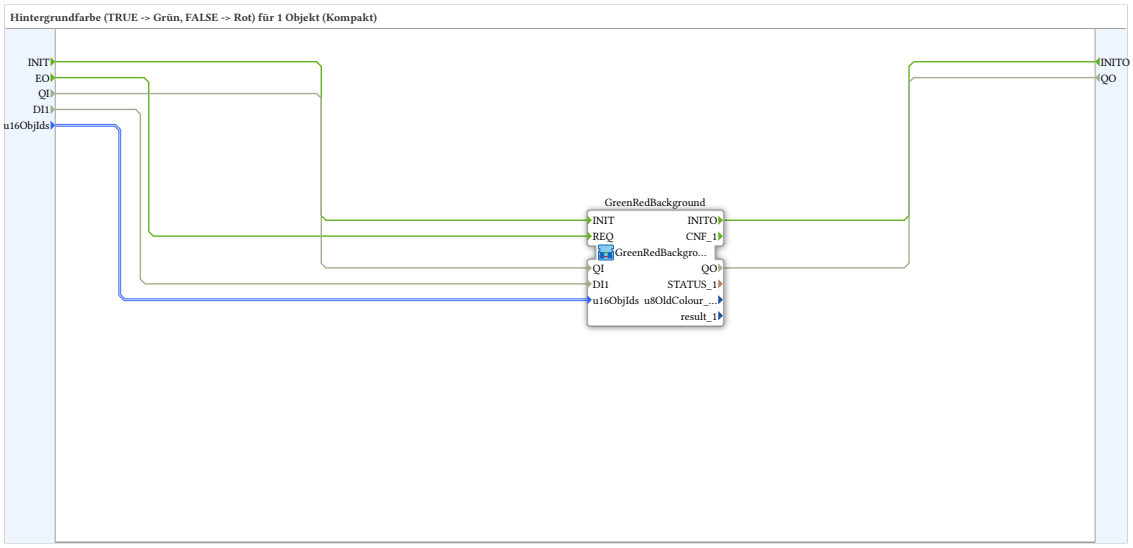
Summary

One of many variants in the background color block family: color pair Grün/Rot, 1 object, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1SC





• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground1SC` is a **compact wrapper** around `GreenRedBackground1S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

In addition, initialization and quality are passed through: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` (initial value `TRUE`) ⟶ `GreenRedBackground.QI` and `GreenRedBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

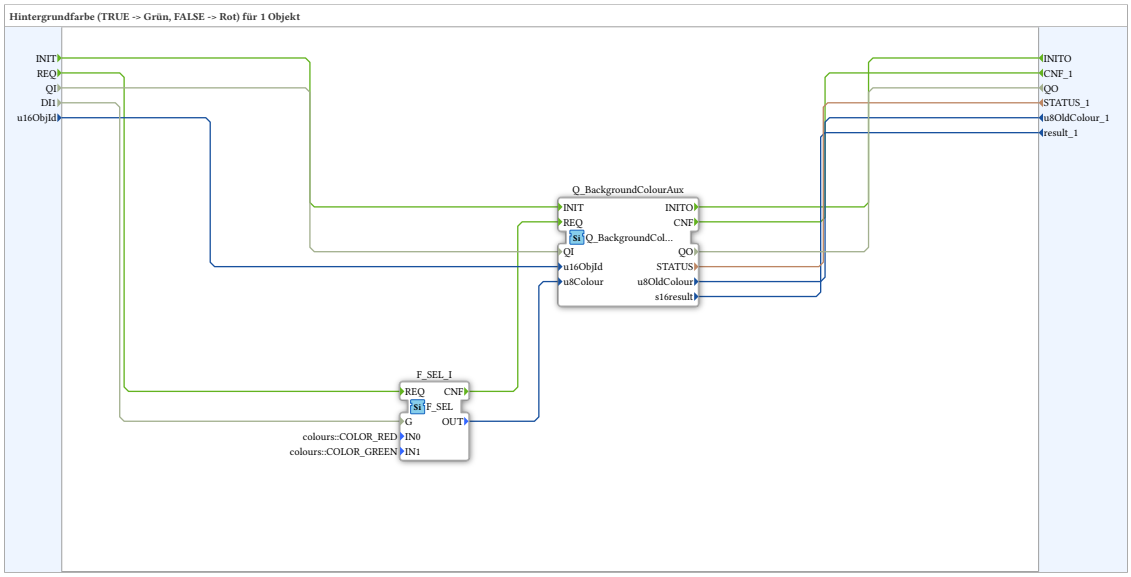
Summary

A pure pass-through to `GreenRedBackground1S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1_aux



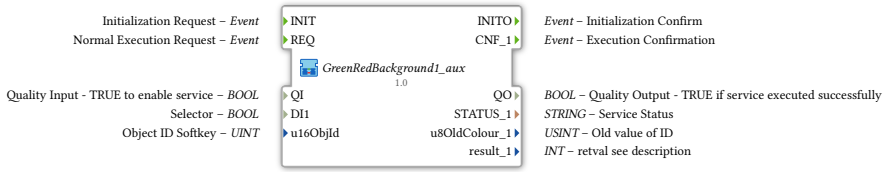
• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground1_aux` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColourAux.QI` and `Q_BackgroundColourAux.QO` to `QO`.



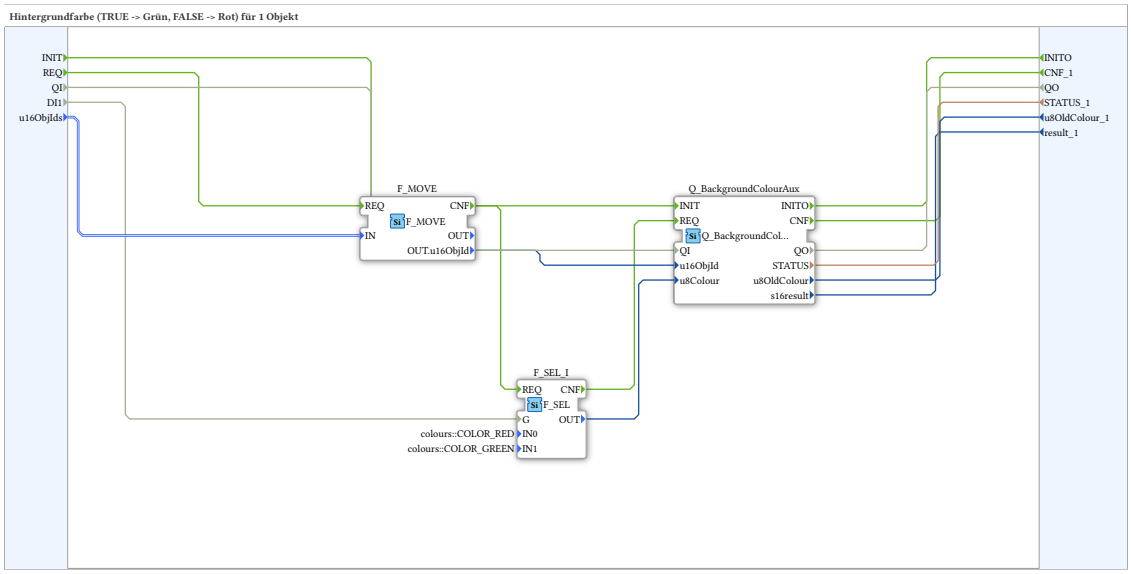
Summary

One of many variants in the background color block family: color pair Grün/Rot, 1 object, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1_auxS



• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground1_auxS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColourAux.QI` and `Q_BackgroundColourAux.QO` to `QO`.



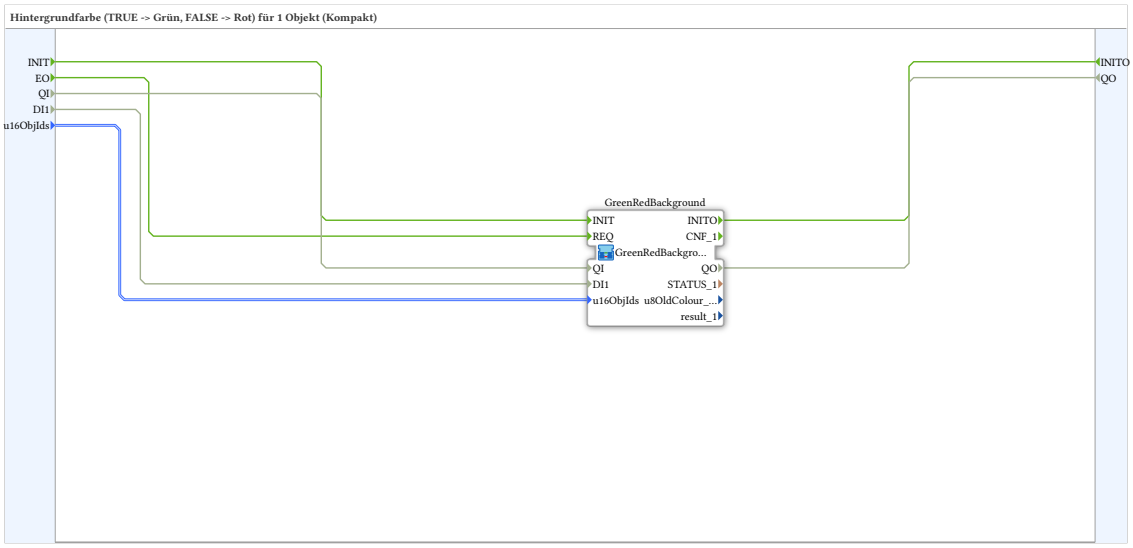
Summary

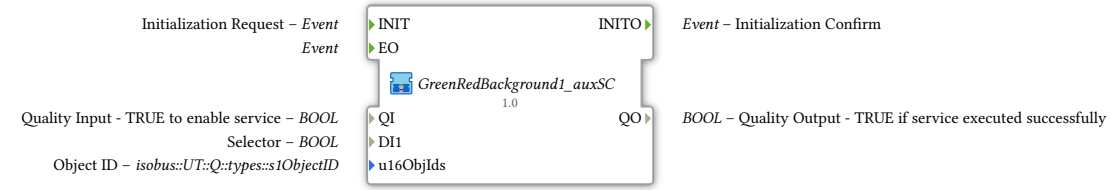
One of many variants in the background color block family: color pair Grün/Rot, 1 object, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground1_auxSC





• — ○ • — ○ • — ○ •

Introduction

`GreenRedBackground1_auxSC` is a **compact wrapper** around `GreenRedBackground1_auxS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `GreenRedBackground`: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenRedBackground.QI`, `GreenRedBackground.QO` ⟶ `QO`.

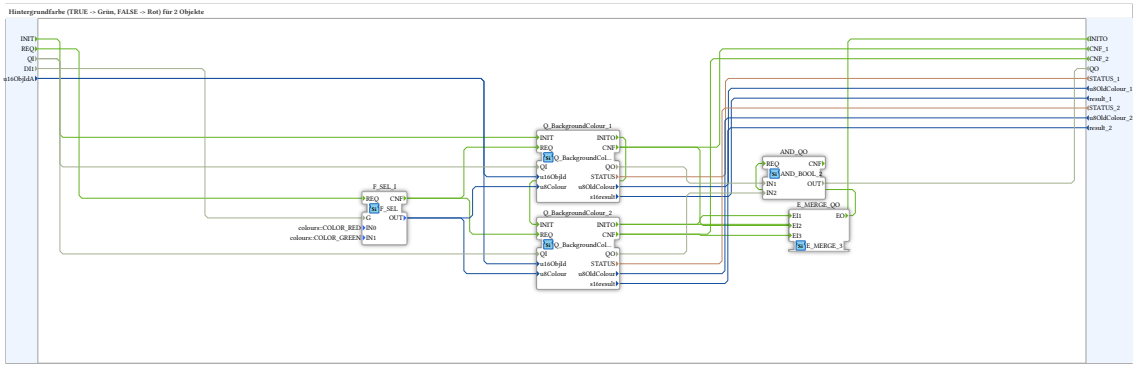
Summary

A pure pass-through to `GreenRedBackground1_auxS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground2



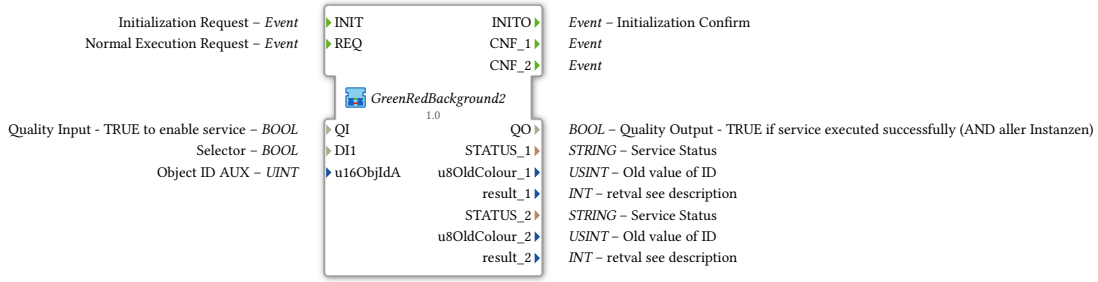
• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground2` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via inputs `u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_3`) merges `Q_BackgroundColour_2.INITO` and the `CNF` events of both instances (`Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`) and thereby triggers `AND_QO.REQ`.

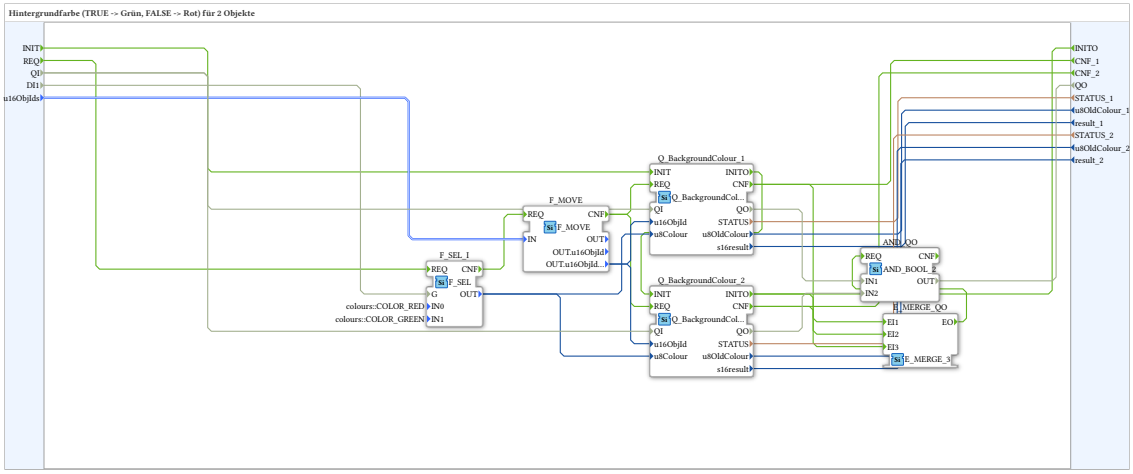
Summary

One of many variants in the background color block family: color pair Grün/Rot, 2 objects, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground2S



• – ○ • – ○ • – ○ •

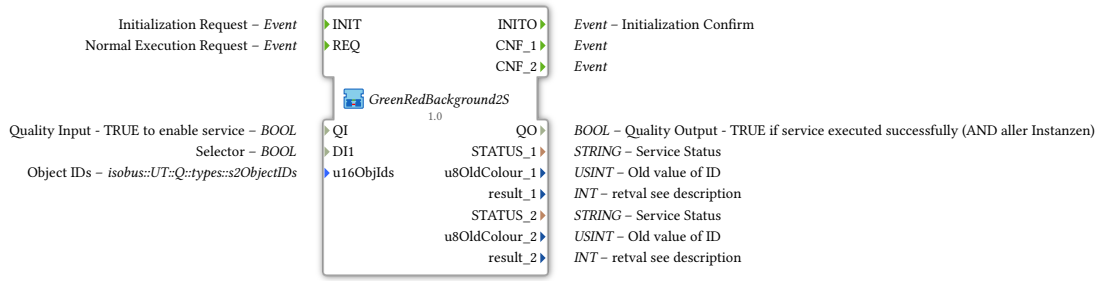
Introduction

`GreenRedBackground2S` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

Interface (quality pattern): event input `INIT` and event output `INITO` (each `EInit`), data input `QI` (`BOOL`, default `TRUE`) and data output `QO` (`BOOL`, AND of all instances). Initialization: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `INITO`. Quality: `QI` ⟶ `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`; `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN1` to `AND_QO.IN2` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_2`); `AND_QO.OUT` ⟶ `QO`. The trigger `AND_QO.REQ` comes via `E_MERGE_QO` (`E_MERGE_3`), which merges `Q_BackgroundColour_2.INITO` and the `CNF` of `Q_BackgroundColour_1` and `_2`; `E_MERGE_QO.EO` ⟶ `AND_QO.REQ`.

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



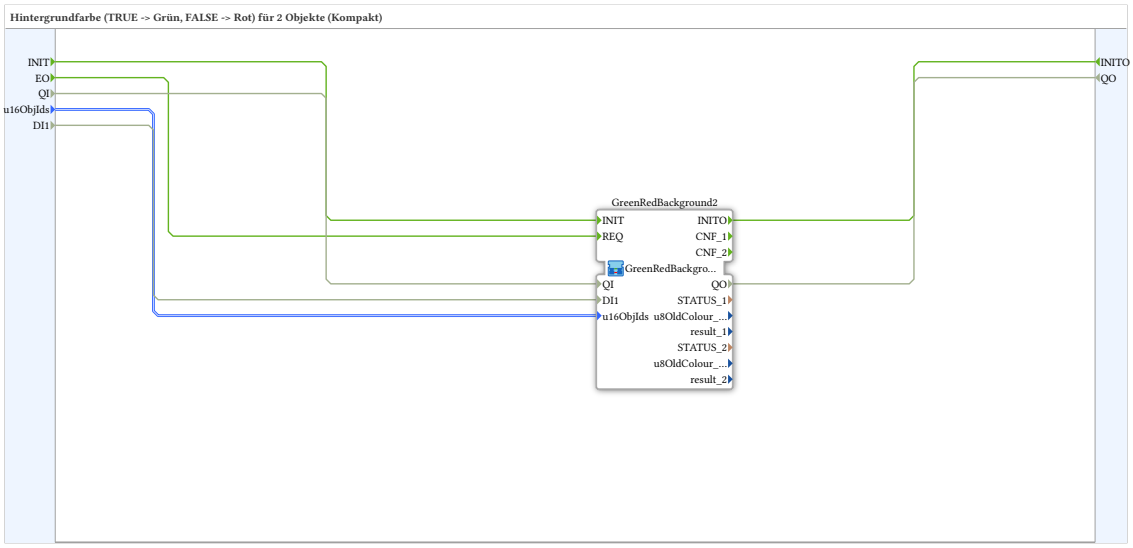
Summary

One of many variants in the background color block family: color pair Grün/Rot, 2 objects, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground2SC





• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground2SC` is a **compact wrapper** around `GreenRedBackground2S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `GreenRedBackground2.QI`
- **Data output** `QO` (`BOOL`, Quality Output): passed through from `GreenRedBackground2.QO`
- **Initialization chain**: `INIT` ⟶ `GreenRedBackground2.INIT`, `GreenRedBackground2.INITO` ⟶ `INITO`

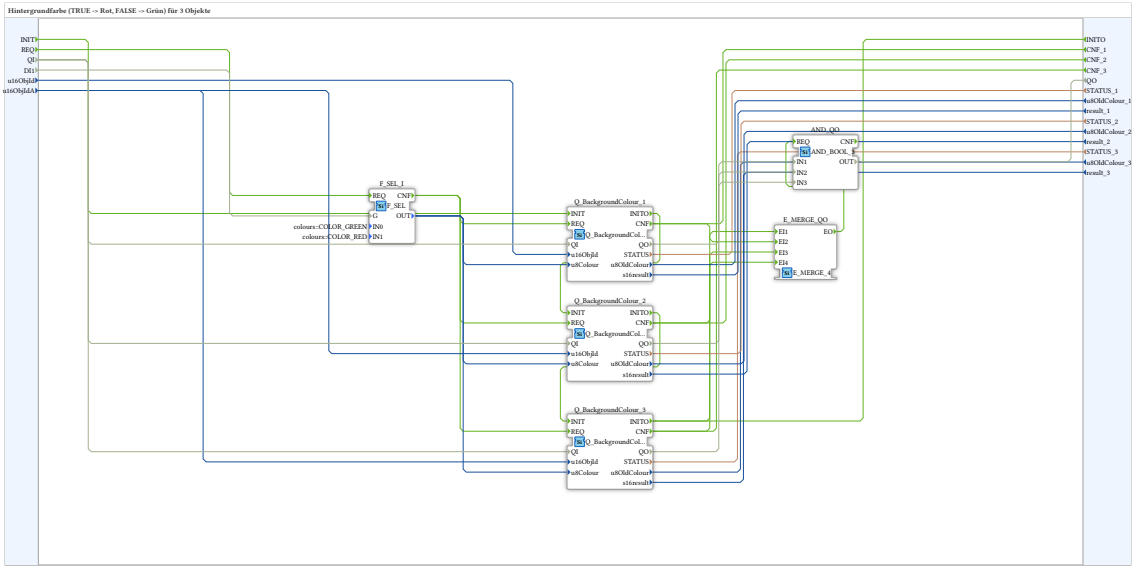
Summary

A pure pass-through to `GreenRedBackground2S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground3



• — ○ • — ○ • — ○ •

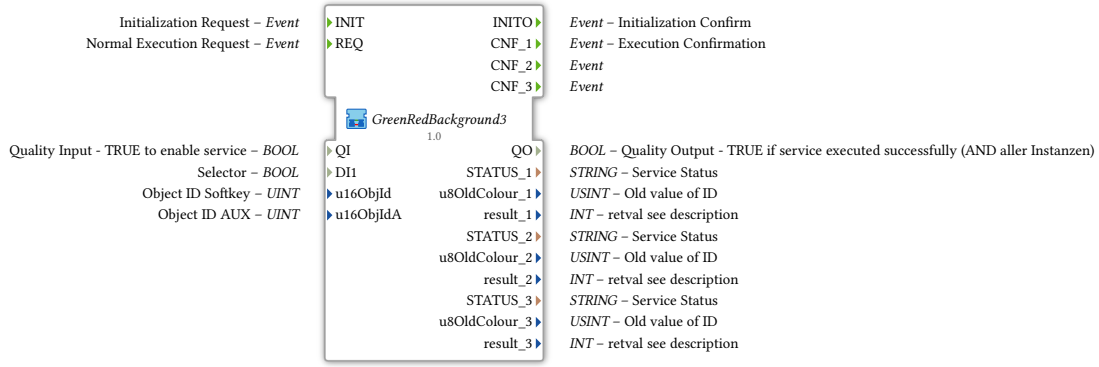
Introduction

`GreenRedBackground3` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Initialization and quality: The SubApp has `INIT`/`INITO` and `QI`/`QO`. `INIT` ⟶ `Q_BackgroundColour_1.INIT` ⟶ `Q_BackgroundColour_2.INIT` ⟶ `Q_BackgroundColour_3.INIT` (each via `INITO`), `Q_BackgroundColour_3.INITO` ⟶ `INITO`. `QI` goes to all three instances. `QO` is the `AND` (`AND_QO`, `AND_BOOL_3`) of the three `QO` outputs; `AND_QO.REQ` is triggered via `E_MERGE_QO` (`E_MERGE_4`: `Q_BackgroundColour_3.INITO`, `Q_BackgroundColour_1.CNF`, `_2.CNF`, `_3.CNF`).



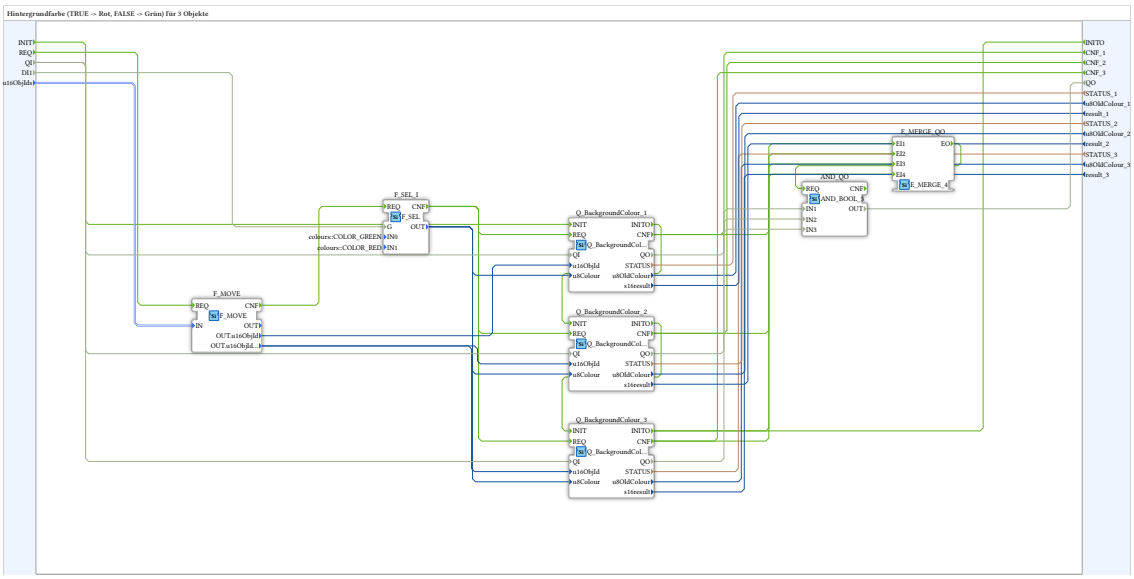
Summary

One of many variants in the background color block family: color pair Rot/Grün, 3 objects, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground3S



• – ○ • – ○ • – ○ •

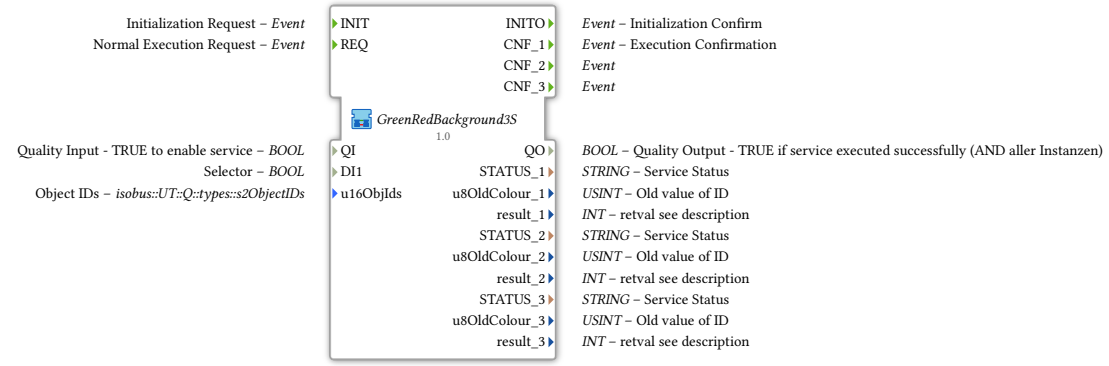
Introduction

`GreenRedBackground3S` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

Initialization and quality: In addition to the previous events, the block has the event input `INIT`, the event output `INITO`, as well as `QI` (BOOL, input, initial value `TRUE`) and `QO` (BOOL, output, AND of all instances). Init chain: `INIT` ⟶ `Q_BackgroundColour_1` ⟶ `Q_BackgroundColour_2` ⟶ `Q_BackgroundColour_3` ⟶ `INITO` (each `INITO` ⟶ next `INIT`). `QI` is passed on to all `Q_BackgroundColour_n.QI`. `QO` is formed by `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `AND_QO.OUT` ⟶ `QO`. `E_MERGE_QO` (`E_MERGE_4`) merges `Q_BackgroundColour_3.INITO` and the `CNF` outputs `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`; `E_MERGE_QO.EO` triggers `AND_QO.REQ`.

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



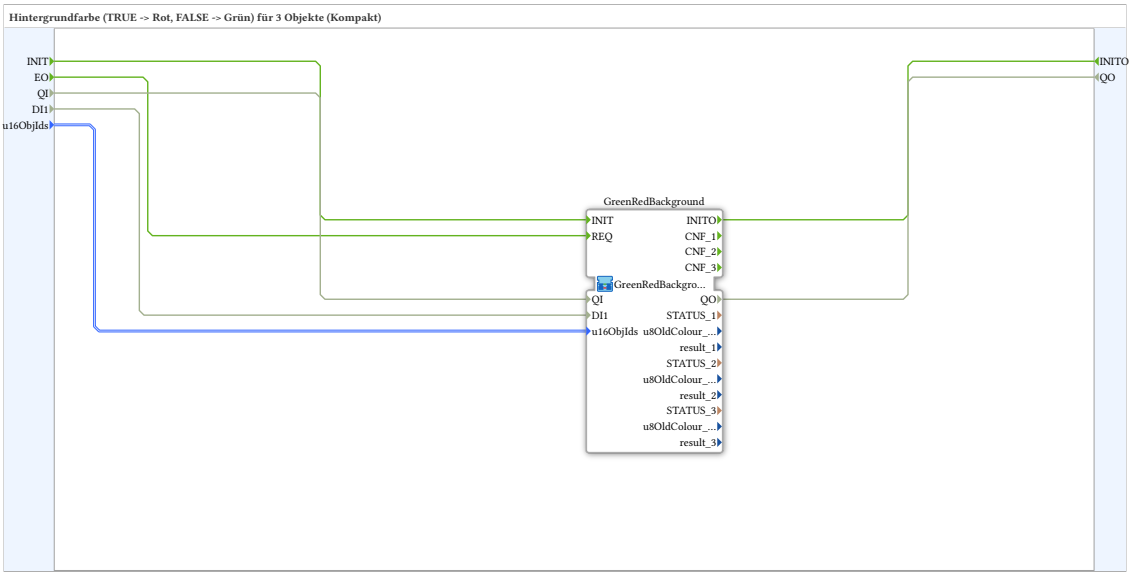
Summary

One of many variants in the background color block family: color pair Rot/Grün, 3 objects, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground3SC





• — ○ • — ○ • — ○ •

Introduction

`GreenRedBackground3SC` is a **compact wrapper** around `GreenRedBackground3S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the

inner instance `GreenRedBackground`: `INIT` ⟶ `GreenRedBackground.INIT`, `GreenRedBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenRedBackground.QI`, `GreenRedBackground.QO` ⟶ `QO`.

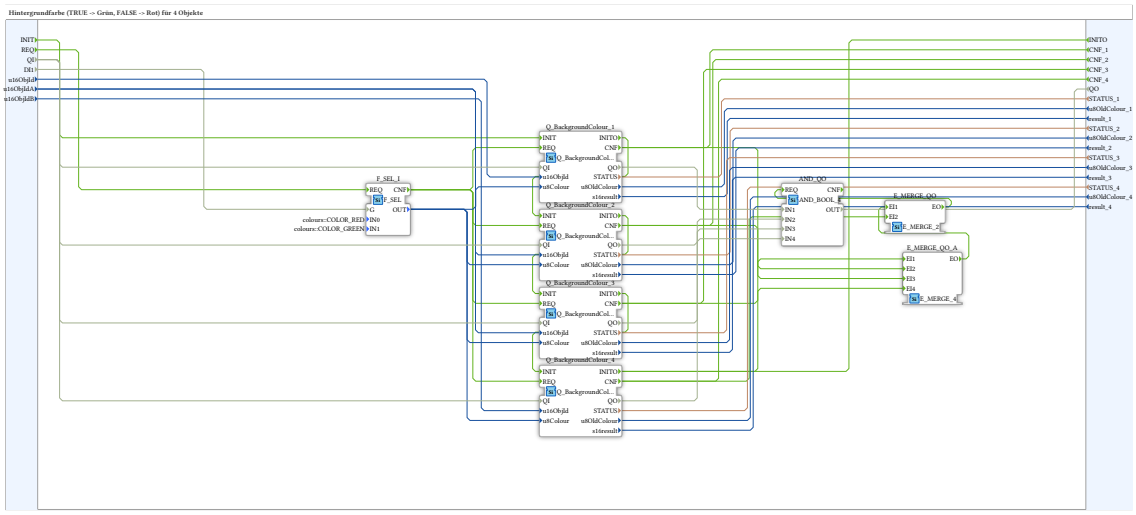
Summary

A pure pass-through to `GreenRedBackground3S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground4



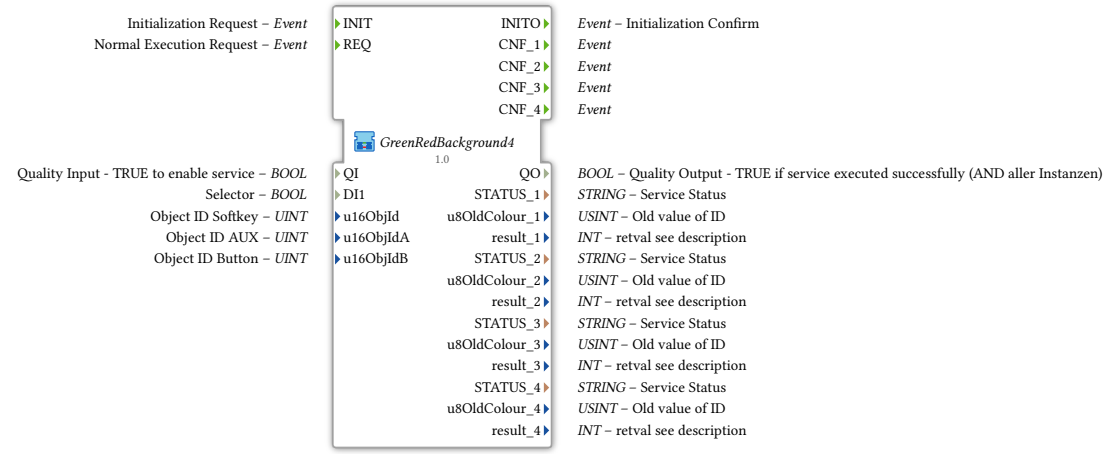
• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground4` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA, u16ObjIdB`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`, `Q_BackgroundColour_4`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO_A` (`iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_4.INITO` and the `CNF` events of instances 1 to 3; `E_MERGE_QO` (`iec61499::events::E_MERGE_2`) merges its output `E_MERGE_QO_A.EO` with `Q_BackgroundColour_4.CNF` and thereby triggers `AND_QO.REQ`.

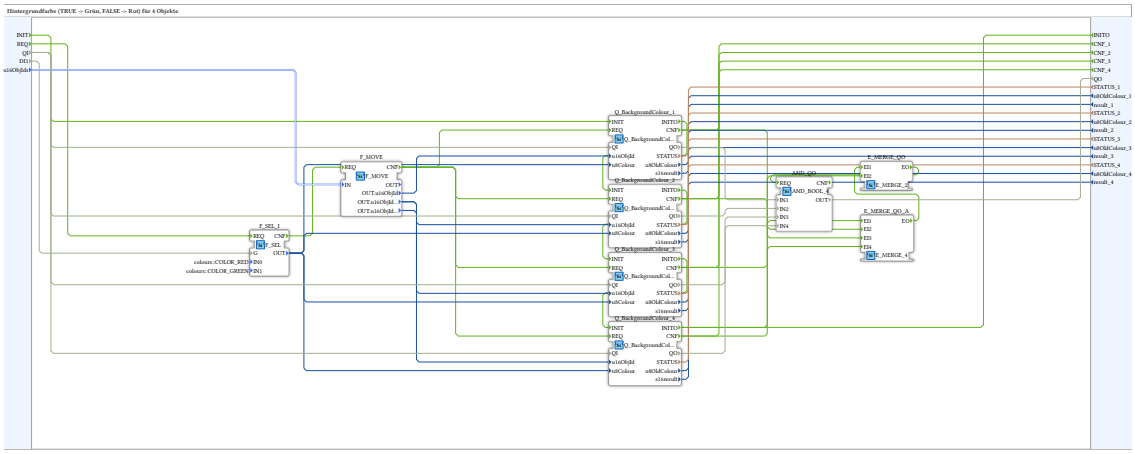
Summary

One of many variants in the background color block family: color pair Grün/Rot, 4 objects, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground4S



• – ○ • – ○ • – ○ •

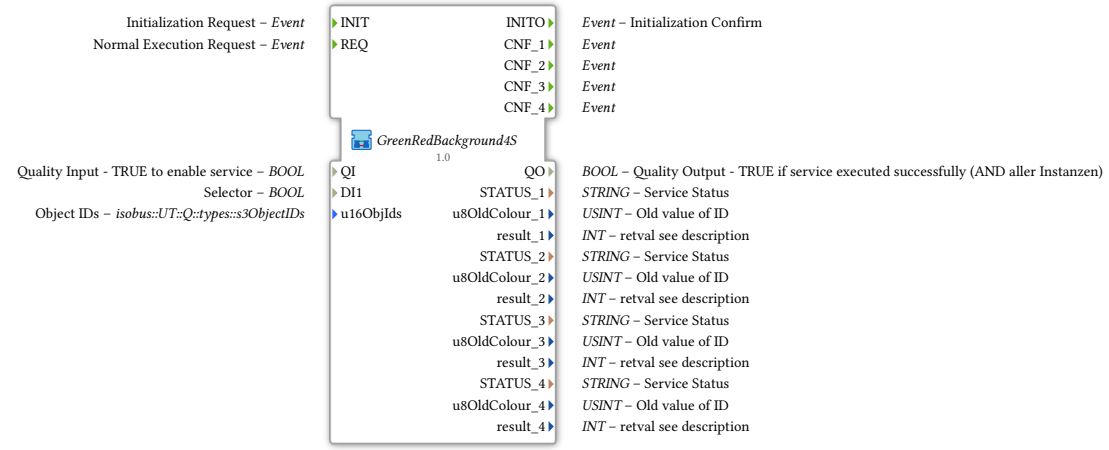
Introduction

`GreenRedBackground4S` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Rot**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (regular object)

Interface (quality pattern): event input `INIT` and event output `INITO` (each `EInit`), data input `QI` (`BOOL`, default `TRUE`) and data output `QO` (`BOOL`, AND of all instances). Initialization: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` ⟶ `INITO`. Quality: `QI` ⟶ `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI`, `Q_BackgroundColour_4.QI`; `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN1` to `AND_QO.IN4` (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_4`); `AND_QO.OUT` ⟶ `QO`. The trigger `AND_QO.REQ` comes via two merge blocks: `E_MERGE_QO_A` (`E_MERGE_4`) merges `Q_BackgroundColour_4.INITO` and the `CNF` of `Q_BackgroundColour_1` to `_3`, and `E_MERGE_QO` (`E_MERGE_2`) merges its output with `Q_BackgroundColour_4.CNF`; `E_MERGE_QO.EO` ⟶ `AND_QO.REQ`.

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



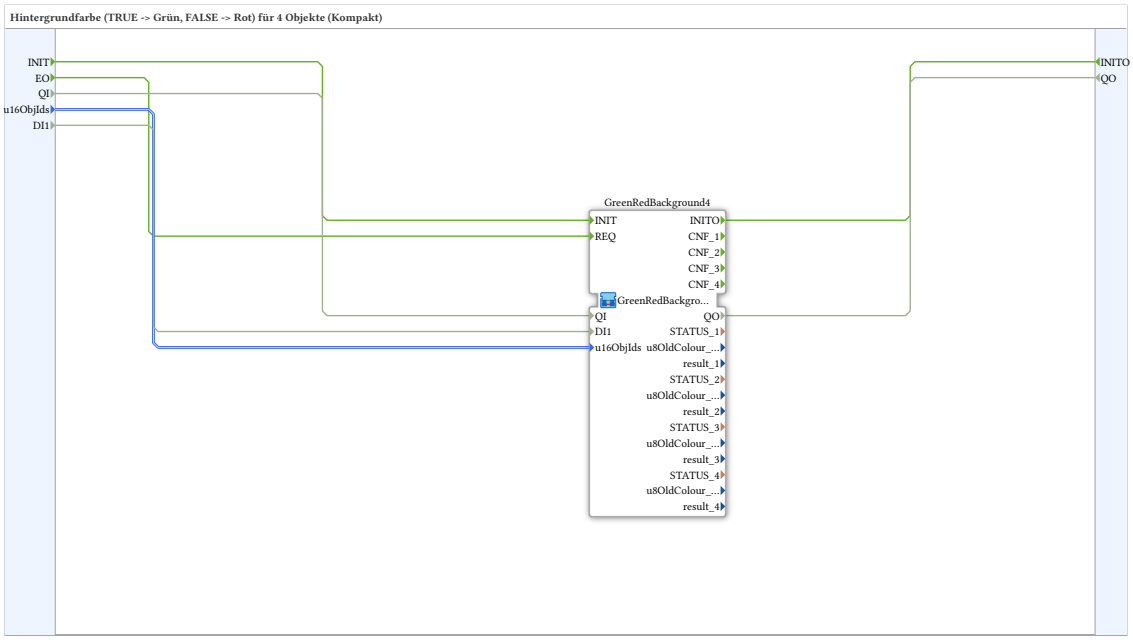
Summary

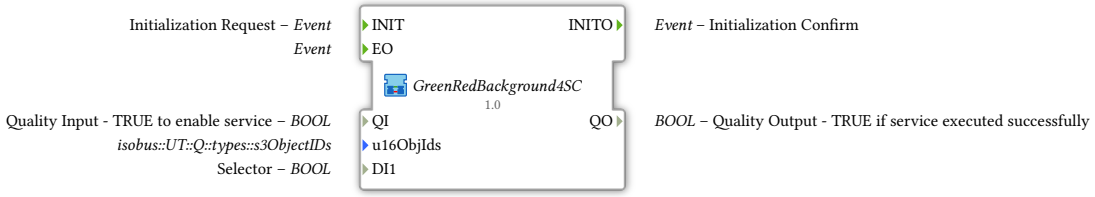
One of many variants in the background color block family: color pair Grün/Rot, 4 objects, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenRedBackground4SC





• – ○ • – ○ • – ○ •

Introduction

`GreenRedBackground4SC` is a **compact wrapper** around `GreenRedBackground4S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `GreenRedBackground4.QI`
- **Data output** `QO` (`BOOL`, Quality Output): passed through from `GreenRedBackground4.QO`
- **Initialization chain**: `INIT` ⟶ `GreenRedBackground4.INIT`, `GreenRedBackground4.INITO` ⟶ `INITO`

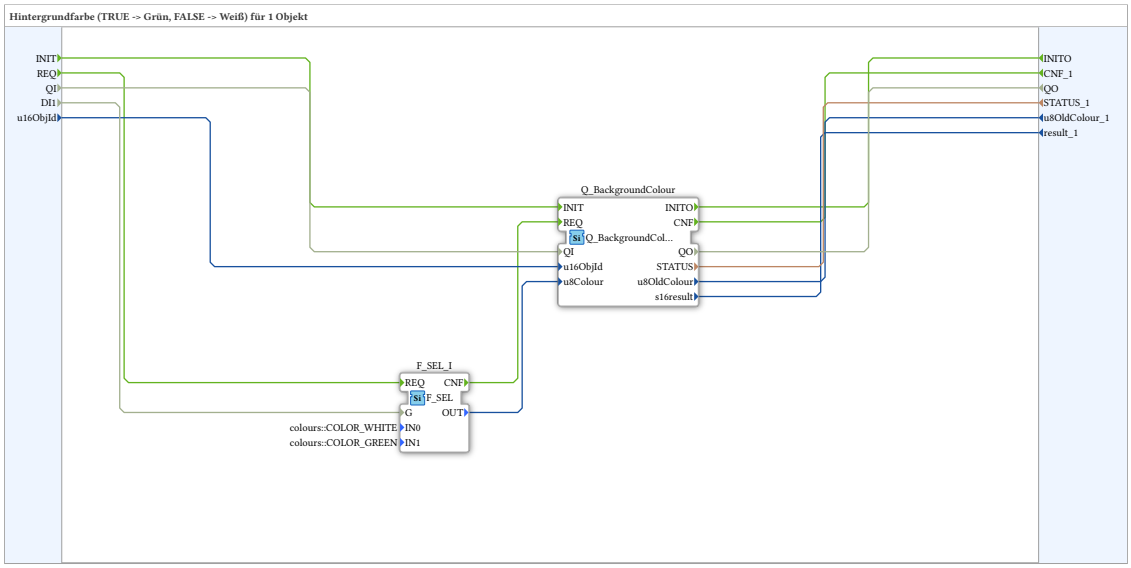
Summary

A pure pass-through to `GreenRedBackground4S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1



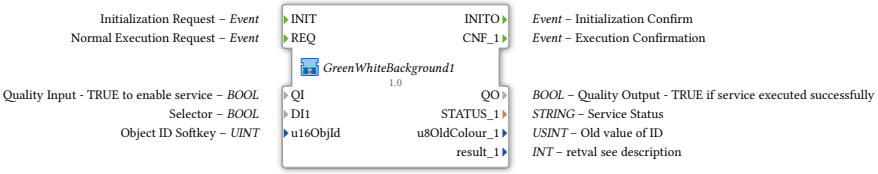
• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground1` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColour.QI` and `Q_BackgroundColour.QO` to `QO`.



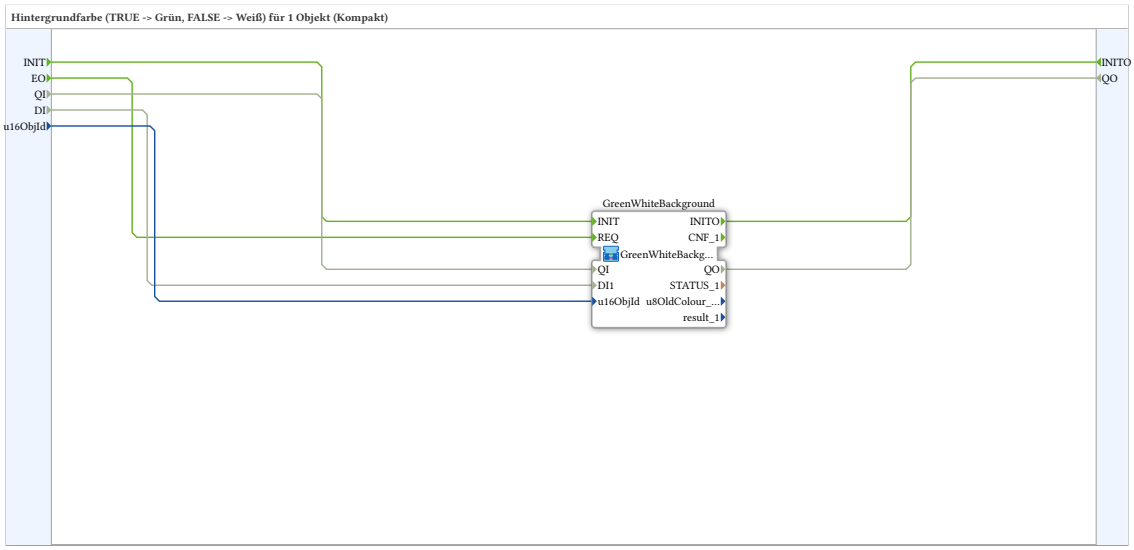
Summary

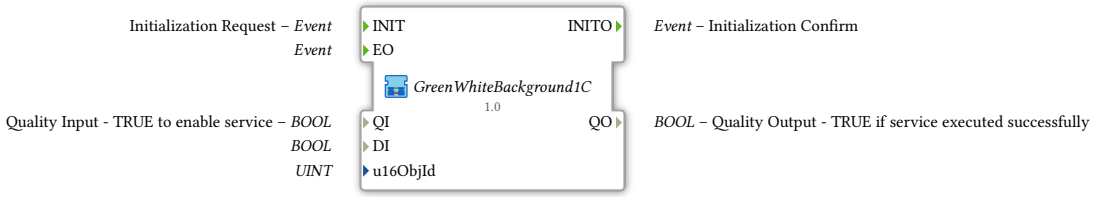
One of many variants in the background color block family: color pair Grün/Weiß, 1 object, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1C





• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground1C` is a **compact wrapper** around `GreenWhiteBackground1` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI` (initial value `TRUE`) and `GreenWhiteBackground.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

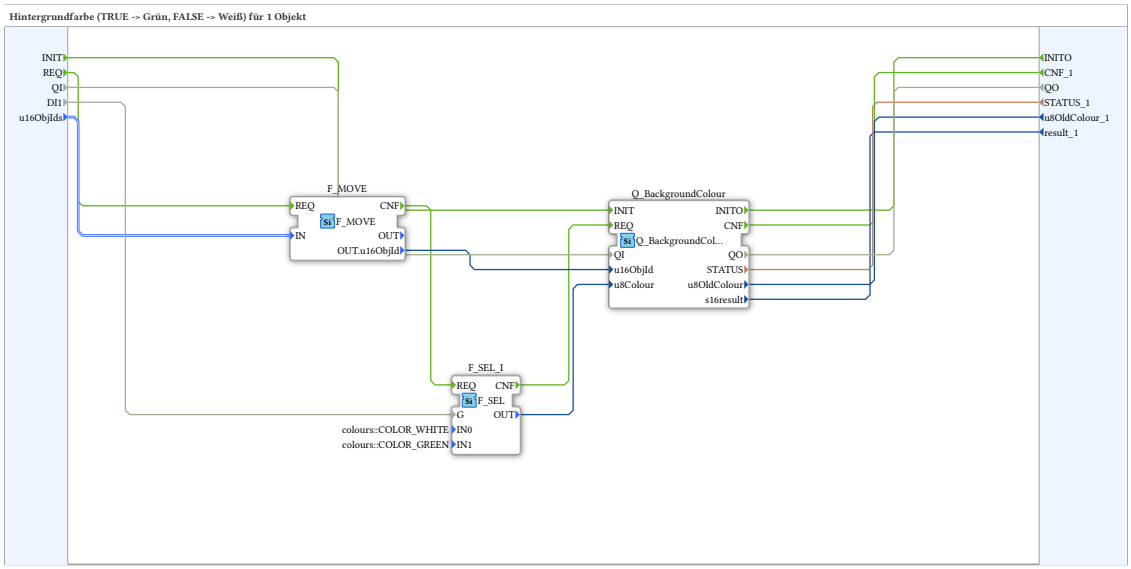
Summary

A pure pass-through to `GreenWhiteBackground1` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1S



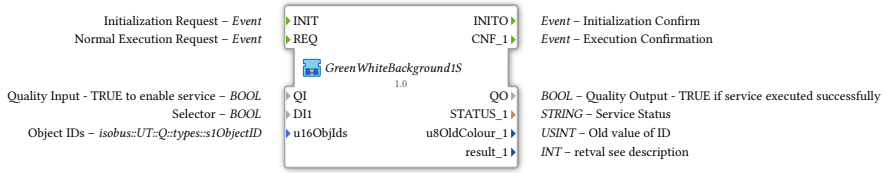
• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground1S` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColour.QI` and `Q_BackgroundColour.QO` to `QO`.



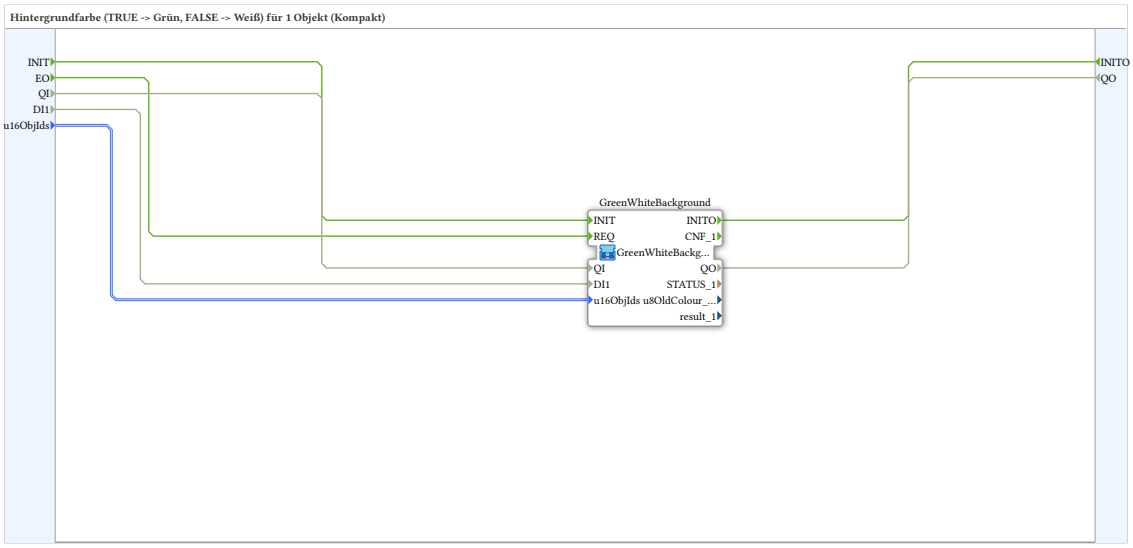
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 1 object, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1SC





• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground1SC` is a **compact wrapper** around `GreenWhiteBackground1S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

In addition, initialization and quality are passed through: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` (initial value `TRUE`) ⟶ `GreenWhiteBackground.QI` and `GreenWhiteBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

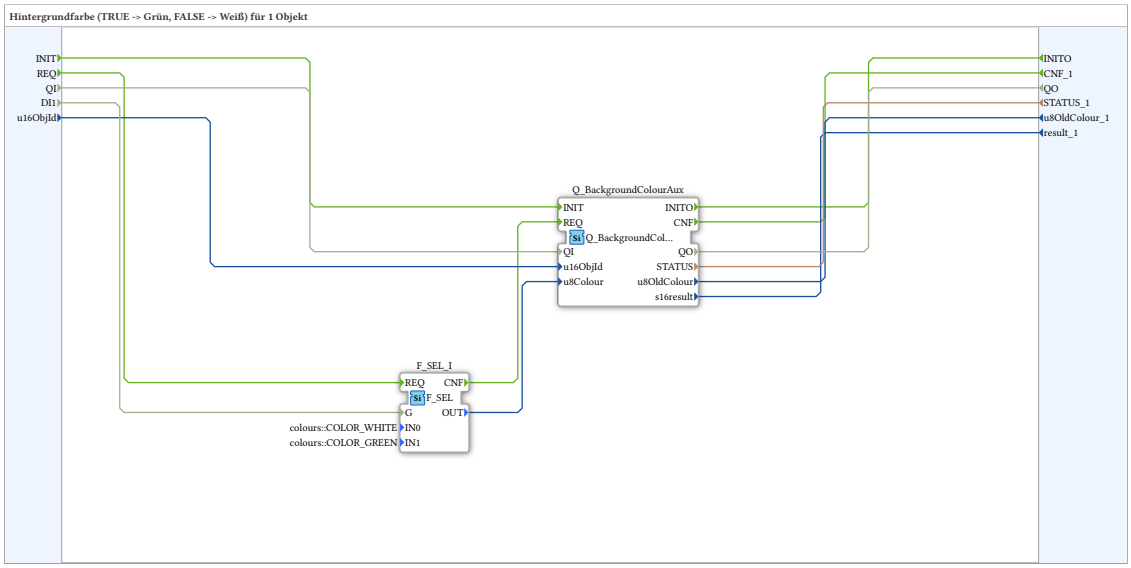
Summary

A pure pass-through to `GreenWhiteBackground1S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1_aux



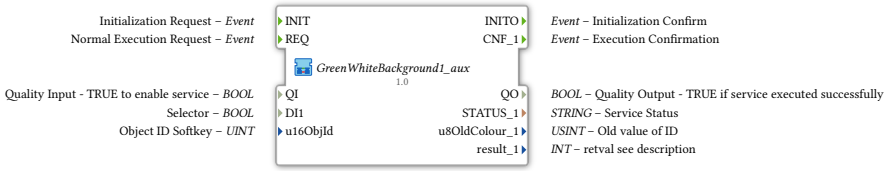
• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground1_aux` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColourAux.QI` and `Q_BackgroundColourAux.QO` to `QO`.



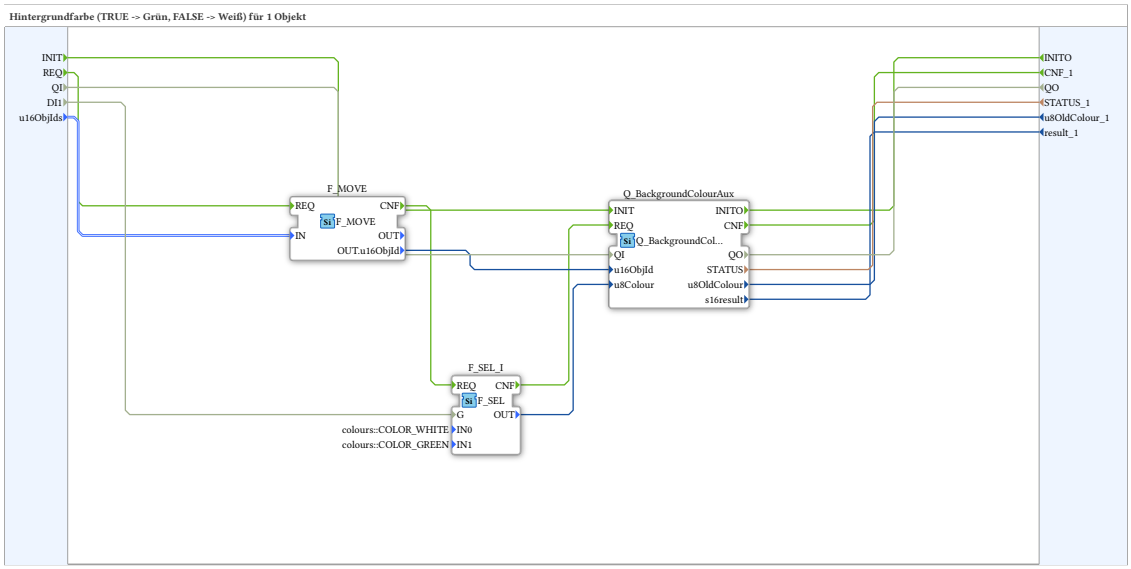
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 1 object, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1_auxS



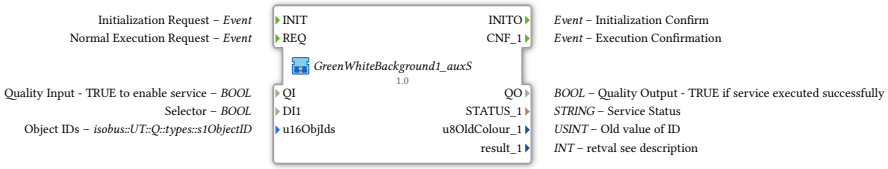
• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground1_auxS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColourAux.QI` and `Q_BackgroundColourAux.QO` to `QO`.



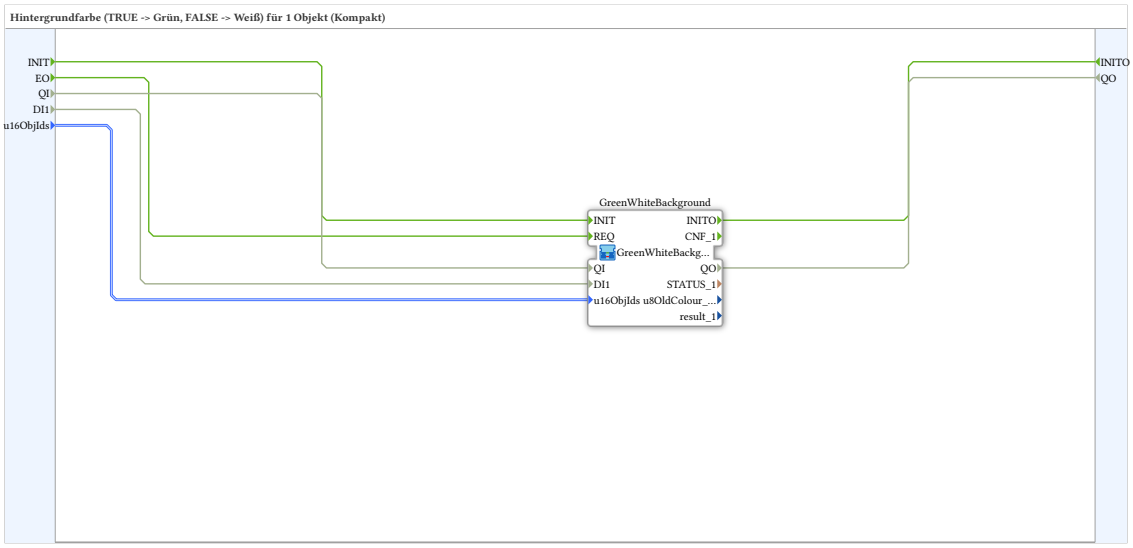
Summary

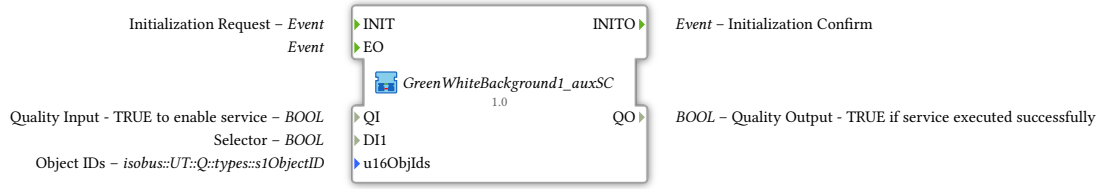
One of many variants in the background color block family: color pair Grün/Weiß, 1 object, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground1_auxSC





• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground1_auxSC` is a **compact wrapper** around `GreenWhiteBackground1_auxS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `GreenWhiteBackground`: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

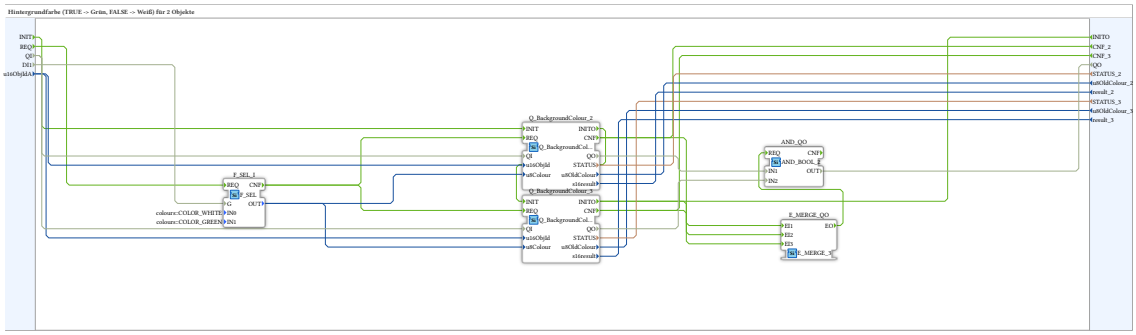
Summary

A pure pass-through to `GreenWhiteBackground1_auxS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground2



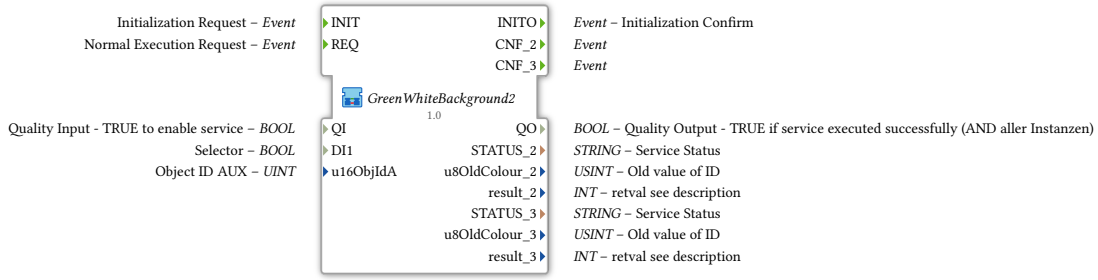
• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground2` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via inputs `u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_2`, `Q_BackgroundColour_3`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_3`) merges `Q_BackgroundColour_3.INITO` and the `CNF` events of both instances (`Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`) and thereby triggers `AND_QO.REQ`.

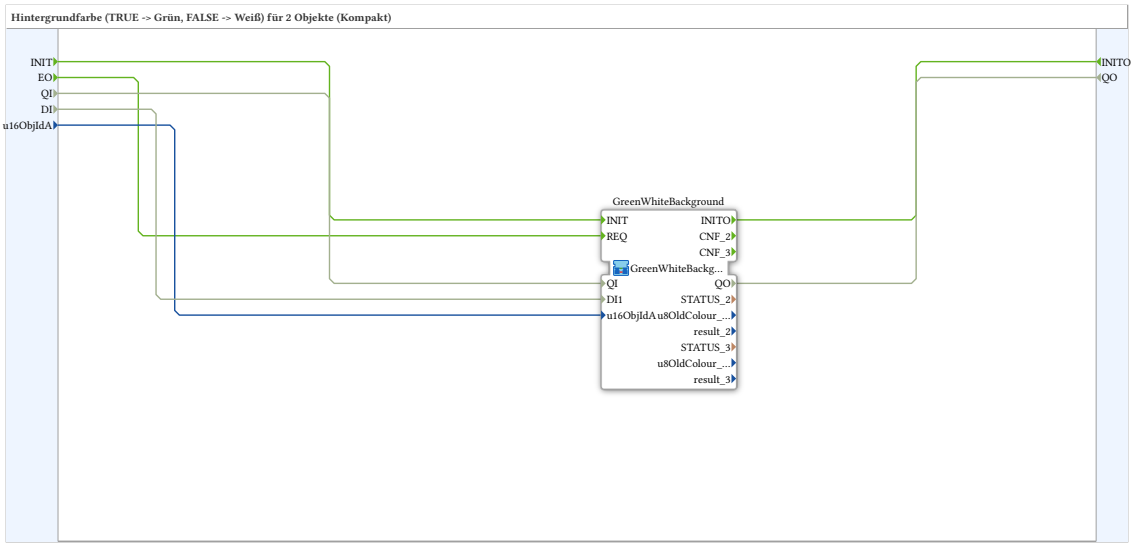
Summary

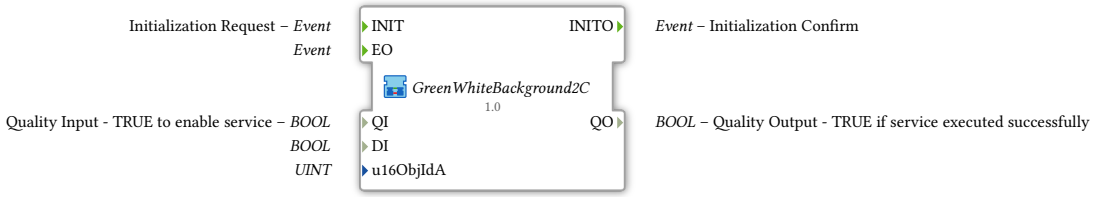
One of many variants in the background color block family: color pair Grün/Weiß, 2 objects, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground2C





• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground2C` is a **compact wrapper** around `GreenWhiteBackground2` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `GreenWhiteBackground`: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

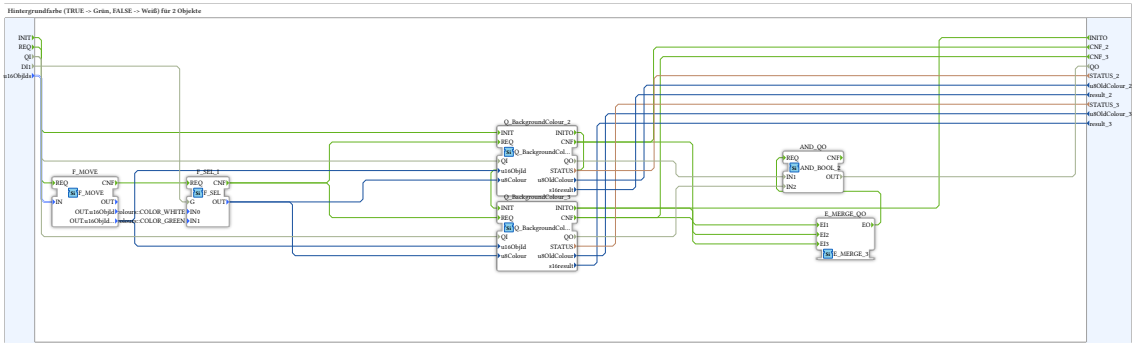
Summary

A pure pass-through to `GreenWhiteBackground2` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground2S



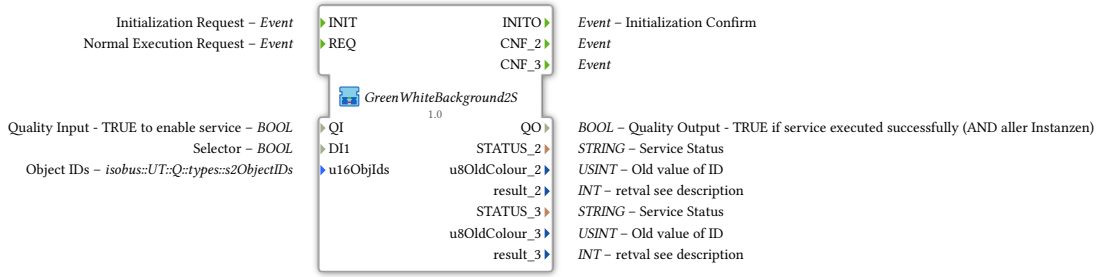
• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground2S` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` → **Grün**, `FALSE` → **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector → `AX_SEL`/`F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_BackgroundColour_2.QI` and `Q_BackgroundColour_3.QI`
- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_2`) of the `QO` outputs of `Q_BackgroundColour_2.QO` and `Q_BackgroundColour_3.QO`
- **Initialization chain**: `INIT` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `INITO`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_3`) merges `Q_BackgroundColour_3.INITO`, `Q_BackgroundColour_2.CNF` and `Q_BackgroundColour_3.CNF` and triggers `AND_QO.REQ`

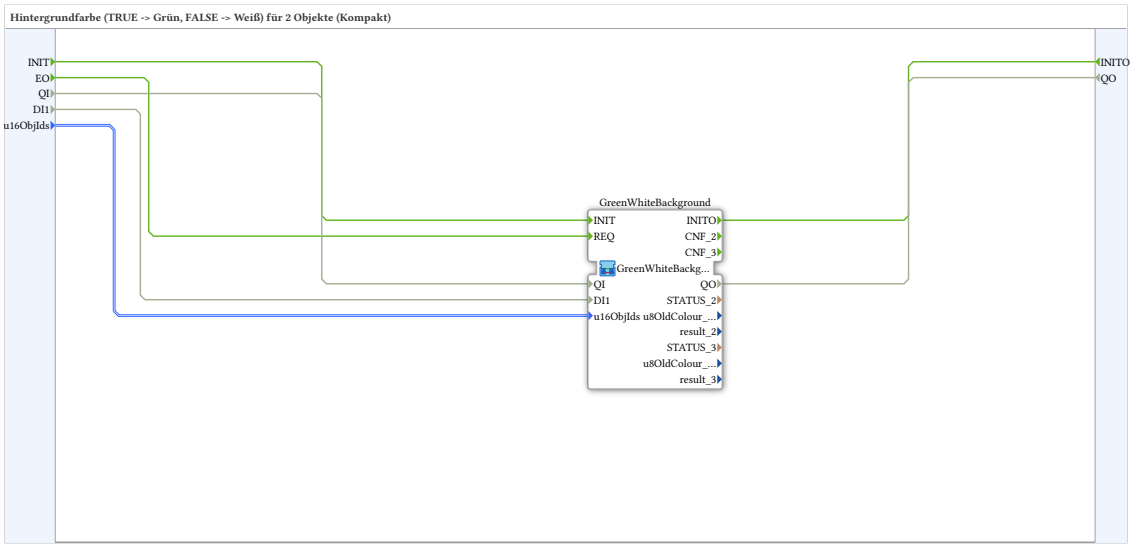
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 2 objects, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground2SC





• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground2SC` is a **compact wrapper** around `GreenWhiteBackground2S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI` (initial value `TRUE`) and `GreenWhiteBackground.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

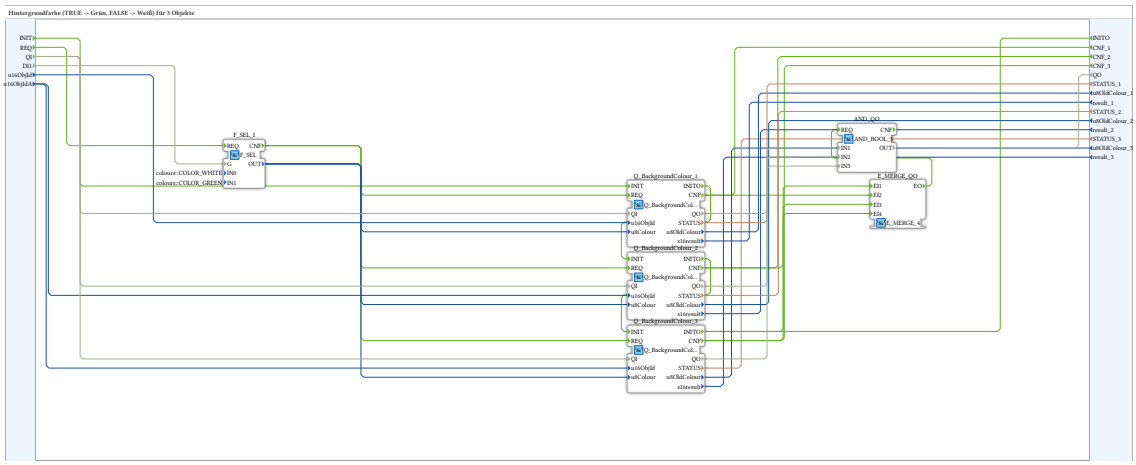
Summary

A pure pass-through to `GreenWhiteBackground2S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground3



• – ○ • – ○ • – ○ •

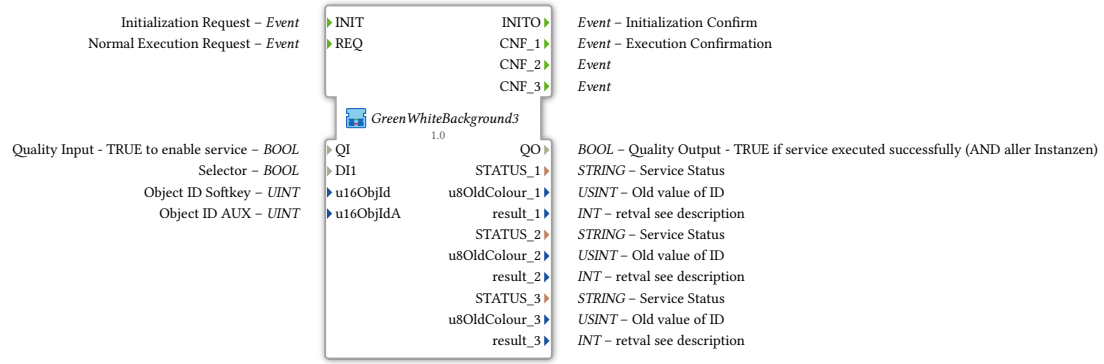
Introduction

`GreenWhiteBackground3` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

Initialization and quality: In addition to the previous events, the block has the event input `INIT`, the event output `INITO`, as well as `QI` (BOOL, input, initial value `TRUE`) and `QO` (BOOL, output, AND of all instances). Init chain: `INIT` ⟶ `Q_BackgroundColour_1` ⟶ `Q_BackgroundColour_2` ⟶ `Q_BackgroundColour_3` ⟶ `INITO` (each `INITO` ⟶ next `INIT`). `QI` is passed on to all `Q_BackgroundColour_n.QI`. `QO` is formed by `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `AND_QO.OUT` ⟶ `QO`. `E_MERGE_QO` (`E_MERGE_4`) merges `Q_BackgroundColour_3.INITO` and the `CNF` outputs `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`; `E_MERGE_QO.EO` triggers `AND_QO.REQ`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



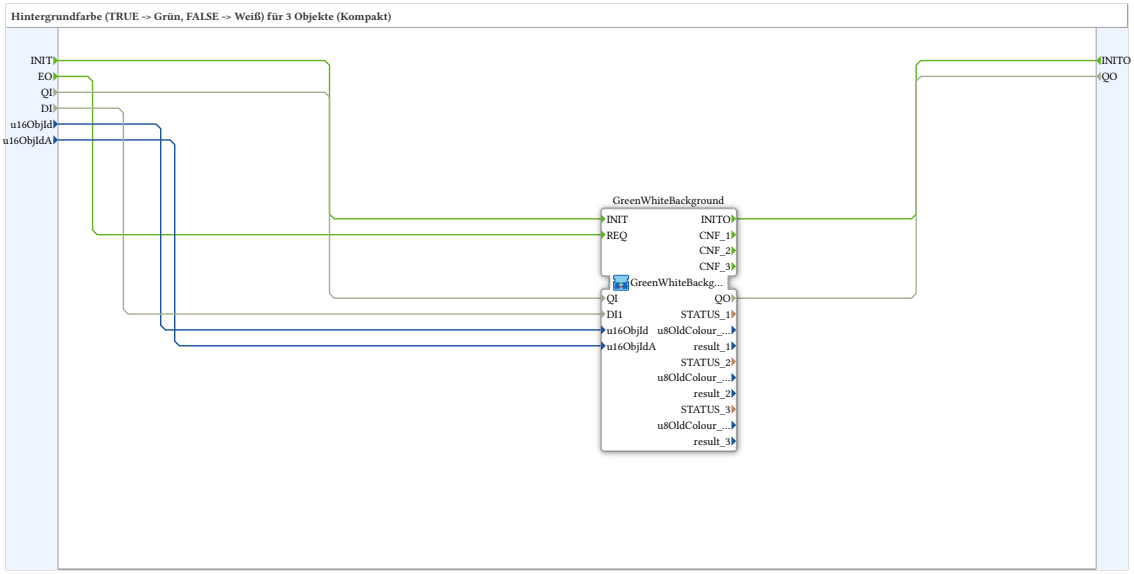
Summary

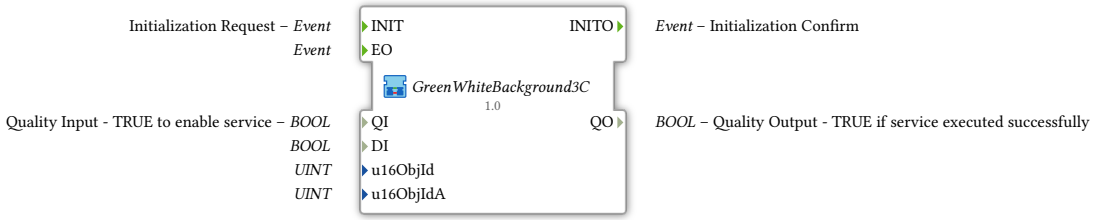
One of many variants in the background color block family: color pair Grün/Weiß, 3 objects, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground3C





• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground3C` is a **compact wrapper** around `GreenWhiteBackground3` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `GreenWhiteBackground`: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

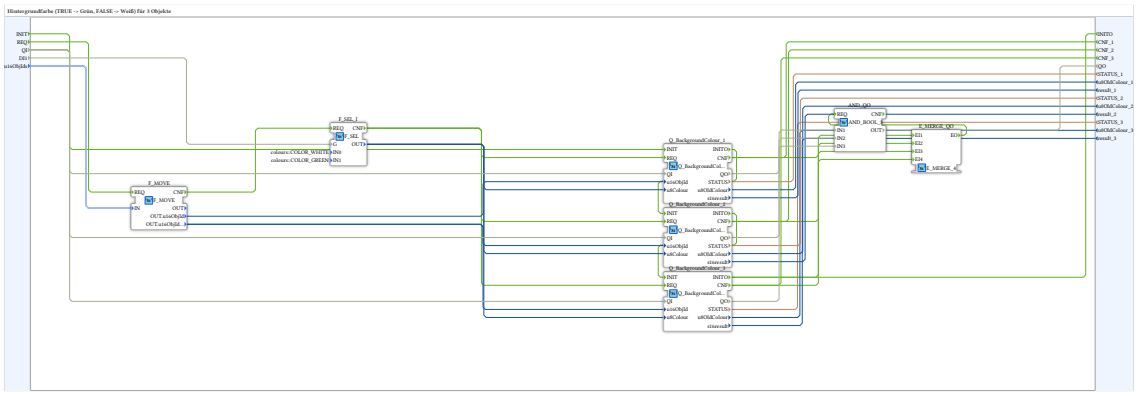
Summary

A pure pass-through to `GreenWhiteBackground3` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground3S



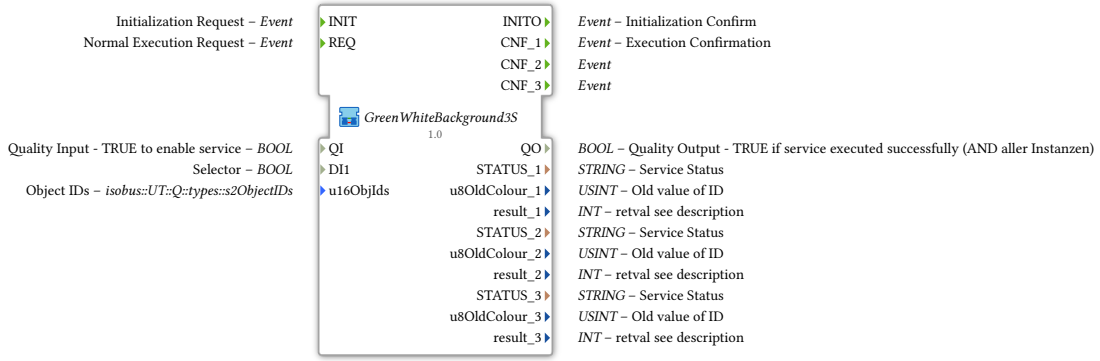
• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground3S` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s2ObjectIDs` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_3.INITO` and the `CNF` events of all three instances (`Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`) and thereby triggers `AND_QO.REQ`.

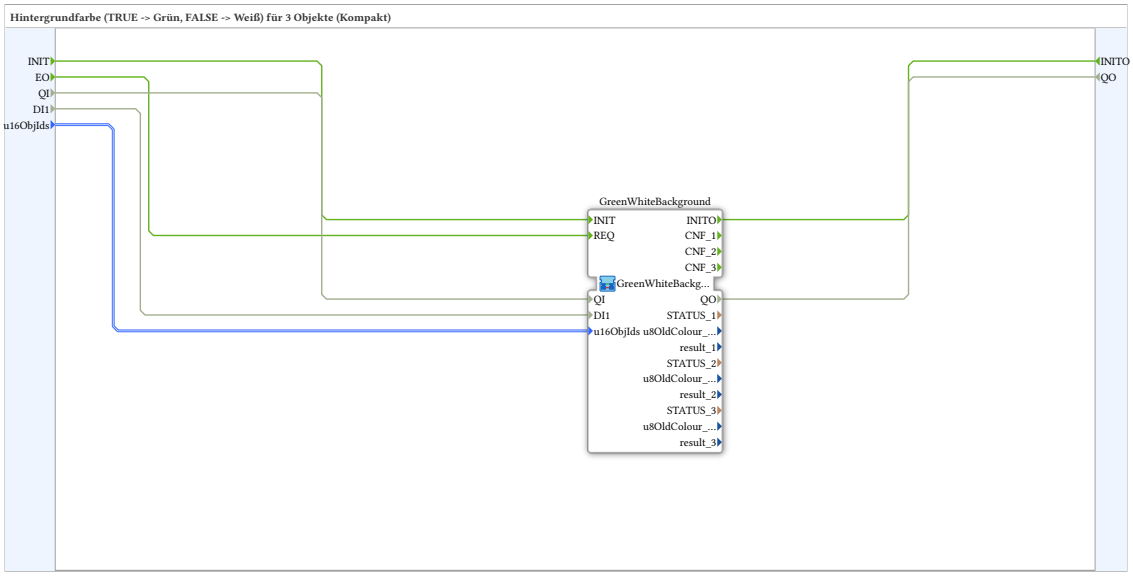
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 3 objects, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground3SC





• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground3SC` is a **compact wrapper** around `GreenWhiteBackground3S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `GreenWhiteBackground`: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

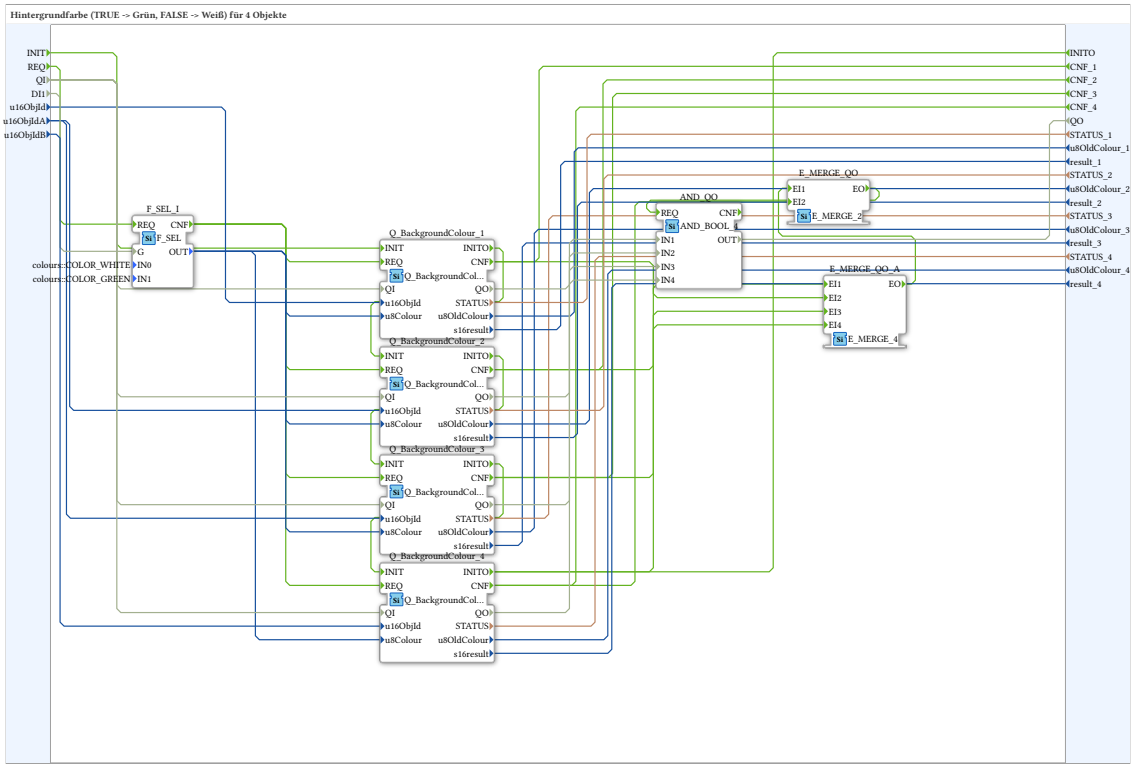
Summary

A pure pass-through to `GreenWhiteBackground3S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground4



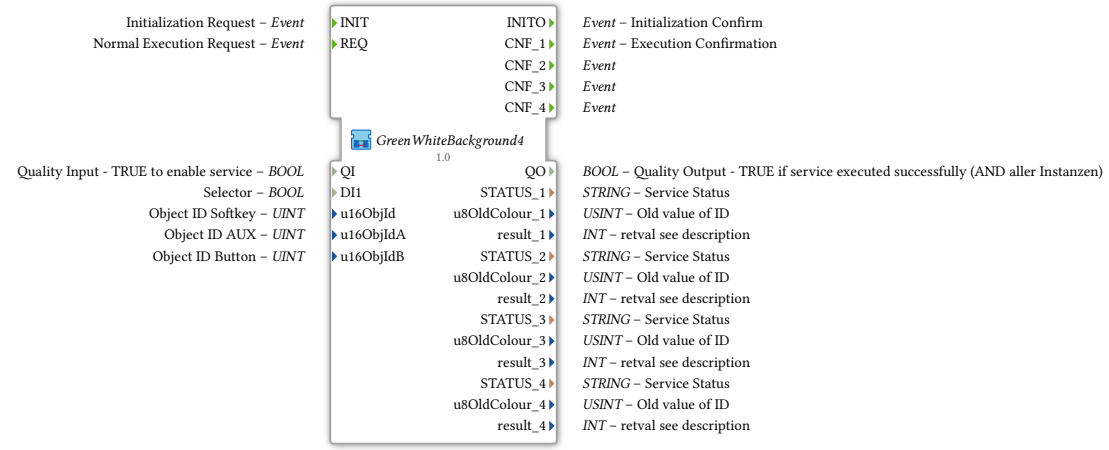
• — ○ • — ○ • — ○ •

Introduction

`GreenWhiteBackground4` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA, u16ObjIdB`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI`, `Q_BackgroundColour_3.QI` and `Q_BackgroundColour_4.QI`
- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_4`) of the `QO` outputs of `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO`, `Q_BackgroundColour_3.QO` and `Q_BackgroundColour_4.QO`
- **Initialization chain**: `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`, `Q_BackgroundColour_4.INITO` → `INITO`
- `E_MERGE_QO_A` (type `iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_4.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` and `Q_BackgroundColour_3.CNF` and passes the result to `E_MERGE_QO.EI1`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_2`) merges `E_MERGE_QO_A.EO` and `Q_BackgroundColour_4.CNF` and triggers `AND_QO.REQ`

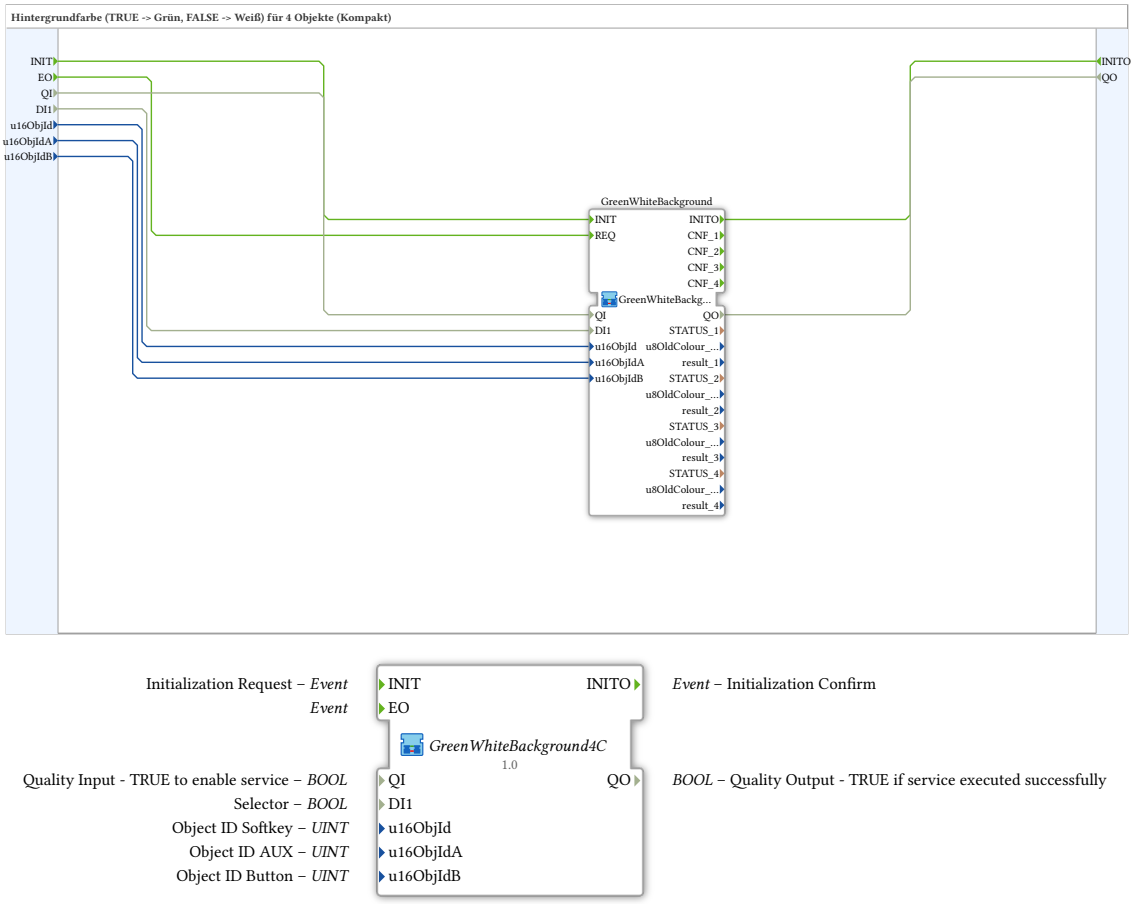
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 4 objects, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground4C



• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground4C` is a **compact wrapper** around `GreenWhiteBackground4` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI` (initial value `TRUE`) and `GreenWhiteBackground.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

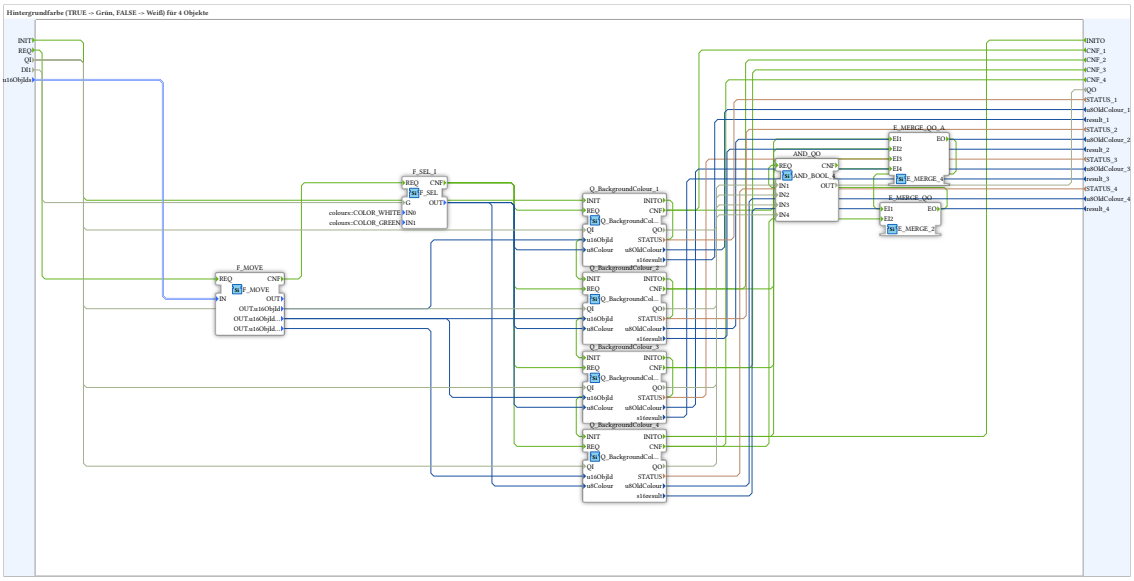
Summary

A pure pass-through to `GreenWhiteBackground4` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground4S



• – ○ • – ○ • – ○ •

Introduction

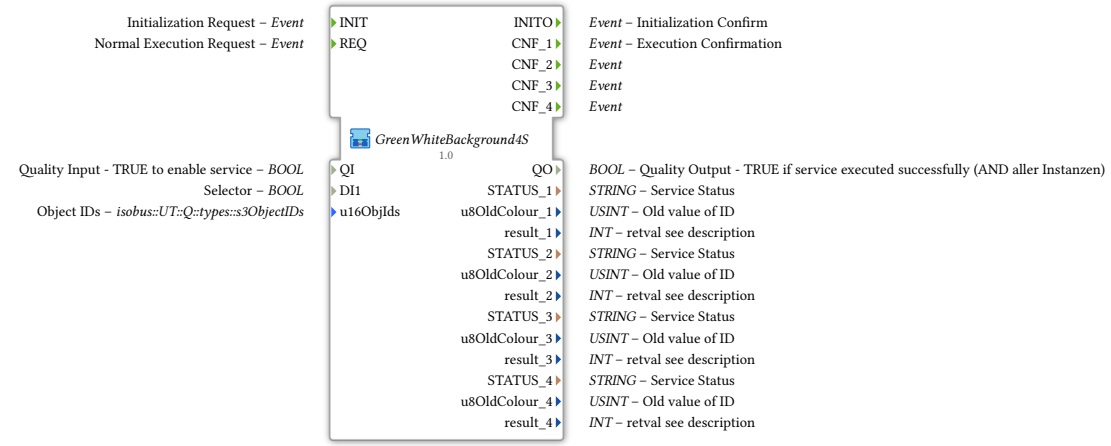
`GreenWhiteBackground4S` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Grün**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (regular object)

Initialization and quality: In addition to the previous events, the block has the event input `INIT`, the event output `INITO`, as well as `QI` (BOOL, input, initial value `TRUE`) and `QO` (BOOL, output, AND of all instances). Init chain: `INIT` ⟶ `Q_BackgroundColour_1` ⟶ `Q_BackgroundColour_2` ⟶ `Q_BackgroundColour_3` ⟶ `Q_BackgroundColour_4` ⟶ `INITO` (each `INITO` ⟶ next `INIT`). `QI` is passed on to all `Q_BackgroundColour_n.QI`. `QO` is formed by `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`, `AND_QO.OUT` ⟶ `QO`. `E_MERGE_QO_A` (`E_MERGE_4`) merges

`Q_BackgroundColour_4.INITO` and the `CNF` outputs of `Q_BackgroundColour_1`, `Q_BackgroundColour_2` and `Q_BackgroundColour_3`, `E_MERGE_QO` (`E_MERGE_2`) adds `Q_BackgroundColour_4.CNF`; `E_MERGE_QO.EO` triggers `AND_QO.REQ`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



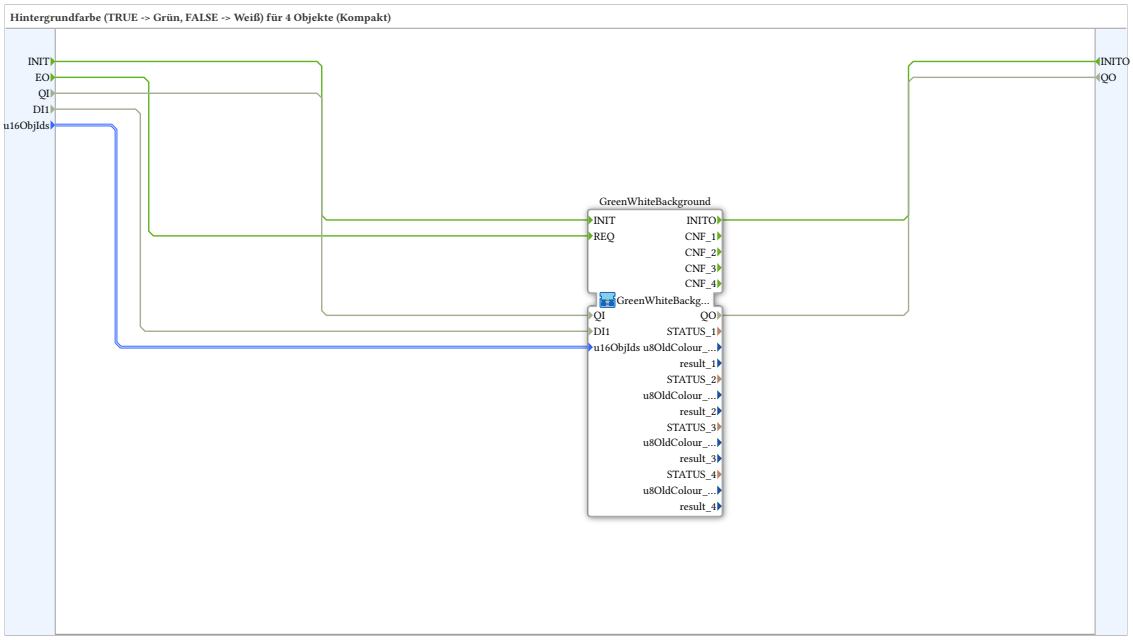
Summary

One of many variants in the background color block family: color pair Grün/Weiß, 4 objects, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

GreenWhiteBackground4SC





• – ○ • – ○ • – ○ •

Introduction

`GreenWhiteBackground4SC` is a **compact wrapper** around `GreenWhiteBackground4S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `GreenWhiteBackground`: `INIT` ⟶ `GreenWhiteBackground.INIT`, `GreenWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `GreenWhiteBackground.QI`, `GreenWhiteBackground.QO` ⟶ `QO`.

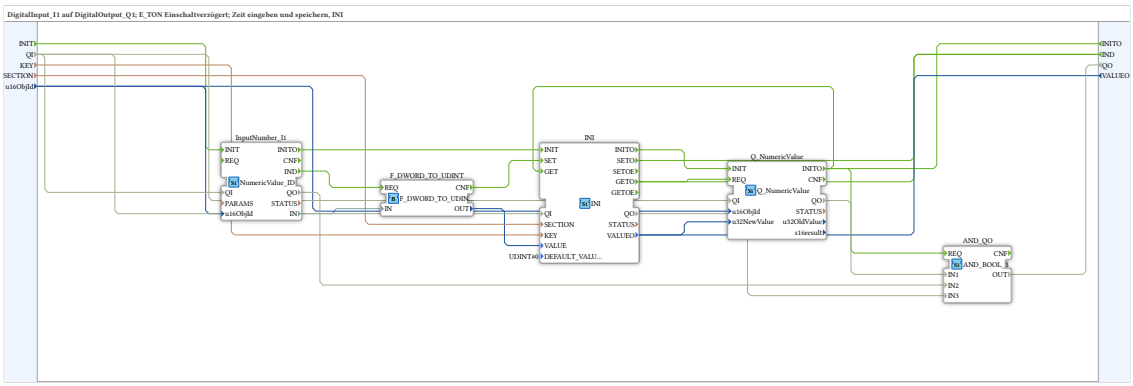
Summary

A pure pass-through to `GreenWhiteBackground4S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_IN_AND_STORE_UDINT



• – ○ • – ○ • – ○ •

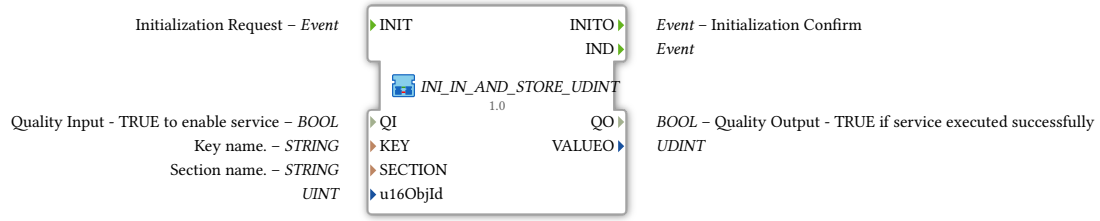
Introduction

`INI_IN_AND_STORE_UDINT` persistently stores a UDINT value entered via the VT in an INI file and automatically reloads it on deployment. This is the MyLib_B variant of the INI storage family known from MyLib_AX.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).

Quality ports: The block additionally has the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `QI` is connected to `Q_NumericValue.QI`, `InputNumber_I1.QI` and `INI.QI`; `QO` is the AND (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_3`) of `Q_NumericValue.QO`, `InputNumber_I1.QO` and `INI.QO`.

In addition, the block has the events `INIT` (input, `EInit`) and `INITO` (output, `EInit`); initialization is chained: `INIT` ⟶ `InputNumber_I1.INIT`; `InputNumber_I1.INITO` ⟶ `INI.INIT`; `INI.INITO` ⟶ `Q_NumericValue.INIT`; `Q_NumericValue.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`).



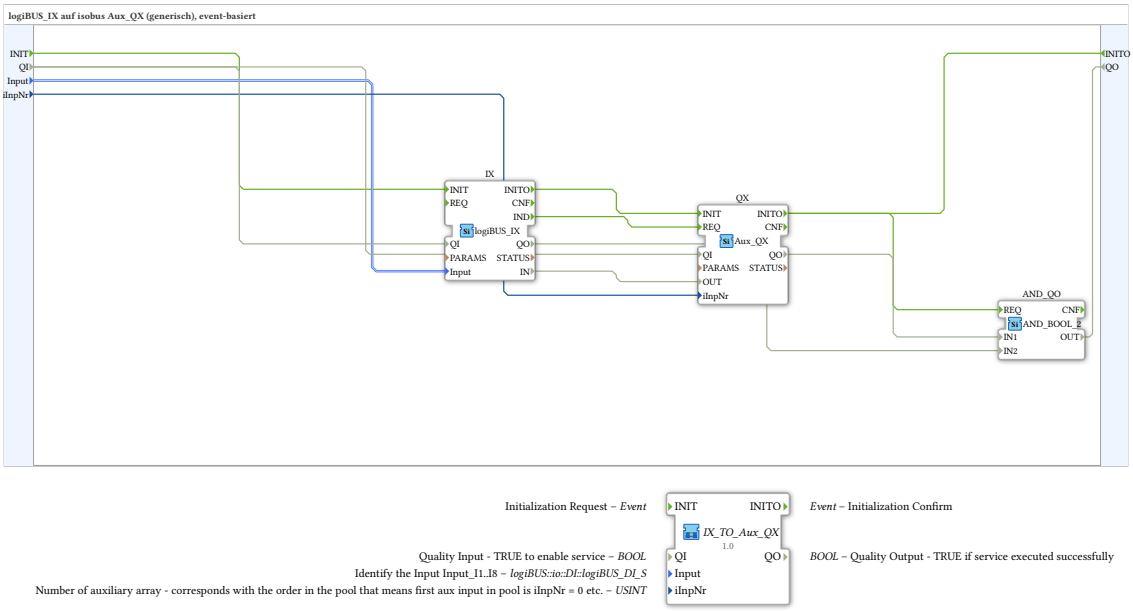
Summary

UDINT variant of the INI storage family (MyLib_B).

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

IX_TO_Aux_QX



• — ○ • — ○ • — ○ •

Introduction

The IX_TO_Aux_QX is an event-based subapplication designed to map digital inputs from a logiBUS (IX) to an isobus auxiliary output (Aux_QX). It provides a generic interface for converting logiBUS digital input signals into isobus auxiliary output requests, suitable for scenarios where auxiliary functions need to be controlled based on input signals. This subapplication encapsulates the logic to bridge the two different industrial communication systems, simplifying integration and promoting reusability.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; passed on to `IX.INIT`.

Event Outputs

- **INITO** (`EInit`): initialization confirm; comes from `QX.INITO`.

Data Inputs

- **Input** (Type: `logiBUS::io::DI::logiBUS_DI_S`): Identifies the logiBUS input signal (e.g., Input_I1 to Input_I8). The initial value is set to `Invalid`.
- **QI** (Type: `BOOL`): Quality input - TRUE enables the service. The initial value is `TRUE`.
- **iInpNr** (Type: `USINT`): Specifies the auxiliary array number, corresponding to the order in the pool. The first auxiliary input has an `iInpNr` of 0. The initial value is 0.

Data Outputs

- **QO** (Type: `BOOL`): Quality output - AND combination (`AND_QO`, `AND_BOOL_2`) of `QX.QO` and `IX.QO`. The processed data itself is used internally.

Adapters

- **None**: No adapters are defined in the subapplication interface.

Functionality

The IX_TO_Aux_QX subapplication encapsulates two primary function blocks: `logiBUS_IX` (IX) and `isobus::UT::io::Auxiliary::OUT::Aux_QX` (QX). It operates as follows:

- When the `logiBUS_IX` block receives a valid input signal, it triggers its `IND` event output.
- The `IND` event is connected to the `REQ` event input of the `Aux_QX` block, initiating a request to the isobus auxiliary output.
- Data mapping is performed via internal data connections:
  - The `Input` data variable is forwarded to `IX.Input`.
  - The `iInpNr` data variable is forwarded to `QX.iInpNr`.
  - The `IX.IN` output is internally connected to `QX.OUT`, although this does not expose a direct output from the subapplication.

Initialization and quality: `INIT` ⟶ `IX.INIT`; `IX.INITO` ⟶ `QX.INIT`; `QX.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`. `QI` is distributed to `IX.QI` and `QX.QI` (there is no hard-wired `QI` parameter any more); `QX.QO` ⟶ `AND_QO.IN1`, `IX.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`.

This structure allows the subapplication to process a logiBUS digital input and trigger an isobus auxiliary output request based on the configured input number and signal.

Technical Features

- **Event-driven Processing**: The subapplication uses an event-based mechanism to connect the input event from `IX` to the output request event of `QX`, ensuring timely response to input changes.
- **Parameterized Configuration**: The `Input` variable allows selection of specific logiBUS inputs (I1–I8), while `iInpNr` defines the order of the auxiliary array, enabling flexible setup.
- **Encapsulation**: By wrapping the function blocks, the subapplication provides a clean interface and hides the complexity of direct block wiring.
- **Reusability**: Designed as a generic solution, it can be reused across different projects requiring logiBUS-to-isobus bridging.
- **Internal Data Flow**: The data connection from `IX.IN` to `QX.OUT` indicates that the processed input data is passed internally, though not exposed externally.

State Overview

This subapplication does not implement a complex state machine. Instead, it relies on the behavior of its internal function blocks. The state is primarily event-driven, where the activation of the `IND` event in `IX` directly influences the `REQ` event in `QX`. The subapplication is stable in idle state until an input event occurs, at which point it processes the data and generates a corresponding output request.

Application Scenarios

- **Agricultural Machinery**: Used to convert physical digital inputs (e.g., switches on a control panel) into isobus auxiliary control commands, such as activating hydraulic functions or implement operations.
- **Industrial Automation**: Bridges logiBUS-based sensor inputs with isobus-controlled auxiliary devices in automated systems.
- **System Integration**: Provides a standardized interface for connecting logiBUS and isobus networks, reducing development effort and minimizing wiring errors.

Comparison with Similar Blocks

Compared to directly implementing `logiBUS_IX` and `Aux_QX` individually, the IX_TO_Aux_QX subapplication offers several advantages:

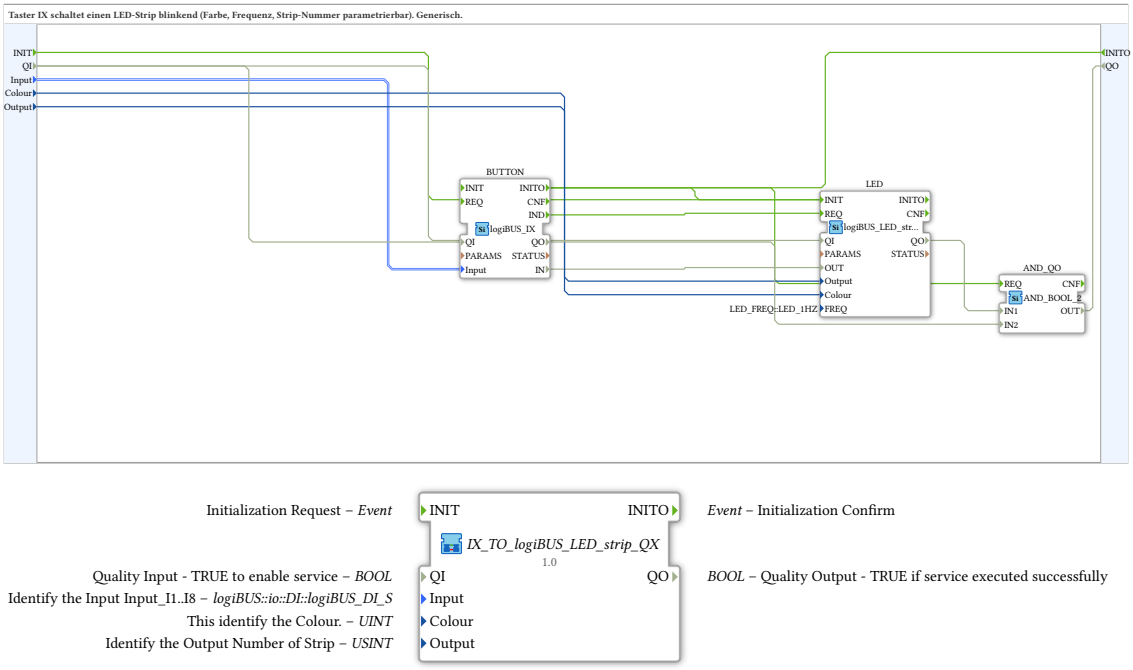
- **Simplified Interface**: Exposes only two data inputs, making it easier to configure than separately wiring multiple blocks.
- **Generic Design**: The subapplication is parameterized to support different input numbers and array indices, whereas manual wiring may require reconfiguration for each instance.
- **Reusability**: As a self-contained unit, it can be copied and integrated into different projects without additional modifications.
- **Maintainability**: Encapsulation improves code organization and simplifies debugging, as the internal logic is isolated.

Similar blocks might include direct block combinations or other bridging subapplications, but IX_TO_Aux_QX is specifically optimized for logiBUS to isobus auxiliary output conversion, offering a balance of flexibility and simplicity.

Conclusion

The IX_TO_Aux_QX subapplication is a valuable component for integrating logiBUS digital inputs with isobus auxiliary outputs. Its event-driven design, parameterized configuration, and encapsulated structure make it a practical solution for systems requiring efficient and reliable signal bridging. By simplifying the interface and promoting reusability, it enhances development productivity and system performance.

IX_TO_logiBUS_LED_strip_QX



Introduction

The `IX_TO_logiBUS_LED_strip_QX` is a generic subapplication that connects a logiBUS pushbutton input to a blinking LED strip output. It is designed for applications where a physical button should switch an LED strip on and off in a blinking mode. The color, the used input, and the target strip can be parameterized through the subapplication interface; initialization is exposed via `INIT`/`INITO` and the status via `QI`/`QO`.

The subapplication encapsulates the event handling and data routing, so it can be reused easily in larger 4diac-ide projects.

Interface Structure

Event Inputs

- `INIT` (`EInit`): Initialization Request

Apart from `INIT`, the subapplication does not expose any event inputs. The trigger is generated internally by the encapsulated pushbutton function block.

Event Outputs

- `INITO` (`EInit`): Initialization Confirm

Data Inputs

Name	Type	Initial Value	Description
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE enables the service.
`Input`	`logiBUS::io::DI::logiBUS_DI_S`	`logiBUS::io::DI::logiBUS_DI::Invalid`	Identifies the physical input, e.g. `Input_I1` to `Input_I8`.

Colour	UINT	LED_COLOURS::LED_GREEN	Identifies the color of the LED strip.
Output	USINT	—	Identifies the output number of the LED strip.

Data Outputs

- QO (BOOL): Quality Output - TRUE if the service executed successfully (AND of LED.QO and BUTTON.QO, via the AND_QO block of type iec61131::booleanOperators::AND_BOOL_2).

Adapters

There are no adapters.

Functionality

The subapplication contains two internal function blocks:

- BUTTON of type logiBUS::io::DI::logiBUS_IX
- LED of type logiBUS::io::DO_LED::logiBUS_LED_strip_QX

The BUTTON block monitors the selected physical input. When this input detects a signal change, the event output IND triggers the event input REQ of the LED block.

The data flow is as follows:

- The subapplication input Input selects the physical input for BUTTON.Input.
- The button value BUTTON.IN is forwarded to LED.OUT.
- The subapplication input Output selects the target LED strip via LED.Output.
- The subapplication input Colour sets the color via LED.Colour.

Initialization runs as a chain INIT → LED.INIT, LED.INITO → BUTTON.INIT, BUTTON.INITO → INITO (and additionally → AND_QO.REQ). QI is forwarded to LED.QI and BUTTON.QI; LED.QO and BUTTON.QO are ANDed into QO.

The internal LED block is configured with FREQ = LED_FREQ::LED_1HZ, so the selected LED strip blinks at a frequency of 1 Hz when triggered.

Technical Features

- Generic subapplication for button-controlled blinking LED strips.
- Input selection via logiBUS_DI_S, with Invalid as a safe default.
- Color selection through UINT constants from LED_COLOURS.
- Output selection via USINT.
- Internal event connection: BUTTON.IND -> LED.REQ.
- Internal data forwarding is hidden in the subapplication network.
- Both internal blocks receive QI from the subapplication's QI input (default TRUE).
- Fixed blink frequency of 1 Hz.
- No external event wiring required apart from the optional INIT/INITO initialization chain.

State Overview

The subapplication itself does not define an explicit state machine. Its behavior is determined by the encapsulated function blocks and the event connection between them.

At the subapplication level, the control flow can be described conceptually:

- **Idle state:** The button input is monitored; no LED request is active.
- **Trigger state:** A button event occurs at BUTTON.IND and triggers LED.REQ.
- **Return state:** After the request is processed, the subapplication returns to the idle state.

This simple event-driven behavior makes the subapplication easy to understand and integrate.

Application Scenarios

Typical use cases include:

- Controlling blinking LED strips with standard pushbutton inputs.
- Reusing the same logic for multiple inputs and outputs by parameterizing Input and Output.
- Setting different LED colors by using constants from LED_COLOURS.

- Building modular logiBUS-based lighting control applications in 4diac-ide.
- Encapsulating frequently used button-to-LED logic in a clean, reusable subapplication.

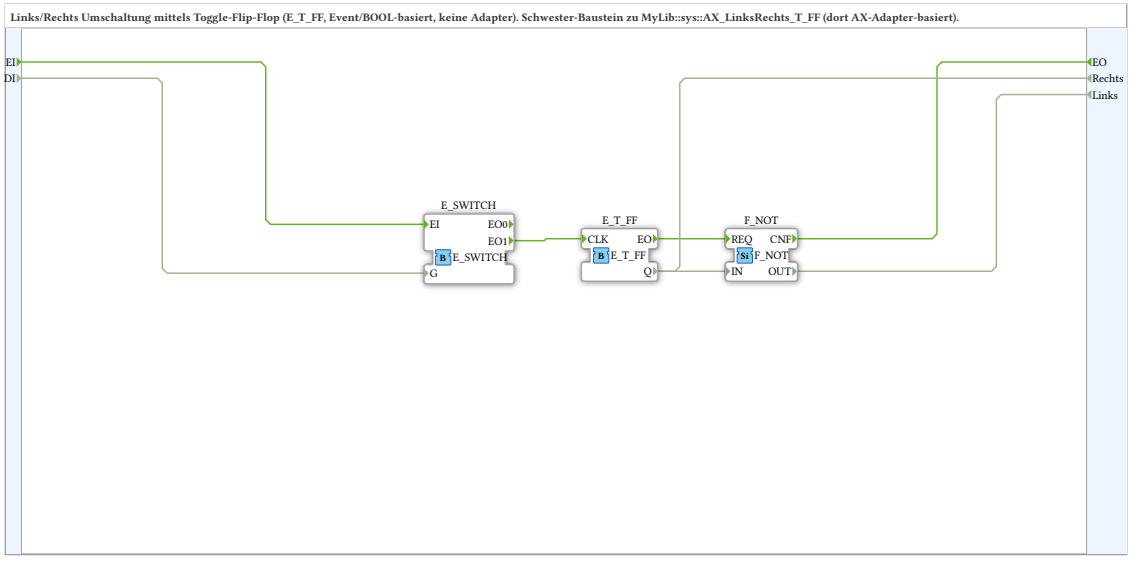
Comparison with Similar Blocks

Approach	Characteristics
Direct wiring of `logiBUS_IX` to `logiBUS_LED_strip_QX`	More wiring effort, manual event connection, no encapsulation.
Subapplication with exposed event inputs	Allows external triggering, but requires additional event handling.
Subapplication with adjustable frequency input	Offers configurable blink frequency, but has a larger interface.
`IX_TO_logiBUS_LED_strip_QX`	Self-contained button-triggered blinking LED strip with fixed 1 Hz frequency and configurable input, color, and output.

Conclusion

The `IX_TO_logiBUS_LED_strip_QX` subapplication provides a compact, generic solution for button-controlled blinking LED strips. It hides the internal event and data wiring, exposes only the necessary parameters, and can be reused in multiple places within a logiBUS-based automation project. It is especially useful when a simple pushbutton should reliably trigger a configurable LED strip output.

LinksRechts_T_FF_Event





Introduction

`LinksRechts_T_FF_Event` is a reusable IEC 61499 subapplication that implements a left/right toggle switch using an event-triggered T flip-flop. It provides two complementary boolean outputs, `Rechts` and `Links`, and can be used wherever a single event should alternately switch between two states.

The subapplication is built from standard function blocks: E_SWITCH, E_T_FF, and F_NOT. It uses only event and BOOL data connections and does not require any adapter interfaces. This makes it easy to integrate into plain event/BOOL-based automation networks.

Interface Structure

The subapplication has one event input, one event output, one BOOL data input, and two BOOL data outputs.

Event Inputs

Name	Type	Description
EI	Event	Master trigger event. If DI is TRUE, this event toggles the internal flip-flop. If DI is FALSE, the event is ignored.

Event Outputs

Name	Type	Description
EO	Event	Completion event. It is issued after the toggle operation has finished and both Rechts and Links have been updated.

Data Inputs

Name	Type	Description
DI	BOOL	Enable signal for the toggle. TRUE allows the next EI event to toggle the state. FALSE suppresses the toggle.

Data Outputs

Name	Type	Description
Rechts	BOOL	Direct output of the internal T flip-flop. It is TRUE when the toggle state is 1.
Links	BOOL	Inverted output of the internal T flip-flop. It is always the complement of Rechts.

Adapters

Name	Type	Description
None	-	This subapplication has no adapter sockets or plugs.

Functionality

The subapplication works as a gated toggle switch:

1. An event arrives at EI.
2. The internal E_SWITCH checks the DI signal:
   - If DI = TRUE, the event is passed to E_T_FF.CLK.
   - If DI = FALSE, the event is passed to the unconnected EO0 output of E_SWITCH and has no effect.
3. When E_T_FF receives the clock event, it toggles its internal state Q.
4. The new Q value is written directly to Rechts.
5. E_T_FF.EO triggers F_NOT.REQ, and F_NOT computes the inverted value of Q.
6. The inverted value is written to Links.
7. F_NOT.CNF issues the external event EO.

This event chain ensures that Rechts and Links are always complementary and are updated together during the same event handling cycle.

Technical Features

- Built entirely from standard 4diac/IEC 61499 function blocks: E_SWITCH, E_T_FF, and F_NOT.
- Pure event and BOOL-based data flow.

- No adapter interfaces required.
- Single event input for triggering and a single completion event output.
- Gating input `DI` enables or disables switching.
- Complementary outputs are generated automatically by the `F_NOT` block.
- Deterministic behavior: only one active event can cause a state change at a time.
- Suitable as a reusable subapplication in higher-level function block networks.

State Overview

The subapplication has no explicit ECC state machine, but it contains an internal memory state represented by the T flip-flop output `Q`.

Internal State `Q`	`Rechts`	`Links`
`FALSE`	`FALSE`	`TRUE`
`TRUE`	`TRUE`	`FALSE`

An active event, i.e. `EI` with `DI = TRUE`, toggles `Q` to the opposite state. The subapplication therefore alternates between the `Rechts` active state and the `Links` active state on each accepted trigger event.

If `DI = FALSE`, the state remains unchanged and no external completion event is generated.

Application Scenarios

- Direction control for motors, conveyors, or actuators where `Rechts` and `Links` represent clockwise/counterclockwise rotation.
- Single-button toggle with an additional enable signal, e.g. manual mode switching.
- Alternating between two operating modes using a single event source.
- Reusable subapplication for machine controls that require two mutually exclusive output signals.

Comparison with Similar Blocks

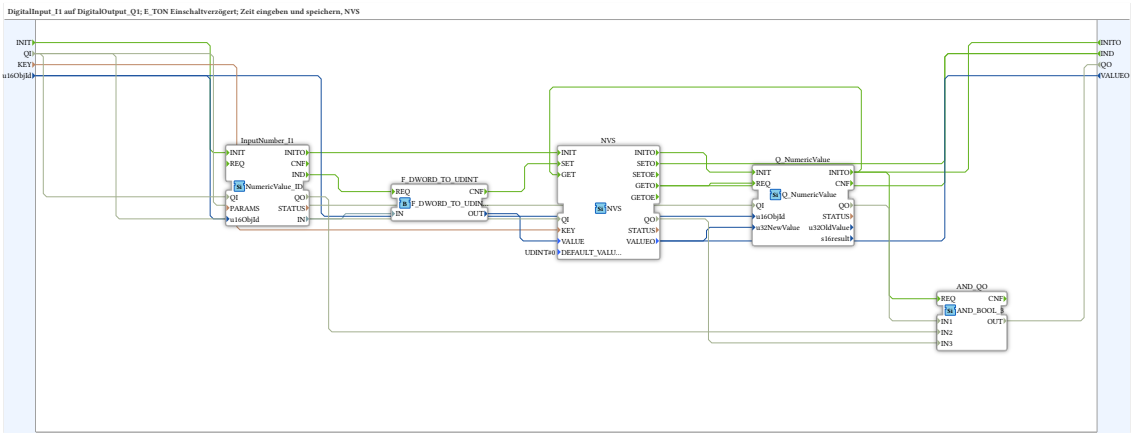
Block	Interface Type	Behavior
`LinksRechts_T_FF_Event`	Event/BOOL	Gated toggle with complementary `Rechts` and `Links` outputs.
`AX_LinksRechts_T_FF`	Adapter-based	Same left/right toggle concept, but uses an adapter interface.
Plain `E_T_FF`	Event/BOOL	Simple toggle flip-flop only; no input gating and no complement output.

Compared to a direct `E_T_FF`, this subapplication adds an enable path and complementary output generation. Compared to the adapter-based sister block, it uses a simpler interface and is therefore easier to embed in applications that do not need adapter abstraction.

Conclusion

`LinksRechts_T_FF_Event` is a compact and reliable subapplication for event-controlled left/right switching. It combines an event gate, a T flip-flop, and an inverter to produce two stable complementary outputs. Because it requires no adapters and uses only standard function blocks, it can be easily reused in a wide range of 4diac-based control applications.

NVS_IN_AND_STORE_UDINT



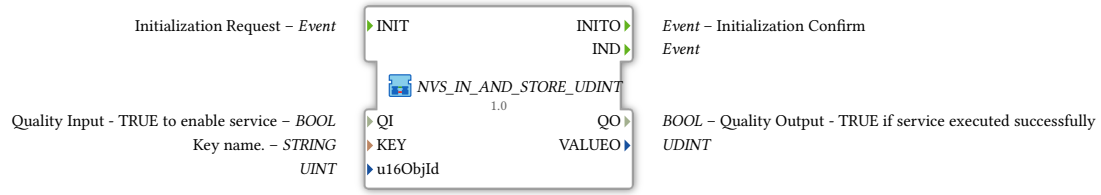
• – ○ • – ○ • – ○ •

Introduction

`NVS_IN_AND_STORE_UDINT` persistently stores a UDINT value entered via the VT in ESP32 flash (NVS) instead of an INI file — otherwise identical to `INI_IN_AND_STORE_UDINT`.

For the general pattern, see INI_IN_AND_STORE / NVS_IN_AND_STORE (shared pattern).

Quality ports: The block additionally has the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `QI` is passed through to `Q_NumericValue.QI`, `InputNumber_I1.QI` and `NVS.QI` (the internal FBs no longer carry a fixed `QI=TRUE`). `QO` is the `AND` (`AND_QO`, `AND_BOOL_3`) of the `QO` outputs of `Q_NumericValue` (`IN1`), `InputNumber_I1` (`IN2`) and `NVS` (`IN3`); `AND_QO.REQ` is triggered by `Q_NumericValue.INITO` (in addition to `Q_NumericValue.INITO` ⟶ `INITO`).



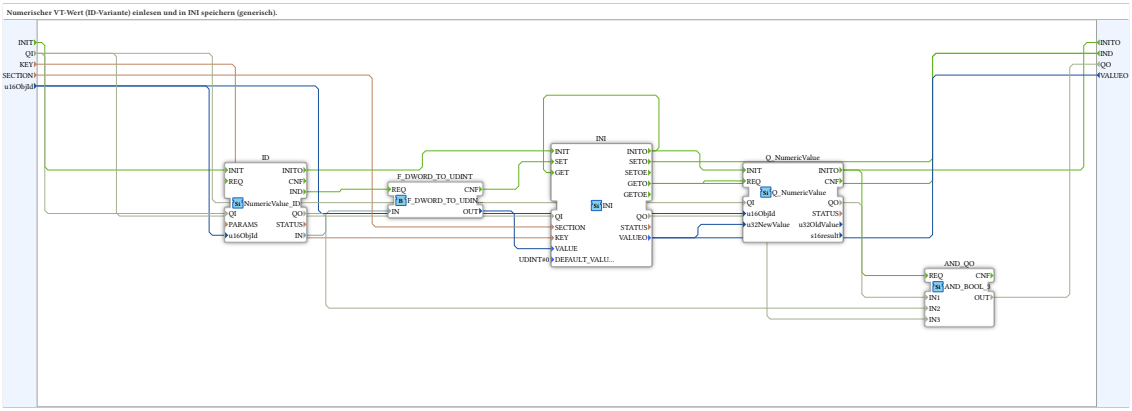
Summary

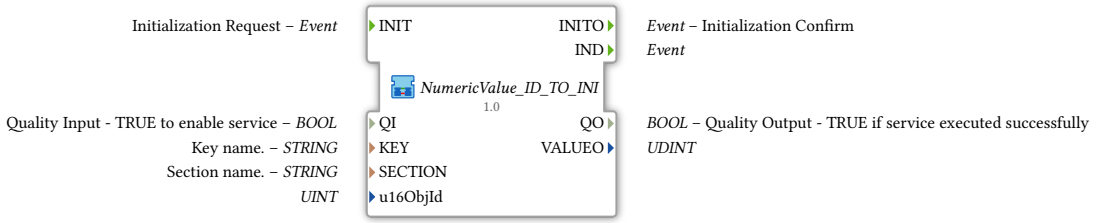
NVS counterpart to `INI_IN_AND_STORE_UDINT` (MyLib_B).

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

NumericValue_ID_TO_INI





• – ◦ • – ◦ • – ◦ •

Introduction

The **NumericValue_ID_TO_INI** subapplication is a generic, reusable component that reads a numeric value from a Virtual Terminal (VT) using an ID-based variant and stores it persistently in an INI file. It combines the ISOBUS UT (Universal Terminal) NumericValue interface with IEC 61131 conversion functions and the Eclipse 4diac INI storage service. The subapplication is designed for scenarios where a numeric VT value, identified by a unique object ID, needs to be captured and saved to a configuration or data file, while also being queryable for further processing.

Interface Structure

Event Inputs

Name	Type	Description
`INIT`	`EInit`	Initialization request; starts the init chain at `ID.INIT`. Further activity is triggered internally via the internal NumericValue FB indication.

Event Outputs

Name	Type	Description
`INITO`	`EInit`	Initialization confirmation; comes from `Q_NumericValue.INITO`, the end of the init chain.
`IND`	Event	Indicates that a read, write, or initialization operation has completed. Emitted either after a successful INI SET operation, after a GET operation, or after querying the VT value.

Data Inputs

Name	Type	Initial Value	Description
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE to enable service; passed through to `ID.QI`, `INI.QI` and `Q_NumericValue.QI`
`KEY`	`STRING`	—	The key (name) under which the numeric value is stored in the INI file.
`SECTION`	`STRING`	—	The section of the INI file where the key is located.
`u16ObjId`	`UINT`	`ID_NULL`	The object ID of the numeric VT value to be read (source identifier for the VT communication).

Data Outputs

Name	Type	Description
`QO`	`BOOL`	Quality Output - TRUE if service executed successfully; AND combination (`AND_QO`, `AND_BOOL_3`) of `Q_NumericValue.QO`, `ID.QO` and `INI.QO`
`VALUEO`	`UDINT`	The numeric value read from the VT and stored/retrieved from the INI file. This output is updated after a GET or SET operation.

Adapters

None.

Functionality

The subapp (including the added AND_QO block for the combined QO) provides a complete data path from a Virtual Terminal numeric object to persistent INI storage:

1. **VT Read**: The internal NumericValue_ID FB (isobus::UT::io::NumericValue::NumericValue_ID) reads the numeric value from the VT based on the provided u16ObjId. When the VT delivers the value, the FB raises its IND output event.
2. **Conversion**: The raw DWORD value from the VT is passed to the F_DWORD_TO_UDINT conversion FB, which converts it to a UDINT data type suitable for storage and further processing.
3. **Persistent Storage**: The converted value is written to the INI file via the INI FB (eclipse4diac::storage::INI). The parameters KEY and SECTION define the exact location in the file where the value is stored. After successful storage, the SETO event is emitted, which triggers the subapp's IND output.
4. **Initialization / Read-back**: Upon initialization, the INI FB performs a GET operation (via the internal connection from INITO to GET) to retrieve the previously stored value. After the GET completes, two actions occur:
   - The GETO event triggers the Q_NumericValue FB, which is used to query or set the VT numeric value with the retrieved stored value (ensuring synchronization).
   - The GETO event also propagates directly to the subapp's IND output, informing the external application that the read-back is complete.
5. **Output**: The retrieved value from the INI FB (VALUEO) is provided on the subapp's VALUEO output and simultaneously sent to the Q_NumericValue FB for VT synchronization.

Initialization and quality ports: The init chain is INIT → ID.INIT, ID.INITO → INI.INIT, INI.INITO → Q_NumericValue.INIT, Q_NumericValue.INITO → INITO and AND_QO.REQ (the connection INI.INITO → INI.GET remains in addition). QI (data input, BOOL, default TRUE) is passed to Q_NumericValue.QI, ID.QI and INI.QI; their QO outputs go to AND_QO.IN1 (Q_NumericValue), IN2 (ID) and IN3 (INI) of the block AND_QO (iec61131::booleanOperators::AND_BOOL_3), AND_QO.OUT → QO (BOOL). This lets the surrounding logic enable all three services and check their joint success.

Technical Features

- **Generic design**: The subapp is fully reusable and not bound to a specific application context – the KEY, SECTION, and u16ObjId parameters make it flexible for any numeric VT object.
- **DWORD → UDINT conversion**: Ensures type compatibility between the ISOBUS layer (which delivers DWORD) and the storage layer (which expects UDINT).
- **Self-initializing**: On startup, the INI FB automatically loads the last stored value via the internal INITO → GET connection.
- **Two-way synchronization**: After reading from the VT, the value is stored to INI; after reading from INI, the value is pushed to the VT via Q_NumericValue.
- **Single event output** (IND): Allows the calling application to react to any of the internal operations (SET, GET, or query) in a unified manner.
- **Direct output visibility**: The current value is always available on VALUEO, allowing external monitoring or further processing without additional event handling.
- **Use of standard libraries**: Relies on established ISOBUS UT, IEC 61131 conversion, and Eclipse 4diac storage functions, ensuring compatibility and maintainability.

State Overview

The subapp operates in a sequential, event-driven manner without explicit state variables. The internal event flow can be summarized as:

1. **Idle**: The subapp waits for an internal trigger (either a VT indication or initialization).
2. **VT Read State**: The NumericValue_ID FB receives a new value from the VT. Upon indication (IND), the value is forwarded to the conversion FB.
3. **Conversion State**: F_DWORD_TO_UDINT processes the value and issues a confirmation (CNF).

4. **Storage State**: The `INI` FB writes the converted value using `KEY`/`SECTION`. After the write completes, `SETO` is raised, and the subapp emits `IND`.
5. **Initialization / Read-back State**: On startup, the `INI` FB automatically issues a `GET`. When the value is retrieved (`GETO`), the `Q_NumericValue` FB is triggered to synchronize the VT, and `IND` is emitted simultaneously.

The use of implicit event chaining (via internal connections) ensures deterministic behavior without race conditions.

Application Scenarios

- **VT parameter persistence**: Storing operator-set numeric parameters (e.g., setpoints, machine limits) from an ISOBUS Virtual Terminal to a local INI file, so that values survive a power cycle.
- **Automated machine configuration**: Reading a numeric value from a VT object and writing it to a configuration file used by other automation components.
- **Data logging / audit trails**: Capturing numeric VT values (e.g., speed, temperature) and storing them in INI-based logs for later analysis.
- **System synchronization**: Using the INI file as a bridge between a VT and a control application – the subapp ensures both ends are updated consistently.
- **Reusable building block**: Since it was extracted from an existing application for reuse, it can be embedded in multiple FB networks without modification.

Comparison with Similar Blocks

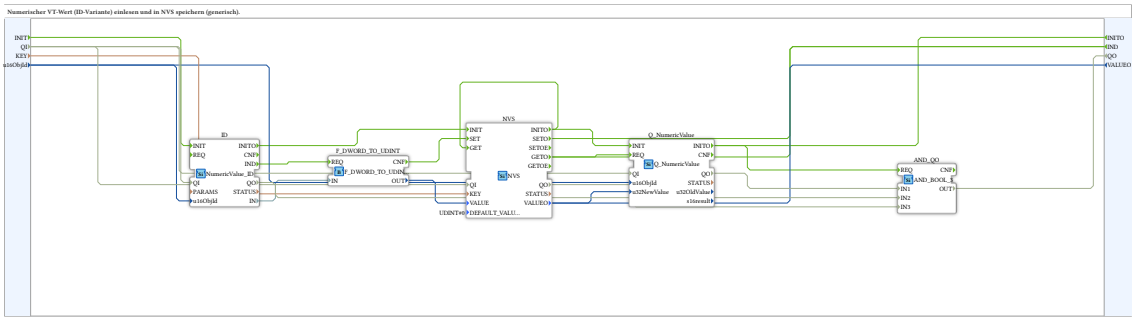
Feature	NumericValue_ID_TO_INI	Direct NumericValue FB (e.g., `NumericValue_ID`)	INI Storage FB (standalone)
Reads from VT	Yes	Yes	No
Writes to INI	Yes	No	Yes
Type conversion	Automatic (DWORD→UDINT)	Not included	Not included
Initialization load	Yes (automatic GET)	No	No
VT synchronization after read	Yes (via `Q_NumericValue`)	No	No
Encapsulation	High – single subapp	Low – requires external wiring	Low – requires external wiring
Reusability	High (generic parameters)	Medium (requires setup)	Medium (requires setup)
Complexity	Moderate (internal logic hidden)	Low but manual effort needed	Low but manual effort needed

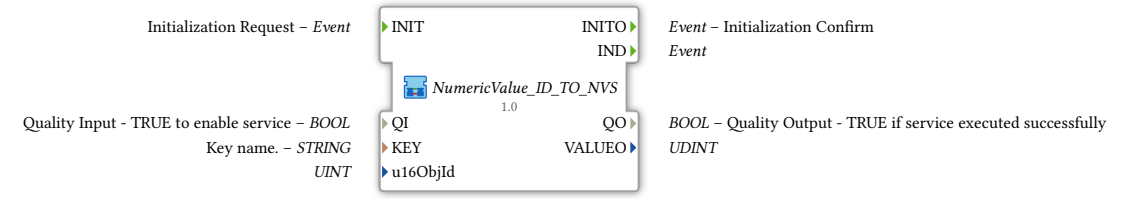
The main advantage of this subapp over using the individual blocks separately is the **complete integration** – the user does not have to wire the event/data connections, handle the initialization sequence, or manage the conversion logic. It provides a ready-to-use, self-contained solution.

Conclusion

The **NumericValue_ID_TO_INI** subapplication is a well-structured, generic component that bridges the gap between ISOBUS Virtual Terminal numeric values and persistent INI storage. It encapsulates a complete data flow – from VT reading, through type conversion, to file storage and back-synchronization – within a single, easy-to-use interface. With only three configuration inputs (plus `QI`) and the event outputs `IND` and `INITO`, it is both simple to integrate and highly reusable. The automatic initialization behavior and the implicit synchronization with the VT make it particularly suitable for applications requiring persistent, consistent numeric parameters. Its design follows established standards (IEC 61499-2, IEC 61131) and leverages proven library components, ensuring reliability and maintainability in industrial automation environments.

NumericValue_ID_TO_NVS





• – ○ • – ○ • – ○ •

Introduction

NumericValue_ID_TO_NVS is a generic IEC 61499 subapplication that reads a numeric value identified by an object ID and stores or retrieves that value in the non-volatile storage (NVS) of an ESP32. It combines an ISOBUS-compatible numeric value input block, a data type conversion function block, and an NVS storage block into a reusable component. The subapplication exposes a simple interface with one event output, two data inputs, and one data output.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request; goes to `ID.INIT` and starts the initialization chain

Event Outputs

Event	Comment
`IND`	Indicates that either a SET or GET operation on the NVS has been completed.
`INITO`	Initialization confirm; comes from `Q_NumericValue.INITO`, the end of the chain `ID` ⟶ `NVS` ⟶ `Q_NumericValue`.

Data Inputs

Name	Type	Initial Value	Comment
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE to enable service; passed through to `ID.QI`, `NVS.QI` and `Q_NumericValue.QI`
`KEY`	`STRING`	—	Key name used in the NVS storage.
`u16ObjId`	`UINT`	`ID_NULL`	Object ID of the numeric value to be processed.

Data Outputs

Name	Type	Comment
`QO`	`BOOL`	Quality Output - TRUE if service executed successfully; AND of `Q_NumericValue.QO`, `ID.QO` and `NVS.QO`
`VALUEO`	`UDINT`	Numeric value that is currently stored in the NVS.

Adapters

None.

Functionality

The subapplication internally uses three main function blocks:

1. `NumericValue_ID`
   Reads a numeric value associated with the input `u16ObjId`. Its event output `IND` triggers the following processing steps.
2. `F_DWORD_TO_UDINT`
   Converts the incoming numeric value from `DWORD` to `UDINT`. This ensures that the value is stored in the NVS in a consistent unsigned double integer format.
3. `NVS`
   Stores the converted value under the key provided by `KEY` and makes it available again on the output `VALUEO`.

The internal event flow works as follows:

- When the `NumericValue_ID` block produces an `IND` event, the conversion block is triggered via `REQ`.
- After conversion, the `CNF` event triggers the NVS block to perform a `SET` operation.
- A completed SET operation is reported on `SETO`, which is propagated to the subapplication output `IND`.
- On initialization, the NVS block performs a GET operation automatically after its `INITO` event. The retrieved value is then assigned to `VALUEO` and also passed to the internal `Q_NumericValue` block.

Initialization and quality ports: `INIT` → `ID.INIT`, `ID.INITO` → `NVS.INIT`, `NVS.INITO` → `Q_NumericValue.INIT`, `Q_NumericValue.INITO` → `INITO` (and → `AND_QO.REQ`); in addition `NVS.INITO` still triggers `NVS.GET`. `QI` (data input, BOOL, default `TRUE`) is passed on to `ID.QI`, `NVS.QI` and `Q_NumericValue.QI`. The output `QO` (BOOL) is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_3`), which combines `Q_NumericValue.QO`, `ID.QO` and `NVS.QO`. This lets the surrounding logic enable the service and check whether it succeeded.

Technical Features

- Generic NVS key handling through the `KEY` input.
- Object-ID-based value selection through `u16ObjId`.
- Automatic conversion from `DWORD` to `UDINT` before storage.
- NVS default value configured as `UDINT#0`.
- Initial object ID default is `ID_NULL`, imported from `isobus::UT::Q::const::IDs`.
- All internal event sequencing is self-contained; no external event input is required.
- Provides a simple completion signal via the `IND` event output.
- This subapplication is suitable for reuse in larger ISOBUS or ESP32-based systems.

State Overview

The subapplication can be described by the following internal operating states:

1. **Initialization State**
   After startup, the NVS block initializes. Its `INITO` event automatically triggers a GET request.
2. **GET State**
   The NVS reads the value associated with `KEY`. When completed, the retrieved value is written to `VALUEO` and the `IND` event is emitted.
3. **Value Update State**
   When a new numeric value is read from the `NumericValue_ID` block, it is converted and then stored in the NVS via a SET operation.
4. **Completion State**
   After the SET operation finishes, the `SETO` event propagates to the external world through `IND`.

Application Scenarios

- Storing ISOBUS numeric implement values in ESP32 non-volatile memory.
- Persisting calibration values or configuration parameters that are addressed by object IDs.
- Building reusable components for V3 / ISOBUS object pool applications.
- Replacing direct manual handling of NVS storage with a generic, event-driven storage pattern.

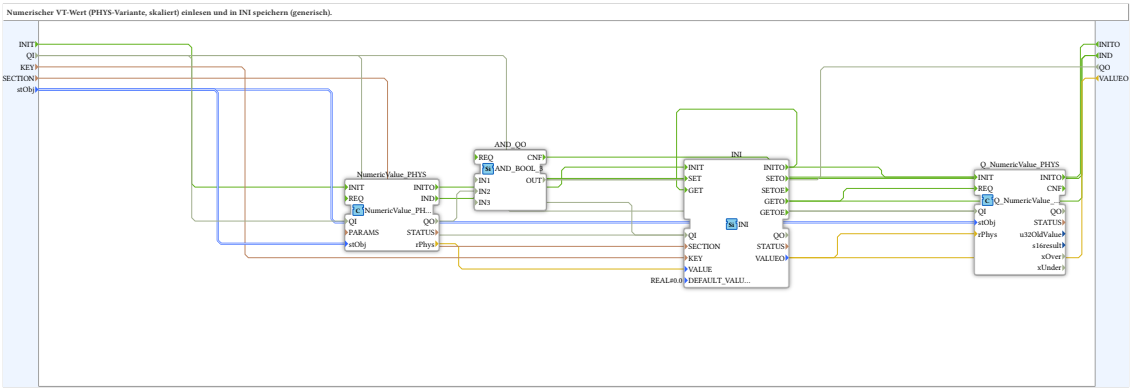
Comparison with Similar Blocks

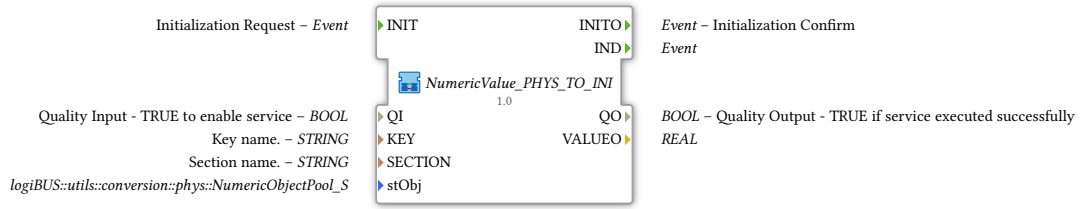
- Compared to a straightforward connection between `NumericValue_ID` and `NVS`, this subapplication encapsulates the required data conversion and event handling, reducing wiring complexity.
- Compared to non-ID numeric storage blocks, this variant selects values by `u16ObjId`, making it suitable for ISOBUS-related data.
- Compared to blocks that only read or only write, this subapplication supports both reading and writing while exposing a single completion event.

Conclusion

NumericValue_ID_TO_NVS is a compact and reusable subapplication for reading numeric values by object ID and persisting them in ESP32 NVS storage. It simplifies integration by hiding conversion, initialization, and event sequencing behind a small interface. It is especially useful in ISOBUS-based systems where numeric values need to be preserved across power cycles.

NumericValue_PHYS_TO_INI





• — ○ • — ○ • — ○ •

Introduction

The `NumericValue_PHYS_TO_INI` subapplication provides a generic, reusable mechanism for reading a scaled numeric process or device value (the PHYS variant) and persistently storing it in an INI storage. It combines a physical-value acquisition block, an INI storage block, and a query-conversion block to deliver a complete read-and-persist chain. The entity is designed to be parameterized via a key, a section name, and a numeric object-pool descriptor that carries scaling, offset, and decimal-place information.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialization request. The further internal processing is triggered by the embedded function blocks.

Event Outputs

Name	Type	Description
`INITO`	EInit	Initialization confirmation (from `Q_NumericValue_PHYS.INITO`).
`IND`	Event	Indicates that a value has been successfully stored in or retrieved from the INI storage.

Data Inputs

Name	Type	Description
QI	BOOL	Quality Input - TRUE to enable service; distributed to Q_NumericValue_PHYS.QI, NumericValue_PHYS.QI and INI.QI
KEY	STRING	Key name under which the value is stored in the INI file.
SECTION	STRING	Section name within the INI file used for grouping the key.
stObj	I4CBUS::utils::conversion::phys::NumericObjectPool_t	Object-pool descriptor containing the numeric object ID, scale factor, offset, and number of decimals. Initial value: (u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0).

Data Outputs

Name	Type	Description
QO	BOOL	Quality Output - TRUE if service executed successfully; AND of Q_NumericValue_PHYS.QO, NumericValue_PHYS.QO and INI.QO
VALUEO	REAL	The current value that has been stored or retrieved from the INI storage.

Adapters

None.

Functionality

The subapplication orchestrates four internal function blocks:

- NumericValue_PHYS – acquires a raw process value and converts it into a scaled physical REAL value using the parameters contained in stObj.
- INI – stores and retrieves that REAL value in an INI file, addressed by KEY and SECTION.
- Q_NumericValue_PHYS – a query function that reconstructs the physical representation from the stored numeric value, using the same object-pool descriptor.
- AND_QO (iec61131::booleanOperators::AND_BOOL_3) – combines Q_NumericValue_PHYS.QO (IN1), NumericValue_PHYS.QO (IN2) and INI.QO (IN3) into the subapplication output QO.

The operational sequence is as follows:

1. Upon initialization, the INI block performs an automatic GET operation (INITO → GET) to load any previously persisted value for the given key and section.
2. When a new value is available, NumericValue_PHYS emits an IND event, triggering the INI.SET operation. The converted physical value (rPhys) is written to the INI storage.
3. Both the completion of a SET and the completion of a GET operation propagate an IND event to the subapplication output.
4. On every GETO, the retrieved value is forwarded to the query block Q_NumericValue_PHYS for further conversion, and simultaneously made available as VALUEO.

The internal data connections ensure that the same stObj descriptor is used consistently by both the acquisition block (for scaling while reading) and the query block (for reverse scaling), while KEY and SECTION only reference the INI storage.

Quality ports: QI (data input, BOOL, default TRUE) is distributed to Q_NumericValue_PHYS.QI, NumericValue_PHYS.QI and INI.QI; the output QO (BOOL) is the AND of the QO outputs of these three blocks (AND_QO). This lets the surrounding logic enable the service and check whether it succeeded.

Initialization: `INIT` → `NumericValue_PHYS.INIT`; `NumericValue_PHYS.INITO` → `INI.INIT`; `INI.INITO` → `Q_NumericValue_PHYS.INIT` (and internally → `INI.GET`); `Q_NumericValue_PHYS.INITO` → `INITO` and → `AND_QO.REQ`.

Technical Features

- **Generic parameterization** – The subapplication does not hard-code any key, section, or object ID. All addressing and scaling information is supplied through the data inputs.
- **Scaling support** – The `stObj` descriptor allows arbitrary `REAL` scaling (`r32Scale`, `i32Offset`) and decimal rounding (`u8Decimals`), making the block suitable for engineering-unit conversions.
- **Non-volatile persistence** – Values are stored in an INI file, surviving restarts of the runtime environment.
- **Default initial value** – If no stored value exists, the `INI` block falls back to a default value of `REAL#0.0`.
- **Automatic init-read** – On start-up the stored value is fetched automatically, so the `VALUEO` output reflects the persisted state without an explicit external trigger.

State Overview

The internal `INI` function block follows this state transition pattern:

- **Uninitialized** – After power-up or before `INIT` is accepted.
- **Initializing** – On receiving the `INI.INITO` event, the block immediately issues a `GET` operation to load the stored value.
- **Operational (Set/Get)** – In this state, the block alternates between `SET` (triggered by `NumericValue_PHYS.IND`) and `GET` operations, each producing a corresponding `SETO` or `GETO` confirmation event.
- **Fault** – If a storage access fails, the `INI` block raises an error condition; the error handling is external to this subapplication and must be monitored via the standard `QO`/`EO` signals of the INI block (not exposed at this level).

The `NumericValue_PHYS` acquisition block operates independently and is enabled via the `QI` of the subapplication; it continuously acquires and converts values as they arrive.

Application Scenarios

- **Machine parameter persistence** – Storing calibrated sensor values or scaled setpoints in an INI configuration file so they remain valid after a controller restart.
- **Data logging with scaling** – Saving scaled engineering values (e.g., temperatures, pressures, flow rates) in a compact textual format, where the original unit conversion is defined by the object-pool descriptor.
- **System configuration exchange** – Using a standardized INI section/key structure to export/import process values between different plant components or HMI layers.
- **Generic library reuse** – Because the subapplication is fully generic, it can be instantiated multiple times within a larger application, each instance configured with its own key, section, and scaling descriptor, without code changes.

Comparison with Similar Blocks

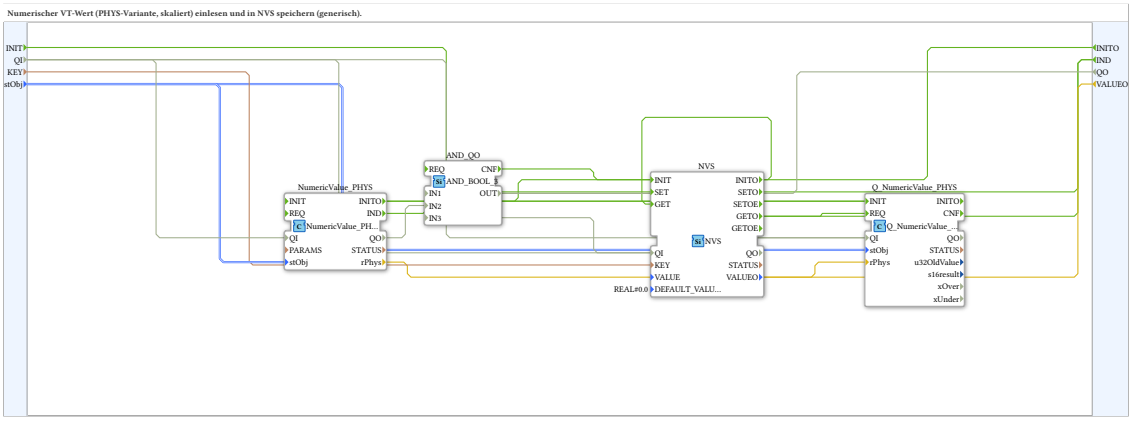
Aspect	`NumericValue_PHYS_TO_INI`	`NumericValue` (raw, no storage)	`NumericValue_TO_INI` (non-phys variant)
Value scaling	Yes – through `stObj` (scale, offset, decimals)	No – raw value only	No – raw value only
Persistence in INI	Yes	No	Yes
Automatic init-read	Yes	N/A	Yes
Required external data	`KEY`, `SECTION`, `stObj`	None (only acquisition parameters)	`KEY`, `SECTION`
Output behavior	`VALUEO` is the stored/retrieved value	Output is the live raw value	Output is the stored raw value

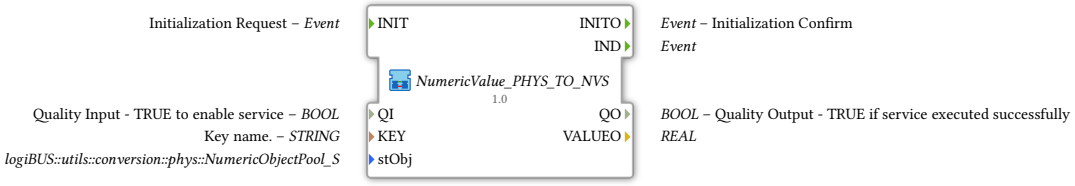
The distinctive advantage of this subapplication is the combination of scaling and persistence in a single reusable component, which reduces wiring effort and ensures consistent conversion parameters between acquisition and storage.

Conclusion

`NumericValue_PHYS_TO_INI` is a compact and generic subapplication that solves the recurring task of reading a scaled physical process value and making it persistable in an INI storage. Its well-defined interface, automatic initialization behavior, and built-in scaling support make it a suitable building block for larger automation projects that require configuration persistence with engineering-unit correctness. The block is particularly valuable in environments where multiple measured values must be archived or restored across restarts without redundant application logic.

NumericValue_PHYS_TO_NVS





• – ○ • – ○ • – ○ •

Introduction

The **NumericValue_PHYS_TO_NVS** subapplication is a generic, reusable component that integrates a physical numeric value sensor (scaled variant) with non-volatile storage (NVS). It reads a physical value using the `NumericValue_PHYS` function block, stores the result into an NVS key, and provides the stored value as an output. The subapp is designed for embedded systems where calibration data or process values need to be persisted across power cycles.

This subapp decouples the scaling logic (performed by `NumericValue_PHYS`) from the storage management (handled by `NVS`), offering a clean interface that exposes only the key name, scaling parameters, and the current stored value.

Interface Structure

Event Outputs

Event	Description
`INITO`	Initialization confirm (EInit). Comes from `NVS.INITO` (end of the initialization chain).
`IND`	Indicates that a storage or retrieval operation has been completed. This event is emitted after a successful `SET` or `GET` operation from the internal NVS block.

Event Inputs

- **INIT** (EInit): Initialization request. Passed on to `Q_NumericValue_PHYS.INIT`, which starts the chain `Q_NumericValue_PHYS` ⟶ `NumericValue_PHYS` ⟶ `NVS`.

Data Inputs

Name	Type	Description
QI	BOOL	Quality Input - TRUE to enable service; distributed to Q_NumericValue_PHYS.QI, NumericValue_PHYS.QI and NVS.QI
KEY	STRING	The unique identifier (key name) under which the value is stored in NVS.
stObj	iomBUS::utils::conversion::phys::NumericObjectPScaling	Scaling and formatting parameters (object ID, scaling factor, offset, decimals). Default: (u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0).

Data Outputs

Name	Type	Description
QO	BOOL	Quality Output - TRUE if service executed successfully; AND combination (AND_QO, AND_BOOL_3) of Q_NumericValue_PHYS.QO, NumericValue_PHYS.QO and NVS.QO
VALUEO	REAL	The current value stored in NVS (read back after initialization or after a write operation).

Adapters

None.

Functionality

The subapp operates internally with three main components and an AND block for the quality output:

1. **NumericValue_PHYS** – Converts a raw physical input into a scaled REAL value (rPhys) using the scaling parameters from stObj. It receives QI from the subapp input, so it is ready to process by default.
2. **NVS** – A non-volatile storage block that handles SET, GET, and INIT operations. It receives QI from the subapp input and has a default value of 0.0.
3. **Q_NumericValue_PHYS** – A query block that reconstructs the physical value from the stored data, making it available for further processing. It receives INIT and QI from outside; its INITO starts NumericValue_PHYS, and its QO feeds AND_QO.IN1.
4. **AND_QO** (iec61131::booleanOperators::AND_BOOL_3) – combines Q_NumericValue_PHYS.QO (IN1), NumericValue_PHYS.QO (IN2) and NVS.QO (IN3) into QO.

Data flow:

- The scaled physical value (rPhys) from NumericValue_PHYS is fed to the VALUE input of NVS.
- The event IND from NumericValue_PHYS triggers the SET operation of NVS, storing the current value under KEY.
- Upon successful SETO (or GETO), the subapp emits an IND event to inform the outside world.
- The subapp event INIT is forwarded to Q_NumericValue_PHYS.INIT; Q_NumericValue_PHYS.INITO → NumericValue_PHYS.INIT; NumericValue_PHYS.INITO → NVS.INIT; NVS.INITO is emitted as the subapp's INITO and also triggers AND_QO.REQ. QI is distributed to Q_NumericValue_PHYS.QI, NumericValue_PHYS.QI and NVS.QI, and AND_QO.OUT is output as the subapp's QO.
- Independently, after the NVS block's initialization (NVS.INITO), it performs a GET operation, retrieving the previously stored value. This value is then output via VALUEO and also passed to Q_NumericValue_PHYS for scaling/interpretation.

The subapp therefore keeps the stored value up-to-date whenever a new physical reading is available and also allows the stored value to be read back after a restart.

Quality ports: QI (data input, BOOL, default TRUE) is distributed to Q_NumericValue_PHYS.QI, NumericValue_PHYS.QI and NVS.QI; the output QO (BOOL) is the AND combination (AND_QO) of the three QO outputs. This lets the surrounding logic enable the service and check whether it succeeded.

Technical Features

- **Generic scalability** – The `stObj` parameter encapsulates all scaling and formatting information, making the subapp reusable for different sensor types or units.
- **Persistent storage** – Leverages the NVS mechanism of the ESP32 platform to retain data across power cycles.
- **Automatic initialization** – On startup, the subapp reads the stored value from NVS and exposes it via `VALUE0`.
- **Minimal external interface** – Only one key name and the scaling object are required as inputs, keeping integration simple.
- **Modular design** – Uses well-defined function blocks from the `isobus` and `logiBUS` libraries, promoting code reuse and maintainability.

State Overview

The subapp does not explicitly define a state machine. Instead, it relies on the built-in behavior of the internal NVS block, which has the following typical states:

- **Uninitialized** – Before `INIT0` is processed.
- **Idle** – Waiting for a `SET` or `GET` request.
- **Busy** – During a read/write operation.
- **Error** – If a storage operation fails (handled internally by the NVS block).

The subapp's output `IND` is emitted after each completed operation, providing a simple handshake mechanism for external logic.

Application Scenarios

- **Calibration data storage** – Store calibration constants or correction factors for analog inputs.
- **Setpoint persistence** – Save user-configured setpoints in an industrial control system.
- **Data logging** – Periodically capture a physical measurement and keep the last value in non-volatile memory.
- **Fail-safe recovery** – Restore the last known good value after a system reboot.

The subapp is particularly useful in IoT devices, smart sensors, and embedded controllers where configuration or measurement data must survive power loss.

Comparison with Similar Blocks

There are other subapplications that handle NVS storage (e.g., a raw value store without scaling or a store with a fixed scaling). The distinguishing feature of **NumericValue_PHYS_TO_NVS** is its tight integration with the physical scaling layer. Unlike a simple `NVS` wrapper, this block:

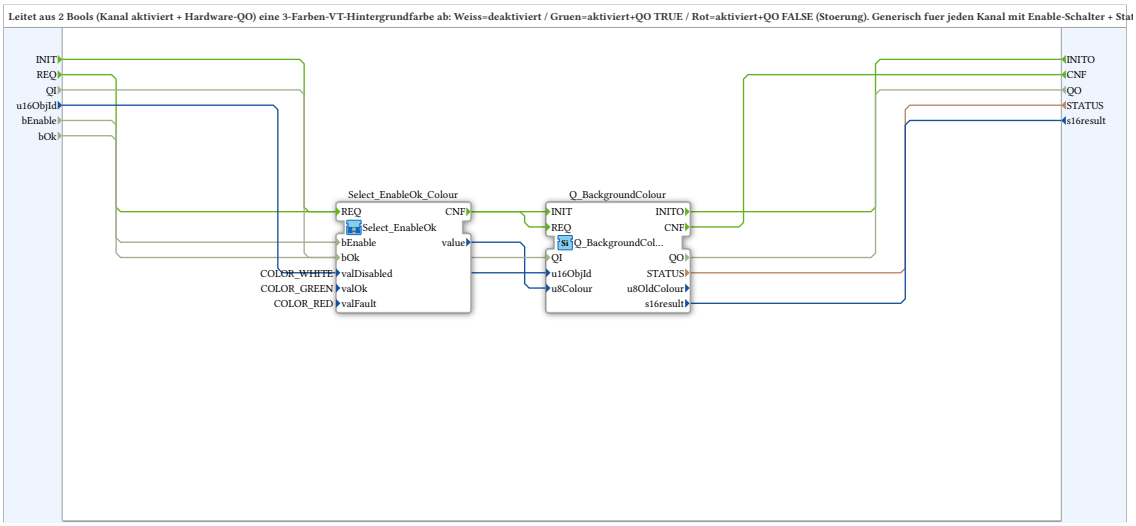
- Automatically applies scaling factors and offsets via `NumericValue_PHYS`.
- Provides a query path (`Q_NumericValue_PHYS`) to convert the stored raw data back to a physical value.
- Combines storage and conversion in a single, configurable component.

Other blocks might require the user to manage scaling separately, increasing complexity and the risk of inconsistencies.

Conclusion

The **NumericValue_PHYS_TO_NVS** subapplication offers a compact and efficient solution for persisting scaled numeric values in non-volatile storage. Its generic interface and internal modularity make it suitable for a wide range of embedded applications. By encapsulating scaling, storage, and retrieval, it simplifies the development of robust systems that require reliable data retention.

Q_BackgroundColour_EnableOk



• — ○ • — ○ • — ○ •

Introduction

`Q_BackgroundColour_EnableOk` derives a 3-color VT background color from 2 bools (channel enabled + hardware QO): white = disabled, green = enabled and QO TRUE, red = enabled and QO FALSE (fault). Generic for any channel with an enable switch + status LED (PWM, PI, ...). The pure selection logic now lives in `Select_EnableOk` this block just instantiates it with the 3 colors as parameters and attaches `Q_BackgroundColour`.

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, “Quality Input - TRUE to enable service”) and the data output `QO` (BOOL, “Quality Output - TRUE if service executed successfully”). `INIT` is passed on to `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColour.QI` and `Q_BackgroundColour.QO` to `QO`.



Function blocks used

- **Select_EnableOk_Colour** (SubApp, type `MyLib::sys::Select_EnableOk`): `valDisabled=COLOR_WHITE`, `valOk=COLOR_GREEN`, `valFault=COLOR_RED`.
- **Q_BackgroundColour** (`isobus::UT::Q::Q_BackgroundColour`): writes the computed color to `u16ObjId`.

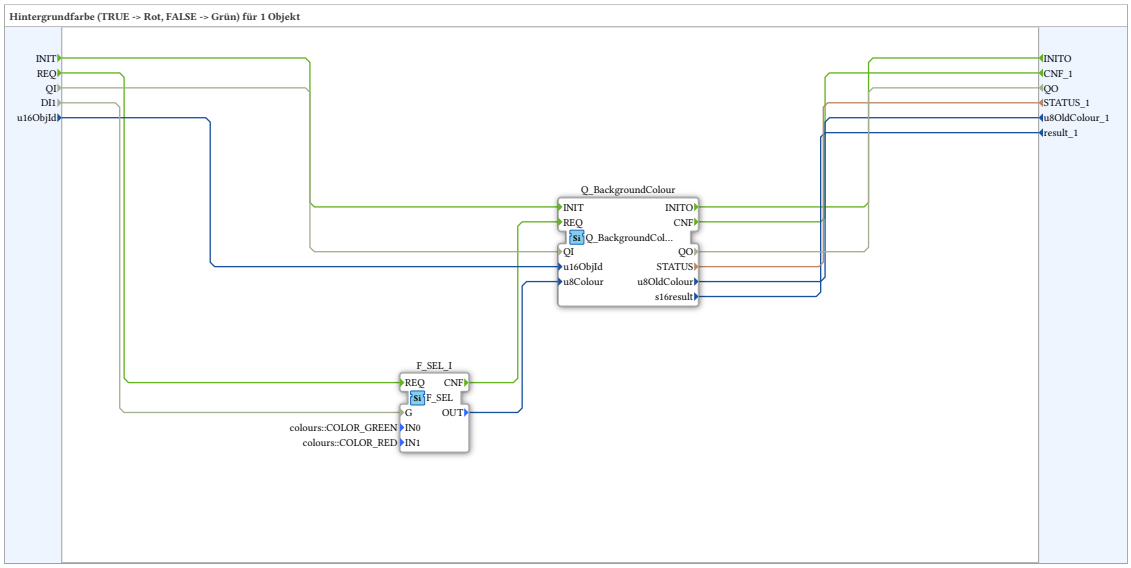
Summary

Ready-parameterized color logic (white/green/red) for enable+ok status displays, built on the generic `Select_EnableOk` selection.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1



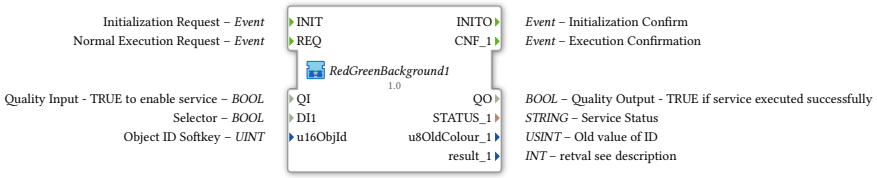
• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground1` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColour.QI` and `Q_BackgroundColour.QO` to `QO`.



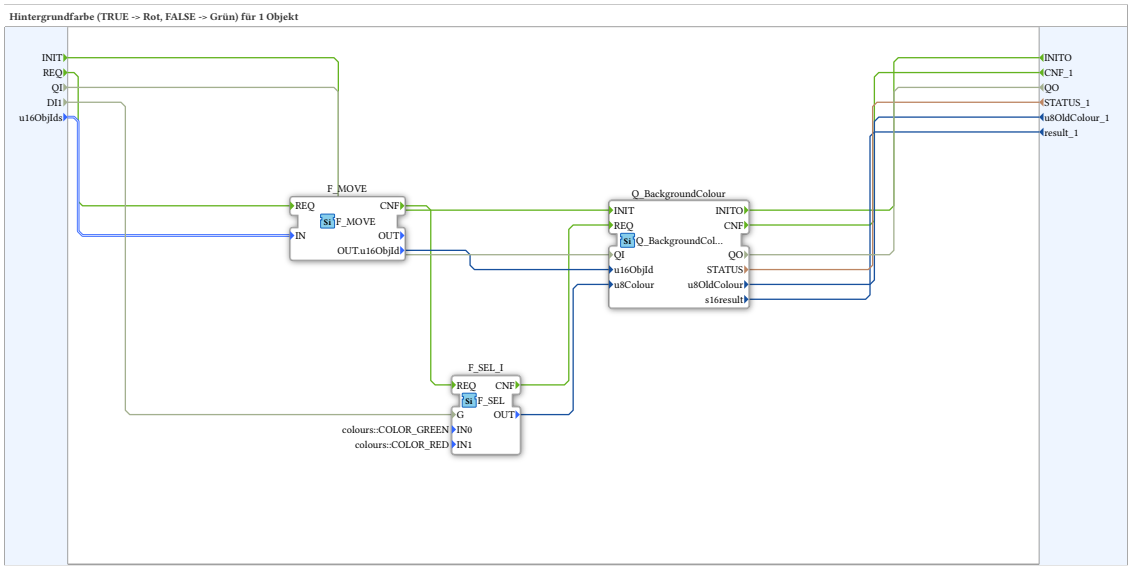
Summary

One of many variants in the background color block family: color pair Rot/Grün, 1 object, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1S



• – ○ • – ○ • – ○ •

Introduction

`RedGreenBackground1S` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColour.QI` and `Q_BackgroundColour.QO` to `QO`.



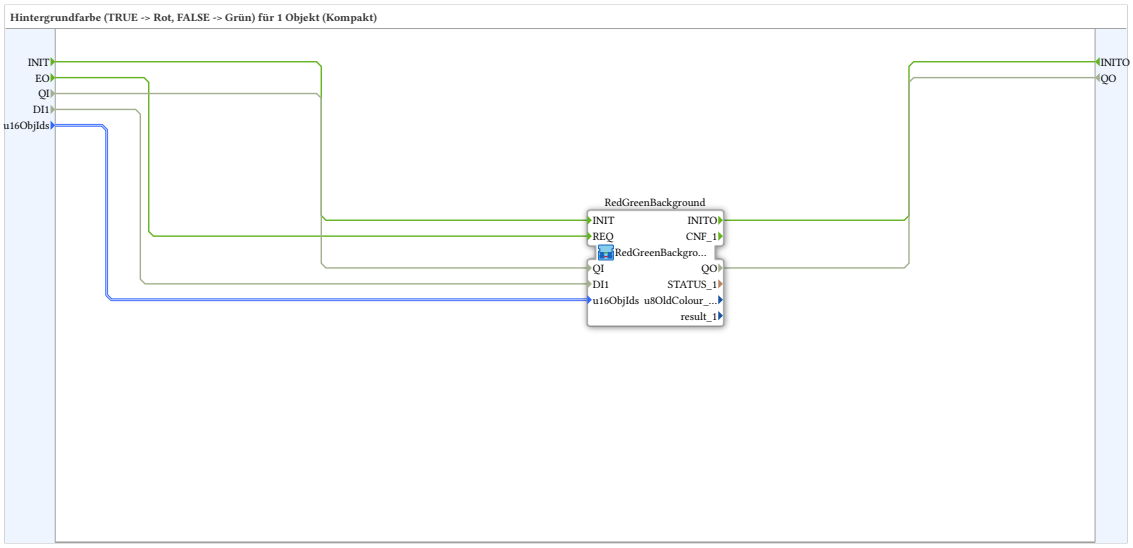
Summary

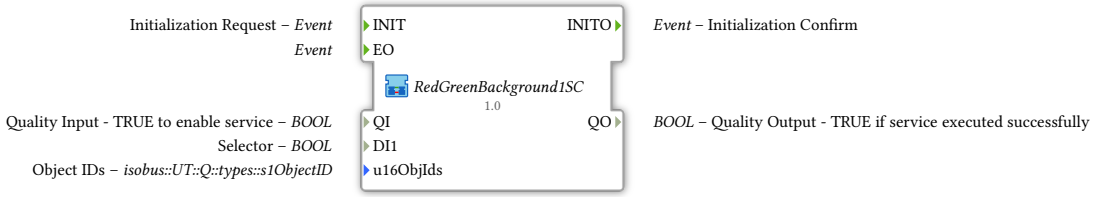
One of many variants in the background color block family: color pair Rot/Grün, 1 object, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1SC





• – ○ • – ○ • – ○ •

Introduction

`RedGreenBackground1SC` is a **compact wrapper** around `RedGreenBackground1S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `RedGreenBackground.QI`
- **Data output** `QO` (`BOOL`, Quality Output): passed through from `RedGreenBackground.QO`
- **Initialization chain**: `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`

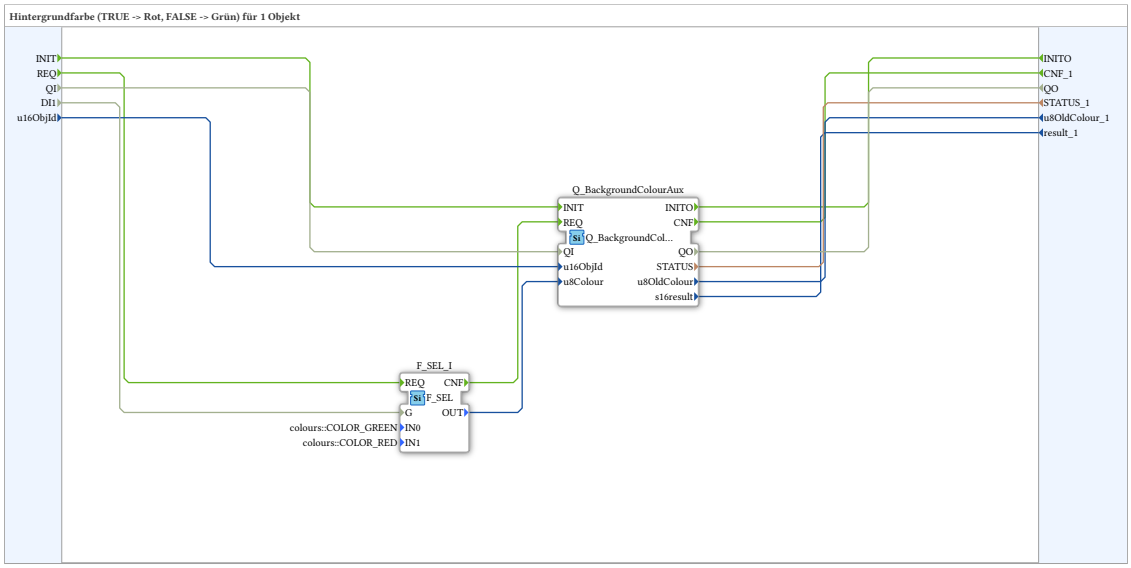
Summary

A pure pass-through to `RedGreenBackground1S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1_aux



• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground1_aux` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColourAux.QI` and `Q_BackgroundColourAux.QO` to `QO`.



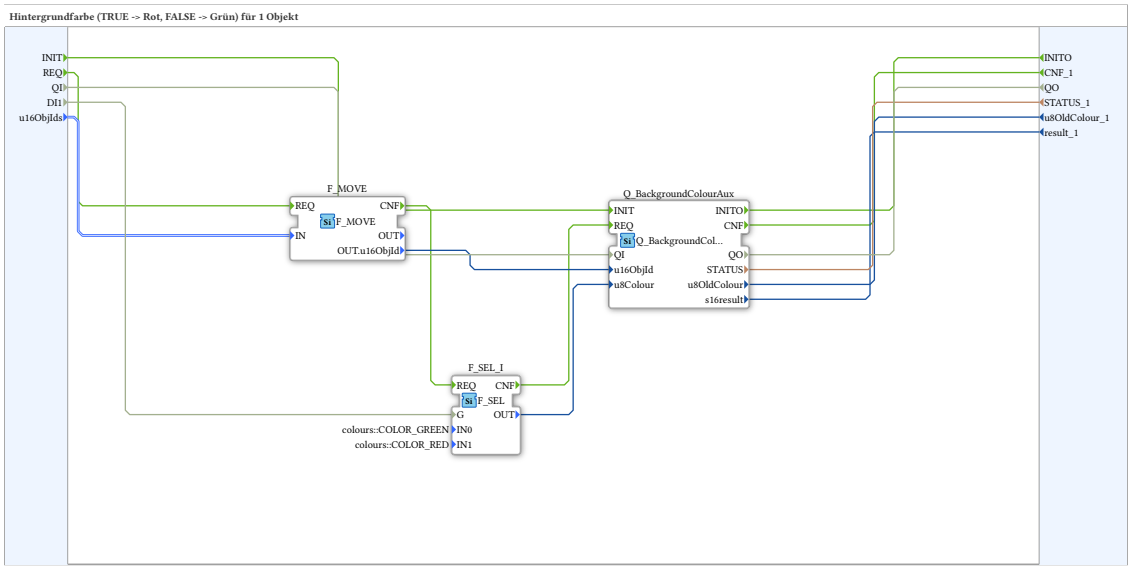
Summary

One of many variants in the background color block family: color pair Rot/Grün, 1 object, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1_auxS



• – ○ • – ○ • – ○ •

Introduction

`RedGreenBackground1_auxS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColourAux.QI` and `Q_BackgroundColourAux.QO` to `QO`.



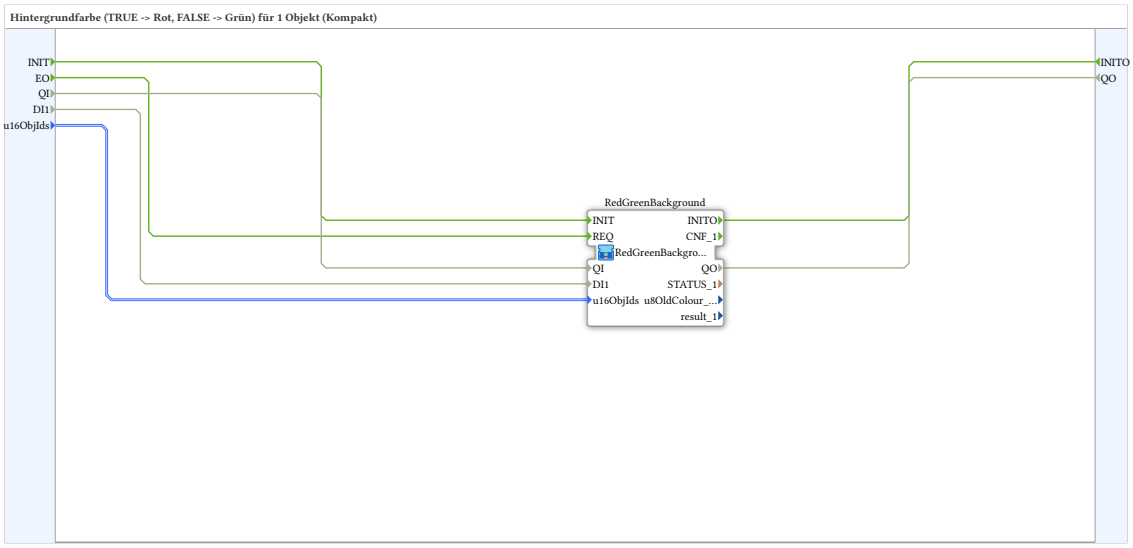
Summary

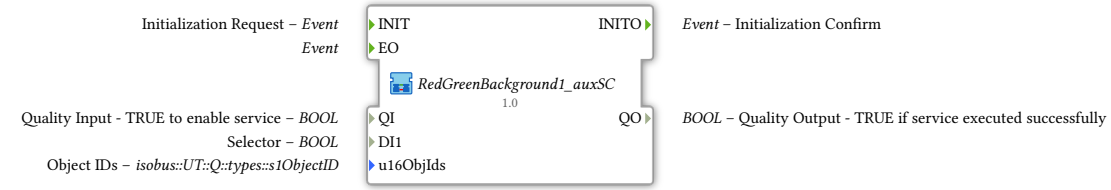
One of many variants in the background color block family: color pair Rot/Grün, 1 object, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground1_auxSC





• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground1_auxSC` is a **compact wrapper** around `RedGreenBackground1_auxS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground.QI` (initial value `TRUE`) and `RedGreenBackground.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

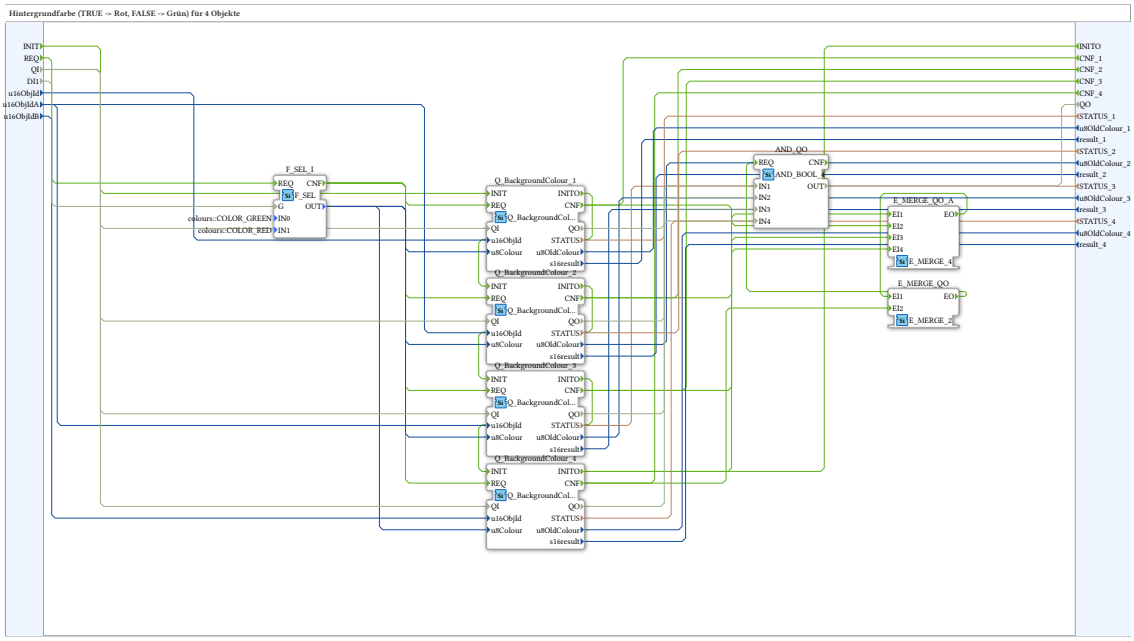
Summary

A pure pass-through to `RedGreenBackground1_auxS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground4



• — ○ • — ○ • — ○ •

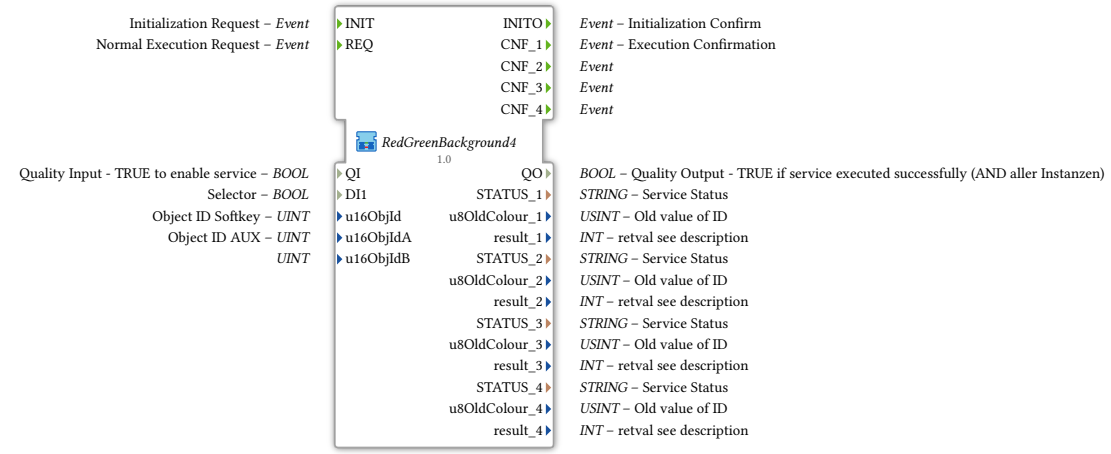
Introduction

`RedGreenBackground4` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA, u16ObjIdB`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`u16ObjIdB`	`Q_BackgroundColour` (regular object)

Initialization and quality: In addition to the previous events, the block has the event input `INIT`, the event output `INITO`, as well as `QI` (BOOL, input, initial value `TRUE`) and `QO` (BOOL, output, AND of all instances). Init chain: `INIT` ⟶ `Q_BackgroundColour_1` ⟶ `Q_BackgroundColour_2` ⟶ `Q_BackgroundColour_3` ⟶ `Q_BackgroundColour_4` ⟶ `INITO` (each `INITO` ⟶ next `INIT`). `QI` is passed on to all `Q_BackgroundColour_n.QI`. `QO` is formed by `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`, `AND_QO.OUT` ⟶ `QO`. `E_MERGE_QO_A` (`E_MERGE_4`) merges `Q_BackgroundColour_4.INITO` and the `CNF` outputs of `Q_BackgroundColour_1`, `Q_BackgroundColour_2` and `Q_BackgroundColour_3`, `E_MERGE_QO` (`E_MERGE_2`) adds `Q_BackgroundColour_4.CNF`; `E_MERGE_QO.EO` triggers `AND_QO.REQ`.

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



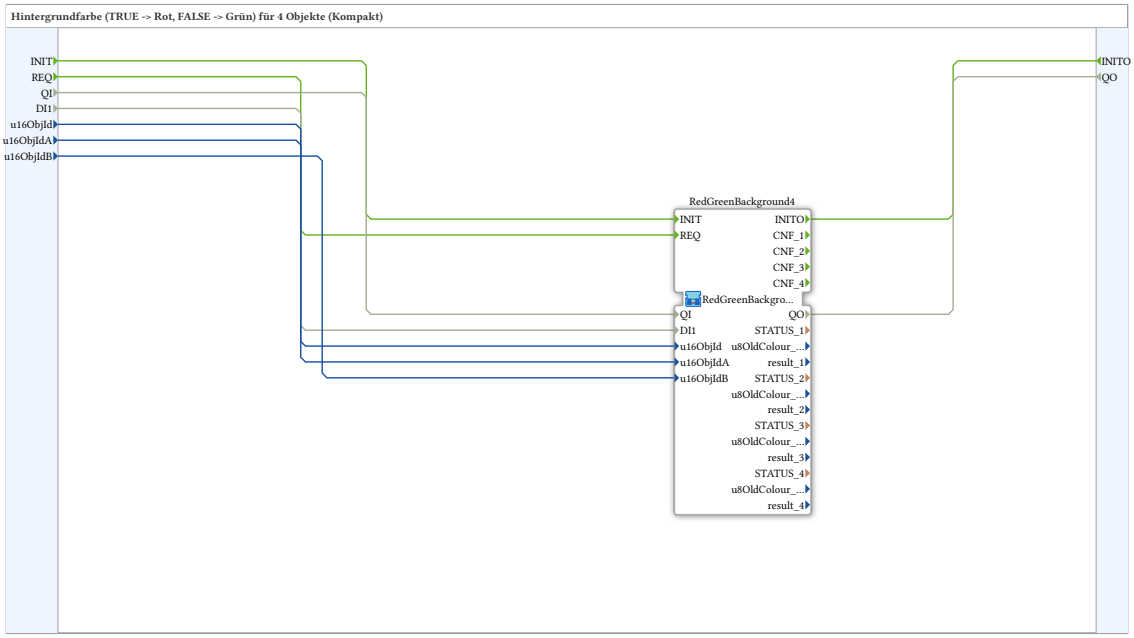
Summary

One of many variants in the background color block family: color pair Rot/Grün, 4 objects, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground4C





• – ○ • – ○ • – ○ •

Introduction

`RedGreenBackground4C` is a **compact wrapper** around `RedGreenBackground4` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `RedGreenBackground4`: `INIT` ⟶ `RedGreenBackground4.INIT`, `RedGreenBackground4.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground4.QI`, `RedGreenBackground4.QO` ⟶ `QO`.

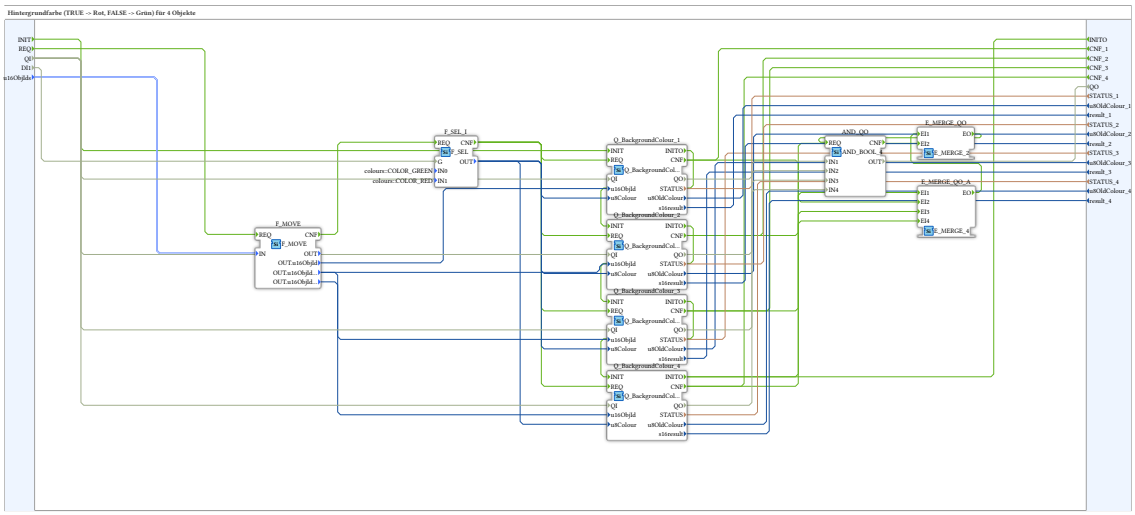
Summary

A pure pass-through to `RedGreenBackground4` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground4S



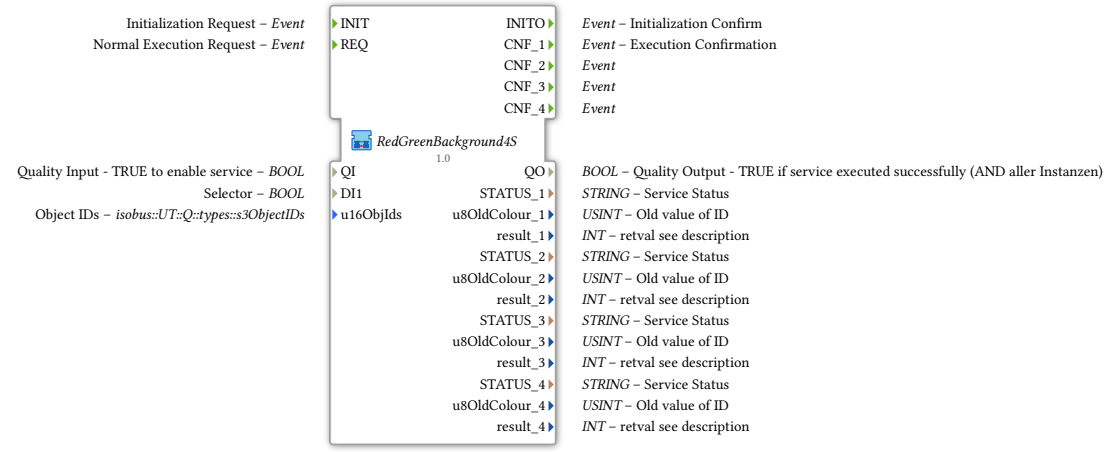
• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground4S` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Grün**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s3ObjectIDs` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` → `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` → `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`, `Q_BackgroundColour_4`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` → `AND_QO.IN1`, `Q_BackgroundColour_2.QO` → `AND_QO.IN2`, `Q_BackgroundColour_3.QO` → `AND_QO.IN3`, `Q_BackgroundColour_4.QO` → `AND_QO.IN4`; `AND_QO.OUT` → `QO`.
- `E_MERGE_QO_A` (`iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_4.INITO` and the `CNF` events of instances 1 to 3; `E_MERGE_QO` (`iec61499::events::E_MERGE_2`) merges its output `E_MERGE_QO_A.EO` with `Q_BackgroundColour_4.CNF` and thereby triggers `AND_QO.REQ`.

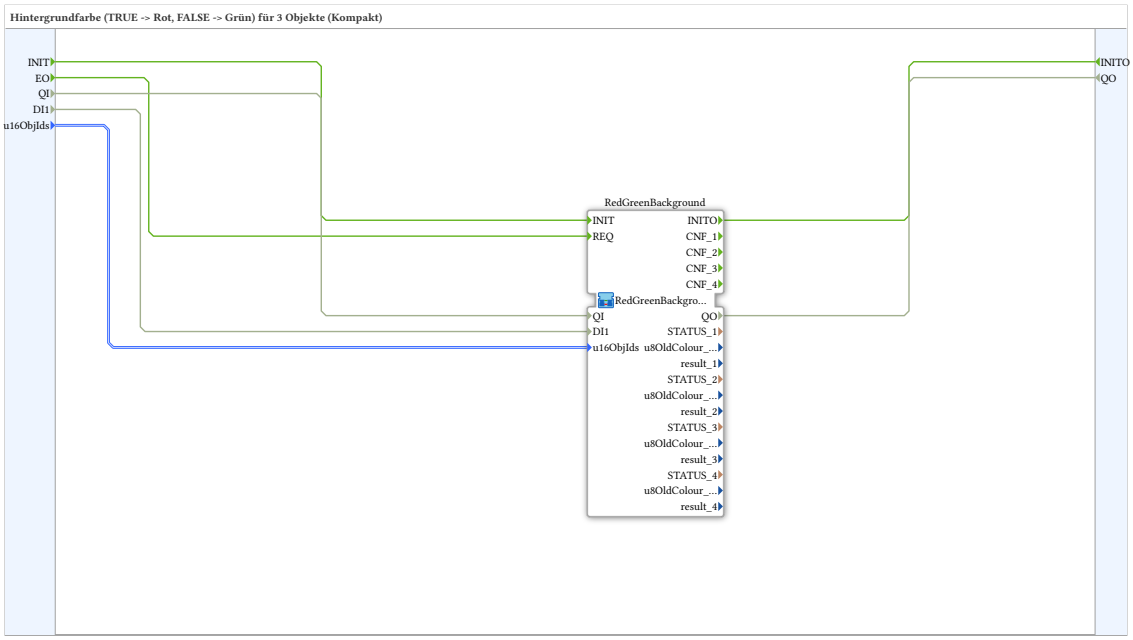
Summary

One of many variants in the background color block family: color pair Rot/Grün, 4 objects, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedGreenBackground4SC





• — ○ • — ○ • — ○ •

Introduction

`RedGreenBackground4SC` is a **compact wrapper** around `RedGreenBackground4S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `RedGreenBackground`: `INIT` ⟶ `RedGreenBackground.INIT`, `RedGreenBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedGreenBackground.QI`, `RedGreenBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

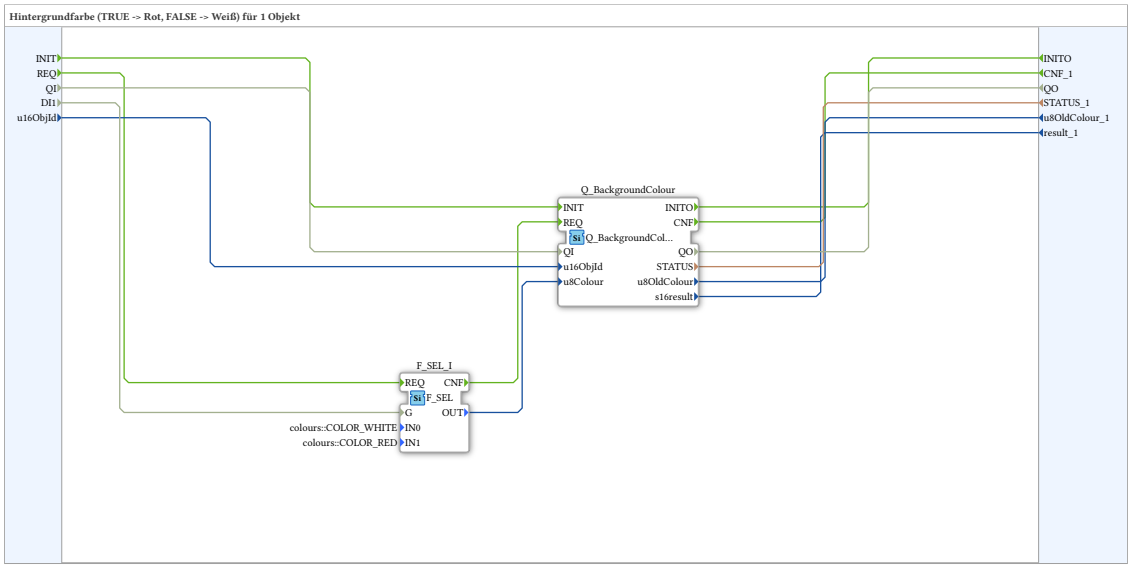
Summary

A pure pass-through to `RedGreenBackground4S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1



• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground1` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColour.QI` and `Q_BackgroundColour.QO` to `QO`.



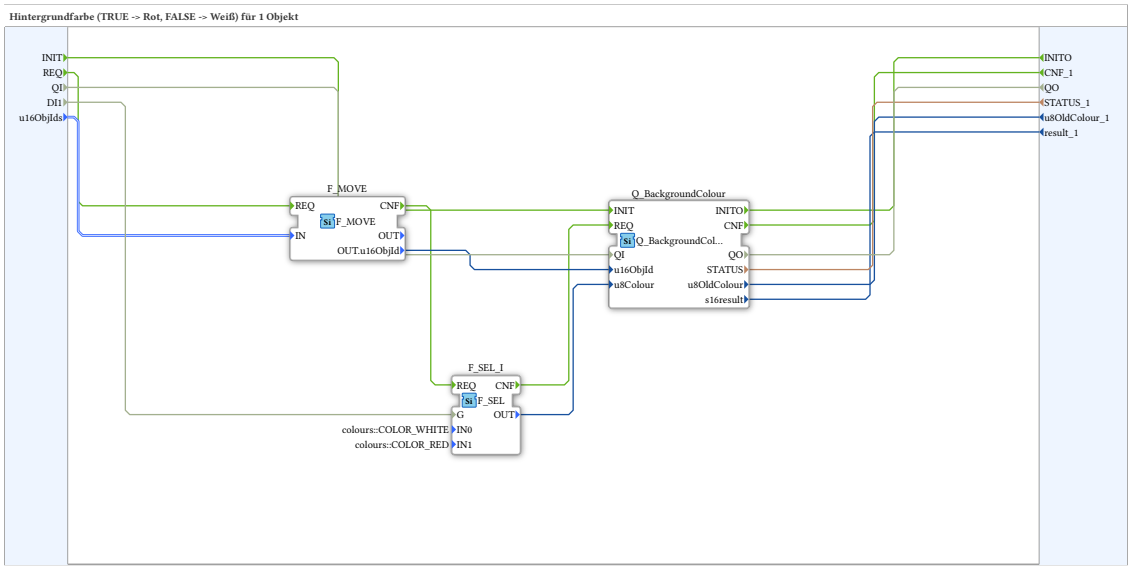
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 1 object, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1S



• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground1S` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, “Quality Input - TRUE to enable service”) and the data output `QO` (BOOL, “Quality Output - TRUE if service executed successfully”). `INIT` is passed on to `Q_BackgroundColour.INIT`, `Q_BackgroundColour.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColour.QI` and `Q_BackgroundColour.QO` to `QO`.



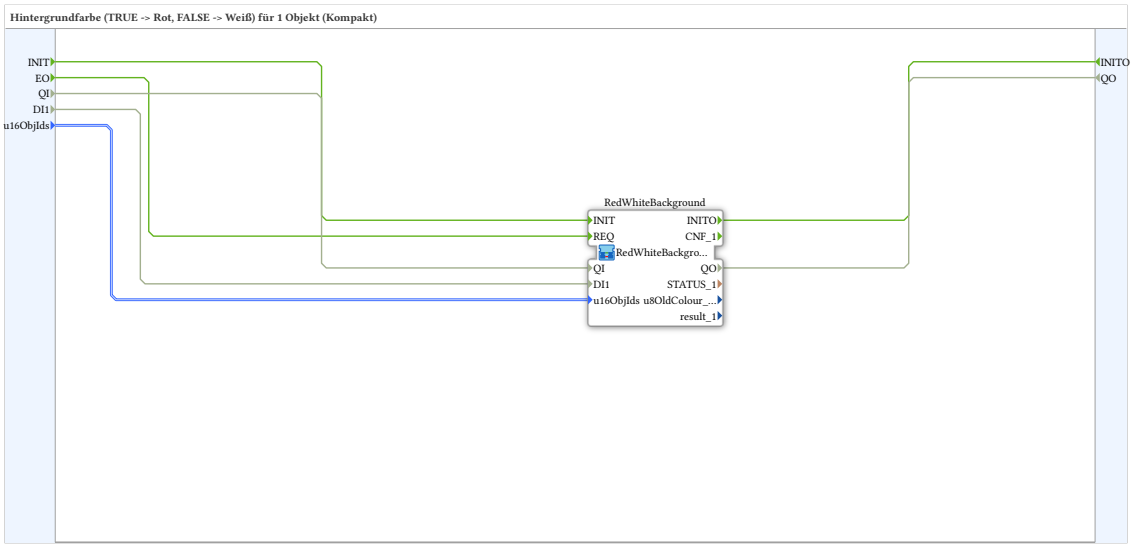
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 1 object, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1SC





• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground1SC` is a **compact wrapper** around `RedWhiteBackground1S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `RedWhiteBackground.QI`
- **Data output** `QO` (`BOOL`, Quality Output): passed through from `RedWhiteBackground.QO`
- **Initialization chain**: `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`

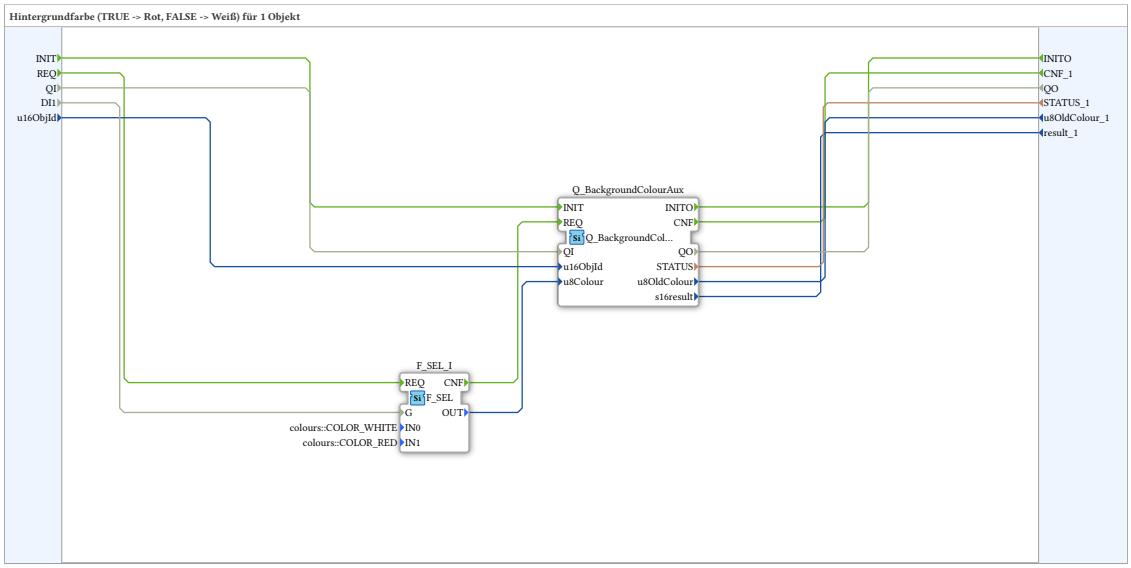
Summary

A pure pass-through to `RedWhiteBackground1S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1_aux



• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground1_aux` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via input `u16ObjId`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColourAux.QI` and `Q_BackgroundColourAux.QO` to `QO`.



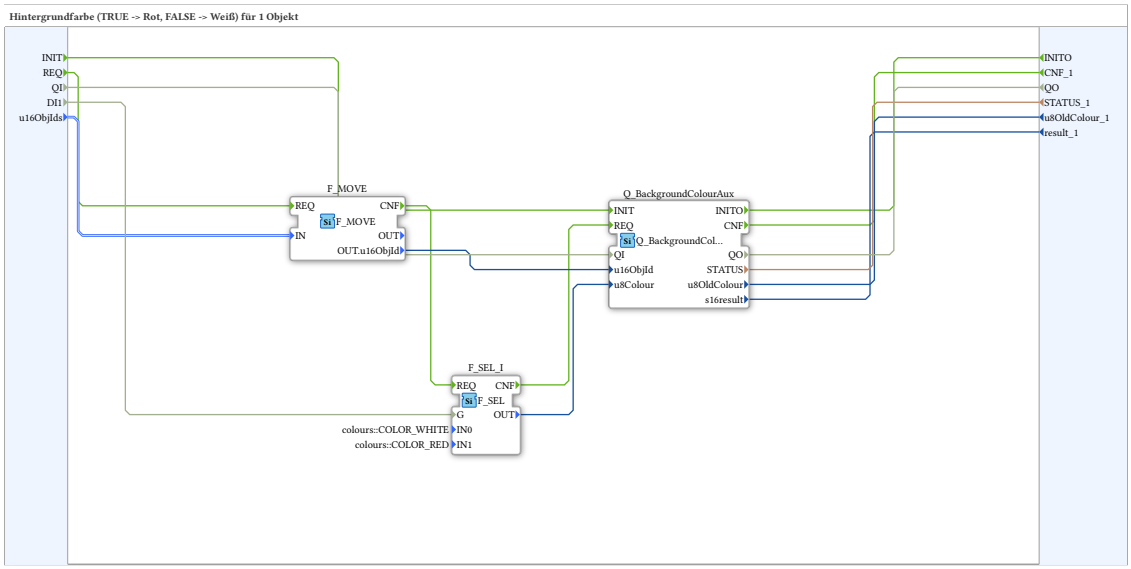
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 1 object, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1_auxS



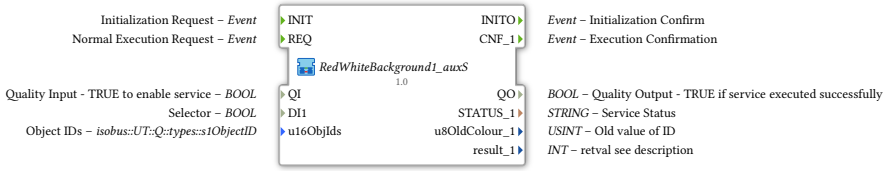
• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground1_auxS` switches the VT background color of one object based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s1ObjectID` (`u16ObjIds`).

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Quality ports: The block additionally has the event `INIT` (input) and `INITO` (output), the data input `QI` (BOOL, default `TRUE`, "Quality Input - TRUE to enable service") and the data output `QO` (BOOL, "Quality Output - TRUE if service executed successfully"). `INIT` is passed on to `Q_BackgroundColourAux.INIT`, `Q_BackgroundColourAux.INITO` is passed to `INITO`; `QI` is connected to `Q_BackgroundColourAux.QI` and `Q_BackgroundColourAux.QO` to `QO`.



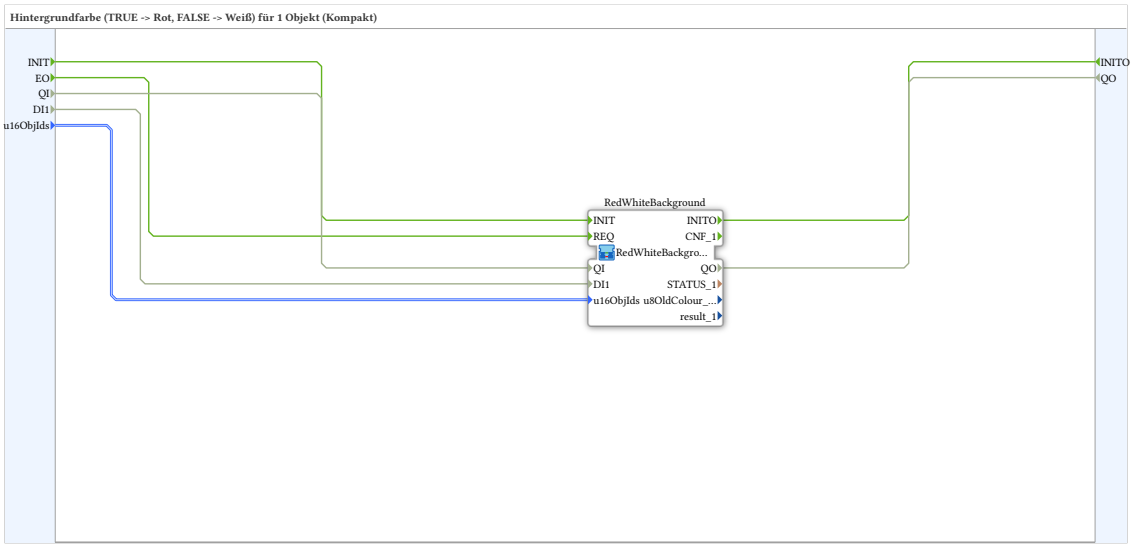
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 1 object, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground1_auxSC





• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground1_auxSC` is a **compact wrapper** around `RedWhiteBackground1_auxS` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`). In addition, `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI` (initial value `TRUE`) and `RedWhiteBackground.QO` ⟶ `QO` are passed through.

For the shared pattern, see Background Color Blocks (shared pattern).

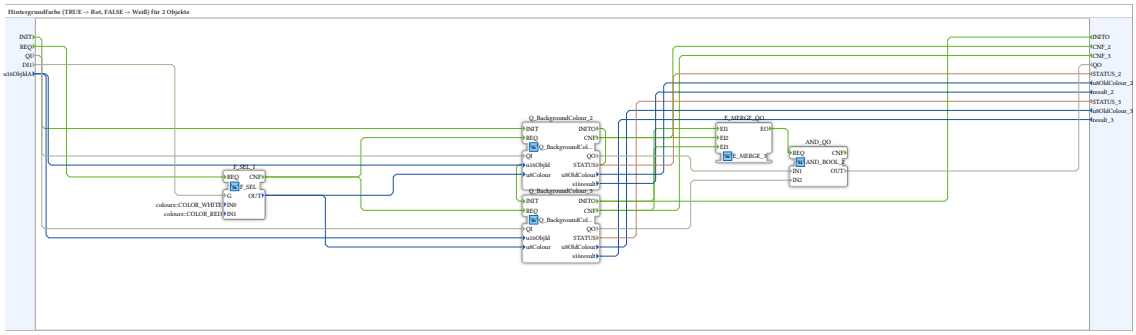
Summary

A pure pass-through to `RedWhiteBackground1_auxS` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground2



• – ○ • – ○ • – ○ •

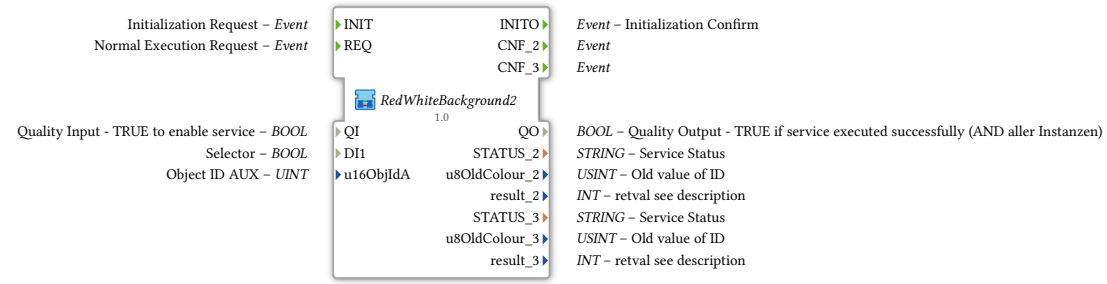
Introduction

`RedWhiteBackground2` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via inputs `u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

Initialization and quality: In addition to the previous events, the block has the event input `INIT`, the event output `INITO`, as well as `QI` (BOOL, input, initial value `TRUE`) and `QO` (BOOL, output, AND of all instances). Init chain: `INIT` ⟶ `Q_BackgroundColour_2` ⟶ `Q_BackgroundColour_3` ⟶ `INITO` (each `INITO` ⟶ next `INIT`). `QI` is passed on to all `Q_BackgroundColour_n.QI`. `QO` is formed by `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN2`, `AND_QO.OUT` ⟶ `QO`. `E_MERGE_QO` (`E_MERGE_3`) merges `Q_BackgroundColour_3.INITO` and the `CNF` outputs `Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`; `E_MERGE_QO.EO` triggers `AND_QO.REQ`.

For the general pattern (selector ⟶ `AX_SEL/F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



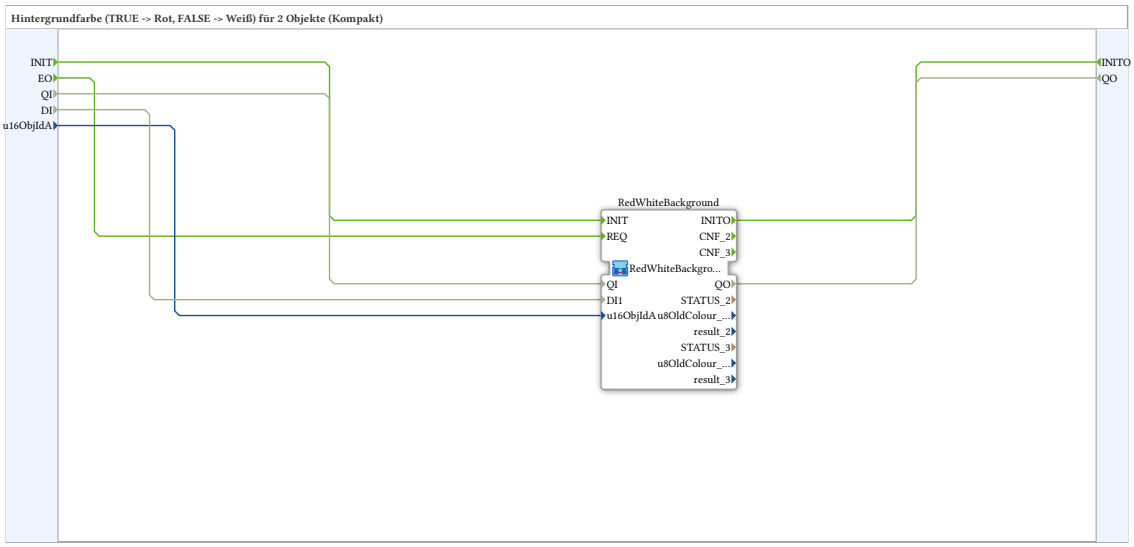
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 2 objects, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground2C





• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground2C` is a **compact wrapper** around `RedWhiteBackground2` — passes `u16ObjId` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

For the shared pattern, see Background Color Blocks (shared pattern).

In addition, the wrapper exposes the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). They are passed 1:1 through to the inner instance `RedWhiteBackground`: `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI`, `RedWhiteBackground.QO` ⟶ `QO`.

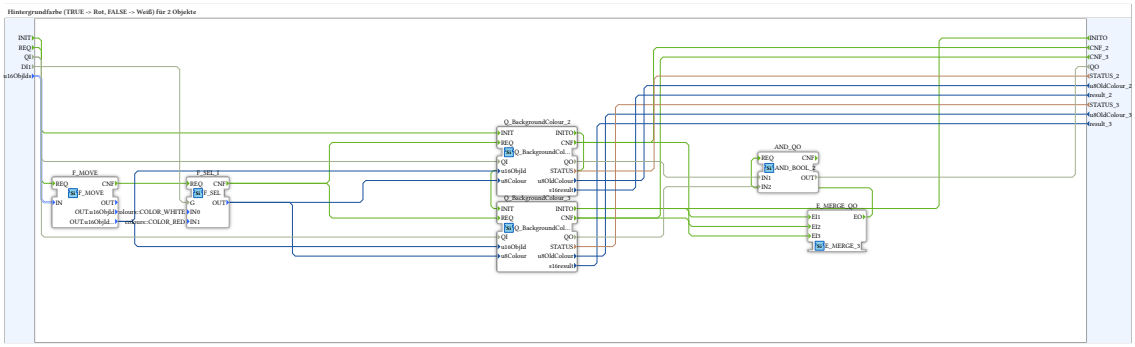
Summary

A pure pass-through to `RedWhiteBackground2` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground2S



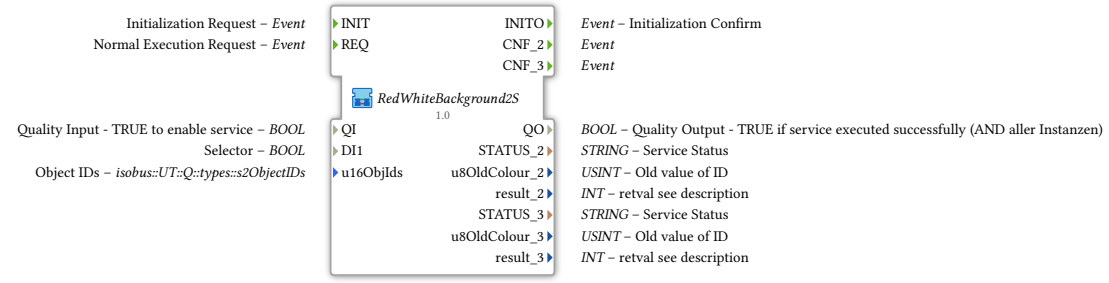
• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground2S` switches the VT background color of 2 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s2ObjectIDs` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_2`, `Q_BackgroundColour_3`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_2`): `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO` (`iec61499::events::E_MERGE_3`) merges `Q_BackgroundColour_3.INITO` and the `CNF` events of both instances (`Q_BackgroundColour_2.CNF`, `Q_BackgroundColour_3.CNF`) and thereby triggers `AND_QO.REQ`.

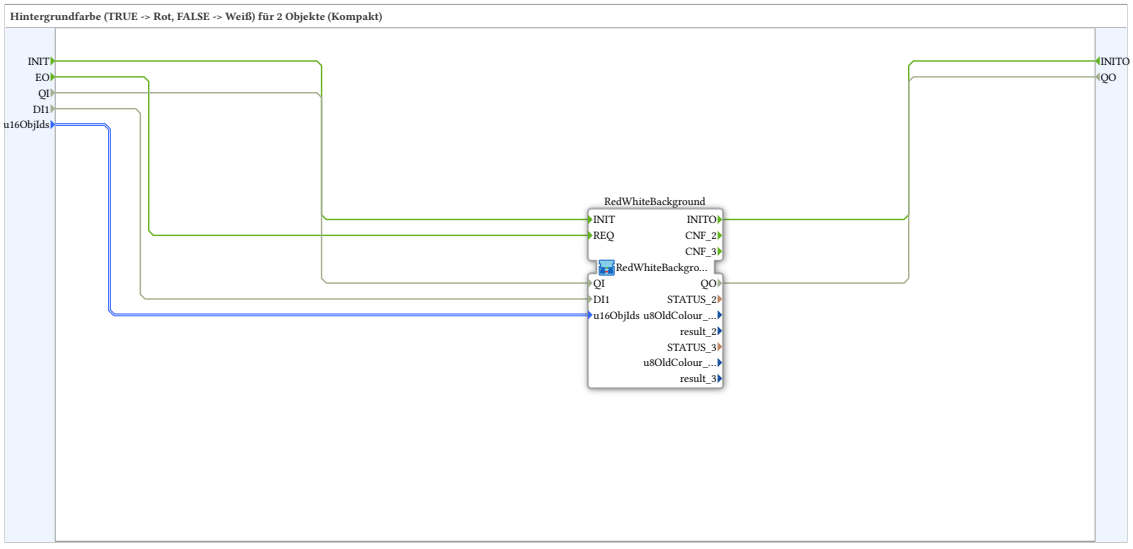
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 2 objects, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground2SC





• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground2SC` is a **compact wrapper** around `RedWhiteBackground2S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `RedWhiteBackground`: `INIT` ⟶ `RedWhiteBackground.INIT`, `RedWhiteBackground.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground.QI`, `RedWhiteBackground.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

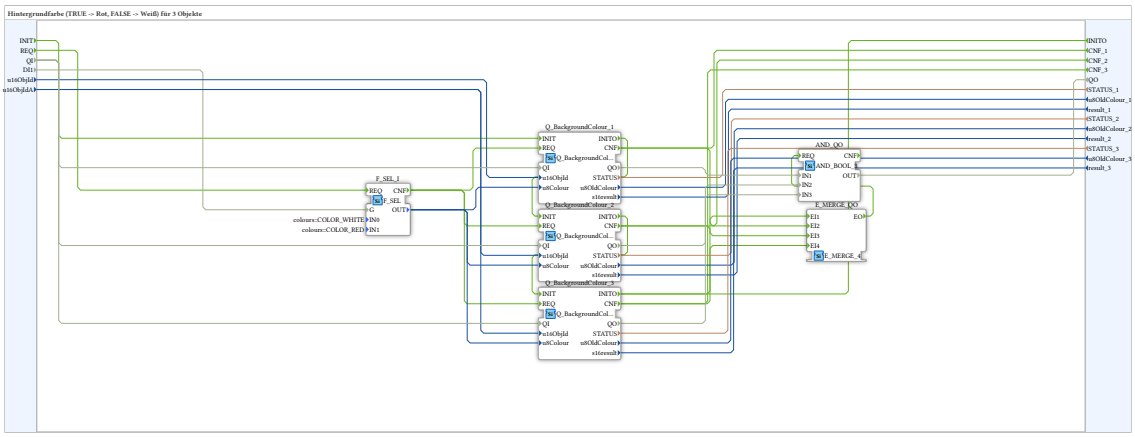
Summary

A pure pass-through to `RedWhiteBackground2S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground3



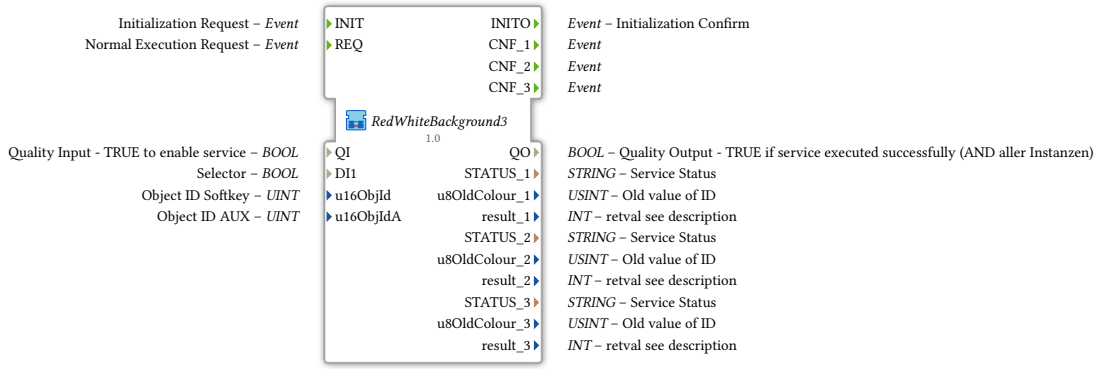
• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground3` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`, Initialization Request) and **event output** `INITO` (`EInit`, Initialization Confirm)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_BackgroundColour_1.QI`, `Q_BackgroundColour_2.QI` and `Q_BackgroundColour_3.QI`
- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_3`) of the `QO` outputs of `Q_BackgroundColour_1.QO`, `Q_BackgroundColour_2.QO` and `Q_BackgroundColour_3.QO`
- **Initialization chain**: `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `INITO`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_3.INITO`, `Q_BackgroundColour_1.CNF`, `Q_BackgroundColour_2.CNF` and `Q_BackgroundColour_3.CNF` and triggers `AND_QO.REQ`

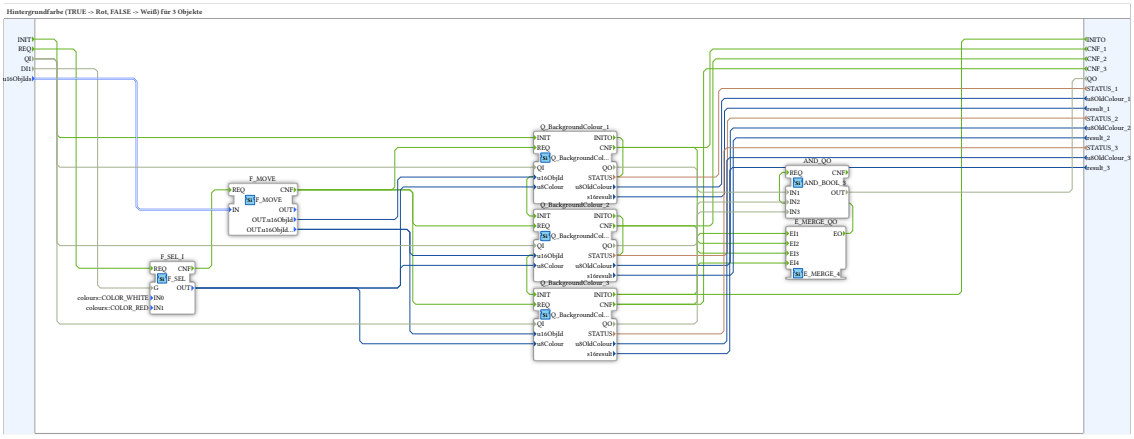
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 3 objects, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground3S



• – ○ • – ○ • – ○ •

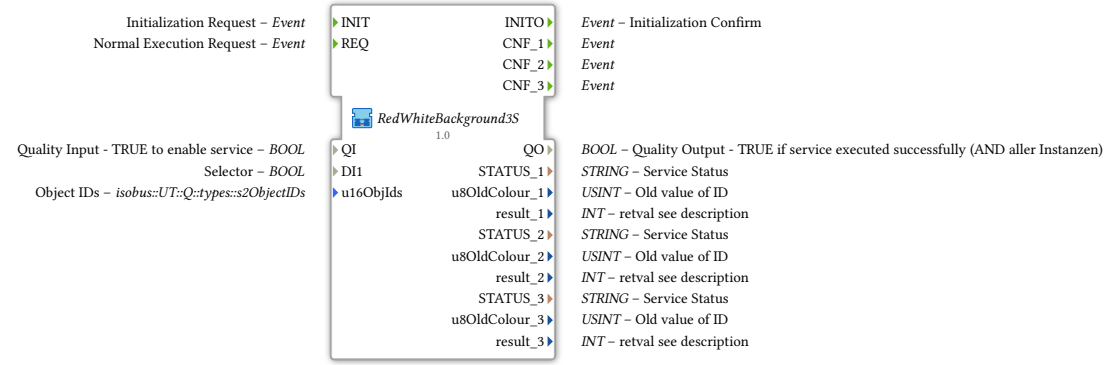
Introduction

`RedWhiteBackground3S` switches the VT background color of 3 objects based on a boolean selector signal: `TRUE` → **Rot**, `FALSE` → **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s2ObjectIDs` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

Initialization and quality: The SubApp has `INIT`/`INITO` and `QI`/`QO`. `INIT` ⟶ `Q_BackgroundColour_1.INIT` ⟶ `Q_BackgroundColour_2.INIT` ⟶ `Q_BackgroundColour_3.INIT` (each via `INITO`), `Q_BackgroundColour_3.INITO` ⟶ `INITO`. `QI` goes to all three instances. `QO` is the `AND` (`AND_QO`, `AND_BOOL_3`) of the three `QO` outputs; `AND_QO.REQ` is triggered via `E_MERGE_QO` (`E_MERGE_4`: `Q_BackgroundColour_3.INITO`, `Q_BackgroundColour_1.CNF`, `_2.CNF`, `_3.CNF`).



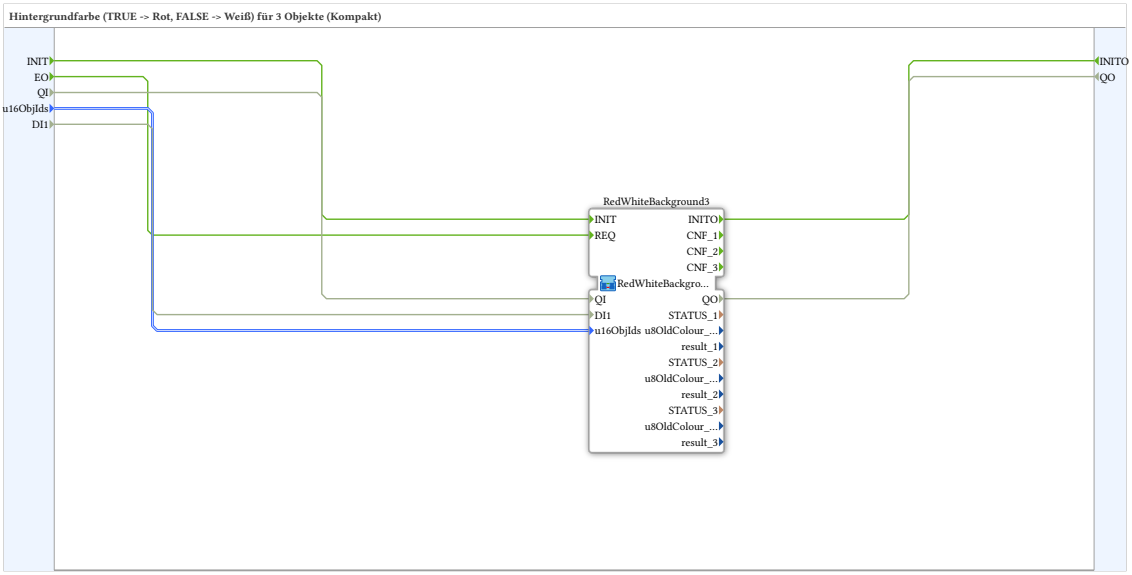
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 3 objects, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground3SC





• — ○ • — ○ • — ○ •

Introduction

`RedWhiteBackground3SC` is a **compact wrapper** around `RedWhiteBackground3S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

In addition, initialization and quality are passed through: `INIT` ⟶ `RedWhiteBackground3.INIT`, `RedWhiteBackground3.INITO` ⟶ `INITO`, `QI` (initial value `TRUE`) ⟶ `RedWhiteBackground3.QI` and `RedWhiteBackground3.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

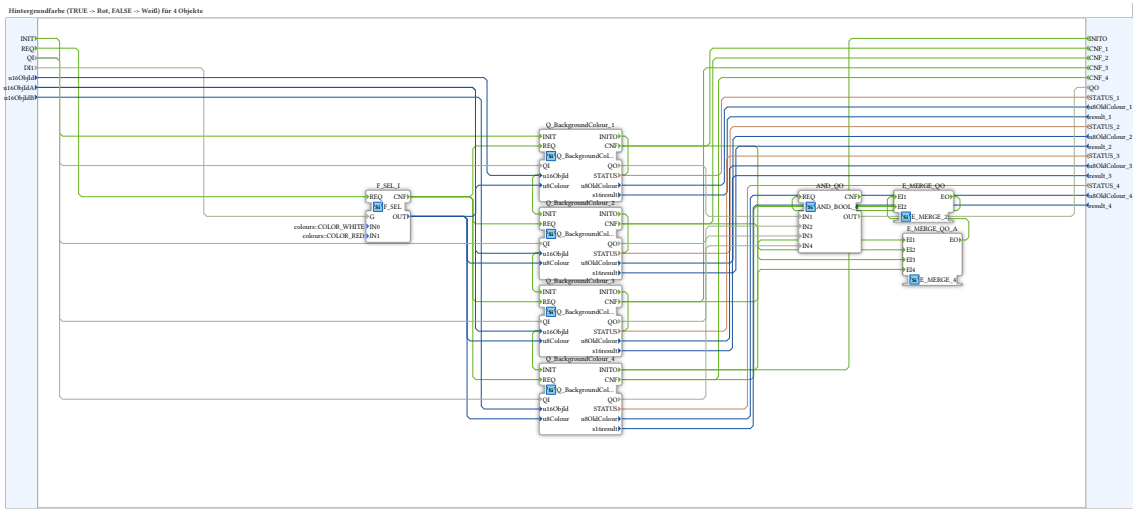
Summary

A pure pass-through to `RedWhiteBackground3S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground4



• — ○ • — ○ • — ○ •

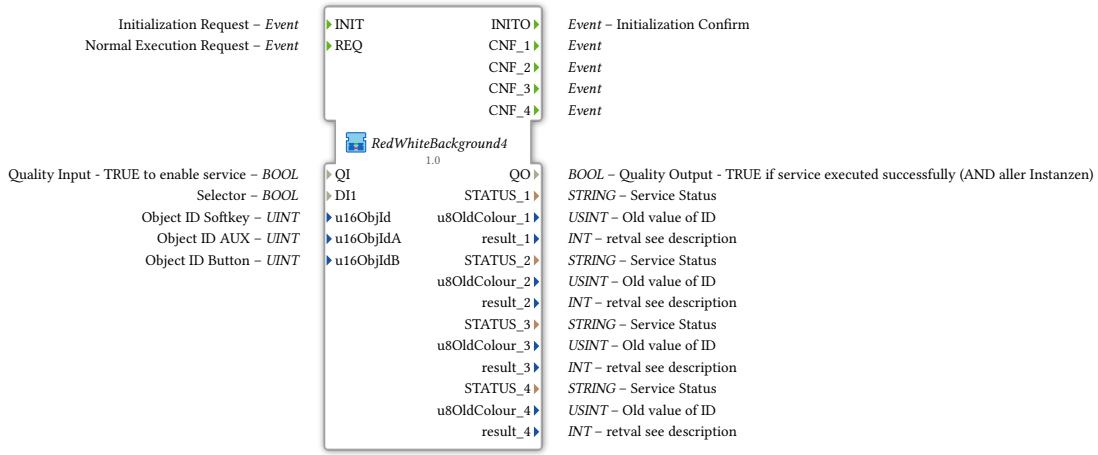
Introduction

`RedWhiteBackground4` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via inputs `u16ObjId, u16ObjIdA, u16ObjIdB`.

Position	Object ID source	Block
1	`u16ObjId`	`Q_BackgroundColour` (regular object)
2	`u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector → `AX_SEL/F_SEL` → `Q_BackgroundColour`), see Background Color Blocks (shared pattern).

In addition, the block has the initialization and quality interface: event input `INIT`, event output `INITO`, data input `QI` (`BOOL`, initial value `TRUE`) and data output `QO` (`BOOL`). `INIT` → `Q_BackgroundColour_1.INIT`, `Q_BackgroundColour_1.INITO` → `Q_BackgroundColour_2.INIT`, `Q_BackgroundColour_2.INITO` → `Q_BackgroundColour_3.INIT`, `Q_BackgroundColour_3.INITO` → `Q_BackgroundColour_4.INIT`, `Q_BackgroundColour_4.INITO` → `INITO` (so the four instances are initialized one after another). `QI` is passed to `QI` of all four instances. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`), which combines the `QO` of the four instances. Its `REQ` is triggered by `E_MERGE_QO` (`E_MERGE_2`), which merges the output of `E_MERGE_QO_A` (`E_MERGE_4`: `Q_BackgroundColour_4.INITO` and the `CNF` of `Q_BackgroundColour_1` to `_3`) and `Q_BackgroundColour_4.CNF`.



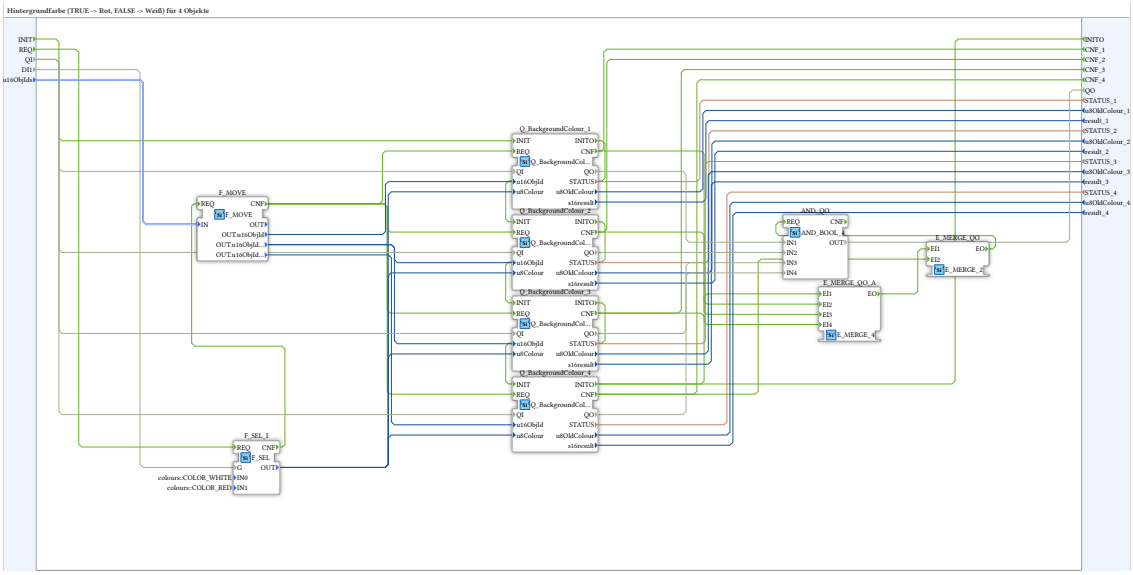
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 4 objects, BOOL selector.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground4S



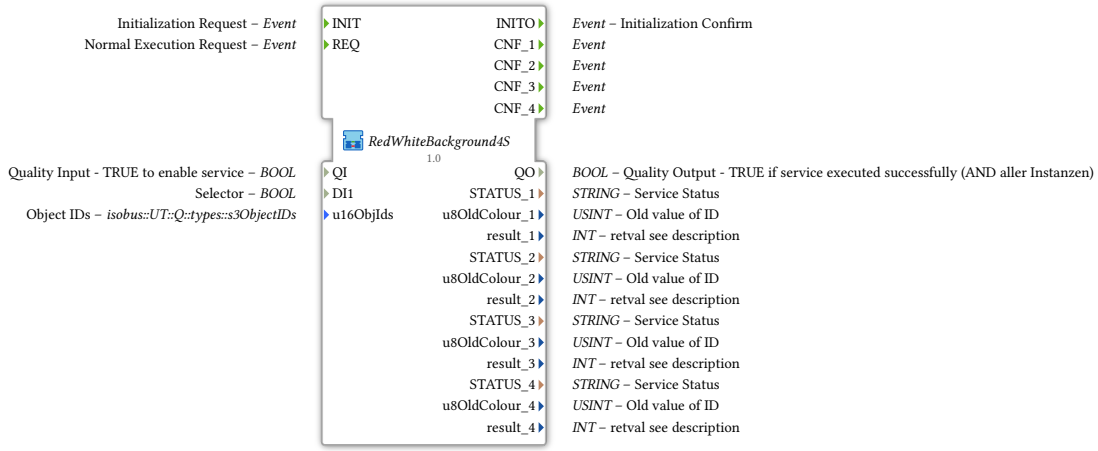
• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground4S` switches the VT background color of 4 objects based on a boolean selector signal: `TRUE` ⟶ **Rot**, `FALSE` ⟶ **Weiß**. The selector signal arrives as a plain `BOOL` data input (`DI1`). The object ID is passed via the structured type `s3ObjectIDs` (`u16ObjIds`).

Position	Object ID source	Block
1	`F_MOVE.OUT.u16ObjId`	`Q_BackgroundColour` (regular object)
2	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColour` (regular object)
3	`F_MOVE.OUT.u16ObjIdA`	`Q_BackgroundColourAux` (auxiliary function object)
4	`F_MOVE.OUT.u16ObjIdB`	`Q_BackgroundColour` (regular object)

For the general pattern (selector ⟶ `AX_SEL`/`F_SEL` ⟶ `Q_BackgroundColour`), see Background Color Blocks (shared pattern).



Initialization and Quality

- **Event input `INIT`** (`EInit`) and **event output `INITO`** (`EInit`): initialization is chained through the `Q_BackgroundColour` instances: `INIT` ⟶ `Q_BackgroundColour_1.INIT`; `Q_BackgroundColour_1.INITO` ⟶ `Q_BackgroundColour_2.INIT`; `Q_BackgroundColour_2.INITO` ⟶ `Q_BackgroundColour_3.INIT`; `Q_BackgroundColour_3.INITO` ⟶ `Q_BackgroundColour_4.INIT`; `Q_BackgroundColour_4.INITO` ⟶ `INITO`.
- **Data input `QI`** (`BOOL`, default `TRUE`) is distributed to `.QI` of all instances (`Q_BackgroundColour_1`, `Q_BackgroundColour_2`, `Q_BackgroundColour_3`, `Q_BackgroundColour_4`).
- **Data output `QO`** (`BOOL`): AND of all instances via `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`): `Q_BackgroundColour_1.QO` ⟶ `AND_QO.IN1`, `Q_BackgroundColour_2.QO` ⟶ `AND_QO.IN2`, `Q_BackgroundColour_3.QO` ⟶ `AND_QO.IN3`, `Q_BackgroundColour_4.QO` ⟶ `AND_QO.IN4`; `AND_QO.OUT` ⟶ `QO`.
- `E_MERGE_QO_A` (`iec61499::events::E_MERGE_4`) merges `Q_BackgroundColour_4.INITO` and the `CNF` events of instances 1 to 3; `E_MERGE_QO` (`iec61499::events::E_MERGE_2`) merges its output `E_MERGE_QO_A.EO` with `Q_BackgroundColour_4.CNF` and thereby triggers `AND_QO.REQ`.

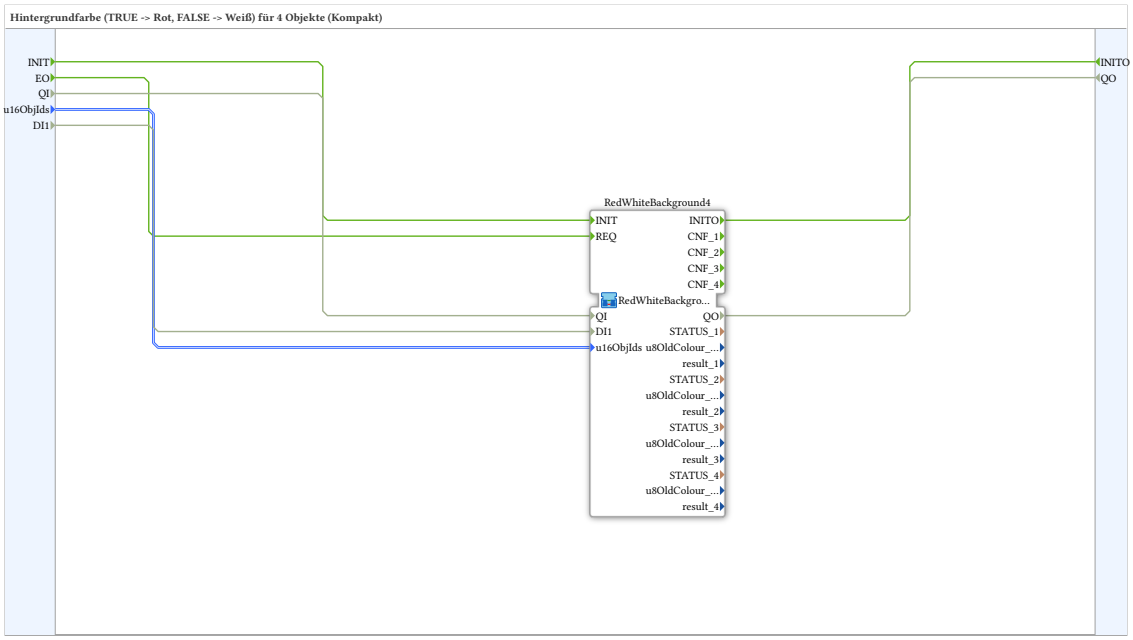
Summary

One of many variants in the background color block family: color pair Rot/Weiß, 4 objects, BOOL selector, struct. object ID.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

RedWhiteBackground4SC





• – ○ • – ○ • – ○ •

Introduction

`RedWhiteBackground4SC` is a **compact wrapper** around `RedWhiteBackground4S` — passes `u16ObjIds` and the selector straight through, without exposing the `CNF` event outputs and diagnostic values (`STATUS_n`, `u8OldColour_n`, `result_n`).

Interface (quality pattern): The wrapper additionally has the event input `INIT` and the event output `INITO` (each `EInit`) as well as the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). They are passed straight through to the inner instance `RedWhiteBackground4`: `INIT` ⟶ `RedWhiteBackground4.INIT`, `RedWhiteBackground4.INITO` ⟶ `INITO`, `QI` ⟶ `RedWhiteBackground4.QI`, `RedWhiteBackground4.QO` ⟶ `QO`.

For the shared pattern, see Background Color Blocks (shared pattern).

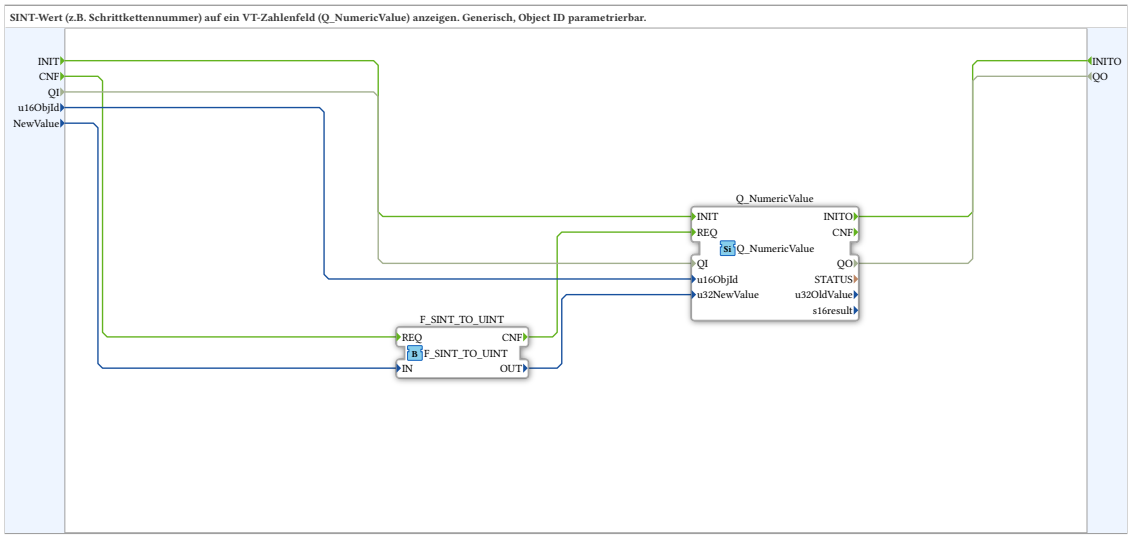
Summary

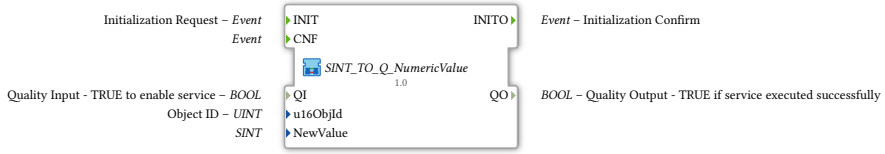
A pure pass-through to `RedWhiteBackground4S` for the common case where the diagnostic values aren't needed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SINT_TO_Q_NumericValue





• – ○ • – ○ • – ○ •

Introduction

The **SINT_TO_Q_NumericValue** subapplication provides a generic and reusable solution for displaying a signed 8-bit integer value (e.g., a step number in a sequence) on a numeric visualization field (Q_NumericValue) within a VT (Visualization Terminal). The object ID of the target display element is parameterizable, making the block adaptable to different HMI layouts without internal modification. It decouples the source data type (SINT) from the display element's expected input type (UINT) by performing a conversion internally.

Interface Structure

Event Inputs

Event	Description
`INIT`	Initialization request; forwarded to `Q_NumericValue.INIT`.
`CNF`	Triggers the conversion and subsequent update of the numeric display field.

Event Outputs

Event	Description
`INITO`	Initialization confirm from `Q_NumericValue.INITO`.

Data Inputs

Data Name	Type	Initial Value	Description
`QI`	BOOL	`TRUE`	Quality Input - TRUE to enable service; passed through to `Q_NumericValue.QI`.
`u16ObjId`	UINT	`ID_NULL`	Object ID of the target Q_NumericValue element on the visualization screen.

NewValue	SINT	–	Signed 8-bit value to be converted and displayed.

Data Outputs

Data Name	Type	Description
QO	BOOL	Quality Output - TRUE if the service executed successfully; taken from Q_NumericValue.QO.

Adapters

None.

Functionality

The INIT event is forwarded to Q_NumericValue.INIT, and its INITO is output at INITO. QI is passed through to Q_NumericValue.QI, and Q_NumericValue.QO is passed through to QO. Upon receiving an event at the CNF input, the subapplication performs the following operations internally:

1. The value on NewValue (SINT) is passed to the conversion function block **F_SINT_TO_UINT** (IEC 61131-3 compliant) via the IN input.
2. The conversion block produces the corresponding unsigned 32-bit representation on its OUT output.
3. This converted value is routed to the u32NewValue input of the **Q_NumericValue** function block, which represents the target visualization element.
4. Simultaneously, the u16ObjId input is forwarded to the u16ObjId input of **Q_NumericValue**, telling the visualization which display field to update.
5. The function block Q_NumericValue is then triggered by the REQ event (sourced from the completion of the conversion, CNF of F_SINT_TO_UINT), causing it to update the display with the new numeric value.

The entire process is combinational – there is no internal state storage; each CNF event produces one display update.

Technical Features

- **Data Type Conversion** – Uses a standard IEC 61131-3 conversion block (F_SINT_TO_UINT) to safely map the signed 8-bit input to an unsigned 32-bit value.
- **Parameterizable Object ID** – The u16ObjId input allows the user to specify which numeric field on the HMI should be updated, enabling reuse of the subapplication across multiple screens and elements.
- **Generic Design** – The block is completely independent of the specific step sequence or machine logic; it only provides the interface and conversion logic.
- **No Internal State** – Because no data is stored, the block is stateless and can be used in cyclic or event-driven environments without concerns about stale data.
- **Compact and Reusable** – Encapsulates common conversion and display logic, reducing duplication in larger projects.

State Overview

This subapplication does not contain any state memory or sequential logic. Every activation of the CNF event immediately leads to a fresh conversion and update of the display field. Therefore, no state diagram is applicable; the behavior is purely event-driven and deterministic.

Application Scenarios

- **Step Number Display** – In a sequential control system (e.g., a packaging machine), a SINT step counter can be shown on an HMI numeric field.
- **Parameter Value Visualization** – Any signed 8-bit parameter (temperature offset, speed correction, etc.) that needs to be presented on a VT.
- **Generic HMI Binding** – When multiple numeric fields are used, each can be connected to a separate instance of this subapplication with a different u16ObjId.
- **Modular Automation Projects** – Used as a building block in larger subapplications or FBs, where conversion and display logic is required repeatably.

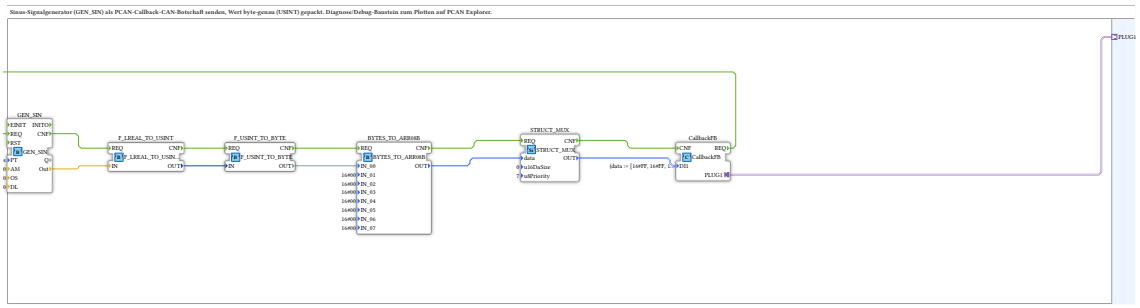
Comparison with Similar Blocks

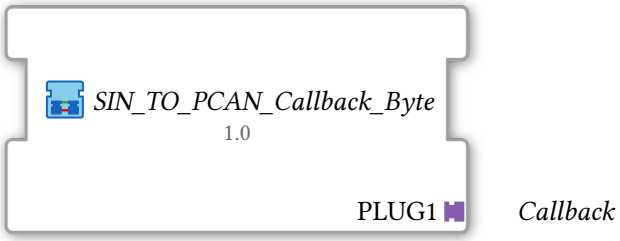
- **Direct Q_NumericValue usage** – Without this subapplication, one would have to connect a UINT source directly to Q_NumericValue and handle the conversion externally. This block encapsulates the conversion, reducing wiring and potential errors.
- **Other conversion blocks** – Standard conversion FBs (e.g., F_SINT_TO_DINT) could be used, but this subapplication combines conversion with HMI communication, offering a higher-level interface tailored to visualization tasks.
- **Custom display logic** – Writing custom code for each display element would be error-prone and not reusable; this subapplication provides a uniform, parameterized solution.

Conclusion

The **SINT_TO_Q_NumericValue** subapplication is a focused, non-intrusive utility that bridges the gap between a signed 8-bit data source and a numeric HMI display. By integrating type conversion and display addressing into a single, stateless block, it simplifies project engineering, improves maintainability, and promotes reuse across different automation systems. Its generic object ID parameter ensures flexibility, while its straightforward event interface makes it easy to integrate into any IEC 61499-based control application.

SIN_TO_PCAN_Callback_Byte





• – ○ • – ○ • – ○ •

Introduction

The **SIN_TO_PCAN_Callback_Byte** subapplication is a diagnostic and debugging tool designed to generate a sinusoidal signal and transmit it as a CAN message via PCAN, using a callback-based adapter interface. The signal value is converted into a single unsigned byte (USINT) and packed into a CAN frame, making it suitable for real-time plotting and analysis in tools such as PCAN Explorer. This subapplication encapsulates the entire signal generation, conversion, and transmission chain into a reusable component.

Interface Structure

The subapplication exposes only one external interface: a plug adapter for the CAN callback. It has no direct event or data inputs/outputs; all control and data flow is handled internally.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Name	Type	Description
PLUG1	isobus::pgn::tx::Callback	Callback interface to send CAN messages (e.g., to PCAN) upon request. This adapter is used to pass the structured CAN message prepared by the internal CallbackFB function block.

Functionality

The subapplication performs the following steps in a continuous loop:

1. **Signal Generation**: The GEN_SIN function block (from the OSCAT library) creates a sinusoidal waveform with the given parameters (period PT, amplitude AM, offset OS, and dead level DL). In this configuration, the period is 10 seconds, amplitude 10.0, and offset 5.0, producing a sine wave oscillating between −5.0 and 15.0.
2. **Conversion to USINT**: The F_LREAL_TO_USINT conversion block transforms the LREAL output from GEN_SIN into an USINT (unsigned 8-bit integer) value.
3. **Byte Conversion**: The F_USINT_TO_BYTE block converts the USINT value into a byte representation.
4. **Array Assembly**: The BYTES_TO_ARR08B block places the converted byte into the first byte of an 8-byte array. This block also supports byte reversal (reversing order) and is configured with initial placeholder values for the remaining seven bytes (all set to 16#00).
5. **Message Structuring**: The STRUCT_MUX block packs the byte array into a structured CAN message of type isobus::pgn::CAN_MSG. It also sets the payload size (u16DaSize) to 0 and the default priority (u8Priority) to 7.
6. **Transmission**: The CallbackFB function block receives the structured CAN message and, when triggered, sends it via the PLUG1 adapter (connected to a callback interface, typically a PCAN driver).

The event chain is: CallbackFB.REQ → GEN_SIN.REQ → GEN_SIN.CNF → F_LREAL_TO_USINT.REQ → F_LREAL_TO_USINT.CNF → F_USINT_TO_BYTE.REQ → F_USINT_TO_BYTE.CNF → BYTES_TO_ARR08B.REQ → BYTES_TO_ARR08B.CNF → STRUCT_MUX.REQ → STRUCT_MUX.CNF → CallbackFB.CNF. Each step passes the required data along the corresponding data connections.

Technical Features

- **Sinusoidal Generation**: Configurable via parameters PT, AM, OS, and DL to adapt the signal characteristics.
- **Precision Conversion**: The signal value is rounded to an unsigned 8-bit integer, ensuring byte-level accuracy.
- **CAN Message Formatting**: Uses a structured type isobus::pgn::CAN_MSG for compatibility with ISO 11783 (ISOBUS) PGN transmission.
- **Callback-based Transmission**: The output adapter PLUG1 uses a callback interface, allowing integration with event-driven PCAN drivers.
- **Reusable Design**: The subapplication can be instantiated multiple times with different parameters or data sources.
- **Debug Friendly**: The byte-only representation simplifies plotting and analysis in external tools.

State Overview

This subapplication does not maintain any internal state machine. It operates purely as a data-flow pipeline, where each function block processes its input and produces an output in a single scan cycle. The only dynamic behavior is the continuous generation of sinusoidal values by GEN_SIN, which is driven by the event request from CallbackFB. All conversions and message assembly are combinatorial in nature.

Application Scenarios

- **Real-time Signal Visualization**: Use the sinusoidal output to test or calibrate CAN-based measurement systems.
- **Function Testing of PCAN Interfaces**: Verify correct data packing and transmission in a PCAN environment.
- **Educational Tool**: Demonstrate how to generate, convert, and send analog signals over CAN in an industrial automation context.
- **Prototyping**: Quickly create a CAN node that broadcasts a sinusoidal test signal for development purposes.

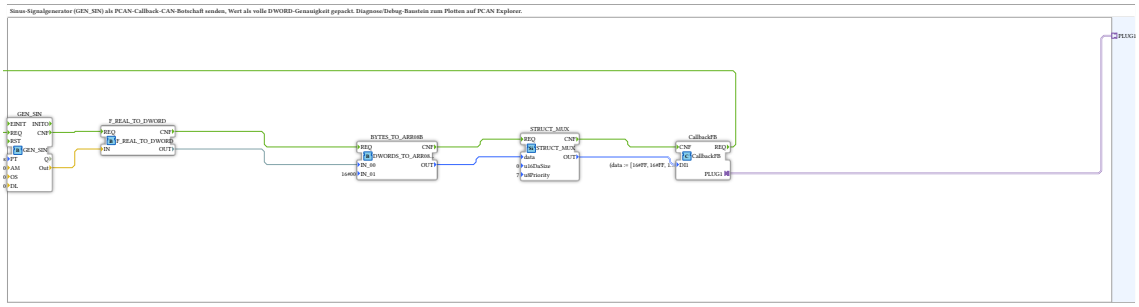
Comparison with Similar Blocks

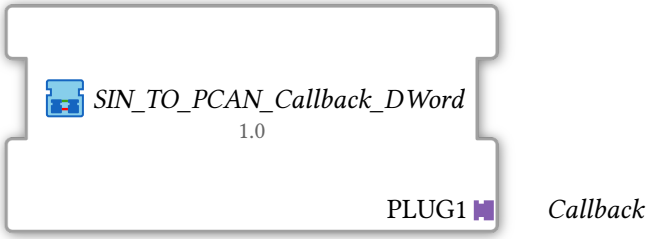
Unlike a standard analog-to-CAN converter that may send floating-point values, this subapplication specifically packs the signal into a single byte, sacrificing resolution for simplicity and compactness. It also uses a callback-based adapter rather than a direct CAN output, making it more flexible for different bus interfaces. Compared to a generic `GEN_SIN` block used standalone, this subapplication integrates the conversion and CAN transmission chain into a single ready-to-use component. It is similar in purpose to dedicated signal generator modules but is optimized for debugging and analysis via PCAN tools.

Conclusion

The **SIN_TO_PCAN_Callback_Byte** subapplication is a compact and reusable component for generating sinusoidal signals and transmitting them as CAN messages with byte-level precision. Its modular design, based on standard 4diac-IDE function blocks, ensures easy integration into larger projects. The clear separation of signal generation, conversion, and transmission makes it an ideal choice for diagnostic tasks, prototype development, and educational use in industrial automation environments.

SIN_TO_PCAN_Callback_DWord





• – ○ • – ○ • – ○ •

Introduction

The **SIN_TO_PCAN_Callback_DWord** subapplication is a diagnostic and debugging oriented function block that generates a sinusoidal signal and transmits it as a CAN message through a PCAN callback adapter. The generated signal value is packed into the CAN data field with full DWORD (32-bit) precision, using a reversed byte order for proper little-endian interpretation on the receiving side. The primary purpose of this block is to provide a real-time, easily plottable signal stream for tools such as PCAN Explorer, allowing engineers to visualize and verify the sine wave directly on the CAN bus.

Interface Structure

The subapplication exposes a single adapter interface and no direct event or data ports. All interaction with the outside world is performed through this adapter.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter Name	Type	Direction	Description
PLUG1	`isobus::pgn::tx::Callback`	Plug	Callback-based CAN transmission interface. Provides the mechanism to send CAN messages to the connected PCAN channel via the callback system.

Functionality

The subapplication implements a self-contained signal-generation and CAN-forwarding pipeline. The functional flow is initiated when the callback adapter receives a request from the connected partner (typically the PCAN callback driver). Upon this trigger, the following processing chain is executed:

1. **Trigger generation** – The `CallbackFB` receives the adapter request and forwards it as an event to the sine generator.
2. **Sine wave synthesis** – The `GEN_SIN` function block produces a sinusoidal output value with the configured parameters: a period of 10 seconds, an amplitude of 10.0, an offset of 5.0, and no dead time.
3. **Data type conversion** – The floating-point sine value is converted to a DWORD (32-bit unsigned integer) using the `F_REAL_TO_DWORD` conversion block, preserving the full bit pattern of the REAL value.
4. **Byte array transformation** – The DWORD is converted into an 8-byte array with reversed byte order (`DWORDS_TO_ARR08B`). This ensures that the DWORD is placed in the least significant four bytes in a little-endian (reversed) arrangement, while the upper four bytes are padded with zero.
5. **Message structure assembly** – The byte array is packed into a `CAN_MSG` structured type using `STRUCT_MUX`. The message is configured with priority 7 and a data size field set to 0, which represents a standard CAN frame with an 8-byte data payload.
6. **Transmission** – The assembled CAN message is handed back to the `CallbackFB`, which sends it over the adapter to the PCAN bus.

Technical Features

- **Full DWORD precision**: The sine value is transported as a raw 32-bit IEEE 754 representation, preserving all decimal places and enabling high-resolution plotting on the receiver side.
- **Reversed byte order**: The DWORD is placed into the first four data bytes in reversed order (byte 3, 2, 1, 0), which matches the little-endian memory layout used by many CAN visualization tools.
- **Parameterizable sine wave**: The generator allows configuration of period (`PT`), amplitude (`AM`), offset (`OS`), and dead time (`DL`), making the block adaptable to various test and debug scenarios.
- **Deterministic event chain**: The functional flow is strictly event-driven, ensuring that each CAN message is only generated when the callback adapter requests a new transmission.
- **Standard CAN message layout**: The output adheres to a common CAN frame structure with priority, data size, and an 8-byte payload, making it compatible with standard CAN analysis tools.

State Overview

The subapplication is essentially an event-driven pipeline with no internal persistent state beyond the standard function block request/confirmation handshakes. The operational sequence follows a linear state progression:

1. **Idle** – Waiting for a request from the callback adapter (`CallbackFB` idle).
2. **Signal Generation** – On request, `GEN_SIN` is activated and produces the sine output.

3. **Conversion** – The REAL value is converted to DWORD.
4. **Byte Packing** – The DWORD is transformed into the reversed byte array and subsequently wrapped into the CAN message structure.
5. **Transmission** – The completed message is forwarded to the callback adapter for transmission on the PCAN bus, after which the block returns to the idle state.

There are no error or exceptional states because the conversion and packing operations are deterministic and do not depend on external conditions other than the availability of the callback adapter.

Application Scenarios

- **PCAN Explorer visualization**: The block provides a continuous, high-precision sine stream that can be directly plotted in PCAN Explorer, allowing engineers to monitor signal quality and timing without additional hardware.
- **CAN bus testing and validation**: As a diagnostic block, it can inject a known sinusoidal signal into a CAN network to verify the correct operation of receivers, data logging, or filter configurations.
- **Signal generator for simulation**: When developing or debugging CAN-based control systems, this block serves as a stable and repeatable signal source, replacing manual or external signal generators.
- **End-to-end data integrity checks**: By comparing the received sine values with the expected parameters (period, amplitude, offset), users can confirm that the CAN transmission chain preserves data integrity.

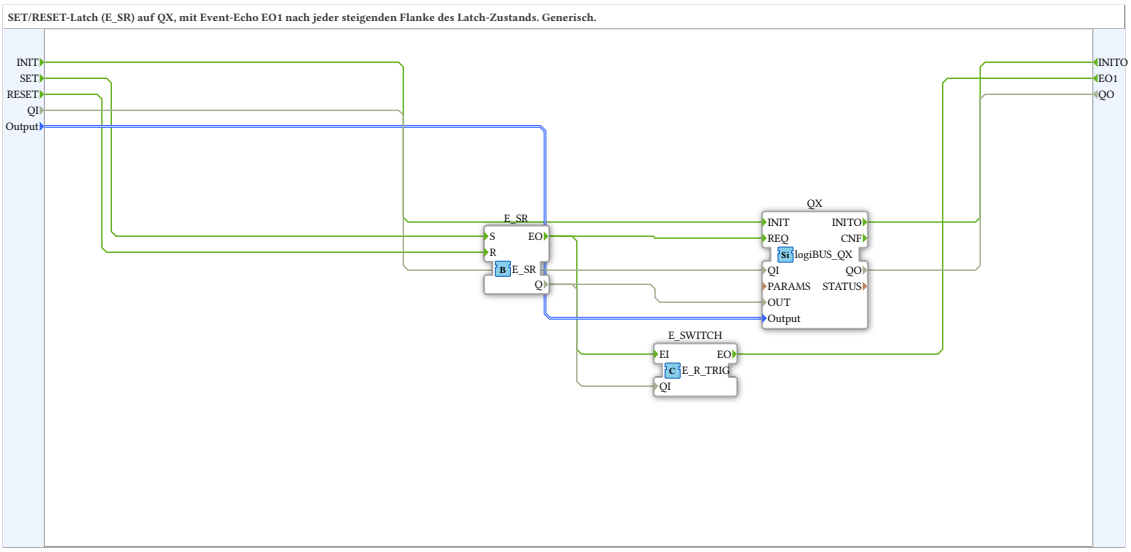
Comparison with Similar Blocks

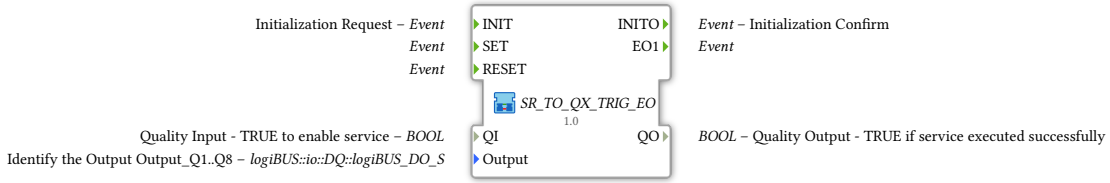
Feature	SIN_TO_PCAN_Callback_DWord	Direct GEN_SIN to CAN (without conversion)	SIN_TO_PCAN with 16-bit precision
Signal precision	Full DWORD (32-bit IEEE 754)	Not directly possible (type mismatch)	Reduced to 16-bit, loses precision
Byte ordering handled internally	Yes (reversed, little-endian)	Manual or external handling required	Manual handling required
Configuration effort	Low (parameters preset in block)	High (multiple conversion steps needed)	Moderate (additional scaling required)
Suitability for high-resolution plotting	Excellent	Not applicable	Limited for values with many decimal places
Dependencies on external functions	Only the callback adapter	Requires additional custom conversion FB	Requires scaling and type conversion FB

Conclusion

The **SIN_TO_PCAN_Callback_DWord** subapplication is a robust and practical solution for generating and transmitting sinusoidal CAN signals with full numerical precision. By integrating the sine generator, REAL-to-DWORD conversion, byte-order reversal, and message assembly into a single encapsulated block, it significantly reduces development effort and improves reliability. Its event-driven design ensures deterministic message generation in response to callback requests, while the parameterized sine configuration offers flexibility across a wide range of diagnostic and testing applications. This block is particularly well suited for engineers working with PCAN Explorer and similar CAN analysis tools who require a dependable, high-resolution signal source.

SR_TO_QX_TRIG_EO





• — ○ • — ○ • — ○ •

Introduction

The **SR_TO_QX_TRIG_EO** subapplication combines a set/reset latch (E_SR) with a digital output (logiBUS_QX) and a rising-edge trigger (E_R_TRIG). It provides a reusable, event-controlled digital output stage that emits an echo event (EO1) whenever the latched output state transitions from false to true. The block is designed as a generic solution: the target physical output (Q1…Q8) is selected via a dedicated data input.

Interface Structure

Event Inputs

Name	Description
`INIT`	Initialization Request (`EInit`); passed on to `QX.INIT`.
`SET`	Event input that sets the internal latch, driving the output Q to `TRUE`.
`RESET`	Event input that resets the internal latch, driving the output Q to `FALSE`.

Event Outputs

Name	Description
`INITO`	Initialization Confirm (`EInit`); passed through from `QX.INITO`.
`EO1`	Echo event emitted on the rising edge of the latch output (transition `FALSE` ⟶ `TRUE`).

Data Inputs

Name	Type	Initial Value	Description
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE enables the service; passed on to `QX.QI`.

Output	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Identifies the physical digital output to be driven (Output_Q1...Q8).

Data Outputs

Name	Type	Description
QO	BOOL	Quality Output - TRUE if the service executed successfully; passed through from QX.QO.

Adapters

None.

Functionality

The subapplication implements a classic latch-and-drive pattern:

1. On SET, the internal E_SR latch is set (Q = TRUE). On RESET, the latch is cleared (Q = FALSE).
2. After every latch update (whether SET or RESET), the Q value is transferred to the logiBUS_QX digital output block, which writes the value to the physical output specified by the Output data input.
3. The same Q signal is also fed to a rising-edge detector (E_R_TRIG). When Q changes from FALSE to TRUE, the detector fires the EO1 event output.

INIT is passed on to QX.INIT, QX.INITO is output as INITO, QI is forwarded to QX.QI and QX.QO is output as QO.

Thus, EO1 acts as an acknowledgment that the latch has been turned on, while the physical output is refreshed on every state change.

Technical Features

- **Event-driven Design** – No cyclic polling; all operations are triggered by events.
- **Generic Output Selection** – The Output data input allows mapping to any logiBUS DQ channel (Q1...Q8) without modification.
- **Rising-Edge Echo** – EO1 only fires when the latched output turns on, providing a clean positive-transition notification.
- **Integrated Output Stage** – Combines latch, physical output, and edge detection in a single reusable unit.
- **Non-Blocking** – Suitable for distributed control applications following the IEC 61499 execution model.

State Overview

The internal latch has two stable states:

State	Output Q	EO1 Emission
Reset (default)	FALSE	No
Set	TRUE	Emitted on rise to TRUE

The EO1 event is only produced during the FALSE ⟶ TRUE transition; RESET operations do not trigger EO1.

Application Scenarios

- **Discrete Output Control** – Switching a digital actuator (e.g., valve, lamp, motor starter) on and off via events.
- **Acknowledgment Signaling** – Providing a confirmation event to a supervisory controller when the output has been successfully switched on.
- **Reusable Library Component** – The block can be instantiated multiple times in a system, each bound to a different output channel.
- **Training / Educational Use** – Suitable for IEC 61499 and 4diac demonstrations of latch, edge detection, and output mapping concepts.

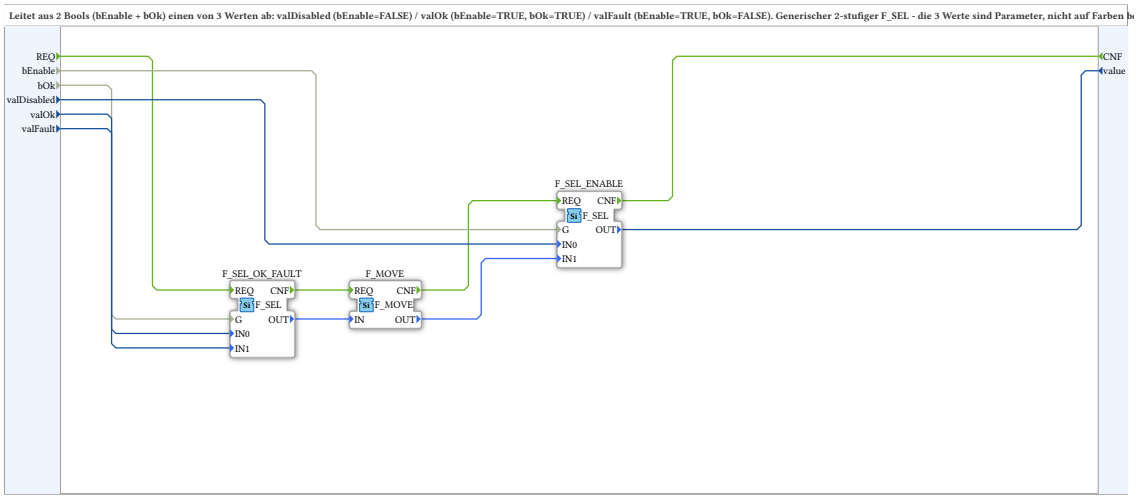
Comparison with Similar Blocks

Block	Key Difference
E_SR (standalone)	Only provides latch functionality without integrated output or echo.
E_RS	Uses a reset-dominant latch behavior; no rising-edge echo or output stage.
logiBUS_QX (direct)	Writes an output directly but lacks latch memory and event echo.
SR_TO_QX_TRIG_EO	Combines latch, physical output update, and rising-edge echo in one block; self-contained and reusable.

Conclusion

The **SR_TO_QX_TRIG_EO** subapplication is a compact, reusable building block for event-based digital output control with latch semantics and a rising-edge acknowledgment event. Its generic output selection makes it flexible for a wide range of logiBUS-based applications, while its clean event interface integrates seamlessly into IEC 61499 systems. By bundling latch, output driver, and edge detection, it reduces wiring effort and improves maintainability.

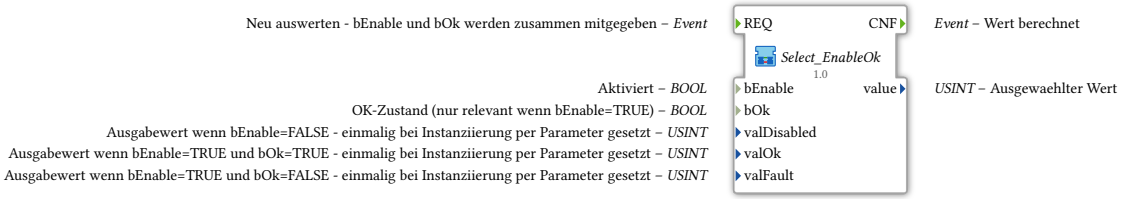
Select_EnableOk



• — ○ • — ○ • — ○ •

Introduction

`Select_EnableOk` derives one of 3 parameterized USINT values from 2 bools (`bEnable`, `bOk`): `valDisabled` (bEnable=FALSE), `valOk` (bEnable=TRUE and bOk=TRUE), `valFault` (bEnable=TRUE and bOk=FALSE). A generic 2-stage `F_SEL`, not limited to colors - per the source comment, extracted from `Q_BackgroundColour_EnableOk` to decouple the pure selection logic from the VT background-color application.



Function blocks used

- **F_SEL_OK_FAULT** (`iec61131::selection::F_SEL`): selects between `valFault`/`valOk` based on `bOk`.
- **F_MOVE** (`iec61131::selection::F_MOVE`, `DataType=USINT`): passes the intermediate value through unchanged (sequencing the two selection stages).
- **F_SEL_ENABLE** (`iec61131::selection::F_SEL`): selects between `valDisabled`/intermediate result based on `bEnable`.

Program flow and connections

`REQ -> F_SEL_OK_FAULT.REQ -> F_MOVE.REQ -> F_SEL_ENABLE.REQ -> CNF`. On the data side: `bOk`/`valFault`/`valOk` -> `F_SEL_OK_FAULT` -> `F_MOVE` -> `F_SEL_ENABLE.IN1`, in parallel `bEnable`/`valDisabled` -> `F_SEL_ENABLE.IN0`/`.G` -> `value`.

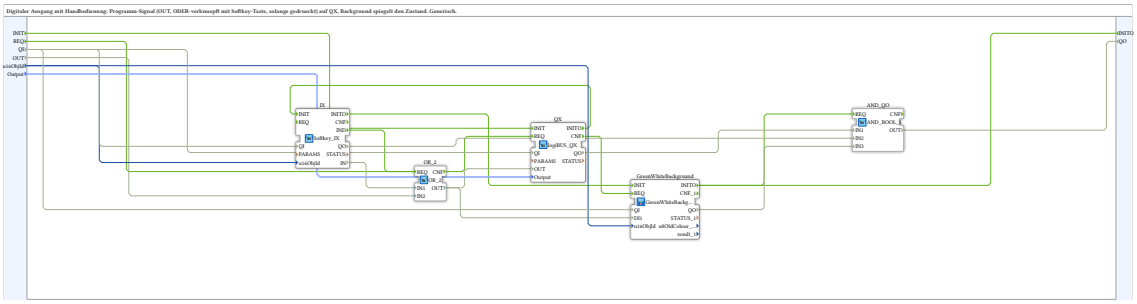
Summary

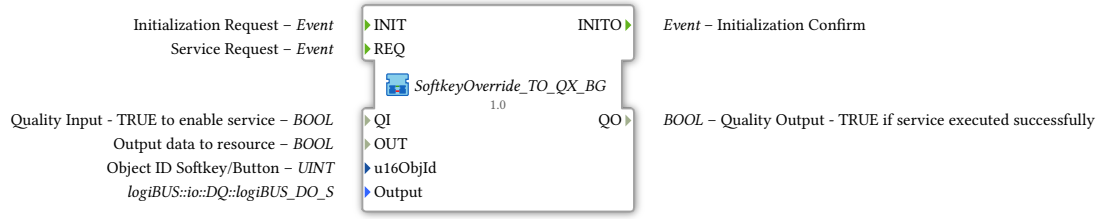
Generic 2-bool-to-1-of-3-values selection, reusable beyond background colors (e.g. for any enable/ok status display).

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SoftkeyOverride_TO_QX_BG





• – ○ • – ○ • – ○ •

Introduction

`SoftkeyOverride_TO_QX_BG` is a reusable IEC 61499 subapplication that combines a program signal with a manual softkey input and writes the result to a digital output of type `logiBUS_QX`. It is designed for digital outputs that need both automatic program control and manual hand operation. The OR combination ensures that the output can be forced active while the softkey is pressed, even if the program signal is inactive. A background visualization subapplication mirrors the output state for HMI purposes.

Interface Structure

Event Inputs

Name	Type	Comment
`INIT`	EInit	Initialization request; starts the chain `QX` ⟶ `IX` ⟶ `GreenWhiteBackground`.
`REQ`	Event	Service Request; triggers evaluation and output update.

Event Outputs

Name	Type	Comment
`INITO`	EInit	Initialization confirmation; comes from `GreenWhiteBackground.INITO`.

There are no other event outputs.

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	TRUE	Quality input, service enable; passed on to QX, IX and GreenWhiteBackground.
OUT	BOOL	—	Output data to resource.
u16ObjId	UINT	ID_NULL	Object ID Softkey/Button.
Output	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Selection of the physical/logical digital output.

Data Outputs

Name	Type	Comment
QO	BOOL	Quality output, AND of the QO outputs of QX, IX and GreenWhiteBackground (via AND_QO, AND_BOOL_3).

The actual output goes through the embedded QX block to the physical channel.

Adapters

There are no adapters.

Functionality

The subapplication implements a digital output stage with manual override capability:

- The program signal OUT is combined with the softkey input IX.IN through an OR_2 function block.
- The softkey input is read by an internal Softkey_IX block, which delivers the current key state and emits an event when the key state changes.
- The external REQ event also triggers the OR evaluation, so the output can be updated cyclically or on demand.
- The result of the OR operation is written to the QX digital output.
- The same result is sent to an internal GreenWhiteBackground subapplication, so the HMI background reflects the current output state.
- The QX.CNF event triggers the background update after the output has been written successfully.
- Initialization and quality: INIT → QX.INIT; QX.INITO → IX.INIT; IX.INITO → GreenWhiteBackground.INIT; GreenWhiteBackground.INITO → INITO and → AND_QO.REQ. QI → QX.QI, IX.QI and GreenWhiteBackground.QI (no fixed QI=TRUE any more on QX and IX); QX.QO → AND_QO.IN1, IX.QO → AND_QO.IN2, GreenWhiteBackground.QO → AND_QO.IN3; AND_QO.OUT → QO.

The behavior is purely combinatorial with respect to the data values: the output is active if either the program signal OUT is true OR the softkey is currently pressed. There is no latching or memory.

Technical Features

- Generic design: the same subapplication can be reused with different softkey object IDs and different digital output channels.
- Parameterizable softkey address via u16ObjId.
- Parameterizable output channel via Output.
- Supports event-driven processing through REQ.
- The softkey block can trigger processing independently via its internal event output.
- Output and background visualization are updated consistently.
- Uses standard logic block OR_2 from IEC 61131 bitwise operators.
- The subapplication is packaged as an IEC 61499 subapplication type and can be embedded in larger applications.

State Overview

The subapplication has no internal state machine, but the output state can be described by the following combinations:

OUT	Softkey pressed	QX output	Background
FALSE	FALSE	FALSE	Inactive
TRUE	FALSE	TRUE	Active
FALSE	TRUE	TRUE	Active
TRUE	TRUE	TRUE	Active

The manual softkey therefore acts as an “on” override. It can force the output to be active while pressed, but it cannot force the output off when the program signal is active.

Application Scenarios

Typical use cases include:

- Digital actuators that must be operable both from a PLC program and from a local HMI softkey.
- Manual test or service functions for machine outputs.
- Lamp, relay, or valve control with operator override.
- HMI background elements that visualize whether an output is active.
- Generic output stages where the same logic must be reused for multiple channels.

Comparison with Similar Blocks

Compared with a simple direct connection from `OUT` to a digital output, this subapplication adds:

- Manual softkey override capability.
- Event-driven softkey processing.
- HMI background mirroring.

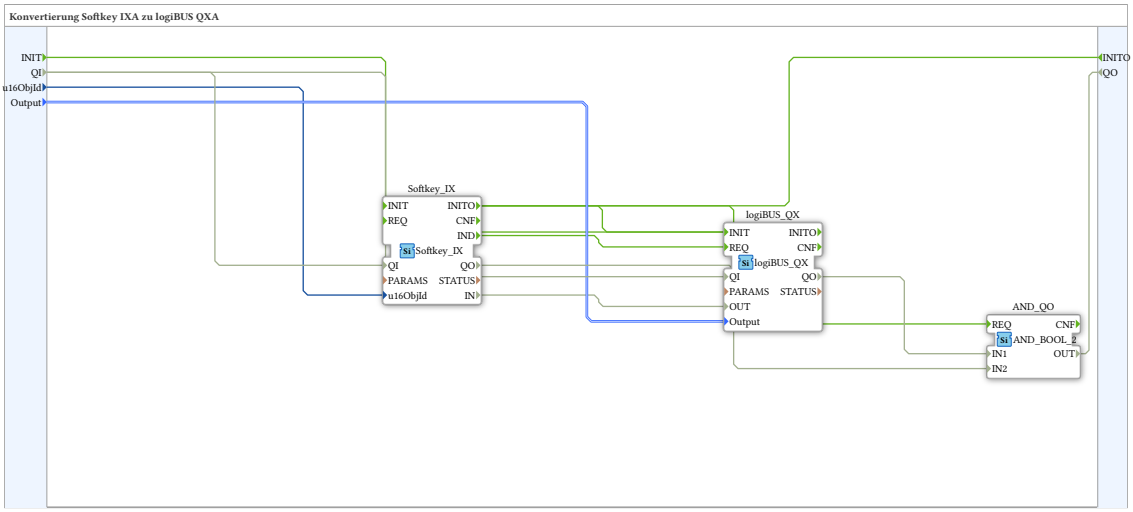
Compared with a flip-flop or latch-based manual override, this subapplication does not store the manual command. The output remains active only while the softkey is pressed. When the softkey is released and `OUT` is false, the output returns to false.

Compared with an AND-based inhibit logic, this subapplication uses an OR combination, so it supports forced activation rather than forced deactivation. If manual off is required, a different logic structure would be needed.

Conclusion

`SoftkeyOverride_TO_QX_BG` is a compact and reusable subapplication for digital outputs requiring both program control and manual hand operation. It combines a softkey input with a program signal using OR logic, writes the result to a `logiBUS_QX` output, and mirrors the state to an HMI background. Its generic parameters make it suitable for many output channels and softkey configurations in IEC 61499-based systems.

Softkey_IX_TO_logiBUS_QX



• – ○ • – ○ • – ○ •

Introduction

`Softkey_IX_TO_logiBUS_QX` is the test_B counterpart to `Softkey_IXA_TO_logiBUS_QXA` — a VT softkey (`Softkey_IX`) directly switches a physical output (`logiBUS_QX`), without adapters.



Function Blocks (FBs) Used

Sub-blocks: Softkey_IX_TO_logiBUS_QX

- **Type**: SubAppType
- **Internal FBs used**:
  - **Softkey_IX**: `isobus::UT::io::Softkey::Softkey_IX` — VT softkey (non-adapter variant).
  - **logiBUS_QX**: `logiBUS::io::DQ::logiBUS_QX` — physical digital output.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — combines `logiBUS_QX.QO` and `Softkey_IX.QO` into the output `QO`.
- **Functionality**: `Softkey_IX.IND` triggers `logiBUS_QX.REQ`; `Softkey_IX.IN` is wired directly to `logiBUS_QX.OUT`. The SubApp interface additionally has the events `INIT`/`INITO` as well as `QI` (BOOL, initial value `TRUE`) and `QO` (BOOL).

Program Flow and Connections

1. `u16ObjId` ⟶ `Softkey_IX.u16ObjId`; `Output` ⟶ `logiBUS_QX.Output`.
2. `Softkey_IX.IND` ⟶ `logiBUS_QX.REQ`; `Softkey_IX.IN` ⟶ `logiBUS_QX.OUT`.
3. Initialization chain: `INIT` ⟶ `logiBUS_QX.INIT`, `logiBUS_QX.INITO` ⟶ `Softkey_IX.INIT`, `Softkey_IX.INITO` ⟶ `INITO` and `AND_QO.REQ`.
4. Quality: `QI` ⟶ `logiBUS_QX.QI` and `Softkey_IX.QI`; `logiBUS_QX.QO` ⟶ `AND_QO.IN1`; `Softkey_IX.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Application Scenarios

- Minimal softkey-to-output block for test_B.

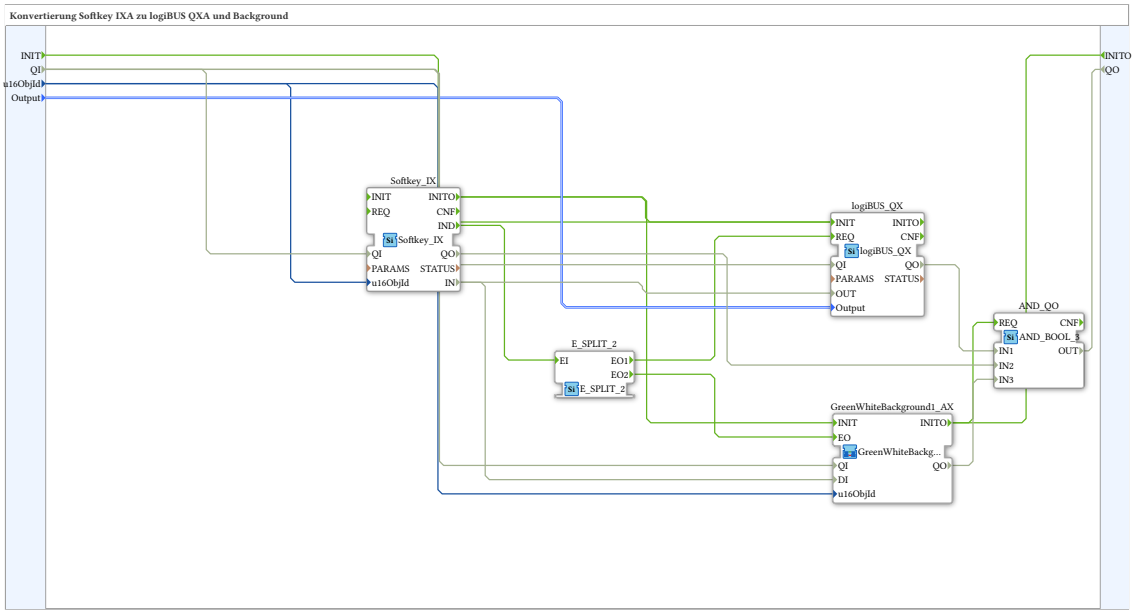
Summary

test_B counterpart to `Softkey_IXA_TO_logiBUS_QXA`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

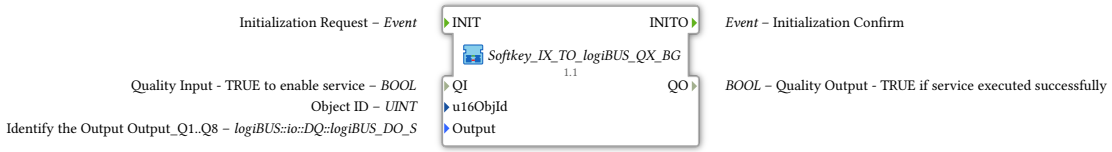
Softkey_IX_TO_logiBUS_QX_BG



• – ○ • – ○ • – ○ •

Introduction

`Softkey_IX_TO_logiBUS_QX_BG` extends `Softkey_IX_TO_logiBUS_QX` with a VT status color — the test_B counterpart to `Softkey_IXA_TO_logiBUS_QXA_BG`.



Function Blocks (FBs) Used

Sub-blocks: Softkey_IX_TO_logiBUS_QX_BG

- **Type**: SubAppType
- **Internal FBs used**:
  - **Softkey_IX**: `isobus::UT::io::Softkey::Softkey_IX` — VT softkey.
  - **logiBUS_QX**: `logiBUS::io::DQ::logiBUS_QX` — physical digital output.
  - **E_SPLIT_2**: `iec61499::events::E_SPLIT_2` — splits the softkey event.
  - **Color block** (compact background variant, see Background Color Blocks): sets the status color.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_3` — combines the `QO` of `logiBUS_QX`, `Softkey_IX` and the color block.
- **Functionality**: `Softkey_IX.IND` is distributed via `E_SPLIT_2` to `logiBUS_QX.REQ` and the color block; `Softkey_IX.IN` feeds `logiBUS_QX.OUT` and the color block.

Program Flow and Connections

1. `u16ObjId` ⟶ `Softkey_IX.u16ObjId` and the color block's `.u16ObjId`; `Output` ⟶ `logiBUS_QX.Output`.
2. `Softkey_IX.IND` ⟶ `E_SPLIT_2.EI` ⟶ `EO1` ⟶ `logiBUS_QX.REQ`, `EO2` ⟶ color block `.EO`.
3. `Softkey_IX.IN` ⟶ `logiBUS_QX.OUT` and ⟶ color block `.DI`.
4. Initialization: event input `INIT` ⟶ `logiBUS_QX.INIT`, `logiBUS_QX.INITO` ⟶ `Softkey_IX.INIT`, `Softkey_IX.INITO` ⟶ color block `.INIT` (instance `GreenWhiteBackground1_AX`), whose `INITO` ⟶ event output `INITO` and ⟶ `AND_QO.REQ`.
5. Quality: data input `QI` (`BOOL`, default `TRUE`) ⟶ `logiBUS_QX.QI`, `Softkey_IX.QI` and color block `.QI`; `logiBUS_QX.QO` ⟶ `AND_QO.IN1`, `Softkey_IX.QO` ⟶ `AND_QO.IN2`, color block `.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ data output `QO`.

Application Scenarios

- VT softkey with visual status feedback for test_B, without OPC-UA.

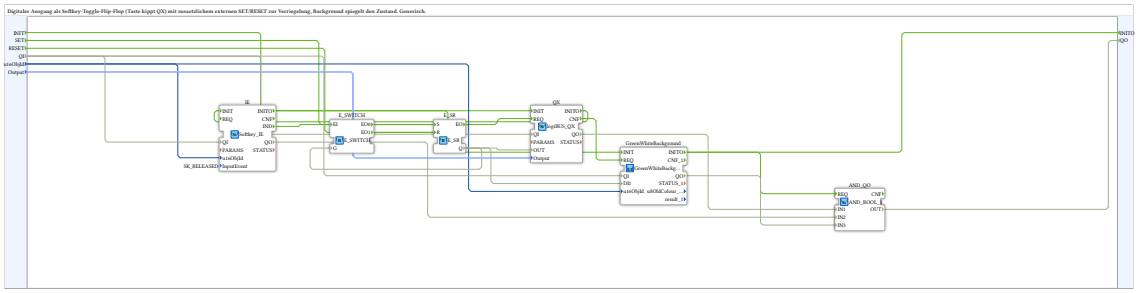
Summary

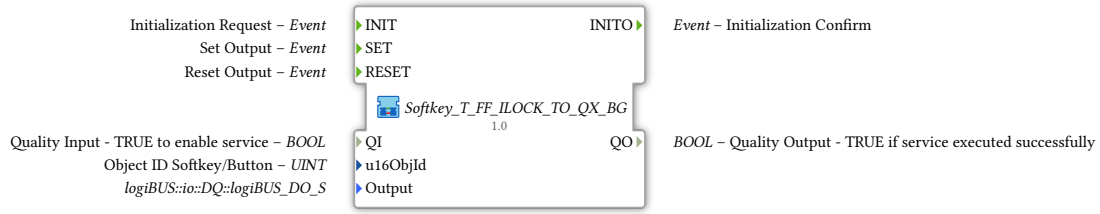
test_B counterpart to `Softkey_IXA_TO_logiBUS_QXA_BG`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Softkey_T_FF_ILOCK_TO_QX_BG





• — ○ • — ○ • — ○ •

Introduction

Softkey_T_FF_ILOCK_TO_QX_BG is a generic subapplication that implements a digital output driven by a softkey (button) toggle flip-flop. Each released key press toggles the state of a logiBUS_QX digital output. In addition, external SET and RESET event inputs allow overriding or locking the flip-flop independently of the key. The internal background visualization reflects the current output state (green/white), providing an immediate visual feedback in the HMI.

Interface Structure

Event Inputs

Name	Description
INIT	Initialization request (`EInit`)
SET	Sets the internal flip-flop output to TRUE
RESET	Resets the internal flip-flop output to FALSE

Event Outputs

Name	Description
INITO	Initialization confirmation (`EInit`, from `GreenWhiteBackground.INITO`)

Data Inputs

Name	Type	Initial Value	Description

QI	BOOL	TRUE	Quality Input, TRUE enables the service
u16ObjId	UINT	ID_NULL	Object ID of the softkey/button
Output	logiBUS::io::DQ::logiBUS_DO_S	logiBUS_DO::Invalid	Digital output configuration (type/address)

Data Outputs

Name	Type	Description
QO	BOOL	Quality Output, AND of `QX.QO`, `IE.QO` and `GreenWhiteBackground.QO` (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_3`)

Adapters

None.

Functionality

The subapplication combines a softkey input event handler, a Set/Reset flip-flop and a digital output function block:

1. The softkey event block (Softkey_IE) monitors the assigned softkey (identified by `u16ObjId`) and generates an event on every key release (`SK_RELEASED`).
2. The E_SWITCH block routes this event either to the Set input or to the Reset input of the E_SR flip-flop, depending on the current state `Q` (routed to the switch control input `G`):
   - If `Q = FALSE` (`G = FALSE`), the event is routed to `E_SR.S` ⟶ the output becomes TRUE.
   - If `Q = TRUE` (`G = TRUE`), the event is routed to `E_SR.R` ⟶ the output becomes FALSE. This yields a toggle behavior: every softkey release inverts the flip-flop state.
3. The external event inputs `SET` and `RESET` are connected directly to the `S` and `R` inputs of the E_SR flip-flop. They take precedence over the key events and can be used to lock the output to a defined state or to override the toggle.
4. The flip-flop output `Q` drives the `QX` function block, which writes the digital output (e.g., logiBUS output module). The value `Output` (data input) defines the output module and channel configuration.
5. The `GreenWhiteBackground` subapplication is triggered after every completed output write (`QX.CNF`) and receives both the object ID and the current state `Q` as `DI1`, so the background color indicates the output state (green = active, white = inactive).
6. `QI` is distributed to `QX.QI`, `IE.QI` and `GreenWhiteBackground.QI` (no longer hard-wired to `TRUE`). Initialization is chained: `INIT` ⟶ `QX.INIT`; `QX.INITO` ⟶ `IE.INIT`; `IE.INITO` ⟶ `GreenWhiteBackground.INIT`; `GreenWhiteBackground.INITO` ⟶ `INITO` (and ⟶ `AND_QO.REQ`). `QX.QO` ⟶ `AND_QO.IN1`; `IE.QO` ⟶ `AND_QO.IN2`; `GreenWhiteBackground.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`.

Technical Features

- Generic design: works with any softkey object ID and any logiBUS digital output configuration.
- Toggle flip-flop based on edge-triggered softkey release events, debounce handled by the softkey driver.
- External SET/RESET override inputs allow interlocking or forced control.
- Integrated background visualization (green/white) for direct HMI feedback.
- Uses standard 61499 blocks (E_SR, E_SWITCH) and logiBUS function blocks (logiBUS_QX, Softkey_IE).
- Combines event-driven logic with a synchronous data flow for state propagation.

State Overview

The E_SR flip-flop defines the central state:

State	Output QX	Background	Trigger to leave state
FALSE	inactive	white	Softkey release (toggle) or external SET
TRUE	active	green	Softkey release (toggle) or external RESET

The E_SWITCH uses the current state `Q` as its switching signal `G`, ensuring the correct event path for the next softkey press.

Application Scenarios

- **Machine operator panel**: A physical softkey toggles a digital output (e.g., lamp, valve, motor) while a PLC program can lock the output via SET/RESET during safety conditions.
- **Manual/Auto override**: The output can be forced to a defined state by the control system even if the operator presses the key.
- **Visual feedback on HMI**: The background of the corresponding HMI element turns green when the output is active, making the state visible at a glance.
- **Reusable subapplication**: Can be instantiated multiple times for different softkeys and output channels within the same project.

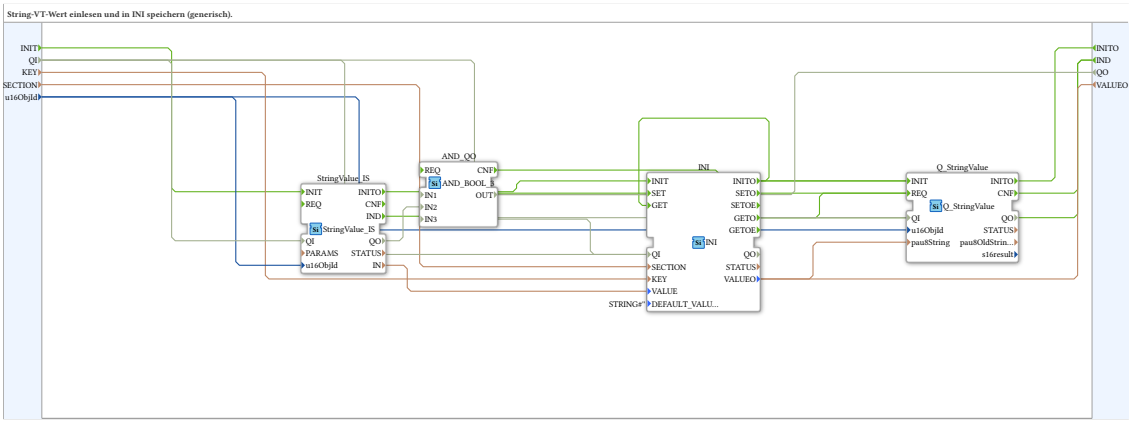
Comparison with Similar Blocks

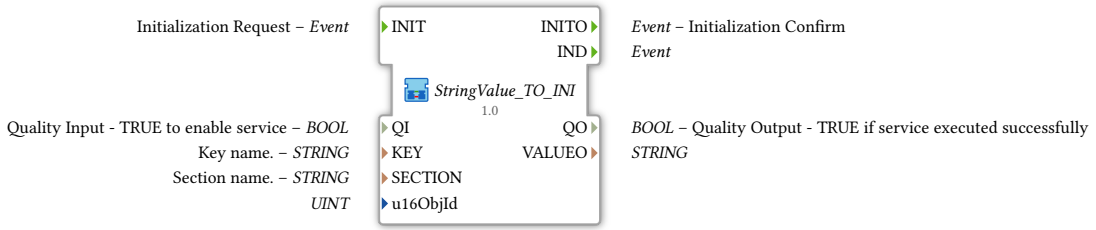
Feature	Softkey_T_FF_ILOCK_TO_QX_BG	Simple Softkey-to-QX (without flip-flop)	Softkey with separate Set/Reset inputs
Toggle on key press	Yes (flip-flop toggles)	No (output follows key)	No
External override	Yes (SET/RESET events)	No	Yes
Background visualization	Yes (green/white)	Optional	Optional
Generic object ID	Yes (`u16ObjId`)	Yes	Yes

Conclusion

Softkey_T_FF_ILOCK_TO_QX_BG is a versatile, generic subapplication that combines a toggle flip-flop softkey logic with external override capability and integrated background visualization. It offers a clean separation between operator input, interlocking logic, and output driving, making it suitable for a wide range of control and visualization tasks in automation systems based on 4diac/logiBUS.

StringValue_TO_INI





• – ○ • – ○ • – ○ •

Introduction

The StringValue_TO_INI subapplication is a generic IEC 61499 component that reads a string value from an ISOBUS Virtual Terminal (VT) object and persistently stores it in an INI-based storage system. It

provides both storage and retrieval functionality, making it suitable for scenarios where VT input values need to be persisted across sessions or used by other parts of a control application.

Interface Structure

Event Inputs

Event	Description
INIT	Initialization request (EInit); starts the initialization chain `StringValue_IS` ⟶ `INI` ⟶ `Q_StringValue`.

Event Outputs

Event	Description
INITO	Initialization confirm (EInit); comes from `Q_StringValue.INITO` (end of the chain).
IND	Indicates that a SET or GET operation on the INI storage has completed successfully.

Data Inputs

Data	Type	Description
QI	BOOL	Quality Input - TRUE to enable service (initial value TRUE); distributed to StringValue_IS.QI, INI.QI and Q_StringValue.QI.
KEY	STRING	The key name under which the value is stored in the INI file.
SECTION	STRING	The section name in the INI file where the key resides.
u16ObjId	UINT	The object identifier of the string object within the ISOBUS Virtual Terminal. Initialized to ID_NULL.

Data Outputs

Data	Type	Description
QO	BOOL	Quality Output - TRUE if the service executed successfully; AND combination (`AND_QO`, `AND_BOOL_3`) of Q_StringValue.QO, StringValue_IS.QO and INI.QO.
VALUEO	STRING	The string value read from the INI storage after a GET operation.

Adapters

None.

Functionality

The StringValue_TO_INI subapplication combines three functional building blocks (plus an AND block `AND_QO`, `iec61131::booleanOperators::AND_BOOL_3`, for the quality output) to perform the following operations:

1. **Read from VT**: The internal StringValue_IS FB reads a string value from the ISOBUS VT object specified by u16ObjId. When this value is available, it triggers the INI SET operation.
2. **Store in INI**: The internal INI FB stores the received string value under the KEY within the SECTION of the INI storage. This allows the value to be persisted and retrieved later.
3. **Retrieve and Queue**: On initialization, the subapplication automatically performs a GET operation to read the stored value. This value is then output via VALUEO and also forwarded to the internal Q_StringValue queue for further processing.

Initialization: `INIT` ⟶ `StringValue_IS.INIT`; `StringValue_IS.INITO` ⟶ `INI.INIT`; `INI.INITO` ⟶ `Q_StringValue.INIT`; `Q_StringValue.INITO` ⟶ `INITO` and ⟶ `AND_QO.REQ`. Quality: `QI` is distributed to `StringValue_IS.QI` and `INI.QI` (and `Q_StringValue.QI`); `Q_StringValue.QO` ⟶ `AND_QO.IN1`,

`StringValue_IS.QO` ⟶ `AND_QO.IN2`, `INI.QO` ⟶ `AND_QO.IN3`; `AND_QO.OUT` ⟶ `QO`. The internal `GET` trigger after `INI.INITO` is unchanged.

The event flow is as follows:

- When the INI storage initializes, a GET operation is automatically triggered.
- Upon completion of a GET operation, the retrieved value appears at VALUEO and is queued via Q_StringValue, and the IND event is emitted.
- When a VT string value is read (StringValue_IS.IND), it is written to the INI storage via a SET operation, and the IND event is emitted upon completion.

Technical Features

- **Generic INI Storage**: Uses the eclipse4diac INI storage FB, making the storage format compatible with standard INI file conventions.
- **ISOBUS VT Integration**: Reads string values from ISOBUS Virtual Terminal objects using the isobus::UT::io::StringValue FB, enabling interaction with operator terminals.
- **Automatic Initialization**: The subapplication performs an initial GET operation during startup to load the persisted value.
- **Queue Support**: The internal Q_StringValue block queues string values, allowing for ordered processing in downstream logic.
- **Configurable Keys**: Both KEY and SECTION are configurable data inputs, making the subapplication reusable for different INI entries.
- **Default Value Handling**: The INI FB uses a default empty string when no value exists in storage.

State Overview

The subapplication exhibits the following operational states:

State	Description
Initialization	During startup, the INI storage initializes. The INITO output triggers a GET operation to retrieve the stored value.
GET Operation	Retrieves the string value associated with KEY/SECTION from the INI storage. Upon completion (GETO), the value is output and queued, and the IND event is issued.
SET Operation	Stores a new string value received from the VT object into INI storage. Triggered by StringValue_IS.IND. Upon completion (SETO), the IND event is issued.
Idle	Waiting for either a VT value to arrive (triggering SET) or for external requests that cause further GET operations.

Application Scenarios

- **Machine Operator Preferences**: Store operator-entered string values (e.g., machine names, batch identifiers) from a VT terminal into persistent INI storage.
- **Configuration Persistence**: Save configuration strings entered via a VT terminal so they are retained across power cycles.
- **Data Exchange Between Applications**: Write VT string values to a shared INI file that other applications can read.
- **Queued Processing**: Use the Q_StringValue queue to process multiple string values in order, e.g., for logging or sequential command execution.

Comparison with Similar Blocks

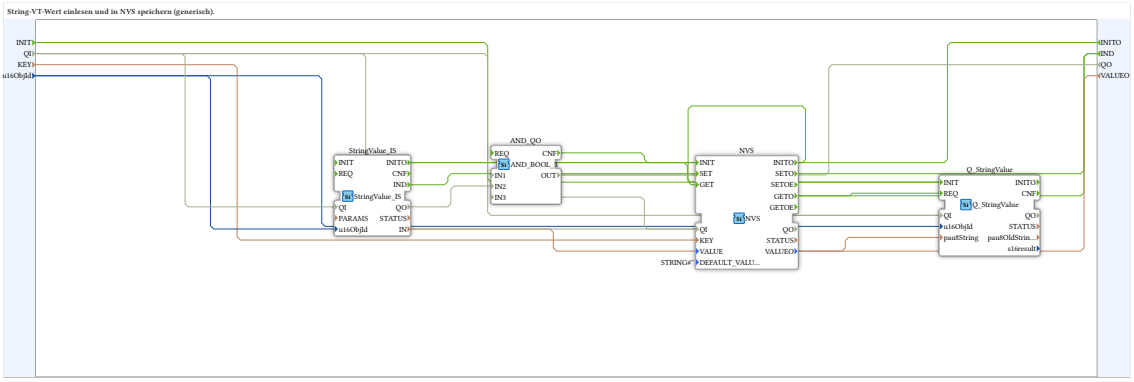
Feature	StringValue_TO_INI	Standard String Store FB	INI Direct Access
VT interface	Yes (reads from ISOBUS VT)	Usually no	No
Persistent storage	Yes (INI file)	Depends on implementation	Yes
Queue support	Yes (Q_StringValue)	No	No
Configurable key/section	Yes	Limited	Yes

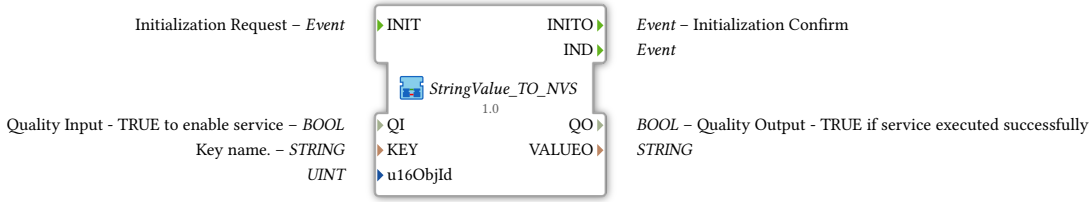
Generic reuse	High	Medium	Low

Conclusion

The StringValue_TO_INI subapplication provides a compact and reusable solution for bridging ISOBUS Virtual Terminal string inputs with persistent INI storage. Its automatic initialization, queue support, and configurable keys make it a versatile building block for industrial automation applications that require both operator interaction and data persistence.

StringValue_TO_NVS





• – ◦ • – ◦ • – ◦ •

Introduction

The **StringValue_TO_NVS** subapplication provides a generic solution for reading a String-VT value and persisting it into the non-volatile storage (NVS) of an ESP32. It combines the `isobus::UT::io::StringValue::StringValue_IS` function block for input acquisition with the `logiBUS::storage::esp32_nvs::NVS` function block for durable storage. The subapplication also exposes the stored value on its output interface. It was extracted from a larger exercise project to be reused as a standalone, generic component.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization request; starts the initialization chain (`Q_StringValue` ⟶ `StringValue_IS` ⟶ `NVS`).

Event Outputs

- **INITO** (EInit): Initialization confirm from `NVS.INITO` (end of the initialization chain).
- **IND** (Event): Emitted after a read or write operation completes, indicating that a new value has been stored or retrieved.

Data Inputs

- **QI** (BOOL): Quality Input - TRUE to enable service (initial value `TRUE`); passed through to `Q_StringValue.QI`, `StringValue_IS.QI` and `NVS.QI`.
- **KEY** (STRING): The key name under which the value is stored in the NVS.
- **u16ObjId** (UINT): The object identifier for the String-VT value. Defaults to `ID_NULL`.

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if the service executed successfully; AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_3`) of the `QO` of `Q_StringValue`, `StringValue_IS` and `NVS`.
- **VALUEO** (STRING): The current string value that has been read from the NVS or acquired from the input.

Adapters

The subapplication does not expose any adapter interfaces.

Functionality

The subapplication operates as follows. In addition, the initialization chain runs `INIT` → `Q_StringValue.INIT`, `Q_StringValue.INITO` → `StringValue_IS.INIT`, `StringValue_IS.INITO` → `NVS.INIT`, `NVS.INITO` → `INITO` and → `AND_QO.REQ`; `QI` goes to `Q_StringValue.QI`, `StringValue_IS.QI` and `NVS.QI`, and their `QO` outputs are ANDed by the `AND_QO` block (`AND_BOOL_3`) into `QO`.

1. **Initialization**: On startup, the internal `NVS` function block performs its initialization. The resulting `INITO` event is immediately routed to the `GET` event of the same block, causing the currently stored value to be read from the NVS.
2. **Value acquisition**: The `StringValue_IS` function block acquires a String-VT value based on the provided `u16ObjId`. When a new value is available, its `IND` output event triggers the `NVS.SET` event.
3. **Storage**: The acquired string value (from `StringValue_IS.IN`) is written to the NVS under the configured `KEY`. After a successful write operation (`NVS.SETO`), the subapplication emits the `IND` output event to signal completion.
4. **Read/Query**: The `GETO` event from the `NVS` block is forwarded to the `Q_StringValue` query function block, which converts the internal representation (`pau8String`) into a proper String-VT format. Simultaneously, `GETO` is also routed to the subapplication's `IND` output.
5. **Output**: The `VALUEO` output continuously reflects the value stored in the NVS. It is updated after each `SET` or `GET` operation.

Technical Features

- **Generic NVS key**: The storage key is configurable via the `KEY` input.
- **Object ID binding**: The `u16ObjId` input selects the correct String-VT object in the ISOBUS context.
- **Automatic initialization**: On startup, the NVS block is initialized and the stored value is read automatically without external triggering.
- **Default value handling**: The NVS block is configured with an empty default value (`STRING#''`) for the first-time retrieval.
- **Event acknowledgment**: The `IND` event output signals successful read or write operations.
- **Internal wiring encapsulated**: All event and data connections between the internal function blocks are hidden behind the subapplication interface, simplifying reuse.

State Overview

The internal state machine is primarily managed by the `NVS` function block from the `logiBUS` library. The typical states are:

1. **Initialization (INIT)**: The NVS block initializes the underlying storage backend.
2. **Get (GET)**: Reads the value from the NVS and makes it available on `VALUEO`.
3. **Set (SET)**: Writes the input value to the NVS under the configured key.

The internal event paths are:

- `NVS.INITO → NVS.GET` (auto-read stored value after initialization)
- `StringValue_IS.IND → NVS.SET` (store a newly acquired value)
- `NVS.GETO → Q_StringValue.REQ` and `NVS.GETO → IND` (query and indication)

Application Scenarios

- **IoT device configuration**: Storing sensor values, device names, or user-adjustable parameters that must survive a reboot or power loss.
- **ISOBUS integration**: Working with ISO 11783 String-VT objects to persist display strings, diagnostic messages, or configuration data in agricultural automation systems.
- **Generic reuse**: Because both the key and object ID are parameterizable, the subapplication can be instantiated multiple times within a project for storing different values.

- **Parameter snapshotting**: Saving runtime states or calibration values for later retrieval during development or commissioning.

Comparison with Similar Blocks

Feature	StringValue_TO_NVS	NVS (storage only)	StringValue_IS (input only)
Reads a String-VT value	✅	❌	✅
Stores value in NVS	✅	✅	❌
Exposes stored value as output	✅	✅	❌
Handles initialization automatically	✅	✅ (via INIT)	❌
Provides query response via Q_StringValue	✅	❌	❌
Encapsulates event wiring	✅	❌	❌

Compared to using the `NVS` and `StringValue_IS` function blocks separately, this subapplication bundles the necessary event wiring, initialization sequence, and query logic into a single reusable component, reducing integration effort and the risk of wiring errors.

Conclusion

The `StringValue_TO_NVS` subapplication is a well-structured, generic solution for reading a String-VT value and persisting it in the ESP32 NVS. It simplifies project integration by encapsulating initialization, storage, and query behavior behind a clean interface. The exposed `IND` event and `VALUE0` output make it straightforward to incorporate into larger IEC 61499 applications, and its parameterizable inputs provide the flexibility needed for reuse across different contexts.

SwitchPic(Col) Building Blocks: Common Pattern

• – ◦ • – ◦ • – ◦ •

Introduction

`MyLib::sys` (test_B) contains a family of building blocks that **display a different VT image (object pointer target) and/or a different background color depending on the state** — e.g., a slider/valve animation with the states Unknown/Closed/Opening/Opened/Closing. This page explains the common pattern; the individual pages only list the specific variations.

Naming Scheme

`SwitchPic[Col]_<Zustände>_<Variante>[_aux]`

Component	Meaning
`Col` (optional)	In addition to changing the image, also toggles a background color (`Q_BackgroundColour`) to match the state, not just the image (`Q_NumericValue`).
States (`2` or `5`)	`2` = Boolean selector `DI1` (`up/down`, type `s2x1ObjectIDs/s2x2ObjectIDs`); `5` = `iSTATE`-Selector (`USINT`, slider state machine Unknown/Closed/Opening/Opened/Closing, type `SchieberStruct/SchieberAuxInStruct`), evaluated via `F_MUX_5`.
Variant (`1/2/3`, only with `SwitchPic`)	Number/Type of VT objects updated simultaneously: `1` = only a normal VT object (Softkey/DataMask), `2` = additionally an AUX object, `3` = additionally an AUX object AND a second normal object (“Button”).

_aux	Only AUX object(s) are switched, no normal VT object (counterpart to variant 1, but exclusively for Auxiliary Function objects).

Functionality

1. A structure `pictures/Sets` (type depends on state/variant) contains the corresponding object ID(s) (image and, if applicable, color) for each possible state.
2. `F_MOVE` unpacks this structure into individual values.
3. A multiplexer (`F_SEL` for 2 states, `F_MUX_5` for 5 states, controlled by `DI1` or `iSTATE`) selects the values matching the current state.
4. `Q_NumericValue/Q_NumericValueAux` sets the selected object pointer value to the VT object identified by `Picture/PictureA/PictureB`; for `Col` variants, `Q_BackgroundColour/Q_BackgroundColourAux` additionally sets the background color to the object identified by `Color`.
5. For multiple target objects (variant 2/3), several `Q_NumericValue(Aux)` instances are chained together in a fixed INIT sequence (each triggers the next via `INITO`) before `INITO` is reported externally.

Summary

A multiplexer pattern: State ⟶ Structure lookup (`F_MOVE`) ⟶ Selection (`F_SEL`/`F_MUX_5`) ⟶ one or more `Q_NumericValue(Aux)`/`Q_BackgroundColour(Aux)` targets. The variant number (1/2/3) and `Col`/`_aux` determine only how many targets of which type (normal/AUX, image/color) are updated simultaneously.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

SwitchPic(Col) Blocks: Shared Pattern

• – ◦ • – ◦ • – ◦ •

Introduction

`MyLib::sys` (test_B) contains a family of blocks that **display a different VT picture (object-pointer target) and/or a different background color depending on state** — e.g. a slide-valve animation with the states Unknown/Closed/Opening/Opened/Closing. This page explains the shared pattern; the individual pages only state the concrete differences.

Naming Scheme

`SwitchPic[Col]_<states>_<variant>[_aux]`

Component	Meaning
Col (optional)	In addition to the picture, also switches a background color (`Q_BackgroundColour`) to match the state, not just a picture (`Q_NumericValue`).
States (2 or 5)	`2` = boolean selector `DI1` (up/down, type `s2x1ObjectIDs`/`s2x2ObjectIDs`); `5` = `iSTATE` selector (`USINT`, slide-valve state machine Unknown/Closed/Opening/Opened/Closing, type `SchieberStruct/SchieberAuxInStruct`), evaluated via `F_MUX_5`.
Variant (1/2/3, `SwitchPic` only)	Number/kind of VT objects updated simultaneously: `1`=regular VT object only (softkey/data mask), `2`=plus an AUX object, `3`=plus an AUX object AND a second regular object (“button”).
_aux	Only AUX object(s) are switched, no regular VT object (counterpart to variant 1, but exclusively for auxiliary function objects).

Functionality

1. A structure `pictures/Sets` (type depending on states/variant) holds, for every possible state, the associated object ID(s) (picture and, where applicable, color).

2. F_MOVE unpacks this structure into individual values.
3. A multiplexer (F_SEL for 2 states, F_MUX_5 for 5 states, driven by DI1 or iSTATE) selects the values matching the current state.
4. Q_NumericValue/Q_NumericValueAux sets the selected object-pointer value on the VT object identified by Picture/PictureA/PictureB; Col variants additionally set Q_BackgroundColour/Q_BackgroundColourAux's background color on the object identified by Color.
5. With multiple target objects (variant 2/3), several Q_NumericValue(Aux) instances are chained in a fixed INIT sequence (each triggers the next via INITO) before INITO is reported outward.

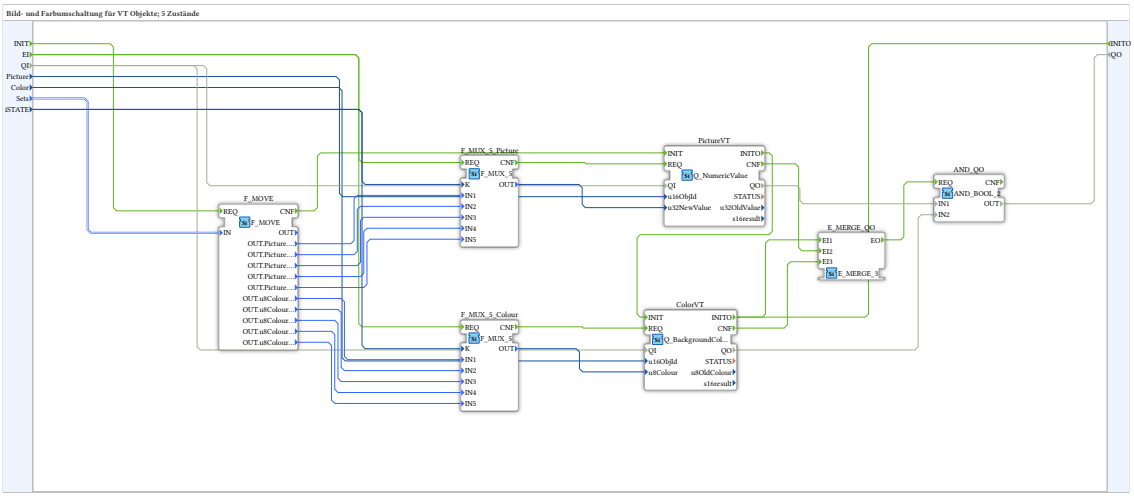
Summary

A multiplexer pattern: state → struct lookup (F_MOVE) → selection (F_SEL/F_MUX_5) → one or more Q_NumericValue(Aux)/Q_BackgroundColour(Aux) targets. The variant number (1/2/3) and Col/_aux only determine how many targets of which kind (regular/AUX, picture/color) get updated simultaneously.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPicCol_5_1



• – ○ • – ○ • – ○ •

Introduction

SwitchPicCol_5_1 switches both a VT picture (Picture, via Q_NumericValue) **and** a VT background color (Color, via Q_BackgroundColour) between 5 states (slide-valve animation) — two parallel F_MUX_5 multiplexers (one for picture IDs, one for color values), both driven by the same iSTATE value, targeting regular (non-AUX) objects.

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).

Initialization and quality: INIT → F_MOVE.REQ; F_MOVE.CNF → PictureVT.INIT; PictureVT.INITO → ColorVT.INIT; ColorVT.INITO → INITO. New are QI (BOOL, default TRUE), which goes to PictureVT.QI and ColorVT.QI, and QO (BOOL): AND (AND_QO, AND_BOOL_2) of PictureVT.QO and ColorVT.QO. AND_QO.REQ is triggered via E_MERGE_QO (E_MERGE_3: ColorVT.INITO, PictureVT.CNF, ColorVT.CNF).



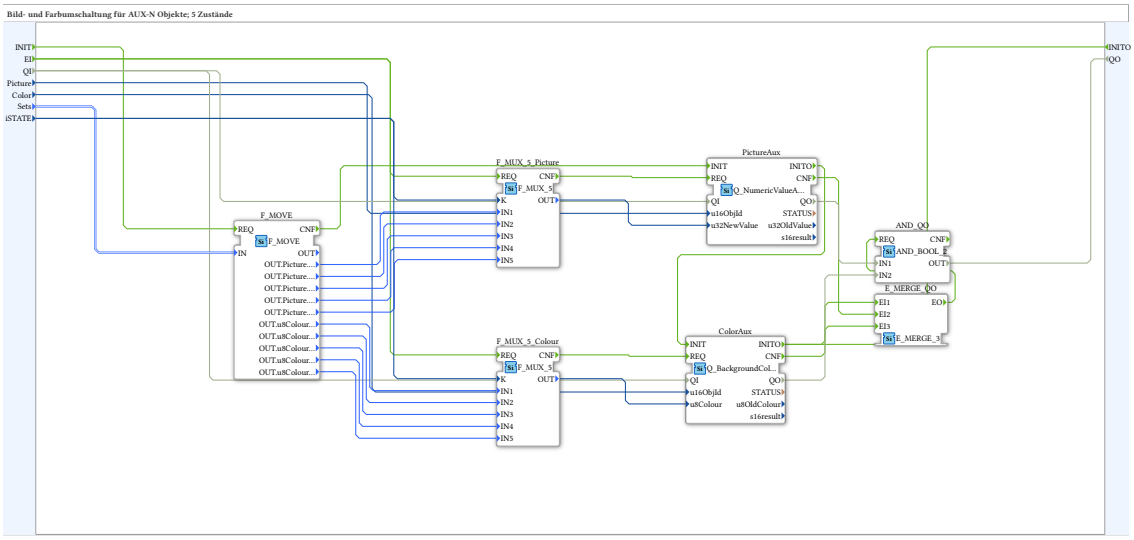
Summary

Combines picture and color switching (Col) for 5 states on regular VT objects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPicCol_5_1_aux



• – ○ • – ○ • – ○ •

Introduction

`SwitchPicCol_5_1_aux` switches both a picture and a background color between 5 states — like `SwitchPicCol_5_1`, but exclusively for **auxiliary function** objects (`Q_NumericValueAux`/ `Q_BackgroundColourAux`) instead of regular VT objects.

Initialization and quality: The SubApp has the event inputs `INIT` and `EI`, the event output `INITO` as well as `QI` (BOOL, initial value `TRUE`) and `QO` (BOOL). Init chain: `INIT` ⟶ `F_MOVE.REQ`, `F_MOVE.CNF` ⟶ `PictureAux.INIT`, `PictureAux.INITO` ⟶ `ColorAux.INIT`, `ColorAux.INITO` ⟶ `INITO`. `QI` is passed on to `PictureAux.QI` and `ColorAux.QI`; their `QO` outputs go to `AND_QO.IN1` and `AND_QO.IN2` respectively (`iec61131::booleanOperators::AND_BOOL_2`), `AND_QO.OUT` ⟶ `QO`. `AND_QO.REQ` is triggered by `E_MERGE_QO` (`E_MERGE_3`), which merges `ColorAux.INITO`, `PictureAux.CNF` and `ColorAux.CNF`.

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).



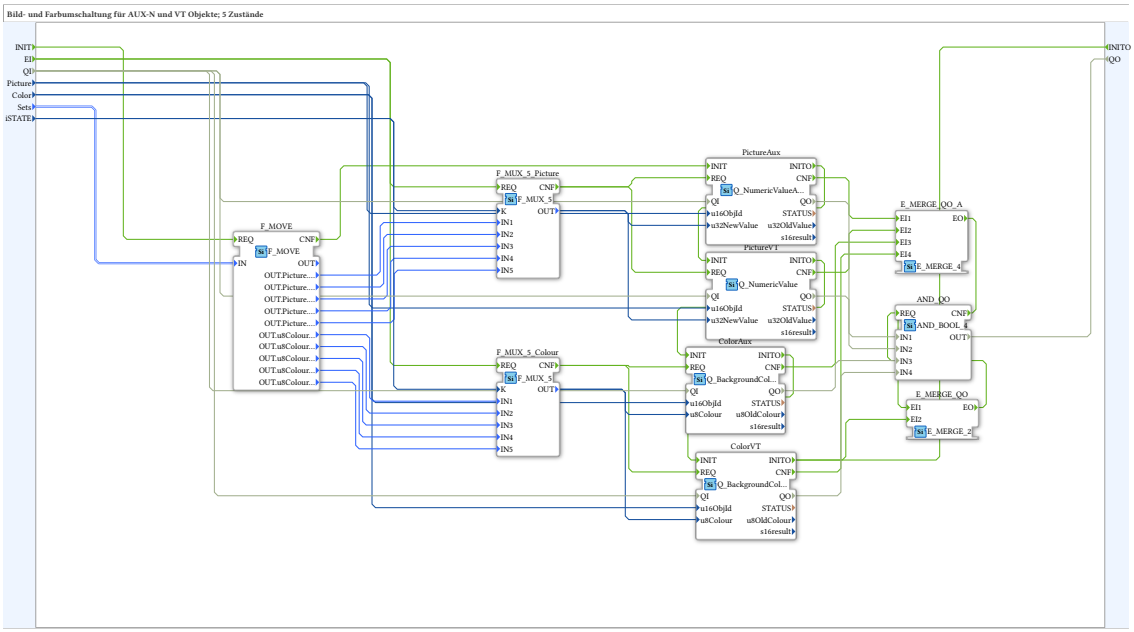
Summary

AUX counterpart to `SwitchPicCol_5_1`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPicCol_5_2

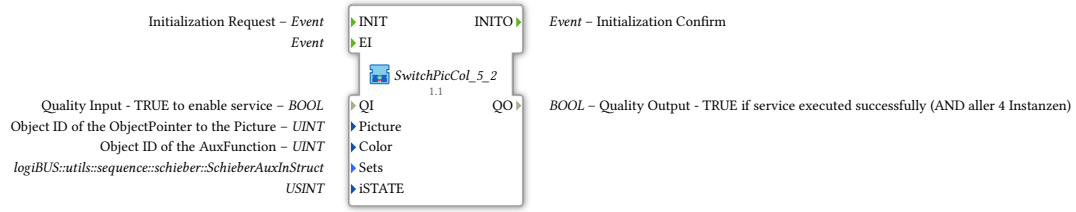


• – ○ • – ○ • – ○ •

Introduction

`SwitchPicCol_5_2` switches both picture and background color between 5 states — on **four** targets at once: a regular picture object (`PictureVT`), an AUX picture object (`PictureAux`), a regular color object (`ColorVT`), and an AUX color object (`ColorAux`), all driven by the same two `F_MUX_5` multiplexers (picture/color).

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).



Initialization and quality

The block has the event input `INIT`, the event output `INITO` (both `EInit`), the data input `QI` (`BOOL`, default `TRUE`) and the data output `QO` (`BOOL`). Initialization runs as a chain: `INIT` ⟶ `F_MOVE.REQ`, `F_MOVE.CNF` ⟶ `PictureAux.INIT`, `PictureAux.INITO` ⟶ `PictureVT.INIT`, `PictureVT.INITO` ⟶ `ColorAux.INIT`, `ColorAux.INITO` ⟶ `ColorVT.INIT`, `ColorVT.INITO` ⟶ `INITO`. `QI` is passed to `PictureAux`, `PictureVT`, `ColorAux` and `ColorVT`. `QO` is the output of the instance `AND_QO` (`iec61131::booleanOperators::AND_BOOL_4`), which combines their four `QO`. Its `REQ` is triggered by `E_MERGE_QO` (`E_MERGE_2`), which merges the output of `E_MERGE_QO_A` (`E_MERGE_4`, merges the `CNF` of `PictureAux`, `PictureVT`, `ColorAux` and `ColorVT`) and `ColorVT.INITO`.

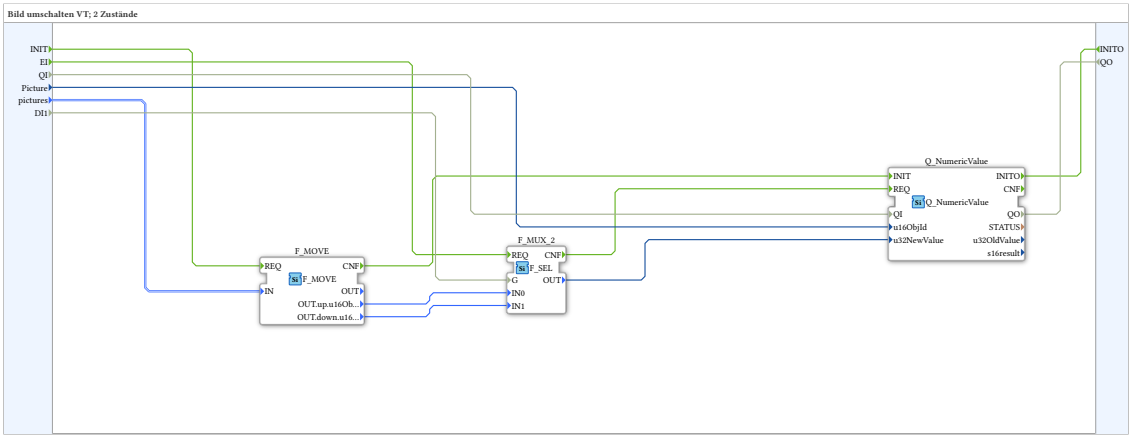
Summary

Extends `SwitchPicCol_5_1` with the AUX objects for both picture and color at once — the most extensive variant in the `SwitchPicCol` series.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPic_2_1

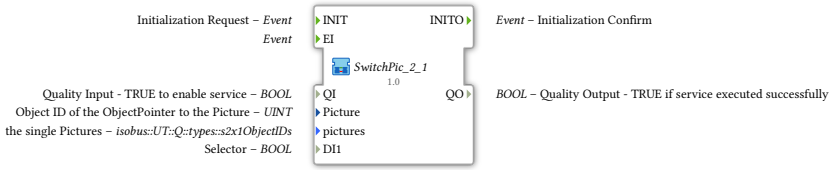


• – ○ • – ○ • – ○ •

Introduction

`SwitchPic_2_1` switches a VT picture (object pointer) between 2 states (`up`/`down`) based on a boolean selector (`DI1`) — only the regular VT object (`Picture`, via `Q_NumericValue`), no AUX object.

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).



Interface (QI/QO)

The block has the data input `QI` (`BOOL`, default `TRUE`, Quality Input) and the data output `QO` (`BOOL`, Quality Output). `QI` is passed through directly to `Q_NumericValue.QI`, and `Q_NumericValue.QO` directly to `QO`.

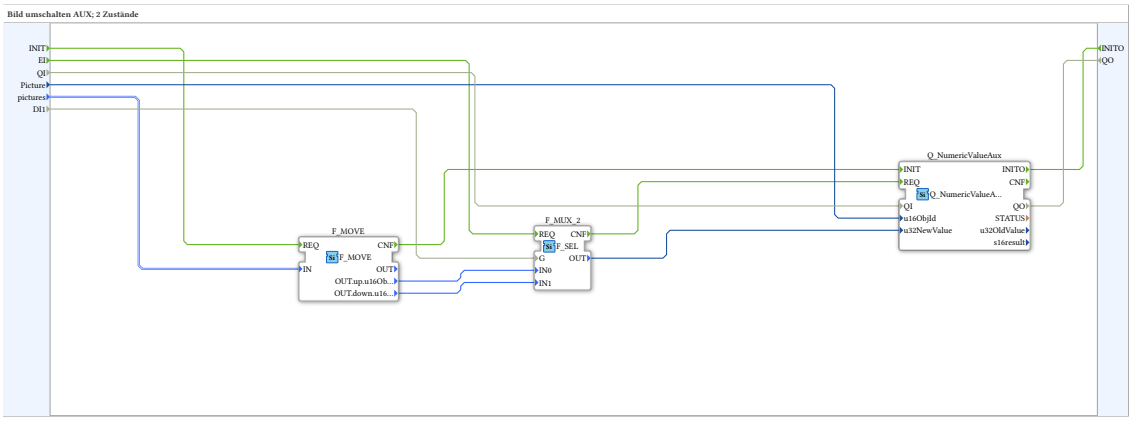
Summary

Variant “1” (regular object only) of the 2-state picture switch.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPic_2_1_aux

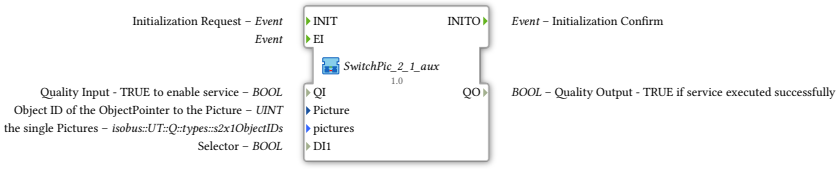


• – ○ • – ○ • – ○ •

Introduction

`SwitchPic_2_1_aux` switches a VT picture between 2 states (`up/down`) based on a boolean selector (`DI1`) — exclusively for an **auxiliary function** object (`Q_NumericValueAux`), no regular VT object.

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).



Interface (QI/QO)

The block has the data input `QI` (`BOOL`, default `TRUE`, Quality Input) and the data output `QO` (`BOOL`, Quality Output). `QI` is passed through directly to `Q_NumericValueAux.QI`, and `Q_NumericValueAux.QO` directly to `QO`.

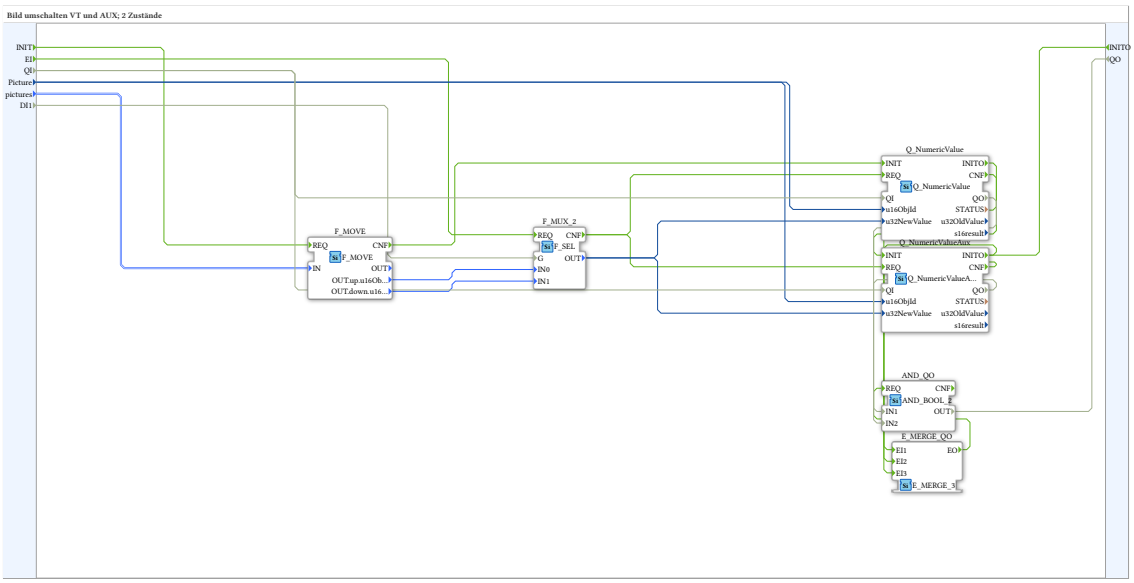
Summary

AUX counterpart to `SwitchPic_2_1`: same 2-state logic, but for an AUX object instead of a regular VT object.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPic_2_2

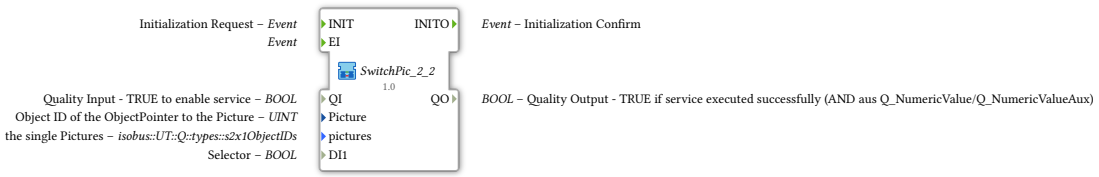


• — ○ • — ○ • — ○ •

Introduction

`SwitchPic_2_2` switches a VT picture between 2 states (`up/down`) based on a boolean selector (`DI1`) — on both a regular VT object (`Picture`, `Q_NumericValue`) and simultaneously an **auxiliary function** object (`Q_NumericValueAux`), both sharing the same object ID (`Picture`).

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).



Initialization and Quality

- **Event inputs** `INIT` (`EInit`) and `EI` (switch request); **event output** `INITO` (`EInit`).
- **Data input** **`QI`** (`BOOL`, default `TRUE`) ⟶ `Q_NumericValue.QI` and `Q_NumericValueAux.QI`; **data output** **`QO`** (`BOOL`): AND of `Q_NumericValue.QO` and `Q_NumericValueAux.QO` (`AND_QO`, `iec61131::booleanOperators::AND_BOOL_2`).
- Initialization: `INIT` ⟶ `F_MOVE.REQ`; `F_MOVE.CNF` ⟶ `Q_NumericValue.INIT`; `Q_NumericValue.INITO` ⟶ `Q_NumericValueAux.INIT`; `Q_NumericValueAux.INITO` ⟶ `INITO`.
- The `AND_QO.REQ` trigger comes from `E_MERGE_QO` (`iec61499::events::E_MERGE_3`), which merges `Q_NumericValueAux.INITO`, `Q_NumericValue.CNF` and `Q_NumericValueAux.CNF`.

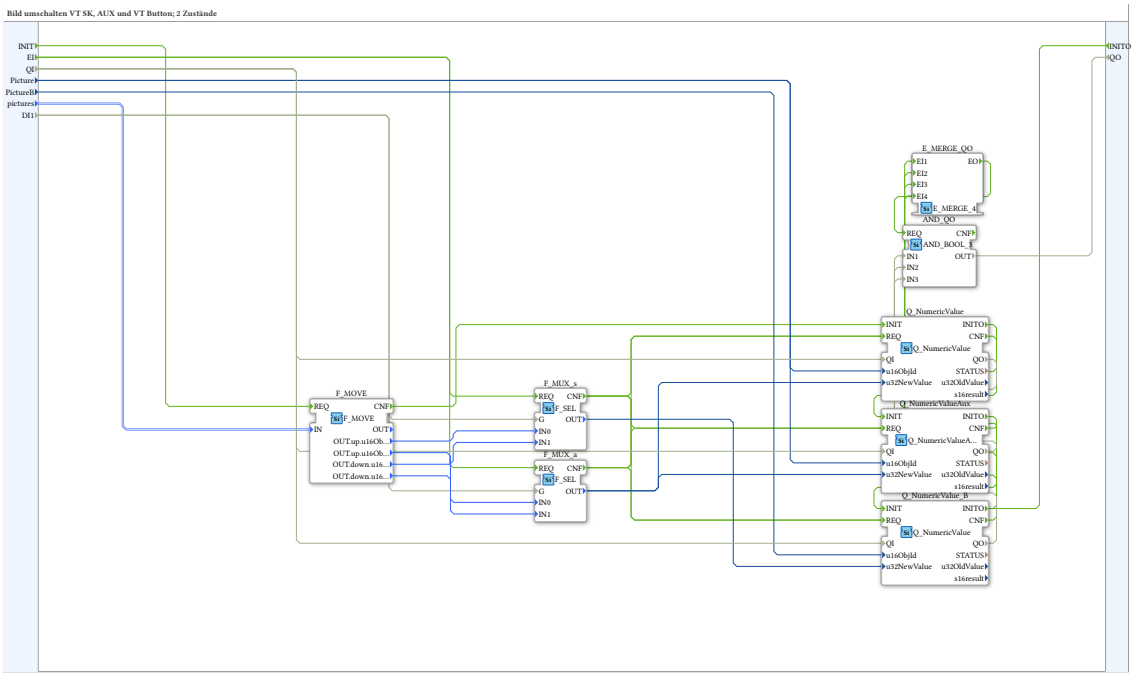
Summary

Variant “2” (regular object + AUX object) of the 2-state picture switch.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPic_2_3



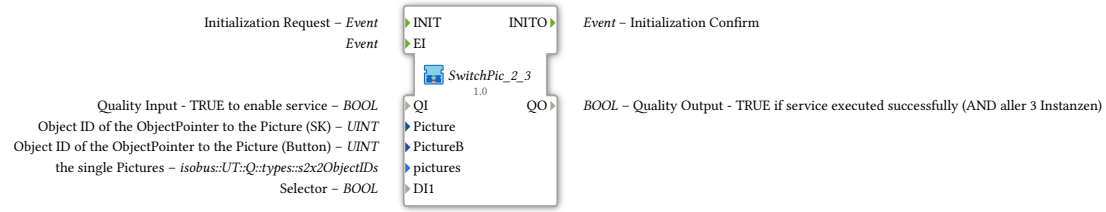
• — ○ • — ○ • — ○ •

Introduction

`SwitchPic_2_3` switches a VT picture between 2 states (up/down) based on a boolean selector (`DI1`) — on three targets at once: a regular softkey object (`Picture`), an AUX object (same ID `Picture`), and a second, independent regular object (“button”, `PictureB`, its own object-ID sub-field `u16ObjIdA` in the structure).

Interface (quality pattern): the data input `QI` (`BOOL`, default `TRUE`) is distributed to `Q_NumericValue`, `Q_NumericValueAux` and `Q_NumericValue_B`; the data output `QO` (`BOOL`) is the AND combination (`AND_QO`: `iec61131::booleanOperators::AND_BOOL_3`) of the `QO` outputs of these three instances (`IN1` to `IN3` in this order). The trigger `AND_QO.REQ` is supplied by `E_MERGE_QO` (`E_MERGE_4`), which merges `Q_NumericValue_B.INITO` and the `CNF` of `Q_NumericValue`, `Q_NumericValueAux` and `Q_NumericValue_B`.

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).



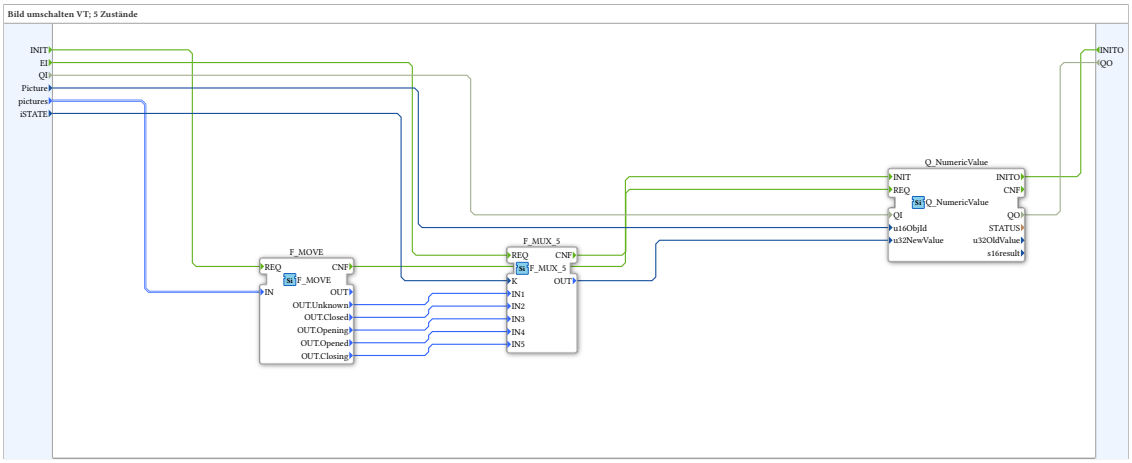
Summary

Variant “3” (regular softkey object + AUX object + second regular button object) of the 2-state picture switch — the most extensive variant in the `SwitchPic_2_*` series.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPic_5_1

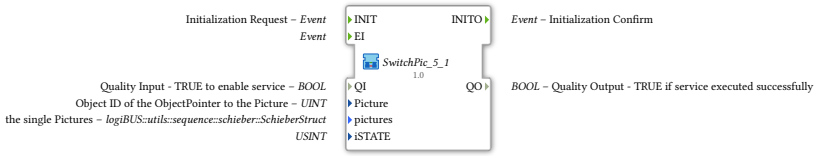


• – ○ • – ○ • – ○ •

Introduction

`SwitchPic_5_1` switches a VT picture between 5 states (slide-valve animation: `Unknown`, `Closed`, `Opening`, `Opened`, `Closing`) based on an `iSTATE` value (`USINT`) — only the regular VT object (`Picture`, via `Q_NumericValue`), no AUX object.

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).



Interface (QI/QO)

The block has the data input `QI` (`BOOL`, default `TRUE`, Quality Input) and the data output `QO` (`BOOL`, Quality Output). `QI` is passed through directly to `Q_NumericValue.QI`, and `Q_NumericValue.QO` directly to `QO`.

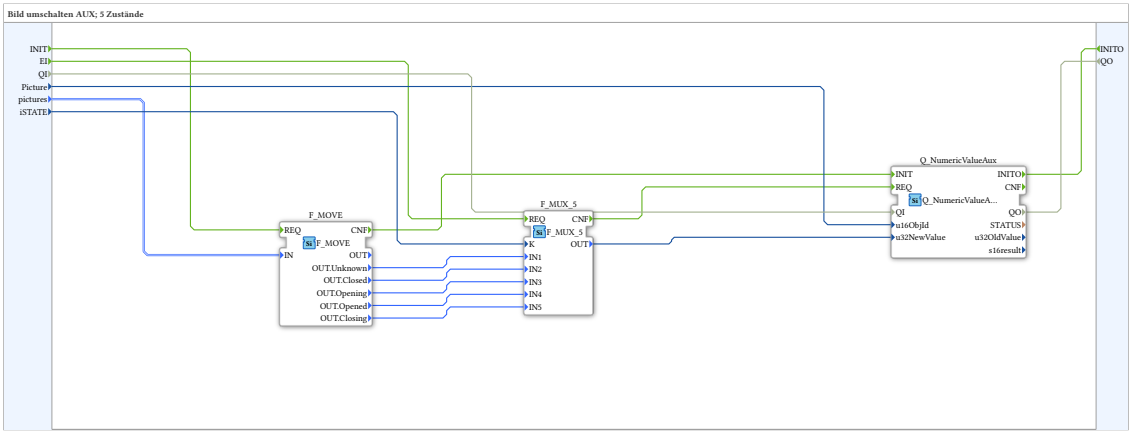
Summary

Variant “1” (regular object only) of the 5-state picture switch — the 5-state counterpart to `SwitchPic_2_1`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPic_5_1_aux

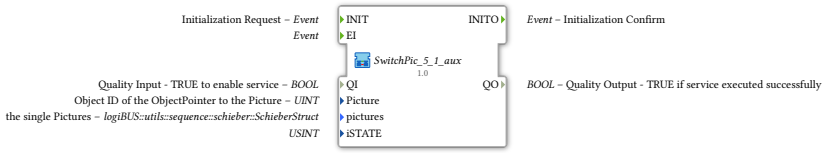


• – ○ • – ○ • – ○ •

Introduction

`SwitchPic_5_1_aux` switches a VT picture between 5 states (slide-valve animation) based on an `iSTATE` value — exclusively for an **auxiliary function** object (`Q_NumericValueAux`), no regular VT object.

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).



Interface (QI/QO)

The block has the data input `QI` (`BOOL`, default `TRUE`, Quality Input) and the data output `QO` (`BOOL`, Quality Output). `QI` is passed through directly to `Q_NumericValueAux.QI`, and `Q_NumericValueAux.QO` directly to `QO`.

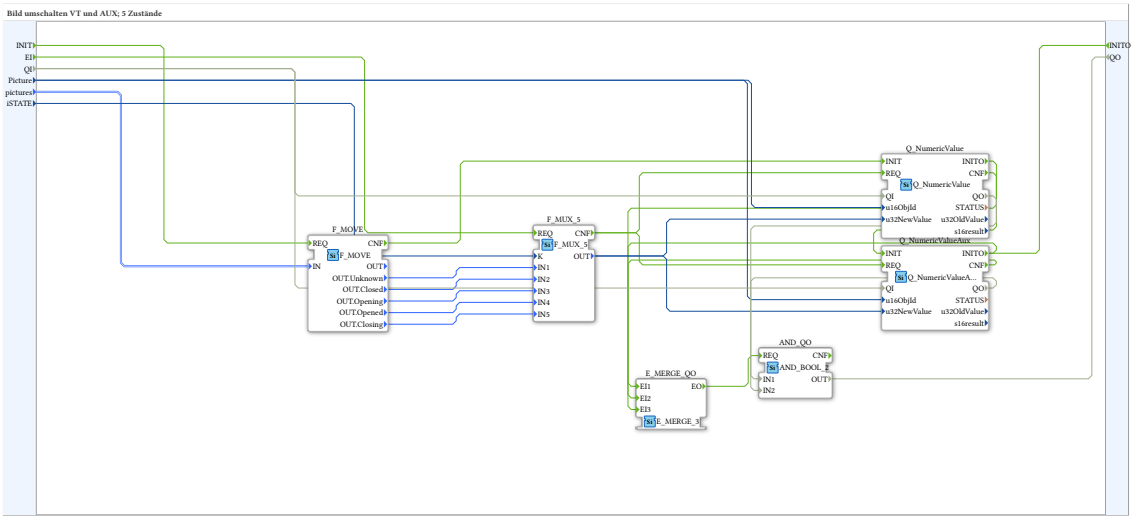
Summary

AUX counterpart to `SwitchPic_5_1`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPic_5_2

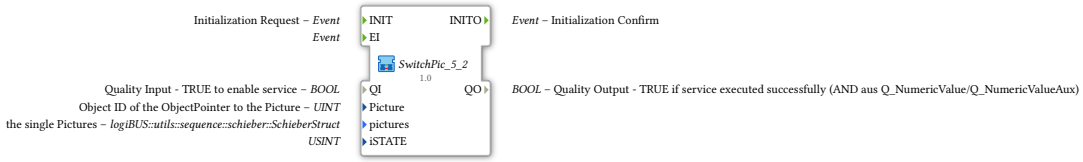


• – ○ • – ○ • – ○ •

Introduction

`SwitchPic_5_2` switches a VT picture between 5 states (slide-valve animation) based on an `iSTATE` value — on both a regular VT object (`Picture`) and simultaneously an AUX object (same ID `Picture`).

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).



Initialization and Status (INIT/INITO, QI/QO)

- **Event input** `INIT` (`EInit`) and **event output** `INITO` (`EInit`)
- **Data input** `QI` (`BOOL`, default `TRUE`, Quality Input) is forwarded to `Q_NumericValue.QI` and `Q_NumericValueAux.QI`
- **Data output** `QO` (`BOOL`, Quality Output): AND (`AND_QO`, type `iec61131::booleanOperators::AND_BOOL_2`) of the `QO` outputs of `Q_NumericValue` and `Q_NumericValueAux`
- **Initialization chain**: `INIT` ⟶ `F_MOVE.REQ`, `F_MOVE.CNF` ⟶ `Q_NumericValue.INIT`, `Q_NumericValue.INITO` ⟶ `Q_NumericValueAux.INIT`, `Q_NumericValueAux.INITO` ⟶ `INITO`
- `E_MERGE_QO` (type `iec61499::events::E_MERGE_3`) merges `Q_NumericValueAux.INITO`, `Q_NumericValue.CNF` and `Q_NumericValueAux.CNF` and triggers `AND_QO.REQ`

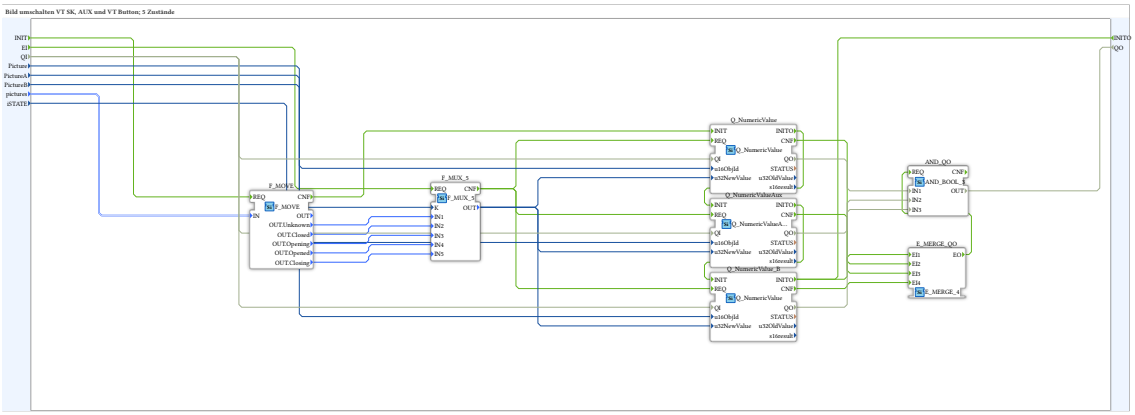
Summary

5-state counterpart to `SwitchPic_2_2`.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SwitchPic_5_3



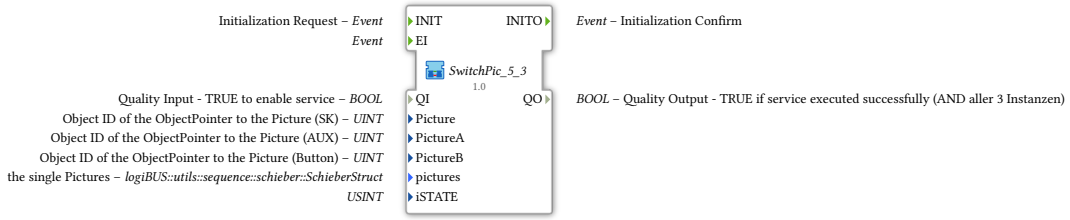
• – ○ • – ○ • – ○ •

Introduction

`SwitchPic_5_3` switches a VT picture between 5 states (slide-valve animation) based on an `iSTATE` value — on three independent targets at once: a regular softkey object (`Picture`), an AUX object (`PictureA`), and a second regular button object (`PictureB`), each with its own object ID.

For the general pattern, see SwitchPic(Col) Blocks (shared pattern).

Initialization and quality: `INIT` ⟶ `F_MOVE.REQ`; `F_MOVE.CNF` ⟶ `Q_NumericValue.INIT` ⟶ `Q_NumericValueAux.INIT` ⟶ `Q_NumericValue_B.INIT` (each via `INITO`); `Q_NumericValue_B.INITO` ⟶ `INITO`. New are `QI` (`BOOL`, default `TRUE`), which goes to all three `Q_NumericValue` instances, and `QO` (`BOOL`): `AND` (`AND_QO`, `AND_BOOL_3`) of the three `QO` outputs. `AND_QO.REQ` is triggered via `E_MERGE_QO` (`E_MERGE_4`: `Q_NumericValue_B.INITO`, `Q_NumericValue.CNF`, `Q_NumericValueAux.CNF`, `Q_NumericValue_B.CNF`).



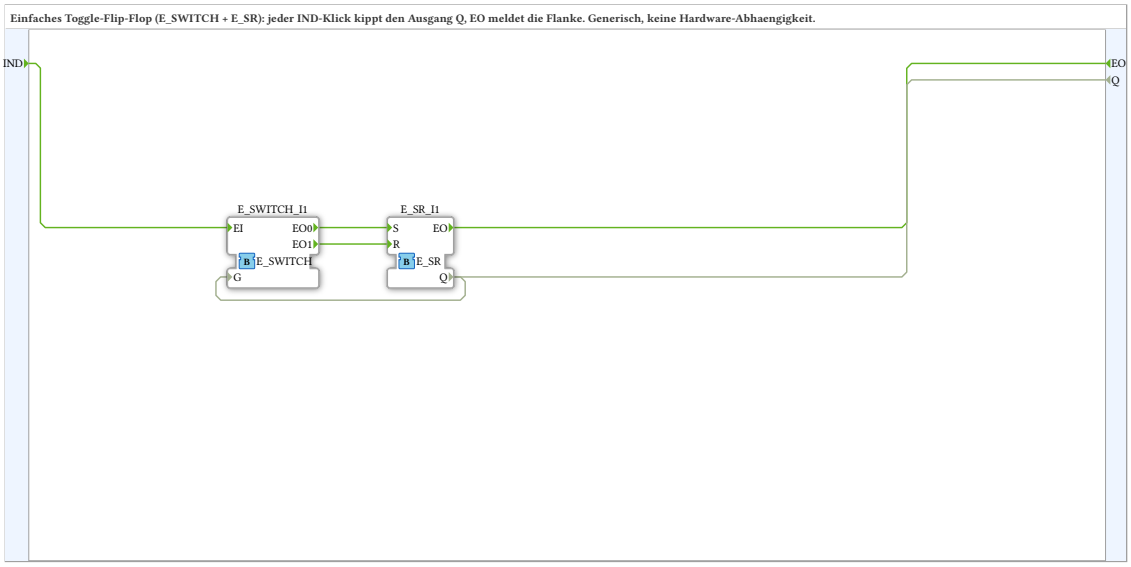
Summary

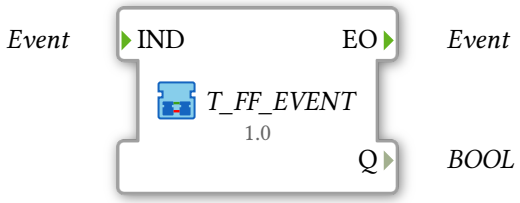
5-state counterpart to `SwitchPic_2_3` — the most extensive variant in the `SwitchPic_5_*` series.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

T_FF_EVENT





• – ○ • – ○ • – ○ •

Introduction

T_FF_EVENT is a generic toggle flip-flop subapplication implemented by combining an `E_SWITCH` and an `E_SR` function block. It provides a simple, event-driven mechanism where every activation of the `IND` event input toggles the boolean output `Q`. A corresponding `EO` event is emitted whenever the state changes, making the block suitable for edge detection and state-based control logic. The subapplication is fully generic and does not depend on any specific hardware platform.

Interface Structure

Event Inputs

Name	Type	Description

IND	Event	Input event that triggers the toggle operation. Each occurrence of this event flips the internal state and updates Q.

Event Outputs

Name	Type	Description
EO	Event	Output event fired after the internal state has changed. It signals that a toggle has occurred and Q has been updated.

Data Inputs

No data inputs are available for this subapplication.

Data Outputs

Name	Type	Description
Q	BOOL	Current state of the toggle flip-flop. It alternates between FALSE and TRUE with each IND event.

Adapters

No adapters are present in this subapplication.

Functionality

The internal network consists of an E_SWITCH block (named E_SWITCH_I1) and an E_SR block (named E_SR_I1), interconnected as follows:

1. The incoming IND event is routed to the EI input of E_SWITCH_I1.
2. The G (gate) input of E_SWITCH_I1 is driven by the current output Q of E_SR_I1.
3. When IND occurs, E_SWITCH_I1 evaluates its gate:
   - If G = FALSE (i.e., Q = FALSE), it issues an event on output EO0, which is connected to the S (set) input of E_SR_I1. This sets Q to TRUE.
   - If G = TRUE (i.e., Q = TRUE), it issues an event on output EO1, which is connected to the R (reset) input of E_SR_I1. This resets Q to FALSE.
4. After the state change, E_SR_I1 emits its EO output event, which is forwarded to the subapplication's EO output.

Thus, every IND event flips the state of Q, and the EO event confirms that the toggle has been completed.

Technical Features

- **Event-driven toggle**: The flip-flop changes state only when an IND event is received, not on a continuous level signal.
- **Generic implementation**: Built from standard IEC 61499 function blocks (E_SWITCH and E_SR), which are part of the basic event function block library.
- **No hardware dependency**: The logic is purely algorithmic and can be applied in any IEC 61499 runtime environment.
- **Single output**: The only data output is the boolean state Q, while the EO event provides a change notification.
- **No data inputs**: The subapplication requires no configuration parameters or external data to operate.

State Overview

The toggle flip-flop has two stable states, determined by the output Q:

State	Q Value	Description
OFF (Reset)	FALSE	Initial state. The next IND event will set Q to TRUE.
ON (Set)	TRUE	Toggled state. The next IND event will reset Q to FALSE.

The transition between states is strictly sequential: each IND event moves the block from one state to the other, and EO is emitted on every transition. There is no hold, debounce, or delay logic inside the block; the toggle is immediate and synchronous with the input event.

Application Scenarios

T_FF_EVENT is well suited for a variety of use cases where a binary state must be toggled by an event signal:

- **Push-button toggling**: Turning a light or actuator on and off with a momentary pushbutton, where each press generates an `IND` event.
- **Mode switching**: Alternating between two operating modes (e.g., automatic/manual) in a control application.
- **Edge detection and reporting**: Using `EO` to notify other parts of the system that the state has changed, e.g., to trigger logging or further processing.
- **State-based sequencing**: As a building block in larger subapplications where a binary flag needs to be inverted under event control.

Because the block is generic and hardware-independent, it can be reused across different projects, from simple logic exercises to industrial control systems.

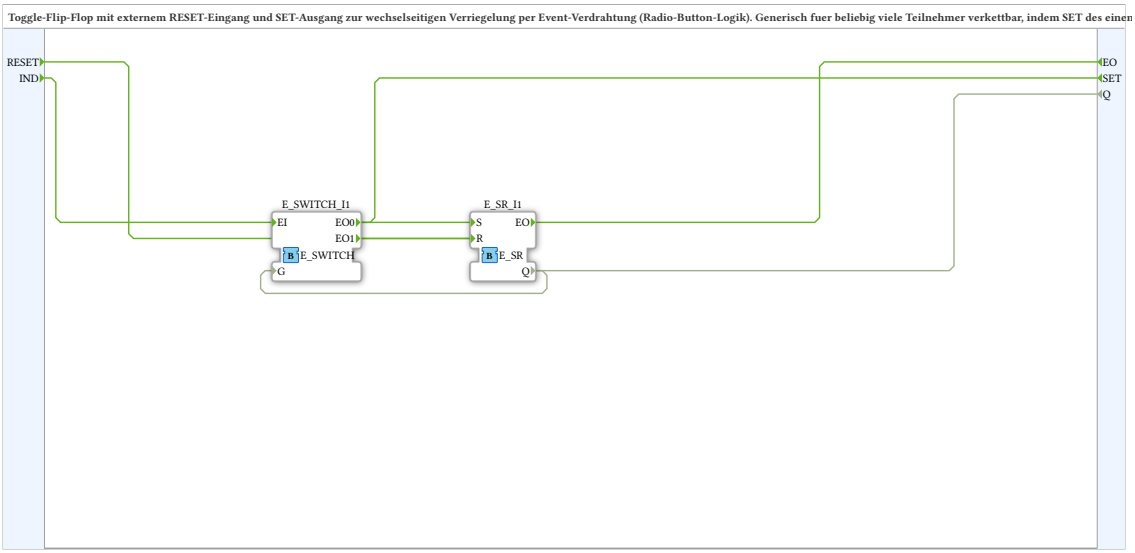
Comparison with Similar Blocks

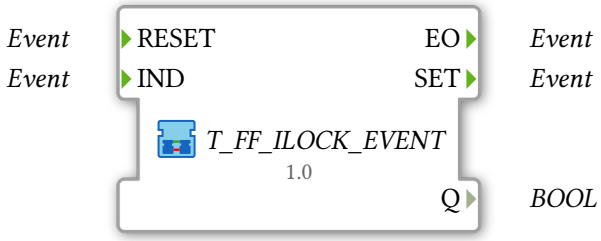
- **E_SR / E_RS (Set-Reset flip-flops)**: These blocks require two separate event inputs (`S` and `R`, or `R` and `S`) to set or reset the output. T_FF_EVENT is different because it requires only a single event input and automatically alternates between set and reset. It is equivalent to an E_SR with its set/reset inputs fed from an E_SWITCH controlled by its own output.
- **SR (data-level flip-flop)**: A standard SR latch uses level signals rather than events. T_FF_EVENT is event-driven and has no data input, so it cannot be forced to a particular state from outside; it always toggles from its current state.
- **T-flip-flop (binary counter)**: A classic T-flip-flop toggles on each active clock edge. T_FF_EVENT implements exactly this behavior using IEC 61499 events, but without a separate clock input; the `IND` event itself acts as the toggle trigger.

Conclusion

T_FF_EVENT is a compact and reusable subapplication that provides a simple toggle behavior with an event output for change notification. By combining an `E_SWITCH` and an `E_SR` in a feedback configuration, it achieves a clean, edge-triggered toggle flip-flop without external dependencies. Its generic design, minimal interface, and clearly defined state behavior make it an ideal building block for event-driven control systems, batch processing, and state management applications.

T_FF_ILOCK_EVENT





• – ◦ • – ◦ • – ◦ •

Introduction

T_FF_ILOCK_EVENT is a reusable IEC 61499 subapplication that implements an event-driven toggle flip-flop with an external reset input and a dedicated SET output for interlocking. It is designed for radio-button style mutual exclusion: when one instance becomes active, its SET event can be connected to the RESET input of another instance. Multiple instances can be chained this way to form an interlocked group.

Interface Structure

Event Inputs

Name	Type	Description
IND	Event	Toggle trigger. Each event toggles the internal state Q.
RESET	Event	Forces Q to FALSE and produces EO.

Event Outputs

Name	Type	Description
EO	Event	Acknowledgment output after an IND or RESET event has been processed.
SET	Event	Emitted only when Q changes from FALSE to TRUE; used to reset other participants.

Data Inputs

This subapplication has no data inputs.

Data Outputs

Name	Type	Description
Q	BOOL	Current state. TRUE means the participant is active.

Adapters

This subapplication exposes no adapters.

Functionality

Internally, T_FF_ILOCK_EVENT uses an E_SR flip-flop and an E_SWITCH event router.

When an IND event arrives, E_SWITCH uses the current state Q to decide whether to set or reset the internal flip-flop:

- If Q = FALSE, the event is routed to the Set input of E_SR. Q becomes TRUE and the SET output is emitted.
- If Q = TRUE, the event is routed to the Reset input of E_SR. Q becomes FALSE and no SET output is emitted.

The external RESET event is connected directly to the Reset input of E_SR. This allows an external event, or the SET output of another participant, to force the block into the inactive state.

After every accepted IND or RESET event, the EO output is emitted.

Technical Features

- Pure event-driven operation; no polling or cyclic execution is required.
- Uses only standard IEC 61499 event and data connections.
- One BOOL output Q for state indication.
- SET output provides a clean pulse for interlocking or resetting other participants.
- RESET input supports master reset and daisy-chain interlocking.
- No data inputs, so no additional configuration is required.
- Scales to any number of participants by connecting SET of one instance to RESET of the next.

State Overview

Current Q	Event	Resulting Q	Output Events
FALSE	IND	TRUE	SET, EO
TRUE	IND	FALSE	EO
FALSE	RESET	FALSE	EO
TRUE	RESET	FALSE	EO

This behavior creates a toggle switch: each IND toggles the local state, while RESET guarantees a defined inactive state.

Application Scenarios

- **Radio-button control groups:** Selecting a new participant automatically deselects the previously active one by connecting SET to the previous participant's RESET.
- **Exclusive modes in machines:** Use one instance per mode, e.g. Auto, Manual, Setup, and chain the SET outputs to the other RESET inputs.
- **Toggle pushbuttons:** Use IND from a button to create a latching toggle output, with RESET available for emergency off or master reset.
- **Multi-station interlocking:** Chain any number of instances for arbitration between stations, recipes, or active states.

Comparison with Similar Blocks

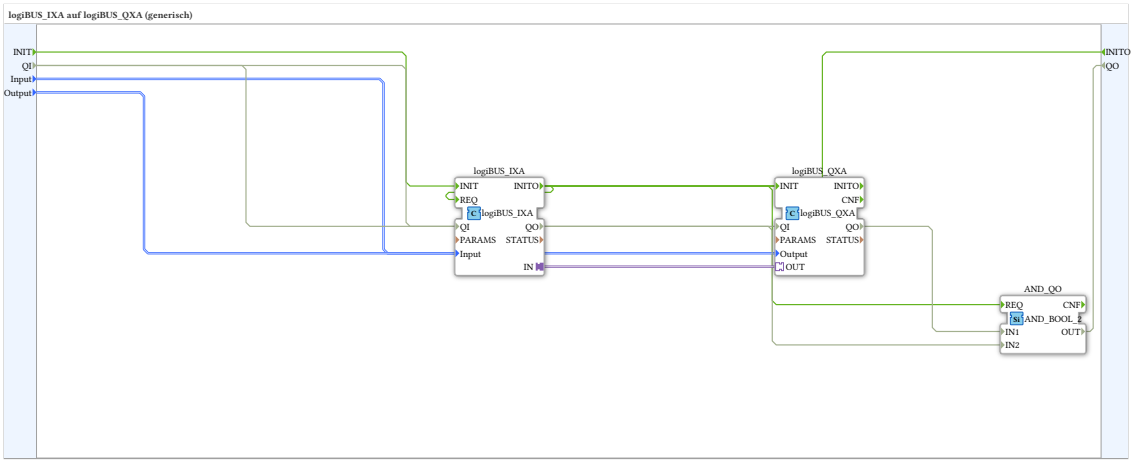
- **E_SR:** A plain set/reset flip-flop requires separate set and reset events. It has no toggle behavior and no interlocking SET output.
- **E_SWITCH:** Routes events based on a selector input but has no memory and cannot store the selected state.
- **A generic T-FF:** Toggles on each input event, but does not provide a dedicated SET output, external reset support, or event-based interlocking convenience.

T_FF_ILOCK_EVENT combines the memory of an E_SR with the routing capability of an E_SWITCH and adds an interlock-oriented SET output, making it especially suitable for event-based radio-button logic.

Conclusion

T_FF_ILOCK_EVENT is a compact, reusable, event-driven interlock subapplication. It provides toggle behavior, external reset, state output, and a SET pulse for daisy-chaining mutual exclusion groups. Because it requires no data inputs and no additional wiring logic, it can be used consistently in HMI panels, control applications, and multi-station systems.

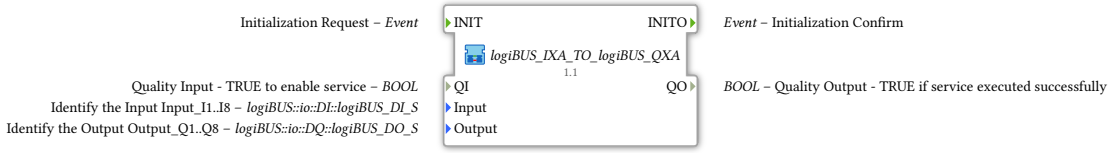
logiBUS_IXA_TO_logiBUS_QXA



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_IXA_TO_logiBUS_QXA` wires a physical digital input (`logiBUS_IXA`) directly to a physical digital output (`logiBUS_QXA`) — a pure hardware pass-through with no VT involvement, adapter-based (acyclic with confirmation, `QI` comes from the input `QI`). For the event-driven variant without an adapter, see `logiBUS_IX_TO_logiBUS_QX`.



Function Blocks Used

Sub-blocks: logiBUS_IXA_TO_logiBUS_QXA

- **Type**: SubAppType
- **Internal FBs used**:
  - **logiBUS_IXA**: `logiBUS::io::DI::logiBUS_IXA` — physical digital input, adapter output `IN`, `QI` comes from the input `QI`.
  - **logiBUS_QXA**: `logiBUS::io::DQ::logiBUS_QXA` — physical digital output, adapter input `OUT`, `QI` comes from the input `QI`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — combines `logiBUS_QXA.QO` and `logiBUS_IXA.QO` into the output `QO`.
- **Operation**: The input's adapter output is wired directly to the output's adapter input — no intermediate logic, no VT connection. The SubApp interface additionally has the events `INIT`/`INITO` as well as `QI` (BOOL, initial value `TRUE`) and `QO` (BOOL).

Program Flow and Connections

1. `Input` ⟶ `logiBUS_IXA.Input`; `Output` ⟶ `logiBUS_QXA.Output`.
2. `logiBUS_IXA.IN` (adapter) ⟶ `logiBUS_QXA.OUT` (adapter) — direct pass-through.
3. Initialization chain: `INIT` ⟶ `logiBUS_QXA.INIT`, `logiBUS_QXA.INITO` ⟶ `logiBUS_IXA.INIT`, `logiBUS_IXA.INITO` ⟶ `INITO` and `AND_QO.REQ` (in addition `logiBUS_IXA.INITO` ⟶ `logiBUS_IXA.REQ`).
4. Quality: `QI` ⟶ `logiBUS_QXA.QI` and `logiBUS_IXA.QI`; `logiBUS_QXA.QO` ⟶ `AND_QO.IN1`; `logiBUS_IXA.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ `QO`.

Application Scenarios

- Pure hardware-to-hardware wiring, e.g. for a physical emergency-stop contact or a fixed interlock that must work independently of the VT.

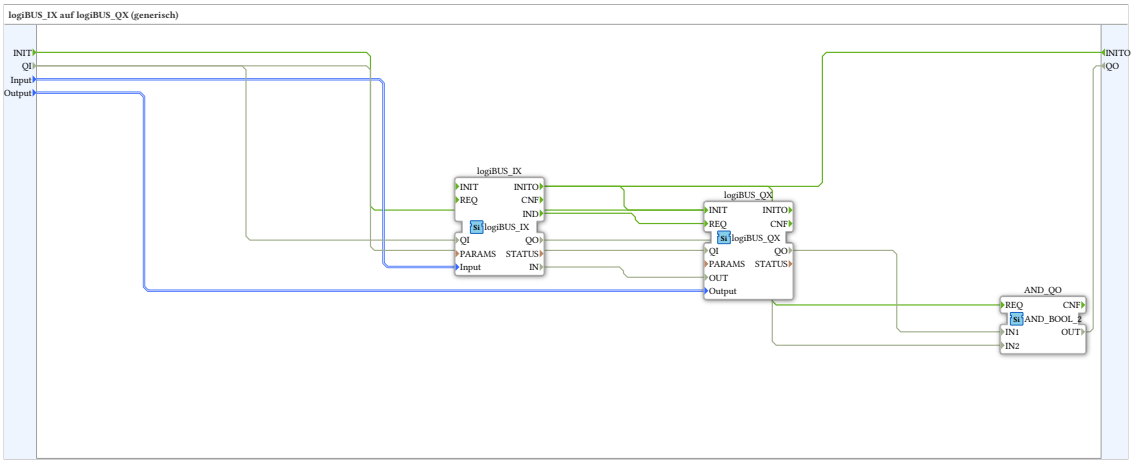
Summary

Adapter-based direct wiring of a physical input to a physical output, with no VT or event logic.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

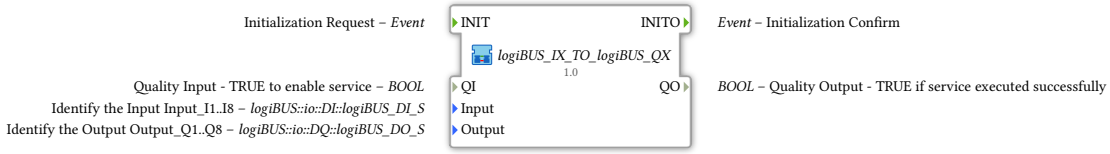
logiBUS_IX_TO_logiBUS_QX



• – ○ • – ○ • – ○ •

Introduction

`logiBUS_IX_TO_logiBUS_QX` wires a physical digital input (`logiBUS_IX`) directly to a physical digital output (`logiBUS_QX`) — functionally equivalent to `logiBUS_IXA_TO_logiBUS_QXA`, but using the non-adapter-based `logiBUS_IX`/`logiBUS_QX` variants: the connection is made via an explicit event connection plus a data connection instead of an adapter.



Function Blocks Used

Sub-blocks: logiBUS_IX_TO_logiBUS_QX

- **Type**: SubAppType
- **Internal FBs used**:
  - **logiBUS_IX**: `logiBUS::io::DI::logiBUS_IX` — physical digital input, fires `IND` on state change.
  - **logiBUS_QX**: `logiBUS::io::DQ::logiBUS_QX` — physical digital output, driven via `REQ`.
  - **AND_QO**: `iec61131::booleanOperators::AND_BOOL_2` — combines the `QO` of `logiBUS_QX` and `logiBUS_IX`.
- **Operation**: The input event `IND` directly triggers `REQ` on the output; the current state is passed through in parallel via a data connection from `IN` to `OUT`.

Program Flow and Connections

1. `Input` ⟶ `logiBUS_IX.Input`; `Output` ⟶ `logiBUS_QX.Output`.
2. `logiBUS_IX.IND` ⟶ `logiBUS_QX.REQ` (event connection).
3. `logiBUS_IX.IN` ⟶ `logiBUS_QX.OUT` (data connection).
4. Initialization: event input `INIT` ⟶ `logiBUS_QX.INIT`, `logiBUS_QX.INITO` ⟶ `logiBUS_IX.INIT`, `logiBUS_IX.INITO` ⟶ event output `INITO` and ⟶ `AND_QO.REQ`.
5. Quality: data input `QI` (`BOOL`, default `TRUE`) ⟶ `logiBUS_QX.QI` and `logiBUS_IX.QI`; `logiBUS_QX.QO` ⟶ `AND_QO.IN1`, `logiBUS_IX.QO` ⟶ `AND_QO.IN2`; `AND_QO.OUT` ⟶ data output `QO`.

Application Scenarios

- Hardware-to-hardware pass-through in training systems that (unlike `test_AX`) rely on the non-adapter-based I/O blocks `logiBUS_IX`/`logiBUS_QX`.

Summary

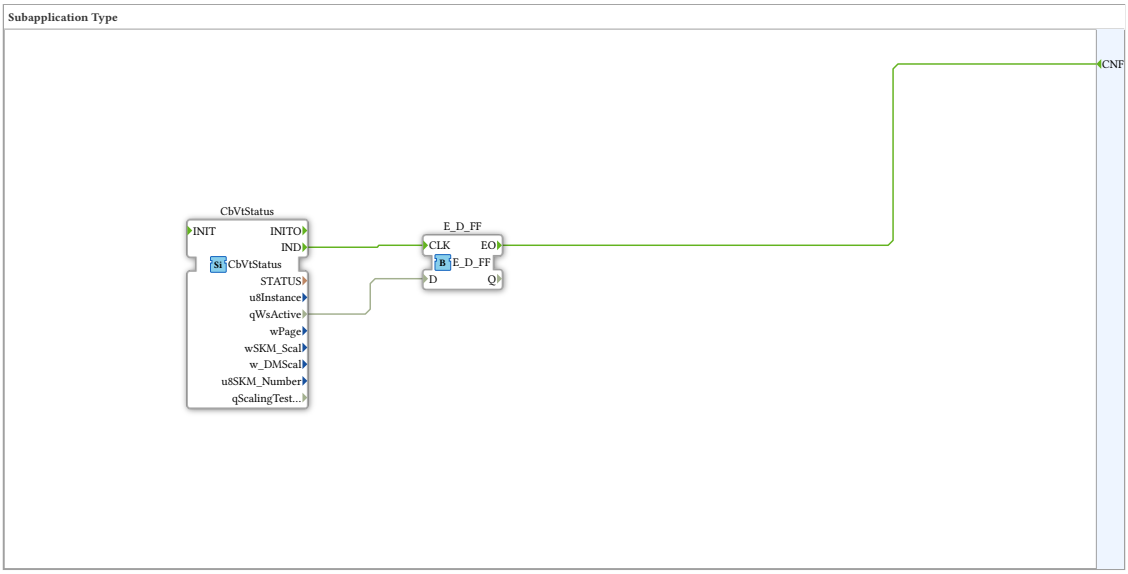
Non-adapter-based counterpart to `logiBUS_IXA_TO_logiBUS_QXA`: same function, explicit event/data connection instead of adapter coupling.

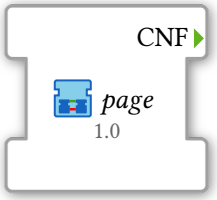
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Page

page





Event – Execution Confirmation

• – ○ • – ○ • – ○ •

Introduction

The `page` subapplication is a compact IEC 61499 composite type that encapsulates an ISOBUS status monitoring block together with an event-triggered D flip-flop. It exposes a single event output `CNF` and provides no external data interface. Internally, the subapplication uses the `CbVtStatus` block from the ISOBUS UT status library to detect a status indication and the standard `E_D_FF` block to sample a boolean value at exactly the moment the indication occurs.

Interface Structure

The subapplication interface is intentionally minimal: it only exposes one event output. All data and event processing is done inside the encapsulated network.

Event Inputs

- None

Event Outputs

- `CNF` : Execution Confirmation
  This event is emitted after the internal event chain has been completed. It confirms that the status indication has been processed and the associated boolean value has been latched.

Data Inputs

- None

Data Outputs

- None

Adapters

- None

Functionality

Internally, the subapplication contains two function blocks:

1. `CbVtStatus` of type `isobus::UT::status::CbVtStatus`
   This block monitors an ISOBUS UT status condition. When a status indication occurs, it produces the event `IND`. It also provides the boolean data output `qWsActive`, which represents the current working-set-active state.
2. `E_D_FF` of type `iec61499::events::E_D_FF`
   This is an event-triggered D flip-flop. It uses the event input `CLK` to sample the boolean value present at its data input `D`. When the clock event occurs, the flip-flop latches the value of `D` and emits the event output `EO`.

The internal connections perform the following task:

- `CbVtStatus.IND` is connected to `E_D_FF.CLK`, so every status indication triggers the flip-flop.
- `CbVtStatus.qWsActive` is connected to `E_D_FF.D`, so the current working-set-active value is used as the data input.
- `E_D_FF.EO` is connected to the subapplication output `CNF`, so the confirmation event is emitted after the value has been latched.

In effect, the subapplication captures the boolean status value exactly at the time of the indication and produces a confirmation event for the surrounding application.

Technical Features

- Encapsulated subapplication with no exposed event or data inputs.
- Single event output `CNF` for handshake-based integration.
- Internal event flow: `CbVtStatus.IND` ⟶ `E_D_FF.CLK` ⟶ `E_D_FF.EO` ⟶ `CNF`.
- Internal data flow: `CbVtStatus.qWsActive` ⟶ `E_D_FF.D`.
- Uses standard IEC 61499 components from the `iec61499::events` library.
- Uses an ISOBUS-specific status block from the `isobus::UT::status` library.
- The externally visible behavior is event-driven and synchronous: one status indication results in one confirmation event.

State Overview

The internal `E_D_FF` block can be considered as having two stable states:

State	Latched Value	Description
Reset	`false`	`qWsActive` was `false` at the last `CLK` event
Set	`true`	`qWsActive` was `true` at the last `CLK` event

The state is held internally and is not exposed through the subapplication data interface. The only visible effect is that each `CLK` event causes the `EO` event to be emitted, which is forwarded as `CNF`.

Application Scenarios

This subapplication is suitable for event-driven ISOBUS applications where the controller must be informed whenever a VT status indication occurs. Typical use cases include:

- Monitoring the working-set-active status of an ISOBUS Universal Terminal.
- Capturing a boolean status value at a precise event boundary.
- Generating a confirmation event for use in a larger function block network.
- Synchronizing status evaluation with downstream event-triggered logic.

Because the subapplication exposes only a confirmation event, it can be integrated cleanly into systems where data transfer is not required or where the status value is evaluated further inside another composite block.

Comparison with Similar Blocks

Compared with a simple direct connection from `CbVtStatus.IND` to the subapplication output, this subapplication additionally samples and latches the `qWsActive` data value. That makes the captured state stable and independent of later changes to `qWsActive`.

Compared with an `E_RS_FF`-based solution, the `E_D_FF` approach uses a data input to define the next state instead of separate set and reset event paths. This is better suited for situations where the state should follow a sampled boolean signal.

Compared with a pure combinatorial event-forwarding block, the internal flip-flop ensures that the data value is consistent with the event that triggered the confirmation.

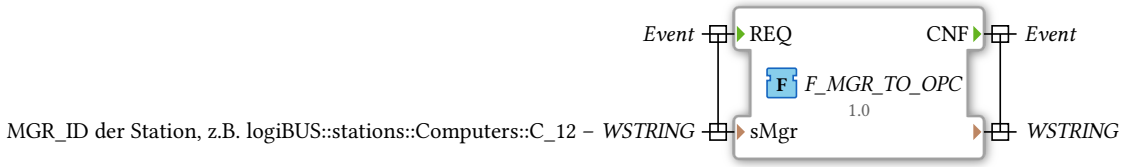
Conclusion

The `page` subapplication is a small but useful reusable component. It combines ISOBUS VT status indication with a D-type event latch to provide a single confirmation event. Its minimal interface makes it easy to embed in larger 4diac applications, while the internal flip-flop guarantees that the associated boolean value is captured at the correct moment.

2.1.7 MyLib_OPC

- F_MGR_TO_OPC
- F_OPC_LOCAL
- F_OPC_REMOTE

2.1.7.1 F_MGR_TO_OPC: Function (IEC 61499 Function)



• – ◦ • – ◦ • – ◦ •

Introduction

`F_MGR_TO_OPC` is not a SubApp but a function type (`.fct`, standard 61499-1) in the package `MyLib_OPC` with the comment “OPC-UA endpoint from MGR_ID: 192.168.1.12:61499 -> opc.tcp:

Function Interface

Instance	Type	Parameters	Explanation
`REQ`	`Event (event input)`	`With = sMgr`	Execution request.
`CNF`	`Event (event output)`	`With` = (unnamed variable)	Confirmation; delivers the return value.
`sMgr`	`WSTRING (input variable)`	- constant (`logiBUS::stations::Computers::C_12` has the value `"192.168.1.12:61499"`).	MGR_ID of the station in the format `host:port`, e.g. `"192.168.1.12:61499"`. This is the value of a station
(unnamed)	`WSTRING (output variable)`	-	Return value `F_MGR_TO_OPC` (empty output name).

Program Flow and Connections

The body is written in Structured Text:

```
PACKAGE MyLib_OPC;

(* OPC-UA-Endpunkt aus MGR_ID: 192.168.1.12:61499 -> opc.tcp://192.168.1.12:4840 *)
FUNCTION F_MGR_TO_OPC : WSTRING

VAR_INPUT
    sMgr : WSTRING; // MGR_ID der Station, z.B. logiBUS::stations::Computers::C_12
END_VAR

F_MGR_TO_OPC := CONCAT("opc.tcp://", LEFT(sMgr, FIND(sMgr, ":")), "4840");

END_FUNCTION
```

Note: The source comment `// MGR_ID der Station, z.B. logiBUS::stations::Computers::C_12` names the station constant. `sMgr` receives its value in the format `host:port` (here `"192.168.1.12:61499"`); a hierarchical name without a colon would not give a valid endpoint.

1. `FIND(sMgr, ":")` returns the position of the first colon.
2. `LEFT(sMgr, ...)` cuts the text up to and including this colon (`192.168.1.12:`); the MGR port `61499` is dropped.
3. `CONCAT` joins `opc.tcp://`, this part and the fixed OPC-UA port `4840`.

Example

Call and result:

```
MyLib_OPC::F_MGR_TO_OPC("192.168.1.12:61499")  ->  opc.tcp://192.168.1.12:4840
```

Usage

Used in `myOpcUaAddresses.gcf` (project `test_VV`, folder `const/OPC_UA`, described in `OPC_UA_Adressen.md`) as the `InitialValue` of a WSTRING constant:

```
InitialValue="MyLib_OPC::F_MGR_TO_OPC(PC_A)"
```

`PC_A_OPC` and `PC_B_OPC` get the OPC-UA endpoint of the respective station via `MyLib_OPC::F_MGR_TO_OPC(PC_A)` and `MyLib_OPC::F_MGR_TO_OPC(PC_B)`.

The expression is evaluated at export time; the boot file therefore contains only the finished strings, no function calls.

Note: The arguments must be constants (literals or other constants of the `.gcf`), because evaluation happens at export time, not at runtime.

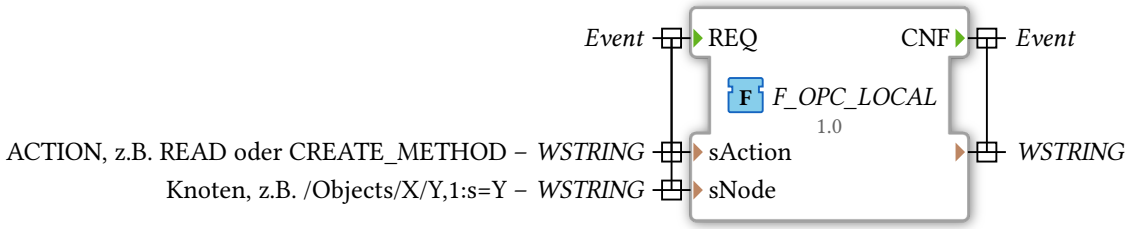
Summary

`F_MGR_TO_OPC` is a small IEC 61499 function of the library `MyLib_OPC` that derives the OPC-UA endpoint from a MGR_ID.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.1.7.2 F_OPC_LOCAL: Function (IEC 61499 Function)



• – ◦ • – ◦ • – ◦ •

Introduction

F_OPC_LOCAL is not a SubApp but a function type (.fct, standard 61499-1) in the package MyLib_OPC with the comment "OPC-UA ID for a local node: opc_ua[ACTION;NODE]". It has one event input REQ, one event output CNF and a Structured Text body; there are no function block instances and no connections.

Function Interface

Instance	Type	Parameters	Explanation
REQ	Event (event input)	With = sAction, sNode	Execution request.
CNF	Event (event output)	With = (unnamed variable)	Confirmation; delivers the return value.
sAction	WSTRING (input variable)	-	ACTION, e.g. READ or CREATE_METHOD
sNode	WSTRING (input variable)	-	Node, e.g. /Objects/X/Y,1:s=Y
(unnamed)	WSTRING (output variable)	-	Return value F_OPC_LOCAL (empty output name).

Program Flow and Connections

The body is written in Structured Text:

```
PACKAGE MyLib_OPC;

(* OPC-UA-ID fuer einen lokalen Knoten: opc_ua[ACTION;NODE] *)
FUNCTION F_OPC_LOCAL : WSTRING

VAR_INPUT
    sAction : WSTRING; // ACTION, z.B. READ oder CREATE_METHOD
    sNode : WSTRING; // Knoten, z.B. /Objects/X/Y,1:s=Y
END_VAR

F_OPC_LOCAL := CONCAT("opc_ua[", sAction, ";", sNode, "]");

END_FUNCTION
```

1. CONCAT joins opc_ua[, sAction, ;, sNode and].

Example

Call and result:

```
MyLib_OPC::F_OPC_LOCAL("READ", "/Objects/X/Y,1:s=Y")  ->  opc_ua[READ;/Objects/X/
Y,1:s=Y]
```

Usage

Used in myOpcUaAddresses.gcf (project test_VV, folder const/OPC_UA, described in OPC_UA_Adressen.md) as the InitialValue of a WSTRING constant:

```
InitialValue="MyLib_OPC::F_OPC_LOCAL(&amp;quot;READ&amp;quot;, Q1_NODE)"
```

Q1_LOCAL_READ and FF1_TRIGGER_LOCAL_METHOD (local on station B) use MyLib_OPC::F_OPC_LOCAL("READ", Q1_NODE) and MyLib_OPC::F_OPC_LOCAL("CREATE_METHOD", FF1_TRIGGER_NODE).

The expression is evaluated at export time; the boot file therefore contains only the finished strings, no function calls.

Note: The arguments must be constants (literals or other constants of the .gcf), because evaluation happens at export time, not at runtime.

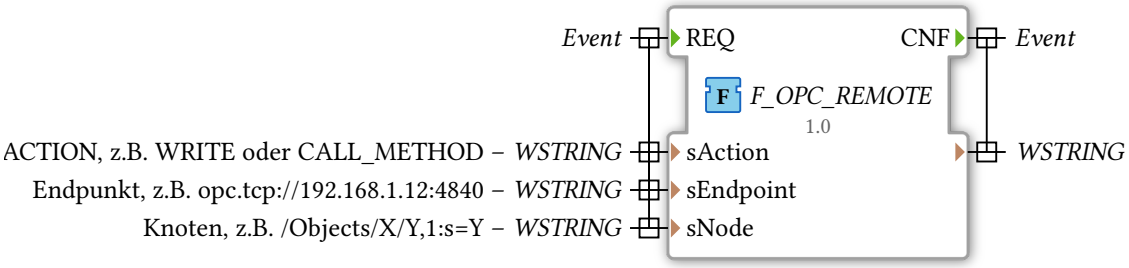
Summary

F_OPC_LOCAL is a small IEC 61499 function of the library MyLib_OPC that assembles the OPC-UA ID for a local node.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.1.7.3 F_OPC_REMOTE: Function (IEC 61499 Function)



• – ◦ • – ◦ • – ◦ •

Introduction

`F_OPC_REMOTE` is not a SubApp but a function type (`.fct`, standard 61499-1) in the package `MyLib_OPC` with the comment "OPC-UA ID for a node on another station: opc_ua[ACTION;ENDPOINT#;NODE]". It has one event input `REQ`, one event output `CNF` and a Structured Text body; there are no function block instances and no connections.

Function Interface

Instance	Type	Parameters	Explanation
`REQ`	`Event (event input)`	`With = sAction, sEndpoint, sNode`	Execution request.
`CNF`	`Event (event output)`	`With` = (unnamed variable)	Confirmation; delivers the return value.
`sAction`	`WSTRING (input variable)`	-	ACTION, e.g. `WRITE` or `CALL_METHOD`
`sEndpoint`	`WSTRING (input variable)`	-	Endpoint, e.g. `opc.tcp://192.168.1.12:4840`
`sNode`	`WSTRING (input variable)`	-	Node, e.g. `/Objects/X/Y,1:s=Y`
(unnamed)	`WSTRING (output variable)`	-	Return value `F_OPC_REMOTE` (empty output name).

Program Flow and Connections

The body is written in Structured Text:

```
PACKAGE MyLib_OPC;

(* OPC-UA-ID fuer einen Knoten auf einer anderen Station: opc_ua[ACTION;ENDPOINT#;NODE]
*)
FUNCTION F_OPC_REMOTE : WSTRING

VAR_INPUT
    sAction : WSTRING; // ACTION, z.B. WRITE oder CALL_METHOD
    sEndpoint : WSTRING; // Endpunkt, z.B. opc.tcp://192.168.1.12:4840
    sNode : WSTRING; // Knoten, z.B. /Objects/X/Y,1:s=Y
END_VAR

F_OPC_REMOTE := CONCAT("opc_ua[", sAction, ";", sEndpoint, "#;", sNode, "]");

END_FUNCTION
```

1. `CONCAT` joins `opc_ua[`, `sAction`, `;`, `sEndpoint`, `#;`, `sNode` and `]`; the `#` terminates the endpoint.

Example

Call and result:

```
MyLib_OPC::F_OPC_REMOTE("WRITE", "opc.tcp://192.168.1.12:4840", "/Objects/X/Y,1:s=Y")  -
>  opc_ua[WRITE;opc.tcp://192.168.1.12:4840#;/Objects/X/Y,1:s=Y]
```

Usage

Used in `myOpcUaAddresses.gcf` (project `test_VV`, folder `const/OPC_UA`, described in `OPC_UA_Adressen.md`) as the `InitialValue` of a WSTRING constant:
`InitialValue="MyLib_OPC::F_OPC_REMOTE("WRITE", PC_B_OPC, Q1_NODE)"`
`Q1_REMOTE_WRITE` (station A writes to station B) uses `MyLib_OPC::F_OPC_REMOTE("WRITE", PC_B_OPC, Q1_NODE)`; the endpoint `PC_B_OPC` comes from `F_MGR_TO_OPC`.

The expression is evaluated at export time; the boot file therefore contains only the finished strings, no function calls.

Note: The arguments must be constants (literals or other constants of the `.gcf`), because evaluation happens at export time, not at runtime.

Summary

`F_OPC_REMOTE` is a small IEC 61499 function of the library `MyLib_OPC` that assembles the OPC-UA ID for a node on another station.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.1.8 OSCAT

2.1.8.1 Basic

Data_types

CONSTANTS_MATH

CONSTANTS_MATH

Structure

Name	Type	Initial Value	Comment
PI	REAL	3.14159265358979323846264338327950288	Circle number PI
PI2	REAL	6.28318530717958647692528676655900576	PI*2
PI4	REAL	12.56637061435917295385057353311801152	PI*4
PI05	REAL	1.5707963267949	PI/2
PI025	REAL	0.785398163397448	PI/4
PI_INV	REAL	0.318309886183791	1/PI
EUL	REAL	2.71828182845904523536028747135266249	Euler constant e
E_INV	REAL	0.367879441171442	1/e
SQ2	REAL	1.4142135623731	root of 2
FACTS	DINT	[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600]	Factorials Array 0..12

Globals

Constants

Global constants

Global Constants

Name	Type	Initial Value
MATH	OSCAT::Basic::Data_types::CONSTANTS_MATH	

POUs

Engineering

Control

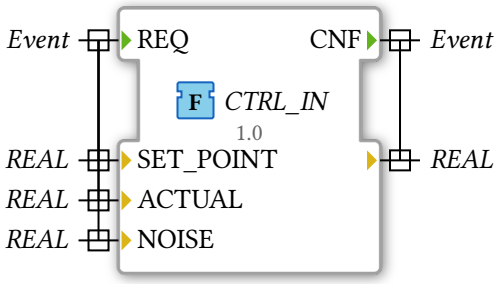
🎧 Podcast

- Eclipse 4diac: Revolutionizing Industrial Control with Open-Source Cyber-Physical Systems
- Simplifying Industrial Control: Your Deep Dive into Eclipse 4diac and IEC 61499
- The Future of Industrial Control: Decoding IEC 61499
- Decoding Industrial Control: Function Blocks, Object-Oriented Principles, and the Power of IEC 61499
- IEC 61499 vs. 61131: Unpacking the Architectural Revolution of Distributed Control and Industry 4.0

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

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

CTRL_IN



calculate the process error DIFF

Interface

Event inputs

Name	Comment	With
REQ		SET_POINT, ACTUAL, NOISE

Event Outputs

Name	Comment	With
CNF		

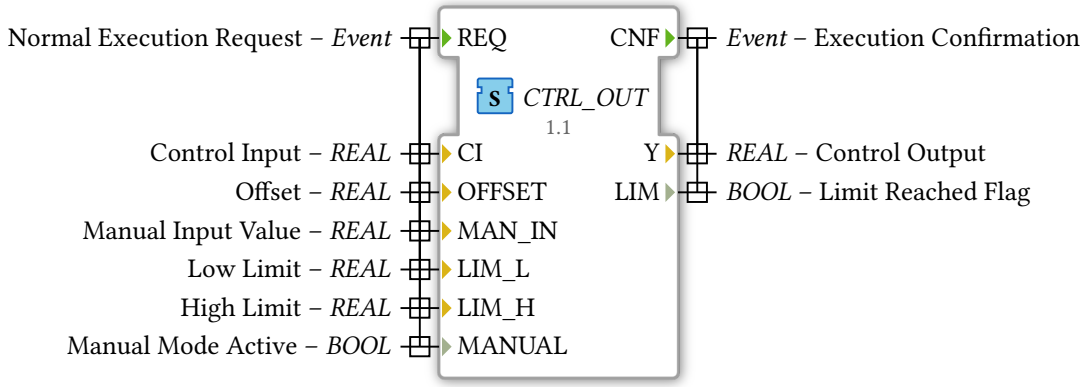
Input Vars

Name	Type	Comment
SET_POINT	REAL	
ACTUAL	REAL	
NOISE	REAL	

Output Vars

Name	Type	Comment
	REAL	

CTRL_OUT



Control Output with Limit and Manual Override CTRL_OUT adds to the CI input the value of OFFSET and returns the result to Y if MANUAL = FALSE. If MANUAL is TRUE at output Y the input value of MAN_IN + OFFSET is issued. Y is always limited to the boundaries defined by LIM_L and LIM_H. If Y reaches one of the limits, then the output LIM is TRUE. CTRL_OUT can be used to build your own rule modules.

Interface

Event inputs

Name	Comment	With
REQ	Normal Execution Request	CI, OFFSET, MAN_IN, LIM_L, LIM_H, MANUAL

Event Outputs

Name	Comment	With
CNF	Execution Confirmation	Y, LIM

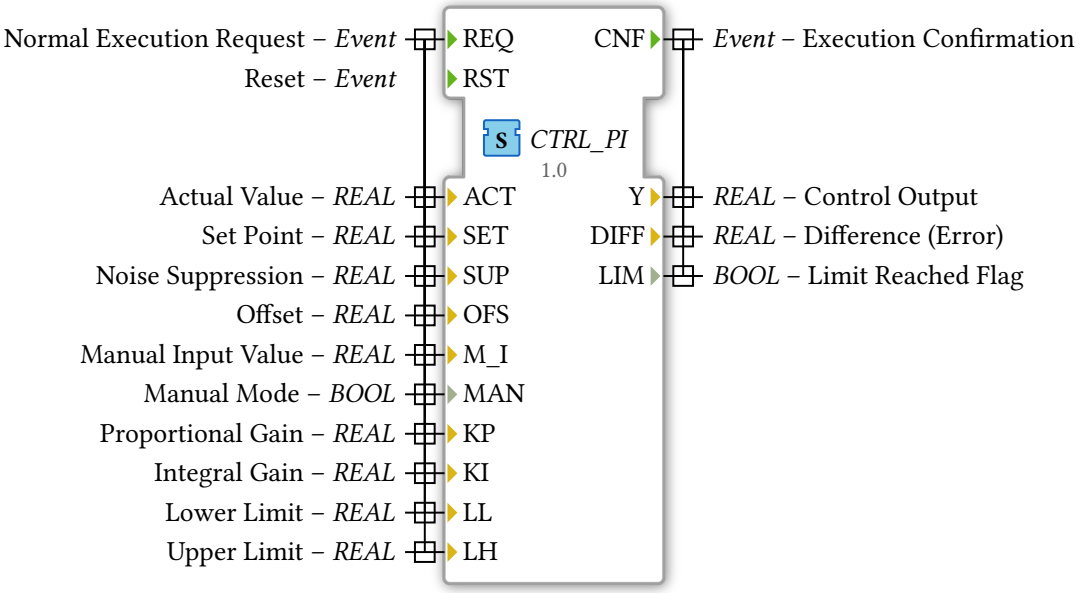
Input Vars

Name	Type	Comment
CI	REAL	Control Input
OFFSET	REAL	offset
MAN_IN	REAL	Manual Input Value
LIM_L	REAL	Low Limit
LIM_H	REAL	High Limit
MANUAL	BOOL	Manual Mode Active

Output Vars

Name	Type	Comment
Y	REAL	Control Output
LIM	BOOL	Limit Reached Flag

CTRL_PI



FT_PI is a PI controller with manual functionality. version 2.0 June 30, 2008 programmer hugo tested by oscat The PID controller works according to the fomula Y = e *(KP+ KI * INTEG(e)) + offset, while e = set_point - actual. a rst will reset all internal data, while a switch to manual will cause the controller to follow the function Y = manual_in + offset. limit_h and Limit_l set the possible output range of Y. The output flags lim will signal that the output limits are active and overflow will signal that the integrator has reached its limits.

since rev 1.1 the “trapezoid rule is used for more accuracy. rev 1.2 added selective integratin which means the integrative component is only active within a small range of the target value this avoids the integrator to go to limits while an input setpoint change happened and is only causing overshoots. the int_band is by default 100 which means the int is active all the time and if set to for example to 0.1 the integrator is only active while the input is between 0.9 and 1.1 of the set_point value.

default values for KP = 1, TN = 1, TV = 1, LIMIT_L = −1000, LIMIT_H = +1000.

Interface

Event inputs

Name	Comment	With
REQ	Normal Execution Request	ACT, SET, SUP, OFS, M_I, MAN, KP, KI, LL, LH
RST	Reset	

Event Outputs

Name	Comment	With
CNF	Execution Confirmation	Y, DIFF, LIM

Input Vars

Name	Type	Initial Value	Comment
ACT	REAL		Actual Value
SET	REAL		Set Point
SUP	REAL		Noise Suppression
OFS	REAL		offset

M_I	REAL		Manual Input Value
MAN	BOOL		Manual Mode
CP	REAL	1.0	Proportional Gain
AI	REAL	1.0	Integral Gain
LL	REAL	−1000.0	Lower Limit
LH	REAL	1000.0	Upper Limit

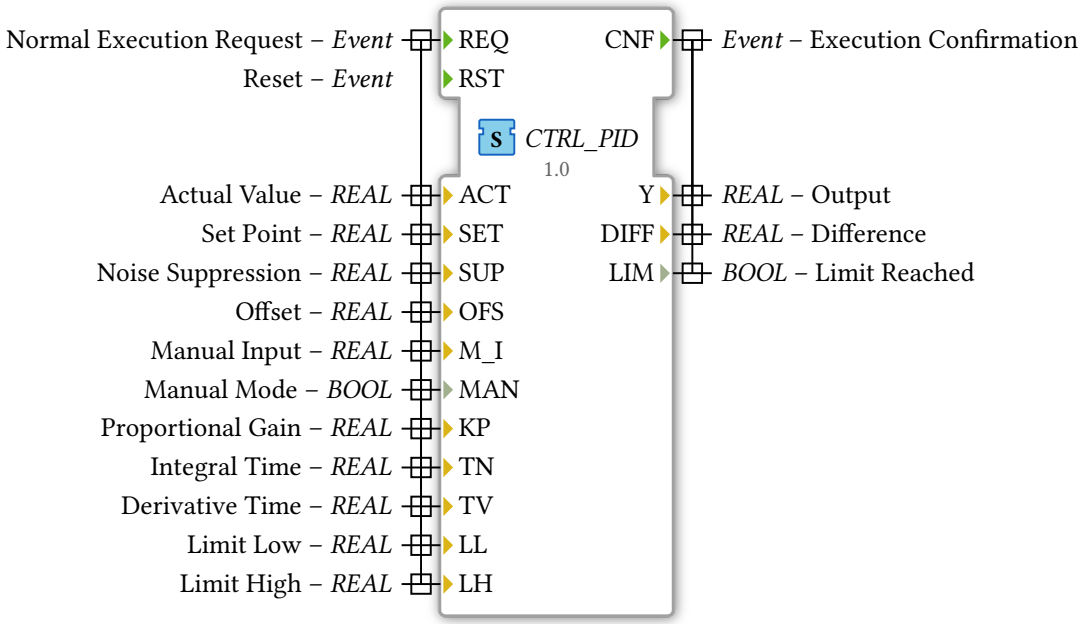
Output Variables

Name	Type	Comment
Y	REAL	Control Output
DIFF	REAL	Difference (Error)
LIM	BOOL	Limit Reached Flag

🛠️ Related Exercises

- Exercise_152

CTRL_PID



FT_PI is a PI controller with manual functionality. version 2.0 June 30th 2008 programmer hugo tested by oscat The PID controller works according to the fomula Y = e *(KP+ KI * INTEG(e)) + offset, while e = set_point - actual. a rst will reset all internal data, while a switch to manual will cause the controller to follow the function Y = manual_in + offset. limit_h and Limit_l set the possible output range of Y. The output flags lim will signal that the output limits are active.

Interface

Event inputs

Name	Comment	With
REQ	Normal Execution Request	ACT, SET, SUP, OFS, M_I, MAN, KP, TN, TV, LL, LH
RST	Reset	

Event Outputs

Name	Comment	With
CNF	Execution Confirmation	Y, DIFF, LIM

Input Vars

Name	Type	Initial Value	Comment
ACT	REAL		Actual Value
SET	REAL		Set Point
SUP	REAL		Noise Suppression
OFS	REAL		offset
M_I	REAL		Manual Input
MAN	BOOL		Manual Mode
CP	REAL	1.0	Proportional Gain
TN	REAL	1.0	Integral Time
TV	REAL	1.0	Derivative Time
LL	REAL	−1000.0	Limit Low
LH	REAL	1000.0	Limit High

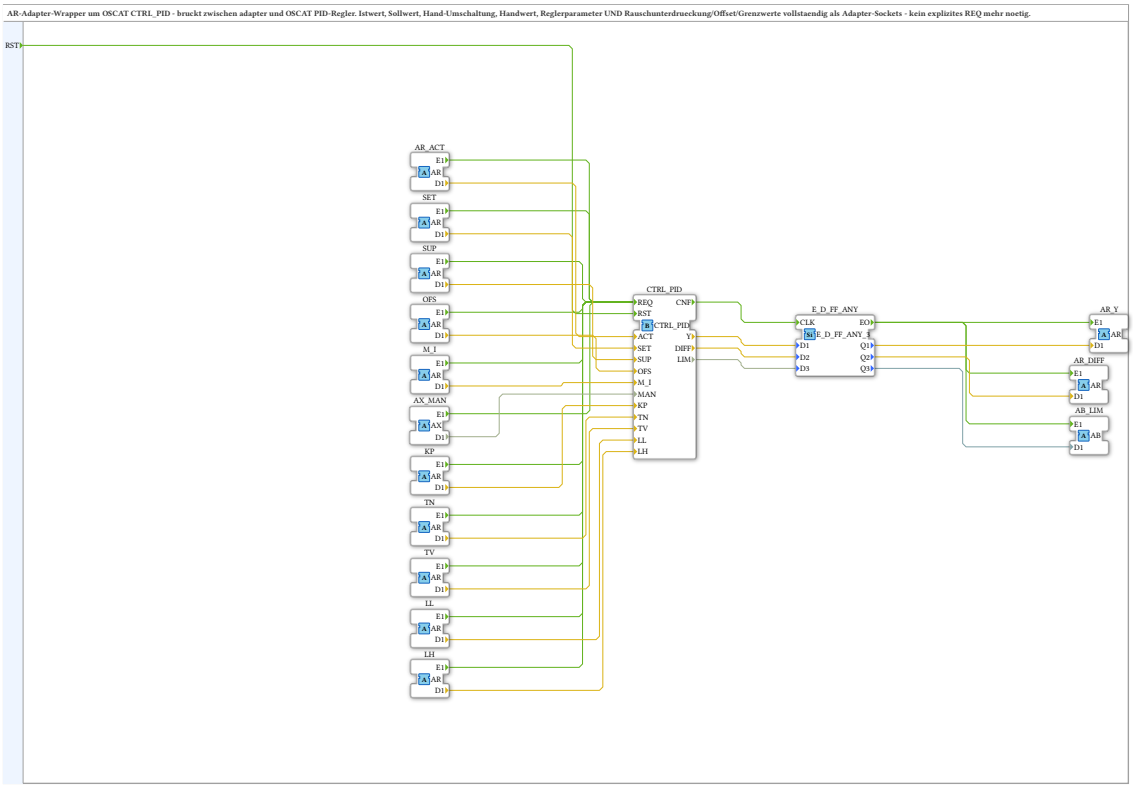
Output Vars

Name	Type	Comment
Y	REAL	Output
DIFF	REAL	Difference
LIM	BOOL	Limit Reached

🛠️ Related exercises

- Uebung_153

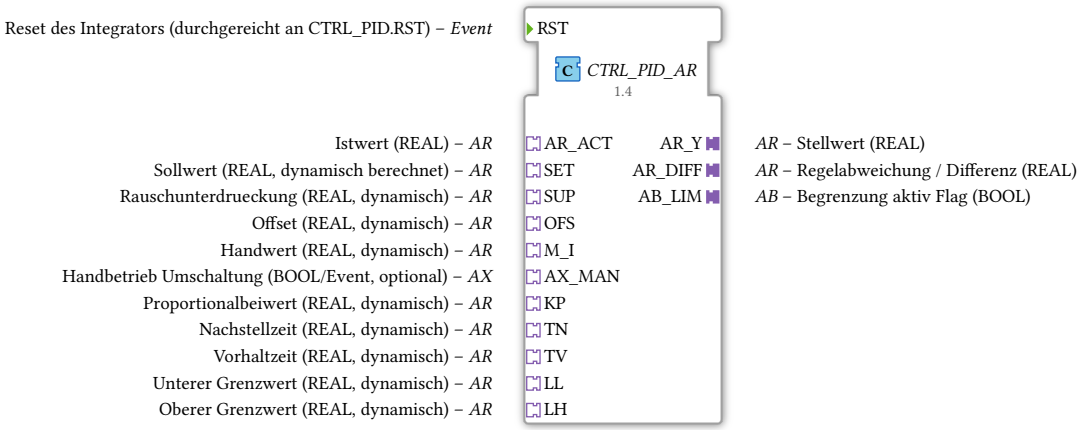
CTRL_PID_AR



• – ○ • – ○ • – ○ •

Introduction

`CTRL_PID_AR` is an AR adapter wrapper around the OSCAT PID controller `CTRL_PID` (`OSCAT::Basic::POUs::Engineering::Control::CTRL_PID`) and bridges between adapters and the OSCAT controller. Actual value, set point, manual switch, manual value, controller parameters as well as noise suppression, offset and limits are all implemented as adapter sockets - in the same order as the input variables of the wrapped `CTRL_PID`. No explicit `REQ` is needed, since each socket triggers `CTRL_PID.REQ` itself when a new value arrives.



Interface Structure

Event Inputs

- **RST**: Reset of the integrator (passed through to `CTRL_PID.RST`)

Event Outputs

No direct event outputs - events run via the adapter plugs

Data Inputs

No data inputs - all inputs are adapter sockets

Data Outputs

No data outputs - all outputs are adapter plugs

Adapters

Sockets (inputs); `E1` triggers `CTRL_PID.REQ`, `D1` supplies the value:

- **AR_ACT** (`AR`): Actual value (REAL)
- **SET** (`AR`): Set point (REAL, dynamically calculated)
- **SUP** (`AR`): Noise suppression (REAL, dynamic)
- **OFS** (`AR`): Offset (REAL, dynamic)
- **M_I** (`AR`): Manual value (REAL, dynamic)
- **AX_MAN** (`AX`): Manual mode switch (BOOL/event, optional)
- **KP** (`AR`): Proportional gain (REAL, dynamic)
- **TN** (`AR`): Integral time (REAL, dynamic)
- **TV** (`AR`): Derivative time (REAL, dynamic)
- **LL** (`AR`): Lower limit (REAL, dynamic)
- **LH** (`AR`): Upper limit (REAL, dynamic)

Plugs (outputs):

- **AR_Y** (`AR`): Control output (REAL)
- **AR_DIFF** (`AR`): Control error (`DIFF = SET - ACT`, REAL)
- **AB_LIM** (`AB`): Limit active flag (`LIM`, BOOL)

Function Blocks (FBs) Used

- **CTRL_PID**: `OSCAT::Basic::POUs::Engineering::Control::CTRL_PID`
- **E_D_FF_ANY**: `iec61499::events::E_D_FF_ANY_3` - 3-fold data latch for `Y`, `DIFF`, `LIM`

Program Flow and Connections

1. Every `E1` of the sockets `AR_ACT`, `SET`, `SUP`, `OFS`, `M_I`, `AX_MAN`, `KP`, `TN`, `TV`, `LL`, `LH` ⟶ `CTRL_PID.REQ`.
2. `D1` of the sockets ⟶ same-named input of `CTRL_PID` (`AR_ACT.D1` ⟶ `ACT`, `AX_MAN.D1` ⟶ `MAN`, otherwise identical names).
3. `RST` ⟶ `CTRL_PID.RST`.
4. `CTRL_PID.CNF` ⟶ `E_D_FF_ANY.CLK`; `CTRL_PID.Y/DIFF/LIM` ⟶ `E_D_FF_ANY.D1/D2/D3`.
5. `E_D_FF_ANY.EO` ⟶ `AR_Y.E1`, `AR_DIFF.E1`, `AB_LIM.E1`; `Q1/Q2/Q3` ⟶ `AR_Y.D1`, `AR_DIFF.D1`, `AB_LIM.D1`.

Technical Details

- The latch (`E_D_FF_ANY`) triggers the outputs only after the controller's `CNF`; there is no external `CNF` event any more.
- Each input socket triggers a calculation; if several values change simultaneously, multiple `REQ` occur.
- The sockets are sorted in the order of the `InputVars` of `CTRL_PID` (version 1.4).

Application Scenarios

- PID control in adapter networks, e.g. speed control of the pickup with a dynamically calculated set point and `AX_MAN` from an automatic/manual toggle

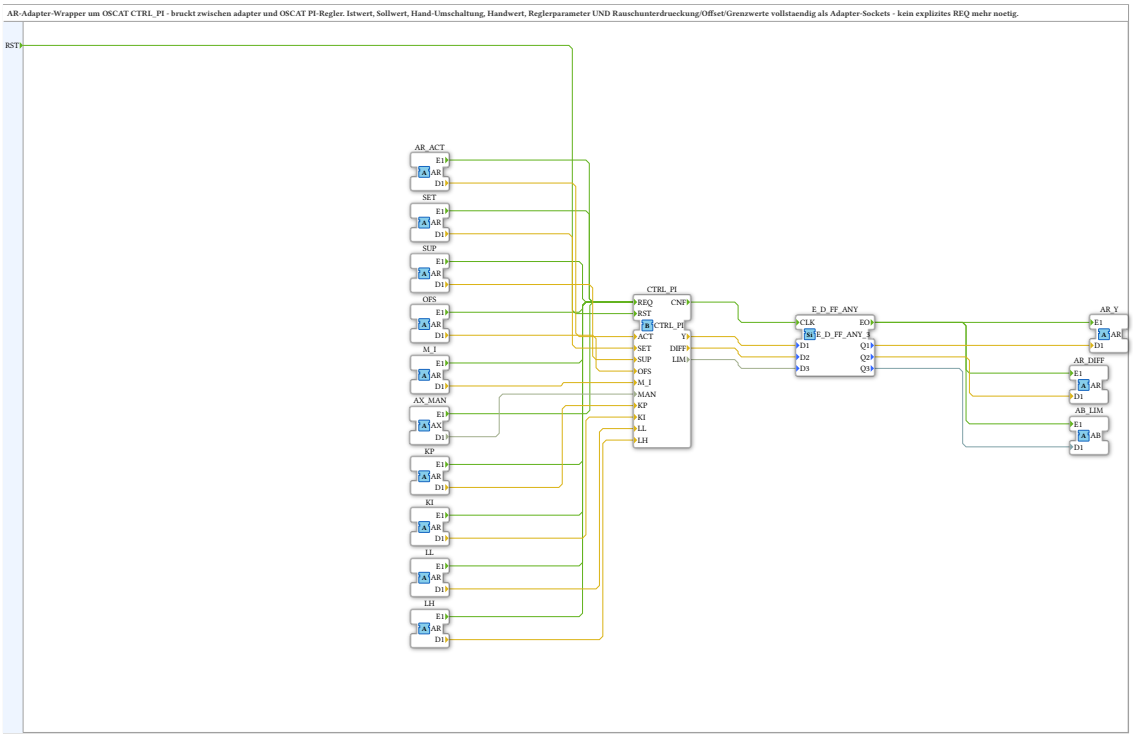
🛠️ Related Exercises

- Uebung_153_AX

Summary

`CTRL_PID_AR` connects the OSCAT PID controller entirely via AR/AX adapters, so that values flow only through adapter connections and no explicit `REQ` is needed.

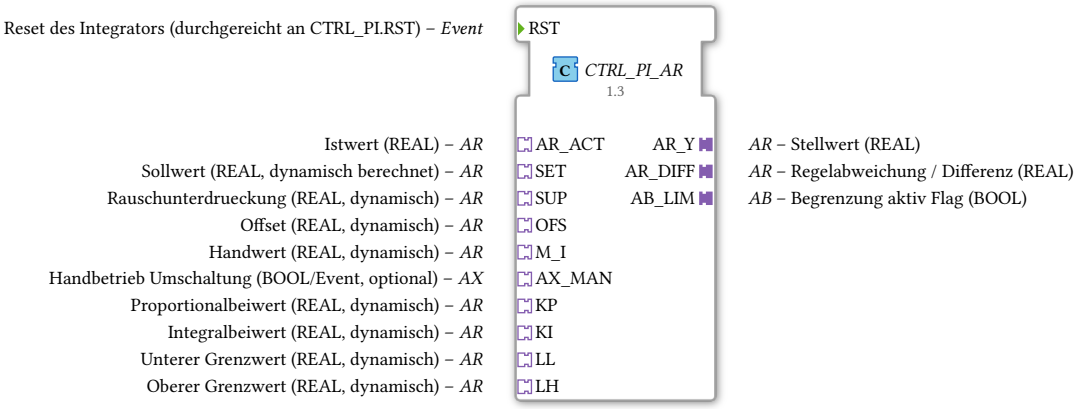
CTRL_PI_AR



• – ○ • – ○ • – ○ •

Introduction

`CTRL_PI_AR` is an AR adapter wrapper around the OSCAT PI controller `OSCAT::Basic::POUs::Engineering::Control::CTRL_PI`. It bridges between adapters and OSCAT without either base library depending on the other. Actual value, set point, noise suppression, offset, manual value, manual mode switch, controller parameters and limits are all implemented as adapter sockets; an explicit `REQ` is no longer needed because every socket triggers the controller itself.



Interface Structure

Event Inputs

- **RST**: reset of the integrator (passed on to `CTRL_PI.RST`)

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AR_ACT** (socket, AR): actual value (REAL)
- **SET** (socket, AR): set point (REAL, dynamically calculated)
- **SUP** (socket, AR): noise suppression (REAL, dynamic)
- **OFS** (socket, AR): offset (REAL, dynamic)
- **M_I** (socket, AR): manual value (REAL, dynamic)
- **AX_MAN** (socket, AX): switch to manual mode (optional)
- **KP** (socket, AR): proportional gain (REAL, dynamic)
- **KI** (socket, AR): integral gain (REAL, dynamic)
- **LL** (socket, AR): lower limit (REAL, dynamic)
- **LH** (socket, AR): upper limit (REAL, dynamic)
- **AR_Y** (plug, AR): control output (REAL)
- **AR_DIFF** (plug, AR): control error `DIFF = SET - ACT` (REAL)
- **AB_LIM** (plug, AB): flag 'limiting active' (BOOL)

Functionality

The network consists of the controller `CTRL_PI` (`OSCAT::Basic::POUs::Engineering::Control::CTRL_PI`) and a change filter `E_D_FF_ANY` (`iec61499::events::E_D_FF_ANY_3`):

- **Triggering**: the event `E1` of every socket (`AR_ACT`, `SET`, `SUP`, `OFS`, `M_I`, `AX_MAN`, `KP`, `KI`, `LL`, `LH`) goes to `CTRL_PI.REQ`. `RST` goes to `CTRL_PI.RST`.
- **Data**: `AR_ACT.D1` ⟶ `ACT`, `SET.D1` ⟶ `SET`, `SUP.D1` ⟶ `SUP`, `OFS.D1` ⟶ `OFS`, `M_I.D1` ⟶ `M_I`, `AX_MAN.D1` ⟶ `MAN`, `KP.D1` ⟶ `KP`, `KI.D1` ⟶ `KI`, `LL.D1` ⟶ `LL`, `LH.D1` ⟶ `LH` of the controller.
- **Output**: `CTRL_PI.CNF` ⟶ `E_D_FF_ANY.CLK`; `CTRL_PI.Y`, `CTRL_PI.DIFF` and `CTRL_PI.LIM` go to `E_D_FF_ANY.D1..D3`. Only if at least one of the three values changes does `E_D_FF_ANY.EO` emit the event to `AR_Y.E1`, `AR_DIFF.E1` and `AB_LIM.E1`, and `Q1..Q3` go to `AR_Y.D1`, `AR_DIFF.D1` and `AB_LIM.D1`.

Technical Details

- The `CNF` event of the controller is now internal only (for the latch of `Y/DIFF/LIM`); there is no external `REQ/CNF` any more
- Every socket triggers the controller, so several controller runs are possible when several values arrive at the same time
- The sockets are sorted in the order of the `CTRL_PI` input variables: `AR_ACT`, `SET`, `SUP`, `OFS`, `M_I`, `AX_MAN`, `KP`, `KI`, `LL`, `LH`
- The adapter bridge resides in the library `OSCAT_adapter`, which depends on `adapter` and `OSCAT`; thus these two need not depend on each other

Application Scenarios

- PI control in purely adapter-based networks, e.g. speed or rotational speed control with a dynamically calculated set point
- Controllers whose parameters (KP, KI) and limits can be changed at runtime via VT/OPC-UA

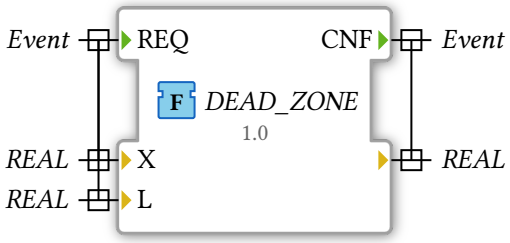
🛠️ Related Exercises

- Uebung_152_AX

Conclusion

`CTRL_PI_AR` makes the OSCAT PI controller adapter-capable: all inputs and outputs are AR/AX/AB adapters, every input triggers the calculation, and results are output only on change.

DEAD_ZONE



dead_zone2 is a linear transfer function which follows a linear function except for x is close to 0.

Interface

Event inputs

Name	Comment	With
REQ		X, L

Event Outputs

Name	Comment	With
CNF		

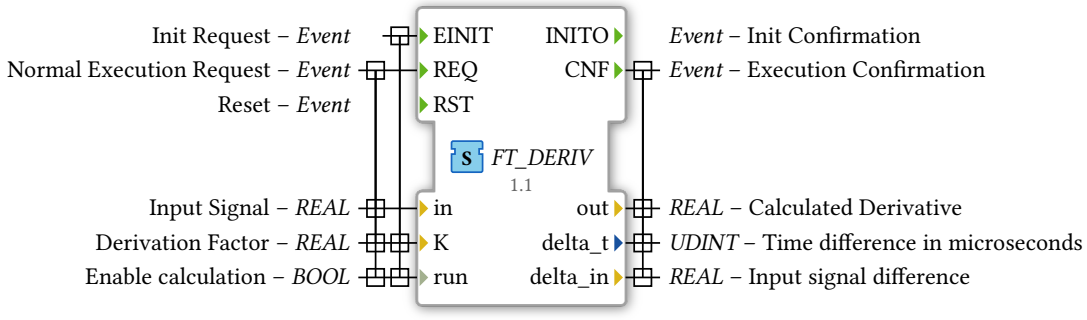
Input Vars

Name	Type	Comment
X	REAL	
L	REAL	

Output Vars

Name	Type	Comment
	REAL	

FT_DERIV



Calculate derivative over signal 'in' with Factor 'K'

Interface

Event inputs

Name	Comment	With
EINIT	Init Request	K, run

REQ	Normal Execution Request	in, K, run
RST	Reset	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	
CNF	Execution Confirmation	out, delta_t, delta_in

Input Vars

Name	Type	Initial Value	Comment
in	REAL		Input signal
K	REAL	1.0	Derivation Factor
run	BOOL	1	Enable calculation

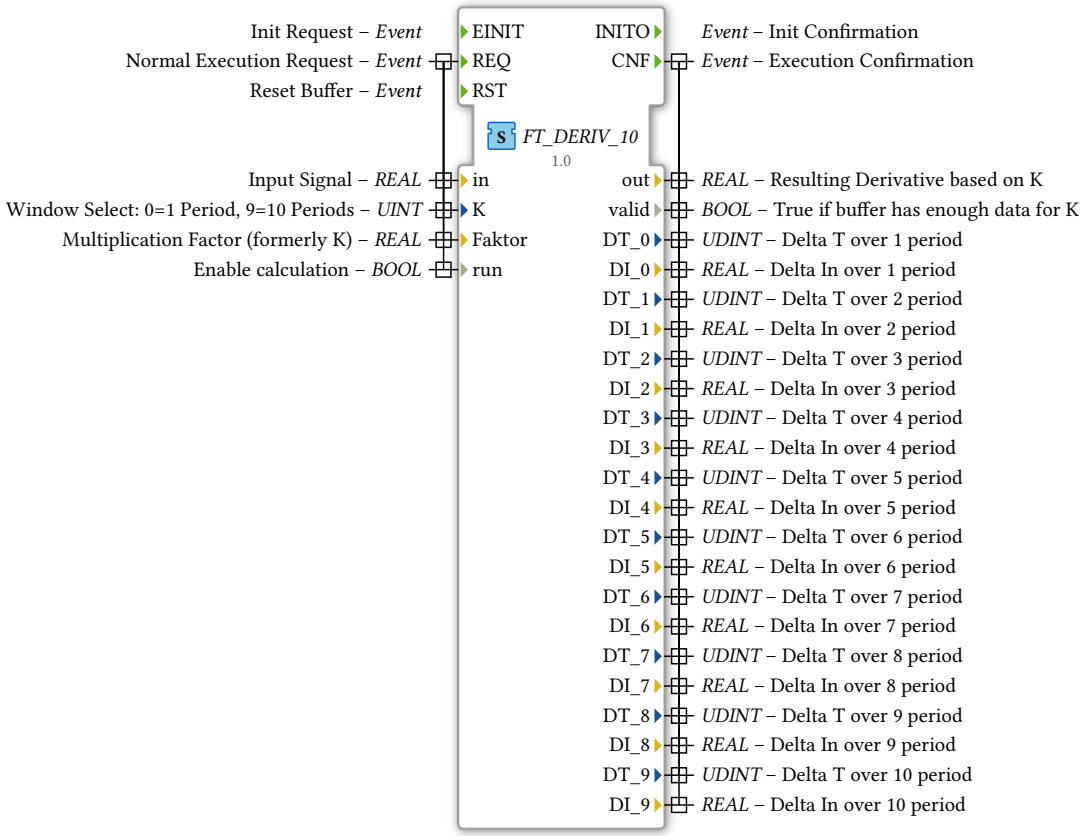
Output Vars

Name	Type	Comment
out	REAL	Calculated Derivative
delta_t	UDINT	Time difference in microseconds
delta_in	REAL	Input signal difference

🛠️ Related exercises

- Uebung_151
- Uebung_151_AX

FT_DERIV_10



Derivative with variable window size (0..9 periods)

Interface

Event inputs

Name	Comment	With
EINIT	Init Request	
REQ	Normal Execution Request	in, K, Faktor, run
RST	Reset Buffer	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	
CNF	Execution Confirmation	out, valid, DT_0, DI_0, DT_1, DI_1, DT_2, DI_2, DT_3, DI_3, DT_4, DI_4, DT_5, DI_5, DT_6, DI_6, DT_7, DI_7, DT_8, DI_8, DT_9, DI_9

Input Vars

Name	Type	Initial Value	Comment
in	REAL		Input signal
K	UINT	0	Window Select: 0=1 Period, 9=10 Periods
Faktor	REAL	1.0	Multiplication Factor (formerly K)

run	BOOL	1	Enable calculation

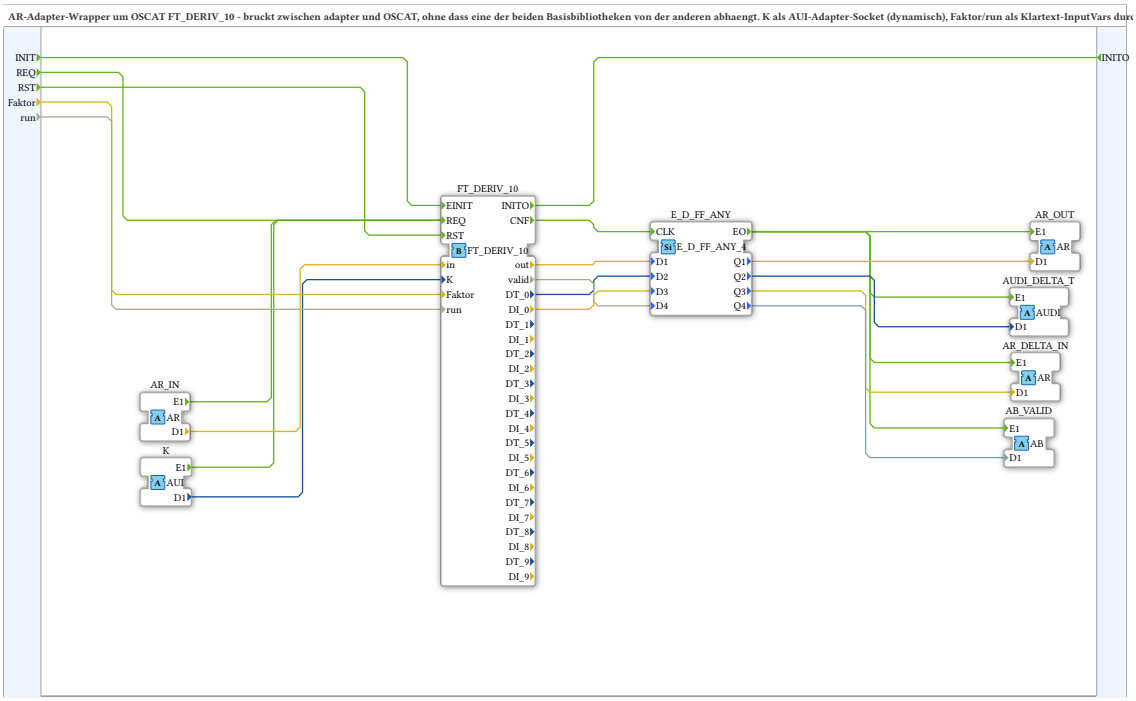
Output Vars

Name	Type	Comment
out	REAL	Resulting Derivative based on K
valid	BOOL	True if buffer has enough data for K
DT_0	UDINT	Delta T over 1 period
DI_0	REAL	Delta In over 1 period
DT_1	UDINT	Delta T over 2 period
DI_1	REAL	Delta In over 2 period
DT_2	UDINT	Delta T over 3 period
DI_2	REAL	Delta In over 3 period
DT_3	UDINT	Delta T over 4 period
DI_3	REAL	Delta In over 4 period
DT_4	UDINT	Delta T over 5 period
DI_4	REAL	Delta In over 5 period
DT_5	UDINT	Delta T over 6 period
DI_5	REAL	Delta In over 6 period
DT_6	UDINT	Delta T over 7 period
DI_6	REAL	Delta In over 7 period
DT_7	UDINT	Delta T over 8 period
DI_7	REAL	Delta In over 8 period
DT_8	UDINT	Delta T over 9 period
DI_8	REAL	Delta In over 9 period
DT_9	UDINT	Delta T over 10 period
DI_9	REAL	Delta In over 10 period

🛠️ Related exercises

- Uebung_152
- Uebung_153

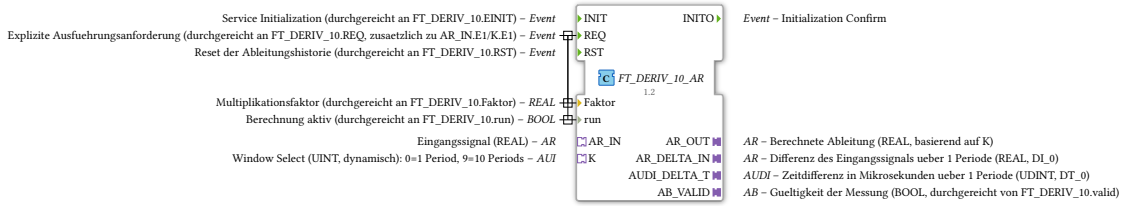
FT_DERIV_10_AR



• – ○ • – ○ • – ○ •

Introduction

`FT_DERIV_10_AR` is an AR adapter wrapper around the OSCAT block `OSCAT::Basic::POUs::Engineering::Control::FT_DERIV_10` (derivative with a selectable window of 1 to 10 periods). It bridges adapters and OSCAT without either base library depending on the other. Compared to `FT_DERIV_AR`, the window `K` is a dynamic `AUI` adapter socket. `Faktor` and `run` remain plain input variables.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Service initialization (passed through to `FT_DERIV_10.EINIT`)
- **REQ**: explicit execution request (passed through to `FT_DERIV_10.REQ`, in addition to `AR_IN.E1`/`K.E1`)
  - **Associated data**: `Faktor`, `run`
- **RST**: Reset of the derivative history (passed through to `FT_DERIV_10.RST`)

Event Outputs

- **INITO** (`EInit`): Initialization confirmation (from `FT_DERIV_10.INITO`)

Data Inputs

- **Faktor** (`REAL`, initial value `1.0`): Multiplication factor (passed through to `FT_DERIV_10.Faktor`)
- **run** (`BOOL`, initial value `TRUE`): Calculation active (passed through to `FT_DERIV_10.run`)

Data Outputs

None. All result values are output via adapter plugs.

Adapters

Interface	Direction	Adapter type	Data type	Description
AR_IN	Socket	AR	REAL	Input signal
K	Socket	AUI	UINT	Window select (dynamic): 0 = 1 period, 9 = 10 periods
AR_OUT	Plug	AR	REAL	Calculated derivative (FT_DERIV_10.out, based on K)
AR_DELTA_IN	Plug	AR	REAL	Input signal difference over 1 period (DI_0)
AUDI_DELTA_T	Plug	AUDI	UDINT	Time difference in microseconds over 1 period (DT_0)
AB_VALID	Plug	AB	BOOL	Measurement validity (FT_DERIV_10.valid)

Function Blocks (FBs) Used

- **FT_DERIV_10** (OSCAT::Basic::POUs::Engineering::Control::FT_DERIV_10): derivative calculation, see FT_DERIV_10
- **E_D_FF_ANY** (iec61499::events::E_D_FF_ANY_4): four-channel change filter/synchronisation of the results

How it works

- AR_IN.E1, K.E1 and REQ all lead to FT_DERIV_10.REQ
- AR_IN.D1 → FT_DERIV_10.in; K.D1 → FT_DERIV_10.K; Faktor → FT_DERIV_10.Faktor; run → FT_DERIV_10.run
- INIT → FT_DERIV_10.EINIT; FT_DERIV_10.INITO → INITO; RST → FT_DERIV_10.RST
- FT_DERIV_10.CNF → E_D_FF_ANY.CLK; the results out, DT_0, DI_0, valid go to E_D_FF_ANY.D1 to D4
- E_D_FF_ANY.Q1...Q4 → AR_OUT.D1, AUDI_DELTA_T.D1, AR_DELTA_IN.D1, AB_VALID.D1; E_D_FF_ANY.EO triggers E1 of all four plugs

Technical Details

- **Pure adapter boundary**: All main outputs of FT_DERIV_10 are output as dedicated adapters; no plain event/data leaves the block (except INITO)
- **K as adapter** (version 1.2, 2026-10-03): K.E1 triggers a calculation just like AR_IN.E1, K.D1 supplies the value; K is therefore no longer bound to REQ via With
- **Explicit REQ** (version 1.1, 2026-09-20): bugfix so that Faktor and run are read reliably via a With binding (per IEC 61499, unbound variables are not read reliably); it also allows manual or periodic execution
- FT_DERIV_10 is a pure OSCAT function without an adapter interface; the bridge lives in the third library project OSCAT_adapter

Application Scenarios

- Frequency/speed calculation from a pulse counter value (AR_IN, e.g. from AD_TO_AR_NUM.AR_OUT) with a window selectable at runtime
- Passing the derivative on, e.g. to CTRL_PID_AR.AR_ACT

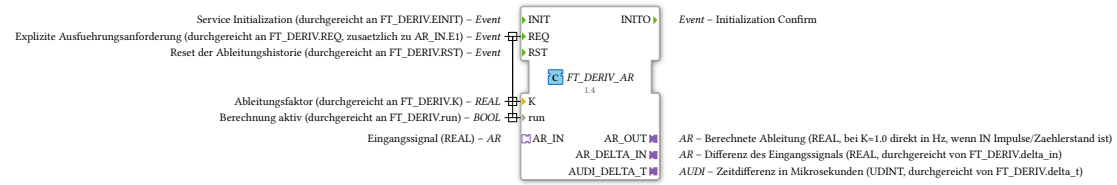
Summary

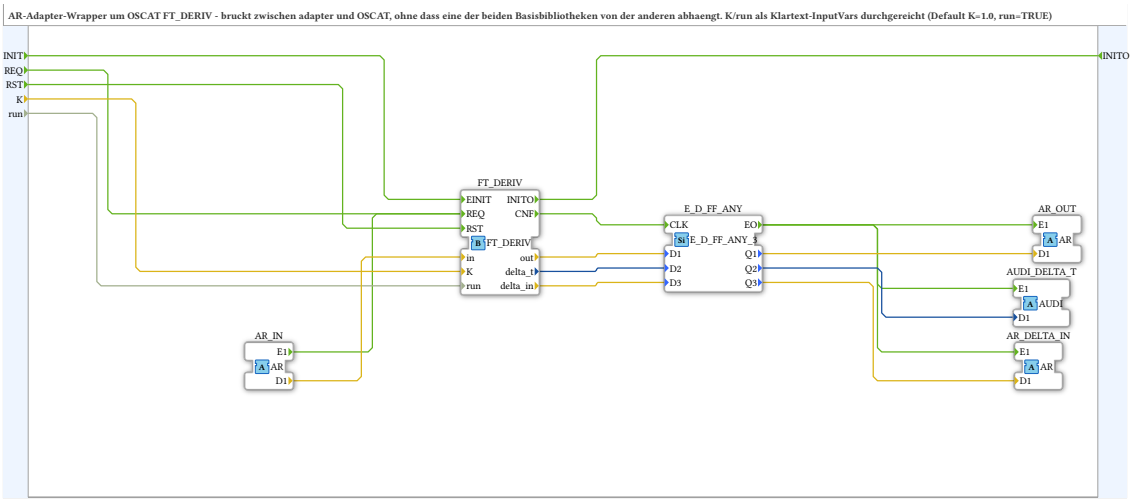
FT_DERIV_10_AR encapsulates FT_DERIV_10 behind a pure adapter interface with a dynamically selectable window K and provides derivative, input difference, time difference and validity as adapters.

🛠️ Related Exercises

- Uebung_152_AX
- Uebung_153_AX

FT_DERIV_AR





• — ○ • — ○ • — ○ •

Introduction

`FT_DERIV_AR` is an adapter wrapper around the OSCAT function block `OSCAT::Basic::POUs::Engineering::Control::FT_DERIV`. It provides a clean adapter-based interface for the derivative calculation so that the block can be used in subapplications that communicate through unidirectional `AR` / `AUDI` adapters.

The input signal is received through an `AR_IN` socket. The calculated derivative and the internal intermediate values are exposed through three adapter plugs. The configuration values `K` and `run` are kept as plain input variables so that they can be set as parameters when the block is instantiated.

The wrapper also solves a library-coupling problem: the generic `adapter` library and the `OSCAT` library can remain independent, while the bridge itself is encapsulated in a dedicated wrapper type.

Interface Structure

Event Inputs

Event	Type	Description
`INIT`	`EInit`	Initialization request; passed through to `FT_DERIV.EINIT`.
`RST`	`Event`	Resets the derivative history; passed through to `FT_DERIV.RST`.

Event Outputs

Event	Type	Description
`INITO`	`EInit`	Initialization confirmation; passed back from `FT_DERIV.INITO`.

Data Inputs

Name	Type	Initial Value	Description
`K`	`REAL`	`1.0`	Derivation factor; passed through to `FT_DERIV.K`.
`run`	`BOOL`	`TRUE`	Calculation enable; passed through to `FT_DERIV.run`.

Data Outputs

This block has no plain data outputs. All calculation results are exposed through adapter plugs.

Adapters

Kind	Name	Adapter Type	Description
Socket	AR_IN	adapter::types::unidirectional::AR	REAL input signal; E1 triggers the calculation and D1 carries the input value.
Plug	AR_OUT	adapter::types::unidirectional::AR	Calculated derivative (out, REAL).
Plug	AR_DELTA_IN	adapter::types::unidirectional::AR	Difference of the input signal (delta_in, REAL).
Plug	AUDI_DELTA_T	adapter::types::unidirectional::AUDI	Time difference in microseconds (delta_t, UDINT).

Functionality

The internal network consists of the OSCAT FB FT_DERIV and a three-channel D flip-flop E_D_FF_ANY_3. The wrapper orchestrates the event and data flow as follows:

1. **Initialization** – INIT is forwarded to FT_DERIV.EINIT. The internal block performs its initialization and returns INITO. If INIT is left unconnected, the EInit event fires automatically once at deployment.
2. **Input request** – When the AR_IN adapter receives an event (AR_IN.E1), the corresponding REAL value (AR_IN.D1) is written to FT_DERIV.in, and FT_DERIV.REQ is triggered.
3. **Derivative calculation** – The internal FT_DERIV calculates the derivative using the current input, the configured factor K, and the enable flag run.
4. **Output latching** – When FT_DERIV.CNF is emitted, the internal E_D_FF_ANY_3 latches out, delta_t, and delta_in into its three channels. The flip-flop then emits its output event EO, which is routed to all three plugs: AR_OUT.E1, AR_DELTA_IN.E1, and AUDI_DELTA_T.E1. The corresponding D1 values are taken from the latched Q1, Q2, and Q3 outputs.
5. **Reset** – RST is passed through to FT_DERIV.RST, clearing the internal derivative history.

Because all three values are latched from the same CNF event, downstream consumers receive a consistent snapshot of the calculation cycle.

Technical Features

- **Adapter-based result boundary** – The three calculation results are available only through AR / AUDI plugs; no plain data output is needed.
- **Composite implementation** – Contains OSCAT::Basic::POUs::Engineering::Control::FT_DERIV and iec61499::events::E_D_FF_ANY_3.
- **Simultaneous output update** – The internal three-channel D flip-flop ensures that AR_OUT, AR_DELTA_IN, and AUDI_DELTA_T are updated together.
- **Decoupled library design** – The wrapper is placed in a dedicated library project that depends on both adapter and OSCAT, avoiding a direct dependency between the two base libraries.
- **Pass-through lifecycle/reset** – INIT, INITO, and RST behave exactly like the underlying FT_DERIV interface.
- **Configuration inputs** – K and run are plain InputVars with default values 1.0 and TRUE.
- **Adapter-based wiring** – The block can be connected to other AR / AUDI blocks using ordinary AdapterConnections; no raw .E1 / .D1 access from a parent network is needed.

State Overview

FT_DERIV_AR is a composite FB type, so it does not define an explicit ECC state diagram. Its runtime behavior can be described by the following phases:

Phase	Description
Initialization	INIT is forwarded to FT_DERIV.EINIT; INITO confirms completion.
Ready / Waiting	The block is idle and waits for an AR_IN.E1 event.

Calculating	FT_DERIV.REQ is active; the internal derivative calculation has not yet completed.
Output Update	FT_DERIV.CNF arrives and the flip-flop latches/emits out, delta_t, and delta_in to the adapter plugs.
Reset	RST clears the internal input/time history stored in FT_DERIV.

When run is FALSE, the internal calculation is disabled and the output plugs are not updated with a new derivative.

Application Scenarios

- **Frequency / rate measurement** – If the input AR_IN receives a pulse counter or incremental counter value, AR_OUT delivers the derivative directly in Hz when K = 1.0.
- **Rate-of-change monitoring** – The block can monitor the velocity of a process value such as pressure, temperature, or fill level and make the result available to adapter-based alarm or supervision blocks.
- **Adapter-based subapplications** – In compositions that already use AR / AUDI adapters, FT_DERIV_AR can be integrated without falling back to plain event/data connections.
- **Plain value conversion** – If a downstream block requires a plain REAL value instead of an adapter, the AR_OUT plug can be connected to a generic conversion block such as adapter::conversion::unidirectional::AR_R_TO_REAL.
- **Batch or mode changes** – RST can be used to clear the derivative history when the process is restarted or when a new operating mode begins.

Comparison with Similar Blocks

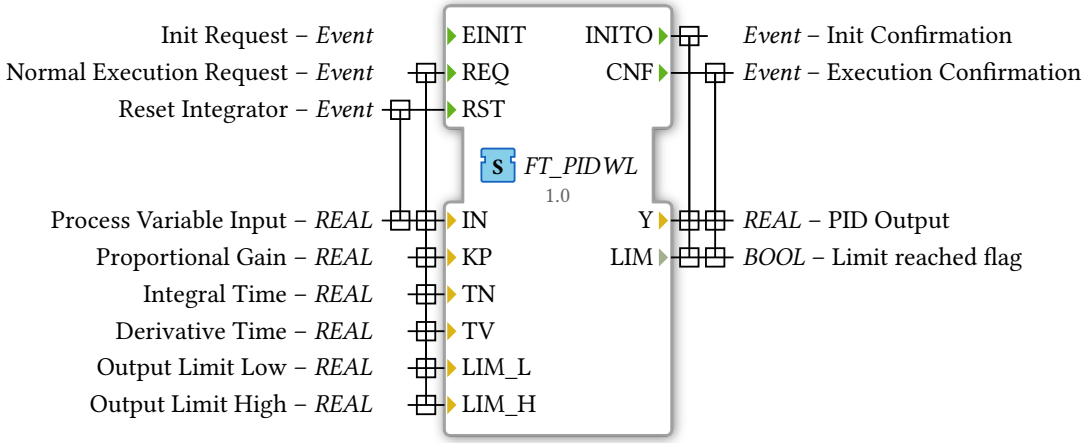
Block	Interface	Outputs	Typical Use
FT_DERIV	Plain events/data, no adapters	out, delta_t, delta_in	OSCAT-native networks where adapter boundaries are not required
FT_DERIV_AR	Adapter-based result boundary	AR_OUT, AR_DELTA_IN, AUDI_DELTA_T	Adapter-oriented 4diac subapplications and library decoupling

FT_DERIV_AR does not provide a new differentiation algorithm; it is a connectivity wrapper around the FT_DERIV calculation. Compared with using FT_DERIV directly, it adds adapter compatibility, output latching, and a clean separation between the adapter and OSCAT libraries.

Conclusion

FT_DERIV_AR is a practical adapter wrapper for the OSCAT FT_DERIV function block. It preserves the original calculation semantics while offering an adapter-friendly interface, consistent output latching, and clean library decoupling. This makes it well suited for modern 4diac subapplication designs that rely on unidirectional AR / AUDI adapters and want to avoid mixing raw .E1 / .D1 access or plain data pins across library boundaries.

FT_PIDWL



PI Controller with Anti-Windup version 1.3 November 13th 2009 programmer hugo tested by tobias FT_PIDWL is a PID controller with dynamic wind_up reset. The PID controller works according to the form Y = KP *(IN + KI * INTEG(e) + DERIV(e)). a rst will reset the integrator to 0 lim_h and Lim_l set the possible output range of the internal integrator. The output flags lim will signal that the output limits are active.

Interface

Event inputs

Name	Comment	With
EINIT	Init Request	
REQ	Normal Execution Request	IN, KP, TN, TV, LIM_L, LIM_H
RST	Reset Integrator	IN

Event Outputs

Name	Comment	With
INITO	Init Confirmation	Y, LIM
CNF	Execution Confirmation	Y, LIM

Input Vars

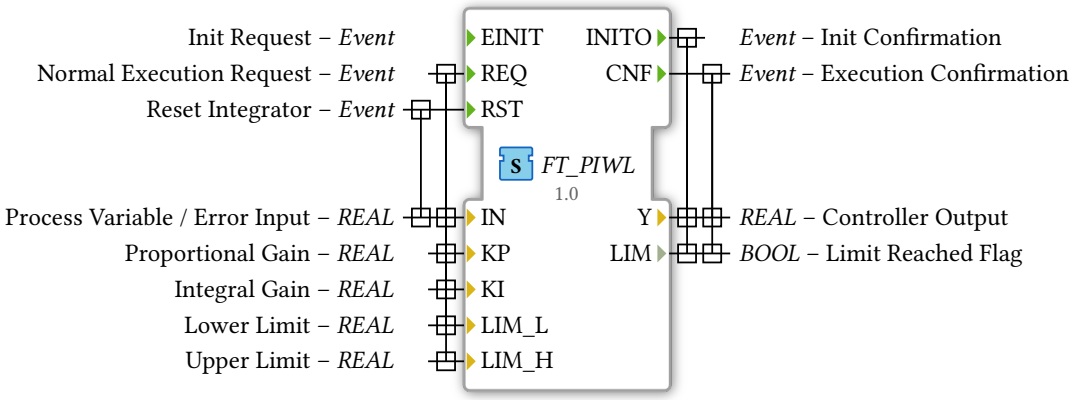
Name	Type	Initial Value	Comment
IN	REAL		Process Variable Input
KP	REAL	1.0	Proportional Gain
TN	REAL	1.0	Integral Time
TV	REAL	1.0	Derivative Time
LIM_L	REAL	−1.0E38	Output Limit Low
LIM_H	REAL	1.0E38	Output Limit High

Output Vars

Name	Type	Comment
Y	REAL	PID Output

LIM	BOOL	Limit reached flag

FT_PIWL



PI Controller with Anti-Windup version 1.3 mar 11 2009 programmer hugo tested by oscat FT_PIWL is a PI controller. The PID controller works according to the formula Y = IN *(KP+ KI * INTEG(e)). a rst will reset the integrator to 0 lim_h and Lim_l set the possible output range of the controller. the output flag lim will signal that the output limits are active. the integrator is equipped with anti wind-up circuitry which limits trhe total output ranke to lim_l and lim_h

default values for KP = 1, KI = 1, ILIM_L = −1E37, iLIM_H = +1E38.

Interface

Event inputs

Name	Comment	With
EINIT	Init Request	
REQ	Normal Execution Request	IN, KP, KI, LIM_L, LIM_H
RST	Reset Integrator	IN

Event Outputs

Name	Comment	With
INITO	Init Confirmation	Y, LIM
CNF	Execution Confirmation	Y, LIM

Input Vars

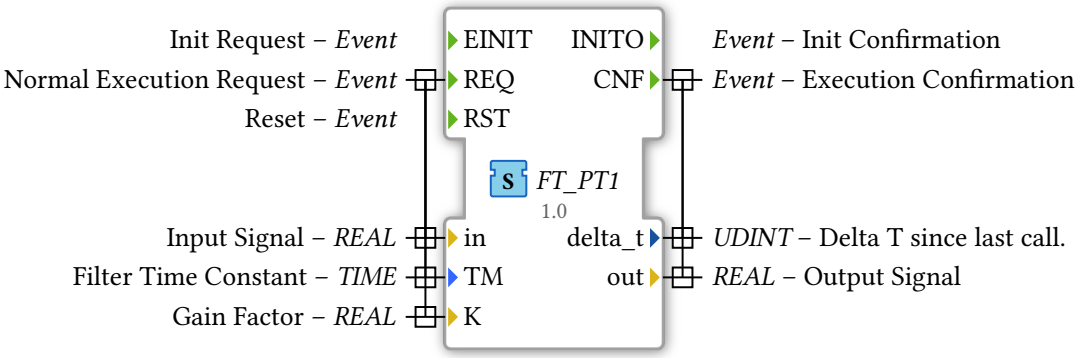
Name	Type	Initial Value	Comment
IN	REAL		Process Variable / Error Input
KP	REAL	1.0	Proportional Gain
KI	REAL	1.0	Integral Gain
LIM_L	REAL	−1.0E38	Lower Limit
LIM_H	REAL	1.0E38	Upper Limit

Output Vars

Name	Type	Comment
Y	REAL	Controller Output

LIM	BOOL	Limit Reached Flag

FT_PT1



• – ◦ • – ◦ • – ◦ •

Introduction

`FT_PT1` is a first-order low-pass filter (PT1 element) from the OSCAT library. It is used to smooth analog signals and reduce noise with a programmable time constant `TM` and gain factor `K`.

The block is implemented as a `SimpleFB` (stateful function block with `EINIT`/`REQ`/`RST`) and performs discrete-time integration according to the PT1 transfer function:

$$T_M \cdot \frac{dy}{dt} + y(t) = K \cdot x(t) \tag{2.1}$$

Interface Structure

Event Inputs

Name	Type	Description	With Data
`EINIT`	`Event`	Service initialization of the filter	
`REQ`	`Event`	Execution request for a new calculation cycle	`in`, `TM`, `K`
`RST`	`Event`	Invalidates initialization state (`init = FALSE`) for re-seeding on next `REQ`	

Event Outputs

Name	Type	Description	With Data
`INITO`	`Event`	Initialization confirmation	
`CNF`	`Event`	Execution confirmation after calculation	`delta_t`, `out`

Input Vars

Name	Type	Initial Value	Description
`in`	`REAL`	`0.0`	Analog input signal
`TM`	`TIME`	`T#0s`	Filter time constant. If `TM = T#0s`, filtering is bypassed and `out = K * in`.
`K`	`REAL`	`1.0`	Gain factor (proportional coefficient)

Output Vars

Name	Type	Description
`delta_t`	`UDINT`	Elapsed time since last call in microseconds (μs)
`out`	`REAL`	Filtered output value

How It Works

1. **Initialization (`EINIT`)**:
   Resets initialization state, sets `out = 0.0`, and signals readiness via `INITO`.
2. **Cyclic Calculation (`REQ`)**:
   On each `REQ` event, the block calculates elapsed time since the last call (Δt) in microseconds using `T_PLC_US()`.
   - If the block is not yet initialized (`init = FALSE`) or `TM = T#0s`, the filter state is (re)initialized: `init := TRUE`, `out := K * in` is seeded directly using fresh input data, `delta_t := 0`, and the timestamp baseline `last` is refreshed.
   - For `TM > T#0s`, the new output value is calculated according to:

$$\text{out}_{\text{new}} = \text{out}_{\text{old}} + (K \cdot \text{in} - \text{out}_{\text{old}}) \cdot \frac{\Delta t}{T_{\text{eff}}} \tag{2.2}$$

     where $\Delta t = \text{delta_t} \cdot 10^{-6}\,\text{s}$ is the measured call interval in seconds (from `T_PLC_US()`), $T_M = \text{TIME_TO_REAL}\ (TM)$ is the filter time constant in seconds, and $T_{\text{eff}} = \max(T_M, \Delta t)$ is the effective time constant in seconds. When the call interval Δt exceeds the configured filter time T_M, T_{eff} is clamped to Δt, capping the weighting factor $\frac{\Delta t}{T_{\text{eff}}}$ to at most 1.0.
   - To prevent denormalized float underruns, values $|out| < 1.0 \times 10^{-20}$ are automatically zeroed.
3. **Filter Reset (`RST`)**:
   Invalidates the initialization state (sets `init := FALSE`) without modifying `out` or timestamps immediately. This defers seeding until the next `REQ` event, ensuring fresh input data is sampled (rather than stale or default `0.0` values) and refreshing the timebase baseline.

Technical Features

- **Cycle-Independent Timebase**: Time differences are measured in microseconds via `T_PLC_US()`, compensating for call cycle variations.
- **Clamped Effective Filter Time Constant ($T_{\text{eff}} = \max(T_M, \Delta t)$)**: If call interval Δt exceeds filter time T_M, the block caps the discretization factor $\frac{\Delta t}{T_{\text{eff}}}$ to at most 1.0. This prevents Euler instability and overshoot when call cycles occur slower than T_M.
- **Filter Bypass at `TM = 0`**: When `TM = T#0s`, damping is disabled and the input signal is passed through scaled by `K`.
- **Clean Reset Recovery**: `RST` invalidates the initialization state (`init := FALSE`) without immediate output overwrite. The subsequent `REQ` event samples fresh input data, seeds `out := K * in`, and resets the timing baseline `last`, preventing spurious step jumps from stale input values.
- **Project-Specific Time Conversion**: Uses the project-specific helper function `TIME_TO_REAL.fct` for clean conversion of `TIME` input `TM` to seconds (`REAL`).

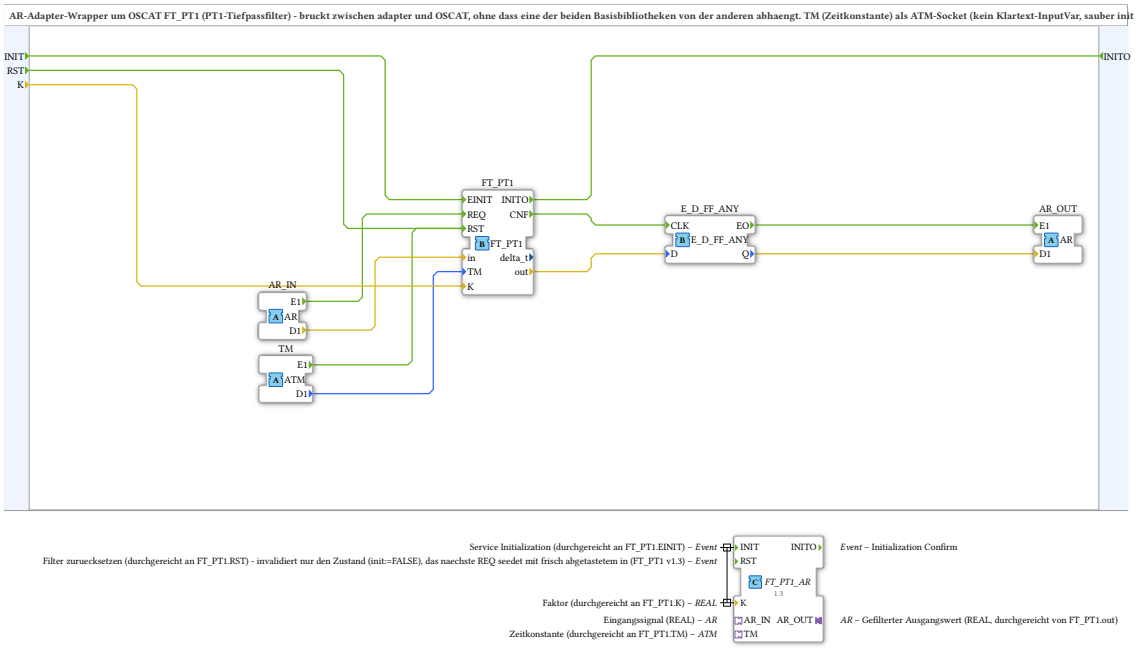
Application Scenarios

- Smoothing fluctuating sensor readings (e.g., pressure, temperature, speed).
- Noise reduction for control loop inputs.
- Soft start/ramp adjustment for setpoints.

See Also

- `FT_PT1_AR` – AR adapter wrapper for `FT_PT1`.
- `FT_PT2` – Second-order low-pass filter.
- `TIME_TO_REAL` – Time conversion helper function.

FT_PT1_AR



Introduction

`FT_PT1_AR` is an AR adapter wrapper around the OSCAT low-pass filter block `FT_PT1`. It encapsulates first-order low-pass filtering behind a purely adapter-based interface for IEC 61499 applications.

The time constant `TM` is provided as an `ATM` socket (per section 13 of the `iec61499-creator` skill), while gain factor `K` remains a plain `InputVar`. Internally, the block uses an `E_D_FF_ANY` D flip-flop to detect changes in the filtered value and fire adapter events only upon initial evaluation or when an actual change occurs.

Interface Structure

Event Inputs

Name	Type	Description
`INIT`	`EInit`	Service initialization, passed to `FT_PT1.INIT`
`RST`	`Event`	Invalidates filter state via `FT_PT1.RST` (`init := FALSE`)

Event Outputs

Name	Type	Description
`INITO`	`EInit`	Initialization confirmation from `FT_PT1`

Input Vars

Name	Type	Initial Value	Description
`K`	`REAL`	`1.0`	Gain factor, passed to `FT_PT1.K`

Output Vars

None. Output is provided exclusively through the adapter plug `AR_OUT`.

Adapters

Interface	Direction	Adapter Type	Data Type	Description

AR_IN	Socket	AR	REAL	Input signal to be filtered
TM	Socket	ATM	TIME	Filter time constant (TM.D1 passed to FT_PT1.TM)
AR_OUT	Plug	AR	REAL	Filtered output value (FT_PT1.out via E_D_FF_ANY)

How It Works

The block connects FT_PT1 (OSCAT) and E_D_FF_ANY in an internal network:

1. **Signal Processing**:
   An event on AR_IN.E1 triggers FT_PT1.REQ. AR_IN.D1 supplies the raw analog value.
2. **Time Constant & Re-initialization**:
   The filter time TM.D1 is supplied by the ATM socket and passed to FT_PT1.TM. An event on TM.E1 also directly triggers FT_PT1.RST internally, invalidating the filter state (init := FALSE) so that the subsequent AR_IN.E1 request reinitializes the filter (out := K * in) using fresh input data and the updated time constant.
3. **Change Detection & Decoupling (E_D_FF_ANY)**:
   After calculation, FT_PT1.CNF triggers CLK on internal E_D_FF_ANY. The flip-flop emits the initial output value on the first clock after start. On subsequent cycles, AR_OUT.E1 and AR_OUT.D1 are updated **only** when a value change occurs.
4. **Reset & Initialization**:
   - INIT controls FT_PT1.INIT and confirms via INITO. Unconnected INIT events auto-fire once upon deployment.
   - RST as well as any incoming TM.E1 event are passed to FT_PT1.RST to invalidate the filter state (init := FALSE). The next AR_IN.E1 request performs clean re-initialization.

Technical Features

- **Clean Adapter Boundary**: Prevents direct accessing of internal .E1/.D1 structures across SubApp networks.
- **Automatic Filter Re-initialization on TM Change**: Receiving a TM.E1 event internally triggers FT_PT1.RST (setting init := FALSE). The subsequent AR_IN.E1 request reinitializes out := K * in with fresh input data and resets the timing baseline, so modified time constants take effect cleanly without jump artifacts or stale seeding.
- **Event Traffic Reduction**: E_D_FF_ANY prevents unnecessary event flooding down the processing chain for unchanged values.
- **ATM Socket Convention**: Timing parameters are routed via ATM sockets rather than bare variables (fed at instantiation site e.g., via initval_ATM).

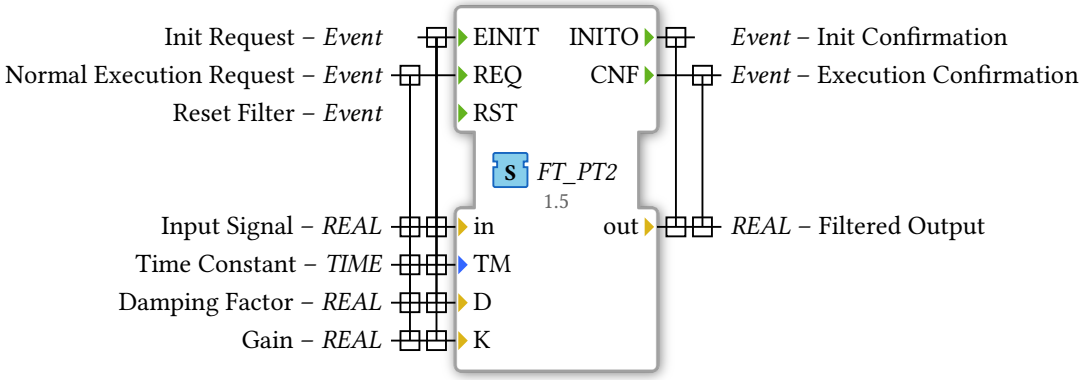
Application Scenarios

- Smoothing analog sensor readings (e.g., pressure, temperature, speed) in adapter-based SubApp architectures.
- Noise filtering prior to thresholding or hysteresis blocks (AR_D_FF_HYS_TMIN).

See Also

- FT_PT1 – Underlying OSCAT block.
- FT_PT2_AR – AR adapter wrapper for 2nd-order PT2 filter.
- FT_DERIV_AR – AR adapter wrapper for differentiator.

FT_PT2



2nd grade filter with programmable times

Interface

Event inputs

Name	Comment	With
EINIT	Init Request	in, TM, D, K
REQ	Normal Execution Request	in, TM, D, K
RST	Reset filters	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	out
CNF	Execution Confirmation	out

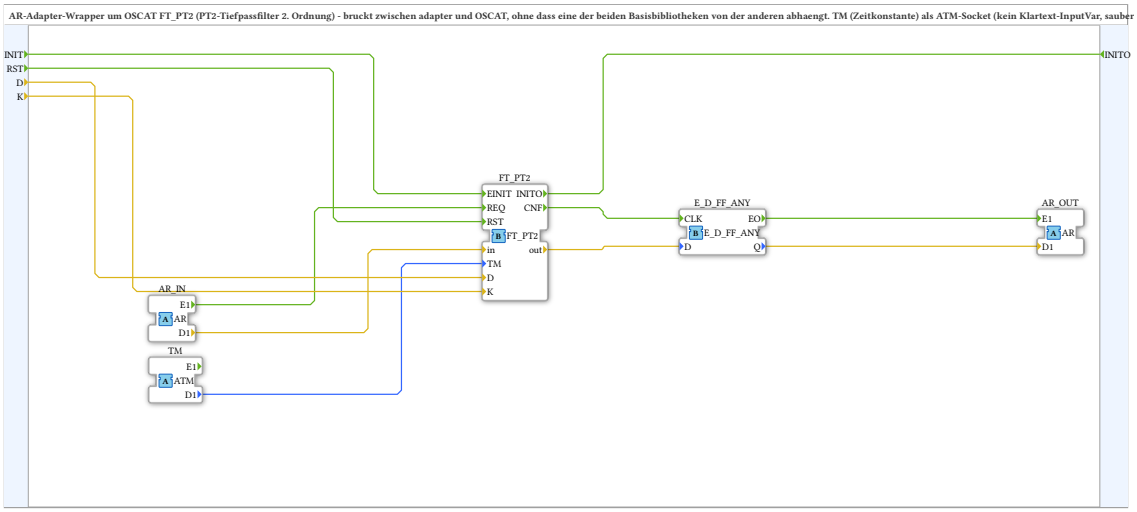
Input Vars

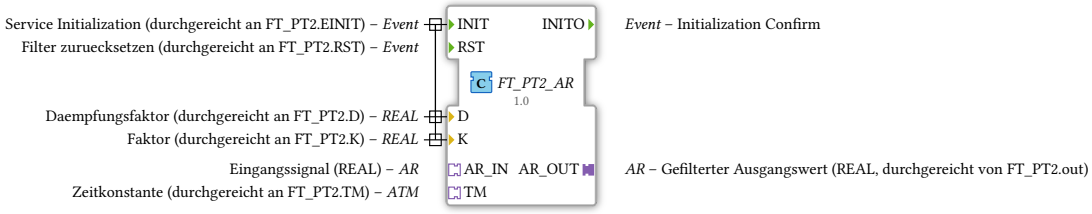
Name	Type	Initial Value	Comment
in	REAL		Input signal
TM	TIME		Time Constant
D	REAL		Damping Factor
K	REAL	1.0	Gain

Output Vars

Name	Type	Comment
out	REAL	Filtered Output

FT_PT2_AR





Introduction

`FT_PT2_AR` is an AR adapter wrapper around the OSCAT 2nd-order low-pass filter block `FT_PT2`. It enables 2nd-order low-pass filtering with configurable time constant `TM`, damping `D`, and gain `K` within purely adapter-based IEC 61499 applications.

The time constant `TM` is provided via an `ATM` adapter socket (per section 13 of the `iec61499-creator` skill), while damping `D` and gain `K` remain plain `InputVars`. An internal `E_D_FF_ANY` D flip-flop ensures that output event `AR_OUT.E1` is fired unconditionally on the first cycle and subsequently only when an actual value change occurs.

Interface Structure

Event Inputs

Name	Type	Description
`INIT`	`EInit`	Service initialization, passed to `FT_PT2.INIT`
`RST`	`Event`	Resets the filter output via `FT_PT2.RST`

Event Outputs

Name	Type	Description
`INITO`	`EInit`	Initialization confirmation from `FT_PT2`

Input Vars

Name	Type	Initial Value	Description
`D`	`REAL`	`0.0`	Filter damping factor (e.g., 0.707 for Butterworth response)
`K`	`REAL`	`1.0`	Gain factor

Output Vars

None. Output is provided exclusively through the adapter plug `AR_OUT`.

Adapters

Interface	Direction	Adapter Type	Data Type	Description
`AR_IN`	Socket	`AR`	`REAL`	Input signal to be filtered
`TM`	Socket	`ATM`	`TIME`	Filter time constant (`TM.D1` passed to `FT_PT2.TM`)
`AR_OUT`	Plug	`AR`	`REAL`	Filtered output value (`FT_PT2.out` via `E_D_FF_ANY`)

How It Works

The block embeds `FT_PT2` inside an internal FB network:

1. **Signal Processing**:
   An event on `AR_IN.E1` triggers `FT_PT2.REQ`. `AR_IN.D1` is supplied to `FT_PT2.in`.
2. **Parameterization**:
   `TM.D1` is read from the `ATM` socket. `D` and `K` are supplied as parameters.
3. **2nd-Order Stateful Integration**:
   `FT_PT2` filters the signal internally through two cascaded integration stages (`INTEGRATE`).
4. **Change Filtering (`E_D_FF_ANY`)**:
   `FT_PT2.CNF` drives internal `E_D_FF_ANY`. The flip-flop forwards the computed output value to `AR_OUT` unconditionally on the first cycle. On subsequent cycles, `AR_OUT.E1` and `AR_OUT.D1` are updated only when the computed filter value differs from the previous value.
5. **Reset & Initialization**:
   `INIT` controls `FT_PT2.INIT`. `RST` clears the internal memory of both integrator stages.

Technical Features

- **2nd-Order Response**: Provides steeper attenuation of high-frequency noise (12 dB/octave roll-off) compared to a PT1 filter.
- **Integrated Event Decoupling**: Eliminates redundant adapter events in downstream SubApps.
- **`ATM` Socket for Timing**: Adheres to project design rules for timing constants.

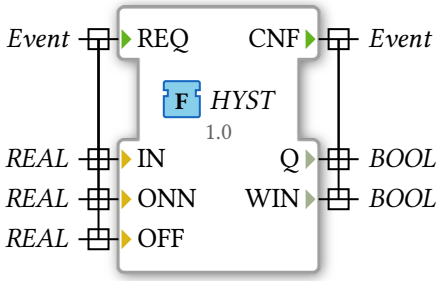
Application Scenarios

- High-frequency noise suppression on analog signals (e.g., force, pressure, or ground speed).
- Vibration damping in closed-loop control systems.

See Also

- `FT_PT2` – Underlying OSCAT block.
- `FT_PT1_AR` – AR adapter wrapper for 1st-order PT1 filter.
- `FT_DERIV_AR` – AR adapter wrapper for differentiator.

HYST



🎧 Podcast

- The relay in detail: Switching amplifiers, protection, and the secrets of A1/A2, 85/86, and the hysteresis

Interface

Event Inputs

Name	Comment	With
REQ		IN, ONN, OFF

Event Outputs

Name	Comment	With
CNF		Q, WIN

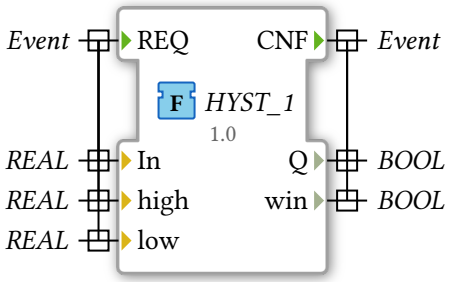
Input Vars

Name	Type	Comment
IN	REAL	
ONN	REAL	
OFF	REAL	

Output Vars

Name	Type	Comment
Q	BOOL	
WIN	BOOL	

HYST_1



Hysteresis function This hysteresis function switches the output high if the input signal reaches obove high and will switch to low when the input falls back below low value. a separate output mid is set if the input stays between low and high value.

Interface

Event inputs

Name	Comment	With
REQ		In, high, low

Event Outputs

Name	Comment	With
CNF		Q, win

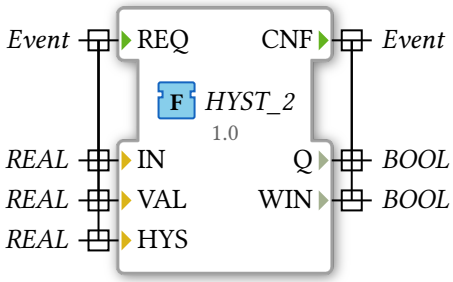
Input Vars

Name	Type	Comment
In	REAL	
high	REAL	
low	REAL	

Output Vars

Name	Type	Comment
Q	BOOL	
win	BOOL	

HYST_2



Hysteresis function This hysteresis function switches the output high if the input signal reaches obove val + hys/2 and will switch to low when the input falls back below val - hys/2 value. a separate output mid is set if the input stays between low and high value.

Interface

Event inputs

Name	Comment	With
REQ		IN, VAL, HYS

Event Outputs

Name	Comment	With
CNF		Q, WIN

Input Vars

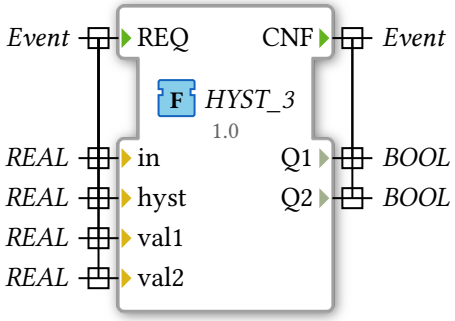
Name	Type	Comment
IN	REAL	
VAL	REAL	
HYS	REAL	

Output Vars

Name	Type	Comment
Q	BOOL	

WIN	BOOL	

HYST_3



Hysteresis function this is a double hysteresis function. Out1 follows a hysteresis function defined by val1and hyst, while out 2 follows val2 and hyst. if the input signal is between the two hysteresis switches (val1 and val2) then non of the outputs is active.

Interface

Event inputs

Name	Comment	With
REQ		in, hyst, val1, val2

Event Outputs

Name	Comment	With
CNF		Q1, Q2

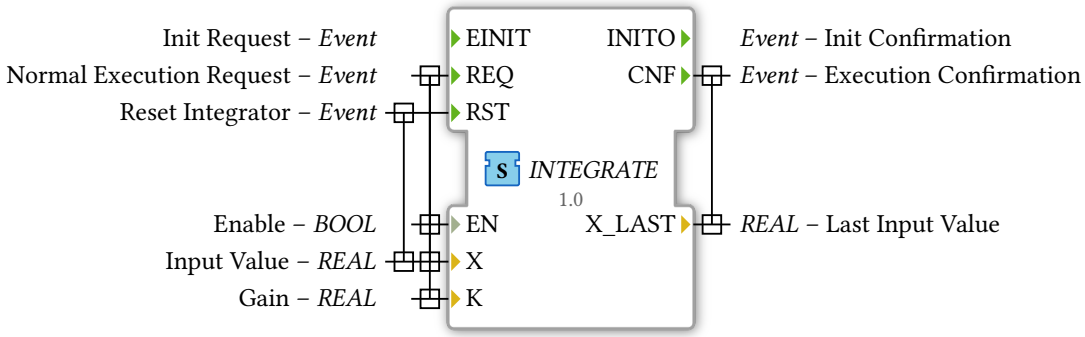
Input Vars

Name	Type	Comment
in	REAL	
hyst	REAL	
val1	REAL	
val2	REAL	

Output Vars

Name	Type	Comment
Q1	BOOL	
Q2	BOOL	

INTEGRATE



Integrator with I/O for out

Interface

Event inputs

Name	Comment	With
EINIT	Init Request	
REQ	Normal Execution Request	EN, X, K, Y
RST	Reset Integrator	X

Event Outputs

Name	Comment	With
INITO	Init Confirmation	
CNF	Execution Confirmation	X_LAST, Y

Input Vars

Name	Type	Initial Value	Comment
EN	BOOL	TRUE	Enable
X	REAL		Input Value
K	REAL	1.0	Gain

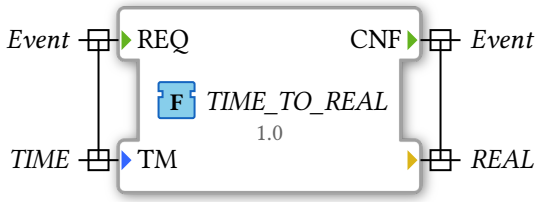
Output Vars

Name	Type	Comment
X_LAST	REAL	Last Input Value

InOut Vars

Name	Type	Comment
Y	REAL	Output Value

TIME_TO_REAL



Introduction

`TIME_TO_REAL` is a project-specific helper function (`FunctionType`) that converts a time duration of type `TIME` into a floating-point value (`REAL`). The return value represents the duration in **seconds**.

This function is utilized within OSCAT control and filter blocks (such as `FT_PT1`) to format PLC time quantities for mathematical differential and integration formulas.

Interface Structure

Inputs

Name	Type	Description
`IN`	`TIME`	Input time duration

Outputs

Name	Type	Description
(Return)	`REAL`	Calculated time duration in seconds

How It Works

The function converts the internal millisecond representation of the `TIME` input into seconds:

$$\text{Return} = \text{TIME_TO_UDINT}\ (\text{IN}) \cdot 1.0 \times 10^{-3} \tag{2.3}$$

Examples:

- `IN = T#1s` → `Return = 1.0`
- `IN = T#500ms` → `Return = 0.5`
- `IN = T#2m30s` → `Return = 150.0`

Technical Features

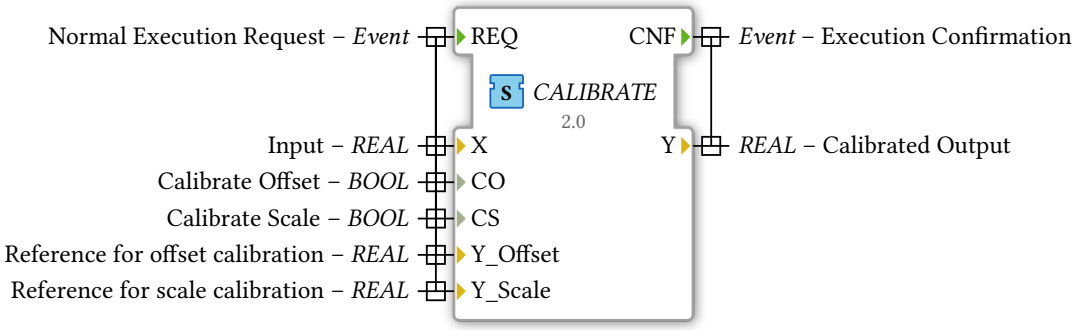
- **Project-Specific Extension**: `TIME_TO_REAL` is not a standard OSCAT library function, but a local IEC 61499 conversion helper.
- **Scaling Precision**: Uses explicit floating-point scaling (1.0×10^{-3}) to prevent integer truncation errors during division.

See Also

- `FT_PT1` – First-order low-pass filter (uses `TIME_TO_REAL`).
- `FT_PT2` – Second-order low-pass filter.

measurements

CALIBRATE



• – ◦ • – ◦ • – ◦ •

Introduction

`CALIBRATE` performs a two-point calibration (offset & scale) of an analog input signal: `Y = (X + OFFSET) * SCALE`. Calibration is not triggered by dedicated events but by the Boolean inputs `CO`/`CS`, which are checked on every `REQ`. For an event-driven variant see E_CALIBRATE. For details on the calibration procedure, see Two- and Three-Point Calibration Procedures.

Interface Structure

Event Inputs

- **REQ**: normal execution request, carries `X`, `CO`, `CS`, `Y_Offset`, `Y_Scale`, `OFFSET`, `SCALE`. Recomputes `Y` on every call and, where applicable, performs a calibration step (see Functionality).

Event Outputs

- **CNF**: confirms execution, carries `Y`, `OFFSET` and `SCALE`.

Data Inputs

- **X** (REAL): raw input value.
- **CO** (BOOL): when `TRUE`, triggers offset calibration.
- **CS** (BOOL): when `TRUE`, triggers scale calibration (only meaningful once `CO` has run first).
- **Y_Offset** (REAL): target value for `Y` at the low reference point (offset calibration).
- **Y_Scale** (REAL): target value for `Y` at the high reference point (scale calibration).

Data Outputs

- **Y** (REAL): calibrated output value.

In/Out Variables

- **OFFSET** (REAL, initial `0.0`): stored offset value – recalculated on `CO`, otherwise held persistently by the caller.
- **SCALE** (REAL, initial `1.0`): stored scale factor – recalculated on `CS`, otherwise held persistently by the caller.

Functionality

On every `REQ`, `CALIBRATE` first checks `CO` and `CS` (as an `ELSIF` chain, so at most one calibration step per call):

- If `CO = TRUE`: `OFFSET := Y_Offset - X` – shifts `X` so that `Y = Y_Offset` at the current reference (only correct when `SCALE = 1`).
- Else if `CS = TRUE`: `SCALE := Y_Scale / (X + OFFSET)` – scales so that `Y = Y_Scale` at the current reference. Requires `OFFSET` to already have been determined via `CO`.

`Y := (X + OFFSET) * SCALE` is then computed either way and output via `CNF`.

Two-point calibration procedure:

1. Apply the low reference, set `Y_Offset` to the desired output, set `CO = TRUE` – yields `OFFSET`.
2. Apply the high reference, set `Y_Scale` to the desired output, set `CS = TRUE` – yields `SCALE`.

Example (4-20 mA pressure sensor via logiBUS, normalized to `0.0..1.0`, desired output range `0.0..500.0`):

Step	Action	Result
1	Apply 4 mA (`X=0.0`), `Y_Offset=0.0`, `CO=TRUE`	`OFFSET = 0`
2	Apply 20 mA (`X=1.0`), `Y_Scale=500.0`, `CS=TRUE`	`SCALE = 500`

Result: `Y = (X + 0) * 500 = X * 500 = 0..500`.

Technical Details

- **Order not enforced**: `CO` must run before `CS`, since `CS` uses the `OFFSET` already determined. As a `SimpleFB` with a single `ECState`, ensuring the correct order is the caller's responsibility.
- **Y after CO only correct when SCALE = 1**: the formula `OFFSET := Y_Offset - X` only yields exactly `Y = Y_Offset` after offset calibration if `SCALE` still has its initial value `1.0`.
- **Triggered via BOOL, not events**: unlike the `E_CALIBRATE*` variants, calibration here runs through the data inputs `CO`/`CS`, re-evaluated on every `REQ` – no dedicated calibration event needed, but also no separate confirmation event per calibration step.

State Overview

`CALIBRATE` is a `SimpleFB` with a single `ECState` (`REQ`): every call immediately runs the `REQ` algorithm and fires `CNF`. There are no state transitions and no real ECC – the "calibration phase" results purely from the value of `CO`/`CS` at call time.

Application Scenarios

- Scaling analog sensors (pressure, temperature, level) with two-point reference calibration
- Systems where the calibration trigger is already available as a Boolean signal (e.g. a button/switch rather than a dedicated event source)
- Simple applications where an accidental wrong order (`CS` before `CO`) poses no risk

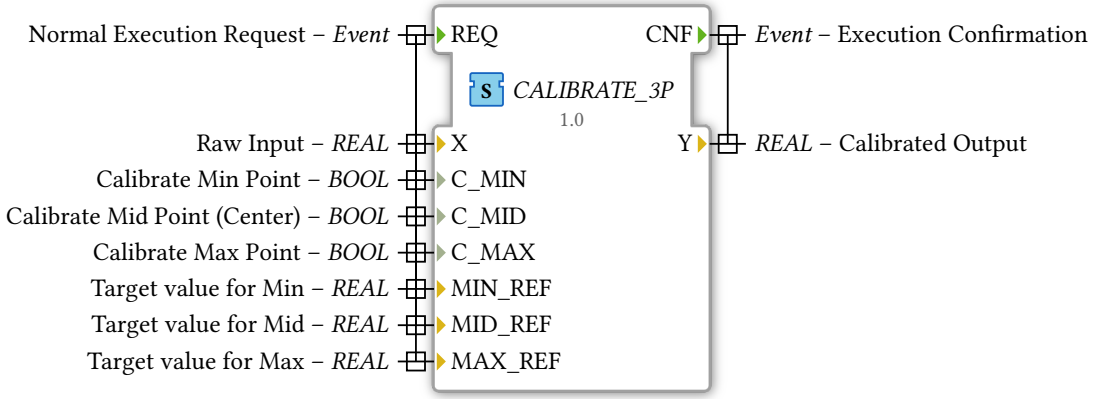
⚖️ Comparison with Similar Blocks

Compare with E_CALIBRATE, which performs the same two-point calibration but event-driven (`EICO`/`EICS` instead of `CO`/`CS`). For background on the calibration theory, see Two- and Three-Point Calibration Procedures. For a three-point calibration (e.g. joysticks with a center position) see CALIBRATE_3P.

Conclusion

`CALIBRATE` provides simple, Boolean-triggered two-point calibration for analog signals, and fits applications where the caller ensures the correct order of offset and scale calibration.

CALIBRATE_3P



• – ◦ • – ◦ • – ◦ •

Introduction

`CALIBRATE_3P` is a three-point calibration for raw inputs with a center position, as typically found on joysticks. Instead of a single linear scale like CALIBRATE, it interpolates piecewise linearly between three

stored reference points (Min, Mid, Max), which also compensates for a center position that doesn't sit exactly in the middle of the raw range.

Interface Structure

Event Inputs

- **REQ**: normal execution request, carries `X`, `C_MIN`, `C_MID`, `C_MAX`, `MIN_REF`, `MID_REF`, `MAX_REF`, `X_MIN`, `X_MID`, `X_MAX`. Recomputes `Y` on every call and, where applicable, performs a calibration step.

Event Outputs

- **CNF**: confirms execution, carries `Y`, `X_MIN`, `X_MID`, `X_MAX`.

Data Inputs

- **X** (REAL): raw input value.
- **C_MIN** (BOOL): when `TRUE`, triggers calibration of the minimum point.
- **C_MID** (BOOL): when `TRUE`, triggers calibration of the mid point.
- **C_MAX** (BOOL): when `TRUE`, triggers calibration of the maximum point.
- **MIN_REF** (REAL, initial `0.0`): target output value for the minimum point.
- **MID_REF** (REAL, initial `50.0`): target output value for the mid point.
- **MAX_REF** (REAL, initial `100.0`): target output value for the maximum point.

Data Outputs

- **Y** (REAL): calibrated, interpolated output value, clipped to `MIN_REF..MAX_REF`.

In/Out Variables

- **X_MIN** (REAL, initial `0.0`): stored raw value at the minimum point.
- **X_MID** (REAL, initial `50.0`): stored raw value at the mid point.
- **X_MAX** (REAL, initial `100.0`): stored raw value at the maximum point.

Functionality

On every `REQ`, `CALIBRATE_3P` first checks `C_MIN`, `C_MID` and `C_MAX` (as an `ELSIF` chain, at most one calibration point per call) and, if the corresponding flag is set, stores the current raw value `X` in `X_MIN`/`X_MID`/`X_MAX`.

`Y` is then interpolated piecewise linearly:

- For `X < X_MID`: interpolation between `(X_MIN, MIN_REF)` and `(X_MID, MID_REF)`.
- For `X >= X_MID`: interpolation between `(X_MID, MID_REF)` and `(X_MAX, MAX_REF)`.
- The result is clipped to the range `MIN_REF..MAX_REF`.

The three calibration points can be calibrated in **any order** and repeated individually as often as needed – but at most one is updated per `REQ` call.

Example (joystick with center drift, raw range `0..1000`, desired output range `-100..0..100`):

Step	Action	Result
1	Move joystick to **minimum**, `C_MIN=TRUE`	`X_MIN = 50` (drift from ideal `0`)
2	Move joystick to **center**, `C_MID=TRUE`	`X_MID = 520` (drift from ideal `500`)
3	Move joystick to **maximum**, `C_MAX=TRUE`	`X_MAX = 980` (drift from ideal `1000`)

Result: `Y` is interpolated and clipped between `MIN_REF=-100`, `MID_REF=0` and `MAX_REF=100` – the joystick's actual (drifted) center position now yields exactly `Y=0`.

Technical Details

- **Center-position compensation**: unlike a plain two-point calibration (see CALIBRATE), the center position is captured explicitly – important for joysticks whose mechanical rest position doesn't sit exactly in the middle of the electrical raw range.
- **Output always clipped**: `Y` is clipped to `MIN_REF..MAX_REF` after interpolation, even if `X` lies outside the calibrated raw values.
- **Degenerate calibration points**: if `X_MID` is not above `X_MIN` (or `X_MAX` not above `X_MID`), the affected interpolation returns a constant `MIN_REF` or `MID_REF` instead of dividing by zero.

- **Order not enforced**: as with `CALIBRATE`, being a `SimpleFB` means it's the caller's responsibility to supply sensible calibration values. For an event-driven variant see E_CALIBRATE_3P.

State Overview

`CALIBRATE_3P` is a `SimpleFB` with a single `ECState` (`REQ`): every call immediately runs the `REQ` algorithm and fires `CNF`. There are no state transitions – which calibration point (if any) gets updated is determined purely by `C_MIN`/`C_MID`/`C_MAX` at call time.

Application Scenarios

- Calibrating joysticks and similar controls with a mechanical center position
- Raw inputs with nonlinear or asymmetric behavior around the zero point
- Applications where individual calibration points need to be readjusted independently without affecting the others

⚖️ Comparison with Similar Blocks

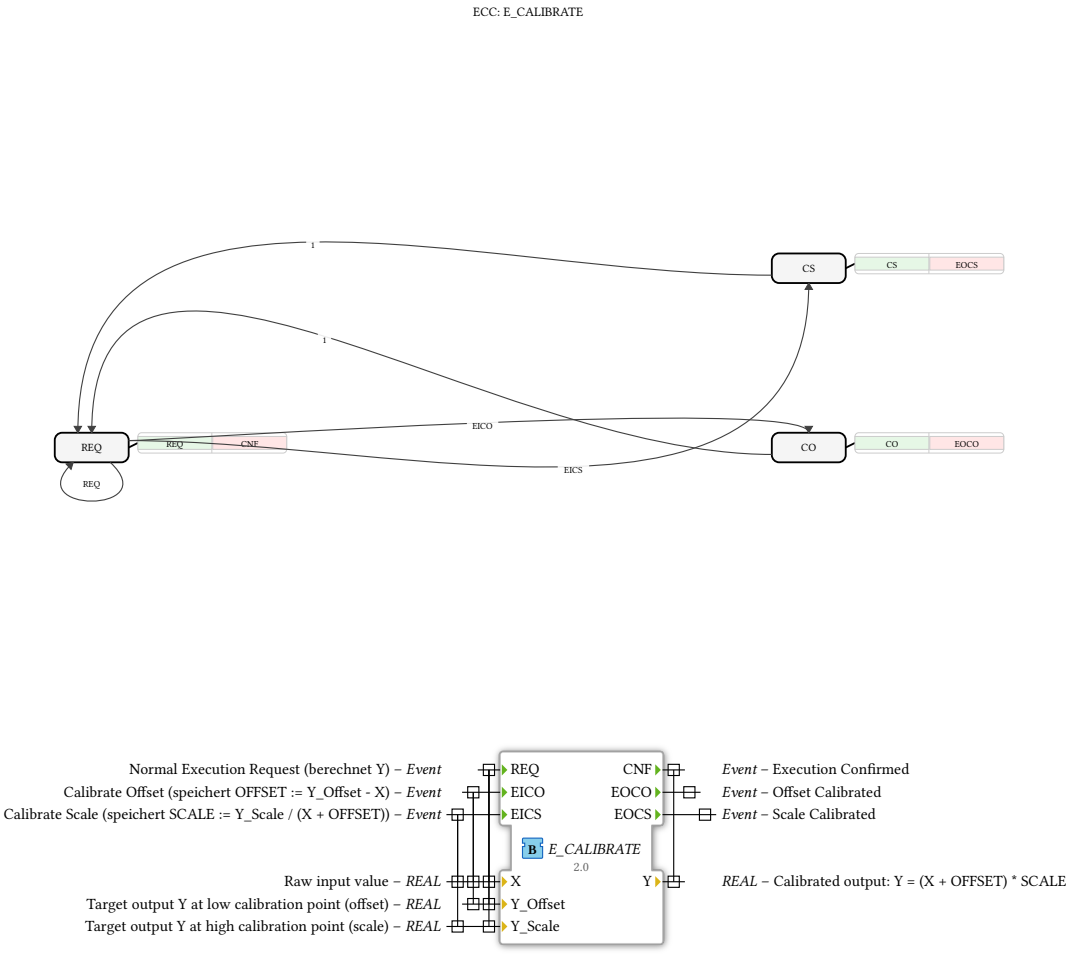
For details on the three-point calibration procedure, see Two- and Three-Point Calibration Procedures.

Compare with CALIBRATE, which uses the same Boolean-triggered approach but only offers a linear two-point calibration without a center position, and with E_CALIBRATE_3P, which provides the same three-point logic event-driven (`EI_MIN`/`EI_MID`/`EI_MAX`) with its own confirmation event per point.

Conclusion

`CALIBRATE_3P` fits anywhere a simple two-point calibration falls short because the raw value range isn't symmetric around a (possibly shifted) center point – most notably for joysticks and similar controls.

E_CALIBRATE



• – ○ • – ○ • – ○ •

Introduction

E_CALIBRATE is the event-driven counterpart to CALIBRATE: the two-point calibration (Y = (X + OFFSET) * SCALE) is triggered not by Boolean data values but by its own events EICO/EICS, each with its own confirmation event. For details on the calibration procedure, see Two- and Three-Point Calibration Procedures.

Interface Structure

Event Inputs

- **REQ**: normal execution request – computes Y. Carries X, Y_Offset, Y_Scale, OFFSET, SCALE.
- **EICO**: calibrates the offset (OFFSET := Y_Offset - X). Carries X, Y_Offset.
- **EICS**: calibrates the scale (SCALE := Y_Scale / (X + OFFSET)). Carries X, Y_Scale, OFFSET.

Event Outputs

- **CNF**: confirms REQ, carries Y.
- **EOCO**: confirms offset calibration, carries OFFSET.
- **EOCS**: confirms scale calibration, carries SCALE.

Data Inputs

- **X** (REAL): raw input value.
- **Y_Offset** (REAL): target output value at the low reference point.
- **Y_Scale** (REAL): target output value at the high reference point.

Data Outputs

- **Y** (REAL): calibrated output value.

In/Out Variables

- **OFFSET** (REAL, initial 0.0): stored offset value.
- **SCALE** (REAL, initial 1.0): stored scale factor.

Functionality

E_CALIBRATE has three ECC states, all directly reachable from state REQ:

- **REQ** (start state): on the REQ event, Y := (X + OFFSET) * SCALE is computed and CNF fires; the block stays in REQ.
- **CO**: on EICO, the block transitions to CO, runs OFFSET := Y_Offset - X, fires EOCO and immediately returns to REQ.
- **CS**: on EICS, the block transitions to CS, runs SCALE := Y_Scale / (X + OFFSET) (only if X + OFFSET <> 0), fires EOCS and immediately returns to REQ.

Since both EICO and EICS are reachable from REQ at any time, the ECC does not enforce ordering – EICS can also be fired before EICO, but then yields an inconsistent result since SCALE would be based on an as-yet-uncalibrated OFFSET.

Example (4-20 mA pressure sensor via logiBUS, normalized to 0.0..1.0, desired output range 0.0..500.0):

Step	Action	Result
1	Apply 4 mA (X=0.0), Y_Offset=0.0, fire EICO	OFFSET = 0
2	Apply 20 mA (X=1.0), Y_Scale=500.0, fire EICS	SCALE = 500

Result: Y = (X + 0) * 500 = X * 500 = 0..500.

Technical Details

- **Y after CO only correct when SCALE = 1**: as with CALIBRATE, OFFSET := Y_Offset - X only yields exactly Y = Y_Offset after offset calibration if SCALE is still 1.0.
- **No order enforcement**: EICO and EICS are both directly reachable from REQ – the ECC does not track whether calibration has already happened.
- **REQ stays available at all times**: normal operation (REQ) is possible regardless of calibration status, even between EICO and EICS.

State Overview

```
REQ --EICO--> CO --1--> REQ     (offset calibration)
REQ --EICS--> CS --1--> REQ     (scale calibration)
REQ --REQ---> REQ --1--> REQ    (normal operation)
```

All three states are pass-through states: the block returns to `REQ` immediately after each action.

Application Scenarios

- Two-point calibration of analog sensors in systems that already provide the calibration trigger as an event (e.g. a button-press event rather than a data value)
- Applications where offset and scale calibration each need their own, immediate confirmation (`EOCO`/`EOCS`)
- Systems with occasional offset re-calibration without redetermining the scale

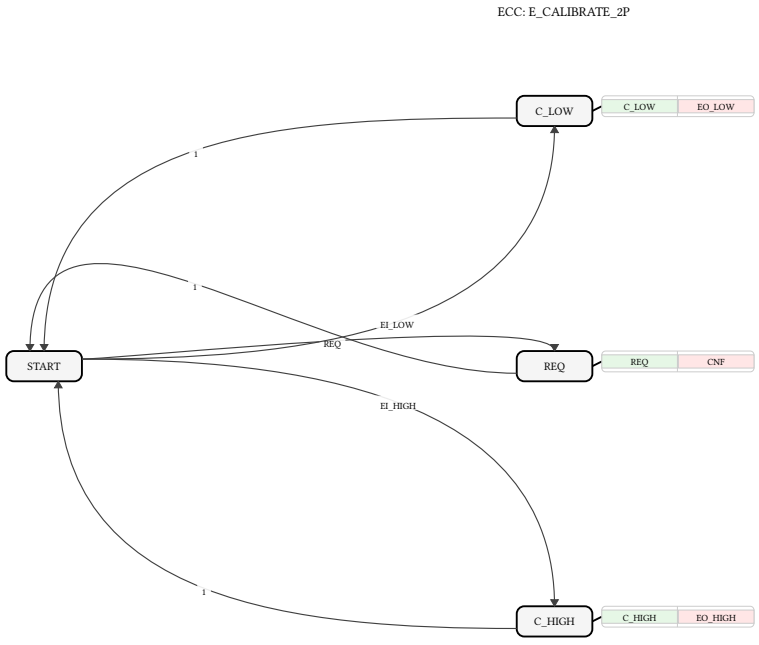
⚖️ Comparison with Similar Blocks

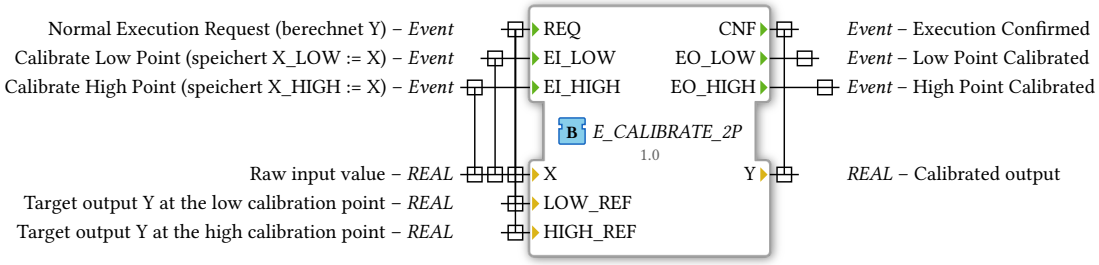
Compare with CALIBRATE, which applies the same formula Boolean-triggered rather than event-driven. For background on the calibration theory, see Two- and Three-Point Calibration Procedures.

Conclusion

`E_CALIBRATE` provides event-driven two-point calibration with its own confirmation event per calibration step, but still leaves the correct order (offset before scale) to the caller.

E_CALIBRATE_2P





• – ○ • – ○ • – ○ •

Introduction

E_CALIBRATE_2P is an event-driven two-point calibration (low/high) with independently stored points. It replaces E_CALIBRATE (E_CALIBRATE_SQ behaviour): the two points are stored completely independently of each other and only combined into a line on reading (REQ). There is therefore no enforced order of EI_LOW/EI_HIGH.

Interface Structure

Event Inputs

- **REQ**: normal execution request, calculates Y. Provides X, LOW_REF, HIGH_REF, X_LOW, X_HIGH.
- **EI_LOW**: calibrates the low point (X_LOW := X). Provides X.
- **EI_HIGH**: calibrates the high point (X_HIGH := X). Provides X.

Event Outputs

- **CNF**: confirms REQ, provides Y.
- **EO_LOW**: low point calibrated, provides X_LOW.
- **EO_HIGH**: high point calibrated, provides X_HIGH.

Data Inputs

- **X** (REAL): raw input value.
- **LOW_REF** (REAL): target output Y at the low calibration point.
- **HIGH_REF** (REAL): target output Y at the high calibration point.

Data Outputs

- **Y** (REAL): calibrated output value.

In/Out Variables (InOut)

- **X_LOW** (REAL, initial value 0.0): stored raw value at the low calibration point.
- **X_HIGH** (REAL, initial value 1.0): stored raw value at the high calibration point.

Adapters

No adapters available.

Functionality

Formula: Y = LOW_REF + (X - X_LOW) * (HIGH_REF - LOW_REF) / (X_HIGH - X_LOW)
The block has a start state START and three action states directly reachable from it:

- **REQ**: calculates Y live by linear interpolation between the two stored points (without clipping, as with E_CALIBRATE_SQ) and fires CNF. If X_HIGH - X_LOW = 0, then Y := LOW_REF.
- **C_LOW**: stores the current raw value in X_LOW and fires EO_LOW.
- **C_HIGH**: stores the current raw value in X_HIGH and fires EO_HIGH.

Every action state returns immediately to START.
Calibration (free order):

1. **Low (EI_LOW)**: apply the low reference, set LOW_REF to the desired output, fire EI_LOW → X_LOW := X
2. **High (EI_HIGH)**: apply the high reference, set HIGH_REF to the desired output, fire EI_HIGH → X_HIGH := X

Either point can be recalibrated on its own at any later time without repeating the other.

Technical Details

- **Independent points instead of incremental OFFSET/SCALE calculation:** E_CALIBRATE_SQ computes OFFSET/SCALE step by step (EICO sets OFFSET from a live SCALE, EICS computes both from internally remembered values) and is only mathematically valid after at least one EICO. That required an ECC-enforced order to avoid silently wrong offsets. E_CALIBRATE_2P stores X_LOW/X_HIGH like E_CALIBRATE_3P stores its points, with no data dependency between EI_LOW and EI_HIGH.
- No clipping of Y to LOW_REF..HIGH_REF (unlike E_CALIBRATE_3P).
- Degenerate calibration points (X_HIGH = X_LOW) yield LOW_REF instead of a division by zero.

State Overview

```
START --REQ-----> REQ    --1--> START   (normal operation)
START --EI_LOW--> C_LOW  --1--> START   (calibrate low point)
START --EI_HIGH-> C_HIGH --1--> START   (calibrate high point)
```

Application Scenarios

- Two-point calibration of analog inputs (e.g. sensors) when the calibration trigger is already available as an event
- Readjusting only one point

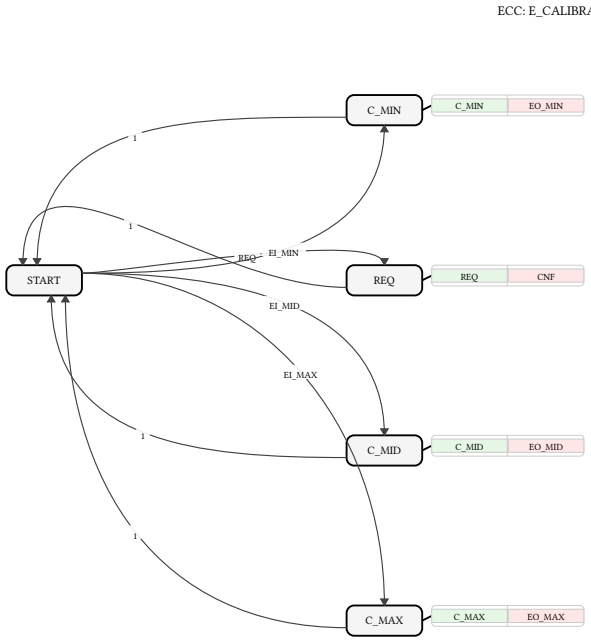
⚖️ Comparison with Similar Building Blocks

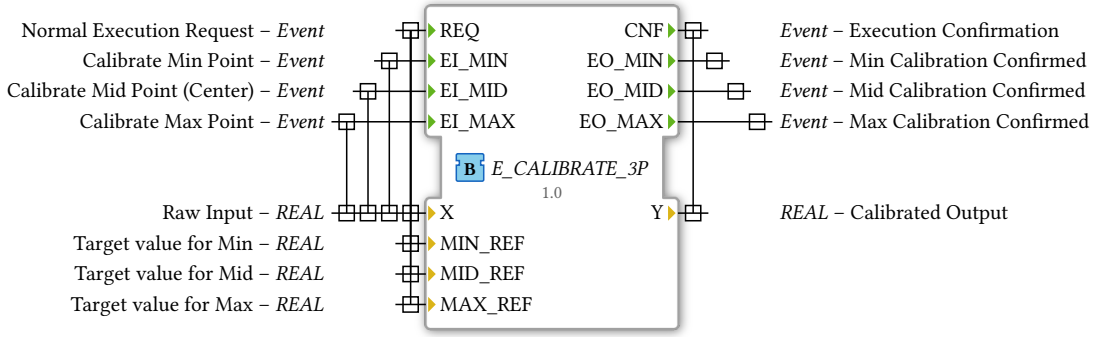
- E_CALIBRATE: variant with `EICO`/`EICS` (`OFFSET`/`SCALE`) in which `EICS` only yields a consistent result after `EICO`
- E_CALIBRATE_3P: three-point variant with piecewise interpolation and clipping

Conclusion

`E_CALIBRATE_2P` provides an event-driven two-point calibration with independently triggerable calibration points and no enforced order.

E_CALIBRATE_3P





• – ○ • – ○ • – ○ •

Introduction

E_CALIBRATE_3P is the event-driven counterpart to CALIBRATE_3P: the three-point calibration with piecewise linear interpolation (Min/Mid/Max) is triggered by its own events EI_MIN/EI_MID/EI_MAX, each with its own confirmation event.

Interface Structure

Event Inputs

- **REQ**: normal execution request – computes Y. Carries X, MIN_REF, MID_REF, MAX_REF, X_MIN, X_MID, X_MAX.
- **EI_MIN**: calibrates the minimum point (X_MIN := X). Carries X.
- **EI_MID**: calibrates the mid point (X_MID := X). Carries X.
- **EI_MAX**: calibrates the maximum point (X_MAX := X). Carries X.

Event Outputs

- **CNF**: confirms REQ, carries Y.
- **EO_MIN**: confirms minimum-point calibration, carries X_MIN.
- **EO_MID**: confirms mid-point calibration, carries X_MID.
- **EO_MAX**: confirms maximum-point calibration, carries X_MAX.

Data Inputs

- **X** (REAL): raw input value.
- **MIN_REF** (REAL, initial 0.0): target output value for the minimum point.
- **MID_REF** (REAL, initial 50.0): target output value for the mid point.
- **MAX_REF** (REAL, initial 100.0): target output value for the maximum point.

Data Outputs

- **Y** (REAL): calibrated, interpolated output value, clipped to MIN_REF..MAX_REF.

In/Out Variables

- **X_MIN** (REAL, initial 0.0): stored raw value at the minimum point.
- **X_MID** (REAL, initial 50.0): stored raw value at the mid point.
- **X_MAX** (REAL, initial 100.0): stored raw value at the maximum point.

Functionality

E_CALIBRATE_3P has a start state START and four action states directly reachable from it:

- **REQ**: interpolates Y piecewise linearly between the stored points (as in CALIBRATE_3P) and fires CNF.
- **C_MIN/C_MID/C_MAX**: store the current raw value X in the corresponding X_MIN/X_MID/X_MAX and fire the matching confirmation event (EO_MIN/EO_MID/EO_MAX).

Each of these four states returns immediately to START, from which all four events are again equally reachable – so the calibration points can be visited in any order and repeated individually as often as needed.

Example (joystick with center drift, raw range 0..1000, desired output range -100..0..100):

Step	Action	Result
1	Move joystick to **minimum**, fire EI_MIN	X_MIN = 50 (drift from ideal 0)
2	Move joystick to **center**, fire EI_MID	X_MID = 520 (drift from ideal 500)
3	Move joystick to **maximum**, fire EI_MAX	X_MAX = 980 (drift from ideal 1000)

Result: Y is interpolated and clipped between MIN_REF=-100, MID_REF=0 and MAX_REF=100.

Technical Details

- **Four equal actions from a shared start state**: there is no dependency between the calibration steps – each point can be re-set independently and in any order.
- **Output always clipped**: Y is clipped to MIN_REF..MAX_REF after interpolation.
- **Degenerate calibration points**: as with CALIBRATE_3P, an interpolation with X_MID <= X_MIN or X_MAX <= X_MID returns a constant MIN_REF or MID_REF instead of dividing by zero.

State Overview

```
START --REQ-----> REQ    --1--> START   (normal operation)
START --EI_MIN--> C_MIN  --1--> START   (calibrate minimum)
START --EI_MID--> C_MID  --1--> START   (calibrate mid)
START --EI_MAX--> C_MAX  --1--> START   (calibrate maximum)
```

All four calibration events are independently reachable from `START` – no enforced order.

Application Scenarios

- Three-point calibration of joysticks in systems that already provide the calibration trigger as an event
- Applications where each calibration point needs its own, immediate confirmation (e.g. for UI feedback "point X calibrated")
- Readjusting individual points (e.g. only `EI_MID` after drift) without needing to recalibrate the other two

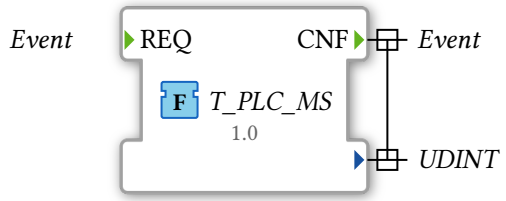
⚖️ Comparison with Similar Blocks

Compare with CALIBRATE_3P, which applies the same three-point logic Boolean-triggered rather than event-driven, and with E_CALIBRATE, which performs a linear two-point rather than three-point calibration. For details see Two- and Three-Point Calibration Procedures.

Conclusion

`E_CALIBRATE_3P` provides event-driven three-point calibration with independently triggerable, individually confirmed calibration points, and fits especially well for joysticks and similar controls with a center position.

T_PLC_MS



T_PLC_MS reads the internal PLC timer and returns the time

Interface

Event inputs

Name	Comment	With
REQ		

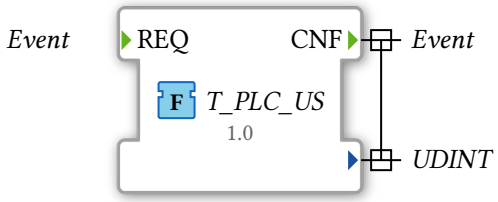
Event Outputs

Name	Comment	With
CNF		

Output Vars

Name	Type	Comment
	UDINT	

T_PLC_US



T_PLC_US reads the internal PLC timer and returns the time

Interface

Event inputs

Name	Comment	With
REQ		

Event Outputs

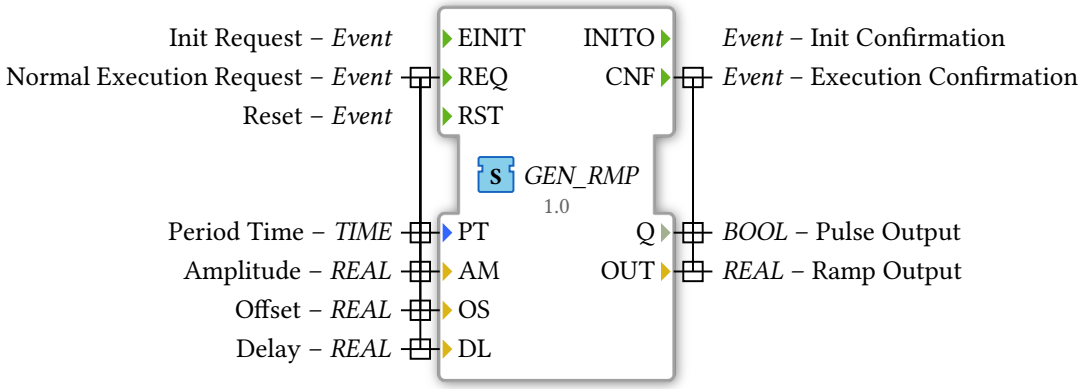
Name	Comment	With
CNF		

Output Vars

Name	Type	Comment
	UDINT	

signal_generators

GEN_RMP



this signal generator generates a ramp wave output this signal generator generates a ramp wave output. The ramp wave signal is defined by period time (PT), amplitude (AM), offset (OS) and a specific delay for the output signal (DL). The Output waveform will have its minimum peak at OS and its maximum peak at AM + OS. The delay input can delay a signal up to PT, this can be useful to synchronize different generators and generate interleaving signals. in addition to the analog output Out there is a second boolean output Q with is true for one cycle when the ramp starts.

Interface

Event inputs

Name	Comment	With

EINIT	Init Request	
REQ	Normal Execution Request	PT, AM, OS, DL
RST	Reset	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	
CNF	Execution Confirmation	Q, OUT

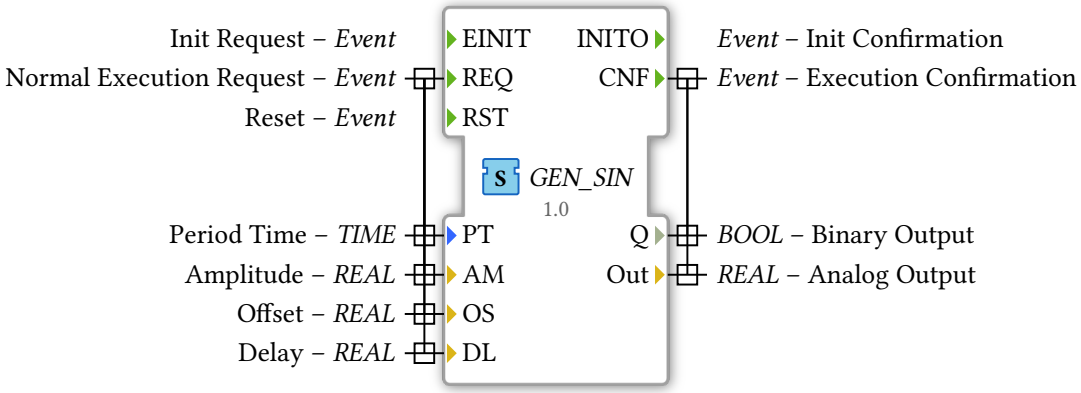
Input Vars

Name	Type	Initial Value	Comment
PT	TIME	T#1s	Period Time
AM	REAL	1.0	Amplitude
OS	REAL		offset
DL	REAL		Delay

Output Vars

Name	Type	Comment
Q	BOOL	Pulse Output
OUT	REAL	Ramp Output

GEN_SIN



this signal generator generates a sine wave output this signal generator generates a sine wave output. The sine wave signal is defined by period time (PT), amplitude (AM), offset (OS) and a specific delay for the output signal (DL). The Output waveform will have its max peak at AM/2 + OS and its minimum peak at -AM/2 + OS. The delay input can delay a signal up to PT, this can be useful to synchronize different generators and generate interleaving signals. A Cos wave can be generated accordingly. in addition to an analog output Out there is a second boolean output Q with the corresponding binary signal.

Interface

Event inputs

Name	Comment	With
EINIT	Init Request	

REQ	Normal Execution Request	PT, AM, OS, DL
RST	Reset	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	
CNF	Execution Confirmation	Q, Out

Input Vars

Name	Type	Initial Value	Comment
PT	TIME		Period Time
AM	REAL	1.0	Amplitude
OS	REAL		offset
DL	REAL		Delay

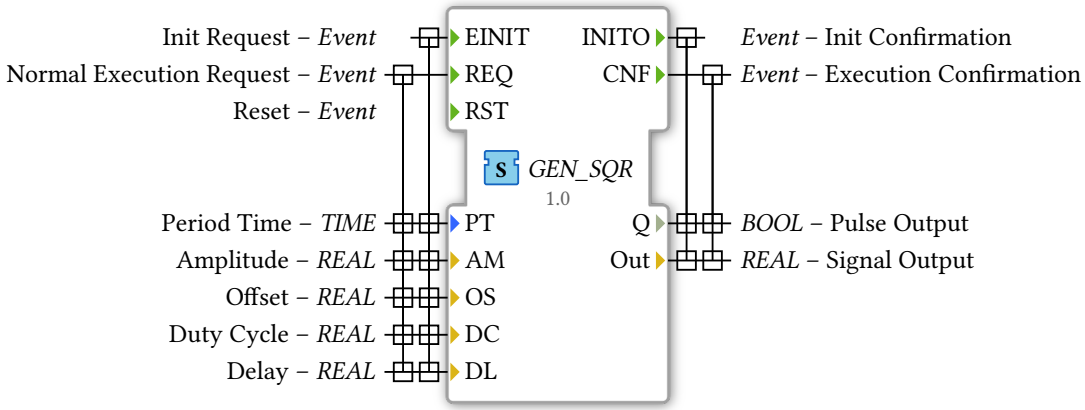
Output Vars

Name	Type	Comment
Q	BOOL	Binary Output
Out	REAL	Analog Output

🛠️ Related exercises

- SIN_TO_PCAN_Callback_DWord
- SIN_TO_PCAN_Callback_Byte

GEN_SQR



this signal generator generates a square wave output this signal generator generates a square wave output. The square wave signal is defined by period time (PT), amplitude (AM), offset (OS), duty cycle (DC) and a specific delay for the output signal (DL). The Output waveform will switch between AM/2 + OS and -AM/2 + OS. The DC input specifies the duty cycle, DC = 0 means output is low at all times and 1 means output is high at all times. The delay input can delay a signal up to PT, this can be useful to synchronize different generators and generate interleaving signals. in addition to an analog output Out there is a second boolean output Q.

Interface

Event inputs

Name	Comment	With
EINIT	Init Request	PT, AM, OS, DC, DL
REQ	Normal Execution Request	PT, AM, OS, DC, DL
RST	Reset	

Event Outputs

Name	Comment	With
INITO	Init Confirmation	Q, Out
CNF	Execution Confirmation	Q, Out

Input Vars

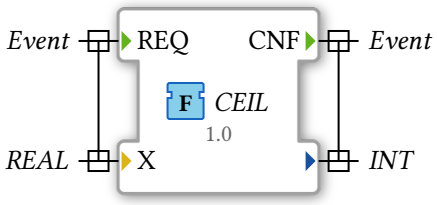
Name	Type	Initial Value	Comment
PT	TIME		Period Time
AM	REAL	1.0	Amplitude
OS	REAL		offset
DC	REAL	0.5	Duty Cycle
DL	REAL		Delay

Output Vars

Name	Type	Comment
Q	BOOL	Pulse Output
Out	REAL	Signal Output

Mathematical

CEIL



• – ◦ • – ◦ • – ◦ •

Introduction

The function `CEIL` is a rounding function (ported from `OSCAT_basic`, `CEIL.TcPOU`) that returns the smallest integer greater than or equal to `X`. The result is clamped to the `INT` range.

Interface Structure

Event Inputs

- `REQ`: Starts the calculation. Associated data: `X`

Event Outputs

- `CNF`: Signals the calculation and the output of the result.

Data Inputs

- `X` (`REAL`): Value to be rounded

Data Outputs

- (no name) (`INT`): Rounded-up value, also the return value of the function

Adapters

This function uses no adapters.

How it works

1. If `X >= 32767.0`, the result is `32767`.
2. If `X <= -32768.0`, the result is `-32768`.
3. Otherwise `X` is converted to an integer with `REAL_TO_INT`. If the result is smaller than `X`, 1 is added.

Examples: `CEIL(3.14) = 4`, `CEIL(-3.14) = -3`.

Technical Details

- Implemented in Structured Text (ST) as a `FUNCTION` without internal state
- Part of the package `OSCAT::Basic::POUs::Mathematical`
- The range check protects against an `INT` overflow (port of 2026-09-17).
- `REAL_TO_INT` is used instead of `TRUNC` (compatibility, revision 1.1 of the OSCAT original).

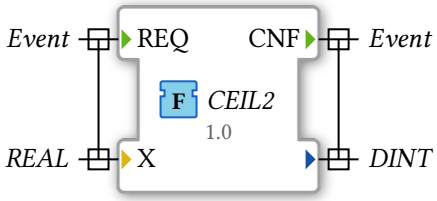
Application Scenarios

- Rounding up numeric values, e.g. for display or count values

Summary

`CEIL` returns the next greater integer of a `REAL` value as `INT` and clamps the result to the `INT` value range.

CEIL2



• – ◦ • – ◦ • – ◦ •

Introduction

`CEIL2` is a rounding function from the OSCAT Basic library: it returns the smallest integer that is greater than or equal to `X` (rounding up). Compared with `CEIL`, the result is a `DINT`, so the function also works outside the `INT` range. Examples: `CEIL2(3.14) = 4`, `CEIL2(-3.14) = -3`. Ported from `OSCAT_basic CEIL2.TcPOU`.

Interface Structure

Event Inputs

- **REQ**: calculation request (associated with `X`)

Event Outputs

- **CNF**: calculation confirmation (associated with the function result)

Data Inputs

- **X** (REAL): value to be rounded

Data Outputs

- **(function result)** (DINT): smallest integer `>= X`, limited to the `DINT` range

Adapters

No adapters are used.

Functionality

The function is implemented in Structured Text:

1. If `X >= 2147483647.0`, `2147483647` is returned.
2. If `X <= -2147483648.0`, `-2147483648` is returned.
3. Otherwise `CEIL2 := REAL_TO_DINT(X)` is calculated; if `DINT_TO_REAL(CEIL2) < X`, the result is increased by 1 (`CEIL2 := CEIL2 + 1`).

Technical Details

- The range check protects against `DINT` overflow: values outside the `DINT` range are limited to the boundary values
- Pure function without state; integrated into the event network as a function block with `REQ`/`CNF`
- The output has no name of its own but is the return value of the function

Application Scenarios

- Rounding up a measured or calculated quantity to the next integer, also for large magnitudes

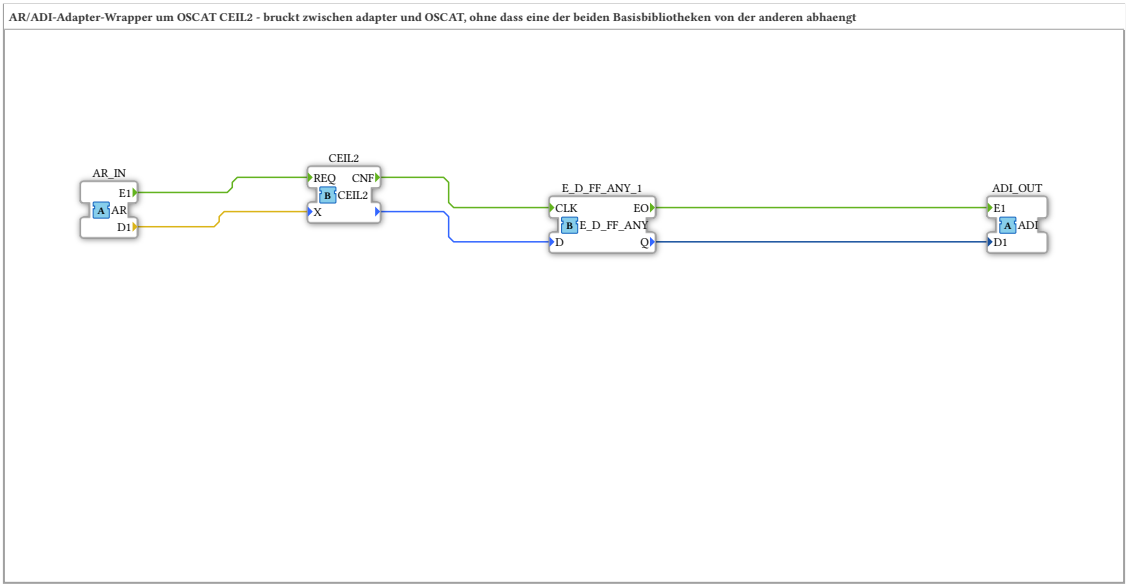
⚖️ Comparison with Similar Blocks

`FLOOR2` is the counterpart for rounding down (largest integer `<= X`).

Conclusion

`CEIL2` rounds a `REAL` value up to the next integer and returns the result as a `DINT` with overflow protection.

CEIL2_AR

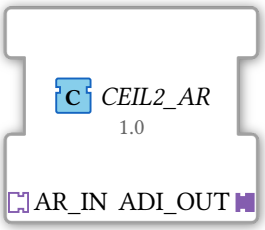


• – ○ • – ○ • – ○ •

Introduction

`CEIL2_AR` is an adapter wrapper around `OSCAT::Basic::POUs::Mathematical::CEIL2` (DINT variant of CEIL for values outside the INT range). It connects the `AR` adapter with the OSCAT function `CEIL2` and outputs the result as an `ADI` adapter (smallest integer greater than or equal to X (DINT)). The wrapper bridges the adapter and OSCAT libraries without either base library depending on the other.

Eingangssignal (REAL) – *AR*



ADI – Kleinste ganze Zahl >= X (DINT)

Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter `AR_IN`.

Event Outputs

No direct event outputs - events leave via the adapter `ADI_OUT`.

Data Inputs

None.

Data Outputs

None.

Adapters

- **AR_IN** (socket, `adapter::types::unidirectional::AR`): input signal (`REAL`)
- **ADI_OUT** (plug, `adapter::types::unidirectional::ADI`): smallest integer greater than or equal to X (DINT)

Function Blocks (FBs) Used

- **`CEIL2`** (`OSCAT::Basic::POUs::Mathematical::CEIL2`): rounding function, see CEIL2
- **`E_D_FF_ANY_1`** (`iec61499::events::E_D_FF_ANY`): change filter

Functionality

- `AR_IN.E1` ⟶ `CEIL2.REQ`; `AR_IN.D1` ⟶ `CEIL2.X`
- `CEIL2.CNF` ⟶ `E_D_FF_ANY_1.CLK`; return value of `CEIL2` ⟶ `E_D_FF_ANY_1.D`
- `E_D_FF_ANY_1.EO` ⟶ `ADI_OUT.E1`; `E_D_FF_ANY_1.Q` ⟶ `ADI_OUT.D1`

Technical Details

- `E_D_FF_ANY_1` (standard library `events-3.0.0`) filters changes: `ADI_OUT.E1` only fires when the result actually changed since the last call. Same pattern as the `FT_DERIV_AR`/`CTRL_PI_AR` wrappers.
- **Why a separate wrapper?** `CEIL2` is a pure OSCAT `Function` without an adapter interface. `adapter` should not depend on `OSCAT` (and vice versa); the bridge therefore lives in the third library project `OSCAT_adapter`, which depends on both.

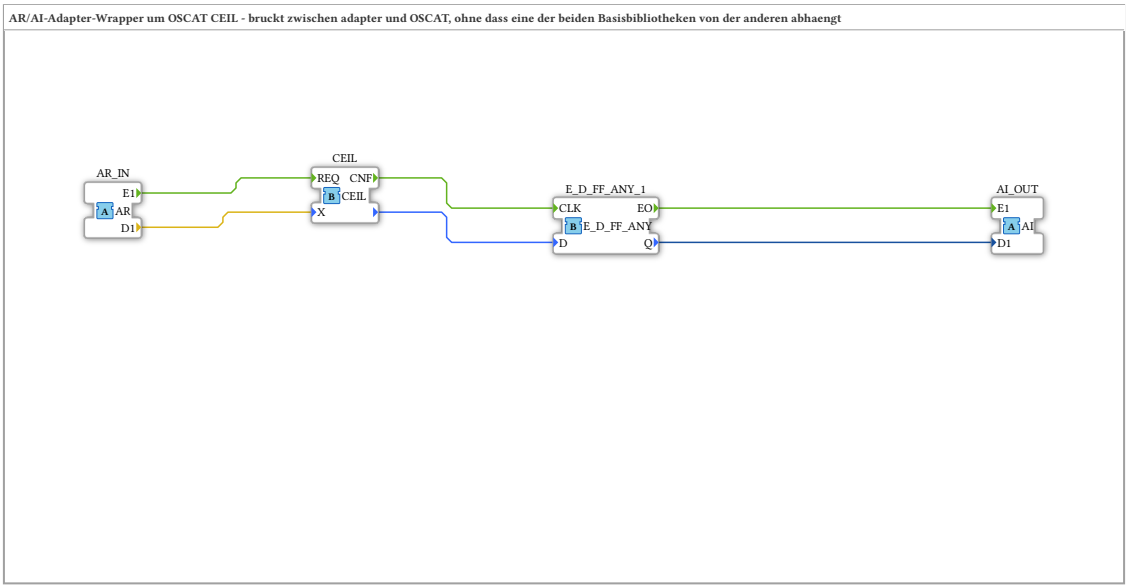
Application Scenarios

- Rounding an `AR` signal in an adapter chain to an integer (`DINT`)

Summary

`CEIL2_AR` rounds an `AR` value with `CEIL2` and outputs the result as an `ADI` adapter only when it changes.

CEIL_AR

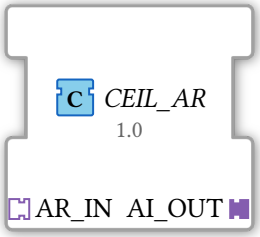


• – ○ • – ○ • – ○ •

Introduction

`CEIL_AR` is an adapter wrapper around `OSCAT::Basic::POUs::Mathematical::CEIL`. It connects the `AR` adapter with the OSCAT function `CEIL` and outputs the result as an `AI` adapter (smallest integer greater than or equal to X (INT)). The wrapper bridges the adapter and OSCAT libraries without either base library depending on the other.

Eingangssignal (REAL) – *AR*



AI – Kleinste ganze Zahl >= X (INT)

Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter `AR_IN`.

Event Outputs

No direct event outputs - events leave via the adapter `AI_OUT`.

Data Inputs

None.

Data Outputs

None.

Adapters

- **AR_IN** (socket, `adapter::types::unidirectional::AR`): input signal (`REAL`)
- **AI_OUT** (plug, `adapter::types::unidirectional::AI`): smallest integer greater than or equal to X (INT)

Function Blocks (FBs) Used

- **`CEIL`** (`OSCAT::Basic::POUs::Mathematical::CEIL`): rounding function, see CEIL

- **E_D_FF_ANY_1** (iec61499::events::E_D_FF_ANY): change filter

Functionality

- AR_IN.E1 ⟶ CEIL.REQ; AR_IN.D1 ⟶ CEIL.X
- CEIL.CNF ⟶ E_D_FF_ANY_1.CLK; return value of CEIL ⟶ E_D_FF_ANY_1.D
- E_D_FF_ANY_1.EO ⟶ AI_OUT.E1; E_D_FF_ANY_1.Q ⟶ AI_OUT.D1

Technical Details

- E_D_FF_ANY_1 (standard library events-3.0.0) filters changes: AI_OUT.E1 only fires when the result actually changed since the last call. Same pattern as the FT_DERIV_AR/CTRL_PI_AR wrappers.
- **Why a separate wrapper?** CEIL is a pure OSCAT Function without an adapter interface. adapter should not depend on OSCAT (and vice versa); the bridge therefore lives in the third library project OSCAT_adapter, which depends on both.

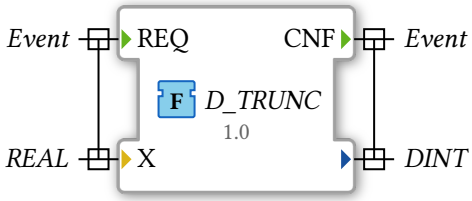
Application Scenarios

- Rounding an AR signal in an adapter chain to an integer (INT)

Summary

CEIL_AR rounds an AR value with CEIL and outputs the result as an AI adapter only when it changes.

D_TRUNC



d_trunc truncates a real to a dint 1.5 will be 1 and −1.5 will be −1

Interface

Event inputs

Name	Comment	With
REQ		X

Event Outputs

Name	Comment	With
CNF		

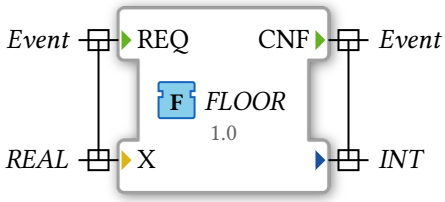
Input Vars

Name	Type	Comment
X	REAL	

Output Vars

Name	Type	Comment
	DINT	

FLOOR



🎧 Podcast

- Unlocking Universal Automation: The IEC 61499 Revolution from Factory Floors to the Seas

Interface

Event inputs

Name	Comment	With
REQ		X

Event Outputs

Name	Comment	With
CNF		

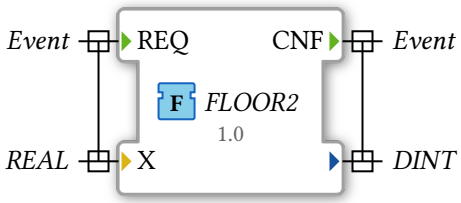
Input Vars

Name	Type	Comment
X	REAL	

Output Vars

Name	Type	Comment
	INT	

FLOOR2



This is a rounding function which returns the largest possible integer which is less or equal to X

Interface

Event inputs

Name	Comment	With
REQ		X

Event Outputs

Name	Comment	With

CNF		

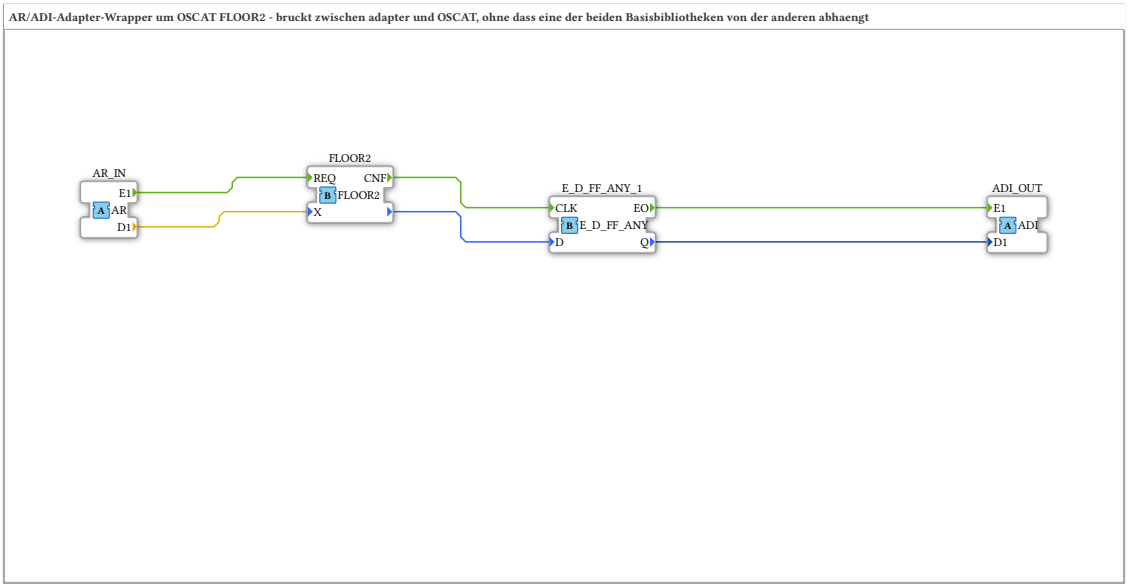
Input Vars

Name	Type	Comment
X	REAL	

Output Vars

Name	Type	Comment
	DINT	

FLOOR2_AR

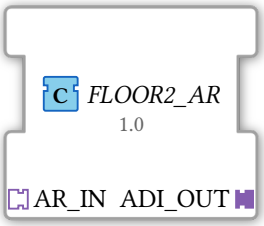


• – ○ • – ○ • – ○ •

Introduction

`FLOOR2_AR` is an AR/ADI adapter wrapper around the OSCAT function `OSCAT::Basic::POUs::Mathematical::FLOOR2` (largest integer `<= X`, `DINT` variant of `FLOOR` for values outside the `INT` range). It bridges between adapters and OSCAT without either base library depending on the other.

Eingangssignal (REAL) – *AR*



ADI – Groesste ganze Zahl <= X (DINT)

Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AR_IN** (socket, `adapter::types::unidirectional::AR`): input signal (REAL)
- **ADI_OUT** (plug, `adapter::types::unidirectional::ADI`): largest integer <= `X` (DINT)

Functionality

- `AR_IN.E1` ⟶ `FLOOR2.REQ` and `AR_IN.D1` ⟶ `FLOOR2.X`: every new input signal starts the calculation.
- `FLOOR2.CNF` ⟶ `E_D_FF_ANY_1.CLK`; the function result goes to `E_D_FF_ANY_1.D`.
- `E_D_FF_ANY_1` (`iec61499::events::E_D_FF_ANY`) filters changes: `E_D_FF_ANY_1.EO` ⟶ `ADI_OUT.E1` and `E_D_FF_ANY_1.Q` ⟶ `ADI_OUT.D1`, but only if the result has actually changed since the last call.

Technical Details

- Change filter: `ADI_OUT.E1` fires only on an actual value change (same pattern as `CTRL_PI_AR`/ `FT_DERIV_AR`)
- `FLOOR2` is a pure OSCAT function without an adapter interface; `adapter` should not depend on `OSCAT` (and vice versa), so the bridge lives in the third library project `OSCAT_adapter`

Application Scenarios

- Rounding down an adapter-based REAL value to an integer (DINT) in an adapter network

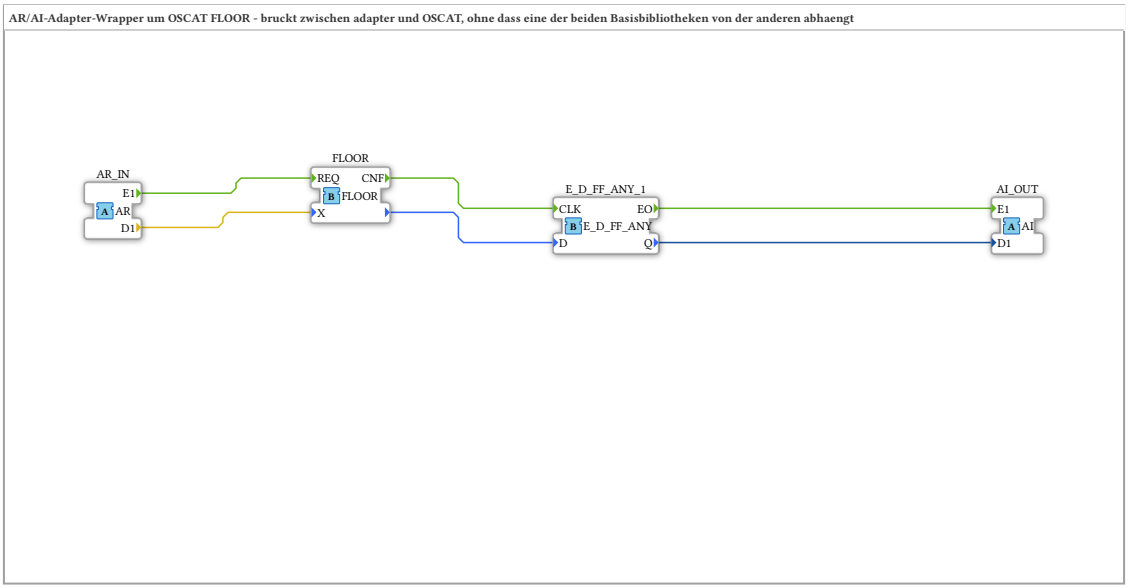
⚖️ Comparison with Similar Blocks

`ROUND_AR` rounds to decimal places (REAL output); `FLOOR2_AR` returns the rounded-down integer as a DINT.

Conclusion

`FLOOR2_AR` provides `FLOOR2` as an adapter block from `AR` to `ADI` and outputs the result only on change.

FLOOR_AR



• – ○ • – ○ • – ○ •

Introduction

`FLOOR_AR` is an adapter wrapper around `OSCAT::Basic::POUs::Mathematical::FLOOR`. It connects the `AR` adapter with the OSCAT function `FLOOR` and outputs the result as an `AI` adapter (largest integer less than

or equal to X (INT)). The wrapper bridges the adapter and OSCAT libraries without either base library depending on the other.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter `AR_IN`.

Event Outputs

No direct event outputs - events leave via the adapter `AI_OUT`.

Data Inputs

None.

Data Outputs

None.

Adapters

- **AR_IN** (socket, `adapter::types::unidirectional::AR`): input signal (`REAL`)
- **AI_OUT** (plug, `adapter::types::unidirectional::AI`): largest integer less than or equal to X (INT)

Function Blocks (FBs) Used

- **`FLOOR`** (`OSCAT::Basic::POUs::Mathematical::FLOOR`): rounding function, see FLOOR
- **`E_D_FF_ANY_1`** (`iec61499::events::E_D_FF_ANY`): change filter

Functionality

- `AR_IN.E1` ⟶ `FLOOR.REQ`; `AR_IN.D1` ⟶ `FLOOR.X`
- `FLOOR.CNF` ⟶ `E_D_FF_ANY_1.CLK`; return value of `FLOOR` ⟶ `E_D_FF_ANY_1.D`
- `E_D_FF_ANY_1.EO` ⟶ `AI_OUT.E1`; `E_D_FF_ANY_1.Q` ⟶ `AI_OUT.D1`

Technical Details

- `E_D_FF_ANY_1` (standard library `events-3.0.0`) filters changes: `AI_OUT.E1` only fires when the result actually changed since the last call. Same pattern as the `FT_DERIV_AR`/`CTRL_PI_AR` wrappers.
- **Why a separate wrapper?** `FLOOR` is a pure OSCAT `Function` without an adapter interface. `adapter` should not depend on `OSCAT` (and vice versa); the bridge therefore lives in the third library project `OSCAT_adapter`, which depends on both.

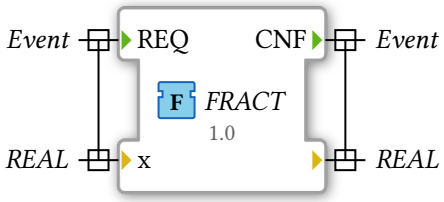
Application Scenarios

- Rounding an `AR` signal in an adapter chain to an integer (`INT`)

Summary

`FLOOR_AR` rounds an `AR` value with `FLOOR` and outputs the result as an `AI` adapter only when it changes.

FRACT



this function returns the fraction of a real number

Interface

Event inputs

Name	Comment	With
REQ		x

Event Outputs

Name	Comment	With
CNF		

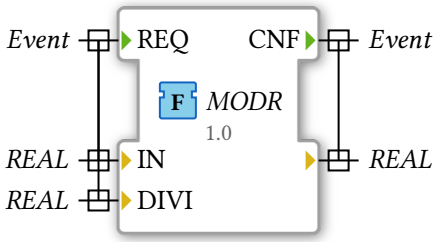
Input Vars

Name	Type	Comment
x	REAL	

Output Vars

Name	Type	Comment
	REAL	

MODR



this is a modulo function for real numbers

Interface

Event inputs

Name	Comment	With
REQ		IN, DIVI

Event Outputs

Name	Comment	With
CNF		

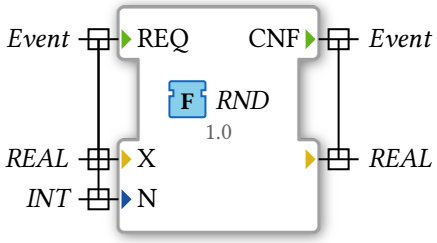
Input Vars

Name	Type	Comment
IN	REAL	
DIVI	REAL	

Output Vars

Name	Type	Comment
	REAL	

RND



• – ◦ • – ◦ • – ◦ •

Introduction

The function `RND` rounds a `REAL` number to `N` **significant** digits (not to digits behind the decimal point). It is ported from `OSCAT_basic`. Unlike `ROUND`, `N` here denotes the total number of significant digits.

Interface Structure

Event Inputs

- **REQ**: starts the calculation. Associated data: `X`, `N`.

Event Outputs

- **CNF**: signals the calculation and the output of the result.

Data Inputs

- **X** (`REAL`): value to round
- **N** (`INT`): number of significant digits

Data Outputs

- **(unnamed)** (`REAL`): rounded value, also the return value of the function

Adapters

This function uses no adapters.

Functionality

1. If `X = 0.0`, the result is `0.0`.
2. Otherwise the factor `M = 10^(LIMIT(1, N, 8) - CEIL(LOG(ABS(X))))` is calculated; the exponent is limited to `-37..8`.
3. If `ABS(X * M) >= 2147483647.0`, `X` is returned unchanged (protection against `DINT` overflow).
4. Otherwise the result is `DINT_TO_REAL(REAL_TO_DINT(X * M)) / M`.

Technical Details

- Implemented in Structured Text (ST) as a `FUNCTION` without internal state, package `OSCAT::Basic::POUs::Mathematical`
- `N` is limited to `1..8`

- Port of 2026-09-17: limiting of `N` (1..8), of the exponent (-37..8) and a range check protect against `DINT` overflow
- Original version 1.1 (4 Apr 2008, hm): type conversion added to avoid warnings under CODESYS 3.0

Application Scenarios

- Rounding measured values to a fixed number of significant digits regardless of magnitude

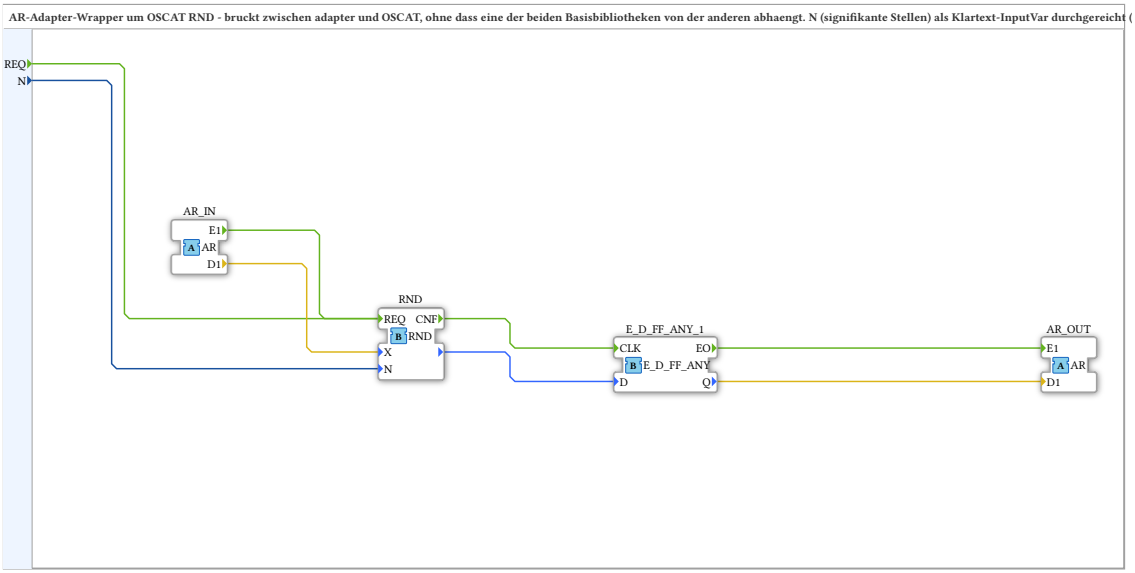
⚖️ Comparison with Similar Building Blocks

- `ROUND` rounds to `N` decimal places, `RND` to `N` significant digits.

Conclusion

`RND` rounds a `REAL` number to `N` significant digits and protects against `DINT` overflow.

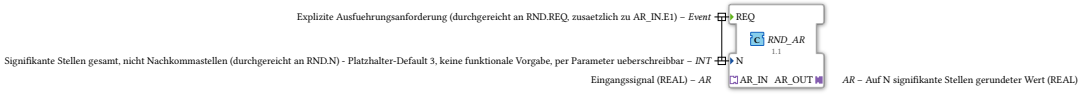
RND_AR



• – ○ • – ○ • – ○ •

Introduction

`RND_AR` is an AR adapter wrapper around the OSCAT function `OSCAT::Basic::POUs::Mathematical::RND` (rounds `X` to `N` **significant digits**, not to decimal places). It bridges adapters and OSCAT without either base library depending on the other. `N` is passed through as a plain input variable (default 3).



Interface Structure

Event Inputs

- **REQ**: explicit execution request (passed through to `RND.REQ`, in addition to `AR_IN.E1`)
  - **Associated data**: `N`

Event Outputs

The function block has no direct event outputs.

Data Inputs

- **N** (`INT`): Total significant digits, not decimal places (passed through to `RND.N`); placeholder default 3, no functional requirement, can be overridden via parameter

Data Outputs

The function block has no direct data outputs.

Adapters

- **AR_IN** (socket, `adapter::types::unidirectional::AR`): Input signal (`REAL`)
- **AR_OUT** (plug, `adapter::types::unidirectional::AR`): Value rounded to `N` significant digits (`REAL`)

Function Blocks (FBs) Used

- **`RND`** (`OSCAT::Basic::POUs::Mathematical::RND`): rounding function, see RND
- **`E_D_FF_ANY_1`** (`iec61499::events::E_D_FF_ANY`): change filter

How it works

- `AR_IN.E1` ⟶ `RND.REQ` and `AR_IN.D1` ⟶ `RND.X`; `N` ⟶ `RND.N`.
- `REQ` also leads to `RND.REQ`, in parallel to `AR_IN.E1`.
- `RND.CNF` ⟶ `E_D_FF_ANY_1.CLK`; the function result goes to `E_D_FF_ANY_1.D`.
- `E_D_FF_ANY_1.EO` ⟶ `AR_OUT.E1` and `E_D_FF_ANY_1.Q` ⟶ `AR_OUT.D1`, but only if the rounded result has actually changed since the last call.

Technical Details

- The explicit `REQ` with `With N` was added as a bugfix in version 1.1 (2026-09-20): previously `N` was only passed to `RND` via a data connection and triggered solely via `AR_IN.E1`; since the wrapper had no event input, there was no `With` binding (per IEC 61499, unbound variables are not read reliably). `REQ` allows manual or periodic execution in addition to automatic triggering via `AR_IN.E1`
- Change filter: `AR_OUT.E1` fires only when the rounded value actually changes (same pattern as `CTRL_PI_AR`/`FT_DERIV_AR`, there using `E_D_FF_ANY_3`/`_4` for several values at once)
- `RND` is a pure OSCAT function without an adapter interface; `adapter` should not depend on `OSCAT` (and vice versa), so the bridge lives in the third library project `OSCAT_adapter`
- Difference to `ROUND_AR`: `RND_AR` rounds to significant digits, `ROUND_AR` to decimal places

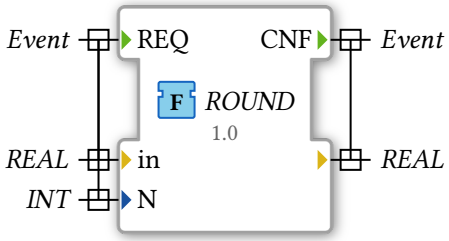
Application Scenarios

- Rounding an `AR` signal to a fixed number of significant digits in an adapter chain, e.g. for display values

Summary

`RND_AR` rounds an `AR` value to `N` significant digits with `RND` and outputs the result as an `AR` adapter only when it changes.

ROUND



• – ◦ • – ◦ • – ◦ •

Introduction

The function `ROUND` rounds a `REAL` value to `N` digits after the decimal point (0 to 8). It was ported from `OSCAT_basic`.

Interface Structure

Event Inputs

- `REQ`: Starts the calculation. Associated data: `in`, `N`

Event Outputs

- `CNF`: Signals the calculation and the output of the result.

Data Inputs

- `in` (`REAL`): Value to be rounded
- `N` (`INT`): Number of decimal places (limited to 0 to 8)

Data Outputs

- (no name) (`REAL`): Rounded value, also the return value of the function

Adapters

This function uses no adapters.

How it works

1. The factor `X := EXPT(10.0, LIMIT(0, N, 8))` is determined.
2. If `ABS(in * X) >= 2147483647.0` (DINT range), `in` is returned unchanged.
3. Otherwise the result is `DINT_TO_REAL(REAL_TO_DINT(in * X)) / X`.

Technical Details

- Implemented in Structured Text (ST) as a `FUNCTION` with local variable `X`, without internal state
- Part of the package `OSCAT::Basic::POUs::Mathematical`
- `EXPT(10.0, LIMIT(0, N, 8))` replaces the global OSCAT constants `decades`; the range check protects against a `DINT` overflow (port of 2026-09-17).
- `REAL_TO_DINT` instead of `TRUNC` fixes the error with negative numbers (OSCAT revision 1.3).
- `N` is limited to 0 to 8.

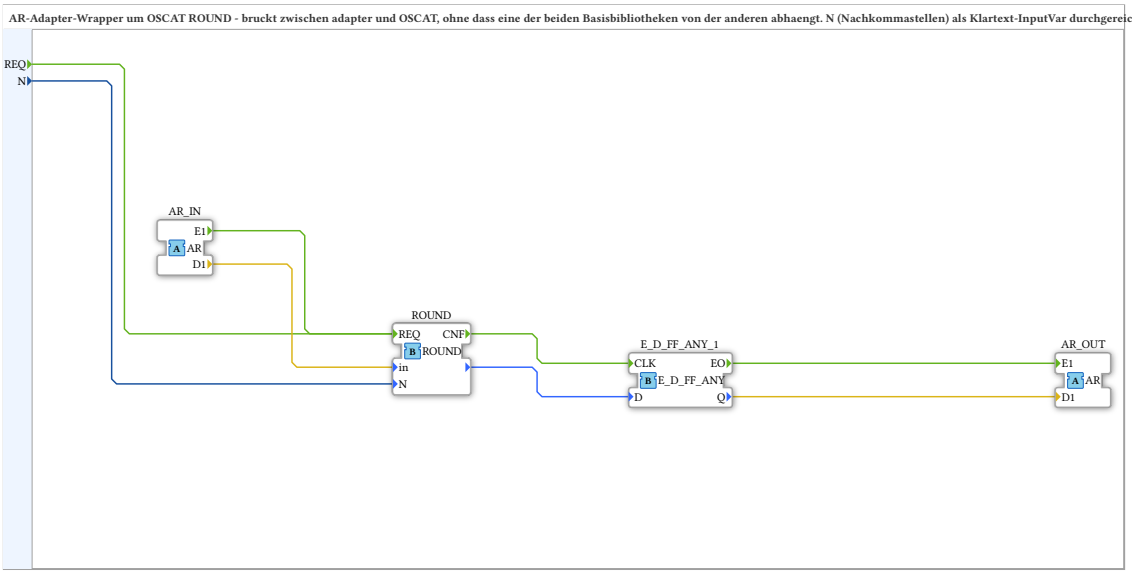
Application Scenarios

- Rounding measured values for display or comparison to a fixed number of decimal places

Summary

`ROUND` rounds a `REAL` value to up to 8 decimal places and returns the input unchanged if the value is too large.

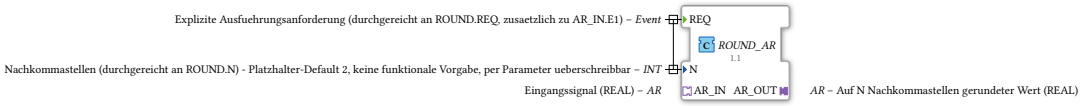
ROUND_AR



• — ○ • — ○ • — ○ •

Introduction

`ROUND_AR` is an AR adapter wrapper around the OSCAT function `OSCAT::Basic::POUs::Mathematical::ROUND` (rounds `in` to `N` decimal places, 0 to 8 digits). It bridges between adapters and OSCAT without either base library depending on the other. `N` is passed through as a plain input variable (default 2).



Interface Structure

Event Inputs

- **REQ**: explicit execution request (passed on to `ROUND.REQ`, in addition to `AR_IN.E1`)
  - **Associated data**: `N`

Event Outputs

The function block has no direct event outputs.

Data Inputs

- **N** (INT): number of decimal places (passed on to `ROUND.N`); placeholder default 2, overridable via parameter

Data Outputs

The function block has no direct data outputs.

Adapters

- **AR_IN** (socket, `adapter::types::unidirectional::AR`): input signal (REAL)
- **AR_OUT** (plug, `adapter::types::unidirectional::AR`): value rounded to `N` decimal places (REAL)

Functionality

- `AR_IN.E1` ⟶ `ROUND.REQ` and `AR_IN.D1` ⟶ `ROUND.in`; `N` ⟶ `ROUND.N`.
- `REQ` also goes to `ROUND.REQ`, in parallel to `AR_IN.E1`.
- `ROUND.CNF` ⟶ `E_D_FF_ANY_1.CLK`; the function result goes to `E_D_FF_ANY_1.D`.
- `E_D_FF_ANY_1` (`iec61499::events::E_D_FF_ANY`) filters changes: `E_D_FF_ANY_1.EO` ⟶ `AR_OUT.E1` and `E_D_FF_ANY_1.Q` ⟶ `AR_OUT.D1`, but only if the rounded result has actually changed since the last call.

Technical Details

- The explicit `REQ` with `With N` was added because `N` would otherwise only be read via a data connection; per IEC 61499, variables not bound via `With` are not read reliably. `REQ` allows manual or periodic execution in addition to automatic triggering via `AR_IN.E1`
- Change filter: `AR_OUT.E1` fires only when the rounded value actually changes (same pattern as `CTRL_PI_AR`/`FT_DERIV_AR`, there with `E_D_FF_ANY_3`/`_4`)
- `ROUND` is a pure OSCAT function without an adapter interface; the bridge lives in the third library project `OSCAT_adapter`

Application Scenarios

- Rounding displayed or forwarded measured values to a fixed number of decimal places in adapter networks

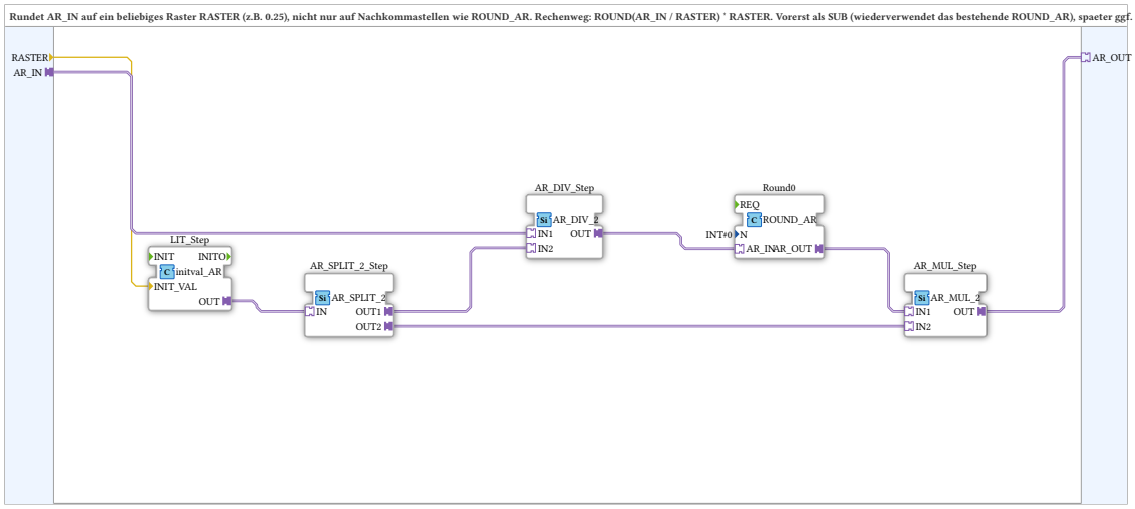
⚖️ Comparison with Similar Blocks

`ROUND_STEP_AR` rounds to an arbitrary grid (e.g. 0.25) instead of only to decimal places and uses `ROUND_AR` internally (`ROUND(AR_IN / RASTER) * RASTER`). `FLOOR2_AR`, in contrast, returns a rounded-down integer as a DINT.

Conclusion

`ROUND_AR` provides `ROUND` as an adapter block from `AR` to `AR`, rounds to `N` decimal places and outputs the result only on change.

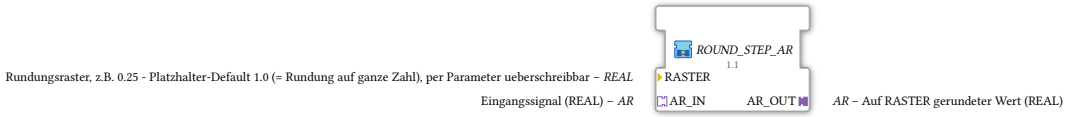
ROUND_STEP_AR



• — ○ • — ○ • — ○ •

Introduction

`ROUND_STEP_AR` rounds `AR_IN` to an **arbitrary grid** `RASTER` (e.g. 0.25), not only to decimal places like `ROUND_AR`. Calculation: `ROUND(AR_IN / RASTER) * RASTER`. The block is for now a SubApp that reuses the existing `ROUND_AR`; possibly a separate basic block later.



Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

- **RASTER** (`REAL`): rounding grid, e.g. `0.25`. Placeholder default `1.0` (= rounding to an integer), can be overridden by parameter.

Data Outputs

None.

Adapters

- **AR_IN** (socket, `adapter::types::unidirectional::AR`): input signal (`REAL`)
- **AR_OUT** (plug, `adapter::types::unidirectional::AR`): value rounded to `RASTER` (`REAL`)

Function Blocks (FBs) Used

- **`LIT_Step`** (`adapter::types::unidirectional::AR::initval::initval_AR`): creates a constant `AR` value from `RASTER`
- **`AR_SPLIT_2_Step`** (`adapter::events::unidirectional::AR_SPLIT_2`): fans `RASTER` out to the division and the multiplication (adapter fan-out is strictly 1:1)
- **`AR_DIV_Step`** (`adapter::iec61131::arithmetic::AR_DIV_2`): `AR_IN / RASTER`
- **`Round0`** (`OSCAT_adapter::Mathematical::ROUND_AR`, `N = 0`): rounds the quotient to an integer
- **`AR_MUL_Step`** (`adapter::iec61131::arithmetic::AR_MUL_2`): multiplies back by `RASTER`

Functionality

- `RASTER` ⟶ `LIT_Step.INIT_VAL`; `LIT_Step.OUT` ⟶ `AR_SPLIT_2_Step.IN`
- `AR_IN` ⟶ `AR_DIV_Step.IN1`; `AR_SPLIT_2_Step.OUT1` ⟶ `AR_DIV_Step.IN2`
- `AR_DIV_Step.OUT` ⟶ `Round0.AR_IN`
- `Round0.AR_OUT` ⟶ `AR_MUL_Step.IN1`; `AR_SPLIT_2_Step.OUT2` ⟶ `AR_MUL_Step.IN2`
- `AR_MUL_Step.OUT` ⟶ `AR_OUT`

Technical Details

- Rounding to a grid instead of decimal places
- `RASTER` is set by parameter at instantiation
- Example: `RASTER = 0.25` rounds to quarters

Application Scenarios

- Rounding setpoints or display values to step sizes (e.g. 0.25, 0.5, 5)

⚖️ Comparison with Similar Building Blocks

- `ROUND_AR` only rounds to `N` decimal places and is used internally here.

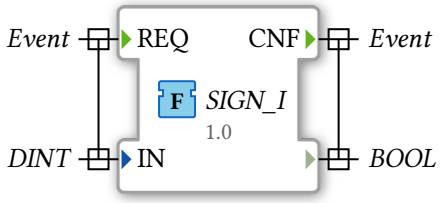
Summary

`ROUND_STEP_AR` rounds an `AR` signal to a freely selectable grid.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SIGN_I



this function returns true if the integer input is negative

Interface

Event inputs

Name	Comment	With
REQ		IN

Event Outputs

Name	Comment	With
CNF		

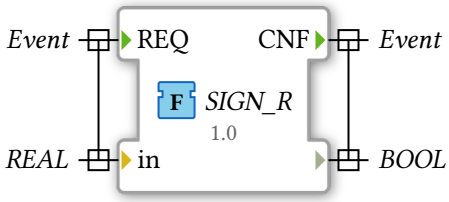
Input Vars

Name	Type	Comment
IN	DINT	

Output Vars

Name	Type	Comment
	BOOL	

SIGN_R



this function returns true if the real input is negative

Interface

Event inputs

Name	Comment	With
REQ		in

Event Outputs

Name	Comment	With
CNF		

Input Vars

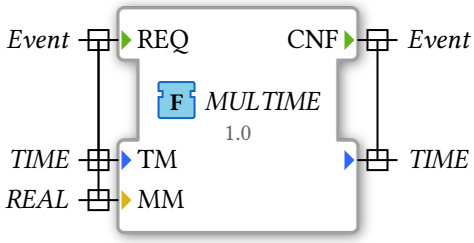
Name	Type	Comment
in	REAL	

Output Vars

Name	Type	Comment
	BOOL	

Time_Date

MULTIME



multiplies a time by a real number and returns a time

Interface

Event inputs

Name	Comment	With
REQ		TM, MM

Event Outputs

Name	Comment	With

CNF		

Input Vars

Name	Type	Comment
TM	TIME	
MM	REAL	

Output Vars

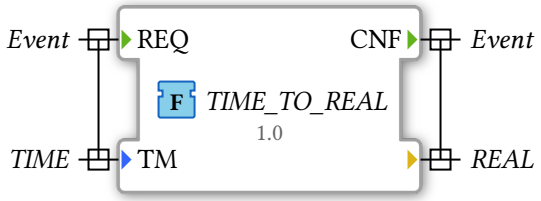
Name	Type	Comment
	TIME	

🛠️ Related exercises

- Uebung_020c2

conversion

TIME_TO_REAL



Convert TIME to REAL

Interface

Event inputs

Name	Comment	With
REQ		TM

Event Outputs

Name	Comment	With
CNF		

Input Vars

Name	Type	Comment
TM	TIME	

Output Vars

Name	Type	Comment
	REAL	

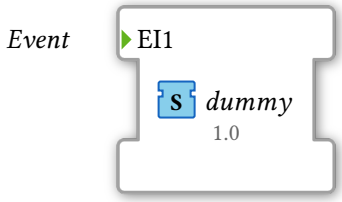
2.1.8.2 Building

🎧 Podcast

- Building Tomorrow's Factories: Bridging OT and IT with IEC 61499

POUs

dummy



does nothing

Interface

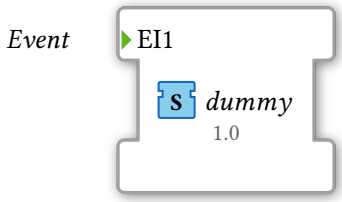
Event inputs

Name	Comment	With
EI1		

2.1.8.3 Network

POUs

dummy



does nothing

Interface

Event inputs

Name	Comment	With
EI1		

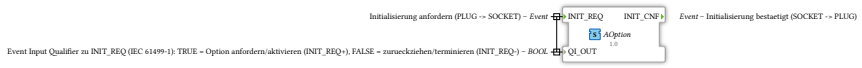
2.1.9 Option

2.1.9.1 AOption

Plug (adapter output)



Socket (adapter input)



• – ○ • – ○ • – ○ •

Introduction

The `AOption` adapter (library `Option`) is an option management adapter: it carries an INIT request/confirm handshake with the event input qualifier `QI` (IEC 61499-1). The plug side (requester) can request/activate (`QI=TRUE`) or withdraw/terminate (`QI=FALSE`) an option at the socket side (option group); the socket acknowledges with `INIT_CNF`.

Interface Structure

Event Inputs

Event	Comment
`INIT_CNF`	Initialization confirmed (SOCKET -> PLUG).

Event Outputs

Event	Comment
`INIT_REQ`	Request initialization (PLUG -> SOCKET). Carries the data `QI_OUT`.

Data Inputs

None.

Data Outputs

Variable	Type	Comment
`QI_OUT`	BOOL	Event input qualifier for `INIT_REQ` (IEC 61499-1): `TRUE` = request/activate option (INIT_REQ+), `FALSE` = withdraw/terminate (INIT_REQ-).

(The directions are given from the plug side; they are mirrored on the socket side.)

Functionality

The plug sends `INIT_REQ` together with `QI_OUT`. The socket evaluates `QI_OUT` (activate or deactivate the option group) and answers with `INIT_CNF` when done. The adapter itself contains no logic; it only defines the interface. To connect it in your own networks, use the converters `AOption_EVENT_TO_X` (holds the plug) and `AOption_X_TO_EVENT` (holds the socket).

Technical Features

- **Handshake**: every request is acknowledged with `INIT_CNF` (request/confirm).
- **Data binding**: `QI_OUT` is bound to the event `INIT_REQ` via `With`.
- **No state machine**: the adapter is a pure interface type.

Application Scenarios

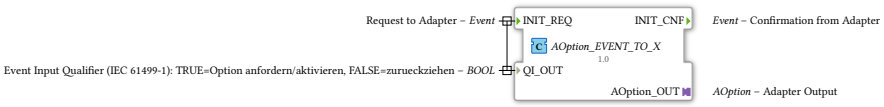
- Switching optional function groups (e.g. groups containing I/O blocks) on and off via a selection block such as `AOption_AD_DEMUX_2`.

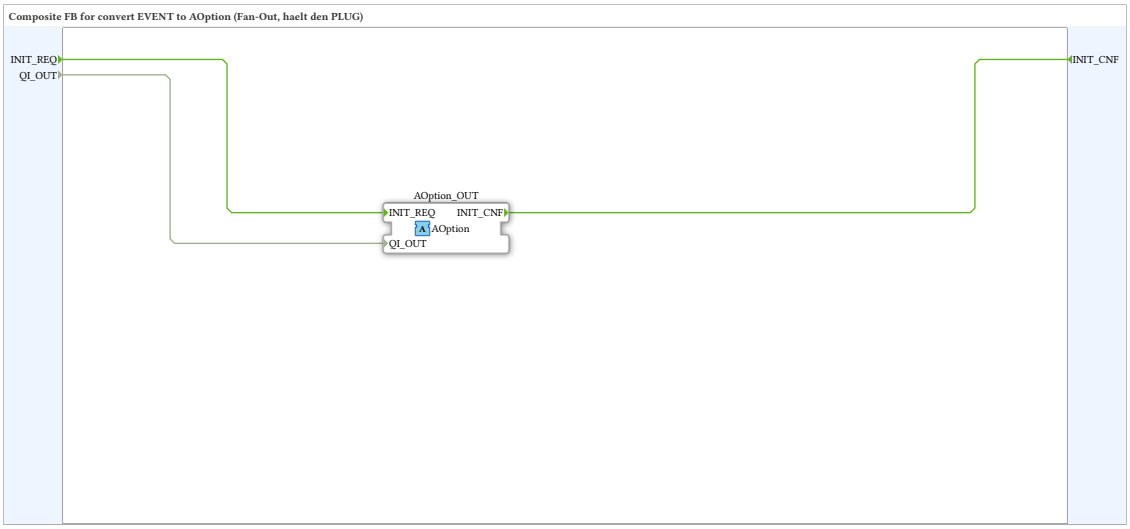
Conclusion

`AOption` provides a lean INIT_REQ/INIT_CNF handshake with `QI` for activating and deactivating optional parts of an application.

2.1.9.2 conversion

AOption_EVENT_TO_X





• – ○ • – ○ • – ○ •

Introduction

`AOption_EVENT_TO_X` is a composite FB that maps events and data onto the option adapter `AOption` (fan-out, holds the PLUG).

Interface Structure

Event Inputs

Event	Type	Comment
`INIT_REQ`	EInit	Request to the adapter. Carries the data `QI_OUT`.

Event Outputs

Event	Type	Comment
`INIT_CNF`	EInit	Confirmation from the adapter.

Data Inputs

Variable	Type	Comment
`QI_OUT`	BOOL	Event input qualifier (IEC 61499-1): `TRUE` = request/activate option, `FALSE` = withdraw.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
`AOption_OUT`	`Option::AOption`	Plug	Adapter output.

Functionality

The internal network consists of connections only (no further FBs):

- `INIT_REQ -> AOption_OUT.INIT_REQ`
- `AOption_OUT.INIT_CNF -> INIT_CNF`
- `QI_OUT -> AOption_OUT.QI_OUT`

A request is thus passed on unchanged to the adapter, and the socket's confirmation returns as `INIT_CNF`.

Technical Features

- Pure pass-through without logic.
- The counterpart is `AOption_X_TO_EVENT`, which holds the SOCKET.

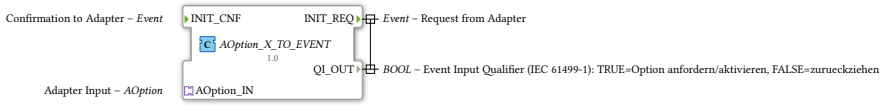
Application Scenarios

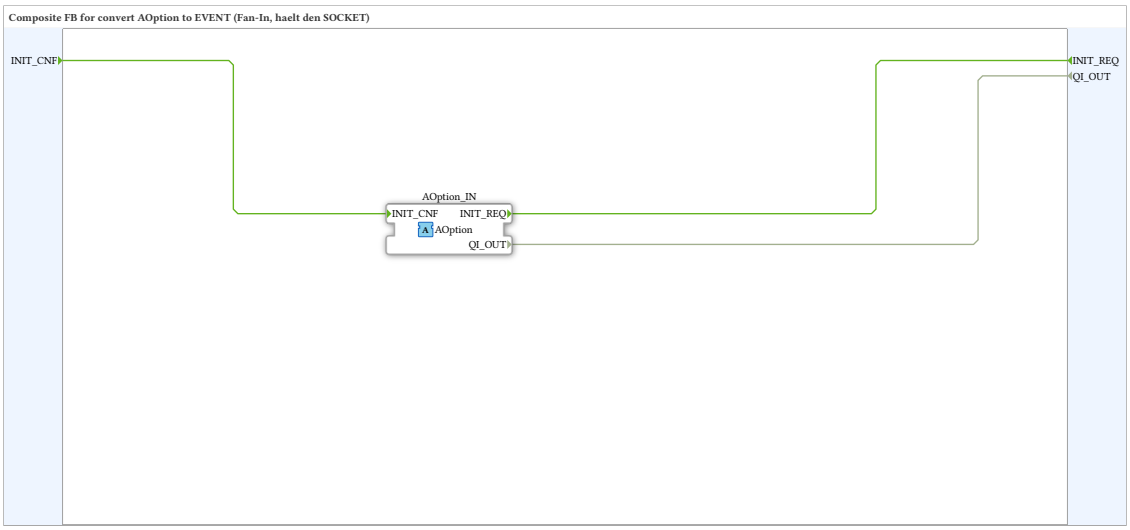
- A requester network sets an option at an option group via this block.

Conclusion

`AOption_EVENT_TO_X` exposes the plug of the `AOption` adapter as an ordinary event/data interface (`INIT_REQ`, `QI_OUT`, `INIT_CNF`).

AOption_X_TO_EVENT





• – ○ • – ○ • – ○ •

Introduction

`AOption_X_TO_EVENT` is a composite FB that converts the option adapter `AOption` into events and data (fan-in, holds the SOCKET). It is used in option groups to receive and acknowledge the request (`INIT_REQ`, `QI_OUT`).

Interface Structure

Event Inputs

Event	Type	Comment
`INIT_CNF`	EInit	Confirmation to the adapter.

Event Outputs

Event	Type	Comment
`INIT_REQ`	EInit	Request from the adapter. Carries the data `QI_OUT`.

Data Inputs

None.

Data Outputs

Variable	Type	Comment
QI_OUT	BOOL	Event input qualifier (IEC 61499-1): TRUE = request/activate option, FALSE = withdraw.

Adapters

Adapter	Type	Direction	Comment
AOption_IN	Option::AOption	Socket	Adapter input.

Functionality

The internal network consists of connections only (no further FBs):

- AOption_IN.INIT_REQ -> INIT_REQ
- INIT_CNF -> AOption_IN.INIT_CNF
- AOption_IN.QI_OUT -> QI_OUT

The plug's request appears as the event INIT_REQ with QI_OUT; once the group is initialized, INIT_CNF is fed into the block and reported back to the plug.

Technical Features

- Pure pass-through without logic.
- The counterpart is AOption_EVENT_TO_X, which holds the PLUG.
- The SubApps OptionalGroup_001 and DummyGroup use it.

Application Scenarios

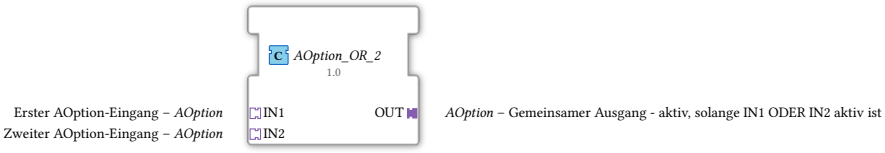
- Option group: QI_OUT activates/deactivates the group, INIT_CNF acknowledges the activation.

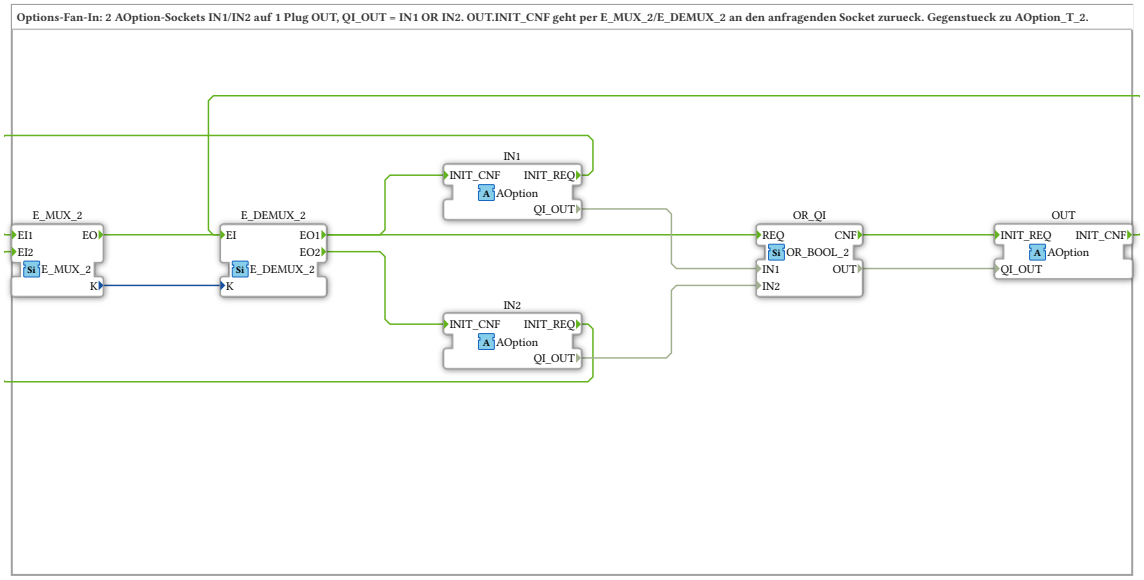
Conclusion

AOption_X_TO_EVENT exposes the socket of the AOption adapter as an ordinary event/data interface (INIT_REQ, QI_OUT, INIT_CNF).

2.1.9.3 Merging

AOption_OR_2





• – ◦ • – ◦ • – ◦ •

Introduction

`AOption_OR_2` is a composite FB that serves as a **fan-in** (OR combination) for the option adapter `AOption`: two option requests at the sockets `IN1` and `IN2` are merged into a single request at the plug `OUT`. `OUT` is active as long as `IN1` **or** `IN2` is active. The block is the counterpart of `AOption_T_2`.

Interface Structure

Event Inputs

None (the events run through the adapters).

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
`IN1`	`Option::AOption`	Socket	First AOption input.
`IN2`	`Option::AOption`	Socket	Second AOption input.
`OUT`	`Option::AOption`	Plug	Common output - active as long as `IN1` OR `IN2` is active.

Functionality

The internal network contains three FBs: `E_MUX_2` (`iec61499::events::E_MUX_2`), `OR_QI` (`iec61131::booleanOperators::OR_BOOL_2`) and `E_DEMUX_2` (`iec61499::events::E_DEMUX_2`).

Events:

- `IN1.INIT_REQ` -> `E_MUX_2.EI1` (`K = 0`) and `IN2.INIT_REQ` -> `E_MUX_2.EI2` (`K = 1`)
- `E_MUX_2.EO` -> `OR_QI.REQ`
- `OR_QI.CNF` -> `OUT.INIT_REQ`
- `OUT.INIT_CNF` -> `E_DEMUX_2.EI`
- `E_DEMUX_2.EO1` -> `IN1.INIT_CNF` and `E_DEMUX_2.EO2` -> `IN2.INIT_CNF`

Data:

- IN1.QI_OUT -> OR_QI.IN1, IN2.QI_OUT -> OR_QI.IN2
- OR_QI.OUT -> OUT.QI_OUT
- E_MUX_2.K -> E_DEMUX_2.K

E_MUX_2 remembers in K which socket the request came from, and only then triggers the OR operation. Its result is passed to OUT as INIT_REQ with QI_OUT. E_DEMUX_2 routes the confirmation OUT.INIT_CNF back, based on K, to exactly the socket that made the request.

Technical Notes

- **Confirmation only after OUT:** an input is confirmed only once the group at OUT has finished. An upstream DEMUX or AOption_T_2 therefore sees the chain as done only when it actually is.
- **Order enforced by wiring:** because OR_QI is triggered via E_MUX_2.EO, K is always set before OUT.INIT_REQ is sent.
- **Only one outstanding request:** K holds exactly one sender. If a second request arrives before OUT.INIT_CNF has returned, the assignment is lost. Behind a DEMUX (break-before-make) this cannot happen; independent sources would need a basic FB with an ECC.
- **A DEMUX output only once:** if the same output must also feed a group of its own, an AOption_T_2 is inserted first.

Application Scenarios

- A shared option group that should be active for several options (K = 2 or K = 3) but not for K = 1 (DummyGroup).

Summary

AOption_OR_2 merges two AOption requests into one (OUT active when IN1 or IN2 is active) and returns the confirmation via E_MUX_2/E_DEMUX_2 to the requesting socket.

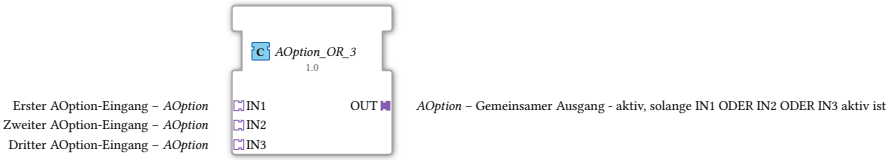
🛠️ Related Exercises

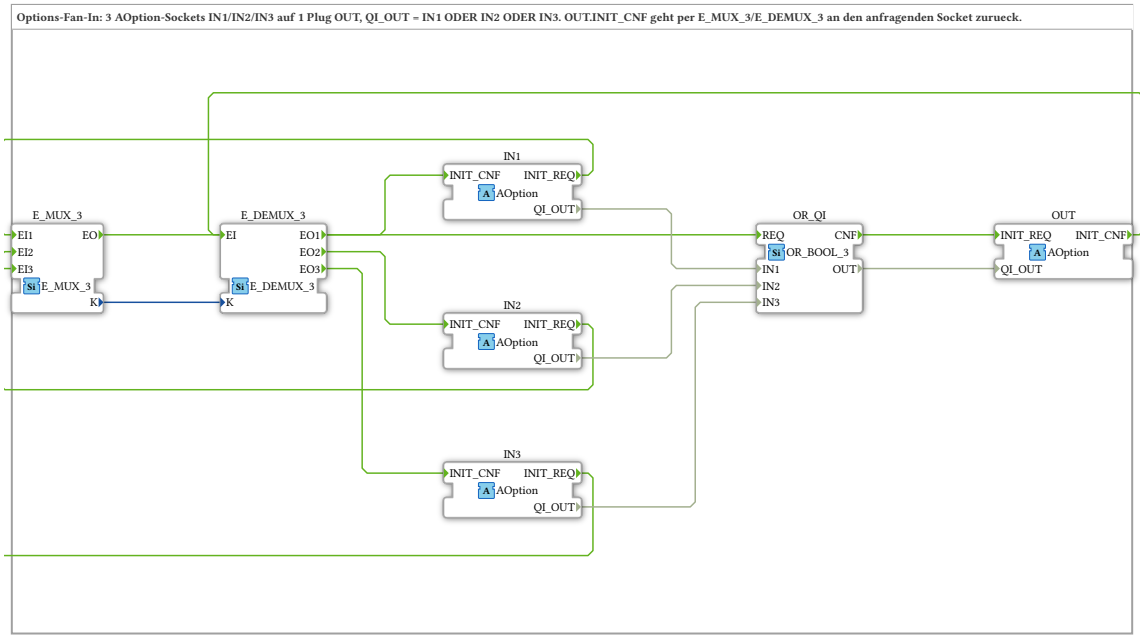
- Uebung_094b3e_AX
- Uebung_094b3f_AX

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AOption_OR_3





• – ○ • – ○ • – ○ •

Introduction

`AOption_OR_3` is a composite FB that serves as a **fan-in** (OR combination) for the option adapter `AOption`: three option requests at the sockets `IN1`, `IN2` and `IN3` are merged into a single request at the plug `OUT`. `OUT` is active as long as `IN1` **or** `IN2` **or** `IN3` is active. Variant of `AOption_OR_2` with 3 inputs.

Interface Structure

Event Inputs

None (the events run through the adapters).

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
`IN1`	`Option::AOption`	Socket	First AOption input.
`IN2`	`Option::AOption`	Socket	Second AOption input.
`IN3`	`Option::AOption`	Socket	Third AOption input.
`OUT`	`Option::AOption`	Plug	Common output - active as long as `IN1` OR `IN2` is active.

Functionality

The internal network contains three FBs: `E_MUX_3` (`iec61499::events::E_MUX_3`), `OR_QI` (`iec61131::booleanOperators::OR_BOOL_3`) and `E_DEMUX_3` (`iec61499::events::E_DEMUX_3`).

Events:

- `IN1.INIT_REQ` -> `E_MUX_3.EI1` (`K = 0`), `IN2.INIT_REQ` -> `E_MUX_3.EI2` (`K = 1`) and `IN3.INIT_REQ` -> `E_MUX_3.EI3` (`K = 2`)

- E_MUX_3.EO -> OR_QI.REQ
- OR_QI.CNF -> OUT.INIT_REQ
- OUT.INIT_CNF -> E_DEMUX_3.EI
- E_DEMUX_3.EO1 -> IN1.INIT_CNF, E_DEMUX_3.EO2 -> IN2.INIT_CNF and E_DEMUX_3.EO3 -> IN3.INIT_CNF

Data:

- IN1.QI_OUT -> OR_QI.IN1, IN2.QI_OUT -> OR_QI.IN2, IN3.QI_OUT -> OR_QI.IN3
- OR_QI.OUT -> OUT.QI_OUT
- E_MUX_3.K -> E_DEMUX_3.K

E_MUX_3 remembers in K which socket the request came from, and only then triggers the OR operation. Its result is passed to OUT as INIT_REQ with QI_OUT. E_DEMUX_3 routes the confirmation OUT.INIT_CNF back, based on K, to exactly the socket that made the request.

Technical Notes

- **Confirmation only after OUT:** an input is confirmed only once the group at OUT has finished. An upstream DEMUX or AOption_T_2 therefore sees the chain as done only when it actually is.
- **Order enforced by wiring:** because OR_QI is triggered via E_MUX_3.EO, K is always set before OUT.INIT_REQ is sent.
- **Only one outstanding request:** K holds exactly one sender. If a second request arrives before OUT.INIT_CNF has returned, the assignment is lost. Behind a DEMUX (break-before-make) this cannot happen; independent sources would need a basic FB with an ECC.
- **A DEMUX output only once:** if the same output must also feed a group of its own, an AOption_T_2 is inserted first.

Application Scenarios

- A shared option group that should be active for several options without cascading AOption_OR_2.

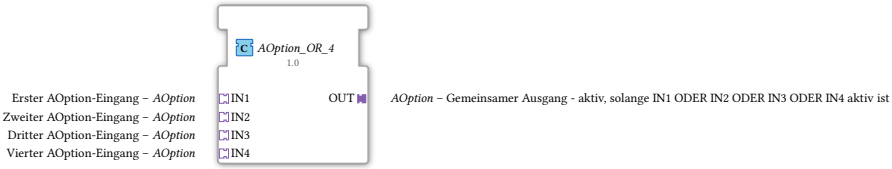
Summary

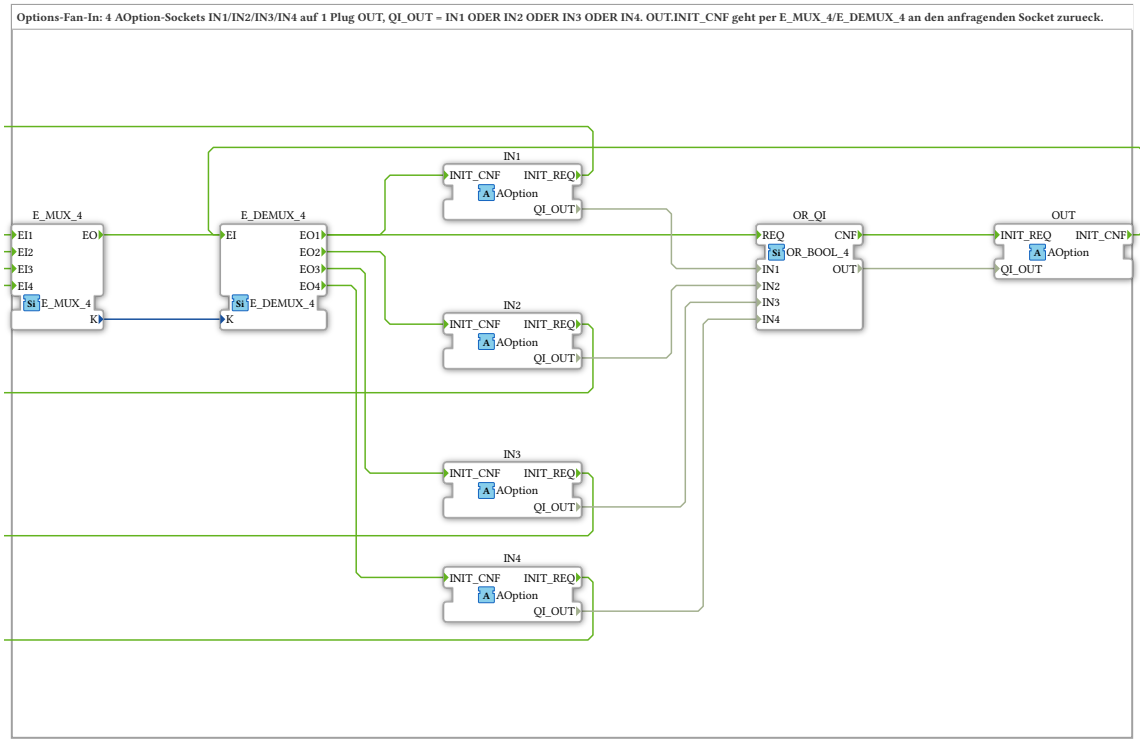
AOption_OR_3 merges three AOption requests into one (OUT active when IN1, IN2 or IN3 is active) and returns the confirmation via E_MUX_3/E_DEMUX_3 to the requesting socket.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AOption_OR_4





• — ○ • — ○ • — ○ •

Introduction

`AOption_OR_4` is a composite FB that serves as a **fan-in** (OR combination) for the option adapter `AOption`: four option requests at the sockets `IN1`, `IN2`, `IN3` and `IN4` are merged into a single request at the plug `OUT`. `OUT` is active as long as `IN1` **or** `IN2` **or** `IN3` **or** `IN4` is active. Variant of `AOption_OR_2` with 4 inputs.

Interface Structure

Event Inputs

None (the events run through the adapters).

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
`IN1`	`Option::AOption`	Socket	First AOption input.
`IN2`	`Option::AOption`	Socket	Second AOption input.
`IN3`	`Option::AOption`	Socket	Third AOption input.
`IN4`	`Option::AOption`	Socket	Fourth AOption input.
`OUT`	`Option::AOption`	Plug	Common output - active as long as `IN1` OR `IN2` is active.

Functionality

The internal network contains three FBs: `E_MUX_4` (`iec61499::events::E_MUX_4`), `OR_QI` (`iec61131::booleanOperators::OR_BOOL_4`) and `E_DEMUX_4` (`iec61499::events::E_DEMUX_4`).

Events:

- `IN1.INIT_REQ` -> `E_MUX_4.EI1` (`K = 0`), `IN2.INIT_REQ` -> `E_MUX_4.EI2` (`K = 1`), `IN3.INIT_REQ` -> `E_MUX_4.EI3` (`K = 2`) and `IN4.INIT_REQ` -> `E_MUX_4.EI4` (`K = 3`)
- `E_MUX_4.EO` -> `OR_QI.REQ`
- `OR_QI.CNF` -> `OUT.INIT_REQ`
- `OUT.INIT_CNF` -> `E_DEMUX_4.EI`
- `E_DEMUX_4.EO1` -> `IN1.INIT_CNF`, `E_DEMUX_4.EO2` -> `IN2.INIT_CNF`, `E_DEMUX_4.EO3` -> `IN3.INIT_CNF` and `E_DEMUX_4.EO4` -> `IN4.INIT_CNF`

Data:

- `IN1.QI_OUT` -> `OR_QI.IN1`, `IN2.QI_OUT` -> `OR_QI.IN2`, `IN3.QI_OUT` -> `OR_QI.IN3`, `IN4.QI_OUT` -> `OR_QI.IN4`
- `OR_QI.OUT` -> `OUT.QI_OUT`
- `E_MUX_4.K` -> `E_DEMUX_4.K`

`E_MUX_4` remembers in `K` which socket the request came from, and only then triggers the OR operation. Its result is passed to `OUT` as `INIT_REQ` with `QI_OUT`. `E_DEMUX_4` routes the confirmation `OUT.INIT_CNF` back, based on `K`, to exactly the socket that made the request.

Technical Notes

- **Confirmation only after OUT:** an input is confirmed only once the group at `OUT` has finished. An upstream DEMUX or `AOption_T_2` therefore sees the chain as done only when it actually is.
- **Order enforced by wiring:** because `OR_QI` is triggered via `E_MUX_4.EO`, `K` is always set before `OUT.INIT_REQ` is sent.
- **Only one outstanding request:** `K` holds exactly one sender. If a second request arrives before `OUT.INIT_CNF` has returned, the assignment is lost. Behind a DEMUX (break-before-make) this cannot happen; independent sources would need a basic FB with an ECC.
- **A DEMUX output only once:** if the same output must also feed a group of its own, an `AOption_T_2` is inserted first.

Application Scenarios

- A shared option group that should be active for several options without cascading `AOption_OR_2`.

Summary

`AOption_OR_4` merges four `AOption` requests into one (`OUT` active when `IN1`, `IN2`, `IN3` or `IN4` is active) and returns the confirmation via `E_MUX_4`/`E_DEMUX_4` to the requesting socket.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.1.9.4 selection

AOption_AD_DEMUX_1



• – ○ • – ○ • – ○ •

Introduction

`AOption_AD_DEMUX_1` is a bitmask demultiplexer (native generic FB `GEN_AOption_AD_DEMUX`, no ECC and no algorithm in this type) for `N=1` option outputs of type `Option::AOption`. Via the bidirectional bitmask adapter `K` (`adapter::types::bidirectional::AD2`, DWORD) any number of the 1 outputs can be activated at the same time (multi-select).

Interface Structure

Event Inputs / Outputs

No direct event connections; the events run through the adapters.

Data Inputs / Outputs

No direct data connections; the data runs through the adapters.

Adapters

Adapter	Type	Direction	Comment
`OUT1`	`Option::AOption`	Plug	Option 1, active when bit 0 of `K` is set (value 1).
`K`	`adapter::types::bidirectional::AD2`	Socket	Bitmask (Bit0=OUT1, Bit1=OUT2, ...), bidirectional: `EO1`/`DO1` = new bitmask request in, `EI1`/`DI1` = acknowledgement with the accepted bitmask back.

Functionality

Bit assignment (multi-select):

- `Bit 0` (value 1): `OUT1`

Example: `K = 1` activates `OUT1`, `K = 0` deactivates it. Any number of outputs, or all of them, can be active at the same time.

Switching behaviour: Only bits that change compared to the previous `K` value trigger an `INIT_REQ` at the corresponding output (0->1: `QI=TRUE`, activate; 1->0: `QI=FALSE`, deactivate). Unchanged bits are not triggered again; each output is triggered at most once per incoming `K` value.

Busy gating via the K adapter: While `INIT_CNF` is still awaited from at least one output affected by the last `K` change, every further bitmask request arriving via `K` (plug -> socket, `EO1`/`DO1`) is ignored; it is neither buffered nor caught up later. Once all triggered `INIT_CNF` have arrived, the block reports this backwards via `K` (socket -> plug, `EI1`/`DI1`) with the accepted bitmask (not with the state of the outputs; bits beyond the existing outputs are acknowledged unchanged as well). Only then are new `K` requests accepted again.

Technical Features

- **Multi-select**, in contrast to the single-select of `AOption_AUI_DEMUX_1`.
- **Native generic FB**: the behaviour is implemented in the runtime, not as ECC or algorithm.
- **Acknowledgement**: `K` is only reported back after all `INIT_CNF` have arrived; requests in between are lost.

Application Scenarios

- Switching several optional function groups (e.g. `OptionalGroup_001`) on and off simultaneously via a bitmask.

Conclusion

`AOption_AD_DEMUX_1` switches arbitrary combinations of 1 options via the bitmask `K` and reports the accepted bitmask only after all activations have completed.

AOption_AD_DEMUX_2



• – ○ • – ○ • – ○ •

Introduction

`AOption_AD_DEMUX_2` is a bitmask demultiplexer (native generic FB `GEN_AOption_AD_DEMUX`, no ECC and no algorithm in this type) for `N=2` option outputs of type `Option::AOption`. Via the bidirectional

bitmask adapter K (adapter::types::bidirectional::AD2, DWORD) any number of the 2 outputs can be activated at the same time (multi-select).

Interface Structure

Event Inputs / Outputs

No direct event connections; the events run through the adapters.

Data Inputs / Outputs

No direct data connections; the data runs through the adapters.

Adapters

Adapter	Type	Direction	Comment
OUT1	Option::AOption	Plug	Option 1, active when bit 0 of K is set (value 1).
OUT2	Option::AOption	Plug	Option 2, active when bit 1 of K is set (value 2).
K	adapter::types::bidirectional::AD2	Socket	Bitmask (Bit0=OUT1, Bit1=OUT2, ...), bidirectional: EO1/DO1 = new bitmask request in, EI1/DI1 = acknowledgement with the accepted bitmask back.

Functionality

Bit assignment (multi-select):

- Bit 0 (value 1): OUT1
- Bit 1 (value 2): OUT2

Example: K = 3 (bit 0 and bit 1 set) activates OUT1 and OUT2; all other outputs stay off. Any number of outputs, or all of them, can be active at the same time.

Switching behaviour: Only bits that change compared to the previous K value trigger an INIT_REQ at the corresponding output (0->1: QI=TRUE, activate; 1->0: QI=FALSE, deactivate). Unchanged bits are not triggered again; each output is triggered at most once per incoming K value.

Busy gating via the K adapter: While INIT_CNF is still awaited from at least one output affected by the last K change, every further bitmask request arriving via K (plug -> socket, EO1/DO1) is ignored; it is neither buffered nor caught up later. Once all triggered INIT_CNF have arrived, the block reports this backwards via K (socket -> plug, EI1/DI1) with the accepted bitmask (not with the state of the outputs; bits beyond the existing outputs are acknowledged unchanged as well). Only then are new K requests accepted again.

Technical Features

- **Multi-select**, in contrast to the single-select of AOption_AUI_DEMUX_2.
- **Native generic FB**: the behaviour is implemented in the runtime, not as ECC or algorithm.
- **Acknowledgement**: K is only reported back after all INIT_CNF have arrived; requests in between are lost.

Application Scenarios

- Switching several optional function groups (e.g. OptionalGroup_001) on and off simultaneously via a bitmask.

Conclusion

AOption_AD_DEMUX_2 switches arbitrary combinations of 2 options via the bitmask K and reports the accepted bitmask only after all activations have completed.

🛠️ Related Exercises

- Uebung_094b5_AX
- Uebung_094b5b_AX

AOption_AD_DEMUX_3



• — ○ • — ○ • — ○ •

Introduction

`AOption_AD_DEMUX_3` is a bitmask demultiplexer (native generic FB `GEN_AOption_AD_DEMUX`, no ECC and no algorithm in this type) for `N=3` option outputs of type `Option::AOption`. Via the bidirectional bitmask adapter `K` (`adapter::types::bidirectional::AD2`, DWORD) any number of the 3 outputs can be activated at the same time (multi-select).

Interface Structure

Event Inputs / Outputs

No direct event connections; the events run through the adapters.

Data Inputs / Outputs

No direct data connections; the data runs through the adapters.

Adapters

Adapter	Type	Direction	Comment
`OUT1`	`Option::AOption`	Plug	Option 1, active when bit 0 of `K` is set (value 1).
`OUT2`	`Option::AOption`	Plug	Option 2, active when bit 1 of `K` is set (value 2).
`OUT3`	`Option::AOption`	Plug	Option 3, active when bit 2 of `K` is set (value 4).
`K`	`adapter::types::bidirectional::AD2`	Socket	Bitmask (Bit0=OUT1, Bit1=OUT2, ...), bidirectional: `EO1/DO1` = new bitmask request in, `EI1/DI1` = acknowledgement with the accepted bitmask back.

Functionality

Bit assignment (multi-select):

- `Bit 0` (value 1): `OUT1`
- `Bit 1` (value 2): `OUT2`
- `Bit 2` (value 4): `OUT3`

Example: `K = 3` (bit 0 and bit 1 set) activates `OUT1` and `OUT2`; all other outputs stay off. Any number of outputs, or all of them, can be active at the same time.

Switching behaviour: Only bits that change compared to the previous `K` value trigger an `INIT_REQ` at the corresponding output (0->1: `QI=TRUE`, activate; 1->0: `QI=FALSE`, deactivate). Unchanged bits are not triggered again; each output is triggered at most once per incoming `K` value.

Busy gating via the K adapter: While `INIT_CNF` is still awaited from at least one output affected by the last `K` change, every further bitmask request arriving via `K` (plug -> socket, `EO1/DO1`) is ignored; it is neither buffered nor caught up later. Once all triggered `INIT_CNF` have arrived, the block reports this backwards via `K` (socket -> plug, `EI1/DI1`) with the accepted bitmask (not with the state of the outputs; bits beyond the existing outputs are acknowledged unchanged as well). Only then are new `K` requests accepted again.

Technical Features

- **Multi-select**, in contrast to the single-select of `AOption_AUI_DEMUX_3`.
- **Native generic FB**: the behaviour is implemented in the runtime, not as ECC or algorithm.

- **Acknowledgement**: `K` is only reported back after all `INIT_CNF` have arrived; requests in between are lost.

Application Scenarios

- Switching several optional function groups (e.g. `OptionalGroup_001`) on and off simultaneously via a bitmask.

Conclusion

`AOption_AD_DEMUX_3` switches arbitrary combinations of 3 options via the bitmask `K` and reports the accepted bitmask only after all activations have completed.

AOption_AD_DEMUX_4



• – ○ • – ○ • – ○ •

Introduction

`AOption_AD_DEMUX_4` is a bitmask demultiplexer (native generic FB `GEN_AOption_AD_DEMUX`, no ECC and no algorithm in this type) for `N=4` option outputs of type `Option::AOption`. Via the bidirectional bitmask adapter `K` (`adapter::types::bidirectional::AD2`, DWORD) any number of the 4 outputs can be activated at the same time (multi-select).

Interface Structure

Event Inputs / Outputs

No direct event connections; the events run through the adapters.

Data Inputs / Outputs

No direct data connections; the data runs through the adapters.

Adapters

Adapter	Type	Direction	Comment
`OUT1`	`Option::AOption`	Plug	Option 1, active when bit 0 of `K` is set (value 1).
`OUT2`	`Option::AOption`	Plug	Option 2, active when bit 1 of `K` is set (value 2).
`OUT3`	`Option::AOption`	Plug	Option 3, active when bit 2 of `K` is set (value 4).
`OUT4`	`Option::AOption`	Plug	Option 4, active when bit 3 of `K` is set (value 8).
`K`	`adapter::types::bidirectional::AD2`	Socket	Bitmask (Bit0=OUT1, Bit1=OUT2, ...), bidirectional: `EO1/DO1` = new bitmask request in, `EI1/DI1` = acknowledgement with the accepted bitmask back.

Functionality

Bit assignment (multi-select):

- `Bit 0` (value 1): `OUT1`
- `Bit 1` (value 2): `OUT2`
- `Bit 2` (value 4): `OUT3`
- `Bit 3` (value 8): `OUT4`

Example: `K = 3` (bit 0 and bit 1 set) activates `OUT1` and `OUT2`; all other outputs stay off. Any number of outputs, or all of them, can be active at the same time.

Switching behaviour: Only bits that change compared to the previous `K` value trigger an `INIT_REQ` at the corresponding output (0->1: `QI=TRUE`, activate; 1->0: `QI=FALSE`, deactivate). Unchanged bits are not triggered again; each output is triggered at most once per incoming `K` value.

Busy gating via the K adapter: While `INIT_CNF` is still awaited from at least one output affected by the last `K` change, every further bitmask request arriving via `K` (plug -> socket, `EO1`/`DO1`) is ignored; it is neither buffered nor caught up later. Once all triggered `INIT_CNF` have arrived, the block reports this backwards via `K` (socket -> plug, `EI1`/`DI1`) with the accepted bitmask (not with the state of the outputs; bits beyond the existing outputs are acknowledged unchanged as well). Only then are new `K` requests accepted again.

Technical Features

- **Multi-select**, in contrast to the single-select of `AOption_AUI_DEMUX_4`.
- **Native generic FB**: the behaviour is implemented in the runtime, not as ECC or algorithm.
- **Acknowledgement**: `K` is only reported back after all `INIT_CNF` have arrived; requests in between are lost.

Application Scenarios

- Switching several optional function groups (e.g. `OptionalGroup_001`) on and off simultaneously via a bitmask.

Conclusion

`AOption_AD_DEMUX_4` switches arbitrary combinations of 4 options via the bitmask `K` and reports the accepted bitmask only after all activations have completed.

AOption_AD_DEMUX_5



• – ○ • – ○ • – ○ •

Introduction

`AOption_AD_DEMUX_5` is a bitmask demultiplexer (native generic FB `GEN_AOption_AD_DEMUX`, no ECC and no algorithm in this type) for `N=5` option outputs of type `Option::AOption`. Via the bidirectional bitmask adapter `K` (`adapter::types::bidirectional::AD2`, DWORD) any number of the 5 outputs can be activated at the same time (multi-select).

Interface Structure

Event Inputs / Outputs

No direct event connections; the events run through the adapters.

Data Inputs / Outputs

No direct data connections; the data runs through the adapters.

Adapters

Adapter	Type	Direction	Comment
`OUT1`	`Option::AOption`	Plug	Option 1, active when bit 0 of `K` is set (value 1).
`OUT2`	`Option::AOption`	Plug	Option 2, active when bit 1 of `K` is set (value 2).
`OUT3`	`Option::AOption`	Plug	Option 3, active when bit 2 of `K` is set (value 4).

OUT4	Option::AOption	Plug	Option 4, active when bit 3 of K is set (value 8).
OUT5	Option::AOption	Plug	Option 5, active when bit 4 of K is set (value 16).
K	adapter::types::bidirectional::AD2	Socket	Bitmask (Bit0=OUT1, Bit1=OUT2, ...), bidirectional: EO1/DO1 = new bitmask request in, EI1/DI1 = acknowledgement with the accepted bitmask back.

Functionality

Bit assignment (multi-select):

- Bit 0 (value 1): OUT1
- Bit 1 (value 2): OUT2
- Bit 2 (value 4): OUT3
- Bit 3 (value 8): OUT4
- Bit 4 (value 16): OUT5

Example: K = 3 (bit 0 and bit 1 set) activates OUT1 and OUT2; all other outputs stay off. Any number of outputs, or all of them, can be active at the same time.

Switching behaviour: Only bits that change compared to the previous K value trigger an INIT_REQ at the corresponding output (0->1: QI=TRUE, activate; 1->0: QI=FALSE, deactivate). Unchanged bits are not triggered again; each output is triggered at most once per incoming K value.

Busy gating via the K adapter: While INIT_CNF is still awaited from at least one output affected by the last K change, every further bitmask request arriving via K (plug -> socket, EO1/DO1) is ignored; it is neither buffered nor caught up later. Once all triggered INIT_CNF have arrived, the block reports this backwards via K (socket -> plug, EI1/DI1) with the accepted bitmask (not with the state of the outputs; bits beyond the existing outputs are acknowledged unchanged as well). Only then are new K requests accepted again.

Technical Features

- **Multi-select**, in contrast to the single-select of AOption_AUI_DEMUX_5.
- **Native generic FB**: the behaviour is implemented in the runtime, not as ECC or algorithm.
- **Acknowledgement**: K is only reported back after all INIT_CNF have arrived; requests in between are lost.

Application Scenarios

- Switching several optional function groups (e.g. OptionalGroup_001) on and off simultaneously via a bitmask.

Conclusion

AOption_AD_DEMUX_5 switches arbitrary combinations of 5 options via the bitmask K and reports the accepted bitmask only after all activations have completed.

AOption_AD_DEMUX_6



• – ○ • – ○ • – ○ •

Introduction

AOption_AD_DEMUX_6 is a bitmask demultiplexer (native generic FB GEN_AOption_AD_DEMUX, no ECC and no algorithm in this type) for N=6 option outputs of type Option::AOption. Via the bidirectional bitmask adapter K (adapter::types::bidirectional::AD2, DWORD) any number of the 6 outputs can be activated at the same time (multi-select).

Interface Structure

Event Inputs / Outputs

No direct event connections; the events run through the adapters.

Data Inputs / Outputs

No direct data connections; the data runs through the adapters.

Adapters

Adapter	Type	Direction	Comment
OUT1	Option::AOption	Plug	Option 1, active when bit 0 of K is set (value 1).
OUT2	Option::AOption	Plug	Option 2, active when bit 1 of K is set (value 2).
OUT3	Option::AOption	Plug	Option 3, active when bit 2 of K is set (value 4).
OUT4	Option::AOption	Plug	Option 4, active when bit 3 of K is set (value 8).
OUT5	Option::AOption	Plug	Option 5, active when bit 4 of K is set (value 16).
OUT6	Option::AOption	Plug	Option 6, active when bit 5 of K is set (value 32).
K	adapter::types::bidirectional::AD2	Socket	Bitmask (Bit0=OUT1, Bit1=OUT2, ...), bidirectional: EO1/DO1 = new bitmask request in, EI1/DI1 = acknowledgement with the accepted bitmask back.

Functionality

Bit assignment (multi-select):

- Bit 0 (value 1): OUT1
- Bit 1 (value 2): OUT2
- Bit 2 (value 4): OUT3
- Bit 3 (value 8): OUT4
- Bit 4 (value 16): OUT5
- Bit 5 (value 32): OUT6

Example: K = 3 (bit 0 and bit 1 set) activates OUT1 and OUT2; all other outputs stay off. Any number of outputs, or all of them, can be active at the same time.

Switching behaviour: Only bits that change compared to the previous K value trigger an INIT_REQ at the corresponding output (0->1: QI=TRUE, activate; 1->0: QI=FALSE, deactivate). Unchanged bits are not triggered again; each output is triggered at most once per incoming K value.

Busy gating via the K adapter: While INIT_CNF is still awaited from at least one output affected by the last K change, every further bitmask request arriving via K (plug -> socket, EO1/DO1) is ignored; it is neither buffered nor caught up later. Once all triggered INIT_CNF have arrived, the block reports this backwards via K (socket -> plug, EI1/DI1) with the accepted bitmask (not with the state of the outputs; bits beyond the existing outputs are acknowledged unchanged as well). Only then are new K requests accepted again.

Technical Features

- **Multi-select**, in contrast to the single-select of AOption_AUI_DEMUX_6.
- **Native generic FB**: the behaviour is implemented in the runtime, not as ECC or algorithm.
- **Acknowledgement**: K is only reported back after all INIT_CNF have arrived; requests in between are lost.

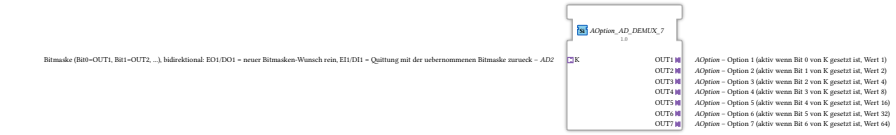
Application Scenarios

- Switching several optional function groups (e.g. `OptionalGroup_001`) on and off simultaneously via a bitmask.

Conclusion

`AOption_AD_DEMUX_6` switches arbitrary combinations of 6 options via the bitmask `K` and reports the accepted bitmask only after all activations have completed.

AOption_AD_DEMUX_7



• – ○ • – ○ • – ○ •

Introduction

The function block AOption_AD_DEMUX_7 is a demultiplexer (native generic FB, no ECC and no algorithm in this type) for 7 option outputs of type `Option::AOption`. Selection is done by a **bit mask** on the bidirectional adapter `K` (`adapter::types::bidirectional::AD2`, DWORD). This is the multi-select variant: any number of outputs can be active at the same time.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **K** (Socket, `adapter::types::bidirectional::AD2`): bit mask (bit 0 = OUT1, bit 1 = OUT2, ...). Bidirectional: `EO1/DO1` = new bit mask request comes in, `EI1/DI1` = acknowledgement with the accepted bit mask goes back
- **OUT1** (Plug, `Option::AOption`): option 1, active when bit 0 of K is set (value 1)
- **OUT2** (Plug, `Option::AOption`): option 2, active when bit 1 of K is set (value 2)
- **OUT3** (Plug, `Option::AOption`): option 3, active when bit 2 of K is set (value 4)
- **OUT4** (Plug, `Option::AOption`): option 4, active when bit 3 of K is set (value 8)
- **OUT5** (Plug, `Option::AOption`): option 5, active when bit 4 of K is set (value 16)
- **OUT6** (Plug, `Option::AOption`): option 6, active when bit 5 of K is set (value 32)
- **OUT7** (Plug, `Option::AOption`): option 7, active when bit 6 of K is set (value 64)

Each output `OUTx` uses the adapter `Option::AOption`: the event `INIT_REQ` (together with `QI_OUT`) runs in the plug → socket direction, the confirmation `INIT_CNF` in the opposite direction.

Functionality

Bit assignment (multi-select):

- Bit 0 (value 1) — `OUT1`
- Bit 1 (value 2) — `OUT2`
- Bit 2 (value 4) — `OUT3`
- Bit 3 (value 8) — `OUT4`
- Bit 4 (value 16) — `OUT5`
- Bit 5 (value 32) — `OUT6`

- Bit 6 (value 64) — `OUT7`

Example: `K = 3` (bit 0 and bit 1 set) activates `OUT1` and `OUT2`, all other outputs are off. Any number of outputs, or all of them, can be active simultaneously.

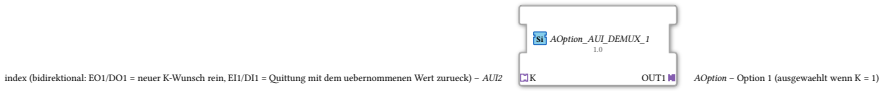
Switching behavior: Only bits that change compared to the previous `K` value trigger an `INIT_REQ` on the corresponding output (0⟶1: `QI=TRUE`, activate; 1⟶0: `QI=FALSE`, deactivate). Unchanged bits are not triggered again — each output is triggered at most once per incoming `K` value.

Busy gating via the bidirectional K adapter: As long as `INIT_CNF` is still awaited from at least one output affected by the last K change, every further bit mask request arriving via `K` (plug ⟶ socket direction, `EO1`/`DO1`) is ignored — it is neither buffered nor replayed. Once all triggered `INIT_CNF` have arrived, the block reports this back via `K` (socket ⟶ plug direction, `EI1`/`DI1`) with the accepted bit mask (not with the state of the outputs: bits beyond the available outputs are acknowledged unchanged as well). Only then are new K requests accepted again.

⚖️ Comparison with Similar Blocks

AOption_AUI_DEMUX_7 is the single-select variant with an index (`AUI2`) instead of a bit mask: at most one output is active there.

AOption_AUI_DEMUX_1



• – ○ • – ○ • – ○ •

Introduction

The function block AOption_AUI_DEMUX_1 is a demultiplexer (native generic FB, no ECC and no algorithm in this type) for 1 option output of type `Option::AOption`. Selection is done by an **index** on the bidirectional adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). This is the single-select variant: at most one output is active at a time.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **K** (Socket, `adapter::types::bidirectional::AUI2`): index. Bidirectional: `EO1`/`DO1` = new K request comes in, `EI1`/`DI1` = acknowledgement with the accepted value goes back
- **OUT1** (Plug, `Option::AOption`): option 1, selected when K = 1

Each output `OUTx` uses the adapter `Option::AOption`: the event `INIT_REQ` (together with `QI_OUT`) runs in the plug ⟶ socket direction, the confirmation `INIT_CNF` in the opposite direction.

Functionality

Index assignment:

- `K = 0` — all outputs OFF
- `K = 1` — `OUT1` ON, all others OFF
- any other value (> 1) — all outputs OFF

Switching behavior (break-before-make): If K changes from one active output to another, the previously active output is deactivated first (INIT_REQ with QI=FALSE). Only once INIT_CNF has arrived from that output is the newly selected output activated (INIT_REQ with QI=TRUE). Thus never more than one option is active at the same time. Each output is triggered at most once per incoming K value; unaffected outputs are left untouched.

Busy gating via the bidirectional K adapter: While a switchover is in progress (waiting for INIT_CNF from the previous and/or new output), every further index request arriving via K (plug → socket direction, EO1/DO1) is ignored — it is neither buffered nor replayed. When finished, the block reports this back via K (socket → plug direction, EI1/DI1) with the accepted K value (not with the state of the outputs: an invalid value is acknowledged unchanged as well). Only then are new index requests accepted again. What counts is always the state after completion of the last change actually processed.

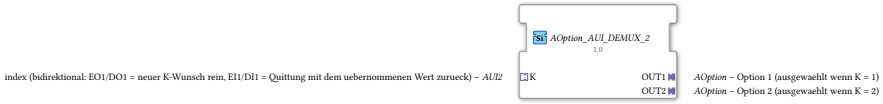
⚖️ Comparison with Similar Blocks

AOption_AD_DEMUX_1 is the multi-select variant with a bit mask (AD2): several outputs can be active at the same time there.

🛠️ Related Exercises

- Uebung_094c2_AX
- Uebung_094c2b_AX
- Uebung_094c_AX

AOption_AUI_DEMUX_2



• — ○ • — ○ • — ○ •

Introduction

The function block AOption_AUI_DEMUX_2 is a demultiplexer (native generic FB, no ECC and no algorithm in this type) for 2 option outputs of type Option::AOption. Selection is done by an **index** on the bidirectional adapter K (adapter::types::bidirectional::AUI2, UINT). This is the single-select variant: at most one output is active at a time.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **K** (Socket, adapter::types::bidirectional::AUI2): index. Bidirectional: EO1/DO1 = new K request comes in, EI1/DI1 = acknowledgement with the accepted value goes back
- **OUT1** (Plug, Option::AOption): option 1, selected when K = 1
- **OUT2** (Plug, Option::AOption): option 2, selected when K = 2

Each output OUTx uses the adapter Option::AOption: the event INIT_REQ (together with QI_OUT) runs in the plug → socket direction, the confirmation INIT_CNF in the opposite direction.

Functionality

Index assignment:

- `K = 0` — all outputs OFF
- `K = 1` — `OUT1` ON, all others OFF
- `K = 2` — `OUT2` ON, all others OFF
- any other value (> 2) — all outputs OFF

Switching behavior (break-before-make): If `K` changes from one active output to another, the previously active output is deactivated first (`INIT_REQ` with `QI=FALSE`). Only once `INIT_CNF` has arrived from that output is the newly selected output activated (`INIT_REQ` with `QI=TRUE`). Thus never more than one option is active at the same time. Each output is triggered at most once per incoming `K` value; unaffected outputs are left untouched.

Busy gating via the bidirectional K adapter: While a switchover is in progress (waiting for `INIT_CNF` from the previous and/or new output), every further index request arriving via `K` (plug → socket direction, `EO1`/`DO1`) is ignored — it is neither buffered nor replayed. When finished, the block reports this back via `K` (socket → plug direction, `EI1`/`DI1`) with the accepted K value (not with the state of the outputs: an invalid value is acknowledged unchanged as well). Only then are new index requests accepted again. What counts is always the state after completion of the last change actually processed.

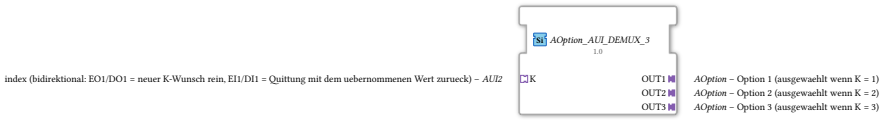
⚖️ Comparison with Similar Blocks

AOption_AD_DEMUX_2 is the multi-select variant with a bit mask (`AD2`): several outputs can be active at the same time there.

🛠️ Related Exercises

- Uebung_094b4_AX
- Uebung_094b4b_AX
- Uebung_094b_AX

AOption_AUI_DEMUX_3



• — ○ • — ○ • — ○ •

Introduction

The function block AOption_AUI_DEMUX_3 is a demultiplexer (native generic FB, no ECC and no algorithm in this type) for 3 option outputs of type `Option::AOption`. Selection is done by an **index** on the bidirectional adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). This is the single-select variant: at most one output is active at a time.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **K** (Socket, `adapter::types::bidirectional::AUI2`): index. Bidirectional: `EO1`/`DO1` = new K request comes in, `EI1`/`DI1` = acknowledgement with the accepted value goes back
- **OUT1** (Plug, `Option::AOption`): option 1, selected when K = 1
- **OUT2** (Plug, `Option::AOption`): option 2, selected when K = 2
- **OUT3** (Plug, `Option::AOption`): option 3, selected when K = 3

Each output OUTx uses the adapter Option::AOption: the event INIT_REQ (together with QI_OUT) runs in the plug ⟶ socket direction, the confirmation INIT_CNF in the opposite direction.

Functionality

Index assignment:

- K = 0 — all outputs OFF
- K = 1 — OUT1 ON, all others OFF
- K = 2 — OUT2 ON, all others OFF
- K = 3 — OUT3 ON, all others OFF
- any other value (> 3) — all outputs OFF

Switching behavior (break-before-make): If K changes from one active output to another, the previously active output is deactivated first (INIT_REQ with QI=FALSE). Only once INIT_CNF has arrived from that output is the newly selected output activated (INIT_REQ with QI=TRUE). Thus never more than one option is active at the same time. Each output is triggered at most once per incoming K value; unaffected outputs are left untouched.

Busy gating via the bidirectional K adapter: While a switchover is in progress (waiting for INIT_CNF from the previous and/or new output), every further index request arriving via K (plug ⟶ socket direction, EO1/DO1) is ignored — it is neither buffered nor replayed. When finished, the block reports this back via K (socket ⟶ plug direction, EI1/DI1) with the accepted K value (not with the state of the outputs: an invalid value is acknowledged unchanged as well). Only then are new index requests accepted again. What counts is always the state after completion of the last change actually processed.

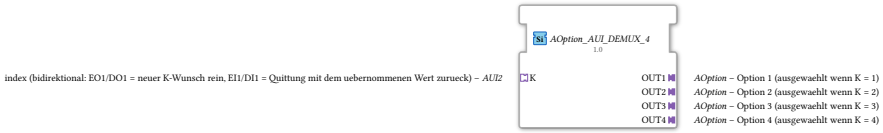
⚖️ Comparison with Similar Blocks

AOption_AD_DEMUX_3 is the multi-select variant with a bit mask (AD2): several outputs can be active at the same time there.

🛠️ Related Exercises

- Uebung_094b2_AX
- Uebung_094b3_AX
- Uebung_094b3b_AX

AOption_AUI_DEMUX_4



• — ○ • — ○ • — ○ •

Introduction

The function block AOption_AUI_DEMUX_4 is a demultiplexer (native generic FB, no ECC and no algorithm in this type) for 4 option outputs of type Option::AOption. Selection is done by an **index** on the bidirectional adapter K (adapter::types::bidirectional::AUI2, UINT). This is the single-select variant: at most one output is active at a time.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **K** (Socket, `adapter::types::bidirectional::AUI2`): index. Bidirectional: `EO1/DO1` = new K request comes in, `EI1/DI1` = acknowledgement with the accepted value goes back
- **OUT1** (Plug, `Option::AOption`): option 1, selected when K = 1
- **OUT2** (Plug, `Option::AOption`): option 2, selected when K = 2
- **OUT3** (Plug, `Option::AOption`): option 3, selected when K = 3
- **OUT4** (Plug, `Option::AOption`): option 4, selected when K = 4

Each output `OUTx` uses the adapter `Option::AOption`: the event `INIT_REQ` (together with `QI_OUT`) runs in the plug → socket direction, the confirmation `INIT_CNF` in the opposite direction.

Functionality

Index assignment:

- `K = 0` — all outputs OFF
- `K = 1` — `OUT1` ON, all others OFF
- `K = 2` — `OUT2` ON, all others OFF
- `K = 3` — `OUT3` ON, all others OFF
- `K = 4` — `OUT4` ON, all others OFF
- any other value (> 4) — all outputs OFF

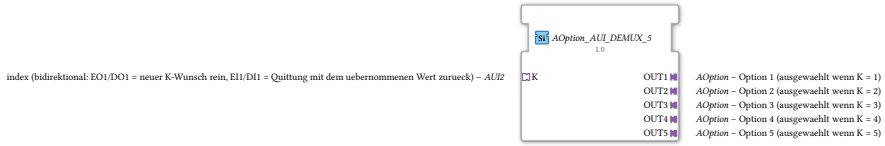
Switching behavior (break-before-make): If `K` changes from one active output to another, the previously active output is deactivated first (`INIT_REQ` with `QI=FALSE`). Only once `INIT_CNF` has arrived from that output is the newly selected output activated (`INIT_REQ` with `QI=TRUE`). Thus never more than one option is active at the same time. Each output is triggered at most once per incoming `K` value; unaffected outputs are left untouched.

Busy gating via the bidirectional K adapter: While a switchover is in progress (waiting for `INIT_CNF` from the previous and/or new output), every further index request arriving via `K` (plug → socket direction, `EO1/DO1`) is ignored — it is neither buffered nor replayed. When finished, the block reports this back via `K` (socket → plug direction, `EI1/DI1`) with the accepted K value (not with the state of the outputs: an invalid value is acknowledged unchanged as well). Only then are new index requests accepted again. What counts is always the state after completion of the last change actually processed.

⚖️ Comparison with Similar Blocks

AOption_AD_DEMUX_4 is the multi-select variant with a bit mask (`AD2`): several outputs can be active at the same time there.

AOption_AUI_DEMUX_5



• – ○ • – ○ • – ○ •

Introduction

The function block AOption_AUI_DEMUX_5 is a demultiplexer (native generic FB, no ECC and no algorithm in this type) for 5 option outputs of type `Option::AOption`. Selection is done by an **index** on the bidirectional adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). This is the single-select variant: at most one output is active at a time.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **K** (Socket, `adapter::types::bidirectional::AUI2`): index. Bidirectional: `EO1/DO1` = new K request comes in, `EI1/DI1` = acknowledgement with the accepted value goes back
- **OUT1** (Plug, `Option::AOption`): option 1, selected when K = 1
- **OUT2** (Plug, `Option::AOption`): option 2, selected when K = 2
- **OUT3** (Plug, `Option::AOption`): option 3, selected when K = 3
- **OUT4** (Plug, `Option::AOption`): option 4, selected when K = 4
- **OUT5** (Plug, `Option::AOption`): option 5, selected when K = 5

Each output `OUTx` uses the adapter `Option::AOption`: the event `INIT_REQ` (together with `QI_OUT`) runs in the plug ⟶ socket direction, the confirmation `INIT_CNF` in the opposite direction.

Functionality

Index assignment:

- `K = 0` — all outputs OFF
- `K = 1` — `OUT1` ON, all others OFF
- `K = 2` — `OUT2` ON, all others OFF
- `K = 3` — `OUT3` ON, all others OFF
- `K = 4` — `OUT4` ON, all others OFF
- `K = 5` — `OUT5` ON, all others OFF
- any other value (> 5) — all outputs OFF

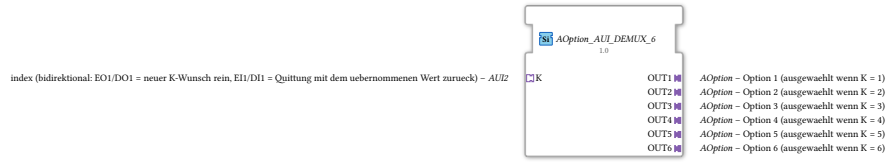
Switching behavior (break-before-make): If `K` changes from one active output to another, the previously active output is deactivated first (`INIT_REQ` with `QI=FALSE`). Only once `INIT_CNF` has arrived from that output is the newly selected output activated (`INIT_REQ` with `QI=TRUE`). Thus never more than one option is active at the same time. Each output is triggered at most once per incoming `K` value; unaffected outputs are left untouched.

Busy gating via the bidirectional K adapter: While a switchover is in progress (waiting for `INIT_CNF` from the previous and/or new output), every further index request arriving via `K` (plug ⟶ socket direction, `EO1/DO1`) is ignored — it is neither buffered nor replayed. When finished, the block reports this back via `K` (socket ⟶ plug direction, `EI1/DI1`) with the accepted K value (not with the state of the outputs: an invalid value is acknowledged unchanged as well). Only then are new index requests accepted again. What counts is always the state after completion of the last change actually processed.

⚖️ Comparison with Similar Blocks

AOption_AD_DEMUX_5 is the multi-select variant with a bit mask (`AD2`): several outputs can be active at the same time there.

AOption_AUI_DEMUX_6



• — ○ • — ○ • — ○ •

Introduction

The function block AOption_AUI_DEMUX_6 is a demultiplexer (native generic FB, no ECC and no algorithm in this type) for 6 option outputs of type `Option::AOption`. Selection is done by an **index** on the bidirectional adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). This is the single-select variant: at most one output is active at a time.

Interface Structure

Event Inputs
The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs
The function block has no direct event outputs.

Data Inputs
The function block has no direct data inputs.

Data Outputs
The function block has no direct data outputs.

Adapters
- **K** (Socket, `adapter::types::bidirectional::AUI2`): index. Bidirectional: `EO1`/`DO1` = new K request comes in, `EI1`/`DI1` = acknowledgement with the accepted value goes back
- **OUT1** (Plug, `Option::AOption`): option 1, selected when K = 1
- **OUT2** (Plug, `Option::AOption`): option 2, selected when K = 2
- **OUT3** (Plug, `Option::AOption`): option 3, selected when K = 3
- **OUT4** (Plug, `Option::AOption`): option 4, selected when K = 4
- **OUT5** (Plug, `Option::AOption`): option 5, selected when K = 5
- **OUT6** (Plug, `Option::AOption`): option 6, selected when K = 6

Each output `OUTx` uses the adapter `Option::AOption`: the event `INIT_REQ` (together with `QI_OUT`) runs in the plug → socket direction, the confirmation `INIT_CNF` in the opposite direction.

Functionality
Index assignment:

- `K = 0` — all outputs OFF
- `K = 1` — `OUT1` ON, all others OFF
- `K = 2` — `OUT2` ON, all others OFF
- `K = 3` — `OUT3` ON, all others OFF
- `K = 4` — `OUT4` ON, all others OFF
- `K = 5` — `OUT5` ON, all others OFF
- `K = 6` — `OUT6` ON, all others OFF
- any other value (> 6) — all outputs OFF

Switching behavior (break-before-make): If `K` changes from one active output to another, the previously active output is deactivated first (`INIT_REQ` with `QI=FALSE`). Only once `INIT_CNF` has arrived from that output is the newly selected output activated (`INIT_REQ` with `QI=TRUE`). Thus never more than one option is active at the same time. Each output is triggered at most once per incoming `K` value; unaffected outputs are left untouched.

Busy gating via the bidirectional K adapter: While a switchover is in progress (waiting for `INIT_CNF` from the previous and/or new output), every further index request arriving via `K` (plug → socket direction, `EO1`/`DO1`) is ignored — it is neither buffered nor replayed. When finished, the block reports this back via `K` (socket → plug direction, `EI1`/`DI1`) with the accepted K value (not with the state of the outputs: an invalid value is acknowledged unchanged as well). Only then are new index requests accepted again. What counts is always the state after completion of the last change actually processed.

⚖️ Comparison with Similar Blocks
AOption_AD_DEMUX_6 is the multi-select variant with a bit mask (`AD2`): several outputs can be active at the same time there.

AOption_AUI_DEMUX_7



• — ◦ • — ◦ • — ◦ •

Introduction

The function block AOption_AUI_DEMUX_7 is a demultiplexer (native generic FB, no ECC and no algorithm in this type) for 7 option outputs of type `Option::AOption`. Selection is done by an **index** on the bidirectional adapter `K` (`adapter::types::bidirectional::AUI2`, UINT). This is the single-select variant: at most one output is active at a time.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **K** (Socket, `adapter::types::bidirectional::AUI2`): index. Bidirectional: `EO1/DO1` = new K request comes in, `EI1/DI1` = acknowledgement with the accepted value goes back
- **OUT1** (Plug, `Option::AOption`): option 1, selected when K = 1
- **OUT2** (Plug, `Option::AOption`): option 2, selected when K = 2
- **OUT3** (Plug, `Option::AOption`): option 3, selected when K = 3
- **OUT4** (Plug, `Option::AOption`): option 4, selected when K = 4
- **OUT5** (Plug, `Option::AOption`): option 5, selected when K = 5
- **OUT6** (Plug, `Option::AOption`): option 6, selected when K = 6
- **OUT7** (Plug, `Option::AOption`): option 7, selected when K = 7

Each output `OUTx` uses the adapter `Option::AOption`: the event `INIT_REQ` (together with `QI_OUT`) runs in the plug ⟶ socket direction, the confirmation `INIT_CNF` in the opposite direction.

Functionality

Index assignment:

- `K = 0` — all outputs OFF
- `K = 1` — `OUT1` ON, all others OFF
- `K = 2` — `OUT2` ON, all others OFF
- `K = 3` — `OUT3` ON, all others OFF
- `K = 4` — `OUT4` ON, all others OFF
- `K = 5` — `OUT5` ON, all others OFF
- `K = 6` — `OUT6` ON, all others OFF
- `K = 7` — `OUT7` ON, all others OFF
- any other value (> 7) — all outputs OFF

Switching behavior (break-before-make): If `K` changes from one active output to another, the previously active output is deactivated first (`INIT_REQ` with `QI=FALSE`). Only once `INIT_CNF` has arrived from that output is the newly selected output activated (`INIT_REQ` with `QI=TRUE`). Thus never more than one

option is active at the same time. Each output is triggered at most once per incoming `K` value; unaffected outputs are left untouched.

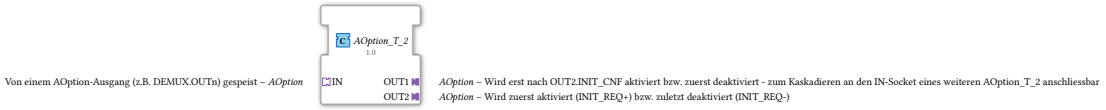
Busy gating via the bidirectional K adapter: While a switchover is in progress (waiting for `INIT_CNF` from the previous and/or new output), every further index request arriving via `K` (plug ⟶ socket direction, `EO1`/`DO1`) is ignored — it is neither buffered nor replayed. When finished, the block reports this back via `K` (socket ⟶ plug direction, `EI1`/`DI1`) with the accepted K value (not with the state of the outputs: an invalid value is acknowledged unchanged as well). Only then are new index requests accepted again. What counts is always the state after completion of the last change actually processed.

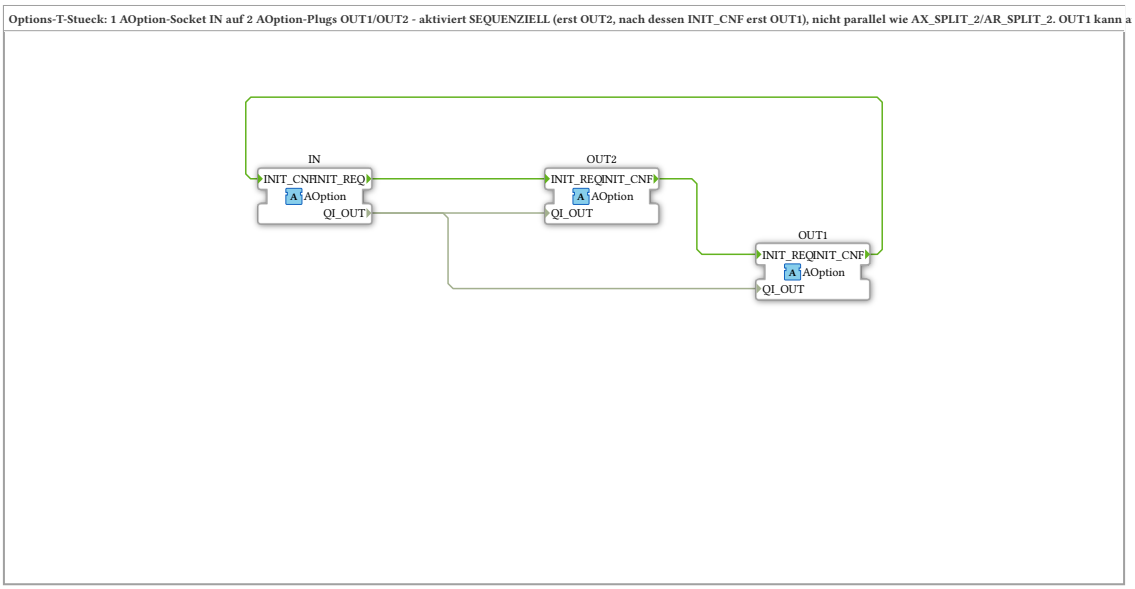
⚖ Comparison with Similar Blocks

AOption_AD_DEMUX_7 is the multi-select variant with a bit mask (`AD2`): several outputs can be active at the same time there.

2.1.9.5 splitting

AOption_T_2





• — ○ • — ○ • — ○ •

Introduction

`AOption_T_2` is a composite FB that serves as a sequential **T-piece** for the option adapter `AOption`: it receives a single option request via the socket `IN` and activates two downstream option consumers (plugs `OUT2` and `OUT1`) **one after the other**, not in parallel. `OUT2` is addressed first, and only after its `INIT_CNF` does `OUT1` follow. `IN` is confirmed only once both outputs have confirmed.

Interface Structure

Event Inputs

None (the events travel via the adapters).

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
IN	Option::AOption	Socket	Fed by an AOption output (e.g. DEMUX.OUTn).
OUT2	Option::AOption	Plug	Activated first (INIT_REQ with QI_OUT=TRUE) or deactivated last (QI_OUT=FALSE).
OUT1	Option::AOption	Plug	Activated only after OUT2.INIT_CNF, or deactivated first; can be connected to the socket IN of another AOption_T_2 for cascading.

Functionality

The internal network consists of connections only (no further FBs, no algorithms, no ECC):
Events:

- IN.INIT_REQ -> OUT2.INIT_REQ
- OUT2.INIT_CNF -> OUT1.INIT_REQ
- OUT1.INIT_CNF -> IN.INIT_CNF

Data:

- IN.QI_OUT -> OUT2.QI_OUT
- IN.QI_OUT -> OUT1.QI_OUT

A request at IN therefore triggers the chain IN.INIT_REQ -> OUT2 -> (waits for OUT2.INIT_CNF) -> OUT1 -> (waits for OUT1.INIT_CNF) -> IN.INIT_CNF. Deactivation uses the same wiring in the same order; QI_OUT then carries FALSE instead of TRUE. The event chain itself does not distinguish between activation and deactivation.

Technical Features

- **Sequential instead of parallel:** The *_SPLIT_2 blocks (e.g. AX_SPLIT_2, AR_SPLIT_2) forward an event or value in parallel to several outputs. AOption_T_2 deliberately serializes the activation so that a group that depends on a previously activated group only starts once that group is actually done.
- **Cascadable:** OUT1 can be connected to the IN socket of another AOption_T_2, so any number of groups can be started strictly one after another without each option group having to provide its own forwarding plug (cf. the daisy-chain style in OptionalGroup_001a).
- **Composite instead of ECC basic FB:** No state logic of its own is needed; the plain chaining of the events already enforces the order.

Application Scenarios

- One option request (e.g. from an AOption demultiplexer) is to activate two option groups in sequence, where the second may only start after the first has finished.
- Several cascaded AOption_T_2 for longer chains of option groups.

Summary

AOption_T_2 distributes an AOption request sequentially to two outputs (OUT2, then OUT1) without logic of its own and confirms IN only after both confirmations.

🛠️ Related Exercises

- Uebung_094b3d_AX

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.1.10 SafeArithmetic

Saturating (clamping) arithmetic function blocks, developed under the EPL-2.0 license by HR Agrartechnik GmbH. Motivated by a real hardware finding (`Uebung_011b3`): plain IEC 61131-3 arithmetic silently wraps around on overflow/underflow for integer types, and produces `INF`/`NaN` for REAL/LREAL — both dangerous for real measurement values. Every block in this library instead clamps the result into the valid range and reports the limit condition via a `LIMIT_HIT` output.

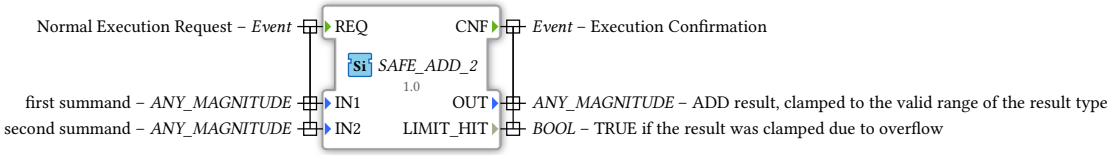
- Arithmetic Functions

2.1.10.1 Arithmetic Functions

Here you will find the SafeArithmetic library's saturating (clamping) arithmetic function blocks — drop-in, safety-oriented replacements for the standard library's plain arithmetic blocks that report an overflow/underflow/division-by-zero via `LIMIT_HIT` instead of silently wrapping around (integers) or producing `INF`/`NaN` (REAL/LREAL).

- SAFE_ADD_2
- SAFE_ADD_3
- SAFE_ADD_4
- SAFE_DIV
- SAFE_MUL_2
- SAFE_MUL_3
- SAFE_MUL_4
- SAFE_SUB

SAFE_ADD_2



• – ◦ • – ◦ • – ◦ •

Introduction

SAFE_ADD_2 is part of the **SafeArithmetic** library (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). It is a generic, 2-input addition function block that behaves like the standard library's `ADD_2`, but never lets an overflow pass silently: on overflow the result is **clamped** (saturated) into the valid range of the result type, and `LIMIT_HIT` reports that clamping happened.

This library exists because plain IEC 61131-3 arithmetic function blocks (like `ADD_2` itself, for integer types) compute the standard's specified two's-complement wraparound on overflow — correct per the standard, but dangerous for real measurement values. `Uebung_011b3` demonstrates this on real hardware: plain `F_SUB` computes `UDINT#1 - UDINT#12 = UDINT#4294967285`, a silent underflow wraparound.

Interface Structure

Event Inputs

- `REQ`: Normal Execution Request

Event Outputs

- `CNF`: Execution Confirmation

Data Inputs

- `IN1` (ANY_MAGNITUDE): first summand
- `IN2` (ANY_MAGNITUDE): second summand

Data Outputs

- `OUT` (ANY_MAGNITUDE): ADD result, clamped to the valid range of the result type
- `LIMIT_HIT` (BOOL): TRUE if the result was clamped due to overflow

Functionality

1. **Calculation Request**: `REQ` event with values for IN1, IN2.
2. **Calculation**: computes `IN1 + IN2` left-to-right, one pairwise step at a time. Each step checks for overflow; if a step overflows, its result is clamped into the result type's valid range and `LIMIT_HIT` is set (sticky — once set by any step, it stays set for the rest of the fold).
3. **Result Output**: `CNF` event with the calculated `OUT` and `LIMIT_HIT`.

Overflow / Saturation Behavior

- **Signed integer** types: on overflow, clamps to `numeric_limits::max()` or `::min()`, direction determined by the operand signs of the step that overflowed.
- **Unsigned integer** types: on overflow, clamps to `numeric_limits::max()`.
- **REAL/LREAL**: a non-finite intermediate or final result is always caught, regardless of whether the *inputs* to that step were already non-finite — `+INF`/`-INF` clamp to `numeric_limits::max()`/`::lowest()`, and `NaN` (e.g. from a prior step going non-finite and combining in a way that produces `NaN`) becomes a deterministic `0`. `LIMIT_HIT` is set in every case.
- **Incompatible types** (e.g. mixing a magnitude type with an unrelated type where no `ADD` is defined): the offending step is skipped (logged as an error), `OUT` keeps whatever value the fold had accumulated so far, and `LIMIT_HIT` is **not** set for this case — it is a type error, not a numeric limit event.

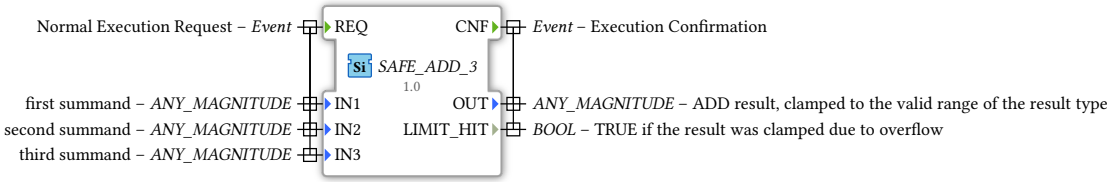
Related Function Blocks

- ADD_2 — the plain, non-saturating standard-library equivalent (silent wraparound/`INF` on overflow instead of clamping and reporting).
- Uebung_011b3 — the hardware finding that motivated this library.

Conclusion

SAFE_ADD_2 gives you the same 2-input addition as `ADD_2`, but with an explicit, type-range-aware limit signal instead of a silent wraparound (integers) or a silently-corrupted result (floats) — at the cost of the arity-≥3/signed chaining caveat documented above, which is a deliberate design trade-off, not an oversight.

SAFE_ADD_3



• – ◦ • – ◦ • – ◦ •

Introduction

SAFE_ADD_3 is part of the **SafeArithmetic** library (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). It is a generic, 3-input addition function block that behaves like the standard library's `ADD_3`, but never lets an overflow pass silently: on overflow the result is **clamped** (saturated) into the valid range of the result type, and `LIMIT_HIT` reports that clamping happened.

This library exists because plain IEC 61131-3 arithmetic function blocks (like `ADD_3` itself, for integer types) compute the standard's specified two's-complement wraparound on overflow — correct per the standard, but dangerous for real measurement values. `Uebung_011b3` demonstrates this on real hardware: plain `F_SUB` computes `UDINT#1 - UDINT#12 = UDINT#4294967285`, a silent underflow wraparound.

Interface Structure

Event Inputs

- `REQ`: Normal Execution Request

Event Outputs

- `CNF`: Execution Confirmation

Data Inputs

- IN1 (ANY_MAGNITUDE): first summand
- IN2 (ANY_MAGNITUDE): second summand
- IN3 (ANY_MAGNITUDE): third summand

Data Outputs

- OUT (ANY_MAGNITUDE): ADD result, clamped to the valid range of the result type
- LIMIT_HIT (BOOL): TRUE if the result was clamped due to overflow

Functionality

1. **Calculation Request**: REQ event with values for IN1, IN2, IN3.
2. **Calculation**: computes IN1 + IN2 + IN3 left-to-right, one pairwise step at a time. Each step checks for overflow; if a step overflows, its result is clamped into the result type's valid range and LIMIT_HIT is set (sticky — once set by any step, it stays set for the rest of the fold).
3. **Result Output**: CNF event with the calculated OUT and LIMIT_HIT.

Overflow / Saturation Behavior

- **Signed integer** types: on overflow, clamps to numeric_limits::max() or ::min(), direction determined by the operand signs of the step that overflowed.
- **Unsigned integer** types: on overflow, clamps to numeric_limits::max().
- **REAL/LREAL**: a non-finite intermediate or final result is always caught, regardless of whether the *inputs* to that step were already non-finite — +INF/-INF clamp to numeric_limits::max()/::lowest(), and NaN (e.g. from a prior step going non-finite and combining in a way that produces NaN) becomes a deterministic 0. LIMIT_HIT is set in every case.
- **Incompatible types** (e.g. mixing a magnitude type with an unrelated type where no ADD is defined): the offending step is skipped (logged as an error), OUT keeps whatever value the fold had accumulated so far, and LIMIT_HIT is **not** set for this case — it is a type error, not a numeric limit event.

Known Limitation for Chained (Arity ≥ 3) addition With SIGNED Types

SAFE_ADD_3 evaluates as a strict left-to-right pairwise fold: ((IN1 + IN2) + IN3) + ..., clamping after every step. For **signed** integer types (and REAL/LREAL, which have no unsigned notion) this means a transient intermediate overflow that a later input would otherwise cancel back into range gets permanently baked into OUT, together with a LIMIT_HIT that no longer reflects the true total.

Example (INT, IN1=20000, IN2=20000, IN3=−30000): the true result is 10000, well within INT's range, but the fold clamps after the second step and never recovers.

This is **not** a bug being silently patched — it is documented, intentional behavior: the fold evaluates as a real chain of fixed-width operations, exactly like the plain (non-safe) ADD_3 does, and reports a genuine intermediate limit condition rather than silently reordering the computation.

Unsigned chains of any arity are unaffected and always report a true overflow: for addition, a later non-negative term can never bring an already-overflowed running total back in range (the total only grows).

SAFE_ADD_2 (single pairwise operation, no chaining) always reports a true, correct overflow regardless of signedness — the ambiguity only exists for arity ≥ 3.

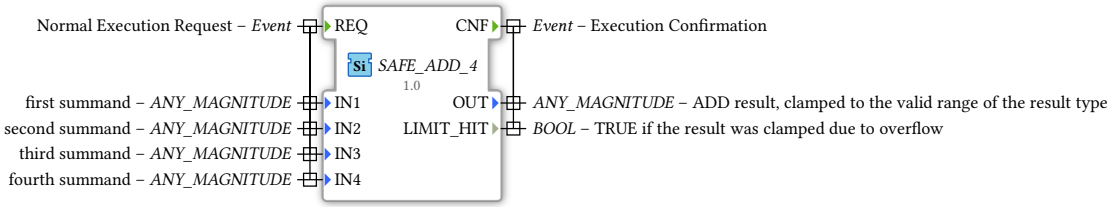
Related Function Blocks

- ADD_3 — the plain, non-saturating standard-library equivalent (silent wraparound/INF on overflow instead of clamping and reporting).
- Uebung_011b3 — the hardware finding that motivated this library.

Conclusion

SAFE_ADD_3 gives you the same 3-input addition as ADD_3, but with an explicit, type-range-aware limit signal instead of a silent wraparound (integers) or a silently-corrupted result (floats) — at the cost of the arity-≥-3/signed chaining caveat documented above, which is a deliberate design trade-off, not an oversight.

SAFE_ADD_4



• – ◦ • – ◦ • – ◦ •

Introduction

SAFE_ADD_4 is part of the **SafeArithmetic** library (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). It is a generic, 4-input addition function block that behaves like the standard library's `ADD_4`, but never lets an overflow pass silently: on overflow the result is **clamped** (saturated) into the valid range of the result type, and `LIMIT_HIT` reports that clamping happened.

This library exists because plain IEC 61131-3 arithmetic function blocks (like `ADD_4` itself, for integer types) compute the standard's specified two's-complement wraparound on overflow — correct per the standard, but dangerous for real measurement values. `Uebung_011b3` demonstrates this on real hardware: plain `F_SUB` computes `UDINT#1 - UDINT#12 = UDINT#4294967285`, a silent underflow wraparound.

Interface Structure

Event Inputs

- `REQ`: Normal Execution Request

Event Outputs

- `CNF`: Execution Confirmation

Data Inputs

- `IN1` (ANY_MAGNITUDE): first summand
- `IN2` (ANY_MAGNITUDE): second summand
- `IN3` (ANY_MAGNITUDE): third summand
- `IN4` (ANY_MAGNITUDE): fourth summand

Data Outputs

- `OUT` (ANY_MAGNITUDE): ADD result, clamped to the valid range of the result type
- `LIMIT_HIT` (BOOL): TRUE if the result was clamped due to overflow

Functionality

1. **Calculation Request**: `REQ` event with values for IN1, IN2, IN3, IN4.
2. **Calculation**: computes `IN1 + IN2 + IN3 + IN4` left-to-right, one pairwise step at a time. Each step checks for overflow; if a step overflows, its result is clamped into the result type's valid range and `LIMIT_HIT` is set (sticky — once set by any step, it stays set for the rest of the fold).
3. **Result Output**: `CNF` event with the calculated `OUT` and `LIMIT_HIT`.

Overflow / Saturation Behavior

- **Signed integer** types: on overflow, clamps to `numeric_limits::max()` or `::min()`, direction determined by the operand signs of the step that overflowed.
- **Unsigned integer** types: on overflow, clamps to `numeric_limits::max()`.
- **REAL/LREAL**: a non-finite intermediate or final result is always caught, regardless of whether the *inputs* to that step were already non-finite — `+INF`/`-INF` clamp to `numeric_limits::max()`/`::lowest()`, and `NaN` (e.g. from a prior step going non-finite and combining in a way that produces `NaN`) becomes a deterministic `0`. `LIMIT_HIT` is set in every case.
- **Incompatible types** (e.g. mixing a magnitude type with an unrelated type where no `ADD` is defined): the offending step is skipped (logged as an error), `OUT` keeps whatever value the fold had accumulated so far, and `LIMIT_HIT` is **not** set for this case — it is a type error, not a numeric limit event.

Known Limitation for Chained (Arity ≥ 3) addition With SIGNED Types

`SAFE_ADD_4` evaluates as a strict left-to-right pairwise fold: `((IN1 + IN2) + IN3) + ...`, clamping after every step. For **signed** integer types (and REAL/LREAL, which have no unsigned notion) this means a transient intermediate overflow that a later input would otherwise cancel back into range gets permanently baked into `OUT`, together with a `LIMIT_HIT` that no longer reflects the true total.

Example (INT, IN1=20000, IN2=20000, IN3=−30000, IN4=0): the true result is `10000`, well within INT's range, but the fold clamps after the second step and never recovers.

This is **not** a bug being silently patched — it is documented, intentional behavior: the fold evaluates as a real chain of fixed-width operations, exactly like the plain (non-safe) `ADD_4` does, and reports a genuine intermediate limit condition rather than silently reordering the computation.

Unsigned chains of any arity are unaffected and always report a true overflow: for addition, a later non-negative term can never bring an already-overflowed running total back in range (the total only grows).

`SAFE_ADD_2` (single pairwise operation, no chaining) always reports a true, correct overflow regardless of signedness — the ambiguity only exists for arity ≥ 3.

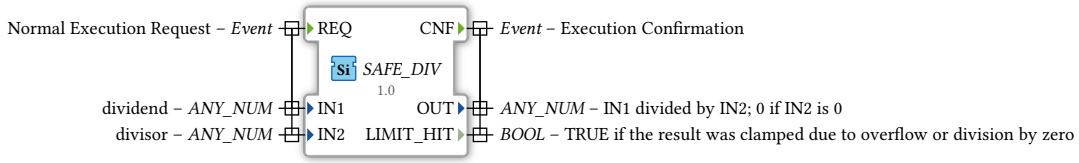
Related Function Blocks

- ADD_4 — the plain, non-saturating standard-library equivalent (silent wraparound/`INF` on overflow instead of clamping and reporting).
- Uebung_011b3 — the hardware finding that motivated this library.

Conclusion

SAFE_ADD_4 gives you the same 4-input addition as `ADD_4`, but with an explicit, type-range-aware limit signal instead of a silent wraparound (integers) or a silently-corrupted result (floats) — at the cost of the arity-≥-3/signed chaining caveat documented above, which is a deliberate design trade-off, not an oversight.

SAFE_DIV



• – ◦ • – ◦ • – ◦ •

Introduction

SAFE_DIV is part of the **SafeArithmetic** library (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). It is a fixed 2-input division function block that behaves like the standard library's `F_DIV`, but never traps, produces `INF`/`NaN`, or lets an overflow pass silently: division by zero and overflow are treated as **clamp** conditions, and `LIMIT_HIT` reports that clamping happened.

Like `SAFE_SUB`, `SAFE_DIV` is **not** generic-arity: division is neither associative nor commutative, so a "chained" division has no well-defined meaning the way extensible ADD/MUL do.

Interface Structure

Event Inputs

- `REQ`: Normal Execution Request

Event Outputs

- `CNF`: Execution Confirmation

Data Inputs

- `IN1` (ANY_NUM): dividend
- `IN2` (ANY_NUM): divisor

Data Outputs

- `OUT` (ANY_NUM): IN1 divided by IN2; 0 if IN2 is 0
- `LIMIT_HIT` (BOOL): TRUE if the result was clamped due to overflow or division by zero

Functionality

1. **Calculation Request**: `REQ` event with values for IN1 and IN2.
2. **Division**: computes `IN1 / IN2`.
3. **Result Output**: `CNF` event with the calculated `OUT` and `LIMIT_HIT`.

Overflow / Division-by-Zero Behavior

- **Any integer** type, `IN2 == 0`: clamps to `0` with `LIMIT_HIT = TRUE`, instead of trapping (integer division by zero is undefined behavior in plain C++ and would crash `F_DIV`).
- **Signed integer** types: the one remaining case that can overflow, `INT_MIN / -1`, clamps to `numeric_limits::max()`.
- **REAL/LREAL**, `IN2 == 0.0`: clamps to `0` with `LIMIT_HIT = TRUE`, instead of producing IEEE-754 `+INF`/`-INF`/`NaN` (`0.0 / 0.0`).
- **REAL/LREAL**, `IN2 != 0.0`: a non-finite intermediate or final result is always caught, regardless of whether the *inputs* were already non-finite — `+INF`/`-INF` clamp to `numeric_limits::max()`/`::lowest()`, and `NaN` becomes a deterministic `0`. `LIMIT_HIT` is set in every case.
- **Incompatible types** (e.g. mixing a numeric type with an unrelated type where no division is defined): the result is undefined by this operation (logged as an error), `OUT` is left unchanged, and `LIMIT_HIT` is **not** set for this case — it is a type error, not a numeric limit event.

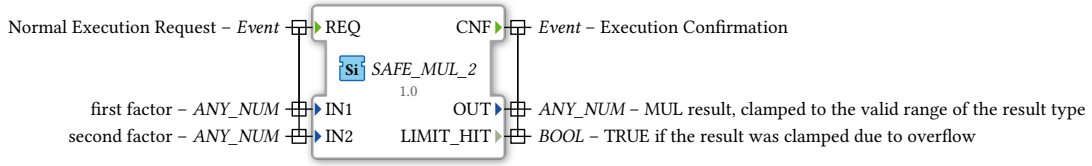
Related Function Blocks

- F_DIV — the plain, non-saturating standard-library equivalent (undefined behavior on integer division by zero, `INF`/`NaN` on float division by/of zero, instead of clamping and reporting).
- Uebung_011b7 — a division-by-zero demo with `SAFE_DIV`, `LIMIT_HIT` wired to a second output so the clamping is visible on hardware.

Conclusion

SAFE_DIV gives you the same division as `F_DIV`, but with an explicit, type-range-aware limit signal instead of undefined behavior (integer division by zero) or a silently-corrupted result (float overflow/`NaN`).

SAFE_MUL_2



• – ◦ • – ◦ • – ◦ •

Introduction

SAFE_MUL_2 is part of the **SafeArithmetic** library (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). It is a generic, 2-input multiplication function block that behaves like the standard library's `F_MUL`, but never lets an overflow pass silently: on overflow the result is **clamped** (saturated) into the valid range of the result type, and `LIMIT_HIT` reports that clamping happened.

This library exists because plain IEC 61131-3 arithmetic function blocks (like `F_MUL` itself, for integer types) compute the standard's specified two's-complement wraparound on overflow — correct per the standard, but dangerous for real measurement values. `Uebung_011b3` demonstrates this on real hardware: plain `F_SUB` computes `UDINT#1 - UDINT#12 = UDINT#4294967285`, a silent underflow wraparound.

Interface Structure

Event Inputs

- `REQ`: Normal Execution Request

Event Outputs

- `CNF`: Execution Confirmation

Data Inputs

- `IN1` (ANY_NUM): first factor

- IN2 (ANY_NUM): second factor

Data Outputs

- OUT (ANY_NUM): MUL result, clamped to the valid range of the result type
- LIMIT_HIT (BOOL): TRUE if the result was clamped due to overflow

Functionality

1. **Calculation Request**: REQ event with values for IN1, IN2.
2. **Calculation**: computes IN1 * IN2 left-to-right, one pairwise step at a time. Each step checks for overflow; if a step overflows, its result is clamped into the result type's valid range and LIMIT_HIT is set (sticky — once set by any step, it stays set for the rest of the fold).
3. **Result Output**: CNF event with the calculated OUT and LIMIT_HIT.

Overflow / Saturation Behavior

- **Signed integer** types: on overflow, clamps to numeric_limits::max() or ::min(), direction determined by the operand signs of the step that overflowed.
- **Unsigned integer** types: on overflow, clamps to numeric_limits::max().
- **REAL/LREAL**: a non-finite intermediate or final result is always caught, regardless of whether the *inputs* to that step were already non-finite — +INF/-INF clamp to numeric_limits::max()/::lowest(), and NaN (e.g. from a prior step going non-finite and combining in a way that produces NaN) becomes a deterministic 0. LIMIT_HIT is set in every case.
- **Incompatible types** (e.g. mixing a magnitude type with an unrelated type where no MUL is defined): the offending step is skipped (logged as an error), OUT keeps whatever value the fold had accumulated so far, and LIMIT_HIT is **not** set for this case — it is a type error, not a numeric limit event.

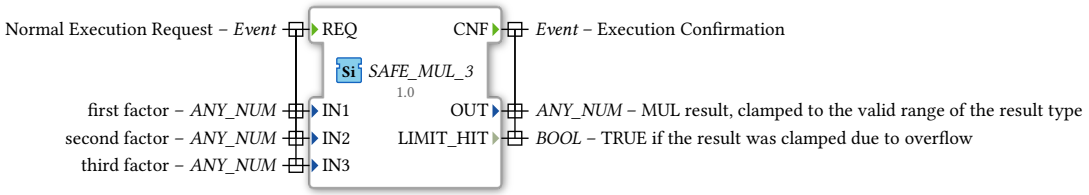
Related Function Blocks

- F_MUL — the plain, non-saturating standard-library equivalent (silent wraparound/INF on overflow instead of clamping and reporting).
- Uebung_011b3 — the hardware finding that motivated this library.

Conclusion

SAFE_MUL_2 gives you the same 2-input multiplication as F_MUL, but with an explicit, type-range-aware limit signal instead of a silent wraparound (integers) or a silently-corrupted result (floats) — at the cost of the arity-≥-3/signed chaining caveat documented above, which is a deliberate design trade-off, not an oversight.

SAFE_MUL_3



• – ◦ • – ◦ • – ◦ •

Introduction

SAFE_MUL_3 is part of the **SafeArithmetic** library (SafeArithmetic::arithmetic, EPL-2.0, HR Agrartechnik GmbH). It is a generic, 3-input multiplication function block that extends the standard library's fixed 2-input F_MUL pattern to 3 inputs (the standard library has no generic multi-input MUL_n yet, unlike ADD_2/3/4), and never lets an overflow pass silently: on overflow the result is **clamped** (saturated) into the valid range of the result type, and LIMIT_HIT reports that clamping happened.

This library exists because plain IEC 61131-3 arithmetic function blocks (like F_MUL itself, for integer types) compute the standard's specified two's-complement wraparound on overflow — correct per the standard, but dangerous for real measurement values. Uebung_011b3 demonstrates this on real hardware: plain F_SUB computes UDINT#1 - UDINT#12 = UDINT#4294967285, a silent underflow wraparound.

Interface Structure

Event Inputs

- `REQ`: Normal Execution Request

Event Outputs

- `CNF`: Execution Confirmation

Data Inputs

- `IN1` (ANY_NUM): first factor
- `IN2` (ANY_NUM): second factor
- `IN3` (ANY_NUM): third factor

Data Outputs

- `OUT` (ANY_NUM): MUL result, clamped to the valid range of the result type
- `LIMIT_HIT` (BOOL): TRUE if the result was clamped due to overflow

Functionality

1. **Calculation Request**: `REQ` event with values for IN1, IN2, IN3.
2. **Calculation**: computes `IN1 * IN2 * IN3` left-to-right, one pairwise step at a time. Each step checks for overflow; if a step overflows, its result is clamped into the result type's valid range and `LIMIT_HIT` is set (sticky — once set by any step, it stays set for the rest of the fold).
3. **Result Output**: `CNF` event with the calculated `OUT` and `LIMIT_HIT`.

Overflow / Saturation Behavior

- **Signed integer** types: on overflow, clamps to `numeric_limits::max()` or `::min()`, direction determined by the operand signs of the step that overflowed.
- **Unsigned integer** types: on overflow, clamps to `numeric_limits::max()`.
- **REAL/LREAL**: a non-finite intermediate or final result is always caught, regardless of whether the *inputs* to that step were already non-finite — `+INF`/`-INF` clamp to `numeric_limits::max()`/`::lowest()`, and `NaN` (e.g. from a prior step going non-finite and combining in a way that produces `NaN`) becomes a deterministic `0`. `LIMIT_HIT` is set in every case.
- **Incompatible types** (e.g. mixing a magnitude type with an unrelated type where no `MUL` is defined): the offending step is skipped (logged as an error), `OUT` keeps whatever value the fold had accumulated so far, and `LIMIT_HIT` is **not** set for this case — it is a type error, not a numeric limit event.

Known Limitation for Chained (Arity ≥ 3) multiplication With SIGNED Types

`SAFE_MUL_3` evaluates as a strict left-to-right pairwise fold: `((IN1 * IN2) * IN3) * ...`, clamping after every step. For **signed** integer types (and REAL/LREAL, which have no unsigned notion) this means a transient intermediate overflow that a later input would otherwise cancel back into range gets permanently baked into `OUT`, together with a `LIMIT_HIT` that no longer reflects the true total.

Example (INT, IN1=40000, IN2=40000, IN3=0): the true result is `0`, well within INT's range, but the fold clamps after the second step and never recovers.

This is **not** a bug being silently patched — it is documented, intentional behavior: the fold evaluates as a real chain of fixed-width operations, exactly like the plain (non-safe) `F_MUL` does, and reports a genuine intermediate limit condition rather than silently reordering the computation.

Unsigned chains of any arity are unaffected and always report a true overflow: a later factor is either 0 (collapses the result cleanly to 0 regardless of any prior clamping) or ≥ 1 (keeps the product non-decreasing), so an unsigned chain's intermediate clamp can never diverge from the true total.

`SAFE_MUL_2` (single pairwise operation, no chaining) always reports a true, correct overflow regardless of signedness — the ambiguity only exists for arity ≥ 3.

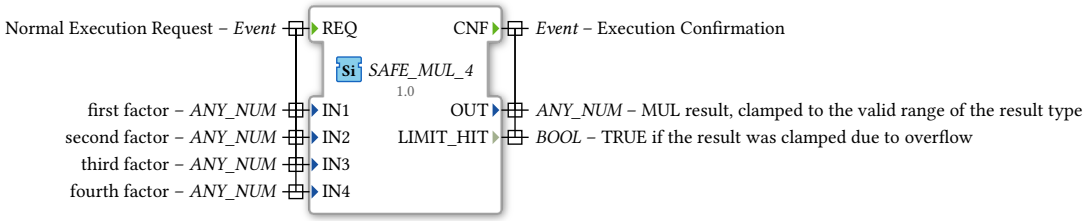
Related Function Blocks

- F_MUL — the fixed 2-input, non-saturating standard-library block this pattern extends (silent wrap-around/`INF` on overflow instead of clamping and reporting; no 3-input standard equivalent exists).
- Uebung_011b3 — the hardware finding that motivated this library.

Conclusion

SAFE_MUL_3 extends F_MUL's multiplication to 3 inputs, with an explicit, type-range-aware limit signal instead of a silent wraparound (integers) or a silently-corrupted result (floats) — at the cost of the arity-≥-3/signed chaining caveat documented above, which is a deliberate design trade-off, not an oversight.

SAFE_MUL_4



• – ◦ • – ◦ • – ◦ •

Introduction

SAFE_MUL_4 is part of the **SafeArithmetic** library (`SafeArithmetic::arithmetic`, EPL-2.0, HR Agrartechnik GmbH). It is a generic, 4-input multiplication function block that extends the standard library's fixed 2-input `F_MUL` pattern to 4 inputs (the standard library has no generic multi-input `MUL_n` yet, unlike `ADD_2/3/4`), and never lets an overflow pass silently: on overflow the result is **clamped** (saturated) into the valid range of the result type, and `LIMIT_HIT` reports that clamping happened.

This library exists because plain IEC 61131-3 arithmetic function blocks (like `F_MUL` itself, for integer types) compute the standard's specified two's-complement wraparound on overflow — correct per the standard, but dangerous for real measurement values. `Uebung_011b3` demonstrates this on real hardware: plain `F_SUB` computes `UDINT#1 - UDINT#12 = UDINT#4294967285`, a silent underflow wraparound.

Interface Structure

Event Inputs

- `REQ`: Normal Execution Request

Event Outputs

- `CNF`: Execution Confirmation

Data Inputs

- `IN1` (ANY_NUM): first factor
- `IN2` (ANY_NUM): second factor
- `IN3` (ANY_NUM): third factor
- `IN4` (ANY_NUM): fourth factor

Data Outputs

- `OUT` (ANY_NUM): MUL result, clamped to the valid range of the result type
- `LIMIT_HIT` (BOOL): TRUE if the result was clamped due to overflow

Functionality

1. **Calculation Request**: `REQ` event with values for IN1, IN2, IN3, IN4.
2. **Calculation**: computes `IN1 * IN2 * IN3 * IN4` left-to-right, one pairwise step at a time. Each step checks for overflow; if a step overflows, its result is clamped into the result type's valid range and `LIMIT_HIT` is set (sticky — once set by any step, it stays set for the rest of the fold).
3. **Result Output**: `CNF` event with the calculated `OUT` and `LIMIT_HIT`.

Overflow / Saturation Behavior

- **Signed integer** types: on overflow, clamps to `numeric_limits::max()` or `::min()`, direction determined by the operand signs of the step that overflowed.
- **Unsigned integer** types: on overflow, clamps to `numeric_limits::max()`.
- **REAL/LREAL**: a non-finite intermediate or final result is always caught, regardless of whether the *inputs* to that step were already non-finite — `+INF`/`-INF` clamp to `numeric_limits::max()`/`::lowest()`,

and NaN (e.g. from a prior step going non-finite and combining in a way that produces NaN) becomes a deterministic 0. LIMIT_HIT is set in every case.
- **Incompatible types** (e.g. mixing a magnitude type with an unrelated type where no MUL is defined): the offending step is skipped (logged as an error), OUT keeps whatever value the fold had accumulated so far, and LIMIT_HIT is **not** set for this case — it is a type error, not a numeric limit event.

Known Limitation for Chained (Arity ≥ 3) multiplication With SIGNED Types

SAFE_MUL_4 evaluates as a strict left-to-right pairwise fold: ((IN1 * IN2) * IN3) * ..., clamping after every step. For **signed** integer types (and REAL/LREAL, which have no unsigned notion) this means a transient intermediate overflow that a later input would otherwise cancel back into range gets permanently baked into OUT, together with a LIMIT_HIT that no longer reflects the true total.

Example (INT, IN1=40000, IN2=40000, IN3=0, IN4=1): the true result is 0, well within INT's range, but the fold clamps after the second step and never recovers.

This is **not** a bug being silently patched — it is documented, intentional behavior: the fold evaluates as a real chain of fixed-width operations, exactly like the plain (non-safe) F_MUL does, and reports a genuine intermediate limit condition rather than silently reordering the computation.

Unsigned chains of any arity are unaffected and always report a true overflow: a later factor is either 0 (collapses the result cleanly to 0 regardless of any prior clamping) or ≥ 1 (keeps the product non-decreasing), so an unsigned chain's intermediate clamp can never diverge from the true total.

SAFE_MUL_2 (single pairwise operation, no chaining) always reports a true, correct overflow regardless of signedness — the ambiguity only exists for arity ≥ 3.

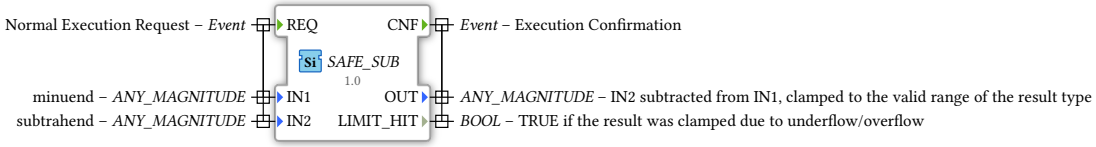
Related Function Blocks

- F_MUL — the fixed 2-input, non-saturating standard-library block this pattern extends (silent wrap-around/INF on overflow instead of clamping and reporting; no 4-input standard equivalent exists).
- Uebung_011b3 — the hardware finding that motivated this library.

Conclusion

SAFE_MUL_4 extends F_MUL's multiplication to 4 inputs, with an explicit, type-range-aware limit signal instead of a silent wraparound (integers) or a silently-corrupted result (floats) — at the cost of the arity-≥-3/signed chaining caveat documented above, which is a deliberate design trade-off, not an oversight.

SAFE_SUB



• – ◦ • – ◦ • – ◦ •

Introduction

SAFE_SUB is part of the **SafeArithmetic** library (SafeArithmetic::arithmetic, EPL-2.0, HR Agrartechnik GmbH). It is a fixed 2-input subtraction function block that behaves like the standard library's F_SUB, but never lets an underflow/overflow pass silently: on a limit violation the result is **clamped** (saturated) into the valid range of the result type, and LIMIT_HIT reports that clamping happened.

This is the block that motivated the whole library: Uebung_011b3 demonstrates on real ESP32-P4 hardware that plain F_SUB computes UDINT#1 - UDINT#12 = UDINT#4294967285 — exactly the IEC 61131-3-specified two's-complement wraparound for unsigned types, but dangerous for real measurement values (setpoint differences, remaining-distance calculations). SAFE_SUB computes the same subtraction instead as UDINT#0 with LIMIT_HIT = TRUE.

Unlike SAFE_ADD_2/3/4/SAFE_MUL_2/3/4, SAFE_SUB is **not** generic-arity: subtraction is neither associative nor commutative, so an N-ary "extensible" SAFE_SUB_n would not have a well-defined meaning the way extensible ADD/MUL do.

Interface Structure

Event Inputs

- `REQ`: Normal Execution Request

Event Outputs

- `CNF`: Execution Confirmation

Data Inputs

- `IN1` (ANY_MAGNITUDE): minuend
- `IN2` (ANY_MAGNITUDE): subtrahend

Data Outputs

- `OUT` (ANY_MAGNITUDE): IN2 subtracted from IN1, clamped to the valid range of the result type
- `LIMIT_HIT` (BOOL): TRUE if the result was clamped due to underflow/overflow

Functionality

1. **Calculation Request**: `REQ` event with values for IN1 and IN2.
2. **Subtraction**: computes `IN1 - IN2`.
3. **Result Output**: `CNF` event with the calculated `OUT` and `LIMIT_HIT`.

Overflow / Underflow Behavior

- **Signed integer** types: on overflow/underflow, clamps to `numeric_limits::min()` or `::max()`, direction determined by the sign of `IN2`.
- **Unsigned integer** types: subtraction can only ever underflow (go below 0), so it always clamps to `0` — this is exactly the `Uebung_011b3` case: `UDINT#1 - UDINT#12` clamps to `UDINT#0` with `LIMIT_HIT = TRUE` instead of wrapping to `UDINT#4294967285`.
- **REAL/LREAL**: a non-finite intermediate or final result is always caught, regardless of whether the *inputs* were already non-finite — `+INF`/`-INF` clamp to `numeric_limits::max()`/`::lowest()`, and `NaN` (e.g. `INF - INF`) becomes a deterministic `0`. `LIMIT_HIT` is set in every case.
- **Incompatible types** (e.g. mixing a magnitude type with an unrelated type where no subtraction is defined): the result is undefined by this operation (logged as an error), `OUT` is left unchanged, and `LIMIT_HIT` is **not** set for this case — it is a type error, not a numeric limit event.

Related Function Blocks

- F_SUB — the plain, non-saturating standard-library equivalent (silent wraparound/`INF` on overflow instead of clamping and reporting).
- Uebung_011b3 — the hardware finding that motivated this library.
- Uebung_011b5 — the same `1 - 12` scenario re-run with `SAFE_SUB`, with `LIMIT_HIT` wired to a second output so the clamping is visible on hardware.

Conclusion

SAFE_SUB gives you the same subtraction as `F_SUB`, but with an explicit, type-range-aware limit signal instead of a silent wraparound (integers) or a silently-corrupted result (floats) — directly closing the gap `Uebung_011b3` found in the standard library.

2.1.11 adapter

2.1.11.1 🎧 Podcast

- Code Renovation with AX Adapters: How Eclipse 4diac™ Conquers Complexity Through Signal Bundling
- From 1400 Errors to Clean Code: Migrating the “Grain Hoe” to Eclipse 4diac™ 3.0 and the Power of AX Adapters

---

🌐 Related Topic Subpages on ms-muc-docs.de

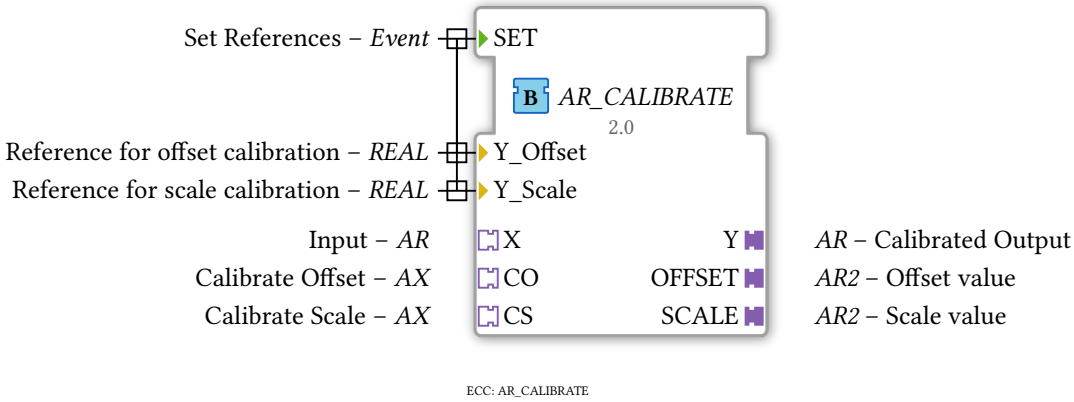
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
-

2.1.11.2 Engineering

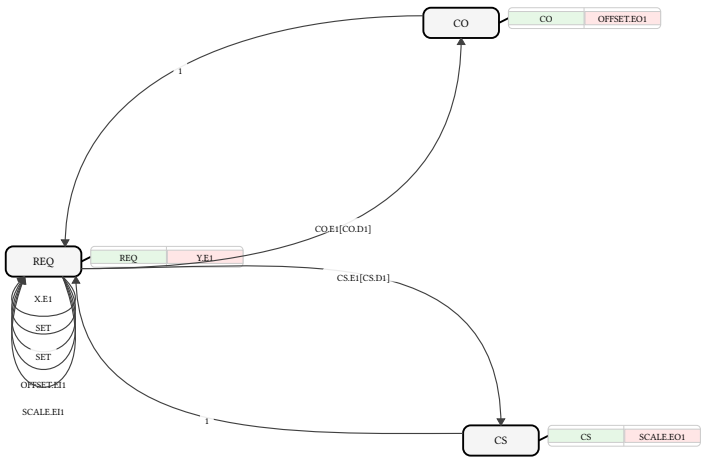
Numerical Index (Engineering)

Here you will find all exercises from the `Engineering` workspace, listed in ascending order. ...

AR_CALIBRATE



ECC: AR_CALIBRATE



• – ○ • – ○ • – ○ •

Introduction

The function block `AR_CALIBRATE` is used for offset and scale calibration of an analog input signal provided via an adapter. It enables two-stage calibration: First, the offset is determined by comparison with a reference value during active calibration (`CO`), and then the scale is determined using a second reference (`CS`). The calibrated output signal is calculated continuously.

Interface Structure

Event Inputs

Name	Type	With	Description
`SET`	`EInit`	`Y_Offset`, `Y_Scale`	Sets the reference values for offset and scale calibration

CO	Event		Atomic trigger for offset calibration (reads current raw value from X)
CS	Event		Atomic trigger for scale calibration (reads current raw value from X)

Event Outputs

This function block does not have its own event outputs. However, events are sent via the connected adapters (e.g., Y.E1, OFFSET.EO1, SCALE.EO1).

Data Inputs

Name	Type	Description
Y_Offset	REAL	Reference value for offset calibration
Y_Scale	REAL	Reference value for scale calibration

Data Outputs

The FB has no direct data outputs. The calculated values are provided via the adapters Y, OFFSET, and SCALE.

Adapters

Plugs (providing interfaces):

Adapter	Type	Description
Y	adapter::types::unidirectional::AR	Calibrated output (data + event)
OFFSET	adapter::types::bidirectional::AR2	Returns the calculated offset value (bidirectional)
SCALE	adapter::types::bidirectional::AR2	Returns the calculated scale factor (bidirectional)

Sockets (user interfaces):

Adapter	Type	Description
X	adapter::types::unidirectional::AR	Analog input value (unidirectional)

Functionality

The function block operates with three basic algorithms, which are executed in the states REQ, CO, and CS:

- **REQ** (Normal Operation):
  `Y.D1 := (X.D1 + OFFSET.DI1) * SCALE.DI1`
  The output value is calculated from the input, the current offset, and the scale factor.
- **CO** (Offset Calibration):
  `OFFSET.DO1 := Y_Offset - X.D1`
  The offset is determined as the difference between the reference value Y_Offset and the current input value.
- **CS** (Scale Calibration):
  `SCALE.DO1 := Y_Scale / (X.D1 + OFFSET.DI1)`
  The scale factor is calculated by dividing the reference value Y_Scale by the input value corrected by the offset.

Calibration Procedure:

1. In state REQ, the function block (FB) is transitioned to state CO by receiving event CO. There, the offset is calculated and output via adapter OFFSET. The FB then immediately returns to state REQ.
2. Scale calibration is performed analogously via event CS and state CS.
3. The event SET updates the reference values Y_Offset and Y_Scale without exiting the calibration state.
4. Repeated measurements (via X.E1) update the output Y with the current calibration parameters.

The transitions between the states are defined as follows:

- REQ ⟶ REQ: at X.E1, SET, OFFSET.EI1, or SCALE.EI1
- REQ ⟶ CO: when event CO is received
- REQ ⟶ CS: when event CS is received
- CO ⟶ REQ: always (condition 1)
- CS ⟶ REQ: always (condition 1)

Technical Features

- **Atomic Event Triggering**: Calibration commands CO and CS are defined as direct event inputs, eliminating intermediate AX adapters and data-guard checks.
- **Bidirectional Calibration Adapters**: The adapters OFFSET and SCALE are bidirectional (AR2), allowing them to be written by the calibration FB and read by external modules.
- **Two-Stage Calibration**: Offset and scaling are calibrated sequentially. The scaling factor already uses the determined offset to ensure correct linear correction.
- **No Own Event Output**: The function block does not generate its own events but triggers them via the adapter events (e.g., Y.E1 after each calculation).

State Overview

State	Description	Action
REQ	Normal Operation – Calculation of the Calibrated Output	Executes algorithm REQ, sends event to Y.E1
CO	Offset Calibration	Executes algorithm CO, sends event to OFFSET.EO1
CS	Scale Calibration	Executes algorithm CS, sends event to SCALE.EO1

Application Scenarios

- **Industrial Measurement Processing**: Calibration of pressure sensors, temperature sensors, or other analog transmitters where offset and gain need readjustment.
- **Measurement Chain Calibration**: After connecting a sensor, the entire chain can be linearized by establishing a zero point (offset) and a second reference point (scale).
- **Automated Recalibration**: Triggering event inputs CO and CS can compensate for sensor drift during operation.

Comparison with Similar Blocks

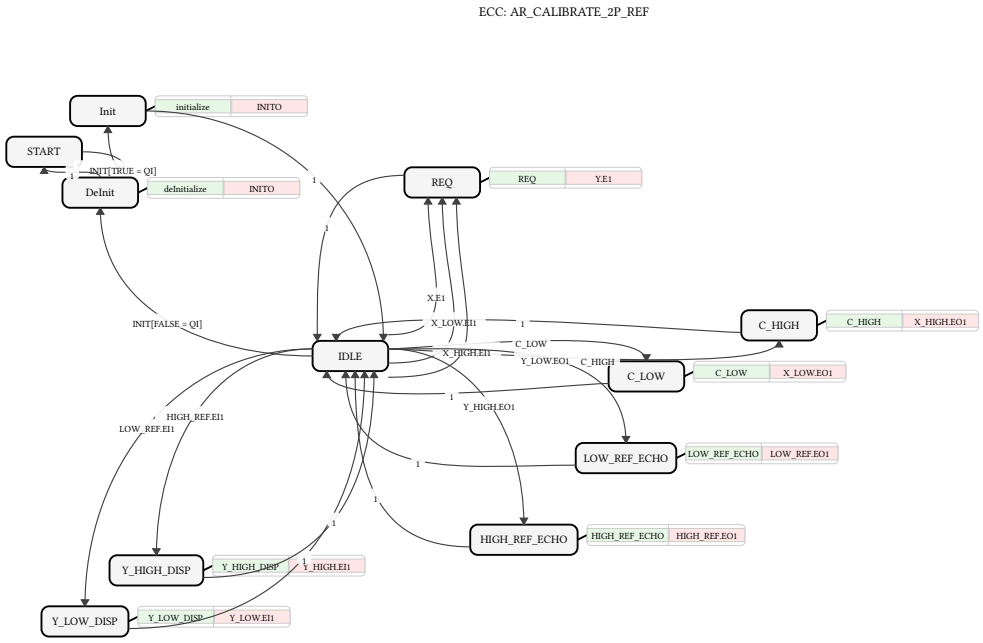
Details on calibration methods can be found under Two-point and Three-point Calibration Methods.

Simple scaling blocks (e.g., SCALE) offer fixed multiplication and addition without automatic calibration. In contrast, AR_CALIBRATE integrates the complete calibration logic—including reference values and event-driven activation—making it suitable for dynamic environments where parameters must be adjusted during operation.

Conclusion

The AR_CALIBRATE function block provides a robust solution for offset and scale calibration of analog signals. Thanks to its atomic event inputs and clear state machine, it integrates seamlessly into automation solutions and enables accurate, repeatable measurement corrections.

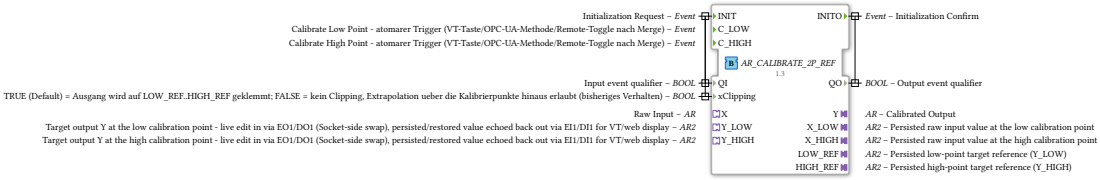
AR_CALIBRATE_2P_REF



• — ○ • — ○ • — ○ •

Introduction

`AR_CALIBRATE_2P_REF` is an adapter-based 2-point calibration (Low/High). Both points are stored independently of each other, so there is no ordering dependency as with `AR_CALIBRATE_SQ_REF`. The reference targets (`Y_LOW`/`Y_HIGH`) are live `AR2` sockets and are persisted via two additional `AR2` plugs (`LOW_REF`/`HIGH_REF`). Interpolation happens live on every `X.E1`.



Interface Structure

Event Inputs

- `INIT`: Initialization or deinitialization. Associated data: `QI`, `xClipping`.
- `C_LOW`: Atomic trigger to calibrate the low point (VT button, OPC UA method or remote toggle after merge).
- `C_HIGH`: Atomic trigger to calibrate the high point.

Event Outputs

- `INITO`: Initialization confirmation. Associated data: `QO`.

Data Inputs

Name	Data type	Default	Comment
`QI`	`BOOL`		Input qualifier (TRUE = enable, FALSE = disable).
`xClipping`	`BOOL`	`TRUE`	TRUE: output is clamped to `LOW_REF..HIGH_REF`; FALSE: no clipping, extrapolation beyond the calibration points allowed.

Data Outputs

- `QO` (`BOOL`): Output qualifier.

Adapters

Direction	Name	Adapter type	Comment
Plug (output)	`Y`	`adapter::types::unidirectional::AR`	Calibrated output.
Plug (output)	`X_LOW`	`adapter::types::bidirectional::AP2`	Persisted raw value at the low calibration point.
Plug (output)	`X_HIGH`	`adapter::types::bidirectional::AP2`	Persisted raw value at the high calibration point.
Plug (output)	`LOW_REF`	`adapter::types::bidirectional::AP2`	Persisted low-point target (Y_LOW).
Plug (output)	`HIGH_REF`	`adapter::types::bidirectional::AP2`	Persisted high-point target (Y_HIGH).
Socket (input)	`X`	`adapter::types::unidirectional::AR`	Raw input.
Socket (input)	`Y_LOW`	`adapter::types::bidirectional::AP2`	Target output at the low point: live edit in via `EO1`/`DO1`, the persisted/restored value is echoed back out via `EI1`/`DI1` for VT/web display.
Socket (input)	`Y_HIGH`	`adapter::types::bidirectional::AP2`	Same as `Y_LOW`, for the high point.

Functionality

The block is a basic FB. In state `IDLE` it reacts to the following events:

Event	State	Action
`X.EI1`, `X_LOW.EI1`, `X_HIGH.EI1`	`REQ`	Computes the calibrated value and emits `Y.EI1`.
`C_LOW`	`C_LOW`	`X_LOW.DO1 := X.D1`, emits `X_LOW.EO1`.
`C_HIGH`	`C_HIGH`	`X_HIGH.DO1 := X.D1`, emits `X_HIGH.EO1`.
`Y_LOW.EO1`	`LOW_REF_ECHO`	`LOW_REF.DO1 := Y_LOW.DO1`, emits `LOW_REF.EO1` (persist).
`Y_HIGH.EO1`	`HIGH_REF_ECHO`	`HIGH_REF.DO1 := Y_HIGH.DO1`, emits `HIGH_REF.EO1`.
`LOW_REF.EI1`	`Y_LOW_DISP`	`Y_LOW.DI1 := LOW_REF.DI1`, emits `Y_LOW.EI1` (display after boot and after every change).
`HIGH_REF.EI1`	`Y_HIGH_DISP`	`Y_HIGH.DI1 := HIGH_REF.DI1`, emits `Y_HIGH.EI1`.

All action states return to `IDLE` immediately. Initialization runs `START` → `Init` (`INIT` with `QI = TRUE`, `QO := QI`, `INITO`) → `IDLE`; `INIT` with `QI = FALSE` leads from `IDLE` via `DeInit` (`QO := FALSE`, `INITO`) back to `START`.

Calculation in state `REQ`:

1. `QO := QI`. The calculation is only performed when `QI = TRUE`.
2. Internal latched flags: `xLowCal` / `xHighCal` become TRUE as soon as `X_LOW.DI1` / `X_HIGH.DI1` differs from the default `0.0`.
3. If both points are calibrated: `Y.D1 = LOW_REF + (X - X_LOW) * (HIGH_REF - LOW_REF) / (X_HIGH - X_LOW)`; if `X_HIGH = X_LOW`, `LOW_REF` is output.
4. If only low is calibrated, `LOW_REF` is output; if only high is calibrated, `HIGH_REF`; if nothing is calibrated, `LOW_REF`.
5. Clipping to `[LOW_REF, HIGH_REF]` only if `xClipping` is TRUE and both points are calibrated.

Technical Details

- **No ordering dependency:** `C_LOW` and `C_HIGH` store their raw value independently; each point can be re-calibrated individually at any time.
- **Partial calibration:** As long as only one point is calibrated, the block does not interpolate against the other point still at `0.0` but outputs the known REF value.
- **Persistence:** `X_LOW`, `X_HIGH`, `LOW_REF` and `HIGH_REF` are bidirectional `AR2` plugs; permanent storage is done by a downstream block (e.g. `INI_AR2/NVS_AR2`). Values read back at boot reappear at `Y_LOW/Y_HIGH` via `LOW_REF.EI1/HIGH_REF.EI1`.
- **Atomic triggers:** `C_LOW/C_HIGH` are plain event inputs, not an `AX` adapter with an edge guard.

Application Scenarios

- Calibrating an analog sensor at two reference points (e.g. zero and end value) with the target values entered on the VT.
- Re-trimming a drifted zero point later without re-calibrating the high point.

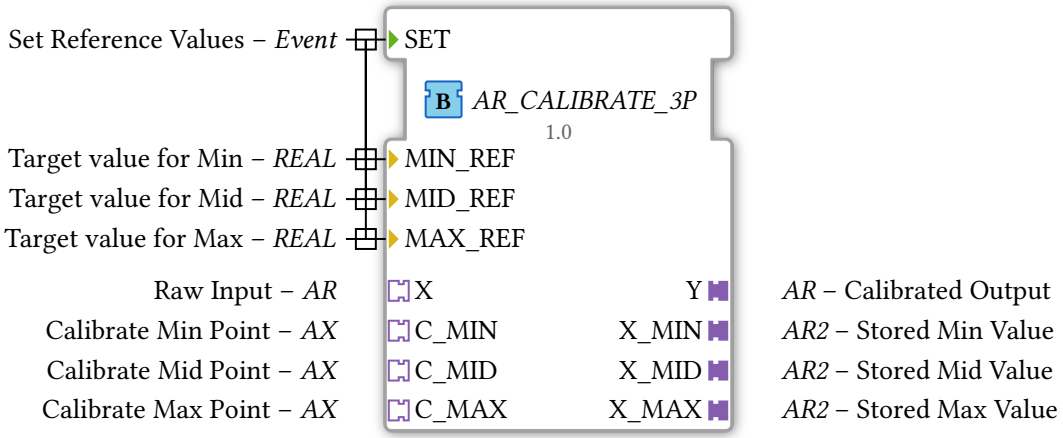
Summary

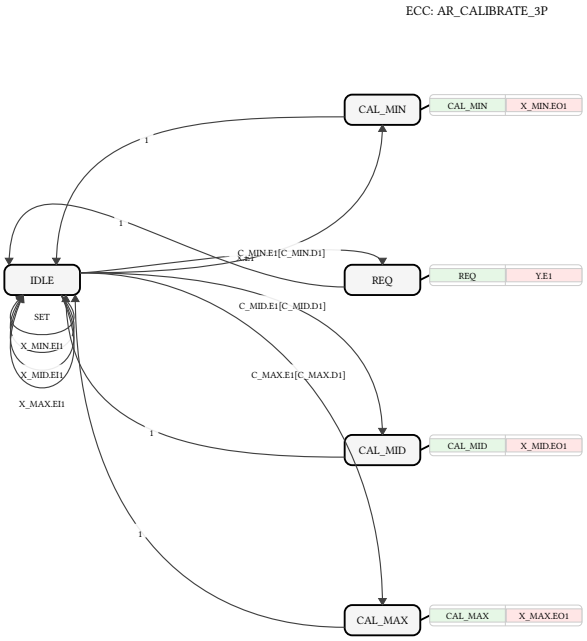
`AR_CALIBRATE_2P_REF` calibrates an `AR` signal using two independently stored raw values and two live-editable, persistable reference targets, and outputs the linearly interpolated (optionally clamped) result via `Y`.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AR_CALIBRATE_3P





• – ○ • – ○ • – ○ •

Introduction

The **AR_CALIBRATE_3P** function block enables 3-point calibration of an analog input signal using adapters. It is specifically designed for joysticks that exhibit center drift and corrects this drift by linearizing between three reference points: minimum, mean, and maximum. The calibration points are saved and can be reset as needed.

Interface Structure

Event Inputs

Name	Type	With	Comment
`SET`	`Event`	`MIN_REF`, `MID_REF`, `MAX_REF`	Sets the reference values for the calibration curve. Does not trigger a calculation, but only sets the target output values.
`C_MIN`	`Event`		Atomic trigger for calibrating the minimum point (reads current raw value from `X`).
`C_MID`	`Event`		Atomic trigger for calibrating the midpoint (reads current raw value from `X`).
`C_MAX`	`Event`		Atomic trigger for calibrating the maximum point (reads current raw value from `X`).

Event Outputs

No explicit event outputs are available. Output is exclusively via the **Y** adapter.

Data Inputs

Name	Data Type	Default Value	Comment
`MIN_REF`	`REAL`	`0.0`	Target value for the smallest input value (Min).

MID_REF	REAL	50.0	Target value for the midpoint (Mid).
MAX_REF	REAL	100.0	Target value for the largest input value (Max).

Data Outputs

No direct data outputs – all outputs are provided via **plugs** (output adapters).

Adapters

Direction	Name	Adapter Type	Comment
Plug (Output)	Y	adapter::types::unidirectional::AC	Calibrated output value (analog value plus event).
Plug (Output)	X_MIN	adapter::types::bidirectional::AR2	Stored minimum value (from the raw value).
Plug (Output)	X_MID	adapter::types::bidirectional::AR2	Stored midpoint value (from the raw value).
Plug (Output)	X_MAX	adapter::types::bidirectional::AR2	Stored maximum value (from the raw value).
Socket (Input)	X	adapter::types::unidirectional::AR	Raw value from the sensor (analog value plus event).

Functionality

The calibration is based on piecewise linear interpolation between three stored raw values (X_MIN, X_MID, X_MAX) and the corresponding reference values (MIN_REF, MID_REF, MAX_REF).

1. **Calibration of the Points:**
   An event at one of the calibration inputs (C_MIN, C_MID, C_MAX) saves the currently applied raw value (X.D1) to the corresponding stored value (X_MIN.DO1, X_MID.DO1, X_MAX.DO1).
2. **Calculation of the Calibrated Value:**
   As soon as an event arrives from the raw value adapter (X.E1), the function block becomes active and executes the **REQ** algorithm. The raw value X.D1 is then linearly mapped:
   - If the raw value is below the stored midpoint X_MID.DI1, the lower branch of the characteristic curve is used:
     `Y.D1 = MIN_REF + (X.D1 – X_MIN.DI1) * (MID_REF – MIN_REF) / (X_MID.DI1 – X_MIN.DI1)`
     If the intervals are invalid (division by zero or negative range), MIN_REF is used.
   - If the raw value is above or equal to X_MID.DI1, the upper branch is calculated:
     `Y.D1 = MID_REF + (X.D1 – X_MID.DI1) * (MAX_REF – MID_REF) / (X_MAX.DI1 – X_MID.DI1)`
     Here too, invalid intervals result in MID_REF being output.
3. **Clipping:**
   The calculated output value is clipped to the interval [MIN_REF, MAX_REF] to ensure physically meaningful results.
4. **Output:**
   The calibrated value is output via the adapter Y (event Y.E1 and data Y.D1).

Technical Features

- **Bidirectional Adapters for Calibration Points:** The stored raw values (X_MIN, X_MID, X_MAX) are bidirectional adapters of type AR2. They can be both written to (during calibration) and read from (during calculation). This makes the calibration points available for subsequent calculations (non-volatile retention across reinitialization or power loss requires an external storage block such as INI/NVS).
- **Atomic Event Triggering:** Calibration of the three points is triggered directly by specific event inputs (C_MIN, C_MID, C_MAX), eliminating intermediate AX adapters and data guards.
- **Protection Against Invalid Intervals:** The algorithms check whether the ranges of the stored raw values are positive. If not (e.g., if the system is not yet calibrated), safe default values are output.

- **No Self-Calibration:** The function block does not store a history – the system must explicitly set the calibration points by triggering the calibration events.

State Overview

State	Description
IDLE	Waiting state. Transitions: For `SET` ⟶ IDLE; for `X_MIN.EI1`, `X_MID.EI1`, `X_MAX.EI1` ⟶ IDLE; for `C_MIN` ⟶ CAL_MIN; for `C_MID` ⟶ CAL_MID; for `C_MAX` ⟶ CAL_MAX; for `X.E1` ⟶ REQ.
REQ	Calculates the calibrated output value. Returns to IDLE immediately after execution.
CAL_MIN	Stores the current raw value as minimum (`X_MIN.DO1 := X.D1`). Returns to IDLE automatically.
CAL_MID	Stores the current raw value as midpoint (`X_MID.DO1 := X.D1`). Returns to IDLE automatically.
CAL_MAX	Stores the current raw value as maximum (`X_MAX.DO1 := X.D1`). Returns to IDLE automatically.

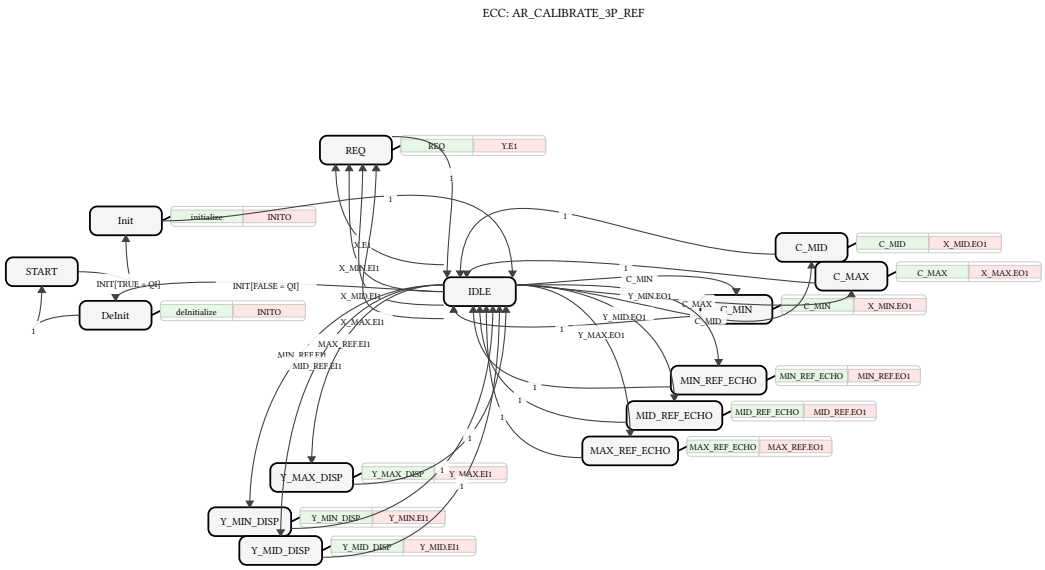
Transition Conditions:

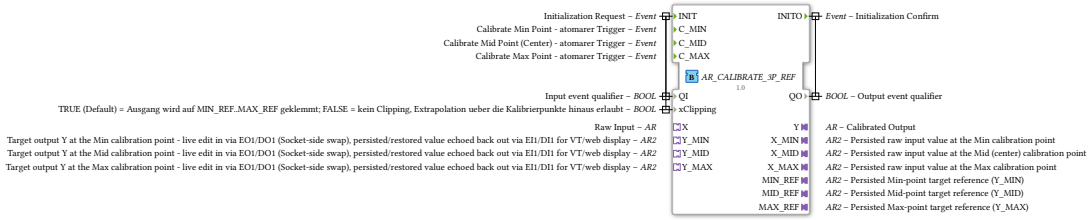
- `X.E1` ⟶ Start of calculation
- `C_MIN` ⟶ Calibration of the minimum point
- `C_MID` ⟶ Calibration of the midpoint
- `C_MAX` ⟶ Calibration of the maximum point
- `SET`, `X_MIN.EI1`, `X_MID.EI1`, `X_MAX.EI1` ⟶ No state change (remains in IDLE)

Application Scenarios

- **Joystick Calibration:** A joystick with an analog output (e.g., 0–10 V) exhibits component-related deviations at center and end positions. The operator moves the joystick to the three positions (Min, Mid, Max) and triggers the calibration events via pushbuttons. `Y` then delivers a linearized value normalized to target limits.
- **Analog Potentiometer:** A sliding potentiometer affected by wear can be corrected via 3-point calibration.

AR_CALIBRATE_3P_REF





• – ○ • – ○ • – ○ •

Introduction

`AR_CALIBRATE_3P_REF` is an adapter-based three-point calibration for joysticks and sensors with a centre position (Min, Mid, Max), in free order like AR_CALIBRATE_3P. The reference targets (`Y_MIN`/`Y_MID`/`Y_MAX`) are **live `AR2` sockets** and are persisted via three additional `AR2` plugs (`MIN_REF`/`MID_REF`/`MAX_REF`). The same live-echo pattern as in `AR_CALIBRATE_SQ_REF` and AR_CALIBRATE_2P_REF applies.

Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation request. Associated with `QI`, `xClipping`.
- **C_MIN**: calibrate the min point, atomic trigger.
- **C_MID**: calibrate the mid point (centre), atomic trigger.
- **C_MAX**: calibrate the max point, atomic trigger.

Event Outputs

- **INITO** (`EInit`): initialisation confirm. Associated with `QO`.

Data Inputs

- **QI** (`BOOL`): event input qualifier
- **xClipping** (`BOOL`, default `TRUE`): `TRUE` = output is clamped to `MIN_REF..MAX_REF`; `FALSE` = no clipping, extrapolation beyond the calibration points allowed

Data Outputs

- **QO** (`BOOL`): event output qualifier

Adapters

Plugs:

- **Y** (`adapter::types::unidirectional::AR`): calibrated output
- **X_MIN**, **X_MID**, **X_MAX** (`adapter::types::bidirectional::AR2`): persisted raw input value at the min/mid/max calibration point
- **MIN_REF**, **MID_REF**, **MAX_REF** (`adapter::types::bidirectional::AR2`): persisted target reference (`Y_MIN`/`Y_MID`/`Y_MAX`)

Sockets:

- **X** (`adapter::types::unidirectional::AR`): raw input
- **Y_MIN**, **Y_MID**, **Y_MAX** (`adapter::types::bidirectional::AR2`): target output `Y` at the min/mid/max point. Live edit in via `EO1`/`DO1` (socket-side swap); the persisted/restored value is echoed back out via `EI1`/`DI1` for VT/web display.

Functionality

The block starts in state `START`; `INIT` with `QI = TRUE` leads via `Init` to `IDLE` (`QO := QI`), `INIT` with `QI = FALSE` leads via `DeInit` back to `START` (`QO := FALSE`). From `IDLE` the following action states are reachable, each returning immediately to `IDLE`:

- **REQ** (on `X.E1` or an event `X_MIN.EI1`/`X_MID.EI1`/`X_MAX.EI1`): calculates `Y` and fires `Y.E1`
- **C_MIN**/**C_MID**/**C_MAX**: store the raw input `X.D1` in `X_MIN.DO1`/`X_MID.DO1`/`X_MAX.DO1` and fire `X_MIN.EO1`/`X_MID.EO1`/`X_MAX.EO1`
- **MIN_REF_ECHO**/**MID_REF_ECHO**/**MAX_REF_ECHO** (on `Y_MIN.EO1`/`Y_MID.EO1`/`Y_MAX.EO1`): transfer the live reference `Y_xxx.DO1` to `xxx_REF.DO1` (persistence)

- **Y_MIN_DISP/Y_MID_DISP/Y_MAX_DISP** (on MIN_REF.EI1/MID_REF.EI1/MAX_REF.EI1): echo the persisted/restored reference back to Y_xxx.DI1 (after boot and after every change)

Formula: piecewise linear interpolation between (X_MIN, MIN_REF), (X_MID, MID_REF) and (X_MAX, MAX_REF), clipped to MIN_REF..MAX_REF unless xClipping = FALSE.

Partial calibration (before all three points are set): each point is tracked internally (xMinCal/xMidCal/xMaxCal, latched TRUE once its raw value differs from the INI default 0.0; stays correctly set across restarts because the boot restore returns the persisted value). Until all three are calibrated, REQ falls back cleanly instead of pivoting on a still-uncalibrated X_MID.DI1 (which would sit at 0.0 and make the MIN branch permanently unreachable):

- Only one point calibrated: Y = that point's own REF value (no interpolation possible)
- Two points calibrated: two-point line between exactly those two, the third is ignored
- All three calibrated: full piecewise interpolation
- None calibrated: Y := MID_REF

Clipping only takes place if xClipping is set and both MIN and MAX are calibrated; otherwise the still-uncalibrated bound at its INI default 0.0 would corrupt the partial-calibration fallback. MID is irrelevant for this.

Calibration (three-point, free order):

1. **Min (C_MIN)**: move to minimum, set Y_MIN, trigger C_MIN ⟶ X_MIN := X
2. **Mid (C_MID)**: move to centre, set Y_MID, trigger C_MID ⟶ X_MID := X
3. **Max (C_MAX)**: move to maximum, set Y_MAX, trigger C_MAX ⟶ X_MAX := X

Technical Details

- **Free order** as with AR_CALIBRATE_3P: each point is stored independently and can be recalibrated on its own (e.g. only re-trimming the centre), with no data dependency between the calibration actions.
- **References as live sockets:** Y_MIN/Y_MID/Y_MAX are bidirectional AR2 sockets fed from an external VT/web reader, mirrored outward via the AR2 plugs MIN_REF/MID_REF/MAX_REF for persistence (INI_AR2/NVS_AR2) and echoed back via MIN_REF.EI1/MID_REF.EI1/MAX_REF.EI1, so that a value restored after boot appears on screen without the operator re-entering it.
- Degenerate points (X_MID <= X_MIN or X_MAX <= X_MID) yield the respective REF value instead of a division by zero.

State Overview

```
START --INIT[QI]--> Init --1--> IDLE
IDLE --INIT[!QI]--> DeInit --1--> START
IDLE --X.E1 / X_xxx.EI1--> REQ --1--> IDLE
IDLE --C_MIN / C_MID / C_MAX--> C_xxx --1--> IDLE
IDLE --Y_xxx.EO1--> xxx_REF_ECHO --1--> IDLE
IDLE --xxx_REF.EI1--> Y_xxx_DISP --1--> IDLE
```

Application Scenarios

- Steering angle sensor: Min = full lock left, Mid = straight ahead, Max = full lock right. MIN_REF/MID_REF/MAX_REF hold the desired calibrated angle values (e.g. −30.0 / 0.0 / 30.0 degrees), X_MIN/X_MID/X_MAX the raw sensor values actually measured.
- Joysticks with a centre position and persisted reference values editable on the VT

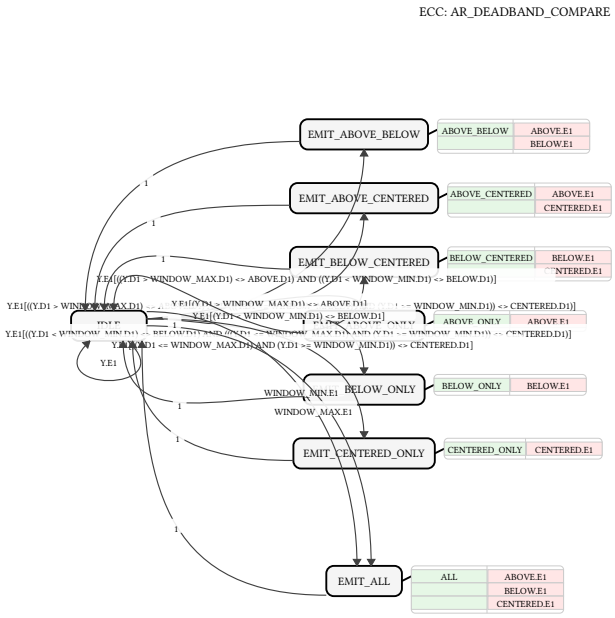
⚖️ Comparison with Similar Building Blocks

- AR_CALIBRATE_3P: three-point calibration without *_REF sockets
- AR_CALIBRATE_2P_REF: two-point variant with the same live references

Conclusion

AR_CALIBRATE_3P_REF calibrates three points in free order, keeps raw values and references persisted via AR2 adapters and falls back cleanly on partial calibration.

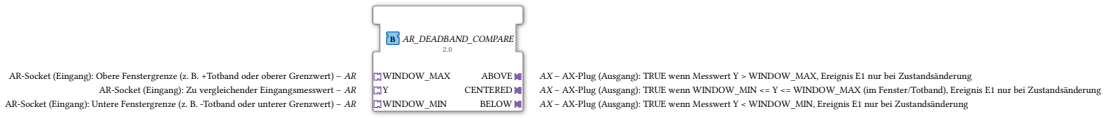
AR_DEADBAND_COMPARE



• — ○ • — ○ • — ○ •

Introduction

`AR_DEADBAND_COMPARE` compares a `REAL` measured value `Y` (via an `AR` socket) against a window `[WINDOW_MIN, WINDOW_MAX]` and reports the result via three `AX` plugs: `ABOVE` (`Y > WINDOW_MAX`), `BELOW` (`Y < WINDOW_MIN`) and `CENTERED` (in between). For a symmetric deadband around zero, set `WINDOW_MIN = -deadband` and `WINDOW_MAX = +deadband` (e.g. steering angle centering); the block can equally be used for arbitrary windows (e.g. colour limits).



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets (`Y.E1`, `WINDOW_MIN.E1`, `WINDOW_MAX.E1`).

Event Outputs

No direct event outputs - events are sent via the adapter plugs (`ABOVE.E1`, `BELOW.E1`, `CENTERED.E1`).

Data Inputs

No direct data inputs (values arrive via the sockets, data port `D1`).

Data Outputs

No direct data outputs (values are output via the plugs, data port `D1`).

Adapters

Sockets (inputs):

Adapter	Type	Description
`Y`	`adapter::types::unidirectional::AR`	Input measured value to be compared

WINDOW_MIN	adapter::types::unidirectional::AR	Lower window limit (e.g. -deadband or lower threshold)
WINDOW_MAX	adapter::types::unidirectional::AR	Upper window limit (e.g. +deadband or upper threshold)

Plugs (outputs):

Adapter	Type	Description
ABOVE	adapter::types::unidirectional::AX	TRUE if Y > WINDOW_MAX; event E1 only on state change
CENTERED	adapter::types::unidirectional::AX	TRUE if WINDOW_MIN <= Y <= WINDOW_MAX (inside window/deadband); event E1 only on state change
BELOW	adapter::types::unidirectional::AX	TRUE if Y < WINDOW_MIN; event E1 only on state change

How it works

The block is a basic FB with an idle state IDLE and seven emit states, each of which returns to IDLE immediately (condition 1) after its action.

- **Y change (Y.E1)**: The transitions compare the newly computed value of ABOVE, BELOW and CENTERED with the value currently stored in the respective plug (<>). Only the outputs whose boolean value actually changes are recomputed and send their E1:
  - EMIT_ABOVE_BELOW: ABOVE and BELOW change
  - EMIT_ABOVE_CENTERED: ABOVE and CENTERED change
  - EMIT_BELOW_CENTERED: BELOW and CENTERED change
  - EMIT_ABOVE_ONLY, EMIT_BELOW_ONLY, EMIT_CENTERED_ONLY: only the respective output changes
  - If none of the values changes, Y.E1 leads directly IDLE ⟶ IDLE with no output event.
- **Window limit change (WINDOW_MIN.E1 or WINDOW_MAX.E1)**: State EMIT_ALL recomputes all three outputs and sends ABOVE.E1, BELOW.E1 and CENTERED.E1 (in practice usually only once at start-up).

Calculations (ST algorithms):

```
ABOVE.D1    := Y.D1 > WINDOW_MAX.D1;
BELOW.D1    := Y.D1 < WINDOW_MIN.D1;
CENTERED.D1 := (Y.D1 <= WINDOW_MAX.D1) AND (Y.D1 >= WINDOW_MIN.D1);
```

Technical Details

- **Event efficiency**: On a Y change, each plug fires only if its own value actually changes (not all three on every update)
- **Window limits**: On a change of WINDOW_MIN/WINDOW_MAX, all three outputs are always set again for simplicity
- **Limit values**: The limits themselves belong to CENTERED (<=/>=); ABOVE and BELOW are strict (>/<)
- A fixed value for a limit can be supplied via adapter::types::unidirectional::AR::initval::initval_AR
- Version history: 1.0 (initial, axle steering centre automation), 1.1 (DEADBAND as AR socket), 2.0 (generalised to an arbitrary window WINDOW_MIN/WINDOW_MAX)

State Overview

State	Description	Action
IDLE	Waits for events on Y.E1, WINDOW_MIN.E1 or WINDOW_MAX.E1	-
EMIT_ABOVE_BELOW	Update ABOVE and BELOW	ABOVE.E1, BELOW.E1
EMIT_ABOVE_CENTERED	Update ABOVE and CENTERED	ABOVE.E1, CENTERED.E1

EMIT_BELOW_CENTERED	Update BELOW and CENTERED	`BELOW.E1`, `CENTERED.E1`
EMIT_ABOVE_ONLY	Update ABOVE only	`ABOVE.E1`
EMIT_BELOW_ONLY	Update BELOW only	`BELOW.E1`
EMIT_CENTERED_ONLY	Update CENTERED only	`CENTERED.E1`
EMIT_ALL	Update all three (window limits changed)	`ABOVE.E1`, `BELOW.E1`, `CENTERED.E1`

Application Scenarios

- **Steering angle centering**: Checking whether a steering sensor lies within the deadband around 0° (`WINDOW_MIN = -deadband`, `WINDOW_MAX = +deadband`)
- **Window/hysteresis monitoring**: Evaluating limit or colour ranges (e.g. green/red in visualisations)

Summary

`AR_DEADBAND_COMPARE` provides the three states above, below and inside a window for an `AR` measured value as `AX` signals and generates events only on an actual state change.

Splitting

Signed Splitting Function Blocks

This library contains function blocks for splitting signed signals (`SINT`, `INT`, `DINT`, `LINT`, `REAL`, `LREAL`) into two positive magnitude components (`NEG_MAG` and `POS_MAG`).

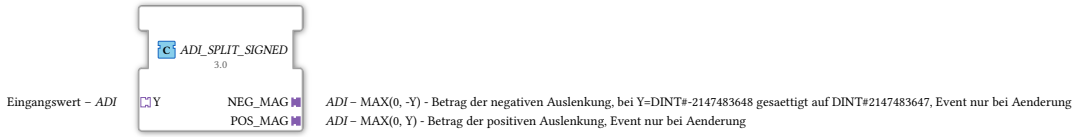
Basic FBs (Pure Calculation)

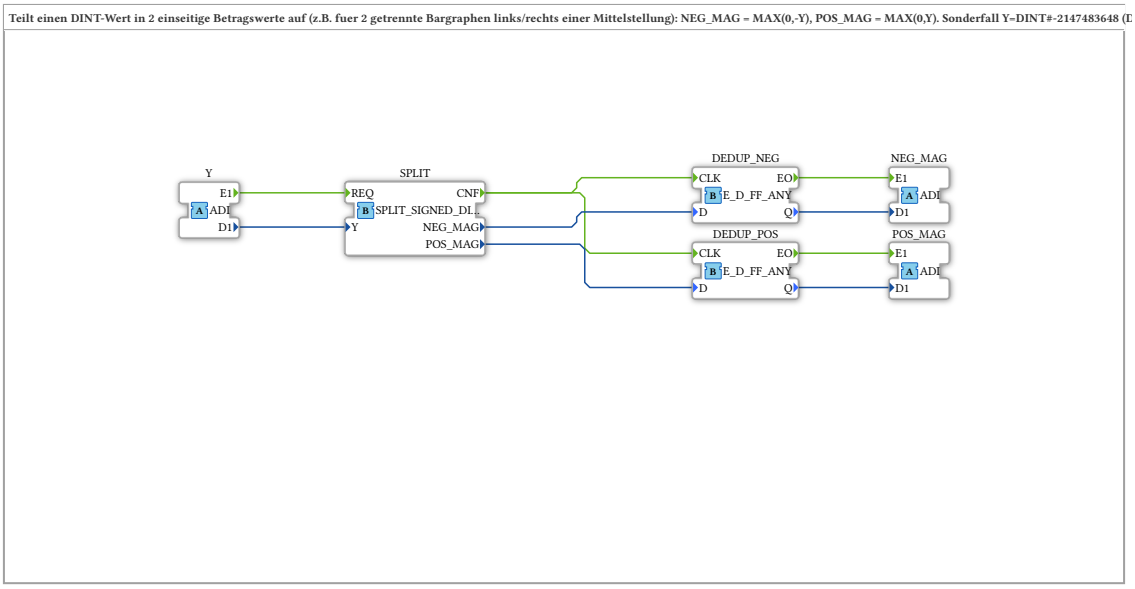
- SPLIT_SIGNED_SINT – Splitting for `SINT`
- SPLIT_SIGNED_INT – Splitting for `INT`
- SPLIT_SIGNED_DINT – Splitting for `DINT`
- SPLIT_SIGNED_LINT – Splitting for `LINT`
- SPLIT_SIGNED_REAL – Splitting for `REAL`
- SPLIT_SIGNED_LREAL – Splitting for `LREAL`

Adapter Wrapper Composite FBs (With D-Flip-Flop Event Filtering)

- AS_SPLIT_SIGNED – Adapter wrapper for `SINT` (`AS`)
- AI_SPLIT_SIGNED – Adapter wrapper for `INT` (`AI`)
- ADI_SPLIT_SIGNED – Adapter wrapper for `DINT` (`ADI`)
- ALI_SPLIT_SIGNED – Adapter wrapper for `LINT` (`ALI`)
- AR_SPLIT_SIGNED – Adapter wrapper for `REAL` (`AR`)
- ALR_SPLIT_SIGNED – Adapter wrapper for `LREAL` (`ALR`)

ADI_SPLIT_SIGNED





• — ○ • — ○ • — ○ •

Introduction

The **ADI_SPLIT_SIGNED** function block is an adapter wrapper (Composite FB) that splits a signed `DINT` measurement value into two single-sided magnitude components using unidirectional adapters (`adapter::types::unidirectional::ADI`):

- `NEG_MAG`: Magnitude of negative deflection (`MAX(0, -Y)`)
- `POS_MAG`: Magnitude of positive deflection (`MAX(0, Y)`)

The block encapsulates calculation FB **SPLIT_SIGNED_DINT** and two **E_D_FF_ANY** D-Flip-Flop blocks to emit the initial event on the first `CLK` and thereafter emit events on plugs `NEG_MAG` and `POS_MAG` **only when the new input value differs from the stored output state** (`D <> Q`).

Interface Structure

Adapter Sockets (Input)

Name	Type	Comment
`Y`	`adapter::types::unidirectional::ADI`	Signed input value (ADI adapter socket)

Adapter Plugs (Output)

Name	Type	Comment
`NEG_MAG`	`adapter::types::unidirectional::ADI`	Negative magnitude (`MAX(0, -Y)`), initial event on 1st call, thereafter on change only
`POS_MAG`	`adapter::types::unidirectional::ADI`	Positive magnitude (`MAX(0, Y)`), initial event on 1st call, thereafter on change only

Functionality

Internally, the composite network consists of three components:

1. **SPLIT (SPLIT_SIGNED_DINT)**: Computes `NEG_MAG` and `POS_MAG` from `Y.D1` upon every `Y.E1` event.
2. **DEDUP_NEG (E_D_FF_ANY)**: An event-driven D-Flip-Flop that emits the initial output event upon the first `CLK` event, and thereafter forwards `CLK` to `EO` and updates `Q := D` only when the new data value `D` differs from the current output state `Q` (`D <> Q`).
3. **DEDUP_POS (E_D_FF_ANY)**: A second D-Flip-Flop of the same type that similarly emits the initial output event on the first call and subsequently emits event `POS_MAG.E1` only when the positive magnitude value changes.

```
Y (ADI Adapter Socket)
 ├──> SPLIT (SPLIT_SIGNED_DINT)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (ADI Adapter
Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (ADI Adapter
Plug)
```

Technical Features

- **Adapter-Driven Architecture:** Fully compatible with unidirectional adapter family `adapter::types::unidirectional::ADI`.
- **D-Flip-Flop Event Filtering:** The `E_D_FF_ANY` blocks guarantee by definition that the initial output event is emitted on the first `CLK`, and subsequent output events `EO` are generated only when the new input `D` differs from the stored output `Q`.
- **Overflow Safety:** Inherits saturation logic from `SPLIT_SIGNED_DINT` for two's complement integer limits.

Application Scenarios

- Connecting bipolar sensors (e.g. steering angle sensor, tilt sensor, joystick) to dual VT bargraph displays.
- Adapter-driven actuation of hydraulic cylinders (extend/retract).

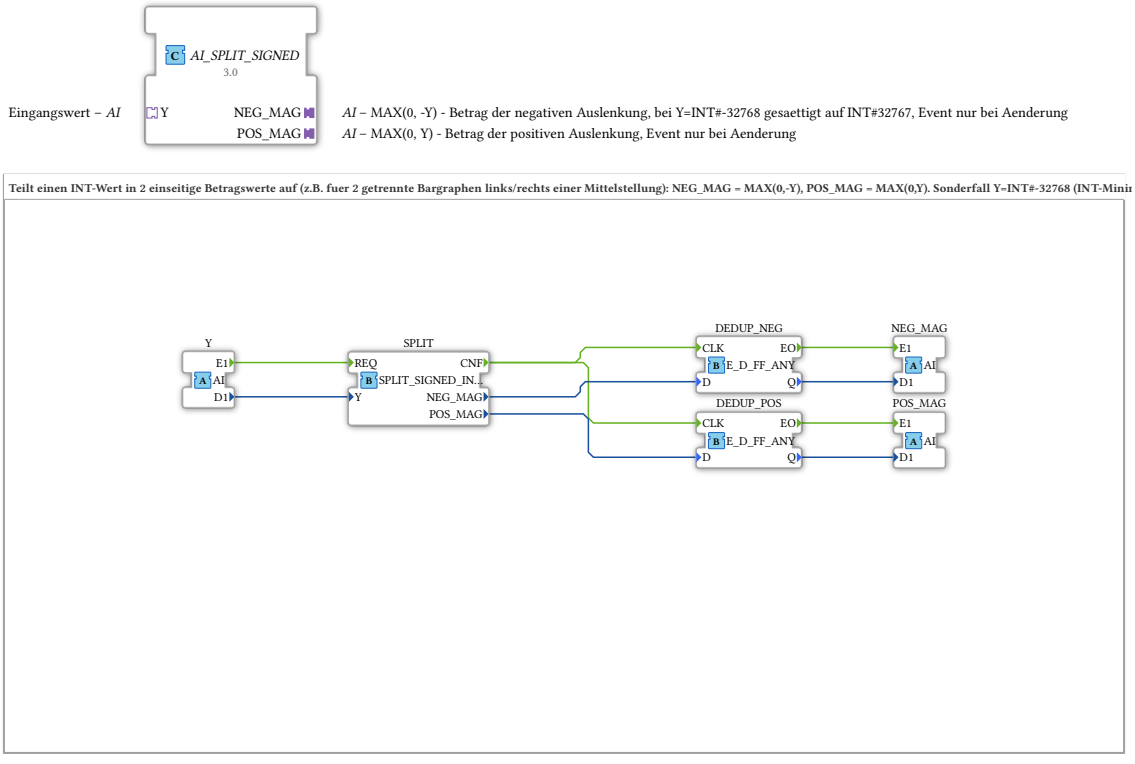
Comparison with Similar Blocks

- **ADI_SPLIT_SIGNED**: Composite FB with adapter interfaces and D-Flip-Flop event filtering.
- **SPLIT_SIGNED_DINT**: Underlying Basic FB for pure signal computation without adapters.

Conclusion

ADI_SPLIT_SIGNED delivers an elegant, adapter-driven, and event-efficient solution for signed signal splitting in IEC 61499 applications.

AI_SPLIT_SIGNED



Introduction

The **AI_SPLIT_SIGNED** function block is an adapter wrapper (Composite FB) that splits a signed `INT` measurement value into two single-sided magnitude components using unidirectional adapters (`adapter::types::unidirectional::AI`):

- `NEG_MAG`: Magnitude of negative deflection (`MAX(0, -Y)`)
- `POS_MAG`: Magnitude of positive deflection (`MAX(0, Y)`)

The block encapsulates calculation FB **SPLIT_SIGNED_INT** and two **E_D_FF_ANY** D-Flip-Flop blocks to emit the initial event on the first `CLK` and thereafter emit events on plugs `NEG_MAG` and `POS_MAG` **only when the new input value differs from the stored output state** (`D <> Q`).

Interface Structure

Adapter Sockets (Input)

Name	Type	Comment
`Y`	`adapter::types::unidirectional::AI`	Signed input value (AI adapter socket)

Adapter Plugs (Output)

Name	Type	Comment
`NEG_MAG`	`adapter::types::unidirectional::AI`	Negative magnitude (`MAX(0, -Y)`), initial event on 1st call, thereafter on change only
`POS_MAG`	`adapter::types::unidirectional::AI`	Positive magnitude (`MAX(0, Y)`), initial event on 1st call, thereafter on change only

Functionality

Internally, the composite network consists of three components:

1. **SPLIT (SPLIT_SIGNED_INT)**: Computes `NEG_MAG` and `POS_MAG` from `Y.D1` upon every `Y.E1` event.
2. **DEDUP_NEG (E_D_FF_ANY)**: An event-driven D-Flip-Flop that emits the initial output event upon the first `CLK` event, and thereafter forwards `CLK` to `EO` and updates `Q := D` only when the new data value `D` differs from the current output state `Q` (`D <> Q`).
3. **DEDUP_POS (E_D_FF_ANY)**: A second D-Flip-Flop of the same type that similarly emits the initial output event on the first call and subsequently emits event `POS_MAG.E1` only when the positive magnitude value changes.

```
Y (AI Adapter Socket)
 ├──> SPLIT (SPLIT_SIGNED_INT)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (AI Adapter Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (AI Adapter Plug)
```

Technical Features

- **Adapter-Driven Architecture:** Fully compatible with unidirectional adapter family `adapter::types::unidirectional::AI`.
- **D-Flip-Flop Event Filtering:** The `E_D_FF_ANY` blocks guarantee by definition that the initial output event is emitted on the first `CLK`, and subsequent output events `EO` are generated only when the new input `D` differs from the stored output `Q`.
- **Overflow Safety:** Inherits saturation logic from `SPLIT_SIGNED_INT` for two's complement integer limits.

Application Scenarios

- Connecting bipolar sensors (e.g. steering angle sensor, tilt sensor, joystick) to dual VT bargraph displays.
- Adapter-driven actuation of hydraulic cylinders (extend/retract).

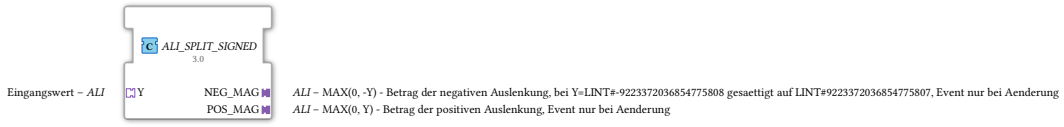
Comparison with Similar Blocks

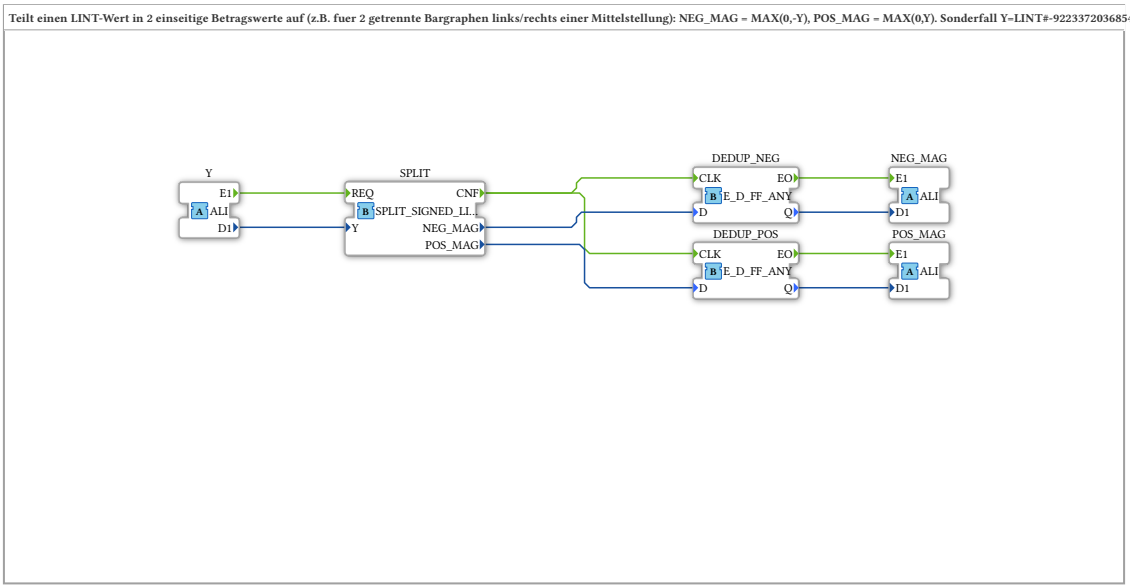
- **AI_SPLIT_SIGNED**: Composite FB with adapter interfaces and D-Flip-Flop event filtering.
- **SPLIT_SIGNED_INT**: Underlying Basic FB for pure signal computation without adapters.

Conclusion

AI_SPLIT_SIGNED delivers an elegant, adapter-driven, and event-efficient solution for signed signal splitting in IEC 61499 applications.

ALI_SPLIT_SIGNED





• – ○ • – ○ • – ○ •

Introduction

The **ALI_SPLIT_SIGNED** function block is an adapter wrapper (Composite FB) that splits a signed `LINT` measurement value into two single-sided magnitude components using unidirectional adapters (`adapter::types::unidirectional::ALI`):

- `NEG_MAG`: Magnitude of negative deflection (`MAX(0, -Y)`)
- `POS_MAG`: Magnitude of positive deflection (`MAX(0, Y)`)

The block encapsulates calculation FB **SPLIT_SIGNED_LINT** and two **E_D_FF_ANY** D-Flip-Flop blocks to emit the initial event on the first `CLK` and thereafter emit events on plugs `NEG_MAG` and `POS_MAG` **only when the new input value differs from the stored output state** (`D <> Q`).

Interface Structure

Adapter Sockets (Input)

Name	Type	Comment
`Y`	`adapter::types::unidirectional::ALI`	Signed input value (ALI adapter socket)

Adapter Plugs (Output)

Name	Type	Comment
`NEG_MAG`	`adapter::types::unidirectional::ALI`	Negative magnitude (`MAX(0, -Y)`), initial event on 1st call, thereafter on change only
`POS_MAG`	`adapter::types::unidirectional::ALI`	Positive magnitude (`MAX(0, Y)`), initial event on 1st call, thereafter on change only

Functionality

Internally, the composite network consists of three components:

1. **SPLIT (SPLIT_SIGNED_LINT)**: Computes `NEG_MAG` and `POS_MAG` from `Y.D1` upon every `Y.E1` event.
2. **DEDUP_NEG (E_D_FF_ANY)**: An event-driven D-Flip-Flop that emits the initial output event upon the first `CLK` event, and thereafter forwards `CLK` to `EO` and updates `Q := D` only when the new data value `D` differs from the current output state `Q` (`D <> Q`).
3. **DEDUP_POS (E_D_FF_ANY)**: A second D-Flip-Flop of the same type that similarly emits the initial output event on the first call and subsequently emits event `POS_MAG.E1` only when the positive magnitude value changes.

```
Y (ALI Adapter Socket)
 ├──> SPLIT (SPLIT_SIGNED_LINT)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (ALI Adapter
Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (ALI Adapter
Plug)
```

Technical Features

- **Adapter-Driven Architecture:** Fully compatible with unidirectional adapter family `adapter::types::unidirectional::ALI`.
- **D-Flip-Flop Event Filtering:** The `E_D_FF_ANY` blocks guarantee by definition that the initial output event is emitted on the first `CLK`, and subsequent output events `EO` are generated only when the new input `D` differs from the stored output `Q`.
- **Overflow Safety:** Inherits saturation logic from `SPLIT_SIGNED_LINT` for two's complement integer limits.

Application Scenarios

- Connecting bipolar sensors (e.g. steering angle sensor, tilt sensor, joystick) to dual VT bargraph displays.
- Adapter-driven actuation of hydraulic cylinders (extend/retract).

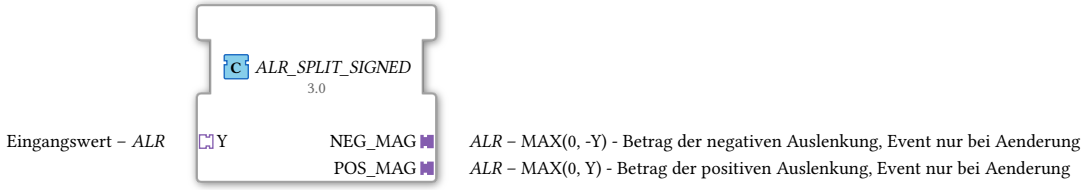
Comparison with Similar Blocks

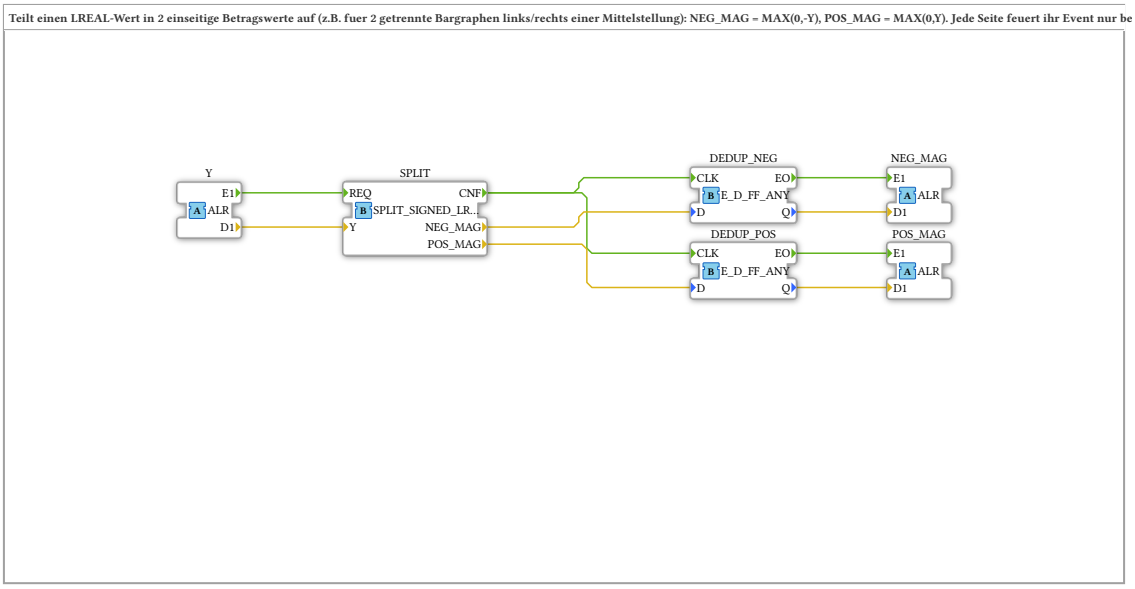
- **ALI_SPLIT_SIGNED**: Composite FB with adapter interfaces and D-Flip-Flop event filtering.
- **SPLIT_SIGNED_LINT**: Underlying Basic FB for pure signal computation without adapters.

Conclusion

ALI_SPLIT_SIGNED delivers an elegant, adapter-driven, and event-efficient solution for signed signal splitting in IEC 61499 applications.

ALR_SPLIT_SIGNED





• — ○ • — ○ • — ○ •

Introduction

The **ALR_SPLIT_SIGNED** function block is an adapter wrapper (Composite FB) that splits a signed `LREAL` measurement value into two single-sided magnitude components using unidirectional adapters (`adapter::types::unidirectional::ALR`):

- `NEG_MAG`: Magnitude of negative deflection (`MAX(0, -Y)`)
- `POS_MAG`: Magnitude of positive deflection (`MAX(0, Y)`)

The block encapsulates calculation FB **SPLIT_SIGNED_LREAL** and two **E_D_FF_ANY** D-Flip-Flop blocks to emit the initial event on the first `CLK` and thereafter emit events on plugs `NEG_MAG` and `POS_MAG` **only when the new input value differs from the stored output state** (`D <> Q`).

Interface Structure

Adapter Sockets (Input)

Name	Type	Comment
`Y`	`adapter::types::unidirectional::ALR`	Signed input value (ALR adapter socket)

Adapter Plugs (Output)

Name	Type	Comment
`NEG_MAG`	`adapter::types::unidirectional::ALR`	Negative magnitude (`MAX(0, -Y)`), initial event on 1st call, thereafter on change only
`POS_MAG`	`adapter::types::unidirectional::ALR`	Positive magnitude (`MAX(0, Y)`), initial event on 1st call, thereafter on change only

Functionality

Internally, the composite network consists of three components:

1. **SPLIT (SPLIT_SIGNED_LREAL)**: Computes `NEG_MAG` and `POS_MAG` from `Y.D1` upon every `Y.E1` event.
2. **DEDUP_NEG (E_D_FF_ANY)**: An event-driven D-Flip-Flop that emits the initial output event upon the first `CLK` event, and thereafter forwards `CLK` to `EO` and updates `Q := D` only when the new data value `D` differs from the current output state `Q` (`D <> Q`).
3. **DEDUP_POS (E_D_FF_ANY)**: A second D-Flip-Flop of the same type that similarly emits the initial output event on the first call and subsequently emits event `POS_MAG.E1` only when the positive magnitude value changes.

```
Y (ALR Adapter Socket)
 ├──> SPLIT (SPLIT_SIGNED_LREAL)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (ALR Adapter
Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (ALR Adapter
Plug)
```

Technical Features

- **Adapter-Driven Architecture:** Fully compatible with unidirectional adapter family `adapter::types::unidirectional::ALR`.
- **D-Flip-Flop Event Filtering:** The `E_D_FF_ANY` blocks guarantee by definition that the initial output event is emitted on the first `CLK`, and subsequent output events `EO` are generated only when the new input `D` differs from the stored output `Q`.
- **Overflow Safety:** Inherits saturation logic from `SPLIT_SIGNED_LREAL` for two's complement integer limits.

Application Scenarios

- Connecting bipolar sensors (e.g. steering angle sensor, tilt sensor, joystick) to dual VT bargraph displays.
- Adapter-driven actuation of hydraulic cylinders (extend/retract).

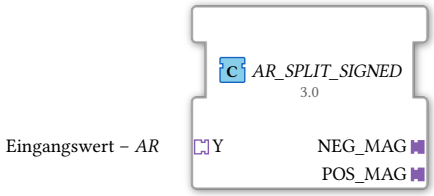
Comparison with Similar Blocks

- **ALR_SPLIT_SIGNED**: Composite FB with adapter interfaces and D-Flip-Flop event filtering.
- **SPLIT_SIGNED_LREAL**: Underlying Basic FB for pure signal computation without adapters.

Conclusion

ALR_SPLIT_SIGNED delivers an elegant, adapter-driven, and event-efficient solution for signed signal splitting in IEC 61499 applications.

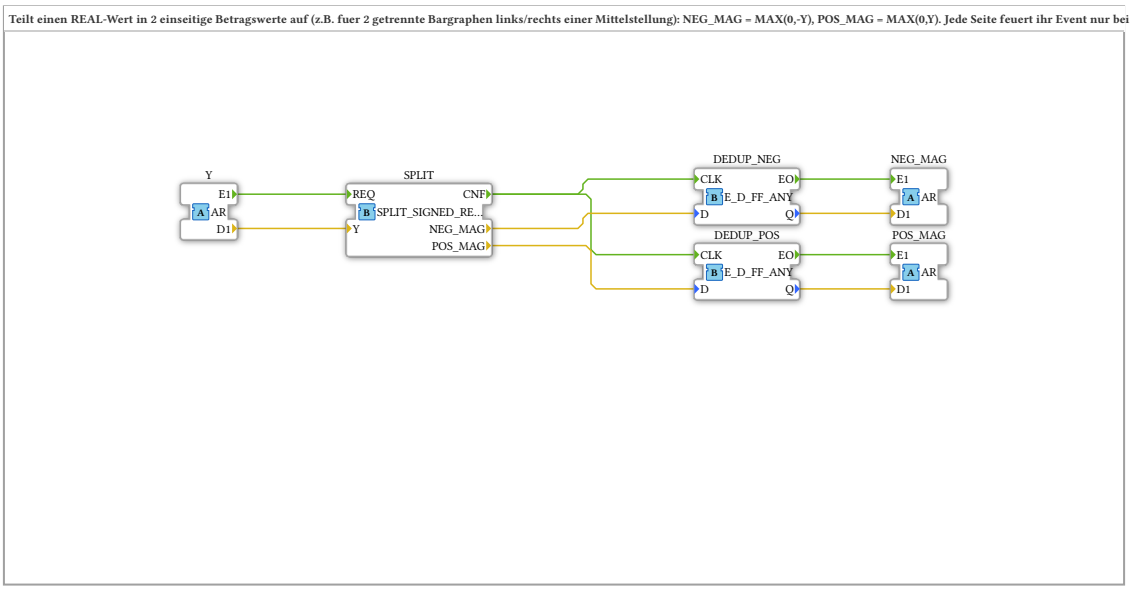
AR_SPLIT_SIGNED



Eingangswert – *AR*

AR – MAX(0, -Y) - Betrag der negativen Auslenkung, Event nur bei Aenderung

AR – MAX(0, Y) - Betrag der positiven Auslenkung, Event nur bei Aenderung



• – ○ • – ○ • – ○ •

Introduction

The **AR_SPLIT_SIGNED** function block is an adapter wrapper (Composite FB) that splits a signed `REAL` measurement value into two single-sided magnitude components using unidirectional adapters (`adapter::types::unidirectional::AR`):

- `NEG_MAG`: Magnitude of negative deflection (`MAX(0, -Y)`)
- `POS_MAG`: Magnitude of positive deflection (`MAX(0, Y)`)

The block encapsulates calculation FB **SPLIT_SIGNED_REAL** and two **E_D_FF_ANY** D-Flip-Flop blocks to emit the initial event on the first `CLK` and thereafter emit events on plugs `NEG_MAG` and `POS_MAG` **only when the new input value differs from the stored output state** (`D <> Q`).

Interface Structure

Adapter Sockets (Input)

Name	Type	Comment
`Y`	`adapter::types::unidirectional::AR`	Signed input value (AR adapter socket)

Adapter Plugs (Output)

Name	Type	Comment
`NEG_MAG`	`adapter::types::unidirectional::AR`	Negative magnitude (`MAX(0, -Y)`), initial event on 1st call, thereafter on change only
`POS_MAG`	`adapter::types::unidirectional::AR`	Positive magnitude (`MAX(0, Y)`), initial event on 1st call, thereafter on change only

Functionality

Internally, the composite network consists of three components:

1. **SPLIT (SPLIT_SIGNED_REAL)**: Computes `NEG_MAG` and `POS_MAG` from `Y.D1` upon every `Y.E1` event.
2. **DEDUP_NEG (E_D_FF_ANY)**: An event-driven D-Flip-Flop that emits the initial output event upon the first `CLK` event, and thereafter forwards `CLK` to `EO` and updates `Q := D` only when the new data value `D` differs from the current output state `Q` (`D <> Q`).
3. **DEDUP_POS (E_D_FF_ANY)**: A second D-Flip-Flop of the same type that similarly emits the initial output event on the first call and subsequently emits event `POS_MAG.E1` only when the positive magnitude value changes.

```
Y (AR Adapter Socket)
 ├──> SPLIT (SPLIT_SIGNED_REAL)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (AR Adapter Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (AR Adapter Plug)
```

Technical Features

- **Adapter-Driven Architecture:** Fully compatible with unidirectional adapter family `adapter::types::unidirectional::AR`.
- **D-Flip-Flop Event Filtering:** The `E_D_FF_ANY` blocks guarantee by definition that the initial output event is emitted on the first `CLK`, and subsequent output events `EO` are generated only when the new input `D` differs from the stored output `Q`.
- **Overflow Safety:** Inherits saturation logic from `SPLIT_SIGNED_REAL` for two's complement integer limits.

Application Scenarios

- Connecting bipolar sensors (e.g. steering angle sensor, tilt sensor, joystick) to dual VT bargraph displays.
- Adapter-driven actuation of hydraulic cylinders (extend/retract).

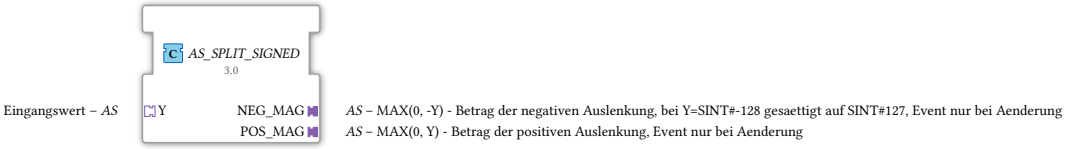
Comparison with Similar Blocks

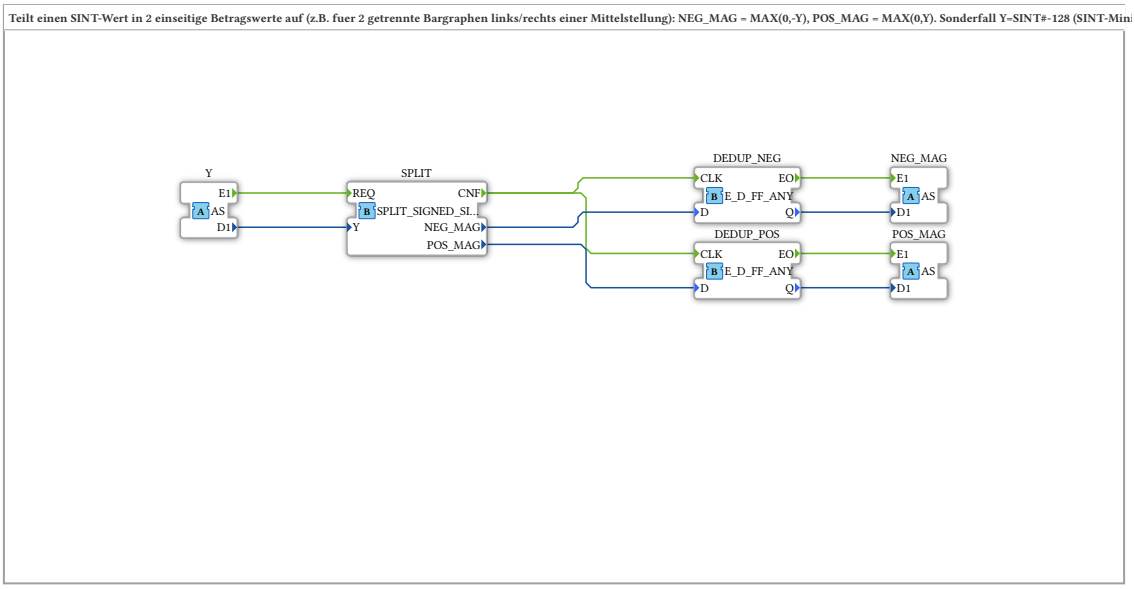
- **AR_SPLIT_SIGNED**: Composite FB with adapter interfaces and D-Flip-Flop event filtering.
- **SPLIT_SIGNED_REAL**: Underlying Basic FB for pure signal computation without adapters.

Conclusion

AR_SPLIT_SIGNED delivers an elegant, adapter-driven, and event-efficient solution for signed signal splitting in IEC 61499 applications.

AS_SPLIT_SIGNED





• – ○ • – ○ • – ○ •

Introduction

The **AS_SPLIT_SIGNED** function block is an adapter wrapper (Composite FB) that splits a signed `SINT` measurement value into two single-sided magnitude components using unidirectional adapters (`adapter::types::unidirectional::AS`):

- `NEG_MAG`: Magnitude of negative deflection (`MAX(0, -Y)`)
- `POS_MAG`: Magnitude of positive deflection (`MAX(0, Y)`)

The block encapsulates calculation FB **SPLIT_SIGNED_SINT** and two **E_D_FF_ANY** D-Flip-Flop blocks to emit the initial event on the first `CLK` and thereafter emit events on plugs `NEG_MAG` and `POS_MAG` **only when the new input value differs from the stored output state** (`D <> Q`).

Interface Structure

Adapter Sockets (Input)

Name	Type	Comment
`Y`	`adapter::types::unidirectional::AS`	Signed input value (AS adapter socket)

Adapter Plugs (Output)

Name	Type	Comment
`NEG_MAG`	`adapter::types::unidirectional::AS`	Negative magnitude (`MAX(0, -Y)`), initial event on 1st call, thereafter on change only
`POS_MAG`	`adapter::types::unidirectional::AS`	Positive magnitude (`MAX(0, Y)`), initial event on 1st call, thereafter on change only

Functionality

Internally, the composite network consists of three components:

1. **SPLIT (SPLIT_SIGNED_SINT)**: Computes `NEG_MAG` and `POS_MAG` from `Y.D1` upon every `Y.E1` event.
2. **DEDUP_NEG (E_D_FF_ANY)**: An event-driven D-Flip-Flop that emits the initial output event upon the first `CLK` event, and thereafter forwards `CLK` to `EO` and updates `Q := D` only when the new data value `D` differs from the current output state `Q` (`D <> Q`).
3. **DEDUP_POS (E_D_FF_ANY)**: A second D-Flip-Flop of the same type that similarly emits the initial output event on the first call and subsequently emits event `POS_MAG.E1` only when the positive magnitude value changes.

```
Y (AS Adapter Socket)
 ├──> SPLIT (SPLIT_SIGNED_SINT)
       ├──> NEG_MAG ──> DEDUP_NEG (E_D_FF_ANY D-Flip-Flop) ──> NEG_MAG (AS Adapter Plug)
       └──> POS_MAG ──> DEDUP_POS (E_D_FF_ANY D-Flip-Flop) ──> POS_MAG (AS Adapter Plug)
```

Technical Features

- **Adapter-Driven Architecture:** Fully compatible with unidirectional adapter family `adapter::types::unidirectional::AS`.
- **D-Flip-Flop Event Filtering:** The `E_D_FF_ANY` blocks guarantee by definition that the initial output event is emitted on the first `CLK`, and subsequent output events `EO` are generated only when the new input `D` differs from the stored output `Q`.
- **Overflow Safety:** Inherits saturation logic from `SPLIT_SIGNED_SINT` for two's complement integer limits.

Application Scenarios

- Connecting bipolar sensors (e.g. steering angle sensor, tilt sensor, joystick) to dual VT bargraph displays.
- Adapter-driven actuation of hydraulic cylinders (extend/retract).

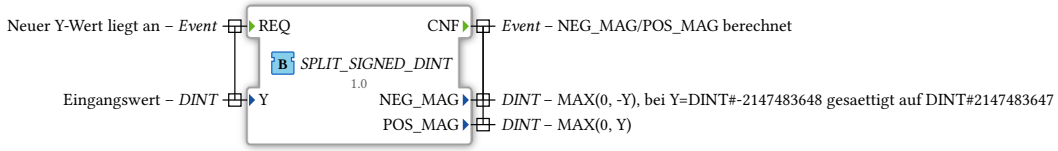
Comparison with Similar Blocks

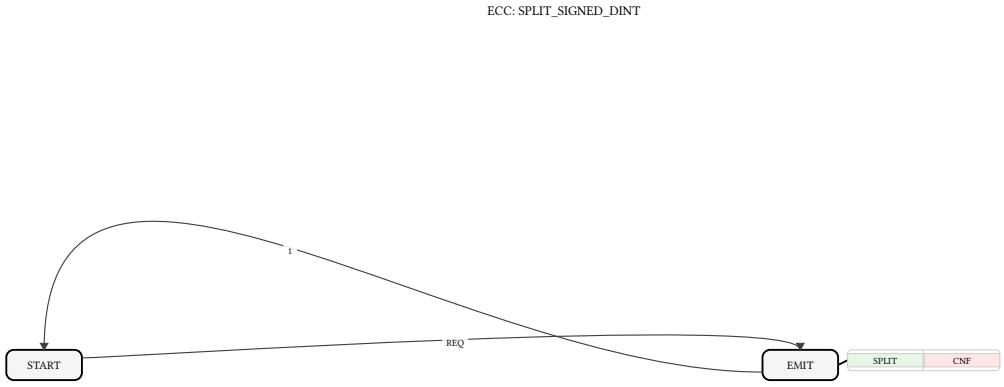
- **AS_SPLIT_SIGNED**: Composite FB with adapter interfaces and D-Flip-Flop event filtering.
- **SPLIT_SIGNED_SINT**: Underlying Basic FB for pure signal computation without adapters.

Conclusion

AS_SPLIT_SIGNED delivers an elegant, adapter-driven, and event-efficient solution for signed signal splitting in IEC 61499 applications.

SPLIT_SIGNED_DINT





• – ○ • – ○ • – ○ •

Introduction

The **SPLIT_SIGNED_DINT** function block splits a signed `DINT` input value `Y` into two single-sided magnitude components:

- `NEG_MAG = MAX(0, -Y)` (magnitude of negative deflection)
- `POS_MAG = MAX(0, Y)` (magnitude of positive deflection)

It serves as the core calculation block for directional indicators (e.g. separate bargraphs for left/right deflection from center position) or dual directional actuators (e.g. raise/lower or forward/reverse).

Interface Structure

Event Inputs

Name	Type	Comment
`REQ`	`Event`	Request calculation (with `Y`)

Event Outputs

Name	Type	Comment
`CNF`	`Event`	Calculation confirmation (with `NEG_MAG`, `POS_MAG`)

Data Inputs

Name	Type	Comment
`Y`	`DINT`	Signed input value

Data Outputs

Name	Type	Comment
`NEG_MAG`	`DINT`	`MAX(0, -Y)` – Magnitude of negative deflection

POS_MAG	DINT	MAX(0, Y) – Magnitude of positive deflection

Functionality

Upon receiving event REQ, the block executes the following splitting logic:

```
IF Y = DINT#-2147483648 THEN
    NEG_MAG := DINT#2147483647;
ELSE
    NEG_MAG := MAX(DINT#0, -Y);
END_IF;
POS_MAG := MAX(DINT#0, Y);
```

The block emits confirmation event CNF on every REQ. Event filtering for unchanged values is handled at the adapter wrapper level (**ADI_SPLIT_SIGNED**) using **E_D_FF_ANY** D-Flip-Flop instances.

Overflow Handling & Saturation

For two's complement integer types, the magnitude of the negative minimum (-2147483648) is 1 greater than the representable positive maximum (2147483647). Mathematical negation of DINT#-2147483648 without special handling would result in integer overflow and incorrect sign reversal.

SPLIT_SIGNED_DINT explicitly handles this edge case:

- When Y = DINT#-2147483648, NEG_MAG saturates to the maximum representable positive value DINT#2147483647.
- Control engineering advantage: Saturation at range limit instead of overflow or sign flip.

Application Scenarios

- Splitting bipolar signals for separate visual components (e.g. left/right bargraphs).
- Driving two unidirectional actuators (e.g. raise vs. lower valves) from a single bipolar setpoint.
- Pre-processing in control loops with direction-dependent deadbands or curves.

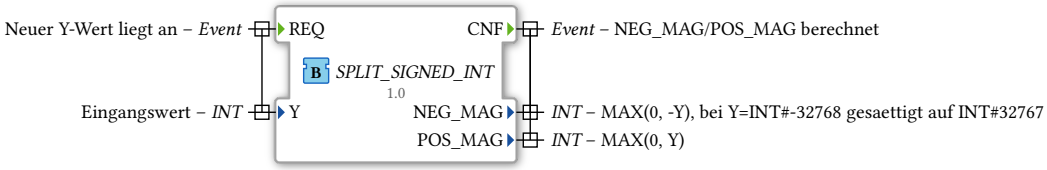
Comparison with Similar Blocks

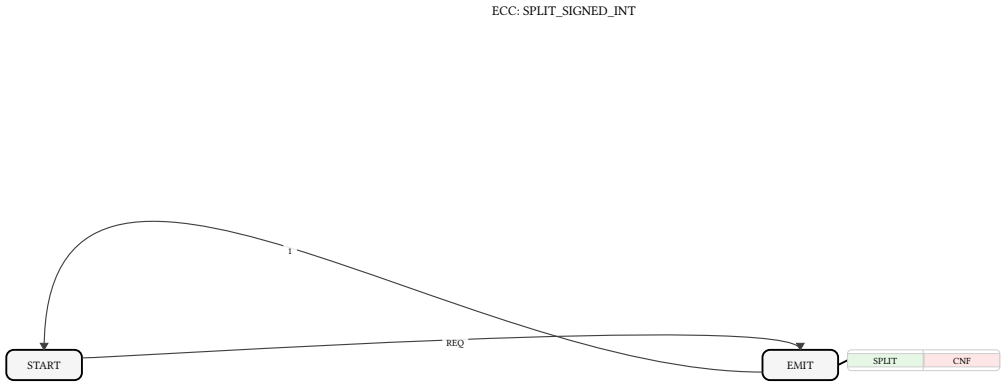
- **SPLIT_SIGNED_DINT**: Pure value/event calculation block for DINT (Basic FB).
- **ADI_SPLIT_SIGNED**: Composite FB wrapper featuring adapter interfaces (adapter::types::unidirectional::ADI) and automatic per-side D-Flip-Flop event filtering (E_D_FF_ANY).

Conclusion

SPLIT_SIGNED_DINT provides robust, overflow-safe splitting of DINT values into positive magnitude components.

SPLIT_SIGNED_INT





• – ○ • – ○ • – ○ •

Introduction

The **SPLIT_SIGNED_INT** function block splits a signed `INT` input value `Y` into two single-sided magnitude components:

- `NEG_MAG = MAX(0, -Y)` (magnitude of negative deflection)
- `POS_MAG = MAX(0, Y)` (magnitude of positive deflection)

It serves as the core calculation block for directional indicators (e.g. separate bargraphs for left/right deflection from center position) or dual directional actuators (e.g. raise/lower or forward/reverse).

Interface Structure

Event Inputs

Name	Type	Comment
`REQ`	`Event`	Request calculation (with `Y`)

Event Outputs

Name	Type	Comment
`CNF`	`Event`	Calculation confirmation (with `NEG_MAG`, `POS_MAG`)

Data Inputs

Name	Type	Comment
`Y`	`INT`	Signed input value

Data Outputs

Name	Type	Comment
`NEG_MAG`	`INT`	`MAX(0, -Y)` – Magnitude of negative deflection

POS_MAG	INT	MAX(0, Y) – Magnitude of positive deflection

Functionality

Upon receiving event REQ, the block executes the following splitting logic:

```
IF Y = INT#-32768 THEN
    NEG_MAG := INT#32767;
ELSE
    NEG_MAG := MAX(INT#0, -Y);
END_IF;
POS_MAG := MAX(INT#0, Y);
```

The block emits confirmation event CNF on every REQ. Event filtering for unchanged values is handled at the adapter wrapper level (**AI_SPLIT_SIGNED**) using **E_D_FF_ANY** D-Flip-Flop instances.

Overflow Handling & Saturation

For two's complement integer types, the magnitude of the negative minimum (-32768) is 1 greater than the representable positive maximum (32767). Mathematical negation of INT#-32768 without special handling would result in integer overflow and incorrect sign reversal.

SPLIT_SIGNED_INT explicitly handles this edge case:

- When Y = INT#-32768, NEG_MAG saturates to the maximum representable positive value INT#32767.
- Control engineering advantage: Saturation at range limit instead of overflow or sign flip.

Application Scenarios

- Splitting bipolar signals for separate visual components (e.g. left/right bargraphs).
- Driving two unidirectional actuators (e.g. raise vs. lower valves) from a single bipolar setpoint.
- Pre-processing in control loops with direction-dependent deadbands or curves.

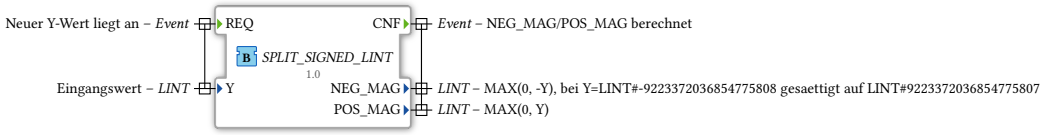
Comparison with Similar Blocks

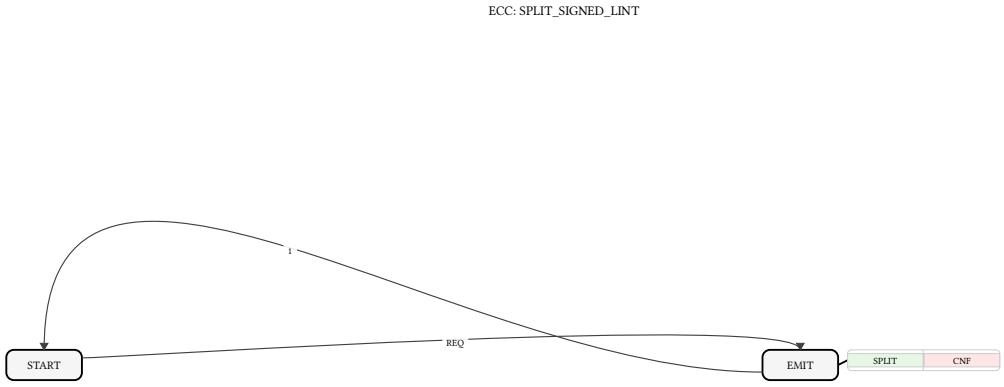
- **SPLIT_SIGNED_INT**: Pure value/event calculation block for INT (Basic FB).
- **AI_SPLIT_SIGNED**: Composite FB wrapper featuring adapter interfaces (adapter::types::unidirectional::AI) and automatic per-side D-Flip-Flop event filtering (E_D_FF_ANY).

Conclusion

SPLIT_SIGNED_INT provides robust, overflow-safe splitting of INT values into positive magnitude components.

SPLIT_SIGNED_LINT





• – ○ • – ○ • – ○ •

Introduction

The **SPLIT_SIGNED_LINT** function block splits a signed `LINT` input value `Y` into two single-sided magnitude components:

- `NEG_MAG = MAX(0, -Y)` (magnitude of negative deflection)
- `POS_MAG = MAX(0, Y)` (magnitude of positive deflection)

It serves as the core calculation block for directional indicators (e.g. separate bargraphs for left/right deflection from center position) or dual directional actuators (e.g. raise/lower or forward/reverse).

Interface Structure

Event Inputs

Name	Type	Comment
`REQ`	`Event`	Request calculation (with `Y`)

Event Outputs

Name	Type	Comment
`CNF`	`Event`	Calculation confirmation (with `NEG_MAG`, `POS_MAG`)

Data Inputs

Name	Type	Comment
`Y`	`LINT`	Signed input value

Data Outputs

Name	Type	Comment
`NEG_MAG`	`LINT`	`MAX(0, -Y)` – Magnitude of negative deflection

POS_MAG	LINT	MAX(0, Y) – Magnitude of positive deflection

Functionality

Upon receiving event REQ, the block executes the following splitting logic:

```
IF Y = LINT#-9223372036854775808 THEN
    NEG_MAG := LINT#9223372036854775807;
ELSE
    NEG_MAG := MAX(LINT#0, -Y);
END_IF;
POS_MAG := MAX(LINT#0, Y);
```

The block emits confirmation event CNF on every REQ. Event filtering for unchanged values is handled at the adapter wrapper level (**ALI_SPLIT_SIGNED**) using **E_D_FF_ANY** D-Flip-Flop instances.

Overflow Handling & Saturation

For two's complement integer types, the magnitude of the negative minimum (-9223372036854775808) is 1 greater than the representable positive maximum (9223372036854775807). Mathematical negation of LINT#-9223372036854775808 without special handling would result in integer overflow and incorrect sign reversal.

SPLIT_SIGNED_LINT explicitly handles this edge case:

- When Y = LINT#-9223372036854775808, NEG_MAG saturates to the maximum representable positive value LINT#9223372036854775807.
- Control engineering advantage: Saturation at range limit instead of overflow or sign flip.

Application Scenarios

- Splitting bipolar signals for separate visual components (e.g. left/right bargraphs).
- Driving two unidirectional actuators (e.g. raise vs. lower valves) from a single bipolar setpoint.
- Pre-processing in control loops with direction-dependent deadbands or curves.

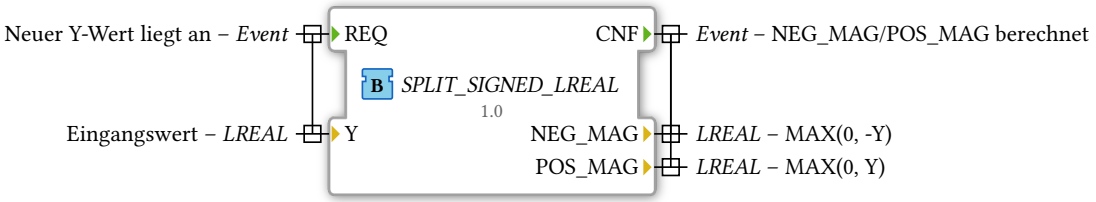
Comparison with Similar Blocks

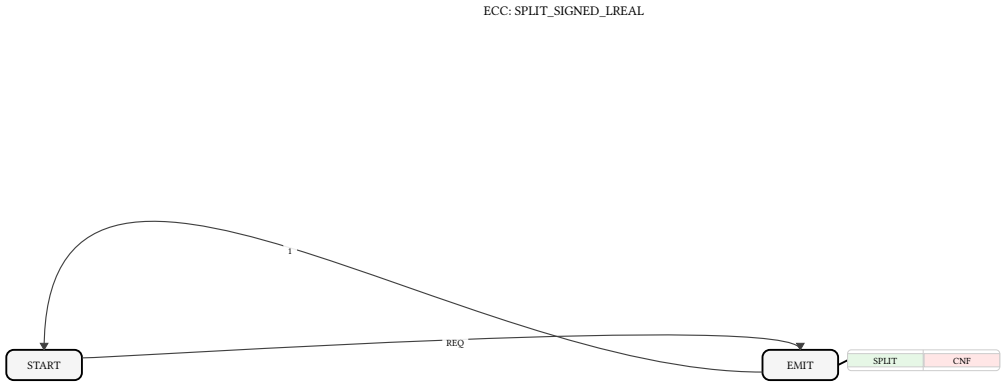
- **SPLIT_SIGNED_LINT**: Pure value/event calculation block for LINT (Basic FB).
- **ALI_SPLIT_SIGNED**: Composite FB wrapper featuring adapter interfaces (adapter::types::unidirectional::ALI) and automatic per-side D-Flip-Flop event filtering (E_D_FF_ANY).

Conclusion

SPLIT_SIGNED_LINT provides robust, overflow-safe splitting of LINT values into positive magnitude components.

SPLIT_SIGNED_LREAL





• – ○ • – ○ • – ○ •

Introduction

The **SPLIT_SIGNED_LREAL** function block splits a signed `LREAL` input value `Y` into two single-sided magnitude components:

- `NEG_MAG = MAX(0, -Y)` (magnitude of negative deflection)
- `POS_MAG = MAX(0, Y)` (magnitude of positive deflection)

It serves as the core calculation block for directional indicators (e.g. separate bargraphs for left/right deflection from center position) or dual directional actuators (e.g. raise/lower or forward/reverse).

Interface Structure

Event Inputs

Name	Type	Comment
`REQ`	`Event`	Request calculation (with `Y`)

Event Outputs

Name	Type	Comment
`CNF`	`Event`	Calculation confirmation (with `NEG_MAG`, `POS_MAG`)

Data Inputs

Name	Type	Comment
`Y`	`LREAL`	Signed input value

Data Outputs

Name	Type	Comment
`NEG_MAG`	`LREAL`	`MAX(0, -Y)` – Magnitude of negative deflection

POS_MAG	LREAL	MAX(0, Y) – Magnitude of positive deflection

Functionality

Upon receiving event REQ, the block executes the following splitting logic:

```
NEG_MAG := MAX(LREAL#0.0, -Y);
POS_MAG := MAX(LREAL#0.0, Y);
```

The block emits confirmation event CNF on every REQ. Event filtering for unchanged values is handled at the adapter wrapper level (**ALR_SPLIT_SIGNED**) using **E_D_FF_ANY** D-Flip-Flop instances.

Floating-Point Behavior

Since LREAL is a floating-point type, the two's complement integer asymmetry issue does not apply. Negation -Y is representable across the full range without overflow.

Application Scenarios

- Splitting bipolar signals for separate visual components (e.g. left/right bargraphs).
- Driving two unidirectional actuators (e.g. raise vs. lower valves) from a single bipolar setpoint.
- Pre-processing in control loops with direction-dependent deadbands or curves.

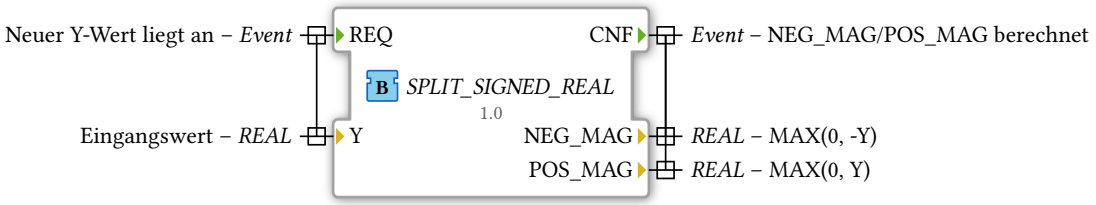
Comparison with Similar Blocks

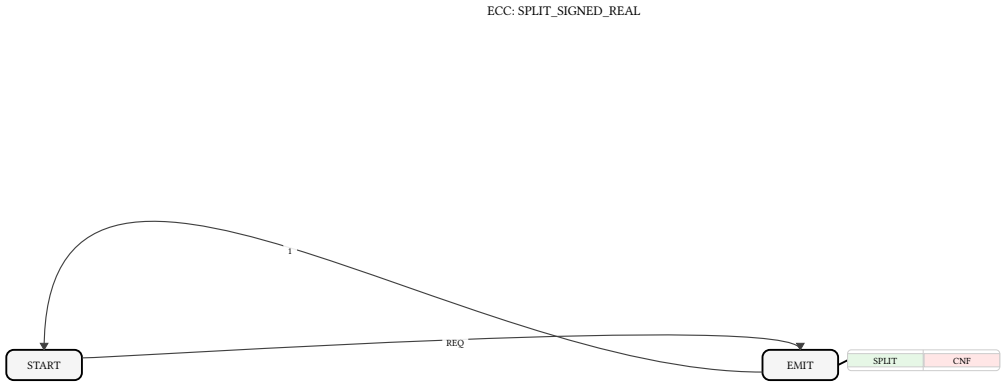
- **SPLIT_SIGNED_LREAL**: Pure value/event calculation block for LREAL (Basic FB).
- **ALR_SPLIT_SIGNED**: Composite FB wrapper featuring adapter interfaces (adapter::types::unidirectional::ALR) and automatic per-side D-Flip-Flop event filtering (E_D_FF_ANY).

Conclusion

SPLIT_SIGNED_LREAL provides robust, overflow-safe splitting of LREAL values into positive magnitude components.

SPLIT_SIGNED_REAL





• – ○ • – ○ • – ○ •

Introduction

The **SPLIT_SIGNED_REAL** function block splits a signed `REAL` input value `Y` into two single-sided magnitude components:

- `NEG_MAG = MAX(0, -Y)` (magnitude of negative deflection)
- `POS_MAG = MAX(0, Y)` (magnitude of positive deflection)

It serves as the core calculation block for directional indicators (e.g. separate bargraphs for left/right deflection from center position) or dual directional actuators (e.g. raise/lower or forward/reverse).

Interface Structure

Event Inputs

Name	Type	Comment
`REQ`	`Event`	Request calculation (with `Y`)

Event Outputs

Name	Type	Comment
`CNF`	`Event`	Calculation confirmation (with `NEG_MAG`, `POS_MAG`)

Data Inputs

Name	Type	Comment
`Y`	`REAL`	Signed input value

Data Outputs

Name	Type	Comment
`NEG_MAG`	`REAL`	`MAX(0, -Y)` – Magnitude of negative deflection

POS_MAG	REAL	MAX(0, Y) – Magnitude of positive deflection

Functionality

Upon receiving event REQ, the block executes the following splitting logic:

```
NEG_MAG := MAX(REAL#0.0, -Y);
POS_MAG := MAX(REAL#0.0, Y);
```

The block emits confirmation event CNF on every REQ. Event filtering for unchanged values is handled at the adapter wrapper level (**AR_SPLIT_SIGNED**) using **E_D_FF_ANY** D-Flip-Flop instances.

Floating-Point Behavior

Since REAL is a floating-point type, the two's complement integer asymmetry issue does not apply. Negation -Y is representable across the full range without overflow.

Application Scenarios

- Splitting bipolar signals for separate visual components (e.g. left/right bargraphs).
- Driving two unidirectional actuators (e.g. raise vs. lower valves) from a single bipolar setpoint.
- Pre-processing in control loops with direction-dependent deadbands or curves.

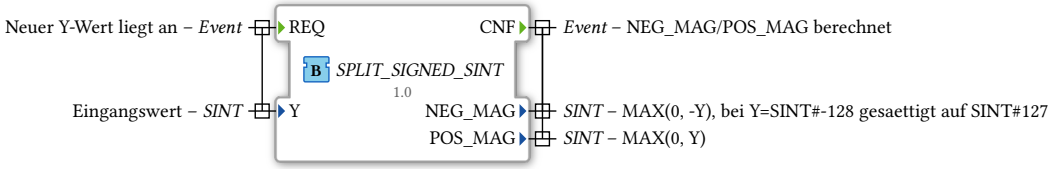
Comparison with Similar Blocks

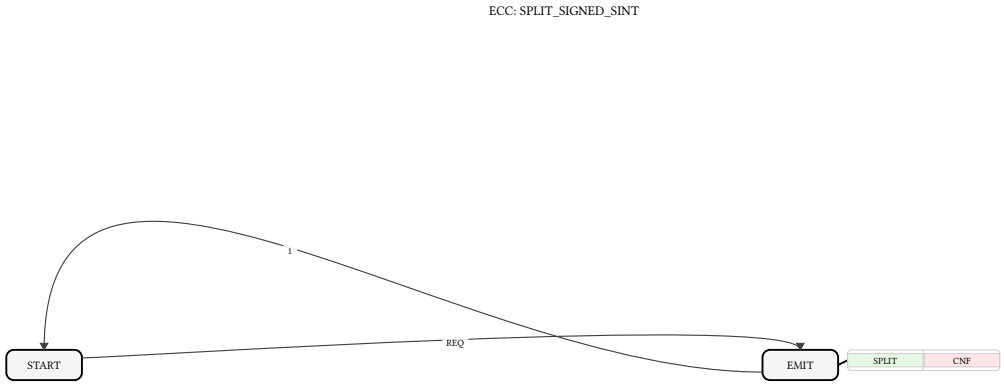
- **SPLIT_SIGNED_REAL**: Pure value/event calculation block for REAL (Basic FB).
- **AR_SPLIT_SIGNED**: Composite FB wrapper featuring adapter interfaces (adapter::types::unidirectional::AR) and automatic per-side D-Flip-Flop event filtering (E_D_FF_ANY).

Conclusion

SPLIT_SIGNED_REAL provides robust, overflow-safe splitting of REAL values into positive magnitude components.

SPLIT_SIGNED_SINT





• – ○ • – ○ • – ○ •

Introduction

The **SPLIT_SIGNED_SINT** function block splits a signed `SINT` input value `Y` into two single-sided magnitude components:

- `NEG_MAG = MAX(0, -Y)` (magnitude of negative deflection)
- `POS_MAG = MAX(0, Y)` (magnitude of positive deflection)

It serves as the core calculation block for directional indicators (e.g. separate bargraphs for left/right deflection from center position) or dual directional actuators (e.g. raise/lower or forward/reverse).

Interface Structure

Event Inputs

Name	Type	Comment
`REQ`	`Event`	Request calculation (with `Y`)

Event Outputs

Name	Type	Comment
`CNF`	`Event`	Calculation confirmation (with `NEG_MAG`, `POS_MAG`)

Data Inputs

Name	Type	Comment
`Y`	`SINT`	Signed input value

Data Outputs

Name	Type	Comment
`NEG_MAG`	`SINT`	`MAX(0, -Y)` – Magnitude of negative deflection

POS_MAG	SINT	MAX(0, Y) – Magnitude of positive deflection

Functionality

Upon receiving event REQ, the block executes the following splitting logic:

```
IF Y = SINT#-128 THEN
    NEG_MAG := SINT#127;
ELSE
    NEG_MAG := MAX(SINT#0, -Y);
END_IF;
POS_MAG := MAX(SINT#0, Y);
```

The block emits confirmation event CNF on every REQ. Event filtering for unchanged values is handled at the adapter wrapper level (**AS_SPLIT_SIGNED**) using **E_D_FF_ANY** D-Flip-Flop instances.

Overflow Handling & Saturation

For two's complement integer types, the magnitude of the negative minimum (-128) is 1 greater than the representable positive maximum (127). Mathematical negation of SINT#-128 without special handling would result in integer overflow and incorrect sign reversal.

SPLIT_SIGNED_SINT explicitly handles this edge case:

- When Y = SINT#-128, NEG_MAG saturates to the maximum representable positive value SINT#127.
- Control engineering advantage: Saturation at range limit instead of overflow or sign flip.

Application Scenarios

- Splitting bipolar signals for separate visual components (e.g. left/right bargraphs).
- Driving two unidirectional actuators (e.g. raise vs. lower valves) from a single bipolar setpoint.
- Pre-processing in control loops with direction-dependent deadbands or curves.

Comparison with Similar Blocks

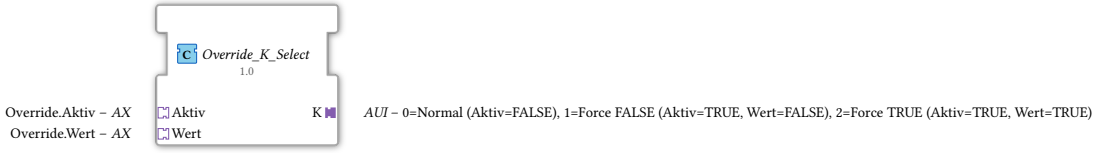
- **SPLIT_SIGNED_SINT**: Pure value/event calculation block for SINT (Basic FB).
- **AS_SPLIT_SIGNED**: Composite FB wrapper featuring adapter interfaces (adapter::types::unidirectional::AS) and automatic per-side D-Flip-Flop event filtering (E_D_FF_ANY).

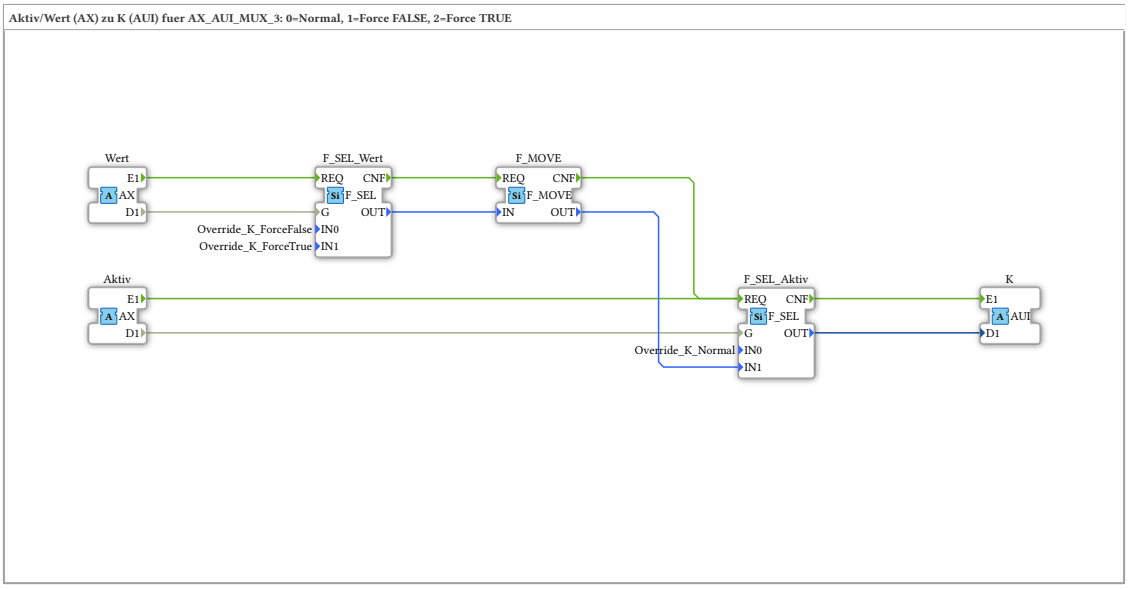
Conclusion

SPLIT_SIGNED_SINT provides robust, overflow-safe splitting of SINT values into positive magnitude components.

2.1.11.3 OverrideK

Override_K_Select





• — ○ • — ○ • — ○ •

Introduction

The `Override_K_Select` function block converts a pair of boolean override signals (`Aktiv`/`Wert`, each as an AX adapter) into a single `UINT` selector `K` (as an AUI adapter), as expected by an `AX_AUI_MUX_3`. It encapsulates the recurring project pattern "a sensor value can be forced to either `FALSE` or `TRUE` via an RW override" (see `Sensor.Leermelder.Override.*` in `docs/datenpunkte.md` of the Getreideannahme application) in a single, reusable block, instead of re-wiring the conversion at every call site.

Interface Structure

Event Inputs

None of its own. Events flow exclusively through the `Aktiv`/`Wert` adapter sockets.

Event Outputs

None of its own. The result is emitted through the `K` adapter plug.

Data Inputs

None of its own. All data is carried through the adapter sockets.

Data Outputs

None of its own. The result is emitted through the `K` adapter plug.

Adapters

Direction	Name	Type	Description
Socket (input)	`Aktiv`	`adapter::types::unidirectional::AX`	`Override.Aktiv`
Socket (input)	`Wert`	`adapter::types::unidirectional::AX`	`Override.Wert`
Plug (output)	`K`	`adapter::types::unidirectional::AUI`	0=Normal (`Aktiv`=FALSE), 1=Force FALSE (`Aktiv`=TRUE, `Wert`=FALSE), 2=Force TRUE (`Aktiv`=TRUE, `Wert`=TRUE)

Functionality

Internally the block uses two chained, generic `F_SEL` blocks (the standard IEC 61131-3 selector `OUT := G ? IN1 : IN0`) instead of custom ST logic:

1. **F_SEL_Wert**: `G := Wert.D1`. Returns `Override_K_ForceFalse` (1) when `Wert = FALSE`, otherwise `Override_K_ForceTrue` (2). Triggered by `Wert.E1`.
2. **F_MOVE** (attribute `DataType="UINT"`): captures `F_SEL_Wert.OUT`, triggered by `F_SEL_Wert.CNF`. Acts as a type anchor for `F_SEL`'s generic `ANY` result (see Technical Details) and then re-triggers `F_SEL_Aktiv`.
3. **F_SEL_Aktiv**: `G := Aktiv.D1`. Returns `Override_K_Normal` (0) when `Aktiv = FALSE`, otherwise the value most recently held by `F_MOVE` (1 or 2). Triggered both directly by `Aktiv.E1` and indirectly via `F_MOVE.CNF`, so a change in `Wert` alone still produces a fresh `K` even when `Aktiv` itself hasn't changed.
4. `F_SEL_Aktiv.OUT`/`F_SEL_Aktiv.CNF` are wired directly onto the data output variable `K.D1` and the event `K.E1` of the block's own plug — no separate `_TO_AUI` conversion FB in between (see Technical Details).

Technical Details

- **Writing directly onto its own plug pins.** Instead of routing through a `UINT_TO_AUI`-style conversion FB, the block connects `F_SEL_Aktiv.OUT`/`.CNF` directly to `K.D1`/`K.E1` — the event/data variables the `AUI` adapter type itself declares on the plug side. The same technique is used by const_AX_TRUE and const_AX_FALSE for their own `OUT` plug.
- **`F_MOVE` as a type anchor for `F_SEL`'s generic `ANY` result.** `F_SEL.OUT` is generic (`ANY`); without a downstream `F_MOVE` carrying a fixed `DataType` attribute, the value can't be cleanly passed to the second `F_SEL` call as a `UINT`.
- **The K values are named constants, not literals.** `Override_K_Normal`/`Override_K_ForceFalse`/`Override_K_ForceTrue` come from the global constants package Override_K (`adapter::OverrideK::const`), not as bare `UINT#0`/`UINT#1`/ `UINT#2` literals directly in the `F_SEL` parameters.
- **No `REQ`/`CNF` of its own.** The block reacts exclusively to the sockets' own events (`Aktiv.E1`, `Wert.E1`) — there is no external trigger input.

State Overview

The block has no state machine of its own (a composite FB without an ECC); its behavior is purely combinational, recomputed on every change to `Aktiv` or `Wert`.

Aktiv	Wert	K
FALSE	—	0 (Normal)
TRUE	FALSE	1 (Force FALSE)
TRUE	TRUE	2 (Force TRUE)

Application Scenarios

- **Sensor override via RW OPC-UA nodes**: feed an `Aktiv`/`Wert` pair subscribed via `AX_SUBSCRIBE_1` (e.g. `Sensor.Leermelder.Override.Aktiv`/`.Wert`) directly into the `K` input of an `AX_AUI_MUX_3`, whose three inputs supply the normal value, `FALSE`, and `TRUE`.
- **Diagnostic/test pages**: any boolean sensor that a web UI needs to force to a fixed value for testing can reuse the same `Aktiv`/`Wert`/`K` pattern.

⚖️ Comparison with Similar Blocks

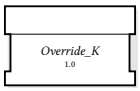
- **AX_AUI_MUX_3**: the actual multiplexer that `Override_K_Select` drives with its `K` output. `Override_K_Select` only computes the selector; it does not select a value itself.
- **const_AX_TRUE** / **const_AX_FALSE**: supply the fixed `IN2`/`IN3` values (Force FALSE/TRUE) that the selector computed by `Override_K_Select` picks out on the `AX_AUI_MUX_3`.
- **Override_K**: the global constants package with the three named K values this block uses internally.

Conclusion

`Override_K_Select` bundles the conversion of an Aktiv/Wert override pair into a MUX selector in a single, reusable block, built entirely from standard blocks (`F_SEL`, `F_MOVE`) with no custom ST algorithms of its own. Combined with `AX_AUI_MUX_3` and the `const_AX_TRUE`/`const_AX_FALSE` constants, this forms the complete, reusable pattern for a boolean sensor value that can be forced via OPC-UA.

const

Override_K



• – ◦ • – ◦ • – ◦ •

Introduction

The global constants package `Override_K` supplies the three named `UINT` values that Override_K_Select uses to populate its `K` output for `AX_AUI_MUX_3`. Instead of wiring `UINT#0`/`UINT#1`/`UINT#2` as literals into the `F_SEL` parameters, the block references these three constants by their descriptive name. The package is defined in the `adapter::OverrideK::const` namespace.

Interface Structure

Being a global-constants package, it has no event- or data-flow-based interface in the conventional function-block sense.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The constants are replaced by their fixed defined values at compile time:

- `Override_K_Normal`: `UINT#0` — the K value for "no override" (`AX_AUI_MUX_3.IN1`, the normal value).
- `Override_K_ForceFalse`: `UINT#1` — the K value for "override forced to FALSE" (`AX_AUI_MUX_3.IN2`).
- `Override_K_ForceTrue`: `UINT#2` — the K value for "override forced to TRUE" (`AX_AUI_MUX_3.IN3`).

Technical Details

- **Directly coupled to `AX_AUI_MUX_3`'s input order.** The numeric values 0/1/2 correspond exactly to `AX_AUI_MUX_3`'s input numbering (`IN1`/`IN2`/`IN3`) — any change to that order in Override_K_Select would need to be kept in sync here.
- **Used only by `Override_K_Select`.** Unlike, say, `OPC_UA_Endpoints`, this package is tightly coupled to a single block rather than several independent call sites.

State Overview

Being static, constant values, there is no state machine.

Application Scenarios

- Referenced in the `IN0`/`IN1` `Parameter` values of the two `F_SEL` blocks in Override_K_Select.
- Comparison values for a consumer that wants to evaluate an `AX_AUI_MUX_3`'s raw `K` value rather than just its muxed output.

⚖️ Comparison with Similar Blocks

- **IDs** (ISOBUS UT library): follows the same GCF pattern (named constants instead of literals), but covers ISOBUS object IDs rather than a MUX selector.

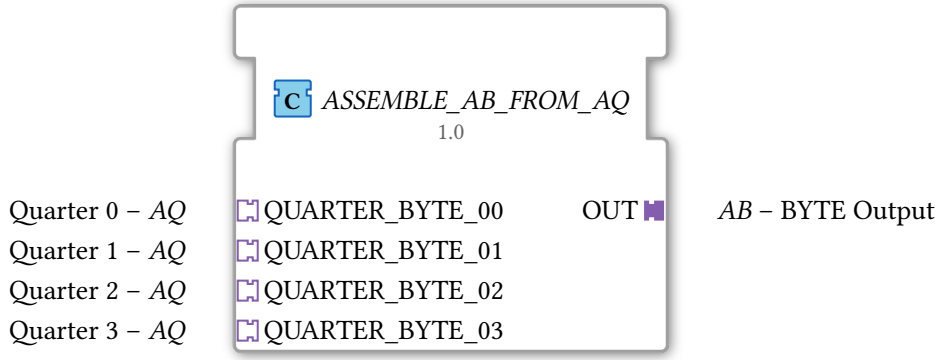
Conclusion

`Override_K` is a small, tightly-coupled constants package that names the three K values used by `Override_K_Select`, making them more readable and maintainable than scattered `UINT#0/1/2` literals.

2.1.11.4 Numeric Index (adapter)

Here you will find all exercises from the `adapter` workspace, listed in ascending order. ...

ASSEMBLE_AB_FROM_AQ



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASSEMBLE_AB_FROM_AQ** combines four **AQ** quarter adapters (quarter bytes) into a single **AB** byte adapter. It encapsulates the logic for assembling a complete byte from four incoming 4-bit values and providing it via a standardized adapter interface. The block is specifically designed for use in distributed automation systems according to IEC 61499.

Interface Structure

Event Inputs

The block does not have dedicated top-level event inputs.

All incoming events are transmitted via the **adapter sockets**. Each application adapter of type `AQ` has an event output `E1`, which is internally connected to the REQ input of the assembly module.

Event Outputs

- **OUT.E1**: Event output of the byte adapter. It becomes active as soon as a new byte has been fully assembled and clocked via the internal flip-flop.

Data Inputs

Data is read in via the **adapter sockets** in the form of 4-bit values (nibbles):

- **QUARTER_BYTE_00.D1**: Nibble for the least significant bits (bits 0-3)
- **QUARTER_BYTE_01.D1**: Nibble for bits 4-7
- **QUARTER_BYTE_02.D1**: Nibble for bits 8-11
- **QUARTER_BYTE_03.D1**: Nibble for bits 12-15

(Note: The actual bit order may vary depending on the implementation of the internal `ASSEMBLE_BYTE_FROM_QUARTERS` module; usually, the bits are combined in ascending order.)

Data Outputs

- **OUT.D1**: Output of the byte adapter. Returns the fully assembled byte (8 bits) as an integer value.

Adapter

Type	Direction	Name	Description
`adapter::types::unidirectional::AQ`	Socket (Input)	`QUARTER_BYTE_00` .. `QUARTER_BYTE_03`	Four quarter adapters, each providing a nibble and an event.
`adapter::types::unidirectional::PAB`	Plug (Output)	`OUT`	Byte adapter providing the assembled byte and an event.

Functionality

The function block operates entirely event-driven. As soon as one of the four connected AQ adapters sends an event on its `E1` output, this event is forwarded to the internal component `ASSEMBLE_BYTE_FROM_QUARTERS` (event `REQ`). Simultaneously, the current nibble values from all four AQ adapter inputs are transferred to the assembly component via data connections.

The assembly component internally combines the four nibbles into an 8-bit value. After successful processing, it sends an acknowledgment event `CNF` to the edge-triggered D flip-flop (`E_D_FF_ANY`). The flip-flop takes the data value (from the assembly output) on a rising edge and passes it to the data output `OUT.D1`. Simultaneously, the output event `OUT.E1` is generated.

Thus, the output is always clock-synchronized: Only when a complete byte has been calculated is it passed through. Multiple events on different quarter inputs lead to repeated calculations, with all four current nibble values being reprocessed each time.

Technical Features

- **Internal Cascading**: The function block uses two internal blocks – `ASSEMBLE_BYTE_FROM_QUARTERS` for bit combination and `E_D_FF_ANY` for edge-triggered output.
- **Event Synchronization**: Every incoming event triggers the assembly; it is not necessary for all four adapters to deliver an event simultaneously – the block operates with the currently available data.
- **Adapter Interface**: Input and output are exclusively via unidirectional adapters, which simplifies reuse in application networks.
- **No state storage in the function block itself**: The function block is a pure network component that completely delegates logic.

State Overview

The function block does not have its own state machine. The internal logic is determined by the states of `ASSEMBLE_BYTE_FROM_QUARTERS` and `E_D_FF_ANY`:

1. **Waiting for input event**: After initialization or processing, the function block waits for an event from one of the AQ adapters.
2. **Assembly**: At `REQ`, the nibbles are assembled; the D flip-flop waits for the `CNF` event.
3. **Output**: After the flip-flop is clocked, the result is sent to the OUT adapter. The function block then returns to the wait state.

Application Scenarios

- **Bus Data Merging**: When four separate 4-bit lines (e.g., from sensors or data sources) need to be combined into a single byte.
- **Adapter-Based Data Integration**: In systems based on IEC 61499 adapter architecture, this function block can serve as a generic “byte assembler” at the network layer.
- **Protocol Conversion**: From a quarter-byte protocol to a full-byte protocol, e.g., in serial communication.
- **Test and Simulation Environments**: For easily connecting test adapters.

Comparison with Similar Function Blocks

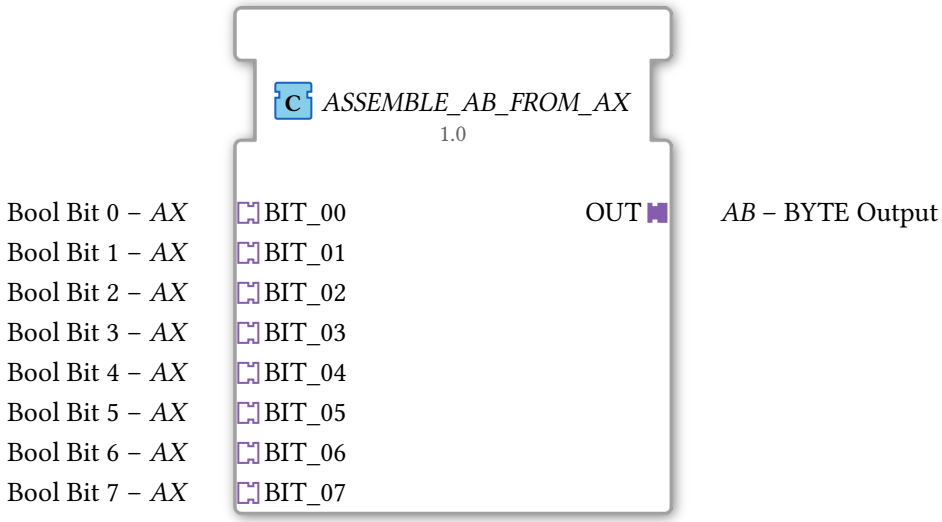
- **ASSEMBLE_BYTE_FROM_QUARTERS** (direct): This function block operates without adapters – it expects four separate data and event inputs. This function block encapsulates this interface in adapters, which increases modularity.
- **ARRAY_TO_BYTE** or similar function blocks: Often implemented using arrays; here specifically for exactly four nibbles and with adapter support.

- **Custom Adapter-Based Assembler**: If needed, `ASSEMBLE_AB_FROM_AQ` can be easily modified for other data widths by adjusting the internal function blocks.

Conclusion

The **ASSEMBLE_AB_FROM_AQ** function block offers an elegant, adapter-based solution for merging four quarter-byte data streams into a full byte. Thanks to the internal cascading of assembly logic and edge-triggered output, the behavior is deterministic and integrates well into event-driven systems. It is particularly suitable for modular automation projects based on IEC 61499 adapter architecture that require a clean separation of data and event flows.

ASSEMBLE_AB_FROM_AX



• – ◦ • – ◦ • – ◦ •

Introduction

The **ASSEMBLE_AB_FROM_AX** function block combines eight Boolean signals, provided via AX adapters (unidirectional, Bool), into a single byte and outputs it via an AB adapter (unidirectional, BYTE). It encapsulates the logic for byte generation and provides a modular, adapter-based interface for processing 8 bits.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are output exclusively via the adapter plug.

Data Inputs

None. All data is transmitted via the adapter sockets.

Data Outputs

None. All data is output via the adapter plug.

Adapter

Direction	Name	Type	Description
Socket (Input)	`BIT_00`	`adapter::types::unidirectional::AX`	Boolean value for bit 0 (least significant bit)
	`BIT_01`	`adapter::types::unidirectional::AX`	Boolean value for bit 1

	BIT_02	adapter::types::unidirectional::AX	Boolean value for bit 2
	BIT_03	adapter::types::unidirectional::AX	Boolean value for bit 3
	BIT_04	adapter::types::unidirectional::AX	Boolean value for bit 4
	BIT_05	adapter::types::unidirectional::AX	Boolean value for bit 5
	BIT_06	adapter::types::unidirectional::AX	Boolean value for bit 6
	BIT_07	adapter::types::unidirectional::AX	Boolean value for bit 7 (most significant bit)
Plug (output)	OUT	adapter::types::unidirectional::AB	Composite byte (BYTE)

Each AX adapter provides the Boolean value via its data input `D1` and the corresponding event via its event input `E1`. The AB adapter provides the byte via its data output `D1` and the acknowledgment event via its event output `E1`.

Functionality

The function block is implemented as a composite module and consists internally of:

1. **ASSEMBLE_BYTE_FROM_BOOLS** – Receives the eight Boolean values (from `BIT_00` to `BIT_07`) and assembles them into a byte.
2. **E_D_FF_ANY** – A D flip-flop that buffers the calculated byte value and only releases it on a rising edge at the clock input.

Procedure:

- As soon as an event arrives at **one** of the AX adapters (e.g., `BIT_00`), it is forwarded to the `REQ` input of the internal module **ASSEMBLE_BYTE_FROM_BOOLS**.
- The internal component calculates the byte from the current Boolean values of all eight adapters and places it at its data output.
- After the calculation is complete, `ASSEMBLE_BYTE_FROM_BOOLS` sends a `CNF` event, which triggers the clock input (`CLK`) of the D flip-flop **E_D_FF_ANY**.
- The flip-flop receives the current byte value and outputs it at its output `Q`.
- Simultaneously, the flip-flop's event `EO` is passed to the AB adapter's event output `OUT.E1`.

This ensures that the output byte is only updated when an input bit changes, and that the output is stable and synchronized.

Technical Features

- **Adapter-Based Interface** – The component uses only adapters (`AX`/`AB`) instead of individual event and data ports. This allows for easy encapsulation and reuse in modular designs.
- **Internal D Flip-Flop** – The flip-flop prevents intermediate states and only releases the completed byte after the calculation is finished. It also acts as a buffer if multiple input events arrive in quick succession.
- **Efficient Event Control** – Every event at one of the eight AX sockets triggers a recalculation. Unnecessary updates are avoided because the output only occurs after a clock cycle.

State Overview

The functional block does not have its own state machine; It is structured as a pure network consisting of two sub-modules. Its behavior is entirely determined by the internal logic of **ASSEMBLE_BYTE_FROM_BOOLS** and **E_D_FF_ANY**.

Application Scenarios

- **Combining 8 digital sensors** – e.g., limit switches, light barriers, or binary inputs of a PLC, whose states are to be transmitted as bytes.
- **Bit-parallel data transmission** – Conversion of an 8-bit parallel signal into a serial byte for another module (e.g., via adapter coupling).
- **Modular automation functions** – Integration into hierarchies where multiple `ASSEMBLE_AB_FROM_AX` blocks are used to assemble larger data words (e.g., WORD, DWORD).

Comparison with similar modules

Module	Description	Difference
`ASSEMBLE_BYTE_FROM_BOOLS`	Internal block that generates a byte from 8 Boolean inputs (without adapters)	`ASSEMBLE_AB_FROM_AX` encapsulates this block and uses adapters for connectivity.
`eclipse4diac::utils::assembling::ASSEMBLE_BYTE_FROM_BOOLS`	Same function, but with direct event/data ports	`ASSEMBLE_AB_FROM_AX` provides an adapter-based interface and adds a D flip-flop for synchronization.
Custom-built byte assembler	Can be implemented as desired, e.g., using the ST algorithm	Adapters `AX`/`AB` are predefined standard types in 4diac that promote reusability and interchangeability.

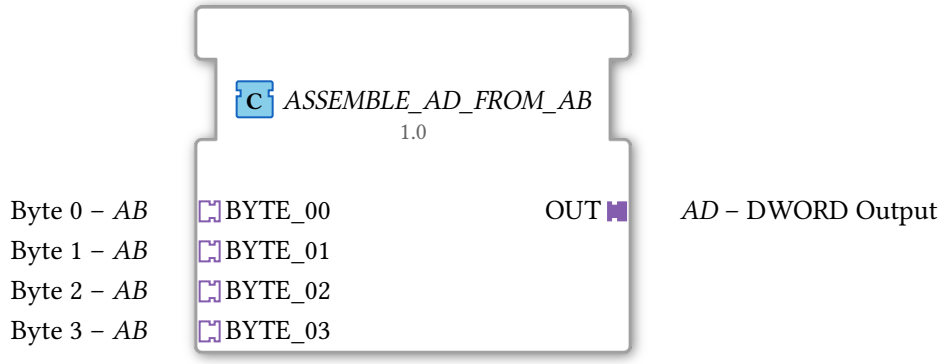
Conclusion

The function block **ASSEMBLE_AB_FROM_AX** is a practical, adapter-based tool for converting eight Boolean signals into one byte. Thanks to the integration of a D flip-flop, it operates reliably and avoids inconsistent intermediate states. Its modular design facilitates reuse in larger project structures, making it a useful component in automation technology with 4diac.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

ASSEMBLE_AD_FROM_AB



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASSEMBLE_AD_FROM_AB** is used to combine four individual byte values, received via unidirectional **AB** adapters (bytes), into a 32-bit **DWORD** value and output it via a unidirectional **AD** adapter. The bytes are combined in the order BYTE_00 (least significant byte) to BYTE_03 (most significant byte). The function block uses an internal combiner and an edge-triggered flip-flop to update the output only after the combination is complete.

Interface Structure

Event Inputs

The function block does not have direct event inputs. Events are received via the connected **Socket Adapters** (BYTE_00 ... BYTE_03).

Event Outputs

There is no direct event output. The output event is provided via the **Plug Adapter** (OUT).

Data Inputs

There are no direct data inputs. All data is received via the **Socket Adapters** (BYTE_00 ... BYTE_03).

Data Outputs

There is no direct data output. The merged DWORD value is output via the **Plug Adapter** (OUT).

Adapters

Name	Type	Direction	Description
BYTE_00	`adapter::types::unidirectional::AB`	Socket	Byte 0 (least significant byte)
BYTE_01	`adapter::types::unidirectional::AB`	Socket	Byte 1
BYTE_02	`adapter::types::unidirectional::AB`	Socket	Byte 2
BYTE_03	`adapter::types::unidirectional::AB`	Socket	Byte 3 (most significant byte)
OUT	`adapter::types::unidirectional::AD`	Plug	32-bit DWORD output

Functionality

The component operates entirely within a sub-application network:

1. An event at one of the four socket adapters (BYTE_00 ... BYTE_03) triggers the **REQ** input of the internal `ASSEMBLE_DWORD_FROM_BYTES` component via the respective `E1` line.
2. This component reads the current values at its data inputs (`BYTE_00` ... `BYTE_03`), combines them into a 32-bit value, and signals this with **CNF**.
3. The **CNF** event triggers the **CLK** input of the internal `E_D_FF_ANY` flip-flop. The flip-flop receives the data value (the merged DWORD) present at its **D** input.
4. The flip-flop's **Q** output is permanently connected to the **D1** data output of the plug adapter **OUT**.
5. Simultaneously, the flip-flop output signal triggers the **E1** event of the plug adapter, informing the receiving side of the new value.

This ensures that the output is only updated when an event has arrived at at least one socket adapter and the merge is complete. The last merged value is retained until a new process is completed.

Technical Features

- **Event Synchronization**: All four socket events are placed on a common `REQ` of the merge block. This triggers processing anew with each incoming byte event – it is not necessary to update all four bytes simultaneously.
- **Caching**: The built-in `E_D_FF_ANY` prevents output during the merging phase. Only when the result is stable is it passed on to the plug.
- **Adapter-based**: The component has no direct inputs/outputs, but communicates exclusively via standardized unidirectional adapters (`AB` and `AD`). This allows for flexible integration into existing adapter interfaces.

State Overview

The component itself does not have an explicit state machine. Its behavior is determined by the internal components:

- **Waiting for Event**: No new event is present at the sockets. The output value is static.
- **Summation & Update**: An event occurs → merging is triggered → flip-flop takes on the new value → output is updated.
- **Wait for Next Event**: After a successful update, the state is retained until the next socket event.

Application Scenarios

- **Data Stream Merging**: Merging four byte-oriented sensor or communication data into a single 32-bit value (e.g., for a fieldbus interface).
- **Protocol Wrapper**: Combining individual bytes from different sources into a complete data word, which is then passed to higher-level logic via an AD adapter.
- **Test and Simulation Environments**: Used as a converter between existing byte adapter components and DWORD adapter components.

Comparison with Similar Function Blocks

- **Direct Data Type Converter** (e.g., `BYTE_TO_DWORD`): These often work with simple data inputs/outputs without an event log. The `ASSEMBLE_AD_FROM_AB`, on the other hand, integrates event control and is tailored to the adapter interface.
- **Manual Chaining of Adapters**: Instead of four separate `AB` adapters and a manually configured network, this function block offers a pre-built and tested solution that requires less wiring effort.

Conclusion

The **ASSEMBLE_AD_FROM_AB** is a specialized, adapter-based function block for merging four byte values into a DWORD. Through the internal use of a merger and a flip-flop, reliable, event-driven updates of the output value are achieved. It is ideally suited for systems that rely on standardized unidirectional adapters and require clean, flicker-free data transmission.

ASSEMBLE_AD_FROM_AQ



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_AD_FROM_AQ` is used to combine sixteen separate `AQ` adapters (quarters) into a single `AD` output adapter (DWORD). The term “quarter” indicates that each `AQ` adapter represents part of a 32-bit word—specifically, one byte (8 bits). The block combines these 16 bytes into a complete DWORD and makes it available via a `AD` plug. It is particularly suitable for applications where data arrives in smaller units and only needs to be combined into a larger data type at the receiving end.

Interface Structure

The function block has no independent event or data inputs/outputs. All input and output is handled exclusively via adapters.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Sockets (Inputs)	QUARTER_BYTE_00, ..., QUARTER_BYTE_15	`adapter::types::unidirectional::AQ`	16 identical adapters, each providing an 8-bit data value (quarter/byte) and a corresponding event (`E1`).
Plug (Output)	OUT	`adapter::types::unidirectional::AQ`	Output adapter providing the DWORD (`D1`) composed of the 16 quarter values and an event (`E1`).

Functionality

The function block operates entirely event-driven:

1. As soon as one of the 16 `QUARTER_BYTE` sockets receives an event (`E1`), it is forwarded to the internal function block `ASSEMBLE_DWORD_FROM_QUARTERS` (type `eclipse4diac::utils::assembling::ASSEMBLE_DWORD_FROM_QUARTERS`).
2. The internal function block combines the current data of all 16 quarter values into a 32-bit DWORD (bits 0…7 = Quarter 0, bits 8…15 = Quarter 1, … bits 120…127 = Quarter 15).
3. The result is passed to an edge-triggered D flip-flop (`E_D_FF_ANY`).
4. On the next rising edge of its clock input (`CLK`), the flip-flop outputs the assembled DWORD to the `OUT` adapter and generates an event (`E0`), which in turn triggers the output event `E1` of the `OUT` adapter.

The use of the flip-flop ensures that the output is only updated when the data is fully assembled and stable – even if multiple quarter-wave events arrive virtually simultaneously.

Technical Features

- **No dedicated inputs/outputs:** The component communicates exclusively via adapters, enabling clean encapsulation and reusability in different project contexts.
- **Synchronous Update:** The D flip-flop prevents data inconsistencies if multiple quarters change their values almost simultaneously. The output only changes after all inputs have been evaluated.
- **Autonomous Triggering:** Every quarter event (regardless of the socket) triggers a recalculation. The component always operates with the currently available values of all 16 quarter data.
- **Internal Indirection:** The actual assembly is performed by a specialized sub-component (`ASSEMBLE_DWORD_FROM_QUARTERS`), which keeps the design modular and maintainable.

State Overview

The function block itself does not have its own state machine. It behaves like combinational logic with downstream synchronization. However, the internal flip-flop `E_D_FF_ANY` has an internal state (the stored DWORD value). This state is only changed when an event arrives at `CLK`.

In the idle state (no event at a quarter socket), the output value of the `OUT` adapter remains unchanged.

Application Scenarios

- **Distributed Measurement System:** Multiple sensors each deliver one byte; these are combined in a central processing unit to form a complete DWORD.
- **Data Packet Reconstruction:** Fragmented communication frames (e.g., 16-byte payloads) are reassembled.
- **Mediator Structures:** As part of an adapter chain for converting data widths in IEC 61499 applications.

- **Control Word Synthesis:** Multiple 8-bit control channels are combined into a 32-bit command word.

Comparison with Similar Components

Component	Number of Inputs	Output	Synchronization	Properties
`ASSEMBLE_AD_FROM_AQ`	16 (AQ adapter)	1 (AD adapter)	Yes (flip-flop)	For event-driven byte collection.
`ASSEMBLE_DWORD_FROM_QUARTERS` (internal)	16 data inputs (no adapters)	1 DWORD output	No	Pure data assembly without event synchronization.
Simple `MERGE` component (fictional)	2 ... n data inputs	1 Data Output	Often no	Data concatenation only, no adapter structure.

This component is characterized by the encapsulation of the adapter interfaces and integrated synchronization, making it particularly suitable for heterogeneous, event-driven environments.

Conclusion

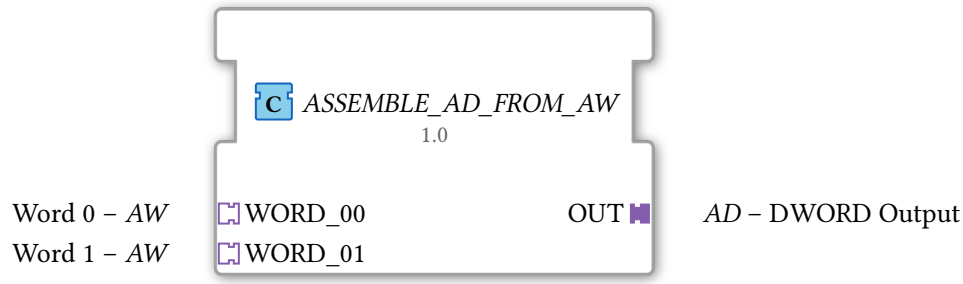
ASSEMBLE_AD_FROM_AQ` is a specialized adapter component for concatenating 16-byte-wide quarter data into a DWORD. Its purely adapter-based interface makes it flexible in its application, while internal synchronization via a flip-flop ensures data consistency during asynchronous events. It is ideally suited for use in modular IEC 61499 applications where data arrives in smaller units and must be assembled into a complete word at the receiving end.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ASSEMBLE_AD_FROM_AW



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASSEMBLE_AD_FROM_AW** combines two word values (WORD) from a single unidirectional AW adapter into a double word (DWORD), which is output via a unidirectional AD adapter. The block encapsulates the logical combination of two 16-bit inputs into a 32-bit output and stores the result using an edge-triggered D flip-flop.

Interface Structure

Event Inputs

The function block has no directly visible event inputs. Event control is handled indirectly via the incoming adapters **WORD_00** and **WORD_01**: Each of these adapters provides a signal via its implicit event output (`E1`) that triggers the internal processing sequence.

Event Outputs

There are no direct event outputs here either. The output adapter **OUT** provides the output event via its implicit event input (`E1`), which is triggered after processing is complete and the data value has been transferred.

Data Inputs

Data is read in via the two socket adapters:

Adapter	Type	Description
`WORD_00`	`adapter::types::unidirectional::AW`	First 16-bit word (lower-order part of the double word)
WORD_01`	`adapter::types::unidirectional::AW`	Second 16-bit word (higher-order part of the double word)

Each of these adapters provides a data output (`D1`) containing the actual WORD value.

Data Outputs

The output is via a plug adapter:

Adapter	Type	Description
OUT`	`adapter::types::unidirectional::AD`	Composite 32-bit Double Word (DWORD)

The adapter `OUT` has one data input (`D1`) that is internally connected to the stored result.

Adapters

The function block provides two incoming adapters (sockets) and one outgoing adapter (plug), all of which are unidirectional:

- **WORD_00**, **WORD_01**: Each returns a WORD and an associated event.
- **OUT**: Receives a DWORD and passes it on with an event.

Functionality

1. As soon as one of the two input events (`WORD_00.E1` or `WORD_01.E1`) arrives, the internal function block `ASSEMBLE_DWORD_FROM_WORDS` is activated via its event input `REQ`.
2. This internal block assembles the 32-bit double word from the two incoming 16-bit values (`WORD_00.D1` and `WORD_01.D1`). `WORD_00` is interpreted as the lower-order word and `WORD_01` as the higher-order word.
3. After the calculation is complete, `ASSEMBLE_DWORD_FROM_WORDS` signals the clock signal for an edge-triggered D flip-flop (`E_D_FF_ANY`) with the event `CNF`.
4. The flip-flop receives the calculated DWORD value at its data input `D` and outputs it at its output `Q`.
5. The stored value is transmitted via the data connection to the adapter `OUT.D1`. Simultaneously, the adapter's event input `OUT.E1` is activated via the flip-flop's output event (`EO`).

... This ensures that the output value is only updated when there are actual changes or valid new inputs and remains stable until the next event.

Technical Features

- The module uses an internal function block (`ASSEMBLE_DWORD_FROM_WORDS`) for word combination and an edge-triggered D flip-flop (`E_D_FF_ANY`) for output storage.
- Event processing is asynchronous: Each incoming event at either of the two AW adapters triggers a new calculation and subsequent output. The state of the input events is not stored – the flip-flop always uses the last calculated DWORD.
- Word composition is performed at the hardware level: WORD_00 = low-order 16-bit word (bits 0-15), WORD_01 = high-order 16-bit word (bits 16-31).

State Overview

The function block itself does not have an explicit state machine. However, its internal process can be characterized by the states of the D flip-flop:

State	Description

Waiting for Event	The flip-flop holds the last calculated value; no new input event is pending.
Calculation Active	An event from WORD_00 or WORD_01 triggers the merge, and the flip-flop is clocked.
Output Active	After the clock cycle, the new value is passed to `OUT`, and the output event is sent.

The state transitions occur strictly according to the event chain.

Application Scenarios

- **Data Format Conversion**: Combining two 16-bit sensor values (e.g., from two separate analog input modules) into a 32-bit data word for a higher-level controller.
- **Adapter Integration**: Integrating AW-based components (e.g., word adapters from bus systems) into systems that expect an AD adapter (DWORD).
- **Interface Adaptation**: Simplifying wiring when two logically related 16-bit channels need to be combined into a double word.

Comparison with Similar Function Blocks

- **ASSEMBLE_DWORD_FROM_WORDS** (internal FB): Offers the same data merging, but without adapter interfaces and without output storage. `ASSEMBLE_AD_FROM_AW` extends this functionality with adapter encapsulation and a D flip-flop.
- **SPLIT_AD_TO_AW**: Performs the reverse operation (splitting a DWORD into two words) and provides the values via an AW adapter.
- **MUX_WORDS_TO_DWORD**: An alternative function block for word combination that typically operates without an adapter or edge-triggered storage.

Conclusion

ASSEMBLE_AD_FROM_AW` is a specialized function block for cleanly encapsulating word-to-double-word conversion in an adapter-based environment. The combination of a pure arithmetic unit and edge-triggered storage ensures stable output values despite changing input events. This function block is particularly suitable for modular automation solutions that rely on unidirectional adapter interfaces.

ASSEMBLE_AD_FROM_AX



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASSEMBLE_AD_FROM_AX** is used to combine up to 32 Boolean signals, provided via AX adapters (type: `adapter::types::unidirectional::AX`), into a 32-bit double word (DWORD) and output it via an AD adapter (type: `adapter::types::unidirectional::AD`). This enables the compact transmission of multiple discrete binary signals over a single data connection.

Interface Structure

The function block has only **adapter interfaces** (sockets and plugs). There are no direct event or data inputs/outputs at the top level.

Event Inputs

No explicit event inputs. Events are received implicitly via the AX adapters:

- Each of the 32 AX adapters (`BIT_00 … BIT_31`) provides an event output (`E1`) that is activated when the value of the associated BOOL signal changes.

Event Outputs

No explicit event outputs. The AD output adapter (`OUT`) triggers an event (`E1`) as soon as the composite DWORD assumes a new value.

Data Inputs

No explicit data inputs. The Boolean input values are obtained via the data ports (`D1`) of the AX adapters:

- `BIT_00.D1` ... `BIT_31.D1`: each a **BOOL** (bit 0 ... bit 31 of the resulting DWORD).

Data Outputs

No explicit data outputs. The composite DWORD is output via the data port (`D1`) of the AD adapter:

- `OUT.D1`: **DWORD** (the 32-bit word composed from the 32 BOOL inputs).

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AX`	`BIT_00` ... `BIT_31`	Socket (Input)	32 Boolean individual signals, each with its own event (data change)
`adapter::types::unidirectional::AD`	`OUT`	Plug (Output)	Output of the compound double word with update event

Functionality

1. **Event Receipt**: As soon as one of the 32 AX adapters reports a change in its Boolean value (event `E1`), the event is forwarded to the internal block `ASSEMBLE_DWORD_FROM_BOOLS.REQ`.
2. **Data Assembly**: The internal block `ASSEMBLE_DWORD_FROM_BOOLS` combines all 32 Boolean values (from `BIT_00.D1` to `BIT_31.D1`) into a single DWORD. Bit 0 corresponds to `BIT_00`, bit 1 to `BIT_01`, and so on.
3. **Storage and Output**: The resulting DWORD is passed to the next internal block, `E_D_FF_ANY`. This edge-triggered flip-flop stores the value and outputs it at its output `Q`. Simultaneously, it generates an event (`E0`) that is signaled externally via the AD adapter (`OUT.E1`) indicating that a new DWORD value is present.
4. **Buffering**: The flip-flop ensures that the output value is only updated when actual changes occur and does not output the same value multiple times in succession for each individual bit event.

Technical Features

- **Adapter-based Interface**: The function block uses only adapters (AX, AD) and no direct input/output pins. This allows for flexible coupling with other adapters in a service-oriented architecture.
- **Event Synchronization**: The integrated `E_D_FF_ANY` (arbitrary-edge E-D flip-flop) acts as a debouncing and synchronization stage. It prevents multiple output events from being generated when several bits are changed simultaneously – the output is updated only once per change cycle.
- **Bit Order**: Bit 0 (LSB) corresponds to adapter `BIT_00`, and bit 31 (MSB) corresponds to adapter `BIT_31`. A consistent assignment must be observed when wiring.
- **Type Hash**: The function block contains an attribute `eclipse4diac::core::TypeHash`, which is used to validate the block definition.

State Overview

The function block itself does not have an explicit state machine. The internal function block `E_D_FF_ANY` implements a simple memory state:

- **State 0**: Flip-flop output `Q` contains the last loaded DWORD value.
- **State Transition**: Upon an event at `ASSEMBLE_DWORD_FROM_BOOLS.CNF`, the new DWORD value is transferred to the flip-flop, and the output is updated.

Application Scenarios

- **Digital Input Summarization**: In a controller, 32 discrete sensors (e.g., limit switches, light barriers) are read via AX adapters. The function block summarizes their states in a DWORD, which can be transmitted as a compact data word via a fieldbus or other interface.
- **Parallel-to-Serial Conversion**: Preparation of parallel binary data for serial transmission, where the DWORD is sent as a single telegram.
- **Status Query**: A central function block (FB) regularly queries the output adapter and receives the current overall status of all 32 binary digits at once.

Comparison with Similar Function Blocks

- **ASSEMBLE_DWORD_FROM_BOOLS**: This function block has direct BOOL inputs and a DWORD output, but no adapter interface. `ASSEMBLE_AD_FROM_AX` encapsulates this logic in an adapter-based component and adds flip-flop memory.
- **AD_TO_AX_SPLITTER**: The reverse process—splitting a DWORD into individual BOOL signals—is enabled by a corresponding splitter function block.
- **Direct Bit Manipulation**: In some environments, the bits could also be assembled using logical operations. However, the FB described here offers a standardized, reusable, and event-driven solution.

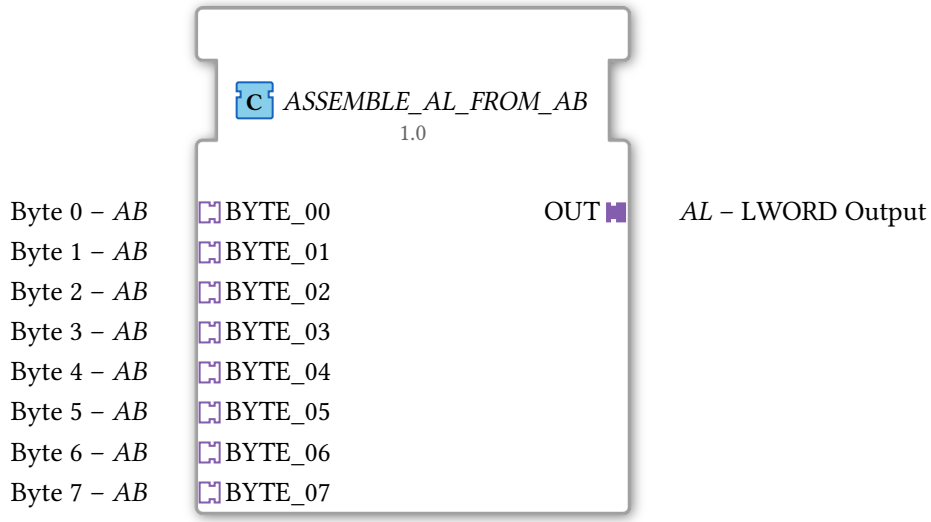
Conclusion

The `ASSEMBLE_AD_FROM_AX` is a useful building block for efficiently bundling a large number of binary signals into a single data word. The use of adapters allows for loose coupling and facilitates integration into service-oriented automation architectures. Integrated flip-flop synchronization prevents unnecessary output events and ensures a stable, up-to-date overall value. Its clear structure makes it particularly suitable for applications where many discrete signals need to be centrally acquired and processed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

ASSEMBLE_AL_FROM_AB



• – ◦ • – ◦ • – ◦ •

Introduction

The function block ASSEMBLE_AL_FROM_AB combines eight individual byte pieces of information from adapters of type AB (unidirectional) into a single, contiguous 64-bit data word (LWORD). The result is output via an output adapter of type AL. This function block serves as a central repository for byte-oriented data and enables modular and clear data structuring.

Interface Structure

Event Inputs

None. Event control is handled exclusively via the connected adapters.

Event Outputs

None. The output event is provided via the adapter OUT.

Data Inputs

None. Data inputs are supplied via the byte adapters.

Data Outputs

None. The composite LWORD is output via the adapter OUT.

Adapters

Sockets (Input Adapters)

Name	Data Type	Description
BYTE_00	adapter::types::unidirectional::AB	Byte 0 (least significant byte)
BYTE_01	adapter::types::unidirectional::AB	Byte 1
BYTE_02	adapter::types::unidirectional::AB	Byte 2
BYTE_03`	adapter::types::unidirectional::AB	Byte 3
BYTE_04`	adapter::types::unidirectional::AB	Byte 4
BYTE_05`	adapter::types::unidirectional::AB	Byte 5
BYTE_06`	adapter::types::unidirectional::AB	Byte 6
BYTE_07`	adapter::types::unidirectional::AD	Byte 7 (most significant byte)

Each byte adapter provides one event (E1) and one data byte (D1).

Plugs (Output Adapters)

Name	Data Type	Description
OUT	adapter::types::unidirectional::AL	Output: composite LWORD; event output via E1, data value via D1

Functionality

The function block is implemented as a composite FB and contains two internal function blocks: ASSEMBLE_LWORD_FROM_BYTES and E_D_FF_ANY.

1. **Data Collection**: As soon as one of the eight byte adapters (BYTE_00 ... BYTE_07) delivers an event at its output E1, this event is forwarded to the input REQ of ASSEMBLE_LWORD_FROM_BYTES. Simultaneously, the corresponding byte values from each adapter are routed to the appropriate data inputs of the assembler block (BYTE_00 ... BYTE_07).
2. **Conversion**: The internal component ASSEMBLE_LWORD_FROM_BYTES combines the eight bytes in the order BYTE_00 to BYTE_07 into a 64-bit word (LWORD). BYTE_00 is the least significant byte, and BYTE_07 is the most significant. After completion of the operation, an acknowledgment event (CNF) is issued, and the constructed LWORD is made available at its data output.
3. **Buffering and Output**: The event CNF triggers the edge-triggered flip-flop E_D_FF_ANY. This flip-flop takes the current LWORD value and stores it temporarily. On each rising edge at the clock input (CLK),

the value at the data output `Q` is updated. The output `Q` is connected to the data output `D1` of the adapter plug `OUT`. The stored event `E0` of the flip-flop is provided as the output event `E1` of the adapter plug `OUT`.

Thus, with each update of one of the eight byte inputs, the entire LWORD is recalculated and output via the output adapter.

Technical Features

- **Adapter-based interfaces**: The component uses adapters exclusively for input and output. This enables flexible and loose coupling with other components that use the same adapter type.
- **Composite Architecture**: The internal structure is built from proven standard components (`ASSEMBLE_LWORD_FROM_BYTES` and `E_D_FF_ANY`). This hides complexity and increases maintainability.
- **Flip-Flop Buffering**: The `E_D_FF_ANY` buffers the assembled value and only releases it after assembly is complete. This prevents inconsistent output during individual byte updates and ensures that a complete data value is always available.
- **Event Synchronization**: All byte adapter signals are routed to a common `REQ` input. This ensures that any change to any byte is processed immediately. However, the merging of all bytes always occurs based on the currently available values.

State Overview

Since this is a composite function block (FB), it does not have its own state machine. The internal `E_D_FF_ANY` has an internal state: the stored LWORD value. This value only changes on rising edges at the clock input, i.e., after each successful assembly. The state is retained until another assembly is triggered.

Application Scenarios

- **Data Aggregation in Sensor Networks**: Several byte-oriented sensors (e.g., temperature, pressure, humidity) are connected via separate adapters. The function block combines them into a 64-bit data telegram, which is forwarded to a central controller.
- **Protocol Conversion**: A byte stream received via multiple adapters (e.g., a serial interface with 8 payload bytes) is converted into a single LWORD, which is easier to handle for further processing.
- **Modular I/O Expansion**: In a modular automation solution, eight digital/analog byte inputs (e.g., via I/O modules) can be combined into a single 64-bit variable.

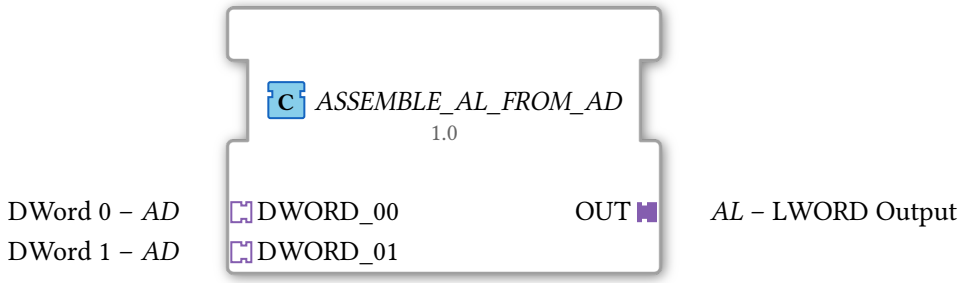
Comparison with Similar Components

- **`ARRAY_TO_LWORD` Components**: These components typically use arrays as inputs and have a different interface (no adapters). The `ASSEMBLE_AL_FROM_AB` offers an adapter-based, event-driven alternative that fits better into modular, typed adapter structures.
- **Multiplexer (MUX)**: A multiplexer selects one of several inputs, while this component combines all inputs simultaneously into a larger data word.
- **Direct Bit Concatenation**: Without this component, byte concatenation would have to be performed manually using bitwise shifting and OR operations. The `ASSEMBLE_AL_FROM_AB` encapsulates this logic and reduces development and testing effort.

Conclusion

The `ASSEMBLE_AL_FROM_AB`This is a specialized function block for the modular and event-driven aggregation of eight-byte data sets into a single LWORD. Its exclusive use of adapters makes it ideal for use in adapter-based architectures and promotes a clear separation of data sources and sinks. Integrated buffering via the flip-flop and triggering by each incoming event ensure that a consistent, complete data value is always output. This component significantly simplifies data aggregation and contributes to maintainable and reusable control logic.

ASSEMBLE_AL_FROM_AD



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASSEMBLE_AL_FROM_AD** is used to combine two 32-bit values provided via AD adapters (DWORD) into a 64-bit LWORD and output this LWORD via an AL adapter. It encapsulates the combination logic and provides a buffered, event-driven interface for 64-bit data.

Interface Structure

Event Inputs

- **DWORD_00.E1** – Event from the first AD adapter that triggers the combination.
- **DWORD_01.E1** – Event from the second AD adapter that also triggers the combination.

Event Outputs

- **OUT.E1** – Event at the AL adapter indicating that a new LWORD value is present at the output.

Data Inputs

- **DWORD_00.D1** – First 32-bit DWORD (lower-order part).
- **DWORD_01.D1** – Second 32-bit DWORD (higher-order part).

Data Outputs

- **OUT.D1** – Composite 64-bit LWORD.

Adapters

- **DWORD_00** (Socket, type `adapter::types::unidirectional::AD`) – Input adapter for the first DWORD.
- **DWORD_01** (Socket, type `adapter::types::unidirectional::AD`) – Input adapter for the second DWORD.
- **OUT** (Plug, type `adapter::types::unidirectional::AL`) – Output adapter for the resulting LWORD.

Functionality

The FB operates internally as an FB network consisting of two sub-blocks:

1. **ASSEMBLE_LWORD_FROM_DWORDS** – Combines the two DWORD inputs `DWORD_00` and `DWORD_01` into one LWORD. The function block is triggered by the event `REQ`, which can be sent by **both** AD adapters (via `DWORD_00.E1` and `DWORD_01.E1`). After the combination is complete, the event `CNF` is output.
2. **E_D_FF_ANY** – Any D flip-flop that buffers the combined LWORD. The `CNF` event clocks the flip-flop (`CLK`), with the combined value present at the data input `D`. The stored value appears at output `Q` and is passed on via `OUT.D1`. Simultaneously, the flip-flop output event `EO` is triggered as `OUT.E1`.

Thus, the LWORD is only updated when one of the two AD adapters sends an event – the flip-flop prevents multiple or unwanted outputs when the inputs remain unchanged.

Technical Features

- **Event-driven processing** – The combination is only triggered by incoming events from the AD adapters, thus avoiding unnecessary processing load.

- **Buffered output** – The D flip-flop ensures that a calculated LWORD remains stable at the output until the next update.
- **Independence of inputs** – Each of the two DWORD adapters can send an event at any time; The combination is then performed using the currently available values.
- **No dedicated event inputs at the top level** – The function block reacts exclusively via the adapter interfaces.

State Overview

The function block does not use an explicit ECC, but rather an internal function block network. The functional sequence is:

1. **Waiting** for an event from `DWORD_00.E1` or `DWORD_01.E1`.
2. **Combining** – ASSEMBLE_LWORD_FROM_DWORDS processes the current DWORD values and creates the LWORD.
3. **Buffering** – The flip-flop takes the LWORD and outputs it via `OUT.D1`.
4. **Signaling** – The event `OUT.E1` is triggered.
5. Return to wait state.

A simultaneous request from both inputs results in only one processing step, as both events trigger the same `REQ` function of the combination block.

Application Scenarios

- **64-bit Data Processing** – When a system operates with 32-bit interfaces and large values (e.g., high-resolution timestamps, counter readings, physical measurements) need to be combined into a single LWORD.
- **Protocol Conversion** – In fieldbuses or controllers that deliver data in separate DWORD segments and require 64-bit processing.
- **Secure Data Combination** – The buffered output prevents intermediate values from occurring with asynchronously arriving DWORDs.

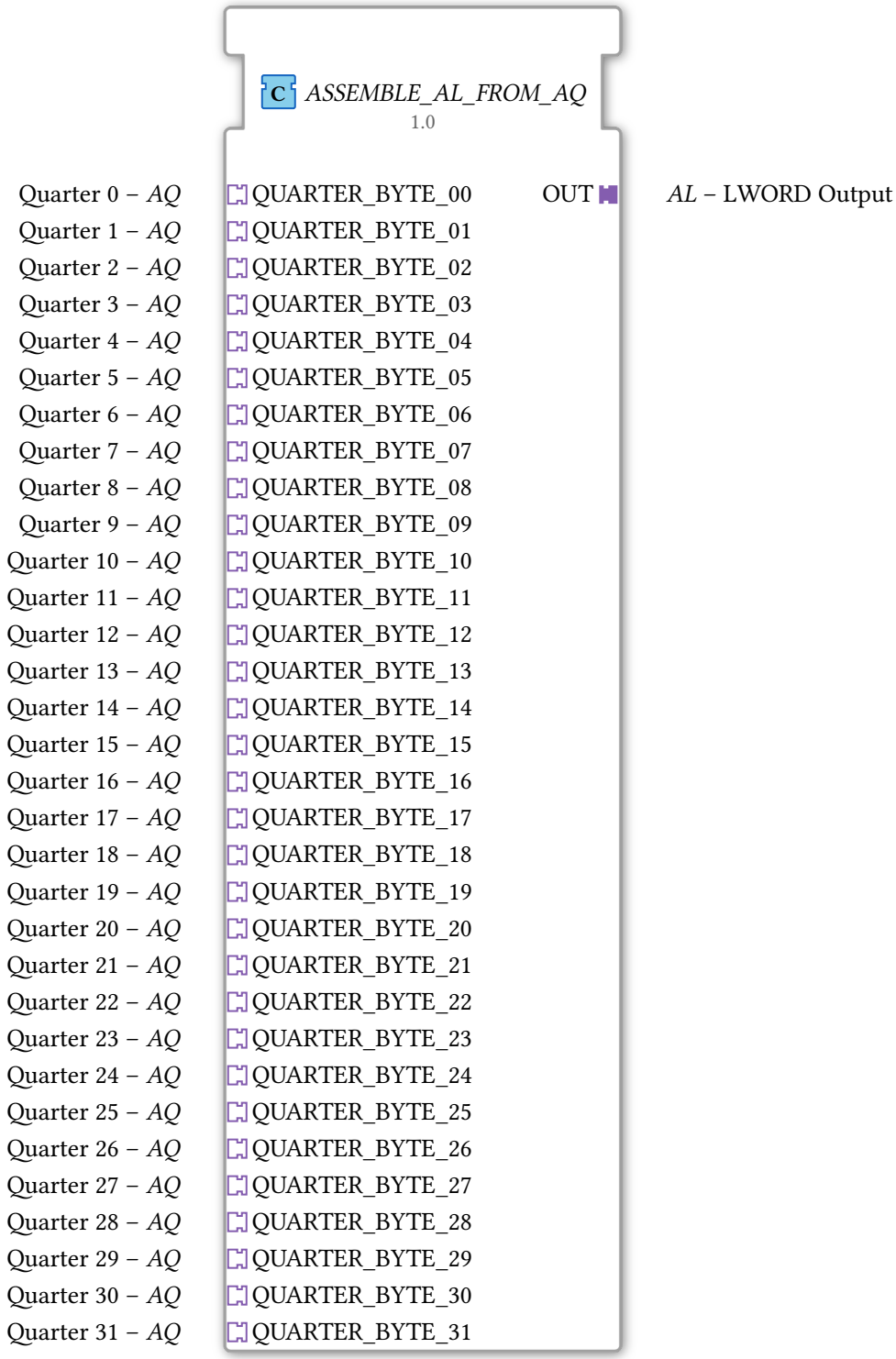
Comparison with Similar Function Blocks

- **ASSEMBLE_DWORD_FROM_WORDS** – Combines two WORDs into one DWORD; analogous principle, but for different word widths.
- **ASSEMBLE_LWORD_FROM_DWORDS** – The internal function block that performs the pure combination without buffering; ASSEMBLE_AL_FROM_AD adds event-driven buffering and the adapter interface.
- **Direct Use of a Type Converter** – Would be possible, but without event synchronization; the present function block offers a clean, industrialized solution with adapters.

Conclusion

The **ASSEMBLE_AL_FROM_AD** function block is a specialized but useful building block for combining two DWORDs into one LWORD. It combines event-driven processing with stable output buffering and is ideal for control systems that exchange 64-bit data via standardized adapters. The use of adapters allows for easy integration into existing model structures.

ASSEMBLE_AL_FROM_AQ



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_AL_FROM_AQ` is used to combine 32 unidirectional AQ adapters (“quarters”) into a single AL adapter (LWORD). It is implemented as a pure composition block without its own event or data ports at the top level and uses internal sub-function blocks to combine the incoming quarter data into a 64-bit word and stabilize it on the output side.

Interface Structure

Event Inputs

The block has **no** its own event inputs at its top level.
Event control is handled exclusively via the integrated AQ adapters (sockets) and the internal wiring.

Event Outputs

There are **no** event output ports.
The output signal is transmitted externally via the AL plug adapter (E1) in an event-driven manner.

Data Inputs

There are **no** directly accessible data inputs.
Data is fed into the module via the 32 AQ adapters (sockets).

Data Outputs

There are **no** directly accessible data outputs.
The composite LWORD is output via the AL plug adapter (D1).

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AQ`	`QUARTER_BYTE_00` ... `QUARTER_BYTE_31`	Socket (Input)	32 identical adapters, each providing a 2-bit value (“quarter”). Each socket has an event output (E1) and a data output (D1).
`adapter::types::unidirectional::AL`	`OUT`	Plug (Output)	Output adapter that passes a 64-bit LWORD (D1) and an associated event (E1).

Functionality

1. **Data Acquisition** – Each of the 32 `QUARTER_BYTE_xx` sockets provides its data value (D1) upon receiving an incoming event (E1).
2. **Assembly** – The event is forwarded to the internal sub-function block `ASSEMBLE_LWORD_FROM_QUARTERS` (all 32 events are connected to the same `REQ` input). Simultaneously, all 32 quarter data values are routed to the corresponding inputs of this sub-block.
3. **Internal Processing** – `ASSEMBLE_LWORD_FROM_QUARTERS` combines the 32 2-bit quarters into a 64-bit LWORD. Its output signal is then routed through an edge-triggered D flip-flop (`E_D_FF_ANY`).
4. **Output Stabilization** – The flip-flop is clocked by the completion event (`CNF`) of `ASSEMBLE_LWORD_FROM_QUARTERS`. It holds the combined value until a new assembly cycle is complete. The flip-flop’s output feeds the data port `OUT.D1` of the AL plug adapter, and the corresponding event `OUT.E1` is triggered after the clock signal.

Technical Features

- **Pure Adapter Composition** – The component has no conventional input/output ports at the top level, but uses only adapters. This enables modular encapsulation and easy reuse in different contexts.
- **Internal Data Aggregation** – The 32 quarter values (2 bits each) are combined into a 64-bit LWORD. This fully utilizes the LWORD capacity.
- **Edge-Triggered Output Synchronization** – The D flip-flop prevents flickering or incomplete data words at the output by only passing the value after the aggregation is complete and on a rising clock edge.
- **No Internal State Logic** – The component does not contain an ECC (Execution Control Chart); the logic is entirely derived from the interconnection of the sub-blocks.

State Overview

The component does not have its own state machine.

The state logic is implemented by the internal blocks ASSEMBLE_LWORD_FROM_QUARTERS (data-driven) and E_D_FF_ANY (event-driven).

Application Scenarios

- **Combining Bit Subwords** – In industrial control engineering, data from multiple sensors or subsystems (e.g., 32 switching states, each encoding 2 bits) often needs to be combined into a single, coherent data word.
- **Querying Parallel Data Sources** – If 32 independent modules each deliver a 2-bit signal via adapters, ASSEMBLE_AL_FROM_AQ can combine these into a 64-bit word in one step and provide it in an event-driven manner.
- **Platform-Independent Adapter Interfaces** – Thanks to its pure adapter usage, this component is suitable for use in heterogeneous systems where data types are abstracted by the adapter definitions.

Comparison with Similar Components

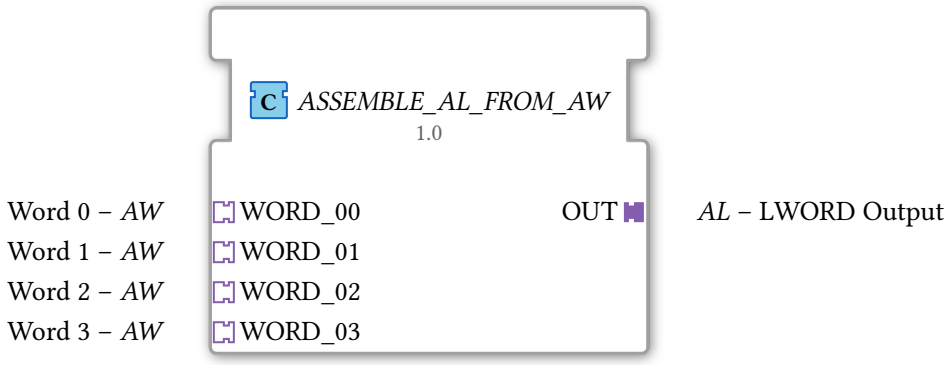
Component	Operating Principle	Number of Inputs	Output Type	Special Feature
ASSEMBLE_AL_FROM_AQ	Adapter-Based Composition	32 × 2-Bit (AQ)	1 × LWORD (AL)	Edge-Triggered Output, Pure Composition
ASSEMBLE_LWORD_FROM_QUARTERS (internal)	Data-Oriented Composition	32 × 2-Bit (direct)	LWORD (Data)	No adapters, no flip-flop
Classic multiplexer (e.g., MUX)	Input selection via control line	n inputs, 1 selection	Simple data type	Requires dedicated address signals

Compared to a multiplexer, ASSEMBLE_AL_FROM_AQ offers the advantage that *all* quarter values are combined in parallel and without selection logic to form a complete word. The additional flip-flop ensures clean, event-driven output.

Conclusion

The ASSEMBLE_AL_FROM_AQ function block provides an elegant and modular solution for combining 32 2-bit quarter data values into a 64-bit LWORD using adapters. The strict separation of data and event paths, as well as the internal use of an edge-triggered flip-flop, ensures reliable and deterministic output signal generation. Due to its purely compositional structure, it is particularly suitable for use in IEC 61499 systems that rely on adapter-based interfaces.

ASSEMBLE_AL_FROM_AW



• – ◦ • – ◦ • – ◦ •

Introduction

The function block ASSEMBLE_AL_FROM_AW combines the data from four **AW (WORD)** adapter interfaces into one **AL (LWORD)** output value. It is used to combine multiple 16-bit words into a 64-bit word and output it via a clocked signal.

Interface Structure

Event Inputs

None – event control is handled exclusively via the connected adapters (see section Adapters).

Event Outputs

None – event output is provided via the OUT adapter (`OUT.E1`).

Data Inputs

No direct data inputs – the data to be processed is obtained via the four AW sockets.

Data Outputs

No direct data outputs – the result is output via the OUT adapter.

Adapters

Name	Type	Comment
Sockets		
`WORD_00`	`adapter::types::unidirectional::AW`	Word 0 (least significant 16-bit word)
`WORD_01`	`adapter::types::unidirectional::AW`	Word 1
`WORD_02`	`adapter::types::unidirectional::AW`	Word 2
`WORD_03`	`adapter::types::unidirectional::AW`	Word 3 (highest-order 16-bit word)
Plugs		
`OUT`	`adapter::types::unidirectional::AL`	LWORD output (64-bit)

Functionality

The module utilizes an internal network of two function blocks:

1. **`ASSEMBLE_LWORD_FROM_WORDS`** – performs the logical combination of the four 16-bit words (`WORD_00` to `WORD_03`) into a 64-bit LWORD.
2. **`E_D_FF_ANY`** – a clock-edge-triggered D flip-flop that only transmits the combined value to the output adapter `OUT` when a rising edge is present at the clock input.

The events from the four input adapters (`WORD_00.E1` … `WORD_03.E1`) are all connected to the `REQ` event input of `ASSEMBLE_LWORD_FROM_WORDS`. As soon as an event arrives at one of the input adapters, the combination is recalculated. Upon completion (`CNF`), `E_D_FF_ANY` is clocked, which then adopts the calculated value into its internal state and outputs it via `OUT.D1`. The event at the output adapter (`OUT.E1`) signals the availability of the new value.

As soon as an event arrives at one of the input adapters, the combination is recalculated.

Technical Features

- **Clock Edge Triggered Output** – The use of the `E_D_FF_ANY` prevents unwanted intermediate values and ensures that only fully composed LWORDs are output.
- **Arbitrary Order of Input Events** – Since all four sockets are combined into the same `REQ` input, each incoming event triggers a recalculation. The function block therefore reacts to each partial word, but only updates the output after the complete composition.
- **Adapter-Based Interface** – The use of unidirectional adapters (AW and AL) enables loose coupling and reuse in different contexts.

State Overview

The function block does not have its own ECC (Execution Control Chart), but consists of a combinational circuit with a downstream flip-flop. The internal state logic is fully implemented by the two sub-modules `ASSEMBLE_LWORD_FROM_WORDS` and `E_D_FF_ANY`.

Application Scenarios

- **Industrial Data Acquisition** – Combining four 16-bit sensor values (e.g., temperature, pressure, flow rate, level) into a single 64-bit data packet.
- **Communication Interfaces** – Processing separately arriving 16-bit data blocks for a 64-bit interface (e.g., Modbus register ⟶ OPC UA).
- **Bit-Parallel Processing** – Combining four microcontroller registers into a single large register for advanced logic applications.

Comparison with Similar Modules

Module	Description
`ASSEMBLE_AL_FROM_BYTES`	Combines multiple 8-bit values into an LWORD
`ASSEMBLE_AL_FROM_DWORDS`	Combines two 32-bit double words into a 64-bit word
`ASSEMBLE_AL_FROM_AW` (this one)	Combines four 16-bit words into an LWORD

All these building blocks follow the same principle but differ in the word width of the input data and the number of sockets required.

Conclusion

ASSEMBLE_AL_FROM_AW` is a specialized function block for concatenating four 16-bit words into a single 64-bit word. Clocked output via a D flip-flop ensures consistent data transmission. Thanks to its adapter-based interface, it integrates seamlessly into modular 4diac projects and simplifies word-to-long-word conversions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 MCU vs. MPU Comparison Guide on ms-muc-docs.de

]

ASSEMBLE_AL_FROM_AX

(Image of the function block is displayed here in the IDE – shows a function block with 64 sockets (left) and one plug (right))


ASSEMBLE_AL_FROM_AX
1.0
Bool Bit 0 – AX
BIT_00
OUT
AL – LWORD Output
Bool Bit 1 – AX
BIT_01
Bool Bit 2 – AX
BIT_02
Bool Bit 3 – AX
BIT_03
Bool Bit 4 – AX
BIT_04
Bool Bit 5 – AX
BIT_05
Bool Bit 6 – AX
BIT_06
Bool Bit 7 – AX
BIT_07
Bool Bit 8 – AX
BIT_08
Bool Bit 9 – AX
BIT_09
Bool Bit 10 – AX
BIT_10
Bool Bit 11 – AX
BIT_11
Bool Bit 12 – AX
BIT_12
Bool Bit 13 – AX
BIT_13
Bool Bit 14 – AX
BIT_14
Bool Bit 15 – AX
BIT_15
Bool Bit 16 – AX
BIT_16
Bool Bit 17 – AX
BIT_17
Bool Bit 18 – AX
BIT_18
Bool Bit 19 – AX
BIT_19
Bool Bit 20 – AX
BIT_20
Bool Bit 21 – AX
BIT_21
Bool Bit 22 – AX
BIT_22
Bool Bit 23 – AX
BIT_23
Bool Bit 24 – AX
BIT_24
Bool Bit 25 – AX
BIT_25
Bool Bit 26 – AX
BIT_26
Bool Bit 27 – AX
BIT_27
Bool Bit 28 – AX
BIT_28
Bool Bit 29 – AX
BIT_29
Bool Bit 30 – AX
BIT_30
Bool Bit 31 – AX
BIT_31
Bool Bit 32 – AX
BIT_32
Bool Bit 33 – AX
BIT_33
Bool Bit 34 – AX
BIT_34
Bool Bit 35 – AX
BIT_35
Bool Bit 36 – AX
BIT_36
Bool Bit 37 – AX
BIT_37
Bool Bit 38 – AX
BIT_38
Bool Bit 39 – AX
BIT_39
Bool Bit 40 – AX
BIT_40
Bool Bit 41 – AX
BIT_41
Bool Bit 42 – AX
BIT_42
Bool Bit 43 – AX
BIT_43
Bool Bit 44 – AX
BIT_44
Bool Bit 45 – AX
BIT_45
Bool Bit 46 – AX
BIT_46
Bool Bit 47 – AX
BIT_47
Bool Bit 48 – AX
BIT_48
Bool Bit 49 – AX
BIT_49
Bool Bit 50 – AX
BIT_50
Bool Bit 51 – AX
BIT_51
Bool Bit 52 – AX
BIT_52
Bool Bit 53 – AX
BIT_53
Bool Bit 54 – AX
BIT_54
Bool Bit 55 – AX
BIT_55
Bool Bit 56 – AX
BIT_56
Bool Bit 57 – AX
BIT_57
Bool Bit 58 – AX
BIT_58
Bool Bit 59 – AX
BIT_59
Bool Bit 60 – AX
BIT_60
Bool Bit 61 – AX
BIT_61
Bool Bit 62 – AX
BIT_62
Bool Bit 63 – AX
BIT_63

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASSEMBLE_AL_FROM_AX** is used to combine up to 64 Boolean signals from **AX adapters** (type: `adapter::types::unidirectional::AX`) into a single **LWORD** value and output it via an **AL adapter** (type: `adapter::types::unidirectional::AL`). The block encapsulates the logic for bit combination and provides the result stably as soon as one of the input bits returns an event.

Interface Structure

Event Inputs

The function block does not have traditional event inputs (EVENT). Event control is handled indirectly via the **AX adapters (sockets)**. Each socket `BIT_00` ... `BIT_63` can receive a triggering event (E1).

Name	Type	Description
`BIT_00` ... `BIT_63`	AX adapter	Event from an external Boolean source that triggers the assembly of the LWORD

Event Outputs

No traditional event outputs. The output of the assembled LWORD is event-driven via the **AL adapter (plug)** `OUT`.

| Name | Type | Description | | |—|—|—————| | `OUT` | AL Adapter | Provides the composite LWORD when the internal flip-flop is clocked |

Data Inputs

Each AX adapter carries a Boolean data value (D1). These are used as input bits for the LWORD.

Name	Type	Description
`BIT_00.D1` ... `BIT_63.D1`	BOOL (via adapter)	Boolean signal for bits 0 ... 63

Data Outputs

The combined result is output as a 64-bit word via the AL adapter.

Name	Type	Description
`OUT.D1`	LWORD (via adapter)	Combined LWORD from the 64 Boolean inputs

Adapter

Type	Direction	Name	Description
`adapter::types::unidirectional::AX`	Socket (input)	`BIT_00` ... `BIT_63`	Boolean Input Adapter
`adapter::types::unidirectional::AL`	Plug (Output)	`OUT`	LWORD Output Adapter

Functionality

1. **Event Reception**: As soon as an event arrives at one of the 64 AX adapters (sockets) (at its E1), an internal REQ event is sent to the function block `ASSEMBLE_LWORD_FROM_BOOLS`.
2. **Bit Assembly**: The internal function block `ASSEMBLE_LWORD_FROM_BOOLS` receives the 64 Boolean values from the data ports `BIT_00.D1` ... `BIT_63.D1` and assembles them into an LWORD value. Each input is assigned a corresponding bit (0 for BIT_00, 1 for BIT_01, ... 63 for BIT_63).
3. **Central Storage (Flip-Flop)**: The assembled LWORD is passed to the data input of an **E_D_FF_ANY** (flip-flop). The result event (CNF) of `ASSEMBLE_LWORD_FROM_BOOLS` clocks the flip-flop. This sets the current LWORD value at the flip-flop's output Q.
4. **Output**: The stored value is made available as an LWORD via the AL plug `OUT`. Simultaneously, an event is generated at the flip-flop's output (EO), which triggers the connected AL adapter input (E1).

The logic operates **event-driven**: The output is only updated when any input bit changes (via the event). This saves processing power and ensures a stable output.

Technical Features

- **Full Bit Mapping**: All 64 bits of an LWORD are derived from individual Boolean adapters. Exactly 64 sockets are required – no gaps.
- **Event Synchronization**: An event from *any* bit adapter triggers a complete recalculation of the LWORD. It is not necessary to update all inputs simultaneously.
- **Flip-Flop Buffering**: The composite value is buffered, so the output remains stable even if the inputs fluctuate during processing.
- **Adapter-based interface**: The inputs and outputs are implemented as standardized 4diac adapters, enabling easy chaining with other adapter-based function blocks.
- **Packages**: The function block is organized in the package `adapter::assembling`.

State overview

The function block does not have an explicit state machine. The internal logic operates as a combination of two components:

- **ASSEMBLE_LWORD_FROM_BOOLS**: Stateless – executes the assembly on every request.
- **E_D_FF_ANY**: Has two states: stored value (Q) and current data input (D). The state changes on every rising edge at the CLK input.

Event-driven behavior:

- **Wait for event**: No calculation is performed; the output remains unchanged.
- **Event Arrived**: Flip-flop assembly + clocking ⟶ output is updated.

Application Scenarios

- **Parallel Digital Inputs**: Combines 64 digital sensors or switches into a machine-readable LWORD for easy further processing.
- **Bit-Oriented Controls**: In applications where many binary status messages (e.g., error bits, position feedback) need to be bundled.
- **Fieldbus Communication**: An LWORD can be efficiently transmitted via PROFINET, EtherCAT, or similar; this function block converts the individual bits into the compact data format.
- **Adapter-Based Module Libraries**: In modular control architectures based on standardized adapters, this function block can serve as a universal "bit collector."

Comparison with Similar Function Blocks

Function Block	Number of Inputs	Output Type	Special Feature
ASSEMBLE_AL_FROM_AX	64 BOOL (AX)	LWORD (AL)	Adapter-based with flip-flop buffering
`ASSEMBLE_DWORD_FROM_BOOLS`	32 BOOL	DWORD	Classic data input/output, without adapter
`ASSEMBLE_WORD_FROM_BOOLS`	16 BOOL	WORD	As above, for 16-bit word
`DISASSEMBLE_AL_TO_AX`	1 LWORD (AL)	64 AX	Counterpart: Decomposes an LWORD into individual BOOL adapters

The key difference lies in the **adapter interface** and the **buffer buffering** provided by the flip-flop, which ensures robust behavior in event-driven environments.

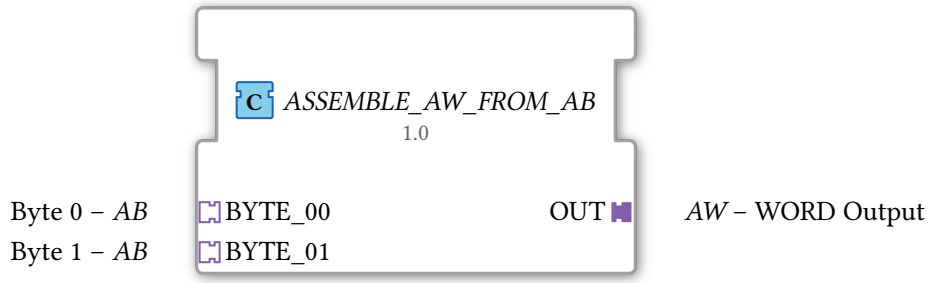
Conclusion

The **ASSEMBLE_AL_FROM_AX** function block is a powerful tool for creating a compact 64-bit LWORD from 64 distributed Boolean signals. Its adapter-based interface and internal flip-flop buffering make it particularly suitable for clean, event-driven, and modular automation solutions. Full bit coverage without scaling issues makes it a universal solution for applications that need to bundle many digital inputs.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

ASSEMBLE_AW_FROM_AB



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASSEMBLE_AW_FROM_AB** is used to combine two unidirectional byte adapters (type `AB`) into a unidirectional word adapter (type `AW`). It combines the data from two byte inputs into a word output, with event logic ensuring that output occurs only when the information is updated.

Interface Structure

The function block has no direct event or data inputs/outputs, but communicates exclusively via adapter interfaces:

Adapter (Plugs – Output)

Name	Type	Direction	Description
OUT	`adapter::types::unidirectional::AW`	Plug	WORD output (16-bit)

Adapter (Sockets – Inputs)

Name	Type	Direction	Description
BYTE_00	`adapter::types::unidirectional::AB`	Socket	Byte 0 (low-order byte)
BYTE_01	`adapter::types::unidirectional::AB`	Socket	Byte 1 (higher-order byte)

Note: Adapters of type `AB` and `AW` are unidirectional. Each adapter contains the signals `E1` (socket/plug event output – internally connected) and `D1` (data signal).

Functionality

The function block (FB) is based internally on two sub-components:

1. **`ASSEMBLE_WORD_FROM_BYTES`** – a predefined assembler that combines two input bytes into a WORD.
2. **`E_D_FF_ANY`** – a D flip-flop that temporarily stores the combined value and passes it on at a clock event.

Process:

- Each incoming event (`E1`) from a BYTE adapter (BYTE_00 or BYTE_01) triggers the assembler (`REQ`).
- The assembler combines the current data from both BYTE inputs into a WORD.

The following applies:
OUT.WORD = (BYTE_01.D1 << 8) | BYTE_00.D1`

- After successful combination, the assembler sends an acknowledgment event (`CNF`).
- This event clocks the D flip-flop (`CLK`), which takes the current WORD value.
- The flip-flop output (`Q`) is continuously passed to the OUT adapter (`D1`).
- The flip-flop's output event (`EO`) activates `OUT.E1`, informing the receiving component about the updated WORD output.

Important feature:

Since both BYTE events lead to the same `REQ` input in the assembler, assembly is re-executed with each event from either input. The output value is therefore always generated from the currently present byte values. The D flip-flop ensures a stable output until the next assembly event arrives.

Important feature: Since both BYTE events lead to the same `REQ` input in the assembler, the assembly is re-executed with each event from either input.

Technical Features

- **Pure Adapter Communication:** The function block (FB) has no conventional input/output variables; all data and event transmission occurs via unidirectional adapters.
- **Event Synchronization:** The single-step assembly process avoids race situations, as the flip-flop only updates the output after successful computation.
- **Reusability:** The FB can be embedded in any environment that supports unidirectional `AB`/`AW` adapters.

State Overview

The FB does not have an explicit state diagram, as its internal logic operates purely event-driven. Essentially, the following operating states can be distinguished:

- **Ready:** No event is pending; the output holds the last stored WORD value.
- **Assemble:** An incoming event from a BYTE adapter starts the calculation.
- **Output:** After successful assembly, the new WORD value is transferred to the flip-flop, and the output event is triggered.

Application Scenarios

- **Protocol Conversion:** Combining two serial byte streams into a WORD data word for a subsequent processing module.
- **Sensor Fusion:** Combining two 8-bit sensor data (e.g., temperature and humidity) into a 16-bit value.
- **Hardware Control:** Generating a 16-bit output signal from two separate 8-bit register values.

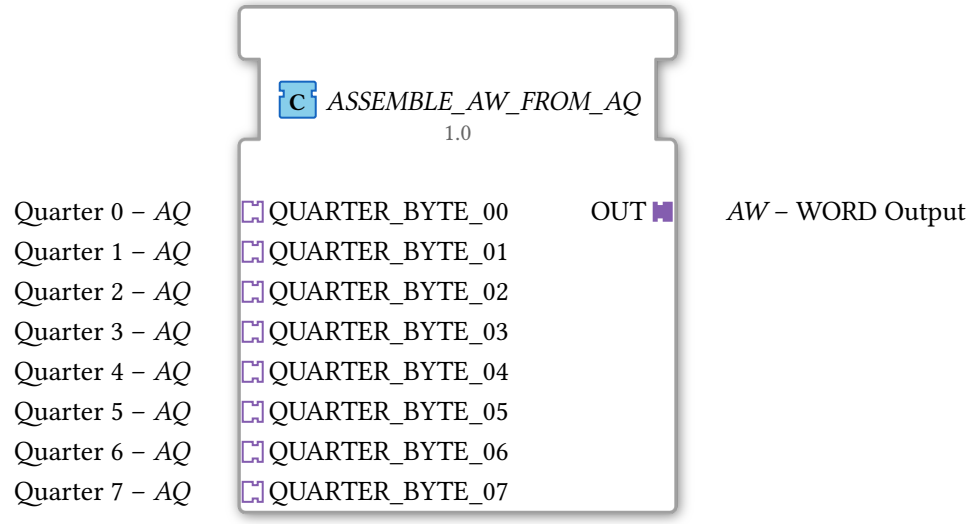
Comparison with Similar Function Blocks

- **`ASSEMBLE_WORD_FROM_BYTES`** – pure assembler without memory. It expects discrete data inputs and outputs the result immediately (provided the event and data are supplied synchronously).
- **`SPLIT_BYTE_FROM_WORD`** – inverse function (WORD -> two BYTEs); symmetrically structured, also uses unidirectional adapters.
- **Custom Function Block with Memory:** This function block integrates the flip-flop, so the output remains stable until new data arrives – unlike function blocks that recalculate with each event but offer no intermediate storage.

Conclusion

The **ASSEMBLE_AW_FROM_AB** function block offers a reliable and elegant way to combine two unidirectional BYTE adapters into a WORD output. The combination of assembly and flip-flop logic creates a stable output that is only updated when data actually changes. Its simple adapter interface makes it flexible for use in modular control architectures according to IEC 61499.

ASSEMBLE_AW_FROM_AQ



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_AW_FROM_AQ` combines eight individual **AQ (Quarter)** adapters into a single **AW (Word)** adapter. Each quarter represents a 4-bit data packet (half a byte). The block assembles these eight quarter data packets—a total of 32 bits—into a complete word (WORD, 32 bits) and makes it available via a single AW output adapter. Typical applications include the reconstruction of wide data buses from serially or partially arriving fragments.

Interface Structure

The function block has only **adapter interfaces**: There are no direct event or data inputs/outputs at the top level. All communication takes place via the connected adapters.

Event Inputs

- No direct event inputs.
- Events are received via the sockets of the **AQ adapters** (each AQ adapter provides an event output `E1`).

Event Outputs

- No direct event outputs.
- The output event is provided via the **AW adapter** (plug `OUT`) as event `E1`.

Data Inputs

- No direct data inputs.
- The data to be processed is read in via the **AQ adapters** as data input `D1` (4 bytes each).

Data Outputs

- No direct data outputs.
- The compound word (4 bytes) is output via the **AW adapter** as data output `D1`.

Adapter

Name	Type	Direction	Comment
`OUT`	adapter::AW	Plug	Word output (4 bytes)
`QUARTER_BYTE_00`	adapter::AQ	Socket	Quarter 0 (least significant)
`QUARTER_BYTE_01`	adapter::AQ	Socket	Quarter 1
`QUARTER_BYTE_02`	adapter::AQ	Socket	Quarter 2
`QUARTER_BYTE_03`	adapter::AQ	socket	Quarter 3
`QUARTER_BYTE_04`	adapter::AQ	socket	Quarter 4
`QUARTER_BYTE_05`	adapter::AQ	socket	Quarter 5
`QUARTER_BYTE_06`	adapter::AQ	socket	Quarter 6
`QUARTER_BYTE_07`	adapter::AQ	socket	Quarter 7 (highest value)

Functionality

The FB contains two internal function blocks:

1. **`ASSEMBLE_WORD_FROM_QUARTERS`**: A specialized assembler that combines the data from all eight quarter-time inputs (4 bytes each) in the order 00 (LSB) to 07 (MSB) into a 32-bit word.
2. **`E_D_FF_ANY`**: An edge-triggered D flip-flop that buffers the assembled word and outputs it at `Q`.

Process:

- As soon as a connected AQ adapter sends an event at its output `E1`, this event is forwarded to the `REQ` input of the assembler. All eight AQ adapters are connected to the same `REQ`, so each incoming event (regardless of the sender) triggers a new processing step.
- The assembler processes the current data from all eight quarters and places the result at the data output.
- After the calculation is complete, the assembler sends an acknowledgment event (`CNF`), which triggers the clock input (`CLK`) of the D flip-flop.

- The flip-flop receives the calculated value and passes it stably to `Q`. The OUT adapter then makes both the event `E1` and the data value `D1` available on the output side.
- After the calculation is complete, the assembler sends an acknowledgment event (`CNF`), which triggers the clock input (`CLK`) of the D flip-flop. This structure ensures that the output word is only updated after a complete calculation has been performed—regardless of which quarter was updated last.

Technical Features

- **Flip-Flop Intermediate Storage**: The use of a D flip-flop prevents glitches or incomplete data transmission if multiple quarters are updated in rapid succession. The output only changes when the assembler has calculated a new, valid value.
- **Adapter-Based Interface**: All inputs and outputs are handled via adapters, making the function block particularly suitable for modular, typed data flow architectures in 4diac.
- **Synchronous Processing**: Processing is triggered by each incoming event of a quarter, but only generates an output event when the calculation is complete.
- **Adapter-Based Interface**:

Status Overview

The function block (FB) does not have its own execution control chart (ECC) – the internal logic is implemented entirely by the included function blocks `ASSEMBLE_WORD_FROM_QUARTERS` and `E_D_FF_ANY`. The function block therefore behaves like a combinational circuit with an edge-triggered memory stage.

Application Scenarios

- **Reconstruction of Wide Data Paths**: In automation technology, data is often transmitted in small packets ("nibbles"), e.g., across multiple parallel I/O modules. The FB reassembles these into a complete 32-bit value.
- **Serial-Parallel Conversion**: If a sensor or actuator delivers its data in 4-bit quads (e.g., via asynchronous interfaces), the FB can buffer the incoming fragments and combine them into a consistent total value.
- **Data Consolidation**: In hierarchical control systems where multiple sub-modules each provide a quarter value, this function block (FB) can centrally consolidate the values.

Comparison with Similar Function Blocks

- **`ASSEMBLE_BYTE_FROM_NIBBLES`**: Combines two half-bytes into one byte (8 bits). This FB operates at the next higher level with 8 quarters (32 bits).
- **Simple Data Converters**: Converters like `WORD_TO_BYTE` or `BYTE_TO_NIBBLE` usually operate without intermediate storage and event handling. `ASSEMBLE_AW_FROM_AQ` offers event-driven, stable forwarding with optional storage.

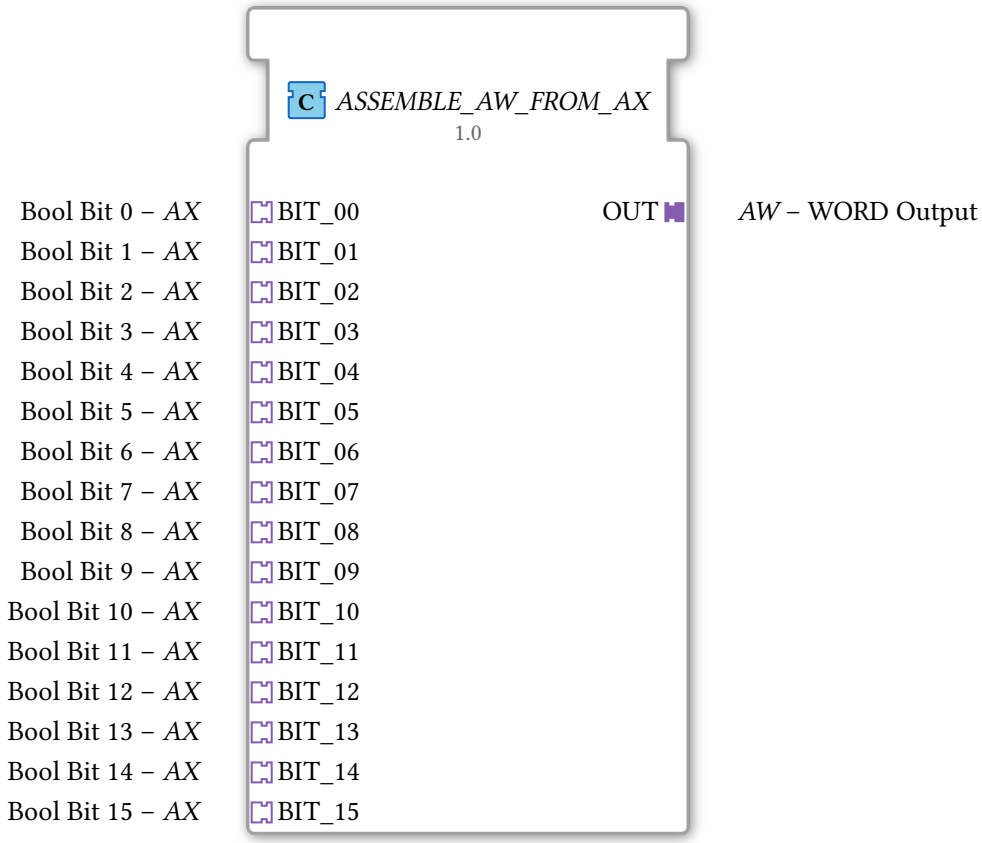
Conclusion

The function block `ASSEMBLE_AW_FROM_AQ` provides a robust and flexible solution for generating a consistent 32-bit word from eight independent quarter data points (4 bits each). Thanks to its adapter-based interfaces and integrated D flip-flop, it is particularly well-suited for real-time applications where data packets arrive asynchronously and must be reliably combined into a single value.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

ASSEMBLE_AW_FROM_AX



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_AW_FROM_AX` combines 16 Boolean signals, provided by separate AX adapters, into a 16-bit word (WORD) and outputs it via an AW adapter. It enables the conversion of a discrete group of digital signals into a uniform data word value for further processing.

Interface Structure

Event Inputs

No direct event inputs. Events are received via the 16 AX adapters (sockets): Each AX adapter triggers an event (`E1`) upon a data change, which initiates processing.

Event Outputs

No direct event outputs. The compound word is output via the AW adapter (Plug). The output adapter releases the data value as soon as it receives an event (`E1`) from the internal flip-flop.

Data Inputs

No direct data inputs. The 16 individual values (BOOL) are read in via the AX adapters (data output `D1`).

Data Outputs

No direct data outputs. The resulting 16-bit word (WORD) is output via the AW adapter (data input `D1`).

Adapters

Type	Name	Description
Socket (Input)	`BIT_00`	AX Adapter, Bool Bit 0
Socket (Input)	`BIT_01`	AX Adapter, Bool Bit 1

Socket (Input)	BIT_02	AX Adapter, Bool Bit 2
Socket (Input)	BIT_03	AX Adapter, Bool Bit 3
Socket (Input)	BIT_04	AX Adapter, Bool Bit 4
Socket (Input)	BIT_05	AX Adapter, Bool Bit 5
Socket (Input)	BIT_06	AX Adapter, Bool Bit 6
Socket (Input)	BIT_07	AX Adapter, Bool Bit 7
Socket (Input)	BIT_08	AX Adapter, Bool Bit 8
Socket (Input)	BIT_09	AX Adapter, Bool Bit 9
Socket (Input)	BIT_10	AX Adapter, Bool Bit 10
Socket (Input)	BIT_11	AX Adapter, Bool Bit 11
Socket (Input)	BIT_12	AX Adapter, Bool Bit 12
Socket (Input)	BIT_13	AX Adapter, Bool Bit 13
Socket (Input)	BIT_14	AX Adapter, Bool Bit 14
Socket (Input)	BIT_15	AX Adapter, Bool Bit 15
Plug (Output)	OUT	AW Adapter, WORD Output

Functionality

1. **Input Signal Acquisition**: Each of the 16 AX adapters (BIT_00 to BIT_15) provides a Boolean value via its data channel (D1). As soon as the value of one of these adapters changes, it sends an event via its output E1.
2. **Combination**: The events from all 16 adapters are fed to the input REQ of an internal module ASSEMBLE_WORD_FROM_BOOLS. This module reads the current Boolean values from all 16 channels and combines them into a 16-bit word (WORD).
3. **Synchronization**: The completed word is passed to an edge-triggered D flip-flop (E_D_FF_ANY). The flip-flop receives the value at its D input as soon as it receives a clock signal (from the output CNF of the merge block).
4. **Output**: After successful transfer, the flip-flop outputs a clock signal at its output EO and places the stored word at its Q output. This event triggers the AW adapter OUT, which makes the WORD available on its data channel (D1).

Technical Features

- **Use of a flip-flop**: Output occurs only after a complete merge cycle. This prevents inconsistent or fluctuating data – the output remains stable until a new event occurs.
- **No state machine**: The function block is purely event-driven and has no inherent state; its behavior is determined by the internal function blocks ASSEMBLE_WORD_FROM_BOOLS and E_D_FF_ANY.
- **Adapter interface**: Both the inputs and the output are implemented as adapters. This allows for flexible reuse and encapsulation of the signal types.

State overview

The function block does not contain an explicit state machine. The internal D flip-flop E_D_FF_ANY has two states (Q = 0 or Q = 1) that store the current data value. The flip-flop's output is updated only on a rising edge at the clock input.

Application Scenarios

- **Bundling of Discrete Signals**: In automation technology, 16 individual digital sensors or switches (e.g., limit switches, pushbuttons) are often required. This function block combines these into a single word, which can be used via a fieldbus or as an input for a higher-level controller.
- **Data Preparation for Communication**: Before transmission via a network adapter (e.g., PROFINET, EtherCAT), multiple binary signals must be packed into a single data word.

- **Signal Register**: The component can be used as a simple 16-bit register that buffers the current state of all inputs and updates them only when changes occur.

Comparison with Similar Components

- **Classic Boolean-to-WORD Components**: Standard components often combine Boolean inputs directly into an integer type, but without an adapter interface. `ASSEMBLE_AW_FROM_AX` uses an adapter, which enables clearly separated signal and event transmission and simplifies reuse in modular architectures.
- **Adapter-Based Alternatives**: Similar components exist for other word sizes (e.g., BYTE, DWORD) or with additional filtering capabilities. This component focuses on the essentials and uses a flip-flop output for clean synchronization.

Conclusion

The `ASSEMBLE_AW_FROM_AX` function block offers a clean, synchronized way to combine 16 Boolean signals (via AX adapter) into a 16-bit word (via AW adapter). The combination of the merging block and the edge-triggered flip-flop ensures stable output and is particularly suitable for use in time-critical or signal processing environments.

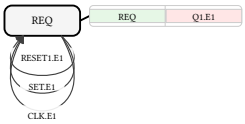
2.1.11.5 Numeric Index (adapter)

Here you will find all exercises from the `adapter` workspace, listed in ascending order. ...

AX_FB_RS_T_FF



ECC: AX_FB_RS_T_FF



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AX_FB_RS_T_FF** implements a reset-dominant bistable latch with an additional toggle function. The block communicates via standardized adapter interfaces and enables flexible integration with other components.

Interface Structure

The block has no direct event or data inputs/outputs at the FB level. All communication takes place via **adapters**, which in turn provide their own event and data channels.

Event Inputs

The following adapter events trigger processing:

- **SET.E1** – Event to set the output
- **RESET1.E1** – Event to reset the output (priority)
- **CLK.E1** – Event for the clock signal, which triggers a toggle on its rising edge

Event Outputs

- **Q1.E1** – Event sent after each execution of the algorithm (regardless of the result)

Data Inputs

The following data values are received via the adapters:

- **SET.D1** (BOOL) – Set input: `TRUE` ⟶ sets Q1 to `TRUE`
- **RESET1.D1** (BOOL) – Reset input: `TRUE` ⟶ sets Q1 to `FALSE` (has priority over SET)
- **CLK.D1** (BOOL) – Clock input: Edge transition from `FALSE` to `TRUE` triggers toggle

Data Outputs

- **Q1.D1** (BOOL) – Output value of the flip-flop

Adapter

Adapter	Type	Role	Comment
`SET`	adapter::types::unidirectional::AX	Input	Set
`RESET1`	adapter::types::unidirectional::AX	Input	Reset
`CLK`	adapter::types::unidirectional::AX	Input	Clock
`Q1`	adapter::types::unidirectional::AX	Output	Output

All adapters are of type `AX` (unidirectional) and each provides one event channel (`.E1`) and one data channel (`.D1`).

| `Q1` | | |] ...

Functionality

The module operates according to the following deterministic algorithm, which is executed upon each incoming event (SET, RESET1, or CLK):

1. **Priority: Reset Dominates**

If `RESET1.D1 = TRUE` is present, `Q1.D1` is set to `FALSE`.

1. **Otherwise: Set Takes Precedence**

If only `SET.D1 = TRUE` (and `RESET1.D1 = FALSE`) is present, `Q1.D1` is set to `TRUE`.

1. **Otherwise: Toggle on rising edge**

If neither RESET1 nor SET is present, the program checks whether `CLK.D1 = TRUE` was present in the previous cycle and `FALSE` in the previous cycle (rising edge). In this case, the output `Q1` is inverted (`NOT Q1.D1`).

1. **Edge Detection**

The internal variable `EDGE` stores the last value of `CLK.D1`. It is updated after each iteration (`EDGE := CLK.D1`). This detects the rising edge as soon as `CLK.D1` transitions from `FALSE` to `TRUE`.

After the algorithm executes, the event `Q1.E1` is always sent.

Technical Features

- **Adapter-Based Communication**: The device uses adapters exclusively for input and output, facilitating loose coupling and reusability in different contexts.
- **Reset Dominance**: The reset input takes precedence over the set input. This corresponds to the typical behavior of an RS flip-flop with reset priority.
- **Edge-Triggered Toggle**: The toggle occurs only on a rising edge of the clock signal, not on a static high level. Internal edge detection prevents multiple switching operations during a sustained `TRUE` event at the clock.
- **Single-State ECC**: The function block has only one state (`REQ`). All transitions return to this state, enabling continuous processing of each incoming event without additional state logic.

State Overview

The function block is implemented as a **Basic FB** with the following ECC:

State	Action	Output
`REQ`	Execute algorithm `REQ`	`Q1.E1` (after algorithm)

Transitions:

- From `REQ` to `REQ` at **RESET1.E1**
- From `REQ` to `REQ` at **SET.E1**
- From `REQ` to `REQ` at **CLK.E1**

There are no other states. The algorithm is always executed in the same context.

Application Scenarios

- **Safety Controllers**: Reset-dominant behavior is useful when a fault state (reset) must have the highest priority, e.g., in emergency stop circuits.
- **Toggle Function with Clock**: Switching between two states on each clock pulse, e.g., for a flashing light or pulse counter.
- **Hybrid Circuits**: Combines set/reset and toggle functionality in a single component, saving space and logic.

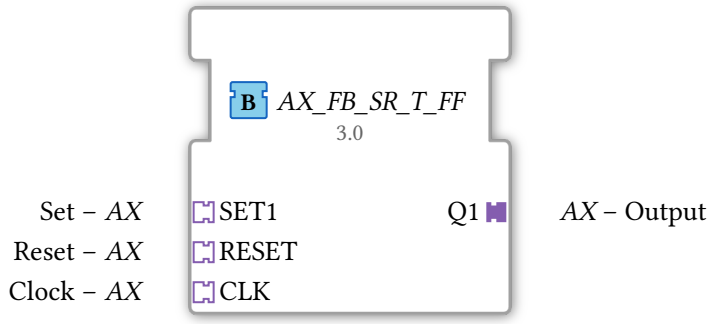
Comparison with Similar Components

- **RS Flip-Flop (Reset-Dominant)**: Behaves like a conventional RS flip-flop with reset priority, but does not offer toggle functionality.
- **T Flip-Flop**: Can only toggle; it does not have separate set/reset inputs.
- **JK Flip-Flop**: Offers set, reset, and toggle functionality, but with different priority logic (no explicit reset dominance).
- **AX_FB_RS_T_FF** combines reset dominance and edge-triggered toggle functionality in a single component, utilizing adapters for platform-independent connectivity.

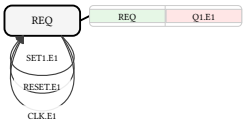
Conclusion

The **AX_FB_RS_T_FF** is a versatile, adapter-based function block for bistable circuits with reset-dominant logic and an integrated toggle function. It can be easily integrated into modular automation systems using adapters. Edge detection and clear input priority make it robust and reliable for typical control tasks.

AX_FB_SR_T_FF



ECC: AX_FB_SR_T_FF



• – ○ • – ○ • – ○ •

Introduction

The function block **AX_FB_SR_T_FF** implements a bistable, set-dominant latch (SR flip-flop) combined with a toggle function. All communication takes place via adapter interfaces that bundle both event and data signals.

Interface Structure

The block has no direct event or data inputs/outputs, but only adapters. These adapters each provide one event (`E1`) and one data value (`D1`) of type `BOOL`.

Event Inputs

No direct event inputs. The following adapters provide the trigger mechanism via their event `E1`:

- **SET1** – Set event
- **RESET** – Reset event
- **CLK** – Clock event (for toggle)

Event Outputs

No direct event outputs. The adapter **Q1** provides confirmation of the output change via its event `E1`.

Data Inputs

The three input adapters each provide a data value `D1` (BOOL):

- **SET1.D1** – Set signal (active when TRUE)
- **RESET.D1** – Reset signal (active when TRUE)
- **CLK.D1** – Clock signal (rising edge triggers toggle)

Data Outputs

The output adapter **Q1** outputs the current state of the latch as a BOOL value via **Q1.D1**.

Adapters

Adapter	Type	Direction	Description
`SET1`	`adapter::types::unidirectional::AX`	Socket (Input)	Set Input
`RESET`	`adapter::types::unidirectional::AX`	Socket (Input)	Reset Input
`CLK`	`adapter::types::unidirectional::AX`	Socket (Input)	Clock Input for Toggle
`Q1`	`adapter::types::unidirectional::AX`	Plug (Output)	Latch Output

Functionality

The function block operates according to the following priority-controlled algorithm:

1. **Set Dominance**: If `SET1.D1 = TRUE` is active, the output `Q1.D1` is set to `TRUE`.
2. **Reset**: Otherwise, if `RESET.D1 = TRUE` is active, `Q1.D1` is set to `FALSE`.
3. **Toggle**: If neither the set nor the reset operation is active, and a **rising edge** (transition from FALSE to TRUE) is detected at the clock input `CLK.D1`, the current output value is toggled (`Q1.D1 := NOT Q1.D1`).

The edge detection is performed by the internal variable `EDGE`: It stores the previous value of `CLK.D1`. The algorithm checks whether `CLK.D1 = TRUE` and `EDGE = FALSE` are active. After the calculation, `EDGE` is set to the current `CLK.D1`.

Technical Features

- **Pure Adapter Interface**: The module does not use traditional event/data ports, but rather encapsulates the signals in adapters. This allows for flexible reuse and encapsulation of interfaces.
- **Set Dominant Behavior**: Simultaneous setting and resetting always results in the output being set.
- **Toggle with Upward Edge**: The toggle function reacts only to rising edges of the clock signal, not to static levels.
- **Single ECC State**: The ECC consists only of the state `REQ`. Each incoming event (`SET1.E1`, `RESET.E1`, or `CLK.E1`) immediately triggers the re-execution of the algorithm.

State Overview

The function block has only one state:

- **REQ** – Waits for incoming events. The algorithm `REQ` is executed upon each event. All transitions return to `REQ`.

Internally, the variable `EDGE` is used for edge detection; it is part of the internal state but is not visible as an ECC state.

Application Scenarios

- **Priority Control**: When a set pulse should always take precedence over a reset pulse, e.g., in safety circuits.
- **Combined Set/Reset and Toggle Function**: For applications where an output can be set/reset by external signals and also toggled by a push button (e.g., manual operation with priority).
- **Adapter-Based Systems**: In a modular environment where interfaces are standardized via adapters (e.g., AX adapter system).

Comparison with Similar Function Blocks

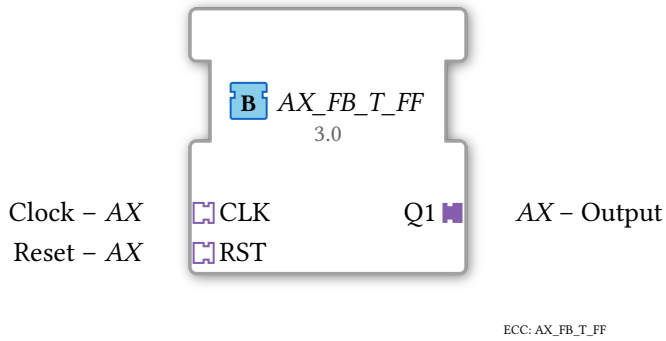
- **SR Flip-Flop (Set-Dominant)** – pure set/reset function without toggle. This function block extends this by adding the toggle capability.

- **T Flip-Flop** – toggle only (no set/reset). AX_FB_SR_T_FF offers additional set and reset priority.
- **Reset-Dominant SR Flip-Flops** – resets in case of conflicts. This function block behaves in the opposite way.

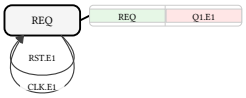
Conclusion

The **AX_FB_SR_T_FF** combines a set-dominant SR latch with an edge-triggered toggle function in a compact module. Its exclusive use of adapters makes it ideal for modular, adapter-based automation solutions. Clear prioritization and edge detection provide predictable and robust switching behavior.

AX_FB_T_FF



ECC: AX_FB_T_FF



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AX_FB_T_FF** implements a clock-edge-triggered toggle flip-flop (T-FF). It toggles its output state on each active clock edge and can be reset asynchronously.

Inputs and outputs are exclusively via adapters of type `AX`, which transmit both an event and a Boolean data value.

Interface Structure

The function block has no direct event or data interfaces.

All communication takes place via **adapters** (plugs and sockets).

Event Inputs

None (events are received via adapters `CLK` and `RST`).

Event Outputs

None (Events are sent via adapter `Q1`).

Data Inputs

None (Data is received via adapters `CLK` and `RST`).

Data Outputs

None (Data is sent via adapter `Q1`).

Adapter

Direction	Name	Type	Description
Socket (Input)	`CLK`	`adapter::types::unidirectional::AX`	Clock signal – with each incoming event (E1), the Boolean value (D1) is evaluated as the clock level.
Socket (Input)	`RST`	`adapter::types::unidirectional::AX`	Reset – with an incoming event (E1), the output is set to FALSE, independent of the clock (asynchronous reset).
Plug (Output)	`Q1`	`adapter::types::unidirectional::AX`	Output – with each clock change or reset, an event (E1) is triggered and the current Boolean value (D1) is sent.

The adapters of type `unidirectional::AX` contain internally:

- An event (E1)
- A data (D1) of type `BOOL`

Functionality

The device operates as a **toggle flip-flop with positive edge detection**.
It has an internal memory `EDGE` that stores the last clock level.
Algorithm (in REQ state):

1. When a reset event arrives at `RST.E1`:

- Output `Q1.D1` is set to `FALSE`.
- Output event `Q1.E1` is triggered.

1. Otherwise, if a clock event arrives at `CLK.E1`:

- Check if the current clock level is `CLK.D1 = TRUE` and the previous level is `EDGE = FALSE` (i.e., a rising edge).
- If so: `Q1.D1` is inverted (`NOT Q1.D1`).
- Regardless of the edge, `EDGE := CLK.D1` is set (level marker).
- Output event `Q1.E1` is triggered.

1. If the output remains unchanged when `RST.E1` or `CLK.E1` does not arrive, no event is sent.

Technical Features

- **Adapter-based interface**: The module uses adapters exclusively, which facilitates modular coupling with other modules of the same type.
- **Edge detection**: The internal variable `EDGE` implements simple positive edge detection. A key change from 0⟶1 triggers the toggling; a constant high level triggers only once.
- **Asynchronous reset**: The reset takes precedence over the clock edge – it immediately resets the output, even if a clock event is simultaneously occurring.
- **Only one ECC state**: The entire process takes place in the state `REQ`. Both transitions (`CLK.E1` and `RST.E1`) return to this state.

State Overview

The ECC consists of a single state, `REQ`.

Each incoming event pulse (via `CLK.E1` or `RST.E1`) triggers the execution of the algorithm `REQ` and an immediate output event on `Q1.E1`.

Current State	Incoming Event	Next State	Executed Action
REQ	`RST.E1`	REQ	RESET: Q1.D1 = FALSE
REQ	`CLK.E1`	REQ	Toggle on rising edge and update EDGE

No other states or dwell times.

Application Scenarios

- **Clock-Controlled State Transition**: Switching an output on each rising edge of a clock signal (e.g., for frequency dividers or counters).
- **Key Debouncing**: Combined with a debouncing component, the flip-flop can generate a stable state transition with each key press.
- **Signal Switching**: Switching a signal on and off using repeated pulses.
- **Control in Automation Systems**: E.g., switching between two operating modes.

Comparison with Similar Components

- **Standard T-FF (e.g., `F_TRIG`/`R_TRIG`)**: These components only detect edges but do not toggle the output. `AX_FB_T_FF` combines edge detection and toggle functionality.
- **SR Flip-Flop (Set/Reset)**: Unlike the SR flip-flop, the T flip-flop has only one reset input and toggles on every clock cycle instead of being controlled by separate set and reset signals.
- **Adapter-Based Variants**: Other T flip-flops in 4diac often use direct inputs/outputs. This component allows loose coupling via adapters, which increases reusability.

Conclusion

The `AX_FB_T_FF` is a compact and flexible function block that implements a toggle flip-flop with positive edge detection and asynchronous reset.

Its pure adapter interface makes it particularly well-suited for modular, adapter-based designs where data and event transmission occur over a single channel.

Its simple logic and minimal state machine make it reliable and easy to understand.

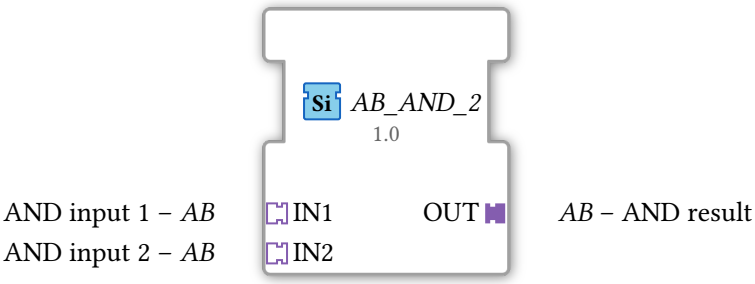
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.1.11.6 bitwiseOperators

AB_AND_2



• – ◦ • – ◦ • – ◦ •

Introduction

AB_AND_2 is a generic function block for the bitwise AND operation across 2 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AB)

Functionality
As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features
- **Generic function block**: The FB is defined as a generic type (`GEN_AB_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview
Since this is a combinational logic block, AB_AND_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios
- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks
Unlike `AX_AND_2`, which combines individual Boolean truth values, `AB_AND_2` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AB_AND_2` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_AND_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

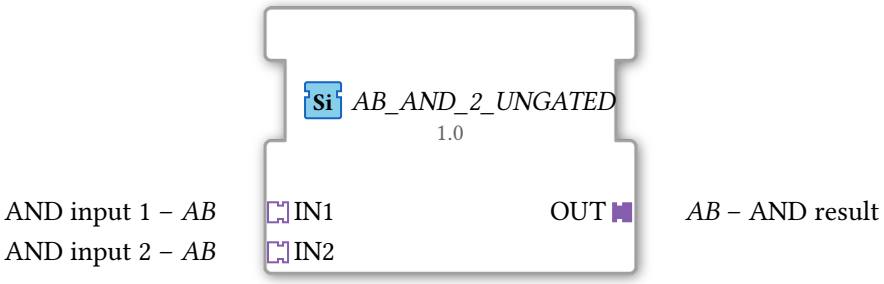
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_AND_2 offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_AND_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_AND_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_AND_2_UNGATED is a generic function block for the bitwise AND operation across 2 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_AND_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_2`, which combines individual Boolean truth values, `AB_AND_2_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AB_AND_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_AND_2`**: The gated variant – updates the output only on an actual value change.

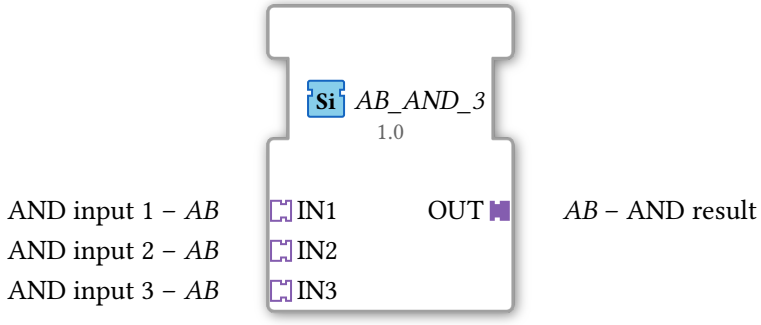
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_AND_2_UNGATED offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_AND_3



• – ◦ • – ◦ • – ◦ •

Introduction

AB_AND_3 is a generic function block for the bitwise AND operation across 3 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_AND_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_3`, which combines individual Boolean truth values, `AB_AND_3` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AB_AND_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_AND_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

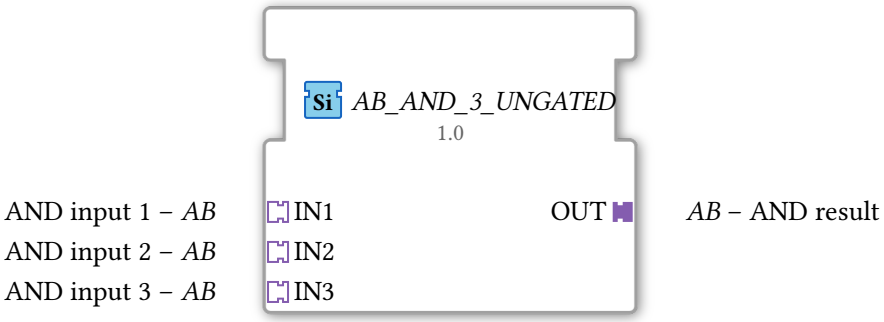
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_AND_3 offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_AND_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_AND_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_AND_3_UNGATED is a generic function block for the bitwise AND operation across 3 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_AND_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_3`, which combines individual Boolean truth values, `AB_AND_3_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AB_AND_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_AND_3`**: The gated variant – updates the output only on an actual value change.

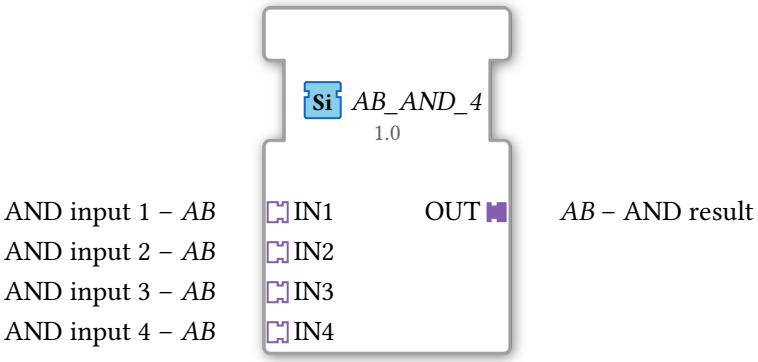
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_AND_3_UNGATED offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_AND_4



• – ◦ • – ◦ • – ◦ •

Introduction

AB_AND_4 is a generic function block for the bitwise AND operation across 4 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: AND input 4 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.

- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_AND_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_4`, which combines individual Boolean truth values, `AB_AND_4` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AB_AND_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_AND_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

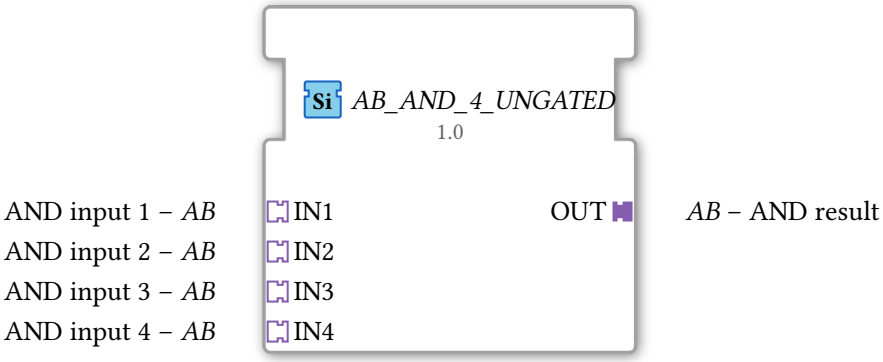
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_AND_4 offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_AND_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_AND_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_AND_4_UNGATED is a generic function block for the bitwise AND operation across 4 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: AND input 4 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_AND_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_4`, which combines individual Boolean truth values, `AB_AND_4_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AB_AND_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_AND_4`**: The gated variant – updates the output only on an actual value change.

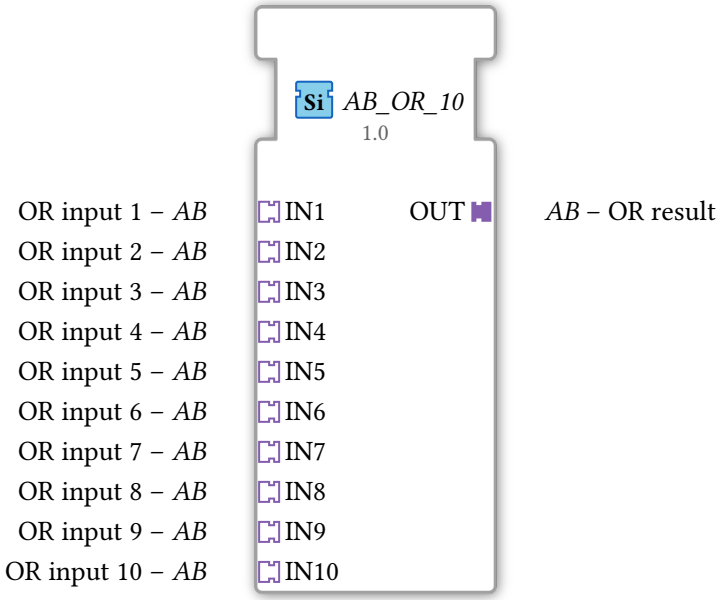
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_AND_4_UNGATED offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_10



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_10 is a generic function block for the bitwise OR operation across 10 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AB)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AB)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AB)
- **IN10**: OR input 10 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 10 input adapters (`IN1` … `IN10`), the block combines the bit patterns of all 10 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_10 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_10`, which combines individual Boolean truth values, `AB_OR_10` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_10` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_OR_10_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

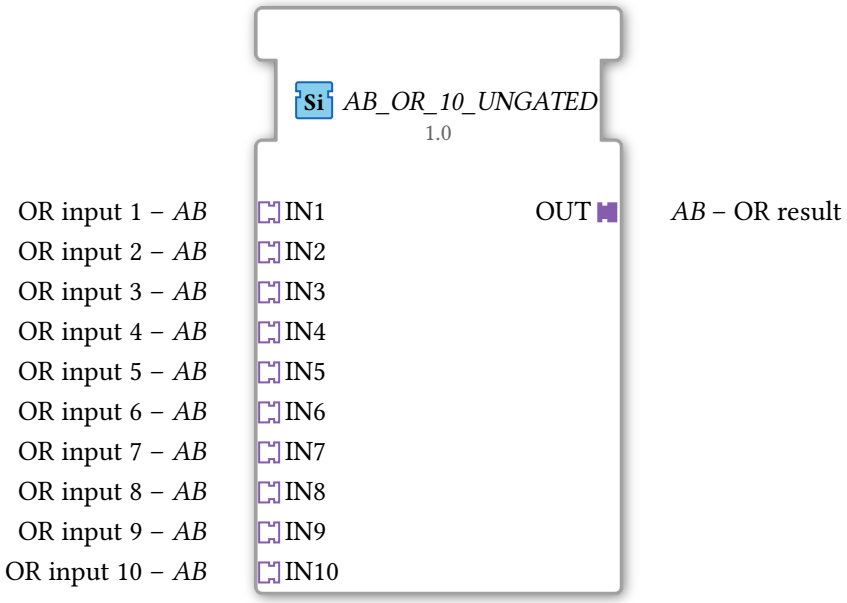
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_OR_10 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_10_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_OR_10`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_10_UNGATED is a generic function block for the bitwise OR operation across 10 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)

- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AB)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AB)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AB)
- **IN10**: OR input 10 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 10 input adapters (`IN1` … `IN10`), the block combines the bit patterns of all 10 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_10_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_10`, which combines individual Boolean truth values, `AB_OR_10_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_10_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_OR_10`**: The gated variant – updates the output only on an actual value change.

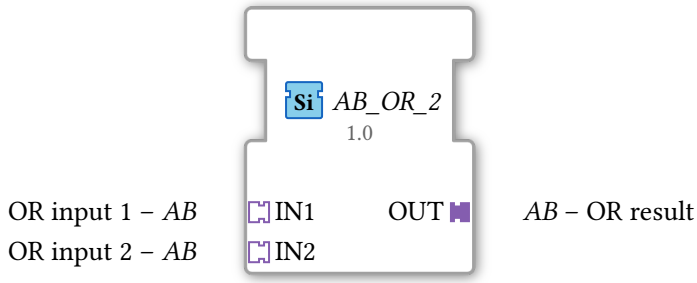
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_OR_10_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_2



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_2 is a generic function block for the bitwise OR operation across 2 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_2`, which combines individual Boolean truth values, `AB_OR_2` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_2` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_OR_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

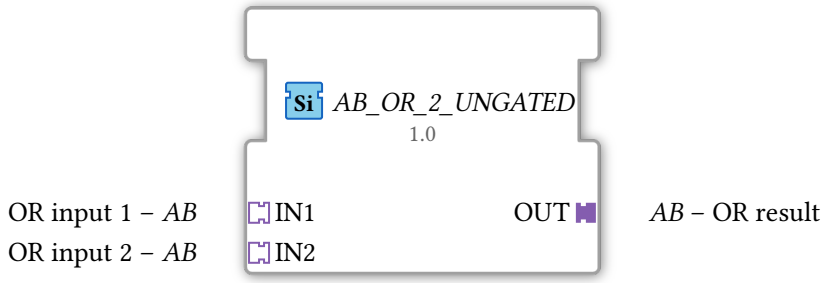
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_OR_2 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_OR_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AB_OR_2_UNGATED is a generic function block for the bitwise OR operation across 2 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_2`, which combines individual Boolean truth values, `AB_OR_2_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AB_OR_2**: The gated variant – updates the output only on an actual value change.

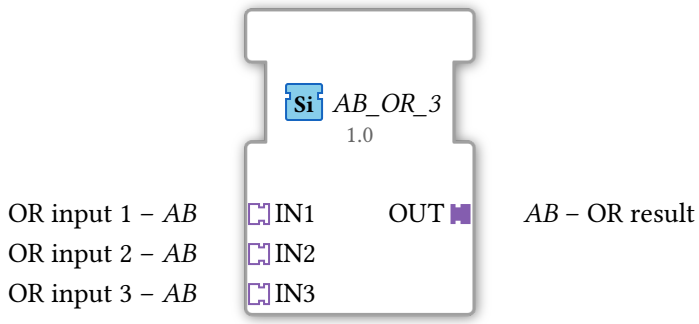
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_OR_2_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_3



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_3 is a generic function block for the bitwise OR operation across 3 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_3`, which combines individual Boolean truth values, `AB_OR_3` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_OR_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

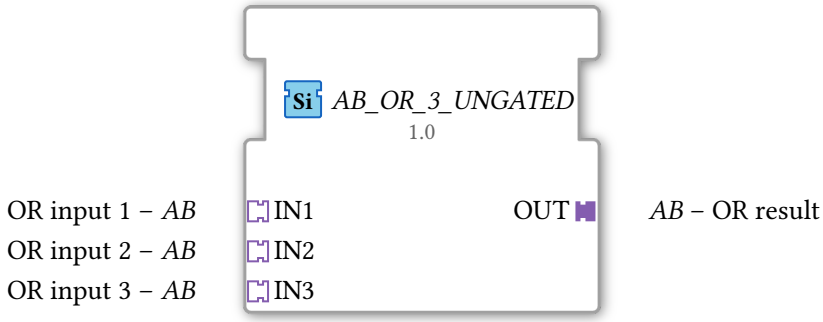
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_OR_3 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_OR_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_3_UNGATED is a generic function block for the bitwise OR operation across 3 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_OR_3, which combines individual Boolean truth values, AB_OR_3_UNGATED operates on the full bit pattern of a BYTE value. Compared to the standard block OR_2, AB_OR_3_UNGATED uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AB_OR_3**: The gated variant – updates the output only on an actual value change.

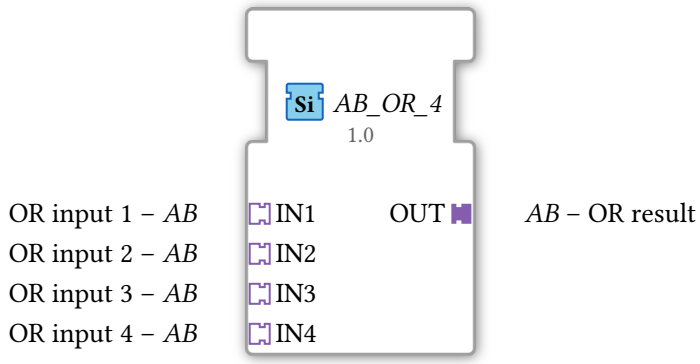
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_OR_3_UNGATED offers a reliable, generic implementation of the bitwise OR function for BYTE values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_4



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_4 is a generic function block for the bitwise OR operation across 4 input values of type BYTE (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the AX_OR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_OR_4`, which combines individual Boolean truth values, `AB_OR_4` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_OR_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

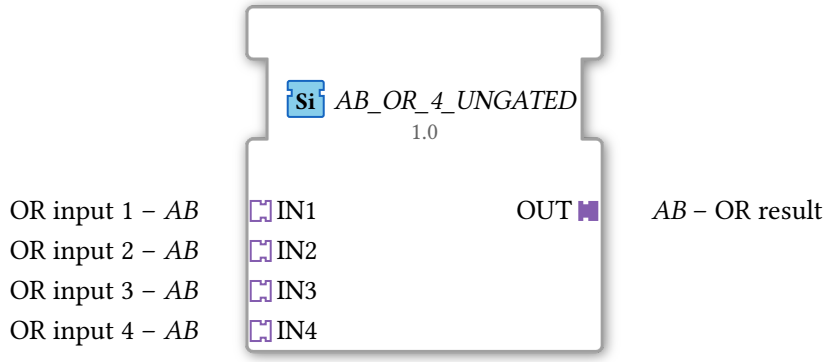
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_OR_4 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_OR_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_4_UNGATED is a generic function block for the bitwise OR operation across 4 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.

- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_4`, which combines individual Boolean truth values, `AB_OR_4_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AB_OR_4`: The gated variant – updates the output only on an actual value change.

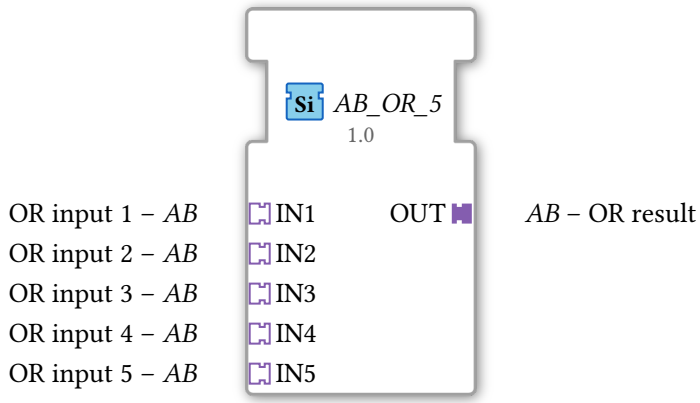
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_OR_4_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_5



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_5 is a generic function block for the bitwise OR operation across 5 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_5 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_5`, which combines individual Boolean truth values, `AB_OR_5` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_5` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_OR_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

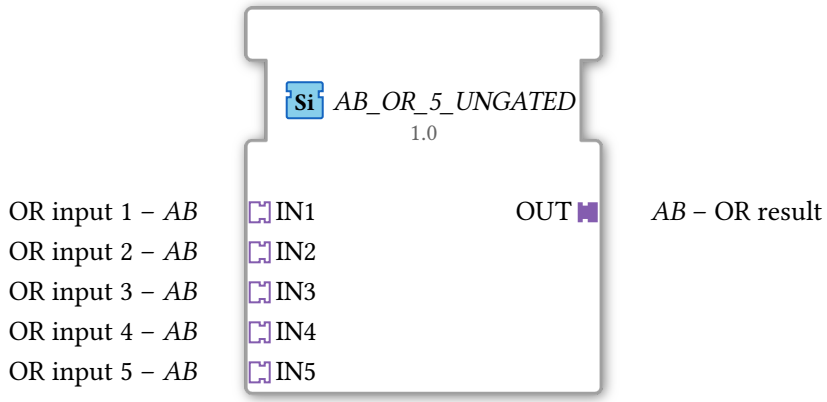
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_OR_5 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_OR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_5_UNGATED is a generic function block for the bitwise OR operation across 5 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)

- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_5_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_5`, which combines individual Boolean truth values, `AB_OR_5_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_5_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AB_OR_5`: The gated variant – updates the output only on an actual value change.

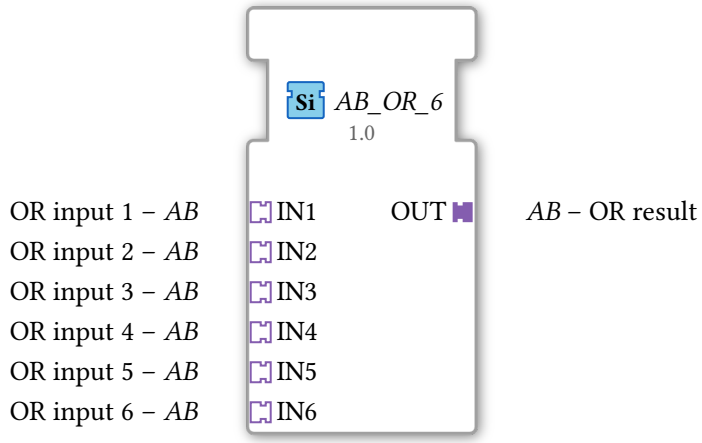
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_OR_5_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_6



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_6 is a generic function block for the bitwise OR operation across 6 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_6 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_6`, which combines individual Boolean truth values, `AB_OR_6` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_6` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_OR_6_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

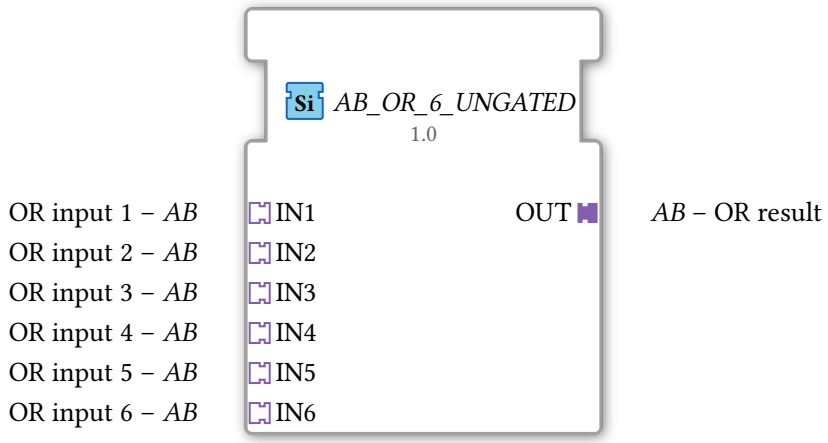
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_OR_6 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_OR_6. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_6_UNGATED is a generic function block for the bitwise OR operation across 6 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_6_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_6`, which combines individual Boolean truth values, `AB_OR_6_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_6_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AB_OR_6`: The gated variant – updates the output only on an actual value change.

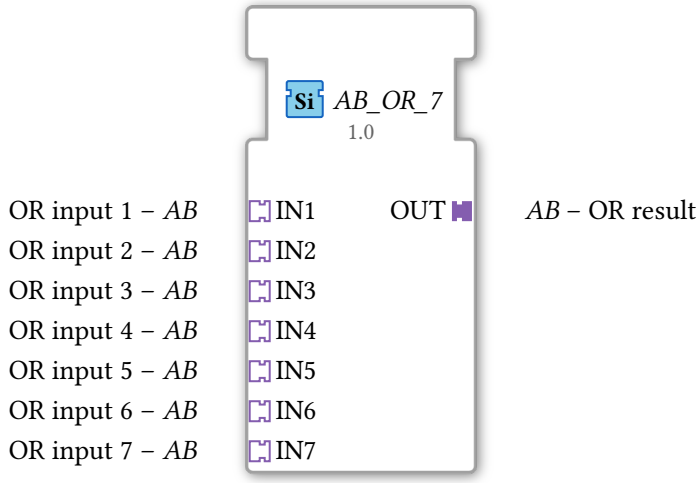
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_OR_6_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_7



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_7 is a generic function block for the bitwise OR operation across 7 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_7 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_OR_7, which combines individual Boolean truth values, AB_OR_7 operates on the full bit pattern of a BYTE value. Compared to the standard block OR_2, AB_OR_7 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AB_OR_7_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

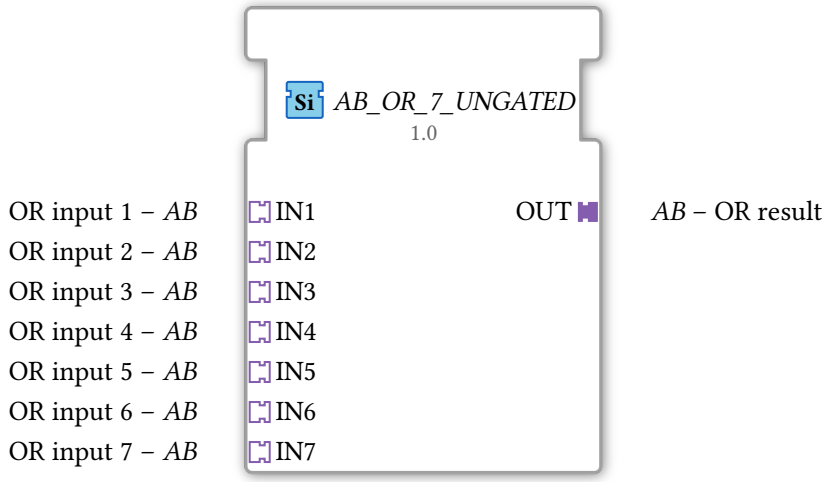
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_OR_7 offers a reliable, generic implementation of the bitwise OR function for BYTE values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_OR_7. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_7_UNGATED is a generic function block for the bitwise OR operation across 7 input values of type BYTE (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the AX_OR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_7_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_7`, which combines individual Boolean truth values, `AB_OR_7_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_7_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_OR_7`**: The gated variant – updates the output only on an actual value change.

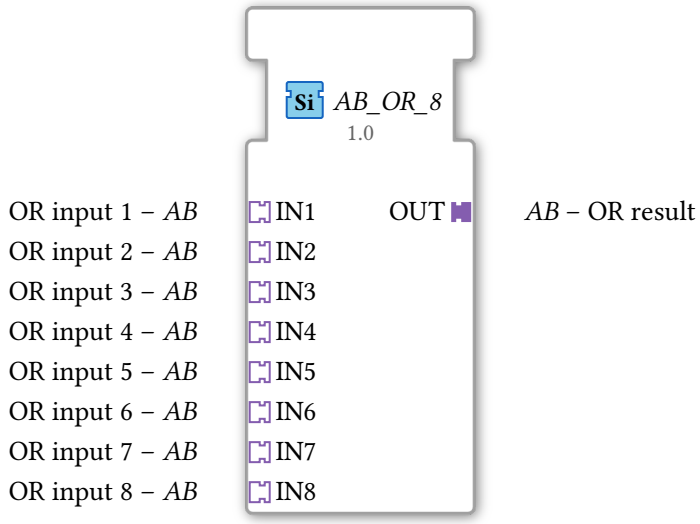
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_OR_7_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_8



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_8 is a generic function block for the bitwise OR operation across 8 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AB)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_8 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_OR_8`, which combines individual Boolean truth values, `AB_OR_8` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_8` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AB_OR_8_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

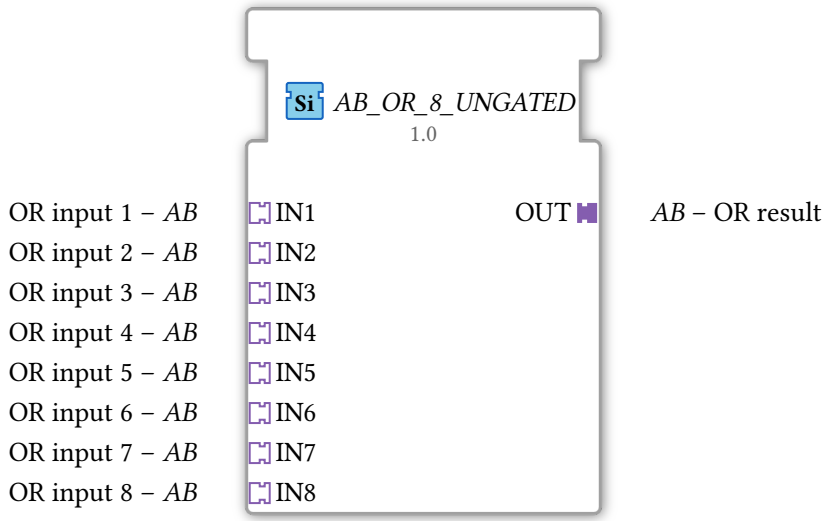
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_OR_8 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_OR_8. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_8_UNGATED is a generic function block for the bitwise OR operation across 8 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AB)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_8_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_8`, which combines individual Boolean truth values, `AB_OR_8_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_8_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AB_OR_8`: The gated variant – updates the output only on an actual value change.

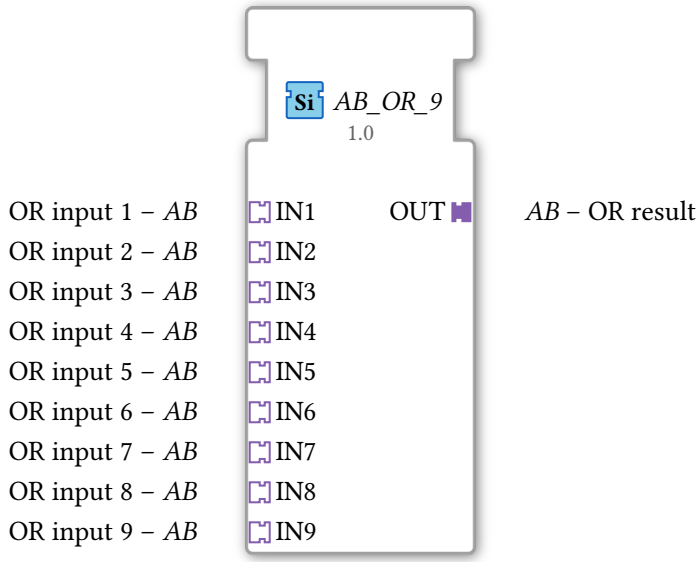
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_OR_8_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_9



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_9 is a generic function block for the bitwise OR operation across 9 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AB)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AB)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 9 input adapters (`IN1` … `IN9`), the block combines the bit patterns of all 9 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_9 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_9`, which combines individual Boolean truth values, `AB_OR_9` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_9` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_OR_9_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

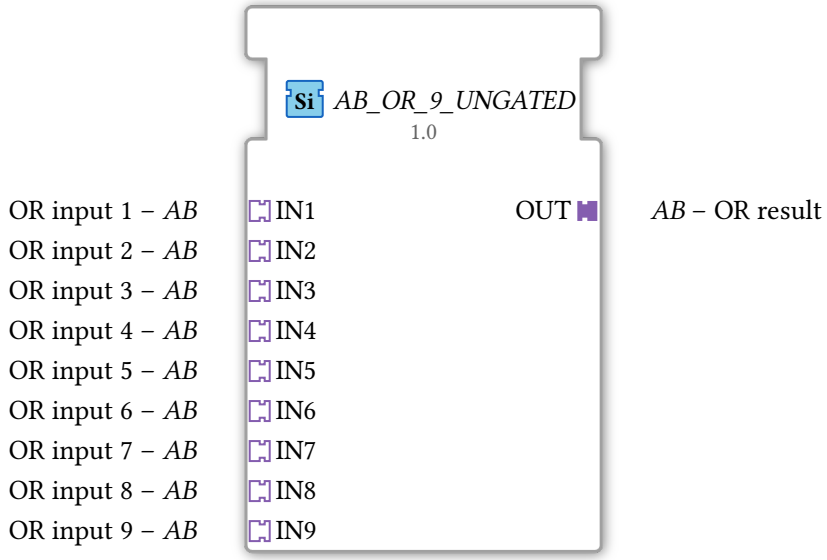
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_OR_9 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_OR_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_OR_9. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_OR_9_UNGATED is a generic function block for the bitwise OR operation across 9 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AB)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AB)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 9 input adapters (`IN1` … `IN9`), the block combines the bit patterns of all 9 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the

operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_OR_9_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_9`, which combines individual Boolean truth values, `AB_OR_9_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AB_OR_9_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AB_OR_9`: The gated variant – updates the output only on an actual value change.

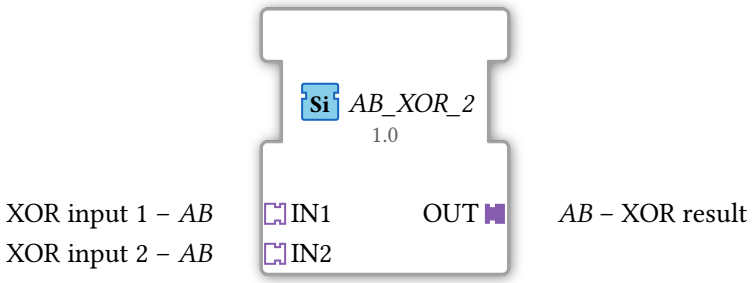
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_OR_9_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_2



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_2 is a generic function block for the bitwise XOR (exclusive OR) operation across 2 input values of type BYTE (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the AX_XOR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 2 input adapters (IN1 … IN2), the block combines the bit patterns of all 2 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter OUT. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on OUT is OUT rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (GEN_AB_XOR) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean AX_XOR blocks, every bit of the BYTE data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type unidirectional::AB – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type BYTE into a single result.
- **Signal aggregation**: Merging multiple BYTE data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_XOR_2, which combines individual Boolean truth values, AB_XOR_2 operates on the full bit pattern of a BYTE value. Compared to the standard block XOR_2, AB_XOR_2 uses adapter-based interfaces

instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AB_XOR_2_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

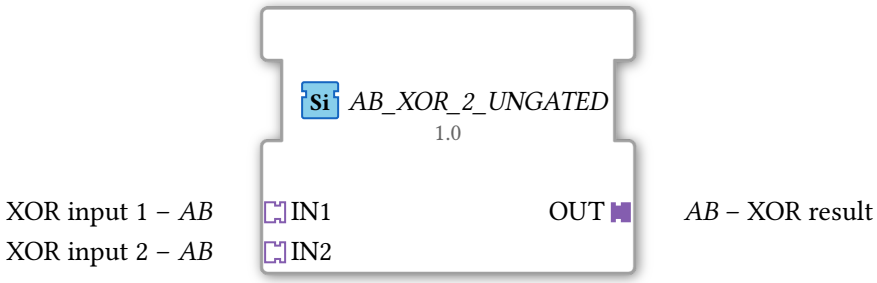
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_XOR_2 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_XOR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_2_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 2 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)

- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_2`, which combines individual Boolean truth values, `AB_XOR_2_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_2`**: The gated variant – updates the output only on an actual value change.

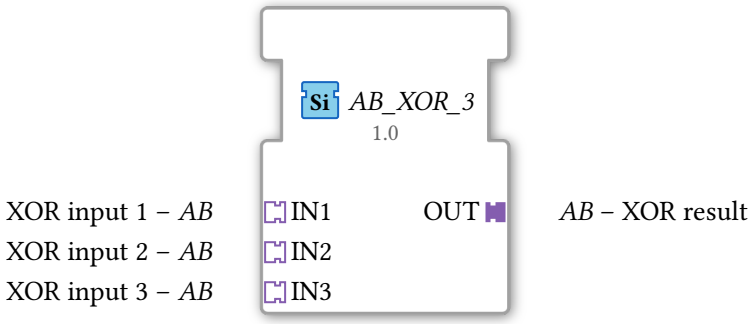
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_XOR_2_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_3



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_3 is a generic function block for the bitwise XOR (exclusive OR) operation across 3 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_3`, which combines individual Boolean truth values, `AB_XOR_3` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

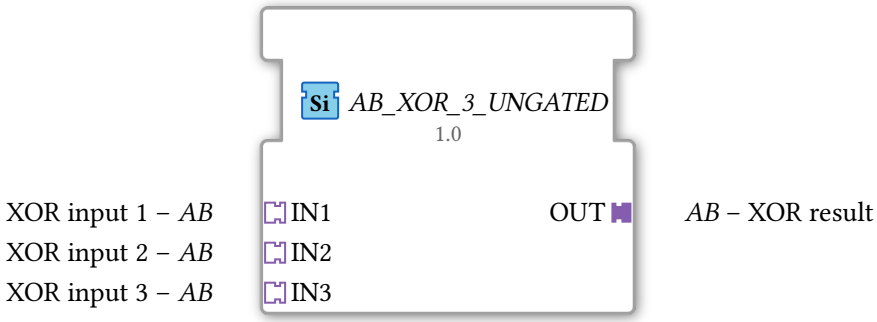
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_XOR_3 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_XOR_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_3_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 3 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality
As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features
- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview
Since this is a combinational logic block, AB_XOR_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios
- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks
Unlike `AX_XOR_3`, which combines individual Boolean truth values, `AB_XOR_3_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_3`**: The gated variant – updates the output only on an actual value change.

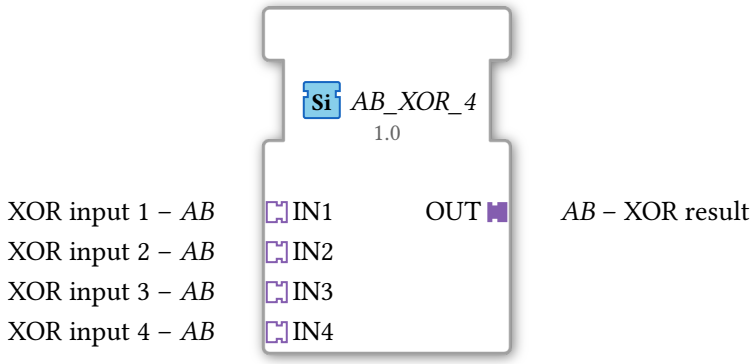
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_XOR_3_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_4



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_4 is a generic function block for the bitwise XOR (exclusive OR) operation across 4 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The

starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_4`, which combines individual Boolean truth values, `AB_XOR_4` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

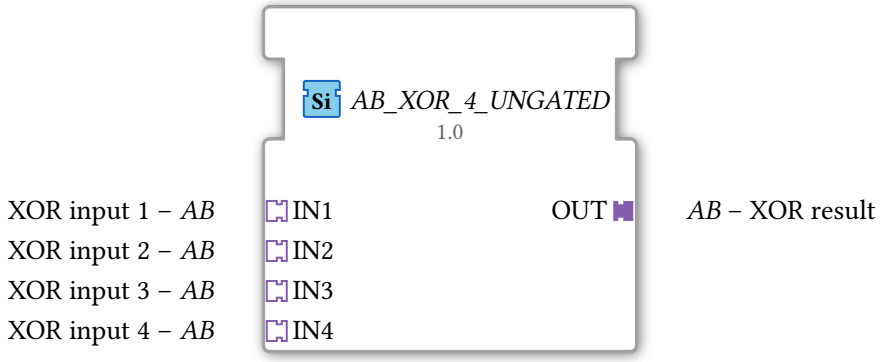
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_XOR_4 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_XOR_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_4_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 4 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.

- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_XOR_4`, which combines individual Boolean truth values, `AB_XOR_4_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_4`**: The gated variant – updates the output only on an actual value change.

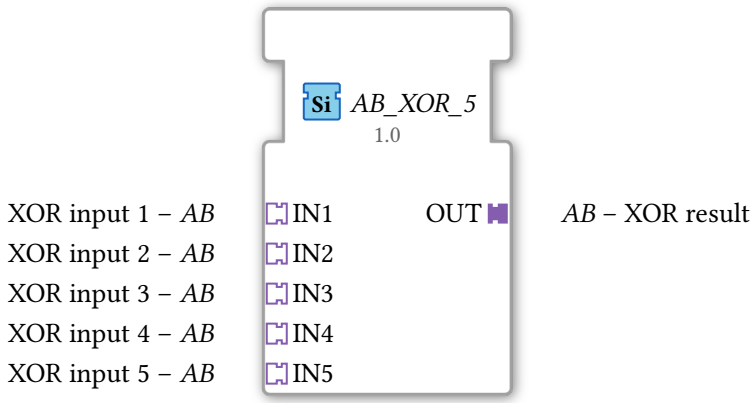
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_XOR_4_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_5



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_5 is a generic function block for the bitwise XOR (exclusive OR) operation across 5 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_5 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_5`, which combines individual Boolean truth values, `AB_XOR_5` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_5` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

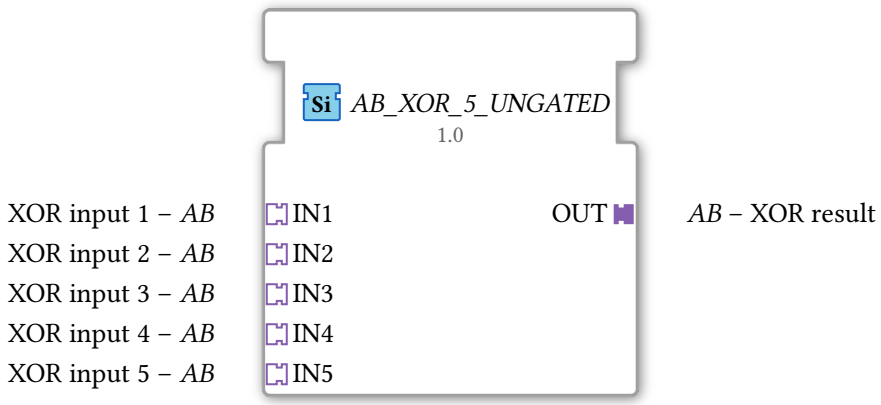
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_XOR_5 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_XOR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_5_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 5 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)

- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_5_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_5`, which combines individual Boolean truth values, `AB_XOR_5_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_5_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_5`**: The gated variant – updates the output only on an actual value change.

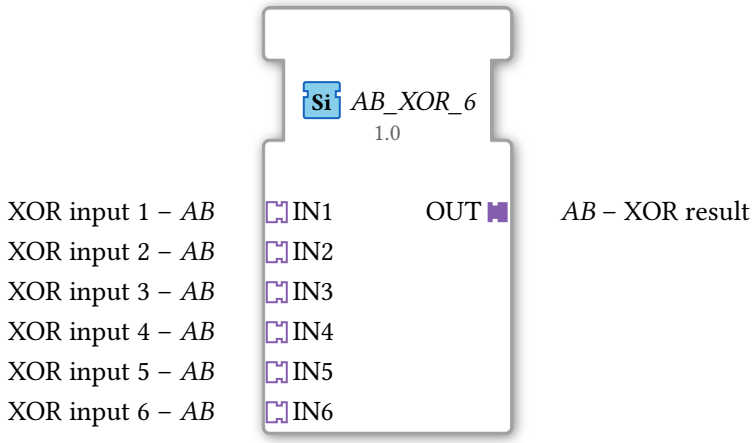
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_XOR_5_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_6



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_6 is a generic function block for the bitwise XOR (exclusive OR) operation across 6 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_6 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_6`, which combines individual Boolean truth values, `AB_XOR_6` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_6` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_6_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

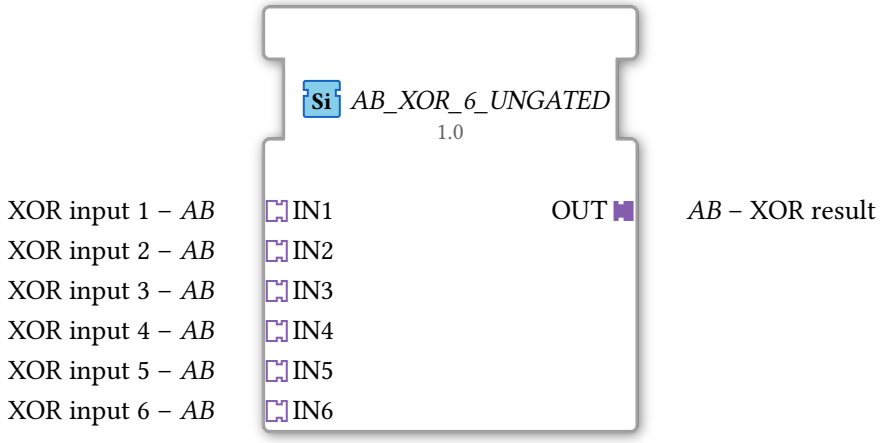
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_XOR_6 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_XOR_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_6_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 6 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_6_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_6`, which combines individual Boolean truth values, `AB_XOR_6_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_6_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AB_XOR_6`: The gated variant – updates the output only on an actual value change.

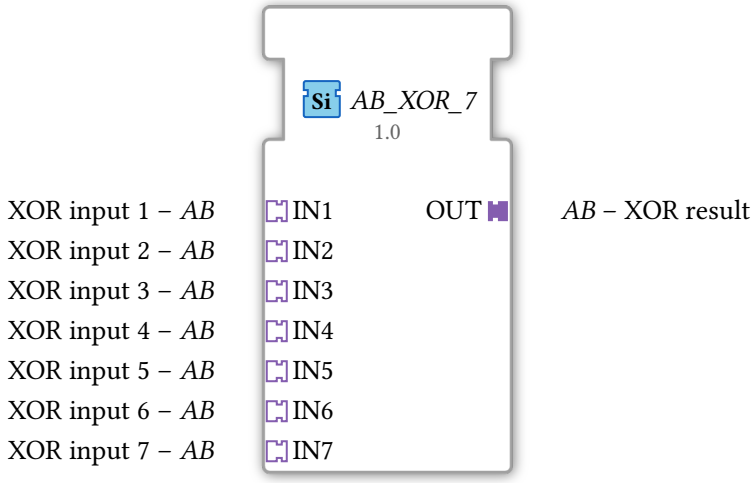
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_XOR_6_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_7



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_7 is a generic function block for the bitwise XOR (exclusive OR) operation across 7 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_7 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_7`, which combines individual Boolean truth values, `AB_XOR_7` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_7` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_7_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

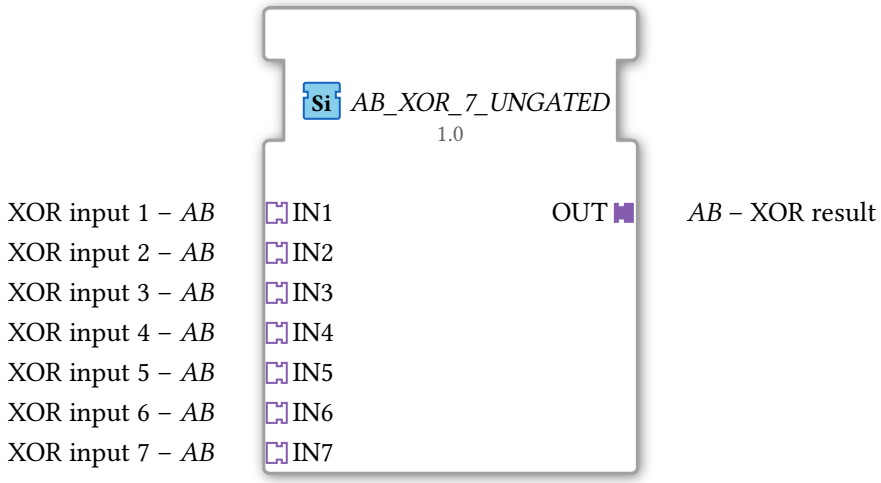
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_XOR_7 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_XOR_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_7_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 7 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_7_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_7`, which combines individual Boolean truth values, `AB_XOR_7_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_7_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_7`**: The gated variant – updates the output only on an actual value change.

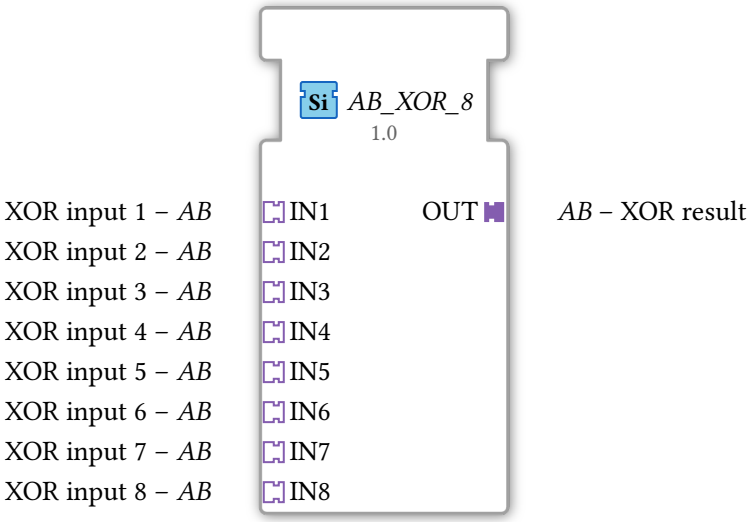
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_XOR_7_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for BYTE values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_8



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_8 is a generic function block for the bitwise XOR (exclusive OR) operation across 8 input values of type BYTE (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the AX_XOR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AB)
- **IN8**: XOR (exclusive OR) input 8 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_8 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_XOR_8`, which combines individual Boolean truth values, `AB_XOR_8` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_8` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_8_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

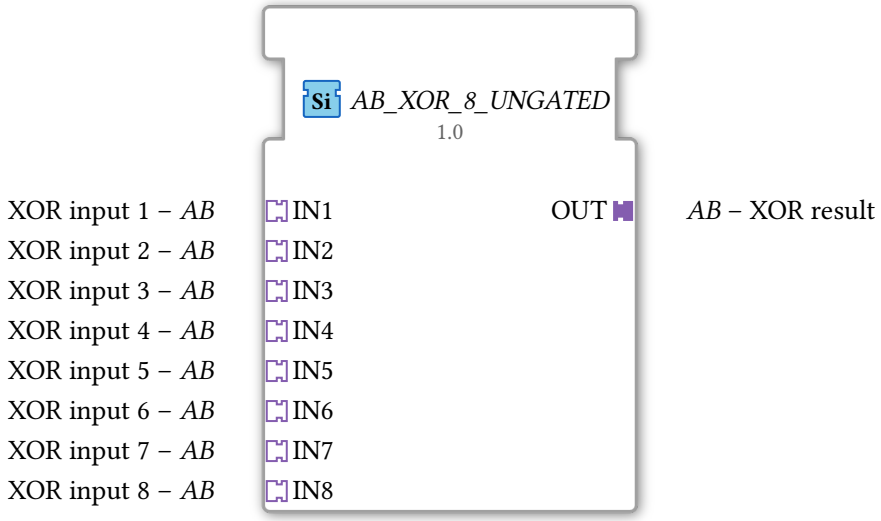
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AB_XOR_8 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AB_XOR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_XOR_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_XOR_8_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 8 input values of type `BYTE` (8-bit bit pattern (byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AB)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AB)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AB)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AB)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AB)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AB)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AB)
- **IN8**: XOR (exclusive OR) input 8 (Type: adapter::types::unidirectional::AB)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AB)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AB_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AB` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AB_XOR_8_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_8`, which combines individual Boolean truth values, `AB_XOR_8_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AB_XOR_8_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AB_XOR_8`**: The gated variant – updates the output only on an actual value change.

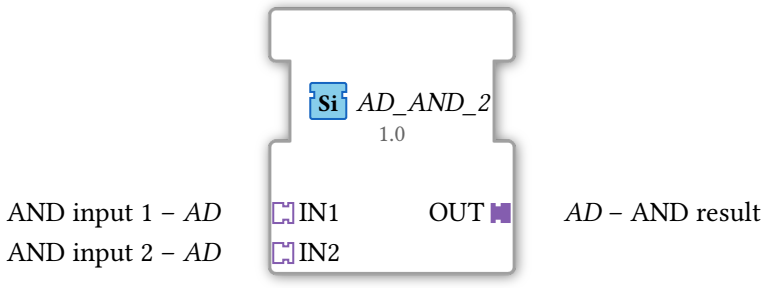
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_XOR_8_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_AND_2



• – ◦ • – ◦ • – ◦ •

Introduction

AD_AND_2 is a generic function block for the bitwise AND operation across 2 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AD)

Functionality
As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features
- **Generic function block**: The FB is defined as a generic type (`GEN_AD_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview
Since this is a combinational logic block, AD_AND_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios
- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks
Unlike `AX_AND_2`, which combines individual Boolean truth values, `AD_AND_2` operates on the full bit pattern of a `DWORD` value. Compared to the standard block AND_2, `AD_AND_2` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_AND_2_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

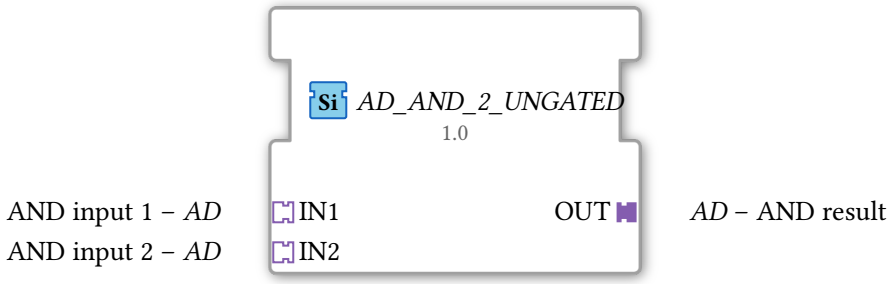
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_AND_2 offers a reliable, generic implementation of the bitwise AND function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_AND_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_AND_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_AND_2_UNGATED is a generic function block for the bitwise AND operation across 2 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_AND_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_2`, which combines individual Boolean truth values, `AD_AND_2_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block AND_2, `AD_AND_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_AND_2`**: The gated variant – updates the output only on an actual value change.

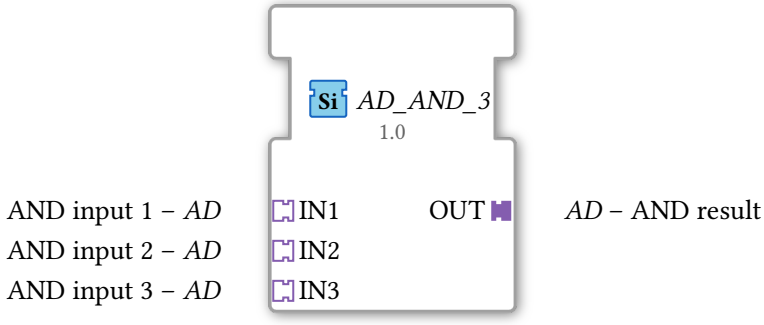
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_AND_2_UNGATED offers a reliable, generic implementation of the bitwise AND function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_AND_3



• – ◦ • – ◦ • – ◦ •

Introduction

AD_AND_3 is a generic function block for the bitwise AND operation across 3 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_AND_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_3`, which combines individual Boolean truth values, `AD_AND_3` operates on the full bit pattern of a `DWORD` value. Compared to the standard block AND_2, `AD_AND_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_AND_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

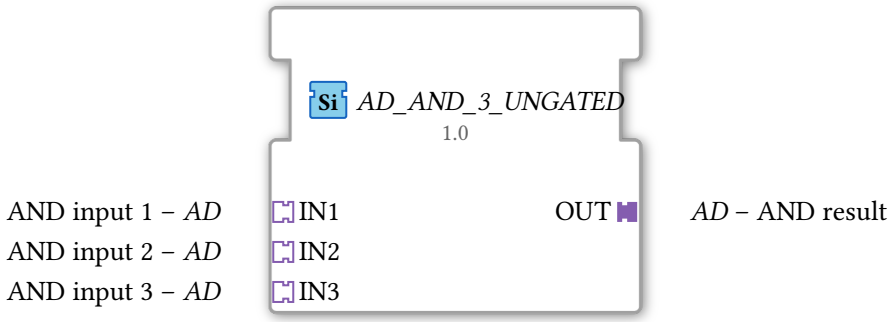
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_AND_3 offers a reliable, generic implementation of the bitwise AND function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_AND_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_AND_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_AND_3_UNGATED is a generic function block for the bitwise AND operation across 3 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_AND_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_3`, which combines individual Boolean truth values, `AD_AND_3_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block AND_2, `AD_AND_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_AND_3`**: The gated variant – updates the output only on an actual value change.

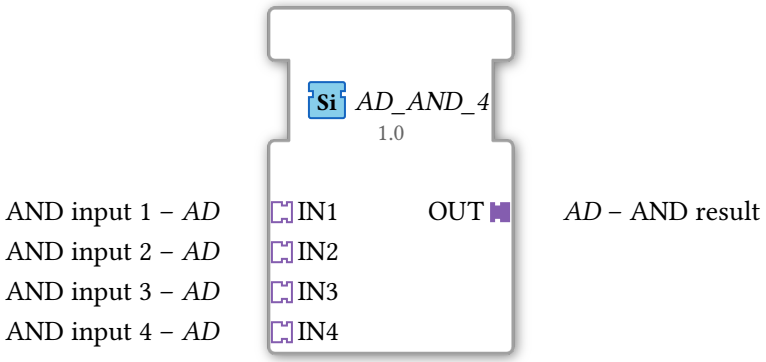
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_AND_3_UNGATED offers a reliable, generic implementation of the bitwise AND function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_AND_4



• – ◦ • – ◦ • – ◦ •

Introduction

AD_AND_4 is a generic function block for the bitwise AND operation across 4 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: AND input 4 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.

- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_AND_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_4`, which combines individual Boolean truth values, `AD_AND_4` operates on the full bit pattern of a `DWORD` value. Compared to the standard block AND_2, `AD_AND_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_AND_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

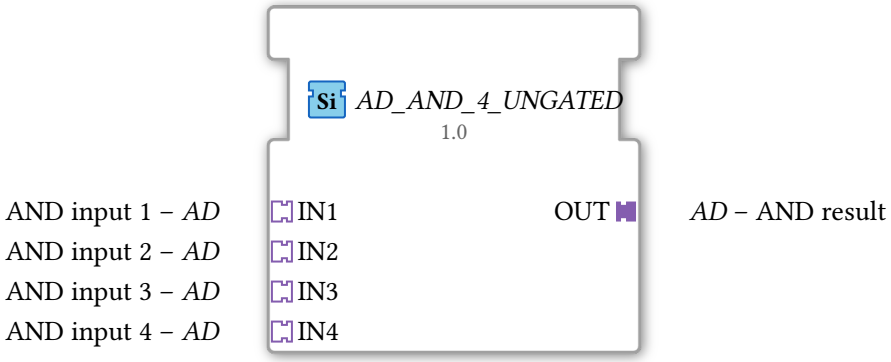
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_AND_4 offers a reliable, generic implementation of the bitwise AND function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_AND_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_AND_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_AND_4_UNGATED is a generic function block for the bitwise AND operation across 4 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: AND input 4 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_AND_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_4`, which combines individual Boolean truth values, `AD_AND_4_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block AND_2, `AD_AND_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_AND_4`**: The gated variant – updates the output only on an actual value change.

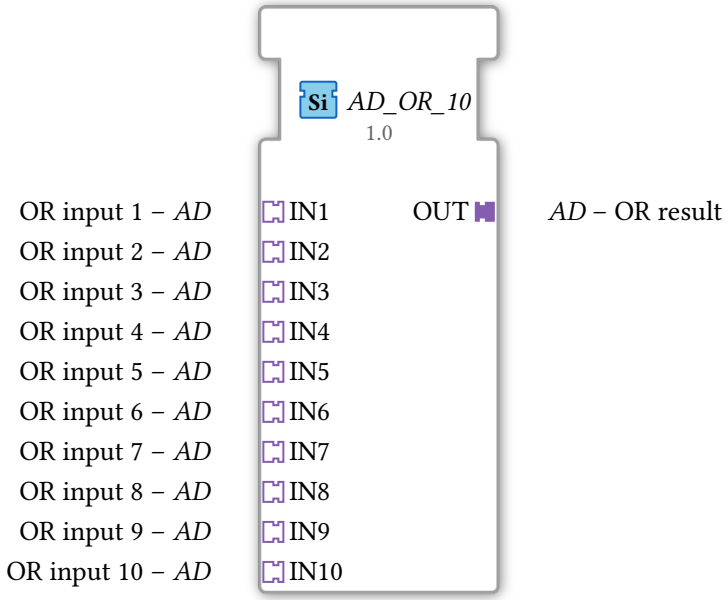
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_AND_4_UNGATED offers a reliable, generic implementation of the bitwise AND function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_10



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_10 is a generic function block for the bitwise OR operation across 10 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AD)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AD)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AD)
- **IN10**: OR input 10 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 10 input adapters (`IN1` … `IN10`), the block combines the bit patterns of all 10 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_10 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_10`, which combines individual Boolean truth values, `AD_OR_10` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_10` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_OR_10_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

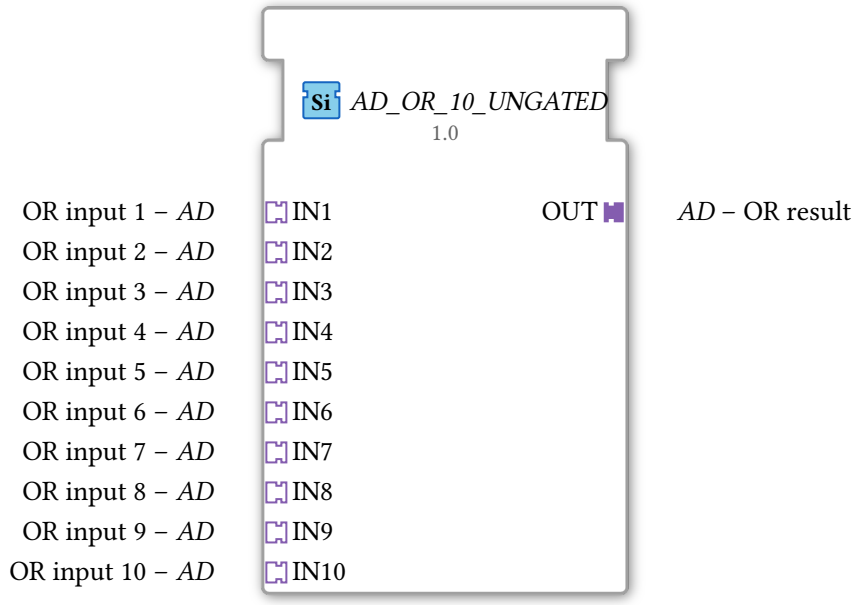
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_OR_10 offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_10_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_OR_10`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_10_UNGATED is a generic function block for the bitwise OR operation across 10 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)

- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AD)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AD)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AD)
- **IN10**: OR input 10 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 10 input adapters (`IN1` … `IN10`), the block combines the bit patterns of all 10 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_10_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_10`, which combines individual Boolean truth values, `AD_OR_10_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_10_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_OR_10`**: The gated variant – updates the output only on an actual value change.

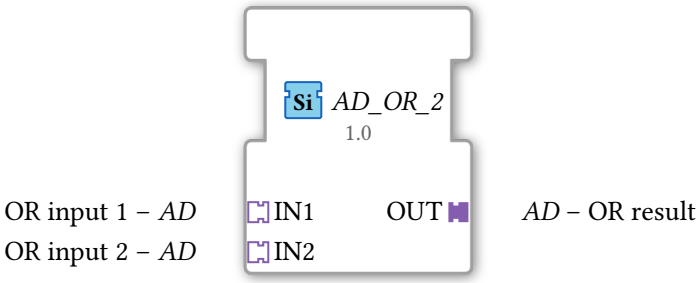
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_OR_10_UNGATED offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_2



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_2 is a generic function block for the bitwise OR operation across 2 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_2`, which combines individual Boolean truth values, `AD_OR_2` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_2` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_OR_2_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

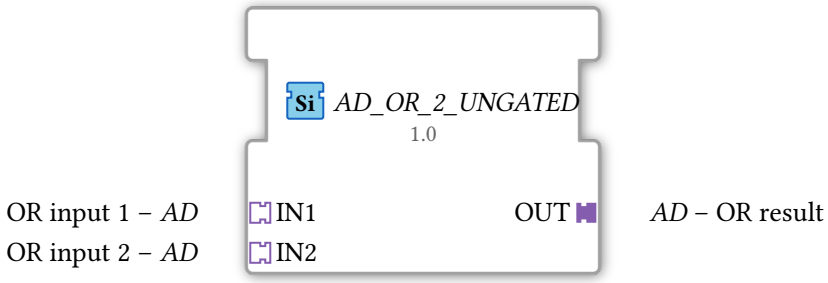
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_OR_2 offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_OR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_2_UNGATED is a generic function block for the bitwise OR operation across 2 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_2`, which combines individual Boolean truth values, `AD_OR_2_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_OR_2`**: The gated variant – updates the output only on an actual value change.

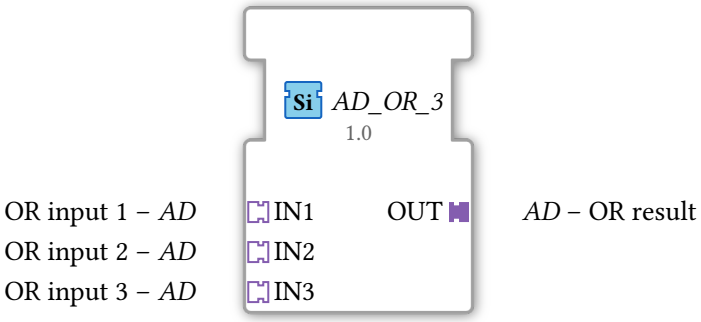
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_OR_2_UNGATED offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_3



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_3 is a generic function block for the bitwise OR operation across 3 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_3`, which combines individual Boolean truth values, `AD_OR_3` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_OR_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

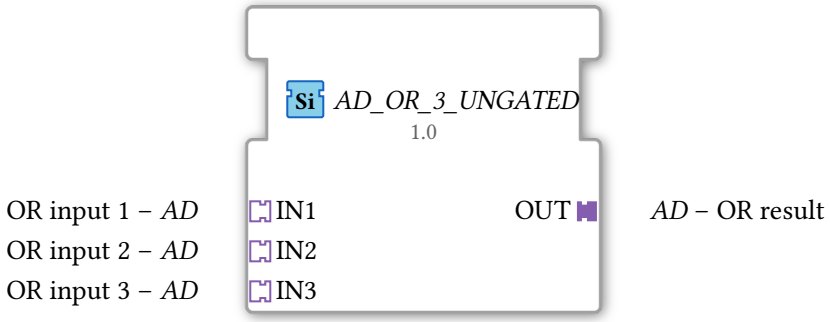
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_OR_3 offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_OR_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_3_UNGATED is a generic function block for the bitwise OR operation across 3 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_3`, which combines individual Boolean truth values, `AD_OR_3_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_OR_3`**: The gated variant – updates the output only on an actual value change.

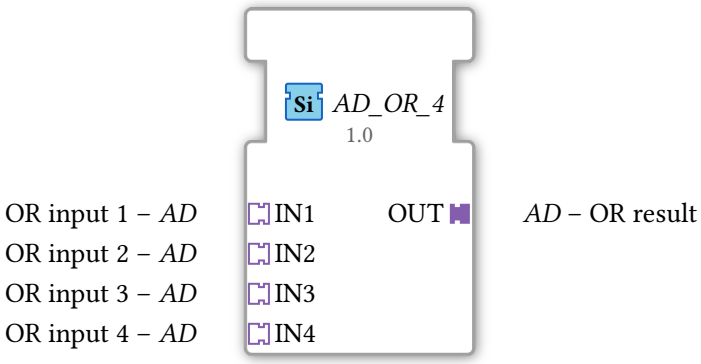
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_OR_3_UNGATED offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_4



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_4 is a generic function block for the bitwise OR operation across 4 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_OR_4`, which combines individual Boolean truth values, `AD_OR_4` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_OR_4_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

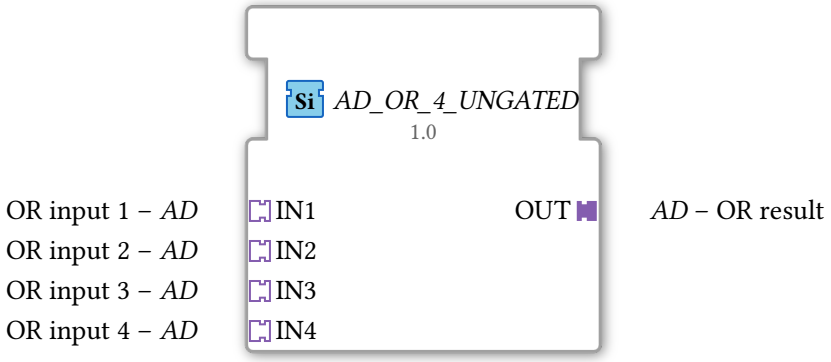
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_OR_4 offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_OR_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_4_UNGATED is a generic function block for the bitwise OR operation across 4 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.

- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_4`, which combines individual Boolean truth values, `AD_OR_4_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_OR_4`: The gated variant – updates the output only on an actual value change.

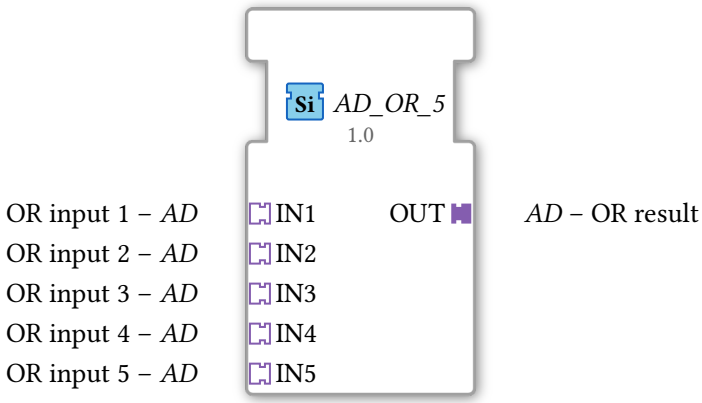
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_OR_4_UNGATED offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_5



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_5 is a generic function block for the bitwise OR operation across 5 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_5 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_5`, which combines individual Boolean truth values, `AD_OR_5` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_5` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_OR_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

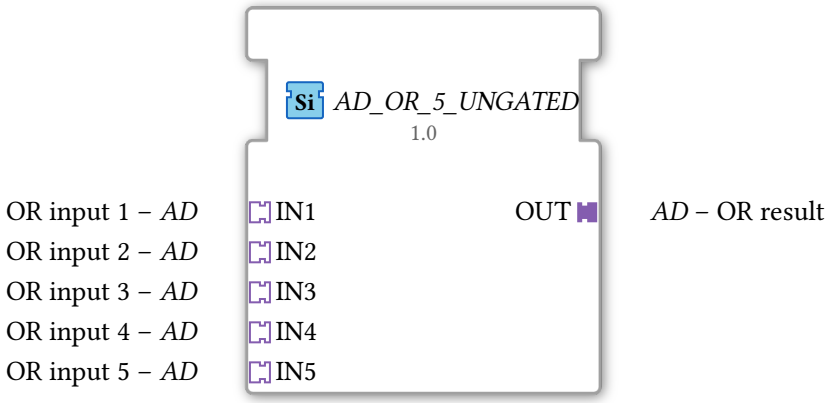
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_OR_5 offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_OR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_5_UNGATED is a generic function block for the bitwise OR operation across 5 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)

- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_5_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_5`, which combines individual Boolean truth values, `AD_OR_5_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_5_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_OR_5`**: The gated variant – updates the output only on an actual value change.

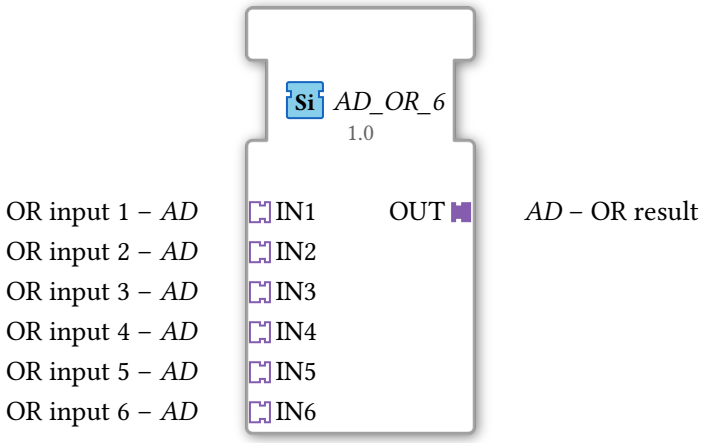
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_OR_5_UNGATED offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_6



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_6 is a generic function block for the bitwise OR operation across 6 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_6 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_6`, which combines individual Boolean truth values, `AD_OR_6` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_6` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_OR_6_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

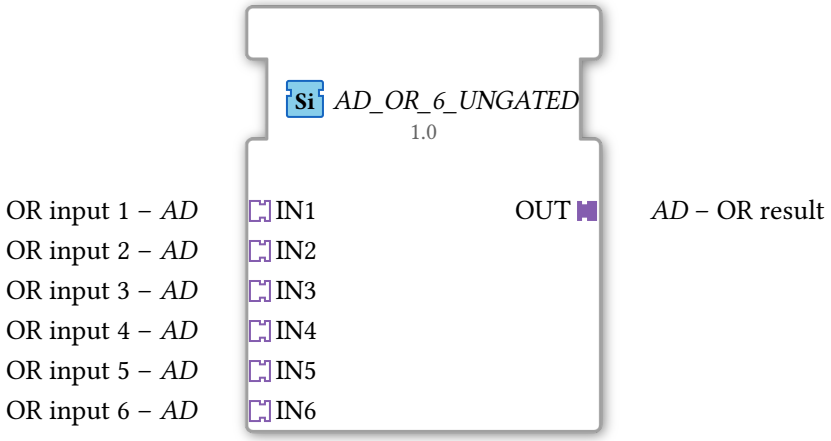
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_OR_6 offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_OR_6. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_6_UNGATED is a generic function block for the bitwise OR operation across 6 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_6_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_6`, which combines individual Boolean truth values, `AD_OR_6_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_6_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_OR_6`: The gated variant – updates the output only on an actual value change.

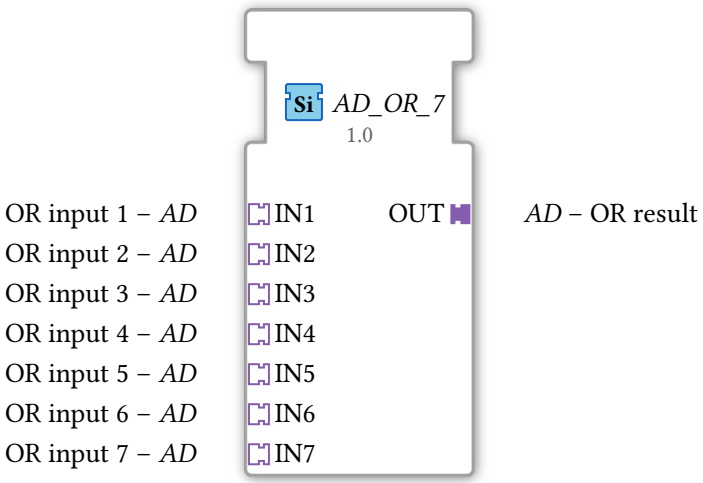
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_OR_6_UNGATED offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_7



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_7 is a generic function block for the bitwise OR operation across 7 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_7 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_OR_7, which combines individual Boolean truth values, AD_OR_7 operates on the full bit pattern of a DWORD value. Compared to the standard block OR_2, AD_OR_7 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AD_OR_7_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

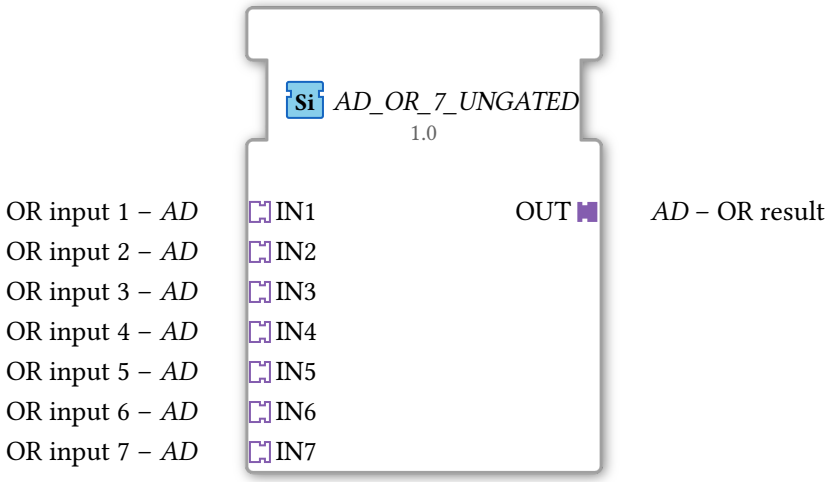
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_OR_7 offers a reliable, generic implementation of the bitwise OR function for DWORD values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_OR_7. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_7_UNGATED is a generic function block for the bitwise OR operation across 7 input values of type DWORD (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the AX_OR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_7_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_7`, which combines individual Boolean truth values, `AD_OR_7_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_7_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_OR_7`: The gated variant – updates the output only on an actual value change.

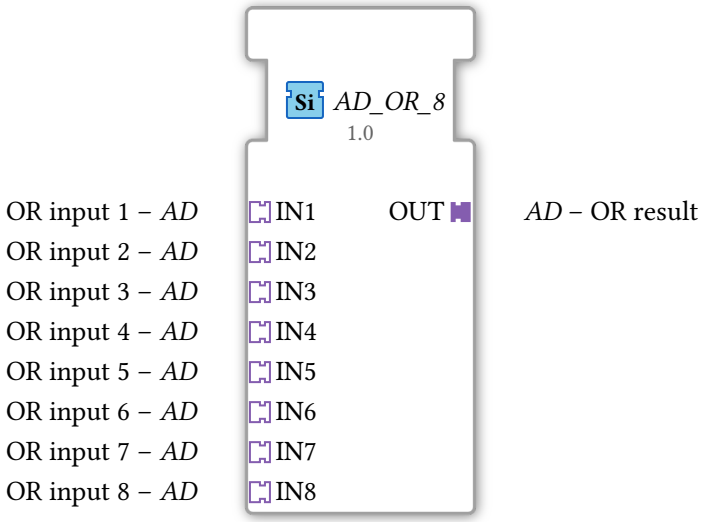
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_OR_7_UNGATED offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_8



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_8 is a generic function block for the bitwise OR operation across 8 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AD)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_8 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_8`, which combines individual Boolean truth values, `AD_OR_8` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_8` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_OR_8_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

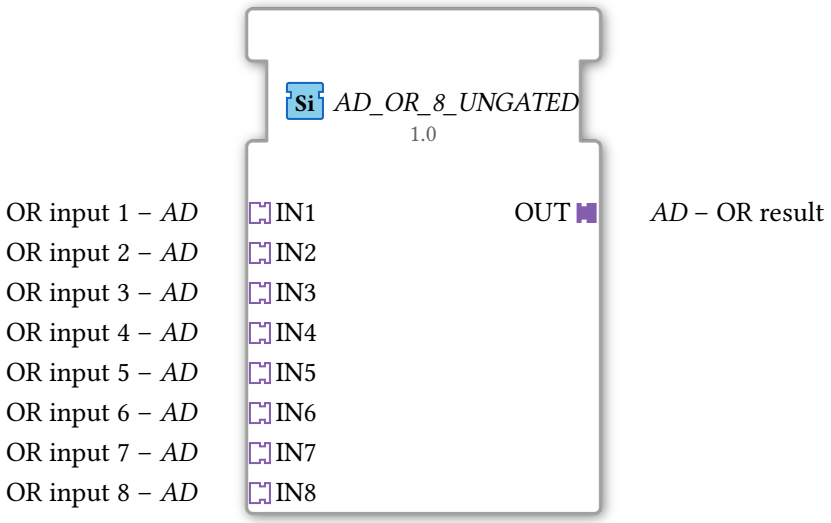
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_OR_8 offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_OR_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_8_UNGATED is a generic function block for the bitwise OR operation across 8 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AD)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_8_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_8`, which combines individual Boolean truth values, `AD_OR_8_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_8_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_OR_8`**: The gated variant – updates the output only on an actual value change.

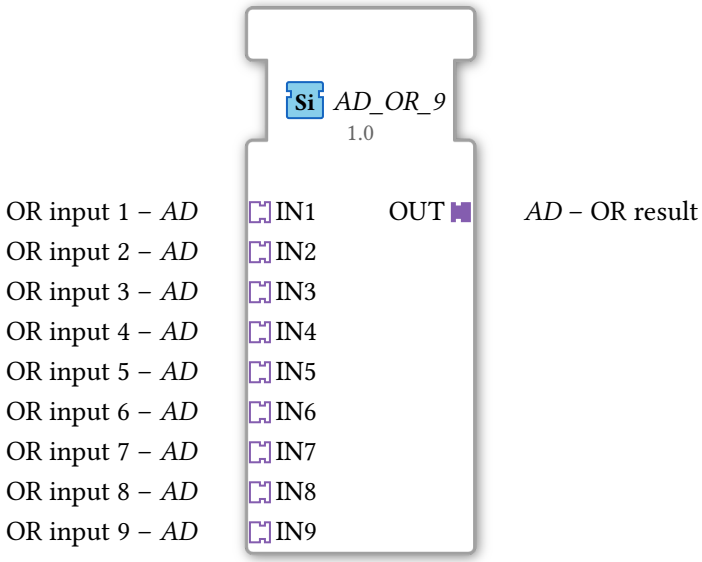
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_OR_8_UNGATED offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_9



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_9 is a generic function block for the bitwise OR operation across 9 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AD)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AD)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 9 input adapters (`IN1` … `IN9`), the block combines the bit patterns of all 9 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_9 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_9`, which combines individual Boolean truth values, `AD_OR_9` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_9` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_OR_9_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

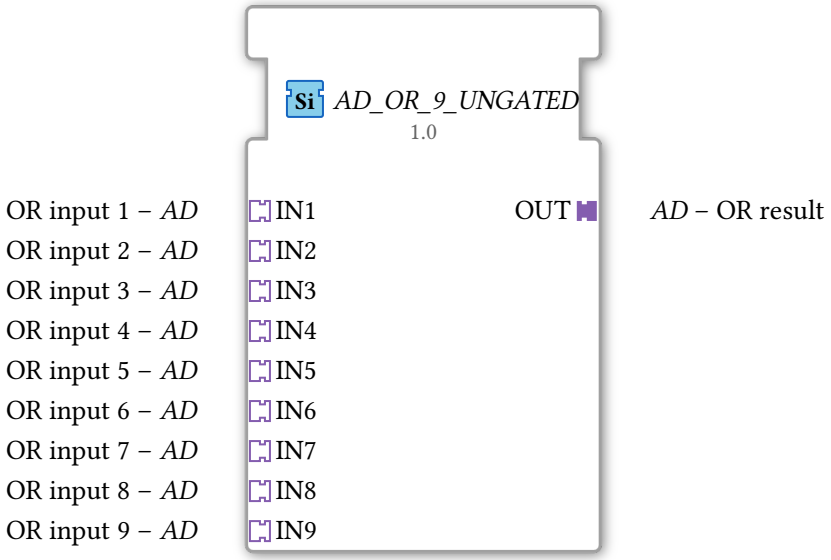
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_OR_9 offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_OR_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_OR_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_OR_9_UNGATED is a generic function block for the bitwise OR operation across 9 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AD)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AD)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 9 input adapters (`IN1` … `IN9`), the block combines the bit patterns of all 9 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the

operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_OR_9_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_9`, which combines individual Boolean truth values, `AD_OR_9_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block OR_2, `AD_OR_9_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_OR_9`: The gated variant – updates the output only on an actual value change.

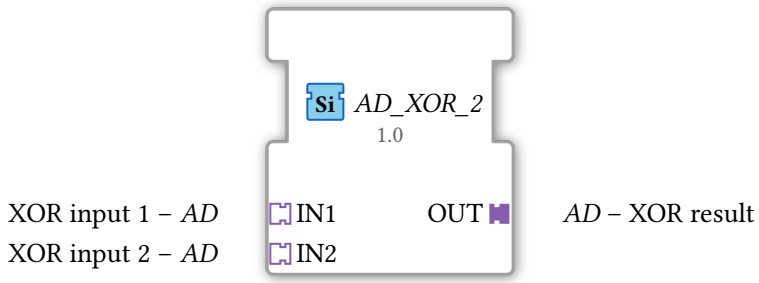
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_OR_9_UNGATED offers a reliable, generic implementation of the bitwise OR function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_2



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_2 is a generic function block for the bitwise XOR (exclusive OR) operation across 2 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_2`, which combines individual Boolean truth values, `AD_XOR_2` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_2` uses adapter-based interfaces

instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_XOR_2_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

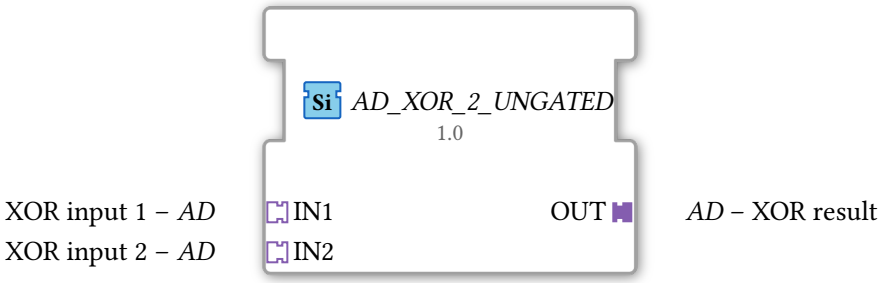
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_XOR_2 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_XOR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_2_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 2 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)

- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_2`, which combines individual Boolean truth values, `AD_XOR_2_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_XOR_2`**: The gated variant – updates the output only on an actual value change.

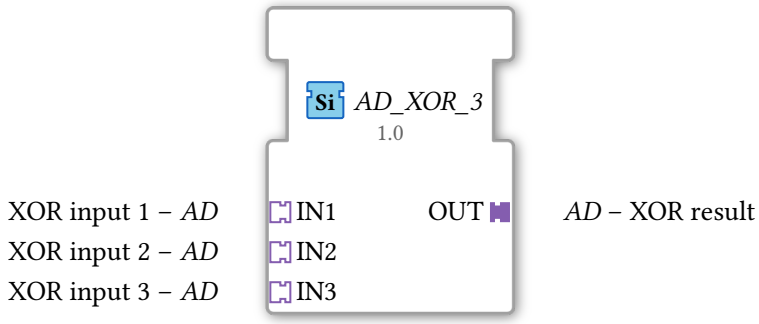
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_XOR_2_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_3



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_3 is a generic function block for the bitwise XOR (exclusive OR) operation across 3 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_3`, which combines individual Boolean truth values, `AD_XOR_3` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_XOR_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

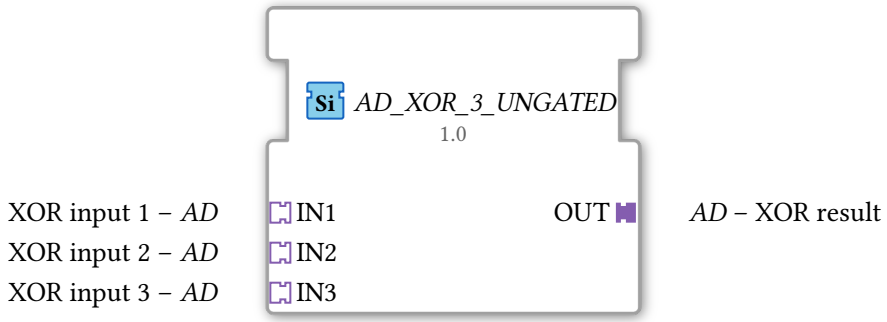
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_XOR_3 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_XOR_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_3_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 3 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality
As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features
- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview
Since this is a combinational logic block, AD_XOR_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios
- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks
Unlike `AX_XOR_3`, which combines individual Boolean truth values, `AD_XOR_3_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_XOR_3`**: The gated variant – updates the output only on an actual value change.

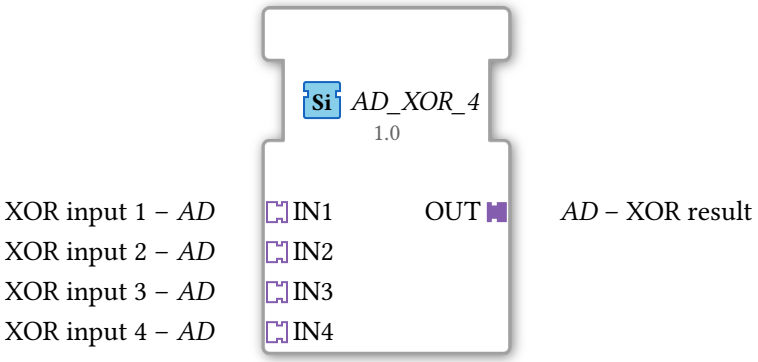
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_XOR_3_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_4



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_4 is a generic function block for the bitwise XOR (exclusive OR) operation across 4 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The

starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_4`, which combines individual Boolean truth values, `AD_XOR_4` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_XOR_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

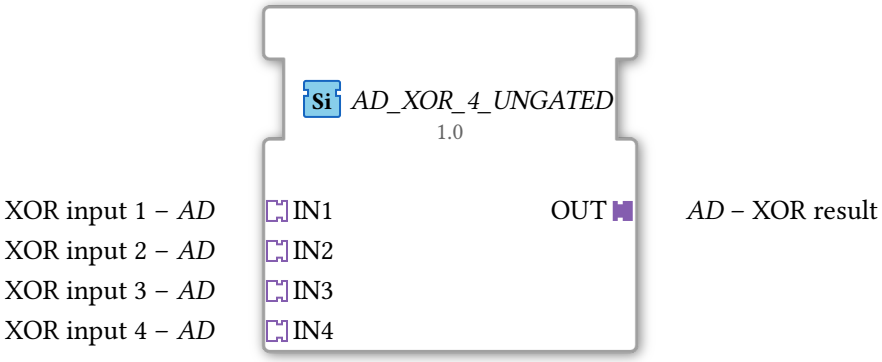
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_XOR_4 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_XOR_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_4_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 4 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.

- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_4`, which combines individual Boolean truth values, `AD_XOR_4_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_XOR_4`: The gated variant – updates the output only on an actual value change.

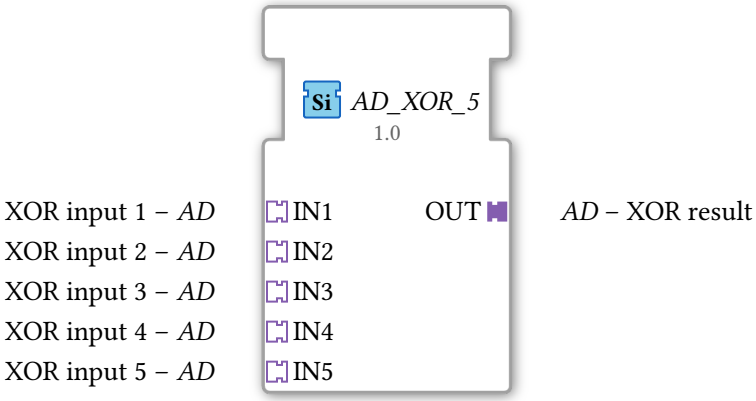
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_XOR_4_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_5



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_5 is a generic function block for the bitwise XOR (exclusive OR) operation across 5 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_5 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_5`, which combines individual Boolean truth values, `AD_XOR_5` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_5` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_XOR_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

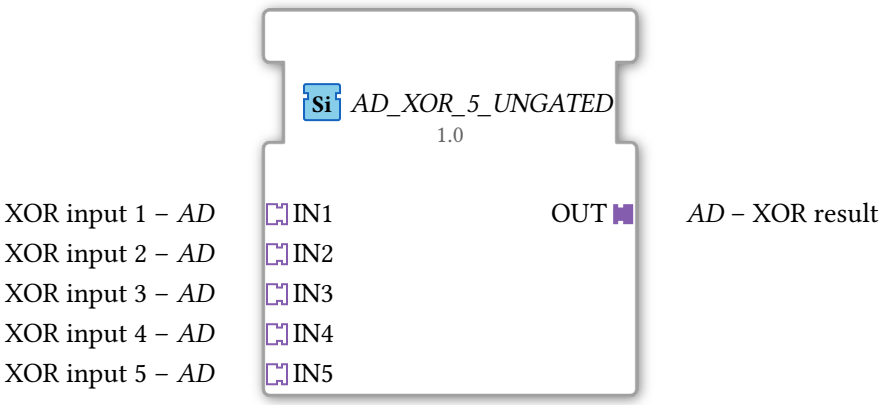
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_XOR_5 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_XOR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_5_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 5 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)

- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_5_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_5`, which combines individual Boolean truth values, `AD_XOR_5_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_5_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_XOR_5`**: The gated variant – updates the output only on an actual value change.

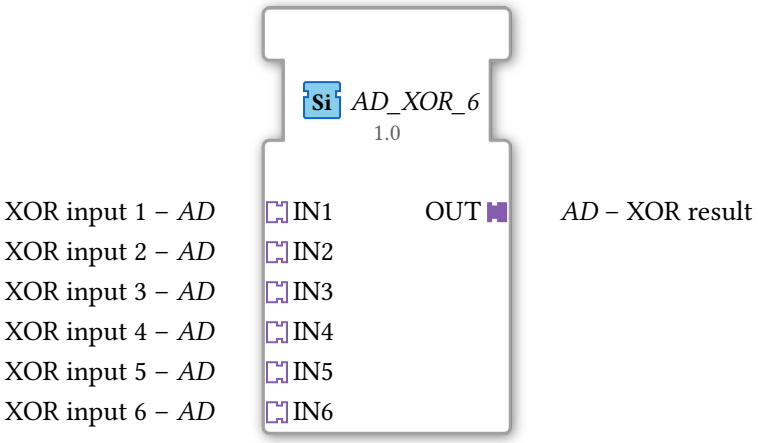
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_XOR_5_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_6



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_6 is a generic function block for the bitwise XOR (exclusive OR) operation across 6 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_6 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_6`, which combines individual Boolean truth values, `AD_XOR_6` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_6` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_XOR_6_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

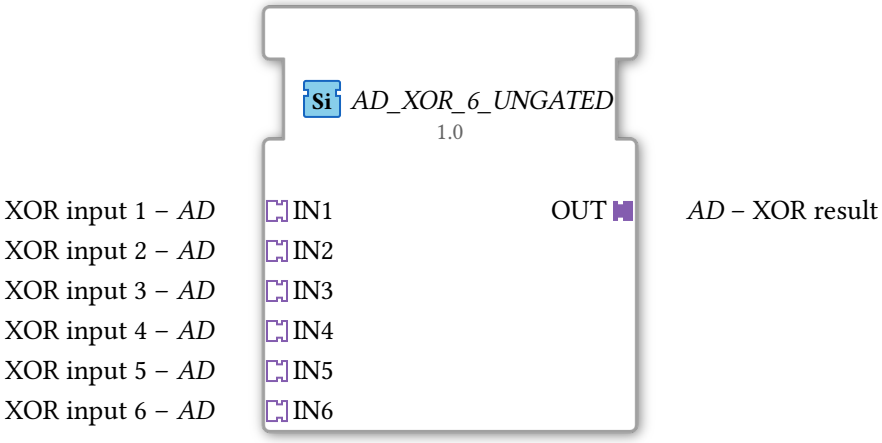
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_XOR_6 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_XOR_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_6_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 6 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_6_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_XOR_6`, which combines individual Boolean truth values, `AD_XOR_6_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_6_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_XOR_6`: The gated variant – updates the output only on an actual value change.

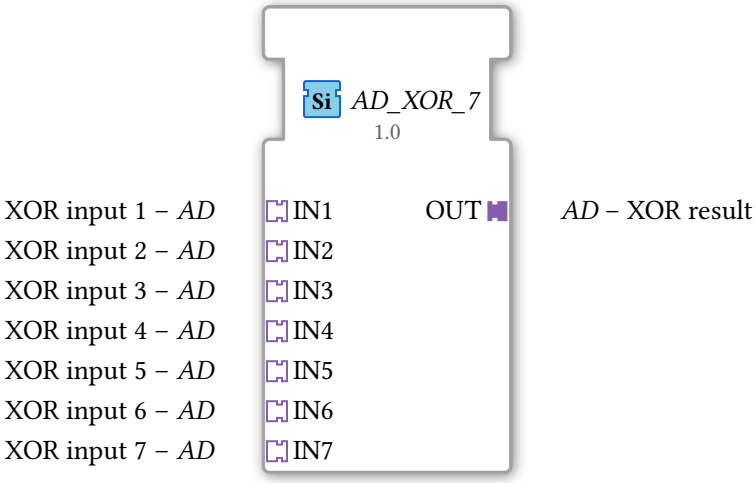
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_XOR_6_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_7



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_7 is a generic function block for the bitwise XOR (exclusive OR) operation across 7 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_7 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_XOR_7, which combines individual Boolean truth values, AD_XOR_7 operates on the full bit pattern of a DWORD value. Compared to the standard block XOR_2, AD_XOR_7 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AD_XOR_7_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

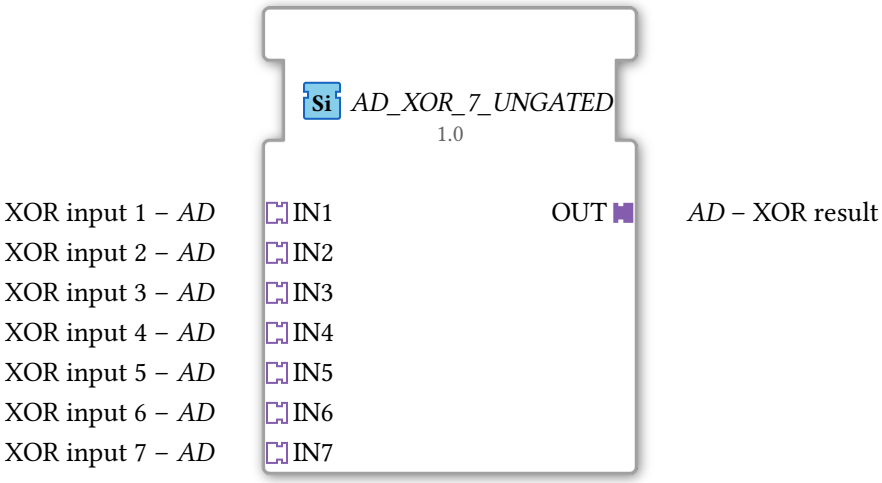
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_XOR_7 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for DWORD values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_XOR_7. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_7_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 7 input values of type DWORD (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the AX_XOR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_7_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_7`, which combines individual Boolean truth values, `AD_XOR_7_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_7_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_XOR_7`**: The gated variant – updates the output only on an actual value change.

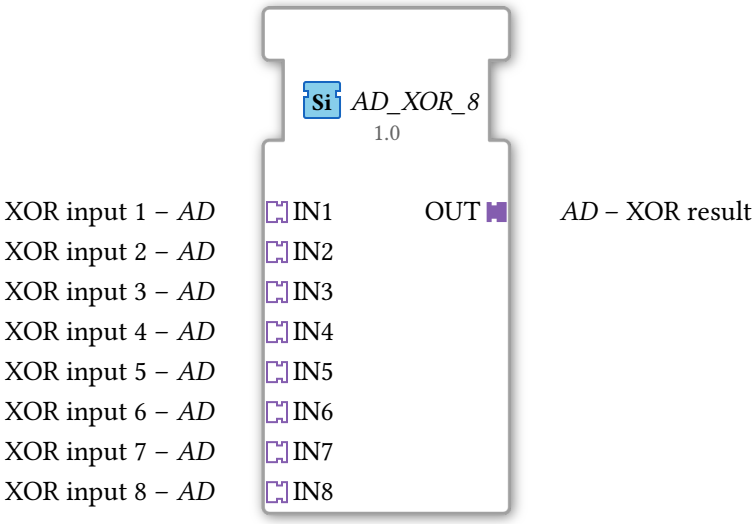
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_XOR_7_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_8



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_8 is a generic function block for the bitwise XOR (exclusive OR) operation across 8 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AD)
- **IN8**: XOR (exclusive OR) input 8 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_8 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_8`, which combines individual Boolean truth values, `AD_XOR_8` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_8` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AD_XOR_8_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

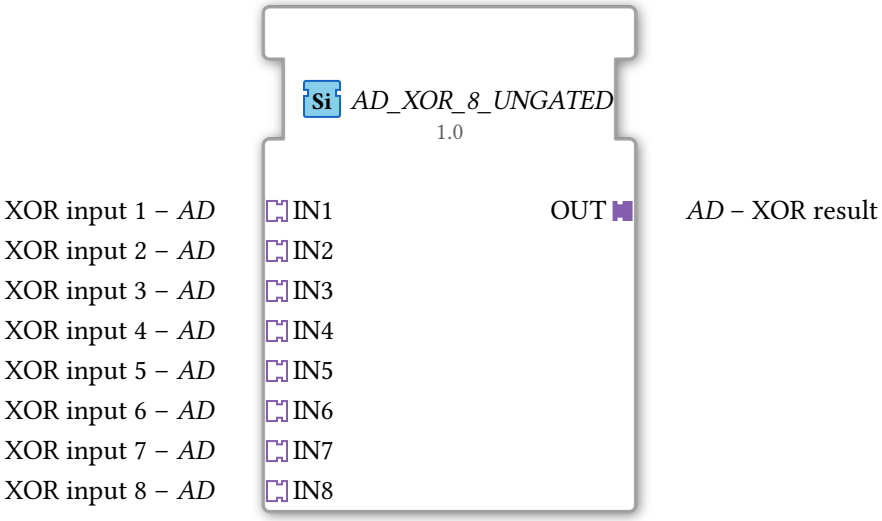
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AD_XOR_8 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AD_XOR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_XOR_8. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_XOR_8_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 8 input values of type `DWORD` (32-bit bit pattern (double word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AD)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AD)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AD)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AD)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AD)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AD)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AD)
- **IN8**: XOR (exclusive OR) input 8 (Type: adapter::types::unidirectional::AD)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AD)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AD_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `DWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AD` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AD_XOR_8_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `DWORD` into a single result.
- **Signal aggregation**: Merging multiple `DWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_8`, which combines individual Boolean truth values, `AD_XOR_8_UNGATED` operates on the full bit pattern of a `DWORD` value. Compared to the standard block XOR_2, `AD_XOR_8_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AD_XOR_8`: The gated variant – updates the output only on an actual value change.

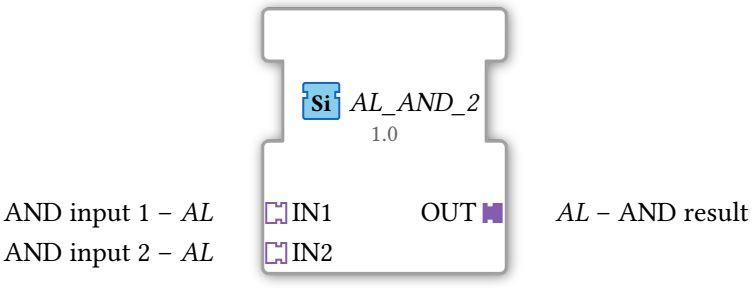
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_XOR_8_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `DWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_AND_2



• – ◦ • – ◦ • – ◦ •

Introduction

AL_AND_2 is a generic function block for the bitwise AND operation across 2 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AL)

Functionality
As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features
- **Generic function block**: The FB is defined as a generic type (`GEN_AL_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview
Since this is a combinational logic block, AL_AND_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios
- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks
Unlike `AX_AND_2`, which combines individual Boolean truth values, `AL_AND_2` operates on the full bit pattern of a `LWORD` value. Compared to the standard block AND_2, `AL_AND_2` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_AND_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

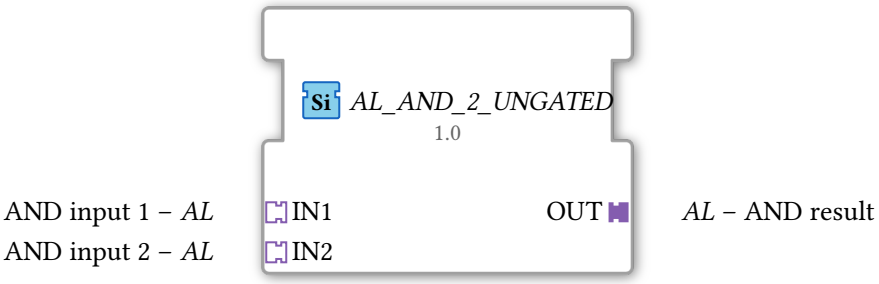
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_AND_2 offers a reliable, generic implementation of the bitwise AND function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_AND_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_AND_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_AND_2_UNGATED is a generic function block for the bitwise AND operation across 2 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_AND_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_2`, which combines individual Boolean truth values, `AL_AND_2_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block AND_2, `AL_AND_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_AND_2`**: The gated variant – updates the output only on an actual value change.

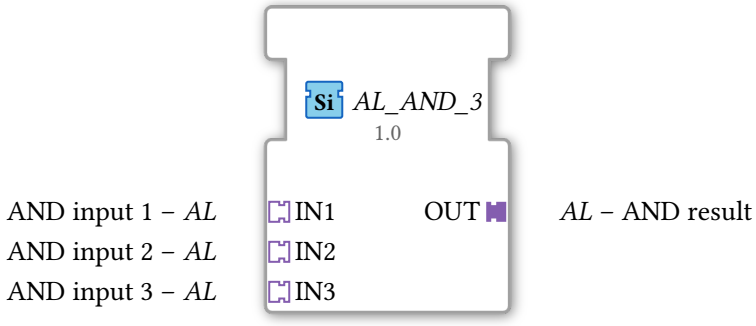
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_AND_2_UNGATED offers a reliable, generic implementation of the bitwise AND function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_AND_3



• – ◦ • – ◦ • – ◦ •

Introduction

AL_AND_3 is a generic function block for the bitwise AND operation across 3 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_AND_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_AND_3, which combines individual Boolean truth values, AL_AND_3 operates on the full bit pattern of a LWORD value. Compared to the standard block AND_2, AL_AND_3 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AL_AND_3_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

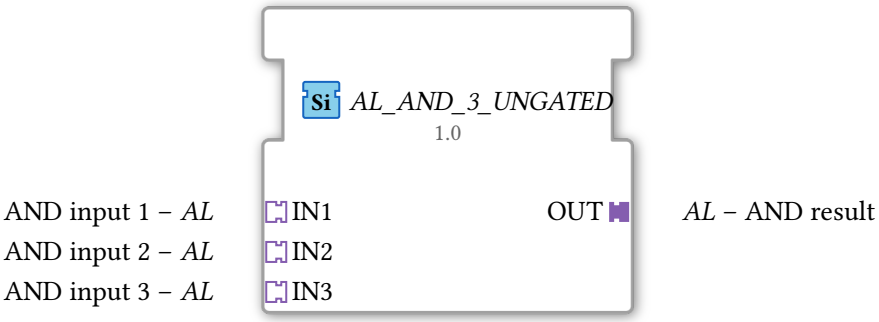
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_AND_3 offers a reliable, generic implementation of the bitwise AND function for LWORD values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_AND_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_AND_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_AND_3_UNGATED is a generic function block for the bitwise AND operation across 3 input values of type LWORD (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the AX_AND blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_AND_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_3`, which combines individual Boolean truth values, `AL_AND_3_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block AND_2, `AL_AND_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_AND_3`**: The gated variant – updates the output only on an actual value change.

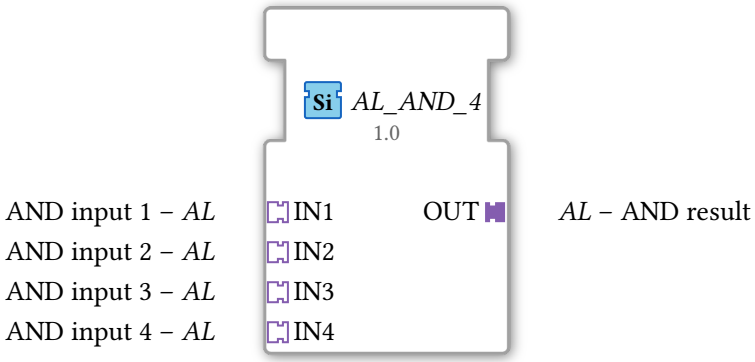
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_AND_3_UNGATED offers a reliable, generic implementation of the bitwise AND function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_AND_4



• – ◦ • – ◦ • – ◦ •

Introduction

AL_AND_4 is a generic function block for the bitwise AND operation across 4 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: AND input 4 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.

- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_AND_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_4`, which combines individual Boolean truth values, `AL_AND_4` operates on the full bit pattern of a `LWORD` value. Compared to the standard block AND_2, `AL_AND_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_AND_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

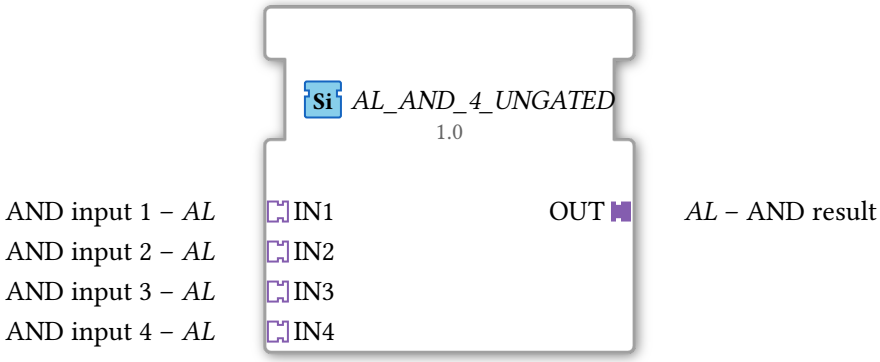
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_AND_4 offers a reliable, generic implementation of the bitwise AND function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_AND_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_AND_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_AND_4_UNGATED is a generic function block for the bitwise AND operation across 4 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: AND input 4 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_AND_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_AND_4, which combines individual Boolean truth values, AL_AND_4_UNGATED operates on the full bit pattern of a LWORD value. Compared to the standard block AND_2, AL_AND_4_UNGATED uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AL_AND_4**: The gated variant – updates the output only on an actual value change.

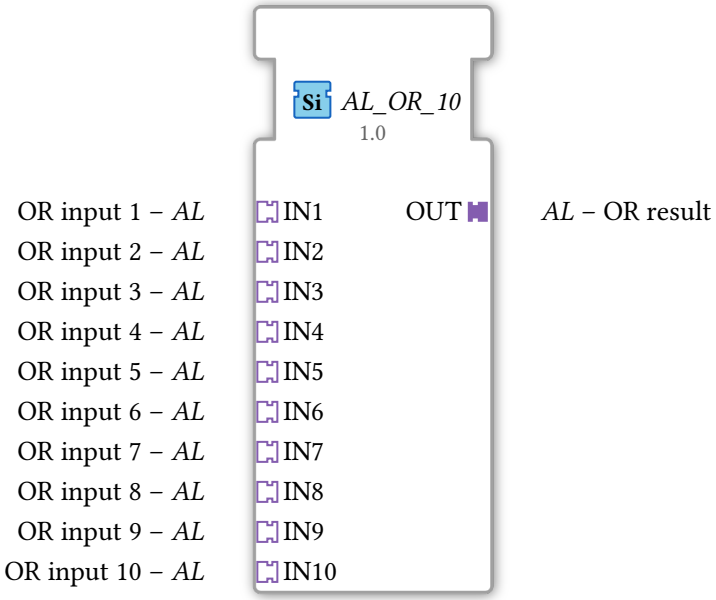
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_AND_4_UNGATED offers a reliable, generic implementation of the bitwise AND function for LWORD values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_10



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_10 is a generic function block for the bitwise OR operation across 10 input values of type LWORD (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the AX_OR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AL)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AL)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AL)
- **IN10**: OR input 10 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 10 input adapters (`IN1` … `IN10`), the block combines the bit patterns of all 10 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_10 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_10`, which combines individual Boolean truth values, `AL_OR_10` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_10` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_10_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

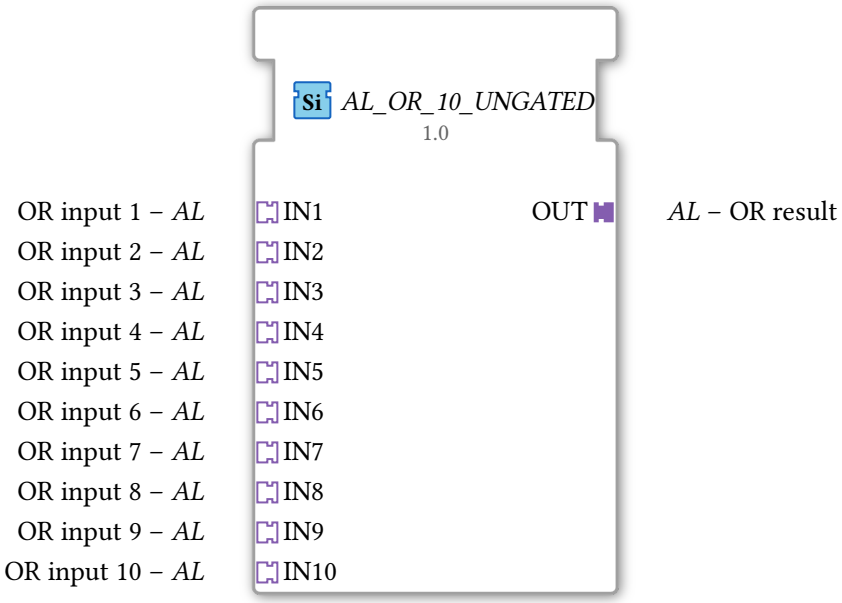
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_OR_10 offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_10_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_OR_10`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_10_UNGATED is a generic function block for the bitwise OR operation across 10 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)

- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AL)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AL)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AL)
- **IN10**: OR input 10 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 10 input adapters (`IN1` … `IN10`), the block combines the bit patterns of all 10 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_10_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_10`, which combines individual Boolean truth values, `AL_OR_10_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_10_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_10`**: The gated variant – updates the output only on an actual value change.

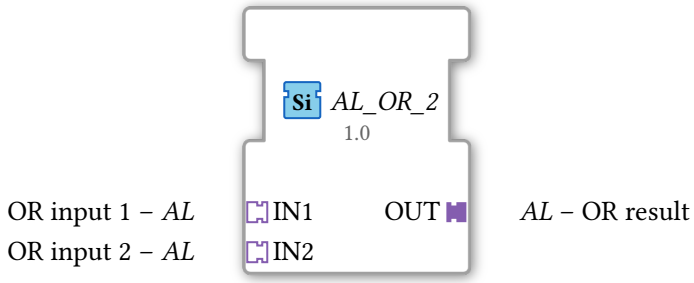
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_OR_10_UNGATED offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_2



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_2 is a generic function block for the bitwise OR operation across 2 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_2`, which combines individual Boolean truth values, `AL_OR_2` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_2` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

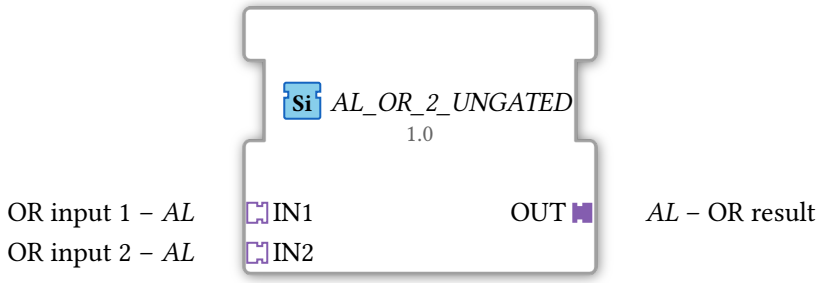
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_OR_2 offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_OR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_2_UNGATED is a generic function block for the bitwise OR operation across 2 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_2`, which combines individual Boolean truth values, `AL_OR_2_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AL_OR_2`: The gated variant – updates the output only on an actual value change.

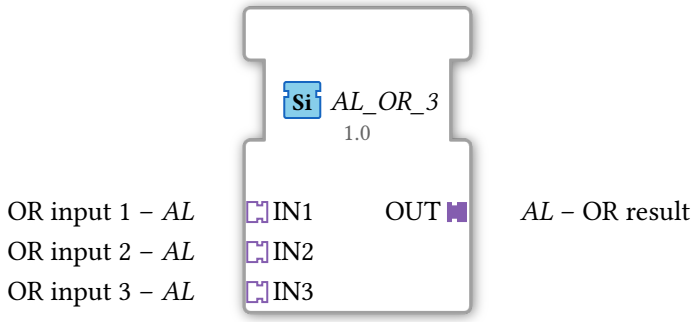
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_OR_2_UNGATED offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_3



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_3 is a generic function block for the bitwise OR operation across 3 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_3`, which combines individual Boolean truth values, `AL_OR_3` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

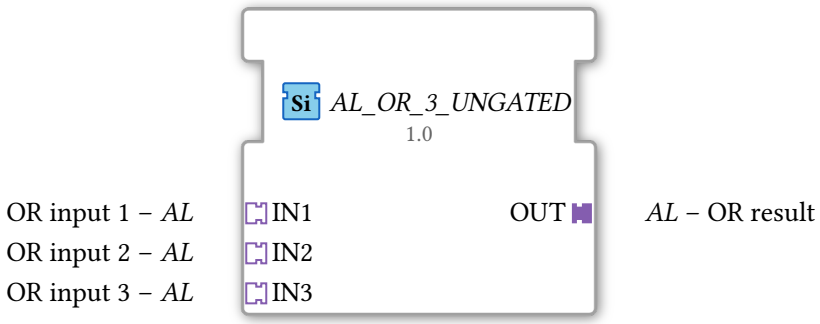
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_OR_3 offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_OR_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_3_UNGATED is a generic function block for the bitwise OR operation across 3 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_OR_3, which combines individual Boolean truth values, AL_OR_3_UNGATED operates on the full bit pattern of a LWORD value. Compared to the standard block OR_2, AL_OR_3_UNGATED uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AL_OR_3**: The gated variant – updates the output only on an actual value change.

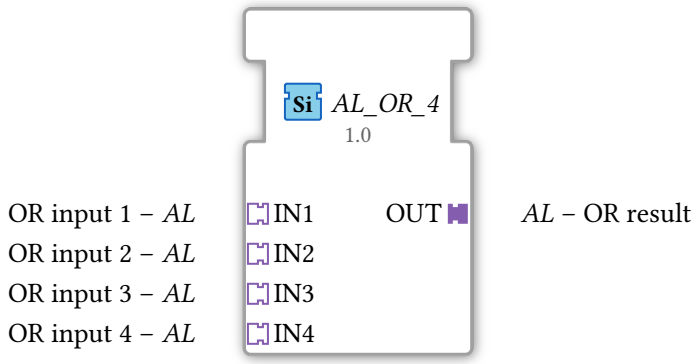
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_OR_3_UNGATED offers a reliable, generic implementation of the bitwise OR function for LWORD values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_4



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_4 is a generic function block for the bitwise OR operation across 4 input values of type LWORD (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the AX_OR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_OR_4`, which combines individual Boolean truth values, `AL_OR_4` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

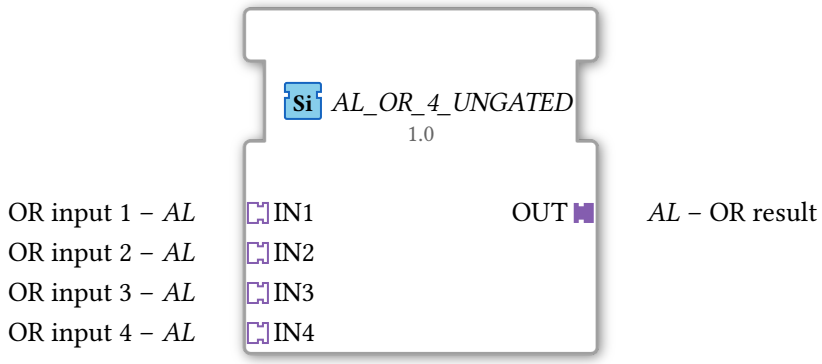
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_OR_4 offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_OR_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_4_UNGATED is a generic function block for the bitwise OR operation across 4 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.

- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_OR_4`, which combines individual Boolean truth values, `AL_OR_4_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_4`**: The gated variant – updates the output only on an actual value change.

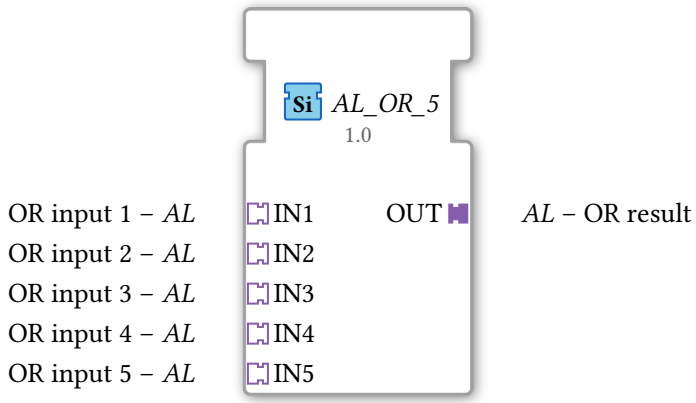
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_OR_4_UNGATED offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_5



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_5 is a generic function block for the bitwise OR operation across 5 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_5 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_5`, which combines individual Boolean truth values, `AL_OR_5` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_5` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

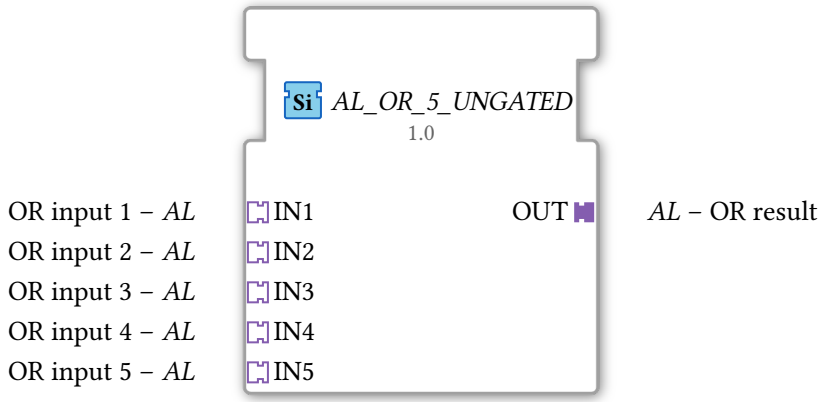
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_OR_5 offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_OR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_5_UNGATED is a generic function block for the bitwise OR operation across 5 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)

- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_5_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_5`, which combines individual Boolean truth values, `AL_OR_5_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_5_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AL_OR_5`: The gated variant – updates the output only on an actual value change.

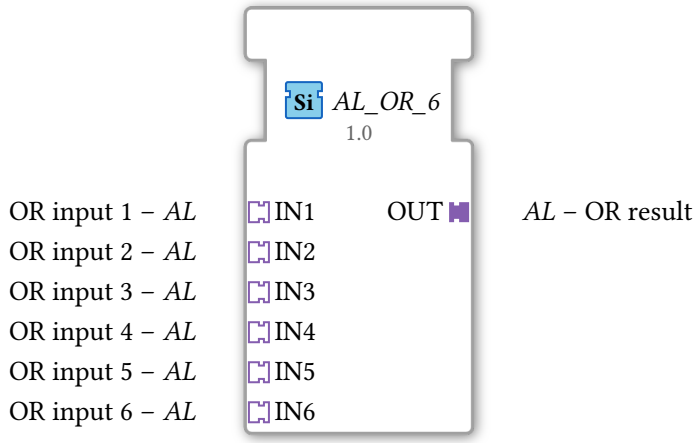
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_OR_5_UNGATED offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_6



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_6 is a generic function block for the bitwise OR operation across 6 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_6 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_6`, which combines individual Boolean truth values, `AL_OR_6` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_6` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_6_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

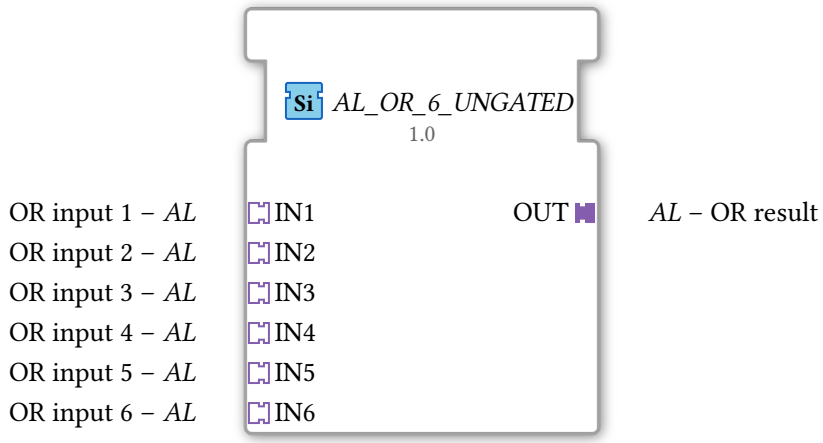
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_OR_6 offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_OR_6. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_6_UNGATED is a generic function block for the bitwise OR operation across 6 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_6_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_6`, which combines individual Boolean truth values, `AL_OR_6_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_6_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AL_OR_6`: The gated variant – updates the output only on an actual value change.

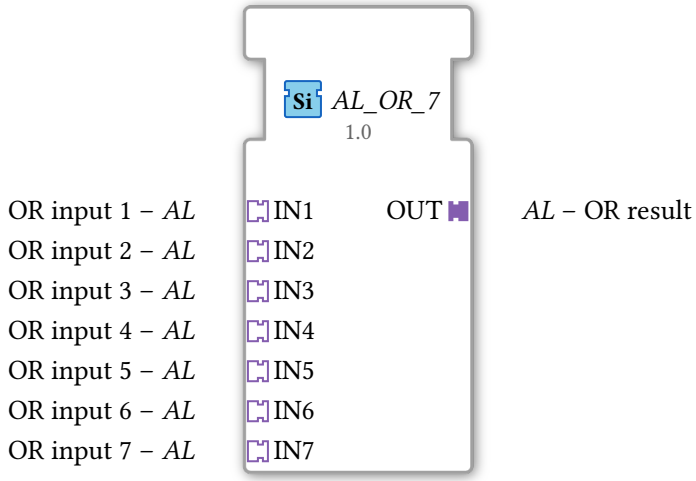
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_OR_6_UNGATED offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_7



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_7 is a generic function block for the bitwise OR operation across 7 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_7 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_OR_7, which combines individual Boolean truth values, AL_OR_7 operates on the full bit pattern of a LWORD value. Compared to the standard block OR_2, AL_OR_7 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AL_OR_7_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

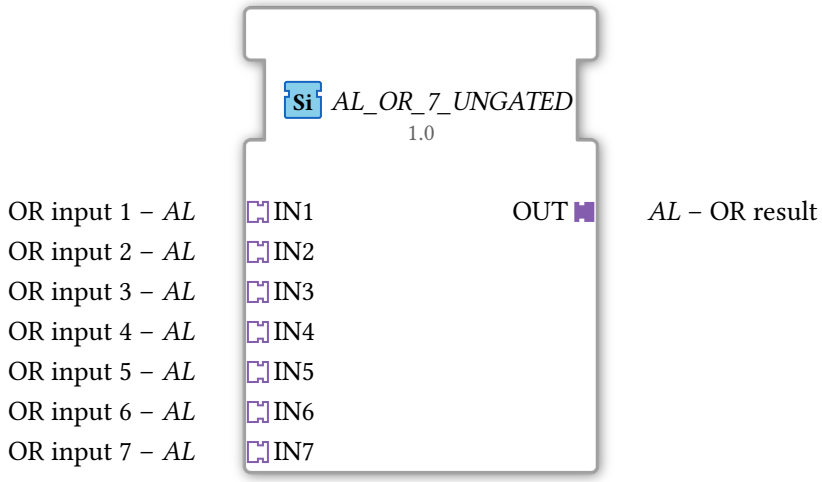
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_OR_7 offers a reliable, generic implementation of the bitwise OR function for LWORD values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_OR_7. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_7_UNGATED is a generic function block for the bitwise OR operation across 7 input values of type LWORD (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the AX_OR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_7_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_7`, which combines individual Boolean truth values, `AL_OR_7_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_7_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_7`**: The gated variant – updates the output only on an actual value change.

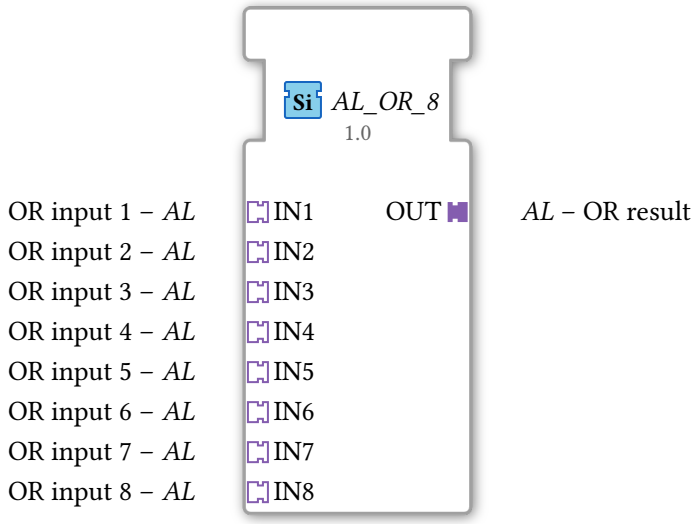
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_OR_7_UNGATED offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_8



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_8 is a generic function block for the bitwise OR operation across 8 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AL)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_8 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_8`, which combines individual Boolean truth values, `AL_OR_8` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_8` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AL_OR_8_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

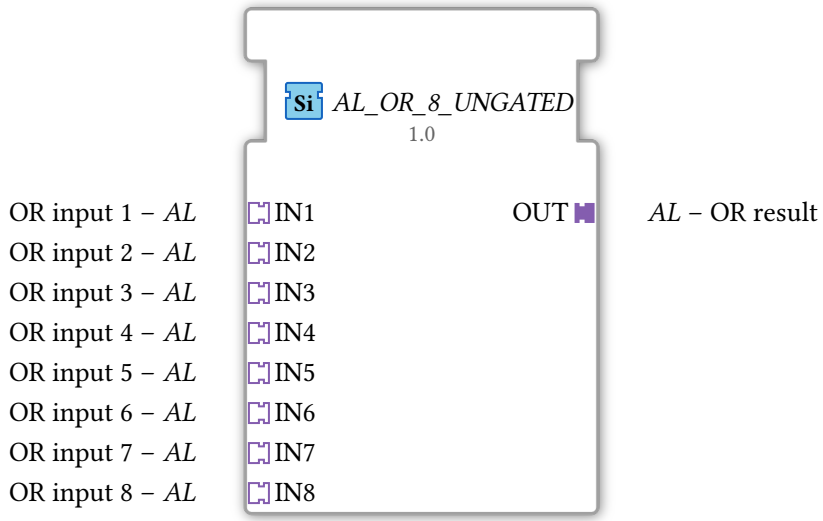
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_OR_8 offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_OR_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_8_UNGATED is a generic function block for the bitwise OR operation across 8 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AL)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_8_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_8`, which combines individual Boolean truth values, `AL_OR_8_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_8_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_8`**: The gated variant – updates the output only on an actual value change.

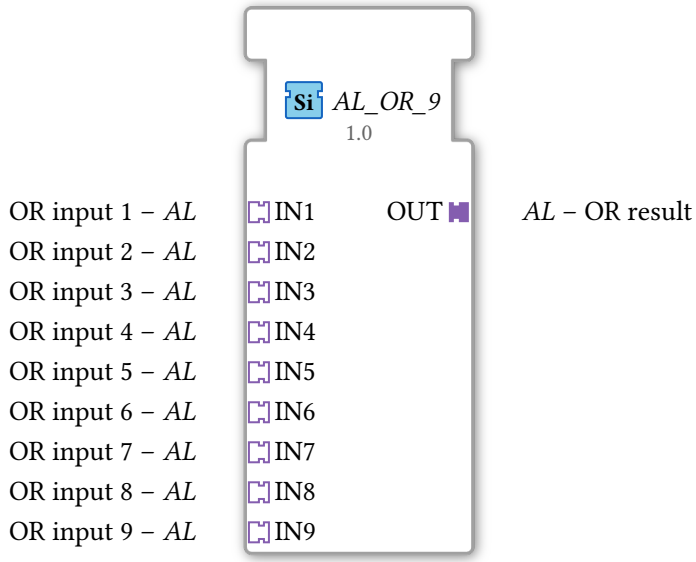
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_OR_8_UNGATED offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_9



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_9 is a generic function block for the bitwise OR operation across 9 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AL)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AL)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 9 input adapters (`IN1` … `IN9`), the block combines the bit patterns of all 9 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_9 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_9`, which combines individual Boolean truth values, `AL_OR_9` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_9` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_OR_9_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

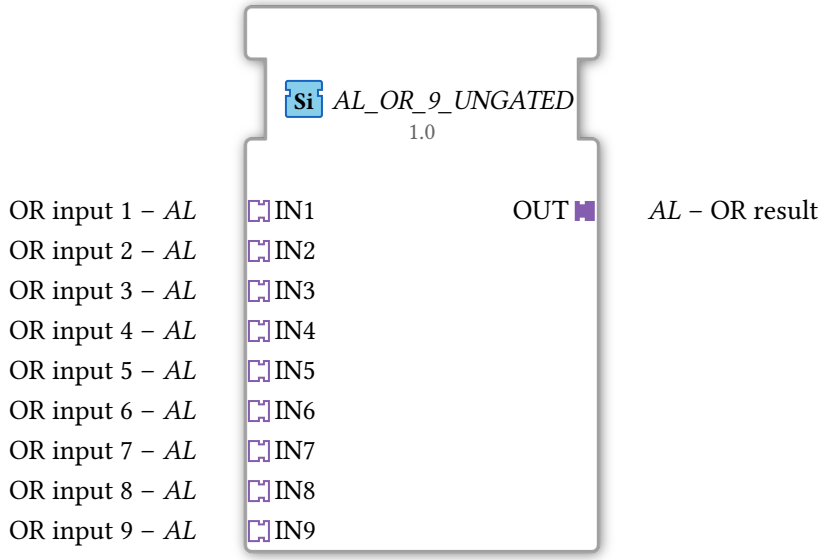
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_OR_9 offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_OR_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_OR_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_OR_9_UNGATED is a generic function block for the bitwise OR operation across 9 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AL)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AL)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 9 input adapters (`IN1` … `IN9`), the block combines the bit patterns of all 9 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the

operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_OR_9_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_9`, which combines individual Boolean truth values, `AL_OR_9_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block OR_2, `AL_OR_9_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AL_OR_9`: The gated variant – updates the output only on an actual value change.

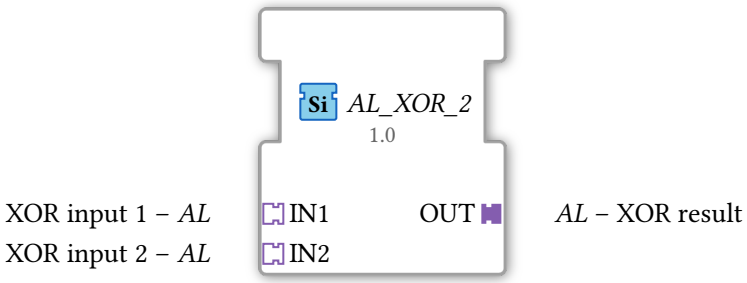
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_OR_9_UNGATED offers a reliable, generic implementation of the bitwise OR function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_2



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_2 is a generic function block for the bitwise XOR (exclusive OR) operation across 2 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_2`, which combines individual Boolean truth values, `AL_XOR_2` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_2` uses adapter-based interfaces

instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AL_XOR_2_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

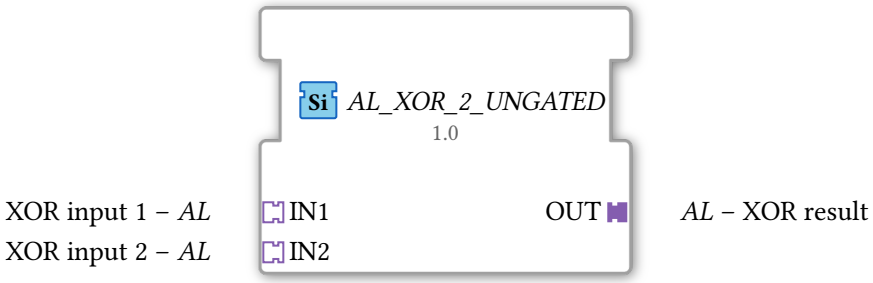
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_XOR_2 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_XOR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_2_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 2 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)

- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_2`, which combines individual Boolean truth values, `AL_XOR_2_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_XOR_2`**: The gated variant – updates the output only on an actual value change.

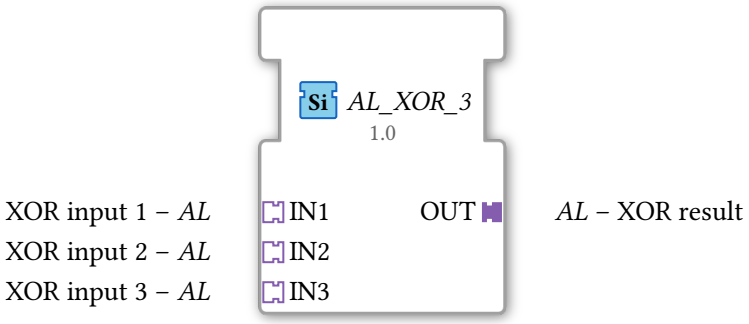
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_XOR_2_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_3



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_3 is a generic function block for the bitwise XOR (exclusive OR) operation across 3 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_3`, which combines individual Boolean truth values, `AL_XOR_3` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_XOR_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

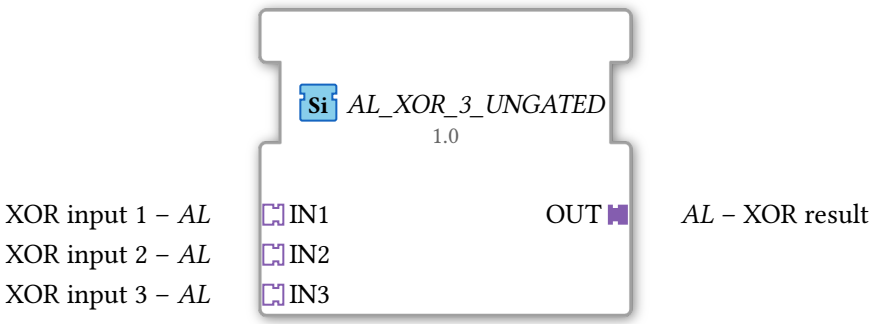
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_XOR_3 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_XOR_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_3_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 3 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality
As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features
- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview
Since this is a combinational logic block, AL_XOR_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios
- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks
Unlike `AX_XOR_3`, which combines individual Boolean truth values, `AL_XOR_3_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_XOR_3`**: The gated variant – updates the output only on an actual value change.

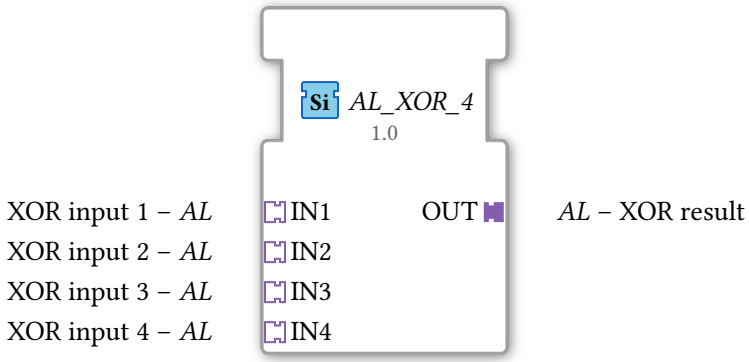
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_XOR_3_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_4



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_4 is a generic function block for the bitwise XOR (exclusive OR) operation across 4 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The

starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on OUT is OUT rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (GEN_AL_XOR) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean AX_XOR blocks, every bit of the LWORD data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type unidirectional::AL – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type LWORD into a single result.
- **Signal aggregation**: Merging multiple LWORD data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_XOR_4, which combines individual Boolean truth values, AL_XOR_4 operates on the full bit pattern of a LWORD value. Compared to the standard block XOR_2, AL_XOR_4 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AL_XOR_4_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

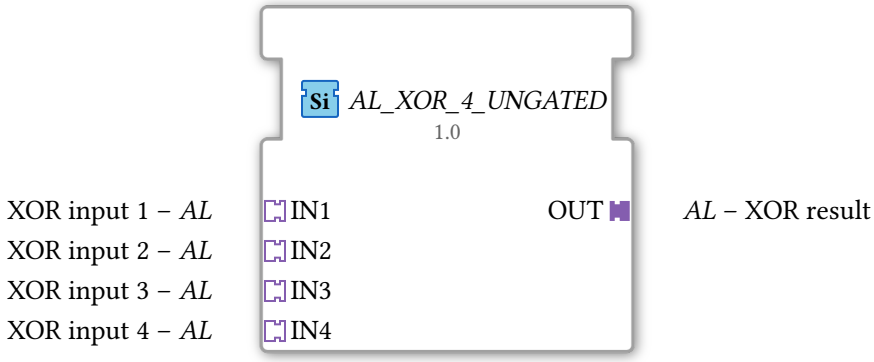
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_XOR_4 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for LWORD values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_XOR_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_4_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 4 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.

- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_4`, which combines individual Boolean truth values, `AL_XOR_4_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_XOR_4`**: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_XOR_4_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_5



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_5 is a generic function block for the bitwise XOR (exclusive OR) operation across 5 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_5 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_5`, which combines individual Boolean truth values, `AL_XOR_5` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_5` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_XOR_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

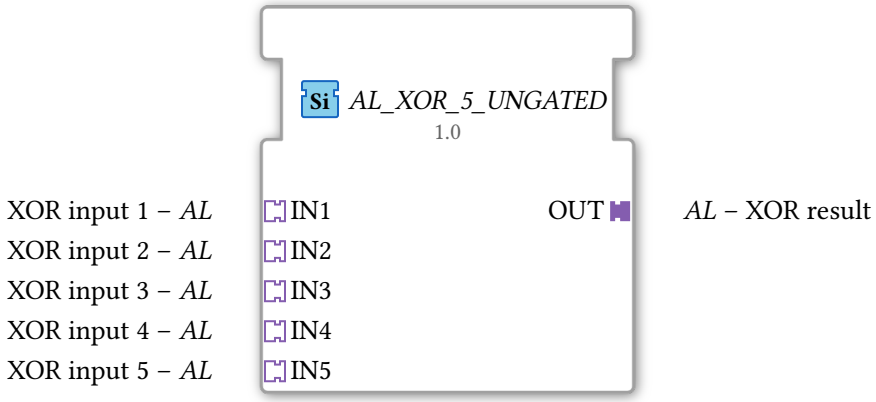
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_XOR_5 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_XOR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_5_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 5 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)

- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_5_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_5`, which combines individual Boolean truth values, `AL_XOR_5_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_5_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_XOR_5`**: The gated variant – updates the output only on an actual value change.

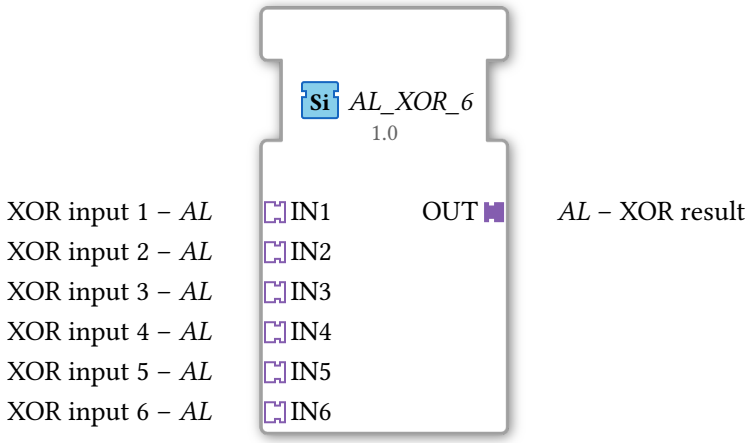
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_XOR_5_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_6



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_6 is a generic function block for the bitwise XOR (exclusive OR) operation across 6 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_6 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_6`, which combines individual Boolean truth values, `AL_XOR_6` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_6` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_XOR_6_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

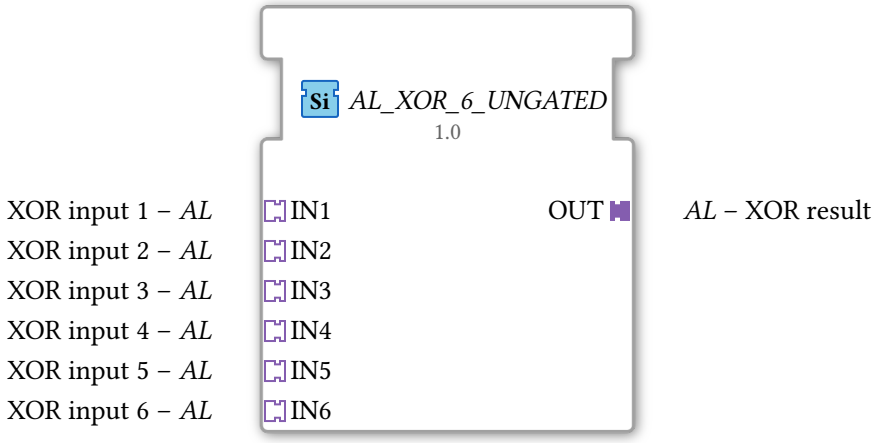
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_XOR_6 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_XOR_6. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_6_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 6 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_6_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_6`, which combines individual Boolean truth values, `AL_XOR_6_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_6_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AL_XOR_6`: The gated variant – updates the output only on an actual value change.

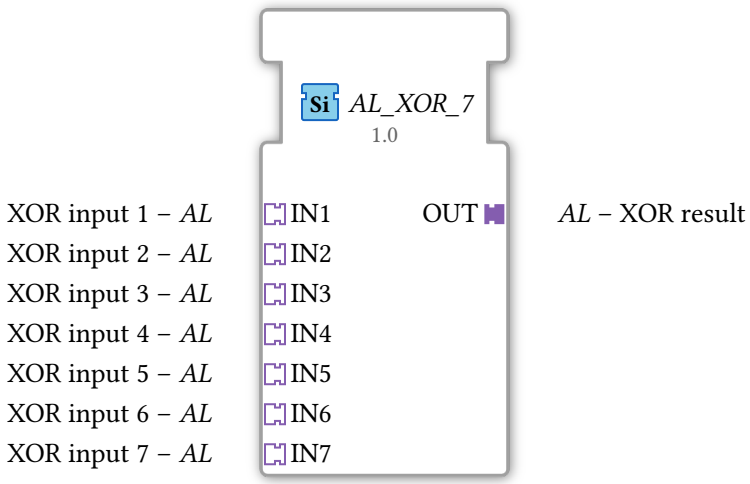
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_XOR_6_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_7



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_7 is a generic function block for the bitwise XOR (exclusive OR) operation across 7 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_7 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_XOR_7, which combines individual Boolean truth values, AL_XOR_7 operates on the full bit pattern of a LWORD value. Compared to the standard block XOR_2, AL_XOR_7 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AL_XOR_7_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

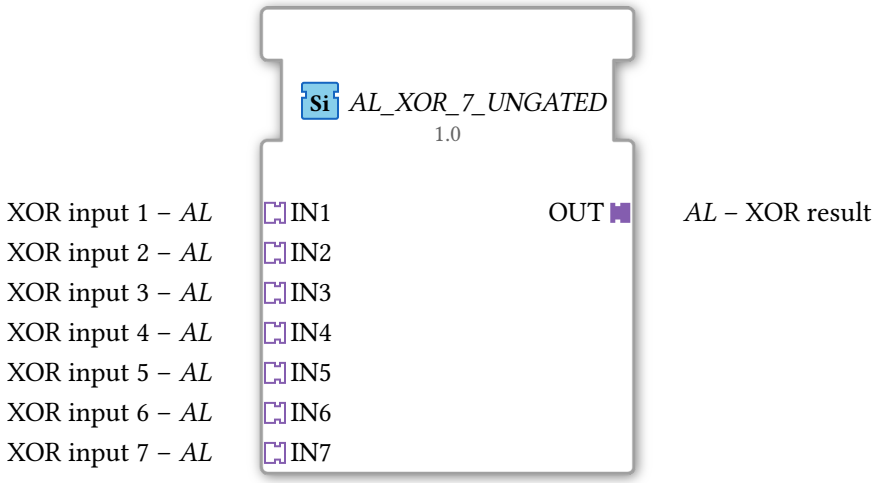
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_XOR_7 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for LWORD values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_XOR_7. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_7_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 7 input values of type LWORD (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the AX_XOR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_7_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_7`, which combines individual Boolean truth values, `AL_XOR_7_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_7_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_XOR_7`**: The gated variant – updates the output only on an actual value change.

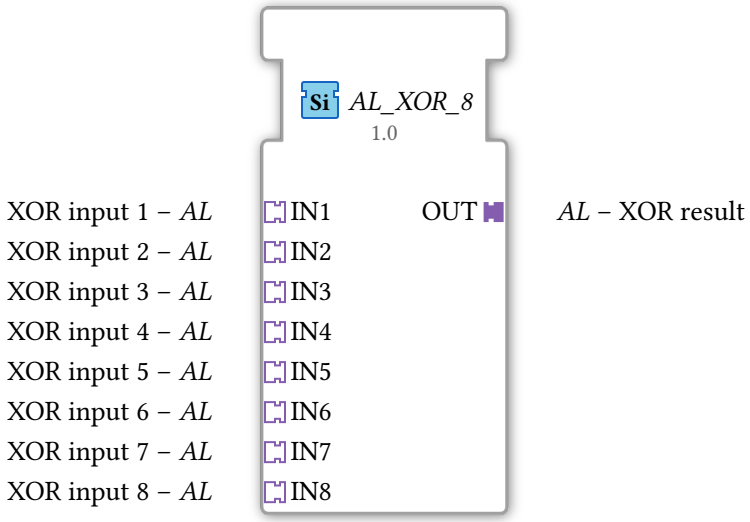
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_XOR_7_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_8



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_8 is a generic function block for the bitwise XOR (exclusive OR) operation across 8 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AL)
- **IN8**: XOR (exclusive OR) input 8 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_8 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_XOR_8`, which combines individual Boolean truth values, `AL_XOR_8` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_8` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AL_XOR_8_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

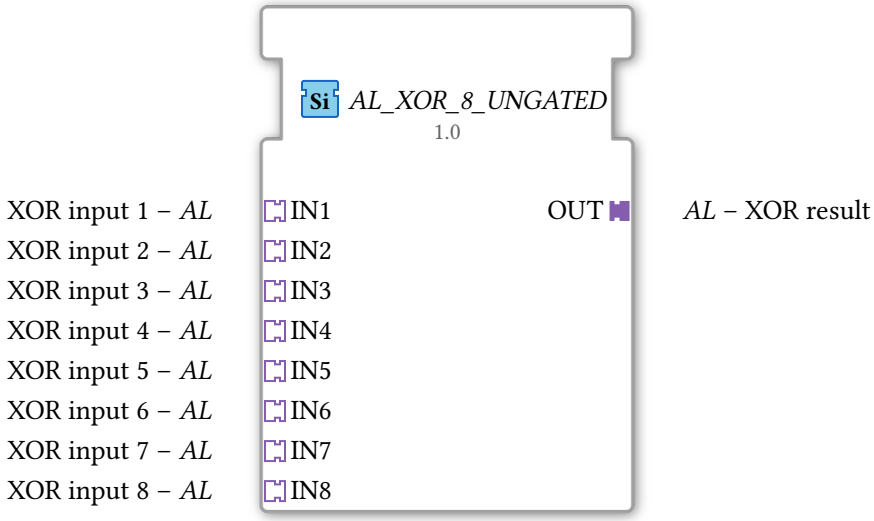
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AL_XOR_8 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AL_XOR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_XOR_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AL_XOR_8_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 8 input values of type `LWORD` (64-bit bit pattern (long word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AL)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AL)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AL)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AL)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AL)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AL)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AL)
- **IN8**: XOR (exclusive OR) input 8 (Type: adapter::types::unidirectional::AL)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AL)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AL_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `LWORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AL` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AL_XOR_8_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `LWORD` into a single result.
- **Signal aggregation**: Merging multiple `LWORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_XOR_8`, which combines individual Boolean truth values, `AL_XOR_8_UNGATED` operates on the full bit pattern of a `LWORD` value. Compared to the standard block XOR_2, `AL_XOR_8_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AL_XOR_8`: The gated variant – updates the output only on an actual value change.

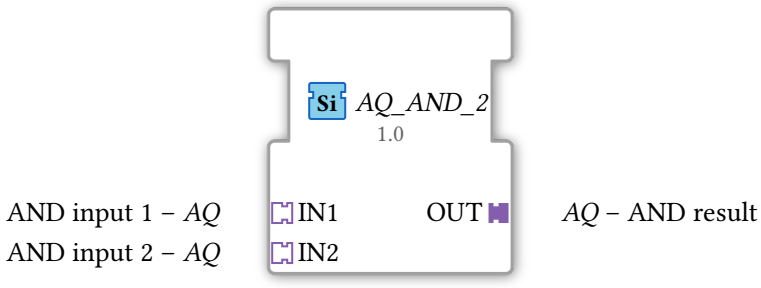
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_XOR_8_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `LWORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_AND_2



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_AND_2 is a generic function block for the bitwise AND operation across 2 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_AND_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_2`, which combines individual Boolean truth values, `AQ_AND_2` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AQ_AND_2` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_AND_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

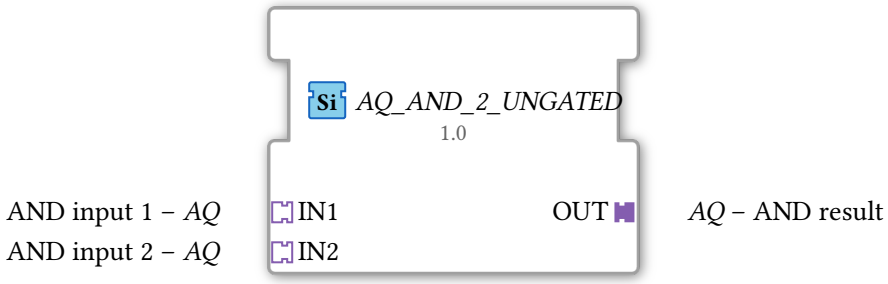
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_AND_2 offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_AND_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_AND_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_AND_2_UNGATED is a generic function block for the bitwise AND operation across 2 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_AND_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_2`, which combines individual Boolean truth values, `AQ_AND_2_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AQ_AND_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_AND_2`**: The gated variant – updates the output only on an actual value change.

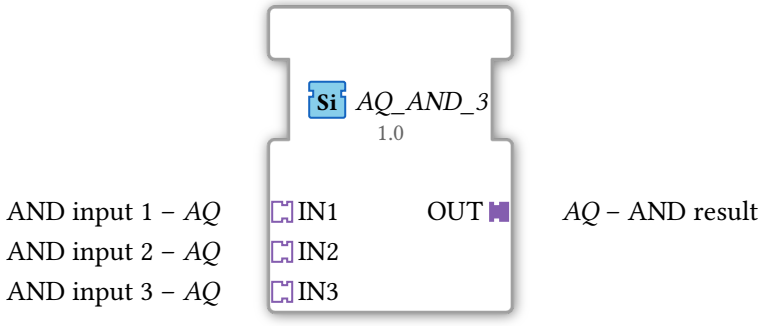
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_AND_2_UNGATED offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_AND_3



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_AND_3 is a generic function block for the bitwise AND operation across 3 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_AND_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_AND_3, which combines individual Boolean truth values, AQ_AND_3 operates on the full bit pattern of a BYTE value. Compared to the standard block AND_2, AQ_AND_3 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AQ_AND_3_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

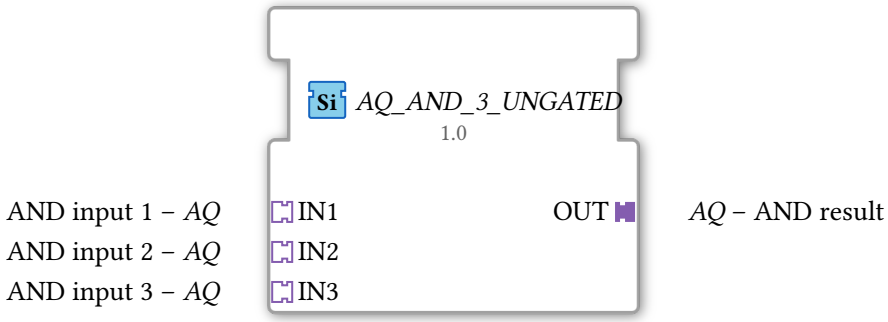
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_AND_3 offers a reliable, generic implementation of the bitwise AND function for BYTE values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_AND_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AQ_AND_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_AND_3_UNGATED is a generic function block for the bitwise AND operation across 3 input values of type BYTE (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the AX_AND blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_AND_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_3`, which combines individual Boolean truth values, `AQ_AND_3_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AQ_AND_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_AND_3`**: The gated variant – updates the output only on an actual value change.

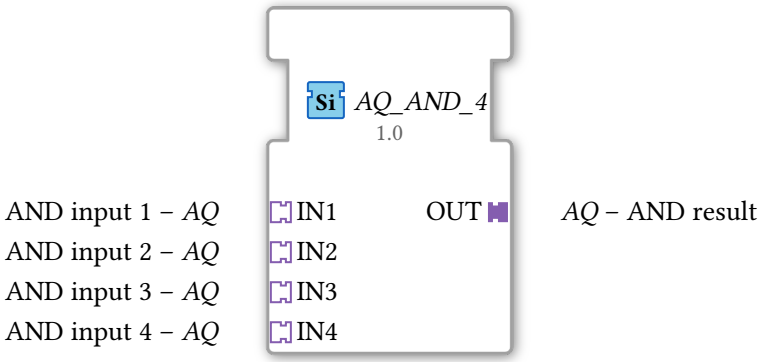
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_AND_3_UNGATED offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_AND_4



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_AND_4 is a generic function block for the bitwise AND operation across 4 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: AND input 4 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.

- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_AND_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_4`, which combines individual Boolean truth values, `AQ_AND_4` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AQ_AND_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_AND_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

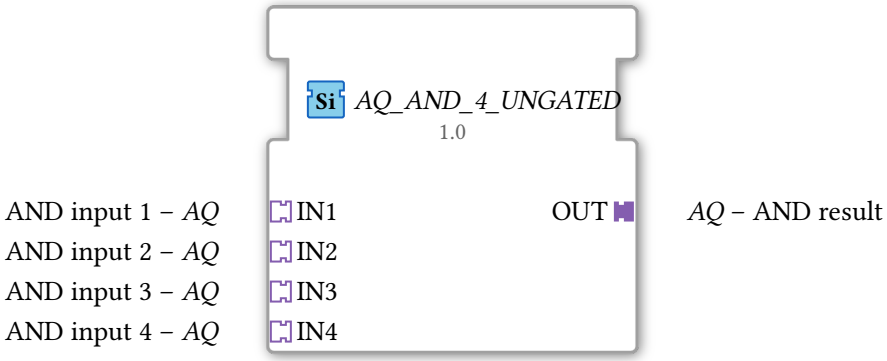
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_AND_4 offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_AND_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_AND_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_AND_4_UNGATED is a generic function block for the bitwise AND operation across 4 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: AND input 4 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_AND_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_4`, which combines individual Boolean truth values, `AQ_AND_4_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block AND_2, `AQ_AND_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_AND_4`**: The gated variant – updates the output only on an actual value change.

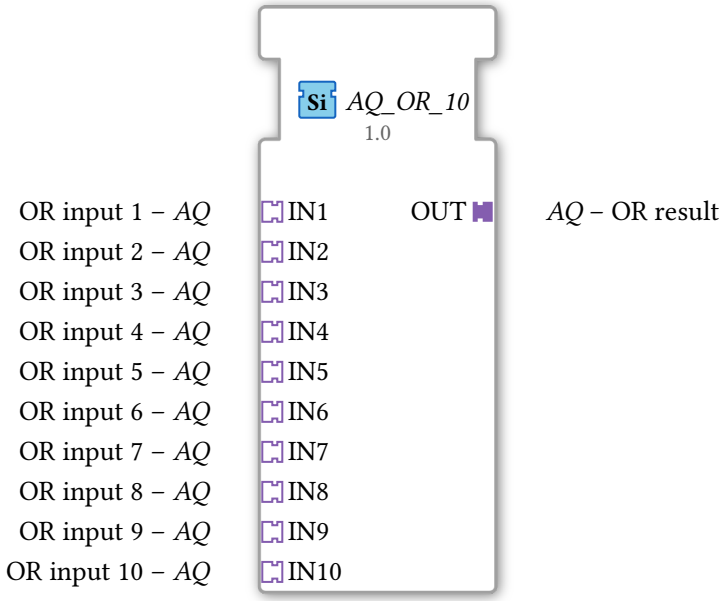
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_AND_4_UNGATED offers a reliable, generic implementation of the bitwise AND function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_10



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_10 is a generic function block for the bitwise OR operation across 10 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AQ)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AQ)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AQ)
- **IN10**: OR input 10 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 10 input adapters (`IN1` … `IN10`), the block combines the bit patterns of all 10 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_10 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_10`, which combines individual Boolean truth values, `AQ_OR_10` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_10` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_OR_10_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

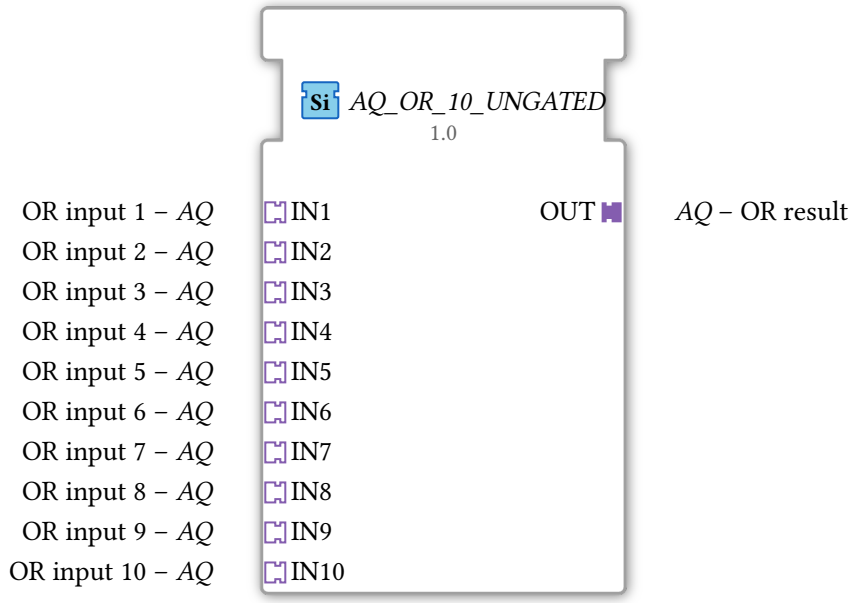
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_OR_10 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_10_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_OR_10`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_10_UNGATED is a generic function block for the bitwise OR operation across 10 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)

- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AQ)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AQ)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AQ)
- **IN10**: OR input 10 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 10 input adapters (`IN1` … `IN10`), the block combines the bit patterns of all 10 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_10_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_10`, which combines individual Boolean truth values, `AQ_OR_10_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_10_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_OR_10`**: The gated variant – updates the output only on an actual value change.

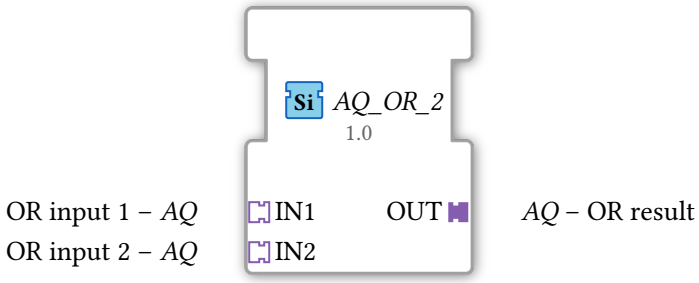
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_OR_10_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_2



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_2 is a generic function block for the bitwise OR operation across 2 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_2`, which combines individual Boolean truth values, `AQ_OR_2` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_2` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_OR_2_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

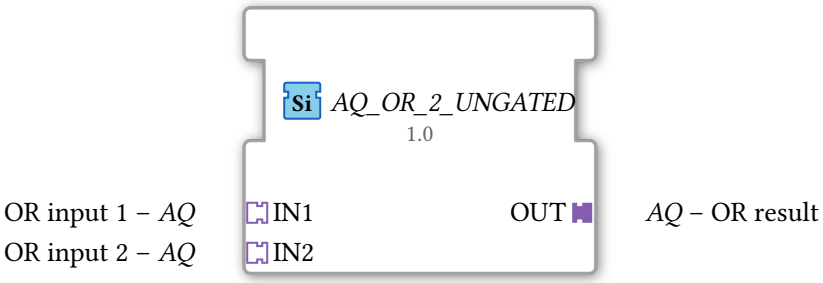
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_OR_2 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_OR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_2_UNGATED is a generic function block for the bitwise OR operation across 2 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_2`, which combines individual Boolean truth values, `AQ_OR_2_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_OR_2`**: The gated variant – updates the output only on an actual value change.

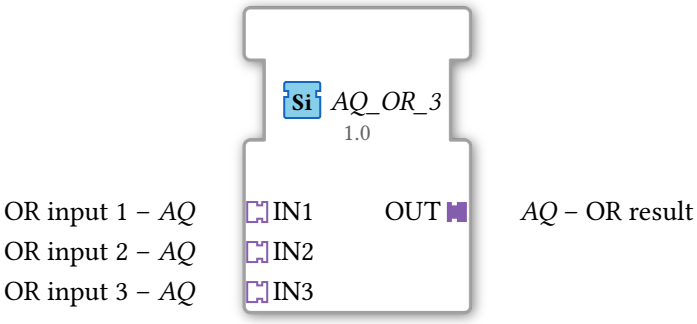
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_OR_2_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_3



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_3 is a generic function block for the bitwise OR operation across 3 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_3`, which combines individual Boolean truth values, `AQ_OR_3` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_OR_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

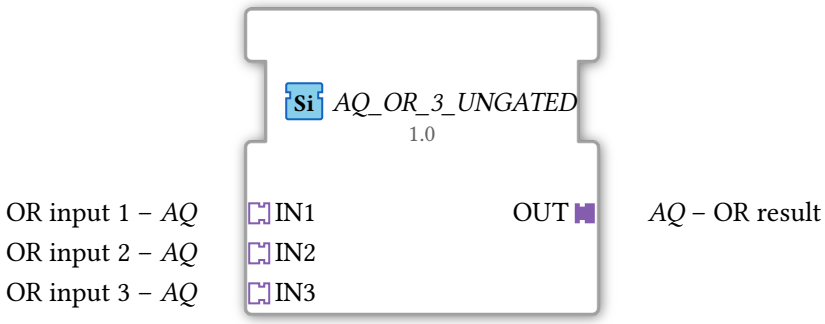
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_OR_3 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_OR_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_3_UNGATED is a generic function block for the bitwise OR operation across 3 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_OR_3, which combines individual Boolean truth values, AQ_OR_3_UNGATED operates on the full bit pattern of a BYTE value. Compared to the standard block OR_2, AQ_OR_3_UNGATED uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AQ_OR_3**: The gated variant – updates the output only on an actual value change.

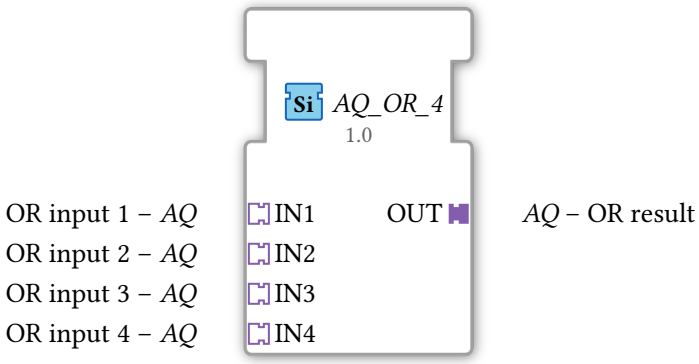
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_OR_3_UNGATED offers a reliable, generic implementation of the bitwise OR function for BYTE values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_4



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_4 is a generic function block for the bitwise OR operation across 4 input values of type BYTE (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the AX_OR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_4`, which combines individual Boolean truth values, `AQ_OR_4` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_OR_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

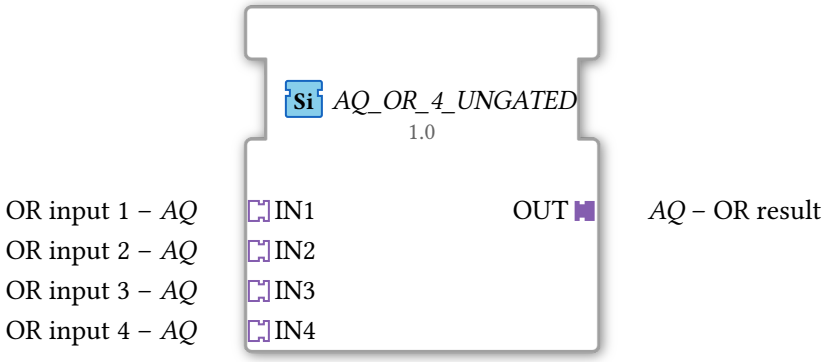
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_OR_4 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AQ_OR_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_4_UNGATED is a generic function block for the bitwise OR operation across 4 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.

- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_4`, which combines individual Boolean truth values, `AQ_OR_4_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_OR_4`: The gated variant – updates the output only on an actual value change.

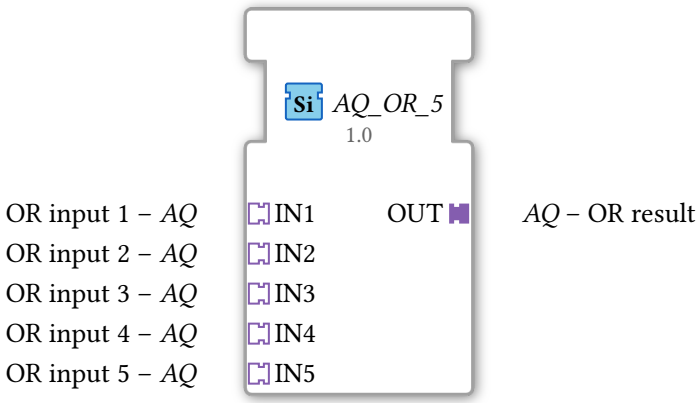
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_OR_4_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_5



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_5 is a generic function block for the bitwise OR operation across 5 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_5 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_5`, which combines individual Boolean truth values, `AQ_OR_5` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_5` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_OR_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

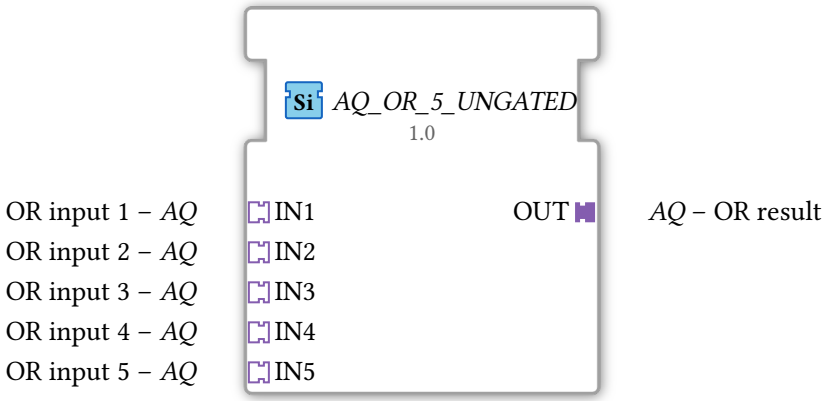
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_OR_5 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_OR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_5_UNGATED is a generic function block for the bitwise OR operation across 5 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)

- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_5_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_5`, which combines individual Boolean truth values, `AQ_OR_5_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_5_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_OR_5`: The gated variant – updates the output only on an actual value change.

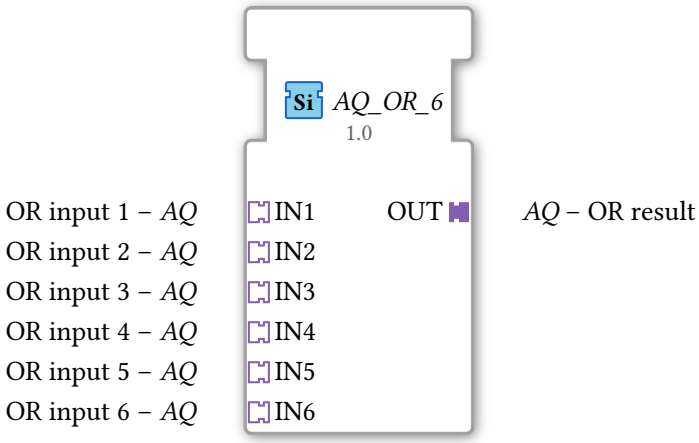
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_OR_5_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_6



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_6 is a generic function block for the bitwise OR operation across 6 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_6 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_6`, which combines individual Boolean truth values, `AQ_OR_6` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_6` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_OR_6_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

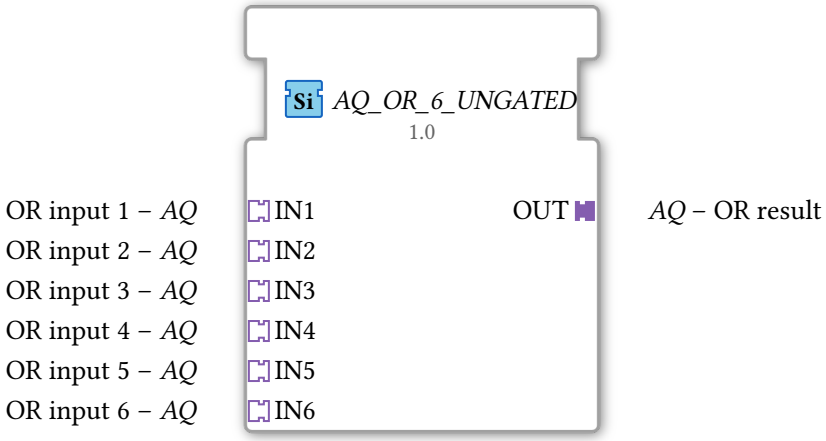
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_OR_6 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of AQ_OR_6. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_6_UNGATED is a generic function block for the bitwise OR operation across 6 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_6_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_6`, which combines individual Boolean truth values, `AQ_OR_6_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_6_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_OR_6`: The gated variant – updates the output only on an actual value change.

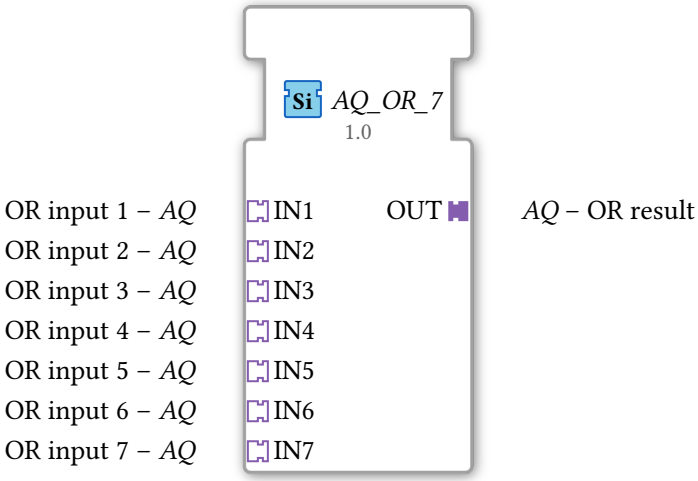
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_OR_6_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_7



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_7 is a generic function block for the bitwise OR operation across 7 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_7 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_OR_7, which combines individual Boolean truth values, AQ_OR_7 operates on the full bit pattern of a BYTE value. Compared to the standard block OR_2, AQ_OR_7 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AQ_OR_7_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

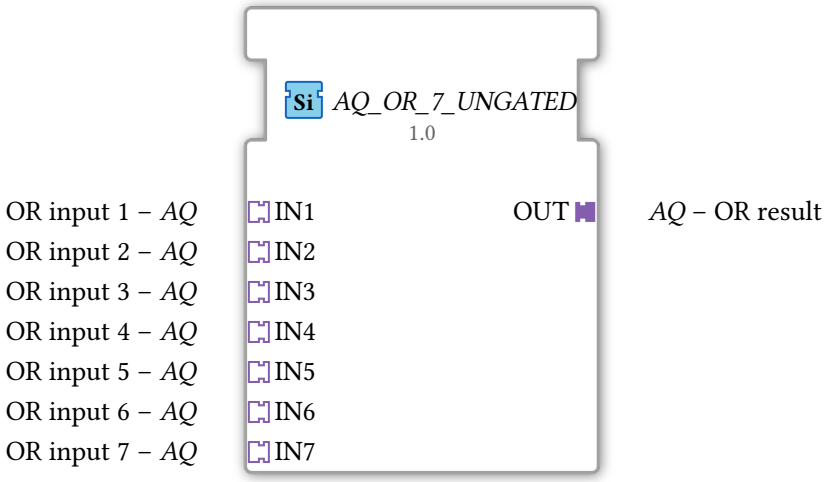
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_OR_7 offers a reliable, generic implementation of the bitwise OR function for BYTE values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of AQ_OR_7. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_7_UNGATED is a generic function block for the bitwise OR operation across 7 input values of type BYTE (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the AX_OR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_7_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_7`, which combines individual Boolean truth values, `AQ_OR_7_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_7_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_OR_7`: The gated variant – updates the output only on an actual value change.

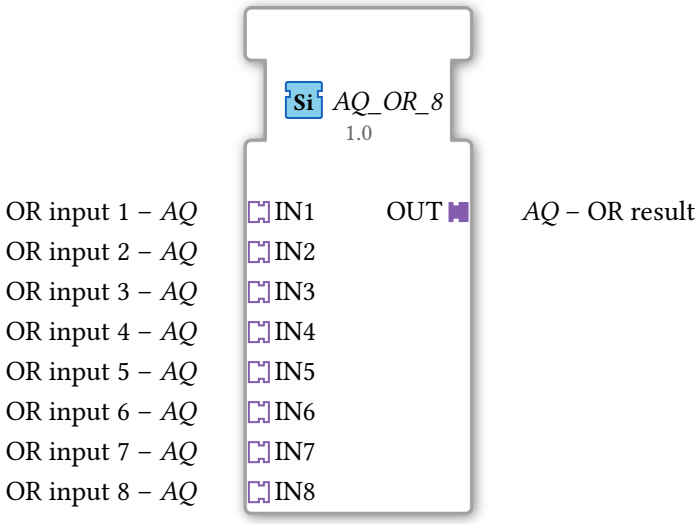
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_OR_7_UNGATED offers a reliable, generic implementation of the bitwise OR function for BYTE values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_8



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_8 is a generic function block for the bitwise OR operation across 8 input values of type BYTE (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the AX_OR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AQ)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_8 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_8`, which combines individual Boolean truth values, `AQ_OR_8` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_8` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_OR_8_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

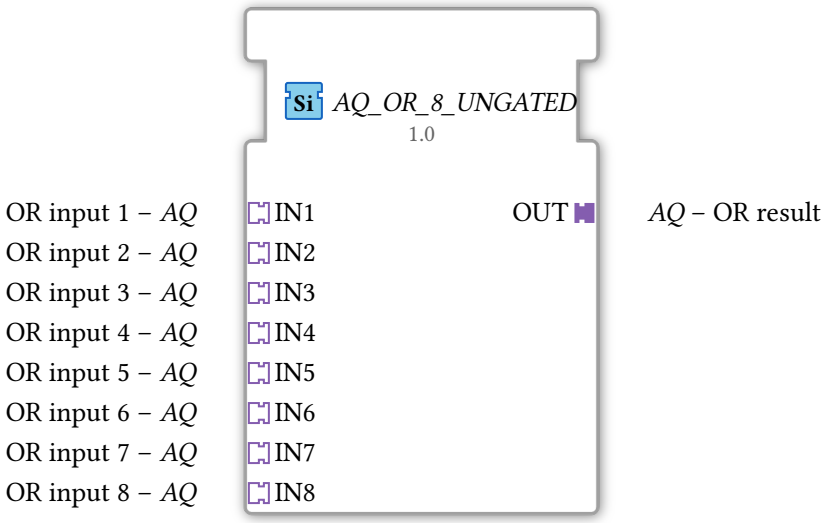
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_OR_8 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_OR_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_8_UNGATED is a generic function block for the bitwise OR operation across 8 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AQ)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_8_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_8`, which combines individual Boolean truth values, `AQ_OR_8_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_8_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_OR_8`**: The gated variant – updates the output only on an actual value change.

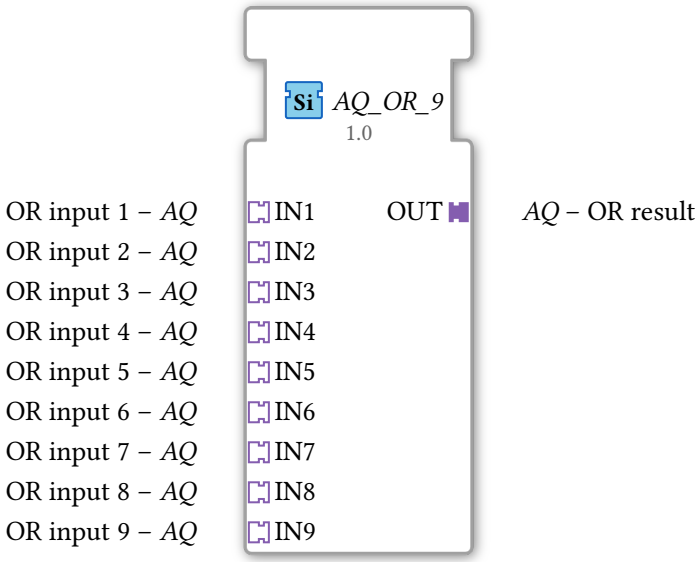
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_OR_8_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_9



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_9 is a generic function block for the bitwise OR operation across 9 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AQ)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AQ)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 9 input adapters (`IN1` … `IN9`), the block combines the bit patterns of all 9 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_9 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_9`, which combines individual Boolean truth values, `AQ_OR_9` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_9` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_OR_9_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

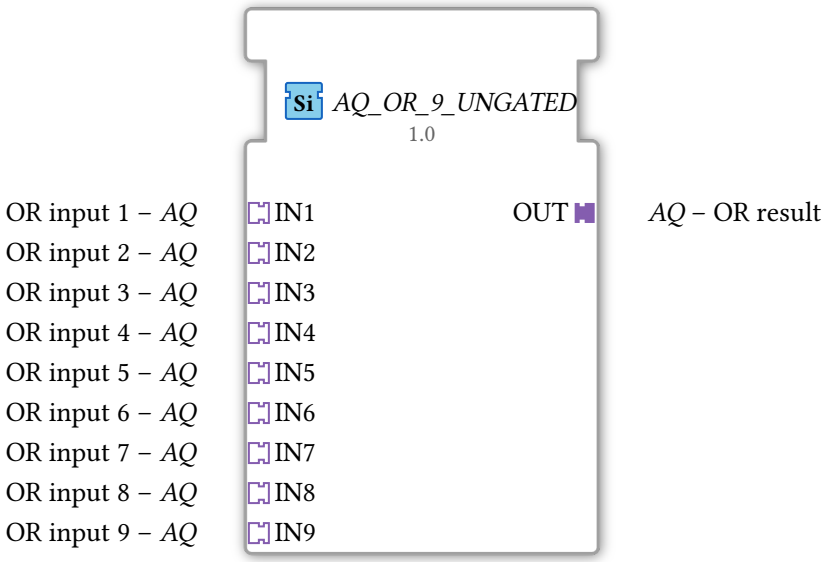
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_OR_9 offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_OR_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_OR_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_OR_9_UNGATED is a generic function block for the bitwise OR operation across 9 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AQ)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AQ)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 9 input adapters (`IN1` … `IN9`), the block combines the bit patterns of all 9 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the

operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_OR_9_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_9`, which combines individual Boolean truth values, `AQ_OR_9_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block OR_2, `AQ_OR_9_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_OR_9`: The gated variant – updates the output only on an actual value change.

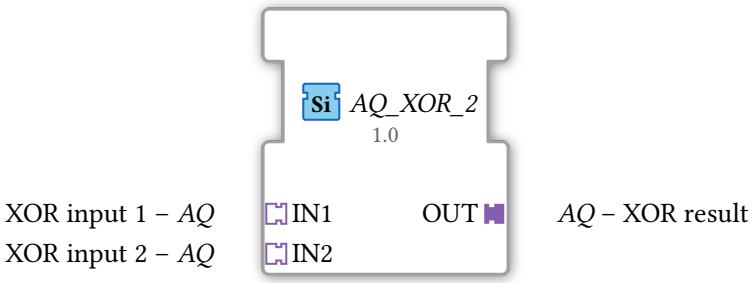
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_OR_9_UNGATED offers a reliable, generic implementation of the bitwise OR function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_2



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_2 is a generic function block for the bitwise XOR (exclusive OR) operation across 2 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_2`, which combines individual Boolean truth values, `AQ_XOR_2` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_2` uses adapter-based interfaces

instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_XOR_2_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

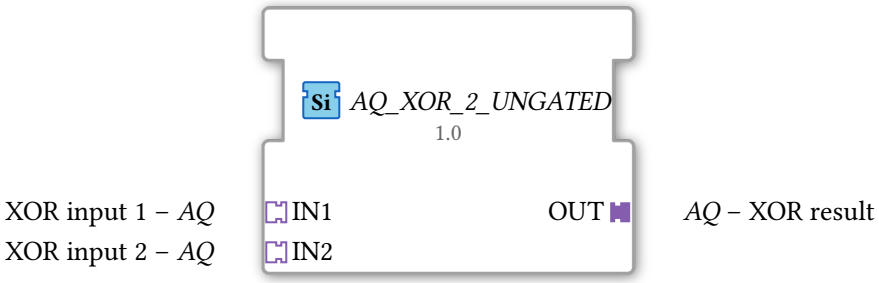
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_XOR_2 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_XOR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_2_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 2 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)

- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_2`, which combines individual Boolean truth values, `AQ_XOR_2_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_XOR_2`**: The gated variant – updates the output only on an actual value change.

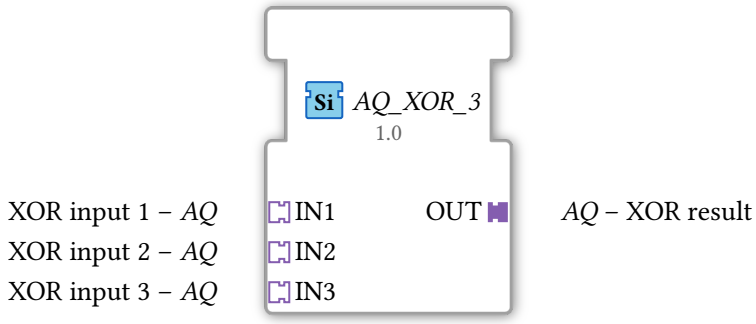
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_XOR_2_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_3



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_3 is a generic function block for the bitwise XOR (exclusive OR) operation across 3 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_3`, which combines individual Boolean truth values, `AQ_XOR_3` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_XOR_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

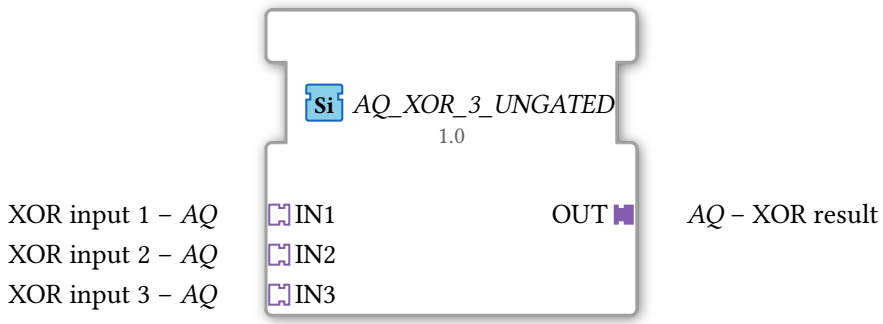
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_XOR_3 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_XOR_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_3_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 3 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality
As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features
- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview
Since this is a combinational logic block, AQ_XOR_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios
- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks
Unlike `AX_XOR_3`, which combines individual Boolean truth values, `AQ_XOR_3_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_XOR_3`**: The gated variant – updates the output only on an actual value change.

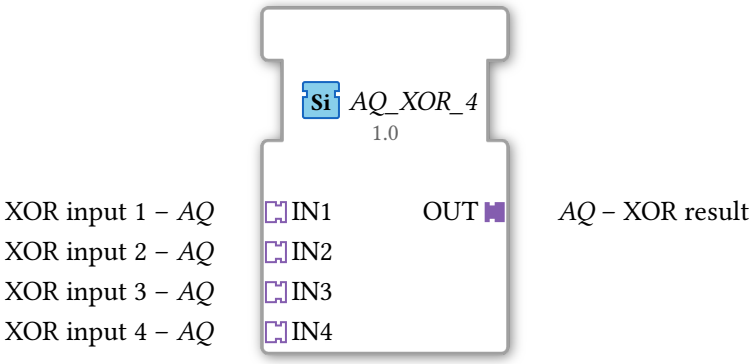
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_XOR_3_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_4



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_4 is a generic function block for the bitwise XOR (exclusive OR) operation across 4 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The

starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on OUT is OUT rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (GEN_AQ_XOR) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean AX_XOR blocks, every bit of the BYTE data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type unidirectional::AQ – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type BYTE into a single result.
- **Signal aggregation**: Merging multiple BYTE data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_XOR_4, which combines individual Boolean truth values, AQ_XOR_4 operates on the full bit pattern of a BYTE value. Compared to the standard block XOR_2, AQ_XOR_4 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AQ_XOR_4_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

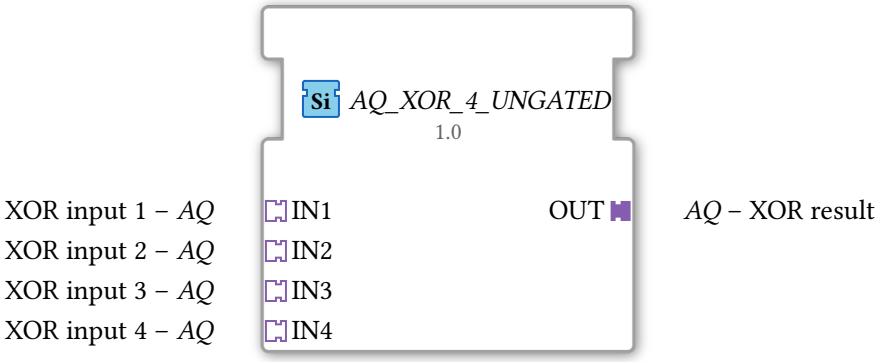
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_XOR_4 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for BYTE values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AQ_XOR_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_4_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 4 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.

- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_4`, which combines individual Boolean truth values, `AQ_XOR_4_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_XOR_4`**: The gated variant – updates the output only on an actual value change.

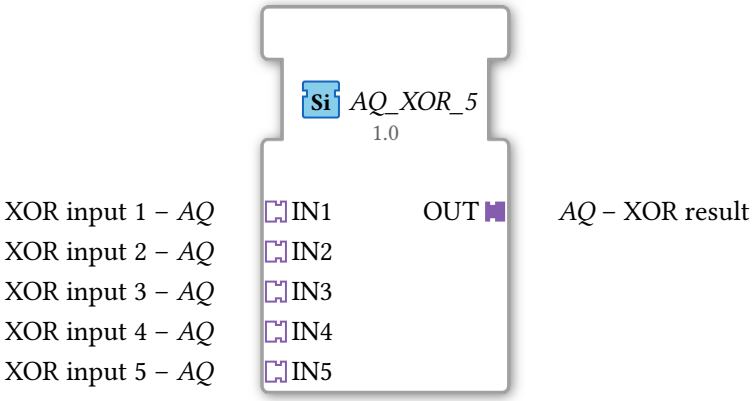
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_XOR_4_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_5



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_5 is a generic function block for the bitwise XOR (exclusive OR) operation across 5 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_5 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_5`, which combines individual Boolean truth values, `AQ_XOR_5` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_5` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_XOR_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

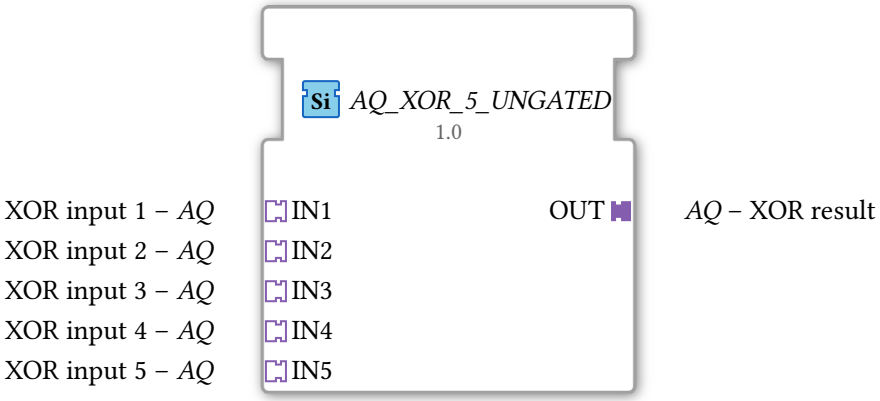
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_XOR_5 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_XOR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_5_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 5 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)

- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_5_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_5`, which combines individual Boolean truth values, `AQ_XOR_5_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_5_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_XOR_5`**: The gated variant – updates the output only on an actual value change.

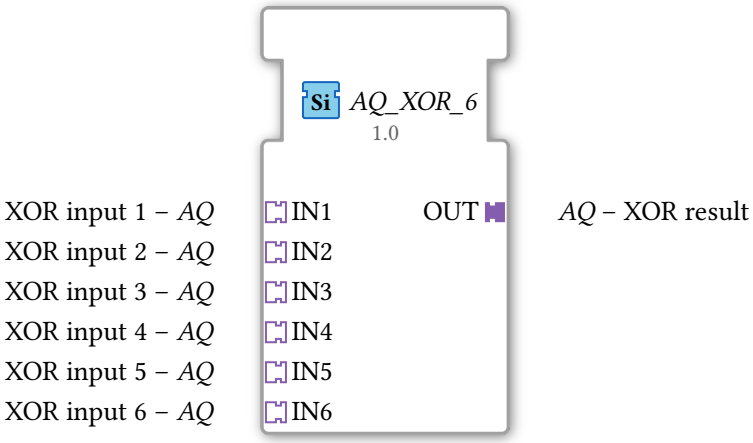
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_XOR_5_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_6



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_6 is a generic function block for the bitwise XOR (exclusive OR) operation across 6 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_6 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_6`, which combines individual Boolean truth values, `AQ_XOR_6` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_6` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_XOR_6_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

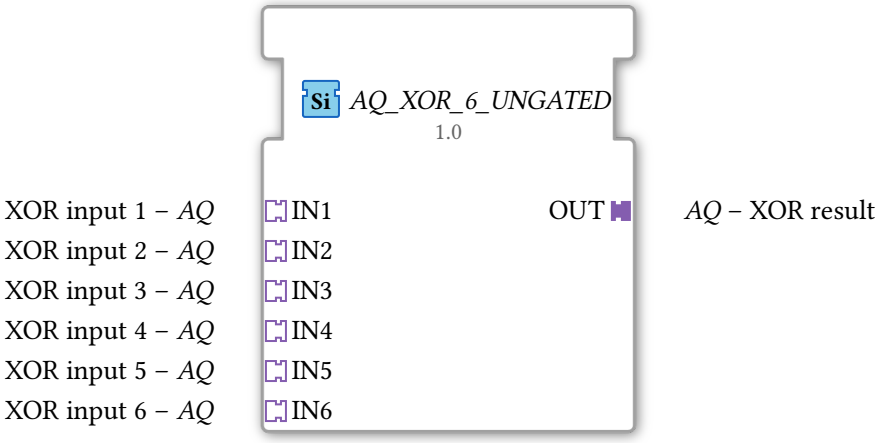
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_XOR_6 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_XOR_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_6_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 6 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_6_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_XOR_6`, which combines individual Boolean truth values, `AQ_XOR_6_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_6_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_XOR_6`: The gated variant – updates the output only on an actual value change.

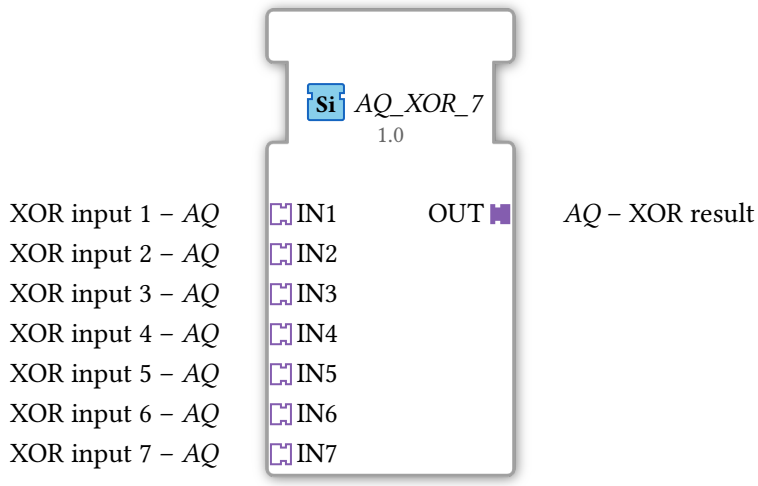
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_XOR_6_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_7



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_7 is a generic function block for the bitwise XOR (exclusive OR) operation across 7 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_7 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_XOR_7, which combines individual Boolean truth values, AQ_XOR_7 operates on the full bit pattern of a BYTE value. Compared to the standard block XOR_2, AQ_XOR_7 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AQ_XOR_7_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

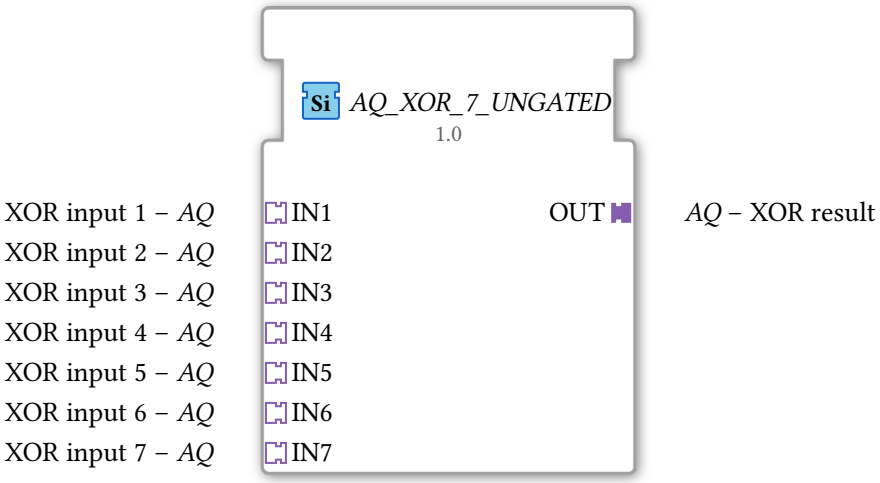
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_XOR_7 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for BYTE values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of AQ_XOR_7. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_7_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 7 input values of type BYTE (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the AX_XOR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_7_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_7`, which combines individual Boolean truth values, `AQ_XOR_7_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_7_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_XOR_7`**: The gated variant – updates the output only on an actual value change.

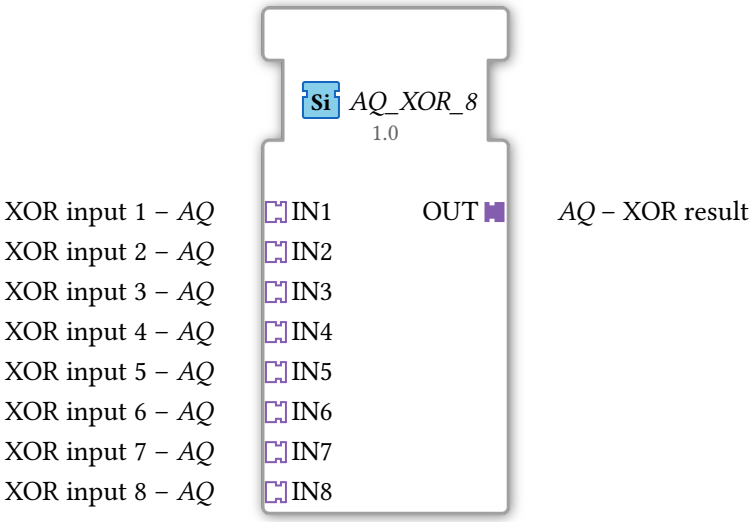
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_XOR_7_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_8



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_8 is a generic function block for the bitwise XOR (exclusive OR) operation across 8 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AQ)
- **IN8**: XOR (exclusive OR) input 8 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_8 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_8`, which combines individual Boolean truth values, `AQ_XOR_8` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_8` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AQ_XOR_8_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

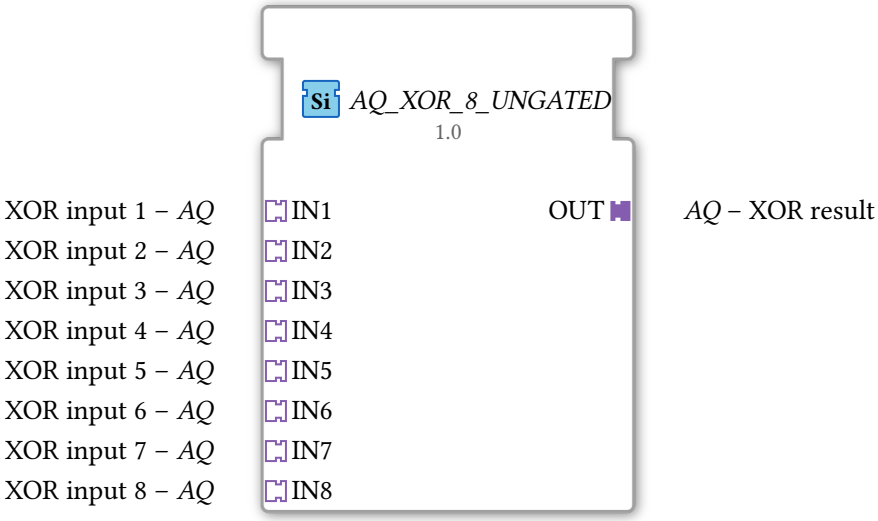
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AQ_XOR_8 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AQ_XOR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_XOR_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_XOR_8_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 8 input values of type `BYTE` (2-bit value (quarter byte, transmitted as a byte)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AQ)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AQ)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AQ)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AQ)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AQ)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AQ)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AQ)
- **IN8**: XOR (exclusive OR) input 8 (Type: adapter::types::unidirectional::AQ)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AQ)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AQ_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `BYTE` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AQ` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AQ_XOR_8_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `BYTE` into a single result.
- **Signal aggregation**: Merging multiple `BYTE` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_8`, which combines individual Boolean truth values, `AQ_XOR_8_UNGATED` operates on the full bit pattern of a `BYTE` value. Compared to the standard block XOR_2, `AQ_XOR_8_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AQ_XOR_8`: The gated variant – updates the output only on an actual value change.

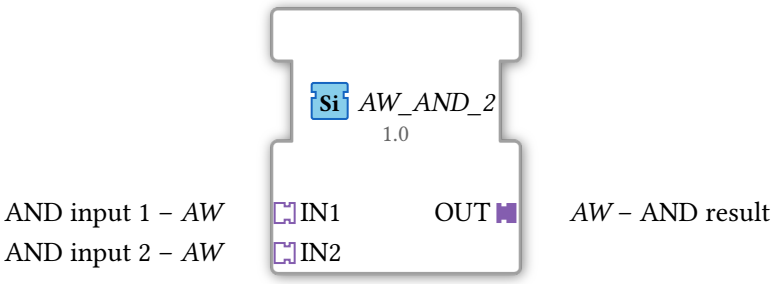
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AQ_XOR_8_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `BYTE` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_AND_2



• – ◦ • – ◦ • – ◦ •

Introduction

AW_AND_2 is a generic function block for the bitwise AND operation across 2 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AW)

Functionality
As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features
- **Generic function block**: The FB is defined as a generic type (`GEN_AW_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview
Since this is a combinational logic block, AW_AND_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios
- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks
Unlike `AX_AND_2`, which combines individual Boolean truth values, `AW_AND_2` operates on the full bit pattern of a `WORD` value. Compared to the standard block AND_2, `AW_AND_2` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_AND_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

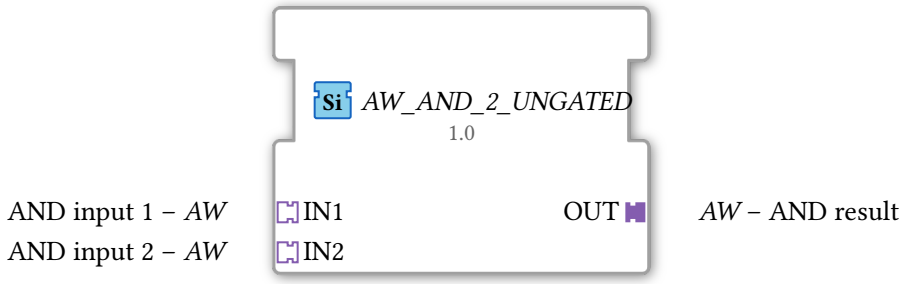
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_AND_2 offers a reliable, generic implementation of the bitwise AND function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_AND_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_AND_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_AND_2_UNGATED is a generic function block for the bitwise AND operation across 2 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_AND_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_2`, which combines individual Boolean truth values, `AW_AND_2_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block AND_2, `AW_AND_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_AND_2`**: The gated variant – updates the output only on an actual value change.

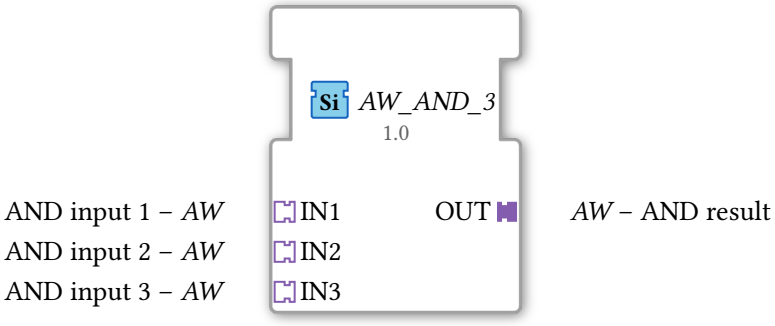
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_AND_2_UNGATED offers a reliable, generic implementation of the bitwise AND function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_AND_3



• – ◦ • – ◦ • – ◦ •

Introduction

AW_AND_3 is a generic function block for the bitwise AND operation across 3 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_AND_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_3`, which combines individual Boolean truth values, `AW_AND_3` operates on the full bit pattern of a `WORD` value. Compared to the standard block AND_2, `AW_AND_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_AND_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

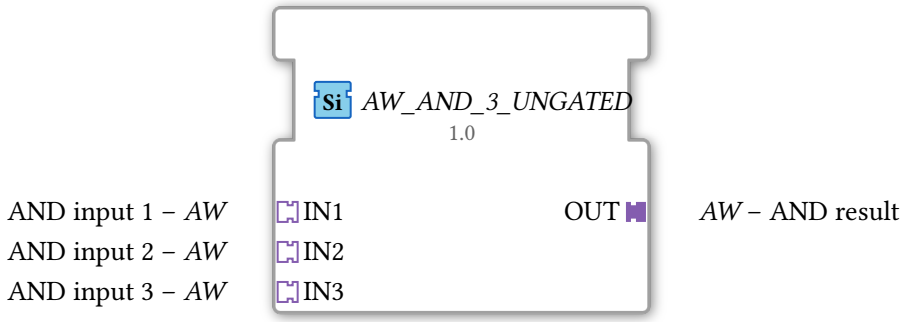
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_AND_3 offers a reliable, generic implementation of the bitwise AND function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_AND_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_AND_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_AND_3_UNGATED is a generic function block for the bitwise AND operation across 3 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_AND_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_3`, which combines individual Boolean truth values, `AW_AND_3_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block AND_2, `AW_AND_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_AND_3`**: The gated variant – updates the output only on an actual value change.

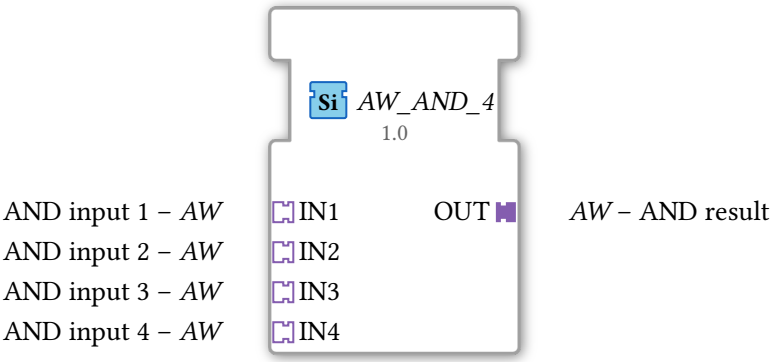
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_AND_3_UNGATED offers a reliable, generic implementation of the bitwise AND function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_AND_4



• – ◦ • – ◦ • – ◦ •

Introduction

AW_AND_4 is a generic function block for the bitwise AND operation across 4 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: AND input 4 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.

- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_AND_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_4`, which combines individual Boolean truth values, `AW_AND_4` operates on the full bit pattern of a `WORD` value. Compared to the standard block AND_2, `AW_AND_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_AND_4_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

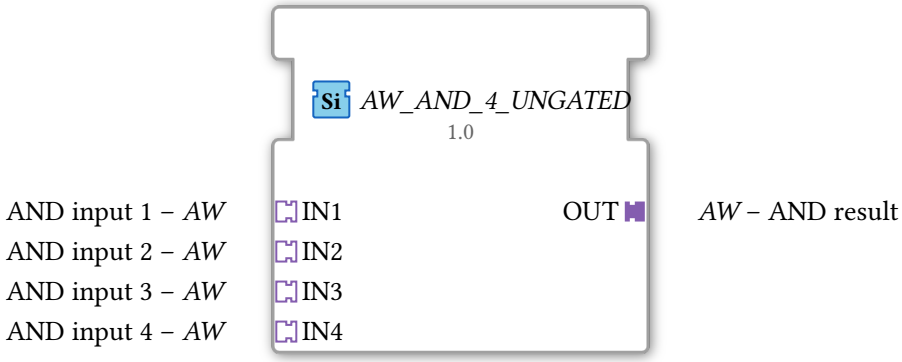
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_AND_4 offers a reliable, generic implementation of the bitwise AND function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_AND_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_AND_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_AND_4_UNGATED is a generic function block for the bitwise AND operation across 4 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_AND` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: AND input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: AND input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: AND input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: AND input 4 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: AND result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **AND** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits set (identity element of the AND operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_AND`) and covers all arities (2 to 4 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_AND` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_AND_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared AND operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_AND_4`, which combines individual Boolean truth values, `AW_AND_4_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block AND_2, `AW_AND_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_AND_4`**: The gated variant – updates the output only on an actual value change.

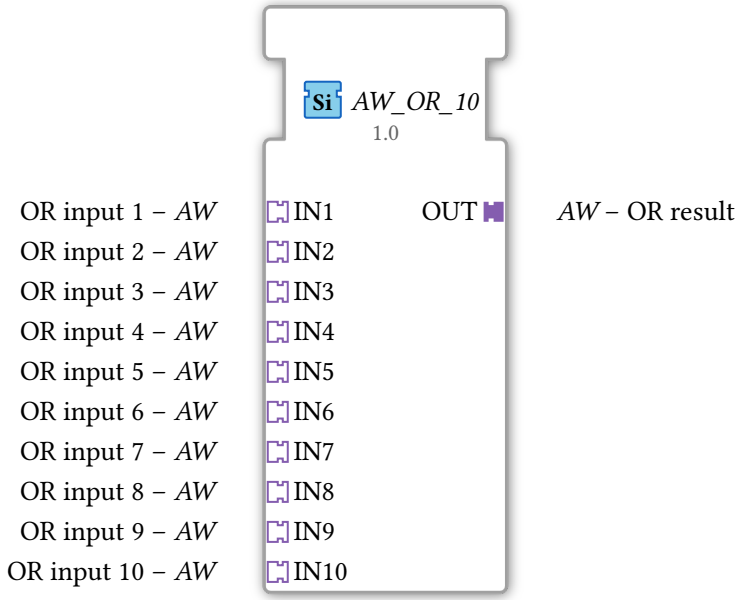
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_AND_4_UNGATED offers a reliable, generic implementation of the bitwise AND function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_10



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_10 is a generic function block for the bitwise OR operation across 10 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AW)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AW)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AW)
- **IN10**: OR input 10 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 10 input adapters (`IN1` … `IN10`), the block combines the bit patterns of all 10 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_10 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_10`, which combines individual Boolean truth values, `AW_OR_10` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_10` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_10_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

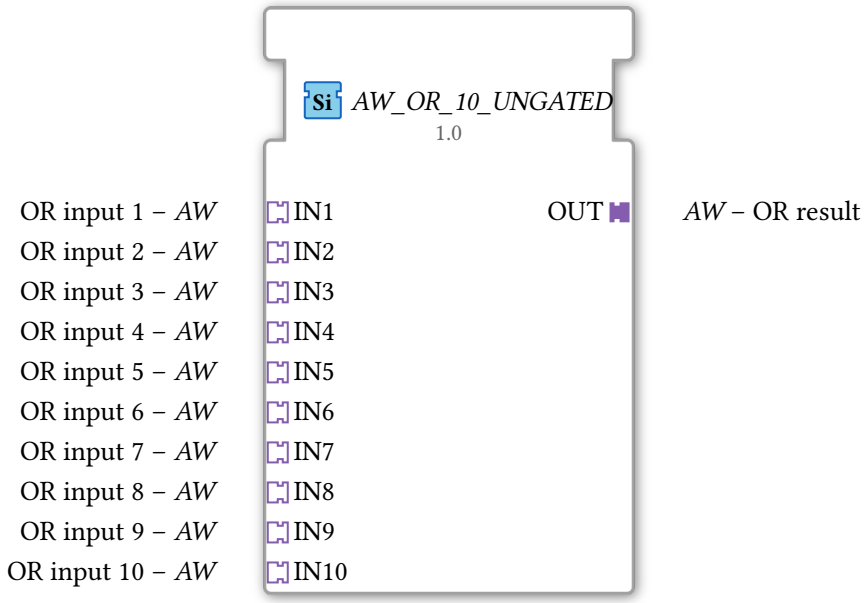
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_OR_10 offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_10_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_OR_10`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_10_UNGATED is a generic function block for the bitwise OR operation across 10 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)

- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AW)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AW)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AW)
- **IN10**: OR input 10 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 10 input adapters (`IN1` … `IN10`), the block combines the bit patterns of all 10 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_10_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_10`, which combines individual Boolean truth values, `AW_OR_10_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_10_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_10`**: The gated variant – updates the output only on an actual value change.

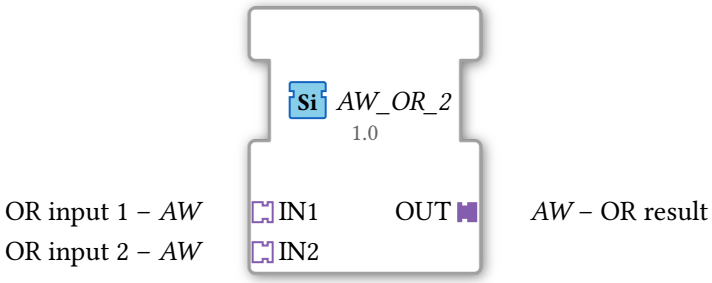
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_OR_10_UNGATED offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_2



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_2 is a generic function block for the bitwise OR operation across 2 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_2`, which combines individual Boolean truth values, `AW_OR_2` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_2` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_OR_2_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

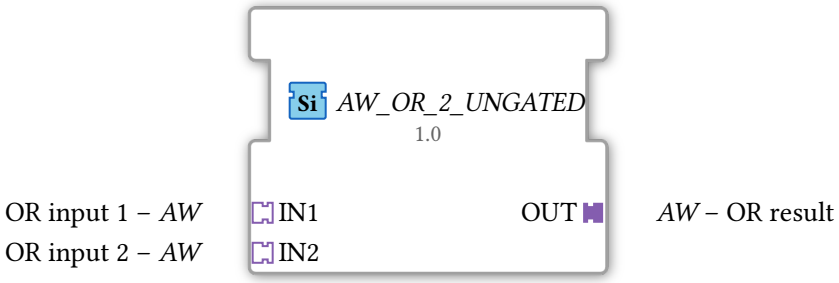
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_OR_2 offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_OR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_2_UNGATED is a generic function block for the bitwise OR operation across 2 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_2`, which combines individual Boolean truth values, `AW_OR_2_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_OR_2`: The gated variant – updates the output only on an actual value change.

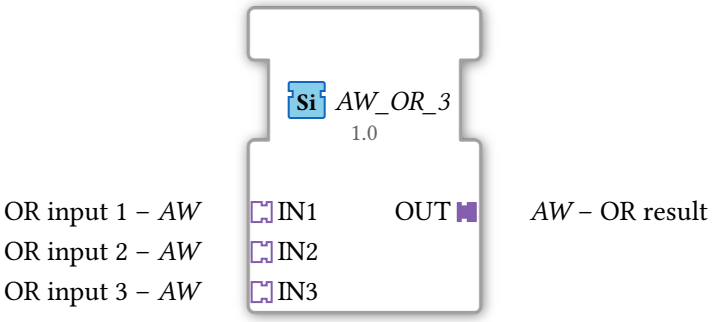
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_OR_2_UNGATED offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_3



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_3 is a generic function block for the bitwise OR operation across 3 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_3`, which combines individual Boolean truth values, `AW_OR_3` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

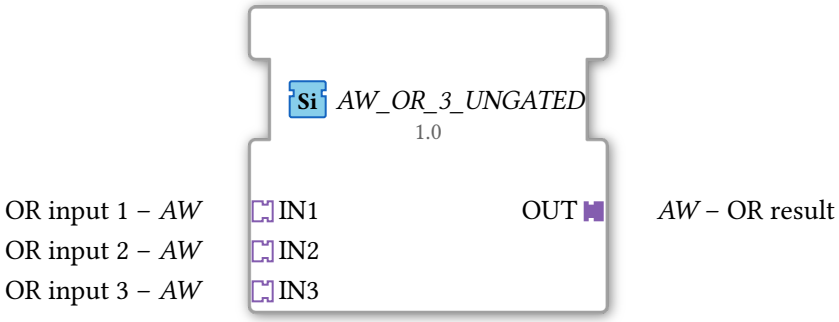
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_OR_3 offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AW_OR_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_3_UNGATED is a generic function block for the bitwise OR operation across 3 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_3`, which combines individual Boolean truth values, `AW_OR_3_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_3`**: The gated variant – updates the output only on an actual value change.

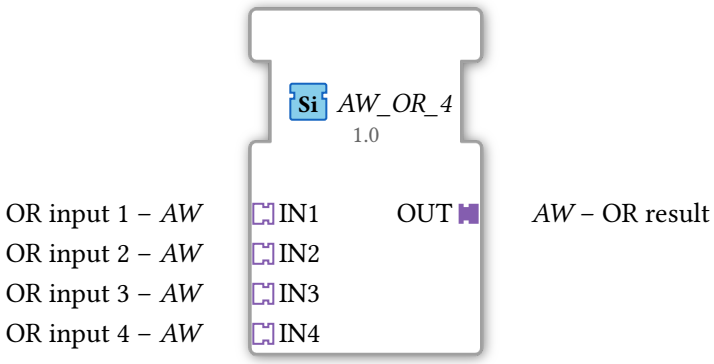
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_OR_3_UNGATED offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_4



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_4 is a generic function block for the bitwise OR operation across 4 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ **Comparison with Similar Blocks**

Unlike `AX_OR_4`, which combines individual Boolean truth values, `AW_OR_4` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

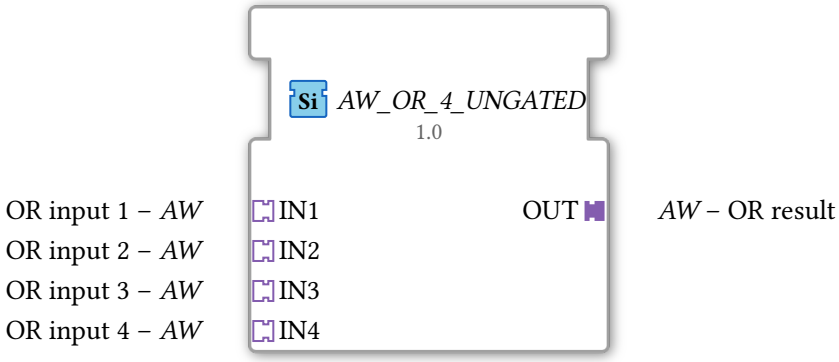
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_OR_4 offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_OR_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_4_UNGATED is a generic function block for the bitwise OR operation across 4 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.

- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_4`, which combines individual Boolean truth values, `AW_OR_4_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_4`**: The gated variant – updates the output only on an actual value change.

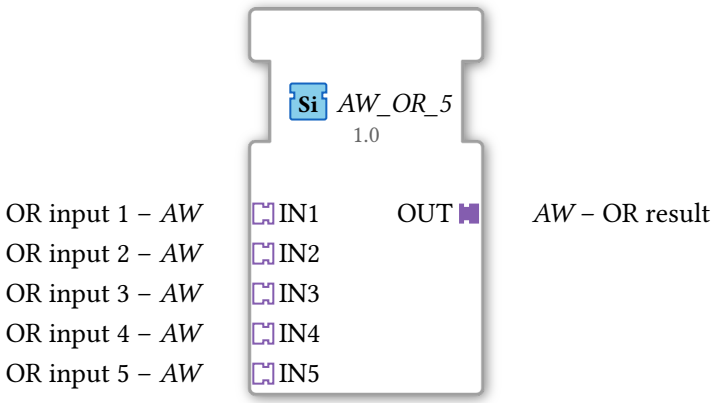
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_OR_4_UNGATED offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_5



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_5 is a generic function block for the bitwise OR operation across 5 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_5 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_5`, which combines individual Boolean truth values, `AW_OR_5` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_5` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

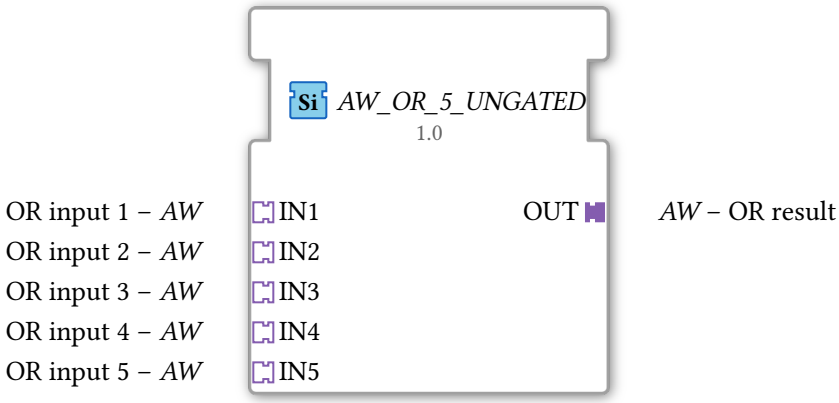
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_OR_5 offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_OR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_5_UNGATED is a generic function block for the bitwise OR operation across 5 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)

- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_5_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_5`, which combines individual Boolean truth values, `AW_OR_5_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_5_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_OR_5`: The gated variant – updates the output only on an actual value change.

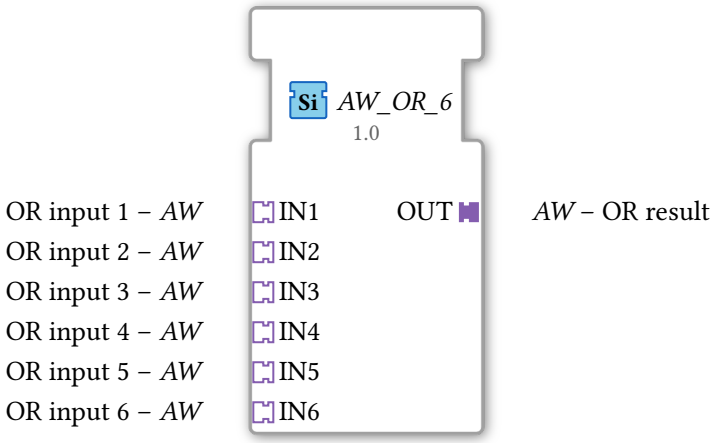
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_OR_5_UNGATED offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_6



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_6 is a generic function block for the bitwise OR operation across 6 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_6 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_6`, which combines individual Boolean truth values, `AW_OR_6` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_6` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_6_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

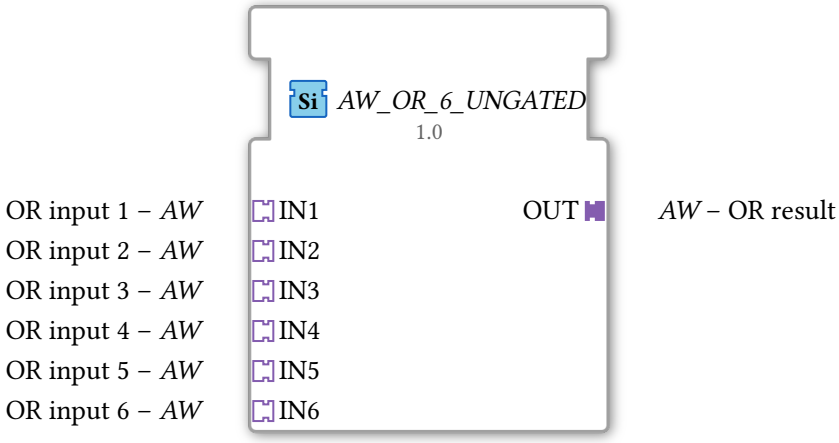
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_OR_6 offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_OR_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_6_UNGATED is a generic function block for the bitwise OR operation across 6 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_6_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_6`, which combines individual Boolean truth values, `AW_OR_6_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_6_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_OR_6`: The gated variant – updates the output only on an actual value change.

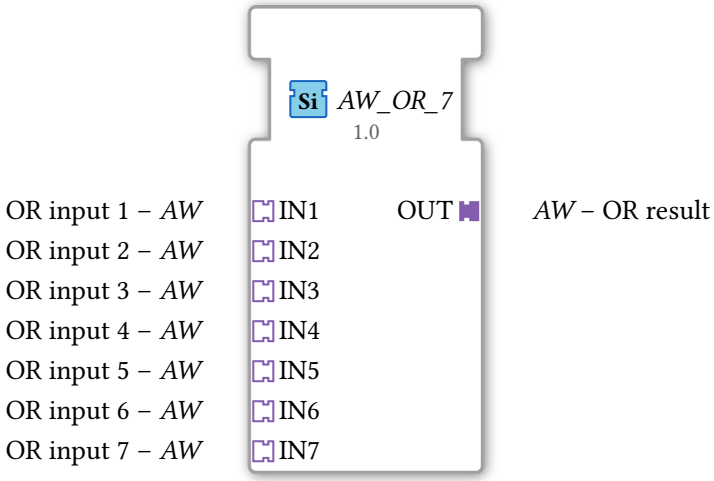
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_OR_6_UNGATED offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_7



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_7 is a generic function block for the bitwise OR operation across 7 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_7 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_OR_7, which combines individual Boolean truth values, AW_OR_7 operates on the full bit pattern of a WORD value. Compared to the standard block OR_2, AW_OR_7 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AW_OR_7_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

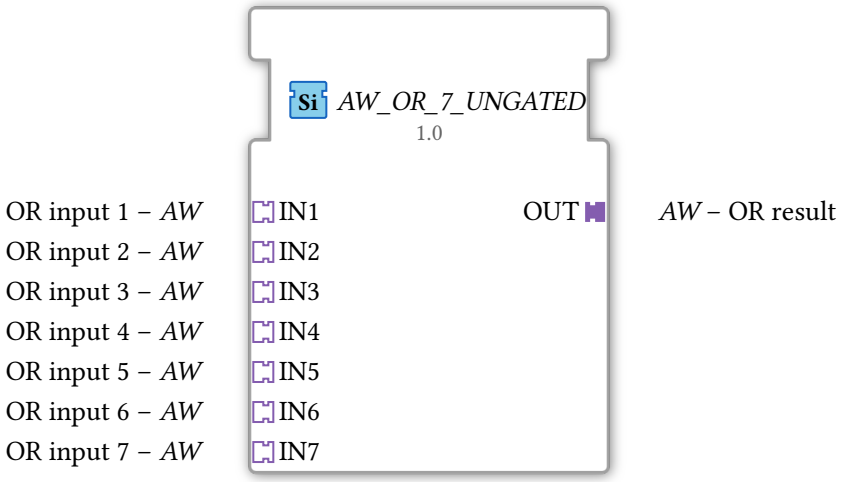
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_OR_7 offers a reliable, generic implementation of the bitwise OR function for WORD values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of AW_OR_7. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_7_UNGATED is a generic function block for the bitwise OR operation across 7 input values of type WORD (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the AX_OR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_7_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_7`, which combines individual Boolean truth values, `AW_OR_7_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_7_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_OR_7`: The gated variant – updates the output only on an actual value change.

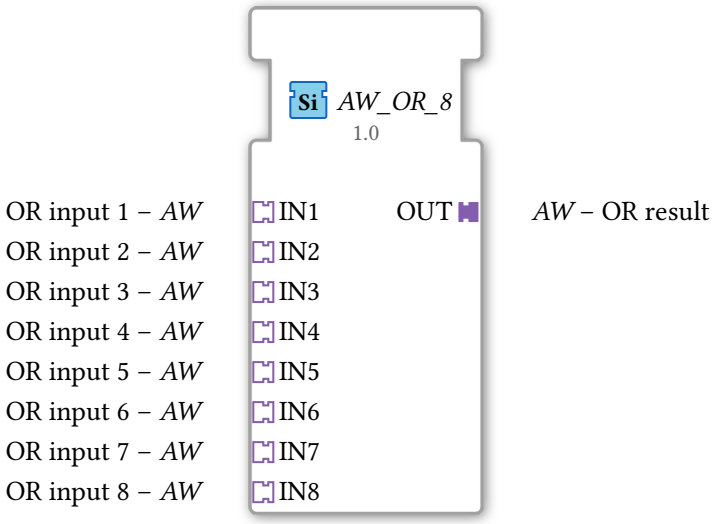
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_OR_7_UNGATED offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_8



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_8 is a generic function block for the bitwise OR operation across 8 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AW)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_8 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_8`, which combines individual Boolean truth values, `AW_OR_8` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_8` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_8_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

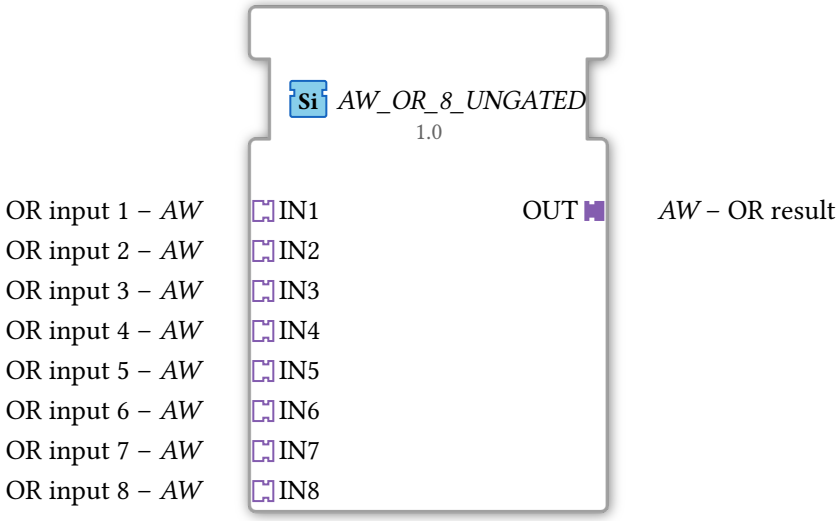
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_OR_8 offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_OR_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_8_UNGATED is a generic function block for the bitwise OR operation across 8 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AW)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_8_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_8`, which combines individual Boolean truth values, `AW_OR_8_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_8_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_8`**: The gated variant – updates the output only on an actual value change.

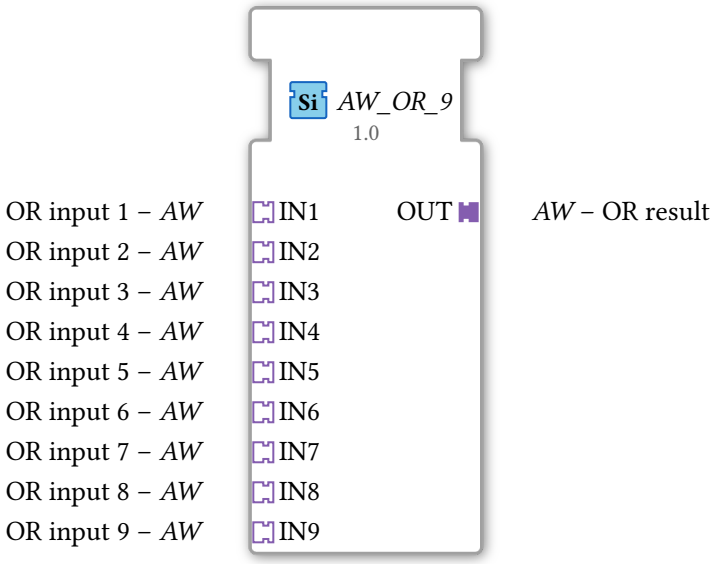
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_OR_8_UNGATED offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_9



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_9 is a generic function block for the bitwise OR operation across 9 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AW)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AW)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 9 input adapters (`IN1` … `IN9`), the block combines the bit patterns of all 9 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_9 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_9`, which combines individual Boolean truth values, `AW_OR_9` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_9` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_OR_9_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

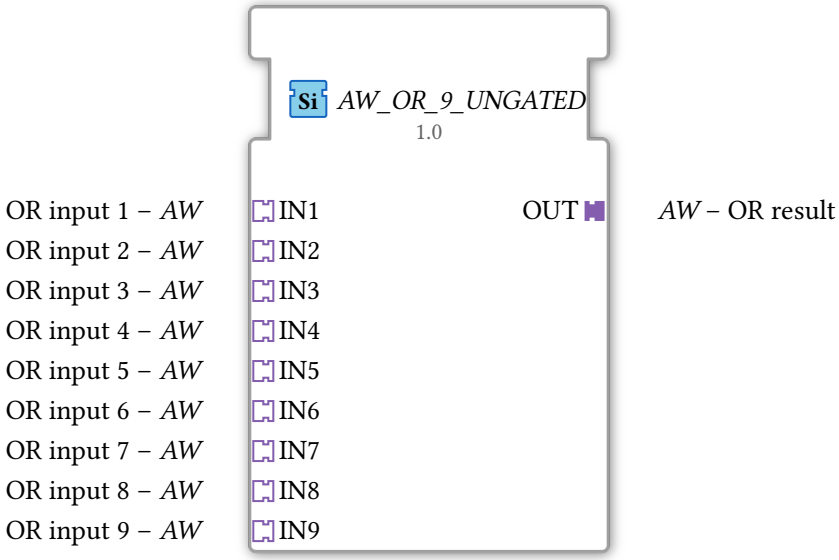
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_OR_9 offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_OR_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_OR_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_OR_9_UNGATED is a generic function block for the bitwise OR operation across 9 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_OR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: OR input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: OR input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: OR input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: OR input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: OR input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: OR input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: OR input 7 (Type: adapter::types::unidirectional::AW)
- **IN8**: OR input 8 (Type: adapter::types::unidirectional::AW)
- **IN9**: OR input 9 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: OR result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 9 input adapters (`IN1` … `IN9`), the block combines the bit patterns of all 9 inputs bitwise using **OR** and writes the result to the output adapter `OUT`. The starting value of the

operation is the identity element (all bits cleared / 0 (identity element of the OR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_OR`) and covers all arities (2 to 10 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_OR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_OR_9_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared OR operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_OR_9`, which combines individual Boolean truth values, `AW_OR_9_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block OR_2, `AW_OR_9_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_OR_9`: The gated variant – updates the output only on an actual value change.

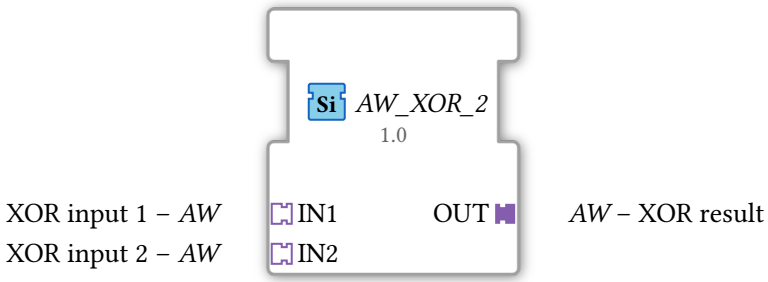
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_OR_9_UNGATED offers a reliable, generic implementation of the bitwise OR function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_2



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_2 is a generic function block for the bitwise XOR (exclusive OR) operation across 2 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_2 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_2`, which combines individual Boolean truth values, `AW_XOR_2` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_2` uses adapter-based interfaces

instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_XOR_2_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

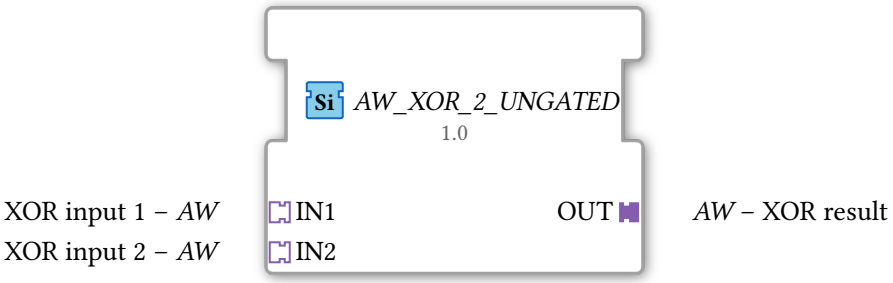
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_XOR_2 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_XOR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_2_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 2 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)

- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 2 input adapters (`IN1` … `IN2`), the block combines the bit patterns of all 2 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_2_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_2`, which combines individual Boolean truth values, `AW_XOR_2_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_2_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_XOR_2`**: The gated variant – updates the output only on an actual value change.

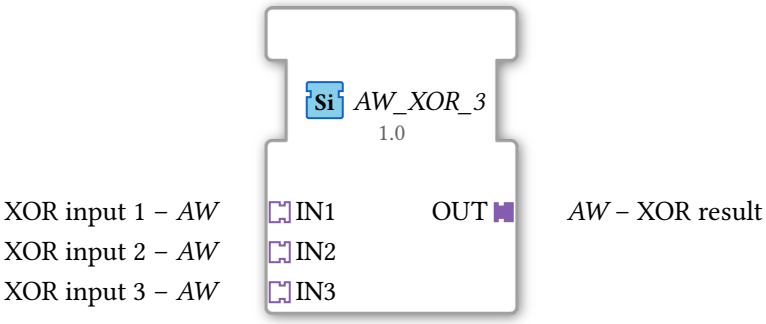
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_XOR_2_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_3



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_3 is a generic function block for the bitwise XOR (exclusive OR) operation across 3 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_3 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_3`, which combines individual Boolean truth values, `AW_XOR_3` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_3` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_XOR_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

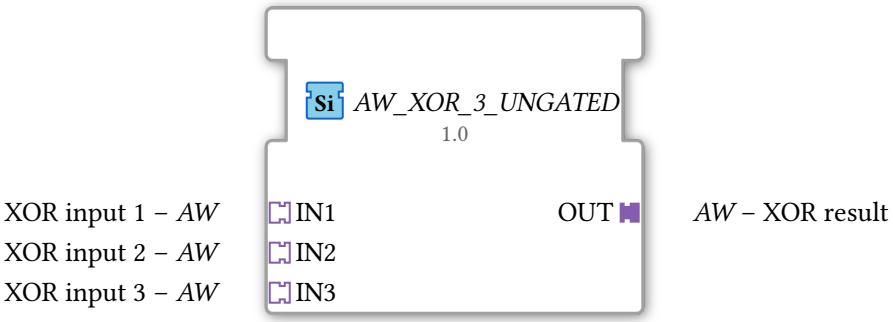
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_XOR_3 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_XOR_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_3_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 3 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality
As soon as an event arrives at one of the 3 input adapters (`IN1` … `IN3`), the block combines the bit patterns of all 3 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / `0` (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features
- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview
Since this is a combinational logic block, AW_XOR_3_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios
- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks
Unlike `AX_XOR_3`, which combines individual Boolean truth values, `AW_XOR_3_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_3_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_XOR_3`**: The gated variant – updates the output only on an actual value change.

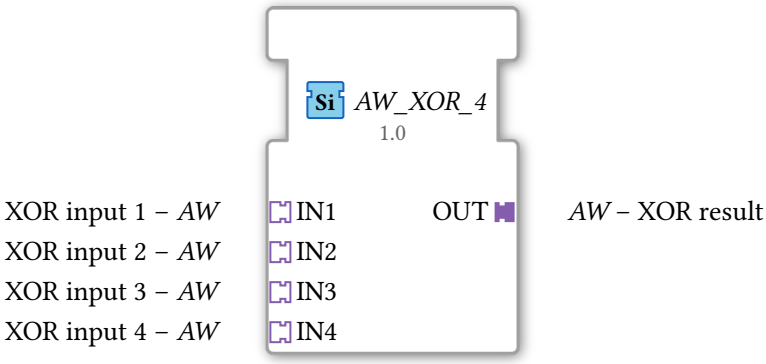
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_XOR_3_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_4



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_4 is a generic function block for the bitwise XOR (exclusive OR) operation across 4 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The

starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_4 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_4`, which combines individual Boolean truth values, `AW_XOR_4` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_4` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_XOR_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

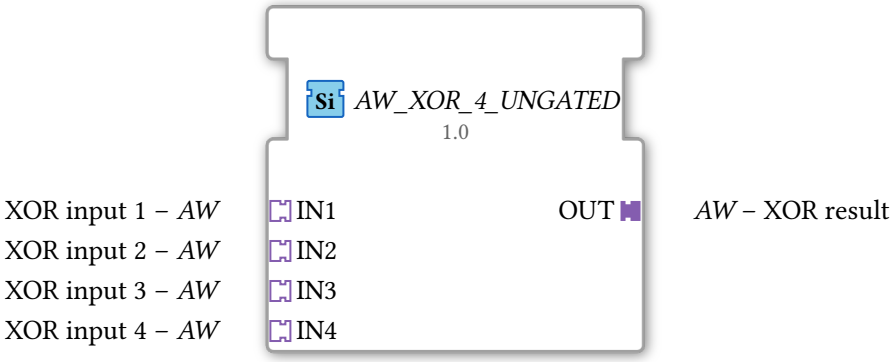
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_XOR_4 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_XOR_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_4_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 4 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 4 input adapters (`IN1` … `IN4`), the block combines the bit patterns of all 4 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.

- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_4_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_4`, which combines individual Boolean truth values, `AW_XOR_4_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_4_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_XOR_4`**: The gated variant – updates the output only on an actual value change.

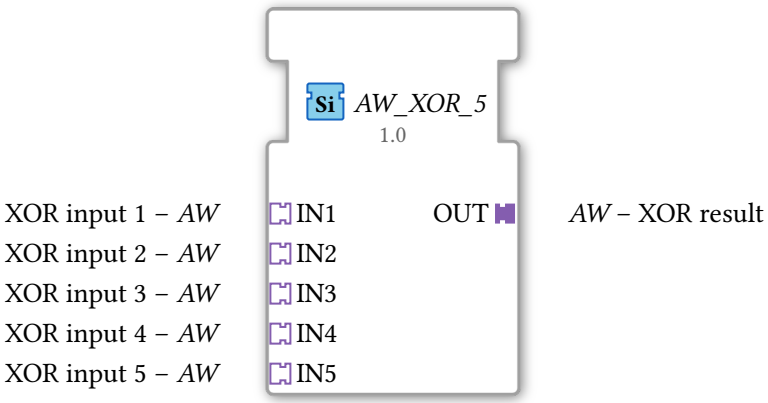
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_XOR_4_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_5



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_5 is a generic function block for the bitwise XOR (exclusive OR) operation across 5 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_5 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_5`, which combines individual Boolean truth values, `AW_XOR_5` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_5` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_XOR_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

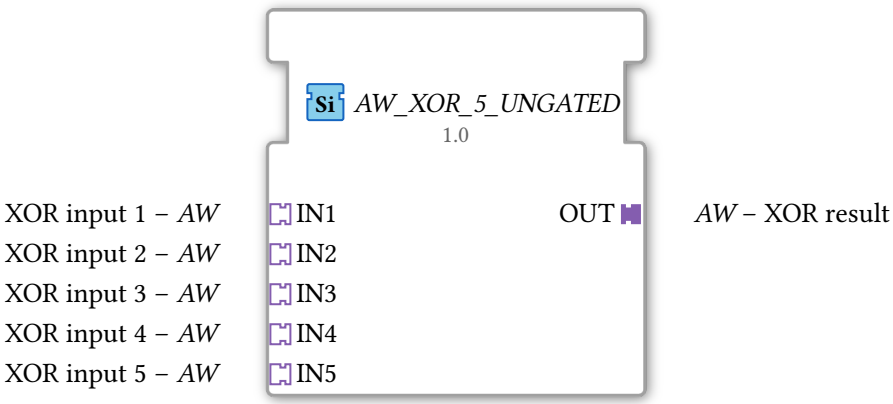
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_XOR_5 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_XOR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_5_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 5 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)

- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 5 input adapters (`IN1` … `IN5`), the block combines the bit patterns of all 5 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_5_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_5`, which combines individual Boolean truth values, `AW_XOR_5_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_5_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_XOR_5`**: The gated variant – updates the output only on an actual value change.

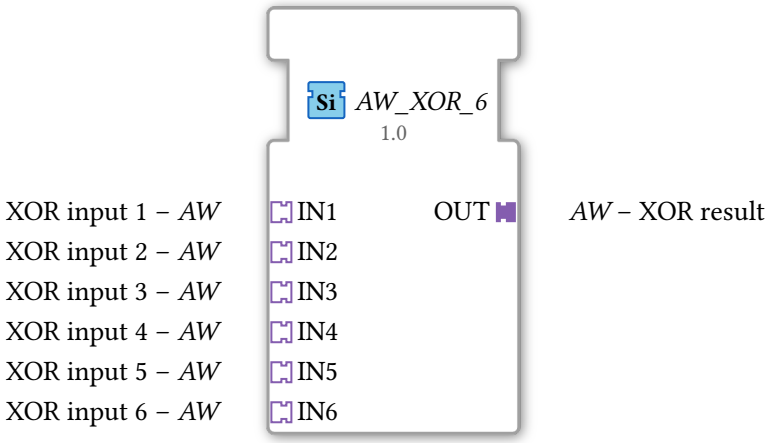
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_XOR_5_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_6



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_6 is a generic function block for the bitwise XOR (exclusive OR) operation across 6 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_6 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_6`, which combines individual Boolean truth values, `AW_XOR_6` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_6` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_XOR_6_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

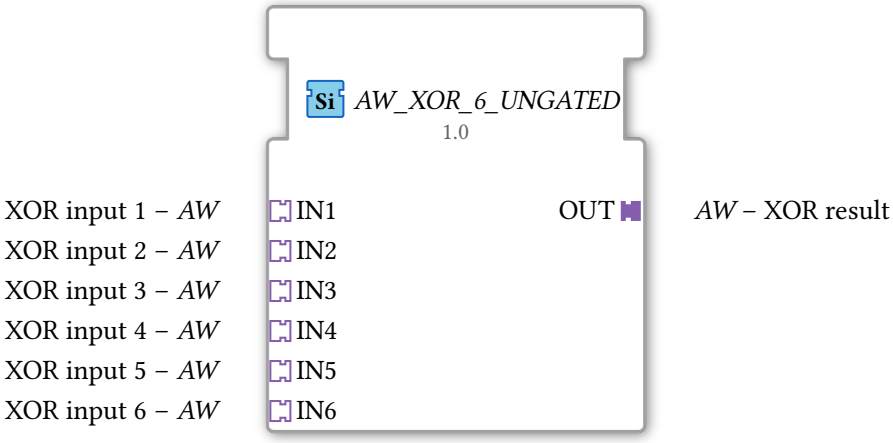
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_XOR_6 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_XOR_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_6_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 6 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 6 input adapters (`IN1` … `IN6`), the block combines the bit patterns of all 6 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_6_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_6`, which combines individual Boolean truth values, `AW_XOR_6_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_6_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_XOR_6`: The gated variant – updates the output only on an actual value change.

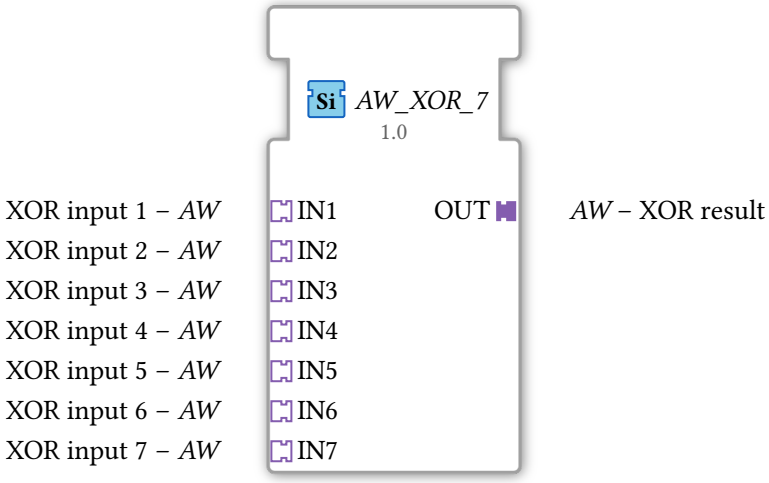
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_XOR_6_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_7



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_7 is a generic function block for the bitwise XOR (exclusive OR) operation across 7 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_7 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike AX_XOR_7, which combines individual Boolean truth values, AW_XOR_7 operates on the full bit pattern of a WORD value. Compared to the standard block XOR_2, AW_XOR_7 uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **AW_XOR_7_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

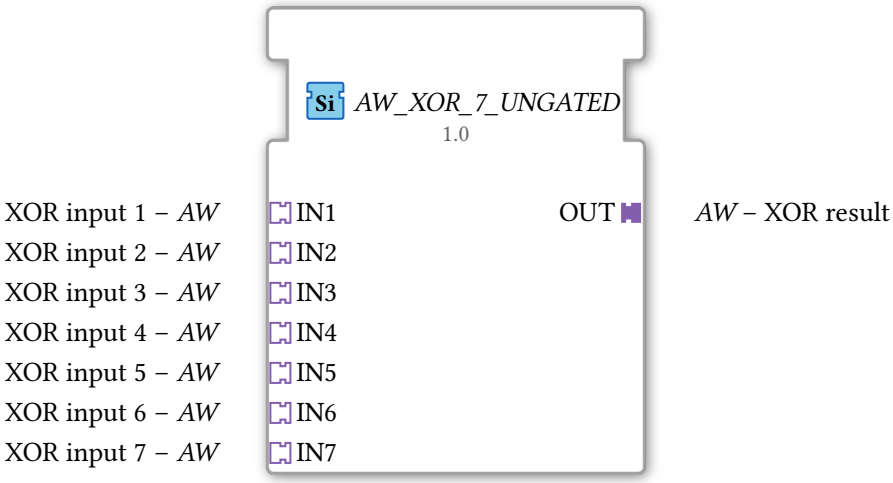
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_XOR_7 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for WORD values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of AW_XOR_7. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_7_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 7 input values of type WORD (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the AX_XOR blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 7 input adapters (`IN1` … `IN7`), the block combines the bit patterns of all 7 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_7_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_7`, which combines individual Boolean truth values, `AW_XOR_7_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_7_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- **`AW_XOR_7`**: The gated variant – updates the output only on an actual value change.

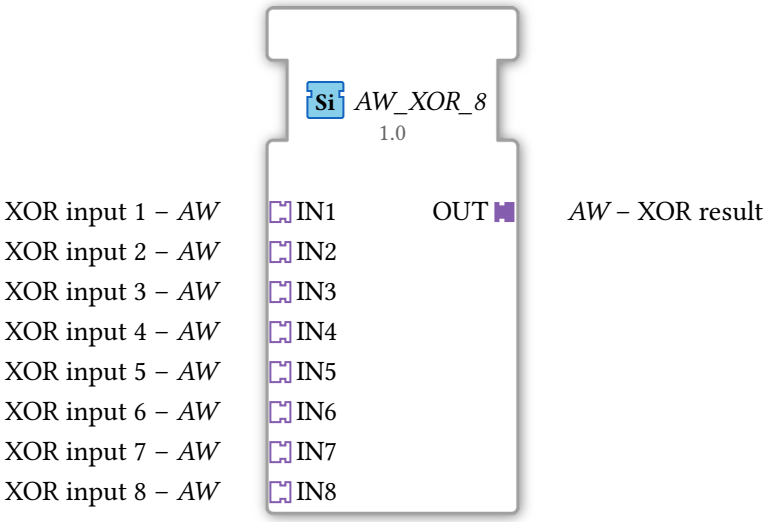
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_XOR_7_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_8



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_8 is a generic function block for the bitwise XOR (exclusive OR) operation across 8 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AW)
- **IN8**: XOR (exclusive OR) input 8 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_8 has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_8`, which combines individual Boolean truth values, `AW_XOR_8` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_8` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_XOR_8_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

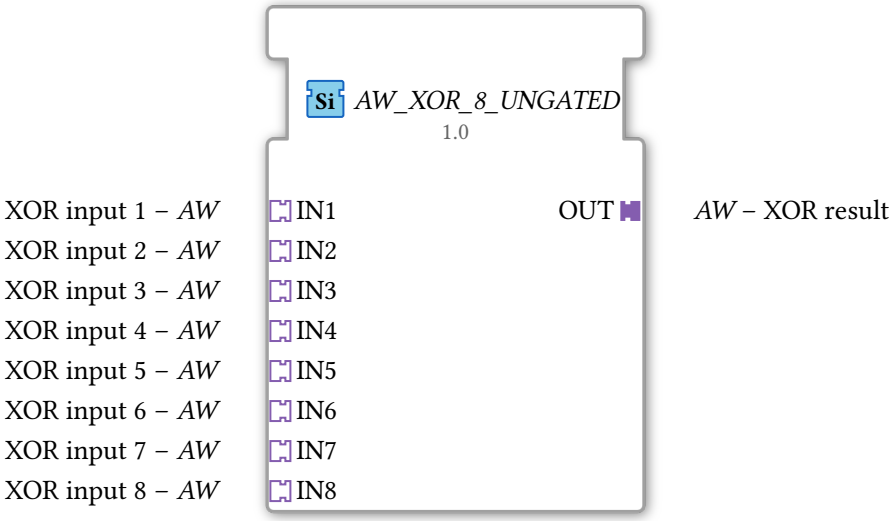
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AW_XOR_8 offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

AW_XOR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_XOR_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_XOR_8_UNGATED is a generic function block for the bitwise XOR (exclusive OR) operation across 8 input values of type `WORD` (16-bit bit pattern (word)). Unlike the Boolean operation on single truth values (as in the `AX_XOR` blocks), every bit of the data word is combined independently here.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1**: XOR (exclusive OR) input 1 (Type: adapter::types::unidirectional::AW)
- **IN2**: XOR (exclusive OR) input 2 (Type: adapter::types::unidirectional::AW)
- **IN3**: XOR (exclusive OR) input 3 (Type: adapter::types::unidirectional::AW)
- **IN4**: XOR (exclusive OR) input 4 (Type: adapter::types::unidirectional::AW)
- **IN5**: XOR (exclusive OR) input 5 (Type: adapter::types::unidirectional::AW)
- **IN6**: XOR (exclusive OR) input 6 (Type: adapter::types::unidirectional::AW)
- **IN7**: XOR (exclusive OR) input 7 (Type: adapter::types::unidirectional::AW)
- **IN8**: XOR (exclusive OR) input 8 (Type: adapter::types::unidirectional::AW)

Output Adapter:

- **OUT**: XOR (exclusive OR) result (Type: adapter::types::unidirectional::AW)

Functionality

As soon as an event arrives at one of the 8 input adapters (`IN1` … `IN8`), the block combines the bit patterns of all 8 inputs bitwise using **XOR (exclusive OR)** and writes the result to the output adapter `OUT`. The starting value of the operation is the identity element (all bits cleared / 0 (identity element of the XOR operation)), so that with only one input actually connected, its value is passed through unchanged.

Only if the newly computed result differs from the value currently held on `OUT` is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

Technical Features

- **Generic function block**: The FB is defined as a generic type (`GEN_AW_XOR`) and covers all arities (2 to 8 inputs) of the same underlying logic via the GenericClassName mechanism.
- **Bitwise operation**: Unlike the Boolean `AX_XOR` blocks, every bit of the `WORD` data word is combined individually here, not just a single truth value.
- **Unidirectional adapters**: All adapters are of type `unidirectional::AW` – data flows only from the socket to the plug.
- **Standard compliance**: The block implements the operation according to IEC 61499-2 / IEC 61131-3.

State Overview

Since this is a combinational logic block, AW_XOR_8_UNGATED has no internal states. The output is recalculated directly from the current input values on every incoming event.

Application Scenarios

- **Bit mask combination**: Combining multiple status registers or flag bytes of type `WORD` into a single result.
- **Signal aggregation**: Merging multiple `WORD` data sources (e.g., from different modules) via a shared XOR (exclusive OR) operation.
- **Diagnostics and status evaluation**: Checking bit patterns for commonly or differently set bits.

⚖️ Comparison with Similar Blocks

Unlike `AX_XOR_8`, which combines individual Boolean truth values, `AW_XOR_8_UNGATED` operates on the full bit pattern of a `WORD` value. Compared to the standard block XOR_2, `AW_XOR_8_UNGATED` uses adapter-based interfaces instead of direct data/event inputs/outputs, enabling more flexible integration into adapter-based system architectures.

- `AW_XOR_8`: The gated variant – updates the output only on an actual value change.

Change Detection

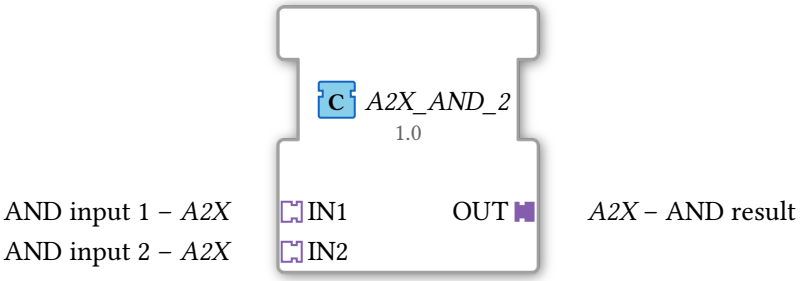
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

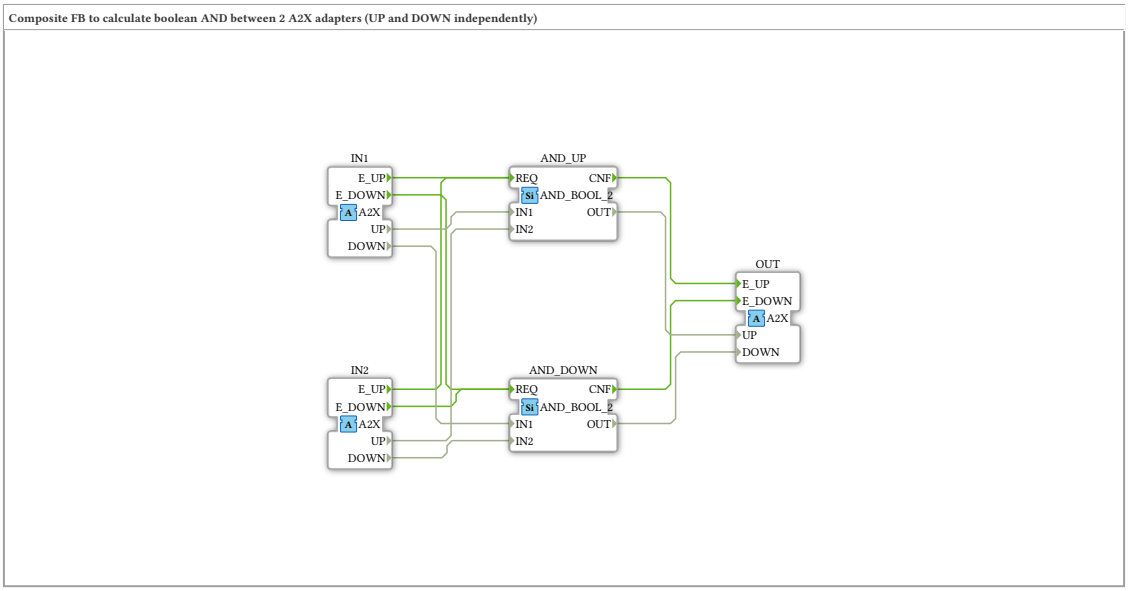
Conclusion

AW_XOR_8_UNGATED offers a reliable, generic implementation of the bitwise XOR (exclusive OR) function for `WORD` values with adapter-based interfaces. Its generic nature makes it versatile for use in automation projects developed according to the IEC 61499 standard that need to combine multiple bit patterns of the same data type.

2.1.11.7 booleanOperators

A2X_AND_2





• – ○ • – ○ • – ○ •

Introduction

The A2X_AND_2 function block computes the logical AND of two A2X adapters — separately for the UP and DOWN channels. Unlike the single-channel, generic `AX_AND` blocks, A2X_AND_2 is a hand-built composite function block that internally wires two standard `AND_BOOL_2` blocks, one per channel.

Interface Structure

Event Inputs

The function block has no direct event inputs — events flow through the `IN1`/`IN2` adapters.

Event Outputs

The function block has no direct event outputs — events flow through the `OUT` adapter.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

Input Adapters:

- **IN1**: AND input 1 (type: `adapter::types::unidirectional::A2X`)
- **IN2**: AND input 2 (type: `adapter::types::unidirectional::A2X`)

Output Adapter:

- **OUT**: AND result (type: `adapter::types::unidirectional::A2X`)

Functionality

The block contains two internal instances of `iec61131::booleanOperators::AND_BOOL_2`: `AND_UP` combines the UP channels of `IN1` and `IN2`, `AND_DOWN` the DOWN channels. When a UP event (`E_UP`) arrives on `IN1` or `IN2`, `AND_UP.REQ` is triggered, the result of `IN1.UP AND IN2.UP` is computed, and it is output via `OUT.E_UP`/`OUT.UP`. The DOWN channel runs the same logic independently through `AND_DOWN`. Because an event destination — unlike a data destination — may have multiple sources, both `IN1.E_UP` and `IN2.E_UP` are allowed to feed into `AND_UP.REQ`; the result is recomputed on either triggering event.

Technical Details

- A genuine composite block (not the `GenericClassName` mechanism used by `AX_AND_2`), built from two standard `AND_BOOL_2` instances
- Two independent channels (UP/DOWN), each with its own event pair — no shared state between channels
- Since IEC 61499 allows multiple sources on an event destination but not on a data destination, one logic block is inserted per channel instead of wiring the two input values directly onto a shared variable

State Overview

The block is a combinational logic block with no internal state; every incoming event recomputes the result directly:

- IN1.E_UP, IN2.E_UP → AND_UP.REQ; IN1.UP, IN2.UP → AND_UP.IN1/IN2; AND_UP.CNF → OUT.E_UP; AND_UP.OUT → OUT.UP
- IN1.E_DOWN, IN2.E_DOWN → AND_DOWN.REQ; IN1.DOWN, IN2.DOWN → AND_DOWN.IN1/IN2; AND_DOWN.CNF → OUT.E_DOWN; AND_DOWN.OUT → OUT.DOWN

Application Scenarios

- Safety logic where two UP/DOWN signal sources must be active simultaneously before an actuator is enabled
- Interlocks between two redundant A2X signal sources
- Combinational logic in controllers built on A2X adapters instead of plain BOOL signals

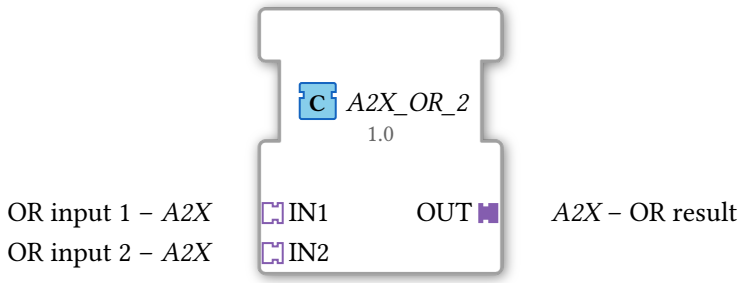
⚖️ Comparison with Similar Blocks

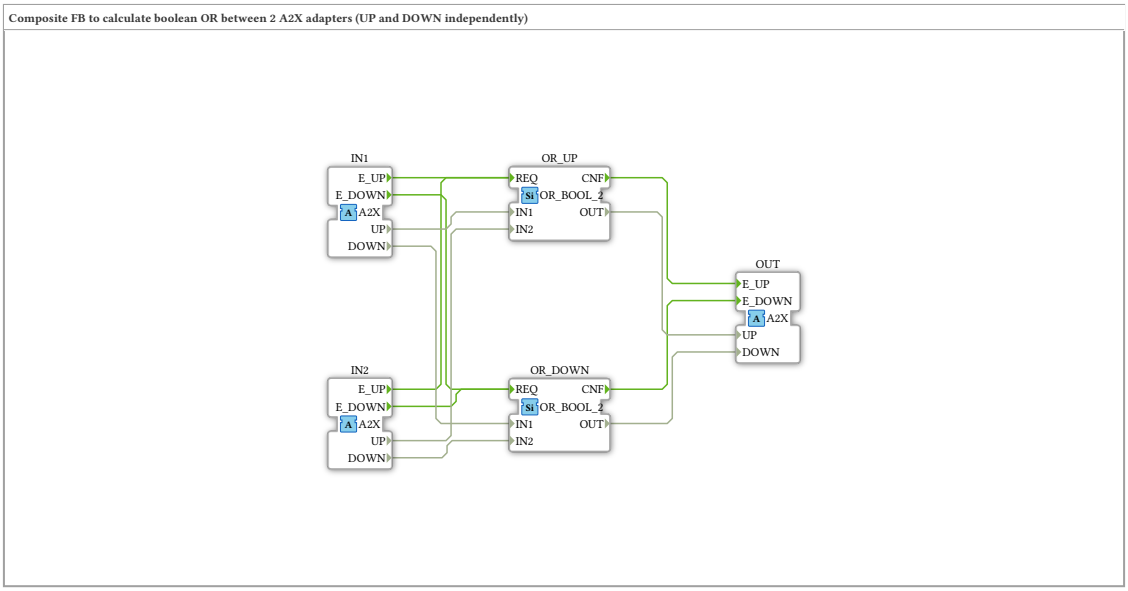
A2X_OR_2 and A2X_XOR_2 are structurally identical but use `OR_BOOL_2` and `XOR_BOOL_2` instead of `AND_BOOL_2`. The single-channel predecessor AX_AND_2 combines only a single Boolean signal and, unlike A2X_AND_2, is implemented as a generic block (`GEN_AX_AND`). Unlike AB_AND_2, which combines a complete `BYTE` bit pattern, A2X_AND_2 processes individual Boolean truth values per channel.

Conclusion

A2X_AND_2 brings the logical AND operation to the two-channel A2X world: two independent, event-driven AND combinations for UP and DOWN, cleanly separated and without multiple writers on a shared variable.

A2X_OR_2





• – ○ • – ○ • – ○ •

Introduction

The A2X_OR_2 function block computes the logical OR of two A2X adapters — separately for the UP and DOWN channels. Like A2X_AND_2, it is a hand-built composite function block that internally wires two standard `OR_BOOL_2` blocks, one per channel.

Interface Structure

Event Inputs

The function block has no direct event inputs — events flow through the `IN1`/`IN2` adapters.

Event Outputs

The function block has no direct event outputs — events flow through the `OUT` adapter.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

Input Adapters:

- **IN1**: OR input 1 (type: `adapter::types::unidirectional::A2X`)
- **IN2**: OR input 2 (type: `adapter::types::unidirectional::A2X`)

Output Adapter:

- **OUT**: OR result (type: `adapter::types::unidirectional::A2X`)

Functionality

The block contains two internal instances of `iec61131::booleanOperators::OR_BOOL_2`: `OR_UP` combines the UP channels of `IN1` and `IN2`, `OR_DOWN` the DOWN channels. When a UP event (`E_UP`) arrives on `IN1` or `IN2`, `OR_UP.REQ` is triggered, the result of `IN1.UP OR IN2.UP` is computed, and it is output via `OUT.E_UP`/`OUT.UP`. The DOWN channel runs the same logic independently through `OR_DOWN`.

Technical Details

- A genuine composite block (not the `GenericClassName` mechanism), built from two standard `OR_BOOL_2` instances

- Two independent channels (UP/DOWN), each with its own event pair — no shared state between channels
- Since IEC 61499 allows multiple sources on an event destination but not on a data destination, one logic block is inserted per channel instead of wiring the two input values directly onto a shared variable

State Overview

The block is a combinational logic block with no internal state; every incoming event recomputes the result directly:

- IN1.E_UP, IN2.E_UP ⟶ OR_UP.REQ; IN1.UP, IN2.UP ⟶ OR_UP.IN1/IN2; OR_UP.CNF ⟶ OUT.E_UP; OR_UP.OUT ⟶ OUT.UP
- IN1.E_DOWN, IN2.E_DOWN ⟶ OR_DOWN.REQ; IN1.DOWN, IN2.DOWN ⟶ OR_DOWN.IN1/IN2; OR_DOWN.CNF ⟶ OUT.E_DOWN; OR_DOWN.OUT ⟶ OUT.DOWN

Application Scenarios

- Merging two redundant or alternative UP/DOWN signal sources into a single result
- Group alarms where triggering any one of several A2X signal sources is sufficient
- Combinational logic in controllers built on A2X adapters instead of plain BOOL signals

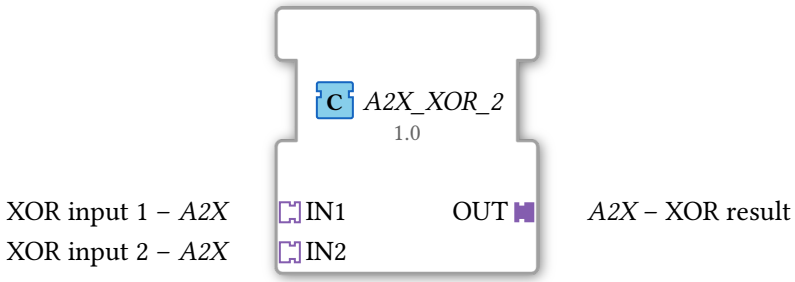
⚖️ Comparison with Similar Blocks

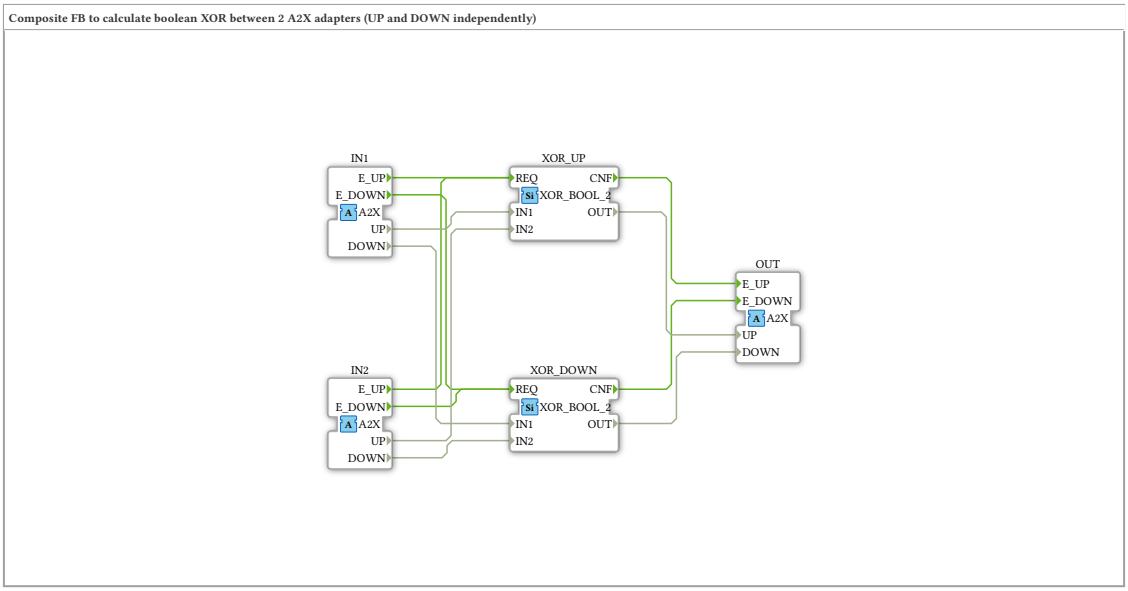
A2X_AND_2 and A2X_XOR_2 are structurally identical but use `AND_BOOL_2` and `XOR_BOOL_2` instead of `OR_BOOL_2`. The single-channel predecessor AX_OR_2 combines only a single Boolean signal and, unlike A2X_OR_2, is implemented as a generic block. Unlike bitwise blocks such as AB_AND_2, A2X_OR_2 processes individual Boolean truth values per channel.

Conclusion

A2X_OR_2 brings the logical OR operation to the two-channel A2X world: two independent, event-driven OR combinations for UP and DOWN, cleanly separated and without multiple writers on a shared variable.

A2X_XOR_2





• – ○ • – ○ • – ○ •

Introduction

The A2X_XOR_2 function block computes the logical exclusive-OR of two A2X adapters — separately for the UP and DOWN channels. Like A2X_AND_2 and A2X_OR_2, it is a hand-built composite function block that internally wires two standard `XOR_BOOL_2` blocks, one per channel.

Interface Structure

Event Inputs

The function block has no direct event inputs — events flow through the `IN1`/`IN2` adapters.

Event Outputs

The function block has no direct event outputs — events flow through the `OUT` adapter.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

Input Adapters:

- **IN1**: XOR input 1 (type: `adapter::types::unidirectional::A2X`)
- **IN2**: XOR input 2 (type: `adapter::types::unidirectional::A2X`)

Output Adapter:

- **OUT**: XOR result (type: `adapter::types::unidirectional::A2X`)

Functionality

The block contains two internal instances of `iec61131::booleanOperators::XOR_BOOL_2`: `XOR_UP` combines the UP channels of `IN1` and `IN2`, `XOR_DOWN` the DOWN channels. When a UP event (`E_UP`) arrives on `IN1` or `IN2`, `XOR_UP.REQ` is triggered, the result of `IN1.UP XOR IN2.UP` is computed, and it is output via `OUT.E_UP`/`OUT.UP`. The DOWN channel runs the same logic independently through `XOR_DOWN`.

Technical Details

- A genuine composite block (not the `GenericClassName` mechanism), built from two standard `XOR_BOOL_2` instances

- Two independent channels (UP/DOWN), each with its own event pair — no shared state between channels
- Since IEC 61499 allows multiple sources on an event destination but not on a data destination, one logic block is inserted per channel instead of wiring the two input values directly onto a shared variable

State Overview

The block is a combinational logic block with no internal state; every incoming event recomputes the result directly:

- IN1.E_UP, IN2.E_UP → XOR_UP.REQ; IN1.UP, IN2.UP → XOR_UP.IN1/IN2; XOR_UP.CNF → OUT.E_UP; XOR_UP.OUT → OUT.UP
- IN1.E_DOWN, IN2.E_DOWN → XOR_DOWN.REQ; IN1.DOWN, IN2.DOWN → XOR_DOWN.IN1/IN2; XOR_DOWN.CNF → OUT.E_DOWN; XOR_DOWN.OUT → OUT.DOWN

Application Scenarios

- Detecting contradictions between two UP/DOWN signal sources that should normally run in sync
- Toggle or exclusive-select logic where exactly one signal is allowed to be active
- Combinational logic in controllers built on A2X adapters instead of plain BOOL signals

⚖️ Comparison with Similar Blocks

A2X_AND_2 and A2X_OR_2 are structurally identical but use `AND_BOOL_2` and `OR_BOOL_2` instead of `XOR_BOOL_2`. The single-channel predecessor AX_XOR_2 combines only a single Boolean signal. Unlike bitwise blocks such as AB_AND_2, A2X_XOR_2 processes individual Boolean truth values per channel.

Conclusion

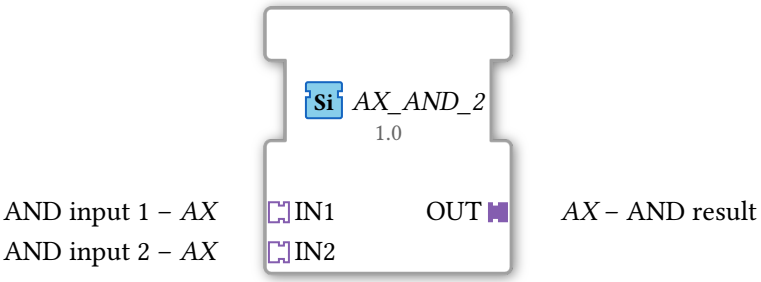
A2X_XOR_2 brings the logical exclusive-OR operation to the two-channel A2X world: two independent, event-driven XOR combinations for UP and DOWN, cleanly separated and without multiple writers on a shared variable.

AX_AND_2

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_AND_2 is a generic function block for calculating the logical AND operation. The block processes two Boolean input signals and outputs the logical AND result.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapter:

- **IN1** - AND input 1 (Adapter type: adapter::types::unidirectional::AX)
- **IN2** - AND input 2 (Adapter type: adapter::types::unidirectional::AX)

Output Adapter:

- **OUT** - AND result (Adapter type: adapter::types::unidirectional::AX)

Functionality

The function block performs a logical AND operation on the two input signals IN1 and IN2. The result is output via the output adapter OUT. Processing is unidirectional via the adapter interfaces.

Technical Features

- Generic function block with the specific class name 'GEN_AX_AND'
- Uses unidirectional adapters for signal transmission
- Implemented according to the IEC 61499-2 standard

State Overview

Since it is a combinational logic block, the AX_AND_2 has no internal states. The output is calculated directly from the current input values.

Application Scenarios

- Safety-critical controllers where two conditions must be met simultaneously
- Linking sensor signals in industrial automation systems
- Logical operations in control systems
- Safety shutdowns with multiple conditions

⚖️ Comparison with Similar Blocks

Compared to standard AND blocks, the AX_AND_2 uses adapter-based interfaces instead of direct data and event inputs/outputs. This enables more flexible integration into adapter-based system architectures.
Comparison AND_2

- `AX_AND_2_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

🛠️ Related Exercises

- Exercise_002a_AX
- Exercise_002b3_AX
- Exercise_006a3_AX

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_AND_2 offers a reliable and standards-compliant implementation of the logical AND function with adapter-based interfaces. Its generic nature makes it versatile for use in various automation projects developed according to the IEC 61499 standard.

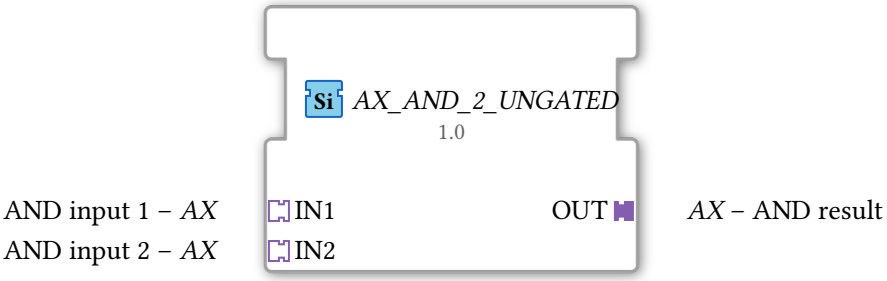
AX_AND_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_AND_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_AND_2_UNGATED is a generic function block for calculating the logical AND operation. The block processes two Boolean input signals and outputs the logical AND result.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapter:

- **IN1** - AND input 1 (Adapter type: adapter::types::unidirectional::AX)
- **IN2** - AND input 2 (Adapter type: adapter::types::unidirectional::AX)

Output Adapter:

- **OUT** - AND result (Adapter type: adapter::types::unidirectional::AX)

Functionality

The function block performs a logical AND operation on the two input signals IN1 and IN2. The result is output via the output adapter OUT. Processing is unidirectional via the adapter interfaces.

Technical Features

- Generic function block with the specific class name ‘GEN_AX_AND’
- Uses unidirectional adapters for signal transmission
- Implemented according to the IEC 61499-2 standard

State Overview

Since it is a combinational logic block, the AX_AND_2_UNGATED has no internal states. The output is calculated directly from the current input values.

Application Scenarios

- Safety-critical controllers where two conditions must be met simultaneously
- Linking sensor signals in industrial automation systems
- Logical operations in control systems
- Safety shutdowns with multiple conditions

⚖️ Comparison with Similar Blocks

Compared to standard AND blocks, the AX_AND_2_UNGATED uses adapter-based interfaces instead of direct data and event inputs/outputs. This enables more flexible integration into adapter-based system architectures.

Comparison AND_2

- **AX_AND_2**: The gated variant – updates the output only on an actual value change.

🛠️ Related Exercises

- Exercise_002a_AX
- Exercise_002b3_AX
- Exercise_006a3_AX

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

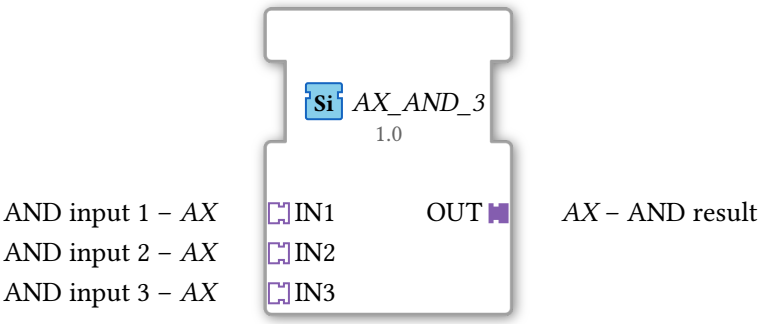
The AX_AND_2_UNGATED offers a reliable and standards-compliant implementation of the logical AND function with adapter-based interfaces. Its generic nature makes it versatile for use in various automation projects developed according to the IEC 61499 standard.

AX_AND_3

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_AND_3 is a generic function block for calculating a three-input logical AND operation. The block performs a Boolean AND operation on three independent input signals and outputs the result via an adapter output.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters (Sockets):

- **IN1** - AND Input 1 (Type: adapter::types::unidirectional::AX)

- **IN2** - AND Input 2 (Type: adapter::types::unidirectional::AX)
- **IN3** - AND Input 3 (Type: adapter::types::unidirectional::AX)

Output Adapters (Plugs):

- **OUT** - AND Result (Type: adapter::types::unidirectional::AX)

Functionality

The AX_AND_3 block performs a logical AND operation on the three input signals. The output signal is only active (TRUE) if all three input signals are active (TRUE) simultaneously. Processing is continuous based on the current input values.

Technical Features

- Uses unidirectional adapters for signal transmission
- Implemented as a generic function block with the generic class 'GEN_AX_AND'
- Works with AX-type adapters for standardized signal transmission
- No event-driven control - continuous evaluation

State Overview

The function block does not have an internal state machine, as it operates as a combinational logic circuit. The output is calculated directly from the current combination of input values. ...

Application Scenarios

- Safety controllers where multiple conditions must be met simultaneously
- Linking sensor signals in industrial controllers
- Logical processing in automation systems
- Monitoring systems with multiple conditions

⚖️ Comparison with similar function blocks

Compared to standard AND blocks, AX_AND_3 offers:

- Three inputs instead of the typical two inputs
- Adapter-based interface instead of direct data inputs/outputs
- Specific AX type compatibility
- Unidirectional signal transmission

Comparison with AND_3

🛠️ Related Exercises

- Exercise_002a6_AX

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_AND_3 is a specialized logic function block for applications requiring triple AND operation with standardized AX adapters. Its adapter-based architecture enables easy integration into existing control systems and offers a reliable solution for complex logical connections.

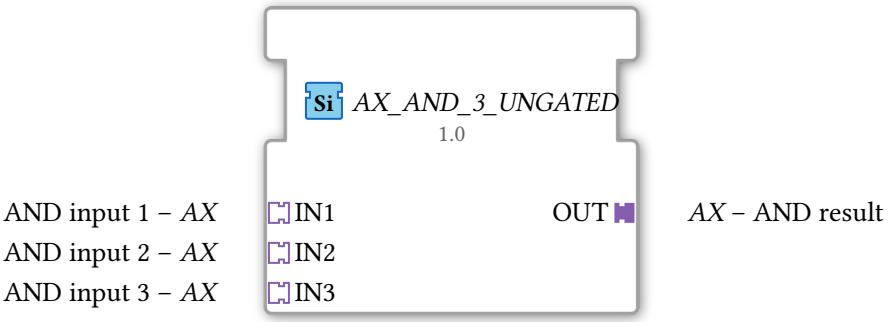
AX_AND_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_AND_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_AND_3_UNGATED is a generic function block for calculating a three-input logical AND operation. The block performs a Boolean AND operation on three independent input signals and outputs the result via an adapter output.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters (Sockets):

- **IN1** - AND Input 1 (Type: adapter::types::unidirectional::AX)
- **IN2** - AND Input 2 (Type: adapter::types::unidirectional::AX)
- **IN3** - AND Input 3 (Type: adapter::types::unidirectional::AX)

Output Adapters (Plugs):

- **OUT** - AND Result (Type: adapter::types::unidirectional::AX)

Functionality

The AX_AND_3_UNGATED block performs a logical AND operation on the three input signals. The output signal is only active (TRUE) if all three input signals are active (TRUE) simultaneously. Processing is continuous based on the current input values.

Technical Features

- Uses unidirectional adapters for signal transmission
- Implemented as a generic function block with the generic class 'GEN_AX_AND'
- Works with AX-type adapters for standardized signal transmission
- No event-driven control - continuous evaluation

State Overview

The function block does not have an internal state machine, as it operates as a combinational logic circuit. The output is calculated directly from the current combination of input values. ...

Application Scenarios

- Safety controllers where multiple conditions must be met simultaneously
- Linking sensor signals in industrial controllers
- Logical processing in automation systems

- Monitoring systems with multiple conditions

⚖️ Comparison with similar function blocks

Compared to standard AND blocks, AX_AND_3_UNGATED offers:

- Three inputs instead of the typical two inputs
- Adapter-based interface instead of direct data inputs/outputs
- Specific AX type compatibility
- Unidirectional signal transmission

Comparison with AND_3

🛠️ Related Exercises

- Exercise_002a6_AX

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

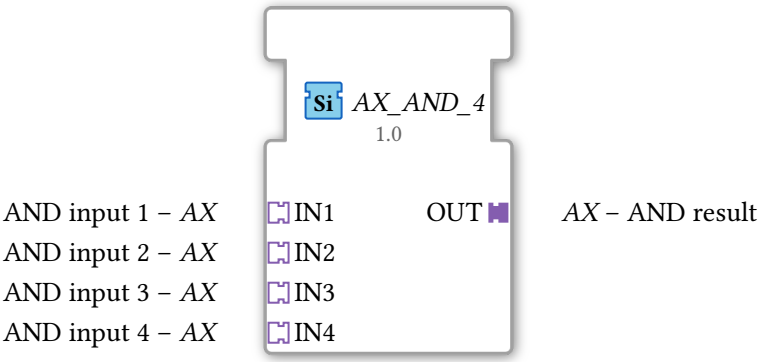
The AX_AND_3_UNGATED is a specialized logic function block for applications requiring triple AND operation with standardized AX adapters. Its adapter-based architecture enables easy integration into existing control systems and offers a reliable solution for complex logical connections.

AX_AND_4

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_AND_4 function block is a generic function block for calculating a four-input logical AND operation. It is used to process Boolean signals in automation systems and allows the combination of multiple input signals into a single output signal.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Plug Adapter:

- **OUT**: AND result (Adapter type: adapter::types::unidirectional::AX)

Socket Adapter:

- **IN1**: AND input 1 (Adapter type: adapter::types::unidirectional::AX)
- **IN2**: AND input 2 (Adapter type: adapter::types::unidirectional::AX)
- **IN3**: AND input 3 (Adapter type: adapter::types::unidirectional::AX)
- **IN4**: AND input 4 (Adapter type: adapter::types::unidirectional::AX)

Functionality

This function block performs a logical AND operation across four inputs. The output signal is only active (TRUE) if all four input signals are active (TRUE) simultaneously. As soon as at least one of the four inputs is inactive (FALSE), the output also becomes inactive.

Technical Features

- Generic function block with predefined class name 'GEN_AX_AND'
- Uses unidirectional adapters for signal transmission
- Implemented as part of the "adapter::booleanOperators" package
- Supports four independent input signals

State Overview

Since it is a combinational logic block, the AX_AND_4 has no internal states. The output is determined solely by the current input values.

Application Scenarios

- Safety controllers where multiple conditions must be met simultaneously
- Monitoring of multiple sensors
- Linking enable signals in process controllers
- Logical linking of control commands

Comparison with AND_4

⚖️ Comparison with similar function blocks

Compared to simple AND blocks with fewer inputs, AX_AND_4 offers the ability to directly link four signals without additional chaining. Compared to OR operations, the AND operation places stricter demands on activation.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_AND_4 function block provides an efficient solution for linking four Boolean input signals using a logical AND operation. Through the use of adapters, it enables flexible integration into various control systems and is particularly suitable for applications where multiple safety or enable conditions must be met simultaneously.

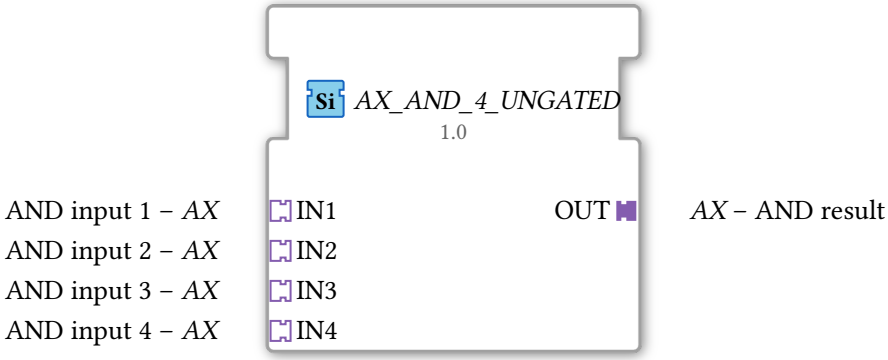
AX_AND_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_AND_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_AND_4_UNGATED function block is a generic function block for calculating a four-input logical AND operation. It is used to process Boolean signals in automation systems and allows the combination of multiple input signals into a single output signal.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Plug Adapter:

- **OUT**: AND result (Adapter type: adapter::types::unidirectional::AX)

Socket Adapter:

- **IN1**: AND input 1 (Adapter type: adapter::types::unidirectional::AX)
- **IN2**: AND input 2 (Adapter type: adapter::types::unidirectional::AX)
- **IN3**: AND input 3 (Adapter type: adapter::types::unidirectional::AX)
- **IN4**: AND input 4 (Adapter type: adapter::types::unidirectional::AX)

Functionality

This function block performs a logical AND operation across four inputs. The output signal is only active (TRUE) if all four input signals are active (TRUE) simultaneously. As soon as at least one of the four inputs is inactive (FALSE), the output also becomes inactive.

Technical Features

- Generic function block with predefined class name ‘GEN_AX_AND’
- Uses unidirectional adapters for signal transmission
- Implemented as part of the “adapter::booleanOperators” package
- Supports four independent input signals

State Overview

Since it is a combinational logic block, the AX_AND_4_UNGATED has no internal states. The output is determined solely by the current input values.

Application Scenarios

- Safety controllers where multiple conditions must be met simultaneously

- Monitoring of multiple sensors
- Linking enable signals in process controllers
- Logical linking of control commands

Comparison with AND_4

⚖️ Comparison with similar function blocks

Compared to simple AND blocks with fewer inputs, AX_AND_4_UNGATED offers the ability to directly link four signals without additional chaining. Compared to OR operations, the AND operation places stricter demands on activation.

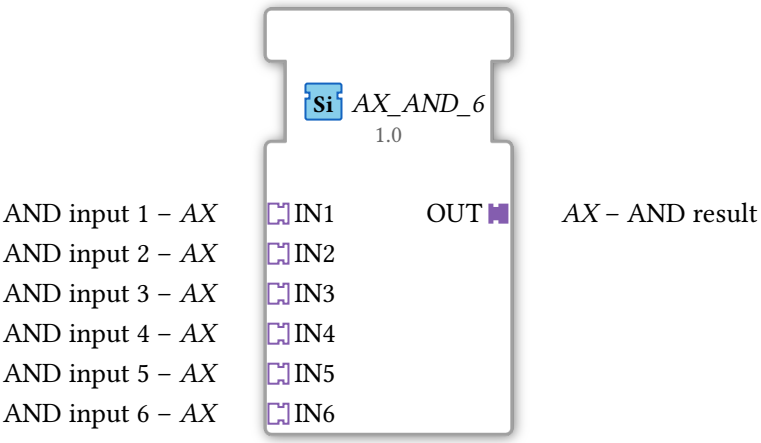
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AX_AND_4_UNGATED function block provides an efficient solution for linking four Boolean input signals using a logical AND operation. Through the use of adapters, it enables flexible integration into various control systems and is particularly suitable for applications where multiple safety or enable conditions must be met simultaneously.

AX_AND_6



Introduction

The `AX_AND_6` function block is a generic block for calculating a logical AND operation across six unidirectional AX adapter inputs. It is used to aggregate and monitor up to 6 input signals (e.g., emergency stop chains, enable or status signals) into a single adapter output signal.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Plug Adapter (Output):

- **OUT** – AND result (Adapter type: `adapter::types::unidirectional::AX`)

Socket Adapters (Inputs):

- **IN1** – AND input 1 (Adapter type: `adapter::types::unidirectional::AX`)
- **IN2** – AND input 2 (Adapter type: `adapter::types::unidirectional::AX`)
- **IN3** – AND input 3 (Adapter type: `adapter::types::unidirectional::AX`)
- **IN4** – AND input 4 (Adapter type: `adapter::types::unidirectional::AX`)
- **IN5** – AND input 5 (Adapter type: `adapter::types::unidirectional::AX`)
- **IN6** – AND input 6 (Adapter type: `adapter::types::unidirectional::AX`)

How It Works

The function block evaluates the logical AND operation of all 6 AX inputs. The output `OUT.D1` is `TRUE` only when all six sockets `IN1.D1` through `IN6.D1` are simultaneously `TRUE`. If at least one input is `FALSE`, the output switches to `FALSE`.

Change Filtering

The result is written to output plug `OUT` and its adapter event `OUT.E1` is fired only when the newly calculated value differs from the currently held value. If the logical result remains unchanged, no adapter event is generated – preventing event flooding in downstream SubApps.

Technical Features

- **Generic Function Block**: Utilizes system class name `GEN_AX_AND` within the `adapter::booleanOperators` package.
- **Pure Adapter Interface**: Enables direct connection without manual event/data extraction.

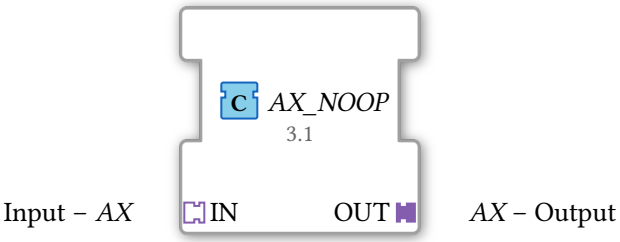
Application Scenarios

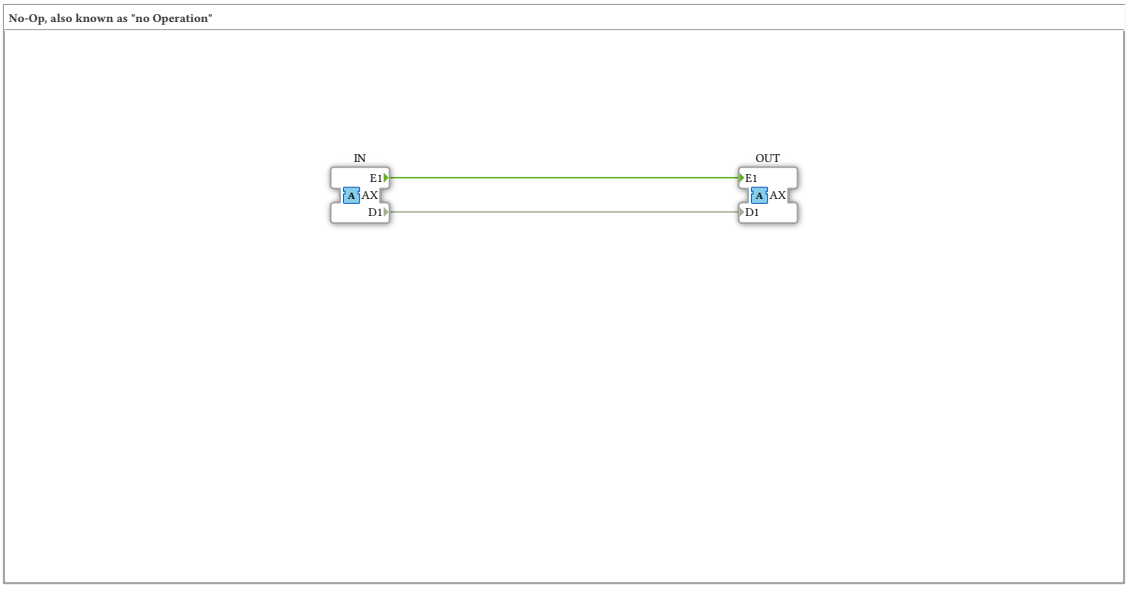
- Aggregated monitoring of up to 6 emergency stop or safety channels (e.g., STG1 I1–I6 emergency stop monitoring).
- Grouping multiple safety and interlock criteria.

See Also

- `AX_AND_2` – 2-input AND block.
- `AX_AND_3` – 3-input AND block.
- `AX_AND_4` – 4-input AND block.

AX_NOOP





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_NOOP** (No Operation) serves as a pure pass-through function. It connects an incoming adapter (`IN`) directly to an outgoing adapter (`OUT`) without modifying data or processing events. This function block is designed as a basic block for testing, debugging, and placeholder purposes.

Interface Structure

Event Inputs

No direct event inputs are available.
Events are received via the adapter `IN`.

Event Outputs

No direct event outputs are available.
Events are passed on via the adapter `OUT`.

Data Inputs

No direct data inputs available.
Data is received via the `IN` adapter.

Data Outputs

No direct data outputs available.
Data is passed on via the `OUT` adapter.

Adapters

Type	Name	Direction	Comment
`adapter::types::unidirectional::AX`	IN	Socket (Input)	Receives events and data
`adapter::types::unidirectional::AX`	OUT	Plug (Output)	Passes on events and data unchanged

Functionality

This module establishes a point-to-point connection between the input adapter `IN` and the output adapter `OUT`. Internally, the following connections are implemented:

- **Event Connection**: `IN.E1` ⟶ `OUT.E1`
- **Data Connection**: `IN.D1` ⟶ `OUT.D1`

Every incoming event at `IN.E1` is immediately forwarded to `OUT.E1`. Every data value at `IN.D1` is passed on unchanged to `OUT.D1`. No logic, delay, or state change occurs.

Technical Features

- Pure passthrough without processing.
- Uses the adapter type `AX` for communication.
- No modifiable parameters or configurations.
- Ideally suited as a placeholder or for debugging adapter connections in the 4diac IDE.

State Overview

The function block does not have its own state machine. Its behavior is static and corresponds to direct wiring.

Application Scenarios

- **Test Environments**: Replacing a complex function block during the test phase to check the wiring.
- **Debugging**: Injecting a measurement or trace point into an adapter path.
- **Placeholder**: Temporarily inserting during the step-by-step implementation of a larger system.
- **Adapter Transparency**: Checking whether an adapter type is correctly connected without functional logic.

Comparison with Similar Function Blocks

Function Block	Description	Difference
`E_NOP`	Event NOP with direct event/data inputs/outputs	Uses standard I/O instead of adapters
`MUX`	Selection between multiple data sources	Provides selection, not pure passthrough
`BYE`	Explicit end-of-service marker	Has special semantics, not passthrough
AX_NOOP	Pure adapter passthrough	Specifically designed for the adapter type `AX`

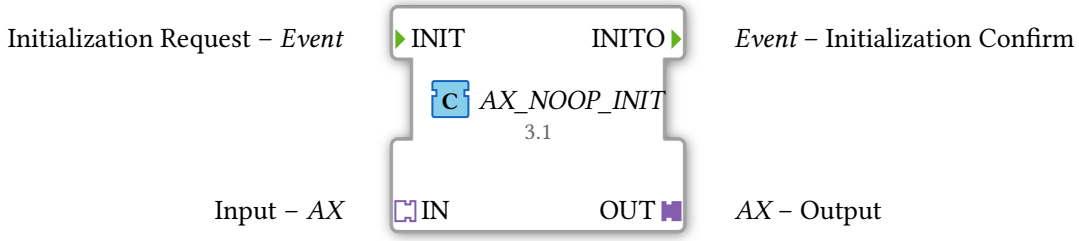
Conclusion

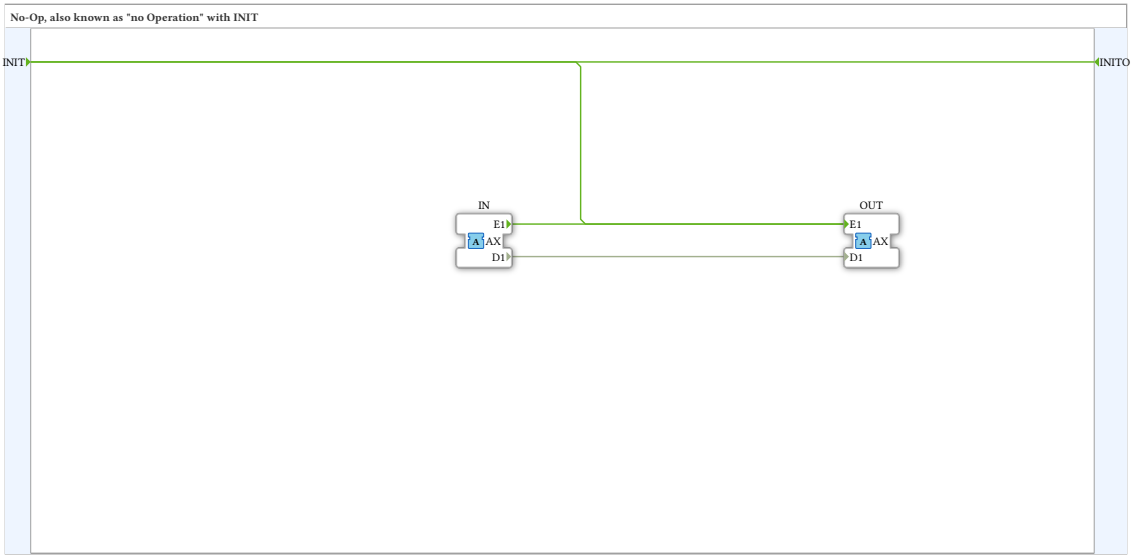
The **AX_NOOP** is a minimalist yet useful component for working with adapter interfaces. Thanks to its simple pass-through functionality, it is ideally suited for testing, debugging, and as a placeholder in the 4diac IDE. Despite lacking its own logic, it plays a crucial role in the step-by-step development and verification of connection-intensive applications.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AX_NOOP_INIT





• – ○ • – ○ • – ○ •

Introduction

The function block `AX_NOOP_INIT` is a no-operation block that forwards incoming events and data from an adapter interface unchanged to an output interface. It also has an initialization function that, upon arrival of the INIT event, triggers both the INITO acknowledgment and the adapter's output event. It serves as a simple pass-through or placeholder block in automation technology.

Interface Structure

Event Inputs

- **INIT** (Type: `EInit`): Initialization request. Upon arrival, it triggers the event outputs `INITO` and `OUT.E1`.

Event Outputs

- **INITO** (Type: `EInit`): Initialization confirmation. Triggered directly by `INIT`.

Data Inputs

This function block does not have separate data inputs at the top level. All data is transmitted via the incoming adapter (`IN`).

Data Outputs

This function block does not have separate data outputs at the top level. All data is transmitted via the outgoing adapter (`OUT`).

Adapters

- **IN** (Socket): Input adapter of type `adapter::types::unidirectional::AX`. Provides an event interface `E1` and a data interface `D1`.
- **OUT** (Plug): Output adapter of type `adapter::types::unidirectional::AX`. Provides an event interface `E1` and a data interface `D1`.

Functionality

The function block operates as a simple pass-through module:

1. **Event Forwarding**:

- The INIT event is forwarded to `INITO` and to `OUT.E1`.
- The event `IN.E1` is passed directly to `OUT.E1`.

1. **Data Forwarding**:

- The value of `IN.D1` is copied unchanged to `OUT.D1`.

No processing or modification of the data takes place. The function block is stateless and simply combines the signal and data paths.

Technical Features

- The function block combines the pure pass-through function of a no-operation function block with an initialization-controlled output to the output adapter.
- The use of adapters (plug/socket) enables clean encapsulation and reusability of the interface definition.
- There are no internal states or timing behaviors – the response occurs immediately upon event input.

State Overview

The `AX_NOOP_INIT` has no states of its own. It reacts immediately to every incoming event and forwards it along with the associated data. A state machine is not present.

Application Scenarios

- **Initialization Cascades**: When multiple components in a chain need to be initialized, `AX_NOOP_INIT` can serve as a link, both acknowledging an INIT request and passing it on to the subsequent component.
- **Test and Simulation Environments**: As a placeholder to bridge interfaces and test functionalities without actual logic.
- **Adapter Passthrough**: When an adapter signal needs to be passed unchanged from a source to a destination without additional processing.

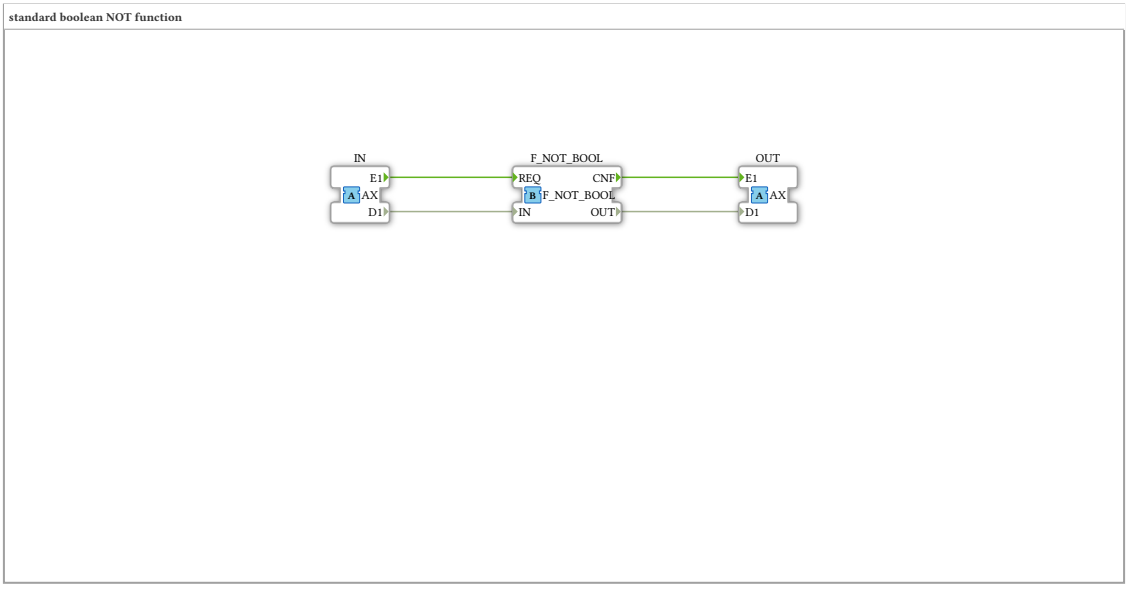
Comparison with Similar Components

A simple `AX_NOOP` (without INIT) would only pass through the IN/OUT events and data, but would not generate a separate initialization acknowledgment (`INITO`). The `AX_NOOP_INIT` extends this functionality with INIT handling, allowing it to be used directly in initialization processes.

Conclusion

The `AX_NOOP_INIT` is a simple yet useful function block for passing events and data at the adapter level. Its INIT functionality makes it particularly valuable in initialization chains or as a replacement for a test block. Its stateless nature makes it easy to understand and resource-efficient.

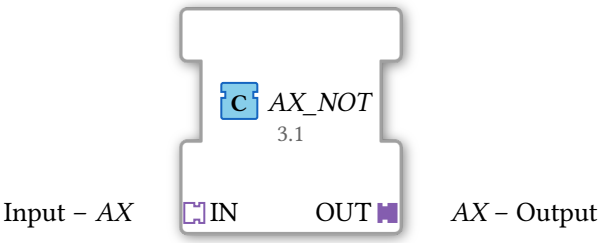
AX_NOT



• – ○ • – ○ • – ○ •

Introduction

The AX_NOT function block is a standardized Boolean NOT operator implemented according to the IEC 61131-3 standard. This block performs a logical negation (NOT operation) on the input values and outputs the inverted result.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **IN** (Socket): Input adapter of type `adapter::types::unidirectional::AX` - used to transmit input data
- **OUT** (Plug): Output adapter of type `adapter::types::unidirectional::AX` - used to output the processed data

Functionality

The AX_NOT function block performs a logical negation on the Boolean values received via the IN adapter. The result of the NOT operation is output via the OUT adapter. Processing is bidirectional between the application and resource layers.

Technical Features

- Implemented according to the IEC 61131-3 standard for Boolean functions
- Uses unidirectional AX adapters for data transmission
- Works with the `adapter::booleanOperators` package
- Supports TypeHash functionality for type identification

State Overview

The function block operates statelessly and performs the NOT operation directly on the incoming data without storing any internal state.

Application Scenarios

- Logical inversion of Boolean signals in control applications
- Signal reversal in automation systems
- Use in combinational circuits
- Integration into larger control logic for signal processing

⚖️ Comparison with Similar Function Blocks

Compared to other Boolean operators such as AND or OR, AX_NOT performs only the negation operation. The use of adapters instead of direct inputs/outputs allows for more flexible integration into more complex system architectures.

Comparison with F_NOT

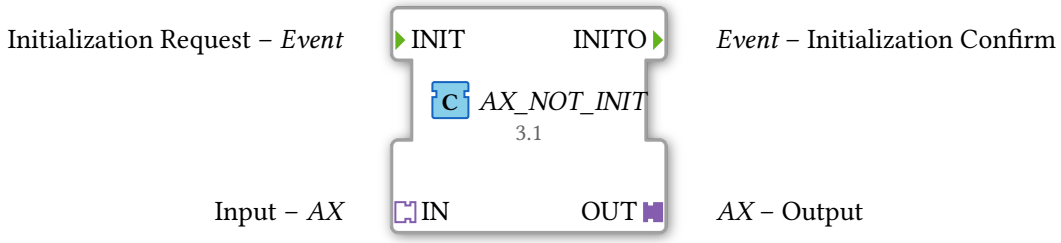
🛠️ Related Exercises

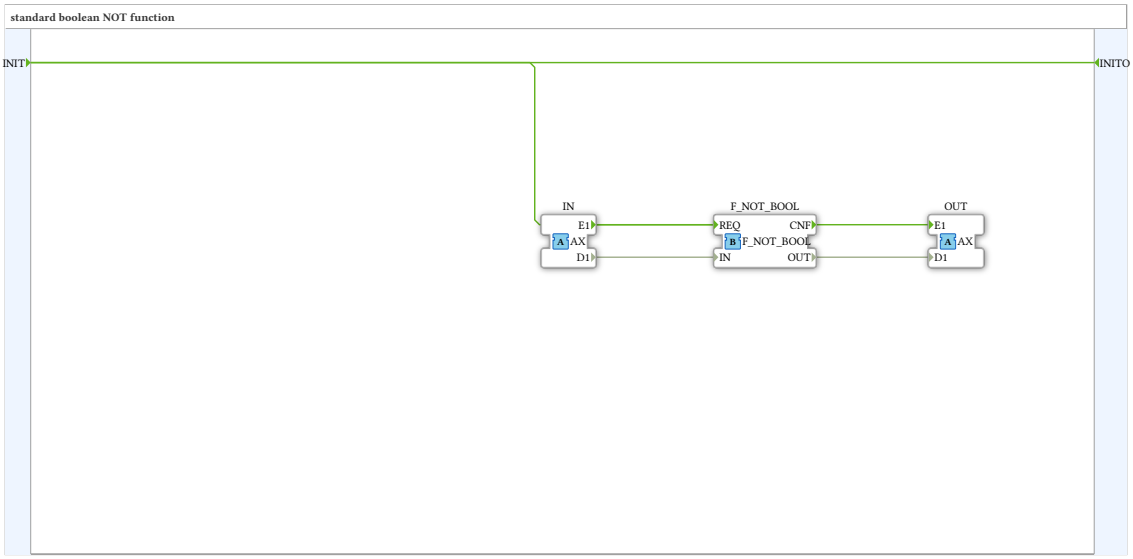
- Exercise_006a3_sub_AX

Conclusion

The AX_NOT function block offers a reliable and standards-compliant implementation of the Boolean NOT operation. Its adapter-based interface design makes it particularly suitable for modular system designs and enables easy integration into existing automation solutions.

AX_NOT_INIT





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AX_NOT_INIT** implements a Boolean negation (NOT function) based on an adapter type. Unlike classic NOT blocks, it operates via a standardized adapter interface and is specifically designed for use in modular automation systems. Initialization occurs via the INIT event input, which triggers the first calculation.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirmation

Data Inputs

This function block has no direct data inputs. Input data is obtained via the **IN** adapter.

Data Outputs

This function block has no direct data outputs. Output data is provided via the **OUT** adapter.

Adapters

Name	Type	Comment	Direction
IN	adapter::types::unidirectional::AX	Input Value	Socket
OUT	adapter::types::unidirectional::AX	Output Value (Negated)	Plug

Functionality

The function block performs a bitwise negation of the value provided via the **IN** adapter and outputs the result via the **OUT** adapter. Processing is initiated by an event at **IN.E1** (an event from the IN adapter) or by the INIT event. Internal steps:

1. An event at **INIT** or **IN.E1** activates the embedded function block **F_NOT_BOOL**.
2. **F_NOT_BOOL** retrieves the value from **IN.D1**, inverts it, and writes the result to **OUT.D1**.
3. Upon completion, an event is output via **OUT.E1** (to the OUT adapter) as well as **INITO**.

If the INIT event is received, **INITO** and **OUT.E1** are triggered immediately – the first negation occurs without a prior input event.

Technical Features

- **Adapter-Based Communication**: The function block uses exclusively unidirectional adapters of type `AX`, which modularly encapsulates the data and event interfaces. This facilitates communication between components using standardized protocols.
- **Initialization Behavior**: The INIT input performs an immediate initial calculation and outputs an acknowledgment event, which is useful for initial value settings or reset processes.
- **Reuse**: Internally, the IEC 61131-3 function block `F_NOT_BOOL` is used, which implements simple negation on Boolean values.

State Overview

The function block does not have an explicit state machine – it reacts event-driven. Simplified process:
Ereignis (INIT oder IN.E1) ⟶ Erfasse IN.D1 ⟶ Berechne NOT ⟶ Setze OUT.D1 ⟶ Sende INITO und/oder OUT.E1 The function block is **stateless**, meaning that each new request is processed independently of previous values.

Application Scenarios

- **Signal Negation in Modular Systems**: If inputs and outputs are defined via adapters, the function block can be used as a universal NOT gate, e.g., for inverting actuator or sensor values.
- **Initialization Logic**: The INIT input ensures that the output provides a defined negated value at startup.
- **Adapter Compatibility**: In systems using the `AX` adapter protocol, the function block serves as a basic functional building block for logic operations.

Comparison with Similar Function Blocks

- **F_NOT_BOOL**: Standard IEC 61131-3 NOT function without an adapter. **AX_NOT_INIT** encapsulates this functionality in an adapter interface and adds initialization behavior.
- **AX_AND / AX_OR**: Analog adapter blocks for AND and OR, respectively. **AX_NOT_INIT** is specific for negation and has only one input.
- **F_TRIG / R_TRIG**: Edge detection that also uses events but does not perform logical negation.

Conclusion

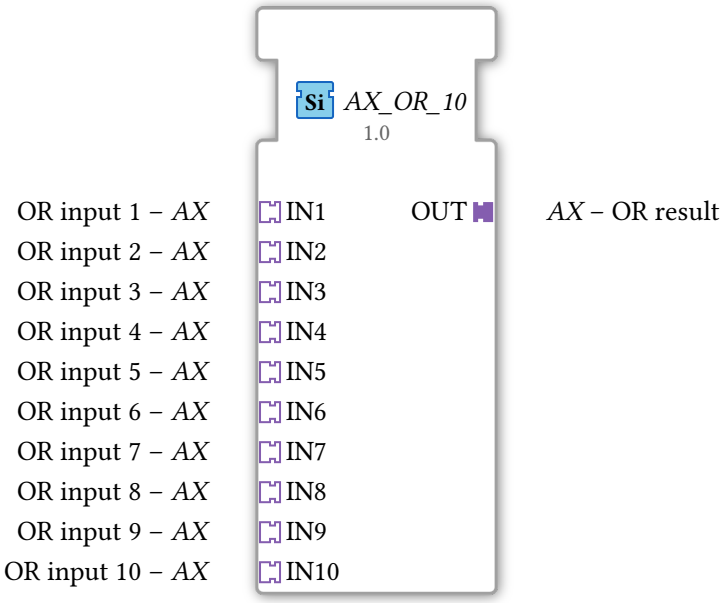
The **AX_NOT_INIT** function block offers an efficient and standardized way to negate Boolean values via the `AX` adapter. By integrating initialization logic, it is particularly suitable for systems that require a defined initial state at startup. It complements the family of adapter-based logic blocks and simplifies the modular structuring of automation solutions.

AX_OR_10

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_10 is a generic function block for calculating the logical OR operation of 10 inputs.



Interface Structure

Adapters

Input Adapters:

- **IN1** to **IN10** (adapter::types::unidirectional::AX)

Output Adapters:

- **OUT** (adapter::types::unidirectional::AX)

Functionality

The function block performs a logical OR operation on the 10 input signals.

Technical Features

- Generic function block with the specific class name 'GEN_AX_OR'
- Uses unidirectional adapters.

State Overview

Combinatory logic block without states.

Application Scenarios

Logical logic gates with many inputs.

⚖️ **Comparison with similar building blocks**

- **AX_OR_2...9**: Variants with fewer inputs.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

Adapter-based OR gate with 10 inputs.

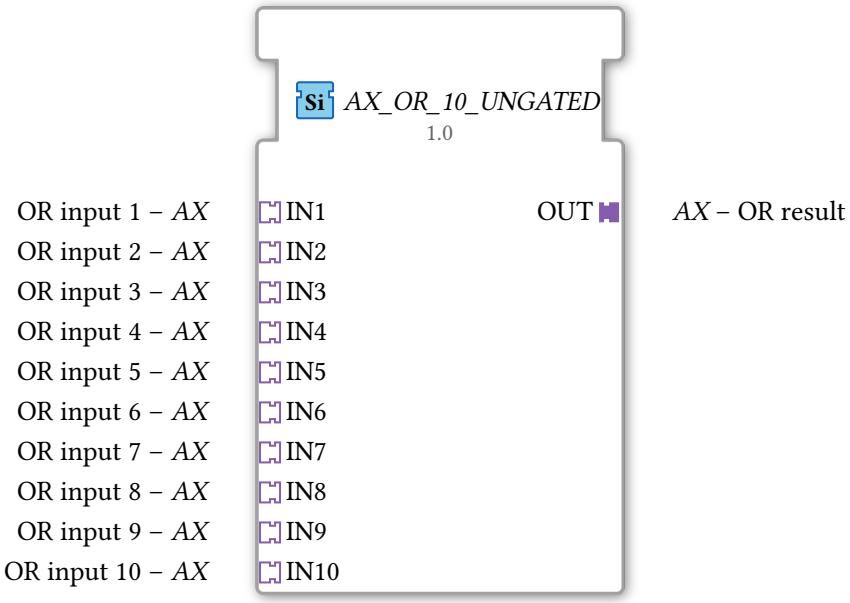
AX_OR_10_UNGATED

i UNGATED variant: This block is the ungated version of `AX_OR_10`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_10_UNGATED is a generic function block for calculating the logical OR operation of 10 inputs.



Interface Structure

Adapters

Input Adapters:

- **IN1** to **IN10** (adapter::types::unidirectional::AX)

Output Adapters:

- **OUT** (adapter::types::unidirectional::AX)

Functionality

The function block performs a logical OR operation on the 10 input signals.

Technical Features

- Generic function block with the specific class name 'GEN_AX_OR'
- Uses unidirectional adapters.

State Overview

Combinatory logic block without states.

Application Scenarios

Logical logic gates with many inputs.

⚖️ Comparison with similar building blocks

- **AX_OR_2...9**: Variants with fewer inputs.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

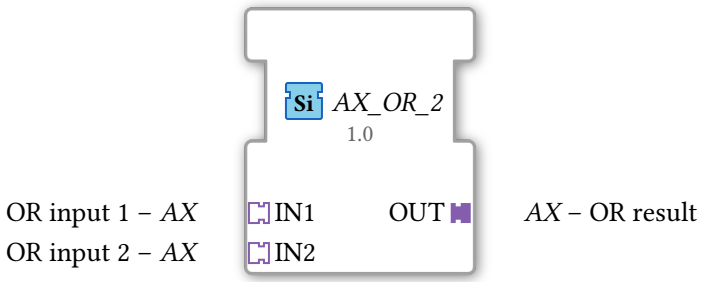
Adapter-based OR gate with 10 inputs.

AX_OR_2

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_2 function block is a generic function block for calculating Boolean OR operations. It processes two input signals and outputs the logical OR result. This function block is specifically designed for use in automation systems.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapter:

- **IN1** - OR Input 1 (Adapter Type: adapter::types::unidirectional::AX)
- **IN2** - OR Input 2 (Adapter Type: adapter::types::unidirectional::AX)

Output Adapter:

- **OUT** - OR Result (Adapter Type: adapter::types::unidirectional::AX)

Functionality

The AX_OR_2 function block performs a logical OR operation between the two input signals IN1 and IN2. The result is output via the output adapter OUT. Processing occurs continuously as soon as input signals are available.

Technical Features

- Generic function block with the class 'GEN_AX_OR'
- Uses unidirectional adapters for signal transmission

- Implemented as part of the “adapter::booleanOperators” package
- Supports the Eclipse 4diac standard specification 61499-2

State Overview

Since it is a combinational logic block, AX_OR_2 has no internal state. The output is determined solely by the current input values.

Application Scenarios

- Safety circuits with redundant sensors
- Parallel control commands
- Monitoring systems with multiple input signals
- Logical operations in control algorithms

⚖️ Comparison with similar components

Compared to other OR components, AX_OR_2 is distinguished by its adapter-based interface, which enables flexible integration into various system architectures. The unidirectional adapters ensure a clear signal flow direction.

Comparison with OR_2

🛠️ Related exercises

- Uebung_002a3_AX
- Uebung_002b3_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_160b_AX

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_OR_2 function block offers a reliable A simple solution for Boolean OR operations in automation systems. Its adapter-based architecture allows for flexible use in various control contexts.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

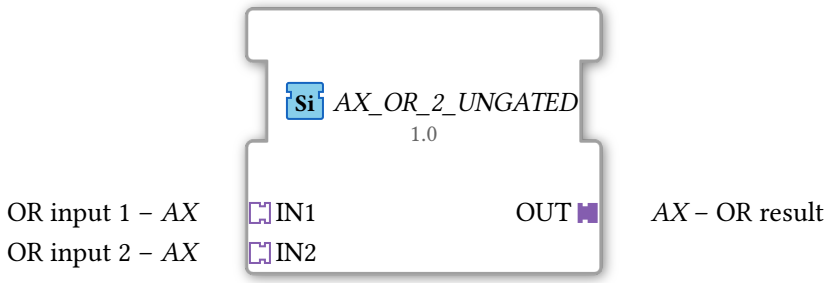
AX_OR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_OR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_2_UNGATED function block is a generic function block for calculating Boolean OR operations. It processes two input signals and outputs the logical OR result. This function block is specifically designed for use in automation systems.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapter:

- **IN1** - OR Input 1 (Adapter Type: adapter::types::unidirectional::AX)
- **IN2** - OR Input 2 (Adapter Type: adapter::types::unidirectional::AX)

Output Adapter:

- **OUT** - OR Result (Adapter Type: adapter::types::unidirectional::AX)

Functionality

The AX_OR_2_UNGATED function block performs a logical OR operation between the two input signals IN1 and IN2. The result is output via the output adapter OUT. Processing occurs continuously as soon as input signals are available.

Technical Features

- Generic function block with the class 'GEN_AX_OR'
- Uses unidirectional adapters for signal transmission
- Implemented as part of the "adapter::booleanOperators" package
- Supports the Eclipse 4diac standard specification 61499-2

State Overview

Since it is a combinational logic block, AX_OR_2_UNGATED has no internal state. The output is determined solely by the current input values.

Application Scenarios

- Safety circuits with redundant sensors
- Parallel control commands
- Monitoring systems with multiple input signals
- Logical operations in control algorithms

⚖️ Comparison with similar components

Compared to other OR components, AX_OR_2_UNGATED is distinguished by its adapter-based interface, which enables flexible integration into various system architectures. The unidirectional adapters ensure a clear signal flow direction.

Comparison with OR_2

🛠️ Related exercises

- Uebung_002a3_AX
- Uebung_002b3_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_160b_AX

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AX_OR_2_UNGATED function block offers a reliable A simple solution for Boolean OR operations in automation systems. Its adapter-based architecture allows for flexible use in various control contexts.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

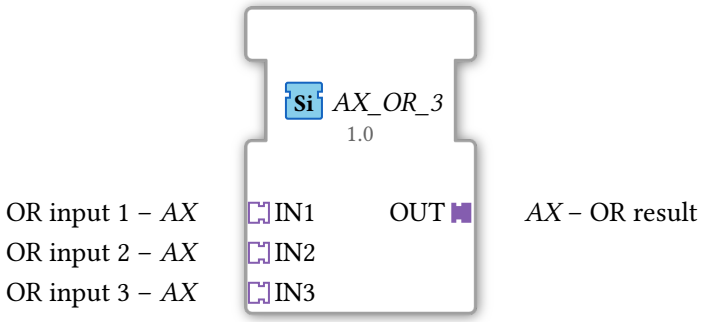
]

AX_OR_3

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_3 function block is a generic function block for calculating the logical OR operation with three inputs. It is used to process Boolean signals in control applications and allows the flexible combination of multiple input signals into a single output signal.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapter

Input Adapter:

- **IN1**: OR Input 1 (Type: adapter::types::unidirectional::AX)
- **IN2**: OR Input 2 (Type: adapter::types::unidirectional::AX)
- **IN3**: OR Input 3 (Type: adapter::types::unidirectional::AX)

Output Adapter:

- **OUT**: OR Result (Type: adapter::types::unidirectional::AX)

Functionality

The function block continuously calculates the logical OR operation of the three input signals. The output signal is true (TRUE) if at least one of the three inputs is true. Only if all three inputs are false (FALSE) will the output also be false.

The logical function can be represented as follows:
OUT = IN1 OR IN2 OR IN3

Technical Features

- Uses unidirectional adapters for signal transmission
- Implemented as a generic function block with the class 'GEN_AX_OR'
- Operates without event control, computs continuously
- Optimized for use in adapter-based architectures

State Overview

Since it is a combinational function block without a memory function, AX_OR_3 has no internal states. The output is determined solely by the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with alternative activation conditions
- Linking status messages from various sources

⚖️ Comparison with Similar Function Blocks

Compared to standard OR blocks, AX_OR_3 offers the advantage of three inputs in a single block, which simplifies wiring. Compared to function blocks with a variable number of inputs, AX_OR_3 is specifically optimized for triple OR operation.

Comparison with OR_3

🛠️ Related Exercises

- Exercise_002a5_AX
- Exercise_002a5b_AX

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_OR_3 function block provides an efficient solution for three-input OR logic operations. Its adapter-based interface allows for flexible integration into various control architectures and makes it particularly suitable for applications where multiple conditions can be met alternatively.

AX_OR_3_UNGATED

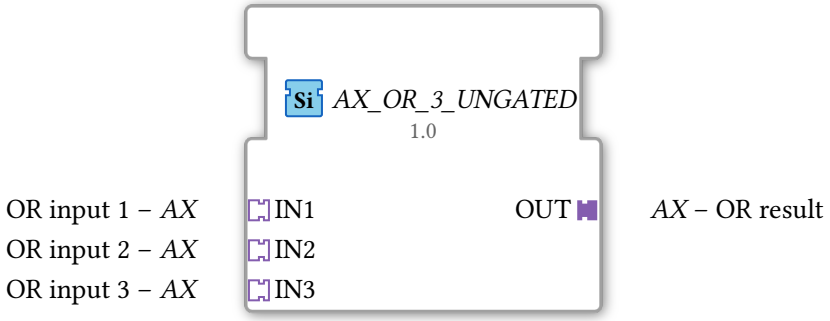
> **i UNGATED variant:** This block is the ungated version of `AX_OR_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/

frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_3_UNGATED function block is a generic function block for calculating the logical OR operation with three inputs. It is used to process Boolean signals in control applications and allows the flexible combination of multiple input signals into a single output signal.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapter

Input Adapter:

- **IN1**: OR Input 1 (Type: adapter::types::unidirectional::AX)
- **IN2**: OR Input 2 (Type: adapter::types::unidirectional::AX)
- **IN3**: OR Input 3 (Type: adapter::types::unidirectional::AX)

Output Adapter:

- **OUT**: OR Result (Type: adapter::types::unidirectional::AX)

Functionality

The function block continuously calculates the logical OR operation of the three input signals. The output signal is true (TRUE) if at least one of the three inputs is true. Only if all three inputs are false (FALSE) will the output also be false.

The logical function can be represented as follows:
OUT = IN1 OR IN2 OR IN3

Technical Features

- Uses unidirectional adapters for signal transmission
- Implemented as a generic function block with the class 'GEN_AX_OR'
- Operates without event control, computs continuously
- Optimized for use in adapter-based architectures

State Overview

Since it is a combinational function block without a memory function, AX_OR_3_UNGATED has no internal states. The output is determined solely by the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with alternative activation conditions
- Linking status messages from various sources

⚖️ Comparison with Similar Function Blocks

Compared to standard OR blocks, AX_OR_3_UNGATED offers the advantage of three inputs in a single block, which simplifies wiring. Compared to function blocks with a variable number of inputs, AX_OR_3_UNGATED is specifically optimized for triple OR operation.

Comparison with OR_3

🛠️ Related Exercises

- Exercise_002a5_AX
- Exercise_002a5b_AX

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

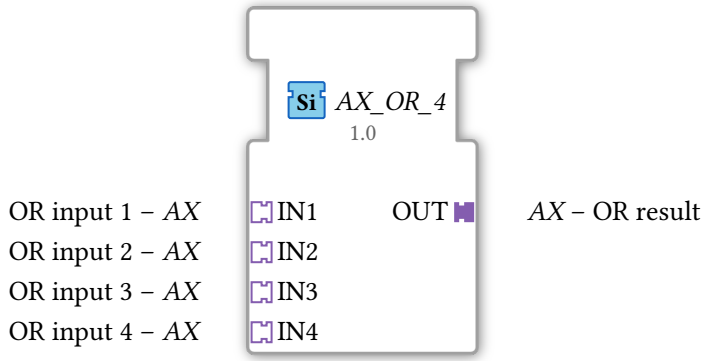
The AX_OR_3_UNGATED function block provides an efficient solution for three-input OR logic operations. Its adapter-based interface allows for flexible integration into various control architectures and makes it particularly suitable for applications where multiple conditions can be met alternatively.

AX_OR_4

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_4 is a generic function block for calculating a four-input logical OR operation. The block processes Boolean signals and outputs the result of the OR operation via an adapter output.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Incoming Adapters (Sockets):

- **IN1**: OR Input 1 (Type: adapter::types::unidirectional::AX)
- **IN2**: OR Input 2 (Type: adapter::types::unidirectional::AX)
- **IN3**: OR Input 3 (Type: adapter::types::unidirectional::AX)
- **IN4**: OR Input 4 (Type: adapter::types::unidirectional::AX)

Outgoing Adapters (Plugs):

- **OUT**: OR Result (Type: adapter::types::unidirectional::AX)

Functionality

This function block continuously calculates the logical OR operation of the four input signals. The result is output via the outgoing adapter OUT. The OR operation returns a logical TRUE if at least one of the four inputs is TRUE. Only if all four inputs are FALSE will the result be FALSE.

Technical Features

- Uses unidirectional adapters for communication
- Implemented as a generic function block with the class 'GEN_AX_OR'
- Works with the AX adapter type for Boolean values
- No event control - operates continuously

State Overview

Since it is a combinational function block without event control, the AX_OR_4 does not have a state machine. The output is calculated continuously based on the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with parallel conditions
- Alarm systems with multiple triggers

⚖️ Comparison with Similar Function Blocks

Compared to simpler OR function blocks, the AX_OR_4 offers the advantage of four inputs in a single block, which simplifies wiring and saves space. Compared to event-driven function blocks, AX_OR_4 operates continuously without explicit trigger events.

Comparison with OR_4

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_OR_4 is an efficient and compact function block for four-input logical OR gates. Its adapter-based interface allows for flexible integration into larger control systems, while its continuous operation ensures an immediate response to input changes.

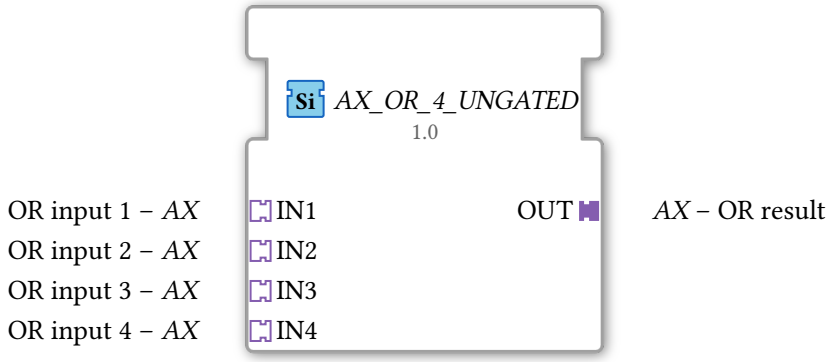
AX_OR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_OR_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_4_UNGATED is a generic function block for calculating a four-input logical OR operation. The block processes Boolean signals and outputs the result of the OR operation via an adapter output.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Incoming Adapters (Sockets):

- **IN1**: OR Input 1 (Type: adapter::types::unidirectional::AX)
- **IN2**: OR Input 2 (Type: adapter::types::unidirectional::AX)
- **IN3**: OR Input 3 (Type: adapter::types::unidirectional::AX)
- **IN4**: OR Input 4 (Type: adapter::types::unidirectional::AX)

Outgoing Adapters (Plugs):

- **OUT**: OR Result (Type: adapter::types::unidirectional::AX)

Functionality

This function block continuously calculates the logical OR operation of the four input signals. The result is output via the outgoing adapter OUT. The OR operation returns a logical TRUE if at least one of the four inputs is TRUE. Only if all four inputs are FALSE will the result be FALSE.

Technical Features

- Uses unidirectional adapters for communication
- Implemented as a generic function block with the class 'GEN_AX_OR'
- Works with the AX adapter type for Boolean values
- No event control - operates continuously

State Overview

Since it is a combinational function block without event control, the AX_OR_4_UNGATED does not have a state machine. The output is calculated continuously based on the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with parallel conditions
- Alarm systems with multiple triggers

⚖️ Comparison with Similar Function Blocks

Compared to simpler OR function blocks, the AX_OR_4_UNGATED offers the advantage of four inputs in a single block, which simplifies wiring and saves space. Compared to event-driven function blocks, AX_OR_4_UNGATED operates continuously without explicit trigger events.

Comparison with OR_4

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

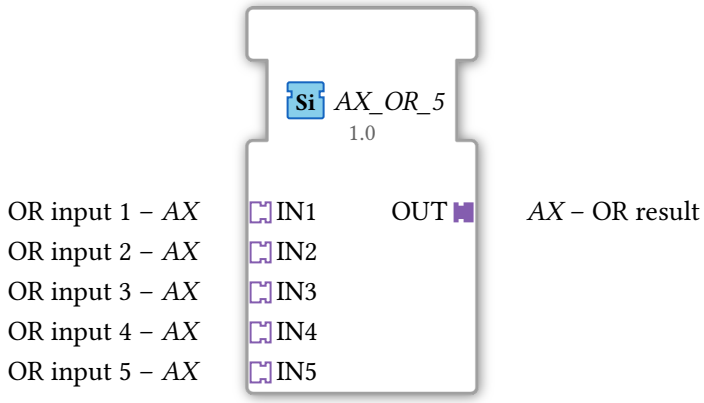
The AX_OR_4_UNGATED is an efficient and compact function block for four-input logical OR gates. Its adapter-based interface allows for flexible integration into larger control systems, while its continuous operation ensures an immediate response to input changes.

AX_OR_5

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_5 function block is a generic block for calculating Boolean OR operations with five inputs. It is used for the logical processing of signals in automation systems and allows the combination of multiple input signals into a single output signal.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Plug Adapter:

- **OUT**: Unidirectional adapter for the OR result

Socket Adapter:

- **IN1**: Unidirectional adapter for OR input 1
- **IN2**: Unidirectional adapter for OR input 2
- **IN3**: Unidirectional adapter for OR input 3
- **IN4**: Unidirectional adapter for OR input 4
- **IN5**: Unidirectional adapter for OR input 5

Functionality

This function block calculates the logical OR operation of the five input signals. The output signal is TRUE if at least one of the five inputs is TRUE. If all inputs are FALSE, the output signal is also FALSE. Processing is performed via adapter interfaces, with all inputs and outputs configured unidirectionally.

Technical Features

- Generic function block with five inputs
- Uses unidirectional adapters for all interfaces
- Implemented as a five-operand OR gate
- No event-driven logic, continuous signal processing

State Overview

The block has no internal state and operates stateless. The output is continuously calculated based on the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with parallel conditions
- Alarm systems with multiple triggers

⚖️ Comparison with Similar Blocks

Compared to standard OR blocks with fewer inputs, AX_OR_5 offers the ability to process five signals simultaneously, simplifying wiring and saving space. Compared to cascaded OR gates, this block provides an integrated solution.

Comparison with OR_5

- `AX_OR_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_OR_5 function block provides an efficient solution for five-input OR logic operations. Using adapters, it allows for flexible integration into various system architectures and is particularly suitable for applications requiring logical combinations of multiple signals.

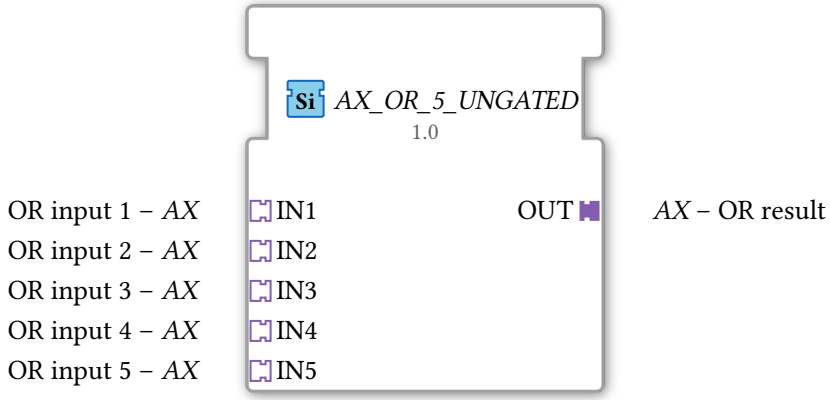
AX_OR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_OR_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_5_UNGATED function block is a generic block for calculating Boolean OR operations with five inputs. It is used for the logical processing of signals in automation systems and allows the combination of multiple input signals into a single output signal.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Plug Adapter:

- **OUT**: Unidirectional adapter for the OR result

Socket Adapter:

- **IN1**: Unidirectional adapter for OR input 1
- **IN2**: Unidirectional adapter for OR input 2
- **IN3**: Unidirectional adapter for OR input 3
- **IN4**: Unidirectional adapter for OR input 4
- **IN5**: Unidirectional adapter for OR input 5

Functionality

This function block calculates the logical OR operation of the five input signals. The output signal is TRUE if at least one of the five inputs is TRUE. If all inputs are FALSE, the output signal is also FALSE. Processing is performed via adapter interfaces, with all inputs and outputs configured unidirectionally.

Technical Features

- Generic function block with five inputs
- Uses unidirectional adapters for all interfaces
- Implemented as a five-operand OR gate
- No event-driven logic, continuous signal processing

State Overview

The block has no internal state and operates stateless. The output is continuously calculated based on the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with parallel conditions
- Alarm systems with multiple triggers

⚖️ Comparison with Similar Blocks

Compared to standard OR blocks with fewer inputs, AX_OR_5_UNGATED offers the ability to process five signals simultaneously, simplifying wiring and saving space. Compared to cascaded OR gates, this block provides an integrated solution.

Comparison with OR_5

- **AX_OR_5**: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

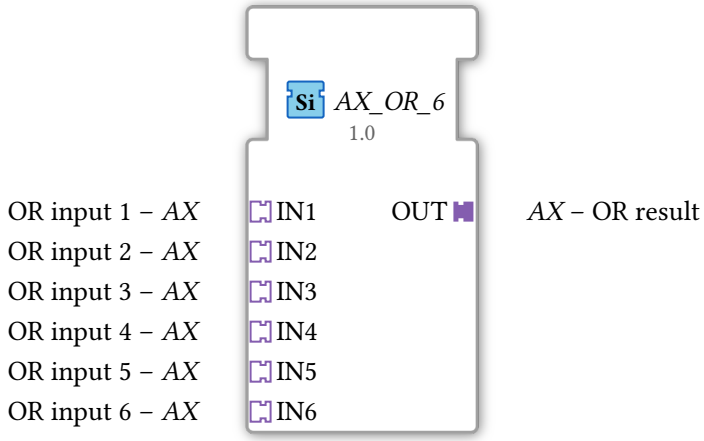
The AX_OR_5_UNGATED function block provides an efficient solution for five-input OR logic operations. Using adapters, it allows for flexible integration into various system architectures and is particularly suitable for applications requiring logical combinations of multiple signals.

AX_OR_6

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_6 is a generic function block for calculating a logical OR operation with six inputs. This block is used to process Boolean signals in automation systems and outputs the result of the OR operation via an adapter output.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

Input Adapters:

- **IN1** - OR Input 1 (Type: adapter::types::unidirectional::AX)
- **IN2** - OR Input 2 (Type: adapter::types::unidirectional::AX)
- **IN3** - OR Input 3 (Type: adapter::types::unidirectional::AX)
- **IN4** - OR Input 4 (Type: adapter::types::unidirectional::AX)
- **IN5** - OR Input 5 (Type: adapter::types::unidirectional::AX)
- **IN6** - OR input 6 (Type: adapter::types::unidirectional::AX)

Output adapter:

- **OUT** - OR result (Type: adapter::types::unidirectional::AX)

Functionality

The function block continuously calculates the logical OR operation of all six input signals. The output signal is TRUE if at least one of the six inputs is TRUE. Only if all six inputs are FALSE will the output also be FALSE.

Technical Features

- Generic function block with the class 'GEN_AX_OR'
- Uses unidirectional AX adapters for signal transmission
- Six independent inputs for flexible applications
- Real-time processing without event control

State Overview

The function block has no internal state and operates stateless. The output is continuously calculated based on the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with alternative activation conditions
- Linking multiple status messages

⚖️ Comparison with similar components

Compared to simpler OR components with fewer inputs, the AX_OR_6 offers greater flexibility thanks to its six separate inputs. The exclusive use of adapters instead of conventional data inputs/outputs enables a modular system architecture.

Comparison with OR_6

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_OR_6 is a specialized logic component for applications requiring a multi-input OR gate. Its adapter-based interface makes it particularly suitable for modular system designs and allows for clear separation of signal paths.

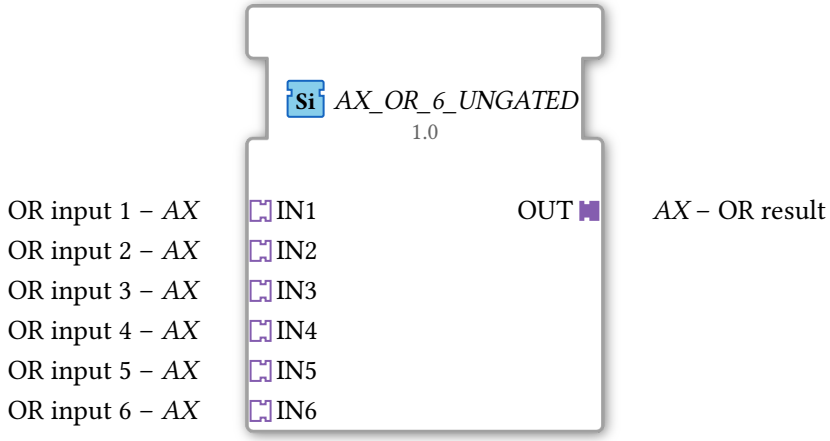
AX_OR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_OR_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_6_UNGATED is a generic function block for calculating a logical OR operation with six inputs. This block is used to process Boolean signals in automation systems and outputs the result of the OR operation via an adapter output.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

Input Adapters:

- **IN1** - OR Input 1 (Type: adapter::types::unidirectional::AX)
- **IN2** - OR Input 2 (Type: adapter::types::unidirectional::AX)
- **IN3** - OR Input 3 (Type: adapter::types::unidirectional::AX)
- **IN4** - OR Input 4 (Type: adapter::types::unidirectional::AX)
- **IN5** - OR Input 5 (Type: adapter::types::unidirectional::AX)
- **IN6** - OR input 6 (Type: adapter::types::unidirectional::AX)

Output adapter:

- **OUT** - OR result (Type: adapter::types::unidirectional::AX)

Functionality

The function block continuously calculates the logical OR operation of all six input signals. The output signal is TRUE if at least one of the six inputs is TRUE. Only if all six inputs are FALSE will the output also be FALSE.

Technical Features

- Generic function block with the class ‘GEN_AX_OR’
- Uses unidirectional AX adapters for signal transmission
- Six independent inputs for flexible applications
- Real-time processing without event control

State Overview

The function block has no internal state and operates stateless. The output is continuously calculated based on the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with alternative activation conditions
- Linking multiple status messages

⚖️ Comparison with similar components

Compared to simpler OR components with fewer inputs, the AX_OR_6_UNGATED offers greater flexibility thanks to its six separate inputs. The exclusive use of adapters instead of conventional data inputs/outputs enables a modular system architecture.

Comparison with OR_6

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

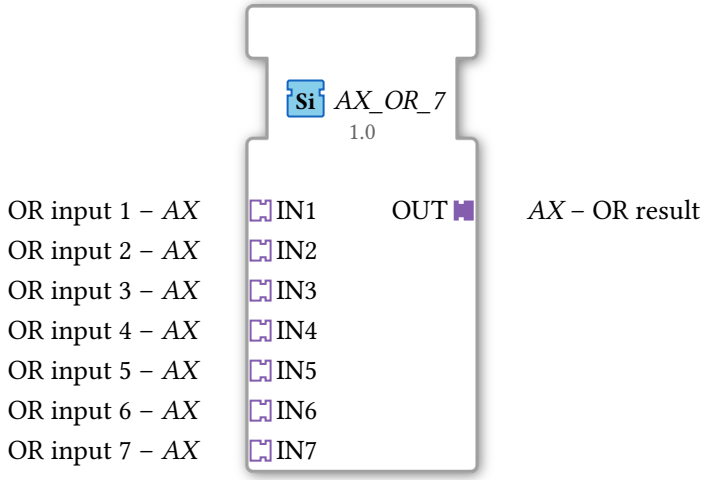
The AX_OR_6_UNGATED is a specialized logic component for applications requiring a multi-input OR gate. Its adapter-based interface makes it particularly suitable for modular system designs and allows for clear separation of signal paths.

AX_OR_7

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_7 function block is a generic function block for calculating Boolean OR operations. It has seven inputs and calculates the logical OR of all input signals.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Input Adapter:

- **IN1** - OR Input 1
- **IN2** - OR Input 2
- **IN3** - OR Input 3
- **IN4** - OR Input 4
- **IN5** - OR Input 5
- **IN6** - OR Input 6
- **IN7** - OR Input 7

Output Adapter:

- **OUT** - OR Result

All adapters use the type `adapter::types::unidirectional::AX`.

Functionality

The function block continuously calculates the Boolean OR operation of all seven input signals. The result is output via the output adapter OUT. The output signal is TRUE if at least one of the seven inputs is TRUE.

Technical Features

- Generic function block with the class 'GEN_AX_OR'
- Uses unidirectional AX adapters for all interfaces
- No event control - operates continuously
- Seven independent inputs for maximum flexibility

State Overview

Since it is a purely combinational function block without event control, the AX_OR_7 does not have a state machine. The output is calculated directly from the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Multiple operator stations in machine controls
- Monitoring systems with multiple sensors
- Linking multiple enable signals

⚖️ Comparison with Similar Function Blocks

Compared to standard OR blocks, the AX_OR_7 offers the unique feature of seven inputs, which simplifies wiring because fewer blocks need to be connected in series. The use of adapters instead of direct data connections allows for more flexible integration into various system architectures.

Comparison with OR_7

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_OR_7 is a specialized OR gate for applications with many input signals. With its seven inputs and adapter-based interface, it is particularly suitable for complex control tasks where multiple signals need to be logically combined.

AX_OR_7_UNGATED

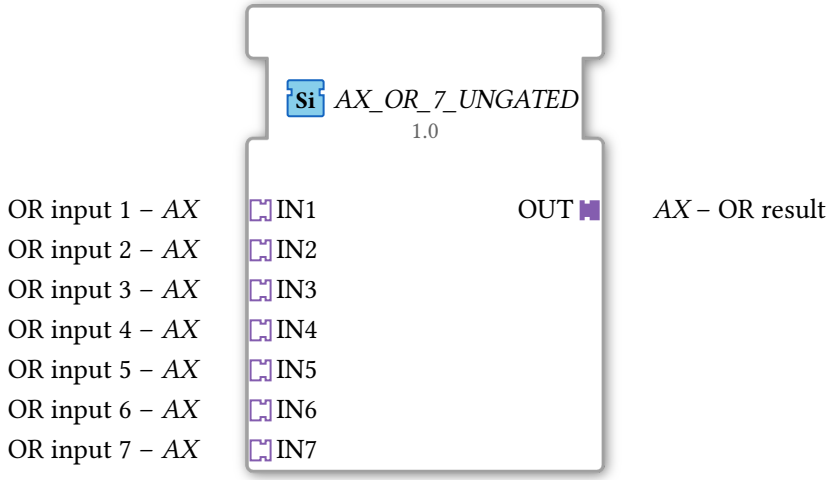
> **i UNGATED variant:** This block is the ungated version of `AX_OR_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change.

This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_7_UNGATED function block is a generic function block for calculating Boolean OR operations. It has seven inputs and calculates the logical OR of all input signals.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Input Adapter:

- **IN1** - OR Input 1
- **IN2** - OR Input 2
- **IN3** - OR Input 3
- **IN4** - OR Input 4
- **IN5** - OR Input 5
- **IN6** - OR Input 6
- **IN7** - OR Input 7

Output Adapter:

- **OUT** - OR Result

All adapters use the type `adapter::types::unidirectional::AX`.

Functionality

The function block continuously calculates the Boolean OR operation of all seven input signals. The result is output via the output adapter OUT. The output signal is TRUE if at least one of the seven inputs is TRUE.

Technical Features

- Generic function block with the class ‘GEN_AX_OR’
- Uses unidirectional AX adapters for all interfaces
- No event control - operates continuously
- Seven independent inputs for maximum flexibility

State Overview

Since it is a purely combinational function block without event control, the AX_OR_7_UNGATED does not have a state machine. The output is calculated directly from the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Multiple operator stations in machine controls
- Monitoring systems with multiple sensors
- Linking multiple enable signals

⚖ Comparison with Similar Function Blocks

Compared to standard OR blocks, the AX_OR_7_UNGATED offers the unique feature of seven inputs, which simplifies wiring because fewer blocks need to be connected in series. The use of adapters instead of direct data connections allows for more flexible integration into various system architectures.

Comparison with OR_7

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

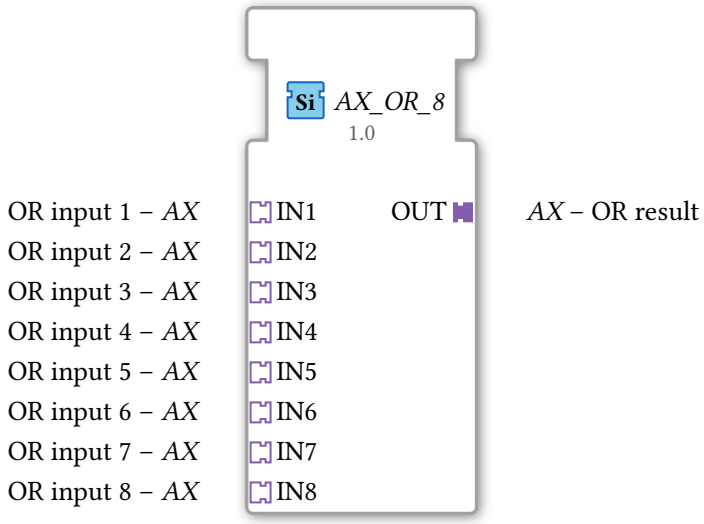
The AX_OR_7_UNGATED is a specialized OR gate for applications with many input signals. With its seven inputs and adapter-based interface, it is particularly suitable for complex control tasks where multiple signals need to be logically combined.

AX_OR_8

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_8 function block is a generic function block for calculating an eight-input logical OR operation. It is used to process Boolean signals in control applications and allows the combination of multiple input signals into a single output signal.



Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapters:

- **IN1** - OR input 1 (unidirectional AX adapter)
- **IN2** - OR input 2 (unidirectional AX adapter)
- **IN3** - OR input 3 (unidirectional AX adapter)
- **IN4** - OR input 4 (unidirectional AX adapter)
- **IN5** - OR input 5 (unidirectional AX adapter)
- **IN6** - OR input 6 (unidirectional AX adapter) AX Adapter
- **IN7** - OR input 7 (unidirectional AX adapter)
- **IN8** - OR input 8 (unidirectional AX adapter)

Output Adapter:

- **OUT** - OR result (unidirectional AX adapter)

Functionality
The AX_OR_8 block performs a logical OR operation on all eight inputs. The output signal is set to TRUE if at least one of the eight inputs is TRUE. Only if all inputs are FALSE will the output also be FALSE.

The functionality corresponds to the Boolean equation:
OUT = IN1 ∨ IN2 ∨ IN3 ∨ IN4 ∨ IN5 ∨ IN6 ∨ IN7 ∨ IN8

Technical Features
- Generic function block with the class name identifier ‘GEN_AX_OR’
- Uses unidirectional AX adapters for all interfaces
- No internal states or memory functions
- Immediate processing without delay
- Part of the “adapter::booleanOperators” package

State Overview
The function block is stateless and performs the OR operation immediately upon any change to the inputs. No internal states are stored.

Application Scenarios
- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with parallel conditions
- Alarm systems with multiple triggers
- Linking multiple enable signals

⚖️ Comparison with similar function blocks
Compared to simple OR function blocks with fewer inputs, AX_OR_8 offers the ability to directly link up to eight signals without additional chaining of multiple function blocks. The use of adapters instead of direct data connections enables more flexible integration into various system architectures.

Comparison with OR_8

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_OR_8 function block represents an efficient solution for complex OR logic operations with up to eight inputs. Due to the use of standard adapters and its generic implementation, it is particularly suitable for reusable control logic in industrial automation applications.

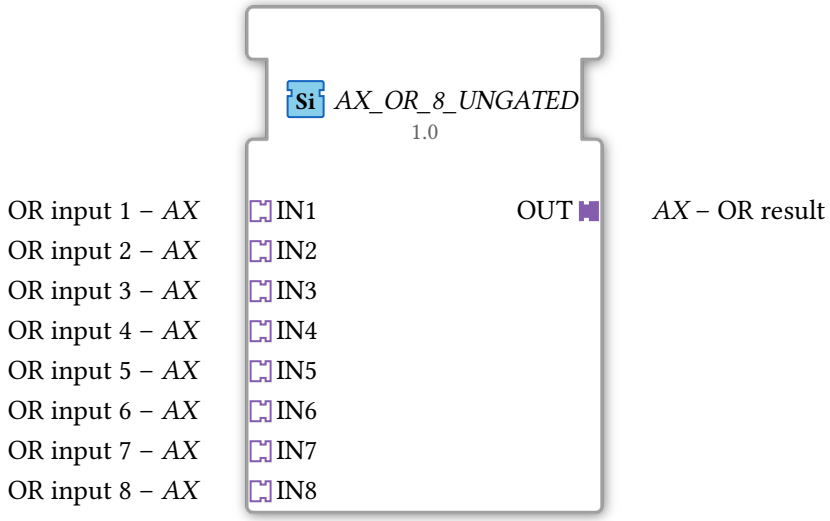
AX_OR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_OR_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_8_UNGATED function block is a generic function block for calculating an eight-input logical OR operation. It is used to process Boolean signals in control applications and allows the combination of multiple input signals into a single output signal.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN1** - OR input 1 (unidirectional AX adapter)
- **IN2** - OR input 2 (unidirectional AX adapter)
- **IN3** - OR input 3 (unidirectional AX adapter)
- **IN4** - OR input 4 (unidirectional AX adapter)
- **IN5** - OR input 5 (unidirectional AX adapter)
- **IN6** - OR input 6 (unidirectional AX adapter) AX Adapter
- **IN7** - OR input 7 (unidirectional AX adapter)
- **IN8** - OR input 8 (unidirectional AX adapter)

Output Adapter:

- **OUT** - OR result (unidirectional AX adapter)

Functionality

The AX_OR_8_UNGATED block performs a logical OR operation on all eight inputs. The output signal is set to TRUE if at least one of the eight inputs is TRUE. Only if all inputs are FALSE will the output also be FALSE.

The functionality corresponds to the Boolean equation:
OUT = IN1 ∨ IN2 ∨ IN3 ∨ IN4 ∨ IN5 ∨ IN6 ∨ IN7 ∨ IN8

Technical Features

- Generic function block with the class name identifier ‘GEN_AX_OR’
- Uses unidirectional AX adapters for all interfaces
- No internal states or memory functions
- Immediate processing without delay
- Part of the “adapter::booleanOperators” package

State Overview

The function block is stateless and performs the OR operation immediately upon any change to the inputs. No internal states are stored.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with parallel conditions
- Alarm systems with multiple triggers
- Linking multiple enable signals

⚖️ Comparison with similar function blocks

Compared to simple OR function blocks with fewer inputs, AX_OR_8_UNGATED offers the ability to directly link up to eight signals without additional chaining of multiple function blocks. The use of adapters instead of direct data connections enables more flexible integration into various system architectures.

Comparison with OR_8

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

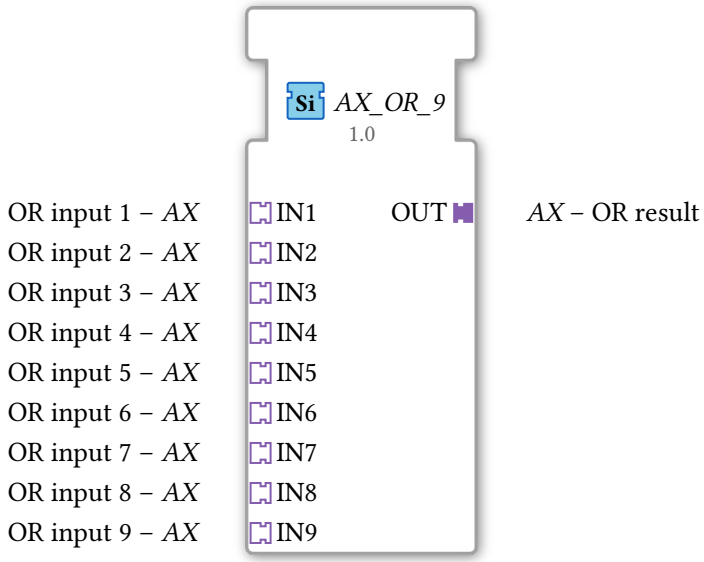
The AX_OR_8_UNGATED function block represents an efficient solution for complex OR logic operations with up to eight inputs. Due to the use of standard adapters and its generic implementation, it is particularly suitable for reusable control logic in industrial automation applications.

AX_OR_9

• − ◦ • − ◦ • − ◦ •

Introduction

The AX_OR_9 is a generic function block for calculating the Boolean OR operation with nine inputs. This block is used for the logical processing of signals in automation systems and outputs the result of the OR operation on all input signals.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Sockets (Inputs):

- **IN1** - OR Input 1
- **IN2** - OR Input 2
- **IN3** - OR Input 3
- **IN4** - OR Input 4
- **IN5** - OR Input 5
- **IN6** - OR Input 6
- **IN7** - OR Input 7
- **IN8** - OR Input 8
- **IN9** - OR Input 9

Plugs (Outputs):

- **OUT** - OR Result

Functionality

The function block continuously calculates the Boolean OR operation of all nine input signals. The output signal is TRUE (1) if at least one of the nine inputs is TRUE (1). Only if all inputs are FALSE (0) will the output also be FALSE (0).

Technical Features

- Generic function block with a fixed number of 9 inputs
- Uses unidirectional AX adapters for inputs and outputs
- No event-driven control - operates continuously
- Implemented as a generic function block with a specific class name

State Overview

The function block has no internal state and operates stateless. The output is calculated solely based on the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with redundant input signals
- Alarm systems with multiple triggers

⚖️ Comparison with similar function blocks

Compared to standard OR function blocks with fewer inputs, AX_OR_9 offers the ability to process up to nine signals simultaneously, simplifying the wiring of more complex logic. Unlike configurable OR function blocks, this block has a fixed number of inputs, simplifying configuration for known requirements.

Comparison with OR_9

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_OR_9 is a specialized OR function block for applications that need to logically combine exactly nine input signals. Its fixed number of inputs and simple operation make it particularly suitable for clearly defined control tasks with redundant input signals.

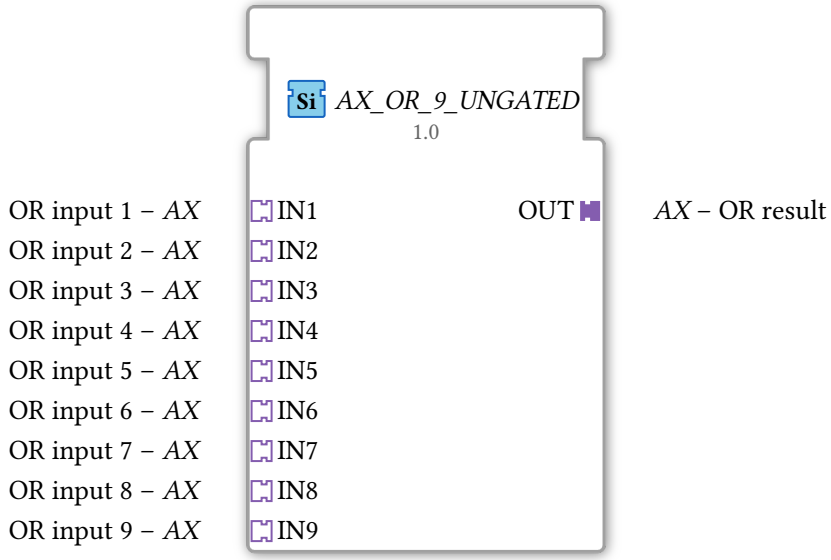
AX_OR_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_OR_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_OR_9_UNGATED is a generic function block for calculating the Boolean OR operation with nine inputs. This block is used for the logical processing of signals in automation systems and outputs the result of the OR operation on all input signals.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Sockets (Inputs):

- **IN1** - OR Input 1
- **IN2** - OR Input 2
- **IN3** - OR Input 3
- **IN4** - OR Input 4
- **IN5** - OR Input 5
- **IN6** - OR Input 6
- **IN7** - OR Input 7
- **IN8** - OR Input 8
- **IN9** - OR Input 9

Plugs (Outputs):

- **OUT** - OR Result

Functionality

The function block continuously calculates the Boolean OR operation of all nine input signals. The output signal is TRUE (1) if at least one of the nine inputs is TRUE (1). Only if all inputs are FALSE (0) will the output also be FALSE (0).

Technical Features

- Generic function block with a fixed number of 9 inputs
- Uses unidirectional AX adapters for inputs and outputs
- No event-driven control - operates continuously
- Implemented as a generic function block with a specific class name

State Overview

The function block has no internal state and operates stateless. The output is calculated solely based on the current input values.

Application Scenarios

- Safety circuits with multiple emergency stop buttons
- Monitoring systems with multiple sensors
- Control logic with redundant input signals
- Alarm systems with multiple triggers

⚖️ Comparison with similar function blocks

Compared to standard OR function blocks with fewer inputs, AX_OR_9_UNGATED offers the ability to process up to nine signals simultaneously, simplifying the wiring of more complex logic. Unlike configurable OR function blocks, this block has a fixed number of inputs, simplifying configuration for known requirements.

Comparison with OR_9

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

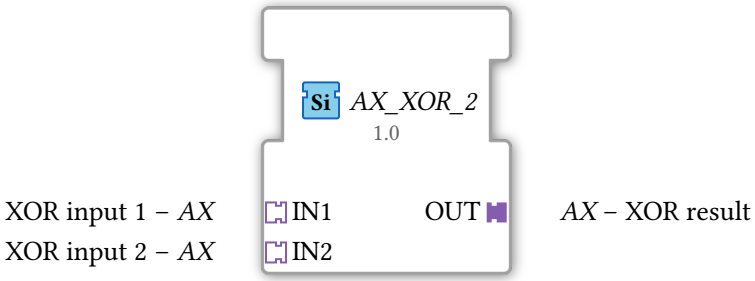
The AX_OR_9_UNGATED is a specialized OR function block for applications that need to logically combine exactly nine input signals. Its fixed number of inputs and simple operation make it particularly suitable for clearly defined control tasks with redundant input signals.

AX_XOR_2

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_2 function block is a generic function block for calculating the Boolean XOR operation (exclusive OR). It processes two input signals and outputs the logical XOR result.



Interface Structure

Event Inputs

No direct event inputs are available – processing is handled via adapter interfaces.

Event Outputs

No direct event outputs are available – output is handled via adapter interfaces.

Data Inputs

No direct data inputs are available – input is handled via adapter interfaces.

Data Outputs

No direct data outputs are available – output is handled via adapter interfaces.

Adapter

- **IN1** (Socket): XOR Input 1 - Receives the first input signal
- **IN2** (Socket): XOR Input 2 - Receives the second input signal
- **OUT** (Plug): XOR Result - Outputs the calculated XOR result

All adapters use the type `adapter::types::unidirectional::AX` for unidirectional communication.

Functionality

The function block continuously calculates the XOR operation between the two input signals IN1 and IN2. The XOR function outputs a true signal (TRUE) if and only if exactly one of the two inputs is true (TRUE). If both inputs are the same (both TRUE or both FALSE), a false signal (FALSE) is output.

Technical Features

- Generic function block with the class `GEN_AX_XOR`
- Uses unidirectional adapters for inputs and outputs
- Implemented according to the IEC 61499-2 standard
- Part of the `adapter::booleanOperators` package

State Overview

The function block has a continuous operating state and constantly calculates the XOR result based on the current input values.

Application Scenarios

- Logic circuits in control systems
- Comparison of two binary signals
- Detection of signal differences
- Safety-related circuits
- Digital signal processing

⚖️ Comparison with similar function blocks

Compared to other logical operators such as AND or OR, XOR has the unique feature that a TRUE signal is only generated when the input values are different. This makes it particularly suitable for comparison and difference detection tasks.

Comparison with XOR_2

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_XOR_2 function block provides an efficient and reliable solution for XOR operations in 4diac-based control systems. Its use of adapter interfaces allows for flexible integration into various system architectures and makes it particularly suitable for applications where signal differences need to be detected.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

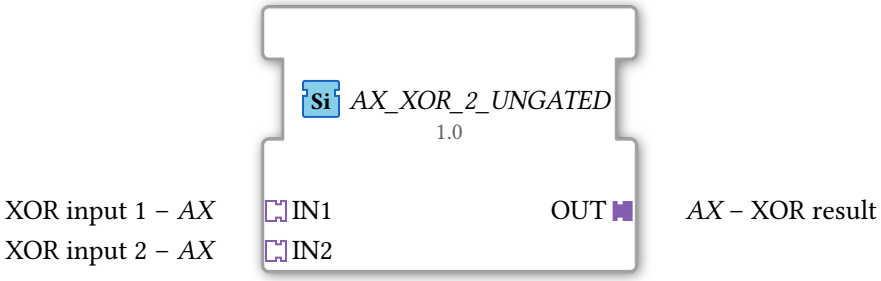
AX_XOR_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_XOR_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_2_UNGATED function block is a generic function block for calculating the Boolean XOR operation (exclusive OR). It processes two input signals and outputs the logical XOR result.



Interface Structure

Event Inputs

No direct event inputs are available – processing is handled via adapter interfaces.

Event Outputs

No direct event outputs are available – output is handled via adapter interfaces.

Data Inputs

No direct data inputs are available – input is handled via adapter interfaces.

Data Outputs

No direct data outputs are available – output is handled via adapter interfaces.

Adapter

- **IN1** (Socket): XOR Input 1 - Receives the first input signal
- **IN2** (Socket): XOR Input 2 - Receives the second input signal
- **OUT** (Plug): XOR Result - Outputs the calculated XOR result

All adapters use the type `adapter::types::unidirectional::AX` for unidirectional communication.

Functionality

The function block continuously calculates the XOR operation between the two input signals IN1 and IN2. The XOR function outputs a true signal (TRUE) if and only if exactly one of the two inputs is true (TRUE). If both inputs are the same (both TRUE or both FALSE), a false signal (FALSE) is output.

Technical Features

- Generic function block with the class `GEN_AX_XOR`
- Uses unidirectional adapters for inputs and outputs
- Implemented according to the IEC 61499-2 standard
- Part of the `adapter::booleanOperators` package

State Overview

The function block has a continuous operating state and constantly calculates the XOR result based on the current input values.

Application Scenarios

- Logic circuits in control systems
- Comparison of two binary signals
- Detection of signal differences
- Safety-related circuits
- Digital signal processing

⚖️ Comparison with similar function blocks

Compared to other logical operators such as AND or OR, XOR has the unique feature that a TRUE signal is only generated when the input values are different. This makes it particularly suitable for comparison and difference detection tasks.

Comparison with XOR_2

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AX_XOR_2_UNGATED function block provides an efficient and reliable solution for XOR operations in 4diac-based control systems. Its use of adapter interfaces allows for flexible integration into various system architectures and makes it particularly suitable for applications where signal differences need to be detected.

🌐 Related topic subpages on ms-muc-docs.de

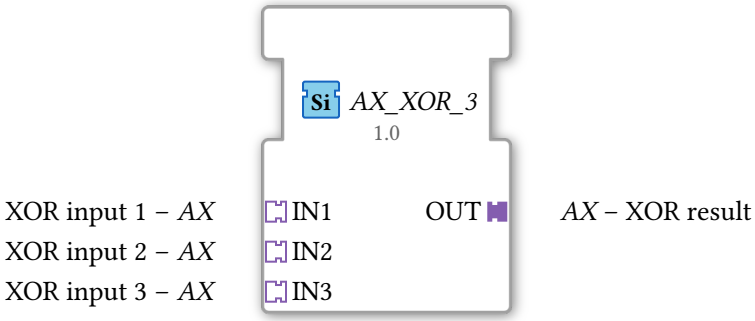
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AX_XOR_3

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_3 function block is a generic function block for calculating the Boolean XOR operation with three inputs. It implements the exclusive OR operation for three binary input signals and outputs the corresponding result.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Sockets (Inputs):

- **IN1**: XOR input 1 (Adapter type: unidirectional::AX)
- **IN2**: XOR input 2 (Adapter type: unidirectional::AX)
- **IN3**: XOR input 3 (Adapter type: unidirectional::AX)

Plugs (Outputs):

- **OUT**: XOR result (Adapter type: unidirectional::AX)

Functionality

This function block calculates the XOR operation for three input signals. The XOR operation outputs a "true" signal if and only if an odd number of the inputs are "true". With three inputs, this means:

- The result is true if exactly one or all three inputs are true.
- The result is false if none or exactly two inputs are true.

Technical Features

- Generic function block with the class name identifier 'GEN_AX_XOR'
- Uses unidirectional adapters for communication
- Implemented according to the IEC 61499-2 standard
- No internal states or storage required

State Overview

The function block has no internal states and operates statically. The output is calculated solely based on the current input values.

Application Scenarios

- Three-input switching networks
- Parity checks
- Safety-critical controllers where odd-numbered activation is required
- Logical operations in automation systems

⚖️ Comparison with similar function blocks

Compared to standard two-input XOR blocks, AX_XOR_3 offers the extension to three inputs. While a two-input XOR only returns true if the inputs are unequal, the three-input XOR takes the parity of all three signals into account.

Comparison with XOR_3

🛠️ Related Exercises

- Exercise_002a7_AX

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_XOR_3 function block provides a reliable and efficient solution for triple XOR operations in IEC 61499-based control systems. By using adapters, it enables flexible integration into different system architectures and is particularly suitable for applications that require parity checking or special logical conditions with three input signals.

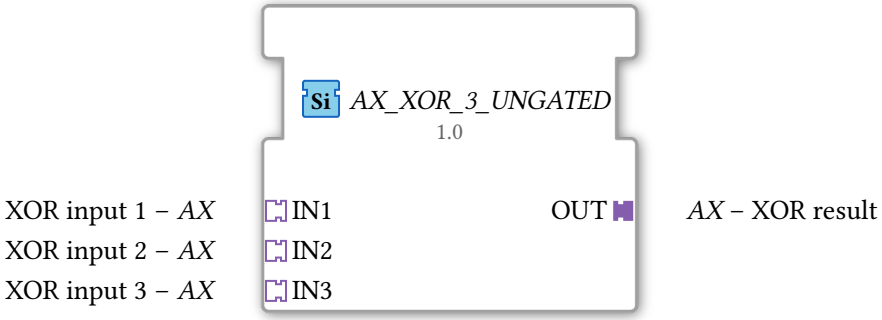
AX_XOR_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_XOR_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_3_UNGATED function block is a generic function block for calculating the Boolean XOR operation with three inputs. It implements the exclusive OR operation for three binary input signals and outputs the corresponding result.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Sockets (Inputs):

- **IN1**: XOR input 1 (Adapter type: unidirectional::AX)
- **IN2**: XOR input 2 (Adapter type: unidirectional::AX)
- **IN3**: XOR input 3 (Adapter type: unidirectional::AX)

Plugs (Outputs):

- **OUT**: XOR result (Adapter type: unidirectional::AX)

Functionality

This function block calculates the XOR operation for three input signals. The XOR operation outputs a “true” signal if and only if an odd number of the inputs are “true”. With three inputs, this means:

- The result is true if exactly one or all three inputs are true.
- The result is false if none or exactly two inputs are true.

Technical Features

- Generic function block with the class name identifier ‘GEN_AX_XOR’
- Uses unidirectional adapters for communication
- Implemented according to the IEC 61499-2 standard
- No internal states or storage required

State Overview

The function block has no internal states and operates statically. The output is calculated solely based on the current input values.

Application Scenarios

- Three-input switching networks
- Parity checks

- Safety-critical controllers where odd-numbered activation is required
- Logical operations in automation systems

⚖️ Comparison with similar function blocks

Compared to standard two-input XOR blocks, AX_XOR_3_UNGATED offers the extension to three inputs. While a two-input XOR only returns true if the inputs are unequal, the three-input XOR takes the parity of all three signals into account.

Comparison with XOR_3

🛠️ Related Exercises

- Exercise_002a7_AX

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

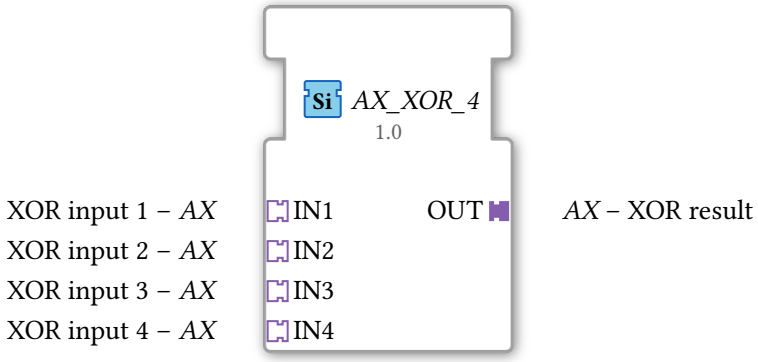
The AX_XOR_3_UNGATED function block provides a reliable and efficient solution for triple XOR operations in IEC 61499-based control systems. By using adapters, it enables flexible integration into different system architectures and is particularly suitable for applications that require parity checking or special logical conditions with three input signals.

AX_XOR_4

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_4 function block is a generic function block for calculating the Boolean XOR operation with four inputs. It implements the exclusive OR operation for up to four different input signals.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Plug Adapter:

- **OUT**: Unidirectional adapter for the XOR result

Socket Adapter:

- **IN1**: Unidirectional adapter for XOR input 1
- **IN2**: Unidirectional adapter for XOR input 2
- **IN3**: Unidirectional adapter for XOR input 3
- **IN4**: Unidirectional adapter for XOR input 4

Functionality

The function block calculates the XOR operation across four inputs. The XOR operation returns a "true" signal if and only if an odd number of inputs are active. With four inputs, this means:

- Result = 1 if 1 or 3 inputs are active
- Result = 0 if 0, 2, or 4 inputs are active

Communication occurs exclusively via the defined adapter interfaces.

Technical Features

- Generic function block with the class 'GEN_AX_XOR'
- Uses unidirectional adapters for all inputs and outputs
- No direct event or data interfaces
- Fully adapter-based architecture

State Overview

Since it is a combinational logic block, the AX_XOR_4 has no internal states. The output is determined solely by the current input values.

Application Scenarios

- Parity check in control systems
- Odd parity check
- Safety-critical circuits with multiple inputs
- Distributed logic operations in automation systems

⚖️ Comparison with similar function blocks

Compared to standard XOR function blocks, AX_XOR_4 offers:

- Four inputs instead of the typical two inputs
- A purely adapter-based interface
- Specialization for unidirectional AX adapters
- No event-driven control, but continuous operation

Comparison with XOR_4

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_XOR_4 function block represents a specialized solution for four-input XOR operations in adapter-based system architectures. Its purely adapter-based interface makes it particularly suitable for modular system designs where communication takes place via standardized adapters.

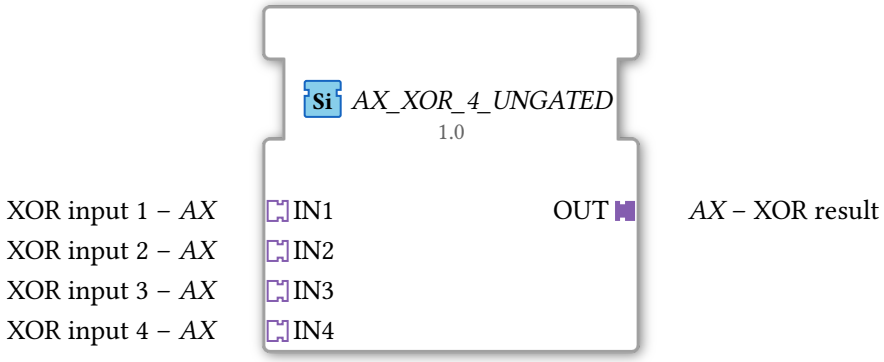
AX_XOR_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_XOR_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_4_UNGATED function block is a generic function block for calculating the Boolean XOR operation with four inputs. It implements the exclusive OR operation for up to four different input signals.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Plug Adapter:

- **OUT**: Unidirectional adapter for the XOR result

Socket Adapter:

- **IN1**: Unidirectional adapter for XOR input 1
- **IN2**: Unidirectional adapter for XOR input 2
- **IN3**: Unidirectional adapter for XOR input 3
- **IN4**: Unidirectional adapter for XOR input 4

Functionality

The function block calculates the XOR operation across four inputs. The XOR operation returns a “true” signal if and only if an odd number of inputs are active. With four inputs, this means:

- Result = 1 if 1 or 3 inputs are active
- Result = 0 if 0, 2, or 4 inputs are active

Communication occurs exclusively via the defined adapter interfaces.

Technical Features

- Generic function block with the class ‘GEN_AX_XOR’
- Uses unidirectional adapters for all inputs and outputs
- No direct event or data interfaces
- Fully adapter-based architecture

State Overview

Since it is a combinational logic block, the AX_XOR_4_UNGATED has no internal states. The output is determined solely by the current input values.

Application Scenarios

- Parity check in control systems
- Odd parity check
- Safety-critical circuits with multiple inputs
- Distributed logic operations in automation systems

⚖️ Comparison with similar function blocks

Compared to standard XOR function blocks, AX_XOR_4_UNGATED offers:

- Four inputs instead of the typical two inputs
- A purely adapter-based interface
- Specialization for unidirectional AX adapters
- No event-driven control, but continuous operation

Comparison with XOR_4

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

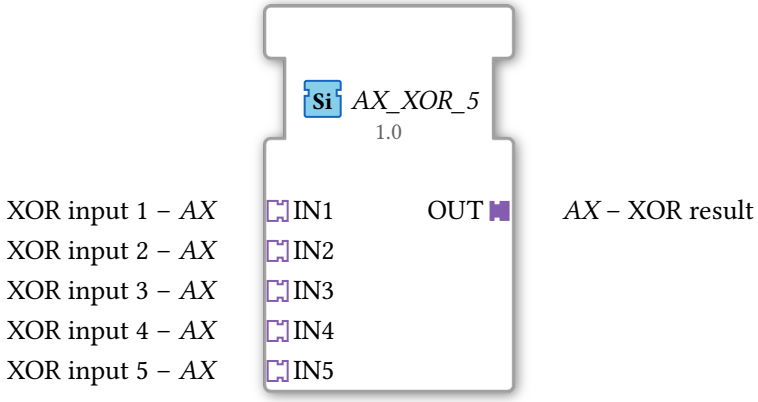
The AX_XOR_4_UNGATED function block represents a specialized solution for four-input XOR operations in adapter-based system architectures. Its purely adapter-based interface makes it particularly suitable for modular system designs where communication takes place via standardized adapters.

AX_XOR_5

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_5 is a generic function block for calculating the Boolean XOR operation with five inputs. The block implements the exclusive OR operation for up to five input signals and outputs the result via an adapter output.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Input Adapters:

- **IN1**: XOR Input 1 (unidirectional AX adapter)
- **IN2**: XOR Input 2 (unidirectional AX adapter)
- **IN3**: XOR Input 3 (unidirectional AX adapter)
- **IN4**: XOR Input 4 (unidirectional AX adapter)
- **IN5**: XOR Input 5 (unidirectional AX adapter)

Output Adapters:

- **OUT**: XOR Result (unidirectional AX adapter)

Functionality

This function block calculates the XOR operation across all five inputs. The XOR operation (exclusive OR) returns a "true" signal if and only if an odd number of inputs are active. With an even number of active inputs, it returns "false".

The mathematical formula is:
OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5

Technical Features

- Uses unidirectional AX adapters for all interfaces
- Implemented as a generic function block named 'GEN_AX_XOR'
- Supports up to five independent input signals
- No event control - operates continuously based on the adapter inputs

State Overview

Since it is a combinational logic block without memory, the AX_XOR_5 has no internal states. The output is determined solely by the current input values.

Application Scenarios

- Parity checking in digital circuits
- Fault detection in safety systems
- Control logic with multiple conditions
- Distributed decision-making in automation systems
- Safety-critical applications with redundant sensors

⚖️ Comparison with Similar Blocks

Compared to standard XOR blocks with fewer inputs, the AX_XOR_5 offers the ability to process up to five signals simultaneously. While simple XOR blocks typically have only two inputs, this function block enables more complex logical operations without additional nesting.

Comparison with XOR_5

- `AX_XOR_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_XOR_5 function block provides a flexible solution for multi-input XOR operations. Its adapter-based interface allows for easy integration into existing control systems, while its generic implementation ensures reusability and extensibility.

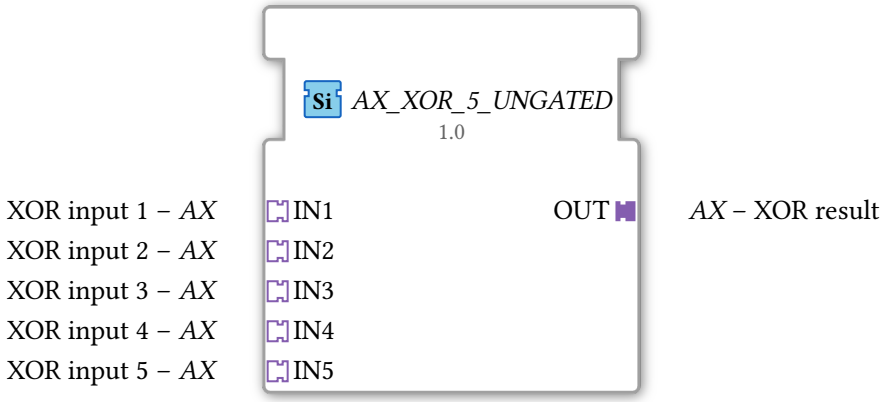
AX_XOR_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of AX_XOR_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_5_UNGATED is a generic function block for calculating the Boolean XOR operation with five inputs. The block implements the exclusive OR operation for up to five input signals and outputs the result via an adapter output.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Input Adapters:

- **IN1**: XOR Input 1 (unidirectional AX adapter)
- **IN2**: XOR Input 2 (unidirectional AX adapter)
- **IN3**: XOR Input 3 (unidirectional AX adapter)
- **IN4**: XOR Input 4 (unidirectional AX adapter)
- **IN5**: XOR Input 5 (unidirectional AX adapter)

Output Adapters:

- **OUT**: XOR Result (unidirectional AX adapter)

Functionality

This function block calculates the XOR operation across all five inputs. The XOR operation (exclusive OR) returns a “true” signal if and only if an odd number of inputs are active. With an even number of active inputs, it returns “false”.

The mathematical formula is:
OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5

Technical Features

- Uses unidirectional AX adapters for all interfaces
- Implemented as a generic function block named 'GEN_AX_XOR'
- Supports up to five independent input signals
- No event control - operates continuously based on the adapter inputs

State Overview

Since it is a combinational logic block without memory, the AX_XOR_5_UNGATED has no internal states. The output is determined solely by the current input values.

Application Scenarios

- Parity checking in digital circuits
- Fault detection in safety systems
- Control logic with multiple conditions
- Distributed decision-making in automation systems
- Safety-critical applications with redundant sensors

⚖️ Comparison with Similar Blocks

Compared to standard XOR blocks with fewer inputs, the AX_XOR_5_UNGATED offers the ability to process up to five signals simultaneously. While simple XOR blocks typically have only two inputs, this function block enables more complex logical operations without additional nesting.

Comparison with XOR_5

- **AX_XOR_5**: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

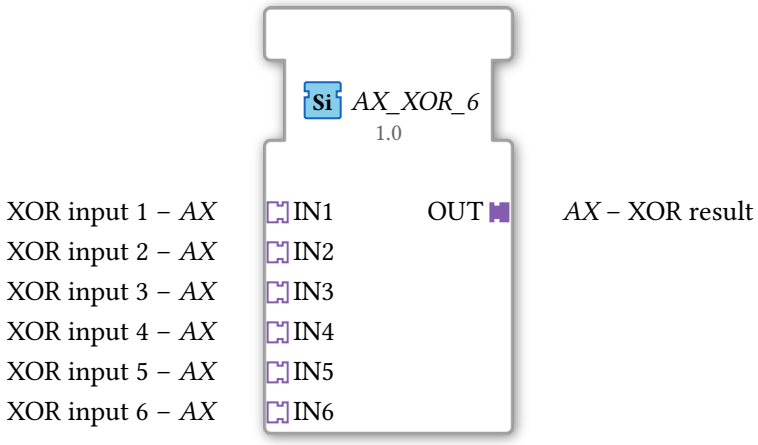
The AX_XOR_5_UNGATED function block provides a flexible solution for multi-input XOR operations. Its adapter-based interface allows for easy integration into existing control systems, while its generic implementation ensures reusability and extensibility.

AX_XOR_6

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_6 is a generic function block for calculating the Boolean XOR operation with six inputs. The block implements the exclusive OR operation for up to six binary input signals and outputs the result via an adapter output.



Interface Structure

Event Inputs
No event inputs available.

Event Outputs
No event outputs available.

Data Inputs
No direct data inputs available.

Data Outputs
No direct data outputs available.

Adapters
Plug Adapter:

- **OUT** (Type: adapter::types::unidirectional::AX) - XOR result

Socket Adapter:

- **IN1** (Type: adapter::types::unidirectional::AX) - XOR input 1
- **IN2** (Type: adapter::types::unidirectional::AX) - XOR input 2
- **IN3** (Type: adapter::types::unidirectional::AX) - XOR input 3
- **IN4** (Type: adapter::types::unidirectional::AX) - XOR input 4
- **IN5** (Type: adapter::types::unidirectional::AX) - XOR input 5
- **IN6** (Type: adapter::types::unidirectional::AX) - XOR input 6

Functionality
The function block calculates the XOR operation of all six inputs. The XOR operation returns a “TRUE” signal if and only if an odd number of inputs are set to “TRUE”. An even number of “TRUE” inputs returns “FALSE”.

Technical Features
- Generic function block with the generic class ‘GEN_AX_XOR’
- Uses unidirectional AX adapters for communication
- No event control - operates continuously based on the input values
- Supports six independent inputs for complex XOR calculations

State Overview
Since this is a purely combinational function block without event control, the AX_XOR_6 has no internal states. The output is determined solely by the current input values.

Application Scenarios
- Parity checking in data transmission systems
- Safety-critical controllers with multiple redundancy
- Error detection in binary signal chains
- Encryption algorithms with XOR operations
- Control systems with multi-input logic

⚖️ Comparison with similar function blocks
Compared to standard XOR function blocks with fewer inputs, the AX_XOR_6 offers the ability to process up to six signals simultaneously. While simple XOR blocks typically only have two inputs, this function block enables more complex logical operations without the need for additional nesting of multiple blocks.

Comparison with XOR_6

Change Detection
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_XOR_6 is a specialized function block for demanding multi-input XOR operations. Its adapter-based interface allows for easy integration into existing control systems and provides an efficient solution for complex Boolean operations without additional wiring complexity.

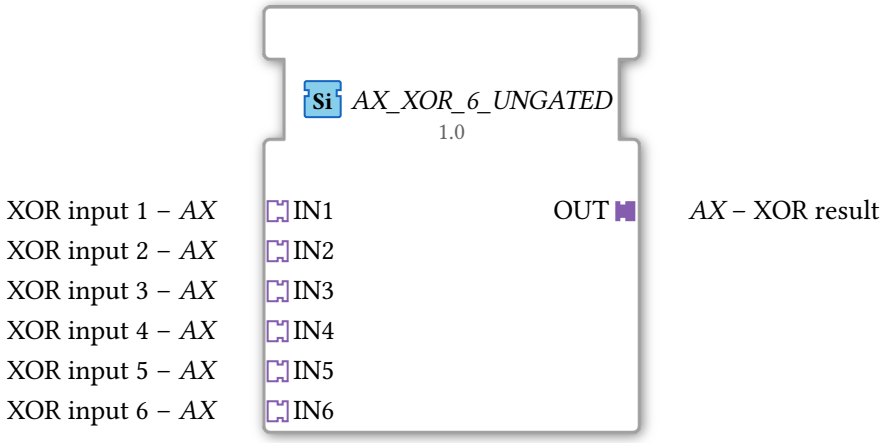
AX_XOR_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_XOR_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_6_UNGATED is a generic function block for calculating the Boolean XOR operation with six inputs. The block implements the exclusive OR operation for up to six binary input signals and outputs the result via an adapter output.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Plug Adapter:

- **OUT** (Type: adapter::types::unidirectional::AX) - XOR result

Socket Adapter:

- **IN1** (Type: adapter::types::unidirectional::AX) - XOR input 1
- **IN2** (Type: adapter::types::unidirectional::AX) - XOR input 2
- **IN3** (Type: adapter::types::unidirectional::AX) - XOR input 3
- **IN4** (Type: adapter::types::unidirectional::AX) - XOR input 4

- **IN5** (Type: adapter::types::unidirectional::AX) - XOR input 5
- **IN6** (Type: adapter::types::unidirectional::AX) - XOR input 6

Functionality

The function block calculates the XOR operation of all six inputs. The XOR operation returns a “TRUE” signal if and only if an odd number of inputs are set to “TRUE”. An even number of “TRUE” inputs returns “FALSE”.

Technical Features

- Generic function block with the generic class ‘GEN_AX_XOR’
- Uses unidirectional AX adapters for communication
- No event control - operates continuously based on the input values
- Supports six independent inputs for complex XOR calculations

State Overview

Since this is a purely combinational function block without event control, the AX_XOR_6_UNGATED has no internal states. The output is determined solely by the current input values.

Application Scenarios

- Parity checking in data transmission systems
- Safety-critical controllers with multiple redundancy
- Error detection in binary signal chains
- Encryption algorithms with XOR operations
- Control systems with multi-input logic

⚖️ Comparison with similar function blocks

Compared to standard XOR function blocks with fewer inputs, the AX_XOR_6_UNGATED offers the ability to process up to six signals simultaneously. While simple XOR blocks typically only have two inputs, this function block enables more complex logical operations without the need for additional nesting of multiple blocks.

Comparison with XOR_6

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

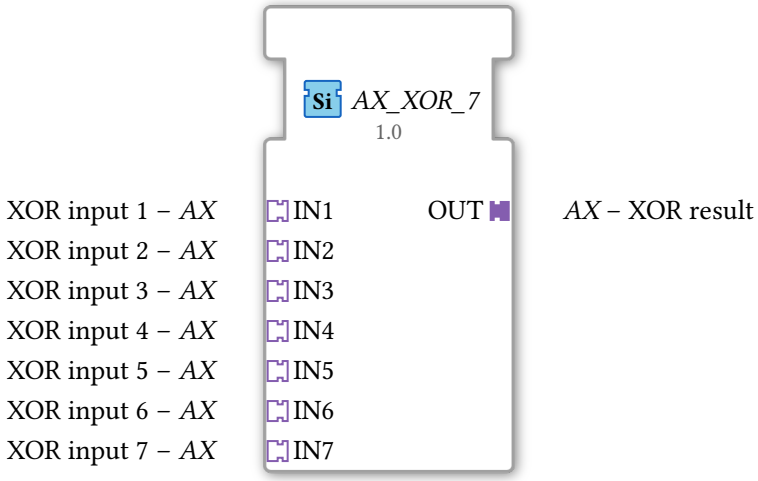
The AX_XOR_6_UNGATED is a specialized function block for demanding multi-input XOR operations. Its adapter-based interface allows for easy integration into existing control systems and provides an efficient solution for complex Boolean operations without additional wiring complexity.

AX_XOR_7

• − ◦ • − ◦ • − ◦ •

Introduction

The AX_XOR_7 function block is a generic function block for calculating the Boolean XOR operation with seven inputs. It allows the processing of up to seven input signals and outputs the result of the XOR operation.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Plug Adapter:

- `OUT` - XOR result (Adapter type: `adapter::types::unidirectional::AX`)

Socket Adapter:

- `IN1` - XOR input 1 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN2` - XOR input 2 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN3` - XOR input 3 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN4` - XOR input 4 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN5` - XOR input 5 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN6` - XOR input 6 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN7` - XOR input 7 (Adapter type: `adapter::types::unidirectional::AX`)

Functionality

This function block performs the XOR operation across all seven inputs. The XOR operation returns a "TRUE" signal if and only if an odd number of inputs are set to "TRUE". An even number of "TRUE" inputs results in a "FALSE" output.

Technical Features

- Generic function block with the generic class name `GEN_AX_XOR`
- Uses unidirectional AX adapters for inputs and outputs
- Supports exactly seven inputs for XOR calculations
- Implemented as an adapter-based solution without direct event or data interfaces

State Overview

The function block has no internal state and operates stateless. The output is calculated solely based on the current input values.

Application Scenarios

- Parity checking in data transmission systems
- Safety-critical controllers with majority voting
- Error detection in binary signal chains
- Logical operations in industrial automation systems

⚖️ Comparison with Similar Function Blocks

Compared to standard XOR blocks with fewer inputs, AX_XOR_7 offers the ability to perform XOR operations with up to seven inputs. While simple XOR blocks typically process only two inputs, this function block enables more complex logic operations within a single block.

Comparison with XOR_7

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_XOR_7 function block provides an efficient solution for multi-input XOR operations. Its adapter-based architecture allows for flexible integration into various control systems and makes it particularly suitable for applications requiring odd parity checks across multiple signals.

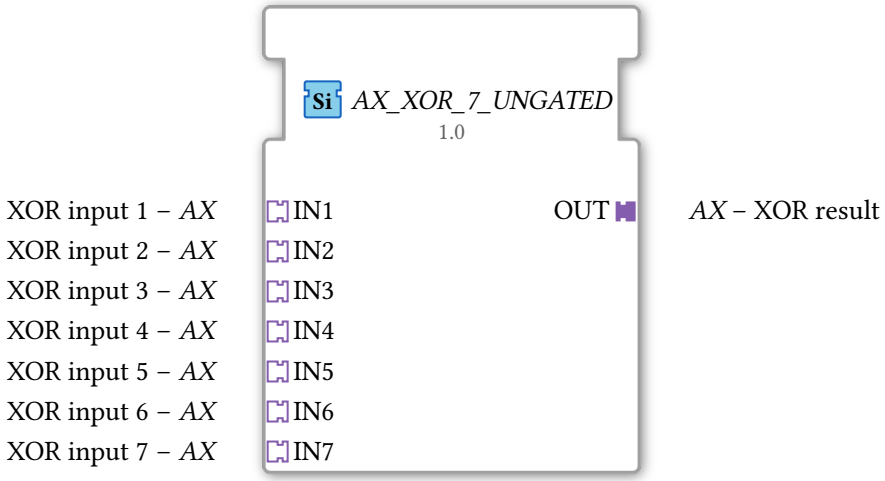
AX_XOR_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_XOR_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_7_UNGATED function block is a generic function block for calculating the Boolean XOR operation with seven inputs. It allows the processing of up to seven input signals and outputs the result of the XOR operation.



Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Plug Adapter:

- `OUT` - XOR result (Adapter type: `adapter::types::unidirectional::AX`)

Socket Adapter:

- `IN1` - XOR input 1 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN2` - XOR input 2 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN3` - XOR input 3 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN4` - XOR input 4 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN5` - XOR input 5 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN6` - XOR input 6 (Adapter type: `adapter::types::unidirectional::AX`)
- `IN7` - XOR input 7 (Adapter type: `adapter::types::unidirectional::AX`)

Functionality

This function block performs the XOR operation across all seven inputs. The XOR operation returns a "TRUE" signal if and only if an odd number of inputs are set to "TRUE". An even number of "TRUE" inputs results in a "FALSE" output.

Technical Features

- Generic function block with the generic class name `GEN_AX_XOR`
- Uses unidirectional AX adapters for inputs and outputs
- Supports exactly seven inputs for XOR calculations
- Implemented as an adapter-based solution without direct event or data interfaces

State Overview

The function block has no internal state and operates stateless. The output is calculated solely based on the current input values.

Application Scenarios

- Parity checking in data transmission systems
- Safety-critical controllers with majority voting
- Error detection in binary signal chains
- Logical operations in industrial automation systems

⚖️ Comparison with Similar Function Blocks

Compared to standard XOR blocks with fewer inputs, AX_XOR_7_UNGATED offers the ability to perform XOR operations with up to seven inputs. While simple XOR blocks typically process only two inputs, this function block enables more complex logic operations within a single block.

Comparison with XOR_7

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

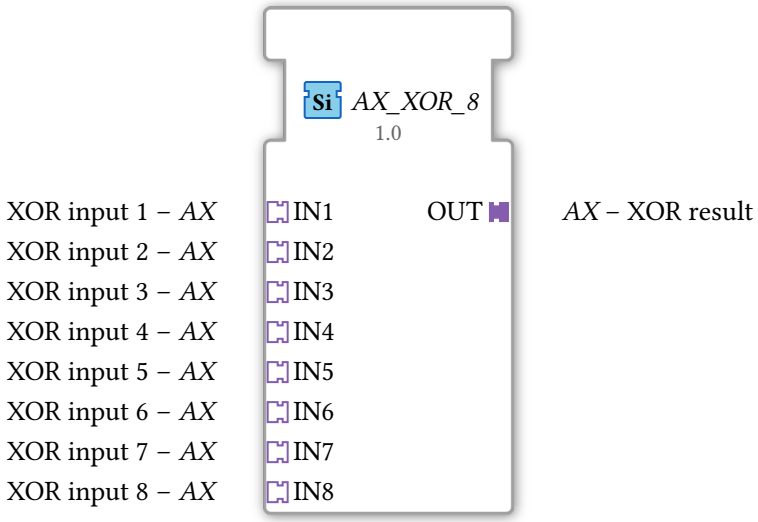
The AX_XOR_7_UNGATED function block provides an efficient solution for multi-input XOR operations. Its adapter-based architecture allows for flexible integration into various control systems and makes it particularly suitable for applications requiring odd parity checks across multiple signals.

AX_XOR_8

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_8 function block is a generic block for calculating Boolean XOR operations with 8 inputs. It enables the processing of logic signals according to the exclusive-OR principle and is designed for use in control applications.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Plug Adapter:

- **OUT**: Unidirectional adapter for the XOR result

Socket Adapter:

- **IN1**: Unidirectional adapter for XOR input 1
- **IN2**: Unidirectional adapter for XOR input 2
- **IN3**: Unidirectional adapter for XOR input 3
- **IN4**: Unidirectional adapter for XOR input 4
- **IN5**: Unidirectional adapter for XOR input 5
- **IN6**: Unidirectional adapter for XOR input 6
- **IN7**: Unidirectional adapter for XOR input 7
- **IN8**: Unidirectional adapter for XOR input 8

Functionality

This function block calculates the XOR operation across all 8 inputs. The XOR operation (exclusive OR) returns a TRUE signal if and only if an odd number of inputs are TRUE. If an even number of inputs are TRUE, FALSE is output.

Technical Features

- Generic function block with the class 'GEN_AX_XOR'
- Uses unidirectional adapters for all inputs and outputs
- No direct event or data interfaces
- Adapter-based architecture for flexible connectivity

State Overview

Since it is a combinational logic block, AX_XOR_8 has no internal state. The output is determined solely by the current input values.

Application Scenarios

- Parity checking in data transmission systems
- Safety-critical controllers with multiple redundancy
- Logical operations in automation systems
- Error detection in binary signal chains

⚖️ Comparison with similar components

Compared to simple XOR components with fewer inputs, AX_XOR_8 offers the ability to combine up to 8 signals simultaneously. The exclusive use of adapters instead of direct data and event inputs enables greater flexibility in complex system architectures.

Comparison with XOR_8

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AX_XOR_8 is a specialized logic component for demanding multi-input XOR operations. Its adapter-based interface makes it particularly suitable for modular system designs where flexible connections between functional blocks are required.

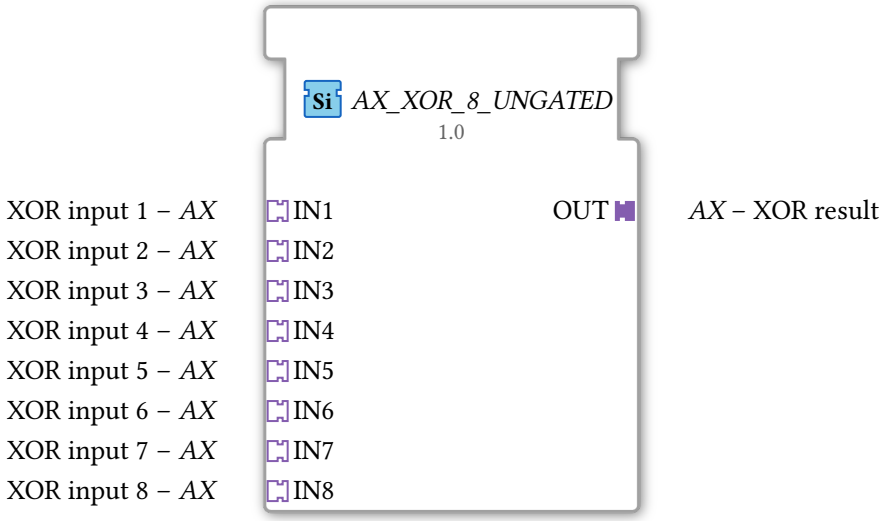
AX_XOR_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_XOR_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_XOR_8_UNGATED function block is a generic block for calculating Boolean XOR operations with 8 inputs. It enables the processing of logic signals according to the exclusive-OR principle and is designed for use in control applications.



Interface Structure

Event Inputs
No event inputs available

Event Outputs
No event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Plug Adapter:

- **OUT**: Unidirectional adapter for the XOR result

Socket Adapter:

- **IN1**: Unidirectional adapter for XOR input 1
- **IN2**: Unidirectional adapter for XOR input 2
- **IN3**: Unidirectional adapter for XOR input 3
- **IN4**: Unidirectional adapter for XOR input 4
- **IN5**: Unidirectional adapter for XOR input 5
- **IN6**: Unidirectional adapter for XOR input 6
- **IN7**: Unidirectional adapter for XOR input 7
- **IN8**: Unidirectional adapter for XOR input 8

Functionality
This function block calculates the XOR operation across all 8 inputs. The XOR operation (exclusive OR) returns a TRUE signal if and only if an odd number of inputs are TRUE. If an even number of inputs are TRUE, FALSE is output.

Technical Features

- Generic function block with the class 'GEN_AX_XOR'
- Uses unidirectional adapters for all inputs and outputs
- No direct event or data interfaces
- Adapter-based architecture for flexible connectivity

State Overview

Since it is a combinational logic block, AX_XOR_8_UNGATED has no internal state. The output is determined solely by the current input values.

Application Scenarios

- Parity checking in data transmission systems
- Safety-critical controllers with multiple redundancy
- Logical operations in automation systems
- Error detection in binary signal chains

⚖️ Comparison with similar components

Compared to simple XOR components with fewer inputs, AX_XOR_8_UNGATED offers the ability to combine up to 8 signals simultaneously. The exclusive use of adapters instead of direct data and event inputs enables greater flexibility in complex system architectures.

Comparison with XOR_8

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AX_XOR_8_UNGATED is a specialized logic component for demanding multi-input XOR operations. Its adapter-based interface makes it particularly suitable for modular system designs where flexible connections between functional blocks are required.

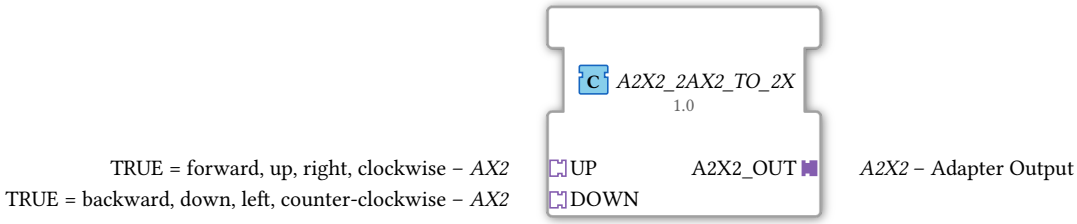
2.1.11.8 conversion

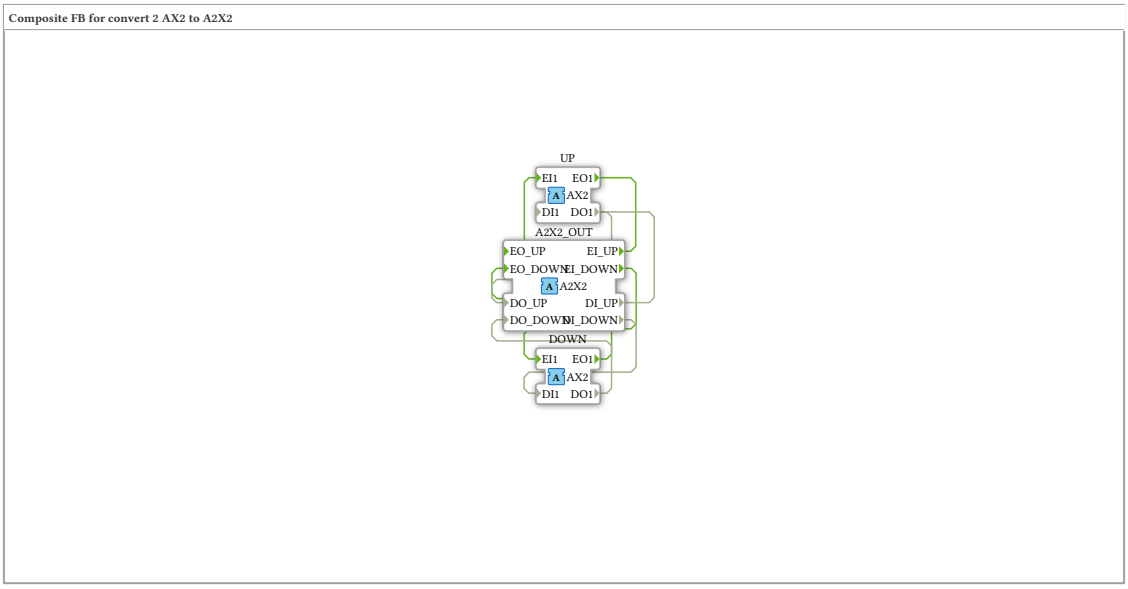
bidirectional

Numerical Index (bidirectional)

Here you will find all exercises from the `bidirectional` workspace in ascending order. ...

A2X2_2AX2_TO_2X





• – ○ • – ○ • – ○ •

Introduction

The A2X2_2AX2_TO_2X function block composes an A2X2 plug from two AX2 sockets — one for the UP channel, one for the DOWN channel. Since AX2 is itself already bidirectional (1 event/1 bool per direction), a single AX2 per channel is enough to cover both directions of A2X2.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **A2X2_OUT** (Plug): composed output of type `adapter::types::bidirectional::A2X2`
- **UP** (Socket): UP channel, of type `adapter::types::bidirectional::AX2` — TRUE = forward, up, right, clockwise
- **DOWN** (Socket): DOWN channel, of type `adapter::types::bidirectional::AX2` — TRUE = backward, down, left, counter-clockwise

Functionality

For each channel, the full bidirectional wiring between A2X2 and the corresponding AX2 socket is established: whatever `A2X2_OUT` receives on its request side (`EI_UP`/`DI_UP`) is forwarded to `UP.EI1`/`UP.DI1`; whatever `UP` delivers on its indication side (`EO1`/`DO1`) appears on `A2X2_OUT.EO_UP`/`DO_UP`. The same wiring applies to DOWN via the `DOWN` socket. Both channels are fully independent.

Technical Details

- Directly exploits AX2's bidirectionality — each channel needs only a single adapter, not two
- Pure wiring, no logic or state
- Every destination variable has exactly one writer, no fan-in on data connections

State Overview

The block is stateless:

- A2X2_OUT.EI_UP ⟶ UP.EI1, A2X2_OUT.DI_UP ⟶ UP.DI1
- UP.EO1 ⟶ A2X2_OUT.EO_UP, UP.DO1 ⟶ A2X2_OUT.DO_UP
- A2X2_OUT.EI_DOWN ⟶ DOWN.EI1, A2X2_OUT.DI_DOWN ⟶ DOWN.DI1
- DOWN.EO1 ⟶ A2X2_OUT.EO_DOWN, DOWN.DO1 ⟶ A2X2_OUT.DO_DOWN

Application Scenarios

- Building an A2X2 endpoint from two already existing, independent AX2 channels
- Systems where UP and DOWN are historically wired separately as AX2 but should present themselves as a single A2X2 externally
- Modular composition of larger adapters from smaller, reusable blocks

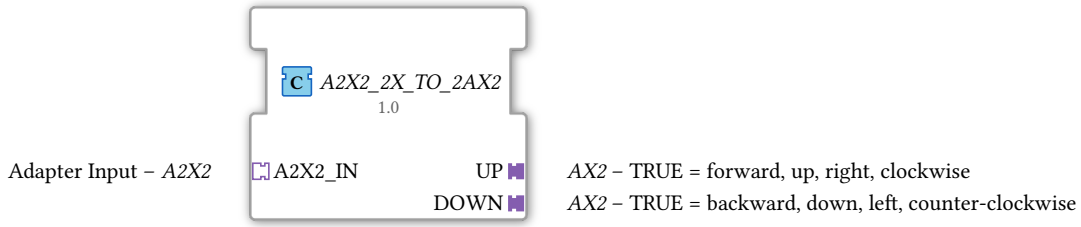
⚖️ Comparison with Similar Blocks

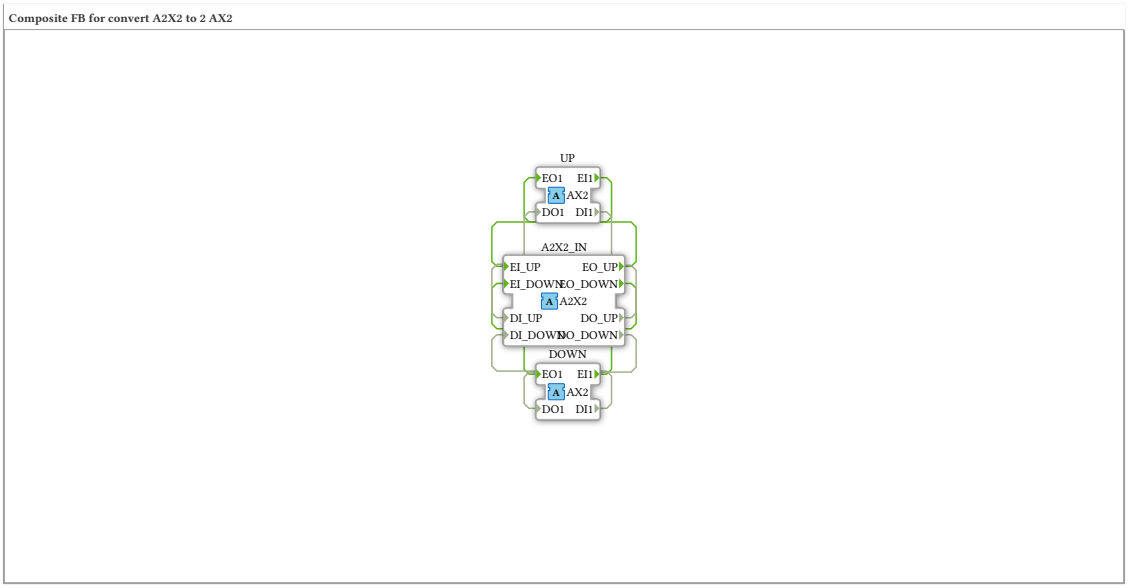
The counterpart A2X2_2X_TO_2AX2 decomposes instead of composing. For the same task, A2X2_4AX_TO_2X offers an alternative that uses four unidirectional AX adapters instead of two bidirectional AX2 (two per channel, one per direction) — useful when only unidirectional AX infrastructure is available. The unidirectional predecessor A2X_2AX_TO_2X composes analogously from two plain AX adapters.

Conclusion

A2X2_2AX2_TO_2X is the most efficient way to build an A2X2 from two existing AX2 channels, since both adapters are already bidirectional and no extra logic is needed.

A2X2_2X_TO_2AX2





• – ○ • – ○ • – ○ •

Introduction

The A2X2_2X_TO_2AX2 function block is the inverse of A2X2_2AX2_TO_2X: it decomposes an A2X2 socket into two AX2 plugs — one for UP, one for DOWN.

Interface Structure

Event Inputs
The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs
The function block has no direct event outputs.

Data Inputs
The function block has no direct data inputs.

Data Outputs
The function block has no direct data outputs.

Adapters
- **A2X2_IN** (Socket): input to be decomposed, of type `adapter::types::bidirectional::A2X2`
- **UP** (Plug): UP channel, of type `adapter::types::bidirectional::AX2` — TRUE = forward, up, right, clockwise
- **DOWN** (Plug): DOWN channel, of type `adapter::types::bidirectional::AX2` — TRUE = backward, down, left, counter-clockwise

Functionality
For each channel, the full bidirectional wiring between the A2X2 socket and the corresponding AX2 plug is established: whatever `A2X2_IN` receives on its indication side (`EO_UP`/`DO_UP`) is passed to `UP.EO1`/`UP.DO1`; whatever arrives at the `UP` plug on its request side (`EI1`/`DI1`) is reported back to `A2X2_IN.EI_UP`/`DI_UP`. The same wiring applies to DOWN via the `DOWN` plug. Both channels are fully independent.

Technical Details
- Directly exploits AX2's bidirectionality — each channel needs only a single adapter, not two
- Pure wiring, no logic or state
- Every destination variable has exactly one writer, no fan-in on data connections

State Overview
The block is stateless:

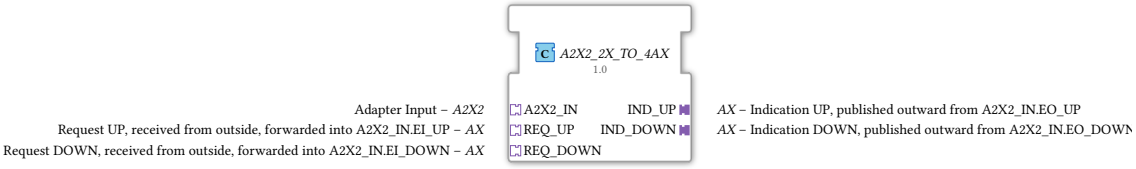
- A2X2_IN.EO_UP ⟶ UP.EO1, A2X2_IN.DO_UP ⟶ UP.DO1
- UP.EI1 ⟶ A2X2_IN.EI_UP, UP.DI1 ⟶ A2X2_IN.DI_UP
- A2X2_IN.EO_DOWN ⟶ DOWN.EO1, A2X2_IN.DO_DOWN ⟶ DOWN.DO1
- DOWN.EI1 ⟶ A2X2_IN.EI_DOWN, DOWN.DI1 ⟶ A2X2_IN.DI_DOWN

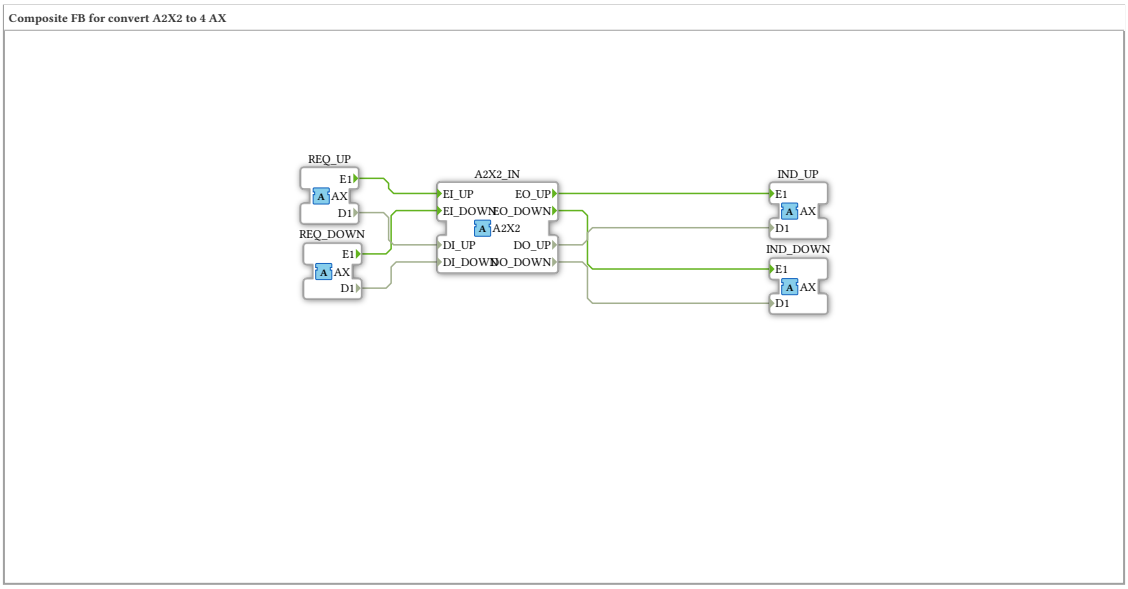
Application Scenarios
- Splitting an A2X2 bus signal into two independently processable AX2 channels
- Connecting existing AX2-based subsystems to a central A2X2
- Test setups that need to observe or control each channel individually

⚖️ Comparison with Similar Blocks
The counterpart A2X2_2AX2_TO_2X composes instead of decomposing. For the same task, A2X2_4AX_TO_2X / A2X2_2X_TO_4AX offer an alternative with four unidirectional AX adapters instead of two bidirectional AX2. The unidirectional predecessor A2X_2X_TO_2AX decomposes analogously into two plain AX adapters.

Conclusion
A2X2_2X_TO_2AX2 is the most efficient way to split an A2X2 signal into two independent AX2 channels, since both adapters are already bidirectional and no extra logic is needed.

A2X2_2X_TO_4AX





• – ○ • – ○ • – ○ •

Introduction

The A2X2_2X_TO_4AX function block decomposes an A2X2 socket into **four** unidirectional AX adapters. Since A2X2 has two channels (UP/DOWN) and each of them is itself bidirectional, representing it with unidirectional AX needs two AX per channel to cover both directions — one for the indication direction, one for the request direction. 2 channels × 2 directions = 4 AX.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **A2X2_IN** (Socket): input to be decomposed, of type `adapter::types::bidirectional::A2X2`
- **REQ_UP** (Socket): UP request, received from outside and forwarded to `A2X2_IN.EI_UP`, of type `adapter::types::unidirectional::AX`
- **REQ_DOWN** (Socket): DOWN request, received from outside and forwarded to `A2X2_IN.EI_DOWN`, of type `adapter::types::unidirectional::AX`
- **IND_UP** (Plug): UP indication, published outward from `A2X2_IN.EO_UP`, of type `adapter::types::unidirectional::AX`
- **IND_DOWN** (Plug): DOWN indication, published outward from `A2X2_IN.EO_DOWN`, of type `adapter::types::unidirectional::AX`

Functionality

Whatever the A2X2 socket receives on its indication side (`EO_UP`/`DO_UP` or `EO_DOWN`/`DO_DOWN`) is published unchanged via the unidirectional plugs `IND_UP`/`IND_DOWN`. Conversely, whatever arrives from outside at the unidirectional sockets `REQ_UP`/`REQ_DOWN` is forwarded to the request side of the A2X2 socket (`EI_UP`/`DI_UP` or `EI_DOWN`/`DI_DOWN`). This makes both directions of each channel available as their own plain unidirectional AX adapter.

Technical Details

- Four separate AX adapters instead of two AX2, because AX does not combine an `EventInputs`/`EventOutputs` pair in one adapter — each direction needs its own adapter
- Pure wiring, no logic or state
- Every destination variable has exactly one writer, no fan-in on data connections

State Overview

The block is stateless:

- A2X2_IN.EO_UP ⟶ IND_UP.E1, A2X2_IN.DO_UP ⟶ IND_UP.D1
- A2X2_IN.EO_DOWN ⟶ IND_DOWN.E1, A2X2_IN.DO_DOWN ⟶ IND_DOWN.D1
- REQ_UP.E1 ⟶ A2X2_IN.EI_UP, REQ_UP.D1 ⟶ A2X2_IN.DI_UP
- REQ_DOWN.E1 ⟶ A2X2_IN.EI_DOWN, REQ_DOWN.D1 ⟶ A2X2_IN.DI_DOWN

Application Scenarios

- Connecting an A2X2 signal to infrastructure that only knows unidirectional AX adapters (e.g. older subsystems)
- Processing the request and indication directions separately, e.g. for independent logging or filtering
- Systems where bidirectional AX2 adapters are unavailable or undesired

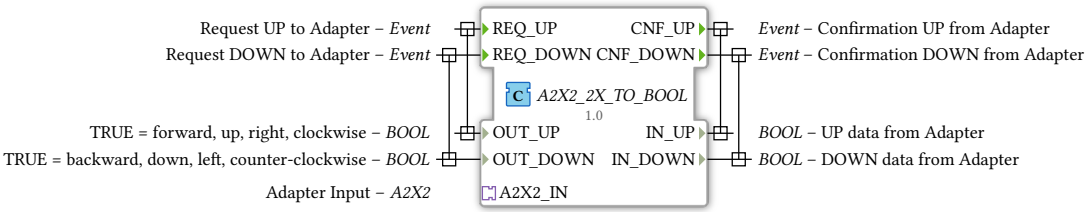
⚖️ Comparison with Similar Blocks

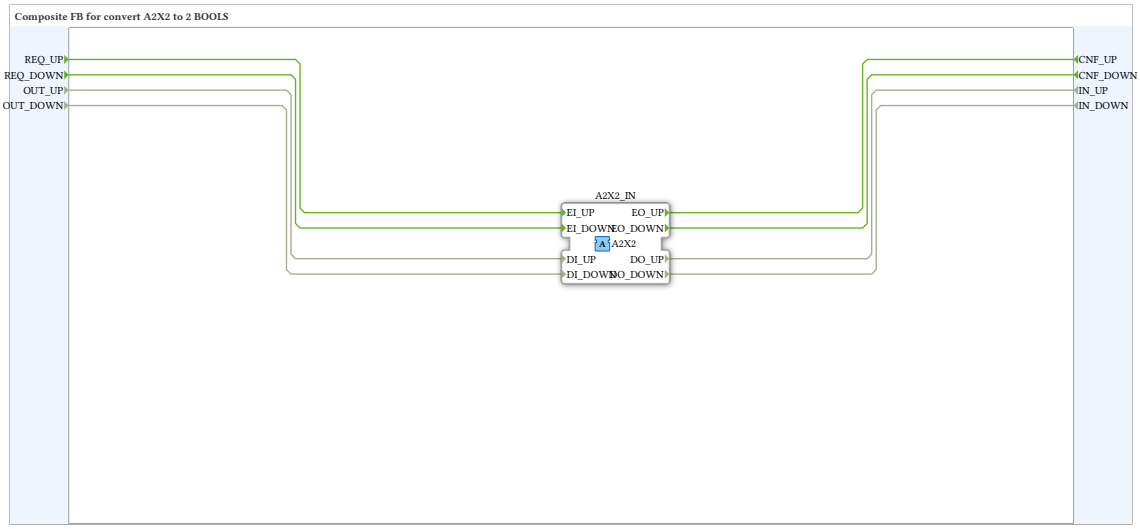
For the same task (decomposing A2X2), A2X2_2X_TO_2AX2 offers a simpler alternative that only needs two bidirectional AX2 instead of four unidirectional AX — where AX2 infrastructure is available, that is the leaner solution. The counterpart to this block is A2X2_4AX_TO_2X, which recomposes an A2X2 from four AX.

Conclusion

A2X2_2X_TO_4AX is the right choice when an A2X2 signal must be integrated into a purely unidirectional AX landscape — at the cost of twice as many adapters as the AX2-based alternative.

A2X2_2X_TO_BOOL





• – ○ • – ○ • – ○ •

Introduction

The A2X2_2X_TO_BOOL function block is the socket counterpart of A2X2_BOOL_TO_2X: it translates the bidirectional A2X2 adapter format back into two plain BOOL channels (UP and DOWN) by exposing an A2X2 socket.

Interface Structure

Event Inputs

- **REQ_UP**: request event for the UP channel, delivers `OUT_UP`
- **REQ_DOWN**: request event for the DOWN channel, delivers `OUT_DOWN`

Event Outputs

- **CNF_UP**: confirmation event for the UP channel, delivers `IN_UP`
- **CNF_DOWN**: confirmation event for the DOWN channel, delivers `IN_DOWN`

Data Inputs

- **OUT_UP**: BOOL, TRUE = forward, up, right, clockwise
- **OUT_DOWN**: BOOL, TRUE = backward, down, left, counter-clockwise

Data Outputs

- **IN_UP**: BOOL, UP data from the adapter
- **IN_DOWN**: BOOL, DOWN data from the adapter

Adapters

- **A2X2_IN** (Socket): input adapter of type `adapter::types::bidirectional::A2X2`

Functionality

When the socket `A2X2_IN` receives events on its indication side (`EO_UP`/`DO_UP` or `EO_DOWN`/`DO_DOWN`), they are immediately reported outward as `CNF_UP`/`IN_UP` or `CNF_DOWN`/`IN_DOWN`. Conversely, when a `REQ_UP` or `REQ_DOWN` event arrives, the corresponding `OUT_UP`/`OUT_DOWN` value is sent out via the socket (`EI_UP`/`DI_UP` or `EI_DOWN`/`DI_DOWN`). Both channels operate completely independently of each other.

Technical Details

- Pure 1:1 pass-through, no internal state, no gates or logic blocks needed
- Uses an A2X2 socket, i.e. this block is the counterpart that an A2X2 plug connects to
- Identical interface to A2X2_BOOL_TO_2X, only the adapter role (socket instead of plug) differs

State Overview

The block is stateless; every connection acts immediately and directly:

- A2X2_IN.EO_UP ⟶ CNF_UP, A2X2_IN.DO_UP ⟶ IN_UP
- A2X2_IN.EO_DOWN ⟶ CNF_DOWN, A2X2_IN.DO_DOWN ⟶ IN_DOWN
- REQ_UP ⟶ A2X2_IN.EI_UP, OUT_UP ⟶ A2X2_IN.DI_UP
- REQ_DOWN ⟶ A2X2_IN.EI_DOWN, OUT_DOWN ⟶ A2X2_IN.DI_DOWN

Application Scenarios

- Reading the UP/DOWN channels of an A2X2 bus participant as plain BOOL signals
- Integrating an A2X2 endpoint into a classically wired controller
- Test blocks acting as the counterpart of A2X2_BOOL_TO_2X

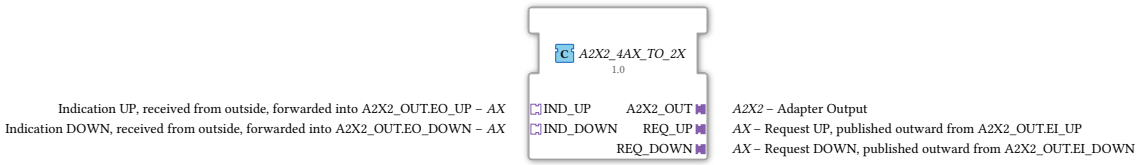
⚖ Comparison with Similar Blocks

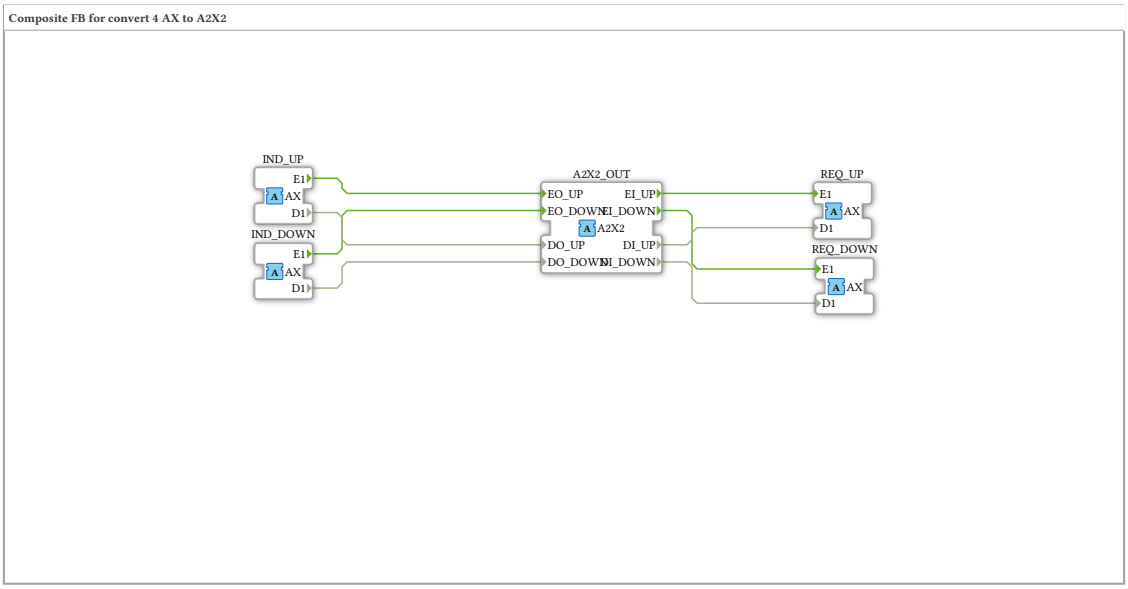
The plug counterpart is A2X2_BOOL_TO_2X. For the unidirectional case, see A2X_2X_TO_BOOL. The single-channel bidirectional predecessor is AX2_X_TO_BOOL.

Conclusion

A2X2_2X_TO_BOOL is the direct counterpart of A2X2_BOOL_TO_2X and makes an A2X2 bus signal usable as two plain BOOL channels for the rest of the application.

A2X2_4AX_TO_2X





• – ○ • – ○ • – ○ •

Introduction

The A2X2_4AX_TO_2X function block is the inverse of A2X2_2X_TO_4AX: it composes an A2X2 plug from **four** unidirectional AX adapters. Two channels (UP/DOWN) × two directions (indication/request) = four AX.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **A2X2_OUT** (Plug): composed output of type `adapter::types::bidirectional::A2X2`
- **REQ_UP** (Plug): UP request, published outward from `A2X2_OUT.EI_UP`, of type `adapter::types::unidirectional::AX`
- **REQ_DOWN** (Plug): DOWN request, published outward from `A2X2_OUT.EI_DOWN`, of type `adapter::types::unidirectional::AX`
- **IND_UP** (Socket): UP indication, received from outside and forwarded to `A2X2_OUT.EO_UP`, of type `adapter::types::unidirectional::AX`
- **IND_DOWN** (Socket): DOWN indication, received from outside and forwarded to `A2X2_OUT.EO_DOWN`, of type `adapter::types::unidirectional::AX`

Functionality

Whatever arrives from outside at the unidirectional sockets `IND_UP`/`IND_DOWN` is passed through to the indication side of the A2X2 plug (`EO_UP`/`DO_UP` or `EO_DOWN`/`DO_DOWN`). Conversely, whatever the A2X2 plug receives on its request side (`EI_UP`/`DI_UP` or `EI_DOWN`/`DI_DOWN`) is published via the unidirectional plugs `REQ_UP`/`REQ_DOWN`.

Technical Details

- Four separate AX adapters instead of two AX2, because AX does not combine an `EventInputs`/`EventOutputs` pair in one adapter — each direction needs its own adapter
- Pure wiring, no logic or state
- Every destination variable has exactly one writer, no fan-in on data connections

State Overview

The block is stateless:

- IND_UP.E1 ⟶ A2X2_OUT.EO_UP, IND_UP.D1 ⟶ A2X2_OUT.DO_UP
- IND_DOWN.E1 ⟶ A2X2_OUT.EO_DOWN, IND_DOWN.D1 ⟶ A2X2_OUT.DO_DOWN
- A2X2_OUT.EI_UP ⟶ REQ_UP.E1, A2X2_OUT.DI_UP ⟶ REQ_UP.D1
- A2X2_OUT.EI_DOWN ⟶ REQ_DOWN.E1, A2X2_OUT.DI_DOWN ⟶ REQ_DOWN.D1

Application Scenarios

- Building an A2X2 endpoint from four already existing unidirectional AX signals
- Migration scenarios where an existing AX infrastructure is gradually converted to A2X2
- Systems where bidirectional AX2 adapters are unavailable or undesired

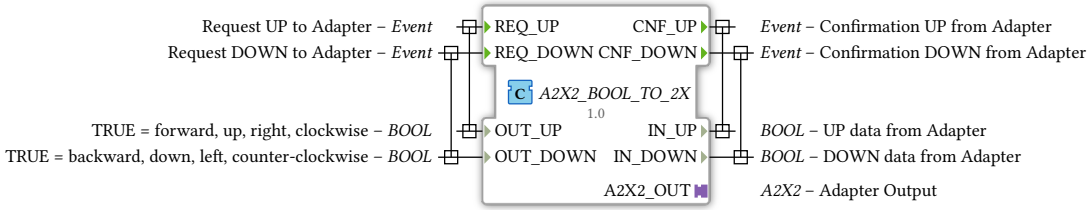
⚖️ Comparison with Similar Blocks

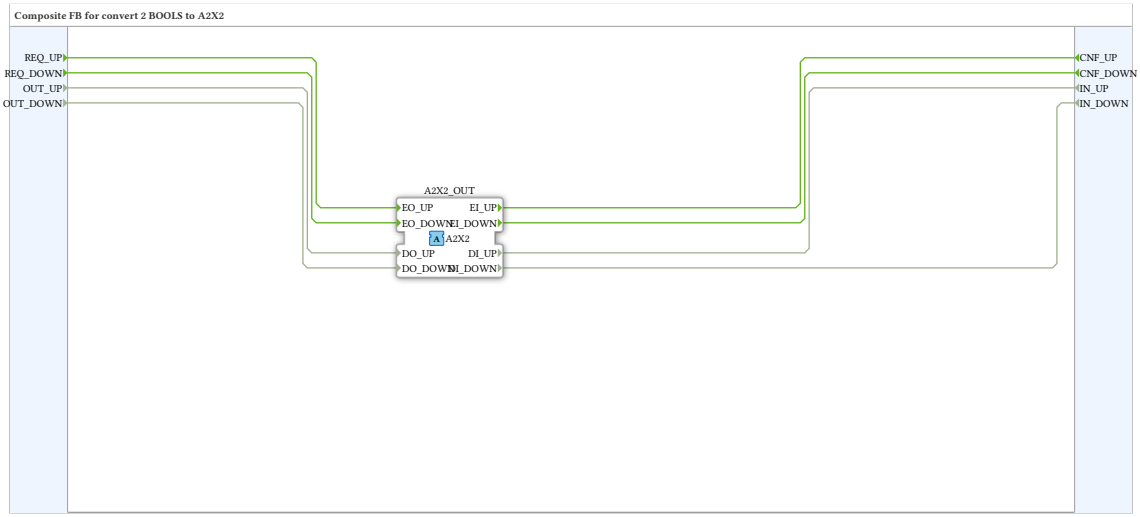
For the same task (composing A2X2), A2X2_2AX2_TO_2X offers a simpler alternative that only needs two bidirectional AX2 instead of four unidirectional AX — where AX2 infrastructure is available, that is the leaner solution. The counterpart to this block is A2X2_2X_TO_4AX, which decomposes an A2X2 back into four AX.

Conclusion

A2X2_4AX_TO_2X is the right choice when an A2X2 endpoint must be built from purely unidirectional AX infrastructure — at the cost of twice as many adapters as the AX2-based alternative.

A2X2_BOOL_TO_2X





• – ○ • – ○ • – ○ •

Introduction

The A2X2_BOOL_TO_2X function block is a composite function block that translates two plain BOOL channels (UP and DOWN) into the bidirectional A2X2 adapter format. It exposes an A2X2 plug and maps each of the two channels independently through its own request/confirmation event pair.

Interface Structure

Event Inputs

- **REQ_UP**: request event for the UP channel, delivers `OUT_UP`
- **REQ_DOWN**: request event for the DOWN channel, delivers `OUT_DOWN`

Event Outputs

- **CNF_UP**: confirmation event for the UP channel, delivers `IN_UP`
- **CNF_DOWN**: confirmation event for the DOWN channel, delivers `IN_DOWN`

Data Inputs

- **OUT_UP**: BOOL, TRUE = forward, up, right, clockwise
- **OUT_DOWN**: BOOL, TRUE = backward, down, left, counter-clockwise

Data Outputs

- **IN_UP**: BOOL, UP data from the adapter
- **IN_DOWN**: BOOL, DOWN data from the adapter

Adapters

- **A2X2_OUT** (Plug): output adapter of type `adapter::types::bidirectional::A2X2`

Functionality

When a `REQ_UP` event arrives, the current value of `OUT_UP` is sent out via `A2X2_OUT.EO_UP`/`A2X2_OUT.DO_UP`; the same happens for `REQ_DOWN`/`OUT_DOWN` via `EO_DOWN`/`DO_DOWN`. Conversely, whatever the adapter receives on its confirmation side (`A2X2_OUT.EI_UP`/`A2X2_OUT.EI_DOWN` with `DI_UP`/`DI_DOWN`)

is forwarded unchanged as `CNF_UP`/`IN_UP` and `CNF_DOWN`/`IN_DOWN` respectively. Both channels operate completely independently of each other.

Technical Details

- Pure 1:1 pass-through, no internal state, no gates or logic blocks needed
- Each channel (UP/DOWN) has its own event pair — there is no shared variable that two sources would need to write at once
- Uses the A2X2 plug, i.e. this block acts as the "endpoint" that an A2X2 socket connects to

State Overview

The block is stateless; every connection acts immediately and directly:

- REQ_UP ⟶ A2X2_OUT.EO_UP, OUT_UP ⟶ A2X2_OUT.DO_UP
- REQ_DOWN ⟶ A2X2_OUT.EO_DOWN, OUT_DOWN ⟶ A2X2_OUT.DO_DOWN
- A2X2_OUT.EI_UP ⟶ CNF_UP, A2X2_OUT.DI_UP ⟶ IN_UP
- A2X2_OUT.EI_DOWN ⟶ CNF_DOWN, A2X2_OUT.DI_DOWN ⟶ IN_DOWN

Application Scenarios

- Connecting two plain BOOL signals (e.g. two buttons or limit switches for UP/DOWN) to an A2X2 bus system
- Test blocks that simulate A2X2 traffic without real hardware
- Bridging classic BOOL wiring and adapter-based subnetworks

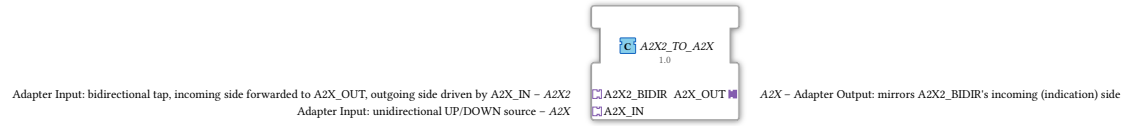
⚖️ Comparison with Similar Blocks

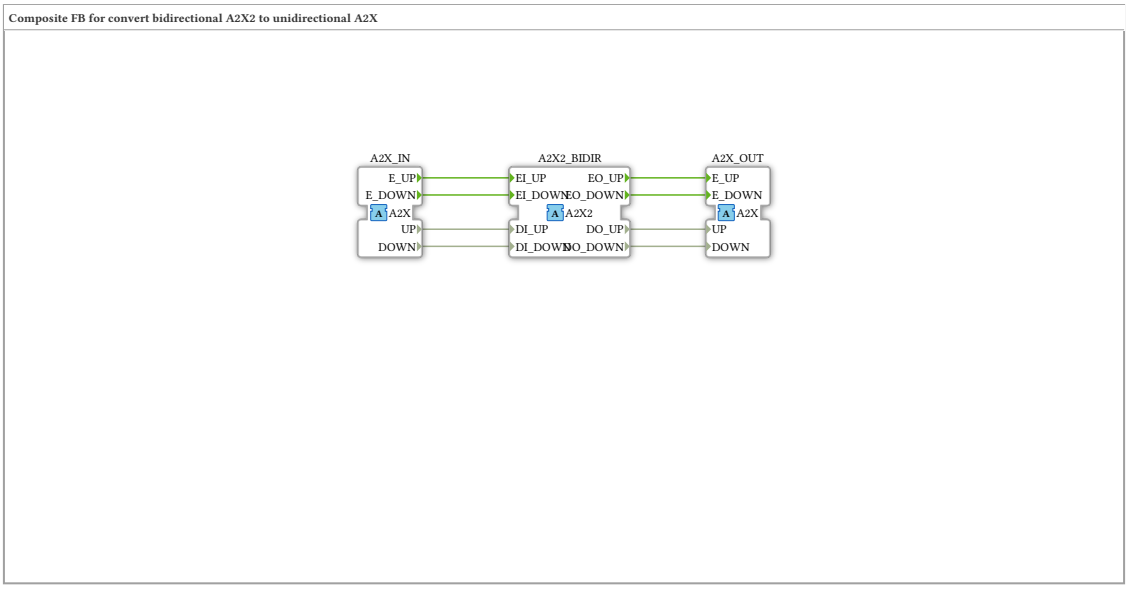
The socket counterpart A2X2_2X_TO_BOOL has the same interface but uses an A2X2 socket instead of a plug. For the unidirectional case, A2X_BOOL_TO_2X implements the same basic idea but without a request side (only `E_UP`/`E_DOWN`). The single-channel bidirectional predecessor is AX2_BOOL_TO_X, whose pattern this block doubles onto two independent channels.

Conclusion

A2X2_BOOL_TO_2X is the simplest way to connect plain BOOL signals to an A2X2 adapter — no state, no logic, a direct 1:1 pass-through per channel.

A2X2_TO_A2X





• — ○ • — ○ • — ○ •

Introduction

The A2X2_TO_A2X function block is the mirror image of A2X_TO_A2X2: here the bidirectional A2X2 tap sits as a socket on the left (input) side, complemented by a second unidirectional A2X socket and a unidirectional A2X plug.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **A2X2_BIDIR** (Socket): bidirectional tap whose incoming side is forwarded to `A2X_OUT` and whose outgoing side is fed by `A2X_IN`, of type `adapter::types::bidirectional::A2X2`
- **A2X_IN** (Socket): unidirectional UP/DOWN source, of type `adapter::types::unidirectional::A2X`
- **A2X_OUT** (Plug): mirrors the incoming (indication) side of `A2X2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::A2X`

Functionality

The block operates in two directions at once: whatever the socket `A2X2_BIDIR` receives on its indication side (`EO_UP`/`DO_UP`, `EO_DOWN`/`DO_DOWN` — i.e. whatever a bidirectional peer attached to A2X2_BIDIR reports) is output unchanged as a standalone unidirectional signal via the plug `A2X_OUT`. Conversely, whatever arrives at the socket `A2X_IN` (`E_UP`/`UP`, `E_DOWN`/`DOWN`) is published onto the request side of `A2X2_BIDIR` (`EI_UP`/`DI_UP`, `EI_DOWN`/`DI_DOWN`) — so A2X2_BIDIR forwards the same UP/DOWN signal to the attached bidirectional peer.

Technical Details

- Three adapters instead of two: the direction received from the tap and the separately injected direction are deliberately kept apart rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `A2X_OUT` only reflects what is reported via `A2X2_BIDIR` by an attached bidirectional peer — without such a peer, `A2X_OUT` stays idle

State Overview

The block is stateless:

- A2X2_BIDIR.EO_UP ⟶ A2X_OUT.E_UP, A2X2_BIDIR.DO_UP ⟶ A2X_OUT.UP
- A2X2_BIDIR.EO_DOWN ⟶ A2X_OUT.E_DOWN, A2X2_BIDIR.DO_DOWN ⟶ A2X_OUT.DOWN
- A2X_IN.E_UP ⟶ A2X2_BIDIR.EI_UP, A2X_IN.UP ⟶ A2X2_BIDIR.DI_UP
- A2X_IN.E_DOWN ⟶ A2X2_BIDIR.EI_DOWN, A2X_IN.DOWN ⟶ A2X2_BIDIR.DI_DOWN

Application Scenarios

- Reading the status of a bidirectional A2X2 peer as a plain unidirectional A2X signal, combined with injecting a separate unidirectional command signal
- Systems where a bidirectional peer needs to be monitored without converting the rest of the control logic to A2X2
- Bridging a bidirectional A2X2 zone and a unidirectional A2X zone

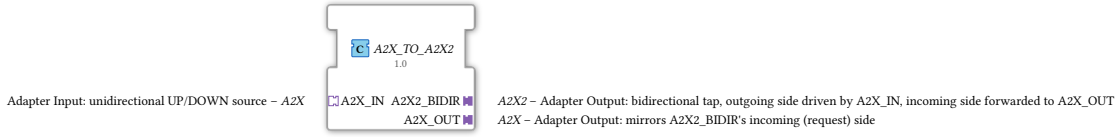
⚖️ Comparison with Similar Blocks

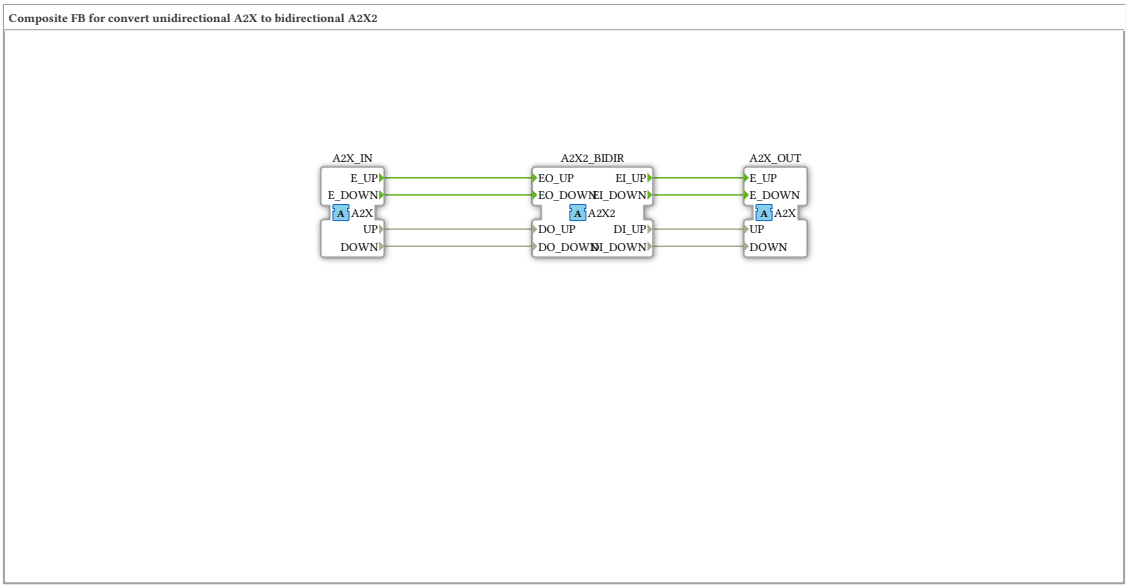
A2X_TO_A2X2 is the mirror image of this block — there, `A2X2_BIDIR` sits as a plug on the right side. For a pure, non-tapping 1:1 conversion between the single-channel adapters, see the conversion blocks between AX and AX2 in the same library.

Conclusion

A2X2_TO_A2X is not a simple converter but a tap point: it makes the state of a bidirectional A2X2 peer visible as a standalone unidirectional A2X signal while simultaneously injecting a separate unidirectional signal into the bidirectional side.

A2X_TO_A2X2





• – ○ • – ○ • – ○ •

Introduction

The A2X_TO_A2X2 function block inserts a bidirectional A2X2 tap into a unidirectional A2X chain. Rather than a simple 1:1 conversion, it uses **three** adapters: an incoming unidirectional A2X signal is both published onto the bidirectional A2X2 interface and — whatever A2X2 returns on its request side — output again as a standalone unidirectional A2X signal.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **A2X_IN** (Socket): unidirectional UP/DOWN source, of type `adapter::types::unidirectional::A2X`
- **A2X2_BIDIR** (Plug): bidirectional tap whose outgoing side is fed by `A2X_IN` and whose incoming side is forwarded to `A2X_OUT`, of type `adapter::types::bidirectional::A2X2`
- **A2X_OUT** (Plug): mirrors the incoming (request) side of `A2X2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::A2X`

Functionality

The block operates in two directions at once: whatever arrives at the socket `A2X_IN` (`E_UP`/`UP`, `E_DOWN`/`DOWN`) is published directly onto the outgoing side of `A2X2_BIDIR` (`EO_UP`/`DO_UP`, `EO_DOWN`/`DO_DOWN`) — so A2X2_BIDIR reports outward the same UP/DOWN signal coming in via A2X_IN. Conversely, whatever `A2X2_BIDIR` receives on its request side (`EI_UP`/`DI_UP`, `EI_DOWN`/`DI_DOWN` — i.e. whatever a bidirectional peer attached to A2X2_BIDIR "sends back") is output unchanged as a standalone unidirectional signal via the plug `A2X_OUT`. A2X2_BIDIR thus acts as a tap point that doesn't break the unidirectional chain but extends it with a bidirectional observation/injection point.

Technical Details

- Three adapters instead of two: the incoming signal direction and the direction coming back from the tap are deliberately kept separate rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `A2X_OUT` only reflects what comes back via `A2X2_BIDIR` from an attached bidirectional peer — without such a peer, `A2X_OUT` stays idle

State Overview

The block is stateless:

- A2X_IN.E_UP ⟶ A2X2_BIDIR.EO_UP, A2X_IN.UP ⟶ A2X2_BIDIR.DO_UP
- A2X_IN.E_DOWN ⟶ A2X2_BIDIR.EO_DOWN, A2X_IN.DOWN ⟶ A2X2_BIDIR.DO_DOWN
- A2X2_BIDIR.EI_UP ⟶ A2X_OUT.E_UP, A2X2_BIDIR.DI_UP ⟶ A2X_OUT.UP
- A2X2_BIDIR.EI_DOWN ⟶ A2X_OUT.E_DOWN, A2X2_BIDIR.DI_DOWN ⟶ A2X_OUT.DOWN

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional A2X signal chain without breaking it
- Systems where a bidirectional peer (e.g. an operator panel) needs to both display the status of a unidirectional signal source and send commands back
- Bridging a unidirectional A2X zone and a bidirectional A2X2 zone

⚖️ Comparison with Similar Blocks

A2X2_TO_A2X is the mirror image of this block — there, `A2X2_BIDIR` sits as a socket on the left side. For a pure, non-tapping 1:1 conversion between the single-channel adapters, see the conversion blocks between AX and AX2 in the same library.

Conclusion

A2X_TO_A2X2 is not a simple converter but a tap point: it makes a unidirectional A2X signal visible at a bidirectional A2X2 point without breaking the original unidirectional signal chain, while forwarding the request signal received there as a standalone A2X signal.

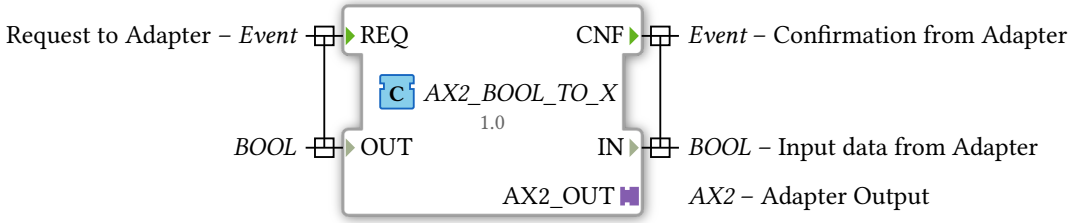
AX2_BOOL_TO_X



• – ○ • – ○ • – ○ •

Introduction

The AX2_BOOL_TO_X function block serves as a composite function block for converting BOOL values into the AX2 adapter format. It enables bidirectional communication between simple BOOL data and the specific AX2 adapter interface.



Interface Structure

Event Inputs

- **REQ**: Request event to activate the adapter
- Linked to data output OUT
- Y position: 46

Event Outputs

- **CNF**: Acknowledge event from the adapter
- Linked to data input IN
- Y position: 46

Data Inputs

- **OUT**: BOOL data input for output
- Y position: 31

Data Outputs

- **IN**: BOOL data output for input
- Receives data from the adapter
- Y position: 47

Adapter

- **AX2_OUT**: AX2 adapter for bidirectional communication
- Adapter type: adapter::types::bidirectional::AX2

- Position: x=−3900, y=−500

Functionality

This function block acts as a bridge between BOOL data and the AX2 adapter. Upon receiving a REQ event, the OUT value is sent via the AX2_OUT adapter. Simultaneously, incoming data can be received via the same adapter and made available as an IN output with a CNF event.

Technical Features

- Implemented as a composite function block
- Supports bidirectional data transmission
- Uses a specific AX2 adapter interface
- Automatic event and data forwarding between the main interface and the adapter

State Overview

The function block operates statelessly and forwards events and data directly between the interfaces:

- REQ ⟶ AX2_OUT.EO1
- AX2_OUT.EI1 ⟶ CNF
- OUT ⟶ AX2_OUT.DO1
- AX2_OUT.DI1 ⟶ IN

Application Scenarios

- Integration of BOOL signals into AX2-based systems
- Protocol translation between different automation levels
- Bidirectional data exchange scenarios
- Adapter-based system integration

⚖️ Comparison with Similar Blocks

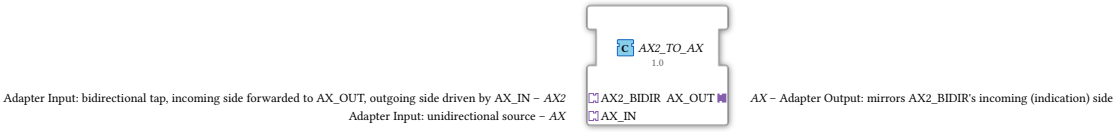
Compared to simple conversion blocks, AX2_BOOL_TO_X offers:

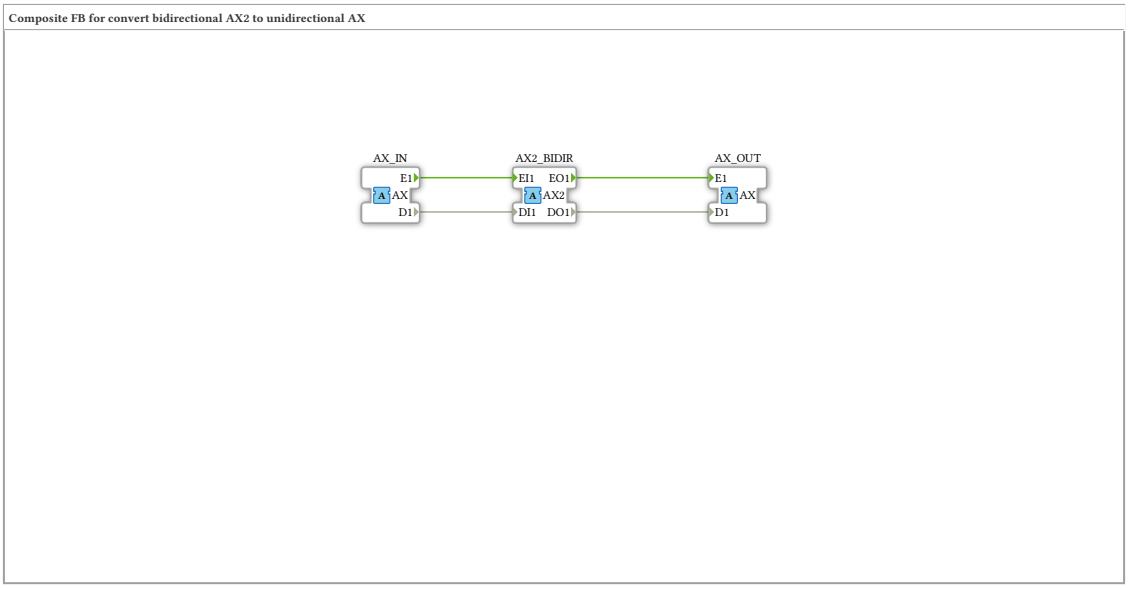
- Bidirectional functionality
- Integrated adapter interface
- Complete event handling
- Higher level of abstraction Composite Design

Conclusion

AX2_BOOL_TO_X is a specialized composite function block that efficiently bridges BOOL data and the AX2 adapter protocol. Its bidirectional architecture and integrated event handling make it ideal for systems requiring seamless integration between different protocol layers.

AX2_TO_AX





• – ○ • – ○ • – ○ •

Introduction

The AX2_TO_AX function block is the mirror image of AX_TO_AX2: here the bidirectional AX2 tap sits as a socket on the left (input) side, complemented by a second unidirectional AX socket and a unidirectional AX plug.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AX2_BIDIR** (Socket): bidirectional tap whose incoming side is forwarded to `AX_OUT` and whose outgoing side is fed by `AX_IN`, of type `adapter::types::bidirectional::AX2`
- **AX_IN** (Socket): unidirectional BOOL source, of type `adapter::types::unidirectional::AX`
- **AX_OUT** (Plug): mirrors the incoming (indication) side of `AX2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::AX`

Functionality

The block operates in two directions at once: whatever the socket `AX2_BIDIR` receives on its indication side (`EO1`/`DO1` — i.e. whatever a bidirectional peer attached to AX2_BIDIR reports) is output unchanged as a standalone unidirectional signal via the plug `AX_OUT`. Conversely, whatever arrives at the socket `AX_IN` (`E1`/`D1`) is published onto the request side of `AX2_BIDIR` (`EI1`/`DI1`) — so AX2_BIDIR forwards the same signal to the attached bidirectional peer.

Technical Details

- Three adapters instead of two: the direction received from the tap and the separately injected direction are deliberately kept apart rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer

- `AX_OUT` only reflects what is reported via `AX2_BIDIR` by an attached bidirectional peer — without such a peer, `AX_OUT` stays idle

State Overview

The block is stateless:

- AX2_BIDIR.EO1 ⟶ AX_OUT.E1, AX2_BIDIR.DO1 ⟶ AX_OUT.D1
- AX_IN.E1 ⟶ AX2_BIDIR.EI1, AX_IN.D1 ⟶ AX2_BIDIR.DI1

Application Scenarios

- Reading the status of a bidirectional AX2 peer as a plain unidirectional AX signal, combined with injecting a separate unidirectional command signal
- Systems where a bidirectional peer needs to be monitored without converting the rest of the control logic to AX2
- Bridging a bidirectional AX2 zone and a unidirectional AX zone

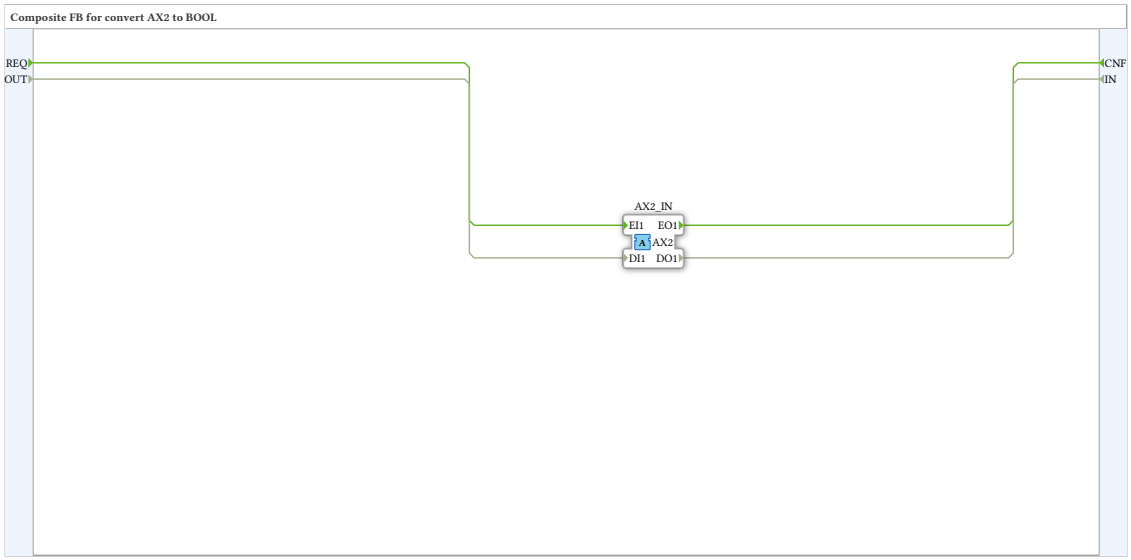
⚖️ Comparison with Similar Blocks

AX_TO_AX2 is the mirror image of this block — there, `AX2_BIDIR` sits as a plug on the right side. For a pure, non-tapping 1:1 conversion between AX2 and a plain BOOL signal, see AX2_X_TO_BOOL/ AX2_BOOL_TO_X. The 2-channel A2X2/A2X counterpart is A2X2_TO_A2X.

Conclusion

AX2_TO_AX is not a simple converter but a tap point: it makes the state of a bidirectional AX2 peer visible as a standalone unidirectional AX signal while simultaneously injecting a separate unidirectional signal into the bidirectional side.

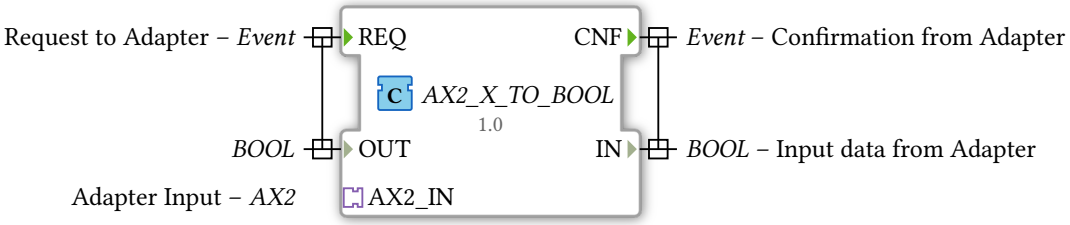
AX2_X_TO_BOOL



• – ◦ • – ◦ • – ◦ •

Introduction

The AX2_X_TO_BOOL function block is a composite function block for converting AX2 data to BOOL values. It serves as a bidirectional adapter for data type conversion between different interfaces in automation systems.



Interface Structure

Event Inputs

- **REQ** (Request to Adapter): Initiates the conversion request to the adapter

Event Outputs

- **CNF** (Confirmation from Adapter): Confirms the completion of the conversion from the adapter

Data Inputs

- **OUT** (BOOL): Output data for transfer to the adapter

Data Outputs

- **IN** (BOOL): Input data from the adapter after successful conversion

Adapter

- **AX2_IN** (adapter::types::bidirectional::AX2): Bidirectional AX2 adapter for data type conversion

Functionality

The function block receives a conversion request via the REQ event and forwards it to the AX2 adapter. The adapter converts AX2 data to BOOL values. After successful conversion, the CNF event containing the converted data is output at the IN output. Simultaneously, BOOL data can be returned to the adapter via the OUT input.

Technical Features

- Bidirectional data transfer between AX2 and BOOL data types
- Composite function block with integrated adapter
- Automatic event forwarding between interfaces
- Positioned elements for optimized display in the development environment

State Transitions

1. **Wait State**: No active conversion
2. **Active State**: On REQ event - data is forwarded to the adapter
3. **Confirmation State**: Upon completion of the conversion - CNF event is triggered

Application Scenarios

- Integration of AX2 sensor data into BOOL-based control systems
- Data type conversion in mixed automation environments
- Bidirectional communication between different protocols
- Adapter solutions for existing systems with different data types

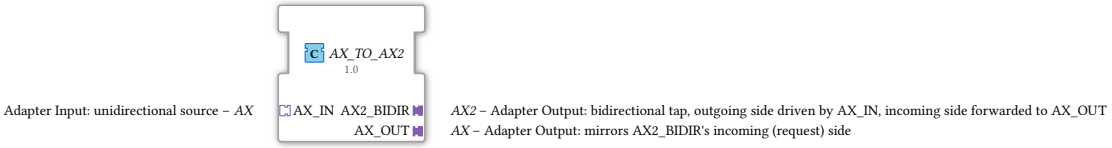
⚖️ Comparison with similar blocks

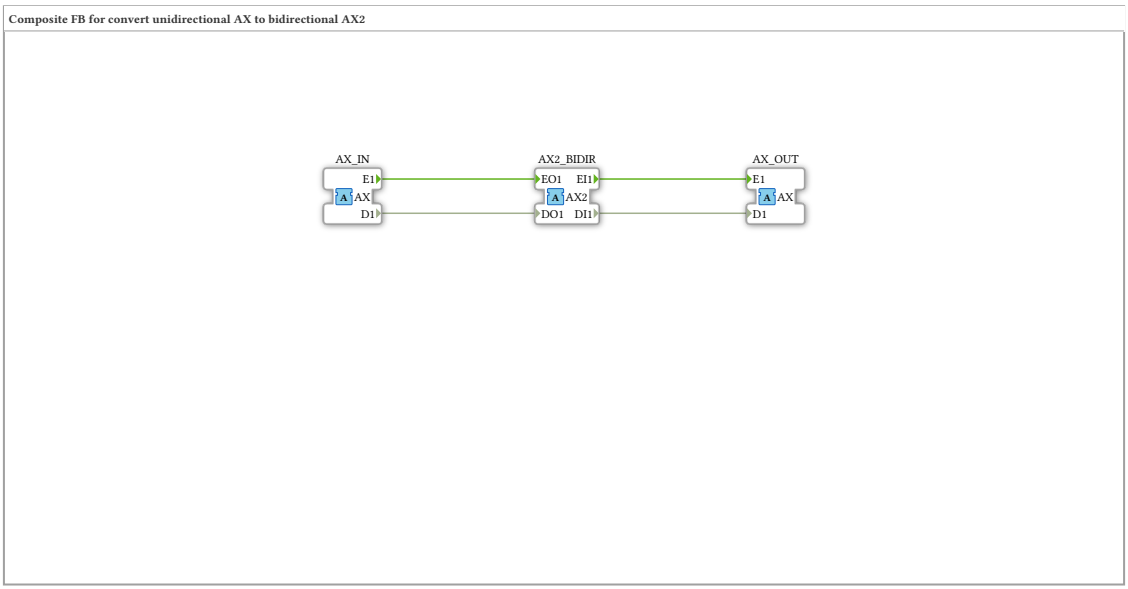
Compared to simple conversion blocks, AX2_X_TO_BOOL offers bidirectional functionality with an integrated adapter interface. While simple converters typically only work in one direction, this block enables bidirectional data exchange.

Conclusion

The AX2_X_TO_BOOL function block provides a robust solution for bidirectional conversion between AX2 and BOOL data types. Its integrated adapter architecture and clear event handling make it particularly suitable for complex automation applications with mixed data type environments.

AX_TO_AX2





• – ○ • – ○ • – ○ •

Introduction

The AX_TO_AX2 function block inserts a bidirectional AX2 tap into a unidirectional AX chain. Rather than a simple 1:1 conversion, it uses **three** adapters: an incoming unidirectional AX signal is both published onto the bidirectional AX2 interface and — whatever AX2 returns on its request side — output again as a standalone unidirectional AX signal.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AX_IN** (Socket): unidirectional BOOL source, of type `adapter::types::unidirectional::AX`
- **AX2_BIDIR** (Plug): bidirectional tap whose outgoing side is fed by `AX_IN` and whose incoming side is forwarded to `AX_OUT`, of type `adapter::types::bidirectional::AX2`
- **AX_OUT** (Plug): mirrors the incoming (request) side of `AX2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::AX`

Functionality

The block operates in two directions at once: whatever arrives at the socket `AX_IN` (`E1`/`D1`) is published directly onto the outgoing side of `AX2_BIDIR` (`EO1`/`DO1`) — so AX2_BIDIR reports outward the same signal coming in via AX_IN. Conversely, whatever `AX2_BIDIR` receives on its request side (`EI1`/`DI1` — i.e.

whatever a bidirectional peer attached to AX2_BIDIR "sends back") is output unchanged as a standalone unidirectional signal via the plug `AX_OUT`. AX2_BIDIR thus acts as a tap point that doesn't break the unidirectional chain but extends it with a bidirectional observation/injection point.

Technical Details

- Three adapters instead of two: the incoming signal direction and the direction coming back from the tap are deliberately kept separate rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `AX_OUT` only reflects what comes back via `AX2_BIDIR` from an attached bidirectional peer — without such a peer, `AX_OUT` stays idle

State Overview

The block is stateless:

- AX_IN.E1 ⟶ AX2_BIDIR.EO1, AX_IN.D1 ⟶ AX2_BIDIR.DO1
- AX2_BIDIR.EI1 ⟶ AX_OUT.E1, AX2_BIDIR.DI1 ⟶ AX_OUT.D1

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional AX signal chain without breaking it
- Systems where a bidirectional peer (e.g. an operator panel) needs to both display the status of a unidirectional signal source and send commands back
- Bridging a unidirectional AX zone and a bidirectional AX2 zone

⚖️ Comparison with Similar Blocks

AX2_TO_AX is the mirror image of this block — there, `AX2_BIDIR` sits as a socket on the left side. For a pure, non-tapping 1:1 conversion between a plain BOOL signal and AX2, see AX2_BOOL_TO_X/ AX2_X_TO_BOOL. The 2-channel A2X/A2X2 counterpart is A2X_TO_A2X2.

Conclusion

AX_TO_AX2 is not a simple converter but a tap point: it makes a unidirectional AX signal visible at a bidirectional AX2 point without breaking the original unidirectional signal chain, while forwarding the request signal received there as a standalone AX signal.

BOOL

🎧 Podcast

- George Boole: How an autodidact laid the foundation of our digital world with 0 and 1

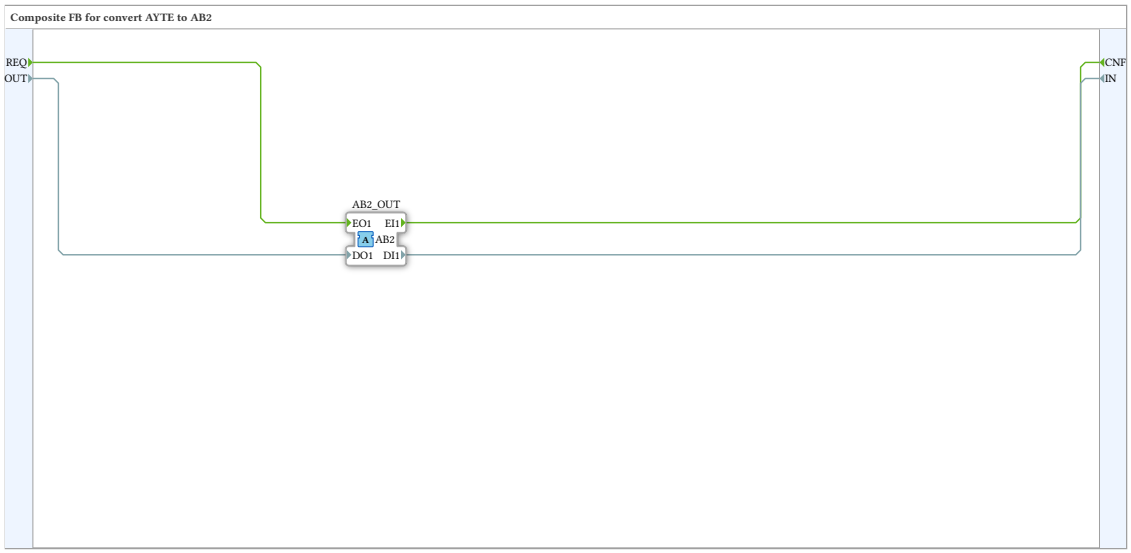
🛠️ Related exercises

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

Numerical Index (bidirectional)

Here you will find all exercises from the `bidirectional` workspace in ascending order. ...#

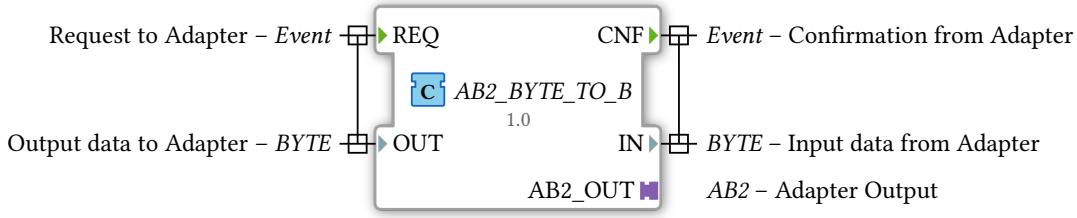
AB2_BYTE_TO_B



• – ○ • – ○ • – ○ •

Introduction

The AB2_BYTE_TO_B function block is a composite function block used for conversion between BYTE data types and the AB2 adapter. It acts as a bidirectional converter and enables communication between system components with different interfaces.



Interface Structure

Event Inputs

- **REQ** (Request to Adapter): Starts the conversion and data transfer to the adapter

Event Outputs

- **CNF** (Confirmation from Adapter): Signals the successful completion of the conversion and data transfer from the adapter

Data Inputs

- **OUT** (BYTE): Output data to be transferred to the adapter

Data Outputs

- **IN** (BYTE): Input data received by the adapter

Adapter

- **AB2_OUT** (adapter::types::bidirectional::AB2): Bidirectional adapter for data conversion

Functionality

The function block receives a conversion request via the REQ event and forwards the incoming BYTE data (OUT) to the AB2 adapter. The adapter performs the actual conversion and, upon successful completion, sends back a CNF event along with the converted BYTE data (IN). ...

Technical Features

- Composite function block with integrated adapter
- Bidirectional data transmission
- Uses BYTE as the primary data type
- Positioned interfaces for optimized display in the development environment

State Transitions

1. **Idle State**: Waits for REQ event
2. **Processing State**: Upon REQ, data is forwarded to AB2_OUT
3. **Acknowledgement State**: Upon feedback from the adapter, CNF is triggered

Application Scenarios

- Integration of BYTE-based components into AB2-compatible systems
- Protocol conversion in distributed automation systems
- Bidirectional data exchange interfaces
- Adapter-based system integration

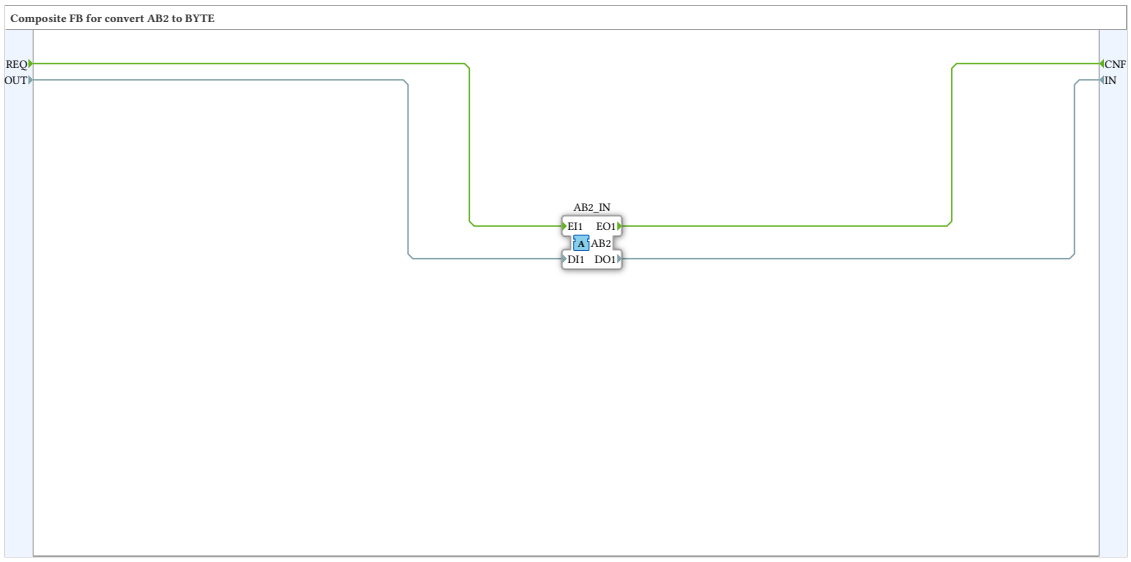
⚖️ Comparison with Similar Blocks

Compared to simple conversion blocks, AB2_BYTE_TO_B offers bidirectional functionality with adapter integration. While simple converters typically only operate in one direction, this block enables communication in both directions via standardized adapter interfaces.

Conclusion

AB2_BYTE_TO_B is a specialized composite function block that provides reliable bidirectional conversion between BYTE data and the AB2 adapter standard. Its structured interface and clear event control make it ideal for use in complex automation systems with standardized communication protocols.

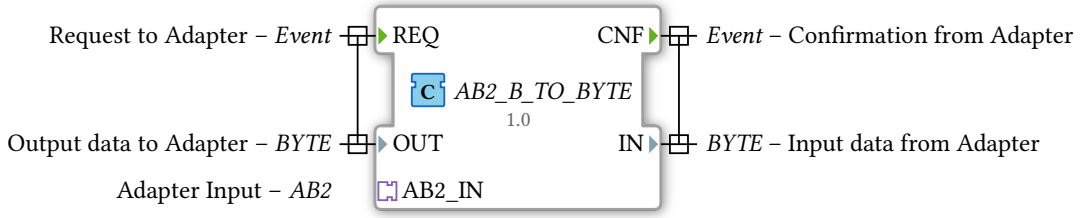
AB2_B_TO_BYTE



• – ○ • – ○ • – ○ •

Introduction

The AB2_B_TO_BYTE is a composite function block used for bidirectional conversion between AB2 and BYTE data types. This function block acts as an adapter converter, enabling the seamless integration of different data types into automation systems.



Interface Structure

Event Inputs

- **REQ** (Request to Adapter): Starts the conversion operation and requests the data output

Event Outputs

- **CNF** (Confirmation from Adapter): Signals the successful completion of the conversion operation

Data Inputs

- **OUT** (BYTE): Output data to be sent to the adapter

Data Outputs

- **IN** (BYTE): Input data received by the adapter

Adapter

- **AB2_IN** (adapter::types::bidirectional::AB2): Bidirectional AB2 adapter for data type conversion

Functionality

This function block operates as a bidirectional converter between AB2 and BYTE data types. Upon receiving a REQ event, the BYTE data at OUT is converted via the AB2_IN adapter. After successful conversion, the CNF event is triggered, and the converted data is output to IN.

The internal networking shows the following signal flows:

- REQ → AB2_IN.EI1 (Adapter input event)
- AB2_IN.EO1 → CNF (Adapter output event)
- OUT → AB2_IN.DI1 (Adapter data input)
- AB2_IN.DO1 → IN (Adapter data output)

Technical Features

- Implemented as a composite function block with an integrated adapter
- Supports bidirectional data conversion
- Positioning attributes for graphical representation in the development environment
- Embedded in the “adapter::conversion::bidirectional” package

State Overview

The function block goes through the following states:

1. **Ready**: Waiting for a REQ event
2. **Conversion in progress**: Processing data via the AB2 adapter
3. **Acknowledgement**: Sends a CNF upon successful completion

Application Scenarios

- Integration of AB2 components into BYTE-based Control Systems
- Data Type Conversion in Heterogeneous Automation Networks
- Adapter Solutions for Legacy Systems with AB2 Interfaces
- Protocol Conversion Between Different Communication Standards

⚖️ Comparison with Similar Function Blocks

Compared to simple conversion blocks, AB2_B_TO_BYTE offers:

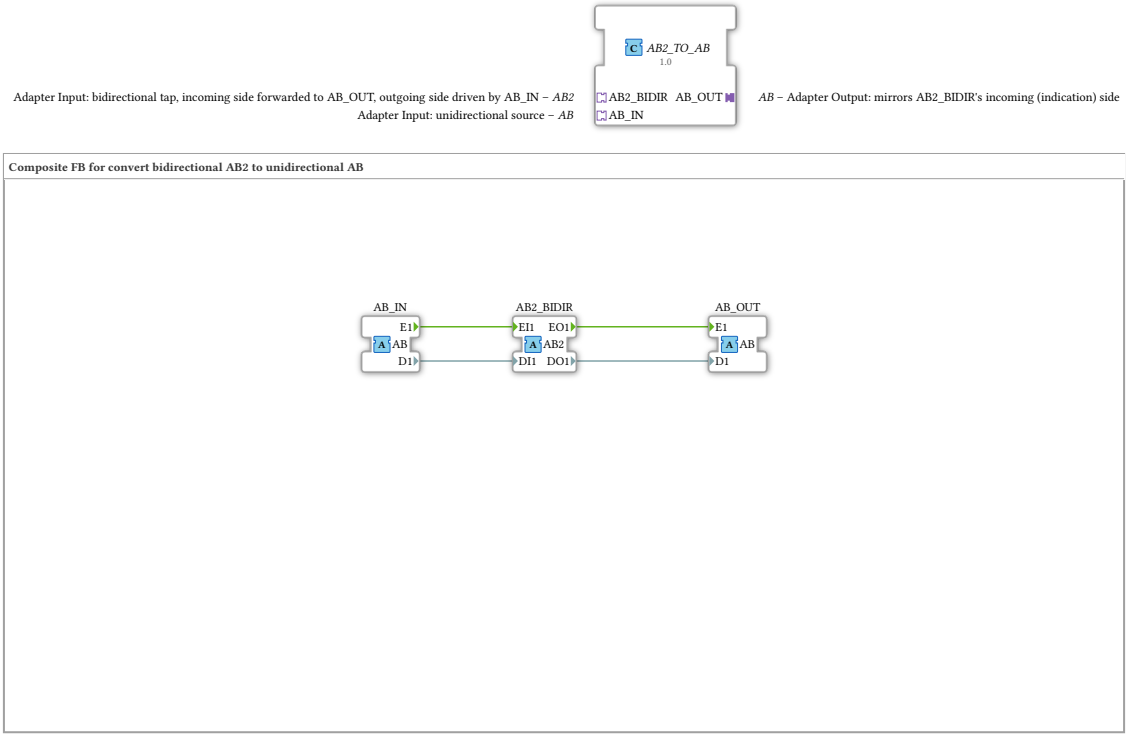
- Bidirectional functionality in a single block
- Integrated adapter interface for AB2 types

- Composite architecture with extended functionality
- Standardized event handling with REQ/CNF pair

Conclusion

The AB2_B_TO_BYTE function block provides a robust solution for bidirectional conversion between AB2 and BYTE data types. Its composite architecture with integrated adapter enables easy integration into existing automation systems and offers reliable data type compatibility.

AB2_TO_AB



• – ○ • – ○ • – ○ •

Introduction

The AB2_TO_AB function block is the mirror image of AB_TO_AB2: here the bidirectional AB2 tap sits as a socket on the left (input) side, complemented by a second unidirectional AB socket and a unidirectional AB plug.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AB2_BIDIR** (Socket): bidirectional tap whose incoming side is forwarded to `AB_OUT` and whose outgoing side is fed by `AB_IN`, of type `adapter::types::bidirectional::AB2`
- **AB_IN** (Socket): unidirectional BYTE source, of type `adapter::types::unidirectional::AB`

- **AB_OUT** (Plug): mirrors the incoming (indication) side of `AB2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::AB`

Functionality

The block operates in two directions at once: whatever the socket `AB2_BIDIR` receives on its indication side (`EO1`/`DO1` — i.e. whatever a bidirectional peer attached to AB2_BIDIR reports) is output unchanged as a standalone unidirectional signal via the plug `AB_OUT`. Conversely, whatever arrives at the socket `AB_IN` (`E1`/`D1`) is published onto the request side of `AB2_BIDIR` (`EI1`/`DI1`) — so AB2_BIDIR forwards the same signal to the attached bidirectional peer.

Technical Details

- Three adapters instead of two: the direction received from the tap and the separately injected direction are deliberately kept apart rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `AB_OUT` only reflects what is reported via `AB2_BIDIR` by an attached bidirectional peer — without such a peer, `AB_OUT` stays idle

State Overview

The block is stateless:

- AB2_BIDIR.EO1 ⟶ AB_OUT.E1, AB2_BIDIR.DO1 ⟶ AB_OUT.D1
- AB_IN.E1 ⟶ AB2_BIDIR.EI1, AB_IN.D1 ⟶ AB2_BIDIR.DI1

Application Scenarios

- Reading the status of a bidirectional AB2 peer as a plain unidirectional AB signal, combined with injecting a separate unidirectional command signal
- Systems where a bidirectional peer needs to be monitored without converting the rest of the control logic to AB2
- Bridging a bidirectional AB2 zone and a unidirectional AB zone

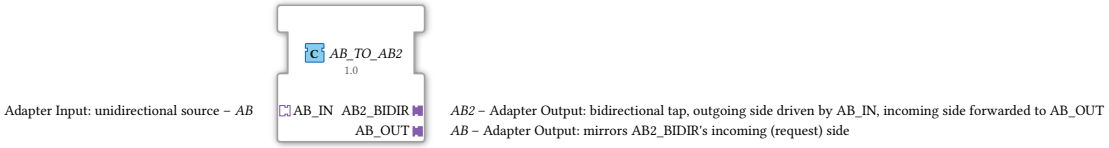
⚖️ Comparison with Similar Blocks

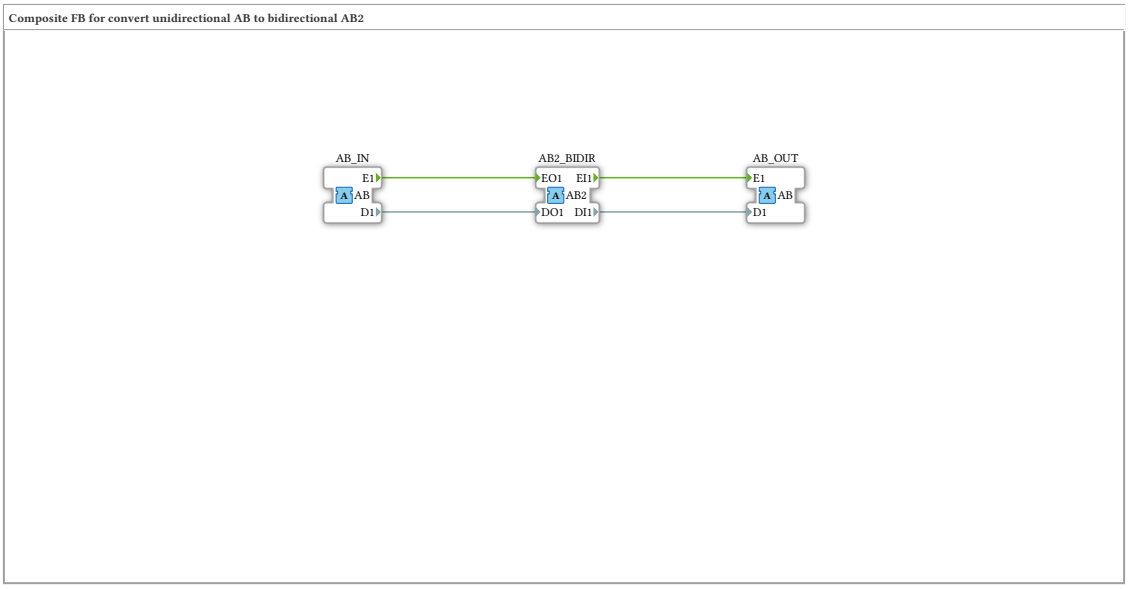
AB_TO_AB2 is the mirror image of this block — there, `AB2_BIDIR` sits as a plug on the right side. For a pure, non-tapping 1:1 conversion between AB2 and a plain BYTE signal, see AB2_BYTE_TO_B/ AB2_B_TO_BYTE.

Conclusion

AB2_TO_AB is not a simple converter but a tap point: it makes the state of a bidirectional AB2 peer visible as a standalone unidirectional AB signal while simultaneously injecting a separate unidirectional signal into the bidirectional side.

AB_TO_AB2





• – ○ • – ○ • – ○ •

Introduction

The AB_TO_AB2 function block inserts a bidirectional AB2 tap into a unidirectional AB chain. Rather than a simple 1:1 conversion, it uses **three** adapters: an incoming unidirectional AB signal is both published onto the bidirectional AB2 interface and — whatever AB2 returns on its request side — output again as a standalone unidirectional AB signal.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AB_IN** (Socket): unidirectional BYTE source, of type `adapter::types::unidirectional::AB`
- **AB2_BIDIR** (Plug): bidirectional tap whose outgoing side is fed by `AB_IN` and whose incoming side is forwarded to `AB_OUT`, of type `adapter::types::bidirectional::AB2`
- **AB_OUT** (Plug): mirrors the incoming (request) side of `AB2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::AB`

Functionality

The block operates in two directions at once: whatever arrives at the socket `AB_IN` (`E1`/`D1`) is published directly onto the outgoing side of `AB2_BIDIR` (`EO1`/`DO1`) — so AB2_BIDIR reports outward the same signal coming in via AB_IN. Conversely, whatever `AB2_BIDIR` receives on its request side (`EI1`/`DI1` — i.e. whatever a bidirectional peer attached to AB2_BIDIR "sends back") is output unchanged as a standalone unidirectional signal via the plug `AB_OUT`. AB2_BIDIR thus acts as a tap point that doesn't break the unidirectional chain but extends it with a bidirectional observation/injection point.

Technical Details

- Three adapters instead of two: the incoming signal direction and the direction coming back from the tap are deliberately kept separate rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `AB_OUT` only reflects what comes back via `AB2_BIDIR` from an attached bidirectional peer — without such a peer, `AB_OUT` stays idle

State Overview

The block is stateless:

- AB_IN.E1 ⟶ AB2_BIDIR.EO1, AB_IN.D1 ⟶ AB2_BIDIR.DO1
- AB2_BIDIR.EI1 ⟶ AB_OUT.E1, AB2_BIDIR.DI1 ⟶ AB_OUT.D1

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional AB signal chain without breaking it
- Systems where a bidirectional peer (e.g. an operator panel) needs to both display the status of a unidirectional byte signal source and send commands back
- Bridging a unidirectional AB zone and a bidirectional AB2 zone

⚖️ Comparison with Similar Blocks

AB2_TO_AB is the mirror image of this block — there, `AB2_BIDIR` sits as a socket on the left side. For a pure, non-tapping 1:1 conversion between a plain BYTE signal and AB2, see AB2_B_TO_BYTE/ AB2_BYTE_TO_B.

Conclusion

AB_TO_AB2 is not a simple converter but a tap point: it makes a unidirectional AB signal visible at a bidirectional AB2 point without breaking the original unidirectional signal chain, while forwarding the request signal received there as a standalone AB signal.

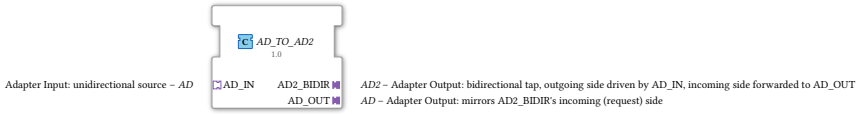
BYTE

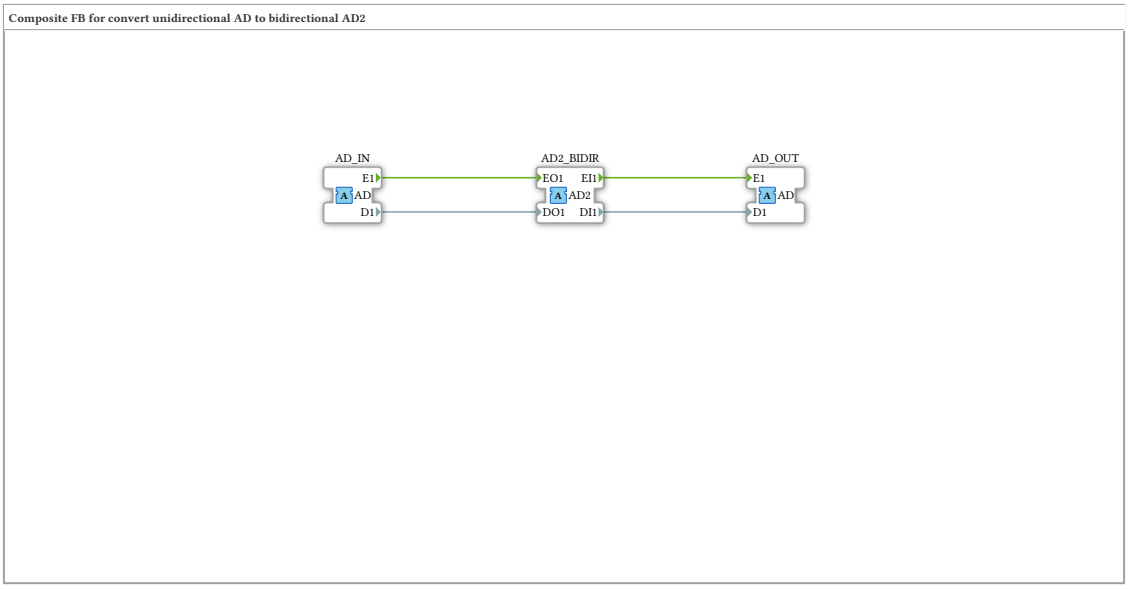
🛠️ Related exercises

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

DWORD

AD_TO_AD2





• – ○ • – ○ • – ○ •

Introduction

The AD_TO_AD2 function block inserts a bidirectional AD2 tap into a unidirectional AD chain. Rather than a simple 1:1 conversion, it uses **three** adapters: an incoming unidirectional AD signal is both published onto the bidirectional AD2 interface and — whatever AD2 returns on its request side — output again as a standalone unidirectional AD signal.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AD_IN** (Socket): unidirectional DWORD source, of type `adapter::types::unidirectional::AD`
- **AD2_BIDIR** (Plug): bidirectional tap whose outgoing side is fed by `AD_IN` and whose incoming side is forwarded to `AD_OUT`, of type `adapter::types::bidirectional::AD2`
- **AD_OUT** (Plug): mirrors the incoming (request) side of `AD2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::AD`

Functionality

The block operates in two directions at once: whatever arrives at the socket `AD_IN` (`E1`/`D1`) is published directly onto the outgoing side of `AD2_BIDIR` (`EO1`/`DO1`) — so AD2_BIDIR reports outward the same signal coming in via AD_IN. Conversely, whatever `AD2_BIDIR` receives on its request side (`EI1`/`DI1` — i.e. whatever a bidirectional peer attached to AD2_BIDIR "sends back") is output unchanged as a standalone unidirectional signal via the plug `AD_OUT`. AD2_BIDIR thus acts as a tap point that doesn't break the unidirectional chain but extends it with a bidirectional observation/injection point.

Technical Details

- Three adapters instead of two: the incoming signal direction and the direction coming back from the tap are deliberately kept separate rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `AD_OUT` only reflects what comes back via `AD2_BIDIR` from an attached bidirectional peer — without such a peer, `AD_OUT` stays idle

State Overview

The block is stateless:

- AD_IN.E1 ⟶ AD2_BIDIR.EO1, AD_IN.D1 ⟶ AD2_BIDIR.DO1
- AD2_BIDIR.EI1 ⟶ AD_OUT.E1, AD2_BIDIR.DI1 ⟶ AD_OUT.D1

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional AD signal chain without breaking it
- Systems where a bidirectional peer (e.g. an operator panel) needs to both display the status of a unidirectional byte signal source and send commands back
- Bridging a unidirectional AD zone and a bidirectional AD2 zone

⚖️ Comparison with Similar Blocks

AD_TO_AD2 belongs to the conversion family of tap blocks (analogous to AB_TO_AB2 for BYTE): a unidirectional AD signal is tapped at a bidirectional AD2 point.

Conclusion

AD_TO_AD2 is not a simple converter but a tap point: it makes a unidirectional AD signal visible at a bidirectional AD2 point without breaking the original unidirectional signal chain, while forwarding the request signal received there as a standalone AD signal.

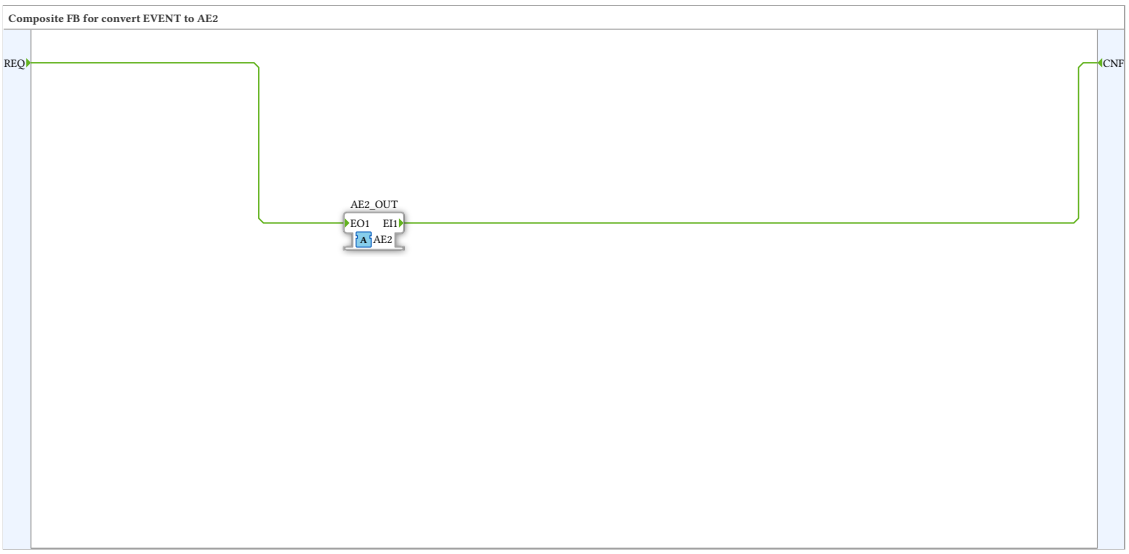
🛠️ Related Exercises

- Uebung_094b5_AX

Numerical Index (bidirectional)

Here you will find all exercises from the `bidirectional` workspace in ascending order. …

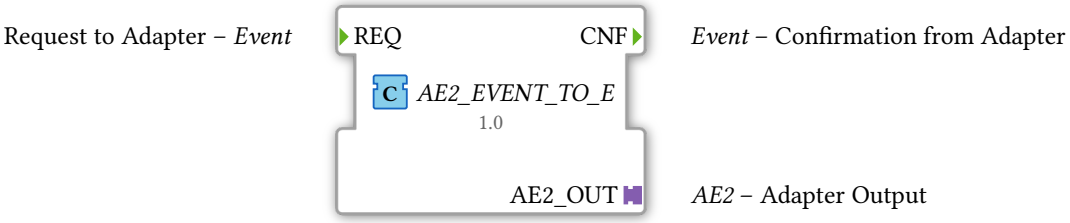
AE2_EVENT_TO_E



• – ◦ • – ◦ • – ◦ •

Introduction

The AE2_EVENT_TO_E function block is a composite function block used to convert EVENT signals into the AE2 adapter format. It enables bidirectional communication between standard events and the specific AE2 adapter.



Interface Structure

Event Inputs

- **REQ** (Request to Adapter): Starts the conversion and sends a request to the AE2 output adapter

Event Outputs

- **CNF** (Confirmation from Adapter): Confirms the successful completion of the conversion and receipt of the response from the AE2 adapter

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

- **AE2_OUT** (Adapter Output): Bidirectional AE2 output adapter for communication with AE2-compatible systems

Functionality

This function block forwards incoming REQ events directly to the AE2_OUT adapter (EO1 port). Simultaneously, incoming acknowledgments from the AE2 adapter (EI1 port) are output as CNF events. This enables transparent conversion between the standard EVENT format and the specific AE2 adapter protocol.

Technical Features

- Implemented as a composite function block
- Supports bidirectional communication
- Uses the AE2 adapter type from the adapter::types::bidirectional package
- Positioned adapter interface for optimized visualization

State Overview

The function block features a simple state machine:

1. **Idle State**: Waits for a REQ event
2. **Request State**: After REQ input, sends to AE2_OUT.EO1
3. **Confirmation State**: After receiving AE2_OUT.EI1, outputs CNF

Application Scenarios

- Integration of EVENT-based systems into AE2-compatible environments
- Protocol conversion in distributed automation systems
- Bidirectional communication bridges between different automation components
- Adapter-based system integration

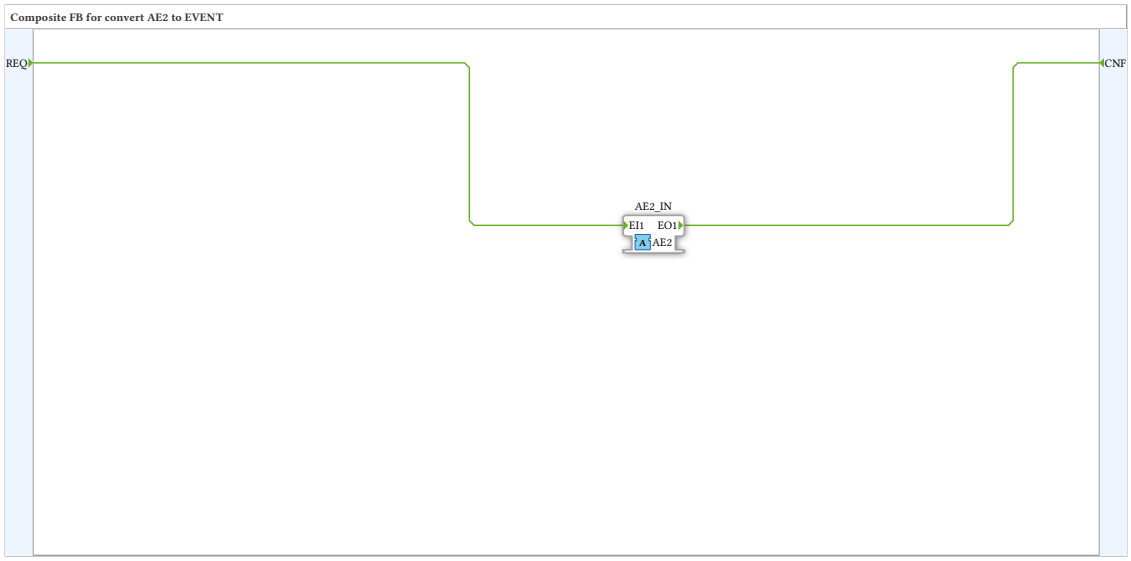
⚖️ Comparison with Similar Blocks

Compared to simple EVENT converters, this offers AE2_EVENT_TO_E provides specific AE2 adapter support with bidirectional communication. While standard converters often operate only unidirectionally, this block enables complete request-response cycles with the AE2 protocol.

Conclusion

The AE2_EVENT_TO_E function block offers a reliable solution for converting between standard event signals and the AE2 adapter protocol. Its bidirectional functionality and ease of integration make it particularly suitable for systems requiring seamless communication between different protocol environments.

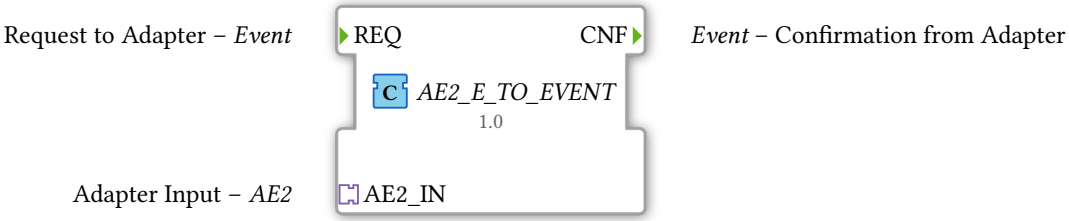
AE2_E_TO_EVENT



• – ◦ • – ◦ • – ◦ •

Introduction

The AE2_E_TO_EVENT function block is a composite function block used to convert AE2 adapter signals into events. It enables bidirectional communication between adapter-based components and event-driven function blocks.



Interface Structure

Event Inputs

- **REQ** (Request to Adapter): Starts the conversion and sends a request to the adapter

Event Outputs

- **CNF** (Confirmation from Adapter): Confirms successful completion of the conversion

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapter

- **AE2_IN** (Adapter Input): Bidirectional AE2 adapter input for communication with AE2-compatible components

Functionality

This function block acts as a simple converter between the AE2 adapter protocol and the standard 4diac event system. Upon receiving a REQ event, a request is sent via the AE2 adapter. The adapter's response is received via the EO1 signal and output as a CNF event.

Technical Features

- Implemented as a composite function block
- Uses bidirectional AE2 adapter communication
- Simple event forwarding without data manipulation
- The CNF output position is fixed at Y position 46

State Overview

The function block has two main states:

1. **Wait State**: No active conversion
2. **Active State**: Conversion in progress (from REQ input to CNF output)

Application Scenarios

- Integration of AE2-compatible devices into 4diac controllers
- Bridge functionality between different communication protocols
- Simple event conversion in automation systems
- Protocol conversion in distributed control systems

⚖️ Comparison with Similar Blocks

Compared to other adapter converters, AE2_E_TO_EVENT stands out due to its specific focus on the AE2 adapter protocol. While general adapter converters support various protocols, this module is specifically optimized for bidirectional AE2 communication.

Conclusion

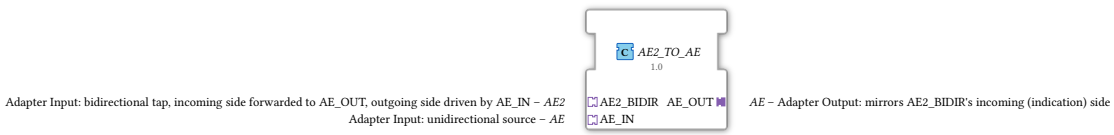
The AE2_E_TO_EVENT function block offers a reliable and simple solution for integrating AE2 adapters into event-driven 4diac systems. Its clear interface definition and straightforward functionality make it a practical tool for protocol conversions in industrial automation applications.

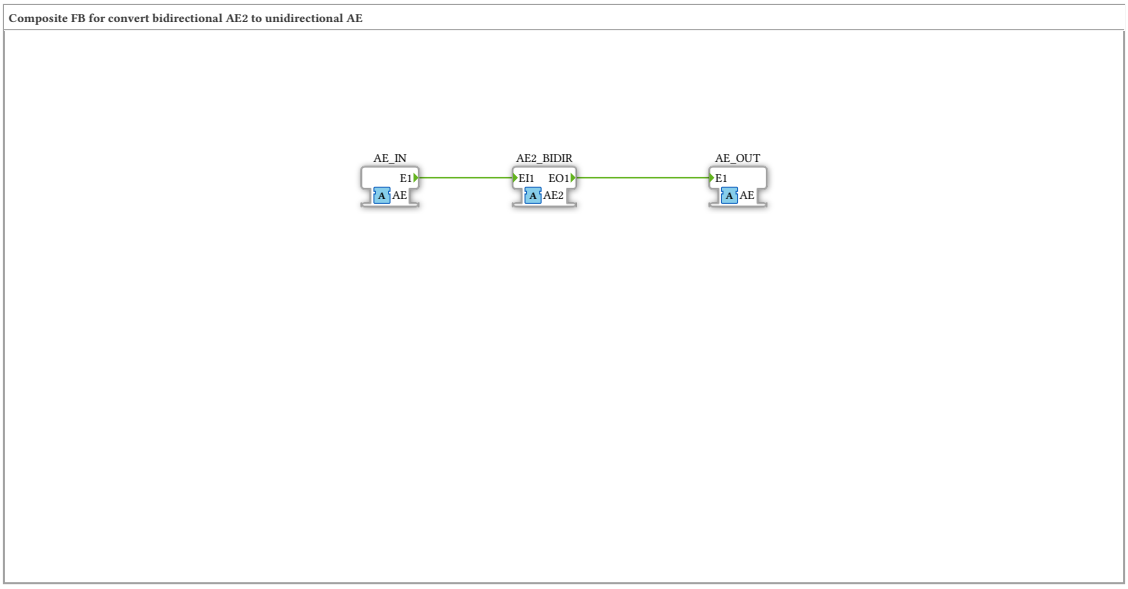
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AE2_TO_AE





• – ○ • – ○ • – ○ •

Introduction

The AE2_TO_AE function block is the mirror image of AE_TO_AE2: here the bidirectional AE2 tap sits as a socket on the left (input) side, complemented by a second unidirectional AE socket and a unidirectional AE plug.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no data inputs or outputs at all — AE and AE2 carry only events, never payload data.

Data Outputs

The function block has no data inputs or outputs at all — AE and AE2 carry only events, never payload data.

Adapters

- **AE2_BIDIR** (Socket): bidirectional tap whose incoming side is forwarded to `AE_OUT` and whose outgoing side is fed by `AE_IN`, of type `adapter::types::bidirectional::AE2`
- **AE_IN** (Socket): unidirectional EVENT source, of type `adapter::types::unidirectional::AE`
- **AE_OUT** (Plug): mirrors the incoming (indication) side of `AE2_BIDIR` as a unidirectional event, of type `adapter::types::unidirectional::AE`

Functionality

The block operates in two directions at once: whatever the socket `AE2_BIDIR` receives on its indication side (`EO1` — i.e. whatever a bidirectional peer attached to AE2_BIDIR reports) is output unchanged as a standalone unidirectional event via the plug `AE_OUT`. Conversely, whatever arrives at the socket `AE_IN` (`E1`) is published onto the request side of `AE2_BIDIR` (`EI1`) — so AE2_BIDIR forwards the same event to the attached bidirectional peer. Since AE/AE2 are pure event adapters with no payload data, there are no DataConnections at all.

Technical Details

- Three adapters instead of two: the direction received from the tap and the separately injected direction are deliberately kept apart rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- No data transport: AE/AE2 carry only events, so this block has no DataConnections
- `AE_OUT` only reflects what is reported via `AE2_BIDIR` by an attached bidirectional peer — without such a peer, `AE_OUT` stays idle

State Overview

The block is stateless:

- AE2_BIDIR.EO1 ⟶ AE_OUT.E1
- AE_IN.E1 ⟶ AE2_BIDIR.EI1

Application Scenarios

- Reading a bidirectional AE2 peer as a plain unidirectional AE event, combined with injecting a separate unidirectional event
- Systems where a bidirectional peer needs to be monitored without converting the rest of the control logic to AE2
- Bridging a bidirectional AE2 zone and a unidirectional AE zone

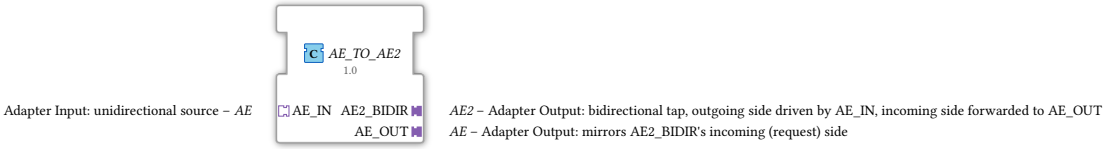
⚖️ Comparison with Similar Blocks

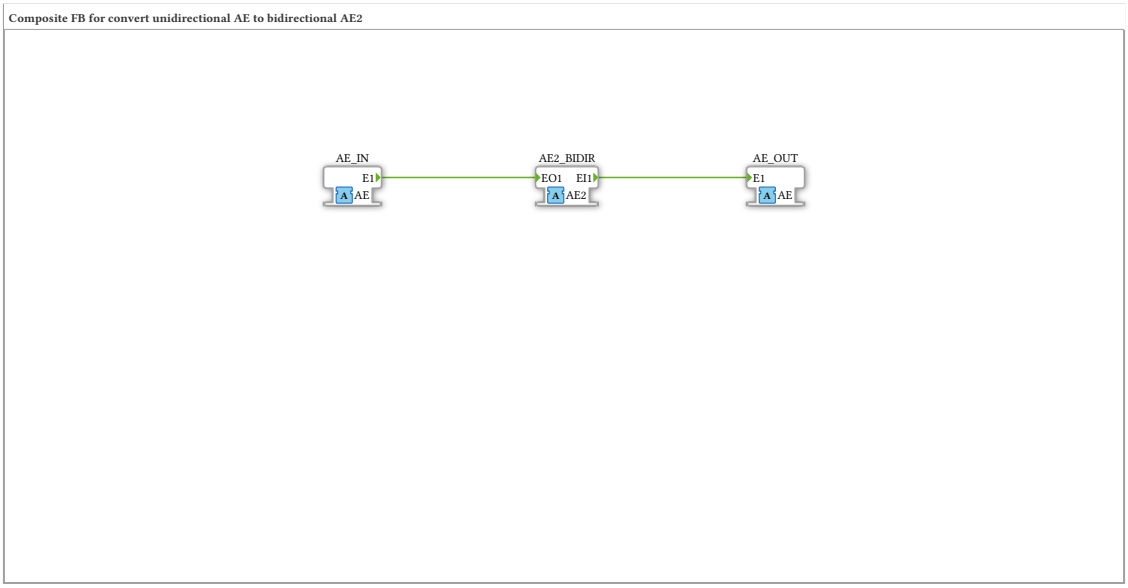
AE_TO_AE2 is the mirror image of this block — there, `AE2_BIDIR` sits as a plug on the right side. For a pure, non-tapping 1:1 conversion between AE2 and a plain EVENT signal, see AE2_E_TO_EVENT/ AE2_EVENT_TO_E. For two coupled events (SET/RESET), see ASR2_TO_ASR.

Conclusion

AE2_TO_AE is not a simple converter but a tap point: it makes the state of a bidirectional AE2 peer visible as a standalone unidirectional AE event while simultaneously injecting a separate unidirectional event into the bidirectional side.

AE_TO_AE2





• – ○ • – ○ • – ○ •

Introduction

The AE_TO_AE2 function block inserts a bidirectional AE2 tap into a unidirectional AE event chain. Rather than a simple 1:1 conversion, it uses **three** adapters: an incoming unidirectional AE event is both published onto the bidirectional AE2 interface and — whatever AE2 returns on its request side — output again as a standalone unidirectional AE event.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no data inputs or outputs at all — AE and AE2 carry only events, never payload data.

Data Outputs

The function block has no data inputs or outputs at all — AE and AE2 carry only events, never payload data.

Adapters

- **AE_IN** (Socket): unidirectional EVENT source, of type `adapter::types::unidirectional::AE`
- **AE2_BIDIR** (Plug): bidirectional tap whose outgoing side is fed by `AE_IN` and whose incoming side is forwarded to `AE_OUT`, of type `adapter::types::bidirectional::AE2`
- **AE_OUT** (Plug): mirrors the incoming (request) side of `AE2_BIDIR` as a unidirectional event, of type `adapter::types::unidirectional::AE`

Functionality

The block operates in two directions at once: whatever arrives at the socket `AE_IN` (`E1`) is published directly onto the outgoing side of `AE2_BIDIR` (`EO1`) — so AE2_BIDIR reports outward the same event coming in via AE_IN. Conversely, whatever `AE2_BIDIR` receives on its request side (`EI1` — i.e. whatever a bidirectional peer attached to AE2_BIDIR "sends back") is output unchanged as a standalone unidirectional event via the plug `AE_OUT`. Since AE/AE2 are pure event adapters with no payload data, there are no DataConnections at all.

Technical Details

- Three adapters instead of two: the incoming event direction and the direction coming back from the tap are deliberately kept separate rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- No data transport: AE/AE2 carry only events, so this block has no DataConnections
- `AE_OUT` only reflects what comes back via `AE2_BIDIR` from an attached bidirectional peer — without such a peer, `AE_OUT` stays idle

State Overview

The block is stateless:

- AE_IN.E1 → AE2_BIDIR.EO1
- AE2_BIDIR.EI1 → AE_OUT.E1

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional AE event chain without breaking it
- Systems where a bidirectional peer (e.g. an operator panel) needs to both display a unidirectional event and send an event back
- Bridging a unidirectional AE zone and a bidirectional AE2 zone

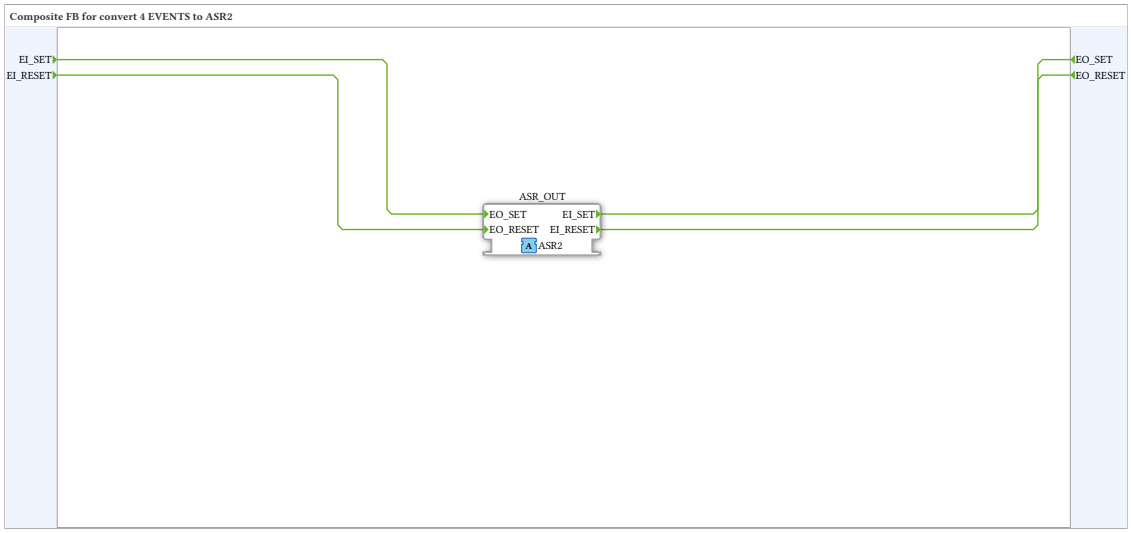
⚖️ Comparison with Similar Blocks

AE2_TO_AE is the mirror image of this block — there, `AE2_BIDIR` sits as a socket on the left side. For a pure, non-tapping 1:1 conversion between a plain EVENT signal and AE2, see AE2_EVENT_TO_E/ AE2_E_TO_EVENT. For two coupled events (SET/RESET), see ASR_TO_ASR2.

Conclusion

AE_TO_AE2 is not a simple converter but a tap point: it makes a unidirectional AE event visible at a bidirectional AE2 point without breaking the original unidirectional event chain, while forwarding the request event received there as a standalone AE event.

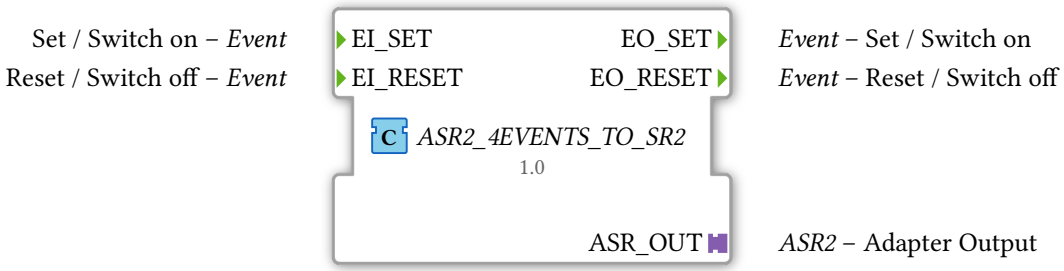
ASR2_4EVENTS_TO_SR2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block ASR2_4EVENTS_TO_SR2 is a composite function block used to convert four events into the ASR2 adapter format. It enables bidirectional conversion between individual set/reset events and the standardized ASR2 adapter.



Interface Structure

Event Inputs

- **EI_SET**: Set Event (Power On)
- **EI_RESET**: Reset Event (Power Off)

Event Outputs

- **EO_SET**: Set Event (Power On)
- **EO_RESET**: Reset Event (Power Off)

Adapters

- **ASR_OUT**: ASR2 Adapter Output (bidirectional)

Functionality

This function block acts as a bridge between individual events and the ASR2 adapter. It forwards incoming set and reset events to the ASR2 adapter and simultaneously passes corresponding events from the ASR2 adapter to the outputs. This enables seamless integration of individual events into systems that use the ASR2 adapter format.

Technical Features

- Bidirectional signal transmission
- Direct event forwarding without data processing
- Compatible with the ASR2 adapter standard
- Composite function block implementation

State Overview

The function block has no internal state, as it functions solely as a pass-through element. All events are forwarded directly and without delay.

Application Scenarios

- Integration of single-event systems into ASR2-based architectures
- Protocol conversion between different control systems
- Adapter solutions for existing automation systems
- Bidirectional communication between different event formats

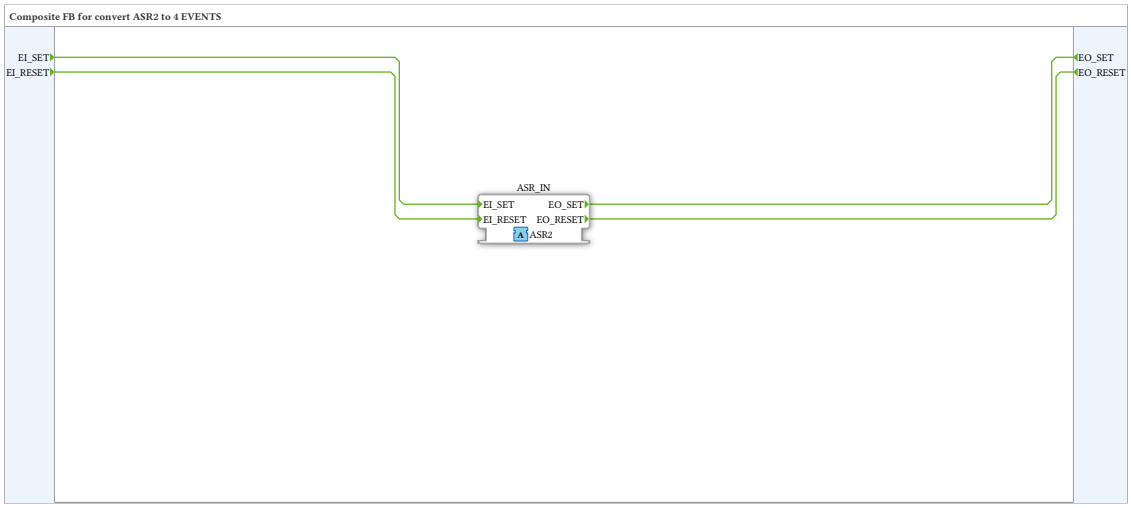
⚖️ Comparison with similar blocks

Compared to simple event blocks, ASR2_4EVENTS_TO_SR2 offers specific adapter functionality for the ASR2 format. While standard blocks typically only offer unidirectional signal processing, this block enables bidirectional communication with ASR2-compatible systems.

Conclusion

The ASR2_4EVENTS_TO_SR2 function block represents an efficient solution for integrating individual events into ASR2-based systems. Its bidirectional functionality and direct event forwarding make it a valuable component for system integration and protocol conversion in automated control systems.

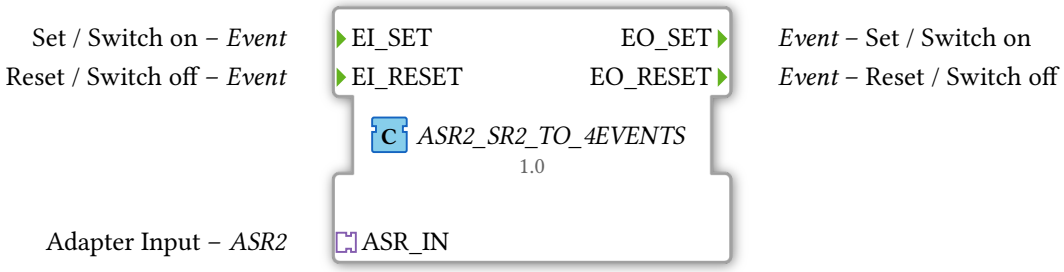
ASR2_SR2_TO_4EVENTS



• – ○ • – ○ • – ○ •

Introduction

The ASR2_SR2_TO_4EVENTS function block is a composite function block that provides the ASR2 adapter signals as four discrete event endpoints. It enables bidirectional conversion between ASR2 adapter interfaces and discrete event inputs and outputs.



Interface Structure

Event Inputs

- **EI_SET**: Set / Turn On
- **EI_RESET**: Reset / Turn Off

Event Outputs

- **EO_SET**: Set / Turn On
- **EO_RESET**: Reset / Turn Off

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

- **ASR_IN**: Adapter input of type `adapter::types::bidirectional::ASR2`

Functionality

This function block acts as a bridge between the ASR2 adapter and discrete events. The event inputs EI_SET and EI_RESET are passed directly to the ASR2 adapter, while the event outputs EO_SET and EO_RESET are fed directly from the ASR2 adapter. This enables bidirectional communication between the two interfaces.

Technical Features

- Implemented as a composite function block
- Uses bidirectional adapter communication
- Contains no internal data processing logic, but primarily serves as a signal relay
- Supports Eclipse 4diac standard compliance

State Overview

This function block has no internal states, as it functions exclusively as a pass-through block for events. All incoming events are immediately forwarded to the corresponding interface.

Application Scenarios

- Integration of ASR2-compatible components into 4diac systems
- Conversion between adapter-based and event-based interfaces
- Bidirectional signal transmission in automation systems
- Coupling of different communication protocols

⚖️ Comparison with similar modules

Compared to simple converter blocks, ASR2_SR2_TO_4EVENTS offers the unique feature of bidirectional communication and specific adaptation to ASR2 adapters. Other converter blocks are often unidirectional or support different adapter types.

Conclusion

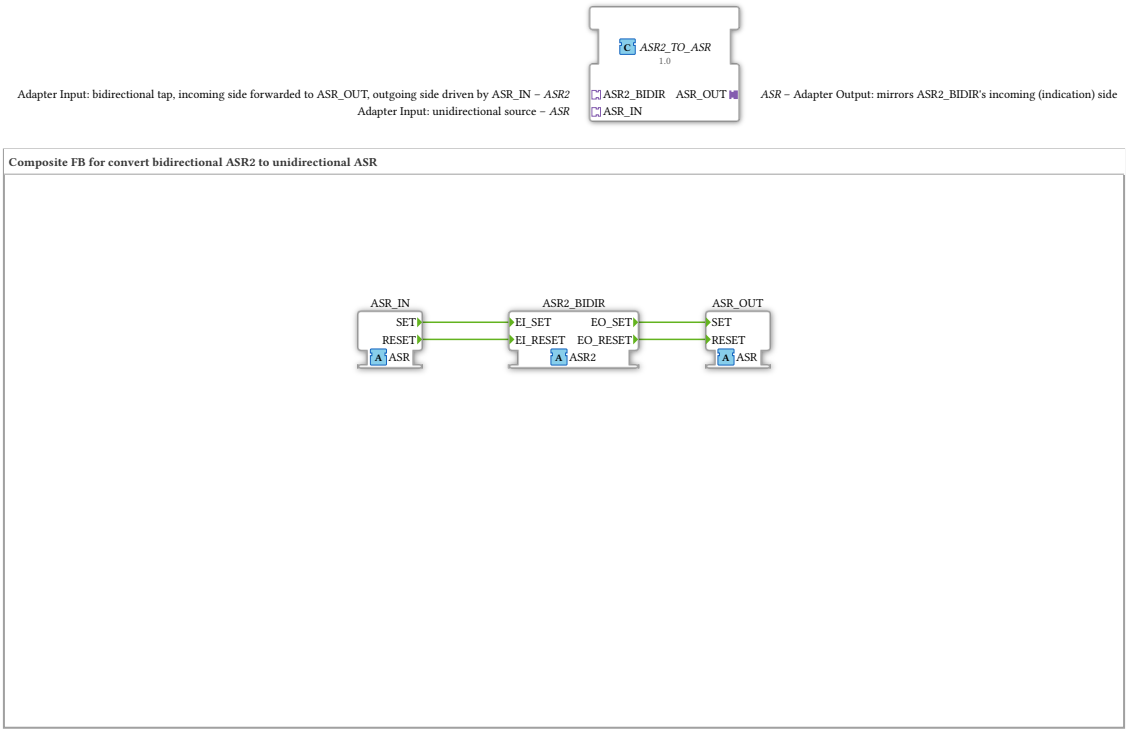
The ASR2_SR2_TO_4EVENTS function block represents an efficient solution for integrating ASR2 interfaces into 4diac-based control systems. Its simple yet effective pass-through architecture enables seamless communication between different interface types without additional processing latency.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ASR2_TO_ASR



• — ○ • — ○ • — ○ •

Introduction

The ASR2_TO_ASR function block is the mirror image of ASR_TO_ASR2: here the bidirectional ASR2 tap sits as a socket on the left (input) side, complemented by a second unidirectional ASR socket and a unidirectional ASR plug.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no data inputs or outputs at all — ASR and ASR2 carry only the SET/RESET events, never payload data.

Data Outputs

The function block has no data inputs or outputs at all — ASR and ASR2 carry only the SET/RESET events, never payload data.

Adapters

- **ASR2_BIDIR** (Socket): bidirectional tap whose incoming side is forwarded to `ASR_OUT` and whose outgoing side is fed by `ASR_IN`, of type `adapter::types::bidirectional::ASR2`
- **ASR_IN** (Socket): unidirectional SET/RESET source, of type `adapter::types::unidirectional::ASR`
- **ASR_OUT** (Plug): mirrors the incoming (indication) side of `ASR2_BIDIR` as unidirectional SET/RESET events, of type `adapter::types::unidirectional::ASR`

Functionality

The block operates in two directions at once: whatever the socket `ASR2_BIDIR` receives on its indication side (`EO_SET`/`EO_RESET` — i.e. whatever a bidirectional peer attached to ASR2_BIDIR reports) is output unchanged as standalone unidirectional events via the plug `ASR_OUT` (`SET`/`RESET`). Conversely, whatever arrives at the socket `ASR_IN` (`SET`/`RESET`) is published onto the request side of `ASR2_BIDIR` (`EI_SET`/`EI_RESET`) — so ASR2_BIDIR forwards the same events to the attached bidirectional peer. Since ASR/ASR2 are pure event adapters with no payload data, there are no DataConnections at all.

Technical Details

- Three adapters instead of two: the direction received from the tap and the separately injected direction are deliberately kept apart rather than mixed
- Two parallel but independently routed event pairs (SET and RESET), never merged or combined
- Pure wiring, no logic or state, every destination variable has exactly one writer
- No data transport: ASR/ASR2 carry only SET/RESET events, so this block has no DataConnections
- `ASR_OUT` only reflects what is reported via `ASR2_BIDIR` by an attached bidirectional peer — without such a peer, `ASR_OUT` stays idle

State Overview

The block is stateless:

- ASR2_BIDIR.EO_SET ⟶ ASR_OUT.SET, ASR2_BIDIR.EO_RESET ⟶ ASR_OUT.RESET
- ASR_IN.SET ⟶ ASR2_BIDIR.EI_SET, ASR_IN.RESET ⟶ ASR2_BIDIR.EI_RESET

Application Scenarios

- Reading a bidirectional ASR2 peer as a plain unidirectional SET/RESET pair, combined with injecting a separate unidirectional SET/RESET pair
- Systems where a bidirectional peer needs to be monitored without converting the rest of the control logic to ASR2
- Bridging a bidirectional ASR2 zone and a unidirectional ASR zone

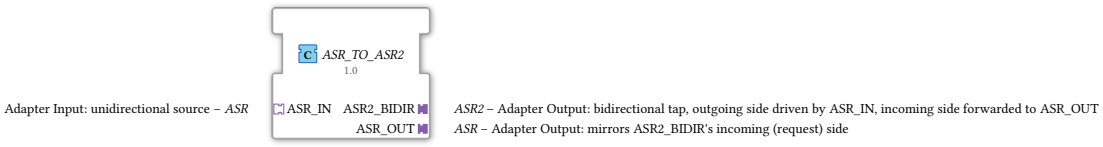
⚖️ Comparison with Similar Blocks

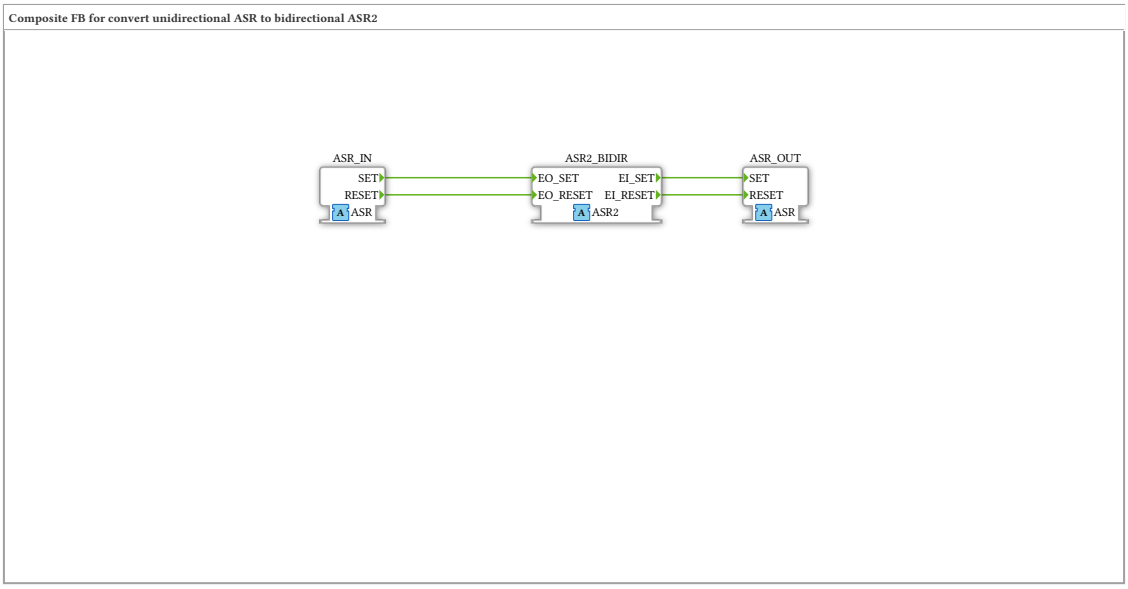
ASR_TO_ASR2 is the mirror image of this block — there, `ASR2_BIDIR` sits as a plug on the right side. For a pure, non-tapping 1:1 conversion between ASR2 and two plain EVENT signals, see ASR2_SR2_TO_4EVENTS/ASR2_4EVENTS_TO_SR2. For a single event without SET/RESET coupling, see AE2_TO_AE.

Conclusion

ASR2_TO_ASR is not a simple converter but a tap point: it makes the state of a bidirectional ASR2 peer visible as a standalone unidirectional SET/RESET pair while simultaneously injecting a separate unidirectional SET/RESET pair into the bidirectional side.

ASR_TO_ASR2





• – ○ • – ○ • – ○ •

Introduction

The ASR_TO_ASR2 function block inserts a bidirectional ASR2 tap into a unidirectional ASR chain. ASR/ASR2 carry two coupled events, SET and RESET, rather than a single E1. Like the other bridge blocks, ASR_TO_ASR2 uses **three** adapters: incoming unidirectional SET/RESET events are both published onto the bidirectional ASR2 interface and — whatever ASR2 returns on its request side — output again as standalone unidirectional SET/RESET events.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no data inputs or outputs at all — ASR and ASR2 carry only the SET/RESET events, never payload data.

Data Outputs

The function block has no data inputs or outputs at all — ASR and ASR2 carry only the SET/RESET events, never payload data.

Adapters

- **ASR_IN** (Socket): unidirectional SET/RESET source, of type `adapter::types::unidirectional::ASR`
- **ASR2_BIDIR** (Plug): bidirectional tap whose outgoing side is fed by `ASR_IN` and whose incoming side is forwarded to `ASR_OUT`, of type `adapter::types::bidirectional::ASR2`
- **ASR_OUT** (Plug): mirrors the incoming (request) side of `ASR2_BIDIR` as unidirectional SET/RESET events, of type `adapter::types::unidirectional::ASR`

Functionality

The block operates in two directions at once: whatever arrives at the socket `ASR_IN` (`SET`/`RESET`) is published directly onto the outgoing side of `ASR2_BIDIR` (`EO_SET`/`EO_RESET`) — so ASR2_BIDIR reports outward the same events coming in via ASR_IN. Conversely, whatever `ASR2_BIDIR` receives on its request side (`EI_SET`/`EI_RESET` — i.e. whatever a bidirectional peer attached to ASR2_BIDIR "sends back") is output unchanged as standalone unidirectional events via the plug `ASR_OUT` (`SET`/`RESET`). Since ASR/ASR2 are pure event adapters with no payload data, there are no DataConnections at all.

Technical Details

- Three adapters instead of two: the incoming event direction and the direction coming back from the tap are deliberately kept separate rather than mixed
- Two parallel but independently routed event pairs (SET and RESET), never merged or combined
- Pure wiring, no logic or state, every destination variable has exactly one writer
- No data transport: ASR/ASR2 carry only SET/RESET events, so this block has no DataConnections
- `ASR_OUT` only reflects what comes back via `ASR2_BIDIR` from an attached bidirectional peer — without such a peer, `ASR_OUT` stays idle

State Overview

The block is stateless:

- ASR_IN.SET ⟶ ASR2_BIDIR.EO_SET, ASR_IN.RESET ⟶ ASR2_BIDIR.EO_RESET
- ASR2_BIDIR.EI_SET ⟶ ASR_OUT.SET, ASR2_BIDIR.EI_RESET ⟶ ASR_OUT.RESET

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional ASR SET/RESET chain without breaking it
- Systems where a bidirectional peer (e.g. an operator panel) needs to both display SET/RESET state and send SET/RESET events back
- Bridging a unidirectional ASR zone and a bidirectional ASR2 zone

⚖️ Comparison with Similar Blocks

ASR2_TO_ASR is the mirror image of this block — there, `ASR2_BIDIR` sits as a socket on the left side. For a pure, non-tapping 1:1 conversion between two plain EVENT signals and ASR2, see ASR2_4EVENTS_TO_SR2/ASR2_SR2_TO_4EVENTS. For a single event without SET/RESET coupling, see AE_TO_AE2.

Conclusion

ASR_TO_ASR2 is not a simple converter but a tap point: it makes a unidirectional ASR SET/RESET pair visible at a bidirectional ASR2 point without breaking the original unidirectional chain, while forwarding the request SET/RESET pair received there as a standalone ASR signal.

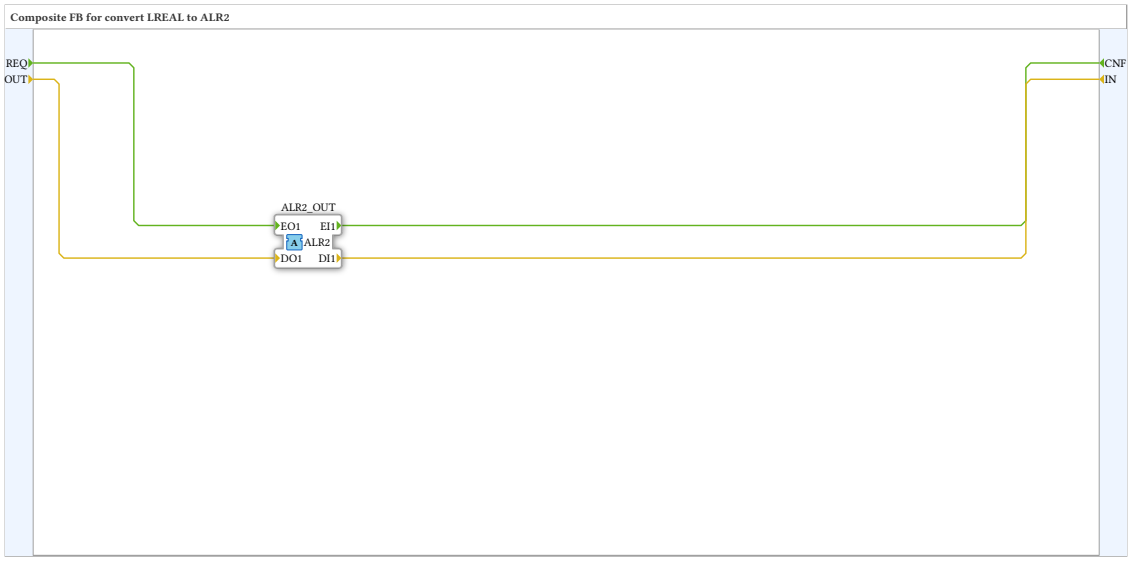
EVENT

🎧 Podcast

- E_TOF vs. FB_TOF: The event timer that doesn't tick cyclically – a revolution for automation systems?
- Event FBs according to IEC 61499: Lego bricks of automation – how event function blocks work
- IEC 61499: Revolution of automation – event-driven FBs and distributed systems explained
- E_REND: Event Synchronization in IEC 61499
- The Unstoppable Counter: Why IEC 61499's ECTU Guarantees Safe, Event-Driven Control (and Never Overflows)

LREAL

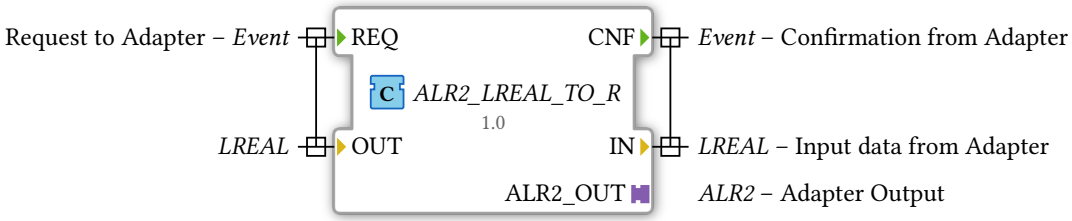
ALR2_LREAL_TO_R



• – ◦ • – ◦ • – ◦ •

Introduction

`ALR2_LREAL_TO_R` is a composite FB that mediates between an `LREAL` data pair (`OUT`/`IN`) and the bidirectional adapter `ALR2`. It completes the conversion family analogously to `AR2`/`AB2`/`AE2`.



Interface Structure

Event Inputs

- `REQ`: Request to the adapter. Associated data: `OUT`.

Event Outputs

- `CNF`: Confirmation from the adapter. Associated data: `IN`.

Data Inputs

- `OUT` (`LREAL`): Value sent to the adapter.

Data Outputs

- `IN` (`LREAL`): Value received from the adapter.

Adapters

Direction	Name	Adapter type	Comment
Plug (output)	`ALR2_OUT`	`adapter::types::bidirectional::ALR2`	Adapter output.

Functionality

The block contains no internal FBs, only connections:

- `REQ` ⟶ `ALR2_OUT.EO1` and `OUT` ⟶ `ALR2_OUT.DO1`: a `REQ` event sends the value `OUT` through the adapter.

- `ALR2_OUT.EI1` ⟶ `CNF` and `ALR2_OUT.DI1` ⟶ `IN`: an event received through the adapter is output as `CNF`, and the received value appears at `IN`.

Technical Details

- Pure pass-through without logic.
- The data type is `LREAL` (64-bit floating point).
- The naming `OUT`/`IN` is chosen from the adapter's point of view: `OUT` goes out to the adapter, `IN` comes in from the adapter.

Application Scenarios

- Output an `LREAL` value to a bidirectional `ALR2` adapter and at the same time read its backward channel as a data output.

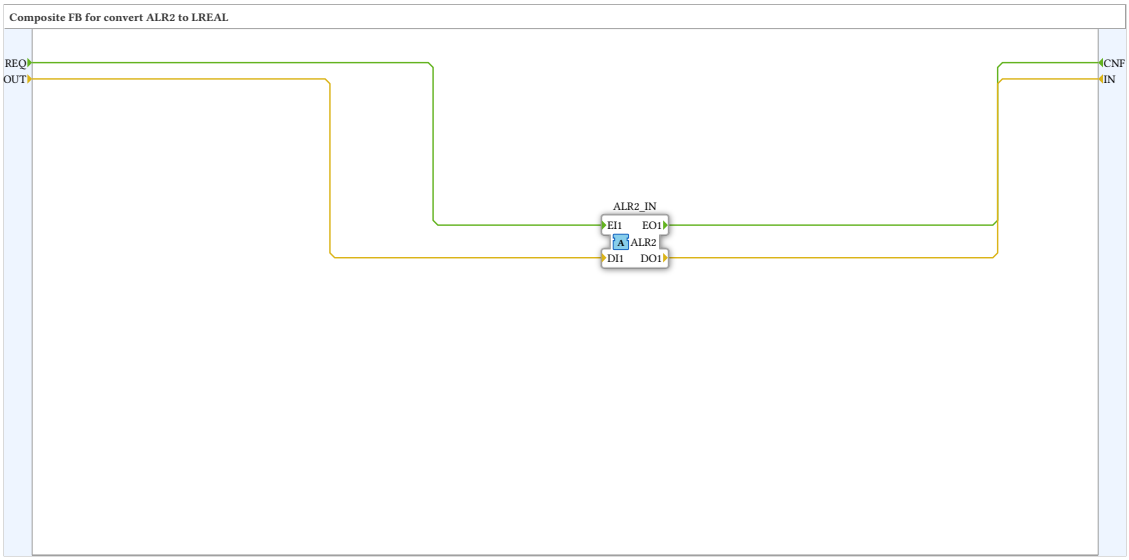
Summary

`ALR2_LREAL_TO_R` connects events and `LREAL` data 1:1 with the bidirectional adapter `ALR2`.

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

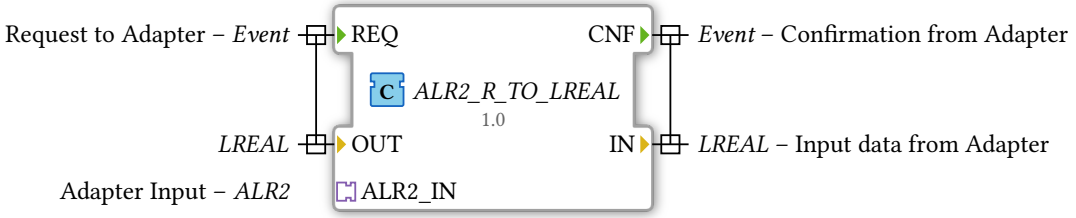
ALR2_R_TO_LREAL



• – ◦ • – ◦ • – ◦ •

Introduction

`ALR2_R_TO_LREAL` is a composite FB that mediates between a `LREAL` data pair (`OUT`/`IN`) and the bidirectional adapter `ALR2`. In contrast to the plug variant `ALR2_LREAL_TO_R` it has an adapter **socket** (input). It completes the conversion family analogously to `AR2`/`AB2`/`AE2`.



Interface Structure

Event Inputs

- `REQ`: Request to the adapter. Associated data: `OUT`.

Event Outputs

- `CNF`: Confirmation from the adapter. Associated data: `IN`.

Data Inputs

- `OUT` (`LREAL`): Value sent to the adapter.

Data Outputs

- `IN` (`LREAL`): Value received from the adapter.

Adapters

Direction	Name	Adapter type	Comment
Socket (input)	`ALR2_IN`	`adapter::types::bidirectional::ALR2`	Adapter input.

Functionality

The block contains no internal FBs, only connections:

- `REQ` ⟶ `ALR2_IN.EI1` and `OUT` ⟶ `ALR2_IN.DI1`: a `REQ` event sends the value `OUT` through the adapter.
- `ALR2_IN.EO1` ⟶ `CNF` and `ALR2_IN.DO1` ⟶ `IN`: an event coming from the adapter is output as `CNF`, and the received value appears at `IN`.

Technical Details

- Pure pass-through without logic and without internal state.
- The data type is `LREAL` (64-bit floating point).
- The naming `OUT`/`IN` is chosen from the adapter's point of view: `OUT` goes out to the adapter, `IN` comes in from the adapter.

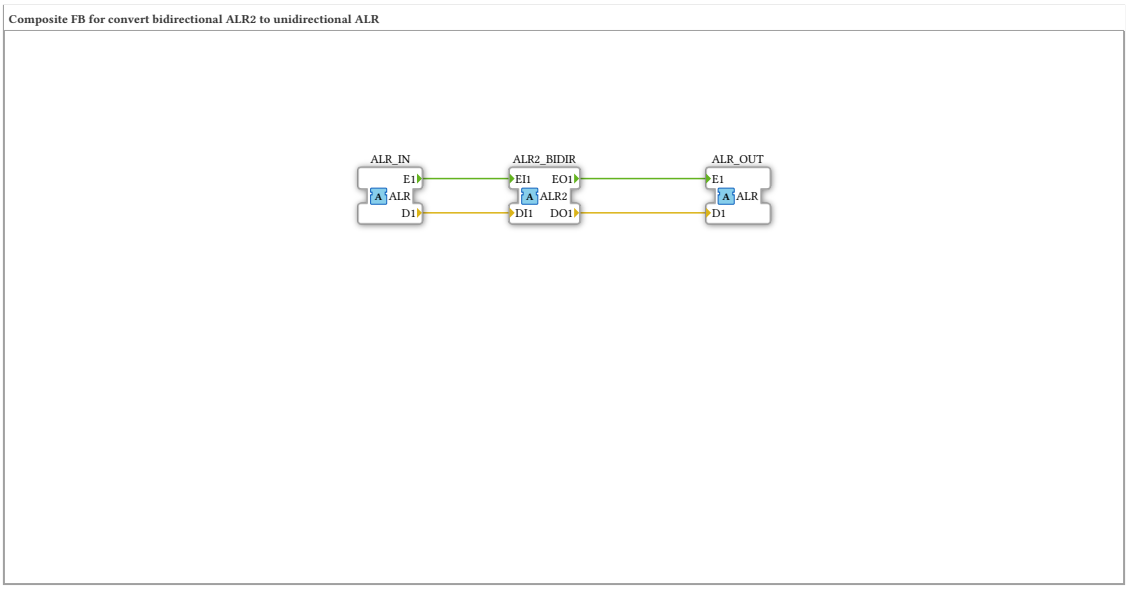
Application Scenarios

- Output a `LREAL` value to a bidirectional `ALR2` adapter and at the same time read its backward channel as a data output, when the adapter is present as a socket.

Summary

`ALR2_R_TO_LREAL` connects events and `LREAL` data 1:1 with the bidirectional adapter `ALR2` (socket variant).

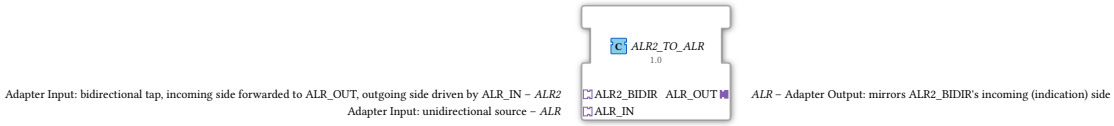
ALR2_TO_ALR



• – ○ • – ○ • – ○ •

Introduction

ALR2_TO_ALR is a composite function block that connects a bidirectional ALR2 tap with unidirectional ALR adapters (LREAL): the incoming side of ALR2_BIDIR is output as a plain unidirectional signal via ALR_OUT, and ALR_IN is published onto the outgoing side of ALR2_BIDIR.



Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **ALR2_BIDIR** (Socket): bidirectional tap whose incoming side is forwarded to ALR_OUT and whose outgoing side is fed by ALR_IN, of type adapter::types::bidirectional::ALR2
- **ALR_IN** (Socket): unidirectional source, of type adapter::types::unidirectional::ALR
- **ALR_OUT** (Plug): mirrors the incoming (indication) side of ALR2_BIDIR, of type adapter::types::unidirectional::ALR

Functionality

The block consists of just four connections:

- Event: ALR2_BIDIR.EO1 ⟶ ALR_OUT.E1; data: ALR2_BIDIR.DO1 ⟶ ALR_OUT.D1
- Event: ALR_IN.E1 ⟶ ALR2_BIDIR.EI1; data: ALR_IN.D1 ⟶ ALR2_BIDIR.DI1

Whatever a bidirectional peer attached to ALR2_BIDIR reports therefore appears unchanged at ALR_OUT; whatever arrives at ALR_IN is passed on to the bidirectional peer.

Technical Details

- Pure wiring, no logic or state
- Every destination variable has exactly one writer
- Without an attached bidirectional peer, ALR_OUT stays idle

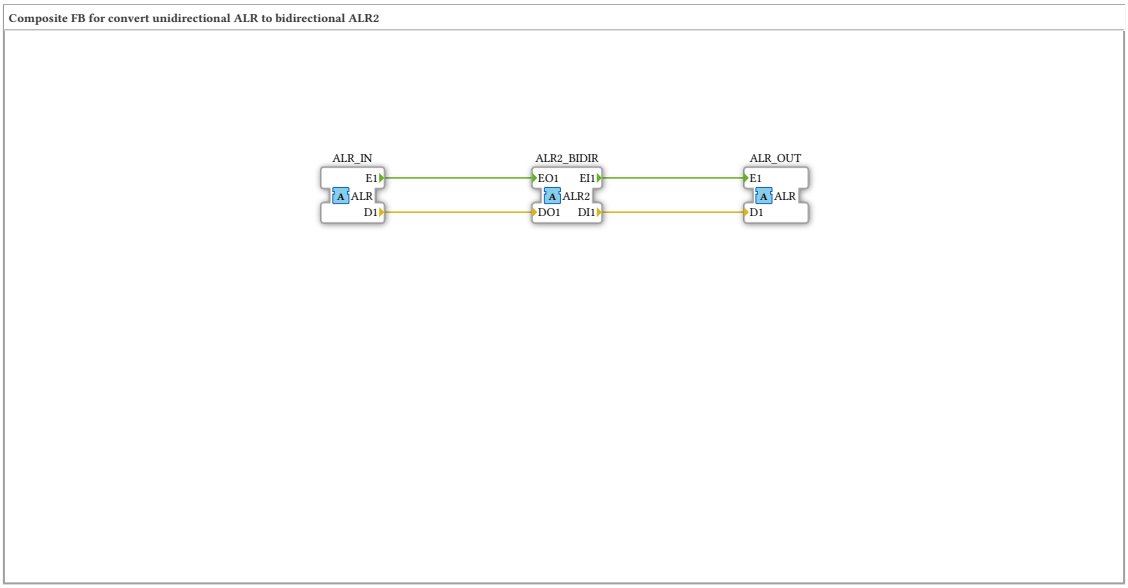
Application Scenarios

- Reading a bidirectional ALR2 peer as a plain unidirectional ALR signal while injecting a separate unidirectional signal
- Bridging a bidirectional ALR2 zone and a unidirectional ALR zone

Conclusion

ALR2_TO_ALR is a tap point without logic: it makes the incoming side of a bidirectional ALR2 peer visible as a unidirectional ALR signal and simultaneously feeds a unidirectional signal into the outgoing side.

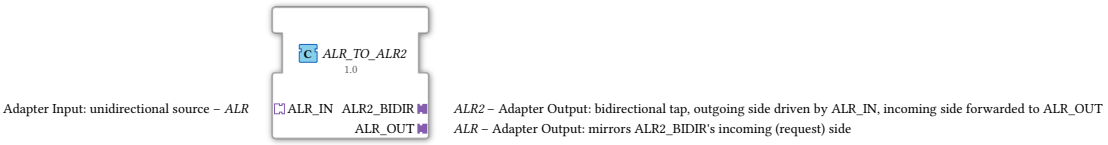
ALR_TO_ALR2



• – ○ • – ○ • – ○ •

Introduction

`ALR_TO_ALR2` is a composite function block that converts a **unidirectional** `ALR` adapter into a **bidirectional** `ALR2` adapter. `ALR_IN` is written onto the outgoing side (`EO1`/`DO1`) of `ALR2_BIDIR`; whatever `ALR2_BIDIR` receives on its incoming side (`EI1`/`DI1`) is output again as a plain unidirectional signal via `ALR_OUT`.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **ALR_IN** (socket): Unidirectional adapter of type `adapter::types::unidirectional::ALR` - unidirectional source
- **ALR2_BIDIR** (plug): Bidirectional adapter of type `adapter::types::bidirectional::ALR2` - outgoing side driven by `ALR_IN`, incoming side forwarded to `ALR_OUT`
- **ALR_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::ALR` - mirrors the incoming (request) side of `ALR2_BIDIR`

How it works

1. `ALR_IN.E1` is forwarded to `ALR2_BIDIR.EO1` and `ALR_IN.D1` to `ALR2_BIDIR.DO1`.
2. `ALR2_BIDIR.EI1` is forwarded to `ALR_OUT.E1` and `ALR2_BIDIR.DI1` to `ALR_OUT.D1`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- Part of the conversion family (analogous to `AR2`/`AB2`/`AE2`); the data type is `LREAL`

Application Scenarios

- Connecting unidirectional `ALR` sources and sinks to a bidirectional `ALR2` adapter

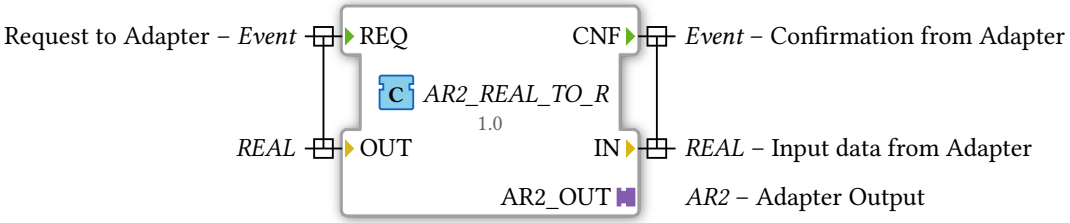
Summary

`ALR_TO_ALR2` combines a unidirectional `ALR` input and output into a bidirectional `ALR2` adapter.

Numeric Index (bidirectional)

Here you will find all exercises from the `bidirectional` workspace, listed in ascending order. …

AR2_REAL_TO_R





• – ○ • – ○ • – ○ •

Introduction

The function block `AR2_REAL_TO_R` is a composite function block (FB) for converting a REAL value into an AR2 format. It encapsulates a bidirectional adapter of type `adapter::types::bidirectional::AR2` and acts as a transparent interface between a simple REAL variable and the more complex AR2 data type. The conversion logic resides entirely within the adapter, while the FB only handles signal and data transmission.

Interface Structure

Event Inputs

Event	Comment
`REQ`	Request for processing via the adapter (triggers adapter output)

Event Outputs

Event	Comment
CNF	Confirmation after adapter processing is complete

Data Inputs

Variable	Type	Comment
OUT	REAL	Output data sent to the adapter (actually: input value for conversion)

Data Outputs

Variable	Type	Comment
IN	REAL	Input data returned by the adapter (actually: converted value)

Adapter

Adapter	Type	Comment
AR2_OUT	adapter::types::bidirectional::AR2	Bidirectional interface adapter for AR2 conversion

Functionality

The function block operates as a simple forwarding block:

1. An incoming event at the **event input REQ** is forwarded directly to the event output EO1 of the adapter AR2_OUT.
2. The current value of the **data input OUT** is simultaneously sent to the data output DO1 of the adapter.
3. The adapter internally performs the conversion from REAL to AR2 format (and back if necessary) and sends an acknowledgment event back via its event input EI1.
4. This event is passed on to the **event output CNF** of the function block.
5. The value provided by the adapter at its data input DI1 is transferred to the **data output IN** of the function block.

The actual conversion (REAL ↔ AR2) takes place exclusively within the adapter – the function block itself does not perform any arithmetic operations.

Technical Features

- **Composite Block**: The functionality is implemented through the internal wiring of an external adapter type. The function block is therefore dependent on the implementation of the adapter adapter::types::bidirectional::AR2.
- **Bidirectional Data Flow**: The variable names (OUT as input, IN as output) can be confusing. In reality, the data flows as follows:
- REAL value ⟶ Adapter ⟶ AR2 object (forward)
- AR2 object ⟶ Adapter ⟶ REAL value (reverse)

The function block provides both directions via the event handlers REQ / CNF.

- **No State Logic**: The function block (FB) does not have its own state machine – all processes are event-driven and direct.

State Overview

Since the FB does not have an internal state machine, there is no classic state overview. The process is purely sequential per event:

1. Wait for REQ
2. Send to adapter
3. Wait for adapter response
4. Output of CNF and updated IN

Application Scenarios

- **Interface to External Systems**: When an external device or another function block expects data in AR2 format, but the user code works with REAL values.
- **Data Preprocessing**: Integration into a chain of conversion blocks to convert REAL data into a specific binary or structured format (AR2).
- **Testing and Simulation**: Used in test environments to verify adapter functionality without requiring in-depth knowledge of the adapter itself.

Comparison with Similar Function Blocks

- **AR2_REAL_TO_R vs. Simple Type Conversion (REAL_TO_INT etc.)**:

Simple converters work directly with elementary data types. `AR2_REAL_TO_R`, on the other hand, uses an adapter that encapsulates a complex conversion (e.g., to an array or structure).

- **AR2_REAL_TO_R vs. Direct Adapter Use**:

The function block simplifies handling by performing the signal wiring (event and data connections). The user only needs to instantiate the function block and connect it to their code.

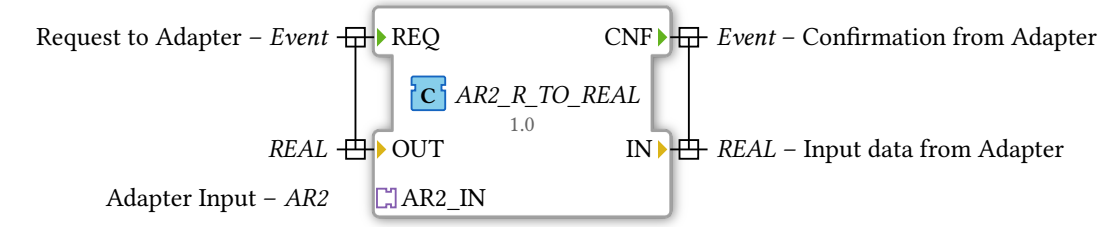
- **AR2_REAL_TO_R vs. CONVERT Adapter Blocks**:

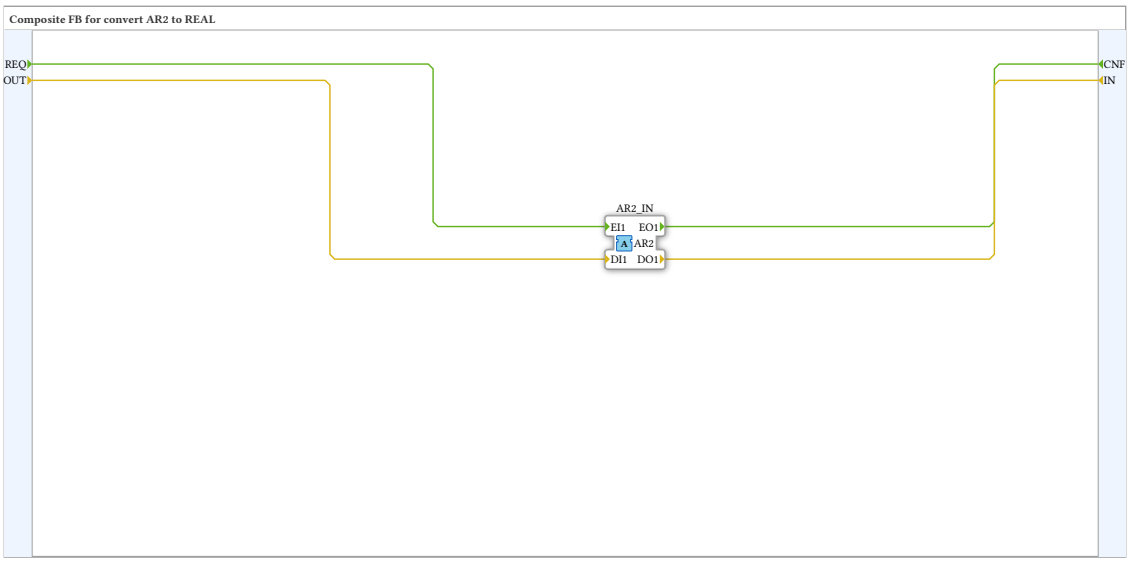
Similar blocks (e.g., from the IEC 61499 library) usually offer generic conversion. This function block is specifically for coupling REAL and AR2.

Conclusion

The function block `AR2_REAL_TO_R` provides a simple, event-driven interface for using a bidirectional AR2 adapter. It is ideal for all applications that require conversion between the standard REAL data type and a project-specific AR2 format. Its design keeps the complexity hidden within the adapter, and the block can be integrated into applications like a normal function block.

AR2_R_TO_REAL





• – ◦ • – ◦ • – ◦ •

Introduction

The **AR2_R_TO_REAL** function block is a composite module that acts as a bridge between an AR2 adapter interface and REAL values. It enables bidirectional data transfer and conversion, with the actual conversion being performed by an external adapter of type `adapter::types::bidirectional::AR2`.

Interface Structure

Event Inputs

Event	Description
`REQ`	Starts data transfer to the adapter. The data value `OUT` is sent.

Event Outputs

Event	Description
`CNF`	Acknowledges the adapter's response. The converted value is stored as `IN`.

Data Inputs

Variable	Type	Description
`OUT`	REAL	Data value passed to the adapter.

Data Outputs

Variable	Type	Description
`IN`	REAL	Converted value returned by the adapter.

Adapter

Name	Type	Direction	Description
`AR2_IN`	`adapter::types::bidirectional::AR2`	Socket	Bidirectional adapter for AR2 communication and conversion.

Functionality

The module functions purely as a mediation layer:

1. An incoming event `REQ` triggers the transfer of the value `OUT` (REAL) to the connected adapter (via `AR2_IN.DI1`).
2. The adapter performs the necessary conversion (e.g., AR2 ↔ REAL) and sends an acknowledgment event to its output `EO1`.
3. This event is forwarded externally as `CNF`, and the converted value is made available via `AR2_IN.DO1` as `IN` (REAL).

The actual conversion direction is determined by the implementation of the connected adapter. The name of the function block suggests a primary conversion from AR2 to REAL, but the bidirectional adapter interface also allows for the reverse direction.

Technical Features

- **Composite Function Block**: Contains no internal logic; it simply combines an adapter socket with event and data connections.
- **Bidirectional Adapter**: The interface `AR2_IN` must provide an adapter of type `adapter::types::bidirectional::AR2`, which implements the actual conversion.
- **Simple Coupling**: The function block separates the REAL world from the AR2 world, thus enabling modular integration.

State Overview

The function block has no internal states of its own. Its behavior is entirely controlled by the external events `REQ` and the adapter's feedback.

Application Scenarios

- **Connecting AR2-based sensors/actuators** to a control system that uses REAL values.
- **Bidirectional data bridge** in mixed automation environments where AR2 and REAL protocols interact.
- **Test and simulation setups** where AR2 data needs to be replaced with easily manageable REAL values.

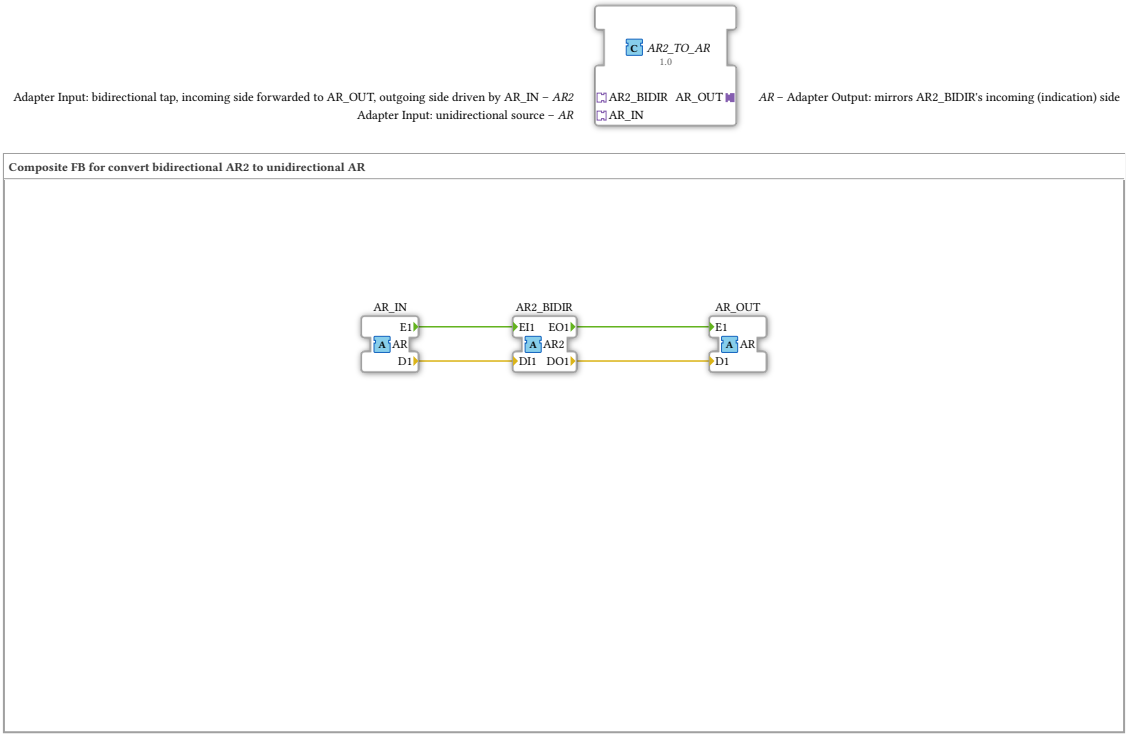
Comparison with Similar Components

While direct conversion components (e.g., AR2_TO_REAL) hardwire the conversion, this composite component allows for the flexible connection of an external adapter. This decouples the conversion logic, allowing it to be replaced depending on the application without changing the rest of the configuration.

Conclusion

The **AR2_R_TO_REAL** is a useful bridging component for systems that need to process both AR2 and REAL data. By using a bidirectional adapter, a clear separation between the interface and the conversion logic is achieved, which improves reusability and maintainability.

AR2_TO_AR



• — ○ • — ○ • — ○ •

Introduction

The AR2_TO_AR function block is the mirror image of AR_TO_AR2: here the bidirectional AR2 tap sits as a socket on the left (input) side, complemented by a second unidirectional AR socket and a unidirectional AR plug.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AR2_BIDIR** (Socket): bidirectional tap whose incoming side is forwarded to `AR_OUT` and whose outgoing side is fed by `AR_IN`, of type `adapter::types::bidirectional::AR2`
- **AR_IN** (Socket): unidirectional REAL source, of type `adapter::types::unidirectional::AR`
- **AR_OUT** (Plug): mirrors the incoming (indication) side of `AR2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::AR`

Functionality

The block operates in two directions at once: whatever the socket `AR2_BIDIR` receives on its indication side (`EO1`/`DO1` — i.e. whatever a bidirectional peer attached to AR2_BIDIR reports) is output unchanged as a standalone unidirectional signal via the plug `AR_OUT`. Conversely, whatever arrives at the socket `AR_IN` (`E1`/`D1`) is published onto the request side of `AR2_BIDIR` (`EI1`/`DI1`) — so AR2_BIDIR forwards the same signal to the attached bidirectional peer.

Technical Details

- Three adapters instead of two: the direction received from the tap and the separately injected direction are deliberately kept apart rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `AR_OUT` only reflects what is reported via `AR2_BIDIR` by an attached bidirectional peer — without such a peer, `AR_OUT` stays idle

State Overview

The block is stateless:

- AR2_BIDIR.EO1 ⟶ AR_OUT.E1, AR2_BIDIR.DO1 ⟶ AR_OUT.D1
- AR_IN.E1 ⟶ AR2_BIDIR.EI1, AR_IN.D1 ⟶ AR2_BIDIR.DI1

Application Scenarios

- Reading the status of a bidirectional AR2 peer as a plain unidirectional AR signal, combined with injecting a separate unidirectional setpoint signal
- Systems where a bidirectional peer needs to be monitored without converting the rest of the control logic to AR2
- Bridging a bidirectional AR2 zone and a unidirectional AR zone

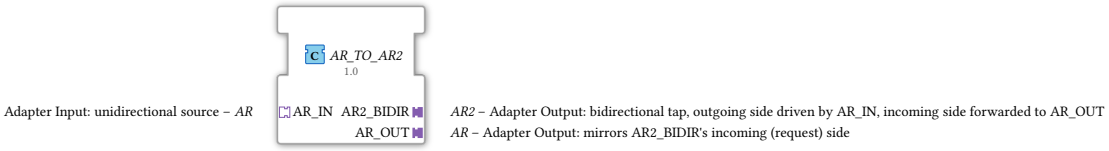
⚖️ Comparison with Similar Blocks

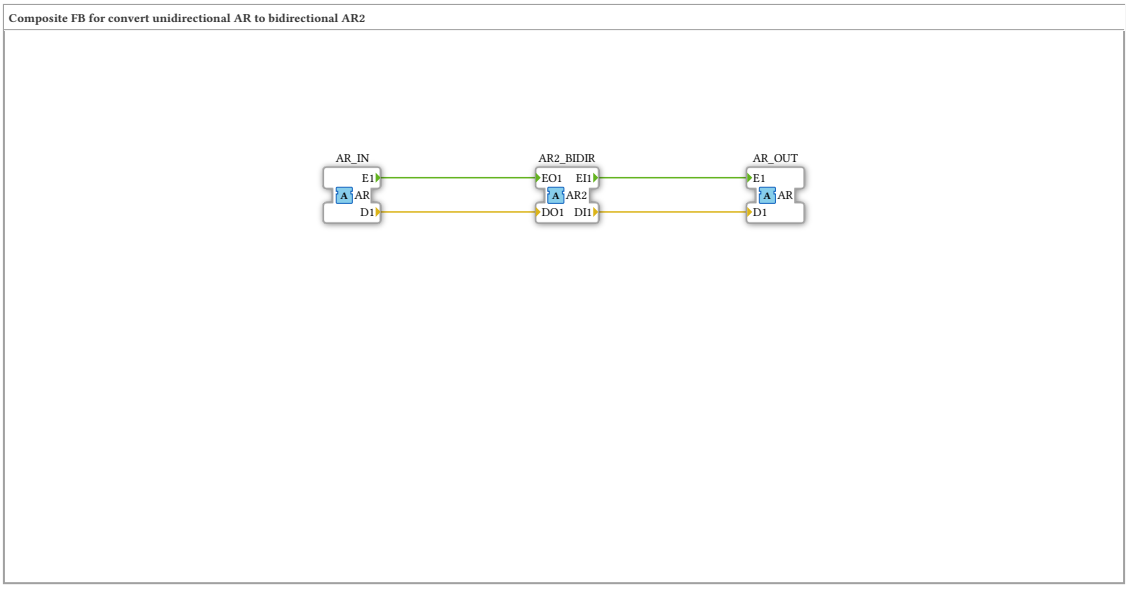
AR_TO_AR2 is the mirror image of this block — there, `AR2_BIDIR` sits as a plug on the right side. For a pure, non-tapping 1:1 conversion between AR2 and a plain REAL signal, see AR2_REAL_TO_R/ AR2_R_TO_REAL.

Conclusion

AR2_TO_AR is not a simple converter but a tap point: it makes the state of a bidirectional AR2 peer visible as a standalone unidirectional AR signal while simultaneously injecting a separate unidirectional signal into the bidirectional side.

AR_TO_AR2





• – ○ • – ○ • – ○ •

Introduction

The AR_TO_AR2 function block inserts a bidirectional AR2 tap into a unidirectional AR chain. Rather than a simple 1:1 conversion, it uses **three** adapters: an incoming unidirectional AR signal is both published onto the bidirectional AR2 interface and — whatever AR2 returns on its request side — output again as a standalone unidirectional AR signal.

Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AR_IN** (Socket): unidirectional REAL source, of type `adapter::types::unidirectional::AR`
- **AR2_BIDIR** (Plug): bidirectional tap whose outgoing side is fed by `AR_IN` and whose incoming side is forwarded to `AR_OUT`, of type `adapter::types::bidirectional::AR2`
- **AR_OUT** (Plug): mirrors the incoming (request) side of `AR2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::AR`

Functionality

The block operates in two directions at once: whatever arrives at the socket `AR_IN` (`E1`/`D1`) is published directly onto the outgoing side of `AR2_BIDIR` (`EO1`/`DO1`) — so AR2_BIDIR reports outward the same signal coming in via AR_IN. Conversely, whatever `AR2_BIDIR` receives on its request side (`EI1`/`DI1` — i.e. whatever a bidirectional peer attached to AR2_BIDIR "sends back") is output unchanged as a standalone unidirectional signal via the plug `AR_OUT`. AR2_BIDIR thus acts as a tap point that doesn't break the unidirectional chain but extends it with a bidirectional observation/injection point.

Technical Details

- Three adapters instead of two: the incoming signal direction and the direction coming back from the tap are deliberately kept separate rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `AR_OUT` only reflects what comes back via `AR2_BIDIR` from an attached bidirectional peer — without such a peer, `AR_OUT` stays idle

State Overview

The block is stateless:

- AR_IN.E1 ⟶ AR2_BIDIR.EO1, AR_IN.D1 ⟶ AR2_BIDIR.DO1
- AR2_BIDIR.EI1 ⟶ AR_OUT.E1, AR2_BIDIR.DI1 ⟶ AR_OUT.D1

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional AR signal chain without breaking it
- Systems where a bidirectional peer (e.g. an operator panel) needs to both display the status of a unidirectional measurement source and send setpoints back
- Bridging a unidirectional AR zone and a bidirectional AR2 zone

⚖️ Comparison with Similar Blocks

AR2_TO_AR is the mirror image of this block — there, `AR2_BIDIR` sits as a socket on the left side. For a pure, non-tapping 1:1 conversion between a plain REAL signal and AR2, see AR2_R_TO_REAL/ AR2_REAL_TO_R.

Conclusion

AR_TO_AR2 is not a simple converter but a tap point: it makes a unidirectional AR signal visible at a bidirectional AR2 point without breaking the original unidirectional signal chain, while forwarding the request signal received there as a standalone AR signal.

SINT

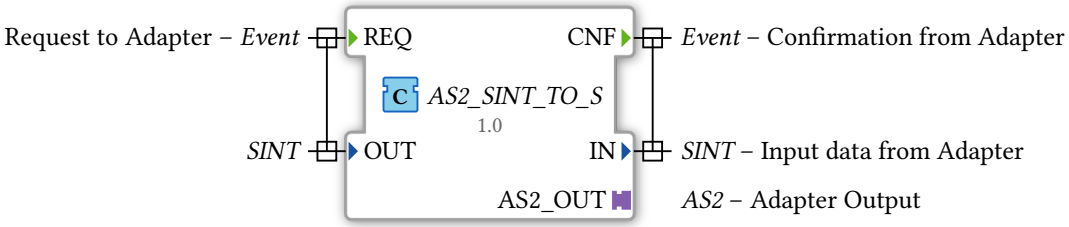
AS2_SINT_TO_S



• – ○ • – ○ • – ○ •

Introduction

`AS2_SINT_TO_S` is a composite function block that mediates between classic event/data connections of type `SINT` and a **bidirectional** `AS2` adapter. On `REQ`, the input `OUT` is written onto the outgoing side of the adapter plug `AS2_OUT`; the value reported via the adapter's backward direction appears at output `IN` and is acknowledged with `CNF`.



Interface Structure

Event Inputs

- **REQ**: Request to the adapter. Associated data: `OUT`

Event Outputs

- **CNF**: Confirmation from the adapter. Associated data: `IN`

Data Inputs

- **OUT** (`SINT`): Data sent to the adapter

Data Outputs

- **IN** (`SINT`): Data received from the adapter

Adapters

- **AS2_OUT** (plug): Bidirectional adapter of type `adapter::types::bidirectional::AS2` (adapter output)

How it works

1. `REQ` is forwarded to `AS2_OUT.EO1`, `OUT` to `AS2_OUT.DO1`.
2. `AS2_OUT.EI1` is forwarded to `CNF`, `AS2_OUT.DI1` to `IN`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- Part of the conversion family (analogous to `AR2`/`AB2`/`AE2`)

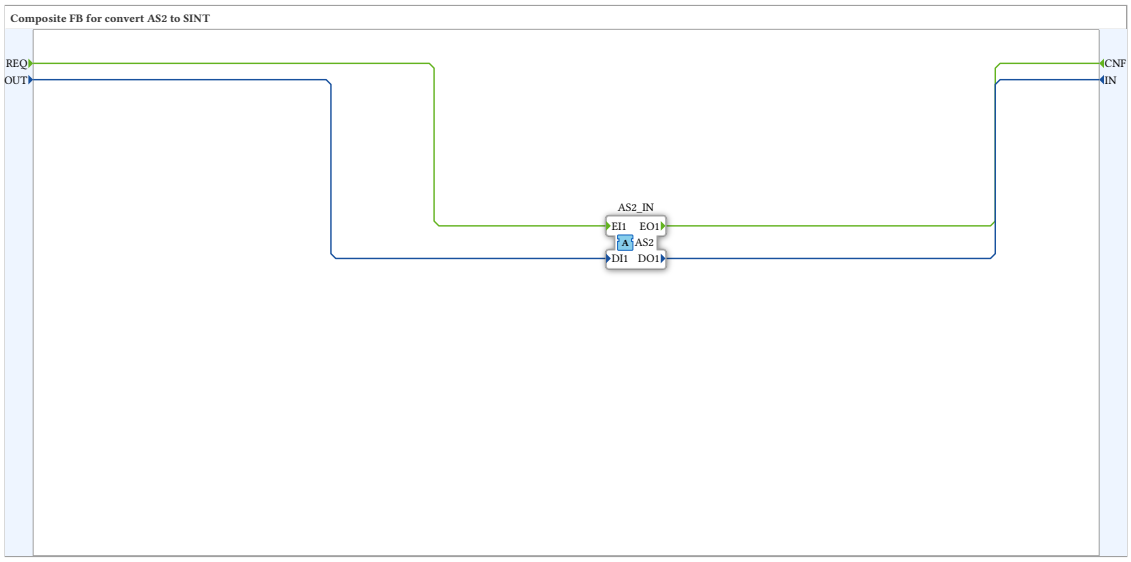
Application Scenarios

- Connecting `SINT` values from classic function block networks to a bidirectional `AS2` adapter

Summary

`AS2_SINT_TO_S` maps `SINT` event/data connections 1:1 onto the two directions of an `AS2` adapter.

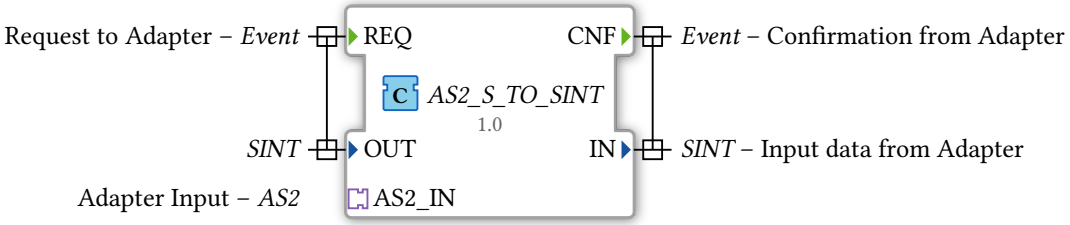
AS2_S_TO_SINT



• – ○ • – ○ • – ○ •

Introduction

`AS2_S_TO_SINT` is a composite FB that mediates between a `SINT` data pair (`OUT`/`IN`) and the bidirectional adapter `AS2`. In contrast to the plug variant `AS2_SINT_TO_R` it has an adapter **socket** (input). It completes the conversion family analogously to `AR2`/`AB2`/`AE2`.



Interface Structure

Event Inputs

- `REQ`: Request to the adapter. Associated data: `OUT`.

Event Outputs

- `CNF`: Confirmation from the adapter. Associated data: `IN`.

Data Inputs

- `OUT` (`SINT`): Value sent to the adapter.

Data Outputs

- `IN` (`SINT`): Value received from the adapter.

Adapters

Direction	Name	Adapter type	Comment
Socket (input)	`AS2_IN`	`adapter::types::bidirectional::AS2`	Adapter input.

Functionality

The block contains no internal FBs, only connections:

- `REQ` ⟶ `AS2_IN.EI1` and `OUT` ⟶ `AS2_IN.DI1`: a `REQ` event sends the value `OUT` through the adapter.

- `AS2_IN.EO1` ⟶ `CNF` and `AS2_IN.DO1` ⟶ `IN`: an event coming from the adapter is output as `CNF`, and the received value appears at `IN`.

Technical Details

- Pure pass-through without logic and without internal state.
- The data type is `SINT` (signed 8-bit integer).
- The naming `OUT`/`IN` is chosen from the adapter's point of view: `OUT` goes out to the adapter, `IN` comes in from the adapter.

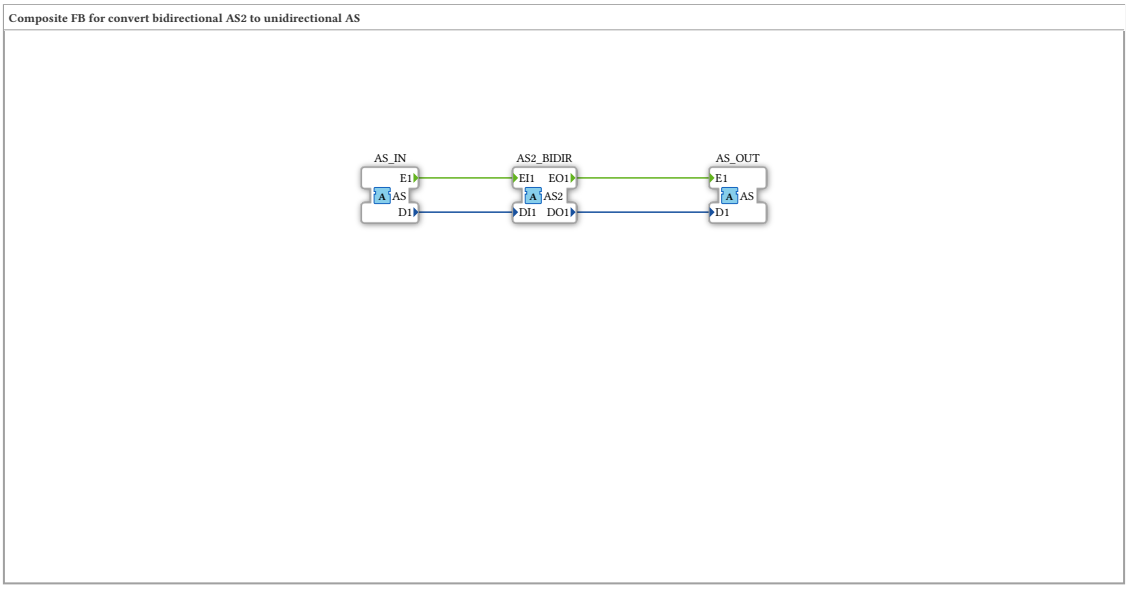
Application Scenarios

- Output a `SINT` value to a bidirectional `AS2` adapter and at the same time read its backward channel as a data output, when the adapter is present as a socket.

Summary

`AS2_S_TO_SINT` connects events and `SINT` data 1:1 with the bidirectional adapter `AS2` (socket variant).

AS2_TO_AS



• – ○ • – ○ • – ○ •

Introduction

The AS2_TO_AS function block is the mirror image of AS_TO_AS2: here the bidirectional AS2 tap sits as a socket on the left (input) side, complemented by a second unidirectional AS socket and a unidirectional AS plug.



Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AS2_BIDIR** (Socket): bidirectional tap whose incoming side is forwarded to `AS_OUT` and whose outgoing side is fed by `AS_IN`, of type `adapter::types::bidirectional::AS2`
- **AS_IN** (Socket): unidirectional SINT source, of type `adapter::types::unidirectional::AS`
- **AS_OUT** (Plug): mirrors the incoming (indication) side of `AS2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::AS`

Functionality

The block operates in two directions at once: whatever the socket `AS2_BIDIR` receives on its indication side (`EO1`/`DO1` — i.e. whatever a bidirectional peer attached to AS2_BIDIR reports) is output unchanged as a standalone unidirectional signal via the plug `AS_OUT`. Conversely, whatever arrives at the socket `AS_IN` (`E1`/`D1`) is published onto the request side of `AS2_BIDIR` (`EI1`/`DI1`) — so AS2_BIDIR forwards the same signal to the attached bidirectional peer.

Technical Details

- Three adapters instead of two: the direction received from the tap and the separately injected direction are deliberately kept apart rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `AS_OUT` only reflects what is reported via `AS2_BIDIR` by an attached bidirectional peer — without such a peer, `AS_OUT` stays idle

State Overview

The block is stateless:

- AS2_BIDIR.EO1 ⟶ AS_OUT.E1, AS2_BIDIR.DO1 ⟶ AS_OUT.D1
- AS_IN.E1 ⟶ AS2_BIDIR.EI1, AS_IN.D1 ⟶ AS2_BIDIR.DI1

Application Scenarios

- Reading the status of a bidirectional AS2 peer as a plain unidirectional AS signal, combined with injecting a separate unidirectional command signal
- Systems where a bidirectional peer needs to be monitored without converting the rest of the control logic to AS2
- Bridging a bidirectional AS2 zone and a unidirectional AS zone

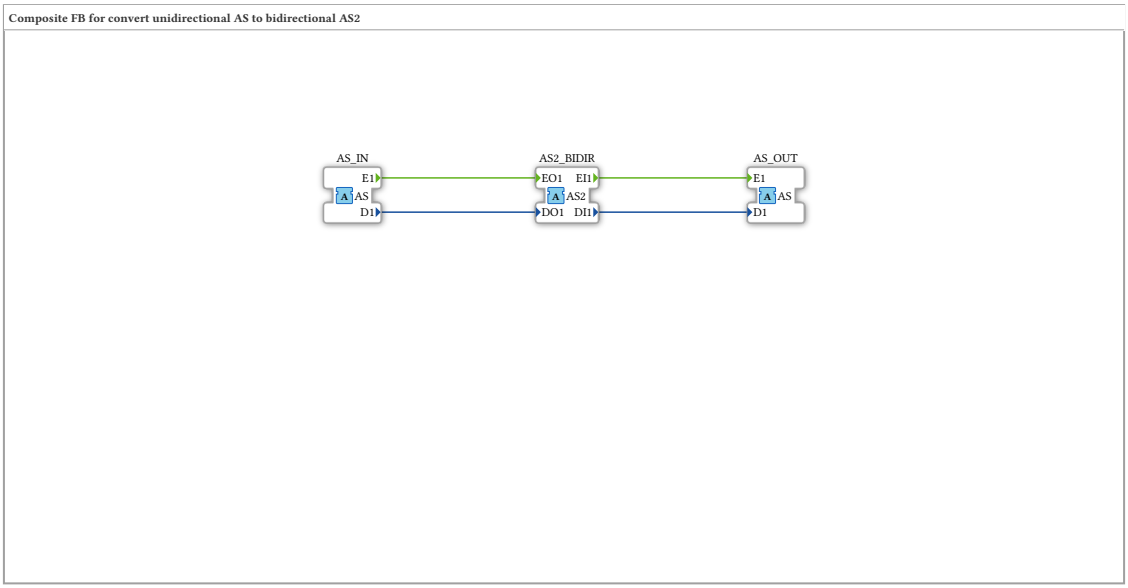
⚖️ Comparison with Similar Blocks

AS_TO_AS2 is the mirror image of this block — there, `AS2_BIDIR` sits as a plug on the right side.

Conclusion

AS2_TO_AS is not a simple converter but a tap point: it makes the state of a bidirectional AS2 peer visible as a standalone unidirectional AS signal while simultaneously injecting a separate unidirectional signal into the bidirectional side.

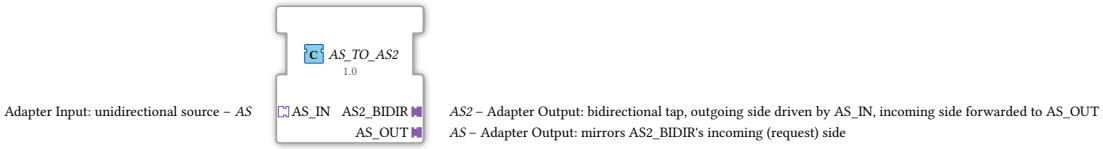
AS_TO_AS2



• – ○ • – ○ • – ○ •

Introduction

The `AS_TO_AS2` function block inserts a bidirectional `AS2` tap into a unidirectional `AS` chain (SINT). It uses **three** adapters: an incoming unidirectional AS signal is published onto the bidirectional AS2 interface, and whatever AS2 returns on its request side is output again as a standalone unidirectional AS signal.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AS_IN** (socket): unidirectional SINT source, type `adapter::types::unidirectional::AS`
- **AS2_BIDIR** (plug): bidirectional tap whose outgoing side is fed by `AS_IN` and whose incoming side is forwarded to `AS_OUT`, type `adapter::types::bidirectional::AS2`
- **AS_OUT** (plug): mirrors the incoming (request) side of `AS2_BIDIR` as a unidirectional signal, type `adapter::types::unidirectional::AS`

Functionality

The block operates in two directions at once: whatever arrives at the socket `AS_IN` (`E1`/`D1`) is published directly onto the outgoing side of `AS2_BIDIR` (`EO1`/`DO1`). Conversely, whatever `AS2_BIDIR` receives on its request side (`EI1`/`DI1`) is output unchanged as a standalone unidirectional signal via the plug `AS_OUT`.

AS2_BIDIR thus acts as a tap point that does not break the unidirectional chain but extends it with a bidirectional observation/injection point.

Technical Details

- Three adapters instead of two: the incoming signal direction and the direction coming back from the tap are deliberately kept separate
- Pure wiring, no logic or state, every destination variable has exactly one writer
- AS_OUT only reflects what comes back via AS2_BIDIR from an attached bidirectional peer; without such a peer, AS_OUT stays idle

State Overview

The block is stateless:

- AS_IN.E1 ⟶ AS2_BIDIR.EO1, AS_IN.D1 ⟶ AS2_BIDIR.DO1
- AS2_BIDIR.EI1 ⟶ AS_OUT.E1, AS2_BIDIR.DI1 ⟶ AS_OUT.D1

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional AS signal chain without breaking it
- Bridging a unidirectional AS zone and a bidirectional AS2 zone

⚖️ Comparison with Similar Blocks

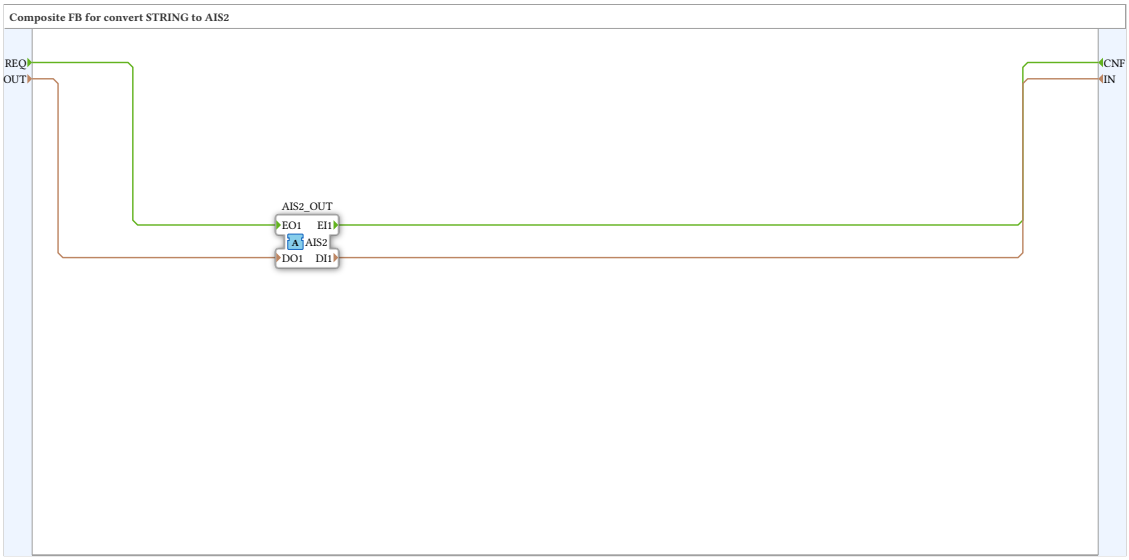
The counterpart in the opposite direction is the block AS2_TO_AS (there, AS2_BIDIR is a socket on the left side). The blocks of this conversion family exist analogously for the adapters AR/AR2, AB/AB2 and AE/AE2.

Conclusion

AS_TO_AS2 is not a simple converter but a tap point: it makes a unidirectional AS signal visible at a bidirectional AS2 point without breaking the original signal chain, while forwarding the request signal received there as a standalone AS signal.

STRING

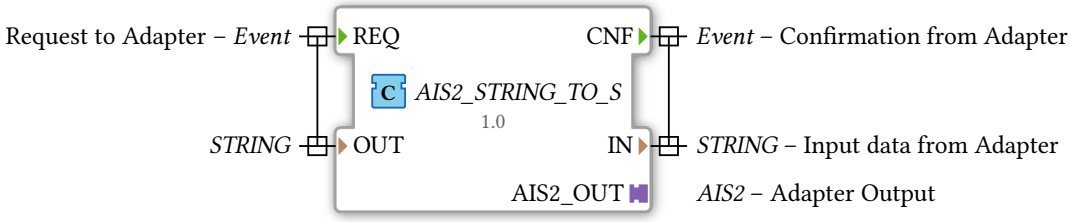
AIS2_STRING_TO_S



• – ○ • – ○ • – ○ •

Introduction

AIS2_STRING_TO_S is a composite function block that connects a bidirectional AIS2 adapter (plug AIS2_OUT) to a classic event/data interface of type STRING. The event REQ with the data input OUT is written to the adapter; event and value coming from the adapter appear as CNF with the data output IN. The block belongs to the conversion family (analogous to AR2/AB2/AE2).



Interface Structure

Event Inputs

- **REQ**: Request to the adapter (associated data: `OUT`)

Event Outputs

- **CNF**: Confirmation from the adapter (associated data: `IN`)

Data Inputs

- **OUT** (`STRING`): Value sent to the adapter

Data Outputs

- **IN** (`STRING`): Value received from the adapter (input data from adapter)

Adapters

- **AIS2_OUT** (plug): Bidirectional adapter of type `adapter::types::bidirectional::AIS2` (adapter output)

How it works

1. **Send**: `REQ` is routed to `AIS2_OUT.EO1` and `OUT` to `AIS2_OUT.DO1`.
2. **Receive**: `AIS2_OUT.EI1` is routed to `CNF` and `AIS2_OUT.DI1` to `IN`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- The data type is `STRING` on both sides, no value conversion takes place
- Version 1.0 (2026-09-12): conversion family completed

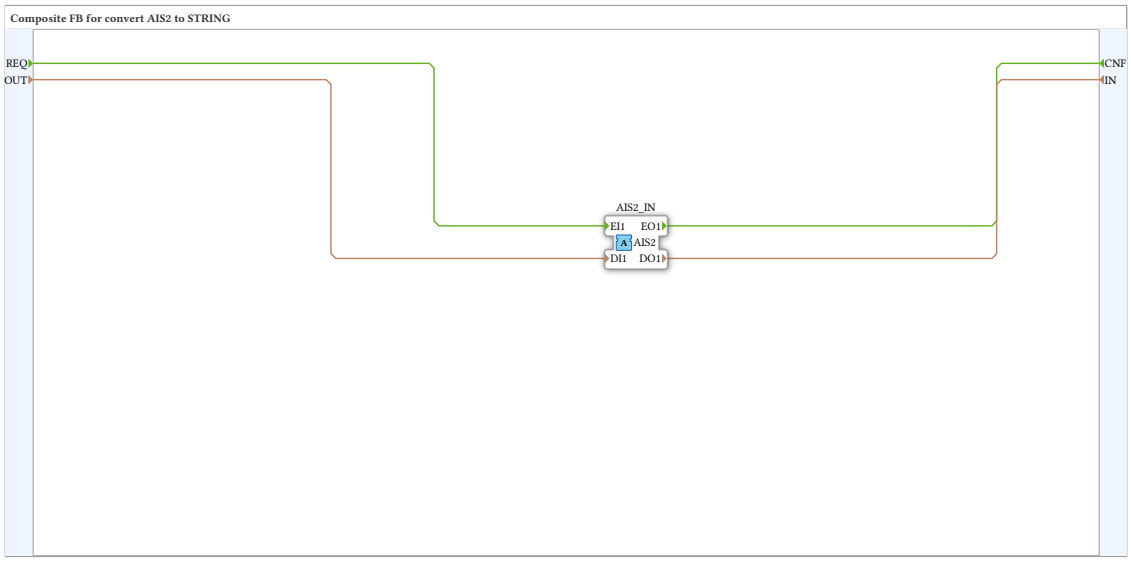
Application Scenarios

- Connecting a block with a simple `REQ`/`CNF` interface to a bidirectional `AIS2` adapter connection
- Bridge between event/data ports and adapter for `STRING` values

Summary

`AIS2_STRING_TO_S` translates a simple `STRING` event/data interface 1:1 into a bidirectional `AIS2` adapter and back.

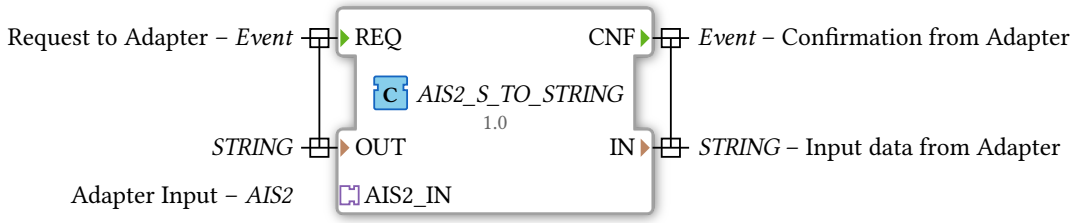
AIS2_S_TO_STRING



• – ◦ • – ◦ • – ◦ •

Introduction

`AIS2_S_TO_STRING` is a composite function block that maps the bidirectional adapter AIS2 onto plain event and STRING data ports. It belongs to the conversion family analogous to AR2/AB2/AE2.



Interface Structure

Event Inputs

- **REQ** (Request to Adapter): Sends the data `OUT` through the adapter (associated with `OUT`)

Event Outputs

- **CNF** (Confirmation from Adapter): Triggered when the adapter delivers an event on its incoming side (associated with `IN`)

Data Inputs

- **OUT** (STRING): Data sent to the adapter

Data Outputs

- **IN** (STRING): Data received from the adapter

Adapters

- **AIS2_IN** (Socket): Adapter of type `adapter::types::bidirectional::AIS2`

Functionality

The block consists of wiring only, without any logic of its own:

- `REQ` ⟶ `AIS2_IN.EI1` and `OUT` ⟶ `AIS2_IN.DI1`: The request event stream together with its data is published onto the outgoing side of the adapter.
- `AIS2_IN.EO1` ⟶ `CNF` and `AIS2_IN.DO1` ⟶ `IN`: Whatever is reported on the incoming side of the adapter appears as event `CNF` with the data `IN`.

Technical Details

- Stateless composite block without algorithms
- Every destination variable has exactly one writer
- The adapter data is passed through unchanged, no value conversion takes place

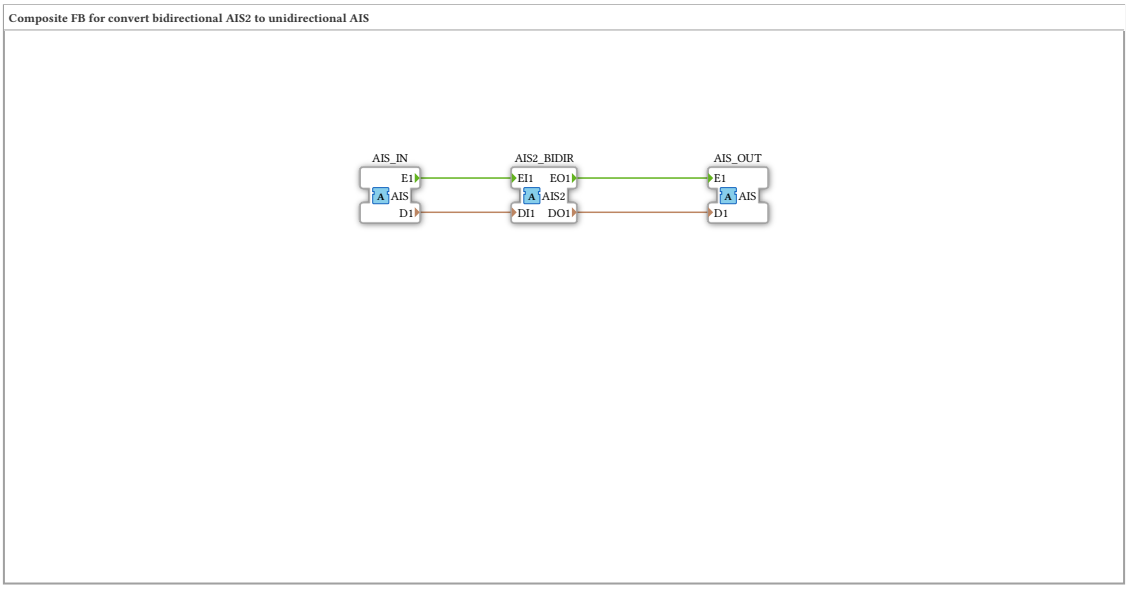
Application Scenarios

- Connecting classic event/data ports to a bidirectional AIS2 adapter
- Integrating blocks with STRING ports into adapter-based networks

Conclusion

`AIS2_S_TO_STRING` maps the bidirectional AIS2 adapter onto the ports `REQ`/`OUT` and `CNF`/`IN` without any logic.

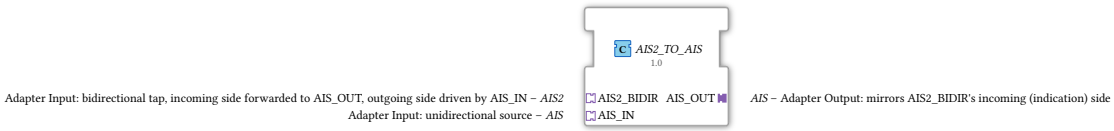
AIS2_TO_AIS



• – ○ • – ○ • – ○ •

Introduction

`AIS2_TO_AIS` is a composite function block that converts a **bidirectional** `AIS2` adapter into a **unidirectional** `AIS` adapter. The incoming side (`EO1`/`DO1`) of the bidirectional adapter is output as a plain unidirectional signal via `AIS_OUT`; via `AIS_IN` a value is written onto the outgoing side (`EI1`/`DI1`) of the bidirectional adapter.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **AIS2_BIDIR** (socket): Bidirectional adapter of type `adapter::types::bidirectional::AIS2` - the incoming side is forwarded to `AIS_OUT`, the outgoing side is driven by `AIS_IN`
- **AIS_IN** (socket): Unidirectional adapter of type `adapter::types::unidirectional::AIS` - unidirectional source
- **AIS_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AIS` - mirrors the incoming (indication) side of `AIS2_BIDIR`

How it works

1. `AIS2_BIDIR.EO1` is forwarded to `AIS_OUT.E1` and `AIS2_BIDIR.DO1` to `AIS_OUT.D1`.
2. `AIS_IN.E1` is forwarded to `AIS2_BIDIR.EI1` and `AIS_IN.D1` to `AIS2_BIDIR.DI1`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- Part of the conversion family (analogous to `AR2`/`AB2`/`AE2`); the data type is `STRING`

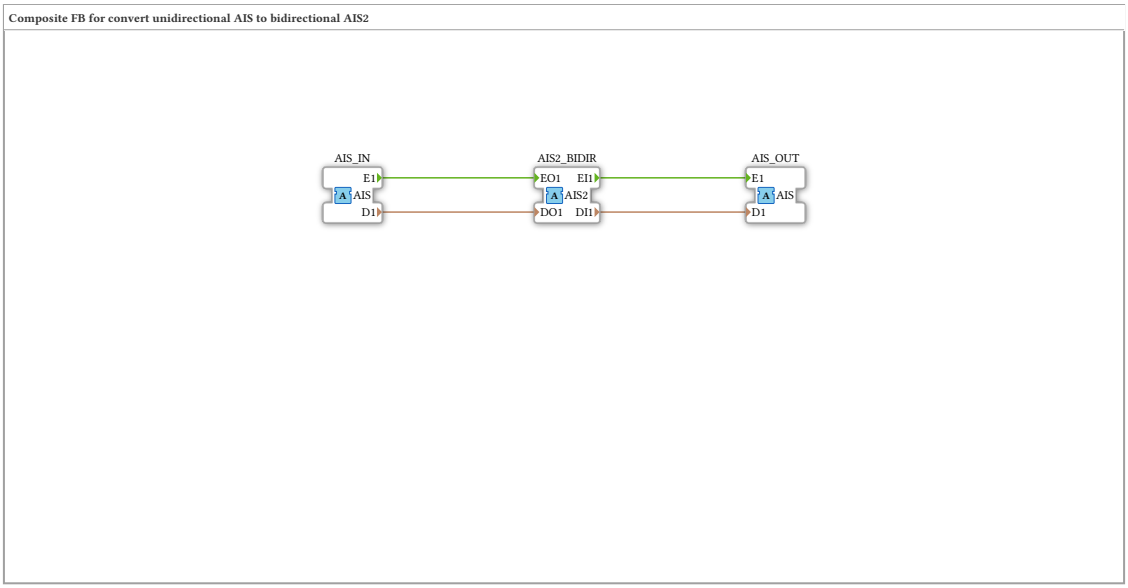
Application Scenarios

- Connecting a unidirectional `AIS` consumer or producer to a bidirectional `AIS2` adapter

Summary

`AIS2_TO_AIS` separates the two directions of a bidirectional `AIS2` adapter into a unidirectional output (`AIS_OUT`) and a unidirectional input (`AIS_IN`).

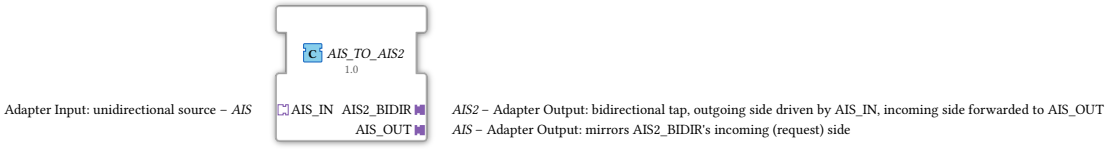
AIS_TO_AIS2



• – ○ • – ○ • – ○ •

Introduction

The `AIS_TO_AIS2` function block inserts a bidirectional `AIS2` tap into a unidirectional `AIS` chain (STRING). It uses **three** adapters: an incoming unidirectional AIS signal is published onto the bidirectional AIS2 interface, and whatever AIS2 returns on its request side is output again as a standalone unidirectional AIS signal.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AIS_IN** (socket): unidirectional STRING source, type `adapter::types::unidirectional::AIS`
- **AIS2_BIDIR** (plug): bidirectional tap whose outgoing side is fed by `AIS_IN` and whose incoming side is forwarded to `AIS_OUT`, type `adapter::types::bidirectional::AIS2`
- **AIS_OUT** (plug): mirrors the incoming (request) side of `AIS2_BIDIR` as a unidirectional signal, type `adapter::types::unidirectional::AIS`

Functionality

The block operates in two directions at once: whatever arrives at the socket `AIS_IN` (`E1`/`D1`) is published directly onto the outgoing side of `AIS2_BIDIR` (`EO1`/`DO1`). Conversely, whatever `AIS2_BIDIR` receives on its request side (`EI1`/`DI1`) is output unchanged as a standalone unidirectional signal via the plug `AIS_OUT`. `AIS2_BIDIR` thus acts as a tap point that does not break the unidirectional chain but extends it with a bidirectional observation/injection point.

Technical Details

- Three adapters instead of two: the incoming signal direction and the direction coming back from the tap are deliberately kept separate
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `AIS_OUT` only reflects what comes back via `AIS2_BIDIR` from an attached bidirectional peer; without such a peer, `AIS_OUT` stays idle

State Overview

The block is stateless:

- AIS_IN.E1 ⟶ AIS2_BIDIR.EO1, AIS_IN.D1 ⟶ AIS2_BIDIR.DO1
- AIS2_BIDIR.EI1 ⟶ AIS_OUT.E1, AIS2_BIDIR.DI1 ⟶ AIS_OUT.D1

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional AIS signal chain without breaking it
- Bridging a unidirectional AIS zone and a bidirectional AIS2 zone

⚖️ Comparison with Similar Blocks

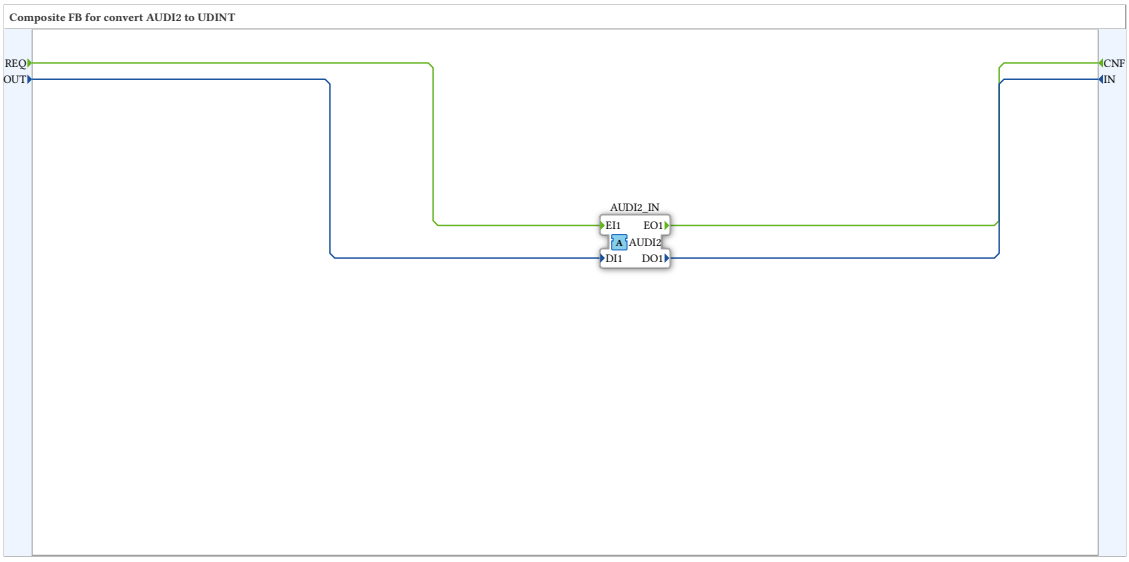
The counterpart in the opposite direction is the block `AIS2_TO_AIS` (there, `AIS2_BIDIR` is a socket on the left side). The blocks of this conversion family exist analogously for the adapters `AR`/`AR2`, `AB`/`AB2` and `AE`/`AE2`.

Conclusion

`AIS_TO_AIS2` is not a simple converter but a tap point: it makes a unidirectional AIS signal visible at a bidirectional AIS2 point without breaking the original signal chain, while forwarding the request signal received there as a standalone AIS signal.

UDINT

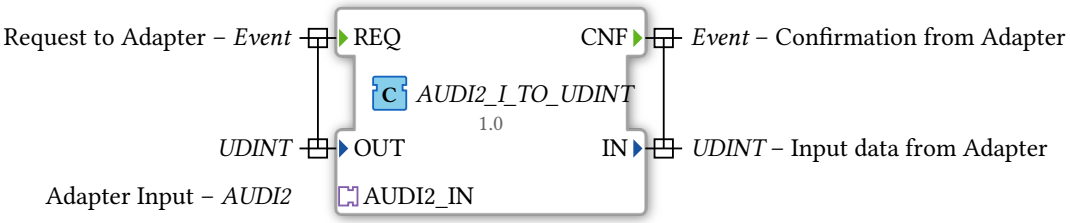
AUDI2_I_TO_UDINT



• – ◦ • – ◦ • – ◦ •

Introduction

`AUDI2_I_TO_UDINT` is a composite function block that maps the bidirectional adapter AUDI2 onto plain event and UDINT data ports. It belongs to the conversion family analogous to AR2/AB2/AE2.



Interface Structure

Event Inputs

- **REQ** (Request to Adapter): Sends the data `OUT` through the adapter (associated with `OUT`)

Event Outputs

- **CNF** (Confirmation from Adapter): Triggered when the adapter delivers an event on its incoming side (associated with `IN`)

Data Inputs

- **OUT** (UDINT): Data sent to the adapter

Data Outputs

- **IN** (UDINT): Data received from the adapter

Adapters

- **AUDI2_IN** (Socket): Adapter of type `adapter::types::bidirectional::AUDI2`

Functionality

The block consists of wiring only, without any logic of its own:

- `REQ` ⟶ `AUDI2_IN.EI1` and `OUT` ⟶ `AUDI2_IN.DI1`: The request event stream together with its data is published onto the outgoing side of the adapter.
- `AUDI2_IN.EO1` ⟶ `CNF` and `AUDI2_IN.DO1` ⟶ `IN`: Whatever is reported on the incoming side of the adapter appears as event `CNF` with the data `IN`.

Technical Details

- Stateless composite block without algorithms
- Every destination variable has exactly one writer
- The adapter data is passed through unchanged, no value conversion takes place

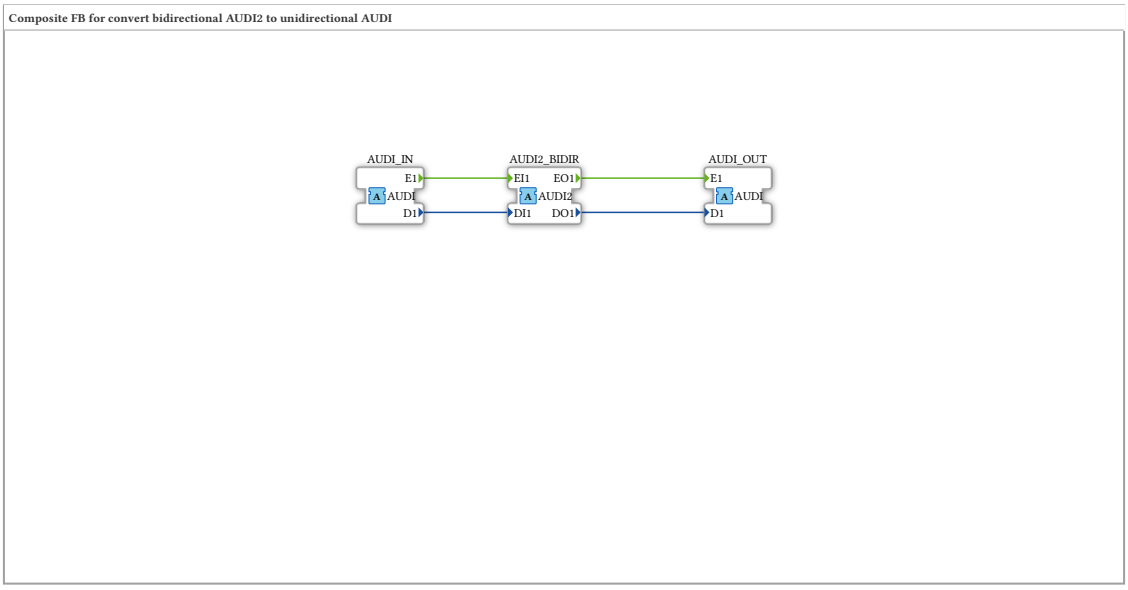
Application Scenarios

- Connecting classic event/data ports to a bidirectional AUDI2 adapter
- Integrating blocks with UDINT ports into adapter-based networks

Conclusion

`AUDI2_I_TO_UDINT` maps the bidirectional AUDI2 adapter onto the ports `REQ`/`OUT` and `CNF`/`IN` without any logic.

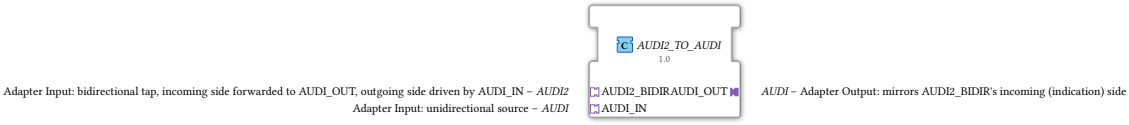
AUDI2_TO_AUDI



• — ○ • — ○ • — ○ •

Introduction

The AUDI2_TO_AUDI function block is the mirror image of AUDI_TO_AUDI2: here the bidirectional AUDI2 tap sits as a socket on the left (input) side, complemented by a second unidirectional AUDI socket and a unidirectional AUDI plug.



Interface Structure

Event Inputs

The function block has no direct event inputs — communication happens exclusively through the adapters.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AUDI2_BIDIR** (Socket): bidirectional tap whose incoming side is forwarded to `AUDI_OUT` and whose outgoing side is fed by `AUDI_IN`, of type `adapter::types::bidirectional::AUDI2`
- **AUDI_IN** (Socket): unidirectional UDINT source, of type `adapter::types::unidirectional::AUDI`
- **AUDI_OUT** (Plug): mirrors the incoming (indication) side of `AUDI2_BIDIR` as a unidirectional signal, of type `adapter::types::unidirectional::AUDI`

Functionality

The block operates in two directions at once: whatever the socket `AUDI2_BIDIR` receives on its indication side (`EO1`/`DO1` — i.e. whatever a bidirectional peer attached to AUDI2_BIDIR reports) is output unchanged as a standalone unidirectional signal via the plug `AUDI_OUT`. Conversely, whatever arrives at the socket `AUDI_IN` (`E1`/`D1`) is published onto the request side of `AUDI2_BIDIR` (`EI1`/`DI1`) — so AUDI2_BIDIR forwards the same signal to the attached bidirectional peer.

Technical Details

- Three adapters instead of two: the direction received from the tap and the separately injected direction are deliberately kept apart rather than mixed
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `AUDI_OUT` only reflects what is reported via `AUDI2_BIDIR` by an attached bidirectional peer — without such a peer, `AUDI_OUT` stays idle

State Overview

The block is stateless:

- AUDI2_BIDIR.EO1 ⟶ AUDI_OUT.E1, AUDI2_BIDIR.DO1 ⟶ AUDI_OUT.D1
- AUDI_IN.E1 ⟶ AUDI2_BIDIR.EI1, AUDI_IN.D1 ⟶ AUDI2_BIDIR.DI1

Application Scenarios

- Reading the status of a bidirectional AUDI2 peer as a plain unidirectional AUDI signal, combined with injecting a separate unidirectional command signal
- Systems where a bidirectional peer needs to be monitored without converting the rest of the control logic to AUDI2
- Bridging a bidirectional AUDI2 zone and a unidirectional AUDI zone

⚖️ Comparison with Similar Blocks

AUDI_TO_AUDI2 is the mirror image of this block — there, `AUDI2_BIDIR` sits as a plug on the right side.

Conclusion

AUDI2_TO_AUDI is not a simple converter but a tap point: it makes the state of a bidirectional AUDI2 peer visible as a standalone unidirectional AUDI signal while simultaneously injecting a separate unidirectional signal into the bidirectional side.

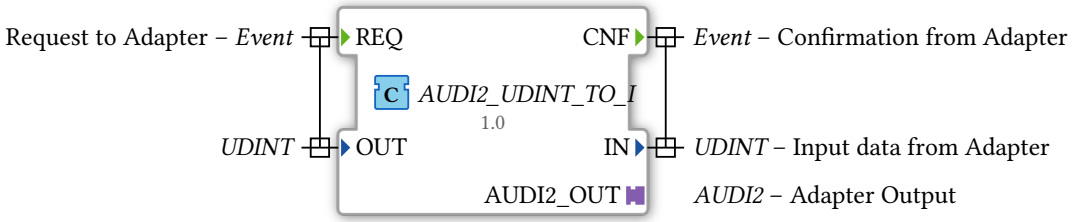
AUDI2_UDINT_TO_I



• – ○ • – ○ • – ○ •

Introduction

`AUDI2_UDINT_TO_I` is a composite function block that mediates between classic event/data connections of type `UDINT` and a **bidirectional** `AUDI2` adapter. On `REQ`, the input `OUT` is written onto the outgoing side of the adapter plug `AUDI2_OUT`; the value reported via the adapter's backward direction appears at output `IN` and is acknowledged with `CNF`.



Interface Structure

Event Inputs

- **REQ**: Request to the adapter. Associated data: `OUT`

Event Outputs

- **CNF**: Confirmation from the adapter. Associated data: `IN`

Data Inputs

- **OUT** (`UDINT`): Data sent to the adapter

Data Outputs

- **IN** (`UDINT`): Data received from the adapter

Adapters

- **AUDI2_OUT** (plug): Bidirectional adapter of type `adapter::types::bidirectional::AUDI2` (adapter output)

How it works

1. `REQ` is forwarded to `AUDI2_OUT.EO1`, `OUT` to `AUDI2_OUT.DO1`.
2. `AUDI2_OUT.EI1` is forwarded to `CNF`, `AUDI2_OUT.DI1` to `IN`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- Part of the conversion family (analogous to `AR2`/`AB2`/`AE2`)

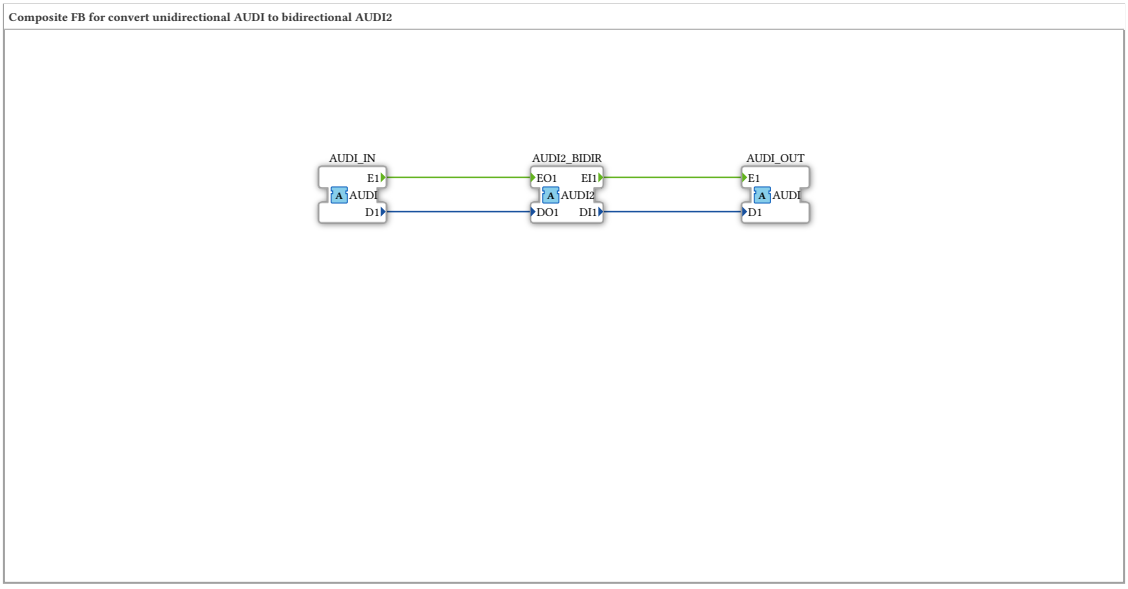
Application Scenarios

- Connecting `UDINT` values from classic function block networks to a bidirectional `AUDI2` adapter

Summary

`AUDI2_UDINT_TO_I` maps `UDINT` event/data connections 1:1 onto the two directions of an `AUDI2` adapter.

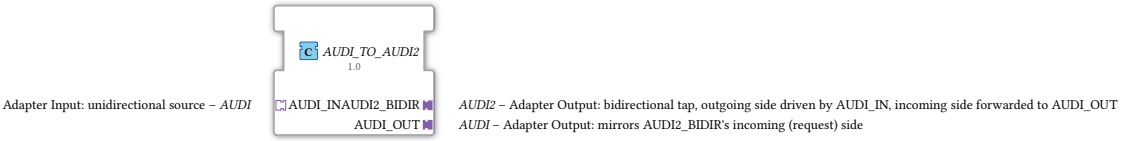
AUDI_TO_AUDI2



• — ○ • — ○ • — ○ •

Introduction

The `AUDI_TO_AUDI2` function block inserts a bidirectional `AUDI2` tap into a unidirectional `AUDI` chain (UDINT). It uses **three** adapters: an incoming unidirectional AUDI signal is published onto the bidirectional AUDI2 interface, and whatever AUDI2 returns on its request side is output again as a standalone unidirectional AUDI signal.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AUDI_IN** (socket): unidirectional UDINT source, type `adapter::types::unidirectional::AUDI`
- **AUDI2_BIDIR** (plug): bidirectional tap whose outgoing side is fed by `AUDI_IN` and whose incoming side is forwarded to `AUDI_OUT`, type `adapter::types::bidirectional::AUDI2`
- **AUDI_OUT** (plug): mirrors the incoming (request) side of `AUDI2_BIDIR` as a unidirectional signal, type `adapter::types::unidirectional::AUDI`

Functionality

The block operates in two directions at once: whatever arrives at the socket `AUDI_IN` (`E1`/`D1`) is published directly onto the outgoing side of `AUDI2_BIDIR` (`EO1`/`DO1`). Conversely, whatever `AUDI2_BIDIR` receives on its request side (`EI1`/`DI1`) is output unchanged as a standalone unidirectional signal via the plug `AUDI_OUT`. `AUDI2_BIDIR` thus acts as a tap point that does not break the unidirectional chain but extends it with a bidirectional observation/injection point.

Technical Details

- Three adapters instead of two: the incoming signal direction and the direction coming back from the tap are deliberately kept separate
- Pure wiring, no logic or state, every destination variable has exactly one writer
- `AUDI_OUT` only reflects what comes back via `AUDI2_BIDIR` from an attached bidirectional peer; without such a peer, `AUDI_OUT` stays idle

State Overview

The block is stateless:

- AUDI_IN.E1 ⟶ AUDI2_BIDIR.EO1, AUDI_IN.D1 ⟶ AUDI2_BIDIR.DO1
- AUDI2_BIDIR.EI1 ⟶ AUDI_OUT.E1, AUDI2_BIDIR.DI1 ⟶ AUDI_OUT.D1

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional AUDI signal chain without breaking it
- Bridging a unidirectional AUDI zone and a bidirectional AUDI2 zone

⚖️ Comparison with Similar Blocks

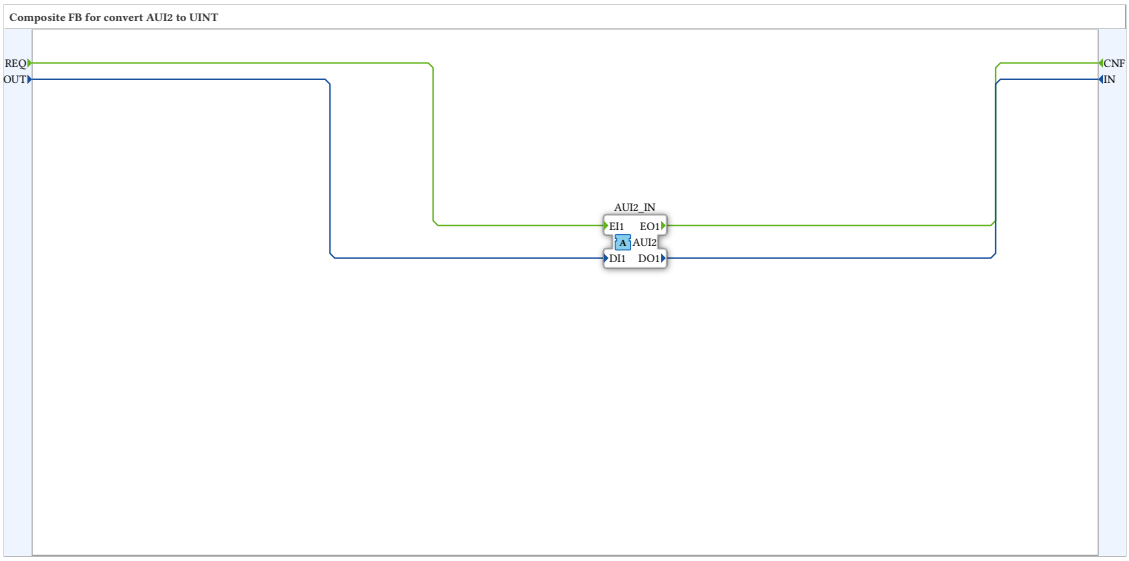
The counterpart in the opposite direction is the block `AUDI2_TO_AUDI` (there, `AUDI2_BIDIR` is a socket on the left side). The blocks of this conversion family exist analogously for the adapters `AR`/`AR2`, `AB`/`AB2` and `AE`/`AE2`.

Conclusion

`AUDI_TO_AUDI2` is not a simple converter but a tap point: it makes a unidirectional AUDI signal visible at a bidirectional AUDI2 point without breaking the original signal chain, while forwarding the request signal received there as a standalone AUDI signal.

UINT

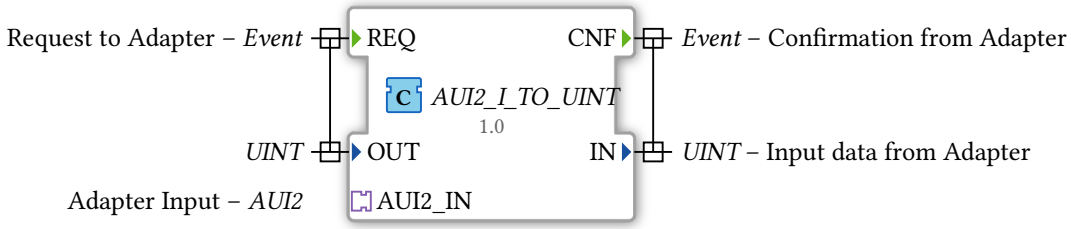
AUI2_I_TO_UINT



• – ○ • – ○ • – ○ •

Introduction

`AUI2_I_TO_UINT` is a composite FB that mediates between a `UINT` data pair (`OUT`/`IN`) and the bidirectional adapter `AUI2`. In contrast to the plug variant `AUI2_UINT_TO_R` it has an adapter **socket** (input). It completes the conversion family analogously to `AR2`/`AB2`/`AE2`.



Interface Structure

Event Inputs

- `REQ`: Request to the adapter. Associated data: `OUT`.

Event Outputs

- `CNF`: Confirmation from the adapter. Associated data: `IN`.

Data Inputs

- `OUT` (`UINT`): Value sent to the adapter.

Data Outputs

- `IN` (`UINT`): Value received from the adapter.

Adapters

Direction	Name	Adapter type	Comment
Socket (input)	`AUI2_IN`	`adapter::types::bidirectional::AUI2`	Adapter input.

Functionality

The block contains no internal FBs, only connections:

- `REQ` ⟶ `AUI2_IN.EI1` and `OUT` ⟶ `AUI2_IN.DI1`: a `REQ` event sends the value `OUT` through the adapter.
- `AUI2_IN.EO1` ⟶ `CNF` and `AUI2_IN.DO1` ⟶ `IN`: an event coming from the adapter is output as `CNF`, and the received value appears at `IN`.

Technical Details

- Pure pass-through without logic and without internal state.
- The data type is `UINT` (unsigned 16-bit integer).
- The naming `OUT`/`IN` is chosen from the adapter's point of view: `OUT` goes out to the adapter, `IN` comes in from the adapter.

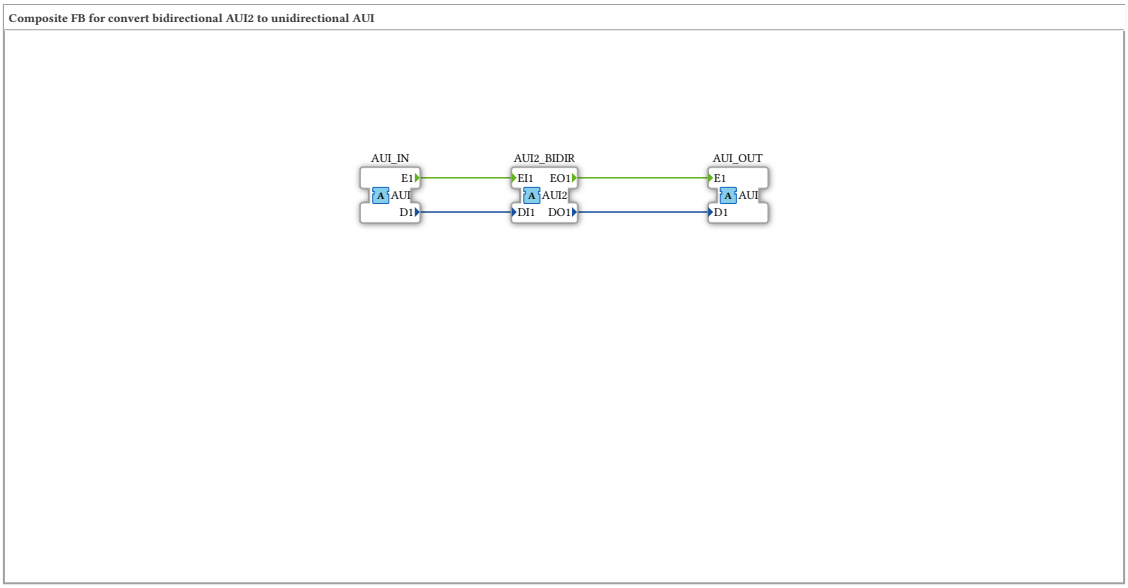
Application Scenarios

- Output a `UINT` value to a bidirectional `AUI2` adapter and at the same time read its backward channel as a data output, when the adapter is present as a socket.

Summary

`AUI2_I_TO_UINT` connects events and `UINT` data 1:1 with the bidirectional adapter `AUI2` (socket variant).

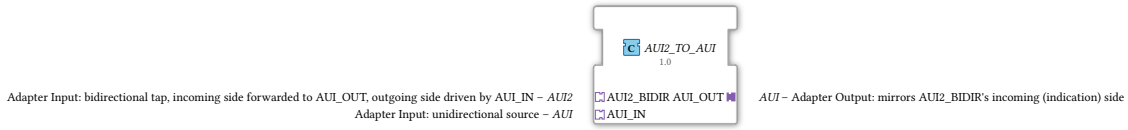
AUI2_TO_AUI



• — ○ • — ○ • — ○ •

Introduction

`AUI2_TO_AUI` is a composite function block that converts a **bidirectional** `AUI2` adapter into a **unidirectional** `AUI` adapter. The incoming side (`EO1`/`DO1`) of the bidirectional adapter is output as a plain unidirectional signal via `AUI_OUT`; via `AUI_IN` a value is written onto the outgoing side (`EI1`/`DI1`) of the bidirectional adapter.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **AUI2_BIDIR** (socket): Bidirectional adapter of type `adapter::types::bidirectional::AUI2` - the incoming side is forwarded to `AUI_OUT`, the outgoing side is driven by `AUI_IN`
- **AUI_IN** (socket): Unidirectional adapter of type `adapter::types::unidirectional::AUI` - unidirectional source
- **AUI_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AUI` - mirrors the incoming (indication) side of `AUI2_BIDIR`

How it works

1. `AUI2_BIDIR.EO1` is forwarded to `AUI_OUT.E1` and `AUI2_BIDIR.DO1` to `AUI_OUT.D1`.
2. `AUI_IN.E1` is forwarded to `AUI2_BIDIR.EI1` and `AUI_IN.D1` to `AUI2_BIDIR.DI1`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- Part of the conversion family (analogous to `AR2`/`AB2`/`AE2`)

Application Scenarios

- Connecting a unidirectional `AUI` consumer or producer to a bidirectional `AUI2` adapter

Summary

`AUI2_TO_AUI` separates the two directions of a bidirectional `AUI2` adapter into a unidirectional output (`AUI_OUT`) and a unidirectional input (`AUI_IN`).

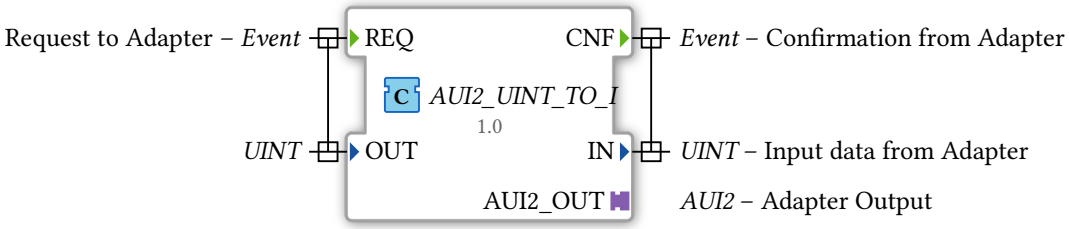
AUI2_UINT_TO_I



• – ○ • – ○ • – ○ •

Introduction

`AUI2_UINT_TO_I` is a composite function block that maps the plain event and `UINT` data ports `REQ`/`OUT` and `CNF`/`IN` onto the bidirectional adapter `AUI2` (as a plug). It belongs to the conversion family analogous to AR2/AB2/AE2.



Interface Structure

Event Inputs

- **REQ** (request to adapter): sends the data `OUT` through the adapter (associated with `OUT`)

Event Outputs

- **CNF** (confirmation from adapter): fires when the adapter delivers an event on its incoming side (associated with `IN`)

Data Inputs

- **OUT** (UINT): data that is sent to the adapter

Data Outputs

- **IN** (UINT): data received from the adapter

Adapters

- **AUI2_OUT** (plug): adapter of type `adapter::types::bidirectional::AUI2`

Functionality

The block consists of wiring only, without any logic of its own:

- `REQ` ⟶ `AUI2_OUT.EO1` and `OUT` ⟶ `AUI2_OUT.DO1`: the request event stream together with its data is published onto the outgoing side of the adapter.
- `AUI2_OUT.EI1` ⟶ `CNF` and `AUI2_OUT.DI1` ⟶ `IN`: whatever is reported on the incoming side of the adapter appears as the event `CNF` with the data `IN`.

Technical Details

- Stateless composite block without algorithms
- Every destination variable has exactly one writer
- The adapter data is passed through unchanged; no value conversion takes place
- The adapter is used as a plug here, so the block sits at the start of an AUI2 connection

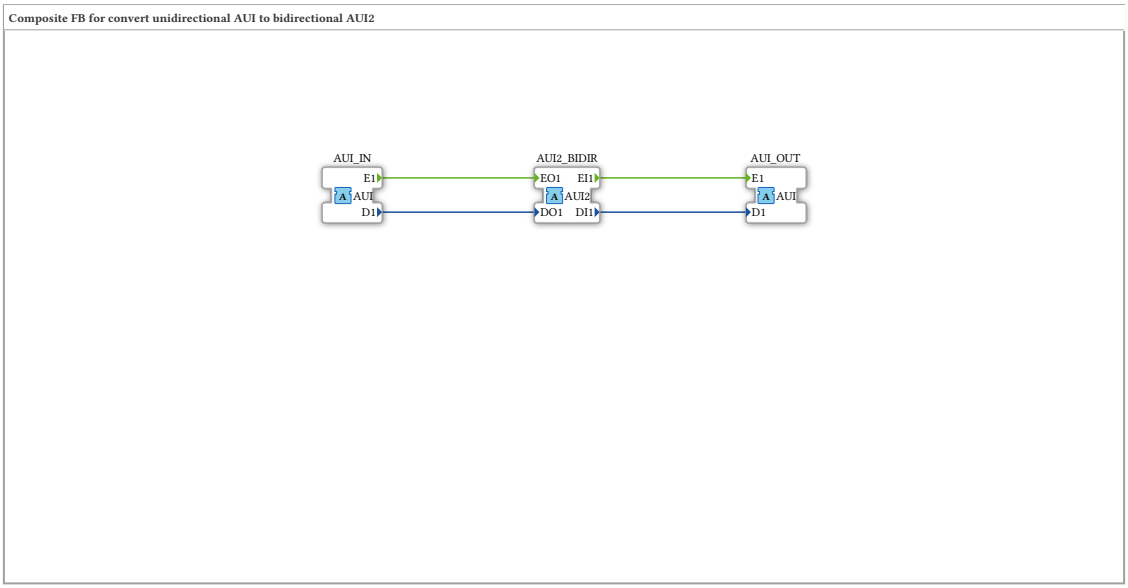
Application Scenarios

- Connecting classic event/data ports to a bidirectional AUI2 adapter
- Integrating blocks with `UINT` ports into adapter-based networks

Conclusion

`AUI2_UINT_TO_I` maps the ports `REQ`/`OUT` and `CNF`/`IN` onto the bidirectional AUI2 adapter without any logic.

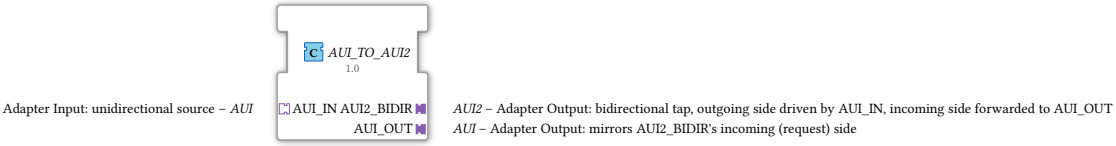
AUI_TO_AUI2



• – ○ • – ○ • – ○ •

Introduction

The `AUI_TO_AUI2` function block inserts a bidirectional `AUI2` tap into a unidirectional `AUI` chain (UINT). It uses **three** adapters: an incoming unidirectional AUI signal is published onto the bidirectional AUI2 interface, and whatever AUI2 returns on its request side is output again as a standalone unidirectional AUI signal.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **AUI_IN** (socket): unidirectional UINT source, type `adapter::types::unidirectional::AUI`
- **AUI2_BIDIR** (plug): bidirectional tap whose outgoing side is fed by `AUI_IN` and whose incoming side is forwarded to `AUI_OUT`, type `adapter::types::bidirectional::AUI2`
- **AUI_OUT** (plug): mirrors the incoming (request) side of `AUI2_BIDIR` as a unidirectional signal, type `adapter::types::unidirectional::AUI`

Functionality

The block operates in two directions at once: whatever arrives at the socket `AUI_IN` (`E1`/`D1`) is published directly onto the outgoing side of `AUI2_BIDIR` (`EO1`/`DO1`). Conversely, whatever `AUI2_BIDIR` receives on its request side (`EI1`/`DI1`) is output unchanged as a standalone unidirectional signal via the plug `AUI_OUT`.

AUI2_BIDIR thus acts as a tap point that does not break the unidirectional chain but extends it with a bidirectional observation/injection point.

Technical Details

- Three adapters instead of two: the incoming signal direction and the direction coming back from the tap are deliberately kept separate
- Pure wiring, no logic or state, every destination variable has exactly one writer
- AUI_OUT only reflects what comes back via AUI2_BIDIR from an attached bidirectional peer; without such a peer, AUI_OUT stays idle

State Overview

The block is stateless:

- AUI_IN.E1 ⟶ AUI2_BIDIR.EO1, AUI_IN.D1 ⟶ AUI2_BIDIR.DO1
- AUI2_BIDIR.EI1 ⟶ AUI_OUT.E1, AUI2_BIDIR.DI1 ⟶ AUI_OUT.D1

Application Scenarios

- Retrofitting a bidirectional diagnostic or control point into an existing unidirectional AUI signal chain without breaking it
- Bridging a unidirectional AUI zone and a bidirectional AUI2 zone

⚖️ Comparison with Similar Blocks

The counterpart in the opposite direction is the block AUI2_TO_AUI (there, AUI2_BIDIR is a socket on the left side). The blocks of this conversion family exist analogously for the adapters AR/AR2, AB/AB2 and AE/AE2.

Conclusion

AUI_TO_AUI2 is not a simple converter but a tap point: it makes a unidirectional AUI signal visible at a bidirectional AUI2 point without breaking the original signal chain, while forwarding the request signal received there as a standalone AUI signal.

unidirectional conversion

- 📖 Numeric vs. bitwise: the FORTE conversion trap

Numeric vs. bitwise: the FORTE conversion trap

• – ◦ • – ◦ • – ◦ •

Introduction

Every block in this folder (AB_TO_AR, AD_TO_ADI, AI_TO_AR, etc.) as well as the underlying standard function blocks iec61131::conversion::F_X_TO_Y (part of the 4diac IDE standard library, not vendored in this repository) convert a value from one IEC 61131 data type to another. For some of these combinations that is a **real numeric value conversion** for others it is a **pure bit-reinterpretation**, where the numeric value is deliberately ignored and the raw bit pattern is copied instead. Not knowing the difference easily produces a silent, hard-to-find bug — see AD_TO_AR_TODO.md in the source repository for the concrete case that triggered this page.

The four type categories

IEC 61131-3 distinguishes four relevant categories:

Category	Types	Meaning
ANY_BIT	BOOL, BYTE, WORD, DWORD, LWORD	pure bit patterns with no numeric semantics of their own
ANY_INT (signed)	SINT, INT, DINT, LINT	signed integers
ANY_INT (unsigned)	USINT, UINT, UDINT, ULINT	unsigned integers
ANY_REAL	REAL, LREAL	IEEE754 floating-point numbers

The adapter prefixes in this folder correspond to: AB=BYTE, AW=WORD, AD=DWORD, AL=LWORD, AX=BOOL, AS=SINT, AI=INT, ADI=DINT, ALI=LINT, AUS=USINT, AUI=UINT, AUDI=UDINT, AULI=ULINT, AR=REAL, ALR=LREAL.

The conversion matrix

Verified against the FORTE core (`core/include/forte/datatypes/forte_any.h`, `CIEC_ANY::cast<U,T>`, and `forte_real.cpp/forte_lreal.cpp`, `CIEC_REAL::castRealData`) — the same logic behind every `F_X_TO_Y` block and every adapter wrapper in this folder:

Source ⊠ Destination	→ ANY_BIT	→ ANY_INT	→ ANY_REAL
ANY_BIT (except BOOL)	bit copy (structural, no numeric meaning)	bit-reinterpretation — **value-preserving** if the destination is same-or-wider width; truncates otherwise	⚠ **bit-reinterpretation — NOT a numeric value!** IEEE754 misinterpretation
BOOL	parity/LSB test	numeric (0/1)	numeric (0.0/1.0) — special case, see below
ANY_INT	stores the bit pattern (expected behavior for a bit-string destination)	numeric (sign-/zero-extension is safe, narrowing may truncate)	**numeric** (correct cast)
ANY_REAL	bit extraction (intentional, e.g. serialization via `F_REAL_TO_DWORD`)	numeric (rounding, `llrint`)	numeric (widen/narrow precision)

The one genuine trap is therefore the **ANY_BIT (except BOOL) → ANY_REAL** cell: using `BYTE`/`WORD`/`DWORD`/`LWORD` as the source of a conversion to `REAL`/`LREAL`. In this library that concretely affects two blocks:

- `AD_TO_AR` (DWORD→REAL) — safe replacement: `AD_TO_AR_NUM`
- `AL_TO_ALR` (LWORD→LREAL) — safe replacement: `AL_TO_AULI` + `AULI_TO_ALR`

(There are no `AB_TO_AR`/`AB_TO_ALR`/`AW_TO_AR`/`AW_TO_ALR` pairs in this library — only width-matched bit↔real combinations were offered as adapters.)

Why is BOOL→REAL safe even though BOOL belongs to ANY_BIT? FORTE treats BOOL as an explicit special case when casting to REAL (`case e_BOOL: setTFLOAT(...)` instead of the generic bit-string copy) — the one exception in the matrix.

Why is ANY_BIT→ANY_REAL implemented this way at all?

It's not arbitrary: for bit-string destinations (`ANY_BIT→ANY_BIT`) and for the reverse direction `ANY_REAL→ANY_BIT`, bit-reinterpretation is exactly the intended, documented behavior — e.g. to pack a REAL's IEEE754 bit pattern into a DWORD for transport (`F_REAL_TO_DWORD`) and later unpack it again with `AD_TO_AR`. The pitfall is **confusion**: the same mechanism gets used by mistake when a raw counter or analog value (not a serialized bit pattern) is actually meant to become that numeric value in REAL.

Practical rule of thumb

- **Does the value come from an `F_REAL_TO_X`/`F_X_TO_REAL` round trip, or from fieldbus/protocol deserialization where bit patterns are explicitly transported?** → Bit-reinterpretation is correct (`AD_TO_AR`, `AL_TO_ALR`).
- **Is the value a raw counter, analog, or other integer value meant to carry the same numeric value in REAL?** → Use the numeric variant (`AD_TO_AR_NUM`, or in general: chain through the matching `ANY_INT` intermediate type, e.g. `AD_TO_AUDI` → `AUDI_TO_AR`).
- **Every other cell of the matrix** is either unambiguously numeric or unambiguously structural (a bit operation) — without the hidden misinterpretation risk of the ANY_BIT→ANY_REAL cell. Every block in this folder carries a short note about this in its "Technical Details" section.

Also affects the standard conversion blocks

The adapter blocks documented here (`AD_TO_AR` etc.) are thin wrappers around the standard 4diac function blocks `iec61131::conversion::F_X_TO_Y` (e.g. `F_DWORD_TO_REAL`). These standard blocks are part of the 4diac IDE distribution itself (not vendored in this repository) and are subject to exactly the same matrix

— `F_DWORD_TO_REAL`, `F_WORD_TO_REAL`, `F_BYTE_TO_LREAL` etc. are likewise bit-reinterpretations, while `F_DINT_TO_REAL`, `F_UDINT_TO_REAL` etc. are numerically correct.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numeric vs. Bitwise: The Conversion Trap in FORTE

• – ◦ • – ◦ • – ◦ •

Introduction

All the building blocks in this folder (`AB_TO_AR`, `AD_TO_ADI`, `AI_TO_AR`, etc.) and the underlying standard function blocks `iec61131::conversion::F_X_TO_Y` (part of the 4diac IDE standard library, not vendored in this repository) convert a value from one IEC 61131 data type to another. For some of these combinations, this is a **true numeric value conversion** for others, it is a **pure bit reinterpretation**, where the numeric value is deliberately ignored and the raw bit pattern is used instead. Those who don't know the difference can easily create a silent, hard-to-find bug—see `AD_TO_AR_TODO.md` in the source repository for the specific case that triggered this page.

The Four Type Categories

IEC 61131-3 distinguishes four relevant categories:

Category	Types	Meaning
ANY_BIT	`BOOL`, `BYTE`, `WORD`, `DWORD`, `LWORD`	pure bit patterns without their own numerical semantics
ANY_INT (signed)	`SINT`, `INT`, `DINT`, `LINT`	Signed Integers
ANY_INT (Unsigned)	`USINT`, `UINT`, `UDINT`, `ULINT`	Unsigned Integers
ANY_REAL	`REAL`, `LREAL`	IEEE 754 Floating-Point Numbers

The adapter prefixes in this folder correspond to: `AB`=BYTE, `AW`=WORD, `AD`=DWORD, `AL`=LWORD, `AX`=BOOL, `AS`=SINT, `AI`=INT, `ADI`=DINT, `ALI`=LINT, `AUS`=USINT, `AUI`=UINT, `AUDI`=UDINT, `AULI`=ULINT, `AR`=REAL, `ALR`=LREAL.

The Conversion Matrix

Verified in the FORTE core (`core/include/forte/datatypes/forte_any.h`, `CIEC_ANY::cast<U,T>`, as well as `forte_real.cpp`/`forte_lreal.cpp`, `CIEC_REAL::castRealData`) — the same logic behind every `F_X_TO_Y` block and every adapter wrapper in this folder:

Source < Destination	→ BOOL	→ ANY_BIT (except BOOL)	→ ANY_INT	→ ANY_REAL
ANY_BIT (except BOOL)	LSB test	Bit-pattern transfer (structural, no numeric value)	Bit reinterpretation — a **wider** target preserves the numeric value (zero extension); a **same-width** target keeps the bit pattern but a signed target can change the numeric value (e.g. `0xFF` as `BYTE` becomes `-1` as `SINT`, not `255`); a **narrower** target truncates the high bits	⚠️ **Bit reinterpretation — NO numeric value!** IEEE 754 misinterpretation

BOOL	Identity	Numeric `0`/`1`	numeric (0/1)	numeric (0.0/1.0) — special case, see below
ANY_INT	Boolean conversion (`!= 0`)	Bit-pattern transfer (expected behavior for a bit string target)	numeric (sign expansion/zero expansion safe, narrowing can truncate)	**numeric** (correct cast)
ANY_REAL	Boolean conversion (`!= 0.0`)	Bit extraction (intentional, e.g., serialization via `F_REAL_TO_DWORD`)	numeric (rounding, `llrint`)	numeric (rounding up/down precision)

The most conspicuous pitfall is the cell **ANY_BIT (except BOOL) → ANY_REAL** (highlighted in red): `BYTE`/`WORD`/`DWORD`/`LWORD` as the source of a conversion to `REAL`/`LREAL`. In this library, this specifically affects two building blocks:

A further pitfall, already noted in the matrix above, affects `ANY_BIT` (except BOOL) ⟶ `ANY_INT` at the same bit width: the bit pattern is preserved, but a signed target can change the numeric value (e.g. `0xFF` as `BYTE` becomes `-1` as `SINT`). Unlike `ANY_BIT`⟶`ANY_REAL`, the resulting value stays within a plausible numeric range instead of being turned into nonsense by an IEEE 754 misinterpretation, so the error is usually caught faster while debugging.

- `AD_TO_AR` (DWORD⟶REAL) — safe replacement: `AD_TO_AR_NUM`
- `AL_TO_ALR` (LWORD⟶LREAL) — safe replacement: `AL_TO_AULI` + `AULI_TO_ALR`

(There are no `AB_TO_AR`/`AB_TO_ALR`/`AW_TO_AR`/`AW_TO_ALR` pairs in this library — only width-matched bit↔real combinations were offered as adapters.)

Why is BOOL⟶REAL safe even though BOOL belongs to ANY_BIT? FORTE treats BOOL as an explicit special case when casting to REAL (`case e_BOOL: setTFLOAT(...)` instead of the generic bit string copy) — the only exception in the matrix.

Why is ANY_BIT⟶ANY_REAL implemented this way at all?

Not arbitrarily: For bit string destinations (`ANY_BIT`⟶`ANY_BIT`) and for the return path `ANY_REAL`⟶`ANY_BIT`, bit reinterpretation is precisely the desired, documented behavior — e.g., For example, to pack the IEEE 754 bit pattern of a REAL into a DWORD for transmission (`F_REAL_TO_DWORD`) and later unpack it again using `AD_TO_AR`. The problem is the **risk of confusion**: the same mechanism is incorrectly used when a raw counter or analog value (not a serialized bit pattern) is actually intended to be numerically transferred to REAL.

Practical Rule of Thumb

- **Does the value come from a `F_REAL_TO_X`/`F_X_TO_REAL` round trip or a fieldbus/protocol deserialization where bit patterns are explicitly transmitted?** ⟶ Bit reinterpretation is correct (`AD_TO_AR`, `AL_TO_ALR`).
- **Is the value a raw counter, analog, or other integer value that should be represented as the same numeric value in REAL?** ⟶ Use the numeric variant (`AD_TO_AR_NUM`, or more generally: a two-stage conversion via the appropriate intermediate type `ANY_INT`, e.g., `AD_TO_AUDI` ⟶ `AUDI_TO_AR`).
- **All other cells in the matrix** are either uniquely numeric or uniquely structural (bit operation) — without the hidden risk of misinterpretation inherent in the ANY_BIT⟶ANY_REAL cell. Each component in this folder includes a brief note about this in its "Technical Specifications" section.

Also applies to the standard conversion blocks

The adapter blocks documented here (`AD_TO_AR` etc.) are thin wrappers around the standard 4diac function blocks `iec61131::conversion::F_X_TO_Y` (e.g., `F_DWORD_TO_REAL`).These standard building blocks are part of the 4diac IDE distribution itself (not vendored in this repository) and are subject to the exact same matrix—`F_DWORD_TO_REAL`, `F_WORD_TO_REAL`, `F_BYTE_TO_LREAL`, etc., are bit reinterpretations, while `F_DINT_TO_REAL`, `F_UDINT_TO_REAL`, etc., are numerically correct.

🌐 Related topic subpages on ms-muc-docs.de

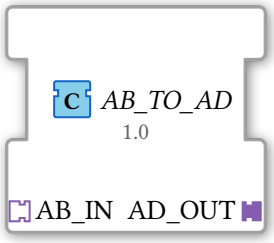
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Numeric Index (unidirectional)

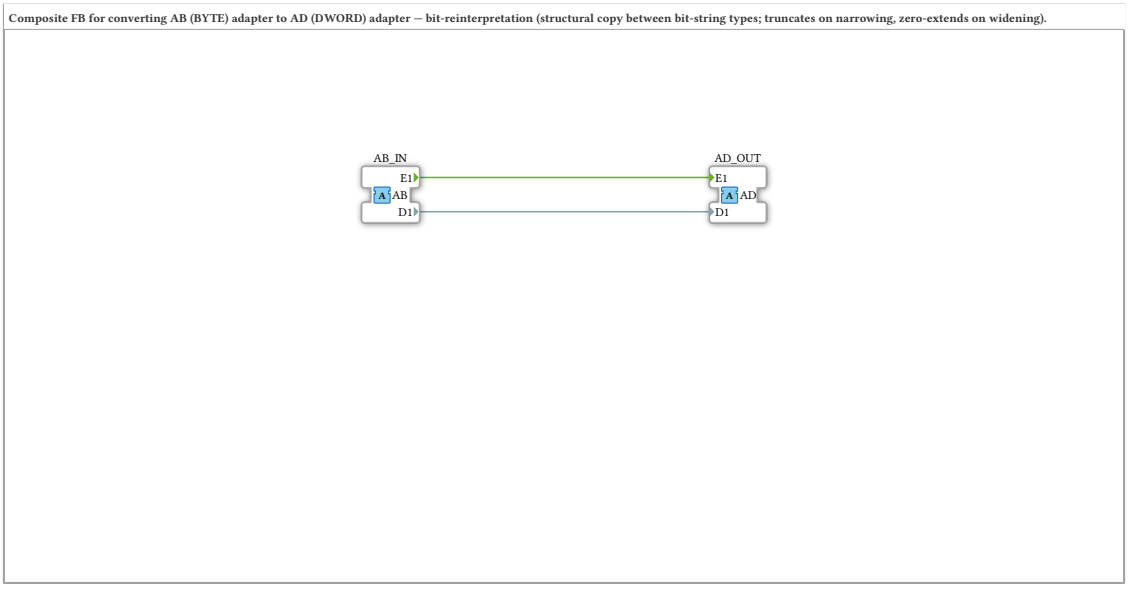
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AB_TO_AD

BYTE Adapter Input – *AB*



AD – DWORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AB_TO_AD** is a composite function block (FB) that converts a BYTE adapter (type `AB`) into a DWORD adapter (type `AD`). It serves as an interface converter in unidirectional communication and enables the easy connection of components based on different data word widths.

Interface Structure

Event Inputs

- **AB_IN.E1** – Event input from the BYTE adapter. It is forwarded directly to the DWORD adapter.

Event Outputs

- **AD_OUT.E1** – Event output to the DWORD adapter. Corresponds to the forwarded event of **AB_IN.E1**.

Data Inputs

- **AB_IN.D1** – Data input from the BYTE adapter (1 byte data width). Passed directly to the DWORD adapter.

Data Outputs

- **AD_OUT.D1** – Data output to the DWORD adapter (4 bytes data width). Contains the same value as the input **AB_IN.D1**, but in DWORD (Double Word) format.

Adapters

- **AB_IN** (Socket) – Unidirectional BYTE adapter (type `AB`) that serves as an input.
- **AD_OUT** (Plug) – Unidirectional DWORD adapter (type `AD`) that serves as an output.

Functionality

This function block connects the interfaces of a BYTE adapter (input) to those of a DWORD adapter (output). Every incoming event (**E1**) is forwarded to the output without delay or processing. Similarly, the incoming data value (**D1**) is passed to the output unchanged, but as a 4-byte DWORD. The function block has no logic of its own; it functions solely as a passthrough between the two adapter types.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Unidirectionality:** The function block supports only one direction of data flow, from the BYTE adapter to the DWORD adapter. Reverse data transfer is not supported.
- **Data Width Adjustment:** The incoming BYTE value (1 byte) is embedded within a DWORD value (4 bytes). The least significant byte of the DWORD corresponds to the incoming BYTE; the more significant bytes are set to zero (or remain unchanged, depending on the implementation of the target adapter).
- **Simple Coupling:** The FB is a pure connection block in the FBNetwork and does not require its own state machine or algorithms.

State Overview

The block has no state logic of its own, as it functions solely as a connection between the adapters. Its behavior is event-driven: Every event at the input is immediately passed on to the output.

Application Scenarios

- **System Integration:** If a subsystem uses BYTE adapters, but a subsequent subsystem expects DWORD adapters, `AB_TO_AD` can be used as an intermediary converter.
- **Data Aggregation:** Several such blocks can be used to combine individual bytes into a DWORD if the source adapters each only supply one byte.
- **Protocol Adaptation:** In fieldbus or sensor connections where the data width varies between components, the block serves as a simple adaptation layer.

Comparison with Similar Modules

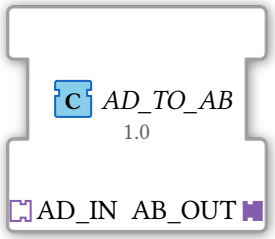
Analog converter modules exist for other data widths, e.g., `AB_TO_WORD` (BYTE to WORD) or `WORD_TO_DWORD`. This module specializes in the combination `AB` ⟶ `AD` and is therefore ideal for directly replacing or coupling corresponding adapters. Unlike more complex converters with data manipulation or buffers, it operates without delay and without additional logic.

Conclusion

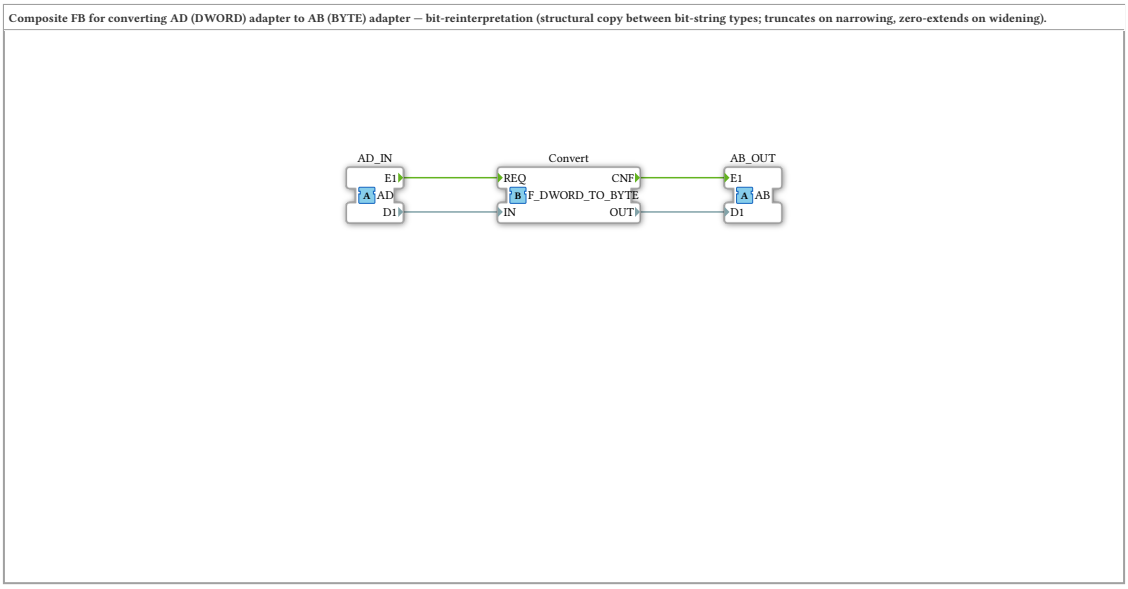
AB_TO_AD` is a lean, clearly structured converter module that enables the unidirectional conversion of a BYTE adapter to a DWORD adapter. Its simple pass-through makes it ideal for the rapid integration of different interfaces into IEC 61499-based control systems without introducing additional functional complexity.

AD_TO_AB

DWORD Adapter Input – *AD*



AB – BYTE Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The **AD_TO_AB** function block is a composite module for the unidirectional conversion of a 32-bit DWORD adapter (AD) to an 8-bit BYTE adapter (AB). It serves as a bridge between components designed for different data widths and encapsulates the necessary type conversion according to IEC 61131-3.

Interface Structure

Event Inputs

The module has no directly visible event inputs. Event control is handled via the **AD_IN** adapter.

Event Outputs

The module has no directly visible event outputs. Event feedback is provided via the **AB_OUT** adapter.

Data Inputs

This function block has no directly visible data inputs. The DWORD value is provided via the **AD_IN** adapter.

Data Outputs

This function block has no directly visible data outputs. The BYTE value is output via the **AB_OUT** adapter.

Adapters

Adapter	Type	Direction	Description
AD_IN	`adapter::types::unidirectional::AD`	Socket (Input)	Accepts a DWORD (32-bit) and an associated event (here `E1`).
AB_OUT	`adapter::types::unidirectional::AB`	Plug (Output)	Outputs a converted byte (8-bit) and an acknowledgment event (here `E1`).

Both adapters are unidirectional. The exact signature of the adapters (number of event and data channels) corresponds to the types defined in the 4diac library.

Functionality

The function block operates purely event-driven and encapsulates the conversion using the internal base function block **F_DWORD_TO_BYTE** from the IEC 61131-1 library.

1. An incoming event at **AD_IN.E1** triggers the call to the internal conversion function block via its input `REQ`.
2. The value from **AD_IN.D1** (DWORD) is forwarded to the converter's data input `IN`.
3. After the conversion is complete, the converter signals the readiness of the result with `CNF`.
4. This event is passed to **AB_OUT.E1**, and the converted BYTE value (from `OUT`) is output to **AB_OUT.D1**.

The conversion is performed according to IEC 61131-3: The least significant 8 bits of the incoming DWORD are interpreted as a BYTE.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- The module is implemented as a composite, meaning it does not contain its own state machine but delegates all logic to the internal base module.
- Neither buffering nor error handling is implemented; data transfer occurs directly synchronously with the event.
- This module is designed for use in systems that require a clear separation between adapters for different data widths.

State Overview

Since this is a pure composite module without its own state machine, there is no explicit state machine. Its behavior is entirely determined by the internal **F_DWORD_TO_BYTE** module, which has a simple wait state (IDLE) and a processing state.

Application Scenarios

- **Connecting Sensors/Actuators**: A sensor delivers 32-bit raw data via an AD adapter, which, after conversion to an 8-bit value, is passed on to a downstream BYTE adapter.
- **Protocol Conversion**: In a communication chain, a DWORD interface is mapped to a BYTE interface, e.g., when integrating older fieldbus devices.
- **Data Width Reduction**: Targeted extraction of the least significant byte from a DWORD for subsequent processing steps.

Comparison with Similar Function Blocks

Function Block	Conversion	Direction	Adapter Type
AD_TO_AB	DWORD ⟶ BYTE	Unidirectional	AD ↔ AB
AD_TO_AW	DWORD ⟶ WORD	Unidirectional	AD ↔ AW
AW_TO_AB	WORD ⟶ BYTE	Unidirectional	AW ↔ AB

The **AD_TO_AB** function differs in its specific target data width and the use of the corresponding adapters. It is specifically designed for systems that use standard unidirectional adapters.

Conclusion

The composite block **AD_TO_AB** offers a simple, event-driven way to convert a DWORD adapter to a BYTE adapter. It cleanly encapsulates the type conversion and can be integrated into existing 4diac networks without its own state logic. Reusing the standardized **F_DWORD_TO_BYTE** block guarantees correct behavior according to IEC 61131-3.

–

🌐 Related topic subpages on ms-muc-docs.de

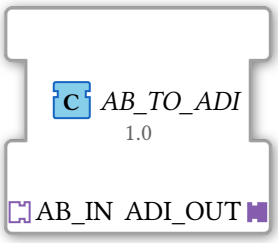
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

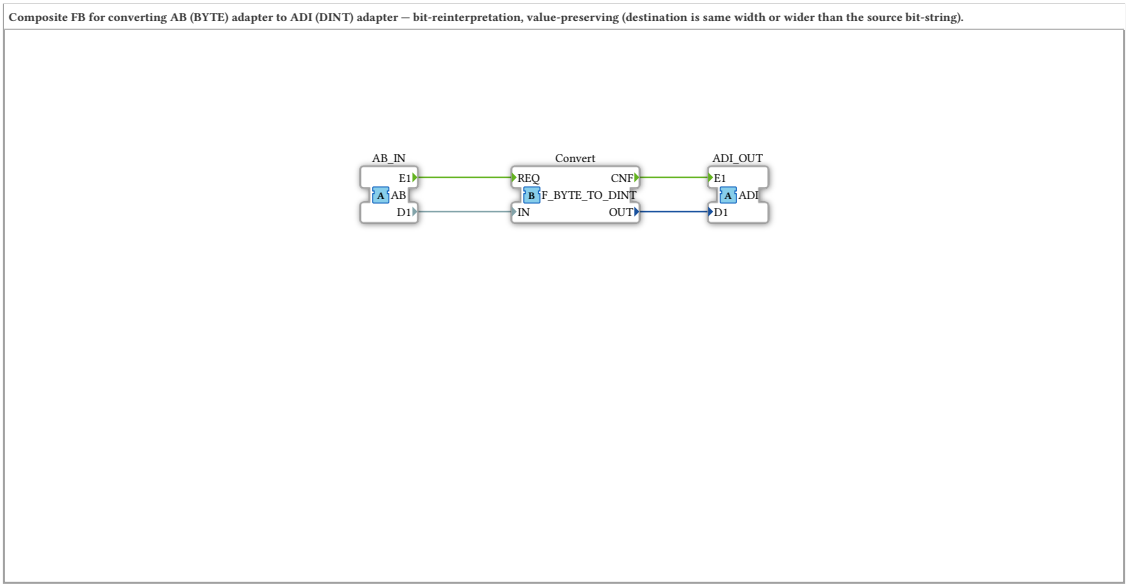
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AB_TO_ADI



BYTE Adapter Input – *AB* *ADI* – DINT Adapter Output



• – ◦ • – ◦ • – ◦ • –

Introduction

The function block **AB_TO_ADI** is a composite function block (FB) that converts an incoming BYTE value from an **AB** adapter into a DINT value and outputs it via an **ADI** adapter. It serves as a simple, unidirectional conversion interface between these two adapter types.

Interface Structure

Event Inputs

None – event control is handled exclusively via the incoming adapter (`AB_IN`).

Event Outputs

None – events are only passed on via the outgoing adapter (`ADI_OUT`).

Data Inputs

None – the data is provided via the `AB_IN` adapter.

Data Outputs

None – the converted data is output via the `ADI_OUT` adapter.

Adapters

Type	Name	Direction	Comment
Socket	`AB_IN`	Input	BYTE adapter (input)
Plug	`ADI_OUT`	Output	DINT adapter (output)

Functionality

The function block (FB) internally contains a single conversion block, `F_BYTE_TO_DINT` (from the IEC 61131 library). The incoming event `E1` from adapter `AB_IN` triggers the conversion of the data value `D1` (BYTE). The resulting DINT value is forwarded via `OUT` to the data output `D1` of adapter `ADI_OUT`. Upon completion of the conversion, the output adapter's event `E1` is triggered.

The control is purely data flow-driven: Each event at the input adapter causes a one-time conversion and forwarding.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Composite Block** – No dedicated state logic; instead, it uses the internal function block `F_BYTE_TO_DINT`.
- **Adapter-Based** – The interfaces are defined as adapters, enabling high reusability and interchangeability with other function blocks.
- **License** – The function block is released under the **Eclipse Public License 2.0**.
- **Package** – The function block is organized in the package `adapter::conversion::unidirectional`.

State Overview

The function block does not have its own state machine. Its functionality is derived solely from the internal circuitry of the conversion block. Processing is strictly event-driven and combinatorial.

Application Scenarios

- Conversion of BYTE-based sensor data (e.g., digital status values) into integer DINT values for further processing by control logic.
- Connecting components using the **AB** adapter to systems that expect the **ADI** adapter.
- Use in adapter-based communication architectures where a uniform data representation is required.

Comparison with similar modules

- **AB_TO_AI** – Converts BYTE to INT (16-bit integer).
- **AB_TO_AREAL** – Converts BYTE to REAL.
- **AB_TO_ADI** – Specifically for BYTE to DINT (32-bit integer).

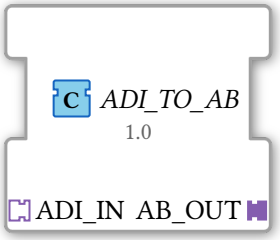
All these modules follow the same principle: Input and output are defined as unidirectional adapters that internally use a standard conversion function.

Conclusion

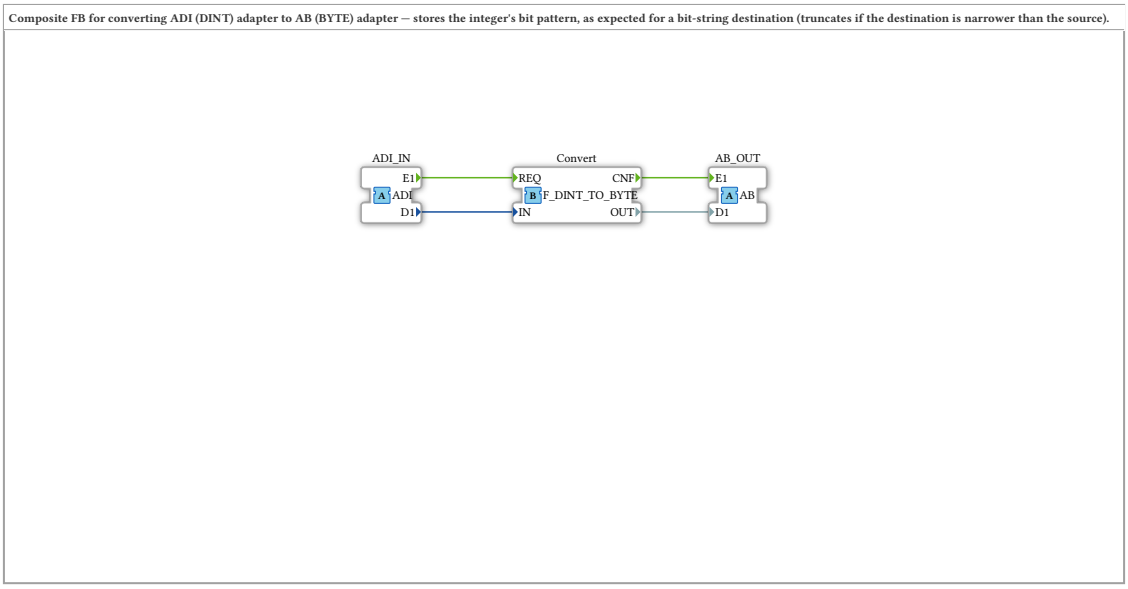
The FB `AB_TO_ADI` represents a minimalist and reliable solution for converting BYTE to DINT data. Its adapter interfaces enable seamless integration into modular control systems without requiring reimplementation of the underlying conversion logic. It is particularly suitable for applications that value a clear separation of communication protocols and data formats.

ADI_TO_AB

DINT Adapter Input – *ADI*



AB – BYTE Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block `ADI_TO_AB` serves as a composite module for converting a 32-bit integer value (DINT) into a byte value (BYTE). It encapsulates the conversion between two unidirectional adapters: the **ADI adapter** (input) and the **AB adapter** (output). The module is used to transfer data from a DINT adapter to a BYTE adapter without requiring the user to implement the actual conversion logic.

Interface Structure

Event Inputs

The module does not have direct event inputs. Event reception is handled via the **ADI_IN socket adapter**:

Adapter Port	Type	Description
ADI_IN.E1	Event	Starts the conversion at the input.

Event Outputs

Event output is also handled exclusively via the **AB_OUT plug adapter**:

Adapter Port	Type	Description
AB_OUT.E1	Event	Signals that the value has been converted and is ready at the output.

Data Inputs

The data is provided via the ADI_IN adapter:

Adapter Port	Type	Description
ADI_IN.D1	DINT	32-bit integer input value.

Data Outputs

The converted data is output via the AB_OUT adapter:

Adapter Port	Type	Description
AB_OUT.D1	BYTE	8-bit byte output value.

Adapters

Name	Direction	Adapter Type	Description

ADI_IN	Socket	`adapter::types::unidirectional::ADI`	DINT adapter as input.
AB_OUT	Plug	`adapter::types::unidirectional::AB`	BYTE adapter as output.

Functionality

Internally, the function block uses the predefined IEC 61131-3 function block `F_DINT_TO_BYTE` to perform the actual data type conversion. The event and data flows are as follows:

1. An event at **ADI_IN.E1** triggers the conversion.
2. The data value from **ADI_IN.D1** is forwarded to the input `IN` of the internal `F_DINT_TO_BYTE` function block.
3. After the conversion is complete, the internal function block generates the event **CNF**.
4. This event is output as **AB_OUT.E1**.
5. Simultaneously, the converted value is available at the output `OUT` of the internal function block and is output via **AB_OUT.D1**.

All the logic is encapsulated in a composite function block, so the user only needs to connect the adapters.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Function Block**: The function block consists of an internal network and does not have its own state machine (ECC). It performs the conversion strictly event-driven.
- **Use of IEC 61131-3 standard function blocks**: Conversion is performed using the standardized function block `F_DINT_TO_BYTE`, ensuring portability and compliance with industry standards.
- **Adapter-based input/output**: All data and events are routed via adapters – not individual ports. This simplifies integration with similar adapter interfaces in the 4diac environment.

State Overview

Since this is a composite function block without its own state machine, there are no explicit states. The function block reacts to the incoming event, converts the value, and outputs the result. After outputting the result, it is immediately ready for another conversion.

Application Scenarios

- **Data Protocol Adaptation**: When a sensor or controller delivers DINT values, but the connected device or communication channel expects BYTE values.
- **Adapter Mediation**: In a modular architecture that allows the exchange of adapters with different data types without having to rewrite the connection logic each time.
- **Interface Coupling**: Between components that use unidirectional adapters, e.g., in a pipeline structure.

Comparison with Similar Components

There are numerous conversion components for direct data type conversions (e.g., `F_DINT_TO_INT`, `F_DINT_TO_WORD`). The `ADI_TO_AB` differs in that it uses complete adapters instead of individual data and event ports. This improves encapsulation and increases reusability. A similar concept exists for other data type pairs (e.g., `ADI_TO_AC` for DINT to CHAR) that follow the same design logic.

Conclusion

The function block `ADI_TO_AB` offers a cleanly encapsulated, adapter-based conversion of DINT to BYTE values. It uses standard IEC function blocks and event-driven control, allowing for seamless integration into 4diac projects. Its adapter interfaces make it particularly suitable for modular and reusable control applications.

🌐 Related topic subpages on ms-muc-docs.de

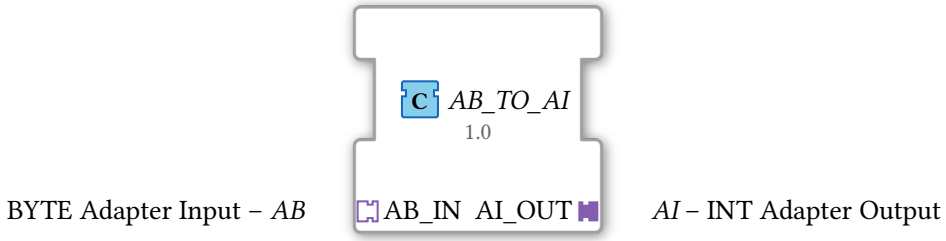
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

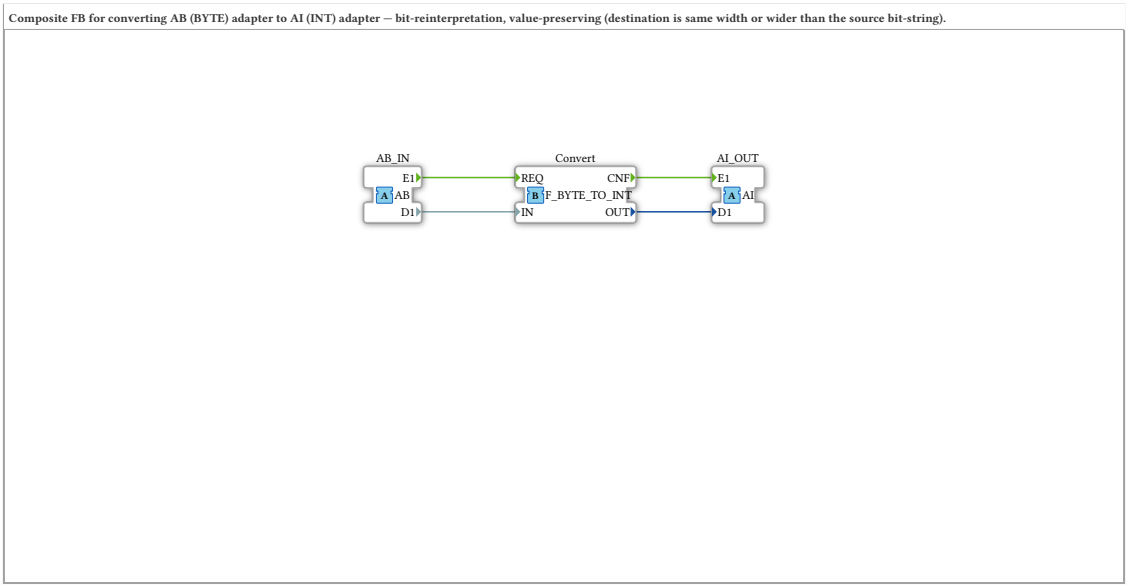
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AB_TO_AI





• – ○ • – ○ • – ○ •

Introduction

The **AB_TO_AI** is a composite function block for converting an **AB adapter (BYTE)** to an **AI adapter (INT)**. It enables the seamless integration of BYTE-based adapter interfaces into systems that expect INT values. Internally, the block uses the IEC 61131 conversion block `F_BYTE_TO_INT`.

Interface Structure

Event Inputs

The function block does not have direct event inputs. The event is provided via the connected adapter **AB_IN** (socket):

- **AB_IN.E1** – Event that triggers the conversion of an incoming BYTE value.

Event Outputs

The FB has no direct event outputs. The output event is sent via the connected adapter **AI_OUT** (Plug):

- **AI_OUT.E1** – Event indicating that the converted INT value is present at the output.

Data Inputs

The FB has no direct data inputs. The input value is provided via the adapter **AB_IN** (Socket):

- **AB_IN.D1** – BYTE value (0...255) to be converted into an INT value.

Data Outputs

The FB has no direct data outputs. The output value is provided via the adapter **AI_OUT** (Plug):

- **AI_OUT.D1** – INT value representing the converted BYTE.

Adapters

Name	Type	Direction	Description
AB_IN	`adapter::types::unidirectional::AB`	Socket (Input)	BYTE adapter that provides the value to be converted and the start event.
AI_OUT	`adapter::types::unidirectional::AI`	Plug (Output)	INT adapter that provides the converted value and the confirmation event.

Functionality

1. An incoming event on **AB_IN.E1** triggers the processing.
2. The BYTE value provided by **AB_IN.D1** is passed to the internal block `F_BYTE_TO_INT`.
3. `F_BYTE_TO_INT` performs the type conversion and provides the INT value at its output `OUT`.
4. After successful conversion, the event **AI_OUT.E1** is sent, and the INT value is available at **AI_OUT.D1**.

The conversion is performed without loss of value range: BYTE (0…255) is directly mapped to INT.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Composite Function Block:** The logic is implemented entirely through an internal network consisting of a single conversion block.
- **Reusable:** The block can be integrated into projects that require unidirectional adapter conversion.
- **License:** The block and its associated materials are licensed under the **Eclipse Public License 2.0**.
- **Package:** `adapter::conversion::unidirectional`

State Overview

The function block (FB) does not have its own internal state machine. Its behavior is determined solely by the triggering events and the embedded block `F_BYTE_TO_INT`. There are no persistent states.

Application Scenarios

- **Sensor Connection:** A sensor delivers data as BYTE via an AB adapter; downstream processing expects INT values via an AI adapter.
- **Protocol Adaptation:** In heterogeneous communication paths, BYTE-encoded information must be converted into an INT-based data model.
- **Adapter Mediation:** As an intermediary between two system components that use different adapter types.

Comparison with Similar Function Blocks

Direct conversion blocks like `F_BYTE_TO_INT` (IEC 61131) often exist, but these do not offer adapter event control. The **AB_TO_AI** encapsulates this conversion in a complete adapter-to-adapter transition with event propagation. Similar blocks for other data types (e.g., WORD to DINT) could be built according to the same pattern.

Conclusion

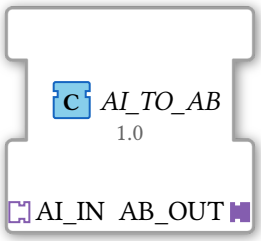
The **AB_TO_AI** is a useful and simple function block that bridges the gap between BYTE- and INT-based adapter interfaces. It facilitates the integration of components with different data type interfaces in the 4diac IDE and follows proven design patterns of the IEC 61499 standard. Thanks to its composite structure, it is easily extensible and can serve as a template for similar adapter conversions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

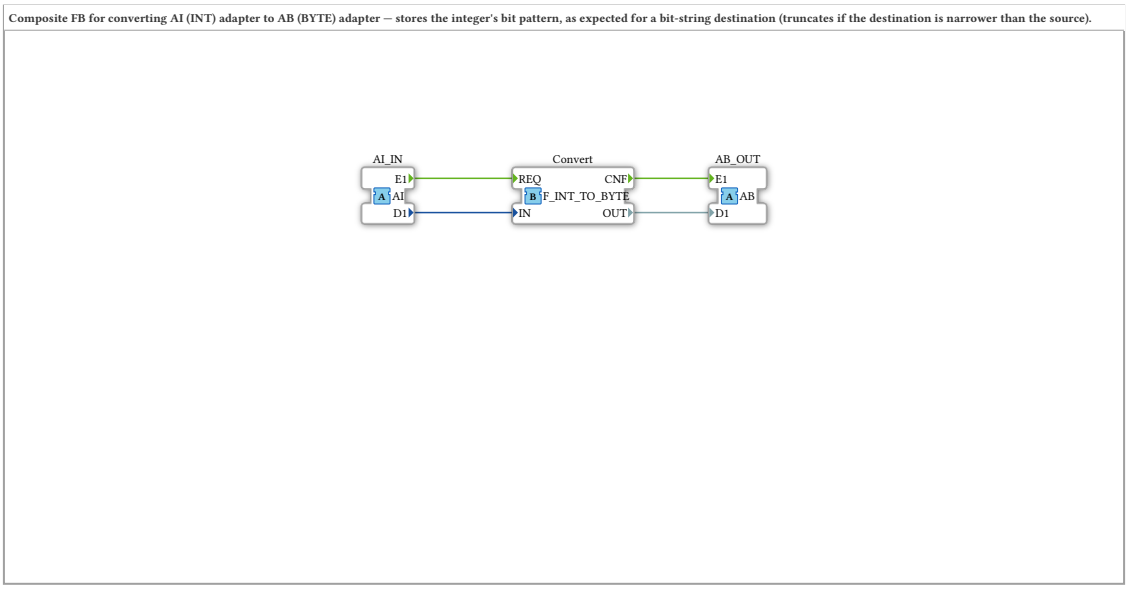
]

AI_TO_AB



INT Adapter Input – *AI*

AB – BYTE Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The composite function block `AI_TO_AB` converts an **INT** value, provided via a unidirectional **AI** adapter, into a **BYTE** value and forwards it via a unidirectional **AB** adapter. The actual conversion is performed by the internally used function block `F_INT_TO_BYTE`. This function block is part of an adapter conversion library and enables the seamless integration of components with different data types in IEC 61499 systems.

Interface Structure

Event Inputs

No standalone event inputs. Event-driven communication is handled via the AI_IN adapter (see Adapters).

Event Outputs

No standalone event outputs. Event-driven output is handled via the AB_OUT adapter (see Adapters).

Data Inputs

No standalone data inputs. The INT value to be converted is provided via the AI_IN adapter.

Data Outputs

No standalone data outputs. The converted BYTE value is output via the AB_OUT adapter.

Adapters

Direction	Name	Type	Description
Socket (Input)	`AI_IN`	`adapter::types::unidirectional::AI`	Provides the INT input value (D1) and the triggering event (E1).

Plug (Output)	AB_OUT	adapter::types::unidirectional::AB	Outputs the converted BYTE value (D1) and the associated event (E1).

Functionality

The FB AI_TO_AB operates as a pure data converter based on event control:

1. An event **E1** is received at the **AI_IN** adapter.
2. This event triggers the internal FB Convert of type F_INT_TO_BYTE via its **REQ** input.
3. Simultaneously, the INT data value from AI_IN.D1 is forwarded to the **IN** input of Convert.
4. After the conversion is complete, Convert generates an acknowledgment event (**CNF**).
5. This event is sent to **AB_OUT.E1**, and the converted BYTE value (Convert.OUT) is passed to **AB_OUT.D1**.

Thus, every incoming INT value is synchronously converted into a BYTE value via an event and made available through the output adapter.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite FB:** The function block encapsulates the logic in a network consisting of a single conversion function block. It is not itself an elementary function block but utilizes the reuse of F_INT_TO_BYTE.
- **Adapter-Based Interface:** Events and data are transmitted exclusively via unidirectional adapters, resulting in loose coupling between source and target components.
- **Type Conversion:** The actual conversion of INT -> BYTE follows the IEC 61131-3 convention (e.g., value range <0,255>; an overflow occurs if the range is exceeded). This depends on the implementation of the F_INT_TO_BYTE function block used.
- **No Internal State Machine:** The function block is purely combinatorial with event-driven execution.

State Overview

Since it is a composite function block, AI_TO_AB does not have its own state machine. The internal function block F_INT_TO_BYTE may have a simple state machine (IDLE, EXECUTING), depending on the implementation, but this is not visible within the scope of this function block. The data and event flows are deterministic and are controlled by the interaction of the connections.

Application Scenarios

- **Sensor Connection:** A sensor delivers INT values that must be processed as BYTE by a control unit (e.g., for simple I/O cards).
- **Data Format Conversion:** in heterogeneous IEC 61499 systems to connect components with different adapter types.
- **Adapter Translation:** in libraries designed for unidirectional communication, e.g., when using fieldbus protocols that expect BYTE-oriented data.

Comparison with Similar Components

- **Direct Converters (e.g., F_INT_TO_BYTE):** These operate at the data port level and require explicit event control. AI_TO_AB encapsulates this in an adapter-based interface.
- **Additional Adapter Converters:** Possible modules include AI_TO_AW (INT to WORD) or AB_TO_AI (BYTE to INT). AI_TO_AB is specifically designed for combining the two unidirectional adapters AI and AB.
- **Composite Converters vs. Elementary Converters:** Composite function blocks offer a higher level of abstraction and reusability, while elementary function blocks allow more control over the internal logic.

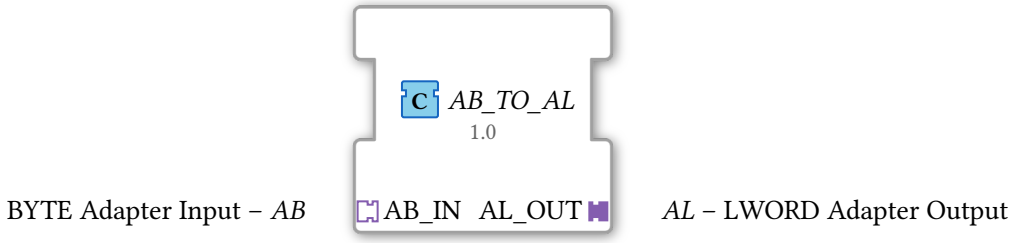
Conclusion

The composite function block AI_TO_AB provides an efficient, adapter-based solution for converting INT to BYTE values. It simplifies the integration of components with different data types into IEC 61499 applications by encapsulating the conversion logic in a standardized, reusable interface. Event-driven execution and the use of an established conversion block ensure reliable and deterministic data processing.

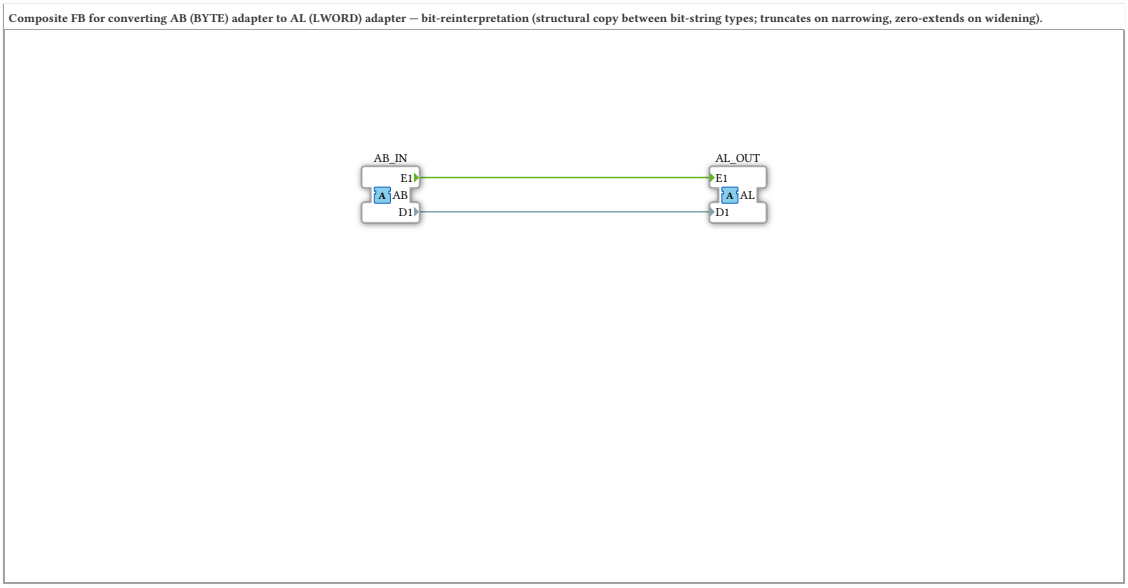
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AB_TO_AL



BYTE Adapter Input – *AB* — *AL* – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `AB_TO_AL` is a composite block for converting a unidirectional BYTE adapter (type `AB`) into a unidirectional LWORD adapter (type `AL`). It is used to convert data from an 8-bit data format to a 64-bit data format, thereby transferring the event control of the source adapter to the target adapter.

Interface Structure

Event Inputs

The block does not have direct event inputs. Incoming events are received indirectly via the socket adapter `AB_IN`. The adapter `AB` provides an internal event input (`E1`) that is connected to the event output of the target adapter.

Event Outputs

This function block does not have direct event outputs. The outgoing event flow is indirect via the plug adapter `AL_OUT`. The adapter `AL` provides an internal event output (`E1`) that is activated by the internal connection.

Data Inputs

Name	Data Type	Description
`AB_IN.D1`	BYTE	8-bit input value (via the socket adapter)

Data Outputs

Name	Data Type	Description
AL_OUT.D1	LWORD	64-bit output value (via the plug adapter)

Adapters

Name	Direction	Adapter Type	Description
AB_IN	Socket	adapter::types::unidirectional::AB	Input adapter, returns BYTE data and an event
AL_OUT	Plug	adapter::types::unidirectional::AL	Output adapter, expects LWORD data and passes on an event

Functionality

The component is implemented as a pure wiring (composite) circuit. Internally, the following connections are established:

- Event: AB_IN.E1 ⟶ AL_OUT.E1
- Data: AB_IN.D1 ⟶ AL_OUT.D1

As soon as an event arrives at the socket adapter AB_IN, the corresponding BYTE value (D1) is passed to the data output AL_OUT.D1, and simultaneously the event is forwarded to AL_OUT.E1. An implicit type conversion from BYTE (8 bits) to LWORD (64 bits) takes place – the upper 56 bits are padded with zeros. The component has no internal state logic or delay.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Pure Composite Block**: No algorithm, no state machine – functionality is derived solely from the internal wiring.
- **Unidirectional Adapters**: Data flow is only possible from AB_IN to AL_OUT; there is no reverse direction.
- **Automatic Type Conversion**: The 4diac IDE implicitly performs the conversion from BYTE to LWORD (big-endian extension with zeros).
- **No Parameters or Configuration**: The block requires no additional settings.

State Overview

As a composite block without its own state machine, there is no defined state. Its behavior is entirely determined by the incoming events and data of the socket adapter.

Application Scenarios

- **Interface Adaptation**: A sensor delivers BYTE data via an AB adapter, but the downstream system expects LWORD data via an AL adapter.
- **Data Preparation**: Simplified integration of 8-bit devices into 64-bit control architectures.
- **Protocol Conversion**: When events and data are encapsulated via adapters, AB_TO_AL can serve as a universal conversion stage.

Comparison with Similar Components

Other adapter conversion components exist, such as WORD_TO_DWORD, ARRAY_TO_STRUCT, or specific type converters. However, AB_TO_AL is distinguished by its complete encapsulation within adapters – both events and data are converted together. This simplifies the exchange of complete interfaces without the need to model separate data and event connections.

Conclusion

AB_TO_AL` is a simple yet useful composite building block for converting a BYTE adapter to an LWORD adapter. It reduces the modeling effort for type conversions in adapter-based networks and ensures that

events are passed on synchronously with the data. Due to its clearly defined interface, it is particularly well-suited for modular and reusable automation solutions.

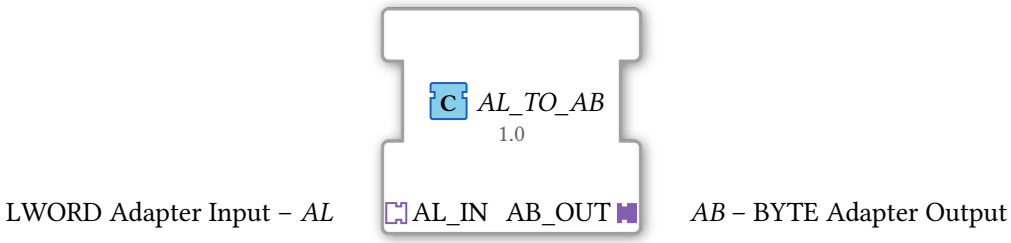
–

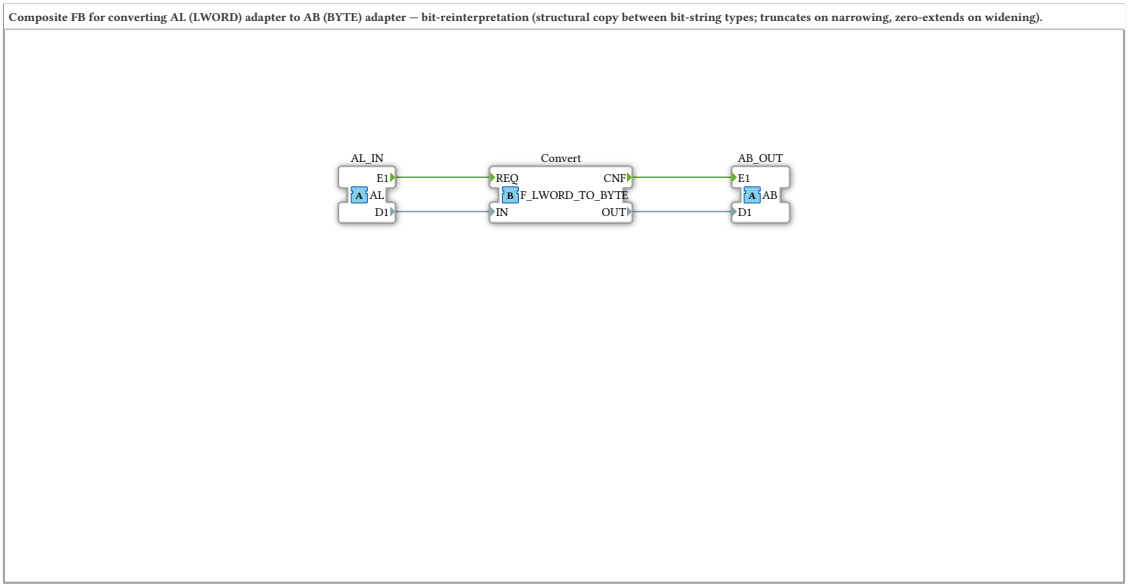
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AL_TO_AB





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_TO_AB** is a composite function block that enables unidirectional conversion from an LWORD adapter (AL) to a BYTE adapter (AB). It is used to convert data from an LWORD-based adapter to the format of a BYTE adapter, with the conversion being performed internally by the IEC 61131-3 block `F_LWORD_TO_BYTE`.

Interface Structure

Adapter (Input)

Name	Type	Description
AL_IN	`adapter::types::unidirectional::AL`	LWORD adapter socket. This adapter receives the LWORD data to be converted. By default, the adapter provides an event output `E1` and a data output `D1` (data type LWORD).

Adapter (Output)

Name	Type	Description
AB_OUT	`adapter::types::unidirectional::AB`	BYTE adapter plug. This adapter outputs the converted BYTE data. The adapter has an event input `E1` and a data input `D1` (data type BYTE).

Note: The function block (FB) has no separate event or data inputs/outputs; all communication takes place via the two adapter interfaces.

Functionality

1. An event (at the adapter's event output `E1`) is received via socket `AL_IN`, which simultaneously provides an LWORD value via data output `D1`.
2. This event internally triggers the function block `F_LWORD_TO_BYTE` (type: `iec61131::conversion::F_LWORD_TO_BYTE`), which converts the LWORD value into a BYTE value.
3. After successful conversion, `F_LWORD_TO_BYTE` signals completion via its event output `CNF`, which forwards the event to plug `AB_OUT`.
4. Simultaneously, the converted BYTE value is transferred via the converter's data output `OUT` to the data input `D1` of `AB_OUT`.
5. The connected receiver on plug `AB_OUT` thus receives the event and the converted data value.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Integration of IEC 61131-3**: The conversion uses the standardized function block `F_LWORD_TO_BYTE` from the IEC 61131-3 library. This ensures that the conversion logic is standards-compliant and portable.
- **Composite Structure**: The function block consists exclusively of an internal network, has no independent state (ECC), and therefore operates in an event-driven manner without delay.
- **Unidirectional Adapters**: The unidirectional adapters `AL` and `AB` are used, which define a clear direction for data and event flows.
- **Package Structure**: The function block is organized in the package `adapter::conversion::unidirectional`, which facilitates reuse in modular projects.

State Overview

The function block does not have its own state machine. The entire process control is managed via the event chain of the internal network. Therefore, status monitoring is not required.

Application Scenarios

- **Protocol Adaptation**: When an adapter in a distributed system that delivers LWORD data needs to be replaced by an adapter that only processes BYTE data.
- **System Integration**: Coupling components that use different adapter formats (e.g., AL and AB adapters) without modifying the original components.
- **Data Reduction**: Targeted conversion of 64-bit LWORD to 8-bit BYTE when the higher bytes are not needed or a narrower interface is required.

Comparison with Similar Components

Component	Function	Difference
`LWORD_TO_BYTE`	Pure data type conversion without adapters	No adapter interface; direct data and event ports
AB_TO_AL`	Reverse conversion (BYTE → LWORD)	Opposite direction; also composite with adapters
WORD_TO_BYTE`	Conversion from WORD (16-bit) to BYTE	Different data type; similar structure, but different adapter types

The **AL_TO_AB** is specifically designed for use with the unidirectional adapters `AL` and `AB` and offers clean, reusable encapsulation.

Conclusion

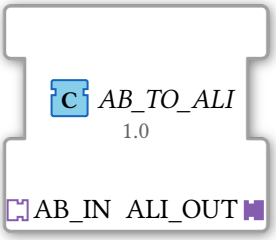
The `AL_TO_AB` function block provides a simple yet effective solution for converting an LWORD adapter signal into a BYTE adapter signal. By using a standardized IEC 61131-3 converter and a consistent adapter interface, it is easily integrated into existing systems and promotes a modular, maintainable design.

Numeric Index (unidirectional)

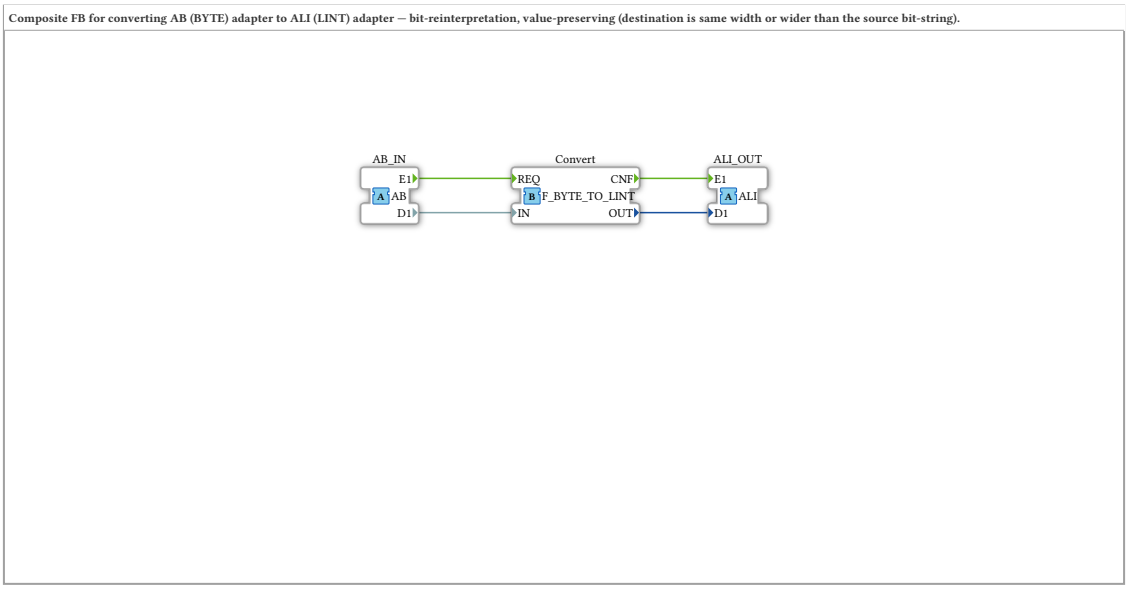
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AB_TO_ALI

BYTE Adapter Input – *AB*



ALI – LINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `AB_TO_ALI` is a composite function block that performs conversion from a BYTE adapter (`AB`) to a LINT adapter (`ALI`). It enables seamless integration between interfaces based on different data types and provides data transfer via standardized adapters.

Interface Structure

Event Inputs

The function block has no direct event inputs. Events are received via the socket adapter `AB_IN`.

Event Outputs

The function block has no direct event outputs. Events are output via the plug adapter `ALI_OUT`.

Data Inputs

No direct data inputs for the function block. Data is received via the socket adapter `AB_IN`.

Data Outputs

No direct data outputs for the function block. Data is output via the plug adapter `ALI_OUT`.

Adapters

- **Socket `AB_IN`** (Type: `adapter::types::unidirectional::AB`)

Receives a BYTE signal (via data port `D1`) and an event (via event port `E1`).

- **Plug `ALI_OUT`** (Type: `adapter::types::unidirectional::ALI`)

Sends a LINT signal (via data port `D1`) and an event (via event port `E1`).

Functionality

The function block operates as a pure data conversion loop:

1. An incoming event at socket `AB_IN` (port `E1`) triggers the internal conversion module `F_BYTE_TO_LINT`.
2. The BYTE value (`D1`), also present at the socket, is passed to the input `IN` of the converter.
3. The converter transforms the BYTE value into a LINT value and makes it available at its output `OUT`.
4. The converter's acknowledgment event (`CNF`) is forwarded to the plug `ALI_OUT` (port `E1`), and the converted LINT value is output on the plug's data port `D1`.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- This is a **composite function block** that internally uses the standard function block `iec61131::conversion::F_BYTE_TO_LINT`.
- The conversion is **event-driven** – any change at the input triggers processing.
- The function block is designed for unidirectional adapters and does not support reverse conversion.
- The adapter type information (`AB` and `ALI`) is defined as unidirectional interfaces and each contains one event port and one data port.

State Overview

The function block does not have its own state machine. The conversion is **stateless** – with each incoming event, the current value is immediately converted and output. There is no delay, buffering, or internal state.

Application Scenarios

- **Integration of fieldbus systems** or sensors that deliver BYTE data into a control environment that expects LINT values (e.g., for further processing in counters or long integer operations).
- **Adapter conversion** in modular function block networks where the interfaces use different data types (e.g., in agricultural or automation technology).
- **Bridging** between older BYTE-based modules and modern LINT-based components.

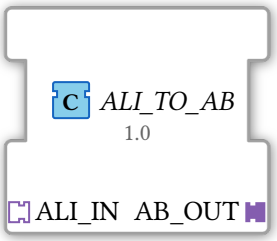
Comparison with Similar Function Blocks

- `AB_TO_ALI` is specifically for converting **BYTE** to **LINT**. Analogous function blocks exist for other data type combinations (e.g., `WORD_TO_DINT`, `BYTE_TO_INT`).
- Unlike generic converters or scaling blocks, this function block is optimized for a fixed type conversion and requires no parameterization.
- The use of adapters instead of direct input/output ports enables greater reusability and a clean separation of interfaces.

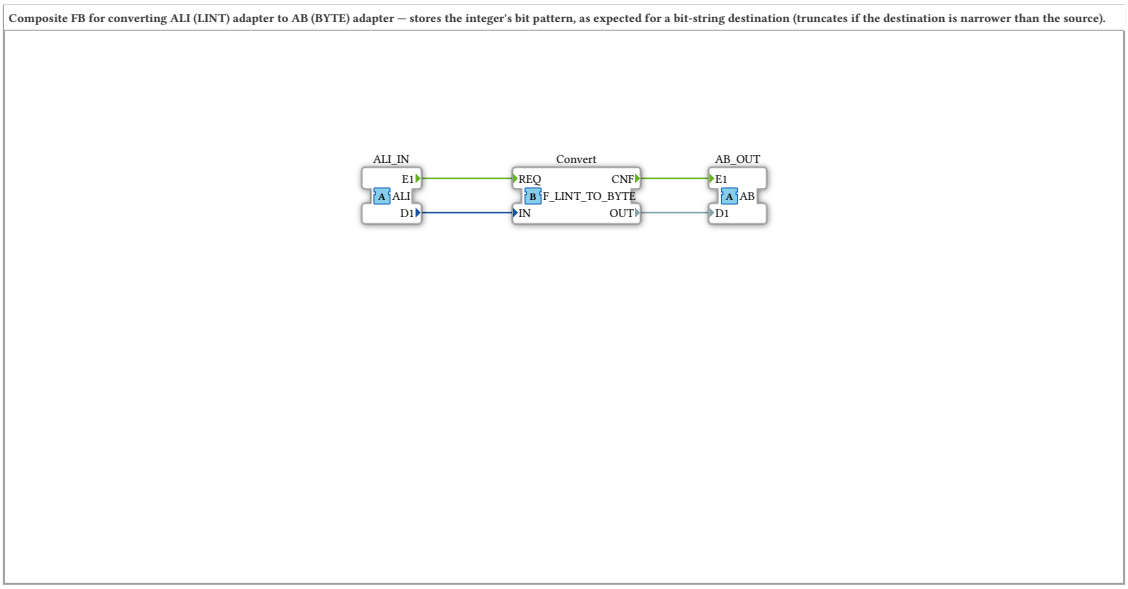
Conclusion

The `AB_TO_ALI` function block offers a simple and reliable way to convert BYTE data to LINT data via an adapter. By using standard conversion modules and a clear event-driven architecture, it is ideally suited for modular automation systems that require consistent data conversion between different adapter types.

ALI_TO_AB



LINT Adapter Input – *ALI* *AB* – BYTE Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **ALI_TO_AB** function block is a composite function block (FB) according to IEC 61499-2, used to convert a LINT value (64-bit integer) into a BYTE value (8-bit). The conversion is performed via two unidirectional adapter interfaces: The ALI_receiver (socket) receives a LINT value, and the AB_sender (plug) transmits the converted BYTE value. Internally, the conversion block `F_LINT_TO_BYTE` from the IEC 61131 library is used.

Interface Structure

Event Inputs

The FB has no direct event inputs. Events are received exclusively via the **ALI_IN** adapter (socket).

Event	Description
E1 (via ALI_IN)	Signal to trigger the conversion.

Event Outputs

The FB has no direct event outputs. Events are sent exclusively via the **AB_OUT** adapter (plug).

Event	Description
E1 (via AB_OUT)	Acknowledgement after successful conversion.

Data Inputs

The FB has no direct data inputs. Data is provided exclusively via the **ALI_IN** adapter (socket).

Data	Type	Description

D1 (via ALI_IN)	LINT	64-bit integer to be converted.

Data Outputs

The FB has no direct data outputs. Data is output exclusively via the **AB_OUT** adapter (plug).

Data	Type	Description
D1 (via AB_OUT)	BYTE	8-bit converted value.

Adapters

Adapter	Direction	Type	Description
ALI_IN	Socket (Input)	`adapter::types::unidirectional::ALI`	Receives the LINT value and an associated event.
AB_OUT	Plug (Output)	`adapter::types::unidirectional::AB`	Outputs the converted BYTE value and an acknowledgment event.

Functionality

1. The function block waits for an event at the **ALI_IN** adapter (E1).
2. As soon as the event arrives, the data value D1 (LINT) is passed to the internal conversion block `F_LINT_TO_BYTE` and converted into a BYTE value.
3. After the conversion is complete, the BYTE value is output via the **AB_OUT** adapter (D1) and the acknowledgment event (E1) is sent.

The conversion from LINT (64 bits) to BYTE (8 bits) is performed by truncating the 8 least significant bits. Higher-order bits are lost in the process.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Type Conversion with Loss of Value:** Since LINT is 64 bits wide, but BYTE is only 8 bits, only the lower 8 bits of the LINT value are retained. A value range of 0 to 255 is correctly represented; larger numbers are reduced modulo 256.
- **Pure Adapter Interface:** The component has no direct inputs/outputs, but communicates exclusively via two unidirectional adapters. This enables loose coupling in adapter-based architectures.
- **Standard Compliance:** The function block (FB) is based on the IEC 61131 block `F_LINT_TO_BYTE` and can therefore be used in environments that support this library.

State Overview

As a composite block, **ALI_TO_AB** does not have its own internal state machine. The states of the included `F_LINT_TO_BYTE` (e.g., IDLE, BUSY) are not visible to the user. The FB therefore behaves like a combinational or event-driven function without an explicit state mask.

Application Scenarios

- **Adapter-Based Data Conversion:** In systems connected via adapters with ALI and AB interfaces, ALI_TO_AB can be used as a bridge between a LINT-supplying module and a BYTE-receiving module.
- **Data Width Reduction:** This function block is suitable when a 64-bit value (e.g., a counter reading) needs to be mapped to an 8-bit signal (e.g., a simple status byte).
- **Protocol Conversion:** In heterogeneous control systems using different adapter types, ALI_TO_AB enables simple type conversion.

Comparison with Similar Function Blocks

- **ALI_TO_Ax (LINT to ANY):** A more general converter that converts LINT to any other data type. ALI_TO_AB is specialized for BYTE and is therefore simpler and more direct.
- **F_LINT_TO_BYTE:** The underlying IEC 61131 function block. ALI_TO_AB encapsulates this in an adapter-based interface, allowing its use in environments that only support adapter communication.

- **Direct Adapter Connection:** Without conversion, an ALI adapter could not be directly connected to an AB adapter. ALI_TO_AB closes this gap.

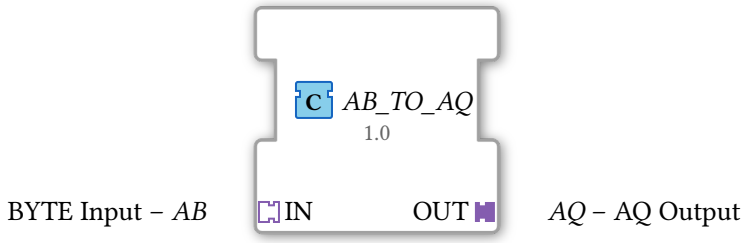
Conclusion

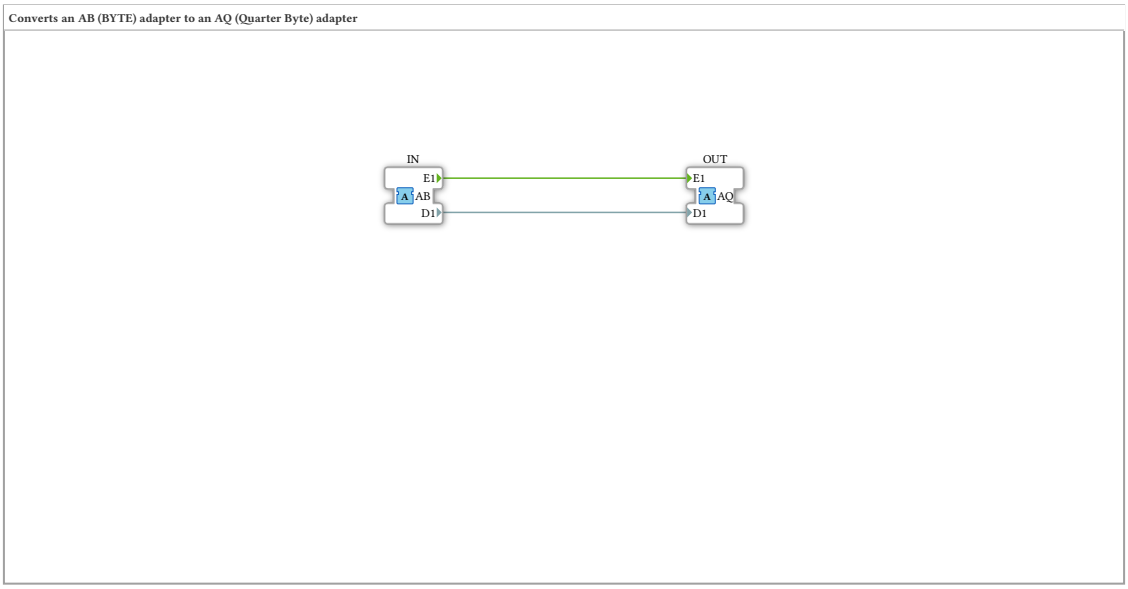
ALI_TO_AB is a simple yet useful conversion block for adapter-based communication. It enables the conversion of a LINT value to a BYTE value using a standardized IEC 61131 block. Its complete encapsulation within two unidirectional adapters allows for seamless integration into adapter networks without requiring the user to implement the internal type conversion. The loss of value due to the reduced data width should be considered when using this feature.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AB_TO_AQ





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_TO_AQ** converts a unidirectional adapter of type **AB (BYTE)** to an adapter of type **AQ (Quarter Byte)**. It directly passes an event and a data value from the input interface to the output interface, implicitly reducing the data width from 8 bits to 2 bits.

Interface Structure

Event Inputs

- **IN.E1** (via adapter IN): Event input of the BYTE adapter.

Event Outputs

- **OUT.E1** (via adapter OUT): Event output of the Quarter Byte adapter.

Data Inputs

- **IN.D1** (via adapter IN): Data input of the BYTE adapter (1 byte).

Data Outputs

- **OUT.D1** (via adapter OUT): Data output of the quarter-byte adapter (1 quarter byte).

Adapters

Adapter	Direction	Type	Description
IN	Socket	`adapter::types::unidirectional::AB`	BYTE input adapter (8 bits)
OUT	Plug	`adapter::types::unidirectional::AQ`	Quarter-byte output adapter (2 bits)

Functionality

This function block forwards an incoming event (E1) and the associated data (D1) directly from the IN adapter to the OUT adapter. No internal processing or state changes occur; the data is simply passed through. The conversion from an 8-bit to a 2-bit format is determined by the different adapter types – typically, the least significant bits (LSB) of the BYTE value are converted to the quarter byte, unless otherwise specified.

Technical Features

- Pure pass-through block without its own state logic (no ECC).
- Event and data paths are directly connected: `IN.E1 → OUT.E1` and `IN.D1 → OUT.D1`.
- Data reduction from 8 bits to 2 bits is achieved solely through type conversion; explicit conversion (e.g., truncation or rounding) is not implemented in the function block itself.
- The block is designed as a unidirectional adapter converter and does not support bidirectional communication.

State Overview

There is no state machine. The function block (FB) operates purely combinatorially: Every event at the input is immediately forwarded as an event at the output, accompanied by the current data value.

Application Scenarios

- Connecting a byte-oriented interface to logic that processes only two-bit wide data (e.g., status bits or switching signals).
- Reducing the data width in an adapter cascade when the original byte information can be compressed to the lower two bits.
- Test and simulation environments where different adapter formats need to be interconnected.

Comparison with Similar Function Blocks

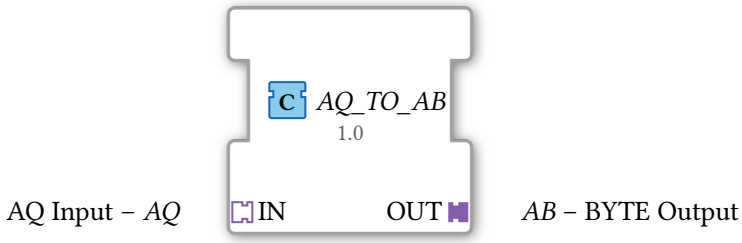
- **AB_TO_AQ** is similar to other unidirectional adapter converters such as `AB_TO_AB` (identical types) or `AQ_TO_AB` (extension to one byte). The difference lies in the data width and the type of implicit conversion.

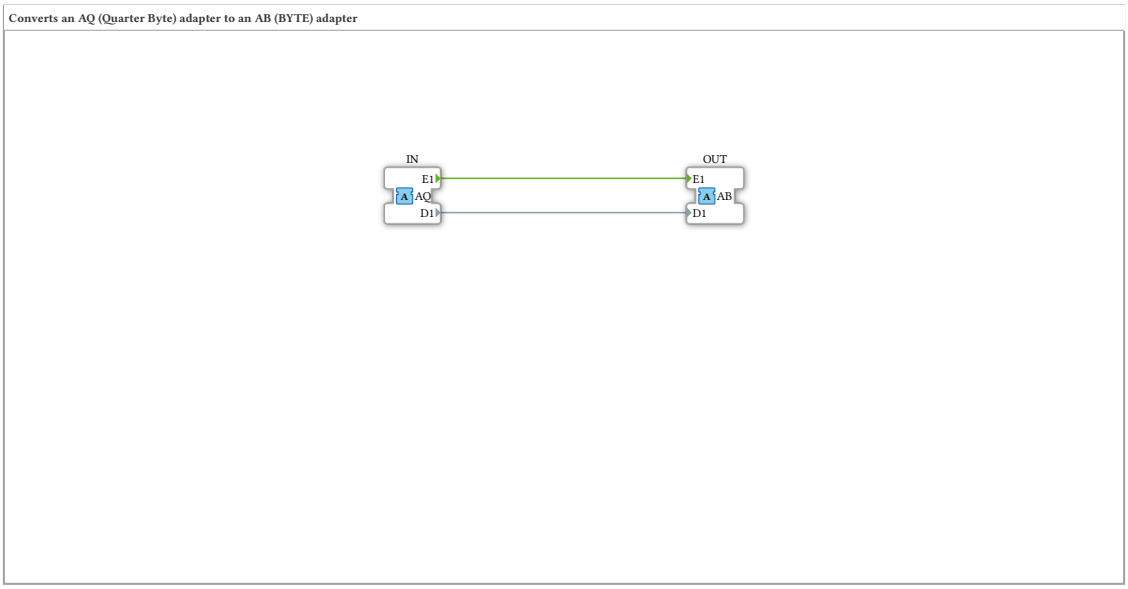
AB_TO_AQ Unlike converters with explicit algorithms (e.g., scaling or bit masking), this function block offers no configuration options – the conversion is fixed.

Conclusion

The **AB_TO_AQ** function block is a simple yet useful adapter converter that establishes a direct electrical and logical connection between two different unidirectional adapter types. It is ideally suited for situations where byte information needs to be reduced to the two most important bits and passed on without generating additional processing overhead.

AQ_TO_AB





• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AQ_TO_AB` converts an AQ adapter (Quarter Byte) to an AB adapter (BYTE). It enables the seamless connection of components using different adapter interfaces by passing the signals through unchanged. The block is implemented as a pure “pass-through” function block that does not manipulate any data.

Interface Structure

Event Inputs

The function block has no explicit top-level event inputs. Events are transmitted via the adapter interfaces.

Event Outputs

The function block has no explicit top-level event outputs. Events are transmitted via the adapter interfaces.

Data Inputs

The FB has no explicit top-level data inputs. Data transmission occurs via the adapter interfaces.

Data Outputs

The FB has no explicit top-level data outputs. Data transmission occurs via the adapter interfaces.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AQ`	IN	Input (Socket)	AQ adapter as input interface for quarter-byte data

`adapter::types::unidirectional::AB`	OUT	Output (Plug)	AB adapter as output interface for byte data

Functionality

The function block internally connects the event and data channels of the input adapter `IN` directly to the corresponding channels of the output adapter `OUT`. Specifically, the following connections are established:

- Event connection: `IN.E1` ⟶ `OUT.E1`
- Data connection: `IN.D1` ⟶ `OUT.D1`

Thus, all incoming events and data from the AQ adapter are passed on to the AB adapter unchanged. The function block itself does not process or convert the user data – it functions solely as an interface adapter.

Technical Features

- **Unidirectional Data Flow:** The function block supports only one direction (AQ ⟶ AB).
- **No State Change:** Since no logic or memory is present, this is a purely combinational function block.
- **Simple Loop-Through:** The implementation is limited to two direct connections in the FBNetwork.
- **Adapter Conversion:** The function block allows the integration of AQ-compatible components into a system that expects AB adapters without requiring any modifications to the connected function blocks.

State Overview

The function block does not have a state machine. Its behavior is deterministic: Every incoming event immediately triggers the corresponding output event, and every incoming data value is forwarded to the output without delay.

Application Scenarios

- **System Integration:** Connecting a sensor module with an AQ interface to a controller that uses AB adapters.
- **Protocol Implementation:** Use in libraries to make different adapter types compatible.
- **Test Environments:** Use as a simple adapter in test setups to adapt interfaces.

Comparison with Similar Function Blocks

- **AB_TO_AQ:** Performs the reverse conversion (AB ⟶ AQ).
- **AQ_TO_BYTE_EXT:** A hypothetical function block that additionally performs data conversions (e.g., scaling, formatting), while `AQ_TO_AB` only changes the interface.
- **Generic Adapter Wrappers:** Other platforms often offer generic conversion blocks, but these require more complex configuration. `AQ_TO_AB` is specialized and therefore very lightweight.

Conclusion

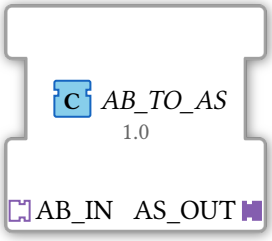
The function block `AQ_TO_AB` offers a lean and efficient solution for adapter conversion from AQ to AB. Its simple pass-through structure makes it reliable, resource-efficient, and ideal for situations where only the interface type needs to be changed without affecting the signals. It is particularly suitable for modular systems with different adapter standards.

Numeric Index (unidirectional)

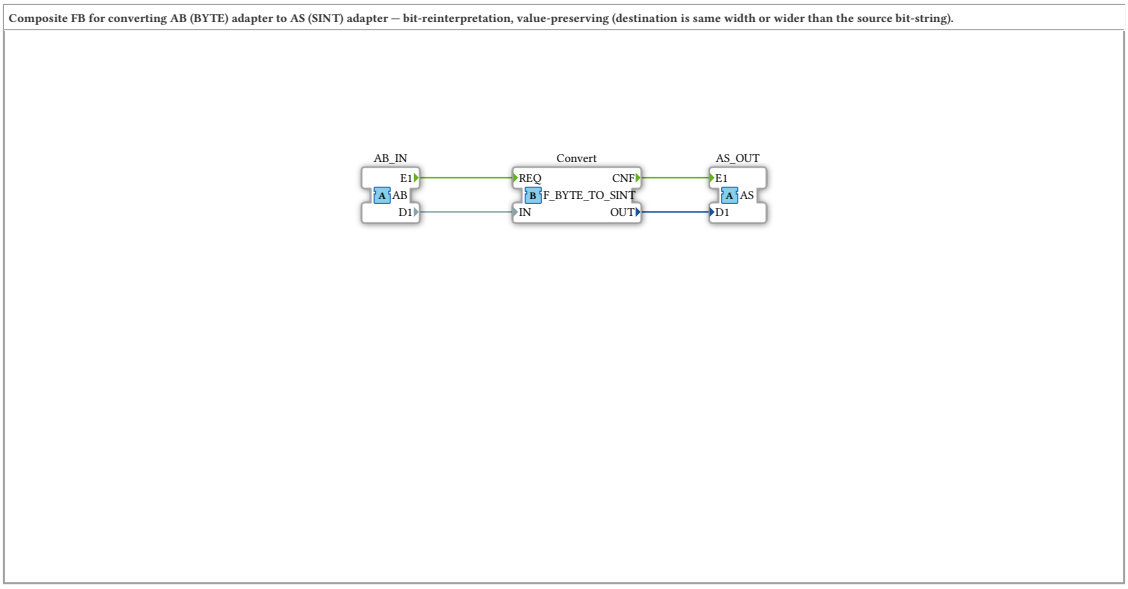
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AB_TO_AS

BYTE Adapter Input – *AB*



AS – SINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AB_TO_AS** is a composite function block (FB) that converts a unidirectional AB adapter (BYTE) into a unidirectional AS adapter (SINT). It serves as a conversion bridge between two different data types in an IEC 61499-based control application.

Interface Structure

The FB does not have its own top-level event or data ports. All communication takes place via two adapter interfaces:

Event Inputs

- **E1** (via **AB_IN**, socket, type `adapter::types::unidirectional::AB`)

This event is received by the connected source adapter and triggers the conversion of a BYTE value to a SINT value.

Event Outputs

- **E1** (via **AS_OUT**, Plug, Type `adapter::types::unidirectional::AS`)

This event is sent to the target adapter after successful conversion to signal the new SINT value.

Data Inputs

- **D1** (BYTE, via **AB_IN** adapter)

Contains the BYTE value to be converted. It is sampled at the input via the **E1** event.

Data Outputs

- **D1** (SINT, via **AS_OUT** adapter)

Provides the converted SINT value. It becomes valid at the output after the conversion is complete with the **E1** event.

Adapters

Adapter	Direction	Type	Description
AB_IN	Socket (Input)	`adapter::types::unidirectional::AB`	Receives BYTE data and a trigger event from the source system.

AS_OUT	Plug (Output)	`adapter::types::unidirectional::AS`	Provides the converted SINT data with an acknowledgment event to the target system.

Functionality

The **AB_TO_AS** function block internally contains a single function module of type `iec61131::conversion::F_BYTE_TO_SINT`. This module performs the actual data type conversion from BYTE to SINT.

1. An incoming event **AB_IN.E1** triggers the **F_BYTE_TO_SINT** function block via its **REQ** input.
2. Simultaneously, the data value from **AB_IN.D1** (BYTE) is placed on the **IN** input of the converter.
3. After processing, the converter signals its **CNF** event, which is forwarded to the adapter output **AS_OUT.E1**.
4. The converted SINT value is transferred from the **OUT** output of the converter to **AS_OUT.D1**.

This implements an asynchronous, event-driven translation of a BYTE value into a SINT value.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Composite FB**: The logic is completely encapsulated within the internal FB network structure. This facilitates maintenance and reuse.
- **License**: This module is provided under the **Eclipse Public License 2.0** (SPDX license identifier: EPL-2.0).
- **Version**: 1.0, released 2026.
- **Dependencies**: The internal converter module `F_BYTE_TO_SINT` from the library `iec61131::conversion` is required.

State Overview

The **AB_TO_AS** module does not have its own state machine. Processing is purely data-flow and event-driven via the embedded **F_BYTE_TO_SINT** module. Therefore, there are no explicit states, only the implicit processes: *Waiting for input event*, *Conversion active*, and *Output event sent*.

Application Scenarios

- **System Integration**: Connecting two subsystems that use different adapter data types (BYTE vs. SINT).
- **Data Format Conversion**: When an existing control algorithm expects SINT values, but the source only provides BYTE adapters.
- **Protocol Adaptation**: In heterogeneous automation environments where function blocks from different manufacturers or libraries need to be combined.

Comparison with Similar Function Blocks

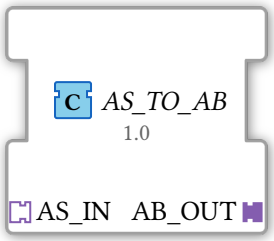
- **F_BYTE_TO_SINT** (Direct Function Block): Offers the same conversion logic, but without adapter encapsulation. Requires manual wiring of event and data lines.
- **AB_TO_INT** or **AB_TO_USINT**: Similar adapter converters for other target data types. **AB_TO_AS** is specifically designed for SINT.
- **Custom Implementation**: The composite design allows for easy extension, e.g., by adding plausibility checks or scaling, without changing the basic functionality.

Company Implementation

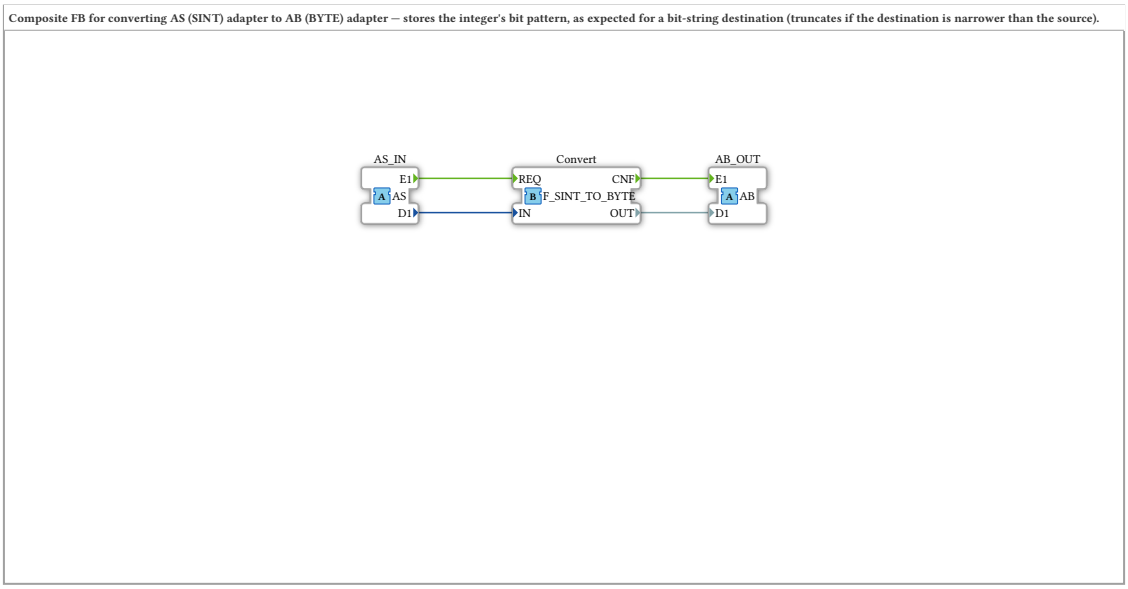
Conclusion

The **AB_TO_AS** is a specialized yet simple composite function block that offers a clean and reusable solution for adapter-based data type conversion from BYTE to SINT. By utilizing a standardized converter block and clear event control, it integrates seamlessly into IEC 61499 networks. Its licensing under EPL 2.0 promotes open use and further development.

AS_TO_AB



SINT Adapter Input – *AS* *AB* – BYTE Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AS_TO_AB** is a composite function block that converts an AS adapter (SINT) into an AB adapter (BYTE). It serves as a bridge between different adapter types and enables data type conversion within an adapter-based communication chain. The block internally uses the IEC 61131 function **F_SINT_TO_BYTE** and is implemented as a pure data converter without state logic.

Interface Structure

Since the function block does not have direct event or data ports, inputs and outputs are provided via the following adapters:

Event Inputs

Port	Description
`AS_IN.E1`	Starts the conversion of a SINT value.

Event Outputs

Port	Description
`AB_OUT.E1`	Signals the completion of the conversion.

Data Inputs

Port	Data Type	Description
`AS_IN.D1`	SINT	The value to be converted (Signed Short Integer).

Data Outputs

Port	Data Type	Description
AB_OUT.D1	BYTE	The converted value (byte).

Adapter

Label	Type	Direction	Comment
AS_IN	adapter::types::unidirectional::AS	Socket (Input)	SINT Adapter Input
AB_OUT	adapter::types::unidirectional::AB	Plug (Output)	BYTE Adapter Output

Functionality

1. An incoming event on AS_IN.E1 triggers processing.
2. The current value of AS_IN.D1 (SINT) is read.
3. The internal instance F_SINT_TO_BYTE performs the conversion to a BYTE value.
4. After successful conversion, an event is output on AB_OUT.E1, and the calculated BYTE value is made available on AB_OUT.D1.

The conversion strictly adheres to IEC 61131 semantics: The SINT value is mapped bitwise to the BYTE data type, without any sign or range considerations (since SINT comprises 8 bits, identical to BYTE).

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite FB**: The function block encapsulates the conversion logic and provides it as an adapter converter.
- **Unidirectional Adapters**: Both the input and output adapters are unidirectional – data flows in only one direction.
- **No Internal State**: The FB behaves like a pure function; each activation produces the correct result regardless of the previous state.
- **Dependency**: The function block imports the IEC 61131 function F_SINT_TO_BYTE, which must be available in the target environment.

State Overview

The FB has **no state diagram** – it is event-driven and performs a one-time conversion immediately upon each activation.

The internal element F_SINT_TO_BYTE is also stateless.

Application Scenarios

- **Adapter Coupling**: If a sensor or actuator provides an AS adapter (SINT), but the downstream system expects an AB adapter (BYTE), this function block (FB) is used as an intermediary.
- **Protocol Conversion**: In PLC or automation systems, different data types often need to be converted between adapters – specifically SINT ↔ BYTE.
- **Modularization**: The FB allows conversion logic to be encapsulated in reusable adapter blocks, instead of having to implement it anew in each application FB.

Comparison with Similar Function Blocks

- **AS_TO_WORD / AS_TO_INT**: Convert SINT to WORD or INT; AS_TO_AB specializes in BYTE.
- **Direct use of F_SINT_TO_BYTE**: The composite function block offers the advantage that the conversion is performed via adapter ports, eliminating the need for manual wiring of event and data flows.
- **Other adapter converters**: If similar adapter converters exist in the library (e.g., AB_TO_AS), this function block complements the unidirectional conversion in the opposite direction.

Conclusion

The **AS_TO_AB** function block is a simple yet useful converter for adapter-based architectures.

It allows the seamless integration of SINT and BYTE-communicating components without requiring the user to program the underlying conversion themselves.

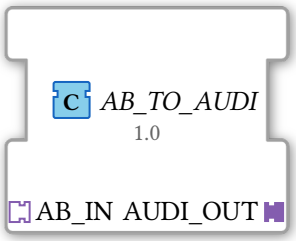
Thanks to its composite structure, the logic remains clear and maintainable – ideal for use in industrial control systems according to IEC 61499.

Numeric Index (unidirectional)

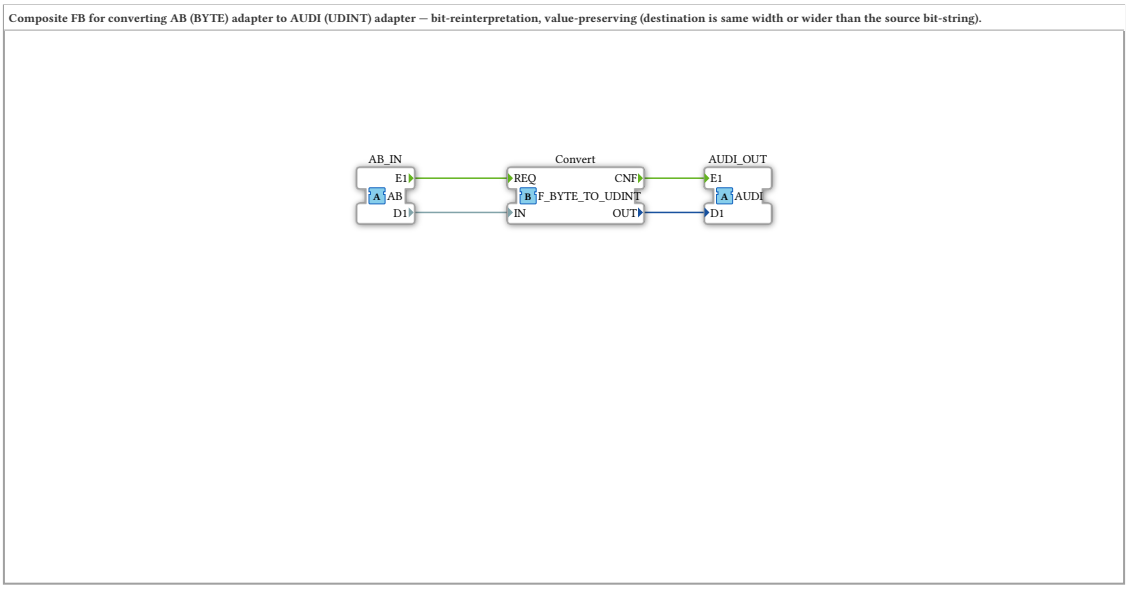
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AB_TO_AUDI

BYTE Adapter Input – *AB*



AUDI – UDINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AB_TO_AUDI** is a composite function block (FB) that converts an AB adapter (data type BYTE) into an AUDI adapter (data type UDINT). It serves as an interface between different data formats in an IEC 61499-based automation environment.

Interface Structure

The function block has no direct event or data input/output at the top level. Communication occurs exclusively via two adapters:

Event Inputs

- None (events are provided via the socket adapter `AB_IN`)

Event Outputs

- None (events are output via the plug adapter `AUDI_OUT`)

Data Inputs

- None (data is provided via the socket adapter `AB_IN`)

Data Outputs

- None (data is output via the plug adapter `AUDI_OUT`)

Adapter

- **`AB_IN`** (Socket) – Type: `adapter::types::unidirectional::AB`

Accepts a BYTE signal. The adapter provides an event `E1` and a data value `D1` (BYTE).

- **`AUDI_OUT`** (Plug) – Type: `adapter::types::unidirectional::AUDI`

Outputs the converted UDINT signal. The adapter expects an event `E1` and a data value `D1` (UDINT).

Functionality

The module internally uses a predefined conversion module `F_BYTE_TO_UDINT` from the IEC 61131 library. The process:

1. An event `E1` is received via socket `AB_IN`, triggering the start of the conversion.
2. Simultaneously, the corresponding BYTE data value from `AB_IN.D1` is forwarded to the converter `F_BYTE_TO_UDINT.IN`.
3. The converter calculates the UDINT value and outputs it at its output `OUT`.
4. The converter's acknowledgment event `CNF` is sent to the plug `AUDI_OUT.E1`.
5. The converted UDINT value is passed to the connected component via `AUDI_OUT.D1`.

This enables unidirectional BYTE-to-UDINT conversion with event-driven control.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Unidirectional Adapters**: Both the input and output adapters are unidirectional – data flows in only one direction (BYTE → UDINT).
- **Synchronous Operation**: The conversion is strictly event-driven: Each incoming event `E1` triggers exactly one conversion and one outgoing event.
- **Standard Conversion**: The internal function block `F_BYTE_TO_UDINT` is an established IEC 61131 function block, ensuring that the conversion is standards-compliant and portable.

State Overview

Since this is a purely composite component without its own state machine, the state logic results from the interaction of the internal connections:

- **Ready**: After initialization, wait for event `AB_IN.E1`.
- **Conversion Active**: After receiving `AB_IN.E1`, while the internal converter is working.
- **Output Ready**: After confirmation by `Convert.CNF`, until event `AUDI_OUT.E1` and the associated data value are transmitted.

Application Scenarios

- **Protocol Adaptation**: When a sensor or actuator with a BYTE interface is to be connected to a control unit that expects UDINT (e.g., for counts, frequencies, or time values).
- **Data Preparation in Automation Networks**: Used in PLC or distributed systems to harmonize different data widths.
- **Adapter Bridge**: As an intermediary between predefined adapter types in an IEC 61499 application.

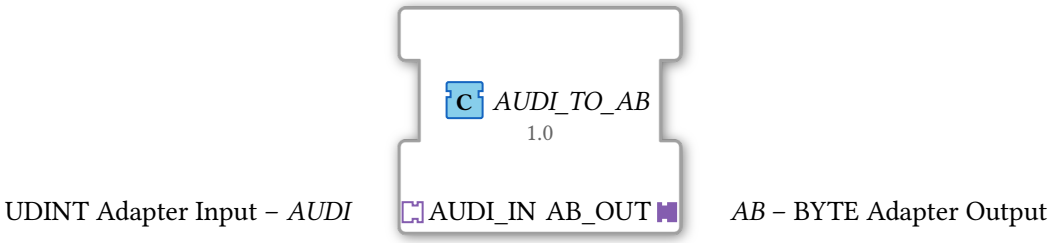
Comparison with Similar Function Blocks

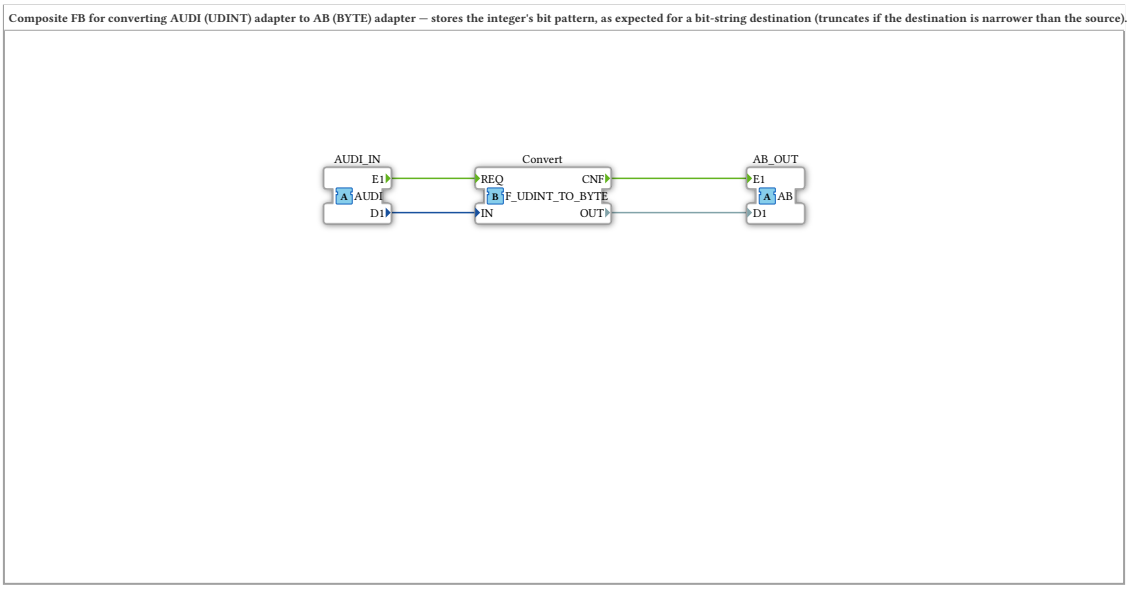
- **AB_TO_DINT**: Converts BYTE to DINT (signed) – here, the conversion is to UDINT (unsigned).
- **Direct Converters** such as `F_BYTE_TO_UDINT`: These are simple function blocks without an adapter interface. `AB_TO_AUDI` encapsulates this function block in an adapter structure and integrates event control.

Conclusion

The function block `AB_TO_AUDI` provides a compact and reusable solution for converting BYTE data via an AB adapter to UDINT data via an AUDI adapter. The use of standardized conversion and adapters ensures high compatibility and clear interfaces in IEC 61499 systems.

AUDI_TO_AB





• – ◦ • – ◦ • – ◦ •

Introduction

The **AUDI_TO_AB** function block is a composite function block (FB) that converts a UDINT interface (AUDI, 32-bit unsigned integer) into a BYTE interface (AB, 8-bit). It encapsulates the conversion logic and provides two unidirectional adapters: a socket for the AUDI input and a plug for the AB output. This function block is typically used to enable data communication between systems that use different data types via adapters.

Interface Structure

Event Inputs

Name	Type	Description
`AUDI_IN.E1`	Event (via adapter)	Trigger for data conversion – an incoming event starts the conversion of the current UDINT value to a BYTE value.

Event Outputs

Name	Type	Description
`AB_OUT.E1`	Event (via adapter)	Conversion confirmation – after successful conversion, an event is output that signals the new BYTE value to the downstream system.

Data Inputs

Name	Type	Description

AUDI_IN.D1	UDINT (32-bit)	The 32-bit unsigned integer value to be converted, provided via the AUDI adapter.

Data Outputs

Name	Type	Description
AB_OUT.D1	BYTE (8-bit)	The converted 8-bit value, output via the AB adapter.

Adapters

Name	Type	Direction	Description
AUDI_IN	adapter::types::unidirectional::AUDI	Socket (Input)	Unidirectional adapter that provides events and data of type UDINT.
AB_OUT	adapter::types::unidirectional::AB	Plug (Output)	Unidirectional adapter that outputs events and data of type BYTE.

Functionality

1. An external system sends an event via the adapter AUDI_IN (E1).
2. This event triggers the internal conversion block F_UDINT_TO_BYTE (from the IEC 61131 library) via its REQ input.
3. Simultaneously, the current data value from AUDI_IN.D1 is transferred to the IN input of the conversion module.
4. The conversion module converts the UDINT value into a BYTE value and stores the result at its OUT output.
5. After the conversion is complete, F_UDINT_TO_BYTE sends an acknowledgment event (CNF), which is forwarded to the AB_OUT adapter (E1).
6. Simultaneously, the converted BYTE value is output to AB_OUT.D1 and is available to the receiving system. ...`` The entire processing is event-driven and without intermediate storage – each incoming event generates exactly one outgoing event.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Structure**: The function block is implemented as a network of sub-functions. It uses the predefined conversion function block F_UDINT_TO_BYTE from the IEC 61131 library, ensuring standardized and testable conversion.
- **Unidirectional Adapters**: Both the input and output adapters are unidirectional. This reduces coupling and enables easy integration into data flows that only require one direction.
- **No State Storage**: The function block is purely combinatorial (no internal states). It reacts to each incoming event immediately and outputs the result without delay.
- **Compatibility**: Complies with the IEC 61499-2 standard and can be used in environments that support unidirectional adapters (AUDI/AB).

State Overview

This function block does not have an explicit state machine. It operates event-driven according to the principle "input event → conversion → output event". There are no queues or delays – processing is atomic and occurs within a single cycle.

Application Scenarios

- **Data Adaptation in Adapter-Based Systems**: If a component or protocol delivers a UDINT value via an AUDI adapter, but the downstream function block expects a BYTE value via an AB adapter, this function block can be used for seamless coupling.

- **Transition from 32-bit to 8-bit interfaces**: For example, in sensor-actuator communication, where a sensor provides 32-bit raw data, but the actuator only processes 8-bit commands.
- **Test and simulation environments**: Allows easy replacement of adapters without having to change the entire data path logic.

Comparison with similar function blocks

Function block	Description	Difference to AUDI_TO_AB
`F_UDINT_TO_BYTE`	Direct conversion function without adapters	Expects standard event and data interfaces; AUDI_TO_AB encapsulates the conversion in adapter signatures.
`UDINT_TO_BYTE` (Composite with standard ports)	Similar logic, but with a SOCKET/PLUG interface	May use different adapter types or bidirectional interfaces; AUDI_TO_AB is specialized for unidirectional adapters.

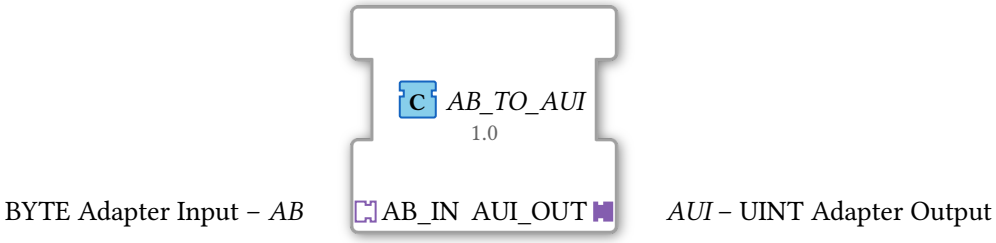
Conclusion

The **AUDI_TO_AB** function block offers a clean, reusable solution for type conversion between UDINT and BYTE values using unidirectional adapters. Its compact composite structure, based on a standardized IEC conversion function, makes it reliable and easy to integrate. Its event-driven architecture without state storage makes it particularly suitable for time-critical data flows in automation technology.

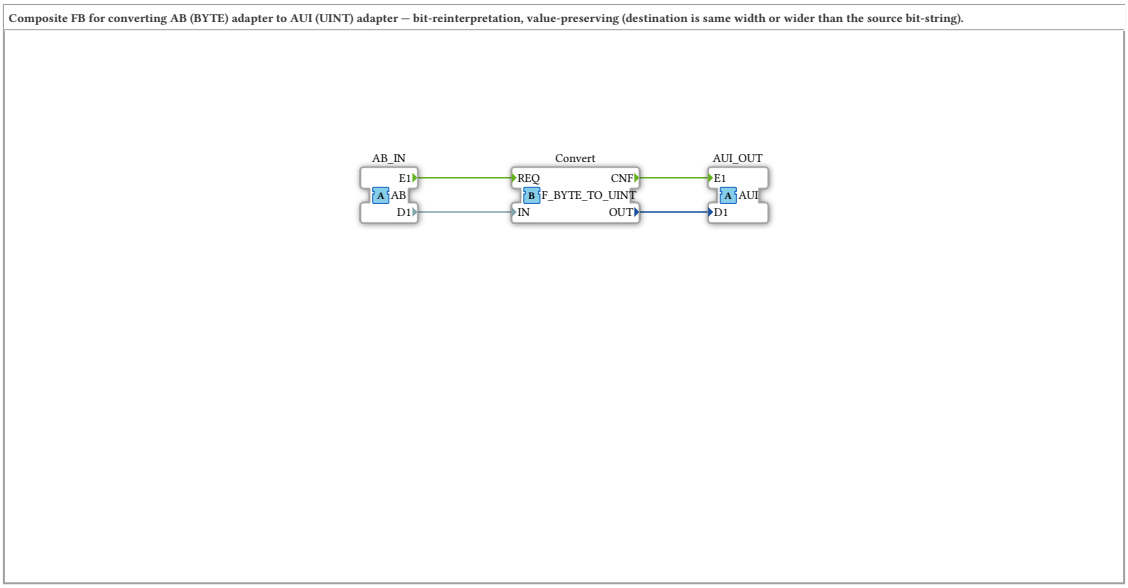
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AB_TO_AUI



BYTE Adapter Input – *AB* — *AUI* – UINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AB_TO_AUI** serves as an adapter converter between an AB adapter (BYTE-based) and an AUI adapter (UINT-based). It performs a unidirectional conversion from BYTE to UINT data type. The block is implemented as a composite FB and internally uses the IEC 61131-3 conversion function `F_BYTE_TO_UINT`. It is specifically designed for connecting sensors or actuators that deliver their data in BYTE format but must be processed in a UINT-based environment.

Interface Structure

Event Inputs

The FB does not have direct event inputs at the top level. Event control is handled via the connected adapters.

Event Outputs

The FB has no direct event outputs at the top level. Events are passed on via the connected adapters.

Data Inputs

The FB has no direct data inputs at the top level. Data is received via the connected AB adapter.

Data Outputs

The FB has no direct data outputs at the top level. Data is output via the connected AUI adapter.

Adapters

Direction	Name	Adapter Type	Description
Socket	AB_IN	`adapter::types::unidirectional::AB`	Input adapter for BYTE data (source adapter)
Plug	AUI_OUT	`adapter::types::unidirectional::AUI`	Output adapter for UINT data (destination adapter)

Note: Adapters of type `AB` and `AUI` each have an event input (`E1`), an event output (`E1`), a data input (`D1` of type BYTE or UINT), and a data output (`D1` of type BYTE or UINT). See the corresponding adapter definitions for complete details.

Functionality

The module functions as a pure converter: As soon as an event occurs at socket `AB_IN` (typically via the adapter's event input), it is forwarded to the internal function block `F_BYTE_TO_UINT`. Simultaneously, the BYTE value provided by the adapter (via the AB adapter's data input `D1`) is passed to the conversion function block. The conversion function block converts the BYTE value into an equivalent UINT value. After successful conversion, an event is triggered via the event output of the internal function block, which is then passed to plug `AUI_OUT`. The converted UINT value is transferred via the conversion function block's data output to the AUI adapter's data input.

The following diagram shows the internal wiring:

- Event connection: `AB_IN.E1` ⟶ `Convert.REQ` ⟶ `Convert.CNF` ⟶ `AUI_OUT.E1`
- Data connection: `AB_IN.D1` ⟶ `Convert.IN` ⟶ `Convert.OUT` ⟶ `AUI_OUT.D1`

The entire process is strictly sequential and event-driven.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Unidirectional Operation:** The function block only supports data flow from AB (BYTE) to AUI (UINT). Reverse conversion is not possible with this function block.
- **Use of IEC 61131-3 Conversion:** The conversion is performed using the standardized function block `F_BYTE_TO_UINT` from the IEC 61131 library. This ensures reliable and portable conversion.
- **Adapter-Based Interface:** The function block is intentionally designed as an adapter converter and not as a standalone function block with direct inputs/outputs. This allows for easy integration into existing adapter-based communication structures.

- **No State Storage:** The function block is stateless, meaning it does not store intermediate values. Each conversion process is independent of previous ones.

State Overview

The function block does not have its own state machines. Its behavior is determined by the internal function block `F_BYTE_TO_UINT` (a simple function block without states) and event chaining. Therefore, the function block is always ready to perform a conversion as soon as an event arrives at the input adapter.

Application Scenarios

- **Sensor Connection:** A sensor delivers measured values in BYTE format (e.g., via an AB adapter), while the control system processes only UINT values. The AB_TO_AUI function block converts the values accordingly.
- **Protocol Adaptation:** In heterogeneous fieldbus systems, the function block can be used as a simple data type converter between different adapter types.
- **Testing and Simulation:** During development, this function block can be used to convert BYTE data from a test environment into UINT data for the target application.

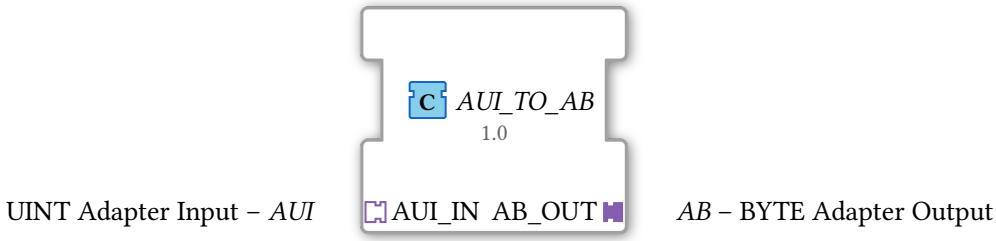
Comparison with Similar Function Blocks

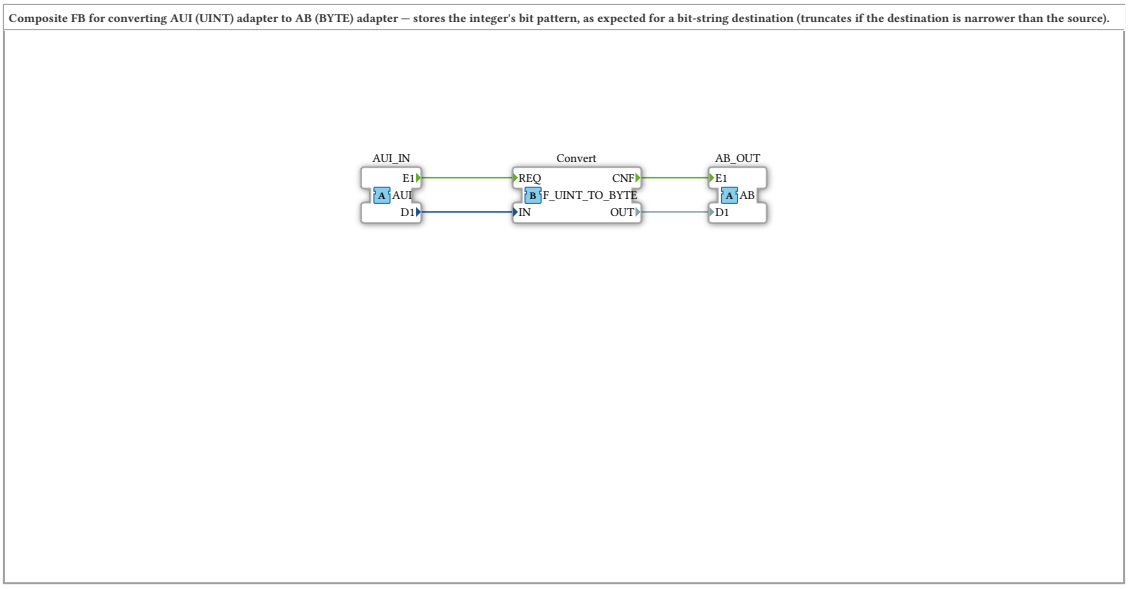
Other adapter converters exist, such as `AB_TO_UDINT`, `BYTE_TO_WORD`, or direct converters like `F_BYTE_TO_SINT`. The `AB_TO_AUI` focuses specifically on adapter conversion while preserving the adapter structure. Unlike pure function blocks, it retains the adapter interfaces, thus enabling easy encapsulation in adapter-based architectures. Other blocks, such as the `F_BYTE_TO_UINT`, operate at the signal level and require separate interfaces.

Conclusion

The `AB_TO_AUI` is a compact and specialized composite function block for the unidirectional conversion of BYTE to UINT data using standardized adapter interfaces. It allows for seamless integration into adapter-based systems and reduces the effort required for manual data conversion. Thanks to the use of the IEC 61131-3 conversion block, its functionality is reliable and portable. This block is particularly suitable for applications that rely on loose coupling between data sources and sinks.

AUI_TO_AB





No image available

• – ○ • – ○ • – ○ •

Introduction

The **AUI_TO_AB** function block is a composite block that converts an AUI interface (UINT) to an AB interface (BYTE). It encapsulates the conversion using the internal IEC 61131 block `F_UINT_TO_BYTE` and provides a uniform, unidirectional data flow via adapters. This block was developed for use in automation systems that need to transfer different data types between adapters.

Interface Structure

Event Inputs

Name	Description	Data Type
`AUI_IN.E1`	Event via the AUI adapter (socket) that triggers the conversion.	Event

Event Outputs

Name	Description	Data Type
`AB_OUT.E1`	Event via the AB adapter (plug) that is sent after successful conversion.	Event

Data Inputs

Name	Description	Data Type
`AUI_IN.D1`	UINT value that is received and converted via the AUI adapter.	UINT

Data Outputs

Name	Description	Data Type
`AB_OUT.D1`	Converted BYTE value output via the AB adapter.	BYTE

Adapters

Type	Name	Direction	Comment
`adapter::types::unidirectional::AUI`	`AUI_IN`	Socket	UINT adapter input

adapter::types::unidirectional::AB	AB_OUT	Plug	BYTE Adapter Output

Functionality

This component functions as a pure signal converter:

1. An incoming event at **AUI_IN.E1** is forwarded to the internal function block `Convert` (type `F_UINT_TO_BYTE`).
2. Simultaneously, the data value **AUI_IN.D1** is passed to the input `Convert.IN`.
3. The converter transforms the UINT value into a BYTE value (data truncation to the least significant byte).
4. Upon completion, the event **AB_OUT.E1** is triggered, and the converted value is made available via **AB_OUT.D1**.

Internally, the mapping logic is implemented by a single IEC 61131 component, resulting in minimal latency and deterministic operation.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Structure**: The function block is implemented as a composite, meaning it contains an internal network with a single function block. This facilitates reuse and encapsulation.
- **Adapter Concept**: Input and output are handled exclusively via typed adapters (`AUI` and `AB`). This enables loose coupling and easy exchange of communication protocols within the application.
- **License**: The function block is licensed under the Eclipse Public License 2.0 (EPL-2.0). The source code is open and can be freely modified.
- **Dependency**: It requires the IEC 61131 function block `F_UINT_TO_BYTE`, which must be included in the package `iec61131::conversion`.

State Overview

Since the function block does not have an internal state machine (composite without its own ECC), its state is determined solely by the internal function block `F_UINT_TO_BYTE`. This block operates in an event-driven manner and does not have a persistent state. After each event, the function block returns to its idle state.

Application Scenarios

- **Data Conversion**: When a sensor delivers data in UINT format via an AUI adapter, but the downstream system expects a BYTE value via an AB adapter.
- **System Integration**: As an intermediary between different adapter types in heterogeneous automation networks.
- **Protocol Conversion**: In scenarios where existing libraries only support certain adapter types and type conversion is required.

Comparison with Similar Function Blocks

- **Direct `F_UINT_TO_BYTE`**: This IEC function block is also used internally, but operates without an adapter – it requires direct data and event ports. `AUI_TO_AB` encapsulates this interface in adapter form.
- **Other Adapter Converters**: Similar function blocks exist, such as `AB_TO_AUI` or `ADI_TO_AB`, for other type combinations. They follow the same pattern and are interchangeable.

The key advantage of `AUI_TO_AB` is its complete integration into adapter-based architectures – without the application needing to know the internal conversion logic.

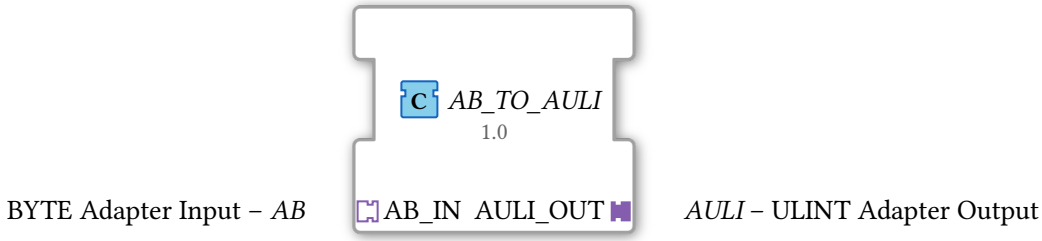
Conclusion

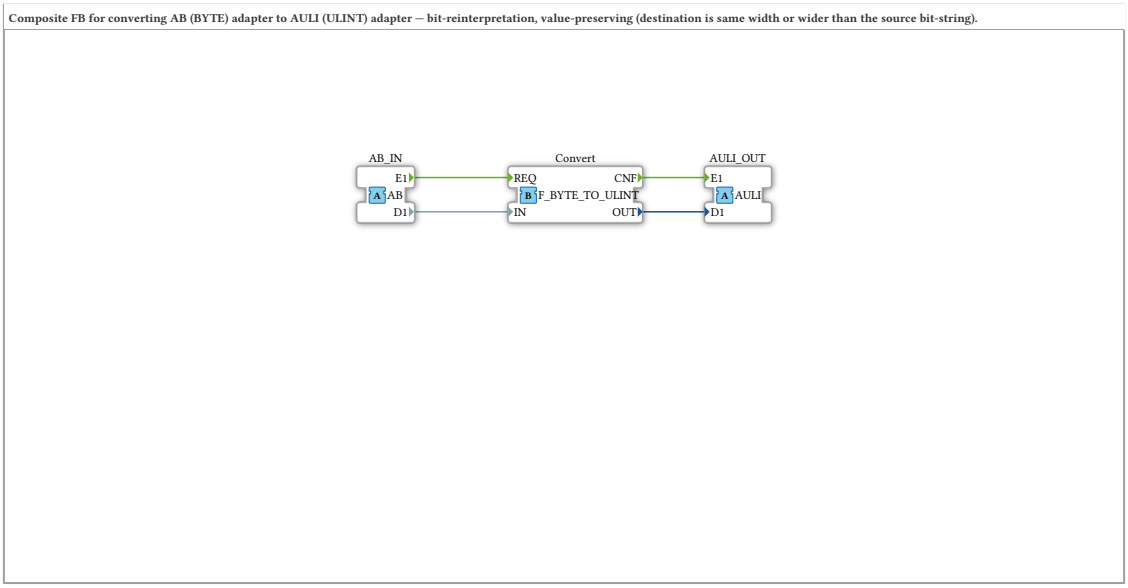
The **AUI_TO_AB** function block is a simple yet useful composite function block for type conversion between two unidirectional adapters. Its clear structure and the use of a standardized converter make it reliable and easy to understand. It represents an elegant solution for all applications requiring clean adapter separation.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AB_TO_AULI





• – ○ • – ○ • – ○ •

Introduction

The function block **AB_TO_AULI** is a composite function block (FB) that converts a unidirectional adapter of type `AB` (BYTE) into a unidirectional adapter of type `AULI` (ULINT). It serves as a converter between different data types in adapter communication according to IEC 61499-2. The FB internally uses the standardized conversion block `F_BYTE_TO_ULINT` from the IEC 61131 library.

Interface Structure

Event Inputs

- **AB_IN.E1** – Event input of the AB adapter socket. Triggered by the incoming adapter protocol, this starts the conversion.

Event Outputs

- **AULI_OUT.E1** – Event output of the AULI adapter plug. Signals the completion of the conversion to the downstream adapter path.

Data Inputs

- **AB_IN.D1** (Data type: BYTE) – Receives the value to be converted via the AB adapter.

Data Outputs

- **AULI_OUT.D1** (Data type: ULINT) – Provides the converted ULINT value via the AULI adapter.

Adapters

- **Socket “AB_IN”** – Type: `adapter::types::unidirectional::AB`

Unidirectional input adapter for the BYTE data type.

- **Plug "AULI_OUT"** – Type: `adapter::types::unidirectional::AULI`

Unidirectional output adapter for the ULINT data type.

Functionality

The function block (FB) waits for an event at the input adapter `AB_IN`. As soon as the event arrives, the BYTE value at `AB_IN.D1` is forwarded to the internal conversion block `F_BYTE_TO_ULINT` and converted into a ULINT value. After the conversion is complete, the resulting value is output to `AULI_OUT.D1`, and the completion event is sent to `AULI_OUT.E1`. The entire process is event-driven and does not use state machines.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- The function block does not implement any states or algorithms of its own – it is simply a standard conversion FB interconnect.
- The conversion is performed according to the rules of IEC 61131-3: The BYTE value is interpreted as an 8-bit representation of a ULINT and converted into an unsigned 64-bit integer.
- The adapters are based on a unidirectional communication pattern, meaning that data and events flow only in one direction (from the socket to the plug).
- The function block (FB) is not stateful; each request is processed independently.

State Overview

The function block does not have its own state machine. Execution follows a simple event-driven sequence:

1. Wait for an input event (`AB_IN.E1`).
2. Convert the data value.
3. Output the result and trigger the output event (`AULI_OUT.E1`).

Application Scenarios

- **Connecting BYTE-based sensors** to a controller that uses ULINT as a uniform data type for longer measured values.
- **Data format conversion** in adapter chains when a protocol communicates at the BYTE level, but the application expects a ULINT value.
- **Test and debugging setups** where a BYTE test adapter needs to be converted to a ULINT interface.

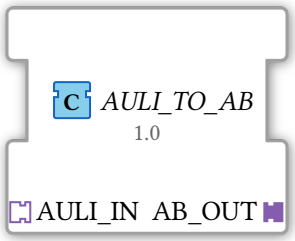
Comparison with Similar Function Blocks

- **Direct conversion without adapters** (e.g., `F_BYTE_TO_ULINT`): These function blocks operate at the data level, not the adapter level. The `AB_TO_AULI` encapsulates the conversion, including the adapter interfaces.
- **Adapter converters for other data types** (e.g., `AB_TO_AUDINT`): These are structurally similar but differ in the target data type and, if applicable, in the conversion function used.
- **Bidirectional adapter converters**: If required, two unidirectional converters (e.g., `AB_TO_AULI` and `AULI_TO_AB`) would need to be combined.

Conclusion

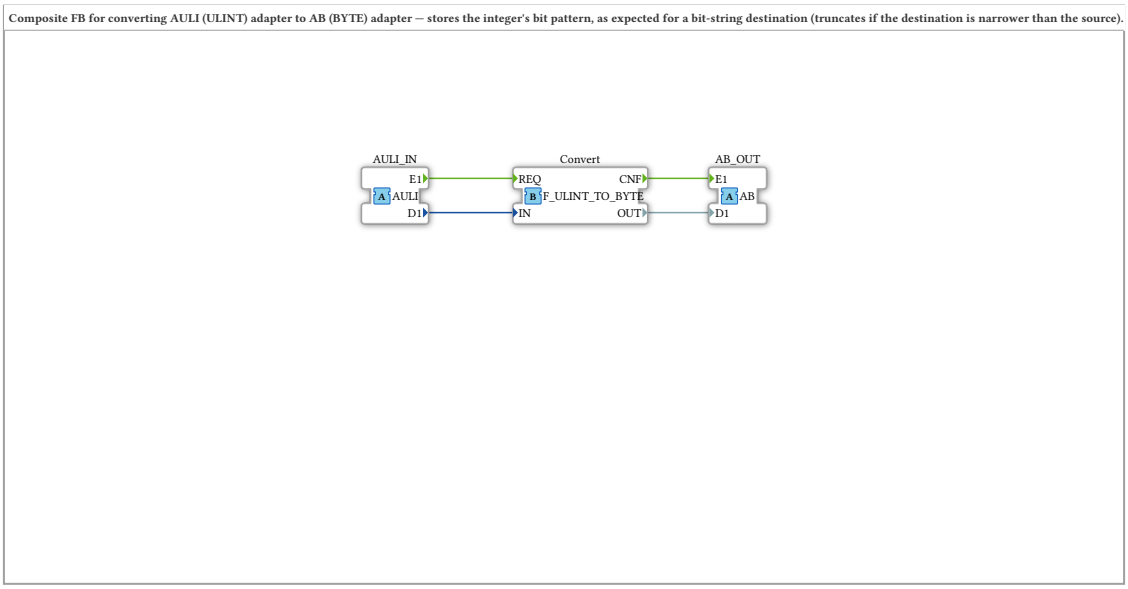
The function block `AB_TO_AULI` provides a simple, standards-compliant way to convert a BYTE adapter to a ULINT adapter. By using an established IEC 61131 conversion block, the conversion is reliable and easily traceable. The block is ideal for adapter-based communication chains that require a data type change at the adapter level.

AULI_TO_AB



ULINT Adapter Input – *AULI*

AB – BYTE Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `AULI_TO_AB` is a composite function block that converts an AULI adapter (ULINT data type) into an AB adapter (BYTE data type). It enables the seamless connection of components using different adapter interfaces by converting the incoming data and providing it as a compatible output.

Interface Structure

The function block has no direct event or data inputs/outputs. All input and output is handled via adapters.

Event Inputs

None (events are received via the AULI adapter).

Event Outputs

None (events are sent via the AB adapter).

Data Inputs

None (data is received via the AULI adapter).

Data Outputs

None (data is sent via the AB adapter).

Adapters

Name	Type	Direction	Comment
`AULI_IN`	`adapter::types::unidirectional::AULI`	Socket (Input Adapter)	ULINT adapter that provides the values to be converted and the trigger event

AB_OUT	adapter::types::unidirectional::AB	Plug (Output Adapter)	BYTE adapter that provides the converted values and the confirmation event

Functionality

The component internally uses the IEC 61131 component F_ULINT_TO_BYTE for ULINT to BYTE conversion. The process is as follows:

1. **Input**: An event (E1) is received at socket AULI_IN, triggering the conversion. Simultaneously, the data value (D1) is present as a ULINT.
2. **Processing**: The event and data are forwarded to the internal component F_ULINT_TO_BYTE (event REQ, data IN). This component performs the type conversion.
3. **Output**: After successful conversion, the internal block signals an event (CNF) and delivers the result (OUT) as a BYTE. Both are transferred to the plug AB_OUT (event E1, data D1).

The entire process occurs synchronously within one execution cycle of the composite block.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Block**: The logic is implemented entirely through the internal network of predefined blocks – there is no separate ECC (Execution Control Chart).
- **License and Copyright**: The block is licensed under the Eclipse Public License 2.0 (EPL-2.0) and is subject to third-party copyright.
- **Dependencies**: It requires the IEC 61131 function block F_ULINT_TO_BYTE from the library iec61131::conversion.
- **Package Structure**: The function block belongs to the package adapter::conversion::unidirectional.

State Overview

Since the function block does not have its own ECC, there are no explicit states. The internal flow is strictly defined by the event chain AULI_IN.E1 → Convert.REQ → Convert.CNF → AB_OUT.E1.

Application Scenarios

- Integration of a ULINT-providing sensor (e.g., a high-resolution distance sensor) into a control system based on BYTE adapter interfaces.
- Retrofitting existing systems where components with different adapter types (ULINT vs. BYTE) need to be integrated.
- Prototype development where ULINT values are temporarily processed via a BYTE channel (e.g., simulation or testing).

Comparison with similar function blocks

Analog conversion function blocks exist for other data types (e.g., AULI_TO_xx or xx_TO_AB). The AULI_TO_AB is specifically optimized for converting ULINT to BYTE and utilizes the corresponding adapter types. Unlike generic function blocks, it avoids the manual wiring of individual events and data and offers an encapsulated, adapter-based interface.

Conclusion

The AULI_TO_AB is a compact, reliable composite function block for converting between two common adapter types in the 4diac IDE. It reduces wiring effort and simplifies the integration of heterogeneous components. By using the standardized IEC 61131 conversion block, the conversion remains type-safe and efficient.

🌐 Related topic subpages on ms-muc-docs.de

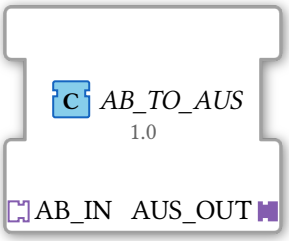
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Numeric Index (unidirectional)

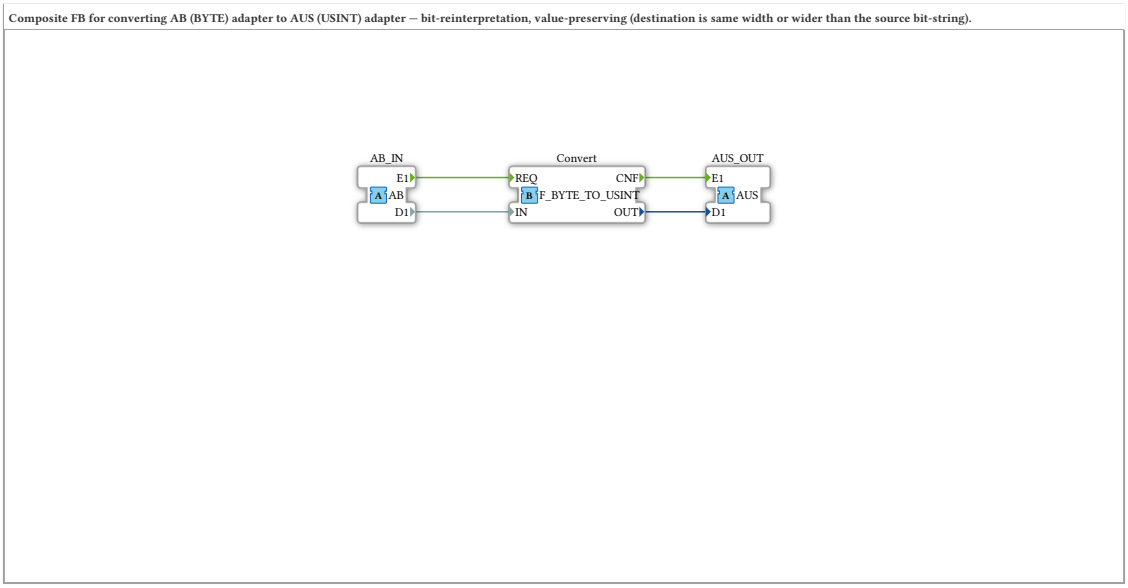
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AB_TO_AUS



BYTE Adapter Input – *AB*

AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AB_TO_AUS** is a composite block that performs a unidirectional conversion of data from the **AB adapter** (BYTE) to the **AUS adapter** (USINT). It encapsulates the necessary type conversion and event passing, thus enabling the seamless integration of components based on different data types.

Interface Structure

Event Inputs

The function block does not have separate event inputs at the top level. The triggering event is provided via the socket adapter **AB_IN** (available there as `E1`).

Event Outputs

There are no direct event outputs at the composite function block level. The resulting event is output via the plug adapter **AUS_OUT** (there as `E1`).

Data Inputs

No data inputs are defined at the top level. The input data (BYTE) is received via the socket adapter **AB_IN** (there as `D1`).

Data Outputs

There are no direct data outputs. The converted data (USINT) is provided via the plug adapter **AUS_OUT** (there as `D1`).

Adapters

Type	Name	Direction	Description
Socket	**AB_IN**	Input	Unidirectional adapter of type `adapter::types::unidirectional::AB`. Returns a BYTE (`D1`) and an event (`E1`).
Plug	**OFF_OUT**	Output	Unidirectional adapter of type `adapter::types::unidirectional::AUS`. Provides a USINT (`D1`) and an event (`E1`).

Functionality

1. An event at `AB_IN.E1` triggers processing.
2. The value of `AB_IN.D1` (BYTE) is passed to the internal converter **F_BYTE_TO_USINT**.
3. The converter transforms the BYTE into a USINT value. 4. The result is output to `AUS_OUT.D1` (USINT).
4. Simultaneously, an event is generated on `AUS_OUT.E1` to signal the subsequent processing.

The entire conversion is event-driven and performed in a single step.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Unidirectional Adapter Coupling**: The function block connects two unidirectional adapters (ON → OFF) and ensures that the data flow direction is clearly defined.
- **Reusable IEC Conversion**: Internally, the standardized function block `iec61131::conversion::F_BYTE_TO_USINT` is used.
- **Composite Architecture**: The conversion logic is encapsulated and can be easily integrated into other projects or replaced by alternative converters.
- **No Internal State**: The function block does not have its own state machine; it forwards events and data directly.

State Overview

The **AB_TO_AUS** function block does not have its own state machine. Its operation is based on a purely event-driven data conversion. The internal converter `F_BYTE_TO_USINT` also operates without state storage.

Application Scenarios

- **System Integration**: Connecting a device that provides measured values as BYTE via the AB adapter to a controller that expects USINT data via the AUS adapter.
- **Protocol Conversion**: Converting BYTE-encoded commands to USINT-encoded values for further processing in libraries that exclusively support USINT.
- **Modular Reuse**: Encapsulating the conversion as a standalone function block to improve clarity in large automation projects.

Comparison with Similar Function Blocks

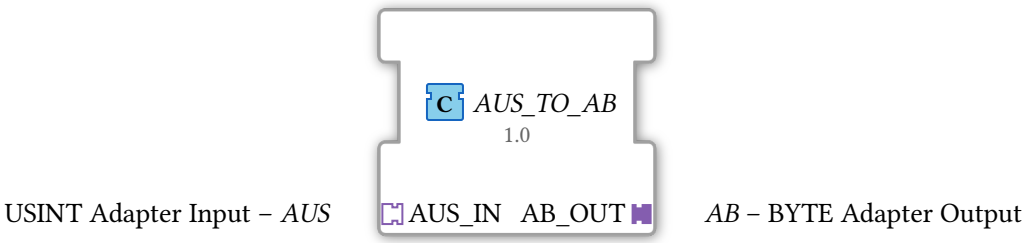
Function Block	Description
F_BYTE_TO_USINT	Simple IEC converter without adapter connection. Must be manually wired with event and data connections.
AB_TO_AUS	Composite function block that encapsulates the entire adapter conversion, including event forwarding, thus providing a clean, abstracted interface.
AUS_TO_AB (hypothetical)	Reverse conversion from USINT to BYTE – mirrored functionality for the opposite direction.

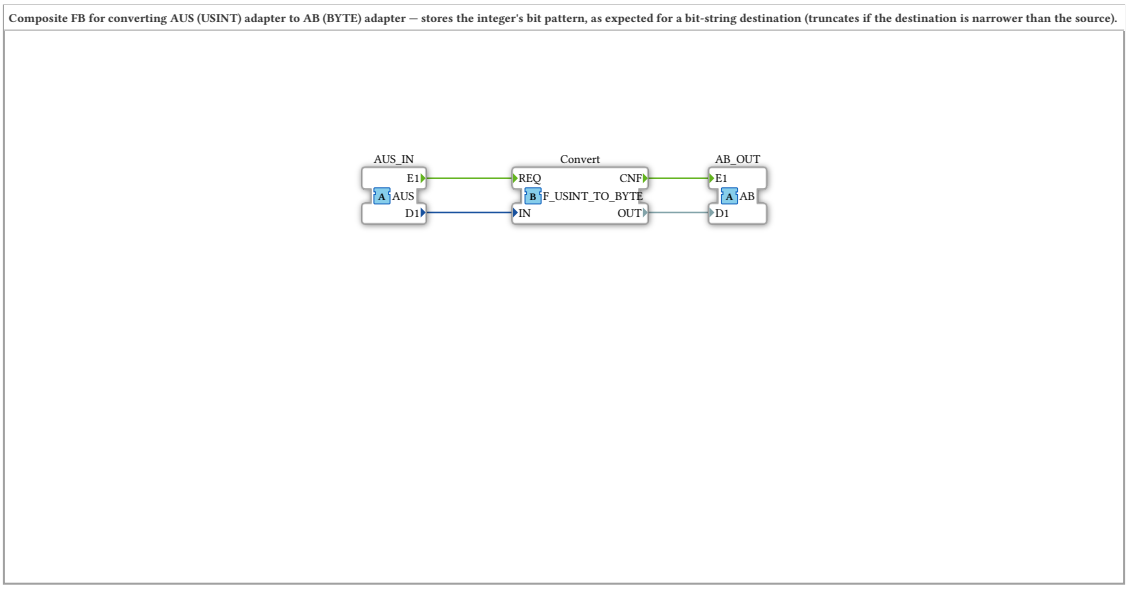
The **AB_TO_AUS** function block stands out due to its ease of use and direct adapter compatibility.

Conclusion

The **AB_TO_AUS** function block offers an elegant and standardized way to convert BYTE data from an AB adapter to USINT data for an AUS adapter. Encapsulating the conversion logic in a composite block promotes reusability and simplifies system integration. It is particularly suitable for modular automation solutions based on IEC 61499.

OFF_TO_AB





• – ○ • – ○ • – ○ •

Introduction

The **OFF_TO_AB** is a composite function block for converting a unidirectional **OFF adapter** (USINT data type) to an **AB adapter** (BYTE data type). It serves as a bridge between different adapter interfaces in automation systems and enables the seamless integration of components based on different data types. Internally, the block uses the standard conversion block *F_USINT_TO_BYTE* from the IEC 61131 library.

Interface Structure

Event Inputs

The block does not have its own event input ports. Events are received via the **OFF_IN** adapter (socket):

- **OFF_IN.E1** – Trigger for data conversion and forwarding to the output.

Event Outputs

Events are output via the **AB_OUT** adapter (plug):

- **AB_OUT.E1** – Confirmation that the converted value is present at the output.

Data Inputs

Data is provided via the **AUS_IN** adapter:

- **AUS_IN.D1** (USINT) – The 8-bit unsigned integer value to be converted.

Data Outputs

Data is passed through the **AB_OUT** adapter:

- **AB_OUT.D1** (BYTE) – The converted byte value.

Adapters

Name	Type	Direction	Description
AUS_IN	adapter::types::unidirectional::AUS	Socket	Input adapter for USINT data and events.
AB_OUT	adapter::types::unidirectional::AB	Plug	Output adapter for BYTE data and events.

Functionality

The **AUS_TO_AB** is implemented as a pure composite function block without its own logic. It connects the AUS_IN and AB_OUT adapters via the internal function block **F_USINT_TO_BYTE**:

1. An incoming event on **AUS_IN.E1** triggers the conversion function block **Convert** (`F_USINT_TO_BYTE`) via its **REQ** input.
2. Simultaneously, the data value from **AUS_IN.D1** is passed to the **Convert.IN** input.
3. After the conversion (USINT ⟶ BYTE) is complete, **Convert** acknowledges with an event on **CNF**.
4. This event is forwarded directly to **AB_OUT.E1**.
5. The converted byte value from **Convert.OUT** is passed to **AB_OUT.D1**.

This ensures synchronized, event-driven data transfer.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite FB**: Does not contain its own state machine; all logic is implemented by the internal FB `F_USINT_TO_BYTE`.
- **Unidirectional Adapters**: Both the incoming and outgoing adapters are unidirectional – data flows occur exclusively in one direction.
- **Standard Compliance**: The block is based on IEC 61499-2 and uses the IEC 61131 conversion function `F_USINT_TO_BYTE`. This makes it immediately usable in compatible runtime environments (e.g., 4diac-FORTE).
- **License**: Released under the Eclipse Public License 2.0 (EPL-2.0). Copyright © 2026 HR Agrartechnik GmbH.

State Overview

Since this is a composite function block without its own ECC (Execution Control Chart), there is no explicit state machine. Its behavior is purely event-driven and determined by the internal interconnection of the individual function blocks. After startup, the system is always ready – an incoming event is processed immediately.

Application Scenarios

- **System integration** in agricultural technology or other automation environments where an **OFF adapter** (USINT) needs to be connected to an **AB adapter** (BYTE).
- **Retrofitting older components** that provide a BYTE output with modern USINT-based sensors or actuators.
- **Prototyping and test setups** where rapid switching between different adapter types is required.
- **Data processing** in communication nodes that standardize different adapter protocols.

Comparison with Similar Function Blocks

- **F_USINT_TO_BYTE** – a direct function block for pure data type conversion, but without adapter integration. **AUS_TO_AB** adds the adapter and event logic.
- **BYTE_TO_USINT** – inverse conversion (BYTE ⟶ USINT) if the reverse adapter direction is required (not included with this function block).

- **Generic Adapter Converters** – more complex blocks that convert multiple data types simultaneously. **AUS_TO_AB** is optimized for the specific use case and is therefore more compact.

Conclusion

The **AUS_TO_AB** is a useful and lightweight function block for converting an **AUS adapter (USINT)** to an **AB adapter (BYTE)**. Its composite structure minimizes implementation effort, while the use of a standardized conversion block ensures reliability. This module is particularly suitable for the rapid integration of heterogeneous adapter interfaces into IEC 61499-based control systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

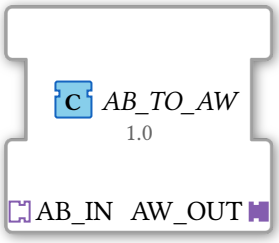
]

Numeric Index (unidirectional)

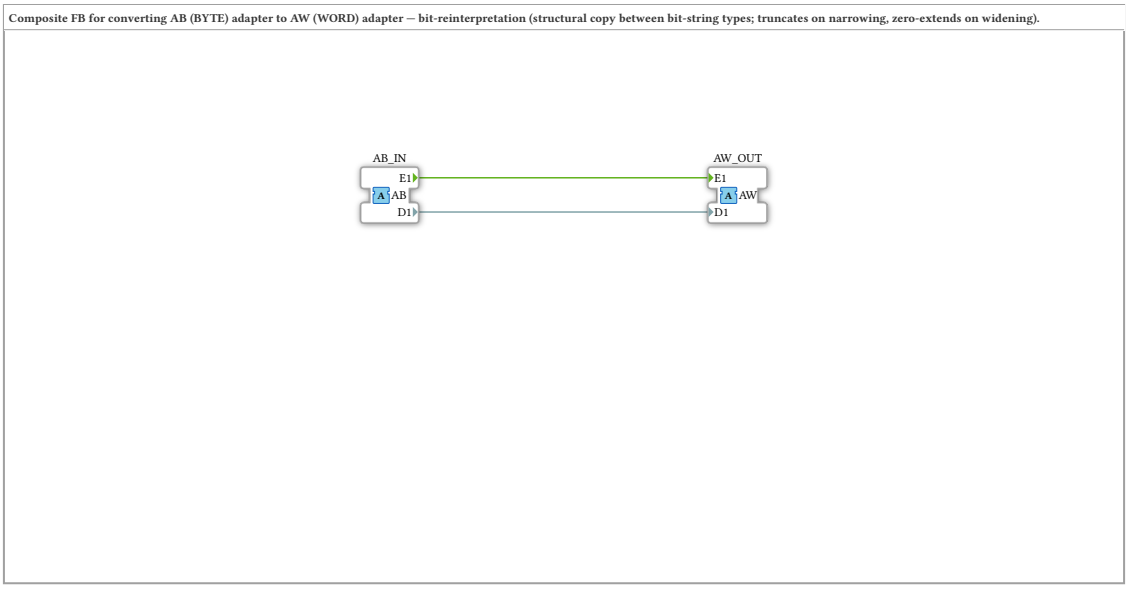
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AB_TO_AW

BYTE Adapter Input – *AB*



AW – WORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_TO_AW** is a composite block that performs a unidirectional conversion from an AB adapter (BYTE-based) to an AW adapter (WORD-based). It serves as an interface converter when a system component delivers data in BYTE format, but the downstream component expects a WORD format.

Interface Structure

The block does not have any classic event or data inputs/outputs at the top level, but uses only adapter interfaces.

Event Inputs

- No direct event inputs are available. Events are received via the **AB_IN** adapter.

Event Outputs

- No direct event outputs are available. Events are sent via the **AW_OUT** adapter.

Data Inputs

- No direct data inputs are available. Data is received via the **AB_IN** adapter.

Data Outputs

- No direct data outputs are available. Data is sent via the **AW_OUT** adapter.

Adapters

Direction	Name	Type	Comment
Input (Socket)	**AB_IN**	`adapter::types::unidirectional::AB`	BYTE adapter input, provides a BYTE date and an event
Output (Plug)	**AW_OUT**	`adapter::types::unidirectional::AW`	WORD adapter output, provides a WORD date and an event

Functionality

This function block forwards the incoming signals from the **AB_IN** adapter directly to the **AW_OUT** adapter:

- The **E1** event of the **AB_IN** is linked to the **E1** event of the **AW_OUT**.
- The **D1** (BYTE) data of the **AB_IN** is linked to the **D1** (WORD) data of the **AW_OUT**.

Since this is a simple pass-through, no data conversion in the sense of a value change occurs; the BYTE data is interpreted as WORD data (implicit type conversion via the data connection). The actual conversion (e.g., extending the value range) depends on the implementation of the underlying adapters.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- The function block is implemented as a **composite function block**, meaning its behavior is entirely defined by the internal network (FBNetwork).
- The conversion is unidirectional: only from the AB adapter to the AW adapter.
- The connection is established at the event and data level without intermediate storage or synchronization.
- The function block is part of the package structure `adapter::conversion::unidirectional`.

State Overview

The function block does **not have its own state machine**. Its behavior is purely combinatorial: Every incoming event is immediately forwarded to the output, as is the data.

Application Scenarios

- **Connecting Sensors/Actuators**: A module provides a BYTE value (e.g., via an AB adapter), but downstream logic expects WORD format.
- **Protocol Adaptation**: In heterogeneous systems where different adapter types interact.
- **System Integration**: When an existing BYTE-based sequence control needs to be integrated into a WORD-based environment.

Comparison with Similar Components

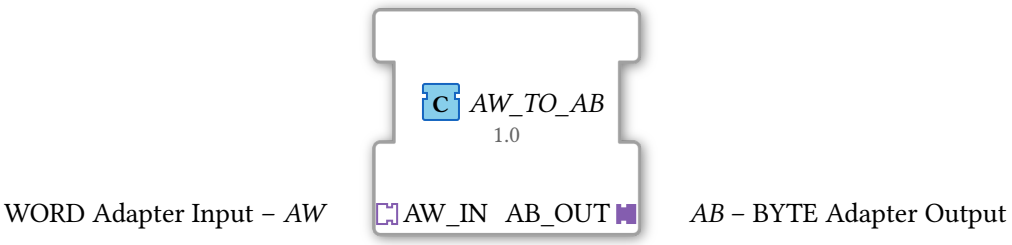
- **WORD_TO_BYTE**: An analogous component may exist for the reverse direction (WORD ⟶ BYTE).
- **Direct Adapter Connection**: Without this component, the programmer would have to manually link the event and data in the network; **AB_TO_AW** encapsulates this connection and increases reusability.

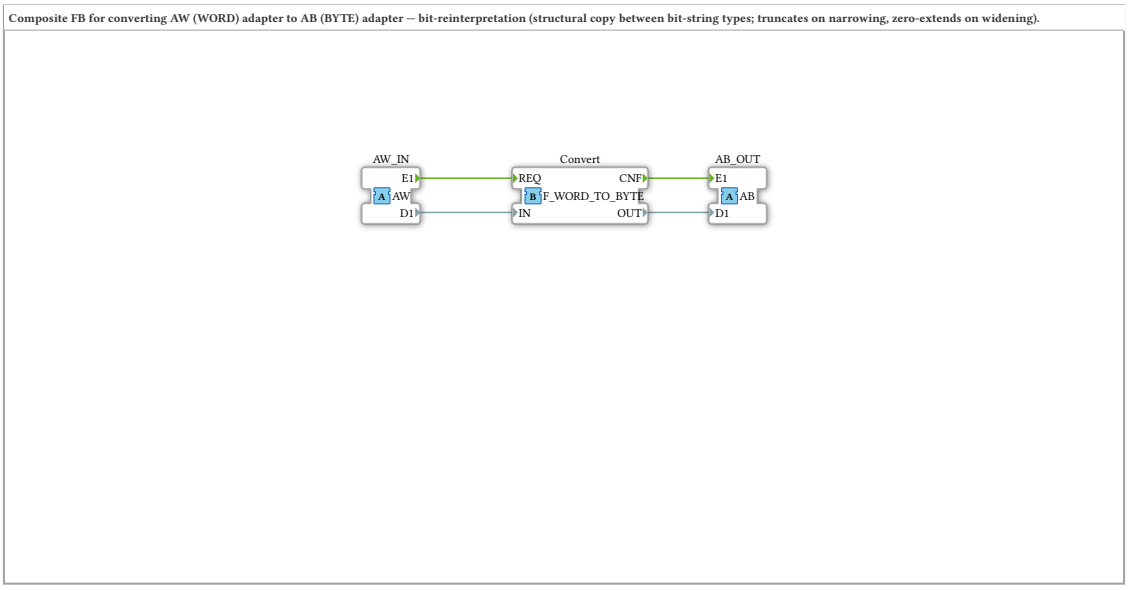
- **Type Converter**: Pure data converters (e.g., `BYTE_TO_WORD`) often operate without event forwarding; here, the event is also passed through.

Conclusion

The **AB_TO_AW** is a simple but useful interface component for the unidirectional conversion of a BYTE adapter to a WORD adapter. By encapsulating the connection logic, clarity in the application network is improved and the reuse of standardized conversion patterns is promoted.

AW_TO_AB





• – ○ • – ○ • – ○ •

Introduction

The function block `AW_TO_AB` is a composite block that converts a unidirectional WORD adapter (`AW`) into a unidirectional BYTE adapter (`AB`). It serves as an interface converter between components based on different data types. The conversion is performed by the integrated standard function block `F_WORD_TO_BYTE`.

Interface Structure

Event Inputs

The block does not have its own event inputs. Instead, the trigger signal is provided via the adapter input `AW_IN` (event `E1`).

Event Outputs

The function block does not have its own event outputs. The result is signaled via the adapter output `AB_OUT` (event `E1`).

Data Inputs

The function block does not have its own data inputs. The WORD value to be converted is provided via the adapter input `AW_IN` (data `D1`).

Data Outputs

The function block does not have its own data outputs. The converted BYTE result is output via the adapter output `AB_OUT` (data `D1`).

Adapter

Name	Direction	Type	Description
`AW_IN`	Socket (Input)	`adapter::types::unidirectional::AW`	WORD adapter with one event `E1` and one data value `D1` (WORD)
`AB_OUT`	Plug (Output)	`adapter::types::unidirectional::AB`	BYTE adapter with one event `E1` and one data value `D1` (BYTE)

Functionality

This function block performs a simple data conversion from WORD (16 bits) to BYTE (8 bits). The process is as follows:

1. An event at the adapter input `AW_IN.E1` triggers the internal function block `F_WORD_TO_BYTE`.
2. The data value `AW_IN.D1` (WORD) is passed as an input to `F_WORD_TO_BYTE.IN`.
3. After the conversion, the result is forwarded via `F_WORD_TO_BYTE.OUT` to the adapter output `AB_OUT.D1`.
4. The output `AB_OUT.E1` is activated to signal completion.

The conversion according to IEC 61131-3 returns the least significant byte (low byte) of the WORD value by default.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Unidirectional Adapter**: The module operates in only one data flow direction – from the socket to the plug.
- **Integrated Conversion**: The actual conversion is performed by the established module `F_WORD_TO_BYTE`, so no custom logic needs to be implemented.
- **Data Loss**: Since a WORD (16 bits) is mapped to a BYTE (8 bits), the upper 8 bits are lost. This must be taken into account in the application.
- **Package Structure**: The function block is located in package `adapter::conversion::unidirectional` and uses the import `iec61131::conversion::F_WORD_TO_BYTE`.

State Overview

The composite function block does not have its own state machine. Its operation is purely event-driven:

- **Idle**: Waiting for an event at socket `AW_IN`.
- **Conversion Active**: After receiving an event, the internal function block `F_WORD_TO_BYTE` is executed (combinatorial logic without wait cycles).
- **Output**: Once the conversion is complete, the event is output to plug `AB_OUT`, along with the new data value.

Application Scenarios

- **Protocol Adaptation**: When a sensor provides a WORD value, but the actuator only supports BYTE-capable adapters.
- **Interface Reduction**: In hierarchical control systems where byte sizes are used for performance or memory reasons.
- **Adapter Bridge**: As an intermediary in a 4diac IDE application to connect components with incompatible adapter types.

Comparison with Similar Blocks

- **`F_WORD_TO_BYTE`**: A pure data conversion block without adapter integration. `AW_TO_AB` encapsulates this block and provides a standardized adapter interface.
- **`AW_TO_AW` (identical type)**: Not useful; this block is specifically designed for type conversion.

- **Manual Conversion**: Without this component, the developer would have to manually insert `F_WORD_TO_BYTE` into the network and wire the adapter connections themselves – this composite component simplifies this process.

Conclusion

AW_TO_ABis `a simple yet useful composite component for converting a WORD adapter to a BYTE adapter. It reduces the wiring effort in 4diac applications and ensures consistent conversion by reusing the standard componentF_WORD_TO_BYTE`. Users should be aware of the potential data loss due to the reduced bit width.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

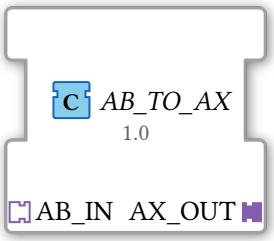
]

Numeric Index (unidirectional)

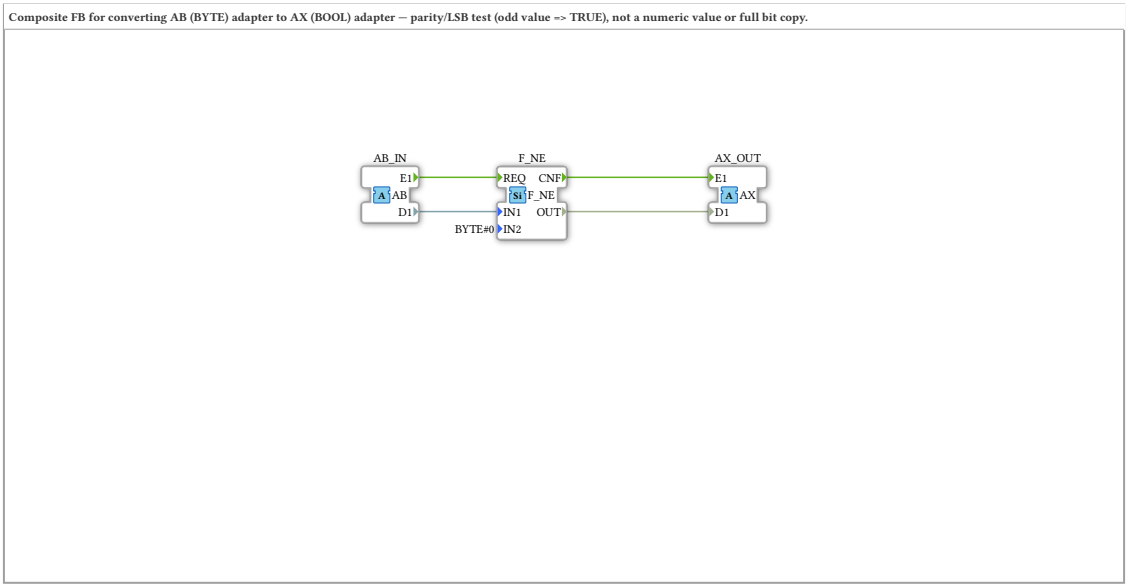
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AB_TO_AX

BYTE Adapter Input – *AB*



AX – BOOL Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_TO_AX** is a composite function block (FB) for converting a BYTE signal (via the AB adapter) into a BOOL signal (via the AX adapter). It serves as a converter between two different adapter types of the unidirectional adapter family.

Interface Structure

Adapter (Input)

- **AB_IN** (Type: `adapter::types::unidirectional::AB`):

Accepts a BYTE value and an associated event. The adapter provides the data via the internal channels D1 (BYTE) and E1 (event).

Adapter (Output)

- **AX_OUT** (Type: `adapter::types::unidirectional::AX`):

Outputs a BOOL value and a triggering event. The adapter provides the result via D1 (BOOL) and E1 (event).

Functionality

The function block internally uses a `F_NE` comparison block (not equal to). The incoming BYTE signal (AB_IN.D1) is compared to the constant value `BYTE#0`.

- If the BYTE value is not equal to 0, the comparison returns `TRUE`.
- If the BYTE value is equal to 0, the comparison returns `FALSE`.

The process is started by the event `AB_IN.E1`. This triggers the `F_NE` function block. After the comparison is complete, the event `AX_OUT.E1` is triggered, and the result (BOOL) is output via `AX_OUT.D1`.

Technical Features

- **Parity/LSB test** (odd value => TRUE), not a numeric value or full bit copy.
- The function block is implemented as a composite, meaning its functionality results from the interconnection of standard function blocks (here, `F_NE`).
- It is designed for use in the 4diac IDE with the IEC 61499-2 specification.
- The adapters `AB` and `AX` are unidirectional, ensuring a clear separation of data and event flows.
- The internal comparison is binary: Each non-zero BYTE value is interpreted as `TRUE`.

State Overview

The function block does not have its own state machine. All processing is event-driven via internal connections. The `F_NE` function block operates deterministically without intermediate states.

Application Scenarios

- **Signal Conversion**: A digital sensor provides a byte value (e.g., counter reading or status). AB_TO_AX converts this into a Boolean signal, which, for example, triggers an alarm as soon as the value is not zero.
- **Adapter Bridge**: An AB adapter is used in a control component, but the downstream logic expects an AX adapter. The function block acts as a simple translator.
- **Threshold Replacement**: Replaces an explicit comparison block with a compact adapter converter.

Comparison with Similar Function Blocks

Other converter function blocks, such as `BOOL_TO_BYTE` or `BYTE_TO_BOOL`, exist as simple data converters, but usually operate at the data level without an adapter. AB_TO_AX integrates the adapter interface, thus enabling seamless integration into adapter-based architectures. Unlike a general comparison function block (`F_NE`), this function block encapsulates the specific conversion logic and offers a clean interface.

Conclusion

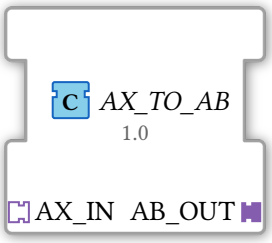
The **AB_TO_AX** function block provides a clean and simple method for converting a BYTE signal from an AB adapter into a BOOL signal for an AX adapter. By using the standard comparison `Ungleich Null`, it is particularly suitable for binary threshold decisions and simplifies reuse in adapter-oriented control projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

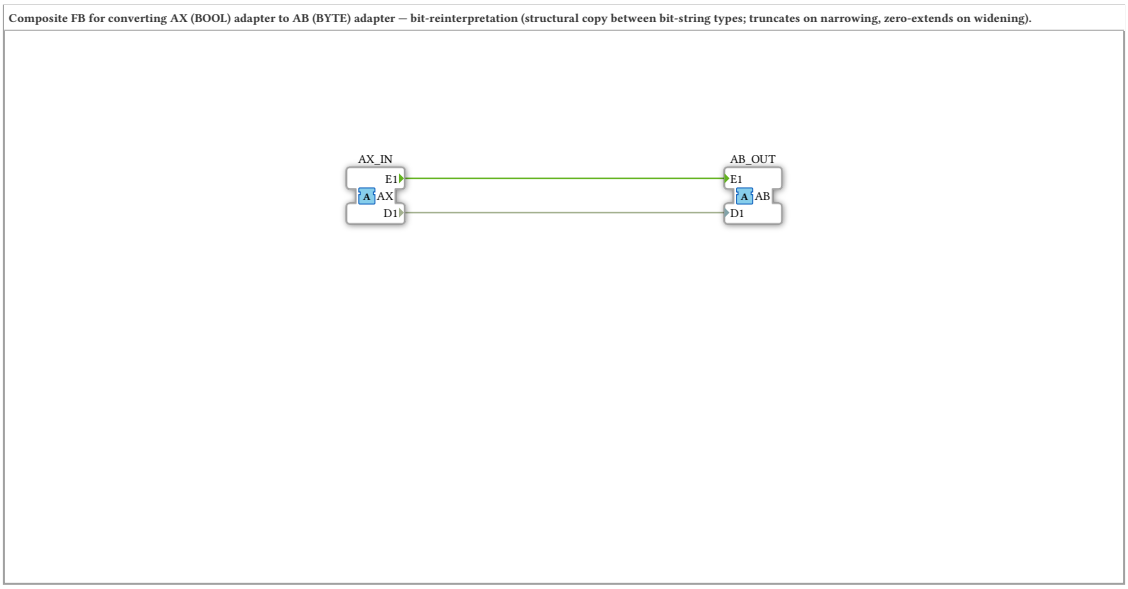
]

AX_TO_AB



BOOL Adapter Input – *AX*

AB – BYTE Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The composite function block `AX_TO_AB` converts a BOOL adapter signal of type `AX` into a BYTE adapter signal of type `AB`. It encapsulates the type conversion and event passing in a reusable unit.

Interface Structure

Event Inputs

Name	Description
`E1` (from `AX_IN`)	Event input received via the BOOL adapter and passed on unchanged to the output.

Event Outputs

Name	Description
`E1` (from `AB_OUT`)	Event output that provides the event passed through from the input.

Data Inputs

Name	Data Type	Description
`D1` (from `AX_IN`)	`BOOL`	The Boolean signal to be converted (TRUE or FALSE).

Data Outputs

Name	Data Type	Description
`D1` (from `AB_OUT`)	`BYTE`	The converted signal: `TRUE` becomes `1`, `FALSE` becomes `0`.

Adapter

Type	Direction	Identifier	Comment
adapter::types::unidirectional::AX	Socket (Input)	AX_IN	BOOL adapter that provides the input data and events.
adapter::types::unidirectional::AB	Plug (Output)	AB_OUT	BYTE adapter that outputs the converted data and events.

Functionality

Internally, the function block consists only of connections that route the event and data from the socket `AX_IN` to the plug `AB_OUT`.

- The event `E1` is passed on without delay or modification.
- The data value `D1` is converted from type `BOOL` to type `BYTE`. The Boolean value `TRUE` is converted to the byte `1`, and `FALSE` is converted to the byte `0`. This conversion is performed implicitly by the 4diac runtime environment.

Since this is a composite function block, all internal connections are defined at design time; there is no dynamic logic or state machines.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Composite Type**: The function block is a pure network function block without its own algorithms. Conversion is achieved by connecting two different adapter types.
- **Type Conversion**: The conversion from `BOOL` to `BYTE` is standards-compliant and uses the implicit cast function of the 4diac IDE. This eliminates the need for a separate conversion function block.
- **Versioning**: The function block is version `1.0` dated February 17, 2026, and is intended for use in the package `adapter::conversion::unidirectional`.

State Overview

The function block has no internal states – it operates purely combinatorially. The output values are a direct function of the input values at any given time.

Application Scenarios

- **Connecting Sensors/Actuators**: If a device or controller operates as a BOOL signal, but the target system expects a BYTE adapter format, `AX_TO_AB` can be used as an intermediary.
- **Adapter Bridges**: In heterogeneous automation networks where different adapter protocols are used, the function block (FB) allows for a simple and type-safe connection.
- **Testing and Simulation**: The FB can be used as a simple converter in test environments to translate signals between different logic layers.

Comparison with Similar Function Blocks

Special conversion FBs exist, such as `BOOL_TO_BYTE` or `BYTE_TO_BOOL`, but these are usually implemented as basic function blocks (FBs) with an algorithm. In contrast, `AX_TO_AB` is a composite function block at the adapter level and encapsulates all communication (events and data). It is useful when entire adapter interfaces need to be implemented, not just individual data points.

Conclusion

The composite function block `AX_TO_AB` offers a compact and elegant solution for converting a BOOL adapter to a BYTE adapter. Through implicit type conversion and direct event forwarding, it reduces manual adjustments and increases reusability in modular automation projects.

🌐 Related topic subpages on ms-muc-docs.de

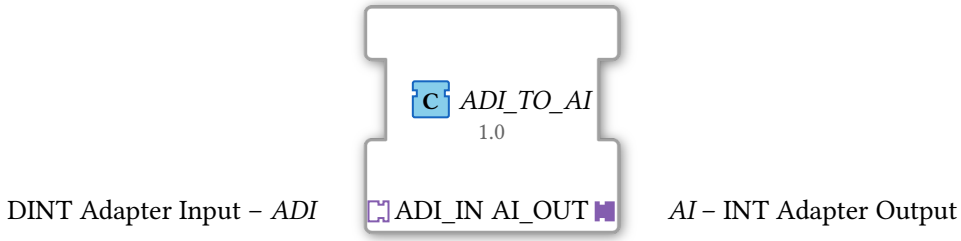
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

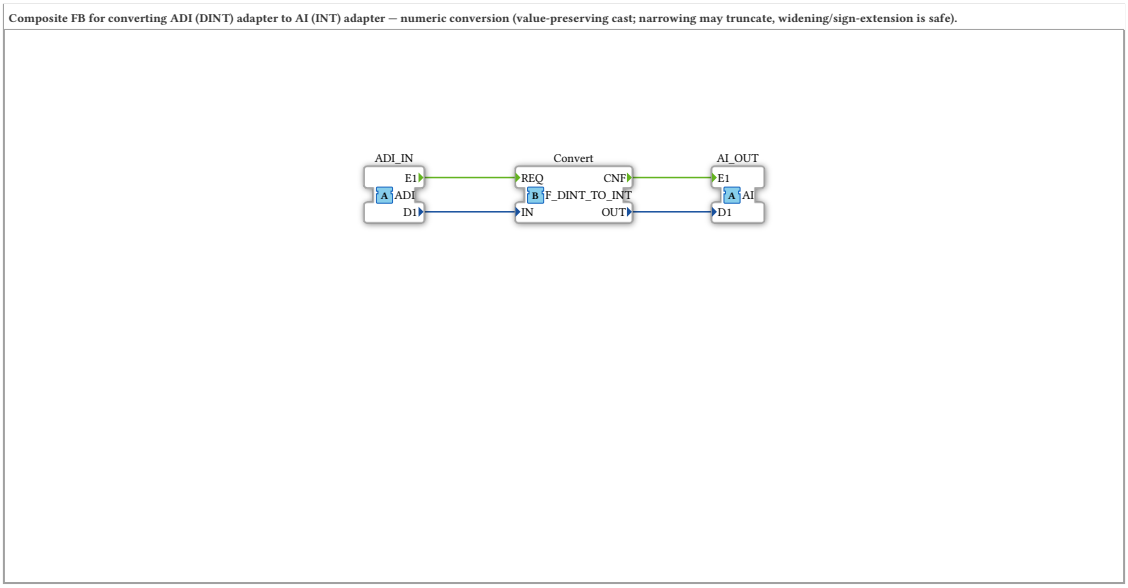
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ADI_TO_AI





• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_TO_AI** is a composite FB that converts a DINT adapter (ADI) into an INT adapter (AI). It serves as a unidirectional conversion interface between two adapter types and internally uses the block `F_DINT_TO_INT` for type conversion of the data values.

Interface Structure

Event Inputs

The FB does not have separate event inputs. Event-driven processing is handled via the **Socket ADI_IN**:

- **ADI_IN.E1** – Event to request the conversion (internally forwarded to `Convert.REQ`).

Event Outputs

The FB has no separate event outputs. Result feedback is provided via the **AI_OUT** plug:

- **AI_OUT.E1** – Event triggered after successful conversion (corresponds to `Convert.CNF`).

Data Inputs

The FB has no separate data inputs. Input data is provided via the **ADI_IN** socket:

- **ADI_IN.D1** – DINT value (32-bit integer) to be converted to an INT value.

Data Outputs

The FB has no separate data outputs. Result data is output via the **AI_OUT** plug:

- **AI_OUT.D1** – INT value (16-bit integer), the result of the conversion.

Adapter

- **Socket**: `ADI_IN` (Type `adapter::types::unidirectional::ADI`) – DINT adapter input.
- **Plug**: `AI_OUT` (Type `adapter::types::unidirectional::AI`) – INT adapter output.

Functionality

This function block connects the adapter interfaces directly to an internal instance of the conversion function block `F_DINT_TO_INT`:

1. An event at `ADI_IN.E1` triggers `Convert.REQ`.
2. The value of `ADI_IN.D1` is passed to `Convert.IN`.
3. The conversion `DINT → INT` is performed.
4. Upon completion, `Convert.CNF` signals the event to `AI_OUT.E1`, and the converted value from `Convert.OUT` is forwarded to `AI_OUT.D1`.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **No Dedicated State Machine**: The function block is a pure composite (network) without ECC. Its behavior is determined solely by the internal connection and the conversion block used.
- **Type Conversion**: The conversion from DINT to INT is performed according to IEC 61131-3 (overflow on value range exceeding is not handled; INT range: −32768 … 32767).
- **Package**: The function block (FB) is organized in the package `adapter::conversion::unidirectional`.

State Overview

The FB does not have an explicit state machine. Its behavior is purely data flow-driven: Each incoming event results in exactly one conversion and one output.

Application Scenarios

- **Integration of DINT-based components** into an environment that only supports INT adapters (e.g., older controllers or standardized interfaces).
- **Type adaptation** in heterogeneous automation systems that use different data widths.

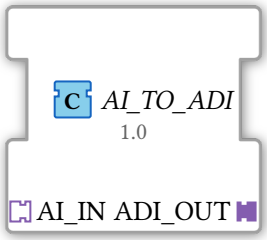
Comparison with Similar Function Blocks

- **AI_TO_ADI**: The reverse conversion (INT ⟶ DINT). While `ADI_TO_AI` reduces the value range (potential data loss), `AI_TO_ADI` extends the value range without information loss.
- **Direct Adapter Coupling**: Without conversion, the adapter types would have to match exactly. `ADI_TO_AI` enables communication between incompatible types.

Conclusion

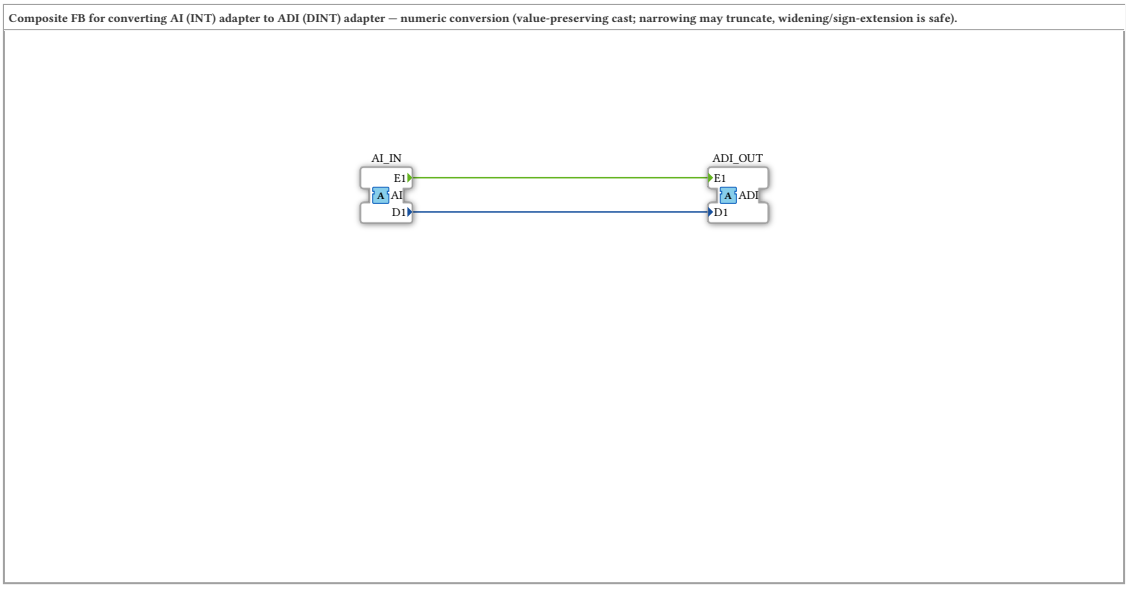
The `ADI_TO_AI` module is a simple yet useful converter that bridges the gap between DINT and INT adapter interfaces. Its composite structure makes it easy to understand and maintain. Using the standardized `F_DINT_TO_INT` ensures correct conversion.

AI_TO_ADI



INT Adapter Input – *AI*

ADI – DINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The AI_TO_ADI function block is a composite function block that converts an AI adapter (integer values, type INT) into an ADI adapter (double integer values, type DINT). It serves as a bridge between components that use different adapter interfaces.

Interface Structure

Event Inputs

The function block does not have its own event inputs. The event is provided via the incoming adapter AI_IN:

- **AI_IN.E1** (Event): Trigger for forwarding the data value and the event to the output.

Event Outputs

The function block does not have its own event outputs. The event is output via the outgoing adapter ADI_OUT:

- **ADI_OUT.E1** (Event): Triggered after successful data transmission.

Data Inputs

The function block (FB) does not have its own data inputs. The data is provided via the incoming adapter AI_IN:

- **AI_IN.D1** (INT): The integer value to be converted.

Data Outputs

The function block (FB) does not have its own data outputs. The data is output via the outgoing adapter ADI_OUT:

- **ADI_OUT.D1** (DINT): The converted double-integer value.

Adapters

Name	Direction	Type	Description
AI_IN	Socket	adapter::types::unidirectional::AI	Input adapter for integer values (INT)
ADI_OUT	Plug	adapter::types::unidirectional::ADI	Output adapter for double integer values (DINT)

Functionality

This function block operates as a simple forwarder: As soon as the event `E1` is present at the incoming adapter `AI_IN`, the current value of `AI_IN.D1` is read and passed unchanged to `ADI_OUT.D1`. Subsequently, the event `E1` is triggered at the outgoing adapter `ADI_OUT`. The type conversion from `INT` to `DINT` is performed automatically via the adapter binding; no explicit conversion logic is required.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Implicit Type Conversion:** The function block (FB) performs the conversion from 16-bit integer (INT) to 32-bit double integer (DINT) without data loss, as the target data type is larger.
- **Composite Implementation:** The function block is implemented as a composite FB, meaning it contains an internal network that directly connects the two adapters. It can be used in control projects like a primitive FB.
- **Unidirectional Communication:** Both adapters are designed to be unidirectional – data flows only from input to output.
- **No Latency or Buffering:** Data forwarding is event-driven and occurs without any additional delay.
- **No Latency or Buffering:**

State Overview

The AI_TO_ADI has no internal state machine (ECC). It behaves passively and immediately forwards every incoming event and its associated data. There are no internal states, wait times, or value storage.

Application Scenarios

- **Adapter Bridge:** If a sensor module provides an AI adapter (INT), but a subsequent processing module expects an ADI adapter (DINT), the AI_TO_ADI can be used as an intermediary.
- **System Integration:** In heterogeneous automation systems where different data types are used for adapter interfaces, this function block enables easy adaptation without modifying the original components.
- **Prototyping:** Rapid coupling of function blocks from different libraries based on different adapter types.

Comparison with Similar Function Blocks

Function Block	Type Conversion	Interface	Use Case
AI_TO_ADI	INT -> DINT	Adapter (Socket/Plug)	Bridge between adapter-based components
INT_TO_DINT	INT -> DINT	Direct inputs/outputs (e.g., DI1, DO1)	Simple data conversion without adapters
CONV_ANY_TO_ANY	Generic	Flexible inputs/outputs	Universal conversion (often more complex)

While function blocks like `INT_TO_DINT` offer direct data conversion with their own inputs and outputs, AI_TO_ADI seamlessly integrates the conversion into an adapter-based architecture, thus facilitating the reuse of existing adapter definitions.

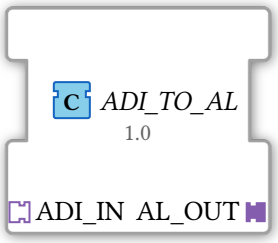
Conclusion

The AI_TO_ADI is a simple yet valuable function block for converting integer to double-integer adapter interfaces. Its composite structure and implicit type conversion allow it to be integrated into adapter-based control systems without additional programming logic, simplifying the integration of components with different data widths.

Numeric Index (unidirectional)

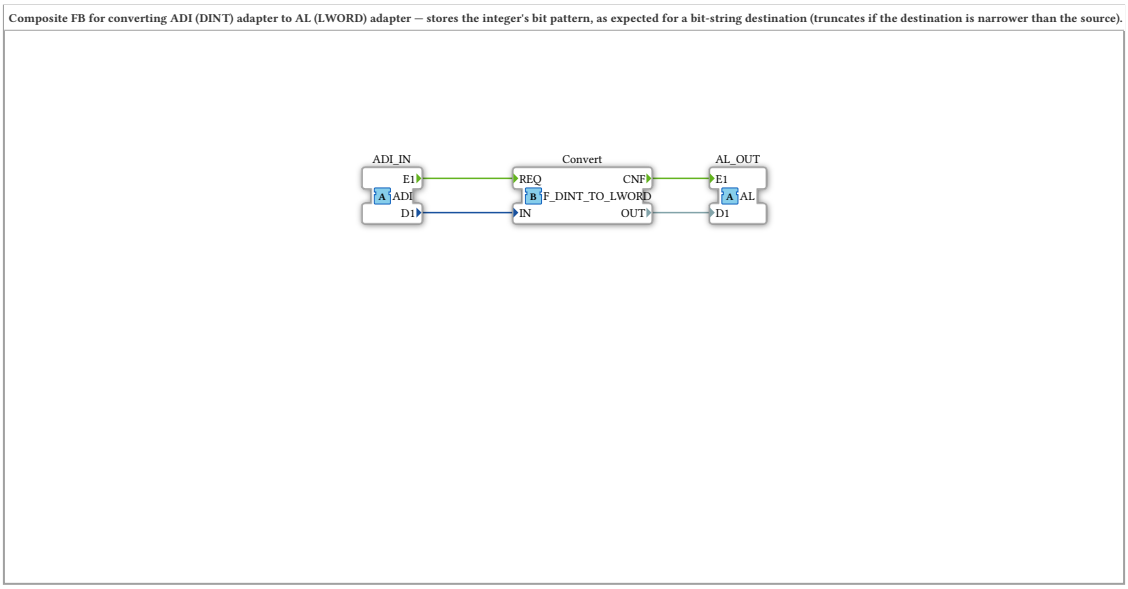
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

ADI_TO_AL



DINT Adapter Input – *ADI*

AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `ADI_TO_AL` serves as a composite block for converting an ADI adapter (DINT) to an AL adapter (LWORD). It encapsulates a type conversion from a 32-bit integer value to a 64-bit word and provides the connection via unidirectional adapters. The block is ideal for systems where different data type representations between components need to be bridged.

Interface Structure

The block has no direct event or data inputs/outputs. All communication takes place via two adapter interfaces:

Event Inputs

- **No direct event inputs**

The event is received via the ADI adapter `ADI_IN.E1`.

Event Outputs

- **No direct event outputs**

The outgoing event is sent via the AL adapter `AL_OUT.E1`.

Data Inputs

- **No direct data inputs**

The input data is provided via the ADI adapter `ADI_IN.D1` (data type: DINT).

Data Outputs

- **No direct data outputs**

The output data is provided via the AL adapter `AL_OUT.D1` (data type: LWORD).

Adapter

Name	Type	Direction	Comment
ADI_IN	adapter::types::unidirectional::ADI	Socket (Input)	Returns a DINT value and controls processing via event E1.
AL_OUT	adapter::types::unidirectional::AL	Plug (Output)	Outputs the converted LWORD value and acknowledges it via event E1.

Functionality

The function block implements a simple type conversion chain:

1. An event received via the ADI adapter ADI_IN (E1) starts the conversion.
2. The internal function block F_DINT_TO_LWORD converts the incoming DINT value (ADI_IN.D1) into an LWORD value.
3. After successful conversion, the result is output as an LWORD via the AL adapter (AL_OUT.D1), and an event (E1) is sent.

Communication between the adapters and the conversion function block is asynchronous, using event and data connections within the composite block's network.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Unidirectional Adapters** – The block uses exclusively unidirectional adapters (socket for input, plug for output), which define a clear data flow direction.
- **Internal Conversion** – The actual type conversion is handled by the standard block F_DINT_TO_LWORD from the IEC 61131 library, eliminating the need for a custom implementation.
- **License** – The block is licensed under the Eclipse Public License 2.0 (EPL-2.0) and can be freely used, modified, and distributed.
- **No States** – The composite block has no state logic of its own; all control is achieved through event chaining.

State Overview

The block does not have its own state machine. Its behavior is purely reactive: An incoming event at the ADI adapter is immediately reflected in the output of the AL adapter. Internal delays are undefined.

Application Scenarios

- **Data Bridge Between Different Bus Systems** – When a sensor or controller delivers a DINT value, but a subsequent component expects an LWORD value (e.g., with 64-bit actuators or fieldbus interfaces).
- **Middleware Adaptation** – Use in heterogeneous automation systems to harmonize adapter types without modifying the original components.
- **Test and Simulation Environments** – Easy conversion of data types for debugging or backup connections.

Comparison with Similar Function Blocks

- **Single Conversion Function Blocks** – Function blocks like DINT_TO_LWORD (direct conversion without an adapter) offer a more compact interface but require their own event and data lines.

Middleware Adaptation – Use in heterogeneous automation systems to harmonize adapter types without modifying the original components. - **Universal Adapter Converters** – Generic converters exist that can convert any data type via adapters. ADI_TO_AL is specifically optimized for the DINT→LWORD combination and is therefore easier to understand and maintain.

- **Bidirectional Adapters** – Unlike bidirectional variants, this module is limited to one data flow direction, which reduces the potential for errors.

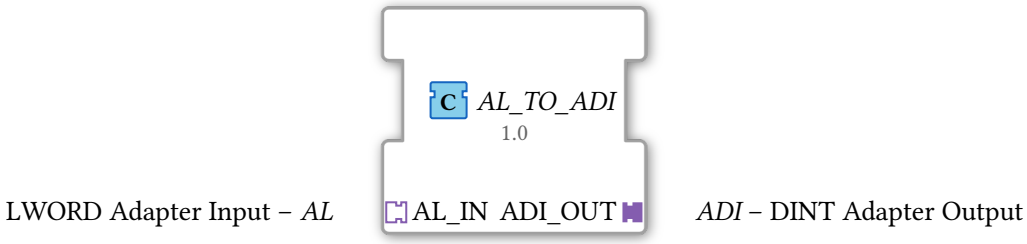
Conclusion

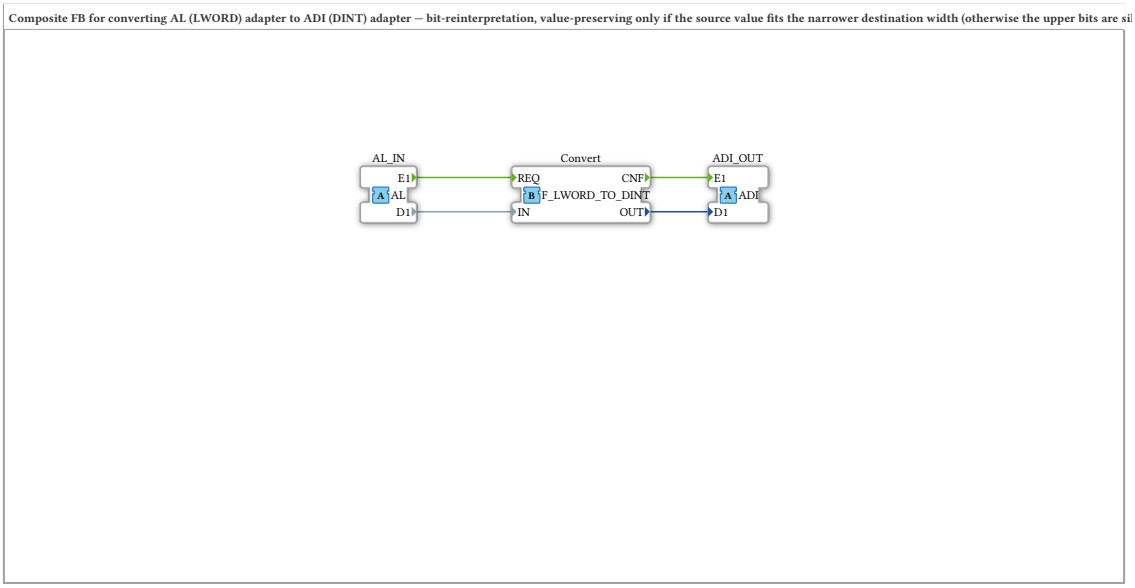
The `ADI_TO_AL` module offers a clean, streamlined solution for converting DINT to LWORD using unidirectional adapters. It can be easily integrated into existing 4diac projects, requires no custom state modeling, and benefits from the reusability of the internal standard conversion module. This makes it ideal for all applications where fast and reliable data conversion between different adapter types is required.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AL_TO_ADI





• – ○ • – ○ • – ○ •

The composite function block `AL_TO_ADI` serves to standardize the conversion of a unidirectional LWORD adapter (`AL`) into a unidirectional DINT adapter (`ADI`). It enables the seamless integration of components that work with different data types in a 4diac adapter landscape.

Since the function block is implemented as a composite, it does not have its own event or data interfaces outside of the adapters. All inputs and outputs are handled via the integrated adapters.

No dedicated event inputs. The event triggering the conversion is received via socket `AL_IN` (specifically via event `E1`).

No dedicated event outputs. The confirmation event after the conversion is complete is sent via plug `ADI_OUT` (specifically via event `E1`).

No dedicated data inputs. The LWORD value to be converted is provided via socket `AL_IN` (specifically via date `D1`).

No dedicated data outputs. The converted DINT value is output via plug `ADI_OUT` (specifically, via the date `D1`).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- **AL_IN (Socket)**

Type: `adapter::types::unidirectional::AL`
Includes a unidirectional LWORD adapter. Provides the trigger event (`E1`) and the LWORD input value (`D1`).

- **ADI_OUT (Plug)**

Type: `adapter::types::unidirectional::ADI`
Provides a unidirectional DINT adapter. Returns the confirmation event (`E1`) and the converted DINT output value (`D1`).

Functionality

The FB is a composite function block that internally uses the IEC 61131 standard function `F_LWORD_TO_DINT`. The process is as follows:

1. An event at socket `AL_IN.E1` activates the internal converter (`Convert.REQ`).
2. The converter reads the LWORD value from `AL_IN.D1`, converts it to a DINT, and places the result at `Convert.OUT`.
3. After successful conversion, an event from `Convert.CNF` is forwarded to the plug `ADI_OUT.E1`.
4. Simultaneously, the converted DINT value from `Convert.OUT` is transferred to `ADI_OUT.D1`.

This ensures a complete and timely conversion from an LWORD adapter to a DINT adapter.

- The function block uses the IEC 61131-3 function `F_LWORD_TO_DINT` – a standardized and reliable type conversion.
- This is a purely **unidirectional** conversion: Data and events flow only from the socket (`AL_IN`) to the plug (`ADI_OUT`).
- The function block (FB) is modeled as a composite, which allows for a clear separation of conversion logic and adapter interfaces and promotes reuse.
- When converting from LWORD (64-bit) to DINT (32-bit), values outside the DINT range can lead to overflows or limits – this is defined by the IEC function.

The FB does not have its own state machine (ECC) because it is implemented as a composite without sequential logic. Its functionality is implemented purely through event-driven chaining of the internal elements. A one-time conversion pass occurs with each incoming event at `AL_IN`.

- **Sensor Connection:** A sensor provides LWORD data (e.g., a time signal or a large counter variable) that must be further processed as DINT in a controller.
- **Library Extension:** Deployment as part of a collection of unidirectional type converter adapters for various IEC 61131-3 data types.

Similar converter adapters exist in 4diac, for example, for `AL_TO_UDINT`, `BYTE_TO_DINT`, or `LWORD_TO_REAL`. The `AL_TO_ADI` is specifically designed for the LWORD ⟶ DINT combination. Unlike simple function blocks that only convert data, it offers full adapter coupling with event control. Compared to bidirectional adapters, it is fixed to the data flow direction from AL to ADI. Potential data loss due to the reduction in bit width (64 ⟶ 32 bits) must be considered during the selection process.

The Composite Function Block `AL_TO_ADI` provides a clean and standardized solution for converting a unidirectional LWORD adapter to a DINT adapter. By utilizing the IEC 61131 function and embedding it within the 4diac adapter concept, it significantly simplifies the integration of heterogeneous system

components. It is easy to maintain, reusable, and follows established design patterns in industrial automation.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).

State Overview

Application Scenarios

Comparison with Similar Function Blocks

Conclusion

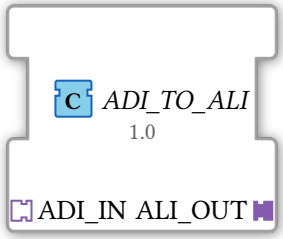
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

Numeric Index (unidirectional)

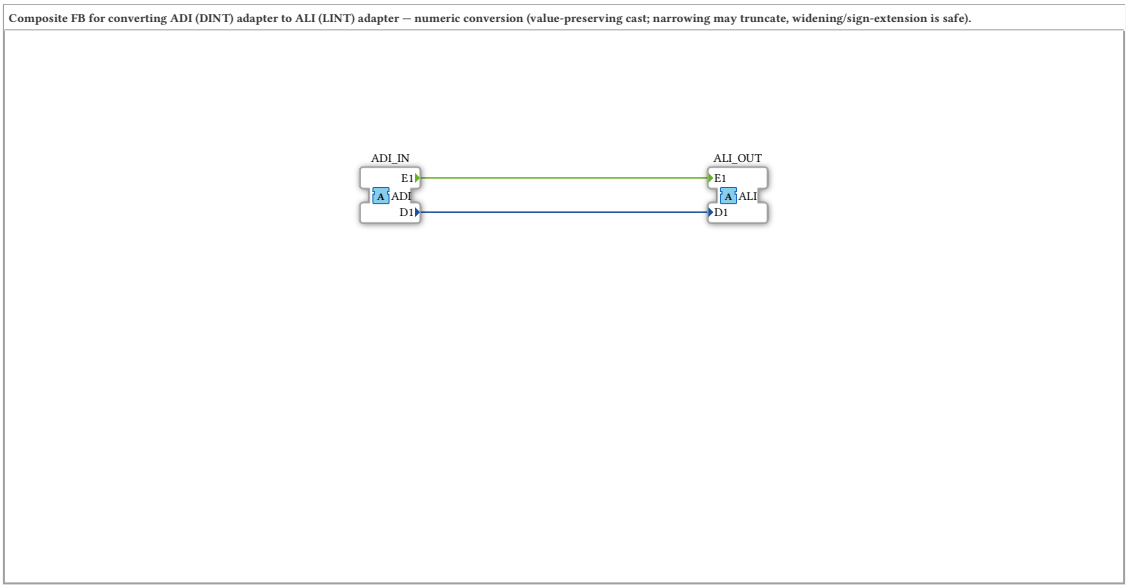
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ADI_TO_ALI

DINT Adapter Input – *ADI*



ALI – LINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **ADI_TO_ALI** function block is a composite block that converts a unidirectional ADI adapter (DINT data type) into a unidirectional ALI adapter (LINT data type). It adapts the data width from 32 bits to 64 bits, thus enabling the seamless integration of components of different integer types in automation systems according to IEC 61499-2. The block is available under the Eclipse Public License 2.0.

Interface Structure

Event Inputs
None

Event Outputs
None

Data Inputs
None

Data Outputs
None

Adapters
- **ADI_IN** (Socket) – Type: `adapter::types::unidirectional::ADI`

Accepts a DINT value (32-bit) and its associated event.

- **ALI_OUT** (Plug) – Type: `adapter::types::unidirectional::ALI`

Outputs the converted LINT value (64-bit) and the forwarded event.

Functionality
This function block implements direct passthrough:

- The event `E1` from the ADI_IN socket is passed to the ALI_OUT plug without delay.
- Simultaneously, the data value `D1` (DINT) is transmitted; this involves an implicit type conversion from 32-bit integer (DINT) to 64-bit integer (LINT).
- No intermediate storage or processing logic takes place – every incoming event-data combination is immediately forwarded to the output side.

Technical Features
- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **No internal state** – The function block is entirely combinatorial and has no memory.
- **Automatic type conversion** via the data connection; the value range of DINT is fully covered by LINT, so no information loss occurs.
- **Adapter-based communication** according to IEC 61499-2 enables loose coupling and reusability.
- The function block is implemented as a composite, meaning its behavior is defined solely by the internal function block network (one event and one data connection).

State Overview
This function block does not have a state machine. Its functionality is limited to the direct forwarding of adapter signals.

Application Scenarios
- **System migration** from 32-bit to 64-bit data processing, e.g., during the modernization of controllers.
- **Integration of older components** that provide DINT values into new modules that require LINT interfaces.
- **Adapter bridge** in heterogeneous networks to adapt different integer types between function blocks.

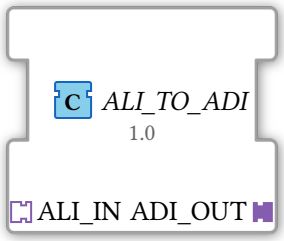
Comparison with Similar Function Blocks
Simple type converters (e.g., `DINT_TO_LINT`) usually operate at the data level without event forwarding. The `ADI_TO_ALI` block, on the other hand, encapsulates events and data in an adapter interface and is therefore particularly suitable for modular, event-driven architectures. Compared to manual conversion over the network, it reduces complexity and increases system readability.

Conclusion

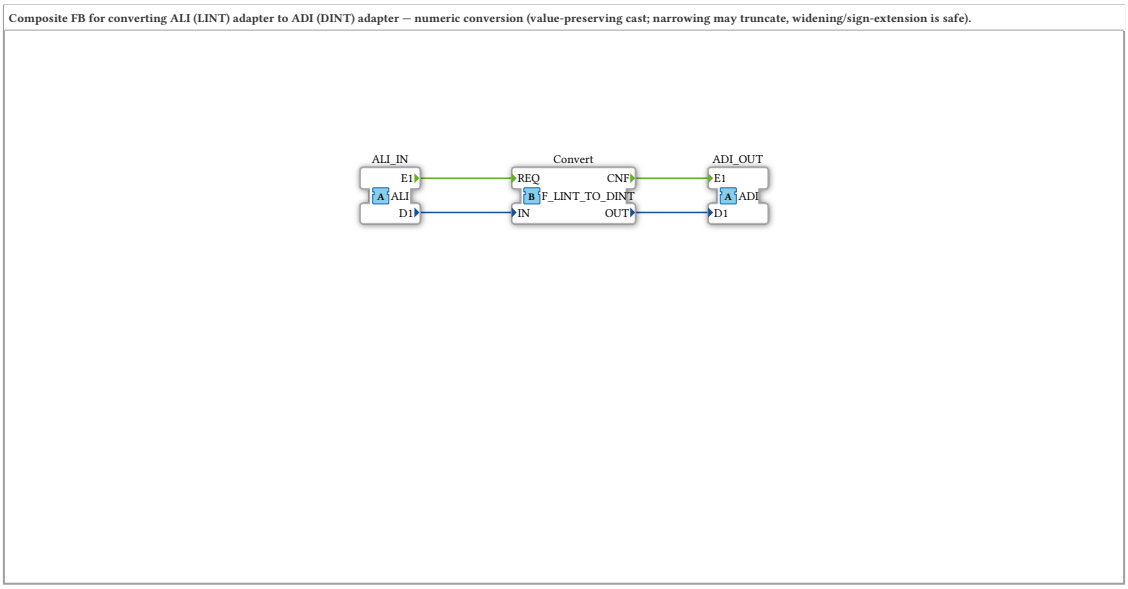
ADI_TO_ALI` is a minimal yet effective block for adapting the data width between DINT and LINT adapters. Its simple pass-through logic and automatic type conversion make it a useful tool for bridging integer types in IEC 61499 systems. Compliance with the standard and licensing under EPL 2.0 ensure open usability and interoperability.

ALI_TO_ADI

LINT Adapter Input – *ALI*



ADI – DINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

This composite function block converts an **ALI** adapter type (LINT, 64-bit integer) to an **ADI** adapter type (DINT, 32-bit integer). It enables seamless integration of components using different adapter formats by encapsulating data conversion and synchronizing it via event control.

Interface Structure

Event Inputs

No direct event inputs are available. Control is handled via the incoming adapter **ALI_IN**.

Event Outputs

No direct event outputs are available. Output is clocked via the outgoing adapter **ADI_OUT**.

Data Inputs

No direct data inputs are available. The input data is provided via the **ALI_IN** adapter (of type ALI).

Data Outputs

No direct data outputs are available. The converted data is output via the **ADI_OUT** adapter (of type ADI).

Adapters

Name	Type	Direction	Description
ALI_IN	`adapter::types::unidirectional::ALI`	Socket (Input)	Returns the LINT value and an associated event (E1).
ADI_OUT	`adapter::types::unidirectional::ADI`	Plug (Output)	Outputs the converted DINT value and an acknowledgment event (E1).

Functionality

The component contains the instance `Convert` of type `iec61131::conversion::F_LINT_TO_DINT`.
The event and data connections are configured as follows:

- The event `ALI_IN.E1` triggers the conversion via `Convert.REQ`.
- The data value `ALI_IN.D1` is passed to `Convert.IN`.
- After successful conversion, `Convert.CNF` signals the event `ADI_OUT.E1`.
- The result value `Convert.OUT` is forwarded to `ADI_OUT.D1`.

Thus, with each event at the input adapter, the LINT value is converted to a DINT value and synchronously provided via the output adapter.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block**: The function block encapsulates the conversion in a reusable adapter-to-adapter converter.
- **Data Type Conversion**: A 64-bit to 32-bit integer conversion takes place. Values outside the DINT range (-2^{31} … $2^{31}-1$) result in an overflow or a limit (depending on the implementation).
- **Event Synchronization**: The conversion is strictly timed through the event chain; a new value is only processed after a previous event.

State Overview

The block does not have its own state machine (ECC). Its entire logic is implemented by the internal function block `F_LINT_TO_DINT`, which operates as a simple conversion block without state behavior.

Application Scenarios

- Connecting a component that provides a 64-bit counter or timestamp (LINT) to a system that only supports 32-bit DINT adapters.
- Migrating legacy systems with 32-bit arithmetic to newer 64-bit sensors without changing the entire adapter landscape.
- Intermediate layer in a hierarchy where different integer widths meet.

Comparison with Similar Blocks

Separate conversion blocks such as `F_LINT_TO_DINT` exist for direct data conversions. The `ALI_TO_ADI` function block stands out because it fully integrates the conversion into the adapter interface. Instead of multiple discrete function blocks and manual adapter wiring, all the logic is combined in a reusable composite block that automatically handles the event and data streams.

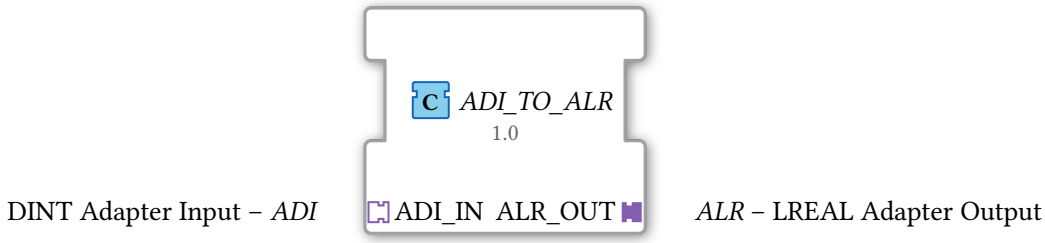
Conclusion

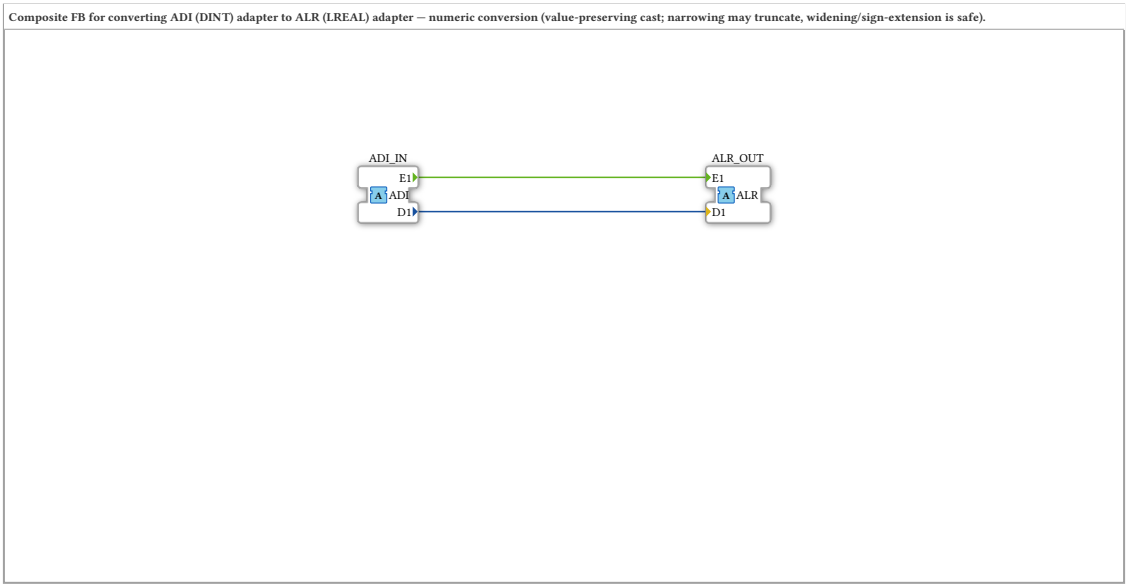
The `ALI_TO_ADI` function block offers a clean, encapsulated solution for converting LINT to DINT adapters. By using a composite block, it reduces application complexity and ensures standardized, event-driven conversion between different integer widths in the IEC 61499 environment.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ADI_TO_ALR





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ADI_TO_ALR** is a composite function block (FB) that converts an ADI adapter (data type DINT) into an ALR adapter (data type LREAL). It serves as a unidirectional converter between two standardized adapter interfaces and enables the seamless integration of integer values into systems that require floating-point numbers.

Interface Structure

Event Inputs

- No dedicated event inputs are available.

Event forwarding is handled via the **ADI_IN** adapter (see Adapter section).

Event Outputs

- No dedicated event outputs are available.

The event is output via the **ALR_OUT** adapter (see Adapter section).

Data Inputs

- No dedicated data inputs are available.

The value to be converted is provided via the **ADI_IN** adapter (see Adapter section).

Data Outputs

- No dedicated data outputs are available.

The converted value is output via the **ALR_OUT** adapter (see Adapter section).

Adapters

Name	Type	Direction	Description
ADI_IN	`adapter::types::unidirectional::ADI`	Socket	Input: DINT value and event
ALR_OUT	`adapter::types::unidirectional::ALR`	Plug	Output: LREAL Value and Event

Note: The adapters each imply an event and a data signal (`E1` and `D1`).

Functionality

The module performs direct internal routing:

- **Event:** The incoming event `ADI_IN.E1` is transmitted to the output `ALR_OUT.E1` without delay.
- **Data:** The integer value `ADI_IN.D1` (data type DINT) is converted to the LREAL data type without scaling or conversion and output as `ALR_OUT.D1`. Type conversion is performed implicitly by the runtime environment of the IEC 61499 system.

No further processing (filtering, limiting, delaying) takes place.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block:** The conversion is implemented via a simple wire-through in the FB network – no state machine, no proprietary algorithms.
- **Implicit Type Conversion:** DINT → LREAL is a lossless conversion (LREAL offers 64-bit accuracy).
- **Unidirectional:** Data flows exclusively from the socket to the plug.
- **Reusability:** Encapsulated as an adapter converter, the block can be used in any system that relies on the corresponding adapter interfaces.

State Overview

The block has **no state machine**. It is purely combinatorial (event-driven) and executes the pass-through immediately upon receiving an event.

Application Scenarios

- **Counter Values as Floating-Point Numbers:** An integer counter (e.g., quantities, pulses) is to be processed as an LREAL value in a controller.
- **Interface Adaptation:** System components that already work with ADI adapters can be seamlessly connected to components that expect ALR adapters.
- **Prototypical Development:** Simple conversion without additional logic to quickly link different adapter protocols.

Comparison with Similar Function Blocks

Function Block	Type Conversion	Event Behavior
ADI_TO_ALR	DINT → LREAL	Direct Pass-Through without Delay
`INT_TO_REAL` (4diac)	INT → REAL	Standalone function block with its own event interface
Individual converters	Any number of	Often with state machines or parameterization

This function block is characterized by its **simple, adapter-oriented structure**, which enables direct coupling in adapter networks without having to define separate data and event channels.

Conclusion

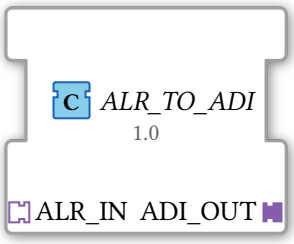
ADI_TO_ALR is a minimal yet useful converter for the unidirectional transfer of DINT values to the LREAL format. It is robust, fast, and requires no configuration. Thanks to its composite structure, it can be easily integrated into existing applications based on adapter interfaces.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

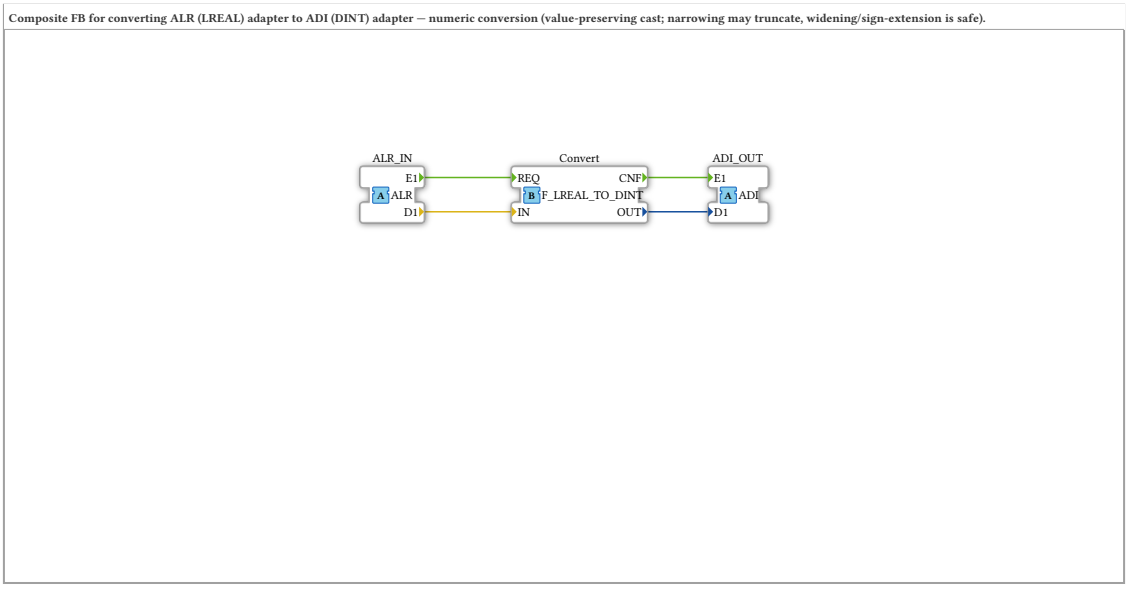
]

ALR_TO_ADI



LREAL Adapter Input – *ALR*

ADI – DINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **ALR_TO_ADI** function block is a composite block that enables conversion between two different adapter types. It translates an **ALR adapter** (which provides LREAL data) into an **ADI adapter** (which provides DINT data). It is used in environments based on the IEC 61499 standard to standardize communication between components with different data type interfaces.

Interface Structure

The function block does not have any independent event or data inputs/outputs at the top level. All communication takes place via the two adapter interfaces.

Event Inputs

The function block does not have direct event inputs. Events are received via the **ALR_IN** socket.

Event Outputs

The FB has no direct event outputs. Events are sent via the **ADI_OUT** plug.

Data Inputs

The FB has no direct data inputs. Data is received via the **ALR_IN** socket.

Data Outputs

The FB has no direct data outputs. Data is sent via the **ADI_OUT** plug.

Adapters

Adapter	Direction	Type	Description
ALR_IN	Socket	adapter::types::unidirectional::ALR	Returns an event (E1) and a data value (D1) of type LREAL.
ADI_OUT	Plug	adapter::types::unidirectional::ADI	Expects an event (E1) and outputs a data value (D1) of type DINT.

Functionality

The function block is implemented as a composite and internally contains a single conversion function block: `iec61131::conversion::F_LREAL_TO_DINT`. The connection is as follows:

1. **Event Forwarding**: An event (E1) arriving at socket **ALR_IN** is directly forwarded to the `REQ` input of the conversion function block.
2. **Data Transfer**: The data value **D1** of the **ALR_IN** adapter is transferred to the `IN` input of the conversion function block.
3. **Conversion**: The function block `F_LREAL_TO_DINT` converts the LREAL value into a DINT value.
4. **Feedback**: The confirmation event (CNF) of the conversion function block triggers the event **E1** at the **ADI_OUT** plug.
5. **Data Output**: The converted DINT value from the output `OUT` of the conversion function block is transferred to the data output **D1** of the **ADI_OUT** adapter.

This results in a synchronized, event-driven conversion: Each input pulse generates an output pulse with the converted data value.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block**: The implementation uses an internal standard function block from the IEC 61131 library, which promotes maintainability and reusability.
- **Unidirectionality**: The adapters are declared as unidirectional, meaning data flows in only one direction (ALR → ADI).
- **Package Structure**: The function block is organized in the package `adapter::conversion::unidirectional`, which allows for clear integration into a library.
- **No State Storage**: The conversion is purely combinatorial with event-driven triggering – no internal state is stored.

State Overview

Since the function block does not have its own state diagram, the state logic is limited to that of the internal conversion function block. There are essentially two phases:

- **Waiting**: The function block waits for an event at the ALR_IN socket.
- **Converting**: After receiving the event, the conversion is performed, and the result is output at the ADI_OUT socket.

There are no delays or error handling – the conversion is instantaneous and always produces a valid DINT value (in case of overflow or invalid values, the conversion function block behaves according to IEC 61131).

Application Scenarios

- **System Integration**: One system component provides data as LREAL (e.g., floating-point measurements), while another requires integer DINT values (e.g., for counter registers or index calculations).
- **Adapter Adaptation**: If one component only supports ALR adapters, but another component expects ADI adapters, this function block can act as an intermediary.
- **Protocol Conversion**: In standalone subsystems connected via adapters, this function block enables clean type conversion without changing the original adapter definition.
-

Comparison with Similar Function Blocks

- **F_LREAL_TO_DINT (Direct Function Block)**: This function block performs the conversion itself but requires its own event and data interfaces and no adapters. `ALR_TO_ADI` encapsulates this function block to make it compatible with adapter-based connections.
- **ALR_TO_DINT (Hypothetical)**: A direct function block with its own input/output interface (event + data) would not use any adapters. This function block is specifically designed for use in an adapter framework.
- **Reverse Conversion (ADI_TO_ALR)**: A function block for the reverse direction (DINT ⟶ LREAL) likely exists, based on the same conversion principle.

Conclusion

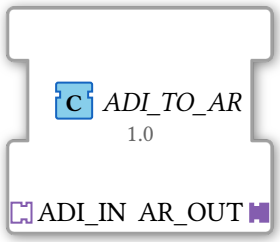
The composite function block `ALR_TO_ADI` offers an elegant solution for mediating between adapter types with different data types. By reusing a standardized conversion block, the implementation remains lean and reliable. The function block is particularly suitable for modular, adapter-based systems where type conflicts between components need to be resolved.

Numeric Index (unidirectional)

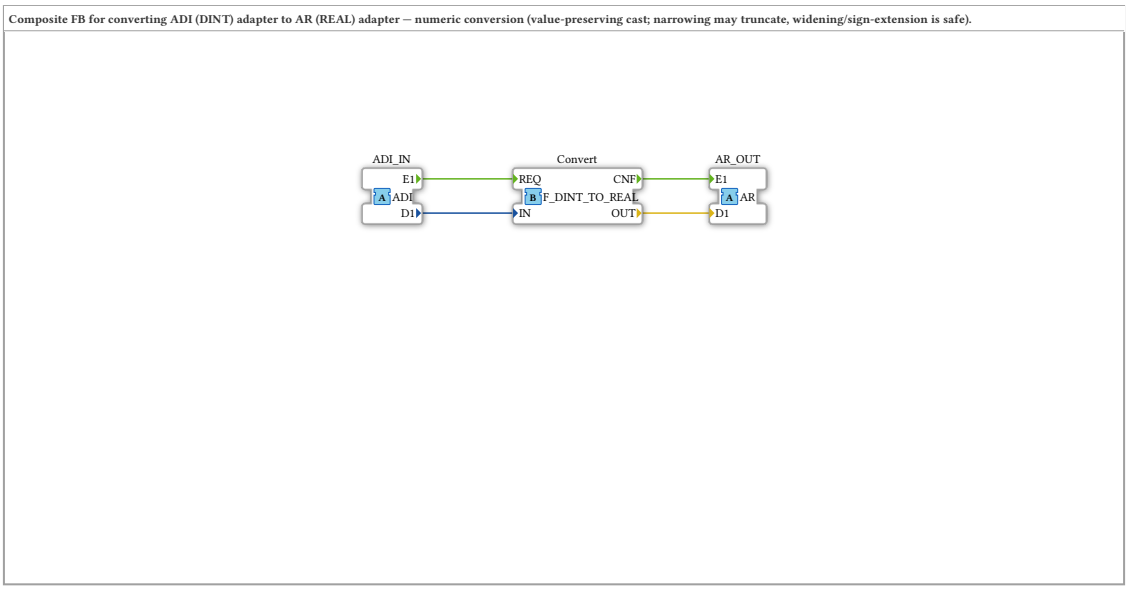
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ADI_TO_AR

DINT Adapter Input – *ADI*



AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **ADI_TO_AR** function block is a composite module that converts a DINT adapter (ADI) into a REAL adapter (AR). It acts as an intermediary between components that communicate via unidirectional adapters, performing a data type conversion from integer (DINT) to floating-point number (REAL).

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **ADI_IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **AR_OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input data value is provided via the **ADI_IN** adapter (data point D1 of type DINT).

Data Outputs

- No directly visible data outputs. The output data value is provided via the **AR_OUT** adapter (data point D1 of type REAL).

Adapters

Adapter	Role	Type	Description
ADI_IN	Socket	adapter::types::unidirectional::ADI	Unidirectional DINT adapter as input interface.
AR_OUT	Plug	adapter::types::unidirectional::AR	Unidirectional REAL adapter as output interface.

Functionality

The function block internally contains a single function block, `F_DINT_TO_REAL`, from the IEC 61131-3 standard library. The sequence control is purely event-driven:

1. An event at the input adapter **ADI_IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **ADI_IN.D1** (DINT) is passed to the input **IN** of the converter.
3. Upon completion of the conversion, the output **CNF** of the converter signals an event.
4. This event is forwarded to the output adapter **AR_OUT.E1**.
5. Simultaneously, the converted REAL value (output **OUT**) is passed to the data point **AR_OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Type Conversion according to IEC 61131-3**: The internal function block `F_DINT_TO_REAL` guarantees standards-compliant conversion of 32-bit integers to floating-point numbers (single precision).
- **Unidirectional Adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported, which simplifies implementation and clearly defines the data flow.
- **Composite Structure**: The function block consists of a network that is completely encapsulated within the function block itself. It is therefore particularly well-suited for modular, reusable interfaces.
- **No States**: There is no state machine – the functionality is purely combinatorial with an event-driven trigger.

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **ADI_IN.E1**
2	Convert the data value from DINT to REAL

3	Output the converted value to **AR_OUT.D1** and event to **AR_OUT.E1**
4	Return to step 1

Application Scenarios

- **Sensor Integration**: A sensor provides integer measurements (e.g., speed, counter reading) via an ADI adapter. This function block is used as a converter for downstream control or visualization systems that expect REAL values.
- **Interface Adaptation**: In heterogeneous control systems where different components use different data types, this function block enables seamless integration without modifying existing adapter definitions.
- **Type-safe data processing**: Explicit conversion avoids runtime errors that can occur with implicit type conversions in IEC 61499.

Comparison with similar function blocks

- **F_DINT_TO_REAL**: A direct call to this standard function block would serve the same purpose, but requires manual configuration of event and data connections. The adapter-based approach simplifies reuse and improves network readability.
- **ADI_TO_AR** in contrast to other conversion function blocks (e.g., REAL_TO_DINT): This function block is specifically designed for unidirectional adapter communication and does not offer any other conversion directions.
- **Composite function blocks without adapters**: Other implementations use direct data and event ports, which limits flexibility when replacing components. The adapter interface allows for simple plug-and-play in predefined architectures.

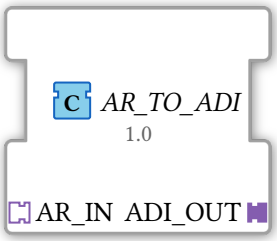
F_DINT_TO_REAL: ** ...

Conclusion

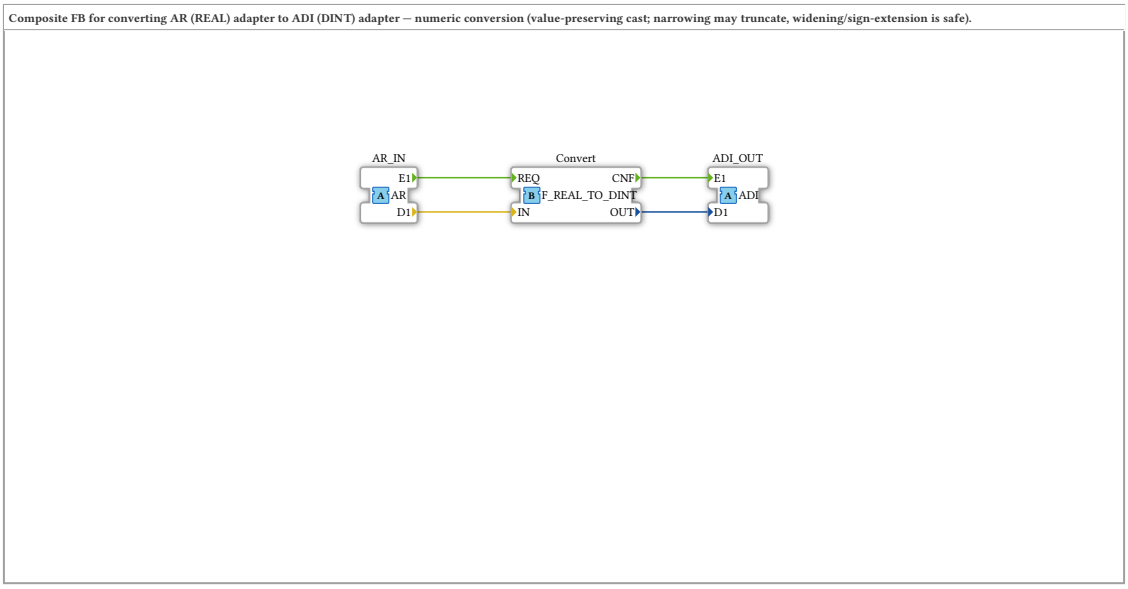
The **ADI_TO_AR** function block represents an elegant and standards-compliant solution for converting DINT values to REAL values, fully embedded in the adapter-based communication structure of IEC 61499. By encapsulating the conversion process in a reusable composite component, it supports the development of modular, type-safe, and well-documented automation systems.

AR_TO_ADI

REAL Adapter Input – *AR*



ADI – DINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block **AR_TO_ADI** is a composite block that converts a REAL adapter (AR) into a DINT adapter (ADI). It is used to mediate between interfaces with different data types, with the conversion occurring in an event-driven manner.

Interface Structure

The block's interface consists exclusively of adapters that bundle both events and data. There are no separate inputs or outputs.

Event Inputs

- **AR_IN.E1** – Event input via the AR adapter

Triggers the conversion of the incoming REAL value.

Event Outputs

- **ADI_OUT.E1** – Event output via the ADI adapter

Activated after successful conversion.

Data Inputs

- **AR_IN.D1** (Data type REAL) – The floating-point value to be converted.

Data Outputs

- **ADI_OUT.D1** (Data type DINT) – The converted integer value.

Adapters

Adapter	Name	Direction	Description
AR	AR_IN	Socket (Input)	Unidirectional REAL adapter that provides the input signal.
ADI	ADI_OUT	Plug (Output)	Unidirectional DINT adapter that forwards the converted signal.

Functionality

The function block is implemented as a composite function block. It contains an instance of the IEC 61131 standard function block `F_REAL_TO_DINT`.

As soon as an event arrives at the event input **AR_IN.E1**, the REAL value present at **AR_IN.D1** is passed to the internal converter. After the conversion is complete, the result is forwarded to **ADI_OUT.D1** and the event **ADI_OUT.E1** is triggered.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- The function block uses the data type converter `F_REAL_TO_DINT` from the IEC 61131 library.
- It is a composite function block that does not have its own state logic; instead, the functionality is implemented through the internal function block.
- The interface is purely adapter-based, allowing for easy integration into existing adapter networks.

State Overview

Since the FB does not contain a state machine, the following simple sequence applies:

1. **Waiting** – The FB is ready to receive an event on **AR_IN.E1**.
2. **Conversion** – As soon as an event arrives, the REAL value is converted to DINT.
3. **Output** – After the conversion is complete, the result is output and the output event is sent. The FB then returns to the wait state.

Application Scenarios

- Adapting a sensor that delivers REAL values to a controller that expects DINT values.
- Bridging the gap between two adapter-based components with different data types in an IEC 61499 system.
- Use in data preprocessing chains where floating-point numbers need to be converted to integers.

Comparison with similar function blocks

- **`F_REAL_TO_DINT`** – This function block performs the actual conversion but expects separate event and data connections. **AR_TO_ADI** encapsulates this functionality in an adapter-based interface, thus simplifying integration into adapter networks.
- **Other adapter converters** – Similar function blocks exist for other data type combinations (e.g., AR_TO_AINT, ADI_TO_AR). They all follow the same pattern: using an internal conversion function block embedded in a uniform adapter shell.

Conclusion

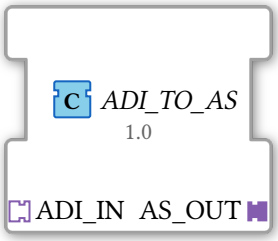
AR_TO_ADI is a compact, reusable composite function block that enables clean conversion from REAL to DINT adapters. Thanks to its simple, event-driven structure and the use of standardized conversion modules, it is ideally suited for adapter-based designs in industrial control systems.

Numeric Index (unidirectional)

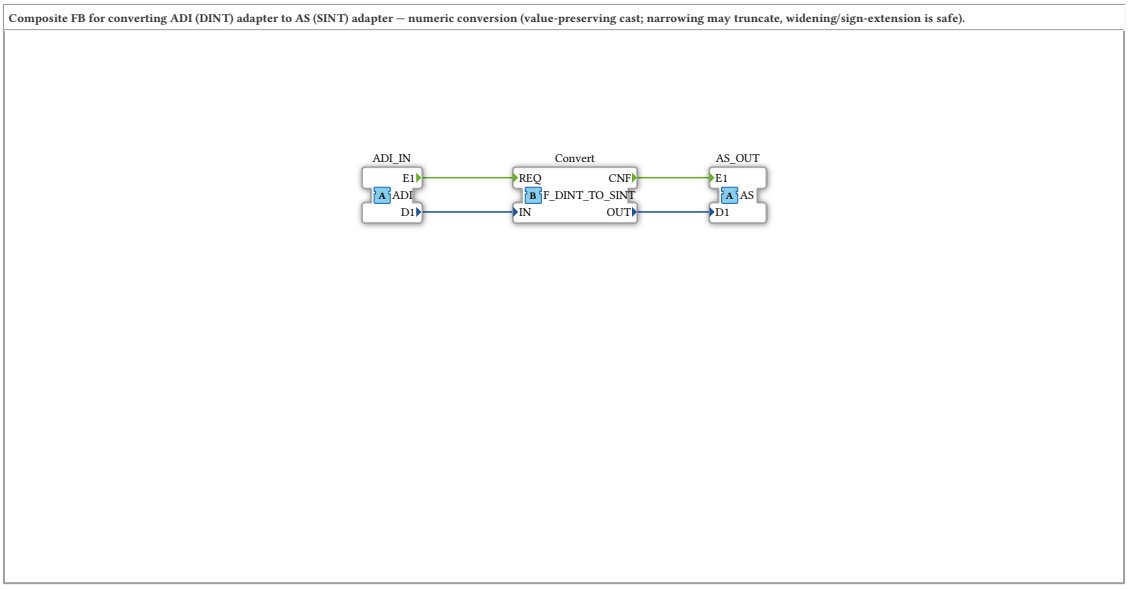
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ADI_TO_AS

DINT Adapter Input – *ADI*



AS – SINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The composite function block **ADI_TO_AS** is used to convert a **DINT adapter signal** (type `ADI`) into a **SINT adapter signal** (type `AS`). It is used when data and event exchange is required via unidirectional adapters and a data type conversion from DINT (32-bit signed integer) to SINT (8-bit signed integer) is necessary.

Interface Structure

Event Inputs

No separate event inputs – event control is handled via the integrated **ADI-IN adapter** (socket).

Event Outputs

No separate event outputs – event forwarding is handled via the integrated **AS-OUT adapter** (plug).

Data Inputs

No separate data inputs – the data value is provided via the **ADI-IN adapter**.

Data Outputs

No separate data outputs – the converted value is output via the **AS-OUT adapter**.

Adapters

Adapter	Direction	Type	Comment
ADI_IN	Socket (Input)	`adapter::types::unidirectional::ADI`	DINT Adapter Input
AS_OUT	Plug (Output)	`adapter::types::unidirectional::AS`	SINT Adapter Output

The **ADI-IN Adapter** provides an event interface `E1` and a data interface `D1` (type DINT). The **AS-OUT Adapter** provides `E1` and `D1` (type SINT).

Functionality

Inside the composite module, a single function block `Convert` of type `iec61131::conversion::F_DINT_TO_SINT` is used. The signal connections are as follows:

1. An incoming event at the **ADI-IN adapter** (`E1`) triggers the **Convert FB** via its event input `REQ`.
2. Simultaneously, the DINT value is transferred from the adapter's data port `ADI_IN.D1` to the **Convert input** `IN`.
3. After the conversion is complete, **Convert** generates an event at its output `CNF`.
4. This event is forwarded directly to the **AS-OUT adapter** (`E1`).

5. The converted SINT value is available at the **Convert output** `OUT` and is transferred to the adapter data port `AS_OUT.D1`.

This enables complete, event-driven conversion from DINT to SINT via the adapter interfaces.

Technical Details

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **License and Origin:** This module is subject to the **Eclipse Public License 2.0** and is from **HR Agrartechnik GmbH** (Version 1.0, Author: Franz Höpfinger, Date: 2026-02-17).
- **Dependency:** It uses the library module `iec61131::conversion::F_DINT_TO_SINT` from the IEC 61131 conversion package.
- **Composite Structure:** No separate states or algorithms are defined – all logic is executed by the internal network.
- **Packet Assignment:** The function block is organized in the package `adapter::conversion::unidirectional`.

State Overview

As a composite function block, **ADI_TO_AS** does not have its own state machine. Its behavior results solely from the execution of the embedded function block `F_DINT_TO_SINT` and the hard-wired connections.

Application Scenarios

- **Adapter Type Conversion:** When a component (e.g., a sensor or actuator) communicates via a unidirectional DINT adapter, but the target system expects a SINT adapter.
- **Data Range Reduction:** Applications where a 32-bit value needs to be converted to an 8-bit value, e.g.,... B. for controlling simple peripherals with limited resolution.
- **Interface adaptation:** Integration into a toolchain based on 4diac's unidirectional adapter concept, requiring a clean separation of event and data flows.

Comparison with similar modules

Other adapter conversion modules exist, such as `ADI_TO_AI` (DINT → INT) or `ADI_TO_ADI` (identical passthrough). The **ADI_TO_AS** module differs in its specific target type conversion to SINT. All share a uniform adapter interface and event-driven processing via an integrated conversion module.

Conclusion

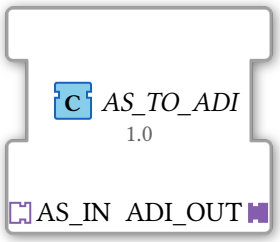
The **ADI_TO_AS** is a compact, pre-built composite module that reliably and easily converts a DINT adapter signal to a SINT adapter signal. It is ideal for all applications requiring clean type conversion within the 4diac adapter concept, without the need to manually implement separate conversion logic. Thanks to its EPL 2.0 license, it can be freely used and adapted in your own projects.

Related topic subpages on ms-muc-docs.de

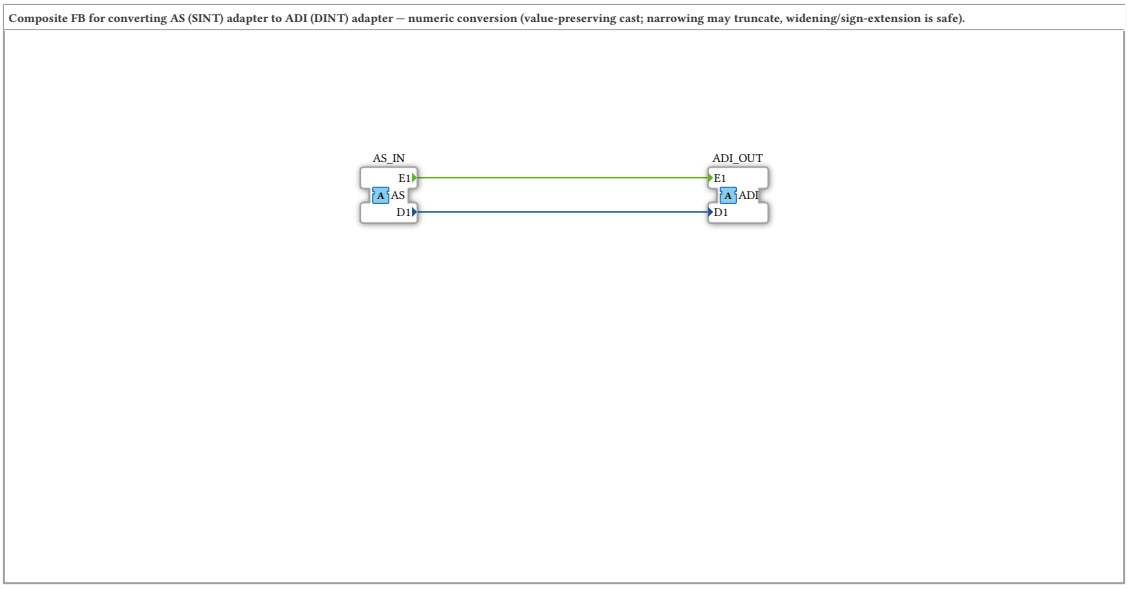
- Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AS_TO_ADI



SINT Adapter Input – *AS* *ADI* – DINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The **AS_TO_ADI** function block converts a SINT adapter (AS) into a DINT adapter (ADI). It is a composite function block that establishes a unidirectional, type-converting connection between the two different data adapters.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Type	Direction	Name	Description
Socket	Input	AS_IN	Adapter type `AS` (SINT, 8-bit signed integer) – receives the SINT signal.
Plug	Output	ADI_OUT	Adapter type `ADI` (DINT, 32-bit signed integer) – outputs the converted DINT signal.

Functionality

The module internally links the event and data lines of the two adapters directly:

- The incoming event `E1` of socket `AS_IN` is transferred unchanged to the output plug `ADI_OUT` at its event `E1`.
- The data value `D1` from the SINT adapter (`AS_IN.D1`) is forwarded to data port `D1` of the DINT adapter (`ADI_OUT.D1`).

Since the data types are different (SINT ⟶ DINT), implicit value preservation occurs: The SINT value (value range −128 … 127) is copied into a signed 32-bit value (sign extension). The function block itself does not execute its own conversion logic but relies on the type conversion defined by the adapter interfaces.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Function Block**: The function block does not have its own behavior model (ECC) but only an internal function block network that connects the adapters.
- **Type Conversion**: The conversion from SINT to DINT is handled system-wide by the runtime environment (type compatibility for data connections).
- **Package**: The function block is located in the package `adapter::conversion::unidirectional`.
- **Fully Predefined**: The internal wiring is fixed; the adapters must each have an event port and a data port (`E1`, `D1`).

State Overview

Since this is a composite function block without its own state machine, there are no states or steps. The function block operates purely data-flow synchronously – every incoming event immediately triggers the forwarding.

Application Scenarios

- **Sensor Connection**: A SINT adapter (e.g., from an 8-bit sensor) is to be connected to a controller that expects a DINT adapter as its interface.
- **System Integration**: In systems where different accuracy levels (8-bit vs. 32-bit) are used, this function block enables simple, standardized conversion.
- **Protocol Adaptation**: As part of an adapter cascade to switch from a simple data interface standard to a more powerful one.

Comparison with Similar Function Blocks

- **AS_TO_UDI** – Converts SINT to UINT (unsigned 16-bit integer), different value range interpretation.
- **ADI_TO_AS** – Backward conversion from DINT to SINT (with potential overflow risks).
- **DIRECT_LINK** – Pure passthrough without type conversion, usable only with identical adapter types.

Unlike universal converters (e.g., using `MATH` function blocks), `AS_TO_ADI` offers a specialized, compact solution without additional parameterization.

Conclusion

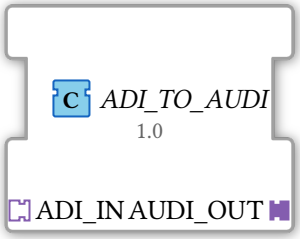
The **AS_TO_ADI** function block enables fast and reliable type conversion between SINT and DINT adapters. Its simple, composite-based structure makes it ideal for standardized adapter conversions in automation technology where unidirectional 8-bit to 32-bit conversion is required.

Numeric Index (unidirectional)

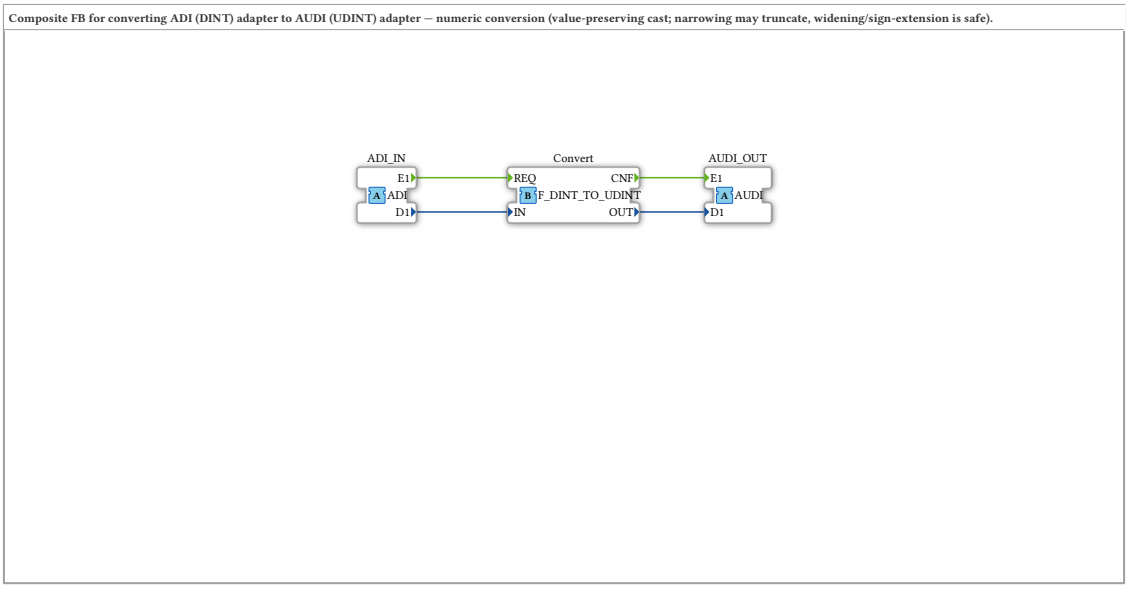
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ADI_TO_AUDI

DINT Adapter Input – *ADI*



AUDI – UDINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block ADI_TO_AUDI converts an ADI (DINT) adapter to an AUDI (UDINT) adapter. It is a composite function block (FB) that uses an internal conversion block to convert the data types from signed 32-bit integer (DINT) to unsigned 32-bit integer (UDINT). The block enables the seamless integration of components that use different adapter types.

Interface Structure

Event Inputs

The function block does not have direct event inputs. Events are provided via the incoming adapter **ADI_IN**. The ADI adapter defines an event output E1, which is connected to the event input REQ of the internal conversion block.

Label	Data Type	Description
(via ADI_IN.E1)	EVENT	Trigger for Conversion

Event Outputs

This function block does not have direct event outputs. Events are passed on via the outgoing adapter **AUDI_OUT**. The internal conversion function block signals completion via CNF, which is connected to the event input E1 of the AUDI adapter.

Name	Data Type	Description
(via AUDI_OUT.E1)	EVENT	Confirmation of Completed Conversion

Data Inputs

This function block has no direct data inputs. The DINT value to be converted is provided via the incoming adapter **ADI_IN** (output D1).

Name	Data Type	Description
(via ADI_IN.D1)	DINT	The value in DINT format to be converted to UDINT

Data Outputs

This function block has no direct data outputs. The converted UDINT value is output via the outgoing adapter **AUDI_OUT** (input D1).

Name	Data Type	Description
(via AUDI_OUT.D1)	UDINT	The result of the conversion as UDINT

Adapters

The function block provides two adapter interfaces:

Name	Direction	Type	Description
ADI_IN	Socket	adi (unidirectional)	Input adapter – provides DINT data and a trigger event
AUDI_OUT	Plug	audi (unidirectional)	Output adapter – outputs converted UDINT data and an acknowledgment event

The adapters are unidirectional, meaning they transmit data and events in only one direction.

Functionality

The function block operates as a pure wrapper (composite) and contains a single internal function block of type `iec61131::conversion::F_DINT_TO_UDINT`. As soon as an event arrives at input adapter E1 on the ADI_IN input, the internal function block is triggered via its REQ input. This block reads the DINT value present at ADI_IN.D1, converts it to a UDINT value, and outputs the result to its output OUT. Simultaneously, it generates an acknowledgment event at CNF, which is forwarded via the output adapter AUDI_OUT.E1. The converted value is then available at AUDI_OUT.D1.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- The function block is implemented as a composite function block and does not contain any logic of its own. All functionality is provided by the internal conversion block.
- The adapters are of type "unidirectional," meaning that data and event flows are directed.
- The function block is part of the adapter conversion library and facilitates the connection of components with different adapter data types.
- The data type conversion from DINT to UDINT is typed; An overflow for negative values must be considered (e.g., –1 becomes 4294967295).

State Overview

The function block does not have its own state machine. Its behavior results solely from the interaction of the internal block `F_DINT_TO_UDINT` and the event connections. The internal block has simple state logic: In its idle state, it waits for a REQ event, then performs the conversion and sends CNF. There are no other states.

Application Scenarios

- **Adapter Replacement**: When an application needs to replace a DINT-based adapter (ADI) with a UDINT-based adapter (AUDI) without changing the surrounding logic.
- **System Integration**: Connecting components that use different integer types (e.g., a sensor provides DINT, an actuator expects UDINT).
- **Protocol Adaptation**: In heterogeneous control systems where adapter types are not directly compatible.

Comparison with Similar Function Blocks

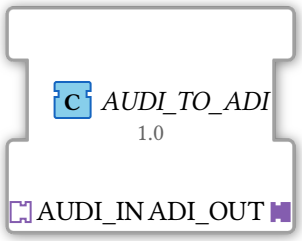
- **ADI_TO_AUDI** (this function block): Converts DINT adapters to UDINT adapters.
- **AUDI_TO_ADI**: A hypothetical counterpart that would convert UDINT to DINT (does not exist in the given XML, but is conceivable).
- **Direct Conversion with F_DINT_TO_UDINT**: Instead of using the adapter, the conversion function block could be used directly, but the adapter structure would then have to be manually recreated. ADI_TO_AUDI encapsulates this adaptation and provides a clean interface.

Conclusion

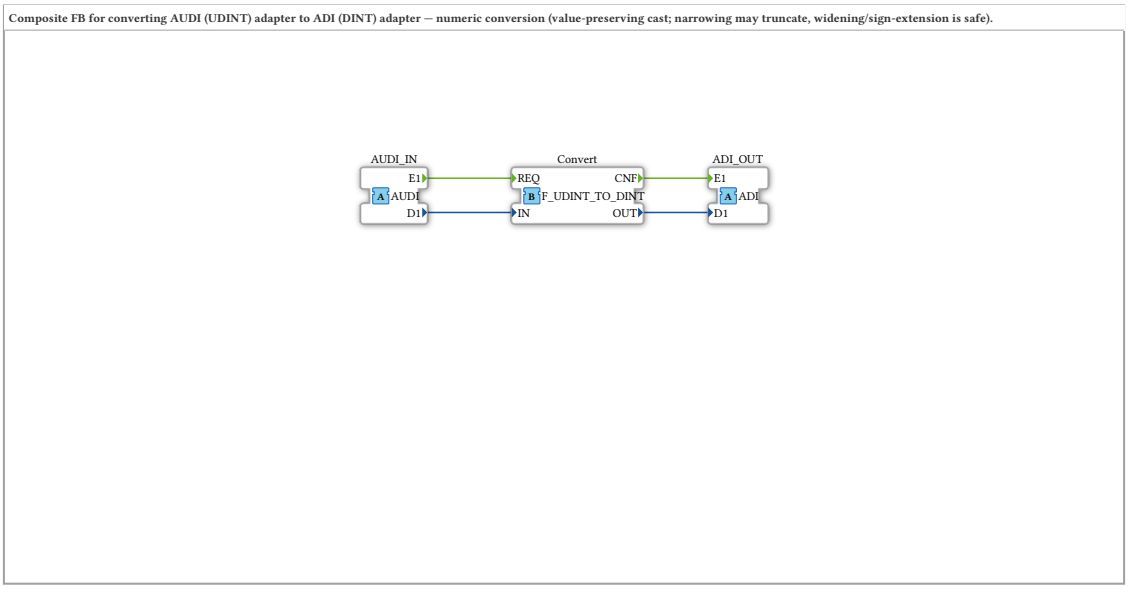
The ADI_TO_AUDI function block provides a simple and effective solution for converting DINT to UDINT adapters. As a composite function block, it utilizes a proven standard conversion block and offers a clear, unidirectional interface via its adapters. It is particularly well-suited for system integration and adapter adaptation in IEC 61499-based automation solutions.

AUDI_TO_ADI

UDINT Adapter Input – *AUDI*



ADI – DINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AUDI_TO_ADI** function block is a composite block that converts a unidirectional AUDI adapter (UDINT) into an ADI adapter (DINT). It serves as a bridge between components that use different data types via adapter interfaces. The conversion is performed by the inner block `iec61131::conversion::F_UDINT_TO_DINT`.

Interface Structure

Event Inputs

Name	Type	Comment
AUDI_IN.E1	EVENT	Event input via the AUDI adapter; triggers the conversion

Event Outputs

Name	Type	Comment
ADI_OUT.E1	EVENT	Event output via the ADI adapter; signals completion of conversion

Data Inputs

Name	Type	Comment
AUDI_IN.D1	UDINT	Unsigned 32-bit value converted to a signed DINT

Data Outputs

Name	Type	Comment
ADI_OUT.D1	DINT	Converted signed 32-bit value

Adapter

Name	Type	Direction	Comment
AUDI_IN	adapter::types::unidirectional::AUDI	Socket (Input)	UDINT Adapter Interface
ADI_OUT	adapter::types::unidirectional::ADI	Plug (Output)	DINT Adapter Interface

Functionality

The function block receives a UDINT value via socket `AUDI_IN` along with an event at input `E1`. This event triggers the conversion of the incoming UDINT value to a DINT value. The conversion is performed internally by the function block `F_UDINT_TO_DINT`. After successful conversion, the converted DINT value is provided via the plug `ADI_OUT`, and an event is sent at the output `E1`.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- The block is a composite function block and does not contain its own algorithm – all logic is delegated to the embedded `F_UDINT_TO_DINT`.
- The conversion follows the IEC 61131-3 definition of `UDINT_TO_DINT`, meaning that values outside the DINT range (−2147483648 … 2147483647) are clamped to the nearest valid value.
- Event forwarding is strictly causal: the output event is only activated after the conversion is complete.

State Overview

The function block has no internal states of its own. Its behavior is purely reactive:

An input event is always followed by an output event – provided the data is valid. A dedicated state machine does not exist.

Application Scenarios

- Connecting a UDINT-providing sensor (e.g., counter or position value) to an actuator that expects a DINT adapter.
- Integrating PLC components that use different adapter data types without requiring the user to manually program the conversion.
- A unified adapter interface in a control architecture that works partly with signed and partly with unsigned values.

Comparison with Similar Function Blocks

- **DINT_TO_AUDI**: Performs the reverse conversion (DINT → UDINT).
- **F_UDINT_TO_DINT**: Direct use of the conversion function block without adapter encapsulation.

This function block encapsulates the data type conversion in an adapter context, thus simplifying the interconnection of components that rely on adapter interfaces.

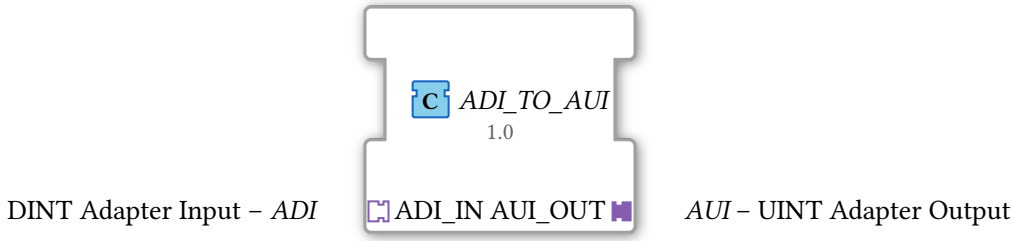
Conclusion

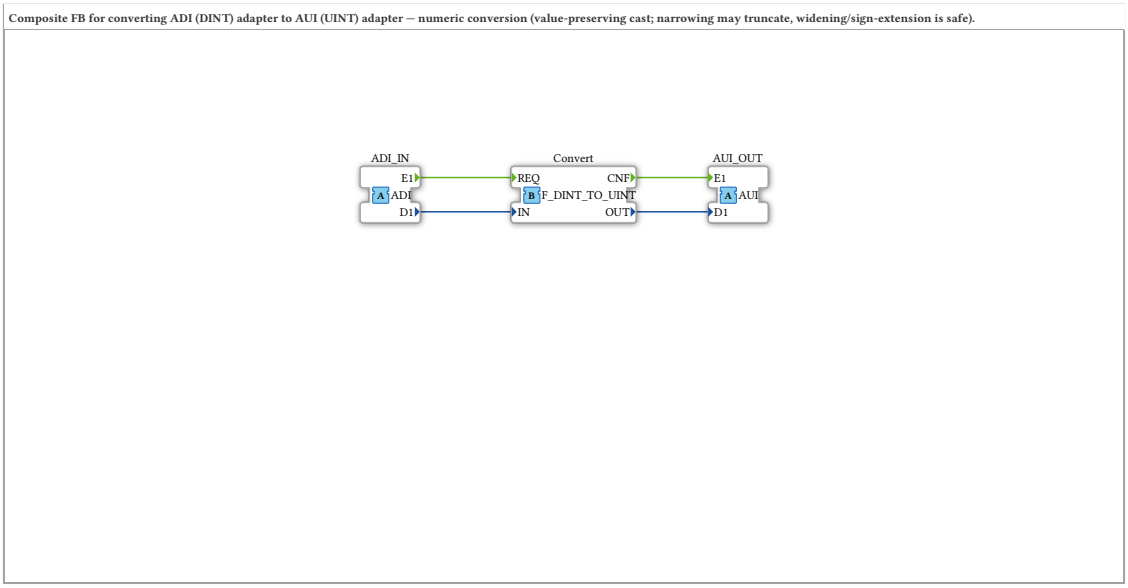
AUDI_TO_ADI offers a clean, compact solution for converting UDINT adapter data to DINT adapter data. The use of a standardized conversion function block ensures reliable and easily traceable functionality. The composite approach reduces complexity in the higher-level system and facilitates the reusability of the adapter logic.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ADI_TO_AUI





• – ○ • – ○ • – ○ •

Introduction

The **ADI_TO_AUI** function block is a composite function block that implements interface conversion between the ADI adapter (DINT, 32-bit integer) and the AUI adapter (UINT, 16-bit integer). It enables the unidirectional transfer of a value from a DINT adapter to a UINT adapter using a standardized IEC 61131 conversion block.

Interface Structure

Event Inputs

None – event control is handled exclusively via the adapters.

Event Outputs

None.

Data Inputs

None – all data is provided via the socket adapter.

Data Outputs

None – all data is output via the plug adapter.

Adapters

Type	Name	Direction	Description

Socket	ADI_IN	Input	ADI adapter (DINT) – provides the value to be converted and an event
Plug	AUI_OUT	Output	AUI adapter (UINT) – provides the converted value and the associated event

Functionality

The function block does not have its own input/output events or top-level data points. All control and data transmission is handled via the two adapters:

1. **Event Forwarding**:

The event E1 received at socket ADI_IN is forwarded directly to the internal block F_DINT_TO_UINT (type: iec61131::conversion) to trigger a conversion.

1. **Data Path**:

The DINT value provided via ADI_IN.D1 is passed to input IN of F_DINT_TO_UINT. After successful conversion, the result (UINT) is available at output OUT and is assigned to the data output D1 of the plug adapter AUI_OUT.

1. **Feedback**:

The internal function block signals the end of the conversion with the event CNF. This event is transferred to the output event E1 of the plug adapter AUI_OUT – synchronously with the provided data value.

Thus, the function block implements a **unidirectional, event-driven** transfer of a DINT value to a UINT value via the adapter interfaces.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Function Block**: The function block fully encapsulates the conversion logic and offers a clean, reusable interface.
- **Core Function Block Used**: iec61131::conversion::F_DINT_TO_UINT – a standardized IEC 61131 function for converting DINT to UINT (value range adjustment: DINT to UINT is considered without a sign; negative values are converted accordingly).
- **Adapter Types**: The adapters ADI and AUI are defined as unidirectional adapters (adapter::types::unidirectional). They each provide an event pair (E1/E1) and a data point (D1 in the corresponding data type).
- **Package Structure**: The function block is organized in the package adapter::conversion::unidirectional.
- **License**: The function block is subject to the Eclipse Public License 2.0 (EPL-2.0). Copyright belongs to HR Agrartechnik GmbH.

State Overview

As a composite function block, **ADI_TO_AUI** does not have its own internal state machine. The state is determined exclusively by the embedded function block F_DINT_TO_UINT, which performs the conversion after receiving the REQ event and completes with CNF. The function block is therefore **stateless** in the sense of a pure function block (no stored states outside the core function block).

Application Scenarios

- **Interface adaptation** between system components that use different data types for analog or digital values (e.g., sensors with DINT output and actuators with UINT input).
- **Data consolidation** in heterogeneous IEC 61499 networks where adapters of different data types interact.
- **Prototypes and Migration**: Easy integration of function blocks that originally work with DINT into an environment that expects UINT – without manual conversion at each connection step.

Comparison with Similar Function Blocks

Directly comparable adapter conversion blocks exist for other data types (e.g., BOOL_TO_UINT, INT_TO_DINT). The difference with **ADI_TO_AUI** lies in the specific combination of adapter interfaces

(ADI ↔ AUI). Unlike simple conversion function blocks (`F_DINT_TO_UINT`), this composite function block encapsulates both the adapter connection and the event and data synchronization in a reusable module.

An alternative design could use a generic conversion block with configurable adapter types; however, the dedicated solution offers clearer documentation and easier handling within the 4diac IDE.

Conclusion

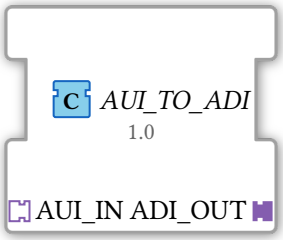
The **ADI_TO_AUI** function block provides a clean, maintainable, and standardized way to convert DINT data via an ADI adapter to a UINT signal via an AUI adapter. By using an IEC 61131 core block, the conversion is standards-compliant and reliable. The composite approach increases reusability and simplifies engineering in distributed automation systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

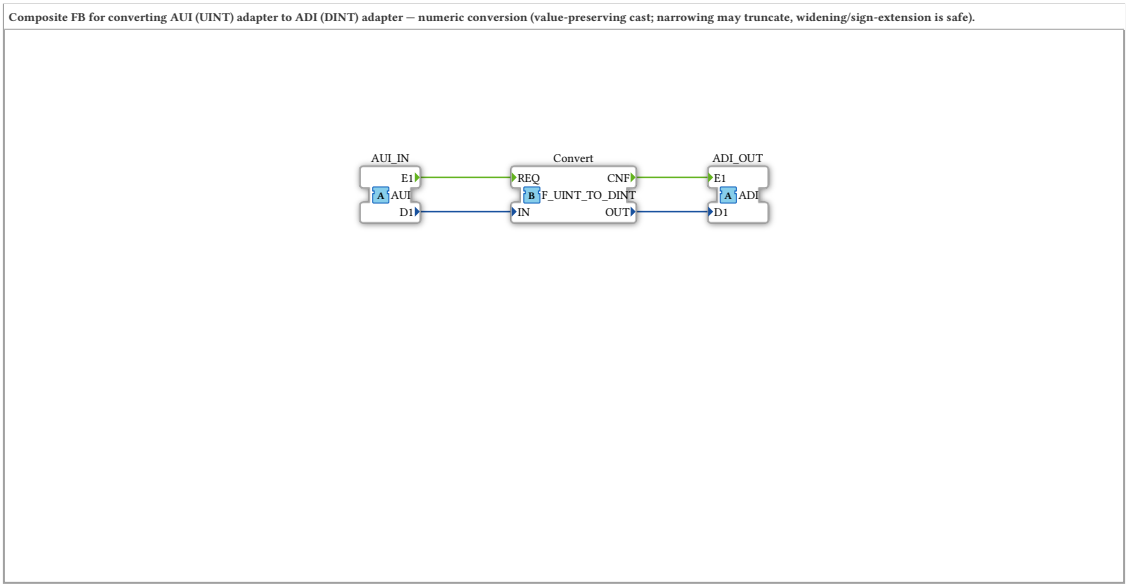
]

AUI_TO_ADI



UINT Adapter Input – *AUI*

ADI – DINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The AUI_TO_ADI function block is a composite module that converts a unidirectional AUI adapter (data type UINT) into a unidirectional ADI adapter (data type DINT). It serves as a standardized interface between components that use different data types for analog input or output values.

Interface Structure

Event Inputs

None. Event control is handled via the connected AUI adapter (socket AUI_IN).

Event Outputs

None. Event output is handled via the ADI adapter (plug ADI_OUT).

Data Inputs

None. Data input is handled via the connected AUI adapter.

Data Outputs

None. Data output is handled via the ADI adapter.

Adapter

Name	Direction	Type	Description
AUI_IN	Socket (Input)	`adapter::types::unidirectional::AUI`	UINT adapter for receiving input data and events.
ADI_OUT	Plug (Output)	`adapter::types::unidirectional::ADI`	DINT adapter for sending the converted data and events.

Functionality

The function block receives an event and a UINT value via the AUI_IN socket. This event triggers the internal conversion block `F_UINT_TO_DINT`, which converts the value from type UINT to type DINT. After successful conversion, an event is output via the ADI_OUT plug, along with the converted DINT value. The conversion is performed according to the IEC 61131 standard and is lossless, as the entire value range of UINT (0…65535) can be mapped to DINT.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- The function block (FB) is implemented as a **composite block**, meaning its functionality is based on an integrated IEC 61131 block (`F_UINT_TO_DINT`).
- Communication occurs exclusively via **unidirectional adapters** – no separate event or data inputs/outputs are required at the top level.
- The internal converter operates **event-driven**: Every incoming event at the AUI_IN adapter triggers a conversion.

State Overview

The FB does not have its own state machine. The state logic is fully implemented by the internal function block `F_UINT_TO_DINT`, which performs a one-time conversion for each REQ event and acknowledges it with CNF.

Application Scenarios

- **Interface Adaptation**: A sensor provides measured values via a UINT-based AUI adapter, but the downstream logic requires DINT values (e.g., for arithmetic operations or visualization).
- **System Integration**: In heterogeneous automation systems where different components use different adapter types, the function block can be used as an intermediary layer.
- **Migration**: Existing systems with UINT data paths are to be gradually converted to DINT.

Comparison with Similar Function Blocks

- **F_UINT_TO_DINT**: Direct IEC 61131 converter, but without adapter encapsulation. Requires separate event and data lines.
- **AUI_TO_ABOOL**: Analog converter for binary outputs (UINT ⟶ BOOL), not for numeric types.
- **ATO_TO_ADI**: Converts ATO adapters (DWORD) to ADI, different data width.

This function block stands out due to its complete adapter integration and unidirectional design, which facilitates its use in adapter-based architectures.

Conclusion

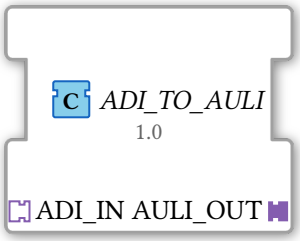
The `AUI_TO_ADI` function block offers a clean, encapsulated solution for type conversion from UINT to DINT via standardized adapters. It reduces wiring effort and increases reusability in modular automation solutions. The use of a proven IEC 61131 function block ensures the correctness of the conversion.

Numeric Index (unidirectional)

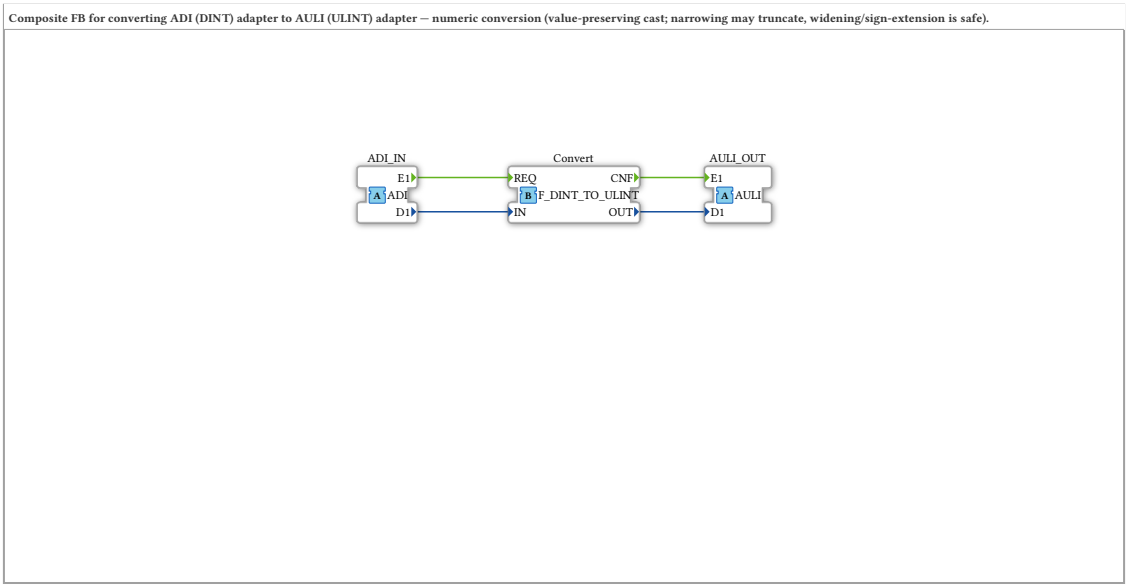
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

ADI_TO_AULI

DINT Adapter Input – *ADI*



AULI – ULINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **ADI_TO_AULI** function block is a composite function block (FB) according to IEC 61499. It converts a DINT value (signed 32-bit integer) received via an ADI adapter (input interface) into a ULINT value (unsigned 64-bit integer) output via an AULI adapter (output interface). Internally, the standardized conversion block `F_DINT_TO_ULINT` from the IEC 61131 library is used. This function block simplifies the integration of adapters and type conversion in automation solutions.

Interface Structure

Event Inputs

Name	Type	Description
ADI_IN.E1	Event (implicit)	Trigger for conversion; provided by the ADI adapter.

Event Outputs

Name	Type	Description
AULI_OUT.E1	Event (implicit)	Signals completion of conversion; passed to the AULI adapter.

Data Inputs

Name	Type	Description
ADI_IN.D1	DINT	Input value to be converted to ULINT.

Data Outputs

Name	Type	Description
AULI_OUT.D1	ULINT	Converted output value (unsigned).

Adapter

Name	Type (Interface)	Direction	Description
ADI_IN	ADI (Socket)	Input	Receives the DINT value via a unidirectional ADI adapter.
AULI_OUT	AULI (Plug)	Output	Provides the converted ULINT value via a unidirectional AULI adapter.

Functionality

The function block waits for an event at the input adapter `ADI_IN.E1`. As soon as this event arrives, the current data value `ADI_IN.D1` (type DINT) is passed to the internal conversion function block `F_DINT_TO_ULINT`. After successful conversion, the result (type ULINT) is forwarded to the output `AULI_OUT.D1`, and an event is triggered at the output `AULI_OUT.E1`. The entire logic is purely combinational – there are no delays or states other than the signal processing of the internal module.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-Based Interface:** The module uses only adapter plugs and sockets for input and output. This allows for clean encapsulation and reuse in modular control architectures.
- **Unidirectional Adapters:** Both the ADI and AULI adapters are declared as unidirectional and transmit only in one direction (input or output).
- **Internal Conversion:** The function block `F_DINT_TO_ULINT`, responsible for the actual conversion, is taken from the package `iec61131::conversion` and provides a standardized, sequence-invariant function.
- **Namespace:** The composite function block belongs to the package `adapter::conversion::unidirectional`, which clearly indicates its role as an adapter wrapper.

State Overview

As a composite function block, `ADI_TO_AULI` does not have its own state machine. Its behavior is entirely determined by the internal linking of event and data flows between the socket `ADI_IN`, the conversion function block, and the plug `AULI_OUT`. The state logic of the internal function block `F_DINT_TO_ULINT` is transparent to its operation.

Application Scenarios

- **Transition between control systems:** When a system with DINT signals (e.g., PLC data) needs to communicate with another system that expects ULINT values (e.g., visualization or long-term archiving with 64-bit timestamps).
- **Adapter bridge:** In a modular automation environment where adapters for different data types are defined, this function block serves as a simple "translation link" between ADI and AULI interfaces.
- **Prototypes or migration:** Rapid integration of components with different integer representations without application-specific conversion logic.

Comparison with similar function blocks

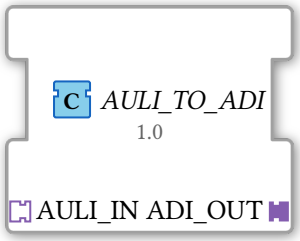
- **Separate `F_DINT_TO_ULINT`:** The internal function block used here is directly responsible for the pure type conversion but requires separate event and data connections. `ADI_TO_AULI` encapsulates these connections and the adapter interface in a single block.
- **Other Adapter Wrappers:** Similar composite blocks exist for other type conversions (e.g., `ADI_TO_AUDI`, `ASI_TO_AULI`), which are structured according to the same pattern. They differ only in the data types involved.
- **Direct Type Conversion Without Adapters:** Some systems offer conversion functions as pure function blocks without an adapter layer. The present block is specifically designed for use with adapters.

Conclusion

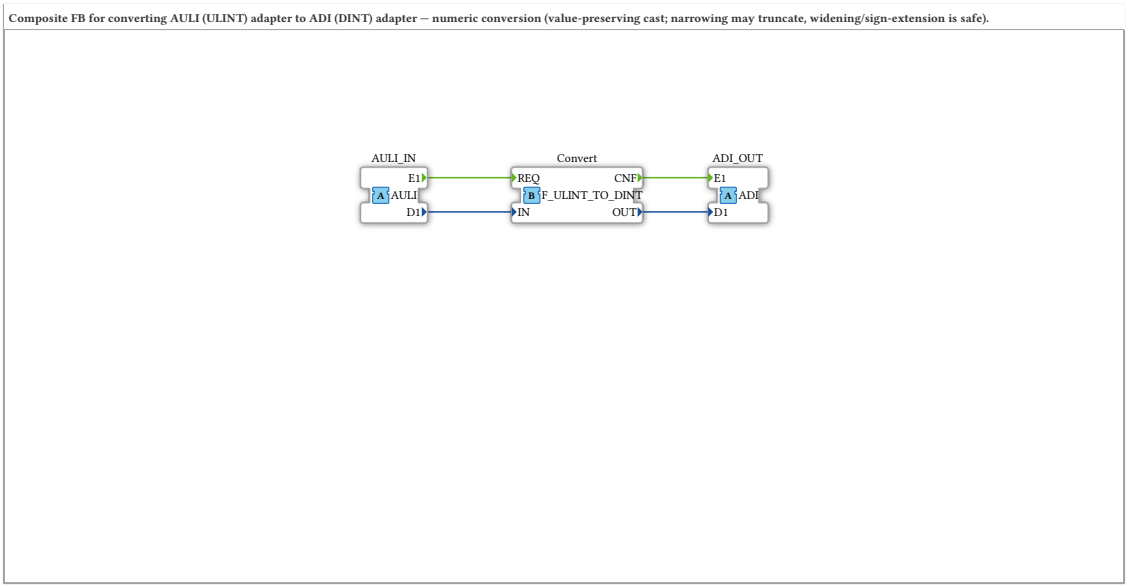
The `ADI_TO_AULI` block is a compact, adapter-based wrapper for converting DINT to ULINT values. By cleanly separating adapter and conversion logic, it increases the reusability and readability of control programs. It is particularly suitable for modular automation solutions that rely on unidirectional adapters and require standardized type conversion.

AULI_TO_ADI

ULINT Adapter Input – *AULI*



ADI – DINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AULI_TO_ADI** function block is a composite module for converting an **AULI** adapter (unsigned 64-bit value) into an **ADI** adapter (signed 32-bit value). It enables simple conversion between two unidirectional adapter interfaces and internally utilizes an IEC 61131 type conversion function.

Interface Structure

The module has no event or data inputs/outputs of its own. All communication takes place exclusively via the two adapters.

Event Inputs

None (exclusively via adapters).

Event Outputs

None (exclusively via adapters).

Data Inputs

None (exclusively via adapter).

Data Outputs

None (exclusively via adapter).

Adapters

Type	Name	Direction	Description
adapter::types::unidirectional::AULI	AULI_IN	Socket / Input	Receives a ULINT value via event and data channels (E1, D1)
adapter::types::unidirectional::ADI	ADI_OUT	Plug / Output	Provides the converted DINT value via event and data channels (E1, D1)

Functionality

The module operates as a pure cascade:

1. An event at socket `AULI_IN.E1` triggers the data transfer from `AULI_IN.D1` (ULINT).
2. This event is transferred to the input `REQ` of the internal conversion module `F_ULINT_TO_DINT` (IEC 61131 type conversion).
3. The converter transforms the ULINT value into a DINT value and places it at its output `OUT`.
4. After the conversion is complete, the converter signals the event with `CNF`, which triggers the plug `ADI_OUT.E1` and simultaneously makes the converted value available on `ADI_OUT.D1`.
5. The connected receiver thus receives a DINT value that corresponds to the original ULINT value – insofar as it can be represented (DINT value range: -2^{31} … $2^{31}-1$).

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-based**: The function block does not implement direct input/output variables, but uses only adapter sockets and plugs for loose coupling.
- **Type conversion**: The conversion is performed using the IEC 61131 standard function block `F_ULINT_TO_DINT`. ULINT values outside the DINT range can lead to overflows or unexpected results.
- **Compact Structure**: Since there is no state logic or branching, the function block operates deterministically and without delay beyond the pure conversion time.
- **Package**: The function block is packaged in `adapter::conversion::unidirectional`.

State Overview

The function block has no internal state machine. It behaves like a pure functional mapping: An input event is immediately followed by an output event with the converted value.

Application Scenarios

- **Interface Adaptation**: When a sensor or controller provides a ULINT adapter, but the downstream system expects a DINT adapter.
- **Data Reduction**: Reducing the bit width from 64 to 32 bits in cases where a 32-bit resolution is sufficient.
- **Adapter Cascades**: Used as a link in a seamless conversion chain between different data adapter types.

Comparison with Similar Function Blocks

- **AULI_TO_UDI**: Converts ULINT to UDINT (unsigned 32-bit) – the value range remains positive, and overflows only occur with values > $2^{32}-1$.

- **DIRECT_CONVERT**: A generic function block with direct data input and output that does not use adapters – AULI_TO_ADI, on the other hand, offers loose coupling and standardized adapter interfaces.
- **Manual Conversion**: Without this function block, the conversion would have to be implemented in the application function block itself – AULI_TO_ADI encapsulates this task and promotes reusability.

Conclusion

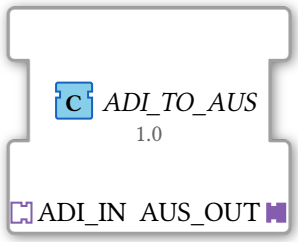
AULI_TO_ADI is a specialized, adapter-based conversion module that efficiently and in a standardized manner handles the conversion of ULINT to DINT data via unidirectional adapters. By utilizing an IEC 61131-compliant conversion function block, the conversion is reliable, and the simple adapter interface allows for flexible integration into existing systems. This module is particularly well-suited for loosely coupled architectures where data formats need to be adapted without burdening the application logic with type conversion.

Numeric Index (unidirectional)

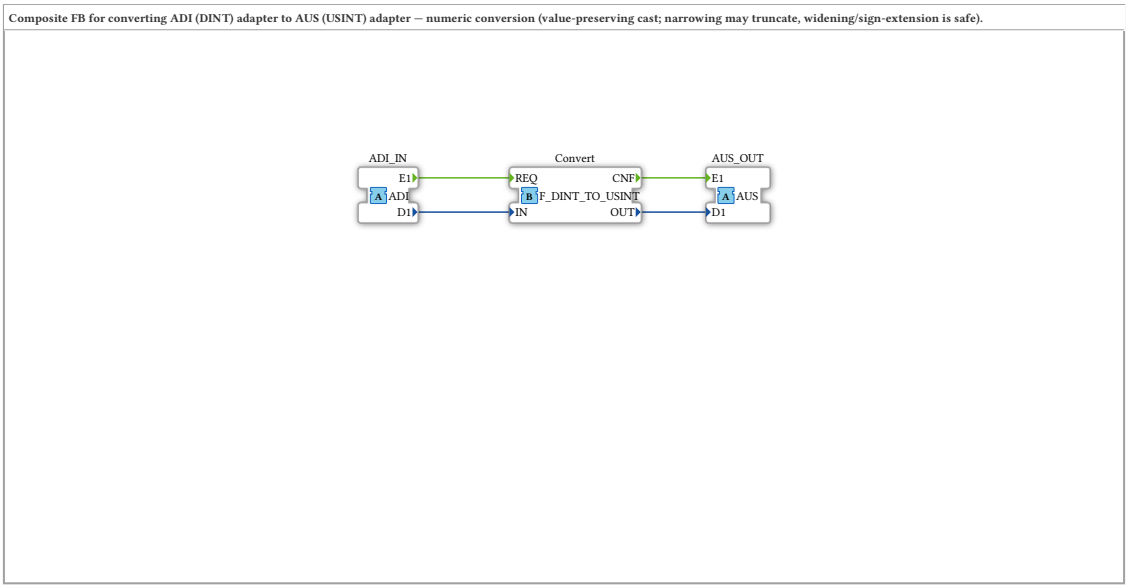
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

ADI_TO_AUS

DINT Adapter Input – *ADI*



AUS – USINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The ADI_TO_AUS function block acts as an adapter converter from a DINT adapter (ADI) to a USINT adapter (AUS). It converts a DINT value to a USINT value and enables the seamless integration of components with different data types in the 4diac IDE.

Interface Structure

Event Inputs

- **E1** (via ADI_IN): This event triggers the conversion.

Event Outputs

- **E1** (via AUS_OUT): Signals the completion of the conversion and the availability of the output value.

Data Inputs

- **D1** (via ADI_IN): The DINT value to be converted.

Data Outputs

- **D1** (via AUS_OUT): USINT value, the result of the conversion.

Adapters

- **ADI_IN**: Socket (input) of type `adapter::types::unidirectional::ADI`. Provides DINT data and events.
- **AUS_OUT**: Plug (output) of type `adapter::types::unidirectional::AUS`. Receives USINT data and events for further processing.

Functionality

The function block is implemented as a composite FB. Upon an event at event input E1 (via ADI_IN), the corresponding DINT value is read from data input D1 and passed to the internal conversion block `F_DINT_TO_USINT`. This function block converts the value according to the IEC 61131 standard (e.g., by limiting it to the USINT range 0–255 or by type conversion). The result is output via data output D1, and an event at output E1 (via AUS_OUT) signals the availability of the new value.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- The function block uses an IEC 61131-compliant conversion block for type conversion from DINT to USINT.
- It is a composite function block (nested block) that maps the logic in a network of a single Convert function block.
- Communication takes place exclusively via adapter interfaces, which enables modular encapsulation.
- During conversion, values outside the USINT range (0–255) can be truncated or limited to the maximum or minimum value (depending on the conversion block used).

State Overview

Since the function block does not have its own state machine, its behavior is purely event-driven: It reacts to every incoming event with an immediate conversion and output. There are no internal states or delays.

Application Scenarios

- Connecting a sensor that delivers DINT values to a controller that expects USINT.
- Integrating adapter-based components with different data types into a 4diac application.
- Migrating existing systems where data formats need to be adapted.

Comparison with Similar Function Blocks

- A direct conversion function block like `F_DINT_TO_USINT` performs the same conversion but operates on standard inputs/outputs. The ADI_TO_AUS encapsulates the logic in adapter interfaces and is therefore specifically designed for adapter-based communication paths.
- This function block differs from other adapter converters due to the specific type combination DINT↔USINT.

Conclusion

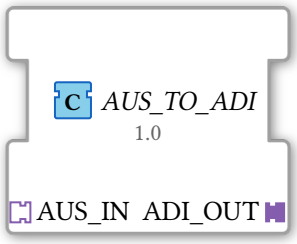
The ADI_TO_AUS is a practical building block for the seamless integration of DINT and USINT adapters in the 4diac IDE. Its simple, event-driven functionality and encapsulation as a composite function block make it flexible and easily maintainable. It is particularly suitable for projects that require a clear separation between adapter interfaces and conversion logic.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

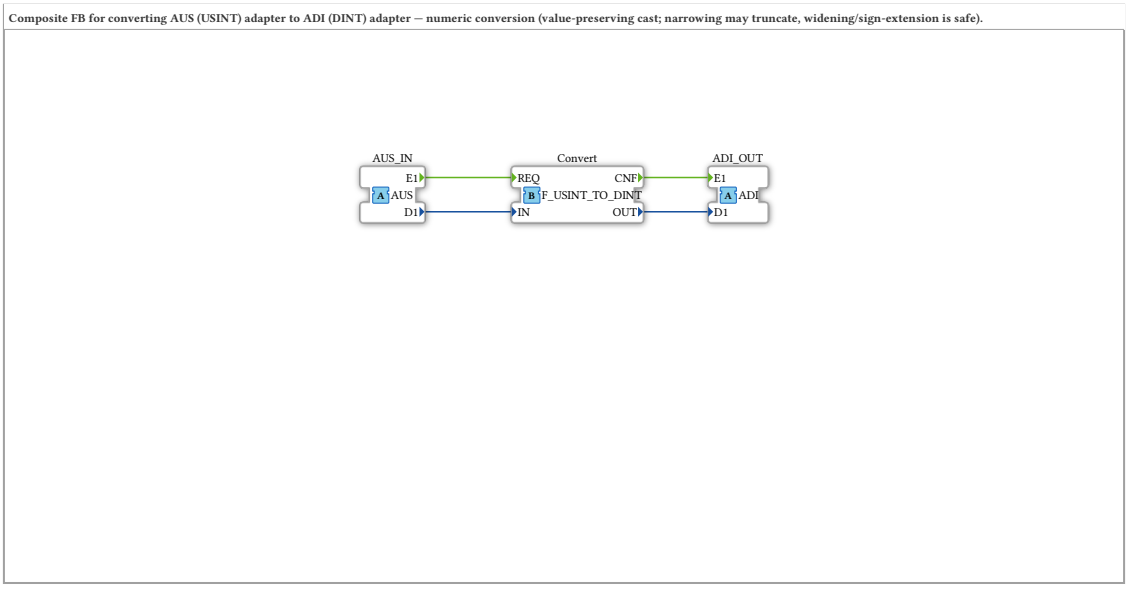
]

AUS_TO_ADI



USINT Adapter Input – *AUS*

ADI – DINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block AUS_TO_ADI is a composite function block that performs a unidirectional adapter conversion from `AUS` (data of type `USINT`) to `ADI` (data of type `DINT`). It encapsulates an IEC 61131 conversion block, thus enabling a clean separation of adapter types in the 4diac IDE.

Interface Structure

Event Inputs

No separate event inputs are available. Event control is handled via the socket adapter `AUS_IN` (see section **Adapters**).

Event Outputs

No separate event outputs are available. Event forwarding is handled via the plug adapter `ADI_OUT` (see section **Adapters**).

Data Inputs

No separate data inputs are available. Data is provided via the socket adapter `AUS_IN`.

Data Outputs

No separate data outputs are available. The converted data is output via the plug adapter `ADI_OUT`.

Adapters

Name	Direction	Type	Description
AUS_IN	Socket (Input)	adapter::types::unidirectional::AUS	Unidirectional adapter input that provides a USINT value (and an associated event).
ADI_OUT	Plug (Output)	adapter::types::unidirectional::ADI	Unidirectional adapter output that outputs the converted DINT value (and an associated event).

Functionality

The module operates in an event-driven manner:

1. An incoming event at socket AUS_IN.E1 triggers the internal conversion module Convert (type F_USINT_TO_DINT).
2. The conversion module reads the data value AUS_IN.D1 (USINT) and converts it into a DINT value.
3. After the conversion is complete, an event is generated at output Convert.CNF.
4. This event is forwarded to the plug adapter ADI_OUT.E1, and the converted value is output via ADI_OUT.D1.

All logic is represented as data flow within the internal function block network – no state machine logic is used within the composite block.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- The block uses the IEC 61131 block F_USINT_TO_DINT (type conversion of unsigned 8-bit to signed 32-bit integer).
- It is a pure **composite block** – its behavior is entirely defined by the internal function block network.
- The adapters are designed to be **unidirectional**: data flows only from the socket to the plug.
- By encapsulating the conversion in a separate block, different adapter types can be interconnected without requiring the surrounding application to convert the types.

State Overview

The block does not have its own state machine. Its behavior is controlled directly by the internal conversion block and the event connections. Error handling (e.g., range exceeding during conversion) is not explicitly implemented and would need to be added either in the calling context or by extending the function block.

Application Scenarios

- **Adapter Bridge**: Exchange of data between components that use different adapter types (e.g., sensors with USINT values and controllers that expect DINT values).
- **Data Integration**: Integration of function blocks from IEC 61131 libraries into a 4diac application based on adapters.
- **Type Conversion**: Fast conversion of data types at the adapter level without having to change the logic of the connected function blocks.

Comparison with Similar Modules

- Unlike directly using the conversion module F_USINT_TO_DINT, AUS_TO_ADI offers an adapter-oriented interface. This makes the conversion completely transparent for the user, who only works with the adapters.
- Similar adapter conversions may exist for other data types (e.g., BOOL_TO_ADI, INT_TO_ADI). This module is specifically designed for converting USINT to DINT.

Conclusion

The AUS_TO_ADI composite block offers a simple and clean way to convert a AUS adapter (USINT-based) into a ADI adapter (DINT-based). Thanks to the internal use of the IEC 61131 conversion block, the con-

version is standards-compliant and reusable. The block is particularly suitable for modular applications where different adapter types need to be harmonized.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

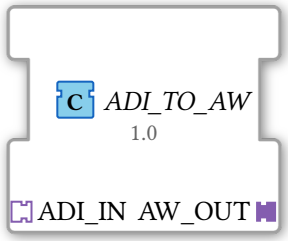
]

Numeric Index (unidirectional)

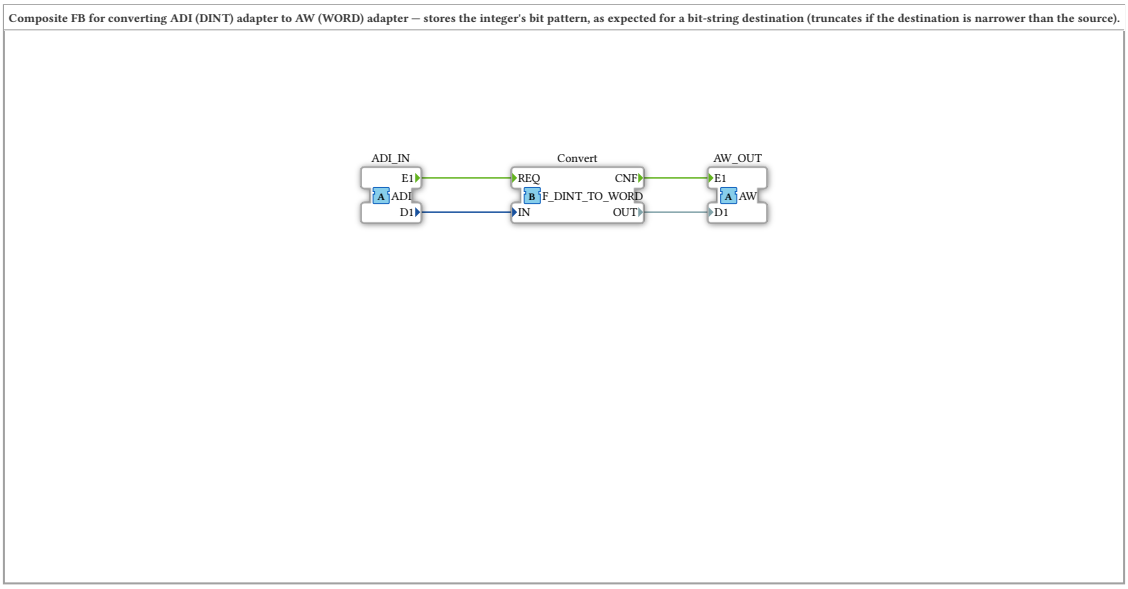
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

ADI_TO_AW

DINT Adapter Input – *ADI*



AW – WORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_TO_AW** is a composite module that converts a DINT value into a WORD value. It acts as a unidirectional converter between an **ADI** adapter (DINT input) and an **AW** adapter (WORD output). The function block encapsulates the conversion logic and provides a standardized interface for data exchange at the adapter level.

Interface Structure

The function block has no direct event or data inputs/outputs. All communication takes place via adapters:

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::ADI`	`ADI_IN`	Socket (Input)	DINT adapter providing the value to be converted

adapter::types::unidirectional::AW	AW_OUT	Plug (Output)	WORD adapter outputting the converted value

The internal event and data connections of the composite function block result from the adapter interfaces used:

Event Inputs (via ADI_IN)

Event	Description
E1	Trigger for converting the DINT value

Event Outputs (via AW_OUT)

Event	Description
E1	Confirmation of successful conversion and output of the WORD value

Data Inputs (via ADI_IN)

Identifier	Type	Description
D1	DINT	The integer value to be converted

Data Outputs (via AW_OUT)

Identifier	Type	Description
D1	WORD	The converted bit pattern value (16 bits)

Functionality

The function block operates in an event-driven manner:

1. An incoming event at ADI_IN.E1 is internally forwarded to the conversion block F_DINT_TO_WORD (via its REQ input).
2. Simultaneously, the data value from ADI_IN.D1 is transferred as DINT to the IN input of the conversion function.
3. The function block F_DINT_TO_WORD converts the DINT value to a WORD value (according to IEC 61131-3).
4. After the conversion is complete, the converter's output event (CNF) is sent.
5. This event is forwarded to AW_OUT.E1, and the converted WORD value (from OUT) is passed to AW_OUT.D1.

Thus, the entire conversion occurs in a strict event/data flow without any additional states.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Adapter-Based Encapsulation:** The function block (FB) uses adapters exclusively for input and output, enabling modular and reusable integration in 4diac projects.
- **Unidirectional Conversion:** Data conversion only occurs from socket to plug; reverse conversion is not supported.
- **Use of IEC 61131 Libraries:** The core function F_DINT_TO_WORD is derived from the iec61131::conversion package, ensuring standards-compliant and high-performance conversion.
- **No Internal States:** The FB does not have an internal state diagram; its logic is determined solely by the behavior of the contained block.

State Overview

Since this is a purely structured composite FB without its own Execution Control Chart (ECC), there is no explicit state model. The behavior is purely data stream-driven and corresponds to that of the internal conversion block (IDLE → REQ → CNF → IDLE).

Application Scenarios

- **PLC Coupling:** Connecting a DINT-based sensor (e.g., encoder) to a subsystem that expects WORD values.
- **Data Format Conversion in Adapter Chains:** Use in a 4diac application where different adapter types need to be harmonized.
- **Interface Standardization:** Implementing a standardized WORD interface for downstream logic blocks, while the source is in DINT format.

Comparison with Similar Blocks

Block	Function	Difference
`F_DINT_TO_WORD`	Direct DINT to Word conversion	No adapter encapsulation; must be manually integrated into a network
ADI_TO_AW`	Adapter-based encapsulation of the same conversion	Provides ready-to-use adapter interfaces; simplifies reuse
WORD_TO_DINT`	Reverse conversion	Other direction; requires a custom adapter type

The `ADI_TO_AW` is specifically optimized for environments where adapters are required as standardized interfaces.

Conclusion

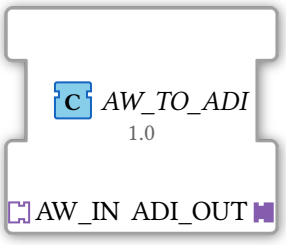
The function block `ADI_TO_AW` provides a reusable, adapter-based solution for converting `DINT` to `WORD`. It cleanly encapsulates the conversion logic and enables easy integration into 4diac architectures based on the adapter concept. The use of standard libraries ensures reliability and adherence to specifications. This component is particularly suitable for scenarios where different data formats need to be connected via adapter bridges.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

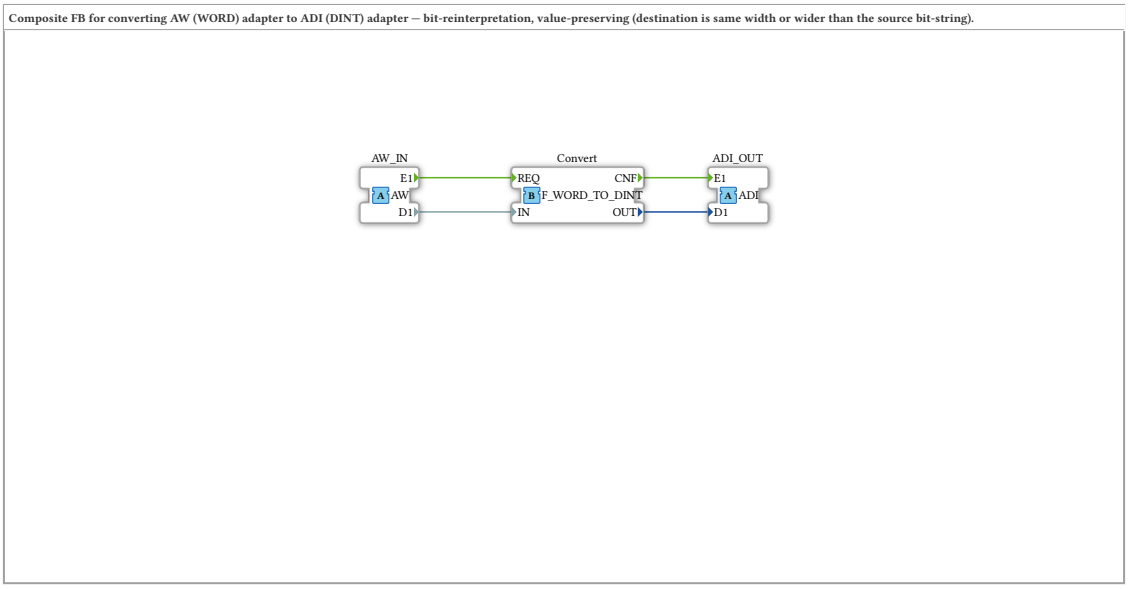
]

AW_TO_ADI



WORD Adapter Input – *AW*

ADI – DINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The **AW_TO_ADI** function block is a composite block that converts a unidirectional WORD adapter (AW) into a unidirectional DINT adapter (ADI). It enables the seamless integration of components that work with WORD data into systems that expect DINT values. Internally, the block uses the standard conversion block `F_WORD_TO_DINT` from the IEC 61131 library.

Interface Structure

Event Inputs

- **No Direct Event Inputs**

The block does not have any standalone event inputs. The triggering event is provided via the socket adapter `AW_IN`.

Event Outputs

- **No Direct Event Outputs**

This function block has no independent event outputs. The result event is passed on via the plug adapter `ADI_OUT`.

Data Inputs

- **No Direct Data Inputs**

The WORD value to be converted is received via the socket adapter `AW_IN`.

Data Outputs

- **No Direct Data Outputs**

The converted DINT value is output via the plug adapter `ADI_OUT`.

Adapters

Name	Type	Direction	Description
`AW_IN`	`adapter::types::unidirectional::AW`	Socket (Input)	WORD adapter as input. Returns the trigger event (E1) and the WORD data value (D1).

ADI_OUT	adapter::types::unidirectional::ADI	Plug (Output)	DINT adapter as output. Returns the acknowledgment event (E1) and the converted DINT value (D1).

Functionality

1. An incoming event at socket AW_IN.E1 triggers the internal function block F_WORD_TO_DINT via its event input REQ.
2. Simultaneously, the WORD data value from AW_IN.D1 is transferred to the data input IN of F_WORD_TO_DINT.
3. After the conversion is complete, F_WORD_TO_DINT generates an acknowledgment event at CNF.
4. This event is forwarded to plug ADI_OUT.E1, while the converted DINT value is transferred from F_WORD_TO_DINT.OUT to ADI_OUT.D1.

The entire process is thus synchronous: A pulse at the WORD adapter results in a pulse at the DINT adapter with the converted value.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Pure Coupling:** The block contains no logic of its own, but simply connects the adapter interfaces to an existing conversion block.
- **Use of IEC 61131 Standard Blocks:** The actual conversion is performed by F_WORD_TO_DINT, which provides a defined and standardized conversion.
- **Unidirectional Adapters:** Both AW_IN and ADI_OUT are designed as unidirectional adapters, meaning they only transport events and data in one direction.
- **No Internal States:** As a composite device, AW_TO_ADI has no internal states of its own – its behavior is entirely determined by the interconnection of its constituent elements.

State Overview

Since this is a composite device without its own state machine, there is no separate state overview. The runtime behavior is determined by the embedded F_WORD_TO_DINT, which performs a processing step after an input event and outputs the result.

Application Scenarios

- **Agricultural Technology:** Connecting sensors or actuators that deliver WORD data (e.g., via field-buses with 16-bit values) to controllers that require DINT.
- **Migration of Legacy Systems:** Replacing WORD-based interfaces with modern DINT components without redesigning the entire architecture.
- **Adapter Encapsulation:** Integrating libraries or modules designed for different data types using a single adapter converter.

Comparison with Similar Components

- **Direct Conversion:** Unlike a simple F_WORD_TO_DINT, AW_TO_ADI offers an adapter-based interface that enables loose coupling between components.
- **Bidirectional Converters:** There are modules that combine both forward and reverse conversion (e.g., WORD ↔ DINT) in a single module. This module is intentionally designed to be unidirectional.
- **Type Conversion via Flexible Adapters:** Other adapter converters use generic or polymorphic interfaces; this module is specifically designed for WORD-to-DINT conversion.

Conclusion

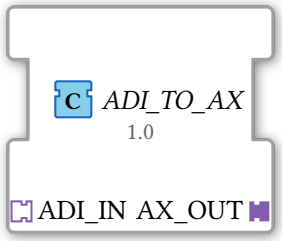
The AW_TO_ADI is a simple yet effective composite module for converting a WORD adapter to a DINT adapter. By using a standardized conversion module and clear adapter interfaces, it is particularly well-suited for use in modular automation solutions where data type conversions between components are required. Its structure is transparent, easy to understand, and allows for rapid integration into existing projects.

Numeric Index (unidirectional)

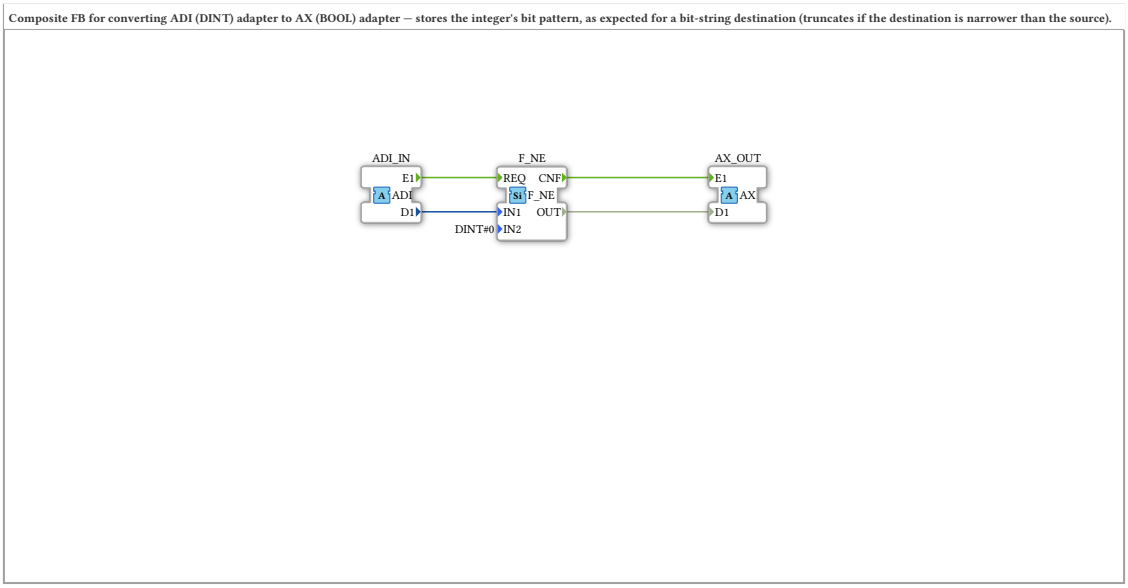
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

ADI_TO_AX

DINT Adapter Input – *ADI*



AX – BOOL Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **ADI_TO_AX** function block is a composite module that performs a simple signal conversion between two adapter types. It converts a DINT value (type: *adapter::types::unidirectional::ADI*) received via the **ADI_IN** socket into a BOOL value and outputs it via the **AX_OUT** plug (type: *adapter::types::unidirectional::AX*). The conversion is performed by comparing the received DINT value with the constant value 0. If the value is not equal to 0, the BOOL output is set to TRUE; if they are equal, it is set to FALSE. This module is particularly suitable for connecting digital signal sources encoded as integers to Boolean interfaces.

Interface Structure

Event Inputs

Name	Description
(No explicit event inputs)*	The function block (FB) uses only the events connected via the ADI_IN adapter. In fact, the event `E1` of the ADI_IN adapter is internally forwarded to the function block `F_NE`.

Event Outputs

Name	Description
(No explicit event outputs)*	The output is via the AX_OUT adapter; the event `E1` of the AX_OUT adapter is sent after successful processing of the `F_NE` call.

Data Inputs

Name	Type	Description
(via ADI_IN adapter)	`DINT`	The DINT value (e.g., measured value or status code) connected via the adapter socket is received as input `D1` of the ADI_IN adapter.

Data Outputs

Name	Type	Description
(via AX_OUT adapter)	`BOOL`	The output value is provided as `D1` of the AX_OUT adapter. TRUE if the received DINT value is not 0; otherwise FALSE.

Adapter

Adapter	Name	Direction	Type	Description
Socket	ADI_IN	Input	`adapter::types::unidirectional::ADI`	Receives a DINT value (with associated event).
Plug	AX_OUT	Output	`adapter::types::unidirectional::AX`	Passes on the resulting BOOL value (with associated event).

Functionality

Internally, the function block consists of a single function block **F_NE** of type `iec61131::comparison::F_NE`. This implements the IEC 61131 function "Not Equal" (NE). Behavior:

1. An incoming event **E1** from the ADI_IN adapter is forwarded to the **REQ** input of **F_NE**.
2. The data value **D1** of the ADI_IN adapter is applied to the **IN1** input of the **F_NE**. The second comparison value, **IN2**, is fixed to the value `DINT#0`.
3. The **F_NE** compares `IN1 != IN2`. The result (BOOL) is output at the **OUT** output.
4. After processing is complete, **F_NE.CNF** signals completion. This event is sent to the **E1** input of the AX_OUT adapter, which then passes the output value (also the **OUT** signal) to **D1** of the AX_OUT adapter.

Thus, the function block implements a simple threshold logic: Any non-zero DINT value is interpreted as logically "true". The function block operates strictly event-driven – each incoming event triggers exactly one comparison and output action.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Adapter Coupling:** The function block is designed as a composite block that connects two unidirectional adapters. It can therefore be seamlessly integrated into an adapter-based communication architecture.
- **No Internal States:** Since the function block has no internal state variables or timers, its behavior is purely combinatorial and deterministic.
- **Direct Impedance:** The function block used, `F_NE` from the IEC 61131 library, is high-performance and does not require dynamic memory management.
- **Type Safety:** The constant `DINT#0` is explicitly declared as a DINT, so no implicit type conversions occur.
- **License Notice:** This function block is subject to the Eclipse Public License 2.0 (EPL-2.0), Copyright 2026 HR Agrartechnik GmbH.

State Overview

This function block has no states of its own, as it is entirely based on the internal functional block **F_NE**. Therefore, its behavior requires no delays or storage. The sequence is as follows:

- **Waiting for Event:** Inactive until an event arrives via the ADI_IN adapter.
- **Processing:** Upon receiving an event, the comparison is performed; the output is updated immediately. After the comparison, the output event is sent.

Application Scenarios

- **Digitization of Analog Values:** In automation, DINT values (e.g., counter readings, error codes) are often interpreted as Boolean signals. This function block is suitable for converting non-zero conditions into binary signals.
- **Adapter Bridging:** If a component only provides an ADI adapter, but the downstream logic only accepts AX adapters, this function block can be used as a lossless converter.
- **Simple Switching Thresholds:** With minimal effort, it's possible to determine whether a value deviates from zero – useful, for example, for monitoring counting pulses or limit violations.

Comparison with Similar Function Blocks

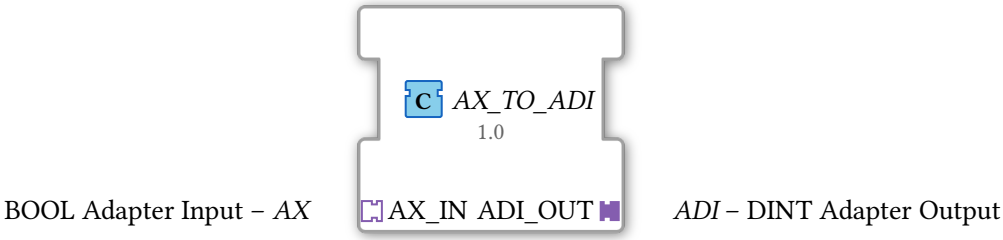
Function Block	Type	Function
ADI_TO_AX (this function block)	Composite	DINT ≠ 0 ⟶ BOOL
`BOOL_TO_DINT` (Standard Library)	Function	Direct conversion from BOOL to DINT (e.g., TRUE ⟶ 1)
`F_NE` (IEC 61131)	Function	General Non-Equal Comparison of Arbitrary Data Types
Custom Adapter Converter	–	Can be easily extended with other comparison functions (F_EQ, F_GT, etc.)

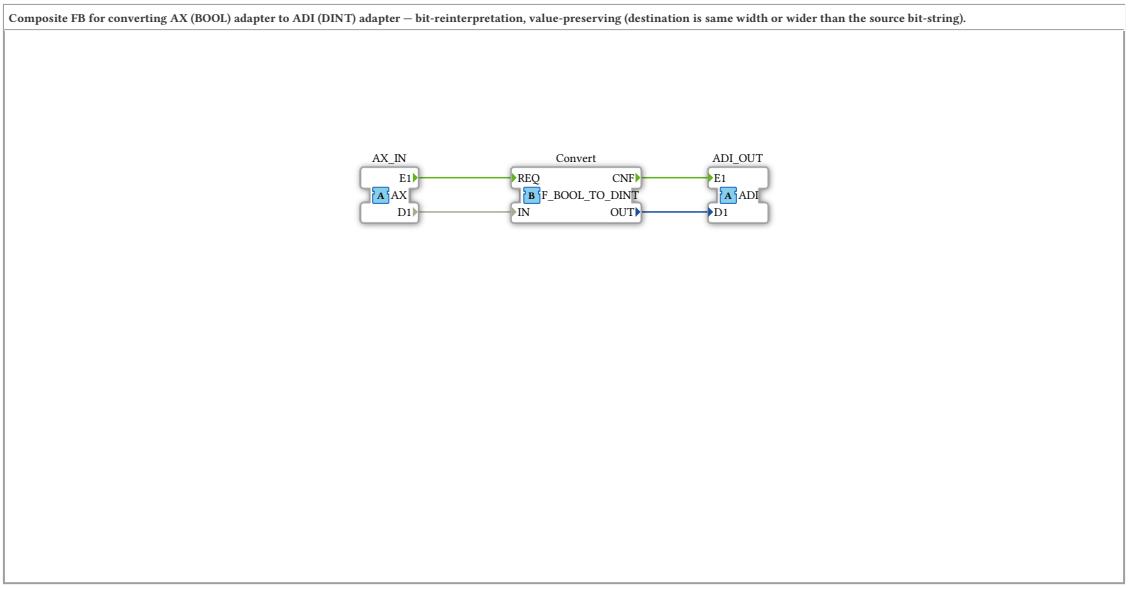
Compared to a pure BOOL-to-DINT conversion, **ADI_TO_AX** offers the advantage that the DINT value is retained exactly (e.g., 42 ⟶ TRUE), whereas a Boolean conversion can only distinguish between TRUE/FALSE and 1/0.

Conclusion

The **ADI_TO_AX** function block is a lean and robust adapter converter for unidirectional interfaces. It utilizes the proven IEC 61131 comparison block `F_NE` and thus implements a simple yet useful logic: "Any non-zero value is true." The function block is particularly suitable for use in modular automation systems where different adapter standards need to be coordinated. Thanks to its composite structure, it can be easily integrated into existing networks and expanded as needed.

AX_TO_ADI





• — ○ • — ○ • — ○ •

Introduction

The **AX_TO_ADI** function block is a composite function block (FB) that converts an AX adapter with a Boolean interface (BOOL) into an ADI adapter with an integer interface (DINT). It serves as a converter between unidirectional adapters and enables the seamless integration of binary signals into systems that use integer-based adapters.

Interface Structure

Since it is a composite FB, all inputs and outputs are implemented via the included adapters. The following list shows the logical interfaces.

Event Inputs

The function block does not have direct event inputs. Event control is handled via the **AX_IN** adapter:

- **E1** (BOOT): Event input of the AX adapter that triggers the data conversion.

Event Outputs

This function block does not have direct event outputs. Event forwarding is handled via the **ADI_OUT** adapter:

- **E1** (CNF): Event output of the ADI adapter, signals completion of the conversion.

Data Inputs

This function block does not have direct data inputs. Data is acquired via the **AX_IN** adapter:

- **D1** (BOOL): Boolean input value that is converted to a DINT value.

Data Outputs

This function block does not have direct data outputs. Data output is handled via the **ADI_OUT** adapter:

- **D1** (DINT): Converted integer output value (0 for FALSE, 1 for TRUE).

Adapter

- **AX_IN** (Socket): Receives an AX adapter (type: `adapter::types::unidirectional::AX`) with a Boolean data and event key.
- **ADI_OUT** (Plug): Provides an ADI adapter (type: `adapter::types::unidirectional::ADI`) with a DINT data and event key.

Functionality

The function block internally uses the IEC 61131-3 converter `F_BOOL_TO_DINT`. As soon as an event arrives at **AX_IN.E1**, the current value of **AX_IN.D1** is read and passed to the converter. The converter sets the value 0 for FALSE and the value 1 for TRUE. After the conversion is complete, the result is passed to **ADI_OUT.D1** and an event is simultaneously sent to **ADI_OUT.E1**.

The event chain is thus fully looped:
AX_IN.E1→ `Conversion` →ADI_OUT.E1`.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Pure Data Conversion:** The function block does not perform any state logic or time delay – every event activation at the input immediately generates an output.
- **Use of IEC 61131 Libraries:** The conversion is based on the standardized function block `F_BOOL_TO_DINT` from the IEC 61131 conversion library.
- **Unidirectional Adapters:** Both AX and ADI are unidirectional adapters, meaning they only support one direction (input or output). This simplifies data flow control.
- **Composite Function Block without its own state:** As a composite function block, it has no internal state machine; all logic is implemented by the embedded function block.

State Overview

As a composite function block, **AX_TO_ADI** does not have its own state machine. Its behavior is determined solely by the embedded `F_BOOL_TO_DINT`, which has no state-dependent logic. The output is always a direct function of the input at the time of event activation.

Application Scenarios

- **Binary Signal Conversion:** When a sensor or switch provides a Boolean signal via an AX adapter that needs to be further processed in a system with ADI adapters (DINT).
- **Gateway Function:** Connection between subsystems using different adapter types without changing the underlying logic.
- **Testing and Simulation:** Generation of a defined DINT value from a Boolean input for testing purposes.

Comparison with Similar Function Blocks

- **AX_TO_ADI (this function block):** Converts BOOL to DINT via adapters.
- **ADI_TO_AX (hypothetical):** Would convert DINT to BOOL, e.g., by comparison with zero or a threshold.
- **Direct Command F_BOOL_TO_DINT:** This function block is an encapsulated version specifically designed for use with unidirectional adapters. Unlike the basic function block, the adapter interfaces are already integrated.

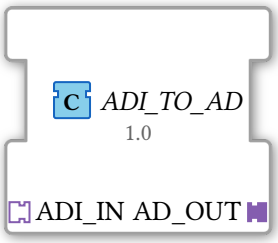
Conclusion

AX_TO_ADI is a simple yet useful converter function block that converts Boolean adapter signals into integer-based adapter signals. It enhances interoperability between different components of a system and enables a clean separation of data and event flows. Due to its low complexity, it can be deployed quickly and easily integrated into larger composite function block chains.

Numeric Index (unidirectional)

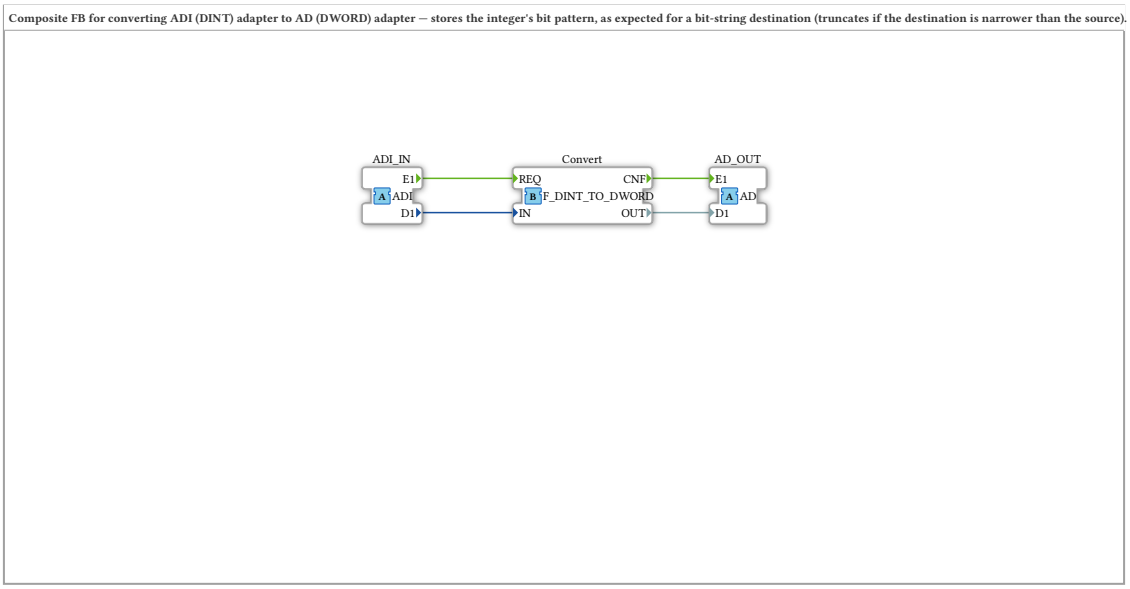
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ADI_TO_AD



DINT Adapter Input – *ADI*

AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **ADI_TO_AD** function block is a composite block that performs interface conversion from an **ADI** adapter (DINT-based) to an **AD** adapter (DWORD-based). It functions as a unidirectional converter and enables seamless integration of components with different data adapter types in the 4diac IDE.

Interface Structure

Event Inputs

The function block does not have its own dedicated event inputs at the interface level. Event control is handled exclusively via the connected **ADI_IN** socket adapter:

- **ADI_IN.E1** – Event that triggers the data conversion.

Event Outputs

There are no separate event outputs at this level either. The output is provided via the **AD_OUT** plug adapter:

- **AD_OUT.E1** – Event triggered after successful conversion.

Data Inputs

- **ADI_IN.D1** – Input of the DINT value to be converted (source adapter).

Data Outputs

- **AD_OUT.D1** – Output of the converted DWORD value (destination adapter).

Adapters

- **ADI_IN** (Socket) – of type `adapter::types::unidirectional::ADI` (DINT data).
- **AD_OUT** (Plug) – of type `adapter::types::unidirectional::AD` (DWORD data).

Both adapters are unidirectional (one event channel and one data channel each).

Functionality

The function block contains an internal block `F_DINT_TO_DWORD` from the IEC 61131 library, which performs the actual type conversion. The sequence of operations is as follows:

1. An incoming event at **ADI_IN.E1** triggers the **Convert.REQ** input of the internal converter.
2. The converter converts the DINT value received at **ADI_IN.D1** into a DWORD value.
3. After the conversion is complete, **Convert.CNF** signals successful completion and fires the event at **AD_OUT.E1**, while simultaneously outputting the converted value to **AD_OUT.D1**.

This enables transparent, event-driven conversion between the adapter types.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Type Conversion**: The actual conversion is performed using the standardized IEC 61131 function block `F_DINT_TO_DWORD`, which ensures bit-perfect transfer of the integer value to the DWORD space (no logical interpretation, only type conversion).
- **Unidirectionality**: The function block is strictly unidirectional and expects exactly one input and one output adapter. Reverse conversion is not supported.
- **Package Structure**: The function block is organized in the package `adapter::conversion::unidirectional` and imports the corresponding IEC library.

State Overview

As a composite function block, **ADI_TO_AD** has no explicit states of its own. Its behavior is entirely determined by the internal converter:

- **Idle**: Waits for an event at the ADI_IN.

Idle }: Waits for an event at the ADI_IN. - **Conversion active: When `Convert.REQ` is set.

- **Completion**: After `Convert.CNF`, the result is provided at the output.

Application Scenarios

- **Interface adaptation**: When a component provides a DINT adapter, but a downstream module expects a DWORD adapter (e.g., when integrating controllers with different data widths).
- **System modernization**: Older DINT-based sensors or actuators (via ADI) are to be connected to new DWORD-enabled bus systems (via AD).
- **Prototypical development**: Rapid switching between adapter variants without changing the application logic.

Comparison with Similar Function Blocks

- **Direct Converter Function Blocks**: Unlike pure type converters (such as `F_DINT_TO_DWORD`), **ADI_TO_AD** operates at the adapter level and integrates event control and interface adaptation into a single function block.
- **Bidirectional Adapters**: Function blocks also exist that offer conversion in both directions (e.g., ADI_TO_AD and AD_TO_ADI combined). This function block is specialized for one direction and is therefore more streamlined.
- **Manual Wiring**: Without this function block, the adapters would have to be wired manually internally, and the event signals routed separately – the composite function block encapsulates this complexity.

Conclusion

The **ADI_TO_AD** function block is a useful and clearly structured composite function block for easy adaptation between DINT- and DWORD-based adapters. It reduces manual wiring effort, utilizes proven IEC conversion logic, and integrates seamlessly into unidirectional data flows. Ideal for developers who need to switch quickly and cleanly between adapter types without delving deeply into internal signal processing.

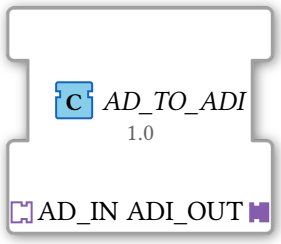
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

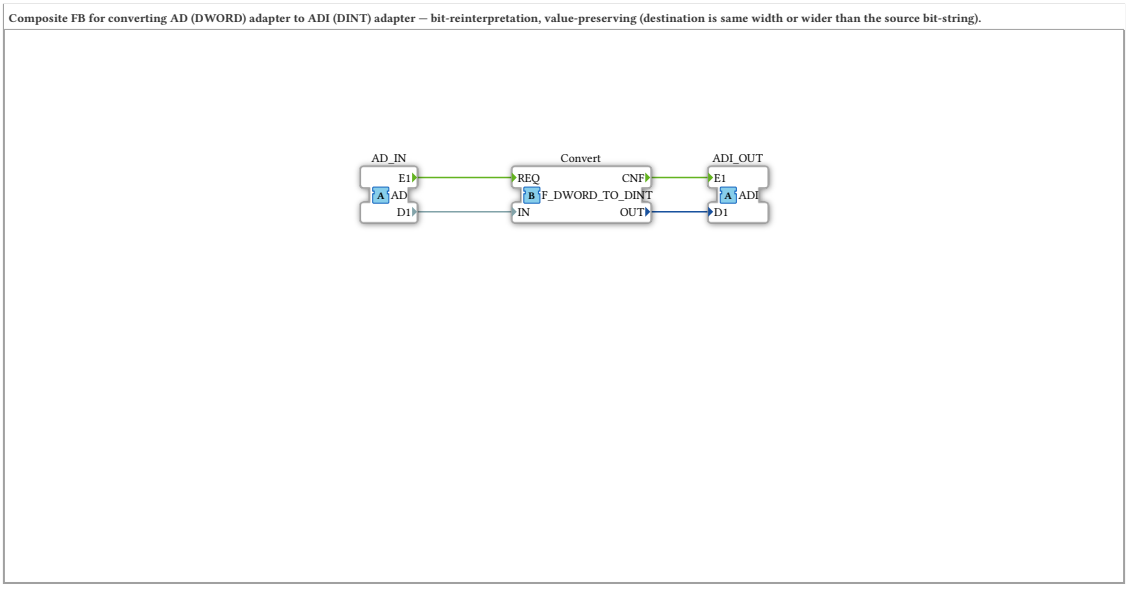
]

AD_TO_ADI

DWORD Adapter Input – *AD*



ADI – DINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **AD_TO_ADI** function block is a composite module for converting a DWORD adapter (AD) to a DINT adapter (ADI). It enables the seamless integration of components with different data types into an IEC 61499 application without requiring the user to implement the conversion logic themselves.

Interface Structure

Event Inputs

No direct event inputs. Event control is handled via the socket adapter `AD_IN`.

Event Outputs

No direct event outputs. Event output is handled via the plug adapter `ADI_OUT`.

Data Inputs

No direct data inputs. Data is transferred via the socket adapter `AD_IN`.

Data Outputs

No direct data outputs. Data is transferred via the plug adapter `ADI_OUT`.

Adapters

Name	Type	Direction	Comment

AD_IN	adapter::types::unidirectional::AD	Socket (Input)	DWORD Adapter Input – returns a DWORD date and an event
ADI_OUT	adapter::types::unidirectional::ADI	Plug (Output)	DINT Adapter Output – Returns a Converted DINT Data and an Event

Functionality

This function block encapsulates a call to the IEC 61131-3 conversion function `F_DWORD_TO_DINT`. The process is as follows:

1. An event at socket `AD_IN.E1` triggers the conversion function block `Convert` (event `REQ`).
2. Simultaneously, the DWORD data `AD_IN.D1` is passed to input `Convert.IN`.
3. The conversion function block converts the value into a DINT and signals completion via the event `CNF`.
4. This event is forwarded to plug `ADI_OUT.E1`, and the converted DINT value (`Convert.OUT`) is placed in `ADI_OUT.D1`.

As a result, from the application's perspective, the composite function block behaves like a direct adapter converter without any side effects.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **IEC 61131-3 Compatibility** – The conversion is performed via the standardized library function `F_DWORD_TO_DINT`, which is available for common PLC systems.
- **Unidirectional Adapters** – Both the input and output adapters are unidirectional; feedback or reverse-direction data is not supported.
- **No State Information** – The composite itself has no internal state variables; each conversion is atomic and time-independent.
- **Autonomy** – The function block can be used in a network configuration without manually integrating other function blocks.

State Overview

The composite function block does not have its own state machine. Its behavior results purely from the sequential processing of its internal connections:

- **Waiting for Event** – No activity occurs as long as no event is present at `AD_IN.E1`.
- **Conversion** – After an event is received, the result is converted once and output.
- **Completion** – The function block is immediately ready for the next event.

Application Scenarios

- **Gateway between systems with different data widths** – e.g., connecting a DWORD sensor value to DINT-based control logic.
- **Adapter Bridge in IEC 61499 Architectures** – When components need to communicate via standardized unidirectional adapters (AD/ADI).
- **Encapsulation of plausibility checks** – the DWORD value can be validated in upstream logic before conversion to the DINT format.

Comparison with similar function blocks

- **AD_TO_ADI** specializes in converting DWORD to DINT and follows a strict unidirectional adapter profile.
- Similar function blocks such as `AD_TO_AI` (DWORD to INT) or `ADI_TO_AD` (reverse conversion) exist for other data types but share the same design pattern.
- Unlike a generic conversion function block (e.g., `F_DWORD_TO_DINT` alone), the composite function block offers direct adapter integration, eliminating the need for users to create manual adapter interfaces.

Conclusion

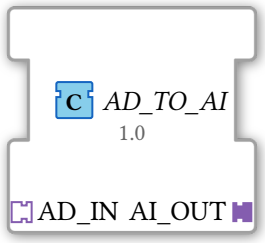
The composite function block **AD_TO_ADI** provides a simple, robust, and standards-compliant solution for converting DWORD to DINT adapter data. It reduces configuration effort within IEC 61499 networks and increases the reusability of function blocks from different sources. Thanks to its purely functional implementation without state issues, it is particularly suitable for time-critical and safety-related applications.

Numeric Index (unidirectional)

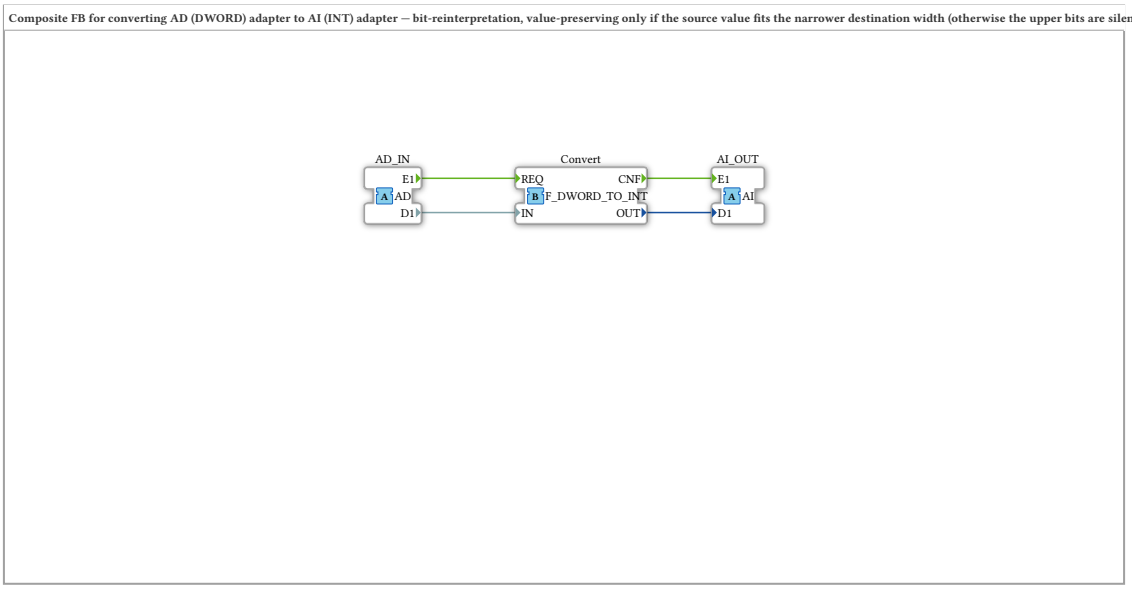
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AD_TO_AI

DWORD Adapter Input – *AD*



AI – INT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AD_TO_AI** acts as a converter between two different adapter interfaces: It converts an **AD adapter (DWORD)** into an **AI adapter (INT)**. The block is implemented as a composite function block and internally uses the IEC 61131-3 function `F_DWORD_TO_INT` to perform the actual type conversion. The goal is to facilitate the integration of components based on different data types while ensuring event-driven data transfer within the 4diac IDE.

Interface Structure

Event Inputs

The function block does not have direct event inputs at the top level. Event control is handled via the connected adapters:

- **AD_IN.E1** (Socket adapter event) – triggers the conversion.

Event Outputs

These are also provided exclusively via the adapters:

- **AI_OUT.E1** (Plug adapter event) – signals the completion of the converted data.

Data Inputs

- **AD_IN.D1** (DWORD) – The original data value supplied by the AD adapter.

Data Outputs

- **AI_OUT.D1** (INT) – The converted value output via the AI adapter.

Adapters

Adapter Type	Direction	Type	Description
Socket	Input	`adapter::types::unidirectional::AD`	Receives a DWORD value (via event E1 and data D1).
Plug	Output	`adapter::types::unidirectional::AI`	Provides the converted INT value (event E1, data D1).

Functionality

The function block operates in an event-driven manner:

1. An event at socket **AD_IN.E1** triggers the internal instance `Convert` (type `F_DWORD_TO_INT`) via its event input **REQ**.
2. Simultaneously, the data value **AD_IN.D1** (DWORD) is passed to the data input **IN** of `Convert`.
3. The function `F_DWORD_TO_INT` performs the type conversion from DWORD to INT.
4. After the calculation is complete, `Convert` sends an event at its output **CNF**, which is connected to the plug event **AI_OUT.E1**.
5. In parallel, the converted INT value is forwarded from the output **OUT** of `Convert` to the data output **AI_OUT.D1**.

Thus, after each incoming event at the socket, a converted data value is automatically made available on the plug.

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- **Composition:** The FB is a pure composite function block – it contains no internal logic but delegates the conversion to the standard function block `F_DWORD_TO_INT` from the IEC 61131 library.
- **Data Types:** DWORD is a 32-bit value, while INT is a 16-bit signed integer. Data loss can occur during conversion if the DWORD value is outside the INT range (−32768 … 32767) or does not fit within this range. The user must take this into account when using the function block.
- **Adapter Coupling:** The connection between the socket and the plug is established exclusively via the internal event and data flows. The adapters themselves are of type `unidirectional`, i.e., They only transfer data in one direction.
- **Package Structure:** The function block is provided in package `adapter::conversion::unidirectional` and imports the conversion function from `iec61131::conversion`.

State Overview

Since this is a composite function block without its own state machine, there are no explicit states. The function block behaves like a transparent converter:

- **Idle:** Waits for an event at the socket.
- **Processing:** After receiving an event, the conversion is started; no other events are active during this phase.
- **Done:** After the conversion, the output event is displayed; the function block returns to the idle state.

The internal function block `F_DWORD_TO_INT` has its own system-defined state logic, which is not visible due to its encapsulation within the composite function block.

Application Scenarios

- **Integration of Different Adapter Types:** A sensor delivers data as DWORD via an AD adapter, but downstream logic expects INT values via an AI adapter. The function block handles the type conversion seamlessly.
- **Heterogeneous System Architectures:** In systems where different components use different data types, `AD_TO_AI` enables simple and standardized connectivity.
- **Prototyping and Migration:** When switching from DWORD-based data processing to INT-based components, the function block can be used as a temporary or permanent converter.

Comparison with Similar Function Blocks

Other converter function blocks exist, such as `AD_TO_AI`, `AI_TO_AD`, `INT_TO_DWORD`, etc. This function block focuses specifically on the adapter level and operates entirely event-driven. Unlike simple data conversion function blocks (e.g., `F_DWORD_TO_INT`), it offers a pre-built adapter interface that enables direct coupling to adapter-oriented architectures without requiring the manual drawing of additional connections.

Other similar composite function blocks differ primarily in the direction (e.g., `AI_TO_AD` for reverse conversion) or in the data types used.

Conclusion

The composite function block **AD_TO_AI** is a useful tool for type conversion between DWORD and INT adapters. It encapsulates the conversion logic in a clean, event-driven interface, thus facilitating reusability and integration into existing 4diac projects. Its simple structure and the use of a proven standard function block make it reliable and easy to understand. However, users should always keep the limitations of the INT data type in mind to avoid unintentional data loss.

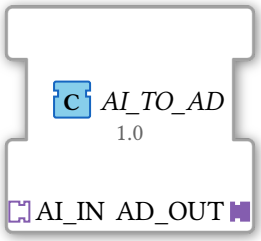
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

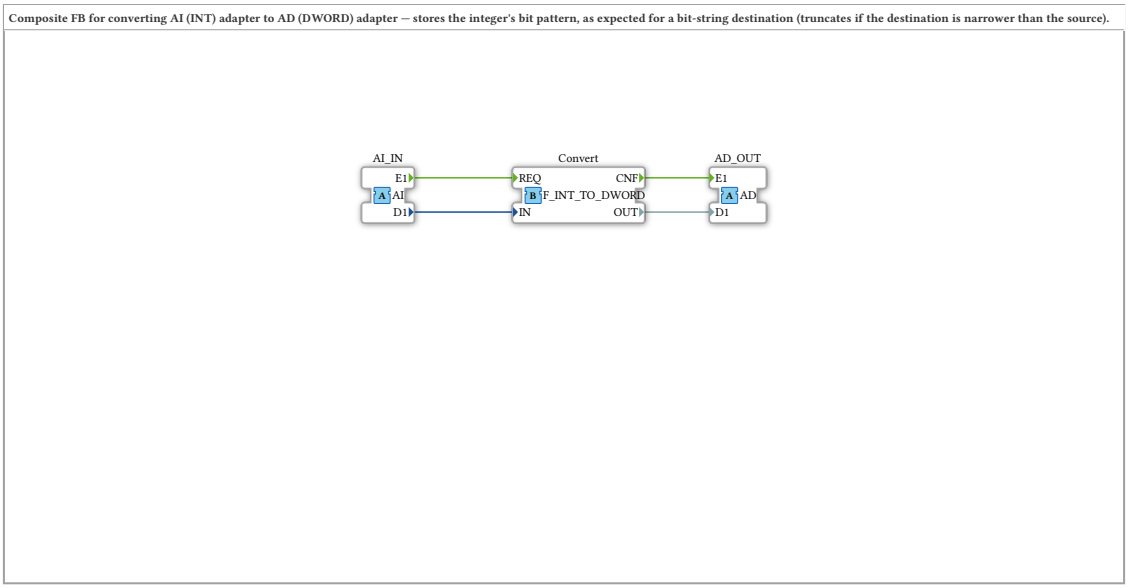
]

AI_TO_AD

INT Adapter Input – *AI*



AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The composite function block **AI_TO_AD** converts an INT-based adapter interface (AI) to a DWORD-based adapter interface (AD). It encapsulates the type conversion from `INT` to `DWORD`, thus enabling the seamless integration of different adapter types into control applications. The block is unidirectional and event-driven.

Interface Structure

Event Inputs

The block does not have direct event inputs. Event-driven triggering occurs via the adapter **AI_IN** (socket).

- **E1 (via AI_IN)**: Trigger event for converting the incoming INTEGER value.

Event Outputs

This function block has no direct event outputs. Conversion confirmation is provided via the **AD_OUT** adapter (plug).

- **E1 (via AD_OUT)**: Confirmation event after successful conversion.

Data Inputs

This function block has no direct data inputs. The INTEGER value to be converted is provided via the **AI_IN** adapter.

- **D1 (via AI_IN)**: Input value of type `INT`.

Data Outputs

This function block has no direct data outputs. The converted DWORD value is output via the **AD_OUT** adapter.

- **D1 (via AD_OUT)**: Output value of type `DWORD`.

Adapters

- **AI_IN** (Socket) – Input adapter type `adapter::types::unidirectional::AI`. It provides an INTEGER input (D1) and a trigger event (E1).
- **AD_OUT** (Plug) – Output adapter type `adapter::types::unidirectional::AD`. It provides a DWORD output (D1) and an acknowledgement event (E1).

Functionality

The function block is implemented as a composite. Internally, the IEC 61131-compliant function block **F_INT_TO_DWORD** is used.

Process:

1. An event at the input adapter **AI_IN.E1** triggers the data transfer.
2. The value of **AI_IN.D1** (INT) is forwarded to input `IN` of `F_INT_TO_DWORD`.
3. The internal function block performs the type conversion `INT → DWORD`.
4. The result is present at output `OUT` of `F_INT_TO_DWORD` and is passed to **AD_OUT.D1**.
5. Simultaneously, the event **AD_OUT.E1** is triggered to signal processing in the subsequent network.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Function Block**: The functionality is implemented by an embedded function block, so no separate state machine (ECC) is required.
- **Unidirectional**: Data flows exclusively from the input to the output adapter; feedback is not provided.
- **Type Conversion**: The function block uses the standardized IEC 61131 function block `F_INT_TO_DWORD`, ensuring compatibility with many automation systems.

- **Adapter Encapsulation**: The conversion logic is fully embedded in the adapter interfaces, simplifying reuse in different contexts.

State Overview

The function block does not have its own state machine. Internal processing is entirely controlled by the embedded function block `F_INT_TO_DWORD`, which implements simple event-driven data processing without states. Error behavior and timeouts are not defined.

Application Scenarios

- **Adapter Bridge**: When a system component provides an INTEGER adapter (AI), but the target component expects a DWORD adapter (AD).
- **Data Word Generation**: Conversion of analog values (e.g., 16-bit measured values) into a 32-bit data word for data transmission or storage.
- **Library Independence**: Encapsulation of the conversion function, so application developers do not need to directly access IEC 61131 function blocks.

Comparison with Similar Function Blocks

- **F_INT_TO_DWORD**: The direct IEC 61131 function provides pure type conversion without adapter embedding. `AI_TO_AD` adds the adapter interfaces, thus facilitating integration into adapter-based architectures.
- **DWORD_TO_INT (inverse)**: A corresponding function block for reverse conversion would be conceivable, but is not part of this implementation.
- **Custom Adapter Wrappers**: Other function blocks could provide similar conversions for other data types (e.g., `REAL_TO_DWORD`).

Conclusion

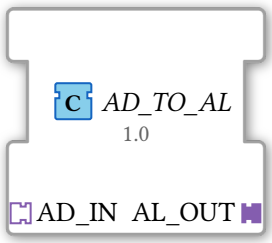
AI_TO_AD is a compact, adapter-oriented conversion function block that seamlessly integrates the conversion of INTEGER to DWORD values into existing control networks. By using standardized IEC 61131 function blocks and clearly defining interfaces via adapters, it is particularly well-suited for modular, reusable automation solutions.

Numeric Index (unidirectional)

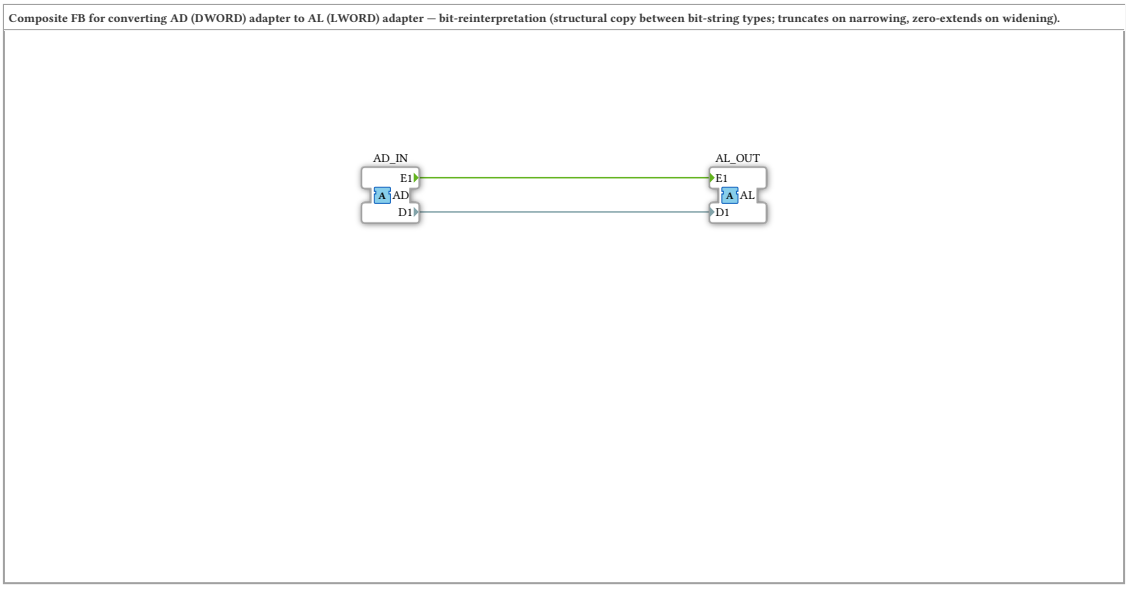
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AD_TO_AL

DWORD Adapter Input – *AD*



AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AD_TO_AL** function block is a composite function block that performs a unidirectional conversion of an AD adapter (DWORD) to an AL adapter (LWORD). It serves as a bridge between subsystems that use different adapter types but require the same logical connection. The function block passes event and data signals through one-to-one, thus enabling the seamless integration of heterogeneous components.

Interface Structure

The function block has only adapter interfaces; separate event or data inputs/outputs are not available.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Direction	Name	Comment
Socket (Input)	Input	`AD_IN`	DWORD adapter input of type `adapter::types::unidirectional::AD`
Plug (Output)	Output	`AL_OUT`	LWORD adapter output of type `adapter::types::unidirectional::AL`

Functionality

This function block is a simple forwarding block. Within the composite function block, the adapter channels are directly connected:

- **Event Connection:** `AD_IN.E1` ⟶ `AL_OUT.E1` Every incoming event at the AD adapter is passed unchanged to the AL adapter.

- **Data Connection:** `AD_IN.D1` ⟶ `AL_OUT.D1` The incoming data value (DWORD) is passed to the output as an LWORD without transformation.

Type conversion occurs implicitly through the coupling of the adapters; no explicit data type conversion takes place. The runtime environment ensures that the values are handled according to the adapter definition.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Composite Function Block:** The function block contains no internal logic (no state machine, no algorithms) and consists exclusively of internal connections.
- **Unidirectionality:** The adapters are designed to be unidirectional (only one data direction and one event direction).
- **Package Structure:** The function block (FB) is located in the package `adapter::conversion::unidirectional`.
- **Type Conversion:** AD (DWORD) and AL (LWORD) are defined as independent adapter types; the FB enables their interconnection without manual coding.
- **Metadata:** This function block originates from *HR Agrartechnik GmbH*, version 1.0, created on February 17, 2026.

State Overview

The function block does not have an internal state machine. Its state is identical to the state of the connected adapters. Passing through occurs without delay or logic.

Application Scenarios

- **System Integration:** Connecting components that use different adapter protocols (e.g., a controller with a DWORD interface and a bus coupler with an LWORD interface).
- **Test Environments:** Easy injection of test data by replacing adapters.
- **Modularization:** Separation of interface definition and data flow logic in large IEC 61499 projects.

Comparison with Similar Blocks

- `AL_TO_AD`: Performs the reverse conversion (LWORD ⟶ DWORD).
- Direct adapter conversion blocks (e.g., `BYTE_TO_WORD`) generally do not replace adapters but convert elementary data types.
- Other "composite" blocks in the `adapter::conversion` package offer similar forwarding for other adapter types (e.g., `AI_TO_AO`, `DI_TO_DO`).

Unlike function blocks with their own logic, `AD_TO_AL` is limited to simple pass-through, ensuring maximum transparency and the lowest possible latency.

Conclusion

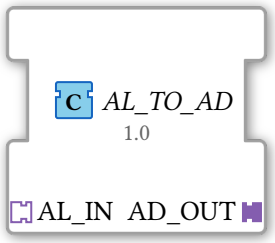
The `AD_TO_AL` function block is a lean yet valuable tool for adapter conversion in the 4diac IDE. It reduces manual wiring effort and increases the reusability of components that rely on different adapter standards. Thanks to its simple structure, it is robust, performant, and easy to understand – ideal for typical work with unidirectional adapter flows.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

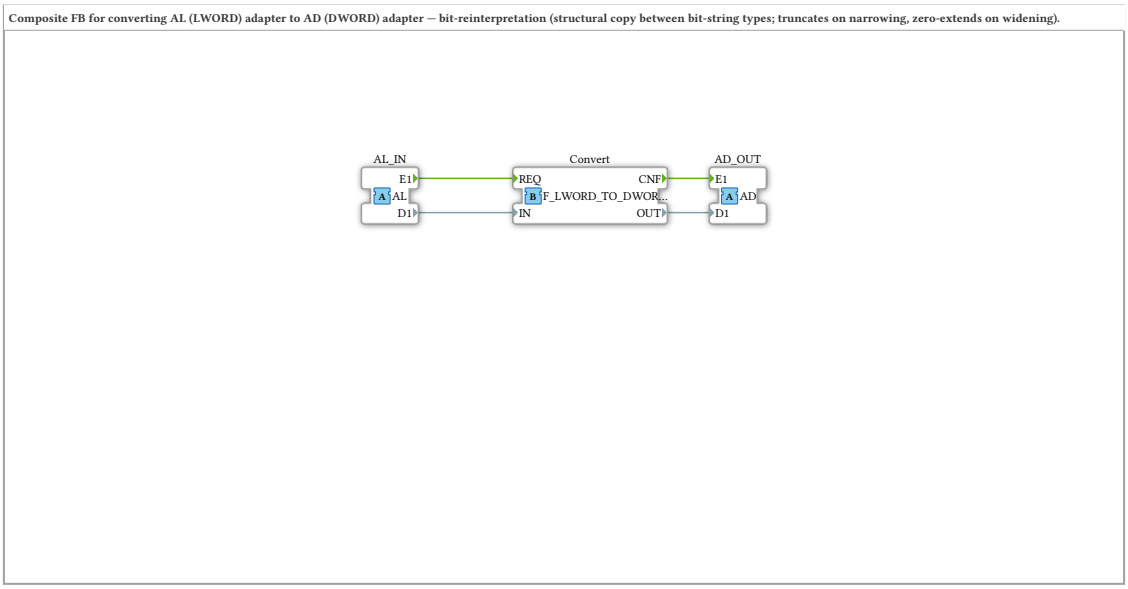
]

AL_TO_AD



LWORD Adapter Input – *AL*

AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **AL_TO_AD** function block is a composite function block that converts a unidirectional LWORD adapter (AL) into a unidirectional DWORD adapter (AD). It enables data conversion between two different adapter types within the 4diac IDE, based on the IEC 61499 standard.

Interface Structure

Event Inputs

- None (the FB has no dedicated event inputs; event control is handled via the AL_IN socket)

Event Outputs

- None (the FB has no dedicated event outputs; event forwarding is handled via the AD_OUT plug)

Data Inputs

- None (data is provided exclusively via the AL_IN socket)

Data Outputs

- None (data is output exclusively via the AD_OUT plug)

Adapter

Type	Name	Direction	Description
Socket	AL_IN	Input	Unidirectional LWORD adapter with one event input (`E1`) and one data input (`D1` of type `LWORD`)
Plug	AD_OUT	Output	Unidirectional DWORD adapter with one event output (`E1`) and one data output (`D1` of type `DWORD`)

Functionality

The function block receives an LWORD data word (`D1`) along with an event (`E1`) via the **AL_IN** socket. The event iterates through the internal function block **F_LWORD_TO_DWORD**, which performs the conversion according to the IEC 61131 standard. After successful conversion, the result is output as a DWORD (`AD_OUT.D1`) via the plug, accompanied by an event (`AD_OUT.E1`).

The event then passes through the internal function block **F_LWORD_TO_DWORD**, which performs the conversion according to the IEC 61131 standard. The internal network structure consists of:

- A **F_LWORD_TO_DWORD** block (from the library `iec61131::conversion`)
- Event connection: `AL_IN.E1` ⟶ `Convert.REQ` / `Convert.CNF` ⟶ `AD_OUT.E1`
- Data connection: `AL_IN.D1` ⟶ `Convert.IN` / `Convert.OUT` ⟶ `AD_OUT.D1`

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Standard Compliance**: The conversion is performed using the IEC 61131-compliant FB `F_LWORD_TO_DWORD`, which enables lossless conversion from 64-bit to 32-bit (low-order 32-bit) support.
- **Unidirectional Adapters**: Both the input and output adapters are unidirectional, reducing complexity and ensuring a clearly defined data flow direction.
- **Event-driven Communication**: Conversion is triggered by an event at the input adapter; the result is also passed on in an event-driven manner.
- **Embedded in Composite Function Blocks**: By encapsulating the function block as a composite block, the function block can be easily integrated into larger networks.

State Overview

The **AL_TO_AD** function block does not have its own state machine, as it is implemented as a composite function block. The state logic is fully implemented by the internal **F_LWORD_TO_DWORD** function block:

1. **Wait** – No event is present at the input.
2. **Convert** – After receiving an event, the conversion of the LWORD data is performed (one clock cycle).
3. **Output** – The DWORD result and an acknowledgment event are provided at the output adapter.

Application Scenarios

- **System Integration** – When a component (e.g., a PLC) provides LWORD data, but a downstream module expects DWORD data.
- **Adapter Bridge** – In heterogeneous networks where devices use different data types for the same physical value.
- **Data Preparation** – Preprocessing of 64-bit values that are to be further processed on a 32-bit basis (e.g., for measured values or counters).

Comparison with Similar Function Blocks

- **WORD_TO_DWORD** – Converts 16-bit words to 32-bit double words, but on simple data types, not on adapters.
- **Direct Adapter Coupling** – Without conversion, both systems would have to use the same adapter type (either AL or AD), which is often not the case.
- **F_LWORD_TO_DWORD (Isolated)** – This function block operates only at the data layer; the adapter connection must be established separately. **AL_TO_AD** simplifies handling by encapsulating the adapter logic and conversion.

Conclusion

The **AL_TO_AD** function block provides a lean and standards-compliant solution for converting LWORD to DWORD adapter data. Combining adapter and conversion logic in a single composite function block increases reusability and simplifies network creation. It is particularly well-suited for integrating various IEC 61499 components and contributes to the modularity of industrial automation solutions.

AL_TO_AD —

Related topic subpages on ms-muc-docs.de

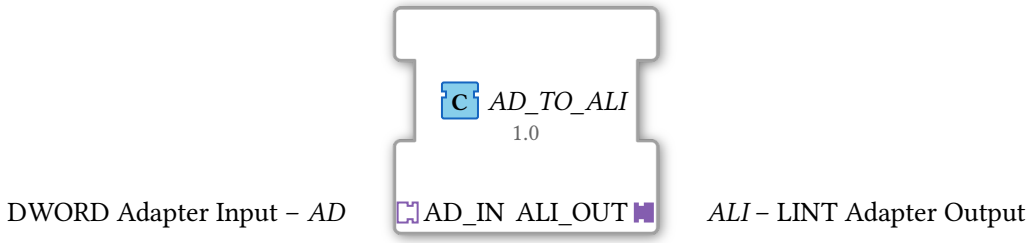
- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

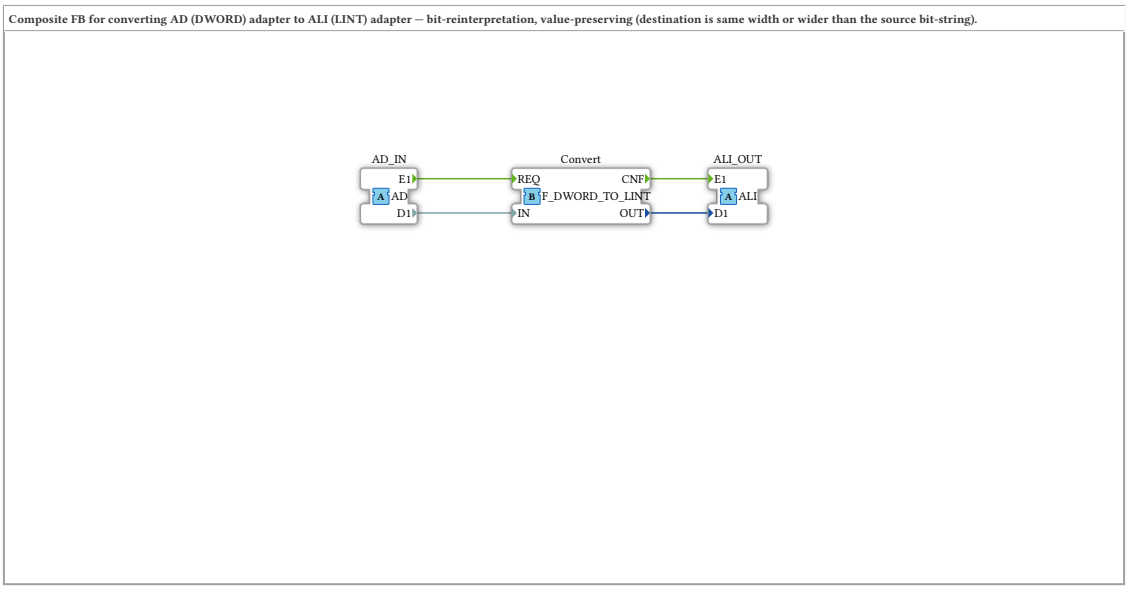
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AD_TO_ALI





• – ○ • – ○ • – ○ •

The **AD_TO_ALI** is a composite function block for the unidirectional conversion of a DWORD value (via an AD adapter) to a LINT value (via an ALI adapter). The conversion is performed using the integrated standard function block `F_DWORD_TO_LINT`. The block enables simple and standardized integration of components that exchange different data types via adapter interfaces. No separate event inputs are available. Event control is handled via the **AD_IN** adapter (event *E1*).

No separate event outputs are available. Output events are provided via the **ALI_OUT** adapter (event *E1*).

No separate data inputs are available. The input data (DWORD) is obtained via the **AD_IN** adapter (Data *D1*).

No separate data outputs are available. The output data (LINT) is output via the **ALI_OUT** adapter (Data *D1*).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Adapter	Type	Direction	Description
AD_IN	`adapter::types::unidirectional::AD`	Socket	Receives a DWORD value (*D1*) and an input event (*E1*).
ALI_OUT	`adapter::types::unidirectional::ALI`	Plug	Sends the converted LINT value (*D1*) and an output event (*E1*).

Functionality

The function block waits for an event at the **AD_IN** adapter (event *E1*). As soon as this event arrives, the currently incoming DWORD value (Data *D1*) is read from the adapter and passed to the internal function block **Convert** (type `F_DWORD_TO_LINT`). This block performs the data type conversion. After successful conversion, the resulting LINT value is passed to the **ALI_OUT** adapter (Data *D1*), and an event (*E1*) is triggered for confirmation.

- **Pure Composite Logic**: The function block does not contain its own state machine but implements the conversion exclusively through the interconnection of the embedded function block `F_DWORD_TO_LINT`.
- **Standard Compliance**: The conversion function used complies with IEC 61131-3 (type conversion `DWORD_TO_LINT`).
- **Synchronization**: Events and data are forwarded from input to output within a single execution cycle (E1 → REQ → CNF → E1).

As a composite function block, **AD_TO_ALI** does not have its own state machine. Its behavior is determined solely by its internal interconnection and the states of the function block `F_DWORD_TO_LINT`. A graphical state overview is not provided.

- **Industrial Automation**: Conversion of DWORD-based sensor data (e.g., counter readings) into LINT values for further processing in systems with 64-bit integers.
- **Data Integration**: Connecting components that use different adapter protocols via a standardized conversion interface.

Analog conversion function blocks such as `AD_TO_ALI` exist for other data types (e.g., `AD_TO_ALI` for DWORD→LINT, `AI_TO_ALI` for INT→LINT). This function block specifically covers the conversion from **DWORD** to **LINT** and is optimized for this combination. It does not replace the general `F_DWORD_TO_LINT` function, but rather encapsulates it in an adapter-based interface.

The **AD_TO_ALI** offers a clean, reusable solution for converting DWORD adapters to LINT adapters. Thanks to its composite structure, the logic remains transparent and easily integrated into existing 4diac projects. It is particularly suitable for scenarios requiring a uniform adapter interface but needing to combine different data types.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).

State Overview

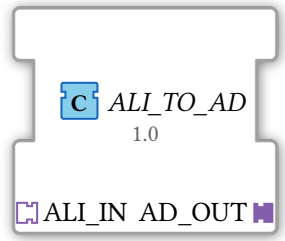
Application Scenarios

Comparison with Similar Function Blocks

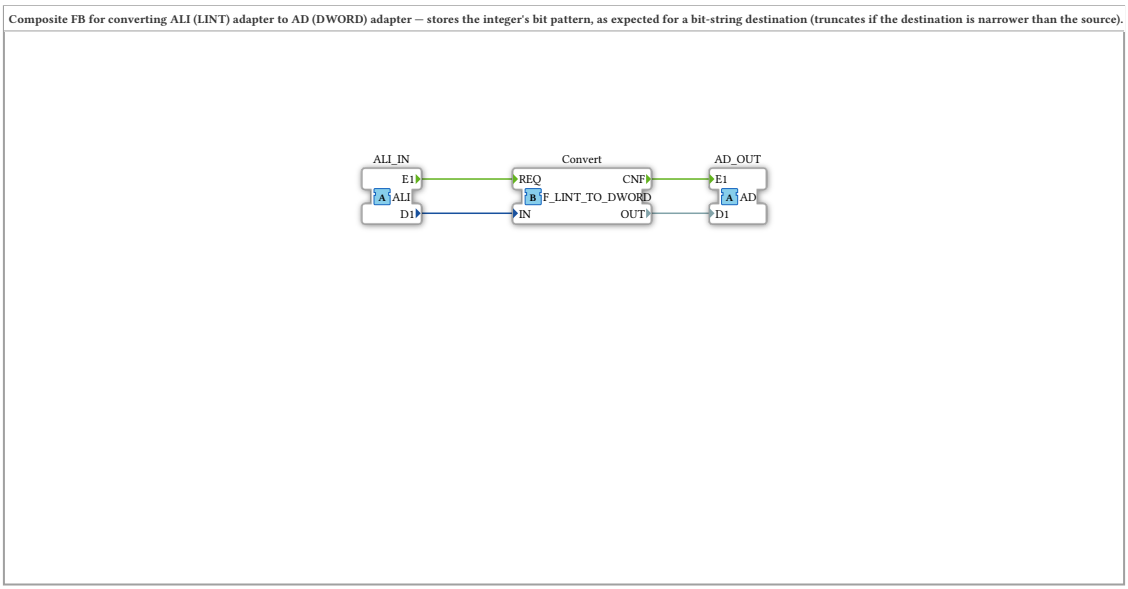
Conclusion

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

ALI_TO_AD



LINT Adapter Input – *ALI* *AD* – DWORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `ALI_TO_AD` is a composite block for converting a LINT adapter interface (ALI) to a DWORD adapter interface (AD). It serves as a unidirectional translator between two different data types within an adapter-based communication structure.

Interface Structure

The block has no independent event or data inputs/outputs. All communication takes place exclusively via predefined adapters.

Event Inputs

None. Event control is handled via the incoming socket adapter `ALI_IN`.

Event Outputs

None. The output events are provided via the outgoing plug adapter `AD_OUT`.

Data Inputs

None. Input data is transmitted via the socket adapter `ALI_IN`.

Data Outputs

None. Output data is transmitted via the plug adapter `AD_OUT`.

Adapters

Direction	Name	Type	Description
Socket	ALI_IN	`adapter::types::unidirectional::ALI`	Input adapter of type LINT
Plug	AD_OUT	`adapter::types::unidirectional::AD`	Output Adapter of Type DWORD

Functionality

Inside the function block, a single instance of the function block `iec61131::conversion::F_LINT_TO_DWORD` is used. The event and data connections are wired as follows:

1. An event from socket `ALI_IN.E1` triggers the input `REQ` of the converter.
2. The converter reads the LINT value from input `ALI_IN.D1` and converts it to a DWORD value.
3. After successful conversion, the converter generates the confirmation event `CNF`, which is forwarded to plug `AD_OUT.E1`.
4. Simultaneously, the converted DWORD value is transferred from output `Convert.OUT` to data output `AD_OUT.D1`.

The entire conversion process is event-driven: An incoming event triggers the conversion, and the result is acknowledged with an output event.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- The module is fully implemented as a composite and requires no dedicated runtime logic.
- The internal converter `F_LINT_TO_DWORD` is an IEC 61131-3 module that performs a bit-perfect conversion of a 64-bit integer value (LINT) to a 32-bit double-word value (DWORD).
- The adapters are defined as unidirectional, meaning that data flows exclusively from the input to the output adapter.
- This function block is designed for use in automation systems that utilize adapter interfaces according to the IEC 61499 standard.

State Overview

The function block does not have its own explicit state machine. Its behavior is entirely determined by event propagation and the internal logic of the converter block. From the user's perspective, it behaves like a transparent data conversion that is executed once for each incoming event.

Application Scenarios

- **Integration of Heterogeneous Field Devices** – When a device or subnetwork provides data in LINT format (e.g., as a counter value or timestamp), but the target system expects DWORD values.
- **Adapter Bridge Between Subsystems** – In a distributed control system according to IEC 61499, `ALI_TO_AD` can be used to connect two subsystems with different adapter definitions.
- **Data Format Adaptation in Migration Projects** – When modernizing systems where older components still provide LINT adapters, but newer components require DWORD adapters.

Comparison with Similar Components

- **`INT_TO_DWORD`** – Converts a 16-bit integer value to DWORD, but is not offered at the adapter level.
- **`DWORD_TO_LINT`** – Reverses the conversion described here and converts DWORD back to LINT.
- **Custom Combination Components** – Instead of a hardwired adapter converter, the conversion could also be performed manually using a `F_LINT_TO_DWORD` and separate adapters. The composite component offers a compact and reusable solution.

Conclusion

ALI_TO_AD` is a simple yet useful composite component that enables the unidirectional conversion of a LINT adapter to a DWORD adapter. It allows for a clean separation of data types at the adapter level and

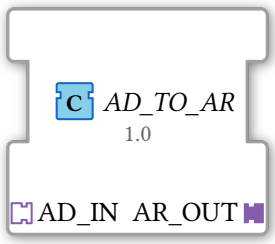
simplifies the integration of different systems in an IEC 61499 environment. The use of a standardized IEC 61131 converter ensures defined and reproducible data conversion.

Numeric Index (unidirectional)

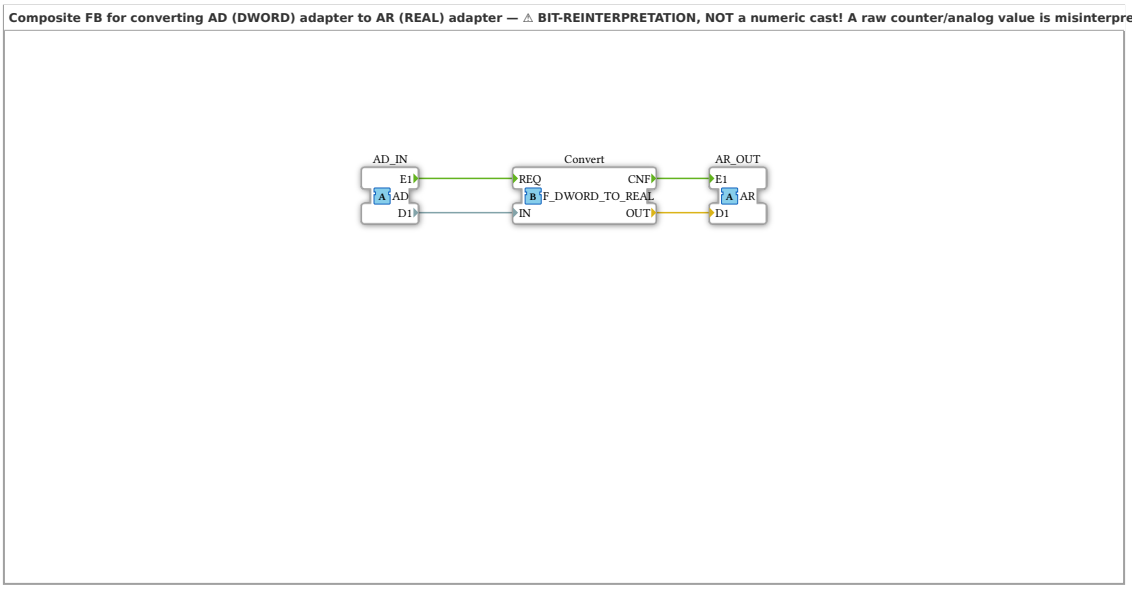
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AD_TO_AR

DWORD Adapter Input – *AD*



AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AD_TO_AR** function block is a composite function block that maps a DWORD value from a unidirectional AD adapter (socket) onto a REAL value at an AR adapter (plug).

> ⚠️ **Warning — not a numeric value conversion:** `AD_TO_AR` internally uses `F_DWORD_TO_REAL`, which in the FORTE core is a pure IEEE754 **bit-reinterpretation** for bit-string source types (BYTE/WORD/DWORD/LWORD), not a numeric cast. A raw counter or analog value such as `DWORD#2048` will therefore **not** become `REAL#2048.0`, but a meaningless value near zero. For an actual numeric DWORD→REAL conversion, see `AD_TO_AR_NUM` (via UDINT). `AD_TO_AR` is only correct when `AD_IN` is already a bit pattern meant to be interpreted as REAL (e.g. the result of `F_REAL_TO_DWORD`).

Interface Structure

The function block has no direct (individual) event or data inputs/outputs. All communication takes place via two adapter interfaces:

Event Inputs

- *No direct event inputs* – event control is handled via the incoming adapter.

Event Outputs

- *No direct event outputs* – event output is handled via the outgoing adapter.

Data Inputs

- *No direct data inputs* – data is received via the incoming adapter.

Data Outputs

- *No direct data outputs* – data is output via the outgoing adapter.

Adapters

Name	Type	Direction	Description
AD_IN	`adapter::types::unidirectional::AD`	Socket	Receives an event (E1) and a DWORD value (D1) for conversion.
AR_OUT	`adapter::types::unidirectional::AR`	Plug	Sends an event (E1) with the converted REAL value (D1).

Functionality

The function block operates as an event-driven pipeline:

1. An incoming event at socket **AD_IN.E1** triggers the conversion.
2. The data value **AD_IN.D1** (DWORD) is passed to the internal function block `F_DWORD_TO_REAL`.
3. After successful conversion, the internal function block signals an output event (**CNF**).
4. This event is forwarded to plug **AR_OUT.E1**, and simultaneously, the converted REAL value is output via **AR_OUT.D1**.

`F_DWORD_TO_REAL` merely reinterprets the DWORD's 32-bit pattern as an IEEE754 `REAL` (a bit copy) — **no** numeric value conversion takes place.

Technical Features

- ⚠️ **Bit-reinterpretation, not numeric conversion**: See the warning in the introduction. This affects all bit-string source types (`AB`/BYTE, `AW`/WORD, `AD`/DWORD, `AL`/LWORD) converting to `AR`/REAL or `ALR`/LREAL — the only other instance in this library is `AL_TO_ALR` (LWORD→LREAL).
- **Composite Function Block**: The conversion logic is implemented entirely within an internal network consisting of a single function block.
- **Type Conversion via Adapter**: This function block enables the connection of components that communicate exclusively via adapter interfaces, eliminating the need for additional manual conversions.
- **Standard Compliance**: Utilizes the IEC 61131 library function `F_DWORD_TO_REAL`, making it portable and well-tested.
- **Unidirectional Adapters**: Both input and output adapters are unidirectional, meaning data flows in only one direction.

State Overview

The function block does not have its own state machine. Processing is strictly causal: An event at the input immediately triggers the conversion and output. There are no delays or intermediate states.

Application Scenarios

- **Bit pattern pass-through**: `AD_IN` already carries a bit pattern meant to be interpreted as REAL (e.g. the result of an `F_REAL_TO_DWORD` elsewhere in the network, or deserialized raw float data from a fieldbus/protocol).
- **NOT suitable** for raw counter, analog, or other integer values meant to carry the same numeric value as REAL — use `AD_TO_AR_NUM` for that.

Comparison with Similar Function Blocks

- **`AD_TO_AR_NUM`** (numeric, DWORD→UDINT→REAL): the safe replacement when an actual numeric value is meant.
- **`AI_TO_AR`/`ADI_TO_AR`/`AUDI_TO_AR`** (INT/DINT/UDINT→REAL): already numerically correct, since their source types are `ANY_INT` (not `ANY_BIT`) — no trap.
- **`AL_TO_ALR`** (LWORD→LREAL): the same bit-reinterpretation trap, just with the 64-bit counterparts.

Conclusion

The **AD_TO_AR** block offers a compact way to expose a DWORD bit pattern as REAL via adapter interfaces — **but not a numeric value conversion**. For raw counter or analog values meant to carry the same numeric value in REAL, use AD_TO_AR_NUM.

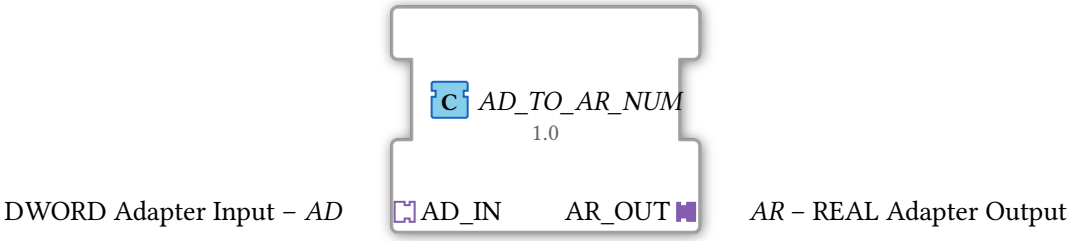
📖 Background

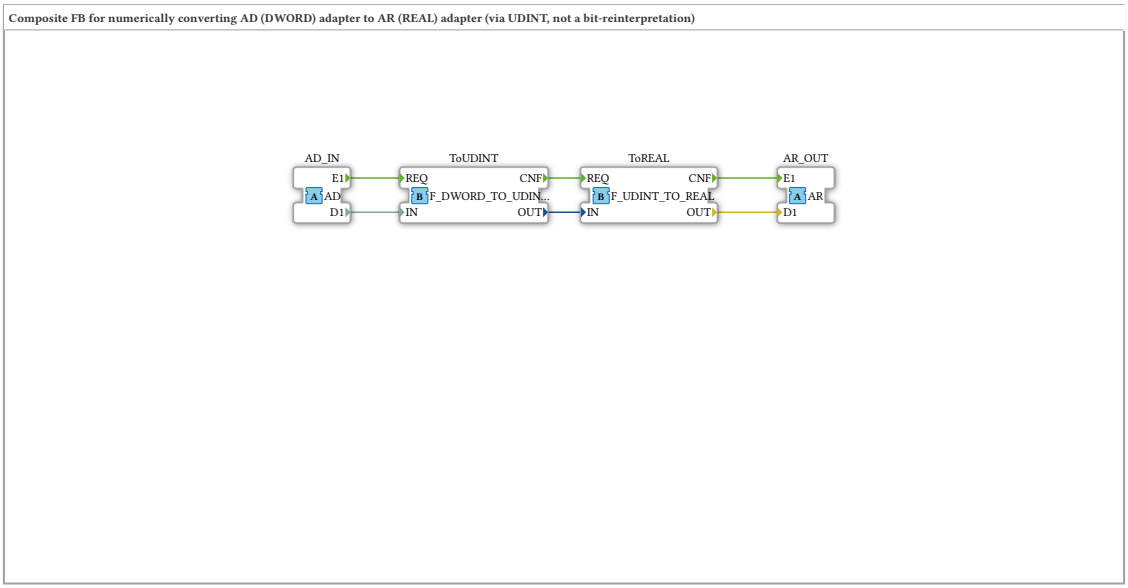
- Numeric vs. bitwise: the FORTE conversion trap

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AD_TO_AR_NUM





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_TO_AR_NUM** is the numerically correct alternative to AD_TO_AR: it converts a DWORD value from a unidirectional AD adapter (socket) into a REAL value output via an AR adapter (plug), via an internal UDINT step — an **actual value conversion**, not a bit-reinterpretation.

Why not just use AD_TO_AR?

`AD_TO_AR` internally uses `F_DWORD_TO_REAL`, which in the FORTE core is a pure IEEE754 bit-reinterpretation for bit-string source types (BYTE/WORD/DWORD/LWORD), not a numeric cast. A raw counter or analog value such as `DWORD#2048` would therefore **not** become `REAL#2048.0`, but a meaningless value near zero (see AD_TO_AR and `AD_TO_AR_TODO.md` for the derivation).

Interface Structure

The function block has no direct (individual) event or data inputs/outputs. All communication takes place via two adapter interfaces:

Adapters

Name	Type	Direction	Description
AD_IN	adapter::types::unidirectional::AD	Socket	Receives an event (E1) and a DWORD raw value (D1) for conversion.
AR_OUT	adapter::types::unidirectional::AR	Plug	Sends an event (E1) with the numerically converted REAL value (D1).

Functionality

The function block works in two stages instead of a single (dangerous) one:

1. An incoming event at socket **AD_IN.E1** triggers the conversion.
2. The data value **AD_IN.D1** (DWORD) is passed to `ToUDINT` (`F_DWORD_TO_UDINT`) — a bit-reinterpretation DWORD⟶UDINT that is **valid** here, since both types share the same 32-bit unsigned-integer representation.
3. `ToUDINT.CNF` triggers `ToREAL` (`F_UDINT_TO_REAL`) — a **real numeric cast** that correctly converts the UDINT numeric value to REAL.
4. The result is output via plug **AR_OUT.E1**/**AR_OUT.D1**.

Technical Features

- **Identical interface to `AD_TO_AR`**: `AD_TO_AR_NUM` is a drop-in replacement — same socket/plug types (`AD`/`AR`), easily swapped in once it's clear a numeric value (not a bit pattern) is intended.
- **Two-stage DWORD⟶UDINT⟶REAL chain**: the same chain already wired by hand in several exercises (`Uebung_028a_AR` and others), packaged here as a single block.
- **No precision loss in the relevant value range**: `UDINT` (32 bit, max. 4,294,967,295) transfers into `REAL` (IEEE754 single precision, 24-bit mantissa) without loss as long as the value stays within the exactly representable integers (up to 2^{24} = 16,777,216) — not an issue for typical analog/counter raw values (e.g. 0–64255 or 0–4095).

Application Scenarios

- **Raw analog values**: An analog input or counter delivers a DWORD raw value (e.g. 0–4095, 0–64255) that should be further processed as an actual numeric value in REAL (scaling, display, OPC-UA publish).
- **Replacement for the manual `AD_TO_AUDI` + `AUDI_TO_AR` chain**: where two separate adapter instances were previously needed in the SubApp network, one now suffices.

Comparison with Similar Function Blocks

- `AD_TO_AR`: the bit-reinterpretation variant — use only when `AD_IN` is already a bit pattern meant to be interpreted as REAL.
- **`AI_TO_AR`/`ADI_TO_AR`/`AUDI_TO_AR`**: already numerically correct, since their source types are `ANY_INT` — if the raw value is already available as an INT/DINT/UDINT adapter, no `_NUM` variant is needed.

Conclusion

AD_TO_AR_NUM closes the gap between the tempting-but-dangerous `AD_TO_AR` and the actually needed numeric DWORD⟶REAL conversion — as a single, drop-in-compatible block instead of a manually wired two-block chain.

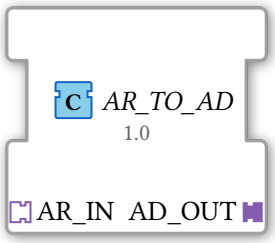
📖 Background

- Numeric vs. bitwise: the FORTE conversion trap

🌐 Related topic subpages on ms-muc-docs.de

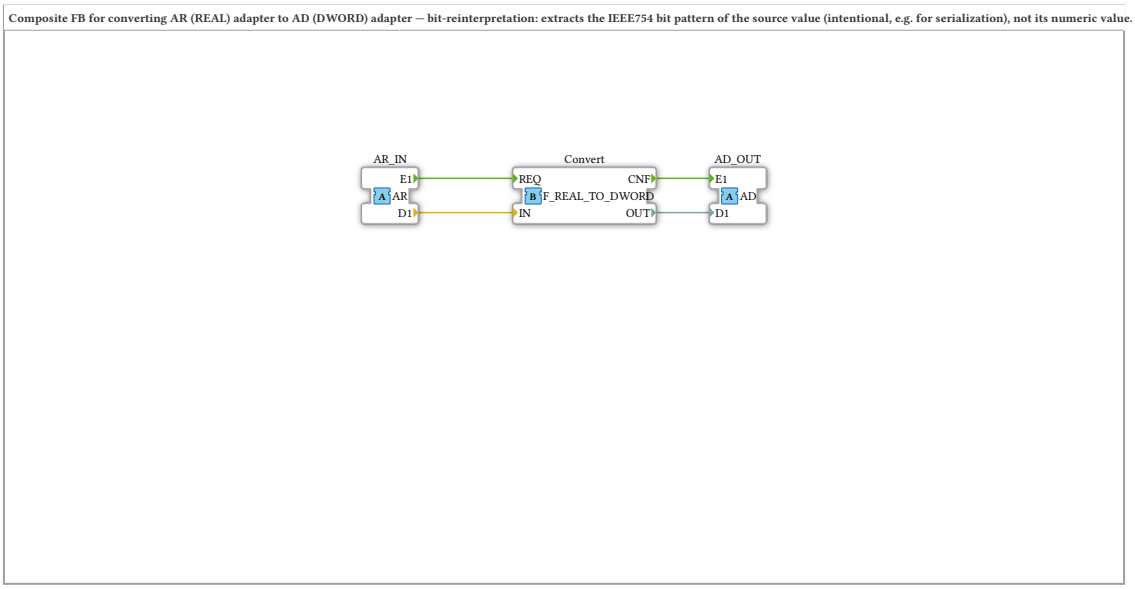
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AR_TO_AD



REAL Adapter Input – *AR*

AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AR_TO_AD` serves as a composite function block (FB) for converting a REAL value received via a unidirectional AR adapter into a DWORD value output via a unidirectional AD adapter. The conversion is performed according to IEC 61131-3 using the function block `F_REAL_TO_DWORD`. The FB encapsulates the necessary event and data connections and allows for easy integration into existing adapter-based communication structures.

Interface Structure

Event Inputs

Input	Type	Description
`AR_IN.E1`	Adapter Event	Receives the trigger event from the AR adapter, which starts the conversion.

Event Outputs

Output	Type	Description
`AD_OUT.E1`	Adapter Event	Sends an event to the AD adapter as soon as the converted DWORD value is available.

Data Inputs

Input	Type	Description
`AR_IN.D1`	REAL	The REAL value to be converted, provided by the AR adapter.

Data Outputs

Output	Type	Description
AD_OUT.D1	DWORD	The converted DWORD value that is passed to the AD adapter.

Adapters

The FB has a **socket** of type adapter::types::unidirectional::AR (labeled AR_IN) and a **plug** of type adapter::types::unidirectional::AD (labeled AD_OUT). The actual input and output signals are accessible via the event and data pins of these adapters.

Functionality

1. The function block (FB) waits for an event at the event input AR_IN.E1.
2. Simultaneously, the current value at the data input AR_IN.D1 (REAL) is retrieved.
3. Internally, the function block F_REAL_TO_DWORD (from the IEC 61131-3 library) is called, which performs the conversion.
4. After successful conversion, the resulting DWORD value is assigned to the data output AD_OUT.D1.
5. Subsequently, an event is output at the output AD_OUT.E1 to notify the connected Active Directory adapter.

The entire process occurs synchronously in a single step; that is, the FB is purely event-driven and does not maintain any internal states.

Technical Features

- **Bit-reinterpretation**: extracts the IEEE754 bit pattern of the source value (intentional, e.g. for serialization), not its numeric value.
- **Composite FB:** The logic is implemented exclusively through the combination of the internal F_REAL_TO_DWORD function block. There is no separate state machine or complex algorithms.
- **Adapter Coupling:** The function block encapsulates the conversion at the adapter level, allowing it to be integrated into adapter-based architectures (e.g., distributed systems with 4diac) without additional modifications.
- **IEC 61131-3 Compliance:** The actual conversion is performed by a standardized function block, which increases portability and standards compliance.
- **Package Name:** The function block is provided under the package adapter::conversion::unidirectional and imports iec61131::conversion::F_REAL_TO_DWORD.

State Overview

Since this is a combinational composite function block, it does not have an internal state machine. The functionality is purely event-driven and is executed once for each event at the input.

Application Scenarios

- **Data Protocol Adaptation:** When a REAL value (e.g., a measured value) can only be transmitted as a DWORD (e.g., in many fieldbus protocols) via a bus system.
- **Adapter Bridge:** In systems that use both REAL and DWORD adapters, this function block can serve as a simple conversion unit between the two adapter types.
- **Reusable Conversion:** Due to its encapsulation, the conversion can be used multiple times in different networks without having to manually establish the connections each time.

Comparison with Similar Function Blocks

- **F_REAL_TO_DWORD:** The direct function block from IEC 61131-3 offers the same conversion but requires manual wiring of event and data lines. AR_TO_AD encapsulates this in an adapter interface, simplifying integration into adapter networks.
- **AR_TO_SINT / AR_TO_INT:** Comparable conversions exist for other integer data types, but not as adapter versions. AR_TO_AD is specifically designed for DWORD and adapters.
- **User-defined composite function blocks:** In principle, any user could create similar encapsulation. However, this function block offers a standardized solution with a clear package structure.

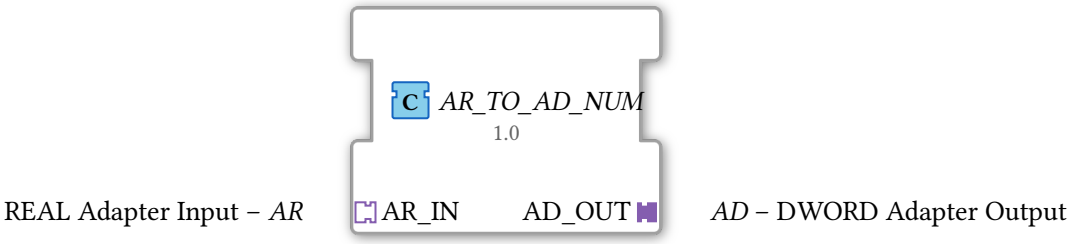
Conclusion

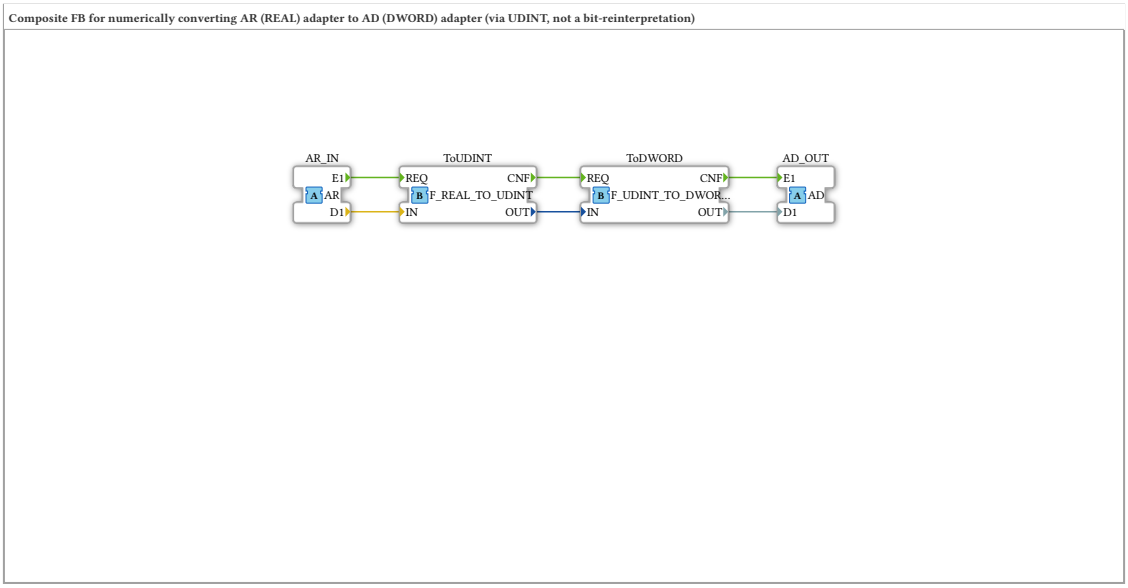
The `AR_TO_AD` function block is a specialized composite function block that enables the conversion of a REAL value to a DWORD value at the adapter level. It is characterized by its ease of use, standards compliance, and seamless integration into unidirectional adapter communication. The use of the proven IEC 61131-3 module `F_REAL_TO_DWORD` ensures reliable and portable conversion.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AR_TO_AD_NUM





• – ◦ • – ◦ • – ◦ •

Introduction

`AR_TO_AD_NUM` is a composite function block that converts a REAL adapter input into a DWORD adapter output using a numerically correct conversion path. Instead of reinterpreting the REAL value as its IEEE 754 bit pattern, it first converts REAL to UDINT and then moves the resulting unsigned 32-bit integer into a DWORD. This makes it suitable for applications where a numeric setpoint such as `REAL#50.0` should become `DWORD#50` rather than the raw floating-point bit pattern of 50.0.

The block is a drop-in-compatible alternative to `AR_TO_AD` for adapter-based connections that require numeric REAL-to-DWORD conversion, and it forms the counterpart to `AD_TO_AR_NUM` for the reverse direction.

Interface Structure

Event Inputs

None.
`AR_TO_AD_NUM` has no directly exposed event inputs. The input event is received through the `AR_IN` adapter.

Event Outputs

None.
The output event is delivered through the AD_OUT adapter.

Data Inputs

None.
The input data is received through the AR_IN adapter.

Data Outputs

None.
The output data is delivered through the AD_OUT adapter.

Adapters

Role	Name	Type	Description
Socket	AR_IN	adapter::types::unidirectional::AR	REAL adapter input. Supplies the REAL value on AR_IN.D1 and the corresponding event on AR_IN.E1.
Plug	AD_OUT	adapter::types::unidirectional::AD	DWORD adapter output. Provides the converted DWORD value on AD_OUT.D1 and the completion event on AD_OUT.E1.

Functionality

The internal network consists of two IEC 61131-2 conversion function blocks:

1. F_REAL_TO_UDINT — performs a real numeric conversion from REAL to UDINT.
2. F_UDINT_TO_DWORD — transfers the UDINT bit pattern to DWORD.

When an event arrives through AR_IN.E1, the REAL value from AR_IN.D1 is passed to ToUDINT.IN. The F_REAL_TO_UDINT block converts the value and confirms with CNF. This confirmation triggers ToDWORD.REQ, passing the UDINT result as input. F_UDINT_TO_DWORD then provides the equivalent DWORD representation and confirms with CNF, which causes AD_OUT.E1 to be issued with the result available on AD_OUT.D1.

This two-step conversion chain guarantees that the resulting DWORD carries the same numeric value as the original REAL input, rather than the raw IEEE 754 representation of that value.

Technical Features

- Composite FB implemented entirely through an internal function block network.
- Uses standard IEC 61131 conversion function blocks.
- Converts REAL to DWORD through a UDINT intermediate step.
- Avoids the IEEE 754 bit-reinterpretation behavior of a direct REAL-to-DWORD conversion.
- Unidirectional event flow: input adapter event triggers the conversion pipeline and produces one output adapter event.
- Uses the same adapter types as AR_TO_AD, allowing it to be used as a drop-in replacement in adapter-based networks.
- No custom state machine or internal algorithms are required.

State Overview

AR_TO_AD_NUM has no explicitly defined ECC state machine because it is a composite function block. Its behavior can be described as a three-stage request/confirmation pipeline:

Stage	Active Element	Description
1	ToUDINT	AR_IN.E1 triggers ToUDINT.REQ; AR_IN.D1 is passed as the REAL input.
2	ToDWORD	ToUDINT.CNF triggers ToDWORD.REQ; the converted UDINT value is passed to ToDWORD.IN.
3	AD_OUT	ToDWORD.CNF triggers AD_OUT.E1; the resulting DWORD value is placed on AD_OUT.D1.

The block is idle until an event arrives from the connected REAL adapter.

Application Scenarios

AR_TO_AD_NUM is useful wherever a REAL number must be delivered as the same numeric value in DWORD format, particularly when the DWORD is interpreted by a device or driver as an integer-like value.

Typical use cases include:

- **PWM duty cycle output**: A controller computes a duty cycle as REAL#50.0 and must send DWORD#50 to a PWM output such as logiBUS_QDA_PWM.
- **Numeric setpoint conversion**: A REAL setpoint from an adapter-based input must be converted into a DWORD value for a downstream integer-oriented interface.
- **Replacing manual conversion chains**: AR_TO_AD_NUM packages the equivalent of AR_TO_AUDI followed by AUDI_TO_AD into a single function block.
- **Interfacing with unidirectional adapters**: The block is designed for use in the adapter::conversion::unidirectional package and fits into adapter-based data flow architectures.

For the opposite direction, AD_TO_AR_NUM provides the corresponding numeric DWORD-to-REAL conversion.

Comparison with Similar Blocks

Approach	Conversion Semantics	Recommended Use
AR_TO_AD_NUM	REAL → UDINT → DWORD; numeric value is preserved	Numeric setpoints, PWM duty cycles, integer-oriented DWORD outputs
AR_TO_AD	REAL → DWORD using IEEE 754 bit reinterpretation	Serializing a REAL value as its floating-point bit pattern for transport to a matching DWORD-to-REAL block
Manual AR_TO_AUDI + AUDI_TO_AD	Same numeric semantics as AR_TO_AD_NUM, but requires two separate blocks and explicit wiring	When a conversion chain must be made visible or extended in the network
AD_TO_AR_NUM	DWORD → UDINT → REAL; reverse numeric conversion	Converting a numeric DWORD setpoint back into a REAL value

The key difference is whether the REAL value is treated as a number or as a bit pattern. AR_TO_AD_NUM always preserves the numeric value, while a direct REAL-to-DWORD conversion preserves the IEEE 754 bit layout.

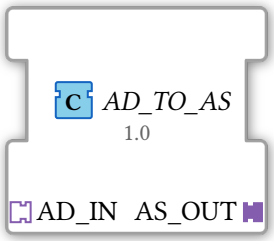
Conclusion

AR_TO_AD_NUM provides a safe and semantically correct way to convert a REAL adapter value into a DWORD adapter value. By routing the conversion through UDINT, it ensures that a numeric value such as REAL#50.0 becomes DWORD#50 and not the raw binary representation of the REAL number. This makes it particularly valuable in automation applications where DWORD values are used as setpoints, duty cycles, or other integer-interpreted data.

Numeric Index (unidirectional)

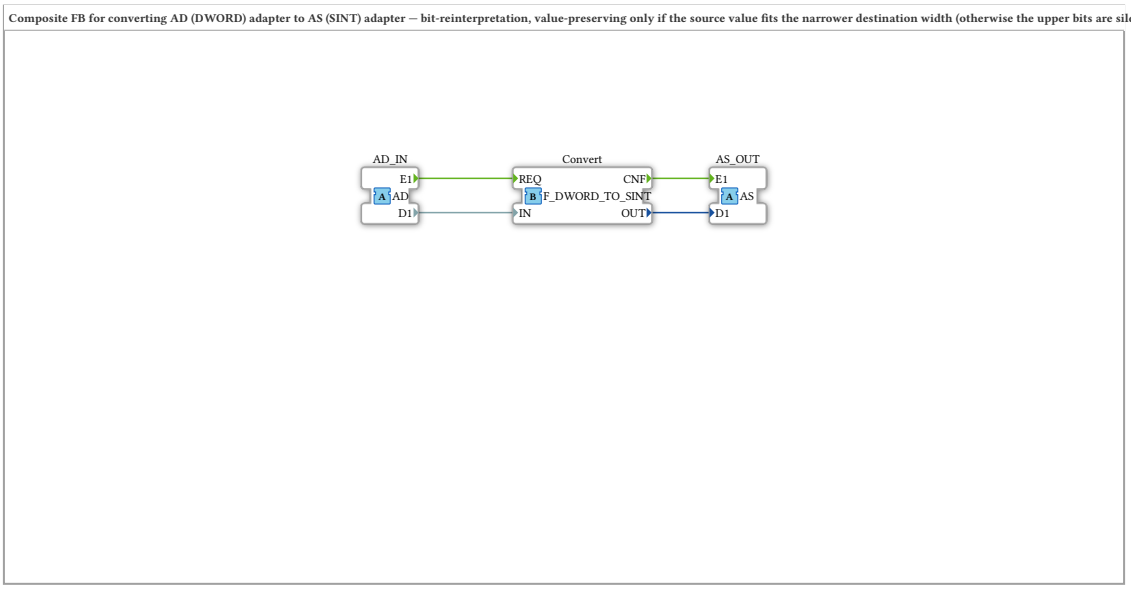
Here you will find all exercises from the unidirectional workspace, listed in ascending order. …

AD_TO_AS



DWORD Adapter Input – *AD*

AS – SINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **AD_TO_AS** function block is a composite block for converting a DWORD adapter (type `AD`) into a SINT adapter (type `AS`). It enables data conversion between different adapter standards within the 4diac IDE and is provided in version 1.0. The internal conversion is performed via the IEC 61131 block `F_DWORD_TO_SINT`.

Interface Structure

Event Inputs

No direct event inputs.

The block receives events exclusively via the adapter socket **AD_IN** (event port E1 of the AD adapter).

Event Outputs

No direct event outputs.

The function block sends events exclusively via the adapter plug **AS_OUT** (event port E1 of the AS adapter).

Data Inputs

No direct data inputs.

The function block receives data exclusively via the adapter socket **AD_IN** (data port D1 of the AD adapter, data type **DWORD**).

Data Outputs

No direct data outputs.

The function block sends data exclusively via the adapter plug **AS_OUT** (data port D1 of the AS adapter, data type **SINT**).

Adapter

- **Socket AD_IN**:

Type: `adapter::types::unidirectional::AD`
Description: Receives a DWORD value and an associated event.

- **Plug AS_OUT**:

Type: `adapter::types::unidirectional::AS`
Description: Sends a SINT value and an associated event.

Functionality

The function block operates in an event-driven manner:

1. An event at socket **AD_IN.E1** triggers processing.
2. The internal block `Convert` (type `F_DWORD_TO_SINT`) is started. It receives the DWORD value from **AD_IN.D1** at its input `IN` and performs the conversion to a SINT value.
3. After the conversion is complete, `Convert` signals an event via its output `CNF`.
4. This event is forwarded to the plug **AS_OUT.E1**, and simultaneously the converted SINT value from `Convert.OUT` is passed to **AS_OUT.D1**.

This establishes a complete, unidirectional coupling between the AD and AS adapters.

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- **Value Range Loss**: Converting DWORD (32-bit, unsigned) to SINT (8-bit, signed) can result in data loss. Only the lower 8 bits of the DWORD are used; the SINT value range is between −128 and +127.
- **Library Used**: The function block uses the IEC 61131 conversion function `F_DWORD_TO_SINT`, which is defined in the package group `iec61131::conversion`.
- **Unidirectional Data Flow**: The function block only allows data flow in one direction (from AD to AS) and has no feedback.

State Overview

Since this is a composite function block, there is no explicit internal state machine. The process can be described as a simple state sequence as follows:

- **WAIT**: The function block waits for an event at socket AD_IN.
- **CONVERT**: After receiving the event, the conversion `F_DWORD_TO_SINT` is performed.
- **SEND**: After the conversion is complete, the output event is sent to plug AS_OUT, and the data value is passed on.

This returns the function block to the **WAIT** state.

Application Scenarios

- **Data Conversion Between Different Adapter Types**: If a sending module provides a DWORD value (e.g., a 32-bit counter reading) via an AD adapter, but the receiver expects a SINT value via an AS adapter, this module is used for mediation.
- **Event-Driven Signal Conditioning**: In control applications where events must be transmitted asynchronously between adapters, `AD_TO_AS` serves as a bridge for type conversion.

Comparison with Similar Modules

- **AD_TO_INT / AD_TO_BYTE**: Analog conversion modules that convert DWORD values to other integer types (e.g., INT, BYTE).
- **AD_TO_INT / AD_TO_BYTE**: Analog conversion modules that convert DWORD values to other integer types (e.g., INT, BYTE). - **Direct Type Conversion**: Unlike using a simple `F_DWORD_TO_SINT` as an embedded function block, `AD_TO_AS` offers complete adapter encapsulation, allowing the interfaces to be integrated into a 4diac system without additional connection logic.

Conclusion

The **AD_TO_AS** function block provides a compact and standardized solution for converting DWORD to SINT data in an event-driven adapter environment. By utilizing the IEC 61131 library function and its clear,

event-based execution, it is reliable and easily integrated into existing control projects. The unidirectional structure limits its application to a simple data flow, which is sufficient for many use cases.

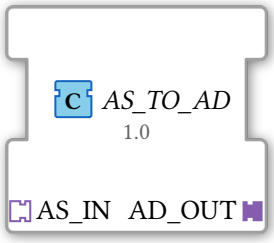
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

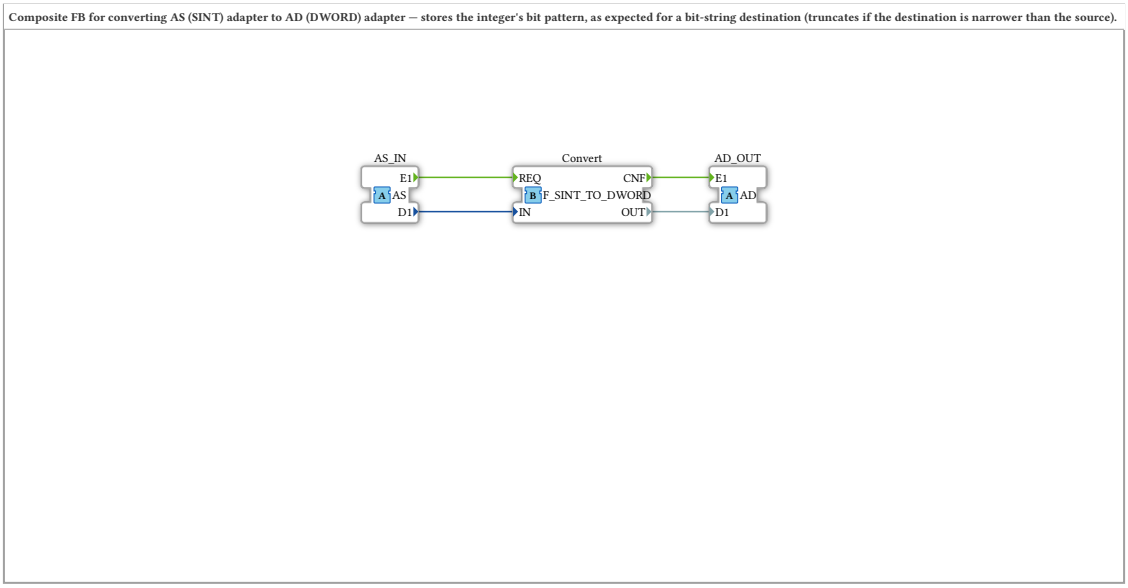
]

AS_TO_AD

SINT Adapter Input – *AS*



AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_TO_AD** is a composite function block (FB) that converts a SINT adapter interface (AS) to a DWORD adapter interface (AD). It serves as a converter between two unidirectional adapter types and is typically used to adapt data from a SINT-based signal path to a DWORD-based signal path.

Interface Structure

The function block does not have its own event or data inputs/outputs, but provides only adapter interfaces.

Event Inputs

- No direct event inputs – event control is handled via the incoming adapter **AS_IN** (socket).

The adapter provides the start signal for the conversion via its event output **E1**.

Event Outputs

- No direct event outputs – event output is handled via the outgoing adapter **AD_OUT** (Plug).

After conversion is complete, the event **E1** is sent to the outgoing adapter.

Data Inputs

- No direct data inputs – the SINT value to be converted is provided via the adapter **AS_IN** (Socket) through its data output **D1**.

Data Outputs

- No direct data outputs – the converted DWORD value is output via the adapter **AD_OUT** (Plug) through its data input **D1**.

Adapters

Adapter Name	Direction	Type	Description
AS_IN	Socket (Input)	adapter::types::unidirectional::AS	Input adapter for the SINT value (including event E1)
AD_OUT	Plug (Output)	adapter::types::unidirectional::AD	Output adapter for the converted DWORD value (including event E1)

Functionality

This function block operates as a simple pass-through with data type conversion:

1. An event is received at output **E1** via the **AS_IN** socket. Simultaneously, a value of type `SINT` is present at the **D1** data output of the socket.
2. The event is forwarded to the internal function block **Convert** (type `F_SINT_TO_DWORD`) and triggers its conversion.
3. The `F_SINT_TO_DWORD` block converts the received `SINT` value into a `DWORD` value and outputs the result at its **OUT** output.
4. After successful conversion, the **CNF** event of the internal block is passed to the **AD_OUT** plug, which then triggers its **E1** event. Simultaneously, the converted `DWORD` value is forwarded to the **D1** data input of the plug.

The entire process is synchronous and without data buffering – each conversion is initiated immediately by an incoming event, and the result is output instantly.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Unidirectional Adapters**: The block uses only unidirectional adapters (socket for input, plug for output). Reverse communication is not supported.
- **Reusable Standard Block**: The conversion is implemented using the IEC 61131-compliant block `F_SINT_TO_DWORD`, which is available in the library `iec61131::conversion`.
- **Composite Architecture**: The block is implemented as a composite function block (FB) and allows for easy adaptation or reuse of the internal network.
- **License**: The block is licensed under the Eclipse Public License 2.0 (EPL-2.0) and includes a corresponding copyright notice.

State Overview

The block does not have its own state machine. Its operation is purely event-driven and data flow-based:

1. **Waiting for Event** – The internal FB `Convert` waits for a `REQ` event (provided by **AS_IN.E1**).
2. **Conversion** – An incoming event initiates the conversion of the SINT value to a DWORD value.
3. **Output** – Upon completion, the **CNF** event is generated by the internal function block, and the converted value is passed to the output adapter.

Note: Since no delay or buffering occurs, no other event can be processed during the conversion.

Application Scenarios

- **Sensor Integration**: A sensor provides data as SINT (e.g., 8-bit values), while the control unit processes only DWORD values.
- **System Integration**: Existing SINT-based communication interfaces are to be integrated into a system that expects DWORD adapters.
- **Protocol Conversion**: Modular manufacturing systems use various adapter types that can be interconnected using such converters.

Comparison with Similar Function Blocks

- **BYTE_TO_WORD, BYTE_TO_DWORD, WORD_TO_DWORD** – Corresponding converters for other data types are available in the IEC 61131 library.
- **AS_TO_AD** differs in that it uses adapter interfaces instead of direct inputs/outputs. This makes it particularly suitable for the modular design of adapter chains.
- **Direct Converter FBs** such as `F_SINT_TO_DWORD` offer a simpler interface (inputs/outputs), while the Composite FB has the advantage of adapter compatibility.

Conclusion

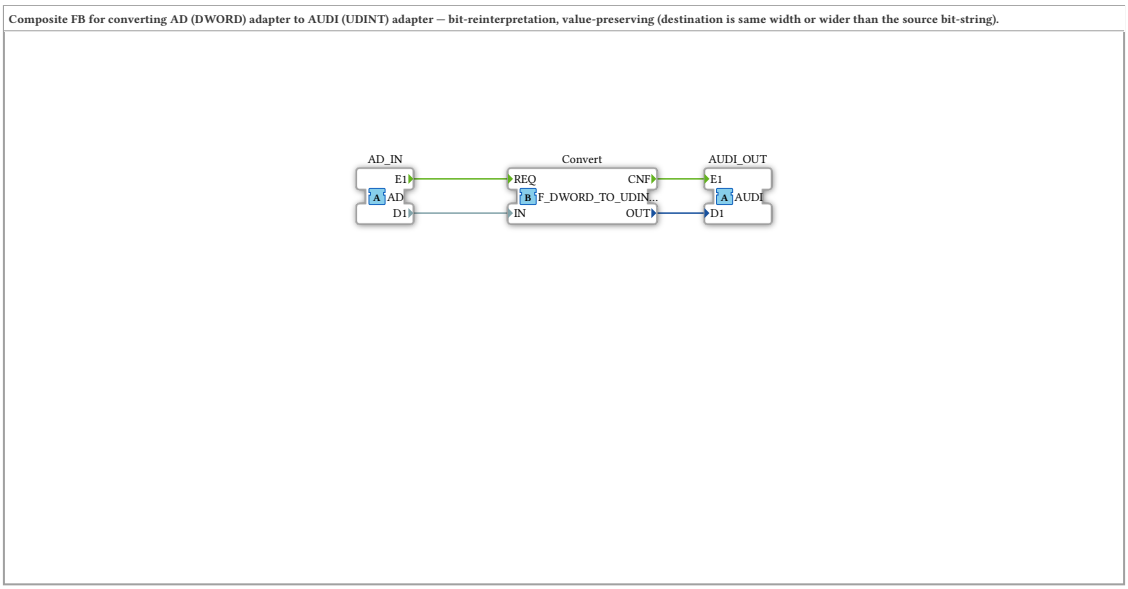
The **AS_TO_AD** function block is a specialized adapter converter that enables seamless integration of a SINT-based data path into a DWORD-based architecture. By using the standardized function block `F_SINT_TO_DWORD` and implementing it as a composite function block, it is reliable, extensible, and easy to integrate into existing 4diac projects. Its unidirectional design makes it ideal for applications where data only needs to flow in one direction.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AD_AUDI Conversion

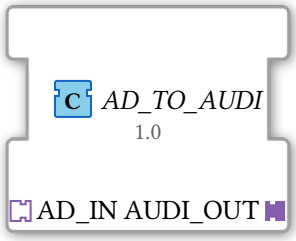
AD_TO_AUDI



Introduction

The AD_TO_AUDI function block is a specialized adapter conversion module. It is used to convert a unidirectional **AD** (DWORD) adapter socket into a unidirectional **AUDI** (UDINT) adapter plug. Internally, the module uses the standard conversion function `F_DWORD_TO_UDINT` to ensure clean type conversion between the bit string format (DWORD) and the numeric format (UDINT).

DWORD Adapter Input – *AD*



AUDI – UDINT Adapter Output

Interface Structure

Adapters

- **AD_IN** (Socket): Input adapter of type `AD` (DWORD).
- **AUDI_OUT** (Plug): Output adapter of type `AUDI` (UDINT).

Functionality

As soon as an event (`E1`) arrives at `AD_IN` along with data (`D1`), the DWORD value is internally converted into a UDINT value. This is then immediately passed on via the `AUDI_OUT` adapter (event `E1`, data `D1`).

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).

✓ **Direct adapter-to-adapter coupling**.
✓ **Standard-compliant conversion** through the use of `F_DWORD_TO_UDINT`.
✓ **Event-driven**: Every update at the input immediately triggers an update at the output.

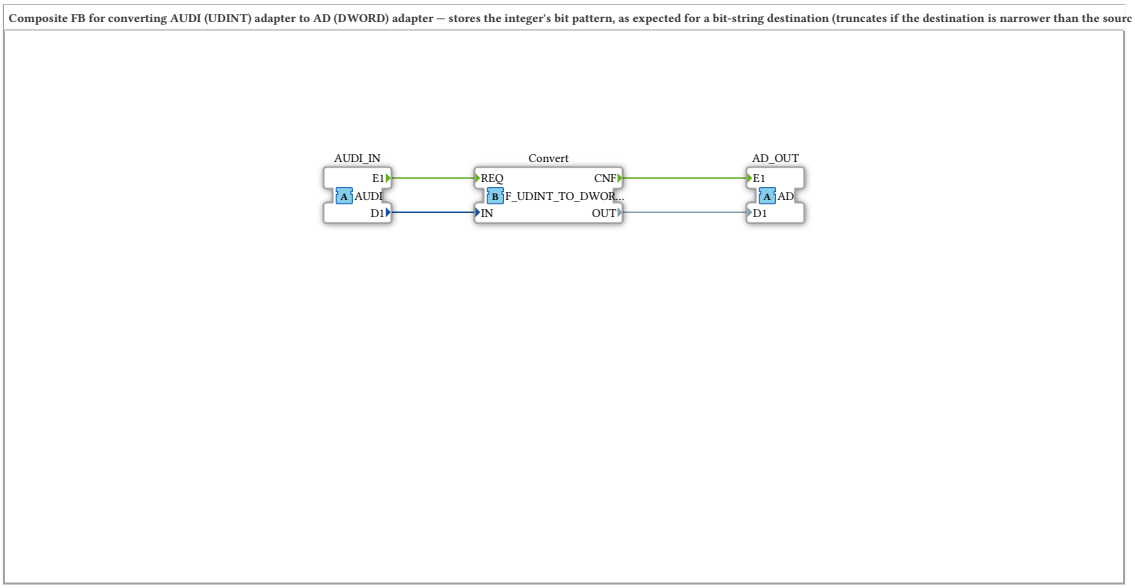
Application Scenarios

- Connecting ISOBUS input blocks (which often return DWORD values) to adapter-based numerical processing logic.
- Protocol and type adaptation in complex block networks.

🛠️ Related Exercises

- Exercise_011_AUDI

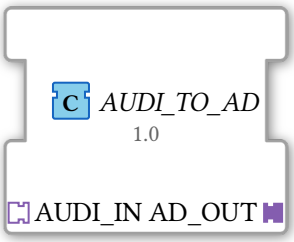
AUDI_TO_AD



Introduction

The AUDI_TO_AD function block is a specialized adapter conversion module. It converts a unidirectional **AUDI** (UDINT) adapter socket into a unidirectional **AD** (DWORD) adapter plug. Internally, the module uses the standard conversion function `F_UDINT_TO_DWORD` to ensure clean type conversion between the numeric format (UDINT) and the bit string format (DWORD).

UDINT Adapter Input – *AUDI*



AD – DWORD Adapter Output

Interface Structure

Adapters

- **AUDI_IN** (Socket): Input adapter of type `AUDI` (UDINT).
- **AD_OUT** (Plug): Output adapter of type `AD` (DWORD).

Functionality

As soon as an event (`E1`) arrives at `AUDI_IN` along with data (`D1`), the UDINT value is internally converted into a DWORD value. This is then immediately passed on via the `AD_OUT` adapter (event `E1`, data `D1`).

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).

✓ **Direct adapter-to-adapter coupling**.
✓ **Standard-compliant conversion** through the use of `F_UDINT_TO_DWORD`.
✓ **Event-driven**: Every update at the input immediately triggers an update at the output.

Application Scenarios

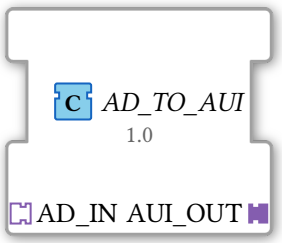
- Feeding numerical calculation results back to ISOBUS output modules (e.g., bar graph or numeric display), which often expect DWORD as the input format.
- Compatibility layer in adapter-based control architectures.

Numeric Index (unidirectional)

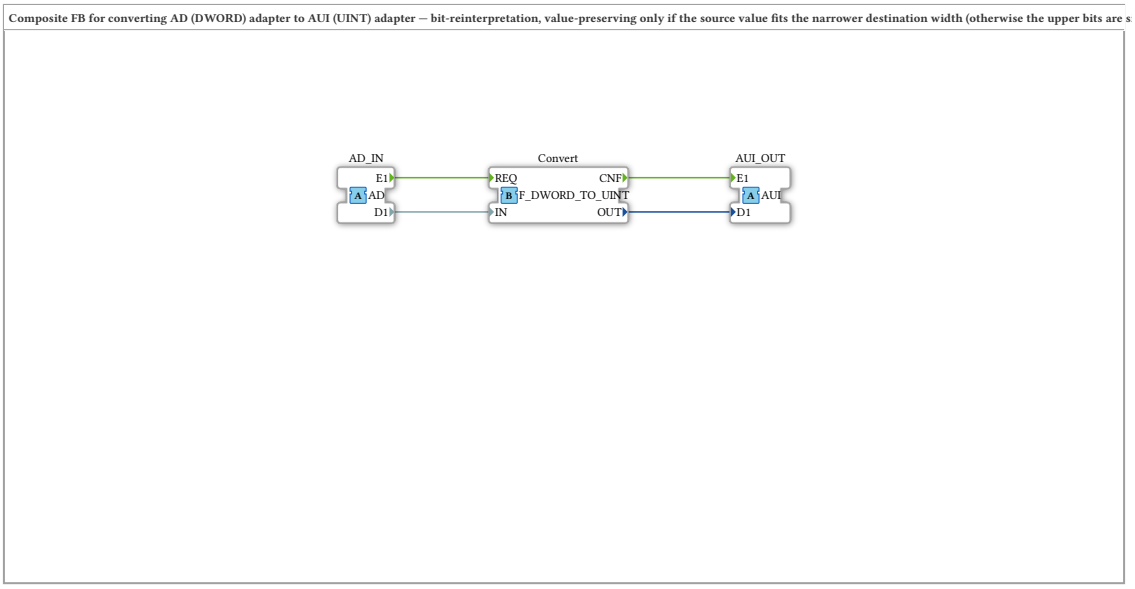
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AD_TO_AUI

DWORD Adapter Input – *AD*



AUI – UINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block `AD_TO_AUI` is a composite block that converts an AD adapter (DWORD-based) into an AUI adapter (UINT-based). It enables the seamless integration of components that work with different data types by converting incoming DWORD values to UINT values and forwarding the corresponding events.

Interface Structure

Event Inputs

Name	Type	Description
(none of its own)	-	The function block does not have its own event inputs. Event control is handled via the adapter socket `AD_IN`.

Event Outputs

Name	Type	Description
(none built-in)	-	The function block has no built-in event outputs. Event forwarding is handled via the adapter plug `AUI_OUT`.

Data Inputs

Name	Data Type	Description
(none built-in)	-	All data is provided via the adapter socket `AD_IN`.

Data Outputs

Name	Data Type	Description
(No custom)	-	All data is output via the adapter plug `AUI_OUT`.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AD`	`AD_IN`	Socket (Input)	Accepts DWORD values and associated events.

adapter::types::unidirectional::AUI	AUI_OUT	Plug (Output)	Outputs converted UINT values and associated events.

The adapter AD_IN provides an event E1 and a data element D1 (type DWORD).
The adapter AUI_OUT expects an event E1 and a data element D1 (type UINT).
The mapping is performed as described in the Functionality section.

Functionality

The function block contains an instance of the standard conversion module iec61131::conversion::F_DWORD_TO_UINT. Internal Circuitry:

1. **Event Control**:

- The event AD_IN.E1 is forwarded directly to the conversion block (Convert.REQ).
- After successful conversion, Convert.CNF signals the event to AUI_OUT.E1.

1. **Data Flow**:

- The data value AD_IN.D1 (DWORD) is passed to the input Convert.IN.
- The output Convert.OUT delivers the converted value (UINT) to AUI_OUT.D1.

The function block thus performs an event-driven, unidirectional type conversion from DWORD to UINT. The conversion is performed according to the IEC 61131-3 standard function DWORD_TO_UINT.

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- **Composite Block**: The function block encapsulates an existing standard conversion block and two adapter interfaces. This simplifies reuse in larger applications.
- **Data Type Conversion**: The conversion from DWORD (32-bit, unsigned) to UINT (16-bit, unsigned) can lead to information loss depending on the value range (e.g., if the DWORD value is greater than 65535). The user is responsible for monitoring the value range.
- **Unidirectional Interface**: Both the input and output adapters are unidirectional (only one event/data direction).
- **License**: This function block is available under the Eclipse Public License 2.0 (EPL-2.0).

State Overview

The AD_TO_AUI function block does not have its own internal state. Its behavior is purely combinatorial in conjunction with the event-driven execution of the included conversion block. There are no initialization or error states; the conversion occurs immediately upon each incoming event.

Application Scenarios

- **Interface Adaptation**: When an existing function block or subsystem only supports an AUI adapter (UINT), but the data source provides an AD adapter (DWORD).
- **Data Reduction**: Targeted reduction of a 32-bit value to 16 bits for downstream processing logic.
- **Integration into Existing Controllers**: Exchange of adapters between different bus systems or communication protocols that use different data types.

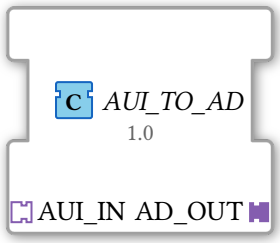
Comparison with Similar Function Blocks

- **AUI_TO_AD**: Performs the reverse conversion (UINT to DWORD). Both function blocks are symmetrically structured and utilize the corresponding standard conversion functions.
- **Direct Type Conversion**: Manual implementation using a F_DWORD_TO_UINT function block and adapters would be possible, but the composite function block offers an encapsulated, reusable solution.

Conclusion

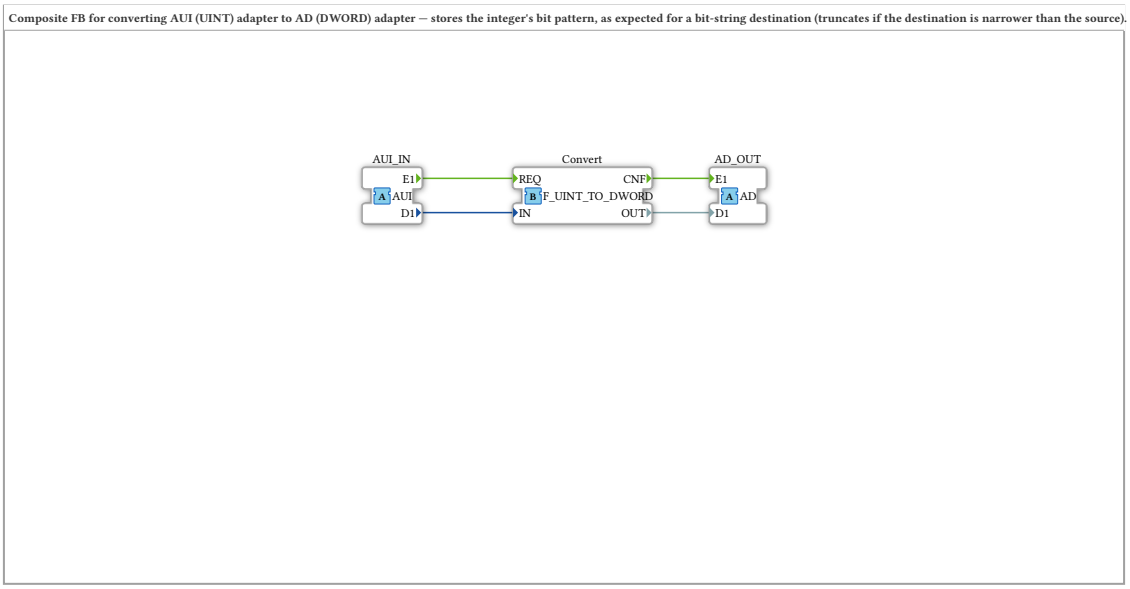
The AD_TO_AUI function block provides a simple and reliable way to convert between two common adapter types. It reduces development effort by encapsulating the necessary circuitry and conversion and providing it as a reusable component. Despite the potential value range limitation (DWORD → UINT), it is a useful tool for integrating heterogeneous system components in control engineering.

AUI_TO_AD



UINT Adapter Input – *AUI*

AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_TO_AD** is a composite function block (FB) that converts an AUI adapter (UINT data type) into an AD adapter (DWORD data type). It serves as a unidirectional conversion interface within an IEC 61499-based control application. The block is implemented as an encapsulation of the standard converter `F_UINT_TO_DWORD` from the library `iec61131::conversion` and enables adapter-based data transfer.

Interface Structure

Event Inputs

Name	Description
AUI_IN.E1	Event from the AUI adapter that triggers the conversion.

Event Outputs

Name	Description
AD_OUT.E1	Event to the AD adapter that signals the availability of the converted value.

Data Inputs

Name	Data Type	Description
AUI_IN.D1	UINT	Input value from the AUI adapter that is converted to a DWORD value.

Data Outputs

Name	Data Type	Description
AD_OUT.D1	DWORD	Output value that passes the converted DWORD value to the AD adapter.

Adapter

Type	Name	Role
`adapter::types::unidirectional::AUI`	AUI_IN	Socket – Input adapter for the UINT value and the associated event.
`adapter::types::unidirectional::AD`	AD_OUT	Plug – Output adapter for the converted DWORD value and the confirmation event.

Functionality

The function block **AUI_TO_AD** operates as an encapsulation of the function block `F_UINT_TO_DWORD`. As soon as an event arrives at the event input `AUI_IN.E1`, the connected internal converter `Convert` is called. This converter reads the value from the data input `AUI_IN.D1` (type UINT) and converts it into a DWORD value. The result is provided at the data output `AD_OUT.D1`. Simultaneously, after successful conversion, an event is generated at the output `AD_OUT.E1`, which signals completion to the downstream AD adapter.

The conversion is performed according to the IEC 61131-3 standard for the function `UINT_TO_DWORD`. The value range of the UINT input (0 to 65535) is mapped to the lower 16 bits of the DWORD output; the upper 16 bits are padded with zeros.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- The function block is implemented as a composite function block (FB) and does not contain its own state machines (ECCs). Its behavior is entirely determined by the integrated converter `F_UINT_TO_DWORD`.
- The connection between the adapters and the internal converter is established via event and data connections.
- The function block is designed for unidirectional data transfer – there is no reverse conversion from AD to AUI.
- The package name (`packageName`) is `adapter::conversion::unidirectional`, indicating a clear structure of the conversion adapters.

State Overview

As a composite function block, **AUI_TO_AD** does not have its own state machine. The functional sequence is purely event-driven: An event at the input triggers the conversion, after which an event is generated at the output. There are no internal states between these events.

•

Application Scenarios

- **Data Integration**: An AUI adapter provides a UINT value (e.g., from a sensor module) that is required in a system that processes only DWORD addresses (e.g., via an AD adapter).
- **Protocol Adaptation**: If existing components work with different data types, this function block can be used as a simple converter between the adapter interfaces.
- **Modularization**: Within a 4diac application, standardized conversion function blocks can be grouped as adapters to increase reusability.

Comparison with Similar Function Blocks

Comparable conversion function blocks exist for other data types, e.g., `WORD_TO_DWORD`, `BYTE_TO_DWORD`, or generic `TO_DWORD` functions. The unique feature of **AUI_TO_AD** lies in the encapsulation of the adapter protocols: Instead of a direct data connection, the values are provided via adapters, enabling loose coupling

between the components. Unlike a direct call to `F_UINT_TO_DWORD` in an application, this function block offers a standardized, reusable interface at the adapter level.

Conclusion

The **AUI_TO_AD** function block provides a clean, adapter-based solution for converting UINT to DWORD values. It is easy to use, requires no custom programming, and integrates seamlessly into the 4diac environment with its event and adapter networks. The encapsulation of the standard converter ensures robust and standards-compliant functionality. This function block is particularly suitable for applications where different adapter types need to be harmonized.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

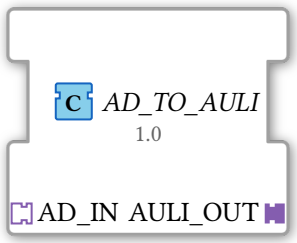
]

Numeric Index (unidirectional)

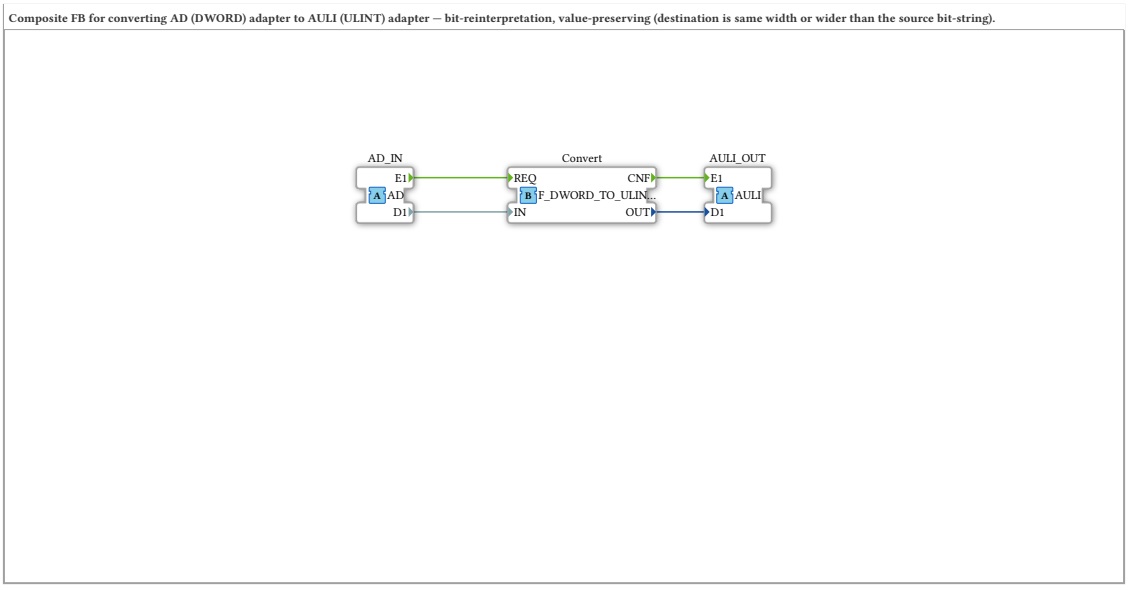
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AD_TO_AULI

DWORD Adapter Input – *AD*



AULI – ULINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The AD_TO_AULI function block is a composite FB for converting a DWORD adapter (AD) to a ULINT adapter (AULI). It uses the IEC 61131 conversion function F_DWORD_TO_ULINT and encapsulates the event and data connections for easy adapter-based conversion.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

- **AD_IN** (Socket): Type AD (DWORD), unidirectional input adapter for the DWORD.
- **AULI_OUT** (Plug): Type AULI (ULINT), unidirectional output adapter for the converted ULINT.

Functionality

The function block receives a DWORD via the AD_IN adapter. An event at the E1 input of AD_IN triggers the REQ event input of the internal converter F_DWORD_TO_ULINT. This converts the data value (AD_IN.D1) and signals completion via CNF. The CNF event then triggers the E1 event at the AULI_OUT adapter, and the converted value (ULINT) is output via AULI_OUT.D1.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- The function block is implemented as a composite and uses the `iec61131::conversion::F_DWORD_TO_ULINT` library.
- No direct event or data inputs/outputs are used; all communication takes place exclusively via adapters.
- The conversion is performed according to IEC 61499 event control: The output event is only sent after the conversion is complete.

State Overview

The composite block itself does not have its own states. The integrated function block F_DWORD_TO_ULINT goes through the following states:

- **IDLE**: Waiting for a REQ event.
- **REQ**: Conversion is being performed.
- **CNF**: Conversion complete, output data is being updated.

Application Scenarios

- Conversion of measured values or control data, which are available as DWORD (32-bit), into a ULINT (64-bit) adapter for subsequent processing.
- Use in modular automation systems based on adapter interfaces.
- Bridging between components with different adapter types.

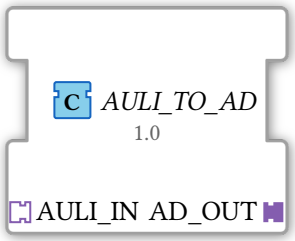
Comparison with Similar Function Blocks

- Direct type conversion function blocks like `F_DWORD_TO_ULINT` operate at the data level, while AD_TO_AULI encapsulates the adapter interfaces.
- Similar composite function blocks exist for other data type combinations (e.g., DWORD_TO_LWORD), specifically utilizing the unidirectional AD and AULI adapters.

Conclusion

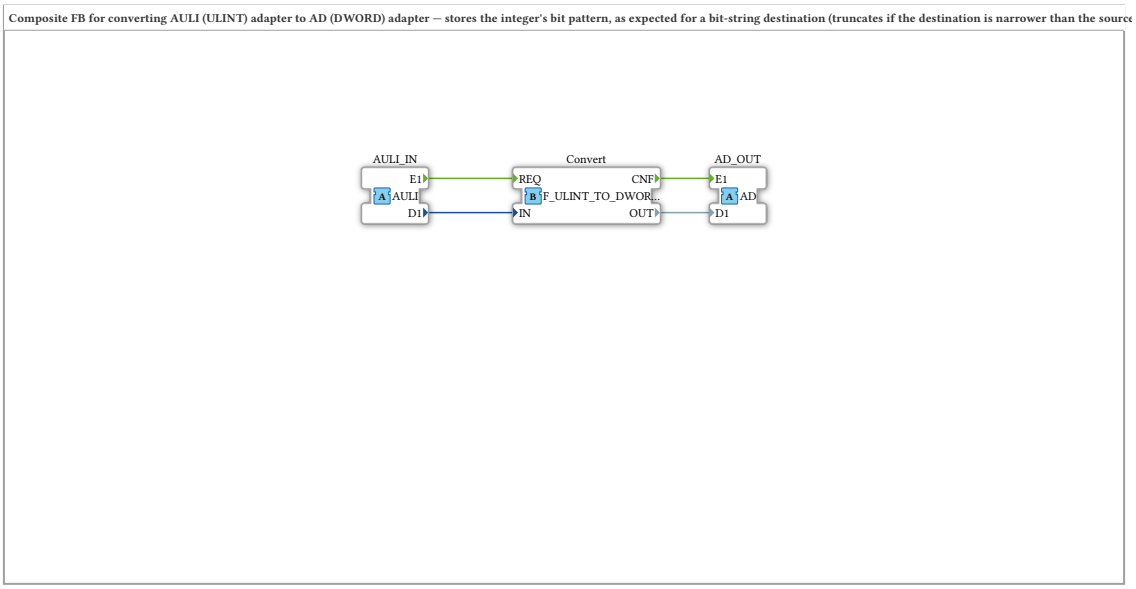
AD_TO_AULI offers a simple and reusable solution for adapter-based conversion from DWORD to ULINT. Encapsulating the conversion logic within a composite function block increases reusability and reduces network complexity.

AULI_TO_AD



ULINT Adapter Input – *AULI*

AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **AULI_TO_AD** function block is a composite function block that converts a unidirectional adapter of type **AULI** (ULINT) to a unidirectional adapter of type **AD** (DWORD). It serves as a bridge between components that provide data in ULINT format and those that expect a DWORD format. The conversion is performed using the IEC 61131 function `F_ULINT_TO_DWORD`.

Interface Structure

The function block does not have its own event or data inputs/outputs, but communicates exclusively via two adapters:

Event Inputs

Adapter	Event	Description
`AULI_IN`	`E1`	Starts the conversion

Event Outputs

Adapter	Event	Description
`AD_OUT`	`E1`	Signals conversion complete

Data Inputs

Adapter	Variable	Data Type	Description
`AULI_IN`	`D1`	ULINT	Input value for conversion

Data Outputs

Adapter	Variable	Data Type	Description
AD_OUT	D1	DWORD	Converted Output Value

Adapter

Name	Direction	Type	Description	
AULI_IN	Socket	adapter::types::unidirectional::AULI	Input Adapter (ULINT)	
AD_OUT	Plug	adapter::types::unidirectional::AD	Output Adapter (DWORD)	

Functionality

This function block internally connects socket AULI_IN to function block F_ULINT_TO_DWORD and its output to plug AD_OUT. The sequence is controlled via events:

1. An incoming event at AULI_IN.E1 activates the input of the conversion function.
2. The data value AULI_IN.D1 (ULINT) is passed to the input IN of the function.
3. After the conversion, the result (DWORD) is provided at output OUT.
4. Simultaneously, an event is output to AD_OUT.E1 to inform the connected function block about the new data.

The entire process is event-driven and occurs without any delay or state storage.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Structure**: The function block is implemented as a network consisting of a single function block (F_ULINT_TO_DWORD). All connections are hard-wired.
- **Package Import**: The conversion function used comes from the package iec61131::conversion. The function block is organized in the package adapter::conversion::unidirectional.
- **No State Storage**: The function block is purely combinatorial – each input event immediately generates an output event with the converted value.
- **Type Safety**: The adapter interfaces are strictly defined as ULINT or DWORD, so incorrect connections are detected during development.

State Overview

Since this is a composite block without its own algorithms, there is no explicit state machine. The overall state is derived solely from the internal state of the function block used, F_ULINT_TO_DWORD, which returns to its idle state after each execution cycle. During normal operation, the block cycles between:

- **Waiting**: Waiting for an event at AULI_IN.E1.
- **Converting**: Briefly executing F_ULINT_TO_DWORD.
- **Output**: Sending the result and the event to AD_OUT.E1.

Application Scenarios

- **Data Adaptation in Agricultural Engineering**: Conversion of ULINT values (e.g., from sensors or control units) into the DWORD format required by downstream components.
- **Adapter-Based Communication**: Use in modules that are connected exclusively via adapter interfaces, without direct input/output pins.
- **System Integration**: If one existing system provides ULINT data, but another only accepts DWORD adapters, this module serves as a simple converter.

Comparison with Similar Modules

Module	Description
F_ULINT_TO_DWORD	Simple conversion function, but without adapter integration.

AULI_TO_AD	Same functionality, but as a composite block with adapter interfaces.
Other adapter converters	Blocks exist for related types (e.g., LINT, DINT) that operate on the same principle.

The key difference lies in the interface type: While the pure function uses direct data and event pins, AULI_TO_AD encapsulates the conversion in an adapter-based solution.

Conclusion

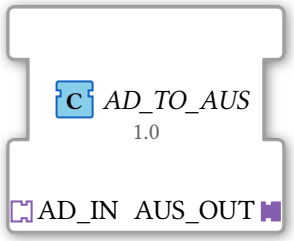
The AULI_TO_AD function block offers a clean, standardized way to convert ULINT data into the DWORD format of an AD adapter via an AULI adapter. Thanks to its composite nature, it is easy to use, requires no custom programming, and integrates seamlessly into an adapter-oriented architecture. It is particularly useful in environments where components communicate via defined adapter interfaces.

Numeric Index (unidirectional)

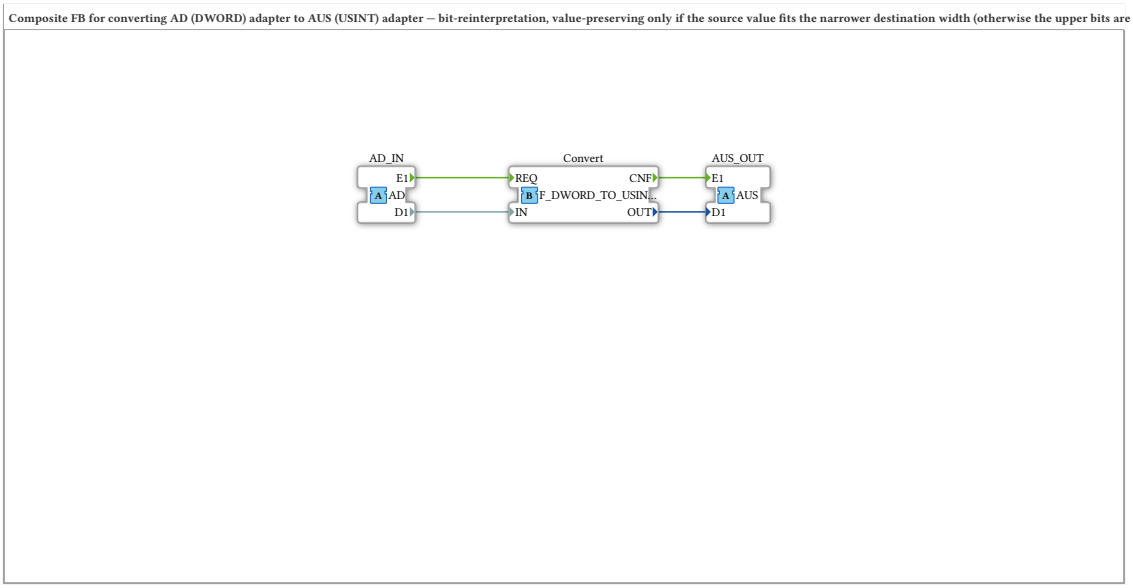
Here you will find all exercises from the unidirectional workspace, listed in ascending order. ...

AD_TO_AUS

DWORD Adapter Input – *AD*



AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AD_TO_AUS** is a composite function block (FB) that converts a DWORD adapter (AD) into a USINT adapter (AUS). It serves as a unidirectional interface for converting a 32-bit data value into an 8-bit value. The block encapsulates the IEC 61131 conversion function F_DWORD_TO_USINT and provides the adapter interfaces, allowing it to be seamlessly integrated into an adapter-based communication structure.

Interface Structure

Event Inputs

The **E1** event is received via the adapter input **AD_IN**. It triggers the conversion of the incoming DWORD value.

Event Outputs

The **E1** event is sent via the adapter output **AUS_OUT** as soon as the conversion is complete and the converted USINT value is available.

Data Inputs

The adapter input **AD_IN** provides a DWORD value (32-bit) via the **D1** data line.

Data Outputs

The adapter output **AUS_OUT** provides the converted USINT value (8-bit) via the **D1** data line.

Adapters

Adapter	Type	Direction	Description
AD_IN	`adapter::types::unidirectional::AD`	Input (Socket)	Receives DWORD data and control events.
AUS_OUT	`adapter::types::unidirectional::AUS`	Output (Plug)	Outputs USINT data and acknowledgment events.

Functionality

1. The function block waits for an event at the adapter input **AD_IN** (signal **E1**).
2. Upon arrival of the event, the DWORD value present at **AD_IN.D1** is forwarded to the internal function block `F_DWORD_TO_USINT`, and its `REQ` input is activated.
3. The integrated conversion block performs the type conversion from DWORD to USINT (type conversion according to IEC 61131-3).
4. After the conversion is complete, the internal function block (FB) signals the event `CNF`, which is passed to the adapter output **AUS_OUT** (signal **E1**). Simultaneously, the converted USINT value is made available via **AUS_OUT.D1**.

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- **Adapter Coupling:** The FB has no direct event or data inputs/outputs, but communicates exclusively via adapter interfaces. This enables loose coupling in a modular control architecture.
- **Library Used:** The conversion is performed using the standardized function block `iec61131::conversion::F_DWORD_TO_USINT`. This ensures IEC 61131 compliance.
- **Package name:** `adapter::conversion::unidirectional` (see CompilerInfo) places the function block (FB) in a specific adapter conversion library.

State Overview

The **AD_TO_OFF** is a pure function block without its own state machine. It operates strictly causally: Each input event leads to exactly one output event. The latency results from the execution time of the internal conversion block.

Application Scenarios

- **Interface Adaptation:** If a sensor or actuator delivers data in DWORD format (32 bits), but the controller can only process USINT values (8 bits), the FB is used as an intermediary.
- **Adapter-based systems:** In a 4diac environment that relies on unidirectional adapters, the function block can be seamlessly integrated into data flow paths without changing the event/data structure of the surrounding components.
- **Data reduction:** Targeted truncation of higher-order bytes when only the lower-order part of a DWORD is relevant (e.g., for counters or simple analog values).

Comparison with similar function blocks

- **AD_TO_AUS** is specific for the unidirectional conversion of a DWORD adapter to a USINT adapter. Alternatively, the IEC 61131 function `F_DWORD_TO_USINT` could be used directly, but then the adapter coupling would have to be performed manually.

Other adapter conversion blocks (e.g., `AD_TO_BYTE` or `AUS_TO_DWORD`) follow the same pattern but work with different data types. This function block fills the gap for the specific combination DWORD ⟶ USINT.

Conclusion

The **AD_TO_AUS** is a simple yet useful composite function block for type conversion between two unidirectional adapter interfaces. It reduces implementation effort by encapsulating the standard conversion and enabling clean, event-driven adapter communication. Thanks to its clear structure, it is suitable for both rapid design and production use in IEC 61499-based controllers.

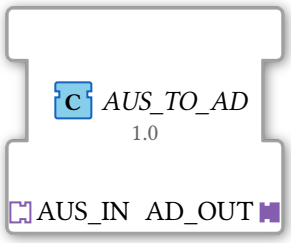
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

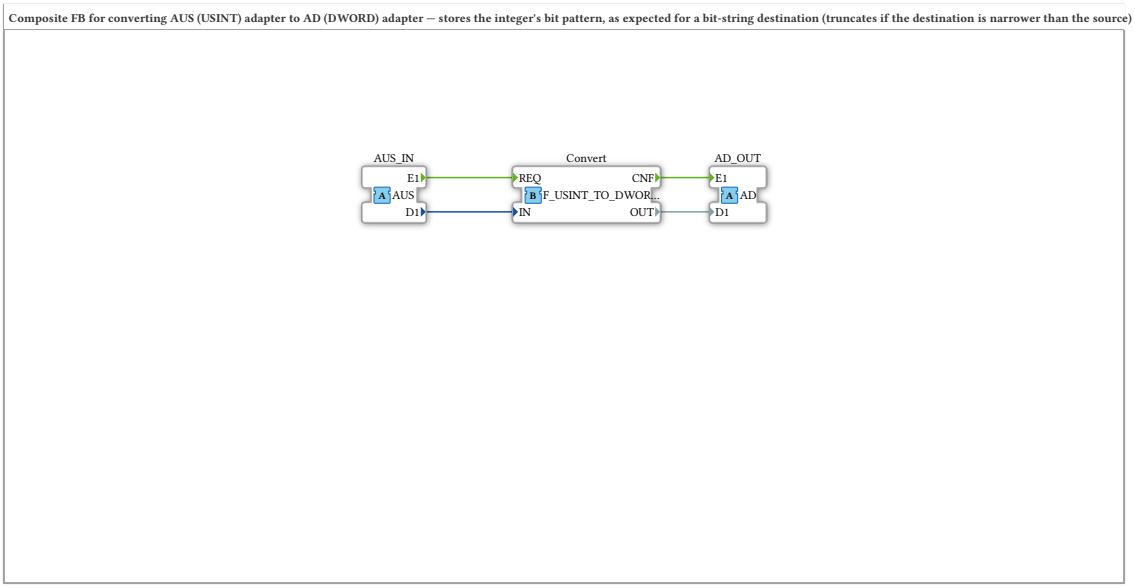
]

AUS_TO_AD

USINT Adapter Input – *AUS*



AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUS_TO_AD** is a composite block that performs a simple conversion from a USINT adapter (type AUS) to a DWORD adapter (type AD). It encapsulates the conversion of an unsigned 8-bit value (USINT) to a 32-bit DWORD value and makes this accessible via standardized adapter interfaces.

Interface Structure

Event Inputs

Name	Type	Description
`E1`	Event (from socket `AUS_IN`)	Starts the conversion of the incoming USINT value. The input is provided via socket `AUS_IN`.

Event Outputs

Name	Type	Description
E1	Event (to plug AD_OUT)	Signals successful conversion and the presence of the DWORD value at the output. The output is provided via plug AD_OUT.

Data Inputs

Name	Data Type	Description
D1	USINT (from socket AUS_IN)	The 8-bit value to be converted, in the range 0…255.

Data Outputs

Name	Data Type	Description
D1	DWORD (to plug AD_OUT)	The converted 32-bit value (direct type conversion of the USINT to DWORD).

Adapter

Role	Name	Type	Direction	Description
Socket	AUS_IN	adapter::types::unidirectional::AUS	Input	Receives the USINT value and the associated event.
Plug	AD_OUT	adapter::types::unidirectional::AD	Output	Provides the DWORD value and the acknowledgment event.

Functionality

This function block implements a simple, end-to-end data conversion:

1. An incoming event at the adapter input AUS_IN.E1 triggers the internal conversion.
2. The connected USINT value (AUS_IN.D1) is converted into a DWORD value using the standardized IEC 61131 function block F_USINT_TO_DWORD.
3. After successful conversion, the output event AD_OUT.E1 is sent, and the DWORD value is made available as AD_OUT.D1.

The function block operates without its own state logic – it performs the conversion immediately for each event (transparent behavior).

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Design:** The function block consists solely of an internal network with a single function block (F_USINT_TO_DWORD) and direct event and data connections.
- **Reuse:** The conversion uses the IEC 61131-3 standard function F_USINT_TO_DWORD, which is included in the library iec61131::conversion.
- **Adapter Interfaces:** The interfaces are implemented as unidirectional adapters (adapter::types::unidirectional::AUS and AD, respectively). Each adapter consists of exactly one event channel and one data channel.
- **No State Machine:** Since this is a pure composite device without its own ECC, no states are managed.

State Overview

The device does not have its own state machine. Its behavior is purely data flow-driven: For each incoming event, exactly one conversion is performed, and the event is forwarded without delay.

Application Scenarios

- **Interface Adaptation:** When a USINT value (e.g., a byte sensor) needs to be used in a system that internally only processes DWORD values.
- **Adapter Bridging:** Use in modular control architectures where adapter types are not directly compatible and conversion between AUS and AD adapters is required.
- **Prototype/Test Environments:** Fast connection of two components with different adapter definitions without requiring their own conversion logic.

Comparison with Similar Function Blocks

- **F_USINT_TO_DWORD:** The underlying conversion block is a pure data converter without an adapter interface. It requires direct data and event connections.

AUS_TO_AD`, on the other hand, encapsulates this function and offers a standardized adapter interface.

- **Other adapter converters (e.g., BYTE_TO_DWORD, WORD_TO_DWORD):** Similar function blocks exist for other data types, but usually require different adapter types. `AUS_TO_AD` is specifically optimized for the combination of the adapters `AUS` (USINT) and `AD` (DWORD).

Conclusion

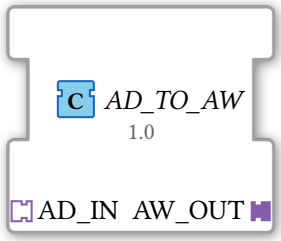
The function block `AUS_TO_AD` offers a simple and robust way to convert a unidirectional USINT adapter into a DWORD adapter. By using the standardized conversion module and adapter interfaces, it can be seamlessly integrated into IEC 61499-based automation systems. It reduces the manual effort required for type conversions and ensures clean, reusable interface adaptation.

Numeric Index (unidirectional)

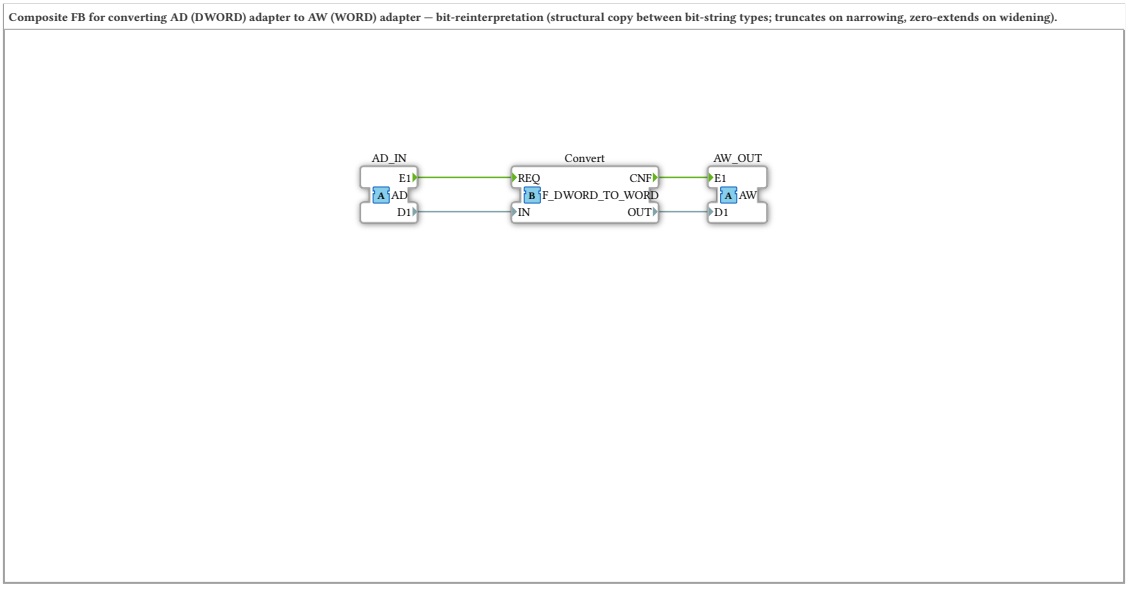
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AD_TO_AW

DWORD Adapter Input – *AD*



AW – WORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_TO_AW** is a composite function block that converts a DWORD adapter (type `AD`) into a WORD adapter (type `AW`). It encapsulates the conversion of a DWORD value to a WORD value and provides the corresponding event and data forwarding. The implementation is based on an internal call to the standard conversion block `F_DWORD_TO_WORD` from the IEC 61131 library.

Interface Structure

The block has no direct input/output events or data ports. All signals are routed via the connected adapters (socket and plug).

Event Inputs

- **AD_IN.E1** (via socket `AD_IN`): This event triggers the conversion of the incoming DWORD value.

Event Outputs

- **AW_OUT.E1** (via plug `AW_OUT`): This event is sent after successful conversion.

Data Inputs

- **AD_IN.D1** (via socket `AD_IN`): Input data of type `DWORD`, which is converted into a WORD value.

Data Outputs

- **AW_OUT.D1** (via Plug `AW_OUT`): Output data of type `WORD`, containing the result of the conversion.

Adapters

Role	Name	Type	Direction	Comment
Socket	AD_IN	`Adapter::types::unidirectional::AD`	Input (DWORD)	Returns DWORD data and event.
Plug	AW_OUT	`Adapter::types::unidirectional::AW`	Output (WORD)	Provides WORD data and event.

Both adapters are of type *unidirectional*, i.e., Communication is unidirectional.

Functionality

1. As soon as the event `E1` arrives at socket `AD_IN`, the value of `AD_IN.D1` (DWORD) is passed to the internal conversion module `F_DWORD_TO_WORD`.
2. The internal module performs the conversion and creates the result (WORD) at its output `OUT`.
3. The converted value is immediately forwarded to plug `AW_OUT.D1`.
4. Simultaneously, the event `E1` is triggered at plug `AW_OUT` to inform the downstream logic.

The entire process is event-driven and synchronous.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Encapsulation**: The function block uses the standardized IEC 61131 function `F_DWORD_TO_WORD`, which ensures clean and portable conversion.
- **Adapter-based**: Inputs and outputs are implemented exclusively via adapters – there are no separate event/data ports at the block level.
- **Unidirectional**: The adapters used support only one direction of data flow, which simplifies handling in simple conversion chains.
- **Composite Design**: The function block does not have its own ECC (Execution Control Chart); control is handled entirely by the internal block.

State Overview

As a composite block without its own ECC, **AD_TO_AW** has no internal states. Execution is purely event-driven and occurs atomically with each incoming event at the socket.

AD_TO_AW

Application Scenarios

- **Adapter Adaptation**: When a control system provides data via a DWORD adapter, but a connected module or component expects a WORD adapter.
- **Data Reduction**: Targeted truncation of a DWORD value to the lower 16 bits (WORD) for subsequent processing steps.
- **Bridging**: Use in heterogeneous system landscapes where different adapter types interact.

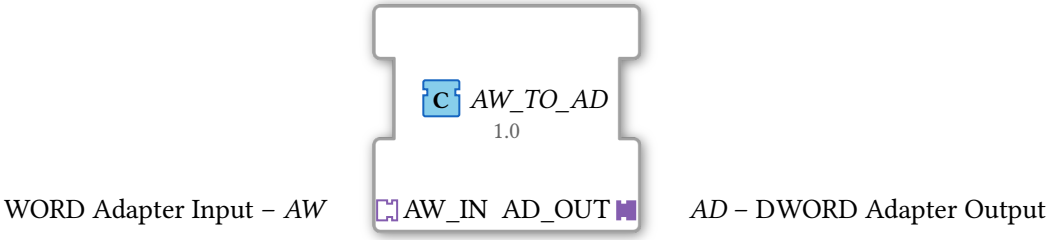
Comparison with Similar Components

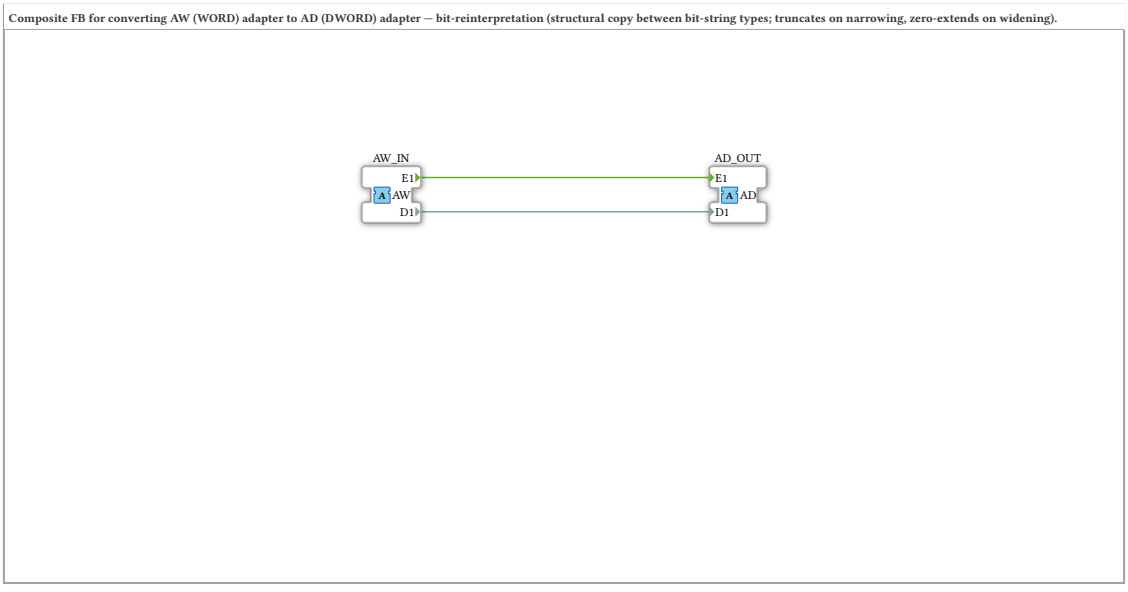
- **WORD_TO_DWORD**: Performs the reverse conversion (WORD → DWORD) and appends the value (usually by adding zeros).
- **AD_TO_AD (e.g., AD_TO_AD with other types)**: Other composite components can encapsulate similar adapter conversions for additional data types (e.g., BYTE, INT, REAL).
- **Direct use of `F_DWORD_TO_WORD`**: Instead of a composite block, the conversion function could be directly integrated into a function block network – however, the composite offers a reusable, encapsulated solution.

Conclusion

The **AD_TO_AW** function block is a simple yet useful composite block for converting a DWORD adapter to a WORD adapter. It uses a standardized conversion function, is easy to understand, and can be used in many adapter-based automation systems. Encapsulating it in its own function block makes the network topology clearer and increases reusability.

AW_TO_AD





• – ○ • – ○ • – ○ •

Introduction

The function block **AW_TO_AD** is a composite function block (FB) that converts a unidirectional WORD adapter (AW) into a unidirectional DWORD adapter (AD). It serves as an interface converter and enables the connection of components that use different adapter types. The FB forwards events and data directly

from the input to the output adapter, implicitly performing a type conversion from 16-bit WORD to 32-bit DWORD.

Interface Structure

Event Inputs

- **AW_IN.E1** – Event input via socket `AW_IN`. An incoming event is forwarded directly to the output.

Event Outputs

- **AD_OUT.E1** – Event output via the `AD_OUT` socket. The event passed through from the input is output here.

Data Inputs

- **AW_IN.D1** – Data input (Type: WORD, 16-bit) via the `AW_IN` socket.

Data Outputs

- **AD_OUT.D1** – Data output (Type: DWORD, 32-bit) via the `AD_OUT` socket. The received 16 bits are mapped to the lower 16 bits of the DWORD; the upper 16 bits are padded with zeros (implicit conversion).

Adapters

Name	Type	Direction	Description
`AW_IN`	`adapter::types::unidirectional::AW`	Socket (Input)	WORD adapter providing event and data inputs
`AD_OUT`	`adapter::types::unidirectional::AD`	Plug (Output)	DWORD adapter providing event and data outputs

Functionality

This module is designed as a simple forwarder. Within the internal FB network, the event and data lines between `AW_IN` and `AD_OUT` are directly connected:

- An event at `AW_IN.E1` is immediately forwarded to `AD_OUT.E1`.
- The 16-bit data value from `AW_IN.D1` is transferred to `AD_OUT.D1`. This involves an implicit expansion to 32 bits: The WORD value is copied into the less significant 16 bits of the DWORD, and the more significant bits are set to `0`.

No processing or intermediate storage takes place – the FB functions as a simple adapter converter.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Composite Function Block without its own state logic:** The function block does not have an ECC (Execution Control Chart) and operates statelessly. All processes are determined by the direct connections within the function block network.
- **Unidirectional operation:** Conversion only occurs in one direction (AW ⟶ AD). A separate function block would be required for the reverse direction.
- **Implicit type conversion:** Data is cast from WORD to DWORD without explicit conversion logic. If a different mapping (e.g., sign handling) is required, a separate function block must be used.
- **Package & License:** The function block is available in package `adapter::conversion::unidirectional` under the Eclipse Public License 2.0.

State Overview

Since the function block is implemented as a composite without an ECC, there is no internal state machine. Its entire functionality is limited to the direct transmission of events and data. The function block (FB) is therefore **stateless** and reacts immediately to every incoming event.

Application Scenarios

- **Interface Migration:** Replacing a component that expects a WORD adapter with a component that uses a DWORD adapter.
- **System Integration:** Connecting an existing WORD-based module to a DWORD-based controller.
- **Protocol Adaptation:** When a subsystem delivers data in WORD format, but the target requires a DWORD input.
- **Testing and Simulation:** Simple conversion for testing purposes without having to implement complex logic.

Comparison with Similar Function Blocks

Function Block	Description
`WORD_TO_DWORD`	Pure data conversion without an adapter interface
`AW_TO_AD`	Converts complete unidirectional adapters (event + data)
`AD_TO_AW` (not available)	Theoretical building block for the reverse direction

Unlike a pure data converter, `AW_TO_AD` also considers event forwarding and can be directly integrated into adapter-based architectures.

Conclusion

AW_TO_AD` is a simple yet useful composite function block for converting adapters from WORD to DWORD. It enables quick and clean adaptation of interfaces in 4diac projects without requiring the writing of custom conversion logic. Due to its statelessness and direct passthrough, it is particularly suitable for unidirectional connections in existing automation solutions.

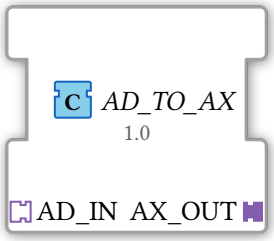
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

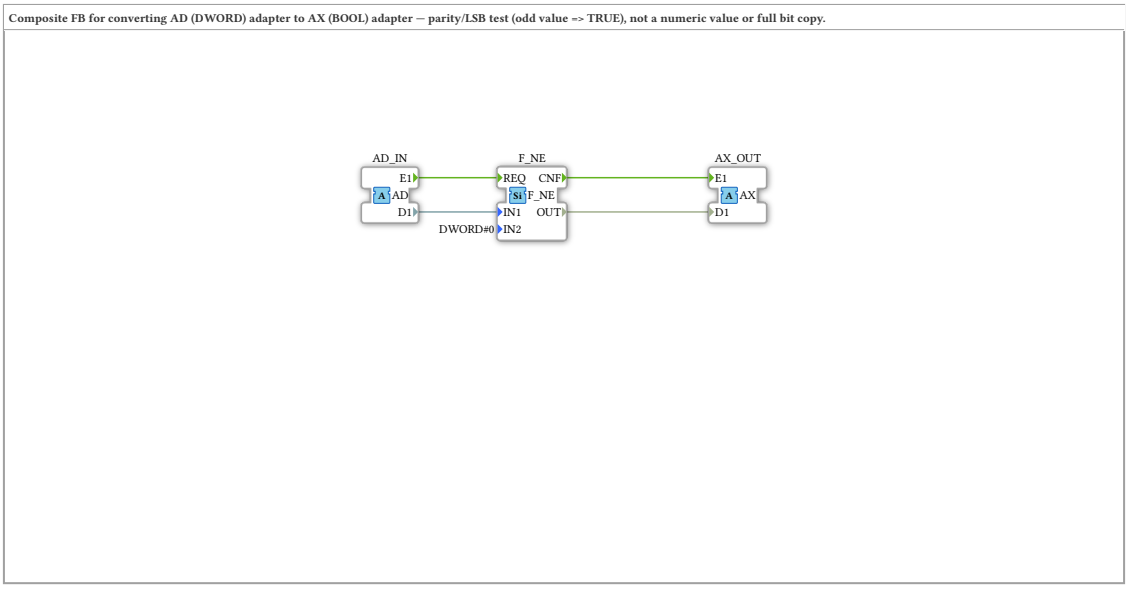
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AD_TO_AX



DWORD Adapter Input – *AD* — *AX* – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `AD_TO_AX` serves as a composite function block for converting an AD adapter (data word – DWORD) into an AX adapter (Boolean value – BOOL). It encapsulates a comparison mechanism that checks whether the incoming DWORD value is non-zero and forwards the result as a BOOL to the output adapter. This block simplifies adapter communication in control systems by converting signal types between different components.

Interface Structure

Event Inputs

No standalone event inputs. The event is provided via the socket adapter `AD_IN` (see section **Adapters**).

Event Outputs

No standalone event outputs. The result event is provided via the plug adapter `AX_OUT` (see section **Adapters**).

Data Inputs

No standalone data inputs. The input data is provided via the socket adapter `AD_IN` (see section **Adapters**).

Data Outputs

No standalone data outputs. The output data is provided via the plug adapter `AX_OUT` (see section **Adapters**).

Adapters

Adapter	Direction	Type	Comment
`AD_IN`	Socket	`adapter::types::unidirectional::AD`	DWORD input adapter – provides an event and DWORD data.
`AX_OUT`	Plug	`adapter::types::unidirectional::AX`	BOOL output adapter – outputs an event and a Boolean result.

The adapters follow a unidirectional pattern and each provides an event output (`E1`) and a data output (`D1`):

- **AD_IN.E1** ⟶ initiates processing.
- **AD_IN.D1** ⟶ contains the DWORD source value.
- **AX_OUT.E1** ⟶ signals the end of processing.
- **AX_OUT.D1** ⟶ contains the Boolean result value.

Functionality

The FB AD_TO_AX is implemented as a composite FB. It contains an instance of the comparison block F_NE ("not equal") from the IEC 61131 library.

1. An incoming event at AD_IN.E1 triggers the input F_NE.REQ.
2. The block F_NE compares the value of AD_IN.D1 with the constant value DWORD#0.
3. The comparison result (BOOL) is provided at the output F_NE.OUT.
4. This result is transmitted to AX_OUT.D1 via a data connection.
5. Simultaneously, F_NE.CNF signals the end of the calculation and triggers AX_OUT.E1.

Thus, a BOOL value is derived from a DWORD value:

- DWORD = 0 ⟶ Output FALSE
- DWORD ≠ 0 ⟶ Output TRUE

Technical Features

- **Parity/LSB test** (odd value => TRUE), not a numeric value or full bit copy.
- **Composite Function Block**: The function block consists of an internal network that implements the actual logic – this facilitates reuse and encapsulation.
- **Use of F_NE**: The comparison function block used, from the IEC 61131 library, is standardized and platform-independent.
- **Adapter-based communication**: Instead of individual inputs/outputs, adapters are used, enabling structured connections to other adapter-compatible modules.
- **Unidirectional adapters**: Both input and output adapters are unidirectional, meaning they only transport data and events in one direction – this avoids feedback loops and simplifies the design.

State overview

The function block (FB) does not have an explicit state machine, as it consists solely of a logical operation. The internal process is event-driven:

- **Idle state**: Waiting for an event at input AD_IN.E1.
- **Execution**: After the event, F_NE performs the calculation and forwards the result – this happens in one step without intermediate states.

Application scenarios

- **Adapter conversion**: A system uses DWORD adapters, but a connected module expects BOOL signals. The AD_TO_AX function block handles the conversion.
- **Threshold Check**: Checks whether a DWORD value deviates from zero (e.g., active signal, counter reading > 0).
- **Signal Conditioning**: Simplifies the integration of sensors or actuators whose output is encoded as a DWORD into Boolean control logic.

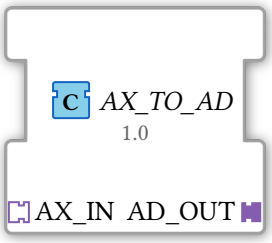
Comparison with Similar Function Blocks

- **Direct DWORD_TO_BOOL Converter**: Such a function block would typically only perform the data conversion but would not offer an adapter structure or event handling. AD_TO_AX, on the other hand, encapsulates both event and data paths in adapters.
- **General Adapter Converters**: Similar function blocks exist for other data types (e.g., AD_TO_AX for WORD, INT, etc.). This function block is specifically designed for converting DWORD to BOOL.
- **Composite vs. Basic Function Block**: As a composite function block, this block is easily adaptable because the internal network can be modified as needed.

Conclusion

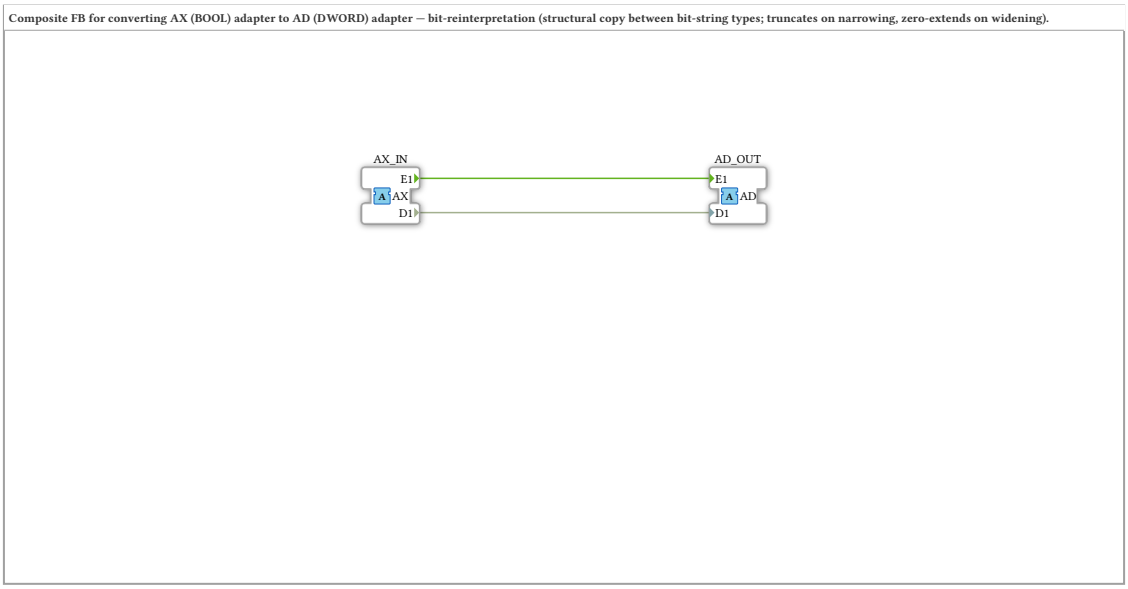
The function block AD_TO_AX offers an elegant and standardized way to convert DWORD adapter signals into BOOL adapters. By using the established comparison function block F_NE, the logic is robust and traceable. The adapter structure facilitates integration into modular automation systems and promotes reusability. This function block is particularly suitable for scenarios requiring event-driven conversion of digital values to Boolean states.

AX_TO_AD



BOOL Adapter Input – *AX*

AD – DWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The composite function block `AX_TO_AD` converts a Boolean adapter (AX) to a DWORD adapter (AD). It forwards event and data signals from the input adapter directly to the output adapter and implicitly converts the data type from Boolean to DWORD.

Interface Structure

Event Inputs

The function block has no direct event inputs. Event control is handled via the socket `AX_IN`.

Event Outputs

There are no direct event outputs. Events are passed via the plug `AD_OUT`.

Data Inputs

There are no direct data inputs. Data is acquired via socket `AX_IN`.

Data Outputs

There are no direct data outputs. Data output is handled via plug `AD_OUT`.

Adapters

- **AX_IN** (Socket): Adapter of type `adapter::types::unidirectional::AX`. Provides a BOOL signal (D1) and a trigger event (E1).
- **AD_OUT** (Plug): Adapter of type `adapter::types::unidirectional::AD`. Expects a DWORD signal (D1) and a trigger event (E1).

Functionality

The function block (FB) links the input adapter `AX_IN` directly to the output adapter `AD_OUT`:

- **Event Path**: The event from `AX_IN.E1` is forwarded unchanged to `AD_OUT.E1`.
- **Data Path**: The Boolean value from `AX_IN.D1` is transferred to `AD_OUT.D1`. An implicit type conversion occurs: `TRUE` becomes `1` (DWORD), and `FALSE` becomes `0` (DWORD).

The function block itself contains no internal logic or state machines – it functions purely as a pass-through converter between the two adapter types.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Composite FB**: The implementation consists solely of wiring within the FB network, without any additional algorithms.
- **Implicit Data Type Conversion**: The BOOL value is automatically converted to a DWORD value (bits 0-31: 0 or 1).
- **Unidirectional Adapters**: Both the input and output use unidirectional adapters that support only one data and one event connection.

State Overview

The `AX_TO_AD` function block does not have its own internal state. Its behavior is purely reactive: An incoming event at the input is immediately passed on to the output, optionally accompanied by the updated data signature.

- **Implicit Data Type Conversion**: The BOOL value is automatically converted to a DWORD value (bits 0-31: 0 or 1).
- **Unidirectional Adapters**:** Both the input and output use unidirectional adapters that support only one data and one event connection.
- **Composite FB**:**

Application Scenarios

- Integration of a digital sensor (e.g., switch, light barrier) into a system that expects a DWORD adapter for data acquisition.
- Gradual migration of an application from BOOL-based to DWORD-based adapter interfaces.
- Prototypical connection between different libraries or components with incompatible adapter types.

Comparison with Similar Function Blocks

- **BOOL_TO_DWORD** (function block): Performs pure data type conversion, but without adapter encapsulation.
- **AX_TO_AD** specializes in adapter interfaces and is therefore particularly suitable for exchanging data between different communication patterns in 4diac.
- Similar function blocks exist for other type combinations, e.g., `INT_TO_REAL_AD` or `WORD_TO_BOOL_AD`.

Conclusion

The `AX_TO_AD` function block offers a simple and clean way to convert a BOOL adapter to a DWORD adapter without requiring the user to write any conversion logic. It is ideally suited for linking different adapter worlds in 4diac projects.

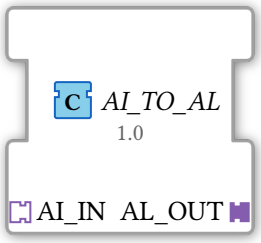
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Numeric Index (unidirectional)

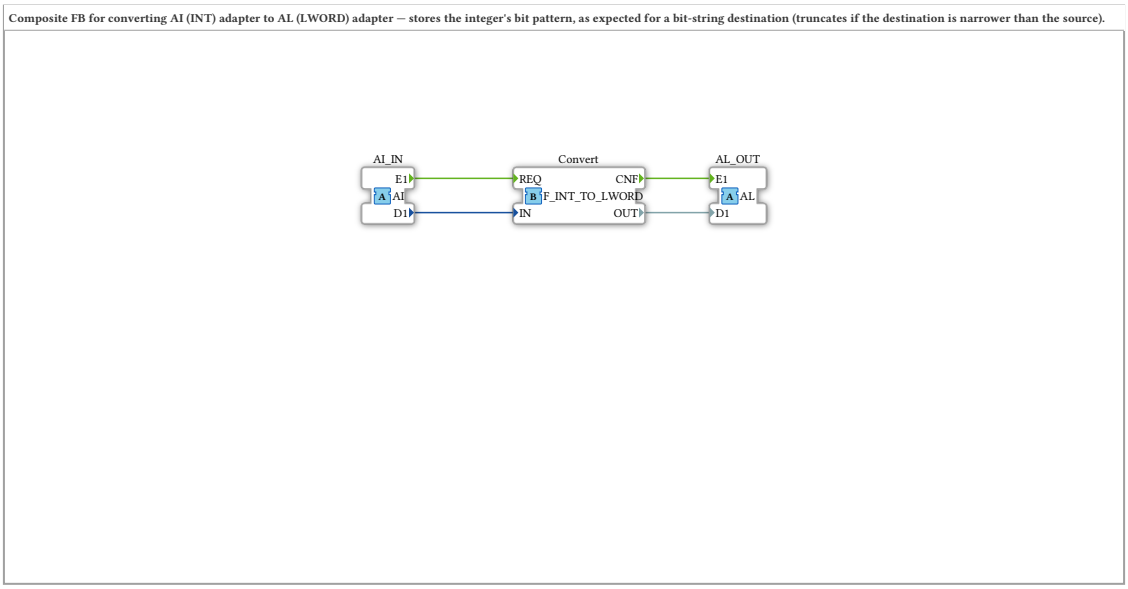
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AI_TO_AL



INT Adapter Input – *AI*

AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The composite function block **AI_TO_AL** converts an INT value provided via the unidirectional AI adapter (type `adapter::types::unidirectional::AI`) into an LWORD value output via the unidirectional AL adapter (type `adapter::types::unidirectional::AL`). The block encapsulates the conversion logic, thus enabling a clean separation between INT- and LWORD-based adapter interfaces.

Interface Structure

Event Inputs

No standalone event inputs on the block. Event control is handled exclusively via the integrated socket `AI_IN`, which provides the event `E1` as a trigger for the conversion.

Event Outputs

No independent event outputs on the block. The result of the internal conversion is output via the plug `AL_OUT` as event `E1`.

Data Inputs

No independent data inputs on the block. The INT value to be converted is read into the block via the socket `AI_IN` (data point `D1`).

Data Outputs

No independent data outputs on the block. The converted LWORD value is output via the plug `AL_OUT` (data point `D1`).

Adapter

Label	Type	Direction	Comment

AI_IN	adapter::types::unidirectional::AI	Socket (Input)	INT Adapter Input
AL_OUT	adapter::types::unidirectional::AL	Plug (Output)	LWORD Adapter Output

Functionality

1. An incoming event E1 at the adapter AI_IN activates the data conversion.
2. The internal function block F_INT_TO_LWORD (from the library iec61131::conversion) is triggered via its event input REQ and converts the INT value present at IN into an LWORD value.
3. After the conversion is complete, the function block signals this via the event output CNF.
4. This event is forwarded to the adapter AL_OUT as event E1 and simultaneously outputs the converted LWORD value via D1.

This event is forwarded to the adapter AL_OUT as event E1 and simultaneously outputs the converted LWORD value via D1.

... The data paths are:

- AI_IN.D1 ⟶ Convert.IN
- Convert.OUT ⟶ AL_OUT.D1

The event paths are:

- AI_IN.E1 ⟶ Convert.REQ
- Convert.CNF ⟶ AL_OUT.E1

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Block:** The AI_TO_AL is implemented as a composite and does not contain its own algorithmic logic. It delegates the conversion entirely to the defined function block F_INT_TO_LWORD.
- **Standard Conversion:** The conversion function used is part of the IEC 61131 library and ensures standards-compliant type conversion from INT (16-bit) to LWORD (64-bit).
- **Adapter-Based:** The function block operates exclusively via the specified adapter interfaces and can therefore be seamlessly integrated into an adapter-based architecture.
- **License and Version:** The function block is licensed under the Eclipse Public License 2.0 (EPL-2.0) and was released in version 1.0 by HR Agrartechnik GmbH.
- **Date of Creation:** February 17, 2026, Author: Franz Höpfinger.

State Overview

As a composite function block, AI_TO_AL does not have its own internal state machine. The entire state logic is controlled by the integrated function block F_INT_TO_LWORD, which executes a simple request-acknowledgment sequence (REQ/CNF). The function block is therefore purely data flow-driven and does not have persistent states.

Application Scenarios

- **Bus Coupling:** When a sensor or actuator delivers an INT value via an AI adapter, but the downstream system expects an LWORD value via an AL adapter (e.g., when connected to a fieldbus with 64-bit data types).
- **Protocol Adaptation:** In heterogeneous automation systems where different adapter types (AI and AL) need to be interconnected.
- **Data Preparation:** Before further processing of INT data in a controller that internally uses LWORD.

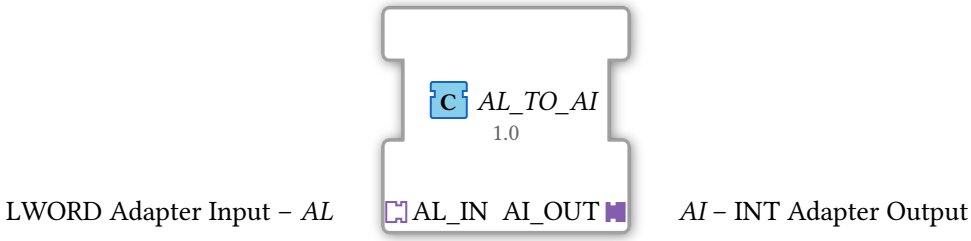
Comparison with Similar Components

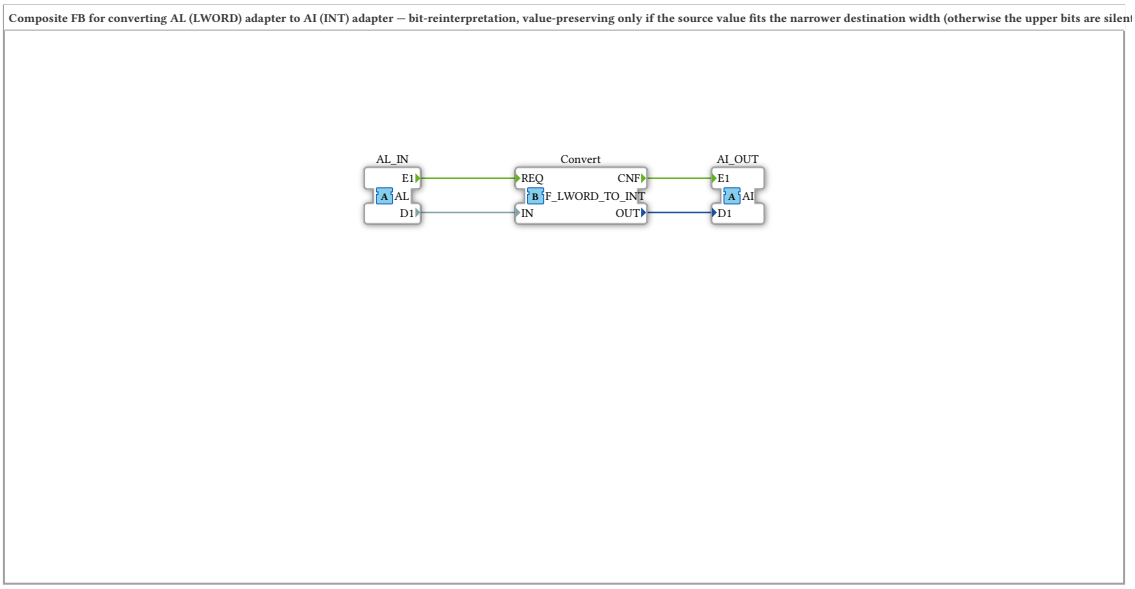
- **Other Conversion Adapters:** Similar composite components exist for other data type pairs, such as WORD_TO_DWORD, BYTE_TO_WORD, or DINT_TO_LWORD. The AI_TO_AL is specifically designed for the INT ⟶ LWORD combination.
- **Direct Conversion:** Alternatively, conversion could be performed directly with F_INT_TO_LWORD without adapter encapsulation. The advantage of AI_TO_AL lies in its standardized adapter interface, which facilitates exchange and reusability in adapter-based networks.

Conclusion

The `AI_TO_AL` function block is a simple yet valuable adapter converter that bridges the gap between INT- and LWORD-based adapter interfaces. By encapsulating the conversion logic within a single composite block, the application logic remains clear and reusability is increased. It is particularly well-suited for use in modular automation systems based on unidirectional adapter communication.

AL_TO_AI





• – ○ • – ○ • – ○ •

Introduction

The AL_TO_AI function block is a composite block that converts a unidirectional AL adapter (LWORD) into a unidirectional AI adapter (INT). It serves as an interface converter for adapter-based communication and enables easy integration between components that use different data types. The conversion is performed internally using the standardized IEC 61131-3 function block F_LWORD_TO_INT.

Interface Structure

Event Inputs

The function block does not have explicit event inputs. Event control is handled indirectly via the socket adapter **AL_IN**, which provides the event **E1**.

Event Outputs

The function block does not have explicit event outputs. Event control is handled indirectly via the plug adapter **AI_OUT**, which provides the event **E1**.

Data Inputs

The function block (FB) has no explicit data inputs. The data to be received is provided as an LWORD via the socket adapter **AL_IN**.

Data Outputs

The FB has no explicit data outputs. The converted data is output as an INT via the plug adapter **AI_OUT**.

Adapters

Adapter	Direction	Type	Description
AL_IN	Socket (Input)	`adapter::types::unidirectional::AL`	Provides an LWORD (via data port D1) and an event (E1).
AI_OUT	Plug (Output)	`adapter::types::unidirectional::AI`	Provides an INT (via data port D1) and an event (E1).

Functionality

The FB acts as a transparent converter between the two adapters. As soon as the event **E1** arrives at socket **AL_IN**, the internal function block **F_LWORD_TO_INT** is triggered with the passed LWORD value (from **AL_IN.D1**). The conversion block converts the value to an INT and signals completion via the event **CNF**. This event is forwarded directly to plug **AI_OUT** (E1), and the converted INT (from **F_LWORD_TO_INT.OUT**) is output on **AI_OUT.D1**. The entire conversion occurs synchronously within a single event cycle.

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- **Composite Architecture**: The function block (FB) internally uses a standardized IEC 61131-3 function block (`F_LWORD_TO_INT`), which can be implemented platform-independently.
- **Unidirectional Adapters**: Both the input and output adapters are of type `unidirectional`, meaning they only support one data and event direction (from socket to plug).
- **No Internal Logic**: The FB contains no state logic or sequences; it functions solely as a coupler and converter.
- **Type Conversion**: Converting from LWORD (64-bit) to INT (16-bit) can lead to data loss (truncation of higher-order bits), depending on the value range. This must be considered in the application context.

State Overview

The composite function block does not have its own states. The behavior is entirely determined by the internal **F_LWORD_TO_INT** function, which is a pure function without state. The sequence control is event-driven: an input event is immediately followed by an output event.

Application Scenarios

- **Adapter Bridge**: Connecting a component that provides LWORD via an AL adapter to a component that expects INT via an AI adapter.
- **System Integration**: Use in heterogeneous automation systems where different adapter types interact.
- **Protocol Adaptation**: Data conversion in a service-oriented architecture (e.g., 4diac) to adapt interfaces from various IEC 61499 or IEC 61131 libraries.

Comparison with Similar Function Blocks

- **AL_TO_AI** is specific to the combination of unidirectional AL and AI adapters. Other converters, such as `LWORD_TO_INT`, operate at a pure data level without adapter encapsulation.
- The 4diac ecosystem includes other adapter converters (e.g., `AI_TO_AL`) that cover the reverse direction. `AL_TO_AI` fills the gap for unidirectional LWORD→INT conversion.
- Compared to directly using `F_LWORD_TO_INT`, the Composite FB offers the advantage of a standardized adapter interface, enabling simple plug-and-play integration.

Conclusion

The **AL_TO_AI** function block is a useful, lightweight converter for switching between unidirectional LWORD and INT adapters. It encapsulates the proven IEC 61131-3 conversion in an adapter-friendly format, thus facilitating the reuse and replacement of components in IEC 61499-based systems. Thanks to

its event-driven handling, it integrates seamlessly into event-driven workflows and requires no additional configuration.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

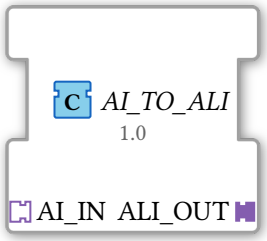
]

Numeric Index (unidirectional)

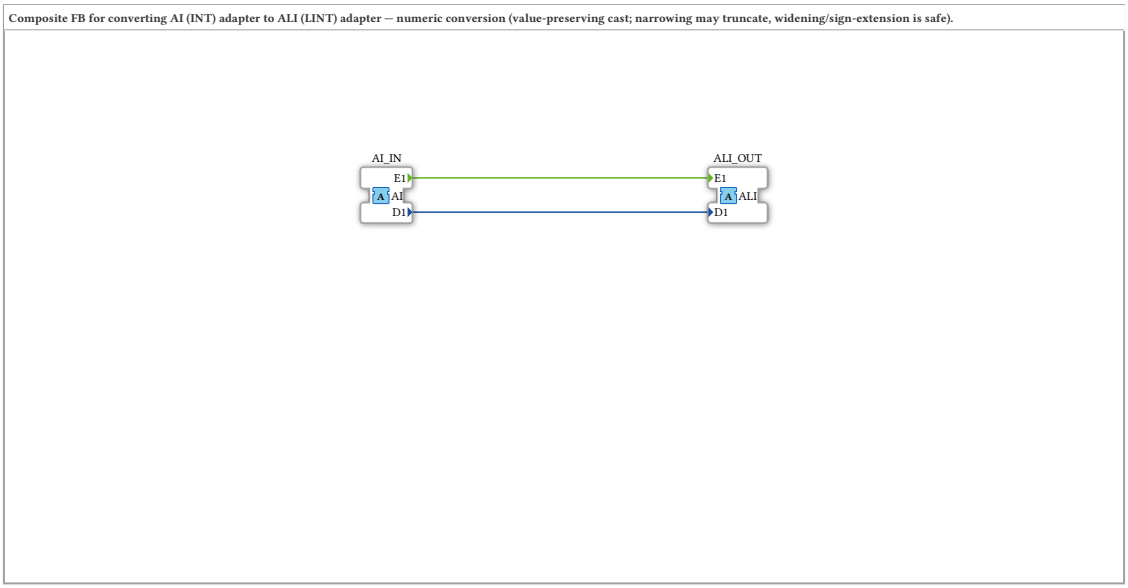
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AI_TO_ALI

INT Adapter Input – *AI*



ALI – LINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The AI_TO_ALI function block is a composite function block that converts an AI (INT) input adapter into an ALI (LINT) output adapter. It serves as a simple conversion interface between different adapter types for unidirectional communication.

Interface Structure

Event Inputs

No dedicated event inputs. Events are transmitted via the AI_IN input adapter.
Adapter AI_IN provides event input E1.

Event Outputs

No dedicated event outputs. Events are transmitted via the ALI_OUT output adapter.
Adapter ALI_OUT provides event output E1.

Data Inputs

No dedicated data inputs. Data transmission occurs via the AI_IN input adapter.
Adapter AI_IN provides data input D1 (data type INT).

Data Outputs

No dedicated data outputs. Data is passed through the output adapter ALI_OUT.
Adapter ALI_OUT provides data output D1 (data type LINT).

Adapters

- **AI_IN** (Socket): Type `adapter::types::unidirectional::AI` – INT input interface.
- **ALI_OUT** (Plug): Type `adapter::types::unidirectional::ALI` – LINT output interface.

Note: The actual event and data interfaces are defined by the adapters. The function block itself does not have its own interfaces.

Functionality

The function block operates as a simple pass-through: The event and data signal from input adapter AI_IN is forwarded directly to output adapter ALI_OUT. The INT data value is implicitly converted to a LINT value, as LINT covers a larger range of values. The conversion occurs without any additional logic or processing.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- This is a composite function block (FB) that does not implement any functionality of its own, but simply connects two adapters.
- The conversion from INT to LINT is lossless, as LINT (64-bit) covers the entire value range of INT (32-bit).
- The FB is designed for use in unidirectional communication paths where data type adaptation between adapter interfaces is required.
- The use of adapters enables loose coupling between components.

State Overview

The FB does not have its own state machine, as its functionality is limited to the pure transmission of events and data. The internal state is irrelevant; the behavior is deterministic and reacts instantaneously to incoming events.

•

Application Scenarios

- **Interface Adaptation:** One module returns an INT value via an AI adapter, while a subsequent module expects a LINT value via an ALI adapter.
- **Extension of Value Ranges:** If an existing INT data path needs to be converted to a larger value range, this function block (FB) can be used as an intermediary.
- **Adapter Protocol Conversion:** In heterogeneous systems where different adapter types are used, this FB serves as a simple converter.

Comparison with Similar Function Blocks

- **AI_TO_ALI** is specific for converting from AI to ALI. Other function blocks could, for example, implement AI_TO_AL (INT to WORD) or different directions (ALI_TO_AI).
- Compared to a generic type converter (e.g., with parameterization), this FB is specialized and therefore easier to use, but less flexible.

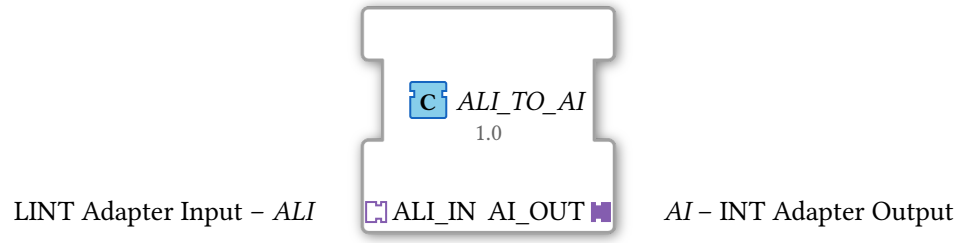
Composite function blocks like this one offer a clear separation of responsibilities and can be stored in libraries for standard conversions.

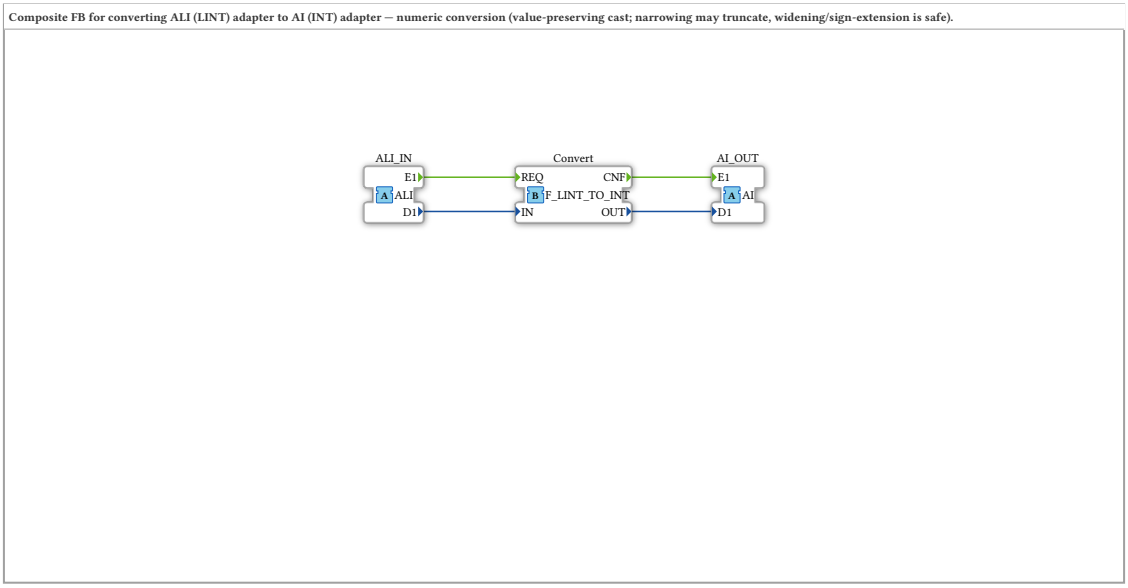
Conclusion

The AI_TO_ALI function block is a useful tool for adapter conversion in IEC 61499-based systems. It enables seamless integration of components with different data types at the adapter level. Its simple, pass-

through nature makes it reliable and easy to understand. It is particularly well-suited for projects that rely on the adapter-based, unidirectional communication model.

ALI_TO_AI





• – ○ • – ○ • – ○ •

Introduction

The **ALI_TO_AI** function block is a composite function block (FB) that converts an **ALI adapter** (input of a LINT value interface) into an **AI adapter** (output of an INT value interface). It serves to convert data from a LINT-based adapter to the INT-based adapter type without requiring the user to implement the conversion themselves.

Interface Structure

Event Inputs

Name	Data Type	Description
ALI_IN	Event	Control signal of the ALI adapter, triggers the conversion

Event Outputs

Name	Data Type	Description
AI_OUT	Event	Confirmation signal from the AI adapter after successful conversion

Data Inputs

Name	Data Type	Description
ALI_IN.D1	LINT	The LINT value to be converted

Data Outputs

Name	Data Type	Description
AI_OUT.D1	INT	The converted INT value

Adapter

Direction	Name	Type	Description
Socket	ALI_IN	ALI (unidirectional)	Returns LINT values and control events
Plug	AI_OUT	AI (unidirectional)	Outputs converted INT values and events

Functionality

This function block operates as a simple coupling of an ALI socket to an AI plug. Internally, the available conversion function block **F_LINT_TO_INT** is used.

1. An incoming event at **ALI_IN.E1** triggers the conversion function block.
2. Simultaneously, the current value of **ALI_IN.D1** is passed to the input of the converter. 3. After the conversion is complete, the result is forwarded to **AI_OUT.D1**.
3. An acknowledgment event is output via **AI_OUT.E1**.

The entire process is atomic within a single cycle – delays only occur due to the runtime of the conversion block.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Pure Adapter Coupling**: The block does not contain its own state logic or timing.
- **Use of IEC 61131 Conversion**: The actual type conversion is performed by the standardized block `F_LINT_TO_INT`.
- **Simple Composition**: All connections are direct and without intermediate logic.
- **Type- and Event-Driven**: The conversion is only executed when an event occurs at the ALI input.

State Overview

As a composite function block without its own state diagram, **ALI_TO_AI** does not have its own state machine. The internal conversion function block `F_LINT_TO_INT` operates according to the simple pattern:

- **IDLE**: Waiting for an event.
- **CONVERT**: Executing the conversion and outputting the result.

Application Scenarios

- **System Coupling**: When a sensor module provides LINT values via an ALI adapter, but the control system expects INT values via an AI adapter.
- **Protocol or Interface Adaptation**: Integration of components from different manufacturers that use different adapter types.
- **Data Path Cleanup**: Simplifying the signal flow through defined conversion at a central location.

Comparison with Similar Function Blocks

Function Block	Function	Difference
ALI_TO_AI	LINT → INT via adapter	Specifically for unidirectional ALI/AI interfaces
LINT_TO_INT	Direct conversion LINT → INT	No adapter integration – pure data conversion
AI_TO_ALI	INT → LINT (reverse direction)	Same structure, but opposite data direction

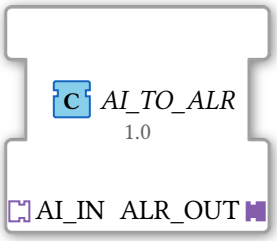
Conclusion

ALI_TO_AI is a small but useful compound block that bridges the gap between two common IEC 61499 adapter types. Encapsulating the conversion logic simplifies reuse and accelerates system integration. The block is robust because it uses a standardized IEC 61131 converter and requires no configuration.

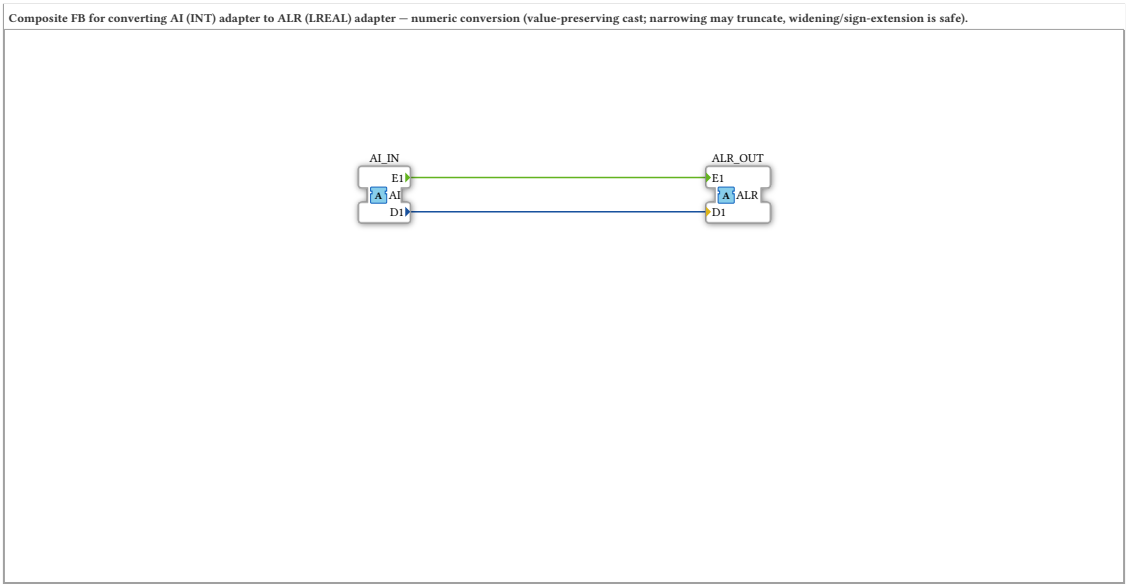
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AI_TO_ALR



INT Adapter Input – *AI* *ALR* – LREAL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AI_TO_ALR** is a composite function block that converts an **AI (INT adapter)** to an **ALR (LREAL adapter)**. It serves as a pure conversion interface by directly routing the event and data channels of the input adapter to the output adapter. The actual type conversion from Integer (INT) to Long Real (LREAL) is handled by the underlying adapter definitions.

Interface Structure

Event Inputs

The function block does not have direct event inputs. The event is received via the socket adapter `AI_IN`:

- **AI_IN.E1** – Event of the INT adapter (input).

Event Outputs

There are no direct event outputs here either. The event is sent via the plug adapter `ALR_OUT`:

- **ALR_OUT.E1** – Event of the LREAL adapter (output).

Data Inputs

- **AI_IN.D1** – Data value of type INT (via the adapter).

Data Outputs

- **ALR_OUT.D1** – Data value of type LREAL (via the adapter).

Adapters

- **AI_IN** (Socket) – Adapter of type `AI` (INT adapter), input interface.
- **ALR_OUT** (Plug) – Adapter of type `ALR` (LREAL adapter), output interface.

Functionality

This component connects the signals of the socket adapter `AI_IN` directly to the plug adapter `ALR_OUT`:

- **Event Path:** The event received by `AI_IN.E1` is forwarded unchanged to `ALR_OUT.E1`.
- **Data Path:** The INT value received by `AI_IN.D1` is passed to `ALR_OUT.D1`. The adapters themselves implement the necessary type conversion from INT to LREAL.

Since this is a composite component without its own algorithm or state machine, no further processing takes place. The component functions purely as a routing and adaptation layer between different adapter types.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Pure Passthrough Structure:** No internal logic, no latency, and no buffering.
- **Adapter Definition Dependency:** Correct operation requires that the adapters `AI` and `ALR` are defined according to the 4diac standard and include the INT>LREAL conversion.
- **Composite Component:** Easily extensible, e.g., through additional filters or network scaling.
- **Package Structure:** The component is located in the package `adapter::conversion::unidirectional`.

State Overview

The component does not have its own state diagram (ECC). Signal processing is purely event-driven via the passed-through adapter events. Therefore, a state is not defined.

Application Scenarios

- **System Integration:** Connecting an INT-based sensor (e.g., digital value from a counter) to a component that expects LREAL values (e.g., a control algorithm).
- **Adapter Bridge:** Use in modular automation systems where different adapter types interact.
- **Prototype Development:** Rapid coupling of test modules with different data formats.

Comparison with Similar Modules

While 4diac libraries often contain separate converter modules for basic data types (e.g., `INT_TO_REAL`), `AI_TO_ALR` encapsulates the conversion at the adapter level. The advantage lies in the reusability of adapter interfaces and the easy exchange of the conversion logic with other adapters. The disadvantage is the dependence on the specific adapter definitions.

Conclusion

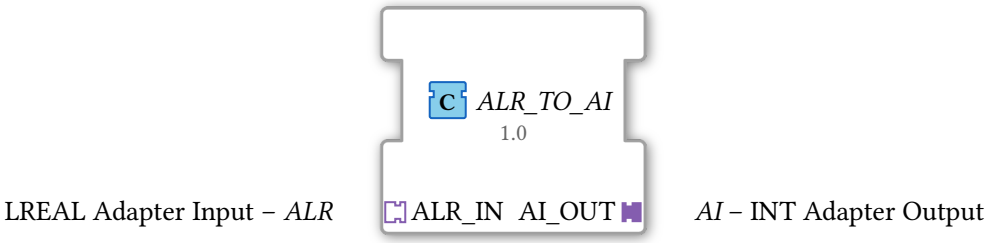
The **AI_TO_ALR** function block is a minimalist yet effective composite function block for type conversion between INT and LREAL adapters. It reduces wiring effort in complex control networks and enables a clean separation of interfaces and conversion logic. It is an ideal solution for applications requiring simple pass-through without additional processing.

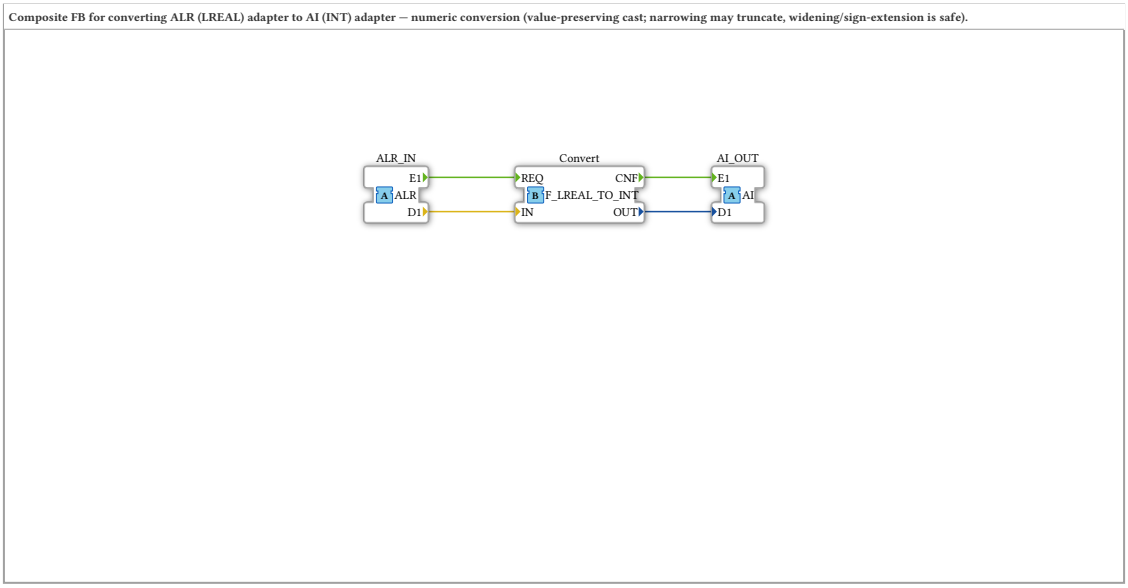
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ALR_TO_AI





• – ○ • – ○ • – ○ •

The **ALR_TO_AI** function block is a composite block that converts an ALR (LREAL data) adapter to an AI (INT data) adapter. It is used to convert signals from agricultural technology (e.g., sensor values) to a standardized integer-based adapter interface. The block encapsulates the conversion and event forwarding, thus simplifying integration into existing 4diac applications.

The function block does not have its own top-level event inputs. Event control is handled via the connected socket `ALR_IN`:

Input	Type	Description
`ALR_IN`	ALR (Socket)	Adapter that triggers the conversion via its event (`E1`)

The FB does not have any independent event outputs at the top level. Event feedback is provided via the connected plug `AI_OUT`:

Output	Type	Description
`AI_OUT`	AI (Plug)	Adapter that outputs the converted event (`E1`) after successful conversion

The **data input signal** is provided via socket `ALR_IN`:

Adapter Input	Data Type	Description
`ALR_IN.D1`	LREAL	The floating-point value to be converted

The **data output signal** is provided via plug `AI_OUT`:

Adapter Output	Data Type	Description
`AI_OUT.D1`	INT	The converted integer value

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter
The function block has two adapter interfaces:

- **Socket (Input)**: `ALR_IN` – Adapter for unidirectional LREAL data (Type `adapter::types::unidirectional::ALR`)
- **Plug (Output)**: `AI_OUT` – Adapter for unidirectional INT data (Type `adapter::types::unidirectional::AI`)

Functionality
The **ALR_TO_AI** function block contains an internal conversion block `F_LREAL_TO_INT` from the IEC 61131 library.

1. When socket `ALR_IN` sends an event at its output `E1`, it is forwarded to the input `REQ` of the internal converter.
2. The converter transforms the incoming LREAL value (`ALR_IN.D1`) into an INT value and outputs the result via its output `OUT`, along with an acknowledgment event `CNF`.
3. The `CNF` event is passed to plug `AI_OUT.E1`, informing the downstream component about data availability.
4. In parallel, the converted INT value is applied to the data output `AI_OUT.D1`.

The entire process is synchronous and performed in a single step.

- The function block (FB) is a **composite function block** – it does not have its own state machine (ECC) but implements the logic via an internal subnetwork.
- The conversion follows the IEC 61131-3 function `LREAL_TO_INT`: decimal places are truncated (truncation to zero), and overflows or underflows are implementation-dependent.
- The function block is designed as a unidirectional adapter coupling – no feedback channels are supported.

As a composite FB, **ALR_TO_AI** does not have its own state diagram. The internal converter `F_LREAL_TO_INT` operates in an event-driven manner: An event at input `REQ` triggers the conversion, and output `CNF` signals the end of the operation. The function block behaves like a transparent block for event and data forwarding.

- **Sensor Value Processing**: Conversion of an LREAL signal (e.g., from an analog sensor adapter) into an INT signal, which is then processed by a PLC or a controller with integer-based adapters.
- **Adapter Bridge**: Connecting adapter types of different physical units when only the data type, not the scaling, needs to be changed.
- **Interface Adaptation**: Used in agricultural control systems (e.g., HR Agricultural Technology – general), where LREAL values from sensors are mapped to a CAN-based INT adapter interface.

FB	Function	Difference
ALR_TO_AI	Converts LREAL ⟶ INT via adapter	Composite, no dedicated state machine; uses IEC 61131 converter
`F_LREAL_TO_INT`	Pure data conversion without adapter	Direct FB, requires separate adapter connection
`AI_TO_ALR`	Reverse direction INT ⟶ LREAL	Different data type and conversion direction

The **ALR_TO_AI** function block simplifies reuse by encapsulating the adapter logic and conversion in a single block.

The **ALR_TO_AI** is a practical composite function block for the standardized conversion of LREAL-based to INT-based adapter interfaces. It allows for a clean separation of data conversion and adapter logic and facilitates integration into larger automation projects. Thanks to its clear event forwarding and the use of an IEC 61131 basic function, it is reliable and easy to use.

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).

State Overview

Application Scenarios

Comparison with Similar Function Blocks

Conclusion

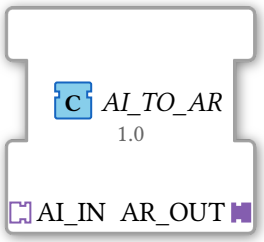
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

Numeric Index (unidirectional)

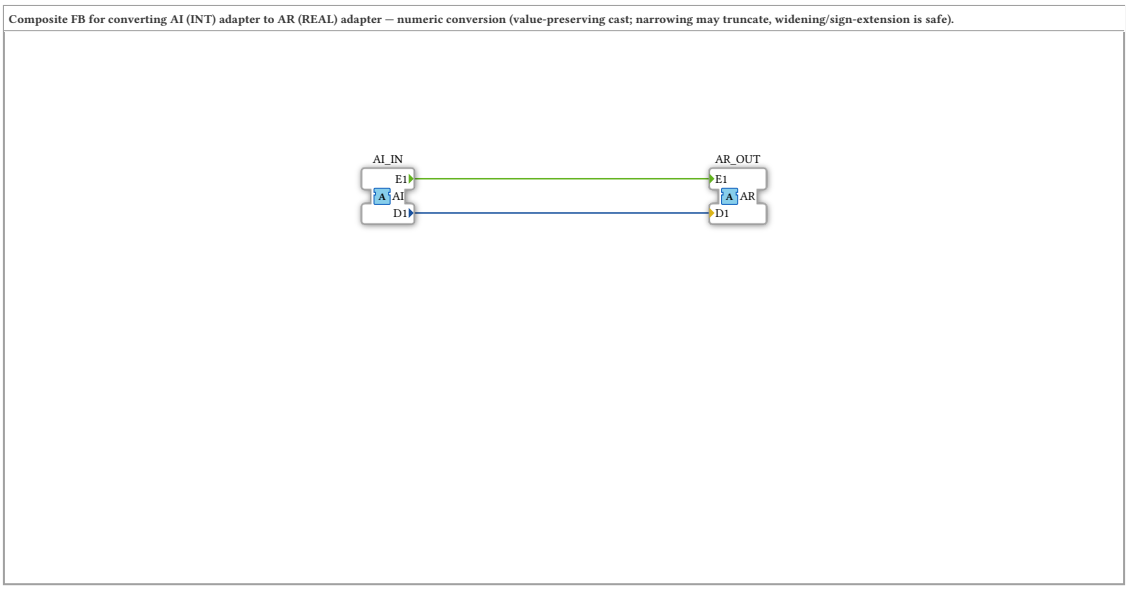
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AI_TO_AR

INT Adapter Input – *AI*



AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AI_TO_AR** function block is a composite module that enables interface conversion from an **AI** adapter (integer type) to an **AR** adapter (real type). It serves purely as a pass-through and adaptation unit between different adapter protocols in an industrial control system.

Interface Structure

Since **AI_TO_AR** does not have its own event or data interfaces, all communication takes place exclusively via the connected adapters.

Event Inputs

– not present – Event control is handled via the incoming adapter **AI_IN**.

Event Outputs

– not present – Event forwarding is handled via the outgoing adapter **AR_OUT**.

Data Inputs

– not available – The data is provided as an integer value via the **AI_IN** adapter.

Data Outputs

– not available – The data is output as a real value via the **AR_OUT** adapter.

Adapters

Name	Type	Direction	Description
AI_IN	`adapter::types::unidirectional::AI`	Socket	Integer Adapter (Input)
AR_OUT	`adapter::types::unidirectional::AR`	Plug	Real Adapter (Output)

Functionality

In the FB network of the composite block, the event and data lines of the two adapters are directly connected:

- The event **E1** of the AI_IN adapter is routed to the event **E1** of the AR_OUT adapter.
- The data value **D1** (type INT) of the AI_IN adapter is transferred to the data value **D1** (type REAL) of the AR_OUT adapter.

The block itself does not perform any explicit type conversion; the conversion from integer to real is either implicitly handled by the runtime environment or must be ensured by the connected context. Thus, the block functions purely as a pass-through for events and data between the two adapter interfaces.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- The block does not contain its own event or data interface; all its communication takes place via the adapters.
- There is no internal state machine (ECC) – the block operates purely event-driven through the connections.
- The use of adapters achieves loose coupling between the inputs and outputs, increasing the reusability and interchangeability of the interfaces.

State Overview

The component does not have a state machine. Processing occurs directly without delay or state dependency.

Application Scenarios

- **Adapter Conversion**: Use in systems where an integer-based sensor signal (AI) must be transferred to a component that expects a real signal (AR).

- **Interface Adaptation**: Use in modular control solutions to mediate between different adapter types without affecting the actual data processing.
- **Prototypes and Testing**: Quick assembly of different adapter components during the development phase.

Comparison with Similar Components

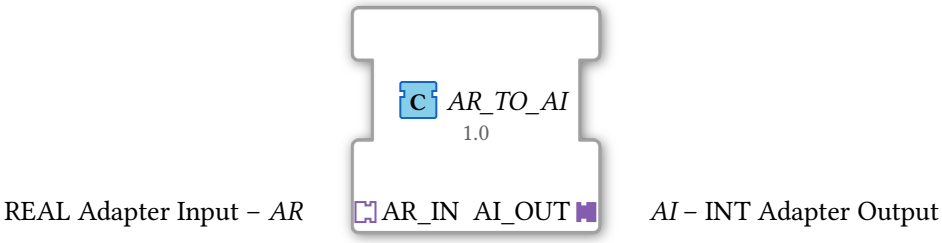
Component	Function	Special Feature
AI_TO_AR	Pass-through from INT to REAL via adapter	No dedicated logic, pure adapter switching
REAL_TO_INT	Conversion from REAL to INT (usually with rounding)	Often includes its own algorithm
ADAPTER_COUPLER	General adapter coupling without type conversion	Requires identical data types

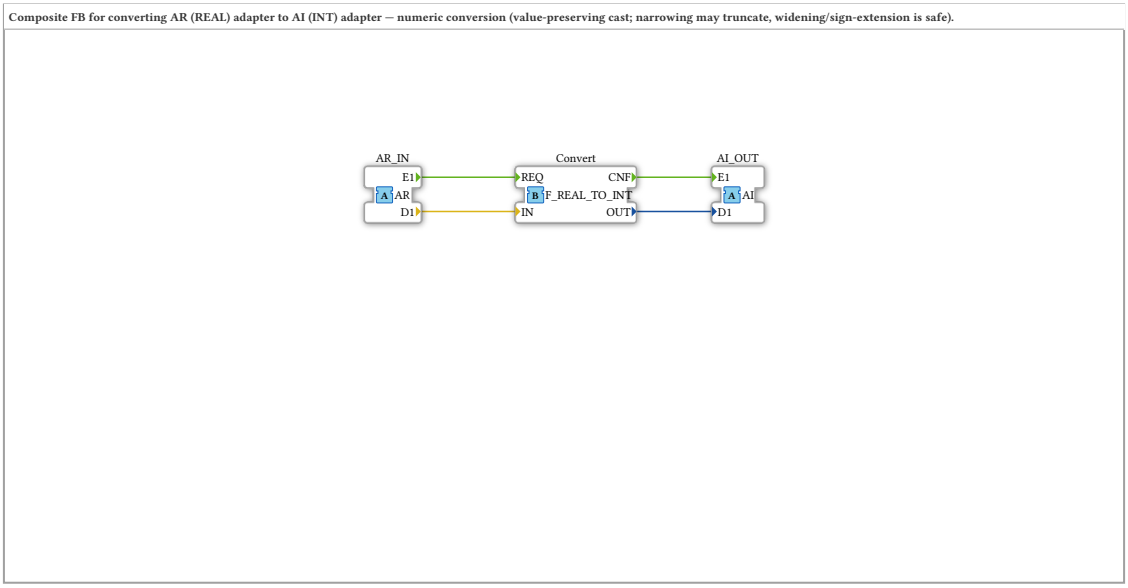
Unlike dedicated converters, **AI_TO_AR** does not perform active value conversion but relies on the compatibility of the connected adapters.

Conclusion

AI_TO_AR is a simple yet effective composite block for connecting two different adapter interfaces (INT to REAL). It is particularly suitable for cross-system signal transmission in IEC 61499-based controllers where the type conversion logic is located outside the block. Its clear structure makes it a useful tool for modular and flexible automation solutions.

AR_TO_AI





• – ○ • – ○ • – ○ •

Introduction

The **AR_TO_AI** function block is a composite module that converts a unidirectional AR adapter (input with data type REAL) into a unidirectional AI adapter (output with data type INT). It is used for numerical type conversion within 4diac networks and encapsulates an IEC 61131 conversion module.

Interface Structure

Event Inputs

The event input is provided via the **AR_IN** socket. The **AR_IN.E1** event triggers the conversion.

Event Outputs

The event output is provided via the **AI_OUT** plug. After successful conversion, the event **AI_OUT.E1** is sent.

Data Inputs

The data input is provided via the socket **AR_IN**. The input value **AR_IN.D1** is of type REAL.

Data Outputs

The data output is provided via the plug **AI_OUT**. The output value **AI_OUT.D1** is of type INT.

Adapters

Label	Direction	Type	Remark
AR_IN	Input	`adapter::types::unidirectional::AR`	REAL Adapter (Input)
AI_OUT	Output	`adapter::types::unidirectional::AI`	INT Adapter (Output)

Functionality

The function block (FB) is implemented as a composite and contains a single **F_REAL_TO_INT** block from the IEC 61131 library (`iec61131::conversion::F_REAL_TO_INT`).

As soon as an event arrives at socket **AR_IN.E1**, the value of **AR_IN.D1** (REAL) is passed to the input `IN` of the converter, and its processing is started (`REQ`). After the conversion is complete, the converter signals via `CNF` that the result is available at output `OUT` (INT). This result is immediately forwarded to the plug output **AI_OUT.D1** (INT), and the event **AI_OUT.E1** is triggered.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- The function block (FB) is fully event-driven; it does not have its own state machine.
- Conversion is performed according to the IEC 61131 standard: REAL is mapped to INT (decimal places are truncated, corresponding to a type conversion).
- The function block is located in the package `adapter::conversion::unidirectional`, which allows for clear separation from other conversion function blocks.
- The use of adapters improves encapsulation and reusability in 4diac projects.

State Overview

The **AR_TO_AI** function block does not have an internal state machine. Its operation is purely combinatorial and depends on the incoming event:

- **Waiting:** No event is pending at AR_IN.E1.
- **Converting:** Upon an event, the REAL value is converted immediately, and the result is output as an INT.

Application Scenarios

- **Sensor Connection:** Real-value sensors (e.g., analog voltage measurement) deliver REAL data via an AR adapter; downstream logic requires INT values (e.g., for counters or index calculations).
- **Protocol Conversion:** When a subsystem sends data in REAL format, but the target system only accepts INT data (e.g., with fieldbus gateways).

- **Test and Simulation Environments:** For easy conversion between REAL and INT adapters in test setups.

Comparison with Similar Function Blocks

- **AI_TO_AR:** Performs the reverse conversion (INT → REAL).
- **F_REAL_TO_INT (Direct):** The encapsulated basic function block is designed for point-to-point connections without an adapter. The AR_TO_AI offers an adapter-based interface that can be better integrated into adapter-oriented networks.
- **Custom Composite Blocks:** Other projects could create similar adapter conversions for other data types (e.g., LREAL, DINT).

Conclusion

The **AR_TO_AI** is a compact, adapter-based conversion block that enables the conversion of REAL to INT data in event-driven 4diac networks. By encapsulating the standard block `F_REAL_TO_INT` in an adapter interface, integration into existing adapter socket architectures is simplified, and the overall system readability is improved.

🌐 Related topic subpages on ms-muc-docs.de

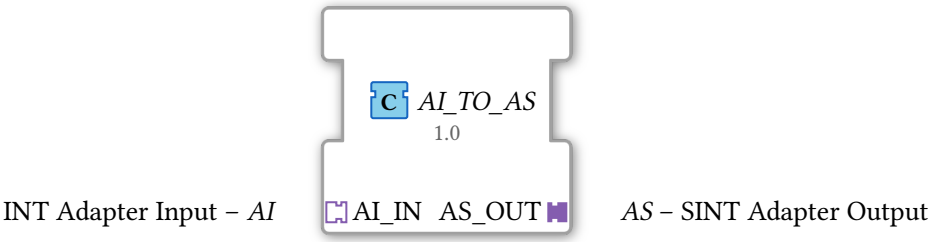
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

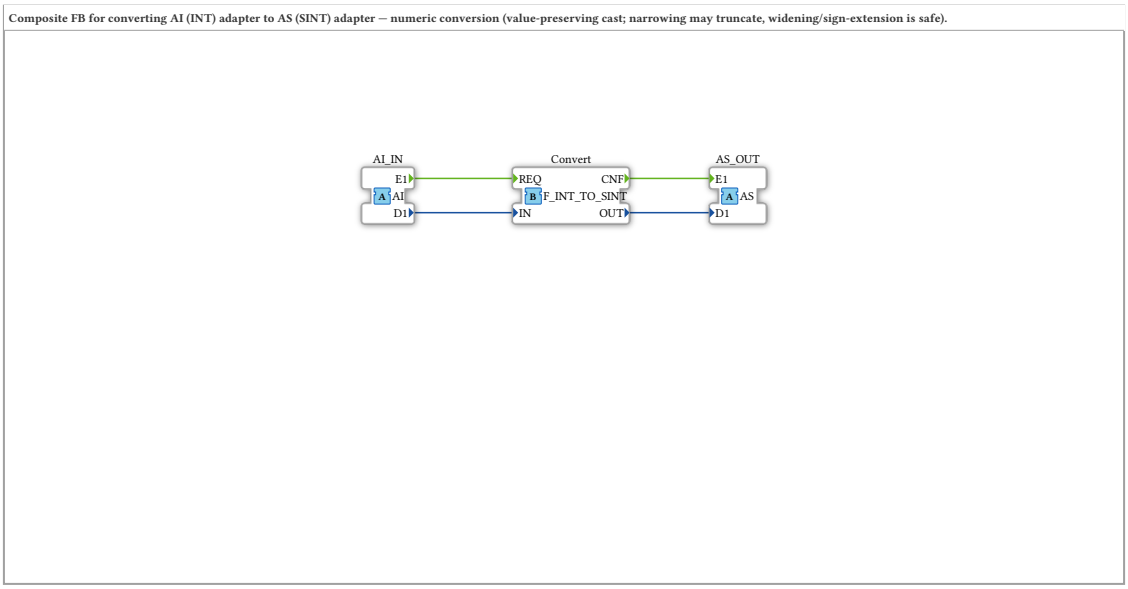
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AI_TO_AS





• – ○ • – ○ • – ○ •

Introduction

The function block **AI_TO_AS** is a composite block that converts an adapter of type `AI` (based on `INT`) into an adapter of type `AS` (based on `SINT`). It serves as an interface converter between different data formats in the IEC 61499 architecture.

Interface Structure

Event Inputs

The function block does not have its own event inputs. Event control is handled entirely via the adapter input `AI_IN` (see section Adapter). Internally, the adapter's event `E1` is processed.

Event Outputs

The function block (FB) does not have its own event outputs. The result event is provided via the adapter output `AS_OUT` (see section Adapter). After successful conversion, the event `E1` is triggered on the output adapter.

Data Inputs

The FB does not have its own data inputs. The value to be converted, `INT`, is read in via the adapter input `AI_IN` (data point `D1`).

Data Outputs

The FB does not have its own data outputs. The converted `SINT` value is output via the adapter output `AS_OUT` (data point `D1`).

Adapter

Adapter	Direction	Type	Description
`AI_IN`	Input (Socket)	`adapter::types::unidirectional::AI`	INT-based adapter input; provides the event `E1` and the data value `D1` (type `INT`).
`AS_OUT`	Output (Plug)	`adapter::types::unidirectional::AS`	SINT-based adapter output; receives the event `E1` and passes on the converted data value `D1` (type `SINT`).

Functionality

The function block performs a **type conversion from `INT` to `SINT`**.

As soon as an event `E1` arrives at the adapter input `AI_IN`, the corresponding data value `D1` (INT) is read. This value is passed to the internal function block `F_INT_TO_SINT`. The conversion is performed according to the IEC 61131-3 definition; the `INT` value is limited to the value range of `SINT` (-128...127). After successful conversion, the event `E1` is generated at the adapter output `AS_OUT`, and the converted `SINT` value is passed via `D1`.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Structure**: The function block is implemented as a composite that uses the standard function block `F_INT_TO_SINT`. This enables easy reuse and a clear separation of the conversion logic.
- **Adapter-based interface**: The inputs and outputs are implemented as unidirectional adapters (`AI` and `AS`), allowing flexible connection to other components with compatible adapter interfaces.
- **No range check**: The underlying function block `F_INT_TO_SINT` does not perform explicit error handling when a value exceeds its range. If the value of `INT` is outside the range of `SINT`, unexpected results or overflows may occur (depending on the target platform).

State overview

The function block does not have its own state machine. The internal logic follows the simple sequence of function block `F_INT_TO_SINT`:

State	Description
Idle	Waiting for an event at `AI_IN.E1`.
Convert	Upon arrival of `E1`, the value from `AI_IN.D1` is read, converted, and passed to `AS_OUT.D1`. Then, `AS_OUT.E1` is triggered. The function block immediately returns to the idle state.

Application Scenarios

- **Data Format Adaptation**: When an existing INT interface needs to be integrated into a system that processes only SINT values (e.g., older controllers or limited bus protocols).
- **Adapter Replacement**: Replacing an AI adapter with an AS adapter in an existing component structure without modifying the internal logic.
- **Prototypes/Test Setups**: Fast conversion of test signals between different data widths.

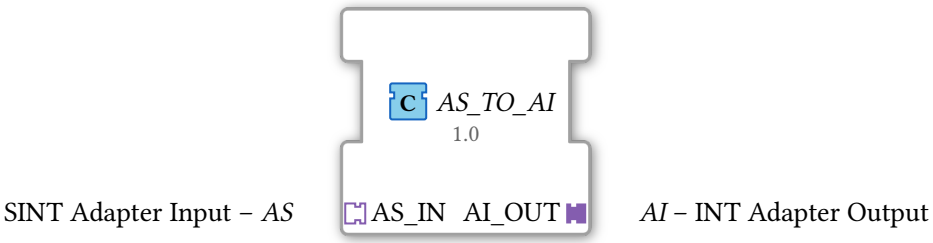
Comparison with Similar Components

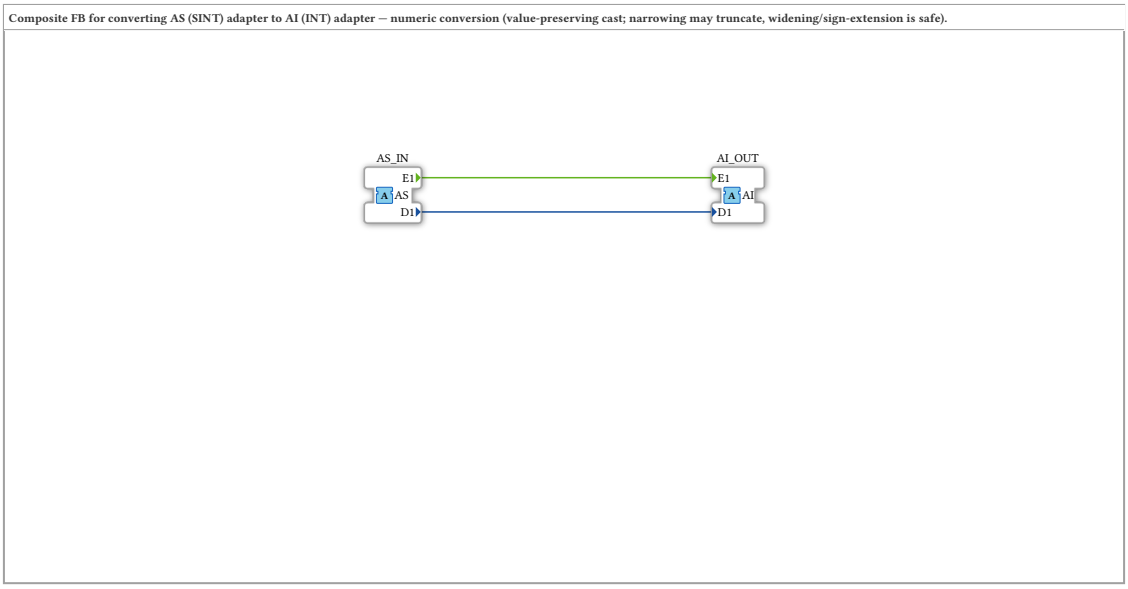
- **AI_TO_AS (INT → SINT)**: Specifically designed for converting INT to SINT using unidirectional adapters.
- **AY_TO_AS (DINT → SINT)**: Converts DINT values to SINT; essentially the same structure, but a different data type.
- **Direct Conversion Without Adapters**: Function blocks like `F_INT_TO_SINT` offer pure type conversion but require separate event and data connections. The `AI_TO_AS` encapsulates these in adapter logic.

Conclusion

The `AI_TO_AS` function block provides a simple and compact solution for converting an INT-based adapter to a SINT-based adapter. Its composite structure and clear interface definition make it easy to integrate into IEC 61499 applications. The lack of range checking should be considered in safety-critical applications. Overall, it is a useful tool for signal conditioning in heterogeneous automation systems.

AS_TO_AI





• — ○ • — ○ • — ○ •

Introduction

The function block `AS_TO_AI` is a composite function block that converts an AS adapter (data type SINT) to an AI adapter (data type INT). It serves as a simple converter for connecting components with different adapter types within the 4diac IDE.

Interface Structure

Event Inputs

- None

Event Outputs

- None

Data Inputs

- None

Data Outputs

- None

Adapter

- **AS_IN** (Socket, Type: `adapter::types::unidirectional::AS`)

Expects a SINT value and a corresponding event E1.

- **AI_OUT** (Plug, Type: `adapter::types::unidirectional::AI`)

Provides an INT value and an event E1.

Functionality

The function block forwards the event E1 from the AS_IN socket directly to the AI_OUT plug. Simultaneously, the data value D1 is transferred from the AS_IN socket to the AI_OUT plug. This involves an implicit type conversion from SINT (8-bit signed) to INT (16-bit signed). No further processing or state logic takes place.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).

- The type conversion from SINT to INT is performed automatically by the 4diac runtime environment without requiring any additional code blocks.
- The function block is implemented as a **Composite FB** and does not contain its own ECC (Execution Control Chart).
- It is located in the package `adapter::conversion::unidirectional`, which indicates unidirectional adapter conversion.

State Overview

Since it is a composite function block, `AS_TO_AI` does not have its own states. Its behavior is determined solely by the wiring of its internal connections.

Application Scenarios

- A sensor provides data via an AS adapter (SINT), but the connected processing component expects an AI adapter (INT).
- Migration from 8-bit to 16-bit data paths without changing the source or target component.
- Prototyping and system integration when different adapter versions interact.

Comparison with Similar Components

- **AI_TO_AS**: Performs the reverse conversion (INT ⟶ SINT).
- **AI_TO_AD** or similar: Converts between other integer data types (e.g., INT to DINT).
- Unlike these function blocks, `AS_TO_AI` operates losslessly, as the target type (INT) can fully map to the source type (SINT).

Conclusion

The `AS_TO_AI` function block is a simple yet effective tool for ensuring adapter compatibility in 4diac applications. It handles the automatic type conversion from SINT to INT, thus simplifying the integration of heterogeneous components without requiring manual adjustment of the data path.

🌐 Related topic subpages on ms-muc-docs.de

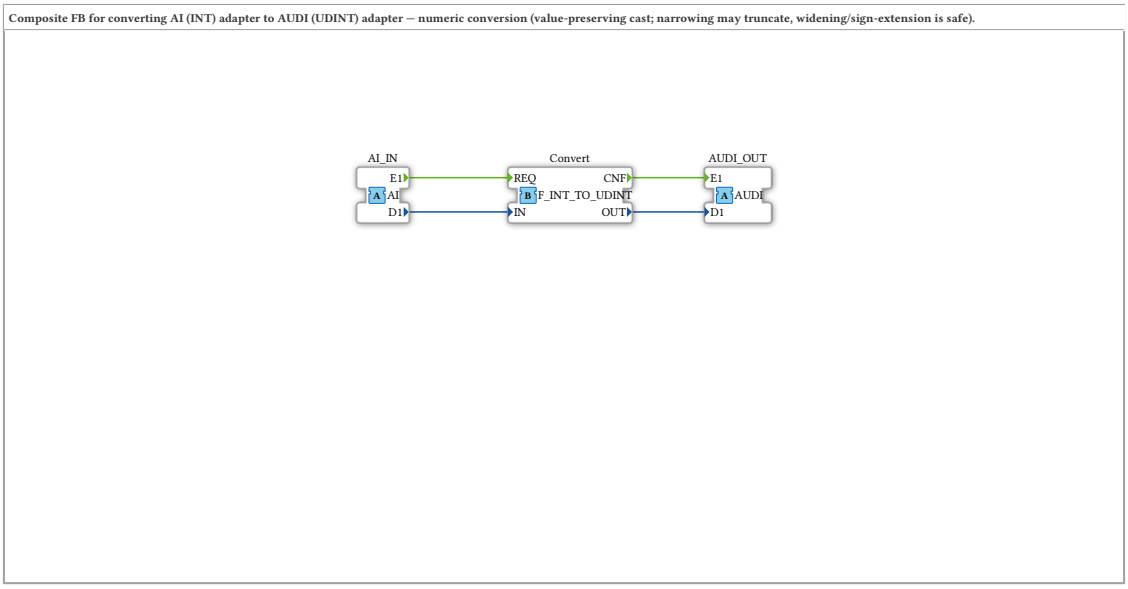
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AI_TO_AUDI





• — ○ • — ○ • — ○ •

Introduction

The function block `AI_TO_AUDI` is a composite module that converts an **AI adapter** (data of type `INT`) into an **AUDI adapter** (data of type `UDINT`). It serves as a unidirectional interface between different adapter types and enables seamless data conversion within the 4diac IDE.

Interface Structure

The module does not have traditional event and data inputs/outputs, but instead implements its inputs and outputs via **adapter interfaces**. The following ports are available via the internal adapters:

Event Inputs

Port	Adapter	Description
`E1`	`AI_IN`	Starts the conversion of the INT value.

Event Outputs

Port	Adapter	Description
`E1`	`AUDI_OUT`	Signals completion of the conversion.

Data Inputs

Port	Adapter	Data Type	Description
`D1`	`AI_IN`	`INT`	Integer value to be converted.

Data Outputs

Port	Adapter	Data Type	Description
`D1`	`AUDI_OUT`	`UDINT`	Converted unsigned integer.

Adapter

Direction	Name	Type	Description
Socket	`AI_IN`	`adapter::types::unidirectional::AI`	Input adapter (INT).

Plug	AUDI_OUT	adapter::types::unidirectional::AUDI	Output adapter (UDINT).

Functionality

The function block is implemented as a composite function block. Internally, it uses the standard conversion function block iec61131::conversion::F_INT_TO_UDINT.

Process:

1. An event arrives at the input E1 of socket AI_IN, accompanied by an INT value at D1.
2. This event is forwarded to the internal function block Convert (type F_INT_TO_UDINT).
3. The internal function block performs the type conversion from INT to UDINT (according to IEC 61131-3).
4. After successful conversion, an event is generated at the output CNF of the internal function block.
5. This event is passed to plug AUDI_OUT (event output E1), and simultaneously the converted UDINT value is available at data output D1.

This event is passed to plug AUDI_OUT (event output E1), and the converted UDINT value is simultaneously available at data output D1.

... The connections in the FBNetwork are:

- AI_IN.E1 ⟶ Convert.REQ
- Convert.CNF ⟶ AUDI_OUT.E1
- AI_IN.D1 ⟶ Convert.IN
- Convert.OUT ⟶ AUDI_OUT.D1

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block:** Simplifies the reuse and encapsulation of the conversion logic.
- **Standard Conversion:** Uses the IEC 61131-3 block F_INT_TO_UDINT, which ensures clean and portable type conversion.
- **Unidirectional Adapters:** Both the input and output adapters are implemented as unidirectional interfaces (socket and plug, respectively) – data flows in only one direction.
- **Event-driven:** The conversion is triggered by an event at the input adapter and acknowledged by an event at the output adapter (handshake principle).

State Overview

The function block itself does not have its own state machine, as it consists solely of a direct chain of adapter ports and the internal conversion block. The internal function block F_INT_TO_UDINT performs the conversion in one step – therefore, a state overview is not required.

Application Scenarios

- **Data Bridge Between Different Adapter Types:** When a higher-level system or library provides adapters of type AI (with INT data), but downstream components expect a AUDI adapter (with UDINT data).
- **Signal Conversion in Automation Solutions:** For example, when processing counter values or position data that arrive as signed 16-bit values but must be passed on as unsigned 32-bit values.
- **Modular Adapter Conversion:** Easily replace or extend the conversion logic by replacing the internal function block.

Comparison with Similar Function Blocks

Other adapter conversion function blocks exist in the library, e.g. B.:

- AI_TO_AI – forwarding without conversion
- AUDI_TO_AI – reverse direction (UDINT ⟶ INT)
- Direct use of the F_INT_TO_UDINT block with manual adapter wiring

The AI_TO_AUDI block not only encapsulates the type conversion but also the complete adapter connection, making the design clearer and preventing errors due to incorrect wiring.

Conclusion

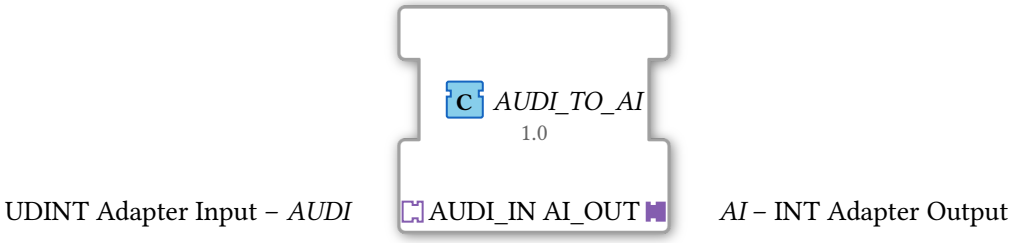
The `AI_TO_AUDI` composite block offers a simple yet clean solution for converting an INT-based adapter protocol to a UDINT-based protocol. Thanks to the use of the standardized IEC block and event-driven coupling, it is reliable and easily integrated into larger 4diac projects.

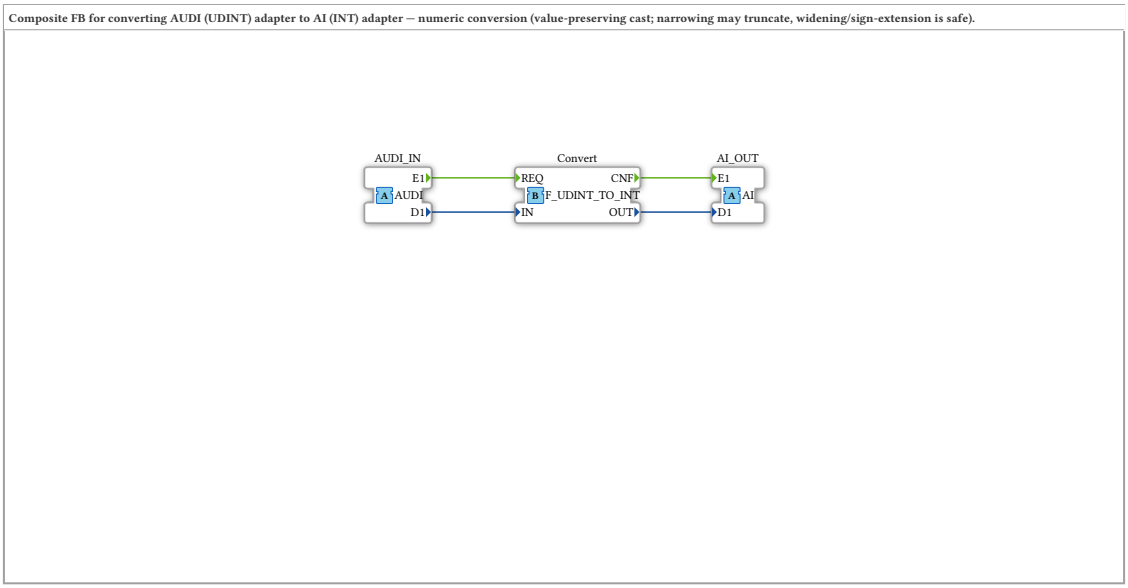
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUDI_TO_AI





• – ◦ • – ◦ • – ◦ •

Introduction

The composite function block `AUDI_TO_AI` converts a unidirectional AUDI adapter (data type `UDINT`) into a unidirectional AI adapter (data type `INT`). It encapsulates the conversion logic and enables easy integration into adapter-based communication structures according to IEC 61499.

Interface Structure

Event Inputs

The function block does not have direct event inputs. Event control is handled via socket `AUDI_IN` of type `adapter::types::unidirectional::AUDI`. This adapter provides an event input, `E1`, which triggers the conversion.

Event Outputs

The function block (FB) has no direct event outputs. Event output is provided via the plug `AI_OUT` of type `adapter::types::unidirectional::AI`. This adapter provides an event output, `E1`, which signals the completion of the conversion.

Data Inputs

No direct data inputs. The socket `AUDI_IN` provides the value to be converted, `UDINT`, via its data input `D1`.

Data Outputs

No direct data outputs. Plug `AI_OUT` outputs the converted `INT` value via its data output `D1`.

Adapters

- **Socket `AUDI_IN`**: Type `adapter::types::unidirectional::AUDI` – receives a unidirectional data value of type `UDINT` and an associated event.
- **Plug `AI_OUT`**: Type `adapter::types::unidirectional::AI` – sends a unidirectional data value of type `INT` and an associated event.

Functionality

The function block is implemented as a composite block. Internally, socket `AUDI_IN` is connected to the standard conversion block `iec61131::conversion::F_UDINT_TO_INT`, whose output is in turn linked to plug `AI_OUT`.

When an event is received via the adapter input `AUDI_IN.E1`, the corresponding data value `AUDI_IN.D1` is passed to the conversion block. After successful conversion of `UDINT` to `INT`, the event `Convert.CNF` is generated and output via `AI_OUT.E1`. Simultaneously, the converted value is available at `AI_OUT.D1`.

…

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- Uses a standardized IEC 61131 conversion block that can be implemented platform-independently.
- The conversion from `UDINT` (32 unsigned bits) to `INT` (16 signed bits) is potentially lossy. Depending on the implementation of the conversion block, an overflow or limit may occur if the value range is exceeded.
- The use of unidirectional adapters ensures a clean data flow and facilitates integration into modular systems.
- The function block does not have its own state machine; all logic is executed event-driven via the internal wiring.

State Overview

The function block does not have an internal state machine. Its functionality arises solely from the causal chaining of event and data connections. There is only one active state in which an incoming event-data combination is processed and the output is generated.

Application Scenarios

- Connecting a sensor that delivers measured values as `UDINT` to a control system that expects `INT` values (e.g., a PLC with 16-bit analog inputs).
- Integration into PROFINET or EtherCAT applications where adapters are required for type conversion between different data formats.
- Reusable encapsulation of the conversion in an adapter-based library for industrial automation solutions.

Comparison with Similar Function Blocks

- A direct conversion function block like `F_UDINT_TO_INT` requires separate event and data connections and is not designed for direct use in adapter interfaces.
- `AUDI_TO_AI` encapsulates the entire adapter conversion, thus simplifying reuse in architectures based on unidirectional adapters.
- Analogous function blocks for other data types (e.g., `REAL_TO_INT` or `DINT_TO_INT`) follow the same design pattern and can be implemented using the same scheme.

Conclusion

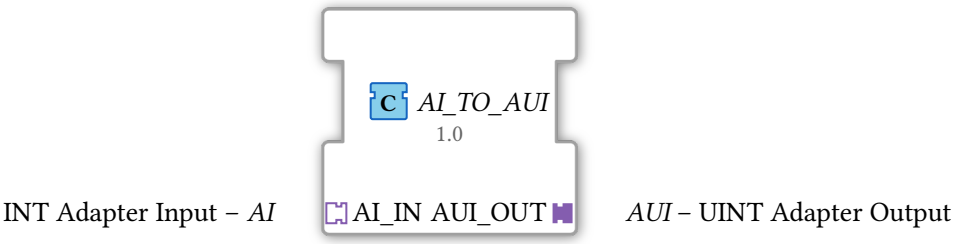
The composite function block `AUDI_TO_AI` provides a clean and reusable solution for lossy conversion of `UDINT` values to `INT` values using unidirectional adapters. It simplifies integration into IEC 61499 systems

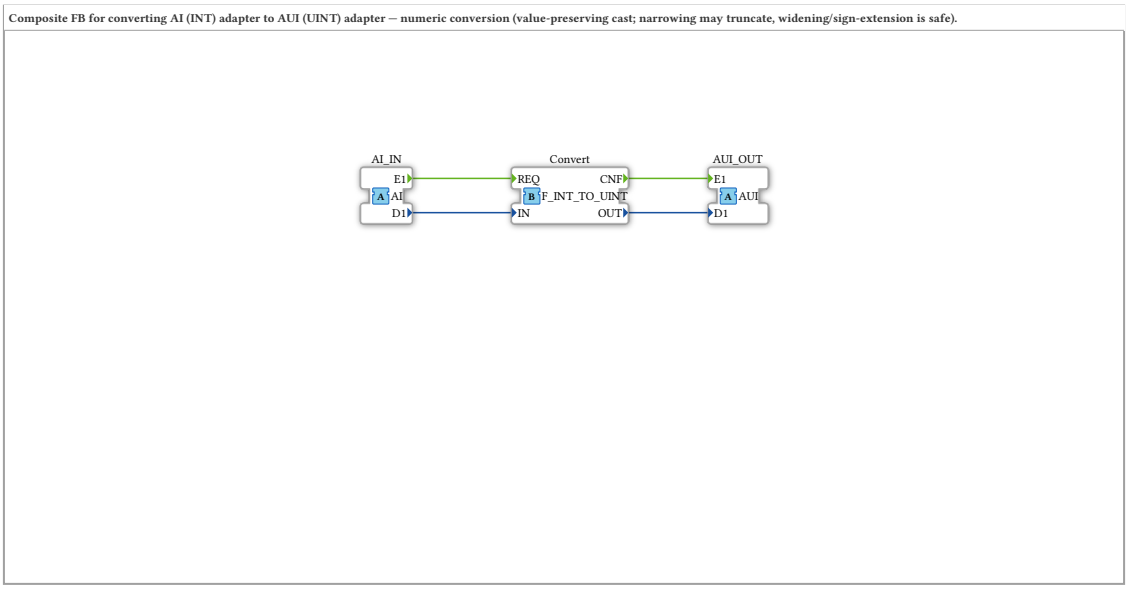
that rely on adapter communication and encapsulates the conversion logic in a single, clearly defined function block.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AI_TO_AUI





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_TO_AUI** is a composite block that converts an INT adapter (AI) into a UINT adapter (AUI). It serves as an interface converter in systems that rely on unidirectional adapter communication. Internally, it uses the block `F_INT_TO_UINT` from the IEC 61131 library to convert the integer value of `AI_IN` into an unsigned integer and make it available via `AUI_OUT`.

Interface Structure

Event Inputs

No separate event inputs are available. Event control is handled exclusively via the adapters (`AI_IN.E1` for request, `AUI_OUT.E1` for acknowledgement).

Event Outputs

No separate event outputs are available. Event feedback is provided via the `AUI_OUT.E1` adapter.

Data Inputs

No separate data inputs are available. The value to be converted is received via the `AI_IN.D1` adapter (type INT).

Data Outputs

No separate data outputs are available. The converted value is provided via the `AUI_OUT.D1` adapter (type UINT).

Adapters

Direction	Name	Type	Description
Socket (Input)	`AI_IN`	`adapter::types::unidirectional::AI`	INT adapter input. Returns the event `E1` and the data value `D1` (INT).
Plug (Output)	`AUI_OUT`	`adapter::types::unidirectional::AUI`	UINT adapter output. Provides the event `E1` and the data value `D1` (UINT).

Functionality

The function block operates as a simple pipeline:

1. An event is received via `AI_IN.E1`.
2. This event triggers the internal conversion block `F_INT_TO_UINT` via its `REQ` input.
3. The value of `AI_IN.D1` (INT) is forwarded to `Convert.IN`.
4. After the conversion is complete, an event is sent to `AUI_OUT.E1` via `Convert.CNF`.
5. The converted value (UINT) is available at `AUI_OUT.D1`.

The module is purely causal – it reacts to an incoming event and, after conversion, outputs an outgoing event.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-based communication**: Instead of classic input/output pins, the interfaces are implemented via adapters (`AI` and `AUI`). This enables loose coupling in modular systems.
- **Unidirectional data flow**: Both adapters are unidirectional, meaning they only transport data and events in one direction (input or output).
- **Packaging**: The module is located in the package `adapter::conversion::unidirectional`, which indicates its role as a converter between adapter types.
- **Type Conversion**: The conversion from `INT` to `UINT` is performed using the IEC 61131 function block `F_INT_TO_UINT`, which ensures value preservation for positive numbers. Negative values are converted to the corresponding UINT space (e.g., two's complement).

State Overview

The function block does not have an explicit state machine. Internally, it only executes the following sequence: Receive event → Trigger conversion → Send event. There are no delayed responses or persistent states.

Application Scenarios

- **Sensor Interface Adaptation**: A sensor delivers measured values as `INT` (e.g., temperature or pressure) via an AI adapter; The subsequent controller expects `UINT` via an AUI adapter.
- **System Integration**: In heterogeneous systems using different adapter types, `AI_TO_AUI` enables seamless connection of components.
- **Data Preparation**: Before further processing in a PLC that only supports unsigned values (e.g., for counters or scaling).

Comparison with Similar Function Blocks

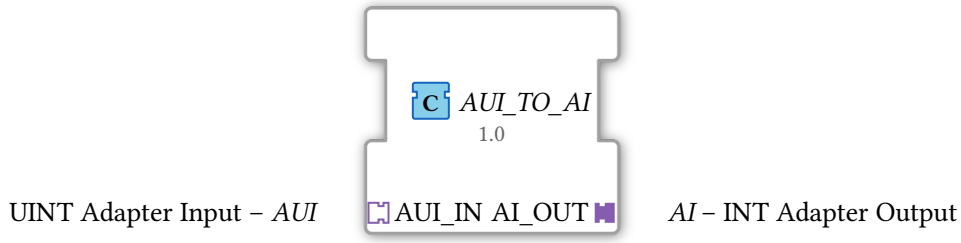
- **`AI_TO_AUI` vs. Direct Conversion with `F_INT_TO_UINT`**: The composite function block encapsulates the adapter interfaces and simplifies the application for the developer – individual function blocks with signal connections do not need to be wired.

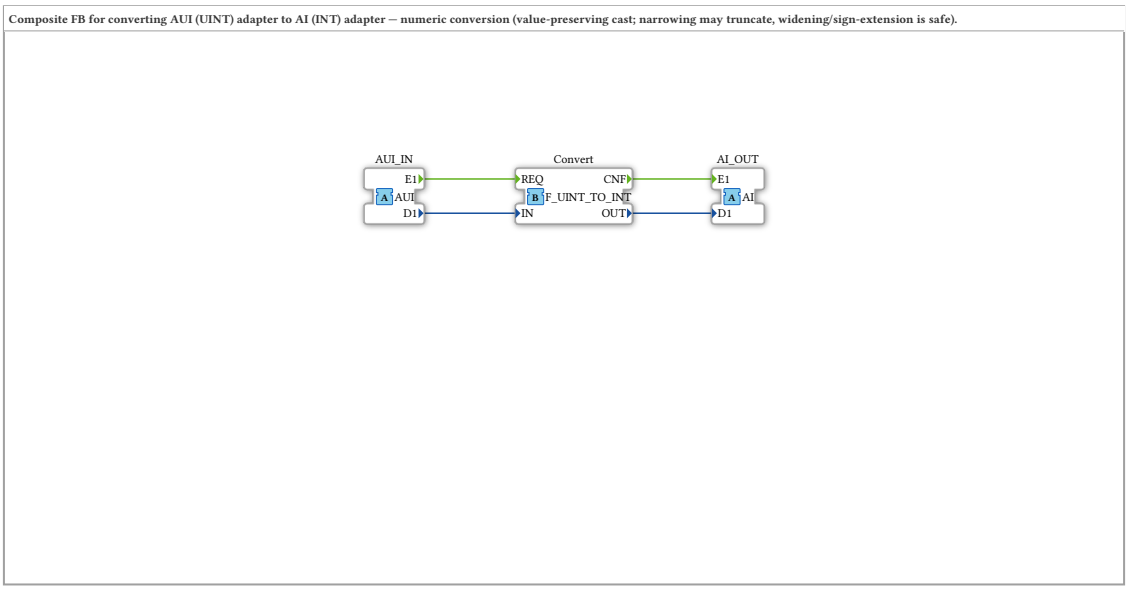
- **Similar Adapter Converters**: The library may contain other modules such as `AUI_TO_AI` (reverse conversion) or converters for other data types (e.g., `REAL_TO_INT`). `AI_TO_AUI` is specifically designed for combining INT and UINT adapters.

Conclusion

The module `AI_TO_AUI` offers a compact, reusable solution for converting an INT adapter data stream to a UINT adapter data stream. It reduces the complexity of the interconnection and ensures that the data conversion is correct and event-driven. Thanks to its adapter interfaces, it integrates seamlessly into modular, unidirectional architectures.

AUI_TO_AI





• – ○ • – ○ • – ○ •

Introduction

Composite function block (FB) for converting an AUI (UINT) adapter into an AI (INT) adapter. It encapsulates the conversion using the function block `F_UINT_TO_INT` and provides a standardized adapter interface.

Interface Structure

Event Inputs

No direct event inputs at the FB level. The event is received via the **AUI_IN socket** (event `E1`).

Event Outputs

No direct event outputs at the FB level. The event is output via the **AI_OUT plug** (event `E1`).

Data Inputs

No direct data inputs at the FB level. The UINT value is provided via the **Socket AUI_IN** (data point `D1`).

Data Outputs

No direct data outputs at the FB level. The INT value is output via the **Plug AI_OUT** (data point `D1`).

Adapters

- **Socket AUI_IN** (Type: `adapter::types::unidirectional::AUI`) – Receives UINT data and an event.
- **Plug AI_OUT** (Type: `adapter::types::unidirectional::AI`) – Sends INT data and an event.

Functionality

This function block connects the AUI adapter socket to a conversion function block of type `F_UINT_TO_INT` via an FBNetwork:

1. The event `E1` of the AUI_IN adapter triggers the conversion function block at the **REQ** input.
2. The conversion function block reads the UINT value from the data point `AUI_IN.D1` and converts it to INT.
3. After the conversion, the function block sends an acknowledgment event via **CNF**, which triggers the event `E1` of the AI_OUT adapter.
4. The event `E1` of the AI_OUT adapter.
5. The conversion function block reads the UINT value from the data point `AUI_IN.D1` and converts it to INT. 4. The converted INT value is passed to the AI adapter via the data point `AI_OUT.D1`.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- Composite function block without its own state logic (no ECC).
- Uses the standard function block `iec61131::conversion::F_UINT_TO_INT` for type conversion.
- Contains license information according to the Eclipse Public License 2.0.

State Overview

The function block does not have an internal state machine. Control is achieved solely through event chaining in the FBNetwork.

Application Scenarios

- Integration of a component that provides an AUI adapter (UINT) into a system that expects an AI adapter (INT).
- Encapsulation of the conversion to facilitate reuse and maintain a clear network structure.

Comparison with Similar Function Blocks

- **AUI_TO_AI** converts UINT to INT.
- An analogous function block **AI_TO_AUI** would provide the reverse conversion (INT to UINT).
- Unlike direct type conversion function blocks (such as `F_UINT_TO_INT`), this function block operates at the adapter level and integrates seamlessly into IEC 61499 adapter concepts.

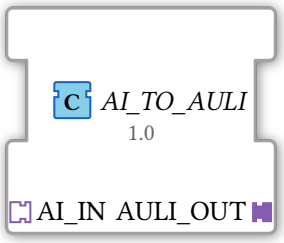
Conclusion

The **AUI_TO_AI** function block offers a simple and encapsulated solution for converting AUI adapters to AI adapters. It facilitates interoperability between components with different data types and contributes to a clean, modular system architecture.

Numeric Index (unidirectional)

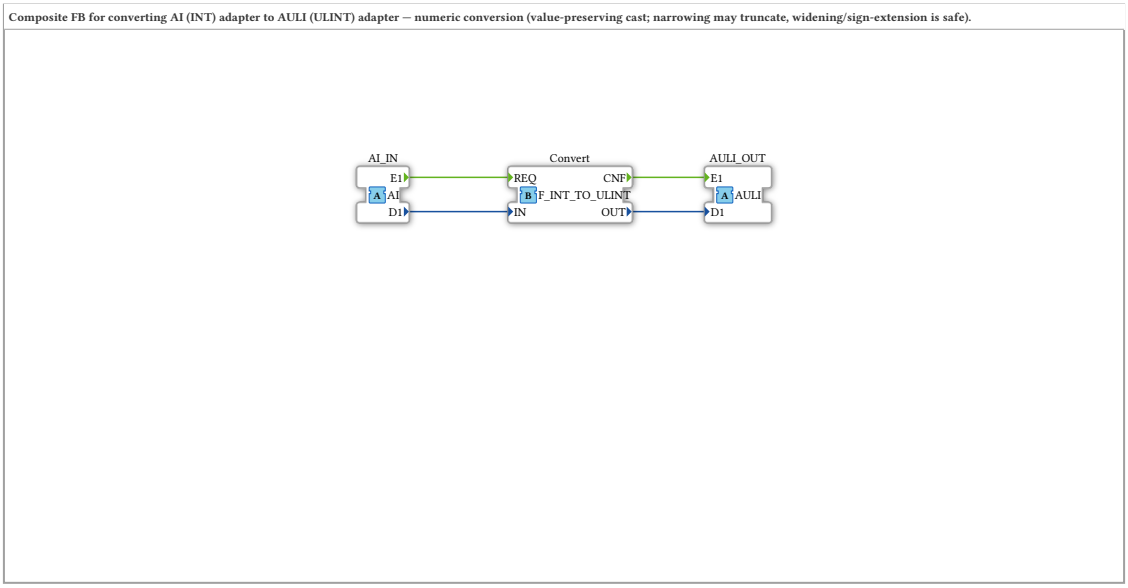
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AI_TO_AULI



INT Adapter Input – *AI*

AULI – ULINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AI_TO_AULI** serves as a composite block for converting a unidirectional adapter of type `AI` (based on the data type `INT`) into a unidirectional adapter of type `AULI` (based on `ULINT`). It enables the seamless integration of components that use different integer data types without requiring the user to implement the conversion logic

---

Interface Structure

The block does not have traditional event or data inputs/outputs, but only adapter interfaces (socket & plug) through which both events and data are transmitted.

Event Inputs

The event input is provided via the socket adapter `AI_IN`.

Name	Adapter	Comment
`E1`	`AI_IN`	Event that triggers the conversion

Event Outputs

The event output is provided via the plug adapter `AULI_OUT`.

Name	Adapter	Comment
`E1`	`AULI_OUT`	Event after successful conversion

Data Inputs

Data input is provided via the socket adapter `AI_IN`.

Name	Adapter	Type	Comment
D1	AI_IN	INT	Input data (16-bit integer)

Data Outputs

Data output is provided via the plug adapter AULI_OUT.

Name	Adapter	Type	Comment
D1	AULI_OUT	ULINT	Output data (64-bit unsigned integer)

Adapter

- **Socket** AI_IN : Unidirectional adapter with the interfaces E1 (event) and D1 (INT).
- **Plug** AULI_OUT: Unidirectional adapter with the interfaces E1 (event) and D1 (ULINT).

Functionality

This module performs a simple type conversion from INT to ULINT. Internally, the IEC 61131 module F_INT_TO_ULINT is used. The process is event-driven:

1. An incoming event at socket AI_IN.E1 triggers the input REQ of the conversion module.
2. The data value AI_IN.D1 (INT) is passed to the input IN of the conversion block.
3. The conversion is performed, and the result (ULINT) is provided at the output OUT.
4. After the conversion is complete, an event is generated at the output CNF of the conversion block, which is forwarded to the plug AULI_OUT.E1.
5. Simultaneously, the converted value is sent to the plug AULI_OUT.D1.

This ensures a complete, synchronized transfer of the event and data.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-Based Communication:** The module encapsulates the conversion logic in a composite, so the connected components only need to know the respective adapter interfaces.
- **Unidirectional Data Transfer:** Both AI and AULI are designed as unidirectional adapters; reverse data transfer is not supported.
- **Type Conversion:** The internal module F_INT_TO_ULINT performs an extended conversion from 16-bit integer (INT) to 64-bit unsigned integer (ULINT). Negative INT values are converted into large positive ULINT values using two's complement (e.g., -1 ⟶ 2^64-1).
- **No Eigenstates:** The component does not have its own state machine but forwards events and data directly.

State Overview

Since the component is implemented as a pure composite without its own stateful logic, there is no explicit state machine. Its behavior is determined solely by the coupling of the internal components. The functionality is purely combinatorial and event-triggered.

Application Scenarios

- **Adapter Converter:** Integrating sensors or actuators provided as AI adapters (INT) into a system that expects AULI adapters (ULINT).
- **System Integration:** Use in PLC or automation architectures where different data types need to be standardized.
- **Protocol Adaptation:** Converting 16-bit counter values to 64-bit counter values for extended range.

–

Comparison with Similar Function Blocks

Function Block	Source Type	Target Type	Adapter-Based	Custom States
`F_INT_TO_ULINT`	`INT`	`ULINT`	No (pure function block)	No
`AI_TO_AULI`	`AI`	`AULI`	Yes (adapter)	No
`ANY_TO_ANY`	variable	variable	possible	depending on implementation

While the basic function block `F_INT_TO_ULINT` only performs data conversion, `AI_TO_AULI` encapsulates this along with event control in an adapter-compatible interface. This increases reusability in adapter-based systems.

… —

Conclusion

The function block `AI_TO_AULI` offers a clean, adapter-based solution for converting `INT` to `ULINT` data. It reduces manual effort and the potential for errors during type conversion and integrates seamlessly into existing 4diac projects. Thanks to its simple, event-triggered structure, it is particularly well-suited for unidirectional data flows.

–

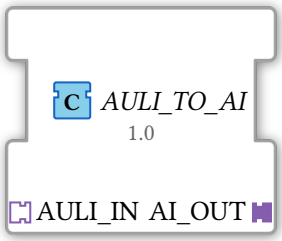
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

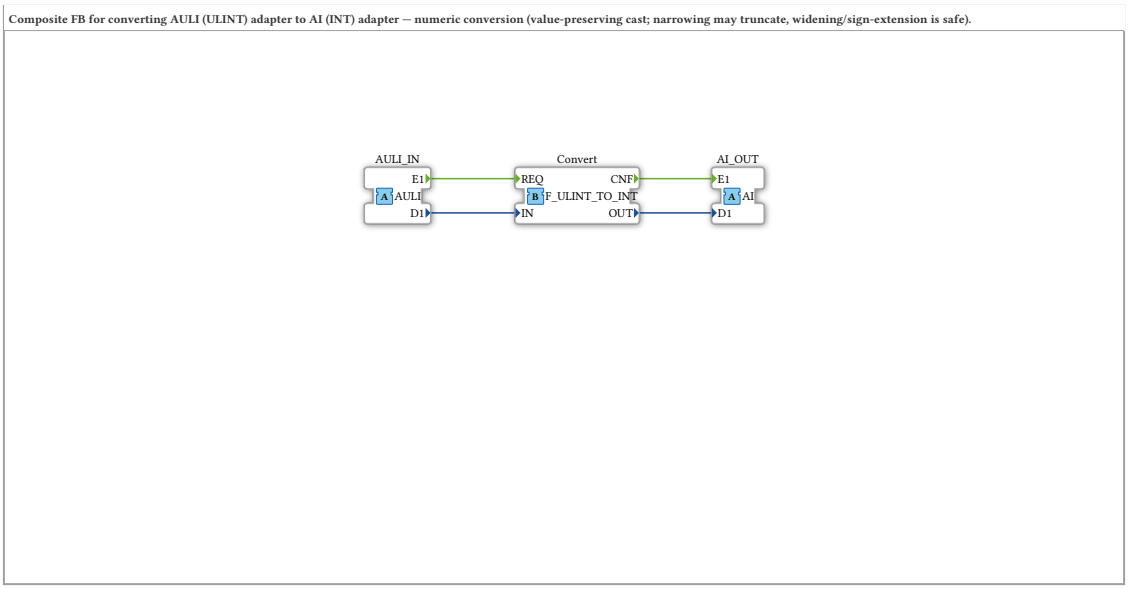
]

AULI_TO_AI

ULINT Adapter Input – *AULI*



AI – INT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_TO_AI** is a composite function block (FB) that converts an incoming adapter of type **AULI** (ULINT-based) into an outgoing adapter of type **AI** (INT-based). It serves as an interface converter between components that use different data types for analog values.

Interface Structure

Event Inputs

The function block itself has no direct event inputs. Event control is handled via the connected adapters.

Event Outputs

There are also no dedicated event outputs at this level; output events are passed on via the AI_OUT adapter.

Data Inputs

- No direct data inputs at the block level. The data is transferred to the block via socket `AULI_IN`.

Data Outputs

- No direct data outputs at the block level. The converted data is output via plug `AI_OUT`.

Adapters

Name	Direction	Type	Description
`AULI_IN`	Socket	`adapter::types::unidirectional::AULI`	ULINT input adapter (analog value)
`AI_OUT`	Plug	`adapter::types::unidirectional::AI`	INT Output Adapter (Converted Value)

Functionality

The function block internally uses the predefined function block `F_ULINT_TO_INT` from the IEC 61131 library. Its functionality is as follows:

1. An event from socket `AULI_IN.E1` triggers the conversion of the data value `AULI_IN.D1` (ULINT) via the internal function block `Convert`.
2. The function block `Convert` performs the type conversion from `ULINT` to `INT`.
3. After successful conversion, the output event `Convert.CNF` is triggered, which activates the event `AI_OUT.E1`.
4. The converted data value (`Convert.OUT`) is forwarded to `AI_OUT.D1`.

This enables seamless, event-driven conversion from AULI to AI adapters.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composition:** The function block is implemented as a composite function block, meaning it encapsulates the logic in a clear and concise block and uses a standardized conversion block.
- **Unidirectional Adapters:** Both `AULI_IN` and `AI_OUT` are unidirectional adapters that support data flow in only one direction.
- **Event-driven:** The conversion process only starts upon an incoming event, enabling efficient processing.
- **License notice:** This function block is subject to the Eclipse Public License 2.0.

State overview

Since the function block operates purely event-driven and does not store any internal states, there is no explicit state machine. It behaves like a functional transformation: Each input event triggers a conversion and an output event.

Internal flow (simplified):

- **Idle:** Waiting for an event from `AULI_IN`.

- **Processing:** Conversion is running (instant, as it is a synchronous function block).
- **Done:** Output event is sent, return to Idle.

Application scenarios

- **System integration:** Connecting components that use different data types for analog signals (e.g., a sensor provides ULINT, an actuator expects INT).
- **Adapter Cascades:** Extending protocol conversions in automation projects.
- **Data Type Conversion:** When a fieldbus or gateway module only provides ULINT values, but the control system requires INT.

Comparison with Similar Function Blocks

Function Block	Input Type	Output Type	Description
`AULI_TO_AI`	AULI (ULINT)	AI (INT)	Converts entire adapter interfaces.
`F_ULINT_TO_INT`	ULINT	INT	Data conversion only, no adapters.
`AI_TO_AULI` (if applicable)	AI (INT)	AULI (ULINT)	Reverse direction (not available).

The key difference compared to pure data conversion blocks lies in the adapter input/output, which enables complete interface conversion.

Conclusion

The `AULI_TO_AI` block offers an elegant and standardized way to perform adapter-specific type conversions between ULINT and INT-based analog interfaces. Encapsulating the conversion logic in a composite function block increases reusability and clarity. It is particularly well-suited for modular automation architectures where different protocols or data types interact.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AI_TO_AUS



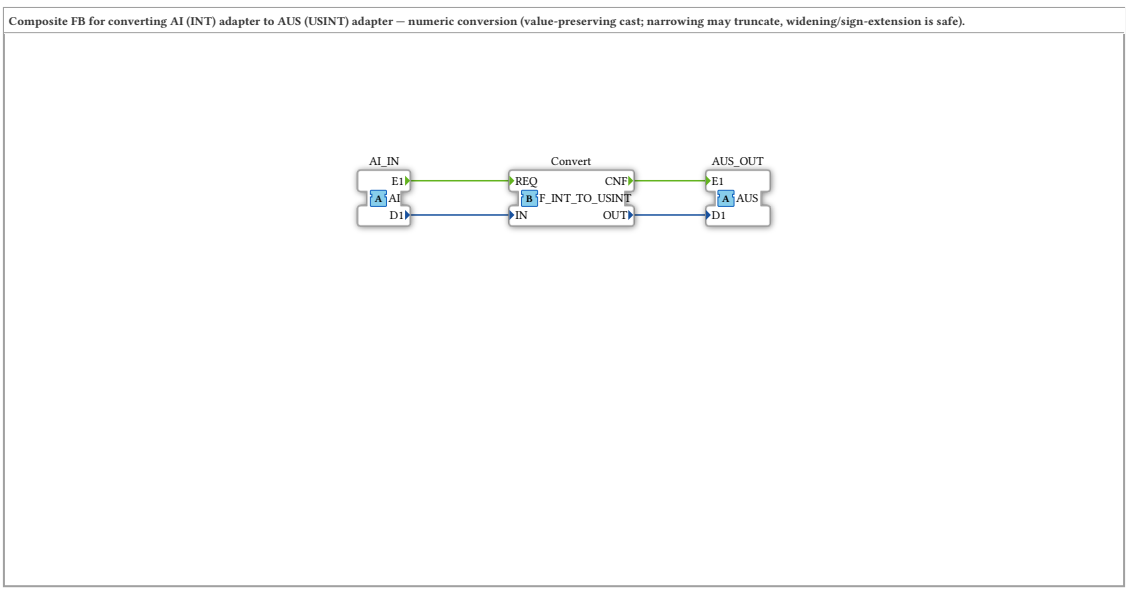


Image of the function block – not included in the XML, therefore no automatically generated image is available.

• – ◦ • – ◦ • – ◦ •

Introduction

The **AI_TO_AUS** function block is a composite function block that converts an **AI adapter** (data stream of type `INT`) into an **AUS adapter** (data stream of type `USINT`). It serves as a bridge between components that use different data types for analog input and output signals. The implementation uses the internal function block `F_INT_TO_USINT` from the `iec61131::conversion` library.

Interface Structure

The function block does not have its own separate event or data ports, but communicates exclusively via adapters.

Event Inputs

- No direct event inputs. Event control is handled indirectly via the `AI_IN` adapter (socket) – this is where the event `E1` is received.

Event Outputs

- No direct event outputs. The converted event is output as `E1` via the `AUS_OUT` adapter (plug).

Data Inputs

- No direct data inputs. The data value to be converted (`INT`) is obtained as `D1` via the `AI_IN` adapter (socket).

Data Outputs

- No direct data outputs. The converted value (`USINT`) is provided as `D1` via the adapter `AUS_OUT` (Plug).

Adapters

- **`AI_IN`** (Socket)

Type: `adapter::types::unidirectional::AI`
Description: Accepts a `INT` value and an associated event.

- **`AUS_OUT`** (Plug)

Type: `adapter::types::unidirectional::AUS`
Description: Outputs the converted `USINT` value along with an event.

Functionality

1. An incoming event at the adapter `AI_IN.E1` triggers the internal function block `Convert` (`F_INT_TO_USINT`) via its event input `REQ`.
2. Simultaneously, the data value `AI_IN.D1` (type `INT`) is passed to the input `IN` of the conversion block.
3. The block `F_INT_TO_USINT` performs the type conversion from `INT` to `USINT`.
4. After the conversion is complete, `Convert` sends an acknowledgment event (`CNF`) to the output adapter, and the converted value (`OUT`) is forwarded to `AUS_OUT.D1`.
5. The event `AUS_OUT.E1` signals to the subsequent component that a new data value is available at the adapter.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Library Used:** The function block uses the IEC 61131 standard function block `F_INT_TO_USINT`, which implements a typical conversion with possible range limitation or overflow behavior.
- **Adapter Encapsulation:** The composite block abstracts the conversion at the adapter level, so only the two matching adapters need to be connected in the network configuration.
- **Data Range Conflict:** `INT` encompasses values from −32,768 to +32,767, while `USINT` can only represent 0…255. For input values outside the USINT range, the behavior depends on `F_INT_TO_USINT` (e.g.,

saturation to 0 or 255, or modular overflow). The exact mapping rule must be verified in the specific target system.

State Overview

Since the block consists exclusively of sequential data processing (repeated conversion with each event), it does not have its own state machines. The states of the internal block `F_INT_TO_USINT` are not visible to the function; the composite block operates event-driven and without internal state storage.

Application Scenarios

- **Sensor-Actuator Coupling:** A sensor provides measured values as `INT` (e.g., temperature −10…50 °C scaled to integers), while a downstream actuator expects a control variable as `USINT` (e.g., 0…100%). The block handles the necessary type conversion.
- **Adapter Compatibility:** In a 4diac application, two modules are defined via different adapters (`AI` and `AUS`). The Composite Function Block (FB) enables direct connection without additional conversion logic in the main network.

Comparison with Similar Function Blocks

- **Direct Converters (`F_INT_TO_USINT`):** Provides pure type conversion at the data level but requires separate event and data connections. `AI_TO_AUS` extends this with adapter connectivity, thus simplifying network design.
- **Other Adapter Converters (`AI_TO_...`, `..._TO_AUS`):** Depending on the required adapter types, corresponding Composite Function Blocks are conceivable. `AI_TO_AUS` covers the specific combination `INT` ⟶ `USINT`.

Conclusion

The **AI_TO_AUS** function block is a compact composite module for easy adapter conversion from `INT` to `USINT`. It encapsulates the IEC 61131 converter and provides data and event flows via standardized adapters. This significantly simplifies the integration of components with different analog interfaces into 4diac applications. When using this module, potential data type overflow should be considered.

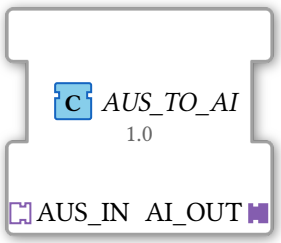
–

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

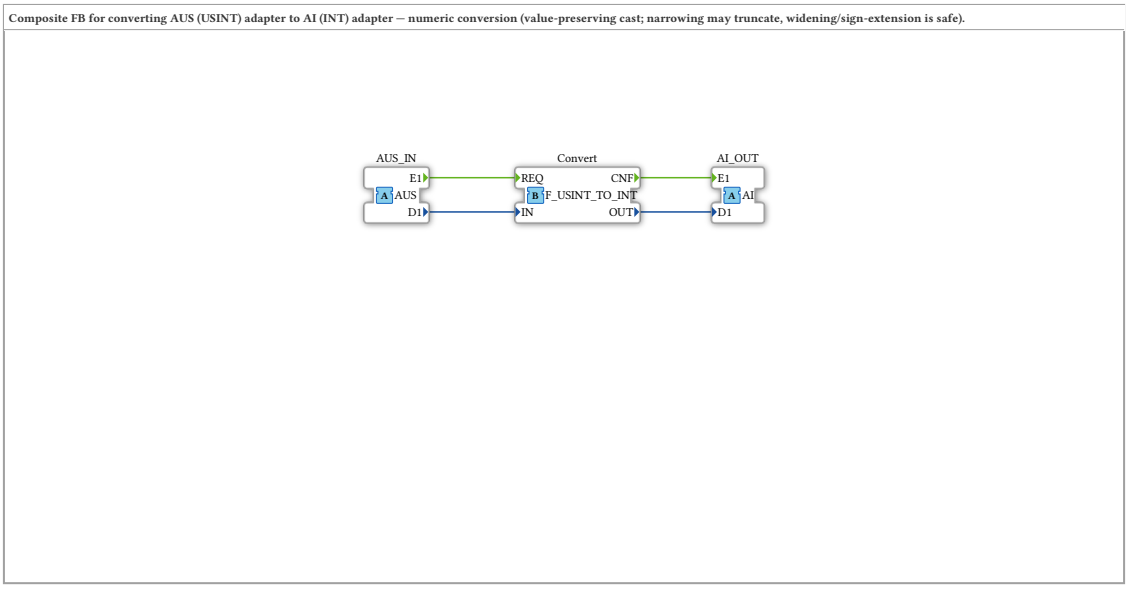
]

AUS_TO_AI



USINT Adapter Input – *AUS*

AI – INT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block `AUS_TO_AI` is a composite block that converts a `AUS` adapter (USINT-based) into a `AI` adapter (INT-based). It serves as a unidirectional interface between system components that use different data types for analog inputs. The block encapsulates the conversion logic and facilitates the integration of components based on `USINT` adapters into environments that expect `INT` adapters.

Interface Structure

The function block has **no direct event or data inputs or outputs**. All signal communication is handled via the two built-in adapters:

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Direction	Type	Description
`AUS_IN`	Socket	`adapter::types::unidirectional::AUS`	USINT-based adapter input (source of the value to be converted)
`AI_OUT`	Plug	`adapter::types::unidirectional::AI`	INT-based adapter output (destination of the converted value)

The socket `AUS_IN` receives an event signal and a USINT data value. After conversion, the plug `AI_OUT` outputs an event signal and the corresponding INT value.

Functionality

The function block operates purely combinatorially based on an internal instance of the IEC 61131 function block `F_USINT_TO_INT`. Sequence control is implemented via the adapters' event connections:

1. **Event E1 on the OFF_IN adapter** is routed to the **REQ input** of the conversion block `Convert`.
2. The conversion block converts the USINT value at its **IN** input into an INT value and signals completion with the **CNF** event.
3. The **CNF** event is passed to the **E1 input of the AI_OUT adapter**, and simultaneously, the converted INT value is output to the **D1 data output** of the AI_OUT adapter.

This creates a causal chain: An incoming event at the AUS_IN adapter triggers the conversion and results in the output of a corresponding event and the converted data value at the AI_OUT adapter.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Pure Adapter Coupling** – The block does not have its own input/output variables but only defines adapter interfaces (socket/plug). This follows the concept of adapter-based communication in IEC 61499.
- **Unidirectional Conversion** – The direction is fixed: from USINT to INT. Reverse conversion is not supported.
- **Use of a Standard IEC 61131 Function Block** – The actual type conversion is performed by the established function block `F_USINT_TO_INT`, which is already available in many runtime environments.
- **No Internal State** – The composite function block is stateless; the conversion is triggered anew with each incoming event.

State Overview

Since the function block does not have its own state machine, only the implicit states of the embedded `F_USINT_TO_INT` exist. In practice, the function block behaves like a **combinatorial function**: After an event is received, the output signal is available for the next execution cycle. No defined error state exists – the conversion is performed according to the IEC component specification.

Application Scenarios

- **Migration from USINT-based hardware to INT-based controllers** – A system that originally operates with 8-bit values (USINT) can be connected via this adapter to components that expect 16-bit analog values (INT).
- **Coupling of different subsystems** – In a modular plant, a sensor adapter uses USINT, while the higher-level control unit provides INT interfaces.
- **Prototypes and test environments** – Rapid integration of components with different data types without separate conversion logic at the application level.

Comparison with Similar Function Blocks

- **Direct Conversion Function Blocks** – Function blocks like `F_USINT_TO_INT` or `F_INT_TO_USINT` offer pure type conversion but require manual wiring of events and data. `AUS_TO_AI` encapsulates this in an adapter-to-adapter concept, simplifying integration into adapter-based architectures.
- **Other Adapter Converters** – Symmetrical function blocks like `AI_TO_AUS` (INT to USINT) or bidirectional variants are conceivable. `AUS_TO_AI` is specifically designed for the unidirectional conversion from `AUS` to `AI`.

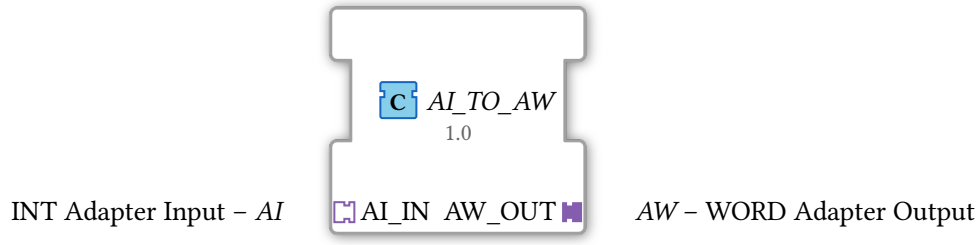
Conclusion

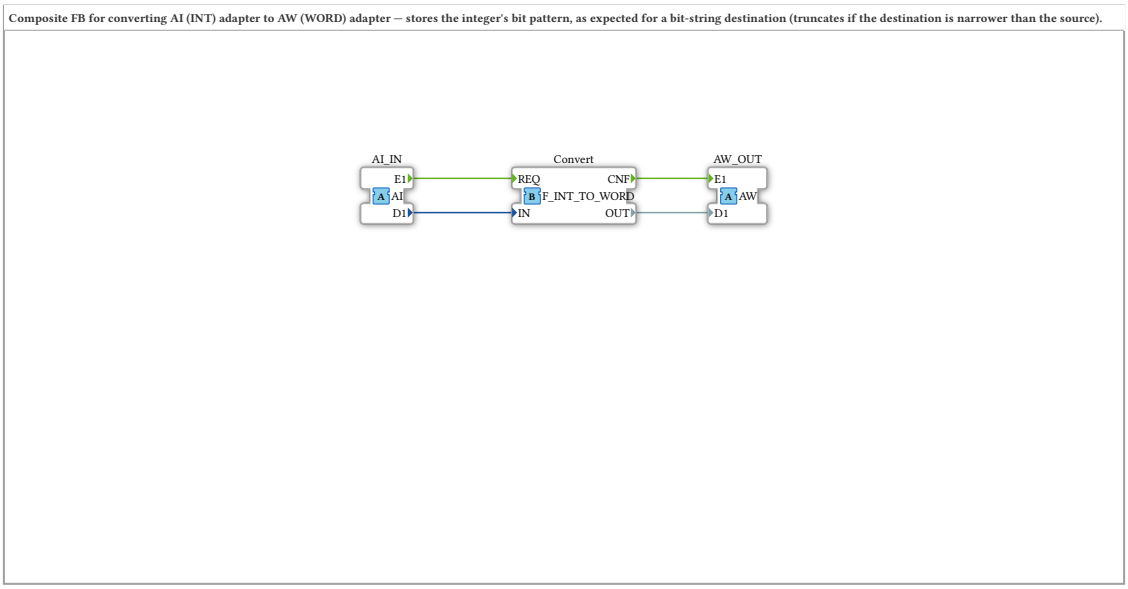
The `AUS_TO_AI` composite module offers a clean, adapter-based solution for converting a USINT value to an INT value. By encapsulating the standard `F_USINT_TO_INT` module and providing clear event/data flow control, it is ideally suited for integrating heterogeneous components into IEC 61499 systems. The module requires minimal maintenance, is stateless, and adheres to established adapter interface conventions.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AI_TO_AW





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_TO_AW** is a composite module for converting an INT adapter (AI) to a WORD adapter (AW). It is used to convert the interface of an AI adapter (type INT) to an AW adapter (type WORD) so that data and events can be seamlessly transferred between different adapter types.

Interface Structure

Event Inputs

- **AI_IN.E1** (via Socket AI_IN): Event input that triggers the conversion process.

Event Outputs

- **AW_OUT.E1** (via Plug AW_OUT): Event output that is activated after successful conversion.

Data Inputs

- **AI_IN.D1** (via Socket AI_IN): Data input of type **INT**, providing the value to be converted.

Data Outputs

- **AW_OUT.D1** (via Plug AW_OUT): Data output of type **WORD**, providing the converted result.

Adapters

- **AI_IN** (Socket): Expects an adapter of type `adapter::types::unidirectional::AI` (INT-based).
- **AW_OUT** (Plug): Provides an adapter of type `adapter::types::unidirectional::AW` (WORD-based).

Functionality

The function block operates as an encapsulation of a simple type conversion function block (`F_INT_TO_WORD`) within a composite structure. The process is as follows:

1. An event at input **AI_IN.E1** is forwarded to the internal function block `F_INT_TO_WORD`.

2. Simultaneously, the value of **AI_IN.D1** (INT) is passed to input `IN` of the conversion block.
3. The internal function block performs the conversion from INT to WORD and places the result at its output `OUT`.
4. After the conversion is complete, the event **AW_OUT.E1** is sent, and the converted value (WORD) is output to **AW_OUT.D1**.

Thus, the entire adapter conversion is controlled by a single event, and the data is converted automatically.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Block**: The logic is implemented by an internal sub-block, allowing for a clear separation of conversion logic and interface adapters.
- **Reusability**: The block is based on the standard block `iec61131::conversion::F_INT_TO_WORD` and can therefore be used in any IEC 61499-compliant environment.
- **License**: The block is released under the Eclipse Public License 2.0 (EPL-2.0).
- **Naming Scheme**: The block is located in the package `adapter::conversion::unidirectional`, indicating its function as a unidirectional adapter conversion.

State Overview

Since this is a composite block without its own state machine, the state behavior is entirely determined by the internal `F_INT_TO_WORD` block. There are essentially two states:

- **Idle**: Waiting for an event at input **AI_IN.E1**.
- **Converting**: After receiving the event, the conversion is performed. No delays or waiting times are implemented – the output occurs immediately in the same cycle.

Application Scenarios

- **System Integration**: When a subsystem or component provides data via an INT adapter (AI), but the target component expects a WORD adapter (AW).
- **Protocol Adaptation**: In plant control systems where sensor data is delivered as INT, but the communication interface is WORD-based.
- **Type Conversion in Adapter Chains**: The function block can be used as part of an adapter chain to connect different adapter types without requiring manual conversion.

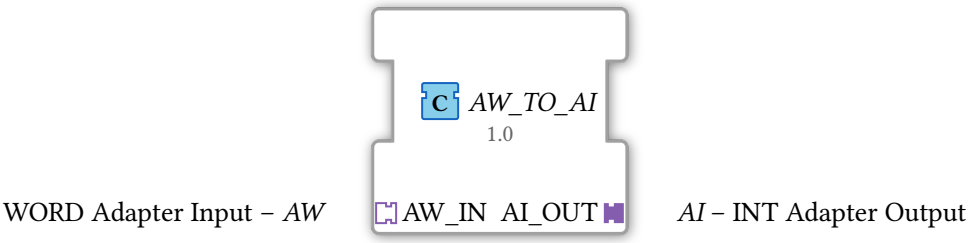
Comparison with Similar Function Blocks

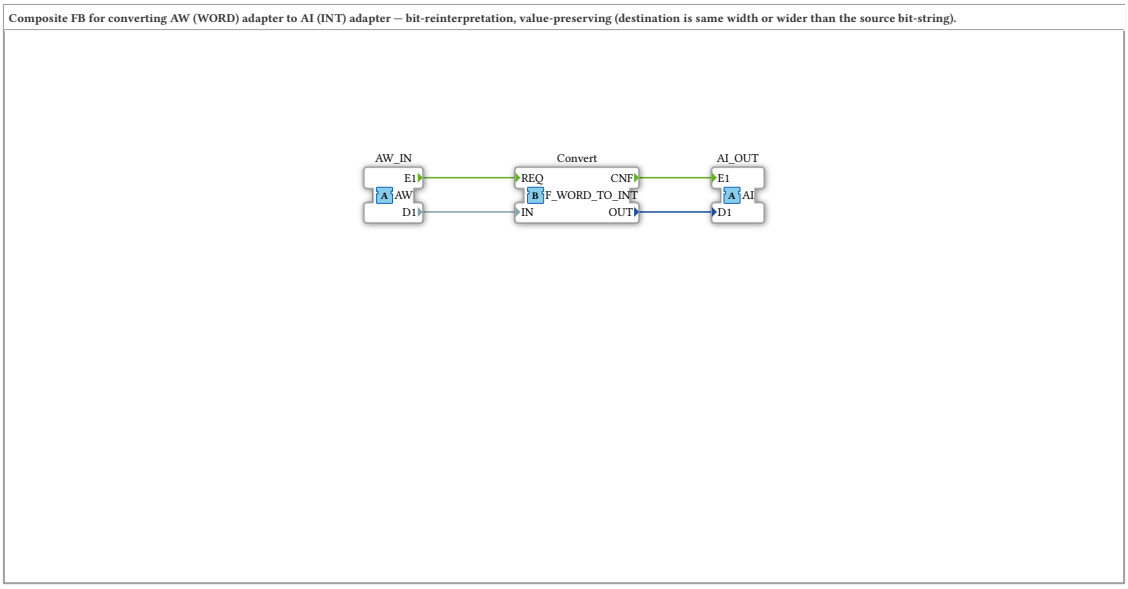
Other adapter conversion blocks exist, such as `WORD_TO_INT` or `REAL_TO_DINT` – however, `AI_TO_AW` focuses specifically on the adapter conversion from INT (AI) to WORD (AW). Unlike simple type conversion blocks (e.g., `F_INT_TO_WORD`), this block integrates the adapter interfaces directly, eliminating the need for the developer to create separate adapter instantiations.

Conclusion

The **AI_TO_AW** function block offers a simple and robust solution for converting an INT adapter to a WORD adapter. Its composite structure keeps the logic manageable and extensible. The component is particularly suitable for scenarios in which different adapter types need to be harmonized in an IEC 61499 application, and significantly reduces the implementation effort for such conversions.

AW_TO_AI





• — ○ • — ○ • — ○ •

Introduction

The **AW_TO_AI** function block is a composite function block that converts a unidirectional **WORD adapter (AW)** to a unidirectional **INT adapter (AI)**. It encapsulates the type conversion from 16-bit word values to integer values, thus enabling the seamless integration of components that use different adapter types.

Interface Structure

Event Inputs

The following events are received via the connected AW socket:

Name	Description
E1 (AW_IN.E1)	Trigger for converting the incoming WORD value

Event Outputs

The following events are sent via the AI plug:

Name	Description
E1 (AI_OUT.E1)	Confirms successful conversion and provides the INT value

Data Inputs

Input via the AW socket:

Name	Data Type	Description
D1 (AW_IN.D1)	WORD	The 16-bit word value to be converted

Data Outputs

Output via the AI plug:

Name	Data Type	Description
D1 (AI_OUT.D1)	INT	The converted integer value (16-bit signed)

Adapter

Direction	Adapter	Type	Description

Socket (Input)	AW_IN	adapter::types::unidirectional::AW	Receives WORD values and associated events
Plug (Output)	AI_OUT	adapter::types::unidirectional::AI	Sends INT values and associated events

Functionality

The function block contains an instance of the standard conversion block `F_WORD_TO_INT` from the IEC 61131 library. The internal network is wired as follows:

1. An incoming event **AW_IN.E1** triggers the input `REQ` of the conversion block.
2. Simultaneously, the data value **AW_IN.D1** is set to the input `IN` of the block.
3. The block converts the WORD value to an INT value and outputs it at its output `OUT`.
4. The acknowledgment event `CNF` generated by the block is forwarded directly to **AI_OUT.E1**.
5. The converted integer value (output `OUT`) is written to **AI_OUT.D1**.

All processing is strictly event-driven: Conversion only occurs when a new event is received at the input.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Pure Data Conversion:** No states are stored, nor is any additional logic executed.
- **Standard Library:** The `F_WORD_TO_INT` used is an established conversion module and ensures a correct mapping of the WORD value (0…65535) to the INT value (-32768…32767).
- **Unidirectional:** Both the input and output adapters are unidirectional, meaning they transmit event and data flows in only one direction (input → output).
- **No data loss due to type conversion:** The bit width remains identical (16 bits); only the interpretation as unsigned (WORD) or signed (INT) changes.

State Overview

The FB is a **Composite FB** and does not have its own state machine (ECC). Its behavior is entirely determined by the wiring of the internal block. Only the following implicit states exist:

- **Idle:** Waiting for an event at the input.
- **Conversion active:** From the receipt of `AW_IN.E1` until the arrival of `Convert.CNF`.

Application Scenarios

- **Adapter conversion in heterogeneous systems:** A control module operates with INT values, but a connected sensor delivers WORD data via an AW adapter.
- **Library Integration:** Existing function blocks that only accept an AI adapter should be connected to components that provide an AW adapter.
- **Prototype Systems:** Rapid coupling of different function blocks during development without changing the underlying adapter definitions.

Comparison with Similar Function Blocks

- **AW_TO_AI** converts **WORD → INT** (unidirectionally).
- Comparable converters from the same library (`iec61131::conversion`) cover other type conversions, e.g., `F_INT_TO_WORD` or `F_WORD_TO_DINT`. These are available as traditional function blocks without adapters.
- If bidirectional conversion or other adapter types are required, similar composite function blocks with corresponding conversion blocks are conceivable (e.g., `AI_TO_AW`).

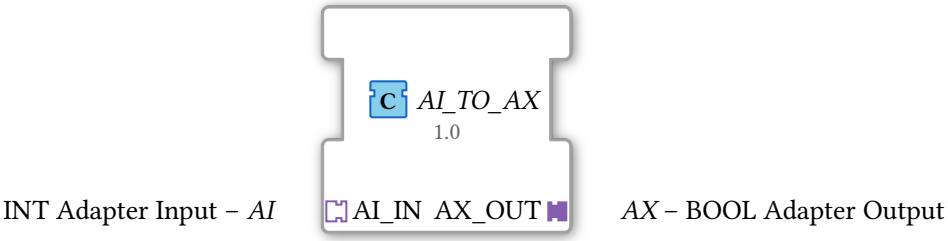
Conclusion

The composite function block **AW_TO_AI** provides a simple and reliable way to convert a unidirectional WORD adapter into an INT adapter. By using the IEC 61131 standard function block, the conversion remains transparent and efficient. The function block is particularly useful when different adapter types need to be harmonized within a system without modifying the original components.

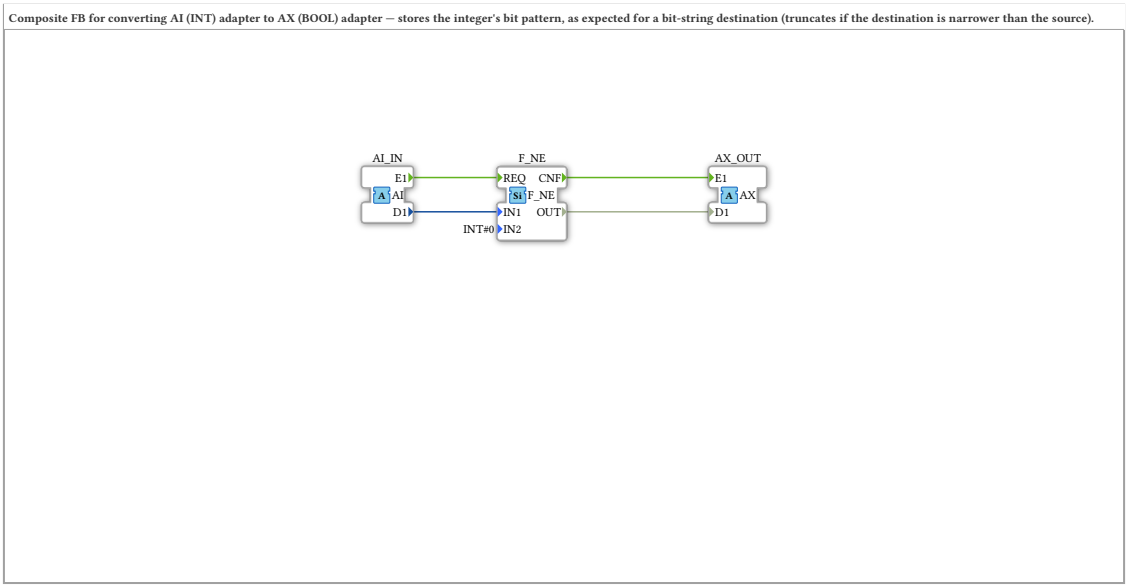
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AI_TO_AX



INT Adapter Input – *AI* — *AX* – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `AI_TO_AX` is a composite function block (type adapter) that converts a unidirectional interface of an INT adapter (`AI`) into a corresponding BOOL interface of an adapter (`AX`). It is used to convert analog integer signals (e.g., from sensors) into binary Boolean values, which can be used, for example, for switching signals or alarm messages.

Interface Structure

The function block only has adapter connections (socket and plug) through which events and data flow. The following table summarizes the logical inputs/outputs resulting from the connected adapters.

Event Inputs

Label	Data Type	Description
`AI_IN.E1`	EVENT	Input event that triggers the conversion.

Event Outputs

Label	Data Type	Description
`AX_OUT.E1`	EVENT	Output event that is sent after the conversion is complete.

Data Inputs

Label	Data Type	Description

AI_IN.D1	INT	Analog integer input value (e.g., 0…32767).

Data Outputs

Label	Data Type	Description
AX_OUT.D1	BOOL	Boolean output value: TRUE if the input value is not equal to 0, otherwise FALSE.

Adapter

| Type | Name | Direction | | |—–|—–|—–| | AI (Input Adapter) | AI_IN | Socket | | AX (Output Adapter) | AX_OUT | Plug |

Functionality

The module internally uses the IEC 61131 comparison module F_NE (Not Equal). As soon as an event arrives at the event input AI_IN.E1, F_NE.REQ is triggered. This compares the INT value received via AI_IN.D1 with the constant value INT#0.

- If the input value is **not equal to 0**, F_NE.OUT returns the value TRUE.
- If the input value is **equal to 0**, F_NE.OUT returns the value FALSE.

The result is passed to the data output AX_OUT.D1, and simultaneously the event AX_OUT.E1 is triggered to notify downstream components.

```
If the input value is **equal to 0**, thenF_NE.OUTreturns the valueFALSE`.
```

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Block:** The AI_TO_AX consists of interconnected basic blocks, allowing it to be used as a reusable component in various projects.
- **Use of IEC 61131 Blocks:** The comparison logic is implemented using the standardized block F_NE, which supports portability and compliance with the standard.
- **No Independent State:** The block is stateless – each event pulse generates an immediate calculation without storing intermediate values.

State Overview

The block does not have an explicit finite state machine. The internal logic is purely combinatorial: Upon the arrival of an event, the data is processed immediately and the result is output.

Application Scenarios

- **Analog-to-Digital Conversion at the Signal Level:** An analog measured value (e.g., pressure sensor, temperature sensor) is converted into an "on/off" signal – e.g., activating a pump when a pressure threshold is exceeded.
- **Threshold Decision:** The function block can be used for simple non-zero detection, e.g., to signal the presence of a signal or an activated counter reading.
- **Adapter Coupling:** In systems that use different adapter interfaces, AI_TO_AX enables seamless connectivity between INT- and BOOL-based components.

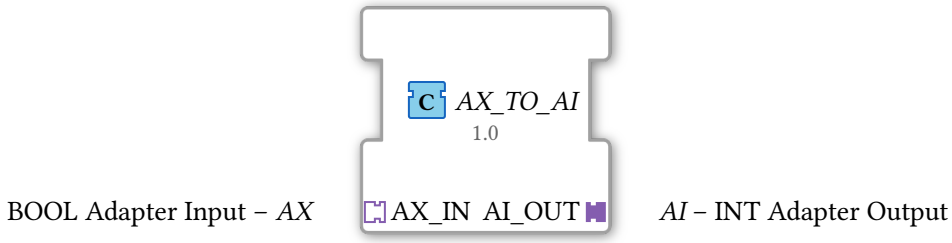
Comparison with Similar Function Blocks

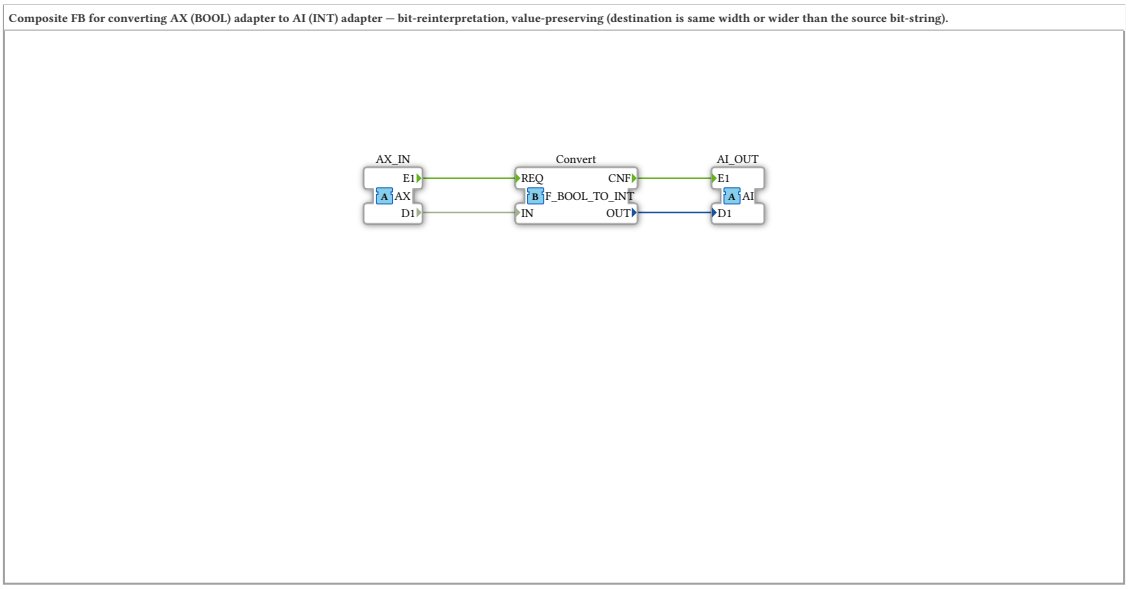
- **Direct Comparison Function Blocks (e.g., F_EQ, F_GT):** These provide Boolean results for arbitrary comparisons but require two inputs and are not designed as adapter converters.
- **Custom Adapter Wrappers:** Developers could create their own encapsulation; however, the AI_TO_AX offers a standardized, ready-to-use solution with reduced development effort.
- **Non-Event-Driven Type Converters:** Some converters operate continuously without event triggers; the AI_TO_AX, on the other hand, only responds to events and is therefore better integrated into event-driven architectures.

Conclusion

The `AI_TO_AX` is a convenient composite component that simplifies the conversion of INT adapter signals to BOOL adapter signals. It is ideal for applications where analog measurements need to be interpreted as binary states and, thanks to its event-driven processing, integrates seamlessly into IEC 61499-based automation systems.

AX_TO_AI





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_TO_AI** is a composite function block that converts a BOOL adapter (AX) into an INT adapter (AI). It serves as a bridge between interfaces that process Boolean signals (TRUE / FALSE) and those that expect integer values. The conversion is performed via an internal call to the function block `F_BOOL_TO_INT` from the IEC 61131 library.

Interface Structure

Event Inputs

No direct event inputs. The function block's event input is implemented via socket `AX_IN` (E1).

Event Outputs

No direct event outputs. The event output of the function block is implemented via the plug `AI_OUT` (E1).

Data Inputs

No direct data inputs. The data input of the function block is implemented via the socket `AX_IN` (D1) (BOOL).

Data Outputs

No direct data outputs. The data output of the function block is implemented via the plug `AI_OUT` (D1) (INT).

Adapters

Direction	Name	Type (Adapter)	Explanation
Socket	AX_IN	AX	Boolean Input Adapter (BOOL)
Plug	AI_OUT	AI	Integer Output Adapter (INT)

The adapter AX provides an event input E1 and a data input D1 (BOOL).
The adapter AI provides an event output E1 and a data output D1 (INT).

Functionality

This function block operates as a so-called *Composite FB*, meaning it internally uses another function block to implement its logic.

- A BOOL signal is received at socket AX_IN via the data input D1.
- An incoming event at AX_IN.E1 triggers the internal function block Convert (type: F_BOOL_TO_INT) via its input REQ.
- The function block F_BOOL_TO_INT converts the BOOL value at input IN into an INT value (0 for FALSE, 1 for TRUE).
- The result is provided at output OUT and forwarded to plug AI_OUT.D1.
- After the conversion is complete, the internal block sends an acknowledgment event (CNF), which is output via AI_OUT.E1.

Processing is strictly sequential: first, the Boolean input is processed, then the integer output is provided.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Adapter-based communication:** The block uses only adapter interfaces (socket and plug) and no traditional input/output ports. This enables loose coupling between components and easy reuse in different environments.
- **Unidirectional Data Exchange:** The adapters used, AX and AI, are defined as unidirectional types, meaning data flows only in one direction – from the socket to the plug.
- **External Library:** For conversion, the function block F_BOOL_TO_INT is imported from the library iec61131::conversion. This eliminates the need to implement custom conversion logic.
- **No Internal State Machine:** Because this is a composite function block, it does not have its own Execution Control Chart (ECC). The state logic is fully implemented by the internal function block.

State Overview

The function block does not have its own state machine. Execution is event-driven:

1. **Wait:** The function block waits for an event at socket AX_IN.
2. **Convert:** Upon receiving an event, the internal function block F_BOOL_TO_INT is executed, and the result is passed to plug AI_OUT.
3. **Completed:** An acknowledgment event is sent via AI_OUT.E1; the function block returns to the wait state.

This behavior is implicitly controlled by the event cabling in the network.

Application Scenarios

- **Signal Conversion in Automation Technology:** If a sensor or controller delivers Boolean values (e.g., “door open/closed”), but a downstream module expects integer values (0/1), this function block can handle the conversion.
- **Adapter Integration:** It serves as a converter between different adapter types to connect components from different manufacturers or libraries.
- **Protocol Adaptation:** In heterogeneous systems, Boolean signals can be converted into standardized integer formats using this function block.

Comparison with Similar Function Blocks

Function Block	Function	Difference from AX_TO_AI
`DI_TO_AI`	Converts digital input to INT adapter	Requires separate input/output ports; no adapter interface
`AX_TO_DINT`	BOOL adapter ⟶ DINT adapter	Returns a 32-bit integer; 16-bit INT is used here
`F_BOOL_TO_INT`	Standard conversion (BOOL⟶INT)	No adapter wrapper; available only as an internal block

Unlike these alternatives, AX_TO_AI offers an addressable, event-driven adapter interface and facilitates modular interconnection in IEC 61499 projects.

Conclusion

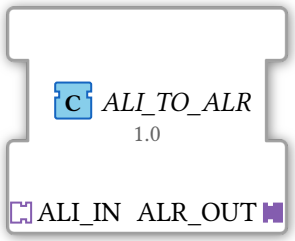
The `AX_TO_AI` function block provides a simple yet effective solution for converting Boolean adapter signals to integer adapter signals. It encapsulates the logic in a reusable composite block, utilizes standard libraries, and enables clean, adapter-based communication. This block is particularly well-suited for modular automation solutions where different interface types need to be coordinated.

Numeric Index (unidirectional)

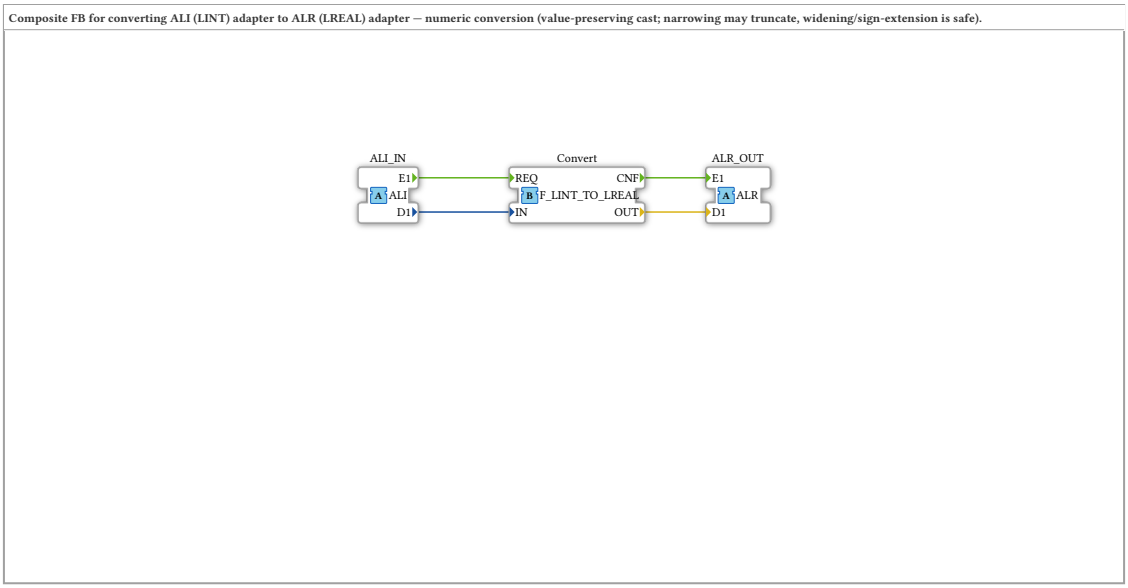
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALI_TO_ALR

LINT Adapter Input – *ALI*



ALR – LREAL Adapter Output



(Image not available)

• – ○ • – ○ • – ○ •

Introduction

The function block **ALI_TO_ALR** is a composite function block that converts an **ALI adapter** (LINT data type) to an **ALR adapter** (LREAL data type). It acts as a bridge between different adapter interfaces and enables seamless data conversion from 64-bit integers to 64-bit floating-point numbers.

Interface Structure

Event Inputs

The function block does not have separate event inputs. Event-driven processing is handled via the **socket** `ALI_IN`, whose event `E1` triggers the conversion.

Event Outputs

There are no standalone event outputs. The result event is output via the **plug** `ALR_OUT` to its event `E1`.

Data Inputs

The FB has no direct data inputs. The LINT data to be converted is provided via the **socket** `ALI_IN` (data point `D1`).

Data Outputs

Similarly, there are no separate data outputs. The converted LREAL value is output via the **plug** `ALR_OUT` (data point `D1`).

Adapters

Type	Name	Direction	Description
Socket	`ALI_IN`	Input	Adapter type `adapter::types::unidirectional::ALI` (LINT)
Plug	`ALR_OUT`	Output	Adapter type `adapter::types::unidirectional::ALR` (LREAL)

Functionality

The function block operates as a simple forwarder and converter:

1. An event at **Socket** `ALI_IN.E1` triggers the internal function block `Convert` (type `F_LINT_TO_LREAL`).
2. Simultaneously, the LINT value from `ALI_IN.D1` is passed to input `Convert.IN`.
3. The conversion block `Convert` converts the value into an LREAL value according to IEC 61131 conversion logic.
4. After the conversion is complete, `Convert` generates the event `CNF`, which is forwarded to the **plug** `ALR_OUT.E1`.
5. At the same time, the converted LREAL value from `Convert.OUT` is passed to `ALR_OUT.D1`.

The entire process is event-driven and does not use its own state logic.
...

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block**: The logic is implemented exclusively through the interconnection of the standardized conversion block `F_LINT_TO_LREAL` – no proprietary algorithm.
- **Adapter-Oriented Interface**: The function block (FB) completely encapsulates the conversion within the adapter interfaces, allowing it to be used in environments with uniform adapter types (unidirectional).
- **No State Machine**: Due to its pure data flow structure, there is no internal state machine.

State Overview

The FB does not have its own state machine. Execution follows the event chain of the internal network exclusively:

- Idle ⟶ Event sent to `ALI_IN.E1` ⟶ Conversion in progress ⟶ Result event sent to `ALR_OUT.E1` ⟶ Returns to Idle.

Application Scenarios

- Integration of components that provide LINT values (e.g., counters, position sensors) into systems that expect LREAL (e.g., control algorithms with floating-point calculations).
- Conversion from integer-based to floating-point-based adapter interfaces without changing the higher-level architecture.
- Intermediate layer in modular control projects where different data types need to be connected.

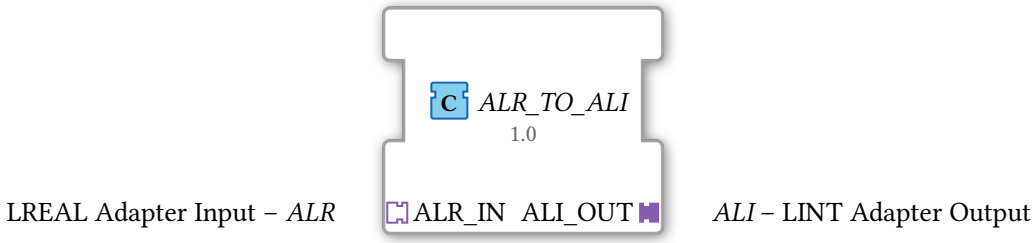
Comparison with Similar Components

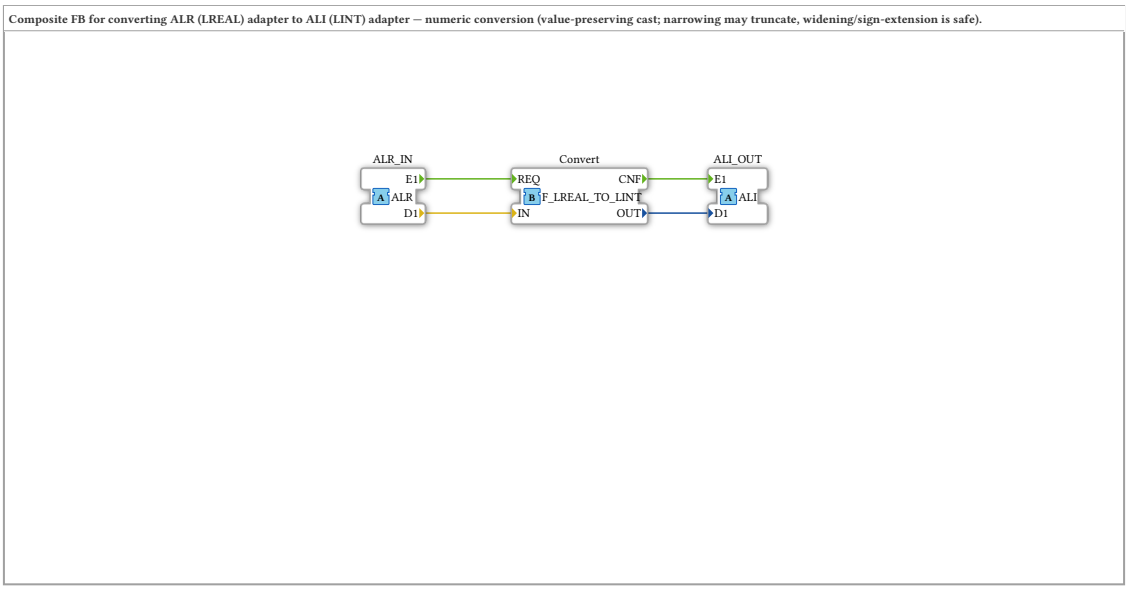
The library contains analogous converters for other data types (e.g., `LINT_TO_REAL`, `DINT_TO_REAL`). `ALI_TO_ALR` is the adapter-specific variant that integrates the conversion into the standardized unidirectional adapters `ALI` and `ALR`. It allows for easy exchange between different adapter types without the calling environment needing to know the details of the conversion.

Conclusion

The FB `ALI_TO_ALR` offers a clean, maintainable, and standards-compliant solution for converting a LINT adapter to an LREAL adapter. By using a proven IEC 61131 conversion block, the implementation remains reliable and efficient. It is particularly suitable for modular control architectures where data type and adapter compatibility is required.

ALR_TO_ALI





• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_TO_ALI** is a composite block that performs an adapter conversion from **ALR (LREAL)** to **ALI (LINT)**. It enables the simple and structured integration of an LREAL-based adapter interface into a LINT-based environment without requiring the implementation of the conversion logic itself. The block is designed for use in IEC 61499-based industrial controllers and utilizes the internal function block `F_LREAL_TO_LINT` according to IEC 61131-3.

Interface Structure

Event Inputs

The function block does not have explicit event inputs. The triggering event is provided via the incoming adapter **ALR_IN** (type `adapter::types::unidirectional::ALR`):

- **ALR_IN.E1** – Input event that starts the conversion.

Event Outputs

The function block (FB) does not have any explicit event outputs. The confirmation event is provided via the outgoing adapter **ALI_OUT** (type `adapter::types::unidirectional::ALI`):

- **ALI_OUT.E1** – Output event that signals the completion of the conversion.

Data Inputs

The FB does not have an explicit data input. The input value is provided via the adapter **ALR_IN**:

- **ALR_IN.D1** – Value of type LREAL to be converted to LINT.

Data Outputs

The FB does not have an explicit data output. The result is output via the **ALI_OUT** adapter:

- **ALI_OUT.D1** – converted value of type LINT.

Adapter

Direction	Name	Type	Description
Socket (Input)	ALR_IN	`adapter::types::unidirectional::ALR`	Unidirectional adapter for LREAL data and events. Includes event port E1 and data port D1 (LREAL).
Plug (Output)	ALI_OUT	`adapter::types::unidirectional::ALI`	Unidirectional adapter for LINT data and events. Includes event port E1 and data port D1 (LINT).

Functionality

The function block (FB) acts as a wrapper for type conversion from LREAL to LINT. The flowchart is as follows:

1. An event (E1) on the input adapter **ALR_IN** triggers the internal block **Convert** (type `iec61131::conversion::F_LREAL_TO_LINT`).
2. The data value from **ALR_IN.D1** (LREAL) is passed to the input **Convert.IN**.
3. The internal block performs the conversion and outputs the result to **Convert.OUT**.
4. The event at the output **Convert.CNF** is forwarded to **ALI_OUT.E1**, so the outgoing adapter signals that the conversion is complete.
5. The converted LINT value is available at **ALI_OUT.D1**.

Call control is achieved via event chaining (E1 → REQ → CNF → E1), ensuring that conversion occurs blocking within a single execution cycle.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Structure**: The function block is implemented as a composite (network) and internally uses the standard function block `F_LREAL_TO_LINT` from the IEC 61131-3 library.
- **Adapter Encapsulation**: The adapter technology allows the specific LREAL and LINT signals to be connected via standardized interfaces (ALR, ALI), enabling high reusability and interchangeability.
- **License**: The source code is licensed under the Eclipse Public License 2.0 (EPL-2.0).
- **Version**: 1.0, released on February 17, 2026, by HR Agrartechnik GmbH, author: Franz Höpfinger.
- **Package Structure**: `adapter::conversion::unidirectional`.

State Overview

The function block does not have an explicit state machine, as it is implemented as a network by a single internal function block. Its behavior is purely event-driven:

- **Idle**: Waiting for an event at the input adapter (ALR_IN.E1).
- **Processing**: Upon arrival of the event, the internal conversion block is activated.
- **Done**: After completion (CNF), the output event (ALI_OUT.E1) is set, and the function block returns to the idle state.

There is no explicit error state, as the conversion block `F_LREAL_TO_LINT` does not provide error handling.

Application Scenarios

- **Migration**: Replacing an ALI-based (LINT) system with an ALR-based (LREAL) system or vice versa, without changing the higher-level logic.
- **Adapter Bridge**: If a component only provides LREAL values, but the subsequent adapter expects LINT, this function block (FB) can be used as an intermediary.
- **Standardized Conversion**: In libraries or frameworks that require a uniform implementation of type conversions via adapters.

Comparison with Similar Function Blocks

In the IEC 61499 and IEC 61131 worlds, numerous conversion function blocks exist:

- **F_LREAL_TO_LINT** (direct): A single function block that converts a simple value but does not provide an adapter interface.
- **ALR_TO_ALI** (this FB): Encapsulates the direct function block in an adapter structure, enabling connection via standardized adapter ports.
- **Other Type Conversions**: Similarly structured function blocks for, e.g., REAL → DINT or LREAL → DWORD, can be created using the same pattern.

The key advantage of **ALR_TO_ALI** is its reusability within the adapter context and the clean separation of events and data.

Conclusion

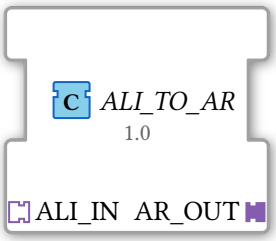
The **ALR_TO_ALI** function block is a useful composite block that enables clean and standardized adapter conversion from LREAL to LINT. It inherits the robustness of the IEC 61131-3 conversion block and seamlessly integrates it into the adapter-based communication architecture. Its open license and clear interface make it well-suited for use in modular automation solutions.

Numeric Index (unidirectional)

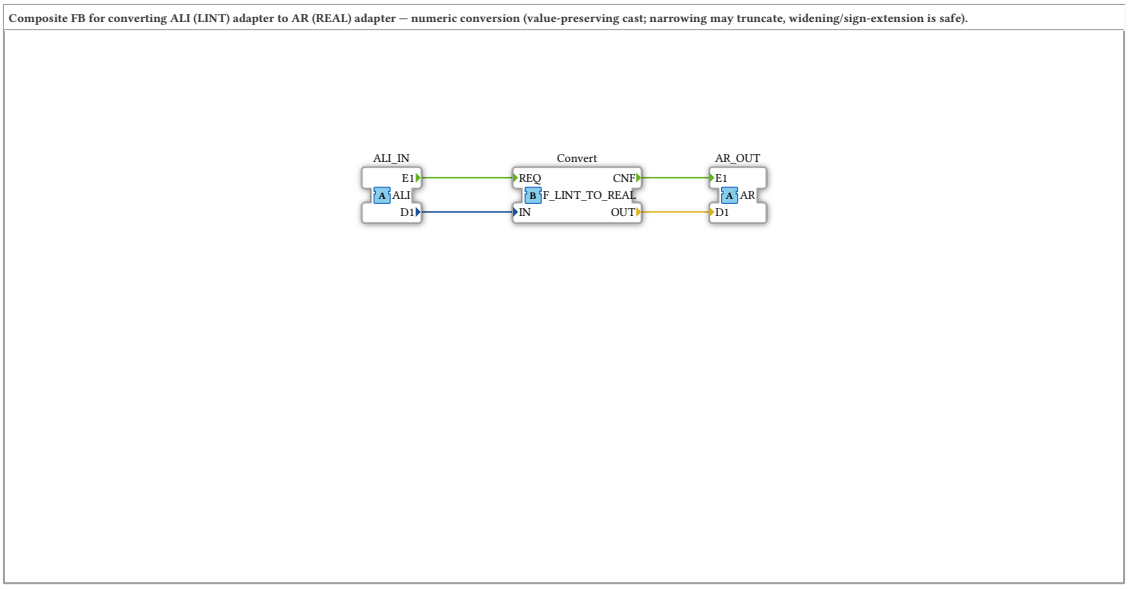
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALI_TO_AR

LINT Adapter Input – *ALI*



AR – REAL Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block **ALI_TO_AR** is a composite function block (FB) that converts a LINT adapter (ALI) into a REAL adapter (AR). It serves as a universal conversion interface to provide 64-bit integer values from the IEC 61131 standard into floating-point values for further processing in control applications. The function block is licensed under the Eclipse Public License 2.0 and is provided by HR Agrartechnik GmbH.

Interface Structure

The function block does not have direct event or data ports. All communication takes place via two predefined unidirectional adapters:

Event Inputs

– None (the event interface is provided via the **ALI_IN** adapter)

Event Outputs

– None (the event interface is provided via the **AR_OUT** adapter)

Data Inputs

– None (the data interface is provided via the **ALI_IN** adapter)

Data Outputs

– None (the data interface is provided via the **AR_OUT** adapter)

Adapter

Direction	Name	Type	Description
Socket	ALI_IN	`adapter::types::unidirectional::ALI`	LINT Input Adapter (64-bit Integer)
Plug	AR_OUT	`adapter::types::unidirectional::AR`	REAL Output Adapter (Floating Point)

Each adapter contains an event channel (E1) and a data channel (D1), which are connected via the module's internal network.

Functionality

The module performs a simple data-driven conversion:

1. An event arrives at the event input **E1** via the **ALI_IN** socket – simultaneously, the corresponding LINT value is provided at the data input **D1**.

2. This event is forwarded to the internal module `iec61131::conversion::F_LINT_TO_REAL` (event **REQ**).
3. The internal module converts the LINT value into a REAL value and signals completion via the event output **CNF**.
4. The converted REAL value is transferred via the data connection to the data input of the plug adapter **AR_OUT** (D1).
5. Simultaneously, the **E1** event of the **AR_OUT** adapter is triggered, allowing the connected logic to process the new REAL value.

The entire conversion process is performed without state storage – each request is processed immediately.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block**: The ALI_TO_AR is a composite function block consisting exclusively of the internal standard block `F_LINT_TO_REAL`.
- **Library Integration**: The conversion utilizes the IEC 61131 library `iec61131::conversion::F_LINT_TO_REAL`, which is declared as an import.
- **Package Structure**: The block is organized within the package `adapter::conversion::unidirectional`.
- **License**: The source code is licensed under the Eclipse Public License 2.0 (EPL-2.0).

State Overview

The function block has no states of its own. It operates purely event-driven and data-flow-oriented. There are no initialization or error-related states – every incoming request is processed directly.

Application Scenarios

- **Counter Evaluation**: A LINT counter (e.g., pulse counter) delivers a large integer that must be converted into a REAL value for control purposes.
- **Data Preprocessing**: In a PLC application, measured values are available as LINT (e.g., from an encoder) that need to be converted into REAL for further calculations or visualization.
- **Adapter Bridge**: The function block serves as a standardized converter between two different adapter types in a cross-device communication infrastructure.

Comparison with Similar Function Blocks

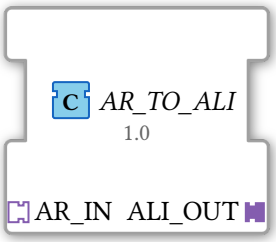
- **ALI_TO_ARI**: Converts LINT to an ARRAY-from-REAL – for multiple values or buffers.
- **LREAL_TO_AR**: Converts LREAL (80-bit floating-point) to REAL – for losses due to lower precision.
- **Direct Type Conversion** to IEC 61131 functions (e.g., `LINT_TO_REAL`) – without adapter encapsulation, requires manual event wiring.

The ALI_TO_AR offers the advantage of a standardized adapter interface over simple function calls, enabling plug-and-play use in modular systems.

Conclusion

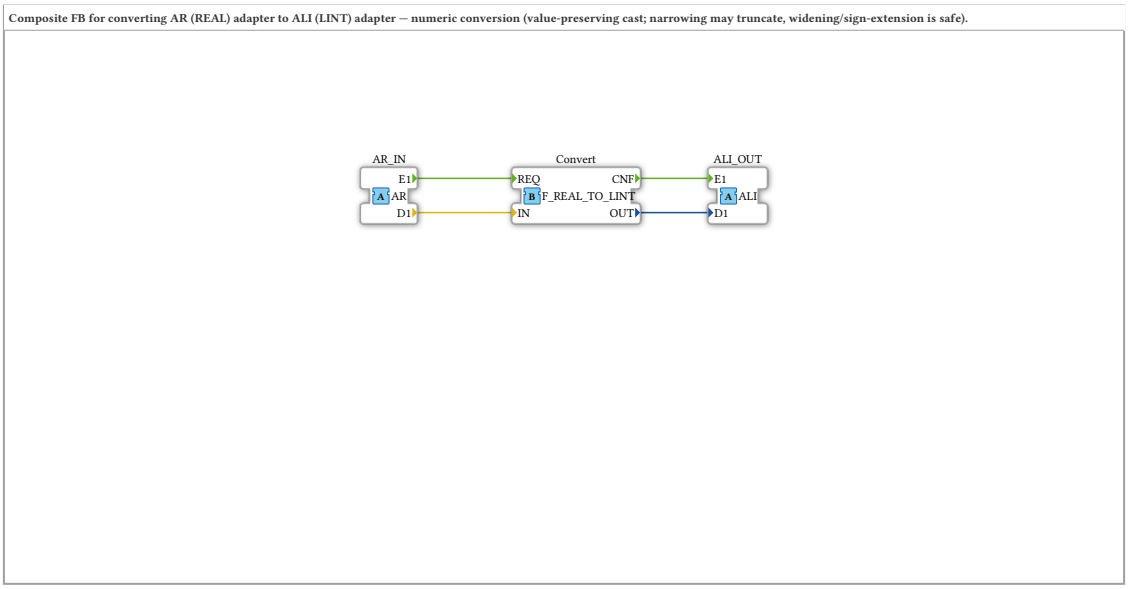
The **ALI_TO_AR** is a compact, reusable function block for converting LINT values to REAL values via an adapter interface. Its simple internal structure and clear event-data coupling make it an indispensable component for any application that needs to convert integer sensor data into floating-point values for control or visualization purposes. Licensing under EPL-2.0 allows unrestricted use in open and commercial projects.

AR_TO_ALI



REAL Adapter Input – *AR*

ALI – LINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The composite function block `AR_TO_ALI` converts a REAL adapter (`AR-IN`) into a LINT adapter (`ALI-OUT`). It enables the seamless integration of floating-point values from adapter `AR` into applications that expect 64-bit integer values via adapter `ALI`. The conversion is performed via an internal function block `F_REAL_TO_LINT` according to IEC 61131.

Interface Structure

The function block does not have any independent event or data inputs/outputs at the block level. All signal transmission occurs via adapter sockets or plugs.

Event Inputs

No dedicated event inputs. The event is received via socket `AR_IN` (interface `E1`).

Event Outputs

No dedicated event outputs. The event is sent via plug `ALI_OUT` (interface `E1`).

Data Inputs

No dedicated data inputs. Data is received via socket `AR_IN` (interface `D1`).

Data Outputs

No dedicated data outputs. The data is sent via the plug `ALI_OUT` (interface `D1`).

Adapter

Direction	Label	Type	Compatible Adapter Type	Description

Socket	AR_IN	iec61131::types::unidirectional::AR	AR Adapter (REAL)	Receiving adapter for the REAL value.
Plug	ALI_OUT	iec61131::types::unidirectional::ALI	ALI Adapter (LINT)	Providing adapter for the converted LINT value.

Functionality

The function block `iec61131::conversion::F_REAL_TO_LINT` is instantiated within the internal network. As soon as an event occurs at socket `AR_IN` (via interface `E1`), the internal conversion block is triggered using the designated event connection `AR_IN.E1 -> Convert.REQ`. Simultaneously, the data value `AR_IN.D1` is passed to input `Convert.IN`. After the conversion (REAL ⟶ LINT) is complete, the converter confirms with the event `Convert.CNF`. This event is forwarded to plug `ALI_OUT.E1`, and the converted value `Convert.OUT` is output to `ALI_OUT.D1`.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Pure Adapter Coupling**: The function block does not use direct event or data inputs, but communicates exclusively via the standardized unidirectional adapters `AR` and `ALI`. This makes it easy to integrate into existing adapter structures.
- **IEC 61131 Compliance**: The conversion block used, `F_REAL_TO_LINT`, complies with the IEC standard and ensures a standardized, deterministic conversion.
- **Runtime Behavior**: The conversion is event-driven and occurs without any additional delay. The output value is available in the same execution cycle in which the input event is received.

State Overview

The Composite FB itself does not have its own state machine. Its behavior is entirely determined by its internal circuitry. The following logical states exist:

- **Idle**: No event at the input. The output `ALI_OUT` remains unchanged.
- **Converting**: An event is received at socket `AR_IN`. The internal converter is activated, and the conversion process begins.
- **Completed**: The conversion is complete; the output event is sent, and the new data value is available at the plug.

Application Scenarios

- **Sensor-Actuator Coupling**: A REAL adapter (e.g., from an analog sensor) must be connected to a control module that processes only LINT values via the ALI adapter.
- **Protocol Conversion**: In heterogeneous automation networks where data formats need to be converted to connect different adapter types.
- **Test and Simulation Environments**: Replacing an ALI adapter with a simulated REAL adapter and converting back for testing purposes.

Comparison with Similar Function Blocks

FB	Conversion	Interface	Note
`AR_TO_ALI`	REAL ⟶ LINT	AR / ALI Adapter	Specifically for Adapter Coupling
`F_REAL_TO_LINT`	REAL ⟶ LINT	Standalone I/O (REQ/CNF, IN/OUT)	Direct converter without adapters
The `AR_TO_ALI` encapsulates the `F_REAL_TO_LINT` and adds the adapter connections. The advantage lies in its simple, adapter-			

network-compatible connection. A direct converter without adapters would not be usable in adapter structures.			

Conclusion

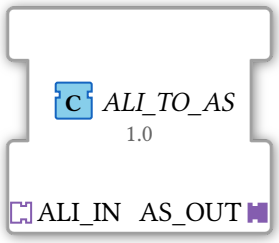
The `AR_TO_ALI` is a compact, purpose-built composite function block that enables the conversion of REAL to LINT values via the standardized unidirectional adapters `AR` and `ALI`. Its clean encapsulation and the use of an IEC-compliant core component provide a reliable and reusable solution for industrial automation. The FB is particularly suitable for scenarios where a clean separation between data format and adapter interface is required.

Numeric Index (unidirectional)

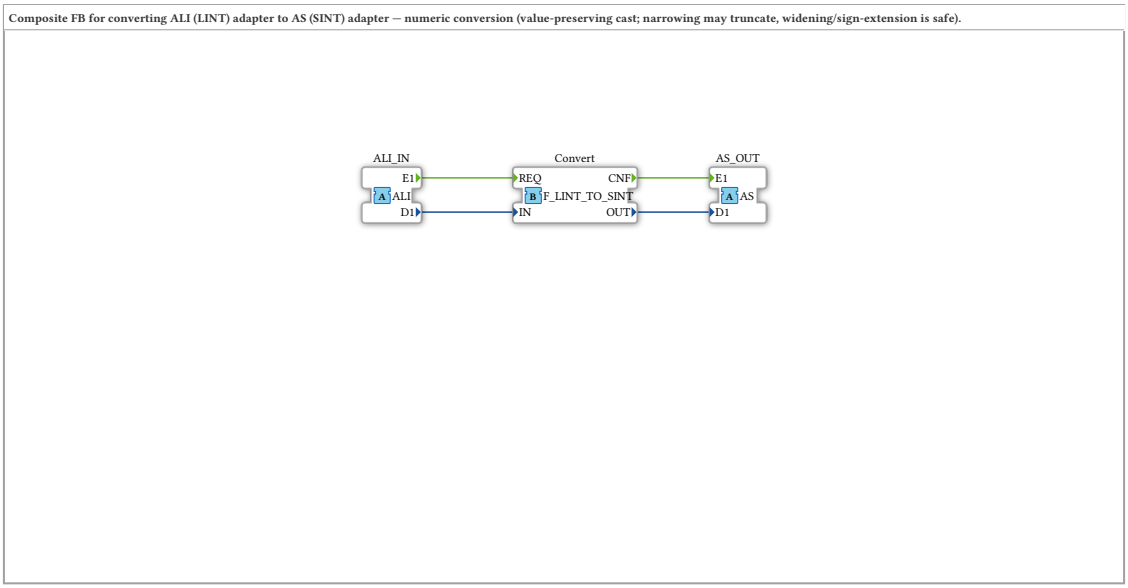
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALI_TO_AS

LINT Adapter Input – *ALI*



AS – SINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **ALI_TO_AS** function block is a composite function block that converts an ALI adapter (data type LINT) into an AS adapter (data type SINT). It encapsulates the conversion of a 64-bit integer value to an 8-bit integer value, thus enabling seamless integration between different adapter interfaces in the 4diac IDE.

Interface Structure

Event Inputs

None. The function block has no separate event inputs; event control is handled via the connected adapters.

Event Outputs

None. Events are forwarded exclusively via the AS adapter.

Data Inputs

None. The LINT signal to be converted is read in via the ALI adapter.

Data Outputs

None. The converted SINT signal is output via the AS adapter.

Adapters

Name	Type	Direction	Description
ALI_IN	`adapter::types::unidirectional::ALI`	Socket (Input)	Receives a LINT value (64-bit) via event E1 and date D1.
AS_OUT	`adapter::types::unidirectional::AS`	Plug (Output)	Provision of the converted SINT value (8-bit) via event E1 and data D1.

Functionality

The function block operates on the principle of pure data flow without its own state machine.

1. The **ALI_IN** adapter receives the **E1** event and provides the corresponding LINT value via data **D1**.
2. This event triggers the internal function block **F_LINT_TO_SINT**, which performs the conversion.
3. After the conversion is complete, the **CNF** event of the conversion block is activated, which in turn triggers the **E1** output event of the **AS_OUT** adapter and passes the converted SINT value via data **D1**.

The conversion corresponds to the IEC 61131 function `LINT_TO_SINT`. For values outside the SINT range (-128…127), an overflow occurs – the function block does not perform any additional range checks.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Structure**: The function block consists exclusively of a single conversion block (`F_LINT_TO_SINT`).
- **Unidirectional Adapters**: Both the input and output interfaces are defined as unidirectional adapters – data flows in only one direction.
- **No State Machine**: Due to the simple transformation, an internal state machine is not used; the logic is limited to event-data chaining.

State Overview

The function block has **no** internal state machine. It operates purely data/event-driven and performs an immediate conversion upon each incoming event.

Application Scenarios

- **Integration of LINT data sources** (e.g., from high-resolution counters, timestamps) into systems that only process SINT values (e.g., simple actuator controls).
- **Bridges between different adapter types** within a 4diac application, without the need for manual conversion blocks.

Comparison with Similar Blocks

- **ALI_TO_INT**: Converts LINT to INT (16-bit).
- **LINT_TO_DINT** or similar: Direct type conversions without adapter encapsulation.

The **ALI_TO_AS** is distinguished by its specific adapter interface; it simplifies the reuse of pre-built adapter couplings in modules.

Conclusion

The **ALI_TO_AS** is a specialized composite component for lossy conversion of a LINT value to a SINT value, embedded in a unidirectional adapter structure. It offers clean, reusable encapsulation of the conversion logic and facilitates the connection of diverse data sources and sinks within the 4diac development environment.

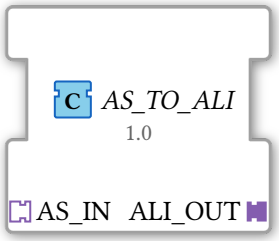
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

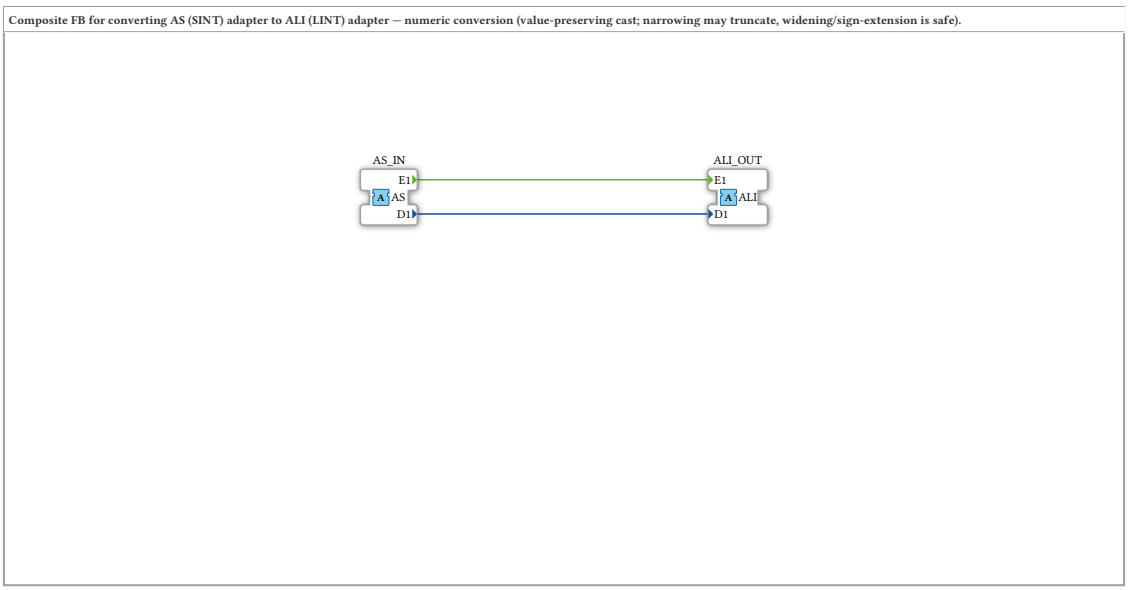
]

AS_TO_ALI

SINT Adapter Input – *AS*



ALI – LINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **AS_TO_ALI** function block is a composite block for converting a SINT adapter input (AS) into a LINT adapter output (ALI). It serves as a bridge between components that use different integer data types and enables the easy integration of SINT-based signals into LINT-based systems.

Interface Structure

Event Inputs

The function block has no directly visible event inputs. Event control is handled via the socket adapter **AS_IN**, which provides an event **E1**.

Event Outputs

The function block has no directly visible event outputs. Event output is handled via the plug adapter **ALI_OUT**, which forwards an event **E1**.

Data Inputs

No direct data inputs. The data value **D1** is read via the socket adapter **AS_IN** (data type SINT).

Data Outputs

No direct data outputs. The converted data value **D1** is output via the plug adapter **ALI_OUT** (data type LINT).

Adapters

- **AS_IN** (Socket, type: `adapter::types::unidirectional::AS`) – SINT adapter input with event E1 and data D1 (SINT).
- **ALI_OUT** (Plug, type: `adapter::types::unidirectional::ALI`) – LINT adapter output with event E1 and data D1 (LINT).

Functionality

The AS_TO_ALI function block is implemented as a composite block. When an event arrives at **AS_IN.E1**, it is forwarded directly to **ALI_OUT.E1**. Simultaneously, the incoming SINT data value **AS_IN.D1** is converted to a LINT value and passed to **ALI_OUT.D1**. The conversion is lossless, as the value range of LINT completely covers that of SINT.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Structure** – no independent state logic, only forwarding and conversion.
- **Synchronized Event and Data Paths** – data is only passed after an event.
- **Implicit Type Conversion** – the conversion from SINT to LINT occurs automatically within the composite block's network.

State Overview

The function block (FB) has no states of its own. Its functionality is limited to event-driven data passing with type conversion.

Application Scenarios

- Connecting a SINT-based sensor to a LINT-based control system.
- Migrating existing SINT components to a LINT infrastructure.
- Heterogeneous systems where different integer data types need to be converted.

Comparison with Similar Function Blocks

Similar converters exist for other data types (e.g., BOOL_TO_BYTE, INT_TO_DINT). The AS_TO_ALI function block is specifically designed for converting SINT to LINT and is conceived as an adapter function block, allowing it to be directly integrated into adapter-based communication structures.

Conclusion

The AS_TO_ALI function block offers a simple, reliable, and efficient solution for converting SINT to LINT data. Its composite structure allows it to be integrated into existing 4diac projects without additional programming and facilitates the use of different data types in automation systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Numeric Index (unidirectional)

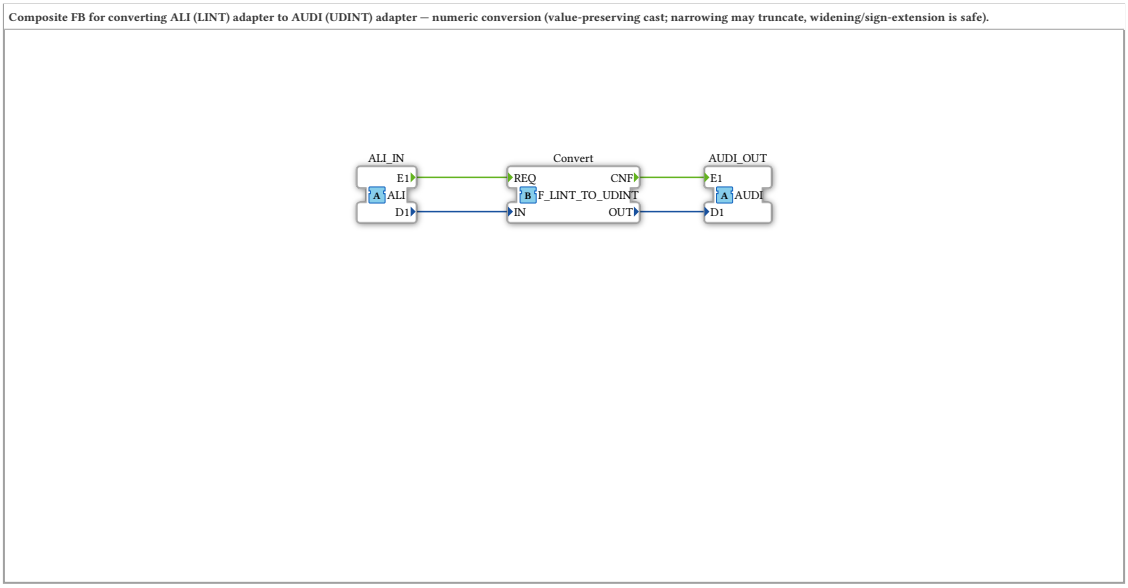
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

ALI_TO_AUDI



LINT Adapter Input – *ALI*

AUDI – UDINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_TO_AUDI** is a composite block that converts an ALI adapter (LINT) into an AUDI adapter (UDINT). It serves as a unidirectional converter between two different data type adapters for the IEC 61499 profile interface. The conversion is performed using an internal block *F_LINT_TO_UDINT* from the IEC 61131 library.

Interface Structure

The function block has no direct event or data inputs/outputs. All communication takes place via two adapter interfaces.

Event Inputs

No dedicated event inputs are available. The event is received via the ALI adapter.

Event Outputs

No dedicated event outputs are available. The result event is sent via the AUDI adapter.

Data Inputs

No dedicated data inputs are available. The input data value is provided via the ALI adapter.

Data Outputs

No dedicated data outputs are available. The converted data value is output via the AUDI adapter.

Adapters

Adapter	Type	Direction	Description

ALI_IN	adapter::types::unidirectional::ALI	Socket (Input)	Returns the LINT value (64-bit signed) and the associated event.
AUDI_OUT	adapter::types::unidirectional::AUDI	Plug (Output)	Provides the converted UDINT value (32-bit unsigned) and the acknowledgment event.

Functionality

As soon as the ALI_IN adapter receives an event (E1), the associated LINT data value (D1) is read. This data value is passed to the internal function block *F_LINT_TO_UDINT*, which converts it into a UDINT value. After successful conversion, the internal function block sends an acknowledgment event (CNF), which is forwarded via the AUDI_OUT adapter (E1). Simultaneously, the converted UDINT value (OUT) is passed to the AUDI_OUT adapter (D1).

The conversion from LINT (64-bit signed) to UDINT (32-bit unsigned) is unsigned and can lead to overflows or information loss for values outside the UDINT range (0 … 4,294,967,295).

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Function Block**: The function block consists of an internal network with a single conversion block.
- **Unidirectional Adapter-to-Adapter Converter**: Data flows only from ALI_IN to AUDI_OUT.
- **No State Logic**: The function block has no internal state machine; the conversion occurs deterministically with each event.

State Overview

The function block does not have its own state machine. The state is determined solely by the internal block *F_LINT_TO_UDINT*, which performs a one-time conversion with each REQ event and immediately delivers the result.

Application Scenarios

- **System Integration**: When one system component (e.g., sensor, drive) provides an ALI adapter (LINT), while another component expects an AUDI adapter (UDINT).
- **Protocol Adaptation**: In heterogeneous control environments where different IEC 61499 adapter profiles for integers are used.
- **Migration**: Exchange of 64-bit LINT values for 32-bit UDINT values, provided the value range limits are respected.

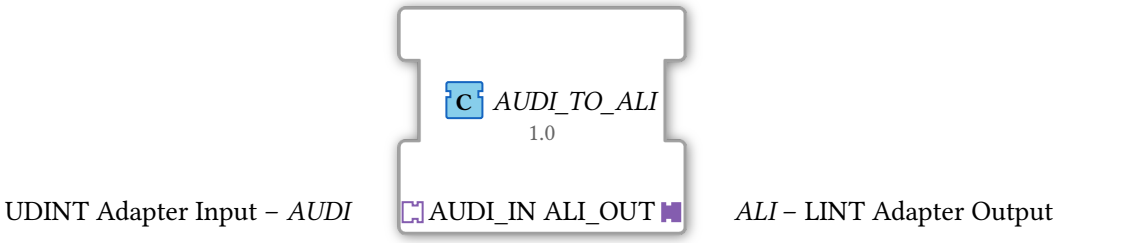
Comparison with Similar Function Blocks

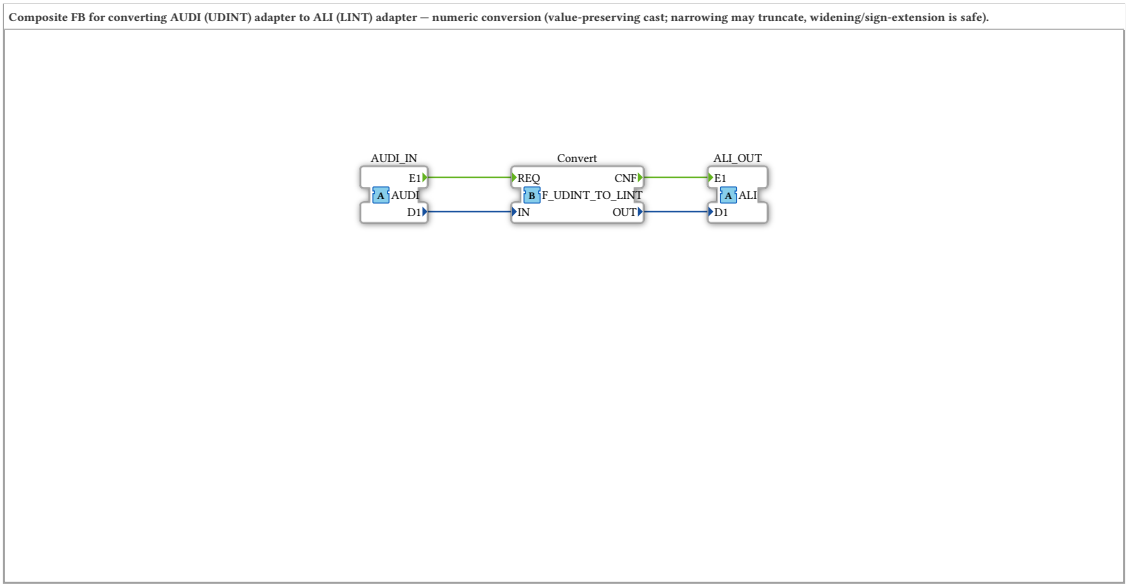
- **F_LINT_TO_UDINT (direct)**: This function block performs pure data conversion without an adapter. *ALI_TO_AUDI* encapsulates this function in an adapter-based interface.
- **ALI_TO_xxx/Converter**: Other adapter converters (e.g., ALI_TO_BOOL, ALI_TO_REAL) use analogous mechanisms but differ in target type and value range.

Conclusion

The *ALI_TO_AUDI* function block is a lean composite function block for simple adapter conversion from LINT to UDINT. It utilizes an existing IEC 61131 component and extends it with a standardized IEC 61499 adapter interface. This facilitates integration into existing systems without requiring the underlying conversion logic to be reimplemented.

AUDI_TO_ALI





• – ◦ • – ◦ • – ◦ •

Introduction

The **AUDI_TO_ALI** function block is a composite module that bridges an **AUDI** adapter (of type *UDINT*) and an **ALI** adapter (of type *LINT*). It converts incoming UDINT values to LINT values using a standardized conversion function and forwards them to the output adapter. This module is typically used in applications that need to connect different address or integer types within a 4diac adapter concept.

Interface Structure

Event Inputs

Name	Description
No dedicated event inputs	Event control is handled via the internal socket AUDI_IN.E1.

Event Outputs

Name	Description
No custom event outputs	Event forwarding is handled via the internal plug ALI_OUT.E1.

Data Inputs

Name	Data Type	Description
No custom data inputs	–	The data value is supplied by socket AUDI_IN.D1.

Data Outputs

Name	Data Type	Description
No custom data outputs	–	The converted value is provided via the ALI_OUT.D1 plug.

Adapter

Type	Name	Direction	Description
AUDI (unidirectional)	`AUDI_IN`	Socket (Input)	UDINT data adapter that provides a value based on event E1 and date D1.
ALI (unidirectional)	`ALI_OUT`	Plug (Output)	LINT data adapter that outputs the converted value based on event E1 and date D1.

Functionality

The module functions as a pure **type conversion bridge** for adapters:

1. **Event Receipt**: An event on socket `AUDI_IN.E1` activates the internal conversion module `F_UDINT_TO_LINT` via its event input `REQ`.
2. **Data Conversion**: The current UDINT value of `AUDI_IN.D1` is placed on the data input `IN` of the converter. `F_UDINT_TO_LINT` performs the IEC 61131-compliant conversion from an unsigned 32-bit integer (UDINT) to a signed 64-bit integer (LINT).
3. **Output**: After successful conversion, the converter signals with `CNF` that the output value is valid. This event is directly passed to the plug `ALI_OUT.E1`, and the converted LINT value from `Convert.OUT` is placed on `ALI_OUT.D1`.

The entire processing is **event-driven** and takes place within a single execution step (without any additional delay).

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block** – it contains no logic of its own but delegates the conversion entirely to the IEC 61131 block `F_UDINT_TO_LINT`.
- **Adapter-oriented interface** – All inputs and outputs are handled via standardized unidirectional adapters (`AUDI` and `ALI`), enabling easy integration into existing adapter networks.
- **License & Copyright** – This block is licensed under the **Eclipse Public License 2.0** and was developed by **HR Agrartechnik GmbH** (as of 2026).
- **Type safety** – Conversion from UDINT (unsigned 32-bit) to LINT (signed 64-bit) is lossless, as the value is simply sign-extended.
- **Package name** – In the CompilerInfo, the block is listed as `adapter::conversion::unidirectional`.

State overview

As a composite block, `AUDI_TO_ALI` does not have its own states. The state logic is contained within the integrated `F_UDINT_TO_LINT` block, which transitions to a processing state after each `REQ` event and acknowledges this with `CNF`.

Application Scenarios

- **Transition between different automation protocols** – when a system delivers UDINT values via an AUDI adapter, but the target system expects LINT values via an ALI adapter.
- **Data adaptation in adapter hierarchies** – redirection of 32-bit counter values (e.g., speed, pulse counter) to a 64-bit bus.
- **Migration** – legacy systems with UDINT sections can be connected to new 64-bit interfaces without changing the adapter structure.
- **Test Environments** – Simple loop-through of type conversion between two adapters, without separate conversion modules in the network.

Comparison with similar modules

Module	Type	Special Feature
`F_UDINT_TO_LINT`	Pure converter	No adapters, direct data I/O; no event handling via adapters, requires its own control.
`AUDI_TO_ALI` (this one)	Adapter converter	Encapsulates the conversion in an adapter-based interface; events and data are automatically routed via the adapters.
`ALI_TO_AUDI` (hypothetical)	Adapter converter	Would convert LINT backwards to UDINT (potentially lossy).

The **main difference** compared to directly using `F_UDINT_TO_LINT` is the seamless integration into adapter networks and the avoidance of additional lines for event and data connections.

Conclusion

The **AUDI_TO_ALI** function block is a practical adapter converter that bridges the gap between a UDINT-based AUDI adapter and a LINT-based ALI adapter. By internally using the standardized IEC 61131 converter `F_UDINT_TO_LINT`, the conversion is efficient and type-safe. The block is particularly suitable for modular architectures where different address widths are encountered and simplifies the reuse of adapter interfaces in heterogeneous automation environments.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

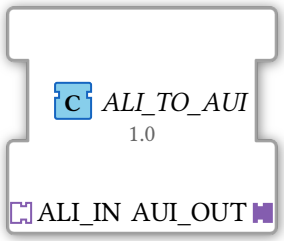
]

Numeric Index (unidirectional)

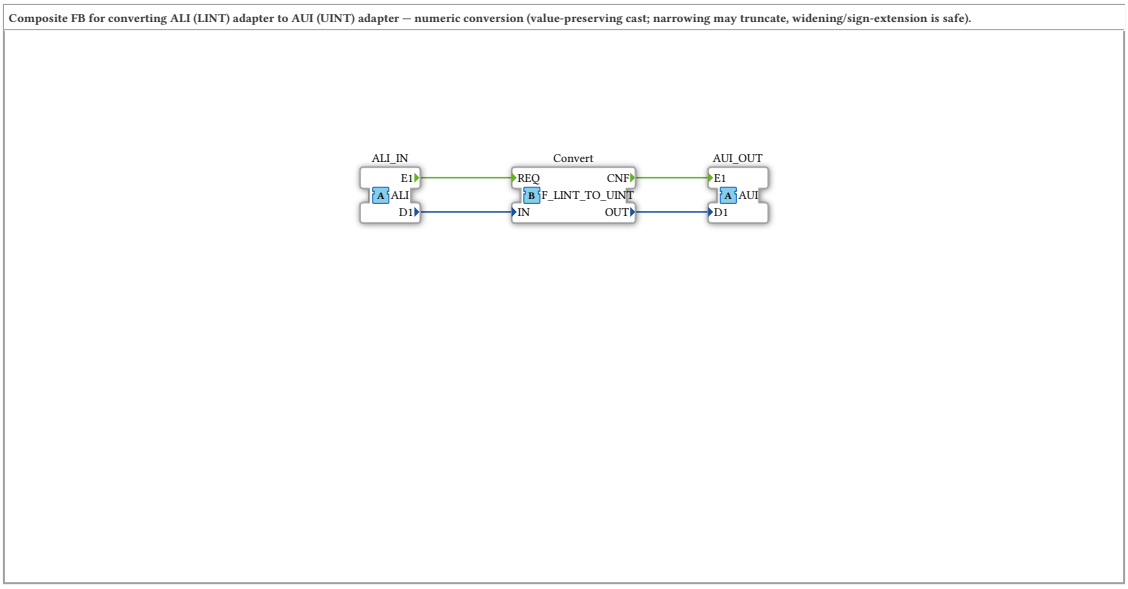
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

ALI_TO_AUI

LINT Adapter Input – *ALI*



AUI – UINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **ALI_TO_AUI** is a composite block for converting an ALI adapter (LINT data type) into an AUI adapter (UINT data type). It allows the seamless integration of components with different adapter definitions in a 4diac environment. The block is based on the internal conversion block `F_LINT_TO_UINT` from the IEC 61131 library.

Interface Structure

Event Inputs

Not present – event control is handled by the adapters.

Data Inputs

Not present – data transfer is handled by the adapters.

Data Outputs

Not present – data is transferred via the adapters.

Adapters

Type	Name	Comment
Socket	`ALI_IN`	Input adapter of type `adapter::types::unidirectional::ALI` – provides LINT value and associated event.
Plug	`AUI_OUT`	Output adapter of type `adapter::types::unidirectional::AUI` – provides the converted UINT value and associated event.

Functionality

Internally, this function block connects socket `ALI_IN` to the input of the conversion function block `F_LINT_TO_UINT`.

- The converter's `E1` event output from `ALI_IN` controls the `REQ` input.
- The data value `D1` from `ALI_IN` (LINT) is placed on the converter's `IN` input.
- After successful conversion, the converter signals the event to the plug `AUI_OUT` with `CNF`, and the converted value (UINT) is written to its `D1` output.

`The event outputE1fromALI_INis controlled via the converter'sREQ` input.

`The data valueD1fromALI_IN(LINT) is placed on theIN` input. ... The entire process is event-driven: An incoming event at the ALI adapter triggers the conversion and generates a corresponding output event at the AUI adapter.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Data Type Conversion:** The internal module converts a 64-bit signed integer (LINT) to a 16-bit unsigned integer (UINT). Since the value range of UINT (0...65535) is significantly smaller than that of LINT, values outside this range result in an overflow/loss.
- **Dependency:** The module requires the library `iec61131::conversion` (specifically the module `F_LINT_TO_UINT`).
- **Adapter-Free Interface:** All data exchange occurs exclusively via adapters, enabling modular and reusable integration into larger networks.

State Overview

The function block itself does not have its own state machine. The state logic is fully implemented by the internal function block `F_LINT_TO_UINT`. This typically has the states **IDLE** (waiting for REQ) and **BUSY** (performing conversion). Upon completion, the result is provided and the CNF event output is activated.

Application Scenarios

- **Adapter Bridge:** A sensor that provides data in ALI format (LINT) is to be connected to an actuator that expects an AUI input (UINT).
- **System Integration:** In heterogeneous automation systems where different adapter definitions are used, the function block serves as a simple converter without manual programming.
- **Prototyping:** Quickly replace adapters during the development phase without having to change the underlying data flow logic.

Prototyping:

Comparison with Similar Components

Analog converter components exist, such as `ALI_TO_UDI` (LINT → UDINT) or `DINT_TO_UINT`. This component specializes in combining the unidirectional ALI and AUI adapters. Unlike directly connecting the adapters via data type conversion at the network layer, this composite function block offers a compact, reusable solution with clearly defined event signals.

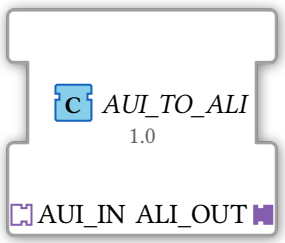
Conclusion

The ALI_TO_AUI component is a practical and easy-to-understand tool for converting between ALI and AUI adapters. Its composite structure encapsulates the necessary data conversion and event forwarding, allowing it to integrate seamlessly into existing 4diac projects. Its ease of use and clear interface separation make it a valuable tool for adapter integration.

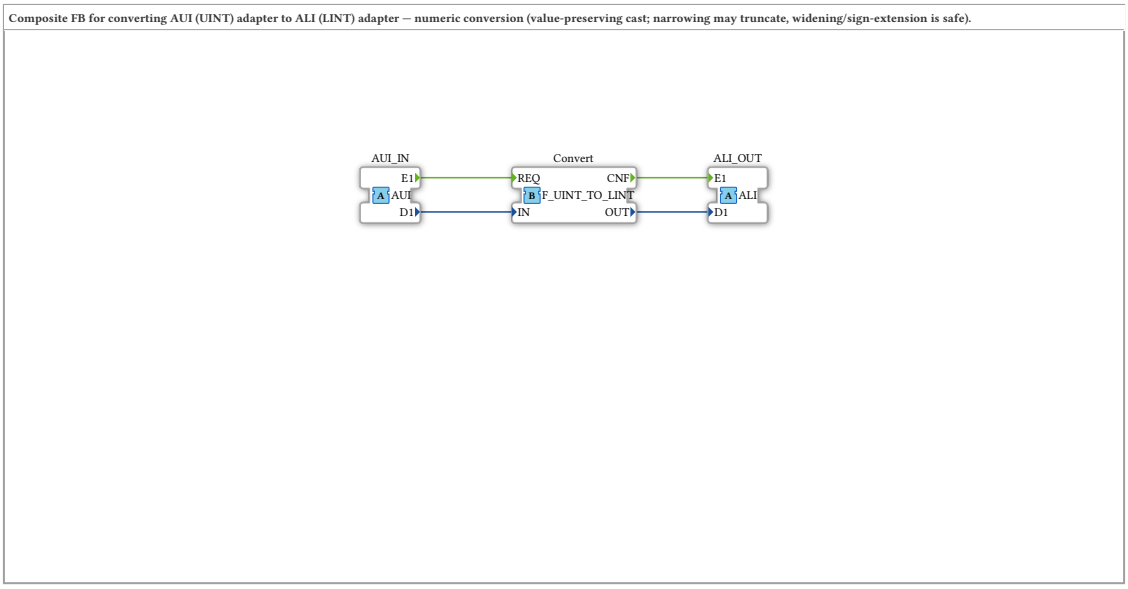
Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AUI_TO_ALI



UINT Adapter Input – *AUI* *ALI* – LINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The AUI_TO_ALI function block is a composite function block that converts an AUI adapter (UINT) into an ALI adapter (LINT). Internally, it uses the IEC 61131 function block F_UINT_TO_LINT for the conversion.

Interface Structure

Event Inputs

- **AUI_IN.E1** (Event via Socket AUI_IN): Starts the conversion when an event is present at the socket (via the AUI adapter).

Event Outputs

- **ALI_OUT.E1** (Event via Plug ALI_OUT): Outputs the output after the conversion is complete (via the ALI adapter).

Data Inputs

- **AUI_IN.D1** (Data via Socket AUI_IN): UINT value to be converted to a LINT value.

Data Outputs

- **ALI_OUT.D1** (Data via Plug ALI_OUT): Converted LINT value.

Adapters

- **Socket AUI_IN** (Type: adapter::types::unidirectional::AUI): Receives UINT data and the associated event.
- **Plug ALI_OUT** (Type: adapter::types::unidirectional::ALI): Sends the converted LINT data and the associated event.

Functionality

The function block waits for an event at the input adapter AUI_IN (E1). As soon as this event arrives, the incoming UINT value (D1) is forwarded to the internal conversion block F_UINT_TO_LINT. The conversion is performed according to the IEC 61131 standard, converting the unsigned UINT value into a signed LINT value (64-bit integer). After the conversion is complete, the output event is triggered at the ALI_OUT adapter (E1), and the converted LINT value is passed to ALI_OUT.D1.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).

- The function block (FB) is implemented as a composite type and encapsulates the conversion logic in a reusable adapter interface.
- Uses the standardized function block F_UINT_TO_LINT from the IEC 61131 library.
- License: Eclipse Public License 2.0 (EPL-2.0).
- Developed by HR Agrartechnik GmbH (Version 1.0, 2026-02-17).

State Overview

The function block (FB) does not have its own state machine, as it passively maps the states of the internal block as a composite. The internal F_UINT_TO_LINT operates according to a simple scheme: wait for the REQ event, process the conversion, and trigger the CNF event.

Application Scenarios

- Integration of a system that provides UINT values via an AUI adapter into a system that expects LINT values via an ALI adapter.
- Data bridge between different adapter types in a distributed controller.
- Extension of 4diac applications when type conversion between UINT and LINT is required.

Comparison with Similar Function Blocks

Analogous adapter conversion blocks exist for other data types (e.g., UINT_TO_DINT, UINT_TO_REAL, etc.). The AUI_TO_ALI function block is specifically designed for combining the unidirectional AUI and ALI adapters, enabling a unified adapter interface without additional manual conversion.

Conclusion

The composite function block AUI_TO_ALI offers a simple and reliable way to convert UINT data via AUI adapters to LINT data via ALI adapters. It is well-encapsulated, uses standard function blocks, and can be easily integrated into 4diac projects.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

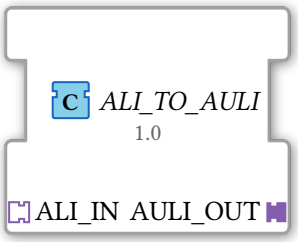
]

Numeric Index (unidirectional)

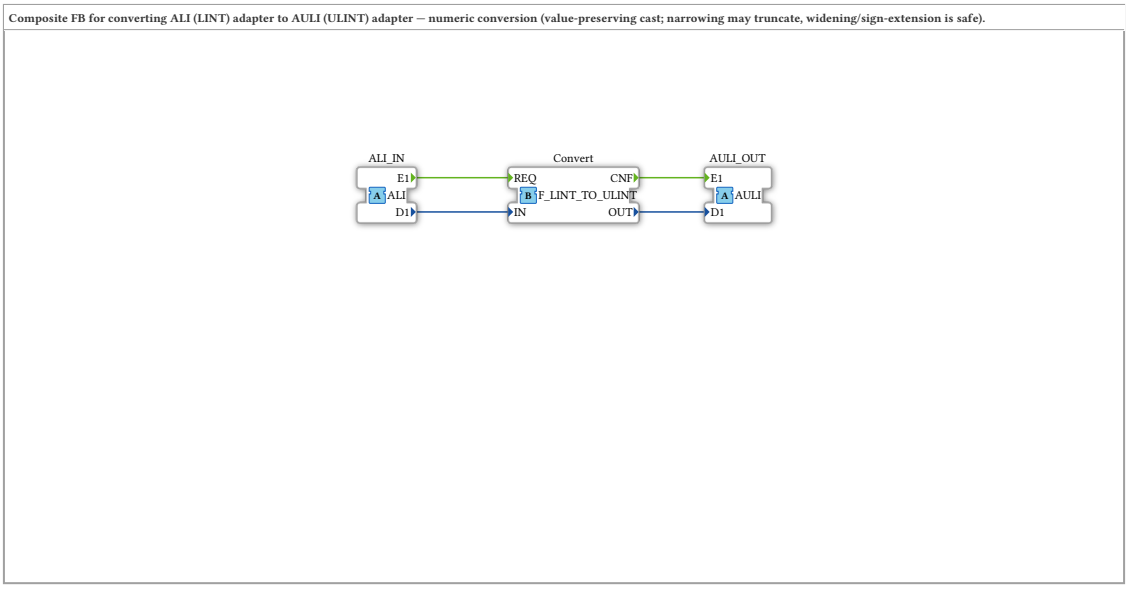
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALI_TO_AULI

LINT Adapter Input – *ALI*



AULI – ULINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block `ALI_TO_AULI` is a composite block that implements bidirectional adapter conversion between the LINT adapter (ALI) and the ULINT adapter (AULI). It serves as a standardized interface for converting a signed 64-bit integer value (LINT) into an unsigned 64-bit integer value (ULINT). The block encapsulates the IEC 61131-3 conversion block `F_LINT_TO_ULINT` and simplifies integration into adapter-based architectures.

Interface Structure

The block does not have its own event or data interfaces, but only two adapter connections (socket and plug). The internal logic is handled by the embedded sub-module and the direct connection of the adapter signals.

Event Inputs

– None (events are received via the ALI adapter)

Event Outputs

– None (events are sent via the AULI adapter)

Data Inputs

– None (data is received via the ALI adapter)

Data Outputs

– None (data is output via the AULI adapter)

Adapter

Adapter Type	Direction	Name	Comment
`adapter::types::unidirectional::ALI`	Socket (Input)	`ALI_IN`	LINT adapter input (provides an event and a LINT value)
`adapter::types::unidirectional::AULI`	Plug (output)	`AULI_OUT`	ULINT adapter output (passes on an event and a ULINT value)

Functionality

The module operates as a simple pass-through conversion:

1. An incoming event at socket `ALI_IN.E1` triggers the internal module `Convert` (`F_LINT_TO_ULINT`) via its event input `REQ`.
2. Simultaneously, the LINT value provided by the socket (`ALI_IN.D1`) is passed to the data input `IN` of `Convert`.
3. After successful conversion, `Convert` signals an event at its output `CNF`, which is then forwarded directly to the plug `AULI_OUT.E1`.
4. The converted ULINT value (`Convert.OUT`) is then passed in parallel to the data input `D1` of the AULI adapter.

Thus, a complete, event-driven, and data-synchronous conversion is achieved without any additional delay or logic.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Sub-block used**: `iec61131::conversion::F_LINT_TO_ULINT` – standard-compliant type conversion according to IEC 61131-3.
- **Adapter coupling**: The block utilizes 4diac's unidirectional adapter concept and enables loose coupling between different data type interfaces.
- **No state management**: As a composite block, `ALI_TO_AULI` does not have its own internal state; all decisions are made by the sub-block.
- **Platform independence**: The block can be used in any 4diac environment that provides the IEC 61131 conversion blocks.

State overview

The block itself does not contain a state machine. All state transitions are controlled by the internal `F_LINT_TO_ULINT`, which switches to the "convert" state after an event at input `REQ` and outputs a confirmation event upon completion. Since these states are not externally visible, a detailed description is omitted.

Application Scenarios

- **System Integration**: Connecting an older control system that uses signed LINT values to a modern component that can only process unsigned ULINT values.
- **Data Conversion in Adapter Chains**: Embedding between two adapters, e.g., in a data preprocessing chain that requires a uniform ULINT interface.
- **Migration**: Replacing LINT-based components with ULINT equivalents without having to change the entire adapter infrastructure.

Comparison with Similar Components

The library contains analogous converters for other data types, such as `DINT_TO_UDINT` or `REAL_TO_LREAL`. `ALI_TO_AULI` differs in its specific adapter interface and the encapsulation of the conversion component within a composite. While the pure IEC components have individual data and event interfaces, this component enables direct adapter-to-adapter conversion, which facilitates the design of modular and interchangeable components.

Conclusion

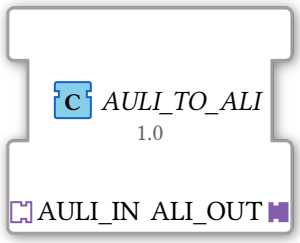
The `ALI_TO_AULI` component offers a simple and reliable way to convert LINT adapters to ULINT adapters. It cleanly encapsulates the standard conversion in an adapter-based architecture, thus contributing to the reusability and interoperability of 4diac applications. By using the proven IEC component, the conversion is correct and platform-independent.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

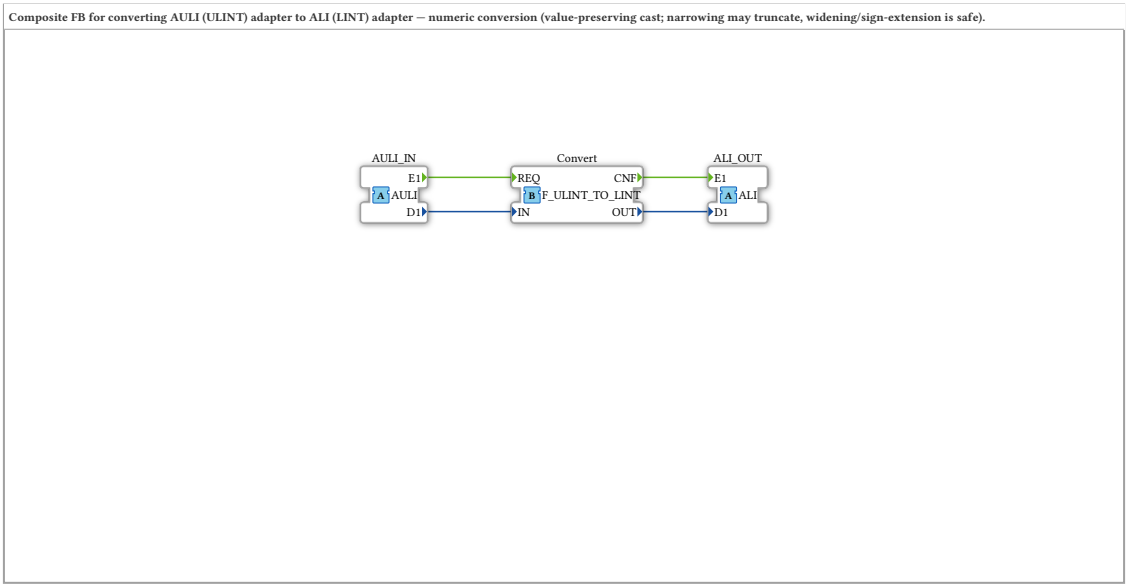
]

AULI_TO_ALI



ULINT Adapter Input – *AULI*

ALI – LINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **AULI_TO_ALI** function block is a composite module that performs an adapter conversion from **AULI** (ULINT adapter) to **ALI** (LINT adapter). It converts an unsigned 64-bit integer value (ULINT) into a signed 64-bit integer value (LINT) and forwards it via the corresponding adapter interfaces.

Interface Structure

Event Inputs

Events are received via the incoming adapter **AULI_IN**:

- `E1` – Starts the conversion of the incoming ULINT value.

Event Outputs

Events are sent via the outgoing adapter **ALI_OUT**:

- `E1` – Acknowledges the completed conversion and indicates that the LINT value is present at the output.

Data Inputs

Data is provided via the incoming adapter **AULI_IN**:

- `D1` (Data type: ULINT) – The unsigned 64-bit value to be converted.

Data Outputs

Data is output via the outgoing adapter **ALI_OUT**:

- `D1` (Data type: LINT) – The converted signed 64-bit value.

Adapters

- **AULI_IN** (Socket) – Unidirectional adapter of type `adapter::types::unidirectional::AULI` for receiving ULINT values and associated events.
- **ALI_OUT** (Plug) – Unidirectional adapter of type `adapter::types::unidirectional::ALI` for outputting LINT values and associated events.

Functionality

The module internally uses the IEC 61131 conversion module `F_ULINT_TO_LINT`. As soon as the event `E1` arrives at the **AULI_IN** adapter, the ULINT value present at `D1` is read and passed to the converter. After conversion, the resulting LINT value is passed to the **ALI_OUT** adapter, and simultaneously, the event `E1` is sent to its output. The data and event flows are strictly synchronized: Conversion only occurs on demand via the event `E1`.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-Based Encapsulation**: The function block completely encapsulates the conversion logic behind standardized unidirectional adapters (AULI/ALI), allowing for easy integration into existing adapter-based data flows.
- **Composite Structure**: Because it is a composite function block, no separate state machine is required; the logic is implemented by linking the sub-function block `F_ULINT_TO_LINT`.
- **Type Safety**: The conversion from ULINT to LINT preserves the value range as long as the ULINT value is not greater than `LINT#9223372036854775807` (maximum LINT). Exceeding this limit results in undefined behavior (no error handling in the function block).

State Overview

The function block does not have its own state machine. It reacts event-driven:

- **Waiting for event `E1`**: No activity.
- **During Conversion**: The internal function block `F_ULINT_TO_LINT` is executed (the duration depends on the clock cycle).
- **Output to `CNF`**: The event `E1` is sent at the output.

Application Scenarios

- **Data type conversion in control systems** when a sensor or source delivers ULINT values, but the subsequent system expects LINT values (signed).
- **Adapter bridge** between components that communicate exclusively via AULI or ALI adapters, e.g., in modular automation architectures.
- **Integration into cross-device data flows** without changing the actual application logic.

Comparison with similar components

Similar adapter conversions exist for other data types, e.g., `UINT_TO_INT` or `UDINT_TO_DINT`. The **AULI_TO_ALI** module specializes in 64-bit integers (ULINT/LINT) and uses the same adapter interface (unidirectional), allowing it to be seamlessly integrated into existing adapter circuits.

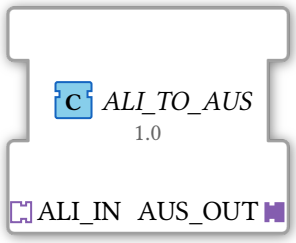
Conclusion

The **AULI_TO_ALI** module is a simple yet effective solution for converting ULINT to LINT values using standardized adapters. It simplifies the integration of different data types into unidirectional communication paths and is particularly well-suited for modular IEC 61499 applications.

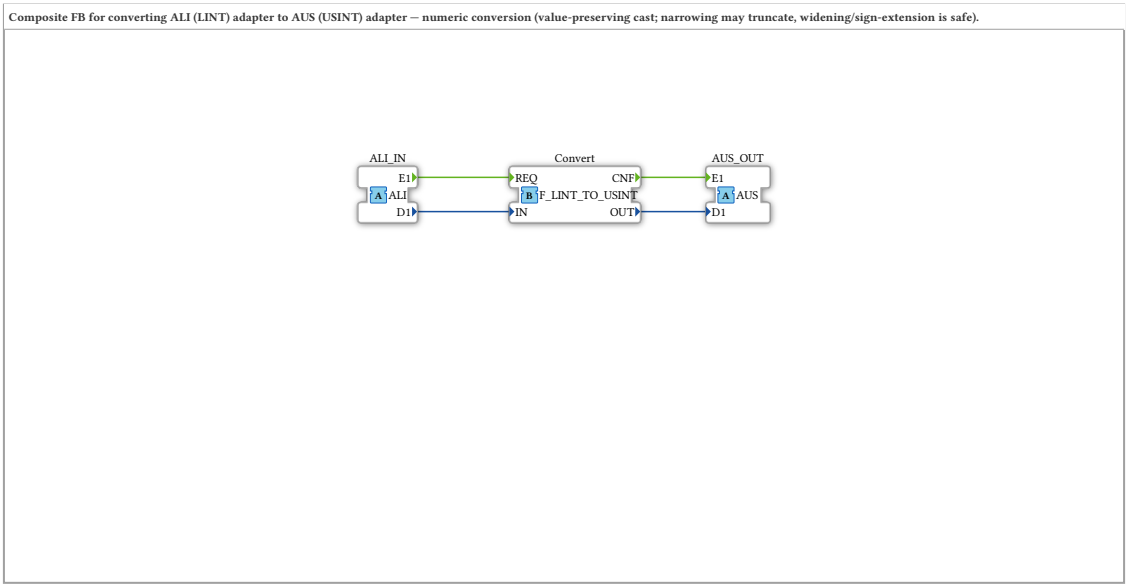
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALI_TO_AUS



LINT Adapter Input – *ALI* *AUS* – USINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **ALI_TO_AUS** is a composite function block according to IEC 61499, used to convert an **ALI adapter** (data of type **LINT**, Long Integer) into an **AUS adapter** (data of type **USINT**, Unsigned Short Integer). The block encapsulates the necessary conversion logic to enable a simple and type-safe connection between differently typed adapter interfaces.

Interface Structure

Event Inputs

- **ALI_IN.E1** (via adapter socket) – triggers the conversion as soon as an event arrives from the connected ALI source.

Event Outputs

- **AUS_OUT.E1** (via adapter plug) – signals the completion of the conversion to the downstream OFF sink.

Data Inputs

- **ALI_IN.D1** (via adapter socket) – the LINT (Long Integer) value to be converted.

Data Outputs

- **AUS_OUT.D1** (via adapter plug) – the converted USINT (Unsigned Short Integer) value.

Adapter

Direction	Name	Type	Description

Socket (Input)	ALI_IN	adapter::types::unidirectional::ALI	Receives the LINT data value and the associated event
Plug (Output)	OFF_OUT	adapter::types::unidirectional::OFF	Provides the converted USINT value and the confirmation event

Functionality

The function block internally contains an instance of the function block **F_LINT_TO_USINT** from the IEC 61131 conversion library. The process is as follows:

1. An event **ALI_IN.E1** triggers the data transfer.
2. The LINT value present at **ALI_IN.D1** is forwarded to the input **IN** of the internal converter.
3. The internal function block performs the type conversion from LINT to USINT (taking value range limits into account).
4. After successful conversion, the internal function block generates a **CNF** event, which is output as **OFF_OUT.E1**.
5. The converted result is sent from **OUT** of the internal block to **AUS_OUT.D1**.

Event and data flows are coupled entirely via the internal network; the user only sees the two adapter interfaces.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Reusable Converter**: The block uses the standardized **F_LINT_TO_USINT** (iec61131::conversion), so the conversion logic does not need to be reimplemented.
- **Adapter-Based**: The entire interface is implemented as a unidirectional adapter, simplifying exchange and reuse in larger systems.
- **License**: The block is provided under the **Eclipse Public License 2.0** (EPL-2.0).

State Overview

Since this is a composite block, there is no explicit top-level state machine. The internal converter **F_LINT_TO_USINT** operates in an event-driven manner and performs exactly one conversion step after each **REQ** event. The function block remains inactive as long as no event is present.

Application Scenarios

- **Interface Adaptation**: Connecting components that use **LINT** (e.g., high-resolution counters) to those that expect **USINT** (e.g., simple actuator controllers).
- **System Integration**: When a predefined adapter type (ALI) is available and a different adapter type (AUS) is required – e.g., when combining modules from different manufacturers.
- **Protocol Conversion**: Used in data preprocessing chains where long integer values need to be mapped to a smaller range.

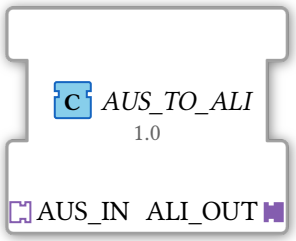
Comparison with Similar Function Blocks

Numerous data type-specific conversion blocks exist (e.g., LINT_TO_INT, LINT_TO_DINT, USINT_TO_LINT). The **ALI_TO_AUS** stands out because it not only encapsulates the pure type conversion but also embeds it directly into an **adapter interface**. This reduces the wiring effort at the block level and increases the readability of the overall design.

Conclusion

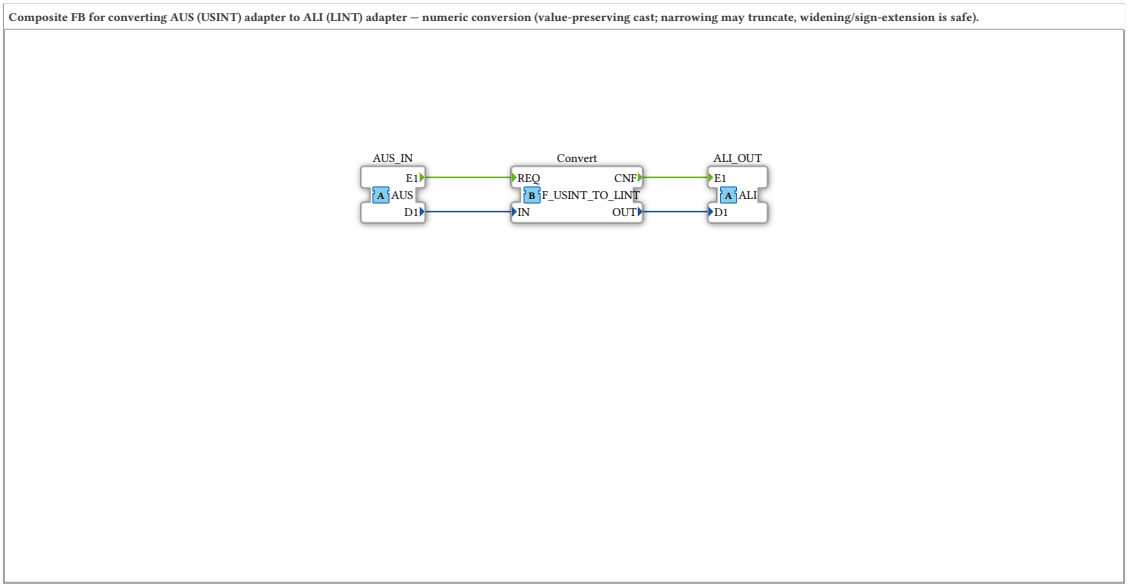
The **ALI_TO_AUS** is a practical and robust function block for converting LINT adapter data to USINT adapter data. By using a standardized converter and clear adapter encapsulation, a reusable, type-safe, and easy-to-maintain solution is provided for various control systems.

AUS_TO_ALI



USINT Adapter Input – *AUS*

ALI – LINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_TO_ALI` is a composite function block (FB) that converts an AUS adapter (USINT values) into an ALI adapter (LINT values). It encapsulates the conversion process and provides a unidirectional interface between these two adapter types. Internally, the function block `F_USINT_TO_LINT` from the IEC 61131 library is used.

Interface Structure

Event Inputs

No dedicated event inputs. Event control is handled via the `AUS_IN` adapter (socket).

Event Outputs

No dedicated event outputs. Event output is handled via the `ALI_OUT` adapter (plug).

Data Inputs

No dedicated data inputs. Data is received via the `AUS_IN` adapter.

Data Outputs

No dedicated data outputs. Data is output via the `ALI_OUT` adapter.

Adapters

- **OFF_IN** (Socket) – Receives an OFF adapter with a unidirectional USINT data type. Includes:
- Event input `E1`
- Data input `D1` (USINT)
- **ALI_OUT** (Plug) – Sends an ALI adapter with a unidirectional LINT data type. Includes:
- Event output `E1`
- Data output `D1` (LINT)

Functionality

The function block operates in an event-driven manner:

1. An event at the `E1` input of the `AUS_IN` adapter initiates processing.
2. The current value of `AUS_IN.D1` (USINT) is passed to the internal function block `Convert` (`F_USINT_TO_LINT`).
3. After conversion, the result (LINT) is output via `ALI_OUT.D1`, and simultaneously an event is sent to the `E1` output of the `ALI_OUT` adapter.

Type conversion is performed according to the IEC 61131 definition of `F_USINT_TO_LINT`: USINT (0..255) is mapped to LINT (64-bit) without loss of sign.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block**: The conversion logic is encapsulated and reusable.
- **Adapter-Based**: Input/output is exclusively via adapters (`AUS` and `ALI`), which facilitates integration into adapter-oriented architectures.
- **Unidirectional**: The function block supports only one data flow direction (OFF → ALI).
- **Library Import**: It uses the function block `iec61131::conversion::F_USINT_TO_LINT` from the IEC 61131 library.
- **Package Structure**: The function block is organized in the package `adapter::conversion::unidirectional`.

State Overview

The `AUS_TO_ALI` function block does not have its own state machines. The internal process is purely event-driven and stateless:

- **Wait**: for an event from `AUS_IN.E1`
- **Convert**: immediate conversion of the USINT value to LINT
- **Output**: immediate sending of the result via `ALI_OUT`

Each conversion is independent of previous calls.

Application Scenarios

- **Interface Adaptation**: A system that provides data as an AUS adapter (USINT) must be connected to a receiver that expects an ALI adapter (LINT).
- **Data Preparation**: Conversion of low-resolution sensor data or manipulated variables (USINT) into a wider range of values (LINT) for further calculations.
- **Adapter Cascading**: As part of a chain of adapter conversions, e.g., in a modular control architecture.

Comparison with Similar Function Blocks

- **Direct Use of `F_USINT_TO_LINT`**: This function block operates at the data level without an adapter. `AUS_TO_ALI` additionally encapsulates the adapter logic and is therefore better suited for adapter-based communication.
- **Further Adapter Conversions**: For other type combinations (e.g., `INT_TO_DINT`, `BYTE_TO_WORD`), analogous composite devices exist that follow the same design pattern.

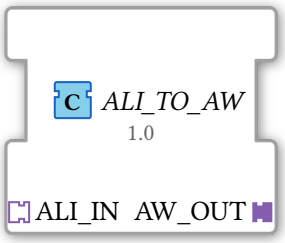
Conclusion

The `AUS_TO_ALI` device is a simple yet effective solution for unidirectional conversion between USINT and LINT adapters. It increases reusability and clarity in systems that rely on adapter interfaces and avoids the manual wiring of conversion logic.

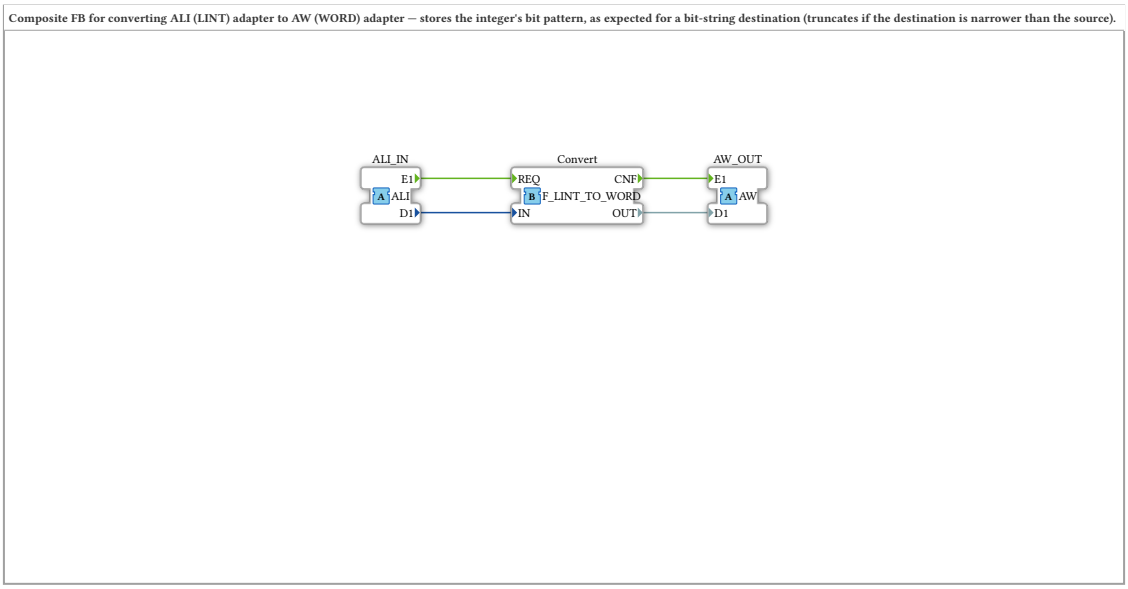
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALI_TO_AW



LINT Adapter Input – *ALI* *AW* – WORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_TO_AW** is a composite function block (FB) that converts a LINT value (ALI adapter) into a WORD value (AW adapter). It encapsulates the conversion logic of the IEC 61131-3 function block `F_LINT_TO_WORD` in an adapter-based interface. The block is designed for use in unidirectional adapter communication paths and enables a clean separation of data format and application.

Interface Structure

The block has no direct event or data inputs/outputs at the FB level. Communication takes place exclusively via two adapters (a socket and a plug).

Event Inputs

No direct event inputs are available. The incoming event is provided via the adapter `ALI_IN.E1`.

Event Outputs

No direct event outputs are available. The outgoing event is provided via the adapter `AW_OUT.E1`.

Data Inputs

No direct data inputs are available. The LINT value is obtained via the adapter `ALI_IN.D1`.

Data Outputs

No direct data outputs are available. The WORD value is output via the adapter `AW_OUT.D1`.

Adapters

Adapter Name	Type	Direction	Comment

ALI_IN	adapter::types::unidirectional::ALI	Socket (Input)	LINT adapter input; sends event to E1 and LINT data to D1
AW_OUT	adapter::types::unidirectional::AW	Plug (Output)	WORD adapter output; sends event to E1 and WORD data to D1

Functionality

1. The module expects an event (to E1) and a corresponding LINT data value (to D1) via socket ALI_IN.
2. The event is forwarded to the internal conversion module F_LINT_TO_WORD (type iec61131::conversion::F_LINT_TO_WORD) (connection ALI_IN.E1 → Convert.REQ). Simultaneously, the LINT value is passed to the converter's data input IN (connection ALI_IN.D1 → Convert.IN).
3. The internal function block performs the conversion and signals completion via its output CNF.
4. This event is passed to the plug AW_OUT.E1. The converted WORD value (output OUT of the converter) is simultaneously applied to the data input AW_OUT.D1.

Thus, the function block makes the data provided by the input adapter available in converted form at the output adapter, controlled by the event chain.

1. This event is then passed to the plug plug AW_OUT.E1. The converted WORD value (output OUT of the converter) is simultaneously applied to the data input AW_OUT.D1.

Thus, the function block makes the data provided by the input adapter available in converted form at the output adapter.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **IEC 61131-3 Conversion**: The actual conversion is performed using the standardized function block F_LINT_TO_WORD, which performs a lossy conversion from 64-bit (LINT) to 16-bit (WORD) (truncation or range checking depending on the implementation).
- **Adapter-Based Interface**: The function block is fully embedded in 4diac's adapter technology and is suitable for unidirectional communication paths (package adapter::conversion::unidirectional).
- **Versioning**: Version 1.0, released under the Eclipse Public License 2.0 (EPL-2.0).
- **No State Management**: The function block delegates all state logic to the inner F_LINT_TO_WORD.

State Overview

Since it is a composite function block, ALI_TO_AW does not have its own state machine. The internal converter F_LINT_TO_WORD typically implements the following implicit states:

State	Description
IDLE	Waiting for an event at the input ALI_IN.E1.
CONVERT	After receiving REQ, the conversion is performed.
DONE	After the conversion is complete, CNF is activated and the result value is provided at the output.

The internal function block then returns to the IDLE state.

Application Scenarios

- **Control Engineering**: Conversion of a 64-bit counter value (e.g., from an encoder) into a 16-bit value for a simple display or a word-based interface.
- **Data Consolidation**: Use in adapter chains when a subsequent component only accepts WORD inputs, but the source provides LINT data.
- **Test and Simulation Environments**: Generation of WORD data from LINT test values to verify adapter communication paths.

Comparison with Similar Components

- **LINT_TO_DWORD / LINT_TO_BYTE**: Other conversions with different target widths; `ALI_TO_AW` is specifically designed for use with the unidirectional adapters `ALI` and `AW`.
- **Direct use of F_LINT_TO_WORD**: `ALI_TO_AW` offers a higher level of abstraction through adapter interfaces and facilitates integration into adapter-based architectures, whereas the pure function block only has event and data interfaces.

Conclusion

The function block `ALI_TO_AW` is a practical and reusable building block for converting LINT to WORD adapters. It encapsulates the typical IEC 61131-3 conversion in an adapter-friendly package, thus simplifying integration into existing or new 4diac projects. Its clear, unidirectional interface and adherence to the Eclipse Public License model make it a robust component for automation solutions that need to connect different data widths.

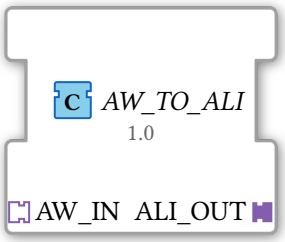
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

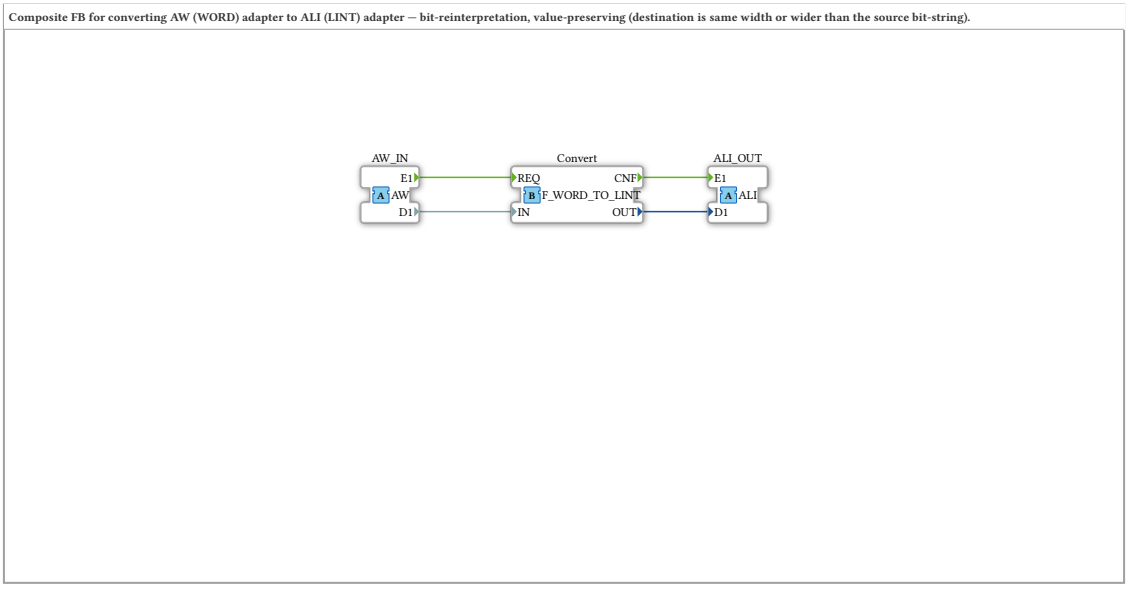
]

AW_TO_ALI

WORD Adapter Input – *AW*



ALI – LINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AW_TO_ALI** function block is a composite function block that converts a unidirectional adapter of type **AW** (WORD) into an adapter of type **ALI** (LINT). It is used to convert data between different adapter interfaces without requiring the user to implement the actual conversion logic. Internally, the function block uses the IEC 61131 function block `F_WORD_TO_LINT` and provides typical event/data control via the supplied adapter interfaces.

Interface Structure

Event Inputs

An event of type **E1** is received via the **Socket AW_IN** (Type: `adapter::types::unidirectional::AW`). This event triggers the conversion of the incoming data value.

Event Outputs

An event **E1** is output via the **Plug ALI_OUT** (Type: `adapter::types::unidirectional::ALI`) after successful conversion. This indicates that the converted value is available at the data output.

Data Inputs

Incoming data is received via the **Socket AW_IN**:

- **D1** (Data Type: WORD) – the 16-bit value to be converted.

Data Outputs

The converted data is output via the **ALI_OUT** plug:

- **D1** (Data type: LINT) – the converted 64-bit value (Long Integer).

Adapters

The module has two adapter interfaces:

- **Socket AW_IN** – Unidirectional input adapter (type AW) that establishes a connection to a previous module that provides WORD data.
- **ALI_OUT** – Unidirectional output adapter (type ALI) that passes the converted LINT data to subsequent modules.

Both adapters are unidirectional, meaning they transmit events and data in only one direction (input → output).

Functionality

The process within the composite block is strictly sequential:

1. An event **E1** at socket **AW_IN** is forwarded to the internal function block **F_WORD_TO_LINT** (instance name *Convert*) and activates its **REQ input**.
2. Simultaneously, the incoming data value **D1** (WORD) is placed at the **IN input** of the converter.
3. The converter `F_WORD_TO_LINT` performs the type conversion from WORD to LINT and provides the result at its **OUT output**.
4. After the conversion is complete, the converter generates a **CNF event**, which is passed to the **E1 input** of the output adapter **ALI_OUT**.
5. Simultaneously, the converted value (LINT) is transferred from the **OUT output** of the converter to the data output **D1** of the **ALI_OUT adapter**.

This completes the data conversion at a safe, event-driven time.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Composite Block** – The conversion logic is completely encapsulated in an internal network consisting of only a single conversion function block. The block does not have its own ECC state machine.
- **Adapter-Based Interface** – Instead of individual event/data inputs/outputs, adapters are used. This enables a modular, reusable connection in adapter-based control architectures.
- **Package Structure** – The function block is located in the package `adapter::conversion::unidirectional` and uses the converter `iec61131::conversion::F_WORD_TO_LINT` from the IEC 61131 conversion library.
- **Unidirectional Direction** – Data flows only from the input adapter to the output adapter; reverse communication is not supported.
- **License** – The source code is licensed under the Eclipse Public License 2.0 (EPL-2.0), developed by HR Agrartechnik GmbH.

State Overview

Since this is a composite function block without its own state machine (ECC), there are no defined states. The functionality is entirely controlled by the embedded function block `F_WORD_TO_LINT`, which performs a one-time, event-driven conversion. The function block is always ready when an event arrives at the input adapter. After the conversion, it waits for the next event.

Application Scenarios

- **Data Conversion in Adapter Chains** – If a control system is based on unidirectional adapters and one component delivers WORD data, but a subsequent component expects LINT data, AW_TO_ALI can be inserted as an intermediary.
- **Connecting Field Devices with Different Bit Widths** – For example, integrating a sensor that delivers 16-bit measured values (WORD) into logic that works with 64-bit values (LINT) (e.g., for high-resolution counters or timestamps).
- **Type Conversion in Libraries** – Extending an existing adapter library with convenient conversion blocks to ensure compatibility between different data formats.

Comparison with Similar Blocks

Block	Input (adapter)	Output (adapter)	Conversion	Special Features
AW_TO_ALI	AW (WORD)	ALI (LINT)	WORD ⟶ LINT	Unidirectional, Composite
`F_WORD_TO_LINT` (IEC 61131)	Single REQ input, IN (WORD)	Single CNF output, OUT (LINT)	WORD ⟶ LINT	Standard FB, no adapter
`ALI_TO_AW` (hypothetical)	ALI (LINT)	AW (WORD)	LINT ⟶ WORD	Reverse direction

The advantage of AW_TO_ALI lies in its seamless integration into adapter-based systems, whereas the pure IEC component `F_WORD_TO_LINT` would require separate wiring. Components with adapter interfaces simplify control design because they offer standardized plugs/sockets.

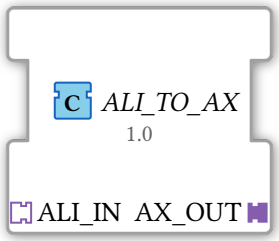
Conclusion

AW_TO_ALI is a compact, adapter-based conversion component for converting WORD to LINT data. By utilizing the proven IEC 61131 converter `F_WORD_TO_LINT` and encapsulating it in a composite structure, it offers a clean, event-driven interface for unidirectional adapter applications. The component is particularly suitable for use in modular automation systems where data needs to be exchanged between components with different data widths.

Numeric Index (unidirectional)

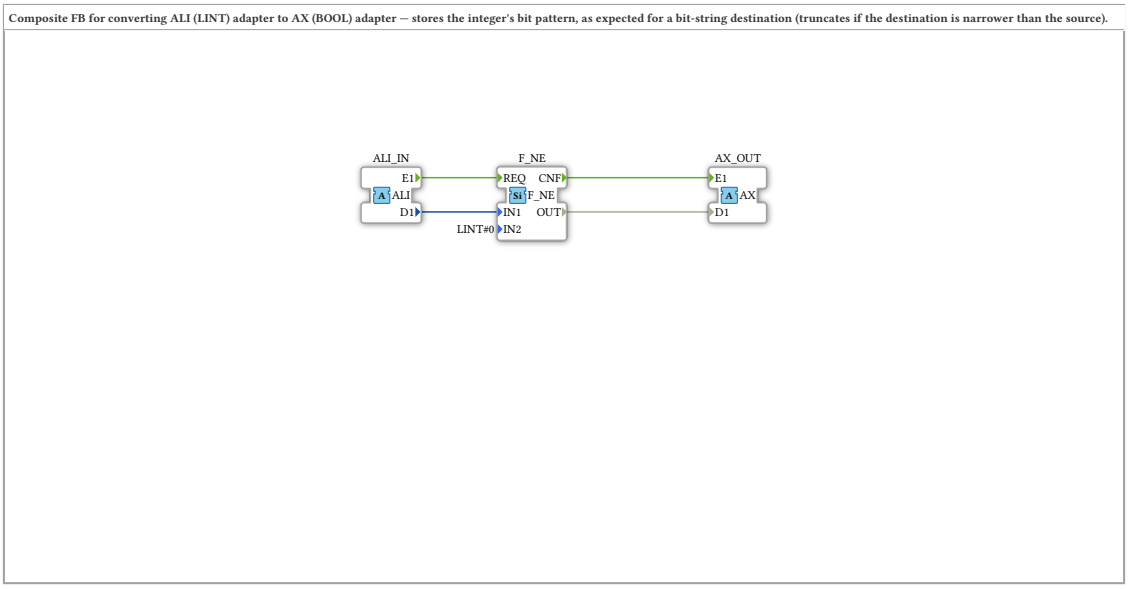
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALI_TO_AX



LINT Adapter Input – *ALI*

AX – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **ALI_TO_AX** is a composite function block that converts an adapter of type **ALI** (containing a LINT value) into an adapter of type **AX** (containing a BOOL value). The conversion follows the rule: A LINT value other than 0 is interpreted as a BOOL value **TRUE**, and a value equal to 0 as **FALSE**.

The function block serves as a bidirectional bridge between adapters based on different data types.

Interface Structure

Event Inputs

- **ALI_IN.E1** – Input event from the ALI adapter. Triggers processing.

Event Outputs

- **AX_OUT.E1** – Output event to the AX adapter. Sent after successful processing.

Data Inputs

(No direct data inputs. Data is provided via the ALI adapter.)

- **ALI_IN.D1** – LINT input value via the adapter.

Data Outputs

(No direct data outputs. Data is output via the AX adapter.)

- **AX_OUT.D1** – BOOL output value via the adapter.

Adapters

Adapter	Direction	Type	Description
ALI_IN	Socket (Input)	`adapter::types::unidirectional::ALI`	Input adapter with a LINT value.
AX_OUT	Plug (Output)	`adapter::types::unidirectional::AX`	Output adapter with a BOOL value.

Functionality

The function block internally uses the comparison function block **F_NE** (“Not Equal”) from the IEC 61131 library. The process is as follows:

1. A LINT value is received via socket `ALI_IN`.
2. The event `ALI_IN.E1` triggers the function block `F_NE` via its `REQ` input.

3. In F_NE, the input value (IN1) is compared with the constant value LINT#0 (IN2).
4. The result OUT is a BOOL:

- **TRUE** if IN1 ≠ 0
- **FALSE** if IN1 = 0

1. After the comparison is complete, F_NE sends a CNF event, which triggers the output adapter AX_OUT via its E1.
2. Simultaneously, the BOOL value is output as AX_OUT.D1.

Thus, any LINT value (including negative numbers) is converted into a binary signal.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Pure Logic Conversion**: No scaling or thresholding takes place. Any non-zero value is interpreted as TRUE.
- **Composite Function Block Use**: The function block encapsulates the adapter conversion in a reusable unit and allows for easy integration into larger networks.
- **Event-Driven**: Processing only occurs upon an incoming event (ALI_IN.E1). The output is only updated after successful processing.
- **License**: The function block is licensed under the **Eclipse Public License 2.0** (EPL-2.0).

State Overview

Since this is a composite function block without its own state machine, the state logic results from the internal chaining:

1. **Waiting for event**: The function block is inactive until ALI_IN.E1 arrives.
2. **Comparison in progress**: After receiving the event, the F_NE function block is executed.
3. **Output result**: After the comparison is complete, the result is set to AX_OUT.D1, and AX_OUT.E1 is triggered.

There are no loops, time delays, or error states – the function is deterministic and error-free.

Application Scenarios

- **Signal Conversion in Control Systems**: When a LINT-based protocol (e.g., counter readings, encoders) needs to be converted into simple binary information (e.g., "active," "presence").
- **Adapter Bridges**: Used in systems that employ heterogeneous adapter types to enable communication between components with different data types.
- **Test Environments**: Fast conversion of numerical values into Boolean signals for debugging or simulation.

Comparison with Similar Function Blocks

- **Direct Compare Function Block**: Instead of F_NE, a F_EQ (equal) followed by negation could also be used – functionally identical.
- **Threshold Function Blocks (e.g., Greater_than)**: Would require an explicit threshold; ALI_TO_AX makes a hard zero decision.
- **Type conversions (e.g., LINT_TO_BOOL)**: Some libraries offer primitive converters – however, these usually do not operate at the adapter level. ALI_TO_AX is specifically designed for adapter interfaces.

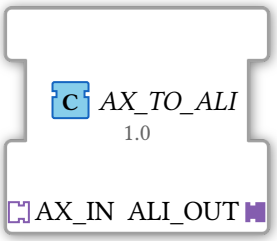
Conclusion

The **ALI_TO_AX** function block provides a simple and reliable method for translating a LINT adapter value to a BOOL adapter value.

Thanks to its clear logic structure and event-driven architecture, it is suitable for both production use and rapid prototyping.

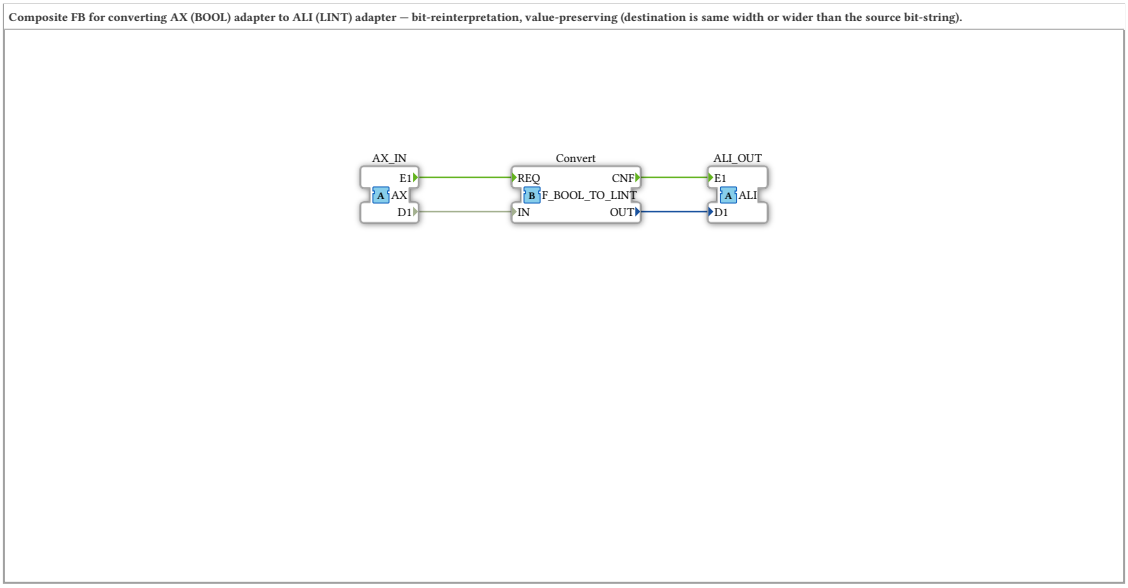
The use of the composite concept promotes modularity and reusability in IEC 61499-based control applications.

AX_TO_ALI



BOOL Adapter Input – *AX*

ALI – LINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AX_TO_ALI** is a composite function block (FB) that converts a unidirectional AX adapter (BOOL) into a unidirectional ALI adapter (LINT). It was developed to convert BOOL values received via an AX socket into LINT values and output them via an ALI plug. The block internally uses the IEC 61131 conversion function `F_BOOL_TO_LINT`.

Interface Structure

The AX_TO_ALI has no direct event or data inputs/outputs. All communication takes place exclusively via adapters.

Event Inputs

No separate event inputs. Events are received indirectly via the AX adapter socket.

Event Outputs

No separate event outputs. Events are output indirectly via the ALI adapter plug.

Data Inputs

No separate data inputs. Data is received indirectly via the AX adapter socket.

Data Outputs

No separate data outputs. Data is output indirectly via the ALI adapter plug.

Adapter

Adapter	Direction	Type	Description

AX_IN	Socket	adapter::types::unidirectional::AX	BOOL adapter input: Receives a BOOL signal via the unidirectional AX adapter.
ALI_OUT	Plug	adapter::types::unidirectional::ALI	LINT adapter output: Outputs the converted LINT value via the unidirectional ALI adapter.

Functionality

The module operates in an event-driven manner and performs the following steps upon each event received via the AX adapter socket:

1. **Event reception**: An event `E1` from the AX adapter is transmitted to the internal conversion module `F_BOOL_TO_LINT` (event `REQ`).
2. **Data Transfer**: The current BOOL value `D1` of the AX adapter is passed to the input `IN` of the conversion function.
3. **Conversion**: The internal function block converts the BOOL value into a LINT value (`FALSE` ⟶ `0`, `TRUE` ⟶ `1`).
4. **Data Output**: The converted LINT result is output via data output `OUT` to the data input `D1` of the ALI adapter.
5. **Event Output**: After successful conversion, an acknowledgment event `CNF` is sent to the ALI adapter output `E1`.

This enables synchronous, event-driven conversion from BOOL to LINT.

Technical Specifications

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **License**: This function block is released under the **Eclipse Public License 2.0 (EPL-2.0)**.

Copyright © 2026 HR Agrartechnik GmbH.

- **Author & Version**: Developed by Franz Höpfinger, Version 1.0, February 17, 2026.
- **Adapter Package**: Uses adapters from the `adapter::types::unidirectional` package, which are designed for one-way data and event transmission.
- **Internal Implementation**: The conversion is performed using the IEC 61131 function block `F_BOOL_TO_LINT` from the library `iec61131::conversion`. This function block operates deterministically and does not require intermediate storage.
- **Hash Attribute**: The function block contains an empty type hash attribute (`eclipse4diac::core::TypeHash`).

State Overview

The AX_TO_ALI function block does not have its own state machine. It forwards events and data directly to the internal conversion function block and outputs the result without delay. Since the internal function block `F_BOOL_TO_LINT` is pure function logic without state management, the entire composite function block behaves stateless. After each event at the AX adapter, the corresponding output is immediately generated at the ALI adapter.

Application Scenarios

- **Type Conversion in Adapter Chains**: When a BOOL value arrives in an IEC 61499 application via a unidirectional AX adapter, but subsequent function blocks expect a LINT value (e.g., for counter functions or analog value processing).
- **Interface Adaptation**: Connecting binary sensors (BOOL) to control logic that works with integer LINT values.
- **Event-Driven Data Conversion**: Simple and fast conversion without intermediate storage, ideal for time-critical paths.

Comparison with Similar Function Blocks

The library `iec61131::conversion` contains numerous function blocks for type conversion, e.g.,... Examples include `F_BOOL_TO_DINT`, `F_BOOL_TO_INT`, and `F_BOOL_TO_USINT`. The AX_TO_ALI differs in that it

is already fully embedded in an adapter package and enables conversion between the unidirectional AX and ALI adapters. This eliminates the need for developers to manually wire sockets/plugs with a separate conversion block and provides a clean, reusable encapsulation.

Conclusion

The AX_TO_ALI is a compact, license-compliant composite function block for reliably converting a BOOL value to a LINT value via unidirectional adapters. Its simple event handling and use of an established IEC function block make it a robust solution for typical conversion tasks in IEC 61499 systems. The clean separation of input and output via adapters facilitates integration into modular control architectures.

Related topic subpages on ms-muc-docs.de

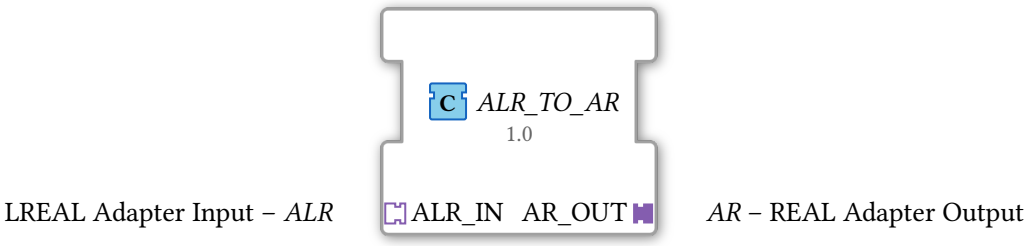
- Eclipse 4diac IDE & color reference on ms-muc-docs.de

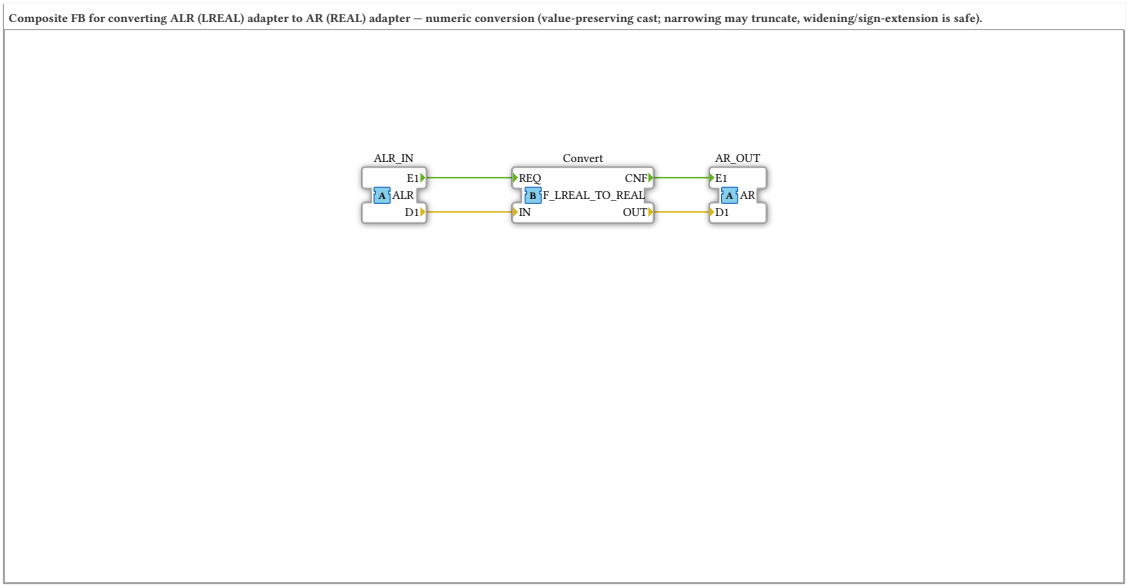
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALR_TO_AR





No image available

• – ○ • – ○ • – ○ •

Introduction

The **ALR_TO_AR** function block is a composite block that converts an LREAL adapter (ALR) into a REAL adapter (AR). It serves as a link between control components operating with different floating-point accuracies and enables clean, event-driven data conversion within an IEC 61499 network.

Interface Structure

Event Inputs

- **E1** (via socket `ALR_IN`): Starts the conversion of the incoming LREAL value.

Event Outputs

- **E1** (via plug `AR_OUT`): Signals that the converted REAL value at the output is valid.

Data Inputs

- **D1** (via socket `ALR_IN`): LREAL input value (64-bit floating point).

Data Outputs

- **D1** (via plug `AR_OUT`): REAL output value (32-bit floating point).

Adapters

- **ALR_IN** (socket): Adapter type `adapter::types::unidirectional::ALR` – accepts LREAL data.
- **AR_OUT** (Plug): Adapter type `adapter::types::unidirectional::AR` – provides REAL data.

Functionality

1. An event at input **E1** (of the ALR_IN socket) triggers the internal block `F_LREAL_TO_REAL`.
2. Simultaneously, the LREAL value from **D1** (ALR_IN) is passed directly to the conversion block.
3. The conversion block converts the 64-bit value into a 32-bit REAL value.
4. After the conversion is complete, an event is generated at output **E1** (AR_OUT), and the REAL value is passed to **D1** (AR_OUT).

Transmission is strictly sequential: first the event, then the data flow – the internal connection ensures reliable synchronization.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block**: The ALR_TO_AR encapsulates the conversion logic and simplifies reuse in different projects.
- **IEC 61131 Function**: The actual type conversion is based on the standardized library function `F_LREAL_TO_REAL`.
- **Adapter Types**: Both adapters are unidirectional and allow a clear separation of input and output sides.
- **No State Machine**: The block operates purely event-driven without an internal state memory.

State Overview

Since this is a composite block without its own state machine, there are no explicit states. The internal process is deterministic:

- **Ready / Idle** – waits for an input event.
- **Conversion active** – from event input until the output event is generated (typically takes one cycle).

Latency is minimal and determined by the execution order of the underlying IEC 61499 runtime system.

Application Scenarios

- **Sensor Integration**: A field device provides LREAL measured values, but downstream logic expects REAL values (e.g., PLC communication).
- **Adapter Coupling**: Connecting two modules that use different adapter interfaces without requiring the developer to manually program the conversion.
- **Data Consistency**: Enforces a clean separation of 64-bit and 32-bit data paths and avoids implicit type conversions.

Comparison with Similar Components

- **F_LREAL_TO_REAL** (direct): Offers the same conversion but requires separate event and data connections. ALR_TO_AR encapsulates these connections and provides a unified adapter interface.

- **LINT_TO_INT** (adapter variant): Analog logic for integer types with different bit widths. The basic structure is identical but tailored to different data types.
- **Custom Composite Converters**: ALR_TO_AR reduces the overhead of recurring adapter conversions and improves network readability.

Conclusion

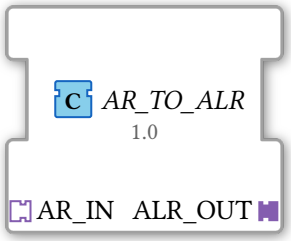
The **ALR_TO_AR** is a simple yet effective composite component that bridges the gap between LREAL and REAL adapter interfaces. It offers clean, event-driven conversion without unnecessary overhead and integrates easily into existing 4diac projects. Thanks to the use of standard library functions, the conversion remains performant and reliable.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

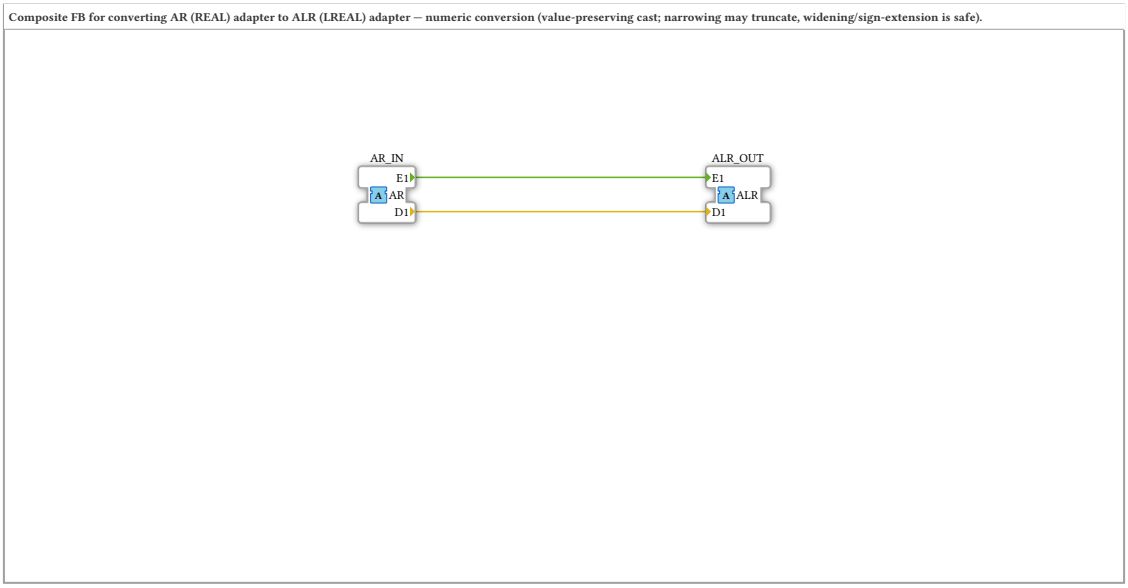
]

AR_TO_ALR



REAL Adapter Input – *AR*

ALR – LREAL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

Composite function block for converting an AR adapter (REAL) to an ALR adapter (LREAL). It is used to adapt data formats in IEC 61499-2 environments and is licensed under the Eclipse Public License 2.0.

Interface Structure

Event Inputs

Name	Description
E1 (from AR_IN)	Event to trigger data transfer

Event Outputs

Name	Description
E1 (from ALR_OUT)	Event output after successful conversion

Data Inputs

Name	Data Type	Description
D1 (from AR_IN)	REAL	32-bit floating-point number

Data Outputs

Name	Data Type	Description
D1 (from ALR_OUT)	LREAL	64-bit floating-point number

Adapters

- **ALR_OUT** (Plug): Type `adapter::types::unidirectional::ALR` – LREAL output adapter
- **AR_IN** (Socket): Type `adapter::types::unidirectional::AR` – REAL input adapter

Functionality

This function block connects the adapter interfaces directly. An incoming event at `AR_IN.E1` is immediately forwarded to `ALR_OUT.E1`. Simultaneously, the data value `AR_IN.D1` (REAL) is copied to `ALR_OUT.D1` (LREAL), with the runtime environment performing an implicit type conversion. There is no state machine; the function is purely combinatorial.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- Composite function block without its own logic (no ECC)
- Use of unidirectional adapters according to IEC 61499-2
- Automatic type conversion via the 4diac runtime (e.g., 4diac-ide)
- Package: `adapter::conversion::unidirectional`

State Overview

No states present – the function block does not have an execution state machine.

Application Scenarios

- Connecting a REAL-providing sensor to a component that expects LREAL
- Migrating existing REAL adapters to LREAL with higher accuracy
- Interface adaptation in heterogeneous automation systems

Comparison with Similar Function Blocks

Similar function blocks exist for other type conversions (e.g., `LREAL_TO_REAL`, `INT_TO_REAL`). This block is specifically designed for the unidirectional adapters `AR` and `ALR` and requires no custom implementation logic.

Conclusion

The `AR_TO_ALR` block enables simple and efficient conversion of REAL to LREAL adapter data. Implemented as a composite function block, it utilizes the direct connection of the adapter interfaces, thus offering high performance with minimal latency.

–

🌐 Related topic subpages on ms-muc-docs.de

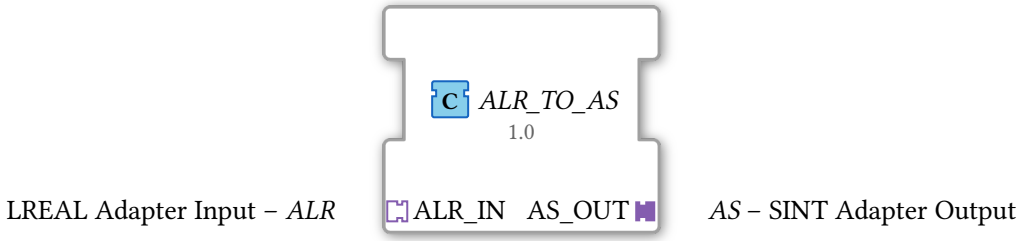
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

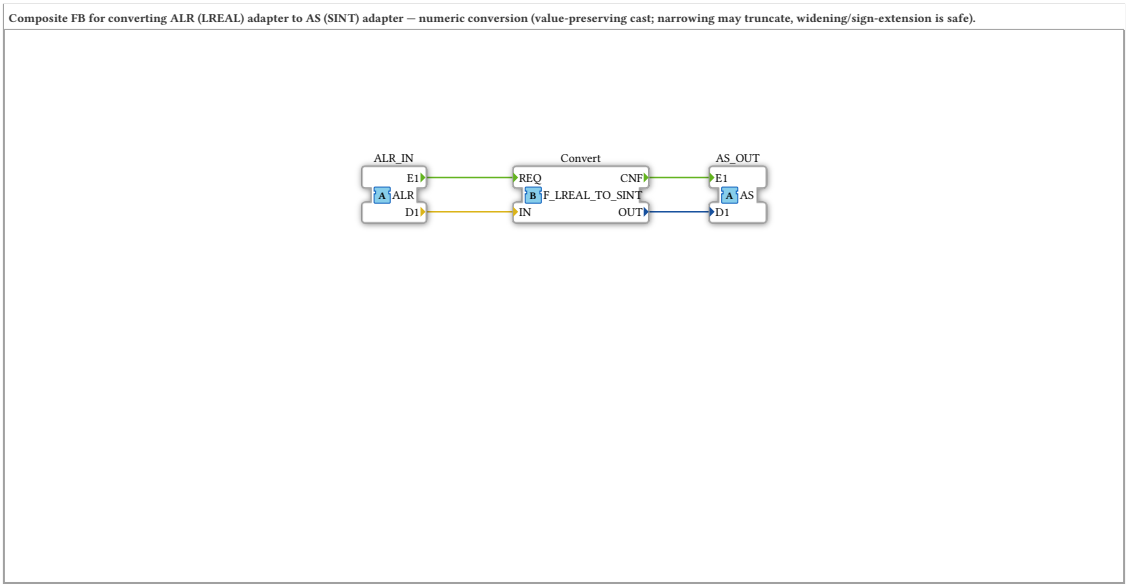
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALR_TO_AS





• – ○ • – ○ • – ○ •

Introduction

The **ALR_TO_AS** function block is a composite function block that converts an **LREAL adapter** (ALR) into a **SINT adapter** (AS). It encapsulates the conversion of a floating-point value (LREAL) into a signed 8-bit integer value (SINT) and provides the result data via the SINT adapter. This function block is typically used in automation technology when data needs to be transferred between different adapter interfaces.

Interface Structure

Event Inputs

The function block does not have separate event inputs. The conversion is controlled by the event of the connected **ALR Adapter Socket** (`ALR_IN.E1`).

Event Outputs

The function block does not have separate event outputs. The completion of the conversion is signaled by the event of the **AS Adapter Plug** (`AS_OUT.E1`).

Data Inputs

The function block does not have separate data inputs. The input data (LREAL value) is provided via the **ALR Adapter** (`ALR_IN.D1`).

Data Outputs

The function block does not have separate data outputs. The output data (SINT value) is provided via the **AS adapter** (`AS_OUT.D1`).

Adapter

Adapter	Direction	Type	Description
`ALR_IN`	Socket	`adapter::types::unidirectional::ALR`	LREAL adapter input: provides the LREAL value to be converted and a trigger event (`E1`).
`AS_OUT`	Plug	`adapter::types::unidirectional::AS`	SINT adapter output: provides the converted SINT value and an acknowledgment event (`E1`).

Functionality

The function block is implemented as a composite FB. Internally, it contains a function block `F_LREAL_TO_SINT` from the IEC 61131 library. The connections within the network cause:

1. **Event-driven activation**: An incoming event at the adapter `ALR_IN` (`E1`) triggers the conversion FB (`Convert.REQ`).
2. **Data conversion**: The LREAL value of `ALR_IN.D1` is forwarded to the input `Convert.IN`. The function block (FB) converts this value into a SINT value (rounding according to IEC 61131 rules).
3. **Result Output**: After successful conversion, `Convert.CNF` signals an event to the output adapter `AS_OUT` (`E1`), and the converted SINT value is passed to `AS_OUT.D1`.

This enables synchronous, event-driven conversion of an LREAL value to a SINT value via adapter interfaces.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Structure**: The function block completely encapsulates the conversion logic in a network, making the implementation transparent and easily extensible.
- **Adapter-based**: Instead of individual event and data ports, standardized unidirectional adapters (`ALR` and `AS`) are used. This simplifies integration into existing adapter-based architectures.
- **Reuse**: The internal function block `F_LREAL_TO_SINT` is a proven IEC function block and ensures correct conversion to the standard.
- **No dedicated state machine**: Since this is a composite function block, there is no independent state machine; control is achieved solely through the connected adapter events.

State Overview

The function block does not have an explicit state machine. Its behavior is determined by the internal function block `F_LREAL_TO_SINT` and the event connections. This internal function block (FB) executes a simple **IDLE → CONVERTING → DONE** cycle, controlled by the events `REQ` and `CNF`. For the user, the block is therefore transparent and usable as an **event-driven converter**.

Application Scenarios

- **Data Bridge Between Systems**: When a control module provides LREAL values via a `ALR` adapter, but a downstream module expects SINT values via a `AS` adapter.
- **Uniform Adapter Interfaces**: Simplifying system design by outsourcing conversions to standalone function blocks.
- **PLC Connection**: Connecting sensors or actuators that operate with LREAL accuracy to a SINT-based bus communication.

Comparison with Similar Components

Compared to directly wiring a `F_LREAL_TO_SINT` component with individual event/data ports, the `ALR_TO_AS` component offers **a higher level of abstraction through adapters**. This facilitates the ex-

change of data between components that are already designed for adapter interfaces. Similar components could be `ALR_TO_ADI` (conversion to INT) or `ALR_TO_UDI` (conversion to UDINT), which are structured analogously but use different target data types.

Conclusion

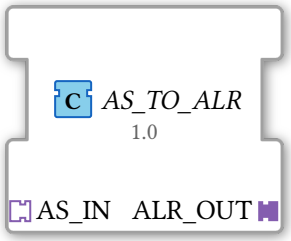
The `ALR_TO_AS` function block is a clean, adapter-based solution for converting LREAL to SINT values in the 4diac IDE. Its composite structure and use of standardized IEC conversion blocks ensure reliable, event-driven data conversion and facilitate integration into modular automation systems.

🌐 Related topic subpages on ms-muc-docs.de

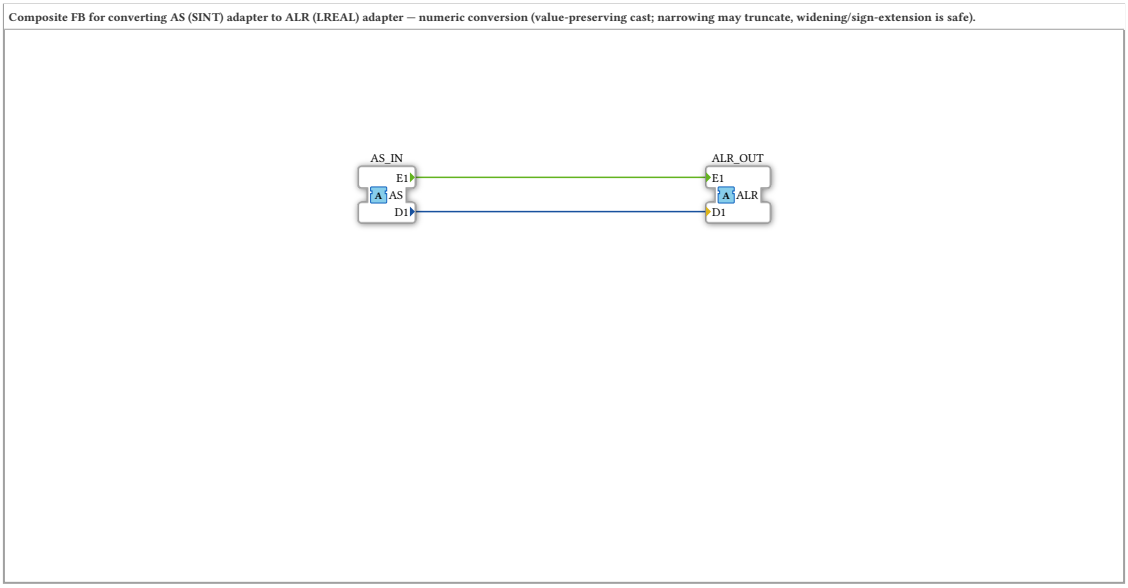
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AS_TO_ALR

SINT Adapter Input – *AS*



ALR – LREAL Adapter Output



No image available.

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_TO_ALR** is a composite function block (FB) that converts an **AS adapter** (data type SINT) into an **ALR adapter** (data type LREAL). It serves as a simple interface conversion between these two unidirectional adapter types and is located in the package `adapter::conversion::unidirectional`.

Interface Structure

Event Inputs

Name	Comment
E1	Event from AS adapter (Socket AS_IN)

Event Outputs

Name	Comment
E1	Event for the ALR adapter (Plug ALR_OUT)

Data Inputs

Name	Data Type	Comment
D1	SINT	Data value from the AS adapter (Socket AS_IN)

Data Outputs

Name	Data Type	Comment
D1	LREAL	Data value for the ALR adapter (Plug ALR_OUT)

Adapters

Direction	Name	Type	Comment
Socket	AS_IN	`adapter::types::unidirectional::AS`	Input Adapter (SINT)
Plug	ALR_OUT	`adapter::types::unidirectional::ALR`	Output Adapter (LREAL)

Functionality

The **AS_TO_ALR** block forwards incoming events and data values directly to the output side without further processing.

- The event **E1** at socket `AS_IN` is passed to plug `ALR_OUT` without delay.
- The data value **D1** of type SINT is forwarded to the output of the same name. The conversion from SINT to LREAL is performed implicitly by the runtime environment – the function block itself does not execute any arithmetic operations.

The function block has no state machine (ECC) and no internal algorithms; its entire functionality is implemented through the included connections.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Structure**: The function block consists exclusively of an internal network (FBNetwork) with direct event and data connections between the adapters.
- **Implicit Type Conversion**: Data conversion from SINT to LREAL is handled by the IEC 61499 runtime environment. Explicit type conversion is not required.
- **Packaging**: The function block is organized in the package `adapter::conversion::unidirectional`, providing a clean separation between adapter and converter logic.

State Overview

The **AS_TO_ALR** function block does not have its own state machine (no ECC diagram). Its behavior is defined solely by the direct connection of its inputs and outputs. There is no defined start or end state.

Application Scenarios

- **Sensor Connection**: A sensor provides data in SINT format (e.g., an 8-bit counter) that needs to be integrated into an environment with an LREAL interface (e.g., an analog adapter) via an AS adapter.
- **Protocol Conversion**: In heterogeneous automation systems where different components expect adapters of various types, **AS_TO_ALR** serves as a simple converter.
- **Interface Migration**: When switching from SINT-based to LREAL-based adapters, this function block can be used as a temporary bridge.

Comparison with Similar Components

- **AS_TO_AR**: Converts an AS adapter (SINT) to an AR adapter (REAL) – similar structure, but with a smaller value range (REAL instead of LREAL).
- **ALR_TO_AS**: The inverse conversion (LREAL ⟶ SINT), which may require explicit range checking or rounding.
- **AS_TO_ALR** is characterized by the larger value range of its output (LREAL), enabling lossless transmission of SINT values.

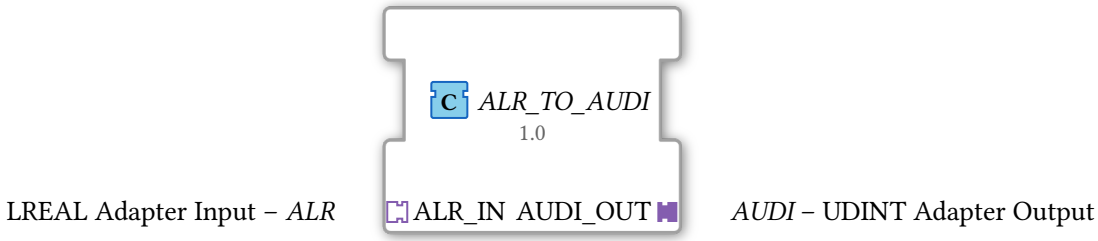
Conclusion

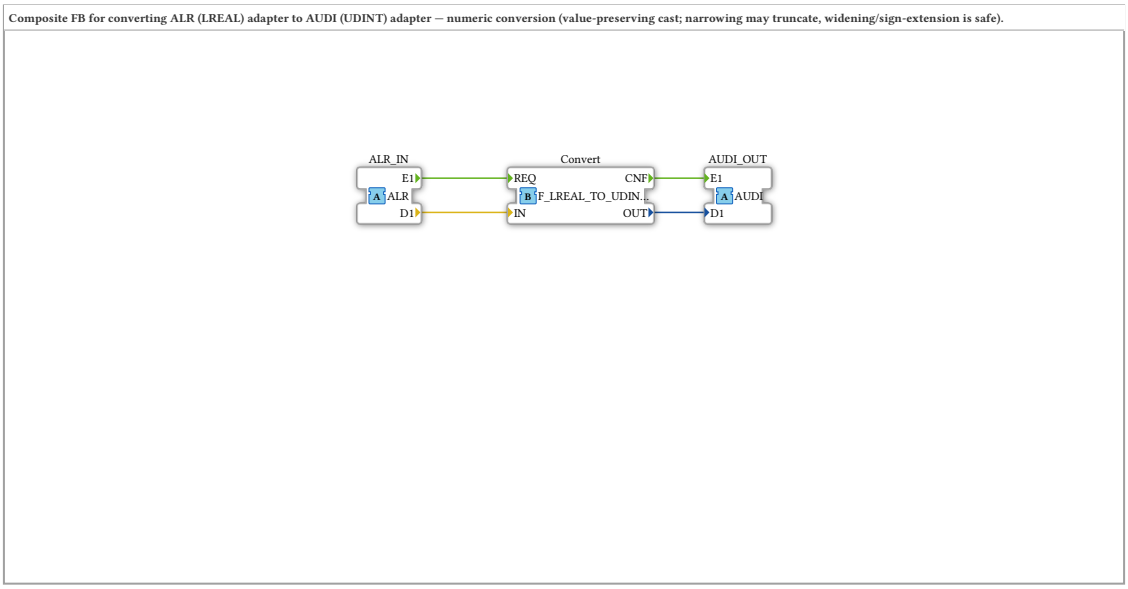
The **AS_TO_ALR** is a minimal yet useful converter component for easily connecting AS adapters to ALR adapters. Its composite architecture and implicit type conversion make it easy to understand and efficient to use.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALR_TO_AUDI





• – ○ • – ○ • – ○ •

Introduction

The ALR_TO_AUDI function block is a composite function block (FB) that converts an **ALR adapter** (LREAL) into an **AUDI adapter** (UDINT). It serves as an interface between an analog LREAL source and a digital UDINT sink in unidirectional communication paths.

Interface Structure

Event Inputs

- `ALR_IN.E1` – Event input of the ALR adapter; triggers the conversion.

Event Outputs

- `AUDI_OUT.E1` – Event output of the AUDI adapter; signals the completion of the conversion.

Data Inputs

- `ALR_IN.D1` – LREAL input value that is converted to a UDINT value.

Data Outputs

- `AUDI_OUT.D1` – UDINT output value, result of the conversion.

Adapters

- **Socket** `ALR_IN` of type `adapter::types::unidirectional::ALR` – Input interface for LREAL values with event.
- **Plug** `AUDI_OUT` of type `adapter::types::unidirectional::AUDI` – Output interface for UDINT values with event.

Functionality

The function block internally uses the conversion function block `iec61131::conversion::F_LREAL_TO_UDINT`. The process is as follows:

1. An event on `ALR_IN.E1` is forwarded to input `REQ` of the internal converter.
2. Simultaneously, the data value from `ALR_IN.D1` is transferred to input `IN` of the converter.
3. After successful conversion, the converter sends an acknowledgment event (`CNF`) to `AUDI_OUT.E1`.
4. The converted UDINT value is at output `OUT` of the converter and is passed on to `AUDI_OUT.D1`.

Thus, the conversion is strictly event-driven and data-synchronous.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-Based Encapsulation** – Data and events are exchanged via standardized adapters of type `unidirectional`, enabling a clean separation of interfaces and implementation.
- **Unidirectional Communication** – The function block supports only one direction of data flow (LREAL ⟶ UDINT). Feedback or error status is not provided.
- **Type Conversion** – The conversion from LREAL (64-bit floating-point) to UDINT (32-bit unsigned integer) can lead to rounding or overflow losses, but is performed according to the IEC 61131 standard.

State Overview

The ALR_TO_AUDI does not have its own internal state machine. Its behavior is entirely determined by the included function block `F_LREAL_TO_UDINT`, which operates as a simple, event-driven conversion instance.

Application Scenarios

- **Analog-to-Digital Conversion in Agricultural Control** – Conversion of LREAL sensor values (e.g., pressure, fill level) into integer UDINT values for displays or digital control loops.
- **Interface Adaptation** – Connecting components that rely on different adapter types (e.g., ALR from a measurement module and AUDI in a controller).
- **Protocol Conversion** – Simple conversion within adapter chains without having to implement custom logic.

Comparison with Similar Components

- **LREAL_TO_DINT** – Converts to a signed 32-bit integer. ALR_TO_AUDI, on the other hand, uses the unsigned UDINT variant.
- **REAL_TO_UDINT** – Works with 32-bit floating-point instead of 64-bit LREAL, resulting in correspondingly lower accuracy.
- **LREAL_TO_LWORD** – Converts to a 64-bit integer, but using a different adapter type (LWORD instead of UDINT).

The ALR_TO_AUDI is specifically designed for unidirectional adapter coupling with the aforementioned types.

Conclusion

The ALR_TO_AUDI is a specialized yet simple component for converting LREAL to UDINT values via adapter interfaces. Thanks to its clear structure and the use of a standardized IEC converter, it can be easily integrated into existing 4diac projects and contributes to the interoperability of different adapter types.

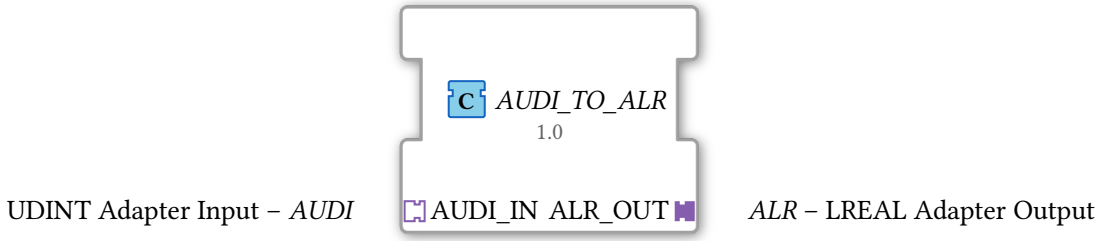
–

🌐 Related topic subpages on ms-muc-docs.de

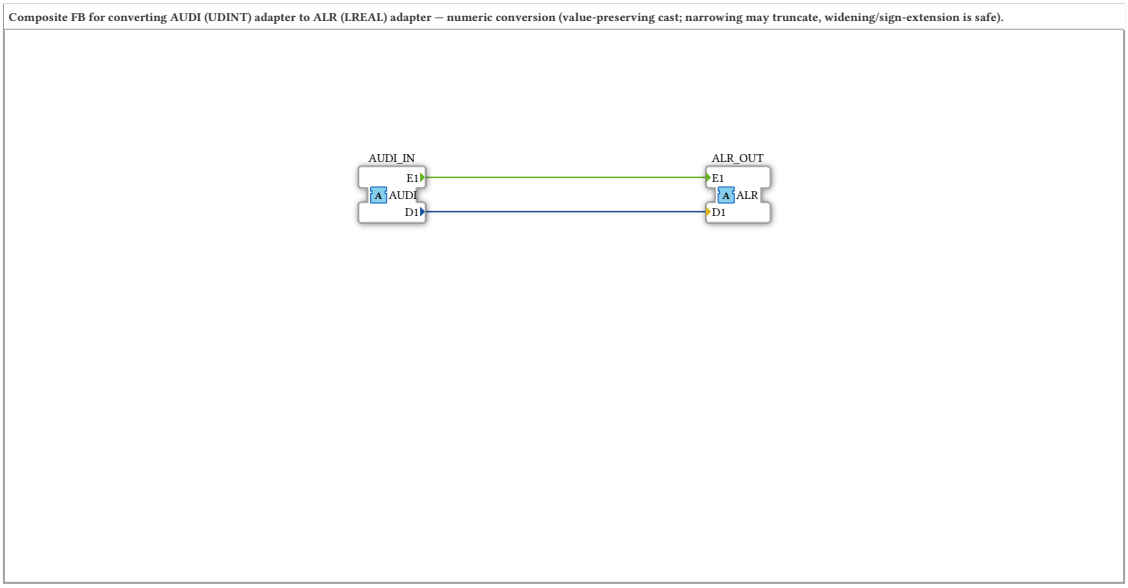
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUDI_TO_ALR



UDINT Adapter Input – *AUDI* *ALR* – LREAL Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUDI_TO_ALR** function block is a composite module that converts a unidirectional adapter of type **AUDI** (data type `UDINT`) into an adapter of type **ALR** (data type `LREAL`). It serves as a simple link between two different adapter interfaces, thus enabling the integration of components based on different data formats.

Interface Structure

Event Inputs

Name	Comment
`AUDI_IN.E1`	Event input of the AUDI adapter (passed through to the ALR output)

Event Outputs

Name	Comment
ALR_OUT.E1	Event output of the ALR adapter (inherited from the AUDI input)

Data Inputs

Name	Data Type	Comment
AUDI_IN.D1	UDINT	Data input of the AUDI adapter (passed directly as LREAL)

Data Outputs

Name	Data Type	Comment
ALR_OUT.D1	LREAL	Data output of the ALR adapter (value from the AUDI input, type-converted)

Adapter

Direction	Name	Type	Comment
Socket (Input)	AUDI_IN	adapter::types::unidirectional::AUDI	Receives the UDINT data and the associated event
Plug (Output)	ALR_OUT	adapter::types::unidirectional::ALR	Provides the converted LREAL data and the event

Functionality

The function block forwards the incoming event E1 from socket AUDI_IN directly to plug ALR_OUT. Simultaneously, the data value D1 (type UDINT) is transferred unchanged to the output data channel D1 (type LREAL). The actual type conversion from UDINT to LREAL takes place within the adapter interfaces used; the composite block itself does not perform any arithmetic or logical transformation, but merely serves as a wiring aid.

...

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Pure Adapter Passthrough:** The module contains no logic or state machines of its own – it simply connects the corresponding channels of the two adapters.
- **Unidirectional Communication:** Both the AUDI and ALR adapters support only one direction of data flow, which simplifies their use in simple sensor/actuator interfaces or when coupling different protocols.
- **No Type Conversion in the Module:** The conversion from UDINT to LREAL is handled by the adapter infrastructure. The developer must ensure that the adapter implementations used provide the necessary conversion.

State Overview

As a composite module, AUDI_TO_ALR does not have its own state machine. Its behavior is purely combinatorial: An incoming event is immediately passed on to the output – without delay or condition.

Application Scenarios

- **Integration of UDINT-based sensors** (e.g., rotary encoders with 32-bit counters) into an LREAL-processing control system.
- **Adapter bridge** between different libraries or components based on different data types but using uniform adapter interfaces.

- **Prototype development** and rapid wiring of adapters without having to create separate conversion logic.

Comparison with Similar Function Blocks

In the 4diac ecosystem, simple conversion function blocks such as `INT_TO_REAL` or `UDINT_TO_LREAL` exist, which are usually implemented as Basic or Service FBs. The present Composite FB differs in that it does not convert the data value itself, but rather connects two complete adapters. It is therefore specifically designed for environments that rely on adapter interfaces and not for simple data flow conversion.

Conclusion

AUDI_TO_ALR` is a minimal yet useful composite block for connecting two unidirectional adapters with different data types. It reduces wiring complexity and makes the type conversion explicitly visible. This block should always be used when a clean separation between adapter types and simple event and data transmission are required.

🌐 Related topic subpages on ms-muc-docs.de

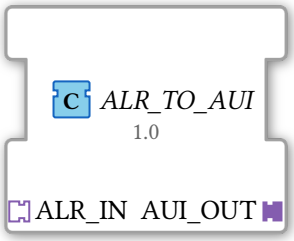
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

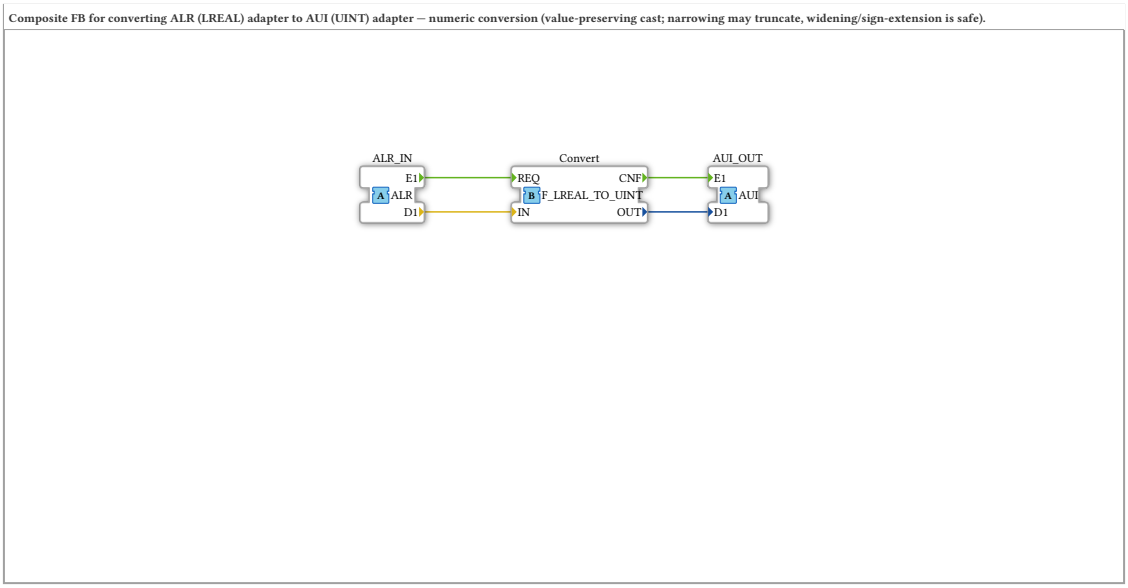
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

ALR_TO_AUI



LREAL Adapter Input – *ALR* *AUI* – UINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_TO_AUI** is a composite function block (FB) that converts an LREAL value, received via an **ALR** adapter (unidirectional), into a UINT value and makes it available via an **AUI** adapter (unidirectional). The conversion is performed using the internal IEC 61131 function block

F_LREAL_TO_UINT. This block encapsulates the adapter conversion and provides a clean interface for data exchange between the two adapter types.

Interface Structure

Event Inputs

The FB does not have its own event inputs. Events are received indirectly via the **ALR_IN** adapter:

- **ALR_IN.E1** – Event to trigger the conversion.

Event Outputs

The FB does not have its own event outputs. Events are sent via the **AUI_OUT** adapter:

- **AUI_OUT.E1** – Confirmation event after successful conversion.

Data Inputs

The data is also provided via the **ALR_IN** adapter:

- **ALR_IN.D1** – Input value of type **LREAL** (according to the ALR adapter).

Data Outputs

The converted data is output via the **AUI_OUT** adapter:

- **AUI_OUT.D1** – Output value of type **UINT** (according to the AUI adapter).

Adapters

Name	Type	Direction	Description
ALR_IN	`adapter::types::unidirectional::ALR`	Socket (Input)	Returns the LREAL value and the trigger event.
AUI_OUT	`adapter::types::unidirectional::AUI`	Plug (Output)	Outputs the converted UINT value and the confirmation event.

Functionality

The function block operates as a pure translation layer between two adapters:

1. An incoming event at socket **ALR_IN.E1** triggers the internal conversion block F_LREAL_TO_UINT via its **REQ** input.
2. Simultaneously, the data value **ALR_IN.D1** is forwarded to the **IN** input of the conversion block.
3. The internal block performs the type conversion from **LREAL** to **UINT** and places the result at its **OUT** output.
4. Once the conversion is complete, the internal block sends an acknowledgment event (**CNF**), which is passed to the plug **AUI_OUT.E1**.
5. Simultaneously, the converted value (**OUT**) is transferred to the data output **AUI_OUT.D1**.

The entire process is event-driven and occurs once per triggering event.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter Encapsulation:** The block combines two different adapter types (ALR and AUI), thus enabling type-safe communication between components that use these adapters.
- **Unidirectional Adapters:** Both the input and output adapters are unidirectional, meaning there is only one direction of data and events (input → output).
- **Internal Conversion:** The actual conversion is performed by the standardized IEC 61131 function block F_LREAL_TO_UINT. This ensures well-defined and platform-independent conversion behavior.
- **No Control:** The composite function block itself does not have its own state machine. Its behavior is entirely determined by the embedded function block.

State Overview

Since this is a composite function block without its own Execution Control Chart (ECC), there is no internal state machine. The process is controlled by the event links:

- **IDLE:** Waiting for an event at socket **ALR_IN.E1**.
- **CONVERT:** After receiving the event, the conversion is triggered; The internal function block iterates through its own state machine (e.g., REQ → CNF).
- **OUTPUT:** After the conversion is complete, the output event **AUI_OUT.E1** and the converted value are output.

Application Scenarios

- **Connecting Sensors/Actuators:** A sensor provides an LREAL value (e.g., temperature, pressure) which is read into a PLC via an ALR adapter. For display or transmission to a system that only processes integers (UINT), the function block is used.
- **Adapter Migration:** In existing systems, adapters need to be converted from ALR to AUI without changing the connected components. The function block acts as an intermediary.
- **Test and Simulation Environments:** Generating UINT values from LREAL sources for testing purposes.

Comparison with Similar Function Blocks

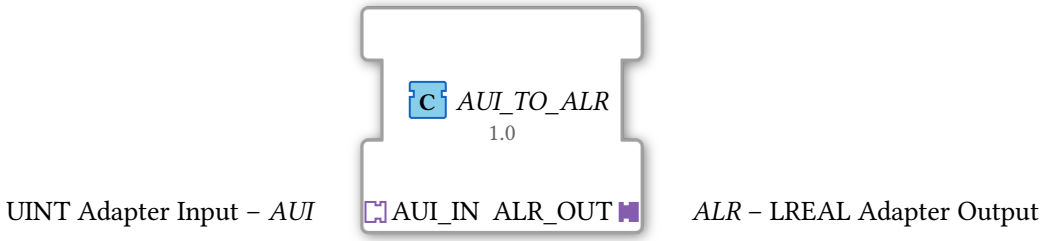
Function Block	Input Type	Output Type	Description
ALR_TO_AUI	ALR (LREAL)	AUI (UINT)	Adapter-based conversion LREAL→UINT.
`F_LREAL_TO_UINT`	LREAL	UINT	Pure data conversion without an adapter.
`LREAL_TO_INT`	LREAL	INT	Conversion to a signed integer.

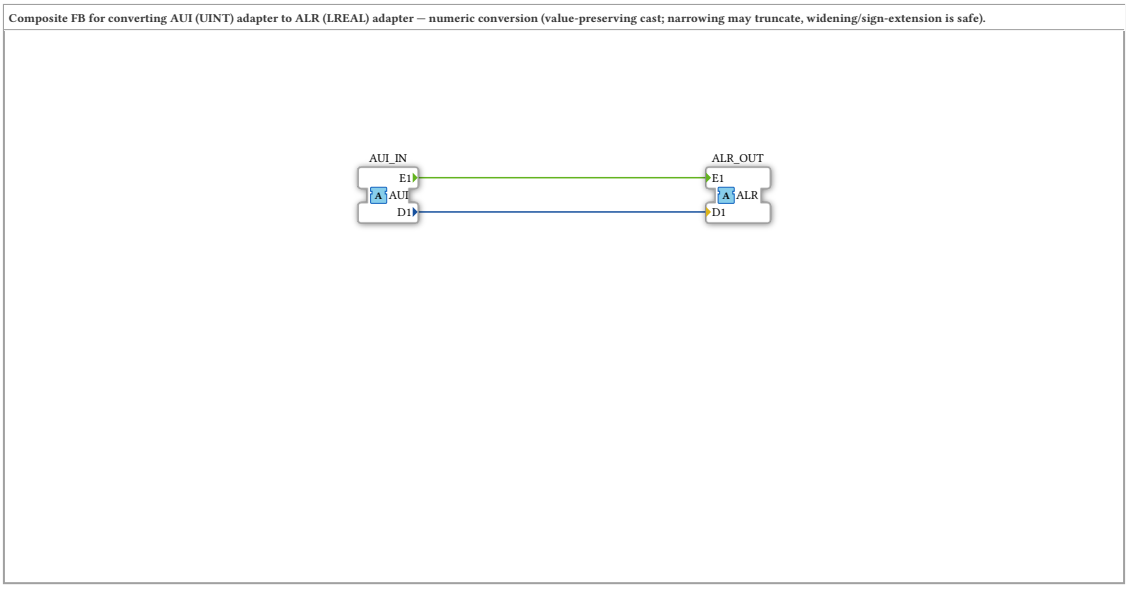
Unlike pure data converters, **ALR_TO_AUI** operates at the adapter level, so both the data and the associated events are exchanged via the adapter interfaces. This simplifies integration into adapter-based architectures.

Conclusion

The **ALR_TO_AUI** function block is a useful tool for converting LREAL values from an ALR adapter to UINT values from an AUI adapter. Encapsulating the conversion in a composite function block increases reusability and clearly defines the system interface. This block is particularly suitable for applications where adapter types need to be harmonized or adapted without altering the internal logic of the connected components.

AUI_TO_ALR





• — ○ • — ○ • — ○ •

Introduction

The AUI_TO_ALR function block is a composite module that enables interface conversion between an AUI adapter (data type UINT) and an ALR adapter (data type LREAL). It acts as a simple pass-through, transmitting both event and data signals from the input to the output adapter.

Interface Structure

Event Inputs

- **Event via Socket AUI_IN (E1)** – Event input from the AUI adapter.

Internally, this is directly connected to the event output of the ALR adapter.

Event Outputs

- **Event via Plug ALR_OUT (E1)** – Event output to the ALR adapter.

Corresponds to the passed-through event from socket AUI_IN.

Data Inputs

- **Data value via socket AUI_IN (D1)** – Data input from the AUI adapter (type UINT).

Internally, it is directly connected to the data output of the ALR adapter.

Data Outputs

- **Data value via plug ALR_OUT (D1)** – Data output to the ALR adapter (type LREAL).

Corresponds to the passed-through data value from socket AUI_IN.

Adapters

- **Socket: AUI_IN** – Source (type `adapter::types::unidirectional::AUI`)

Accepts the input signals (event + UINT data).

- **Plug: ALR_OUT** – Sink (Type `adapter::types::unidirectional::ALR`)

Provides the output signals (event + LREAL data).

Functionality

The FB AUI_TO_ALR functions purely as a coupling point. Events and data are forwarded directly from the socket AUI_IN to the plug ALR_OUT. The actual conversion from UINT to LREAL is not performed within this block, but rather implicitly through the adapter type definitions used (unidirectional protocols). Thus, the composite functions as a standardized interface to connect two different adapter protocols.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Type:** The FB does not have its own Execution Control Chart (ECC), but consists solely of a network connection between its two adapters.
- **Direct Pass-Through:** No internal logic, delays, or state changes – the function block is purely passive.
- **Adapter-Based:** The adapters used are defined as unidirectional interfaces, each carrying one event and one data value.

State Overview

The function block has no states of its own, as it is implemented as a composite without ECC. Its behavior is entirely determined by the connected adapters.

Application Scenarios

- **Protocol Adaptation in Agricultural Technology:** Use in controllers that need to mediate between different sensor or actuator interfaces.
- **Retrofit or Migration:** Enables the use of an ALR-compatible device on an existing AUI connection without having to implement the actual conversion itself.
- **Modular Cabling:** As a simple connecting link in a chain of adapter function blocks, e.g., in a fieldbus or communication hierarchy.

Comparison with Similar Function Blocks

Similar converter function blocks exist for other adapter pairs (e.g., `AUI_TO_AREAL`, `AUI_TO_BOOL`). Unlike these, AUI_TO_ALR does not perform active data conversion but relies on implicit type conversion through the adapter definition. This makes it particularly lightweight and suitable for simple pass-through tasks.

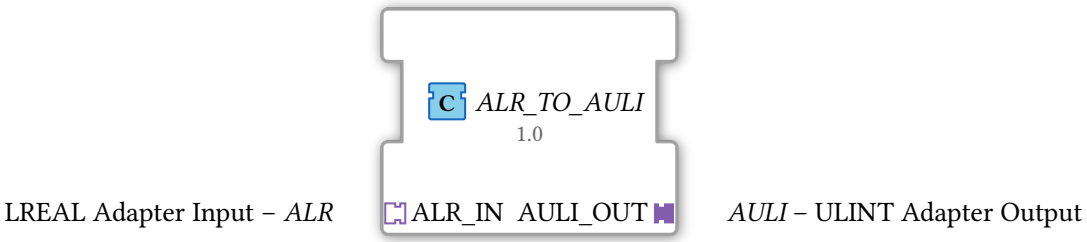
Conclusion

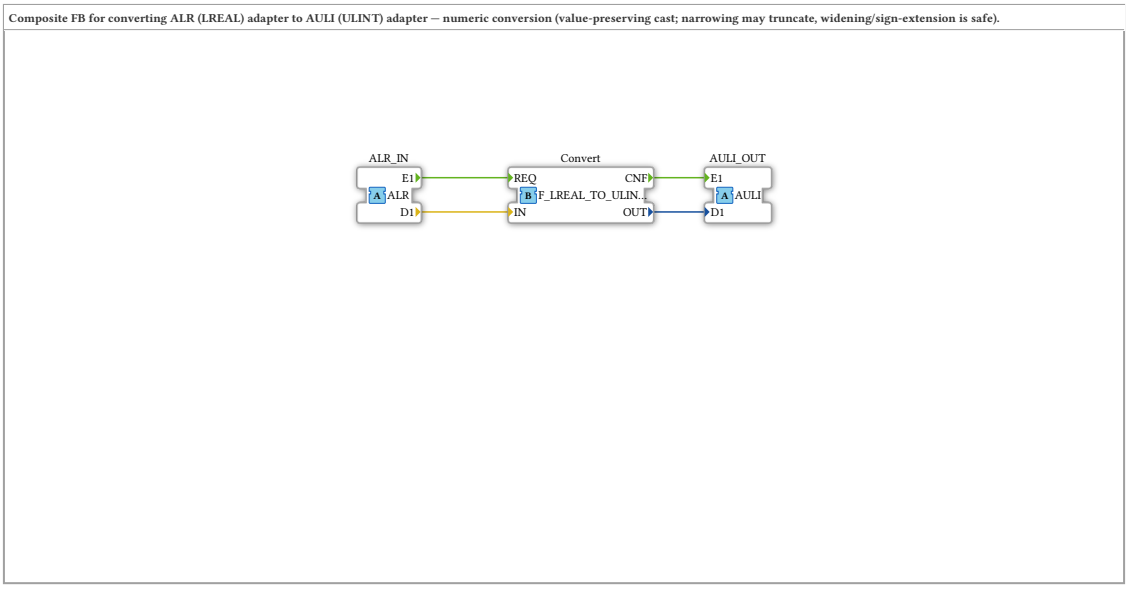
AUI_TO_ALR is a minimal yet clearly defined composite function block that establishes the connection between an AUI and an ALR adapter. Due to its simplicity and transparency, it is ideally suited for structured cabling in IEC 61499-based automation systems, especially in environments where different adapter protocols need to be coordinated.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALR_TO_AULI





• — ○ • — ○ • — ○ •

Introduction

The function block **ALR_TO_AULI** is a **composite FB** for the unidirectional conversion of an **LREAL adapter signal** (type `ALR`) into a **ULINT adapter signal** (type `AULI`). By internally using the IEC 61131 standard FB `F_LREAL_TO_ULINT`, the floating-point number (LREAL) is converted into an unsigned integer (ULINT) and provided via an output adapter. This block is particularly suitable for integrating floating-point data sources into systems that expect an integer adapter interface.

Interface Structure

The FB has **no direct** event or data inputs/outputs. Communication takes place exclusively via the two **adapters** (socket & plug). The following tables describe the signals provided via the adapters.

Event Inputs

Signal	Source	Data Type	Description
E1	via Socket `ALR_IN`	(Event)	Starts the conversion of an incoming LREAL value.

Event Outputs

Signal	Destination	Data Type	Description
E1	via Plug `AULI_OUT`	(Event)	Signals that the converted ULINT value is present at the output.

Data Inputs

Signal	Source	Data Type	Description
D1	via Socket `ALR_IN`	LREAL	The floating-point value to be converted.

Data Outputs

Signal	Destination	Data Type	Description
D1	via Plug `AULI_OUT`	ULINT	The converted unsigned integer value.

Adapter

Name	Type	Direction	Description
ALR_IN	adapter::types::unidirectional::ALR	Socket (Input)	Receives the LREAL value and its associated event.
AULI_OUT	adapter::types::unidirectional::AULI	Plug (Output)	Outputs the converted ULINT value with an acknowledgment event.

Functionality

The FB operates according to the **gearbox principle** of a composite block:

1. An incoming **event E1** at socket ALR_IN activates the internal FB F_LREAL_TO_ULINT (type iec61131::conversion::F_LREAL_TO_ULINT).
2. Simultaneously, the **data value D1** (LREAL) present at socket ALR_IN is forwarded to the input IN of the internal FB.
3. The internal FB immediately performs the conversion from LREAL to ULINT and outputs the result at its output OUT.
4. The event CNF of the internal function block triggers the **output event E1** at plug AULI_OUT.
5. The converted ULINT value is simultaneously applied to the data output D1 of plug AULI_OUT.

The entire conversion is **event-driven** and without its own state logic.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **License**: The function block is provided under the **Eclipse Public License 2.0** (see copyright notice).
- **Standard Compliance**: The internally used conversion block F_LREAL_TO_ULINT complies with the IEC 61131-3 standard.
- **No State Machine**: As a composite function block (FB), this block has no inherent state logic – the conversion is purely combinatorial.
- **Unidirectional Adapters**: Input and output are separate adapter types that define a clear signal direction.

State Overview

Since this is a **composite FB without its own state machine**, there is no explicit state overview. The functionality is event-triggered and behaves like a **combinatorial function**: An event at the input, after internal processing, results in exactly one event at the output.

Application Scenarios

- **Connecting floating-point sensors** to a controller that only processes integer adapter signals (e.g., in agricultural technology).
- **Data processing** in an adapter-based communication chain when the source delivers LREAL, but the destination expects ULINT.
- **Replacement of manual type conversions** in systems that rely on the standardized IEC 61131 functions.

Comparison with similar function blocks

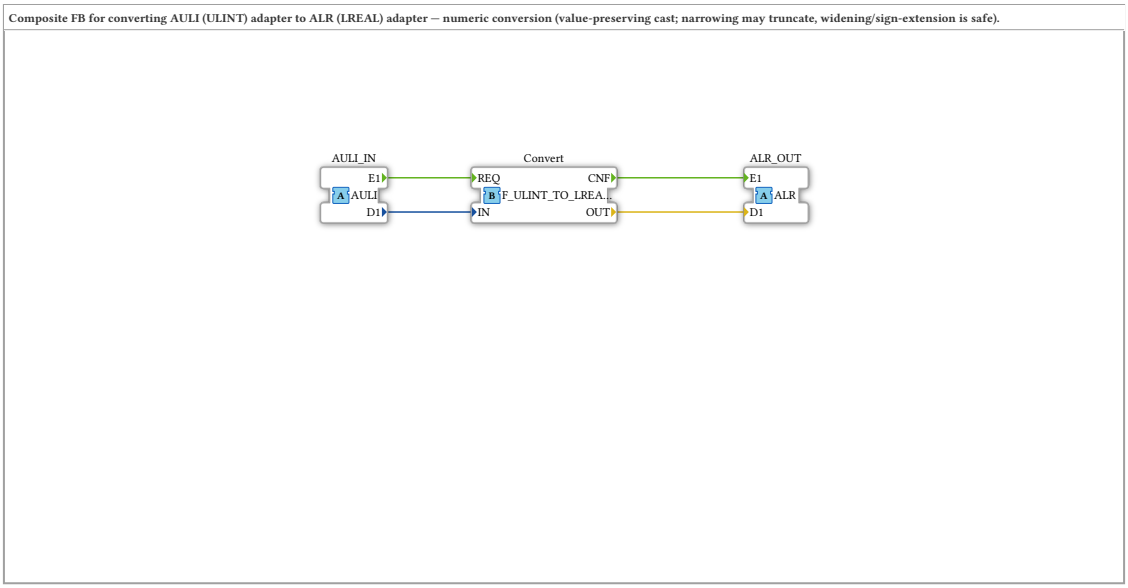
Other conversion adapters (e.g., REAL_TO_DINT, LREAL_TO_LINT) follow the same pattern: They use an internal IEC function block and encapsulate the type conversion in an adapter interface. The advantage of ALR_TO_AULI lies in the **targeted combination of LREAL and ULINT** – a type combination that is not offered as a ready-made adapter in many standard libraries.

Conclusion

ALR_TO_AULI` is a specialized **composite adapter function block** for lossy conversion of floating-point numbers to integers. Thanks to the use of a standardized IEC function block, the conversion is

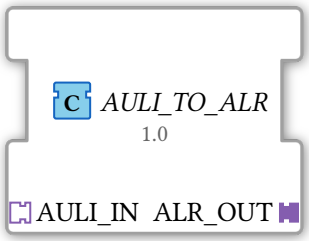
deterministic and easily testable. The function block is ideally suited for creating a clean adapter interface between LREAL and ULINT domains in heterogeneous automation systems.

AULI_TO_ALR



No image available.

ULINT Adapter Input – *AULI*



ALR – LREAL Adapter Output

• – ◦ • – ◦ • – ◦ •

Introduction

The **AULI_TO_ALR** is a composite function block (FB) that converts an incoming AULI adapter (ULINT data) into an outgoing ALR adapter (LREAL data). It serves as a standardized interface between components that communicate different data types via adapters.

Interface Structure

Event Inputs

The block has no direct event inputs. The incoming event is provided via the socket adapter **AULI_IN**:

- **AULI_IN.E1** (Event) – Receives the trigger signal for the conversion.

Event Outputs

The block has no direct event outputs. The outgoing event is provided via the **ALR_OUT** plug adapter:

- **ALR_OUT.E1** (Event) – Signals that the converted data is available at the data output.

Data Inputs

Direct data inputs are not available. The input data is provided via the **AULI_IN** socket adapter:

- **AULI_IN.D1** (ULINT) – The value to be converted in unsigned long integer format.

Data Outputs

Direct data outputs are not available. The output data is provided via the **ALR_OUT** plug adapter:

- **ALR_OUT.D1** (LREAL) – The converted value in long real format (64-bit floating-point number).

Adapters

The function block uses exactly two adapters:

- **Socket AULI_IN** (Type: AULI from `adapter::types::unidirectional`) – Receives the ULINT values and the associated trigger event.
- **Plug ALR_OUT** (Type: ALR from `adapter::types::unidirectional`) – Provides the converted LREAL values and an acknowledgment event.

Functionality

The function block operates according to the request-acknowledgment principle:

1. As soon as an event arrives at **E1** on the input adapter **AULI_IN**, the internal function block **F_ULINT_TO_LREAL** (from the library `iec61131::conversion`) is called.
2. The value **AULI_IN.D1** is fed to the input **IN** of the converter. 3. The converter performs the conversion from ULINT to LREAL (integer to floating-point number).
3. After the conversion is complete, the result is forwarded to **ALR_OUT.D1**.
4. Simultaneously, the event **ALR_OUT.E1** is triggered at the output adapter to signal the downstream function block that the data is being processed.

All logic runs within a function block network and is transparent to the user.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-based communication:** The function block encapsulates the type conversion in an adapter interface, allowing it to be seamlessly integrated into systems with standardized adapter types (e.g., AULI and ALR). This increases compatibility between components.
- **Reuse of existing function blocks:** As a composite function block, it uses the standardized conversion block `F_ULINT_TO_LREAL`, ensuring the maintainability and correctness of the conversion logic.
- **Compatibility with IEC 61499-2:** The function block complies with the standard and can be used in any 4diac IDE project.
- **No State Machine:** The conversion is purely combinatorial, controlled by events – there are no internal states.

State Overview

The function block has no independent states. The sequence control is triggered by the event at **AULI_IN.E1** and ends with the event at **ALR_OUT.E1**. During the conversion, the function block is in the state “waiting for confirmation from the internal function block.” This is not visible to the user.

Application Scenarios

- **Integration of sensors or actuators** that deliver their internal counter values as ULINT into a controller that requires LREAL for calculations (e.g., control algorithms).
- **Adapter bridge** between two subsystems with different adapter definitions, enabling type conversion without changing the interface definitions.
- **Data preparation** in communication protocols where received integers need to be converted into floating-point values for further processing.

Comparison with similar function blocks

Instead of a dedicated adapter converter, one could also use the function block `F_ULINT_TO_LREAL` directly and connect the events/data manually. The advantage of **AULI_TO_ALR** lies in the encapsulation of the adapter interfaces: It saves the developer from manually wiring the adapter events and data and ensures a clean, reusable component. Compared to a custom solution, it offers a standardized, documented interface.

Conclusion

The **AULI_TO_ALR** is a useful, specialized function block for converting ULINT to LREAL values in the context of adapters. It simplifies the integration of components with different data types into uniform adapter structures and increases the maintainability of IEC 61499 systems. Thanks to its standardized conversion function and clear, event-driven operation, it is reliable and easy to understand.

🌐 Related topic subpages on ms-muc-docs.de

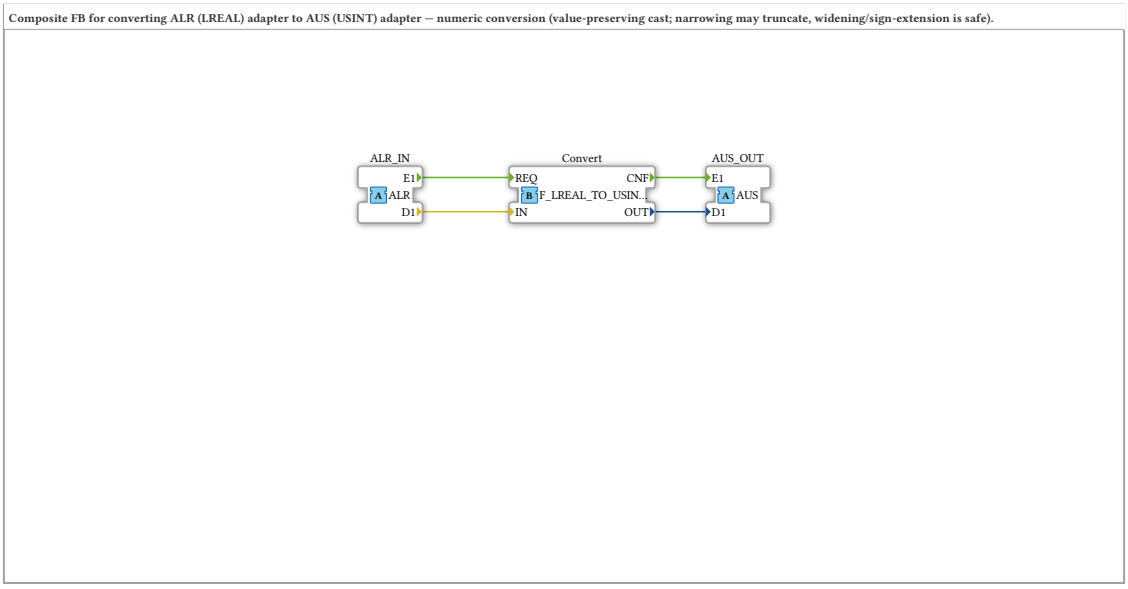
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALR_TO_AUS





• – ○ • – ○ • – ○ •

Introduction

The composite function block **ALR_TO_AUS** serves as a universal converter between two unidirectional adapter protocols. It converts an LREAL value (64-bit floating-point) into a USINT value (8-bit unsigned integer). The block was developed for use in IEC 61499-based automation systems and internally uses the standard function block `F_LREAL_TO_USINT` from the library `iec61131::conversion`.

Interface Structure

Since this is a composite block, all inputs and outputs are implemented via adapters.

Event Inputs

- No direct event inputs at the top level.

The adapter connected via the **Socket ALR_IN** has an event input **E1** that triggers the conversion.

Event Outputs

- No direct event outputs at the top level.

The adapter connected via the **Plug AUS_OUT** provides the event output **E1**, which signals the completion of the conversion.

Data Inputs

- No direct data inputs at the top level.

The **ALR_IN** adapter provides the data input **D1** (type: LREAL), which supplies the floating-point value to be converted.

Data Outputs

- No direct data outputs at the top level.

The **AUS_OUT** adapter provides the data output **D1** (type: USINT), which outputs the converted integer result.

Adapters

Name	Type	Direction	Comment
`ALR_IN`	`adapter::types::unidirectional::ALR`	Socket	Unidirectional LREAL input
`AUS_OUT`	`adapter::types::unidirectional::AUS`	Plug	Unidirectional USINT output

Functionality

The function block operates in an event-driven manner:

1. An incoming event at the adapter input `ALR_IN.E1` is internally connected to the **REQ** input of the conversion block `F_LREAL_TO_USINT`.
2. Simultaneously, the data value `ALR_IN.D1` is transferred to the data input `IN` of the conversion block.
3. The converter performs the conversion from LREAL to USINT. The floating-point value is rounded or truncated to the nearest USINT integer value (0–255) according to IEC 61131-3.
4. After successful conversion, the converter generates the event `CNF`, which is forwarded directly to the adapter output `AUS_OUT.E1`.
5. The converter's result `OUT` is transferred to the data output `AUS_OUT.D1`.

This enables synchronous, data-driven conversion without any additional delay.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Structure**: The block encapsulates the conversion logic in a reusable unit and utilizes the standardized adapter types `ALR` and `AUS`. This allows for seamless integration into existing unidirectional communication protocols.
- **Standard-Compliant Conversion**: The actual conversion is performed by the IEC 61131 block `F_LREAL_TO_USINT`, which guarantees defined behavior (e.g., rounding or capping) according to the standard.
- **EPL 2.0 License**: The block is released under the Eclipse Public License 2.0, which allows unrestricted use in proprietary and open-source projects.
- **No State Machine**: As a composite, the block does not have its own state machine; the internal control is handled entirely by the converter block.

State Overview

The block itself does not have an explicit state machine. The embedded function block `F_LREAL_TO_USINT` typically implements the states **IDLE**, **BUSY**, and **ERROR** during this conversion. For invalid inputs (e.g., USINT overflow), the converter can trigger an error, which, however, is not passed on to the adapter outputs in this version.

Application Scenarios

- **Sensor/Actuator Connection**: When a floating-point sensor value (e.g., pressure, temperature) needs to be converted into an 8-bit signal for a simple actuator (e.g., switch position, brightness).
- **Protocol Adaptation**: In systems that mediate between different adapter types for data transmission, such as in fieldbus couplers or gateway functions.

- **Testing and Simulation**: For the easy conversion of LREAL test data to USINT formats to test the functionality of downstream components.

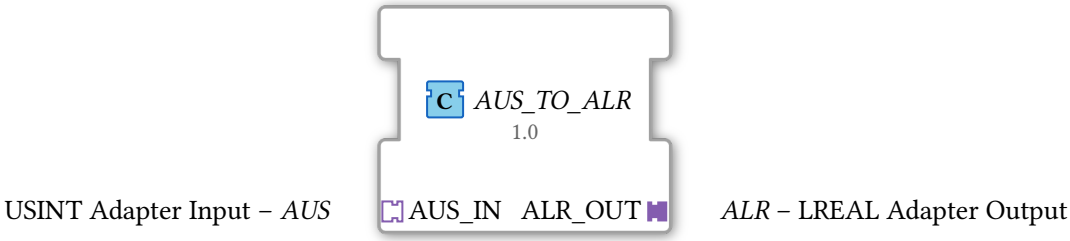
Comparison with Similar Components

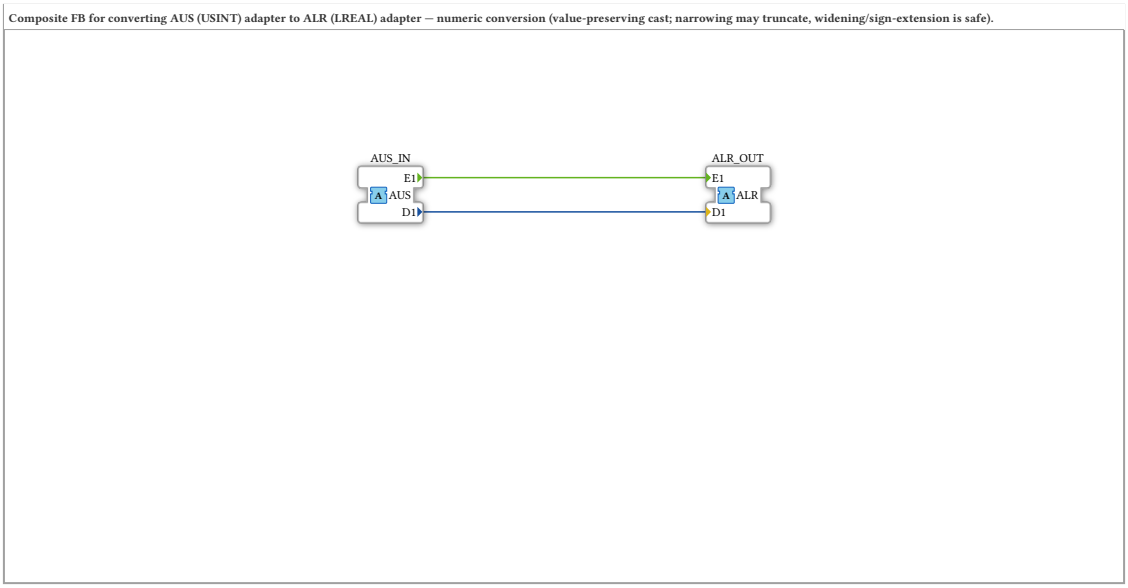
- **LREAL_TO_INT / LREAL_TO_DWORD**: These components provide different output types (INT, DWORD) and are therefore suitable for target platforms with wider or signed integers.
- **REAL_TO_USINT**: A corresponding variant is available for 32-bit floating-point numbers (REAL). However, this component uses the more precise LREAL type.
- **Scaling with Arithmetic Blocks**: Manual conversion using `MUL` and `ADD` would be possible, but less standardized and more prone to errors. The `ALR_TO_AUS` block offers a tested, standards-compliant solution.

Conclusion

The **ALR_TO_AUS** function block is a specialized but indispensable tool for converting LREAL to USINT data in IEC 61499 applications. Its composite design allows for a clean separation of the conversion logic and easy integration via standardized adapters. It is particularly suitable for scenarios where precise floating-point values need to be mapped to a narrow integer range.

OFF_TO_ALR





• – ○ • – ○ • – ○ •

Introduction

The **OFF_TO_ALR** function block is a composite function block that converts an **OFF adapter** (with USINT data) into an **ALR adapter** (with LREAL data). It serves purely as a forwarding interface and does not perform any complex logic or conversion calculations.

Interface Structure

Event Inputs

No separate event inputs are available. Event forwarding is handled via the **OFF_IN** adapter.

Event Outputs

No separate event outputs are available. Event forwarding is handled via the **ALR_OUT** adapter.

Data Inputs

No separate data inputs are available. Data is forwarded via the **AUS_IN** adapter.

Data Outputs

No separate data outputs are available. Data is forwarded via the **ALR_OUT** adapter.

Adapters

Role	Name	Type	Comment
Socket (Input)	AUS_IN	`adapter::types::unidirectional::AUS`	USINT Adapter Input
Plug (Output)	ALR_OUT	`adapter::types::unidirectional::ALR`	LREAL Adapter Output

- The **AUS_IN** adapter provides the interface for incoming USINT data, including an associated event.
- The **ALR_OUT** adapter provides the interface for outgoing LREAL data, including an associated event.
- Inside the function block, the events (`E1`) and data (`D1`) are directly routed from the *AUS_IN* interface to the *ALR_OUT* interface.

Functionality

1. **Event Forwarding**: The event (`E1`) received by the **AUS_IN** adapter is passed unchanged to the **ALR_OUT** adapter (`E1`).
2. **Data Forwarding**: The data value (`D1`) from the **OFF_IN** adapter is passed directly to the **ALR_OUT** adapter (`D1`).
3. **Data Type Conversion**: The actual conversion from `USINT` (8-bit integer value) to `LREAL` (64-bit floating-point number) does **not take place within the function block** – the **OFF_TO_ALR** block merely provides a structural framework for this. The conversion must be implemented either by the connected adapters or by upstream function blocks.

The function block itself contains no computation logic or state machine; it consists solely of connections between the adapters.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Function Block**: The function block is implemented as a composite function block (CFB) whose internal network contains only two connections.
- **No Internal Logic**: All functionality is provided by external function blocks or the adapter implementation.
- **Type Compliance**: The connection between `AUS.D1` (USINT) and `ALR.D1` (LREAL) requires that the underlying adapter implementation allows implicit type conversion. For caution, explicit conversion outside the function block is necessary.

State Overview

The function block has **no internal state machine**. It operates transparently: As soon as an event arrives at the input adapter, it is immediately forwarded to the output adapter. There are no delays, storage, or internal states.

Application Scenarios

- **Connecting Heterogeneous Control Systems**: A sensor delivers values as USINT via an AUS adapter, while the subsequent control module expects LREAL via an ALR adapter.
- **Structural Adaptation**: The function block (FB) serves as a pure conversion adapter when protocol/interface adapter types use different data formats.
- **Test and Simulation Environments**: Simple pass-through of data and events for testing adapter communication.

Comparison with Similar Function Blocks

Feature	AUS_TO_ALR	Typical Conversion FBs (e.g., USINT_TO_LREAL)
Input / Output	Adapter Interfaces	Individual Data Inputs/Outputs
Event Processing	Pass-Through via Adapter	Custom Event I/C Pairs
Implementation	Composite (Connections Only)	Executable Algorithms
Type Conversion	Not Included in the Function Block	Explicit Conversion via Algorithm

While a classic *USINT_TO_LREAL* function block performs the value conversion itself, **AUS_TO_ALR** only provides adapter adaptation. A separate function block must be interposed for the actual data conversion.

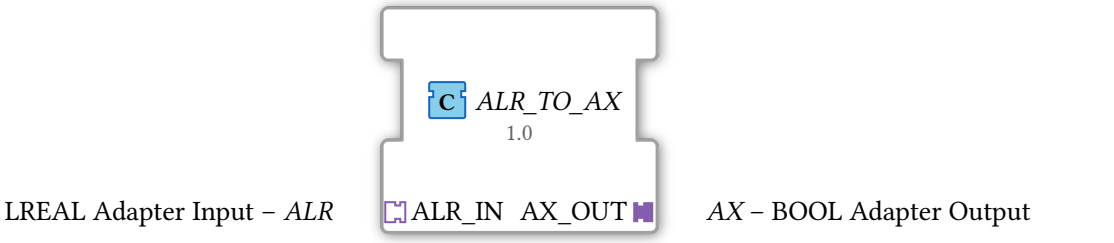
Conclusion

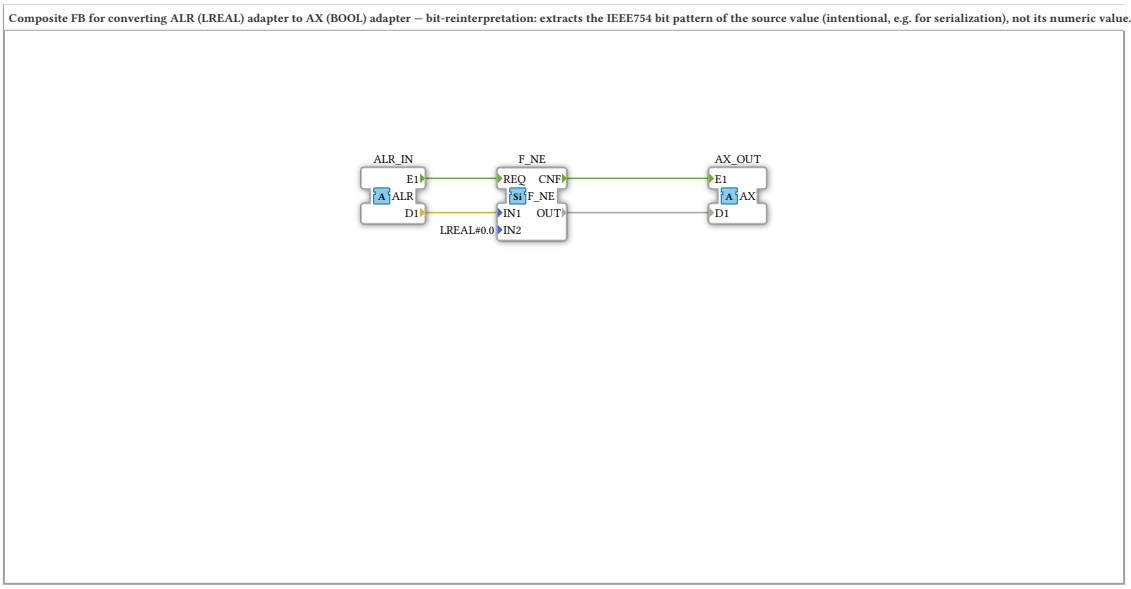
The **AUS_TO_ALR** function block is a simple yet useful composite block for structural adaptation between an AUS and an ALR adapter. It does not perform data conversion but enables the seamless forwarding of events and values between different adapter interfaces. Due to its minimalist design, it is particularly suitable for integration and testing tasks where the actual type conversion is performed externally.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALR_TO_AX





• – ◦ • – ◦ • – ◦ •

Introduction

The **ALR_TO_AX** function block is a composite function block for converting an LREAL value (via the unidirectional ALR adapter) into a BOOL value (via the unidirectional AX adapter). It is typically used to convert an analog input signal into a digital output signal – for example, to detect a threshold being exceeded. The function block is based on a simple zero comparison and triggers the output event as soon as a valid value is present at the input.

Interface Structure

Event Inputs

- **ALR_IN.E1** – Event from the ALR adapter; indicates that a new LREAL value is available.

Event Outputs

- **AX_OUT.E1** – Event to the AX adapter; Indicates that the converted BOOL value is valid.

Data Inputs

- **ALR_IN.D1** – LREAL input value (via the ALR adapter).

Data Outputs

- **AX_OUT.D1** – BOOL output value (via the AX adapter); derived by comparing the input value to 0.0.

Adapters

- **ALR_IN** – Socket (Type: *adapter::types::unidirectional::ALR*): unidirectional LREAL input adapter.
- **AX_OUT** – Plug (Type: *adapter::types::unidirectional::AX*): unidirectional BOOL output adapter.

Functionality

When an event arrives via **ALR_IN.E1**, the data value **ALR_IN.D1** (LREAL) is passed to the internal IEC 61131 block **F_NE**. This compares the value with the constant `LREAL#0.0` and returns **TRUE** if the input is not zero, otherwise **FALSE**. The result is then passed directly to **AX_OUT.D1**, and an acknowledgment event is triggered via **AX_OUT.E1**.

The entire process takes place within a single cycle, as this is a composite block without its own state machine.

Technical Details

- **Bit-reinterpretation**: extracts the IEEE754 bit pattern of the source value (intentional, e.g. for serialization), not its numeric value.
- **License & Origin**: The block is licensed under the Eclipse Public License 2.0 and was provided with version information 1.0.
- **Libraries Used**: The comparison block *F_NE* is from the IEC 61131 library (*iec61131::comparison*).
- **Package Name**: The function block belongs to the package *adapter::conversion::unidirectional*.
- **Compatibility**: It requires the adapter specification of the 4diac framework and is designed for unidirectional adapter types.

State Overview

The function block does not have an explicit state machine. Its logic is purely combinational and event-driven:

1. Wait for an event at **ALR_IN.E1**.
2. Data processing (comparison with 0.0).
3. Output the BOOL result to **AX_OUT.D1** and send the acknowledgment event to **AX_OUT.E1**.

Application Scenarios

- **Analog-to-Digital Conversion**: Detects whether an analog measurement value (e.g., pressure sensor, temperature) is non-zero.
- **Threshold Detection**: Any threshold logic can be implemented through simple modification (replacing *F_NE* with *F_GT* or *F_LT*).
- **Adapter Coupling**: Integrates ALR-based sensors into AX-based control logic without requiring complex type conversions at a higher level.

Comparison with Similar Function Blocks

- **LREAL_TO_BOOL**: While a dedicated LREAL-to-BOOL function block often supports a threshold and hysteresis, *ALR_TO_AX* implements the simplest form – null ≠ null ⟶ TRUE.
- **Comparison blocks (F_NE, F_EQ, etc.)**: These operate directly on data, while *ALR_TO_AX* handles complete adapter integration and event control.
- **Adapter-Converter blocks**: The same package contains additional unidirectional/bidirectional converters that perform similar tasks for other types (e.g., BOOL_TO_ALR).

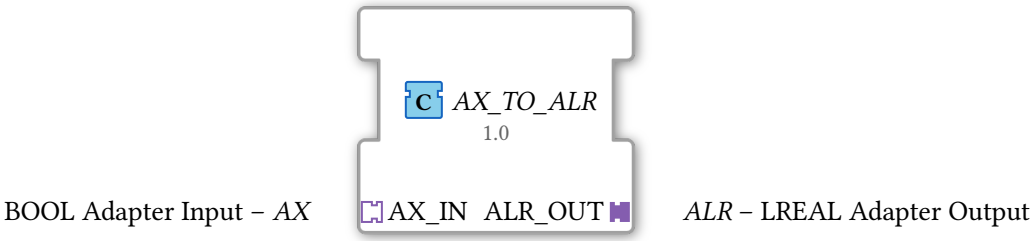
Conclusion

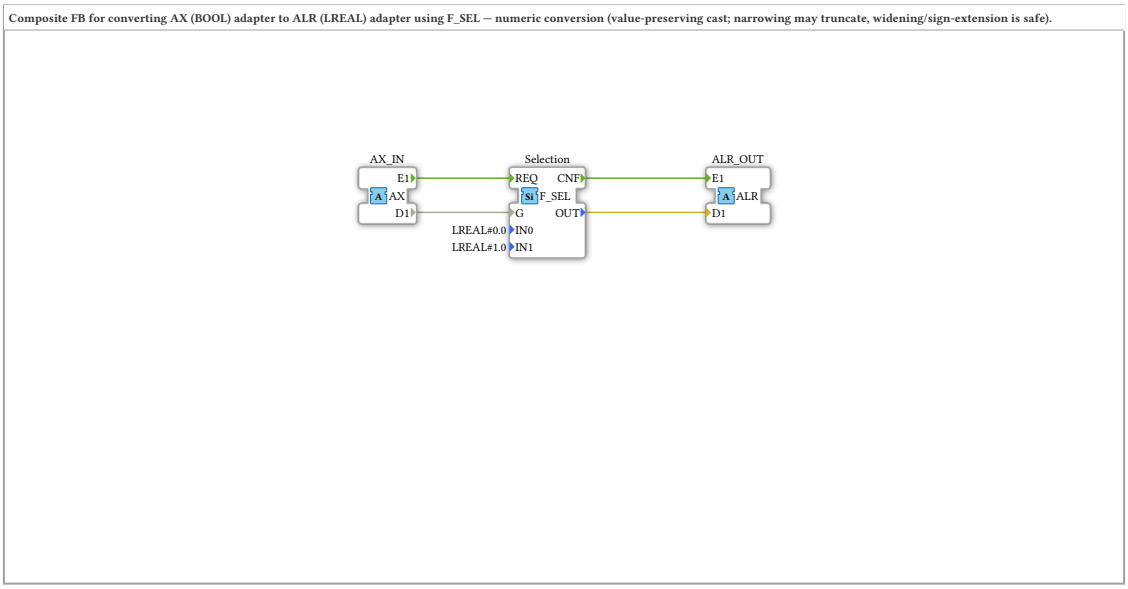
The **ALR_TO_AX** function block offers a streamlined and reliable way to convert an LREAL adapter output to a BOOL adapter input. Thanks to its simple structure, it is ideally suited for basic digitization tasks in IEC 61499-based control systems. It is easy to understand, readily extensible, and follows common design patterns for unidirectional adapter conversion.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AX_TO_ALR





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_TO_ALR** is used to convert a BOOL adapter signal (AX) into an LREAL adapter signal (ALR). Internally, it uses the IEC 61131 standard function `F_SEL` (Selection) to generate a discrete LREAL value from a Boolean input. The block is implemented as a composite function block and is suitable for the simple, adapter-based conversion of digital states into numerical values.

Interface Structure

Event Inputs

Name	Data Type	Description
AX_IN.E1	EVENT	Event from the connected AX adapter; triggers processing.

Event Outputs

Label	Data Type	Description
ALR_OUT.E1	EVENT	Confirmation event after the converted value has been output to the ALR adapter.

Data Inputs

Label	Data Type	Description
AX_IN.D1	BOOL	Boolean value to be converted to an LREAL value.

Data Outputs

Label	Data Type	Description
ALR_OUT.D1	LREAL	Conversion result: 0.0 for FALSE, 1.0 for TRUE.

Adapters

Direction	Label	Type	Description
Socket (Input)	AX_IN	adapter::types::unidirectional::AX	Receives a BOOL signal via the unidirectional AX adapter.
Plug (Output)	ALR_OUT	adapter::types::unidirectional::ALR	Provides the converted LREAL signal via the unidirectional ALR adapter.

Functionality

The function block operates according to the following scheme:

1. An incoming event at AX_IN.E1 triggers the internal processing.
2. The Boolean value of AX_IN.D1 is passed to the gate input (G) of the embedded function F_SEL.
3. Depending on the Boolean value, F_SEL selects one of two constant values:

- FALSE (G = 0) ⟶ Output of **0.0** (LREAL#0.0)
- TRUE (G = 1) ⟶ Output of **1.0** (LREAL#1.0)

1. The result is passed to ALR_OUT.D1, and the acknowledgment event ALR_OUT.E1 is simultaneously output.

`ALR_OUT.D1is passed to the gate input (G) of the embedded functionALR_OUT.E1``. The conversion is event-driven: Each incoming event causes exactly one conversion of the current BOOL value.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter Interface**: The function block uses only unidirectional adapters (socket/plug), allowing it to integrate seamlessly into adapter-based architectures.
- **Internal Use of F_SEL**: The IEC 61131 standard function guarantees deterministic and portable behavior.

- **Fixed Thresholds**: The output values are fixed at `0.0` and `1.0`. Adjusting these thresholds requires modifying the internal network.
- **Packaging**: The function block is packaged in `adapter::conversion::unidirectional`.
- **License**: The function block is available under the Eclipse Public License 2.0.

State Overview

Since this is a composite function block (FB) without its own execution state, the state logic is derived from the event and data processing of the included `F_SEL` block. Two stable states exist:

- **Waiting for Event**: No event at `AX_IN.E1` – no output.
- **Processing Active**: An event is being processed: the Boolean value is read, converted into an LREAL value, and made available at the output; subsequently, the acknowledgment event is sent.

Application Scenarios

- **Digital-to-Analog Conversion**: A digital sensor (e.g., switch, limit switch) provides a Boolean value that is to be further processed in the controller as an LREAL signal (0.0/1.0).
- **Adapter Integration**: Integration into existing systems that already use unidirectional AX and ALR adapters – e.g., in agricultural technology or industrial automation.
- **State Scaling**: Simplified conversion of on/off signals into numerical values for calculations or visualizations.

Comparison with Similar Function Blocks

Function Block	Input Type	Output Type	Special Feature
`AX_TO_ALR`	BOOL (Adapter)	LREAL (Adapter)	Adapter-based, fixed values 0.0 / 1.0
`BOOL_TO_REAL`	BOOL (direct)	REAL (direct)	Conventional IEC function block, values freely configurable
`SEL` / `F_SEL`	BOOL + 2x ANY	ANY	General selection without adapter, values configurable

Compared to direct function blocks, `AX_TO_ALR` offers the advantage of clean adapter encapsulation and easy reuse in adapter-oriented projects.

Conclusion

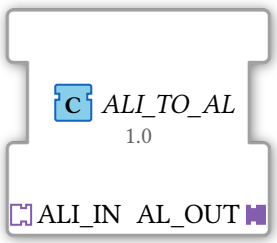
The function block `AX_TO_ALR` provides a compact and reliable solution for converting a Boolean adapter signal into an LREAL adapter value. By using the IEC standard function `F_SEL`, its behavior is deterministic and portable. It is particularly suitable for applications that require a clean separation of digital and analog signal paths via adapters.

Numeric Index (unidirectional)

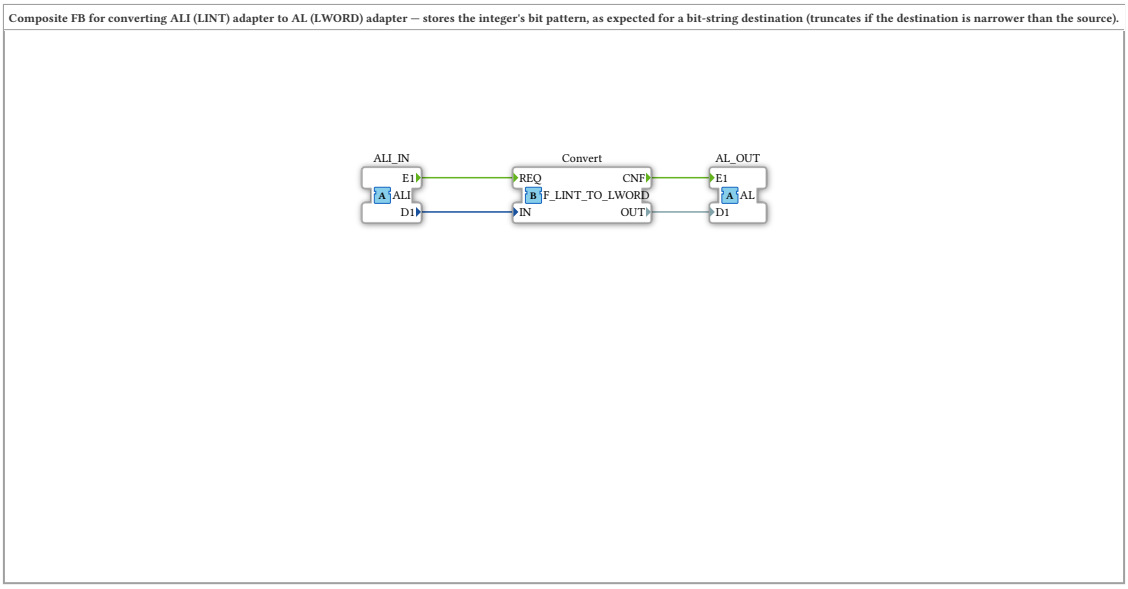
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALI_TO_AL

LINT Adapter Input – *ALI*



AL – LWORD Adapter Output



• – ○ • – ○ • – ○ •

The **ALI_TO_AL** function block is a composite function block that converts an **ALI adapter** (data type LINT) into an **AL adapter** (data type LWORD). It serves as an interface converter between different adapter types and enables seamless data transfer in 4diac and IEC 61499 systems.

- **No direct event inputs**

Events are received exclusively via the **ALI_IN** socket adapter.

- **No direct event outputs**

Output events are sent exclusively via the **AL_OUT** plug adapter.

- **No direct data inputs**

Input data (LINT) is provided via the socket adapter **ALI_IN** (data point D1).

- **No direct data outputs**

Output data (LWORD) is provided via the plug adapter **AL_OUT** (data point D1).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Adapter	Direction	Type	Description
ALI_IN	Socket	`adapter::types::unidirectional::ALI`	Receives a LINT value and an event.
AL_OUT	Plug	`adapter::types::unidirectional::AL`	Sends an LWORD value and an event.

Functionality

The function block operates internally with a single conversion function block (FB) `F_LINT_TO_LWORD`. As soon as an event (E1) arrives at the socket adapter **ALI_IN**, it is forwarded to the internal converter (`REQ` input). Simultaneously, the LINT value (D1) provided by the adapter is passed to the converter's `IN` input.

The converter transforms the LINT value into an LWORD value and signals completion with its `CNF` output. This event is sent to the plug adapter **AL_OUT** (E1), along with the converted LWORD value (OUT) on data channel D1.

- **Composite Function Block**: The actual type conversion is performed by the embedded function block `F_LINT_TO_LWORD` from the IEC 61131 standard library.
- **Unidirectional Adapters**: The data and event flow is defined from ALI_IN to AL_OUT. Reverse communication is not supported.
- **No Internal States**: The function block does not have its own execution state (no ECC), but forwards events and data directly.

The function block has no internal states. It reacts strictly event-driven: Each incoming event at socket **ALI_IN** triggers exactly one conversion and one output at plug **AL_OUT**.

- **Adapter Bridge**: Used in systems that utilize an ALI adapter (LINT) but require an AL interface (LWORD).
- **Compatibility**: Retrofitting older components that expect LINT to an environment that uses LWORD via AL adapters.
- **AL_TO_ALI**: Performs the reverse conversion from LWORD to LINT.
- **F_LINT_TO_LWORD**: Pure conversion function block without adapter integration; direct data exchange without encapsulation.
- **Other Adapter Converters** (e.g., AL_TO_AL): Same adapter type, no type conversion.

The ALI_TO_AL function block stands out due to its simple plug-and-play capability within an adapter-based design, as it encapsulates all conversion logic and event control in a single block.

The composite function block **ALI_TO_AL** enables clean and standards-compliant conversion of LINT to LWORD data via unidirectional adapters. It is ideal for use in modular automation systems that need to integrate different adapter data types. Reliable and high-performance execution is ensured by utilizing the proven IEC 61131 conversion function.

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).

State Overview

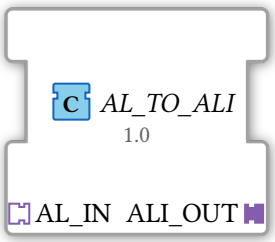
Application Scenarios

Comparison with Similar Function Blocks

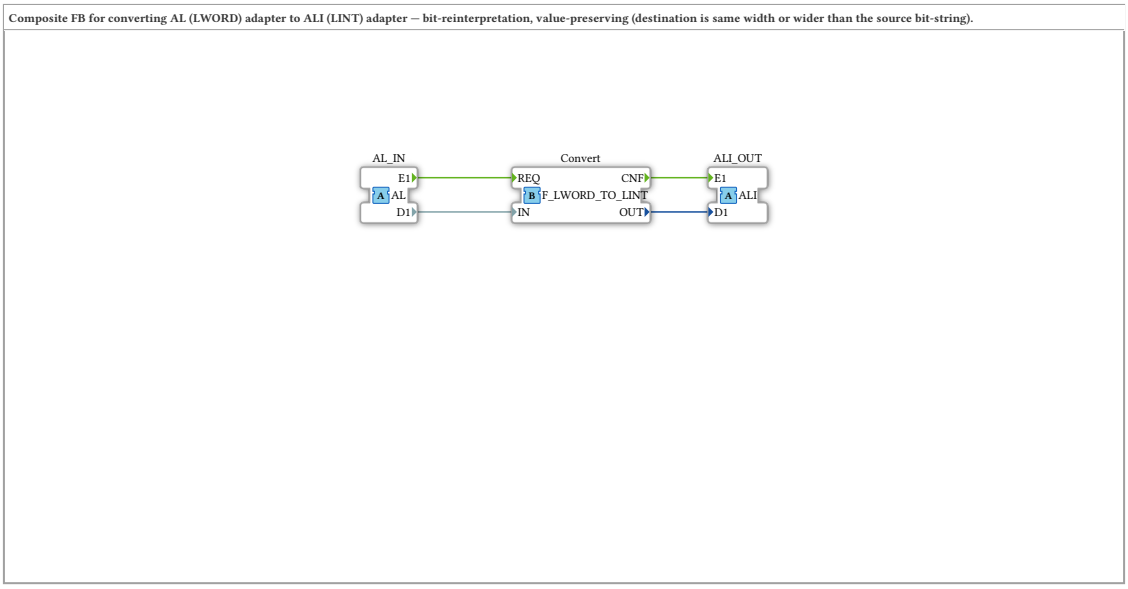
Conclusion

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

AL_TO_ALI



LWORD Adapter Input – *AL* — *ALI* – LINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block `AL_TO_ALI` is a composite function block (FB) for converting an AL adapter (LWORD) to an ALI adapter (LINT). It serves as an interface between LWORD-based and LINT-based data communication within the control logic. Internally, the block uses the conversion block `F_LWORD_TO_LINT` to perform the actual type conversion.

Interface Structure

Event Inputs

The function block does not have its own event inputs. Event control is handled via the connected adapters.

Event Outputs

The function block does not have its own event outputs. Event control is handled via the connected adapters.

Data Inputs

This function block does not have its own data inputs. Input data is provided via socket `AL_IN`.

Data Outputs

This function block does not have its own data outputs. Output data is provided via plug `ALI_OUT`.

Adapters

Type	Name	Direction	Data Type	Description
`adapter::types::unidirectional::AL`	`AL_IN`	Socket (Input)	LWORD	LWORD adapter as input interface. Provides the data to be converted.
`adapter::types::unidirectional::ALI`	`ALI_OUT`	Plug (Output)	LINT	LINT adapter as output interface. Delivers the converted data.

Functionality

1. **Event Triggering:** The `AL_IN` adapter sends a signal via its event output `E1` as soon as new data is available.
2. **Conversion:** This event triggers the internal function block `Convert` (`F_LWORD_TO_LINT`) at its event input `REQ`. Simultaneously, the current data value of `AL_IN.D1` (LWORD) is transferred to the data input `IN` of the conversion module.

3. **Output:** After successful conversion, `Convert` sends an acknowledgment via its event output `CNF` to the `ALI_OUT` adapter (event input `E1`). In parallel, the converted LINT value is forwarded from output `OUT` to the data input `ALI_OUT.D1`.
4. **Result:** The `ALI_OUT` adapter now forwards the new LINT data to the connected function block.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Type Conversion:** This function block converts an LWORD (64-bit sequence) into a LINT (64-bit signed integer). The conversion is bit-for-bit, meaning the binary representation is interpreted as a signed integer.
- **Composition:** This is a composite function block that does not contain its own logic but relies on the standardized conversion function block `iec61131::conversion::F_LWORD_TO_LINT`.
- **Adapter-Based Interface:** Input and output are exclusively via unidirectional adapters (`AL` and `ALI`), enabling a modular and type-checked connection in the 4diac IDE.
- **Adapter-Based Interface: - Package Structure:** The FB is located in the package `adapter::conversion::unidirectional`.

State Overview

Since this is a composite FB, there is no separate state machine. The state logic is entirely determined by the internally used block `F_LWORD_TO_LINT`. This block operates in an event-driven manner:

- **IDLE:** Wait for an event at the `REQ` input.
- **PROCESSING:** Convert the LWORD to LINT (takes one step).
- **FINISHED:** Output the converted value and send the `CNF` event.

Application Scenarios

- **PLC Data Conversion:** When a control program receives LINT values (e.g., counter readings or times) via an LWORD-based adapter and a LINT interface is required.
- **Communication Adapter:** Used as an intermediary in an adapter chain to translate different data types between various hardware or software modules.
- **Protocol Adaptation:** For example, converting raw LWORD data (e.g., from a bus system) into a LINT representation for further processing in the control logic.

Comparison with Similar Function Blocks

- **AL_TO_AL (LWORD ↔ LWORD):** Simple pass-through without conversion.
- **ALI_TO_AL (LINT ↔ LWORD):** Performs the reverse conversion.
- **F_LWORD_TO_LINT (direct conversion block):** Offers the same functionality, but without an adapter interface. `AL_TO_ALI` encapsulates this block and adds the adapter connection.

Conclusion

The `AL_TO_ALI` function block is a simple yet important converter that bridges the gap between LWORD-based and LINT-based adapter interfaces. Its composite structure keeps it clear and allows for flexible integration into existing 4diac projects. The use of standard conversion blocks ensures correct and efficient type conversion.

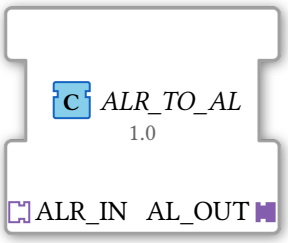
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

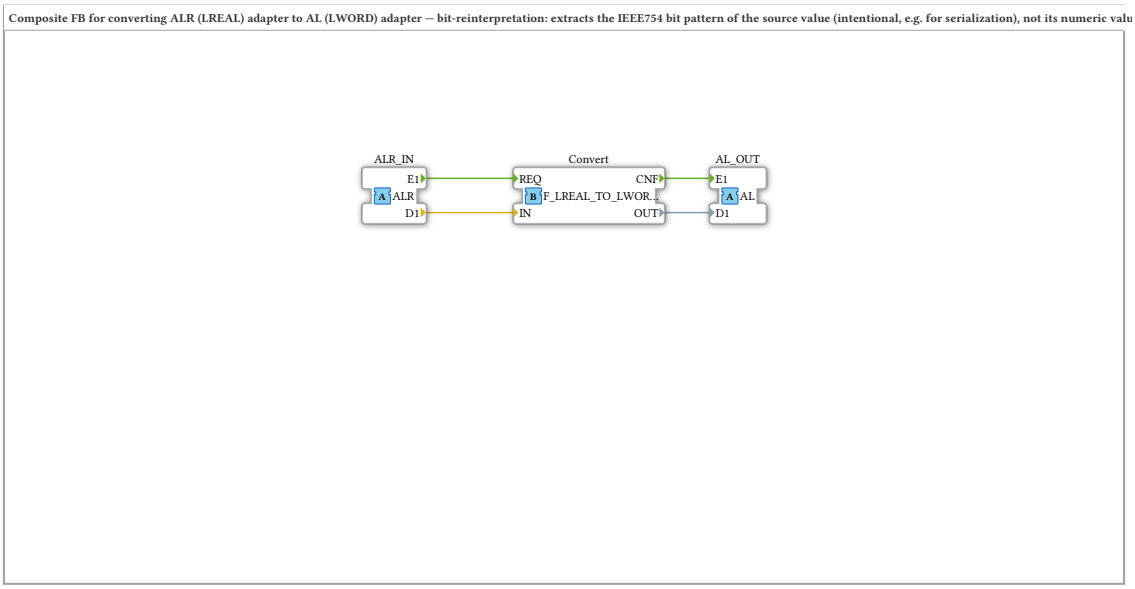
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

ALR_TO_AL



LREAL Adapter Input – *ALR* *AL* – LWORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **ALR_TO_AL** function block is a composite function block according to IEC 61499-2. It is used to convert an **ALR adapter** (LREAL data type) into an **AL adapter** (LWORD data type). The conversion is implemented via the internal function block `F_LREAL_TO_LWORD` from the IEC 61131-3 library.

Interface Structure

Event Inputs

The function block has **no independent event inputs**. Events are provided via the **socket adapter** `ALR_IN`:

| Adapter | Event / Signal | Description | | Adapter | Event / Signal | Description | |---|---|---|---|---|---|---| | ALR_IN | E1 | Trigger event to initiate the conversion |

Event Outputs

This function block has **no independent event outputs**. Events are output via the **plug adapter** `AL_OUT`:

Adapter	Event / Signal	Description
AL_OUT	E1	Confirmation event after conversion is complete

Data Inputs

There are also **no direct data inputs**. Data is supplied via the socket adapter `ALR_IN`:

| Adapter | Data / Signal | Data Type | Description | | Adapter | Data / Signal | Data Type | Description | |---|---|---|---|---|---|---|---| | ALR_IN | D1 | LREAL | Input value in LREAL format |

Data Outputs

No direct data outputs. Output data is provided via the plug adapter `AL_OUT`:

Adapter	Data / Signal	Data Type	Description
AL_OUT	D1	LWORD	Converted output value in LWORD format

Adapters

The module is connected to the environment via two adapters:

- **Socket** `ALR_IN`: Of type `adapter::types::unidirectional::ALR` – provided by a preceding component.
- **Plug** `AL_OUT`: Of type `adapter::types::unidirectional::AL` – offered to a subsequent component.

The adapters transmit both events and data via their respective ports (E1, D1).

Functionality

When an event **E1** arrives at socket `ALR_IN`, the internal module `F_LREAL_TO_LWORD` (type `iec61131::conversion::F_LREAL_TO_LWORD`) is activated via its event input `REQ`. Simultaneously, the data value `D1` of the adapter socket is transferred to the data input `IN` of the conversion module. After successful conversion, the internal module outputs a confirmation event at its output `CNF`. This event is forwarded to the event input `E1` of the plug adapter `AL_OUT`. The converted value (LWORD) is transferred to the adapter plug via the data connection `Convert.OUT` ⟶ `AL_OUT.D1`.

... The conversion itself corresponds to the IEC 61131-3 function `LREAL_TO_LWORD`, which maps the bitwise structure of an LREAL value to an LWORD value (e.g., for transferring floating-point numbers as raw data).

Technical Features

- **Bit-reinterpretation**: extracts the IEEE754 bit pattern of the source value (intentional, e.g. for serialization), not its numeric value.
- **Composite Block**: The functionality is implemented by an internal network circuit that uses the block `F_LREAL_TO_LWORD`.
- **Packaging**: The block is stored in the package `adapter::conversion::unidirectional`, indicating its use in unidirectional adapter chains.
- **Import**: `iec61131::conversion::F_LREAL_TO_LWORD` must be available in the project.
- **Copyright**: This block was developed for HR Agrartechnik GmbH and is licensed under the Eclipse Public License 2.0 (EPL-2.0).

State Overview

Since this is a **composite function block** without its own state machine, its behavior is determined by that of the internal function block `F_LREAL_TO_LWORD`. This block performs a pure, stateless conversion: An event triggers the calculation, and the result is provided immediately. The function block has no internal state that persists across multiple calls.

Application Scenarios

- **Data Format Adaptation**: When a system component delivers data in ALR format (LREAL via adapter), but a subsequent component only understands AL interfaces (LWORD via adapter).
- **Communication Between Different Control Levels**: For example, transferring physical measured values from a PLC (LREAL) to a higher-level control system that expects LWORD as the raw data format.
- **Test and Simulation Setups**: Generating defined LWORD values from LREAL sources without changing the actual adapter logic.

Comparison with Similar Function Blocks

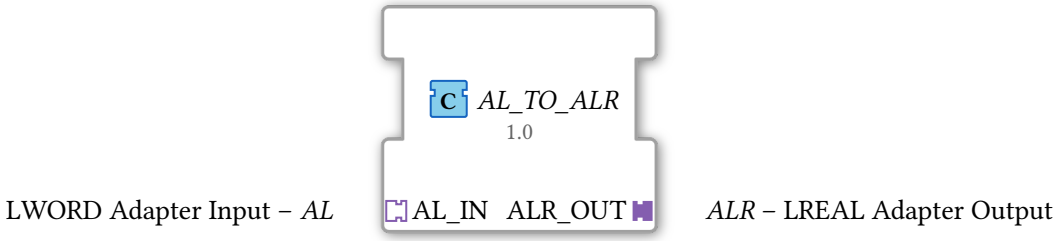
- **AL_TO_ALR**: Performs the reverse conversion (LWORD ⟶ LREAL).
- **Direct Conversion Functions**: IEC 61131-3 provides `LREAL_TO_LWORD` and `LWORD_TO_LREAL` as elementary function blocks – `ALR_TO_AL` encapsulates these in an adapter-compliant composite function block, allowing it to be directly integrated into adapter chains.

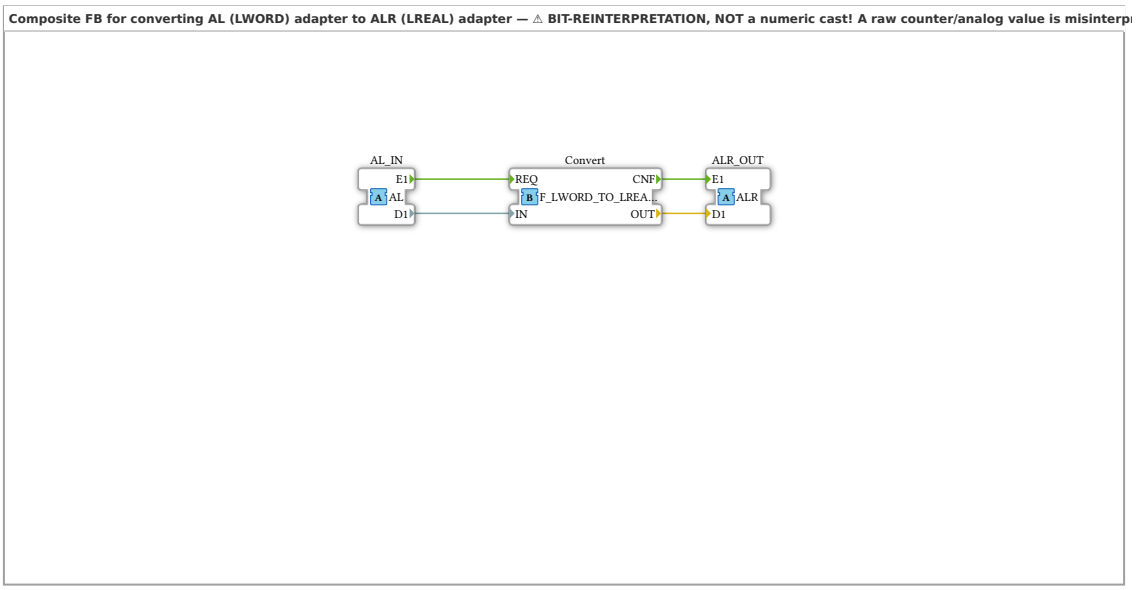
Conclusion

The **ALR_TO_AL** function block is a specialized conversion block for converting an LREAL adapter to an LWORD adapter. Its composite structure enables seamless integration into existing adapter-based

architectures without requiring the user to manually configure the underlying conversion function. The unambiguous event control and the use of standardized data types make it a robust solution for adapting data formats in IEC 61499 control systems.

AL_TO_ALR





• – ○ • – ○ • – ○ •

Introduction

The function block **AL_TO_ALR** is a composite function block (FB) that maps a unidirectional AL adapter (LWORD) onto a unidirectional ALR adapter (LREAL). Internally, it uses the block `F_LWORD_TO_LREAL`.

> ⚠ **Warning — not a numeric value conversion:** `LWORD` is a bit-string type (not a signed integer), and `F_LWORD_TO_LREAL` in the FORTE core merely reinterprets the 64-bit pattern as an IEEE754 `LREAL` (a bit copy) — the same trap as `AD_TO_AR` (DWORD→REAL), just with the 64-bit counterparts. A raw counter or analog value will therefore **not** become the corresponding LREAL number, but a meaningless value. For an actual numeric conversion: chain `AL_TO_AULI` (bit-reinterpretation LWORD→ULINT, valid here since both share the same 64-bit unsigned-integer representation) followed by `AULI_TO_ALR` (a real numeric cast) — the same pattern as `AD_TO_AR_NUM` for the 32-bit case.

Interface Structure

Event Inputs

The FB does not have direct event inputs. The event required to trigger the conversion is provided via the socket **AL_IN** (type `AL`). This event (E1) is internally connected to the `REQ` input of the conversion block.

Event Outputs

The function block (FB) does not have direct event outputs. The conversion completion event is output via the plug **ALR_OUT** (type `ALR`). The event (E1) of the internal block (`CNF`) is forwarded to the output adapter.

Data Inputs

The function block (FB) does not have direct data inputs. The LWORD value to be converted is provided via the **AL_IN** socket (data signal D1) and passed to the `IN` input of the conversion block.

Data Outputs

The function block has no direct data outputs. The converted LREAL value is output via the **ALR_OUT** plug (data signal D1); it originates from the `OUT` output of the internal block.

Adapters

Adapter	Direction	Type	Description
AL_IN	Socket (Input)	`adapter::types::unidirectional::AL`	LWORD adapter that provides the input signal (LWORD) and a trigger event.
ALR_OUT	Plug (Output)	`adapter::types::unidirectional::ALR`	LREAL adapter that outputs the converted signal (LREAL) and an acknowledgment event.

Functionality

`F_LWORD_TO_LREAL` reinterprets the LWORD's 64-bit pattern as an IEEE754 `LREAL` (a bit copy) — **no** numeric value conversion takes place. The function block operates in a simple event-driven sequence:

1. An incoming event at socket **AL_IN.E1** is forwarded to the `REQ` input of the internal function block `F_LWORD_TO_LREAL`.
2. Simultaneously, the data value from **AL_IN.D1** (LWORD) is passed to the `IN` input of the converter.
3. The internal function block (FB) performs the conversion `LWORD → LREAL`.
4. After the conversion is complete, the event `CNF` is sent to the plug **ALR_OUT.E1**.
5. The converted LREAL value is output to **ALR_OUT.D1**.

The entire processing takes place within one clock cycle (no blocking).

Technical Features

- **Adapter-based interface**: The FB uses only unidirectional adapters (`AL` and `ALR`). This enables loose coupling between sender and receiver and easy reuse in different system architectures.
- ⚠️ **Bit-reinterpretation, not numeric conversion**: See the warning in the introduction — affects all bit-string source types converting to AR/REAL or ALR/LREAL; the only other instance in this library is `AD_TO_AR` (DWORD⟶REAL).
- **Pass-Through Event Control**: The function block does not execute any internal state logic; it transparently forwards events and data. The call time corresponds to the execution time of the internal converter.

State Overview

The assembled function block itself has **no states of its own**. The processing is entirely controlled by the internally used function block `F_LWORD_TO_LREAL`, which exhibits the following minimal state behavior:

- **IDLE**: Waiting for a `REQ` event.
- **CONVERT**: Executing the conversion (immediately after `REQ`).
- **SEND**: Outputting the `CNF` event and the converted value.

Since the internal function block operates in a single execution step, the states are not externally observable.

Application Scenarios

- **Bit pattern pass-through**: `AL_IN` already carries a bit pattern meant to be interpreted as LREAL (e.g. serialized raw double data from a fieldbus/protocol).
- **NOT suitable** for raw counter, analog, or other integer values meant to carry the same numeric value as LREAL — use `AL_TO_AULI` + `AULI_TO_ALR` for that.

Comparison with Similar Function Blocks

- **`AL_TO_AULI` + `AULI_TO_ALR`** (numeric, LWORD→ULINT→LREAL): the safe replacement when an actual numeric value is meant.
- **`ALI_TO_ALR`/`AULI_TO_ALR`** (LINT/ULINT→LREAL): already numerically correct, since their source types are `ANY_INT` (not `ANY_BIT`) — no trap.
- **`AD_TO_AR`** (DWORD→REAL): the same bit-reinterpretation trap, just with the 32-bit counterparts.

Conclusion

`AL_TO_ALR` offers a compact way to expose an LWORD bit pattern as LREAL via adapter interfaces — **but not a numeric value conversion**. For raw counter or analog values meant to carry the same numeric value in LREAL, use `AL_TO_AULI` + `AULI_TO_ALR`.

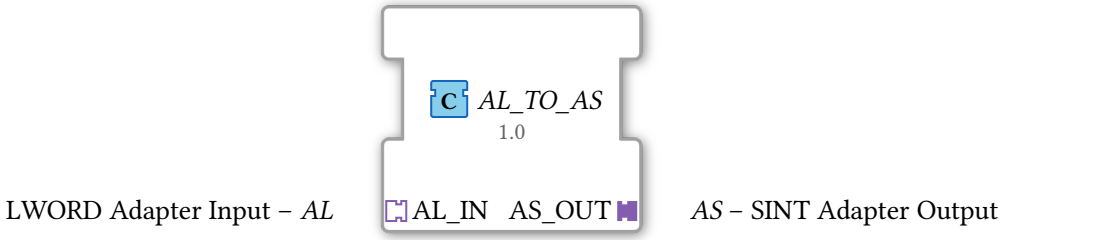
📖 Background

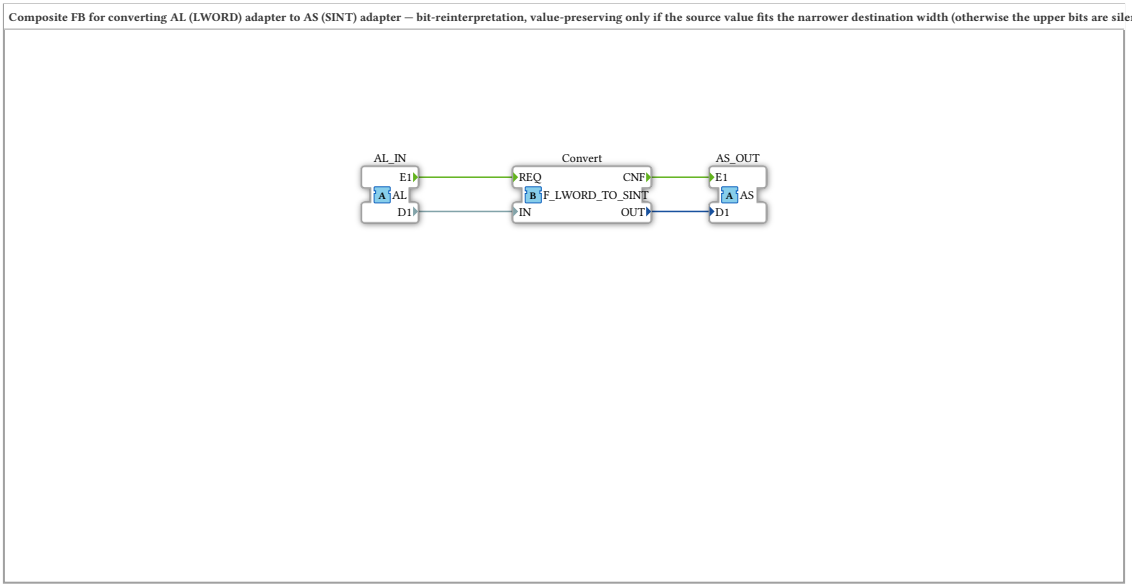
- Numeric vs. bitwise: the FORTE conversion trap

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AL_TO_AS





• – ○ • – ○ • – ○ •

Introduction

The **AL_TO_AS** function block is a composite function block that converts a unidirectional AL adapter (LWORD data) into a unidirectional AS adapter (SINT data). It enables the seamless integration of components that work with different data types by converting the signals and passing them on via the respective adapter interfaces.

Interface Structure

Event Inputs

- None

Event Outputs

- None

Data Inputs

- None

Data Outputs

- None

Adapter

Type	Name	Direction	Description
Socket	AL_IN	Input	Unidirectional AL adapter that provides LWORD data.
Plug	AS_OUT	Output	Unidirectional AS adapter that outputs the converted SINT data.

Functionality

The function block receives events and LWORD data via the `AL_IN` socket. Internally, a function block of type `F_LWORD_TO_SINT` (from the IEC 61131 library) is used to convert the input data (LWORD) to the output data type SINT. The event control works as follows:

- An incoming event at socket `AL_IN.E1` triggers the input `REQ` of the internal conversion module.
- After successful conversion, the output `CNF` of the converter signals an event, which is then passed to plug `AS_OUT.E1`.
- Simultaneously, the converted data value is transferred from the output `OUT` of the converter to the data output `AS_OUT.D1`.

The entire process is event-driven: Only an event at `AL_IN` triggers the conversion and an event at `AS_OUT`.

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- The function block (FB) is implemented as a composite and uses the predefined conversion function `F_LWORD_TO_SINT`, ensuring high reusability and standards compliance.
- Conversion is **unidirectional**, only from LWORD to SINT; a reverse direction is not supported.
- Event forwarding is direct: An event at the input results in exactly one event at the output.
- The data width is reduced from 64 bits (LWORD) to 8 bits (SINT), therefore only values in the range [−128, 127] can be represented. If this range is exceeded, the internal converter behaves according to the IEC 61131 definition (e.g., overflow indication or modulo behavior).

State Overview

The FB does not have its own state machine. Its behavior is purely combinatorial with event-driven triggering. There is no internal state; The conversion always occurs based on the current input data at an event.

Application Scenarios

- **System Integration**: If a software system or device uses LWORD interfaces (AL adapter), but another part of the system only supports AS adapters (SINT), this function block can serve as a bridge.
- **Data Reduction**: Targeted conversion of 64-bit data to an 8-bit value, e.g., for transfer to simple actuators or displays that only process SINT.
- **Protocol Adaptation**: In heterogeneous automation networks where different adapter types are used (unidirectionally), this function block simplifies the conversion at a single logical level.

Comparison with Similar Function Blocks

Function Block	Input Type	Output Type	Special Feature
AL_TO_AS	LWORD (Adapter)	SINT (Adapter)	Composite; uses `F_LWORD_TO_SINT`
`F_LWORD_TO_SINT`	LWORD (Data)	SINT (Data)	Data converter only, no adapter integration
`F_LWORD_TO_DINT`	LWORD	DINT	Conversion to 32-bit integer
`F_SINT_TO_LWORD`	SINT	LWORD	Reverse direction (if available)

The **AL_TO_AS** stands out due to its adapter-to-adapter interface, which enables a direct connection between two different adapter types without the need for additional manual wiring of individual events and data.

Conclusion

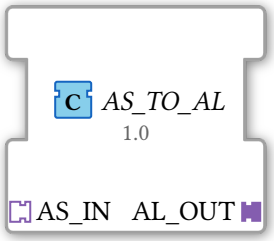
The **AL_TO_AS** function block is a practical, standards-compliant converter module for unidirectional adapters. It encapsulates the necessary type conversion from LWORD to SINT and simplifies integration into existing 4diac projects. Thanks to its composite structure and the use of an established conversion function, it is reliable, easy to understand, and extensible. It is particularly suitable for scenarios where data widths need to be reduced or interfaces adapted.

🌐 Related topic subpages on ms-muc-docs.de

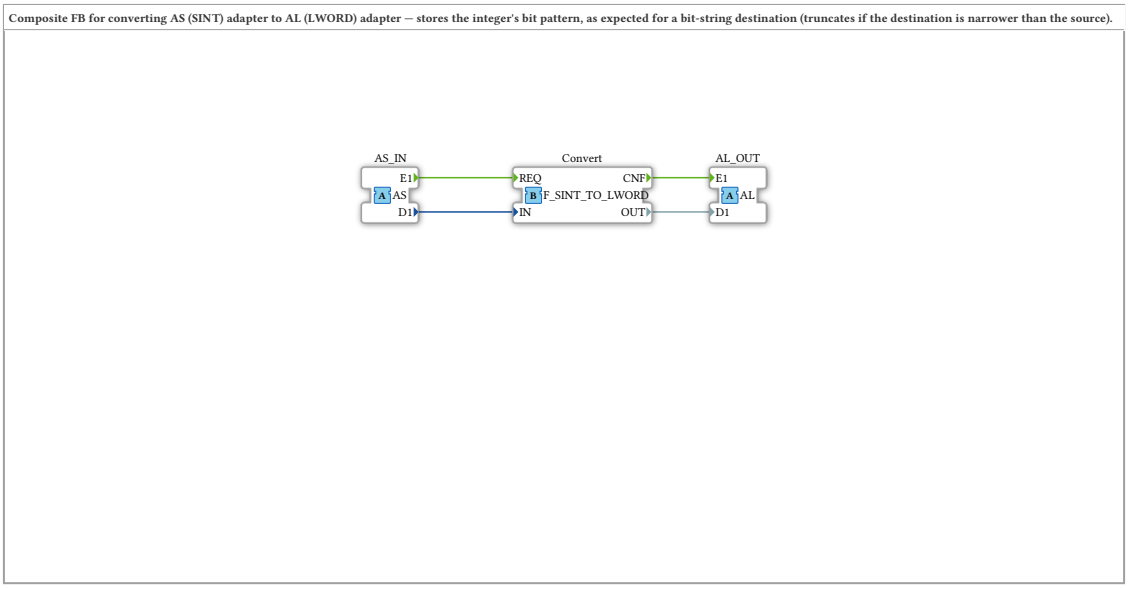
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AS_TO_AL



SINT Adapter Input – *AS* *AL* – LWORD Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block **AS_TO_AL** is a composite block that converts a SINT adapter (AS) into an LWORD adapter (AL). It serves as a bridge between different adapter types in a unidirectional communication path. The block encapsulates the conversion logic and makes it usable for the user as a single, integrated unit.

Interface Structure

The block has **no direct event or data interfaces** at the top level. All input and output is handled via adapters.

Event Inputs

- None (Events are passed in via the AS_IN socket adapter)

Event Outputs

- None (Events are output via the AL_OUT plug adapter)

Data Inputs

- None (Data is provided via the AS_IN socket adapter)

Data Outputs

- None (Data is provided via the AL_OUT plug adapter)

Adapters

Name	Type	Direction	Description
AS_IN	adapter::types::unidirectional::AS	Socket	SINT adapter (input)
AL_OUT	adapter::types::unidirectional::AL	Plug	LWORD adapter (output)

Functionality

The function block internally uses the IEC 61131 conversion block **F_SINT_TO_LWORD**. As soon as an event arrives at the event output **E1** of the socket adapter **AS_IN**, it is forwarded to the input **REQ** of the conversion block. This converts the SINT data value from **AS_IN.D1** into an LWORD value and outputs it to **AL_OUT.D1**. After successful conversion, an acknowledgment event is sent to the event input **E1** of the plug adapter **AL_OUT**.

Technical Specifications

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Standard Compliance**: Complies with IEC 61499-2.
- **Copyright**: The function block is subject to the Eclipse Public License 2.0 (Copyright 2026 HR Agrartechnik GmbH).
- **Version**: 1.0, created by Franz Höpfinger on February 17, 2026.
- **Package Structure**: Organized in the package `adapter::conversion::unidirectional`.
- **Dependencies**: Uses the function block `iec61131::conversion::F_SINT_TO_LWORD`.

State Overview

Since this is a composite function block, it does not have its own state machine. The state logic is implemented by the integrated function block **F_SINT_TO_LWORD**, which performs a simple conversion internally without delay or error states. Execution is strictly event-driven.

Application Scenarios

- **Adapter Conversion**: Connecting components that provide a SINT adapter with components that expect an LWORD adapter.
- **Data Adaptation**: Used in heterogeneous automation systems where different data types need to be harmonized at the adapter level.
- **Modular Reuse**: The function block can be included in libraries for unidirectional conversions.

Comparison with Similar Function Blocks

Similar adapter conversion blocks exist for other data types (e.g., BOOL, INT, DWORD). The **AS_TO_AL** block is specifically designed for converting **SINT** (Signed Short Integer, 8-bit) to **LWORD** (Long Word, 64-bit). Unlike generic function blocks, the conversion is performed directly at the adapter level, simplifying network wiring.

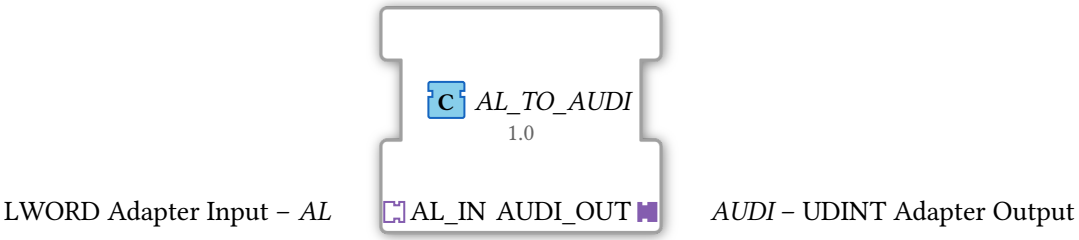
Conclusion

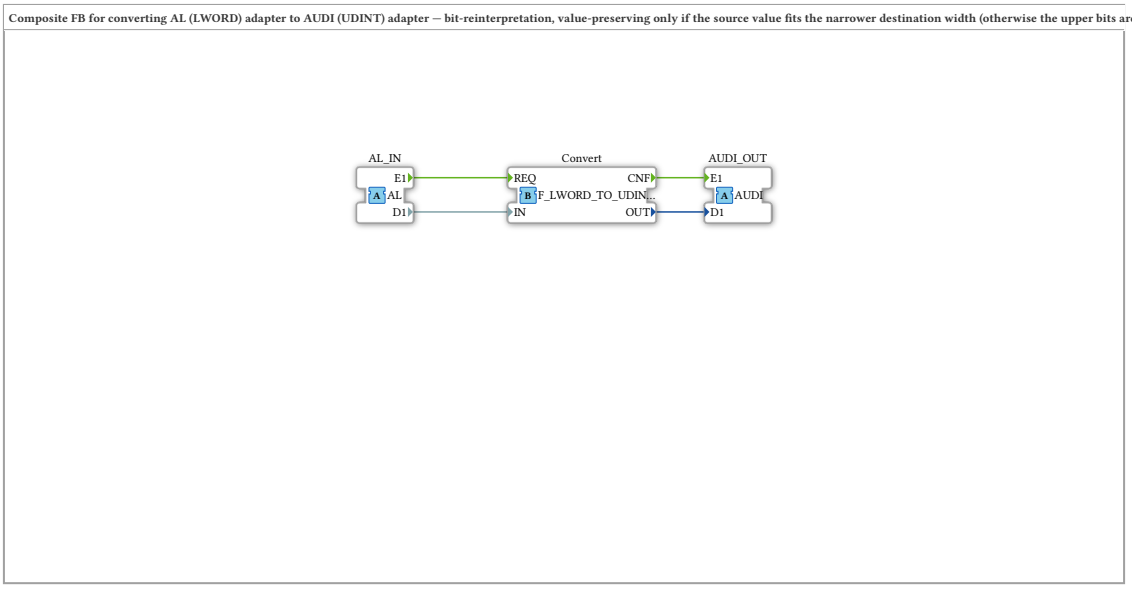
The **AS_TO_AL** composite block offers a compact and standards-compliant solution for converting a SINT adapter to an LWORD adapter. Encapsulating the conversion logic improves the clarity of the application network and promotes reusability. This block is particularly well-suited for modular automation projects where different data types need to be combined at the adapter level.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AL_TO_AUDI





• – ○ • – ○ • – ○ •

Introduction

The function block **AL_TO_AUDI** is a composite block that converts an LWORD adapter input (type `AL`) into a UDINT adapter output (type `AUDI`). It encapsulates the IEC 61131-3 conversion function `F_LWORD_TO_UDINT` and provides a clean, adapter-based interface for unidirectional data flows.

Interface Structure

Event Inputs

The function block does not have direct event inputs. Event control is handled entirely via the integrated socket adapter `AL_IN`.

Event Outputs

The function block (FB) does not have direct event outputs. Events are passed to subsequent logic via the integrated plug adapter `AUDI_OUT`.

Data Inputs

The FB does not have direct data inputs. The LWORD data to be converted is received via the socket adapter `AL_IN`.

Data Outputs

The FB does not have direct data outputs. The converted UDINT data is output via the plug adapter `AUDI_OUT`.

Adapters

Adapter Type	Name	Description
Socket	`AL_IN`	LWORD adapter input (Type: `adapter::types::unidirectional::AL`)
Plug	`AUDI_OUT`	UDINT adapter output (Type: `adapter::types::unidirectional::AUDI`)

Functionality

1. An incoming event at socket `AL_IN` (via its event output `E1`) triggers the conversion instance `Convert` (Type `F_LWORD_TO_UDINT`).
2. Simultaneously, the LWORD data value from `AL_IN.D1` is transferred to the input `IN` of `Convert`.
3. The conversion function converts the LWORD value into a UDINT value.

4. After successful conversion, `Convert` signals this via its event output `CNF`, causing the plug `AUDI_OUT` to receive an event at its event input `E1`.
5. The converted UDINT value is passed from the output `OUT` of `Convert` to the data input `D1` of `AUDI_OUT` and is thus available at the output adapter.

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- **Composite Architecture**: The function block consists solely of the internal use of the function block `F_LWORD_TO_UDINT` and the adapters. It has no independent state logic or additional control.
- **Unidirectional Adapters**: Both `AL_IN` and `AUDI_OUT` are unidirectional adapters (input to output).
- **IEC 61131-3 Compliance**: The conversion uses a standardized IEC 61131-3 function, which increases reusability and interchangeability.

State Overview

The function block has no states of its own. Its behavior is purely combinatorial and is entirely determined by the internal conversion function – the conversion occurs per event without internal storage.
Unidirectional Adapters: ** ...

Application Scenarios

- **System Integration**: Connecting components that use different data types for addresses, counters, or configuration values (e.g., LWORD for wide addresses, UDINT for embedded systems).
- **Adapter Architectures**: Use in modular automation solutions where standardized interfaces are provided via adapters.
- **Data Preparation**: Converting LWORD signals from sensors or bus systems to UDINT for further processing in controllers.

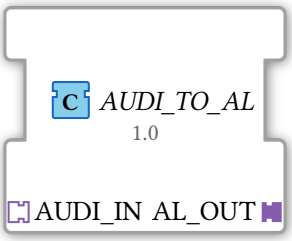
Comparison with Similar Components

- **Simple Type Converters** (e.g., `F_LWORD_TO_UDINT`): These work directly with input/output variables, while `AL_TO_AUDI` encapsulates the conversion in an adapter interface, thus offering a higher level of abstraction.
- **Other adapter converters** (e.g., `UDINT_TO_DINT`): Similar concepts, but for different data types and directions. `AL_TO_AUDI` is specifically for the unidirectional conversion of LWORD to UDINT.

Conclusion

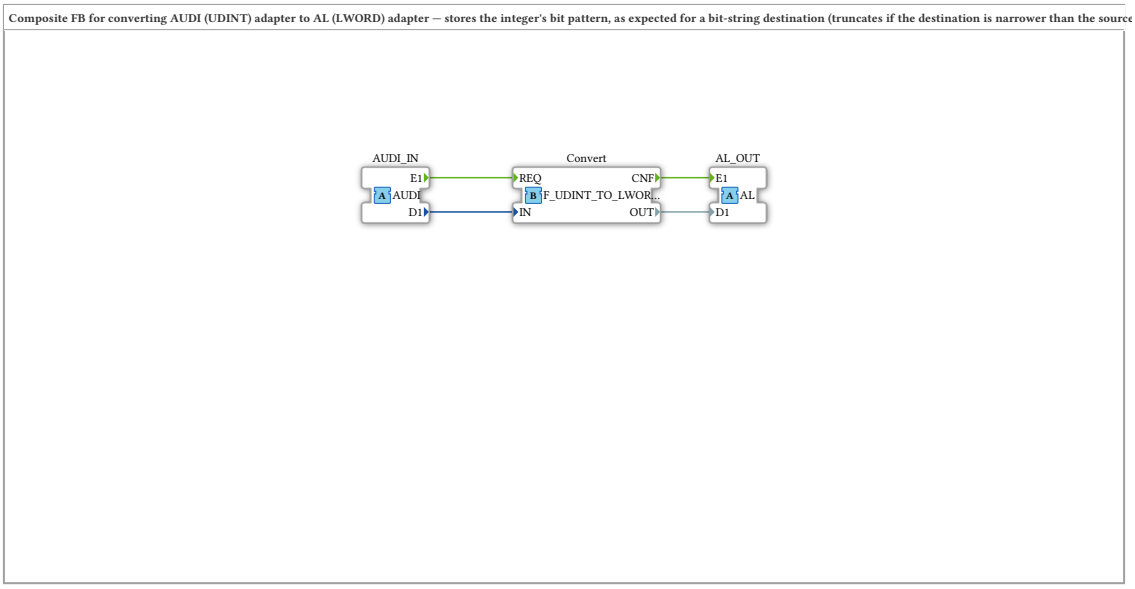
The function block `AL_TO_AUDI` is a compact and reusable building block for converting LWORD to UDINT data via standardized adapters. It simplifies the integration of components with different data types and promotes a modular, adapter-based architecture in automation technology.

AUDI_TO_AL



UDINT Adapter Input – *AUDI*

AL – LWORD Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The **AUDI_TO_AL** function block is a composite function block that converts an **AUDI** adapter (UDINT data format) into an **AL** adapter (LWORD data format). It serves as an interface converter between different adapter types in automation systems based on the IEC 61499 standard.

Interface Structure

Event Inputs

The function block does not have its own event inputs. Event-driven processing is handled via the **AUDI_IN** socket:

- **E1**: Event input of the AUDI adapter – triggers the conversion.

Event Outputs

The function block does not have its own event outputs. The result is passed via the **AL_OUT** plug:

- **E1**: Event output of the AL adapter – signals completion of the conversion.

Data Inputs

This function block does not have its own data inputs. The data to be converted is provided via the **AUDI_IN** socket:

- **D1**: Data input of the AUDI adapter (Type: UDINT) – the value to be converted.

Data Outputs

This function block does not have its own data outputs. The converted result is provided via the **AL_OUT** plug:

- **D1**: Data output of the AL adapter (Type: LWORD) – the converted value.

Adapter

Name	Direction	Type	Description
AUDI_IN	Socket	`adapter::types::unidirectional::AUDI`	UDINT adapter as input interface
AL_OUT	Plug	`adapter::types::unidirectional::AL`	LWORD adapter as output interface

Functionality

This function block is a pure encapsulation of the IEC 61131 function `F_UDINT_TO_LWORD`.

When an event occurs at **AUDI_IN.E1**, the internal function block **Convert** (type `F_UDINT_TO_LWORD`) is activated via its event input **REQ**. Simultaneously, the UDINT value present at **AUDI_IN.D1** is passed to the **IN** data input of **Convert**. Upon completion of the conversion, **Convert** signals its completion via its **CNF** event output. This event is forwarded directly to **AL_OUT.E1**, and the converted LWORD value is output by **Convert.OUT** to **AL_OUT.D1**.

The entire process is event-driven and requires no additional runtime logic within the composite function block.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- The function block is implemented as a **composite function block** and uses only the standardized IEC 61131 conversion function `F_UDINT_TO_LWORD`.
- The conversion is performed directly: **UDINT** (32-bit unsigned integer) is mapped to **LWORD** (64-bit logical word).

Note: When converting a 32-bit value to a 64-bit word, the upper 32 bits are set to zero.

- This function block is licensed under the **Eclipse Public License 2.0**.
- The interfaces (adapters) are unidirectional: Data flows only from the socket to the plug.

State Overview

Since this is a composite function block without its own state machine, **AUDI_TO_AL** does not have an internal state machine. Its behavior is entirely determined by the interconnected function block `F_UDINT_TO_LWORD`, which implements a simple REQ/CNF handshake.

Application Scenarios

- **Adapter Conversion** in IEC 61499 systems where an existing AUDI adapter (UDINT-based) needs to be connected to an AL adapter (LWORD-based) – e.g., to connect sensor data to a 64-bit data processing chain.
- **Interface Adaptation** when replacing hardware components or migrating from 32-bit to 64-bit automation solutions.
- **Prototypical Implementation** in agricultural technology or other areas where UDINT and LWORD data formats are used interchangeably.

Comparison with Similar Components

Unlike generic converters such as `UDINT_TO_LWORD` (an atomic component), **AUDI_TO_AL** offers a pre-configured adapter coupling, freeing developers from manually connecting event and data lines. Similar composite components exist, for example, for `UDINT_TO_DWORD` or `LREAL_TO_LWORD`, but these are specific to different target adapters.

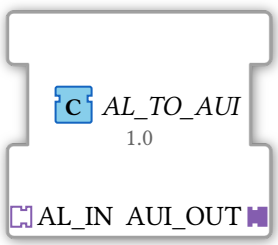
Conclusion

The **AUDI_TO_AL** function block is a compact, reusable component for cleanly separating adapter types. By encapsulating the conversion in a composite, it simplifies the maintainability and readability of applications based on different data format adapters.

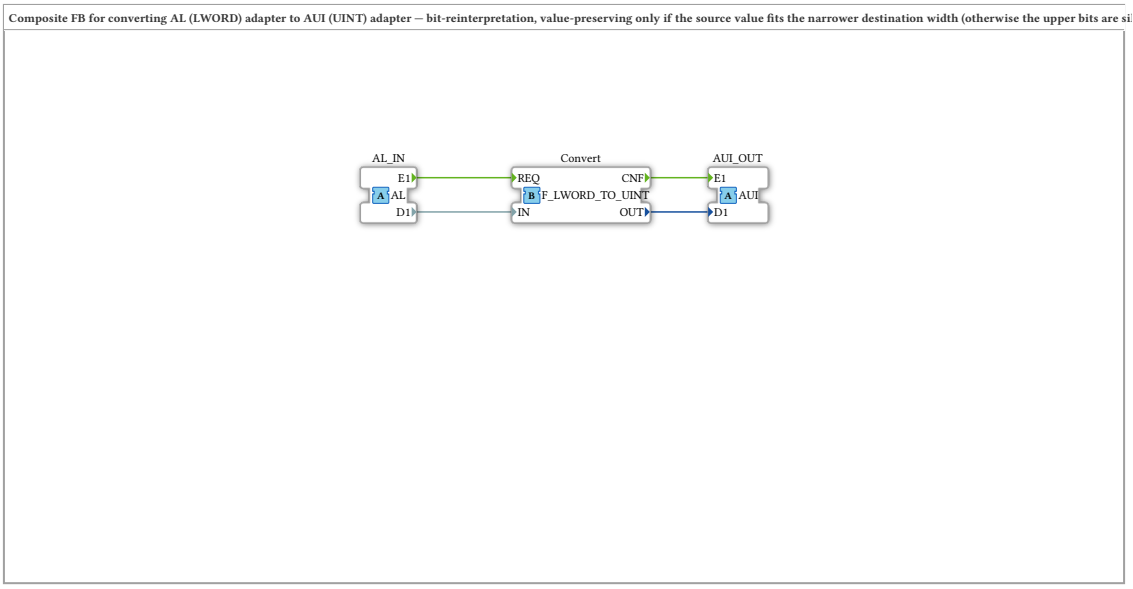
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AL_TO_AUI



LWORD Adapter Input – *AL* *AUI* – UINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The **AL_TO_AUI** function block is a composite component that converts an LWORD adapter (AL) into a UINT adapter (AUI). It enables the seamless integration of LWORD-based interfaces into systems that expect UINT data by encapsulating the conversion within a standardized adapter structure.

Interface Structure

Event Inputs

No standalone event inputs. The **AL_IN** socket provides the trigger signal for the conversion via its event output `E1`.

Event Outputs

No standalone event outputs. The plug **AUI_OUT** provides confirmation of the completed conversion via its event input `E1`.

Data Inputs

No standalone data inputs. The socket **AL_IN** provides the value to be converted to `LWORD` via its data output `D1`.

Data Outputs

No standalone data outputs. The plug **AUI_OUT** provides the converted `UINT` value via its data input `D1`.

Adapter

Type	Direction	Name	Description
Socket	Input	**AL_IN**	Adapter type `adapter::types::unidirectional::AL` – provides an LWORD data and an event to trigger the conversion.
Plug	Output	**AUI_OUT**	Adapter type `adapter::types::unidirectional::AUI` – receives the converted UINT data and forwards the confirmation event.

Functionality

The module consists of an internal network with a single conversion module `F_LWORD_TO_UINT` from the IEC 61131 library. The process is strictly event-driven:

1. An event at socket `AL_IN.E1` is forwarded to the `REQ` input of the conversion module.
2. The conversion block reads the date from `AL_IN.D1` (type `LWORD`) and converts it to a `UINT` value according to IEC 61131 rules.

3. After the conversion is complete, the block generates an event at its `CNF` output, which is directly passed to the plug `AUI_OUT.E1`.
4. Simultaneously, the converted value is passed from the data output `OUT` to the data input `AUI_OUT.D1`.

This completes the entire conversion in a single, unidirectional pass.
...

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- **Composite Structure**: The function block (FB) does not contain its own state machine but delegates all functionality to the embedded conversion block.
- **Unidirectional Adapters**: Both the input and output adapters are of type `unidirectional`. Only forward conversion takes place; reverse conversion and status feedback are not supported.
- **Package Assignment**: The FB is located in the package `adapter::conversion::unidirectional`, indicating its use in modular adapter chains.
- **Reusability**: The function block can be used as a universal converter in any IEC 61499 application that requires LWORD-to-UINT conversion via adapters.

State Overview

Since this is a composite block without its own state machine, the state logic is entirely determined by the internal block `F_LWORD_TO_UINT`. This block operates according to a simple REQ/CNF scheme:

- **Waiting for REQ**: Inactive, no event at the input.
- **Conversion active**: The conversion is performed upon receipt of `REQ`.
- **Output CNF**: Upon completion, `CNF` is set and the output data is updated.

A detailed state description of the conversion block can be found in the IEC 61131 documentation.

Application Scenarios

- **Sensor Data Connection**: A sensor provides values in LWORD format (e.g., 64-bit counter readings), which are fed into the system via an AL adapter. The `AL_TO_AUI` function block converts these values into UINT (16-bit) for transmission to a controller with a UINT interface.
- **Protocol Conversion**: In modular adapter chains, the function block serves as an intermediate stage to switch from LWORD-based to UINT-based subsystems without requiring changes to the original adapter logic.
- **Data Reduction**: If the upper 48 bits of an LWORD value are not needed, the function block can selectively extract only the lower 16 bits as UINT (according to IEC 61131 conversion rules).

Comparison with Similar Function Blocks

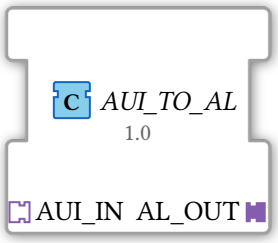
Function Block	Output Type	Special Feature
`AL_TO_AUI`	UINT	Direct Adapter Converter, Unidirectional, Composite
`AL_TO_AUD`	DINT	Converts LWORD to DINT (32-bit)
`AUI_TO_AL`	LWORD	Reverse conversion from UINT to LWORD (not included, would need to be implemented separately)

Compared to directly using `F_LWORD_TO_UINT`, `AL_TO_AUI` offers the advantage of easy integration into adapter interfaces without the need to manually establish event and data connections.

Conclusion

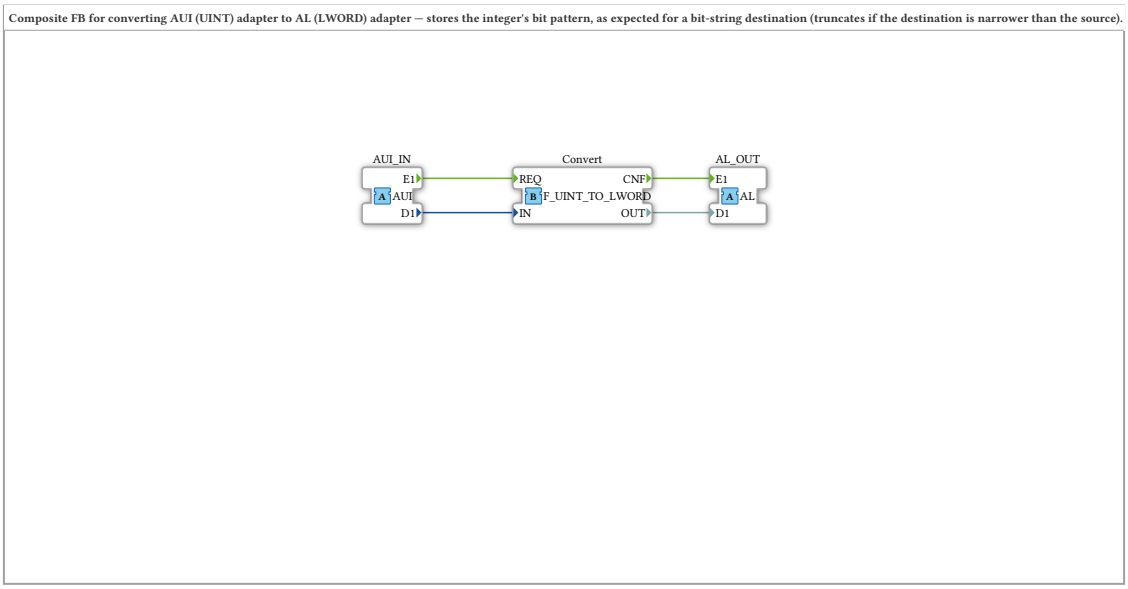
The **AL_TO_AUI** is a convenient, standardized adapter converter for converting LWORD to UINT. Thanks to its composite structure and the use of proven IEC 61131 conversion logic, it integrates seamlessly into modular adapter chains. It is particularly suitable for applications where heterogeneous data types need to be connected via uniform adapter interfaces.

AUI_TO_AL



UINT Adapter Input – *AUI*

AL – LWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_TO_AL** is a composite function block that enables the conversion of an AUI adapter (based on the UINT data type) to an AL adapter (based on the LWORD data type). It encapsulates the standardized conversion function `F_UINT_TO_LWORD` and provides a simple, event-driven interface for unidirectional data transfer.

Interface Structure

Event Inputs

The block does not have explicit individual event inputs, as event control is handled by the connected adapters. The **E1** event of the **AUI_IN** socket is internally routed to the `REQ` input of the conversion block.

Event Outputs

The function block outputs the event **E1** via the **AL_OUT** plug after the conversion is complete (signal from `CNF` of the internal function block).

Data Inputs

The data input is provided exclusively via the socket:

- **AUI_IN.D1** (Type: UINT) – The value to be converted.

Data Outputs

The data output is provided via the plug:

- **AL_OUT.D1** (Type: LWORD) – The converted value (LWORD).

Adapter

Type	Name	Direction	Description
AUI	**AUI_IN**	Input (Socket)	Input adapter for the UINT value and the trigger event.
AL	**AL_OUT**	Output (Plug)	Output adapter for the converted LWORD value and the acknowledgment event.

Functionality

The function block operates in an event-driven manner:

1. An incoming event at socket **AUI_IN.E1** (triggered by the connected transmitter) activates the internal block F_UINT_TO_LWORD via its event input REQ.
2. The value of **AUI_IN.D1** (UINT) is passed to the data input IN of the conversion block.
3. The conversion is performed according to the IEC 61131 standard: A UINT value is converted into an LWORD value (the upper 48 bits are padded with zeros).
4. After the conversion is complete, the result is passed to **AL_OUT.D1** (LWORD).
5. Simultaneously, the acknowledgment event from CNF is sent to **AL_OUT.E1** to signal the availability of the new value to the receiver.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Function Block (FB)**: The implementation hides the internal logic and facilitates reuse in larger adapter chains.
- **Standard Conversion**: Uses the IEC 61131-compliant function F_UINT_TO_LWORD, making it portable and platform-independent.
- **Unidirectional Transmission**: The adapters are designed for one-way data flow only (no return channel).
- **No State Logic**: The function block does not have its own state machine; sequence control is achieved purely through event chaining.

State Overview

The **AUI_TO_AL** function block does not implement explicit states, as it is a purely event-driven mapping. The following deterministic sequence applies:

- **Wait**: The function block waits for an event at the input socket.
- **Convert**: After an event is received, the conversion is performed.
- **Output**: Upon completion, the result is output, and the function block returns to the wait state.

Application Scenarios

- **Protocol Adaptation**: When a data provider supplies UINT values via an AUI adapter, but the consumer expects an AL adapter (LWORD).
- **System Integration**: In distributed automation systems with different data types at the communication layer.
- **Adapter Chaining**: As an intermediate link in an adapter cascade for stepwise type conversion.

Comparison with Similar Function Blocks

In the 4diac environment, other conversion adapters exist, such as AUI_TO_AB (UINT to BYTE) or AUI_TO_AW (UINT to WORD). **AUI_TO_AL** differs in its target size of LWORD (64 bits), which is used for time-critical applications or for combining multiple data words into a single, contiguous value. It does not replace a universal type conversion but offers a dedicated, event-driven adaptation for the aforementioned adapter types.

Conclusion

The **AUI_TO_AL** function block provides a clean, standardized solution for converting a UINT-based adapter (AUI) to an LWORD-based adapter (AL). It is easy to use, event-driven, and utilizes a proven IEC 61131 conversion function. Thanks to its composite nature, it integrates seamlessly into existing 4diac networks and facilitates the integration of different data widths in automation technology.

🌐 Related topic subpages on ms-muc-docs.de

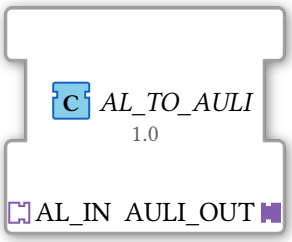
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

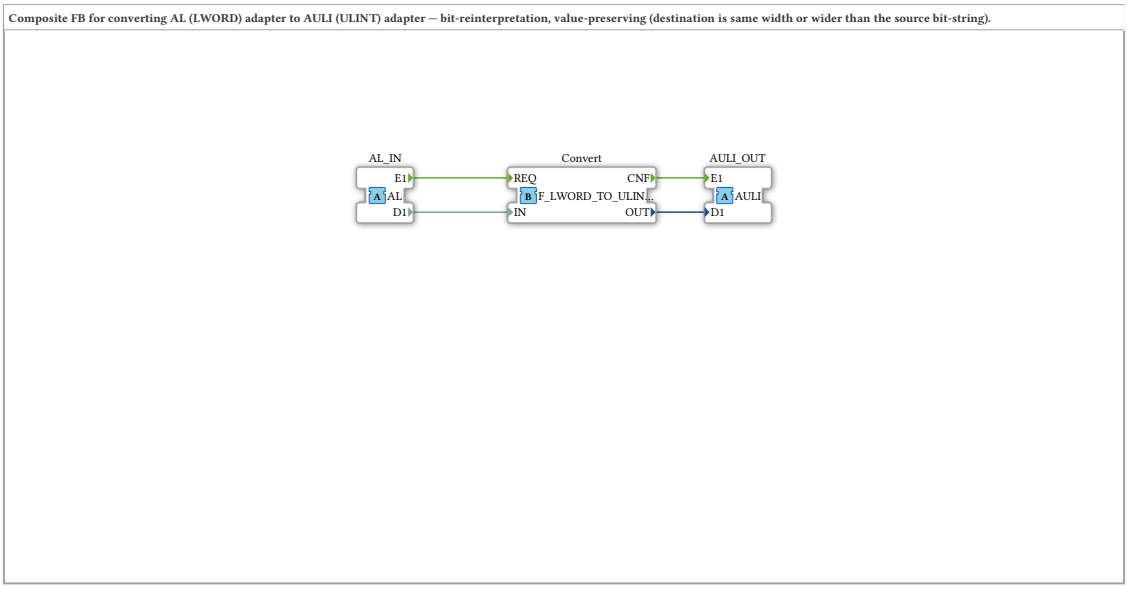
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AL_TO_AULI



LWORD Adapter Input – *AL* *AULI* – ULINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AL_TO_AULI** is a composite function block (FB) that performs a data type conversion from an **AL adapter** (type `LWORD`) to an **AULI adapter** (type `ULINT`). It serves as a bridge between components that work with different data types and enables seamless integration into IEC 61499-based systems.

Interface Structure

Event Inputs

The **E1** event is received via the **AL_IN** socket (adapter `AL`). This event triggers the conversion of the incoming LWORD value.

Event Outputs

The **E1** event is sent via the **AULI_OUT** plug (adapter `AULI`) as soon as the conversion is complete and the ULINT value is available at the output.

Data Inputs

Name	Adapter Port	Data Type	Description
AL_IN	D1	LWORD	Input value in LWORD format

Data Outputs

Name	Adapter Port	Data Type	Description
AULI_OUT	D1	ULINT	Converted Value in ULINT Format

Adapter

Type	Direction	Name	Description	
`AL`	Socket	AL_IN	Input adapter with LWORD data	
`AULI`	Plug	AULI_OUT	Output adapter with ULINT data	

Functionality

The function block (FB) performs the conversion using an internal block `F_LWORD_TO_ULINT` from the library `iec61131::conversion`.

- As soon as an event **E1** arrives at the input adapter **AL_IN**, it is forwarded to the input **REQ** of the internal conversion block.
- Simultaneously, the data value **D1** (LWORD) is passed to the **IN** input of the converter.
- After successful conversion, the internal function block signals this via its **CNF** output, which triggers the **E1** event of the **AULI_OUT** output adapter.
- The converted ULINT value is then available at the **D1** data output of the output adapter.

Technical Specifications

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **License:** This function block is provided under the **Eclipse Public License 2.0**.
- **Version:** 1.0 (as of 2026-02-17)
- **Dependencies:** It requires the function `F_LWORD_TO_ULINT` from the `iec61132::conversion` library.
- **Portability:** As a composite function block, it is platform-independent and can be used in any 4diac IDE environment.
- **Type Hash:** The function block has an attribute `eclipse4diac::core::TypeHash` for unique identification (value empty).

State Overview

The function block **AL_TO_AULI** does not have its own state machine – it delegates all conversion logic to the internal function block `F_LWORD_TO_ULINT`. This makes it behave like a transparent conversion layer that only becomes active upon an incoming event and returns to its idle state after the conversion.

Application Scenarios

- **Data Type Conversion** in PLC or automation systems when an LWORD signal (e.g., from a communication module) needs to be converted into a ULINT value for subsequent processing (e.g., counters or arithmetic).
- **Adapter bridge** between function blocks that work exclusively with the AULI adapter and those that use the AL adapter.
- **Migration** of existing LWORD-based components to ULINT-based systems without changing the signal chain.

Comparison with similar function blocks

- **F_LWORD_TO_ULINT (direct converter):** This pure function block performs the same conversion, but without adapter interfaces. It is suitable for simple data flow models, while **AL_TO_AULI** is specifically optimized for use in adapter-based architectures.
- **Other adapter converters (e.g., AL_TO_UDINT, AL_TO_DWORD):** Similar function blocks exist for other target types (UDINT, DWORD). They differ only in the output data type but follow the same design pattern.

Conclusion

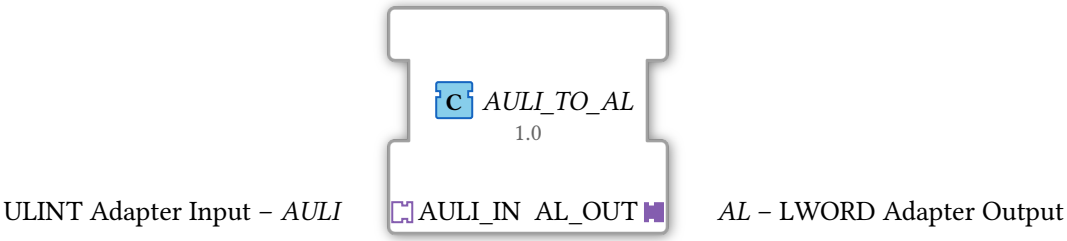
The **AL_TO_AULI** function block offers a simple and robust way to convert LWORD data to ULINT data via adapters. By encapsulating the conversion logic in a composite function block, the interface remains clear and reusable. It is particularly suitable for modular IEC 61499 projects where adapters are used as standardized communication interfaces.

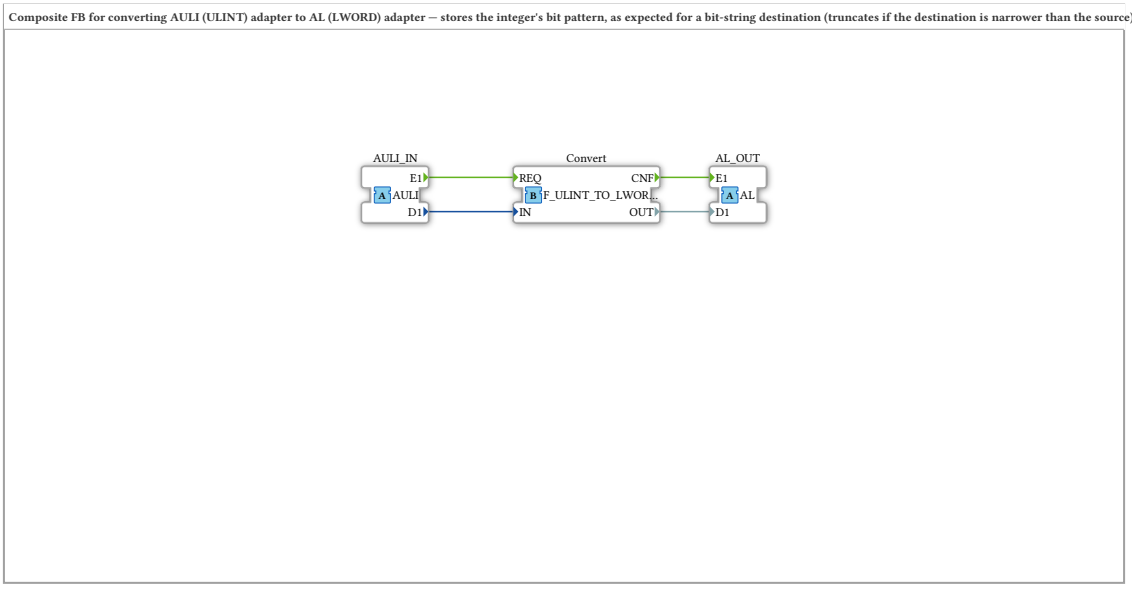
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AULI_TO_AL





• – ○ • – ○ • – ○ •

Introduction

The function block `AULI_TO_AL` is a composite function block for converting an AULI adapter (data type ULINT) into an AL adapter (data type LWORD). Internally, it uses the IEC 61131 conversion function `F_ULINT_TO_LWORD` and is intended for use in unidirectional adapter structures.

Interface Structure

Event Inputs

Name	Description
AULI_IN.E1	Event input via the AULI_IN socket. Starts the conversion.

Event Outputs

Name	Description
AL_OUT.E1	Event output via the AL_OUT plug. Signals the completion of the conversion.

Data Inputs

Name	Data Type	Description
AULI_IN.D1	ULINT	Input value to be converted to LWORD.

Data Outputs

Name	Data Type	Description
AL_OUT.D1	LWORD	Converted output value.

Adapter

Name	Type	Direction	Description
AULI_IN	AULI (ULINT)	Socket	Unidirectional input adapter.
AL_OUT	AL (LWORD)	Plug	Unidirectional output adapter.

Functionality

This function block operates as a simple forwarder: An incoming event at `AULI_IN.E1` triggers the internal function block `F_ULINT_TO_LWORD`, which reads the current value of `AULI_IN.D1` and converts it into an LWORD value. After the conversion is complete, the result value is output to `AL_OUT.D1`, and a completion event is simultaneously sent to `AL_OUT.E1`. The internal logic is limited to the data and event connections between the socket and the internal converter.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Implementation**: The function block does not contain its own execution control logic (ECC) but consists exclusively of a linked function block from the IEC 61131 library.
- **Adapter Architecture**: Both the input and output are designed as unidirectional adapters (socket/plug), enabling loose coupling and easy reuse in various applications.
- **Type Conversion**: The conversion from ULINT (unsigned 64-bit integer) to LWORD (64-bit bit field) is performed according to the IEC 61131 standard.

State Overview

This function block does not have its own state machine. All functionality is implemented by the internal function block `F_ULINT_TO_LWORD`, which operates as a stateless function. Event control is handled exclusively via the connections defined in the netlist.

Application Scenarios

- Integration of sensor systems that deliver ULINT data via an AULI adapter into controllers that expect LWORD data via an AL adapter.
- Data type migration or protocol adaptation in existing automation solutions.

- Uniform adapter interfaces for libraries that represent different physical quantities or address spaces.

Comparison with Similar Function Blocks

Unlike generic converter function blocks (e.g., `ULINT_TO_LWORD`), `AULI_TO_AL` is specifically designed for adapter-based data exchange. Similar adapter converters exist for other data types (e.g., `BOOL_TO_BYTE`, `INT_TO_DINT`), but they differ in the adapter types used and the internal conversion function. This function block adds a clean, typed interface for ULINT↔LWORD to the library.

Conclusion

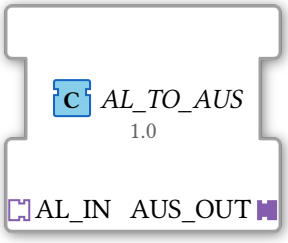
AULI_TO_AL` is a simple yet reliable composite function block for converting between AULI and AL adapters. Direct use of the standardized IEC 61131 function ensures high compatibility and accuracy of the conversion. The adapter interfaces enable modular integration into complex automation systems.

Numeric Index (unidirectional)

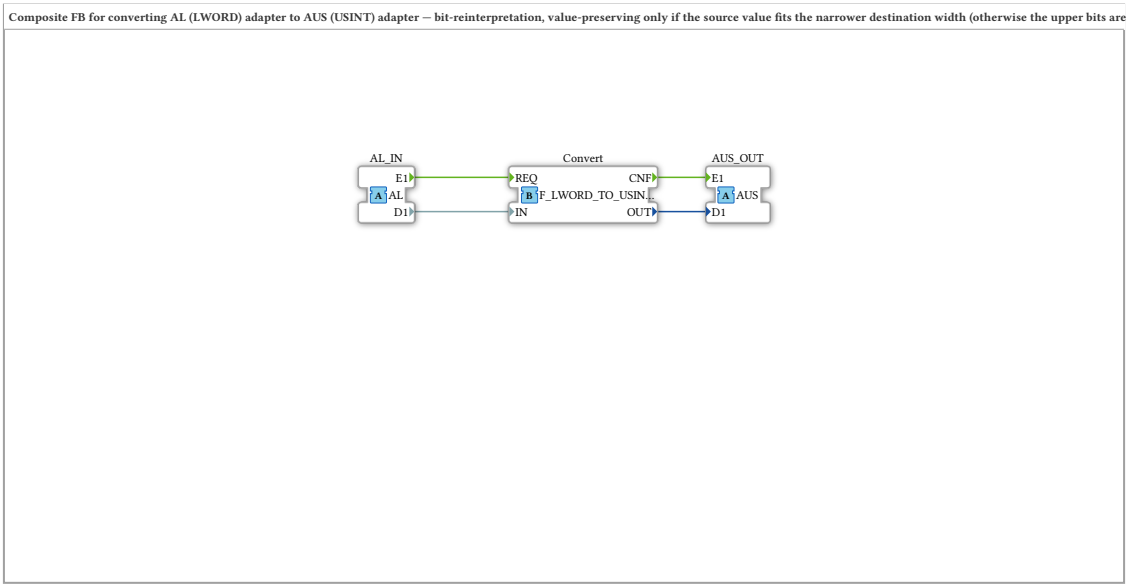
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AL_TO_AUS

LWORD Adapter Input – *AL*



AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block AL_TO_AUS converts an AL adapter (LWORD) into an AUS adapter (USINT). It is implemented as a composite function block and encapsulates the conversion logic using the function block F_LWORD_TO_USINT. Input and output are handled via unidirectional adapter interfaces.

Interface Structure

Event Inputs

The function block does not have direct event inputs. Instead, events are received via the adapter socket `AL_IN`. This socket provides the event input `E1`.

- **`AL_IN.E1`** (event): Trigger for data conversion.

Event Outputs

The FB has no direct event outputs. Events are sent via the adapter plug `AUS_OUT`. This plug provides the event output `E1`.

- **`AUS_OUT.E1`** (Event): Signals the completion of the conversion and the availability of the output value.

Data Inputs

The data is provided via the adapter socket `AL_IN`.

- **`AL_IN.D1`** (LWORD): The LWORD value to be read and converted into a USINT value.

Data Outputs

The converted data is output via the adapter plug `AUS_OUT`.

- **`AUS_OUT.D1`** (USINT): The converted USINT value.

Adapters

- **Socket `AL_IN`** (Type: `adapter::types::unidirectional::AL`): Unidirectional LWORD input adapter.
- **Plug `AUS_OUT`** (Type: `adapter::types::unidirectional::AUS`): Unidirectional USINT output adapter.

Functionality

The function block waits for an event at the input adapter `AL_IN.E1`. As soon as this event occurs, the LWORD value at `AL_IN.D1` is read and passed to the internal conversion function block `F_LWORD_TO_USINT`. This performs the conversion to a USINT value. After successful conversion, the result is forwarded to the output adapter `AUS_OUT.D1`, and an event is simultaneously sent to `AUS_OUT.E1`.

The internal connections are:

- From the event input of the socket to the `REQ` input of the converter.
- From the `CNF` output of the converter to the event output of the plug.
- From the data input of the socket to the `IN` input of the converter.
- From the `OUT` output of the converter to the data output of the plug.

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- The function block is implemented as a composite function block (FB) that utilizes the IEC 61131 conversion function `F_LWORD_TO_USINT`.
- The adapters are unidirectional, meaning data and event flows only in one direction.
- The conversion function block is from the library `iec61131::conversion`.
- Encapsulating the conversion logic in a composite function block increases its reusability and interchangeability.

State Overview

The function block does not have its own defined state machine. The internal logic consists of a single conversion function block and fixed connections. The process is purely event-driven:

1. Wait for an event at the input (`AL_IN.E1`).
2. Perform the conversion.
3. Output the result and set an event at the output (`AUS_OUT.E1`).

Application Scenarios

- Connecting sensors or actuators that provide an LWORD value to a system that expects USINT.
- Protocol implementation in automation systems where different data types need to be converted between adapters.
- Integration into larger composite function blocks that combine multiple conversion steps.

Comparison with Similar Function Blocks

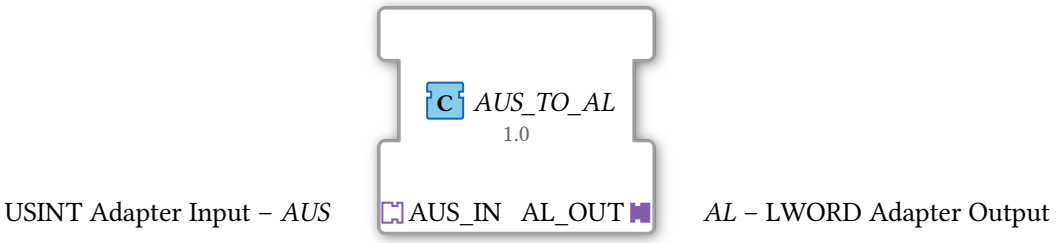
- **F_LWORD_TO_UINT**: Converts LWORD to UINT (16 bits).
- **F_LWORD_TO_DWORD**: Converts LWORD to DWORD (32 bits).

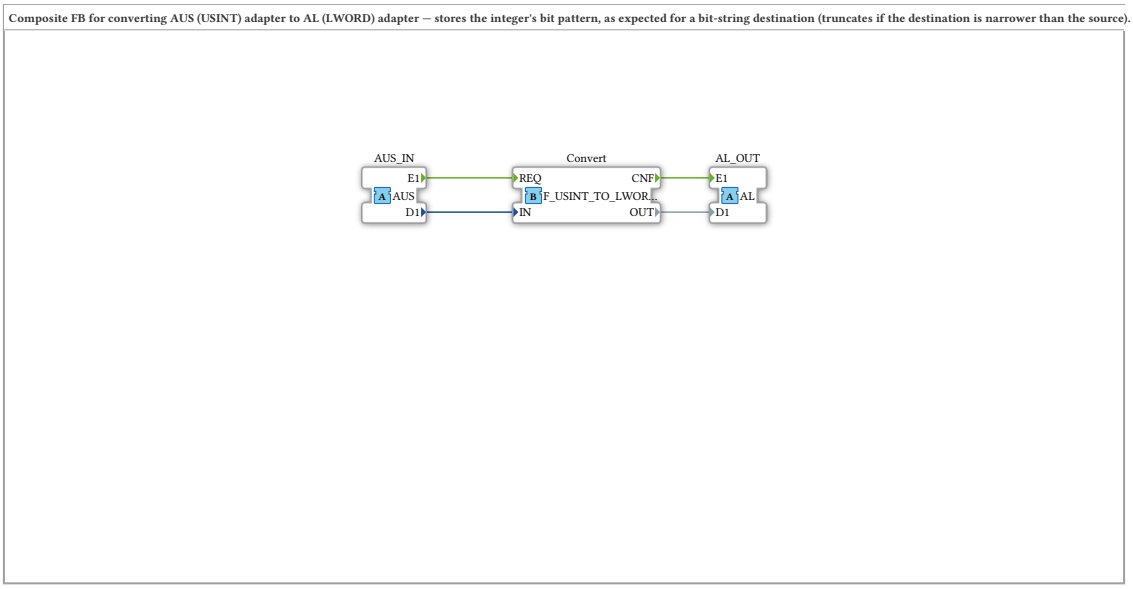
- **F_LWORD_TO_BYTE**: Converts LWORD to BYTE (8 bits), but returns a signed/unsigned value. USINT is unsigned 8 bits.
- This function block fills the gap for direct conversion to the USINT data type within the adapter context.

Conclusion

The composite function block AL_TO_AUS offers a simple and clean solution for converting an LWORD adapter input to a USINT adapter output. By using standardized conversion blocks and encapsulating the conversion in a dedicated function block type, the maintainability and readability of the overall system are improved. It is particularly suitable for scenarios where unidirectional adapter interfaces with different data types need to be connected.

OFF_TO_AL





• – ◦ • – ◦ • – ◦ •

Introduction

The **OFF_TO_AL** function block is a composite function block (FB) that converts a unidirectional **OFF** adapter (USINT data type) into a unidirectional **AL** adapter (LWORD data type). It serves as a bridge between different data formats in a 4diac IDE environment, with the conversion being performed internally by the IEC 61131 FB `F_USINT_TO_LWORD`. The function block facilitates the integration of components based on different adapter types.

Interface Structure

Because it is a composite FB, all I/O signals are provided via the two adapters. The FB itself does not have any independent event or data inputs/outputs.

Event Inputs

- No standalone event inputs.

Event Outputs

- No standalone event outputs.

Data Inputs

- No standalone data inputs.

Data Outputs

- No standalone data outputs.

Adapter

Name	Type	Direction	Description
OFF_IN	adapter::types::unidirectional::AUS	Socket (Input)	Receives the USINT value and the associated event. The adapter provides an internal event input E1 and a data input D1 (USINT).
AL_OUT	adapter::types::unidirectional::AL	Plug (Output)	Outputs the converted LWORD value. The adapter also provides an internal event output E1 and a data output D1 (LWORD).

Functionality

1. The **AUS_IN** socket receives an event at its internal input E1 and the corresponding USINT value at D1.

AL_OUT | adapter::types::unidirectional::AL | Plug (Output) | Outputs the converted LWORD value. 2. This event is internally forwarded to the **REQ** input of the conversion function block F_USINT_TO_LWORD; the USINT value is placed at its **IN** input.

1. The function block F_USINT_TO_LWORD performs the conversion from USINT (8-bit unsigned) to LWORD (64-bit unsigned).
2. After the conversion is complete, an event is generated at the **CNF** output of F_USINT_TO_LWORD, which is sent to the internal event input of the **AL_OUT** adapter (E1). Simultaneously, the converted LWORD value is placed at the data output D1 of the adapter.

The conversion is then performed by the function block F_USINT_TO_LWORD. 5. The parent function block can then retrieve the LWORD value via the **AL_OUT** plug and use the event.

The entire sequence is strictly causal: The conversion is only triggered when an event arrives at **AUS_IN**.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite FB**: The function block encapsulates the conversion logic in a reusable adapter pair.
- **Library Used**: The actual type conversion is performed by iec61131::conversion::F_USINT_TO_LWORD from the IEC 61131 standard library.
- **No State**: The function block is purely combinatorial with event-driven execution; it does not store any internal states.
- **Adapter Unidirectionality**: Both AUS and AL are defined as unidirectional adapters, meaning data flows in only one direction (input → output).

State Overview

This function block does not have its own state diagram. Its behavior is determined exclusively by the event/data flows of the embedded function block F_USINT_TO_LWORD:

- **Idle**: Waiting for an event at **OFF_IN**.
- **Conversion Active**: As soon as an event arrives, the conversion is performed. Upon completion, the output event is generated, and the function block returns to the idle state.

Application Scenarios

- **Data Format Adaptation**: A sensor or controller delivers data as a USINT value (e.g., counter readings), but the downstream component expects LWORD (e.g., for large data ranges or timestamps).

- **Adapter Bridges**: If components with different adapter types need to be connected in a 4diac application, this function block enables clean, type-safe conversion.
- **Legacy System Support**: Legacy systems that output USINT data can be integrated into modern LWORD interfaces using this function block.

Comparison with Similar Function Blocks

- **F_USINT_TO_LWORD**: A pure function block (Basic FB) without adapter encapsulation. It can be used directly in a network but requires manual cabling of events and data.
- **AUS_TO_AL**: Offers a higher level of abstraction through the use of adapters. This improves the reusability and interchangeability of components because the adapter interfaces are standardized.
- **Other Type Converters** (e.g., `INT_TO_DWORD`): Similar logic, but for different data types; this function block fills the specific gap between USINT and LWORD.

Conclusion

The **AUS_TO_AL** function block is a practical composite function block for converting USINT to LWORD data, packaged in an adapter-based interface. It utilizes the proven IEC 61131 conversion and allows for seamless integration into modular 4diac applications. Thanks to the adapter encapsulation, the interface remains clearly defined, and the block can be easily integrated into existing communication structures.

🌐 Related topic subpages on ms-muc-docs.de

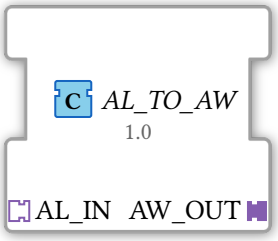
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

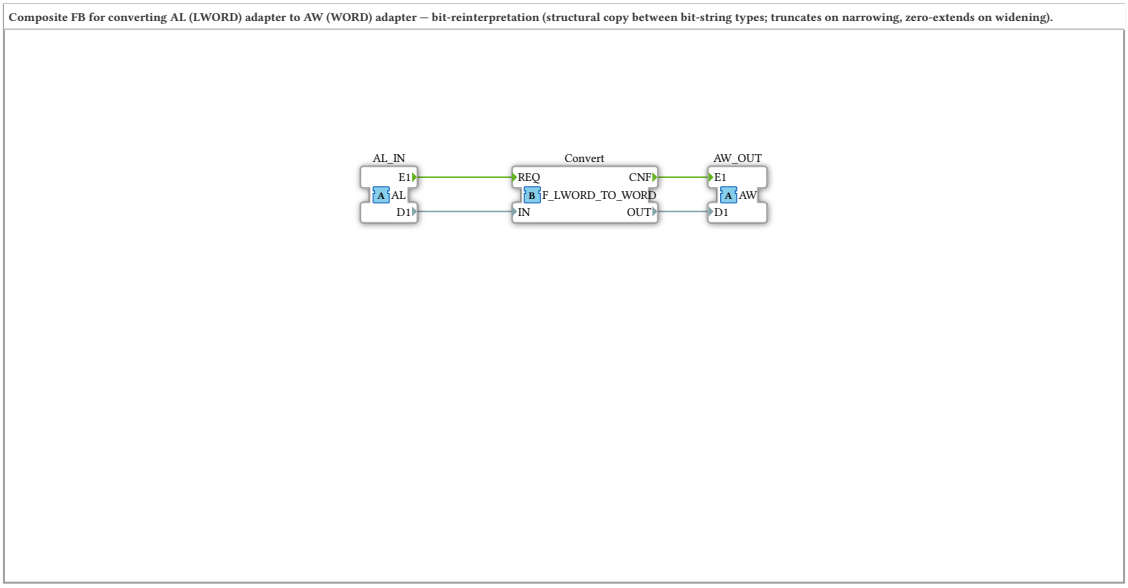
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AL_TO_AW



LWORD Adapter Input – *AL* *AW* – WORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The AL_TO_AW function block is a composite module that converts an LWORD input adapter of type AL (unidirectional) into a WORD output adapter of type AW (unidirectional). It encapsulates the conversion function and provides a simple interface for converting 64-bit data to 16-bit data in the 4diac IDE.

Interface Structure

The module has no discrete event or data ports, but only adapter interfaces (socket/plug). The following overview shows the available adapters:

Event Inputs

No separate event inputs. The event signal is routed via the AL_IN adapter (socket).

Event Outputs

No separate event outputs. The event signal is output via the AW_OUT adapter (Plug).

Data Inputs

No separate data inputs. Data is read via the AL_IN adapter.

Data Outputs

No separate data outputs. Data is output via the AW_OUT adapter.

Adapters

- **AL_IN** (Socket) – Type: `adapter::types::unidirectional::AL`

LWORD input adapter; contains one event signal and one data signal (E1, D1).

- **AW_OUT** (Plug) – Type: `adapter::types::unidirectional::AW`

WORD output adapter; Contains one event signal and one data signal (E1, D1).

Functionality

The function block operates in an event-driven manner:

1. An event at socket **AL_IN** (E1) triggers the internal conversion function block `F_LWORD_TO_WORD`.
2. Simultaneously, the LWORD data (D1) is forwarded from the input adapter to the `IN` input of the conversion function block.
3. The conversion function block performs the conversion from LWORD (64-bit) to WORD (16-bit).
4. After successful conversion, an event is generated at the `CNF` output of the conversion function block, which is then passed to the output adapter **AW_OUT** (E1).
5. The converted WORD data (16 bits) is retrieved from the `OUT` output of the conversion function block and passed to the data output of the adapter (D1).

The entire logic consists of a direct connection chain without intermediate states or delays.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Use of standardized IEC 61131 conversion** – Internally, the function block `F_LWORD_TO_WORD` from the IEC 61131 library is used.
- **Adapter-based interface** – The function block is fully integrated into 4diac's adapter concepts and can be easily integrated into existing systems with unidirectional AL and AW adapters.
- **No side effects** – The conversion occurs without storage or buffering; each incoming event triggers exactly one output.
- **Compatibility** – Suitable for applications where 64-bit LWORD data signals need to be reduced to 16-bit WORD signals (e.g., when connecting older controllers or fieldbuses).

State Overview

The function block has no internal states. It is a purely combinational transformation without sequential logic or state machines.

Application Scenarios

- **Protocol Conversion** – In automation systems that use different data word widths (e.g., converting 64-bit measured values to 16-bit outputs).
- **Interface Adaptation** – When a system provides LWORD data, but a downstream adapter only processes WORD data.
- **Testing and Simulation Tasks** – For easily converting test data between different adapter types.

Comparison with Similar Function Blocks

- **F_LWORD_TO_WORD** (direct function block) – Offers the same conversion, but without adapter encapsulation. AL_TO_AW integrates this conversion directly into an adapter-based environment.
- **Other Adapter Conversions** – Similar function blocks exist for other data type pairs (e.g., DWORD_TO_WORD). AL_TO_AW is specifically designed for the LWORD↔WORD combination.
- **Custom Composite Function Blocks** – This function block demonstrates how to efficiently implement recurring conversion tasks in the 4diac IDE through encapsulation.

Conclusion

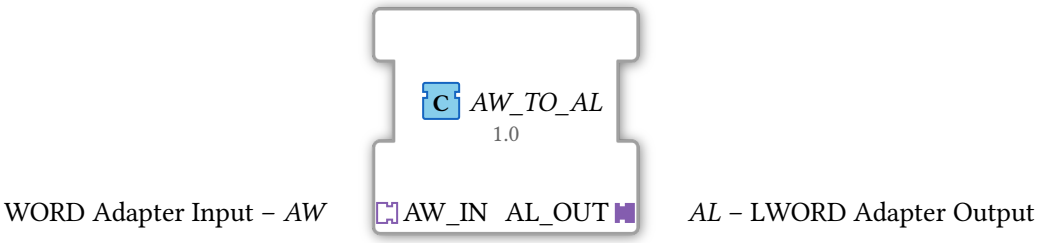
AL_TO_AW is a compact and clearly structured composite function block for converting an LWORD adapter to a WORD adapter. It utilizes the proven IEC conversion function, offers an adapter-compliant interface, and is ideally suited for integration into 4diac projects that require clean separation of data word widths.

🌐 Related topic subpages on ms-muc-docs.de

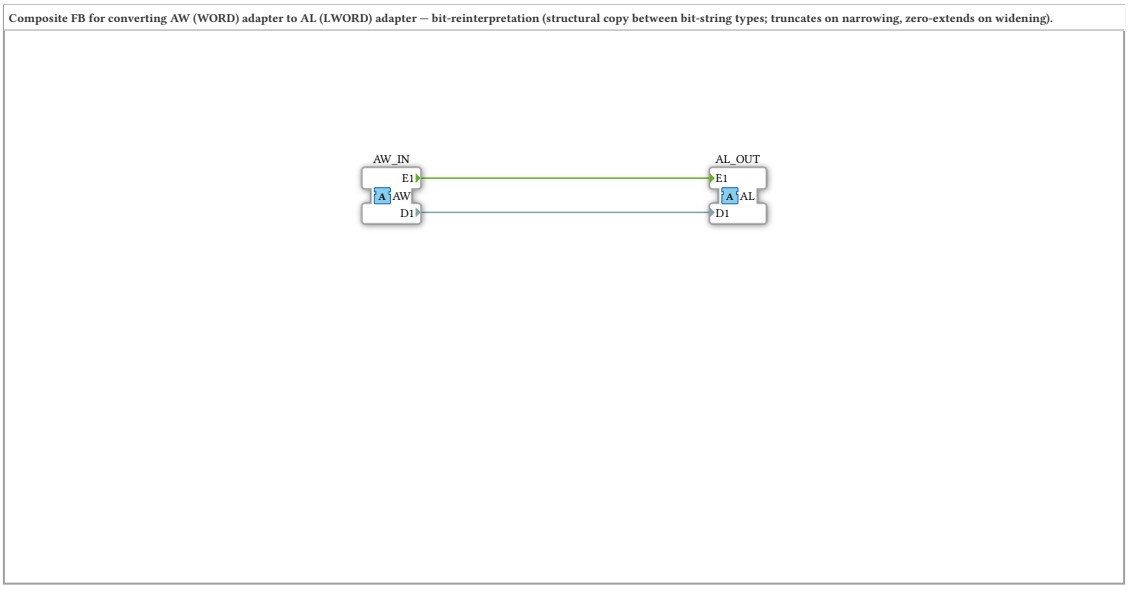
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AW_TO_AL



WORD Adapter Input – *AW* *AL* – LWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The AW_TO_AL function block is a composite block for converting a WORD adapter (AW) to an LWORD adapter (AL). It is used to convert data from a 16-bit format to a 64-bit format, typically in control and automation systems. The block forwards events and data directly from the input side to the output side, ensuring the data type conversion.

Interface Structure

Event Inputs

None (Events are transmitted exclusively via the AW_IN adapter)

Event Outputs

None (Events are output exclusively via the AL_OUT adapter)

Data Inputs

None (Data is received exclusively via the AW_IN adapter)

Data Outputs

None (Data is sent exclusively via the AL_OUT adapter)

Adapters

- **AW_IN** (Socket): Adapter type `adapter::types::unidirectional::AW` – WORD input (16-bit) with event and data channels (E1, D1).
- **AL_OUT** (Plug): Adapter type `adapter::types::unidirectional::AL` – LWORD output (64-bit) with event and data channels (E1, D1).

Functionality

The function block implements a simple passthrough function: The event (E1) received by the AW_IN adapter is directly forwarded to the event output AL_OUT.E1. Simultaneously, the 16-bit data signal from AW_IN.D1 is transferred to the 64-bit data output AL_OUT.D1. Since the AL adapter expects LWORD, the WORD value is implicitly converted to an LWORD (by zero expansion or explicit type conversion according to the runtime environment used). The function block does not perform any further operations.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- The function block is implemented as a composite function block, meaning it contains an internal network that connects the adapters.
- The type conversion from WORD to LWORD occurs automatically through the concatenation of the data lines (implicit type conversion).
- The event and data paths are synchronized: Each incoming event triggers an outgoing event, accompanied by the current data value.
- The function block is unidirectional and does not support reverse communication.

State Overview

Since the function block does not have its own state logic but only implements pass-through, there are no explicit states. Its function is purely combinatorial or event-driven without memory behavior.

Application Scenarios

- **Format Adaptation**: If a sensor or controller only delivers WORD data (16 bits), but the downstream system expects LWORD data (64 bits), this function block can be used for adaptation.
- **Adapter Bridge**: In communication chains where different adapter types are used, the function block enables seamless integration.
- **Protocol Conversion**: As part of a larger adapter chain for converting from 16-bit to 64-bit data buses.

Comparison with Similar Function Blocks

- **AW_TO_AW**: Identical type, no conversion.

- **AL_TO_AW**: Reverse conversion from LWORD to WORD (potentially resulting in data loss).
- **Direct connection without conversion**: If the systems use the same adapter type, no conversion module is required.
- The AW_TO_AL function block is characterized by its simplicity and specific type adaptation.

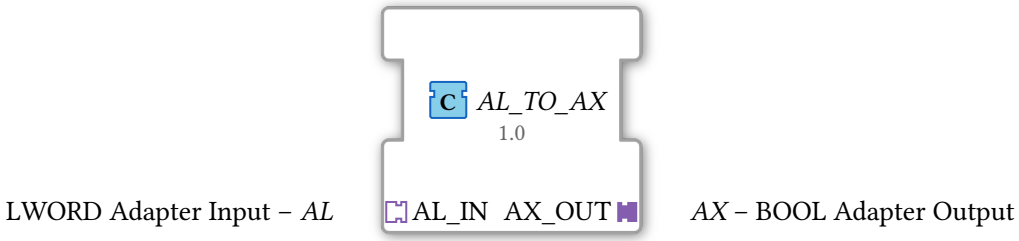
Conclusion

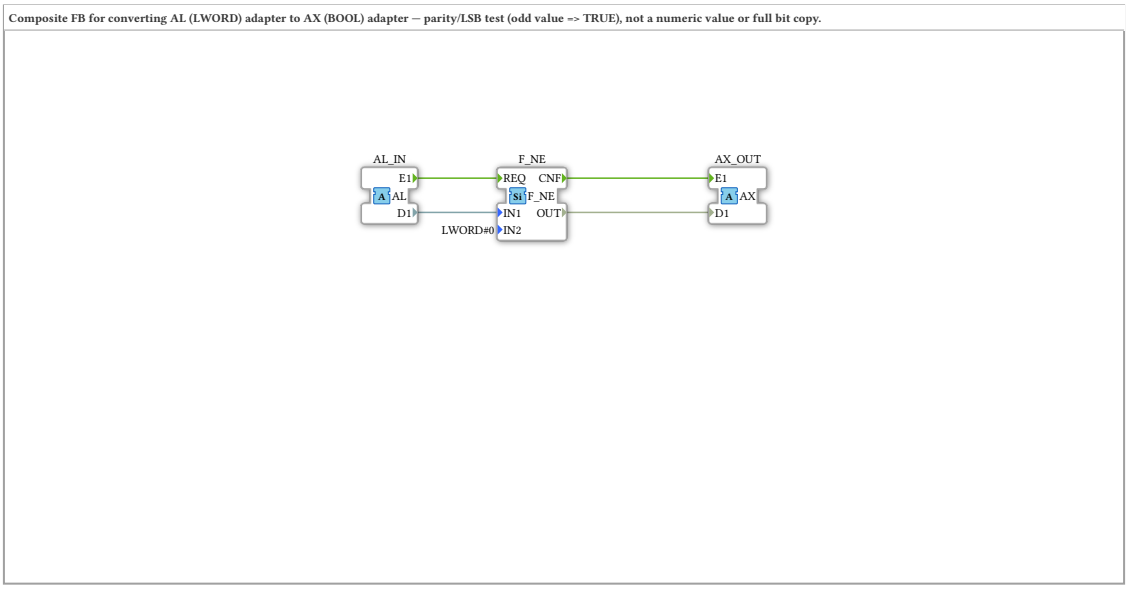
The AW_TO_AL function block offers a straightforward way to convert WORD data to an LWORD format. It is particularly suitable for scenarios where pure type adaptation without additional logic is required. Its implementation as a composite function block ensures transparent and efficient data transfer.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AL_TO_AX





• – ○ • – ○ • – ○ •

Introduction

The function block `AL_TO_AX` is a composite function block that performs a unidirectional adapter conversion from a `AL` adapter (data type `LWORD`) to a `AX` adapter (data type `BOOL`). It enables the simple conversion of an LWORD signal into a Boolean signal, which is then typically passed to subsequent BOOL-based adapters or function blocks. The function block checks whether the incoming LWORD value is non-zero and returns `TRUE` if so.

Interface Structure

The function block does not have its own event or data inputs/outputs, but uses only adapter interfaces:

- **Socket (Input):** `AL_IN` – Receives a `AL` adapter (contains an LWORD and an event).
- **Plug (Output):** `AX_OUT` – Provides a `AX` adapter (contains a BOOL and an event).

Internal connections are handled via the adapter elements:

Event Inputs

(via the AL_IN adapter)

- **`AL_IN.E1`** – Event input; triggers the conversion.

Event Outputs

(via the AX_OUT adapter)

- **`AX_OUT.E1`** – Event output; signals the completion of the conversion.

Data Inputs

(via the AL_IN adapter)

- **`AL_IN.D1`** (`LWORD`) – The LWORD value to be converted.

Data Outputs

(via the AX_OUT adapter)

- **`AX_OUT.D1`** (`BOOL`) – The result of the conversion: `TRUE` if `AL_IN.D1 ≠ 0`; otherwise `FALSE`.

Adapter

- **AL_IN** – Socket for the `adapter::types::unidirectional::AL` adapter (LWORD-based)
- **AX_OUT** – Plug for the `adapter::types::unidirectional::AX` adapter (BOOL-based)

Functionality

The function block internally uses the function block `F_NE` ("not equal") from the IEC 61131 library. This compares the incoming LWORD value (`AL_IN.D1`) with the constant value `LWORD#0`. If the values are not equal, `F_NE.OUT` returns the value `TRUE`; otherwise, it returns `FALSE`.

The event-driven sequence is as follows:

1. An event at `AL_IN.E1` triggers the input `F_NE.REQ`.
2. `F_NE` performs the comparison and places the result at `F_NE.OUT`.
3. Upon completion, `F_NE` sends an event to `F_NE.CNF`, which is connected to `AX_OUT.E1`.
4. Simultaneously, the result `F_NE.OUT` is transferred to `AX_OUT.D1`.

Thus, with each event at the input, an updated Boolean output is provided at the output.

Technical Features

- **Parity/LSB test** (odd value => TRUE), not a numeric value or full bit copy.
- **Type Conversion:** The function block converts an LWORD value not bitwise, but as an integer into a BOOL value. Any non-zero LWORD results in `TRUE`.
- **Adapter-Based:** By using the standardized unidirectional adapters (`AL` and `AX`), the function block can be used in any 4diac network that supports these adapter types.
- **Latency:** Processing occurs synchronously with the input event. The output is updated in the same execution cycle.
- **No State Storage:** The function block is purely combinatorial; it does not store any values between calls.

State Overview

The `AL_TO_AX` function block does not have its own state machine. Its behavior is purely data flow-driven by the incoming event. Therefore, a state overview is not available.

Application Scenarios

- **Threshold Detection:** Conversion of an LWORD sensor signal (e.g., counter reading, bitmask) into a Boolean signal that indicates whether a non-zero value is present.
- **Coupling of LWORD and BOOL Adapters:** In heterogeneous systems where one functional block provides an LWORD-based adapter, but the next block expects a BOOL adapter.
- **Error Detection:** An LWORD error register (e.g., 0 = no error, ≠0 = error) is translated into a simple BOOL signal "error present".

Comparison with Similar Components

- **Direct Converters (e.g., `LWORD_TO_BOOL`):** These usually only convert the least significant bit or define a fixed bit. The `AL_TO_AX` block, on the other hand, uses zero comparison, so any set bit or non-zero value is interpreted as `TRUE`.
- **Adapter Converters:** Generic adapter converters exist that handle various data types (e.g., `ANY_TO_BOOL`). The specialized `AL_TO_AX` block is tailored to the aforementioned adapter combination and requires no external parameterization.

Conclusion

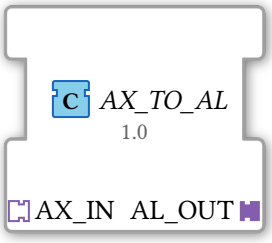
The `AL_TO_AX` function block offers a simple and reliable way to convert a unidirectional LWORD-based adapter into a BOOL-based adapter. The use of zero comparison ensures unambiguous interpretation. Integration into existing 4diac networks is straightforward thanks to the standardized adapter interfaces. This component is particularly well-suited for fast, event-driven conversions without requiring additional memory and is a practical addition to adapter conversion in industrial automation.

🌐 Related topic subpages on ms-muc-docs.de

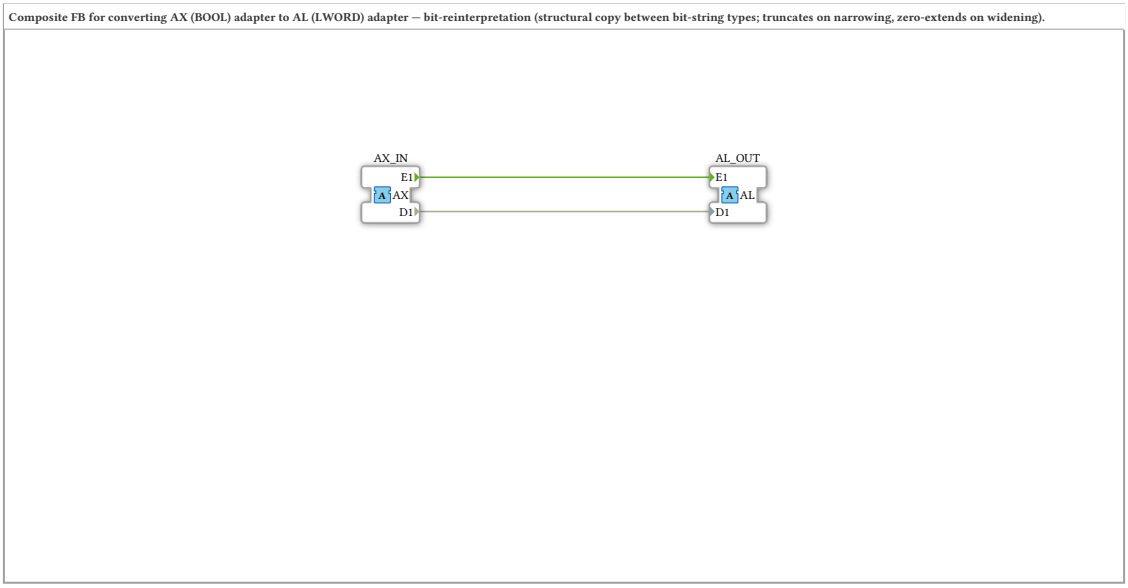
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AX_TO_AL



BOOL Adapter Input – *AX* — *AL* – LWORD Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AX_TO_AL** is a composite block for bidirectional coupling of an AX adapter (BOOL) with an AL adapter (LWORD). It enables the forwarding of events and data between these two adapter interfaces without requiring the implementation of custom logic. The function block serves as a simple conversion bridge and is typically used in automation systems to connect components with different adapter formats.

Interface Structure

Event Inputs

Name	Adapter	Description
AX_IN.E1	AX (BOOL)	Event input connected via the AX adapter.

Event Outputs

Name	Adapter	Description
AL_OUT.E1	AL (LWORD)	Event output that passes the incoming event on unchanged.

Data Inputs

Name	Adapter	Type	Description
AX_IN.D1	AX (BOOL)	BOOL	Data input for a Boolean signal.

Data Outputs

Name	Adapter	Type	Description
AL_OUT.D1	AL (LWORD)	LWORD	Data output that represents the Boolean signal as an LWORD (64-bit memory). Type conversion is performed outside the function block.

Adapter

Type	Name	Direction
Socket	AX_IN	Input (BOOL)
Plug	AL_OUT	Output (LWORD)

Functionality

The FB **AX_TO_AL** functions purely as a pass-through function block. Within the FB network, the event connection `AX_IN.E1 → AL_OUT.E1` and the data connection `AX_IN.D1 → AL_OUT.D1` are established directly. An incoming event at the AX socket immediately triggers the corresponding event at the AL plug. Simultaneously, the Boolean value of `AX_IN.D1` is passed to `AL_OUT.D1`.

The function block itself does not perform any value conversion; the conversion of the Boolean signal to an LWORD (e.g., 1 or 0 in a 64-bit word) is provided by the underlying adapter implementation or the host environment.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Composite Architecture**: The function block contains no internal ECC (Execution Control Chart) or its own logic. It consists solely of two adapter connections and simple passthroughs.
- **Type Mapping**: The mapping from BOOL to LWORD is implicit. The user must ensure that the connected adapters correctly support the conversion.
- **Project Structure**: The function block is located in the package `adapter::conversion::unidirectional`, which indicates its function as a unidirectional conversion adapter.

State Overview

The function block has no state machine and no states of its own. It operates in an event-driven manner and reacts passively to signals from the AX adapter.

Application Scenarios

- Integrating an old Boolean interface (AX) into a system that only supports LWORD adapters (AL).
- Temporary bridging circuit during the migration of fieldbus protocols.
- Test environments where a Boolean signal needs to be interpreted as a 64-bit value.

Comparison with Similar Function Blocks

Other conversion function blocks exist, such as... Examples include `AL_TO_AX` (reverse conversion from LWORD to BOOL) or `BOOL_TO_LWORD` (direct conversion without adapter encapsulation). Unlike these pure value converters, **AX_TO_AL** encapsulates the entire adapter interface, so the user only needs to connect the adapters.

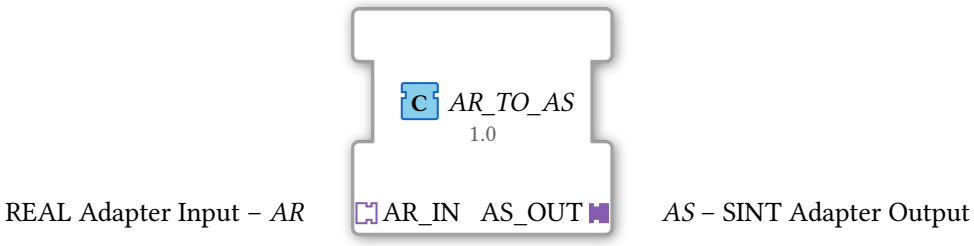
Conclusion

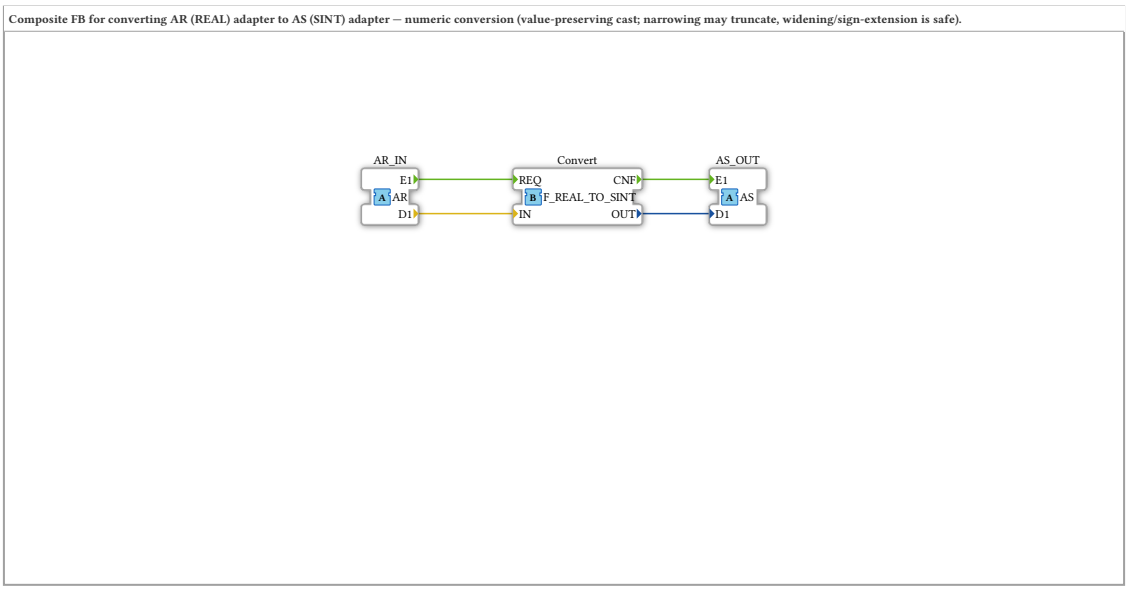
The **AX_TO_AL** function block is a lean yet useful adapter coupler. It reduces the effort required when different adapter formats need to be combined in an automation system and does not require any custom logic or state definitions. Its simple structure facilitates quick integration into existing IEC 61499 networks.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AR_TO_AS





• – ○ • – ○ • – ○ •

The function block **AR_TO_AS** is a composite block that converts a **REAL adapter** (AR) into a **SINT adapter** (AS). It enables simple and standardized conversion between the adapter types and encapsulates the necessary logic in a reusable block. The function block does not have separate event or data ports at the IEC 61499 level. All communication takes place via the integrated adapters. The following table describes the available adapters.

No dedicated event inputs are available. Events are received via the **AS_OUT adapter** (see Adapters).
No dedicated event outputs are available. Events are sent via the **AR_IN adapter** (see Adapters).
No dedicated data inputs. Data is received via the **AS_OUT adapter** (see Adapters).

No dedicated data outputs. Data is sent via the **AR_IN adapter** (see Adapters).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Adapter	Type	Direction	Description
AR_IN	adapter::types::unidirectional::AR	Socket (inbound)	Unidirectional REAL adapter. Returns a REAL value (D1) and an associated event (E1).
AS_OUT	adapter::types::unidirectional::AS	Plug (outbound)	Unidirectional SINT adapter. Receives a SINT value (D1) and an associated event (E1).

Functionality

The function block performs the conversion as follows:

1. An incoming event on **AR_IN.E1** triggers the internal conversion block **F_REAL_TO_SINT**.
2. The REAL value associated with the event from **AR_IN.D1** is forwarded to the input `IN` of the conversion block.
3. **F_REAL_TO_SINT** converts the REAL value into a SINT value and outputs the result to its output `OUT`. 4. After the conversion is complete, an event is sent via **AS_OUT.E1**, along with the converted SINT value to **AS_OUT.D1**.

A synchronous 1:1 conversion takes place – each incoming REAL request generates exactly one outgoing SINT response.

- **Reusability**: The function block is based on the standard FB `F_REAL_TO_SINT` from the IEC 61131 library. This ensures that the conversion logic is standards-compliant and available on many platforms.
- **Adapter Interface**: The use of adapters allows for loose coupling between function blocks and facilitates the exchange or extension of the interface.
- **Value Range**: When converting from REAL (approx. $\pm 3.4 \times 10^{38}$) to SINT (−128 ... 127), value range exceedances can occur. The internal function block `F_REAL_TO_SINT` behaves according to the IEC 61131 definition (overflow or use of saturation behavior). The user should limit the input values accordingly.

The **AR_TO_AS** does not have its own state machine. Since it is a pure composite function block that only establishes the connection between the incoming and outgoing adapters, there is no internal state behavior. The function block operates combinatorially at the event level.

- **Integration of REAL-based function blocks in SINT environments**: If a component provides a REAL adapter, but the subsequent function block expects a SINT adapter, **AR_TO_AS** can be used as an intermediary adapter.
- **Type-safe conversion in modular controllers**: By using adapters, the typing is preserved, and the conversion is explicitly visible in the network.
- **Library extension**: The function block can be used as part of an adapter conversion library to standardize data exchange between different numerical resolutions.

Analogous conversion adapters exist, e.g.,... B.:

- **AR_TO_INT** – REAL ⟶ INT
- **AR_TO_UDINT** – REAL ⟶ UDINT
- **AR_TO_LREAL** – REAL ⟶ LREAL

All of these modules share the use of the corresponding IEC 61131 conversion block. The difference lies in the target type and the adapter definition. **AR_TO_AS** is specifically designed for conversion to the narrow, signed 8-bit SINT type.

The **AR_TO_AS** is a simple yet effective composite module for converting a REAL adapter to a SINT adapter. It cleanly encapsulates the conversion logic and enables type-safe, event-driven communication between function blocks with different numeric interfaces. Thanks to the use of standard libraries, it is reliable and platform-independent.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).

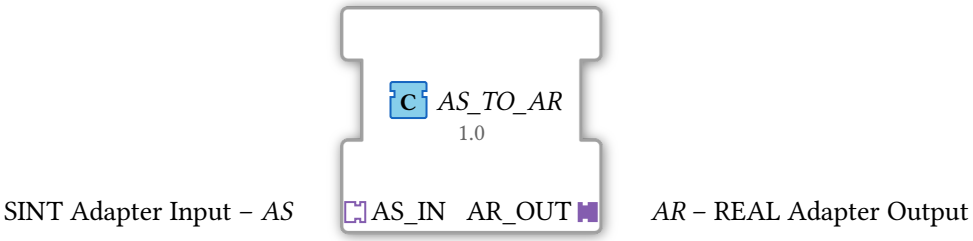
State Overview

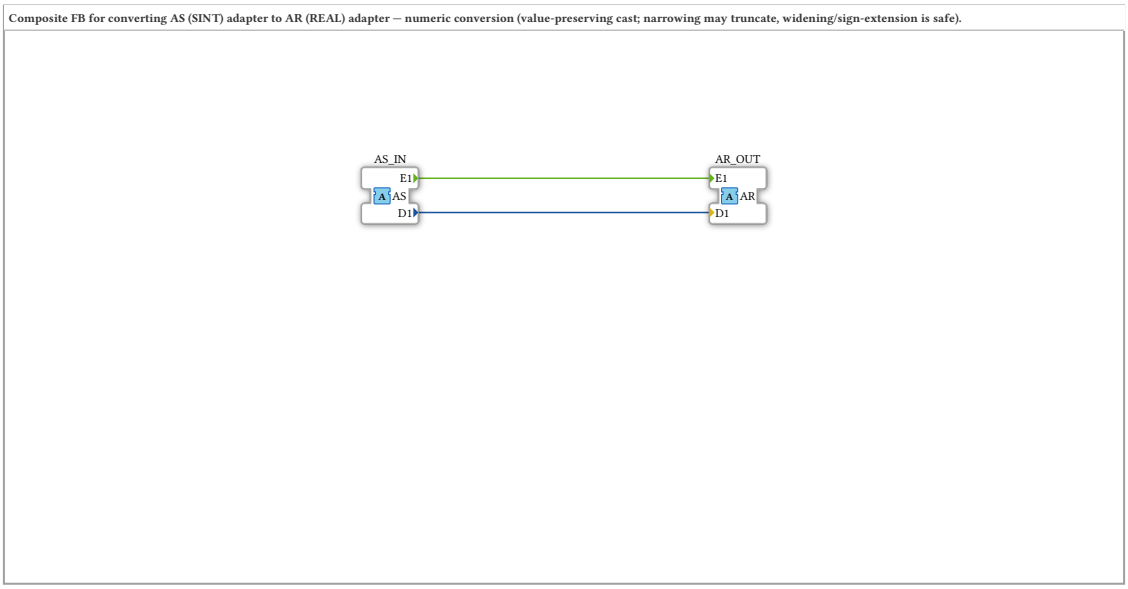
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AS_TO_AR





• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AS_TO_AR` is a composite function block that converts a unidirectional adapter of type **AS** (SINT) to an adapter of type **AR** (REAL). It is used to establish a logical connection between components based on different data types – specifically, from a signed 8-bit integer (SINT) to a floating-point number (REAL).

Interface Structure

Event Inputs

The function block does not have its own dedicated event inputs. Events are received exclusively via the integrated adapter **AS_IN**.

- **AS_IN.E1** – Event from the connected SINT adapter.

Event Outputs

Events are passed on via the integrated adapter **AR_OUT**.

- **AR_OUT.E1** – Event for the connected REAL adapter.

Data Inputs

Data is provided exclusively via the adapter **AS_IN**.

- **AS_IN.D1** – Data value of type **SINT** (signed 8-bit integer).

Data Outputs

Data is output exclusively via the adapter **AR_OUT**.

- **AR_OUT.D1** – Data value of type **REAL** (floating-point number).

Adapters

Type	Name	Direction	Comment
`adapter::types::unidirectional::AS`	**AS_IN**	Socket	Input adapter for SINT data
`adapter::types::unidirectional::AR`	**AR_OUT**	Plug	Output adapter for REAL data

Functionality

The `AS_TO_AR` block behaves as a pure type conversion passthrough. Within the integrated function network, the event `AS_IN.E1` is directly connected to `AR_OUT.E1`, and the data value `AS_IN.D1` is directly connected to `AR_OUT.D1`. The actual conversion from **SINT** to **REAL** is performed implicitly through the adapter definitions of the respective types. The block itself does not execute any additional logic or calculations.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block**: Conversion is achieved solely through the interconnection of adapters in the function network. It does not have its own state machine (ECC).
- **Package Membership**: The block is included in the package `adapter::conversion::unidirectional`.
- **No Data Storage**: All values are passed on instantly; the block has no internal memory.
- **Implicit Type Conversion**: The conversion from integer (SINT) to floating-point number (REAL) takes place in the periphery of the adapter types.

State Overview

The `AS_TO_AR` block does not have a state machine (ECC). Its behavior is purely combinatorial and is limited to the direct forwarding of events and data.

Application Scenarios

- Connecting a sensor that provides its measured values as a SINT adapter to a processing component that expects REAL adapters.
- Transitioning between different '4diac' adapter types in heterogeneous control projects.
- Preparing values for mathematical operators or visualization components that require REAL inputs.

Comparison with Similar Function Blocks

- **AR_TO_AS** – Performs the reverse conversion from REAL to SINT.

- **General type converters** (e.g., `INT_TO_REAL`) operate at the level of individual data ports, while `AS_TO_AR` converts entire adapter interfaces.
- This function block does not offer adjustable parameters (e.g., rounding modes); more specific conversion would have to be implemented using separate function blocks.

Conclusion

AS_TO_AR` is a simple yet useful composite block for adapter type conversion from SINT to REAL. It enables the lossless integration of components with different data types in continuous control sequences and is characterized by its low complexity and direct passthrough.

🌐 Related topic subpages on ms-muc-docs.de

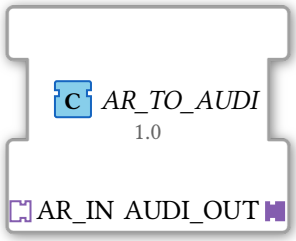
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numeric Index (unidirectional)

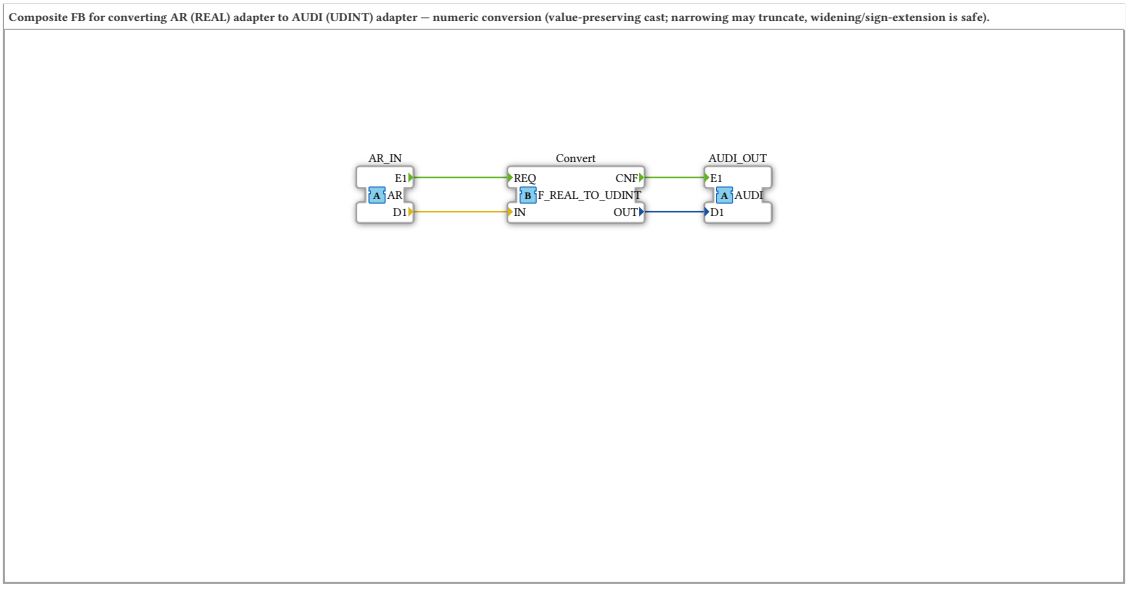
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AR_TO_AUDI

REAL Adapter Input – *AR*



AUDI – UDINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `AR_TO_AUDI` is a composite function block (FB) that receives a REAL value from a unidirectional **AR** adapter, converts it to a **UDINT** value, and outputs it via an **AUDI** adapter. It encapsulates the type conversion `REAL_TO_UDINT` and simplifies integration into adapter-based communication structures.

Interface Structure

The function block has no dedicated event or data inputs/outputs. All communication takes place via two adapters (socket and plug).

Adapter (Socket – Input)

Name	Type	Comment
AR_IN	adapter::types::unidirectional::AR	Unidirectional REAL adapter as input

The adapter AR_IN provides an event E1 and a data input D1 of type REAL.

Adapter (Plug – Output)

Name	Type	Comment
AUDI_OUT	adapter::types::unidirectional::AUDI	Unidirectional UDINT adapter as output

The adapter AUDI_OUT expects an event E1 and a data output D1 of type UDINT.

Functionality

The function block internally contains an instance of the conversion function block F_REAL_TO_UDINT from the library iec61131::conversion. The signal connections implement a simple data flow:

1. **Event from the input adapter**

The event AR_IN.E1 triggers the conversion function block via its event input REQ.

1. **Data Conversion**

Simultaneously, the REAL value of AR_IN.D1 is transferred to the data input IN of F_REAL_TO_UDINT. The conversion occurs without delay within the event cycle.

1. **Output Event and Result**

After successful conversion, F_REAL_TO_UDINT sends an acknowledgment event (CNF) to the output adapter (AUDI_OUT.E1). The converted UDINT value is forwarded to AUDI_OUT.D1 via the data output OUT.

The conversion is synchronous: Each incoming event triggers exactly one output.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block** – The logic is implemented entirely through an internal network; there is no standalone algorithm or state machine.
- **Unidirectional Adapters** – Both the input and output interfaces are unidirectional and transmit only one event/data channel.
- **Library Used** – Conversion is performed using the IEC 61131 block F_REAL_TO_UDINT, which is hardware-independent and widely used.
- **No Error Handling** – The block does not perform range or type checking; the REAL to UDINT conversion follows standard rules (rounding, delimiting).

State Overview

Since the block does not have its own state machine, its operation is determined solely by the event network:

- **Idle** – No event is active at AR_IN.E1; the output adapter remains inactive.
- **Active** – An incoming event immediately triggers the conversion and produces an output event. Upon completion, the function block returns to its idle state (no internal memory).

Application Scenarios

- **Bridging** between system components that provide REAL values (e.g., floating-point sensors) and components that process UDINT values (e.g., counters, index management).
- **Adapter-based communication** in distributed automation systems according to IEC 61499, when the interfaces are defined as unidirectional adapters.
- **Type conversion** in data preprocessing paths before values are passed to programmable logic controllers (PLCs) or visualizations.

Comparison with Similar Function Blocks

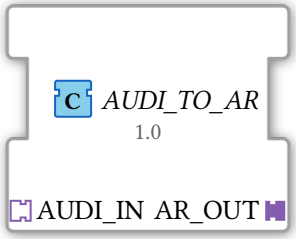
- **REAL_TO_DINT** – converts REAL to a signed 32-bit integer; here, the conversion is to an unsigned integer (UDINT).
- **AR_TO_xx blocks** – Other variants could convert to, for example, `AR_TO_BYTE` or `AR_TO_DWORD`, but without the adapter frame.
- **Direct converter** – The internal block `F_REAL_TO_UDINT` can also be integrated directly without an adapter; `AR_TO_AUDI` offers an encapsulated, easily replaceable interface.

Conclusion

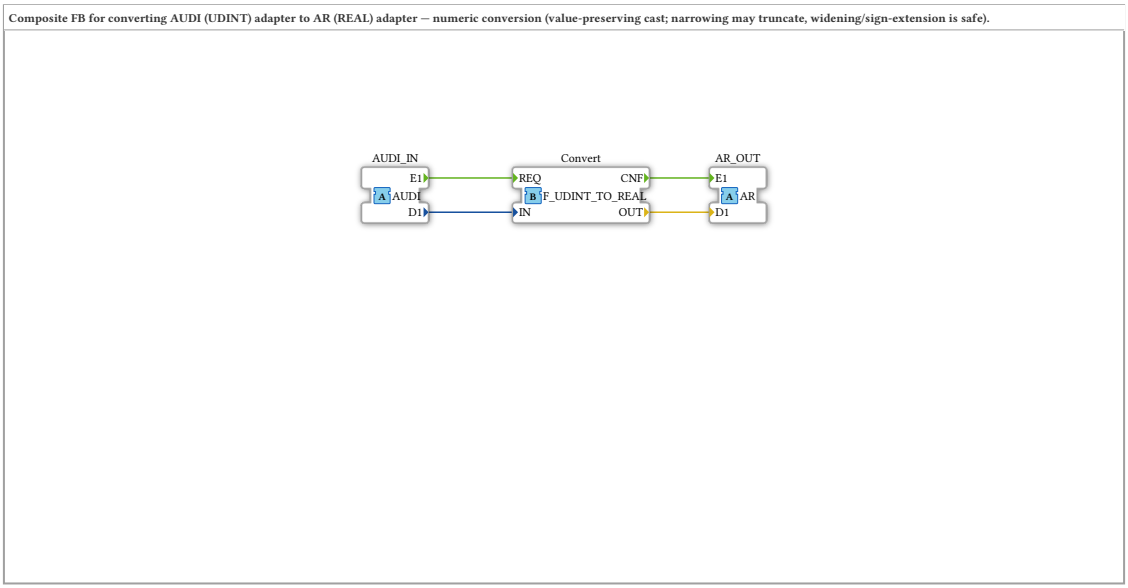
The block `AR_TO_AUDI` provides a cleanly encapsulated, adapter-based type conversion from REAL to UDINT. It reduces wiring effort in modular IEC 61499 projects and ensures reliable, event-driven implementation. The use of standardized conversion blocks makes the solution portable and easy to understand.

AUDI_TO_AR

UDINT Adapter Input – *AUDI*



AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `AUDI_TO_AR` is a composite function block (FB) that converts a UDINT-sized integer received via a unidirectional AUDI adapter into a REAL value and outputs it via a unidirectional AR adapter. It encapsulates the conversion logic, thus enabling clean interface adaptation between different adapter types.

Interface Structure

This FB has no direct event or data inputs/outputs. All communication takes place exclusively via adapter interfaces.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Name	Type	Direction	Description
AUDI_IN	adapter::types::unidirectional::AUDI	Socket (Input)	Returns a UDINT value and an associated event. The adapter provides an internal event output E1 and a data output D1 (UDINT data type).
AR_OUT	adapter::types::unidirectional::AR	Plug (Output)	Outputs the converted REAL value. The adapter has an internal event input E1 and a data input D1 (data type REAL).

Functionality

1. An external signal triggers the event AUDI_IN.E1.
2. This event is forwarded to the internal conversion block F_UDINT_TO_REAL (type iec61131::conversion::F_UDINT_TO_REAL) (Convert.REQ).
3. Simultaneously, the UDINT value of AUDI_IN.D1 is passed to the data input Convert.IN.
4. The conversion block performs the conversion from UDINT to REAL.
5. Upon completion, it triggers the event Convert.CNF, which is passed on to the event input AR_OUT.E1.
6. The converted REAL value is written from Convert.OUT to the data input AR_OUT.D1 and is thus available at the AR adapter.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- The conversion uses an IEC 61131-3 standard function block (F_UDINT_TO_REAL), which is instantiated within the Composite FB.
- The adapters are implemented as unidirectional interfaces, which enforces a clear data flow direction (from the AUDI to the AR adapter).
- The Composite FB does not generate any state logic of its own, but delegates all processing to the internal conversion function block.

State Overview

The function block does not have its own state machine. Its functionality follows the simple sequence:

- **Waiting for input event** – as long as AUDI_IN.E1 is not active, the function block remains passive.
- **Conversion** – after AUDI_IN.E1 arrives, the integrated F_UDINT_TO_REAL is executed.
- **Output** – after successful conversion, AR_OUT.E1 is triggered and the real value is provided.

Application Scenarios

- Connecting a UDINT-based counter (e.g., pulse counter) to a controller that expects real values for calculations or visualizations.
- Converting adapter signals between different data types in a modular, adapter-based communication architecture.
- As a building block in a library for type-safe adapter conversions.

Comparison with similar building blocks

This function block is specifically designed for the combination of AUDI (UDINT) and AR (REAL) adapters. Comparable building blocks exist for other data types, e.g.:

- `SINT_TO_AR` (SINT ⟶ REAL)
- `AUDI_TO_AI` (UDINT ⟶ INT)

The advantage of this building block lies in its clear encapsulation and reusability within an adapter-based system design.

Conclusion

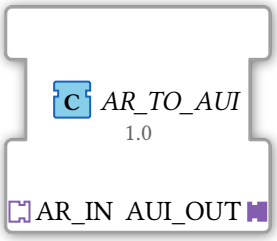
The `AUDI_TO_AR` function block offers a simple yet effective way to convert UDINT data to REAL data via adapter interfaces. By using an IEC standard component and purely adapter-based communication, it is easy to understand, maintain and can be used in many automation projects.

Numeric Index (unidirectional)

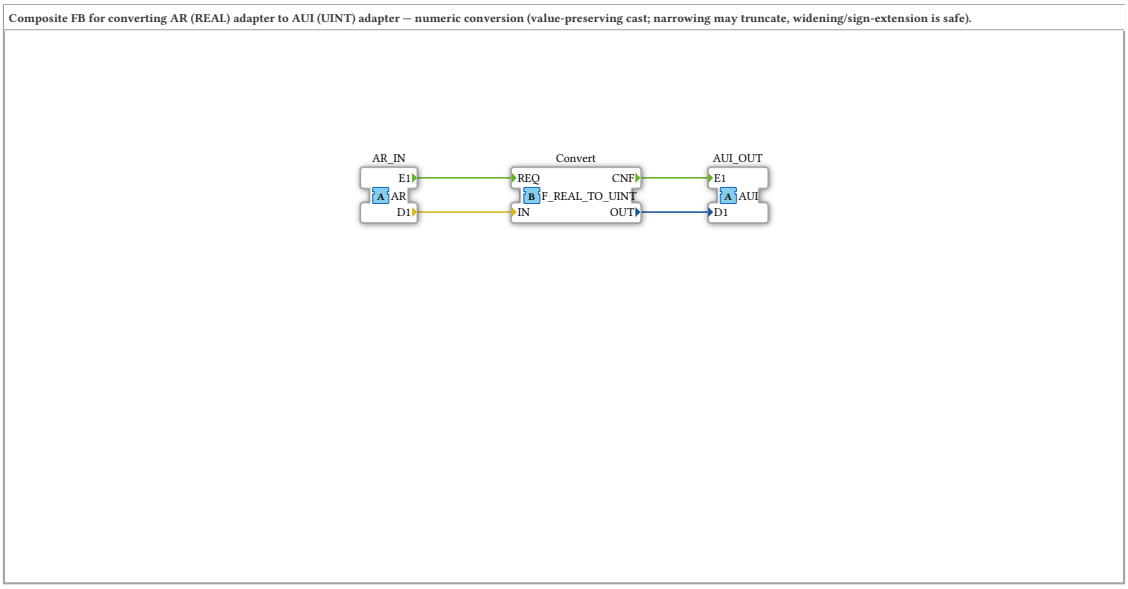
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AR_TO_AUI

REAL Adapter Input – *AR*



AUI – UINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AR_TO_AUI** is a composite function block (FB) that implements a uniform and bidirectional conversion from an AR adapter (REAL) to an AUI adapter (UINT). It encapsulates the type conversion and simplifies communication between components that use different data types via adapters.

Interface Structure

Event Inputs

- **AR_IN.E1** – Conversion trigger: This event is received by the connected AR adapter and starts the conversion of the REAL value.

Event Outputs

- **AUI_OUT.E1** – Conversion completion: After successful conversion, this event is sent to the AUI adapter.

Data Inputs

- **AR_IN.D1** – REAL input value: The floating-point value (REAL) to be converted, provided via the AR adapter.

Data Outputs

- **AUI_OUT.D1** – UINT output value: The result of the conversion (UINT) is passed to the AUI adapter via this data output.

Adapters

- **AR_IN** – Socket of type `AR` (unidirectional REAL adapter)
- **AUI_OUT** – Plug of type `AUI` (unidirectional UINT adapter)

The adapters each combine an event signal and a data signal, so the function block offers a clean, type-safe interface.

Functionality

1. An incoming event on **AR_IN.E1** triggers the processing.
2. The current value of **AR_IN.D1** (REAL) is passed to the integrated conversion block **F_REAL_TO_UINT**.
3. The conversion transforms the REAL value into an unsigned integer value (UINT) – according to IEC 61131-3 rounding rules.
4. Upon completion, the result is output to **AUI_OUT.D1**, and the event **AUI_OUT.E1** is sent simultaneously.

The entire process occurs without intermediate storage or delay – each pulse triggers an immediate conversion.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Stateless Design**: The block has no internal state memory; each conversion is independent of previous ones.
- **Uniform Adapter Interface**: By using unidirectional standard adapters (AR and AUI), the function block can be easily integrated into various architectures.
- **No Side Effects**: No global variables or system resources are affected.

State Overview

The AR_TO_AUI is a pure function block without an internal state machine (ECC). Its execution is strictly event-driven and deterministic. Therefore, a separate state machine is not required.

Application Scenarios

- **Connecting Analog Sensors**: When a sensor delivers its measured value as a REAL via an AR adapter, but the downstream control system expects UINT.
- **Data Bridge**: Mediation between different communication modules or libraries with different adapter types.
- **Protocol Conversion**: For example, in agricultural technology, where REAL values (e.g., speed) often need to be converted into integers (e.g., for counters).

Comparison with Similar Function Blocks

- **AR_TO_ADI**: Converts REAL to INT (signed). AR_TO_AUI, on the other hand, generates unsigned values (UINT), which is advantageous for certain applications such as counter readings or HMI indices.
- **Direct Conversion with F_REAL_TO_UINT**: This single function block offers the same logic but requires manually connecting events and data. The Composite FB encapsulates this effort and reduces the potential for errors in complex networks.

Conclusion

The **AR_TO_AUI** is a practical and reliable function block for easy adapter conversion from REAL to UINT. Its clear interface, purely functional behavior, and reusability make it a useful tool in the 4diac ecosystem. It is particularly suitable for applications requiring a clean separation of adapter types and easy integration into existing projects.

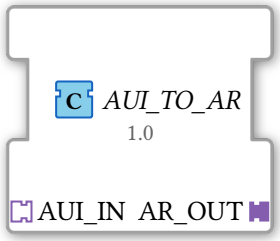
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

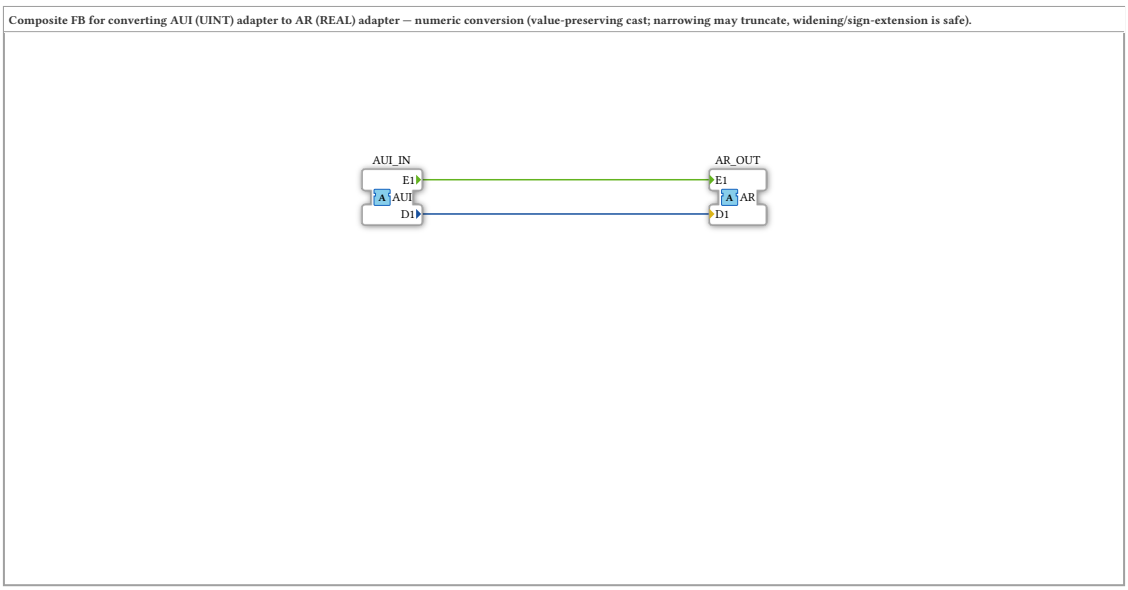
]

AUI_TO_AR

UINT Adapter Input – *AUI*



AR – REAL Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_TO_AR** function block is a composite component that converts an AUI adapter (UINT) into an AR adapter (REAL). It enables the seamless integration of an unsigned integer signal into systems that expect a REAL value by passing the event and data value through one-to-one.

Interface Structure

Event Inputs

The component does not have a direct event input, as events are provided exclusively via the **AUI_IN** input adapter:

- **E1** (from the AUI_IN adapter): Event that triggers data transmission.

Event Outputs

Event outputs are also implemented via the **AR_OUT** output adapter:

- **E1** (from the AR_OUT adapter): Event output after successful conversion.

Data Inputs

Data inputs are only available indirectly via the **AUI_IN** adapter:

- **D1** (from the AUI_IN adapter): Value to be converted as **UINT**.

Data Outputs

Data outputs are provided via the **AR_OUT** adapter:

- **D1** (from the AR_OUT adapter): Converted value as **REAL**.

Adapters

Name	Type	Direction	Description
AUI_IN	`adapter::types::unidirectional::AUI`	Input (Socket)	Provides a UINT signal and an associated event.
AR_OUT	`adapter::types::unidirectional::AR`	Output (Plug)	Outputs the converted REAL signal and an event.

Functionality

The **AUI_TO_AR** function block acts as a simple pass-through component. It connects 1:1:

- **AUI_IN.E1 → AR_OUT.E1**
- **AUI_IN.D1 → AR_OUT.D1**

This means: As soon as an event arrives at the AUI_IN input, the corresponding **UINT** data is passed to the **AR_OUT** output without any additional processing or type conversion. The actual conversion from **UINT** to **REAL** must be performed either by the sending adapter or by a downstream function block – the **AUI_TO_AR** itself does not perform any arithmetic conversion. It merely ensures that the event and data paths between the two different adapter types are compatible.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Function Block:** The implementation is purely via the network of adapter connections, without any internal logic or state machines.
- **No Type Conversion:** The **AUI_TO_AR** does not perform any explicit data type conversion. It assumes that the **AUI_IN** adapter already provides REAL-compliant data or that the type conversion is performed externally.
- **Unidirectional Adapter Type:** Both adapters are unidirectional, meaning they only transmit data and events in one direction (from AUI_IN to AR_OUT).

State Overview

Since the function block is implemented as a pure interconnection network without its own state machine, no state machine exists. Its functionality is limited to the immediate forwarding of incoming events and data.

Application Scenarios

- **System Integration:** Integrating sensors or controllers that provide their measured values as UINT via an AUI adapter into a system that expects REAL values via AR adapters.
- **Protocol Adaptation:** If two subsystems use different adapter types for the same physical quantity, **AUI_TO_AR** can serve as an adapter translator (provided the UINT value already numerically represents the desired REAL value).
- **Library Extension:** As a reusable function block in a library project to streamline adapter conversions.

Comparison with Similar Function Blocks

Function Block	Function	Difference

AUI_TO_AR	Adapter switching UINT ⟶ REAL	Pass-through only; no type conversion, expects compatible data.
AR_TO_AUI	Adapter switching REAL ⟶ UINT	Reverse direction, also pass-through only.
CONV_UINT_TO_REAL	Type conversion from UINT to REAL	Performs actual conversion, but requires direct data inputs/outputs instead of an adapter.

Conclusion

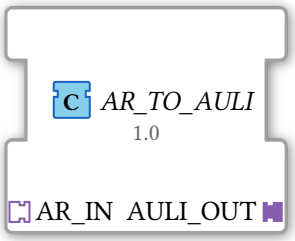
The **AUI_TO_AR** is a simple yet useful composite component that connects the **AUI** and **AR** adapter types without modifying the data. It is particularly suitable for systems where type compatibility is already provided at the adapter level or where subsequent conversion takes place. Its streamlined structure makes it a reliable tool for integrating different adapter protocols.

Numeric Index (unidirectional)

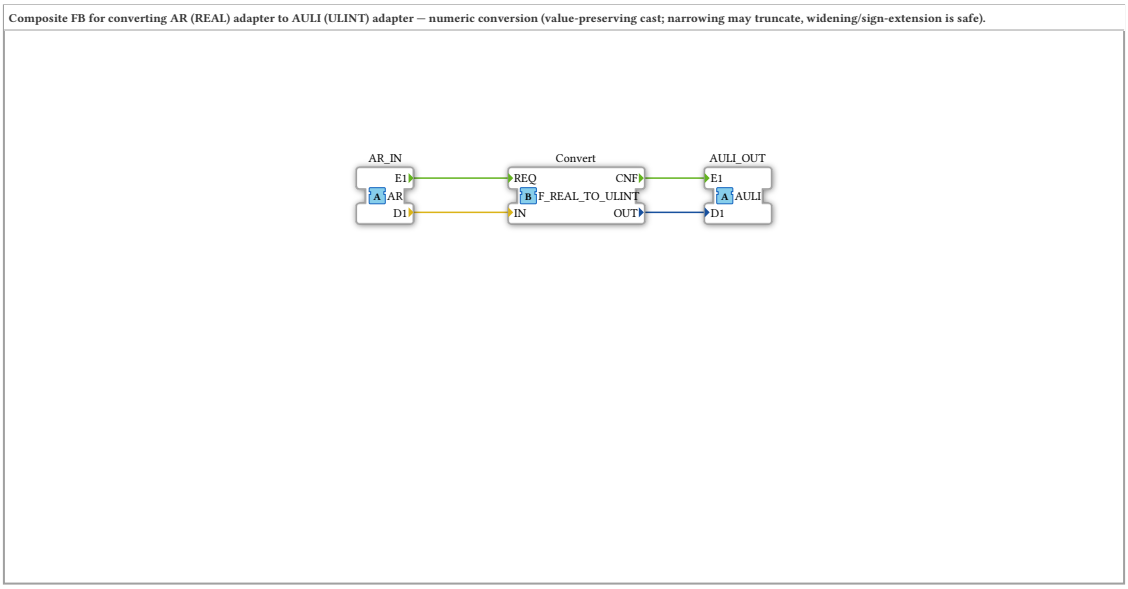
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AR_TO_AULI

REAL Adapter Input – *AR*



AULI – ULINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AR_TO_AULI** is a composite function block that converts a REAL input adapter (AR) into a ULINT output adapter (AULI). It serves as a unidirectional interface between components that work with the REAL data type and those that expect ULINT (unsigned long integer). The function block encapsulates the necessary conversion and automatically ensures event and data forwarding.

Interface Structure

Event Inputs
- **AR_IN.E1** – Input event from the AR adapter; triggers the conversion.

Event Outputs
- **AULI_OUT.E1** – Output event to the AULI adapter; signals that the converted value is available.

Data Inputs
- **AR_IN.D1** – REAL value from the AR adapter to be converted.

Data Outputs
- **AULI_OUT.D1** – ULINT value converted from the REAL input.

Adapters
- **AR_IN** (Socket) – Unidirectional AR adapter (Type: adapter::types::unidirectional::AR) as input.
- **AULI_OUT** (Plug) – Unidirectional AULI adapter (Type: adapter::types::unidirectional::AULI) as output.

Functionality

The function block is implemented as a composite FB and internally contains a single function block **F_REAL_TO_ULINT** from the IEC 61131 library. The operation is as follows:

1. An event at socket **AR_IN.E1** triggers the execution of the converter.
2. The data value **AR_IN.D1** is forwarded to input **Convert.IN**.
3. The converter performs the type conversion from REAL to ULINT and provides the result at output **Convert.OUT**.
4. The acknowledgment event **Convert.CNF** generates an event at plug **AULI_OUT.E1**, which informs the connected function block about the new value.
5. The converted ULINT value is output in parallel to **AULI_OUT.D1**.

The event and data connections are wired directly without additional logic, so the function block is a pure transformation module.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Unidirectionality:** Both the input and output adapters are unidirectional. There is no feedback or acknowledgment from the output to the input.
- **Converter Used:** The embedded function block **F_REAL_TO_ULINT** implements the standardized IEC 61131 conversion. Rounding and overflow behavior comply with IEC standards.
- **Event-Driven Execution:** The conversion is only performed when an event occurs at the input; without an event, the output remains unchanged.
- **Adapter Types:** The adapters are based on a unidirectional variant, meaning they transmit exactly one event and one data value per direction.

State Overview

As a composite function block, **AR_TO_AULI** has no states of its own. The internal state logic is determined solely by the embedded **F_REAL_TO_ULINT**, which typically implements a simple, event-driven sequence without state storage (REQ → Processing → CNF). The function block is therefore stateless and waits for a new input event after each conversion.

Application Scenarios

- **Interface Adaptation:** When a component (e.g., a control algorithm) delivers REAL values, but a downstream module (e.g., a counter or index logic) expects ULINT inputs.
- **Adapter Bridge:** Use in a toolchain that connects different data types via adapters without requiring the developer to perform the conversion manually.

- **Data Integration:** Integrating REAL-based sensors or parameters into a system that internally uses ULINT (e.g., for status codes or cumulative values).

Comparison with Similar Function Blocks

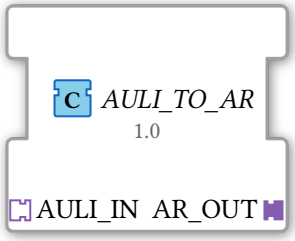
- **F_REAL_TO_ULINT** – Basic function block without adapters; requires direct data and event connections. **AR_TO_AULI** is an encapsulated variant with adapters that simplifies integration into adapter-based architectures.
- AR_TO_ *blocks for other target types** – Corresponding composite blocks exist for conversions from REAL to LINT, DINT, etc. They follow the same pattern and each uses the appropriate IEC converter.
- **Direct conversion in the network** – Alternatively, the F_REAL_TO_ULINT block could be directly integrated into a function block network, but this requires more manual wiring. **AR_TO_AULI** reduces the effort and improves reusability.

Conclusion

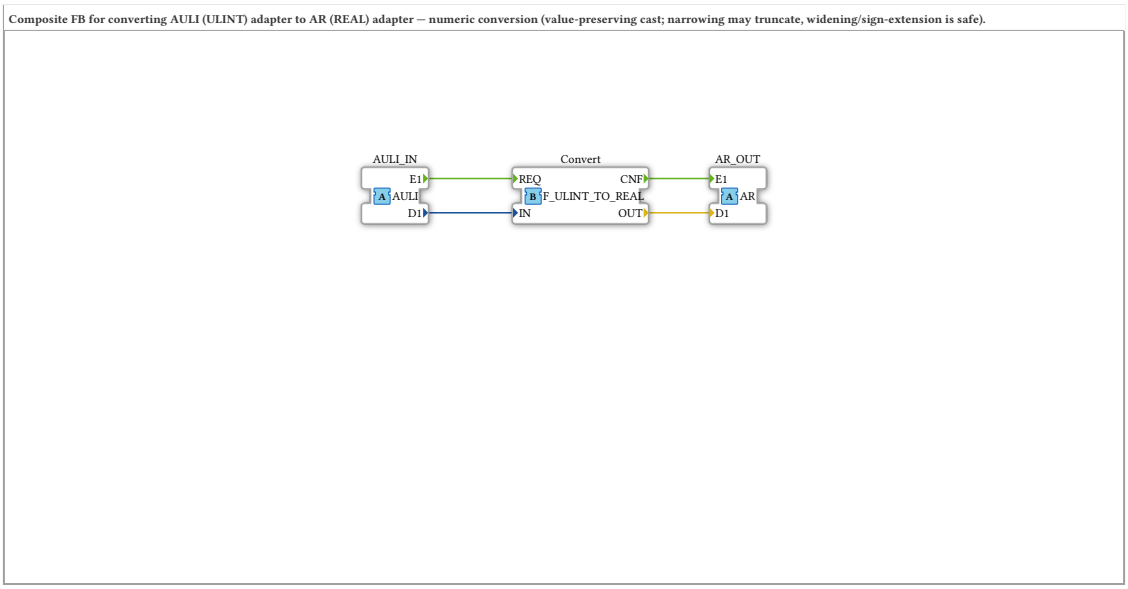
The **AR_TO_AULI** block is a simple yet useful encapsulation of REAL-to-ULINT conversion in an adapter-based format. It extends IEC 61131 conversion with a plug-and-play interface for unidirectional adapters and enables a clean separation of conversion logic and application communication. This component is particularly suitable for modular systems that are based on adapter concepts.

AULI_TO_AR

ULINT Adapter Input – *AULI*



AR – REAL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_TO_AR** serves as a converting composite block that transforms an **AULI** adapter (ULINT) into an **AR** adapter (REAL). It enables the easy integration of components that work with different data types and provides type conversion via standardized adapter interfaces.

Interface Structure

The function block has no direct event or data inputs/outputs, but communicates exclusively via two adapter interfaces.

Event Inputs

- **AULI_IN.E1** (event from socket AULI_IN) – triggers the conversion of the incoming ULINT value.

Event Outputs

- **AR_OUT.E1** (Event for plug AR_OUT) – signals that the converted REAL value is ready.

Data Inputs

- **AULI_IN.D1** (Data from socket AULI_IN) – the ULINT value to be converted.

Data Outputs

- **AR_OUT.D1** (Data for plug AR_OUT) – the converted REAL value.

Adapter

Name	Direction	Type	Description
AULI_IN	Socket	`adapter::types::unidirectional::AULI` (ULINT)	Input Value
AR_OUT	Plug	`adapter::types::unidirectional::AR` (REAL)	Output Value (Converted)

Functionality

The function block (FB) is implemented as a composite block and contains an instance of the conversion FB `F_ULINT_TO_REAL` from the IEC 61131 library.

Process:

1. An event on **AULI_IN.E1** triggers the execution of the internal conversion block `Convert`.
2. The current ULINT value from **AULI_IN.D1** is passed to input `Convert.IN`.
3. The internal FB performs the conversion from `ULINT` to `REAL` and stores the result at `Convert.OUT`. 4. After the conversion is complete, the event **AR_OUT.E1** is triggered, and the converted REAL value is available at **AR_OUT.D1**.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite structure** – enables reuse and encapsulation of the conversion logic.
- **Adapter-based** – no direct inputs/outputs; communication occurs via unidirectional adapters (socket/plug).
- **Use of the IEC 61131 standard function** `F_ULINT_TO_REAL` – guarantees platform-independent and correct type conversion.
- **Event-driven execution** – the conversion is only triggered by an input event when needed.
- **License** – Eclipse Public License 2.0 (see source code).

State Overview

The function block **does not have its own state machine**. The conversion is purely functional and event-driven via the internal function block. After each input event, the conversion is performed once, and the result is output.

Application Scenarios

- **Adapter bridge** between components that supply ULINT and those that expect REAL.
- **Data preparation** in control systems when physical quantities (e.g., counter readings as ULINT) need to be converted into floating-point values for calculations.
- **Modular networking** in IEC 61499 applications, where adapters serve as standardized interfaces between function blocks.

Comparison with similar function blocks

- **AULI_TO_LREAL** – analog conversion to `LREAL` (double precision).

- **Direct conversion function blocks** (e.g., `F_ULINT_TO_REAL`): These operate at the data level but require manual wiring of events and data. The **AULI_TO_AR** encapsulates this wiring and provides a compact adapter interface.
- **Other Adapter Converters** – there are corresponding function blocks for other type combinations (e.g., `LINT_TO_AR`) that are similarly structured.

Conclusion

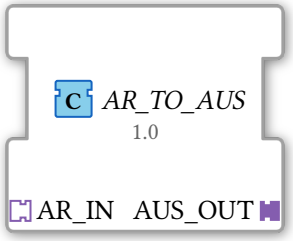
The **AULI_TO_AR** is a useful composite function block for simple and standardized type conversion from ULINT to REAL via adapter interfaces. Thanks to its event-driven execution and the use of an IEC 61131 standard function, it is reliable, reusable, and integrates seamlessly into IEC 61499 projects.

Numeric Index (unidirectional)

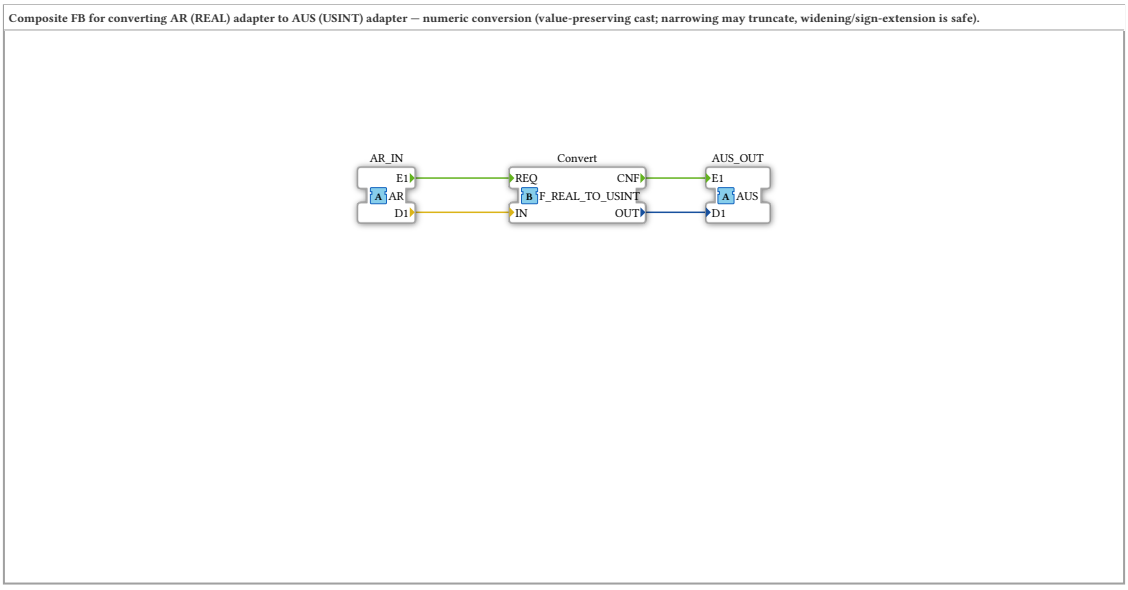
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AR_TO_AUS

REAL Adapter Input – *AR*



AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

The AR_TO_AUS function block is a composite block for converting a REAL value to a USINT value via a unidirectional adapter interface. It encapsulates the conversion process and provides a standardized connection between a REAL adapter socket and a USINT adapter plug. The function block is designed for use in automation systems that require data type conversion.

- **AR_IN.E1** (via socket `AR_IN`): Event to start the conversion of the incoming REAL value.
- **AUS_OUT.E1** (via Plug `AUS_OUT`): Event triggered after successful conversion and output of the USINT value.
- **AR_IN.D1** (via Socket `AR_IN`, Type: REAL): Input value to be converted to USINT.
- **AUS_OUT.D1** (via Plug `AUS_OUT`, Type: USINT): Converted output value.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- **Socket AR_IN** (Type: `adapter::types::unidirectional::AR`): Unidirectional REAL input adapter.

Functionality

The FB AR_TO_AUS functions as a pure data converter without its own state logic. Internally, the predefined block `F_REAL_TO_USINT` from the IEC 61131 standard library is used.

Process:

1. An incoming event at `AR_IN.E1` triggers processing.
2. The current value of `AR_IN.D1` (REAL) is passed to the input `IN` of the internal converter.
3. After the conversion is complete, the result value (USINT) is forwarded to `AUS_OUT.D1`.
4. Simultaneously, the event `AUS_OUT.E1` is triggered to activate downstream function blocks.

The entire data and event flow is unidirectional from the input to the output adapter.

- The function block is implemented as a composite and uses the function block `iec61131::conversion::F_REAL_TO_USINT` from the library `adapter::conversion::unidirectional`.
- No range check is performed; the conversion follows the rules of IEC 61131 (REAL → USINT: decimal places are truncated, values outside the range 0…255 may lead to undefined behavior).
- The function block does not have its own state machine and operates in an event-driven manner.
- The use of adapters enables loose coupling of the interfaces and facilitates reuse in different project contexts.

The function block AR_TO_AUS does not have its own state machine. The runtime behavior is entirely determined by the internal function block `F_REAL_TO_USINT`, which operates statelessly by default. Event control occurs directly from `AR_IN.E1` to `AUS_OUT.E1` without intermediate states.

- **Sensor-Actuator Coupling**: A REAL value (e.g., from an analog sensor) must be transmitted as a USINT to a binary actuator (e.g., a motor controller with an 8-bit value).
- **Interface Adaptation**: When an adapter of type `AR` (REAL) needs to be connected to an adapter of type `AUS` (USINT).
- **Data Preprocessing**: Before transferring data to systems that only accept USINT values (e.g., diagnostic or visualization components).
- **AR_TO_INT** / **AR_TO_BYTE**: Analogous structure for converting to other data types (INT, BYTE). This function block is limited to converting REAL to USINT.
- **Direct Converter Function Blocks** (e.g., `F_REAL_TO_USINT`): These do not offer adapter interfaces but require direct wiring. AR_TO_AUS encapsulates this converter and provides an ad-hoc adapter connection.
- **Bidirectional Converters**: In contrast to these, AR_TO_AUS operates strictly unidirectionally, which increases the clarity of the data flow.

The AR_TO_AUS function block offers a simple, modular solution for converting a REAL adapter to a USINT adapter. By using a standardized conversion block and encapsulating it as a composite function block, it can be easily integrated into existing 4diac projects. It is particularly suitable for scenarios requiring a clear separation of interfaces and unidirectional data transmission.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).

State Overview

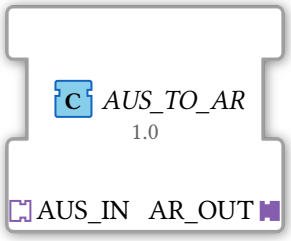
Application Scenarios

Comparison with Similar Function Blocks

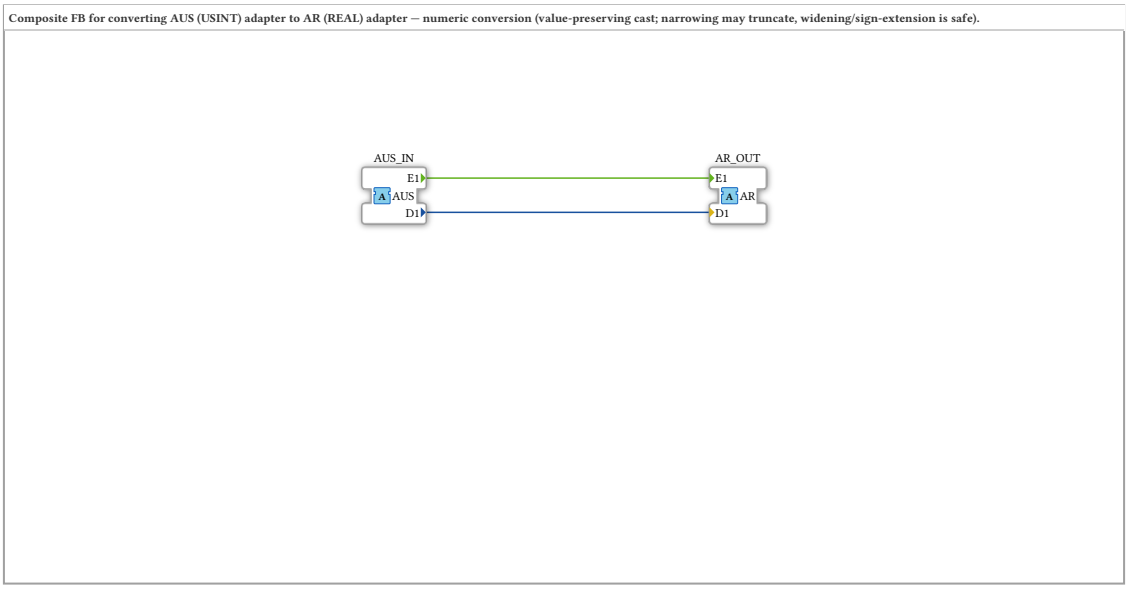
Conclusion

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

AUS_TO_AR



USINT Adapter Input – *AUS* *AR* – REAL Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUS_TO_AR** is a composite FB for converting an AUS adapter interface (based on USINT) to an AR adapter interface (based on REAL). According to its description, it serves to convert between these two adapter types. The current implementation (version 1.0) simply passes through event and data values without performing an explicit conversion – the actual type conversion must be implemented either through the adapter definitions themselves or through subsequent extensions.

Interface Structure

Event Inputs

No separate event inputs are available. Events are transmitted exclusively via the **AUS_IN** adapter.

Event Outputs

No separate event outputs are available. Events are output exclusively via the **AR_OUT** adapter.

Data Inputs

No separate data inputs are available. Data is transmitted exclusively via the **OFF_IN** adapter.

Data Outputs

No separate data outputs are available. Data is output exclusively via the **AR_OUT** adapter.

Adapters

Type	Name	Direction	Comment
`adapter::types::unidirectional::AUS`	OFF_IN	Socket (Input)	USINT-based adapter input
`adapter::types::unidirectional::AR`	AR_OUT	Plug (Output)	REAL-based adapter output

Functionality

This function block directly connects the event and data channels of the AUS_IN adapter to those of the AR_OUT adapter:

- **Event Forwarding**: The event arriving at socket `AUS_IN.E1` is forwarded without delay to plug `AR_OUT.E1`.
- **Data Forwarding**: The data value (type USINT) provided by socket `AUS_IN.D1` is passed directly to `AR_OUT.D1` (type REAL).

Since no type conversion or other processing is implemented in the network, the conversion from USINT to REAL is either implicitly handled by the 4diac runtime environment or must be implemented by adding a conversion function block (e.g., `CONV`) within an extension of the composite function block.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite without internal logic**: The function block network consists exclusively of two direct connections; it contains no other function blocks or state machines.
- **Adapter-based interface**: Instead of individual event/data inputs and outputs, all communication is handled via standardized unidirectional adapters. This facilitates reuse in modular adapter hierarchies.
- **Incomplete implementation**: The block's name and comment suggest active conversion, which is not implemented in the current version (1.0). This can lead to runtime errors if the runtime environment does not support implicit type conversion.

State overview

The block does not contain any state machines (ECCs). Its behavior is purely combinatorial and limited to the immediate forwarding of events and data.

Application Scenarios

- **Replacement for legacy systems**: An existing AUS adapter interface (USINT) needs to be connected to a component that expects an AR adapter interface (REAL).
- **Placeholder during the development phase**: The function block serves as a temporary connection until actual conversion logic is implemented.
- **Test environments**: Fast coupling of two adapter types without data modification for simulations or tests.

Comparison with similar function blocks

Typical conversion function blocks (e.g., `CONV` from the standard library) perform explicit value conversion using arithmetic or logical operations. `AUS_TO_AR`, on the other hand, delegates the conversion to the adapter level and does not perform any transformation itself. It is more of a **structural adaptation** than a true converter.

Conclusion

The function block `AUS_TO_AR` is a simple adapter-to-adapter through-pass block. Its usefulness depends heavily on whether the target system supports automatic type conversions between USINT and REAL.

If not, the function block must be extended with explicit conversion logic. It is therefore suitable as a temporary solution or as a foundation for a complete implementation.

🌐 Related topic subpages on ms-muc-docs.de

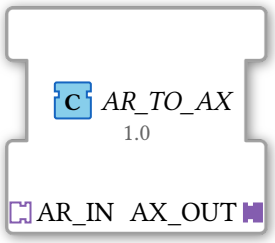
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numeric Index (unidirectional)

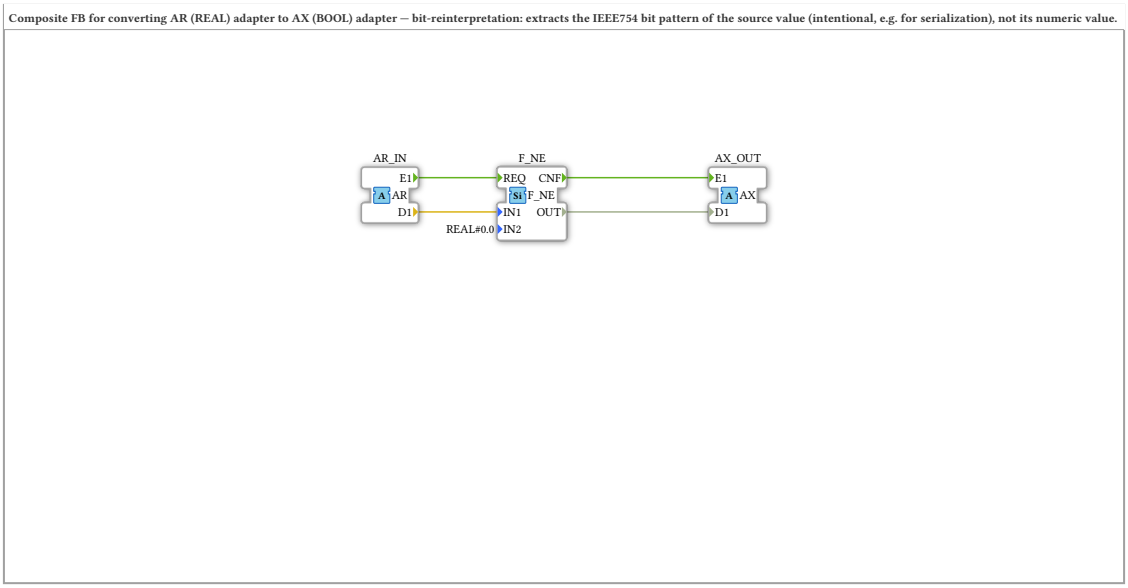
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AR_TO_AX

REAL Adapter Input – *AR*



AX – BOOL Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The AR_TO_AX function block converts a REAL value received via an AR adapter into a BOOL value and outputs it via an AX adapter. The conversion is performed by comparing the REAL value with 0.0. If the value is not zero, TRUE is output; otherwise, FALSE. The block serves as a bridge between analog and digital adapter interfaces.

Interface Structure

Event Inputs

No direct event inputs at the FB level. The event input is contained in the AR adapter (`AR_IN.E1`).

Event Outputs

No direct event outputs at the FB level. The event output is included in the AX adapter (`AX_OUT.E1`).

Data Inputs

No direct data inputs at the FB level. The data input is included in the AR adapter (`AR_IN.D1`).

Data Outputs

No direct data outputs at the FB level. The data output is included in the AX adapter (`AX_OUT.D1`).

Adapters

- **AR_IN** (Socket) – Adapter of type `adapter::types::unidirectional::AR`. Accepts a REAL value and an event to start processing.
- **AX_OUT** (Plug) – Adapter of type `adapter::types::unidirectional::AX`. Provides the BOOL result value and an event to confirm processing.

Functionality

The function block processes incoming REAL values as follows:

1. An event at `AR_IN.E1` triggers the internal comparison function block `F_NE` (not equal).
2. The REAL value received via `AR_IN.D1` is compared to the constant value 0.0.
3. The result of the comparison (BOOL) is forwarded via `F_NE.OUT` to `AX_OUT.D1`.
4. An event at output `F_NE.CNF` is sent to `AX_OUT.E1` to signal the availability of the BOOL value.

The internal function block `F_NE` is an IEC 61131-3 compliant comparator block and outputs TRUE if the input `IN1` is not equal to `IN2` (here 0,0).

Technical Features

- **Bit-reinterpretation**: extracts the IEEE754 bit pattern of the source value (intentional, e.g. for serialization), not its numeric value.
- Uses a composite function block approach, where the logic consists of a single comparator block and adapter connections.
- All inputs and outputs are implemented via standardized unidirectional adapters (`AR`, `AX`), enabling a modular and interchangeable interface definition.
- The comparator value (0,0) is fixed in the function block and cannot be changed at runtime.
- Execution is event-driven: A conversion is only performed when an event occurs at `AR_IN.E1`.

State Overview

Since the function block (FB) is implemented as a composition (network of basic building blocks), it does not have its own state machine. Its function is determined by the sequential processing of event and data flows in the internal network:

- **Waiting State**: No event at `AR_IN.E1`. The output value remains unchanged.
- **Processing State**: An event at `AR_IN.E1` triggers the comparison and generates an event at `AX_OUT.E1` as soon as the BOOL value is available.

Application Scenarios

- **Threshold Switch**: An analog sensor (e.g., a pressure sensor) provides REAL values. The FB outputs TRUE as soon as the value deviates from 0.0 (e.g., when a threshold is exceeded, if the zero point is calibrated).
- **Digitization of Analog Signals**: For connecting analog adapters to digital logic in control engineering.
- **Simple Plausibility Check**: Checks whether a REAL value is non-zero (e.g., a valid measured value).

Comparison with Similar Function Blocks

Unlike a generic `AR_TO_BOOL` function block, which uses direct data without an adapter, AR_TO_AX is specifically designed for use with adapter interfaces. A comparison block with an adjustable threshold (e.g., `F_GT`, `F_LT`) would offer more flexibility but would also require additional configuration effort. This function block provides a simple, fixed conversion: non-zero → TRUE.

Conclusion

The AR_TO_AX function block offers a compact solution for converting a REAL adapter value to a BOOL adapter value using zero comparison. Thanks to the use of adapters and a standardized comparison module, it can be easily integrated into existing 4diac projects and is well-suited for fast, unidirectional signal conversions.

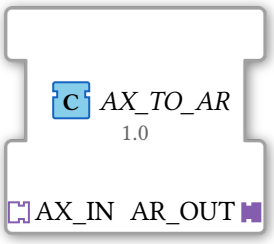
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

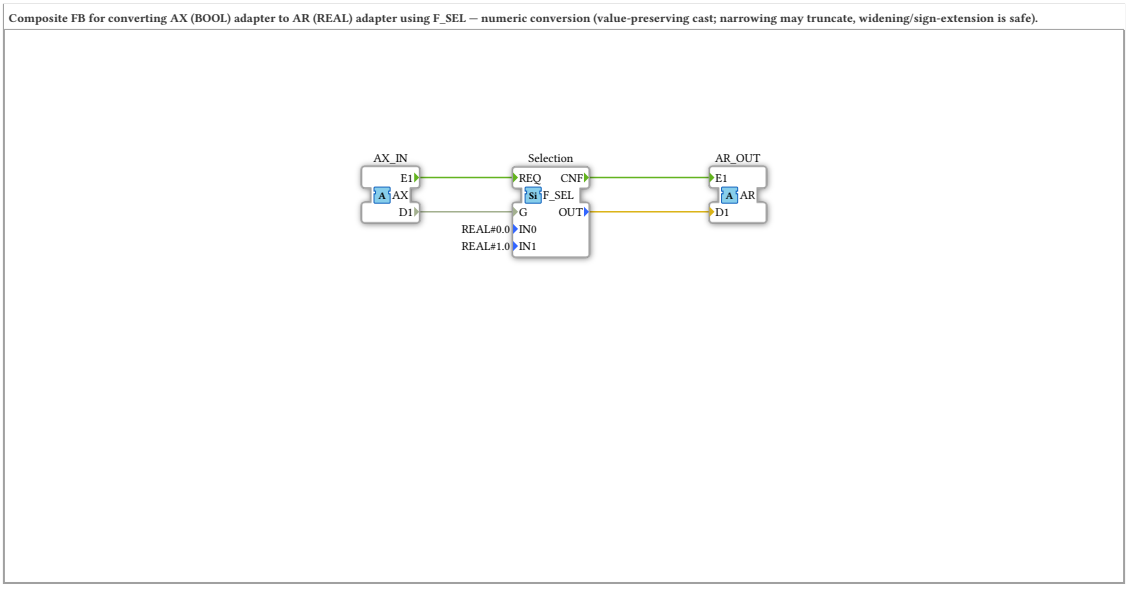
]

AX_TO_AR

BOOL Adapter Input – *AX*



AR – REAL Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **AX_TO_AR** function block is a composite function block that converts a Boolean signal from an AX adapter (type `adapter::types::unidirectional::AX`) into a REAL value (0.0 or 1.0) and outputs it via an AR adapter (type `adapter::types::unidirectional::AR`). The conversion is performed using the IEC 61131 function block **F_SEL**.

Interface Structure

Event Inputs

The function block does not have its own event inputs at the FB level. The internal process is controlled exclusively via the event of the AX_IN adapter, which is connected to the REQ input of the F_SEL.

Event Outputs

There are no event outputs at the FB level. The output event of F_SEL (CNF) is directly forwarded to the event input of the AR_OUT adapter.

Data Inputs

There are no direct data inputs at the FB level. The only data input is provided via the AX_IN adapter:

- **D1** (BOOL) – the Boolean signal to be converted.

Data Outputs

There are no direct data outputs at the FB level. The converted value is output via the AR_OUT adapter:

- **D1** (REAL) – the resulting REAL value (0.0 or 1.0).

Adapters

- **AX_IN** (Socket):
- Type: `adapter::types::unidirectional::AX`
- Provides a Boolean signal and an event pulse.
- **AR_OUT** (Plug):
- Type: `adapter::types::unidirectional::AR`
- Outputs the converted REAL value.

Functionality

The function block operates in an event-driven manner:

1. An incoming event at the AX_IN adapter (E1) triggers the internal F_SEL function block via its REQ input.
2. Simultaneously, the Boolean signal (AX_IN.D1) is applied to the G input of F_SEL.
3. F_SEL selects the following depending on G:

- For `G = FALSE` => Output of **IN0** (value `REAL#0.0`).
- For `G = TRUE` => Output of **IN1** (value `REAL#1.0`).

1. The result is passed via output OUT to data input D1 of the AR_OUT adapter.
2. The CNF event of F_SEL triggers output event E1 of the AR_OUT adapter, allowing the connected logic to accept the new data.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Reusability**: The function block is implemented as a composite function block and uses the standardized IEC 61131 function block `F_SEL` from the library `iec61131::selection`.
- **Adapter Coupling**: All communication takes place via unidirectional adapters (`adapter::types::unidirectional::AX` and `adapter::types::unidirectional::AR`), enabling loose coupling and easy reuse in different contexts.
- **Configuration**: The limit values (IN0=0.0, IN1=1.0) are fixed and can be changed as needed by adjusting the parameters of the internal F_SEL.

State Overview

The function block does not have an internal state machine. It reacts **event-driven** and performs a one-time selection for each event at the AX_IN adapter. There are no stored states – the behavior is purely combinatorial.

Application Scenarios

- Conversion of a binary sensor signal (e.g., switch, limit switch) into a numerical value for further processing in REAL-based control loops or visualizations.
- Integration of Boolean adapter signals into systems that operate entirely with REAL values.
- Simple mapping of digital inputs/outputs to analog values (0/1) in adapter-based automation solutions.

Comparison with Similar Function Blocks

- **Standard conversion function blocks** such as `BOOL_TO_REAL` from IEC libraries offer direct data-level conversion, but without adapter interfaces.
- **`SELECT`** (IEC 61131) can also use a Boolean signal to select between two REAL values, but requires manual wiring.
- **AX_TO_AR** encapsulates the complete adapter conversion in a reusable composite function block, thus significantly reducing the wiring effort in adapter network-based 4diac applications.

Conclusion

The **AX_TO_AR** function block provides a compact and robust solution for converting Boolean adapter signals into real values. By utilizing `F_SEL` and encapsulating it as a composite function block, it is easy to integrate, extensible, and ideally suited for adapter-oriented automation architectures.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

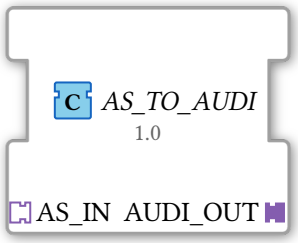
]

Numeric Index (unidirectional)

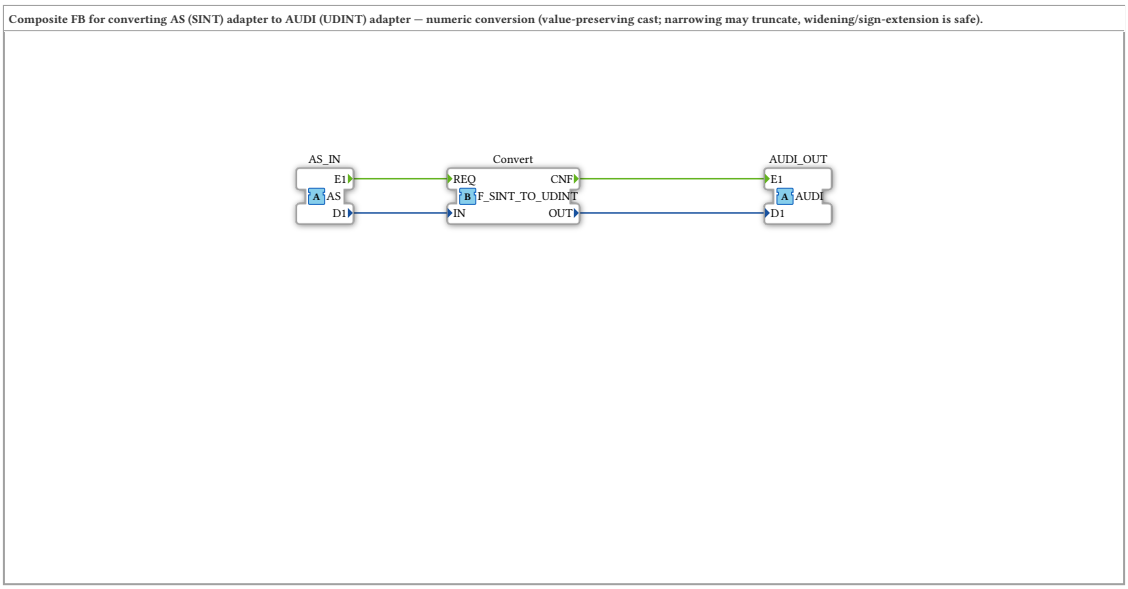
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AS_TO_AUDI

SINT Adapter Input – *AS*



AUDI – UDINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The AS_TO_AUDI function block is a composite function block that performs a conversion from a SINT adapter (AS) to a UDINT adapter (AUDI). It serves as a bridge between two different adapter types and enables the seamless integration of components that use different data types.

Interface Structure

Event Inputs

Not present. Event control is handled via the adapter socket **AS_IN**.

Event Outputs

Not present. Event output is handled via the adapter plug **AUDI_OUT**.

Data Inputs

Not present. Data acquisition is handled via the adapter socket **AS_IN**.

Data Outputs

Not present. Data output is provided via the adapter plug **AUDI_OUT**.

Adapter

Name	Type	Direction	Description
AS_IN	`adapter::types::unidirectional::AS`	Socket (Input)	SINT adapter that provides an input value (SINT) and a corresponding event.
AUDI_OUT	`adapter::types::unidirectional::AUDI`	Plug (Output)	UDINT adapter that outputs the converted value (UDINT) and an event.

The adapters each contain one data element and one event element:

- **AS_IN**: Data `D1` (SINT), Event `E1`.
- **AUDI_OUT**: Data `D1` (UDINT), Event `E1`.

Functionality

The module internally uses the IEC 61131 function `F_SINT_TO_UDINT` to convert the incoming SINT value into a UDINT value. The process is event-driven:

1. An event at element `E1` of socket **AS_IN** (e.g., from an upstream function block) is forwarded to input `REQ` of the internal conversion function block `Convert`.
2. The conversion function block reads the current value of `AS_IN.D1` and converts it into a UDINT value.
3. After successful conversion, the converter's output event `CNF` is triggered, which in turn activates the event at element `E1` of plug **AUDI_OUT**. ...3. The output event is triggered by the converter, which in turn activates the event at element `E1` of plug **AUDI_OUT**.
4. The output event is triggered by an event at element `E1` of plug **AUDI_OUT**.
5. The output event is triggered by an event at element `E1` of plug **AUDI_OUT**.
6. The event is forwarded to the input `Convert` of the internal conversion function block `Convert`.
7. The event at element qzmsdocs00008qz of socket **AS_IN** (e 4. Simultaneously, the converted value is written from `Convert.OUT` to `AUDI_OUT.D1`.

This ensures that the converted UDINT value is only available at the output when the corresponding event is signaled.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Pattern**: The function block is implemented as a composite (composed functional block) that encapsulates a single conversion function block. This results in a reusable and encapsulated unit.
- **Use of IEC 61131 Function Blocks**: The conversion utilizes the standardized function `F_SINT_TO_UDINT`, which promotes portability and traceability.
- **Unidirectional Adapters**: Both the input and output adapters are unidirectional (no feedback). This simplifies the data flow direction.

State Overview

The composite block does not have its own state machine (ECC). The entire sequence control is defined by the event chaining of the internal network. Therefore, there are no defined states or state transitions.

Application Scenarios

- **System Integration**: When a subsystem provides data in SINT format via an adapter, but a subsequent subsystem expects UDINT data via an adapter.
- **Protocol Adaptation**: In heterogeneous automation systems where different data types are exchanged via adapters, this block can serve as a universal converter.
- **Network Structure Simplification**: Instead of performing the conversion directly in the application blocks, it is encapsulated in a separate composite, which improves maintainability.

Comparison with Similar Blocks

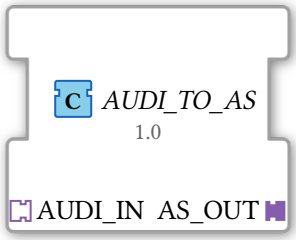
Similar blocks exist for other data type conversions, e.g.,... For example, `AS_TO_ABOOL` (SINT to BOOL) or `AUINT_TO_AUDI` (UINT to UDINT). The present function block differs in its specific combination of SINT input and UDINT output. It adds another conversion option to the library and follows the same design pattern (composite with an internal IEC 61131 converter).

Conclusion

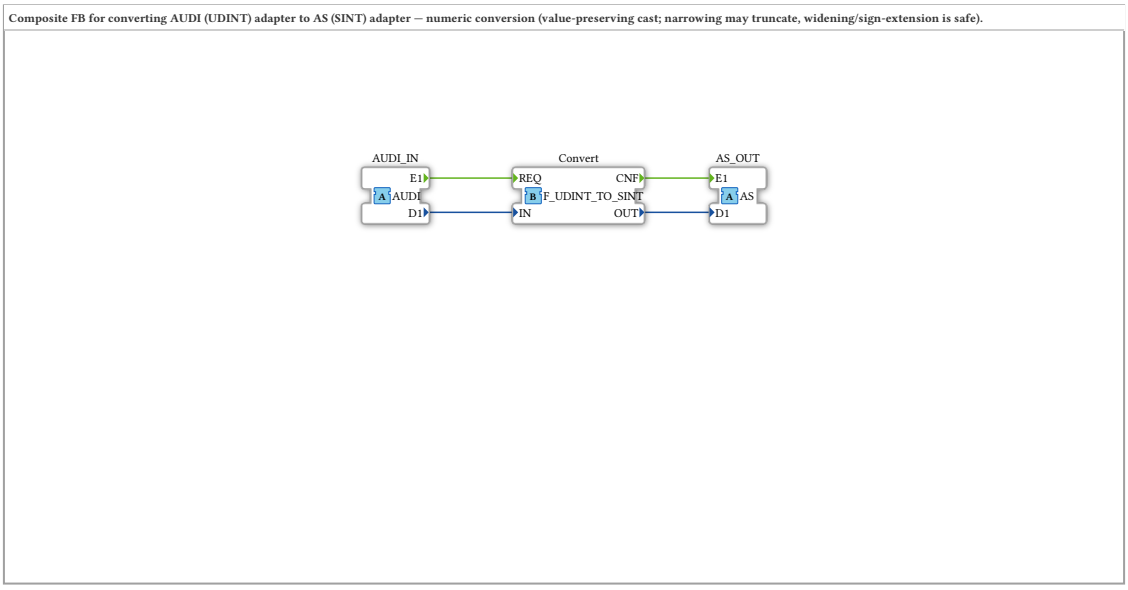
The function block `AS_TO_AUDI` offers an elegant solution for converting SINT adapter data to UDINT adapter data. Encapsulating it in a composite function block improves the reusability and clarity of the overall system. The use of standardized conversion functions ensures that the conversion is correct and efficient.

AUDI_TO_AS

UDINT Adapter Input – *AUDI*



AS – SINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_TO_AS** is a composite block that converts an AUDI adapter (data type `UDINT`) into an AS adapter (data type `SINT`). It serves as a bridge between two different adapter interfaces and enables the lossy conversion of an unsigned 32-bit value into a signed 8-bit value. The conversion is event-driven using the internal conversion function `F_UDINT_TO_SINT`.

Interface Structure

Event Inputs

The function block does not have its own dedicated event inputs. Event control is handled via the **socket adapter `AUDI_IN`**, which provides an event `E1`.

Event Outputs

The function block (FB) does not have its own independent event outputs. The result event is output via the **plug adapter `AS_OUT`** using the event `E1`.

Data Inputs

Data inputs are available exclusively via the **socket adapter `AUDI_IN`**:

- **`D1`** (data type `UDINT`): The 32-bit value to be converted.

Data Outputs

Data outputs are provided via the **`AS_OUT`** plug adapter:

- **`D1`** (data type `SINT`): The converted 8-bit value (value range −128…127).

> Note: A `UDINT` value greater than 127 is either truncated or interpreted during conversion (depending on the implementation of `F_UDINT_TO_SINT`).

Adapter

Name	Type	Direction	Description
`AUDI_IN`	`adapter::types::unidirectional::AUDI`	Socket	Input adapter (UDINT)
`AS_OUT`	`adapter::types::unidirectional::AS`	Plug	Output adapter (SINT)

Functionality

This function block acts as a transparent conversion stage between two unidirectional adapters. An incoming event on `AUDI_IN.E1` triggers the internal function block `F_UDINT_TO_SINT`, which converts the `UDINT` value at `AUDI_IN.D1` into a `SINT` value. After the conversion is complete, an event is output to `AS_OUT.E1`, and the converted value is made available to `AS_OUT.D1`.

The internal process is purely causal and requires no separate states or timings – the response occurs immediately after the input event.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-based interface**: The function block communicates exclusively via special adapters, enabling loose coupling and reusability in different environments.
- **Data loss possible**: Since `SINT` only contains 8 bits, over 127 bits of information are lost with a `UDINT` input value. The exact mapping (e.g., clipping or modular reduction) depends on the internal conversion function.
- **No Internal Logic**: The function block consists entirely of the interconnection of the existing conversion function block and two adapters – a pure composition type.

State Overview

The function block does not have its own state machine. It behaves statelessly: Each incoming event at the socket triggers exactly one output at the plug. Processing is deterministic and event-synchronous.

Application Scenarios

- **Protocol Adaptation**: When a subsystem delivers data as `UDINT` via an AUDI adapter, but the target system only accepts `SINT` via an AS adapter (e.g., with 8-bit actuator controllers).
- **Value Range Reduction**: Explicitly restricting a large number range to a smaller one to ensure compatibility with older or resource-constrained components.
- **Testing and Simulation**: Use in test environments where adapter interfaces of different data types interact.

Comparison with similar function blocks

- **`UDINT_TO_SINT` (direct)**: The pure conversion function block without adapter integration. `AUDI_TO_AS` encapsulates this function block and adds the specific adapters.
- Other adapter converters (e.g., `DINT_TO_AS`, `AUDI_TO_INT`) handle different data type combinations. This function block is specifically optimized for the combination `AUDI` ↔ `AS`.

Conclusion

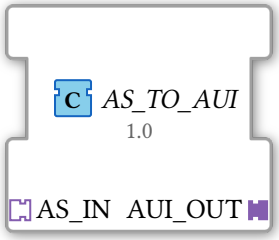
The composite function block `AUDI_TO_AS` provides a simple yet effective way to convert a `UDINT`-based adapter to a `SINT`-based adapter. Thanks to its clear event control and the use of a proven conversion function, it is suitable for all scenarios where lossy data type conversion between these two adapter standards is required.

Numeric Index (unidirectional)

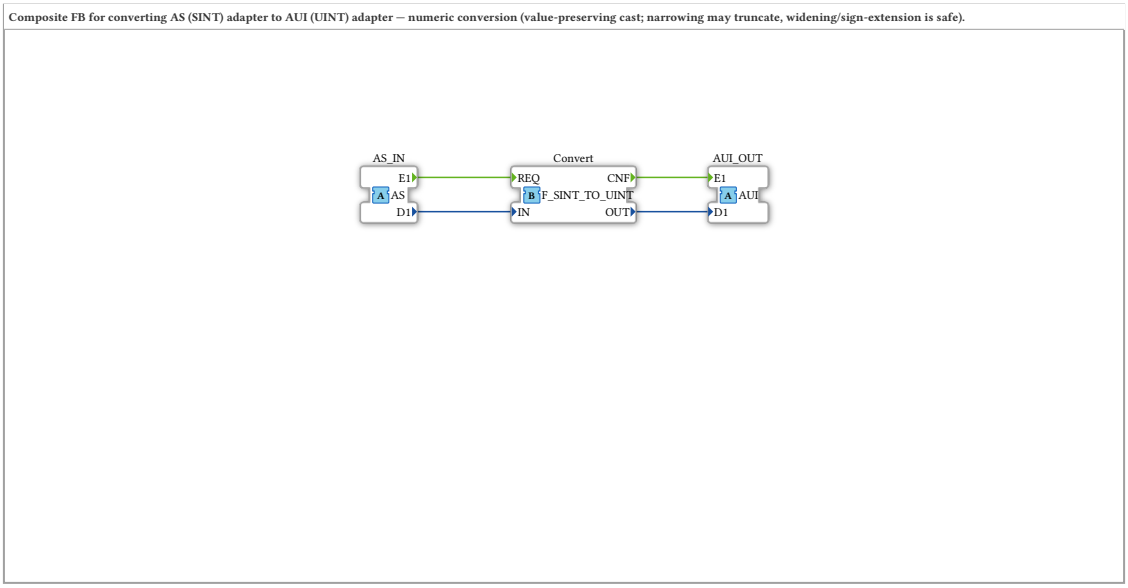
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AS_TO_AUI

SINT Adapter Input – *AS*



AUI – UINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `AS_TO_AUI` is a composite function block (FB) that converts a SINT adapter input (`AS`) into a UINT adapter output (`AUI`). It encapsulates the type conversion and enables a clean, adapter-based interface between components that use different data types.

Interface Structure

Event Inputs

Name	Data Type	Description
`E1` (via `AS_IN`)	EVENT	Starts the conversion of the incoming SINT value.

Event Outputs

Name	Data Type	Description
`E1` (via `AUI_OUT`)	EVENT	Confirms the completion of the conversion and output of the UINT value.

Data Inputs

Name	Data Type	Description
D1 (via AS_IN)	SINT	The signed 8-bit value to be converted.

Data Outputs

Label	Data Type	Description
D1 (via AUI_OUT)	UINT	The converted unsigned 16-bit value.

Adapters

Label	Type	Direction	Description
AS_IN	adapter::types::unidirectional::AS	Socket (Input)	Provides the SINT value and the triggering event.
AUI_OUT	adapter::types::unidirectional::AUI	Plug (Output)	Outputs the converted UINT value and the acknowledgment event.

Functionality

The function block operates in an event-driven manner:

1. When the event E1 arrives at the AS_IN adapter, the incoming SINT data value (D1) is forwarded to the internal conversion function block F_SINT_TO_UINT.
2. The F_SINT_TO_UINT function block performs the data type conversion according to IEC 61131 (SINT → UINT).
3. Once the conversion is complete, the resulting UINT value is passed to the AUI_OUT adapter, and the event E1 is triggered on AUI_OUT.

The entire process is atomic within an event cycle: event in, data conversion, event out.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- The function block internally uses the IEC 61131 conversion function F_SINT_TO_UINT, which is derived from the iec61131::conversion library.
- The adapters are designed as **unidirectional** types (AS for input, AUI for output), which clearly defines the data flow direction and avoids misunderstandings.
- There are no internal states or timing behaviors – the conversion is purely combinatorial, controlled by the event.
- The function block is licensed under the Eclipse Public License 2.0.

State Overview

Since this is a composite function block without its own state machine, there is no internal state machine. The function block is completely event-driven and executes the conversion immediately upon each trigger.

•

Application Scenarios

- **Connecting Different Bus Systems**: If a sensor (e.g., AS-Interface) delivers SINT values, but the controller expects UINT values, this function block can handle the translation.
- **Protocol or Adapter Compatibility**: Use in heterogeneous automation systems where different fieldbuses or components operate with different data types.
- **Data Preparation**: Preparation of SINT data for further processing that only supports UINT.

Comparison with Similar Function Blocks

- **Direct Conversion Function Blocks** (e.g., `F_SINT_TO_UINT`): These operate at the level of individual signals without adapter encapsulation. `AS_TO_AUI` integrates the adapter interface and is therefore easier to integrate into adapter-based architectures.
- **Bidirectional Adapter Converters**: If conversion in both directions is required, a separate module (e.g., `AUI_TO_AS`) must be used. `AS_TO_AUI` is strictly unidirectional.

Conclusion

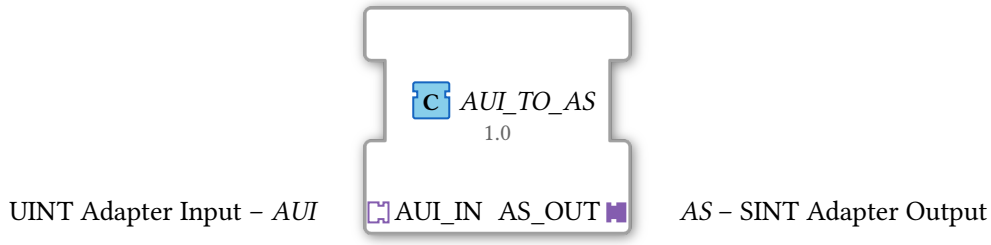
The `AS_TO_AUI` module provides a simple, reliable, and standardized method for converting SINT adapter data to a UINT adapter format. Thanks to the use of IEC 61131 modules and unidirectional adapters, it is robust, easy to understand, and integrates well into existing 4diac projects.

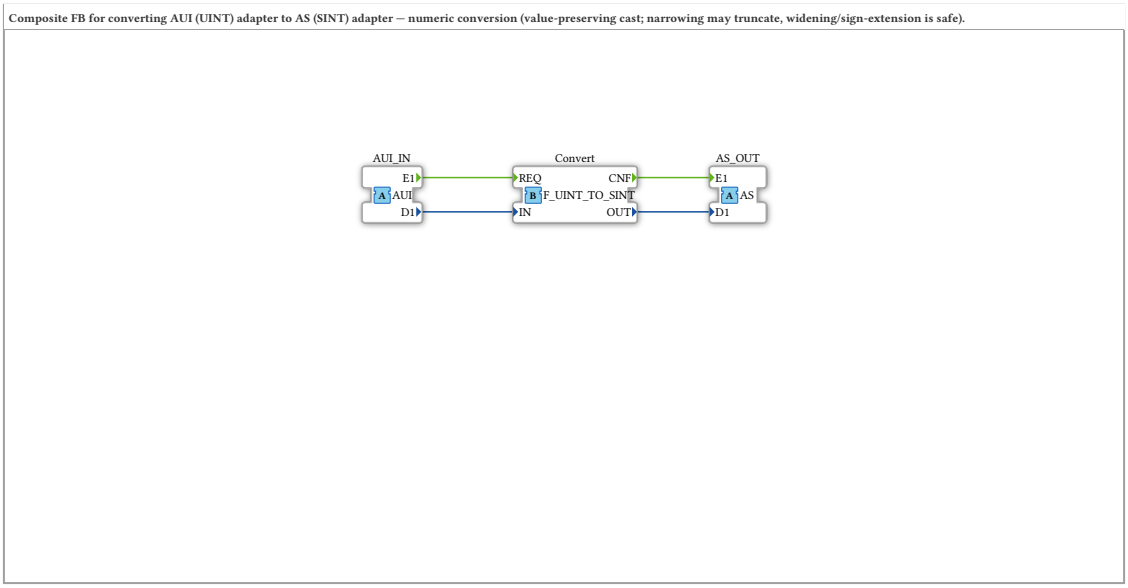
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUI_TO_AS





• – ○ • – ○ • – ○ •

Introduction

The function block `AUI_TO_AS` is a composite block that enables the conversion of an **AUI (UINT)** adapter to an **AS (SINT)** adapter. It encapsulates the necessary signal and data conversion in a reusable unit and provides the converted values via a single adapter output.

Interface Structure

Event Inputs

The block has **no direct event inputs** at its top level.

The triggering event is provided via socket `AUI_IN` (adapter `adapter::types::unidirectional::AUI`). This event (labeled `E1`) starts the conversion internally.

Event Outputs

At the top level, there are **no direct event outputs**.

The conversion completion event is output externally via plug `AS_OUT` (adapter `adapter::types::unidirectional::AS`) as event `E1`.

Data Inputs

There are **no direct data inputs**.

The input data (data type `UINT`) is fed into the function block via socket `AUI_IN` (channel `D1`).

Data Outputs

There are also **no direct data outputs**. The converted data (data type `SINT`) exits the module via plug `AS_OUT` (channel `D1`).

Adapter

Direction	Name	Type	Description
Socket	AUI_IN	`adapter::types::unidirectional::AUI`	Input adapter (UINT data)
Plug	AS_OUT	`adapter::types::unidirectional::AS`	Output adapter (SINT data)

Functionality

The function block functions as a pure encapsulation of a conversion process:

1. An incoming event via `AUI_IN.E1` triggers the internal function block `Convert` (of type `iec61131::conversion::F_UINT_TO_SINT`).
2. Simultaneously, the data value from `AUI_IN.D1` (of type `UINT`) is forwarded to the input `Convert.IN`.
3. After the conversion is complete, `Convert.CNF` sends an event to `AS_OUT.E1`.
4. The converted value (`SINT`) is transferred from `Convert.OUT` to `AS_OUT.D1`.

This enables end-to-end, event-driven conversion from a UINT adapter to a SINT adapter, without requiring the user to implement the conversion details themselves.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block:** All connections are encapsulated within a network. The internal logic remains hidden from the calling environment.
- **Use of the Standard Conversion Function:** The actual conversion is performed by the IEC 61131 block `F_UINT_TO_SINT`. This block is externally referenced and imported as a library block.
- **Unidirectional Adapters:** Both the input and output adapters are unidirectional, meaning data flows in only one direction (from input to output).
- **No Own States:** Because this is a composite device, it does not have its own Execution Control Chart (ECC). Its behavior is determined solely by its internal connections.

State Overview

This device does **not have its own state machine**.

Its operation is purely data flow and event-driven through the interconnection of its constituent elements. There are no defined states such as IDLE, BUSY, or ERROR.

Application Scenarios

- **Adapter Bridging:** If an application uses adapters with different data types (e.g., AUI with `UINT` and AS with `SINT`), this device can be used as a converter between the adapter interfaces.
- **System Integration:** In heterogeneous control systems where components use different data types, this function block facilitates integration without manual type conversion.

- **Reuse:** The composite function block can be used multiple times in different projects or networks to always provide the same conversion logic.

Comparison with Similar Function Blocks

- **Other Conversion Adapters:** Similar function blocks exist, such as `SINT_TO_UINT_AS` or generic type converters. The difference lies in the specific type combination (UINT → SINT) and the use of adapters instead of direct event/data inputs.
- **Direct Conversion Without Adapters:** Instead of an adapter-based function block, one could also use the function `F_UINT_TO_SINT` directly and wire the adapter interfaces separately. The composite function block simplifies this through encapsulation.

Conclusion

The function block `AUI_TO_AS` offers a simple, encapsulated solution for converting a unidirectional UINT adapter to a SINT adapter. By utilizing a standardized IEC 61131 conversion function and clear event handling, it integrates seamlessly into 4diac IDE projects. It is particularly suitable for scenarios where adapters of different data types need to be connected without requiring the user to implement the conversion logic themselves.

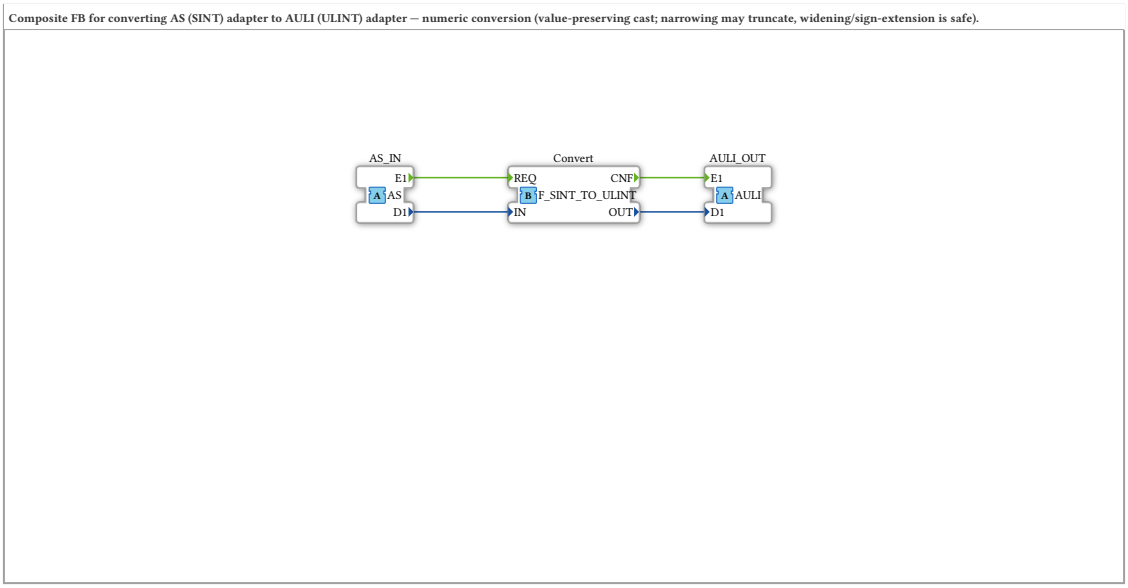
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Numeric Index (unidirectional)

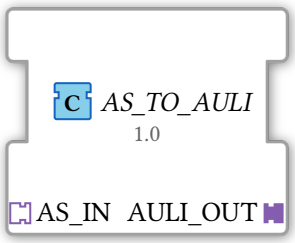
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AS_TO_AULI



Introduction

SINT Adapter Input – *AS*



AULI – ULINT Adapter Output

The composite function block `AS_TO_AULI` converts an **AS (SINT) adapter** to an **AULI (ULINT) adapter**. Internally, it uses the IEC 61131 conversion block `F_SINT_TO_ULINT` to perform the actual type conversion

from SINT to ULINT. The block is designed as part of a unidirectional adapter library for communication between different data types

---

Interface Structure

Event Inputs

The block has **no direct event inputs**. Events are received via the **Socket adapter `AS_IN`**:

- **`AS_IN.E1`** – Event input via the AS adapter, which triggers the data conversion.

Event Outputs

This function block has **no direct event outputs**. Events are output via the **Plug adapter `AULI_OUT`**:

- **`AULI_OUT.E1`** – Event output via the AULI adapter, which is activated after the conversion is complete.

Data Inputs

This function block has **no direct data inputs**. Data input is via the **Socket adapter `AS_IN`**:

- **`AS_IN.D1`** – Data input of type **SINT**, which provides the value to be converted.

Data Outputs

The module has **no direct data outputs**. Data output is via the **Plug adapter `AULI_OUT`**:

- **`AULI_OUT.D1`** – Data output of type **ULINT**, which provides the converted value.

Adapters

Adapter	Type	Direction	Description
`AS_IN`	`adapter::types::unidirectional::AS` (Socket)	Input	Receives SINT values and trigger event `E1`.
`AULI_OUT`	`adapter::types::unidirectional::AULI` (Plug)	Output	Sends ULINT values and acknowledgment event `E1`.

Functionality

1. The function block expects an event at the socket adapter `AS_IN.E1`. Simultaneously, the data value `AS_IN.D1` (type SINT) must be present. ... 2. The event is forwarded to the internal function block `F_SINT_TO_ULINT` (as `REQ`).
2. The internal function block performs the conversion **SINT → ULINT**.
3. After successful conversion, `F_SINT_TO_ULINT` sends an acknowledgment event (`CNF`) to the plug adapter `AULI_OUT.E1`.
4. The converted value (type ULINT) is output via `AULI_OUT.D1`.

The conversion is performed according to the IEC 61131-3 standard: The SINT value (8-bit signed) is converted into an unsigned 64-bit value (ULINT). Negative SINT values result in a negative ULINT representation (two's complement conversion to 64 bits).

Technical Specifications

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Standard Compliance**: The component is specified according to IEC 61499-2.
- **License**: Eclipse Public License 2.0 (EPL-2.0).
- **Dependency**: Imports the IEC 61131 component `F_SINT_TO_ULINT` from the package `iec61131::conversion`.
- **Type Safety**: The adapters `AS` and `AULI` ensure type-safe connections; direct cabling from SINT to ULINT without conversion is not possible.

- **Unidirectional Data Flow**: The function block operates purely unidirectionally – it receives data and events and outputs converted data and acknowledgments.

State Overview

The function block does **not have its own state machine**. It functions purely as an interconnect for the inner `F_SINT_TO_ULINT` function block. Its behavior is strictly event-driven:

- **Waiting** (Initial State) ⟶ Event sent to `AS_IN.E1` ⟶ Transition to **Conversion**.
- **Conversion** ⟶ Internal function block operates asynchronously ⟶ After completion ⟶ Event sent to `AULI_OUT.E1` ⟶ Return to **Waiting**.

Error handling is not explicitly modeled; the inner function block outputs a defined result (e.g., 0) for invalid inputs (e.g., outside the value range).

Application Scenarios

- **Integration into Adapter Chains**: If an upstream function block provides a SINT value, but subsequent components only process ULINT, `AS_TO_AULI` can be used as an intermediate adapter.
- **Sensor Connection**: Sensors with 8-bit output (e.g., rotary encoders, simple switches) can be connected to a 64-bit bus.
- **Legacy Systems**: This adapter makes legacy systems with SINT interfaces compatible with modern ULINT-based controllers.

–

Comparison with Similar Function Blocks

Function Block	Target Type	Source Type	Special Feature
`AS_TO_AULI`	ULINT	SINT	SINT to ULINT conversion
AS_TO_AUDI`	UDINT	SINT	SINT to UDINT conversion (32-bit)
AS_TO_AWORD`	WORD	SINT	SINT to WORD conversion (16-bit)
AULI_TO_AS`	SINT	ULINT	ULINT to SINT conversion (with overflow limitation)

This function block covers the conversion from an 8-bit signed type to a 64-bit unsigned type and is particularly suitable for applications requiring a large output range.

Conclusion

AS_TO_AULI` is a simple yet important composite module for type conversion between SINT and ULINT adapters. It encapsulates the IEC 61131 conversion and enables type-safe, event-driven integration into automation systems. Its clear interface structure and use of standard libraries make it maintainable, reusable, and easily integrated into existing 4diac IDE projects.

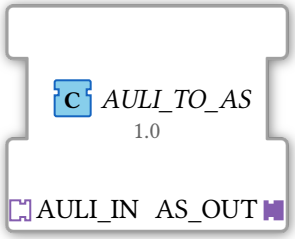
–

🌐 Related topic subpages on ms-muc-docs.de

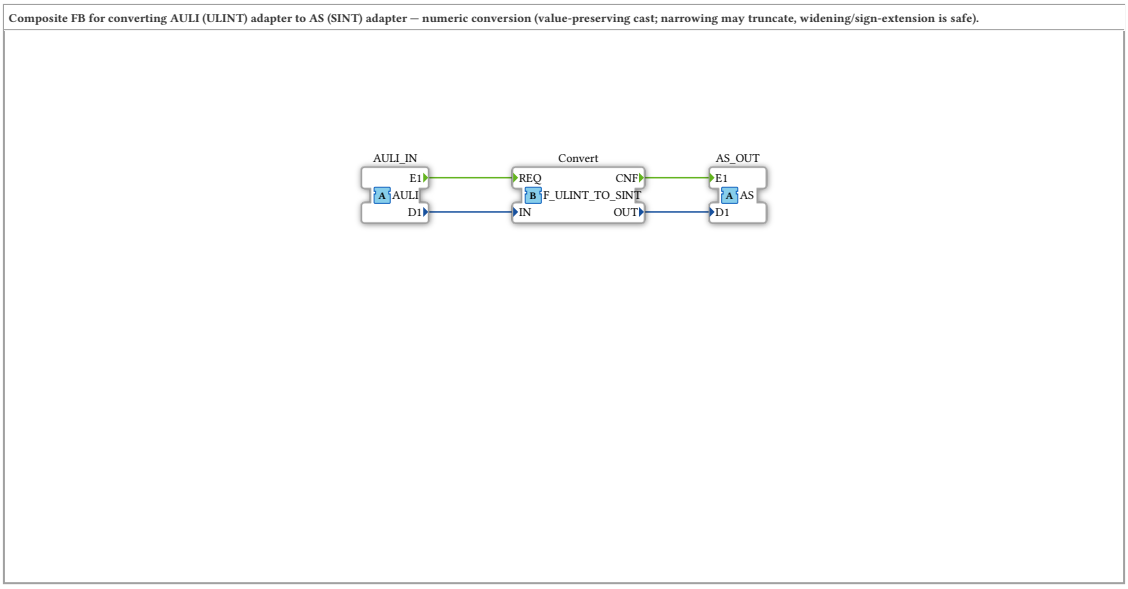
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AULI_TO_AS

ULINT Adapter Input – *AULI*



AS – SINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AULI_TO_AS** function block is a composite module for converting a unidirectional **AULI** adapter (ULINT) to a unidirectional **AS** adapter (SINT). It encapsulates the type conversion of unsigned 64-bit integers to signed 8-bit integers and provides seamless connectivity between components with different data type requirements in the 4diac framework.

Interface Structure

The module does not have separate event or data ports at the top level. All communication takes place exclusively via two adapters.

Adapters

Adapter	Direction	Type	Description
`AULI_IN`	Socket (Input)	`adapter::types::unidirectional::AULI`	Provides an event `E1` and a data value `D1` of type `ULINT`.
`AS_OUT`	Plug (Output)	`adapter::types::unidirectional::AS`	Provides an event `E1` and a data value `D1` of type `SINT`.

Event Inputs (via adapter)

- `AULI_IN.E1` – Starts the conversion of the incoming ULINT value.

Event Outputs (via adapter)

- `AS_OUT.E1` – Confirms completion of the conversion and indicates that the new SINT value is valid.

Data Inputs (via adapter)

- `AULI_IN.D1` – ULINT value to be converted (value range 0 … 2^{24-1}).

Data Outputs (via adapter)

- `AS_OUT.D1` – Conversion result as a SINT value (value range −128 … 127).

Functionality

The module internally uses the IEC 61131 conversion function `F_ULINT_TO_SINT`. When the input adapter receives the event `E1`, the current value of `AULI_IN.D1` is read and converted. The result is then passed to `AS_OUT.D1`, simultaneously triggering the event `AS_OUT.E1`.

The processing is purely event-driven and occurs without intermediate storage of internal states.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Value Range Loss:** Since `ULINT` covers a very large range of values (0 … 2^{64}−1), while `SINT` can only represent −128 … 127, conversion for values outside the SINT range leads to overflow or saturation (according to the IEC 61131 definition). This should be taken into account during application.
- **Unidirectional Adapters:** The block is designed for data flow in one direction – from the input to the output adapter. It cannot be used for bidirectional communication.
- **License:** The block is provided under the Eclipse Public License 2.0.

State Overview

The block does not have an explicit state machine or internal state registers. It behaves like a purely combinatorial function triggered by an event. There is no persistent storage of values or processing states.

Application Scenarios

- **Sensor Integration:** A sensor delivers data in ULINT format, but a downstream module expects SINT values (e.g., for a simple display or threshold logic).
- **Protocol Adaptation:** Heterogeneous automation systems where adapters use different data types can be connected using this function block in a type-compatible manner.
- **Data Reduction:** Targeted conversion of large numerical ranges into smaller formats to save memory or bus bandwidth (at the expense of accuracy).
- **Sensor Integration:**

Comparison with Similar Function Blocks

- **`AULI_TO_INT`** – Converts to `INT` (16-bit, −32768 … 32767), larger value range than SINT, but still lossy with large numbers.
- **`AULI_TO_DINT`** – Converts to `DINT` (32-bit, -2^{31} … $2^{31}-1$), covers a significantly larger portion of the ULINT range and is the better choice for many applications.
- **`ULINT_TO_SINT` (direct)** – A simple function block without adapter encapsulation; `AULI_TO_AS` provides the same service in an adapter-based environment.

Conclusion

The **AULI_TO_AS** is a lean, purpose-built converter for unidirectional adapters. It enables the easy integration of components based on the SINT data type into a ULINT-dominated system. However, due to the large difference in value ranges, users must consider potential overflow or saturation effects and, if necessary, switch to wider data types.

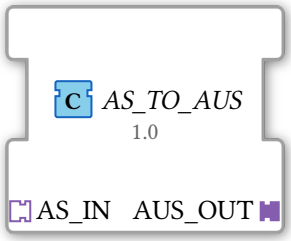
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Numeric Index (unidirectional)

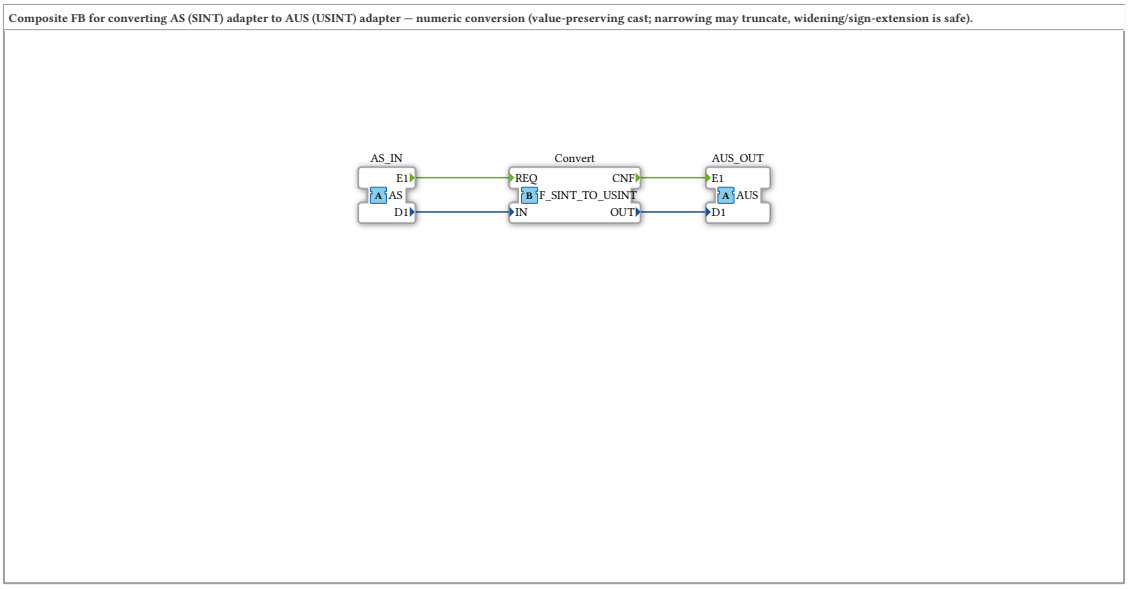
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AS_TO_AUS



SINT Adapter Input – *AS*

AUS – USINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block **AS_TO_AUS** is a composite block that connects an AS adapter (type `adapter::types::unidirectional::AS`) to an AUS adapter (type `adapter::types::unidirectional::AUS`) and converts the data values contained within from `SINT` (signed 8-bit) to `USINT` (unsigned 8-bit). The conversion is performed by the embedded standard function block `F_SINT_TO_USINT` from the IEC 61131 library. This function block enables simple, adapter-based data type conversion in industrial control applications.

Interface Structure

Event Inputs

No direct event inputs. Event control is handled exclusively via the `AS_IN` adapter (socket).

Event Outputs

No direct event outputs. Event transmission is handled exclusively via the `AUS_OUT` adapter (plug).

Data Inputs

No direct data inputs. Data acquisition is handled exclusively via the `AS_IN` adapter.

Data Outputs

No direct data outputs. Data output is handled exclusively via the `AUS_OUT` adapter.

Adapters

Adapter	Type	Direction	Description
`AS_IN`	`adapter::types::unidirectional::AS`	Socket (Input)	Receives a SINT value and an event (E1).
`AUS_OUT`	`adapter::types::unidirectional::AUS`	Plug (Output)	Outputs the converted USINT value and an event (E1).

Functionality

The function block operates as a pure data flow converter:

1. An external event at socket `AS_IN.E1` triggers the conversion.
2. Simultaneously, the corresponding data value `AS_IN.D1` (SINT) is passed to the input `IN` of the internal block `F_SINT_TO_USINT`.

3. The conversion block converts the SINT value to the corresponding USINT value and places it at its output `OUT`.
4. The converter's completion event (`CNF`) triggers the event `AUS_OUT.E1` and passes the converted value to `AUS_OUT.D1`.
5. The output adapter makes the result available to the calling environment.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block**: The function block (FB) does not contain its own state logic but combines the existing standard converter with the predefined adapter interfaces.
- **Unidirectional Adapters**: Both the input and output adapters are unidirectional – data and events flow in only one direction.
- **Type Conversion**: The conversion from SINT to USINT is lossless, as the value range of USINT (0…255) completely covers the positive part of SINT (–128…127). However, negative SINT values are interpreted as follows: e.g., –1 becomes 255 (two's complement). This behavior corresponds to the IEC 61131 function `SINT_TO_USINT`.

State Overview

The FB does not have an internal state machine (ECC). The entire sequence control is implemented through event chaining in the network.

Application Scenarios

- **Sensor Integration**: A sensor delivers a SINT value (e.g., temperature as a signed value), but the subsequent module expects USINT (e.g., for displays or logic with unsigned types).
- **System Coupling**: Between different control components that communicate via adapters, a signed representation must be converted to an unsigned representation.
- **Protocol Adaptation**: In fieldbus or OPC UA connections where the data formats are predefined.

Comparison with Similar Function Blocks

- **F_SINT_TO_USINT**: This standard function block performs pure type conversion without an adapter. This function block encapsulates it in an adapter interface so that it can be directly integrated into adapter-based architectures.
- **Other Converters** (e.g., `F_INT_TO_DINT`): These work analogously, but for different data types. The advantage of `AS_TO_AUS` lies in its specific adapter adaptation for the AS and AUS types.

Conclusion

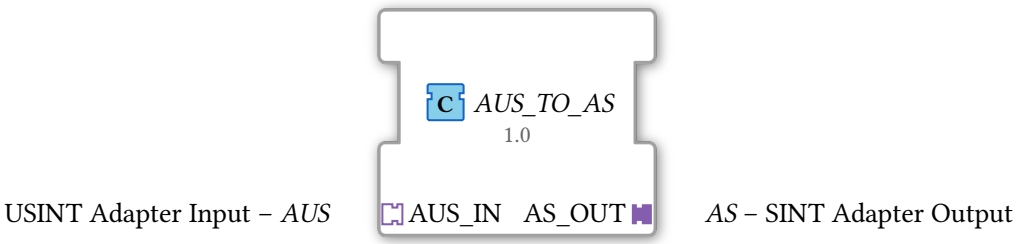
The function block `AS_TO_AUS` is a practical tool for type conversion in adapter-based 4diac applications. It simplifies the integration of components that use different signed and unsigned 8-bit data types and enables a clean separation of the adapter interfaces. Thanks to the use of the established IEC converter, the functionality is reliable and standards-compliant.

–

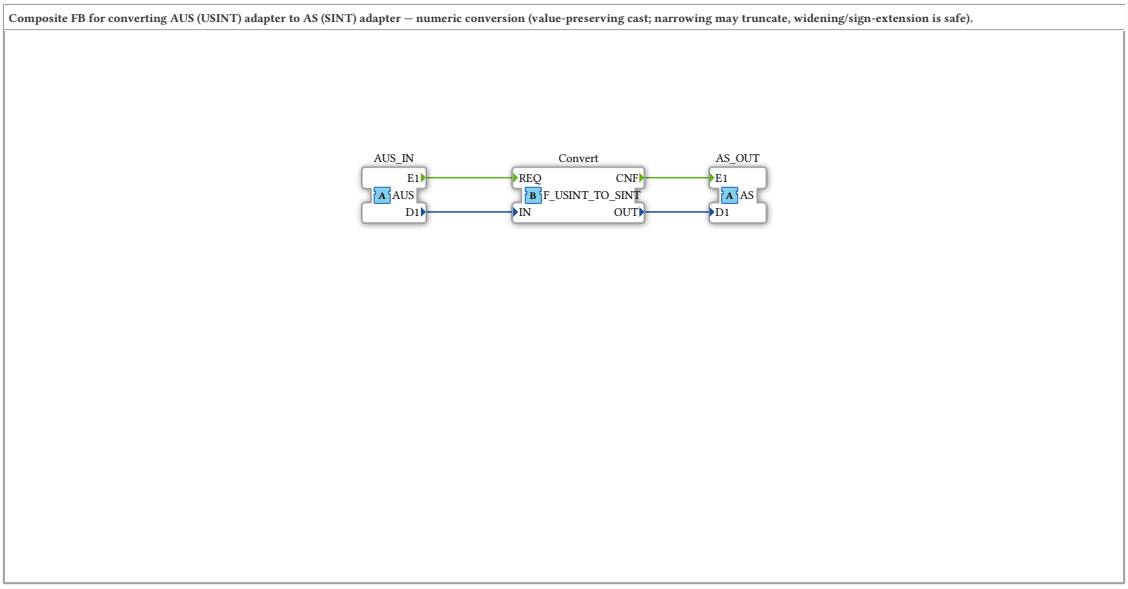
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AUS_TO_AS



USINT Adapter Input – *AUS* — *AS* – SINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block **AUS_TO_AS** is a composite function block (FB) for converting a unidirectional OFF adapter (based on the USINT data type) into a unidirectional AS adapter (based on the SINT data type). It serves as a standardized interface between components that use different unsigned/signed integer types.

Interface Structure

The FB has no direct event or data inputs/outputs, but only two adapter interfaces:

Event Inputs

No directly visible event inputs. Event control is handled via the socket adapter `AUS_IN`.

Event Outputs

No directly visible event outputs. Event feedback is provided via the plug adapter `AS_OUT`.

Data Inputs

No directly visible data inputs. The data is provided via the socket adapter `AUS_IN`.

Data Outputs

No directly visible data outputs. The data is output via the plug adapter `AS_OUT`.

Adapters

Name	Type	Direction	Comment
`AUS_IN`	`adapter::types::unidirectional::AUS` (Socket)	Input	USINT Adapter Input
`AS_OUT`	`adapter::types::unidirectional::AS` (Plug)	Output	SINT Adapter Output

The interface of the adapters themselves contains:

- `AUS_IN.E1` (Event), `AUS_IN.D1` (USINT)
- `AS_OUT.E1` (Event), `AS_OUT.D1` (SINT)

Functionality

The function block is implemented as a composite and contains an internal function block `Convert` of type `iec61131::conversion::F_USINT_TO_SINT`, which performs the actual value conversion.

1. **Data Acquisition:** An incoming USINT value is provided via socket `AUS_IN.D1`.
2. **Triggering:** An event at output `AUS_IN.E1` triggers input `REQ` of the conversion block.

3. **Conversion:** The internal function block `F_USINT_TO_SINT` converts the USINT value into a corresponding SINT value (value range 0..255 ⟶ −128..127; if this range is exceeded, a range limitation according to IEC 61131 applies).
4. **Feedback:** After the conversion is complete, the internal function block signals an event to plug `AS_OUT.E1` via its output `CNF`.
5. **Data Transfer:** The converted SINT value is output to the connected plug via `AS_OUT.D1`.

The connections in the FBNetwork are:

- Event: `AUS_IN.E1` ⟶ `Convert.REQ`
- Event: `Convert.CNF` ⟶ `AS_OUT.E1`
- Data: `AUS_IN.D1` ⟶ `Convert.IN`
- Data: `Convert.OUT` ⟶ `AS_OUT.D1`

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-Based:** The module functions as a pure adapter converter and can be seamlessly integrated into existing adapter-oriented systems.
- Event: `AUS_IN.D1` ⟶ `Convert.IN`
- Data: `Convert.OUT` ⟶ `AS_OUT.D1`
- **Adapter-Based:** The module functions as a pure adapter converter and can be seamlessly integrated into existing adapter-oriented systems.
- Event: `AUS_IN.E1` ⟶ `Convert.IN`
- Event: `Convert.OUT` ⟶ `AS_OUT.D1`
- **Adapter-Based:** - **Composite Structure:** Conversion is performed by an internal, standardized IEC 61131 converter, ensuring reliable and certified conversion.
- **Unidirectional Interface:** Data and event flows are strictly unidirectional (from input to output).
- **No Explicit State Machine:** Control is purely event-driven via adapter events. There are no internal states.

State Overview

The function block (FB) has no visible state machine. It behaves like a transparent gateway that triggers the conversion upon each input event and provides the result at the output.

Application Scenarios

- **Integration of components** with different data types, e.g., when a sensor delivers USINT data, but a subsequent controller expects SINT values.
- **Adapter Conversion** in modular control systems based on unidirectional adapters (e.g., according to IEC 61499).
- **Safety-critical systems** where precise type conversion (without overflow) is essential – the internal function block `F_USINT_TO_SINT` is specification-compliant.

Comparison with similar function blocks

- **Simple converters like `F_USINT_TO_SINT`:** Only convert scalar values but do not use adapters. `AUS_TO_AS` encapsulates this conversion in an adapter-based function block.
- **`INT_TO_DINT` adapter:** Other type conversions (e.g., INT to DINT) follow the same pattern but differ in bit width and sign handling.
- **Bidirectional adapters:** Unlike bidirectional variants, this function block is designed for only one data flow, which reduces complexity.

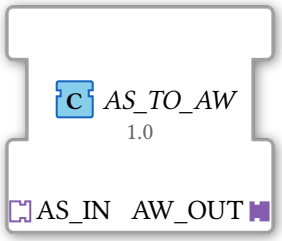
Conclusion

The **AUS_TO_AS** function block offers a clean, adapter-based solution for converting USINT to SINT values. Thanks to its composite structure and the use of a standardized IEC 61131 converter, it is easy to maintain, reliable, and readily integrated into industrial control networks. It is particularly suitable for systems that rely on unidirectional adapters and require explicit type conversion.

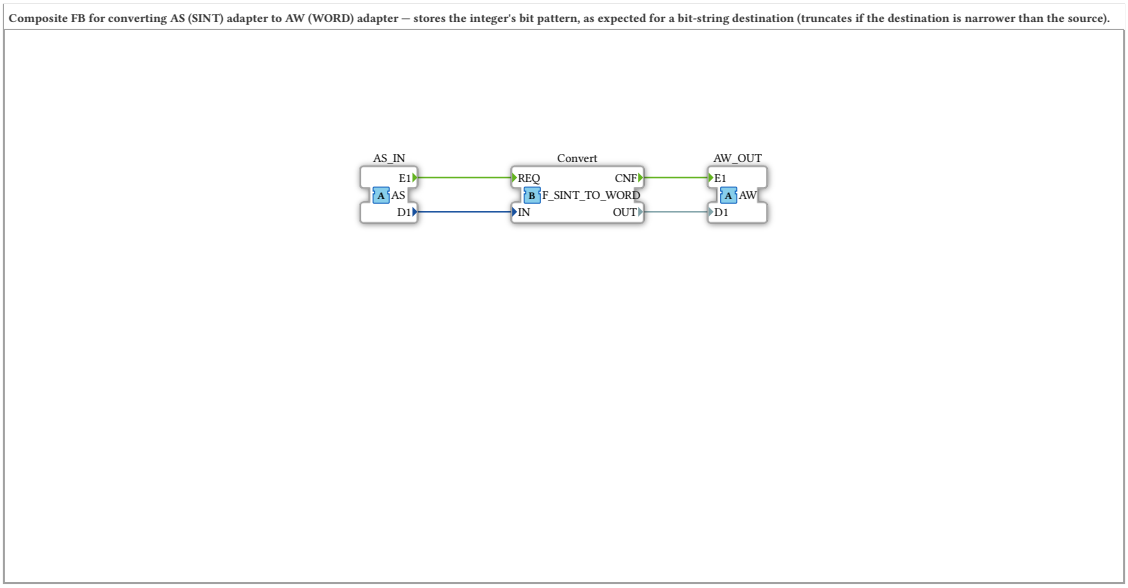
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AS_TO_AW



SINT Adapter Input – *AS* *AW* – WORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AS_TO_AW** is a composite block that converts a unidirectional **SINT** adapter (AS) into a unidirectional **WORD** adapter (AW). It serves as a bridge between different adapter types and enables the further processing of SINT signals as WORD data in a 4diac environment.

Interface Structure

Event Inputs

No direct event inputs.

The block has a **socket** of type `AS`, which indirectly provides an event input:

- **AS_IN.E1** – Event input via the socket; an incoming event triggers the conversion.

Event Outputs

No direct event outputs.

The function block has a **plug** of type `AW`, which indirectly provides an event output:

- **AW_OUT.E1** – Event output via the plug; an event is sent after successful conversion.

Data Inputs

- **AS_IN.D1** (`SINT`) – Input data value of the AS socket; the SINT value to be converted.

Data Outputs

- **AW_OUT.D1** (`WORD`) – Output data value of the AW plug; the converted WORD value.

Adapter

Type	Name	Direction	Comment
`AS` (unidirectional)	**AS_IN**	Socket (input)	SINT Adapter input
`AW` (unidirectional)	**AW_OUT**	Plug (output)	WORD Adapter output

Functionality

The function block contains an internal instance of the conversion function block `iec61131::conversion::F_SINT_TO_WORD`.

Processing is event-driven:

1. An event at **AS_IN.E1** (socket) triggers the internal function block via its `REQ` input.
2. Simultaneously, the value of **AS_IN.D1** (`SINT`) is forwarded to the `IN` input of the converter.
3. The converter `F_SINT_TO_WORD` performs the type conversion and signals its completion via its `CNF` output.
4. This event is passed to **AW_OUT.E1** (Plug), and the converted `WORD` value is available at **AW_OUT.D1**.

Thus, a reliable, one-time conversion is performed upon each event arrival.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite FB**: The function block utilizes 4diac's composite functionality to integrate an existing conversion function block (`F_SINT_TO_WORD`) into an adapter package.
- **Unidirectional Adapters**: Both the input and output adapters are unidirectional, meaning only one data and event direction is supported.
- **IEC 61131-3 Conversion**: The actual type conversion is performed according to the IEC 61131-3 standard, which facilitates compatibility with other systems.
- **Copyright Notice**: The function block includes a copyright notice (EPL 2.0) that governs its use and distribution.

State Overview

Since the function block consists of an internal FB with REQ/CNF handshake, the process can be described as follows:

| State | Description | ** |---|---|---| | **Idle** | Waiting for an event at AS_IN.E1 | | **Busy** | Conversion in progress (internal FB active) | | **Done** | Conversion complete, event sent to AW_OUT.E1 |

The function block is not stateful in the sense of a state machine – the internal FB implements the state logic.

Application Scenarios

- **Signal Conversion**: A sensor delivers data as a SINT (e.g., an 8-bit value), but the downstream system expects a WORD (16-bit). The AS_TO_AW function block handles the conversion.
- **Adapter Integration**: In a heterogeneous control system, different adapter types must be connected. The function block acts as a "translator" between AS and AW interfaces.
- **Test and Simulation Environments**: For bridging adapter incompatibilities during development or when testing function blocks.

Comparison with Similar Blocks

Block	Function	Difference
AS_TO_AW	SINT → WORD	Specific conversion for unidirectional adapters
General Converters (e.g., `F_SINT_TO_WORD`)	Pure data conversion without adapter encapsulation	AS_TO_AW provides a ready-made adapter interface and can be used directly in adapter-based networks

Other Adapter Converters (e.g., `BOOL_TO_BYTE`)	Other data types	AS_TO_AW specializes in SINT ↔ WORD

Conclusion

The **AS_TO_AW** is a useful, compact conversion module that seamlessly integrates the conversion from a SINT adapter (AS) to a WORD adapter (AW) into the 4diac IDE. By using the standardized IEC 61131-3 converter and unidirectional adapter technology, it enables flexible and reliable signal conversion in automation projects.

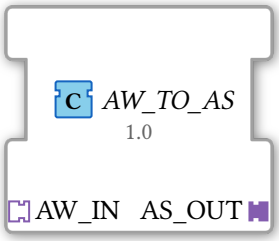
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

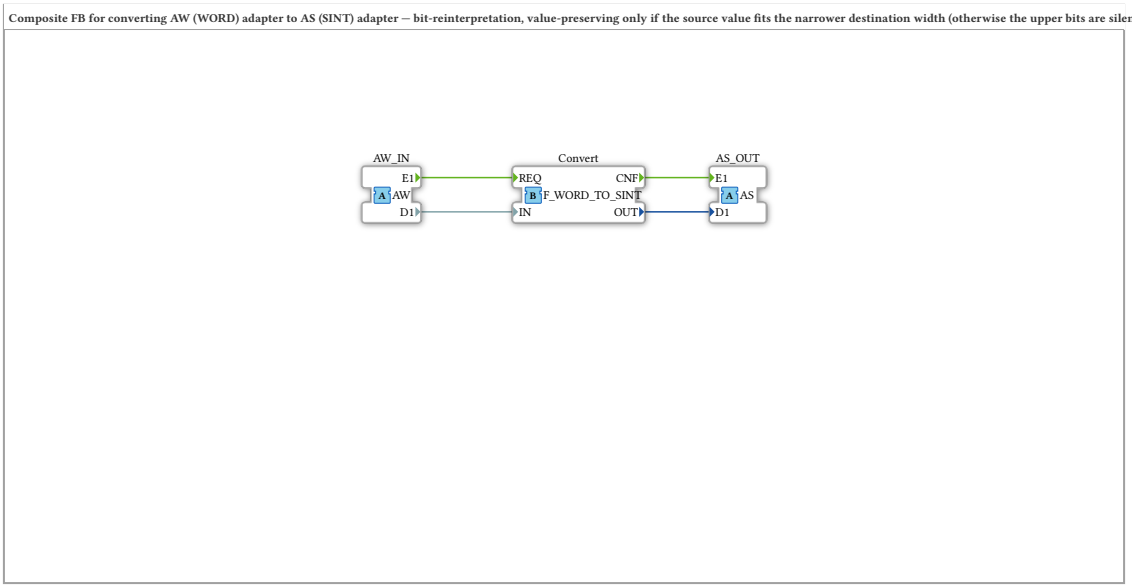
]

AW_TO_AS

WORD Adapter Input – *AW*



AS – SINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `AW_TO_AS` is a composite block that converts a WORD-based adapter (AW) into a SINT-based adapter (AS). It encapsulates the necessary type conversion, thus enabling the easy integration of different adapter interfaces in automation technology.

Interface Structure

Event Inputs

- `AW_IN.E1` – Starts the conversion of the incoming data value.

Event Outputs

- `AS_OUT.E1` – Signals that the converted value is available at the data output.

Data Inputs

- `AW_IN.D1` (WORD) – The 16-bit value to be converted to a SINT value.

Data Outputs

- `AS_OUT.D1` (SINT) – The converted 8-bit value (signed, range –128 … 127).

Adapter

- **Socket (Input)**: `AW_IN` of type `adapter::types::unidirectional::AW`
- **Plug (Output)**: `AS_OUT` of type `adapter::types::unidirectional::AS`

Functionality

The module is implemented as a composite and internally uses the IEC 61131 function block `F_WORD_TO_SINT`. An incoming event at `AW_IN.E1` triggers the conversion: The data value from `AW_IN.D1` is passed to the input `IN` of the internal converter. After successful conversion, this generates an acknowledgment event at `CNF`, which is forwarded to the output adapter `AS_OUT.E1`. Simultaneously, the converted SINT value is output to `AS_OUT.D1`.

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- The conversion follows the IEC 61131 function `WORD_TO_SINT`. An overflow occurs if the incoming WORD value is outside the valid SINT range (-128 … 127) – this must be taken into account in the application.
- The function block operates purely event-driven: Conversion only occurs when an event is present at the input.
- The adapters are unidirectional; conversion back from SINT to WORD is not supported.

State Overview

Since this is a simple composite function block (FB) without its own state machine, there is no explicit state machine. The FB waits for an event, performs the conversion, and outputs the result synchronously. A state table is not required.

Application Scenarios

- Adapting sensor data supplied as WORD (e.g., via an analog input) to a system that processes SINT values.
- Converting fieldbus data formats when coupling different automation components.
- Use in heterogeneous control environments where adapters use different data types.

Comparison with Similar Function Blocks

Analogously structured converters such as `AW_TO_BY` (WORD ⟶ BYTE) or `AINT_TO_AUDINT` (INT ⟶ UDINT) follow the same principle. The `AW_TO_AS` function block stands out due to its specific adapter combination, which enables direct integration into adapter-based architectures. Unlike simple data-level type conversions, this block utilizes an event-driven adapter interface, thus increasing reusability and interchangeability.

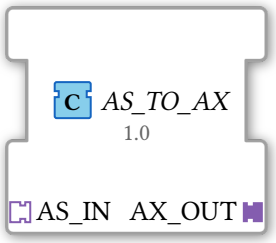
Conclusion

The composite function block `AW_TO_AS` offers a compact and clear solution for converting a WORD adapter to a SINT adapter. Encapsulating the conversion in a separate function block improves the readability and maintainability of the application. This function block is particularly suitable for environments where different data types are handled at the adapter level.

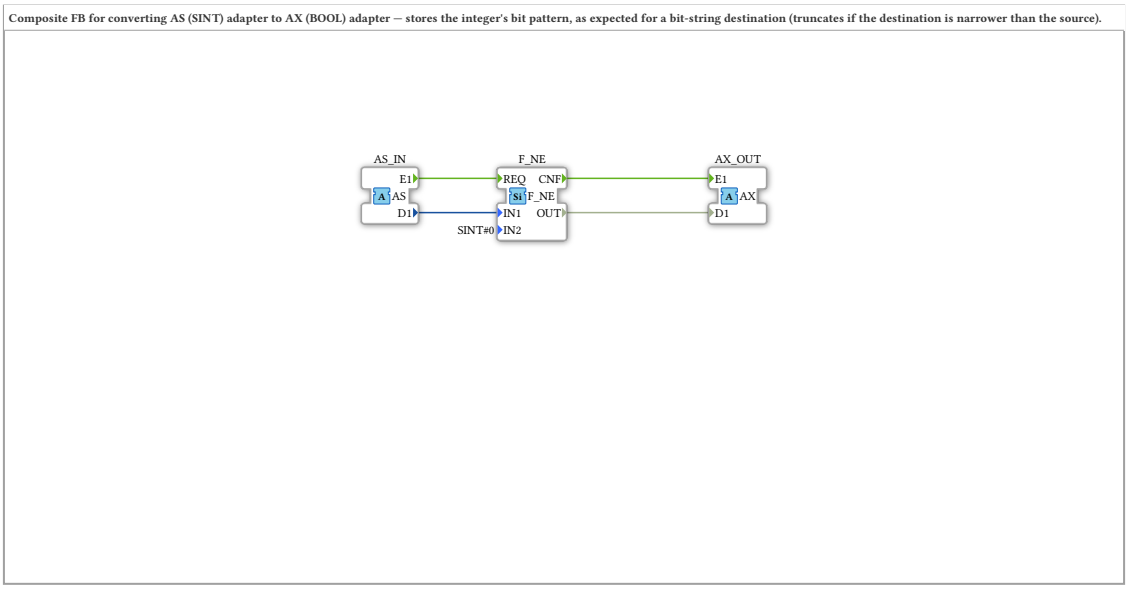
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AS_TO_AX



SINT Adapter Input – *AS* *AX* – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AS_TO_AX** function block is a composite function block that converts a SINT adapter (AS) into a BOOL adapter (AX). It checks the incoming SINT value for inequality to zero and outputs the result as a BOOL signal. The function block is suitable for simple signal conversion in an IEC 61499 application.

Interface Structure

Event Inputs

Name	Type	Description
AS_IN.E1	Event	Receives the trigger signal from the connected AS adapter for data processing.

Event Outputs

Name	Type	Description
AX_OUT.E1	Event	Sent after successful conversion; indicates that the BOOL value is present at the AX adapter.

Data Inputs

Name	Type	Description
AS_IN.D1	SINT	Receives the SINT value, which is checked for inequality to zero.

Data Outputs

Name	Type	Description
AX_OUT.D1	BOOL	Outputs the result of the comparison: `TRUE` if the input value is not zero, otherwise `FALSE`.

Adapter

Name	Type	Direction	Description
AS_IN	adapter::types::unidirectional::AS	Socket (Input)	SINT adapter for the input value and its associated event.
AX_OUT	adapter::types::unidirectional::AX	Plug (Output)	BOOL adapter for the output value and its associated event.

Functionality

The function block internally uses the function block `iec61131::comparison::F_NE` (inequality check).

- When an event is received at input `AS_IN.E1`, processing of `F_NE` (input `REQ`) is started.
- `F_NE` compares the SINT value received via `AS_IN.D1` with the constant parameter `IN2` (fixed to `SINT#0`).
- The result of the comparison (`OUT`) – `TRUE` if inequality, `FALSE` if equality – is forwarded to the data output `AX_OUT.D1`.
- After the calculation is complete, the event `AX_OUT.E1` is triggered via `F_NE.CNF`, signaling the new BOOL value at the AX adapter.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Type Conversion**: The function block implicitly converts a SINT adapter to a BOOL adapter without requiring the user to write direct type conversion logic.
- **Use of Standard Function Blocks**: Internally, the existing IEC 61131 function block `F_NE` is used, increasing reusability and testability.
- **Fixed Comparison Value**: The parameter `IN2` is preset to `SINT#0` – a change is only possible by modifying the composite function block.

State Overview

Since this is a composite function block without its own state machine, its behavior is limited to the data and event flow according to the internal interconnection. The states result from the processing function block `F_NE`:

- **Idle**: No event is pending.
- **Processing**: An event is being processed (comparison in progress).
- **Completed**: The comparison is complete, and the result is available at the output.

Application Scenarios

- **Threshold Check**: Conversion of a SINT counter value into a BOOL signal that indicates whether the value deviates from zero (e.g., "Motor running").
- **Adapter Bridge**: Connection of a component with an AS interface to a component with an AX interface without manual conversion.
- **Simple Status Indication**: Signal transmission indicating whether a data record exists (non-zero) or not.

Comparison with Similar Function Blocks

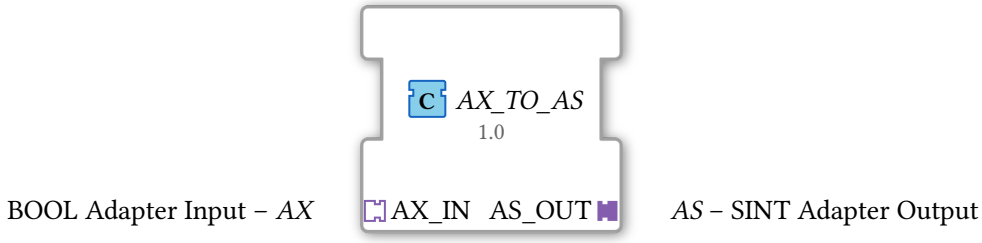
- **Direct Conversion Function Blocks** (e.g., `SINT_TO_BOOL`): Convert the data type without adapter logic, but do not provide event-driven behavior via adapters.
- **Custom Composite Solution**: Offers flexibility, but requires more effort; `AS_TO_AX` cleanly encapsulates the conversion and is reusable.

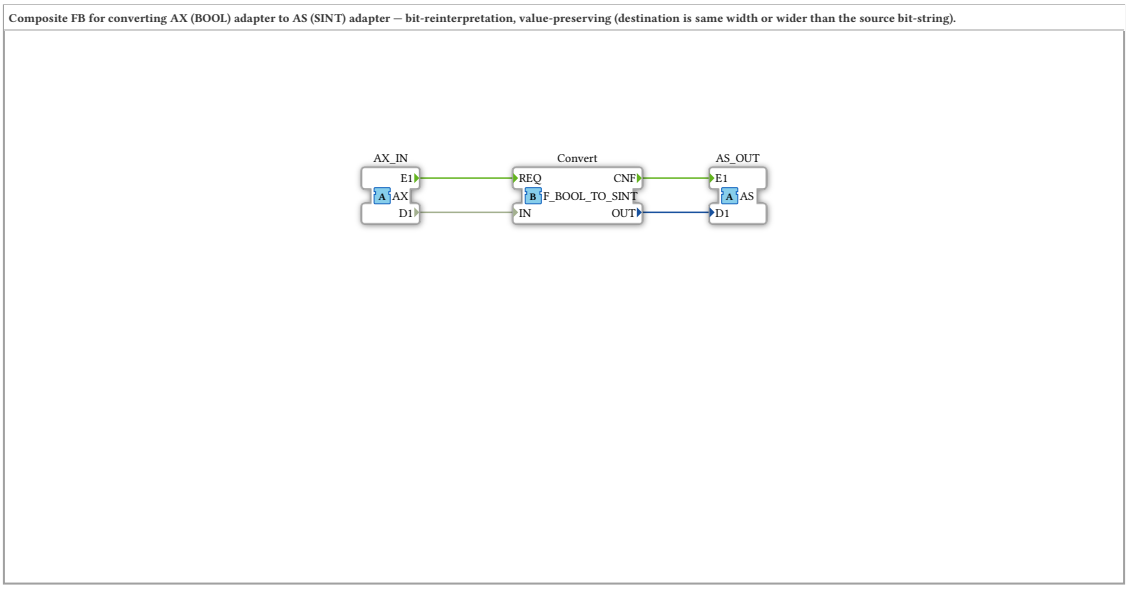
- **Adapter Wrappers**: Other function blocks may use different comparison operations (e.g., `F_EQ`), specifically checking for inequality to zero.

Conclusion

The **AS_TO_AX** function block provides a compact and reliable solution for converting a SINT adapter signal to a BOOL adapter signal. By utilizing the IEC 61131 standard function block `F_NE`, its behavior is deterministic and easily traceable. This function block is particularly suitable for applications requiring simple zero checks across adapter boundaries.

AX_TO_AS





• – ○ • – ○ • – ○ •

Introduction

The function block `AX_TO_AS` is a composite function block that converts a BOOL adapter (type AX) into a SINT adapter (type AS). It encapsulates the conversion of a binary signal into a signed 8-bit value and provides this via a standardized adapter interface.

Interface Structure

Event Inputs

No separate event inputs; event control is handled via the AX_IN adapter (see Adapters).

Event Outputs

No separate event outputs; event output is handled via the AS_OUT adapter (see Adapters).

Data Inputs

No separate data inputs; Data is transferred via the AX_IN adapter (see Adapters).

Data Outputs

No separate data outputs; data is output via the AS_OUT adapter (see Adapters).

Adapters

- **AX_IN** (Socket):

Type `adapter::types::unidirectional::AX` (BOOL).
Provides the binary input signal via the event port `E1` and the data port `D1`.

- **AS_OUT** (Plug):

Type `adapter::types::unidirectional::AS` (SINT).
Outputs the converted SINT signal via the event port `E1` and the data port `D1`.

Functionality

The function block (FB) has an internal network consisting of an instance of the standard function block `F_BOOL_TO_SINT`.

1. An incoming event at the AX_IN adapter (`E1`) triggers the conversion.
2. The data value of `AX_IN.D1` (BOOL) is passed to the input `IN` of the conversion function block.
3. The conversion function block converts the BOOL value into a SINT value (`TRUE` ⟶ 1, `FALSE` ⟶ 0).
4. The result is provided at output `OUT` and forwarded to the AS_OUT adapter (`D1`).
5. An acknowledgment event is output via `AS_OUT.E1`.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Composite Structure**: The function block (FB) is implemented as a composition, making the internal logic transparent and reusable.
- **Standard Conversion**: Uses the standardized FB `iec61131::conversion::F_BOOL_TO_SINT` from the IEC 61131 library.
- **Adapter Interface**: Input and output are exclusively via unidirectional adapters (sockets/plugs), enabling modular connections in adapter-based architectures.

State Overview

The FB does not have any explicitly modeled states. The behavior is purely event-driven: Every event at the input adapter triggers a conversion, after which an output event is sent.

Application Scenarios

- **Binary Sensor Connection**: A binary sensor (e.g., limit switch) is read via an AX adapter and passed to a controller as a SINT value.
- **Protocol Adaptation**: In systems requiring SINT-based adapters, existing BOOL adapters can be connected without additional coding.
- **Test Environments**: Easy switching between BOOL and SINT signals to verify adapter communication paths.

Comparison with Similar Components

- **Direct Conversions (`F_BOOL_TO_SINT`)**: These operate at the data level without adapters. The `AX_TO_AS` encapsulates the conversion in an adapter-based interface and is therefore easier to integrate into adapter networks.
- **Other adapter converters (e.g., `AS_TO_AX`)**: These represent the reverse direction; `AX_TO_AS` complements them as a counterpart.

Conclusion

The `AX_TO_AS` is a simple yet useful composite function block for converting BOOL adapter signals to SINT adapter signals. By using a standardized conversion block and adapter technology, it can be flexibly integrated into existing 4diac projects and contributes to the clean separation of signal types.

🌐 Related topic subpages on ms-muc-docs.de

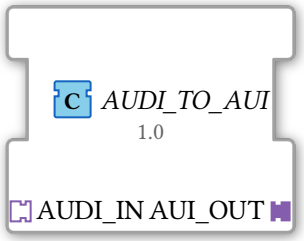
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Numeric Index (unidirectional)

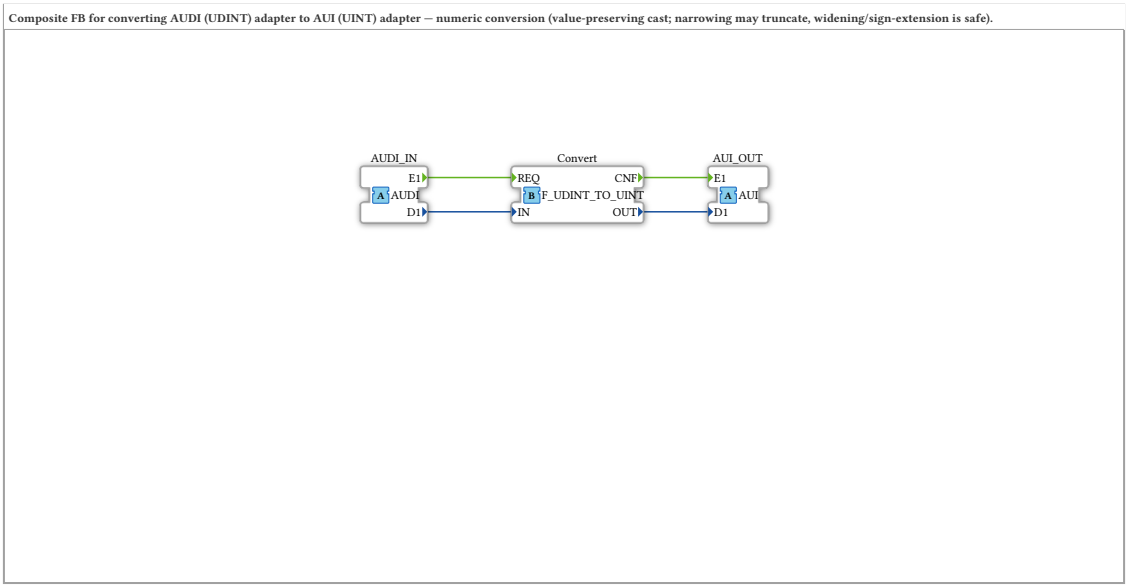
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUDI_TO_AUI

UDINT Adapter Input – *AUDI*



AUI – UINT Adapter Output



No image available.

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_TO_AUI` is a composite block that converts an AUDI adapter (data type UDINT) into an AUI adapter (data type UINT). It serves as a simple interface adapter between components that use different integer data types for their unidirectional adapters. The block encapsulates the conversion and enables seamless integration into IEC 61499 networks.

Interface Structure

Event Inputs

The function block does not have a direct event input at the top level. Events are received via socket `AUDI_IN`:

- **E1** – Trigger for the conversion of a new data value.

Event Outputs

The FB does not have a direct event output at the top level. Events are sent via the plug `AUI_OUT`:

- **E1** – Confirmation that the converted value is present at the output.

Data Inputs

The FB does not have a direct data input at the top level. Data is received via the socket `AUDI_IN`:

- **D1** (`UDINT`) – The input value to be converted.

Data Outputs

The FB does not have a direct data output at the top level. Data is output via the plug `AUI_OUT`:

- **D1** (`UINT`) – The converted output value.

Adapters

- **Socket `AUDI_IN`** – Unidirectional input adapter of type `adapter::types::unidirectional::AUDI`. It returns an event `E1` and a data value `D1` (type `UDINT`).
- **Plug `AUI_OUT`** – Unidirectional output adapter of type `adapter::types::unidirectional::AUI`. It expects an event `E1` and outputs a data value `D1` (type `UINT`).

Functionality

The module is implemented as a composite and internally contains the function block `iec61131::conversion::F_UDINT_TO_UINT`. The sequence is event-driven:

1. An event at socket `AUDI_IN.E1` triggers the input `REQ` of the internal conversion module.
2. Simultaneously, the data value `AUDI_IN.D1` is transferred to the input `IN` of the converter.
3. The converter performs the conversion from `UDINT` to `UINT`.
4. Upon completion, the converter sends an event at its output `CNF`, which is forwarded to plug `AUI_OUT.E1`.
5. The converted value at the converter's output `OUT` is placed on the data output `AUI_OUT.D1`.

The entire conversion takes place within a single event pass (synchronously).

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Architecture:** The function block is designed as a composite, meaning its functionality is entirely determined by the internal network. This facilitates reuse and adaptation.
- **Standard Conversion Block:** The actual type conversion uses the IEC 61131 block `F_UDINT_TO_UINT`, which provides a platform-independent and tested implementation.
- **License:** The block is licensed under the Eclipse Public License 2.0 (EPL-2.0).
- **Version:** 1.0, released on February 17, 2026.

State Overview

The FB (Function Block) has no states of its own. Since the composite network consists of only a single functional block, which itself operates without state, `AUDI_TO_AUI` also behaves without state. Each event pulse immediately triggers a one-time conversion.

Application Scenarios

- **Connecting Sensors/Actuators:** A sensor delivers values as `UDINT` via an AUDI adapter, while the subsequent processing chain expects `UINT` via AUI.
- **System Integration:** Different data types often occur in heterogeneous automation systems. The block serves as an adapter to connect components from different manufacturers or libraries.
- **Prototype Development:** Simple and quick adaptation of interfaces without manual conversion logic.

Comparison with Similar Function Blocks

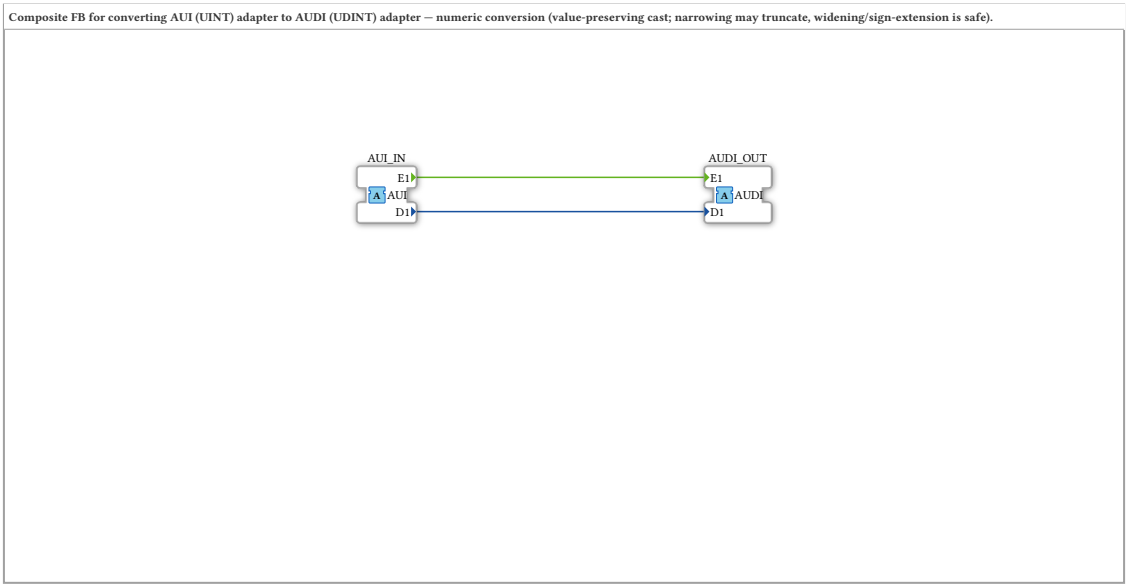
- **`AUDI_TO_XXX` Function Blocks:** Analogous function blocks exist for other target types (e.g., `AUDI_TO_DINT`, `AUDI_TO_SINT`). `AUDI_TO_AUI` is specific for conversion to `UINT`.
- **Direct Adapter Coupling:** Without a conversion block, type conversion would have to be performed in the application itself or via a generic function block. `AUDI_TO_AUI` offers an encapsulated, reusable solution.

Conclusion

The `AUDI_TO_AUI` function block provides a compact and reliable way to mediate between two unidirectional adapters with different data types. Thanks to its composite structure and the use of a standardized conversion module, it requires little maintenance and can be easily integrated into existing IEC 61499 projects.

AUI_TO_AUDI





• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_TO_AUDI** function block is a composite function block that converts an AUI adapter (UINT) into an AUDI adapter (UDINT). It serves to transfer data and events from one adapter type to another, where the underlying data types differ (UINT ⟶ UDINT). The conversion is performed by simply passing the signals through, without any additional logic.

Interface Structure

Event Inputs

No dedicated event inputs: The function block has no direct event inputs. Events are received via the connected socket adapter **AUI_IN**.

Event Outputs

No dedicated event outputs: The function block has no direct event outputs. Events are sent via the connected plug adapter **AUDI_OUT**.

Data Inputs

No dedicated data inputs: This function block has no direct data inputs. Data is received via the connected socket adapter **AUI_IN**.

Data Outputs

No dedicated data outputs: This function block has no direct data outputs. Data is sent via the connected plug adapter **AUDI_OUT**.

Adapters

Name	Type	Direction	Comment

AUI_IN	adapter::types::unidirectional::AUI	Socket (Input)	Receives UINT data and associated event
AUDI_OUT	adapter::types::unidirectional::AUDI	Plug (Output)	Sends UINT data and associated event

Functionality

The function block internally connects the interfaces of the incoming AUI adapter with those of the outgoing AUDI adapter:

- **Event Connection**: The event `E1` of the AUI adapter is directly forwarded to the event `E1` of the AUDI adapter.
- **Data Connection**: The data value `D1` of the AUI adapter (type UINT) is directly transferred to the data value `D1` of the AUDI adapter (type UDINT). This involves an implicit type conversion from UINT (16 bits) to UDINT (32 bits).

The function block does not perform any logic or conversion of its own; it simply acts as a pass-through for adapters with different data types.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Unidirectional Adapters**: Both the input and output adapters are unidirectional, meaning data and events flow only in one direction (from the socket to the plug).
- **Composite FB**: The function block is implemented as a composite function block (FB) that maps the adapter connections in the FBNetwork.
- **Implicit Type Conversion**: The conversion from UINT to UDINT occurs automatically within the data connection, based on the IEC 61499 data type rules.
- **Package Structure**: The function block belongs to the package `adapter::conversion::unidirectional`.

State Overview

The function block does not have its own states or state machines. Its behavior results solely from the forwarding connection of the adapter interfaces.

Application Scenarios

- **Adapter Adaptation**: If a system uses an AUI adapter (UINT) but expects an AUDI adapter (UDINT), this function block can be used as an intermediary converter.
- **Signal Forwarding**: Used in hierarchical systems where a unidirectional data flow from a smaller to a larger data type needs to be mapped.
- **Prototypical Integration**: Rapid coupling of components with different adapter types without additional logic.

Comparison with Similar Function Blocks

- **AUI_TO_AUDI** performs pure type conversion without further processing.
- Similar function blocks like `AUDI_TO_AUI` would map the reverse direction (UDINT ⟶ UINT), possibly with data overflow handling.
- Other adapter converters (e.g., `BOOL_TO_BYTE`) follow the same pattern but differ in the data types used and the direction.

Conclusion

The **AUI_TO_AUDI** block is a simple yet useful composite function block for converting a UINT-based AUI adapter to a UDINT-based AUDI adapter. It is particularly suitable for situations requiring unidirectional signal forwarding with implicit data type conversion, without the need for custom processing logic. Thanks to its clear structure, it can be easily integrated into existing 4diac projects.

Related topic subpages on ms-muc-docs.de

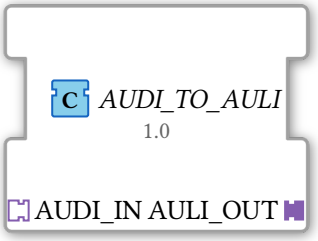
- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUDI_TO_AULI



UDINT Adapter Input – *AUDI* *AULI* – ULINT Adapter Output

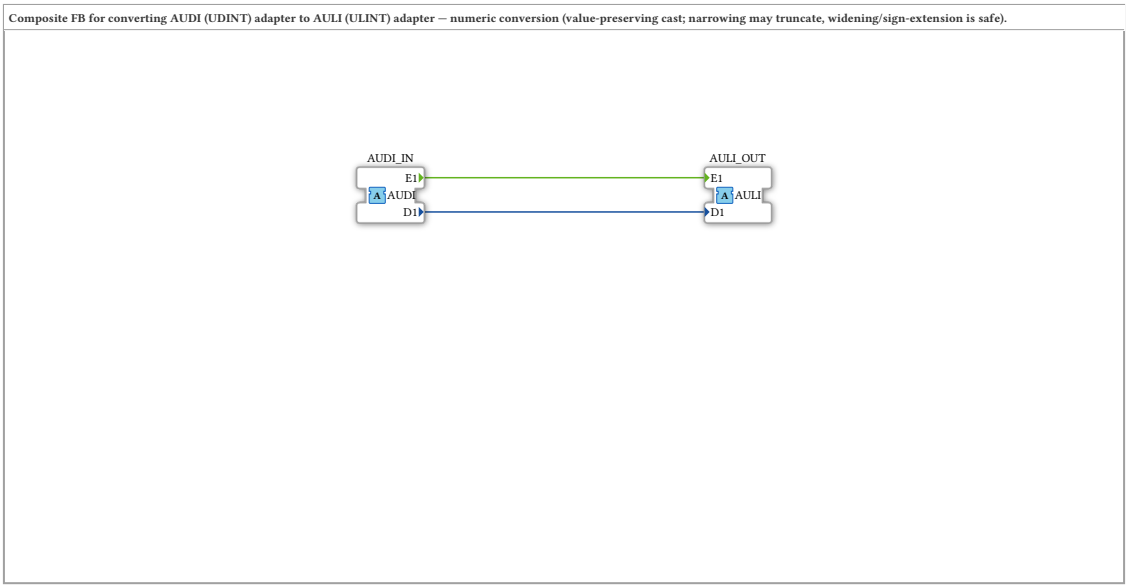


Image not available

• – ○ • – ○ • – ○ •

Introduction

This composite function block converts an AUDI adapter (UDINT) to an AULI adapter (ULINT). It serves as an interface between components that use different adapter types for unsigned integers. The function block is licensed under the Eclipse Public License 2.0 (EPL 2.0) and is provided by HR Agrartechnik GmbH.

Interface Structure

Event Inputs

None (events are received via the AUDI_IN adapter).

Event Outputs

None (events are sent via the AULI_OUT adapter).

Data Inputs

None (data is received via the AUDI_IN adapter).

Data Outputs

None (data is sent via the AULI_OUT adapter).

Adapters

- **AUDI_IN** (Socket): Receives an AUDI adapter (type: `adapter::types::unidirectional::AUDI`). Contains one event input (E1) and one data input (D1) of type UDINT.
- **AULI_OUT** (Plug): Sends an AULI adapter (type: `adapter::types::unidirectional::AULI`). Contains one event output (E1) and one data output (D1) of type ULINT.

Functionality

The function block forwards incoming events and data from the AUDI_IN adapter directly to the AULI_OUT adapter:

- The event `E1` from `AUDI_IN` is transferred to `E1` from `AULI_OUT`.
- The data value `D1` (UDINT) from `AUDI_IN` is transferred to `D1` (ULINT) from `AULI_OUT`. The 32-bit UDINT value is implicitly converted to the 64-bit ULINT value (null extension). No additional logic is performed.

...

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite FB**: Contains no function block logic or state machine, but merely wiring between two adapters.
- **Package**: `adapter::conversion::unidirectional`
- **Version**: 1.0, created on 2026-02-17.
- **License**: EPL 2.0.

State Overview

The FB has no internal state or algorithm. It operates strictly in a flow-oriented manner and reacts immediately to every incoming event.

Application Scenarios

- **System Integration**: If one system component provides data via an AUDI adapter (UDINT), but another component expects an AULI adapter (ULINT), this FB acts as a bridge.
- **Protocol Adaptation**: Use in control systems that use different integer widths for buses or interfaces (e.g., 32-bit to 64-bit).

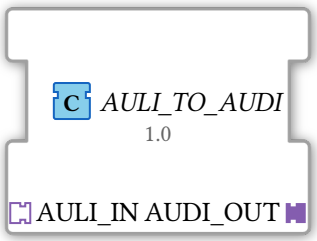
Comparison with Similar Function Blocks

- **Simple Data Type Converters** (e.g., `UDINT_TO_ULINT`): These only convert data without adapters. This function block operates at the adapter level and also forwards events.
- **Adapter Transceivers**: Other function blocks could perform more complex conversions (e.g., scaling, offsetting). This function block is minimal.

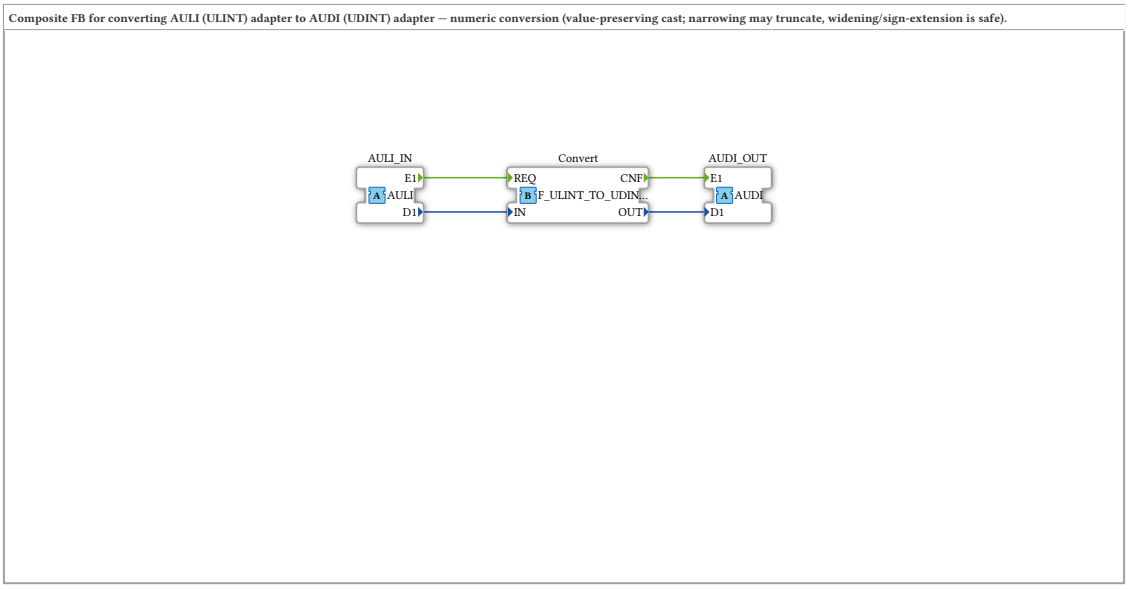
Conclusion

AUDI_TO_AULI` is a practical composite function block for lossless conversion from UDINT to ULINT at the adapter level. It is simple, easy to understand, and ideal for clean interface separation in modular systems.

AULI_TO_AUDI



ULINT Adapter Input – *AULI* — *AUDI* – UDINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block **AULI_TO_AUDI** is a composite block that converts an AULI adapter (ULINT data) to an AUDI adapter (UDINT data). It serves as a unidirectional bridge between two different adapter types and enables the seamless integration of components based on different integer widths.

Interface Structure

Event Inputs

The function block does not have separate event inputs. The required event is received via the socket adapter:

- **E1** (in socket `AULI_IN`): Starts the conversion of a ULINT value.

Event Outputs

There are no direct event outputs. The event is output via the plug adapter:

- **E1** (in plug `AUDI_OUT`): Signals the completion of the conversion and the provision of the UDINT value.

Data Inputs

Data is provided exclusively via the socket adapter:

- **D1** (in socket `AULI_IN`): UDINT input value (64-bit unsigned integer).

Data Outputs

Data is output via the plug adapter:

- **D1** (in plug `AUDI_OUT`): UDINT output value (32-bit unsigned integer).

Adapter

- **Socket AULI_IN**

Type: `adapter::types::unidirectional::AULI`
Receives the ULINT value to be converted and the associated event.

- **Plug AUDI_OUT**

Type: `adapter::types::unidirectional::AUDI`
Provides the converted ULINT value and the completion event.

Functionality

The function block contains an internal network with a single conversion block, `F_ULINT_TO_UDINT` (from the IEC 61131 library).

1. An incoming event **E1** from socket `AULI_IN` triggers the conversion block via its `REQ` event.
2. Simultaneously, the ULINT value from `AULI_IN.D1` is forwarded to the input `IN` of the converter.
3. The converter transforms the 64-bit value into a 32-bit value and outputs the result to its output `OUT`.
4. After successful conversion, the converter sends an acknowledgment event (`CNF`), which is forwarded as **E1** to the plug `AUDI_OUT`.
5. The converted ULINT value is output to `AUDI_OUT.D1`.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite FB**: The function block itself does not have its own logic but uses a standardized IEC 61131 converter.
- **Adapter Conversion**: Enables the connection of components based on different adapter definitions without requiring manual conversion by the user.
- **Unidirectional Data Flow**: Conversion is only performed in one direction (ULINT → UDINT); reverse conversion is not supported.
- **No Data Range Check**: Since UDINT only comprises 32 bits, higher-order bits are lost for ULINT values > $2^{32}-1$.

State Overview

The Composite FB does not have its own state machine. Its behavior is entirely determined by the internal conversion block `F_ULINT_TO_UDINT`, which typically implements a simple wait/process state. After starting (`REQ`), it performs the conversion and acknowledges it with `CNF`.

Application Scenarios

- **Adapter Bridges**: When a system provides an AULI adapter (e.g., for large counters or timestamps), but a downstream component expects an AUDI adapter (e.g., for smaller control values).
- **Protocol or Interface Adaptation**: In heterogeneous automation networks where different integer sizes are used.
- **Data Reduction**: When only the lower 32 bits of a 64-bit size are relevant (e.g., for certain sensor values).

Comparison with Similar Function Blocks

- **Direct Type Conversions** (e.g., `F_ULINT_TO_UDINT`): These operate at the data level without adapters. AULI_TO_AUDI encapsulates the conversion in an adapter-based interface, which facilitates the exchange between different adapter types.
- **Bidirectional Converters**: Function blocks for the reverse conversion (UDINT → ULINT) also exist. This function block (FB) is specialized for one direction.
- **Composite Adapter**: Other composite FBs can contain multiple conversions or additional logic (e.g., range checking) – here, the focus is on simplicity and reusability.

Conclusion

The AULI_TO_AUDI is a compact, specialized composite block that cleanly encapsulates the conversion of an AULI (ULINT) to an AUDI adapter (UDINT). Thanks to the integration of the standardized converter, it is reliable and easy to integrate into existing 4diac projects. It is ideally suited for scenarios where different adapter definitions meet and contributes to the modularity and reusability of function blocks.

🌐 Related topic subpages on ms-muc-docs.de

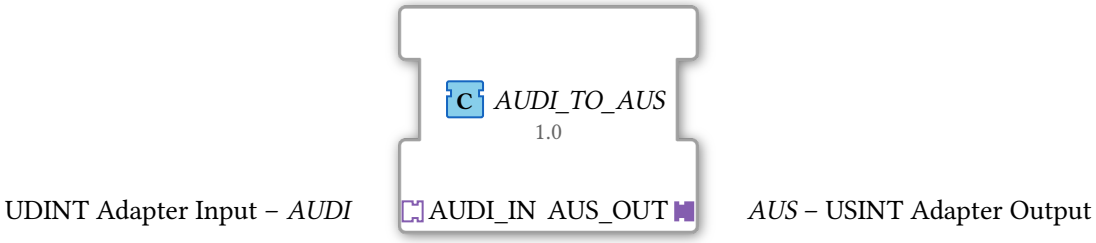
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Numeric Index (unidirectional)

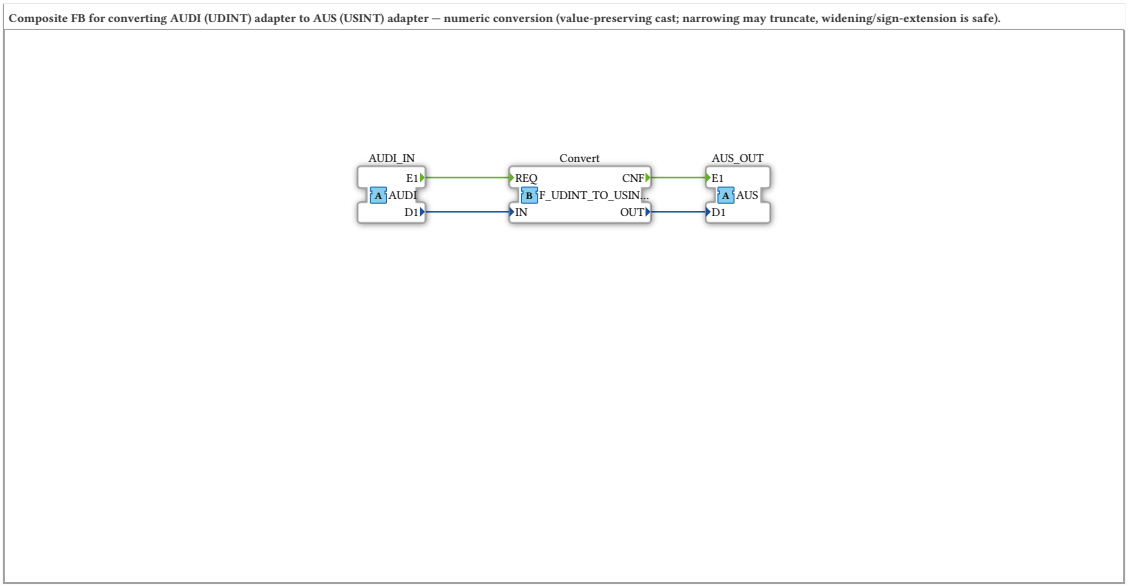
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUDI_TO_AUS



UDINT Adapter Input – *AUDI*

AUS – USINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_TO_AUS** serves as a composite block for converting a UDINT value (via the AUDI adapter) into a USINT value (via the AUS adapter). It encapsulates the conversion and allows for easy integration of components that expect different data types.

Interface Structure

Event Inputs

- **AUDI_IN.E1** (event from the AUDI adapter) – initiates the conversion of the incoming data value.

Event Outputs

- **AUS_OUT.E1** (event to the AUS adapter) – signals the completion of the conversion and the availability of the result.

Data Inputs

- **AUDI_IN.D1** (UDINT from the AUDI adapter) – the 32-bit value to be converted.

Data Outputs

- **AUS_OUT.D1** (USINT to the AUS adapter) – the converted 8-bit value.

Adapters

- **AUDI_IN** (Socket) – adapter of type `adapter::types::unidirectional::AUDI` (UDINT source).
- **AUS_OUT** (Plug) – adapter of type `adapter::types::unidirectional::AUS` (USINT sink).

Functionality

The module is implemented as a simple pass-through. In the FBNetwork, the incoming event pulse (`AUDI_IN.E1`) is forwarded directly to the internal conversion block `F_UDINT_TO_USINT`. Simultaneously, the data value `AUDI_IN.D1` is transferred to the input `IN` of the converter. After successful conversion, the converter sends the confirmation event (`CNF`) to the output adapter (`AUS_OUT.E1`) and the result (`OUT`) to the data output (`AUS_OUT.D1`). No intermediate storage or further logic takes place.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- Uses the standard conversion block `iec61131::conversion::F_UDINT_TO_USINT` from the IEC 61131 library.
- Pure event and data forwarding – no separate state machine required.
- The conversion is lossy: A UDINT (32-bit) is mapped to a USINT (8-bit), meaning higher-order bits are lost (values > 255 are truncated or handled according to the manufacturer's definition).

State Overview

The composite chip has no states of its own; it is entirely event-driven. The internal converter `F_UDINT_TO_USINT` operates according to the standard protocol (REQ/CNF). There are no additional locks or delays.

Application Scenarios

- Connecting a UDINT-based sensor to an actuator that only accepts a USINT value.
- Converting addresses or count values when only the lower-order part is needed.
- Interface adaptation in a mixed IEC 61499 environment where various data types are used.

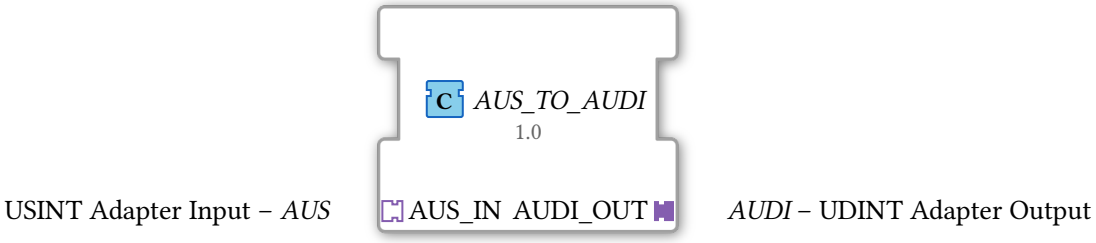
Comparison with similar function blocks

- **AUDI_TO_AUDI** or **AUS_TO_AUS**: Would not perform any conversion, but only a simple forwarding.
- **F_UDINT_TO_USINT** (direct): This function block performs the same conversion, but does not offer an adapter interface. `AUDI_TO_AUS` encapsulates the adapter types and simplifies the network structure.
- If range checking or error handling is required, a separate function block would have to be created; this function block does not perform any plausibility checks.

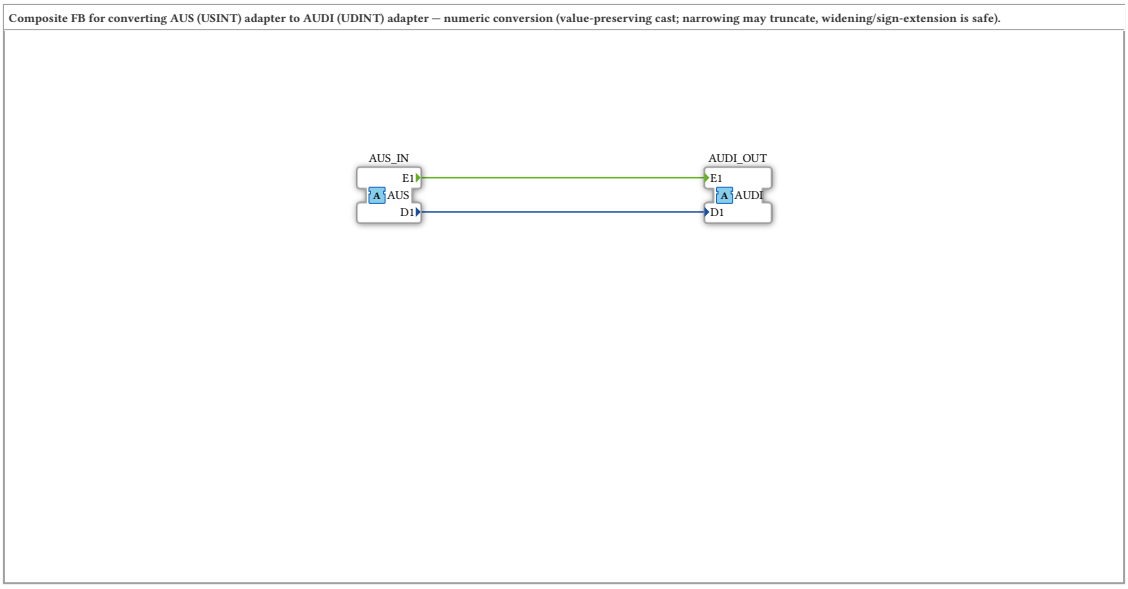
Conclusion

AUDI_TO_AUS` is a simple yet useful strip-down function block for unidirectional UDINT-to-USINT conversion via adapters. It reduces complexity in the FB network by combining adapter coupling and conversion logic into a single block. Due to the lack of error handling, it is only suitable for applications where values > 255 do not occur or where their loss can be tolerated.

AUS_TO_AUDI



USINT Adapter Input – *AUS* *AUDI* – UDINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The composite function block **AUS_TO_AUDI** converts a unidirectional adapter of type **AUS** (based on USINT values) into an adapter of type **AUDI** (based on UDINT values). It is used to translate data between two different adapter interfaces without altering the underlying data logic. The block is implemented as a pure pass-through component, forwarding events and data directly from the input to the output side.

Interface Structure

Event Inputs

The block does not have its own event inputs. Events are provided via the connected **Socket Adapter** `AUS_IN`.

Event Outputs

This function block does not have its own event outputs. Events are passed on via the connected **Plug Adapter** `AUDI_OUT`.

Data Inputs

This function block does not have its own data inputs. Data is received via the **Socket Adapter** `AUS_IN`.

Data Outputs

This function block does not have its own data outputs. Data is output via the **Plug Adapter** `AUDI_OUT`.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AUS`	`AUS_IN`	Socket (Input)	Unidirectional adapter that provides data in USINT format (8 bits) and an event.
`adapter::types::unidirectional::AUDI`	`AUDI_OUT`	Plug (Output)	Unidirectional adapter that expects data in UDINT format (32 bits) and an event.

The internal network structure connects the event `E1` and the data value `D1` of the input adapter directly to the corresponding interfaces of the output adapter.

Functionality

This functional block operates as a simple pass-through. As soon as the event `E1` occurs at socket `AUS_IN`, it is immediately forwarded to plug `AUDI_OUT` as the event `E1`. Simultaneously, the data value `D1` (USINT) of the input adapter is transferred to the data value `D1` (UDINT) of the output adapter. A type conversion from USINT to UDINT occurs implicitly through the data connection (expansion from 8 bits to 32 bits). No validation or range check is performed.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-Based Interface**: The function block does not have traditional input/output variables, but uses only the standardized adapter types `AUS` and `AUDI` from the `adapter::types::unidirectional` package.
- **Simple Passthrough**: Event and data connections are implemented directly without additional logic.
- **Type Expansion**: Conversion from USINT (0–255) to UDINT (0–4294967295) is lossless, as the entire USINT value range can be mapped to the UDINT range.
- **No State Storage**: The function block is stateless; all processing occurs in the same execution instance.

State Overview

The function block does not have an internal state machine. The functionality is limited to a direct 1:1 connection between the input and output adapters. A state description is therefore not required.

Application Scenarios

- **Adapter Bridge**: Used in systems that utilize the AUS adapter (e.g., for simple byte values) but expect an AUDI interface (e.g., for 32-bit counters or addresses).
- **Protocol Conversion**: Used within composite function blocks to mediate between different adapter versions within a device family.
- **Data Preparation**: Prepares USINT data for function blocks that accept only UDINT inputs.

Comparison with Similar Function Blocks

- **INT_TO_DINT**: A standard function block for converting INTEGER types, but with independent input/output variables. `AUS_TO_AUDI`, on the other hand, operates at the adapter level and is optimized for integration into adapter-based architectures.
- **AUS_TO_AUDI (simple)**: Alternative implementations could include additional overflow checks or filter functions; this function block is limited to simple pass-through.

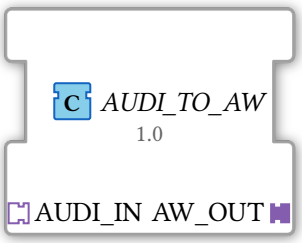
Conclusion

AUS_TO_AUDI` is a compact, specialized composite function block for lossless conversion between the unidirectional adapters OFF (USINT) and AUDI (UDINT). Thanks to the direct pass-through of events and data without additional logic, it is ideally suited as a building block in adapter-based control networks where simple type conversion at the communication layer is required.

Numeric Index (unidirectional)

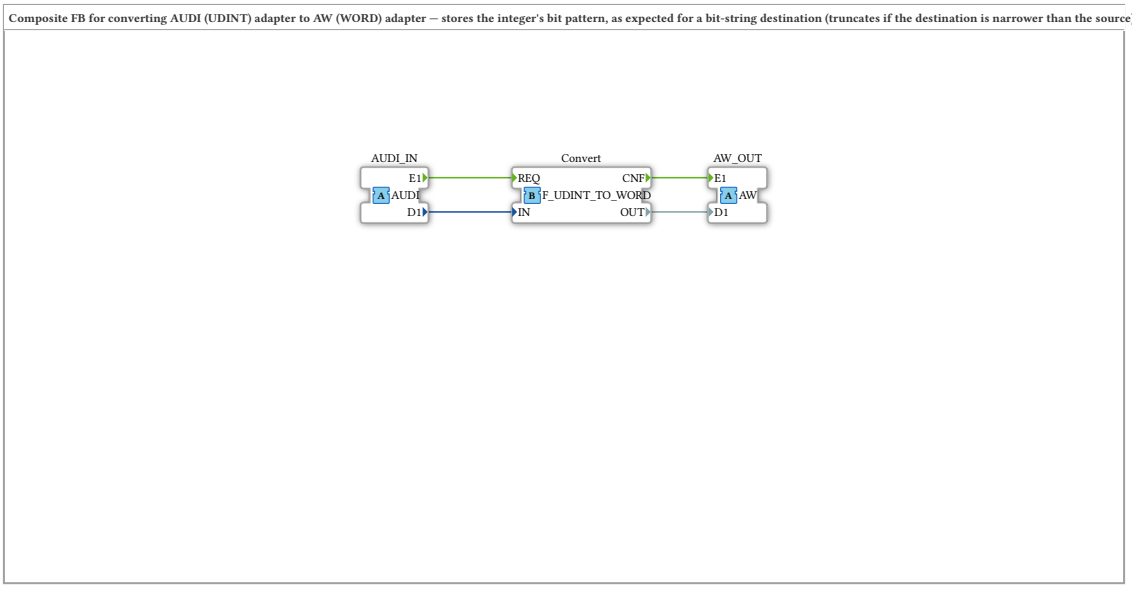
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AUDI_TO_AW



UDINT Adapter Input – *AUDI*

AW – WORD Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The **AUDI_TO_AW** function block is a composite function block for converting an **AUDI adapter** (UDINT) to an **AW adapter** (WORD). It serves as a unidirectional bridge between different data types and enables the seamless integration of components based on different adapter interfaces. The block encapsulates the conversion logic and is typically used in automation projects according to IEC 61499 that require the adaptation of data types between adapters.

Interface Structure

Event Inputs

Name	Type	Description
(via AUDI_IN.E1)	EVENT	Event that triggers the conversion (provided by the connected AUDI adapter)

Event Outputs

Name	Type	Description
(via AW_OUT.E1)	EVENT	Output event after successful conversion (provided by the AW adapter)

Data Inputs

Name	Type	Description
(via AUDI_IN.D1)	UDINT	32-bit unsigned integer to be converted (provided by the AUDI adapter)

Data Outputs

Name	Type	Description
(via AW_OUT.D1)	WORD	16-bit unsigned integer (result of the conversion, provided by the AW adapter)

Adapter

Name	Type	Direction	Description

AUDI_IN	adapter::types::unidirectional::AUDI	Socket (Input)	Receives the UDINT signal and its associated event
AW_OUT	adapter::types::unidirectional::AW	Plug (Output)	Passes on the converted WORD signal and its output event

Functionality

The function block operates as a **Composite FB** containing a single IEC 61131 conversion block `F_UDINT_TO_WORD`. Processing proceeds as follows:

1. The socket `AUDI_IN` receives an event at its event input `E1` and the corresponding data value `D1` (UDINT).
2. This event is internally forwarded to the `REQ` input of the conversion block, while the UDINT value is passed to the `IN` input.
3. The conversion block converts the UDINT value into a WORD and signals completion via its `CNF` output. 4. The `CNF` event is passed to the event output `E1` of plug `AW_OUT`, and the converted WORD is output to its data output `D1`.

The entire process is **event-driven** and takes place within a single execution step.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Unidirectional Data Flow**: The function block converts exclusively from AUDI (UDINT) to AW (WORD); reverse conversion is not supported.
- **Composite Architecture**: The internal logic is fully implemented by a single standard conversion block, making the function block easy to understand and maintain.
- **Use of IEC 61131 libraries**: The function block `F_UDINT_TO_WORD` is derived from the `iec61131::conversion` library, ensuring high portability and standards compliance.
- **No dedicated state machine**: The function block does not have an ECC (Execution Control Chart) and operates purely in a data flow-oriented manner.

State overview

The function block does **not have a state machine**. Processing is event-driven and deterministic according to the internal wiring. There are no internal states or latencies.

Application scenarios

- **Interface adaptation**: A system provides data via an AUDI adapter (UDINT), but a subsequent function block expects an AW adapter (WORD). The AUDI_TO_AW adapter converts the data without manual programming.
- **Data consolidation**: In a controller, multiple sensor data from different adapter types need to be combined. This function block serves as a conversion element.
- **Legacy Integration**: Older components based on UDINT are to be integrated into a modern WORD-based architecture.

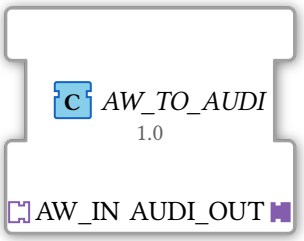
Comparison with Similar Function Blocks

- **F_WORD_TO_UDINT** (analogous, but in reverse): Converts WORD to UDINT and expects an AW input and an AUDI output.
- **General Conversion Function Blocks** such as `F_UDINT_TO_DINT` or `F_UDINT_TO_REAL` offer similar functionality but work with different data types and without adapter connections.
- **Custom Adapter Converters** can be implemented by combining several conversion function blocks and adapters, but are less encapsulated than this composite function block.

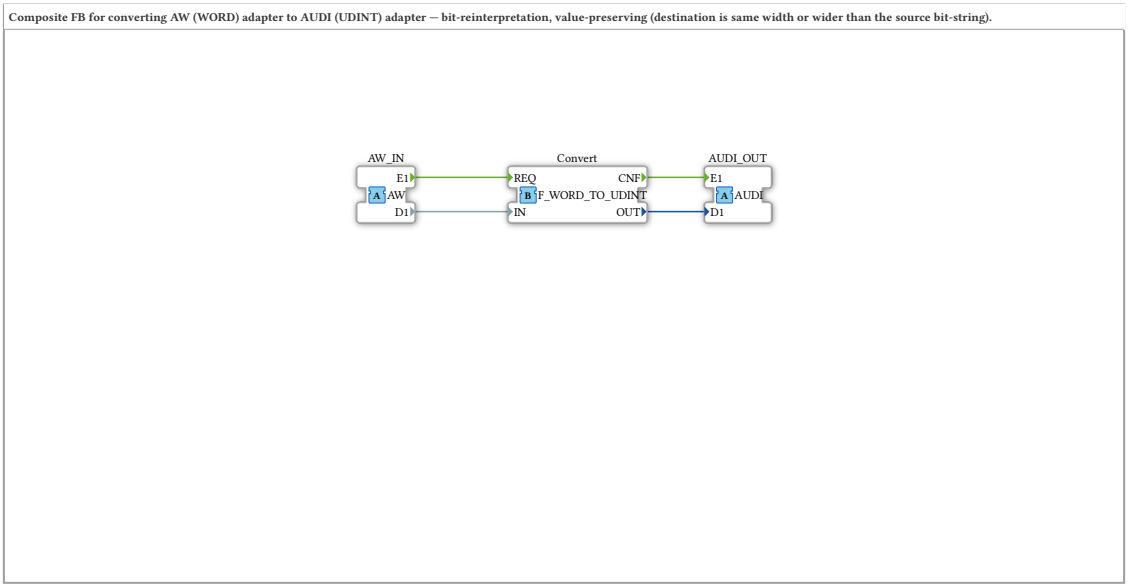
Conclusion

The **AUDI_TO_AW** is a specialized but essential component for adapter-based communication in IEC 61499 systems. By using a standardized conversion block and encapsulating it in a composite function block, it offers a clean, reusable, and testable solution for converting UDINT to WORD data within a unidirectional adapter path.

AW_TO_AUDI



WORD Adapter Input – *AW* *AUDI* – UDINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AW_TO_AUDI` is a composite block that converts a data value received via a WORD adapter (`AW`) into a UDINT value and outputs it via a UDINT adapter (`AUDI`). The conversion is performed using the internally used block `F_WORD_TO_UDINT` from the IEC 61131 library. The function block encapsulates the adapter interfaces and enables a clean separation between different data formats within a system.

Interface Structure

Event Inputs

- **AW_IN.E1** (via socket `AW_IN`): Trigger for conversion. An incoming event starts the conversion of the current WORD value.
- **Custom Event Inputs**: None – the function block receives events exclusively via the adapter `AW_IN`.

Event Outputs

- **AUDI_OUT.E1** (via plug `AUDI_OUT`): Acknowledgement after successful conversion. The outgoing event signals that a new UDINT value is available at the data output.
- **Custom Event Outputs**: None – the function block forwards events exclusively via the adapter `AUDI_OUT`.

Data Inputs

- **AW_IN.D1** (via socket `AW_IN`): The WORD value to be converted (16-bit).
- **Custom Data Inputs**: None.

Data Outputs

- **AUDI_OUT.D1** (via plug `AUDI_OUT`): The converted UDINT value (32-bit, unsigned).
- **Custom Data Outputs**: None.

Adapters

- **Socket `AW_IN`**: Expects an instance of adapter type `adapter::types::unidirectional::AW`. This adapter provides the event and data interface for the WORD input.
- **Plug `AUDI_OUT`**: Provides an instance of the adapter type `adapter::types::unidirectional::AUDI`. This adapter provides the event- and data-side interface for the UDINT output.

Functionality

1. The function block waits for an event at **AW_IN.E1**.
2. Upon receiving the event, the current value of **AW_IN.D1** (WORD) is passed to the embedded function block `F_WORD_TO_UDINT`.
3. `F_WORD_TO_UDINT` performs the conversion and places the result (UDINT) at its output `OUT`.
4. Once the conversion is complete, an event is sent via `Convert.CNF` to **AUDI_OUT.E1**, and the converted value is output via **AUDI_OUT.D1**.

Data flow and event control are synchronous: Each incoming event generates exactly one outgoing event with the updated data value.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Composite Function Block**: The function block consists entirely of an internal network and has no state logic of its own.
- **Adapter Encapsulation**: All communication takes place via the standardized unidirectional adapters `AW` and `AUDI`. This allows the function block to be integrated into any system that supports these adapter interfaces.
- **Reuse**: The internally used conversion function `F_WORD_TO_UDINT` is part of the IEC 61131 library and ensures standards-compliant type conversion.
- **No Side Effects**: The function block (FB) does not perform any state changes or have any side effects outside of the actual conversion.

State Overview

The FB `AW_TO_AUDI` does not have its own state machine. Sequence control is implemented by the internal block `F_WORD_TO_UDINT`, which exhibits simple REQ/CNF behavior:

- **Wait** (before REQ): No event at `AW_IN.E1` ⟶ no action.
- **Processing** (after REQ, before CNF): The conversion is running; the output is not updated.
- **Finished** (according to CNF): A new value is received at `AUDI_OUT.D1`, and an event is sent to `AUDI_OUT.E1`.

The overall function block reflects this behavior externally.

Application Scenarios

- **Integration of Subsystems with Different Data Types**: A sensor delivers its values as WORD (e.g., a 16-bit counter) via an AW adapter, while a control unit expects UDINT values (32-bit) via an AUDI adapter.
- **Interface Adaptation in Modular Automation Systems**: Use in a component that communicates via an adapter but requires the actual data conversion from WORD to UDINT.
- **Creation of Conversion Cascades**: Together with other modules that use, for example, DINT or REAL adapters, any data format adaptations can be created.

Comparison with Similar Modules

Module	Input Adapter	Output Adapter	Conversion	Note
`AW_TO_AUDI`	`AW` (WORD)	`AUDI` (UDINT)	WORD ⟶ UDINT	Composite, encapsulated `F_WORD_TO_UDINT`

Direct use of `F_WORD_TO_UDINT`	No adapters	No adapter	WORD → UDINT	Standard Function Block without Adapter Interfaces
`AW_TO_ADINT` (hypothetical)	`AW` (WORD)	`ADINT` (DINT)	WORD → DINT	Signed Analog Conversion

The `AW_TO_AUDI` differs from directly using the conversion function by integrating adapters, making it immediately usable for adapter-based systems.

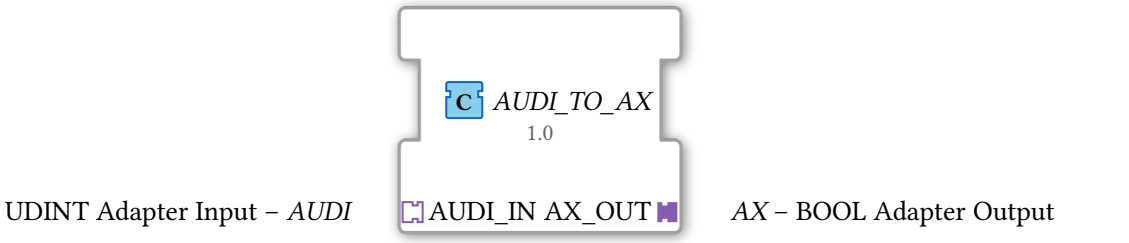
Conclusion

The `AW_TO_AUDI` function block offers a simple and clean way to convert between two unidirectional adapters of types WORD and UDINT. By encapsulating the IEC conversion function and clearly defining the interfaces, it increases reusability and facilitates integration into existing adapter architectures. It is ideal for applications where different data widths or formats need to be exchanged via standardized adapters.

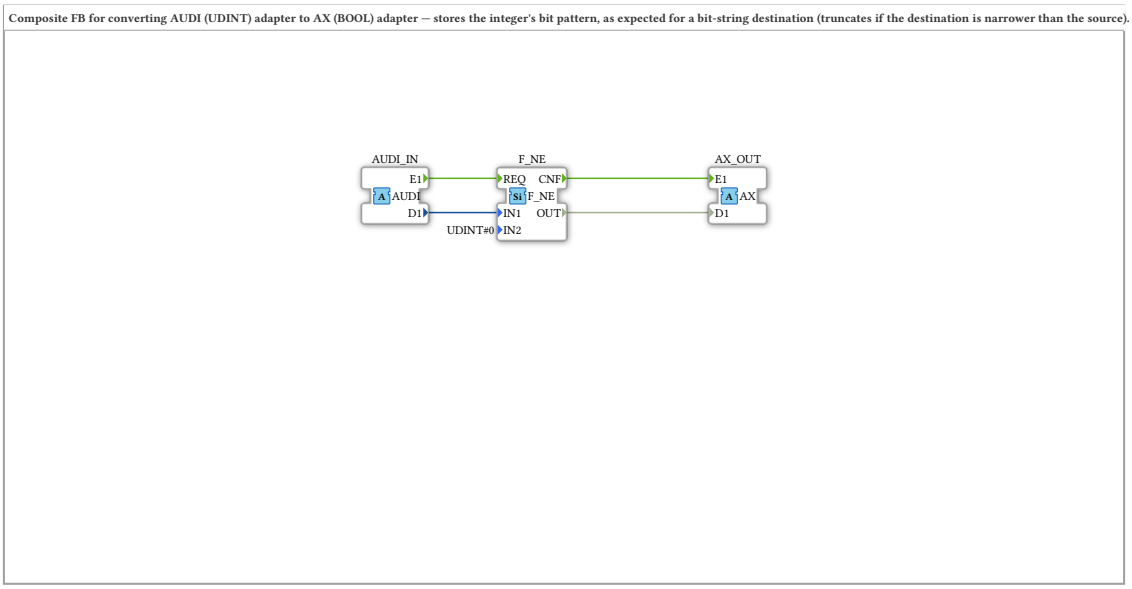
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUDI_TO_AX



UDINT Adapter Input – *AUDI* *AX* – BOOL Adapter Output



• – ◦ • – ◦ • – ◦ •

The function block `AUDI_TO_AX` is a composite function block that converts an **AUDI** adapter (UDINT data) into an **AX** adapter (BOOL data). It serves as a unidirectional converter for applications where an integer value (e.g., from a counter) needs to be mapped to a binary signal – specifically, it checks whether the input value is non-zero. The function block is implemented as a pure network connection using IEC 61131-compliant components. The function block does not have its own event inputs. The event to start processing is provided via the socket adapter `AUDI_IN` (see section Adapters).

The FB has no built-in event outputs. The result event is output via the plug adapter `AX_OUT` (see section Adapters).

The FB has no built-in data inputs. The UDINT value to be processed is provided via the socket adapter `AUDI_IN` (see section Adapters).

The FB has no built-in data outputs. The result (BOOL) is output via the plug adapter `AX_OUT` (see section Adapters).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Name	Type	Adapter Type	Direction	Comment
AUDI_IN	adapter::types::unidirectional::AUDI	Socket	Input	Provides the UDINT value (D1) and the associated event (E1)
AX_OUT	adapter::types::unidirectional::AX	Plug	Output	Provides the BOOL value (D1) and the acknowledgment event (E1)

Functionality

The internal process is divided into three steps:

1. **Event Reception:** The event `E1` of the socket adapter `AUDI_IN` triggers the processing.
2. **Comparison:** The received UDINT value (`AUDI_IN.D1`) is compared in the function block `F_NE` (IEC 61131-3: *not equal*) with the constant value `UDINT#0`.
3. **Result Output:** The result of the comparison (`F_NE.OUT`) – a BOOL – is placed on the output data port `AX_OUT.D1`. Simultaneously, the event `F_NE.CNF` is forwarded to the event input `AX_OUT.E1`, so that the output adapter signals the result.
4. **Result Output:** The result of the comparison (`F_NE.OUT`) – a BOOL – is placed on the output data port `AX_OUT.D1`. **Context:**

- If the input value is not equal to 0 ⟶ Output `TRUE`
- If the input value is equal to 0 ⟶ Output `FALSE`
- **Library Used:** The core consists of the IEC 61131 function block `F_NE` (non-equal comparison). This is imported from the library `iec61131::comparison::F_NE`.
- **No Dedicated ECC:** As a composite function block, `AUDI_TO_AX` does not have its own execution state – the entire logic is purely data flow-driven.
- **Unidirectional Adapters:** Both the input and output adapters are designed as *unidirectional*, meaning they only support a defined data and event flow from the source to the sink.
- – **Parameterization:** The comparison value is fixed at `UDINT#0` and cannot be externally parameterized.

The function block (FB) does not have its own state machine. Its behavior is entirely determined by incoming events: Every event at socket `AUDI_IN.E1` triggers an execution of the internal logic and an output at plug `AX_OUT`.

- **Binary Threshold Output:** A counter value (e.g., number of parts detected) should be passed on to a downstream controller as a switching signal (`TRUE` if present, `FALSE` if zero).
- **Network Simplification:** The function block encapsulates the non-zero logic, thereby reducing the complexity of higher-level function plans.

- **`AUDI_TO_BOOL` (hypothetical):** Would offer the same functionality, but without the adapter concept – possibly with direct data inputs/outputs.
- **Direct Comparison `F_NE`:** A single function block that performs the comparison, but without adapter adaptation.

The advantage of `AUDI_TO_AX` lies in its seamless integration into an adapter-based architecture, as is common in the 4diac IDE. It avoids additional conversion steps between different interface types.

The `AUDI_TO_AX` function block is a specialized yet highly useful converter for the unidirectional transfer of a UDINT value to a BOOL signal. By clearly encapsulating the comparison logic and using standard adapters, it simplifies the signal chain in IEC 61499-based automation systems. It is robust, easy to understand, and requires no complex configuration – ideal for common tasks such as binary meter reading.

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).

State Overview

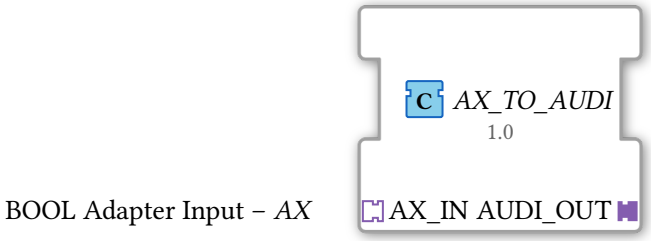
Application Scenarios

Comparison with Similar Function Blocks

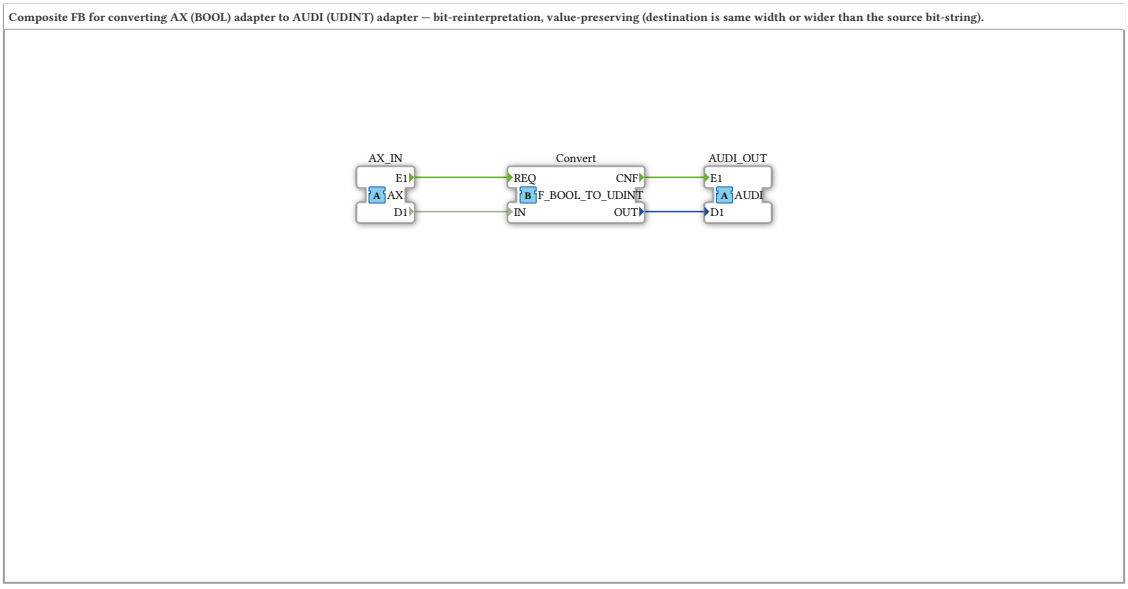
Conclusion

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

AX_TO_AUDI



BOOL Adapter Input – *AX* *AUDI* – UDINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The **AX_TO_AUDI** function block is a composite function block (FB) that converts a Boolean value (BOOL) provided via an AX adapter into an unsigned double-word value (UDINT) and outputs it via an AUDI adapter. It is used to convert signals between adapter interfaces, with the actual conversion performed by an internal IEC 61131 conversion block.

Interface Structure

Event Inputs

- **AX_IN.E1** (via socket adapter AX_IN): Event that triggers the conversion.

Event Outputs

- **AUDI_OUT.E1** (via plug adapter AUDI_OUT): Confirmation event after successful conversion.

Data Inputs

- **AX_IN.D1** (via socket adapter AX_IN, data type **BOOL**): The Boolean value to be converted.

Data Outputs

- **AUDI_OUT.D1** (via plug adapter AUDI_OUT, data type **UDINT**): The converted integer value (0 or 1).

Adapters

- **AX_IN** – Socket adapter of type `adapter::types::unidirectional::AX`

Accepts a Boolean value (and an associated event).

- **AUDI_OUT** – Plug adapter of type `adapter::types::unidirectional::AUDI`

Provides the converted UDINT value (and an acknowledgment event).

Functionality

The function block operates in an event-driven manner:

1. An event at the event input **AX_IN.E1** triggers the internal conversion block `F_BOOL_TO_UDINT` via its event input **REQ**.
2. The data value **AX_IN.D1** (BOOL) is forwarded to the data input **IN** of the conversion block.
3. The conversion block converts the Boolean value into a UDINT (FALSE ⟶ 0, TRUE ⟶ 1) and outputs the result at its output **OUT**.
4. The result is passed to the data output **AUDI_OUT.D1**.
5. After the conversion is complete, the acknowledgment event of the conversion block (**CNF**) is passed to the event output **AUDI_OUT.E1**.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Composite block (FB)** according to IEC 61499-2.
- Uses the internal FB `iec61131::conversion::F_BOOL_TO_UDINT` for the actual conversion.
- The adapter interfaces (`AX` and `AUDI`) enable modular and standardized integration with other blocks.
- The block is open source and licensed under the Eclipse Public License 2.0.

State Overview

The block does not have its own state machines. The behavior is entirely determined by the sequential processing of the internal conversion block:

- **Idle**: Waiting for an event at **AX_IN.E1**.
- **Conversion active**: After receiving the event, the conversion is performed; no further events are processed during this time (single execution).
- **Completion**: The output event is sent, after which the block returns to the idle state.

Application Scenarios

- Conversion of a digital sensor signal (e.g., limit switch) into a numerical value for further processing in a PLC.
- Adaptation between different adapter interfaces in IEC 61499-based automation systems.
- Use in systems where only Boolean values are available via an AX interface, but a UDINT interface is required (e.g., for diagnostic or counting purposes).

Comparison with Similar Function Blocks

- **F_BOOL_TO_UDINT** (direct): Performs the same conversion but requires separate event and data connections. **AX_TO_AUDI** encapsulates this logic in an adapter-based interface.
- AX_TO_ *function blocks**: Analogous to this function block, but with different target data types (e.g., INT, DINT). This function block is based on the UDINT variant.
- **Adapter coupler**: Pure pass-through function blocks without conversion – unlike **AX_TO_AUDI**, which performs an active type conversion.

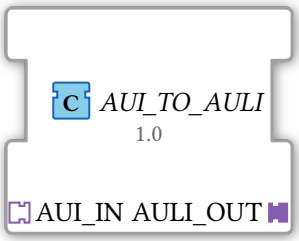
Conclusion

The **AX_TO_AUDI** function block provides a simple yet useful bridge between Boolean and integer adapter interfaces. By encapsulating the conversion in a composite block, the design becomes clearer and reusability is increased. It is particularly suitable for standard conversion tasks in IEC 61499 projects that use modular adapter interfaces.

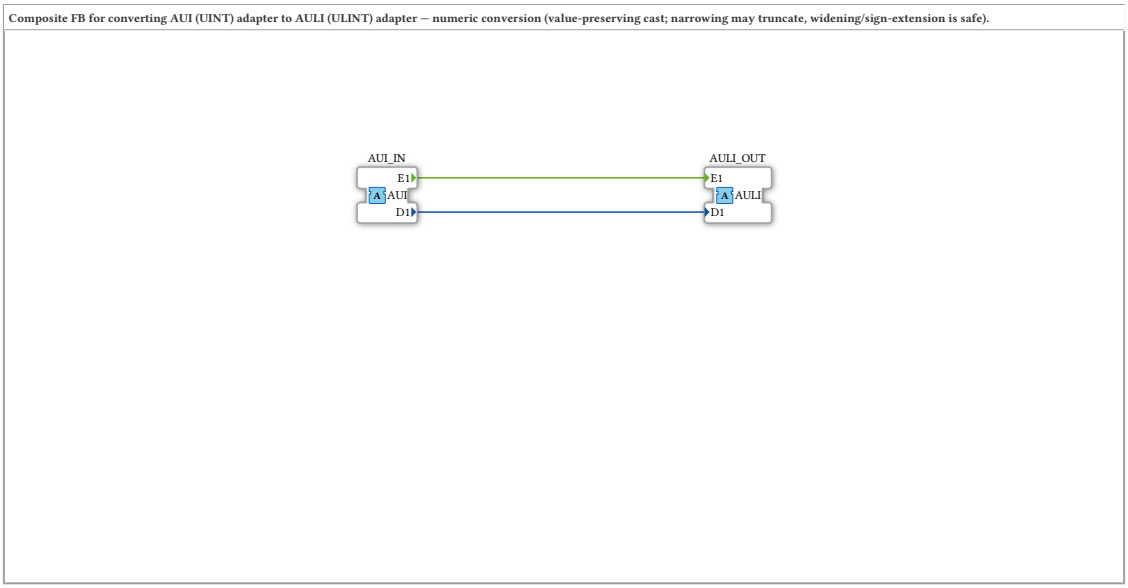
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AUI_TO_AULI



UINT Adapter Input – *AUI* *AULI* – ULINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_TO_AULI` is a composite function block for converting a unidirectional AUI adapter (data type `UINT`) into a unidirectional AULI adapter (data type `ULINT`). It serves as a simple type converter

between the two adapter interfaces, transparently passing on both the event signal and the data value. The function block enables seamless integration of components that use different addressing widths.

Interface Structure

The function block does not have its own event or data inputs/outputs, but uses only adapter interfaces. The following sections describe the signals that are routed externally via the adapters used.

Event Inputs

Signal	Description
(E1 via AUI_IN)	Event input of the connected AUI adapter (socket). An incoming event is forwarded directly to the AULI adapter.

Event Outputs

Signal	Description
(E1 via AULI_OUT)	Event output of the connected AULI adapter (plug). The event received from the AUI adapter is output here.

Data Inputs

Signal	Data Type	Description
(D1 via AUI_IN)	`UINT` (16 bits)	Data input of the AUI adapter. The incoming value is interpreted as a 16-bit integer.

Data Outputs

Signal	Data Type	Description
(D1 via AULI_OUT)	`ULINT` (64 bits)	Data output of the AULI adapter. The value received from the AUI adapter is output as a 64-bit unsigned integer.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AUI`	`AUI_IN`	Socket (Input)	Adapter for receiving events and data in the format `UINT`.
`adapter::types::unidirectional::AULI`	`AULI_OUT`	Plug (Output)	Adapter for sending events and data in the format `ULINT`.

Functionality

This function block forwards incoming events and data from the input adapter `AUI_IN` unchanged to the output adapter `AULI_OUT`. The data type `UINT` is implicitly converted to `ULINT` (the value remains the same; only the bit width is increased from 16 to 64 bits). The connection is established in the FB network of the composite module by directly wiring the respective event and data channels (`E1` and `D1`). No additional logic or state storage is performed.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Implicit Type Conversion:** The conversion from `UINT` to `ULINT` is performed automatically by the 4diac IDE and does not require an explicit conversion module. The value range from 0 to 65535 (UINT) is mapped to the corresponding ULINT range.
- **Implicit Type Conversion:** - **Pass-Through Behavior:** Events and data are transmitted without delay or buffering. The output event occurs simultaneously with the input event.

- **Unidirectional Adapters:** The module supports only one direction of data flow (from the AUI input to the AULI output). Reverse communication is not supported.

State Overview

The module does not have its own states, as it consists exclusively of direct connections. The behavior is purely combinatorial: An event at the input immediately generates an event at the output, and the data value is transmitted continuously.

Application Scenarios

- **Systems with Mixed Address Width:** When a sensor or component delivers data in the format `UINT` (e.g., a 16-bit counter value), but a downstream controller or bus interface only accepts `ULINT` (64-bit).
- **Adapter Coupling:** Used as an intermediary between two different adapter types in a modular control architecture based on the 61499 standard.
- **Prototype Development:** Rapid integration of library blocks that use different adapter interfaces without manual conversion.

Comparison with Similar Blocks

- **INT_TO_LINT / UINT_TO_ULINT (Direct Data Converters):** These blocks operate on pure data levels without event forwarding and require separate event processing. `AUI_TO_AULI` combines event and data conversion in a single adapter interface.
- **Adapter-Adapter Connector:** Other composite blocks that connect two similar adapters (e.g., AUI_TO_AUI) – they differ only in the absence of type conversion.
- **Explicit Multi-Channel Conversion Blocks:** More complex blocks that convert multiple signals simultaneously are overkill for simple point-to-point connections like this one.

Conclusion

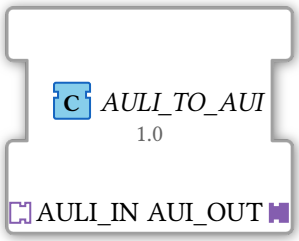
AUI_TO_AULIis a lean, specialized composite block for converting an AUI adapter to an AULI adapter. It simplifies type conversion between 16-bit and 64-bit addressing in IEC 61499 systems by automatically converting and forwarding both event and data paths. Thanks to the implicit type conversion fromUINTtoULINT`, manual implementation of conversion logic is eliminated, facilitating the design of modular and reusable control applications.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

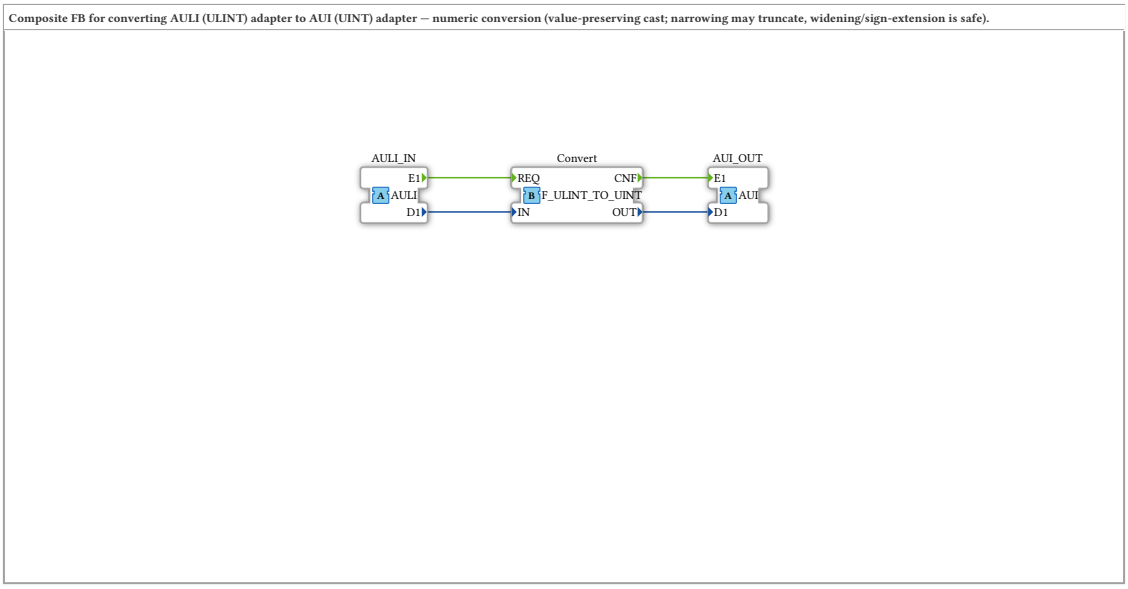
]

AULI_TO_AUI



ULINT Adapter Input – *AULI*

AUI – UINT Adapter Output



• — ○ • — ○ • — ○ •

Introduction

The function block **AULI_TO_AUI** is a composite function block (FB) that converts an **AULI** adapter (based on the data type `ULINT`) into an **AUI** adapter (based on the data type `UINT`). It is used when a 64-bit unsigned integer interface (`ULINT`) needs to be mapped to a 16-bit unsigned integer interface (`UINT`), for example, when connecting different sensor or actuator components in automation technology.

Interface Structure

Event Inputs

An event of type `AULI_IN` is provided via the adapter socket. This is forwarded internally to the conversion module.

Event Outputs

An event of type `AUI_OUT` is output via the adapter plug as soon as the conversion is complete.

Data Inputs

A data value of type `D1` (corresponding to `ULINT`) is received via the adapter socket `AULI_IN`.

Data Outputs

The adapter plug `AUI_OUT` outputs a data value of type `D1` (equivalent to `UINT`).

Adapter

Name	Direction	Type
AULI_IN	Socket	`adapter::types::unidirectional::AULI`
AUI_OUT	Plug	`adapter::types::unidirectional::AUI`

Functionality

The function block operates in an event-driven manner:

1. An event at input `AULI_IN.E1` triggers the internal function block `F_ULINT_TO_UINT` via its input `REQ`.
2. The data value `AULI_IN.D1` is directly passed to the conversion block as input `IN`.
3. The conversion block converts the `ULINT` value into a `UINT` value.
4. After the conversion is complete, the output `CNF` of the conversion block signals an event, which is then forwarded to the output `AUI_OUT.E1`.
5. Simultaneously, the converted value is transferred from the `OUT` output of the conversion module to the `AUI_OUT.D1` output.

The conversion is performed according to IEC 61131 standard `F_ULINT_TO_UINT`. Values greater than `65535` (maximum `UINT` value) may result in an overflow, which must be handled by the application.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Adapter-Based Encapsulation**: The module allows for the seamless replacement of adapters in the 4diac IDE without changing the logic of the surrounding network.
- **Reuse of IEC standard conversion**: The actual type conversion is performed by the standardized IEC function block `F_ULINT_TO_UINT`, ensuring portability and correctness.
- **No state storage**: The function block is purely combinatorial – after each execution, the current result is available at the output.

State overview

The function block does not have its own finite state machine. The internal conversion is event-driven and without internal states. The runtime is determined solely by the underlying `F_ULINT_TO_UINT` function block.

Application scenarios

- **Sensor integration**: A 64-bit sensor (e.g., high-resolution displacement measurement) is connected to a controller that only accepts 16-bit inputs.
- **Communication Gateway**: Adapter interfaces between bus systems (e.g., CANopen with 64-bit data to PROFINET I device with 16-bit data).
- **Data Reduction**: Targeted reduction of the resolution for subsequent processing steps (e.g., display or simple threshold logic).

Comparison with similar components

Component	Input type	Output type	Special feature
`AULI_TO_AUI`	`ULINT`	`UINT`	Adapter Wrapper, Event-Driven
`F_ULINT_TO_UINT`	`ULINT`	`UINT`	Pure Function, No Adapter
`AUI_TO_AULI`	`UINT`	`ULINT`	Reverse Direction (Not Available)

Unlike a bare IEC function, this block offers an adapter-compliant input and output interface, thus enabling easy integration into adapter-based architectures.

Conclusion

The **AULI_TO_AUI** block is a practical solution for mapping 64-bit UINT data to 16-bit UINT data using a standardized adapter mechanism. It combines the robustness of IEC conversion functionality with the flexibility of 4diac adapters and is particularly well-suited for heterogeneous automation systems where different data types interact.

🌐 Related topic subpages on ms-muc-docs.de

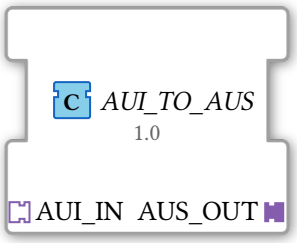
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Numeric Index (unidirectional)

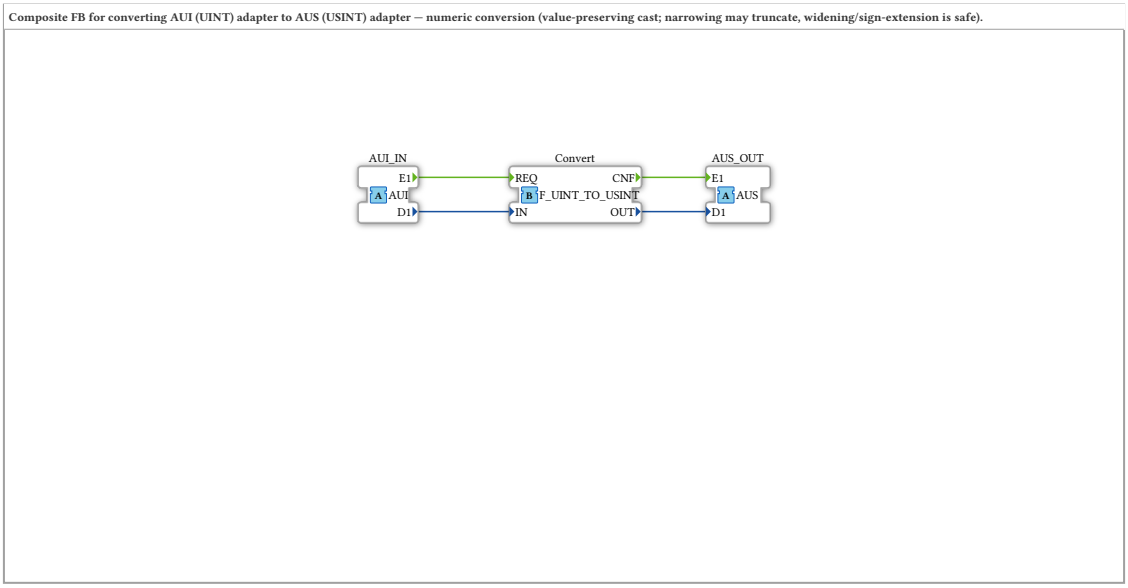
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUI_TO_AUS



UINT Adapter Input – *AUI*

AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

The function block **AUI_TO_AUS** serves as a composite module for converting an AUI adapter (data type UINT) into an AUS adapter (data type USINT). It is typically used to mediate between different adapter types in the 4diac IDE when a reduction in the value range or a type conversion is required. The module encapsulates the conversion logic and enables seamless integration into existing runtime environments.

No separate event inputs. Event control is handled via the connected adapters.
No separate event outputs. Event propagation is handled via the connected adapters.
No separate data inputs. Data input is via the connected adapters.
No separate data outputs. Data output is via the connected adapters.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Direction	Name	Type	Description
Socket	**AUI_IN**	adapter::types::unidirectional::AUI	UINT input adapter (source)
Plug	**AUS_OUT**	adapter::types::unidirectional::AUS	USINT output adapter (destination)

The socket `AUI_IN` accepts the UINT value (16-bit unsigned) and the corresponding event signal. The plug `AUS_OUT` outputs the converted USINT value (8 unsigned bits) along with the corresponding event.

Functionality

The function block operates as a simple pass-through element with type conversion:

1. **Event Reception**: An incoming event at socket `AUI_IN.E1` triggers processing.
2. **Data Acquisition**: The corresponding data value `AUI_IN.D1` (type UINT) is read.
3. **Conversion**: Internally, the function `F_UINT_TO_USINT` from the IEC 61131 library is called. This function converts the UINT value into a USINT value. If the USINT value range (0…255) is exceeded, the value is truncated or handled according to the underlying implementation (usually capped at 255).
4. **Output**: The converted value is made available on `AUS_OUT.D1` (type USINT), and the output event `AUS_OUT.E1` is sent.

All coupling occurs via the event and data connections in the FB network of the composite block.

- **Range Limitation**: Converting UINT (0…65535) to USINT (0…255) inevitably results in information loss if the input value is greater than 255. Users must ensure that the input value range meets expectations.
- **No State Storage**: The FB is stateless – all processing occurs independently of previous conversions.
- **Reusability**: The function block is designed as an encapsulated component and can be used in different projects without having to reimplement the conversion logic.
- **Compatibility**: The availability of the library `iec61131::conversion::F_UINT_TO_USINT` is required.

The function block does not have an explicit state machine. However, the internal process can be described as two-step:

- **Ready (Idle)**: The function block waits for an incoming event at the socket.
- **Processing (Convert)**: After the event is received, the conversion is performed and the output event is triggered. The function block then immediately returns to the ready state.

A detailed state graph is not required, as the function block does not have any delays or multi-stage steps.

- **Sensor Connection**: A sensor delivers values in UINT format that must be passed to subsequent logic with a USINT request.
- **Device Coupling**: Seamless communication is established between two devices or components that use different adapter types.
- **Data Reduction**: In cases where the higher resolution of a UINT is not required, conversion to USINT can offer storage and bandwidth advantages.
- **System Migration**: When switching from older to newer adapter protocols, this function block can serve as a temporary or permanent bridge.

Other conversion function blocks exist in the 4diac library, e.g., `F_UINT_TO_USINT`, which directly maps a UINT input to a USINT output. The **AUI_TO_AUS** described here differs in that it uses adapters (AUI and AUS) instead of basic data inputs. This allows it to encapsulate the entire interface, including event control, simplifying its use in complex, adapter-based architectures. A pure function block like `F_UINT_TO_USINT`, on the other hand, requires separate event and data connections.

The **AUI_TO_AUS** is a compact and focused function block for adapter-based conversion from UINT to USINT. Its composite structure facilitates integration into existing 4diac projects, reduces wiring effort, and improves clarity. Its simple functionality and clear interface make it a useful tool for all applications requiring type conversion at the adapter level.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).

State Overview

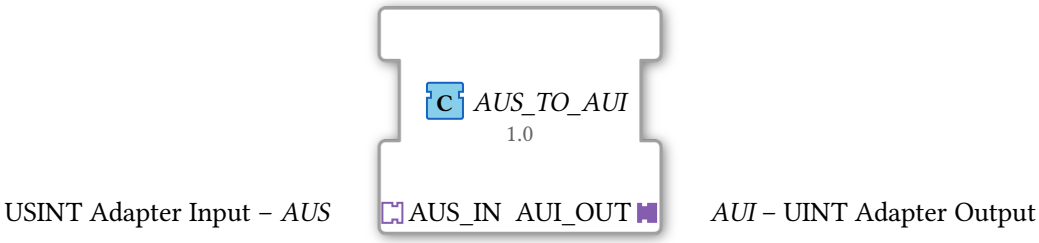
Application Scenarios

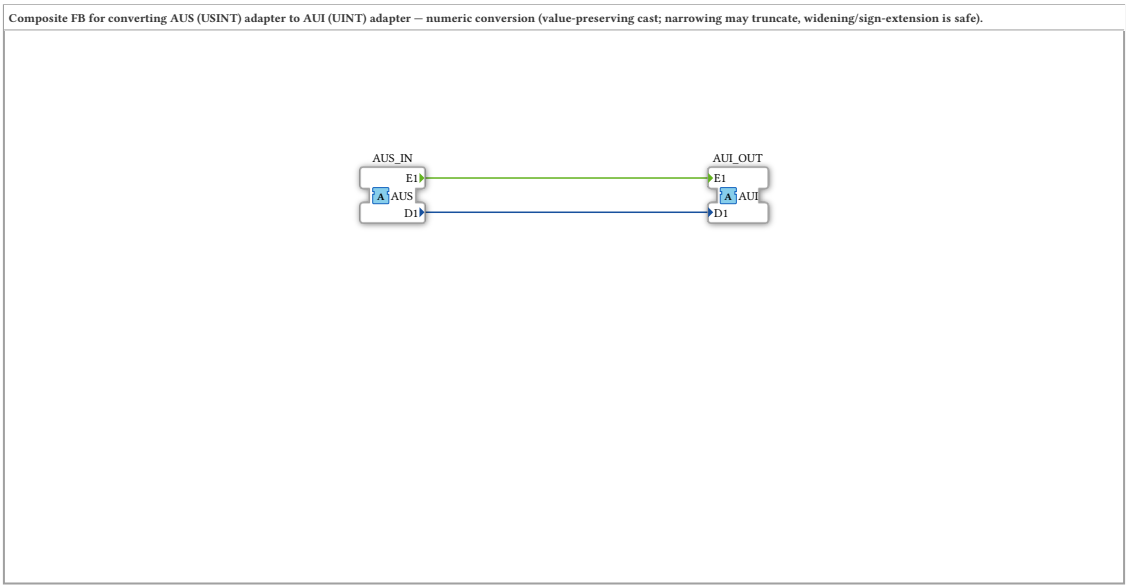
Comparison with Similar Function Blocks

Conclusion

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

AUS_TO_AUI





• – ○ • – ○ • – ○ •

Introduction

The **AUS_TO_AUI** function block is a composite component that performs an interface conversion from an AUS adapter (USINT data type) to an AUI adapter (UINT data type). It serves as a simple passthrough, where both the event and the data value are transferred directly from the input to the output adapter. The implicit type conversion from USINT (8-bit unsigned) to UINT (16-bit unsigned) is performed automatically by the runtime environment.

Interface Structure

Event Inputs

Name	Data Type	Description
E1	Event	Event input of the AUS adapter; passed unchanged to the output.

Event Outputs

Name	Data Type	Description

E1	Event	Event output of the AUI adapter; receives the event from the input.

Data Inputs

Name	Data Type	Description
D1	USINT	Data value (8-bit) from the AUS adapter; converted to UINT and passed on.

Data Outputs

Name	Data Type	Description
D1	UINT	Data value (16-bit) of the AUI adapter; contains the converted USINT value.

Adapter

Type	Name	Direction	Description
adapter::types::unidirectional::AUS	AUS_IN	Socket (Input)	Out adapter with USINT data and one event.
adapter::types::unidirectional::AUI	AUI_OUT	Plug (Output)	AUI adapter with UINT data and one event.

Functionality

The function block operates as a simple passthrough:

- An incoming event at `AUS_IN.E1` is passed directly to `AUI_OUT.E1`.
- The incoming data value from `AUS_IN.D1` (USINT) is passed to `AUI_OUT.D1` without any additional logic. The type conversion from USINT to UINT is performed automatically by the 4diac IDE or the target platform.

There are no internal function blocks or algorithms that manipulate the signal.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Function Block**: It does not contain its own function block instances, but simply connects the adapter ports directly.
- **Implicit Type Conversion**: The conversion from USINT to UINT is not explicitly implemented in the function block, but is supported by the runtime system. The 8-bit value is transferred to the less significant part of the 16-bit word.
- **Unidirectional Adapters**: Both AUS and AUI are defined as unidirectional adapters, meaning data flows are only intended to occur in one direction.

State Overview

The function block has no states of its own, as it contains no internal logic or state machines. It reacts exclusively to input events and forwards them immediately.

Application Scenarios

- **System Integration**: When a control system (e.g., field devices) delivers data in USINT format, but a downstream module expects UINT.
- **Adapter Switching**: In a PLC or distributed control environment, standard adapters of different types can be coupled using such a conversion function block.
- **Prototyping**: Rapid adaptation of interfaces during the development phase without changing the underlying adapters.

Comparison with Similar Function Blocks

- **INT_TO_UINT**: Converts an integer data type to an unsigned integer – also available as a composite or basic function block.
- **AUS_TO_AUI** is specifically designed for the adapter types AUS (USINT) and AUI (UINT). Other function blocks, such as `SINT_TO_USINT`, use analogous passthroughs for other data types.
- Unlike function blocks with arithmetic or logical processing, this function block is limited to pure conversion and event forwarding.

Conclusion

The **AUS_TO_AUI** function block is a streamlined tool for adapting adapter interfaces between USINT and UINT. Its simple passthrough structure makes it performant and easy to understand. It is particularly suitable for integration tasks where existing components with different data widths need to be connected. Due to the lack of internal logic, it is robust and requires no parameterization.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

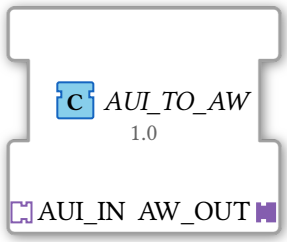
]

Numeric Index (unidirectional)

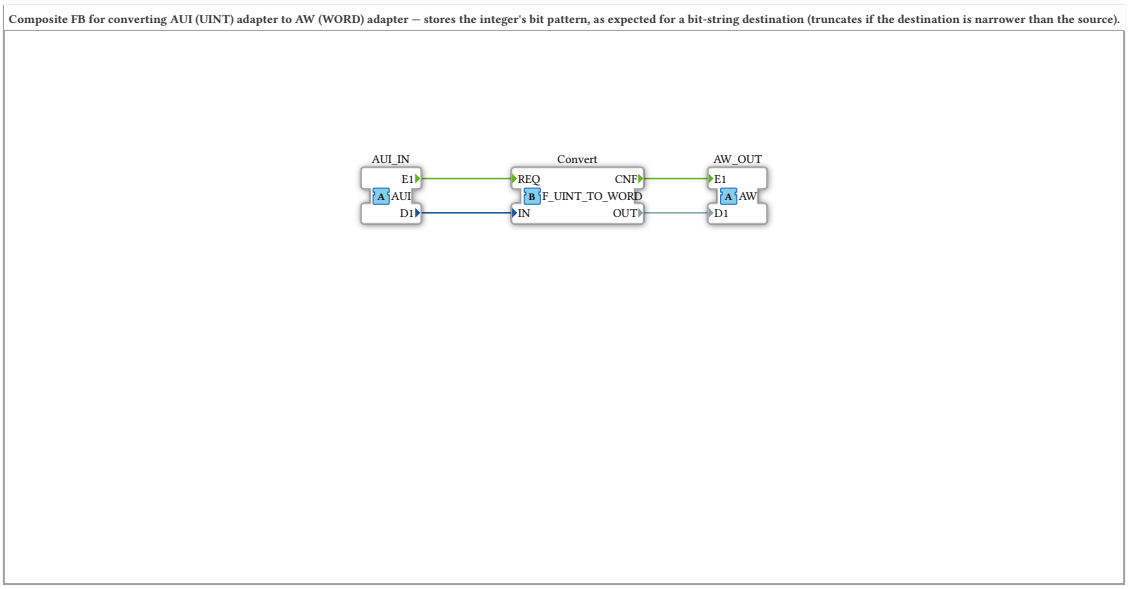
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AUI_TO_AW

UINT Adapter Input – *AUI*



AW – WORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block `AUI_TO_AW` is a composite module that converts an AUI adapter (data type UINT) into an AW adapter (data type WORD). It encapsulates the type conversion and provides it as a standardized adapter interface, enabling seamless integration of system components with different data types.

Interface Structure

Event Inputs

Name	Type	Comment
AUI_IN.E1	Event (via adapter)	Trigger to convert the incoming UINT value

Event Outputs

Name	Type	Comment
AW_OUT.E1	Event (via adapter)	Confirmation that the converted WORD value has been provided

Data Inputs

Name	Data Type	Comment
AUI_IN.D1	UINT (via adapter)	The input value to be converted

Data Outputs

Name	Data Type	Comment
AW_OUT.D1	WORD (via adapter)	Converted Output Value

Adapter

Direction	Name	Type	Comment
Socket (Input)	AUI_IN	adapter::types::unidirectional::AUI	UINT adapter as source
Plug (Output)	AW_OUT	adapter::types::unidirectional::AW	WORD adapter as sink

Functionality

This function block responds to an event on socket AUI_IN.E1. The current value of AUI_IN.D1 (data type UINT) is then forwarded to the internal conversion block F_UINT_TO_WORD. This block converts the value to the WORD data type according to IEC 61131. Once the conversion is complete, the result is passed to the plug AW_OUT.D1, and an acknowledgment event is triggered on AW_OUT.E1.

The interaction of event and data flow ensures secure and deterministic data transmission.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Structure:** The block encapsulates the complete conversion logic in a single function block, eliminating the need for any further conversion steps in the application.
- **Type Conversion:** Conversion is performed using the standardized IEC 61131 function block F_UINT_TO_WORD. UINT and WORD are bit-compatible, so no data loss occurs.
- **Adapter Interface:** The use of adapters enables loose coupling of the components. The function block can be directly integrated into adapter-based networks.
- **No States:** The function block has no state behavior (ECC). It operates purely event-driven and performs an immediate conversion upon each trigger.

State Overview

The function block does not contain its own state machine (ECC). Sequence control is handled solely through the event connections in the composite network. After startup, the function block is immediately ready to process incoming events and perform the conversion.

Application Scenarios

- **Adapter Bridge:** If one system component communicates only via an AUI adapter (UINT), but another component expects an AW adapter (WORD), `AUI_TO_AW` can be used as an intermediary.
- **Data Type Adaptation:** In Industry 4.0 or IIoT environments where devices use different data types, this component harmonizes the interfaces.
- **Modular Chaining:** The component can be easily inserted into a chain of adapters to ensure compatibility between components.

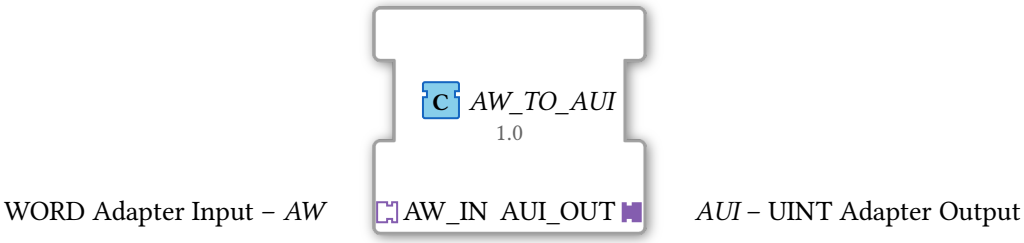
Comparison with Similar Components

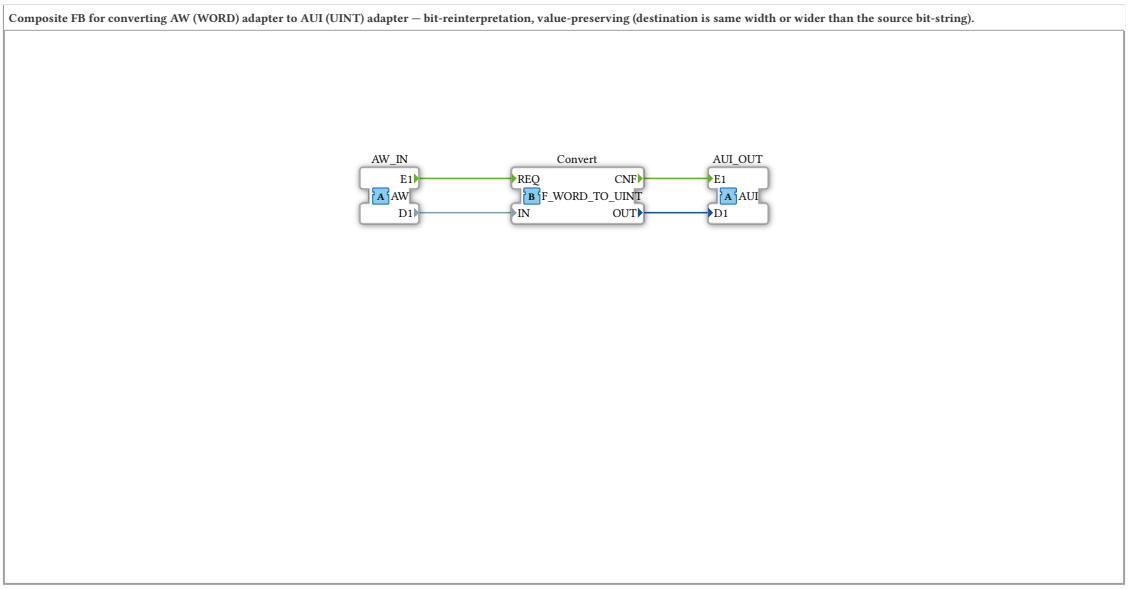
- **Direct Use of `F_UINT_TO_WORD`:** Instead of the composite component, `F_UINT_TO_WORD` could be used directly, and the adapters connected separately. The composite block simplifies cabling and increases reusability.
- **Analog converters (e.g., `AWORD_TO_AUINT`):** Corresponding modules exist for the reverse direction. `AUI_TO_AW` focuses specifically on the conversion from UINT to WORD and offers a clearly named, targeted solution.

Conclusion

The `AUI_TO_AW` function block provides a compact and robust solution for converting a UINT adapter signal to a WORD adapter signal. By encapsulating the standard conversion and using the adapter interface, integration into existing systems is facilitated, and network readability is improved. The module is ideally suited to reconcile data type differences between components without compromising application clarity.

AW_TO_AUI





• – ○ • – ○ • – ○ •

Introduction

The function block **AW_TO_AUI** is a composite function block (FB) that converts an **AW (WORD)** adapter into an **AUI (UINT)** adapter. It enables the seamless integration of components that use different

data types via unidirectional adapters. Internally, it uses the standard conversion block `F_WORD_TO_UINT` for type conversion.

Interface Structure

Event Inputs
The FB does not have its own independent event inputs. The event `E1` is provided via the **socket (input adapter) `AW_IN`** and triggers the data conversion.

Event Outputs
The FB does not have any standalone event outputs. The event `E1` is output via the **plug (output adapter) `AUI_OUT`** after successful conversion.

Data Inputs
The FB does not have any standalone data inputs. The date `D1` of type `WORD` is provided via the **socket `AW_IN`**.

Data Outputs
The FB does not have any standalone data outputs. The date `D1` of type `UINT` is output via the **plug `AUI_OUT`**.

Adapter

Name	Direction	Type	Description
`AW_IN`	Input	`adapter::types::unidirectional::AW`	WORD Adapter Input (Socket)
`AUI_OUT`	Output	`adapter::types::unidirectional::AUI`	UINT Adapter Output (Plug)

Functionality
This function block acts as a conversion layer between two different adapter types. Internally, a single function block, `F_WORD_TO_UINT`, is embedded:

1. **Event Control**:

- An incoming event at socket `AW_IN.E1` triggers the conversion process (`Convert.REQ`).
- After the conversion is complete, the internal block generates an acknowledgment event (`Convert.CNF`), which is forwarded to plug `AUI_OUT.E1`.

1. **Data Flow**:

- The value `WORD`, provided by socket `AW_IN.D1`, is passed to input `Convert.IN`.
- The output `Convert.OUT` returns the converted `UINT` value, which is then passed to the plug `AUI_OUT.D1`.

The conversion is performed according to the IEC 61131-3 standard conversion `WORD_TO_UINT`.

Technical Features
- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Pure Composite Structure**: The function block (FB) does not have its own state machine or logic, but delegates all functionality to the embedded standard function block.
- **Unidirectional Adapters**: Both the input and output adapters are designed as unidirectional types, enabling simple and directional data transfer.
- **Reusability**: The function block can be used in projects where interface adaptation between WORD- and UINT-based components is required.

State Overview
Since this is a composite function block without its own state machine, there are no internal states. The function block behaves strictly event-driven: Every input event is followed—after conversion is complete—by an output event.

Application Scenarios

- **System Integration**: Connecting a component that provides data as `WORD` via an AW adapter to another component that expects `UINT` via an AUI adapter.
- **Protocol Adaptation**: Use in middleware or gateway functions to harmonize different data formats within a control architecture.
- **Modular Extension**: Easy integration into existing 4diac networks without requiring changes to adjacent function blocks.

Comparison with Similar Modules

Other conversion modules (e.g., `F_WORD_TO_DWORD` or `F_INT_TO_UINT`) operate directly on data ports. In contrast, `AW_TO_AUI` is specifically designed for adapter-based communication and encapsulates both the type conversion and the adapter logic in a single unit. This simplifies the handling of adapter interfaces and keeps network design clear.

Conclusion

The `AW_TO_AUI` function block offers a compact and reliable solution for adapter conversion from `WORD` to `UINT`. Thanks to the use of a standardized IEC 61131 conversion module and clear event control, it is easy to understand and maintain. It is particularly suitable for projects that require a clean separation between different data types at the adapter level.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

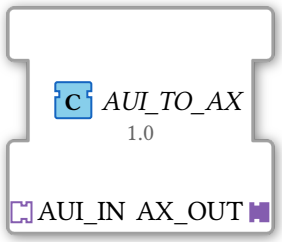
]

Numeric Index (unidirectional)

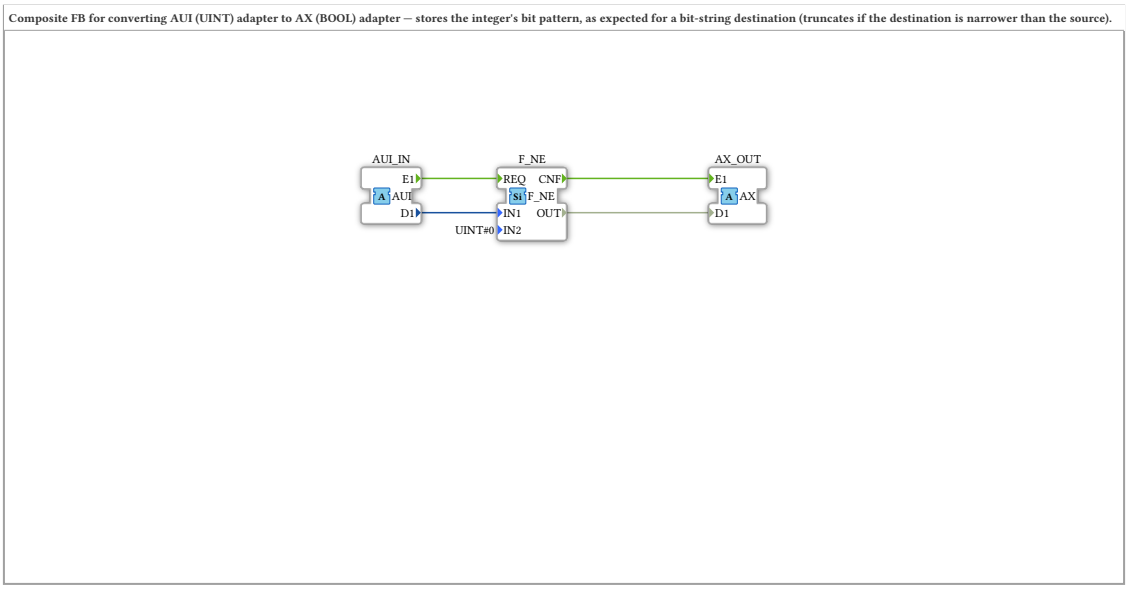
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUI_TO_AX

UINT Adapter Input – *AUI*



AX – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AUI_TO_AX** function block is a composite block that converts an AUI adapter (type *AUI* with a UINT value) into an AX adapter (type *AX* with a BOOL value). It serves as a bidirectional conversion interface between two different adapter types and is typically used to provide digital comparison results for further processing in BOOL-based adapters.

Interface Structure

Event Inputs

No direct event inputs. Event control is handled exclusively via the **AUI_IN** socket adapter.

Event Outputs

No direct event outputs. The result event is passed on via the **AX_OUT** plug adapter.

Data Inputs

No direct data inputs. The UINT value to be processed is provided via the socket adapter **AUI_IN**.

Data Outputs

No direct data outputs. The BOOL result is output via the plug adapter **AX_OUT**.

Adapter

- **AUX_IN (Socket)** – Type: *adapter::types::unidirectional::AUI*
- Event: E1 (triggering event for the comparison)
- Data: D1 (UINT value compared to 0)
- **AX_OUT (Plug)** – Type: *adapter::types::unidirectional::AX*
- Event: E1 (confirmation event after successful comparison)
- Data: D1 (BOOL result: TRUE if D1 ≠ 0, otherwise FALSE)

Functionality

The function block internally contains an instance of the comparison function block *iec61131::comparison::F_NE* (inequality). The process is as follows:

1. An event on socket **AUI_IN.E1** triggers the input **REQ** of the F_NE function block.
2. The UINT value supplied by the socket via **AUI_IN.D1** is compared to the constant *UINT#0*.
3. The comparison result (BOOL) is provided at output **F_NE.OUT**.
4. The result event **F_NE.CNF** is forwarded to plug **AX_OUT.E1**.
5. The BOOL result is simultaneously transmitted to **AX_OUT.D1** via the data connection.

Therefore:

- AUI_IN.D1 = 0 ⟶ AX_OUT.D1 = FALSE
- AUI_IN.D1 ≠ 0 ⟶ AX_OUT.D1 = TRUE

Technical Specifications

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **License:** Eclipse Public License 2.0 (EPL-2.0)
- **Author:** Franz Höpfinger, HR Agrartechnik GmbH
- **Package:** `adapter::conversion::unidirectional`
- **Internal Block:** `iec61131::comparison::F_NE` with parameter `IN2 = UINT#0`
- The block is a pure composite and does not contain its own state machine. All logic is implemented by the embedded F_NE.

State Overview

Since this is a composite block without its own ECC (Execution Control Chart), there is no explicit state machine. The behavior is entirely determined by the internal F_NE block, which performs a simple comparison without intermediate storage. The block is therefore stateless.

Application Scenarios

- **Threshold Detection:** Converting an analog (UINT) sensor value into a digital signal (BOOL), e.g., “non-zero value” as an alarm or control signal.
- **Adapter Conversion:** Use in systems that employ different adapter standards (AUI ↔ AX) to ensure compatibility between components.
- **Simple Logic Chaining:** Use as a binary decision stage in larger composite function blocks.

Comparison with Similar Blocks

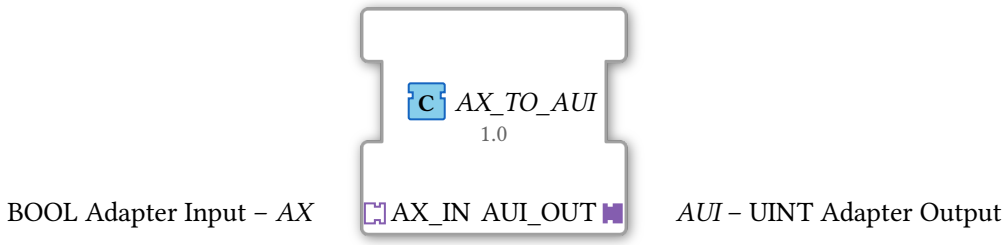
Block	Input Type	Output Type	Function
AUI_TO_AX	AUI (UINT)	AX (BOOL)	Inequality with Zero
AUI_TO_AX_INV (hypothetical)	AUI (UINT)	AX (BOOL)	Equality with Zero (inverted)
CONV_UINT_TO_BOOL (direct)	UINT (Data)	BOOL (Data)	Direct data converter, no adapter

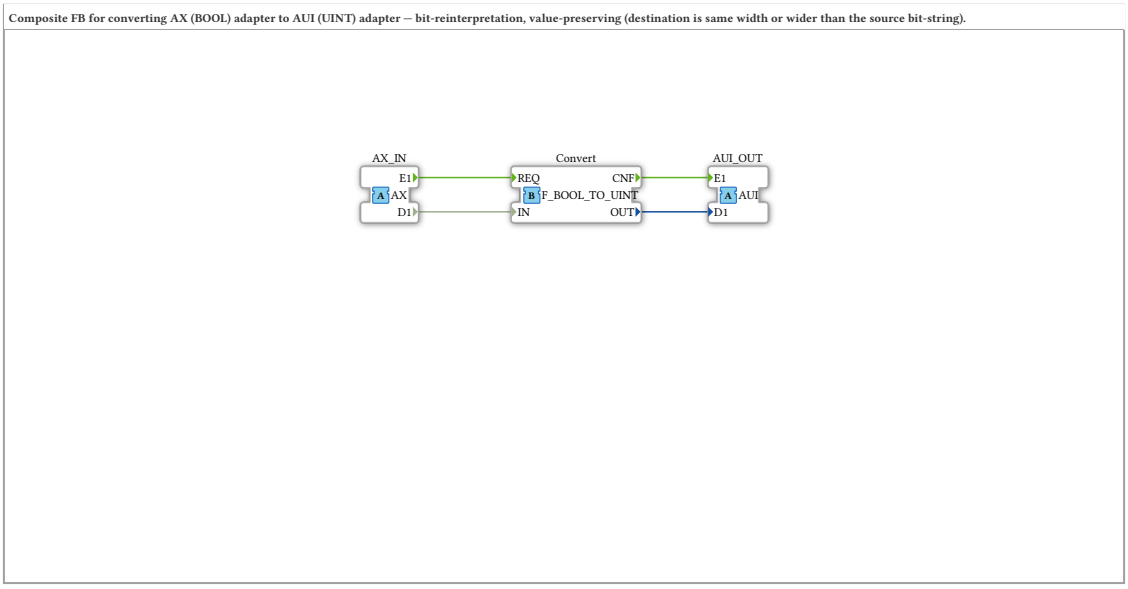
This function block stands out from simple data converters due to its full adapter compliance and integrated event handling.

Conclusion

AUI_TO_AX is a compact, purpose-built composite function block for converting a UINT-based AUI adapter into a BOOL-based AX adapter. Thanks to the use of the standardized F_NE comparison, the function is reliable and easy to understand. The component is particularly suitable for integration into adapter-based automation architectures where a simple zero comparison needs to be elevated to the signal level.

AX_TO_AUI





• – ◦ • – ◦ • – ◦ •

Introduction

The AX_TO_AUI is a composite function block (FB) that converts a BOOL value from an AX adapter into a UINT value for an AUI adapter. It encapsulates the conversion logic and enables seamless integration between unidirectional adapters of different data types. The conversion is event-driven via an internal IEC 61131 conversion block.

Interface Structure

The block has no direct event or data interfaces, but only adapter interfaces. The following list describes the signals provided or required by the adapters.

Event Inputs

- **AX_IN.E1** (Event from the connected AX adapter): Triggers the conversion of the provided BOOL value.

Event Outputs

- **AUI_OUT.E1** (Event for the connected AUI adapter): Signals that the converted UINT value is present.

Data Inputs

- **AX_IN.D1** (BOOL): Boolean input value (e.g., 0 or 1) to be converted to a UINT value.

Data Outputs

- **AUI_OUT.D1** (UINT): Integer output value (0 or 1) corresponding to the converted BOOL input.

Adapter

- **AX_IN** (Socket) – Type: `adapter::types::unidirectional::AX` (BOOL)
- **AUI_OUT** (Plug) – Type: `adapter::types::unidirectional::AUI` (UINT)

Functionality

The function block operates as a simple pipeline: An incoming event at socket AX_IN.E1 triggers the internal function block `F_BOOL_TO_UINT`. This block reads the current value from AX_IN.D1, converts it to a UINT value (BOOL FALSE $\longrightarrow$ 0, BOOL TRUE $\longrightarrow$ 1), and places the result at AUI_OUT.D1. Upon completion, an event is output via AUI_OUT.E1. The entire process is atomic and event-driven; there is no internal state.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- The function block is implemented as a composite function block, meaning... It consists of a network of subordinate FB instances (here only `F_BOOL_TO_UINT`).
- The conversion follows the IEC 61131-3 conversion rule: BOOL FALSE = 0, BOOL TRUE = 1.
- The use of adapters enables loose coupling between components of different domains (BOOL control and UINT communication).
- The FB does not have its own state machine; execution is purely event-driven.

State Overview

Since this is a composite FB without its own ECC (Execution Control Chart), there are no explicit states. Its functionality is limited to the event-driven execution of the internal FB.

Application Scenarios

- **Control Interface**: A BOOL signal (e.g., switch on/off) is converted into a numerical value to be sent to a higher-level controller via an AUI adapter.
- **Protocol Adaptation**: Connecting a BOOL-based sensor to a system that expects UINT values (e.g., for further processing in arithmetic blocks).
- **Standardization of Adapter Interfaces**: The function block acts as an adapter between two different unidirectional adapter types, avoiding redundant development.

Comparison with Similar Function Blocks

- **AX_TO_AUI** converts BOOL to UINT.

- **AX_TO_ASI** would convert BOOL to SINT, **AX_TO_ADI** BOOL to DINT – all follow the same pattern, but with different target types.
- Unlike a generic `F_BOOL_TO_UINT`, this function block offers an encapsulated adapter interface and can be directly integrated into adapter-based architectures.

Conclusion

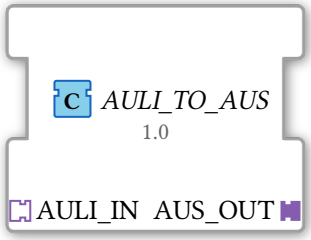
The AX_TO_AUI is a small, specialized function block for converting BOOL to UINT data via adapter interfaces. It fulfills a typical integration task in IEC 61499-based systems, where different data types need to be connected via unidirectional adapters. Its composite structure makes the implementation transparent and extensible.

Numeric Index (unidirectional)

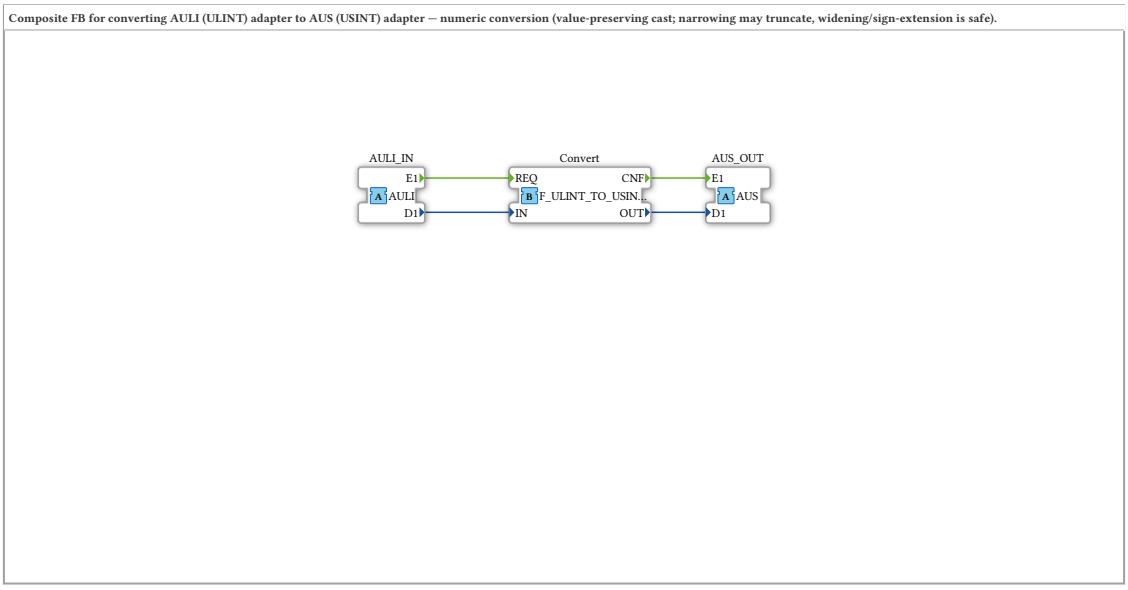
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AULI_TO_AUS

ULINT Adapter Input – *AULI*



AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AULI_TO_AUS** function block is a composite function block that performs a conversion from an **AULI adapter** (ulint, 64-bit unsigned) to an **AUS adapter** (usint, 8-bit unsigned). It encapsulates the IEC 61131 function **F_ULINT_TO_USINT** in an adapter-based environment, enabling data exchange via standardized adapters.

Interface Structure

The function block has no direct event or data inputs/outputs at the FB level. All communication takes place via the two adapters.

Event Inputs

- via **Socket AULI_IN** (Type AULI): Event **E1** – initiates data transfer and conversion.

Event Outputs

- via **Plug AUS_OUT** (Type AUS): Event **E1** – is triggered as soon as the converted value is present at the output.

Data Inputs

- via **Socket AULI_IN** (Type AULI): Data input **D1** (ULINT) – the 64-bit value to be converted.

Data Outputs

- via **Plug AUS_OUT** (Type AUS): Data output **D1** (USINT) – the converted 8-bit value.

Adapter

Name	Type	Direction	Description
AULI_IN	AULI	Socket	Input adapter for ULINT data
AUS_OUT	AUS	Plug	Output adapter for USINT data

Functionality

1. The FB receives an event at **E1** via the **AULI_IN** socket. Simultaneously, the corresponding ULINT value is provided at **D1**.
2. This event is forwarded to the internal function block **F_ULINT_TO_USINT** (name: `Convert`) (via the event connection `REQ`).
3. The ULINT value is passed to the **IN** data input of the converter.
4. After the conversion is complete, the converter generates the **CNF** event.
5. This event is transferred to the **AUS_OUT** plug at its **E1** event input, while the converted USINT value is transferred from the **OUT** output of the converter to the **D1** data output of the plug.
6. The connected receiver can now process the event and the USINT value.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Block** – It contains an internal instance of **F_ULINT_TO_USINT** from the IEC 61131 library.
- **Adapter-Based Communication** – No direct data/events at the top-level function block; input/output is handled exclusively via the supplied adapters (AULI / AUS).
- **Data loss possible** – Since the USINT data type only comprises 8 bits, values > 255 are lost (truncated or wrapped around, depending on the converter implementation).
- **No dedicated state machine** – The function block (FB) is a pure network of existing blocks and does not have its own state machine.

State overview

Due to its composite nature and the lack of a dedicated state machine, a state overview is not available. Its behavior is entirely determined by the internal FB **F_ULINT_TO_USINT**.

Application scenarios

- **Integration of ULINT data sources** (e.g., high-resolution counters, timestamps, or large sensor data) into an automation system that can only process USINT values due to memory or protocol limitations.
- **Adapter conversion** in a distributed application to mediate between two different adapter types (AULI ↔ AUS) without changing the underlying logic.
- **Prototypical conversions** from 64-bit to 8-bit data paths during development.

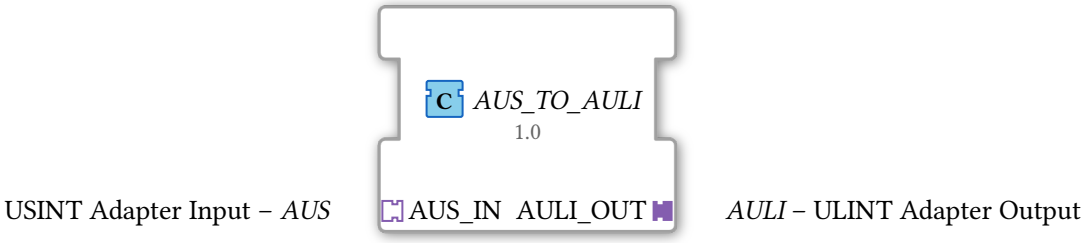
Comparison with similar components

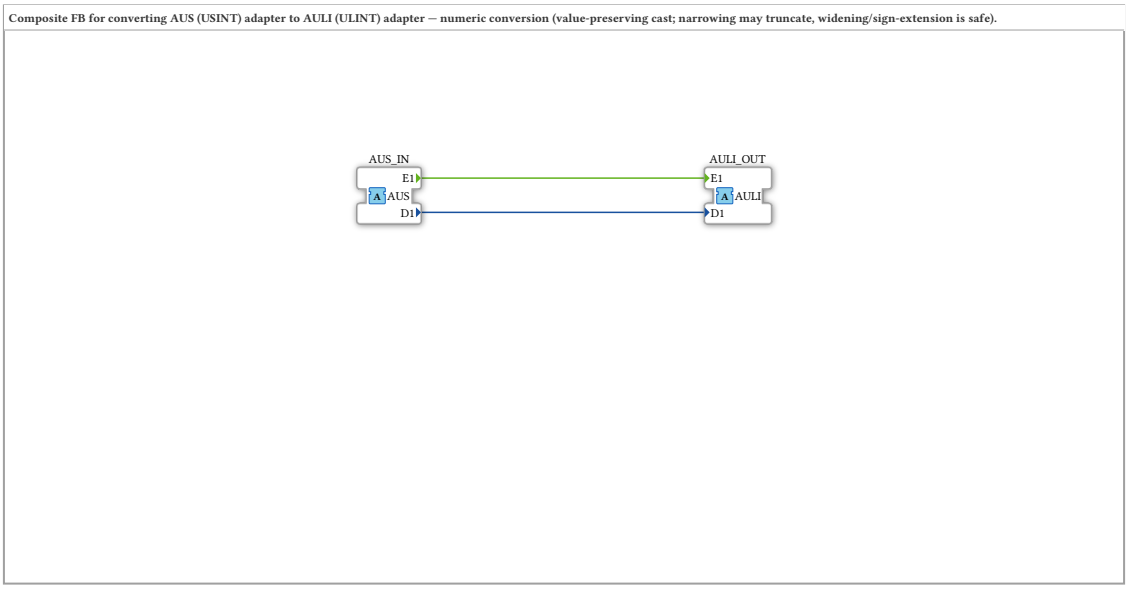
Similar components such as **AULI_TO_AINT** (ULINT → AINT) or **AULI_TO_AWORD** (ULINT → AWORD) implement analogous conversions for other output types. This component specializes in **USINT** – a particularly memory-efficient target variable. Unlike direct application-level data conversion, the adapter-based approach offers a structured, reusable interface encapsulation that facilitates the exchange of data streams in standardized industrial protocols.

Conclusion

AULI_TO_AUS is a practical composite component that embeds an IEC 61131 conversion function in an adapter-based environment. It enables the transparent conversion of ULINT to USINT values via the corresponding adapters and is particularly suitable for scenarios where high accuracy or large value ranges need to be reduced to a smaller data format. The component is easy to use but requires attention to potential data loss.

AUS_TO_AULI





• – ○ • – ○ • – ○ •

Introduction

The function block `AUS_TO_AULI` is a composite block for converting an AUS adapter (UNSIGNED SHORT INTEGER) to an AULI adapter (UNSIGNED LONG INTEGER). It enables the direct forwarding of events and data between these two unidirectional adapter types, with an implicit type conversion from USINT to ULINT.

Interface Structure

Event Inputs

- **E1** (via `AUS_IN`): Event input of the connected AUS adapter. Passed directly to the output.

Event Outputs

- **E1** (via `AULI_OUT`): Event output of the AULI adapter. Corresponds to the incoming event of `AUS_IN.E1`.

Data Inputs

- **D1** (via `AUS_IN`): Data input of type **USINT** (OFF adapter). The value is transferred to the output.

Data Outputs

- **D1** (via `AULI_OUT`): Data output of type **ULINT** (AULI adapter). Contains the converted value of the input.

Adapters

- **OFF_IN** (Socket): Input interface for an OFF adapter (data type USINT, event E1, data D1).
- **AULI_OUT** (Plug): Output interface for an AULI adapter (data type ULINT, event E1, data D1).

Functionality

This function block directly connects the elements of the incoming OFF adapter with the corresponding elements of the outgoing AULI adapter:

- An incoming event at `AUS_IN.E1` is immediately passed on to `AULI_OUT.E1`.
- The data value `AUS_IN.D1` (USINT) is mapped to `AULI_OUT.D1` (ULINT) without any additional logic. The type conversion is implicit through the different adapter definitions; a USINT value is expanded to the corresponding ULINT value.

Technical Features

- **Numeric conversion**: value-preserving cast (narrowing may truncate, widening/sign-extension is safe).
- **Composite Function Block**: The function block does not contain its own ECC (Execution Control Chart) or state logic, but consists solely of an event and data connection.
- **Implicit Type Conversion**: The conversion from USINT to ULINT is provided by the adapter definitions in the `adapter::types::unidirectional` library. Explicit type conversion in the function block (FB) is not required.
- **Unidirectionality**: The FB only supports one direction of data flow (from AUS to AULI). Reverse conversion is not supported.

State Overview

The function block does not have its own state machine. It operates purely event-driven and combinatorially: With each incoming event, the current data value is immediately forwarded. No internal states are stored.

Application Scenarios

- **Adapter Integration**: If an existing module (e.g., a sensor) outputs an AUS adapter, but a subsequent module (e.g., a controller) expects an AULI adapter, this FB can be used as a conversion intermediary layer.
- **System Expansion**: Easy upgrade from USINT-based interfaces to ULINT without changing the surrounding logic.
- **Test Environments**: As a placeholder or bridge in adapter-based test structures.

Comparison with Similar Function Blocks

- **AUS_TO_AULI vs. Generic Type Converters**: Unlike a universal converter, this function block is specifically designed for AUS/AULI adapters and requires no additional parameterization. It is more compact and faster to develop.
- **AUS_TO_AULI vs. Manual Adapter Programming**: Instead of programming an adapter-to-adapter converter yourself, this ready-made function block offers an immediately usable, tested solution.

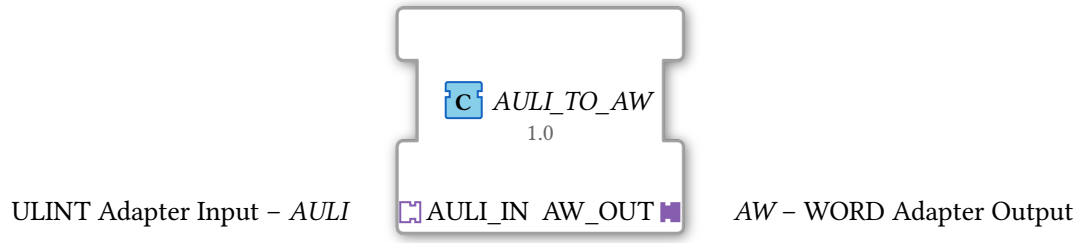
Conclusion

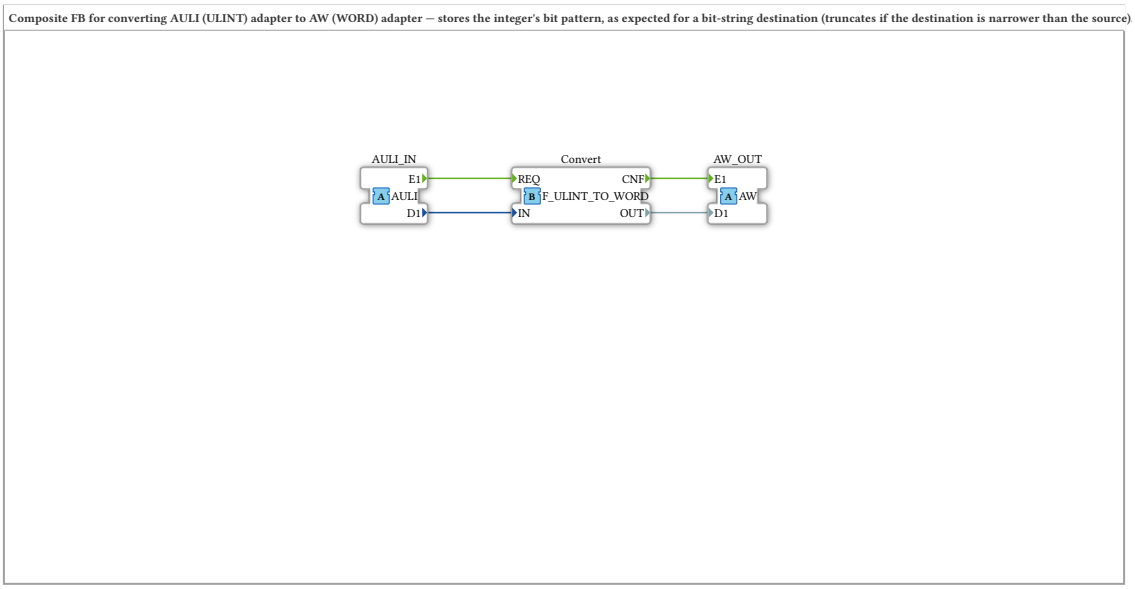
The FB `AUS_TO_AULI` is a simple yet effective composite function block for converting an AUS adapter (USINT) to an AULI adapter (ULINT). Through its direct passing of events and data, as well as implicit type conversion, it represents a clean and maintainable solution for the integration of different adapter interfaces.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AULI_TO_AW





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_TO_AW** is a composite function block (FB) that converts a ULINT adapter interface (AULI) to a WORD adapter interface (AW). It is used where a unidirectional data type conversion from unsigned 64-bit integer to 16-bit word is required.

Interface Structure

Event Inputs

Name	Type	Comment
E1 (via AULI_IN)	Event	Starts the conversion

Event Outputs

Name	Type	Comment
E1 (via AW_OUT)	Event	Signals completion of conversion

Data Inputs

Name	Type	Comment
D1 (via AULI_IN)	ULINT	ULINT input value

Data Outputs

Name	Type	Comment

D1 (via AW_OUT)	WORD	WORD output value

Adapters

Type	Role	Name	Comment
adapter::types::unidirectional::AULI	Socket	AULI_IN	ULINT Adapter Input
adapter::types::unidirectional::AW	Plug	AW_OUT	WORD Adapter Output

Functionality

The function block internally uses the IEC 61131 function **F_ULINT_TO_WORD** to convert the incoming ULINT value (via the AULI_IN adapter) into a WORD value. The process is purely event-driven:

1. An incoming event at the AULI_IN adapter (E1) triggers the conversion.
2. The ULINT value (D1) is passed to the function block **F_ULINT_TO_WORD**.
3. After the conversion is complete, the output event is sent at the AW_OUT adapter (E1), and the WORD value is available at the output data point (D1).

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Unidirectional Adapters**: Both AULI_IN and AW_OUT are unidirectional adapters, meaning data flow is only in one direction.
- **Composition**: The function block is built from a single integrated conversion function block, making it easy to understand and maintain.
- **Type Conversion**: The conversion from ULINT (64-bit) to WORD (16-bit) results in an overflow for values greater than 65535 – the user must ensure that the input value is in the range 0…65535.

State Overview

Since this is a pure composite function block without its own state machine, there are no internal states. All logic is implemented by the embedded function block **F_ULINT_TO_WORD**, which has simple, event-driven function execution.

Application Scenarios

- **Interface Adaptation**: When an AULI adapter (e.g., from a sensor unit) needs to be integrated into a system that only supports Word adapter (AW) interfaces.
- **Data Reduction**: Reducing the data width from 64 bits to 16 bits for subsequent processing steps.
- **Legacy Integration**: Integrating older components that only work with 16-bit data into a modern environment that provides ULINT.

Comparison with Similar Function Blocks

Comparable function blocks include **ULINT_TO_INT** and **ULINT_TO_DWORD**, which offer different target formats. **AULI_TO_AW** is specifically tailored to the unidirectional adapter types of the 4diac world. Unlike a simple type conversion, this function block also handles event and adapter linking, significantly simplifying integration into adapter-based architectures.

Conclusion

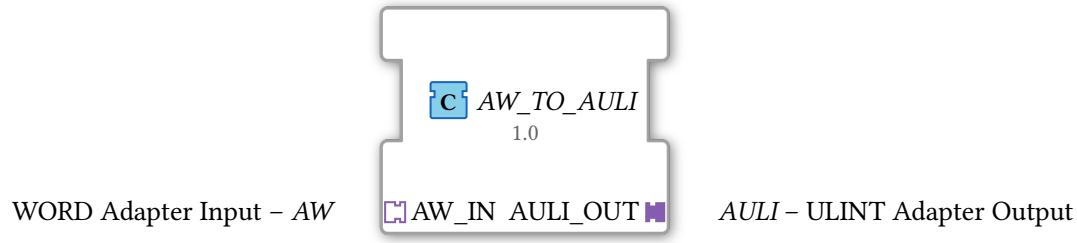
AULI_TO_AW is a practical, compact conversion function block that bridges the gap between ULINT and WORD adapter interfaces. By encapsulating the IEC function in a composite function block, the user is relieved of adapter details and receives a clean, event-driven transformation solution. This function block is particularly suitable for scenarios where data widths need to be reduced while maintaining the adapter structure.

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

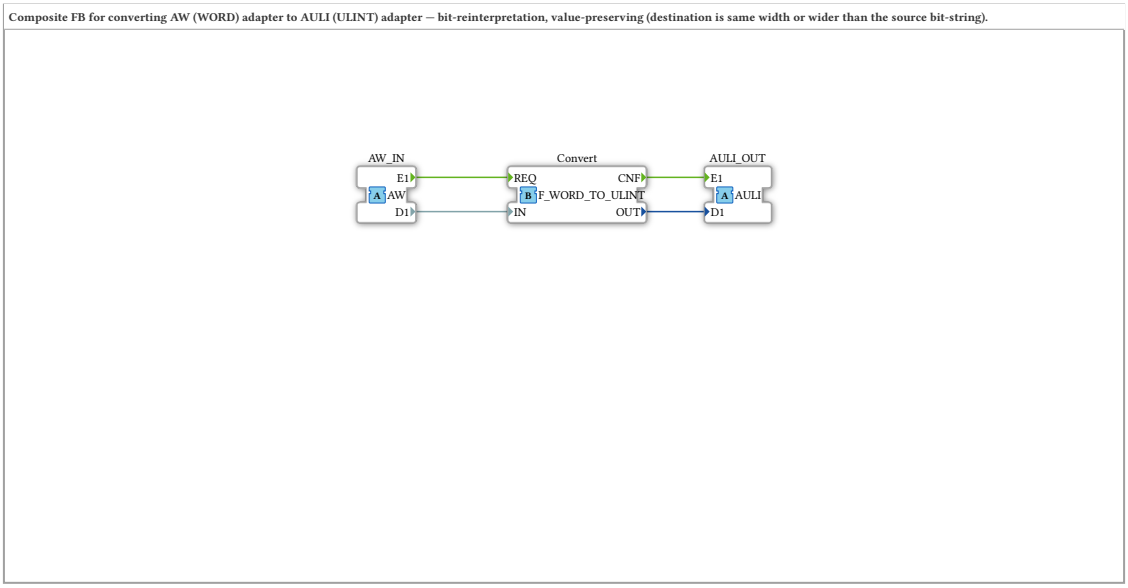
]

AW_TO_AULI



WORD Adapter Input – *AW*

AULI – ULINT Adapter Output



• – ◦ • – ◦ • – ◦ •

Introduction

The composite function block `AW_TO_AULI` converts a unidirectional adapter of type `AW` (WORD data format) into an adapter of type `AULI` (ULINT data format). The conversion is performed internally using the standard function block `F_WORD_TO_ULINT` from the IEC 61131 library.

Interface Structure

Event Inputs

- **AW_IN.E1** – Event input via socket `AW_IN`. Triggers the conversion.

Event Outputs

- **AULI_OUT.E1** – Event output via plug `AULI_OUT`. Signals the completion of the conversion.

Data Inputs

- **AW_IN.D1** – Data input of type WORD (via socket `AW_IN`).

Data Outputs

- **AULI_OUT.D1** – Data output of type ULINT (via plug `AULI_OUT`).

Adapter

- **Socket:** `AW_IN` (Type: `adapter::types::unidirectional::AW`)
- **Plug:** `AULI_OUT` (Type: `adapter::types::unidirectional::AULI`)

Functionality

As soon as an event arrives at event input `AW_IN.E1`, the internal function block `F_WORD_TO_ULINT` is called. This block reads the current value of the data input `AW_IN.D1` (WORD) and converts it into a ULINT

value. After successful conversion, an event is output at event output `AULI_OUT.E1`, and the converted value is made available at data output `AULI_OUT.D1`.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- The function block (FB) is implemented as a composite and fully encapsulates the conversion logic.
- The conversion function used, `F_WORD_TO_ULINT`, complies with the IEC 61131-3 standard.
- The interfaces are based on unidirectional adapters – event and data flow is one-way only.
- The FB does not have a state machine; its functionality is purely event-driven.

State Overview

The FB does not contain any explicit states. Processing is linear: Event ⟶ Conversion ⟶ Output.

Application Scenarios

- Integration of sensors or actuators that deliver WORD data into systems that expect ULINT values.
- Adapter-based protocol conversion in automation technology (e.g., agricultural engineering).
- Reusable encapsulation of the standard conversion function for modular control projects.

Comparison with Similar Function Blocks

- **Direct use of `F_WORD_TO_ULINT`:** Requires manual wiring of event and data lines. The composite function block `AW_TO_AULI` simplifies integration into adapter-oriented architectures.
- **Other conversion adapters** (e.g., `AW_TO_ADINT` for DINT or `AW_TO_ABOOL` for BOOL): Offer analogous conversions for other target data types.

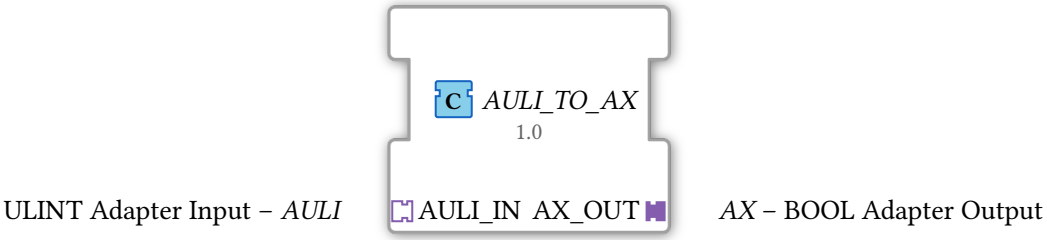
Conclusion

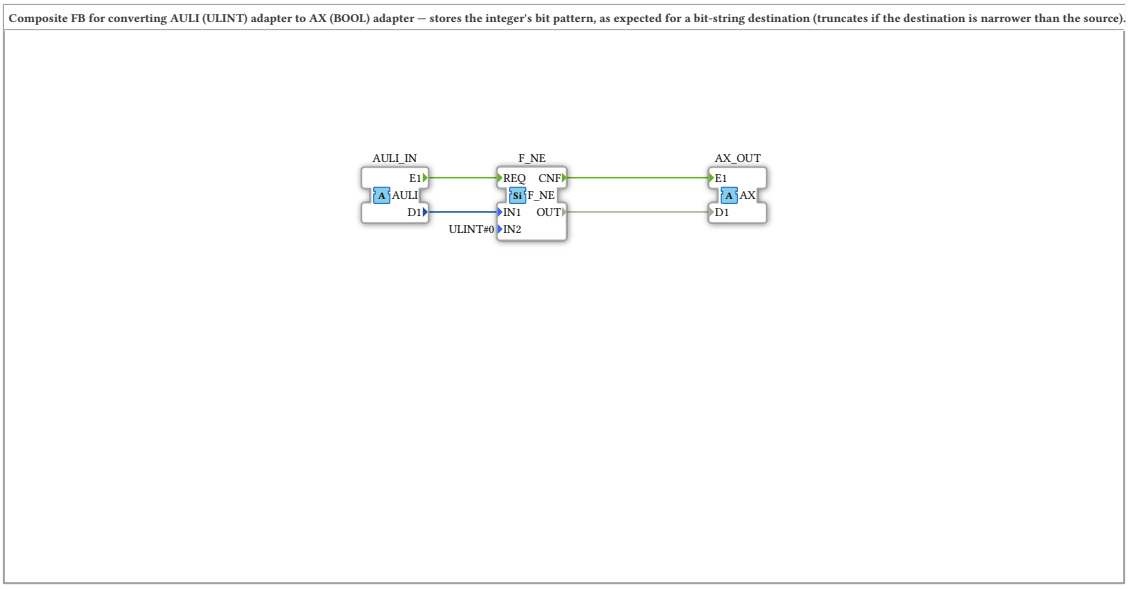
AW_TO_AULI` is a practical and easy-to-use component for converting WORD to ULINT data. The use of standardized adapter interfaces and the IEC conversion function ensures compatibility and reliability. The function block is particularly suitable for modular, adapter-based automation solutions.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AULI_TO_AX





• — ○ • — ○ • — ○ •

Introduction

The **AULI_TO_AX** function block serves as a composite block for converting an **AULI** adapter (data type ULINT) into an **AX** adapter (data type BOOL). It allows the connection of an unsigned integer from an adapter socket to a Boolean adapter plug, where the value 0 is interpreted as FALSE and any other value as TRUE.

Interface Structure

Event Inputs

The function block does not have direct event inputs. The triggering event is provided via the **AULI_IN** adapter:

- **AULI_IN.E1**: Event for processing the ULINT value and triggering the output on the AX adapter.

Event Outputs

Event output is also handled via the adapter:

- **AX_OUT.E1**: Acknowledgement event that, after successful conversion, forwards the new BOOL value to the connected receiver.

Data Inputs

All data inputs are part of the adapter socket:

- **AULI_IN.D1** (ULINT): The input value to be converted.

Data Outputs

Data output is handled via the adapter plug:

- **AX_OUT.D1** (BOOL): Conversion result – TRUE if the input value is not equal to 0, otherwise FALSE.

Adapter

- **AULI_IN** (Socket, Type: `adapter::types::unidirectional::AULI`): Receives the ULINT value and the associated event.
- **AX_OUT** (Plug, Type: `adapter::types::unidirectional::AX`): Provides the converted BOOL value and the output event.

Functionality

The function block (FB) operates purely event-driven and internally uses the comparison block **F_NE** (IEC 61131-3, “not equal”). An incoming event on **AULI_IN.E1** triggers the execution of the FB **F_NE**

by triggering its input **REQ**. The comparison block checks the ULINT value present at **IN1** against the constant value **ULINT#0** (parameter **IN2**). The result **OUT** (BOOL) is forwarded to the data output **AX_OUT.D1**. Once the comparison is complete, **F_NE** generates an acknowledgment event on **CNF**, which is passed to **AX_OUT.E1** to signal the availability of the new value to the connected receiver.

F_NE** is not a validation event.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Function Block**: Functionality is achieved through internal interconnection of standard function blocks – in this case, only a single comparison block.
- **Adapter Encapsulation**: Inputs and outputs are defined exclusively via adapters, enabling uniform and modular connectivity in adapter-based architectures.
- **Use of IEC 61131 Library**: The **F_NE** function block used is from the IEC 61131-3 library, promoting compatibility and reusability.
- **Constant Comparison Reference**: The limit value is fixed at 0; parameterization is not provided.

State Overview

As a composite function block, **AULI_TO_AX** does not have its own states in the sense of a state machine. The sequence is controlled purely by event chaining:

1. **Waiting** for an event at **AULI_IN.E1**.
2. **Processing** by **F_NE** (comparison) – quasi-static, no delay.
3. **Output** of the result to **AX_OUT** (event and data).

Application Scenarios

- **Limit Logic**: Converts a ULINT counter reading (e.g., quantity, fill level) into a simple Boolean signal indicating whether the value deviates from zero.
- **Adapter Conversion**: Replaces direct type conversion in adapter-based control architectures when only Boolean processing is required.
- **Signal Simplification**: Reduces complex numerical information to a binary state for alarm or status messages.

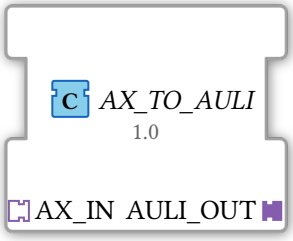
Comparison with Similar Function Blocks

Unlike general comparison function blocks (e.g., **GT**, **EQ**), **AULI_TO_AX** is specifically designed for adapter conversion and does not have configurable comparison operators or variable thresholds. A direct comparison with 0 is hard-coded. Function blocks like **ULINT_TO_BOOL** (without adapters) require their own event and data connections, whereas this function block integrates all adapter handling.

Conclusion

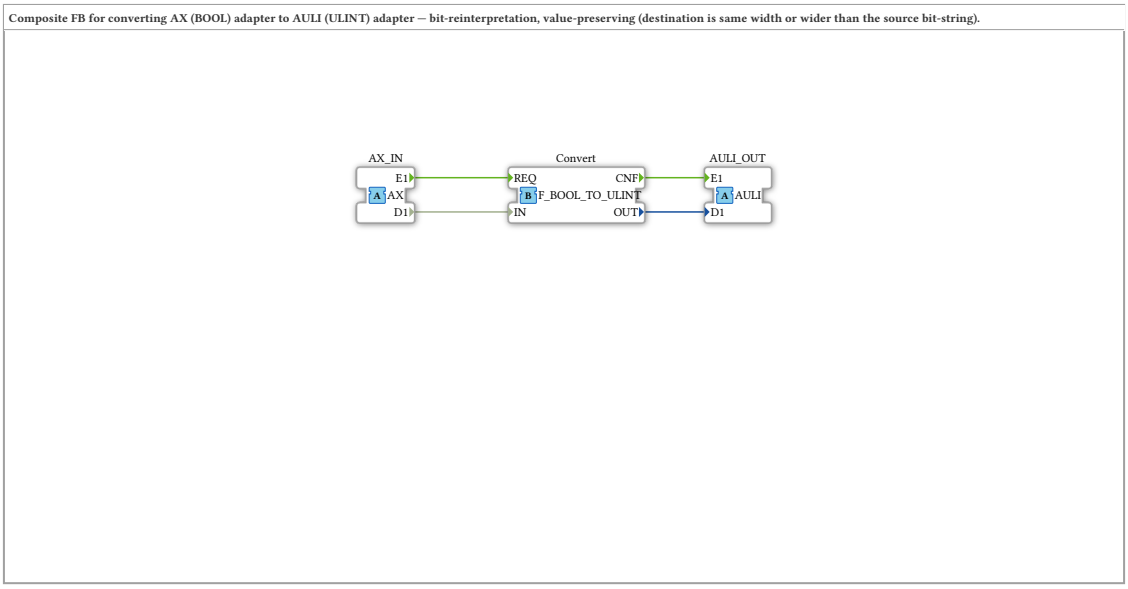
AULI_TO_AX is a compact and specialized composite function block for the simple conversion of a ULINT adapter signal to a Boolean adapter signal. It reduces wiring effort in adapter-based projects and ensures clear, maintainable structures when only the "non-zero" state needs to be evaluated. Implementation using a standardized comparison function block guarantees reliable and resource-efficient execution.

AX_TO_AULI



BOOL Adapter Input – *AX*

AULI – ULINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AX_TO_AULI** is a composite block for converting a BOOL adapter signal (type `AX`) into a ULINT adapter signal (type `AULI`). It encapsulates the type-safe conversion of a binary value into an unsigned 64-bit integer, thus enabling the easy integration of BOOL interfaces into ULINT-based control logic.

The function block is provided in the package `adapter::conversion::unidirectional` and is based on the standardized conversion function block `F_BOOL_TO_ULINT` from the IEC 61131 library.

Interface Structure

Event Inputs

No dedicated event inputs.

Event control is handled via the connected adapters (see section **Adapters**).

Event Outputs

No dedicated event outputs.

Result events are provided via the output adapter (see section **Adapters**).

Data Inputs

No dedicated data inputs.

The data to be converted is received via the input adapter (see section **Adapters**).

Data Outputs

No dedicated data outputs.

The converted result is provided via the output adapter (see section **Adapters**).

No dedicated data outputs.

Adapter

Direction	Name	Type	Description
Socket	`AX_IN`	`adapter::types::unidirectional::AX`	BOOL adapter input. Provides input data via the event `E1` and the date `D1` (BOOL value).
Plug	`AULI_OUT`	`adapter::types::unidirectional::AULI`	ULINT adapter output. Outputs the conversion result via the event `E1` and the date `D1` (ULINT value).

Functionality

The function block contains an internal network consisting of a single instance of the conversion function block `F_BOOL_TO_ULINT`.

The connections are as follows:

1. **Event Handling**:

- The event `AX_IN.E1` (from the input adapter) is passed to the event input `Convert.REQ` and starts the conversion.
- After the conversion is complete, the event `Convert.CNF` is passed to the event output `AULI_OUT.E1` to finally update the output adapter.

1. **Data Flow**:

- The Boolean value of `AX_IN.D1` is mirrored to the data input `Convert.IN`.
- The converted ULINT value of `Convert.OUT` is transferred to `AULI_OUT.D1`.

Thus, on each rising edge of the input event, the current BOOL value is converted to a ULINT value (BOOL $\longrightarrow$ 0 $\longrightarrow$ 0 ULINT, BOOL $\longrightarrow$ 1 $\longrightarrow$ 1 ULINT) and made available via the output adapter.

Technical Specifications

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **License**: The source code is licensed under the Eclipse Public License 2.0 (EPL-2.0).
- **Author**: Developed by **Franz Höpfinger** for **HR Agrartechnik GmbH** (Version 1.0, dated 2026-02-17).
- **Package Structure**: The module is organized in the package `adapter::conversion::unidirectional`.
- **Dependencies**: It imports the module `iec61131::conversion::F_BOOL_TO_ULINT`.
- **Type Safety**: The use of adapters ensures strict type checking at design time.

State Overview

The composite module does not have its own state machine.

The entire sequence control is implemented by the internal module `F_BOOL_TO_ULINT`, which implements a simple event-driven state machine:

- In its idle state, it waits for a `REQ` event.
- Upon arrival of the event, the conversion is performed and the output is immediately confirmed with `CNF`.

Application Scenarios

- **Integration of binary sensors** into controllers that expect ULINT values (e.g., counter registers, addressing logic).
- **Adapter bridging** in heterogeneous systems where a BOOL adapter (e.g., from a switch) needs to be adapted to a ULINT adapter (e.g., a profile/parameter interface).
- **Type-safe conversion** in composite function blocks to standardize the handling of different data types in an application.

Comparison with Similar Function Blocks

- **F_BOOL_TO_ULINT**: The underlying basic function block – it offers pure conversion without adapter encapsulation.
- **AX_TO_UDINT**, **AX_TO_DINT**: Analogously structured composite function blocks for other target data types (UDINT, DINT).
- **Direct Type Conversion**: Numerous standard conversion function blocks exist in the IEC 61131 environment; this one specializes in the adapter interface.

Conclusion

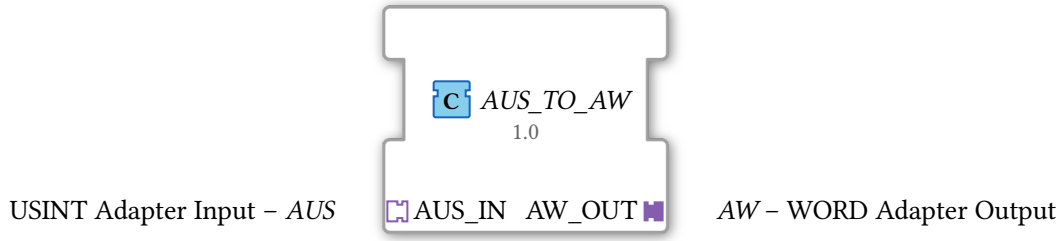
The **AX_TO_AULI** function block is a compact, type-safe, and easily reusable solution for converting a BOOL adapter to a ULINT adapter.

It simplifies the hierarchical structuring of applications and enables a clear separation between data sources and sinks. Thanks to its EPL 2.0 licensing, it can be freely integrated into custom projects.

Numeric Index (unidirectional)

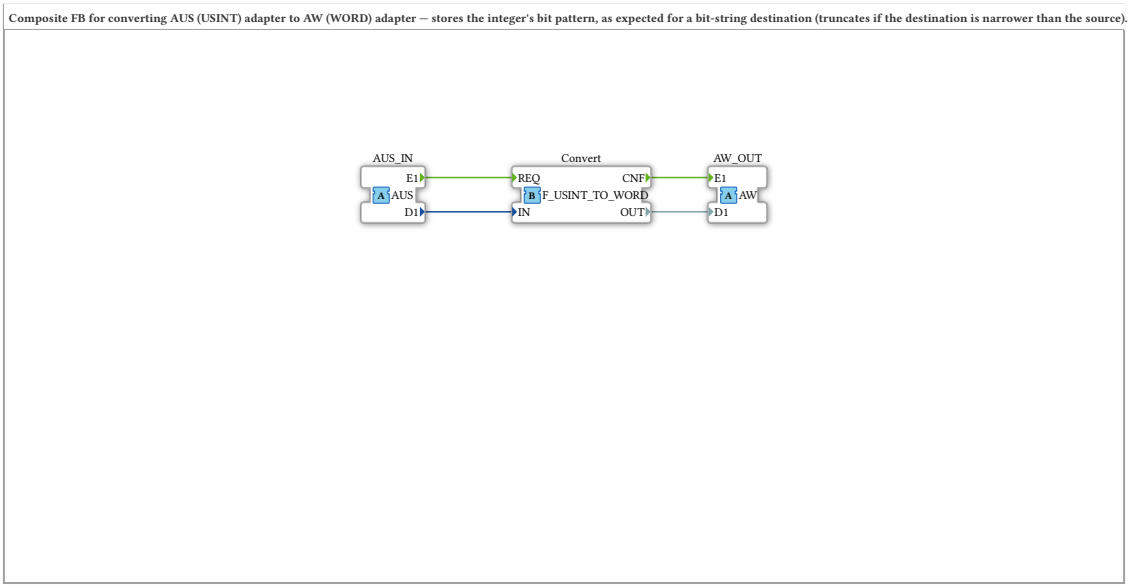
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUS_TO_AW



USINT Adapter Input – *AUS*

AW – WORD Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AUS_TO_AW** function block is a composite function block that enables the simple and standardized conversion of a **USINT** value (0–255) to a **WORD** value (16 bits) via adapter interfaces. It acts as an intermediary between systems that provide or expect different data types via unidirectional adapters.

Interface Structure

The function block itself has no direct inputs or outputs. All communication takes place via the two integrated adapters.

Event Inputs

No direct event inputs are available. Event control is implemented via the **AUS_IN** adapter. This provides an event input **E1**, the activation of which starts the conversion process.

Event Outputs

No direct event outputs are available. An event is output via the **AW_OUT** adapter after successful conversion (event output **E1**).

Data Inputs

No direct data inputs are available. The USINT value to be converted is read via the **D1** data input of the **AUS_IN** adapter (type: USINT).

Data Outputs

No direct data outputs are available. The converted WORD value is output via the **D1** data output of the **AW_OUT** adapter (type: WORD).

Adapters

- **AUS_IN** (Socket): Unidirectional adapter of type `adapter::types::unidirectional::AUS`. Expects a USINT value and an event to trigger the conversion.
- **AW_OUT** (Plug): Unidirectional adapter of type `adapter::types::unidirectional::AW`. Provides the WORD value and a confirmation event after successful conversion.

Functionality

1. An external event at the **E1** input of the **AUS_IN** adapter triggers execution.
2. The internal function block **F_USINT_TO_WORD** (from the IEC 61131 library) takes the USINT value present at **AUS_IN.D1** and converts it to a WORD value.
3. After the conversion is complete, the WORD value at the **OUT** output of **F_USINT_TO_WORD** is forwarded to **AW_OUT.D1**.
4. Simultaneously, an event is generated at the **CNF** output of **F_USINT_TO_WORD**, which signals externally via **AW_OUT.E1** that the converted value is available.

Technical Features

- **Stores the integer's bit pattern**, as expected for a bit-string destination (truncates if the destination is narrower than the source).
- **Composite Block**: The conversion logic is encapsulated in its own network, promoting reuse and clarity.
- **Unidirectional Adapters**: The **OFF** and **AW** adapters are designed as pure data and event pipelines – reverse communication is not supported.
- **Standard Conversion**: The actual type conversion is based on the tested block `F_USINT_TO_WORD` from the IEC 61131 library.
- **No Internal State Logic**: The block delegates all control to the internal converter and the adapter interfaces.

State Overview

The block does not have its own state machine. The process is event-driven and strictly sequential:

- **Idle**: Waiting for an event at the OFF_IN adapter.
- **Conversion active**: Internal **F_USINT_TO_WORD** is executed. Upon completion, the function block automatically returns to the idle state.
- **Output**: The confirmation event is sent via the AW_OUT adapter.

Application Scenarios

- **System Integration**: A sensor delivers measured values as USINT via a unidirectional AUS adapter, but the controller expects WORD values via an AW adapter.
- **Data Preparation**: Simple type conversion without additional logic is required in an adapter-based communication chain.
- **Prototype Development**: Rapid coupling of two components using different data types without modifying the existing adapter structure.

Comparison with Similar Function Blocks

- **F_USINT_TO_WORD** (direct): Provides pure conversion without adapter integration – suitable for internal flows within a device.
- **AUS_TO_AW**: Extends the conversion process with unidirectional adapters, enabling loose coupling between independent systems.
- **Other adapter converters (e.g., INT_TO_DWORD)**: Follow the same pattern but with different target types. **AUS_TO_AW** is specifically optimized for the USINT → WORD pairing.

Conclusion

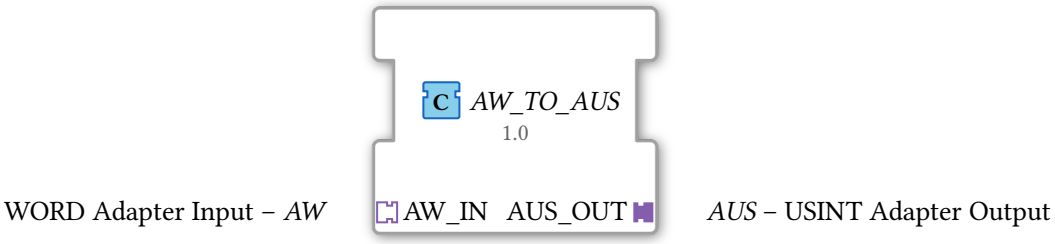
AUS_TO_AW is a compact, reliable composite component that standardizes the conversion of USINT to WORD data via unidirectional adapters and can be seamlessly integrated into existing 4diac projects. It reduces development effort and increases reusability in modular automation systems.

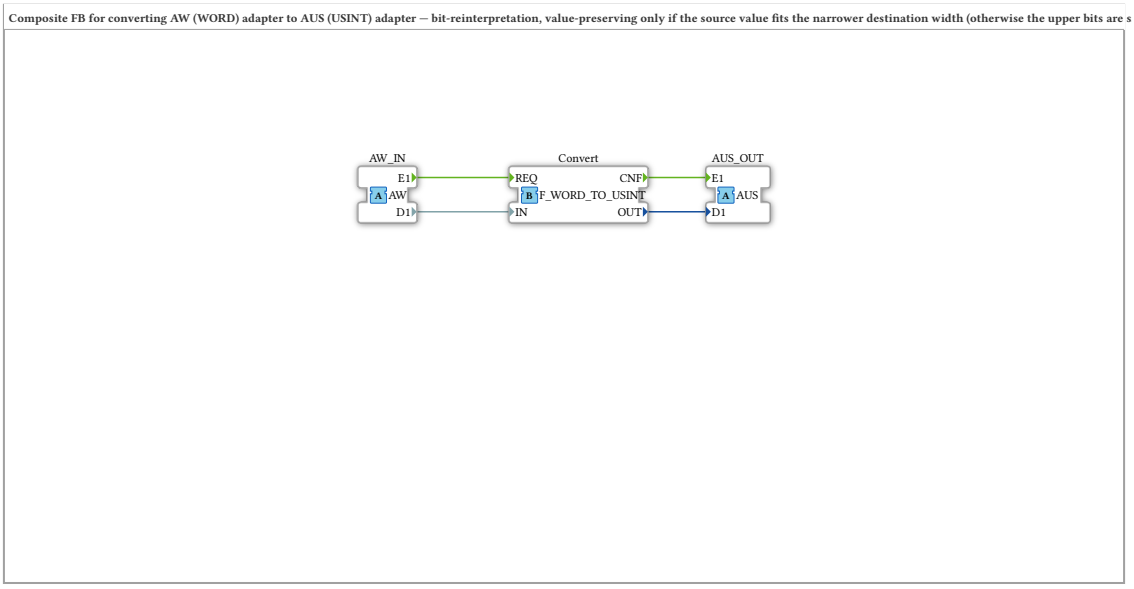
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AW_TO_AUS





• – ◦ • – ◦ • – ◦ •

Introduction

The composite function block `AW_TO_AUS` converts a `WORD` adapter signal into a `USINT` adapter signal. It encapsulates the IEC 61131-3 standard function `F_WORD_TO_USINT` in an adapter-coupled environment, thus enabling seamless integration between different adapter types.

Interface Structure

The function block has no independent event or data ports at the function block interface level. All communication takes place exclusively via the two adapters.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	AW_IN	adapter::types::unidirectional::AW	Expects a WORD value via data port D1 and an event via E1.
Plug (Output)	AUS_OUT	adapter::types::unidirectional::AUS	Returns the converted USINT value via data port D1 and an acknowledgment event via E1.

Functionality

1. An incoming event at the adapter socket AW_IN.E1 triggers the internal conversion module Convert (type F_WORD_TO_USINT) via its event input REQ.
2. Simultaneously, the data value from AW_IN.D1 (type WORD) is forwarded to the data input Convert.IN.
3. After conversion, the result (USINT) is provided at data output Convert.OUT and transferred to data port AUS_OUT.D1.
4. The conversion block signals completion via its event CNF, which directly activates the event output AUS_OUT.E1.

This results in a synchronous, event-driven conversion from a WORD value to a USINT value.

Technical Features

- **Bit-reinterpretation**, value-preserving only if the source value fits the narrower destination width (otherwise the upper bits are silently dropped).
- **Composite Pattern**: The block is implemented as a composite function block (FB) that encapsulates the existing standard function F_WORD_TO_USINT.
- **Adapter Coupling**: The function block simplifies integration into existing systems that rely on the adapters AW and AUS by consolidating the conversion into a single block.
- **No Own State Logic**: All logic is delegated; the function block itself has no integrated event control (ECC) and behaves purely directionally.

State Overview

The function block does not have its own state machine. The sequence is entirely determined by the embedded function F_WORD_TO_USINT and the event connections. Therefore, the function block should be considered **stateless**.

Application Scenarios

- **Adapter Conversion**: When a 4diac application uses an adapter of type AW (with WORD data), but a subsequent component expects an adapter of type AUS (with USINT data).
- **Protocol Adaptation**: Simplifies the connection of components that use different data types via their adapter interfaces.
- **Minimal Conversion Overhead**: Ideal when no additional logic is required and only a simple type conversion (e.g., from a 16-bit value to an 8-bit value) needs to be performed.

Comparison with Similar Function Blocks

- **Direct Use of F_WORD_TO_USINT**: This standard function block operates at the data level but requires manual event and data connections. AW_TO_AUS encapsulates these connections and provides an adapter-based interface.
- **Other Adapter Converters (e.g., AW_TO_ABOOL)**: These differ in their target data type. The presented scheme (Socket -> Conversion Function Block -> Plug) can be implemented analogously for any data type.

Conclusion

AW_TO_AUSis a simple yet useful composite function block for convertingWORDadapter signals toUSINT` adapter signals. It utilizes a proven IEC 61131-3 standard function and offers a clean, adapter-based interface. The function block requires no configuration and integrates seamlessly

into event-driven processes. It is particularly suitable for applications where fast and reliable data type conversion without additional logic is desired.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

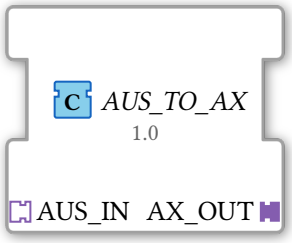
]

Numeric Index (unidirectional)

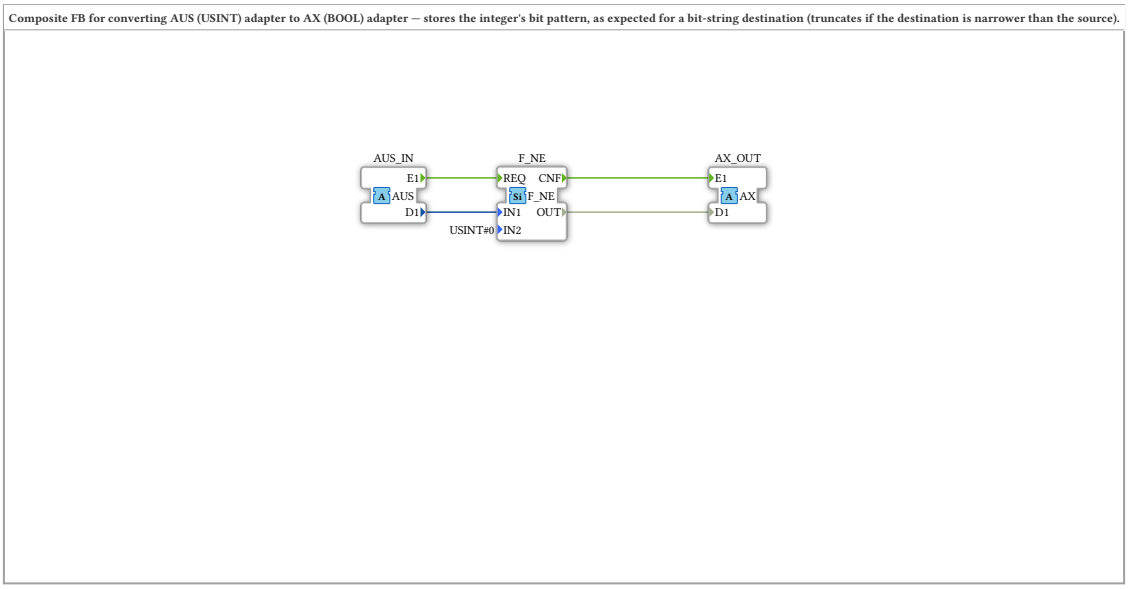
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUS_TO_AX

USINT Adapter Input – *AUS*



AX – BOOL Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_TO_AX** is a composite FB for converting an **AUS** adapter (USINT data) into an **AX** adapter (BOOL data). It compares the incoming USINT value with zero and outputs the result as a Boolean signal. This enables the simple conversion of counter or sensor data into a binary status signal.

Interface Structure

The FB does not offer direct event or data connections, but only adapter interfaces.

Event Inputs

No direct event inputs. The event signal is provided via the **AUS_IN.E1** adapter.

Event Outputs

No direct event outputs. The event signal is output via the **AX_OUT.E1** adapter.

Data Inputs

No direct data inputs. The data signal (USINT) is provided via the adapter **AUS_IN.D1**.

Data Outputs

No direct data outputs. The data signal (BOOL) is output via the adapter **AX_OUT.D1**.

Adapters

Name	Direction	Type	Comment
AUS_IN	Socket	adapter::types::unidirectional::AUS	USINT input value and event
AX_OUT	Plug	adapter::types::unidirectional::AX	BOOL output value and event

Functionality

The function block internally uses an **F_NE** block (comparison for inequality) from the IEC 61131 library.

- An incoming event **AUS_IN.E1** triggers the **REQ** input of **F_NE**.
- **F_NE** compares the data value **AUS_IN.D1** (USINT) with the constant value **USINT#0**.
- The comparison result (BOOL: TRUE if not zero, otherwise FALSE) is forwarded to **AX_OUT.D1**.
- After processing is complete, **F_NE** generates a **CNF** event, which is passed to **AX_OUT.E1**.

Technical Features

- **Not-equal-to-zero comparison, not a bit pattern**: The result is `TRUE` for any value other than 0, not the least significant bit. The comment in the source block ("stores the bit pattern … truncates") describes different behaviour and is outdated; the `F_NE` network is authoritative.
- The function block (FB) is implemented as a **Composite FB**, meaning its logic is represented by an internal network of standard function blocks.
- It uses the **F_NE** comparison block, which is from the **iec61131::comparison** library.
- The package organization is **adapter::conversion::unidirectional**, indicating its use as an adapter converter.
- The function block (FB) does not have its own state machines; processing is exclusively event-driven via internal connections.

State Overview

As a composite FB, **AUS_TO_AX** does not have its own state machine.
All logic is performed by the integrated **F_NE**, which does not require stateful control.
The behavior is purely combinatorial with event-driven execution.

Application Scenarios

- Conversion of a counter value (USINT) into a Boolean signal indicating the state "value ≠ 0" (e.g., "fill level present" or "sensor active").
- Use in **agricultural technology** – the FB originates from HR Agrartechnik GmbH – for processing sensor data via adapter interfaces.
- Simplifies cabling in 4diac IDE projects when heterogeneous adapter types need to be coordinated.

Comparison with similar function blocks

- Direct comparison with a **NE** function block: **AUS_TO_AX** extends the basic comparison function with adapter conversion.
- While an **NE** function block typically expects two simple data inputs (e.g., ANY), this function block works exclusively with predefined adapters (AUS / AX) and hides its internal logic.
- It is therefore specifically optimized for systems with a uniform adapter concept (unidirectional interfaces).

Conclusion

AUS_TO_AX is a simple yet effective adapter converter that converts a USINT value into a Boolean signal by comparing it to zero.

Its encapsulation as a composite function block with adapter connections allows for seamless integration into modular 4diac projects. The function block (FB) does not offer its own states and, due to its low complexity, is reliable and requires little maintenance.

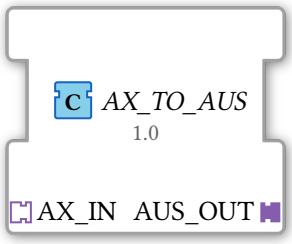
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

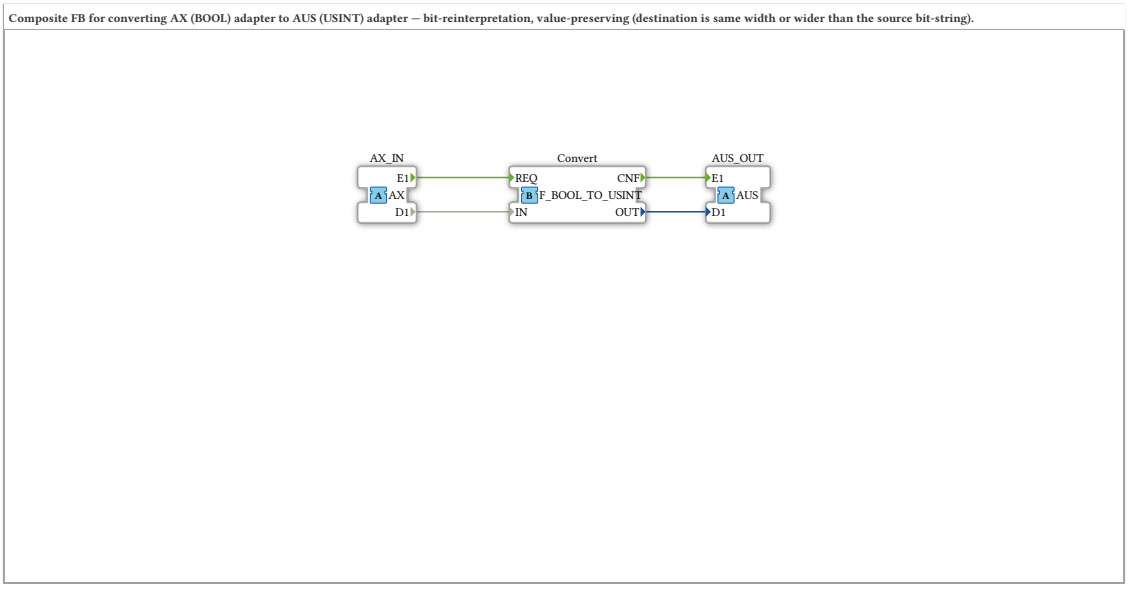
]

AX_TO_OFF

BOOL Adapter Input – *AX*



AUS – USINT Adapter Output



• – ○ • – ○ • – ○ •

Introduction

The **AX_TO_OFF** function block is a composite function block that converts a Boolean input adapter (type `AX`) into an integer output adapter (type `AUS`). It serves as a bridge between components that operate with different data types and enables seamless integration into control applications according to IEC 61499.

Interface Structure

Event Inputs

The function block does not have direct event inputs; however, the incoming adapter `AX_IN` provides the event `E1`. This triggers the conversion.

Event Outputs

There are no direct event outputs here either. The outgoing adapter `AUS_OUT` provides the event `E1`, which is triggered after successful conversion.

Data Inputs

Data is received via the adapter `AX_IN` as a Boolean value (`D1` of type `BOOL`).

Data Outputs

The converted value is output via the adapter `AUS_OUT` as an unsigned 8-bit integer (`D1` of type `USINT`).

Adapter

Direction	Name	Type	Description
Socket	AX_IN	adapter::types::unidirectional::AX	Boolean Input Adapter
Plug	AUS_OUT	adapter::types::unidirectional::AUS	USINT Output Adapter

Functionality

Internally, the function block uses the predefined conversion function block iec61131::conversion::F_BOOL_TO_USINT. The process is as follows:

1. An incoming event at the adapter AX_IN.E1 triggers the internal function block via its input REQ.
2. The Boolean value of AX_IN.D1 is forwarded to the input IN of the internal function block.
3. The internal function block converts the value of BOOL into a value of USINT (false ⟶ 0, true ⟶ 1) and places it at its output OUT.
4. The confirmation event CNF of the internal function block is forwarded to the adapter output AUS_OUT.E1.
5. The converted value from output OUT is transferred to the data output AUS_OUT.D1.

Technical Features

- **Bit-reinterpretation, value-preserving** (destination is same width or wider than the source bit-string).
- **Composite Function Block**: The function block encapsulates the conversion logic and provides a clean, reusable interface via adapters.
- **Plug/Socket Coupling**: The function block uses the adapter pattern (socket for input, plug for output), enabling flexible connectivity in an IEC 61499 application.
- **Dependency**: It imports the function block iec61131::conversion::F_BOOL_TO_USINT, which is part of the IEC 61131 conversion library.

State Overview

The function block does not have its own state machine. All functionality is implemented by the internal F_BOOL_TO_USINT function block, which operates in an event-driven manner and does not store internal states.

Application Scenarios

- **Signal Adaptation**: When a sensor or a Boolean adapter (e.g., switch, limit switch) needs to be connected to a component that expects an integer adapter.
- **Protocol Conversion**: In heterogeneous systems that use different data formats, this function block can serve as a simple converter.
- **Test and Simulation Environments**: To convert Boolean values into numerical values that can be more easily processed by analysis tools.

Comparison with Similar Function Blocks

Numerous conversion blocks exist in the IEC 61499 library, such as F_BOOL_TO_INT, F_BOOL_TO_WORD, and F_BOOL_TO_BYTE. The difference with AX_TO_AUS lies in the adapter interface: This block encapsulates the conversion not only at the data level but also at the event level in a compact adapter pair. Other blocks typically work with simple input/output data and require additional event wiring.

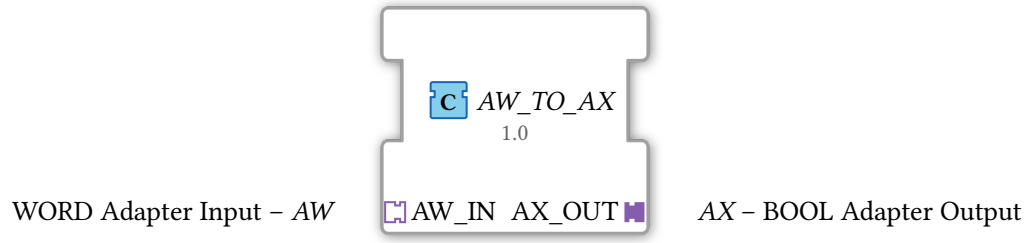
Conclusion

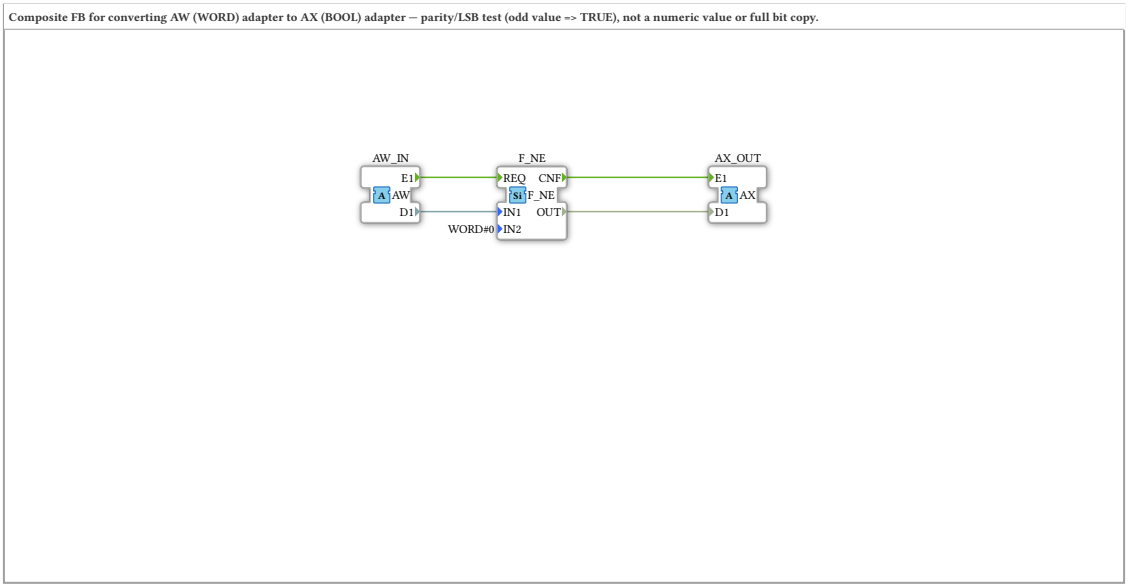
The **AX_TO_AUS** function block is a practical tool for converting Boolean adapter signals to the USINT format. By using adapters, it integrates seamlessly into the event-driven architecture according to IEC 61499 and significantly reduces wiring effort. It is particularly suitable for applications where a clean separation between the Boolean and numeric domains is desired.

Numeric Index (unidirectional)

Here you will find all exercises from the unidirectional workspace, listed in ascending order. …

AW_TO_AX





• – ○ • – ○ • – ○ •

Introduction

The composite function block **AW_TO_AX** converts an **AW adapter** (WORD) into an **AX adapter** (BOOL). It checks whether the 16-bit value received via the AW adapter is non-zero and outputs the result as a Boolean signal via the AX adapter.

Interface Structure

This function block does not have traditional event or data inputs/outputs, but only adapter interfaces.

Event Inputs

No separate event inputs. Events are received via the socket adapter.

Event Outputs

No separate event outputs. Events are sent via the plug adapter.

Data Inputs

No separate data inputs. Data is received via the socket adapter.

Data Outputs

No separate data outputs. Data is sent via the plug adapter.

Adapters

Name	Type	Direction	Comment
AW_IN	adapter::types::unidirectional::AW	Socket	WORD adapter input
AX_OUT	adapter::types::unidirectional::AX	Plug	BOOL adapter output

The socket **AW_IN** provides the event and data interface of a **WORD**-based adapter.

The plug **AX_OUT** provides the event and data interface of a **BOOL**-based adapter.

Functionality

When an event arrives at **AW_IN.E1**, the value of **AW_IN.D1** is compared to the constant value **WORD#0**. Internally, the function block **F_NE** (non-equal comparison from IEC 61131) is used for this comparison.

- If **AW_IN.D1 ≠ 0**, an event is sent to **AX_OUT.E1** and the data value **AX_OUT.D1** is set to **TRUE**.
- Otherwise, no event is generated at the output and the data value remains **FALSE** (i.e., it is not changed).

This behavior is event-driven and follows the IEC 61499 execution semantics: An event at the input triggers the comparison, the result of which is passed on to the output.

- If **AW_IN.D1 ≠ 0**, an event is sent to **AX_OUT.E1** and the data value **AX_OUT.D1** is set to **TRUE**.
-

Technical Features

- **Parity/LSB test** (odd value => TRUE), not a numeric value or full bit copy.
- The function block is implemented as a **composite function block**, meaning its functionality is represented by an internal network consisting of a single **F_NE** function block.
- It uses a **unidirectional adapter** (AW and AX) according to the adapter definition of the 4diac IDE.
- The comparison logic is taken from the IEC 61131 library (`iec61131::comparison::F_NE`).
- It is a pure signal conversion without any state management.

State Overview

The function block has **no internal state machine** (no ECC). The sequence control is derived directly from the internal network: An event at the input adapter triggers the comparison block, which in turn passes the result to the output adapter.

Application Scenarios

- **Signal Conditioning:** Conversion of a digital WORD value (e.g., from a sensor module) into a Boolean signal that indicates whether the value is non-zero.
- **Adapter Integration:** Use in systems that require adapter-based communication between different data types (e.g., linking a WORD-based bus component with BOOL-based control logic).
- **Threshold Check:** By changing the comparison value (in the internal block **F_NE**), the function block could also be adapted to other threshold values (e.g., “Value > 100”).

Comparison with Similar Blocks

- **WORD_TO_BOOL:** Simple, direct conversion of a WORD data type to BOOL (e.g., bit 0). In contrast, **AW_TO_AX** uses adapters and performs a comparison with zero – it reacts to the entire word.
- **Unequal Comparator (F_NE):** The **AW_TO_AX** encapsulates the **F_NE** block and adds the adapter interfaces, enabling its use in adapter-based architectures.

Conclusion

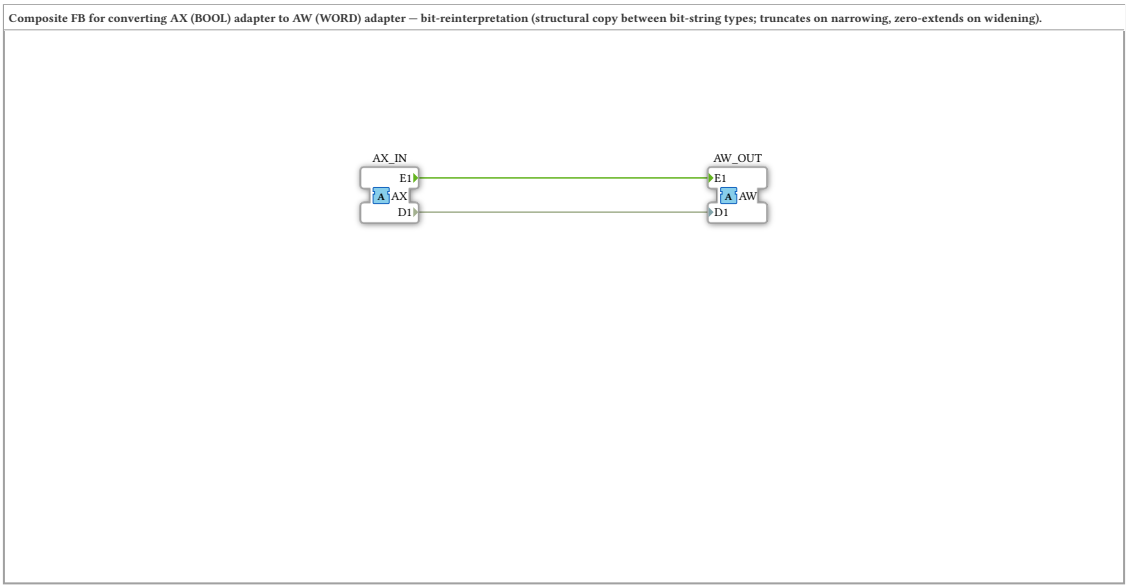
The **AW_TO_AX** is a compact yet useful composite block for converting Word adapters to Bool adapters. Its simple comparison principle makes it ideal for integration into event-driven automation systems based on adapter interfaces. The clear separation of input and output adapters facilitates modular design and reusability.

–

🌐 Related topic subpages on ms-muc-docs.de

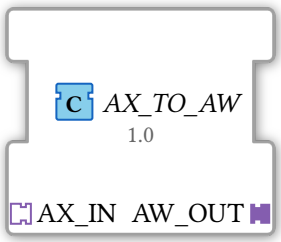
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AX_TO_AW



No image available.

BOOL Adapter Input – *AX*



AW – WORD Adapter Output

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AX_TO_AW` is a composite function block (FB) for converting a unidirectional BOOL adapter (type `AX`) into a unidirectional WORD adapter (type `AW`). It acts as an interface converter, forwarding an incoming event and a BOOL data value directly to the output adapter, with the BOOL value being implicitly converted into a WORD.

Interface Structure

Event Inputs

The function block does not have separate event inputs. The event is provided via the socket adapter `AX_IN` (type `AX`) (internal event `E1`).

Event Outputs

The function block does not have separate event outputs. The event is output via the plug adapter `AW_OUT` (type `AW`) (internal event `E1`).

Data Inputs

The function block does not have separate data inputs. The input data (BOOL) is provided via the socket adapter `AX_IN` (data point `D1`, type `BOOL`).

Data Outputs

This function block does not have separate data outputs. The output data (WORD) is provided via the plug adapter `AW_OUT` (data point `D1`, type `WORD`).

Adapter

Direction	Label	Type	Comment
Socket	AX_IN	adapter::types::unidirectional::AX	BOOL adapter input (consisting of event E1 and data D1:BOOL)
Plug	AW_OUT	adapter::types::unidirectional::AW	WORD adapter output (consisting of event E1 and data D1:WORD)

Functionality

This component forwards the event (E1) received from socket AX_IN directly to plug AW_OUT (E1). Simultaneously, the incoming BOOL value (AX_IN.D1) is passed to the WORD output (AW_OUT.D1) without further processing. An implicit type conversion takes place from BOOL to WORD (typical mapping: FALSE ⟶ 0, TRUE ⟶ 1). No logic, delay, or state change occurs – the function block (FB) functions purely as a pass-through.

Technical Features

- **Bit-reinterpretation** (structural copy between bit-string types; truncates on narrowing, zero-extends on widening).
- **Composite FB:** The function block does not have its own state machine (ECC) but consists solely of wiring the included adapter interfaces.
- **Type Conversion:** The conversion from BOOL to WORD occurs automatically via the data connection between the adapters. No explicit conversion logic is implemented.
- **Performance:** Direct coupling eliminates the need for additional processing steps, ensuring minimal latency.
- **Reusability:** This function block can be used in projects that require connecting adapters of different types without needing separate conversion logic.

State Overview

Since this is a composite function block without its own ECC, there are no explicit states. The function block behaves like a simple signal forwarder and has no behavioral model of its own.

Application Scenarios

- **Connecting a BOOL-based sensor** to a control system that expects WORD interfaces (e.g., digital inputs as bitmasks).
- **Adapter layer** in libraries where components with different adapter types need to be combined without modifying the source function blocks.
- **Prototypical development** for the rapid integration of BOOL and WORD adapters prior to later optimization.

Comparison with Similar Blocks

- **AW_TO_AX (theoretical):** This would be the inverse conversion from WORD to BOOL – requiring a reduction of the value range (e.g., only bit 0 is evaluated).
- **TYPCONV Blocks:** IEC 61499 libraries typically offer explicit type converters (e.g., BOOL_TO_WORD), but these operate as standard function blocks with event and data inputs, not as adapter converters.
- **Direct Adapter Coupling:** Without AX_TO_AW, the adapter types would have to be identical, or a separate conversion would be required at the network layer.

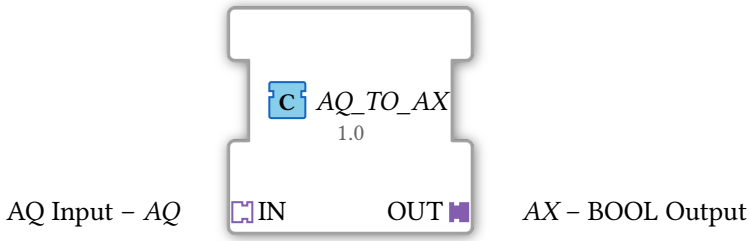
Conclusion

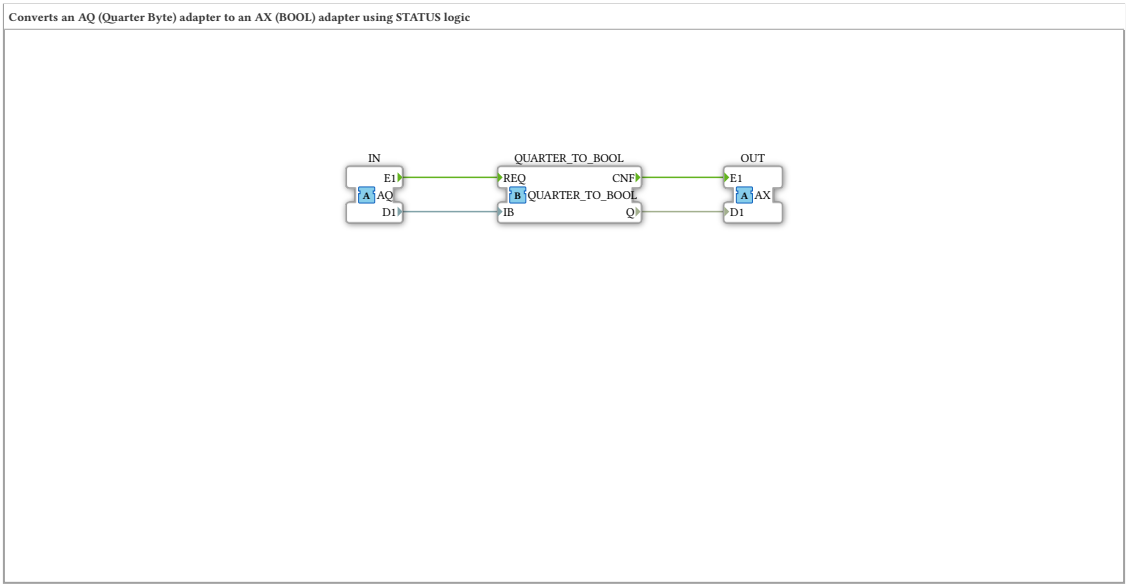
The function block AX_TO_AW is a simple yet useful converter that enables coupling between BOOL and WORD adapters. Thanks to the direct pass-through of events and data with minimal overhead, it is ideally suited for interface adaptations in distributed automation systems. Its composite structure makes it transparent, easy to understand, and simple to integrate into existing projects.

Numeric Index (unidirectional)

Here you will find all exercises from the unidirectional workspace, listed in ascending order. …

AQ_TO_AX





• – ○ • – ○ • – ○ •

Introduction

The function block `AQ_TO_AX` converts an AQ adapter (Quarter Byte) to an AX adapter (BOOL) using STATUS logic. It enables the easy integration of AQ-based signals into systems that operate exclusively with AX interfaces. Internally, it uses the block `QUARTER_TO_BOOL` to perform the data conversion.

Interface Structure

Event Inputs

- No direct event inputs are available. Events are received via the AQ adapter (socket `IN`).

Event Outputs

- No direct event outputs are available. Events are sent via the AX adapter (plug `OUT`).

Data Inputs

- No direct data inputs are available. Data is received via the AQ adapter (socket `IN`).

Data Outputs

- No direct data outputs are available. Data is sent via the AX adapter (plug `OUT`).

Adapter

- **`IN` (Socket)**: Type `adapter::types::unidirectional::AQ` – AQ input (Quarter Byte)
- **`OUT` (Plug)**: Type `adapter::types::unidirectional::AX` – BOOL output (AX)

Functionality

The `AQ_TO_AX` module does not have its own event or data inputs/outputs, but performs the conversion exclusively via two adapter interfaces. Internally, an instance of the function block `QUARTER_TO_BOOL` is used:

1. **Event Forwarding**: The event `E1` of the incoming AQ adapter is connected to the `REQ` input of `QUARTER_TO_BOOL`. Its `CNF` event is output to the `E1` output of the AX adapter.
2. **Data Forwarding**: The data word `D1` of the AQ adapter (quarter byte) is forwarded to the `IB` input of `QUARTER_TO_BOOL`. The converted BOOL value from the output `Q` of `QUARTER_TO_BOOL` is passed to the `D1` output of the AX adapter.

The entire sequence control and data conversion thus takes place transparently within the function block.

Technical Features

- The function block functions purely as an adapter converter and has no states or application logic of its own; all functionality is provided by the embedded `QUARTER_TO_BOOL`.
- The use of adapters (socket/plug) allows for loose coupling in the 4diac IDE and facilitates the exchange of communication interfaces.
- The comment "using STATUS logic" indicates that the conversion may be based on a status query (e.g., valid/invalid data), which is implemented within the internal function block.

State Overview

This function block does not define its own states. The state logic is entirely contained within the internal function block `QUARTER_TO_BOOL`.

Application Scenarios

- **Interface Adaptation**: Integration of an AQ-based sensor or actuator into a system that only supports AX-compatible adapters.
- **Adapter Transformation**: Use in hierarchies where different adapter types need to be coordinated without changing the underlying data logic.
- **Prototypes and Migration**: Rapid switching between AQ and AX variants during the development or testing phase.

Comparison with Similar Function Blocks

- **`QUARTER_TO_BOOL`**: Direct conversion of a quarter-byte value to a BOOL value with its own event interface; `AQ_TO_AX` encapsulates this function block in an adapter context.
- **`BYTE_TO_BOOL` or `WORD_TO_BOOL`**: Similar converters for other bit widths, but not specifically designed for adapter coupling.
- **Direct coupling of AQ and AX**: A connection would not be possible without conversion, as the adapter types are incompatible.

Conclusion

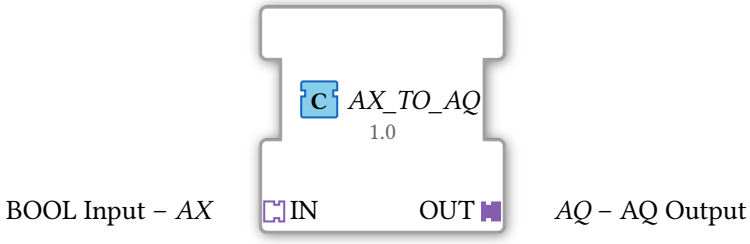
AQ_TO_AX` is a lean yet essential adapter converter that bridges the gap between AQ and AX interfaces. It simplifies the integration of quarter-byte signals into BOOL-based controllers and contributes to the reusability of function blocks in different adapter environments.

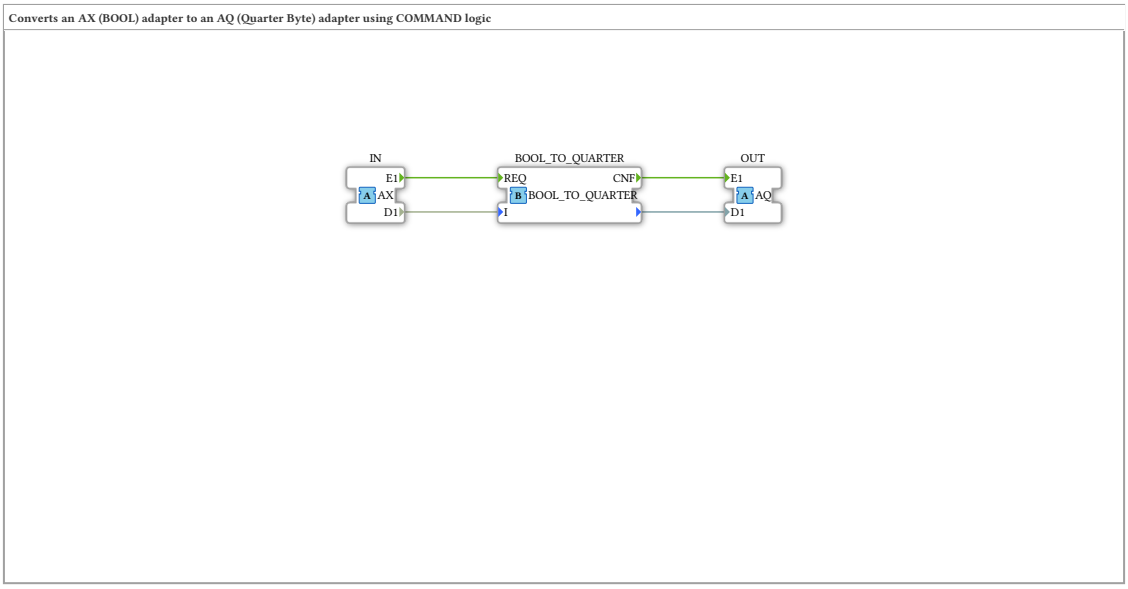
Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AX_TO_AQ





• — ○ • — ○ • — ○ •

Introduction

The **AX_TO_AQ** function block converts a Boolean signal (AX adapter) into a quarter-byte signal (AQ adapter). The conversion is based on event-driven COMMAND logic, where an incoming event triggers the conversion and the result is acknowledged via an output event. The function block encapsulates the conversion logic in an internal sub-function block, thus enabling a clean separation between adapter types.

Interface Structure

Event Inputs

Event	Description
`IN.E1`	Starts the conversion of the incoming Boolean value. Interpreted as a command.

Event Outputs

Event	Description
`OUT.E1`	Confirms the completion of the conversion and signals the validity of the output quarter-byte value.

Data Inputs

Data	Type	Description
`IN.D1`	BOOL (via AX adapter)	Boolean input value to be converted into a quarter byte.

Data Outputs

Data	Type	Description
`OUT.D1`	Quarter Byte (via AQ adapter)	Output value – the converted quarter byte (4-bit representation).

Adapters

Adapter	Direction	Type	Description

IN	Socket	adapter::types::unidirectional::AX	Receives the Boolean signal. The AX adapter provides a unidirectional Boolean channel.
OUT	Plug	adapter::types::unidirectional::AQ	Provides the converted quarter-byte signal. The AQ adapter is a unidirectional quarter-byte channel.

Functionality

1. An event at IN.E1 triggers the internal processing.
2. The sub-function block BOOL_TO_QUARTER is activated via its event input REQ and converts the Boolean value at I into a quarter byte.
3. After the conversion is complete, BOOL_TO_QUARTER reports this via its event CNF, which is then passed on to OUT.E1.
4. Simultaneously, the converted value is transferred via the data connection from the output QB of the sub-block to OUT.D1.

The entire process is event-driven and operates without internal state storage – each command generates a complete conversion.

Technical Features

- **Internal Encapsulation**: The conversion logic is fully implemented by the embedded function block logiBUS::utils::quarter::BOOL_TO_QUARTER. This simplifies maintenance and replacement of the conversion implementation.
- **Unidirectionality**: Both the input and output adapters are unidirectional; Feedback or bidirectional use is not supported.
- **Event Synchronization**: The output event pulse OUT.E1 guarantees that the data at output OUT.D1 is valid before the calling block continues processing.
- **License and Origin**: The block is licensed under EPL-2.0 and is included in the package adapter::conversion::unidirectional (see CompilerInfo).

State Overview

The function block does not have an explicit state machine. It operates as pure combinational logic with event-driven triggering:

- **Idle State**: No event is present at IN.E1. Outputs remain unchanged.
- **Active**: An event at IN.E1 starts the conversion. Upon completion, OUT.E1 is triggered, and the new value is output to OUT.D1. The function block then returns to its idle state.

Application Scenarios

- **Bus Signal Translation**: In an industrial bus system, Boolean control signals (e.g., bit values from sensors) are converted into compact 4-bit data words to reduce bandwidth.
- **Bit-to-Quarter-Byte Converter in Automation Networks**: When a subsystem can only process quarter-byte values but receives Boolean inputs.
- **Adapter Bridge**: The function block acts as an intermediary between an AX-compatible transmitter and an AQ-compatible receiver in a unidirectional chain.

Comparison with Similar Function Blocks

Function Block	Task	Difference
BOOL_TO_BYTE	Converts BOOL to a byte (8-bit)	Works with 8-bit output instead of 4-bit. Requires other adapter types (e.g., AB).
BOOL_TO_WORD	Converts BOOL to a word (16-bit)	Provides a wider output, often for analog or digital word signals.

AX_TO_AQ	Converts BOOL (AX) to a quarter byte (AQ)	Specifically designed for the unidirectional AX↔AQ adapter combination, it generates only 4-bit values.

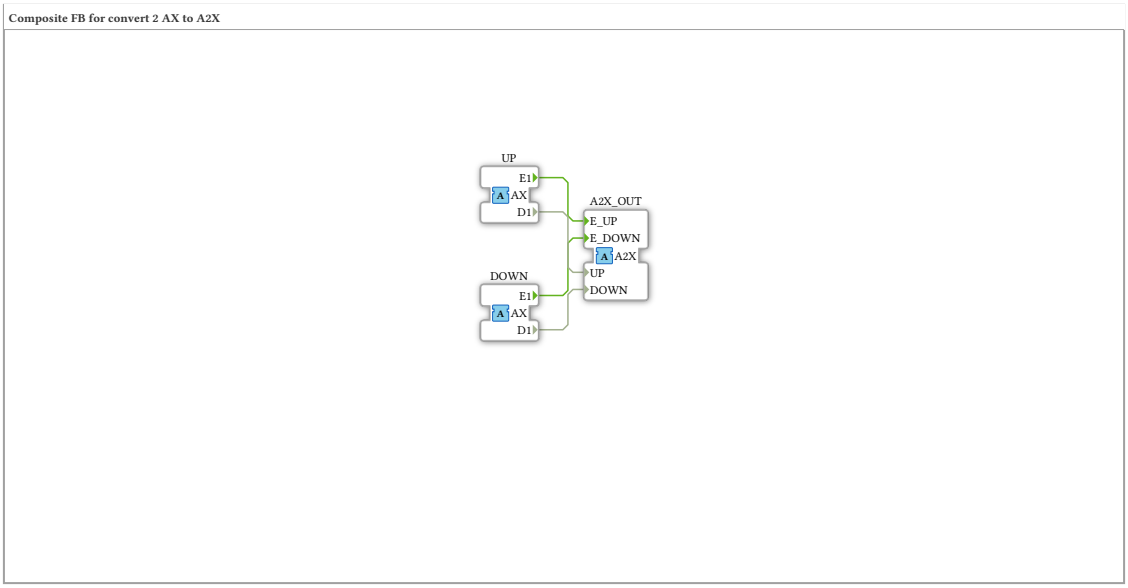
Conclusion

The **AX_TO_AQ** function block is a specialized and compact converter for converting Boolean adapter signals into the quarter-byte format. Its clear, event-driven structure and internal reuse of a standardized conversion block make it ideal for use in automation and bus systems where space-saving transmission of Boolean states is required. The simple interface and unidirectional nature facilitate integration into existing communication workflows.

Numerical Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace in ascending order.

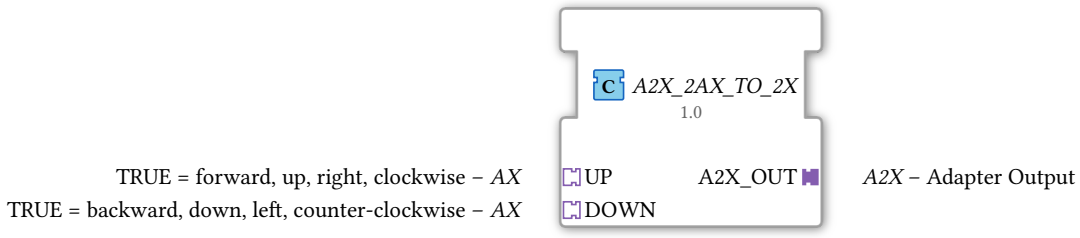
A2X_2AX_TO_2X



• – ◦ • – ◦ • – ◦ •

Introduction

The A2X_2AX_TO_2X is a composite function block used to convert two AX signals into one A2X signal. This block allows the combination of two unidirectional AX adapter signals into a single A2X adapter output.



Interface Structure

Event Inputs

No direct event inputs available

Event Outputs

No direct event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Sockets (Inputs):

- **UP**: AX adapter for positive direction of movement (TRUE = forward, up, right, clockwise)
- **DOWN**: AX adapter for negative direction of movement (TRUE = backward, down, left, counterclockwise)

Plugs (Outputs):

- **A2X_OUT**: A2X adapter output for the combined signals

Functionality

The composite function block connects the two AX adapter inputs (UP and DOWN) directly to the A2X adapter output. Both event and data signals are forwarded:

- UP events and data are forwarded to E_UP and UP of the A2X_OUT adapter.
- DOWN events and data are forwarded to E_DOWN and DOWN of the A2X_OUT adapter.

Technical Features

- Implemented as a composite function block without internal logic
- Direct signal pass-through without delay
- Uses unidirectional adapter types
- No state or internal processing

State Overview

The function block has no internal states, as it is a pure pass-through function.

Application Scenarios

- Combining two separate motion control signals
- Converting AX-based control systems to A2X interfaces
- Integration into larger control systems with A2X interface requirements
- Bidirectional motion sensing in automation systems

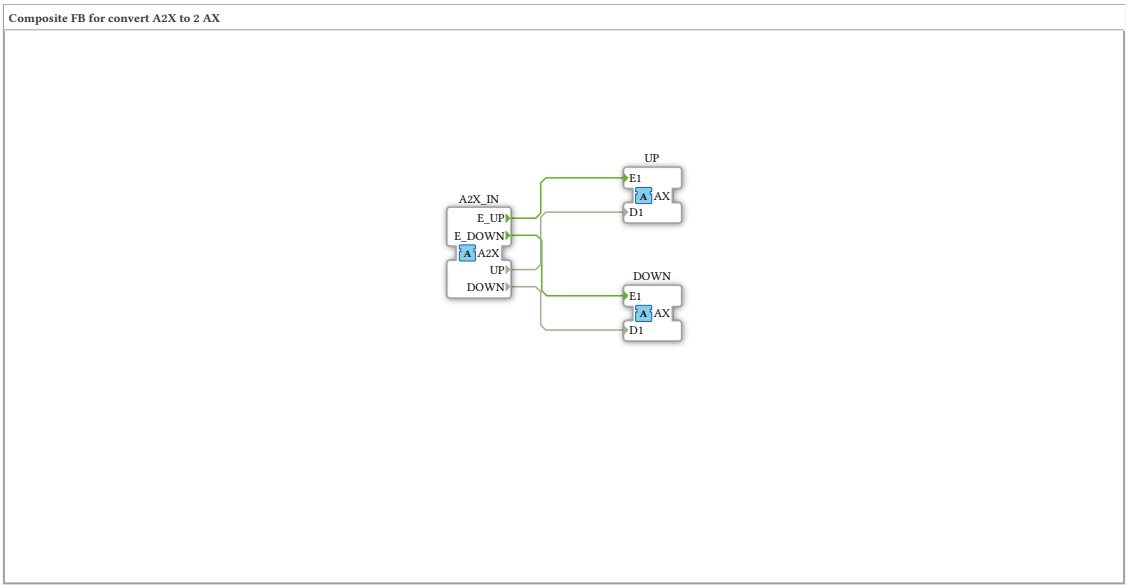
⚖️ Comparison with similar components

Compared to simple AX adapters, this component enables the combination of two opposing motion directions into a single A2X signal. It eliminates the need for manual wiring of two AX adapters to create one A2X adapter.

Conclusion

The A2X_2AX_TO_2X is a simple yet effective composite function block that simplifies the integration of AX-based control components into systems with A2X interfaces. Direct signal forwarding without additional processing ensures reliable and lossless signal transmission.

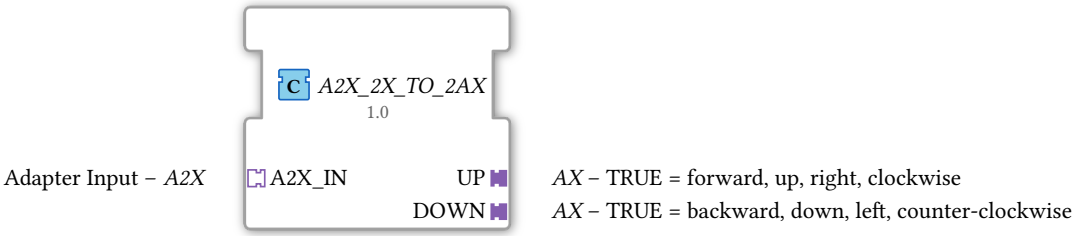
A2X_2X_TO_2AX



• – ○ • – ○ • – ○ •

Introduction

The A2X_2X_TO_2AX is a composite function block used to convert A2X signals into two separate AX signals. This block enables the conversion of bidirectional control signals into unidirectional motion commands.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- No direct event outputs available

Data Inputs

- No direct data inputs available

Data Outputs

- No direct data outputs available

Adapters

Input Adapters:

- `A2X_IN` (Socket) - Adapter input of type `adapter::types::unidirectional::A2X`

Output Adapters:

- `UP` (Plug) - Output for positive movement direction (TRUE = forward, up, right, clockwise)
- `DOWN` (Plug) - Output for negative movement direction (TRUE (backwards, down, left, counterclockwise)

Functionality

The composite function block forwards events and data directly from the A2X_IN adapter to the corresponding UP and DOWN output adapters:

- The E_UP event is forwarded to UP.E1
- The E_DOWN event is forwarded to DOWN.E1
- The UP data is forwarded to UP.D1
- The DOWN data is forwarded to DOWN.D1

Technical Features

- Implemented as a composite function block without internal logic
- Direct signal forwarding without delay
- Uses unidirectional adapter types for clear signal flow

State Overview

The function block has no internal state and operates stateless. All input signals are immediately forwarded to the corresponding outputs.

Application Scenarios

- Conversion of bidirectional drive signals into separate forward/reverse controls
- Control of actuators with separate extension and retraction commands
- Integration into control systems with different signal formats
- Speed control with separate clockwise/counterclockwise controls

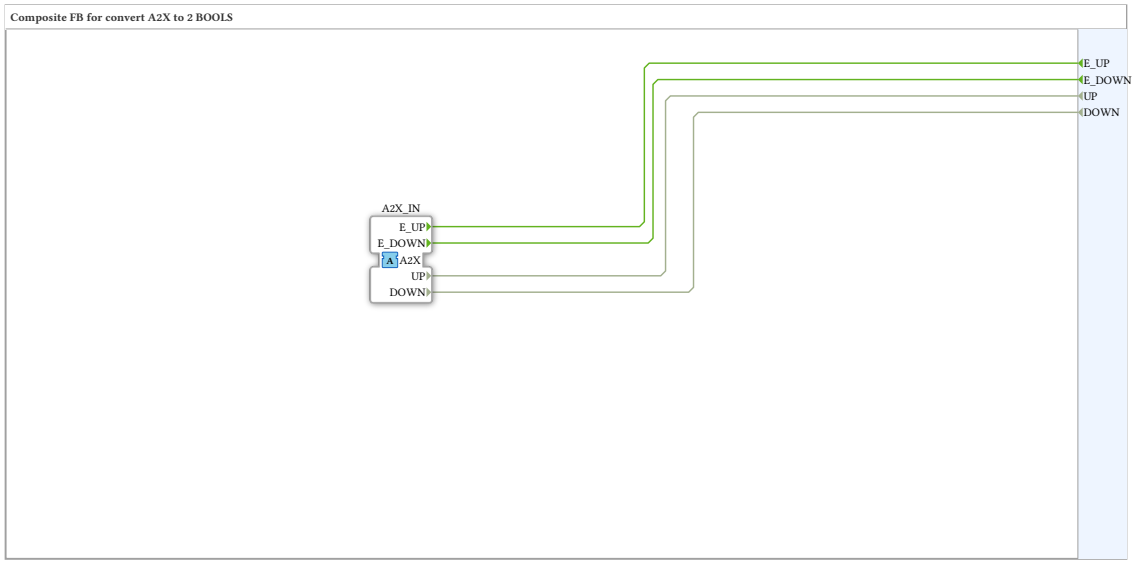
⚖️ Comparison with similar function blocks

Compared to simple converter function blocks, this composite function block offers a structured solution for splitting bidirectional signals into two independent unidirectional channels. The use of standardized adapter types ensures compatibility within the adapter framework.

Conclusion

The A2X_2X_TO_2AX function block represents an efficient and standards-compliant solution for signal conversion. Its simple, direct routing logic and the use of established adapter types make it ideal for integration into more complex control systems that require the separation of motion directions.

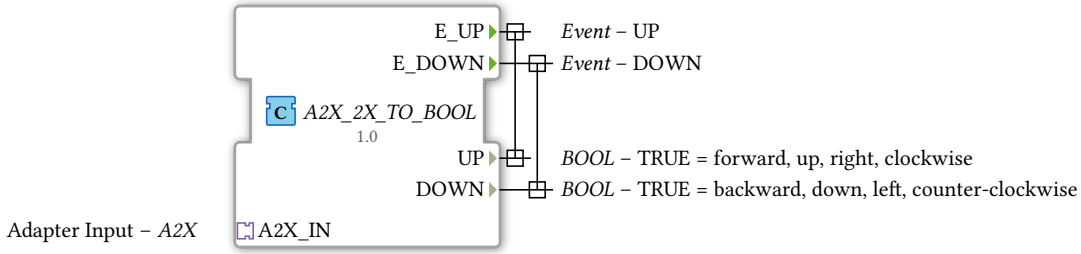
A2X_2X_TO_BOOL



• – ○ • – ○ • – ○ •

Introduction

The A2X_2X_TO_BOOL function block is a composite function block used to convert A2X signals into two separate BOOL signals. It enables the conversion of bidirectional control signals into simple logic outputs for various applications.



Interface Structure

Event Inputs

No direct event inputs available

Event Outputs

- **E_UP**: Activated when an upward/forward movement is detected
- **E_DOWN**: Activated when a downward/backward movement is detected

Data Inputs

No direct data inputs available

Data Outputs

- **UP**: BOOL - TRUE for forward, upward, right, or clockwise
- **DOWN**: BOOL - TRUE for backward, downward, left, or counterclockwise

Adapter

- **A2X_IN**: A2X adapter input for unidirectional communication

Functionality

The function block receives signals via the A2X adapter and forwards them directly to the corresponding outputs. The events and data values are passed directly from the adapter to the output interfaces without any additional processing or delay.

Technical Features

- Composite function block without internal logic
- Direct pass-through of adapter signals
- No state retention or storage
- Real-time signal forwarding

State Overview

The function block has no internal states. It behaves purely combinatorially and immediately forwards incoming signals to the outputs.

Application Scenarios

- Control of motion directions in actuators
- Conversion of complex control signals into simple direction commands
- Connection of A2X-compatible components to BOOL-based controllers
- Direction detection in drive systems

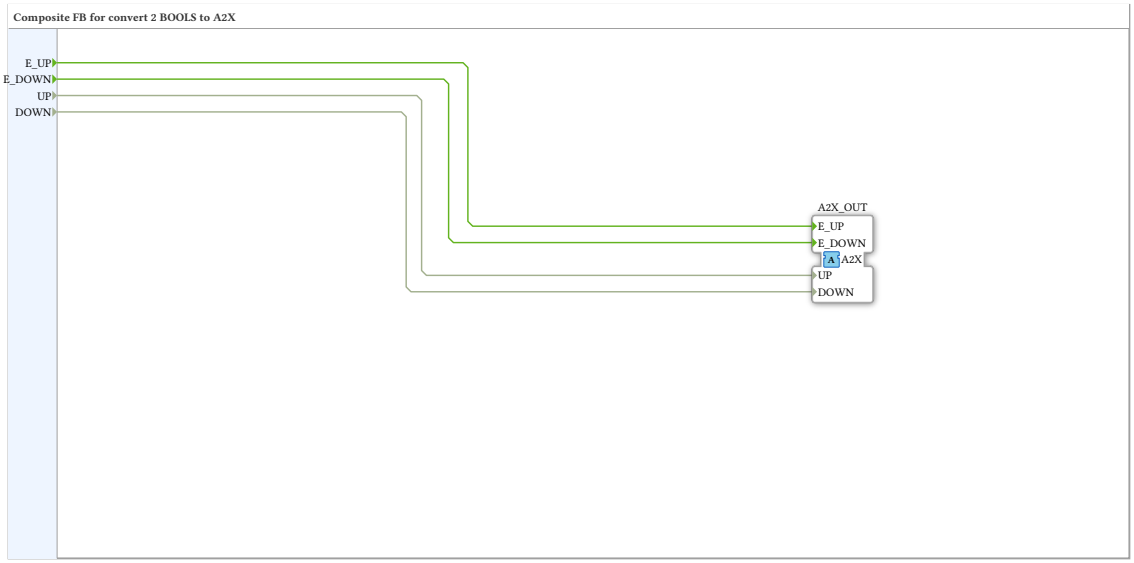
⚖️ Comparison with Similar Blocks

Compared to other conversion blocks, A2X_2X_TO_BOOL offers a specific solution for converting A2X signals into Boolean direction signals. Other function blocks could offer additional features such as filtering, delay, or more complex signal processing.

Conclusion

The A2X_2X_TO_BOOL function block provides a simple and efficient solution for converting A2X control signals into Boolean directional outputs. Its clear interface definition and direct signal routing make it particularly suitable for applications requiring unadulterated transmission of control information.

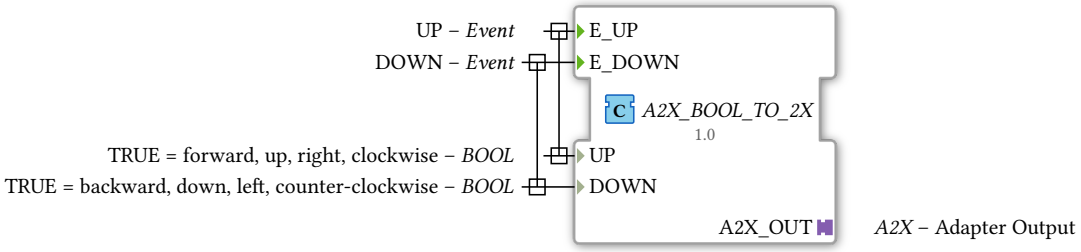
A2X_BOOL_TO_2X



• – ○ • – ○ • – ○ •

Introduction

The A2X_BOOL_TO_2X is a compound function block that converts two BOOL values into an A2X output format. This block serves as a bridge between simple Boolean control signals and the standardized A2X adapter interface.



Interface Structure

Event Inputs

- **E_UP**: Event input for up/forward movement
- **E_DOWN**: Event input for down/backward movement

Event Outputs

No direct event outputs available

Data Inputs

- **UP**: BOOL input for up/forward direction (TRUE = forward, up, right, clockwise)
- **DOWN**: BOOL input for down/backward direction (TRUE = backward, down, left, counterclockwise)

Data Outputs

No direct data outputs available

Adapters

- **A2X_OUT**: Unidirectional A2X output adapter for forwarding the converted signals

Functionality

The function block receives two independent Boolean input signals (UP and DOWN) along with their corresponding event inputs are passed directly to the A2X output adapter, with event and data connections processed in parallel.

Technical Features

- Completely unidirectional signal processing
- Direct pass-through without additional logic processing
- Uses the standardized A2X adapter interface
- No internal states or delays

State Overview

The function block has no internal states and operates stateless. Every incoming event is immediately passed to the corresponding adapter output.

Application Scenarios

- Control of bidirectional actuators (e.g., motors, cylinders)
- Interface between simple switching logic and more complex control systems
- Conversion of push-button/switch signals into standardized motion signals
- Drive controls with forward/reverse functionality

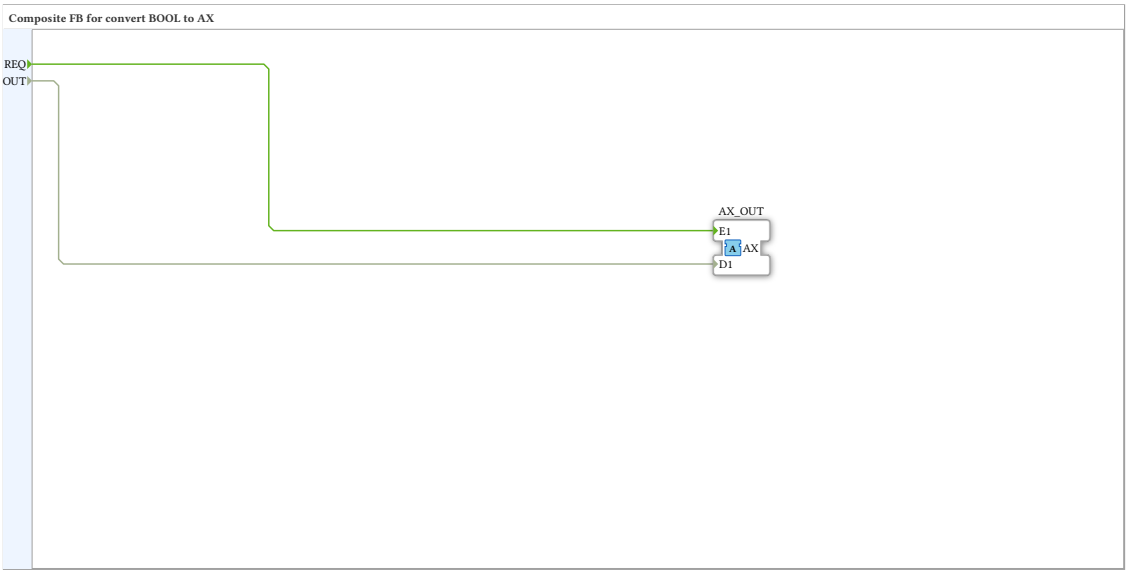
⚖️ Comparison with similar components

Compared to simple conversion components, A2X_BOOL_TO_2X offers the specific combination of two Boolean inputs with a standardized A2X interface. Other components might perform individual BOOL-to-A2X conversions, but this component combines both directions in a single component.

Conclusion

The A2X_BOOL_TO_2X is a specialized interface component that provides a simple and efficient interface between basic Boolean control signals and the standardized A2X protocol. Its clear structure and direct pass-through make it a reliable solution for bidirectional control applications.

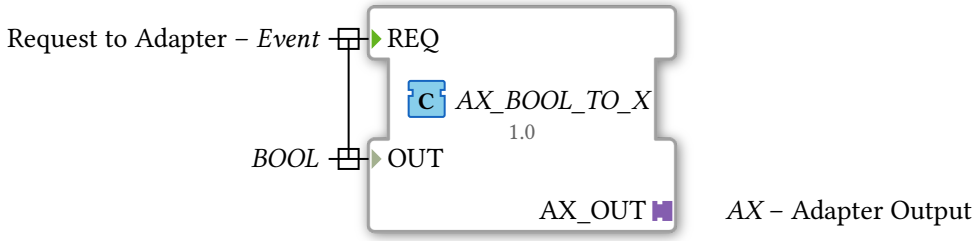
AX_BOOL_TO_X



• — ○ • — ○ • — ○ •

Introduction

The AX_BOOL_TO_X function block is a composite function block used to convert a BOOL value into the AX adapter format. It enables the easy conversion of Boolean values into a standardized output format for further processing in automation systems.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: BOOL input value to be converted

Data Outputs

- No direct data outputs available

Adapters

- **AX_OUT**: Unidirectional AX adapter output for the converted data

Functionality

The function block receives a Boolean value at the OUT data input and converts it to the AX adapter format upon arrival of the REQ event. The conversion is automatic, and the converted value is output via the AX_OUT adapter. The block acts as a bridge between simple Boolean signals and the standardized AX communication format.

Technical Features

- Composite function block with internal networking
- Uses a unidirectional AX adapter for output
- Automatic forwarding of events and data to the adapter
- Simple and direct signal conversion without additional logic

State Overview

The function block has a simple state machine:

1. **Wait state**: Waits for a REQ event
2. **Process state**: Upon a REQ event, the BOOL value is converted and output via the adapter
3. Returns to wait state after conversion is complete

Application Scenarios

- Integration of simple switching states into more complex AX-based systems
- Interface adaptation between different automation components
- Protocol conversion in distributed control systems
- Integration of basic sensors into AX-compliant architectures

⚖️ Comparison with Similar Blocks

Compared to other conversion blocks, AX_BOOL_TO_X stands out due to its specific focus on the AX adapter format. While general conversion blocks often support multiple output formats, this block offers an optimized solution for the specific AX interface.

🛠️ Related Exercises

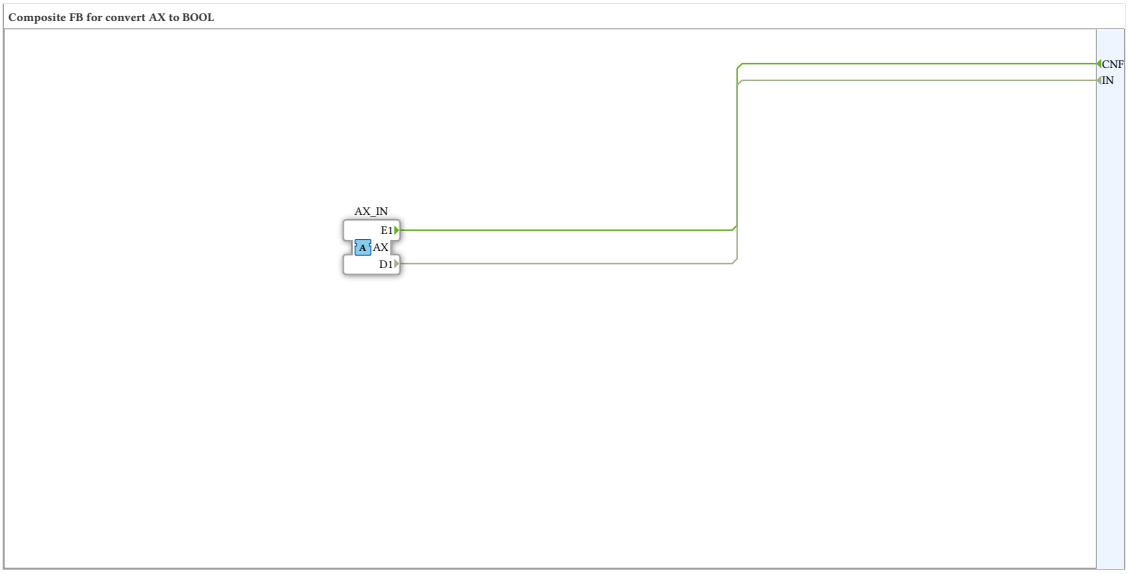
- Exercise_001_AX_b

- Exercise_002a2_AX
- Exercise_004b_AX
- Exercise_004b_AX_ASR
- Exercise_004b_AX_ASR_X

Conclusion

AX_BOOL_TO_X is a specialized conversion block that provides a simple and efficient interface between Boolean signals and the AX adapter format. Its clear structure and direct functionality make it a reliable solution for integration tasks in automation systems.

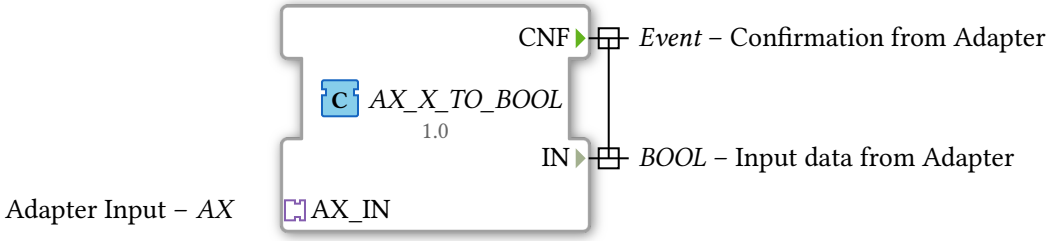
AX_X_TO_BOOL



• – ◦ • – ◦ • – ◦ •

Introduction

The AX_X_TO_BOOL function block is a composite function block designed for converting AX data to the BOOL data type. This block acts as an adapter converter and enables unidirectional data conversion between different data types in 4diac systems.



Interface Structure

Event Inputs

No direct event inputs are available.

Event Outputs

- **CNF** (Confirmation): Confirmation event output by the adapter.

Data Inputs

No direct data inputs are available.

Data Outputs

- **IN** (BOOL): Converted input data from the adapter.

Adapter

- **AX_IN**: Adapter input of type AX for unidirectional data transmission.

Functionality

The function block receives data via the AX_IN adapter and converts it to the BOOL data type. Upon successful conversion, the confirmation event CNF is triggered, and the converted data is made available at the IN output. The internal structure directly connects the adapter event output E1 to the CNF output and the adapter's data output D1 to the IN data output.

Technical Features

- Implemented as a composite function block
- Uses unidirectional adapter communication
- Direct passthrough without additional processing logic
- Simple and efficient data conversion

State Overview

The function block has a simple state:

- **Ready**: Waits for incoming data from the adapter
- Upon receiving data, conversion and output occur immediately

Application Scenarios

- Integration of AX-based components into BOOL-oriented controllers
- Adapter conversion in heterogeneous automation systems
- Interface adaptation between different device types
- Protocol conversion in distributed control systems

⚖️ Comparison with Similar Function Blocks

Compared to other conversion blocks, AX_X_TO_BOOL stands out due to its specific focus on AX-to-BOOL conversion. While universal converters often require more complex parameterization, this block offers an optimized solution for this specific use case.

🛠️ Related exercises

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_094a_AX

Conclusion

AX_X_TO_BOOL is a specialized conversion block that enables efficient and reliable conversion of AX data to BOOL values. Its simple structure and direct functionality make it particularly suitable for systems requiring high-performance data type conversion.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Unidirectional BOOL adapter conversion

🎧 Podcast

- George Boole: How an autodidact laid the foundation of our digital world with 0 and 1

🛠️ Related exercises

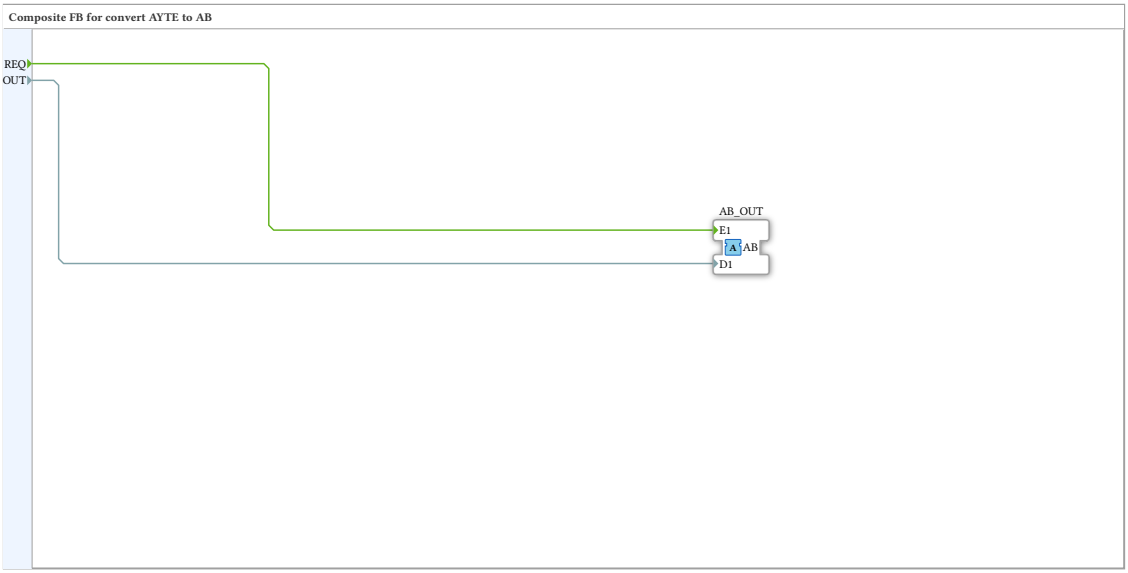
- Uebung_001_AX_b
- Uebung_002a2_AX

- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

Numerical Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace in ascending order.

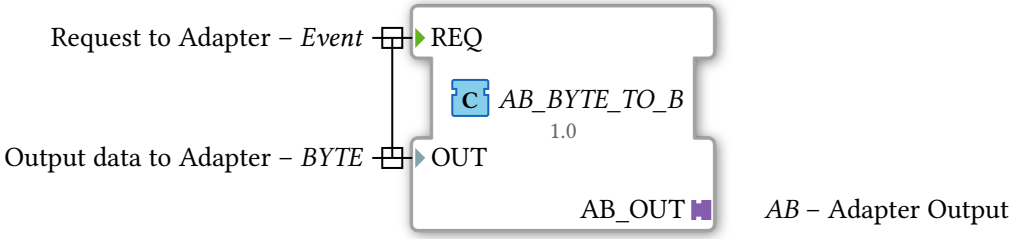
AB_BYTE_TO_B



• – ◦ • – ◦ • – ◦ •

Introduction

The AB_BYTE_TO_B function block is a composite function block used to convert a BYTE data type to the AB adapter format. It enables unidirectional data transfer of byte values via a standardized adapter interface.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger adapter conversion

Event Outputs

No direct event outputs available

Data Inputs

- **OUT**: BYTE input value that is forwarded to the adapter

Data Outputs

No direct data outputs available

Adapter

- **AB_OUT**: Unidirectional AB adapter output for data output

Functionality

The function block receives a request for data conversion via the REQ event input. Simultaneously, the byte value at the OUT data input is processed. Upon arrival of the REQ event, the byte value is output via the AB_OUT adapter. Data transmission is unidirectional from the function block to the connected adapter.

Technical Features

- Implemented as a composite function block with an internal network structure
- Uses the unidirectional AB adapter interface
- Direct event and data connections between inputs and the adapter
- Positioning attributes for graphical representation in the development environment

State Overview

The function block has a simple state machine:

1. **Wait State**: No active processing
2. **Activate State**: Data is forwarded to the adapter upon a REQ event
3. **Return**: Automatic return to wait state after processing

Application Scenarios

- Byte data conversion for adapter-based communication
- Integration into larger control systems with standardized adapter interfaces
- Data preparation for downstream processing components
- Protocol conversion for different data types

⚖️ Comparison with Similar Blocks

Compared to simple conversion blocks, AB_BYTE_TO_B offers the advantage of a standardized adapter interface, which allows for better reusability and integration. in more complex system architectures. Its unidirectional design makes it particularly suitable for output scenarios.

Conclusion

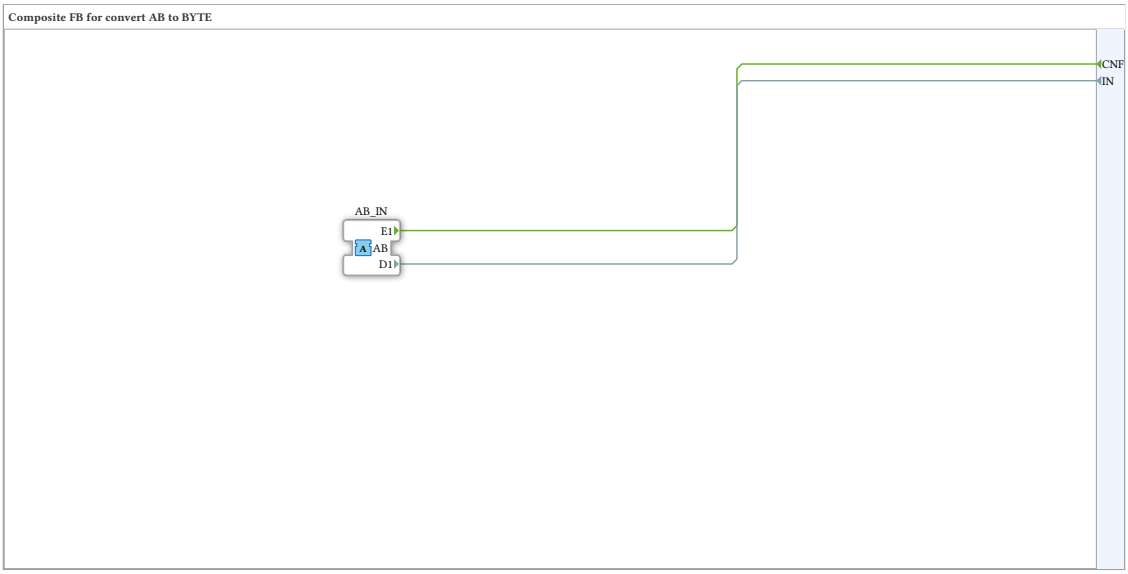
AB_BYTE_TO_B is a specialized conversion module that, thanks to its adapter-based architecture, enables flexible and standards-compliant integration into 4diac-based control systems. Its simple and robust operation makes it a reliable solution for byte-to-adapter conversions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

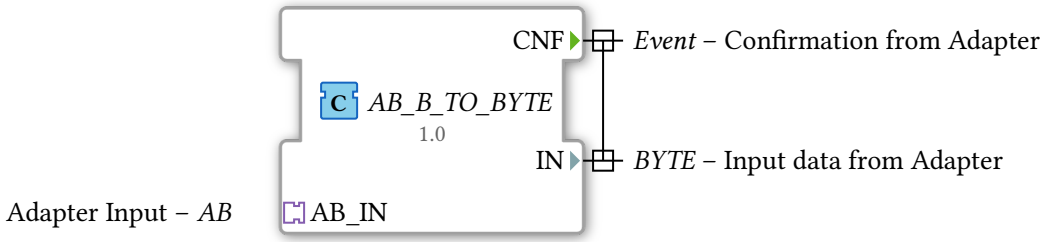
AB_B_TO_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

The AB_B_TO_BYTE function block is a composite function block used to convert AB data to the BYTE format. It acts as an adapter converter and enables the seamless integration of AB data sources into systems that expect BYTE data.



Interface Structure

Event Inputs

No direct event inputs are available – events are received via the adapter.

Event Outputs

- **CNF** (Confirmation): Confirmation event received by the adapter

Data Inputs

No direct data inputs are available – data is received via the adapter.

Data Outputs

- **IN** (BYTE): Input data converted by the adapter

Adapter

- **AB_IN**: Adapter input of type `adapter::types::unidirectional::AB` for receiving AB data

Functionality

The function block receives AB data via the adapter input AB_IN and converts it to the BYTE format. Upon receiving data via the adapter, the acknowledgment event CNF is automatically triggered, and the converted data is made available at the IN output.

Technical Features

- Implemented as a composite function block with an internal network structure
- Uses unidirectional adapter communication
- Automatic event forwarding from the adapter to the CNF output
- Direct data forwarding from the adapter to the IN output

State Overview

The function block operates without state and reacts immediately to incoming adapter events. Each E1 event from the adapter triggers the CNF event and outputs the adapter's current D1 data to IN.

Application Scenarios

- Integration of AB-based sensors or actuators into BYTE-oriented control systems
- Protocol conversion in industrial automation systems
- Adapter-based data conversion in distributed control systems

⚖️ Comparison with Similar Function Blocks

Compared to simple conversion blocks, AB_B_TO_BYTE offers the advantage of adapter integration, enabling a standardized and type-safe interface. Other conversion blocks typically use direct inputs/outputs instead of adapters.

Conclusion

AB_B_TO_BYTE provides an elegant solution for converting AB to BYTE data using adapter interfaces. The composite implementation ensures robust and reliable data conversion, which is particularly advantageous in complex automation systems with standardized communication interfaces.

Unidirectional BYTE adapter conversion

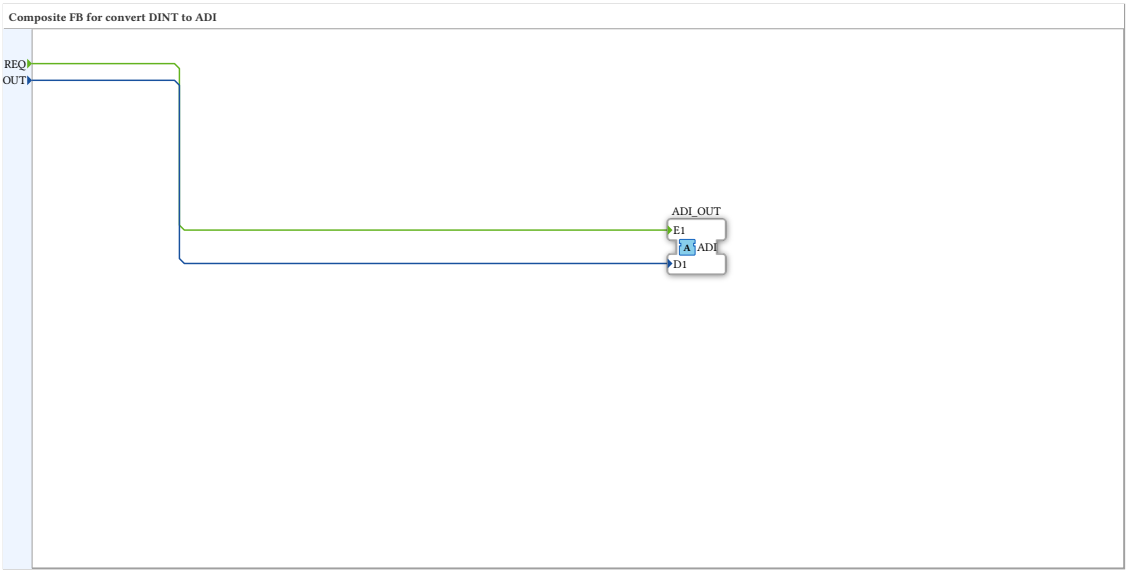
🛠️ Related exercises

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

Numerical Index (unidirectional)

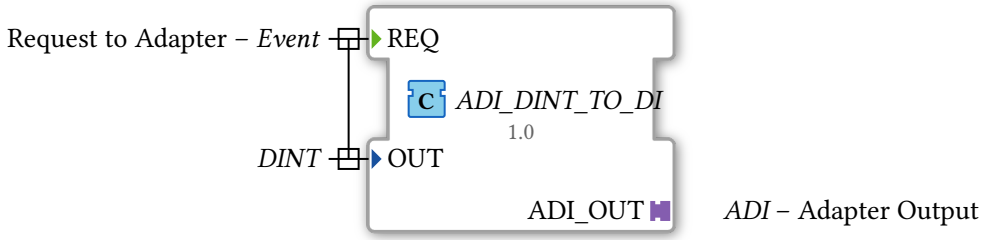
Here you will find all exercises from the `unidirectional` workspace in ascending order.

ADI_DINT_TO_DI



Introduction

The ADI_DINT_TO_DI function block is a composite function block used to convert a DINT value to the ADI adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: DINT input value to be converted

Data Outputs

- No direct data outputs available

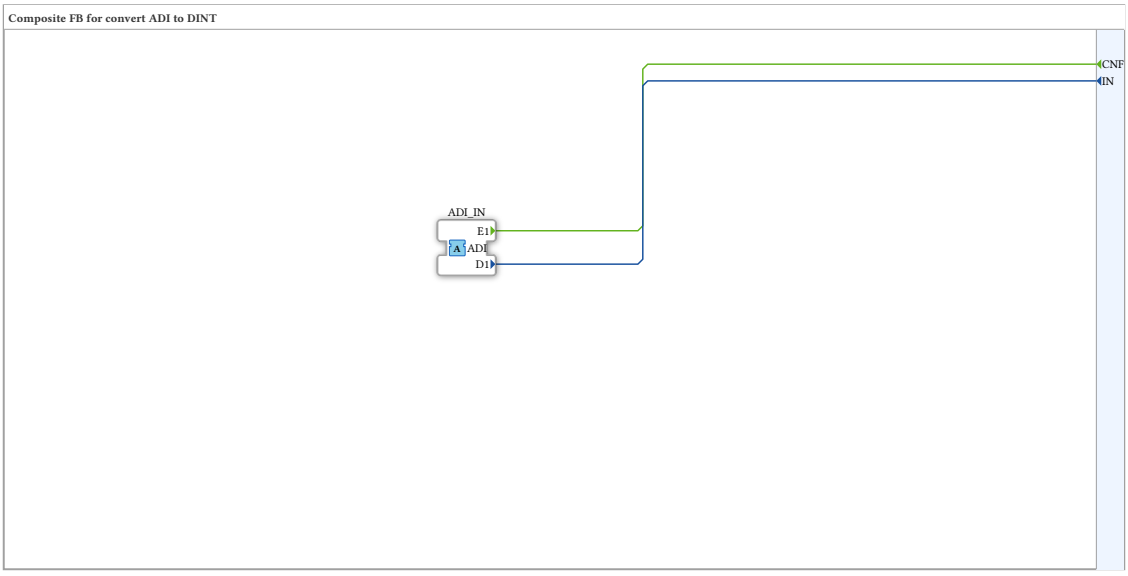
Adapter

- **ADI_OUT**: Plug of type `ADI`

Functionality

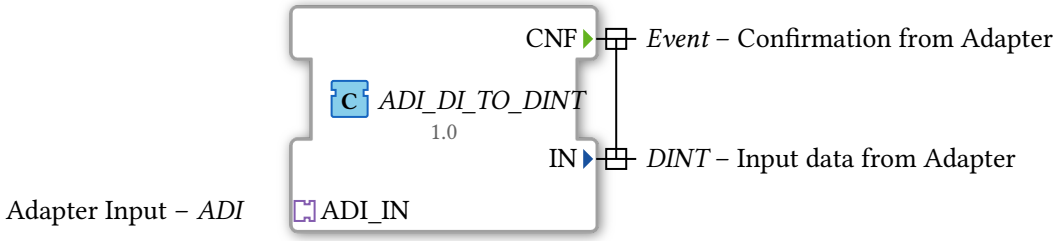
The function block receives a DINT value at the data input OUT and converts it to the ADI adapter format upon arrival of the REQ event.

ADI_DI_TO_DINT



Introduction

The ADI_DI_TO_DINT function block is a composite function block used to convert the ADI adapter format to a DINT value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: DINT output value of the converted data

Adapter

- **ADI_IN**: Socket of type `ADI`

Functionality

The function block receives data via the ADI adapter socket and makes the contained DINT value available at the IN output as soon as an event arrives via the adapter (CNF).

DINT Conversion

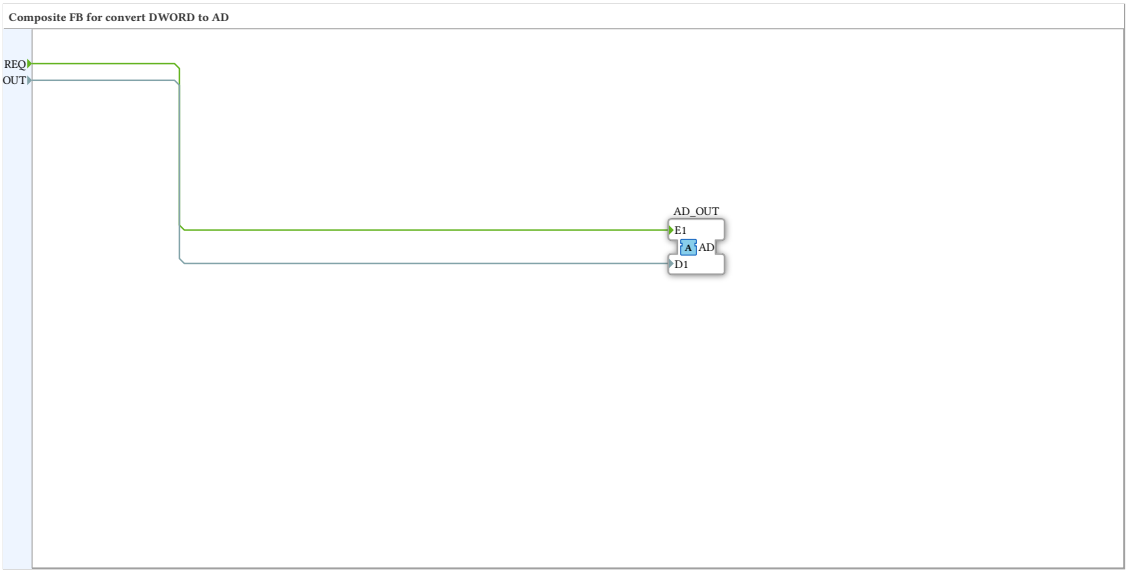
🎧 Podcast

- SINT, INT, DINT: Why the choice of data type determines efficiency and errors ...

Numerical Index (unidirectional)

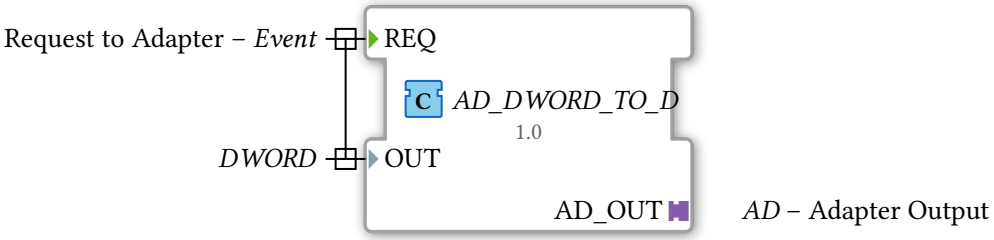
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AD_DWORD_TO_D



Introduction

The AD_DWORD_TO_D function block is a composite function block used to convert a DWORD value to the AD adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: DWORD input value to be converted

Data Outputs

- No direct data outputs available

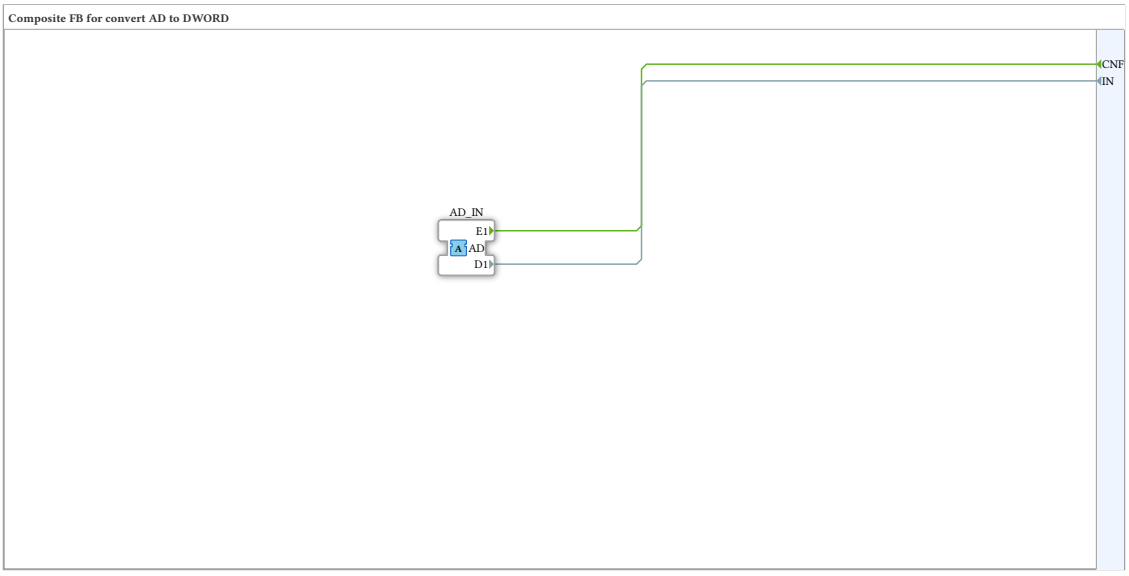
Adapter

- **AD_OUT**: Plug of type `AD`

Functionality

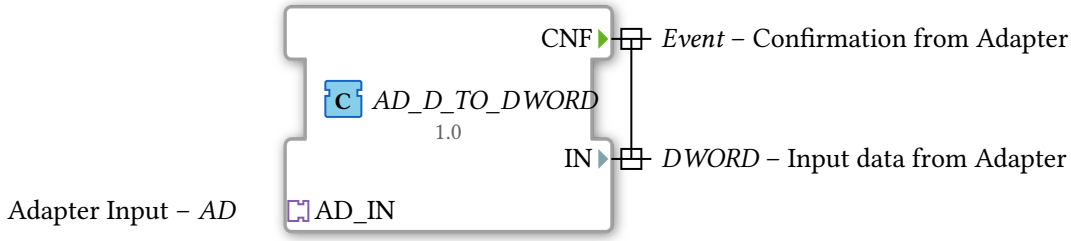
The function block receives a DWORD value at the OUT data input and converts it to the AD adapter format upon arrival of the REQ event.

AD_D_TO_DWORD



Introduction

The AD_D_TO_DWORD function block is a composite function block used to convert the AD adapter format to a DWORD value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: DWORD output value of the converted data

Adapter

- **AD_IN**: Socket of type `AD`

Functionality

The function block receives data via the AD adapter socket and makes the contained DWORD value available at the IN output as soon as an event arrives via the adapter (CNF).

DWORD conversion

🛠️ Related exercises

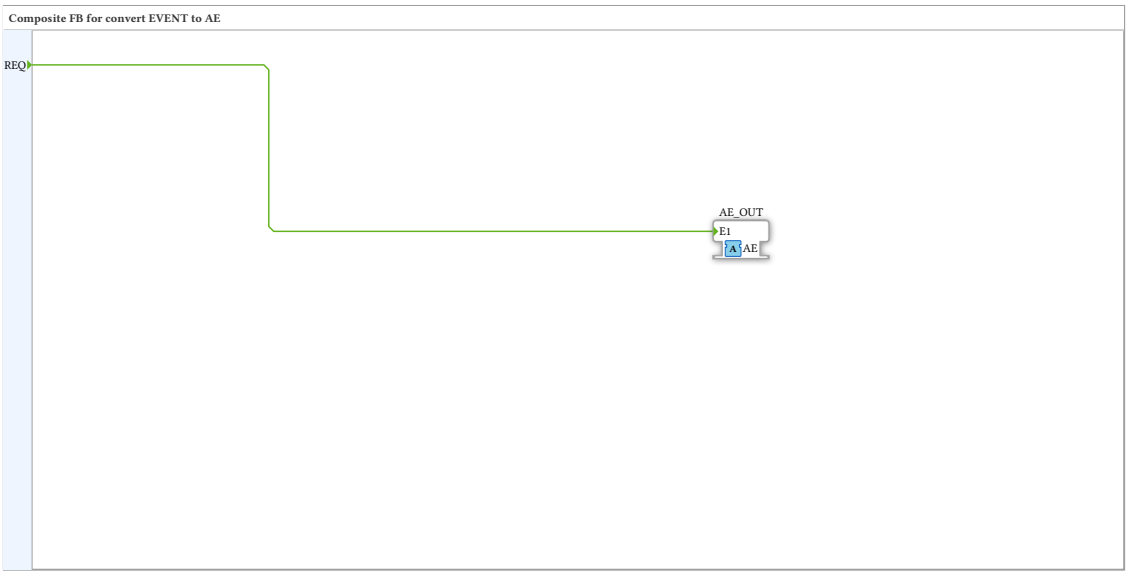
- Uebung_034b
- SIN_TO_PCAN_Callback_DWord
- Uebung_152
- Uebung_153

Numerical Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace in ascending order.

- AE_E_TO_EVENT
- AE_EVENT_TO_E
- ASR_2EVENTS_TO_SR
- ASR_SR_TO_2EVENTS
- ASRT_3EVENTS_TO_SRT
- ASRT_SRT_TO_3EVENTS
- EVENT_Adapter_Conv

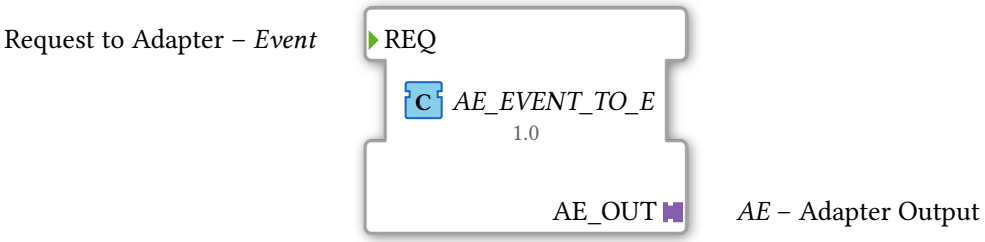
AE_EVENT_TO_E



• – ○ • – ○ • – ○ •

Introduction

The AE_EVENT_TO_E function block is a composite function block used to convert events into AE adapter signals. It enables the conversion of a simple event into an adapter-based output signal, which is particularly useful when integrating various communication interfaces.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger adapter conversion

Event Outputs

- No direct event outputs available

Data Inputs

- No data inputs available

Data Outputs

- No data outputs available

Adapter

- **AE_OUT**: Unidirectional AE adapter output for forwarding the converted events

Functionality

This function block acts as a simple converter that forwards incoming REQ events directly to the AE_OUT adapter. Each time a REQ event arrives, the corresponding E1 event is triggered at the AE adapter output. No data processing or transformation takes place, as the block only forwards events.

Technical Features

- Composite function block with a simple network structure
- Uses a unidirectional AE adapter interface
- Contains no data processing components
- Direct event forwarding without delay

State Overview

The function block has a simple state:

- **Ready**: Waits for incoming REQ events
- Upon arrival of a REQ, AE_OUT.E1 is immediately triggered, and the block returns to the ready state

Application Scenarios

- Integration of event-based systems into adapter-based architectures
- Coupling of different communication protocols
- As a bridge between simple events and more complex adapter interfaces
- In systems requiring a standardized AE interface

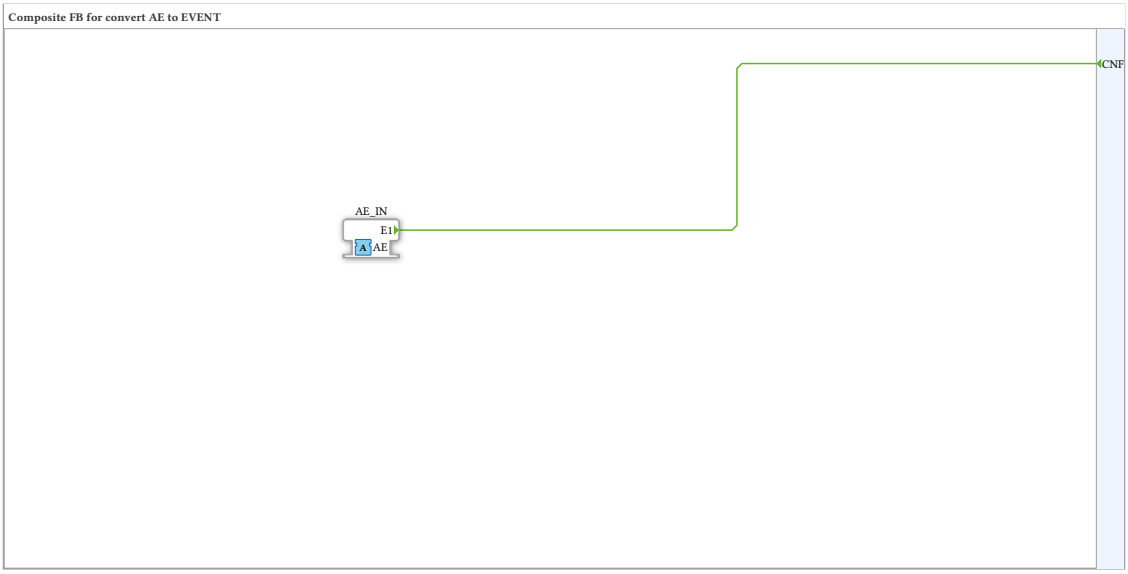
⚖️ Comparison with Similar Blocks

Compared to other conversion blocks, AE_EVENT_TO_E is characterized by its simplicity. While other converters often perform data processing or protocol translation, this block is limited to pure event forwarding. It is specifically optimized for integration into adapter-based systems.

Conclusion

AE_EVENT_TO_E is a specialized function block for the simple conversion of events into AE adapter signals. Its strength lies in its minimalist implementation and reliable event forwarding, making it ideal for system integrations where no data processing is required.

AE_E_TO_EVENT



• – ○ • – ○ • – ○ •

Introduction

The AE_E_TO_EVENT function block is a composite function block used to convert adapter events into standard events. It enables the conversion of adapter signals into generally usable event outputs and serves as a bridge between adapter interfaces and standardized event-based communication structures.



Interface Structure

Event Inputs

No direct event inputs available

Event Outputs

- **CNF** - Acknowledge event from the adapter

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapter

- **AE_IN** - Adapter input of type `adapter::types::unidirectional::AE`

Functionality

The function block forwards the E1 event from the adapter input AE_IN directly to the CNF output. This is a simple pass-through functionality where adapter events are converted into standardized function block events.

Technical Features

- Unidirectional signal processing
- No data processing or conversion
- Direct event forwarding without delay
- Positioning of the CNF output at Y position 46

State Overview

The function block has a simple state:

- **Ready**: Waiting for an E1 event from the adapter
- Upon arrival of E1, CNF is triggered immediately

Application Scenarios

- Integration of adapter interfaces into existing function block networks
- Coupling of different communication protocols
- Extension of systems with specific adapter requirements
- Protocol implementation in distributed automation systems

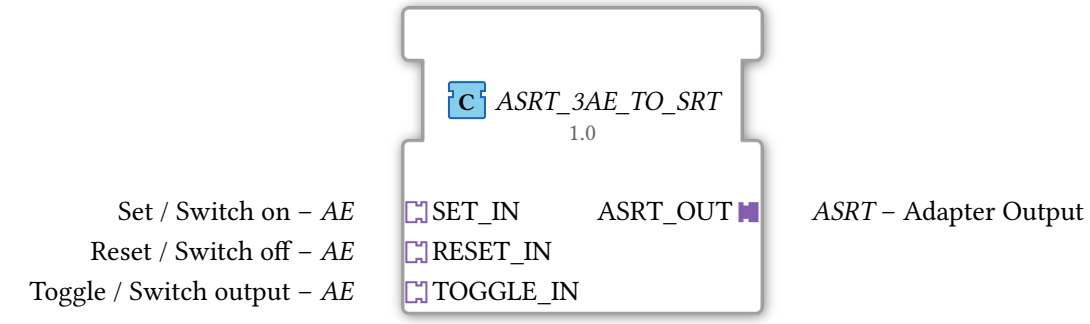
⚖️ Comparison with Similar Blocks

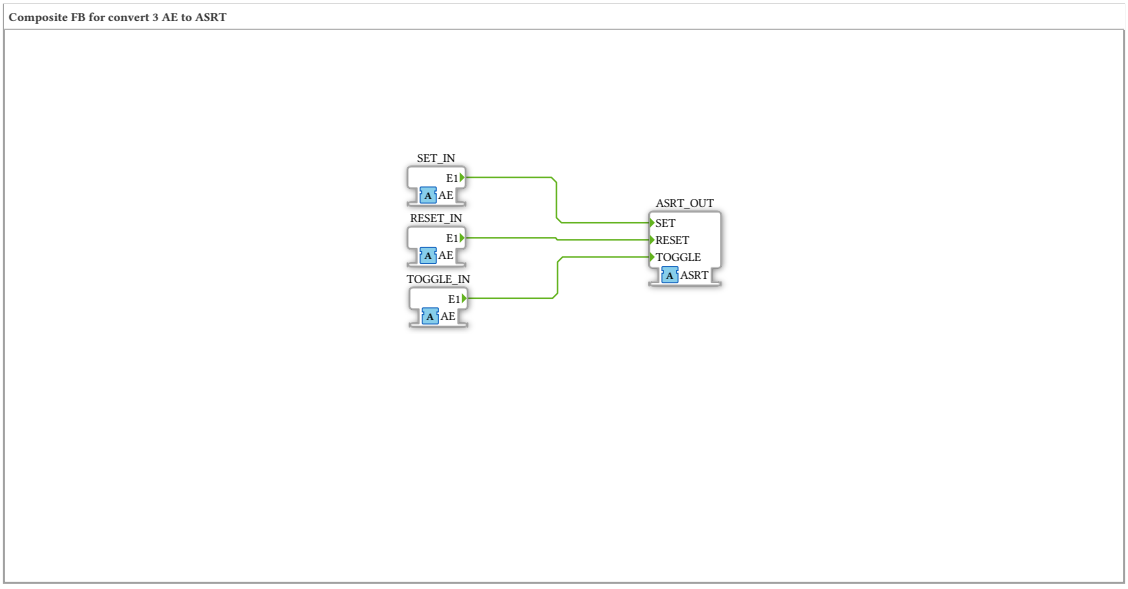
Compared to other conversion blocks, AE_E_TO_EVENT is characterized by its simplicity. While other converters often offer additional data processing or signal conditioning, this block is limited to pure event forwarding.

Conclusion

The AE_E_TO_EVENT function block represents a streamlined and efficient solution for adapter integration. Its simplicity makes it reliable and easy to understand, while simultaneously fulfilling the necessary bridging function between adapter interfaces and standardized event systems.

ASRT_3AE_TO_SRT





• – ○ • – ○ • – ○ •

Introduction

The function block **ASRT_3AE_TO_SRT** is a composite FB that merges three separate unidirectional AE adapters (pure event, no payload data) into one common ASRT adapter (a set/reset/toggle event triple). It is the `TOGGLE`-extended variant of ASR_2AE_TO_SR and the adapter-based counterpart of ASRT_3EVENTS_TO_SRT.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket	`SET_IN`	`adapter::types::unidirectional::AE`	Set / switch on.

Socket	RESET_IN	adapter::types::unidirectional::AE	Reset / switch off.
Socket	TOGGLE_IN	adapter::types::unidirectional::AE	Toggle / invert output.
Plug	ASRT_OUT	adapter::types::unidirectional::ASRT	Merged set/reset/toggle signal.

Functionality

The block is a pure FBNetwork of three event connections with no algorithms of its own: SET_IN.E1 is wired directly to ASRT_OUT.SET, RESET_IN.E1 to ASRT_OUT.RESET, and TOGGLE_IN.E1 to ASRT_OUT.TOGGLE. An event at the respective AE socket immediately triggers the matching event at the ASRT plug.

Technical Features

- **Pure wiring**: A composite FB with no ECC or algorithms, consisting solely of three direct event connections.
- **Adapter-based instead of event-based**: Unlike ASRT_3EVENTS_TO_SRT (classic event inputs), this block uses AE adapter sockets, letting it fit seamlessly into adapter-based networks.

State Overview

The function block has **no state machine**. Every event at SET_IN, RESET_IN, or TOGGLE_IN is forwarded immediately as the matching event type to ASRT_OUT.

Application Scenarios

- **Merging three AE signal sources** (e.g. from ASRT_MERGE_2 or directly from sensors) into a single ASRT adapter for downstream set/reset/toggle logic such as AX_T_FF_SR.
- **Adapterizing** existing AE-based networks to connect them to ASRT interfaces.

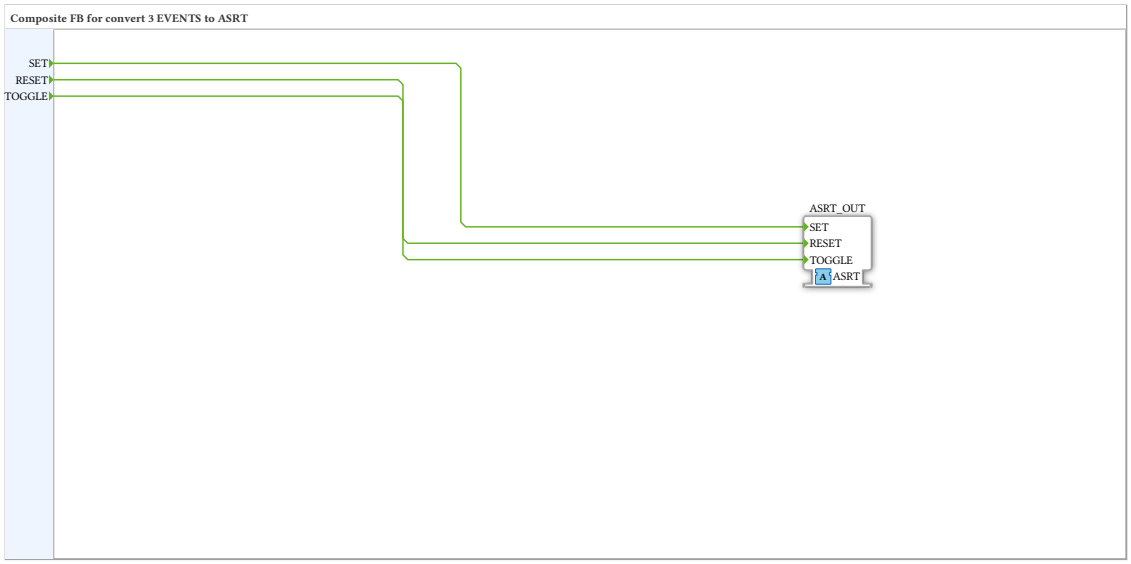
Comparison with Similar Components

- **ASRT_SRT_TO_3AE**: the reverse direction – splits an ASRT signal back into three AE adapters.
- **ASRT_3EVENTS_TO_SRT**: the same functionality with classic event inputs instead of AE adapter sockets.
- **ASR_2AE_TO_SR**: the TOGGLE-reduced variant for ASR.
- **ASRT_SR_AE_TO_SRT**: the same merge, but with SET/RESET already provided as a ready-made ASR adapter instead of two separate AE sockets.

Conclusion

ASRT_3AE_TO_SRT is a simple, pure-wiring solution for merging three AE adapters into one ASRT adapter, suitable for seamlessly integrating AE-based event sources into ASRT interfaces.

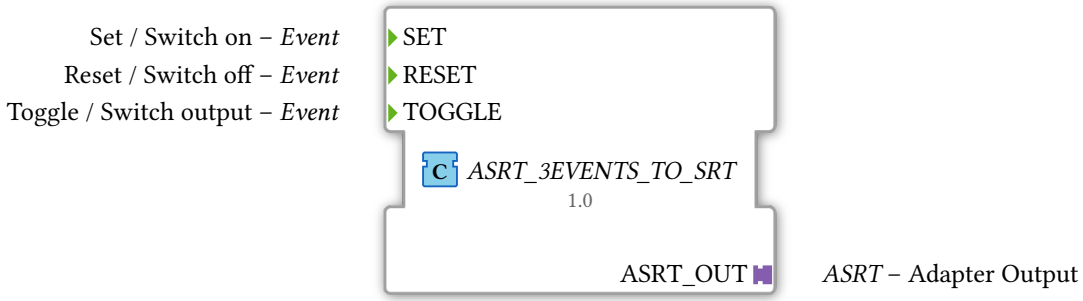
ASRT_3EVENTS_TO_SRT



• – ◦ • – ◦ • – ◦ •

Introduction

The ASRT_3EVENTS_TO_SRT function block is a composite function block that converts three separate events (Set/Reset/Toggle) into an ASRT signal. It serves as a bridge between event-based control signals and the ASRT adapter format, and is the TOGGLE-extended variant of ASR_2EVENTS_TO_SR.



Interface Structure

Event Inputs

- **SET**: Sets the output signal (power on)
- **RESET**: Resets the output signal (power off)
- **TOGGLE**: Inverts the output signal

Event Outputs

No direct event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapter

- **ASRT_OUT**: Unidirectional ASRT adapter output

Functionality

This function block forwards incoming SET, RESET, and TOGGLE events directly to the ASRT adapter. A SET event activates the corresponding SET signal on the ASRT adapter, a RESET event activates the RESET signal, and a TOGGLE event activates the TOGGLE signal.

Technical Features

- Composite function block without internal logic processing
- Direct event pass-through to the adapter
- Uses unidirectional ASRT adapter technology

State Overview

The function block has no internal state. It acts purely as a signal relay between the event inputs and the ASRT adapter.

Application Scenarios

- Integration of event-based control systems into ASRT-compatible environments
- Connection between different control levels
- Protocol implementation in automation systems

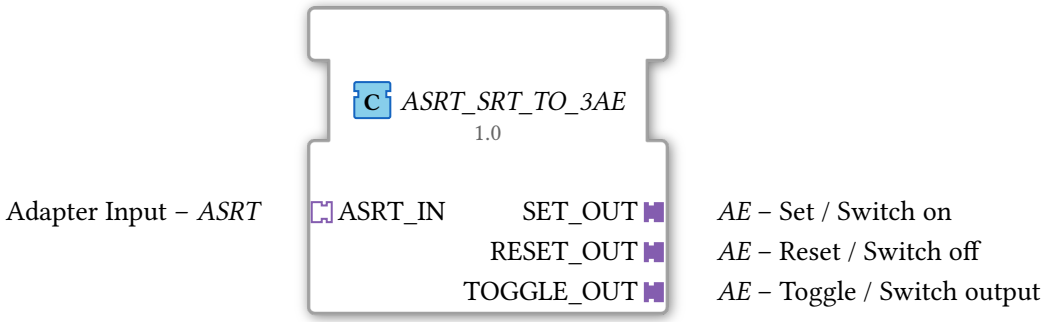
⚖️ Comparison with Similar Blocks

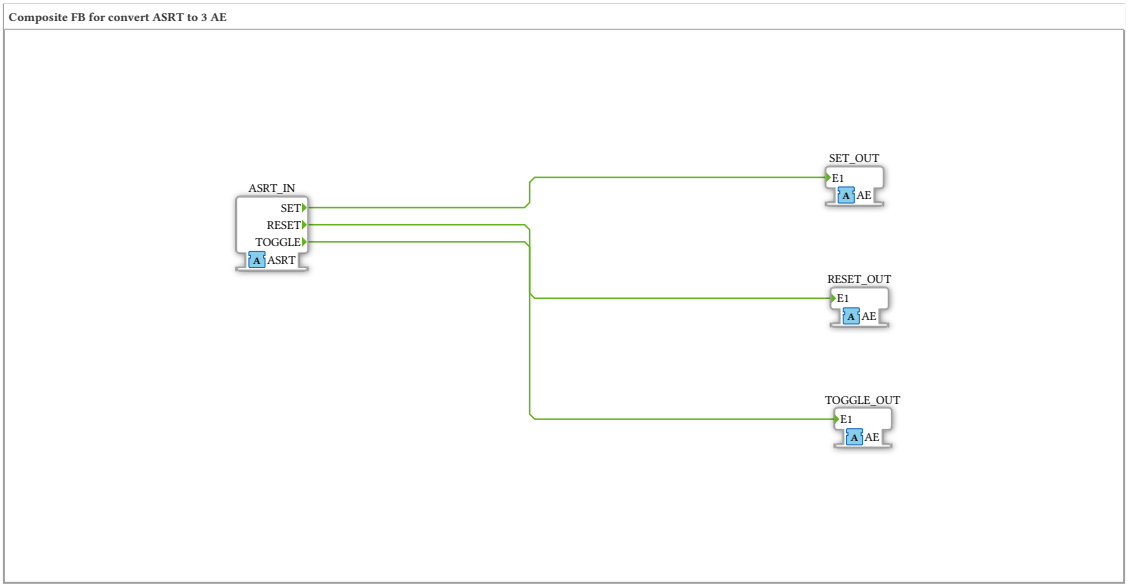
Compared to ASR_2EVENTS_TO_SR, ASRT_3EVENTS_TO_SRT additionally provides the TOGGLE event and therefore feeds an ASRT adapter instead of an ASR adapter.

Conclusion

ASRT_3EVENTS_TO_SRT is a specialized function block for converting three separate events (Set/Reset/Toggle) into the ASRT adapter format. Its simple and straightforward operation makes it a reliable solution for system integrations.

ASRT_SRT_TO_3AE





• – ○ • – ○ • – ○ •

Introduction

The function block **ASRT_SRT_TO_3AE** is a composite FB that splits a unidirectional ASRT adapter (a set/reset/toggle event triple) into three separate AE adapters (pure event, no payload data). It is the reverse of ASRT_3AE_TO_SRT and the adapter-based counterpart of ASRT_SRT_TO_3EVENTS.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter socket.

Event Outputs

None. Events are forwarded exclusively via the adapter plugs.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket	ASRT_IN	adapter::types::unidirectional::ASRT	Incoming set/reset/toggle signal.
Plug	SET_OUT	adapter::types::unidirectional::AE	Set / switch on.
Plug	RESET_OUT	adapter::types::unidirectional::AE	Reset / switch off.
Plug	TOGGLE_OUT	adapter::types::unidirectional::AE	Toggle / invert output.

Functionality

The block is a pure FBNetwork of three event connections with no algorithms of its own: ASRT_IN.SET is wired directly to SET_OUT.E1, ASRT_IN.RESET to RESET_OUT.E1, and ASRT_IN.TOGGLE to TOGGLE_OUT.E1. A SET, RESET, or TOGGLE event at the ASRT socket immediately triggers the matching event at the respective AE plug.

Technical Features

- **Pure wiring**: A composite FB with no ECC or algorithms, consisting solely of three direct event connections.
- **Adapter-based instead of event-based**: Unlike ASRT_SRT_TO_3EVENTS (classic event outputs), this block provides AE adapter plugs, letting it fit seamlessly into adapter-based networks.

State Overview

The function block has **no state machine**. Every SET, RESET, or TOGGLE event at ASRT_IN is forwarded immediately to the respective AE plug.

Application Scenarios

- **Fanning out an ASRT signal** (e.g. from AX_T_FF_SR) into three independently processable AE events.
- **Adapterizing** existing ASRT-based networks for further processing with plain AE adapters.

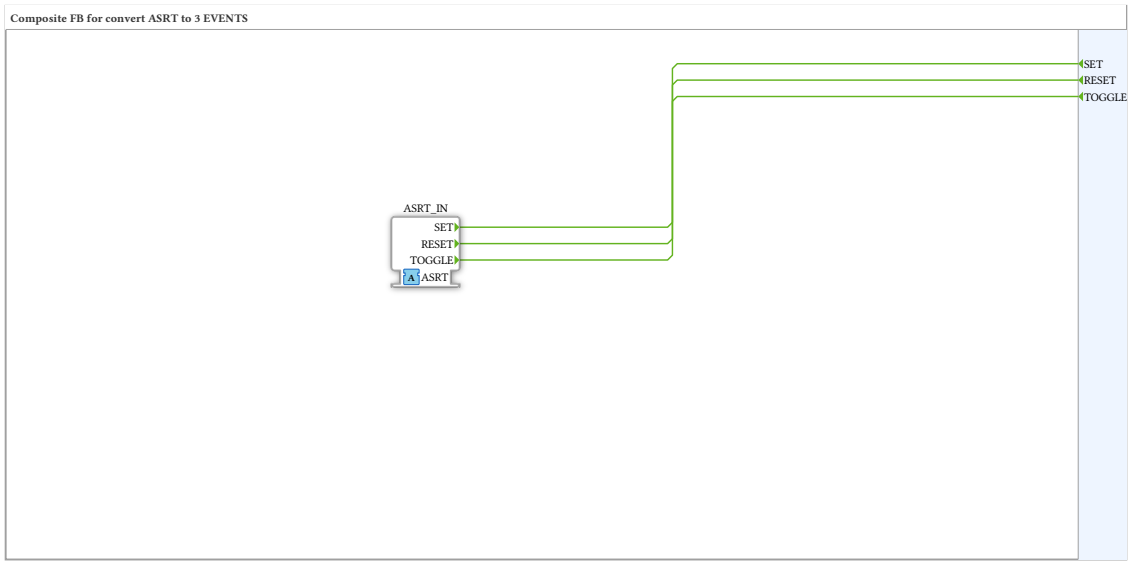
Comparison with Similar Components

- **ASRT_3AE_TO_SRT**: the reverse direction – merges three AE adapters into one ASRT signal.
- **ASRT_SRT_TO_3EVENTS**: the same functionality with classic event outputs instead of AE adapter plugs.
- **ASR_SR_TO_2AE**: the TOGGLE-reduced variant for ASR.
- **ASRT_SRT_TO_SR_AE**: the same split, but providing SET/RESET as a ready-made ASR adapter instead of two separate AE plugs.

Conclusion

ASRT_SRT_TO_3AE is a simple, pure-wiring solution for splitting an ASRT adapter into three AE adapters, suitable for seamlessly integrating ASRT signals into AE-based networks.

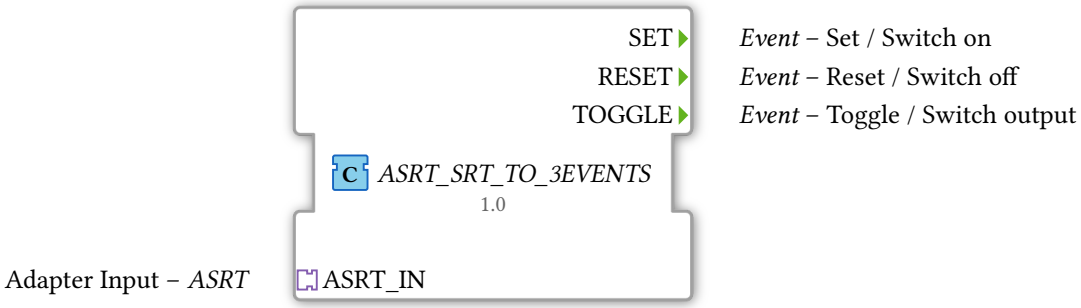
ASRT_SRT_TO_3EVENTS



• – ○ • – ○ • – ○ •

Introduction

The ASRT_SRT_TO_3EVENTS function block is a composite function block used to convert an ASRT signal into three separate events. It enables the conversion of a unidirectional adapter signal into independent SET, RESET, and TOGGLE events, and is the TOGGLE-extended variant of ASR_SR_TO_2EVENTS.



Interface Structure

Event Inputs

No direct event inputs available.

Event Outputs

- **SET**: Set / power on
- **RESET**: Reset / power off
- **TOGGLE**: Inverts the output signal

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

- **ASRT_IN**: Adapter input of type `adapter::types::unidirectional::ASRT`

Functionality

This function block forwards the SET, RESET, and TOGGLE signals of the ASRT adapter directly to the corresponding event outputs. The incoming adapter events are passed through transparently to the outputs of the function block.

Technical Features

- Composite function block without internal logic processing
- Direct connection between adapter events and output events
- No data processing or state management

State Overview

The function block has no internal state and operates statelessly. Every incoming adapter event is immediately forwarded to the corresponding output.

Application Scenarios

- Integration of ASRT adapters into event-based controls
- Conversion of unidirectional adapter signals into independent individual events
- Interface adaptation between different communication protocols

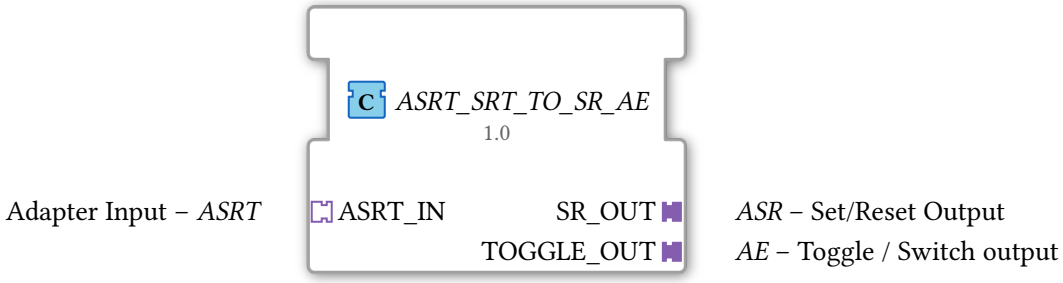
⚖️ Comparison with Similar Blocks

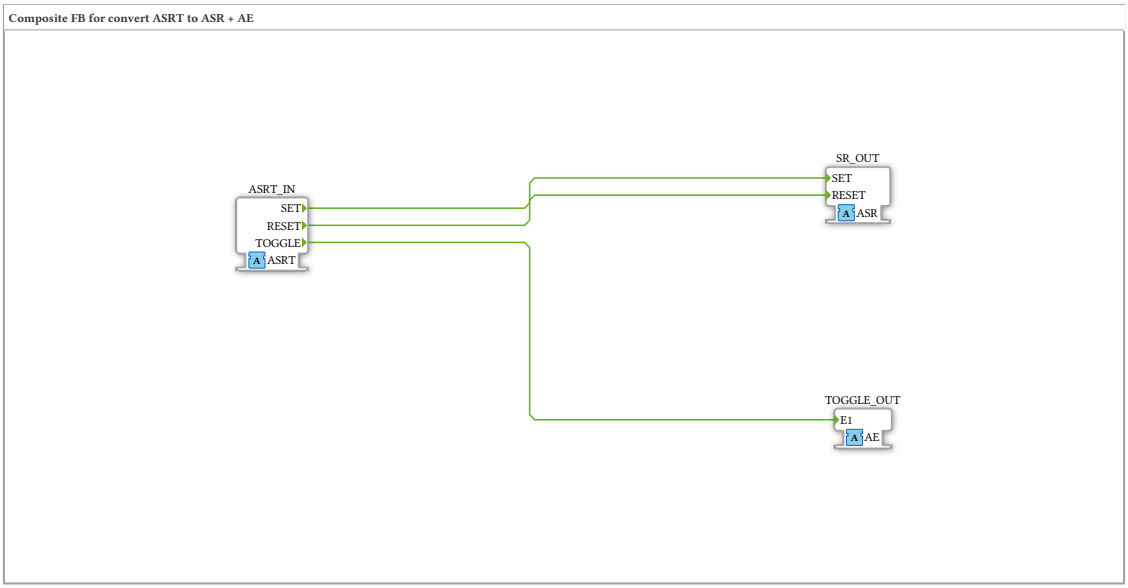
Compared to ASR_SR_TO_2EVENTS, this composite function block additionally handles the TOGGLE signal of an ASRT adapter instead of an ASR adapter.

Conclusion

The ASRT_SRT_TO_3EVENTS function block provides a simple and efficient solution for converting ASRT adapter signals into separate SET, RESET, and TOGGLE events. Its lean implementation without internal logic makes it a reliable and efficient component for interface adaptation.

ASRT_SRT_TO_SR_AE





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_SRT_TO_SR_AE** is a composite FB that splits a unidirectional ASRT adapter (set/reset/toggle) into an ASR adapter (set/reset) and a separate AE adapter (toggle, pure event). It is the reverse of ASRT_SR_AE_TO_SRT and a mixed variant of ASRT_SRT_TO_3AE.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter socket.

Event Outputs

None. Events are forwarded exclusively via the adapter plugs.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket	`ASRT_IN`	`adapter::types::unidirectional::ASRT`	Incoming set/reset/toggle signal.
Plug	`SR_OUT`	`adapter::types::unidirectional::ASR`	Set/reset output.
Plug	`TOGGLE_OUT`	`adapter::types::unidirectional::AE`	Toggle / invert output.

Functionality

The block is a pure FBNetwork of three event connections with no algorithms of its own: `ASRT_IN.SET` is wired directly to `SR_OUT.SET`, `ASRT_IN.RESET` to `SR_OUT.RESET`, and `ASRT_IN.TOGGLE` to `TOGGLE_OUT.E1`. A SET, RESET, or TOGGLE event at the ASRT socket immediately triggers the matching event at `SR_OUT` or `TOGGLE_OUT`.

Technical Features

- **Pure wiring**: A composite FB with no ECC or algorithms, consisting solely of three direct event connections.
- **Mixed adapter outputs**: Provides SET/RESET already as a ready-made ASR adapter instead of two individual AE plugs — useful when downstream logic already expects an ASR input (e.g. `ASR_AX_SR`) and only TOGGLE needs to be handled separately.

State Overview

The function block has **no state machine**. Every `SET`, `RESET`, or `TOGGLE` event at `ASRT_IN` is forwarded immediately to `SR_OUT` or `TOGGLE_OUT`.

Application Scenarios

- **Connecting to existing ASR interfaces**: An ASRT signal needs to be forwarded to a block that only expects SET/RESET as an ASR adapter, while TOGGLE is tapped off separately (e.g. for diagnostics).
- **Simplifying downstream logic** that already processes set/reset bundled as an ASR adapter.

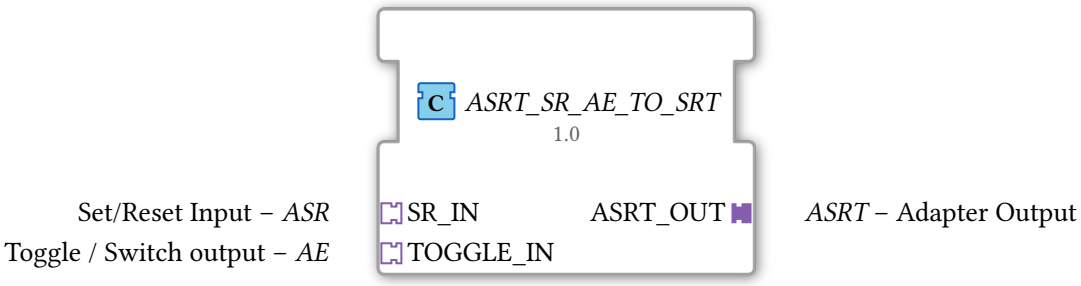
Comparison with Similar Components

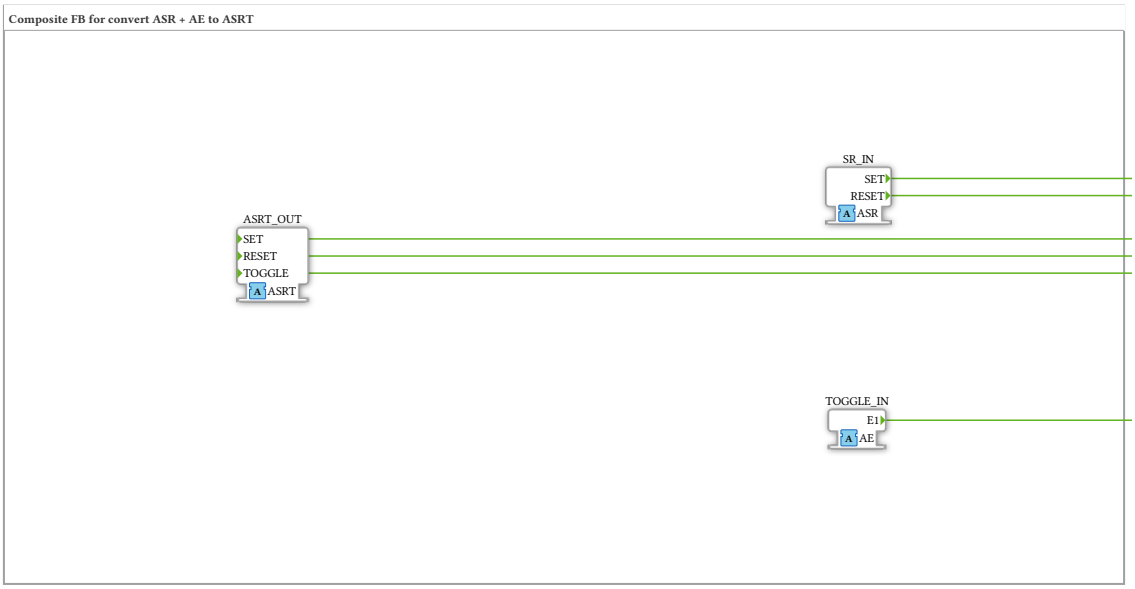
- **ASRT_SR_AE_TO_SRT**: the reverse direction – merges ASR (SET/RESET) and AE (TOGGLE) into one ASRT signal.
- **ASRT_SRT_TO_3AE**: the same split, but with three individual AE plugs instead of a ready-made ASR adapter for SET/RESET.
- **ASR_SR_TO_2AE**: the `TOGGLE`-reduced variant for ASR.

Conclusion

`ASRT_SRT_TO_SR_AE` is a simple, pure-wiring solution that splits an ASRT signal into an ASR adapter (SET/RESET) and a separate toggle event.

ASRT_SR_AE_TO_SRT





• – ○ • – ○ • – ○ •

Introduction

The function block **ASRT_SR_AE_TO_SRT** is a composite FB that merges a unidirectional ASR adapter (set/reset) and a separate AE adapter (toggle, pure event) into one common ASRT adapter (set/reset/toggle). It is a mixed variant of ASRT_3AE_TO_SRT: instead of three individual AE sockets for SET, RESET, and TOGGLE, it accepts SET/RESET already as a ready-made ASR adapter and only adds TOGGLE separately.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket	SR_IN	adapter::types::unidirectional::ASR	Incoming set/reset signal.
Socket	TOGGLE_IN	adapter::types::unidirectional::AE	Toggle / invert output.
Plug	ASRT_OUT	adapter::types::unidirectional::ASRT	Merged set/reset/toggle signal.

Functionality

The block is a pure FBNetwork of three event connections with no algorithms of its own: SR_IN.SET is wired directly to ASRT_OUT.SET, SR_IN.RESET to ASRT_OUT.RESET, and TOGGLE_IN.E1 to ASRT_OUT.TOGGLE. A SET or RESET event at the ASR socket, or an event at the AE socket, immediately triggers the matching event at the ASRT plug.

Technical Features

- **Pure wiring**: A composite FB with no ECC or algorithms, consisting solely of three direct event connections.
- **Mixed adapter inputs**: Combines an already-complete ASR adapter (SET/RESET) with a separate AE adapter (TOGGLE) — useful when SET/RESET already exist as an ASR signal (e.g. from ASR_2AE_TO_SR or an existing ASR network branch) and only TOGGLE needs to be added.

State Overview

The function block has **no state machine**. Every event at SR_IN (SET/RESET) or TOGGLE_IN is forwarded immediately as the matching event type to ASRT_OUT.

Application Scenarios

- **Retrofitting TOGGLE**: An existing ASR network should be extended with a toggle function without rewiring SET/RESET.
- **Reusing existing ASR signal paths** in combination with a separate toggle source, e.g. a button via ASR_MERGE_2.

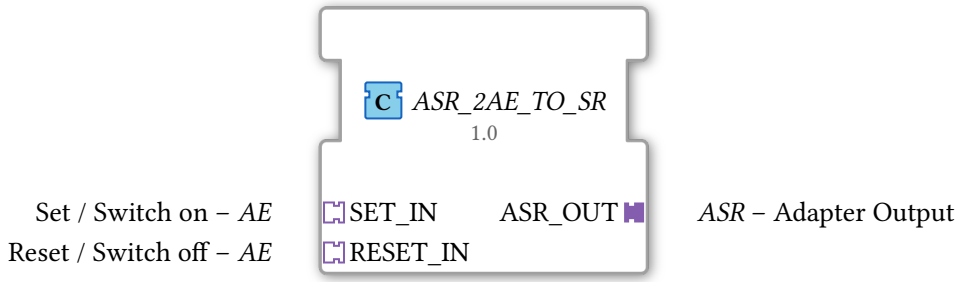
Comparison with Similar Components

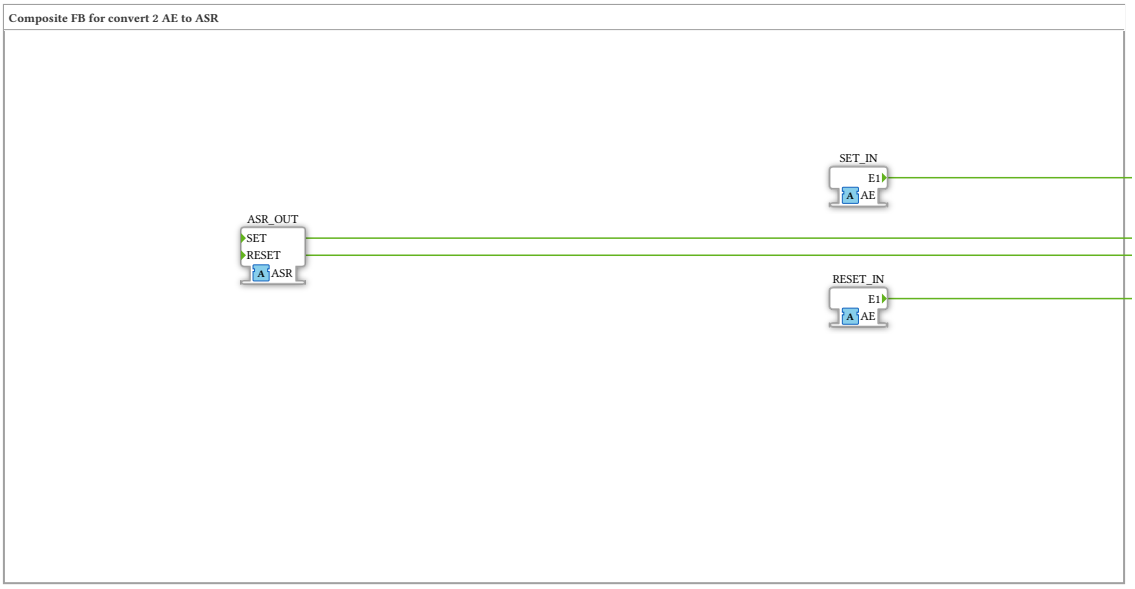
- **ASRT_SRT_TO_SR_AE**: the reverse direction – splits an ASRT signal back into ASR (SET/RESET) and AE (TOGGLE).
- **ASRT_3AE_TO_SRT**: the same merge, but with three individual AE sockets instead of a ready-made ASR adapter for SET/RESET.
- **ASR_2AE_TO_SR**: the TOGGLE-reduced variant for ASR.

Conclusion

ASRT_SR_AE_TO_SRT is a simple, pure-wiring solution that extends an existing ASR adapter with a separate toggle event into a complete ASRT adapter.

ASR_2AE_TO_SR





• – ○ • – ○ • – ○ •

Introduction

The function block **ASR_2AE_TO_SR** is a composite FB that merges two separate unidirectional AE adapters (pure event, no payload data) into one common ASR adapter (a set/reset event pair). It is the adapter-based counterpart of ASR_2EVENTS_TO_SR, but uses AE adapter sockets instead of classic event inputs.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket	`SET_IN`	`adapter::types::unidirectional::AE`	Set / switch on.
Socket	`RESET_IN`	`adapter::types::unidirectional::AE`	Reset / switch off.
Plug	`ASR_OUT`	`adapter::types::unidirectional::ASR`	Merged set/reset signal.

Functionality

The block is a pure FBNetwork of two event connections with no algorithms of its own: `SET_IN.E1` is wired directly to `ASR_OUT.SET`, `RESET_IN.E1` directly to `ASR_OUT.RESET`. An event at the respective AE socket immediately triggers the matching SET or RESET event at the ASR plug.

Technical Features

- **Pure wiring**: A composite FB with no ECC or algorithms, consisting solely of two direct event connections.

- **Adapter-based instead of event-based**: Unlike ASR_2EVENTS_TO_SR (classic event inputs SET/RESET), this block uses AE adapter sockets, letting it fit seamlessly into adapter-based networks.

State Overview

The function block has **no state machine**. Every event at SET_IN or RESET_IN is forwarded immediately as SET or RESET to ASR_OUT.

Application Scenarios

- **Merging two AE signal sources** (e.g. from ASR_MERGE_2 or directly from a sensor) into a single ASR adapter for downstream set/reset logic.
- **Adapterizing** existing AE-based networks to connect them to ASR interfaces (e.g. AX_SR).

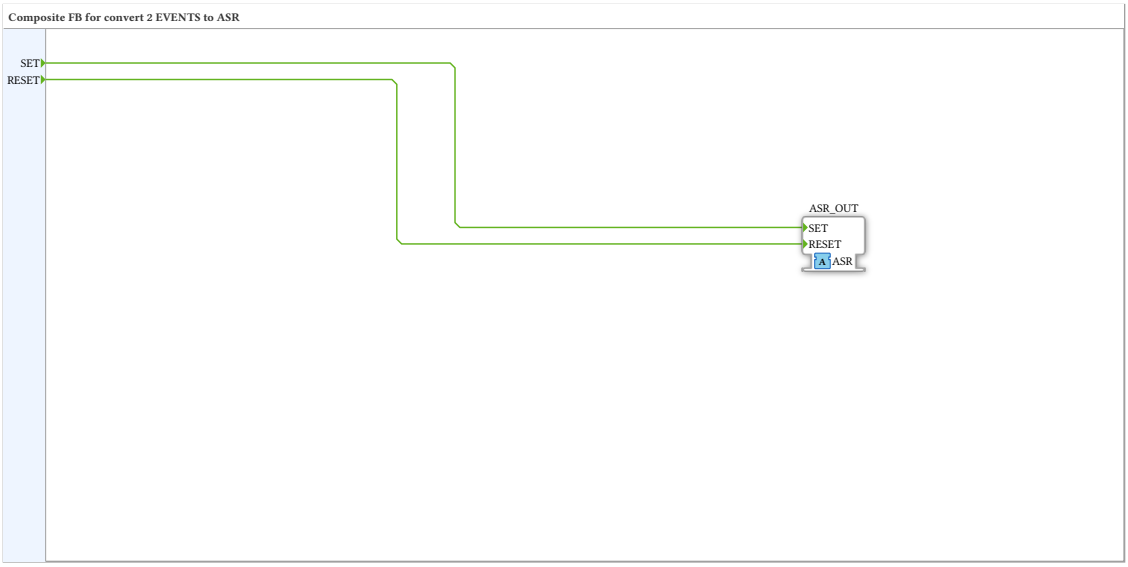
Comparison with Similar Components

- **ASR_SR_TO_2AE**: the reverse direction – splits an ASR signal back into two AE adapters.
- **ASR_2EVENTS_TO_SR**: the same functionality with classic event inputs instead of AE adapter sockets.
- **ASRT_3AE_TO_SRT**: the TOGGLE-extended variant for ASRT.

Conclusion

ASR_2AE_TO_SR is a simple, pure-wiring solution for merging two AE adapters into one ASR adapter, suitable for seamlessly integrating AE-based event sources into ASR interfaces.

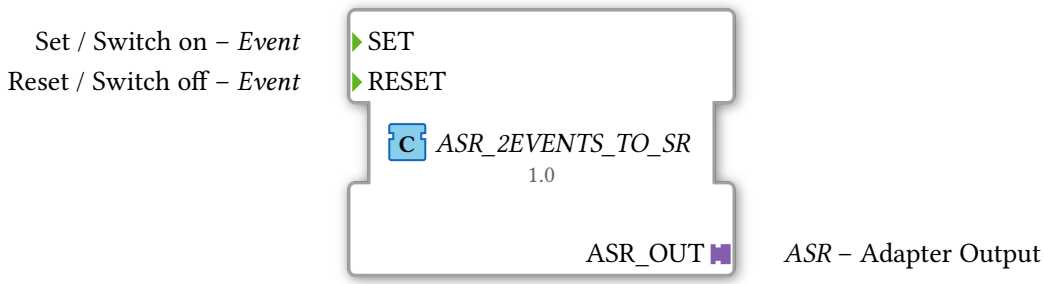
ASR_2EVENTS_TO_SR



• – ◦ • – ◦ • – ◦ •

Introduction

The ASR_2EVENTS_TO_SR function block is a composite function block that converts two separate events into an ASR (Set-Reset) signal. It serves as a bridge between event-based control signals and the ASR adapter format.



Interface Structure

Event Inputs
- **SET**: Sets the output signal (power on)
- **RESET**: Resets the output signal (power off)

Event Outputs
No direct event outputs available

Data Inputs
No data inputs available

Data Outputs
No data outputs available

Adapter
- **ASR_OUT**: Unidirectional ASR adapter output

Functionality
This function block forwards incoming SET and RESET events directly to the ASR adapter. A SET event activates the corresponding SET signal on the ASR adapter, while a RESET event activates the RESET signal.

Technical Features
- Composite function block without internal logic processing
- Direct event pass-through to the adapter
- Uses unidirectional ASR adapter technology

State Overview
The function block has no internal state. It acts purely as a signal relay between the event inputs and the ASR adapter.

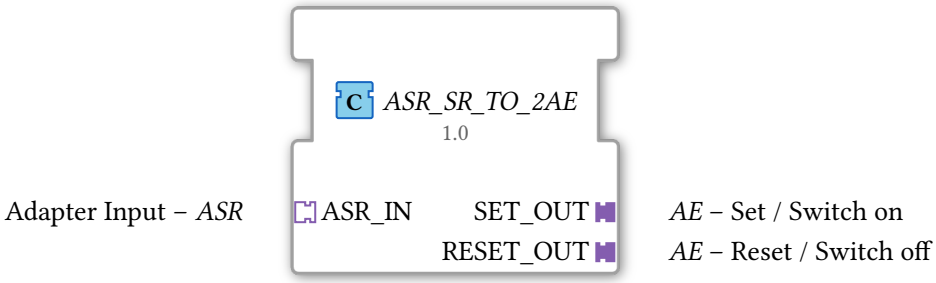
Application Scenarios
- Integration of event-based control systems into ASR-compatible environments
- Connection between different control levels
- Protocol implementation in automation systems

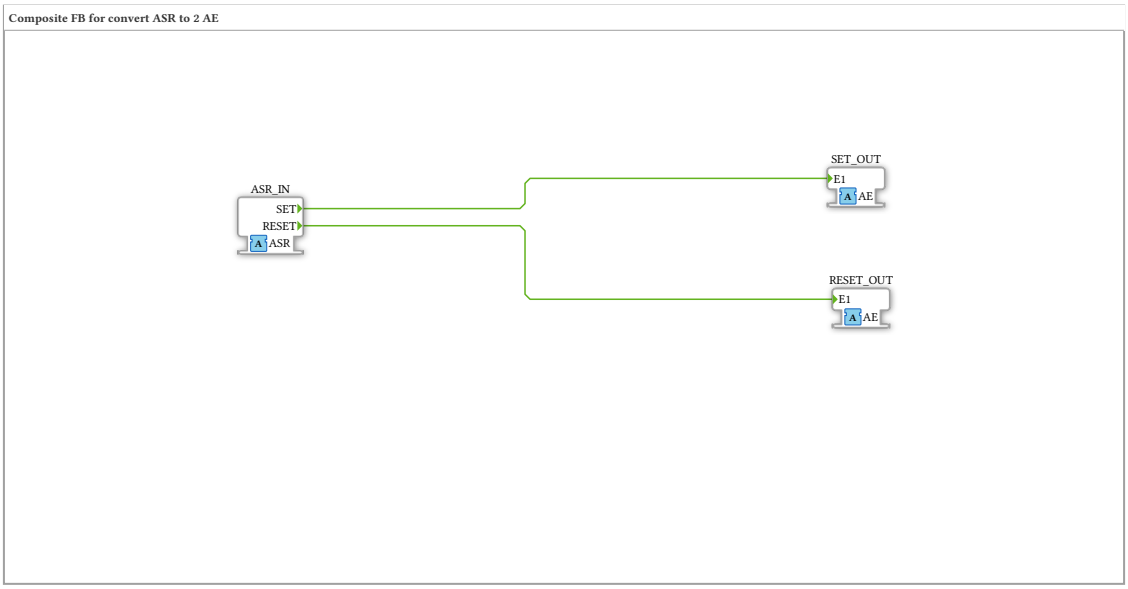
⚖️ Comparison with Similar Blocks
Compared to simple event function blocks, ASR_2EVENTS_TO_SR offers a specific interface to the ASR adapter, while other blocks may use direct data outputs.

🛠️ Related Exercises
- Exercise_171_AX

Conclusion
ASR_2EVENTS_TO_SR is a specialized function block for converting two separate events into the ASR adapter format. Its simple and straightforward operation makes it a reliable solution for system integrations.

ASR_SR_TO_2AE





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_SR_TO_2AE** is a composite FB that splits a unidirectional ASR adapter (a set/reset event pair) into two separate AE adapters (pure event, no payload data). It is the reverse of ASR_2AE_TO_SR and the adapter-based counterpart of ASR_SR_TO_2EVENTS.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter socket.

Event Outputs

None. Events are forwarded exclusively via the adapter plugs.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket	`ASR_IN`	`adapter::types::unidirectional::ASR`	Incoming set/reset signal.
Plug	`SET_OUT`	`adapter::types::unidirectional::AE`	Set / switch on.

Plug	RESET_OUT	adapter::types::unidirectional::AE	Reset / switch off.

Functionality

The block is a pure FBNetwork of two event connections with no algorithms of its own: ASR_IN.SET is wired directly to SET_OUT.E1, ASR_IN.RESET directly to RESET_OUT.E1. A SET or RESET event at the ASR socket immediately triggers the matching event at the respective AE plug.

Technical Features

- **Pure wiring**: A composite FB with no ECC or algorithms, consisting solely of two direct event connections.
- **Adapter-based instead of event-based**: Unlike ASR_SR_TO_2EVENTS (classic event outputs), this block provides AE adapter plugs, letting it fit seamlessly into adapter-based networks.

State Overview

The function block has **no state machine**. Every SET or RESET event at ASR_IN is forwarded immediately to SET_OUT or RESET_OUT.

Application Scenarios

- **Fanning out an ASR signal** into two independently processable AE events, e.g. to route SET and RESET separately to different downstream blocks.
- **Adapterizing** existing ASR-based networks for further processing with plain AE adapters.

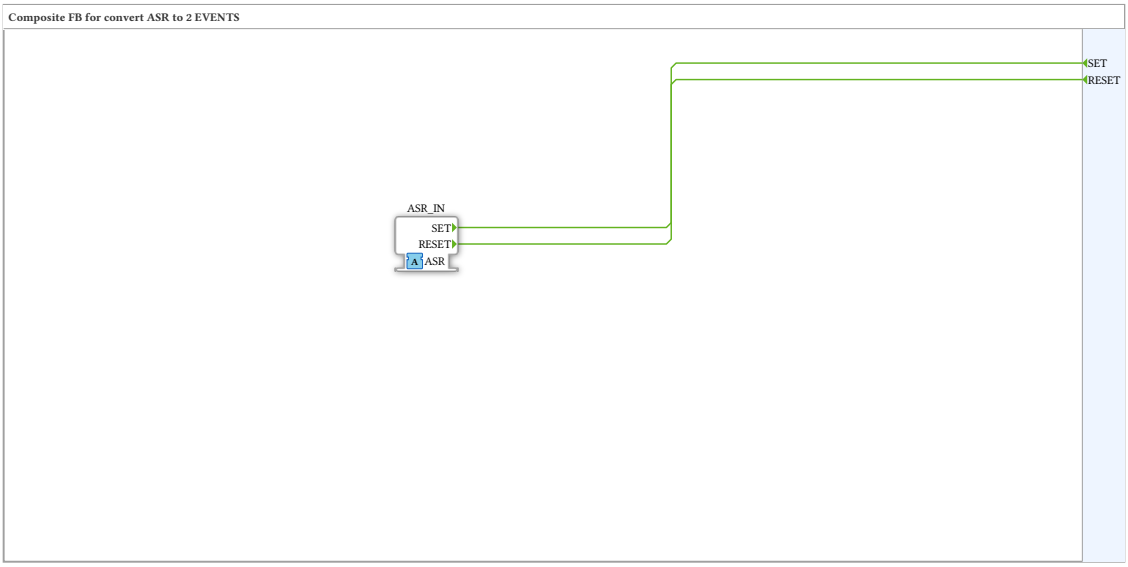
Comparison with Similar Components

- **ASR_2AE_TO_SR**: the reverse direction – merges two AE adapters into one ASR signal.
- **ASR_SR_TO_2EVENTS**: the same functionality with classic event outputs instead of AE adapter plugs.
- **ASRT_SRT_TO_3AE**: the TOGGLE-extended variant for ASRT.

Conclusion

ASR_SR_TO_2AE is a simple, pure-wiring solution for splitting an ASR adapter into two AE adapters, suitable for seamlessly integrating ASR signals into AE-based networks.

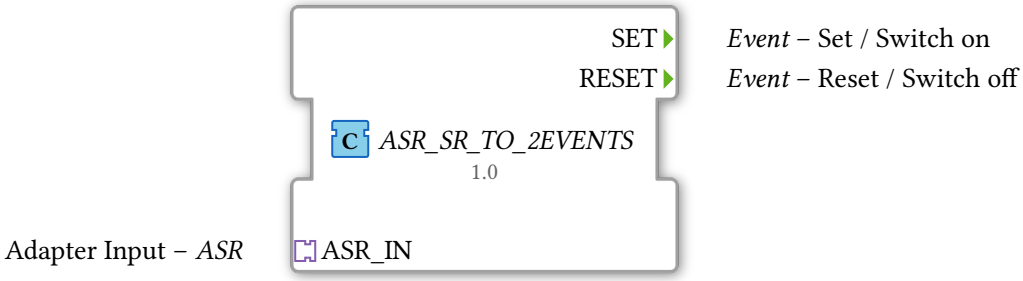
ASR_SR_TO_2EVENTS



• – ○ • – ○ • – ○ •

Introduction

The ASR_SR_TO_2EVENTS function block is a composite function block used to convert an ASR signal into two separate events. It enables the conversion of a bidirectional adapter signal into independent SET and RESET events.



Interface Structure

Event Inputs

No direct event inputs are available.

Event Outputs

- **SET**: Set / Turn On
- **RESET**: Reset / Turn Off

Data Inputs

No data inputs are available.

Data Outputs

No data outputs are available.

Adapter

- **ASR_IN**: Adapter input of type `adapter::types::unidirectional::ASR`

Functionality

The function block forwards the SET and RESET signals of the ASR adapter directly to the corresponding event outputs. The incoming adapter events are transparently passed through to the outputs of the function block.

Technical Features

- Composite function block without internal logic processing
- Direct connection between adapter events and output events
- No data processing or state management

State Overview

The function block has no internal state and operates stateless. Every incoming adapter event is immediately forwarded to the corresponding output.

Application Scenarios

- Integration of ASR adapters into event-based controllers
- Conversion of bidirectional signals into independent single events
- Interface adaptation between different communication protocols

⚖️ Comparison with similar components

Compared to simple adapter blocks, this composite function block offers specific conversion of ASR to event signals, while generic adapters enable more general interface adaptation.

Conclusion

The ASR_SR_TO_2EVENTS function block provides a simple and efficient solution for converting ASR adapter signals into separate SET and RESET events. Its streamlined implementation without internal logic makes it a reliable and high-performance component for interface adaptation.

Unidirectional Event Adapter Conversion

🎧 Podcast

- E_TOF vs. FB_TOF: The Event Timer That Doesn't Tick Cyclically – A Revolution for Automation Systems?
- Event FBs According to IEC 61499: Building Blocks of Automation – How Event Function Blocks Work
- IEC 61499: Revolution in Automation – Event-Driven FBs and Distributed Systems Explained
- E_REND: Event Synchronization in IEC 61499
- The Unstoppable Counter: Why IEC 61499's ECTU Guarantees Safe, Event-Driven Control (and Never Overflows)

Numerical Index (unidirectional)

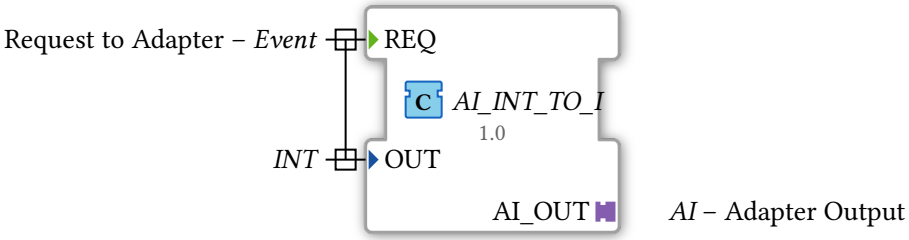
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AI_INT_TO_I



Introduction

The AI_INT_TO_I function block is a composite function block used to convert an INT value to the AI adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: INT input value to be converted

Data Outputs

- No direct data outputs available

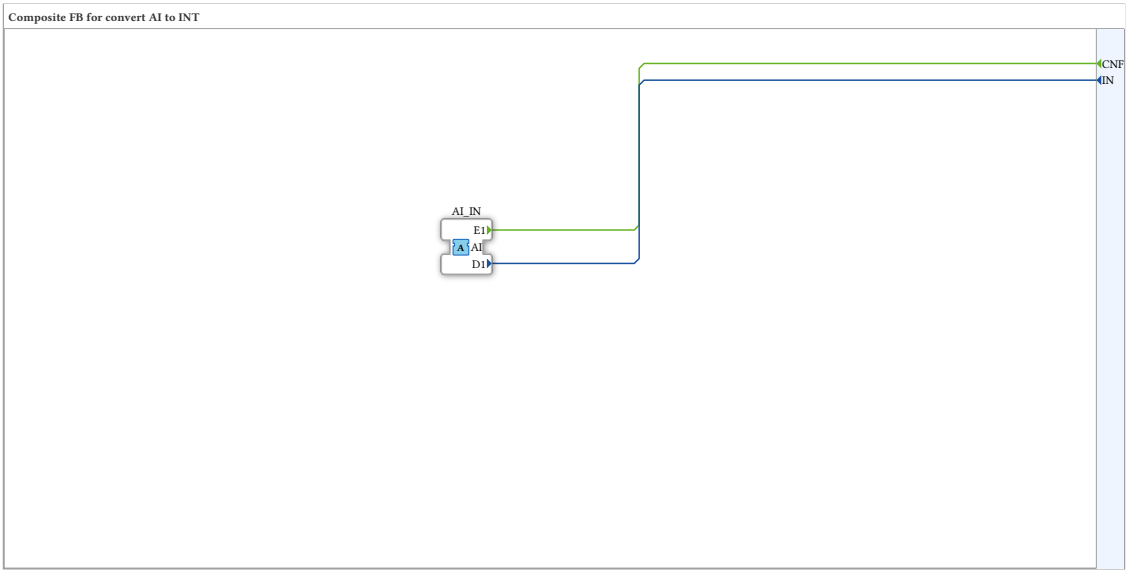
Adapter

- **AI_OUT**: Plug of type `AI`

Functionality

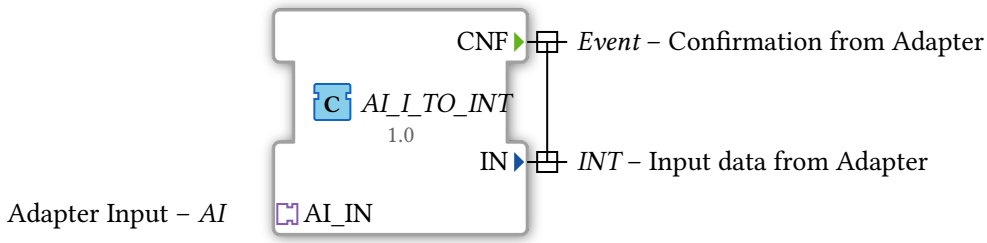
The function block receives an INT value at the data input OUT and converts it to the AI adapter format upon arrival of the REQ event.

AI_I_TO_INT



Introduction

The AI_I_TO_INT function block is a composite function block used to convert the AI adapter format to an INT value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: INT output value of the converted data

Adapter

- **AI_IN**: Socket of type `AI`

Functionality

The function block receives data via the AI adapter socket and makes the contained INT value available at the IN output as soon as an event arrives via the adapter (CNF).

INT Conversion

🎧 Podcast

- Eclipse 4diac 3.0: ST Interpreter, FBE, and 7200 Commits – The Turbo for Distributed Automation
- From PLC to Distributed Intelligence: Understanding IEC 61499 for Industry 4.0
- Simplifying Industrial Control: Your Deep Dive into Eclipse 4diac and IEC 61499
- Events and IEC 61499: The Starting Signal for Intelligent Systems
- ERP Systems: The Digital Brain – Opportunities, Risks, and the Path to Intelligent Production

🛠️ Related Exercises

- SR_TO_QX_TRIG_EO
- Uebung_039_sub_NumbAnsicht
- Softkey_T_FF_ILOCK_TO_QX_BG
- Bool8_TO_logiBUS_QX8

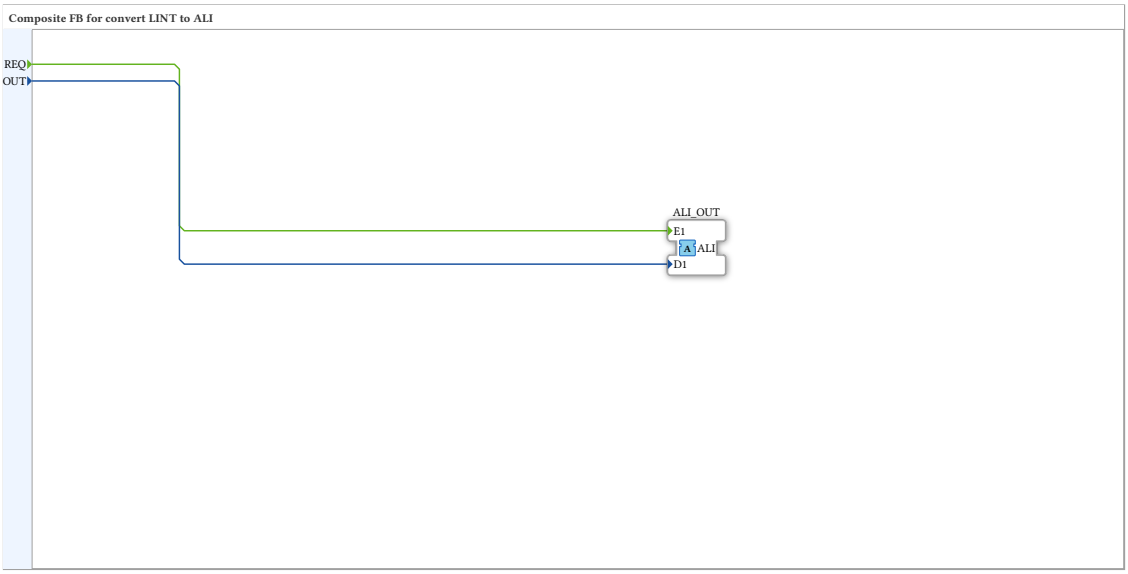
🌐 Matching topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numerical Index (unidirectional)

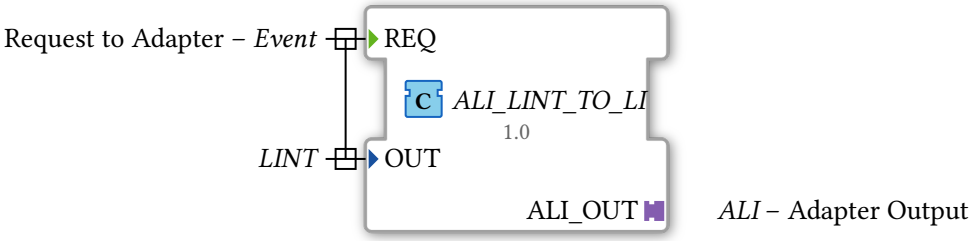
Here you will find all exercises from the `unidirectional` workspace in ascending order.

ALI_LINT_TO_LI



Introduction

The ALI_LINT_TO_LI function block is a composite function block used to convert a LINT value to the ALI adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: LINT input value to be converted

Data Outputs

- No direct data outputs available

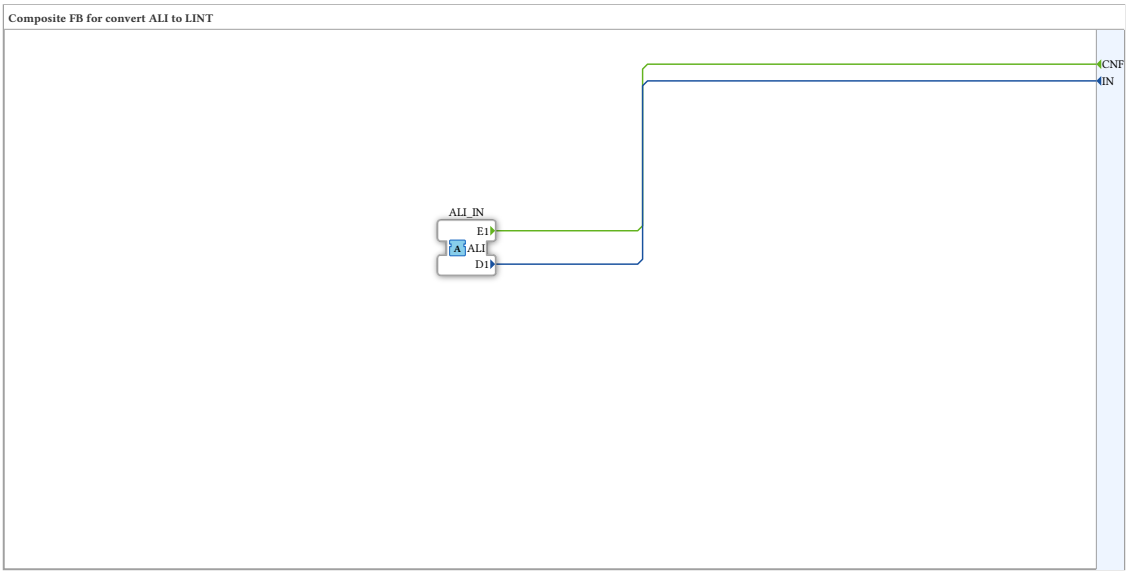
Adapter

- **ALI_OUT**: Plug of type `ALI`

Functionality

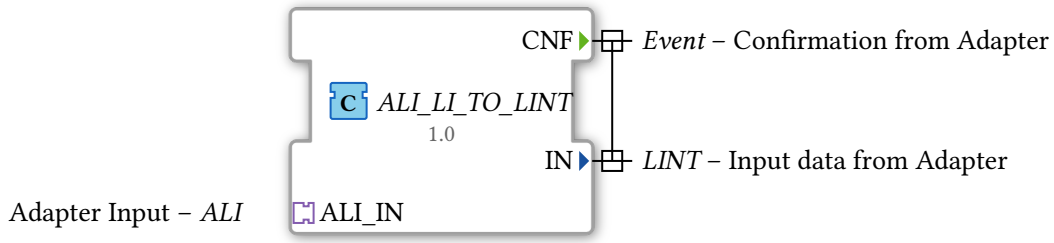
The function block receives a LINT value at the OUT data input and converts it to the ALI adapter format upon arrival of the REQ event.

ALI_LI_TO_LINT



Introduction

The ALI_LI_TO_LINT function block is a composite function block used to convert the ALI adapter format to a LINT value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: LINT output value of the converted data

Adapter

- **ALI_IN**: Socket of type `ALI`

Functionality

The function block receives data via the ALI adapter socket and makes the contained LINT value available at the IN output as soon as an event arrives via the adapter (CNF).

LINT conversion

🛠️ Related exercises

- Uebung_072c

Numerical Index (unidirectional)

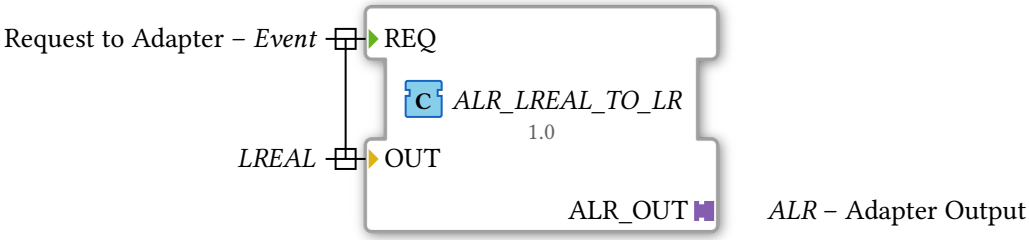
Here you will find all exercises from the `unidirectional` workspace in ascending order.

ALR_LREAL_TO_LR



Introduction

The ALR_LREAL_TO_LR function block is a composite function block used to convert an LREAL value to the ALR adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: LREAL input value to be converted

Data Outputs

- No direct data outputs available

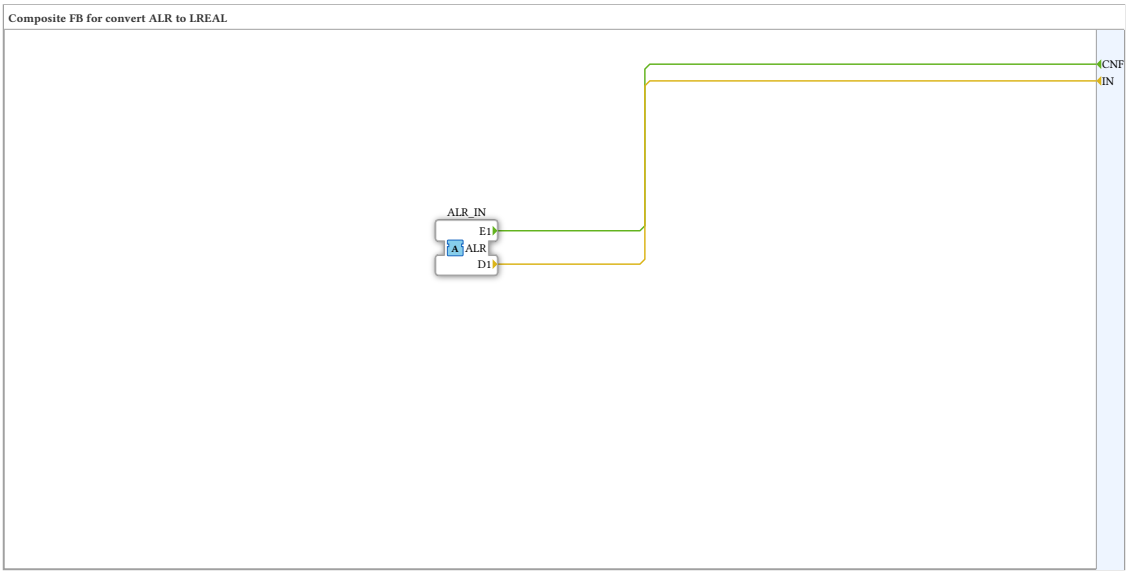
Adapter

- **ALR_OUT**: Plug of type `ALR`

Functionality

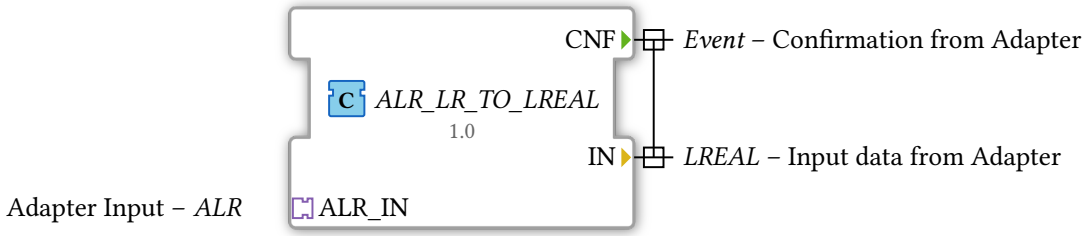
The function block receives an LREAL value at the OUT data input and converts it to the ALR adapter format upon arrival of the REQ event.

ALR_LR_TO_LREAL



Introduction

The ALR_LR_TO_LREAL function block is a composite function block used to convert the ALR adapter format to an LREAL value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: LREAL output value of the converted data

Adapter

- **ALR_IN**: Socket of type `ALR`

Functionality

The function block receives data via the ALR adapter socket and makes the contained LREAL value available at the IN output as soon as an event arrives via the adapter (CNF).

LREAL Conversion

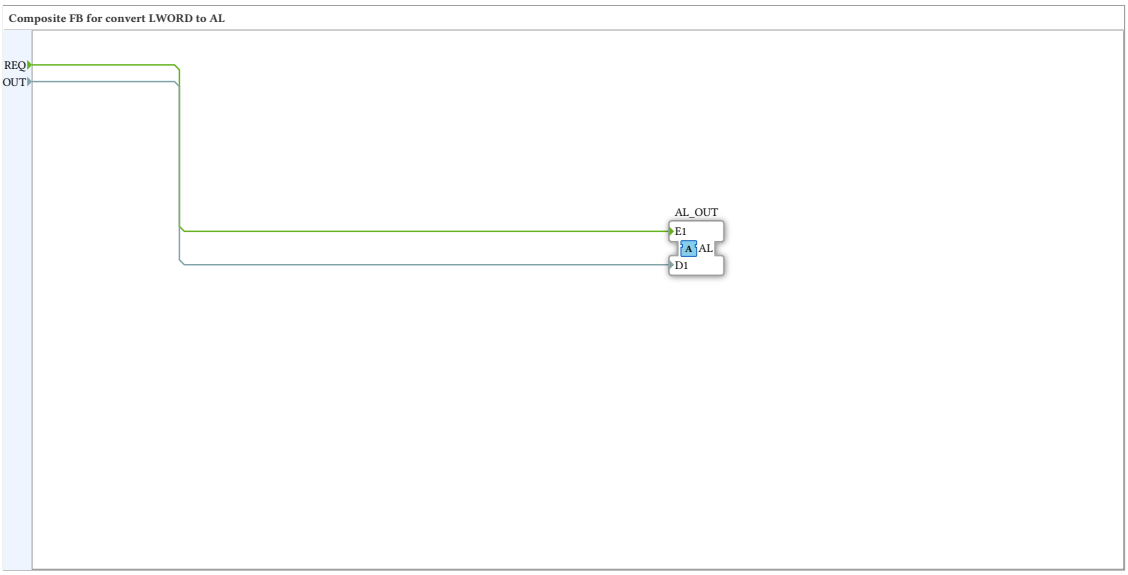
Related Exercises

- Uebung_003a0
- Uebung_003a0_AX
- Uebung_010c2
- Uebung_010c2_AX
- Uebung_035a
- Uebung_035a2
- Uebung_035a3

Numerical Index (unidirectional)

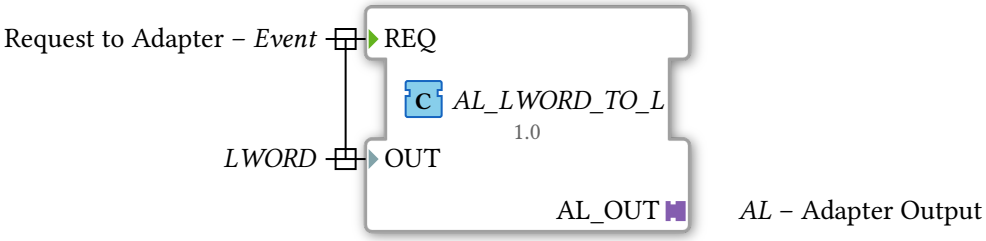
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AL_LWORD_TO_L



Introduction

The AL_LWORD_TO_L function block is a composite function block used to convert an LWORD value to the AL adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: LWORD input value to be converted

Data Outputs

- No direct data outputs available

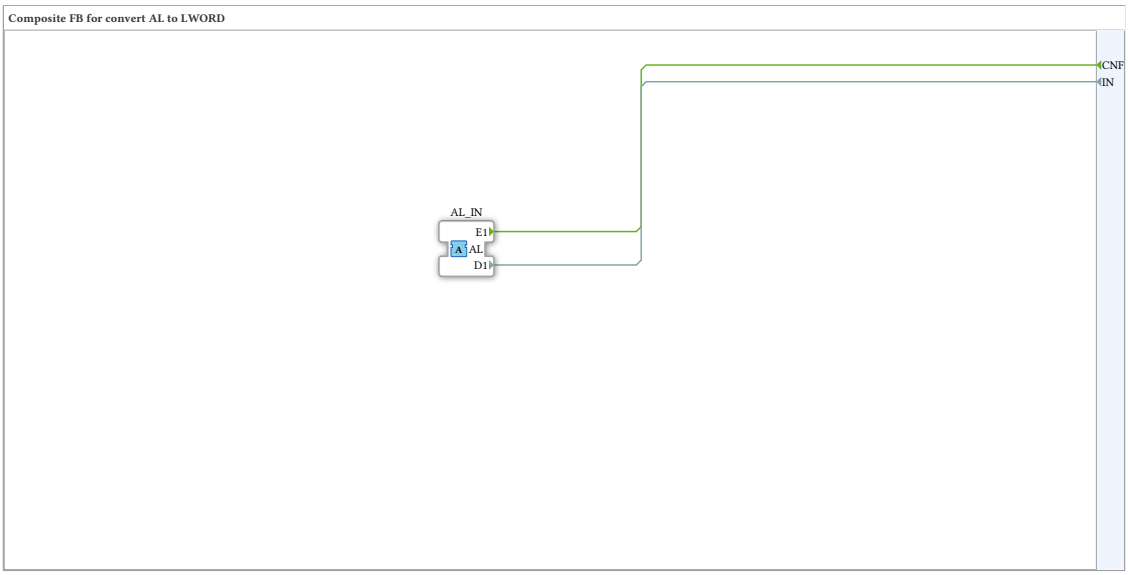
Adapter

- **AL_OUT**: Plug of type `AL`

Functionality

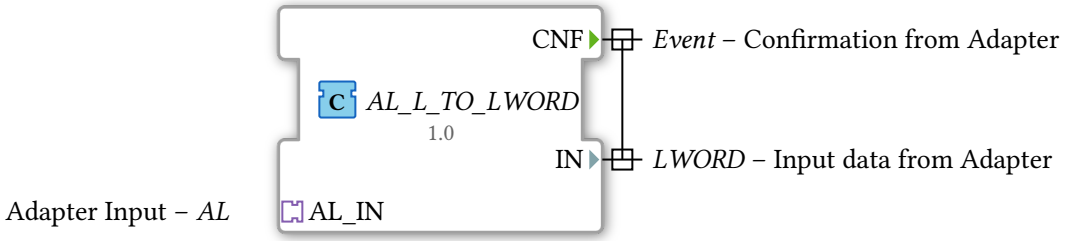
The function block receives an LWORD value at the OUT data input and converts it to the AL adapter format upon arrival of the REQ event.

AL_L_TO_LWORD



Introduction

The AL_L_TO_LWORD function block is a composite function block used to convert the AL adapter format to an LWORD value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: LWORD output value of the converted data

Adapter

- **AL_IN**: Socket of type `AL`

Functionality

The function block receives data via the AL adapter socket and makes the contained LWORD value available at the IN output as soon as an event arrives via the adapter (CNF).

LWORD Conversion

Numerical Index (unidirectional)

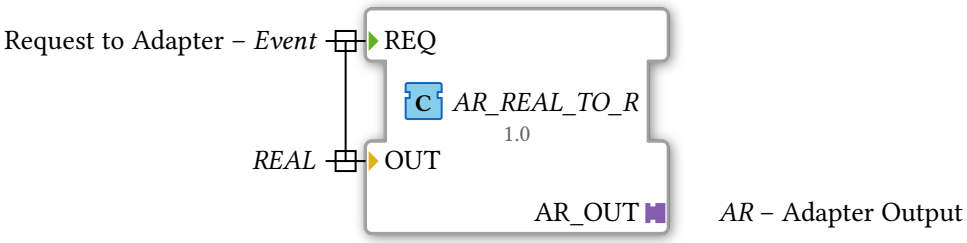
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AR_REAL_TO_R



Introduction

The AR_REAL_TO_R function block is a composite function block used to convert a REAL value to the AR adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: REAL input value to be converted

Data Outputs

- No direct data outputs available

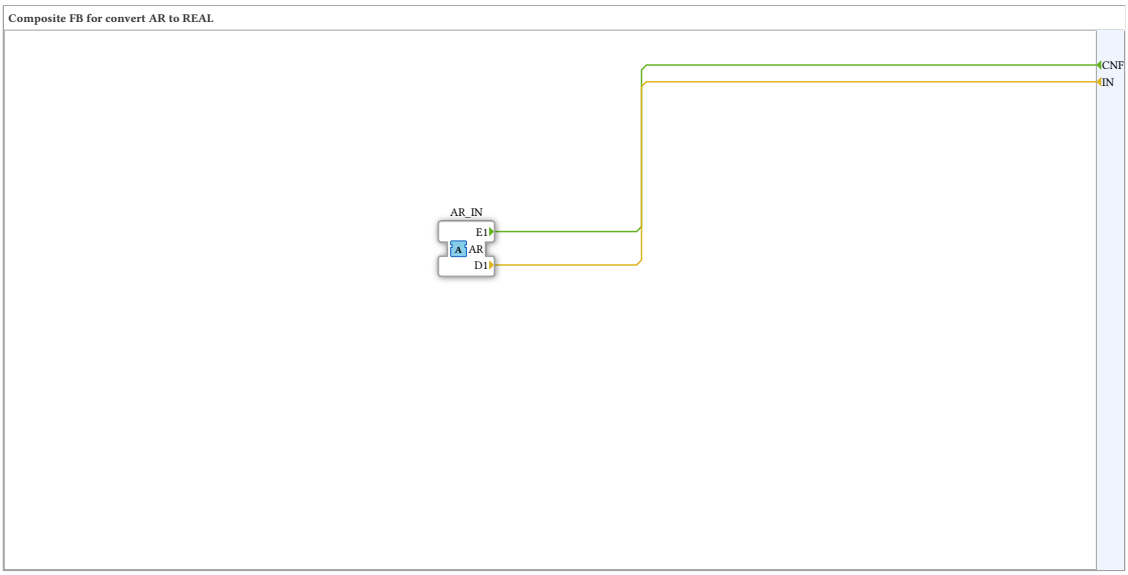
Adapter

- **AR_OUT**: Plug of type `AR`

Functionality

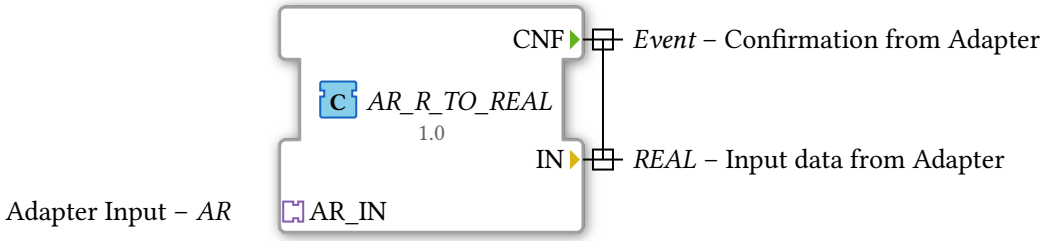
The function block receives a REAL value at the data input OUT and converts it to the AR adapter format upon arrival of the REQ event.

AR_R_TO_REAL



Introduction

The AR_R_TO_REAL function block is a composite function block used to convert the AR adapter format into a REAL value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: REAL output value of the converted data

Adapter

- **AR_IN**: Socket of type `AR`

Functionality

The function block receives data via the AR adapter socket and makes the contained REAL value available at the IN output as soon as an event arrives via the adapter (CNF).

REAL conversion

🎧 Podcast

- IEC 61499: Factory automation in transition - from vision to industrial reality?

- From “Get Away with Electronics” to Essential: How Agricultural Tech Earned Trust and Delivered Real Relief

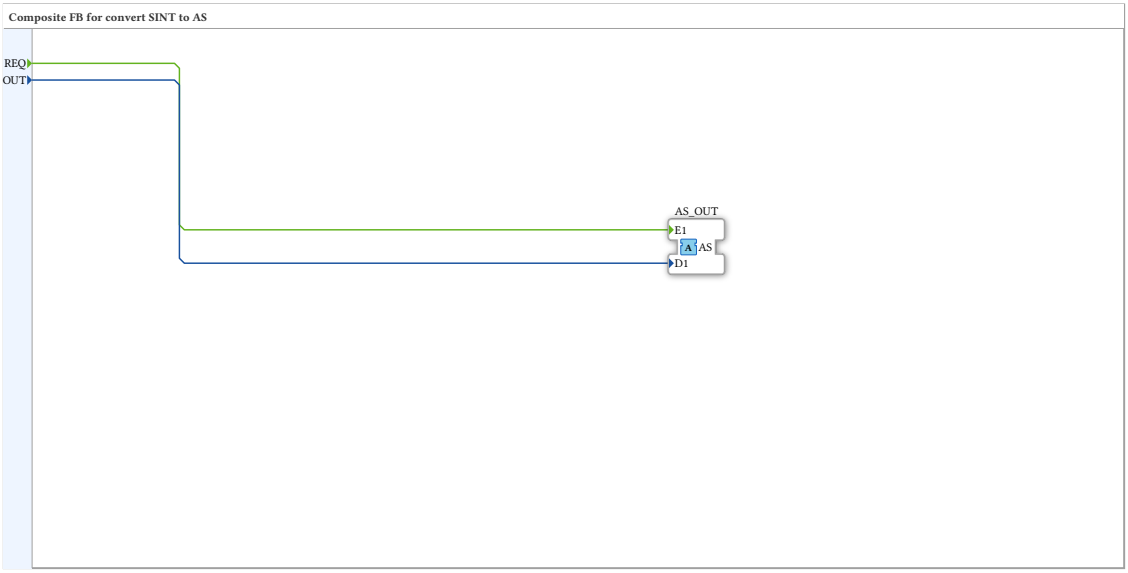
🛠️ Related exercises

- Uebung_072c
- Uebung_151_AX
- Uebung_152
- Uebung_153

Numerical Index (unidirectional)

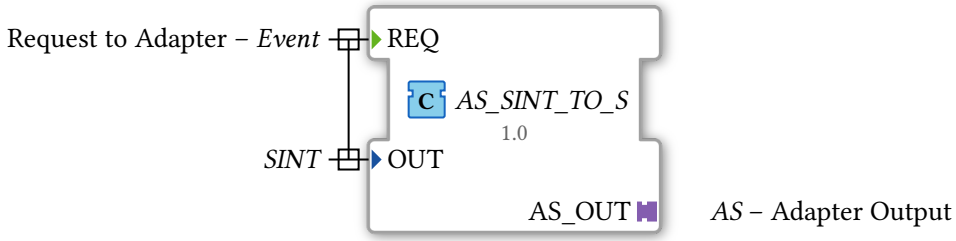
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AS_SINT_TO_S



Introduction

The AS_SINT_TO_S function block is a composite function block used to convert a SINT value into the AS adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: SINT input value to be converted

Data Outputs

- No direct data outputs available

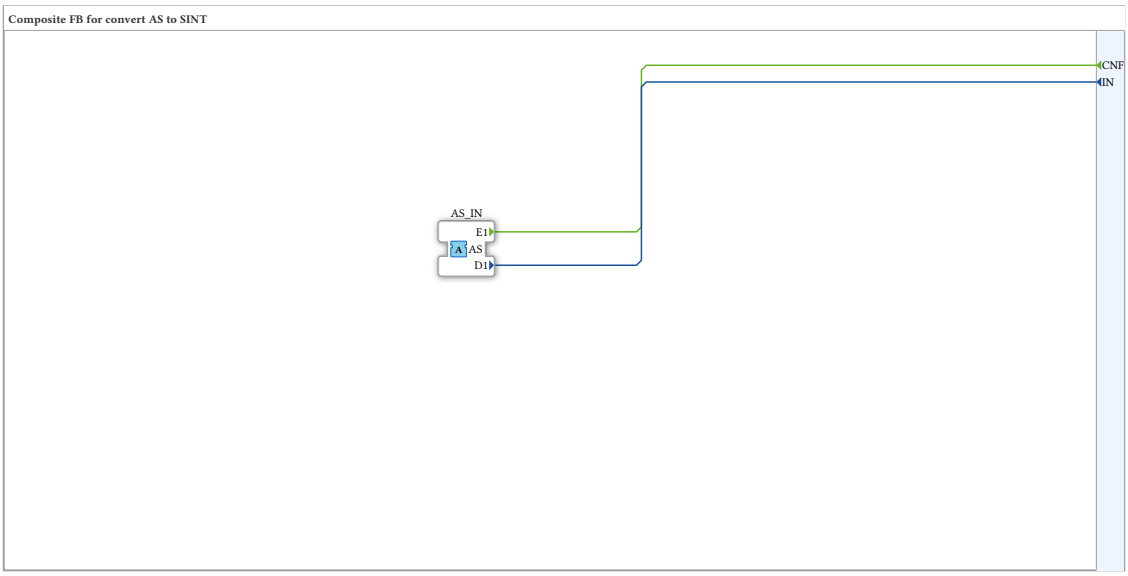
Adapter

- **AS_OUT**: Plug of type `AS`

Functionality

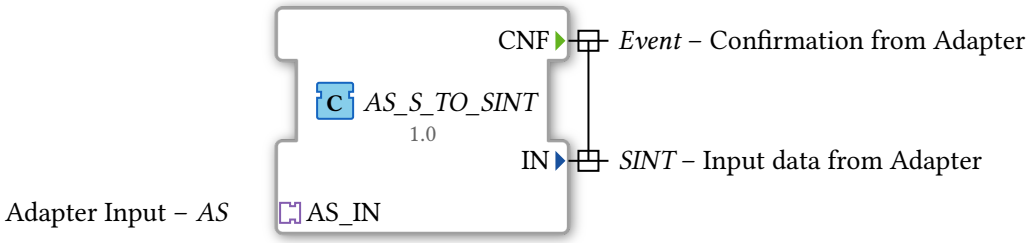
The function block receives a SINT value at the OUT data input and converts it to the AS adapter format upon arrival of the REQ event.

AS_S_TO_SINT



Introduction

The AS_S_TO_SINT function block is a composite function block used to convert the AS adapter format to a SINT value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: SINT output value of the converted data

Adapter

- **AS_IN**: Socket of type `AS`

Functionality

The function block receives data via the AS adapter socket and makes the contained SINT value available at the IN output as soon as an event arrives via the adapter (CNF).

SINT Conversion

🎧 Podcast

- SINT, INT, DINT: Why the choice of data type determines efficiency and errors

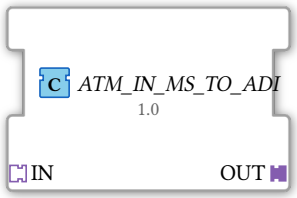
🛠️ Related Exercises

- Exercise_039_sub_NumbDisplay

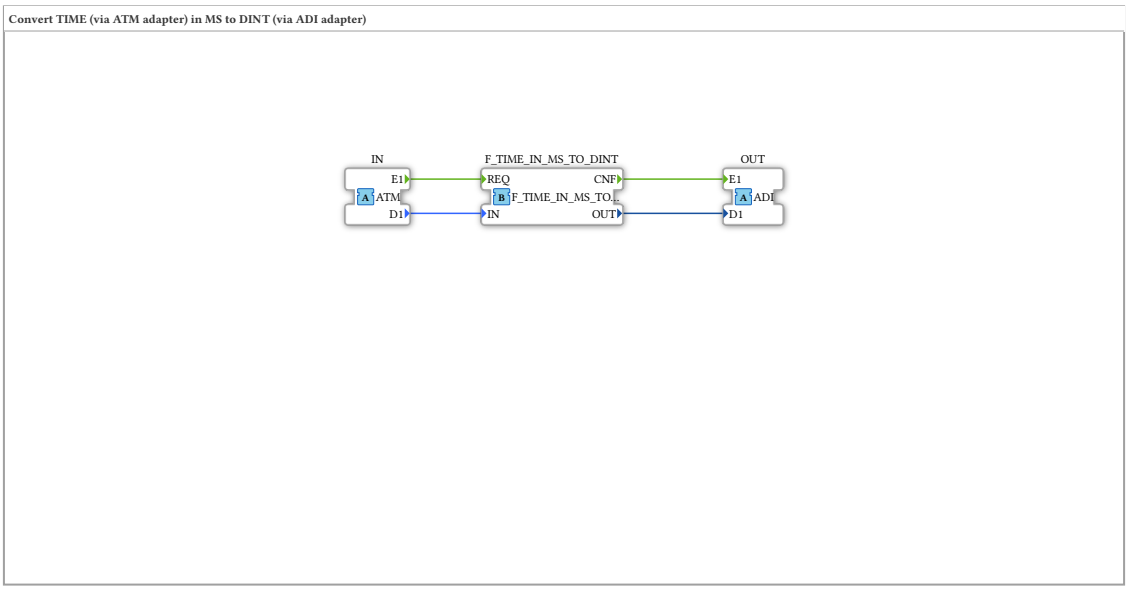
TIME

ATM_IN_MS_TO_ADI

Time value to convert – *ATM*



ADI – TIME value of IN expressed in MS as DINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_MS_TO_ADI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in milliseconds into a `DINT` value (32-bit signed integer), output via the ADI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `DINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ADI	Unidirectional `DINT` adapter as output interface, value of `IN` expressed in milliseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_MS_TO_DINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_MS_TO_DINT`.
3. `F_TIME_IN_MS_TO_DINT` converts the time value into a milliseconds figure of type `DINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_MS_TO_DINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_MS_TO_DINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in milliseconds and returns the result as `DINT` (32-bit signed integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (milliseconds) to `DINT` via `F_TIME_IN_MS_TO_DINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in milliseconds to an adapter socket of type `ADI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

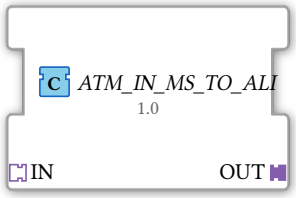
- **Same unit, other target types**: ATM_IN_MS_TO_ALI, ATM_IN_MS_TO_ALR, ATM_IN_MS_TO_AUDI, ATM_IN_MS_TO_AULI
- **Same target type, other units**: ATM_IN_NS_TO_ADI, ATM_IN_S_TO_ADI, ATM_IN_US_TO_ADI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_MS_TO_ADI`.

Conclusion

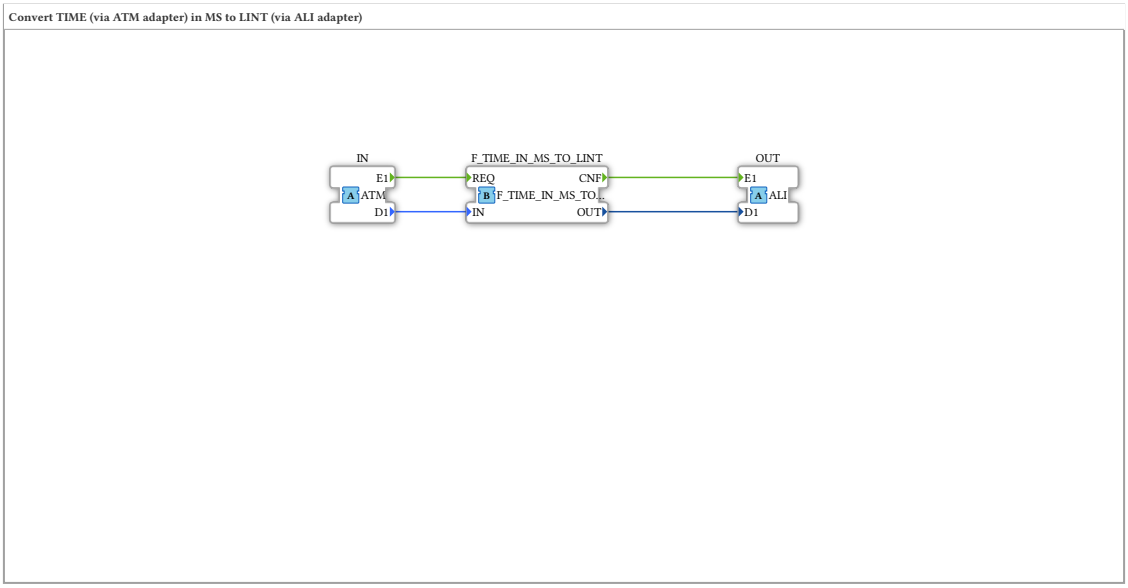
`ATM_IN_MS_TO_ADI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in milliseconds into a `DINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_MS_TO_ALI

Time value to convert – *ATM*



ALI – TIME value of IN expressed in MS as LINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_MS_TO_ALI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in milliseconds into a `LINT` value (64-bit signed integer), output via the ALI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `LINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ALI	Unidirectional `LINT` adapter as output interface, value of `IN` expressed in milliseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_MS_TO_LINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_MS_TO_LINT`.
3. `F_TIME_IN_MS_TO_LINT` converts the time value into a milliseconds figure of type `LINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_MS_TO_LINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_MS_TO_LINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in milliseconds and returns the result as `LINT` (64-bit signed integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (milliseconds) to `LINT` via `F_TIME_IN_MS_TO_LINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in milliseconds to an adapter socket of type `ALI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

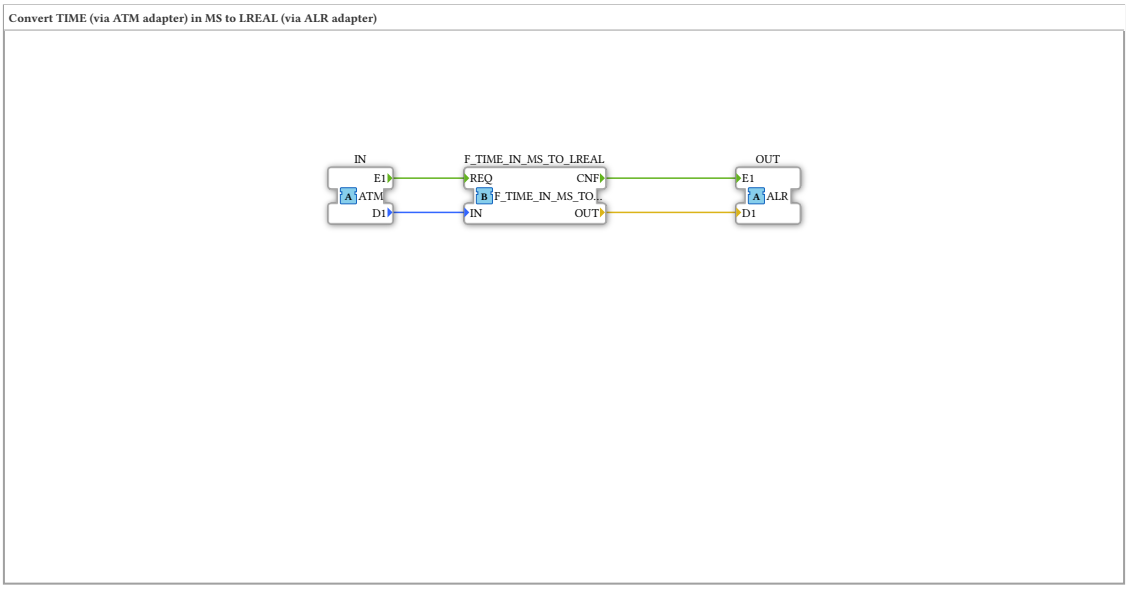
- **Same unit, other target types**: ATM_IN_MS_TO_ADI, ATM_IN_MS_TO_ALR, ATM_IN_MS_TO_AUDI, ATM_IN_MS_TO_AULI
- **Same target type, other units**: ATM_IN_NS_TO_ALI, ATM_IN_S_TO_ALI, ATM_IN_US_TO_ALI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_MS_TO_ALI`.

Conclusion

`ATM_IN_MS_TO_ALI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in milliseconds into a `LINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_MS_TO_ALR





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_MS_TO_ALR** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in milliseconds into a `LREAL` value (64-bit floating-point number), output via the ALR adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `LREAL`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ALR	Unidirectional `LREAL` adapter as output interface, value of `IN` expressed in milliseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_MS_TO_LREAL`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_MS_TO_LREAL`.
3. `F_TIME_IN_MS_TO_LREAL` converts the time value into a milliseconds figure of type `LREAL` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_MS_TO_LREAL`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_MS_TO_LREAL` interprets the `TIME` value (internally held in nanoseconds) as a figure in milliseconds and returns the result as `LREAL` (64-bit floating-point number).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (milliseconds) to `LREAL` via `F_TIME_IN_MS_TO_LREAL`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in milliseconds to an adapter socket of type `ALR`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

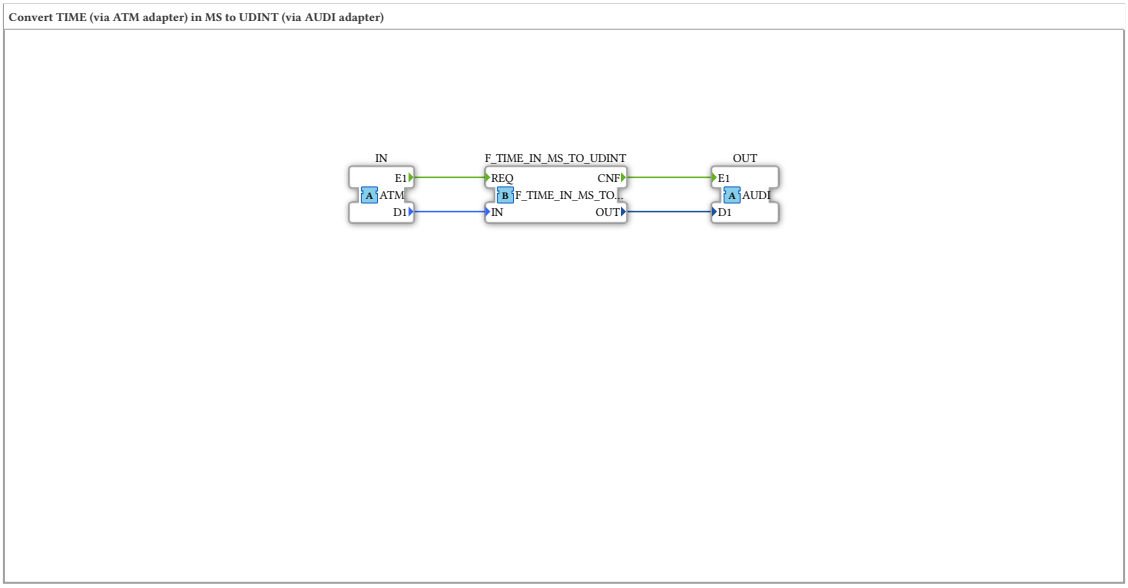
- **Same unit, other target types**: ATM_IN_MS_TO_ADI, ATM_IN_MS_TO_ALI, ATM_IN_MS_TO_AUDI, ATM_IN_MS_TO_AULI
- **Same target type, other units**: ATM_IN_NS_TO_ALR, ATM_IN_S_TO_ALR, ATM_IN_US_TO_ALR
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_MS_TO_ALR`.

Conclusion

`ATM_IN_MS_TO_ALR` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in milliseconds into a `LREAL` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_MS_TO_AUDI





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_MS_TO_AUDI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in milliseconds into a `UDINT` value (32-bit unsigned integer), output via the AUDI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `UDINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::AUDI	Unidirectional `UDINT` adapter as output interface, value of `IN` expressed in milliseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_MS_TO_UDINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_MS_TO_UDINT`.
3. `F_TIME_IN_MS_TO_UDINT` converts the time value into a milliseconds figure of type `UDINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_MS_TO_UDINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_MS_TO_UDINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in milliseconds and returns the result as `UDINT` (32-bit unsigned integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (milliseconds) to `UDINT` via `F_TIME_IN_MS_TO_UDINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in milliseconds to an adapter socket of type `AUDI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

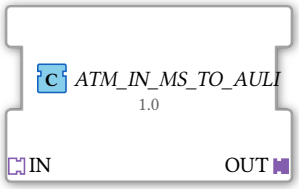
- **Same unit, other target types**: ATM_IN_MS_TO_ADI, ATM_IN_MS_TO_ALI, ATM_IN_MS_TO_ALR, ATM_IN_MS_TO_AULI
- **Same target type, other units**: ATM_IN_NS_TO_AUDI, ATM_IN_S_TO_AUDI, ATM_IN_US_TO_AUDI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_MS_TO_AUDI`.

Conclusion

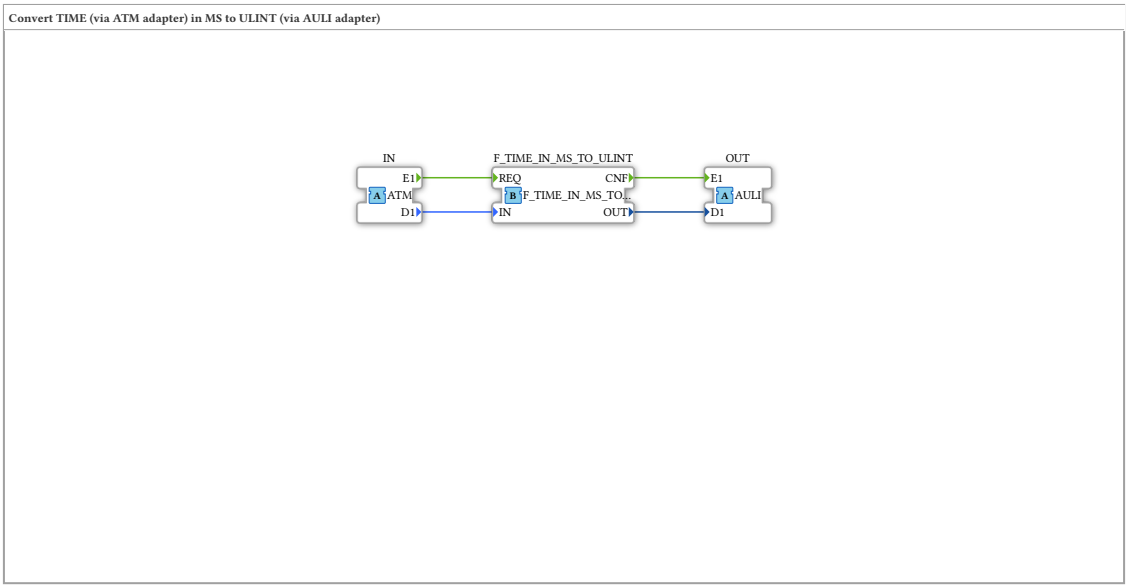
`ATM_IN_MS_TO_AUDI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in milliseconds into a `UDINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_MS_TO_AULI

Time value to convert – *ATM*



AULI – TIME value of IN expressed in MS as ULINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_IN_MS_TO_AULI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in milliseconds into a `ULINT` value (64-bit unsigned integer), output via the AULI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `ULINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::AULI	Unidirectional `ULINT` adapter as output interface, value of `IN` expressed in milliseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_MS_TO_ULINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_MS_TO_ULINT`.
3. `F_TIME_IN_MS_TO_ULINT` converts the time value into a milliseconds figure of type `ULINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_MS_TO_ULINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_MS_TO_ULINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in milliseconds and returns the result as `ULINT` (64-bit unsigned integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (milliseconds) to `ULINT` via `F_TIME_IN_MS_TO_ULINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in milliseconds to an adapter socket of type `AULI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

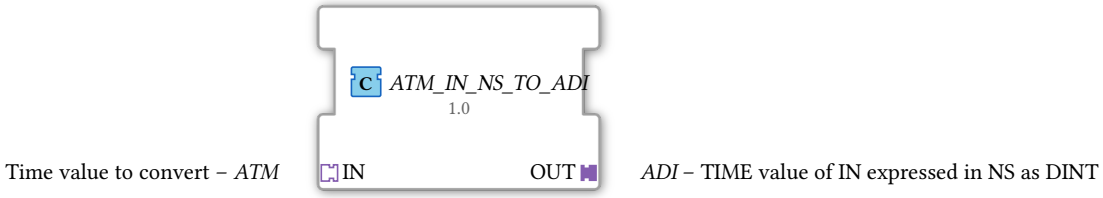
Comparison with similar function blocks

- **Same unit, other target types**: ATM_IN_MS_TO_ADI, ATM_IN_MS_TO_ALI, ATM_IN_MS_TO_ALR, ATM_IN_MS_TO_AUDI
- **Same target type, other units**: ATM_IN_NS_TO_AULI, ATM_IN_S_TO_AULI, ATM_IN_US_TO_AULI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_MS_TO_AULI`.

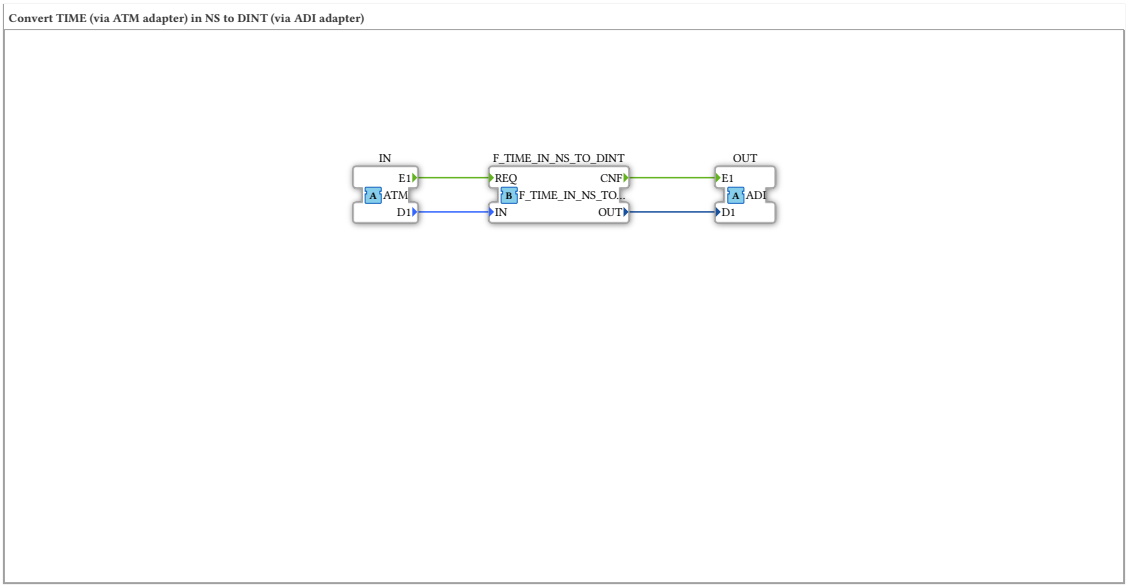
Conclusion

`ATM_IN_MS_TO_AULI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in milliseconds into a `ULINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_NS_TO_ADI



Time value to convert – *ATM* — *ADI* – TIME value of IN expressed in NS as DINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_NS_TO_ADI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in nanoseconds into a `DINT` value (32-bit signed integer), output via the ADI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `DINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ADI	Unidirectional `DINT` adapter as output interface, value of `IN` expressed in nanoseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_NS_TO_DINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_NS_TO_DINT`.
3. `F_TIME_IN_NS_TO_DINT` converts the time value into a nanoseconds figure of type `DINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_NS_TO_DINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_NS_TO_DINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in nanoseconds and returns the result as `DINT` (32-bit signed integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (nanoseconds) to `DINT` via `F_TIME_IN_NS_TO_DINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in nanoseconds to an adapter socket of type `ADI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

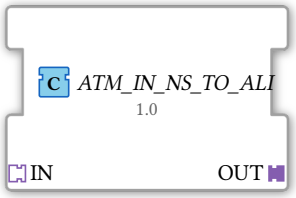
- **Same unit, other target types**: ATM_IN_NS_TO_ALI, ATM_IN_NS_TO_ALR, ATM_IN_NS_TO_AUDI, ATM_IN_NS_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_ADI, ATM_IN_S_TO_ADI, ATM_IN_US_TO_ADI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_NS_TO_ADI`.

Conclusion

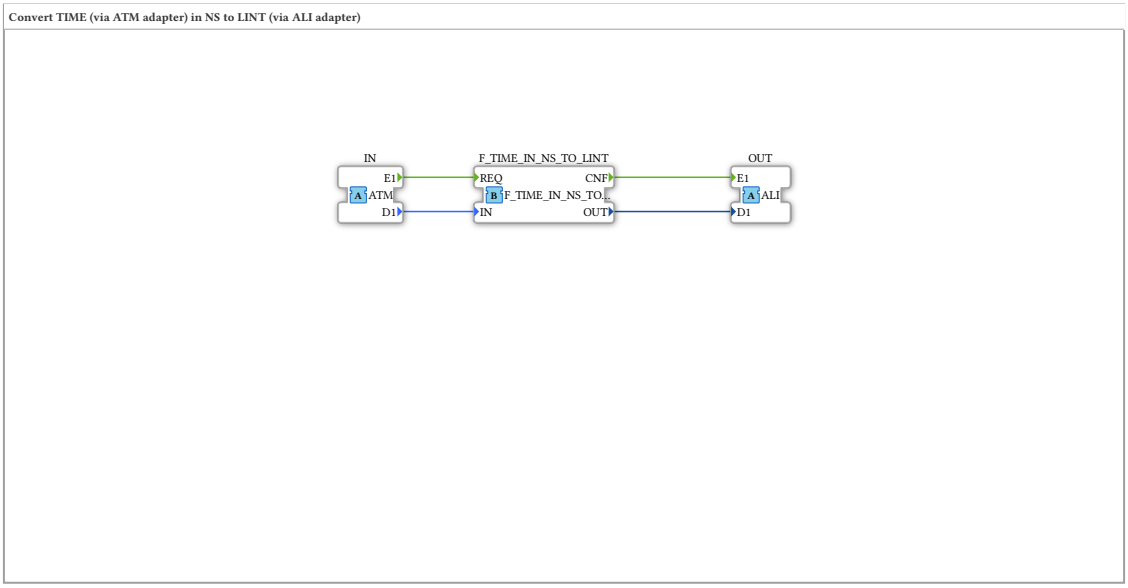
`ATM_IN_NS_TO_ADI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in nanoseconds into a `DINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_NS_TO_ALI

Time value to convert – *ATM*



ALI – TIME value of IN expressed in NS as LINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_NS_TO_ALI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in nanoseconds into a `LINT` value (64-bit signed integer), output via the ALI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `LINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ALI	Unidirectional `LINT` adapter as output interface, value of `IN` expressed in nanoseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_NS_TO_LINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_NS_TO_LINT`.
3. `F_TIME_IN_NS_TO_LINT` converts the time value into a nanoseconds figure of type `LINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_NS_TO_LINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_NS_TO_LINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in nanoseconds and returns the result as `LINT` (64-bit signed integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (nanoseconds) to `LINT` via `F_TIME_IN_NS_TO_LINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in nanoseconds to an adapter socket of type `ALI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

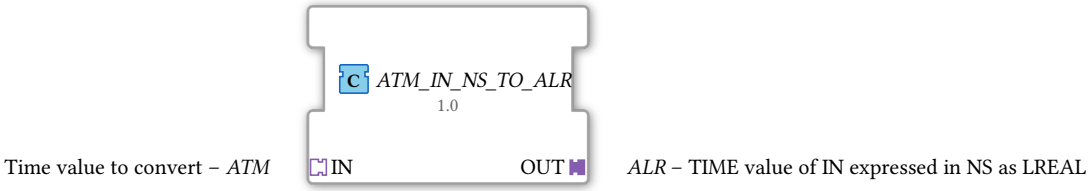
Comparison with similar function blocks

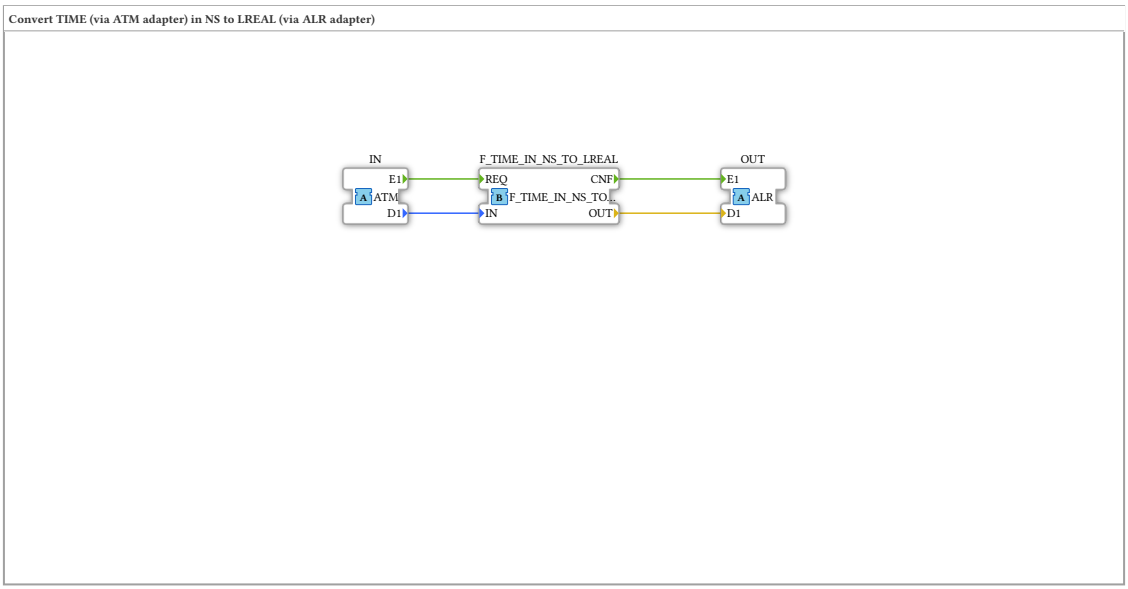
- **Same unit, other target types**: ATM_IN_NS_TO_ADI, ATM_IN_NS_TO_ALR, ATM_IN_NS_TO_AUDI, ATM_IN_NS_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_ALI, ATM_IN_S_TO_ALI, ATM_IN_US_TO_ALI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_NS_TO_ALI`.

Conclusion

`ATM_IN_NS_TO_ALI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in nanoseconds into a `LINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_NS_TO_ALR





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_IN_NS_TO_ALR** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in nanoseconds into a `LREAL` value (64-bit floating-point number), output via the ALR adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `LREAL`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ALR	Unidirectional `LREAL` adapter as output interface, value of `IN` expressed in nanoseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_NS_TO_LREAL`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_NS_TO_LREAL`.
3. `F_TIME_IN_NS_TO_LREAL` converts the time value into a nanoseconds figure of type `LREAL` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_NS_TO_LREAL`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_NS_TO_LREAL` interprets the `TIME` value (internally held in nanoseconds) as a figure in nanoseconds and returns the result as `LREAL` (64-bit floating-point number).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (nanoseconds) to `LREAL` via `F_TIME_IN_NS_TO_LREAL`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in nanoseconds to an adapter socket of type `ALR`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

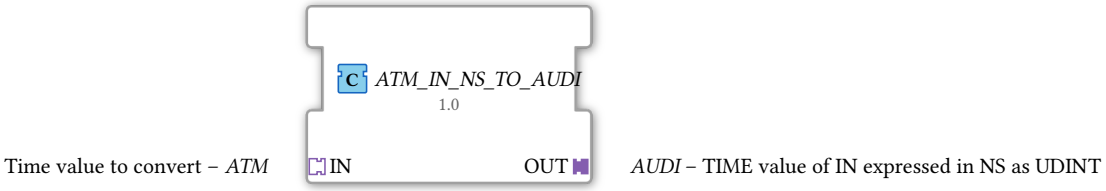
Comparison with similar function blocks

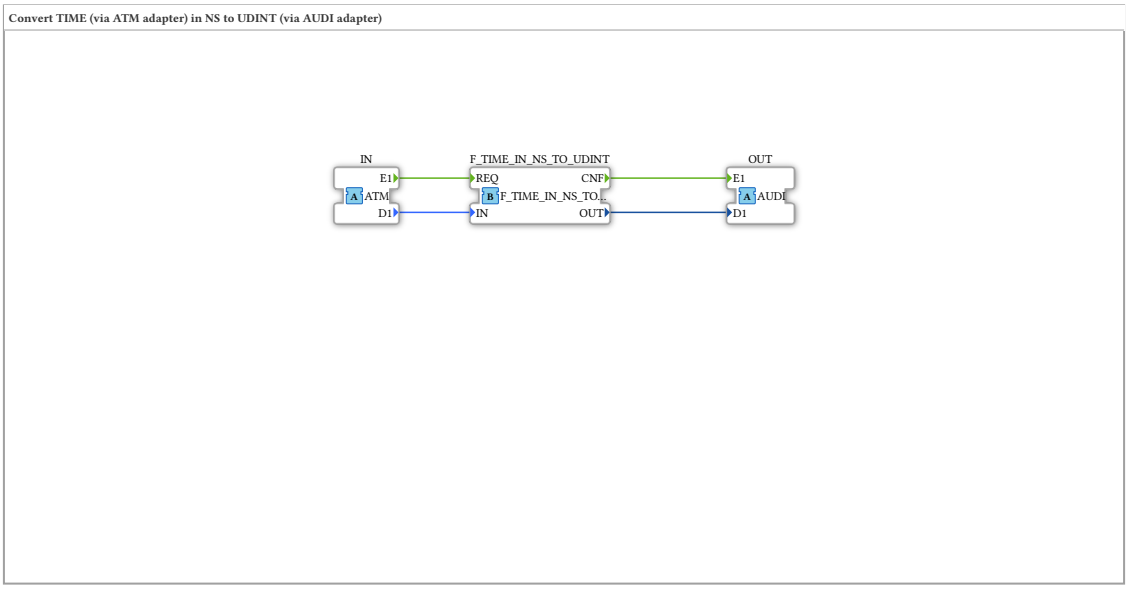
- **Same unit, other target types**: ATM_IN_NS_TO_ADI, ATM_IN_NS_TO_ALI, ATM_IN_NS_TO_AUDI, ATM_IN_NS_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_ALR, ATM_IN_S_TO_ALR, ATM_IN_US_TO_ALR
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_NS_TO_ALR`.

Conclusion

`ATM_IN_NS_TO_ALR` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in nanoseconds into a `LREAL` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_NS_TO_AUDI





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_NS_TO_AUDI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in nanoseconds into a `UDINT` value (32-bit unsigned integer), output via the AUDI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `UDINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::AUDI	Unidirectional `UDINT` adapter as output interface, value of `IN` expressed in nanoseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_NS_TO_UDINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_NS_TO_UDINT`.
3. `F_TIME_IN_NS_TO_UDINT` converts the time value into a nanoseconds figure of type `UDINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_NS_TO_UDINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_NS_TO_UDINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in nanoseconds and returns the result as `UDINT` (32-bit unsigned integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (nanoseconds) to `UDINT` via `F_TIME_IN_NS_TO_UDINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in nanoseconds to an adapter socket of type `AUDI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

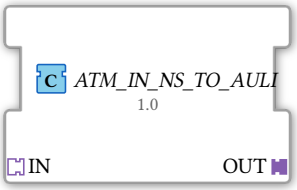
- **Same unit, other target types**: ATM_IN_NS_TO_ADI, ATM_IN_NS_TO_ALI, ATM_IN_NS_TO_ALR, ATM_IN_NS_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_AUDI, ATM_IN_S_TO_AUDI, ATM_IN_US_TO_AUDI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_NS_TO_AUDI`.

Conclusion

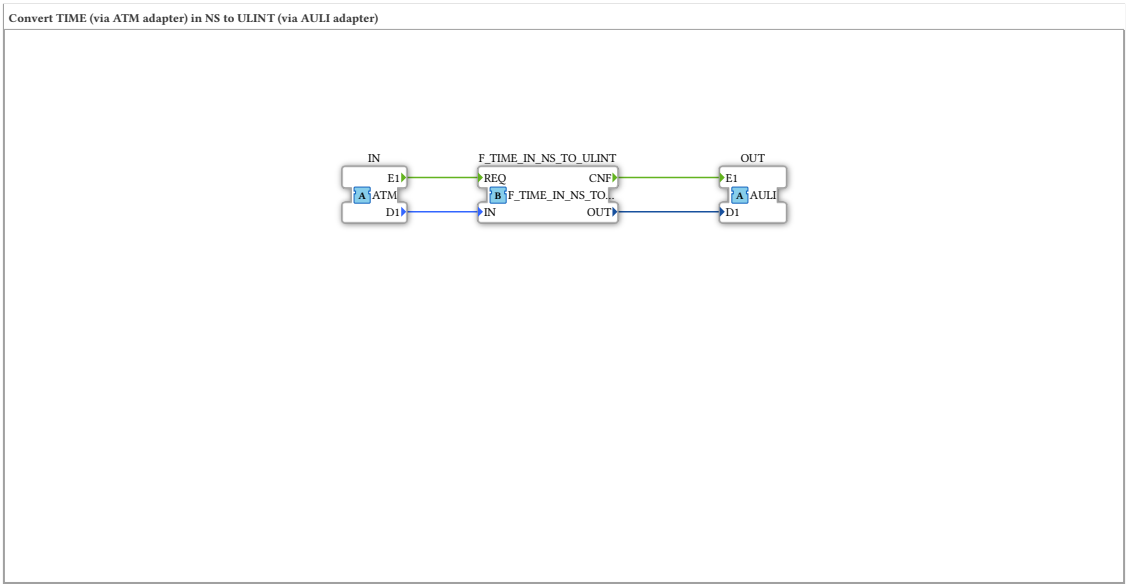
`ATM_IN_NS_TO_AUDI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in nanoseconds into a `UDINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_NS_TO_AULI

Time value to convert – *ATM*



AULI – TIME value of IN expressed in NS as ULINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_NS_TO_AULI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in nanoseconds into a `ULINT` value (64-bit unsigned integer), output via the AULI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `ULINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::AULI	Unidirectional `ULINT` adapter as output interface, value of `IN` expressed in nanoseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_NS_TO_ULINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_NS_TO_ULINT`.
3. `F_TIME_IN_NS_TO_ULINT` converts the time value into a nanoseconds figure of type `ULINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_NS_TO_ULINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_NS_TO_ULINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in nanoseconds and returns the result as `ULINT` (64-bit unsigned integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (nanoseconds) to `ULINT` via `F_TIME_IN_NS_TO_ULINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in nanoseconds to an adapter socket of type `AULI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

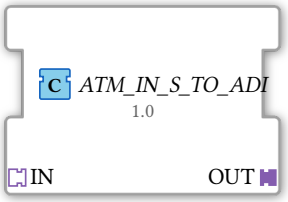
- **Same unit, other target types**: ATM_IN_NS_TO_ADI, ATM_IN_NS_TO_ALI, ATM_IN_NS_TO_ALR, ATM_IN_NS_TO_AUDI
- **Same target type, other units**: ATM_IN_MS_TO_AULI, ATM_IN_S_TO_AULI, ATM_IN_US_TO_AULI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_NS_TO_AULI`.

Conclusion

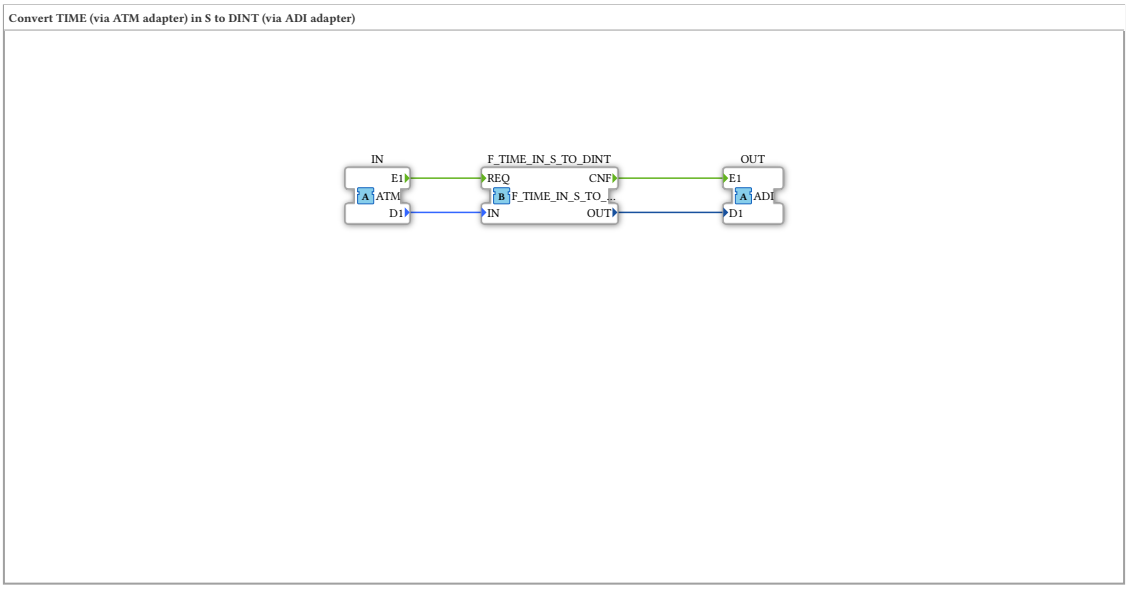
`ATM_IN_NS_TO_AULI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in nanoseconds into a `ULINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_S_TO_ADI

Time value to convert – *ATM*



ADI – TIME value of IN expressed in S as DINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_S_TO_ADI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in seconds into a `DINT` value (32-bit signed integer), output via the ADI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `DINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ADI	Unidirectional `DINT` adapter as output interface, value of `IN` expressed in seconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_S_TO_DINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_S_TO_DINT`.
3. `F_TIME_IN_S_TO_DINT` converts the time value into a seconds figure of type `DINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_S_TO_DINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_S_TO_DINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in seconds and returns the result as `DINT` (32-bit signed integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (seconds) to `DINT` via `F_TIME_IN_S_TO_DINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in seconds to an adapter socket of type `ADI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

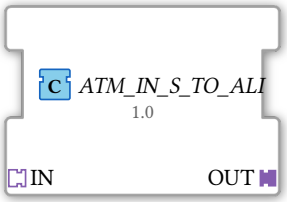
- **Same unit, other target types**: ATM_IN_S_TO_ALI, ATM_IN_S_TO_ALR, ATM_IN_S_TO_AUDI, ATM_IN_S_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_ADI, ATM_IN_NS_TO_ADI, ATM_IN_US_TO_ADI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_S_TO_ADI`.

Conclusion

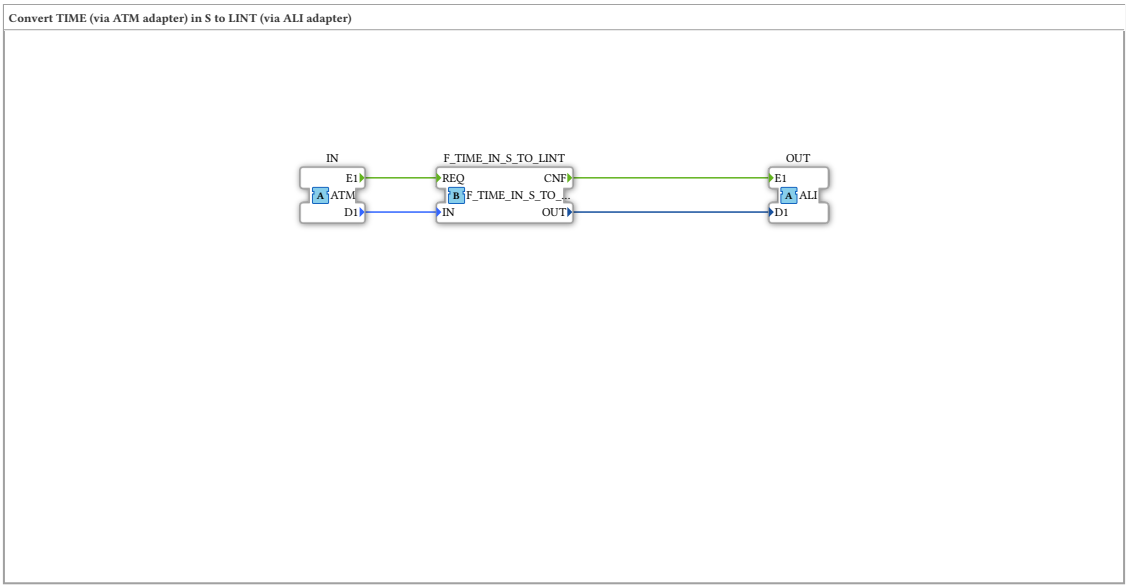
`ATM_IN_S_TO_ADI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in seconds into a `DINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_S_TO_ALI

Time value to convert – *ATM*



ALI – TIME value of IN expressed in S as LINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_S_TO_ALI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in seconds into a `LINT` value (64-bit signed integer), output via the ALI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `LINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ALI	Unidirectional `LINT` adapter as output interface, value of `IN` expressed in seconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_S_TO_LINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_S_TO_LINT`.
3. `F_TIME_IN_S_TO_LINT` converts the time value into a seconds figure of type `LINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_S_TO_LINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_S_TO_LINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in seconds and returns the result as `LINT` (64-bit signed integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (seconds) to `LINT` via `F_TIME_IN_S_TO_LINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in seconds to an adapter socket of type `ALI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

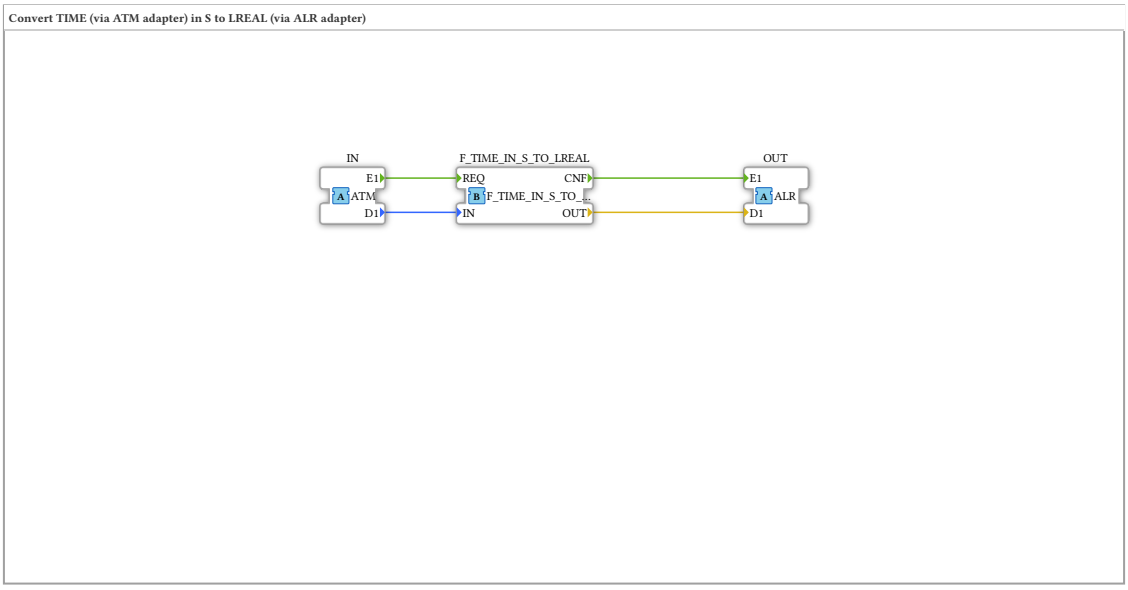
- **Same unit, other target types**: ATM_IN_S_TO_ADI, ATM_IN_S_TO_ALR, ATM_IN_S_TO_AUDI, ATM_IN_S_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_ALI, ATM_IN_NS_TO_ALI, ATM_IN_US_TO_ALI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_S_TO_ALI`.

Conclusion

`ATM_IN_S_TO_ALI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in seconds into a `LINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_S_TO_ALR





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_IN_S_TO_ALR** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in seconds into a `LREAL` value (64-bit floating-point number), output via the ALR adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `LREAL`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ALR	Unidirectional `LREAL` adapter as output interface, value of `IN` expressed in seconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_S_TO_LREAL`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_S_TO_LREAL`.
3. `F_TIME_IN_S_TO_LREAL` converts the time value into a seconds figure of type `LREAL` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_S_TO_LREAL`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_S_TO_LREAL` interprets the `TIME` value (internally held in nanoseconds) as a figure in seconds and returns the result as `LREAL` (64-bit floating-point number).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (seconds) to `LREAL` via `F_TIME_IN_S_TO_LREAL`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in seconds to an adapter socket of type `ALR`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

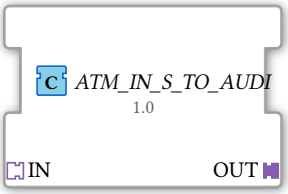
- **Same unit, other target types**: ATM_IN_S_TO_ADI, ATM_IN_S_TO_ALI, ATM_IN_S_TO_AUDI, ATM_IN_S_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_ALR, ATM_IN_NS_TO_ALR, ATM_IN_US_TO_ALR
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_S_TO_ALR`.

Conclusion

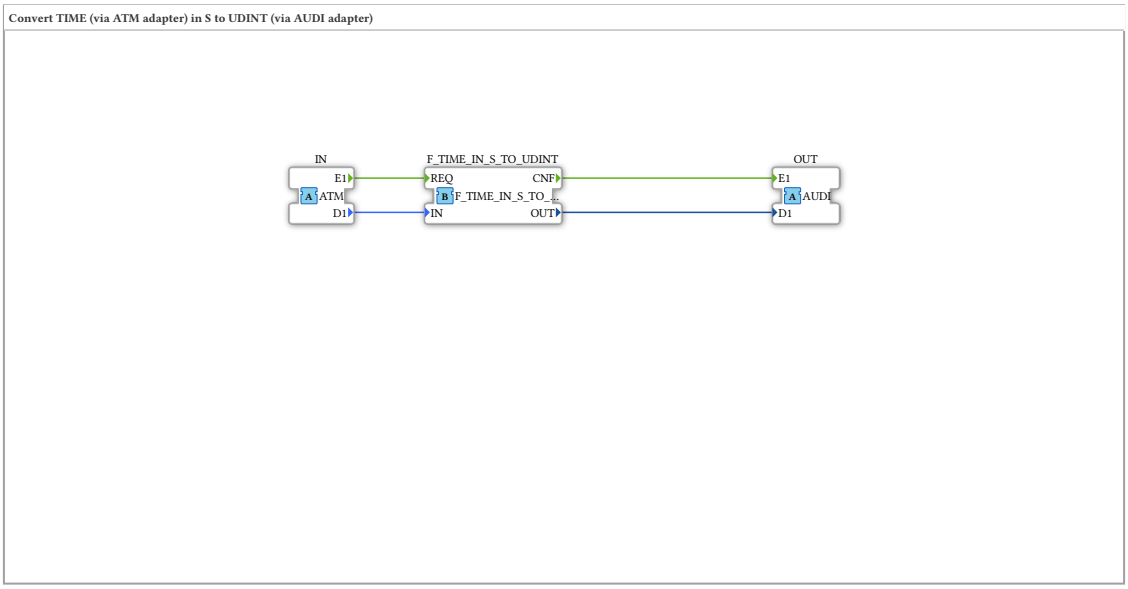
`ATM_IN_S_TO_ALR` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in seconds into a `LREAL` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_S_TO_AUDI

Time value to convert – *ATM*



AUDI – TIME value of IN expressed in S as UDINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_S_TO_AUDI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in seconds into a `UDINT` value (32-bit unsigned integer), output via the AUDI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `UDINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::AUDI	Unidirectional `UDINT` adapter as output interface, value of `IN` expressed in seconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_S_TO_UDINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_S_TO_UDINT`.
3. `F_TIME_IN_S_TO_UDINT` converts the time value into a seconds figure of type `UDINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_S_TO_UDINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_S_TO_UDINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in seconds and returns the result as `UDINT` (32-bit unsigned integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (seconds) to `UDINT` via `F_TIME_IN_S_TO_UDINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in seconds to an adapter socket of type `AUDI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

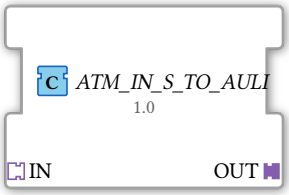
- **Same unit, other target types**: ATM_IN_S_TO_ADI, ATM_IN_S_TO_ALI, ATM_IN_S_TO_ALR, ATM_IN_S_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_AUDI, ATM_IN_NS_TO_AUDI, ATM_IN_US_TO_AUDI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_S_TO_AUDI`.

Conclusion

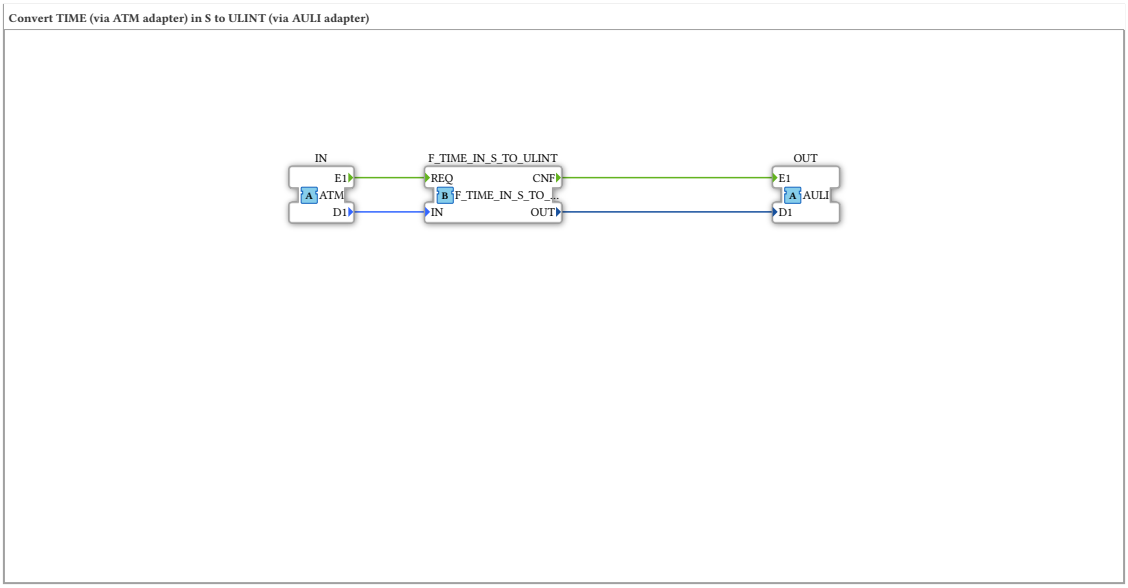
`ATM_IN_S_TO_AUDI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in seconds into a `UDINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_S_TO_AULI

Time value to convert – *ATM*



AULI – TIME value of IN expressed in S as ULINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_IN_S_TO_AULI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in seconds into a `ULINT` value (64-bit unsigned integer), output via the AULI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `ULINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::AULI	Unidirectional `ULINT` adapter as output interface, value of `IN` expressed in seconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_S_TO_ULINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_S_TO_ULINT`.
3. `F_TIME_IN_S_TO_ULINT` converts the time value into a seconds figure of type `ULINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_S_TO_ULINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_S_TO_ULINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in seconds and returns the result as `ULINT` (64-bit unsigned integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (seconds) to `ULINT` via `F_TIME_IN_S_TO_ULINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in seconds to an adapter socket of type `AULI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

- **Same unit, other target types**: ATM_IN_S_TO_ADI, ATM_IN_S_TO_ALI, ATM_IN_S_TO_ALR, ATM_IN_S_TO_AUDI
- **Same target type, other units**: ATM_IN_MS_TO_AULI, ATM_IN_NS_TO_AULI, ATM_IN_US_TO_AULI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_S_TO_AULI`.

Conclusion

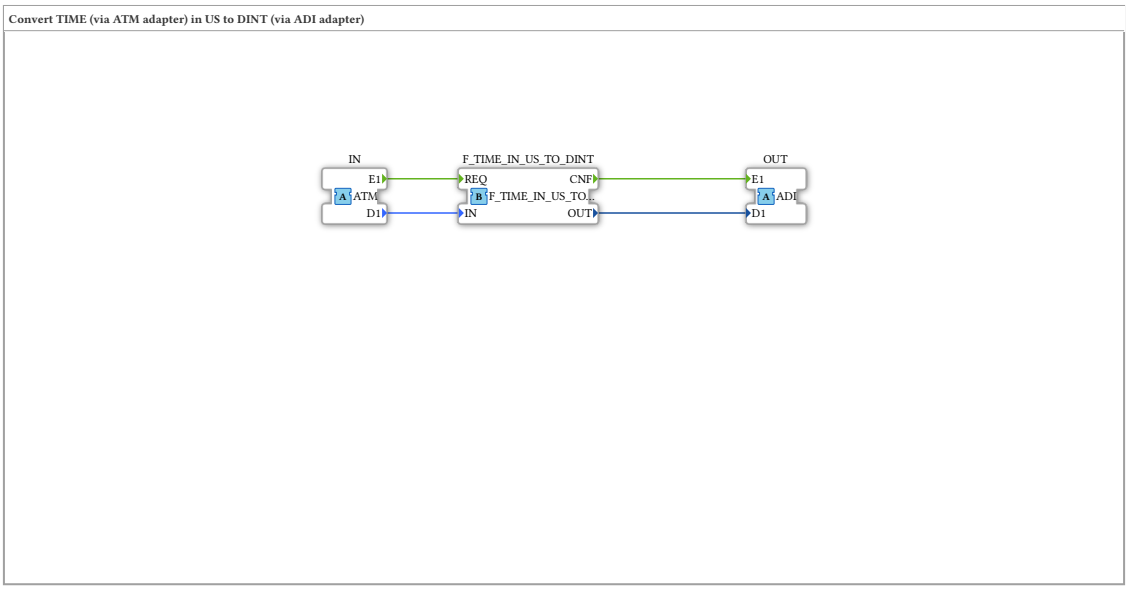
`ATM_IN_S_TO_AULI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in seconds into a `ULINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_US_TO_ADI



Time value to convert – *ATM*

ADI – TIME value of IN expressed in US as DINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_US_TO_ADI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in microseconds into a `DINT` value (32-bit signed integer), output via the ADI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `DINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ADI	Unidirectional `DINT` adapter as output interface, value of `IN` expressed in microseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_US_TO_DINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_US_TO_DINT`.
3. `F_TIME_IN_US_TO_DINT` converts the time value into a microseconds figure of type `DINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_US_TO_DINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_US_TO_DINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in microseconds and returns the result as `DINT` (32-bit signed integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (microseconds) to `DINT` via `F_TIME_IN_US_TO_DINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in microseconds to an adapter socket of type `ADI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

- **Same unit, other target types**: ATM_IN_US_TO_ALI, ATM_IN_US_TO_ALR, ATM_IN_US_TO_AUDI, ATM_IN_US_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_ADI, ATM_IN_NS_TO_ADI, ATM_IN_S_TO_ADI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_US_TO_ADI`.

Conclusion

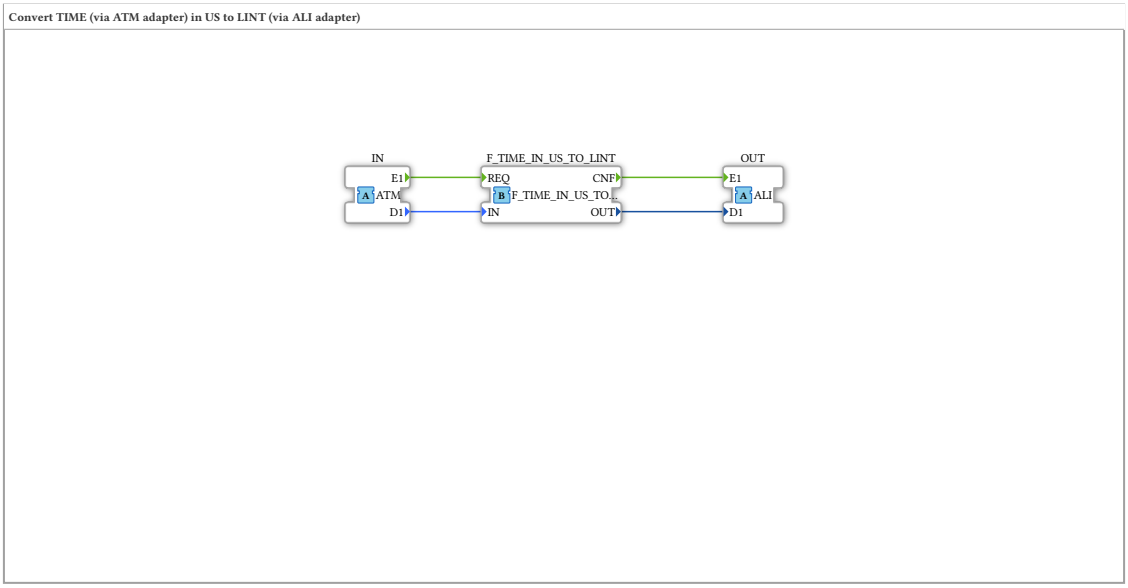
`ATM_IN_US_TO_ADI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in microseconds into a `DINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_US_TO_ALI



Time value to convert – *ATM*

ALI – TIME value of IN expressed in US as LINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_US_TO_ALI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in microseconds into a `LINT` value (64-bit signed integer), output via the ALI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `LINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ALI	Unidirectional `LINT` adapter as output interface, value of `IN` expressed in microseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_US_TO_LINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_US_TO_LINT`.
3. `F_TIME_IN_US_TO_LINT` converts the time value into a microseconds figure of type `LINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_US_TO_LINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_US_TO_LINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in microseconds and returns the result as `LINT` (64-bit signed integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (microseconds) to `LINT` via `F_TIME_IN_US_TO_LINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in microseconds to an adapter socket of type `ALI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

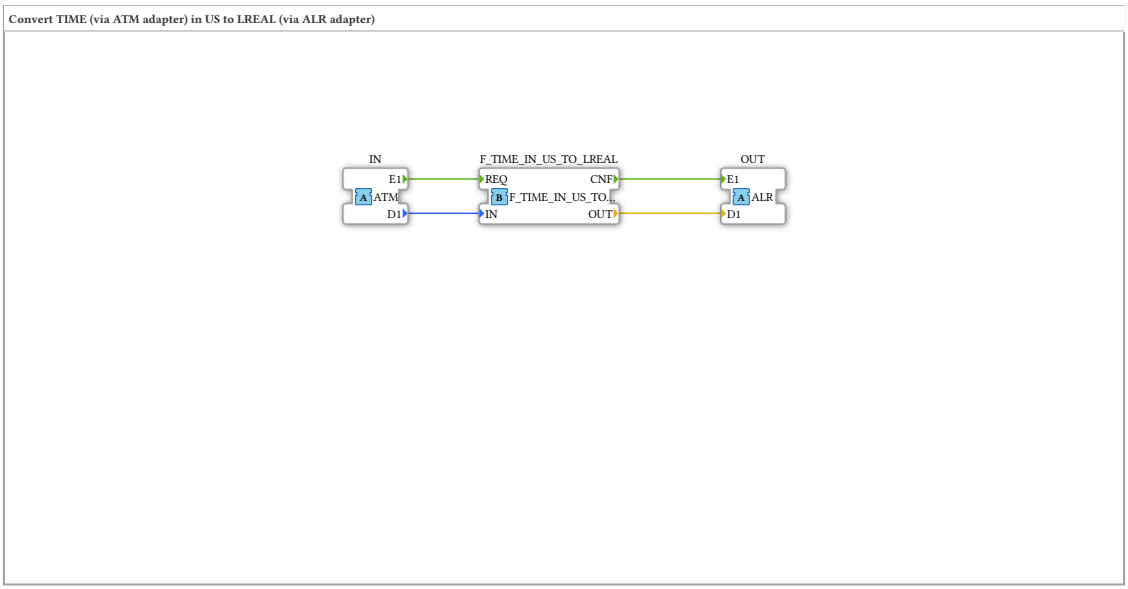
- **Same unit, other target types**: ATM_IN_US_TO_ADI, ATM_IN_US_TO_ALR, ATM_IN_US_TO_AUDI, ATM_IN_US_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_ALI, ATM_IN_NS_TO_ALI, ATM_IN_S_TO_ALI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_US_TO_ALI`.

Conclusion

`ATM_IN_US_TO_ALI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in microseconds into a `LINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_US_TO_ALR





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_IN_US_TO_ALR** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in microseconds into a `LREAL` value (64-bit floating-point number), output via the ALR adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `LREAL`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::ALR	Unidirectional `LREAL` adapter as output interface, value of `IN` expressed in microseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_US_TO_LREAL`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_US_TO_LREAL`.
3. `F_TIME_IN_US_TO_LREAL` converts the time value into a microseconds figure of type `LREAL` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_US_TO_LREAL`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_US_TO_LREAL` interprets the `TIME` value (internally held in nanoseconds) as a figure in microseconds and returns the result as `LREAL` (64-bit floating-point number).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (microseconds) to `LREAL` via `F_TIME_IN_US_TO_LREAL`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in microseconds to an adapter socket of type `ALR`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

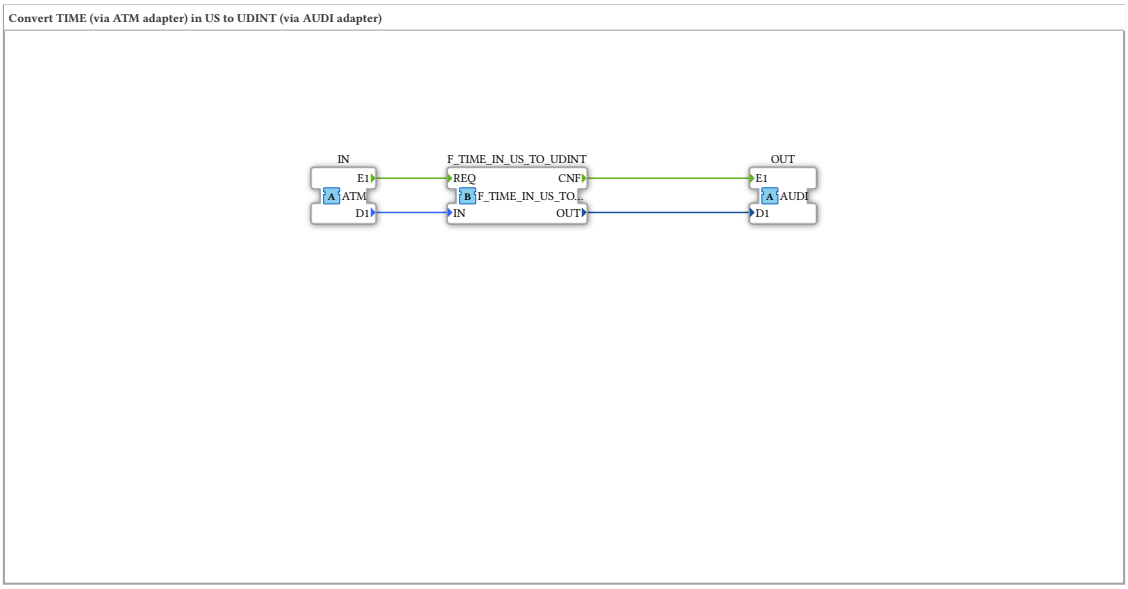
- **Same unit, other target types**: ATM_IN_US_TO_ADI, ATM_IN_US_TO_ALI, ATM_IN_US_TO_AUDI, ATM_IN_US_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_ALR, ATM_IN_NS_TO_ALR, ATM_IN_S_TO_ALR
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_US_TO_ALR`.

Conclusion

`ATM_IN_US_TO_ALR` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in microseconds into a `LREAL` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_US_TO_AUDI





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_US_TO_AUDI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in microseconds into a `UDINT` value (32-bit unsigned integer), output via the AUDI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `UDINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::AUDI	Unidirectional `UDINT` adapter as output interface, value of `IN` expressed in microseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_US_TO_UDINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_US_TO_UDINT`.
3. `F_TIME_IN_US_TO_UDINT` converts the time value into a microseconds figure of type `UDINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_US_TO_UDINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_US_TO_UDINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in microseconds and returns the result as `UDINT` (32-bit unsigned integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (microseconds) to `UDINT` via `F_TIME_IN_US_TO_UDINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in microseconds to an adapter socket of type `AUDI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

Comparison with similar function blocks

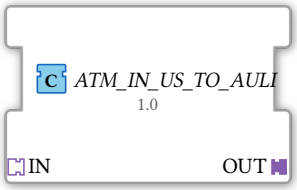
- **Same unit, other target types**: ATM_IN_US_TO_ADI, ATM_IN_US_TO_ALI, ATM_IN_US_TO_ALR, ATM_IN_US_TO_AULI
- **Same target type, other units**: ATM_IN_MS_TO_AUDI, ATM_IN_NS_TO_AUDI, ATM_IN_S_TO_AUDI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_US_TO_AUDI`.

Conclusion

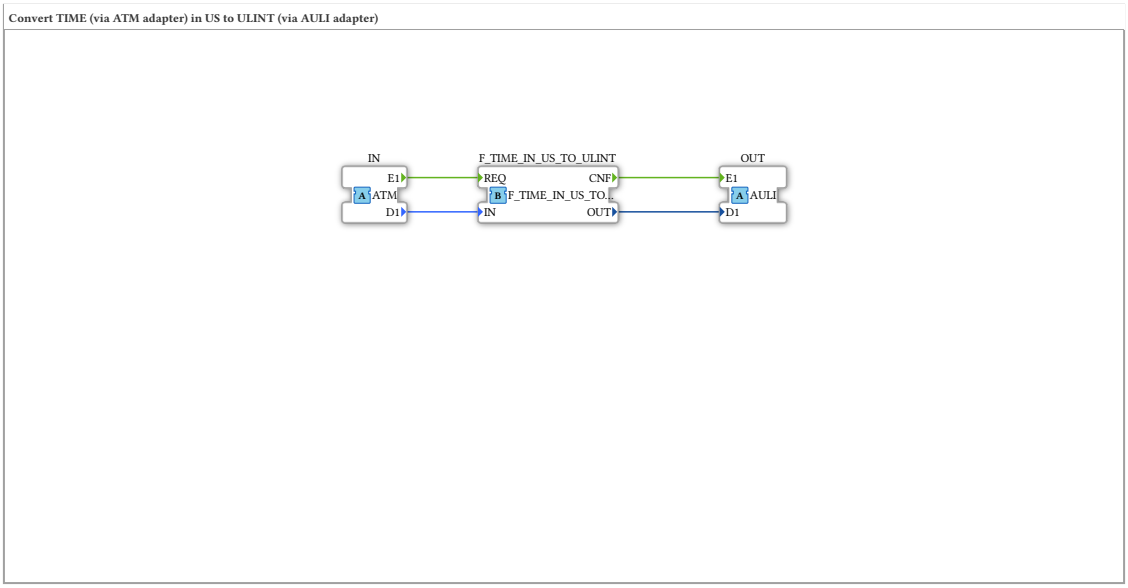
`ATM_IN_US_TO_AUDI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in microseconds into a `UDINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_IN_US_TO_AULI

Time value to convert – *ATM*



AULI – TIME value of IN expressed in US as ULINT



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_IN_US_TO_AULI** is a composite block that converts a `TIME` value (carried via the ATM adapter) expressed in microseconds into a `ULINT` value (64-bit unsigned integer), output via the AULI adapter. It belongs to the `ATM_IN_<UNIT>_TO_<TARGET>` block family, which provides the same conversion for all four time units (MS, NS, S, US) and all five target integer/floating-point types.

Interface Structure

Event Inputs

- No directly visible event inputs. Event control is handled indirectly via the **IN** adapter (event E1).

Event Outputs

- No directly visible event outputs. Event signaling is handled indirectly via the **OUT** adapter (event E1).

Data Inputs

- No directly visible data inputs. The input value is provided via the **IN** adapter (data point D1 of type `TIME`).

Data Outputs

- No directly visible data outputs. The output value is provided via the **OUT** adapter (data point D1 of type `ULINT`).

Adapters

Adapter	Role	Type	Description
IN	Socket	adapter::types::unidirectional::ATM	Unidirectional `TIME` adapter as input interface.
OUT	Plug	adapter::types::unidirectional::AULI	Unidirectional `ULINT` adapter as output interface, value of `IN` expressed in microseconds.

Functionality

The function block internally contains a single function block, `F_TIME_IN_US_TO_ULINT`, from the IEC 61131-3 standard library. Sequence control is purely event-driven:

1. An event at the input adapter **IN.E1** triggers the input **REQ** of the internal conversion block.
2. The data value **IN.D1** (`TIME`) is passed to the input **IN** of `F_TIME_IN_US_TO_ULINT`.
3. `F_TIME_IN_US_TO_ULINT` converts the time value into a microseconds figure of type `ULINT` and signals this via its output **CNF**.
4. This event is forwarded to the output adapter **OUT.E1**.
5. Simultaneously, the converted value (output **OUT** of `F_TIME_IN_US_TO_ULINT`) is passed to the data point **OUT.D1**.

The entire process is synchronous and without intermediate storage – each successful conversion immediately triggers an output event.

Technical Features

- **Unit conversion per IEC 61131-3**: The internal block `F_TIME_IN_US_TO_ULINT` interprets the `TIME` value (internally held in nanoseconds) as a figure in microseconds and returns the result as `ULINT` (64-bit unsigned integer).
- **Unidirectional adapters**: Both the input and output adapters are unidirectional. Reverse communication is not supported.
- **Composite structure without its own logic**: The block consists solely of a data and an event connection to the internal conversion block; there is no ECC of its own.
- **Empty TypeHash**: As with all generated conversion composites in this package, `eclipse4diac::core::TypeHash` is empty (`''`).

State Overview

The function block does not have its own state machine. The process can be described as a simple step:

Step	Action
1	Wait for event at **IN.E1**
2	Convert the data value from `TIME` (microseconds) to `ULINT` via `F_TIME_IN_US_TO_ULINT`
3	Output the converted value to **OUT.D1** and event to **OUT.E1**
4	Return to step 1

Application Scenarios

- **Displaying/processing durations**: A duration carried via an `ATM` adapter (e.g., a cycle time or delay) needs to be forwarded as a figure in microseconds to an adapter socket of type `AULI`, for example for display on a panel or further use in a calculation.

- **Interface adaptation**: In networks that consistently rely on adapter connections instead of classic data wiring, the block enables seamless conversion of a time adapter into a numeric adapter.

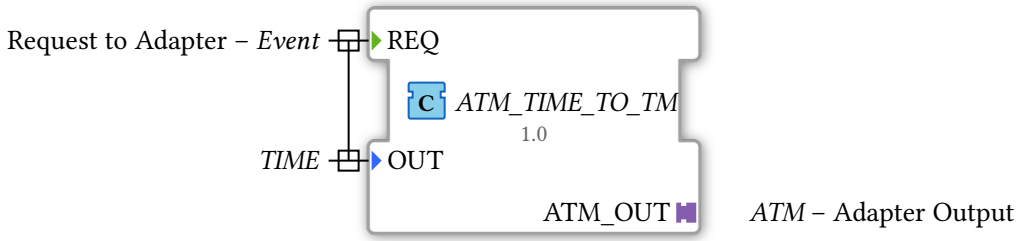
Comparison with similar function blocks

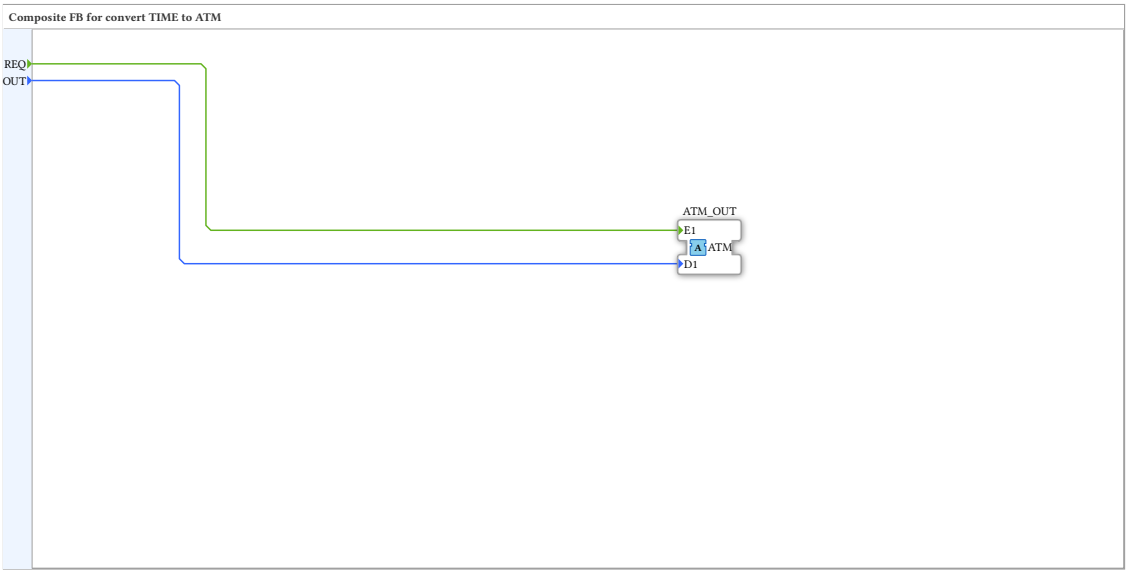
- **Same unit, other target types**: ATM_IN_US_TO_ADI, ATM_IN_US_TO_ALI, ATM_IN_US_TO_ALR, ATM_IN_US_TO_AUDI
- **Same target type, other units**: ATM_IN_MS_TO_AULI, ATM_IN_NS_TO_AULI, ATM_IN_S_TO_AULI
- **ATM_TIME_TO_TM / ATM_TM_TO_TIME**: pure adapter-to-variable passthrough without unit conversion, unlike the numeric conversion performed by `ATM_IN_US_TO_AULI`.

Conclusion

`ATM_IN_US_TO_AULI` provides a standards-compliant, fully adapter-based conversion of a `TIME` value in microseconds into a `ULINT` value, fitting seamlessly into networks that consistently rely on adapter connections instead of classic data wiring.

ATM_TIME_TO_TM





• – ○ • – ○ • – ○ •

Introduction

`ATM_TIME_TO_TM` is a composite function block that converts a plain `TIME` data value into the ATM adapter format. It follows exactly the same pattern as ADI_DINT_TO_DI (DINT→ADI) and AUI_UINT_TO_UI (UINT→AUI): a single data connection and a single event connection, passed straight through to the plug — no conversion logic of its own.

Interface Structure

Event Inputs

- **REQ**: request event that triggers the conversion; carries the value of `OUT` (`With Var="OUT"`).

Event Outputs

None of its own; the event flows exclusively through the ATM_OUT adapter plug.

Data Inputs

- **OUT** (TIME): the time value to convert.

Data Outputs

None of its own; the value is provided exclusively through the ATM_OUT adapter plug.

Adapters

- **ATM_OUT** (plug, type adapter::types::unidirectional::ATM): emits the value of OUT via its E1/D1 adapter event on REQ.

Functionality

When the REQ event arrives, it is forwarded directly to ATM_OUT.E1, while the current value of OUT is simultaneously placed onto ATM_OUT.D1. The ATM_OUT plug thus fires its adapter event carrying the supplied time value — no buffering, no change detection, freshly on every REQ.

Technical Details

- **Pure wiring logic**, no ECC of its own, no ST algorithms — a composite FB that establishes exactly one data and one event connection.
- **No REAL_TO_TIME exists in the standard libraries.** This block does not convert a numeric value into TIME — it accepts an already-finished TIME value and merely forwards it as an adapter. The actual conversion (e.g. from REAL seconds) happens beforehand, typically via iec61131::arithmetic::F_MULTIME (T#1s * REAL#seconds) — see the Override_Timer SubApp (FBs::sys, Getreideannahme application code, not part of this library's documentation) for a concrete usage example.
- **No event is suppressed for an unchanged value** — unlike, say, AX_ATM_FB_TON's ET output, there is no E_D_FF/E_D_FF_ANY buffering here; every REQ produces a fresh adapter event.

State Overview

The block is stateless: every REQ event immediately outputs the current OUT value via ATM_OUT.

Application Scenarios

- **Feeding sockets that expect an ATM adapter instead of a plain TIME variable**, e.g. PT on AX_ATM_FB_TON/ AX_ATM_FB_TOF/ AX_ATM_FB_TP.
- The final step in a conversion chain from REAL seconds to an adapter-based time value (F_MULTIME → ATM_TIME_TO_TM), as used in the Override_Timer SubApp (FBs::sys).

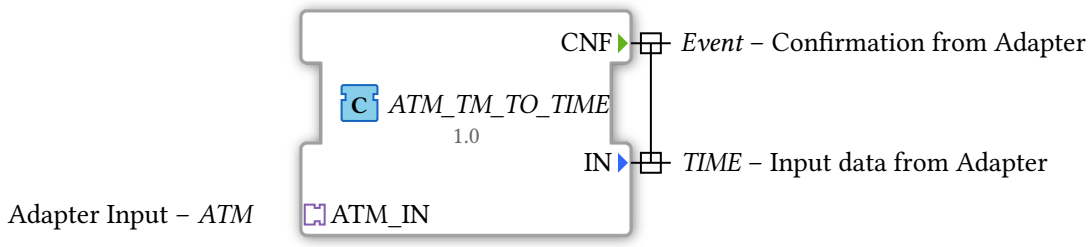
⚖️ Comparison with Similar Blocks

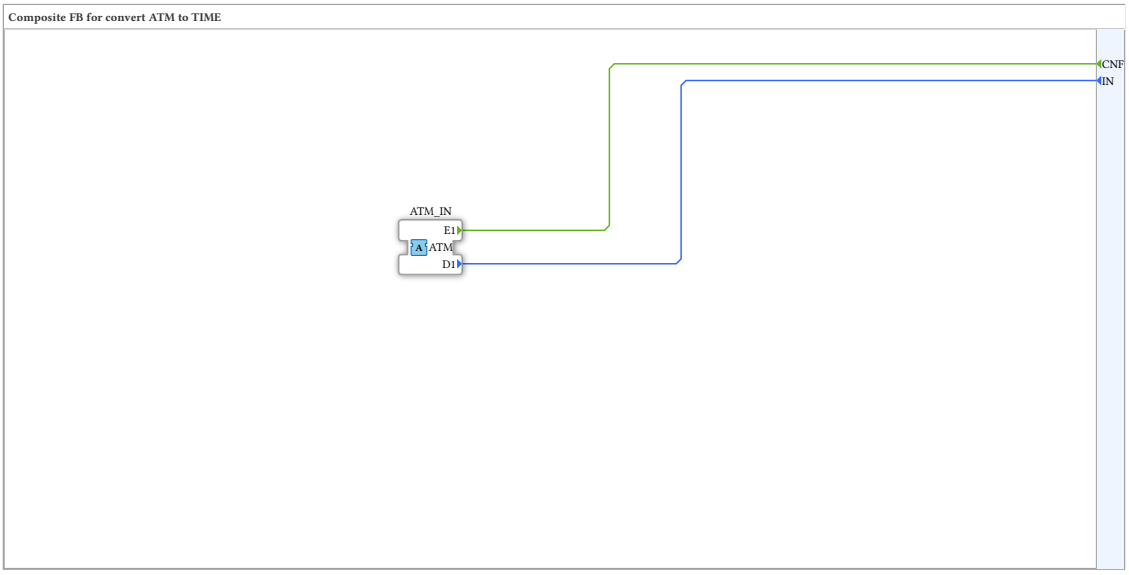
- ATM_TM_TO_TIME: the reverse direction — accepts an ATM adapter socket and exposes the value as a plain TIME data variable.
- ADI_DINT_TO_DI / AUI_UINT_TO_UI / AR_REAL_TO_R: the same "data value → adapter plug" pattern for DINT, UINT, and REAL instead of TIME.

Conclusion

ATM_TIME_TO_TM is the simplest way to expose a TIME data value as an ATM adapter plug, bridging the gap between numeric time calculations (e.g. via F_MULTIME) and adapter-based time sockets such as PT on AX_ATM_FB_TON/TOF/TP.

ATM_TM_TO_TIME





• – ○ • – ○ • – ○ •

Introduction

`ATM_TM_TO_TIME` is the reverse of ATM_TIME_TO_TM: a composite function block that accepts an ATM adapter socket and exposes its value as a plain `TIME` data variable. It follows exactly the same pattern as AUI_UI_TO_UINT (AUI→UINT): a single data connection and a single event connection, read straight off the socket — no conversion logic of its own.

Interface Structure

Event Inputs

None of its own; the event arrives exclusively through the `ATM_IN` adapter socket.

Event Outputs

- **CNF**: confirmation event, fires when the `ATM_IN` adapter socket fires its event; carries the value of `IN` (`With Var="IN"`).

Data Inputs

None of its own; the value arrives exclusively through the `ATM_IN` adapter socket.

Data Outputs

- **IN** (TIME): the time value received from the adapter.

Adapters

- **ATM_IN** (socket, type `adapter::types::unidirectional::ATM`): receives the time value from the connected plug.

Functionality

As soon as the connected plug fires an event on `ATM_IN`, `ATM_IN.E1` is forwarded directly to `CNF`, while `ATM_IN.D1` is simultaneously placed onto the data output variable `IN`. The block thus makes the value of an incoming `ATM` adapter usable by downstream blocks that expect a plain `TIME` variable rather than an adapter socket (e.g. `OSCAT::Basic::POUs::conversion::TIME_TO_REAL`, whose `TM` input is a plain data variable).

Technical Details

- **Pure wiring logic**, no ECC of its own, no ST algorithms — a composite FB that establishes exactly one data and one event connection.
- **Necessary because plugs cannot feed a plain data variable directly.** An adapter plug (e.g. the `ET` output of AX_ATM_FB_TON) can only connect to a socket, never directly to a function's plain `InputVar`. `ATM_TM_TO_TIME` closes that gap.

- **No event is suppressed for an unchanged value** — the block itself buffers nothing; whether an event actually only arrives on a value change depends on the upstream plug (e.g. `AX_ATM_FB_TON.ET` already has its own `E_D_FF_ANY` change detection).

State Overview

The block is stateless: every event on the `ATM_IN` socket immediately outputs the current value via `IN`/`CNF`.

Application Scenarios

- **Making an adapter plug readable for functions with plain data variables**, e.g. forwarding the `ET` output of AX_ATM_FB_TON/ AX_ATM_FB_TOF/ AX_ATM_FB_TP into `OSCAT::Basic::POUs::conversion::TIME_TO_REAL`.
- The first step in a conversion chain from an adapter-based time value back to REAL seconds (`ATM_TM_TO_TIME` → `TIME_TO_REAL` → `F_DIV`), as used in the `Override_Timer` SubApp (`FBs::sys`, Getreideannahme application code, not part of this library's documentation).

⚖️ Comparison with Similar Blocks

- **ATM_TIME_TO_TM**: the reverse direction — accepts a plain `TIME` data variable and exposes it as an `ATM` adapter plug.
- **AUI_UI_TO_UINT / ADI_DI_TO_DINT**: the same "adapter socket → data value" pattern for UINT and DINT instead of TIME.

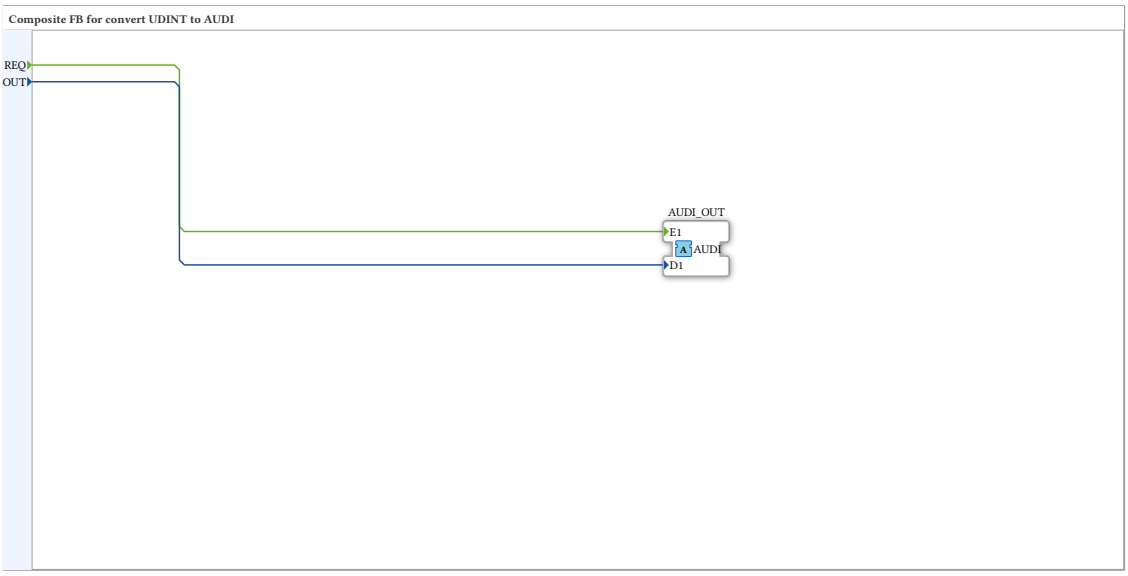
Conclusion

`ATM_TM_TO_TIME` is the simplest way to make an `ATM` adapter plug's value usable as a plain `TIME` variable, bridging the gap between adapter-based time plugs (e.g. `ET` on `AX_ATM_FB_TON/TOF/TP`) and functions that only accept plain data variables.

Numerical Index (unidirectional)

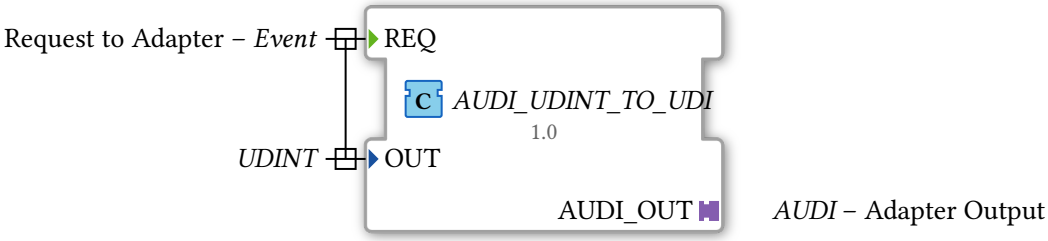
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AUDI_UDINT_TO_UDI



Introduction

The AUDI_UDINT_TO_UDI function block is a composite function block used to convert a UDINT value to the AUDI adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: UDINT input value to be converted

Data Outputs

- No direct data outputs available

Adapter

- **AUDI_OUT**: Plug of type `AUDI`

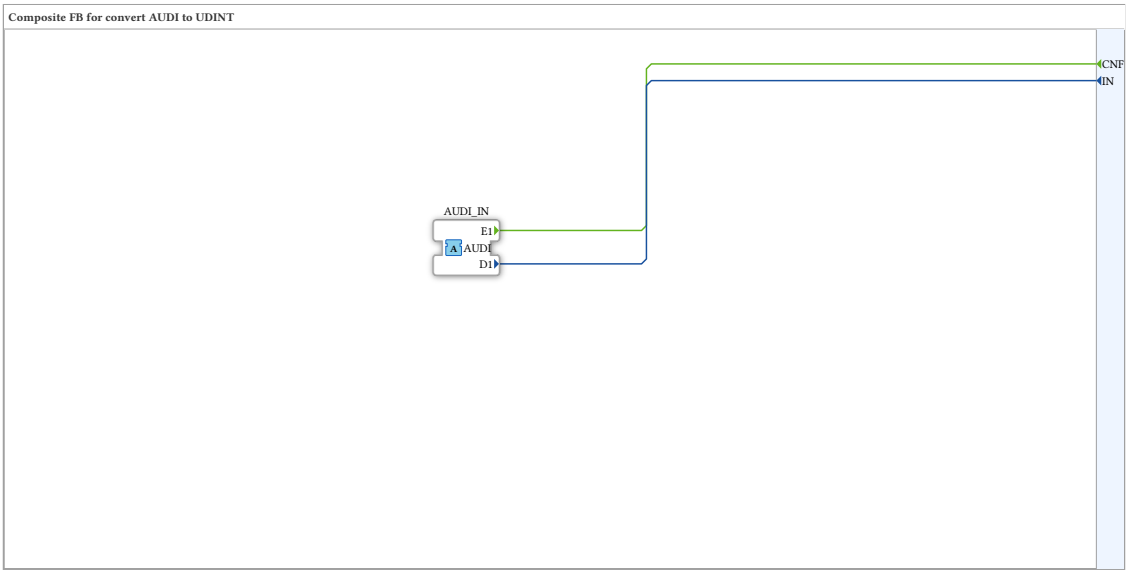
Functionality

The function block receives a UDINT value at the OUT data input and converts it to the AUDI adapter format upon arrival of the REQ event.

🛠️ Related Exercises

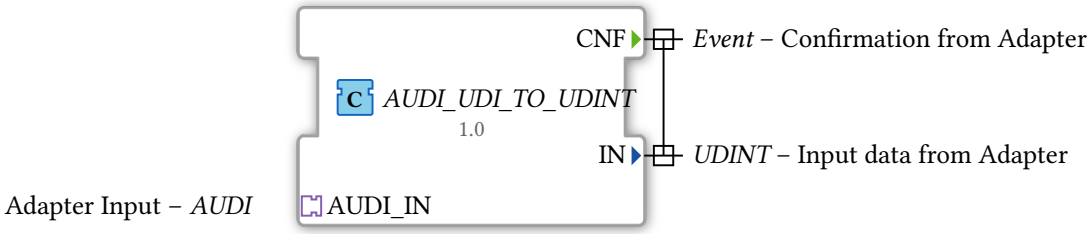
- Exercise_083_AX

AUDI_UDI_TO_UDINT



Introduction

The AUDI_UDI_TO_UDINT function block is a composite function block used to convert the AUDI adapter format to a UDINT value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: UDINT output value of the converted data

Adapter

- **AUDI_IN**: Socket of type `AUDI`

Functionality

The function block receives data via the AUDI adapter socket and makes the contained UDINT value available at the IN output as soon as an event arrives via the adapter (CNF).

UDINT conversion

🛠️ Related exercises

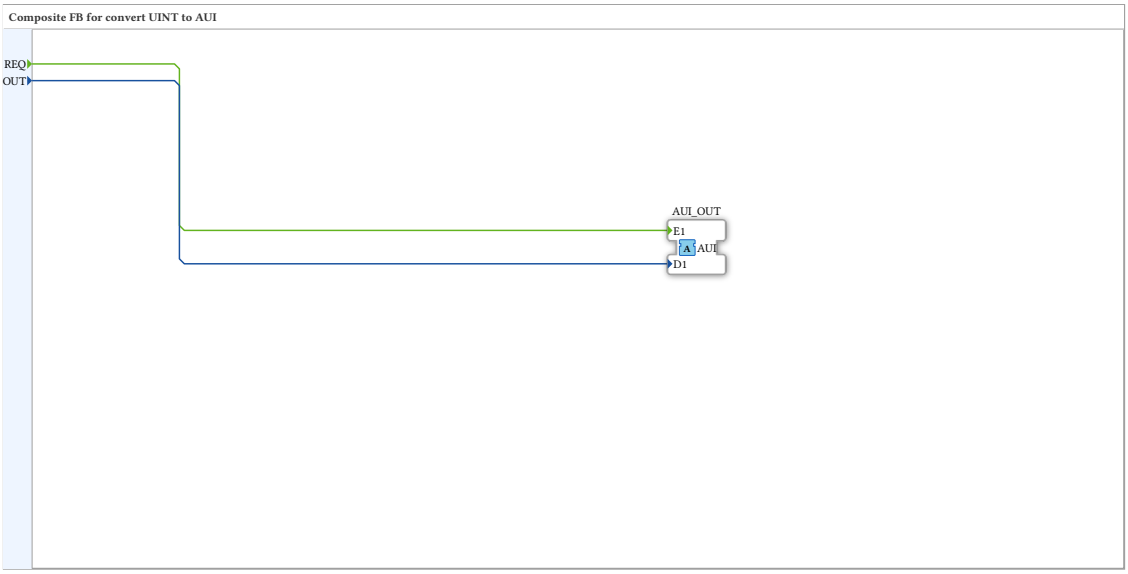
- Uebung_009
- Uebung_009_AX
- Uebung_009a
- Uebung_011
- Uebung_011a
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_012c
- Uebung_015
- Uebung_015a
- Uebung_020c2_sub
- Uebung_028
- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_034b
- Uebung_072c
- Uebung_083
- Uebung_083_AX
- Uebung_103
- Uebung_111
- Uebung_151_AX

- Uebung_152
- Uebung_153

Numerical Index (unidirectional)

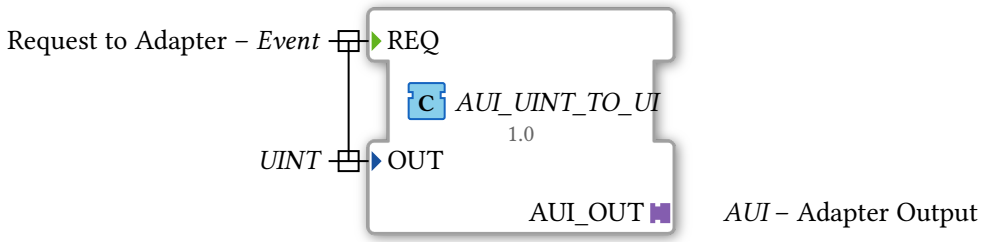
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AUI_UINT_TO_UI



Introduction

The AUI_UINT_TO_UI function block is a composite function block used to convert a UINT value to the AUI adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: UINT input value to be converted

Data Outputs

- No direct data outputs available

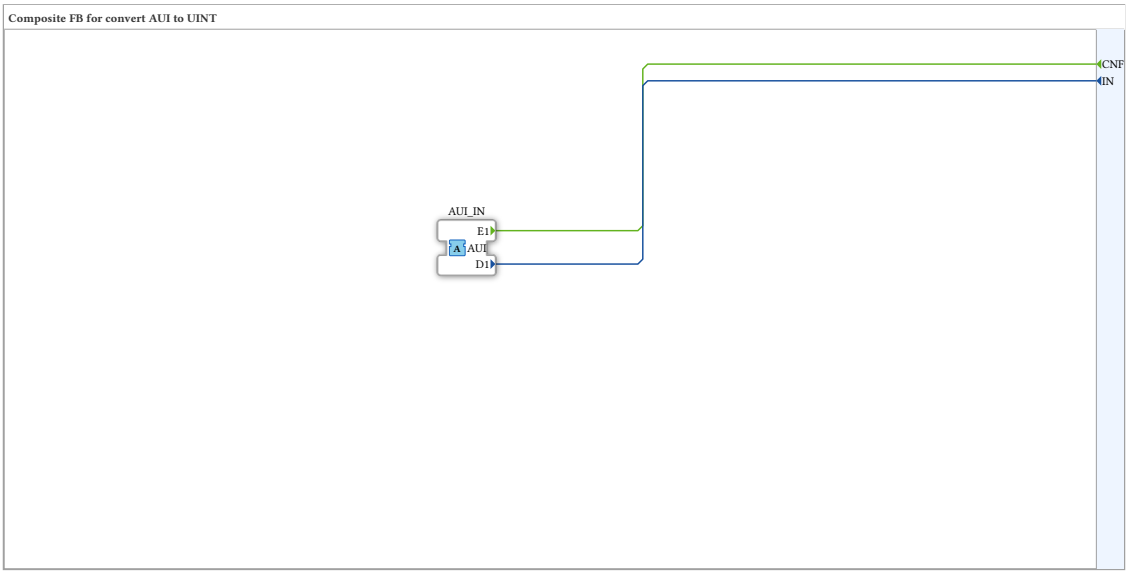
Adapter

- **AUI_OUT**: Plug of type `AUI`

Functionality

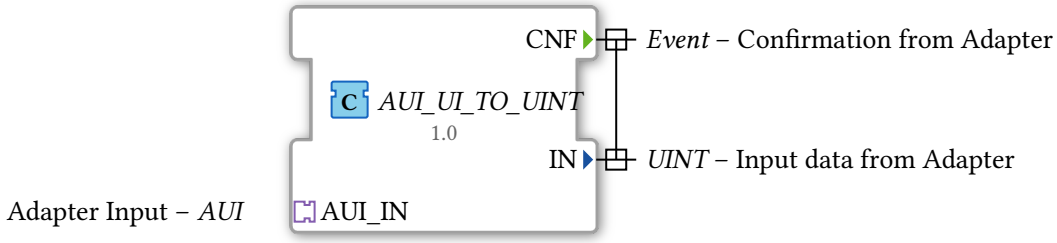
The function block receives a UINT value at the data input OUT and converts it to the AUI adapter format upon arrival of the REQ event.

AUI_UI_TO_UINT



Introduction

The AUI_UI_TO_UINT function block is a composite function block used to convert the AUI adapter format to a UINT value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: UINT output value of the converted data

Adapter

- **AUI_IN**: Socket of type `AUI`

Functionality

The function block receives data via the AUI adapter socket and makes the contained UINT value available at the IN output as soon as an event arrives via the adapter (CNF).

UINT conversion

🛠️ Related exercises

- Uebung_006c
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010c2
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- NumericValue_ID_TO_NVS
- Uebung_020c2_sub
- IX_TO_logiBUS_LED_strip_QX
- Uebung_035
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- Uebung_039_sub_NumbAnsicht
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_040
- Uebung_040_AX
- Uebung_041
- Uebung_087
- Uebung_087a1
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2
- Uebung_090a2_AX
- Uebung_103

Numerical Index (unidirectional)

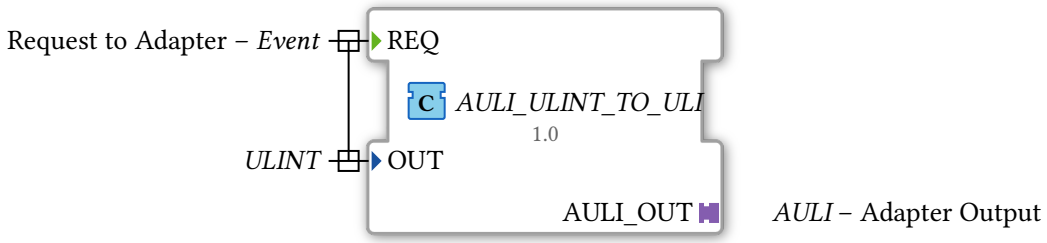
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AULI_ULINT_TO_ULI



Introduction

The AULI_ULINT_TO_ULI function block is a composite function block used to convert a ULINT value to the AULI adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: ULINT input value to be converted

Data Outputs

- No direct data outputs available

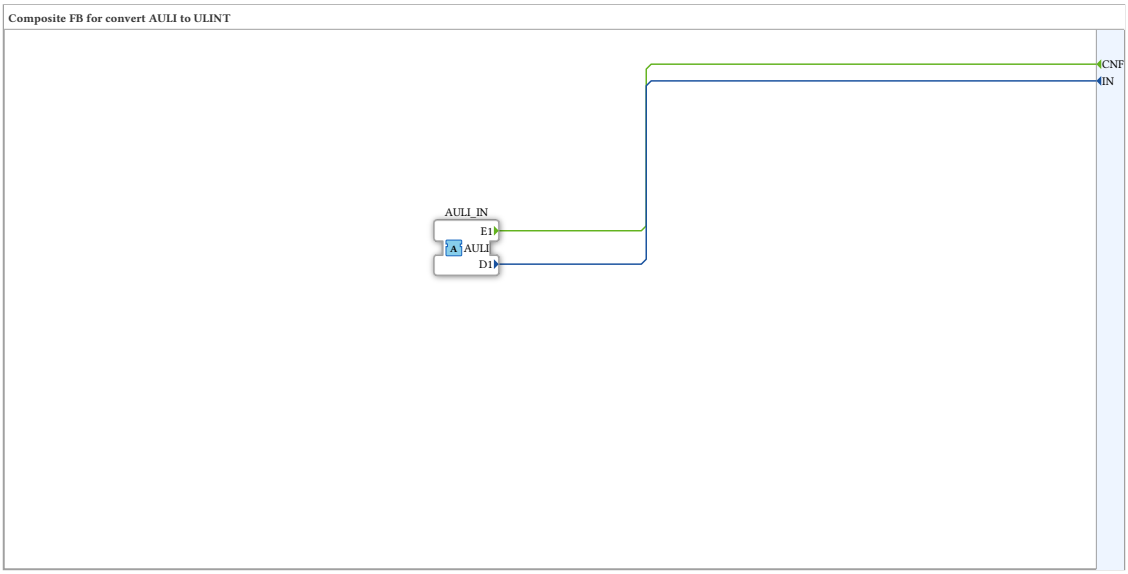
Adapter

- **AULI_OUT**: Plug of type `AULI`

Functionality

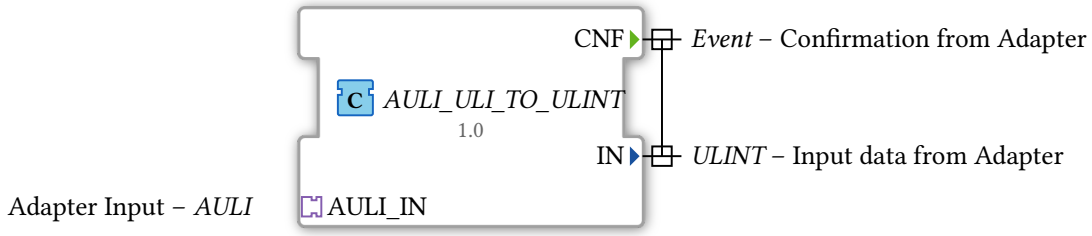
The function block receives a ULINT value at the OUT data input and converts it to the AULI adapter format upon arrival of the REQ event.

AULI_ULI_TO_ULINT



Introduction

The AULI_ULI_TO_ULINT function block is a composite function block used to convert the AULI adapter format to a ULINT value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: ULINT output value of the converted data

Adapter

- **AULI_IN**: Socket of type `AULI`

Functionality

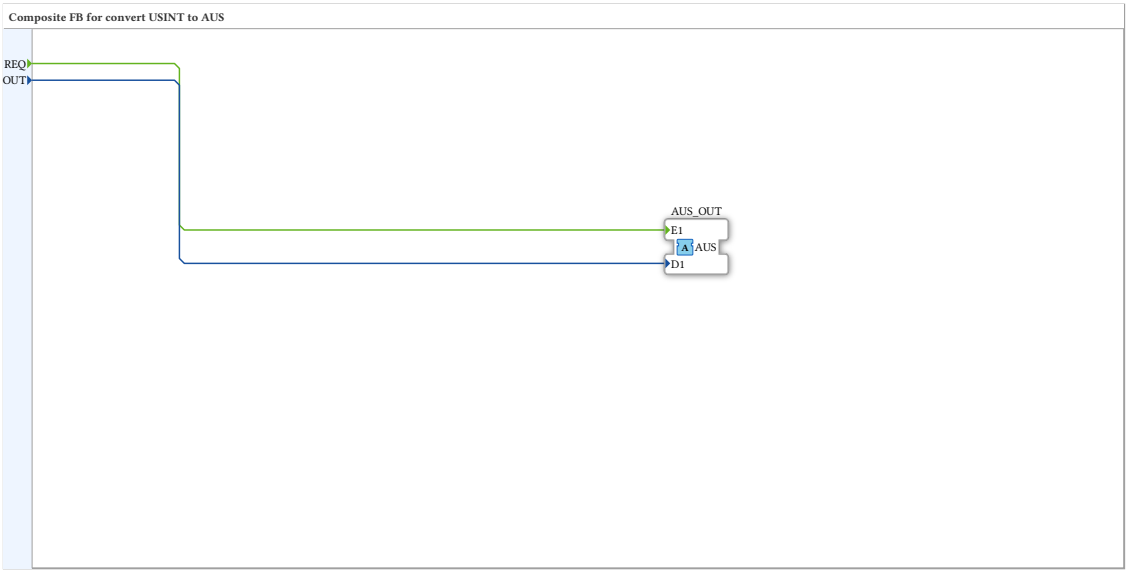
The function block receives data via the AULI adapter socket and makes the contained ULINT value available at the IN output as soon as an event arrives via the adapter (CNF).

ULINT Conversion

Numerical Index (unidirectional)

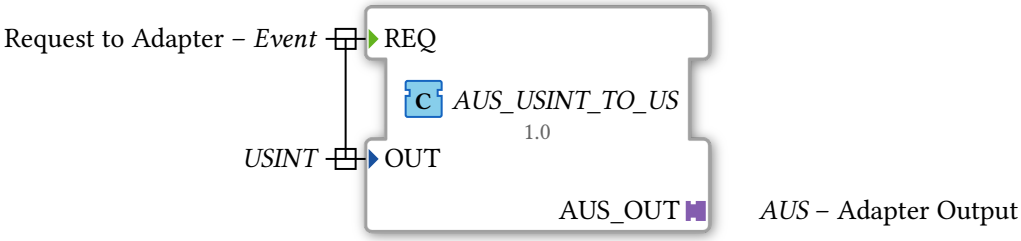
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AUS_USINT_TO_US



Introduction

The AUS_USINT_TO_US function block is a composite function block used to convert a USINT value to the AUS adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: USINT input value to be converted

Data Outputs

- No direct data outputs available

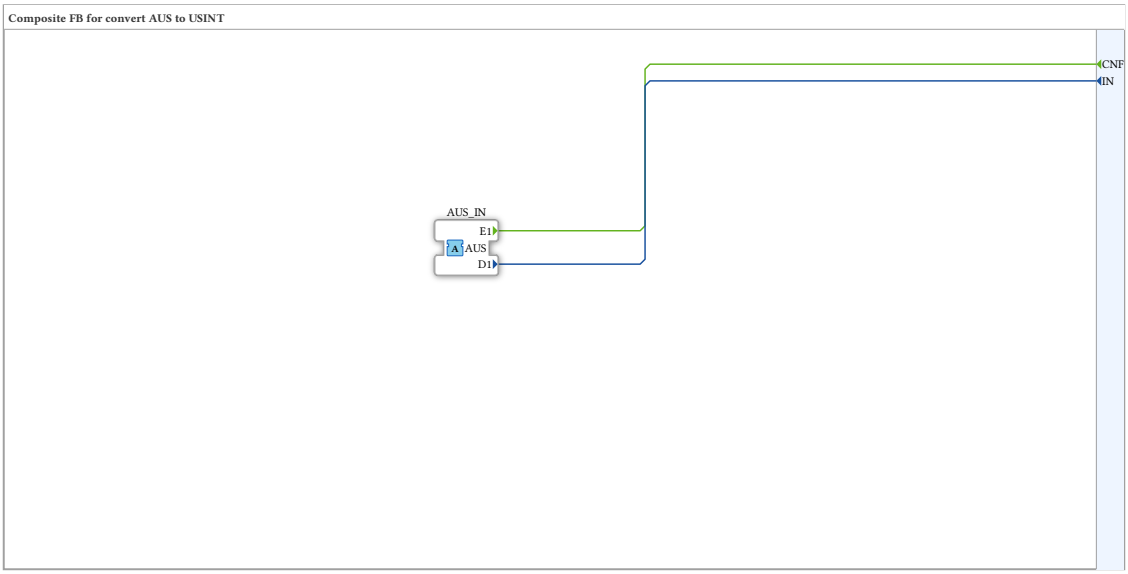
Adapter

- **AUS_OUT**: Plug of type `AUS`

Functionality

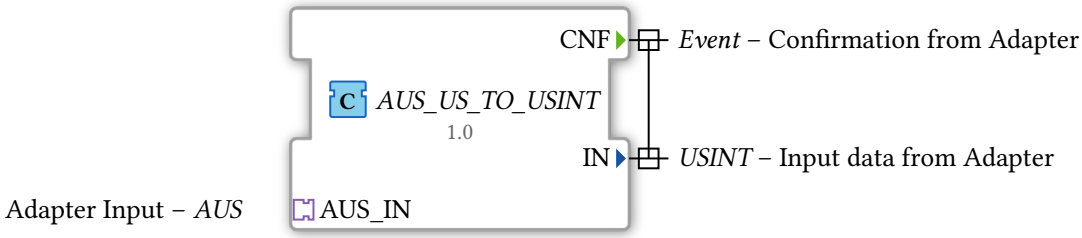
The function block receives a USINT value at the OUT data input and converts it to the AUS adapter format upon arrival of the REQ event.

AUS_US_TO_USINT



Introduction

The AUS_US_TO_USINT function block is a composite function block used to convert the AUS adapter format to a USINT value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: USINT output value of the converted data

Adapter

- **AUS_IN**: Socket of type `AUS`

Functionality

The function block receives data via the AUS adapter socket and makes the contained USINT value available at the IN output as soon as an event arrives via the adapter (CNF).

USINT conversion

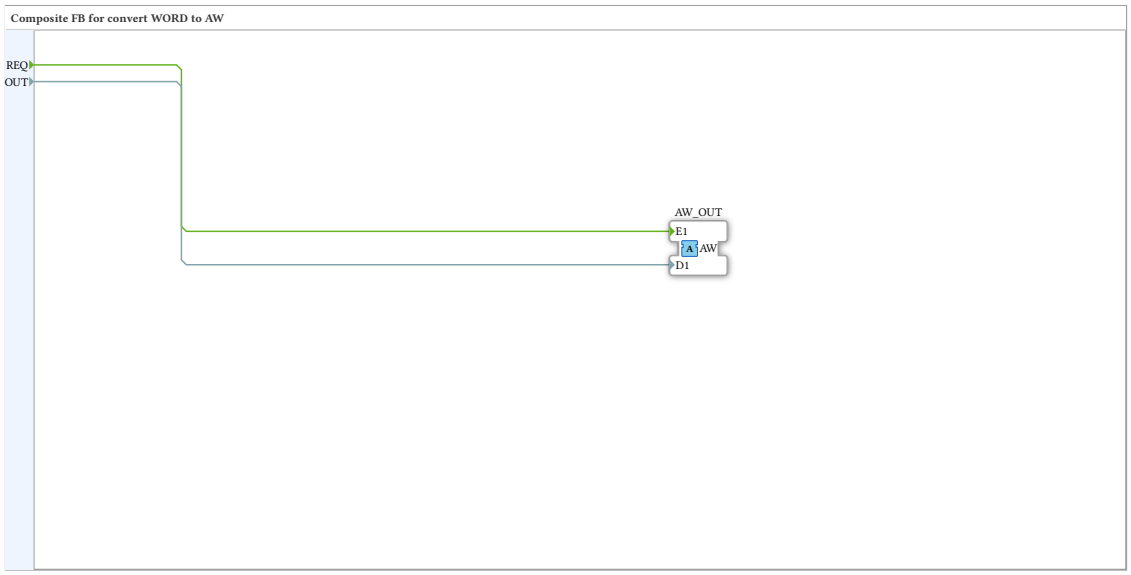
🛠️ Related exercises

- Funk_IX_TO_DataPanel_MI_QX
- Funk_IXA_TO_DataPanel_MI_QXA
- IX_TO_Aux_QX
- IXA_TO_Aux_QXA
- IX_TO_logiBUS_LED_strip_QX
- SIN_TO_PCAN_Callback_Byte

Numerical Index (unidirectional)

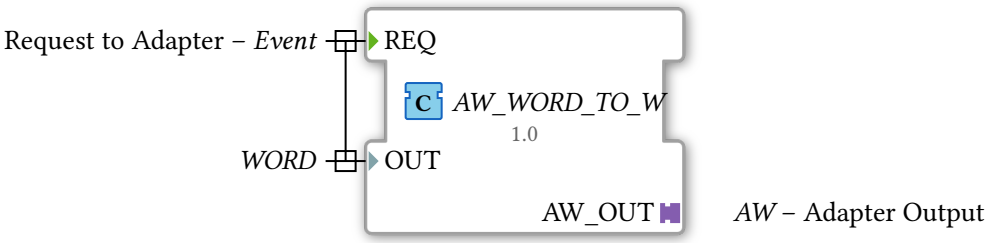
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AW_WORD_TO_W



Introduction

The AW_WORD_TO_W function block is a composite function block used to convert a WORD value to the AW adapter format.



Interface Structure

Event Inputs

- **REQ**: Request event to trigger the conversion

Event Outputs

- No direct event outputs available

Data Inputs

- **OUT**: WORD input value to be converted

Data Outputs

- No direct data outputs available

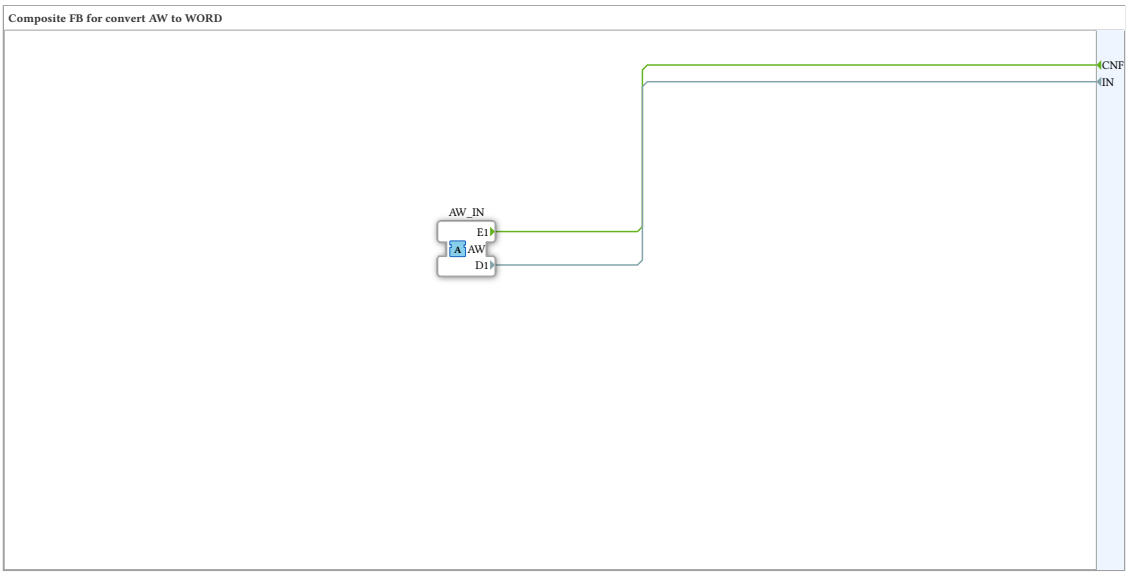
Adapter

- **AW_OUT**: Plug of type `AW`

Functionality

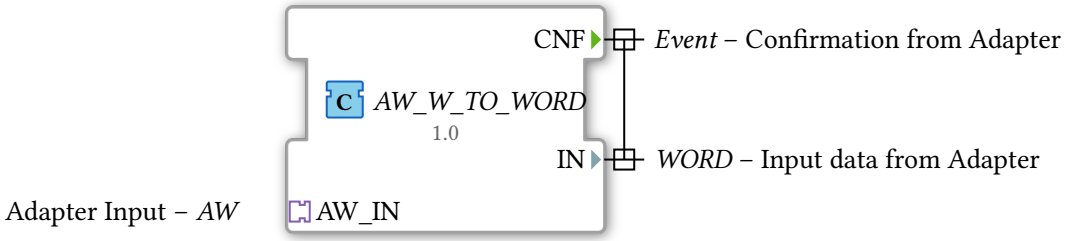
The function block receives a WORD value at the OUT data input and converts it to the AW adapter format upon arrival of the REQ event.

AW_W_TO_WORD



Introduction

The AW_W_TO_WORD function block is a composite function block used to convert the AW adapter format to a WORD value.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- **CNF**: Confirmation event after successful conversion

Data Inputs

- No direct data inputs available

Data Outputs

- **IN**: WORD output value of the converted data

Adapter

- **AW_IN**: Socket of type `AW`

Functionality

The function block receives data via the AW adapter socket and makes the contained WORD value available at the IN output as soon as an event arrives via the adapter (CNF).

Word Conversion

2.1.11.9 events

Podcast

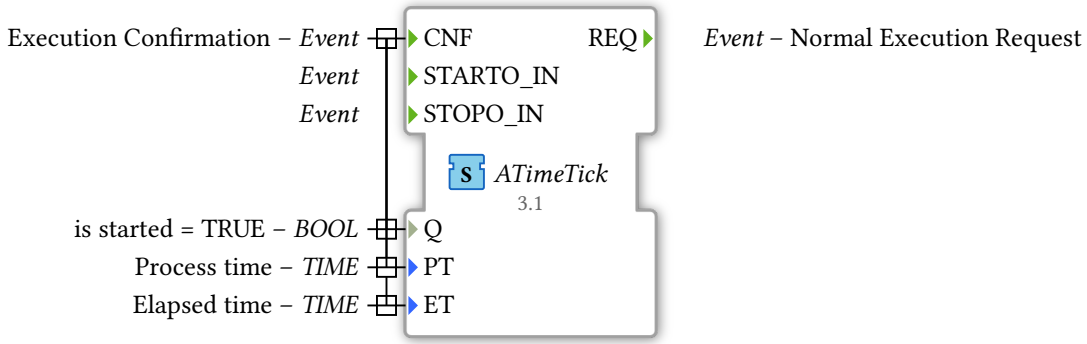
- Taster & Events: GPIO Basics for Agricultural and Construction Machinery Mechatronics Qzmsdocs000000 Qz

---

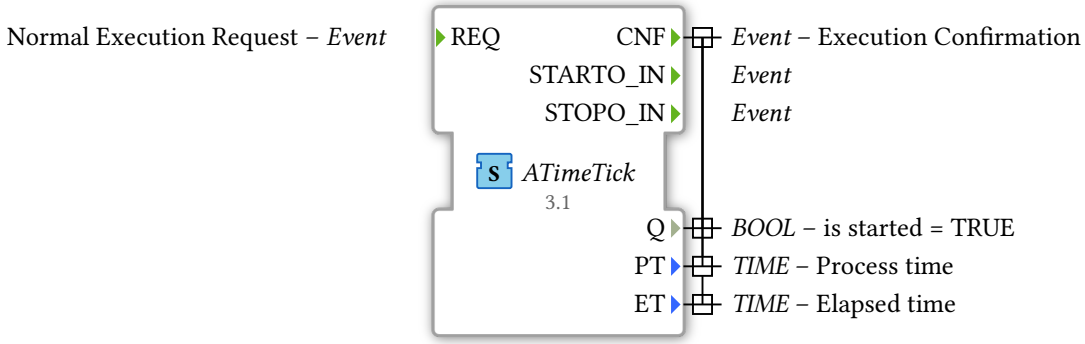
TimeOut

ATimeTick

Plug (adapter output)



Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

The ATimeTick adapter provides a standardized interface for a time-out service, inspired by the ROOM modeling approach. It enables the coordination between a client and a server for starting, stopping, and confirming time-based operations. The adapter supports both plug and socket roles, facilitating communication between function blocks in an IEC 61499 system.

Interface Structure

The adapter exposes the following interface elements:

Event Inputs

- **CNF**: Execution confirmation event, triggered when the time-out service completes or acknowledges a request. It carries associated data (Q, ET, PT).
- **STARTO_IN**: Event to start the time-out operation (input).
- **STOPO_IN**: Event to stop the time-out operation (input).

Event Outputs

- **REQ**: Event output that issues a request to the counterpart, typically used to initiate or query the time-out service.

Data Inputs

- **Q** (BOOL): Indicates whether the time-out is currently active (TRUE when started).
- **PT** (TIME): Process time – the configured time-out duration.
- **ET** (TIME): Elapsed time – the time that has already advanced.

Data Outputs

The adapter has no data outputs. All data is passed as inputs, primarily associated with the CNF event.

Adapters

This is an adapter type itself, so it is intended to be used as a plug or socket in a connection. It defines a service interface with two service sequences: “Timeout” and “NormalOperation”.

Functionality

The ATimeTick adapter manages the lifecycle of a time-out operation. It receives start and stop commands (STARTO_IN, STOPO_IN) and provides a confirmation (CNF) after a time-out event. The REQ output serves as a request trigger to the paired service. The data inputs (Q, PT, ET) allow the status and timing parameters to be monitored.

The service definition shows two interaction patterns:

- **Timeout**: A start command (START with parameters TD) is propagated from the plug to the socket. Then, when a time-out occurs, the socket sends a TimeOut event back to the plug.
- **NormalOperation**: Start and stop (START/STOP) events are passed between plug and socket, without an explicit time-out event—likely for normal operation where the time-out is not triggered.

This implies that the adapter can be used in two modes: one where a time-out triggers an output, and another where it simply manages start/stop sequences.

Technical Features

- **Service Type**: Supports both Plug and Socket roles, enabling bidirectional communication.
- **Event-Driven**: All operations are event-based, ensuring asynchronous coordination.
- **Data Association**: The CNF event is linked with Q, ET, and PT, allowing the confirmation to carry status and timing information.
- **Dynamic Timing**: The PT and ET parameters allow flexible configuration and monitoring of time intervals.
- **Standards Compliance**: Designed according to IEC 61499, with a version history and contributions from multiple organisations.

State Overview

The adapter does not maintain an internal state machine within its definition; instead, it relies on the event sequences defined in the service. However, the Q data input provides a Boolean flag that can be interpreted as the active/inactive state of the time-out. The transitions are triggered by the events STARTO_IN (start), STOPO_IN (stop), and CNF (time-out occurrence).

Application Scenarios

- **Timeout Monitoring**: Use the adapter to enforce a time limit on a process. A function block can send STARTO_IN with a desired PT, and the service will respond with CNF after the time expires, allowing the block to react.
- **Process Synchronization**: In normal operation mode, the adapter can coordinate start/stop signals between two function blocks without triggering a time-out, useful for handshake protocols.
- **Error Detection**: By monitoring the ET value, it is possible to detect when a process takes longer than expected.

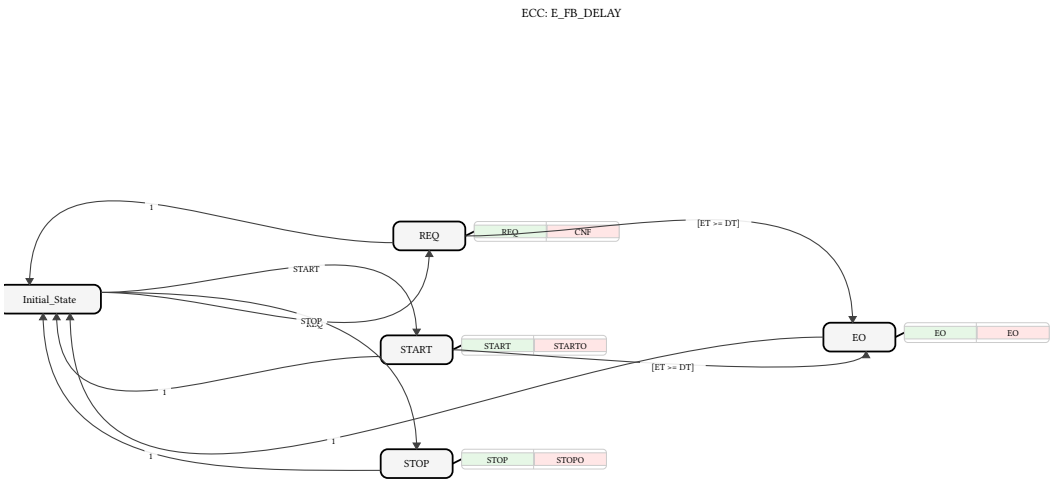
Comparison with Similar Blocks

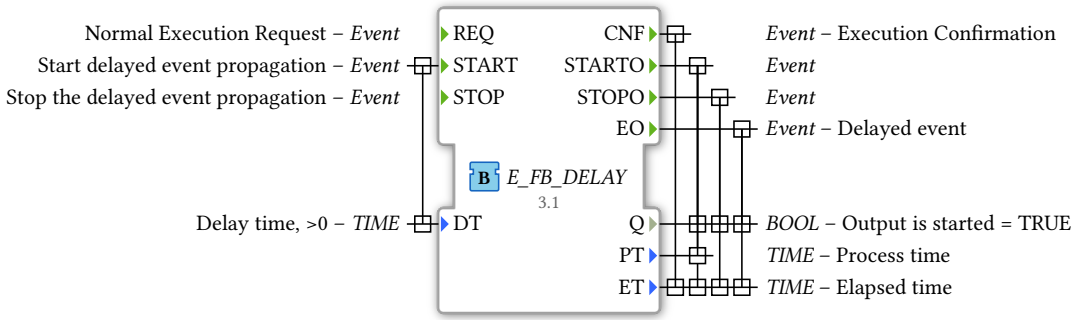
Unlike standard timer function blocks (e.g., TON, TOF) which are self-contained, ATimeTick acts as an interface adapter that separates the timing logic from the requesting block. This promotes reusability and decoupling. Compared to generic adapter types, ATimeTick is specialised for time-out services, providing a clear contract for start/stop and confirmation events.

Conclusion

The ATimeTick adapter is a valuable component for implementing time-based coordination in IEC 61499 systems. Its clear event and data interface, along with its plug/socket capability, allows for flexible integration into distributed control applications. By following a ROOM-inspired design, it offers a structured approach to managing time-outs and normal operation sequences.

E_FB_DELAY





• – ○ • – ○ • – ○ •

Introduction

The **E_FB_DELAY** function block is a basic IEC 61499 FB designed to propagate an event after a predefined delay time. It provides a cyclic interface (similar to a `TON` timer) allowing external logic to start, stop, and poll the elapsed time while the delay is active. When the delay time is reached, an output event `EO` is emitted, indicating that the desired time has elapsed. The FB is particularly useful in scenarios where a timed response or a scheduled action is required, and it can be driven by an external cyclic trigger (e.g., an `E_CYCLE` block) to keep the elapsed time up-to-date.

Interface Structure

Event Inputs

- **REQ** (`Event`): Normal execution request. When the timer is armed (`Q = TRUE`), this event causes the FB to update the elapsed time (`ET`) by reading the system monotonic clock.
- **START** (`Event`): Starts the delayed event propagation. When this event is received, the FB records the current time, sets the output `Q` to `TRUE`, and resets `ET` to zero.

- **STOP** (Event): Stops the delayed event propagation. It cancels the active timer without generating an EO event; the output Q is cleared and ET is reset to zero.

Event Outputs

- **CNF** (Event): Execution confirmation. Emitted whenever the REQ algorithm completes (i.e., after an execution request).
- **STARTO** (Event): Confirmation that the timer has been started. Emitted after the START algorithm executes.
- **STOPO** (Event): Confirmation that the timer has been stopped. Emitted after the STOP algorithm executes.
- **EO** (Event): Delayed event. Emitted when the elapsed time (ET) reaches or exceeds the delay time (DT). This is the main output that signals the completion of the delay period.

Data Inputs

- **DT** (TIME): Delay time. Must be greater than zero. This is the duration the FB waits before generating the EO event.

Data Outputs

- **Q** (BOOL): Output indicating that the timer is currently armed (TRUE when the delay is in progress, FALSE otherwise).
- **PT** (TIME): Preset time (SOLL). Set to DT at START. It is provided as an output for informational purposes.
- **ET** (TIME): Elapsed time. Shows how much time has passed since the timer was started. This value is only updated when the REQ event is triggered while the timer is armed.
- **REST** (TIME): Remaining time = PT - ET. Set to PT at START, tracked on REQ, and reset to zero on STOP/EO.

Adapters

No adapters are defined for this FB.

Functionality

The E_FB_DELAY operates as a cyclic timer that can be started, polled, and stopped. The core logic is implemented as a basic FB with an ECC (Execution Control Chart) and four algorithms:

1. **START**:
   - Records the current monotonic time into the internal variable StartTime.
   - Sets Q = TRUE to indicate the timer is armed.
   - Resets ET = T#0s.
2. **REQ**:
   - If Q = TRUE (timer is armed), it reads the current monotonic time and calculates ET = NOW_MONOTONIC() - StartTime.
   - If Q = FALSE, it performs no operation.
   - This allows an external cyclic trigger (e.g., an E_CYCLE block) to refresh ET periodically.
3. **STOP**:
   - Clears Q = FALSE.
   - Resets ET = T#0s.
   - No EO event is generated.
4. **EO**:
   - Executed when the condition [ET >= DT] is true after a REQ or START event.
   - Clears Q = FALSE.
   - Resets ET = T#0s.
   - This prevents the condition from remaining true, so an EO event is fired only once per active delay period.

The ECC defines the state transitions that drive these algorithms. The block begins in Initial_State. From there:

- REQ → goes to REQ state, executes algorithm REQ, outputs CNF, then either transitions to EO (if ET >= DT) or back to Initial_State.

- START → goes to START state, executes algorithm START, outputs STARTO, then checks ET >= DT; if true, moves to EO, otherwise returns to Initial_State.
- STOP → goes to STOP state, executes algorithm STOP, outputs STOPO, and returns to Initial_State.
- From EO state (which executes EO and outputs the EO event), it always returns to Initial_State.

The block is intended to be driven cyclically: while a delay is active, the application should send REQ events (e.g., from a periodic cyclic function block) to update ET. When ET reaches DT, the FB will generate an EO event automatically. The START event can be used to (re)arm the timer, and STOP can cancel it prematurely.

Technical Features

- **Monotonic Clock Usage**: The FB uses NOW_MONOTONIC() to obtain a monotonic time source, ensuring that the elapsed time is not affected by system clock adjustments.
- **Single-Shot Delay**: The EO event is generated exactly once per START cycle. After EO is emitted, the timer is automatically disarmed and ET reset, so subsequent REQ events will not re-trigger EO.
- **Cyclic Polling Support**: The REQ event allows an external cyclic trigger to update ET continuously. This makes the block suitable for applications where the delay time must be monitored in real time.
- **Simple Interface**: The block exposes only the essential inputs and outputs, making it easy to integrate into larger IEC 61499 systems.
- **No Persistence**: The FB is stateless between activations (except for the StartTime internal variable), but after a START/STOP/EO sequence, all state variables are reset.

State Overview

The ECC contains five states:

1. **Initial_State**: The idle state where no timer is active. The block remains here until a START, STOP, or REQ event is received.
2. **REQ**: A transient state triggered by the REQ event. Executes the REQ algorithm and outputs CNF. After that, it either transitions to EO (if the delay time has elapsed) or returns to Initial_State.
3. **START**: A transient state triggered by the START event. Executes the START algorithm and outputs STARTO. Then it checks if the delay time has already been reached (unlikely since the timer was just started) and transitions to EO if so, otherwise returns to Initial_State.
4. **STOP**: A transient state triggered by the STOP event. Executes the STOP algorithm and outputs STOPO, then returns to Initial_State.
5. **EO**: A transient state that is entered when the condition [ET >= DT] is satisfied. Executes the EO algorithm, outputs the EO event, and then returns to Initial_State.

The transitions are summarized as follows:

- Initial_State → REQ (on REQ)
- Initial_State → START (on START)
- Initial_State → STOP (on STOP)
- REQ → EO (if [ET >= DT])
- REQ → Initial_State (always, after executing REQ and CNF, if condition not met)
- START → EO (if [ET >= DT] – normally false immediately after start)
- START → Initial_State (always, after executing START and STARTO, if condition not met)
- STOP → Initial_State (always, after executing STOP and STOPO)
- EO → Initial_State (always, after executing EO)

Application Scenarios

- **Timed Output Generation**: Trigger an action (e.g., a valve open/close, a light, a data acquisition) after a configurable delay.
- **Synchronization with Cyclic Tasks**: When integrated with a cyclic execution framework (e.g., an E_CYCLE block that sends REQ periodically), the FB provides an accurate timer that can be polled without blocking the system.
- **Timeout Monitoring**: Use START to arm a watchdog, poll ET via REQ to monitor progress, and receive EO if the timeout occurs. STOP can be used to cancel the watchdog when the expected action happens earlier.
- **Sequence Control**: In a sequential logic, the block can be used to enforce a minimum dwell time in a state before allowing a transition to the next step.

Comparison with Similar Blocks

Compared to the standard IEC 61499 **TON** (Timer On-Delay) function block, **E_FB_DELAY** offers a more explicit cyclic interface:

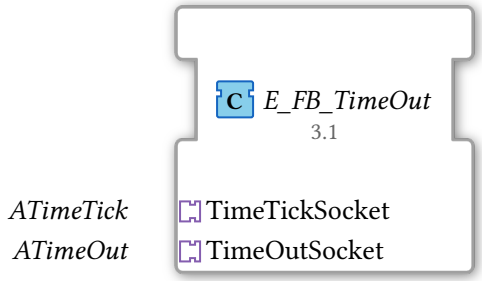
- It requires an external `REQ` event to update `ET`, while a `TON` block typically updates its elapsed time automatically within its own execution context.
- The `REQ` mechanism gives the developer control over when time evaluation occurs, which can be beneficial in deterministic or event-driven systems.
- The block provides separate output events for start/stop confirmation (`STARTO`, `STOPO`) and the actual delay completion (`EO`), whereas a `TON` block typically outputs a boolean `Q` and an elapsed time `ET` but no explicit event for "delay reached".
- The use of `NOW_MONOTONIC()` and the internal `StartTime` variable ensures high accuracy and independence from system time adjustments.

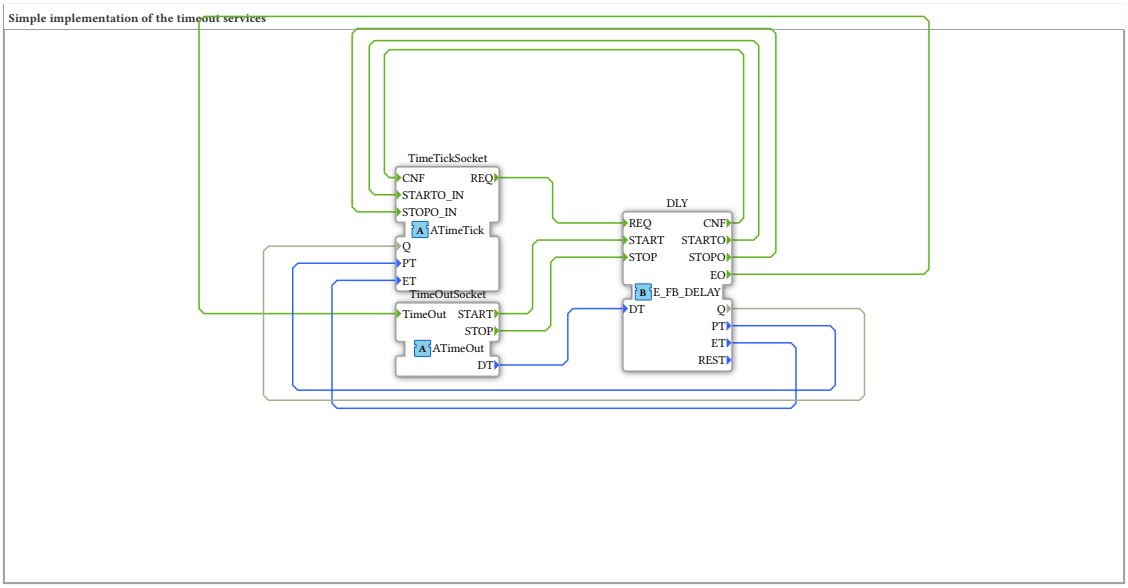
In essence, **E_FB_DELAY** is a more event-oriented timer that fits well into the IEC 61499 event-driven model, while a `TON` block is more oriented toward continuous monitoring.

Conclusion

The **E_FB_DELAY** function block provides a robust and flexible way to implement delayed event propagation in IEC 61499 applications. Its cyclic interface, combined with explicit start/stop controls and a single-shot delay output, makes it suitable for a wide range of timing requirements. The block is implemented as a basic FB with a clear ECC, making it easy to analyze, test, and maintain. Its use of a monotonic clock ensures reliable timing, and its design aligns with the event-driven philosophy of the 4diac IDE and the IEC 61499 standard.

E_FB_TimeOut





• – ○ • – ○ • – ○ •

Introduction

E_FB_TimeOut is a composite IEC 61499 function block that provides a simple timeout service. It wraps an internal timer instance and exposes its functionality through two adapter sockets:

- `TimeOutSocket` for starting, stopping, and receiving timeout events.
- `TimeTickSocket` for cyclic querying and monitoring of the timer state.

The block is designed to offer an `FB_TON`-like behavior with an event-driven cyclic interface. It is suitable for applications where a timeout must be armed, cancelled, and periodically observed.

Interface Structure

The function block has no direct event or data pins. All communication with the outside world is performed through adapter sockets.

Adapter Socket	Adapter Type	Purpose
`TimeOutSocket`	`iec61499::events::ATimeOut`	Standard timeout control and timeout notification.
`TimeTickSocket`	`adapter::events::TimeOut::ATimeTick`	Cyclic query and status observation of the timer.

Event Inputs

There are no direct event inputs on the FB. The following incoming events are provided through the adapter sockets:

Event	Adapter Socket	Description
`START`	`TimeOutSocket`	Starts the timeout evaluation.
`STOP`	`TimeOutSocket`	Stops or cancels the currently active timeout.
`REQ`	`TimeTickSocket`	Requests the current timer state; answered with `CNF`.

Event Outputs

There are no direct event outputs on the FB. The following outgoing events are provided through the adapter sockets:

Event	Adapter Socket	Description
`TimeOut`	`TimeOutSocket`	Emitted when the configured timeout duration has elapsed.
`CNF`	`TimeTickSocket`	Confirmation for a `REQ` query.
`STARTO_IN`	`TimeTickSocket`	Notifies that the internal timer has started.
`STOPO_IN`	`TimeTickSocket`	Notifies that the internal timer has stopped.

Data Inputs

There are no direct data inputs on the FB. The following incoming data is provided through the adapter sockets:

Data	Adapter Socket	Description
`DT`	`TimeOutSocket`	Timeout duration value forwarded to the internal timer.

Data Outputs

There are no direct data outputs on the FB. The following outgoing data is provided through the adapter sockets:

Data	Adapter Socket	Description
`ET`	`TimeTickSocket`	Elapsed time of the internal timer.
`Q`	`TimeTickSocket`	Timer state output of the internal timer.

PT	TimeTickSocket	Configured or current timer time value.

The data outputs are meaningful after an associated event such as CNF, STARTO_IN, STOPO_IN, or TimeOut has been issued.

Adapters

The FB uses two adapter sockets as its complete external interface:

- TimeOutSocket provides the standard timeout operations:
  - START and STOP as incoming events.
  - TimeOut as outgoing event.
  - DT as incoming data.
- TimeTickSocket provides a cyclic observation interface:
  - REQ as incoming query event.
  - CNF, STARTO_IN, and STOPO_IN as outgoing notification events.
  - ET, Q, and PT as outgoing data values.

Functionality

The behavior of E_FB_TimeOut is implemented by an internal instance of the event-based delay function block adapter::events::TimeOut::E_FB_DELAY.

The block works as follows:

1. When TimeOutSocket.START is received, the event is forwarded to the internal timer. The data value TimeOutSocket.DT is supplied as the delay/timeout duration.
2. The internal timer starts and counts the configured time.
3. If no STOP event arrives, the internal timer eventually generates an expiration event. This event is forwarded to TimeOutSocket.TimeOut.
4. When TimeOutSocket.STOP is received, it is forwarded to the internal timer and cancels the active timeout.
5. A cyclic query can be made through TimeTickSocket.REQ. The internal timer responds with CNF and provides the current values of Q, ET, and PT through TimeTickSocket.
6. Start and stop transitions of the internal timer are reported through TimeTickSocket.STARTO_IN and TimeTickSocket.STOPO_IN.

The internal wiring is summarized in the following table:

E_FB_TimeOut Interface	Internal E_FB_DELAY Connection
TimeOutSocket.START	DLY.START
TimeOutSocket.STOP	DLY.STOP
TimeOutSocket.DT	DLY.DT
DLY.EO	TimeOutSocket.TimeOut
TimeTickSocket.REQ	DLY.REQ
DLY.CNF	TimeTickSocket.CNF
DLY.STARTO	TimeTickSocket.STARTO_IN
DLY.STOPO	TimeTickSocket.STOPO_IN
DLY.ET	TimeTickSocket.ET
DLY.Q	TimeTickSocket.Q
DLY.PT	TimeTickSocket.PT

Technical Features

- Composite function block implementation.
- Contains a single internal function block instance: adapter::events::TimeOut::E_FB_DELAY.
- No plain event or data pins; the entire interface is adapter-based.

- Supports both event-driven timeout signaling and cyclic state polling.
- Uses standard IEC 61499 event and data connections for internal wiring.
- Provides `Q`, `ET`, and `PT` values for monitoring and diagnosis.
- Can be embedded in larger IEC 61499 applications where reusable timeout services are required.

State Overview

Although the block is a composite FB and does not define its own ECC state machine, the following observable states can be derived from the timer behavior:

State	Description
Idle	No timeout is active. The timer has not been started or has been stopped.
Running	A `START` event has been accepted and the timer is counting. `STARTO_IN` is emitted when this state is entered.
Expired	The configured timeout duration has elapsed. `TimeOut` is emitted and the timer state output `Q` reflects the timeout condition.
Stopped	A `STOP` event has cancelled the active timeout. `STOPO_IN` is emitted when this state is entered.

During all states, the current timer values can be queried through `TimeTickSocket.REQ` and `TimeTickSocket.CNF`.

Application Scenarios

E_FB_TimeOut is suitable for the following use cases:

- Watchdog supervision of external operations.
- Timeout handling in communication protocols.
- Event-driven state machines requiring a start/stop timeout service.
- Cyclic control applications where the timeout state must be checked every PLC or IEC 61499 execution cycle.
- Reusable service blocks in distributed IEC 61499 systems.

The combination of `ATimeOut` and `ATimeTick` allows both a simple event-based usage and a more detailed cyclic monitoring of timing values.

Comparison with Similar Blocks

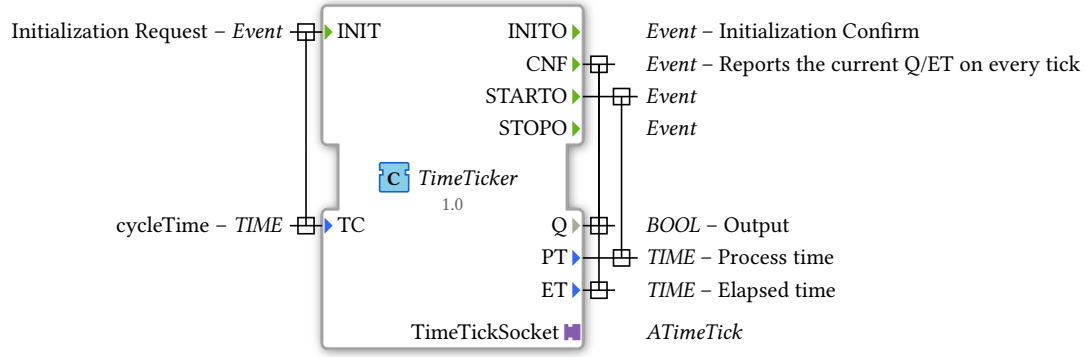
Block / Approach	Interface	Timeout Notification	Cyclic Query
`FB_TON`	Boolean input/output and data pins	`Q` changes, no timeout event	Usually not available
`E_DELAY`	Discrete event/data pins	Expiration event	Often not supported
`E_FB_TimeOut`	Adapter sockets `ATimeOut` and `ATimeTick`	`TimeOut` event	`REQ` / `CNF` with `Q`, `ET`, `PT`

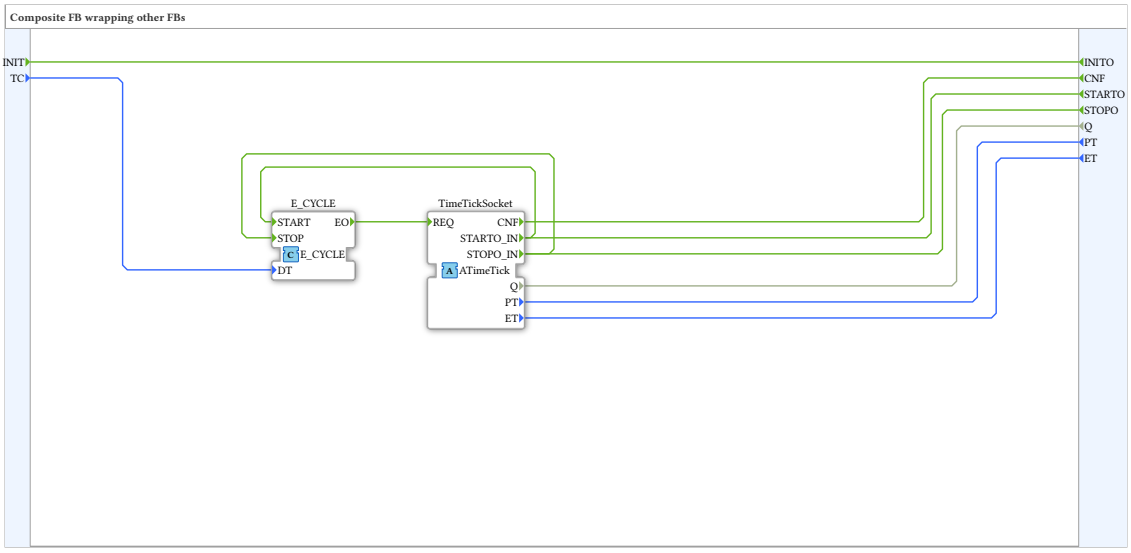
Compared with a simple TON function block, E_FB_TimeOut provides an event-oriented timeout service and additionally exposes a cyclic query interface through `TimeTickSocket`.

Conclusion

E_FB_TimeOut is a compact, adapter-based timeout function block for IEC 61499 applications. It separates timeout control from cyclic status observation, which makes it flexible and reusable. Because it is a composite FB, its behavior is fully determined by the internal `E_FB_DELAY` instance and the adapter connections. This design provides a clean interface for event-driven timeout handling as well as periodic monitoring in automation and control systems.

TimeTicker





• – ○ • – ○ • – ○ •

Introduction

`TimeTicker` is a composite function block that encapsulates an `iec61499::events::E_CYCLE` function block and exposes an adapter-based interface for periodic tick generation. It provides a configurable cycle time, start/stop handling, and event-based reporting of output values such as process time and elapsed time.

The block is intended to be used as a reusable time-tick service inside IEC 61499 systems where adapter-based communication is preferred over direct event/data wiring.

Interface Structure

Event Inputs

Event	Type	With	Description
`INIT`	`EInit`	`TC`	Initialization request, carries the cycle time.

Event Outputs

Event	Type	With	Description
`INITO`	`EInit`	-	Initialization confirmation.
`CNF`	`Event`	`Q`, `ET`	Reports the current `Q` and `ET` on every tick.
`STARTO`	`Event`	`PT`	Indicates that the ticker has been started.
`STOPO`	`Event`	-	Indicates that the ticker has been stopped.

Data Inputs

Data	Type	Initial Value	Description
TC	TIME	T#200ms	Cycle time of the ticker.

Data Outputs

Data	Type	Description
Q	BOOL	Current output state supplied through the adapter.
PT	TIME	Process time, associated with the start event.
ET	TIME	Elapsed time, reported with each confirmation.

Adapters

Adapter	Type	Direction	Description
TimeTickSocket	adapter::events::TimeOut::ATimeTick	Plug	Adapter connection used for tick requests, confirmations, start/stop commands, and time data exchange.

Internally, the adapter is used with the following logical endpoints:

Adapter Endpoint	Role in the Internal Network
REQ	Receives the periodic event from E_CYCLE.EO.
CNF	Triggers the external CNF event output.
STARTO_IN	Starts the internal E_CYCLE and triggers STARTO.
STOPO_IN	Stops the internal E_CYCLE and triggers STOPO.
Q, PT, ET	Provide the data values connected to the outputs.

Functionality

The TimeTicker composite FB contains a single internal E_CYCLE block. The cycle time TC is connected to the DT input of E_CYCLE, so the period of the generated ticks is determined by TC.

When a start command arrives through the adapter (STARTO_IN), the composite FB:

- raises the STARTO event output together with PT,
- sends a start event to the internal E_CYCLE.

While the internal cycle block is running, it generates EO events periodically with the configured interval. Each EO event is forwarded to the adapter as REQ, producing a tick on the adapter interface.

When the adapter returns a confirmation CNF, the composite FB raises its own CNF event output and publishes the associated data values Q and ET.

A stop command arriving through the adapter (STOPO_IN) is forwarded to the stop input of the internal E_CYCLE and also triggers the STOPO output.

An INIT event is passed directly through to INITO, providing an immediate initialization confirmation.

Technical Features

- Composite FB implementation using an internal iec61499::events::E_CYCLE.
- Configurable cycle time through the TC data input, defaulting to 200 ms.
- Adapter-based tick interface using adapter::events::TimeOut::ATimeTick.
- Standard event/data association:
  - INIT is associated with TC.
  - CNF is associated with Q and ET.
  - STARTO is associated with PT.
- No internal algorithms or state-machines are required; the behavior is entirely defined by the function block network.

State Overview

The TimeTicker does not contain an explicit ECC state machine, but it can be described in terms of logical operational states:

State	Description
Initialized	After INIT, the FB confirms initialization via INITO. No ticks are generated yet.
Stopped	The internal E_CYCLE is not running. No periodic REQ events are generated.
Running	After STARTO_IN, the internal E_CYCLE is started. Ticks are emitted periodically on the adapter.
Confirming	After each tick, the adapter returns CNF, causing the CNF output to fire with Q and ET.

Application Scenarios

- Periodic data polling in automation systems.
- Generation of cyclic time-tick events for adapter-based communication.
- Start/stop controlled time supervision with process time and elapsed time reporting.
- Integration into larger IEC 61499 applications that rely on adapters instead of direct wiring.
- Reusable timing component for service interfaces that require a periodic request/confirmation pattern.

Comparison with Similar Blocks

Block	Comparison
E_CYCLE	A low-level periodic event generator. TimeTicker wraps E_CYCLE and adds an adapter interface, start/stop events, and confirmation data.
TON / TOF	Classic timer blocks generate boolean timing outputs. TimeTicker generates periodic events and reports time values through an adapter.
Plain service FB	Many service FBs require manual event handling. TimeTicker encapsulates cycle control and adapter communication in one composite block.

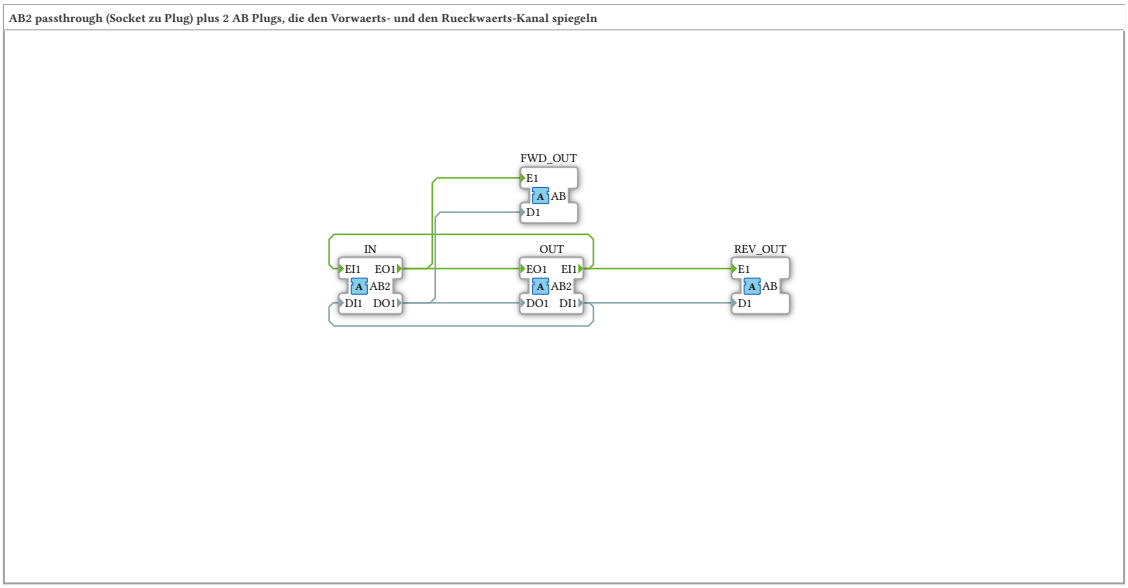
Conclusion

The TimeTicker FB is a practical composite function block for producing periodic tick events in an adapter-based IEC 61499 environment. By combining an internal E_CYCLE with an ATimeTick adapter, it provides a clean and reusable interface for start/stop control, cyclic requests, confirmations, and time-related data exchange. Its configurable cycle time and straightforward internal network make it suitable for many event-driven automation and monitoring scenarios.

bidirectional

- AE2_SPLIT_MERGE
- AE2_SPLIT_MERGE_FORWARD
- AE_AX_AX_SPLIT
- AE_AX_T_FF
- ASR_AX_AX_SPLIT
- ASR_AX_SR_2
- ASRT_AX_AX_SPLIT
- ASRT_AX_T_FF_SR_2

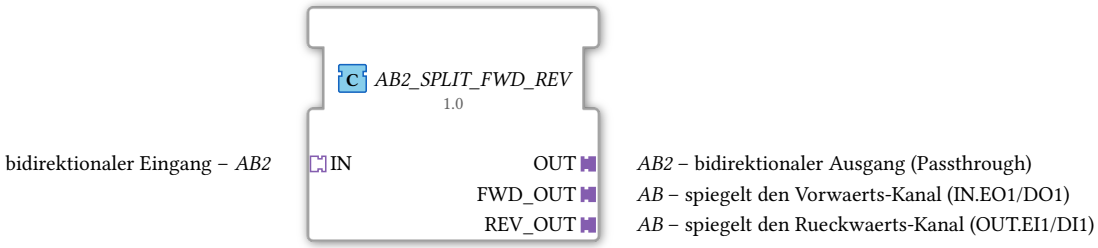
AB2_SPLIT_FWD_REV



• – ○ • – ○ • – ○ •

Introduction

`AB2_SPLIT_FWD_REV` is a composite function block that passes a bidirectional `AB2` adapter from socket `IN` unchanged through to plug `OUT` (passthrough) and additionally mirrors the forward and the backward channel onto two separate, unidirectional `AB` plugs. This allows both directions of an `AB2` connection to be tapped elsewhere without altering the connection itself.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **IN** (socket): Bidirectional adapter of type `adapter::types::bidirectional::AB2` (bidirectional input)
- **OUT** (plug): Bidirectional adapter of type `adapter::types::bidirectional::AB2` (bidirectional output, passthrough)
- **FWD_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AB`, mirrors the forward channel (`IN.EO1/DO1`)

- **REV_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AB`, mirrors the backward channel (`OUT.EI1`/`DI1`)

How it works

1. **Forward (indication)**: `IN.EO1`/`IN.DO1` is forwarded unchanged to `OUT.EO1`/`OUT.DO1` and additionally mirrored to `FWD_OUT.E1`/`FWD_OUT.D1`.
2. **Backward (request)**: `OUT.EI1`/`OUT.DI1` is forwarded unchanged to `IN.EI1`/`IN.DI1` and additionally mirrored to `REV_OUT.E1`/`REV_OUT.D1`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- Each passthrough event output feeds two targets (fan-out); the data runs in parallel with the events
- The two mirror plugs are unidirectional (`AB`) and therefore provide read access to the respective channel only

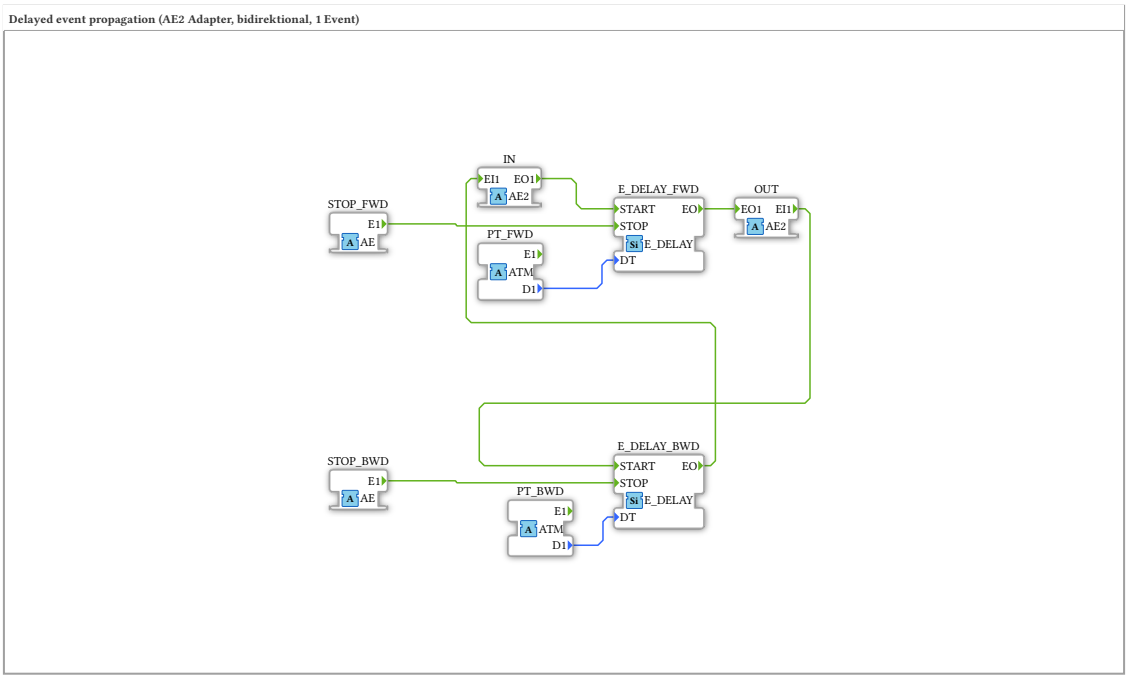
Application Scenarios

- Monitoring/diagnosing both directions of a bidirectional `AB2` connection
- Splitting an `AB2` connection into a separately evaluable forward and backward stream without interrupting the main connection

Summary

`AB2_SPLIT_FWD_REV` passes a bidirectional `AB2` connection through transparently and additionally provides the forward and backward channels as separate `AB` adapters.

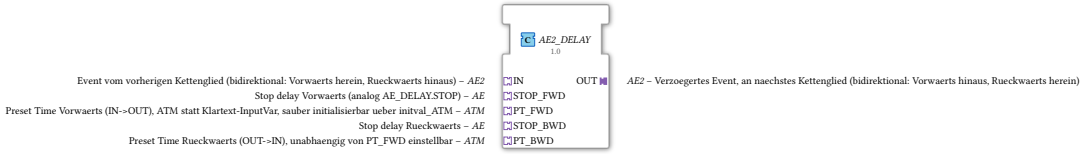
AE2_DELAY



• — ○ • — ○ • — ○ •

Introduction

The `AE2_DELAY` function block delays events of a bidirectional `AE2` adapter. Because AE2 carries events in both directions (`EO1` forward, `EI1` backward), the block has two independent delay paths: one for the forward direction (`IN` ⟶ `OUT`) using the time `PT_FWD` and one for the backward direction (`OUT` ⟶ `IN`) using the time `PT_BWD`. Each direction can be aborted through its own `STOP` socket.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **IN** (socket): event from the previous chain element (forward in, backward out), type `adapter::types::bidirectional::AE2`
- **OUT** (plug): delayed event to the next chain element (forward out, backward in), type `adapter::types::bidirectional::AE2`
- **STOP_FWD** (socket): aborts the running forward delay, type `adapter::types::unidirectional::AE`
- **PT_FWD** (socket): preset time for the forward direction, type `adapter::types::unidirectional::ATM`
- **STOP_BWD** (socket): aborts the running backward delay, type `adapter::types::unidirectional::AE`
- **PT_BWD** (socket): preset time for the backward direction, adjustable independently of `PT_FWD`, type `adapter::types::unidirectional::ATM`

Functionality

The internal network consists of two instances of `E_DELAY`:

- **Forward** (`E_DELAY_FWD`): `IN.EO1` ⟶ `E_DELAY_FWD.START`, `STOP_FWD.E1` ⟶ `E_DELAY_FWD.STOP`, `PT_FWD.D1` ⟶ `E_DELAY_FWD.DT`. When the time has elapsed, `E_DELAY_FWD.EO` is output on `OUT.EO1`.
- **Backward** (`E_DELAY_BWD`): `OUT.EI1` ⟶ `E_DELAY_BWD.START`, `STOP_BWD.E1` ⟶ `E_DELAY_BWD.STOP`, `PT_BWD.D1` ⟶ `E_DELAY_BWD.DT`. When the time has elapsed, `E_DELAY_BWD.EO` is output on `IN.EI1`.

The data of the adapter is not processed; only the events `EO1`/`EI1` are passed on.

Technical Details

- Two independent delay paths (one `E_DELAY` instance per direction), each with its own time and its own `STOP`
- The times are supplied through `ATM` sockets instead of plain input variables and can therefore be initialised cleanly via `initval_ATM` (with `INIT`/`INITO`)
- The time sockets are named `PT` (preset time) rather than `DT`, because `DT` is a reserved keyword (`DATE_AND_TIME`) in IEC 61131-3
- The block itself is stateless; the state (running timer) resides in the two `E_DELAY` instances

Application Scenarios

- Time-shifted forwarding of events in an AE2 chain, e.g. staggered switch-on and switch-off with different times
- Decoupling of request and feedback in bidirectional event chains

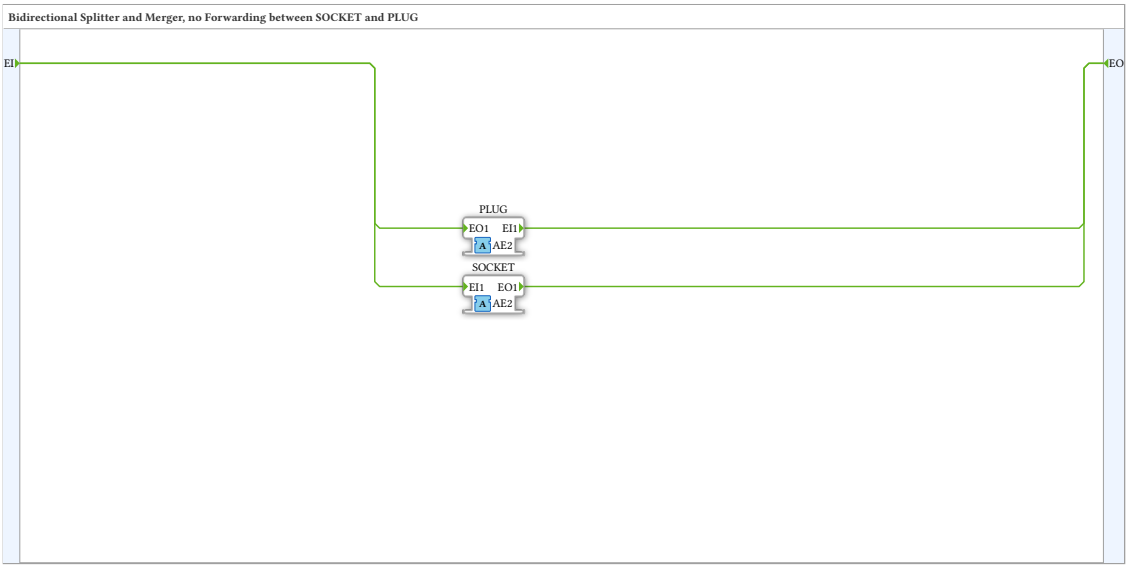
⚖️ Comparison with Similar Blocks

For chains with separately delayed set and reset branches there is `ASR2_DELAY`, whose variant with preset times is `ASR2_DELAY_SUB`. `AE2_DELAY`, in contrast, delays exactly one event per direction.

Conclusion

AE2_DELAY is a pure wiring block built from two E_DELAY instances. It delays events of an AE2 adapter in both directions, each with its own time adjustable through an ATM socket, and each direction can be stopped separately.

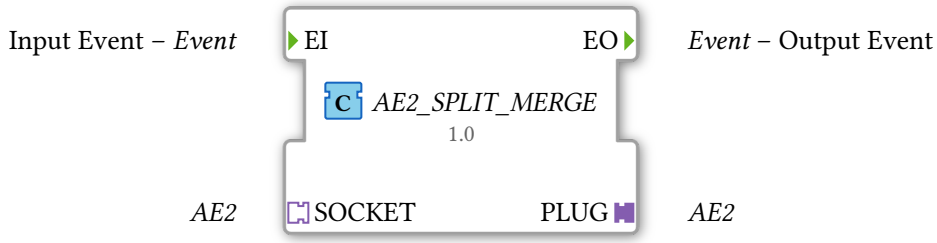
AE2_SPLIT_MERGE



• – ○ • – ○ • – ○ •

Introduction

The AE2_SPLIT_MERGE is a bidirectional splitter and merge function block that can process events in both directions. The block enables the distribution and merging of events via adapter interfaces, without direct routing between the SOCKET and PLUG.



Interface Structure

Event Inputs

- **EI**: Main event input for incoming events

Event Outputs

- **EO**: Main event output for outgoing events

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

- **PLUG**: Adapter interface of type “adapter::types::bidirectional::AE2”
- **SOCKET**: Adapter interface of type “adapter::types::bidirectional::AE2”

Functionality

The function block operates according to the following principles:

1. **Event Distribution**: Events at input EI are simultaneously forwarded to both adapter interfaces (PLUG and SOCKET).
2. **Event Merging**: Events from both adapter interfaces are combined. (PLUG.EI1 and SOCKET.EO1) are combined at output EO.
3. **Bidirectional Isolation**: There is no direct communication between the PLUG and SOCKET adapter interfaces.

Technical Features

- Pure event processing without data transfer
- Symmetrical handling of both adapter directions
- Isolated signal paths between the adapter interfaces
- Real-time event distribution

State Overview

The functional block has no internal state and operates stateless. All incoming events are processed and forwarded immediately.

Application Scenarios

- Distribution of control events to multiple subsystems
- Merging of status events from various sources
- Bidirectional communication in distributed control systems
- Event-based architectures with adapter patterns

⚖️ Comparison with similar building blocks

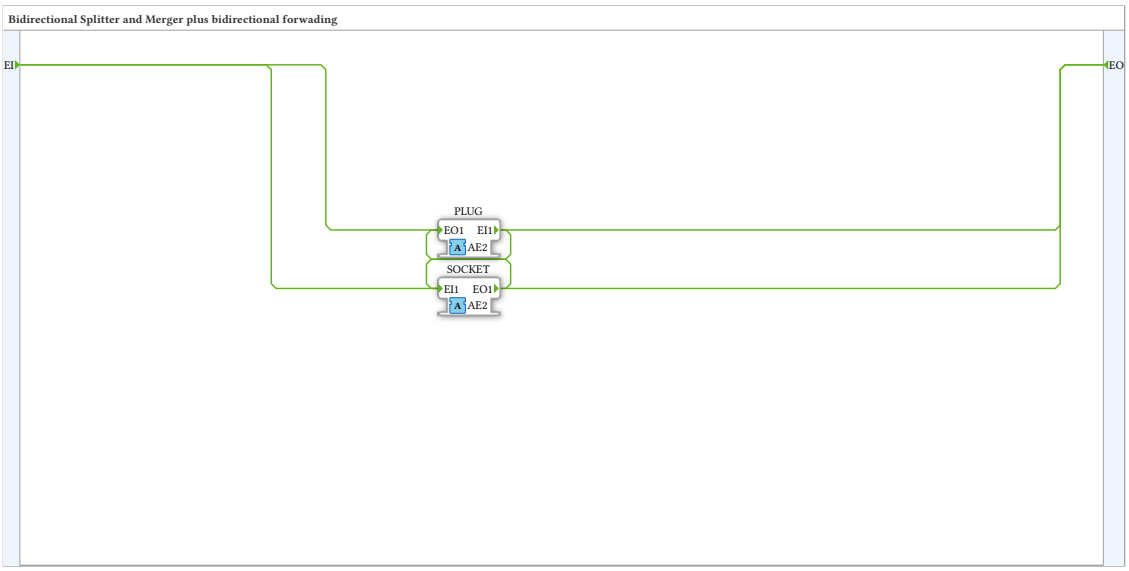
Compared to simple splitter or merger blocks, AE2_SPLIT_MERGE offers:

- Bidirectional functionality in a single block
- Adapter-based interfaces for standardized communication
- Isolated signal paths between different adapter directions

Conclusion

AE2_SPLIT_MERGE is a specialized function block for complex event distribution and merging scenarios in bidirectional communication architectures. Its adapter-based structure enables flexible and standardized integration into larger control systems.

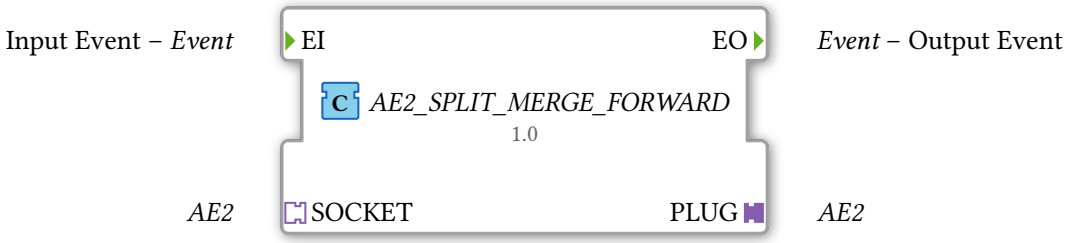
AE2_SPLIT_MERGE_FORWARD



• – ◦ • – ◦ • – ◦ •

Introduction

The AE2_SPLIT_MERGE_FORWARD function block is a bidirectional splitter and merger with bidirectional forwarding. It serves as a universal distributor and collector for events in 4diac systems and enables the flexible distribution of events across various interfaces.



Interface Structure

Event Inputs

- **EI**: Input Event

Event Outputs

- **EO**: Output Event

Adapters

- **PLUG**: Bidirectional AE2 Adapter
- **SOCKET**: Bidirectional AE2 Adapter

Functionality

The function block operates according to the following principles:

1. **Event Distribution**: Every event received at the EI input is simultaneously forwarded to both adapters (PLUG and SOCKET).
2. **Event Merging**: Events received via either adapter (PLUG or SOCKET) are output at the EO output.
3. **Bidirectional Forwarding**: Events are forwarded bidirectionally between the two adapters, allowing PLUG and SOCKET events to be exchanged.

Technical Features

- Uses the bidirectional AE2 adapter type
- Enables parallel event distribution
- Provides full bidirectional communication between all interfaces
- No data connections – purely event-based

State Overview

The function block has no internal states and operates purely combinatorially. All event processing occurs immediately upon the arrival of an event.

Application Scenarios

- Distributor for control events in complex automation networks
- Interface adapters between different system components
- Event routing in modular control architectures
- Bidirectional communication bridges between function blocks

⚖️ Comparison with similar modules

Compared to simple splitter blocks, AE2_SPLIT_MERGE_FORWARD additionally offers:

- Bidirectional functionality
- Adapter-based interfaces
- Event merging
- Direct communication between adapters

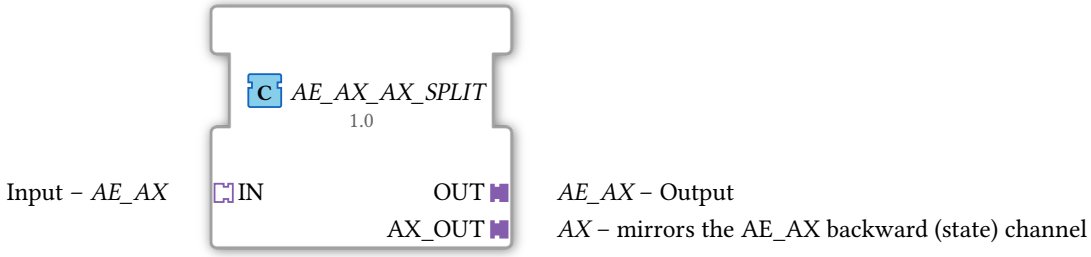
Conclusion

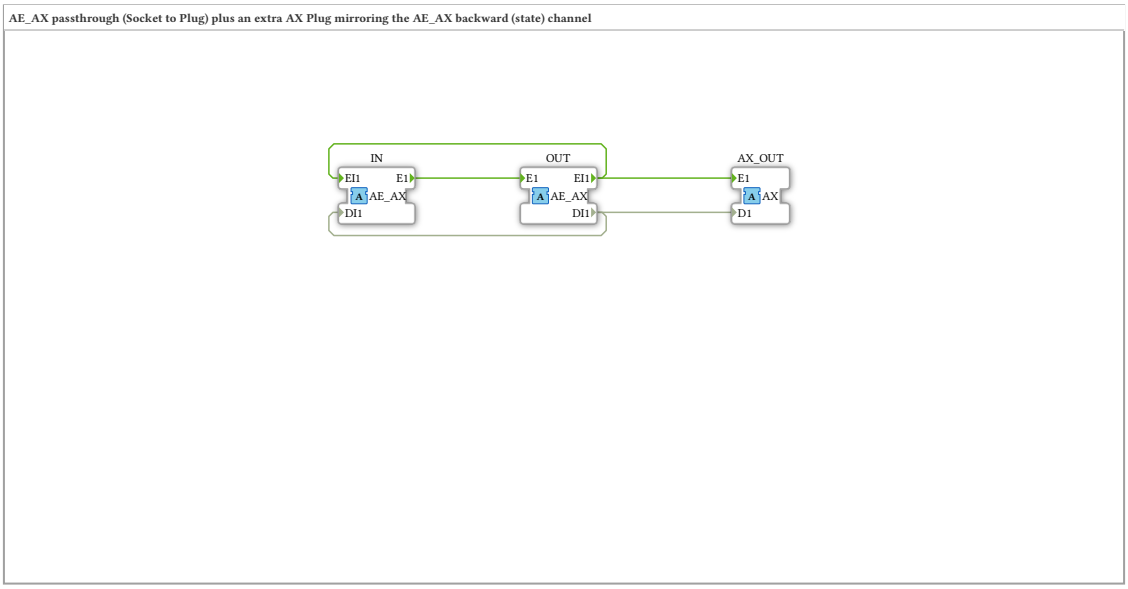
The AE2_SPLIT_MERGE_FORWARD is a versatile function block for complex event distribution and collection tasks in 4diac systems. Its bidirectional nature and adapter interfaces make it particularly suitable for modular and expandable automation solutions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AE_AX_AX_SPLIT





• – ○ • – ○ • – ○ •

Introduction

The AE_AX_AX_SPLIT is a composite function block that passes an incoming AE_AX event unchanged from its socket `IN` to the plug `OUT`, and additionally mirrors the state (event + bool) reported on the backward channel of `OUT` to the outside via a third, unidirectional `AX_OUT` plug. This allows the state of a downstream chain to be tapped in parallel at a second point without interrupting the actual AE_AX connection.

Interface Structure

Event Inputs

No direct event inputs available – events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

- **IN**: Bidirectional adapter socket of type `adapter::types::bidirectional::AE_AX` (input)
- **OUT**: Bidirectional adapter plug of type `adapter::types::bidirectional::AE_AX` (output)
- **AX_OUT**: Unidirectional adapter plug of type `adapter::types::unidirectional::AX`, mirrors the AE_AX backward channel (state)

Functionality

1. Every event arriving at `IN.E1` is forwarded unchanged to `OUT.E1` – the forward direction is passed through 1:1.
2. The backward-channel event `OUT.EI1` (together with its associated datum `OUT.DI1`) is relayed back to `IN.EI1`/`IN.DI1` and, at the same time, forwarded to `AX_OUT.E1`/`AX_OUT.D1`.
3. As a result, the state reported via `OUT` is visible at two points: at the original socket `IN` (as a pure passthrough would also provide) and additionally, in isolation, at the `AX_OUT` plug.

Technical Features

- Pure event/data connections (`FBNetwork`), no own logic or state management
- The forward direction (socket ⟶ plug) is a simple 1:1 passthrough
- The backward channel is duplicated rather than split: both destinations (`IN` and `AX_OUT`) receive the same information

State Overview

The function block has no internal state and operates statelessly. Every incoming event is immediately forwarded or mirrored.

Application Scenarios

- Additional, isolated tapping of the AE_AX backward channel (e.g. for a separate display or logging) without altering the actual connection between `IN` and `OUT`
- Building diagnostic/monitoring paths in AE_AX-based control networks

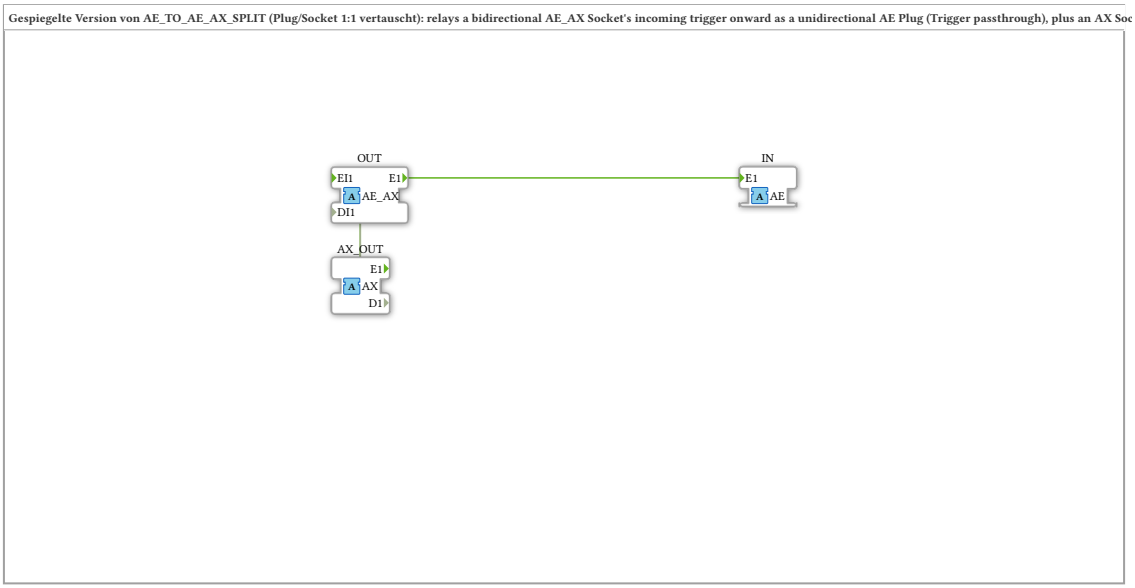
⚖️ Comparison with similar building blocks

Compared to AE2_SPLIT_MERGE, which distributes/merges the forward and backward directions between two separate, isolated adapters, AE_AX_AX_SPLIT is a pure passthrough between `IN` and `OUT` with an additional mirror output for the backward channel – the ASR_AX and ASRT_AX variants ASR_AX_AX_SPLIT and ASRT_AX_AX_SPLIT follow the same pattern with two and three forward events, respectively.

Conclusion

AE_AX_AX_SPLIT makes the AX backward channel of an AE_AX adapter pair additionally available at a second point in the network, without interrupting or altering the actual signal chain.

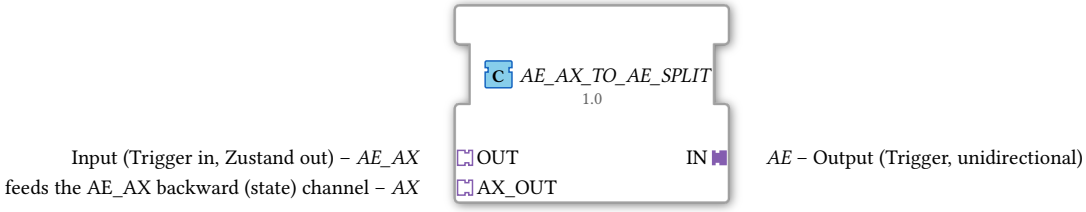
AE_AX_TO_AE_SPLIT



• – ○ • – ○ • – ○ •

Introduction

`AE_AX_TO_AE_SPLIT` is the mirrored version of `AE_TO_AE_AX_SPLIT` (plug and socket swapped 1:1). The block forwards the incoming trigger of a bidirectional `AE_AX` socket onward as a unidirectional `AE` plug (trigger pass-through), and an additional `AX` socket injects an externally supplied value into the backward (state) channel of the `AE_AX` socket.



Interface Structure

Event / Data Inputs and Outputs

No own event or data connections; all events and data travel through the adapters.

Adapters

Direction	Name	Adapter type	Comment
Plug (output)	`IN`	`adapter::types::unidirectional::AE`	Output (trigger, unidirectional).
Socket (input)	`OUT`	`adapter::types::bidirectional::AE_AX`	Input (trigger in, state out).
Socket (input)	`AX_OUT`	`adapter::types::unidirectional::AX`	Feeds the backward (state) channel of the `AE_AX`.

Functionality

The block is a composite FB without any internal function blocks; it consists only of adapter connections:

- **Event connections:** `OUT.E1` ⟶ `IN.E1` (the event arriving at the `AE_AX` socket is output unchanged at the `AE` plug); `AX_OUT.E1` ⟶ `OUT.EI1` (the event received at the `AX` socket is sent out as the backward-channel event via the `AE_AX` socket).
- **Data connection:** `AX_OUT.D1` ⟶ `OUT.DI1` (the value received at the `AX` socket is sent out as the backward-channel datum `DI1` of the `AE_AX` socket).

Technical Details

- Pure wiring, no logic and no delay.
- The names `IN`/`OUT` are inherited from the source `AE_TO_AE_AX_SPLIT` and are therefore swapped: `OUT` is a socket here, `IN` a plug.
- The trigger direction stays unidirectional (`AE`); the backward channel is fed entirely by the `AX` socket.

Application Scenarios

- Splitting a bidirectional `AE_AX` connection into a pure trigger path (`AE`) and a separately fed state path (`AX`).

Summary

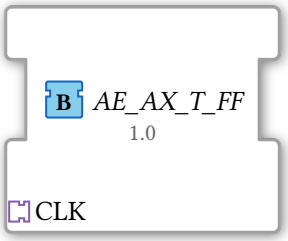
`AE_AX_TO_AE_SPLIT` splits an `AE_AX` socket into a forwarded `AE` trigger and a backward channel fed via `AX`.

🌐 Related topic pages on ms-muc-docs.de

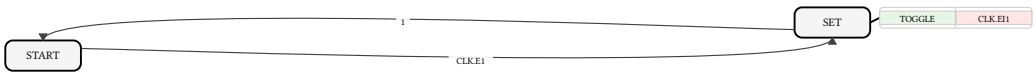
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AE_AX_T_FF

Clock (in), Zustand (out) – *AE_AX*



ECC: AE_AX_T_FF



• – ○ • – ○ • – ○ •

Introduction

The AE_AX_T_FF is an event-driven toggle flip-flop with a single bidirectional `AE_AX` socket: the clock event arrives on the same adapter that also reports the new state back – unlike AX_T_FF, which uses a separate CLK event input and its own AX plug for the state.

Interface Structure

Event Inputs

No direct event inputs available – the clock event arrives via the adapter socket `CLK`

Event Outputs

No direct event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

- **CLK**: Bidirectional adapter socket of type `adapter::types::bidirectional::AE_AX` – clock (input), state (output) via the same adapter

Operation

The AE_AX_T_FF has two operating states:

- **START**: Initial state
- **SET**: State after toggling

Every event arriving at `CLK.E1` moves the function block from START to SET, firing the algorithm `TOGGLE`, which executes `CLK.DI1 := NOT CLK.DI1` and thereby reports the inverted state back over the same adapter via `CLK.EI1`. From the SET state the block unconditionally (condition `1`) returns immediately to START, so that every further `CLK.E1` event triggers the same TOGGLE transition again.

Technical Features

- Bidirectional communication over a single adapter socket instead of a separate event input plus AX plug
- The ECC state SET is only a pass-through state for firing the output event; the actual toggle semantics live in the algorithm assignment `CLK.DI1 := NOT CLK.DI1`
- A drop-in replacement for AX_T_FF in fully adapter-based (AE_AX) applications

State Overview

```
START (initial state)
    │
    └── CLK.E1 ─→ SET (CLK.DI1 := NOT CLK.DI1, reports state via CLK.EI1)
                      │
                      └── (unconditional) ─→ START
```

Application Scenarios

- Toggling a state (e.g. on/off) over a single bidirectional adapter connection, without wiring a separate state output
- Replacement for AX_T_FF in networks that consistently use AE_AX adapters

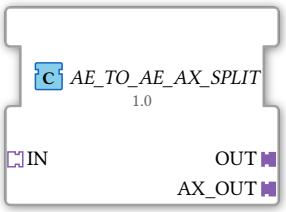
⚖️ Comparison with similar components

Compared to AX_T_FF, which uses a separate `CLK` event input and its own AX plug for the state, AE_AX_T_FF bundles both directions into a single AE_AX socket.

Conclusion

AE_AX_T_FF carries the proven toggle flip-flop pattern over into the bidirectional adapter world: clock input and state feedback share the same AE_AX socket, simplifying wiring in adapter-based networks.

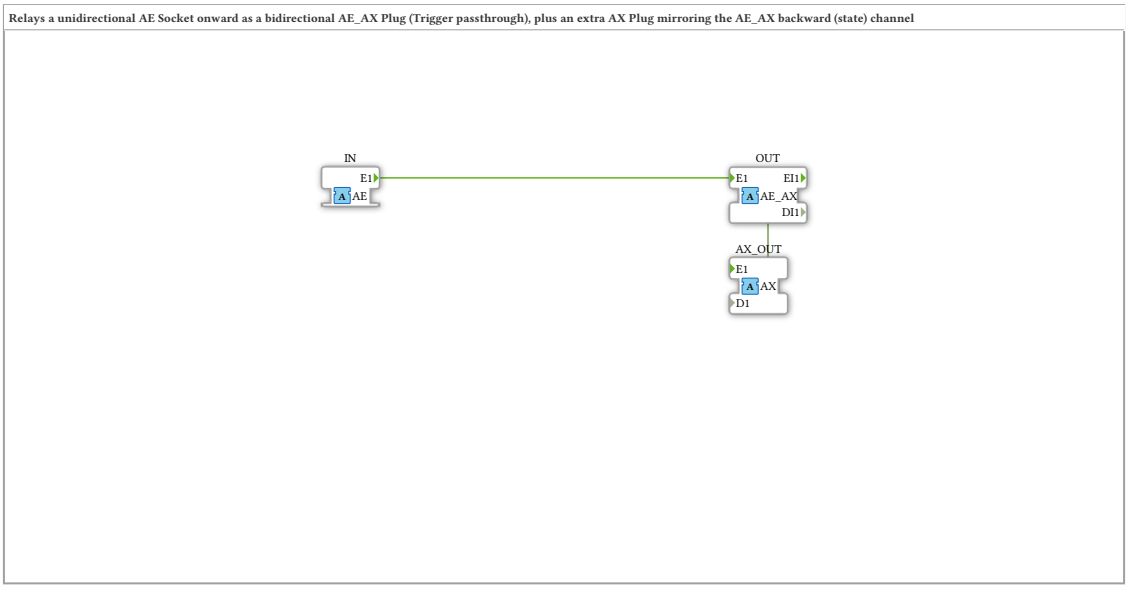
AE_TO_AE_AX_SPLIT



Input (Trigger, unidirectional) – *AE*

AE_AX – Output (Trigger out, Zustand in)

AX – mirrors the AE_AX backward (state) channel



• – ○ • – ○ • – ○ •

Introduction

AE_TO_AE_AX_SPLIT is a composite function block that converts an incoming **unidirectional** AE event at its socket `IN` into a **bidirectional** AE_AX signal at plug `OUT`, and additionally mirrors the state (event + bool) reported on `OUT`'s backward channel out through a third, unidirectional `AX_OUT` plug. Unlike the pure passthrough `AE_AX_AX_SPLIT`, `IN` has no backward channel of its own here - so the backward state cannot be relayed back to `IN` and is exposed exclusively through `AX_OUT`.

Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **IN**: Unidirectional adapter socket of type `adapter::types::unidirectional::AE` (input, no backward channel)
- **OUT**: Bidirectional adapter plug of type `adapter::types::bidirectional::AE_AX` (output, with backward channel)
- **AX_OUT**: Unidirectional adapter plug of type `adapter::types::unidirectional::AX`, mirrors the AE_AX backward channel (state) to the outside

How it works

1. Every event arriving at `IN.E1` is forwarded to `OUT.E1` unchanged.
2. The backward-channel event `OUT.EI1` (with its associated data `OUT.DI1`), sent by the downstream counterpart via `OUT`, is forwarded exclusively to `AX_OUT.E1`/`AX_OUT.D1` - relaying it back to `IN` is technically impossible, since `IN` is a unidirectional AE adapter with no backward channel.
3. `AX_OUT` is therefore the only place where the state reported by `OUT` becomes visible.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic or state management of its own
- The forward direction (socket ⟶ plug) is a simple 1:1 passthrough
- Converts a unidirectional adapter (`IN`) into a bidirectional one (`OUT`) without needing a backward channel on the input itself
- The backward channel is not duplicated (as in AE_AX_AX_SPLIT), but exposed solely via `AX_OUT`

State Overview

The function block has no internal state and works statelessly. Every incoming event is forwarded/mirrored immediately.

Application Scenarios

- Connecting an existing, purely unidirectional AE signal source to a chain that expects a bidirectional AE_AX adapter
- Exposing the state of the downstream AE_AX counterpart via `AX_OUT`, even though the original source (`IN`) itself supports no backward channel

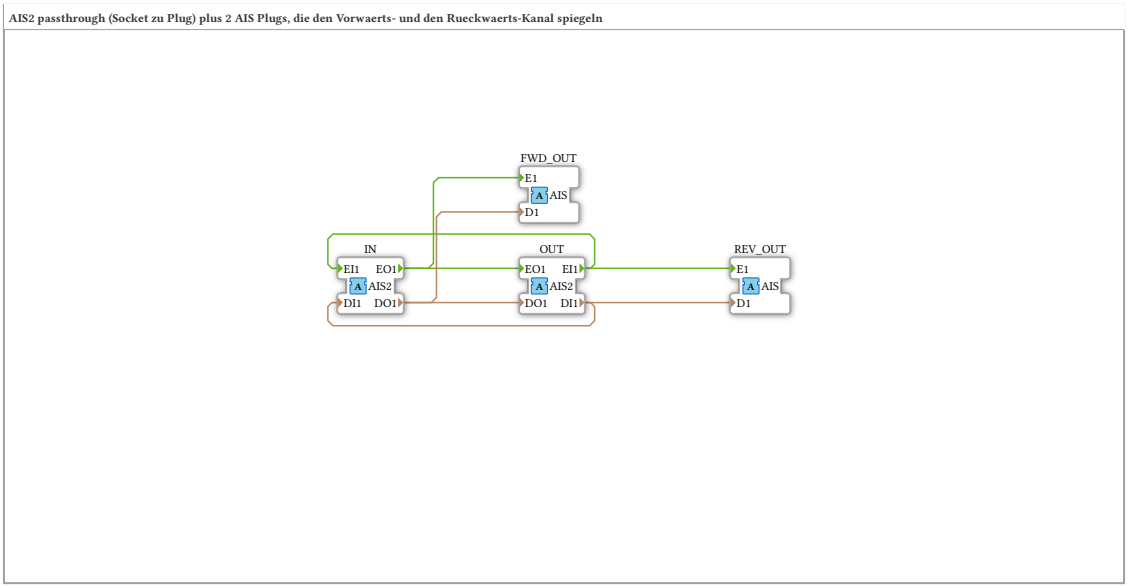
⚖️ Comparison with Similar Blocks

Unlike `AE_AX_AX_SPLIT`, which mediates between two already-bidirectional adapters and also mirrors the backward channel back to `IN`, AE_TO_AE_AX_SPLIT converts a unidirectional input into a bidirectional output - the backward channel can therefore only be observed via `AX_OUT`. The set/reset and set/reset/toggle variants `ASR_TO_ASR_AX_SPLIT` and `ASRT_TO_ASRT_AX_SPLIT` follow the same pattern with two or three forward events instead of a single one.

Summary

AE_TO_AE_AX_SPLIT allows a purely unidirectional AE signal source to connect to a bidirectional AE_AX chain and makes its backward channel available in isolation via `AX_OUT`, without requiring the original input itself to support a backward channel.

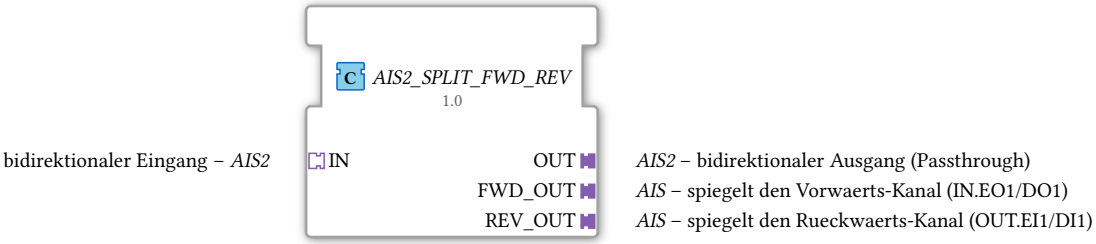
AIS2_SPLIT_FWD_REV



• – ○ • – ○ • – ○ •

Introduction

`AIS2_SPLIT_FWD_REV` is a composite function block that passes a bidirectional `AIS2` adapter from socket `IN` unchanged through to plug `OUT` (passthrough) and additionally mirrors the forward and the backward channel onto two separate, unidirectional `AIS` plugs. This allows both directions of an `AIS2` connection to be tapped elsewhere without altering the connection itself.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **IN** (socket): Bidirectional adapter of type `adapter::types::bidirectional::AIS2` (bidirectional input)
- **OUT** (plug): Bidirectional adapter of type `adapter::types::bidirectional::AIS2` (bidirectional output, passthrough)
- **FWD_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AIS`, mirrors the forward channel (`IN.EO1`/`DO1`)
- **REV_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AIS`, mirrors the backward channel (`OUT.EI1`/`DI1`)

How it works

1. **Forward (indication)**: `IN.EO1`/`IN.DO1` is forwarded unchanged to `OUT.EO1`/`OUT.DO1` and additionally mirrored to `FWD_OUT.E1`/`FWD_OUT.D1`.
2. **Backward (request)**: `OUT.EI1`/`OUT.DI1` is forwarded unchanged to `IN.EI1`/`IN.DI1` and additionally mirrored to `REV_OUT.E1`/`REV_OUT.D1`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- Each passthrough event output feeds two targets (fan-out); the data runs in parallel with the events
- The two mirror plugs are unidirectional (`AIS`) and therefore provide read access to the respective channel only

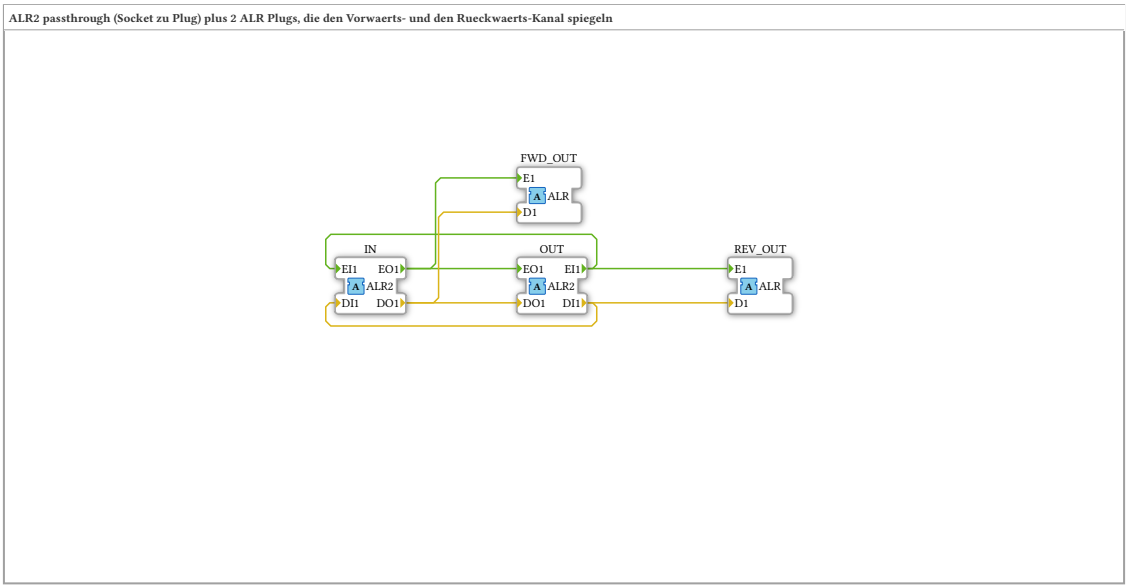
Application Scenarios

- Monitoring/diagnosing both directions of a bidirectional `AIS2` connection
- Splitting an `AIS2` connection into a separately evaluable forward and backward stream without interrupting the main connection

Summary

`AIS2_SPLIT_FWD_REV` passes a bidirectional `AIS2` connection through transparently and additionally provides the forward and backward channels as separate `AIS` adapters.

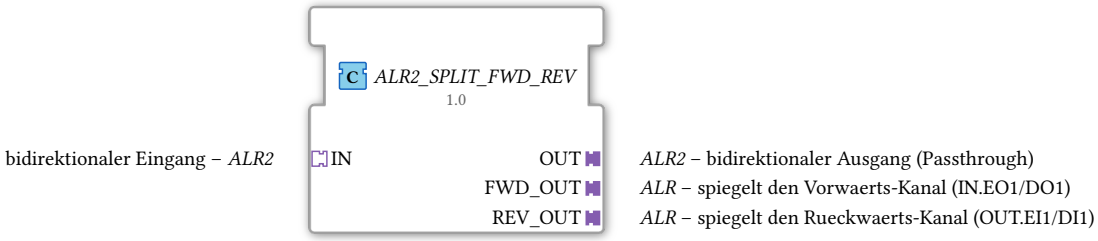
ALR2_SPLIT_FWD_REV



• – ○ • – ○ • – ○ •

Introduction

`ALR2_SPLIT_FWD_REV` is a composite function block that passes a bidirectional `ALR2` adapter from socket `IN` unchanged through to plug `OUT` (passthrough) and additionally mirrors the forward and the backward channel onto two separate, unidirectional `ALR` plugs. This allows both directions of an `ALR2` connection to be tapped elsewhere without altering the connection itself.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **IN** (socket): Bidirectional adapter of type `adapter::types::bidirectional::ALR2` (bidirectional input)
- **OUT** (plug): Bidirectional adapter of type `adapter::types::bidirectional::ALR2` (bidirectional output, passthrough)
- **FWD_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::ALR`, mirrors the forward channel (`IN.EO1/DO1`)

- **REV_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::ALR`, mirrors the backward channel (`OUT.EI1/DI1`)

How it works

1. **Forward (indication)**: `IN.EO1/IN.DO1` is forwarded unchanged to `OUT.EO1/OUT.DO1` and additionally mirrored to `FWD_OUT.E1/FWD_OUT.D1`.
2. **Backward (request)**: `OUT.EI1/OUT.DI1` is forwarded unchanged to `IN.EI1/IN.DI1` and additionally mirrored to `REV_OUT.E1/REV_OUT.D1`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- Each passthrough event output feeds two targets (fan-out); the data runs in parallel with the events
- The two mirror plugs are unidirectional (`ALR`) and therefore provide read access to the respective channel only

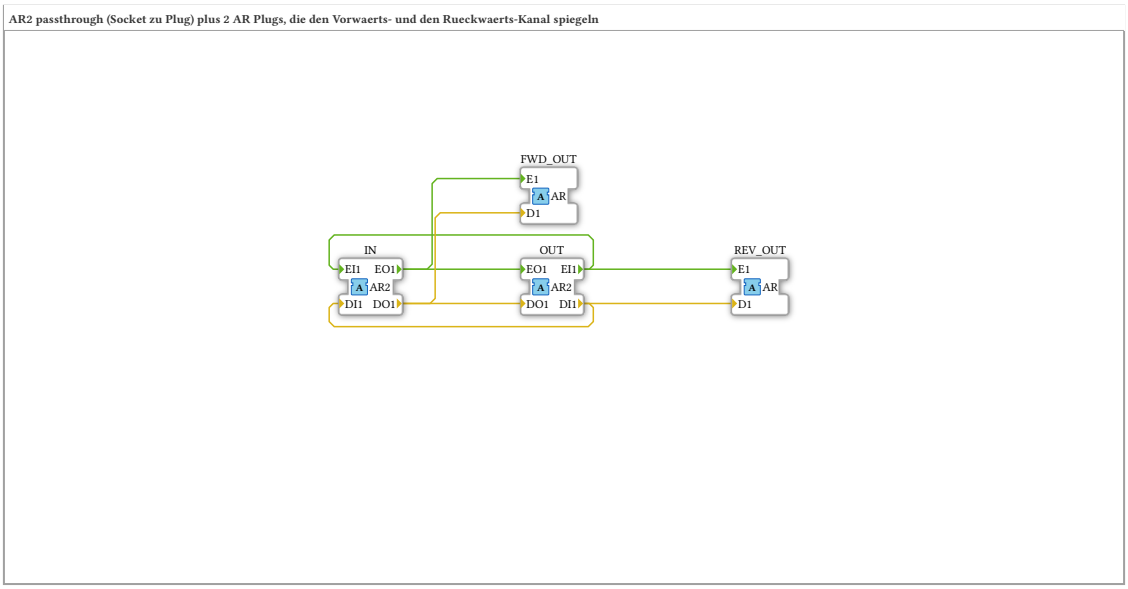
Application Scenarios

- Monitoring/diagnosing both directions of a bidirectional `ALR2` connection
- Splitting an `ALR2` connection into a separately evaluable forward and backward stream without interrupting the main connection

Summary

`ALR2_SPLIT_FWD_REV` passes a bidirectional `ALR2` connection through transparently and additionally provides the forward and backward channels as separate `ALR` adapters.

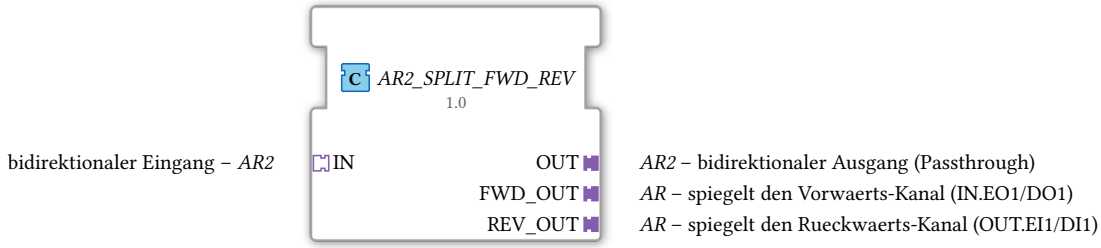
AR2_SPLIT_FWD_REV



• – ○ • – ○ • – ○ •

Introduction

The `AR2_SPLIT_FWD_REV` function block passes a bidirectional `AR2` adapter unchanged from `IN` (socket) to `OUT` (plug) and additionally mirrors both channels onto one unidirectional `AR` plug each: the forward channel (indication, `IN.EO1/DO1`) onto `FWD_OUT` and the backward channel (request, `OUT.EI1/DI1`) onto `REV_OUT`. This allows both directions of a bidirectional AR2 connection to be tapped separately as plain AR signals.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **IN** (socket): bidirectional input, type `adapter::types::bidirectional::AR2`
- **OUT** (plug): bidirectional output (passthrough), type `adapter::types::bidirectional::AR2`
- **FWD_OUT** (plug): mirrors the forward channel (`IN.EO1`/`DO1`), type `adapter::types::unidirectional::AR`
- **REV_OUT** (plug): mirrors the backward channel (`OUT.EI1`/`DI1`), type `adapter::types::unidirectional::AR`

Functionality

The block consists of wiring only:

- **Forward** (indication): `IN.EO1` ⟶ `OUT.EO1` and additionally `IN.EO1` ⟶ `FWD_OUT.E1`; `IN.DO1` ⟶ `OUT.DO1` and additionally `IN.DO1` ⟶ `FWD_OUT.D1`.
- **Backward** (request): `OUT.EI1` ⟶ `IN.EI1` and additionally `OUT.EI1` ⟶ `REV_OUT.E1`; `OUT.DI1` ⟶ `IN.DI1` and additionally `OUT.DI1` ⟶ `REV_OUT.D1`.

Both directions therefore pass through unchanged, and each one is additionally output as a plain unidirectional AR signal.

Technical Details

- Pure wiring without logic or state
- The passthrough `IN` ⟶ `OUT` is fully preserved; `FWD_OUT` and `REV_OUT` are pure observation taps
- Each source is fanned out to two destinations (event and data); the behaviour corresponds to a split onto both branches

Application Scenarios

- Monitoring the forward and the backward direction of a AR2 connection (e.g. for display, logging or diagnostics) without altering the connection itself
- Separate further processing of indication and request as plain AR signals

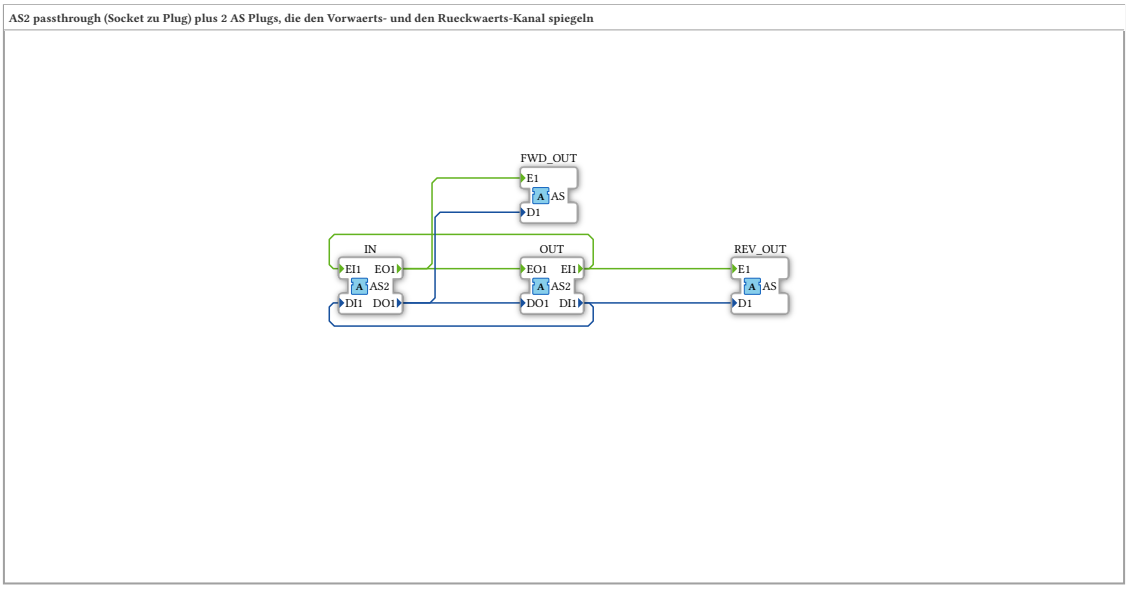
⚖️ Comparison with Similar Blocks

The same structure exists for further bidirectional adapters, e.g. `AR2_SPLIT_FWD_REV`, `AS2_SPLIT_FWD_REV`, `AX2_SPLIT_FWD_REV`, `AIS2_SPLIT_FWD_REV` and `ALR2_SPLIT_FWD_REV`; they differ only in the data type of the adapter.

Conclusion

AR2_SPLIT_FWD_REV is a stateless passthrough for AR2 that additionally mirrors the forward and backward channels onto two unidirectional AR plugs.

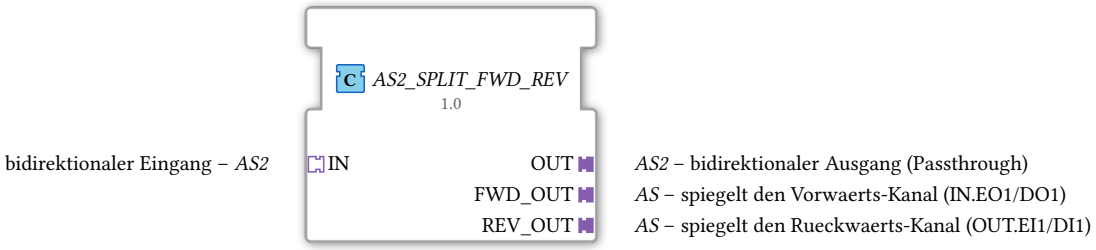
AS2_SPLIT_FWD_REV



• – ◦ • – ◦ • – ◦ •

Introduction

The AS2_SPLIT_FWD_REV function block passes a bidirectional AS2 adapter unchanged from IN (socket) to OUT (plug) and additionally mirrors both channels onto one unidirectional AS plug each: the forward channel (indication, IN.EO1/DO1) onto FWD_OUT and the backward channel (request, OUT.EI1/DI1) onto REV_OUT. This allows both directions of a bidirectional AS2 connection to be tapped separately as plain AS signals.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **IN** (socket): bidirectional input, type adapter::types::bidirectional::AS2

- **OUT** (plug): bidirectional output (passthrough), type `adapter::types::bidirectional::AS2`
- **FWD_OUT** (plug): mirrors the forward channel (`IN.EO1`/`DO1`), type `adapter::types::unidirectional::AS`
- **REV_OUT** (plug): mirrors the backward channel (`OUT.EI1`/`DI1`), type `adapter::types::unidirectional::AS`

Functionality

The block consists of wiring only:

- **Forward** (indication): `IN.EO1` ⟶ `OUT.EO1` and additionally `IN.EO1` ⟶ `FWD_OUT.E1`; `IN.DO1` ⟶ `OUT.DO1` and additionally `IN.DO1` ⟶ `FWD_OUT.D1`.
- **Backward** (request): `OUT.EI1` ⟶ `IN.EI1` and additionally `OUT.EI1` ⟶ `REV_OUT.E1`; `OUT.DI1` ⟶ `IN.DI1` and additionally `OUT.DI1` ⟶ `REV_OUT.D1`.

Both directions therefore pass through unchanged, and each one is additionally output as a plain unidirectional AS signal.

Technical Details

- Pure wiring without logic or state
- The passthrough `IN` ⟶ `OUT` is fully preserved; `FWD_OUT` and `REV_OUT` are pure observation taps
- Each source is fanned out to two destinations (event and data); the behaviour corresponds to a split onto both branches

Application Scenarios

- Monitoring the forward and the backward direction of a AS2 connection (e.g. for display, logging or diagnostics) without altering the connection itself
- Separate further processing of indication and request as plain AS signals

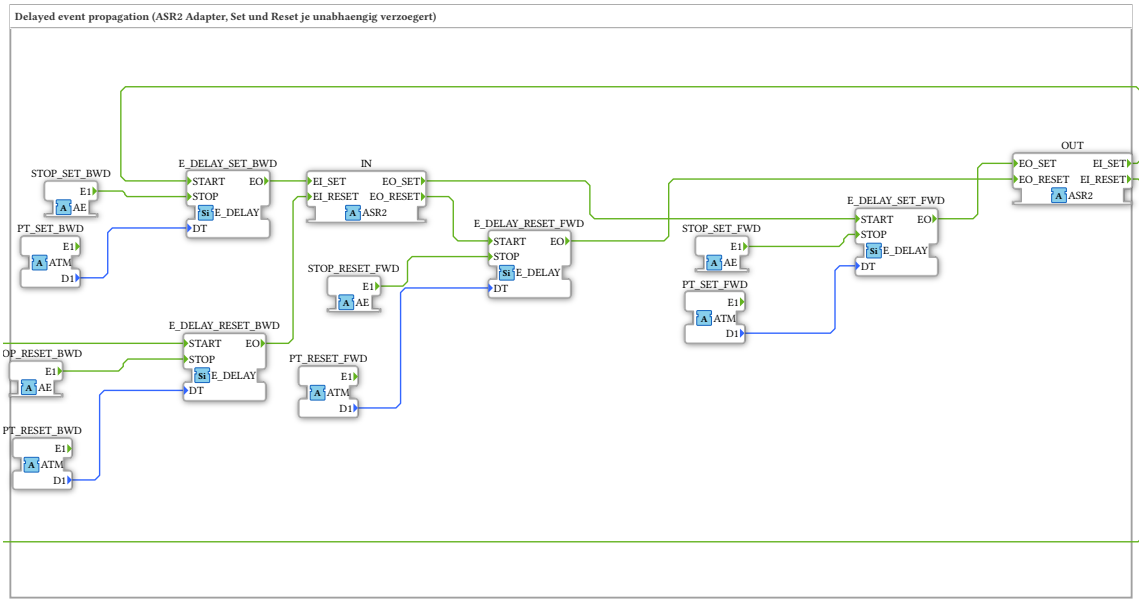
⚖️ Comparison with Similar Blocks

The same structure exists for further bidirectional adapters, e.g. `AR2_SPLIT_FWD_REV`, `AS2_SPLIT_FWD_REV`, `AX2_SPLIT_FWD_REV`, `AIS2_SPLIT_FWD_REV` and `ALR2_SPLIT_FWD_REV`; they differ only in the data type of the adapter.

Conclusion

`AS2_SPLIT_FWD_REV` is a stateless passthrough for AS2 that additionally mirrors the forward and backward channels onto two unidirectional AS plugs.

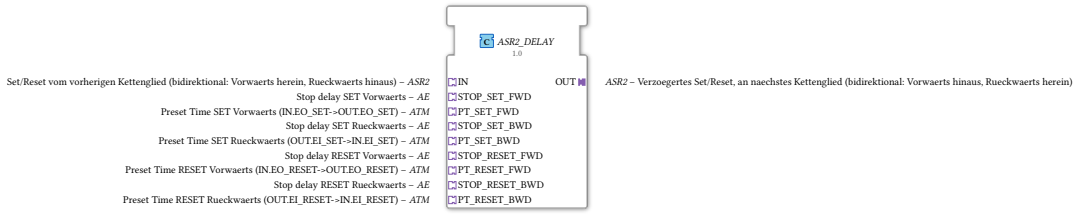
ASR2_DELAY



• – ○ • – ○ • – ○ •

Introduction

ASR2_DELAY is a composite function block that forwards the set/reset events of a bidirectional ASR2 adapter **with a delay** (e.g. for staggered switching on/off of several belts using AX_T_FF_SR_SEQ). Since ASR2 is truly bidirectional (EI_SET/EI_RESET/EO_SET/EO_RESET), **four independent** E_DELAY instances are used: set and reset, each in forward and backward direction, each with its own time and its own stop input.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs - the times are supplied via ATM sockets

Data Outputs

No data outputs

Adapters

- **IN** (socket): `adapter::types::bidirectional::ASR2` - set/reset from the previous chain element (forward in, backward out)
- **OUT** (plug): `adapter::types::bidirectional::ASR2` - delayed set/reset to the next chain element (forward out, backward in)
- **PT_SET_FWD** (socket, `adapter::types::unidirectional::ATM`): delay time SET forward (`IN.EO_SET` → `OUT.EO_SET`)
- **PT_SET_BWD** (socket, `ATM`): delay time SET backward (`OUT.EI_SET` → `IN.EI_SET`)
- **PT_RESET_FWD** (socket, `ATM`): delay time RESET forward (`IN.EO_RESET` → `OUT.EO_RESET`)
- **PT_RESET_BWD** (socket, `ATM`): delay time RESET backward (`OUT.EI_RESET` → `IN.EI_RESET`)
- **STOP_SET_FWD**, **STOP_SET_BWD**, **STOP_RESET_FWD**, **STOP_RESET_BWD** (sockets, `adapter::types::unidirectional::AE`): stop the respective delay

Function Blocks (FBs) Used

- **E_DELAY_SET_FWD**, **E_DELAY_SET_BWD**, **E_DELAY_RESET_FWD**, **E_DELAY_RESET_BWD**: each `iec61499::events::E_DELAY`

How it works

Path	START	STOP	Time (DT)	Output
SET forward	IN.EO_SET	STOP_SET_FWD.E1	PT_SET_FWD.D1	OUT.EO_SET
SET backward	OUT.EI_SET	STOP_SET_BWD.E1	PT_SET_BWD.D1	IN.EI_SET
RESET forward	IN.EO_RESET	STOP_RESET_FWD.E1	PT_RESET_FWD.D1	OUT.EO_RESET
RESET backward	OUT.EI_RESET	STOP_RESET_BWD.E1	PT_RESET_BWD.D1	IN.EI_RESET

Each incoming event starts the associated E_DELAY instance; after the time has elapsed the event is fired at the output, unless the delay was stopped beforehand via the stop socket.

Technical Details

- The four times are provided via `ATM` sockets and can therefore be set independently of each other and initialized via `initval_ATM` with `INIT`/`INITO` (versions 1.1/1.2). The identifiers use the prefix `PT` instead of `DT` because `DT` is a reserved keyword.
- Since version 1.2 both directions are delayed (previously only forward).

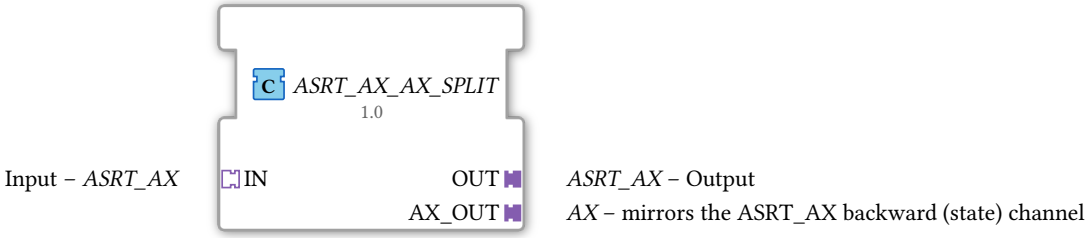
Application Scenarios

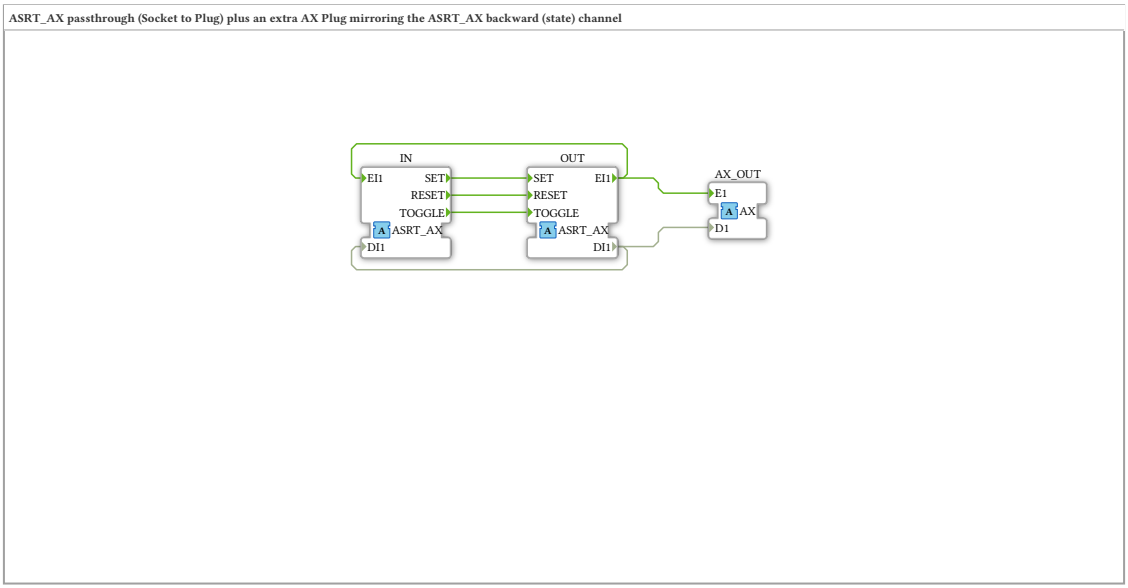
- Staggered switching on/off of consecutive conveyor belts via an `AX_T_FF_SR_SEQ` chain

Summary

`ASR2_DELAY` delays set and reset of an `ASR2` chain element in both directions with four separately adjustable times and four stop inputs.

ASRT_AX_AX_SPLIT





• – ○ • – ○ • – ○ •

Introduction

The ASRT_AX_AX_SPLIT is a composite function block that passes three incoming ASRT_AX events (Set/Reset/Toggle) unchanged from its socket `IN` to the plug `OUT`, and additionally mirrors the state (event + bool) reported on the backward channel of `OUT` to the outside via a third, unidirectional `AX_OUT` plug. It is the Toggle-extended variant of ASR_AX_AX_SPLIT.

Interface Structure

Event Inputs

No direct event inputs available – events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

- **IN**: Bidirectional adapter socket of type `adapter::types::bidirectional::ASRT_AX` (input)
- **OUT**: Bidirectional adapter plug of type `adapter::types::bidirectional::ASRT_AX` (output)
- **AX_OUT**: Unidirectional adapter plug of type `adapter::types::unidirectional::AX`, mirrors the ASRT_AX backward channel (state)

Functionality

1. `IN.SET`, `IN.RESET`, and `IN.TOGGLE` are forwarded unchanged to `OUT.SET`, `OUT.RESET`, and `OUT.TOGGLE` respectively – the forward direction is passed through 1:1.
2. The backward-channel event `OUT.EI1` (together with its associated datum `OUT.DI1`) is relayed back to `IN.EI1`/`IN.DI1` and, at the same time, forwarded to `AX_OUT.E1`/`AX_OUT.D1`.
3. As a result, the state reported via `OUT` is visible at two points: at the original socket `IN` and additionally, in isolation, at the `AX_OUT` plug.

Technical Features

- Pure event/data connections (`FBNetwork`), no own logic or state management
- The forward direction (socket → plug) is a simple 1:1 passthrough for Set, Reset, and Toggle
- The backward channel is duplicated rather than split: both destinations (`IN` and `AX_OUT`) receive the same information

State Overview

The function block has no internal state and operates statelessly. Every incoming event is immediately forwarded or mirrored.

Application Scenarios

- Additional, isolated tapping of the ASRT_AX backward channel (e.g. for a separate display or logging) without altering the actual connection between `IN` and `OUT`
- Building diagnostic/monitoring paths in ASRT_AX-based control networks

⚖️ Comparison with similar building blocks

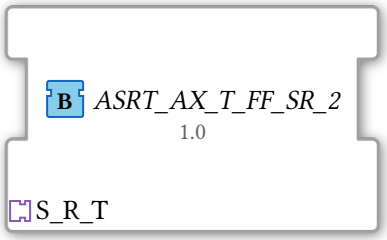
ASR_AX_AX_SPLIT follows the same pattern with only two forward events (no Toggle); AE_AX_AX_SPLIT reduces it to a single forward event.

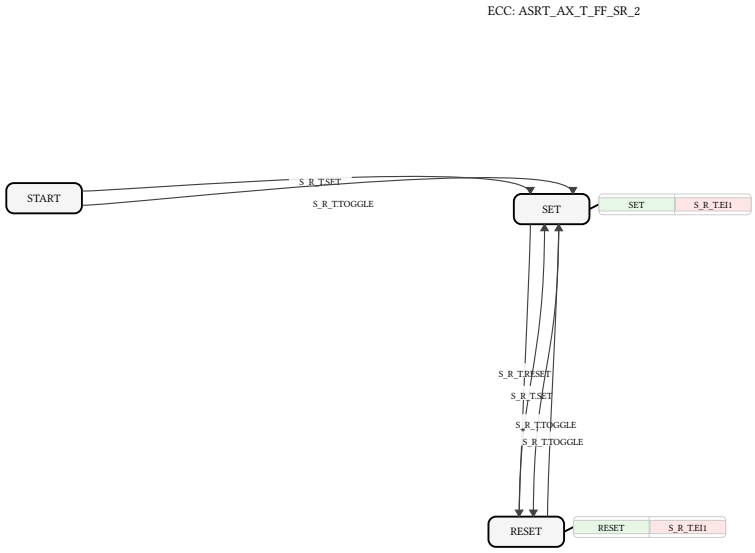
Conclusion

ASRT_AX_AX_SPLIT makes the AX backward channel of an ASRT_AX adapter pair additionally available at a second point in the network, without interrupting or altering the actual signal chain.

ASRT_AX_T_FF_SR_2

Set/Reset/Toggle (in), Zustand (out) – *ASRT_AX*





• – ○ • – ○ • – ○ •

Introduction

The ASRT_AX_T_FF_SR_2 is an event-driven bistable flip-flop with toggle functionality and a single bidirectional `ASRT_AX` socket: Set/Reset/Toggle arrive on the same adapter that also reports the new state back. It is thus the bidirectional counterpart of ASRT_AX_T_FF_SR (unidirectional ASRT socket + separate AX plug) and the Toggle-extended variant of ASR_AX_SR_2.

Interface Structure

Event Inputs

No direct event inputs available – Set/Reset/Toggle arrive via the adapter socket `S_R_T`

Event Outputs

No direct event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

- **S_R_T**: Bidirectional adapter socket of type `adapter::types::bidirectional::ASRT_AX` – Set/Reset/Toggle (input), state (output) via the same adapter

Operation

The ASRT_AX_T_FF_SR_2 has three operating states:

- **START**: Initial state
- **SET**: State TRUE
- **RESET**: State FALSE

The state transitions are controlled by the adapter events arriving at socket `S_R_T`:

- `S_R_T.SET` transitions from any state to the SET state

- `S_R_T.RESET` transitions from any state to the RESET state
- `S_R_T.TOGGLE` toggles the current state (SET ⟶ RESET or RESET ⟶ SET, and from START to SET)

With each state change, the corresponding algorithm is executed, setting `S_R_T.DI1` (`TRUE` in the SET state, `FALSE` in the RESET state) and thereby reporting the state back over the same adapter via `S_R_T.EI1`.

Technical Features

- Bidirectional communication over a single adapter socket instead of a separate ASRT socket plus AX plug
- Identical ECC structure to ASRT_AX_T_FF_SR the only difference is the adapter direction (bidirectional instead of unidirectional plus separate plug)
- A drop-in replacement for ASRT_AX_T_FF_SR in fully adapter-based (ASRT_AX) applications

State Overview

```
START (initial state)
     │
     ├── S_R_T.SET ────→ SET (S_R_T.DI1 = TRUE)
     │
     └── S_R_T.TOGGLE ─→ SET (S_R_T.DI1 = TRUE)

SET (S_R_T.DI1 = TRUE)
     │
     ├── S_R_T.RESET ──→ RESET (S_R_T.DI1 = FALSE)
     │
     └── S_R_T.TOGGLE ─→ RESET (S_R_T.DI1 = FALSE)

RESET (S_R_T.DI1 = FALSE)
     │
     ├── S_R_T.SET ────→ SET (S_R_T.DI1 = TRUE)
     │
     └── S_R_T.TOGGLE ─→ SET (S_R_T.DI1 = TRUE)
```

Application Scenarios

- State storage with Set/Reset/Toggle over a single bidirectional adapter connection, without wiring a separate state output
- Replacement for ASRT_AX_T_FF_SR in networks that consistently use ASRT_AX adapters

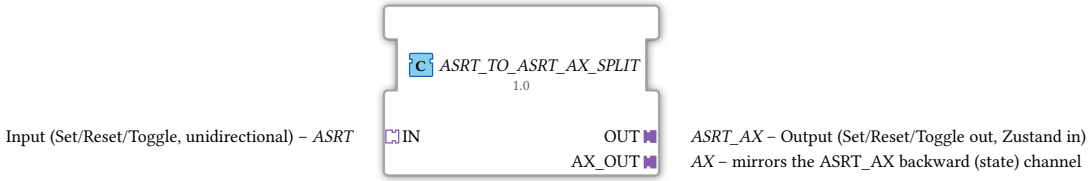
⚖️ Comparison with similar components

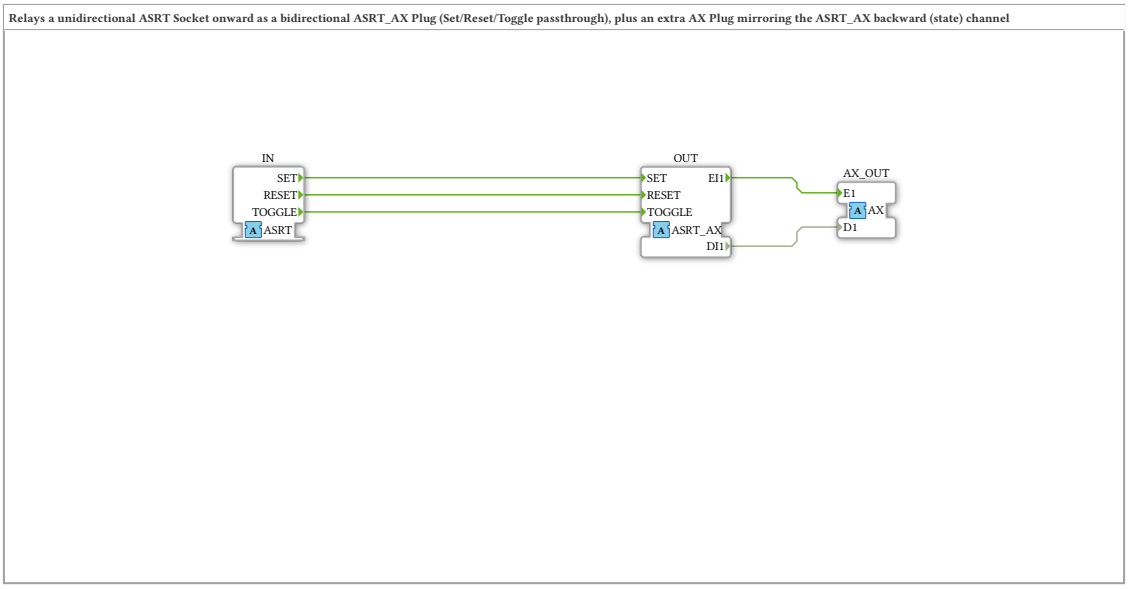
Compared to ASRT_AX_T_FF_SR, which uses a unidirectional ASRT socket and a separate AX plug for the state, ASRT_AX_T_FF_SR_2 bundles both directions into a single ASRT_AX socket. ASR_AX_SR_2 is the same family without the toggle input.

Conclusion

ASRT_AX_T_FF_SR_2 carries the proven Set/Reset/Toggle flip-flop pattern over into the bidirectional adapter world: control input and state feedback share the same ASRT_AX socket, simplifying wiring in adapter-based networks.

ASRT_TO_ASRT_AX_SPLIT





• – ○ • – ○ • – ○ •

Introduction

ASRT_TO_ASRT_AX_SPLIT is the set/reset/toggle variant of `ASR_TO_ASR_AX_SPLIT`: it converts an incoming **unidirectional** ASRT signal (set/reset/toggle, no backward channel) at its socket `IN` into a **bidirectional** ASRT_AX signal at plug `OUT`, and additionally mirrors the state reported on `OUT`'s backward channel out through a third, unidirectional `AX_OUT` plug.

Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **IN**: Unidirectional adapter socket of type `adapter::types::unidirectional::ASRT` (set/reset/toggle input, no backward channel)
- **OUT**: Bidirectional adapter plug of type `adapter::types::bidirectional::ASRT_AX` (set/reset/toggle output, with backward channel)
- **AX_OUT**: Unidirectional adapter plug of type `adapter::types::unidirectional::AX`, mirrors the ASRT_AX backward channel (state) to the outside

How it works

1. Every event arriving at `IN.SET` is forwarded to `OUT.SET` unchanged, likewise `IN.RESET` to `OUT.RESET` and `IN.TOGGLE` to `OUT.TOGGLE` - all three event paths run independently of each other.
2. The backward-channel event `OUT.EI1` (with its associated data `OUT.DI1`), sent by the downstream counterpart via `OUT`, is forwarded exclusively to `AX_OUT.E1`/`AX_OUT.D1` - relaying it back to `IN` is technically impossible, since `IN` is a unidirectional ASRT adapter with no backward channel.
3. `AX_OUT` is therefore the only place where the state reported by `OUT` becomes visible.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic or state management of its own
- Three independent forward paths (SET, RESET, TOGGLE), each a simple 1:1 passthrough
- Converts a unidirectional adapter (`IN`) into a bidirectional one (`OUT`) without needing a backward channel on the input itself
- The backward channel is not duplicated, but exposed solely via `AX_OUT`

State Overview

The function block has no internal state and works statelessly. Every incoming event is forwarded/mirrored immediately.

Application Scenarios

- Connecting an existing, purely unidirectional ASRT signal source (e.g. a SoftKey with set/reset/toggle behavior) to a chain that expects a bidirectional ASRT_AX adapter
- Exposing the state of the downstream ASRT_AX counterpart via `AX_OUT`, even though the original source (`IN`) itself supports no backward channel

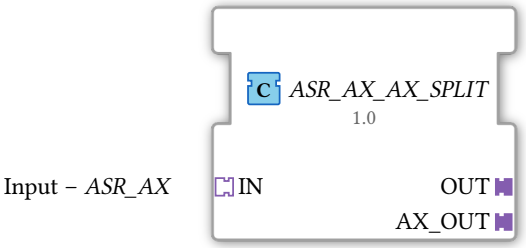
⚖️ Comparison with Similar Blocks

Structurally identical to `ASR_TO_ASR_AX_SPLIT`, just with one additional third forward event (TOGGLE). For the simpler set/reset variant without toggle, see `ASR_TO_ASR_AX_SPLIT` for the pure event variant without set/reset semantics, see `AE_TO_AE_AX_SPLIT`.

Summary

ASRT_TO_ASRT_AX_SPLIT allows a purely unidirectional ASRT signal source to connect to a bidirectional ASRT_AX chain and makes its backward channel available in isolation via `AX_OUT`, without requiring the original input itself to support a backward channel.

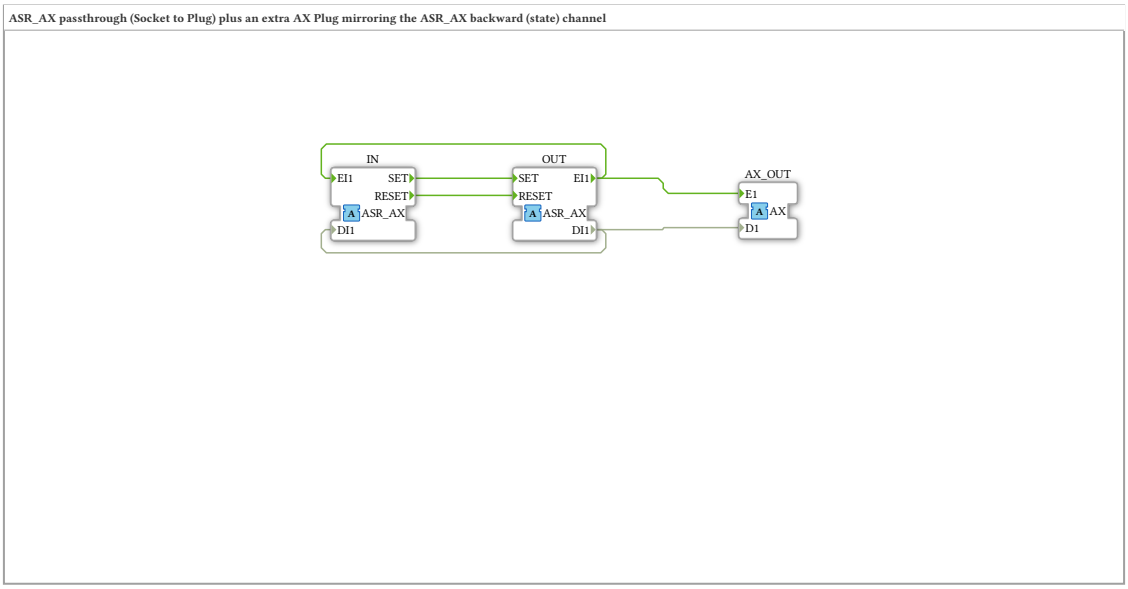
ASR_AX_AX_SPLIT



Input – *ASR_AX*

ASR_AX – Output

AX – mirrors the ASR_AX backward (state) channel



• – ○ • – ○ • – ○ •

Introduction

The ASR_AX_AX_SPLIT is a composite function block that passes two incoming ASR_AX events (Set/Reset) unchanged from its socket `IN` to the plug `OUT`, and additionally mirrors the state (event + bool) reported on the backward channel of `OUT` to the outside via a third, unidirectional `AX_OUT` plug. It is the Set/Reset-extended variant of AE_AX_AX_SPLIT.

Interface Structure

Event Inputs

No direct event inputs available – events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

- **IN**: Bidirectional adapter socket of type `adapter::types::bidirectional::ASR_AX` (input)
- **OUT**: Bidirectional adapter plug of type `adapter::types::bidirectional::ASR_AX` (output)
- **AX_OUT**: Unidirectional adapter plug of type `adapter::types::unidirectional::AX`, mirrors the ASR_AX backward channel (state)

Functionality

1. `IN.SET` and `IN.RESET` are forwarded unchanged to `OUT.SET` and `OUT.RESET` respectively – the forward direction is passed through 1:1.
2. The backward-channel event `OUT.EI1` (together with its associated datum `OUT.DI1`) is relayed back to `IN.EI1`/`IN.DI1` and, at the same time, forwarded to `AX_OUT.E1`/`AX_OUT.D1`.
3. As a result, the state reported via `OUT` is visible at two points: at the original socket `IN` and additionally, in isolation, at the `AX_OUT` plug.

Technical Features

- Pure event/data connections (`FBNetwork`), no own logic or state management
- The forward direction (socket ⟶ plug) is a simple 1:1 passthrough for Set and Reset
- The backward channel is duplicated rather than split: both destinations (`IN` and `AX_OUT`) receive the same information

State Overview

The function block has no internal state and operates statelessly. Every incoming event is immediately forwarded or mirrored.

Application Scenarios

- Additional, isolated tapping of the ASR_AX backward channel (e.g. for a separate display or logging) without altering the actual connection between `IN` and `OUT`
- Building diagnostic/monitoring paths in ASR_AX-based control networks

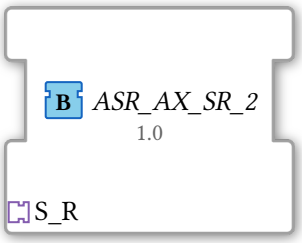
⚖️ Comparison with similar building blocks

AE_AX_AX_SPLIT follows the same pattern with only one forward event; ASRT_AX_AX_SPLIT extends it with a third forward event (Toggle).

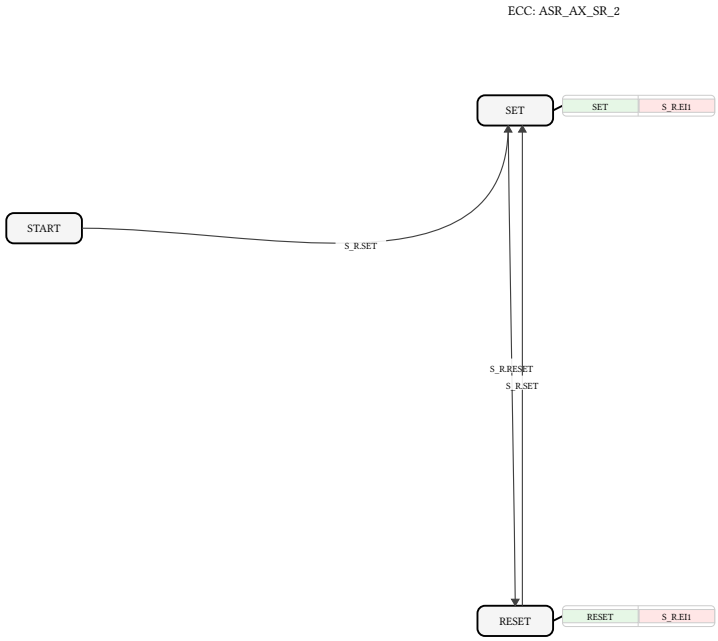
Conclusion

ASR_AX_AX_SPLIT makes the AX backward channel of an ASR_AX adapter pair additionally available at a second point in the network, without interrupting or altering the actual signal chain.

ASR_AX_SR_2



Set/Reset (in), Zustand (out) – *ASR_AX*



• – ○ • – ○ • – ○ •

Introduction

The ASR_AX_SR_2 is an event-driven bistable flip-flop (Set/Reset) with a single bidirectional `ASR_AX` socket: the Set and Reset events arrive on the same adapter that also reports the new state back – unlike ASR_AX_SR, which uses a unidirectional ASR socket and a separate AX plug for the state.

Interface Structure

Event Inputs

No direct event inputs available – Set/Reset arrive via the adapter socket `S_R`

Event Outputs

No direct event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

- **S_R**: Bidirectional adapter socket of type `adapter::types::bidirectional::ASR_AX` – Set/Reset (input), state (output) via the same adapter

Operation

The ASR_AX_SR_2 has three operating states:

- **START**: Initial state
- **SET**: State TRUE
- **RESET**: State FALSE

The state transitions are controlled by the adapter events arriving at socket `S_R`:

- `S_R.SET` transitions from START or RESET to the SET state
- `S_R.RESET` transitions from the SET state to the RESET state

With each state change, the corresponding algorithm is executed, setting `S_R.DI1` (`TRUE` in the SET state, `FALSE` in the RESET state) and thereby reporting the state back over the same adapter via `S_R.EI1`.

Technical Features

- Bidirectional communication over a single adapter socket instead of a separate ASR socket plus AX plug
- No toggle input – a pure Set/Reset flip-flop (unlike ASRT_AX_T_FF_SR_2)
- A drop-in replacement for ASR_AX_SR in fully adapter-based (ASR_AX) applications

State Overview

```
START (initial state)
    │
    └── S_R.SET ──→ SET (S_R.DI1 = TRUE)

SET (S_R.DI1 = TRUE)
    │
    └── S_R.RESET ─→ RESET (S_R.DI1 = FALSE)

RESET (S_R.DI1 = FALSE)
    │
    └── S_R.SET ──→ SET (S_R.DI1 = TRUE)
```

Application Scenarios

- State storage (Set/Reset) over a single bidirectional adapter connection, without wiring a separate state output
- Replacement for ASR_AX_SR in networks that consistently use ASR_AX adapters

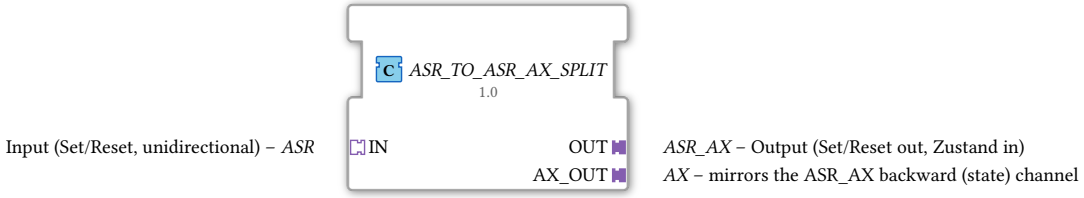
⚖️ Comparison with similar components

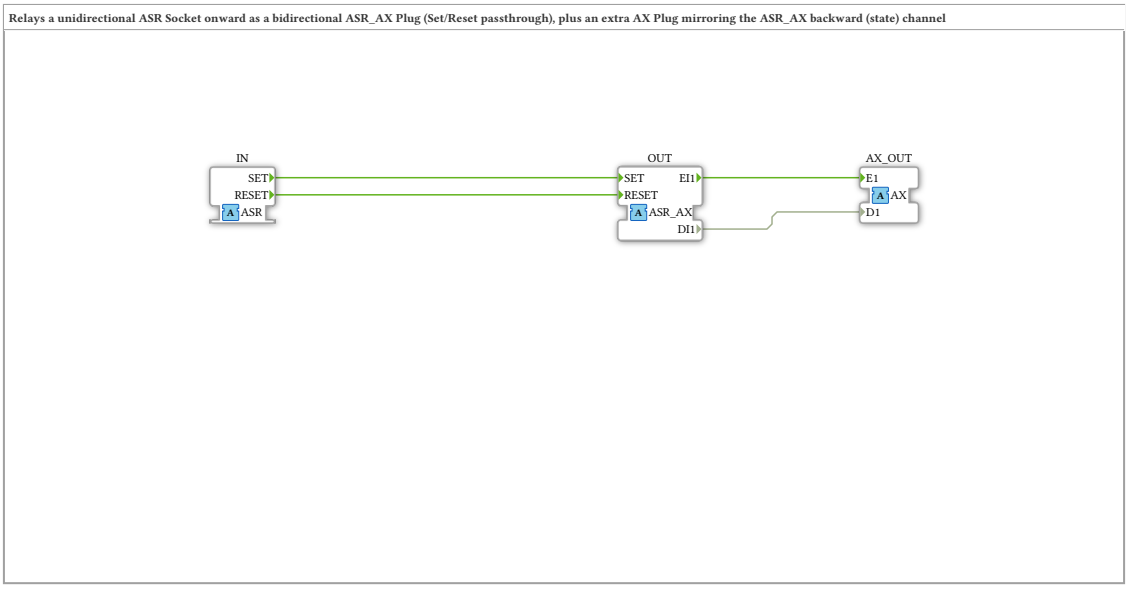
Compared to ASR_AX_SR, which uses a unidirectional ASR socket and a separate AX plug for the state, ASR_AX_SR_2 bundles both directions into a single ASR_AX socket. ASRT_AX_T_FF_SR_2 extends the same pattern with an additional toggle input.

Conclusion

ASR_AX_SR_2 carries the proven Set/Reset flip-flop pattern over into the bidirectional adapter world: Set/Reset input and state feedback share the same ASR_AX socket, simplifying wiring in adapter-based networks.

ASR_TO_ASR_AX_SPLIT





• – ○ • – ○ • – ○ •

Introduction

ASR_TO_ASR_AX_SPLIT is the set/reset variant of AE_TO_AE_AX_SPLIT: it converts an incoming **unidirectional** ASR signal (set/reset, no backward channel) at its socket `IN` into a **bidirectional** ASR_AX signal at plug `OUT`, and additionally mirrors the state reported on `OUT`'s backward channel out through a third, unidirectional `AX_OUT` plug.

Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **IN**: Unidirectional adapter socket of type `adapter::types::unidirectional::ASR` (set/reset input, no backward channel)
- **OUT**: Bidirectional adapter plug of type `adapter::types::bidirectional::ASR_AX` (set/reset output, with backward channel)
- **AX_OUT**: Unidirectional adapter plug of type `adapter::types::unidirectional::AX`, mirrors the ASR_AX backward channel (state) to the outside

How it works

1. Every event arriving at `IN.SET` is forwarded to `OUT.SET` unchanged, and likewise every `IN.RESET` to `OUT.RESET` - both event paths run independently of each other.
2. The backward-channel event `OUT.EI1` (with its associated data `OUT.DI1`), sent by the downstream counterpart via `OUT`, is forwarded exclusively to `AX_OUT.E1`/`AX_OUT.D1` - relaying it back to `IN` is technically impossible, since `IN` is a unidirectional ASR adapter with no backward channel.
3. `AX_OUT` is therefore the only place where the state reported by `OUT` becomes visible.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic or state management of its own
- Two independent forward paths (SET, RESET), each a simple 1:1 passthrough
- Converts a unidirectional adapter (`IN`) into a bidirectional one (`OUT`) without needing a backward channel on the input itself
- The backward channel is not duplicated, but exposed solely via `AX_OUT`

State Overview

The function block has no internal state and works statelessly. Every incoming event is forwarded/mirrored immediately.

Application Scenarios

- Connecting an existing, purely unidirectional ASR signal source (e.g. a simple set/reset push button) to a chain that expects a bidirectional ASR_AX adapter
- Exposing the state of the downstream ASR_AX counterpart via `AX_OUT`, even though the original source (`IN`) itself supports no backward channel

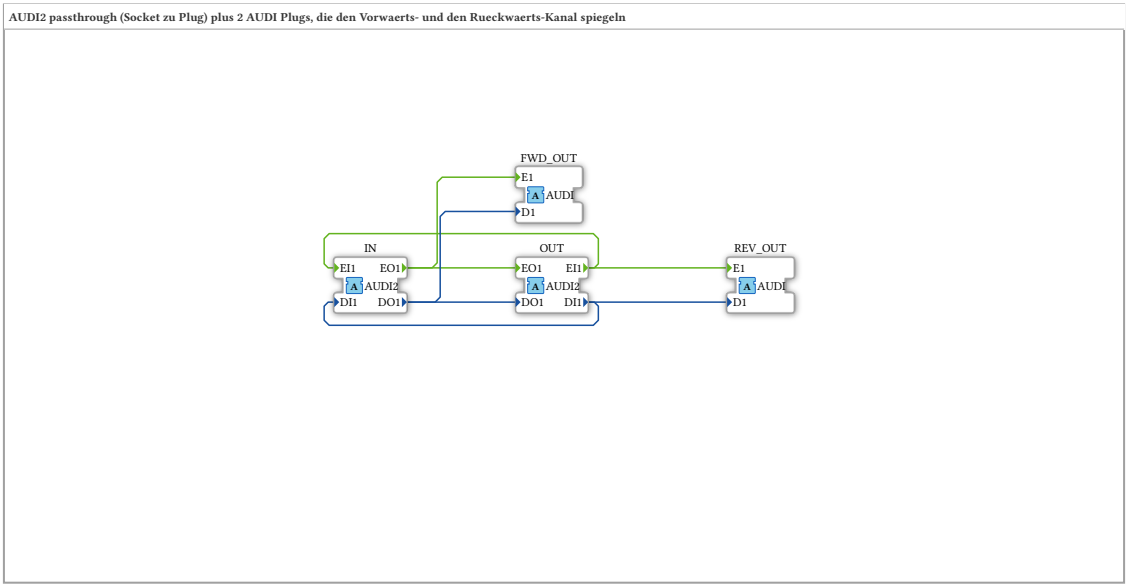
⚖️ Comparison with Similar Blocks

Structurally identical to `AE_TO_AE_AX_SPLIT`, just with two forward events (SET, RESET) instead of one. For the additional TOGGLE variant, see `ASRT_TO_ASRT_AX_SPLIT`.

Summary

ASR_TO_ASR_AX_SPLIT allows a purely unidirectional ASR signal source to connect to a bidirectional ASR_AX chain and makes its backward channel available in isolation via `AX_OUT`, without requiring the original input itself to support a backward channel.

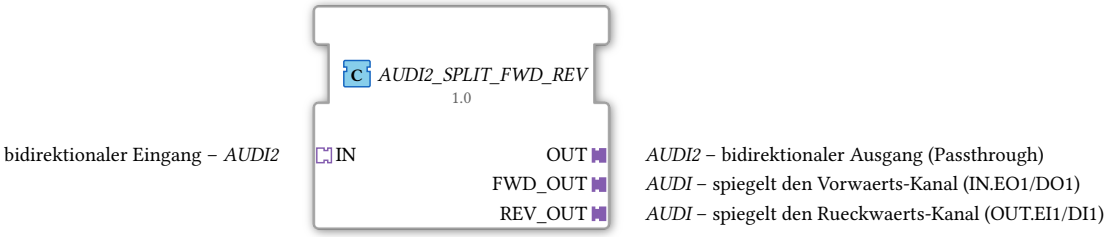
AUDI2_SPLIT_FWD_REV



• — ○ • — ○ • — ○ •

Introduction

`AUDI2_SPLIT_FWD_REV` is a composite function block that passes a bidirectional `AUDI2` adapter from socket `IN` unchanged through to plug `OUT` (passthrough) and additionally mirrors the forward and the backward channel onto two separate, unidirectional `AUDI` plugs. This allows both directions of an `AUDI2` connection to be tapped elsewhere without altering the connection itself.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **IN** (socket): Bidirectional adapter of type `adapter::types::bidirectional::AUDI2` (bidirectional input)
- **OUT** (plug): Bidirectional adapter of type `adapter::types::bidirectional::AUDI2` (bidirectional output, passthrough)
- **FWD_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AUDI`, mirrors the forward channel (`IN.EO1/DO1`)
- **REV_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AUDI`, mirrors the backward channel (`OUT.EI1/DI1`)

How it works

1. **Forward (indication)**: `IN.EO1/IN.DO1` is forwarded unchanged to `OUT.EO1/OUT.DO1` and additionally mirrored to `FWD_OUT.E1/FWD_OUT.D1`.
2. **Backward (request)**: `OUT.EI1/OUT.DI1` is forwarded unchanged to `IN.EI1/IN.DI1` and additionally mirrored to `REV_OUT.E1/REV_OUT.D1`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- Each passthrough event output feeds two targets (fan-out); the data runs in parallel with the events
- The two mirror plugs are unidirectional (`AUDI`) and therefore provide read access to the respective channel only

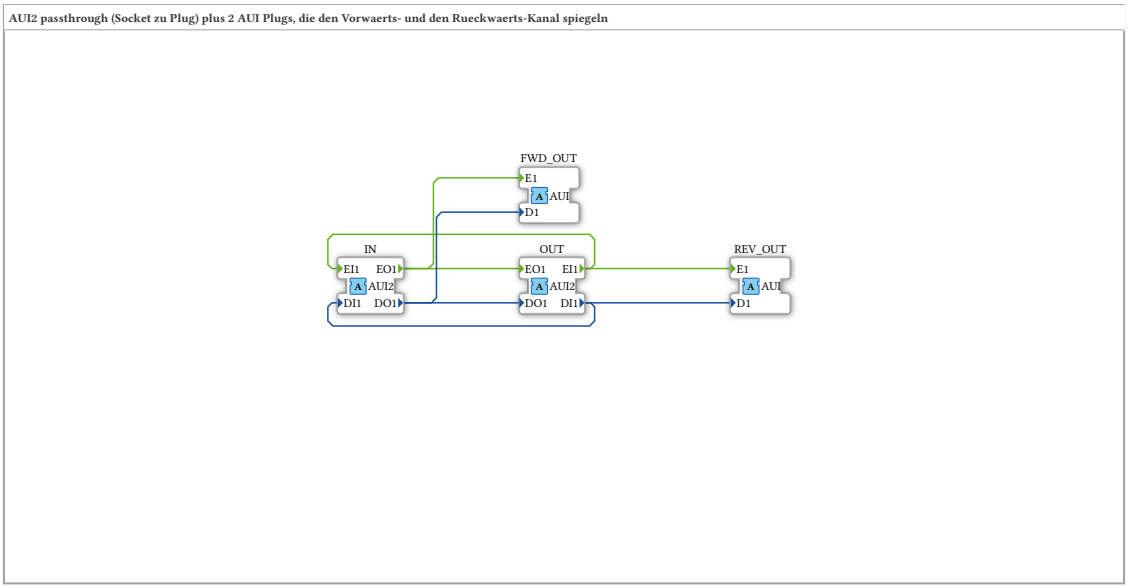
Application Scenarios

- Monitoring/diagnosing both directions of a bidirectional `AUDI2` connection
- Splitting an `AUDI2` connection into a separately evaluable forward and backward stream without interrupting the main connection

Summary

`AUDI2_SPLIT_FWD_REV` passes a bidirectional `AUDI2` connection through transparently and additionally provides the forward and backward channels as separate `AUDI` adapters.

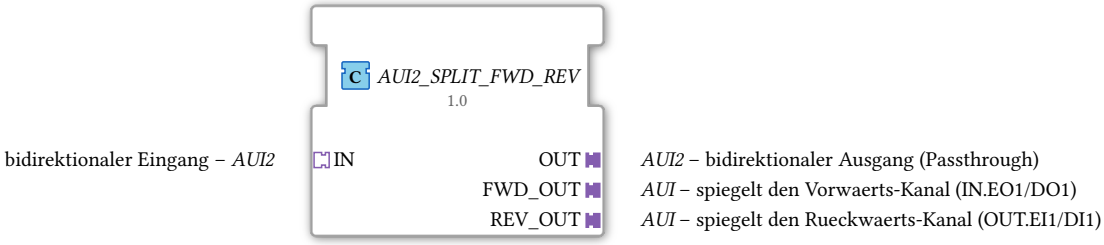
AUI2_SPLIT_FWD_REV



• – ○ • – ○ • – ○ •

Introduction

`AUI2_SPLIT_FWD_REV` is a composite function block that passes a bidirectional `AUI2` adapter from socket `IN` unchanged through to plug `OUT` (passthrough) and additionally mirrors the forward and the backward channel onto two separate, unidirectional `AUI` plugs. This allows both directions of an `AUI2` connection to be tapped elsewhere without altering the connection itself.



Interface Structure

Event Inputs

No direct event inputs - events arrive via the adapter sockets/plugs

Event Outputs

No direct event outputs

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

- **IN** (socket): Bidirectional adapter of type `adapter::types::bidirectional::AUI2` (bidirectional input)
- **OUT** (plug): Bidirectional adapter of type `adapter::types::bidirectional::AUI2` (bidirectional output, passthrough)
- **FWD_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AUI`, mirrors the forward channel (`IN.EO1/DO1`)

- **REV_OUT** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AUI`, mirrors the backward channel (`OUT.EI1/DI1`)

How it works

1. **Forward (indication)**: `IN.EO1/IN.DO1` is forwarded unchanged to `OUT.EO1/OUT.DO1` and additionally mirrored to `FWD_OUT.E1/FWD_OUT.D1`.
2. **Backward (request)**: `OUT.EI1/OUT.DI1` is forwarded unchanged to `IN.EI1/IN.DI1` and additionally mirrored to `REV_OUT.E1/REV_OUT.D1`.

Technical Details

- Pure event/data connections (`FBNetwork`), no logic of its own and no internal state
- Each passthrough event output feeds two targets (fan-out); the data runs in parallel with the events
- The two mirror plugs are unidirectional (`AUI`) and therefore provide read access to the respective channel only

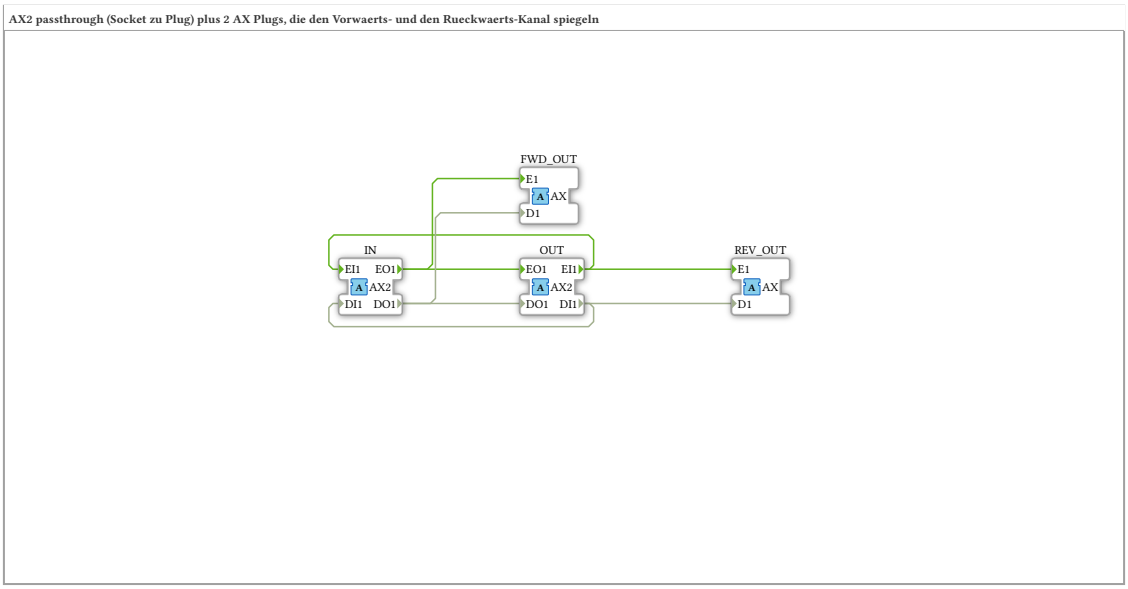
Application Scenarios

- Monitoring/diagnosing both directions of a bidirectional `AUI2` connection
- Splitting an `AUI2` connection into a separately evaluable forward and backward stream without interrupting the main connection

Summary

`AUI2_SPLIT_FWD_REV` passes a bidirectional `AUI2` connection through transparently and additionally provides the forward and backward channels as separate `AUI` adapters.

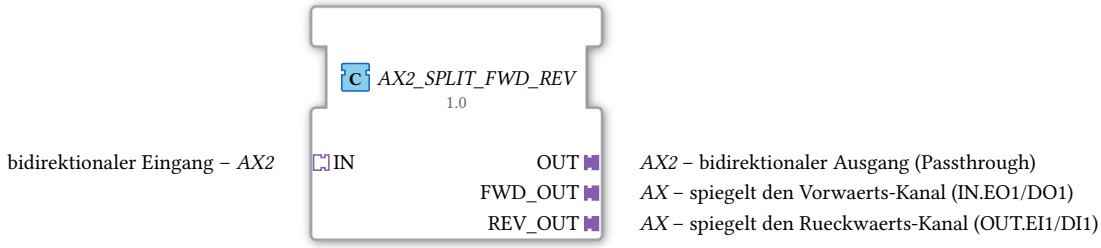
AX2_SPLIT_FWD_REV



• – ◦ • – ◦ • – ◦ •

Introduction

The `AX2_SPLIT_FWD_REV` function block passes a bidirectional `AX2` adapter unchanged from `IN` (socket) to `OUT` (plug) and additionally mirrors both channels onto one unidirectional `AX` plug each: the forward channel (indication, `IN.EO1/DO1`) onto `FWD_OUT` and the backward channel (request, `OUT.EI1/DI1`) onto `REV_OUT`. This allows both directions of a bidirectional AX2 connection to be tapped separately as plain AX signals.



Interface Structure

Event Inputs

The function block has no direct event inputs.

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **IN** (socket): bidirectional input, type `adapter::types::bidirectional::AX2`
- **OUT** (plug): bidirectional output (passthrough), type `adapter::types::bidirectional::AX2`
- **FWD_OUT** (plug): mirrors the forward channel (`IN.EO1`/`DO1`), type `adapter::types::unidirectional::AX`
- **REV_OUT** (plug): mirrors the backward channel (`OUT.EI1`/`DI1`), type `adapter::types::unidirectional::AX`

Functionality

The block consists of wiring only:

- **Forward** (indication): `IN.EO1` ⟶ `OUT.EO1` and additionally `IN.EO1` ⟶ `FWD_OUT.E1`; `IN.DO1` ⟶ `OUT.DO1` and additionally `IN.DO1` ⟶ `FWD_OUT.D1`.
- **Backward** (request): `OUT.EI1` ⟶ `IN.EI1` and additionally `OUT.EI1` ⟶ `REV_OUT.E1`; `OUT.DI1` ⟶ `IN.DI1` and additionally `OUT.DI1` ⟶ `REV_OUT.D1`.

Both directions therefore pass through unchanged, and each one is additionally output as a plain unidirectional AX signal.

Technical Details

- Pure wiring without logic or state
- The passthrough `IN` ⟶ `OUT` is fully preserved; `FWD_OUT` and `REV_OUT` are pure observation taps
- Each source is fanned out to two destinations (event and data); the behaviour corresponds to a split onto both branches

Application Scenarios

- Monitoring the forward and the backward direction of a AX2 connection (e.g. for display, logging or diagnostics) without altering the connection itself
- Separate further processing of indication and request as plain AX signals

⚖️ Comparison with Similar Blocks

The same structure exists for further bidirectional adapters, e.g. `AR2_SPLIT_FWD_REV`, `AS2_SPLIT_FWD_REV`, `AX2_SPLIT_FWD_REV`, `AIS2_SPLIT_FWD_REV` and `ALR2_SPLIT_FWD_REV`; they differ only in the data type of the adapter.

Conclusion

`AX2_SPLIT_FWD_REV` is a stateless passthrough for AX2 that additionally mirrors the forward and backward channels onto two unidirectional AX plugs.

unidirectional

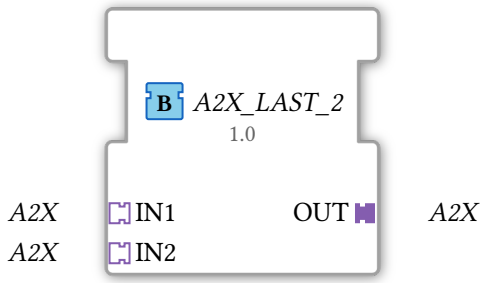
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

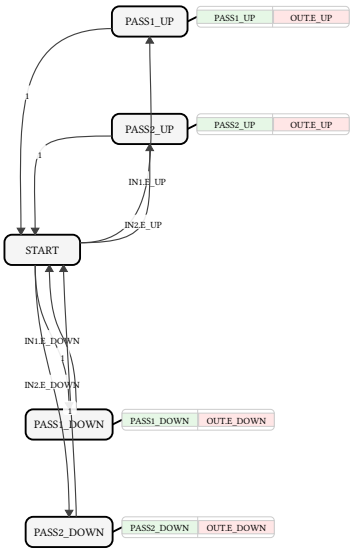
- A2X_SPLIT_2
- A2X_SPLIT_2_UNGATED
- A2X_SPLIT_3
- A2X_SPLIT_3_UNGATED
- A2X_SPLIT_4
- A2X_SPLIT_4_UNGATED
- A2X_SPLIT_5
- A2X_SPLIT_5_UNGATED
- A2X_SPLIT_6
- A2X_SPLIT_6_UNGATED
- A2X_SPLIT_7
- A2X_SPLIT_7_UNGATED
- A2X_SPLIT_8
- A2X_SPLIT_8_UNGATED
- A2X_SPLIT_9
- A2X_SPLIT_9_UNGATED
- ASR_AX_SR
- ASR_AX_SR_SYM
- ASRT_AX_T_FF_SR
- AX_AE_MERGE
- AX_ASR_RF_TRIG
- AX_ASR_RF_TRIG_X
- AX_ASR_SWITCH
- AX_ASR_SWITCH_X
- AX_D_FF
- AX_D_FF_TMIN
- AX_E_PERMIT
- AX_E_SWITCH
- AX_F_TRIG
- AX_PERMIT
- AX_R_TRIG
- AX_RF_TRIG
- AX_RS
- AX_RS_SYM
- AX_RS_SYM_INIT
- AX_SELECT
- AX_SPLIT_2
- AX_SPLIT_2_UNGATED
- AX_SPLIT_3
- AX_SPLIT_3_UNGATED
- AX_SPLIT_4
- AX_SPLIT_4_UNGATED
- AX_SPLIT_5
- AX_SPLIT_5_UNGATED
- AX_SPLIT_6
- AX_SPLIT_6_UNGATED
- AX_SPLIT_7
- AX_SPLIT_7_UNGATED
- AX_SPLIT_8
- AX_SPLIT_8_UNGATED
- AX_SPLIT_9

- AX_SPLIT_9_UNGATED
- AX_SR
- AX_SR_SYM
- AX_SR_SYM_INIT
- AX_SWITCH
- AX_T_FF
- AX_T_FF_INIT
- AX_T_FF_SR
- AX_T_FF_SR_SYM
- AX_T_FF_SR_SYM_INIT
- AX_T_FF_SR_SYM_STORE
- E_RS_SYM
- E_RS_SYM_INIT
- E_SR_SYM
- E_SR_SYM_INIT
- E_T_FF_INIT
- E_T_FF_SR_SYM
- E_T_FF_SR_SYM_INIT
- E_T_FF_SR_SYM_STORE

A2X_LAST_2



ECC: A2X_LAST_2



Introduction

The A2X_LAST_2 function block merges two A2X adapter signals (**IN1**, **IN2**) into one shared output (**OUT**). Unlike the single-channel AX_LAST_2, A2X carries two independent channels (**UP** and **DOWN**), which are therefore also merged independently by last-writer-wins: whichever socket last wrote on UP wins for UP; whichever last wrote on DOWN wins for DOWN - regardless of what happens on the other channel.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: A2X adapter (unidirectional) - first signal source (UP + DOWN)
- **IN2**: A2X adapter (unidirectional) - second signal source (UP + DOWN)

Output adapter (Plug):

- **OUT**: A2X adapter (unidirectional) - merged signal (UP + DOWN)

How it works

The block is implemented as a Basic FB with a 5-state ECC (START, PASS1_UP, PASS2_UP, PASS1_DOWN, PASS2_DOWN) that serves both channels through the same ECC - since only one event is processed per FB invocation, UP and DOWN processing never collide. When an event arrives at IN1.UP or IN2.UP, that UP value is copied to OUT.UP and OUT.E_UP is fired. Symmetrically, an event at IN1.DOWN/IN2.DOWN copies the DOWN value to OUT.DOWN and fires OUT.E_DOWN. The two channels therefore have independent “winners” - IN1 might currently dominate the UP channel while IN2 dominates the DOWN channel.

Technical Details

- Two independent last-writer-wins merges (UP and DOWN) in one block
- Extends the generic single-channel LAST_2 pattern to the 2-channel A2X adapter type
- One shared ECC for both channels, since only one event is processed per invocation - no state conflict possible
- No signal delay: the value is passed through in the same cycle the triggering event arrives

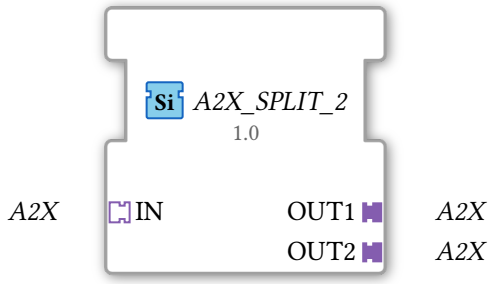
Application Scenarios

- Merging two equally-valid A2X signal sources (e.g. two operator stations for an up/down control) into one downstream adapter path
- Arbitrating between two independent up/down sources where UP and DOWN may come from different origins

⚖️ Comparison with Similar Blocks

A2X_LAST_2 is the two-channel variant of the generic AX_LAST_2 pattern, applied to the A2X adapter type with its two independent UP/DOWN channels. For mixed operation between a data adapter and a pure event adapter, use AX_AE_MERGE instead.

A2X_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `A2X_SPLIT_2` splits an incoming A2X adapter signal into two identical output adapters. It is implemented as a generic function block (`GEN_A2X_SPLIT`) and allows an A2X signal to be forwarded to two independent target function blocks without signal loss or modification.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::A2X`	`IN`	Socket	Input adapter that receives the A2X signal to be distributed.
`adapter::types::unidirectional::A2X`	`OUT1`	Plug	First output adapter that provides an identical copy of the input signal.
`adapter::types::unidirectional::A2X`	`OUT2`	Plug	Second output adapter that provides an identical copy of the input signal.

Functionality

The function block forwards the A2X adapter signal present at socket `IN` directly to the two plugs `OUT1` and `OUT2` without any transformation or delay. Every event transmitted via the A2X adapter appears simultaneously and unchanged at both outputs. The function block has no internal logic, no state, and no latency.

Technical Features

- **Generic Function Block**: The function block is declared as the generic type `GEN_A2X_SPLIT` and utilizes the Eclipse 4diac Generic Class concept. This allows for flexible reuse in different projects.
- **Pure Adapter Forwarding**: No event or data inputs/outputs are used; all communication takes place exclusively via the A2X adapter interfaces.
- **Unidirectional**: The A2X adapter is unidirectional, therefore data and events only flow from the socket to the plugs (input → output).

State Overview

The module has no internal state machine and is completely **stateless**. Every input value is immediately passed to both outputs without storage.

Application Scenarios

- **Signal Multiplication**: An A2X signal provided by a sensor or controller is to be distributed to two different devices (e.g., actuators, displays, superimposed logic) simultaneously.
- **Redundancy or Monitoring**: A signal is to be routed to a diagnostic or monitoring module in parallel with a primary destination.
- **Testing and Simulation**: An A2X signal is to be connected to both the actual hardware and a simulator during testing.

Comparison with Similar Modules

- **A2X_SPLIT_3**: Analog module with three outputs instead of two. Both modules function identically, differing only in the number of output plugs.
- **A2X_MERGE**: Combines two A2X signals into one – opposite functionality.
- **Generic Splitters (e.g., for other adapter types)**: Same operating principle, but adapted to the adapter type.

Conclusion

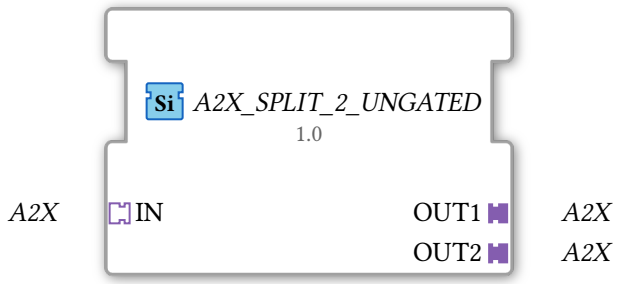
The `A2X_SPLIT_2` is a simple yet useful module for signal distribution in 4diac applications. Its generic nature allows for flexible use without additional configuration. It is particularly suitable for scenarios where an A2X signal needs to be passed on to multiple receivers without compromising signal integrity.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

A2X_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `A2X_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `A2X_SPLIT_2_UNGATED` splits an incoming A2X adapter signal into two identical output adapters. It is implemented as a generic function block (`GEN_A2X_SPLIT`) and allows an A2X signal to be forwarded to two independent target function blocks without signal loss or modification.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Type	Name	Direction	Description

adapter::types::unidirectional::A2X	IN	Socket	Input adapter that receives the A2X signal to be distributed.
adapter::types::unidirectional::A2X	OUT1	Plug	First output adapter that provides an identical copy of the input signal.
adapter::types::unidirectional::A2X	OUT2	Plug	Second output adapter that provides an identical copy of the input signal.

Functionality

The function block forwards the A2X adapter signal present at socket `IN` directly to the two plugs `OUT1` and `OUT2` without any transformation or delay. Every event transmitted via the A2X adapter appears simultaneously and unchanged at both outputs. The function block has no internal logic, no state, and no latency.

Technical Features

- **Generic Function Block**: The function block is declared as the generic type `GEN_A2X_SPLIT` and utilizes the Eclipse 4diac Generic Class concept. This allows for flexible reuse in different projects.
- **Pure Adapter Forwarding**: No event or data inputs/outputs are used; all communication takes place exclusively via the A2X adapter interfaces.
- **Unidirectional**: The A2X adapter is unidirectional, therefore data and events only flow from the socket to the plugs (input → output).

State Overview

The module has no internal state machine and is completely **stateless**. Every input value is immediately passed to both outputs without storage.

Application Scenarios

- **Signal Multiplication**: An A2X signal provided by a sensor or controller is to be distributed to two different devices (e.g., actuators, displays, superimposed logic) simultaneously.
- **Redundancy or Monitoring**: A signal is to be routed to a diagnostic or monitoring module in parallel with a primary destination.
- **Testing and Simulation**: An A2X signal is to be connected to both the actual hardware and a simulator during testing.

Comparison with Similar Modules

- **A2X_SPLIT_3**: Analog module with three outputs instead of two. Both modules function identically, differing only in the number of output plugs.
- **A2X_MERGE**: Combines two A2X signals into one – opposite functionality.
- **Generic Splitters (e.g., for other adapter types)**: Same operating principle, but adapted to the adapter type.

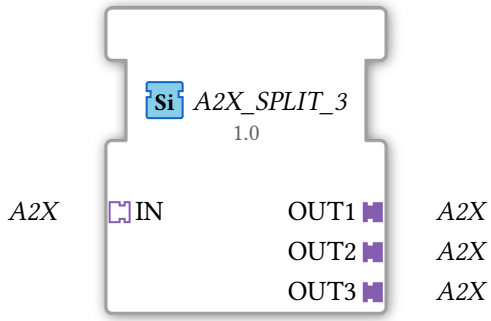
Conclusion

The `A2X_SPLIT_2_UNGATED` is a simple yet useful module for signal distribution in 4diac applications. Its generic nature allows for flexible use without additional configuration. It is particularly suitable for scenarios where an A2X signal needs to be passed on to multiple receivers without compromising signal integrity.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

A2X_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **A2X_SPLIT_3** is used to split an incoming A2X adapter data stream into three identical outputs. It is implemented as a generic function block (FB) and is suitable for applications where a signal is required multiple times.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::A2X`	**IN**	Socket (Input)	Source adapter whose data stream is split.
`adapter::types::unidirectional::A2X`	**OUT1**	Plug (Output)	First identical output.
`adapter::types::unidirectional::A2X`	**OUT2**	Plug (Output)	Second identical output.
`adapter::types::unidirectional::A2X`	**OUT3**	Plug (Output)	Third identical output.

The adapters are all of type **A2X** (unidirectional) and transmit data in one direction. The device is passive, meaning it does not require event control for data transmission.

Functionality

The function block receives the A2X data stream arriving via socket **IN** and forwards it unchanged to the three plugs **OUT1**, **OUT2**, and **OUT3**. No processing, filtering, or buffering takes place – the data stream is distributed 1:1 to all outputs.

Since this is a generic function block, the specific data type of the A2X adapter can be defined at runtime by typing (e.g., via the attribute `GenericClassName`).

Technical Features

- **Generic Design**: The function block is declared as a generic FB (`GenericClassName = 'GEN_A2X_SPLIT'`), allowing the A2X adapter to be parameterized with different specific data structures depending on the application.

- **No State Automation**: The function block does not have an ECC (Execution Control Chart) or event interfaces. It operates purely in a data flow-oriented manner.
- **Runtime Type Adaptation**: A unique type identifier for the generic instance can be stored via the attribute `TypeHash` (currently empty; must be set project-specifically).

State Overview

The function block has no internal states, as it contains no sequential logic or state machine. Data transmission occurs continuously without delay.

Application Scenarios

- **Signal Distribution**: An A2X signal from a sensor or controller must be forwarded in parallel to several downstream function blocks (e.g., displays, loggers, actuators).
- **Redundancy / Monitoring**: A data stream should be sent to both the actual processing unit and a diagnostic or monitoring system.
- **Prototypical Setups**: During the development phase, a single generic splitter can be flexibly used for various adapter types.

Comparison with Similar Function Blocks

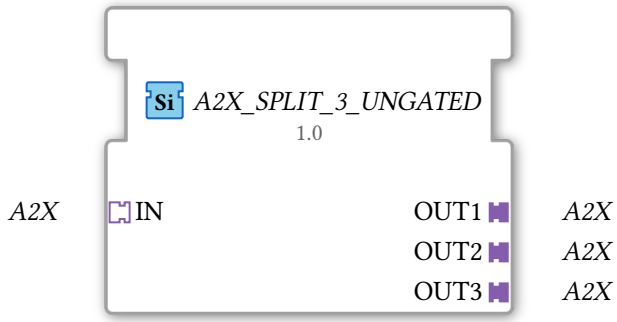
- **A2X_SPLIT_2**: Splits an A2X stream across two outputs. This function block extends this to three outputs.
- **Manual Duplication**: Without a splitter, the architect would have to reference the source adapter multiple times in the configuration, which reduces readability and maintainability.
- **Generic Splitters**: Other splitting function blocks (e.g., for data or event adapters) follow the same principle but are specialized for different adapter types.

Conclusion

The **A2X_SPLIT_3** is a simple yet useful generic function block for duplicating an A2X data stream across three paths. Its generic nature allows it to adapt flexibly to different data structures and facilitates modular interconnection in IEC 61499-based automation systems.

A2X_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `A2X_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **A2X_SPLIT_3_UNGATED** is used to split an incoming A2X adapter data stream into three identical outputs. It is implemented as a generic function block (FB) and is suitable for applications where a signal is required multiple times.

Interface Structure

Event Inputs
No event inputs available.

Event Outputs
No event outputs available.

Data Inputs
No data inputs available.

Data Outputs
No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::A2X`	**IN**	Socket (Input)	Source adapter whose data stream is split.
`adapter::types::unidirectional::A2X`	**OUT1**	Plug (Output)	First identical output.
`adapter::types::unidirectional::A2X`	**OUT2**	Plug (Output)	Second identical output.
`adapter::types::unidirectional::A2X`	**OUT3**	Plug (Output)	Third identical output.

The adapters are all of type **A2X** (unidirectional) and transmit data in one direction. The device is passive, meaning it does not require event control for data transmission.

Functionality
The function block receives the A2X data stream arriving via socket **IN** and forwards it unchanged to the three plugs **OUT1**, **OUT2**, and **OUT3**. No processing, filtering, or buffering takes place – the data stream is distributed 1:1 to all outputs.

Since this is a generic function block, the specific data type of the A2X adapter can be defined at runtime by typing (e.g., via the attribute `GenericClassName`).

Technical Features
- **Generic Design**: The function block is declared as a generic FB (`GenericClassName = 'GEN_A2X_SPLIT'`), allowing the A2X adapter to be parameterized with different specific data structures depending on the application.
- **No State Automation**: The function block does not have an ECC (Execution Control Chart) or event interfaces. It operates purely in a data flow-oriented manner.
- **Runtime Type Adaptation**: A unique type identifier for the generic instance can be stored via the attribute `TypeHash` (currently empty; must be set project-specifically).

State Overview
The function block has no internal states, as it contains no sequential logic or state machine. Data transmission occurs continuously without delay.

Application Scenarios
- **Signal Distribution**: An A2X signal from a sensor or controller must be forwarded in parallel to several downstream function blocks (e.g., displays, loggers, actuators).
- **Redundancy / Monitoring**: A data stream should be sent to both the actual processing unit and a diagnostic or monitoring system.
- **Prototypical Setups**: During the development phase, a single generic splitter can be flexibly used for various adapter types.

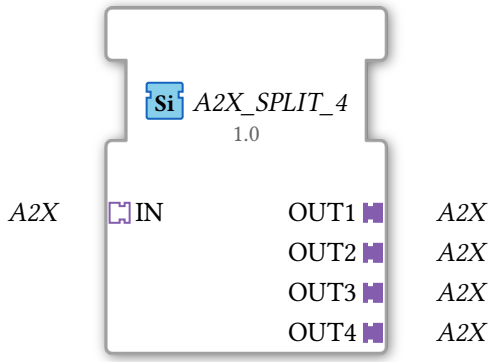
Comparison with Similar Function Blocks
- **A2X_SPLIT_2**: Splits an A2X stream across two outputs. This function block extends this to three outputs.

- **Manual Duplication**: Without a splitter, the architect would have to reference the source adapter multiple times in the configuration, which reduces readability and maintainability.
- **Generic Splitters**: Other splitting function blocks (e.g., for data or event adapters) follow the same principle but are specialized for different adapter types.

Conclusion

The **A2X_SPLIT_3_UNGATED** is a simple yet useful generic function block for duplicating an A2X data stream across three paths. Its generic nature allows it to adapt flexibly to different data structures and facilitates modular interconnection in IEC 61499-based automation systems.

A2X_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **A2X_SPLIT_4** is used to split an incoming A2X adapter signal into four identical A2X outputs. It is designed as a generic block, enabling flexible use in a wide variety of automation environments where an A2X signal is required multiple times.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Identifier	Type	Description
Socket (Input)	IN	`adapter::types::unidirectional::A2X`	One incoming A2X signal (unidirectional)
Plug (Output 1)	OUT1	`adapter::types::unidirectional::A2X`	First outgoing A2X channel
Plug (Output 2)	OUT2	`adapter::types::unidirectional::A2X`	Second outgoing A2X channel
Plug (Output 3)	OUT3	`adapter::types::unidirectional::A2X`	Third outgoing A2X channel
Plug (Output 4)	OUT4	`adapter::types::unidirectional::A2X`	Fourth Outgoing A2X Channel

Functionality

This function block operates as a pure signal distributor. The A2X adapter interface connected to socket **IN** is internally routed to the four output plugs **OUT1** to **OUT4** without any further processing or delay. Each output delivers exactly the same signal as the input. Since these are unidirectional adapters, there is no feedback from the outputs to the input.

Technical Features

- **Generic Function Block** – The function block is instantiated as `GEN_A2X_SPLIT` and can be adapted to different A2X variants using type parameters.
- **No Event or Data Processing** – The entire functionality is limited to forwarding the adapter signal. There are no events or data inputs/outputs.
- **Licensing** – This function block is licensed under the Eclipse Public License 2.0 (EPL-2.0).

State Overview

This function block has no state machines (ECCs) and no internal behavior that is time- or state-dependent. The output signals follow the input signal immediately.

Application Scenarios

- **Distribution of an A2X signal** to multiple downstream function blocks that rely on the same A2X information.
- **Star topologies** in communication between components of a distributed control system.
- **Test and simulation environments** where an A2X signal needs to be recorded or analyzed in parallel.

Comparison with Similar Function Blocks

- **A2X_SPLIT_2** / **A2X_SPLIT_3** / **A2X_SPLIT_N** – These function blocks offer the same functionality with a different number of outputs.

A2X_SPLIT_N Unlike a **multiplexer** or **demultiplexer**, no selection takes place here – the signal is always duplicated to all outputs.

Conclusion

The **A2X_SPLIT_4** is a simple yet useful component for duplicating a unidirectional A2X adapter signal. Its generic design and clear interface make it easy to integrate into existing 4diac IDE projects and contribute to the modularization of control applications.

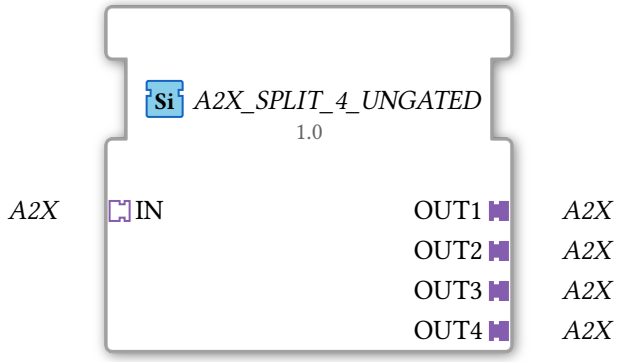
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

A2X_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `A2X_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **A2X_SPLIT_4_UNGATED** is used to split an incoming A2X adapter signal into four identical A2X outputs. It is designed as a generic block, enabling flexible use in a wide variety of automation environments where an A2X signal is required multiple times.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Identifier	Type	Description
Socket (Input)	IN	`adapter::types::unidirectional::A2X`	One incoming A2X signal (unidirectional)
Plug (Output 1)	OUT1	`adapter::types::unidirectional::A2X`	First outgoing A2X channel
Plug (Output 2)	OUT2	`adapter::types::unidirectional::A2X`	Second outgoing A2X channel
Plug (Output 3)	OUT3	`adapter::types::unidirectional::A2X`	Third outgoing A2X channel
Plug (Output 4)	OUT4	`adapter::types::unidirectional::A2X`	Fourth Outgoing A2X Channel

Functionality

This function block operates as a pure signal distributor. The A2X adapter interface connected to socket **IN** is internally routed to the four output plugs **OUT1** to **OUT4** without any further processing or delay. Each output delivers exactly the same signal as the input. Since these are unidirectional adapters, there is no feedback from the outputs to the input.

Technical Features

- **Generic Function Block** – The function block is instantiated as `GEN_A2X_SPLIT` and can be adapted to different A2X variants using type parameters.
- **No Event or Data Processing** – The entire functionality is limited to forwarding the adapter signal. There are no events or data inputs/outputs.
- **Licensing** – This function block is licensed under the Eclipse Public License 2.0 (EPL-2.0).

State Overview

This function block has no state machines (ECCs) and no internal behavior that is time- or state-dependent. The output signals follow the input signal immediately.

Application Scenarios

- **Distribution of an A2X signal** to multiple downstream function blocks that rely on the same A2X information.
- **Star topologies** in communication between components of a distributed control system.
- **Test and simulation environments** where an A2X signal needs to be recorded or analyzed in parallel.

Comparison with Similar Function Blocks

- **A2X_SPLIT_2 / A2X_SPLIT_3 / A2X_SPLIT_N** – These function blocks offer the same functionality with a different number of outputs.

A2X_SPLIT_N Unlike a **multiplexer** or **demultiplexer**, no selection takes place here – the signal is always duplicated to all outputs.

Conclusion

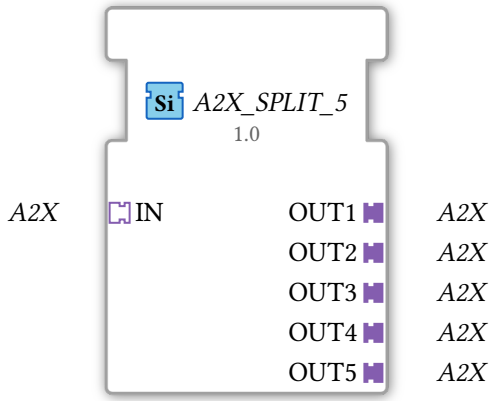
The **A2X_SPLIT_4_UNGATED** is a simple yet useful component for duplicating a unidirectional A2X adapter signal. Its generic design and clear interface make it easy to integrate into existing 4diac IDE projects and contribute to the modularization of control applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

A2X_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **A2X_SPLIT_5** is used to split an incoming A2X adapter into five identical A2X outputs. It is provided as a generic block that can be configured via attributes.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Name	Type	Description
Socket	IN	A2X (unidirectional)	Input adapter (split)
Plug	OUT1	A2X (unidirectional)	First output adapter
Plug	OUT2	A2X (unidirectional)	Second output adapter
Plug	OUT3	A2X (unidirectional)	Third output adapter
Plug	OUT4	A2X (unidirectional)	Fourth output adapter
Plug	OUT5	A2X (unidirectional)	Fifth output adapter

Functionality

The module forwards the A2X adapter input at socket `IN` unchanged and in parallel to all five plugs (`OUT1` to `OUT5`). There is no delay or buffering – any change at the input is immediately propagated to all outputs. Thus, the module behaves like a simple splitter for A2X interfaces.

Technical Features

- **Generic Module:** The actual instance can be configured via the attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`, allowing adaptation to various A2X variants.
- **No Event or Data Ports:** All communication takes place exclusively via adapters. This simplifies integration with other A2X-based components.
- **Scalability:** The module can serve as the basis for splitters with a different number of outputs (e.g., `A2X_SPLIT_2`, `A2X_SPLIT_10`).

State Overview

The module has no internal states or state machines. It operates purely combinatorially as a passive forwarder.

Application Scenarios

- **Distributing a Bus Signal:** When an A2X adapter (e.g., for a fieldbus) needs to be transmitted to multiple downstream modules without interfering with each other.
- **Test and Simulation Setups:** An incoming A2X data stream can be routed in parallel to various analysis or monitoring components.
- **Expanding Control Systems:** A sensor or actuator adapter can be split across multiple function blocks that implement different logic.

Comparison with Similar Function Blocks

- **A2X_SPLIT_2 / A2X_SPLIT_N:** These variants differ only in the number of outputs. `A2X_SPLIT_5` offers a fixed 5-way split.
- **A2X_MERGE:** While the splitter distributes one input to multiple outputs, the merge function block combines multiple A2X inputs into a single output.
- **Data splitters (e.g., SPLIT_INT):** These operate at the data level (e.g., integers) and not at the adapter level. The function block described here operates at the adapter level, which enables structural decoupling.

Conclusion

The `A2X_SPLIT_5` is a simple yet useful adapter splitter for A2X interfaces. Due to its generic nature and pure adapter communication, it integrates seamlessly into IEC 61499-based control systems. It is particularly suitable for applications where an A2X signal needs to be distributed across multiple paths without requiring additional logic or states.

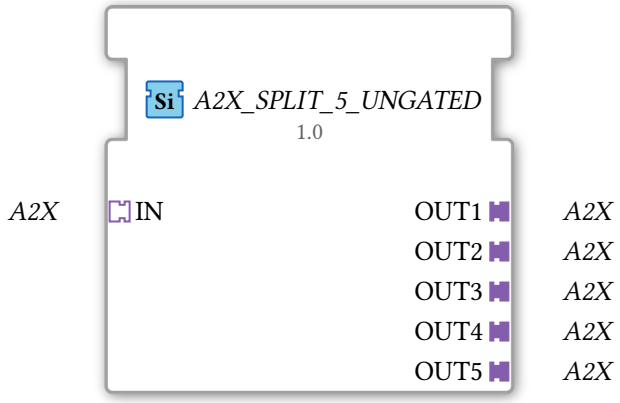
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

A2X_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `A2X_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **A2X_SPLIT_5_UNGATED** is used to split an incoming A2X adapter into five identical A2X outputs. It is provided as a generic block that can be configured via attributes.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Name	Type	Description
Socket	IN	A2X (unidirectional)	Input adapter (split)
Plug	OUT1	A2X (unidirectional)	First output adapter
Plug	OUT2	A2X (unidirectional)	Second output adapter
Plug	OUT3	A2X (unidirectional)	Third output adapter
Plug	OUT4	A2X (unidirectional)	Fourth output adapter

Plug	OUT5	A2X (unidirectional)	Fifth output adapter

Functionality

The module forwards the A2X adapter input at socket `IN` unchanged and in parallel to all five plugs (`OUT1` to `OUT5`). There is no delay or buffering – any change at the input is immediately propagated to all outputs. Thus, the module behaves like a simple splitter for A2X interfaces.

Technical Features

- **Generic Module:** The actual instance can be configured via the attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`, allowing adaptation to various A2X variants.
- **No Event or Data Ports:** All communication takes place exclusively via adapters. This simplifies integration with other A2X-based components.
- **Scalability:** The module can serve as the basis for splitters with a different number of outputs (e.g., `A2X_SPLIT_2`, `A2X_SPLIT_10`).

State Overview

The module has no internal states or state machines. It operates purely combinatorially as a passive forwarder.

Application Scenarios

- **Distributing a Bus Signal:** When an A2X adapter (e.g., for a fieldbus) needs to be transmitted to multiple downstream modules without interfering with each other.
- **Test and Simulation Setups:** An incoming A2X data stream can be routed in parallel to various analysis or monitoring components.
- **Expanding Control Systems:** A sensor or actuator adapter can be split across multiple function blocks that implement different logic.

Comparison with Similar Function Blocks

- **A2X_SPLIT_2 / A2X_SPLIT_N:** These variants differ only in the number of outputs. `A2X_SPLIT_5_UNGATED` offers a fixed 5-way split.
- **A2X_MERGE:** While the splitter distributes one input to multiple outputs, the merge function block combines multiple A2X inputs into a single output.
- **Data splitters (e.g., SPLIT_INT):** These operate at the data level (e.g., integers) and not at the adapter level. The function block described here operates at the adapter level, which enables structural decoupling.

Conclusion

The `A2X_SPLIT_5_UNGATED` is a simple yet useful adapter splitter for A2X interfaces. Due to its generic nature and pure adapter communication, it integrates seamlessly into IEC 61499-based control systems. It is particularly suitable for applications where an A2X signal needs to be distributed across multiple paths without requiring additional logic or states.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

A2X_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `A2X_SPLIT_6` is used to distribute an incoming A2X adapter to six identical A2X outputs. It is implemented as a generic function block (GFB) and enables the forwarding of adapter data without delay or state change. Typical applications include the parallel power supply of multiple devices or signal cascading in industrial control systems.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Name	Type	Description
Input (Socket)	`IN`	`adapter::types::unidirectional::A2X`	Receives the incoming A2X adapter signal
Output (Plug) 1	`OUT1`	`adapter::types::unidirectional::A2X`	First outgoing A2X adapter
Output (Plug) 2	`OUT2`	`adapter::types::unidirectional::A2X`	Second outgoing A2X adapter
Output (Plug) 3	`OUT3`	`adapter::types::unidirectional::A2X`	Third outgoing A2X adapter
Output (Plug) 4	`OUT4`	`adapter::types::unidirectional::A2X`	Fourth outgoing A2X adapter
Output (Plug) 5	`OUT5`	`adapter::types::unidirectional::A2X`	Fifth outgoing A2X adapter
Output (Plug) 6	`OUT6`	`adapter::types::unidirectional::A2X`	Sixth outgoing A2X adapter

Functionality

The module functions as a simple distributor (splitter). As soon as socket `IN` is connected to an A2X adapter, the complete adapter signal—consisting of all data and event channels contained within this

adapter—is passed on to all six plugs (`OUT1` … `OUT6`). This transmission occurs without buffering or internal logic; any change at the input is reflected instantly at all outputs.

Technical Features

- **Unidirectionality** – The adapter `A2X` is designed to be unidirectional; data flow reversal is not supported.
- **Generic Design** – The function block (FB) is stored as a generic type (`GenericClassName = 'GEN_A2X_SPLIT'`), allowing it to be used in various contexts.
- **No States or Events** – There are no internal state machines or event-driven processes; The function block (FB) is purely combinational.
- **Complete Decoupling** – The use of adapters decouples the connected blocks, increasing reusability.

State Overview

The function block does not have its own state machine. It is always passive and only active via the adapter data present at its input. Therefore, there are no defined states such as INIT, RUN, or IDLE.

Application Scenarios

- **Signal Distribution in Automation** – A sensor delivers measured values via an A2X adapter, which must be forwarded in parallel to multiple controllers or evaluation units.
- **Cascading of Adapter Signals** – In complex hierarchies, an A2X signal can be distributed across multiple subordinate modules.
- **Test and Simulation Environments** – For simultaneously connecting multiple test benches or visualization components to a single data source.

Comparison with Similar Function Blocks

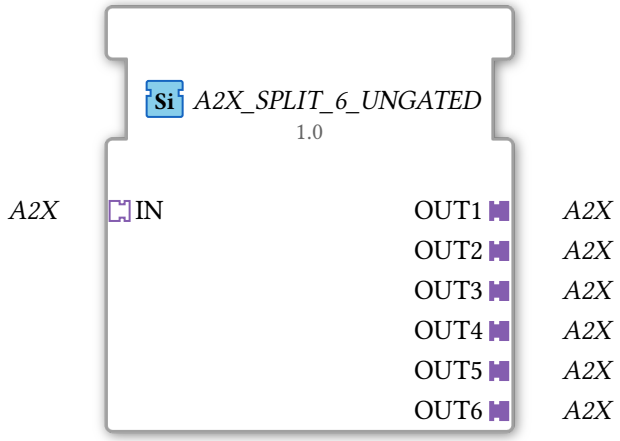
- **A2X_SPLIT_2 / A2X_SPLIT_4 / A2X_SPLIT_N** – These function blocks offer the same functionality, but with two, four, or a configurable number of outputs. `A2X_SPLIT_6` provides a fixed 6-way distribution.
- **A2X_MERGE** – Unlike the splitter, this function block combines multiple A2X inputs into a single output.
- **Standard Data Splitters** (e.g., for simple data types) – Adapter-based splitters like this one operate at a higher level of abstraction and encapsulate complex signal bundles.

Conclusion

The `A2X_SPLIT_6` is a simple yet useful function block for multiplying A2X adapter signals. Its generic structure and pure distribution function make it particularly suitable for scenarios where a signal needs to be sent to multiple receivers without any processing logic. The absence of events and states makes it easy to understand and resource-efficient to use.

A2X_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `A2X_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `A2X_SPLIT_6_UNGATED` is used to distribute an incoming A2X adapter to six identical A2X outputs. It is implemented as a generic function block (GFB) and enables the forwarding of adapter data without delay or state change. Typical applications include the parallel power supply of multiple devices or signal cascading in industrial control systems.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Name	Type	Description
Input (Socket)	`IN`	`adapter::types::unidirectional::A2X`	Receives the incoming A2X adapter signal
Output (Plug) 1	`OUT1`	`adapter::types::unidirectional::A2X`	First outgoing A2X adapter
Output (Plug) 2	`OUT2`	`adapter::types::unidirectional::A2X`	Second outgoing A2X adapter
Output (Plug) 3	`OUT3`	`adapter::types::unidirectional::A2X`	Third outgoing A2X adapter
Output (Plug) 4	`OUT4`	`adapter::types::unidirectional::A2X`	Fourth outgoing A2X adapter
Output (Plug) 5	`OUT5`	`adapter::types::unidirectional::A2X`	Fifth outgoing A2X adapter
Output (Plug) 6	`OUT6`	`adapter::types::unidirectional::A2X`	Sixth outgoing A2X adapter

Functionality

The module functions as a simple distributor (splitter). As soon as socket `IN` is connected to an A2X adapter, the complete adapter signal—consisting of all data and event channels contained within this adapter—is passed on to all six plugs (`OUT1` … `OUT6`). This transmission occurs without buffering or internal logic; any change at the input is reflected instantly at all outputs.

Technical Features

- **Unidirectionality** – The adapter `A2X` is designed to be unidirectional; data flow reversal is not supported.
- **Generic Design** – The function block (FB) is stored as a generic type (`GenericClassName = 'GEN_A2X_SPLIT'`), allowing it to be used in various contexts.
- **No States or Events** – There are no internal state machines or event-driven processes; The function block (FB) is purely combinational.
- **Complete Decoupling** – The use of adapters decouples the connected blocks, increasing reusability.

State Overview

The function block does not have its own state machine. It is always passive and only active via the adapter data present at its input. Therefore, there are no defined states such as INIT, RUN, or IDLE.

Application Scenarios

- **Signal Distribution in Automation** – A sensor delivers measured values via an A2X adapter, which must be forwarded in parallel to multiple controllers or evaluation units.
- **Cascading of Adapter Signals** – In complex hierarchies, an A2X signal can be distributed across multiple subordinate modules.
- **Test and Simulation Environments** – For simultaneously connecting multiple test benches or visualization components to a single data source.

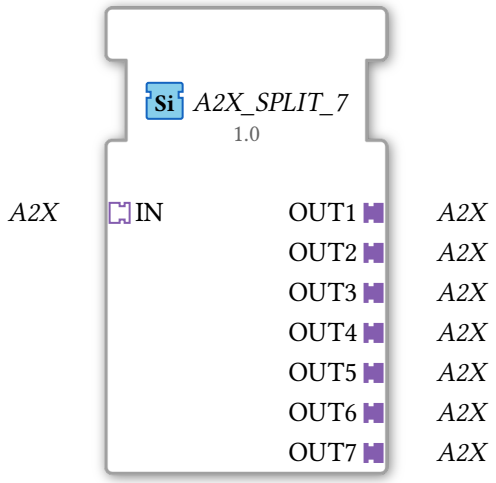
Comparison with Similar Function Blocks

- **A2X_SPLIT_2 / A2X_SPLIT_4 / A2X_SPLIT_N** – These function blocks offer the same functionality, but with two, four, or a configurable number of outputs. `A2X_SPLIT_6_UNGATED` provides a fixed 6-way distribution.
- **A2X_MERGE** – Unlike the splitter, this function block combines multiple A2X inputs into a single output.
- **Standard Data Splitters** (e.g., for simple data types) – Adapter-based splitters like this one operate at a higher level of abstraction and encapsulate complex signal bundles.

Conclusion

The `A2X_SPLIT_6_UNGATED` is a simple yet useful function block for multiplying A2X adapter signals. Its generic structure and pure distribution function make it particularly suitable for scenarios where a signal needs to be sent to multiple receivers without any processing logic. The absence of events and states makes it easy to understand and resource-efficient to use.

A2X_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **A2X_SPLIT_7** is used to distribute an incoming A2X adapter signal to seven identical outputs. It is provided as a generic FB and enables efficient signal multiplication within 4diac IDE projects.

Interface Structure

Event Inputs
- None

Event Outputs
- None

Data Inputs
- None

Data Outputs
- None

Adapter

Type	Name	Description
A2X (Socket)	**IN**	Input adapter for the A2X signal to be distributed
A2X (Plug)	**OUT1** ... **OUT7**	Seven output adapters, each providing an identical copy of the input signal

Functionality
The function block forwards the A2X signal present at the **IN** adapter unchanged to all seven output adapters (**OUT1** to **OUT7**). No data manipulation takes place – the signals are simply passively duplicated. The function block is purely adapter-based and requires neither event nor data inputs.

Technical Features
- **Generic Function Block**: The function block is declared as a generic type (`GEN_A2X_SPLIT`) and can be reused in various contexts.
- **Unidirectional Adapters**: All A2X adapters used are of type `unidirectional`, meaning data flows in only one direction (from the socket to the plug).
- **No Event Control**: Signal transmission is purely data flow-driven without additional events or states.
- **Easily Extensible**: Any number of outputs (e.g., 2, 4, 10) can be implemented by modifying the XML definition.

State Overview
The function block (FB) does not have its own state machine (ECC). The output signals are always an exact representation of the input signal. There are no internal states or time delays.

Application Scenarios
- **Distributing a sensor or control signal** to multiple parallel function blocks or subsystems.
- **Providing identical adapter data** for redundant processing paths (e.g., backups).
- **Simulation or test setups** where a signal is needed multiple times without instantiating the source multiple times.

Comparison with similar function blocks
- **A2X_SPLIT_2, A2X_SPLIT_3, ...**: These function blocks distribute the input signal to two, three, or more outputs. This function block is specifically designed for seven outputs and can be considered a specialization of a generic split function block.
- **Manual multiplication via connection**: Theoretically, an A2X signal could also be distributed by connecting an output multiple times – however, the split function block provides a clean, structured, and reusable solution.

Conclusion
The **A2X_SPLIT_7** is a simple yet useful function block for signal multiplication in A2X-based adapter systems. Its generic nature and clear interface make it ideally suited for modular automation solutions using the 4diac IDE.

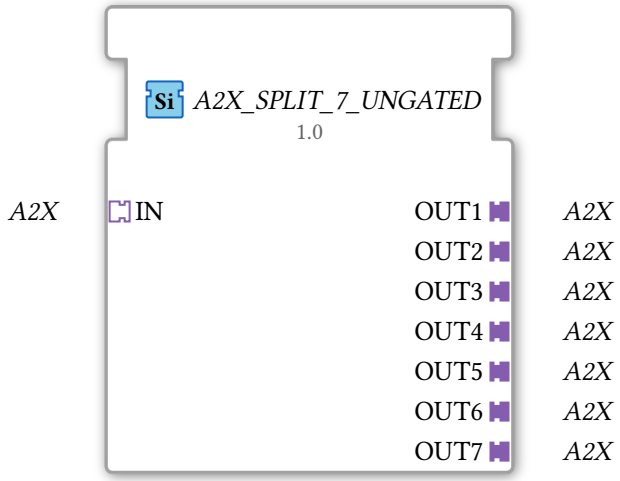
Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

A2X_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `A2X_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **A2X_SPLIT_7_UNGATED** is used to distribute an incoming A2X adapter signal to seven identical outputs. It is provided as a generic FB and enables efficient signal multiplication within 4diac IDE projects.

Interface Structure

Event Inputs

- None

Event Outputs

- None

Data Inputs

- None

Data Outputs

- None

Adapter

Type	Name	Description
A2X (Socket)	**IN**	Input adapter for the A2X signal to be distributed
A2X (Plug)	**OUT1** … **OUT7**	Seven output adapters, each providing an identical copy of the input signal

Functionality

The function block forwards the A2X signal present at the **IN** adapter unchanged to all seven output adapters (**OUT1** to **OUT7**). No data manipulation takes place – the signals are simply passively duplicated. The function block is purely adapter-based and requires neither event nor data inputs.

Technical Features

- **Generic Function Block**: The function block is declared as a generic type (`GEN_A2X_SPLIT`) and can be reused in various contexts.
- **Unidirectional Adapters**: All A2X adapters used are of type `unidirectional`, meaning data flows in only one direction (from the socket to the plug).
- **No Event Control**: Signal transmission is purely data flow-driven without additional events or states.
- **Easily Extensible**: Any number of outputs (e.g., 2, 4, 10) can be implemented by modifying the XML definition.

State Overview

The function block (FB) does not have its own state machine (ECC). The output signals are always an exact representation of the input signal. There are no internal states or time delays.

Application Scenarios

- **Distributing a sensor or control signal** to multiple parallel function blocks or subsystems.
- **Providing identical adapter data** for redundant processing paths (e.g., backups).
- **Simulation or test setups** where a signal is needed multiple times without instantiating the source multiple times.

Comparison with similar function blocks

- **A2X_SPLIT_2, A2X_SPLIT_3, ...**: These function blocks distribute the input signal to two, three, or more outputs. This function block is specifically designed for seven outputs and can be considered a specialization of a generic split function block.
- **Manual multiplication via connection**: Theoretically, an A2X signal could also be distributed by connecting an output multiple times – however, the split function block provides a clean, structured, and reusable solution.

Conclusion

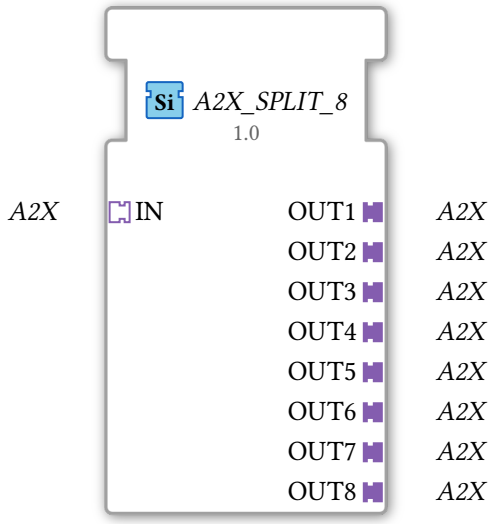
The **A2X_SPLIT_7_UNGATED** is a simple yet useful function block for signal multiplication in A2X-based adapter systems. Its generic nature and clear interface make it ideally suited for modular automation solutions using the 4diac IDE.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

A2X_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The A2X_SPLIT_8 function block serves as a generic building block to distribute an incoming A2X adapter (socket) to eight identical A2X adapter outputs (plugs). It is used in industrial automation when a signal is required multiple times.

Interface Structure

Event Inputs

Not present.

Event Outputs

Not present.

Data Inputs

Not present.

Data Outputs

Not present.

Adapters

The FB has the following adapter interfaces:
Socket (Input):

- `IN` (Type: `adapter::types::unidirectional::A2X`)

Plugs (Outputs):

- `OUT1` to `OUT8` (each type: `adapter::types::unidirectional::A2X`)

Functionality

The FB forwards the data received via socket `IN` unchanged to all eight plugs `OUT1` to `OUT8`. No processing or conversion of the data takes place. The FB functions purely as a distributor (splitter) for A2X adapters.

Technical Features

- The function block is implemented as a generic FB (GenericClassName: `GEN_A2X_SPLIT`), enabling flexible reuse in various contexts.
- There are no events or data points; all communication takes place via the A2X adapters.
- The FB is compliant with the IEC 61499-2 standard.
- The outputs are independent of each other and can be individually connected to other FBs.

State Overview

Since the FB has neither event inputs nor an execution state, there is no state machine. The FB operates continuously without an internal state.

Application Scenarios

- Distribution of a sensor signal (e.g., temperature, pressure) to multiple consumers or controllers.
- Parallel connection of a control signal to multiple actuators.
- Provision of a reference value for multiple control loops.

Comparison with Similar Function Blocks

- **A2X_SPLIT_2**, **A2X_SPLIT_4**: Function blocks with fewer outputs that can be used as needed.
- **A2X_MERGE**: Combines multiple A2X inputs into one output – the opposite function.
- **A2X_SELECT**: Selects one of several A2X inputs, while the splitter operates all outputs simultaneously.

Conclusion

The A2X_SPLIT_8 is a simple yet useful function block for multiplying A2X signals in automation projects. Its generic design and clear separation of inputs and outputs provide a clean and reusable solution.

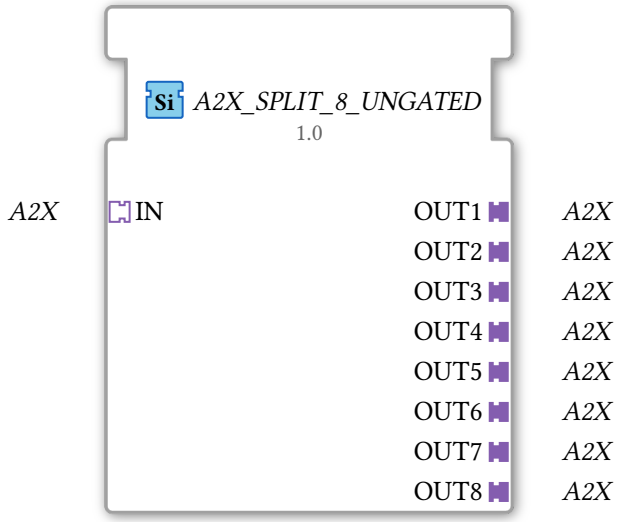
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

]

A2X_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `A2X_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The A2X_SPLIT_8_UNGATED function block serves as a generic building block to distribute an incoming A2X adapter (socket) to eight identical A2X adapter outputs (plugs). It is used in industrial automation when a signal is required multiple times.

Interface Structure

Event Inputs
Not present.

Event Outputs
Not present.

Data Inputs
Not present.

Data Outputs
Not present.

Adapters
The FB has the following adapter interfaces:
Socket (Input):

- `IN` (Type: `adapter::types::unidirectional::A2X`)

Plugs (Outputs):

- `OUT1` to `OUT8` (each type: `adapter::types::unidirectional::A2X`)

Functionality
The FB forwards the data received via socket `IN` unchanged to all eight plugs `OUT1` to `OUT8`. No processing or conversion of the data takes place. The FB functions purely as a distributor (splitter) for A2X adapters.

Technical Features
- The function block is implemented as a generic FB (GenericClassName: `GEN_A2X_SPLIT`), enabling flexible reuse in various contexts.
- There are no events or data points; all communication takes place via the A2X adapters.
- The FB is compliant with the IEC 61499-2 standard.
- The outputs are independent of each other and can be individually connected to other FBs.

State Overview
Since the FB has neither event inputs nor an execution state, there is no state machine. The FB operates continuously without an internal state.

Application Scenarios
- Distribution of a sensor signal (e.g., temperature, pressure) to multiple consumers or controllers.
- Parallel connection of a control signal to multiple actuators.
- Provision of a reference value for multiple control loops.

Comparison with Similar Function Blocks
- **A2X_SPLIT_2**, **A2X_SPLIT_4**: Function blocks with fewer outputs that can be used as needed.
- **A2X_MERGE**: Combines multiple A2X inputs into one output – the opposite function.
- **A2X_SELECT**: Selects one of several A2X inputs, while the splitter operates all outputs simultaneously.

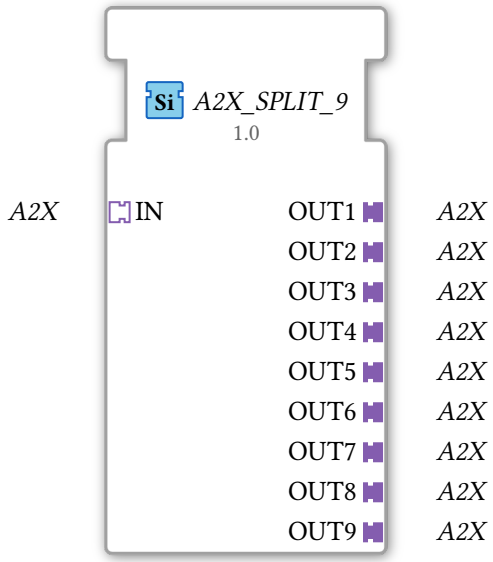
Conclusion
The A2X_SPLIT_8_UNGATED is a simple yet useful function block for multiplying A2X signals in automation projects. Its generic design and clear separation of inputs and outputs provide a clean and reusable solution.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

]

A2X_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The **A2X_SPLIT_9** is a generic function block that splits one incoming A2X adapter (socket) into nine separate A2X output adapters (plugs). It enables the star topology distribution of an A2X signal to up to nine independent receivers without requiring any dedicated logic or state machines.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Type	Name	Direction	Description
Socket	IN	Input	Incoming A2X adapter, whose signals are distributed to the nine outputs.
Plug	OUT1 ... OUT9	Output	Nine identical A2X output adapters that forward the input data.

Functionality

This function block has no internal logic or state machines. It functions purely as a “wiring aid” in the 4diac IDE: The A2X adapter connected to socket **IN** is passed unchanged and in parallel to all nine plugs **OUT1** to **OUT9**. Thus, all connected receivers receive identical signals to the input.

Technical Features

- **Generic Function Block**: The function block is declared as generic (attribute `GenericClassName = 'GEN_A2X_SPLIT'`). It can be parameterized for various specific A2X adapter types in the IDE.

- **No Runtime Logic**: The function block contains no ECC (Execution Control Chart) and no algorithms – all functionality is implemented through the internal connection of the adapters.
- **Unidirectional Adapters**: The adapters used, `adapter::types::unidirectional::A2X`, only allow data flow in one direction (from the socket to the plugs).

State Overview

A state overview is not required, as the function block has no internal state. Data flow is continuous and uncontrolled by events.

Application Scenarios

- **Signal Distribution in Automation**: Distributing an analog or binary A2X signal to multiple actuators or higher-level controllers.
- **Test and Simulation Setups**: Feeding a reference signal into multiple parallel test paths.
- **Topology Implementation**: Star-shaped cabling in the graphical configuration to utilize a central data point multiple times.

Comparison with Similar Function Blocks

- **A2X_SPLIT_N**: A configurable splitter with a variable number of outputs, e.g., 2, 4, or 8. The **A2X_SPLIT_9** is fixed to nine outputs.
- **A2X_MERGE**: The reverse function block that combines multiple inputs into one output (e.g., for bus topologies).
- **Event-Based Splitters**: Function blocks like `SPLIT` or `E_SPLIT` operate with event signals and require control logic – the **A2X_SPLIT_9** does not require events.

Conclusion

The **A2X_SPLIT_9** is a simple yet effective function block for distributing an A2X signal to nine identical outputs. Its generic nature and lack of internal logic make it lightweight and ideal for pure wiring scenarios in the 4diac IDE. It is particularly well-suited for applications where a signal is needed multiple times without requiring additional programming.

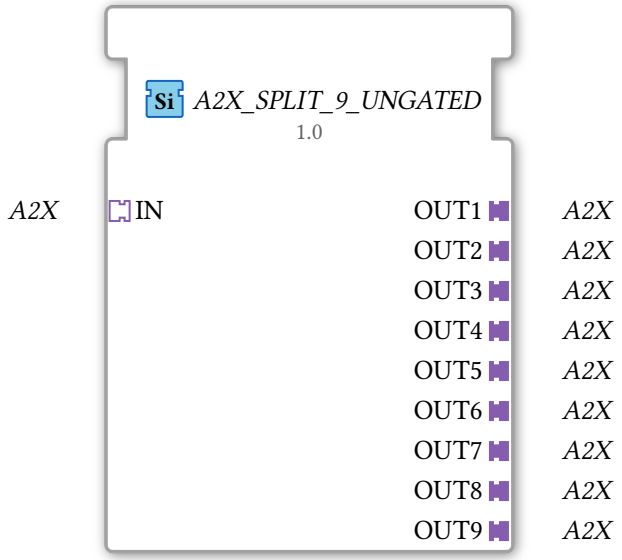
🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

A2X_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `A2X_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **A2X_SPLIT_9_UNGATED** is a generic function block that splits one incoming A2X adapter (socket) into nine separate A2X output adapters (plugs). It enables the star topology distribution of an A2X signal to up to nine independent receivers without requiring any dedicated logic or state machines.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Type	Name	Direction	Description
Socket	IN	Input	Incoming A2X adapter, whose signals are distributed to the nine outputs.
Plug	OUT1 ... OUT9	Output	Nine identical A2X output adapters that forward the input data.

Functionality

This function block has no internal logic or state machines. It functions purely as a “wiring aid” in the 4diac IDE: The A2X adapter connected to socket **IN** is passed unchanged and in parallel to all nine plugs **OUT1** to **OUT9**. Thus, all connected receivers receive identical signals to the input.

Technical Features

- **Generic Function Block**: The function block is declared as generic (attribute `GenericClassName = 'GEN_A2X_SPLIT'`). It can be parameterized for various specific A2X adapter types in the IDE.

- **No Runtime Logic**: The function block contains no ECC (Execution Control Chart) and no algorithms – all functionality is implemented through the internal connection of the adapters.
- **Unidirectional Adapters**: The adapters used, `adapter::types::unidirectional::A2X`, only allow data flow in one direction (from the socket to the plugs).

State Overview

A state overview is not required, as the function block has no internal state. Data flow is continuous and uncontrolled by events.

Application Scenarios

- **Signal Distribution in Automation**: Distributing an analog or binary A2X signal to multiple actuators or higher-level controllers.
- **Test and Simulation Setups**: Feeding a reference signal into multiple parallel test paths.
- **Topology Implementation**: Star-shaped cabling in the graphical configuration to utilize a central data point multiple times.

Comparison with Similar Function Blocks

- **A2X_SPLIT_N**: A configurable splitter with a variable number of outputs, e.g., 2, 4, or 8. The **A2X_SPLIT_9_UNGATED** is fixed to nine outputs.
- **A2X_MERGE**: The reverse function block that combines multiple inputs into one output (e.g., for bus topologies).
- **Event-Based Splitters**: Function blocks like `SPLIT` or `E_SPLIT` operate with event signals and require control logic – the **A2X_SPLIT_9_UNGATED** does not require events.

Conclusion

The **A2X_SPLIT_9_UNGATED** is a simple yet effective function block for distributing an A2X signal to nine identical outputs. Its generic nature and lack of internal logic make it lightweight and ideal for pure wiring scenarios in the 4diac IDE. It is particularly well-suited for applications where a signal is needed multiple times without requiring additional programming.

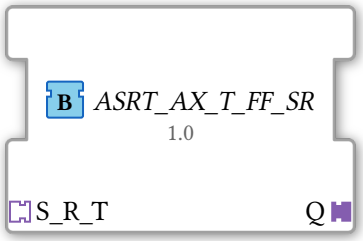
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

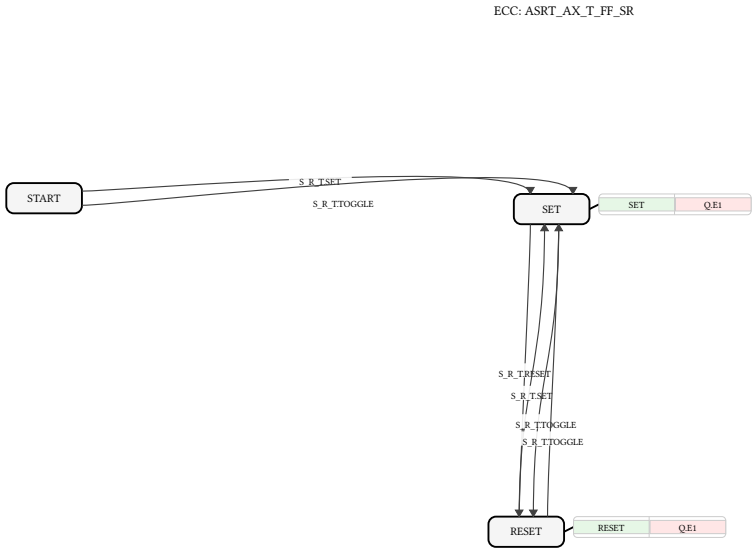
]

ASRT_AX_T_FF_SR

Set/Reset/Toggle – *ASRT*



AX – value of flip flop



• – ○ • – ○ • – ○ •

Introduction

The ASRT_AX_T_FF_SR is an event-driven bistable function block with toggle functionality – the direct counterpart of AX_T_FF_SR, where the three event inputs S/R/CLK are combined into a single unidirectional **ASRT socket** (Set/Reset/Toggle) instead of being routed as three separate event inputs.

Interface Structure

Event Inputs

- No direct event inputs available – Set/Reset/Toggle arrive bundled via the adapter socket `S_R_T`

Event Outputs

- No direct event outputs available

Data Inputs

- No data inputs available

Data Outputs

- No direct data outputs available

Adapters

- **S_R_T**: Unidirectional adapter socket of type `adapter::types::unidirectional::ASRT` (Set/Reset/Toggle)
- **Q**: Unidirectional adapter plug of type `adapter::types::unidirectional::AX` that provides the flip-flop value

Operation

The ASRT_AX_T_FF_SR has three operating states:

- **START**: Initial state
- **SET**: Output Q is TRUE
- **RESET**: Output Q is FALSE

The state transitions are controlled by the adapter events arriving at socket `S_R_T`:

- `S_R_T.SET` transitions from any state to the SET state
- `S_R_T.RESET` transitions from any state to the RESET state
- `S_R_T.TOGGLE` toggles the current state (SET ⟶ RESET or RESET ⟶ SET, and from START to SET)

With each state change, the corresponding algorithm is executed, setting the adapter value `Q.D1` accordingly (`TRUE` in the SET state, `FALSE` in the RESET state).

Technical Features

- Combines SR flip-flop and T flip-flop functionality like AX_T_FF_SR, but with a bundled adapter interface instead of three separate event inputs
- Uses the unidirectional ASRT adapter interface for Set/Reset/Toggle and the AX adapter interface for data output
- Initial state is START, from which a transition to SET can occur directly or via TOGGLE

State Overview

```
START (initial state)
     │
     ├── S_R_T.SET ─────→ SET (Q.D1 = TRUE)
     │
     └── S_R_T.TOGGLE ─→ SET (Q.D1 = TRUE)

SET (Q.D1 = TRUE)
     │
     ├── S_R_T.RESET ──→ RESET (Q.D1 = FALSE)
     │
     └── S_R_T.TOGGLE ─→ RESET (Q.D1 = FALSE)

RESET (Q.D1 = FALSE)
     │
     ├── S_R_T.SET ─────→ SET (Q.D1 = TRUE)
     │
     └── S_R_T.TOGGLE ─→ SET (Q.D1 = TRUE)
```

Application Scenarios

- State storage in control applications where Set/Reset/Toggle are already bundled via an ASRT adapter (e.g. from ASRT_CLIENT_0 or ASRT_3EVENTS_TO_SRT)
- Clock and frequency division
- State machines with memory function in fully adapter-based applications

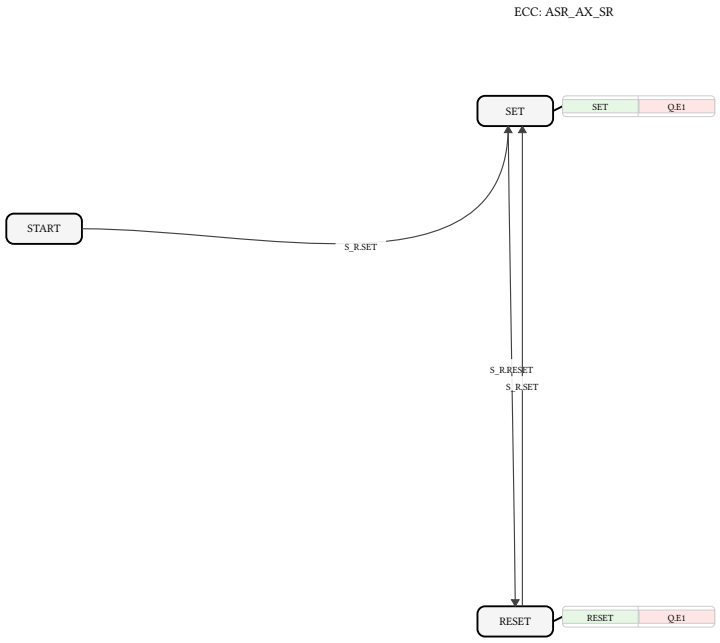
⚖️ Comparison with similar components

Compared to AX_T_FF_SR, the ASRT_AX_T_FF_SR offers the same functionality, but with a single ASRT adapter socket instead of three separate event inputs (S/R/CLK) – allowing it to be connected directly to an ASRT-providing block without wiring the events individually.

Conclusion

The ASRT_AX_T_FF_SR is the adapter-based variant of the proven SR/T flip-flop pattern from AX_T_FF_SR. By bundling Set/Reset/Toggle into a single ASRT socket, it is particularly suited to fully adapter-based control architectures.

ASR_AX_SR

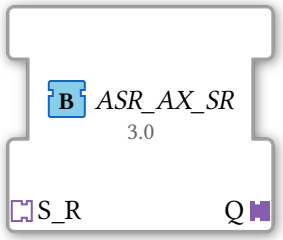


• – ○ • – ○ • – ○ •

Introduction

The ASR_AX_SR is an event-driven bistable function block (flip-flop) that operates according to the set-reset principle. It is used to store a binary state and is controlled via adapter interfaces.

Set/Reset – *ASR*



AX – value of flip flop

Interface Structure

Event Inputs

No direct event inputs are available – control is exclusively via adapters.

Event Outputs

No direct event outputs are available – output is exclusively via adapters.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

- **S_R (Socket)**: Set/Reset control adapter of type `adapter::types::unidirectional::ASR`
- **Q (Plug)**: Output adapter of type `adapter::types::unidirectional::AX` for the flip-flop state

Functionality

The ASR_AX_SR operates as a set/reset flip-flop with three states:

- **START**: Initial state
- **SET**: Set state (Q = TRUE)
- **RESET**: Reset state (Q = FALSE)

Upon a SET event via the S_R adapter, the block switches to the SET state and sets the output Q to TRUE. Upon a RESET event, it switches to the RESET state and sets Q to FALSE.

Technical Features

- **No Prioritization (Dominance)**: Since setting and resetting are triggered by separate events (`S_R.SET`, `S_R.RESET`), there is no simultaneous dominance as in IEC 61131-3. The state is determined by the last event received ("Last Event Wins").
- **Use of Adapters**: Uses unidirectional adapters for input and output (`ASR` for set/reset, `AX` for output).
- **Implementation**: Implemented as a Basic Function Block with an explicit state machine (ECC).
- **Interface**: No direct inputs/outputs; communication is exclusively adapter-based.

State Transitions

START ⟶ SET: bei S_R.SET Ereignis SET ⟶ RESET: bei S_R.RESET Ereignis RESET ⟶ SET: bei S_R.SET Ereignis

Algorithms

- **SET**: Sets the output value Q.D1 to TRUE
- **RESET**: Sets the output value Q.D1 to FALSE

Application Scenarios

- Storage of switching states in control applications
- State storage in sequential control systems
- Flip-flop functionality in distributed automation systems

⚖️ Comparison with similar components

- **SR (IEC 61131-3)**: The classic SR component has a defined set dominance for simultaneous signals. In contrast, the `ASR_AX_SR` behaves in a time-dependent manner (last event counts).
- **E_SR**: Functionally similar (event-driven), but with direct event pins instead of adapters.
- **Conventional Flip-Flops**: Compared to flip-flops with direct inputs/outputs, the adapter design facilitates integration into modular system architectures.

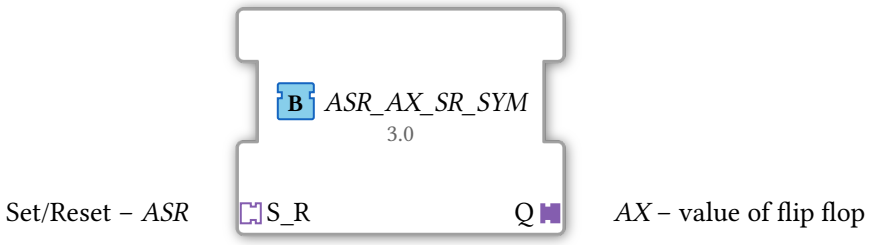
🛠️ Related Exercises

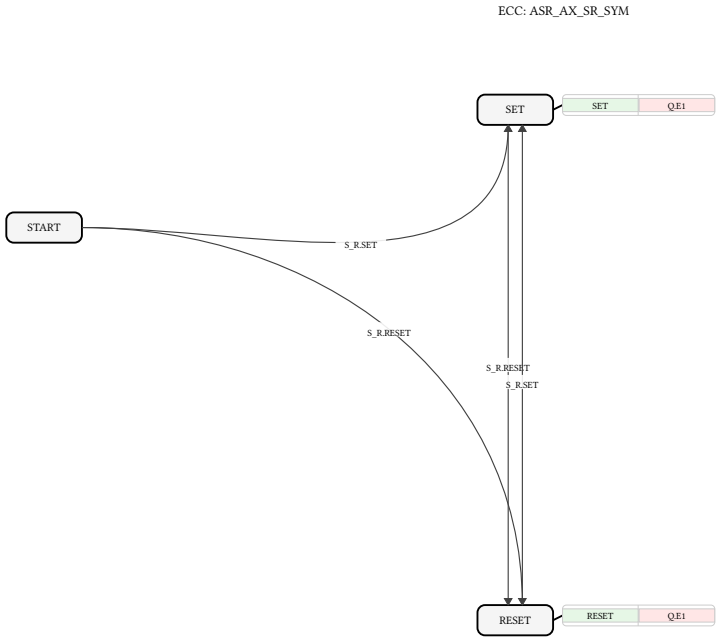
- Exercise_171_AX

Conclusion

The ASR_AX_SR offers a clean, adapter-based implementation of a bistable memory element, ideally suited for use in modular IEC 61499 systems. The exclusive use of adapters allows for high flexibility in system integration.

ASR_AX_SR_SYM





• – ○ • – ○ • – ○ •

Introduction

The function block **ASR_AX_SR_SYM** implements an event-driven, bistable flip-flop (SR flip-flop) with symmetric start behavior. It is implemented as a Basic FB according to IEC 61499-1 (Annex A) and operates exclusively via adapter interfaces – without direct event or data inputs. The block is suitable for applications where the initial state is determined by the first arriving event and neither input has priority.

Interface Structure

Event Inputs

The FB has no direct event inputs. All events are received via the **socket adapter `S_R`** (type `adapter::types::unidirectional::ASR`). This adapter provides the following event inputs:

- `S_R.SET` – Set event
- `S_R.RESET` – Reset event

Event Outputs

The FB has no direct event outputs. Outgoing events are sent via the **plug adapter `Q`** (type `adapter::types::unidirectional::AX`). This adapter provides the following event output:

- `Q.E1` – Output event, triggered by a state change

Data Inputs

No direct data inputs.

Data Outputs

The output value is provided via the plug adapter `Q`:

- `Q.D1` (BOOL) – current state of the flip-flop (`TRUE` = set, `FALSE` = reset)

Adapter

Name	Type	Direction	Description
`S_R`	`adapter::types::unidirectional::ASR`	Socket	Set/Reset Events (Input)

Q	adapter::types::unidirectional::AX	Plug	State Output (Event + Data)

Functionality

After initialization, the module starts in the **START** state. In this state, it waits for the first arriving event at S_R.SET or S_R.RESET. Depending on which event arrives first, it switches to the **SET** or **RESET** state:

- **SET State**:

The data output Q.D1 is set to TRUE, and the event Q.E1 is triggered.
Upon a subsequent S_R.RESET event, the module transitions to the RESET state.
... - **RESET State**:
The data output Q.D1 is set to FALSE, triggering the event Q.E1.
Upon a subsequent event, S_R.SET, the transition to the SET state occurs.
After the first transition, the function block remains in one of the two stable states until the complementary event occurs. There is no precedence between SET and RESET – the first event at startup determines the initial state.

Technical Features

- **Symmetrical Startup Behavior**: Unlike typical SR flip-flops, neither input has priority. The initial state is determined by the first event.
- **Pure Adapter Interface**: Both events and data are exchanged via IEC 61499 adapters, not via direct input/output ports. This simplifies reuse in complex, adapter-based systems.
- **No explicit event I/O**: The function block (FB) does not have its own event inputs/outputs and instead uses the integrated event ports of the adapters.
- **Compact ST implementation**: The two algorithms SET and RESET simply set the data output and trigger the output event.

State Overview

(Note: A graphical state diagram can be extracted from the ECC XML, but is not embedded here as an image.)

State	Description	Action	Output
START	Initial wait state	–	No output
SET	Stored state “set”	Q.D1 := TRUE	Triggering Q.E1
RESET	Saved state “reset”	Q.D1 := FALSE	Triggering Q.E1

Transitions:

- START ⟶ SET: when S_R.SET arrives
- START ⟶ RESET: when S_R.RESET arrives
- SET ⟶ RESET: when S_R.RESET arrives (after the first set)
- RESET ⟶ SET: when S_R.SET arrives (after the first reset)

Application Scenarios

- **Binary State Memory**: Storing an on/off state controlled by two separate events (e.g., “Start” and “Stop”).
- **Controlling Actuators with Set and Reset Inputs**: Used in controllers where an output needs to be set and reset symmetrically without prioritization.
- **First-Event Initialization**: Systems where the start state is not predefined but determined by the first command (set or reset).

Comparison with Similar Function Blocks

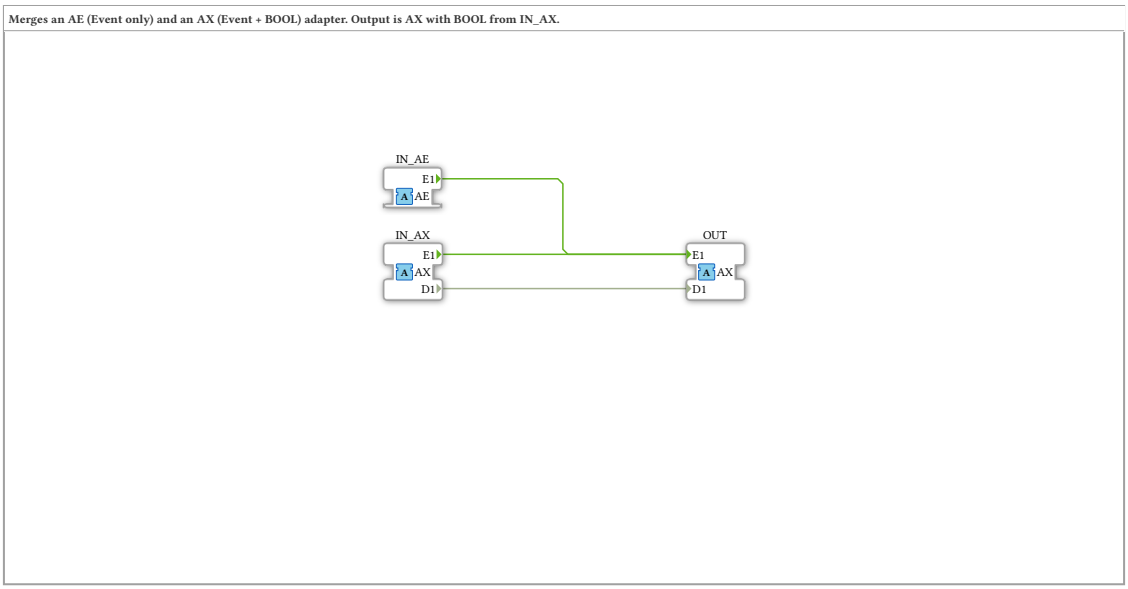
- **Standard SR Flip-Flop (e.g., SR classic)**: Often, one input is dominant (e.g., reset takes precedence). The ASR_AX_SR_SYM treats both inputs equally – at startup, the first event determines the state.
- **RS Flip-Flop with Prioritized Set**: Other implementations prioritize the set input.

This function block is characterized by its symmetrical handling of both events and its exclusive adapter interface, making it flexible for use in adapter-based libraries.

Conclusion

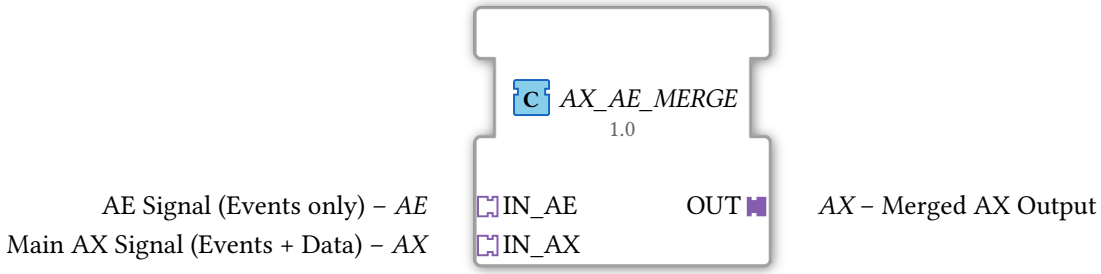
The ASR_AX_SR_SYM function block offers a reliable, event-driven SR flip-flop with symmetric start behavior. Its adapter interfaces allow for seamless integration into IEC 61499 systems, making it particularly suitable for applications requiring equal set/reset control without precedence. Its simple state logic and small number of algorithms make it efficient and easy to understand.

AX_AE_MERGE



Introduction

The AX_AE_MERGE function block allows you to merge a pure event signal (**AE**) with a combined event-data signal (**AX**). The result is an AX signal flow where the data bit from the AX branch is preserved but can be triggered by either source. This is particularly useful when a cyclic event (e.g., from a AE_CYCLE) needs to force a status check or action in an AX-based data stream without modifying the data itself.



Interface Structure

Adapter

- **IN_AX** (Socket): Primary input (event E1 + data D1).
- **IN_AE** (Socket): Additional event input (only event E1).
- **OUT** (Plug): Merged output (event from both inputs, data only from IN_AX).

Functionality

When an event arrives at IN_AX, it is passed to the output along with the current value of D1. When an event arrives at IN_AE, an event is also triggered at the output, with the *last known* value of IN_AX.D1 present at the output.

⚖️ Comparison with similar building blocks

In many exercises (e.g., with the blinker in Uebung_007a3_AX), a AX_OR_2 was incorrectly used to feed a AE event (from AE_CYCLE) into a AX data stream.

- **AX_OR_2**: Expects two fully functional AX adapters. Connecting an AE adapter is technically incorrect, as the data value for one of the inputs remains undefined.
- **AX_AE_MERGE**: Is designed precisely for this mixed operation. It ensures that the event is passed through while the data source (`IN_AX`) remains uniquely defined.

AX_OR_2: Expects two fully functional AX adapters. Connecting an AE adapter is technically incorrect, as the data value for one of the inputs remains undefined.

Technical Features

✓ **Event Merge**: Internally uses a `E_MERGE`.
✓ **Data Preservation**: No data is modified, only additional triggering occurs.
✓ **Clean Typing**: Prevents the direct connection of dissimilar adapter types.

Application Scenarios

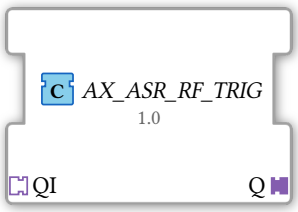
- **Blinker Logic**: A `AE_CYCLE` regularly triggers the transmission of the current state of a `AX_SR` flip-flop.
- **Status Refresh**: A timer regularly triggers the transmission of the current AX state.
- **Multiple Triggering**: An AX signal flow should also be triggered by external AE events (e.g., button clicks).

🛠️ Related exercises

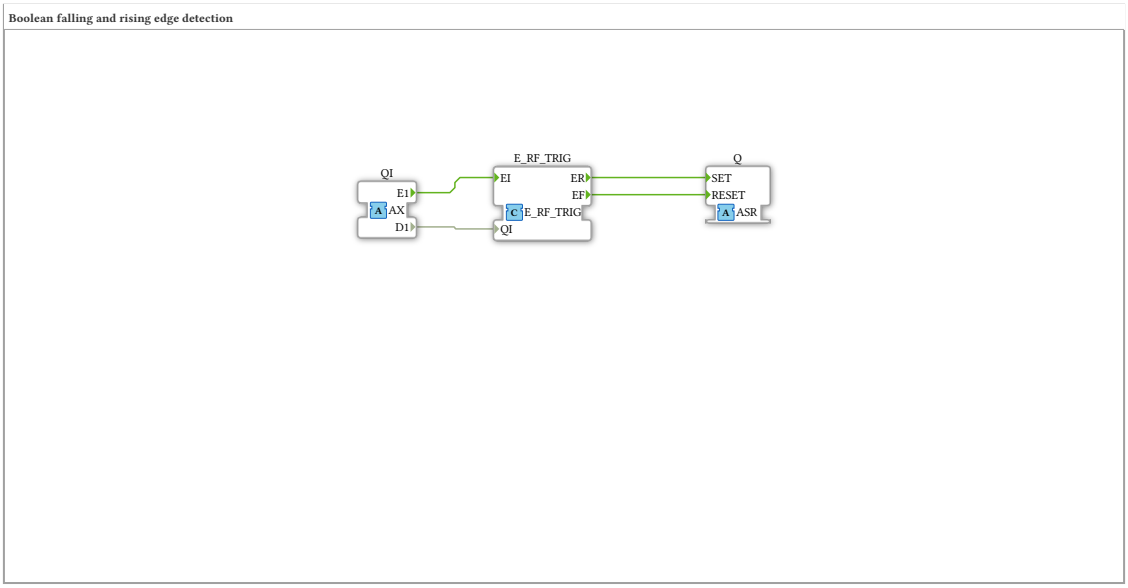
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_009_AX

AX_ASR_RF_TRIG

value to check for a rising or falling edge – *AX*



ASR – Set/Reset Output



• – ○ • – ○ • – ○ •

Introduction

The function block **AX_ASR_RF_TRIG** is used for bidirectional edge detection (rising and falling) of a Boolean input signal. It is designed as an adapter-based envelope of the IEC 61499 standard function block `E_RF_TRIG`. The signal to be monitored is read via two unidirectional adapters, and the result is provided as a set/reset output.

Interface Structure

The function block has **no direct** event or data inputs/outputs. All communication takes place via two adapters.

Event Inputs

- **none**

Event Outputs

- **none**

Data Inputs

- **none**

Data Outputs

- **none**

Adapters

Adapter	Type	Direction	Description
QI	`adapter::types::unidirectional::AX`	Socket	Input adapter – provides the Boolean value to be monitored (via channel `D1`) as well as an event (`E1`) for processing.
Q	`adapter::types::unidirectional::ASR`	Plug	Output adapter – signals the detected edge state via the events `SET` (rising edge) and `RESET` (falling edge).

Functionality

1. An event received at socket **QI** (via `QI.E1`) simultaneously transmits the Boolean value `QI.D1` to the internal `E_RF_TRIG` module.
2. The integrated `E_RF_TRIG` detects a **rising edge** (transition from FALSE to TRUE) and a **falling edge** (transition from TRUE to FALSE).
3. On a rising edge, the internal function block (FB) generates the event `ER`, which is output via the **Q** plug as `Q.SET`.
4. On a falling edge, the internal FB generates the event `EF`, which is output as `Q.RESET`.

Thus, the input signal is converted into a set/reset signal without the user having to deal with the internal event processing.

Technical Features

- **Pure Adapter Interface:** The FB has no direct inputs/outputs in the FB type interface, which enables loose coupling and reusability in different adapter networks.
- **Use of the standard function block `E_RF_TRIG`:** Edge detection is based on the proven IEC 61499 implementation, which reliably detects both rising and falling edges.
- **No state storage outside the internal function block:** All logic is implemented by the embedded `E_RF_TRIG`; the external function block serves solely as encapsulation and adapter adaptation.

State Overview

The function block does **not have its own explicit state machine**. The internal `E_RF_TRIG` operates according to the following implicit state logic:

last value	current value	triggered event
FALSE	TRUE	rising edge (SET)
TRUE	FALSE	Falling Edge (RESET)
else	–	no event

State storage occurs exclusively within the internal function block.

Application Scenarios

- **Button Press Detection:** Monitoring of a button signal; `SET` when pressed, `RESET` when released.
- **Signal Edge Evaluation in Controllers:** Detection of limit exceedances or pulse beginnings/ends.
- **Adapter-Based Sensor Integration:** Integration of sensors with Boolean outputs into an adapter network, e.g., via the AX/ASR adapter family.

Comparison with Similar Function Blocks

Function Block	Edge Detection	Output Signal	Interface
`E_RF_TRIG`	Ascending + Falling	Separate Event Outputs	Direct Events
`E_TRIG`	Ascending only	One Event	Direct Events
`E_F_TRIG`	Falling only	One Event	Direct Events
AX_ASR_RF_TRIG	Ascending + Falling	Set/Reset Events via Adapter	Pure Adapters

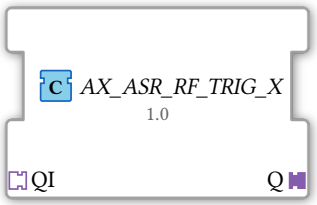
Unlike pure event blocks, this function block offers an adapter interface, enabling simpler connections in hierarchical or typed adapter networks.

Conclusion

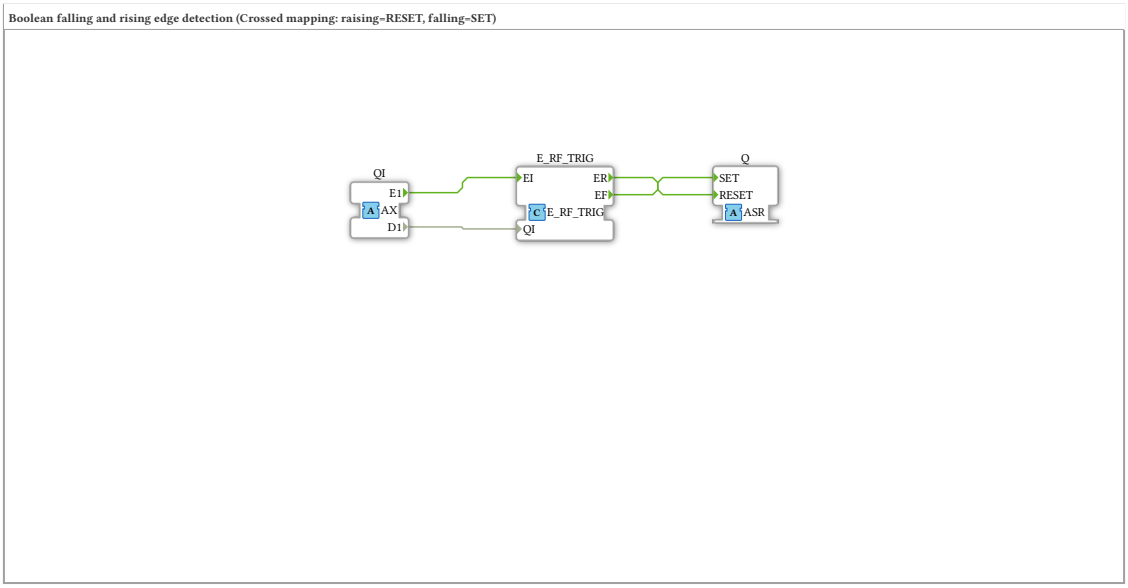
The `AX_ASR_RF_TRIG` is a compact, adapter-based edge detector for Boolean signals. It simplifies the use of the standard `E_RF_TRIG` module in adapter-based architectures and directly converts the detected edges into a set/reset signal. Its loose coupling makes it particularly suitable for modular, reusable control applications.

AX_ASR_RF_TRIG_X

value to check for a rising or falling edge – *AX*



ASR – Set/Reset Output



• – ◦ • – ◦ • – ◦ •

The function block **AX_ASR_RF_TRIG_X** detects the rising and falling edges of a Boolean signal. It maps the detected edges to a set/reset output signal using crossed mapping: The rising edge triggers a reset operation, and the falling edge triggers a set operation. The block is implemented as a pure adapter

block and internally uses the standard function block E_RF_TRIG. The function block has no direct event or data inputs/outputs at the interface level. All communication takes place via two specified adapters.

None (Event control is handled via the **QI** adapter)
None (Event output is handled via the **Q** adapter)
None (Data input is handled via the **QI** adapter)
None (Data output is handled via the **Q** adapter)

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

Adapter	Type	Direction	Description
QI	adapter::types::unidirectional::AX	Socket	Receives the Boolean signal to be checked for edges. Contains the event E1 (trigger) and the data word D1 (value).
Q	adapter::types::unidirectional::ASR	Plug	Outputs the set/reset signals. The corresponding actions are signaled via the events SET (falling edge) and RESET (rising edge).

Functionality

An instance of the standard function block E_RF_TRIG is used in the block's FB network. The connections are configured as follows:

- The event input QI.E1 triggers edge detection and is connected to E_RF_TRIG.EI.
- The data value QI.D1 is forwarded to E_RF_TRIG.QI.
- The output event E_RF_TRIG.EF (falling edge) is assigned to the adapter output Q.SET.
- The output event E_RF_TRIG.ER (rising edge) is assigned to the adapter output Q.RESET.

This configuration results in the desired "crossed mapping":

- **Rising edge** at the input value → **RESET** pulse at the output.
- **Falling edge** at the input value → **SET** pulse at the output.
- The function block is implemented entirely as an adapter function block, i.e., It does not have its own event/data interfaces, but communicates exclusively via the defined adapters **QI** and **Q**.
- The inverse assignment of edges (Rising → Reset, Falling → Set) is a special feature compared to the standard use of E_RF_TRIG.
- The implementation follows the IEC 61499-1 specification and uses the adapter::events::unidirectional library.

The function block does not have its own state machine. Internal edge detection is performed by the integrated function block E_RF_TRIG, whose state logic (waiting for edge) is responsible for the functionality.

- **Reverse Set/Reset Logic**: In controllers where an active HIGH state is to be reset by a rising edge and set by a falling edge.
- **Simplification of Adapter Chains**: By encapsulating the inverse logic in a separate function block, the network structure becomes clearer.
- **E_RF_TRIG** (IEC 61499 Standard): Provides a rising edge on ER and a falling edge on EF. AX_ASR_RF_TRIG_X inverts this assignment.

- **`E_RS_TRIG`**: Detects only rising edges. This function block detects both edges and assigns them to different outputs.
- **`AX_ASR_TRIG_X`**: Possible alternative variants with different mapping logic differ in their edge-output assignment.

The function block `AX_ASR_RF_TRIG_X` offers specialized edge detection with inverted output assignment. By using adapters, it integrates seamlessly into ad-hoc 61499 communication and enables clear, reusable encapsulation of this logic. It is particularly suitable for scenarios where the standard mapping of `E_RF_TRIG` does not meet the desired behavior.

Technical Features

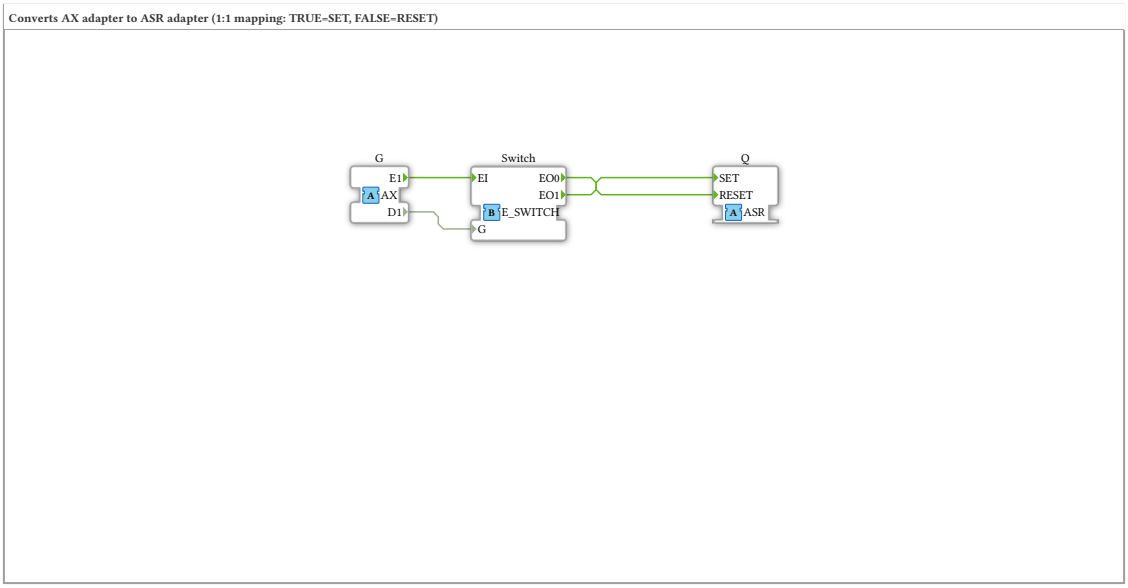
State Overview

Application Scenarios

Comparison with Similar Function Blocks

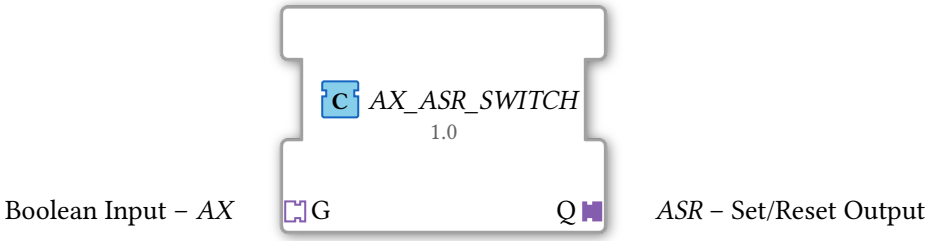
Conclusion

AX_ASR_SWITCH / AX_ASR_SWITCH_X



Introduction

These function blocks serve as a bridge between the data-oriented AX world and the event-oriented ASR world (Set/Reset). They convert a Boolean signal (encapsulated in an AX adapter) into dedicated `SET` and `RESET` events. Since the direction of the events (which sets the state, which clears it) is hard-coded in the adapter, two variants are available.



Variants

AX_ASR_SWITCH (1:1)

- Input `G = TRUE` (1) ➜ Output `SET`

- Input `G = FALSE` (0) ➜ Output `RESET`

AX_ASR_SWITCH_X (Crossed)

- Input `G = TRUE` (1) ➜ Output `RESET`
- Input `G = FALSE` (0) ➜ Output `SET`

Interface Structure

Adapter

- **G** (Socket): Input adapter type `AX`.
- **Q** (Plug): Output adapter type `ASR`.

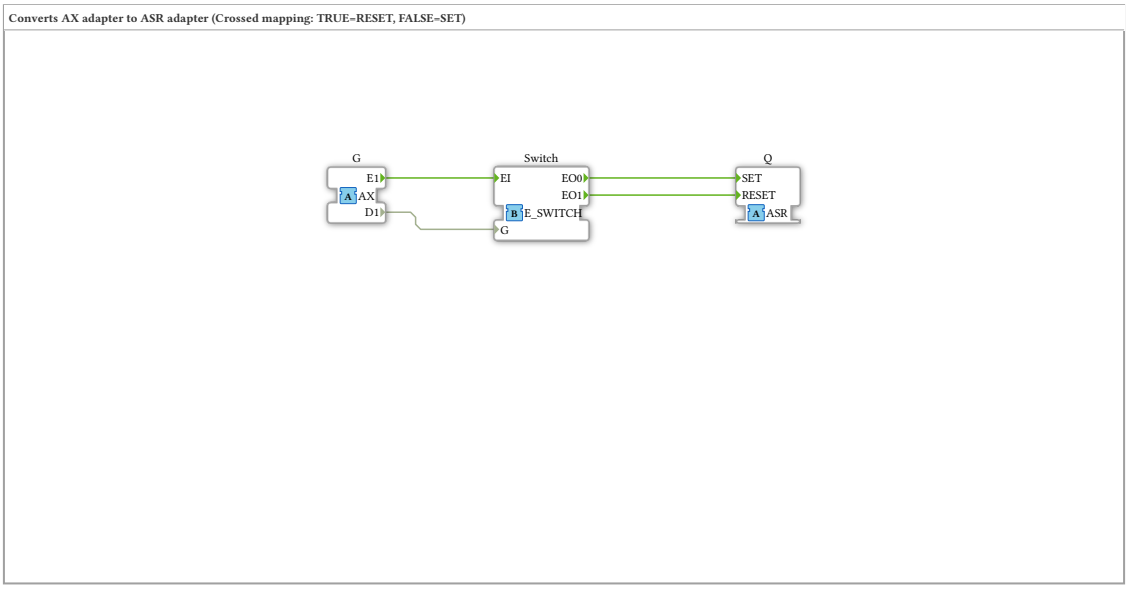
Functionality

The modules internally use a `E_SWITCH`. Each time an event (`E1`) arrives at the input adapter `G`, the current data value (`D1`) is checked, and the corresponding event (`SET` or `RESET`) is triggered at the output adapter `Q`.

Application Scenarios

- Controlling bistable components (flip-flops) with a `ASR` input directly from a `AX` signal flow.
- Simplifying logic circuits where events need to be generated based on a static signal state.

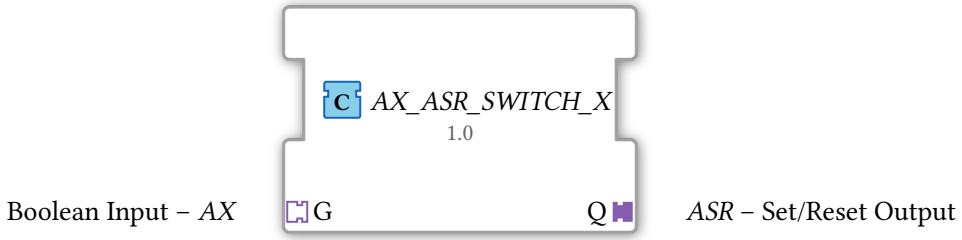
AX_ASR_SWITCH_X



• – ○ • – ○ • – ○ •

Introduction

The function block **AX_ASR_SWITCH_X** is a composite function block responsible for converting between two different adapter types. It converts signals from an **AX** adapter (Boolean Event) into signals from an **ASR** adapter (Asynchronous Set/Reset). The special feature of this function block is the **"Crossed Mapping"** indicated by the suffix "X" and the comment. Unlike a standard mapping, this function block inverts the logic: A `TRUE` signal at the input results in a `RESET` event at the output, while a `FALSE` signal triggers a `SET` event.



Interface Structure

The function block interacts primarily via adapter interfaces.

Event Inputs

This function block does not have direct event inputs in the interface list, as these are encapsulated within the adapter `G`.

Event Outputs

This function block does not have direct event outputs in the interface list, as these are encapsulated within the adapter `Q`.

Data Inputs

This function block does not have direct data inputs in the interface list, as these are encapsulated within the adapter `G`.

Data Outputs

This function block does not have direct data outputs.

Adapter

Name	Type	Component	Comment
G	`adapter::types::unidirectional::AX`	Socket (Input)	Boolean input adapter (contains event `E1` and data `D1`).
Q	`adapter::types::unidirectional::ASR`	Plug (Output)	Set/Reset output adapter (contains events `SET` and `RESET`).

Functionality

The **AX_ASR_SWITCH_X** internally uses a standard `E_SWITCH` block (`iec61499::events::E_SWITCH`) to execute the logic. The circuit is configured as follows:

1. An event at input adapter **G** (event `E1`) triggers processing.
2. The data value from input adapter **G** (data `D1`) is evaluated.
3. **Logic Inversion (Crossed Mapping):**

- If **G.D1** = **TRUE**: The internal switch forwards the event to its output `EO1`. This output is connected to **Q.RESET**.
- If **G.D1** = **FALSE**: The internal switch forwards the event to its output `EO0`. This output is connected to **Q.SET**.
- In summary:
- Input `TRUE` → Output `RESET`
- Input `FALSE` → Output `SET`

Technical Features

- **Adapter Conversion:** Bridge between the simple Boolean event world (AX) and the set/reset control logic (ASR).
- **Inverted Logic:** This block is specifically designed for applications where an active signal (`TRUE`) should trigger a reset or deactivation (`RESET`), and an inactive signal (`FALSE`) should trigger an activation (`SET`) – or vice versa, depending on the interpretation of the connected systems.

State Overview

Since this is a Composite Function Block without its own state machine (ECC), its behavior is defined solely by signal passthrough.

Input G (D1) on Event G (E1)	Triggered Event at Q	Logical Meaning
TRUE	**RESET**	Reset / Stop
FALSE	**SET**	Set / Start

Application Scenarios

- **Fail-Safe Logic:** In systems where an active signal (High/True) represents the "normal state" or "idle state" (Reset), and the absence of the signal (Low/False) should trigger an action (Set) (wire break safety).
- **Logic Inversion:** Adapting sensors that output "Active High" to actuators or control modules that expect "Active Low" (or reset on high).
- **Specific Protocol Adaptation:** When a subsystem communicates via an AX adapter, but the receiving subsystem requires an ASR adapter with inverted polarity.

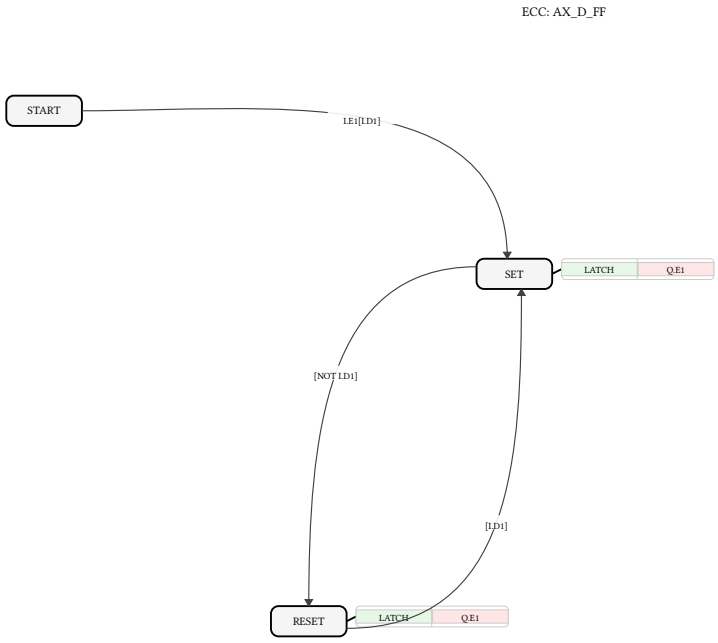
⚖️ Comparison with Similar Modules

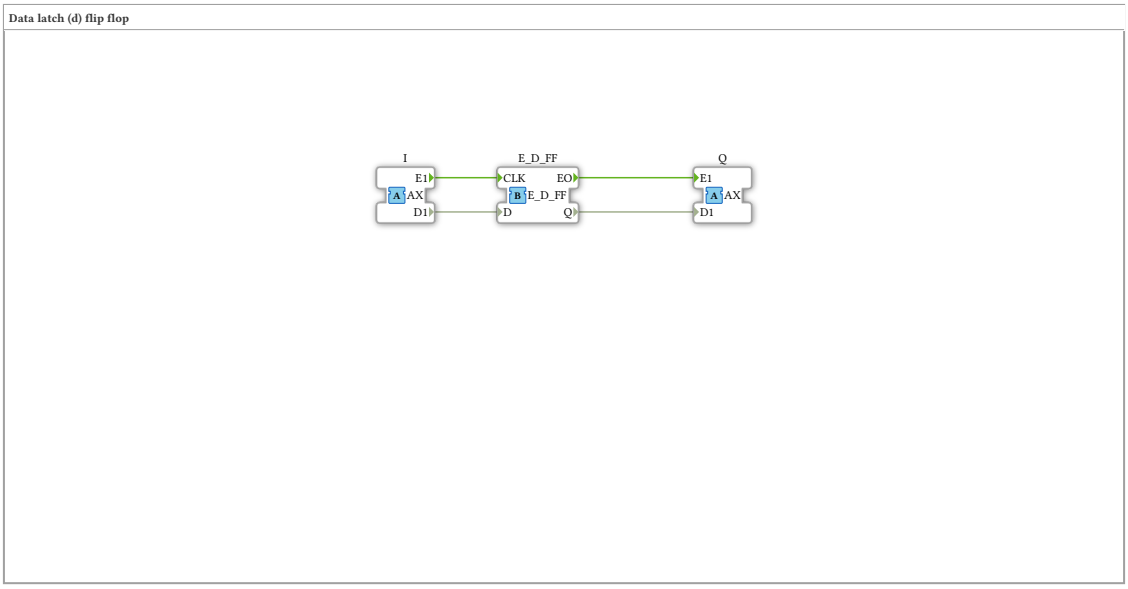
- **AX_ASR_SWITCH (hypothetical/standard):** A standard converter would typically map `TRUE` to `SET` and `FALSE` to `RESET`. The `_X` module described here is the exact opposite.
- **AX_ASR_SWITCH (hypothetical/standard):** * **E_SWITCH:** The underlying IEC 61499 standard function block, which handles the actual switching of events based on a Boolean value, but without the encapsulation in an adapter.

Conclusion

The **AX_ASR_SWITCH_X** is a specialized adapter converter that performs a logical inversion ("crossed mapping") between an AX input and an ASR output. It is essential for system integrations where the signal polarity between source and sink needs to be reversed without placing additional discrete logic devices in the main network.

AX_D_FF

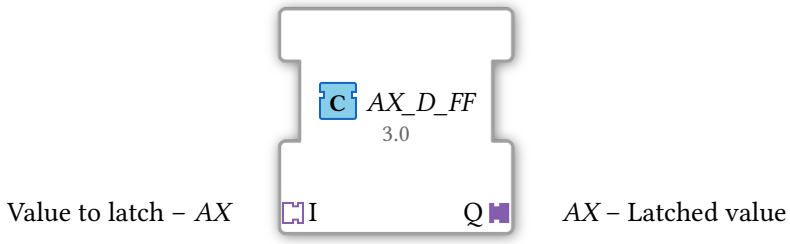




• – ○ • – ○ • – ○ •

Introduction

The AX_D_FF is a data latch flip-flop that serves as a basic building block in digital circuitry. It is a D-type flip-flop that stores the applied data value and outputs it upon certain events.



Interface Structure

Event Inputs

- No direct event inputs available

Event Outputs

- No direct event outputs available

Data Inputs

- No direct data inputs available

Data Outputs

- No direct data outputs available

Adapters

- **I** (Socket): Input adapter of type `adapter::types::unidirectional::AX` - Receives the value to be stored
- **Q** (Plug): Output adapter of type `adapter::types::unidirectional::AX` - Outputs the stored value

Functionality

The AX_D_FF operates as a D-latch flip-flop with three states:

- **START**: Initial state
- **SET**: Stores the input value and outputs it
- **RESET**: Sets the output Back

The LATCH algorithm copies the input value `I.D1` to the output `Q.D1`. The state transitions are controlled by the value of `I.D1`.

Technical Features

- Uses unidirectional adapters for inputs and outputs
- Implemented as a Basic Function Block according to IEC 61499
- Features simple latch functionality without clocking

State Overview

1. **START → SET**: Upon arrival of `I.E1` with `I.D1 = TRUE`
2. **SET → RESET**: When `I.D1 = FALSE`
3. **RESET → SET**: When `I.D1 = TRUE`

Application Scenarios

- Data storage in control systems
- State storage in sequential processes
- Signal delay and buffering
- As a basic building block for more complex flip-flop circuits

⚖️ Comparison with similar components

Compared to clocked D flip-flops, the AX_D_FF is asynchronous and saves the value immediately upon a change in the input conditions. It is a level-triggered element rather than an edge-triggered one.
Compare with E_D_FF

🛠️ Related Exercises

- Exercise_170_AX

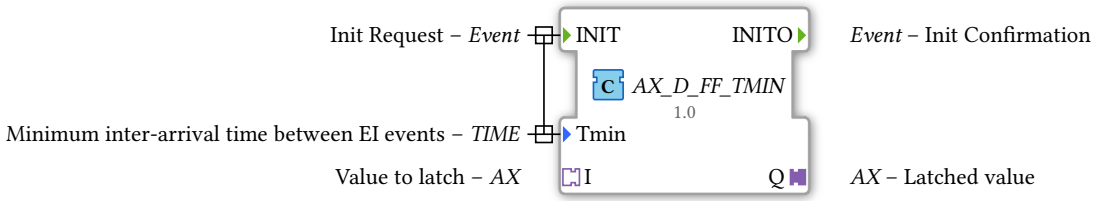
Conclusion

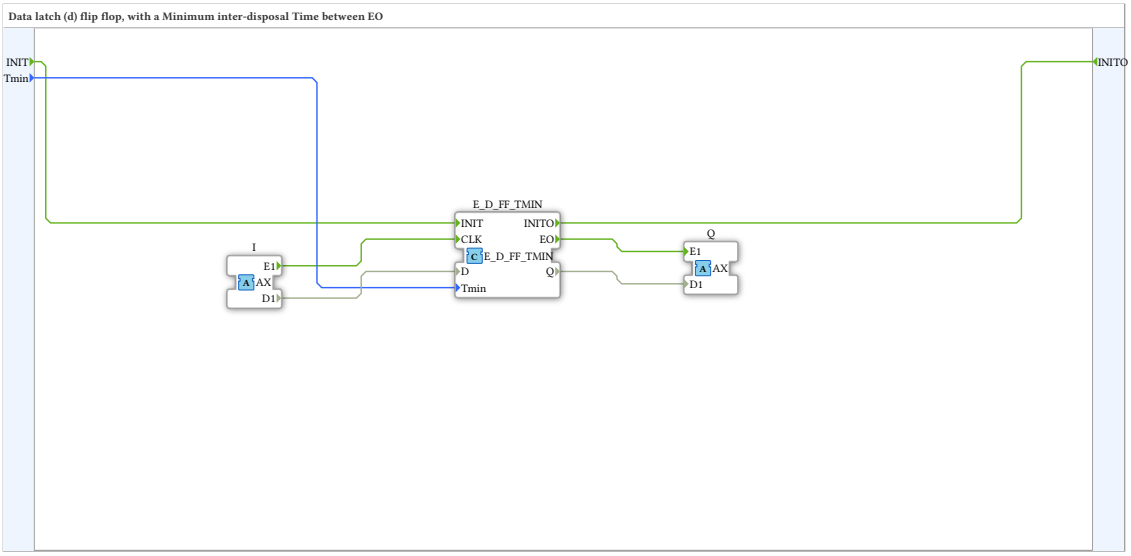
The AX_D_FF offers a simple and effective solution for basic data storage tasks in 4diac control systems. Its clear state logic and the use of standardized adapters make it a reliable component for various use cases.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AX_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_D_FF_TMIN** implements a data-latch D flip-flop that additionally enforces a minimum time between successive event outputs (e.g., EO). It serves as a modular wrapper around the IEC 61499 standard function block `E_D_FF_TMIN` and uses a unidirectional adapter (`adapter::types::unidirectional::AX`) for event-driven data transfer. This function block is particularly suitable for applications requiring reliable data transfer with an adjustable minimum interval between output pulses.

Interface Structure

Event Inputs

Event	Type	Description
`INIT`	EInit	Initialization request; sets the minimum delay time (parameter `Tmin`).

Event Outputs

Event	Type	Description
`INITO`	EInit	Confirmation of successful initialization.

Data Inputs

Name	Type	Description
`Tmin`	TIME	Minimum inter-disposal time between two consecutive events at output `EO`. The value is only adopted upon a `INIT` event.

Data Outputs

This function block has no direct data outputs; the latching data value is provided via the adapter `Q`.

Adapters

Role	Name	Type	Description
Socket	`I`	adapter::types::unidirectional::AX	Input adapter; Provides the data word to be latched (`D1`) and the clock event (`E1`).

Plug	Q	adapter::types::unidirectional::AX	Output adapter; provides the latching data word (D1) and the output event (E1).

The adapters AX are unidirectional: They transmit exactly one event and one associated data value.

Functionality

The function block encapsulates the internal function block E_D_FF_TMIN, which operates as follows:

1. **Initialization**: When a INIT event is received, the current time Tmin is stored as the minimum lock time for the output E0. The function block acknowledges this with INITO.
2. **Clocking (D Flip-Flop)**:

- An event at socket I.E1 is forwarded as a clock signal (CLK) to the internal flip-flop.
- Simultaneously, the data value I.D1 is adopted as the new input value D.
- On each rising edge of the clock (here: each E1 event), the current data value D is acquired and made available at output Q.
- After acquisition, an output event E0 is generated at plug Q.E1 – but only after the minimum lock time since the last E0 has elapsed. This ensures that output pulses do not follow each other faster than specified by Tmin.

1. **Output**: The latching data value appears at plug Q.D1, and the corresponding event at plug Q.E1.

Technical Features

- **Minimum Interval Time**: The function block enforces an adjustable minimum time (Tmin) between two output events. If the clock signal frequency exceeds this time, individual clock events are suppressed so that the output does not exceed a maximum permissible event sequence.
- **Adapter Interface**: By using the unidirectional adapter AX, loose coupling between transmitter and receiver is achieved. The adapter transports both the event and the data value in a single logical unit.
- **Initialization Dependency**: The minimum time Tmin is set only once at INIT and remains constant until the next INIT. Changes during operation are not permitted.

State Overview

The function block does not have an explicit state machine. The internal logic of the embedded E_D_FF_TMIN can assume the following states:

- **Idle**: Waiting for a INIT or a clock event.
- **Initialized**: Tmin has been acquired; the function block is ready for operation.
- **Lockout Time Active**: After an output event, the delay defined by Tmin is applied; incoming clock events are ignored.
- **Ready**: After the lockout time has expired, the next clock event can trigger a new output event.

Application Scenarios

- **Buffered Sensor Values**: A sensor delivers data at an irregular rate. The AX_D_FF_TMIN acquires the last valid value and passes it on to a subsequent controller with a maximum permissible output frequency.
- **Signal Debouncing**: Mechanical pushbuttons or switches generate bounce pulses. The minimal delay suppresses excessively rapid state changes.
- **Data Forwarding with Bandwidth Limiting**: In communication networks, this component can be used to limit the output rate of a data channel without requiring adaptation of the sender.

Comparison with Similar Components

Component	Difference
E_D_FF (Standard D Flip-Flop)	No time limit on output pulses; events are forwarded immediately.
AX_D_FF_TMIN (present)	Inserts the minimum interval Tmin between output events, implemented via the built-in E_D_FF_TMIN.
E_CTD / E_CTU (counter)	Count events; do not offer data transfer or latching functionality.

The AX_D_FF_TMIN combines the latching data transfer of a flip-flop with the time-based filtering of a monostable multivibrator, but is implemented as a pure event-to-data adapter.

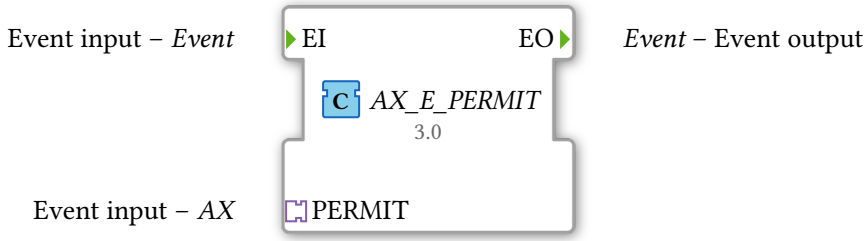
Conclusion

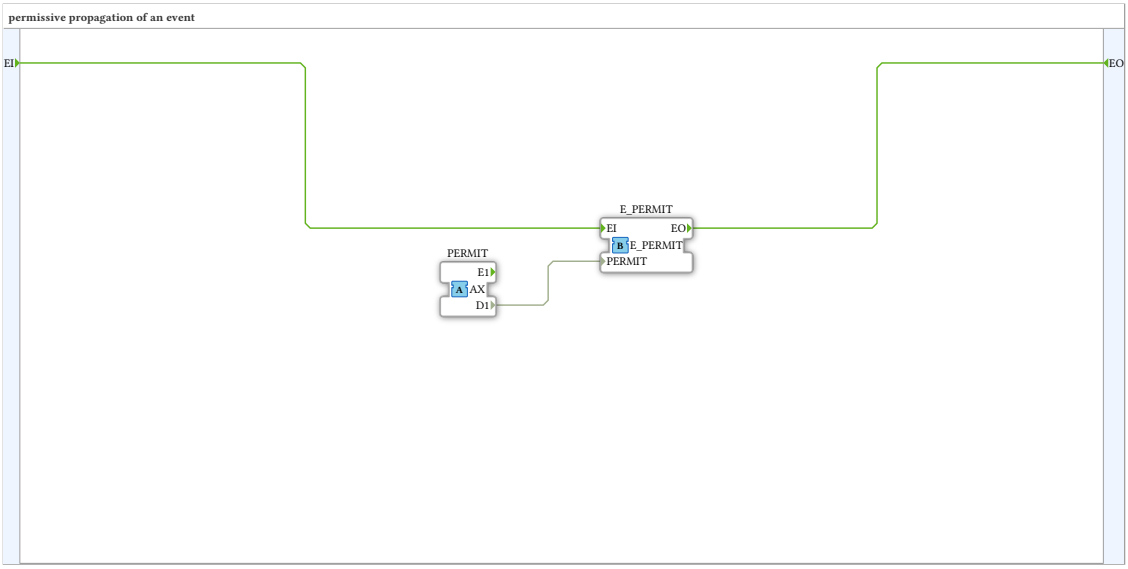
The **AX_D_FF_TMIN** is a specialized function block for applications that require clocked data transfer with a guaranteed minimum time between output events. The encapsulation of the standard function block `E_D_FF_TMIN` and the adapter interface facilitate integration into modular 4diac projects. The function block offers a clean separation of initialization and dynamic operation and is particularly suitable for time-critical, event-driven systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter function block on ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AX_E_PERMIT





• – ○ • – ○ • – ○ •

Introduction

The **AX_E_PERMIT** function block implements conditional event propagation (permissive propagation). An incoming event is only forwarded to the output if a permit signal provided via an adapter has the value `TRUE`. The function block acts as a gate that can interrupt or enable the processing chain depending on external conditions.

Interface Structure

Event Inputs

Name	Type	Comment
EI	Event	Event Input

Event Outputs

Name	Type	Comment
EO	Event	Event Output

Data Inputs

This function block has no data inputs of its own. The enable signal is provided via an adapter.

Data Outputs

This function block has no data outputs.

Adapter

Role	Name	Type	Comment
Socket	PERMIT	`adapter::types::unidirectional::AX`	Provides the enable signal (D1)

The adapter is unidirectional and provides a Boolean value (`D1`) that controls the behavior of the event gate.

Functionality

The function block internally contains a standard function block `E_PERMIT` from the IEC 61499 library. The incoming event `EI` is directly forwarded to the internal input `E_PERMIT`. Its output `EO` is only active if the corresponding enable input (`PERMIT` of the internal block) assumes the value `TRUE`. This enable signal is fed from the adapter `PERMIT` via the connection `PERMIT.D1`.

Therefore:

- **Permit = TRUE:** The event `EI` is passed on to `EO` without delay.
- **Permit = FALSE:** The event is suppressed; `EO` remains inactive.

Technical Features

- **Adapter-Based Interface:** The enable signal is not provided as a classic data input, but via an adapter (`AX`, unidirectional). This enables modular coupling and reuse of the enable logic.
- **No Data Forwarding:** The function block itself does not output any data. It serves solely to control the event flow.
- **Unidirectional Adapter:** The adapter provides only one data value (D1) to the function block; feedback is not provided.
- **Use of a Standard Block:** Internally, the proven `E_PERMIT` block is used, which implements the actual enable logic.

State Overview

The block itself has no visible states. Its internal behavior is determined by the `E_PERMIT` block, which essentially represents a passive wait loop:

- **Waiting for Event:** The block waits for an event at input `EI`.
- **Permit Check:** Upon arrival of an event, the state of the enable signal is evaluated.
- **Forwarding:** The event is only output to `EO` if `PERMIT = TRUE` is present.

There is no explicit state machine; the block operates purely combinatorially at the event level.

Application Scenarios

- **Safety Interlock:** An event (e.g., machine start) is only forwarded if an external safety circuit provides a release signal.
- **Conditional Processing:** In a production control system, data from sensors can only be sent to the higher-level controller when the operating mode is activated.
- **Test and Simulation Environments:** An event can be selectively enabled or disabled via a manual release signal to test specific scenarios.
- **Resource Conservation:** In a complex automation system, unnecessary events can be suppressed by disabling the permit to reduce bus load.

Comparison with Similar Blocks

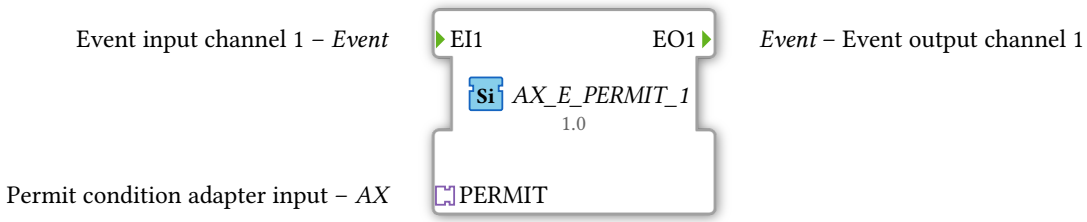
Block	Special Feature
E_PERMIT	Standard block with direct Boolean input. No adapter required.
AX_E_PERMIT	Same logic, but the enable signal is provided via a unidirectional adapter, which facilitates coupling across subsystems.
E_SR (Set-Reset)	Event-based state storage; not directly suitable as a gate.
E_MUX	Event switch that toggles between two outputs depending on a data input – not a pure blocking function.

The advantage of `AX_E_PERMIT` lies in the clean separation of the enable logic by an adapter, which promotes the reuse of control modules.

Conclusion

The `AX_E_PERMIT` function block represents a flexible and modular solution for conditional event forwarding. By using an adapter for the enable signal, the interface is clearly defined and can be used across hierarchical boundaries. The block is particularly suitable for safety-critical or configurable control applications where event processing depends on external conditions.

AX_E_PERMIT_1



• – ◦ • – ◦ • – ◦ •

Introduction

AX_E_PERMIT_1 is a compact event gating function block that propagates a single event channel only when a permit condition is satisfied. Instead of using a direct Boolean data input, the permit condition is provided through a unidirectional adapter socket named `PERMIT`. This makes the block suitable for modular automation solutions in which permission logic is separated from the event consumer.

The block follows the IEC 61499-1 model and is declared as a generic function block type. It has one event input, one event output, and no dedicated data inputs or data outputs. All condition information is received through the adapter interface.

Interface Structure

Event Inputs

Name	Type	Comment
`EI1`	Event	Event input channel 1

Event Outputs

Name	Type	Comment
`EO1`	Event	Event output channel 1

Data Inputs

There are no data inputs.

Data Outputs

There are no data outputs.

Adapters

Direction	Name	Type	Comment
Socket	PERMIT	adapter::types::unidirectional::AX	Permit condition adapter input

The block contains a single adapter socket. The connected adapter supplies the permit or enable condition used to decide whether an incoming event is forwarded.

Functionality

AX_E_PERMIT_1 acts as a permissive event gate.

When an event arrives at EI1, the block evaluates the condition received through the PERMIT adapter socket. If the condition is active, the event is immediately propagated to EO1. If the condition is not active, the event is suppressed and no output event is generated.

The adapter is unidirectional, meaning that the condition data flows from the connected adapter into the socket. The block itself does not send data or events back through the adapter. This keeps the permission source decoupled from the event handling logic.

Because the block has no data inputs, the adapter connection replaces the conventional PERMIT Boolean input that would otherwise be needed. This allows a permission signal to come from a reusable adapter-based logic component without adding explicit data wires to the function block network.

Technical Features

- Single-channel event permit gating.
- Adapter-based condition input using adapter::types::unidirectional::AX.
- No direct data inputs or data outputs.
- Suitable for generic use through the declared generic class name GEN_AX_E_PERMIT.
- Compact interface with only one event input, one event output, and one adapter socket.
- Compatible with IEC 61499-1 Annex A type identification.
- The logic structure makes the block easy to reuse in different permission and interlock scenarios.

State Overview

The function block type does not define an explicit state machine. From the IEC 61499 point of view, the block behaves as a stateless event gate.

Conceptually, the following behavior can be expected:

Condition	Behavior
Event arrives at EI1 and permit is active	EO1 is triggered
Event arrives at EI1 and permit is inactive	The event is ignored
No event arrives at EI1	No output event is generated

Since no internal state is stored between events, the block returns to its idle condition after each event processing step.

Application Scenarios

AX_E_PERMIT_1 is useful wherever an event must be allowed or blocked by an external condition. Typical scenarios include:

- Machine start commands that are only allowed when a safety permission is present.
- Event propagation controlled by an operating mode selector.
- Interlock logic where a process event must be suppressed until a precondition is fulfilled.
- Modular automation architectures where permission signals are provided by reusable adapter blocks.
- Communication between independent control modules that share a common enable condition through an adapter network.

The adapter-based design is particularly helpful when the same permission source must be connected to multiple event consumers without creating complex data wiring.

Comparison with Similar Blocks

Block or Approach	Characteristics	Difference

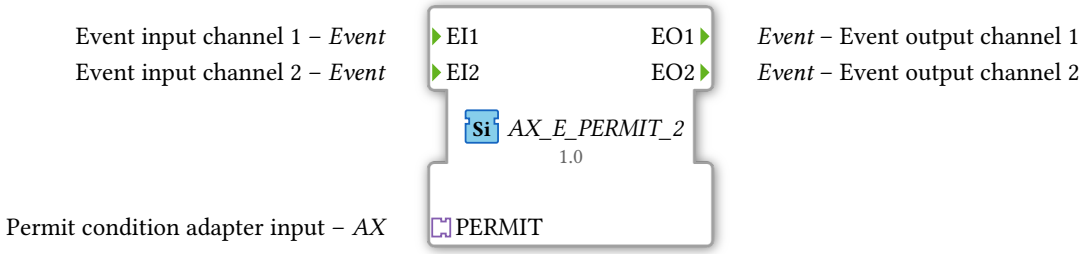
`E_PERMIT` with Boolean input	Uses a direct `PERMIT` data input	Simpler wiring, but permission source must be connected through an explicit Boolean data connection
`E_SWITCH`	Routes an event to one of several outputs based on a selector input	Performs selection, not simple pass/block behavior
`E_SPLIT`	Forwards one event to multiple outputs	Does not apply a permission condition
Generic adapter-based gate	Uses an adapter socket for the enable condition	More modular and reusable, but requires a compatible adapter connection

Compared with a direct Boolean implementation, AX_E_PERMIT_1 trades a small amount of wiring simplicity for a much cleaner and more modular interface. The adapter connection allows the permission logic to be encapsulated, reused, and independently maintained.

Conclusion

AX_E_PERMIT_1 is a focused and reusable event gating block. Its adapter-based permit interface makes it well suited for modular automation systems where conditions are managed through adapter connections rather than plain data signals. With one event input, one event output, and one adapter socket, it provides a simple way to implement permissive event propagation while keeping the overall function block structure clear and maintainable.

AX_E_PERMIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The AX_E_PERMIT_2 is a generic function block designed to propagate events from two independent input channels to corresponding output channels, but only when a permit condition is satisfied. The permit condition is provided externally via a unidirectional adapter of type `AX`. This block is useful for gating event flows based on dynamic or configurable criteria without hardcoding the logic inside the FB itself.

Interface Structure

The block has no data inputs or outputs; it operates purely on event signals and an adapter-based condition input.

Event Inputs

Event	Description
`EI1`	Event input channel 1
`EI2`	Event input channel 2

Event Outputs

Event	Description

`EO1`	Event output channel 1
`EO2`	Event output channel 2

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Type	Description
`PERMIT`	`adapter::types::unidirectional::AX`	Unidirectional adapter providing the permit condition for both event channels.

Functionality

The AX_E_PERMIT_2 acts as a dual-channel event gate. When an event occurs on `EI1` or `EI2`, the block checks the current condition presented by the `PERMIT` adapter. If the condition evaluates to *permit* (i.e., the adapter indicates that events are allowed), the corresponding event is emitted on `EO1` or `EO2`, respectively. If the condition is not satisfied, the event is suppressed and no output is triggered.

The block is generic; the actual permit logic is implemented by the connected adapter instance. This allows the same FB to be reused in different contexts with different permit rules (e.g., based on sensor values, system state, or user-defined logic).

Technical Features

- **Generic Design**: The FB is parameterized via the adapter type, enabling reusability across various permit scenarios.
- **Two Independent Channels**: EI1/EO1 and EI2/EO2 operate independently, each gated by the same adapter condition. This is useful for coordinating multiple event streams under a single permit criterion.
- **No Internal State**: The FB is stateless; it does not store information between event occurrences. It simply passes events through when permitted.
- **Unidirectional Adapter**: The `PERMIT` adapter is unidirectional, meaning the FB reads a value from the adapter without sending data back to the providing block.
- **Event Semantics**: All inputs and outputs are pure event signals, making this block suitable for event-driven control logic.

State Overview

The FB maintains no internal state. Its behavior is deterministic and based solely on the current event input and the instantaneous value of the permit adapter. The block can be considered a combinatorial gate in the event domain.

Application Scenarios

- **Safety Interlocking**: Allow event propagation only when a safety condition (e.g., machine guard closed) is met.
- **Mode-Dependent Operation**: Enable event flows only in certain operating modes of a system (e.g., automatic vs. manual).
- **Load Shedding**: Permit events only when system load is below a threshold.
- **Conditional Logging or Monitoring**: Forward events to a monitoring subsystem only when a specific state is active.

Comparison with Similar Blocks

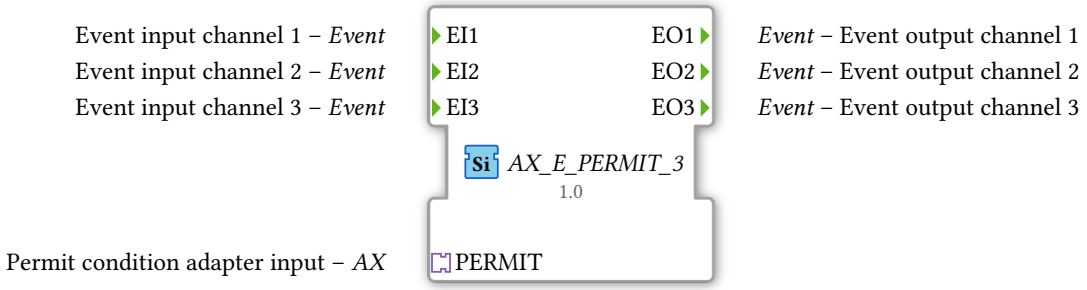
- **Direct Event Propagation (e.g., E_REND)**: Without a permit mechanism, events are always forwarded. AX_E_PERMIT_2 adds a gating condition.
- **Two Separate SINGLE-CHANNEL Permit Blocks**: Using two individual permit blocks would require duplicating the condition logic and potentially cause inconsistent states. AX_E_PERMIT_2 uses a single adapter for both channels, simplifying the design.

- **Event Combination Blocks (e.g., E_MERGE)**: These combine events but do not provide conditional filtering; AX_E_PERMIT_2 filters each channel independently.

Conclusion

The AX_E_PERMIT_2 is a flexible, generic function block that efficiently manages event propagation under a shared permit condition. Its dual-channel structure, combined with an external adapter, makes it a valuable component for building event-driven control systems that require conditional gating. The stateless design ensures predictable behavior and easy integration into larger IEC 61499 orchestration.

AX_E_PERMIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The **AX_E_PERMIT_3** is a generic function block (FB) designed for permissive propagation of three independent event channels. It acts as a gate that forwards each incoming event to its corresponding output only when a global enable condition is satisfied. The enable condition is provided via an adapter interface, making the block highly reusable in various control scenarios where event flow must be conditionally allowed.

This FB is part of the unidirectional event adapter family, specifically tailored for applications that require a single permission signal to control multiple event streams. It is defined as a generic FB, meaning its behavior can be instantiated and customized through type attributes.

Interface Structure

The interface consists of three event input channels, three matching event output channels, and one adapter socket for the permission signal. There are no data inputs or outputs.

Event Inputs

Name	Data Type	Description
EI1	Event	Event input channel 1 – triggers propagation to EO1 if permitted.
EI2	Event	Event input channel 2 – triggers propagation to EO2 if permitted.
EI3	Event	Event input channel 3 – triggers propagation to EO3 if permitted.

Event Outputs

Name	Data Type	Description
EO1	Event	Event output channel 1 – emits an event when EI1 occurs and permission is granted.
EO2	Event	Event output channel 2 – emits an event when EI2 occurs and permission is granted.
EO3	Event	Event output channel 3 – emits an event when EI3 occurs and permission is granted.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Role	Name	Description
adapter::types::unidirectional::AX	Socket	PERMIT	Provides a unidirectional permission signal. The adapter must be connected to a compatible plug that supplies an enable/disable condition. The exact semantics (e.g., boolean value) depend on the specific adapter implementation.

Functionality

The FB implements a simple gating mechanism for three independent event streams. When an event arrives on any of the input channels (EI1, EI2, or EI3), the FB checks the current state of the **PERMIT** adapter. If the permission condition is **true** (or "allowed"), the corresponding output event (EO1, EO2, or EO3 respectively) is emitted. If the permission condition is **false**, the event is discarded and no output event occurs.

The propagation is **not mutually exclusive**: each channel operates independently, and the permission check is identical for all three channels at any given moment. This means that a single permission signal can simultaneously enable or disable all three event paths.

Since the FB has no data inputs or outputs, the permission condition is entirely derived from the adapter. The adapter is expected to provide a boolean-like signal, but the exact representation is defined by the underlying adapter type (in this case, `unidirectional::AX`).

The block is **stateless** in the sense that it does not maintain any internal memory; the decision to forward an event is made at the moment the event occurs based solely on the current adapter value.

Technical Features

- **Generic Implementation**: The FB is marked with the attribute `eclipse4diac::core::GenericClassName = 'GEN_AX_E_PERMIT'`, indicating that it serves as a generic template. This allows the system to instantiate a specialized version of the block for specific adapter definitions via type generation mechanisms.
- **Three-Channel Design**: Supports exactly three event input/output pairs, making it suitable for applications with three parallel event streams.
- **Unidirectional Adapter**: Uses a socket adapter of type `unidirectional::AX`, which is a unidirectional connection. This ensures a clear data flow direction for the permission signal.
- **No Data Interface**: Simplifies usage by eliminating data handling; only event flow control is performed.
- **Direct Event Mapping**: Each input event is mapped one-to-one to its output event with identical naming conventions (EI1→EO1, EI2→EO2, EI3→EO3), simplifying wiring and readability.

State Overview

The FB does not implement an explicit state machine. It operates **combinatorially** with respect to events: each event is processed independently and immediately. There is no history or internal state that could affect subsequent events. The block can be considered as a pure function: given an event on an input and a current permission value, it either emits an output event or does nothing.

In practice, the only "state" is the value of the adapter signal, which is external to the FB.

Application Scenarios

- **Safety Interlocks**: Use a single safety relay or enable signal to block or allow multiple control commands (e.g., start/stop/emergency) simultaneously.
- **Mode Selection**: In machinery, a mode selector can enable or disable a set of event-driven actions (e.g., automatic, manual, maintenance modes).
- **Conditional Data Flow**: When three independent event sources must only be forwarded when a master condition holds (e.g., a supervisory system grants permission).

- **Event Gating in Distributed Systems**: In IEC 61499 based applications, this FB can be used to gate events between function blocks without complex logic, relying on an adapter to provide the permission.
- **Test and Simulation**: The block can be used to simulate an enable/disable switch for a group of event paths in a model.

Comparison with Similar Blocks

Given the generic nature of the FB, several similar blocks exist, often differing in the number of channels or the type of permission signal:

- **AX_E_PERMIT_1** / **AX_E_PERMIT_2** (if they exist): Single- or two-channel versions, where the same permission is applied to fewer event streams. The three-channel version provides a balance between simplicity and parallelism.
- **AX_E_GATE** (generic): May implement a more complex gating logic, possibly with per-channel enable signals instead of a single global permission.
- **AX_E_SELECT** (if available): Might route events to different outputs based on a condition, rather than simply allowing or blocking them.
- **Adapter-Based vs. Data-Based Gating**: Compared to a block with a boolean data input (e.g., `E_PERMIT`), using an adapter decouples the permission source from the FB itself, enabling reuse in different contexts without changing the FB's interface. This makes the adapter-based approach more flexible and modular.

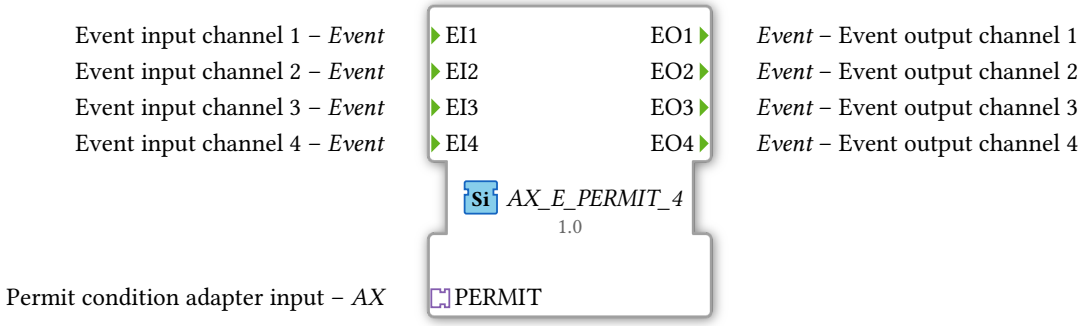
The key differentiator of **AX_E_PERMIT_3** is its **explicit adapter interface** and its **fixed number of three independent channels**, which suits mid-complexity event management tasks.

Conclusion

The **AX_E_PERMIT_3** function block provides a clean, deterministic solution for conditionally propagating three independent event streams under a single permission signal. Its design, leveraging a unidirectional adapter, promotes modularity and reusability in IEC 61499 applications. The simple event-to-event mapping and lack of data handling make it easy to integrate into existing control logic, while its generic nature allows for adaptation to specific adapter definitions when needed.

By offering a straightforward gating mechanism, this block contributes to safer and more controlled event-driven execution in automated systems.

AX_E_PERMIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

AX_E_PERMIT_4 is a generic function block designed for conditional propagation of four independent event channels. It acts as a permissive gate: each incoming event at one of its four event inputs is forwarded to the corresponding event output only if the attached permit condition via the unidirectional adapter socket `PERMIT` allows it. This block is particularly useful in safety-oriented or mode-dependent control applications where event flow must be gated by an external authorization signal.

Interface Structure

Event Inputs

Event	Type	Description

EI1	Event	Event input channel 1
EI2	Event	Event input channel 2
EI3	Event	Event input channel 3
EI4	Event	Event input channel 4

Event Outputs

Event	Type	Description
EO1	Event	Event output channel 1
EO2	Event	Event output channel 2
EO3	Event	Event output channel 3
EO4	Event	Event output channel 4

Data Inputs

This function block has no data inputs.

Data Outputs

This function block has no data outputs.

Adapters

Socket	Type	Description
PERMIT	adapter::types::unidirectional::AX	Unidirectional permit condition input

Functionality

AX_E_PERMIT_4 implements a permissive event propagation scheme. Each pair of input/output events (EI1/EO1, EI2/EO2, EI3/EO3, EI4/EO4) forms an independent channel. When an event is received on one of the event inputs, the block checks the current value of the `PERMIT` adapter. If the permit condition evaluates to true (permission granted), the corresponding event is released on the associated output. If the permit condition is false, the event is suppressed and no output is generated.

The block is generic, meaning it can be instantiated in a project with a concrete adapter type that matches the `adapter::types::unidirectional::AX` interface. The unidirectional nature of the adapter means that the permit signal flows only from the connected partner into this block, without requiring any return data.

Technical Features

- **Generic instantiation**: Declared with `eclipse4diac::core::GenericClassName` equal to `'GEN_AX_E_PERMIT'`, enabling type replacement at instantiation time.
- **Four independent channels**: Allows simultaneous handling of up to four separate event streams with identical gating behavior.
- **Unidirectional adapter**: The permit signal is supplied through a unidirectional adapter socket, simplifying the connection to a permit source (e.g., a safety interlock or operator authorization block).
- **No data processing**: The block does not modify or buffer any data; it purely gates event occurrence.
- **Package compliance**: Aligned with the IEC 61499-1 Annex A standard and located in the `adapter::events::unidirectional` compiler package.

State Overview

The function block does not maintain an explicit internal state machine. Its behavior is combinatorial with respect to events:

- **Permit Granted**: An input event immediately triggers the corresponding output event.
- **Permit Denied**: An input event is discarded; no output event is emitted.

The only dynamic condition is derived from the external permit signal carried by the adapter. Consequently, the block is stateless between event occurrences and responds identically to each event based on the current permit value.

Application Scenarios

- **Safety interlocks**: Suppressing start commands or motion signals when a safety condition is not satisfied.
- **Mode-based operation**: Allowing or blocking commands depending on the current operational mode (e.g., manual vs. automatic).
- **Conditional alarming**: Forwarding alarm events only when a monitoring system is armed.
- **Line gating in production cells**: Passing control events to downstream equipment only when the upstream permit is active.

Comparison with Similar Blocks

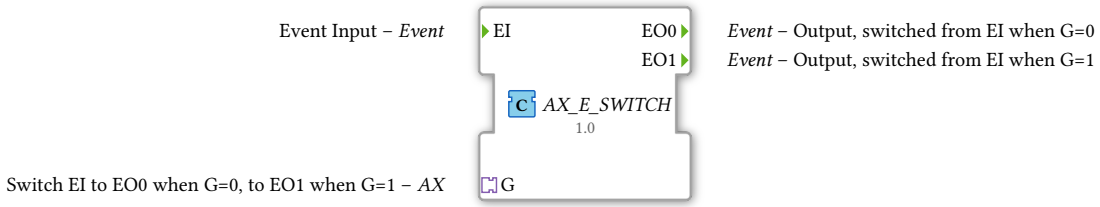
Feature	AX_E_PERMIT_4	AX_E_PERMIT (single channel)	Standard event propagation (no permit)
Number of event channels	4	1	1 or more (direct wiring)
Permit gating	Yes (via unidirectional adapter)	Yes (via unidirectional adapter)	No
Generic type support	Yes	Yes	Not applicable
Data interface	None	None	Usually none
Typical use	Multi-channel safety or mode gating	Simple single-channel gate	Unconditional event forwarding

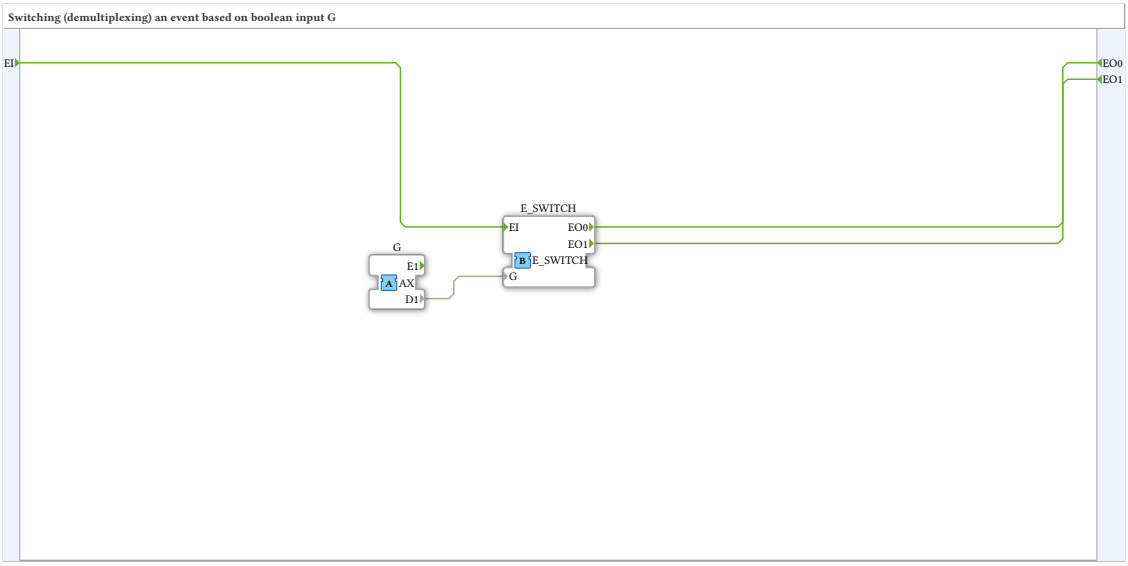
The main advantage of AX_E_PERMIT_4 over a single-channel variant is its compactness when four independent event streams must be gated by the same permit condition. Compared to direct event wiring, it adds a controlled suppression layer, which is essential in safety-related architectures.

Conclusion

AX_E_PERMIT_4 provides a straightforward and reusable solution for conditional event propagation across four independent channels. Its generic design and dependence on a unidirectional permit adapter make it a flexible building block in IEC 61499-based control applications that require event gating under external authorization. The block is easy to integrate, requires no data handling, and contributes to cleaner and more maintainable safety and mode-management logic.

AX_E_SWITCH





• – ○ • – ○ • – ○ •

Introduction

The **AX_E_SWITCH** function block implements event-driven demultiplexing (switching) based on a Boolean signal. It receives an event at input **EI** and forwards it to either output **EO0** or **EO1**, depending on the value of adapter **G**. This allows event flows to be split based on states or decisions

---

Interface Structure

Event Inputs

Name	Type	Description
EI	Event	Event input that is switched

Event Outputs

Name	Type	Description
EO0	Event	Activated when **G** = 0
EO1	Event	Activated when **G** = 1

Data Inputs

This function block has no separate data inputs. The switching information is provided via the **G** adapter.

Data Outputs

No data outputs are available.

Adapter

Name	Type	Description
G	`adapter::types::unidirectional::AX`	Boolean interface; provides the value for switching (0 ⟶ EO0, 1 ⟶ EO1)

Functionality

The **AX_E_SWITCH** is implemented internally as an encapsulation of the standard function block **E_SWITCH**. An incoming event at **EI** is forwarded to the internal function block. The adapter **G** provides the Boolean value (via the data connection `G.D1` ⟶ `E_SWITCH.G`).

- If **G = 0**, the event is output to **EO0**.
- If **G = 1**, the event is output to **EO1**.

The function block operates deterministically and without internal state management; the switching occurs in a single execution step.

–

Technical Features

- **Adapter Interface:** The Boolean switching value is not provided as a classic data input, but via an adapter. This enables loose coupling and reuse of standardized interfaces (type `unidirectional::AX`).
- **Reusability:** Internally, the IEC 61499 standard function block `E_SWITCH` is used, ensuring compatibility and easy traceability.
- **No Side Effects:** The function block is purely component-based and does not modify any global data or states.

State Overview

The **AX_E_SWITCH** does not have its own explicit state machine. Its behavior is purely event-driven and does not differ from a simple switch. Therefore, a state overview is not required.

Application Scenarios

- **Event Switch in Control Logic:** Splitting an event stream into two paths, e.g., depending on a sensor value or operating mode.
- **Quality Control:** Forwarding an event to different processing blocks depending on the test result (pass/fail).
- **Operating Mode Switching:** Controlling processes that react differently in normal or fault mode.

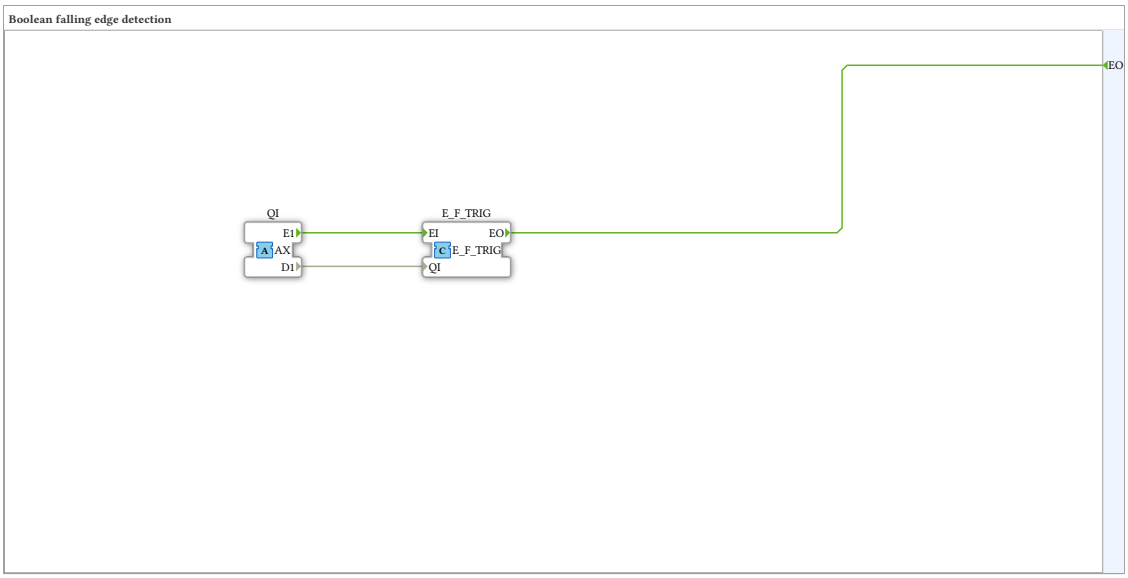
Comparison with Similar Function Blocks

- **Standard function block `E_SWITCH`:** Has a direct Boolean data input instead of an adapter. The **AX_E_SWITCH** encapsulates this input and offers an adapter-based interface, which is preferred in modular architectures.
- **`E_DEMUX` (comparable function):** Some libraries also offer demultiplexers, but usually with multiple outputs or without adapter connectivity. The **AX_E_SWITCH** focuses on the special case of dual outputs with an adapter.

Conclusion

The **AX_E_SWITCH** is a simple yet useful function block for event-based switching. Its adapter interface allows it to be seamlessly integrated into component-oriented control projects and enables a clean separation of event logic and decision data. It is particularly suitable for modular and reusable automation solutions.

AX_F_TRIG



• – ○ • – ○ • – ○ •

The function block **AX_F_TRIG** (Boolean falling edge detection) is used to detect a falling edge (change from TRUE to FALSE) based on an adapter interface. It is specifically designed for use with the unidirectional adapter type `AX`. The block checks the Boolean value received via the adapter and triggers an output event if this value is `FALSE`. In the context of event chains where the adapter's event signals a value change, this corresponds to the detection of a falling edge.

The block has an adapter interface (socket) for input and an event output.

This block does not have direct event inputs. Event control is handled via the adapter.

- **EO** (Event Output): This event is triggered when a falling edge (or a logical FALSE state) is detected at the adapter.

This function block has no direct data inputs. The data is obtained via the adapter.

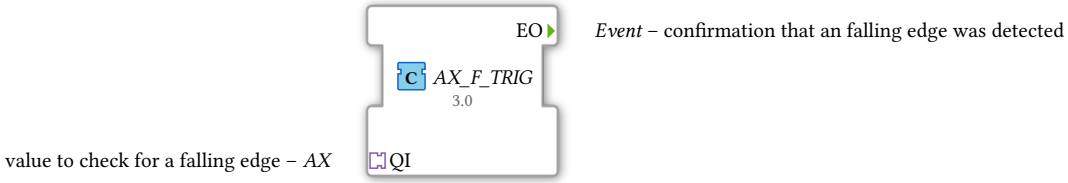
This function block has no direct data outputs.

Data Outputs

Data Inputs

Event Outputs

Event Inputs



Interface Structure

Introduction

Adapter

- **QI** (Type: `adapter::types::unidirectional::AX`): The input socket. The function block receives both the trigger event and the Boolean value to be checked via this adapter.
- *Comment:* Value to be checked for a falling edge.

Functionality

The **AX_F_TRIG** is implemented as a composite function block (FB). Its internal logic is based on processing the signals from the `QI` adapter:

1. The `QI` adapter provides an event (`E1`) and a Boolean data value (`D1`).
2. These signals are passed to a D flip-flop (`E_D_FF`). The event acts as the clock (CLK), and the data as the input (D).
3. The flip-flop stores the value and triggers an acknowledgment event.
4. A downstream `E_SWITCH` component evaluates the output state (`Q`) of the flip-flop.
5. If the value `FALSE` (logic 0) is present, `E_SWITCH` passes the event to output **EO**.
6. If the value `TRUE` (logic 1) is present, no event is generated at output **EO**.

In summary: If the adapter sends an event and the transmitted data value is `FALSE`, the function block `EO` fires.

- **Adapter Integration:** The function block encapsulates the logic for splitting the `AX` adapter into individual events and data for logical processing.
- **Composite Network:** The implementation is achieved purely through the interconnection of standard IEC 61499 components (`E_D_FF`, `E_SWITCH`) and contains no custom algorithms in ST or C++.

Since it is a Composite Function Block, the **AX_F_TRIG** does not have its own ECC (Execution Control Chart). Its state is implicitly determined by the state of the internal components (especially the D flip-flop `E_D_FF`).

- **Signal Processing:** Used in systems that utilize `AX` adapters to communicate binary state changes.
- **Event Filtering:** Blocks events as long as the associated data signal is `TRUE` and allows them to pass if it is `FALSE`.
- **E_F_TRIG:** The standard function block for falling edge detection in IEC 61499. It uses separate event (`EI`) and data inputs (`QI`) instead of an adapter.
- **AX_R_TRIG:** The direct counterpart to this function block. It detects a rising edge (change to `TRUE`) using the same adapter type.
- **E_SWITCH:** A generic switch that routes events based on a Boolean value. **AX_F_TRIG** is essentially a specialization of this for adapters.

The **AX_F_TRIG** is a specialized utility block for structured event processing in 4diac. It significantly simplifies working with `AX` adapters by abstracting the logic for detecting the “low” state (falling edge) and directly providing it as the event `EO`, without requiring the user to manually split and examine the adapter signal.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

State Overview

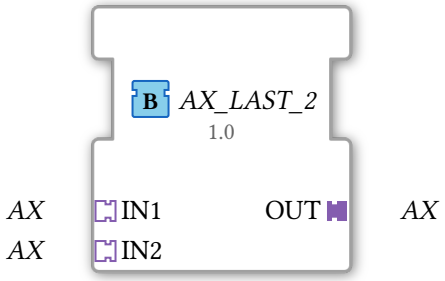
Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

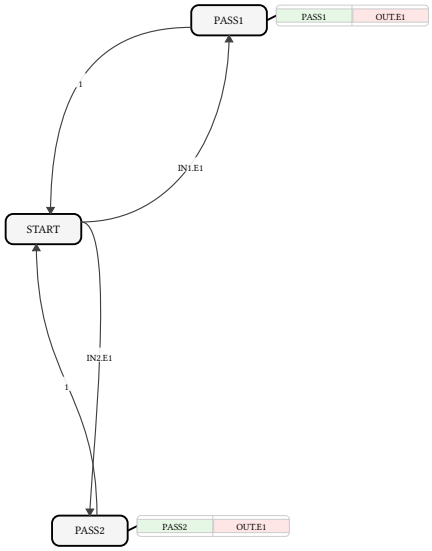
Conclusion

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

AX_LAST_2



ECC: AX_LAST_2



Introduction

The AX_LAST_2 function block merges two same-type Boolean (BOOL)adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: Boolean (BOOL)adapter (unidirectional) - first signal source
- **IN2**: Boolean (BOOL)adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: Boolean (BOOL)adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

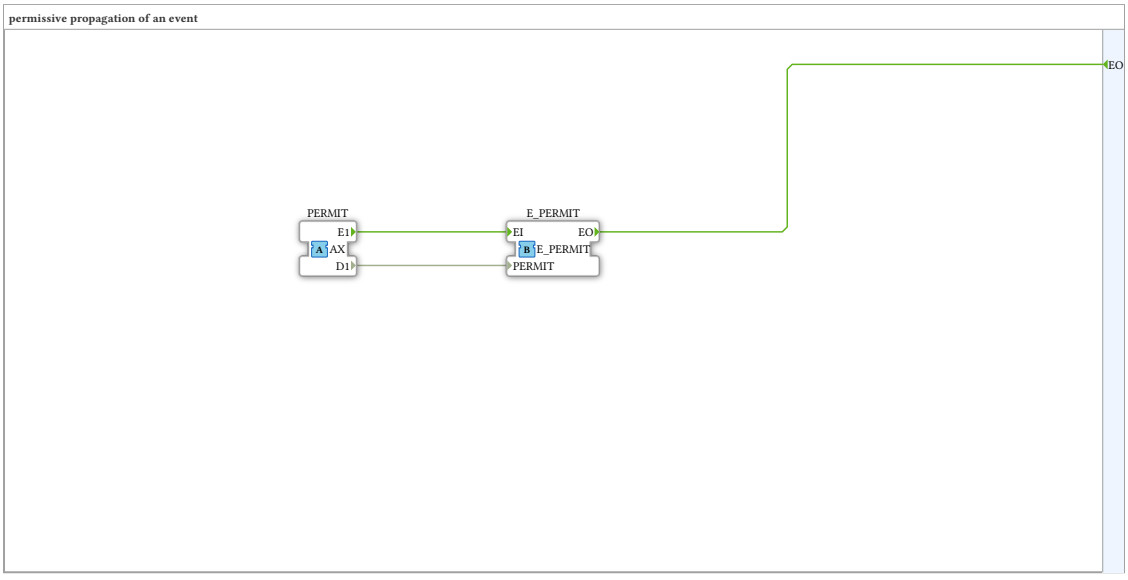
Application Scenarios

- Merging two equally-valid Boolean (BOOL)signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same Boolean (BOOL)value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

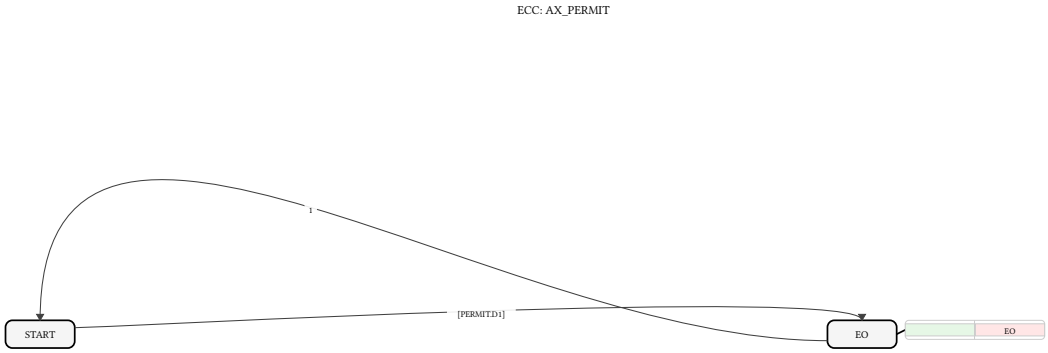
⚖️ Comparison with Similar Blocks

AX_LAST_2 is the inverse of `AX_SPLIT_2`: AX_SPLIT_2 distributes 1 input to 2 outputs, AX_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

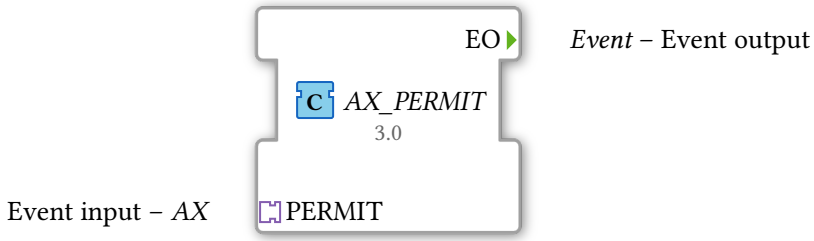
AX_PERMIT (Unidirectional Adapter Permit)



Introduction



The AX_PERMIT function block is an adapter-based variant of the E_PERMIT block, implemented as a **Basic Function Block**. It enables the conditional propagation of an event based on the Boolean data value of an incoming AX adapter. The block receives a combined event and Boolean signal via a AX adapter (referred to as PERMIT). The incoming event is only forwarded to the output (EO) if the Boolean data value of the PERMIT adapter (PERMIT.D1) is TRUE. The data value PERMIT.D1 is **not** passed on to the output (EO), but is only used as a condition.



Interface Structure

Adapter (Socket)

- **PERMIT**: Input adapter of type AX (event E1 + data D1). This adapter controls the function block.

Event Outputs

- **EO**: Pure event output (type Event). This event is triggered when the condition is met.

Functionality

1. **Event Receipt**: When an event (PERMIT.E1) arrives at the PERMIT socket, the corresponding Boolean data value (PERMIT.D1) is checked as a condition.
2. **Conditional Forwarding**: The event output EO is triggered only if PERMIT.D1 = TRUE is true. Otherwise, the event is blocked.
3. **Data Pass-Through**: The Boolean data value (PERMIT.D1) is used by the function block, but **not** passed on via the event output EO.

Technical Features

✓ **Basic Function Block**: Direct implementation of the logic.
✓ **Adapter-Based**: Uses the AX adapter as an input.

✓ **Permissive Logic**: Passes on events conditionally.
✓ **No Data Pass-Through at the Output**: The output `EO` is a pure event output and contains no data (`With Var` is undefined).

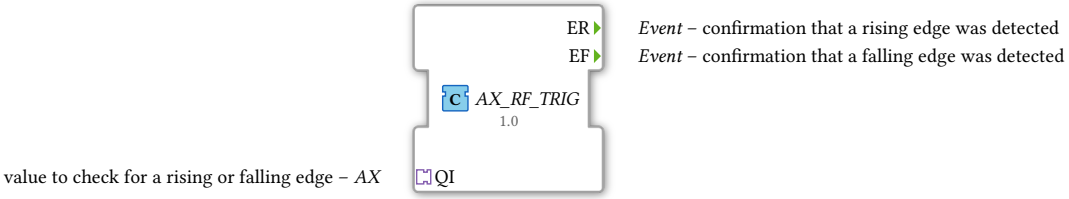
Application Scenarios

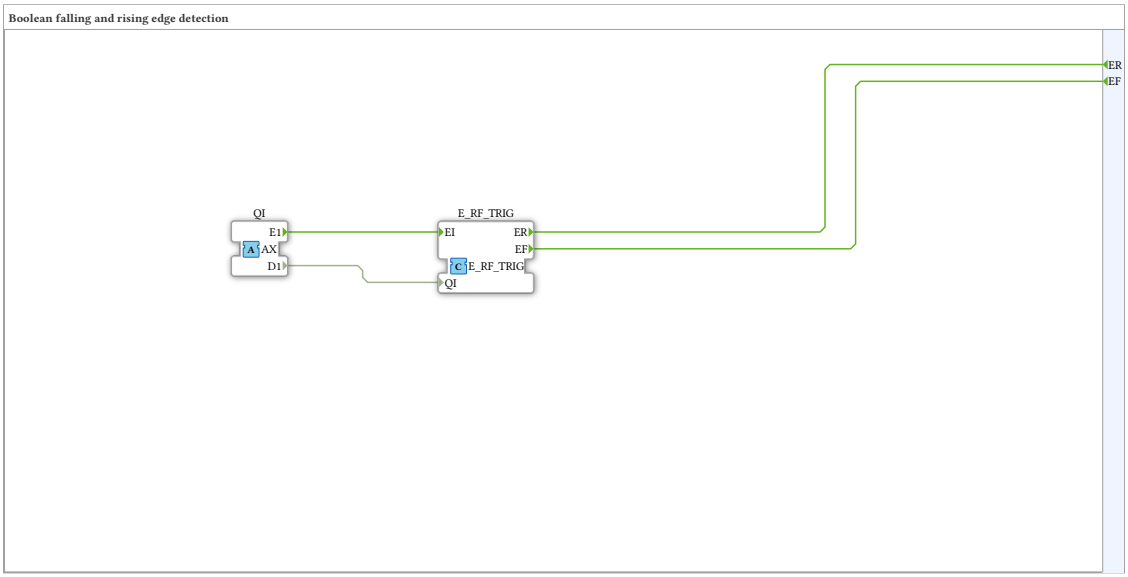
- **Enable Circuit**: A subsequent function block should only be triggered if an enable condition (`PERMIT.D1 = TRUE`) is active.
- **Conditional Event Control**: Control of the event flow based on an external Boolean status received via an AX adapter.

🛠️ Related Exercises

- Exercise_009_AX

AX_RF_TRIG





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_RF_TRIG** is used to detect rising and falling edges of a Boolean signal. It receives the signal value via an adapter and outputs corresponding events when an edge changes. The block encapsulates the standard logic of IEC 61499 edge detection (E_RF_TRIG) and provides it via a uniform adapter interface.

Interface Structure

Event Inputs

The block does not have direct event inputs. Edge detection is triggered by the connected adapter (see Adapter).

Event Outputs

Name	Comment

ER	Confirmation that a rising edge (0⟶1) has been detected.
EF	Confirmation that a falling edge (1⟶0) has been detected.

Data Inputs

No direct data inputs. The Boolean value to be monitored is provided via the adapter.

Data Outputs

No data outputs.

Adapter

Name	Type	Comment
QI	adapter::types::unidirectional::AX	Receives the Boolean value whose edges are to be monitored. The adapter provides both an event (E1) and the data value (D1).

Functionality

The function block internally uses the IEC 61499 standard function block **E_RF_TRIG**. The adapter **QI** queries the current Boolean data value for each incoming event. If the value changes, the internal function block generates an event:

- **ER** on a transition from *false* to *true* (rising edge).
- **EF** on a transition from *true* to *false* (falling edge).

No output events are sent without an edge change.

Technical Features

- The function block is implemented as a pure adapter wrapper. It does not have its own state machine but delegates all logic to the standardized **E_RF_TRIG**.
- The use of an adapter type (here **AX**) enables loose coupling between the signal source and edge detection. The adapter can be reused in different contexts.
- Only event outputs are provided; the current signal value is not passed on as a data output.

State Overview

This function block does not have an explicitly defined state machine. Its behavior corresponds to that of the internal **E_RF_TRIG**, which stores an implicit state (last value). Essentially, two phases can be distinguished:

1. **Waiting for rising edge:** the last value was *false* if *true*, ER is triggered.
2. **Waiting for falling edge:** the last value was *true* if *false*, EF is triggered.

Application Scenarios

- Monitoring of binary sensor signals (e.g., limit switches, light barriers) in automation technology.
- Triggering actions upon signal changes (e.g., starting/stopping a machine).
- Event-driven processing of Boolean states in industrial control systems.

Comparison with Similar Function Blocks

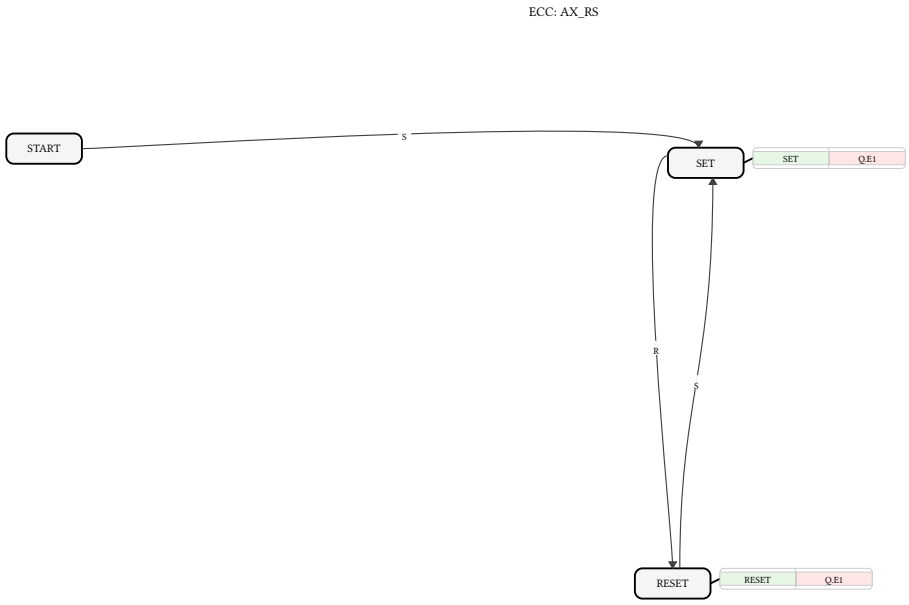
- **E_RF_TRIG:** The standard function block of IEC 61499. **AX_RF_TRIG** encapsulates this and provides an adapter interface, so the signal source does not need to be directly bound to the function block.
- **R_TRIG / F_TRIG:** Detect only one edge direction. **AX_RF_TRIG** detects both directions and outputs corresponding events.
- **Adapter-Based Alternatives:** Other adapter types with similar semantics can also be used for edge detection, provided they supply the Boolean value and an event.

Conclusion

The **AX_RF_TRIG** function block represents a practical, adapter-based solution for detecting both edges of a Boolean signal. By using the standard function block **E_RF_TRIG**, the function is robust and stan-

dards-compliant. The adapter interface facilitates integration into modular systems where signal sources and processing logic need to be decoupled.

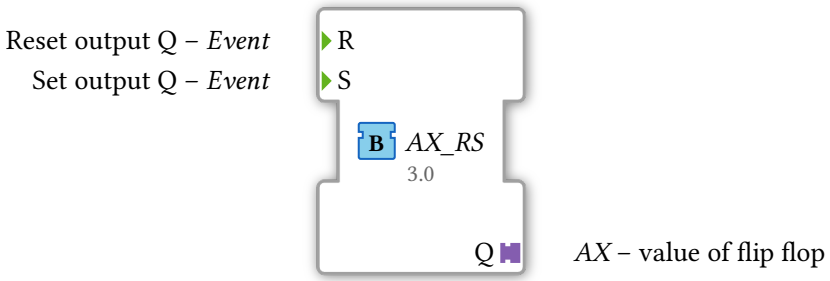
AX_RS



• – ◦ • – ◦ • – ◦ •

Introduction

The AX_RS function block is an event-driven bistable element (flip-flop) based on the RS principle. It is a Basic Function Block that implements a set-reset mechanism and communicates via adapter interfaces.



Interface Structure

Event Inputs

- **R**: Reset event - resets output Q
- **S**: Set event - sets output Q

Event Outputs

No direct event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

- **Q**: Adapter of type `adapter::types::unidirectional::AX` - represents the value of the flip-flop

Functionality

The AX_RS function block operates as an RS flip-flop with the following properties:

- Upon the occurrence of an S event (Set), output Q is set to TRUE
- Upon the occurrence of an R event (Reset), output Q is set to FALSE
- The state is retained until a The opposite event occurs

Technical Features and Comparison with Standards

As with all event-driven bistable elements in IEC 61499 (see also Note 8 in Table A.1 of DIN EN 61499-1), there is no inherent “dominance” of one input, as is known from IEC 61131-3.

- **Comparison to IEC 61131-3**: See RS (Bistable, priority reset). While in the classic PLC world, if S and R1 are TRUE simultaneously, the reset takes precedence, in IEC 61499 each event is processed sequentially. The final state depends on which event was processed last in the execution chain (ECC).
- **Functional Identity**: `AX_RS` is functionally identical to AX_SR. The different naming conventions are solely for consistency with traditional programming and improved readability in the schematic.
- **Adapter Communication**: The component outputs its status exclusively via the adapter `Q` (type `AX`). A change to `Q` triggers the event `Q.E1`.

State Overview

The function block has three states in the ECC:

1. **START**: Initial state
2. **SET**: State after a set operation (Q.D1 = TRUE)
3. **RESET**: State after a reset operation (Q.D1 = FALSE)

State Transitions:

- START ⟶ SET: on S event
- SET ⟶ RESET: on R event
- RESET ⟶ SET: on S event

Application Scenarios

- Storage of binary states with adapter output
- Implementation of interlock circuits in distributed systems
- State storage in sequential processes
- Signal processing in event-driven systems

Related Blocks

- **AX_SR**: Functionally identical, inputs reversed in the symbol.
- **E_RS**: The standard equivalent with direct data/event outputs instead of adapters.

⚖️ Comparison with Similar Building Blocks

Compared to other flip-flop implementations:

- Uses adapter-based communication instead of direct data outputs
- Event-driven state changes
- Simple RS logic without additional clock or enable signals

Comparison with E_RS

🛠️ Related Exercises

- Exercise_006b_AX
- Exercise_020a_AX
- Exercise_020b_AX
- Exercise_020d_AX

Conclusion

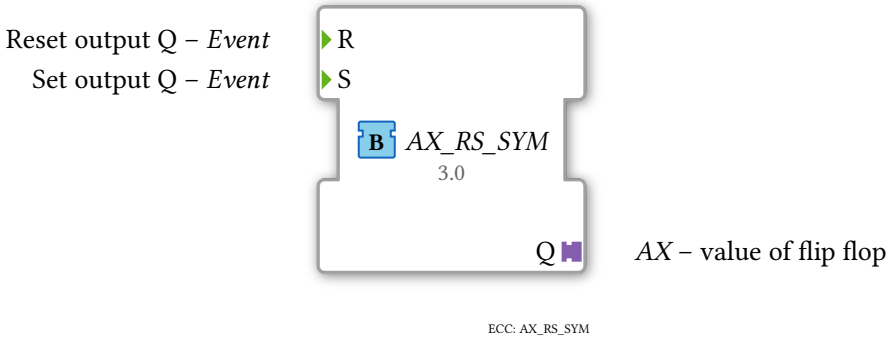
The AX_RS function block provides a simple and efficient implementation of an RS flip-flop for 4diac-based Control systems. Using adapters, it enables flexible integration into various system architectures and is particularly suitable for applications requiring reliable state storage with event-driven updates.

🌐 Related topic subpages on ms-muc-docs.de

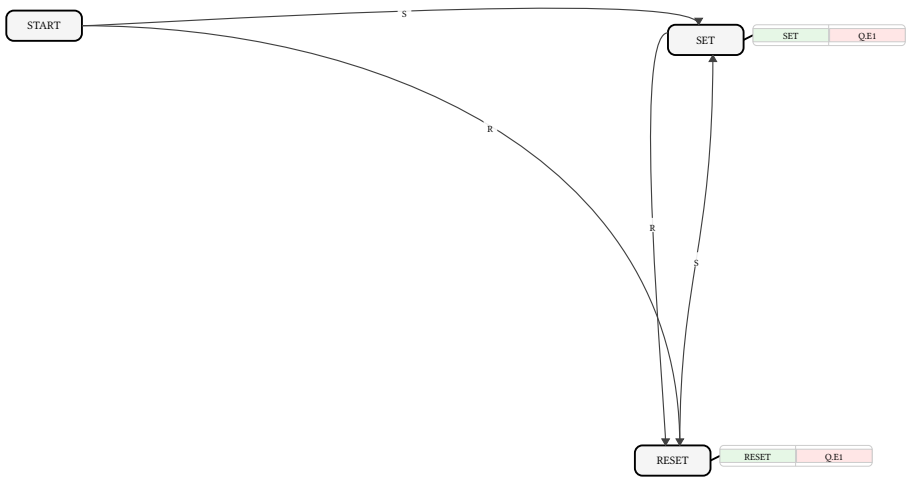
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AX_RS_SYM



ECC: AX_RS_SYM



• – ○ • – ○ • – ○ •

Introduction

The **AX_RS_SYM** is an event-driven, bistable function block (RS flip-flop) with symmetrical start-up behavior. It has two event inputs, `R` (Reset) and `S` (Set), each of which sets the state of the output `Q`. The output is provided as a unidirectional adapter, through which both a data value (D1) and an event (E1) can be transmitted.

Interface Structure

Event Inputs

Name	Type	Comment
R	Event	Sets output Q.D1 to FALSE and fires Q.E1
S	Event	Sets output Q.D1 to TRUE and fires Q.E1

Event Outputs

The function block (FB) does not have its own event outputs. The event is output via the adapter Q using Q.E1.

Data Inputs

None.

Data Outputs

The FB has no separate data outputs. The current state is provided via the adapter Q as Q.D1 (of type BOOL).

Adapters

Name	Type	Comment
Q	adapter::types::unidirectional::AX	Provides the flip-flop value (D1) and an acknowledgment event (E1)

Functionality

The function block implements a classic RS flip-flop:

- Upon arrival of the event S, the output value Q.D1 is set to TRUE, and then the event Q.E1 is triggered.
- Upon arrival of the event R, the output value Q.D1 is set to FALSE, and Q.E1 is also triggered.

The special feature is the **symmetrical start-up behavior**: Immediately after initialization, the function block is in the state START. If the first event received is S, it transitions to state SET; if R is received, it transitions to state RESET. This ensures a defined state from the start – there is no undefined or random initial value.

Technical Features

- **Symmetrical Start Behavior**: Both inputs, R and S, are treated equally. The function block has no preferred initial state; the first incoming pulse determines the state.
- **Adapter-Based Output**: The output is implemented as a unidirectional adapter. This allows for flexible reuse of the value and the event in more complex structures.
- **No undefined state**: The state machine with three states (START, SET, RESET) prevents the function block from remaining in an unknown or metastable state after startup.

State Overview

The internal state machine consists of the following states and transitions:

State	Meaning	Action on entry
START	Initial state after power-on	No action (waits for first event)
SET	Output Q.D1 is TRUE	Executes algorithm SET: Q.D1 := TRUE and fires Q.E1
RESET	Output Q.D1 is FALSE	Executes algorithm RESET: Q.D1 := FALSE and fires Q.E1

` **Transitions:**

- START ⟶ SET on condition S
- START ⟶ RESET on condition R
- SET ⟶ RESET on condition R
- RESET ⟶ SET on condition S

Application Scenarios

- **Simple Switch/Button**: One button press sets a marker bit, a second button press It resets.
- **Interlocks** (e.g., in safety controllers): A set pulse activates an action, a reset pulse deactivates it.
- **State control in sequence sequences**: As a memory block for events that are evaluated later.
- **Start-up behavior**: Particularly suitable when the system should start in a defined state after power-up, but one not yet specified by pre-logic.

Comparison with similar blocks

- **AX_RS**: A frequently used RS flip-flop with asynchronous set/reset. Unlike `AX_RS_SYM`, most implementations define a preferred state (e.g., the "R" signal resets the output at startup). `AX_RS_SYM` handles both events symmetrically, which makes the start-up behavior predictable.
- **AX_SR (Set-dominant)** or **AX_RS (Reset-dominant)**: These function blocks have a dominant input order when S and R arrive simultaneously. However, the symmetry here only applies to the startup behavior; during operation, the order is determined by the event processing of the runtime environment in the case of simultaneous events – the function block itself cannot handle simultaneity.
- **SR flip-flops in IEC 61131-3**: (e.g., `SR` or `RS`) usually operate with static signals. `AX_RS_SYM` is purely event-driven and therefore optimized for event-oriented controllers (e.g., 4diac IDE with FORTE).

Conclusion

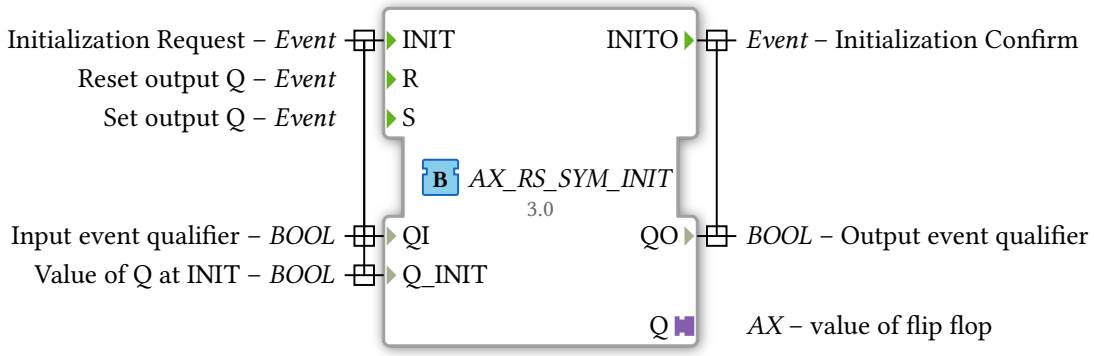
The **AX_RS_SYM** is a simple yet flexible event-driven RS flip-flop. Its unique feature lies in its symmetrical startup behavior, which allows for a defined initial state without a fixed preset. The adapter interface facilitates integration into larger systems. This component is the ideal choice for applications where neither setting nor resetting is preferred after startup.

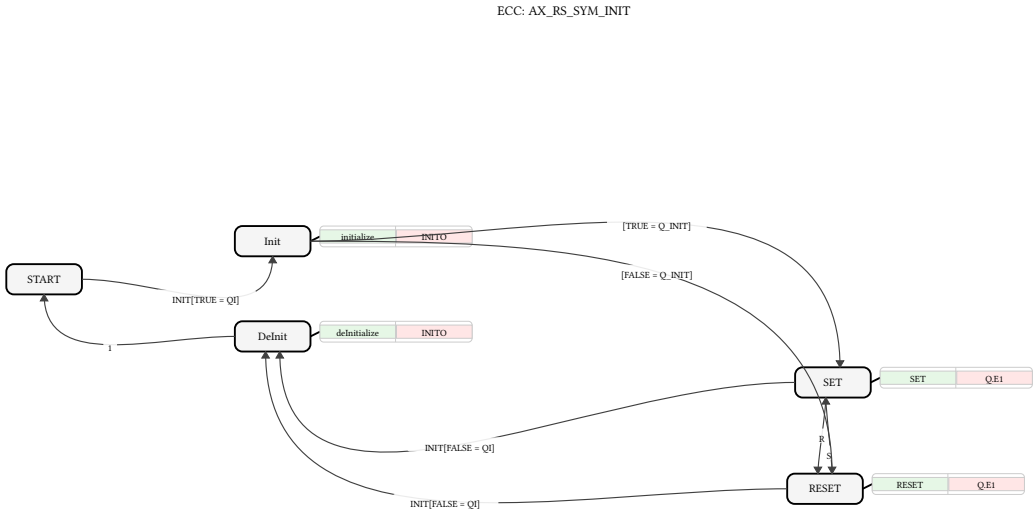
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AX_RS_SYM_INIT





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_RS_SYM_INIT** implements an event-driven, bistable flip-flop (RS flip-flop) with symmetrical startup behavior and an explicit initialization mechanism. It extends a simple RS flip-flop by adding the ability to assume a predefined state at startup (INIT) and to control both initialization and deinitialization via the INIT event input. The actual state of the flip-flop is provided via an adapter, while the data outputs QO and the adapter output Q.D1 are synchronized.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request (with QI, Q_INIT)
R	Event	Reset Signal (resets Q)
S	Event	Set Signal (sets Q)

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Acknowledgement (with QO)

Data Inputs

Name	Type	Comment
QI	BOOL	Event Qualifier – Controls Execution
Q_INIT	BOOL	Desired Value of Q at Initialization

Data Outputs

Name	Type	Comment
QO	BOOL	Event Qualifier – passes QI to INITO or on S/R

Adapter

Name	Type	Comment
Q	adapter::types::unidirectional::AX	Adapter for the value of the flip-flop (D1)

Functionality

The function block cycles through various states, controlled by the INIT, S, and R events.

- In the **START** state, it waits for an INIT event.
- When INIT is received with `QI = TRUE`, it switches to the **INIT** state. The action `initialize` sets `QO := QI`.
- Depending on the value of `Q_INIT`:
- If `Q_INIT = TRUE` ⟶ transition to the **SET** state.
- If `Q_INIT = FALSE` ⟶ transition to the **RESET** state.
- After the action, `INITO` is triggered.
- If INIT is received with `QI = FALSE` (regardless of whether it's from **SET** or **RESET**), it transitions to the **DeInit** state. The action `deInitialize` sets `QO := FALSE`. Then, `INITO` is triggered, and the function block (FB) returns to **START**.
- In the **SET** and **RESET** states, the FB behaves like a conventional RS flip-flop:
- On **S** event, the action `SET` is executed: `QO := QI`; if `QI = TRUE`, `Q.D1 := TRUE` is also set (adapter output).
- On **R** event, the action `RESET` is executed: `QO := QI`; if `QI = TRUE`, `Q.D1 := FALSE` is set.
- The event at `Q.E1` is sent after each action.

The `QI` variable acts as a qualifier: Only if `QI = TRUE` is present are the operations on the adapter (setting/resetting `Q.D1`) actually performed. Otherwise, only `QO` is passed on.

Technical Features

- **Symmetrical Startup Behavior**: Initialization takes the value of `Q_INIT` into account, allowing the flip-flop to start in either the SET or RESET state. The behavior is symmetrical because both possibilities are explicitly implemented.
- **Adapter Coupling**: The flip-flop's state is not directly output as a data output, but is provided via a **unidirectional adapter** (`Q`) with output `D1`. This enables loose coupling with other function blocks.
- **Deinitialization**: INIT with `QI = FALSE` resets the function block and returns it to the START state. All internal variables are set to their defined values (`QO = FALSE`).
- **Event Qualifiers**: `QI` and `QO` control event propagation and provide an additional conditional level.

State Overview

State	Description
START	Wait for INIT event
INIT	Initialization: Sets QO and selects target state
DEINIT	Deinitialization: Sets QO = FALSE
SET	Active set state (Q.D1 = TRUE)
RESET	Active reset state (Q.D1 = FALSE)

Important Transitions:

- START ⟶ Initialize on INIT with QI = TRUE
- Initialize ⟶ SET on Q_INIT = TRUE
- Initialize ⟶ RESET on Q_INIT = FALSE
- SET ⟶ RESET on R
- RESET ⟶ SET on S

- SET/RESET ⟶ DeInit on INIT with QI = FALSE
- DeInit ⟶ START (automatic after completion)

Application Scenarios

- **Startup Control**: A flip-flop that should assume a defined initial state after a start or reset (e.g., a machine starts in the "Ready" state).
- **Safe Initialization**: In safety-critical systems where the start state must be explicitly specified.
- **Modular Design**: Used as a fundamental storage element in an adapter-based architecture where the state is passed to other components via the adapter.

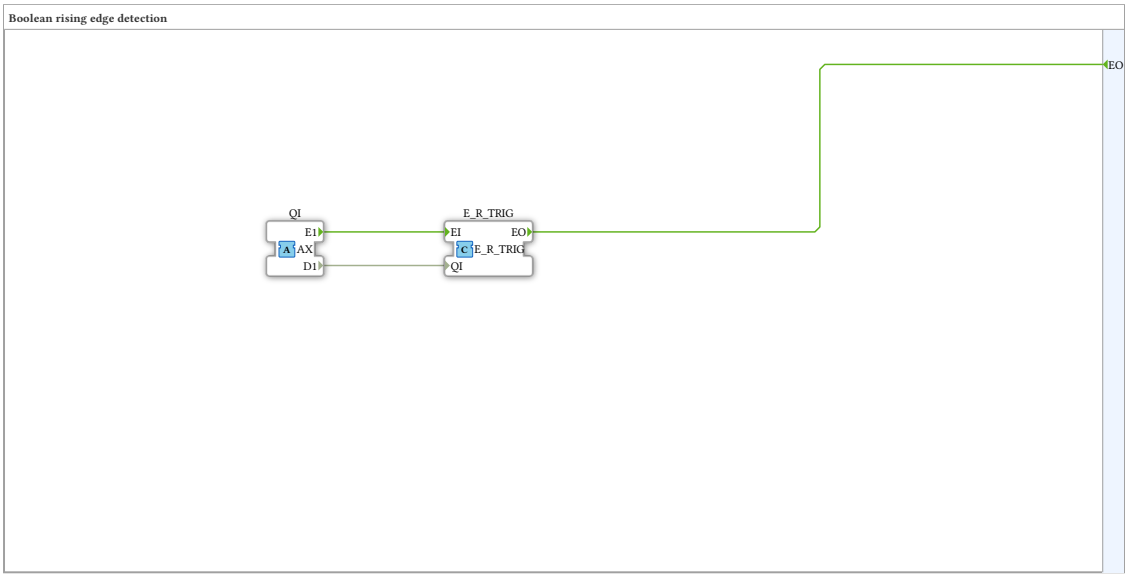
Comparison with Similar Function Blocks

- **Standard RS (e.g., RS Flip-Flop)**: Simple RS flip-flop without INIT and without an adapter. `AX_RS_SYM_INIT` extends this with initialization logic and an adapter interface.
- **Bistable Flip-Flops with Reset**: Many function blocks only offer asynchronous reset. This function block integrates the reset into the event-driven behavior and allows for symmetrical target setting.
- **SR Latch with INIT**: Similar function blocks exist, but usually without the qualifier `QI` and the deinitialization option.

Conclusion

The function block `AX_RS_SYM_INIT` combines classic RS flip-flop behavior with flexible initialization logic that allows both setting and resetting at startup. The use of an adapter for the state value and the inclusion of qualifiers (`QI`, `QO`) make it a robust and versatile building block for event-driven automation applications that require a defined and resettable initial state.

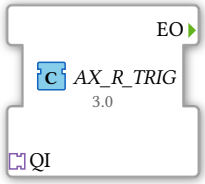
AX_R_TRIG



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AX_R_TRIG** (Boolean rising edge detection) is used to detect a rising edge based on a Boolean signal provided via an adapter. It is specifically designed for use with unidirectional adapters of type `AX` and only forwards an event if the transmitted data value is `TRUE`.



Event – confirmation that an rising edge was detected

value to check for a rising edge – *AX*

Interface Structure

The function block primarily uses an adapter interface for communicating input data and events.

Event Inputs

This function block does not have direct event inputs. Events are received via the adapter `QI`.

Event Outputs

- **EO**: (Event Output) Confirmation that a rising edge (or a high signal) has been detected.

Data Inputs

This function block has no direct data inputs. Data is received via the adapter `QI`.

Data Outputs

This function block has no direct data outputs.

Adapters

- **QI** (Socket): Of type `adapter::types::unidirectional::AX`. This adapter provides the value to be checked (Boolean) and the corresponding event.
- Typically expects an event (e.g., `E1`) and a Boolean value (e.g., `D1`).

Functionality

The component is implemented as a Composite Function Block (CFB) and is based on the internal circuitry of a D flip-flop (`E_D_FF`) and an event switch (`E_SWITCH`).

1. As soon as an event is received via the **QI** adapter, it is routed to the clock input of the internal `E_D_FF`.
2. Simultaneously, the Boolean data value is loaded from the adapter into the flip-flop.
3. The flip-flop passes the stored value to a `E_SWITCH`.
4. If the value of `TRUE` is high, the `E_SWITCH` switches on and triggers the output event **EO**.
5. If the value `FALSE` (Low), the event is blocked and **EO** is not triggered.

Technically, the function block acts as a gate that only allows the adapter event to pass if the adapter data is `TRUE`. In an event-driven system that only sends when values change, this corresponds to the logic of edge detection (0 -> 1).

Technical Features

- **Internal Structure:** Uses standard IEC 61499 function blocks (`E_D_FF`, `E_SWITCH`).
- **Adapter-based:** Unlike classic trigger blocks, which have separate `CLK` and `QI` inputs, this block encapsulates the signals in a `AX` adapter.

State Overview

Since this is a Composite Function Block, it does not have its own state machine (ECC). Its behavior is determined solely by the data flow and the interconnection of its internal sub-blocks.

Application Scenarios

- **Signal Filtering:** Forwarding of events only when a specific state is active.
- **Adapter Logic:** Use in systems that rely heavily on adapter connections to minimize wiring complexity.
- **Start Trigger:** Triggers processes as soon as a Boolean flag within an adapter structure changes to `TRUE`.

⚖️ Comparison with similar function blocks

- **E_R_TRIG:** The standard function block for event-based rising edge detection with separate event and data inputs. `AX_R_TRIG` is the equivalent for adapter connections.
- **R_TRIG:** The classic PLC function block (IEC 61131-3) that is polled cyclically. `AX_R_TRIG`, on the other hand, operates purely event-driven.

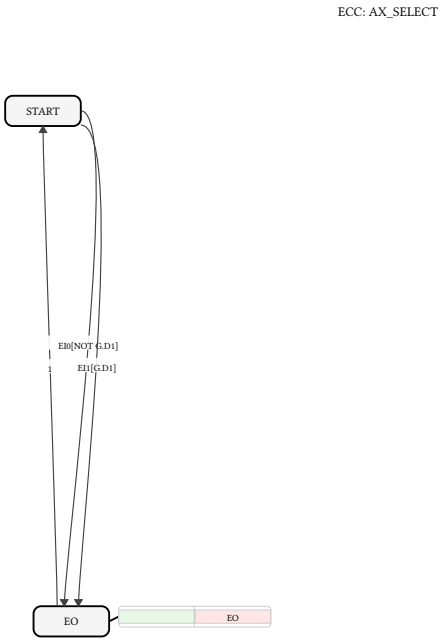
Conclusion

The **AX_R_TRIG** is a specialized component for the 4diac environment that seamlessly integrates the logic of rising edge detection or a high-pass filter into designs using `AX` adapters. It simplifies the network by eliminating the need to explicitly split adapter signals for testing.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AX_SELECT

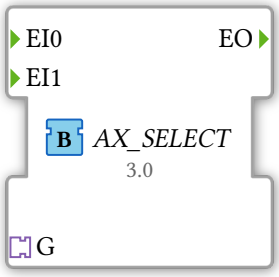


• – ○ • – ○ • – ○ •

Introduction

The AX_SELECT function block, based on a Boolean input, switches one of two AX adapter inputs to the output.

Input event, selected when G=0 – *Event*
Input event, selected when G=1 – *Event*



Event – Output Event

Select EI0 when G=0, EI1 when G=1 – *AX*

Interface Structure

Event Inputs

- **EI0**: Input event, selected when G = 0.
- **EI1**: Input event, selected when G = 1.

Event Outputs

- **EO**: Output event.

Adapters

Sockets (Inputs):

- **G** (adapter::types::unidirectional::AX): Selection signal. EI0 is selected when G = 0, EI1 when G = 1.

Functionality

If G = 0, the event EI0 is output as EO.
If G = 1, the event EI1 is output as EO.

Technical Features

- Uses unidirectional adapters.

State Overview

Combinatorial.

Application Scenarios

Selection of signals.

⚖️ Comparison with Similar Function Blocks

- **E_SELECT**: Standard event select without adapters.

🛠️ Related Exercises

- Exercise_095_AX

Conclusion

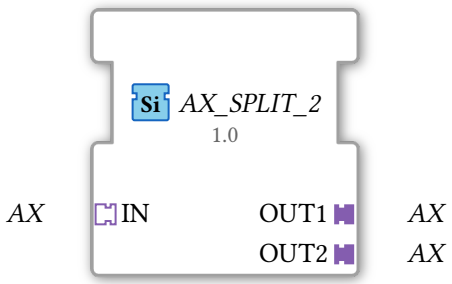
Adapter-based select function block.

AX_SPLIT_2

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_2 function block serves as a generic building block for distributing an AX signal to two separate outputs. The block enables the splitting of an incoming AX signal to two independent output channels.



Interface Structure

Event Inputs

No direct event inputs available

Event Outputs

No direct event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN**: AX adapter (unidirectional) - Receives the incoming AX signal

Output Adapters:

- **OUT1**: AX adapter (unidirectional) - First output channel for the distributed signal
- **OUT2**: AX adapter (unidirectional) - Second output channel for the distributed signal

Functionality

The AX_SPLIT_2 function block receives an AX signal via the IN adapter and simultaneously distributes this signal to both output adapters OUT1 and OUT2. OUT2. This is a 1:2 distribution, where the incoming signal is passed on to both outputs without modification.

Technical Features

- Generic implementation for AX signals
- Unidirectional signal transmission
- No signal delay between input and output
- Simultaneous activation of both outputs

State Overview

The function block operates statelessly – with every incoming signal via the IN adapter, both output adapters are immediately activated.

Application Scenarios

- Signal distribution in control systems
- Parallel supply of multiple components with the same signal
- Branching of AX communication paths
- Redundant signal routing

⚖️ Comparison with Similar Function Blocks

Compared to other distribution blocks, AX_SPLIT_2 offers a specific 1:2 split for AX signals. Other splitter blocks may support different numbers of outputs or other signal types.

Comparison with E_SPLIT

🛠️ Related exercises

- Uebung_002_AX
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_006a3_sub_AX
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- Uebung_010c_AX
- Uebung_020c3_AX
- Uebung_020e2_AX
- Uebung_020f2_AX
- AX_SPLIT_TO_2x_AX_TP
- Exercise_020j_AX
- Exercise_035a2_AX
- Exercise_035a3_AX
- Exercise_094a_AX
- Exercise_160_AX
- Exercise_160b2_AX
- Exercise_160b_AX

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AX_SPLIT_2 function block provides a simple and efficient solution for distributing AX signals to two outputs. Its generic nature and unidirectional architecture make it a versatile building block. in distributed automation systems.

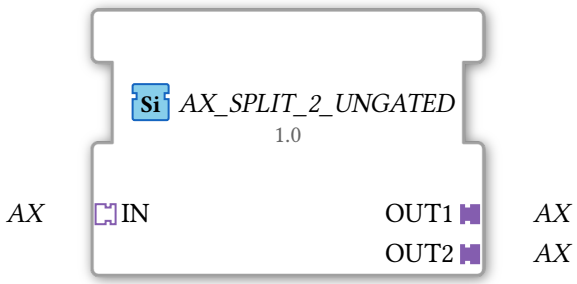
AX_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AX_SPLIT_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_2_UNGATED function block serves as a generic building block for distributing an AX signal to two separate outputs. The block enables the splitting of an incoming AX signal to two independent output channels.



Interface Structure

Event Inputs

No direct event inputs available

Event Outputs

No direct event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN**: AX adapter (unidirectional) - Receives the incoming AX signal

Output Adapters:

- **OUT1**: AX adapter (unidirectional) - First output channel for the distributed signal
- **OUT2**: AX adapter (unidirectional) - Second output channel for the distributed signal

Functionality

The AX_SPLIT_2_UNGATED function block receives an AX signal via the IN adapter and simultaneously distributes this signal to both output adapters OUT1 and OUT2. OUT2. This is a 1:2 distribution, where the incoming signal is passed on to both outputs without modification.

Technical Features

- Generic implementation for AX signals
- Unidirectional signal transmission
- No signal delay between input and output
- Simultaneous activation of both outputs

State Overview

The function block operates statelessly – with every incoming signal via the IN adapter, both output adapters are immediately activated.

Application Scenarios

- Signal distribution in control systems
- Parallel supply of multiple components with the same signal
- Branching of AX communication paths
- Redundant signal routing

⚖️ Comparison with Similar Function Blocks

Compared to other distribution blocks, AX_SPLIT_2_UNGATED offers a specific 1:2 split for AX signals. Other splitter blocks may support different numbers of outputs or other signal types.

Comparison with E_SPLIT

🛠️ Related exercises

- Uebung_002_AX
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_006a3_sub_AX
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- Uebung_010c_AX
- Uebung_020c3_AX
- Uebung_020e2_AX
- Uebung_020f2_AX
- AX_SPLIT_TO_2x_AX_TP
- Exercise_020j_AX
- Exercise_035a2_AX
- Exercise_035a3_AX
- Exercise_094a_AX
- Exercise_160_AX
- Exercise_160b2_AX
- Exercise_160b_AX

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

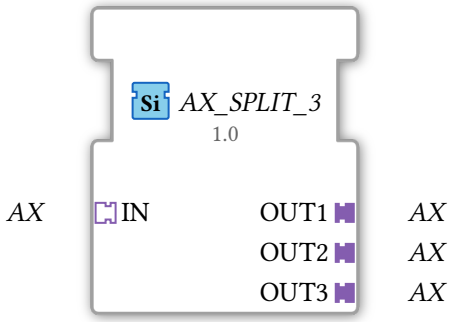
The AX_SPLIT_2_UNGATED function block provides a simple and efficient solution for distributing AX signals to two outputs. Its generic nature and unidirectional architecture make it a versatile building block. in distributed automation systems.

AX_SPLIT_3

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_3 is a generic function block that splits one AX adapter input into three separate AX adapter outputs. This block allows the distribution of an incoming AX signal to three different receivers within a 4diac system.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

Input Adapters:

- **IN** - AX adapter (socket) - Receives the incoming AX signal

Output Adapters:

- **OUT1** - AX adapter (plug) - First output channel
- **OUT2** - AX adapter (plug) - Second output channel
- **OUT3** - AX adapter (plug) - Third output channel

Functionality

The functional block acts as a signal distributor for unidirectional AX adapters. Every incoming signal at the IN adapter is forwarded in parallel to all three output adapters (OUT1, OUT2, OUT3). The distribution is synchronous, so all outputs are activated simultaneously.

Technical Features

- Uses unidirectional AX adapters for communication
- Implemented as a generic function block (GEN_AX_SPLIT)
- No event or data inputs - operates exclusively via adapters
- Plug-and-socket architecture according to IEC 61499 standard

State Overview

The function block has a simple state: In the operating state, it forwards incoming signals unchanged to all three outputs. There are no internal state transitions or delays.

Application Scenarios

- Distribution of control signals to multiple actuators
- Parallel control of multiple devices with the same signal
- Signal branching in complex control architectures

- Redundant signal distribution for safety applications

⚖️ Comparison with similar function blocks

Compared to other distribution function blocks, AX_SPLIT_3 stands out due to its specific focus on AX adapters. While general distribution function blocks can support various adapter types, this function block is specifically optimized for AX adapters.

Comparison with E_SPLIT

🛠️ Related Exercises

- Exercise_002a5b_AX
- Exercise_006a3_AX
- Exercise_009_AX

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AX_SPLIT_3 offers a simple and efficient solution for signal distribution in 4diac systems. Its generic implementation and standards-compliant adapter interface make it a versatile component for various automation applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

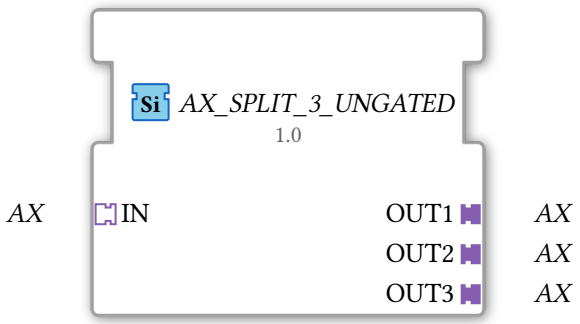
AX_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AX_SPLIT_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_3_UNGATED is a generic function block that splits one AX adapter input into three separate AX adapter outputs. This block allows the distribution of an incoming AX signal to three different receivers within a 4diac system.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

Input Adapters:

- **IN** - AX adapter (socket) - Receives the incoming AX signal

Output Adapters:

- **OUT1** - AX adapter (plug) - First output channel
- **OUT2** - AX adapter (plug) - Second output channel
- **OUT3** - AX adapter (plug) - Third output channel

Functionality

The functional block acts as a signal distributor for unidirectional AX adapters. Every incoming signal at the IN adapter is forwarded in parallel to all three output adapters (OUT1, OUT2, OUT3). The distribution is synchronous, so all outputs are activated simultaneously.

Technical Features

- Uses unidirectional AX adapters for communication
- Implemented as a generic function block (GEN_AX_SPLIT)
- No event or data inputs - operates exclusively via adapters
- Plug-and-socket architecture according to IEC 61499 standard

State Overview

The function block has a simple state: In the operating state, it forwards incoming signals unchanged to all three outputs. There are no internal state transitions or delays.

Application Scenarios

- Distribution of control signals to multiple actuators
- Parallel control of multiple devices with the same signal
- Signal branching in complex control architectures
- Redundant signal distribution for safety applications

⚖️ Comparison with similar function blocks

Compared to other distribution function blocks, AX_SPLIT_3_UNGATED stands out due to its specific focus on AX adapters. While general distribution function blocks can support various adapter types, this function block is specifically optimized for AX adapters.

Comparison with E_SPLIT

🛠️ Related Exercises

- Exercise_002a5b_AX
- Exercise_006a3_AX
- Exercise_009_AX

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AX_SPLIT_3_UNGATED offers a simple and efficient solution for signal distribution in 4diac systems. Its generic implementation and standards-compliant adapter interface make it a versatile component for various automation applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

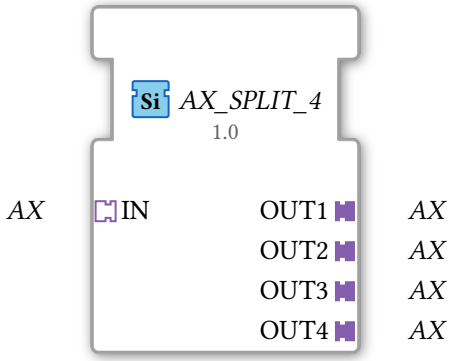
]

AX_SPLIT_4

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_4 function block is a generic function block that splits one AX adapter input into four separate AX adapter outputs. The block acts as a distributor for unidirectional AX adapters and enables the transmission of data and events to multiple receivers.



Interface Structure

Event Inputs

No direct event inputs available (event processing is handled via adapters)

Event Outputs

No direct event outputs available (event processing is handled via adapters)

Data Inputs

No direct data inputs available (data processing is handled via adapters)

Data Outputs

No direct data outputs available (data processing is handled via adapters)

Adapters

Input Adapter:

- `IN` - AX Adapter (Socket) - Receives unidirectional AX data and events

Output Adapter:

- `OUT1` - AX Adapter (Plug) - First output channel
- `OUT2` - AX Adapter (Plug) - Second Output Channel
- `OUT3` - AX Adapter (Plug) - Third Output Channel
- `OUT4` - AX Adapter (Plug) - Fourth Output Channel

Functionality

The AX_SPLIT_4 block receives data and events via the input adapter `IN` and distributes them in parallel to all four output adapters (`OUT1` to `OUT4`). All incoming information is forwarded to all outputs simultaneously, thus achieving a 1:4 distribution.

Technical Features

- Generic implementation for maximum reusability
- Uses unidirectional AX adapters for data transmission
- No buffering or delay in data transmission

- Parallel distribution without output prioritization

State Overview

The function block operates statelessly – incoming data and events are immediately forwarded to all outputs without storing any internal state.

Application Scenarios

- Distribution of control commands to multiple actuators
- Broadcasting of sensor data to various processing units
- Distribution of control information in distributed systems
- Redundant data distribution for safety applications

⚖️ Comparison with Similar Function Blocks

Compared to simple split function blocks, AX_SPLIT_4 offers a specific 1:4 split for AX adapters. Other split variants may support different numbers of outputs or other adapter types.

Comparison with E_SPLIT

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AX_SPLIT_4 function block offers a simple and efficient solution for distributing AX adapter data to four parallel outputs. Its generic nature and the use of standardized adapters make it versatile for use in various automation applications.

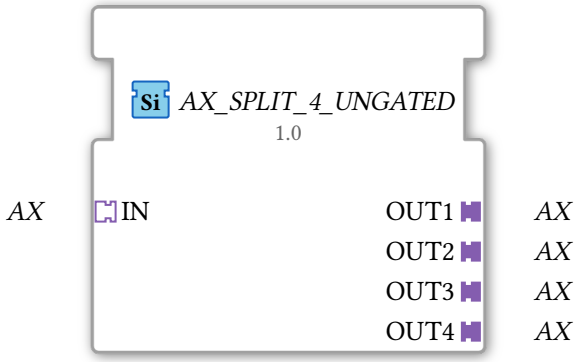
AX_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_4_UNGATED function block is a generic function block that splits one AX adapter input into four separate AX adapter outputs. The block acts as a distributor for unidirectional AX adapters and enables the transmission of data and events to multiple receivers.



Interface Structure

Event Inputs

No direct event inputs available (event processing is handled via adapters)

Event Outputs

No direct event outputs available (event processing is handled via adapters)

Data Inputs

No direct data inputs available (data processing is handled via adapters)

Data Outputs

No direct data outputs available (data processing is handled via adapters)

Adapters

Input Adapter:

- `IN` - AX Adapter (Socket) - Receives unidirectional AX data and events

Output Adapter:

- `OUT1` - AX Adapter (Plug) - First output channel
- `OUT2` - AX Adapter (Plug) - Second Output Channel
- `OUT3` - AX Adapter (Plug) - Third Output Channel
- `OUT4` - AX Adapter (Plug) - Fourth Output Channel

Functionality

The AX_SPLIT_4_UNGATED block receives data and events via the input adapter `IN` and distributes them in parallel to all four output adapters (`OUT1` to `OUT4`). All incoming information is forwarded to all outputs simultaneously, thus achieving a 1:4 distribution.

Technical Features

- Generic implementation for maximum reusability
- Uses unidirectional AX adapters for data transmission
- No buffering or delay in data transmission
- Parallel distribution without output prioritization

State Overview

The function block operates statelessly – incoming data and events are immediately forwarded to all outputs without storing any internal state.

Application Scenarios

- Distribution of control commands to multiple actuators
- Broadcasting of sensor data to various processing units
- Distribution of control information in distributed systems
- Redundant data distribution for safety applications

⚖️ Comparison with Similar Function Blocks

Compared to simple split function blocks, AX_SPLIT_4_UNGATED offers a specific 1:4 split for AX adapters. Other split variants may support different numbers of outputs or other adapter types.

Comparison with E_SPLIT

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

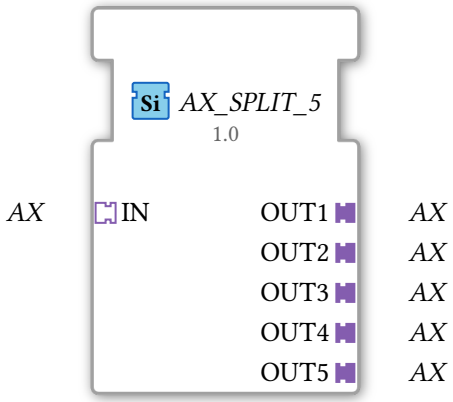
The AX_SPLIT_4_UNGATED function block offers a simple and efficient solution for distributing AX adapter data to four parallel outputs. Its generic nature and the use of standardized adapters make it versatile for use in various automation applications.

AX_SPLIT_5

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_5 function block is a generic function block used to split an AX adapter into five separate AX outputs. The block enables the distribution of an input signal to multiple output channels and is particularly useful in control applications where signals need to be distributed to different components.



Interface Structure

Event Inputs

No direct event inputs available (event processing is handled via adapters)

Event Outputs

No direct event outputs available (event processing is handled via adapters)

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN** - AX unidirectional adapter (socket)

Output Adapters:

- **OUT1** - AX unidirectional adapter (plug)
- **OUT2** - AX unidirectional adapter (plug)
- **OUT3** - AX unidirectional adapter (plug)
- **OUT4** - AX unidirectional adapter (plug)
- **OUT5** - AX Unidirectional Adapter (Plug)

Functionality

The AX_SPLIT_5 function block receives signals via the input adapter IN and simultaneously distributes them to all five output adapters (OUT1 to OUT5). Every event received at the IN adapter is forwarded to all outputs in parallel, thus achieving a 1:5 distribution.

Technical Features

- Generic function block with the class ‘GEN_AX_SPLIT’
- Uses unidirectional AX adapters for communication
- No data processing or transformation
- Pure event distribution without delay

State Overview

The block has a simple state: In its idle state, it waits for incoming events at the IN adapter. Upon receiving an event, all five output adapters are immediately activated.

Application Scenarios

- Distribution of control signals to multiple actuators
- Distribution of sensor values to different processing units
- Parallel activation of multiple subsystems
- Signal distribution in decentralized control architectures

⚖️ Comparison with similar components

Compared to simple split components, the AX_SPLIT_5 offers a fixed number of five outputs. Other split variants can offer different numbers of outputs or configurable distribution logics. The focus here is on the simple, fixed 1:5 distribution.

Comparison with E_SPLIT

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AX_SPLIT_5 is a specialized distribution component whose simple and deterministic operation makes it ideal for applications where a signal needs to be distributed to exactly five receivers. The use of standardized AX adapters ensures compatibility within existing control systems.

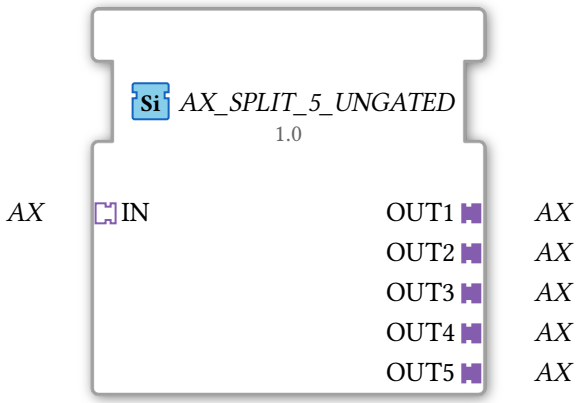
AX_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_5_UNGATED function block is a generic function block used to split an AX adapter into five separate AX outputs. The block enables the distribution of an input signal to multiple output channels and is particularly useful in control applications where signals need to be distributed to different components.



Interface Structure

Event Inputs

No direct event inputs available (event processing is handled via adapters)

Event Outputs

No direct event outputs available (event processing is handled via adapters)

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- **IN** - AX unidirectional adapter (socket)

Output Adapters:

- **OUT1** - AX unidirectional adapter (plug)
- **OUT2** - AX unidirectional adapter (plug)
- **OUT3** - AX unidirectional adapter (plug)
- **OUT4** - AX unidirectional adapter (plug)
- **OUT5** - AX Unidirectional Adapter (Plug)

Functionality

The AX_SPLIT_5_UNGATED function block receives signals via the input adapter IN and simultaneously distributes them to all five output adapters (OUT1 to OUT5). Every event received at the IN adapter is forwarded to all outputs in parallel, thus achieving a 1:5 distribution.

Technical Features

- Generic function block with the class 'GEN_AX_SPLIT'
- Uses unidirectional AX adapters for communication
- No data processing or transformation
- Pure event distribution without delay

State Overview

The block has a simple state: In its idle state, it waits for incoming events at the IN adapter. Upon receiving an event, all five output adapters are immediately activated.

Application Scenarios

- Distribution of control signals to multiple actuators
- Distribution of sensor values to different processing units
- Parallel activation of multiple subsystems
- Signal distribution in decentralized control architectures

⚖️ Comparison with similar components

Compared to simple split components, the AX_SPLIT_5_UNGATED offers a fixed number of five outputs. Other split variants can offer different numbers of outputs or configurable distribution logics. The focus here is on the simple, fixed 1:5 distribution.

Comparison with E_SPLIT

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

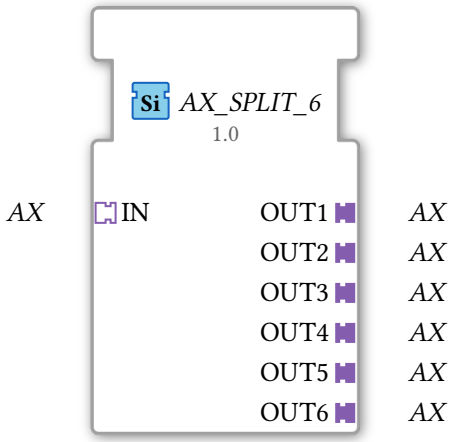
The AX_SPLIT_5_UNGATED is a specialized distribution component whose simple and deterministic operation makes it ideal for applications where a signal needs to be distributed to exactly five receivers. The use of standardized AX adapters ensures compatibility within existing control systems.

AX_SPLIT_6

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_6 function block is a generic component used to split a single AX adapter into six separate AX outputs. This component enables the distribution of signals and data streams from a central source to multiple devices.



Interface Structure

Event Inputs

No direct event inputs available

Event Outputs

No direct event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapter:

- **IN** (Socket): Receives incoming AX signals from the source system

Output Adapter:

- **OUT1** (Plug): First output channel
- **OUT2** (Plug): Second output channel
- **OUT3** (Plug): Third output channel
- **OUT4** (Plug): Fourth output channel
- **OUT5** (Plug): Fifth output channel
- **OUT6** (Plug): Sixth output channel Output Channel

All adapters use the type: `adapter::types::unidirectional::AX`

Functionality

The AX_SPLIT_6 module functions as a signal distributor. It receives signals via the input adapter IN and forwards them in parallel to all six output adapters (OUT1 to OUT6). Since these are unidirectional AX adapters, signal transmission only occurs in one direction – from the input to the outputs.

Technical Features

- Generic function block with the class 'GEN_AX_SPLIT'
- Uses unidirectional AX adapters for communication
- No data processing or modification, only signal distribution
- Parallel output to all six outputs

State Overview

The block has a simple state:

- **Ready**: Receives signals from IN and distributes them to OUT1-OUT6

Application Scenarios

- Distribution of control signals to multiple actuators
- Splitting of sensor values to different processing units
- Signal distribution in redundant systems
- Multi-casting of control commands

⚖️ Comparison with Similar Blocks

Compared to other splitter blocks, AX_SPLIT_6 offers a fixed number of six outputs, which is advantageous for specific applications. Other splitters can offer more or fewer outputs, but may be more complex to configure.

Comparison with E_SPLIT

- `AX_SPLIT_6_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AX_SPLIT_6 is a specialized distribution module ideally suited for applications where a signal needs to be distributed to exactly six receivers. Its simple structure and the use of standardized AX adapters make it a reliable solution for signal distribution tasks in automation systems.

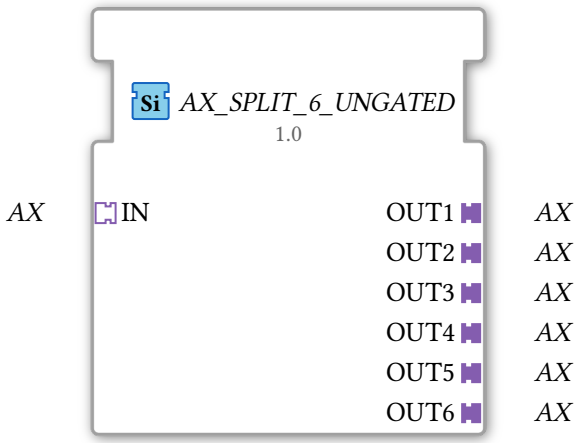
AX_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_6_UNGATED function block is a generic component used to split a single AX adapter into six separate AX outputs. This component enables the distribution of signals and data streams from a central source to multiple devices.



Interface Structure

Event Inputs
No direct event inputs available

Event Outputs
No direct event outputs available

Data Inputs
No direct data inputs available

Data Outputs
No direct data outputs available

Adapters
Input Adapter:

- **IN** (Socket): Receives incoming AX signals from the source system

Output Adapter:

- **OUT1** (Plug): First output channel
- **OUT2** (Plug): Second output channel
- **OUT3** (Plug): Third output channel
- **OUT4** (Plug): Fourth output channel
- **OUT5** (Plug): Fifth output channel
- **OUT6** (Plug): Sixth output channel Output Channel

All adapters use the type: `adapter::types::unidirectional::AX`

Functionality
The AX_SPLIT_6_UNGATED module functions as a signal distributor. It receives signals via the input adapter IN and forwards them in parallel to all six output adapters (OUT1 to OUT6). Since these are unidirectional AX adapters, signal transmission only occurs in one direction – from the input to the outputs.

Technical Features
- Generic function block with the class 'GEN_AX_SPLIT'
- Uses unidirectional AX adapters for communication
- No data processing or modification, only signal distribution
- Parallel output to all six outputs

State Overview
The block has a simple state:

- **Ready**: Receives signals from IN and distributes them to OUT1-OUT6

Application Scenarios
- Distribution of control signals to multiple actuators
- Splitting of sensor values to different processing units
- Signal distribution in redundant systems
- Multi-casting of control commands

⚖️ Comparison with Similar Blocks
Compared to other splitter blocks, AX_SPLIT_6_UNGATED offers a fixed number of six outputs, which is advantageous for specific applications. Other splitters can offer more or fewer outputs, but may be more complex to configure.

Comparison with E_SPLIT

- **`AX_SPLIT_6`**: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

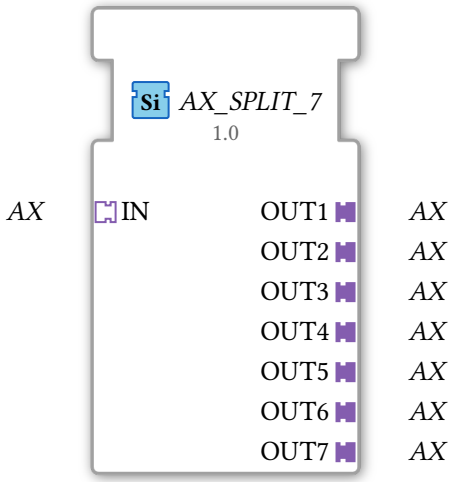
AX_SPLIT_6_UNGATED is a specialized distribution module ideally suited for applications where a signal needs to be distributed to exactly six receivers. Its simple structure and the use of standardized AX adapters make it a reliable solution for signal distribution tasks in automation systems.

AX_SPLIT_7

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_7 function block is a generic component that splits a single AX adapter input into seven separate AX adapter outputs. The component acts as a distributor for unidirectional AX adapters and allows a single input signal to be distributed to multiple receivers.



Interface Structure

Event Inputs

No direct event inputs available (adapter-based communication)

Event Outputs

No direct event outputs available (adapter-based communication)

Data Inputs

No direct data inputs available (adapter-based communication)

Data Outputs

No direct data outputs available (adapter-based communication)

Adapters

Input Adapter:

- **IN** (Socket): Unidirectional AX adapter input

Output Adapter:

- **OUT1** (Plug): Unidirectional AX adapter output 1
- **OUT2** (Plug): Unidirectional AX adapter output 2
- **OUT3** (Plug): Unidirectional AX Adapter Output 3
- **OUT4** (Plug): Unidirectional AX Adapter Output 4
- **OUT5** (Plug): Unidirectional AX Adapter Output 5
- **OUT6** (Plug): Unidirectional AX Adapter Output 6

- **OUT7** (Plug): Unidirectional AX Adapter Output 7

Functionality

The AX_SPLIT_7 function block receives data via the input adapter IN and simultaneously distributes it to all seven output adapters (OUT1 to OUT7). Each signal received at the IN adapter is forwarded in parallel to all seven outputs, thus achieving a 1:7 distribution.

Technical Features

- Generic implementation for maximum reusability
- Uses unidirectional AX adapters for communication
- No internal delays in signal distribution
- Parallel output to all seven outputs

State Overview

The function block has a simple state: In the active state, it immediately forwards incoming signals to all outputs. There are no complex state transitions or internal processing logic.

Application Scenarios

- Distribution of control signals to multiple actuators
- Splitting of sensor data to different processing units
- Signal distribution in branched control architectures
- Redundant signal forwarding to multiple receivers

⚖️ Comparison with Similar Components

Compared to simpler splitter components, AX_SPLIT_7 offers a fixed number of seven outputs, which is optimized for specific use cases with exactly seven receivers. Compared to variable splitters, this component has the advantage of fixed interfaces.

Comparison with E_SPLIT

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AX_SPLIT_7 function block provides an efficient solution for distributing unidirectional AX signals to seven receivers. Its generic nature and simple operation make it a robust and reliable component for distributed control systems.

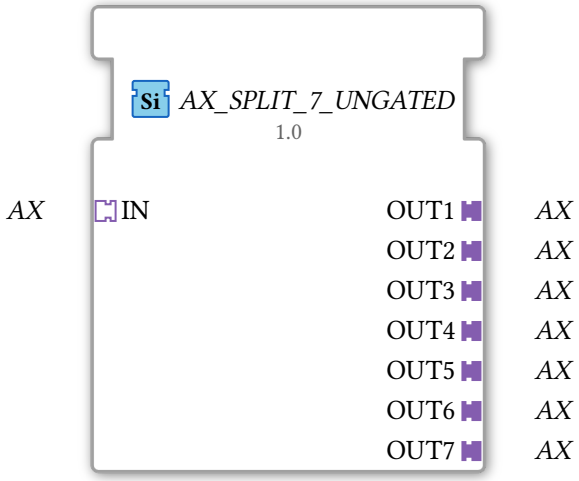
AX_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_7_UNGATED function block is a generic component that splits a single AX adapter input into seven separate AX adapter outputs. The component acts as a distributor for unidirectional AX adapters and allows a single input signal to be distributed to multiple receivers.



Interface Structure

Event Inputs

No direct event inputs available (adapter-based communication)

Event Outputs

No direct event outputs available (adapter-based communication)

Data Inputs

No direct data inputs available (adapter-based communication)

Data Outputs

No direct data outputs available (adapter-based communication)

Adapters

Input Adapter:

- **IN** (Socket): Unidirectional AX adapter input

Output Adapter:

- **OUT1** (Plug): Unidirectional AX adapter output 1
- **OUT2** (Plug): Unidirectional AX adapter output 2
- **OUT3** (Plug): Unidirectional AX Adapter Output 3
- **OUT4** (Plug): Unidirectional AX Adapter Output 4
- **OUT5** (Plug): Unidirectional AX Adapter Output 5
- **OUT6** (Plug): Unidirectional AX Adapter Output 6
- **OUT7** (Plug): Unidirectional AX Adapter Output 7

Functionality

The AX_SPLIT_7_UNGATED function block receives data via the input adapter IN and simultaneously distributes it to all seven output adapters (OUT1 to OUT7). Each signal received at the IN adapter is forwarded in parallel to all seven outputs, thus achieving a 1:7 distribution.

Technical Features

- Generic implementation for maximum reusability
- Uses unidirectional AX adapters for communication
- No internal delays in signal distribution
- Parallel output to all seven outputs

State Overview

The function block has a simple state: In the active state, it immediately forwards incoming signals to all outputs. There are no complex state transitions or internal processing logic.

Application Scenarios

- Distribution of control signals to multiple actuators
- Splitting of sensor data to different processing units
- Signal distribution in branched control architectures
- Redundant signal forwarding to multiple receivers

⚖️ Comparison with Similar Components

Compared to simpler splitter components, AX_SPLIT_7_UNGATED offers a fixed number of seven outputs, which is optimized for specific use cases with exactly seven receivers. Compared to variable splitters, this component has the advantage of fixed interfaces.

Comparison with E_SPLIT

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

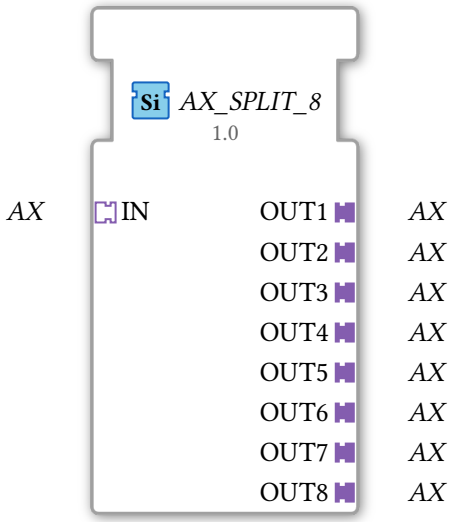
The AX_SPLIT_7_UNGATED function block provides an efficient solution for distributing unidirectional AX signals to seven receivers. Its generic nature and simple operation make it a robust and reliable component for distributed control systems.

AX_SPLIT_8

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_8 function block is a generic function block that splits a single AX adapter into eight separate AX outputs. It is used to distribute adapter signals in complex control architectures.



Interface Structure

Event Inputs

No direct event inputs available

Event Outputs

No direct event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapter:

- **IN** (Socket): AX adapter input type `adapter::types::unidirectional::AX`

Output Adapter:

- **OUT1** (Plug): AX adapter output 1 type `adapter::types::unidirectional::AX`
- **OUT2** (Plug): AX adapter output 2 type `adapter::types::unidirectional::AX`
- **OUT3** (Plug): AX adapter output 3 of type `adapter::types::unidirectional::AX`
- **OUT4** (Plug): AX adapter output 4 of type `adapter::types::unidirectional::AX`
- **OUT5** (Plug): AX adapter output 5 of type `adapter::types::unidirectional::AX`
- **OUT6** (Plug): AX adapter output 6 of type `adapter::types::unidirectional::AX`
- **OUT7** (Plug): AX adapter output 7 of type `adapter::types::unidirectional::AX`
- **OUT8** (Plug): AX adapter output 8 of type `adapter::types::unidirectional::AX`

Functionality

The AX_SPLIT_8 function block distributes incoming Adapter signals from the input adapter IN are simultaneously sent to all eight output adapters (OUT1 to OUT8). As a unidirectional adapter block, it forwards all incoming events and data in parallel to all connected outputs.

Technical Features

- Generic function block with the generic class 'GEN_AX_SPLIT'
- Uses unidirectional AX adapters
- No internal signal processing or filtering
- Parallel signal distribution without delay

State Overview

The block has no internal state and operates stateless. All incoming signals are immediately forwarded to all outputs.

Application Scenarios

- Distribution of control signals to multiple actuators
- Distribution of sensor information to different processing units
- Signal distribution in redundant system architectures
- Cascading of adapter-based control systems

⚖️ Comparison with Similar Blocks

Compared to other split blocks, AX_SPLIT_8 offers a fixed number of eight outputs and operates exclusively at the adapter level. Other split components can have a variable number of outputs or operate at the data or event level.

Comparison with E_SPLIT

- **`AX_SPLIT_8_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AX_SPLIT_8 function block provides a simple and efficient solution for distributing adapter signals in 4diac systems. Its fixed output of eight ports makes it particularly suitable for standardized distribution tasks in industrial control applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

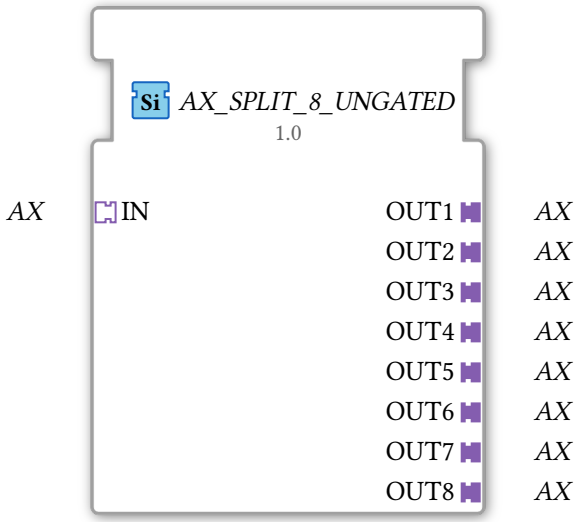
AX_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of AX_SPLIT_8. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_8_UNGATED function block is a generic function block that splits a single AX adapter into eight separate AX outputs. It is used to distribute adapter signals in complex control architectures.



Interface Structure

Event Inputs

No direct event inputs available

Event Outputs

No direct event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapter:

- **IN** (Socket): AX adapter input type `adapter::types::unidirectional::AX`

Output Adapter:

- **OUT1** (Plug): AX adapter output 1 type `adapter::types::unidirectional::AX`
- **OUT2** (Plug): AX adapter output 2 type `adapter::types::unidirectional::AX`
- **OUT3** (Plug): AX adapter output 3 of type `adapter::types::unidirectional::AX`
- **OUT4** (Plug): AX adapter output 4 of type `adapter::types::unidirectional::AX`
- **OUT5** (Plug): AX adapter output 5 of type `adapter::types::unidirectional::AX`
- **OUT6** (Plug): AX adapter output 6 of type `adapter::types::unidirectional::AX`
- **OUT7** (Plug): AX adapter output 7 of type `adapter::types::unidirectional::AX`

- **OUT8** (Plug): AX adapter output 8 of type `adapter::types::unidirectional::AX`

Functionality

The AX_SPLIT_8_UNGATED function block distributes incoming Adapter signals from the input adapter IN are simultaneously sent to all eight output adapters (OUT1 to OUT8). As a unidirectional adapter block, it forwards all incoming events and data in parallel to all connected outputs.

Technical Features

- Generic function block with the generic class 'GEN_AX_SPLIT'
- Uses unidirectional AX adapters
- No internal signal processing or filtering
- Parallel signal distribution without delay

State Overview

The block has no internal state and operates stateless. All incoming signals are immediately forwarded to all outputs.

Application Scenarios

- Distribution of control signals to multiple actuators
- Distribution of sensor information to different processing units
- Signal distribution in redundant system architectures
- Cascading of adapter-based control systems

⚖️ Comparison with Similar Blocks

Compared to other split blocks, AX_SPLIT_8_UNGATED offers a fixed number of eight outputs and operates exclusively at the adapter level. Other split components can have a variable number of outputs or operate at the data or event level.

Comparison with E_SPLIT

- **`AX_SPLIT_8`**: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AX_SPLIT_8_UNGATED function block provides a simple and efficient solution for distributing adapter signals in 4diac systems. Its fixed output of eight ports makes it particularly suitable for standardized distribution tasks in industrial control applications.

🌐 Related topic subpages on ms-muc-docs.de

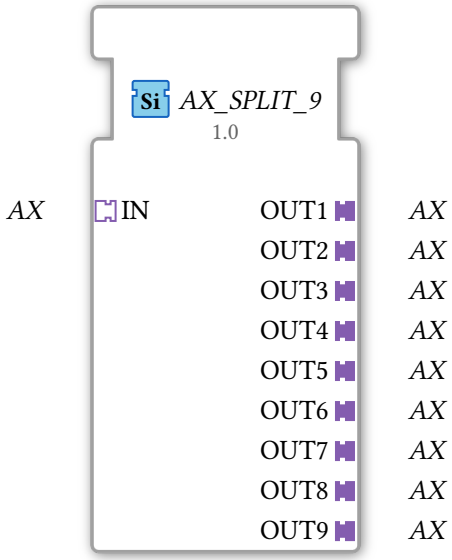
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AX_SPLIT_9

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_9 function block is a generic component that splits a single AX adapter into nine separate AX outputs. The block acts as a distributor for unidirectional adapter connections and allows a single input signal to be distributed across multiple output channels.



Interface Structure

Event Inputs

No direct event inputs available

Event Outputs

No direct event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- `IN` - Unidirectional AX adapter (socket)

Output Adapters:

- `OUT1` to `OUT9` - Nine unidirectional AX adapters (plugs)

Functionality

The AX_SPLIT_9 block functions as an adapter splitter, which receives incoming signals and data from the input adapter. The input signal is distributed in parallel to all nine output adapters (`IN` to `OUT9`). Each output receives an identical copy of the input signal.

Technical Features

- Generic implementation for maximum reusability
- Unidirectional adapter architecture
- Parallel signal distribution without delay
- No data processing or modification

State Overview

The function block has a simple state: When the input adapter is activated, all nine output adapters are activated simultaneously.

Application Scenarios

- Distribution of control signals to multiple actuators
- Distribution of sensor values to different processing units

- Signal distribution in complex control systems
- Redundant signal routing

⚖️ Comparison with Similar Blocks

Compared to simpler splitter blocks, AX_SPLIT_9 offers a higher number of outputs (9 instead of the typical 2-4). Compared to serial distributors, it allows simultaneous activation of all outputs without sequential delay.

Comparison with E_SPLIT

- **AX_SPLIT_9_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AX_SPLIT_9 function block represents an efficient solution for the parallel distribution of unidirectional adapter signals. Its generic nature and the high number of outputs make it particularly suitable for complex control applications where a signal needs to be distributed to multiple receivers.

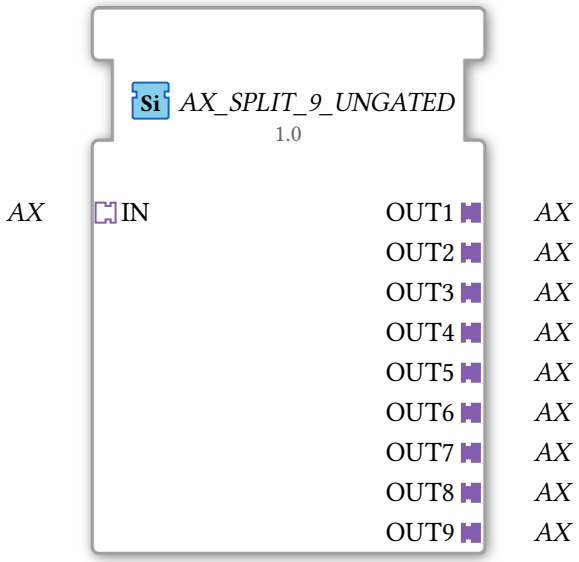
AX_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of AX_SPLIT_9. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SPLIT_9_UNGATED function block is a generic component that splits a single AX adapter into nine separate AX outputs. The block acts as a distributor for unidirectional adapter connections and allows a single input signal to be distributed across multiple output channels.



Interface Structure

Event Inputs

No direct event inputs available

Event Outputs

No direct event outputs available

Data Inputs

No direct data inputs available

Data Outputs

No direct data outputs available

Adapters

Input Adapters:

- `IN` - Unidirectional AX adapter (socket)

Output Adapters:

- `OUT1` to `OUT9` - Nine unidirectional AX adapters (plugs)

Functionality

The AX_SPLIT_9_UNGATED block functions as an adapter splitter, which receives incoming signals and data from the input adapter. The input signal is distributed in parallel to all nine output adapters (`IN` to `OUT9`). Each output receives an identical copy of the input signal.

Technical Features

- Generic implementation for maximum reusability
- Unidirectional adapter architecture
- Parallel signal distribution without delay
- No data processing or modification

State Overview

The function block has a simple state: When the input adapter is activated, all nine output adapters are activated simultaneously.

Application Scenarios

- Distribution of control signals to multiple actuators
- Distribution of sensor values to different processing units
- Signal distribution in complex control systems
- Redundant signal routing

⚖️ Comparison with Similar Blocks

Compared to simpler splitter blocks, AX_SPLIT_9_UNGATED offers a higher number of outputs (9 instead of the typical 2-4). Compared to serial distributors, it allows simultaneous activation of all outputs without sequential delay.

Comparison with E_SPLIT

- **`AX_SPLIT_9`**: The gated variant – updates the output only on an actual value change.

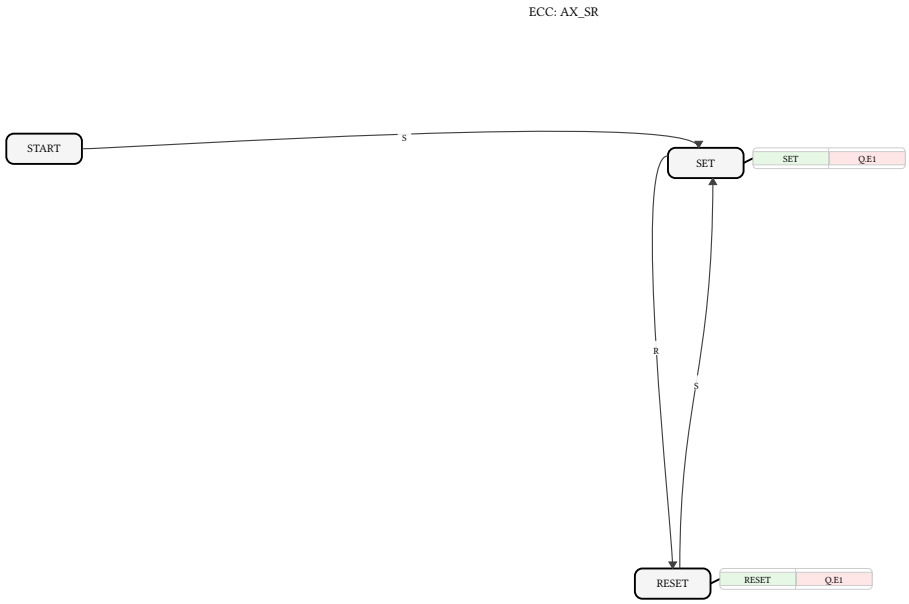
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AX_SPLIT_9_UNGATED function block represents an efficient solution for the parallel distribution of unidirectional adapter signals. Its generic nature and the high number of outputs make it particularly suitable for complex control applications where a signal needs to be distributed to multiple receivers.

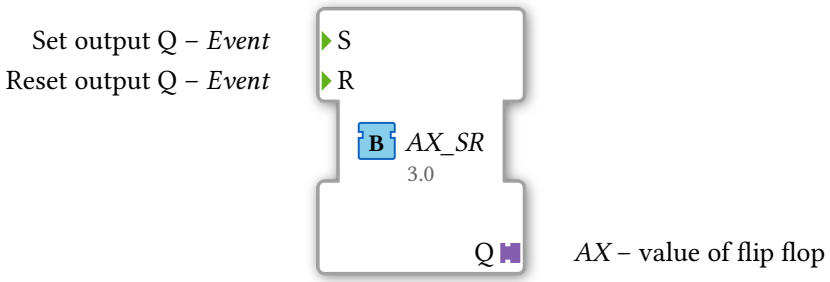
AX_SR



• – ◦ • – ◦ • – ◦ •

Introduction

The AX_SR function block is an event-driven bistable element (flip-flop) that functions as a set-reset memory. It allows setting and resetting a logical state and makes this state available via an adapter interface.



Interface Structure

Event Inputs

- **S**: Sets output Q to TRUE
- **R**: Sets output Q to FALSE

Event Outputs

- No direct event outputs available

Data Inputs

- No data inputs available

Data Outputs

- No direct data outputs available

Adapters

- **Q**: Unidirectional adapter of type “adapter::types::unidirectional::AX”, which provides the current state of the flip-flop

Functionality

The AX_SR function block operates as a set-reset flip-flop with the following properties:

- On an S event, the internal state is set to SET and output Q to TRUE
- On an R event, the internal state is set to RESET and output Q to FALSE
- The state Remains in state until a contrary event occurs.
- State changes are communicated via adapter Q.

Technical Features and Comparison with Standards

As with all event-driven bistable elements in IEC 61499 (see also Note 8 in Table A.1 of DIN EN 61499-1), there is no inherent “dominance” of one input, as is known from IEC 61131-3.

- **Comparison to IEC 61131-3**: See SR (Bistable, set with priority). While in the classic PLC world, if S1 and R are TRUE simultaneously, the set takes precedence, in IEC 61499, each event is processed sequentially. The final state depends on which event was processed last in the execution chain (ECC).
- **Functional Identity**: `AX_SR` is functionally identical to AX_RS. The different naming and pin arrangement are solely for the convenience of developers familiar with IEC 61131-3.
- **Adapter Communication**: The device makes its status available via the adapter `Q`.

State Overview

1. **START**: Initial state
2. **SET**: State after S-event, output Q.D1 = TRUE
3. **RESET**: State after R-event, output Q.D1 = FALSE

State Transitions:

- START ⟶ SET: On S-event
- SET ⟶ RESET: On R-event
- RESET ⟶ SET: On S-event

Application Scenarios

- Storage of switching states in distributed control applications
- State management via adapter interfaces
- Signal processing with memory function
- Monitoring of operating states

Related Function Blocks

- **AX_RS**: Functionally identical, inputs reversed in the symbol.
- **E_SR**: The standard equivalent with direct data/event outputs instead of adapters.

⚖️ Comparison with Similar Building Blocks

Compared to other memory elements, AX_SR offers:

- Clear separation of set and reset functionality
- Adapter-based interface for flexible integration
- Event-driven state changes
- Simple and robust state management

Comparison with E_SR

🛠️ Related Exercises

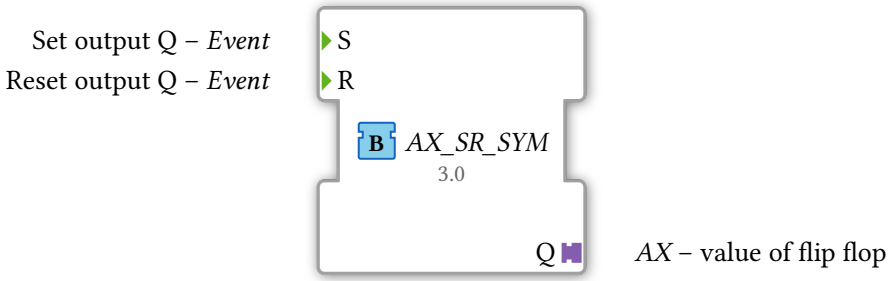
- Exercise_004b_AX
- Exercise_004b_AX_ASR
- Exercise_004b_AX_ASR_X
- Exercise_006_AX
- Exercise_006d_AX
- Exercise_007a3_AX
- Exercise_008_AX

- Exercise_009_AX
- Exercise_013_AX
- Exercise_160b_AX
- Exercise_171_AX

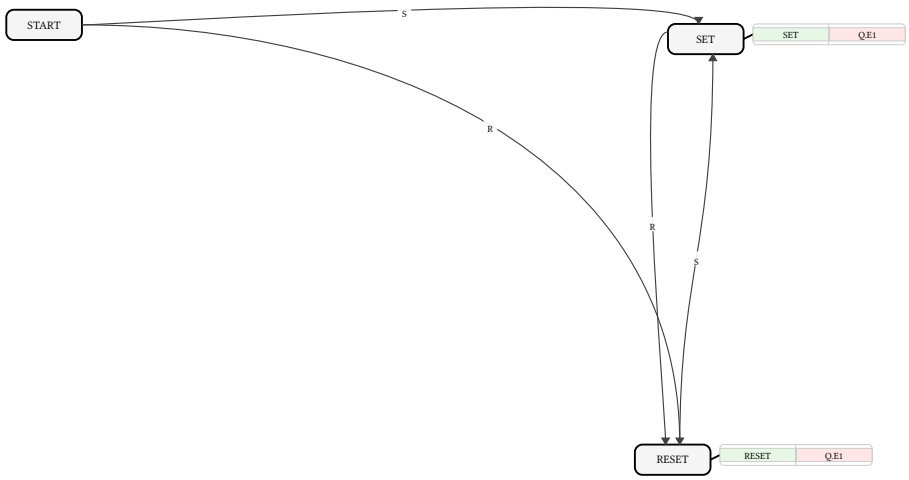
Conclusion

The AX_SR function block provides a reliable and easy-to-use solution for bistable memory functions in distributed automation systems. Through the use of adapters, it enables flexible integration into various system architectures and offers a clear, event-driven interface for Set-reset operations.

AX_SR_SYM



ECC: AX_SR_SYM



• – ○ • – ○ • – ○ •

Introduction

The function block **AX_SR_SYM** is an event-driven, bistable flip-flop with symmetrical turn-on behavior. It implements a set/reset memory, which is accessed via the events `S` (Set) and `R` (Reset). Unlike similar function blocks, the start state is not predefined; the block reacts to the first arriving event – regardless of whether it is `S` or `R`.

Interface Structure

Event Inputs

Name	Type	Description

S	Event	Sets output Q to TRUE.
R	Event	Sets output Q to FALSE.

Event Outputs

None (the output value is provided via an adapter).

Data Inputs

None.

Adapter

Name	Type	Description
Q	`adapter::types::unidirectional::AX`	Provides the current binary state. The adapter provides an event (`E1`) and a data value (`D1`). After each state change, `Q.E1` is triggered, and the new value is output to `Q.D1`.

Functionality

The function block is based on a simple state machine with three states:

- **START**: Initial state after startup. No output value is output.
- **SET**: The output is set to TRUE. Upon entering this state, `Q.E1` is triggered, and `Q.D1` is set to `TRUE`.
- **RESET**: The output is set to FALSE. Upon entering this state, `Q.E1` is triggered, and `Q.D1` is set to `FALSE`.

Transitions occur exclusively through the events `S` and `R`. From the START state, both `S` and `R` lead to SET (symmetrical behavior). From SET, only `R` leads to RESET, and from RESET, only `S` leads back to SET.

Technical Features

- **Symmetrical Start-up Behavior**: The function block starts in the START state without a defined output value. Only the first arriving event (S or R) determines the initial state. This differs from function blocks that initialize to FALSE or TRUE by default.
- **Adapter Interface**: The state is not directly transmitted as a data output, but via a unidirectional adapter (`AX`). Changes are signaled by an event (`Q.E1`), allowing the receiving function block to react asynchronously.

State Overview

State	Description
START	Waiting for first event; no output value defined.
SET	Output value TRUE; set by event `S`.
RESET	Output value FALSE; reset by event `R`.

Transitions:

- START ⟶ SET on event `S`
- START ⟶ RESET on event `R`
- SET ⟶ RESET on event `R`
- RESET ⟶ SET on event `S`

Application Scenarios

- **Storing a binary state** in control logic where the initial value is defined by an event.
- **Event-driven switches** in conjunction with adapter-based communication patterns.
- **Replacing an SR flip-flop** when symmetrical turn-on behavior is required.

Comparison with Similar Function Blocks

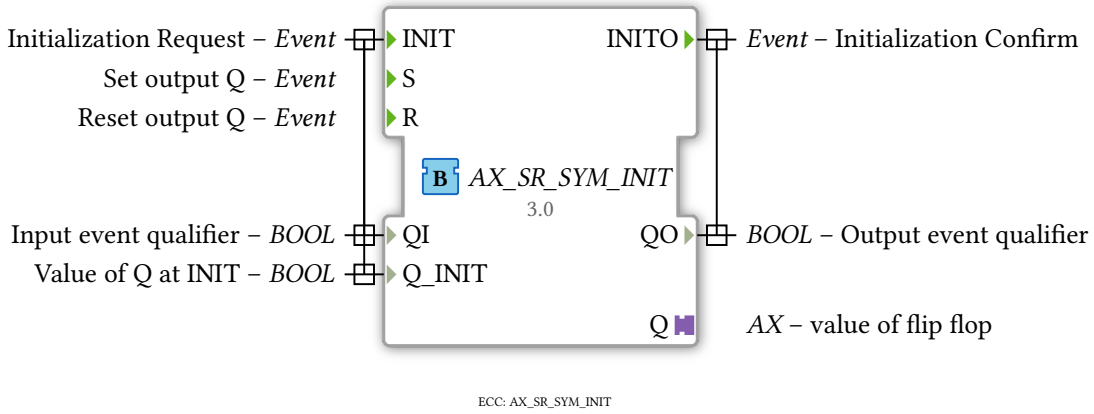
- **SR Flip-Flop** (e.g., `AX_SR`): Often has a defined initial value (e.g., FALSE) and behaves asymmetrically when both inputs are active simultaneously. In contrast, the `AX_SR_SYM` only sets its initial value upon the first event.

- **RS Flip-Flop**: Fundamentally similar logic, but with reversed priority (reset dominates). The `AX_SR_SYM` has no priority – the last received input determines the state.
- **Adapter-Based Function Blocks**: Advantage of event-driven state transfer via adapters; many standard function blocks use direct data outputs instead.

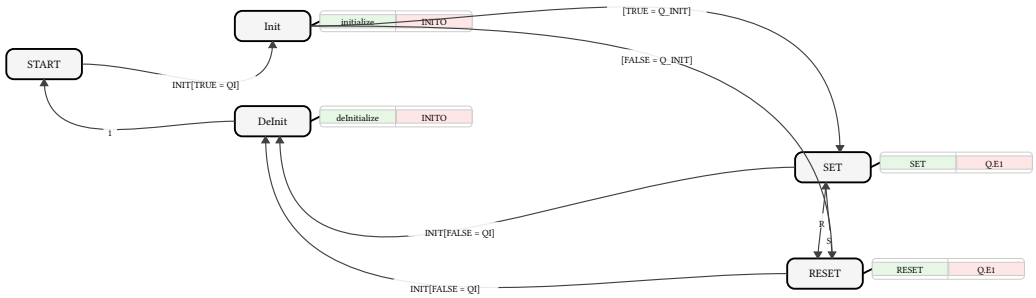
Conclusion

The `AX_SR_SYM` is ideally suited for applications requiring bistable memory behavior with an undefined initial value. The symmetrical response to `S` and `R` in the startup state, as well as the output via an adapter, makes it a flexible component in event-driven automation systems.

AX_SR_SYM_INIT



ECC: AX_SR_SYM_INIT



• – ○ • – ○ • – ○ •

Introduction

The function block **AX_SR_SYM_INIT** implements an event-driven, bistable flip-flop (set-reset) with symmetrical start-up behavior and special initialization logic. It extends the classic SR flip-flop with a qualified initialization and deinitialization sequence, where the output state is specified via the input `Q_INIT`, and the entire logic is only active if the qualifier `QI` has the value `TRUE`.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initialization request; linked to QI and Q_INIT
S	Event	Sets output Q (via adapter)
R	Event	Resets output Q

Event Outputs

Event	Type	Description
INITO	EInit	Initialization confirmation; linked to QO

Data Inputs

Variable	Type	Description
QI	BOOL	Input qualifier – controls whether actions are executed
Q_INIT	BOOL	Desired value of the output Q after initialization

Data Outputs

Variable	Type	Description
QO	BOOL	Output qualifier – reflects the value of QI at the time of the triggering event

Adapters

Adapter	Type	Description
Q	adapter::types::unidirectional::AX	Flip-flop value – set/reset via the interface

Functionality

The function block has four main states: START, Init, DeInit, SET, and RESET.

- **START** – Wait state after system startup.
- **Init** – Activated when INIT and QI = TRUE are both present. In this state, the algorithm initialize is executed, which sets QO := QI. The state then branches depending on Q_INIT:
- If Q_INIT = TRUE is present, the system switches to the SET state (the flip-flop is set).
- If Q_INIT = FALSE is present, the system switches to the RESET state (the flip-flop is reset).
- **DeInit** – Activated when INIT and QI = FALSE occur (from states SET or RESET). The algorithm deInitialize sets QO := FALSE. Afterwards, the state reverts to START.
- **SET** – Reached by the event S (from RESET or after initialization with Q_INIT=TRUE). The algorithm SET sets QO := QI and writes the value TRUE to the adapter output Q.D1 **only if QI = TRUE**. A subsequent S has no effect as long as the state SET is active.
- **RESET** – This is reached by the event R (from SET or after initialization with Q_INIT=FALSE). The algorithm RESET sets QO := QI and writes the value FALSE to Q.D1 **only if QI = TRUE**. A subsequent R has no effect.

The transitions between SET and RESET are triggered only by the events S and R, respectively. The INIT transitions can result from either stable state in DeInit if QI = FALSE is present. …

Technical Features

- **Qualifier-Dependent Actions**: The actual change to the flip-flop output (Q.D1) only occurs if QI = TRUE is present. Otherwise, only the output qualifiers (QO) are updated; the adapter remains unchanged.

- **Symmetrical Startup**: The device can be initialized with either a "set" or a "reset" state, depending on the value of `Q_INIT`.
- **Deinitialization**: If `QI = FALSE` is present during initialization, the device is set to the `DeInit` state, which clears the output qualifier and then returns to the startup state.
- **Guard Conditions**: The state transitions use conditions like `INIT[TRUE = QI]` or `[FALSE = Q_INIT]` to precisely control the logic.

State Overview

- **START** → Idle state after startup.
- **Init** → Initialization run (only with `QI=TRUE`).
- **DeInit** → Deinitialization run (only with `QI=FALSE`).
- **SET** → Stable state in which `Q.D1 = TRUE` is set (if previously set).
- **RESET** → Stable state in which `Q.D1 = FALSE` is reset (if previously reset).

Transitions:

- `START` → `Init` : if `INIT` & `QI=TRUE`
- `Init` → `SET` : if `Q_INIT=TRUE`
- `Init` → `RESET` : if `Q_INIT=FALSE`
- `SET` → `RESET` : if `R`
- `RESET` → `SET`: if `S`
- `SET` → `DeInit`: if `INIT` & `QI=FALSE`
- `RESET` → `DeInit`: if `INIT` & `QI=FALSE`
- `DeInit` → `START`: automatically after completion of the DeInit algorithm

Application Scenarios

- **Control of actuators with defined start behavior**: A motor or valve should perform a certain action after being switched on. Assume a specific state (e.g., closed = `Q=FALSE`), depending on an initialization specification.
- **Qualified State Changes**: In systems where an enable signal (`QI`) allows the actual value change, while the qualifier output (`QO`) signals validity.
- **Resettable Initialization Routine**: A component can be set or reset by an INIT call, but can later be put into a defined idle state by another INIT call with `QI=FALSE`.

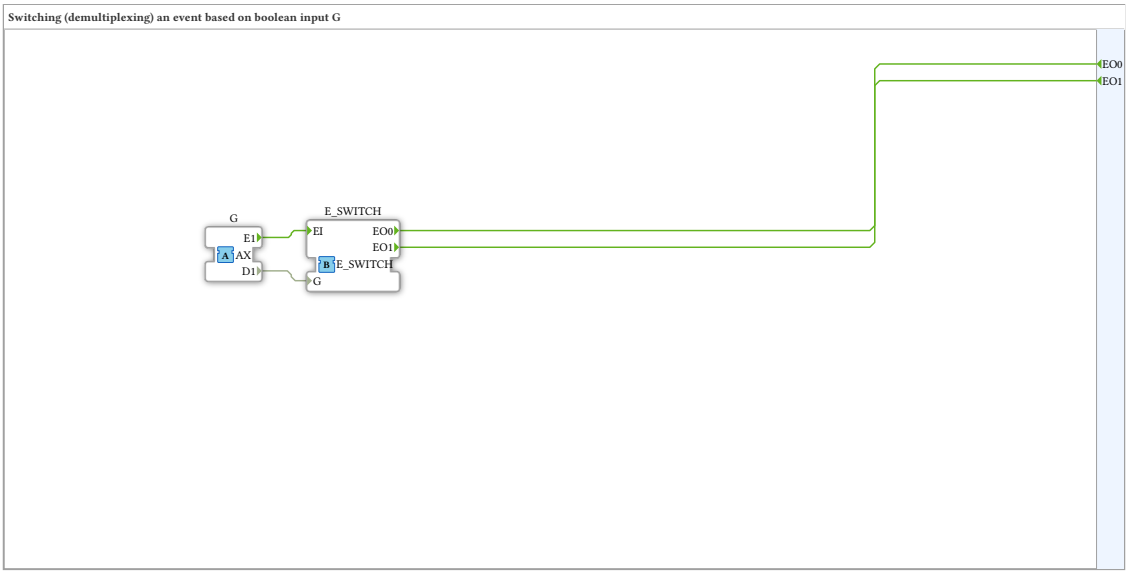
Comparison with Similar Function Blocks

- **Standard SR Flip-Flop (e.g., `SR` or `AX_SR`)**: Simple set/reset without initialization or qualifier. `AX_SR_SYM_INIT` additionally offers a qualified initialization and deinitialization process.
- **Function Blocks with INIT Interface (e.g., `E_SR` with `INIT`)**: These often lack pronounced deinitialization or a qualifier. The function block presented here differs in its symmetrical behavior – both setting and resetting can be specified in the INIT step.
- **Function Blocks with Adapter (`unidirectional::AX`)**)**: These are used to transmit the state externally via a standardized adapter interface. The `AX_SR_SYM_INIT` encapsulates this logic in a function block.

Conclusion

The `AX_SR_SYM_INIT` is a versatile function block for applications that require qualified startup and initialization behavior. By combining event control, qualifier logic, and an adapter interface, it is particularly suitable for modular automation solutions according to IEC 61499, where a defined system startup and clean deinitialization are required.

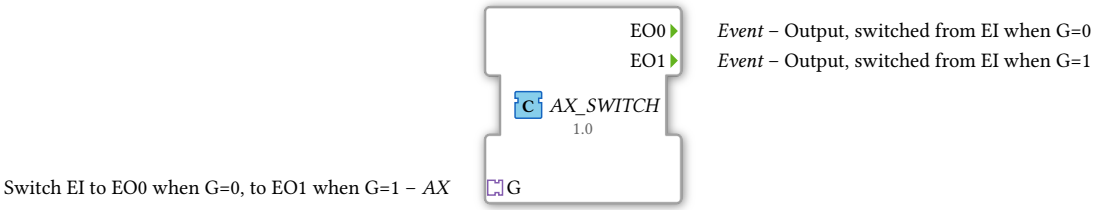
AX_SWITCH



• – ○ • – ○ • – ○ •

Introduction

The AX_SWITCH function block serves as a switching module (demultiplexer) for events based on a Boolean input value. It forwards incoming events to one of two possible outputs, depending on the state of the control signal.



Interface Structure

Event Inputs

- No direct event inputs - event input is handled via adapters

Event Outputs

- **EO0**: Output event that is activated when an event occurs at G=0
- **EO1**: Output event that is activated when an event occurs at G=1

Data Inputs

- No direct data inputs - data is provided via adapters

Data Outputs

- No data outputs available

Adapters

- **G**: Adapter of type `adapter::types::unidirectional::AX`
- Switches incoming events from EI to EO0 when G=0
- Switches incoming events from EI to EO1 when G=1

Functionality

The AX_SWITCH block receives events and control data via adapter G. Based on the Boolean value of the The control signal G forwards each incoming event either to output EO0 (when G=0) or to output

EO1 (when G=1). Internally, the block uses an E_SWITCH function block to implement the switching function.

Technical Features

- Implemented as a wrapper around the standard E_SWITCH function block
- Uses unidirectional adapters for event and data communication
- No direct interfaces; all communication occurs via adapters

State Overview

The block has no internal states in the traditional sense. It operates purely event-driven and immediately forwards each incoming event to the corresponding output based on the current value of G.

Application Scenarios

- Control of alternative process paths
- Switching between operating modes
- Distribution functions in event-based controllers
- Demultiplexing of event streams

⚖️ Comparison with Similar Function Blocks

Compared to the standard E_SWITCH block, AX_SWITCH offers an adapter-based interface, enabling better integration into adapter-based architectures. While E_SWITCH has direct inputs and outputs, AX_SWITCH communicates exclusively via adapters.

Comparison with E_SWITCH

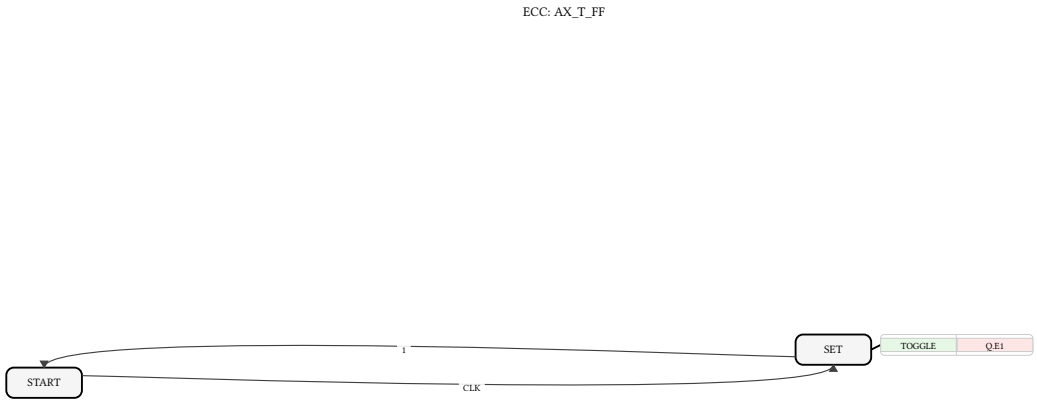
🛠️ Related exercises

- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_005_AX
- Uebung_006a3_sub_AX
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_009_AX
- Uebung_020a_AX
- Uebung_020b_AX
- Uebung_020c3_AX
- Uebung_020d_AX
- Uebung_020e2_AX
- Uebung_020f2_AX
- Uebung_020i_AX

Conclusion

AX_SWITCH is a specialized switching device for event-based systems thanks to its adapter-based architecture It enables flexible integration into more complex control systems. The clear separation of event and data flow via adapters ensures a clear system structure.

AX_T_FF

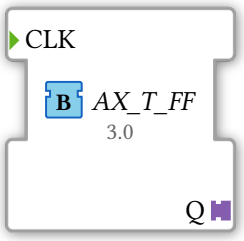


• – ○ • – ○ • – ○ •

Introduction

The AX_T_FF (Toggle Flip-Flop) is a fundamental memory component in 4diac that functions as a toggle flip-flop. With each clock event, the output value toggles between its two possible states. The component implements a simple switching behavior where the output value is toggled on each clock pulse.

Clock for triggering a output toggle – *Event*



AX – value of flip flop

Interface Structure

Event Inputs

- **CLK**: Clock input that triggers an output toggle

Event Outputs

- No direct event outputs available

Data Inputs

- No data inputs available

Data Outputs

- No direct data outputs available

Adapters

- **Q**: Unidirectional adapter of type AX that provides the current flip-flop value

Functionality

The AX_T_FF operates as a simple toggle flip-flop. Upon each incoming CLK event, the TOGGLE algorithm is executed, which inverts the current value of adapter Q.D1. The device starts in the START state and transitions to the SET state upon each CLK event, where the toggle is performed.

Technical Features

- Uses a unidirectional adapter for output
- Implemented as a Basic Function Block (BFB)
- Features a simple two-state machine
- The TOGGLE algorithm performs a logical negation of the output value

State Overview

The block has two states:

1. **START**: Initial state, waits for a CLK event
2. **SET**: Active state in which the TOGGLE algorithm is executed

State Transitions:

- START ⟶ SET: On a CLK event
- SET ⟶ START: Always (Condition "1" = true)

Application Scenarios

- Frequency division of clock signals
- Generation of square waves with half the input frequency
- Counter and divider circuits
- State control with alternating behavior

⚖️ Comparison with Similar Blocks

Compared to other flip-flop types such as RS or D flip-flops, the A toggle flip-flop is a specialized functionality for simple switching operations. It has a simpler structure than more general-purpose flip-flop types and requires no additional data inputs.

Comparison with E_T_FF

🛠️ Related exercises

- Uebung_004a2_AX
- Uebung_004a3_AX
- Uebung_004a4_AX
- Uebung_004a5_AX
- Uebung_004a6_AX
- Uebung_004a8_AX
- Uebung_004a9_AX
- Uebung_004a_AX
- Uebung_004c1_AX
- Uebung_004c2_AX
- Uebung_004c3_AX
- Uebung_004c4_AX
- Uebung_004c5_AX
- Uebung_004c6_AX
- Uebung_004c7_AX
- Uebung_005_AX
- Uebung_006a3_sub_AX
- Uebung_007_AX
- Uebung_007a1_AX
- Uebung_007a2_AX
- Uebung_010b2_AX
- Uebung_010b3_AX
- Uebung_010b6_AX
- Uebung_010b7_AX
- Uebung_010b8_AX

- Uebung_010b9_AX
- Uebung_010bA2_AX
- Uebung_010bA3_AX
- Uebung_010bA4_AX
- Uebung_010bA_AX
- Uebung_035a2_AX
- Exercise_094a_AX
- Exercise_095_AX
- Exercise_150_AX
- Exercise_151_AX

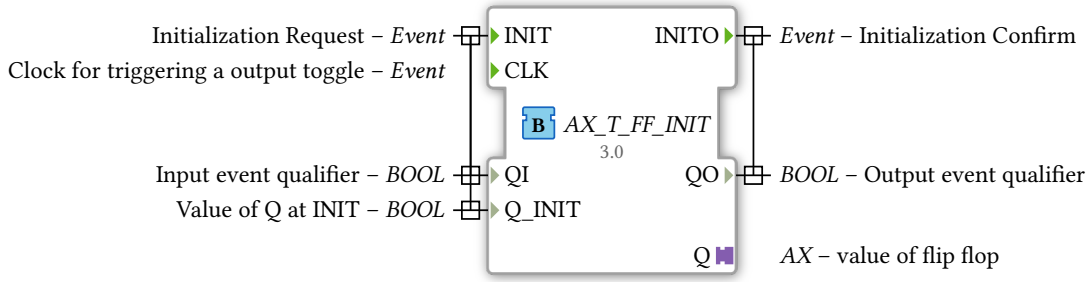
Conclusion

The AX_T_FF is a specialized and efficient component for applications requiring pure toggle behavior. Its simple structure and clear functionality make it a reliable component for frequency division and state switching in control systems.

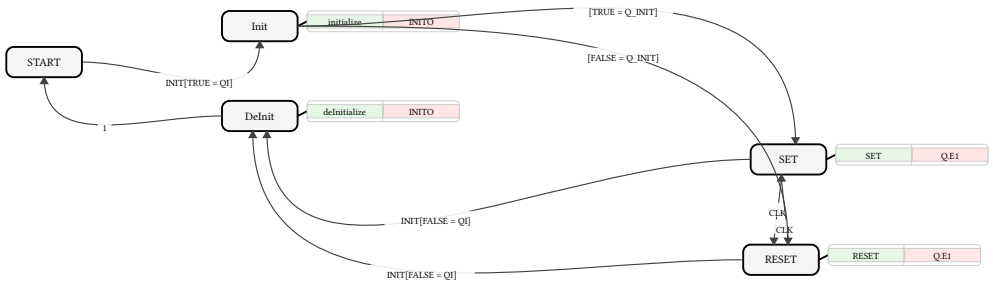
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AX_T_FF_INIT



ECC: AX_T_FF_INIT



• – ○ • – ○ • – ○ •

Introduction

The function block **AX_T_FF_INIT** implements a triggered toggle flip-flop with an initialization function. It can be set to a defined start state via an INIT event and subsequently toggles between the SET and RESET states with each CLK event. The current state is provided via an adapter output. This block is particularly suitable for applications where an output signal needs to be switched at each clock cycle, with the initial state defined by initialization.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization request; triggers the initialization of the flip-flop. The corresponding data inputs QI and Q_INIT are evaluated.
CLK	Event	Clock input; the internal state toggles between SET and RESET with each event.

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization confirmation; output after successful initialization or deinitialization.

Data Inputs

Name	Type	Comment
QI	BOOL	Input qualifier; controls whether actions are actually executed (TRUE = active).
Q_INIT	BOOL	Value assigned to the flip-flop during initialization (TRUE = SET, FALSE = RESET).

Data Outputs

Name	Type	Comment
QO	BOOL	Output qualifier; reflects the value of QI if an action has been performed, otherwise FALSE.

Adapters

Name	Type	Comment
Q	Adapter `adapter::types::unidirectional::AX`	Adapter output that passes the current flip-flop state (TRUE or FALSE) to connected function blocks.

Functionality

The function block has the following internal states: START, Init, DeInit, SET, and RESET.

1. **Initialization (INIT Event)**

- On an INIT event with **QI** = **TRUE**, the system transitions from the start state to the **Init** state.
- In the Init state, the algorithm *initialize* is executed: `QO := QI`.
- Subsequently, the output qualifier QO is set to TRUE, and the **INITO** event is output.
- Depending on the value of **Q_INIT**, the transition occurs:

If Q_INIT = TRUE ⟶ transition to the **SET** state
If Q_INIT = FALSE ⟶ transition to the **RESET** state

1. **Toggle Operation (CLK Event)**

- Starting from **SET** or **RESET**, the state changes with each **CLK** event:
- From SET ⟶ RESET (and vice versa)

- Upon exiting the states, the corresponding algorithms are executed:
- **SET**: `QO := QI;` If QI = TRUE, **Q.D1** = **TRUE** is assigned to the adapter output.
- **RESET**: `QO := QI;` If QI = TRUE, **Q.D1** = **FALSE** is assigned to the adapter output.
- In both cases, the event **Q.E1** is output (via the adapter).

1. **Deinitialization (INIT event with QI = FALSE)**

- If an INIT event with **QI** = **FALSE** occurs during operation (in SET or RESET), the function block switches to the **DeInit** state.
- The *deInitialize* algorithm sets `QO := FALSE`.
- **INITO** is output, and the function block returns to the start state.

The function block remains in the start state until an INIT event with QI = TRUE occurs. Toggle operation is not possible without initialization.

Technical Features

- **Adapter-based output signal**: The current flip-flop state is not provided as a simple data output, but via an **adapter** (`adapter::types::unidirectional::AX`). This enables flexible and typed communication with other function blocks that use the same adapter.
- **State-Dependent Execution**: The actions (setting/resetting the adapter output) are only executed if the qualifier **QI** = TRUE. If QI = FALSE, the adapter output is not changed, but the state still changes.
- **Initialization with Start Value**: The initial state after initialization can be specifically defined via the input **Q_INIT** (TRUE = set, FALSE = reset).

State Overview

State	Meaning
START	Waiting for first INIT event.
Init	Initialization is in progress; QO is set to QI, then transition to SET or RESET depending on Q_INIT.
DeInit	Deinitialization is in progress; QO is set to FALSE, return to START.
SET	Flip-flop is in the set state; Adapter output Q.D1 = TRUE (when QI=TRUE).
RESET	Flip-flop is in the reset state; adapter output Q.D1 = FALSE (when QI=TRUE).

The state transitions are controlled by the INIT event (with a corresponding condition on QI) and CLK.

Application Scenarios

- **Controlling outputs with toggle function**: A digital output should switch between on and off with each clock cycle, e.g., for blink signals or pulse trains.
- **Initializable state machines**: In automation systems where a process must assume a defined start state (set or reset) after a reset.
- **Adapter-based communication**: Devices using the same adapter can directly read the flip-flop state without relying on separate data outputs.

Comparison with similar devices

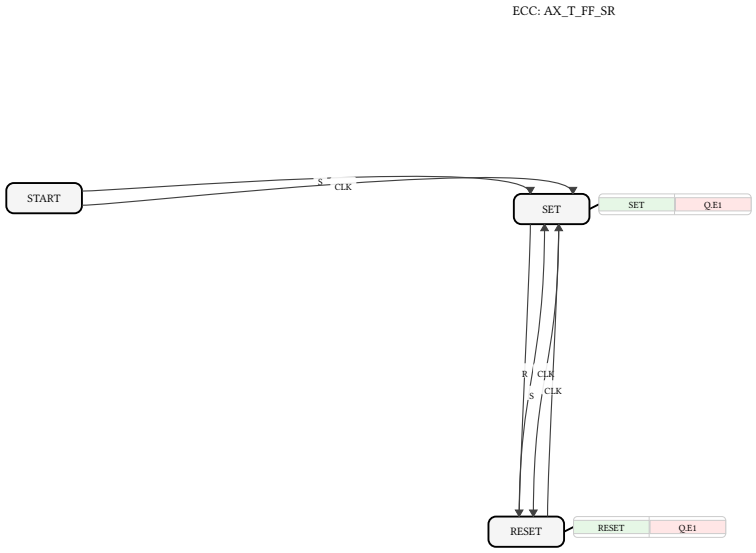
Criterion	AX_T_FF_INIT	Simple Toggle Flip-Flop (without INIT)	SR Flip-Flop
Initialization	Yes, via INIT with Q_INIT	No, state undefined after startup	No (set/reset via separate inputs)
State Change	Clocked by CLK	Clocked by CLK	Asynchronous via S and R
Output	Via adapter (typed)	Often as a BOOL data output	BOOL data output
Qualifier QI	Controls whether an action is executed	Not present	Not present

Compared to a simple toggle flip-flop, the **AX_T_FF_INIT** offers the possibility of defined initialization and uses an adapter for the output, which simplifies configuration in complex systems.

Conclusion

The function block **AX_T_FF_INIT** is a versatile building block for clocked toggle operations with initialization logic. Its combination of INIT and CLK events, the QI qualifier, and the adapter output allows for precise control of the state behavior. It is particularly suitable for industrial automation applications where a defined start value and clocked switching are required.

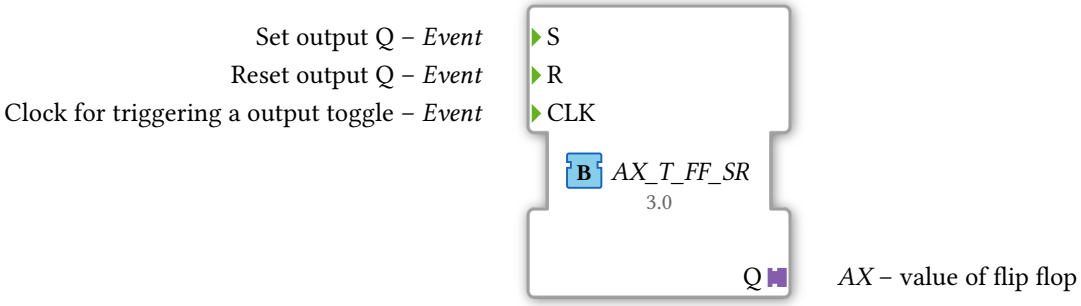
AX_T_FF_SR



• – ○ • – ○ • – ○ •

Introduction

The AX_T_FF_SR is an event-driven bistable function block with toggle functionality. It is a flip-flop element that can operate as both a set-reset flip-flop and a toggle flip-flop. The component combines the properties of an SR flip-flop with additional toggle functionality via a clock input.



Interface Structure

Event Inputs

- **S**: Sets output Q to TRUE
- **R**: Sets output Q to FALSE (Reset)
- **CLK**: Clock signal for toggling the output

Event Outputs

- No direct event outputs available

Data Inputs

- No data inputs available

Data Outputs

- No direct data outputs available

Adapters

- **Q**: Unidirectional AX-type adapter that provides the flip-flop value

Operation

The AX_T_FF_SR has three operating states:

- **START**: Initial state
- **SET**: Output Q is TRUE
- **RESET**: Output Q is FALSE

The state transitions are controlled by the event inputs:

- The S event transitions from any state to the SET state.

The R event transitions from any state to the RESET state.
The CLK event toggles the current state (SET → RESET or RESET → SET).
With each state change, the corresponding algorithm is executed, setting the adapter value Q.D1 accordingly.

Technical Features

- Combines SR flip-flop and T flip-flop functionality
- Uses the adapter interface for data output
- Unidirectional communication via the Q adapter
- Initial state is START, from which a transition to SET can occur directly or via CLK

State Overview

START (Initialzustand) │ ├── S ──→ SET (Q.D1 = TRUE) │ └── CLK ──→ SET (Q.D1 = TRUE)
SET (Q.D1 = TRUE) │ ├── R ──→ RESET (Q.D1 = FALSE) │ └── CLK ──→ RESET (Q.D1 = FALSE)
RESET (Q.D1 = FALSE) │ ├── S ──→ SET (Q.D1 = TRUE) │ └── CLK ──→ SET (Q.D1 = TRUE)

Application Scenarios

- State storage in control applications
- Clock and frequency division
- Event counting
- State machines with memory function
- Switching networks with feedback

⚖️ Comparison with similar components

Compared to a simple E_SR flip-flop, the AX_T_FF_SR offers additional toggle functionality through its CLK input. While a pure SR flip-flop only has set and reset inputs, this component also allows clock-controlled switching of the output state.
Comparison with E_T_FF_SR

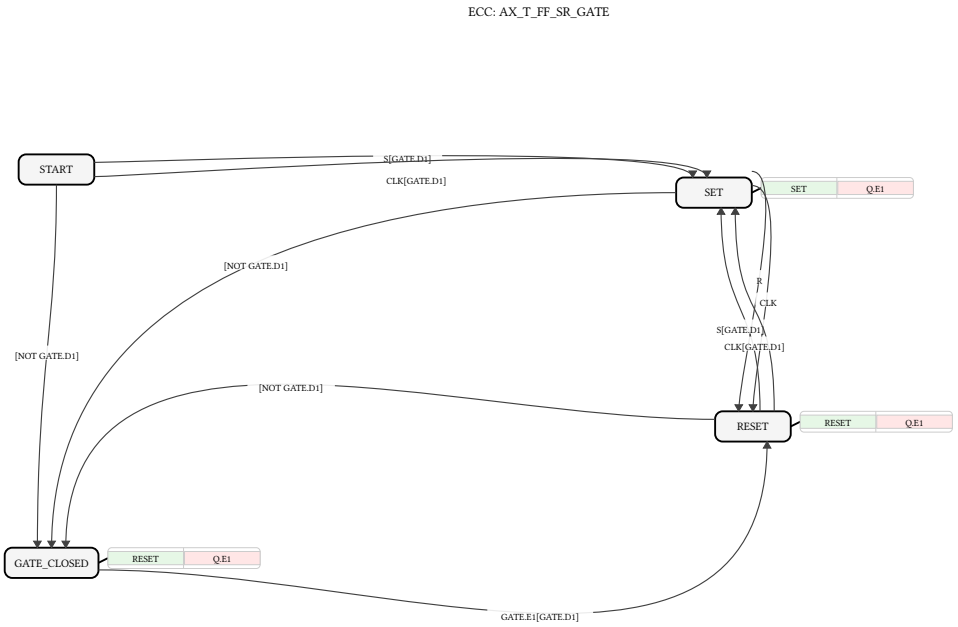
🛠️ Related exercises

- Uebung_004a7_AX
- Uebung_006a2_AX
- Uebung_006a3_AX
- Uebung_006a4_AX
- Uebung_006a_AX

Conclusion

The AX_T_FF_SR is a versatile bistable memory device that combines the advantages of SR and T flip-flops. Thanks to its adapter-based interface, it enables flexible integration into larger control systems and is particularly suitable for applications that require both direct state setting and clock-controlled switching.

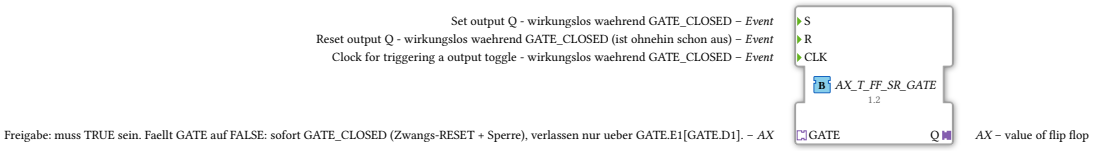
AX_T_FF_SR_GATE



• – ○ • – ○ • – ○ •

Introduction

`AX_T_FF_SR_GATE` corresponds to `AX_T_FF_SR`, but additionally has an enable socket `GATE` (`AX`) and a state of its own, `GATE_CLOSED`, without chaining (no `CHAIN_IN`/`CHAIN_OUT`). It is suited for cases where only the gate lock is needed but no chain delay; for the chained variant see `AX_T_FF_SR_SEQ_GATE`.



Interface Structure

Event Inputs

- **S**: Sets the output `Q` - no effect during `GATE_CLOSED`
- **R**: Resets the output `Q` - no effect during `GATE_CLOSED` (it is off there anyway)
- **CLK**: Clock for toggling the output - no effect during `GATE_CLOSED`

Event Outputs

No direct event outputs - the event is output via `Q.E1`.

Data Inputs / Data Outputs

None.

Adapters

- **Q** (plug, `adapter::types::unidirectional::AX`): Value of the flip-flop

- **GATE** (socket, `adapter::types::unidirectional::AX`): Enable, must be `TRUE`. If `GATE` falls to `FALSE`, the block immediately enters `GATE_CLOSED` (forced reset and lock); this state is left only via `GATE.E1[GATE.D1]`

How it works

States and actions:

- **START**: Initial state
- **SET**: Algorithm `SET` (`Q.D1 := TRUE`), output `Q.E1`
- **RESET**: Algorithm `RESET` (`Q.D1 := FALSE`), output `Q.E1`
- **GATE_CLOSED**: locked (`GATE.D1 = FALSE`), `Q` forced to `FALSE` (algorithm `RESET`, output `Q.E1`); `S`/`R`/`CLK` have no effect here

Transitions (in declaration order):

From	To	Condition
START	GATE_CLOSED	`[NOT GATE.D1]`
START	SET	`S[GATE.D1]`
START	SET	`CLK[GATE.D1]`
SET	GATE_CLOSED	`[NOT GATE.D1]`
SET	RESET	`R`
SET	RESET	`CLK`
RESET	GATE_CLOSED	`[NOT GATE.D1]`
RESET	SET	`S[GATE.D1]`
RESET	SET	`CLK[GATE.D1]`
GATE_CLOSED	RESET	`GATE.E1[GATE.D1]`

Technical Details

- **Lock first**: `[NOT GATE.D1]` comes first in the declaration order of every state. Since `S`/`CLK` without a guard are always `TRUE`, according to IEC 61499-1 Table 1 rule c) the gate lock would otherwise never take effect (bugfix v1.2). If the gate is open, the guard is `FALSE` and is skipped
- **Boot window**: The event-less transition `START` → `GATE_CLOSED` is evaluated immediately on entering `START` and covers the state before a `GATE.E1` has ever fired (v1.1)
- From `GATE_CLOSED` the block returns only to `RESET`, so `Q` stays `FALSE` until `S` or `CLK` takes effect
- Copy of `AX_T_FF_SR` with `GATE` socket and state `GATE_CLOSED`, same pattern as `AX_T_FF_SR_SEQ_GATE`

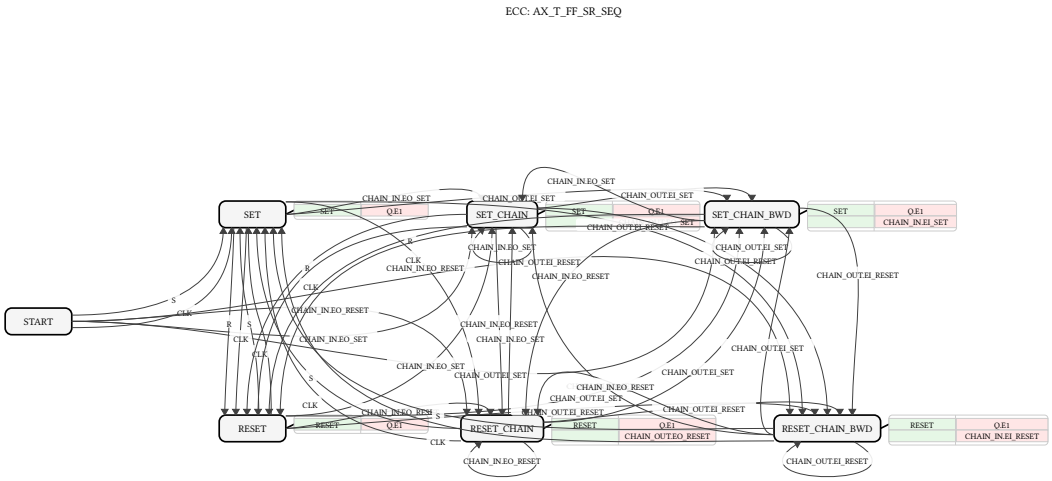
Application Scenarios

- Toggle/set-reset output that must be forcibly locked by an enable (e.g. safety or operating enable)

Summary

`AX_T_FF_SR_GATE` is a toggle/SR flip-flop with enable: if `GATE` is `FALSE`, `Q` is reset and all inputs remain without effect until the enable returns.

AX_T_FF_SR_SEQ



• — ○ • — ○ • — ○ •

Introduction

`AX_T_FF_SR_SEQ` is an event-driven bistable flip-flop with toggle function (a copy of `AX_T_FF_SR`) that is additionally **truly bidirectionally chainable** via the adapter `ASR2` (`CHAIN_IN`/`CHAIN_OUT`). This allows several belts to be switched on and off in a staggered manner (e.g. together with `ASR2_DELAY`). The local events `S`/`R`/`CLK` behave exactly like in `AX_T_FF_SR` and **never** trigger `CHAIN_IN`/`CHAIN_OUT`.



Interface Structure

Event Inputs

- **S**: Sets output `Q` (local, never affects `CHAIN_IN`/`CHAIN_OUT`)
- **R**: Resets output `Q` (local, never affects `CHAIN_IN`/`CHAIN_OUT`)
- **CLK**: Clock for toggling the output (local, never affects `CHAIN_IN`/`CHAIN_OUT`)

Event Outputs

No direct event outputs - events run via the adapter plugs

Data Inputs

No data inputs

Data Outputs

No data outputs - the state is output via the adapter `Q`

Adapters

- **Q** (plug): Unidirectional adapter of type `adapter::types::unidirectional::AX` - value of the flip-flop
- **CHAIN_IN** (socket): Bidirectional adapter of type `adapter::types::bidirectional::ASR2` - forward: receive set/reset from the previous chain element (`EO_SET`/`EO_RESET`); backward: pass set/reset on to the previous chain element (`EI_SET`/`EI_RESET`)
- **CHAIN_OUT** (plug): Bidirectional adapter of type `adapter::types::bidirectional::ASR2` - forward: pass set/reset on to the next chain element (`EO_SET`/`EO_RESET`); backward: receive set/reset from the next chain element (`EI_SET`/`EI_RESET`)

How it works

The block has seven states (`START`, `SET`, `RESET`, `SET_CHAIN`, `RESET_CHAIN`, `SET_CHAIN_BWD`, `RESET_CHAIN_BWD`). On entering a set state the output `Q` is set to `TRUE` by the algorithm `SET`, in a reset state to `FALSE` by `RESET`, and output via `Q.E1`.

There are **four strictly separate, non-branching chain paths**:

1. `CHAIN_IN.EO_SET` ⟶ local `SET` + `CHAIN_OUT.EO_SET` (forward set)
2. `CHAIN_IN.EO_RESET` ⟶ local `RESET` + `CHAIN_OUT.EO_RESET` (forward reset)
3. `CHAIN_OUT.EI_SET` ⟶ local `SET` + `CHAIN_IN.EI_SET` (backward set)
4. `CHAIN_OUT.EI_RESET` ⟶ local `RESET` + `CHAIN_IN.EI_RESET` (backward reset)

A set arriving via path 1 never additionally triggers path 3 or 4 and vice versa - each state fires exactly one chain action. Locally:

- `S` or `CLK` in `START` or `RESET`/`RESET_CHAIN`/`RESET_CHAIN_BWD` ⟶ `SET`
- `R` or `CLK` in `SET`/`SET_CHAIN`/`SET_CHAIN_BWD` ⟶ `RESET`
- The four chain events are possible from every state and lead to the respective chain state.

Technical Details

- Local `S`/`R`/`CLK` only changes `Q`; the chain is not triggered.
- With the backward paths (version 1.1) the same physical chain elements can be switched in a staggered manner both forward (e.g. set, downstream first) and backward (e.g. reset, upstream first), without a forward event also propagating backward or vice versa.
- An `R` in state `START` has no effect, since no transition with `R` exists from there.

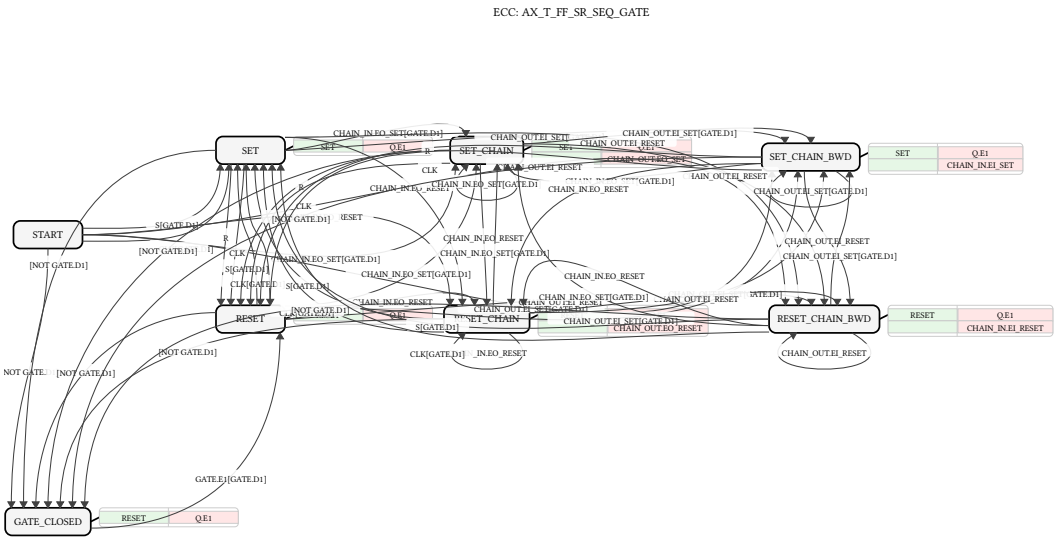
Application Scenarios

- Staggered switching on/off of several conveyor belts (e.g. hedgehog belt, elevator, cross belt) with delay via `ASR2_DELAY`

Summary

`AX_T_FF_SR_SEQ` extends the set/reset/toggle flip-flop by four separate chain paths (forward/backward, set/reset) for cascaded, staggered switching operations.

AX_T_FF_SR_SEQ_GATE



• — ○ • — ○ • — ○ •

Introduction

The `AX_T_FF_SR_SEQ_GATE` is a toggle/set/reset flip-flop with bidirectional chain forwarding, like `AX_T_FF_SR_SEQ`, and additionally has an enable socket `GATE` (type `AX`) and a dedicated state `GATE_CLOSED`. While `GATE` is TRUE the block works normally. If `GATE` falls to FALSE, a RESET is forced immediately and the block is locked; the only way out of `GATE_CLOSED` is `GATE.E1[GATE.D1]`, i.e. reopening the gate.



Interface Structure

Event Inputs

- **S**: sets the output `Q` (local, never acts on `CHAIN_IN`/`CHAIN_OUT`); has no effect during `GATE_CLOSED`
- **R**: resets the output `Q` (local); has no effect during `GATE_CLOSED`
- **CLK**: clock for toggling the output (local); has no effect during `GATE_CLOSED`

Event Outputs

The function block has no direct event outputs.

Data Inputs

The function block has no direct data inputs.

Data Outputs

The function block has no direct data outputs.

Adapters

- **Q** (plug, `adapter::types::unidirectional::AX`): value of the flip-flop
- **CHAIN_OUT** (plug, `adapter::types::bidirectional::ASR2`): forward Set/Reset to the next chain element (`EO_SET`/`EO_RESET`), backward Set/Reset received from the next chain element (`EI_SET`/`EI_RESET`); not during `GATE_CLOSED`
- **CHAIN_IN** (socket, `adapter::types::bidirectional::ASR2`): forward Set/Reset received from the previous chain element (`EO_SET`/`EO_RESET`), backward Set/Reset passed on to the previous chain element (`EI_SET`/`EI_RESET`); not during `GATE_CLOSED`
- **GATE** (socket, `adapter::types::unidirectional::AX`): enable, must be TRUE. If `GATE` falls to FALSE, `GATE_CLOSED` follows immediately (forced RESET and lock); the state is left only via `GATE.E1[GATE.D1]`

Functionality

The ECC has the states `START`, `SET`, `RESET`, `SET_CHAIN`, `RESET_CHAIN`, `SET_CHAIN_BWD`, `RESET_CHAIN_BWD` and `GATE_CLOSED`. There are four strictly separate chain paths (forward set, forward reset, backward set, backward reset) without event duplication:

- **Local**: `S[GATE.D1]` or `CLK[GATE.D1]` leads to `SET` (algorithm `SET`: `Q.D1 := TRUE`, output `Q.E1`); `R` or `CLK` from a set state leads to `RESET` (`Q.D1 := FALSE`). These local events never act on the chain.
- **Forward**: `CHAIN_IN.EO_SET[GATE.D1]` leads to `SET_CHAIN` (sets `Q` and outputs `CHAIN_OUT.EO_SET`), `CHAIN_IN.EO_RESET` leads to `RESET_CHAIN` (resets `Q` and outputs `CHAIN_OUT.EO_RESET`).
- **Backward**: `CHAIN_OUT.EI_SET[GATE.D1]` leads to `SET_CHAIN_BWD` (sets `Q` and outputs `CHAIN_IN.EI_SET`), `CHAIN_OUT.EI_RESET` leads to `RESET_CHAIN_BWD` (resets `Q` and outputs `CHAIN_IN.EI_RESET`).
- **Gate**: every state except `GATE_CLOSED` has `[NOT GATE.D1]` to `GATE_CLOSED` as its first transition. There the algorithm `RESET` is executed with output `Q.E1`; `S`, `R`, `CLK`, `CHAIN_IN` and `CHAIN_OUT` have no effect because no transition is defined. The only way back to `RESET` is `GATE.E1[GATE.D1]`.

Technical Details

- The gate check `[NOT GATE.D1]` is eventless and comes first in the transition order of every state so that it is evaluated before the unguarded event transitions (e.g. `R`, `CLK`, `CHAIN_IN.EO_RESET`)
- The lock is level-based rather than a mere edge pulse on `R`: while the gate is closed, the block cannot be switched on again
- Set requests (`S`, `CLK` for setting, `CHAIN_IN.EO_SET`, `CHAIN_OUT.EI_SET`) are accepted only when `GATE.D1 = TRUE`; reset paths may always pass
- Nothing is forwarded to `CHAIN_IN`/`CHAIN_OUT` during `GATE_CLOSED`

State Overview

State	Meaning
START	initial state; with `GATE.D1 = FALSE` it goes to `GATE_CLOSED` immediately
SET	`Q` = TRUE through local `S/CLK`
RESET	`Q` = FALSE through local `R/CLK`
SET_CHAIN	`Q` = TRUE, set came forward and is passed on to `CHAIN_OUT.EO_SET`
RESET_CHAIN	`Q` = FALSE, reset came forward and is passed on to `CHAIN_OUT.EO_RESET`
SET_CHAIN_BWD	`Q` = TRUE, set came backward and is passed on to `CHAIN_IN.EI_SET`
RESET_CHAIN_BWD	`Q` = FALSE, reset came backward and is passed on to `CHAIN_IN.EI_RESET`
GATE_CLOSED	locked, `Q` = FALSE; exit only via `GATE.E1[GATE.D1]` to `RESET`

Application Scenarios

- Switchable outputs with a higher-level, level-based safety shutdown (e.g. PTO monitoring) that may only be switched on again after the enable has been granted again
- Sequenced chains of several flip-flops in which one element is to be locked depending on an enable

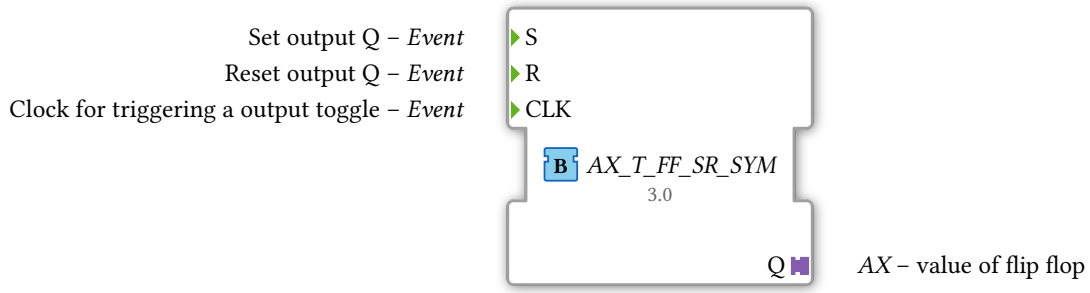
⚖️ Comparison with Similar Blocks

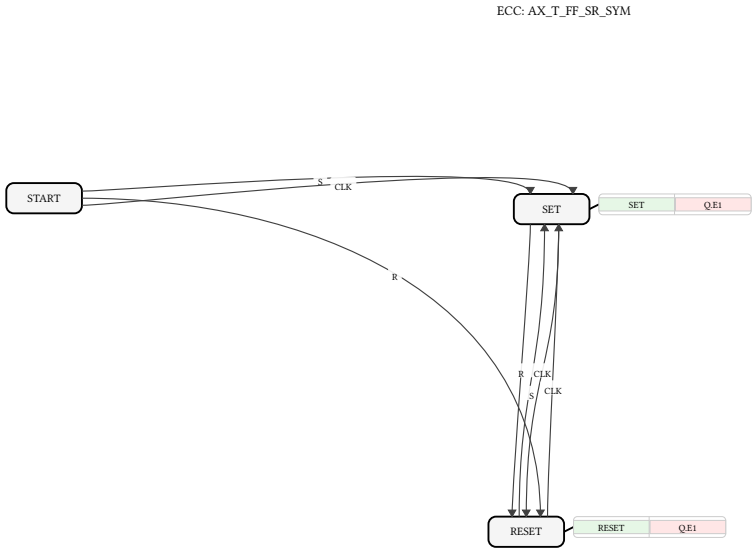
Compared with `AX_T_FF_SR_SEQ` (same four chain paths without an enable), `AX_T_FF_SR_SEQ_GATE` additionally offers the socket `GATE` and the state `GATE_CLOSED`. The lock is truly level-based here and thus solves a problem that a pure edge pulse on `R` does not: immediate re-enabling despite a shutdown that is still present.

Conclusion

The `AX_T_FF_SR_SEQ_GATE` extends the sequenceable AX flip-flop with a level-based enable: if `GATE` falls, the block is reset and locked immediately, and only a renewed `GATE.E1[GATE.D1]` releases it again.

AX_T_FF_SR_SYM





• – ○ • – ○ • – ○ •

Introduction

The AX_T_FF_SR_SYM function block is an event-driven bistable toggle switch with an additional toggle function and symmetrical start behavior. It combines the characteristics of an SR flip-flop (set and reset) with those of a T flip-flop (toggling on every clock signal). The behavior after power-on is symmetrical, meaning the block is initially in a neutral start state and only transitions to a defined state upon an event.

Interface Structure

Event Inputs

- **S** – Set: Sets output Q to TRUE.
- **R** – Reset: Sets output Q to FALSE.
- **CLK** – Clock: Toggles the output Q (TRUE ⟶ FALSE or FALSE ⟶ TRUE).

Event Outputs

No direct event outputs. The state is output via adapter Q. The adapter triggers event Q.E1 upon a state change.

Data Inputs

None.

Data Outputs

No direct data outputs. The current value is provided via adapter Q (Q.D1).

Adapters

- **Q**: Adapter of type `adapter::types::unidirectional::AX` (unidirectional). Provides the current state of the flip-flop as a Boolean value (TRUE/FALSE). The adapter triggers event Q.E1 after each state change.

Functionality

The function block (FB) has three event inputs and a state machine with three states: `START`, `SET`, and `RESET`.

- **START** is the initial state after power-on. In this state, no algorithm is active, and the output value is undefined.
- Upon receiving `S` or `CLK`, the FB transitions to state `SET`.
- Upon receiving `R`, it transitions to state `RESET`.
- In the **SET** state, the algorithm `SET` is executed: `Q.D1 := TRUE`. Then, the event `Q.E1` is triggered.
- At `R` or `CLK`, the function block transitions to the state `RESET`.
- In the **RESET** state, the algorithm `RESET` is executed: `Q.D1 := FALSE`. Then, `Q.E1` is triggered.
- At `S` or `CLK`, the function block (FB) switches back to state `SET`.

The toggle function is implemented by the `CLK` transitions: regardless of the current state, `CLK` leads to the other state.

Technical Features

- **Symmetrical Start Behavior**: After power-on, the FB is in the neutral `START` state. Only the first event determines the initial state. This allows for a defined system initialization.
- **Adapter-Based Output**: Instead of a direct data output, the state is provided via an adapter (`AX`). This enables flexible coupling with other function blocks and easy event propagation.
- **Combination of SR and T flip-flops**: This component combines set, reset, and toggle functions in a single compact unit.

State Overview

State	Description	Output Q.D1	Event by Algorithm
START	Initial state, no defined value	undefined	-
SET	Output is set to TRUE	TRUE	Q.E1
RESET	Output is set to FALSE	FALSE	Q.E1

Transitions (Event ⟶ Target State):

- START ⟶ SET: `S`, `CLK`
- START ⟶ RESET: `R`
- SET ⟶ RESET: `R`, `CLK`
- RESET ⟶ SET: `S`, `CLK`

Application Scenarios

- **Control of Binary Outputs**: An output can be set, reset, or toggled (e.g., manual operation with Set/Reset and automatic toggling via a clock signal).
- **Toggle Switching**: A process that changes its state with each clock signal but can also be explicitly set or reset.
- **Defined Start Sequences**: The symmetrical start behavior ensures that the component does not output any value after being powered on until a command arrives.

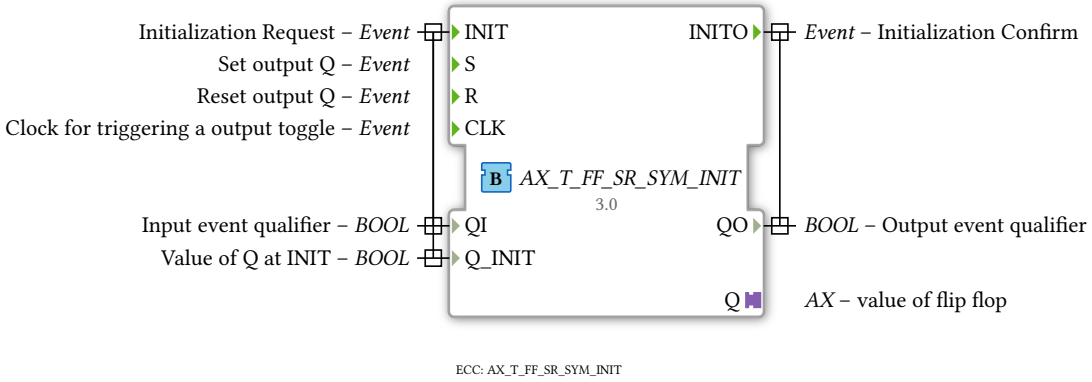
Comparison with Similar Components

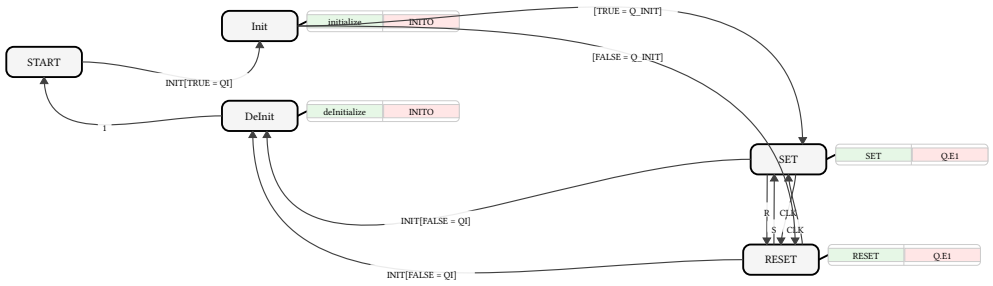
- **E_SR (Pure SR Flip-Flop)**: Has only Set and Reset inputs, no toggle function. After startup, the output is typically FALSE.
- **E_T (Toggle Flip-Flop)**: Only a toggle input, no Set/Reset.
- **AX_T_FF_SR_SYM**: Combines all three functions and additionally offers a neutral start state, allowing for flexible initialization.

Conclusion

The AX_T_FF_SR_SYM is a versatile function block for binary state logic. The combination of Set, Reset, and Toggle, as well as the symmetrical start behavior, makes it particularly suitable for applications requiring both manual control and automatic switching. The adapter interface facilitates integration into modular systems such as IEC 61499.

AX_T_FF_SR_SYM_INIT





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_T_FF_SR_SYM_INIT** implements an event-driven bistable flip-flop with **Set**, **Reset**, and **Toggle** functionality. It features **symmetrical start-up behavior**, where the output state after the INIT event can be defined via the parameter `Q_INIT`. The block is designed according to IEC 61499-1 Annex A and is suitable for applications requiring a resettable, set, or toggleable binary state with an initializable preset.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Initialization Request
S	Event	Set output Q (dependent on QI)
R	Event	Reset output Q (dependent on QI)
CLK	Event	Clock to toggle Q

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization Confirmation

Data Inputs

Variable	Type	Comment
QI	BOOL	Event Qualifier (Enable Actions)
Q_INIT	BOOL	Value of Q on the INIT event

Data Outputs

Variable	Type	Comment
QO	BOOL	Output Event Qualifier

Adapters

Adapter	Type	Comment
Q	adapter::types::unidirectional::AX	Value of the flip-flop (D1)

Functionality

The component has a finite state machine with five states: **START**, **INIT**, **DEINIT**, **SET**, and **RESET**.

- **START** is the initial state after power-on.
- An **INIT** event initiates the initialization:
- When `QI = TRUE` occurs, the system switches to the **Init** state. The algorithms `initialize` set `QO := QI`.
- Subsequently, based on `Q_INIT`, a decision is made whether to transition to the **SET** state (`Q_INIT = TRUE`) or the **RESET** state (`Q_INIT = FALSE`).
- During operation, **S**, **R**, or **CLK** can be processed:
- With **S** (Set), the adapter output `Q.D1` is set to `TRUE` in the algorithm `QI = TRUE`. `QO` receives the value of `QI`.
- With **R** (Reset), `Q.D1` is set to `FALSE`, also only if `QI` is active.
- With **CLK** (Toggle), the state is toggled: If the FB is in the **SET** state, a CLK event leads to the **RESET** state; if it is in **RESET**, it leads to **SET**. Does the toggle function operate independently of `QI`? *Note:* The transitions `SET–CLK→RESET` and `RESET–CLK→SET` are defined without further conditions; however, in the algorithms, `Q.D1` is only changed at `QI = TRUE`. If `QI = FALSE`, the adapter value remains unchanged, but the state transition still occurs. The internal state (SET/RESET) is always toggled, but the actual output to `Q.D1` only occurs if `QI` is true.
- A subsequent **INIT** event with `QI = FALSE` leads to the **DeInit** state, in which `QO := FALSE` is set. Afterward, the function block returns to the **START** state.

The output `QO` is set to the current value of `QI` with each executed algorithm. It serves as an event qualifier for the INITO output.

Technical Features

- **Symmetrical Start-up Behavior**: The initial value of the output is set via `Q_INIT`, allowing for both set and reset start states.
- **Event Qualifier QI**: The actual change to the adapter `Q.D1` (i.e., the flip-flop output) is only executed if `QI = TRUE`. With `QI = FALSE`, the events S, R, and CLK are processed (state change), but the adapter value remains unchanged.
- **Toggle Functionality**: The CLK event allows the output to be switched between 0 and 1, enabling simple square wave generation or state transitions in many applications.

- **INIT Handling**: The INIT event can be used for both initialization (`QI = TRUE`) and deinitialization (`QI = FALSE`).

State Overview

State	Action	Output/Event	Description
START	–	–	Waits for first INIT event.
Init	`initialize`	INITO	Sets `QO := QI`; Then switch to SET or RESET depending on `Q_INIT`.
DeInit	`deInitialize`	INITO	Sets `QO := FALSE`; then returns to START.
SET	`SET`	Q.E1	Sets `Q.D1 := TRUE` (if QI = TRUE) and `QO := QI`.
RESET	`RESET`	Q.E1	Sets `Q.D1 := FALSE` (if QI = TRUE) and `QO := QI`.
DeInit	`RESET`	Q.E1	Sets `Q.D1 := FALSE` (if QI = TRUE) and `QO := QI`.
DeInit	`Q_INIT = FALSE`	INITO	Sets `QO := FALSE` (if QI = TRUE) and `QO := QI`.
Set	`SET`	Q.E1	Sets `Q.D1 := FALSE` (if QI = TRUE) and `QO := QI`.
Set	`Q_INIT = FALSE`	Q.E1	**Transitions** (excerpt):

- START ⟶ Init: `INIT[QI = TRUE]`
- Init ⟶ SET: `Q_INIT = TRUE`
- Init ⟶ RESET: `Q_INIT = FALSE`
- SET ⟶ RESET: `R` or `CLK`
- RESET ⟶ SET: `S` or `CLK`
- SET ⟶ DeInit: `INIT[QI = FALSE]`
- RESET ⟶ DeInit: `INIT[QI = FALSE]`
- DeInit ⟶ START: (always, condition = 1)

Application Scenarios

- **Machine Control**: Defined start state after power-up (e.g., Valve Closed = `Q_INIT = FALSE`).
- **State Change with Toggle**: Switches a light or drive with each CLK pulse.
- **Safe Initialization**: The action can be made conditional via `QI` (e.g., enabling by a higher-level controller).
- **Fault Reset**: A RESET event resets the output, while setting is possible via S.

Comparison with Similar Function Blocks

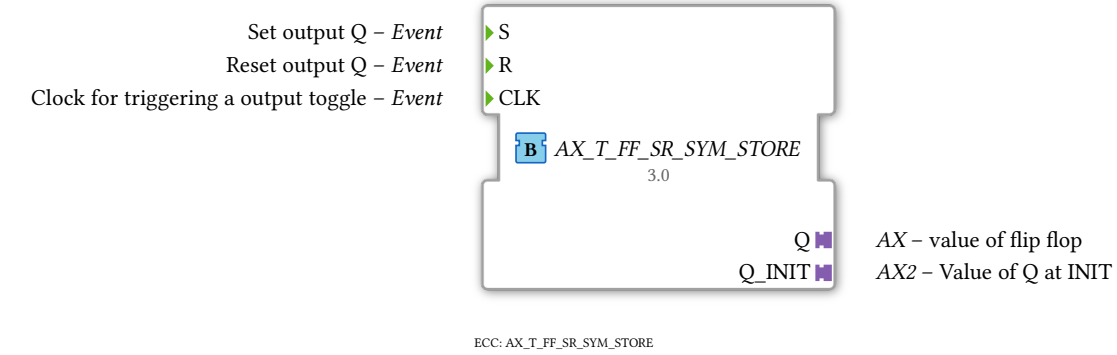
Function Block	Special Feature
`E_SR` (Standard IEC 61499)	Pure set/reset flip-flop without toggle and without INIT symmetry.
`E_RS`	Like E_SR, but Reset takes precedence.
`E_Toggle`	Toggle function only, no Set/Reset, no Initialization.
`AX_T_FF_SR_SYM_INIT`	Combines Set, Reset, **Toggle**, **Symmetric INIT Preset**, and Event Qualifier `QI`.

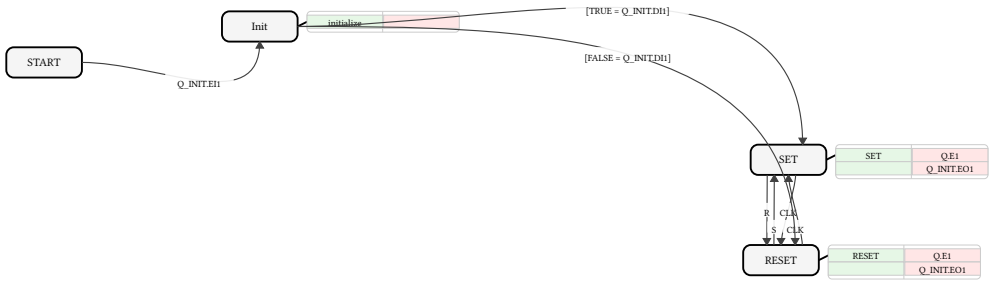
The function block described here thus offers greater flexibility for applications requiring multiple operating modes and a defined initial state.

Conclusion

The **AX_T_FF_SR_SYM_INIT** is a versatile and robust function block for binary state control in IEC 61499 systems. Its combination of Set, Reset, and Toggle functions, coupled with a freely configurable start state via `Q_INIT` and the qualifier `QI`, enables a compact and reliable implementation of many automation requirements. Thanks to its well-designed INIT handling, it is particularly suitable for applications requiring defined and reproducible start-up behavior.

AX_T_FF_SR_SYM_STORE





• – ○ • – ○ • – ○ •

Introduction

The function block `AX_T_FF_SR_SYM_STORE` implements an event-driven, bistable flip-flop with set (S), reset (R), and toggle (CLK) functions. Its special feature is its symmetrical startup behavior: The initial state at system startup can be defined and saved via a special adapter (`Q_INIT`).

Interface Structure

Event Inputs

Name	Type	Comment
S	Event	Sets the output `Q` to TRUE
R	Event	Sets output `Q` to FALSE
CLK	Event	Toggles output `Q` (TRUE ↔ FALSE)

Event Outputs

No direct event outputs are available. Output data is provided exclusively via the adapter interfaces.

Data Inputs

No direct data inputs. Initial value and state are read and output via adapters.

Data Outputs

No direct data outputs. The current state is made available via the adapters.

Adapters

Name	Type	Comment
Q	adapter::types::unidirectional::AX	Returns the current state of the flip-flop (single-wire Boolean value)
Q_INIT	adapter::types::bidirectional::AX2	Bidirectional interface for the initial value of `Q` at INIT (includes three inputs/outputs: e.g., DI1/DO1, EI1/EO1)

Functionality

The function block operates as an SR flip-flop (set-reset) with an additional toggle function. The process is divided into initialization and operation:

1. **Initialization (START → Init)**

After system startup, the function block is in the START state. As soon as the event `EI1` arrives at the adapter `Q_INIT`, it switches to the Init state. There, the initial value provided via `Q_INIT.DI1` is read.

- If the initial value is FALSE, the system subsequently enters the RESET state.
- If the initial value is TRUE, the system enters the SET state.

In both cases, the corresponding outputs `Q.D1` and `Q_INIT.DO1` are set to their initial values.

1. **Operation (SET ↔ RESET)**

Starting from SET or RESET, each event at inputs S, R, or CLK toggles the state:

- **S** → switches to SET (sets Q = TRUE)
- **R** → switches to RESET (sets Q = FALSE)
- **CLK** → toggles the state (from SET to RESET or vice versa)

With each state change, the output data is updated via the adapters, and the events `Q.E1` and `Q_INIT.EO1` are output.

Technical Features

- **Adapter-Based Input/Output**

Instead of classic data inputs/outputs, the module uses adapter interfaces. This enables flexible coupling with other modules or system components that also support the adapter protocol.

- **Symmetrical Startup Behavior**

The bidirectional adapter `Q_INIT` allows the initial state to be set externally and read after initialization. This ensures deterministic restart behavior – particularly important for safety-critical or stateful applications.

- **Toggle Function (CLK)**

In addition to the classic set and reset events, the clock input CLK allows for easy switching of the output without requiring a separate set/reset signal.

- **No Data Inputs/Outputs**

This functional reduction simplifies the interface and avoids confusion with the adapters.

State Overview

State	Description	Output on Event
START	Initial Wait State	Waits for initialization event via Q_INIT.EI1
Init	Initialization	Reads Q_INIT.DI1, sets Q and Q_INIT.DO1 accordingly

SET	Output Q = TRUE	Processes S, R, CLK (see Transitions)
RESET	Output Q = FALSE	Processes S, R, CLK (see Transitions)

State Transitions (Simplified)

- START ⟶ Init (Event Q_INIT.EI1)
- Init ⟶ SET (if Q_INIT.DI1 = TRUE)
- Init ⟶ RESET (if Q_INIT.DI1 = FALSE)
- SET ⟶ RESET (on Event R or CLK)
- RESET ⟶ SET (on Event S or CLK)

Application Scenarios

- **Machine State Storage**

In production plants, this function block can store the current operating mode (e.g., “On” or “Off”) and restore it upon restart.

- **Initialization of Control Sequences**

Scenarios requiring a defined start value (e.g., valve position at the beginning of a batch) benefit from the symmetrical INIT behavior.

- **Event-Driven Toggle Functions**

For example, toggling an indicator light with each key press, without requiring separate on and off commands.

Comparison with Similar Function Blocks

- **Standard E_SR (E_R-SET)**

Simple set-reset flip-flop without toggle functionality and without symmetric start initialization. The initial state is usually undefined or must be set externally.

- **E_RS**

Offers reset prioritization, but also no toggle or INIT functionality.

- **Toggle Function Blocks (e.g., E_Toggle)**

Pure toggle function blocks usually only have a clock input and no bidirectional adapter for initialization. The AX_T_FF_SR_SYM_STORE combines all three functions in one function block.

- **Adapter-Based Function Blocks**

A few function blocks in the 4diac library use adapters for state output. This design allows for a clean separation of event and data paths and facilitates reuse in complex systems.

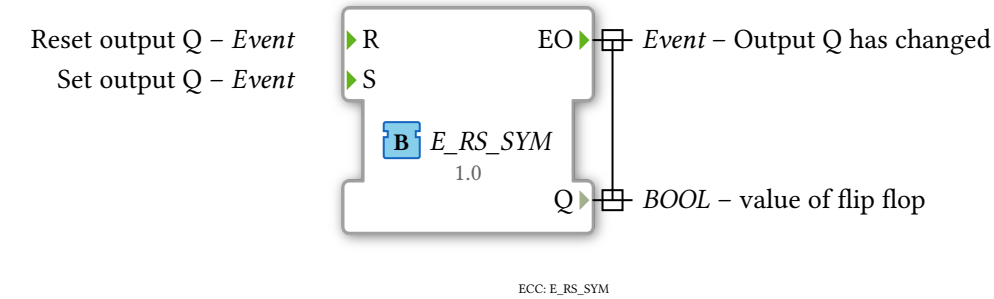
Conclusion

The `AX_T_FF_SR_SYM_STORE` is a powerful and flexible function block for stateful control tasks. Its combination of Set, Reset, and Toggle functions, coupled with symmetrical startup behavior via adapters, makes it ideal for applications requiring defined restart behavior. Through the use of adapters, it integrates seamlessly into modular and distributed automation systems.

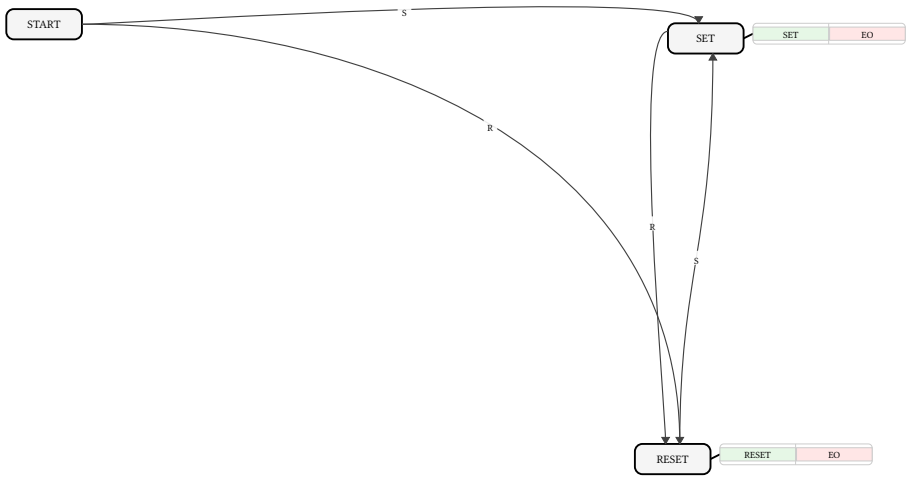
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

E_RS_SYM



ECC: E_RS_SYM



• – ○ • – ○ • – ○ •

Introduction

The function block **E_RS_SYM** is an event-driven, bistable flip-flop with symmetrical start behavior. It implements set-reset functionality, where the output Q is defined by both a set and a reset event. Unlike a classic RS flip-flop, this component has an explicit start state that reacts to both events equally.

Interface Structure

Event Inputs

Name	Type	Description
R	Event	Reset of output Q
S	Event	Set of output Q

Event Outputs

Name	Type	Description	With Data
EO	Event	Output Q has changed	Yes (with Q)

Data Inputs

No data inputs available.

Data Outputs

Name	Type	Description
Q	BOOL	Flip-flop value (TRUE/FALSE)

Adapters

No adapters available.

Functionality

After being switched on, the module is in the **START** state. From this state, an event at S triggers the **SET** state, and an event at R triggers the **RESET** state. In each case, the output Q is set to TRUE (SET) or FALSE (RESET), and the event EO is output.

- In the **SET** state, an event at R returns the module to the **RESET** state.
- In the **RESET** state, an event at S returns the module to the **SET** state.
- The output Q is only updated upon a state change (i.e., transition to SET or RESET).

The symmetrical start behavior means that both a Set and a Reset event are accepted immediately after initialization – there is no default setting for Q.

Technical Features

- **Symmetrical Start Behavior:** No output value is set in the START state. The device responds equally to the first arriving event (S or R). This distinguishes it from typical RS flip-flops, which usually deliver a defined (often FALSE) initial value after startup.
- **Event-Driven Output:** The event EO is triggered precisely when the value of Q changes. Thus, a signal change is explicitly transmitted.
- **No Data Inputs:** This function block operates purely event-driven without additional data parameters.

State Overview

The function block has three states:

State	Description
START	Initial state after power-on; waits for the first S or R event.
SET	Q = TRUE; can be switched to the RESET state by R.
RESET	Q = FALSE; can be switched to the SET state by S.

Transitions:

- START ⟶ SET on event S
- START ⟶ RESET on event R
- SET ⟶ RESET on event R
- RESET ⟶ SET on event S

Application Scenarios

- **Initialization of memory bits** in controllers where the initial value should not be predefined.
- **Symmetrical switching logic** that allows both setting and resetting a marker at the beginning.
- **Alternative to classic RS holders** when a defined startup behavior must be avoided (e.g., in safety-critical applications where an undefined initial state is tolerated).

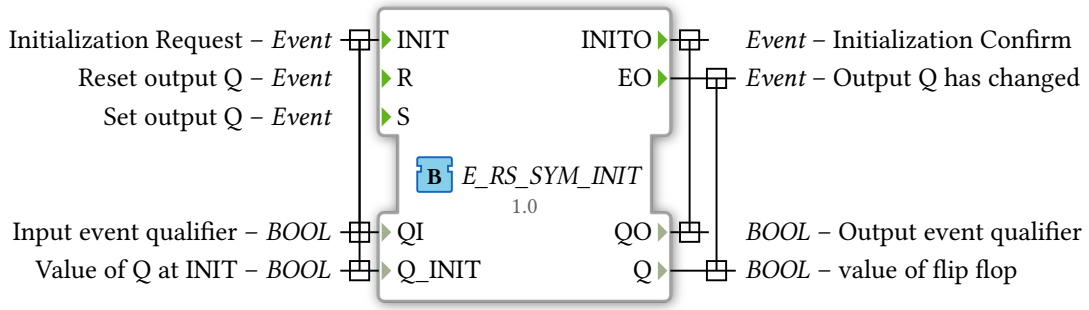
Comparison with Similar Function Blocks

Function Block	Property
E_RS	Classic RS flip-flop with typically asymmetric start behavior (Q initial FALSE).
E_RS_SYM	Like E_RS, but with symmetric start behavior – no preset value, start state accepts both events equally.
SR flip-flop	Similar function, but often with priority for Set or Reset; E_RS_SYM is priority-neutral.

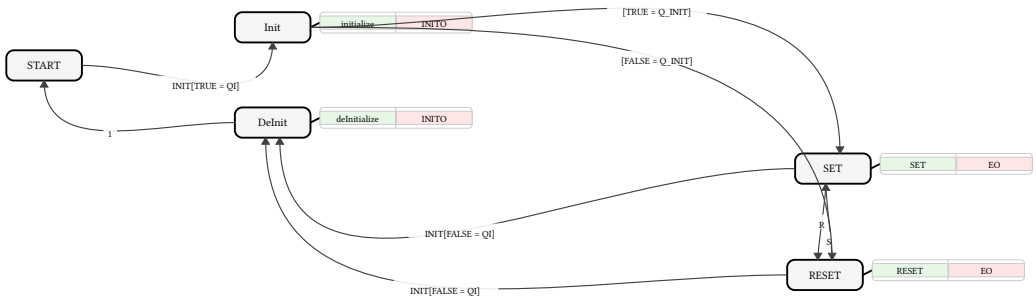
Conclusion

The **E_RS_SYM** is suitable for all applications that require simple, yet start-symmetric memory behavior. Due to the clean separation of event change (EO) and data output (Q), it can be easily integrated into event-driven control architectures according to IEC 61499. Its clear state machine makes it reliable and easy to understand.

E_RS_SYM_INIT



ECC: E_RS_SYM_INIT



• – ○ • – ○ • – ○ •

Introduction

The function block **E_RS_SYM_INIT** is an event-driven, bistable toggle switch (flip-flop) with symmetrical startup behavior and explicit initialization. It implements the familiar RS flip-flop logic, where the output **Q** can be set to either TRUE or FALSE during initialization (event **INIT**) – depending on the value of the parameter **Q_INIT**. This enables defined behavior after a system startup or reinitialization.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Initialization Request; Takes the values of **QI** and **Q_INIT**

R	Event	Resets output **Q**
S	Event	Sets output **Q**

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization confirmation; passes the value of **QO**
EO	Event	Indicates that output **Q** has changed

Data Inputs

Variable	Type	Comment
QI	BOOL	Event Qualifier: Set/reset operations are only executed if *QI=TRUE*. During initialization, *QI* sets the value of **QO**.
Q_INIT	BOOL	Sets the initial value of **Q**, which is inherited during initialization.

Data Outputs

Variable	Type	Comment
QO	BOOL	Output Qualifier: Indicates whether an operation was valid (inheritance of **QI**).
Q	BOOL	The actual flip-flop output.

Adapters

No adapters available.

Functionality

The **E_RS_SYM_INIT** operates as a state machine with five states: **START**, **Init**, **DeInit**, **SET**, and **RESET**.

- **Start Behavior**: After an initial **INIT** event with **QI=TRUE**, the block transitions to the **Init** state. Depending on the value of **Q_INIT**, it transitions either to **SET** (**Q** = TRUE) or to **RESET** (**Q** = FALSE). The output qualifier **QO** takes the value of **QI** (i.e., TRUE) and is output via **INITO**.
- **Setting and Resetting**:
- In the **SET** state, an **R** event leads to the **RESET** state; the action sets **Q** to FALSE (if **QI=TRUE**).
- In the **RESET** state, an **S** event leads to the **SET** state; The action sets **Q** to TRUE (if **QI=TRUE**).
- On every state change, **QO** is set to the current value of **QI**, and the **EO** event is output.
- **Deinitialization**: As soon as an **INIT** event occurs while **QI=FALSE** (e.g., deinitialization), the block transitions from **SET** or **RESET** to the **DeInit** state. There, **QO** is set to FALSE, and the block returns to the **START** state via **INITO**. The flip-flop is not reset – **Q** retains its previous value.
- **Qualifier QI**: The actual change to **Q** only occurs if **QI=TRUE**. If **QI=FALSE**, the set and reset signals are ignored, but **QO** still passes the value of **QI**. This enables conditional behavior, e.g.,... B. for valid/invalid releases.

Technical Features

- **Symmetrical Start Behavior**: The initial state of **Q** is explicitly defined via the parameter **Q_INIT**. This distinguishes the function block from a standard RS flip-flop, whose start state is undefined.
- **INIT as an Event with Parameters**: The INIT event input carries the values **QI** and **Q_INIT** simultaneously, clearly separating initialization and deinitialization.
- **QC (Event Qualifier) Passed Through**: For every valid operation (Set, Reset, or INIT), **QO** is set to the value of **QI**, allowing the calling application to verify the operation's validity.
- **Five-State Machine**: The separation of start, initialization, and deinitialization states enables robust behavior, particularly in automation systems with cyclic restarts.

State Overview

State	Description	Action on Entry	Exit Event
START	Initial sleep state after system startup or deinitialization	none	–
INIT	Initialization requested (**INIT** with **QI=TRUE**)	sets **QO** = **QI**	INITO
DeInit	Deinitialization requested (**INIT** with **QI=FALSE**)	sets **QO** = FALSE	INITO
SET	Flip-flop is set (**Q** = **TRUE**)	Set **Q** to TRUE (only if **QI=TRUE**)	EO
RESET	Flip-flop is reset (**Q** = **FALSE**)	Set **Q** to FALSE (only if **QI=TRUE**)	EO

Transitions:

- START ⟶ Init: INIT ∧ (QI = TRUE)
- Init ⟶ SET: Q_INIT = TRUE
- Init ⟶ RESET: Q_INIT = FALSE
- SET ⟶ RESET: R
- RESET ⟶ SET: S
- SET ⟶ DeInit: INIT ∧ (QI = FALSE)
- RESET ⟶ DeInit: INIT ∧ (QI = FALSE)
- DeInit ⟶ START: 1 (always)

Application Scenarios

- **Controllers with Defined Power-On Behavior**: If a specific start value is required for a flag or output after a restart of the automation system (e.g., TRUE for “system running” or FALSE for “shut down”), **Q_INIT** can be set accordingly.
- SET ⟶ DeInit: INIT ∧ (QI = FALSE)
- SET ⟶ START: 1 (always)

** Application Scenarios**

** Controllers with Defined Power-On Behavior**: If a specific start value is required for a flag or output after a restart of the automation system (e.g., TRUE for “system running” or FALSE for “shut down”), **Q_INIT** can be set accordingly.

** * **Safe Resets in Case of Faults**: By deinitializing (INIT with QI=FALSE), the function block can be reset to a defined “initial state” without clearing the current value of **Q**. This is useful, for example, to implement a “restart after a fault.”

- **Qualified Set and Reset Operations**: A higher-level condition (e.g., “Enable active operation”) can be controlled via **QI**. Only when **QI** = TRUE do setting and resetting actually affect **Q**.
- **Initialization of State Machines**: This function block is ideally suited as a basic building block for custom state machines that are intended to assume a specific initial state upon INIT.

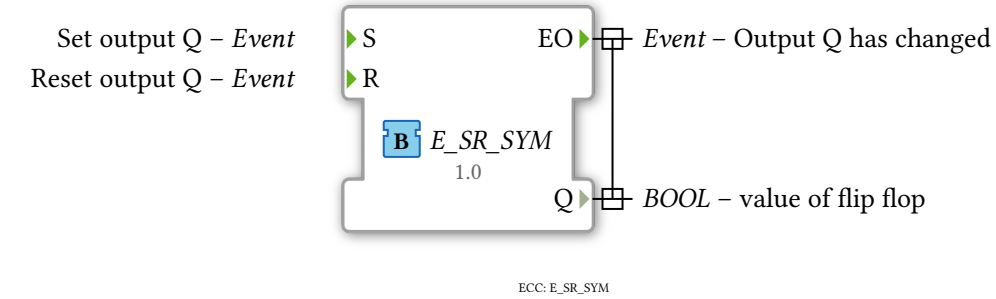
Comparison with Similar Function Blocks

Function Block	Difference to E_RS_SYM_INIT	
E_RS (Standard RS Flip-Flop)	No explicit start behavior; **Q** is not defined during INIT. **Q_INIT** is missing.	
E_SR (Set-Dominant Flip-Flop)	Set dominates over Reset; no initializable start state.	
E_RS_SYM	Symmetric RS logic without an INIT function; no start behavior.	
E_RS_SYM	**E_RS_SYM_INIT** (this one)	Combines symmetrical RS logic with selectable initial value and deinitialization.

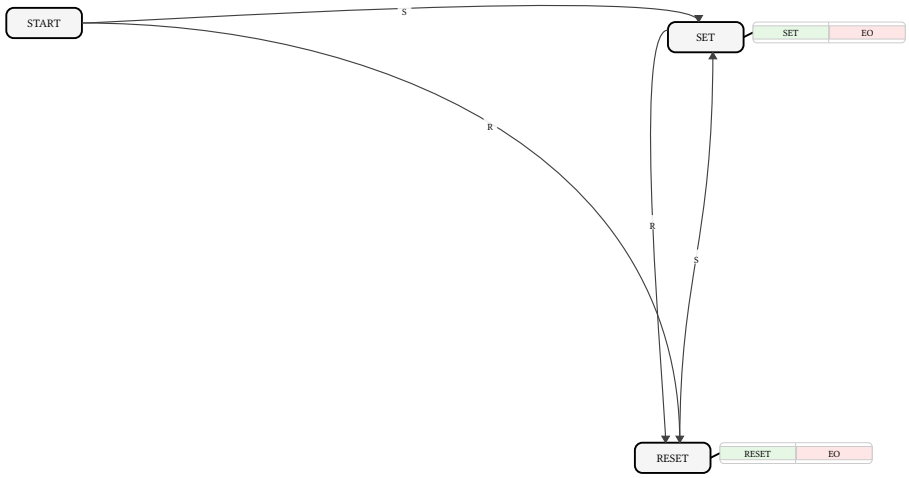
Conclusion

E_RS_SYM_INIT is a powerful and flexible IEC 61499-1 function block for event-driven bistable toggle switches. Due to its explicit initialization with **Q_INIT** and the option for deinitialization while retaining the output value, it is particularly suitable for safety-related and automation applications where defined startup behavior and a clear separation of initialization and operation are required. The inclusion of a **QI** qualifier also enables efficient integration into complex sequence control systems.

E_SR_SYM



ECC: E_SR_SYM



• – ○ • – ○ • – ○ •

Introduction

The E_SR_SYM function block is an event-driven bistable element with symmetric start behavior. It implements a flip-flop that responds to set and reset events and whose initial state is not predetermined but is determined by the first arriving event.

Interface Structure

Event Inputs

- **S** (Event): Sets the output (Q) to `TRUE`.
- **R** (Event): Sets the output (Q) to `FALSE`.

Event Outputs

- **EO** (Event): Triggered when the value of (Q) changes (after an S or R event). The output data memory (Q) is assigned to this event (`With Var="Q"`).

Data Inputs

None.

Data Outputs

- **Q** (BOOL): Current logical state of the flip-flop (`TRUE` or `FALSE`).

Adapters

None.

Functionality

The function block operates in a state-based manner and has three states:

- **START** – Initial state after powering on or resetting the controller.
- **SET** – State after a Set event.
- **RESET** – State after a reset event.

If an event arrives at **S** while the device is in the START state, it switches to the SET state, sets (Q := \text{TRUE}), and sends the **EO** event. If, instead, an event arrives at **R**, it switches to RESET, sets (Q := \text{FALSE}), and also sends **EO**.

If the device is already in one of the active states (SET or RESET), it can be switched by the other event. This results in symmetrical behavior: Neither S nor R has a priority – the last received pulse determines the output.

If the device is already in one of the active states (SET or RESET), it can be switched by the other event.

Technical Features

- **Symmetrical Start Behavior:** Unlike classic SR flip-flops, which deliver a fixed initial value (usually `FALSE`) after startup, E_SR_SYM waits in the START state for the first event. Until then, the output (Q) remains in an undefined or unset state. This allows for flexible initialization by the environment.
- **Three-State Machine:** The ECC uses the states START, SET, and RESET, as well as two algorithms (SET, RESET) in structured text.
- **Compact Interface:** Only two event inputs, one event output, and one data output – making it easy to integrate into larger control systems.

State Overview

| State | Meaning | Action on Entry | Trigger for Transitions | | |———|————|———————|——————————| | **START** | Initial state | None (Waiting for first event) | S → SET, R → RESET | | **SET** | Output (Q) is `TRUE` | Execute algorithm SET ((Q := \text{TRUE})), send EO | R → RESET | | **RESET** | Output (Q) is `FALSE` | Execute algorithm RESET ((Q := \text{FALSE})), send EO | S → SET |

Transitions:

- START → SET at **S**
- START → RESET at **R**
- SET → RESET at **R**
- RESET → SET at **S**

Application Scenarios

- **Event-driven enable signals:** A binary enable signal is set by a set pulse and reset by a reset pulse.
- **Initially indeterminate systems:** Startup behavior where the first valid command (set or reset) determines the subsequent sequence.
- **Simple two-point control:** Switching actuators (e.g., valve open/close) using external command transmitters.
- **Toggle replacement:** Combination with downstream logic blocks also enables toggle functions.

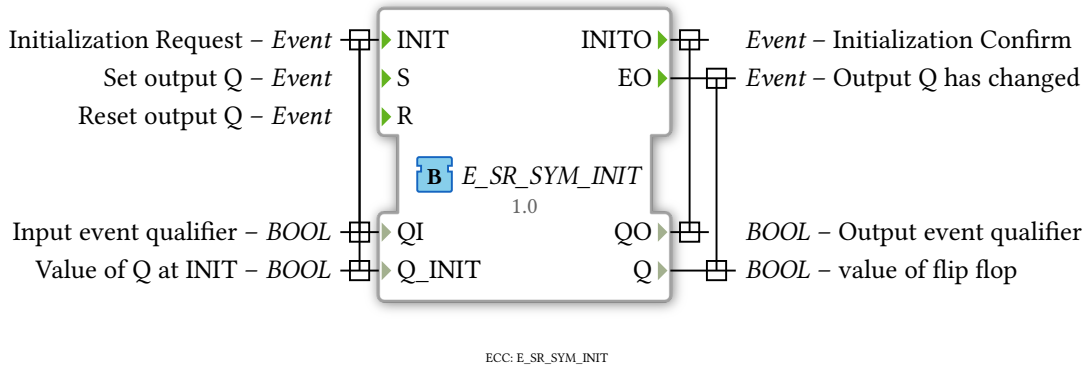
Comparison with Similar Function Blocks

The standard function block **E_SR** (or E_RS) typically has a predefined initial state (usually `FALSE`) and immediately resets to the RESET state after startup. In contrast, the initial state of E_SR_SYM is neutral (START), allowing for symmetrical handling of both events. Unlike dominant variants (E_SR with set dominance or E_RS with reset dominance), there is no priority here – the last event to arrive determines the output.

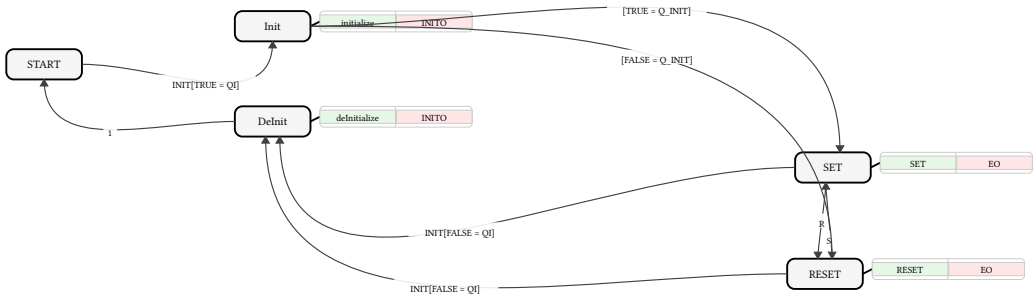
Conclusion

The function block **E_SR_SYM** offers a flexible, event-driven bistable memory cell with symmetrical startup behavior. Its simple state machine and clear interface make it particularly suitable for control tasks where the initial state is not predefined but rather determined by the first impulse from the environment. This distinguishes it from classic SR modules and expands the toolbox for event-based automation solutions.

E_SR_SYM_INIT



ECC: E_SR_SYM_INIT



• – ○ • – ○ • – ○ •

Introduction

The function block `E_SR_SYM_INIT` implements an event-driven, bistable flip-flop with symmetric startup behavior and an INIT interface. It extends a simple SR flip-flop by adding the ability to set the output `Q` to a predefined value (`Q_INIT`) and perform deinitialization during the INIT event. The input qualification `QI` controls whether the operations (S, R, INIT) actually affect `Q`.

Interface Structure

Event Inputs

Event	Type	Comment
`INIT`	EInit	Initialization request; evaluated with `QI` and `Q_INIT`

S	Event	Set output Q to TRUE (only effective if QI = TRUE)
R	Event	Set output Q to FALSE (only effective if QI = TRUE)

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization/Deinitialization Confirmation
EO	Event	Triggered when Q has changed (by S or R)

Data Inputs

Variable	Type	Comment
QI	BOOL	Input Qualifier; controls whether the operations (S, R, INIT) affect Q
Q_INIT	BOOL	Value that Q should assume upon successful initialization (INIT with QI = TRUE)

Data Outputs

Variable	Type	Comment
QO	BOOL	Output qualifier; returns the value of QI during the last action (FALSE upon deinitialization)
Q	BOOL	Value of the flip-flop (bistable output)

Adapter

None.

Functionality

The function block has five states: START, Init, DeInit, SET, and RESET.

- **START**: Idle state after initialization.
- **INIT**: Achieved by INIT with QI = TRUE. The algorithm initialize sets QO := QI. The system then switches to state SET (if Q_INIT = TRUE) or RESET (if Q_INIT = FALSE), depending on Q_INIT. During this process, Q is set to the value of Q_INIT (via the algorithms SET and RESET, respectively), and EO is triggered.

Q ...8qz is then set to the value of Q_INIT (via the algorithms SET and RESET, respectively). - **DeInit**: This is achieved by INIT with QI = FALSE from the states SET or RESET. The algorithm deInitialize sets QO := FALSE. The function block then automatically returns to the state START.

- **SET** and **RESET**: Normal operating states of the flip-flop. On a S event, the algorithm SET is executed: QO := QI, and if QI = TRUE, Q := TRUE is set; EO is triggered. On a R event, the algorithm Q := FALSE is executed analogously. A INIT event with QI = FALSE returns to DeInit. The output INIT with QI = TRUE would only be possible from the state START.

The qualification QI acts as an enable signal: Only if QI = TRUE is active do S, R, and the initialization affect the output Q. The output QO reflects the last valid value of QI (except after DeInit, in which case it is QO = FALSE).

The qualification QI acts as an enable signal: S, R, and the initialization only affect the output Q. The output QO reflects the last valid value of QI (except after DeInit, in which case it is QO = FALSE).

Technical Features

- **Symmetrical Start-up**: After initialization, the value of Q is determined solely by Q_INIT – regardless of the previous state. This enables deterministic behavior after startup.
- **Qualified Operations**: All actions (S, R, INIT) are only executed if QI = TRUE is active. This allows the function block to be switched on and off, for example, via a higher-level controller.

- **Deinitialization**: A `INIT` followed by `QI = FALSE` puts the function block into a defined deinitialization state (`DeInit`), which prevents any further actions until a subsequent `INIT` followed by `QI = TRUE` occurs.
- **No QO Storage**: `QO` is reset with each transition; no separate storage occurs.

State Overview



Application Scenarios

- **Controllers that require a defined output state after power-up** (e.g., machines: valve closed or open).
- **Systems with safety-critical initialization**, where the output may only be set after authorization by a qualifier (`QI`).
- **Block chains**, where deinitialization (e.g., during a system reset) should return the function block to a default state without the actual set/reset signals still being active.

Comparison with Similar Function Blocks

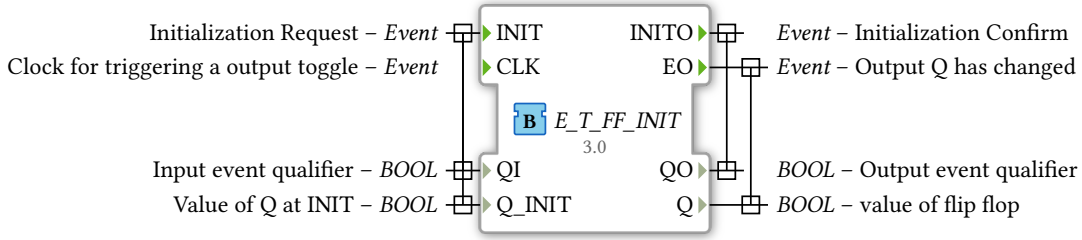
- **E_SR** (Standard SR Flip-Flop): Has no initialization behavior; it starts undefined or with the last value. `E_SR_SYM_INIT` extends this with defined initialization and deinitialization.
- **E_RS**: Swaps set/reset priority, but without an initialization mechanism.
- **E_SR_SYM**: Symmetric SR flip-flop without an INIT interface; `E_SR_SYM_INIT` adds the INIT startup logic.

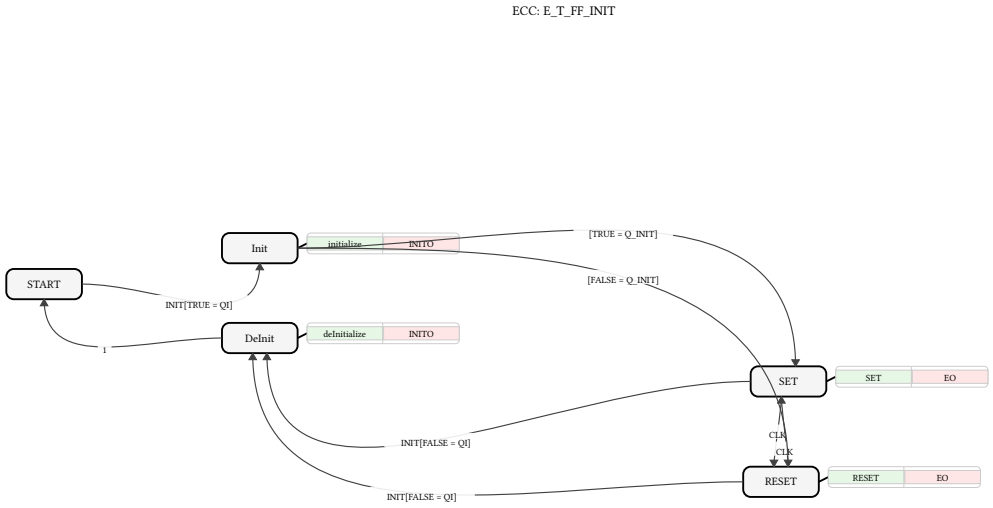
The advantage of `E_SR_SYM_INIT` lies in the combination of qualified initialization with an explicit deinitialization path.

Conclusion

The function block `E_SR_SYM_INIT` offers a robust, qualified flip-flop with a defined startup state. Its symmetric startup logic and deinitialization capability make it ideal for applications requiring reproducible behavior after system startup or reset. The separation of events and qualification allows for flexible integration into higher-level control structures.

E_T_FF_INIT





• – ○ • – ○ • – ○ •

The function block **E_T_FF_INIT** implements a **toggle flip-flop** with integrated initialization logic. It combines the classic behavior of a T flip-flop (switching the output *Q* on each *CLK* event) with targeted output setting during initialization. This block is particularly suitable for applications where a defined initial state is required after startup before normal toggle operation begins.

Event	Type	With Variables	Description
INIT	EInit	QI, Q_INIT	Initialization request; sets *Q* to *Q_INIT*.
CLK	Event	–	Clock signal; triggers the toggling of *Q* on each rising edge (if *QI* = TRUE).
Event	Type	With Variables	Description
–––––	–––––	–––––	–––––––––––––––––––
INITO	EInit	QO	Initialization confirmation; signals the end of initialization.
EO	Event	Q	Output event; is output after each state change of *Q*.
Variable	Type	Description	
–––––	–––	–––––––––––––––––––––	
QI	BOOL	Enable signal for the normal toggle function.	
Q_INIT	BOOL	Desired value of *Q* after initialization (if TRUE, *Q* is set).	
Variable	Type	Description	
–––––	–––	–––––––––––––––––––––	
QO	BOOL	Output qualifier; set to *QI* on INIT, to FALSE on de-initialization.	

Q	BOOL	Toggle output; toggles with each *CLK* (only if *QI* = TRUE).	

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters available.

Functionality

The module operates as a **state-controlled machine (ECC)** with five states: `START`, `Init`, `DeInit`, `SET`, and `RESET`.

1. **Initialization (state *Init*)**:

- Triggered by the **INIT** event, provided *QI* = TRUE.
- The algorithm `initialize` sets *QO* to the value of *QI* (here TRUE).
- The transition from *Init* depends on the value of *Q_INIT*:
- *Q_INIT* = TRUE ⟶ transition to the **SET** state (*Q* becomes TRUE).
- *Q_INIT* = FALSE ⟶ Transition to the **RESET** state (*Q* remains FALSE).
- After the transition, the **INITO** event is output.

1. **De-Initialization (State *DeInit*)**:

- Triggered by an **INIT** event with *QI* = FALSE (from the *SET* or *RESET* states).
- The algorithm `deInitialize` sets *QO* to FALSE.
- Subsequently, the transition back to the **START** start state occurs, and the **INITO** event is output.

1. **Normal Toggle Operation**:

- In the **SET** (*Q* = TRUE) and **RESET** (*Q* = FALSE) states.
- Each **CLK** event toggles between the two states.
- The algorithms `SET` and `RESET` set *QO* to the current value of *QI* and, if *QI* = TRUE, the output *Q* to the corresponding state (TRUE or FALSE).
- After each toggle, the event **EO** is output.

Important: The toggle function is only active if *QI* = TRUE. If *QI* = FALSE, *Q* remains unchanged (the IF condition in the algorithms prevents the change).

- **Combined Init/Toggle Logic**: This function block allows the output *Q* to be set to a defined value during initialization (*Q_INIT*). This distinguishes it from a simple T flip-flop, which usually starts in the RESET state.
- **Enable via QI**: The variable *QI* acts as a global enable. Switching occurs only when *QI* = TRUE at *CLK*. When *QI* = FALSE, INIT events can still be processed, but toggle operation is disabled.
- **Output Qualifier QO**: *QO* reflects the enable state – it is set to TRUE upon successful initialization and to FALSE upon de-initialization. This allows for easy monitoring of the block status.

State	Description	Incoming Transition(s)	Outgoing Action (Algorithm)	Outgoing Event
START	Idle state after initialization (or after de-initialization).	*DeInit* ⟶ START	–	–

Init	Initialization phase, sets *QO* and determines *Q*.	START → Init (when INIT & QI = TRUE)	`initialize`	INITO
DeInit	De-initialization phase, sets *QO* to FALSE.	SET/RESET → DeInit (when INIT & QI = FALSE)	`deInitialize`	INITO
SET	Normal toggle state: *Q* = TRUE.	Init → SET (when Q_INIT = TRUE) RESET → SET (when CLK)	`SET`	EO
RESET	Normal toggle state: *Q* = FALSE.	Init → RESET (when Q_INIT = FALSE) SET → RESET (when CLK)	`RESET`	EO

Transition conditions (simplified):

- `START → Init : INIT & (QI = TRUE)`
- `Init → SET : (Q_INIT = TRUE)`
- `Init → RESET : (Q_INIT = FALSE)`
- `SET ↔ RESET : CLK`
- `SET / RESET → DeInit : INIT & (QI = FALSE)`
- `DeInit → START`: always (condition = 1)

1. **Initializing a Machine with a Defined Start State**

A conveyor belt should always be in the "ready" position (*Q* = TRUE) after being switched on. This is achieved with *Q_INIT* = TRUE and *QI* = TRUE. A clock can then switch the belt between the two states via *CLK*.

1. **Safety-Controlled Toggle**

A safety signal *QI* enables the block. Toggle operations may only be performed when this signal is active. A failure of the enable signal (*QI* = FALSE) leads to de-initialization – the output *Q* remains unchanged and *QO* becomes FALSE.

1. **State Sequence with Initialization Block**

In a sequence control, *E_T_FF_INIT* is used as a flip-flop for a step. Initialization allows the step to be set to either active or inactive immediately after starting, without a separate set or reset pulse.

``## Comparison with Similar Components

Component	Special Feature
E_T_FF	Pure T flip-flop without initialization logic. Always starts in the RESET state.
E_FF	S-R or D flip-flop – allows setting and resetting via separate events.
E_SR	Memory component (set-reset) without toggle functionality.
E_T_FF_INIT	Combines toggle functionality with a freely selectable initial state and enable via *QI*.

Unlike *E_T_FF*, *E_T_FF_INIT* can set the output to TRUE during initialization and provides explicit enable for toggle operation via *QI*.

The function block **E_T_FF_INIT** is an extended variant of a T-flip-flop, specifically designed for controllers that require a defined initial state after startup. The combination of initialization (via *Q_INIT*), enable (via *QI*), and switching logic (via *CLK*) makes it highly versatile. It is suitable for safety-critical processes where operation is only permitted with active enable, as well as for applications requiring a reproducible startup state. The clearly structured state machine ensures deterministic and predictable behavior.

Technical Features

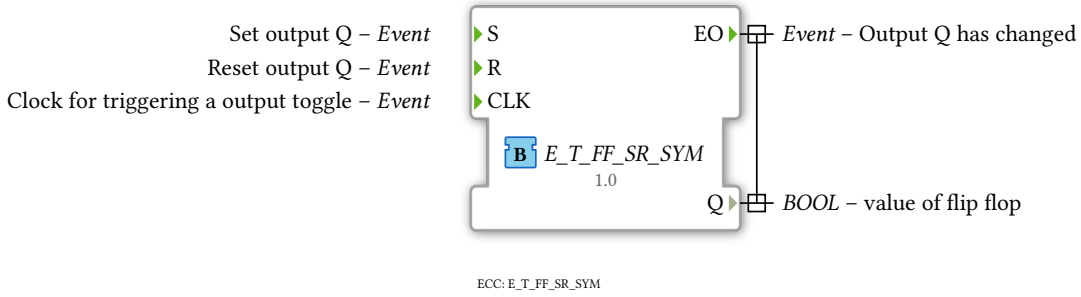
State Overview

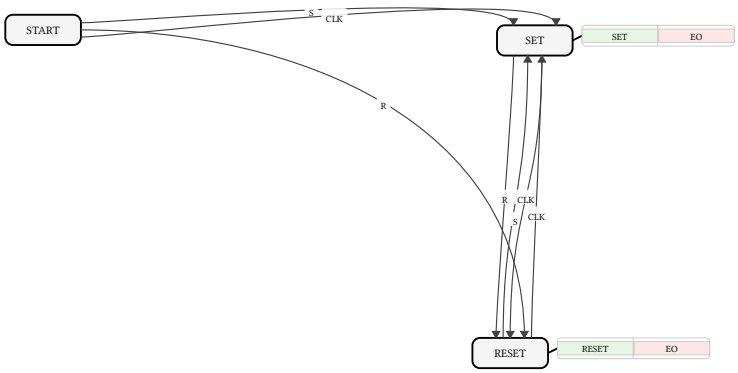
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

E_T_FF_SR_SYM





• – ○ • – ○ • – ○ •

Introduction

The function block **E_T_FF_SR_SYM** is an event-driven, bistable toggle switch with an integrated toggle function. It combines the properties of an SR flip-flop (set-reset) and a toggle flip-flop (toggles on clock). Its unique feature is its symmetrical startup behavior: In the initial state, both set and clock events result in the set state, while a reset event directly results in the reset state. This enables defined and predictable behavior immediately after commissioning.

Interface Structure

Event Inputs

Name	Type	Comment

S	Event	Set output Q (to TRUE)
R	Event	Reset output Q (to FALSE)
CLK	Event	Clock to toggle output Q

Event Outputs

Name	Type	With Variable	Comment
EO	Event	Q	Output Q has changed

Data Inputs

None.

Data Outputs

Name	Type	Comment
Q	BOOL	Flip-flop value (TRUE = set, FALSE = reset)

Adapters

None.

Functionality

The component operates as a finite automaton with three states: **START**, **SET**, and **RESET**.

- In the **START** state (initial state), a **S** event or a **CLK** event causes the component to transition to the **SET** state. A **R** event causes it to transition to the **RESET** state.
- In the **SET** state, an **R** event causes the component to transition to the **RESET** state, as does a **CLK** event. An **S** event would maintain the state (not explicitly modeled, but the output only changes upon transition).
- In the **RESET** state, an **S** event or a **CLK** event causes the component to transition to the **SET** state. An **R** event would maintain the state.

The corresponding algorithm is executed with each state change:

- **SET**: `Q := TRUE;`
- **RESET**: `Q := FALSE;`

The **EO** event is then output, which accompanies the new value of **Q**.

The **toggle** behavior is implemented by ensuring that a CLK event always toggles the current state – regardless of whether SET or RESET is active.

Technical Features

- No data inputs: Control is achieved exclusively via events. This reduces the number of required connections and simplifies timing analysis.
- Symmetrical start behavior: The initial state START is not equivalent to a defined output value. Instead, a defined subsequent state is immediately assumed with each of the three possible start events (S, R, CLK). This prevents undefined states after the start.
- All algorithms are implemented in ST (Structured Text) and consist of simple assignments.
- Output Q is only updated upon a state change – the EO event indicates precisely this change.

State Overview

State	Description
START	Initial state after activation of the function block
SET	Output Q = TRUE
RESET	Output Q = FALSE

Transitions (Event → Target State):

Initial State	Event	Target State
START	S	SET
START	R	RESET
START	CLK	SET
SET	R	RESET
SET	CLK	RESET
RESET	S	SET
RESET	CLK	SET

Note: S events remaining in the SET state or R events remaining in the RESET state are not explicitly mapped – the state is retained, and no EO output is generated.

Application Scenarios

- **Clock Switching Operations**: This function block is suitable for applications where a binary signal needs to be switched by a clock signal, but separate setting or resetting must also be possible (e.g., manual priority control).
- **Symmetrical Start Initialization**: In safety-critical systems, responding equally to S or CLK in the start state prevents the presence of an undefined value.
- **Pulse Width Modulation or Toggle Locking**: As a simple toggle function block with set/reset priority.

Comparison with Similar Function Blocks

Function Block	Difference
E_FF_SR	Pure SR flip-flop without toggle. No clock function.
E_FF_Toggle	Pure toggle flip-flop, no set/reset.
E_T_FF_SR	Similar, but without symmetric start behavior. Here, CLK in the START state may lead to undefined behavior.
E_T_FF_SR_SYM	Combines SR and toggle with defined start behavior for all input signals.

Conclusion

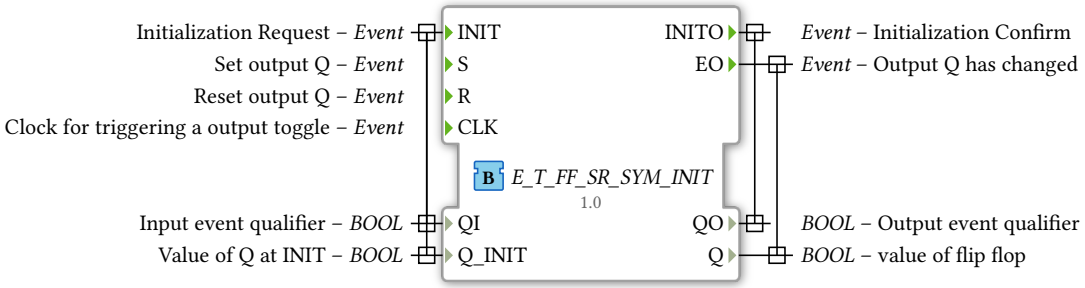
The **E_T_FF_SR_SYM** is a versatile event-driven flip-flop component that combines both set/reset and toggle functions. Its symmetric start behavior makes it particularly suitable for applications where a defined state must be reached immediately after activation – regardless of which event occurs first. Thanks to its purely event-driven control, it can be easily integrated into time-critical control processes.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 The PWM signal & infographic on ms-muc-docs.de

]

E_T_FF_SR_SYM_INIT



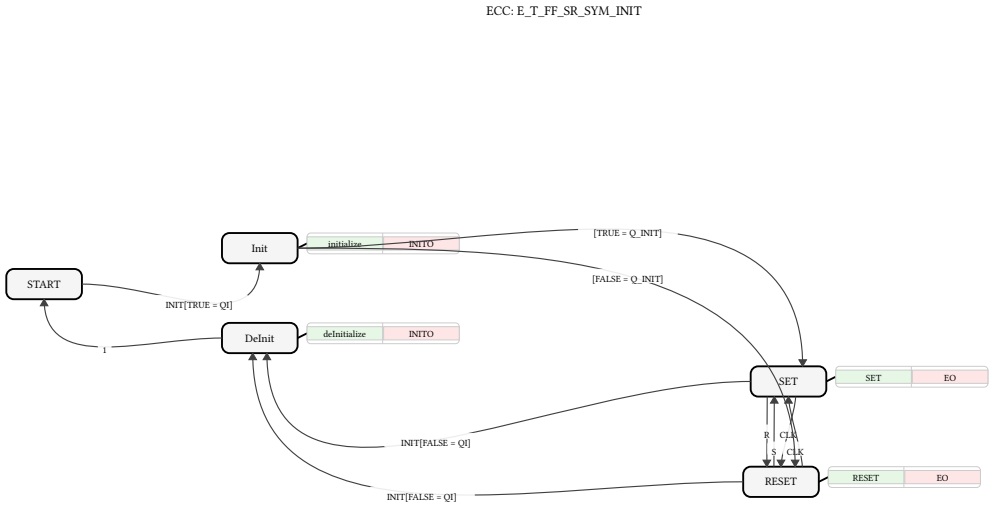


Image of the function block: (not available)

• — ○ • — ○ • — ○ •

Introduction

The function block `E_T_FF_SR_SYM_INIT` implements an event-driven, bistable flip-flop with set, reset, and toggle functionality. It features symmetrical startup behavior: Upon the initialization event (`INIT`), the output `Q` is set to the value specified at the input `Q_INIT`, provided the qualifier `QI` is active. The block combines the properties of an SR flip-flop and a toggle flip-flop in a single unit.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request
S	Event	Sets output Q (set)
R	Event	Resets output Q
CLK	Event	Clock for toggling Q

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Acknowledgement
EO	Event	Signals a change in Q

Data Inputs

Name	Type	Comment
QI	BOOL	Event qualifier (controls execution)

Q_INIT	BOOL	Value that Q should assume at INIT

Data Outputs

Name	Type	Comment
QO	BOOL	Output qualifier (mirror of QI)
Q	BOOL	Flip-flop value

Adapters

None.

Functionality

The function block traverses a finite state machine with five states: `START`, `Init`, `DeInit`, `SET`, and `RESET`.

- **Initialization (`INIT`)**

If a signal is present at the event input `INIT`, the function block distinguishes between the following states depending on the state of the qualifier `QI`:

- If `QI = TRUE`, the function block transitions to state `Init`. There, the output qualifier `QO` is set to the value of `QI` (i.e., `TRUE`), and the output `INITO` is triggered. Subsequently, the automaton—again depending on `Q_INIT`—enters state `SET` (if `Q_INIT = TRUE`) or state `RESET` (if `Q_INIT = FALSE`). Thus, `Q` receives the desired starting value.

If `QI = FALSE`, the function block transitions to state `DeInit`. There, `QO` is set to `FALSE`, and `INITO` is triggered. The machine then returns to its initial state.

- **Setting (`S`)**

When an event occurs at `S`, the algorithm `SET` is executed:
QOreceives the value ofQI. If QI = TRUEis present,Qis set toTRUE. The output EOis triggered to signal a change inQ`.

- **Reset (`R`)**

When an event occurs at `R`, the algorithm `RESET` is executed:
QOreceives the value ofQI. If QI = TRUEis present,Qis set toFALSE. In this case, EO` is also triggered.
QI = TRUEis set toQ. EO is also triggered in this case. - **Toggle (CLK`)**
An event at `CLK` always causes a state change:
If the automaton is in state `SET`, it switches to `RESET` and vice versa. The respective algorithm (`SET` or `RESET`) is executed with the current evaluation from `QI`. The output `Q` is therefore toggled at every clock cycle, provided `QI = TRUE` is in state.

Technical Features

- **Symmetrical Start-up Behavior**: The initial value of `Q` is determined by the configurable input `Q_INIT` and is not set to a fixed default. This allows the system to start reproducibly in a defined state.
- **Event Qualifier `QI`**: All actions (`SET`, `RESET`) are only executed if `QI = TRUE` is present. For `QI = FALSE`, the output qualifier `QO` is simply set to `FALSE`; the value of `Q` remains unchanged. This enables conditional control by upstream function blocks.
- **Deinitialization**: If a `INIT` event with `QI = FALSE` is received during runtime, the function block is deinitialized and enters its idle state (`START`). In this state, it does not respond to any further events until it is reinitialized.

State Overview

State	Description	Action on Occurrence
START	Waiting for initialization	–
Init	Initializing QO	`QO := QI`; Triggering `INITO`
DeInit	Deinitializing (if QI=false)	`QO := FALSE`; Triggering `INITO`

SET	Q is set (TRUE)	Q := TRUE (only if QI=true); Triggering EO
RESET	Q is reset (FALSE)	Q := FALSE (only if QI=true); Trigger EO

Q is reset (FALSE) | Q := FALSE (only if QI=true);
Q is reset (EO) ...80qz) | Q is reset (Q := FALSE) |
Q is reset (q Transitions:

- START ⟶ Init: at INIT and QI = TRUE
- START ⟶ DeInit: at INIT and QI = FALSE
- Init ⟶ SET: when Q_INIT = TRUE
- Init ⟶ RESET: when Q_INIT = FALSE
- SET ⇄ RESET : via S, R or CLK (each taking QI into account)
- SET ⟶ DeInit : if INIT is reused with QI = FALSE
- RESET ⟶ DeInit : if INIT is reused with QI = FALSE
- DeInit ⟶ START : always (transition with condition 1)

Application Scenarios

- **Control with defined start state**: In safety controllers or systems that require a specific initial state after a restart, Q_INIT can be set accordingly.
- **Toggle function for command switches**: A single push button (connected to CLK)) switches an output with each press – e.g., for lighting controls.
- **Conditional Set/Reset Logic**: The qualifier QI allows actions to be made dependent on higher-level conditions without losing the flip-flop's state.

Comparison with Similar Components

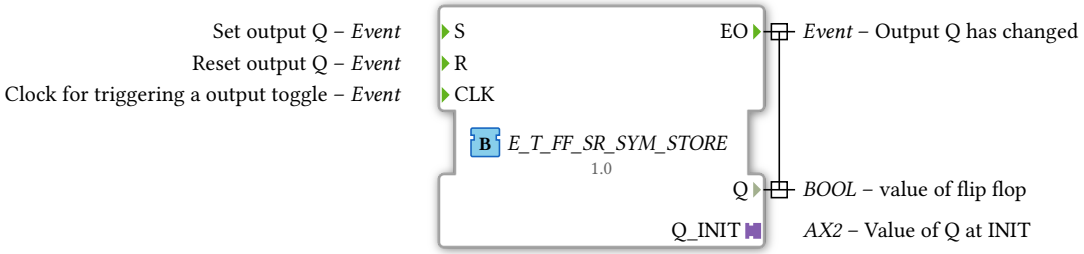
- **Standard SR Flip-Flop (E_SR)**: Offers only set and reset functionality, no toggle function, and no adjustable initial value.
- **Toggle Flip-Flop (E_TOGGLE)**: Only toggles on each clock cycle; no separate set/reset functionality and no defined initial value.
- **SR Flip-Flop with Initialization (E_SR_INIT)**: Similar, but without a toggle function.

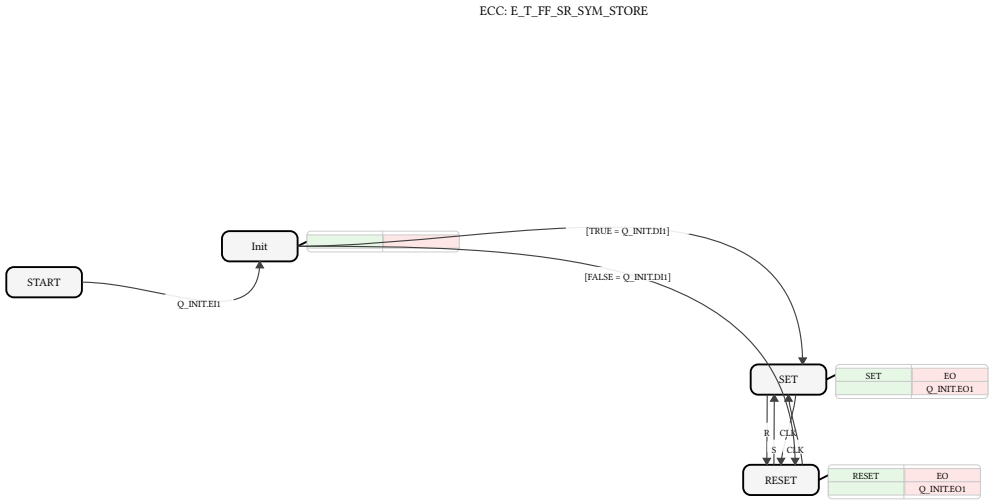
This function block combines all three functions (Set, Reset, Toggle) with flexible initialization behavior – a comprehensive solution for many event-driven control tasks.

Conclusion

E_T_FF_SR_SYM_INITis a versatile function block for IEC 61499-based event-driven automation. By combining Set, Reset, and Toggle functions with configurable initialization, it covers a wide range of requirements. Using the qualifierQI` increases flexibility and enables secure integration into complex sequence control systems.

E_T_FF_SR_SYM_STORE





• – ○ • – ○ • – ○ •

Introduction

The function block `E_T_FF_SR_SYM_STORE` implements an event-driven bistable flip-flop with the functions **Set** (S), **Reset** (R), and **Toggle** (CLK). Its special feature is the **symmetrical start-up behavior**: The initial state of the output Q is set via an adapter (`Q_INIT`) during startup or after a `INIT` event. This ensures that the function block behaves deterministically and repeatably at every start time – an important property for safety-critical or predictable automation systems.

Interface Structure

Event Inputs

Name	Type	Comment
`S`	Event	Set output Q to TRUE
`R`	Event	Reset output Q to FALSE
`CLK`	Event	Toggle – switches Q (TRUE ⟶ FALSE or FALSE ⟶ TRUE)

Event Outputs

Name	Type	With Data	Comment
`EO`	Event	Q	Indicates that the value of Q has changed

Data Inputs

None (all inputs are via events).

Data Outputs

Name	Type	Comment
`Q`	BOOL	Current value of the flip-flop (TRUE or FALSE)

Adapter

Name	Type	Comment
Q_INIT	adapter::types::bidirectional::AX2	Sets the initial value of Q on the INIT event. The adapter provides an event-driven, bidirectional channel: Q_INIT.EI1 triggers the initialization, and Q_INIT.DI1 returns the desired initial value (TRUE or FALSE). Additionally, the function block returns the set value via Q_INIT.DO1.

Functionality

The internal process is controlled by the finite state machine (ECC):

1. **START State:** After power is supplied or a system reset is performed, the function block is in state START.
2. **Initialization (State Init):** As soon as the event Q_INIT.EI1 occurs, state Init is reached. The following is then determined based on the Boolean value of Q_INIT.DI1:

- TRUE ⟶ Transition to state SET
- FALSE ⟶ Transition to state RESET

1. **Operating states SET and RESET:**

- **SET:** Sets Q := TRUE and Q_INIT.DO1 := TRUE. Sends output event EO and adapter output event Q_INIT.EO1.
- **RESET:** Sets Q := FALSE and Q_INIT.DO1 := FALSE. Also sends EO and Q_INIT.EO1.

1. **Toggling between SET and RESET:**

- Event S (when in RESET state) ⟶ toggles to SET.
- Event R (when in SET state) ⟶ toggles to RESET.
- Event CLK (always) ⟶ toggles between the two states.

The output EO is activated with every state change of Q.

Technical Features

- **Symmetrical Start-Up:** Unlike simple SR flip-flops with a fixed start value (e.g., FALSE), the start value can be configured via the adapter. This increases flexibility and reproducibility.
- **Toggle Function:** The CLK input allows changing the state without knowing the current value – useful for counters, flashers, or state machines.
- **Bidirectional Adapter Interface:** The device reports its set value back to the adapter (Q_INIT.DO1), so the initializer can confirm the final state.
- **Prioritization:** If S, R, or CLK arrive simultaneously, only the first processed event is considered (based on the event order in the 4diac IDE). In this specific design, CLK is applied as a toggle to the current state, while S and R set the state independently of the toggle. The state transitions show that CLK transitions from each state to the other.

State Overview

The state machine (ECC) comprises four states:

State	Description	Outgoing Transitions
START	Initial sleep state after power-on	⟶ Init at Q_INIT.EI1
Init	Initialization state – determines the start value	⟶ SET at TRUE = Q_INIT.DI1 ⟶ RESET at FALSE = Q_INIT.DI1
SET	Q = TRUE	⟶ RESET at R or CLK
RESET	Q = FALSE	⟶ SET at S or CLK

After exiting Init, the state SET or RESET is reached; the states START and Init are not traversed during normal operation.

Application Scenarios

- **Machine Control with Defined Start State:** A system should always start in a specific operating mode after a reset. Using `Q_INIT`, for example, a PLC configuration can be used to specify whether a valve position is open or closed at startup.
- **Toggling as a Flashing Function:** The `CLK` input can be triggered periodically (e.g., by a timer block) to implement a flashing signal.
- **Redundant Feedback:** The adapter output `Q_INIT.DO1` returns the set value – ideal for safety functions that require feedback on the switching state.

Comparison with Similar Blocks

Block	Properties	Difference to `E_T_FF_SR_SYM_STORE`
`E_SR_FF` (Standard SR Flip-Flop)	Set, reset; usually no toggle, no configurable start value	Missing toggle function; start value often fixed at FALSE
`E_RS_FF`	Set prioritized	Different prioritization; no toggle
`E_D_FF` (D Flip-Flop)	Receives a data input with clock	No set/reset without data; no symmetric start
`E_CTU` (Counter)	Counts events	No bistable behavior, but counter value

The `E_T_FF_SR_SYM_STORE`Combines SR functionality, toggle switching, and an initialized start value in a single component – a rare combination in the IEC 61499 world.

Conclusion

The **E_T_FF_SR_SYM_STORE** is a powerful event-driven flip-flop component for the 4diac IDE. It offers full SR functionality, a toggle option, and symmetrical start-up behavior configurable via an adapter. This makes it particularly suitable for applications requiring deterministic initialization and flexible switching mechanisms. The bidirectional adapter interface increases the integration depth in complex control architectures.

🌐 Related topic subpages on ms-muc-docs.de

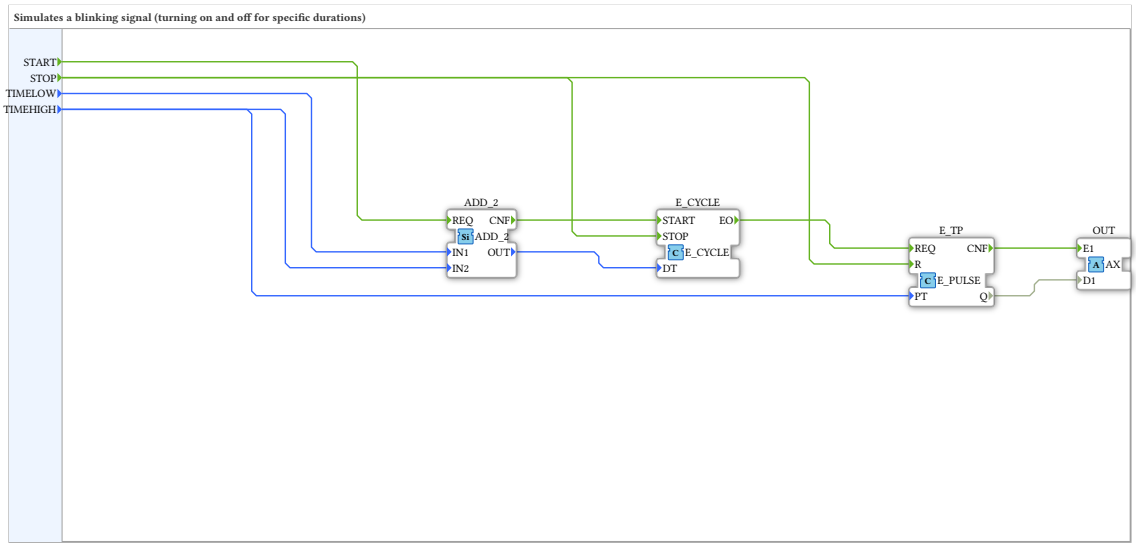
- 🌐 E_CTU Event Counter module on ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

signals

Here you will find all exercises from the `BOOL` workspace, listed in ascending order.

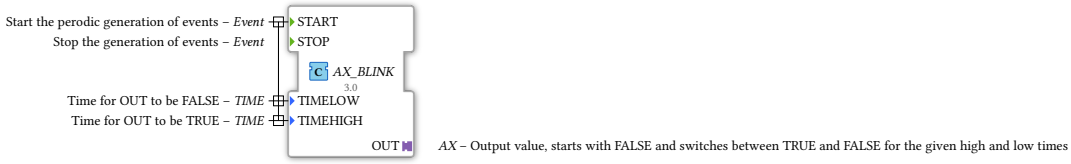
AX_BLINK



• – ○ • – ○ • – ○ •

Introduction

The AX_BLINK is a function block that outputs a flashing signal via an AX adapter.



Interface Structure

Event Inputs

- **START**: Starts the periodic generation of events; carries `TIMELOW` and `TIMEHIGH`.
- **STOP**: Stops the generation of events.

Data Inputs

- **TIMELOW** (TIME): Time for OUT to be FALSE.
- **TIMEHIGH** (TIME): Time for OUT to be TRUE.

Adapters

Plugs (Outputs):

- **OUT** (adapter::types::unidirectional::AX): Output value, starts with FALSE and switches between TRUE and FALSE for the given high and low times.

Functionality

After `START`, the function block alternately switches the output OUT off (FALSE) for the time `TIMELOW` and on (TRUE) for the time `TIMEHIGH`, until `STOP` arrives.

Technical Features

- Uses unidirectional adapters.

State Overview

State-based (flashing).

Application Scenarios

Flashing displays.

⚖️ Comparison with Similar Function Blocks

- **E_BLINK**: Standard flashing function block.

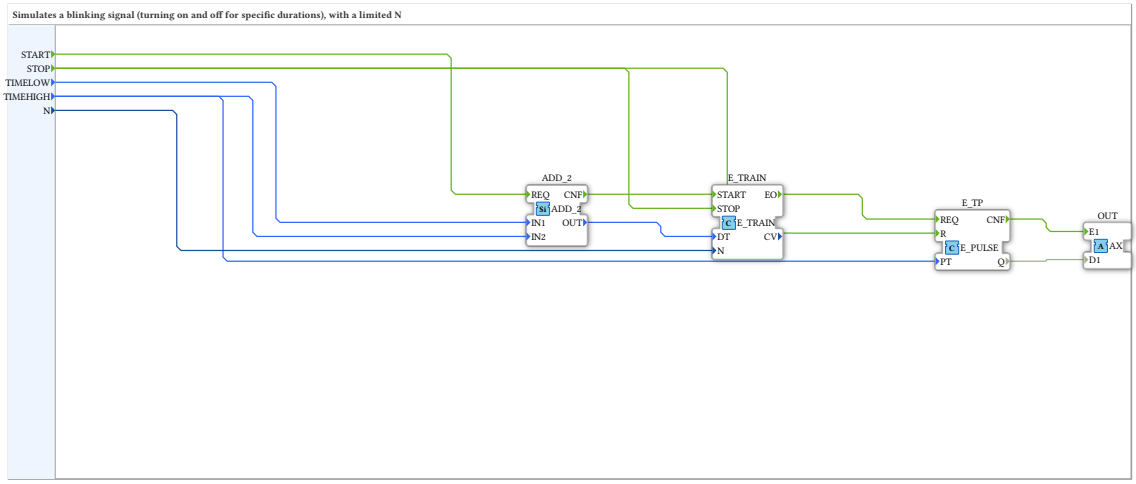
🛠️ Related Exercises

- Exercise_020f3_AX

Conclusion

Adapter-based blink module.

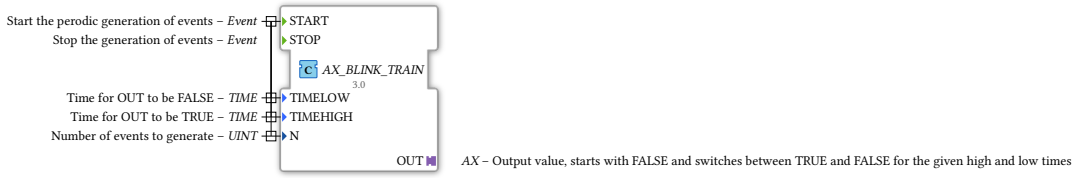
AX_BLINK_TRAIN



• — ○ • — ○ • — ○ •

Introduction

The AX_BLINK_TRAIN is a function block that outputs a flashing signal with a limited number N of events (defined on and off times) via an AX adapter.



Interface Structure

Event Inputs

- **START** (Event): Starts the periodic generation of events (with TIMELOW, TIMEHIGH, N).
- **STOP** (Event): Stops the generation.

Data Inputs

- **TIMELOW** (TIME): Time for OUT to be FALSE.
- **TIMEHIGH** (TIME): Time for OUT to be TRUE.
- **N** (UINT): Number of events to generate.

Adapters

Plugs (Outputs):

- **OUT** (adapter::types::unidirectional::AX): Output value, starts with FALSE and switches between TRUE and FALSE for the given high and low times.

Functionality

After `START`, the function block outputs a flashing signal at the adapter OUT: the output starts with FALSE and is TRUE for TIMEHIGH and FALSE for TIMELOW in each cycle. N events are generated; `STOP` ends the generation.

Technical Features

- Uses unidirectional adapters.

State Overview

State-based.

Application Scenarios

Railway crossings, warning signals.

⚖️ Comparison with similar modules

- No direct standard equivalent.

🛠️ Related Exercises

- Exercise_035a3_AX

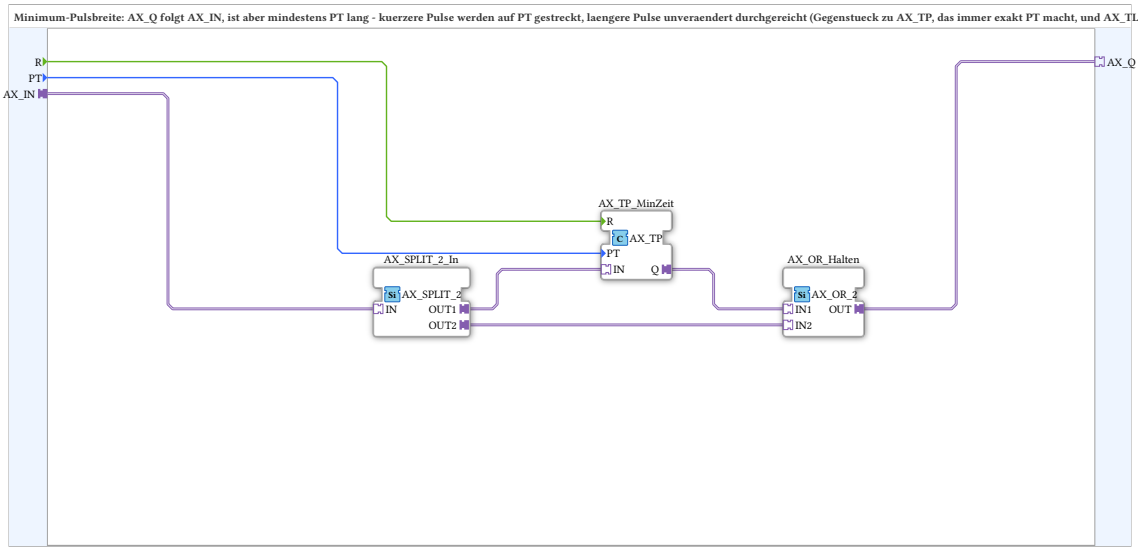
Conclusion

Adapter-based train flashing module.

timers

Here you will find all exercises from the `BOOL` workspace, listed in ascending order.

AX_MIN_TP

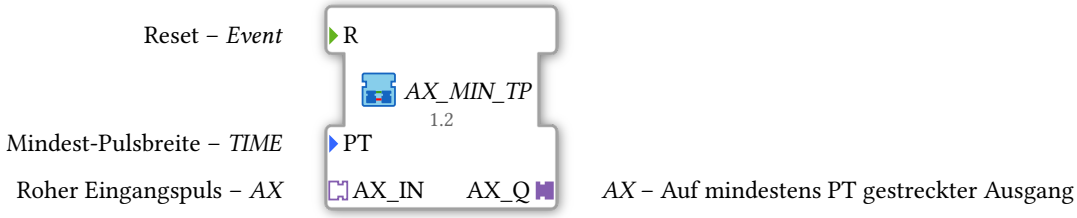


• – ○ • – ○ • – ○ •

Introduction

`AX_MIN_TP` is a minimum pulse width block: `AX_Q` follows `AX_IN`, but is **at least `PT` long**. Shorter pulses are stretched to `PT`, longer pulses are passed through unchanged. It is the counterpart of `AX_TP`, which always outputs exactly `PT`, and of `AX_TLIM`, which caps at a maximum of `PT`. Use case: the hopper floor lead-in in automatic mode needs a minimum time.

Safety note (Martin Schuder): The block itself has **no hard abort**. A limit switch that is to interrupt the pulse immediately at any time must be combined externally via `AX_AND(AX_Q, NOT limit switch)` **after** this block.



Function Blocks (FBs) Used

Sub-blocks: AX_MIN_TP

- **Type**: SubAppType
- **Internal FBs used**:
  - **AX_SPLIT_2_In**: `adapter::events::unidirectional::AX_SPLIT_2` - `AX_IN` is needed for `AX_TP_MinZeit.IN` **and** directly for `AX_OR_Halten`
  - **AX_TP_MinZeit**: `adapter::events::unidirectional::timers::AX_TP` - non-retriggerable: after the rising edge of `AX_IN` it always runs exactly `PT`, regardless of whether `AX_IN` drops again earlier
  - **AX_OR_Halten**: `adapter::booleanOperators::AX_OR_2` - `AX_IN OR AX_TP_MinZeit.Q`: holds the output as long as either the raw pulse is still present (longer pulses) or the minimum time is still running (shorter pulses)

Interface Structure

Event Inputs

- **R**: Reset (passed through to `AX_TP_MinZeit.R`)

Event Outputs

None - events travel via the adapters.

Data Inputs

- **PT** (`TIME`): Minimum pulse width

Data Outputs

None.

Adapters

- **AX_IN** (socket, `adapter::types::unidirectional::AX`): Raw input pulse
- **AX_Q** (plug, `adapter::types::unidirectional::AX`): Output stretched to at least `PT`

Program Flow and Connections

- `AX_IN` ⟶ `AX_SPLIT_2_In.IN`
- `AX_SPLIT_2_In.OUT1` ⟶ `AX_TP_MinZeit.IN`
- `AX_SPLIT_2_In.OUT2` ⟶ `AX_OR_Halten.IN2`
- `AX_TP_MinZeit.Q` ⟶ `AX_OR_Halten.IN1`
- `AX_OR_Halten.OUT` ⟶ `AX_Q`
- `PT` ⟶ `AX_TP_MinZeit.PT`; `R` ⟶ `AX_TP_MinZeit.R`

Technical Details

- No latch and no `AX_TON` needed: `AX_TP` is non-retriggerable, hence `AX_Q = AX_IN OR AX_TP_MinZeit.Q`
- Shorter pulses are stretched (`AX_TP.Q` holds until `PT`), longer ones remain unchanged (`AX_IN` keeps `Q` high even after `AX_TP.Q` has expired)
- Version 1.2 (2026-09-29): simplification compared to v1.1 (`ACTIVE`/`AX_TON`/`AX_R_TRIG` latch construction, discarded); new event input `R`

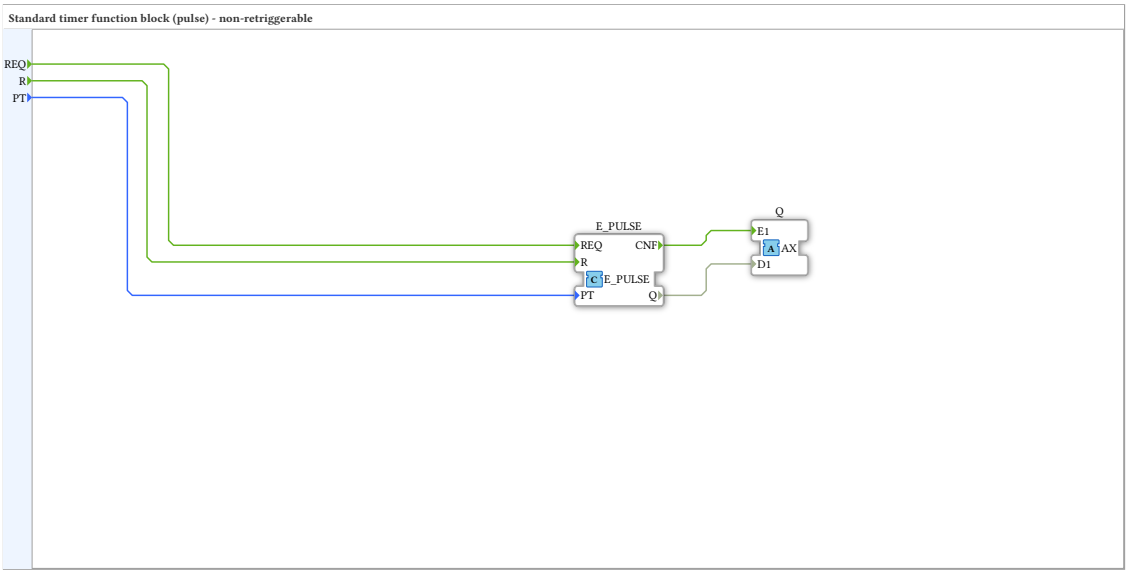
Application Scenarios

- Minimum run time for a drive (e.g. hopper floor lead-in in automatic mode)
- Stretching pulses that are too short

Summary

AX_MIN_TP outputs an AX pulse of at least PT length by ORing the input with a non-retriggerable AX_TP.

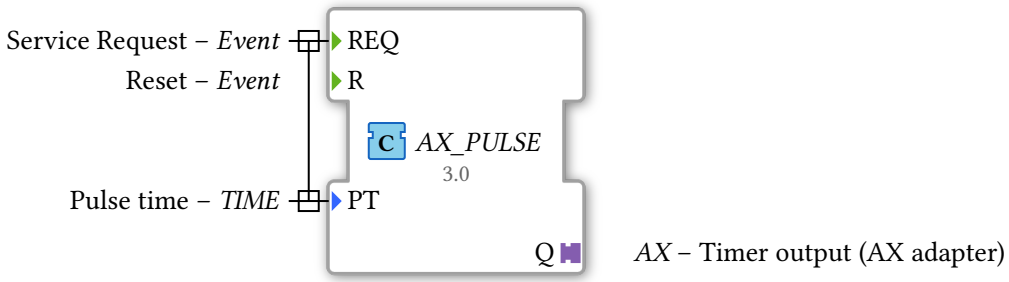
AX_PULSE



• – ◦ • – ◦ • – ◦ •

Introduction

Important Note: This function block requires only one event and no cyclic calls. It has no output ET and does not display the elapsed time. The AX_PULSE is a function block that outputs a pulse via an AX adapter.



Interface Structure

Event Inputs

- **REQ** (Event): Service request (with PT); triggers the pulse.
- **R** (Event): Reset.

Data Inputs

- **PT** (TIME): Pulse time.

Adapters

Plugs (Outputs):

- **Q** (adapter::types::unidirectional::AX): Timer output (AX adapter), delivers the pulse.

Functionality

On REQ, Q is activated for the duration PT (non-retriggerable); R resets the pulse. Internally an E_PULSE is used, whose CNF/Q are routed to the adapter Q.

Technical Features

- Uses unidirectional adapters.

State Overview

Pulse.

Application Scenarios

Signaling

⚖️ Comparison with Similar Components

- **E_PULSE**

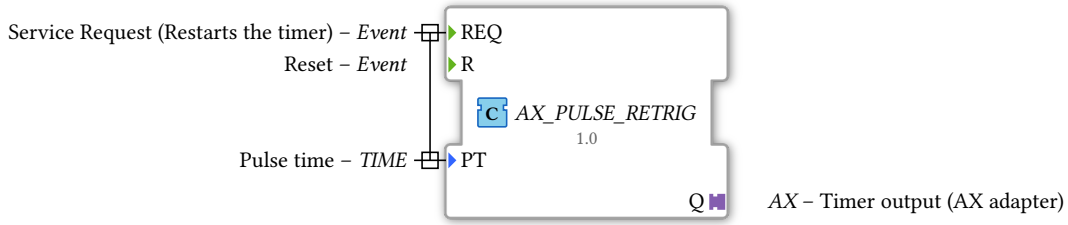
🛠️ Related Exercises

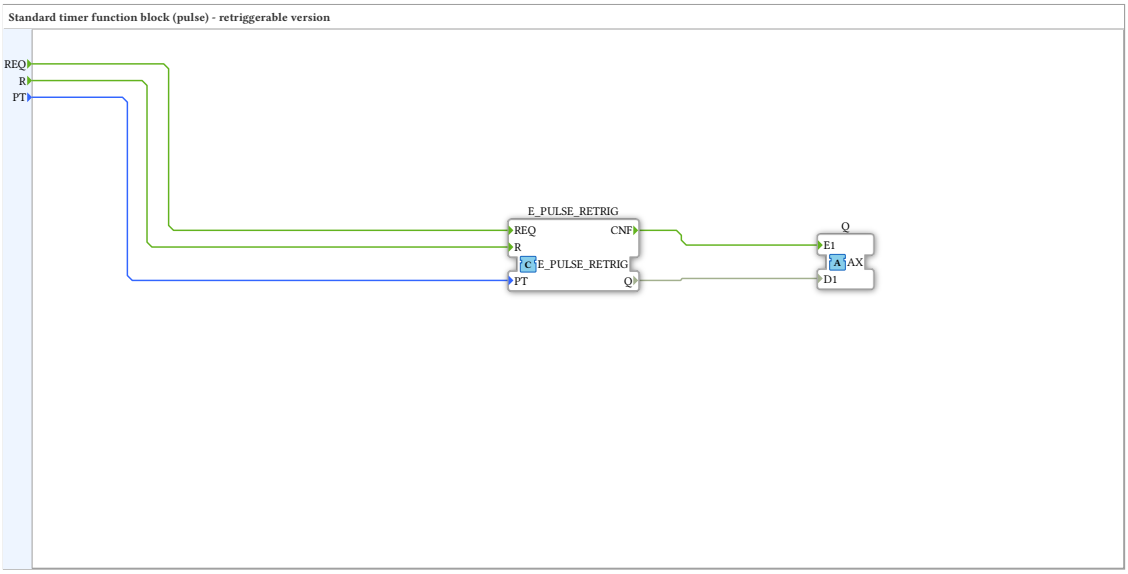
- Exercise_020h_AX
- Exercise_020i_AX

Conclusion

Adapter-Based Pulse Component

AX_PULSE_RETRIG





• – ○ • – ○ • – ○ •

Introduction

The **AX_PULSE_RETRIG** function block is a retriggerable pulse shaper with an AX adapter interface. It generates a pulse with a predefined duration (PT) at its output. A restart command within the pulse duration resets the timer and extends the pulse. This block is typically used to generate precisely timed, extendable output signals in automation systems.

Interface Structure

Event Inputs

Name	Type	Description
REQ	EVENT	Service request to start or restart the timer (connected to PT)
R	EVENT	Reset Timer (End Pulse Immediately)

Event Outputs

This function block does not have direct event outputs. Output signals are provided via the Q adapter.

Data Inputs

Name	Type	Description
PT	TIME	Pulse Duration (Pulse Time)

Data Outputs

This function block does not have direct data outputs. Output data is provided via the Q adapter.

Adapter

Name	Type	Description
Q	`adapter::types::unidirectional::AX`	Adapter interface that outputs the pulse status (event and data)

Functionality

- The function block internally encapsulates the standard function block `E_PULSE_RETRIG` (according to IEC 61499).
- A **REQ** event starts the timer: Output Q (via adapter) becomes active and the pulse begins.
- If another **REQ** event occurs during the pulse duration, the timer is reset and the pulse duration starts again (retrigger function).
- An **R** event immediately interrupts the pulse – output Q becomes inactive.
- Adapter Q transmits the current pulse state as an event (via `Q.E1`) and as a data value (via `Q.D1`).

Technical Features

- Uses the standardized `E_PULSE_RETRIG` function block as an internal implementation.
- Output is exclusively via an AX adapter, which provides both an event signal (upon state change) and the binary pulse state (type BOOL).
- A subsequent trigger restarts the timing – ideal for pushbuttons with a delay or pulse-width modulated signals.
- The pulse duration PT can be changed during propagation, but this change only takes effect upon the next REQ.

State Overview

This function block maps the states of the internal `E_PULSE_RETRIG`:

State	Description
IDLE	No pulse active; output Q = FALSE
PULSE	Pulse running; output Q = TRUE; timer running. A REQ event returns to PULSE with a new timer start.

- Transition IDLE → PULSE via REQ.
- Transition PULSE → IDLE by expiration of the PT time or by Reset R.

Application Scenarios

- **Extendable Switch-On Pulses**: If a sensor triggers multiple times within the pulse duration, the output should remain permanently active (e.g., for level monitoring).
- **Clock Generator with Retriggerable Pulse Width**: Control of valves or lamps that are switched on again for a fixed time upon repeated request.
- **Safety Functions**: Monitoring of machines where a renewed start command resets the downtime.

Comparison with Similar Function Blocks

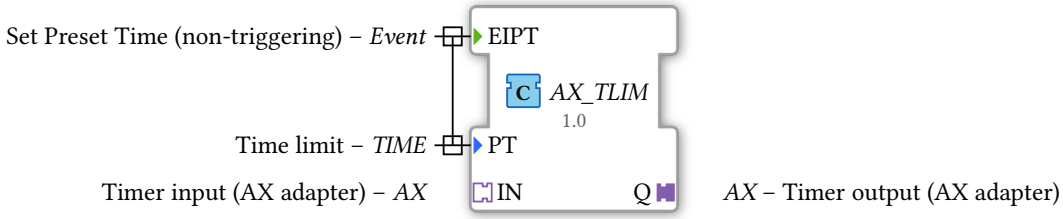
Function Block	Property
E_PULSE	Simple pulse without retrigger – one-time timer start, no restart upon subsequent request.
AX_PULSE_RETRIG	Retriggerable version with AX adapter (event + data).
TON / TOF	Delay blocks (turn-on delay / turn-off delay) – no pulse shaper.

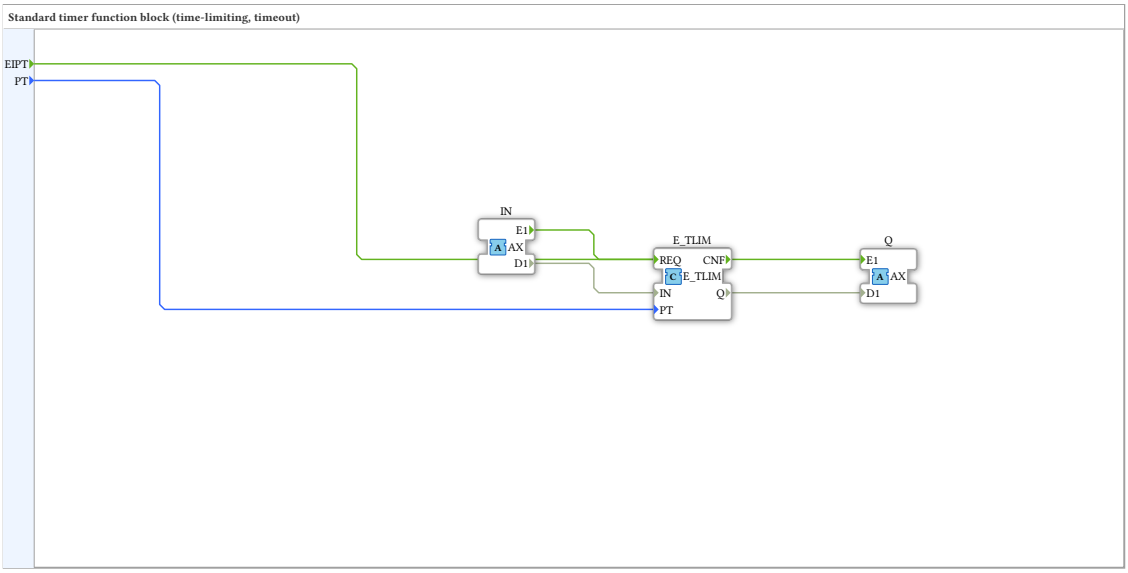
The AX_PULSE_RETRIG thus offers the flexibility of a retriggerable pulse with a standardized adapter interface for easy integration into larger function block networks.

Conclusion

The **AX_PULSE_RETRIG** is a compact, high-performance function block for generating retriggerable pulses. Thanks to the AX adapter interface, it can be seamlessly integrated with other function blocks and is particularly suitable for applications where an output signal needs to be extended for repeated requests. The use of the established `E_PULSE_RETRIG` function block ensures standards-compliant and reliable timing behavior.

AX_TLIM





• – ◦ • – ◦ • – ◦ •

Introduction

The **AX_TLIM** function block is a time-limiting timer according to IEC 61499-2. It monitors a Boolean input signal provided via a unidirectional **AX adapter**. As long as the signal is active (TRUE), the output remains active – but only for a predefined time. If the activation duration exceeds the limit, the output is reset (timeout). A separate event input allows the time limit to be preset without triggering a new execution.

Interface Structure

Event Inputs

Name	Type	Description
EIPT	Event	Resets the time limit PT (does not trigger execution)

Event Outputs

This function block does not have direct event outputs. Instead, the output event and value are provided via the **adapter Q** (see the Adapters section).

Data Inputs

Name	Type	Description
PT	TIME	Time limit (Preset Time) for the timer

Data Outputs

Direct data outputs are not available. The current timer status is output via the **adapter Q**.

Adapter

Name	Type	Direction	Description
IN	AX	Socket	Input adapter: receives (via event E1 and date D1) the Boolean value to be monitored
Q	AX	Plug	Output adapter: sends (via event E1 and date D1) the timer state (TRUE = active, FALSE = inactive or timeout)

Note: The adapter type AX is a unidirectional standard adapter consisting of an event (E1) and a Boolean date (D1).

Functionality

1. **Initial State** – The output variable Q.D1 is FALSE; the timer is not running.

Note: The adapter type AX is a standard unidirectional adapter consisting of an event (E1) and a Boolean date (D1).

1. **Activation** – As soon as the input adapter IN delivers an event (E1) with the date TRUE, Q.D1 is immediately set to TRUE, and an internal timer is started.
2. **Time Monitoring** – The timer counts the time that IN.D1 remains on TRUE. The timer uses the value last set via EIPT, PT.

- *Case A:* If IN.D1 becomes FALSE before PT expires, then Q.D1 is immediately set to FALSE.
- *Case B:* If IN.D1 remains on TRUE longer than PT, then after PT expires, Q.D1 is set to FALSE (timeout).

Case B: If IN.D1 remains on TRUE for longer than PT expires, then Q.D1 is set to FALSE (timeout). 4. **Event Output** – Every change to Q.D1 is signaled via the output adapter Q.E1.

1. **Default** – The event EIPT resets the time limit PT without affecting the current timer execution. Only the next activation of the timer (by IN.E1 with TRUE) uses the new value.

Technical Features

- The function block is implemented as a **composite FB** internally, it uses the standard FB `E_TLIM` (from `iec61499::events::timers`). The underlying logic is identical, but the interface is provided via unidirectional **AX adapters**.
- The use of an adapter socket (`IN`) and an adapter plug (`Q`) enables a clean separation of event and data communication and simplifies integration with other function blocks using compatible adapters.
- The timer is triggered **on an edge**: Only a rising edge on `IN.D1` starts a new timer cycle. A subsequent event on `IN.E1` with `TRUE` during a running timer does not reset the timer (unless the value briefly goes to `FALSE` and then back to `TRUE`).

State Overview

The function block cycles through the following logical states (not explicitly represented as a state machine in the XML):

State	Condition `IN.D1`	Output `Q.D1`	Timer running	Description
Inactive	`FALSE`	`FALSE`	no	Waiting for activation
Active	`TRUE`	`TRUE`	yes	Timer counting, output active
Timeout	`TRUE`	`FALSE`	no	Timeout, timer expired, output inactive
Return	`FALSE`	`FALSE`	no	Input reverts before timeout, output immediately inactive

A switch from **Active** to **Timeout** occurs when the remaining timer time reaches zero. A switch from **Active** to **Revert** occurs when `IN.D1` changes to `FALSE` before the timer expires.

Application Scenarios

- **Actuator Monitoring** – e.g., checking whether a valve remains open longer than permitted.
- **Safety Functions** – Switching off a signal after a maximum activation time.
- **Time-Based Plausibility Check** – Detection of a stuck sensor that continuously reports `TRUE`.
- **Control of blinking or pulse sequences** – the timer can be used as a simple monostable multivibrator with a timeout.

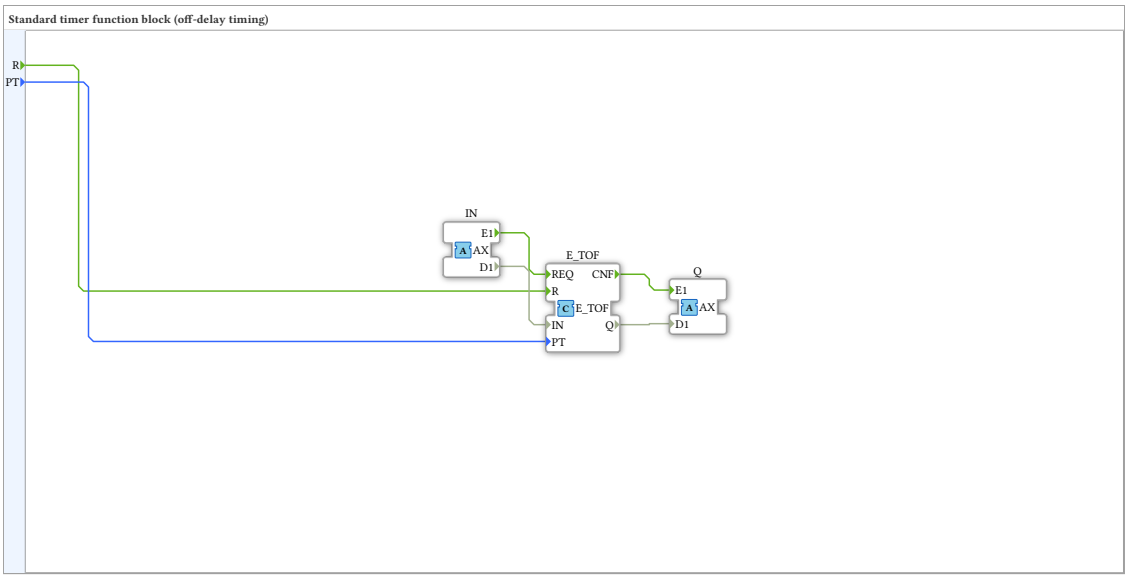
Comparison with similar function blocks

Function block	Description	Difference to AX_TLIM
`E_TLIM`	Standard timer (IEC 61499-2), direct event and data interfaces	AX_TLIM encapsulates `E_TLIM` and provides an adapter-based interface (`AX`) that is easier to integrate into certain architectures.
`E_DELAY`	Delay function block (response delay, fallback delay)	`E_DELAY`Delays signal changes, while AX_TLIM enforces a maximum activation duration.
`E_SR`	Set-Reset Flip-Flop	AX_TLIM does not reset the output as long as the input is active (except via timeout). `E_SR` responds to separate set/reset events.

Conclusion

The **AX_TLIM** function block offers a compact, adapter-based solution for time-limited monitoring of Boolean signals. By encapsulating the standard `E_TLIM` and using unidirectional AX adapters, it can be flexibly integrated into IEC 61499 systems, especially when communication paths need to be encapsulated via adapters. Its operation is intuitive: As long as the input is active, the output remains active – for a maximum of the configurable time `PT`. This makes the component ideal for safety and monitoring tasks in automation technology.

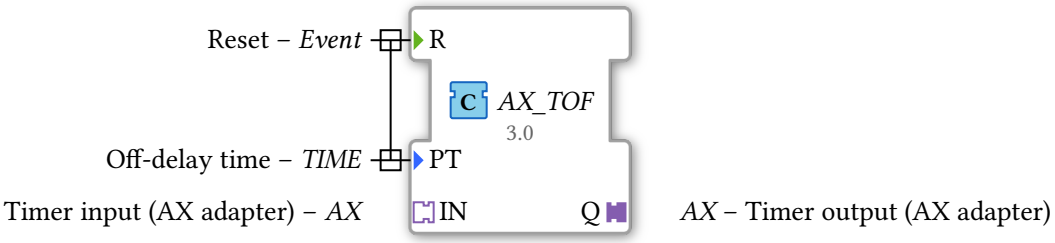
AX_TOF



• – ○ • – ○ • – ○ •

Introduction

Important note: This function block requires only one event and no cyclic calls. It has no output ET and does not display the elapsed time. The AX_TOF function block is a standard timer function block for off-delay timing. It implements a time delay when a signal is switched off and is controlled via adapter interfaces.



Interface Structure

Event Inputs

- **R**: Reset event to reset the timer

Event Outputs

No direct event outputs available

Data Inputs

- **PT**: TIME - Process time for the delay

Data Outputs

No direct data outputs available

Adapters

- **IN** (Socket): Input adapter of type `adapter::types::unidirectional::AX`
- Event: E1
- Data: D1
- **Q** (Plug): Output adapter of type `adapter::types::unidirectional::AX`
- Event: E1
- Data: D1

Functionality

The AX_TOF block implements an off-delay. When an incoming signal is received via the IN adapter, the output is activated immediately. When the input signal is switched off, the timer starts and keeps the output active for the set time PT before it is deactivated. ...

Technical Features

- Internally uses the E_TOF standard timer block
- Unidirectional adapter interfaces for input and output
- Reset functionality for immediate timer reset
- Time base in TIME data type

State Overview

1. **Idle State**: No active signal
2. **Enabled**: Input signal present, output active
3. **Delay**: Input signal switched off, timer running, output remains active
4. **Expired**: Timer expired, output disabled

Application Scenarios

- Control of switch-off delays in industrial processes
- Safety functions with overrun times
- Delayed shutdown of motors or actuators
- Process control with defined shutdown sequences

⚖️ Comparison with Similar Blocks

Compared to other timer blocks, AX_TOF offers:

- Specialized in switch-off delays
- Adapter-based interface for modular design
- Integrated reset functionality
- Standardized TIME Data Types

Comparison with E_TOF

🛠️ Related Exercises

- Exercise_020e_AX

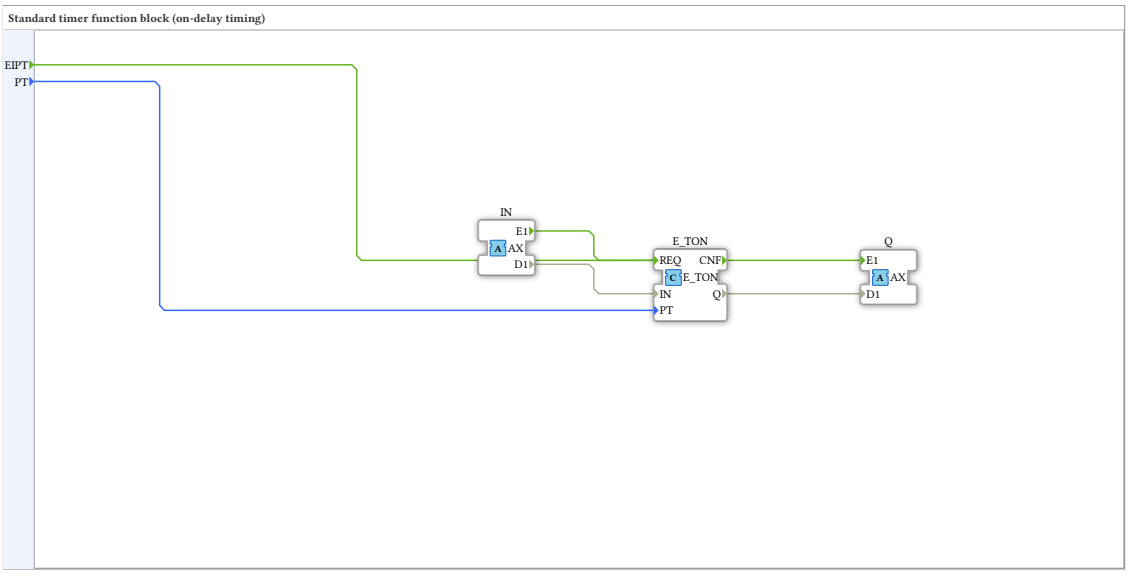
Conclusion

The AX_TOF function block offers a reliable and standardized solution for turn-off delays in 4diac-based control systems. Through the use of adapter interfaces, it enables flexible integration into more complex function block networks.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

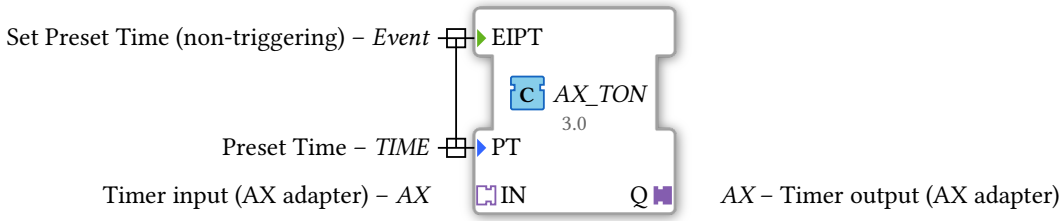
AX_TON



• – ◦ • – ◦ • – ◦ •

Introduction

Important note: This function block requires only one event and no cyclic calls. It has no output ET and does not display the elapsed time. AX_TON is a standardized timer function block that implements on-delay timing. After activation, the block waits a defined time before setting the output signal.



Interface Structure

Event Inputs

- **EIPT** (Event): Set Preset Time (non-triggering); carries PT

Event Outputs

- No direct event outputs available

Data Inputs

- **PT** (TIME): Process time - defines the delay duration

Data Outputs

- No direct data outputs available

Adapters

- **IN** (Input Adapter): Receives input events and data
- **Q** (Output Adapter): Sends output events and data

Functionality

The AX_TON block functions as a power-on delay timer. When an input event is received via the IN adapter, the internal timer starts. After the set process time (PT) has elapsed, an output event with the

status TRUE is sent via the Q adapter. If the input is deactivated during the time monitoring period, the timer is reset.

Technical Features

- Implemented based on the E_TON standard timer block
- Uses unidirectional adapters for inputs and outputs
- Supports TIME data type for precise timing control
- Compatible with IEC 61499-2 standard

State Overview

1. **Idle State**: No active timer, output FALSE
2. **Activation State**: Timer runs after input signal
3. **Delay State**: Timer counts down, output remains FALSE
4. **Trigger State**: After time elapses, output becomes TRUE

Application Scenarios

- Control of time-delayed process starts
- Monitoring of minimum runtimes
- Delayed activation of machine components
- Safety-relevant time monitoring

⚖️ Comparison with Similar Blocks

Compared to other timer blocks, AX_TON offers a standardized interface via adapters, which facilitates reusability and modular integration into larger control systems.

Comparison with E_TON

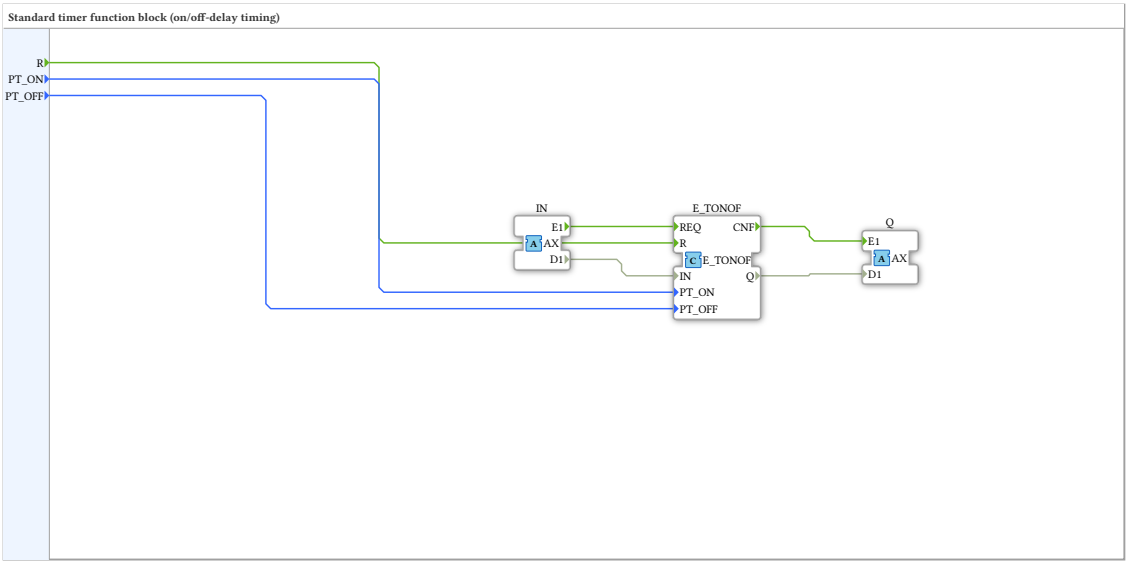
🛠️ Related Exercises

- Exercise_020c_AX
- Exercise_160b2_AX

Conclusion

AX_TON is a robust and reliable timer module for power-on delays, which, thanks to its standardized adapter-based interface, enables flexible integration into various automation systems.

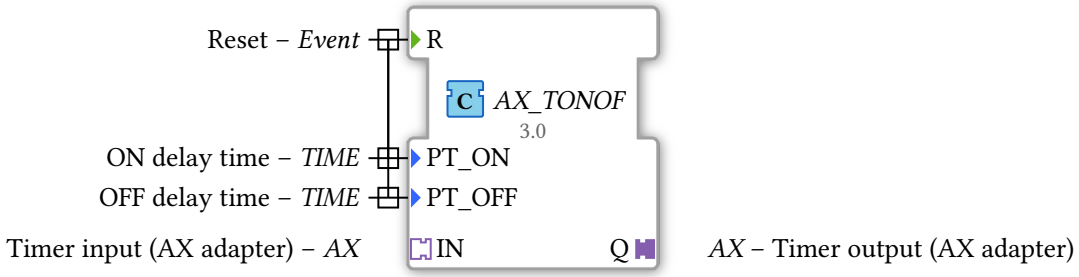
AX_TONOF



• – ○ • – ○ • – ○ •

Introduction

Important note: This function block requires only one event and no cyclic calls. It has no output ET and does not display the elapsed time. AX_TONOF is a standardized timer function block that implements both on and off delays. This function block combines the functionalities of an on-delay and an off-delay timer in a compact solution.



Interface Structure

Event Inputs

- **R**: Reset event to reset the timer

Event Outputs

No direct event outputs available

Data Inputs

- **PT_ON**: TIME - Process time for the power-on delay
- **PT_OFF**: TIME - Process time for the power-off delay

Data Outputs

No direct data outputs available

Adapters

- **IN**: Input adapter (socket) for control signals
- **Q**: Output adapter (plug) for status outputs

Functionality

The AX_TONOF function block processes input signals via the IN adapter and applies the configured delay times depending on the signal state. When an input signal is active, the PT_ON delay is taken into account; when the signal is inactive, the PT_OFF delay is used. The result is output via the Q-adapter.
...))))) ``) ``) ``)) is taken into account. The `PT_ON` delay is taken into account when an

Technical Features

- Supports bidirectional time delays (on and off delay)
- Integrated reset functionality
- Uses adapters for inputs and outputs instead of direct interfaces
- Based on the E_TONOF core function block

State Overview

The function block can assume the following states:

- **Idle State**: No active time measurement
- **On Delay**: Time measurement for PT_ON active
- **Off Delay**: Time measurement for PT_OFF active
- **Reset State**: All active time measurements are canceled

Application Scenarios

- Motor controls with start and stop delays
- Lighting control with on/off delays
- Process controls with staggered switching operations

- Safety-related applications with delayed responses

⚖️ Comparison with Similar Function Blocks

Compared to simple timer function blocks like TON or TOF, AX_TONOF offers the combined functionality of both timer types in a single block. The use of adapters instead of direct inputs/outputs allows for more flexible integration into complex control structures.

Comparison with E_TONOF

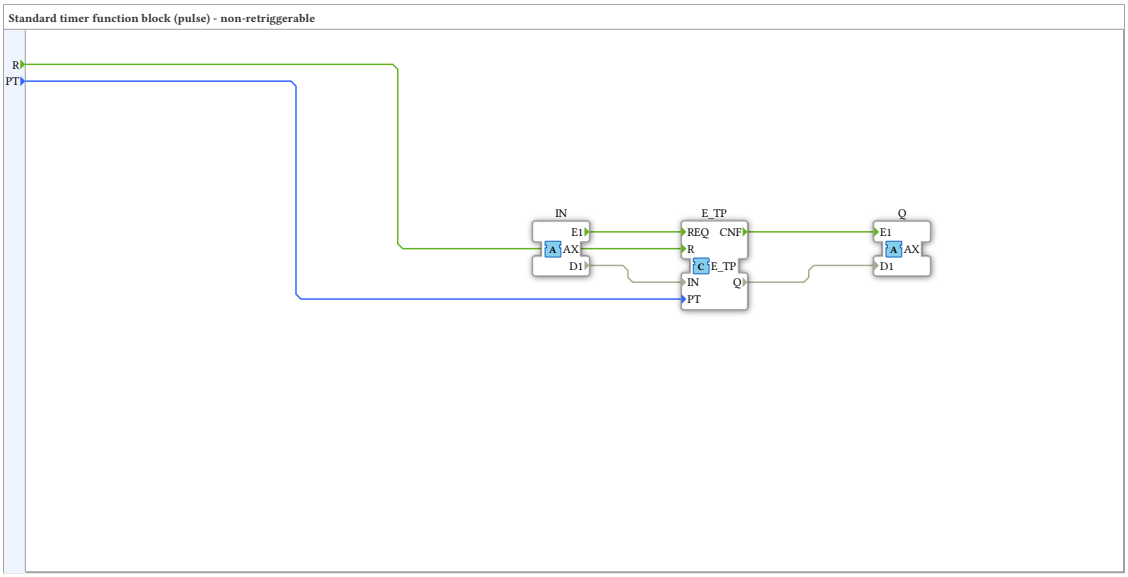
🛠️ Related Exercises

- Exercise_020g_AX

Conclusion

The AX_TONOF is a versatile timer module that, thanks to its combined on/off delay functionality and adapter-based interface design, is particularly well-suited for modular control architectures. The integrated reset function increases flexibility in practical applications.

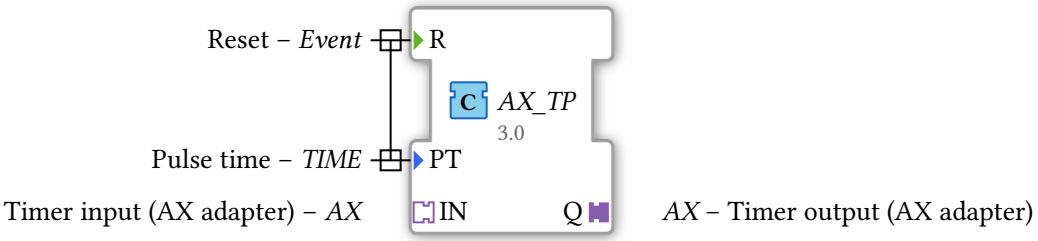
AX_TP



• – ○ • – ○ • – ○ •

Introduction

Important Note: This function block requires only one event and no cyclic calls. It has no output ET and does not display the elapsed time. The AX_TP is a standardized timer function block of type "Pulse" used in 4diac IDE applications. This function block generates a timed pulse output based on a configurable time interval.



Interface Structure

Event Inputs

- **R**: Reset Event - Resets the timer

Event Outputs
- No direct event outputs available

Data Inputs
- **PT**: TIME - Pulse Time (Pulse Duration)

Data Outputs
- No direct data outputs available

Adapters
- **IN**: Input Adapter (Socket) - Receives input events and data
- **Q**: Output Adapter (Plug) - Sends output events and data

Functionality
The AX_TP function block operates as a pulse timer. When an event is received via the IN adapter, a timed pulse is generated, the duration of which is defined by the PT input. The reset input (R) allows the timer to be reset prematurely. The internal E_TP function block handles the actual timer functionality.

Technical Features
- Uses the standardized E_TP timer block as its core component
- Implemented via adapter interfaces for modular connection
- Supports the TIME data type for pulse time configuration
- Provides separate reset functionality

State Overview
The function block has the following operating states:

- **Inactive**: No active timer
- **Active**: Timer is running with configured pulse time
- **Reset**: Timer has been reset

Application Scenarios
- Time-controlled sequences
- Pulse generation for actuators
- Delay circuits
- Process control with defined time intervals

⚖️ Comparison with Similar Blocks
Compared to other timer blocks, AX_TP offers:

- Adapter-based interface for improved modularity
- Explicit reset functionality
- Standardized pulse timer implementation

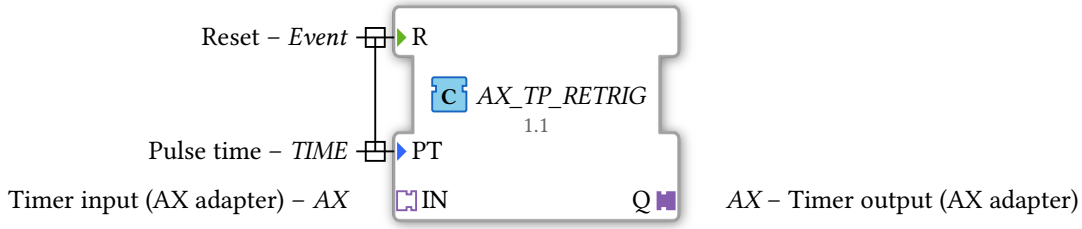
Comparison with E_TP

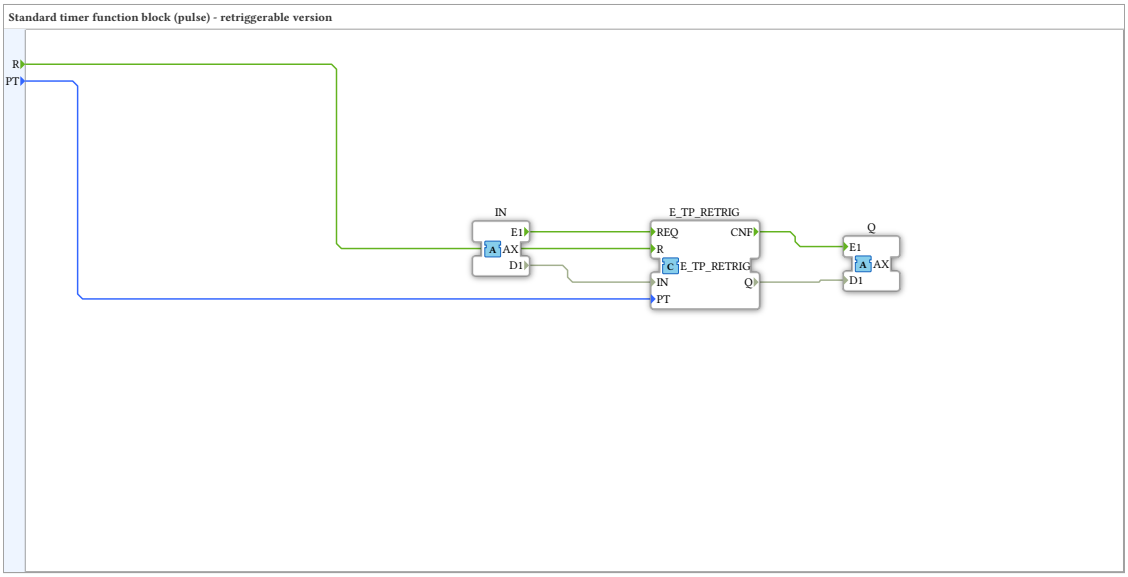
🛠️ Related Exercises
- Exercise_020f_AX
- Exercise_020j2_AX_sub
- Exercise_020j_AX
- Exercise_177_AX
- Exercise_178_AX

Conclusion
The AX_TP is a robust and standards-compliant timer function block, ideally suited for time-controlled applications in distributed automation systems. Its adapter-based architecture enables flexible integration into more complex control structures.

🌐 Related topic subpages on ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AX_TP_RETRIG





• – ○ • – ○ • – ○ •

Introduction

The **AX_TP_RETRIG** function block is a standardized timer function block (pulse) in a retriggerable configuration. It generates a pulse of duration **PT** on the adapter output **Q** as soon as the input signal **IN** receives a rising edge (event REQ). The pulse is restarted (retriggered) with each new REQ event. A reset via the event input **R** advances the pulse duration **PT** and resets the timer. The function block is designed for use in IEC 61499-based automation systems and utilizes the AX adapter interface for efficient, unidirectional signal transmission.

Interface Structure

Event Inputs

Event	Description
R	Reset – Sets the pulse time **PT** and immediately terminates a running pulse.

Event Outputs

No dedicated event outputs – output is provided via the **Q** adapter (event **E1**).

Data Inputs

Name	Type	Description
PT	TIME	Pulse duration (pulse time).

Data Outputs

No dedicated data outputs – pulse status is output via the **Q** adapter (data value **D1**).

Adapters

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::AX`	Socket	Input – Receives the start signal (event **E1**) and the logical value (**D1**). A rising edge on **E1** triggers the pulse.
Q	`adapter::types::unidirectional::AX`	Plug	Output – Outputs the pulse: As long as the pulse is active, **D1** = `TRUE`, otherwise it is FALSE. The event **E1** is sent with every state change.

Functionality

1. **Initial State** – **Q.D1** is `FALSE`, no pulse is active.
2. **Start** – Upon an event **IN.E1** (a rising edge interpreted as REQ), the function block initiates a pulse of duration **PT**. **Q.D1** is immediately set to `TRUE`, and the event **Q.E1** is sent.
3. **Retrigger** – While the pulse is still active, another **IN.E1** resets the timer, and the pulse restarts for the full duration **PT**. **Q.D1** remains `TRUE` for the entire duration of the pulse.
4. **Pulse End** – After **PT** has elapsed (without a new trigger), **Q.D1** is reset to `FALSE`, and **Q.E1** is sent again.
5. **Reset** – The **R** event immediately terminates a running pulse and sets **Q.D1** to `FALSE`. Simultaneously, **PT** is prepended with the current value at input **PT** (this can be used for a new pulse duration).

Technical Features

- **Retriggerability** – Unlike simple timer blocks, a running pulse is not interrupted by a new start pulse, but rather extended (timer restart).
- **Adapter Interface** – Instead of individual event/data ports, the AX adapter interface is used, which enables a standardized, unidirectional connection. This simplifies integration in adapter-based architectures.
- **License** – The block is released under the **Eclipse Public License 2.0** (see copyright information).
- **Implementation** – Internally, the IEC 61499 standard function block **E_TP_RETRIG** is used, which is integrated via the network.

State Overview

The function block goes through the following states (based on the internal timer function block):

State	Description
IDLE	No pulse active; **Q.D1** = `FALSE`.
RUNNING	Pulse active; **Q.D1** = `TRUE`. The timer runs for the duration **PT**. Upon retriggering, the timer is reset (remains in the RUNNING state).
TIMEOUT	Pulse terminated; transition back to IDLE.

Note: The reset (event **R**) always immediately returns to the IDLE state.

Application Scenarios

- **Control of output signals with fixed pulse duration** – e.g., controlling a relay that is switched for a defined time.
- **Clock generator with retriggering** – If a new start signal arrives during a pulse, the pulse duration should be extended (e.g., for a manual push button that is pressed for a longer time).
- **Safety interlocks** – An alarm or action should remain active for a minimum duration, even if the triggering state disappears briefly.
- **Time-controlled enables** – e.g., a machine movement that is executed for the duration **PT** after a start signal, but restarts upon a new start pulse.

Comparison with similar function blocks

Function block	Property

AX_TP_RETRIG	Retriggerable pulse timer with AX adapter.
E_TP (Standard IEC 61499)	Simple pulse timer, non-retriggerable – restarting during the pulse is ignored.
E_TP_RETRIG	Retriggerable version without adapter (pure event/data interface).
E_DELAY (Delay)	Delays a signal by a fixed time, but does not generate a pulse of fixed duration.

The **AX_TP_RETRIG** combines retrigger functionality with a compact adapter interface and is therefore particularly suitable for modular, adapter-based systems.

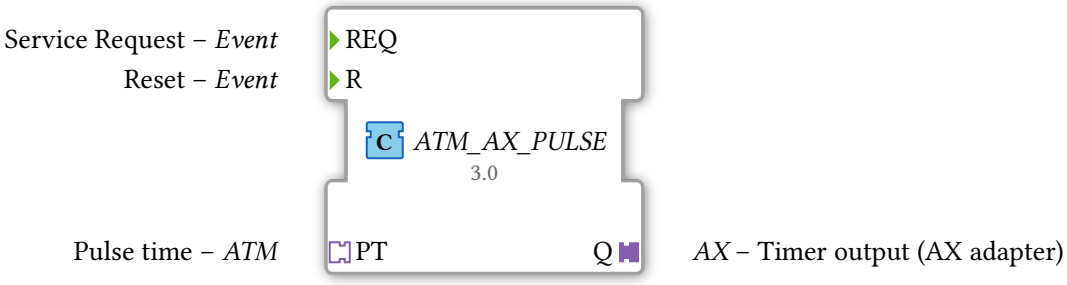
Conclusion

The **AX_TP_RETRIG** is a flexible and standardized timer function block for IEC 61499 applications. Its retrigger capability makes it ideal for situations where a pulse needs to be extended during repeated start commands. The use of the AX adapter interface simplifies integration with other adapter-based components and promotes a clean, modular system architecture. Thanks to the clear separation of event and data flows and adherence to the standard, the module is suitable for reliable use in industrial control systems.

Numeric Index (timers)

Here you will find all exercises from the `timers` workspace, listed in ascending order. …

ATM_AX_PULSE



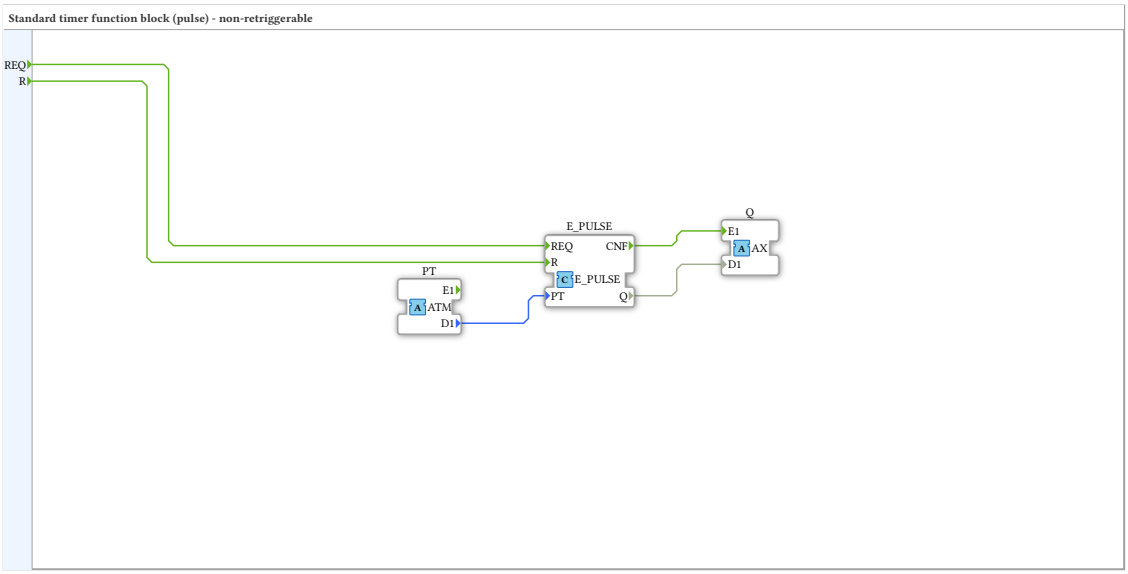


Image not available

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_AX_PULSE** is a standardized timer module for generating a non-retriggerable pulse. It is configured and read via the adapter interfaces **PT** (ATM) and **Q** (AX). The module complies with the IEC 61499-2 standard and is part of the library **adapter::events::unidirectional::timers**.

Interface Structure

Event Inputs

Name	Type	Description
REQ	Event	Service request – starts the pulse.
R	Event	Reset – terminates the pulse prematurely.

Event Outputs

No direct event outputs. Output is provided via the **Q** adapter (see below).

Data Inputs

No direct data inputs. The pulse duration is provided via the **PT** adapter.

Data Outputs

No direct data outputs. The pulse status is queried via the **Q** adapter.

Adapters

Adapter	Direction	Type	Description
PT	Socket	`adapter::types::unidirectional::ATM`	Returns the pulse duration as a time value.
Q	Plug	`adapter::types::unidirectional::AX`	Outputs the pulse status: **D1** = current value (TRUE during the pulse); **E1** = event at the end or change of the pulse.

Functionality

This function block implements the behavior of a non-retriggerable pulse generator (E_PULSE). Upon arrival of the **REQ** event, a pulse is initiated, the duration of which is determined by the value applied to the **PT** adapter. During the pulse, the output **Q.D1** = *TRUE*. After the pulse duration has elapsed, **Q.D1** becomes *FALSE* and the **Q.E1** event is triggered. A subsequent **REQ** during an ongoing pulse is ignored (non-retriggerable). The pulse can be prematurely terminated at any time via the **R** reset input; **Q.E1** is also triggered in this case.

Technical Features

- **Adapter-based communication:** Instead of direct data/event ports, typed adapters (ATM for time, AX for output) are used. This simplifies reuse in modular systems.
- **Non-retriggerable:** A second start pulse during an ongoing pulse has no effect.
- **Validated versions:** The function block has been revised several times (versions 1.0 to 3.0, most recently in 2025) and includes a reset function.
- **Licensed under EPL 2.0** (Eclipse Public License 2.0).

State overview

1. **Idle** – No pulse active. **Q.D1** = FALSE.
2. **PulseActive** – Pulse running. **Q.D1** = TRUE. Entered by **REQ** and exited either by timeout or **R**.

- At time elapse: Transition to Idle, triggering **Q.E1**.
- At **R**: Transition to Idle, triggering **Q.E1**.

1. During **PulseActive**, **REQ** is ignored.

Application Scenarios

- Control of actuators (e.g., valves, lamps) with a defined on-time.
- Timers in production machines that require one-time pulses.
- Signal conditioning in sensor-actuator networks (AX/ATM-compatible peripherals).

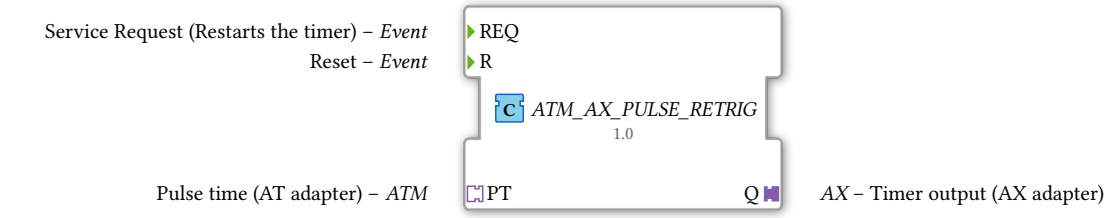
Comparison with Similar Function Blocks

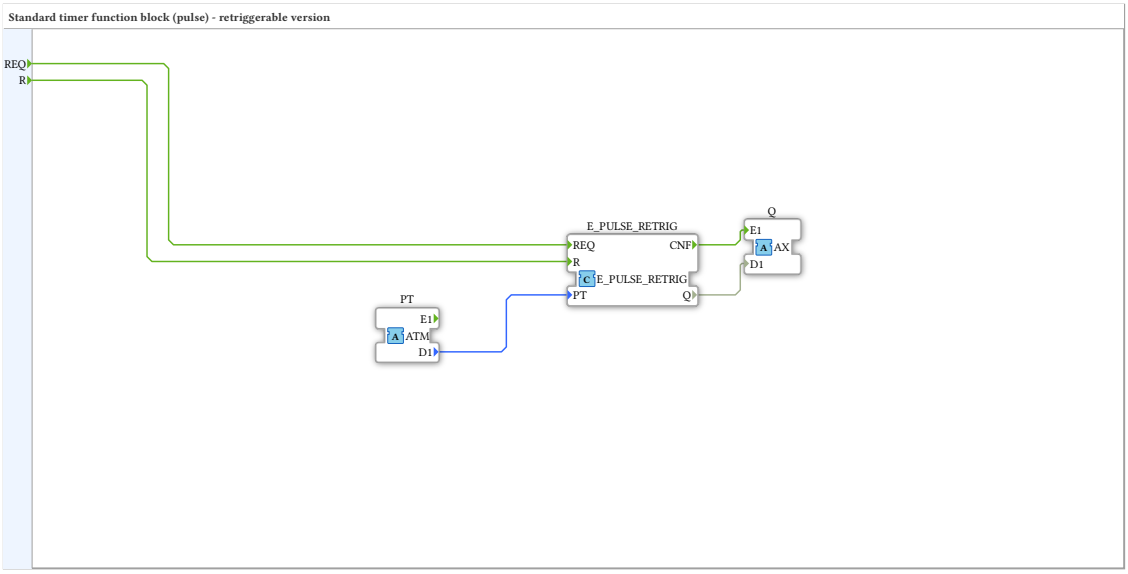
- **E_PULSE (iec61499::events::timers):** Standard implementation without an adapter interface. ATM_AX_PULSE encapsulates this function block and adds the adapter connection.
- **ATM_AX_PULSE_R (retriggerable):** Retriggerable variant; a subsequent **REQ** extends the pulse. The present block is not retriggerable.
- **E_DELAY, E_CYCLE:** Different timing behavior (delay, periodic) – not comparable.

Conclusion

The **ATM_AX_PULSE** is a robust, non-retriggerable pulse block that, thanks to its adapter interface, can be particularly well integrated into modular automation solutions using the AX/ATM protocol. It offers clear, reliable functionality for generating time-defined pulses and meets the requirements of IEC 61499-2.

ATM_AX_PULSE_RETRIG





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_AX_PULSE_RETRIG** generates a pulse of defined duration at its output. It is designed as a retriggerable version, meaning that a new start command during a running pulse resets the pulse duration. Control and parameter transfer are handled exclusively via two standardized adapter interfaces (ATM / AX), so the block does not require direct data inputs/outputs.

Interface Structure

Event Inputs

Name	Type	Comment

REQ	Event	Start command – triggers a new pulse and restarts the timer
R	Event	Reset – immediately terminates a running pulse

Event Outputs
None.

Data Inputs
None. The pulse duration is provided via the **PT** adapter (Socket).

Data Outputs
None. The pulse status (active/inactive) is output via the **Q** adapter (Plug).

Adapters

Role	Name	Type	Direction	Description	
Input	PT	ATM	Socket	Provides the desired pulse duration (D1 → PT)	
Output	Q	AX	Plug	Displays the pulse state (D1 = HIGH during the pulse)	

Functionality
When the function block is triggered by a **REQ** event, it starts a timer with the duration specified by the **PT** adapter (data value D1). The **Q** output (D1) is immediately set to **HIGH** and remains active for exactly this duration. If another **REQ** event arrives during an ongoing pulse, the timer is reset and the pulse duration starts again (retriggerable). A **Reset** event (**R**) terminates the pulse prematurely and sets **Q** to **LOW**.

The function block implements the internal logic using the standardized IEC 61499 function block **E_PULSE_RETRIG**, which handles all timing functions. The adapters (PT → D1 to internal timer, internal output Q → Q.D1) are connected via a hidden network.

Technical Features
- **Retriggerable Behavior**: Each new **REQ** during an active pulse extends the pulse duration to the full value of PT.
- **Adapter-Based Interface**: The module has no direct event outputs or data ports; all communication takes place via the standardized **ATM** (parameter) and **AX** (output) adapters.
- **Reusability**: Due to the strict separation of control and parameterization, the module is particularly suitable for modular and reusable applications.

State Overview
The module has two states:

- **Inactive** (Q.D1 = FALSE): No pulse is active, timer is not running.
- **Active** (Q.D1 = TRUE): Pulse is running; upon entry by a REQ, the timer is started with PT. Each subsequent REQ resets the timer. An R event immediately returns the device to the inactive state.

Application Scenarios
- **Clock Generator for Signal Lights**: A continuous start signal (e.g., push button or sensor) generates a defined light pulse. The retrigger function keeps the light on continuously with a constant signal.
- **Pulse Extension in Controllers**: Short input pulses are extended to a minimum or defined duration.
- **Control of Actuators with Minimum On-Time**: Valves, motors, or relays requiring a specific on-time can be reliably controlled.

Comparison with Similar Components

Component	Retriggerable	Interface	Application Area
ATM_AX_PULSE_RETRIG	Yes	Adapter (ATM/AX)	Modular Controllers with Adapter Concept

ATM_AX_PULSE	No	Adapter (ATM/AX)	Single pulses, no retriggering
Standard E_PULSE_RETRIG	Yes	Direct inputs/outputs	More flexible, but without adapter encapsulation

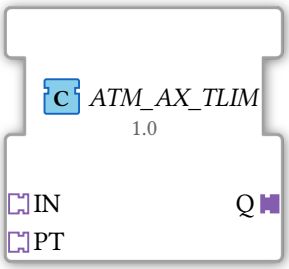
This module stands out due to its exclusive use of adapter interfaces, which simplifies integration into existing adapter-based libraries.

Conclusion

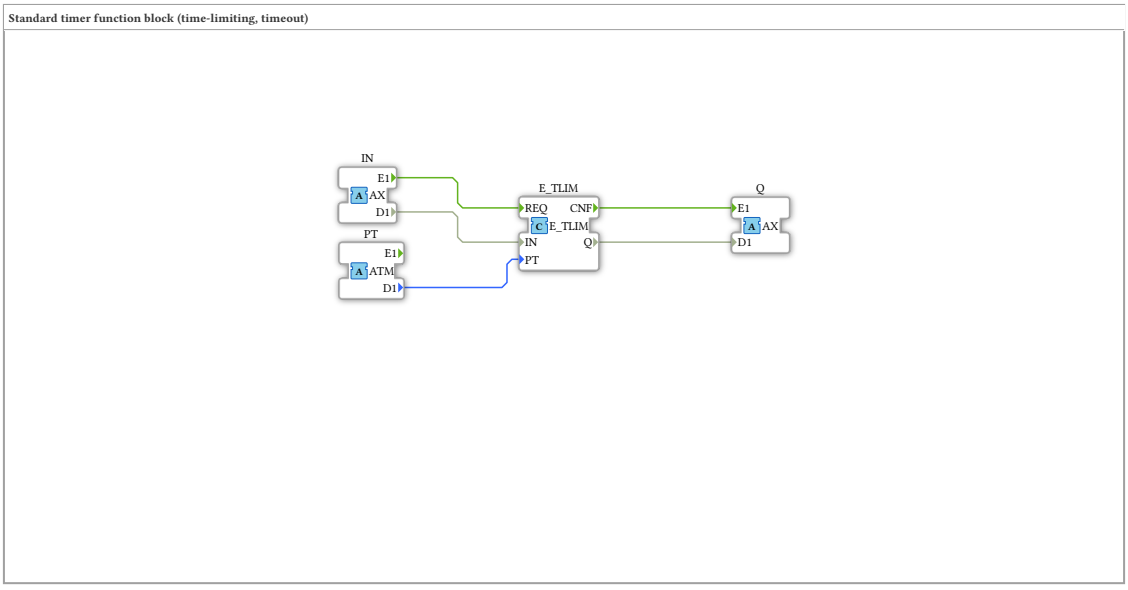
ATM_AX_PULSE_RETRIG is a compact, retriggerable pulse module characterized by its clean encapsulation using an ATM/AX adapter. It is ideally suited for use in distributed control systems based on the IEC 61499 adapter concept. Its ease of use (two events, two adapters) and proven timer behavior make it a reliable component for time-controlled outputs.

ATM_AX_TLIM

Timer input (AX adapter) – *AX*
Time limit (AT adapter) – *ATM*



AX – Timer output (AX adapter)



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_AX_TLIM** is a standardized time monitoring module (timer) with a special adapter interface. It implements time limiting (timeout) for Boolean input signals. The module communicates via adapters instead of individual event and data ports, enabling flexible and encapsulated integration into industrial control systems.

Interface Structure

Event Inputs

Name	Type	Comment
EIPT	Event	Set Preset Time (non-triggering)

Event Outputs

The module does not have separate event outputs. Event forwarding occurs exclusively via the adapter output **Q**, which triggers an event on its `E1` port after a change in the output signal.

Data Inputs

There are no direct data inputs. The required data (input signal IN and time limit PT) is provided via the corresponding adapter ports.

Data Outputs

Similarly, there are no direct data outputs. The result (Boolean value Q) is output via the adapter output **Q** (port `D1`).

Adapters

Name	Type	Direction	Comment
IN	AX	Input	Boolean Input (Adapter)
PT	ATM	Input	Time Limit (Adapter)
Q	AX	Output	Boolean Output (Adapter)

Functionality

The function block internally uses the standard function block **E_TLIM** and wraps it in an adapter. The logic is as follows:

1. **Activation**: An event at the adapter input **IN** (port `E1`) triggers processing. Simultaneously, the Boolean value from **IN.D1** is taken.
2. **Time Measurement**: When the input signal `IN` becomes **TRUE**, the function block sets the output `Q` to **TRUE** and starts a timer with a duration of **PT** (from adapter **PT**, port `D1`).
3. **Timeout**: If `IN` remains active longer than `PT`, `Q` is reset to **FALSE** (timeout).
4. **Premature Termination**: If `IN` becomes **FALSE** before the time expires, the function block `Q` is immediately reset (also to **FALSE**).
5. **Setting the Time Limit**: The event input **EIPT** allows you to specify the value of **PT** without triggering a new event. This is used to configure the time limit before the actual use.
6. **Premature Termination**: The output adapter **Q** outputs an event on its `E1` port as soon as `Q` changes (rising or falling edge).

Technical Features

- **Adapter-Based Communication**: Instead of individual event and data ports, AX/ATM adapters are used. These enable modular and reusable coupling with other components.
- **Internal Standard Component**: The implementation uses the established **E_TLIM** (IEC 61499), ensuring reliable and tested timing logic.
- **Non-Triggering Time Setting**: The **EIPT** input only changes the time limit without starting or resetting the timer. This is useful for dynamically adjusting parameters.
- **Typical Time Measurement**: Suitable for monitoring applications where a signal must not be active for a specific duration.

State Overview

The function block (FB) implicitly cycles through the following states:

State	Description
Idle	IN = FALSE, Q = FALSE, Timer is not running.
Timing	IN = TRUE, Q = TRUE, Timer is running.
Timeout	IN = TRUE, Timer expires, Q = FALSE (remains until IN = FALSE).

A switch back to **Idle** occurs as soon as IN becomes FALSE. The **Timeout** state is only reached when the timeout has occurred.

Application Scenarios

- **Monitoring of signals** in automation technology (e.g., maximum on-time of an actuator).
- **Safety functions** where an output must be deactivated if a signal persists for too long.
- **Timed resets** in communication protocols (watchdog functionality).
- **Simple time limiting** in complex control systems based on adapter interfaces.

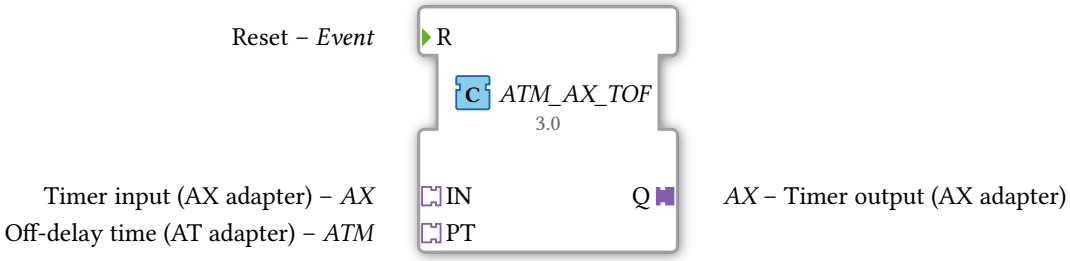
Comparison with similar function blocks

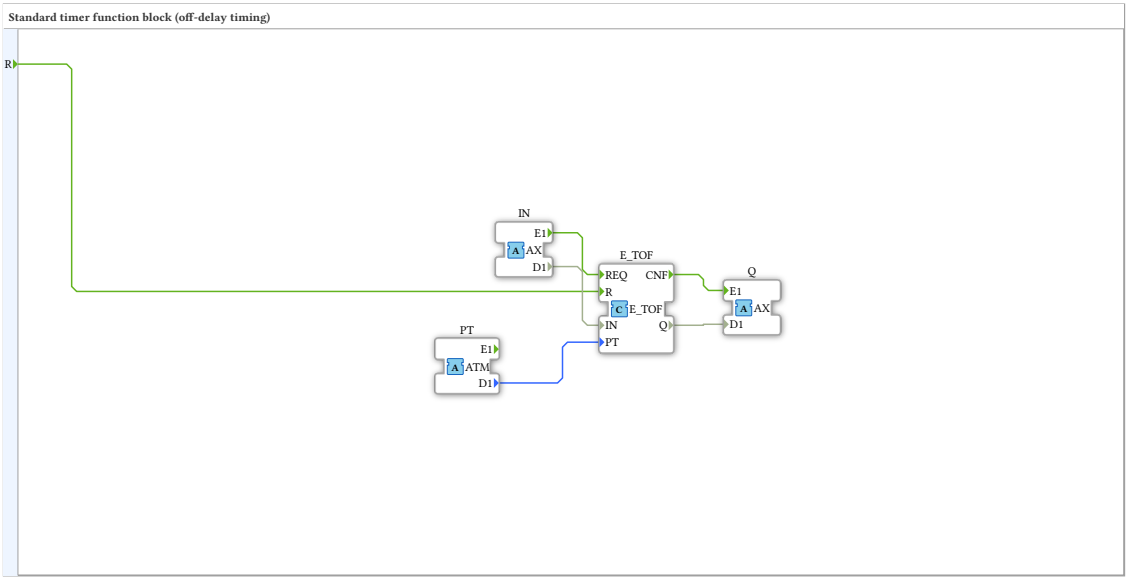
- **E_TLIM (Standard)**: Offers the same core logic, but with classic input/output ports. **ATM_AX_TLIM** encapsulates this logic in an adapter interface and is therefore better suited for modular and reusable components.
- **E_TON / E_TOF**: These function blocks implement turn-on and turn-off delays, respectively. In contrast, **ATM_AX_TLIM** monitors the maximum duration of an active signal and responds with a timeout.
- **E_CYCLE**: A cyclic timer that periodically generates signals has a different objective than simple monitoring.

Conclusion

The **ATM_AX_TLIM** function block offers a compact, adapter-based solution for time monitoring tasks. It combines the proven functionality of the standard timer **E_TLIM** with the flexibility of adapter interfaces and allows dynamic specification of the time limit via **EIPT**. This makes it particularly suitable for modern, modular automation projects based on IEC 61499 that require a clear separation of interfaces and logic.

ATM_AX_TOF





• – ○ • – ○ • – ○ •

Introduction

The function block `ATM_AX_TOF` implements off-delay timing with a modular adapter interface based on the types `AX` and `ATM`. It is typically used to keep an output active for a defined period after an input signal is lost. The block is classified as a standard timer function block according to IEC 61499-2 and is implemented internally with the function block `E_TOF`.

Interface Structure

Event Inputs

Name	Type	Comment
R	Event	Resets the timer

Event Outputs

No direct event outputs are available. The output is provided via the `Q` adapter.

Data Inputs

No direct data inputs are available. Data is provided via the `IN` and `PT` adapters.

Data Outputs

No direct data outputs are available. The output is provided via the `Q` adapter.

Adapters

Name	Direction	Type	Description
IN	Socket	AX	Input adapter: Provides the Boolean trigger signal (D1: BOOL) and an event (E1) to trigger the timer function
PT	Socket	ATM	Time adapter: Provides the delay time (D1: TIME)
Q	Plug	AX	Output adapter: Provides the Boolean output signal (D1: BOOL) and an acknowledgment event (E1)

Functionality

The `ATM_AX_TOF` operates on the principle of an **off-delay**:

1. **Normal operation:** As long as the input `IN.D1` (BOOL) has the value `TRUE`, the output `Q.D1` is immediately `TRUE` (without delay).
2. **Shutdown Delay:** When `IN.D1` changes from `TRUE` to `FALSE`, `Q.D1` remains at `TRUE` for the time specified in `PT.D1` (TIME). After this time has elapsed, `Q.D1` changes to `FALSE`.
3. **Reset:** An event at input `R` immediately resets the internal timer. `Q.D1` immediately becomes `FALSE`, regardless of the current state of `IN.D1` or the remaining time.

The trigger is the event `IN.E1` – a new value at `IN.D1` is only applied after an event. An event on `Q.E1` confirms every state change of `Q.D1`.

Technical Features

- **Adapter-Based Interface:** The use of the universal adapters `AX` and `ATM` enables flexible connection to addressing or other modules without fixed wiring.
- **Internal Implementation:** The function block uses the standard FB `E_TOF` from the IEC 61499 library, ensuring robust and tested timer logic.
- **Reset Function:** The reset input `R` allows for immediate cancellation of the delay, e.g., for controlling safety functions.
- **Compatibility:** Developed for the 4diac IDE and based on the Eclipse Public License 2.0.

State Overview

The function block goes through the following operating states:

State	Conditions	Q.D1	Description
Idle	IN.D1 = FALSE, timer is not running	FALSE	Idle state after expiration or reset
Active	IN.D1 = TRUE	TRUE	Input active, output immediately TRUE
Timing	IN.D1 changes from TRUE to FALSE, timer active	TRUE	Delay phase: Output remains TRUE for the duration of the delay phase

Resetting	Event R during Timing or Active	FALSE	Timer is stopped immediately, output goes to FALSE

Application Scenarios

- **Fan Run-On:** After a motor is switched off, the fan continues to run for an adjustable time to dissipate residual heat.
- **Signal Debouncing:** Short interruptions in a sensor signal are bridged by the delay.
- **Lighting Control:** Lights remain on for a defined run-on time after leaving a room.
- **Safety Shutdowns:** Reset via R enables immediate shutdown in emergencies.

Comparison with Similar Function Blocks

Property	ATM_AX_TOF	E_TOF (Standard)
Interface	Adapter (AX/ATM)	Direct Events/Data
Reset Function	Yes (Event R)	Yes (Event R)
Triggering	Event-driven via adapter event	Event-driven via REQ
Flexibility	Increased through adapter coupling	Simpler, but more rigid
Application Area	Modular automation systems	Basic function in IEC 61499 editors

Conclusion

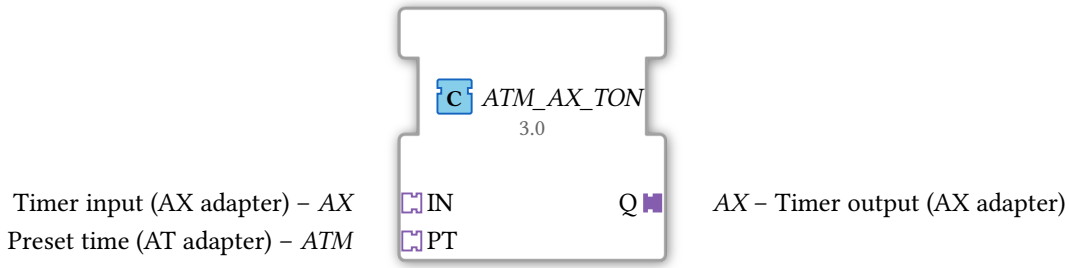
The `ATM_AX_TOF` offers a reliable and flexible turn-off delay for modular automation projects. Thanks to the adapter interface, it can be easily integrated into various system landscapes and extends the functionality of the standard timer `E_TOF` by adding a reset option and loose coupling via AX/AT adapters. It is ideal for applications requiring a delayed shutdown with optional immediate reset.

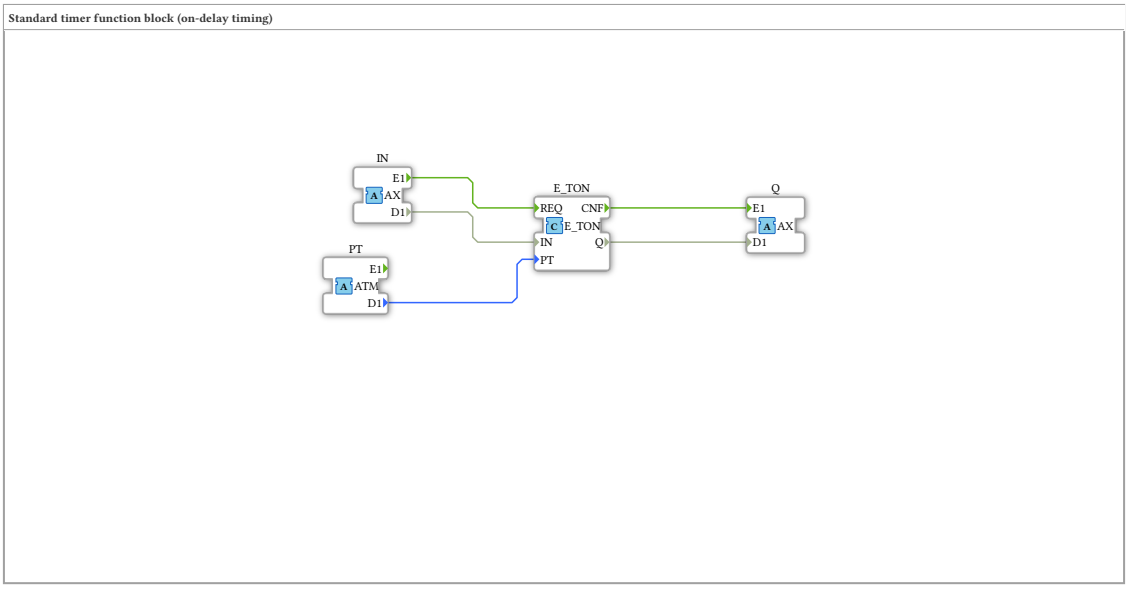
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ATM_AX_TON





• – ○ • – ○ • – ○ •

Introduction

The **ATM_AX_TON** is a function block for on-delay. It implements standard timer functionality according to IEC 61499-2, which is controlled via an adapter interface (AX/AT). The block allows a timer to be started via an input adapter and the output state and confirmation of completion to be provided via an output adapter.

Interface Structure

Event Inputs

Name	Type	Comment
EIPT	Event	Sets the lead time (non-triggering)

Event Outputs

The function block does not have its own event outputs. Acknowledgments and state changes are communicated via the **Q** adapter (E1).

Data Inputs

The function block (FB) has no explicit data inputs. All input data is provided via the adapter interfaces.

Data Outputs

The FB has no explicit data outputs. Output data is provided via the **Q** adapter (D1).

Adapters

Direction	Name	Type	Comment
Socket	IN	adapter::types::unidirectional::AX	Timer input (AX adapter) – provides the Boolean input signal and the start pulse
Socket	PT	adapter::types::unidirectional::ATM	Lead time (AT adapter) – provides the delay time
Plug	Q	adapter::types::unidirectional::AX	Timer output (AX adapter) – outputs the timer status and confirmation

Functionality

The **ATM_AX_TON** encapsulates a standard on-delay timer (E_TON) from the IEC 61499 library. Detailed functionality:

1. **Starting the timer:**

The input adapter **IN** provides the start pulse (REQ) via its event **E1** and the Boolean value **IN** via its data signal **D1**. When **IN.D1** becomes TRUE, the timer starts.

1. **Delay time:**

The desired delay time is provided via the adapter **PT** (via **PT.D1**). Once the timer is running, the output **Q.D1** is set to TRUE after this delay time has elapsed.

1. **Output Signal and Acknowledgement:**

The output adapter **Q** outputs the current timer state via **Q.D1** (TRUE when the delay time has elapsed, FALSE when the input goes to FALSE). The event **Q.E1** is sent after each state change (CNF of the internal E_TON).

1. **Setting the Lead Time Without Triggering:**

According to the specification, the event input **EIPT** is used to set the lead time **PT** without starting the timer. However, in this FBNetwork, this input is not connected internally – it is available for external use if an application requires a time preset.

1. **Fallback:**

If **IN** goes to FALSE, the timer is immediately reset and **Q.D1** becomes FALSE (without delay).

Technical Features

- **Adapter-Based Interface:** The function block uses only adapters (AX and ATM) for communication, not traditional input/output grooves. This enables flexible encapsulation and reuse in complex systems.
- **Unidirectional Adapters:** The adapters used are unidirectional (AX for Boolean values and ATM for time values). The function block expects the adapters to be correctly integrated into the higher-level hierarchy.
- **Unconnected Event Input EIPT:** The **EIPT** input is defined but not connected to the internal E_TON. This could indicate a planned extension or external use by the user. In the current version, it has no direct function within the function block network.
- **Authors and Version:** The function block was originally developed by Franz Höpfinger (HR Agrartechnik GmbH) and later revised by Patrick Aigner (version 3.0, April 2025). It is available under the Eclipse Public License 2.0.

State Overview

The FB itself does not manage its own state machine – the state logic resides in the integrated **E_TON**. Nevertheless, its behavior can be described as follows:

Input IN State	Behavior	Output Q.D1
FALSE → TRUE (rising edge)	Timer starts; after the timer expires, Q becomes TRUE	FALSE (until the timer expires) → TRUE
TRUE (while the timer is running)	Timer continues running	FALSE
TRUE → FALSE (falling edge)	Timer is immediately reset	TRUE → FALSE
TRUE (after the timer expires)	Output remains TRUE as long as IN is TRUE	TRUE
FALSE (any)	Output remains FALSE	FALSE

Application Scenarios

- **Machine Control:** Delaying the activation of drives or valves to prevent multiple components from starting simultaneously.
- **Signal Conditioning:** Debouncing switching signals by only considering a short pulse as valid after a defined time.

- **Process Automation:** Initiating subsequent steps only after a waiting period (e.g., in conveyor belts or dosing systems).
- **Adapter Integration:** Use in systems already based on the AX/AT adapter protocol without additional data lines.

Comparison with Similar Function Blocks

Function Block	Type	Special Feature
E_TON (IEC 61499)	Standard On-Delay	Classic input/output interface; direct event and data ports
ATM_AX_TON	Adapter-based	Encapsulates E_TON and provides an AX/AT adapter; additional EIPT input (not connected)
E_TOF	Off-Delay	Delay on power-off (opposite behavior)
E_CTU	Up-counter	Counts events, not a timer

The **ATM_AX_TON** differs in its pure adapter interface and the lack of direct data coupling. It is particularly suitable for modular, hierarchically structured controllers.

Conclusion

The **ATM_AX_TON** is a ready-made on-delay timer with a modern, adapter-based interface. It simplifies the integration of time delays into larger systems that already use the AX/AT concept. The unconnected **EIPT** input offers potential for future expansions or customer-specific adaptations. Overall, the module is robust, well-documented, and available under an open-source license.

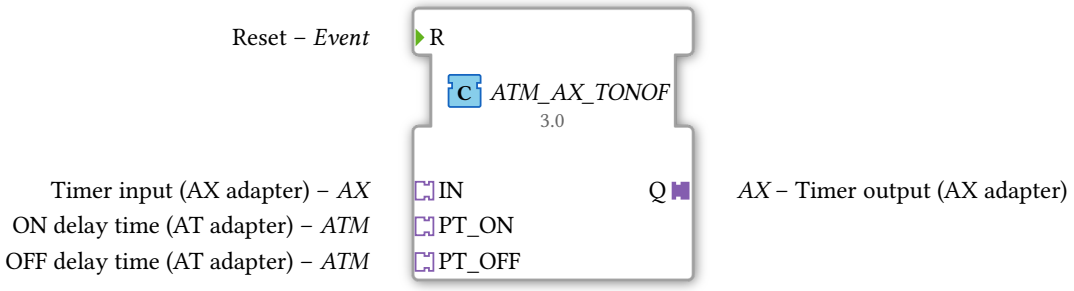
The **ATM_AX_TON** is a ready-to-use on-delay timer with a modern, adapter-based interface. It simplifies the integration of time delays into larger systems that already use the AX/AT concept. The unconnected **EIPT** input offers potential for future expansions or customer-specific adaptations. Overall, the module is robust, well-documented, and available under an open-source license.

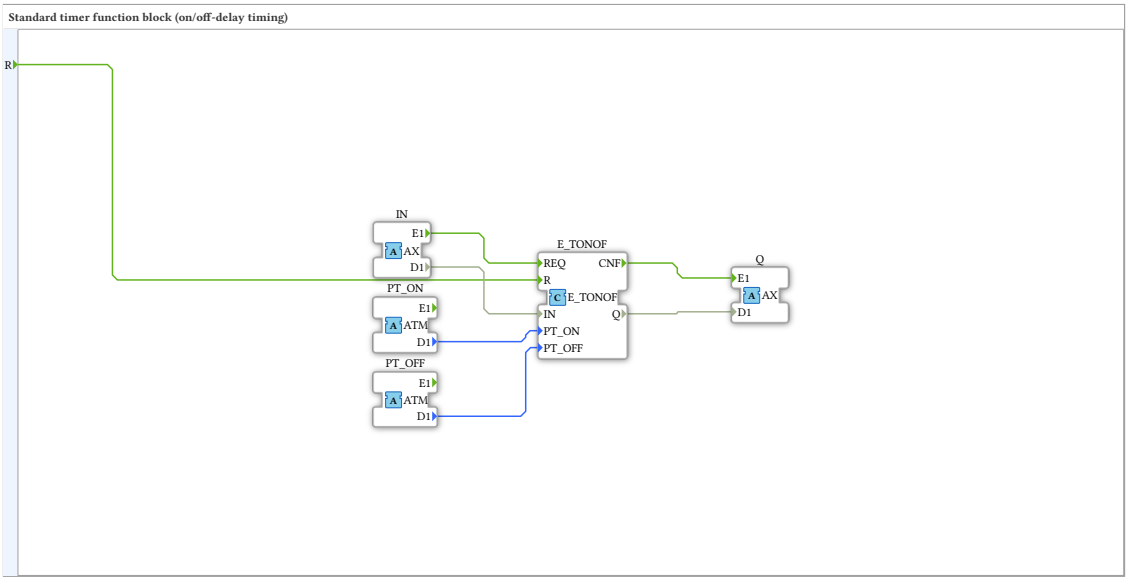
The **ATM_AX_TON** is a ready-to-use, adapter-based interface. —

Related topic subpages on ms-muc-docs.de

- E_CTU Event Counter module on ms-muc-docs.de

ATM_AX_TONOF





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_AX_TONOF** implements a combined on/off delay timer. It is designed as a standard timer module according to IEC 61499 and uses only adapter interfaces for communication. This allows it to be flexibly integrated into various communication structures. The module processes a Boolean input signal and outputs the delayed signal – both the on delay (ON-Delay) and the off delay (OFF-Delay) are configured via separate adapters.

Interface Structure

Event Inputs

Name	Type	Description
R	Event	Resets the timer. Both running delays are aborted, and output Q is immediately set to FALSE.

Event Outputs

The function block does not have any independent event outputs. The output event is provided via the output adapter **Q** (see Adapters).

Data Inputs

The function block does not have any direct data inputs. All necessary data (Boolean input and delay times) is read via the adapter interfaces.

Data Outputs

There are also no dedicated connections for data outputs. The current output state (BOOL) is output via the output adapter **Q**.

Adapters

Name	Direction	Type	Description
IN	Socket	AX (unidirectional)	Input adapter that triggers execution via its event E1 and provides the Boolean input value via its data D1.
PT_ON	Socket	ATM (unidirectional)	Adapter for the power-on delay time (TIME). The value is provided via D1.

PT_OFF	Socket	ATM (unidirectional)	Adapter for the power-off delay time (TIME). The value is provided via D1.
Q	Plug	AX (unidirectional)	Output adapter. Event E1 signals the update of the output value; data D1 contains the Boolean state of Q (TRUE or FALSE).

Functionality

The function block operates according to the following sequence:

- When the input signal **IN** changes from FALSE to TRUE, the timer starts the power-on delay with the time **PT_ON**. After this time has elapsed, the output **Q** is set to TRUE.
- When **IN** changes from TRUE to FALSE, the timer starts the power-off delay with the time **PT_OFF**. After this time has elapsed, **Q** is set to FALSE.
- An event at the reset input **R** immediately stops both running timers and sets **Q** to FALSE, regardless of the current state of **IN**.

The internal implementation uses the standardized function block **E_TONOF** from the IEC 61499 library. This ensures defined, robust timing behavior.

Technical Features

- **Adapter-based interface**: All inputs and outputs are handled via adapters (AX, ATM) instead of discrete event and data ports. This facilitates integration into modular systems and allows for a clean separation of communication and logic.
- **Standard compliance**: The function block is classified as a standard timer according to IEC 61499 and can be used in any 4diac IDE project.
- **Reset function**: The separate reset input enables immediate resetting without dependence on the current timer state.
- **Reusable adapter types**: The AX and ATM adapters are used multiple times in the application, so no additional type definitions are required.

State Overview

The internal timer **E_TONOF** cycles through the following states:

1. **IDLE**: No delay active. Output Q = state of IN (if updated after a previous delay).
2. **START_ON**: After the rising edge of IN – the timer for PT_ON runs.
3. **START_OFF**: After the falling edge of IN – the timer for PT_OFF runs.
4. **RESET**: Triggered by the event R, this interrupts any active timer and returns to the IDLE state with Q = FALSE.

Application Scenarios

- **Delayed Switching On and Off of Actuators**: For example, to prevent bouncing or to stagger switching operations.
- **Motor Protection**: Switch-on delay prevents frequent switching on/off cycles; switch-off delay enables a defined overrun.
- **Door Control**: ON delay for time-delayed closing after release, OFF delay for automatic opening after a waiting period.
- **HMI Filtering**: Suppression of short, unwanted signal changes through adjustable dead times.

Comparison with similar function blocks

- **E_TON (TON)**: Implements only an on delay; the output is set to TRUE after the delay and remains TRUE as long as IN=TRUE.
- **E_TOF (TOF)**: Implements only an off delay; the output is set to TRUE immediately when IN=TRUE and falls back to FALSE after the delay when IN=FALSE.
- **E_TONOF**: Combines both delays in one function block. The **ATM_AX_TONOF** extends this concept with an adapter interface, thus simplifying integration with other adapter-based components. It offers the same functionality but with increased flexibility in system integration.

E_TONOF: Combines both delays in one function block.

Conclusion

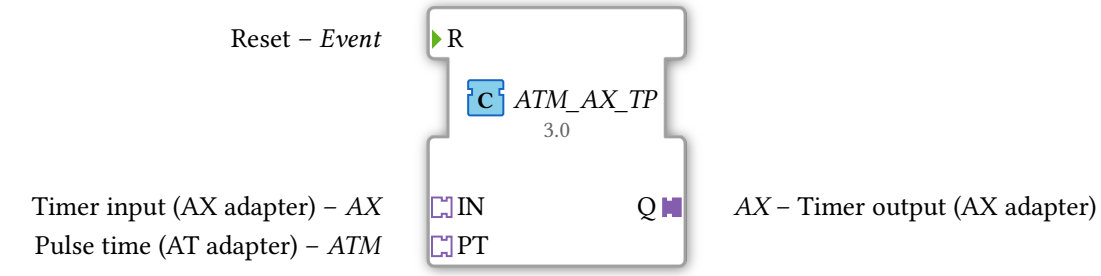
The **ATM_AX_TONOF** is a powerful timer function block for delayed on/off switching operations. Its adapter-based interface allows it to be seamlessly integrated into modular automation solutions while remaining fully compliant with the IEC 61499 standard. Combining ON delay and OFF delay in a single block reduces wiring effort and simplifies application development. Thanks to its separate reset input and robust internal implementation, it is suitable for a wide range of industrial applications.

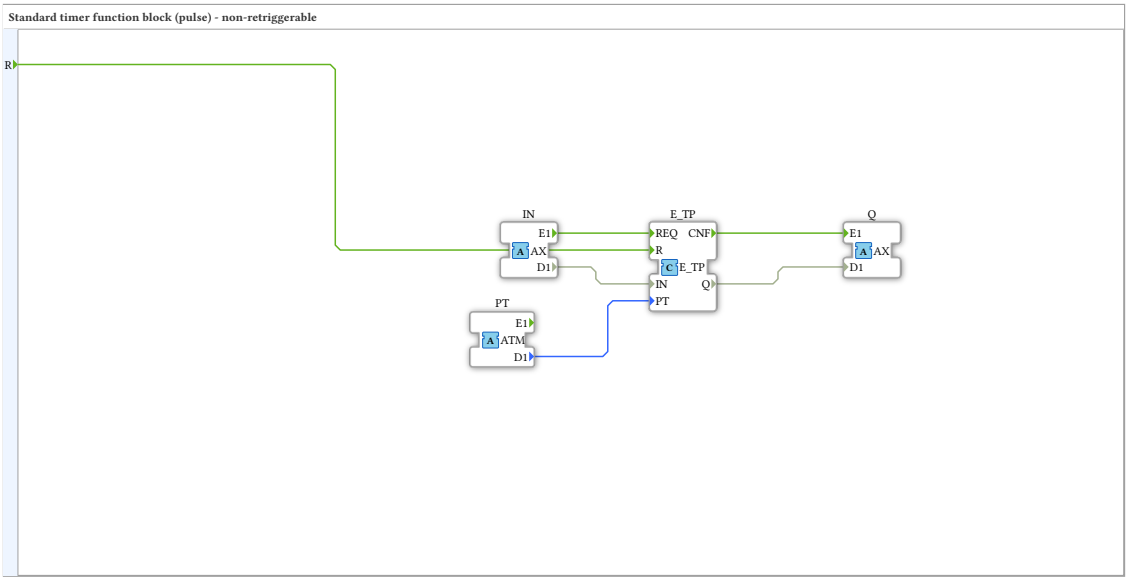
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ATM_AX_TP





• – ○ • – ○ • – ○ •

Introduction

The **ATM_AX_TP** is a special timer function block that generates a time-limited pulse from an input signal. It implements the functionality of a **non-retriggerable pulse shaper**. The block is controlled via an AX adapter and outputs the result via another AX adapter; the pulse duration is specified via a separate ATM adapter. The adapter interfaces make it particularly easy to integrate into modular automation systems.

Interface Structure

Event Inputs

Name	Type	Comment
R	Event	Resets the timer

Event Outputs

No dedicated event outputs – the pulse output is provided via the **Q** adapter (see Adapter section).

Data Inputs

No direct data inputs. The input values (IN and PT) are provided via the corresponding adapter sockets.

Data Outputs

No direct data outputs. The output value (Q) is provided via the **Q** adapter.

Adapters

Name	Type	Direction	Description
Q	AX	Plug	Output: Provides the pulse signal
IN	AX	Socket	Input: Receives the triggering signal
PT	ATM	Socket	Input: Provides the pulse duration (pulse time)

Note: The adapters are of type *adapter::types::unidirectional::AX* (or *ATM*) and transmit an event along with a data value.

Functionality

The **ATM_AX_TP** operates based on the standardized IEC 61499 function block **E_TP**, which is instantiated within the network.

- **Triggering:** A rising edge at input **IN** (event IN.E1 and data value IN.D1 = TRUE) starts the timing.
- **Pulse Generation:** The output **Q** is immediately set to TRUE and remains active for the duration **PT**. After this time has elapsed, Q returns to FALSE.
- **Non-Retriggerable:** During an ongoing pulse, a subsequent rising edge from IN is ignored – the pulse continues at its original length.
- **Reset:** An event at input **R** immediately ends the current pulse (Q = FALSE) and resets the timer. A new pulse can then be started.

Internal wiring:
IN.E1 ⟶ E_TP.REQ `IN.D1 → E_TP.IN` PT.D1 ⟶ E_TP.PT `E_TP.CNF → Q.E1` (Event output at pulse end) E_TP.Q ⟶ Q.D1 `(Output data value) R → E_TP.R`

Technical Features

- **Adapter-based connection:** Unlike a direct E_TP, ATM_AX_TP operates exclusively via adapters (AX and ATM). This allows for loose coupling in distributed systems and simplifies the exchange of signal types.
- **Compatibility:** This function block is optimized for use with the Eclipse 4diac framework (recognizable by the package attribute `adapter::events::unidirectional::timers`).
- **Versioning:** Multiple versions document the evolution (reset function, package changes).
- **License:** Eclipse Public License 2.0 – the source code may be used, modified, and distributed.

State Overview

This function block does not have any explicit state machines at the top level – the state logic is implemented entirely by the internal **E_TP**. This state cycles through the following states:

- **IDLE** – waiting for the rising edge of IN
- **ACTIVE** – pulse is running (Q = TRUE, timer is running)
- **RESET** – returned to IDLE by event R (pulse terminates prematurely)

After the timer expires, E_TP automatically switches from ACTIVE to IDLE.

Application Scenarios

- **Control of momentary actuators:** Valves, lights, signal transmitters that should only be switched on for a specific duration.
- **Debouncing / Pulse shaping:** An incoming signal (e.g., push button) is converted into a precisely defined pulse.

- **Modular automation:** Thanks to the adapter interfaces, the function block is suitable for use in distributed systems where signals are exchanged between different components.

Comparison with similar function blocks

Function block	Properties	Difference from ATM_AX_TP
E_TP (Standard)	Direct event/data connections, no adapter.	ATM_AX_TP encapsulates E_TP and provides adapters.
E_R_TRIG	Edge detection without pulse extension.	Generates only a very short pulse, no time control.
E_CYCLE	Cyclic timer, repeats pulses.	Single pulse.

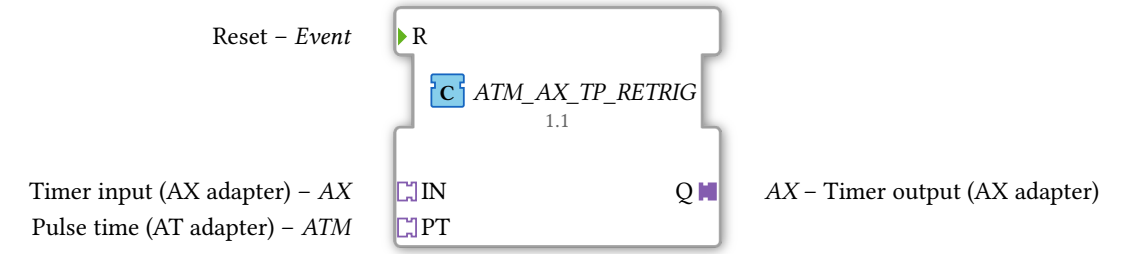
Conclusion

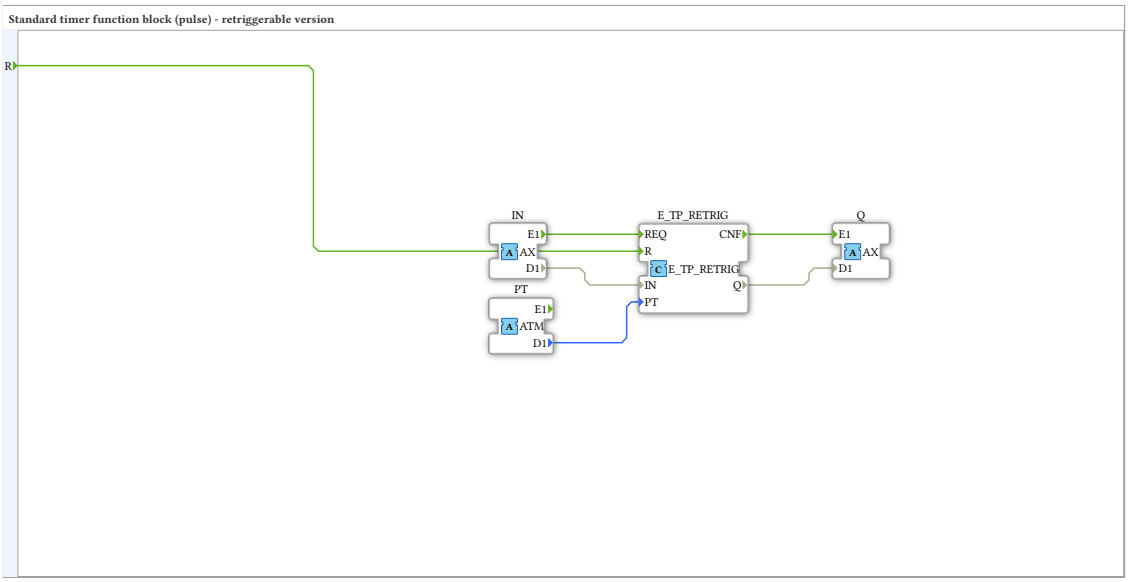
The **ATM_AX_TP** is a useful function block for generating a non-retriggerable pulse with adjustable duration. Its special feature lies in the adapter-based interface, which enables flexible and loose coupling in IEC 61499 systems. The embedded standard timer E_TP ensures robust and proven functionality. A practical function block for all applications where a defined time window needs to be triggered by an external signal.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

ATM_AX_TP_RETRIG





• – ○ • – ○ • – ○ •

Introduction

The `ATM_AX_TP_RETRIG` is a retriggerable pulse timer function block. It generates a pulse of defined duration at the output when the input switches to TRUE. During the pulse's execution, additional triggers can restart the timer (retrigger function). A reset input immediately terminates the pulse. The block uses standardized adapters for input and output, enabling flexible integration into modular control systems.

Interface Structure

Event Inputs

- **R** (Event) – Reset: Resets the timer and terminates the current pulse.

Event Outputs

- **Q.E1** (Event) – Output Event: Triggered via adapter Q when the pulse state changes (start or end).

Data Inputs

- **IN.D1** (BOOL) – Trigger Input: On a rising edge and a TRUE value, the timer starts.
- **PT.D1** (TIME) – Pulse Duration: Specifies the time for which output Q is set to TRUE after a trigger.

Data Outputs

- **Q.D1** (BOOL) – Pulse Output: Displays TRUE during the pulse duration, otherwise FALSE.

Adapters

- **IN** (Socket, Type AX) – Input Adapter: Couples the trigger event (E1) and the Boolean trigger value (D1).
- **PT** (Socket, Type ATM) – Time Adapter: Returns the pulse duration as a TIME value (D1).
- **Q** (Plug, Type AX) – Output Adapter: Provides the pulse state (D1) and an event on state change (E1).

Functionality

The function block internally uses the standard function block `E_TP_RETRIG`. An event at the input **IN** (rising edge of IN.E1) starts the timer, provided **IN.D1** = TRUE. The output **Q.D1** is set to TRUE for the time specified in **PT.D1**. During this time, a subsequent trigger (repeated IN.E1 with IN.D1 = TRUE) triggers a retrigger: The timer is reset, and the pulse starts again. An event at the reset input **R** immediately ends the pulse, and **Q.D1** becomes FALSE. After the pulse duration has elapsed without a further trigger, **Q.D1** automatically returns to FALSE.

Technical Features

- **Retriggerable**: Multiple triggers during a running pulse restart the timer without extending the pulse duration.
- **Adapter-based interface**: The IN, PT, and Q adapters ensure a clean separation of event and data communication.
- **Internal standard module**: The actual timer logic is executed by the IEC 61499-compliant `E_TP_RETRIG` module.
- **Unidirectional adapters**: Both the AX and ATM adapters are unidirectional, clearly defining the data flow direction.

State overview

The module has two main states:

State	Q.D1	Description
IDLE	FALSE	Idle state, no pulse active.
TIMING	TRUE	Pulse running, timer counts down the duration PT.

Transitions:

- IDLE ⟶ TIMING: Trigger (IN.E1 with IN.D1 = TRUE)
- TIMING ⟶ TIMING (Retrigger): A subsequent trigger restarts the timer.
- TIMING ⟶ IDLE: Timer expires or Reset (R)
- IDLE ⟶ IDLE: Reset has no effect

Application Scenarios

- **Pulse Control for Actuators**: Control of valves, lamps, or motors that are activated for a defined time upon receiving a start signal.
- **Safety Applications**: Retriggerable pulses allow for a renewed request during an ongoing action, e.g., for button retriggering.
- **Modular Automation Systems**: Use in combination with other adapter-based components for standardized communication via AX/ATM interfaces.

Comparison with Similar Function Blocks

- *E_TP (non-retriggerable)*: Ignores a subsequent trigger during the running pulse; once started, the pulse always runs its full course.
- *E_TP_RETRIG (Standard)*: Same timer logic, but without an adapter interface – connected directly via event and data inputs.
- *TON / TOF*: Other timer types; TON delays power-on, TOF delays power-off – neither generates pulses.

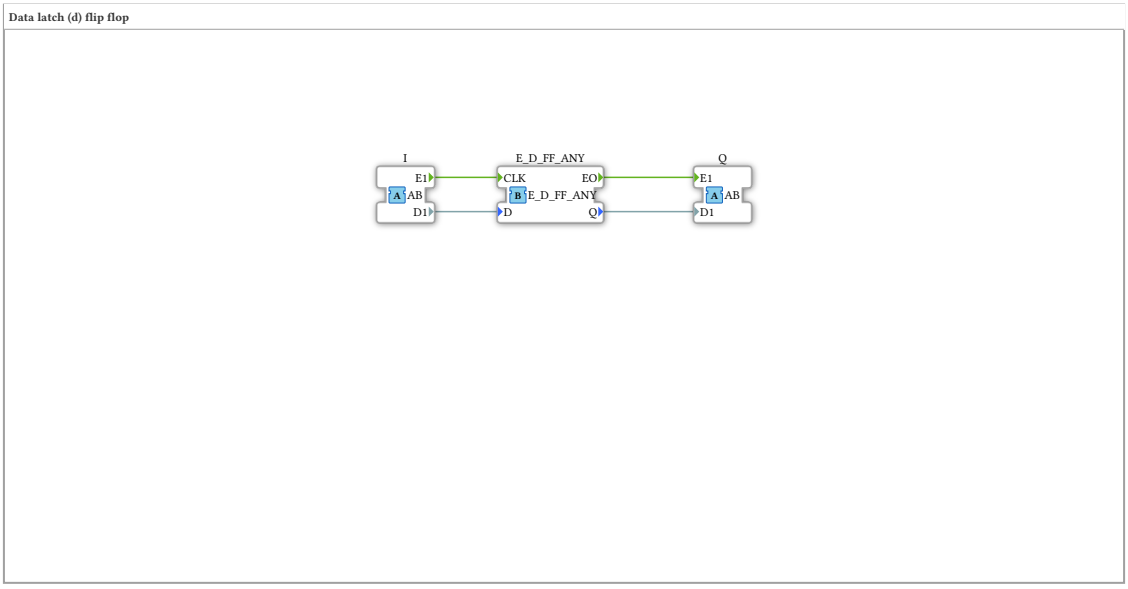
Conclusion

The `ATM_AX_TP_RETRIG` combines the proven retriggerable pulse function with an adapter-based interface. By using standardized AX and ATM adapters, it is particularly well-suited for modular and reusable control solutions. The retrigger feature offers flexibility in applications where a pulse needs to be restarted upon request.

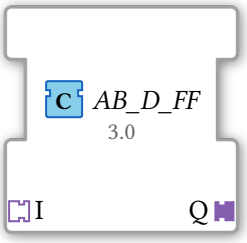
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AB_D_FF



No image available.



Value to latch – *AB*

AB – Latched value

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_D_FF** implements an asynchronous data latch (D flip-flop) based on an adapter concept according to IEC 61499. It receives a data value via the input adapter **I** as soon as an event occurs and outputs this value via the output adapter **Q**. The internal state of the flip-flop is represented by a standardized `E_D_FF_ANY` function block.

Interface Structure

Event Inputs

- **I.E1** (via adapter `I`): Initiates the data value acquisition.

Event Outputs

- **Q.E1** (via adapter `Q`): Signals the output of the latched data value.

Data Inputs

- **I.D1** (via adapter `I`): The data value to be latched (any type, depending on the adapter type used, `adapter::types::unidirectional::AB`).

Data Outputs

- **Q.D1** (via adapter `Q`): The latched and output data value.

Adapters

- **I** (Socket): Input adapter of type `adapter::types::unidirectional::AB` – receives the value to be latched and the control event.
- **Q** (Plug): Output adapter of the same type – outputs the stored value and the associated event.

Functionality

The function block operates in an event-driven manner:

1. As soon as an event arrives at the input adapter **I.E1**, the current value at the data input **I.D1** is transferred to the internal D flip-flop (`E_D_FF_ANY`).
2. Subsequently, the stored value is output at **Q.D1**, and an event is simultaneously generated at the output **Q.E1**.

Thus, the component behaves like a classic edge-triggered D latch, where the transfer and output occur synchronously with the input event.

Technical Features

- **Adapter Interface**: The component uses only adapters (`adapter::types::unidirectional::AB`) for input and output, which allows for flexible typing and loose coupling with other components.
- **Internal Implementation**: The latching core is implemented using the standardized function block `E_D_FF_ANY` from the IEC 61499 library `iec61499::events`. This stores the data value and generates the output event.
- **Event and data paths** are strictly separated: The event triggers the data transfer; data flows independently.

State Overview

The function block does not have an explicit internal state machine but relies on the implicit state of the embedded `E_D_FF_ANY`:

- **Idle**: Waiting for an event on `I.E1`.
- **Latching**: After receiving the event, the data value is transferred and output.
- The internally stored value is retained until the next event.

Application Scenarios

- **Data Synchronization**: Reliable transfer of measured values or control data at defined times (e.g., clock pulse).
- **Buffering**: Intermediate storage of values in sequence control systems between different function blocks.
- **Register Creation**: Construction of shift registers or memory cells in modular automation systems.

Comparison with Similar Function Blocks

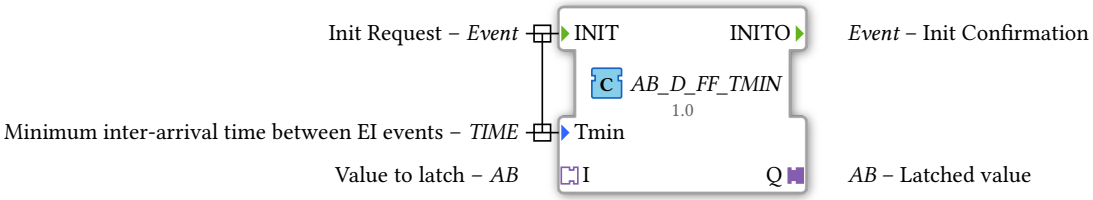
Function Block	Description
AB_SR_FF	Set-Reset Flip-Flop, stores two states (Set/Reset) without a data value.
AB_T_FF	Toggle Flip-Flop, changes the output state with each event.
E_D_FF	Standard D Flip-Flop without an adapter interface, direct event and data pins.

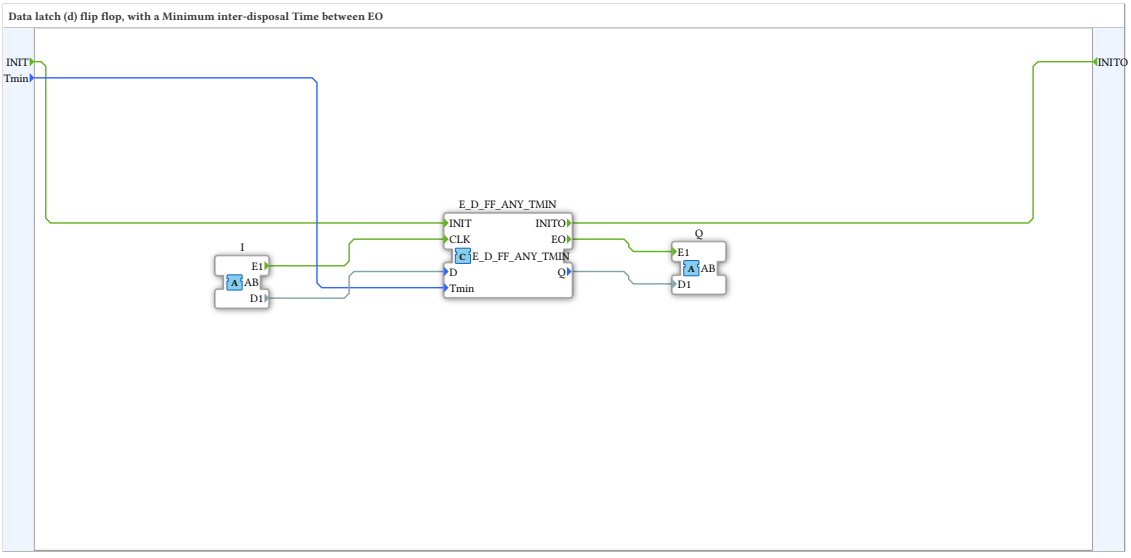
The **AB_D_FF** offers type-safe and reusable encapsulation through its adapter interface, whereas comparable devices often rely on fixed data types.

Conclusion

The **AB_D_FF** is a compact and flexible D-latch device that leverages the advantages of the IEC 61499 architecture. Its adapter interface allows it to be used in a wide variety of contexts without being restricted to a specific data type. Together with the internal standard flip-flop, it ensures reliable and event-driven data storage.

AB_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The **AB_D_FF_TMIN** is an adapter-based function block (FB) according to IEC 61499 that implements the functionality of a data latch (D flip-flop) with a minimum time condition between successive output events. It serves to transfer a digital data value from an adapter socket (input) to an adapter plug (output), ensuring that the output events (EO) do not occur faster than permitted by a configured time parameter `Tmin`. The block abstracts the pure D flip-flop logic by using an internal flip-flop block and encapsulates data and events in adapter interfaces.

Interface Structure

Event Inputs

Name	Type	Comment

INIT	EInit	Initialization Request (sets the minimum time)

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Acknowledgement

Data Inputs

Name	Type	Comment
Tmin	TIME	Minimum time between two output events (EO)

Data Outputs

No direct data outputs; Data output is handled via the adapter plug Q.

Adapter

Direction	Name	Type	Comment
Socket	I	adapter::types::unidirectional::AB	Input adapter: returns event E1 and data value D1 (the value to be latched)
Plug	Q	adapter::types::unidirectional::AB	Output adapter: outputs the latched data value D1 and the event E1

Note: The adapter unidirectional::AB defines an event E1 and a data value D1 (typically a Boolean or any generic type). The exact type definition of the adapter is outside this function block.

Functionality

The function block works internally with the function block E_D_FF_ANY_TMIN (type iec61499::events::E_D_FF_ANY_TMIN), which implements the core logic of the D flip-flop with a minimal hold time between output events. The functionality is structured as follows:

1. **Initialization:**

An event at INIT passes the parameter Tmin (minimum time) to the inner function block. The function block acknowledges this with INITO.

1. **Data Acquisition and Transmission:**

- An event E1 occurs at socket I, which carries the data value D1 (the signal to be latched).
- This event is forwarded as CLK to the inner flip-flop, which simultaneously receives the data value D1 from I.D1.
- The inner flip-flop stores the value D1 and outputs it at its output Q.
- The latched value is passed out via plug Q.D1, and at the same time, the output event EO of the inner flip-flop is triggered.
- This EO is forwarded to Q.E1, so that the receiving adapter is informed of the update.

` 3. **Time Control:**
The internal function block E_D_FF_ANY_TMIN ensures that after an output event EO, no further output events can be generated for the duration of Tmin – regardless of how quickly input events arrive. If the time between two E1 inputs exceeds the Tmin threshold, the value is immediately adopted; otherwise, it is blocked until the minimum time has elapsed.

Technical Features

- **Adapter-Based Interface:**

Instead of individual event/data ports, the function block uses adapters that transport events and data as an encapsulated unit. This increases reusability and reduces the number of external connections.

- **Minimum Output Time (Tmin):**

The setting `Tmin` prevents output events from being triggered too quickly, which is useful in time-critical control systems (e.g., for debouncing or maintaining processing cycles).

- **Internal Use of a Specialized Function Block:**

The function block delegates the actual flip-flop logic to the system-internal function block `E_D_FF_ANY_TMIN`, which is responsible for correct timing. This keeps the external function block lean and error-free.

- **Typed Adapter:**

The adapter type `adapter::types::unidirectional::AB` is defined as unidirectional: data and events flow only in one direction (from the socket to the plug). This makes the function block strictly upward compatible.

State Overview

The function block does not have an explicit visible state machine; The state is determined by the internal flip-flop and the timing:

- **State 0 (initial / not latched):** After INIT or a reset (not explicitly implemented), the state is undefined. The value `Q.D1` is only set after the first valid `E1` event at the socket.
- **State 1 (latched and active):** The internal flip-flop has a data value stored. After an output event, a lock state is maintained for the duration `Tmin`, during which incoming `E1` events are ignored or delayed. After the minimum time has elapsed, the function block is ready for the next takeover.
- **State 1 (latched and active):** (No SFC or ECC is defined in the XML; the state logic is entirely contained within the internal function block.)

Application Scenarios

- **Signal Debouncing:**

If a sensor or button generates multiple rapid pulses (bounce), the function block (FB) can prevent multiple values from being passed on in quick succession by setting `Tmin` to, for example, 20 ms.

- **Cyclic Data Passing with Minimum Interval:**

In a distributed control system, a measured value should only be sent to a higher-level component at certain minimum intervals. The FB acts as a buffer that limits the update rate.

- **Reliable Transfer in Time-Critical Systems:**

When transferring a value from a fast to a slower clock range, the FB can be set to a suitable `Tmin`.Prevent data loss or noise.

Comparison with similar components

Component	Properties
AB_D_FF	Pure D flip-flop without a minimum output time – events are passed on immediately.
E_D_FF_ANY	Standard event data flip-flop without a time limit, but with discrete ports (no adapters).
AB_D_FF_TMIN	Combines adapter encapsulation with a configurable minimum time between output events.
R_TRIG / F_TRIG	Detects rising/falling edges – no functionality for latching a data value.

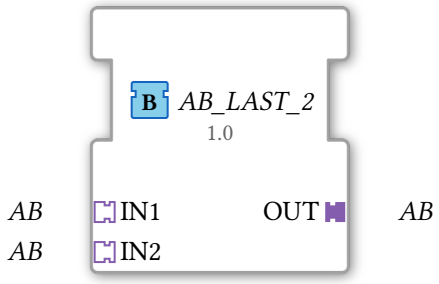
The **AB_D_FF_TMIN** stands out due to the integration of timing control into adapter-based D flip-flop logic and is therefore particularly suitable for applications requiring both data latching and temporal decoupling.

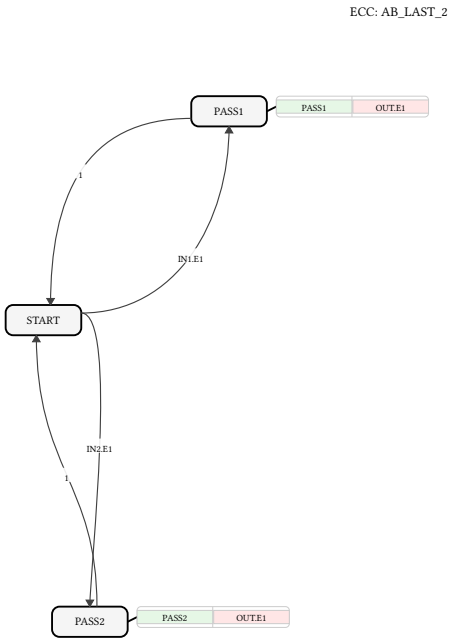
Conclusion

The **AB_D_FF_TMIN** is a compact, adapter-based function block that extends the basic functionality of a D flip-flop by adding the ability to maintain a minimal waiting time between output events. The encapsulated adapter interface and the internal use of a specialized flip-flop component ensure high

reusability and a clear separation of logic and timing control. This component is particularly well-suited for industrial control applications where signal processing with temporal constraints is required.

AB_LAST_2





Introduction

The AB_LAST_2 function block merges two same-type BYTE adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: BYTE adapter (unidirectional) - first signal source
- **IN2**: BYTE adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: BYTE adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event

is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

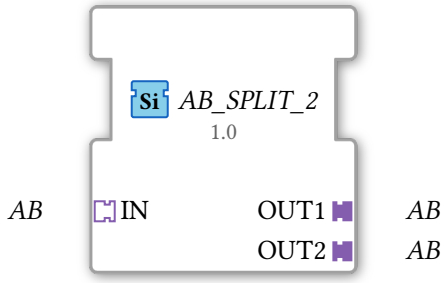
Application Scenarios

- Merging two equally-valid BYTE signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same BYTE value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AB_LAST_2 is the inverse of `AB_SPLIT_2`: AB_SPLIT_2 distributes 1 input to 2 outputs, AB_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

AB_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_2** splits a unidirectional adapter of type `AB` into two separate adapter outputs. It is implemented as a generic function block (Generic FB) and allows the simultaneous forwarding of the incoming adapter to two independent target addresses. This simplifies the topology in control applications when a signal is needed multiple times without affecting the original data flow.

Interface Structure

Event Inputs

No event inputs are available.

Event Outputs

No event outputs are available. Adapter communication occurs without events via the data exchange mechanism of the adapter type.

Data Inputs

No data inputs are available. Data flows exclusively through the adapter input `IN`.

Data Outputs

No data outputs are available. Outputs are provided via the adapter outputs `OUT1` and `OUT2`.

Adapter

Role	Name	Type	Direction
Socket	IN	`adapter::types::unidirectional::AB`	Input

Plug	OUT1	`adapter::types::unidirectional::AB`	Output
Plug	OUT2	`adapter::types::unidirectional::AB`	Output

Functionality

The function block forwards the adapter connected via socket `IN` unchanged to both plugs `OUT1` and `OUT2`. Any change to the data or state of the incoming adapter is propagated simultaneously to both outputs. The distribution occurs without delay or buffering – it is a pure distribution logic.

Technical Features

- **Generic Function Block:** The function block is declared as a generic type (`GEN_AB_SPLIT`), so it can work with different subtypes of the adapter `AB`, provided they have the same unidirectional semantics.
- **Unidirectionality:** The adapter is designed exclusively in one direction – data flows from the socket to the plugs. Backward communication is not supported.
- **License:** This function block is licensed under the Eclipse Public License 2.0 (EPL-2.0), which permits free use, modification, and distribution in your own projects.
- **No State Machines:** The function block has no internal states or event-driven processes, as adapter routing is entirely data-driven.

State Overview

The function block does not have a state machine. The output adapters always reflect the current state of the input adapter.

Application Scenarios

- **Signal Distribution in Modular Systems:** Separation of a sensor signal from a fieldbus component for parallel processing in two independent control logics.
- **Redundancy Setup:** An incoming signal can be sent to different controllers or monitoring units via two separate paths.
- **Prototypes and Test Setups:** Easy duplication of an adapter stream for simulation and debugging purposes without modifying the actual application.

Comparison with Similar Function Blocks

Compared to dedicated **SPLIT FBs** with event/data ports, `AB_SPLIT_2` operates purely on an adapter basis. While classic split function blocks often require triggered data copies, distribution here occurs continuously and without explicit activation. Adapter multiplexers or bus couplers offer similar functionality, but with more complex configurations. This function block is specifically designed for the unidirectional `AB` adapter.

Change Detection

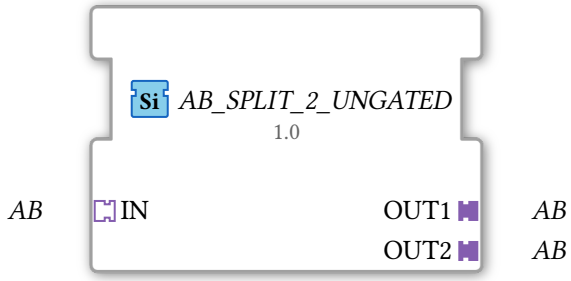
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AB_SPLIT_2` is a simple yet effective generic function block for splitting a unidirectional adapter into two outputs. It increases the flexibility of adapter cabling in industrial control systems according to IEC 61499 and is freely available thanks to the EPL 2.0 license. For applications requiring a 1:2 distribution of adapter data, it offers a clean and maintainable solution.

AB_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_2_UNGATED** splits a unidirectional adapter of type `AB` into two separate adapter outputs. It is implemented as a generic function block (Generic FB) and allows the simultaneous forwarding of the incoming adapter to two independent target addresses. This simplifies the topology in control applications when a signal is needed multiple times without affecting the original data flow.

Interface Structure

Event Inputs

No event inputs are available.

Event Outputs

No event outputs are available. Adapter communication occurs without events via the data exchange mechanism of the adapter type.

Data Inputs

No data inputs are available. Data flows exclusively through the adapter input `IN`.

Data Outputs

No data outputs are available. Outputs are provided via the adapter outputs `OUT1` and `OUT2`.

Adapter

Role	Name	Type	Direction
Socket	IN	`adapter::types::unidirectional::AB`	Input
Plug	OUT1	`adapter::types::unidirectional::AB`	Output
Plug	OUT2	`adapter::types::unidirectional::AB`	Output

Functionality

The function block forwards the adapter connected via socket `IN` unchanged to both plugs `OUT1` and `OUT2`. Any change to the data or state of the incoming adapter is propagated simultaneously to both outputs. The distribution occurs without delay or buffering – it is a pure distribution logic.

Technical Features

- **Generic Function Block:** The function block is declared as a generic type (`GEN_AB_SPLIT`), so it can work with different subtypes of the adapter `AB`, provided they have the same unidirectional semantics.
- **Unidirectionality:** The adapter is designed exclusively in one direction – data flows from the socket to the plugs. Backward communication is not supported.
- **License:** This function block is licensed under the Eclipse Public License 2.0 (EPL-2.0), which permits free use, modification, and distribution in your own projects.
- **No State Machines:** The function block has no internal states or event-driven processes, as adapter routing is entirely data-driven.

State Overview

The function block does not have a state machine. The output adapters always reflect the current state of the input adapter.

Application Scenarios

- **Signal Distribution in Modular Systems:** Separation of a sensor signal from a fieldbus component for parallel processing in two independent control logics.
- **Redundancy Setup:** An incoming signal can be sent to different controllers or monitoring units via two separate paths.
- **Prototypes and Test Setups:** Easy duplication of an adapter stream for simulation and debugging purposes without modifying the actual application.

Comparison with Similar Function Blocks

Compared to dedicated **SPLIT FBs** with event/data ports, `AB_SPLIT_2_UNGATED` operates purely on an adapter basis. While classic split function blocks often require triggered data copies, distribution here occurs continuously and without explicit activation. Adapter multiplexers or bus couplers offer similar functionality, but with more complex configurations. This function block is specifically designed for the unidirectional `AB` adapter.

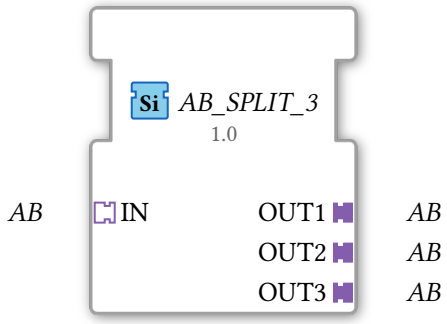
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AB_SPLIT_2_UNGATED` is a simple yet effective generic function block for splitting a unidirectional adapter into two outputs. It increases the flexibility of adapter cabling in industrial control systems according to IEC 61499 and is freely available thanks to the EPL 2.0 license. For applications requiring a 1:2 distribution of adapter data, it offers a clean and maintainable solution.

AB_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_3** is used to split an incoming unidirectional data/signal adapter (type `AB`) into three identical output adapters. It is implemented as a generic function block and represents a simple 1-to-3 distribution without additional logic or delay.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AB`	`IN`	Socket (Input)	Receives a unidirectional AB-type adapter.
`adapter::types::unidirectional::AB`	`OUT1`	Plug (Output)	First output of the incoming signal.
`adapter::types::unidirectional::AB`	`OUT2`	Plug (Output)	Second output of the incoming signal.
`adapter::types::unidirectional::AB`	`OUT3`	Plug (Output)	Third output of the incoming signal.

Functionality

The module forwards the unidirectional adapter connected to socket `IN` **without manipulation** to the three plugs `OUT1`, `OUT2`, and `OUT3` in parallel. No buffering, processing, or timing control takes place. The module functions purely as a **splitter** (distributor) for adapter-based signals.

Since the adapter `AB` is unidirectional, data transmission occurs exclusively in one direction (from the socket to the plugs). The three outputs are independent of each other and always provide the current state of the input.

Technical Features

- **Generic Block**: The function block is marked as generic by the attribute `eclipse4diac::core::GenericClassName('GEN_AB_SPLIT')`. This means the actual adapter type can be specified at runtime or during instantiation, as long as it has the basic structure of a unidirectional AB adapter.
- **Unidirectionality**: Both the input and output adapters are of type `unidirectional::AB`. Reverse communication is not supported.
- **No State Maintenance**: The block has no internal state machine or memory. The routing is passive and continuous.
- **Copyright**: The block was released by HR Agrartechnik GmbH under the Eclipse Public License 2.0.

State Overview

The function block has **no explicit state machine**. The behavior is purely data-flow oriented: As soon as the input adapter provides data, it is immediately propagated to all outputs. No state monitoring or control is required.

Application Scenarios

- **Signal Distribution in Automation Technology**: Splitting a sensor signal (e.g., temperature, pressure) to multiple processing units.
- **Parallelization of Control Sequences**: Providing the same adapter stream to different function blocks that operate independently.
- **Prototyping and Testing**: Using a simulated adapter to supply multiple system components simultaneously.

Comparison with Similar Function Blocks

Function Block	Behavior	Special Feature
`SPLIT_2`	1-to-2 Distribution	Analog function block with two outputs.
`AB_SPLIT_3`	1-to-3 Distribution	This block.
`AB_JOIN`	Combining Multiple Adapters	Counterpart to the splitter.

While `SPLIT_2` offers only two outputs, `AB_SPLIT_3` covers the need for triple distribution. No additional logic functions such as filtering or delay are implemented.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

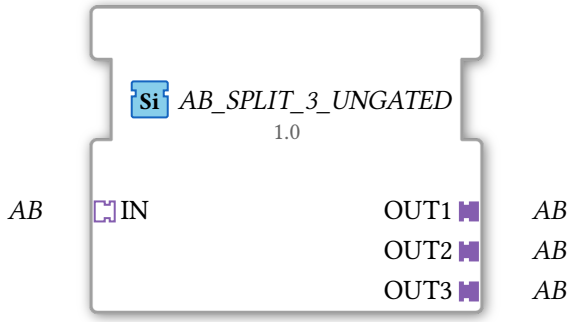
The **AB_SPLIT_3** is a simple yet useful generic splitter block for unidirectional adapters. It enables clean structuring of automation applications by splitting an incoming signal path into three parallel paths. Thanks to its generic nature, it can be used with any AB adapter type and integrates seamlessly into the IEC 61499-2 compliant 4diac development environment.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AB_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_3_UNGATED** is used to split an incoming unidirectional data/signal adapter (type `AB`) into three identical output adapters. It is implemented as a generic function block and represents a simple 1-to-3 distribution without additional logic or delay.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Type	Name	Direction	Description

adapter::types::unidirectional::AB	IN	Socket (Input)	Receives a unidirectional AB-type adapter.
adapter::types::unidirectional::AB	OUT1	Plug (Output)	First output of the incoming signal.
adapter::types::unidirectional::AB	OUT2	Plug (Output)	Second output of the incoming signal.
adapter::types::unidirectional::AB	OUT3	Plug (Output)	Third output of the incoming signal.

Functionality

The module forwards the unidirectional adapter connected to socket `IN` **without manipulation** to the three plugs `OUT1`, `OUT2`, and `OUT3` in parallel. No buffering, processing, or timing control takes place. The module functions purely as a **splitter** (distributor) for adapter-based signals.

Since the adapter `AB` is unidirectional, data transmission occurs exclusively in one direction (from the socket to the plugs). The three outputs are independent of each other and always provide the current state of the input.

Technical Features

- **Generic Block**: The function block is marked as generic by the attribute `eclipse4diac::core::GenericClassName ('GEN_AB_SPLIT')`. This means the actual adapter type can be specified at runtime or during instantiation, as long as it has the basic structure of a unidirectional AB adapter.
- **Unidirectionality**: Both the input and output adapters are of type `unidirectional::AB`. Reverse communication is not supported.
- **No State Maintenance**: The block has no internal state machine or memory. The routing is passive and continuous.
- **Copyright**: The block was released by HR Agrartechnik GmbH under the Eclipse Public License 2.0.

State Overview

The function block has **no explicit state machine**. The behavior is purely data-flow oriented: As soon as the input adapter provides data, it is immediately propagated to all outputs. No state monitoring or control is required.

Application Scenarios

- **Signal Distribution in Automation Technology**: Splitting a sensor signal (e.g., temperature, pressure) to multiple processing units.
- **Parallelization of Control Sequences**: Providing the same adapter stream to different function blocks that operate independently.
- **Prototyping and Testing**: Using a simulated adapter to supply multiple system components simultaneously.

Comparison with Similar Function Blocks

Function Block	Behavior	Special Feature
SPLIT_2	1-to-2 Distribution	Analog function block with two outputs.
AB_SPLIT_3_UNGATED	1-to-3 Distribution	This block.
AB_JOIN	Combining Multiple Adapters	Counterpart to the splitter.

While `SPLIT_2` offers only two outputs, `AB_SPLIT_3_UNGATED` covers the need for triple distribution. No additional logic functions such as filtering or delay are implemented.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

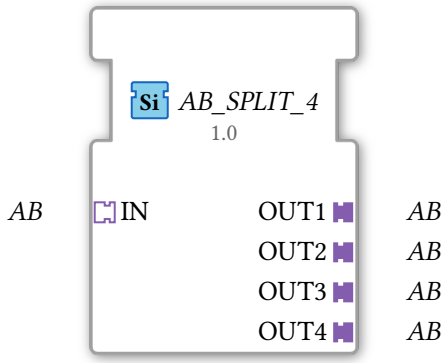
Conclusion

The **AB_SPLIT_3_UNGATED** is a simple yet useful generic splitter block for unidirectional adapters. It enables clean structuring of automation applications by splitting an incoming signal path into three parallel paths. Thanks to its generic nature, it can be used with any AB adapter type and integrates seamlessly into the IEC 61499-2 compliant 4diac development environment.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AB_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_4** is used to distribute a single incoming unidirectional adapter connection of type `AB` to four parallel outgoing adapter connections of the same type. It is designed as a generic block and enables the distribution of a data or signal flow across multiple downstream components.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AB`	Incoming unidirectional adapter connection
Plug	`OUT1`	`adapter::types::unidirectional::AB`	First outgoing adapter connection
Plug	`OUT2`	`adapter::types::unidirectional::AB`	Second outgoing adapter connection
Plug	`OUT3`	`adapter::types::unidirectional::AB`	Third outgoing adapter connection
Plug	`OUT4`	`adapter::types::unidirectional::AB`	Fourth Outgoing Adapter Connection

Functionality

This component functions as a passive splitter at the adapter level. As soon as an adapter connection of type `AB` is established at socket `IN`, the component forwards the data or signals arriving via this connection unchanged to all four plugs `OUT1` to `OUT4`. The distribution occurs without data modification or buffering. The component has no events or data points of its own; all communication takes place exclusively via the adapter interfaces.

Technical Features

- **Generic Design:** The function block is implemented as a generic FB (`GenericClassName = 'GEN_AB_SPLIT'`) and can therefore be used for various specific implementations of the adapter type `AB`.
- **Unidirectionality:** The adapters are declared as unidirectional. This means that data flow occurs only in one direction (from the socket to the plugs). No return channels are provided.
- **No Active Logic:** The function block contains no algorithm or state machine. It functions purely passively as a wiring aid at the architecture level.
- **Simplicity:** The function block reduces the complexity of system wiring by modeling a physical or logical division of an adapter signal.

State Overview

The function block does not have an internal state machine (ECC – Execution Control Chart). It behaves statically and forwards the data present at `IN` to all outputs throughout its entire runtime. There are no interrupt or initialization states.

Application Scenarios

- **Distributing Sensor Values:** A sensor delivers data via a `AB` adapter, which must be forwarded in parallel to several control modules or display units.
- **Test and Diagnostic Environments:** A data stream is sent to a processing unit and simultaneously to a logging or monitoring component.
- **Modular Systems:** Within a larger automation application, the function block can be used to split a common bus signal to several downstream function blocks.

Comparison with Similar Function Blocks

Similar splitter function blocks exist for other adapter types or for data-based forwarding. The `AB_SPLIT_4` is characterized by its specific fit for the adapter type `AB` and its unidirectional design. Unlike function blocks with event or data interfaces, it requires no additional clocking or synchronization – the distribution is implicit in the connection topology.

Change Detection

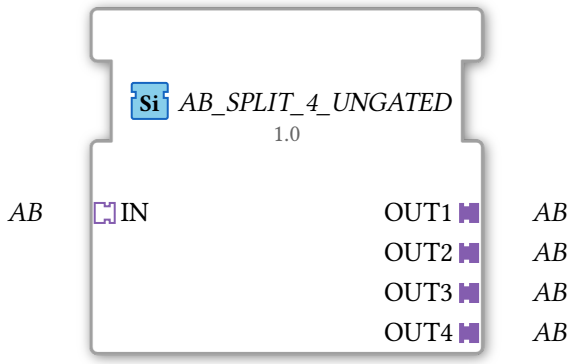
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AB_SPLIT_4` is a simple yet useful generic function block for multiplying a unidirectional adapter connection. It simplifies the structuring of control applications by avoiding redundant wiring and enabling a clear distribution of signal pins. Its generic implementation makes it flexible and usable in various contexts.

AB_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_4_UNGATED** is used to distribute a single incoming unidirectional adapter connection of type `AB` to four parallel outgoing adapter connections of the same type. It is designed as a generic block and enables the distribution of a data or signal flow across multiple downstream components.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AB`	Incoming unidirectional adapter connection
Plug	`OUT1`	`adapter::types::unidirectional::AB`	First outgoing adapter connection
Plug	`OUT2`	`adapter::types::unidirectional::AB`	Second outgoing adapter connection
Plug	`OUT3`	`adapter::types::unidirectional::AB`	Third outgoing adapter connection
Plug	`OUT4`	`adapter::types::unidirectional::AB`	Fourth Outgoing Adapter Connection

Functionality

This component functions as a passive splitter at the adapter level. As soon as an adapter connection of type `AB` is established at socket `IN`, the component forwards the data or signals arriving via this connection unchanged to all four plugs `OUT1` to `OUT4`. The distribution occurs without data modification or buffering. The component has no events or data points of its own; all communication takes place exclusively via the adapter interfaces.

Technical Features

- **Generic Design:** The function block is implemented as a generic FB (`GenericClassName = 'GEN_AB_SPLIT'`) and can therefore be used for various specific implementations of the adapter type `AB`.

- **Unidirectionality:** The adapters are declared as unidirectional. This means that data flow occurs only in one direction (from the socket to the plugs). No return channels are provided.
- **No Active Logic:** The function block contains no algorithm or state machine. It functions purely passively as a wiring aid at the architecture level.
- **Simplicity:** The function block reduces the complexity of system wiring by modeling a physical or logical division of an adapter signal.

State Overview

The function block does not have an internal state machine (ECC – Execution Control Chart). It behaves statically and forwards the data present at `IN` to all outputs throughout its entire runtime. There are no interrupt or initialization states.

Application Scenarios

- **Distributing Sensor Values:** A sensor delivers data via a `AB` adapter, which must be forwarded in parallel to several control modules or display units.
- **Test and Diagnostic Environments:** A data stream is sent to a processing unit and simultaneously to a logging or monitoring component.
- **Modular Systems:** Within a larger automation application, the function block can be used to split a common bus signal to several downstream function blocks.

Comparison with Similar Function Blocks

Similar splitter function blocks exist for other adapter types or for data-based forwarding. The `AB_SPLIT_4_UNGATED` is characterized by its specific fit for the adapter type `AB` and its unidirectional design. Unlike function blocks with event or data interfaces, it requires no additional clocking or synchronization – the distribution is implicit in the connection topology.

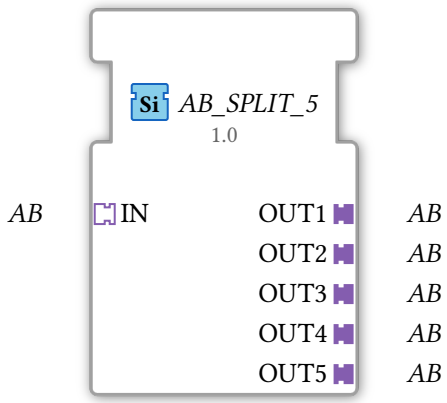
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AB_SPLIT_4_UNGATED` is a simple yet useful generic function block for multiplying a unidirectional adapter connection. It simplifies the structuring of control applications by avoiding redundant wiring and enabling a clear distribution of signal pins. Its generic implementation makes it flexible and usable in various contexts.

AB_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AB_SPLIT_5` is used to distribute one unidirectional AB adapter input (socket) to five identical AB adapter outputs (plugs). It is a generic function block designed for signal distribution in automation systems.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None. Data transmission occurs exclusively via the adapter interfaces.

Data Outputs
None.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AB`	Unidirectional AB adapter providing the input signal.
Plug (Output)	`OUT1`	`adapter::types::unidirectional::AB`	First unidirectional AB adapter output.
Plug (Output)	`OUT2`	`adapter::types::unidirectional::AB`	Second unidirectional AB adapter output.
Plug (Output)	`OUT3`	`adapter::types::unidirectional::AB`	Third unidirectional AB adapter output.
Plug (Output)	`OUT4`	`adapter::types::unidirectional::AB`	Fourth unidirectional AB adapter output.
Plug (Output)	`OUT5`	`adapter::types::unidirectional::AB`	Fifth unidirectional AB adapter output.

Functionality
The module receives a unidirectional AB signal via socket `IN`. This signal is forwarded without modification to all five plugs (`OUT1` to `OUT5`). This creates five identical copies of the input signal, which can be connected in parallel to various subsequent components or systems.

Since the component has no event or data inputs, all communication takes place exclusively via the adapter interfaces. Distribution is static and without delay or buffering.

Technical Features
- **Generic Component**: `AB_SPLIT_5` uses the generic class name `GEN_AB_SPLIT` and enables type-safe instantiation in the 4diac IDE.
- **Unidirectional**: All participating adapters are of type `unidirectional::AB`, meaning data flows only in one direction (from the socket to the plugs).
- **No State Logic**: The function block has no internal state diagram or event processing – it is purely structural and does not perform any data transformation.

State Overview
There is no state machine. The function block operates continuously and permanently passes the received signal to all five outputs.

Application Scenarios
- **Signal Distribution in Fieldbus Systems**: When a sensor value or control signal (as an AB adapter) needs to be sent to multiple actuators or controllers.
- **Multiplexing in Modular Machines**: Distribution of a central AB signal to different modules of a system.

- **Testing and Simulation**: Simultaneously connecting multiple monitoring or logging units to a data source.

Comparison with Similar Function Blocks

Function Block	Number of Outputs	Special Features
`AB_SPLIT_2`	2	Simple 1:2 distribution
`AB_SPLIT_3`	3	1:3 distribution
`AB_SPLIT_5`	5	1:5 distribution (this block)
`AB_SPLIT_N` (generic)	configurable	Individual number via GenericClassName

Compared to a configurable split block, `AB_SPLIT_5` offers a fixed, yet simple and type-safe solution without additional configuration effort.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AB_SPLIT_5` is a lean, specialized function block for the 1:5 distribution of unidirectional AB adapter signals. Thanks to its generic implementation and clear interface structure, it is ideally suited for the modular design of automation systems where a signal is required multiple times.

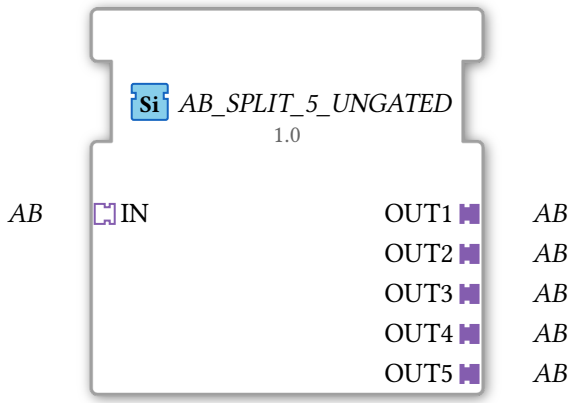
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AB_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AB_SPLIT_5_UNGATED` is used to distribute one unidirectional AB adapter input (socket) to five identical AB adapter outputs (plugs). It is a generic function block designed for signal distribution in automation systems.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None. Data transmission occurs exclusively via the adapter interfaces.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AB`	Unidirectional AB adapter providing the input signal.
Plug (Output)	`OUT1`	`adapter::types::unidirectional::AB`	First unidirectional AB adapter output.
Plug (Output)	`OUT2`	`adapter::types::unidirectional::AB`	Second unidirectional AB adapter output.
Plug (Output)	`OUT3`	`adapter::types::unidirectional::AB`	Third unidirectional AB adapter output.
Plug (Output)	`OUT4`	`adapter::types::unidirectional::AB`	Fourth unidirectional AB adapter output.
Plug (Output)	`OUT5`	`adapter::types::unidirectional::AB`	Fifth unidirectional AB adapter output.

Functionality

The module receives a unidirectional AB signal via socket `IN`. This signal is forwarded without modification to all five plugs (`OUT1` to `OUT5`). This creates five identical copies of the input signal, which can be connected in parallel to various subsequent components or systems.

Since the component has no event or data inputs, all communication takes place exclusively via the adapter interfaces. Distribution is static and without delay or buffering.

Technical Features

- **Generic Component**: `AB_SPLIT_5_UNGATED` uses the generic class name `GEN_AB_SPLIT` and enables type-safe instantiation in the 4diac IDE.
- **Unidirectional**: All participating adapters are of type `unidirectional::AB`, meaning data flows only in one direction (from the socket to the plugs).
- **No State Logic**: The function block has no internal state diagram or event processing – it is purely structural and does not perform any data transformation.

State Overview

There is no state machine. The function block operates continuously and permanently passes the received signal to all five outputs.

Application Scenarios

- **Signal Distribution in Fieldbus Systems**: When a sensor value or control signal (as an AB adapter) needs to be sent to multiple actuators or controllers.
- **Multiplexing in Modular Machines**: Distribution of a central AB signal to different modules of a system.
- **Testing and Simulation**: Simultaneously connecting multiple monitoring or logging units to a data source.

Comparison with Similar Function Blocks

Function Block	Number of Outputs	Special Features
`AB_SPLIT_2`	2	Simple 1:2 distribution
`AB_SPLIT_3`	3	1:3 distribution
`AB_SPLIT_5_UNGATED`	5	1:5 distribution (this block)
`AB_SPLIT_N` (generic)	configurable	Individual number via GenericClassName

Compared to a configurable split block, `AB_SPLIT_5_UNGATED` offers a fixed, yet simple and type-safe solution without additional configuration effort.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

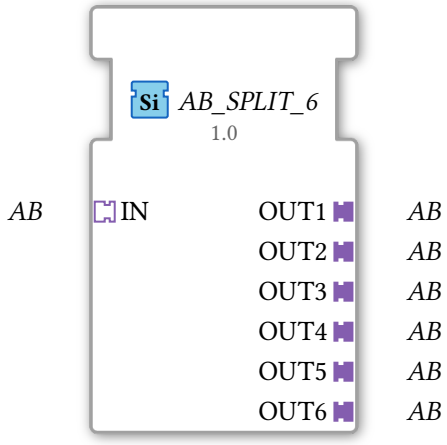
AB_SPLIT_5_UNGATED` is a lean, specialized function block for the 1:5 distribution of unidirectional AB adapter signals. Thanks to its generic implementation and clear interface structure, it is ideally suited for the modular design of automation systems where a signal is required multiple times.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AB_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_6** is a generic adapter splitter. It serves to distribute an incoming unidirectional adapter data stream (type `AB`) simultaneously to six separate outputs. This allows the data from a single adapter to be made available to multiple downstream blocks without data loss.

Interface Structure

Event Inputs
The block has no event inputs. Data is passed exclusively via the adapter interface.

Event Outputs
There are no event outputs. Output is provided solely via the adapters.

Data Inputs
The block has no separate data inputs. The actual user data is transported via the adapter input `IN`.

Data Outputs
There are no explicit data outputs. The distributed data is provided via the adapter outputs `OUT1` to `OUT6`.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AB`	`IN`	Socket	Input adapter that supplies the data to be distributed.
`adapter::types::unidirectional::AB`	`OUT1`	Plug	First output adapter – receives a copy of the input data.
adapter::types::unidirectional::AB`	`OUT2`	Plug	Second output adapter.
adapter::types::unidirectional::AB`	`OUT3`	Plug	Third output adapter.
adapter::types::unidirectional::AB`	`OUT4`	Plug	Fourth output adapter.
adapter::types::unidirectional::AB`	`OUT5`	Plug	Fifth output adapter.
adapter::types::unidirectional::AB`	`OUT6`	Plug	Sixth output adapter.

Functionality
The function block reads the data arriving from adapter `IN` and forwards it identically to all six output adapters `OUT1` to `OUT6`. This is a pure 1:6 distribution without buffering or data processing. As soon as data is present at the input, it is immediately and simultaneously transferred to all outputs. The number of outputs is fixed at six.

Technical Features
- **Generic Function Block**: The function block is implemented as a generic type (`GEN_AB_SPLIT`) and can be used multiple times within the 4diac IDE framework.
- **Unidirectional Interface**: All adapters are of type `unidirectional::AB`, meaning that data flow is only in one direction (from the input to the outputs).
- **No State Dependency**: Since neither events nor trigger mechanisms exist, the function block operates continuously and without an internal state.
- **License**: The function block is licensed under the Eclipse Public License 2.0 (EPL-2.0).

State Overview
The function block has no internal state machine or discrete states. Its operation is purely continuous – there is no initialization, no error states, and no time dependencies.

Application Scenarios
- **Data Distribution in Control Applications**: When a sensor or data source (e.g., an IO-Link master via a `AB` adapter) needs to provide its measured values to several function blocks operating in parallel.
- **Redundant Processing**: An input signal is simultaneously forwarded to multiple independent calculation or monitoring logics.
- **Prototyping**: Easily duplicate an adapter signal during the development phase without having to write your own splitter logic.

Comparison with similar components

- **AB_SPLIT_2 / AB_SPLIT_4**: Analogous design, but with two or four outputs, respectively. This component offers the maximum of six outputs.
- **Generic Splits with Events**: Some splitters operate in an event-driven manner (e.g., `E_SPLIT`), but then require additional triggers. In contrast, `AB_SPLIT_6` operates continuously and without events via the adapter interface.
- **Data Splitters (e.g., `F_SPLIT`)**: These split individual data values (e.g., an array), while `AB_SPLIT_6` copies the entire adapter data stream unchanged.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

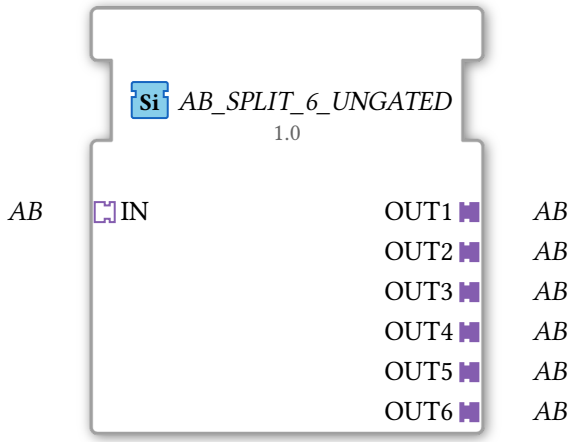
The `AB_SPLIT_6` is a simple yet useful generic function block for multiplying a unidirectional adapter signal to up to six outputs. It is easy to understand, requires no configuration, and is ideally suited for the rapid distribution of adapter data in industrial control applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AB_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_6_UNGATED** is a generic adapter splitter. It serves to distribute an incoming unidirectional adapter data stream (type `AB`) simultaneously to six separate outputs. This allows the data from a single adapter to be made available to multiple downstream blocks without data loss.

Interface Structure

Event Inputs

The block has no event inputs. Data is passed exclusively via the adapter interface.

Event Outputs

There are no event outputs. Output is provided solely via the adapters.

Data Inputs

The block has no separate data inputs. The actual user data is transported via the adapter input `IN`.

Data Outputs

There are no explicit data outputs. The distributed data is provided via the adapter outputs `OUT1` to `OUT6`.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AB`	`IN`	Socket	Input adapter that supplies the data to be distributed.
`adapter::types::unidirectional::AB`	`OUT1`	Plug	First output adapter – receives a copy of the input data.
adapter::types::unidirectional::AB`	`OUT2`	Plug	Second output adapter.
adapter::types::unidirectional::AB`	`OUT3`	Plug	Third output adapter.
adapter::types::unidirectional::AB`	`OUT4`	Plug	Fourth output adapter.
adapter::types::unidirectional::AB`	`OUT5`	Plug	Fifth output adapter.
adapter::types::unidirectional::AB`	`OUT6`	Plug	Sixth output adapter.

Functionality

The function block reads the data arriving from adapter `IN` and forwards it identically to all six output adapters `OUT1` to `OUT6`. This is a pure 1:6 distribution without buffering or data processing. As soon as data is present at the input, it is immediately and simultaneously transferred to all outputs. The number of outputs is fixed at six.

Technical Features

- **Generic Function Block**: The function block is implemented as a generic type (`GEN_AB_SPLIT`) and can be used multiple times within the 4diac IDE framework.
- **Unidirectional Interface**: All adapters are of type `unidirectional::AB`, meaning that data flow is only in one direction (from the input to the outputs).
- **No State Dependency**: Since neither events nor trigger mechanisms exist, the function block operates continuously and without an internal state.
- **License**: The function block is licensed under the Eclipse Public License 2.0 (EPL-2.0).

State Overview

The function block has no internal state machine or discrete states. Its operation is purely continuous – there is no initialization, no error states, and no time dependencies.

Application Scenarios

- **Data Distribution in Control Applications**: When a sensor or data source (e.g., an IO-Link master via a `AB` adapter) needs to provide its measured values to several function blocks operating in parallel.
- **Redundant Processing**: An input signal is simultaneously forwarded to multiple independent calculation or monitoring logics.
- **Prototyping**: Easily duplicate an adapter signal during the development phase without having to write your own splitter logic.

Comparison with similar components

- **AB_SPLIT_2 / AB_SPLIT_4**: Analogous design, but with two or four outputs, respectively. This component offers the maximum of six outputs.
- **Generic Splits with Events**: Some splitters operate in an event-driven manner (e.g., `E_SPLIT`), but then require additional triggers. In contrast, `AB_SPLIT_6_UNGATED` operates continuously and without events via the adapter interface.

- **Data Splitters (e.g., `F_SPLIT`)**: These split individual data values (e.g., an array), while `AB_SPLIT_6_UNGATED` copies the entire adapter data stream unchanged.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

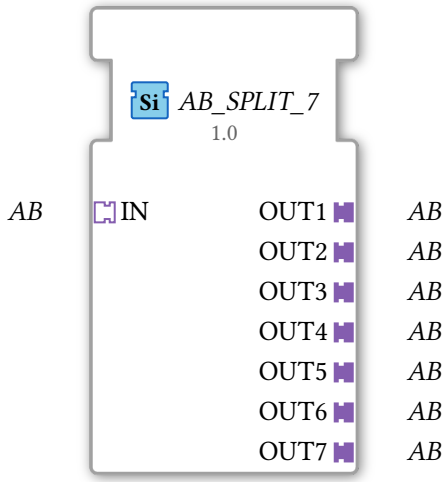
Conclusion

The `AB_SPLIT_6_UNGATED` is a simple yet useful generic function block for multiplying a unidirectional adapter signal to up to six outputs. It is easy to understand, requires no configuration, and is ideally suited for the rapid distribution of adapter data in industrial control applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AB_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_7** is used to split a single incoming adapter of type `adapter::types::unidirectional::AB` into seven identical output adapters. It is implemented as a generic block (`GEN_AB_SPLIT`) and can therefore be instantiated for various specific adapter types that conform to the unidirectional AB protocol. It was developed under the Eclipse Public License 2.0.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapters

Name	Type	Direction	Description

IN	`adapter::types::unidirectional::AB`	Socket (Input)	Incoming adapter that is distributed to the outputs.
OUT1	`adapter::types::unidirectional::AB`	Plug (Output)	First outgoing adapter, identical to the input.
OUT2	`adapter::types::unidirectional::AB`	Plug (Output)	Second outgoing adapter.
OUT3	`adapter::types::unidirectional::AB`	Plug (Output)	Third outgoing adapter.
OUT4	`adapter::types::unidirectional::AB`	Plug (Output)	Fourth outgoing adapter.
OUT5	`adapter::types::unidirectional::AB`	Plug (Output)	Fifth outgoing adapter.
OUT6	`adapter::types::unidirectional::AB`	Plug (Output)	Sixth outgoing adapter.
OUT7	`adapter::types::unidirectional::AB`	Plug (Output)	Seventh outgoing adapter.

Functionality

This component functions purely as a signal distributor for adapter connections. Every data or event flow arriving via socket `IN` is forwarded unchanged and in parallel to all seven output plugs (`OUT1` – `OUT7`). No processing, buffering, or state changes take place. Therefore, this component should be understood as a passive "splitter."

Since these are unidirectional adapters, communication only occurs in one direction – from the socket to the plugs. The outputs are logically identical and can be connected independently to subsequent function blocks.

Technical Features

- **Generic Type**: The function block is declared as a generic FB (GenericClassName `'GEN_AB_SPLIT'`). This allows reuse for different implementations of the unidirectional AB adapter without having to redefine the interface.
- **No Internal Logic**: There are no event triggers, data operations, or state machines. The FB is completely defined by its adapter wiring.
- **Unidirectionality**: The adapter type `unidirectional::AB` supports only one direction of data flow. There are no return channels.

State Overview

The function block does not have an internal state machine (ECC). It is static and does not execute any sequential logic. There are no distinguishable operating states.

Application Scenarios

- **Distributing control information** to multiple parallel consumers (e.g., controlling several identical actuators).
- **Splitting a feedback signal** for separate evaluation in different branches of an automation application.
- **Generating multiple identical adapter strings** for redundancy or separate monitoring.

Due to its generic design, the function block can be used in various contexts once the specific adapter type is defined.

Comparison with Similar Function Blocks

- **AB_MERGE**: Merges multiple incoming adapters into one output – the opposite function to AB_SPLIT_7.
- **AB_SPLIT_N**: Generalization with a configurable number of outputs, if available. AB_SPLIT_7 is a fixed 1:7 split.
- **Direct wiring without a splitter**: If multiple identical copies are needed, the source function block would have to be instantiated multiple times, which increases the design complexity. The splitter reduces redundancy at the adapter connection level.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AB_SPLIT_7** is a simple yet useful function block for multiplying a unidirectional adapter connection. Its generic nature and clear separation of inputs and outputs make it a reusable tool in the 4diac IDE, especially in scenarios requiring parallel signal distribution. It contributes to the modularity and clarity of industrial control applications.

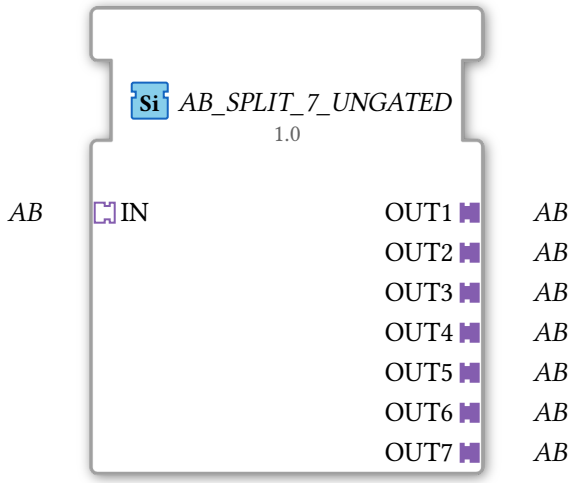
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AB_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_7_UNGATED** is used to split a single incoming adapter of type `adapter::types::unidirectional::AB` into seven identical output adapters. It is implemented as a generic block (`GEN_AB_SPLIT`) and can therefore be instantiated for various specific adapter types that conform to the unidirectional AB protocol. It was developed under the Eclipse Public License 2.0.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapters

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::AB`	Socket (Input)	Incoming adapter that is distributed to the outputs.
OUT1	`adapter::types::unidirectional::AB`	Plug (Output)	First outgoing adapter, identical to the input.
OUT2	`adapter::types::unidirectional::AB`	Plug (Output)	Second outgoing adapter.
OUT3	`adapter::types::unidirectional::AB`	Plug (Output)	Third outgoing adapter.
OUT4	`adapter::types::unidirectional::AB`	Plug (Output)	Fourth outgoing adapter.
OUT5	`adapter::types::unidirectional::AB`	Plug (Output)	Fifth outgoing adapter.
OUT6	`adapter::types::unidirectional::AB`	Plug (Output)	Sixth outgoing adapter.
OUT7	`adapter::types::unidirectional::AB`	Plug (Output)	Seventh outgoing adapter.

Functionality

This component functions purely as a signal distributor for adapter connections. Every data or event flow arriving via socket `IN` is forwarded unchanged and in parallel to all seven output plugs (`OUT1` – `OUT7`). No processing, buffering, or state changes take place. Therefore, this component should be understood as a passive "splitter."

Since these are unidirectional adapters, communication only occurs in one direction – from the socket to the plugs. The outputs are logically identical and can be connected independently to subsequent function blocks.

Technical Features

- **Generic Type**: The function block is declared as a generic FB (GenericClassName `'GEN_AB_SPLIT'`). This allows reuse for different implementations of the unidirectional AB adapter without having to redefine the interface.
- **No Internal Logic**: There are no event triggers, data operations, or state machines. The FB is completely defined by its adapter wiring.
- **Unidirectionality**: The adapter type `unidirectional::AB` supports only one direction of data flow. There are no return channels.

State Overview

The function block does not have an internal state machine (ECC). It is static and does not execute any sequential logic. There are no distinguishable operating states.

Application Scenarios

- **Distributing control information** to multiple parallel consumers (e.g., controlling several identical actuators).
- **Splitting a feedback signal** for separate evaluation in different branches of an automation application.
- **Generating multiple identical adapter strings** for redundancy or separate monitoring.

Due to its generic design, the function block can be used in various contexts once the specific adapter type is defined.

Comparison with Similar Function Blocks

- **AB_MERGE**: Merges multiple incoming adapters into one output – the opposite function to AB_SPLIT_7_UNGATED.

- **AB_SPLIT_N**: Generalization with a configurable number of outputs, if available. AB_SPLIT_7_UNGATED is a fixed 1:7 split.
- **Direct wiring without a splitter**: If multiple identical copies are needed, the source function block would have to be instantiated multiple times, which increases the design complexity. The splitter reduces redundancy at the adapter connection level.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

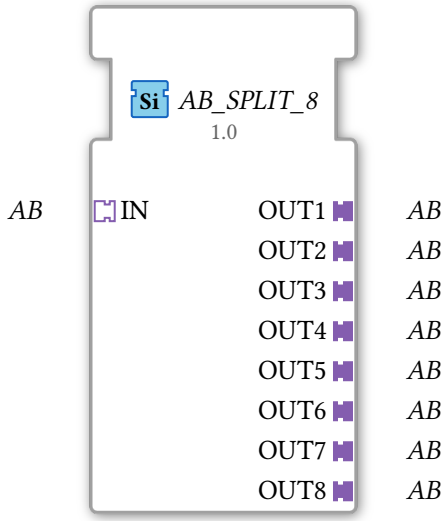
The **AB_SPLIT_7_UNGATED** is a simple yet useful function block for multiplying a unidirectional adapter connection. Its generic nature and clear separation of inputs and outputs make it a reusable tool in the 4diac IDE, especially in scenarios requiring parallel signal distribution. It contributes to the modularity and clarity of industrial control applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AB_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_8** is used to multiply an incoming adapter signal of type `AB` into eight identical outputs. It functions as a passive splitter for unidirectional data exchange via the standardized interface `adapter::types::unidirectional::AB`. The block is designed as a generic function block and is typically used in automation technology to forward an AB signal in parallel to multiple downstream components.

Interface Structure

Event Inputs

None. The function block has no event-driven inputs; data is passed solely via the adapter interface.

Event Outputs

None. No events are explicitly output.

Data Inputs

No direct data inputs. All incoming information is transmitted via the socket adapter.

Data Outputs

No direct data outputs. Outgoing information is provided via the plug adapters.

Adapters

Name	Type	Direction	Description
IN	adapter::types::unidirectional::AB	Socket	Input adapter that receives the signal to be distributed.
OUT1	adapter::types::unidirectional::AB	Plug	First output adapter (identical to the input signal).
OUT2	adapter::types::unidirectional::AB	Plug	Second output adapter.
OUT3`	adapter::types::unidirectional::AB	Plug	Third output adapter.
OUT4`	adapter::types::unidirectional::AB	Plug	Fourth output adapter.
OUT5`	adapter::types::unidirectional::AB	Plug	Fifth output adapter.
OUT6`	adapter::types::unidirectional::AB	Plug	Sixth output adapter.
OUT7`	adapter::types::unidirectional::AB	Plug	Seventh output adapter.
OUT8`	adapter::types::unidirectional::AB	Plug	Eighth Output Adapter.

Functionality

The **AB_SPLIT_8** functions purely as a signal distributor without internal logic or state machine. The AB signal arriving via socket `IN` is forwarded identically and simultaneously to all eight plug outputs `OUT1` to `OUT8`. No buffering or delay occurs. The function block is completely passive – it reacts solely to data changes from the connected adapter and forwards them unchanged.

The distribution is independent of the internal structure of the AB adapter (e.g., data and event connections). The function block itself has no execution state and does not require sequential control.

Technical Features

- **Generic Type**: The function block (FB) is declared as a generic block (`GenericClassName = 'GEN_AB_SPLIT'`), enabling flexible use in various projects.
- **No State Machine**: There is no Execution Control Chart (ECC); the FB is fully data flow controlled.
- **Adapter Type**: The adapter used, `adapter::types::unidirectional::AB`, is standardized and provides a uniform interface.
- **License**: The block is licensed under the **Eclipse Public License 2.0 (EPL-2.0)** and is designed for the **4diac IDE**.

State Overview

The **AB_SPLIT_8** has no internal states or state machines. Signal transmission occurs without delay and without dependence on sequential logic.

Application Scenarios

- **Distributing a single AB sensor signal** to multiple actuators or controllers.
- **Parallel connection of monitoring units** in agricultural technology (e.g., distributing an acknowledge signal).
- **Extension of control cabinet wiring** through software-based signal multiplication.
- **Test environments** where an AB signal is to be sent simultaneously to multiple analysis function blocks (FBs).

Comparison with similar function blocks

- **`AB_SPLIT_2`, `AB_SPLIT_4`**: These function blocks distribute the signal to two or four outputs, respectively. The `AB_SPLIT_8` offers the maximum of eight outputs in a single FB.

- **`DATA_SPLIT`**: Distributes arbitrary data types (e.g., `INT`, `BOOL`). The `AB_SPLIT_8` function block is specifically designed for the `AB` adapter type and retains the complete adapter structure.
- **`EC_SPLIT`**: An event splitter that only multiplies events, but not data. In contrast, `AB_SPLIT_8` operates at the adapter level, including all data and events it contains.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AB_SPLIT_8** function block is a simple yet effective block for multiplying a unidirectional AB adapter signal. Its generic design and passive operation make it ideal for parallelizing control and monitoring signals in automation technology. The absence of state logic ensures low runtime complexity and high reliability. This function block is a valuable addition to any 4diac library that relies on adapter communication.

The **AB_SPLIT_8** function block is a simple yet effective block for multiplying a unidirectional AB adapter signal. —

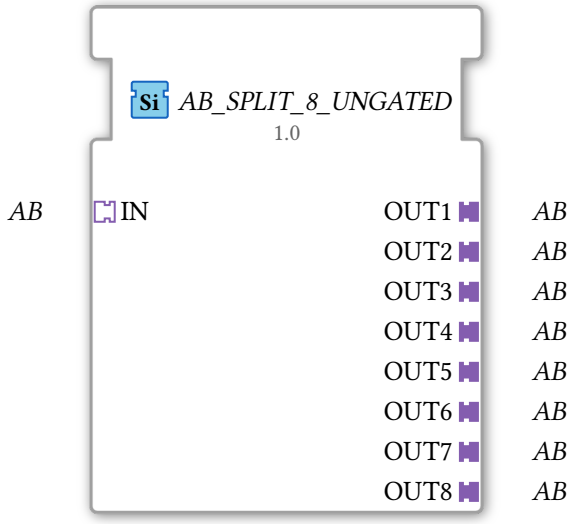
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

]

AB_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_8_UNGATED** is used to multiply an incoming adapter signal of type `AB` into eight identical outputs. It functions as a passive splitter for unidirectional data exchange via the standardized interface `adapter::types::unidirectional::AB`. The block is designed as a generic

function block and is typically used in automation technology to forward an AB signal in parallel to multiple downstream components.

Interface Structure

Event Inputs
None. The function block has no event-driven inputs; data is passed solely via the adapter interface.

Event Outputs
None. No events are explicitly output.

Data Inputs
No direct data inputs. All incoming information is transmitted via the socket adapter.

Data Outputs
No direct data outputs. Outgoing information is provided via the plug adapters.

Adapters

Name	Type	Direction	Description
`IN`	`adapter::types::unidirectional::AB`	Socket	Input adapter that receives the signal to be distributed.
`OUT1`	`adapter::types::unidirectional::AB`	Plug	First output adapter (identical to the input signal).
`OUT2`	`adapter::types::unidirectional::AB`	Plug	Second output adapter.
OUT3`	`adapter::types::unidirectional::AB`	Plug	Third output adapter.
OUT4`	`adapter::types::unidirectional::AB`	Plug	Fourth output adapter.
OUT5`	`adapter::types::unidirectional::AB`	Plug	Fifth output adapter.
OUT6`	`adapter::types::unidirectional::AB`	Plug	Sixth output adapter.
OUT7`	`adapter::types::unidirectional::AB`	Plug	Seventh output adapter.
OUT8`	`adapter::types::unidirectional::AB`	Plug	Eighth Output Adapter.

Functionality
The **AB_SPLIT_8_UNGATED** functions purely as a signal distributor without internal logic or state machine. The AB signal arriving via socket `IN` is forwarded identically and simultaneously to all eight plug outputs `OUT1` to `OUT8`. No buffering or delay occurs. The function block is completely passive – it reacts solely to data changes from the connected adapter and forwards them unchanged.

The distribution is independent of the internal structure of the AB adapter (e.g., data and event connections). The function block itself has no execution state and does not require sequential control.

Technical Features
- **Generic Type**: The function block (FB) is declared as a generic block (`GenericClassName = 'GEN_AB_SPLIT'`), enabling flexible use in various projects.
- **No State Machine**: There is no Execution Control Chart (ECC); the FB is fully data flow controlled.
- **Adapter Type**: The adapter used, `adapter::types::unidirectional::AB`, is standardized and provides a uniform interface.
- **License**: The block is licensed under the **Eclipse Public License 2.0 (EPL-2.0)** and is designed for the **4diac IDE**.

State Overview
The **AB_SPLIT_8_UNGATED** has no internal states or state machines. Signal transmission occurs without delay and without dependence on sequential logic.

Application Scenarios

- **Distributing a single AB sensor signal** to multiple actuators or controllers.
- **Parallel connection of monitoring units** in agricultural technology (e.g., distributing an acknowledge signal).
- **Extension of control cabinet wiring** through software-based signal multiplication.
- **Test environments** where an AB signal is to be sent simultaneously to multiple analysis function blocks (FBs).

Comparison with similar function blocks

- **`AB_SPLIT_2`, `AB_SPLIT_4`**: These function blocks distribute the signal to two or four outputs, respectively. The `AB_SPLIT_8_UNGATED` offers the maximum of eight outputs in a single FB.
- **`DATA_SPLIT`**: Distributes arbitrary data types (e.g., `INT`, `BOOL`). The `AB_SPLIT_8_UNGATED` function block is specifically designed for the `AB` adapter type and retains the complete adapter structure.
- **`EC_SPLIT`**: An event splitter that only multiplies events, but not data. In contrast, `AB_SPLIT_8_UNGATED` operates at the adapter level, including all data and events it contains.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AB_SPLIT_8_UNGATED** function block is a simple yet effective block for multiplying a unidirectional AB adapter signal. Its generic design and passive operation make it ideal for parallelizing control and monitoring signals in automation technology. The absence of state logic ensures low runtime complexity and high reliability. This function block is a valuable addition to any 4diac library that relies on adapter communication.

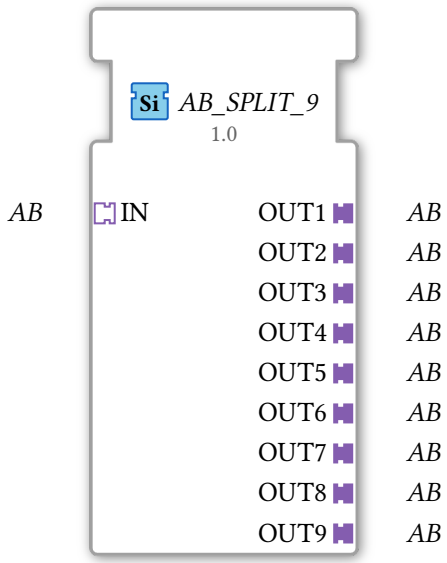
The **AB_SPLIT_8_UNGATED** function block is a simple yet effective block for multiplying a unidirectional AB adapter signal. —

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

]

AB_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_9** is used to split a single incoming adapter of type **AB** into nine identical outgoing adapters (**OUT1** to **OUT9**). It is implemented as a generic block (GenericClassName = `'GEN_AB_SPLIT'`) and can be used in any 4diac project to establish a point-to-multipoint connection via the adapter interface. The block has no event-driven or data-driven inputs/outputs – all communication takes place exclusively via the adapter interfaces. This enables clean, task-oriented coupling of modules.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapters

Interface	Direction	Type	Description
IN	Socket	`adapter::types::unidirectional::AB`	Incoming adapter (source)
OUT1 – OUT9	Plug	`adapter::types::unidirectional::AB`	Nine identical outgoing adapters (sinks)

Functionality

This function block copies all data and event traffic from the incoming adapter **IN** to the nine output adapters **OUT1** to **OUT9**. Each output receives exactly the same information as the input – no filtering, transformation, or delay takes place.

The function block thus behaves like a **passive distributor** (splitting node) at the adapter level. Changes to the input adapter (e.g., new values or events) are immediately propagated to all outputs, provided the connected modules allow this.

Technical Features

- **Generic Type**: The function block is declared as a generic FB (`GenericClassName = 'GEN_AB_SPLIT'`). This allows it to be reused in different contexts, as long as the adapter type `unidirectional::AB` is compatible.
- **No State Logic**: There is no ECC (Execution Control Chart) – the function block is completely declarative and does not require sequential processing.
- **Easy Extensibility**: The principle can be applied to other numbers of outputs (e.g., `AB_SPLIT_4`, `AB_SPLIT_16`) by adapting the XML structure accordingly.

State Overview

The function block has **no** states of its own. Its behavior is purely combinatorial – distribution occurs continuously and without delay. Therefore, a state machine is not required.

Application Scenarios

- **Control of Multiple Parallel Actuators**: A sensor (e.g., a pressure sensor) delivers its measured value via an AB adapter. This value is to be simultaneously transmitted to several actuators (valves, motors) – the split function block distributes the information to all of them.
- **Signal Coupling in Modular Systems**: In a distributed automation solution, multiple subsystems can receive the same data stream without each subsystem having to establish a separate connection to the source.
- **Test and Simulation Environments**: An incoming data stream can be split across multiple monitoring or logging modules without affecting the original connection.

Comparison with Similar Components

Component	Outputs	Special Feature
AB_SPLIT_9	9	Standard split for 1⟶9, generic
AB_SPLIT_4	4	Same functionality, fewer outputs
AB_MERGE	–	Combines multiple inputs into one output (opposite direction)

While AB_SPLIT_9 distributes a single source to multiple sinks, AB_MERGE combines several sources into a single output. Therefore, AB_SPLIT_9 is the ideal choice when a signal needs to be routed to many consumers.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AB_SPLIT_9** offers a simple yet powerful way to split a single adapter-based data stream into nine parallel paths. Its generic nature allows it to be used in any 4diac project, requires no state programming, and enables a clean, modular architecture. It is a fundamental building block for the serial or parallel distribution of adapter signals in automation technology.

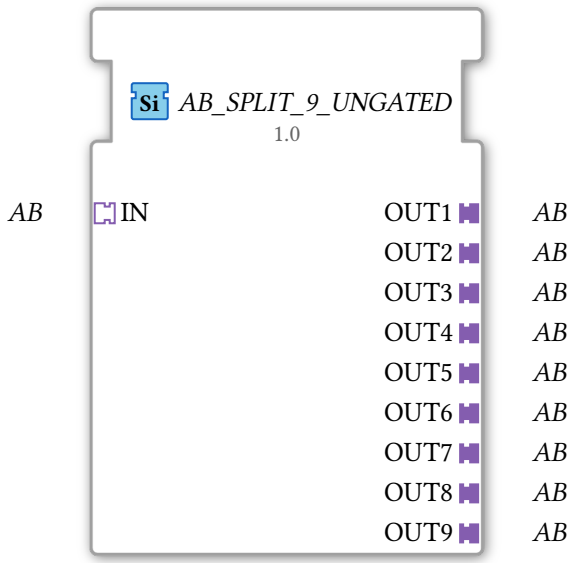
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AB_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_SPLIT_9. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_SPLIT_9_UNGATED** is used to split a single incoming adapter of type **AB** into nine identical outgoing adapters (**OUT1** to **OUT9**). It is implemented as a generic block (GenericClassName = `'GEN_AB_SPLIT'`) and can be used in any 4diac project to establish a point-to-multipoint connection via the adapter interface. The block has no event-driven or data-driven inputs/outputs – all communication takes place exclusively via the adapter interfaces. This enables clean, task-oriented coupling of modules.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapters

Interface	Direction	Type	Description
IN	Socket	`adapter::types::unidirectional::AB`	Incoming adapter (source)
OUT1 – OUT9	Plug	`adapter::types::unidirectional::AB`	Nine identical outgoing adapters (sinks)

Functionality

This function block copies all data and event traffic from the incoming adapter **IN** to the nine output adapters **OUT1** to **OUT9**. Each output receives exactly the same information as the input – no filtering, transformation, or delay takes place.

The function block thus behaves like a **passive distributor** (splitting node) at the adapter level. Changes to the input adapter (e.g., new values or events) are immediately propagated to all outputs, provided the connected modules allow this.

Technical Features

- **Generic Type**: The function block is declared as a generic FB (`GenericClassName = 'GEN_AB_SPLIT'`). This allows it to be reused in different contexts, as long as the adapter type `unidirectional::AB` is compatible.
- **No State Logic**: There is no ECC (Execution Control Chart) – the function block is completely declarative and does not require sequential processing.
- **Easy Extensibility**: The principle can be applied to other numbers of outputs (e.g., `AB_SPLIT_4`, `AB_SPLIT_16`) by adapting the XML structure accordingly.

State Overview

The function block has **no** states of its own. Its behavior is purely combinatorial – distribution occurs continuously and without delay. Therefore, a state machine is not required.

Application Scenarios

- **Control of Multiple Parallel Actuators**: A sensor (e.g., a pressure sensor) delivers its measured value via an AB adapter. This value is to be simultaneously transmitted to several actuators (valves, motors) – the split function block distributes the information to all of them.
- **Signal Coupling in Modular Systems**: In a distributed automation solution, multiple subsystems can receive the same data stream without each subsystem having to establish a separate connection to the source.

- **Test and Simulation Environments**: An incoming data stream can be split across multiple monitoring or logging modules without affecting the original connection.

Comparison with Similar Components

Component	Outputs	Special Feature
`AB_SPLIT_9_UNGATED`	9	Standard split for 1⟶9, generic
`AB_SPLIT_4`	4	Same functionality, fewer outputs
`AB_MERGE`	–	Combines multiple inputs into one output (opposite direction)

While `AB_SPLIT_9_UNGATED` distributes a single source to multiple sinks, `AB_MERGE` combines several sources into a single output. Therefore, `AB_SPLIT_9_UNGATED` is the ideal choice when a signal needs to be routed to many consumers.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AB_SPLIT_9_UNGATED** offers a simple yet powerful way to split a single adapter-based data stream into nine parallel paths. Its generic nature allows it to be used in any 4diac project, requires no state programming, and enables a clean, modular architecture. It is a fundamental building block for the serial or parallel distribution of adapter signals in automation technology.

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

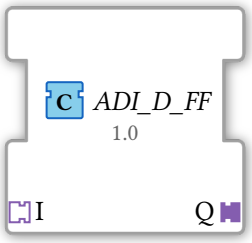
]

Numeric Index (unidirectional)

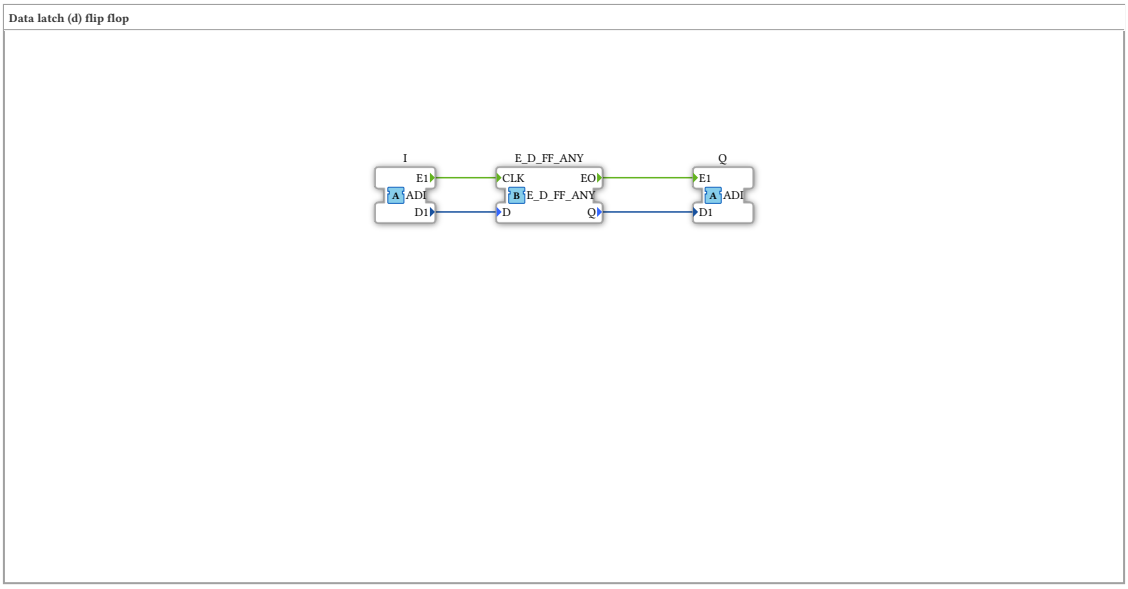
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ADI_D_FF

Value to latch – *ADI*



ADI – Latched value



• — ○ • — ○ • — ○ •

Introduction

The function block `ADI_D_FF` implements a data latch (D flip-flop) based on adapters. It serves to store and output a Boolean value when an event occurs. The block encapsulates the industrial standard function block `E_D_FF_ANY` and communicates exclusively via the unidirectional adapters `ADI`.

Interface Structure

Event Inputs

The function block has no separate event inputs. Event control is handled exclusively via the adapter `I`.

Event Outputs

The function block has no separate event outputs. Event output is exclusively via the `Q` adapter.

Data Inputs

The FB has no separate data inputs. Data is acquired via the `I` adapter.

Data Outputs

The FB has no separate data outputs. Data is output via the `Q` adapter.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::ADI`	`I`	Input (Socket)	Outputs the clock signal (`E1`) and the data value to be latched (`D1`).
`adapter::types::unidirectional::ADI`	`Q`	Output (Plug)	Outputs the stored data value (`D1`) and an acknowledgment event (`E1`).

Functionality

The `ADI_D_FF` operates on the principle of an edge-triggered D flip-flop. Each event arriving at the adapter `I` (`E1`) is interpreted as a clock signal (CLK). At each clock cycle, the current data value `D1` is received from the input, stored internally, and output at the adapter `Q` both as a new data value (`D1`) and as an acknowledgment event (`E1`).

Internally, the standardized IEC 61499 function block `E_D_FF_ANY` is used, which provides precisely this functionality. The adapters serve as uniform, reusable interfaces for events and data.

Technical Features

- All data and event exchange takes place via adapters of type `ADI` (unidirectional event and data transmission). This makes the function block easy to integrate into adapter-based architectures.
- No independent state machine is implemented; The entire logic is implemented by the internal function block `E_D_FF_ANY`.
- The implementation follows the IEC 61499-1 (Annex A) standard and uses the Eclipse 4diac IDE.

State Overview

The function block does not have explicit states in the sense of a state machine. Its behavior corresponds to a simple flip-flop:

- At each clock cycle (event at `I.E1`), the current value is taken from `I.D1` and passed to `Q.D1`.
- The stored value remains constant until the next clock cycle.
- The output `Q.E1` signals the transfer of a new value.

Application Scenarios

- **Synchronization:** Holding an asynchronously arriving Boolean signal at a defined clock time.
- **Latch:** Holding a data value for further processing in subsequent function block networks.
- **Buffering:** Decoupling event and data sources from further processing (e.g., in time-critical processes).

Comparison with Similar Function Blocks

Function Block	Function	Differences
`ADI_SR_FF`	Set-Reset Flip-Flop	Stores based on set/reset behavior, not clock-driven.
`ADI_T_FF`	T Flip-Flop	Toggles the state on every clock cycle.
`ADI_D_FF`	D flip-flop	Stores the incoming data value on the clock edge (as described here).

Conclusion

The `ADI_D_FF` is a compact, adapter-based D flip-flop component for the 4diac IDE. It offers a clean separation of clock and data and is ideally suited for use in event-driven automation solutions. The adapter encapsulation increases reusability and simplifies integration into existing projects.

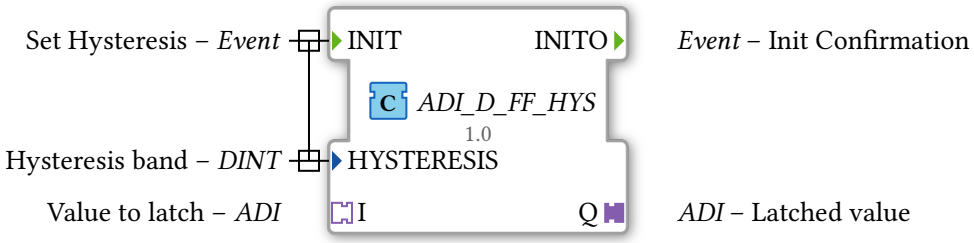
–

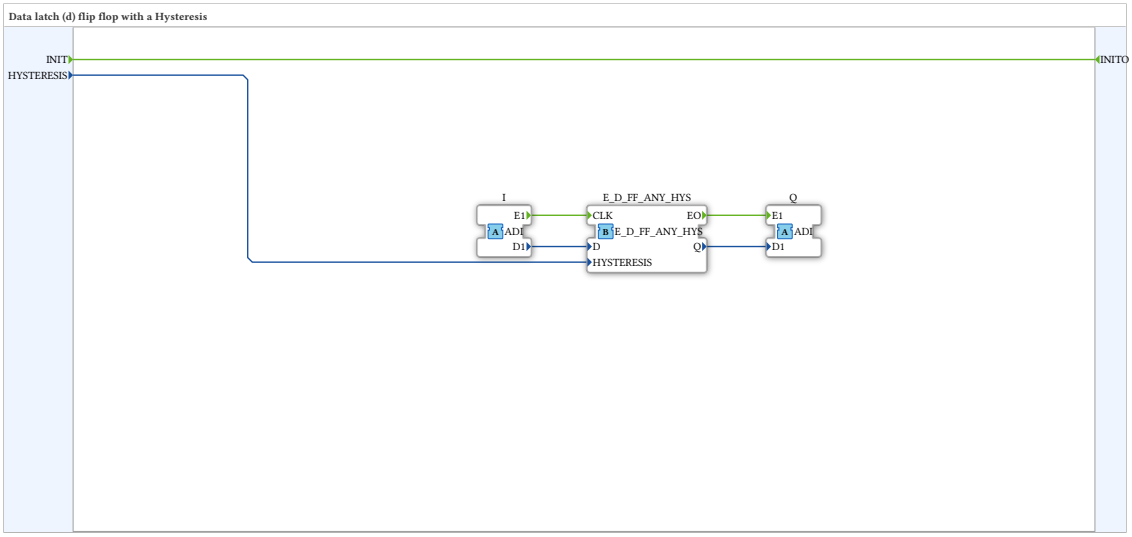
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ADI_D_FF_HYS





• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_D_FF_HYS** implements a data-timed (D) flip-flop with adjustable hysteresis. It receives values via a unidirectional adapter input `I` and outputs the state, including hysteresis, via the adapter output `Q`. The hysteresis bandwidth is set during the initialization event `INIT`.

Interface Structure

Event Inputs

Event	Type	Description
`INIT`	EInit	Sets the hysteresis bandwidth (in conjunction with `HYSTERESIS`).

Event Outputs

Event	Type	Description
`INITO`	EInit	Confirms the execution of `INIT`.

Data Inputs

Variable	Type	Description
`HYSTERESIS`	DINT	Hysteresis bandwidth (integer value).

Data Outputs

This function block does not have direct data outputs; The latched value is provided via the adapter `Q`.

Adapter

Interface	Type	Direction	Description
`I`	`adapter::types::unidirectional::ADI`	Socket	Input value (consisting of Event `E1` and Data `D1`).
`Q`	`adapter::types::unidirectional::ADI`	Plug	Latched output value (consisting of Event `E1` and Data `D1`).

Note: The `ADI` (unidirectional) adapter provides one event channel (E1) and one data channel (D1) by default.

Functionality

1. **Initialization**

An event at `INIT` transfers the value to `HYSTERESIS` and sets the internal hysteresis bandwidth for the flip-flop. The initialization event is then passed directly to `INITO`.

1. **Data Transfer**

The input adapter `I` delivers the current data value to `I.D1` with each event at `I.E1`.
The internal function block `E_D_FF_ANY_HYS` adopts this value according to the configured hysteresis:

- If the input value is **above** the upper threshold (stored value + hysteresis), the output is set to `1`.
- If the input value is **below** the lower threshold (stored value - hysteresis), the output is set to `0`.
- If the value remains within the hysteresis range, the output does not change.

1. **Output**

The internally latched value is passed to the data output `Q.D1`. Simultaneously, an event is triggered at `Q.E1` to signal data processing to downstream logic blocks.

Technical Features

- **Parameterizable Hysteresis:** The hysteresis bandwidth can be adjusted at runtime via `INIT` and `HYSTERESIS` (integer value, data type `DINT`).
- **Adapter-Based Communication:** The function block uses fully unidirectional adapters, enabling a clean separation of signal flow and event control. This makes it particularly suitable for modular and distributed signal processing.
- **Internal Implementation:** The hysteresis logic is implemented using the standardized function block `E_D_FF_ANY_HYS` from the library `logiBUS::signalprocessing::hysteresis`.
- **No Delay Due to INIT:** The initialization (`INIT → INITO`) is passed through in the same cycle; the hysteresis is available from the following cycle.

State Overview

The component does not have an explicit state machine, but is based on the internal state of the D flip-flop `E_D_FF_ANY_HYS`. This stores:

- The current output value (latched).
- The last input value received (for hysteresis analysis).

The output only changes when the new input value leaves the hysteresis band.

Application Scenarios

- **Debouncing of Sensor Switches:** Turn-on and turn-off thresholds with hysteresis prevent edge noise.
- **Threshold Monitoring with Delayed Release:** Temperature, pressure, or level control with defined turn-on and turn-off points.
- **Digitization of Analog Signals:** Conversion of a continuous range of values into a stable binary value (Schmitt trigger behavior).
- **Control of actuators with hysteresis:** Prevents rapid on/off switching, e.g., with thermostats or pumps.

Comparison with similar function blocks

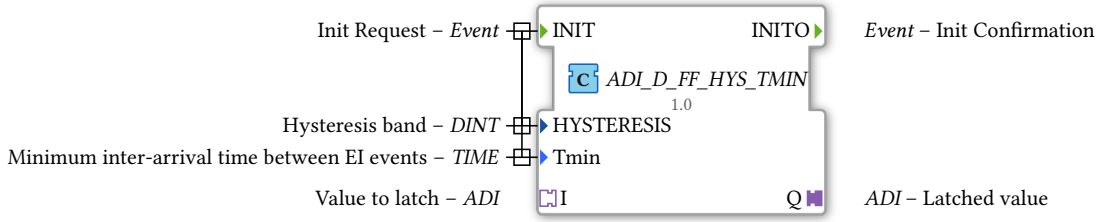
Function block	Properties
E_D_FF	Standard D flip-flop without hysteresis (direct transfer).
E_D_FF_ANY_HYS	Identical function, but with direct data ports instead of adapters.
Schmitt trigger	Similar behavior, but hysteresis cannot be configured at runtime.

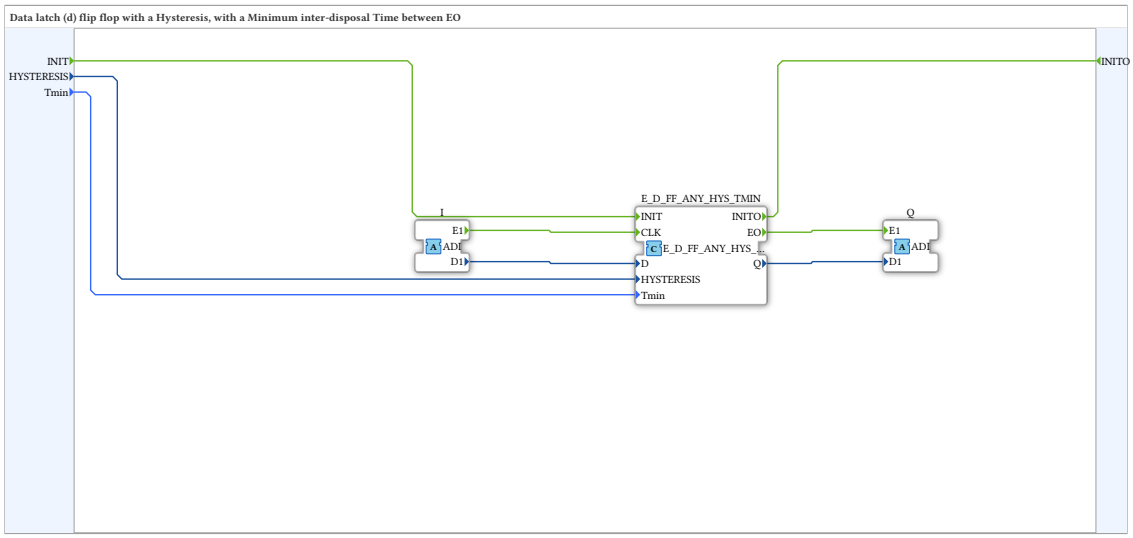
The **ADI_D_FF_HYS** offers particularly flexible integration into heterogeneous systems thanks to its adapter interface, while the hysteresis can be configured at runtime.

Conclusion

The **ADI_D_FF_HYS** is a robust, configurable D flip-flop with hysteresis, ideally suited for signal processing with stable switching thresholds. The use of unidirectional adapters facilitates integration into process-level control applications and enables a clear separation of data transport and event control.

ADI_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_D_FF_HYS_TMIN** implements a data latch (D flip-flop) with hysteresis and a minimal time delay between successive output events. It serves for stable and interference-free signal transmission in industrial control applications based on unidirectional adapters.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization request. This event input initializes the block with the parameters `HYSTERESIS` and `Tmin`.

Event Outputs

- **INITO** (EInit): Initialization confirmation. Outputted after successful initialization.

Data Inputs

- **HYSTERESIS** (DINT): Hysteresis band. Defines the switching threshold to prevent output chatter.
- **Tmin** (TIME): Minimum time between two output events (Inter-Arrival Time). Prevents excessively rapid switching.

Data Outputs

This module does not have its own data outputs. The output data is provided via an adapter.

Adapters

- **I** (Socket, Type: `adapter::types::unidirectional::ADI`): The value to be latched is received via this adapter input (event and data).
- **Q** (Plug, Type: `adapter::types::unidirectional::ADI`): The latched value is output via this adapter output (event and data).

Functionality

The function block internally uses the FB `E_D_FF_ANY_HYS_TMIN`. It processes the events (`I.E1`) and data (`I.D1`) received via the adapter **I**. The actual latching function uses hysteresis, which is defined by the parameter `HYSTERESIS`. Additionally, a minimum waiting time `Tmin` is maintained between two output events to suppress an excessively high event frequency. Output is provided via the adapter **Q** (`Q.E1` and `Q.D1`). Initialization (`INIT`) sets the parameters and activates the internal FB.

Technical Features

- **Adapter-Based Communication:** The function block uses unidirectional adapters (`ADI`) for data exchange. This enables loose coupling of data and events.
- **Configurable Hysteresis:** The hysteresis width can be set application-specifically to suppress noise or bounce effects.
- **Minimum Event Interval Time:** The maximum output frequency can be limited using `Tmin`, which helps to reduce the load on subsequent processing units.

State Overview

The function block has two stable internal states (e.g., HIGH/LOW or TRUE/FALSE) that represent the latched output value. A state change only occurs if the incoming value (via **I**) exceeds the hysteresis threshold and the time elapsed since the last output event is at least `Tmin`. Otherwise, the event is suppressed.

State	Description
LOW	Output value is logic 0 or low
HIGH	Output value is logic 1 or high

Changes between states occur according to the hysteresis and timing conditions.

Application Scenarios

- **Debouncing of switches or sensors:** Prevents multiple switching cycles due to mechanical bouncing.
- **Signal smoothing in process automation:** Suppresses small disturbances or noise on analog or digital signals.
- **Event rate limiting:** Protects subsequent functions from excessively rapid event sequences, e.g., in counters or communication interfaces.

Comparison with Similar Components

A conventional D flip-flop (e.g., `E_D_FF`) switches on every clock cycle without hysteresis or time delay. The `ADI_D_FF_HYS_TMIN`, on the other hand, offers:

- **Hysteresis:** Prevents oscillations with slowly changing or noisy signals.
- **Minimal Inter-Arrival Time:** Reduces the event rate and enables precise timing control.
- **Adapter Interface:** Specifically designed for use in complex, adapter-based function block networks.

Conclusion

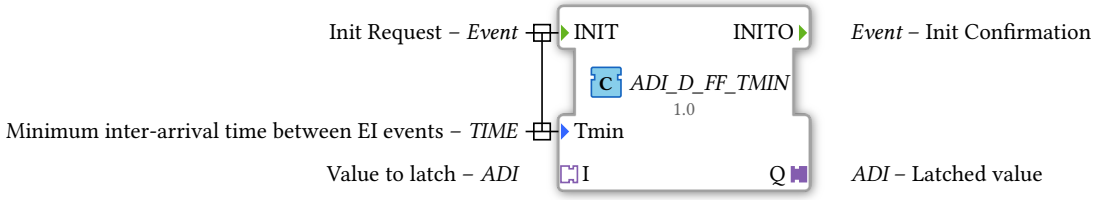
The `ADI_D_FF_HYS_TMIN` is a robust functional block for reliable signal processing with hysteresis and timing control. It is particularly suitable for applications requiring noise-free, filtered, and rate-limited transmission of events and data—typically in industrial control systems based on the 4diac framework and unidirectional adapters.

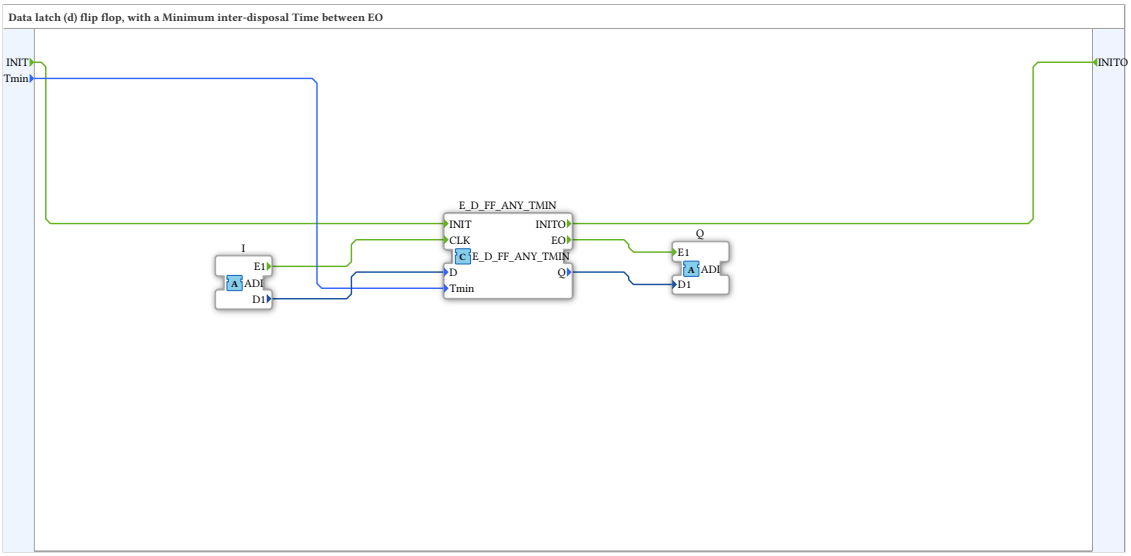
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ADI_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_D_FF_TMIN** implements a data latch (D flip-flop) with a configurable minimum dwell time between two consecutive transfer events. It serves to save a data value at a specific event time and only update it again when a predefined time interval (Tmin) has elapsed since the last transfer.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request; Passes the parameter Tmin

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirmation

Data Inputs

Name	Type	Comment
Tmin	TIME	Minimum time between two events at input I.E1

Data Outputs

No direct data outputs – the latched value is provided via the adapter output Q.

Adapter

Direction	Name	Type	Comment

Socket (Input)	I	adapter::types::unidirectional::ADI	Value and event to latch
Plug (Output)	Q	adapter::types::unidirectional::ADI	Latched Value and Associated Output Event

Functionality

The function block receives an event (I.E1) along with a data value (I.D1) via the adapter socket **I**. After this event, the value is transferred and output as Q.D1 at the adapter plug **Q** simultaneously, an event is sent to Q.E1.

However, at least the time specified in **Tmin** must elapse between two consecutive transfers. As long as this minimum period has not expired, an incoming event at I.E1 is ignored (the value remains unchanged).

The INIT input sets the parameter Tmin and confirms this via INITO, without affecting the flip-flop itself.

The function block (FB) internally uses the block `iec61499::events::E_D_FF_ANY_TMIN`, which encapsulates the actual logic (a D flip-flop with time control).

Technical Features

- **Adapter-based interface:** Instead of directly connected event and data ports, the inputs and outputs are implemented via unidirectional adapters (type `ADI`). This simplifies the reuse and separation of control and data flows.
- **Minimum time guarantee:** The Tmin time is measured from the moment of the last valid output event. If an input event occurs too early, the output remains unchanged – the old value is retained.
- **Initialization required:** The function block must first be initialized with a Tmin value via INIT before it responds to incoming adapter events.

State Overview

The function block (FB) has implicit state logic, which is represented by the internal `E_D_FF_ANY_TMIN`:

- **Idle:** Waiting for the first valid event at I.E1 after initialization.
- **Latched:** A value has been transferred; the timer for Tmin is running. New events at I.E1 are ignored until Tmin has expired.
- **Ready:** Tmin has expired; the function block is ready for the next transfer.

The state transitions are controlled by the events at I.E1 and the internal timer. Initialization returns the function block to the idle state.

Application Scenarios

- **Clock Decoupling in Industrial Applications:** Ensuring that data is transferred only at defined minimum intervals (e.g., for sensor queries with a fixed interval).
- **Event Debouncing:** Prevents duplicate evaluations for closely spaced events (e.g., push-button or switching signals with bounce time).
- **Data Storage in Safety-Critical Systems:** Once latched, values remain valid until the minimum time has elapsed – protecting against noise or transient disturbances.

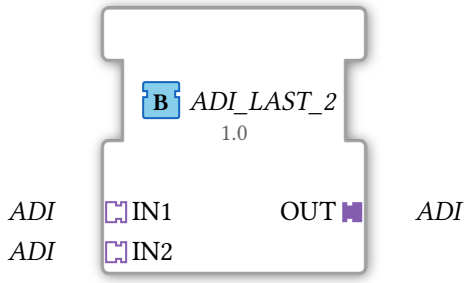
Comparison with Similar Function Blocks

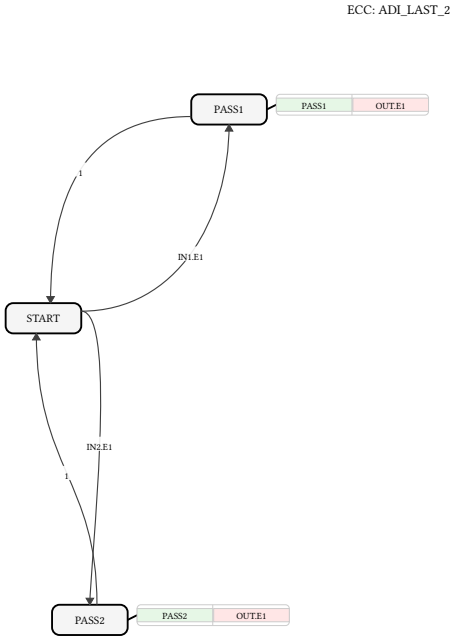
- **Standard D Flip-Flop (without Tmin):** A simple D flip-flop, such as `E_D_FF`, immediately adopts any value upon each event. This function block extends this functionality with a time lock.
- **E_D_FF_ANY_TMIN:** This is the direct internal base function block. `ADI_D_FF_TMIN` encapsulates its functionality and offers an adapter interface instead of separate event/data ports – this increases flexibility when connecting to other adapter-based components.

Conclusion

The `ADI_D_FF_TMIN` is a useful functional block for applications requiring reliable data acquisition with adjustable time decoupling. The use of adapters makes it particularly suitable for modular, adapter-based architectures according to IEC 61499. The minimum time enhances the overall system stability in environments with sporadic or bouncy events.

ADI_LAST_2





Introduction

The ADI_LAST_2 function block merges two same-type DINT adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: DINT adapter (unidirectional) - first signal source
- **IN2**: DINT adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: DINT adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

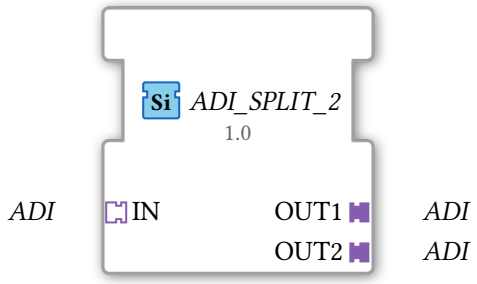
Application Scenarios

- Merging two equally-valid DINT signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same DINT value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

ADI_LAST_2 is the inverse of `ADI_SPLIT_2`: ADI_SPLIT_2 distributes 1 input to 2 outputs, ADI_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

ADI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_SPLIT_2` is used to distribute an incoming ADI adapter (socket) to two identical ADI output adapters (plugs). It is designed as a generic building block and enables simple signal bundling in adapter-based architectures.

Interface Structure

Event Inputs

None – this function block has no event inputs.

Event Outputs

None – this function block has no event outputs.

Data Inputs

None – this function block has no separate data inputs. All data is transported via the ADI adapter.

Data Outputs

None – This function block has no separate data outputs. All data is output via the ADI adapters.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::ADI`	`IN`	Socket	Incoming ADI adapter that is distributed to the two outputs.
`adapter::types::unidirectional::ADI`	`OUT1`	Plug	First output adapter (duplicate of the input).

adapter::types::unidirectional::ADI	OUT2	Plug	Second output adapter (duplicate of the input).

Functionality

The function block forwards the ADI adapter received via socket IN to both plugs OUT1 and OUT2 without modification. This makes the data and events provided by the input adapter available at both outputs simultaneously. No processing, delay, or transformation of the data takes place.

Technical Features

- **Generic Function Block:** The function block is declared as a generic type (GEN_ADI_SPLIT) and can be reused in different ADI environments.
- **No State Machine:** Due to its pure signal forwarding function, the function block does not have its own state machine (ECC).
- **Adapter-based:** All data exchange occurs exclusively via adapters, not via individual data or event ports.

State Overview

This functional block does not have an internal state machine. The routing is static and without time dependencies.

Application Scenarios

- **Signal Multiplication:** When an ADI adapter supplied by a sensor or controller needs to be distributed to multiple downstream functional units (e.g., for display, logging, or parallel processing).
- **Architecture Decoupling:** In systems where an output signal is required multiple times without the source component needing to know the number of destinations.
- **Prototypes and Test Setups:** Rapid expansion of an existing adapter circuit by adding a second branch.

Comparison with Similar Function Blocks

- **Event/Data Splitter:** Conventional splitter function blocks (e.g., SPLIT for BOOL) operate on individual data and event channels. ADI_SPLIT_2, on the other hand, copies an entire adapter, including all its data and events.
- **Mux/Demux Function Blocks:** Multiplexers and demultiplexers perform merging or distribution with selection; ADI_SPLIT_2 distributes rigidly without selection.

Change Detection

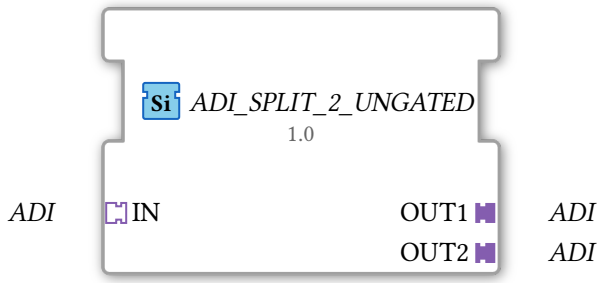
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

ADI_SPLIT_2` is a simple yet useful function block for distributing an ADI adapter to two identical outputs. It simplifies the design of adapter-based controllers and avoids redundant source blocks. Its generic nature and lack of internal logic make it a flexible component in IEC 61499 applications.

ADI_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of ADI_SPLIT_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_SPLIT_2_UNGATED` is used to distribute an incoming ADI adapter (socket) to two identical ADI output adapters (plugs). It is designed as a generic building block and enables simple signal bundling in adapter-based architectures.

Interface Structure

Event Inputs

None – this function block has no event inputs.

Event Outputs

None – this function block has no event outputs.

Data Inputs

None – this function block has no separate data inputs. All data is transported via the ADI adapter.

Data Outputs

None – This function block has no separate data outputs. All data is output via the ADI adapters.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::ADI`	`IN`	Socket	Incoming ADI adapter that is distributed to the two outputs.
`adapter::types::unidirectional::ADI`	`OUT1`	Plug	First output adapter (duplicate of the input).
`adapter::types::unidirectional::ADI`	`OUT2`	Plug	Second output adapter (duplicate of the input).

Functionality

The function block forwards the ADI adapter received via socket `IN` to both plugs `OUT1` and `OUT2` without modification. This makes the data and events provided by the input adapter available at both outputs simultaneously. No processing, delay, or transformation of the data takes place.

Technical Features

- **Generic Function Block:** The function block is declared as a generic type (`GEN_ADI_SPLIT`) and can be reused in different ADI environments.
- **No State Machine:** Due to its pure signal forwarding function, the function block does not have its own state machine (ECC).
- **Adapter-based:** All data exchange occurs exclusively via adapters, not via individual data or event ports.

State Overview

This functional block does not have an internal state machine. The routing is static and without time dependencies.

Application Scenarios

- **Signal Multiplication:** When an ADI adapter supplied by a sensor or controller needs to be distributed to multiple downstream functional units (e.g., for display, logging, or parallel processing).
- **Architecture Decoupling:** In systems where an output signal is required multiple times without the source component needing to know the number of destinations.
- **Prototypes and Test Setups:** Rapid expansion of an existing adapter circuit by adding a second branch.

Comparison with Similar Function Blocks

- **Event/Data Splitter:** Conventional splitter function blocks (e.g., `SPLIT` for `BOOL`) operate on individual data and event channels. `ADI_SPLIT_2_UNGATED`, on the other hand, copies an entire adapter, including all its data and events.
- **Mux/Demux Function Blocks:** Multiplexers and demultiplexers perform merging or distribution with selection; `ADI_SPLIT_2_UNGATED` distributes rigidly without selection.

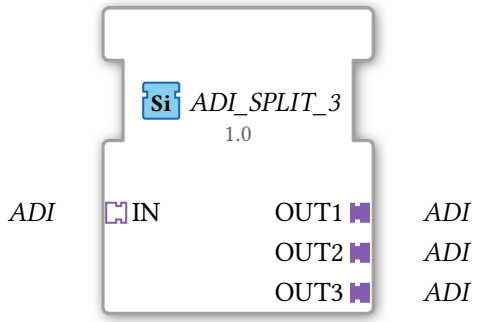
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

ADI_SPLIT_2_UNGATED` is a simple yet useful function block for distributing an ADI adapter to two identical outputs. It simplifies the design of adapter-based controllers and avoids redundant source blocks. Its generic nature and lack of internal logic make it a flexible component in IEC 61499 applications.

ADI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ADI_SPLIT_3** is used to distribute an incoming ADI data stream (adapter interface) to three identical outputs. It is designed as a generic function block and enables simple signal multiplication in adapter-based 4diac applications.

Interface Structure

The function block has only adapter interfaces – neither event nor data inputs/outputs are present.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Name	Type	Direction	Description
IN	adapter::types::unidirectional::ADI	Socket (Input)	Incoming ADI data stream that is split.
OUT1	adapter::types::unidirectional::ADI	Plug (Output)	First outgoing ADI data stream (copy of the input).
OUT2	adapter::types::unidirectional::ADI	Plug (Output)	Second outgoing ADI data stream.
OUT3	adapter::types::unidirectional::ADI	Plug (Output)	Third outgoing ADI data stream.

Functionality

The function block forwards all ADI data packets arriving at its socket `IN` unchanged to the three output plugs `OUT1`, `OUT2`, and `OUT3`. No processing or buffering takes place – distribution occurs directly and without delay. This behavior corresponds to passive signal multiplication (1:N splitter).

Technical Features

- **Generic Type:** The function block is defined using a generic class name (`GEN_ADI_SPLIT`), allowing it to be reused for any ADI-compliant data type.
- **Type Hash:** An empty type hash is stored, indicating a simple, unparameterized implementation.
- **No Runtime Logic:** Since neither event nor data interfaces exist, the function block is entirely controlled by the adapter connections and does not require its own ECC (Execution Control Chart).

State Overview

The function block has no internal states. Its functionality is limited to passively passing through adapter data – therefore, there is no state machine.

Application Scenarios

- **Signal Distribution in Control Systems:** When an ADI sensor value needs to be sent simultaneously to multiple consumers (e.g., visualization, control, logging).
- **Test and Simulation Environments:** A real ADI signal can be distributed to various test modules or simulation components using `ADI_SPLIT_3` without having to access the source multiple times.
- **Redundancy and Safety Concepts:** Distribution of a safety-critical signal to multiple independent evaluation units.

Comparison with Similar Components

- **`ADI_SPLIT_2`** – Splits to two outputs, otherwise identical functionality.
- **`ADI_MERGE`** – Combines multiple ADI inputs into one output (counterpart to the splitter).
- **`ADI_SELECT`** – Selects one of several ADI inputs based on a control signal (not pure distribution).

The `ADI_SPLIT_3` is specifically optimized for situations where exactly three identical copies of an ADI signal are required. If more or fewer outputs are needed, other splitter variants or combinations can be used.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ADI_SPLIT_3** is a simple yet useful generic building block for multiplying ADI data streams in three directions. Its pure adapter interface allows for seamless integration into existing 4diac projects without requiring additional event or data processing. It is particularly well-suited for passive signal distribution in modular, adapter-based automation solutions.

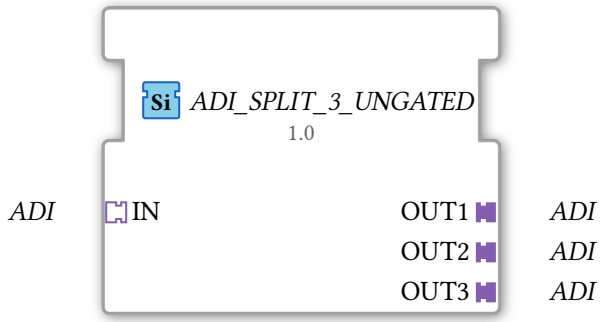
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ADI_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of ADI_SPLIT_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ADI_SPLIT_3_UNGATED** is used to distribute an incoming ADI data stream (adapter interface) to three identical outputs. It is designed as a generic function block and enables simple signal multiplication in adapter-based 4diac applications.

Interface Structure

The function block has only adapter interfaces – neither event nor data inputs/outputs are present.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Name	Type	Direction	Description
IN	adapter::types::unidirectional::ADI	Socket (Input)	Incoming ADI data stream that is split.
OUT1	adapter::types::unidirectional::ADI	Plug (Output)	First outgoing ADI data stream (copy of the input).
OUT2	adapter::types::unidirectional::ADI	Plug (Output)	Second outgoing ADI data stream.
OUT3	adapter::types::unidirectional::ADI	Plug (Output)	Third outgoing ADI data stream.

Functionality

The function block forwards all ADI data packets arriving at its socket `IN` unchanged to the three output plugs `OUT1`, `OUT2`, and `OUT3`. No processing or buffering takes place – distribution occurs directly and without delay. This behavior corresponds to passive signal multiplication (1:N splitter).

Technical Features

- **Generic Type:** The function block is defined using a generic class name (`GEN_ADI_SPLIT`), allowing it to be reused for any ADI-compliant data type.
- **Type Hash:** An empty type hash is stored, indicating a simple, unparameterized implementation.
- **No Runtime Logic:** Since neither event nor data interfaces exist, the function block is entirely controlled by the adapter connections and does not require its own ECC (Execution Control Chart).

State Overview

The function block has no internal states. Its functionality is limited to passively passing through adapter data – therefore, there is no state machine.

Application Scenarios

- **Signal Distribution in Control Systems:** When an ADI sensor value needs to be sent simultaneously to multiple consumers (e.g., visualization, control, logging).
- **Test and Simulation Environments:** A real ADI signal can be distributed to various test modules or simulation components using `ADI_SPLIT_3_UNGATED` without having to access the source multiple times.
- **Redundancy and Safety Concepts:** Distribution of a safety-critical signal to multiple independent evaluation units.

Comparison with Similar Components

- **`ADI_SPLIT_2`** – Splits to two outputs, otherwise identical functionality.
- **`ADI_MERGE`** – Combines multiple ADI inputs into one output (counterpart to the splitter).
- **`ADI_SELECT`** – Selects one of several ADI inputs based on a control signal (not pure distribution).

The `ADI_SPLIT_3_UNGATED` is specifically optimized for situations where exactly three identical copies of an ADI signal are required. If more or fewer outputs are needed, other splitter variants or combinations can be used.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

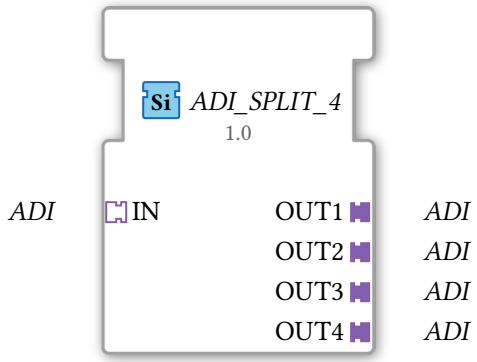
The **ADI_SPLIT_3_UNGATED** is a simple yet useful generic building block for multiplying ADI data streams in three directions. Its pure adapter interface allows for seamless integration into existing 4diac projects without requiring additional event or data processing. It is particularly well-suited for passive signal distribution in modular, adapter-based automation solutions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ADI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ADI_SPLIT_4** serves as a generic splitter for a single ADI data stream. It receives one ADI input via the socket interface `IN` and makes it available to four identical ADI outputs via the plug interfaces `OUT1`, `OUT2`, `OUT3`, and `OUT4`. This block is ideal when an ADI signal needs to be forwarded in parallel to several subsequent components.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Name	Direction	Description
`ADI` (unidirectional)	`IN`	Socket	Receives the ADI data stream to be distributed.
`ADI` (unidirectional)	`OUT1`	Plug	First output – identical copy of the input signal.
`ADI` (unidirectional)	`OUT2`	Plug	Second output – identical copy of the input signal.
`ADI` (unidirectional)	`OUT3`	Plug	Third output – identical copy of the input signal.
`ADI` (unidirectional)	`OUT4`	Plug	Fourth output – identical copy of the input signal.

Functionality

This module functions as a simple **replication** of the ADI signal. As soon as a valid ADI data stream is present at socket `IN`, it is passed on unchanged and without delay to all four plugs (`OUT1` … `OUT4`). No logic, filtering, or data manipulation takes place – the behavior is that of a passive signal distributor.

Since this is a generic function block (FB), the underlying ADI type is only determined at runtime or when it is integrated into a project. The interface definition itself is type-independent.

Technical Features

- **Generic Type** – The function block is declared as `GEN_ADI_SPLIT` and can be instantiated with any specific ADI adapter type.

- **No Events** – Signal distribution occurs purely via the adapter interfaces; there are no event inputs or outputs.
- **Scalability** – The function block is fixed at exactly four outputs. Alternative versions (e.g., `ADI_SPLIT_2`) are available for other numbers.

State Overview

The function block does not have an explicit state machine (ECC). Since it only implements passive signal transmission, no internal state is required.

Application Scenarios

- **Parallel Operation** – An ADI data stream provided by a sensor or predecessor function block is to be used simultaneously by multiple actuators, logic blocks, or visualization components.
- **Prototyping** – During the development phase, the splitter allows for easy testing of multiple paths of the same signal.
- **Redundancy** – The ADI signal can be distributed to multiple redundant evaluation paths.

Comparison with Similar Function Blocks

- **ADI_SPLIT_2** – Distributes one ADI signal to two outputs. Identical functionality, but fewer outputs.
- **ADI_MERGE** – Combines multiple ADI inputs into one output (counterpart to the splitter).
- **Manual Wiring** – Alternatively, distribution could be achieved by connecting multiple outputs to the same output; however, the splitter in the 4diac IDE is the cleaner and reusable solution.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ADI_SPLIT_4** is a simple yet useful generic function block for multiplying an ADI data stream. It requires no configuration and integrates seamlessly into modular automation projects. Its adapter-based interface ensures flexibility and allows it to be integrated into different environments without modifications to its internal logic.

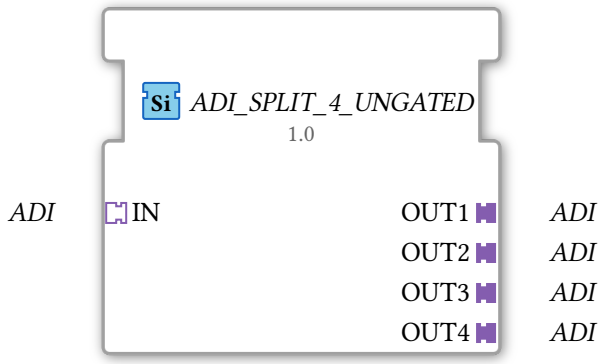
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ADI_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ADI_SPLIT_4_UNGATED** serves as a generic splitter for a single ADI data stream. It receives one ADI input via the socket interface `IN` and makes it available to four identical ADI outputs via the plug interfaces `OUT1`, `OUT2`, `OUT3`, and `OUT4`. This block is ideal when an ADI signal needs to be forwarded in parallel to several subsequent components.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Name	Direction	Description
`ADI` (unidirectional)	`IN`	Socket	Receives the ADI data stream to be distributed.
`ADI` (unidirectional)	`OUT1`	Plug	First output – identical copy of the input signal.
`ADI` (unidirectional)	`OUT2`	Plug	Second output – identical copy of the input signal.
`ADI` (unidirectional)	`OUT3`	Plug	Third output – identical copy of the input signal.
`ADI` (unidirectional)	`OUT4`	Plug	Fourth output – identical copy of the input signal.

Functionality

This module functions as a simple **replication** of the ADI signal. As soon as a valid ADI data stream is present at socket `IN`, it is passed on unchanged and without delay to all four plugs (`OUT1` … `OUT4`). No logic, filtering, or data manipulation takes place – the behavior is that of a passive signal distributor.

Since this is a generic function block (FB), the underlying ADI type is only determined at runtime or when it is integrated into a project. The interface definition itself is type-independent.

Technical Features

- **Generic Type** – The function block is declared as `GEN_ADI_SPLIT` and can be instantiated with any specific ADI adapter type.
- **No Events** – Signal distribution occurs purely via the adapter interfaces; there are no event inputs or outputs.

- **Scalability** – The function block is fixed at exactly four outputs. Alternative versions (e.g., `ADI_SPLIT_2`) are available for other numbers.

State Overview

The function block does not have an explicit state machine (ECC). Since it only implements passive signal transmission, no internal state is required.

Application Scenarios

- **Parallel Operation** – An ADI data stream provided by a sensor or predecessor function block is to be used simultaneously by multiple actuators, logic blocks, or visualization components.
- **Prototyping** – During the development phase, the splitter allows for easy testing of multiple paths of the same signal.
- **Redundancy** – The ADI signal can be distributed to multiple redundant evaluation paths.

Comparison with Similar Function Blocks

- **ADI_SPLIT_2** – Distributes one ADI signal to two outputs. Identical functionality, but fewer outputs.
- **ADI_MERGE** – Combines multiple ADI inputs into one output (counterpart to the splitter).
- **Manual Wiring** – Alternatively, distribution could be achieved by connecting multiple outputs to the same output; however, the splitter in the 4diac IDE is the cleaner and reusable solution.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

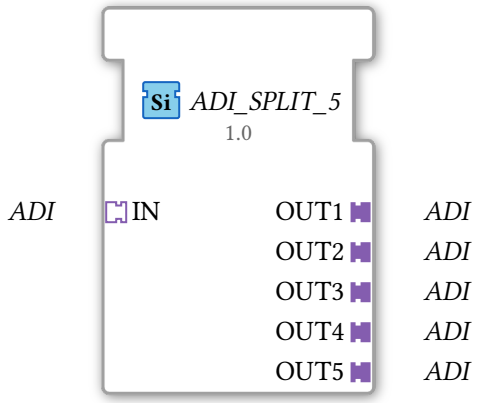
The **ADI_SPLIT_4_UNGATED** is a simple yet useful generic function block for multiplying an ADI data stream. It requires no configuration and integrates seamlessly into modular automation projects. Its adapter-based interface ensures flexibility and allows it to be integrated into different environments without modifications to its internal logic.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ADI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ADI_SPLIT_5** is used to split an incoming ADI adapter (unidirectional) into five identical output adapters. It is designed as a generic function block (Generic FB) and can be used in IEC 61499-based systems to forward a signal or data stream to multiple subsequent function blocks.

Interface Structure

Event Inputs
None

Event Outputs
None

Data Inputs
- **IN** (Socket, Type: `adapter::types::unidirectional::ADI`)

Input adapter that provides the signal to be distributed.

Data Outputs
- **OUT1** (Plug, Type: `adapter::types::unidirectional::ADI`)
- **OUT2** (Plug, Type: `adapter::types::unidirectional::ADI`)
- **OUT3** (Plug, Type: `adapter::types::unidirectional::ADI`)
- **OUT4** (Plug, Type: `adapter::types::unidirectional::ADI`)
- **OUT5** (Plug, Type: `adapter::types::unidirectional::ADI`)

All outputs are independent instances of the same adapter type and receive identical data.

Adapters
This module operates exclusively via adapter interfaces. No event or data inputs/outputs in the traditional sense are used. All communication takes place via the ADI adapters, which implicitly control signal and data forwarding.

Functionality
The function block **ADI_SPLIT_5** receives an ADI adapter via the socket **IN** and forwards the data and events it contains in parallel to the five plugs **OUT1** to **OUT5**. No processing or transformation of the data takes place – the block functions as a passive distributor (fan-out).

Since these are unidirectional ADI adapters, the signal is only passed from the input side to the output side; feedback from the outputs is not intended.

Technical Features
- **Generic Function Block**: The class `GEN_ADI_SPLIT` allows for flexible reuse, e.g., by parameterizing the type or number of outputs (here fixed at five).
- **Adapter-based**: No direct event or data inputs/outputs – all signal transmission occurs via the adapter interfaces.
- **Unidirectional**: The adapters are defined as `unidirectional`, meaning data flows only in one direction (from IN to OUT1-5).
- The function block contains no internal state logic or timing control.

State Overview
The **ADI_SPLIT_5** has no states of its own. It is a purely combinational function block that reacts immediately to incoming adapter data and passes it on unchanged to all outputs.

Application Scenarios
- **Signal Cascade**: Distribution of a sensor signal (e.g., temperature, pressure) to multiple evaluation and control blocks.
- **Diagnostics and Monitoring**: Simultaneous distribution of a data stream to processing and logging modules.
- **Load Balancing**: Parallel processing of the same input using different algorithms (e.g., filters, limit monitoring, visualization).

Comparison with Similar Components
In the IEC 61499 environment, variants such as **SPLIT_2**, **SPLIT_3**, or **SPLIT_N** are common, providing a different number of outputs. **ADI_SPLIT_5** is a specific implementation for exactly five outputs using unidirectional ADI adapter interfaces. Unlike event-based split components (e.g., `E_SPLIT`), distribution here is achieved via adapters, enabling type-based and typically data-oriented communication.

Change Detection

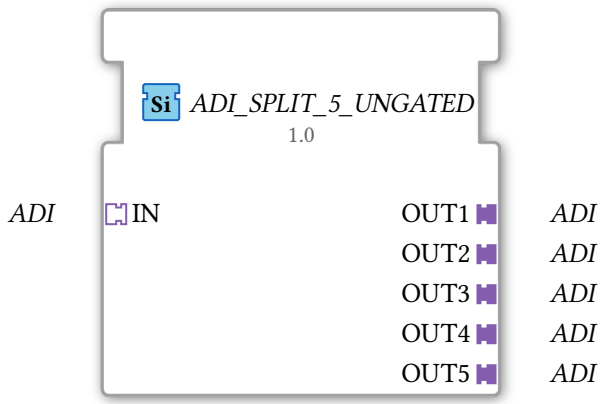
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ADI_SPLIT_5** is a simple yet useful function block for distributing a unidirectional ADI signal to up to five subsequent function blocks. It is particularly suitable for modular automation solutions where an input signal needs to be used multiple times without requiring additional control logic. Its generic implementation allows it to be reused in various contexts.

ADI_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ADI_SPLIT_5_UNGATED** is used to split an incoming ADI adapter (unidirectional) into five identical output adapters. It is designed as a generic function block (Generic FB) and can be used in IEC 61499-based systems to forward a signal or data stream to multiple subsequent function blocks.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

- **IN** (Socket, Type: `adapter::types::unidirectional::ADI`)

Input adapter that provides the signal to be distributed.

Data Outputs

- **OUT1** (Plug, Type: `adapter::types::unidirectional::ADI`)
- **OUT2** (Plug, Type: `adapter::types::unidirectional::ADI`)
- **OUT3** (Plug, Type: `adapter::types::unidirectional::ADI`)
- **OUT4** (Plug, Type: `adapter::types::unidirectional::ADI`)

- **OUT5** (Plug, Type: `adapter::types::unidirectional::ADI`)

All outputs are independent instances of the same adapter type and receive identical data.

Adapters

This module operates exclusively via adapter interfaces. No event or data inputs/outputs in the traditional sense are used. All communication takes place via the ADI adapters, which implicitly control signal and data forwarding.

Functionality

The function block **ADI_SPLIT_5_UNGATED** receives an ADI adapter via the socket **IN** and forwards the data and events it contains in parallel to the five plugs **OUT1** to **OUT5**. No processing or transformation of the data takes place – the block functions as a passive distributor (fan-out).

Since these are unidirectional ADI adapters, the signal is only passed from the input side to the output side; feedback from the outputs is not intended.

Technical Features

- **Generic Function Block**: The class `GEN_ADI_SPLIT` allows for flexible reuse, e.g., by parameterizing the type or number of outputs (here fixed at five).
- **Adapter-based**: No direct event or data inputs/outputs – all signal transmission occurs via the adapter interfaces.
- **Unidirectional**: The adapters are defined as `unidirectional`, meaning data flows only in one direction (from IN to OUT1-5).
- The function block contains no internal state logic or timing control.

State Overview

The **ADI_SPLIT_5_UNGATED** has no states of its own. It is a purely combinational function block that reacts immediately to incoming adapter data and passes it on unchanged to all outputs.

Application Scenarios

- **Signal Cascade**: Distribution of a sensor signal (e.g., temperature, pressure) to multiple evaluation and control blocks.
- **Diagnostics and Monitoring**: Simultaneous distribution of a data stream to processing and logging modules.
- **Load Balancing**: Parallel processing of the same input using different algorithms (e.g., filters, limit monitoring, visualization).

Comparison with Similar Components

In the IEC 61499 environment, variants such as **SPLIT_2**, **SPLIT_3**, or **SPLIT_N** are common, providing a different number of outputs. **ADI_SPLIT_5_UNGATED** is a specific implementation for exactly five outputs using unidirectional ADI adapter interfaces. Unlike event-based split components (e.g., `E_SPLIT`), distribution here is achieved via adapters, enabling type-based and typically data-oriented communication.

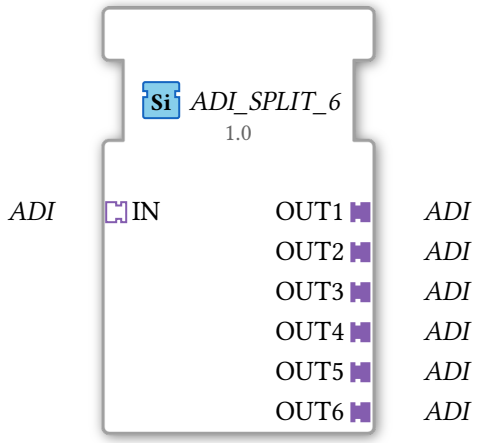
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **ADI_SPLIT_5_UNGATED** is a simple yet useful function block for distributing a unidirectional ADI signal to up to five subsequent function blocks. It is particularly suitable for modular automation solutions where an input signal needs to be used multiple times without requiring additional control logic. Its generic implementation allows it to be reused in various contexts.

ADI_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The ADI_SPLIT_6 function block is used to distribute a single (unidirectional) ADI adapter input to six identical ADI adapter outputs. It is designed as a generic building block for the Eclipse 4diac framework and is typically used when an ADI signal needs to be forwarded to multiple downstream components.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

The module communicates exclusively via adapter interfaces:
Socket (Input):

- `IN`: Type `adapter::types::unidirectional::ADI` – receives a unidirectional ADI signal.

Plugs (Outputs):

- `OUT1` to `OUT6`: each type `adapter::types::unidirectional::ADI` – pass the incoming signal identically to six parallel outputs.

Functionality

The module performs a simple 1:6 distribution of the ADI adapter signal. All data and events arriving at socket `IN` are passed on unchanged and simultaneously to all six plugs `OUT1`…`OUT6`. No buffering, filtering, or delay occurs. The function block thus operates as a passive splitter at the adapter level.

Technical Features

- **Generic Implementation:** The function block is implemented as a generic block (`eclipse4diac::core::GenericClassName = 'GEN_ADI_SPLIT'`) and can be reused in various projects.
- **Unidirectional Adapters:** The ADI adapters used are unidirectional; feedback via the outputs is not provided.
- **No Internal States:** Since the function block does not perform any event or data processing, it does not have an internal state machine.

- **Performance:** Due to the direct forwarding without logic or memory, there is virtually no latency.

State Overview

This function block does not have a state machine or explicit operating states. It is always active and immediately passes the input signal to all outputs.

Application Scenarios

- **Signal Distribution in Control Systems:** An ADI signal provided by a sensor or central logic unit is to be sent in parallel to multiple actuators, displays, or downstream functions.
- **Test Environments:** During simulation or debugging, a single signal can be split across multiple receivers to test different components simultaneously.
- **Redundancy Mapping:** If a signal is needed multiple times (e.g., for monitoring and control paths), this splitter can provide clean decoupling of the outputs.

Comparison with Similar Function Blocks

- **SPLIT for Standard Data Types:** Many frameworks offer splitters for simple data types (e.g., `SPLIT_INT`). The `ADI_SPLIT_6` function block is specifically designed for the `ADI` adapter type and operates at a higher level of abstraction (adapters instead of individual data points).
- **ADI_MERGE / ADI_COMBINE:** While this function block distributes a signal, other function blocks combine multiple adapter inputs into a single output.
- **Other Splitter Variants:** Splitters may exist for other adapter types (e.g., bidirectional ADI). This function block is limited to unidirectional signals.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `ADI_SPLIT_6` function block is a simple yet effective function block for multiplying unidirectional ADI adapter signals. Its generic design and low complexity make it ideally suited for the modular development of control applications in the Eclipse 4diac environment. Its applications range from simple signal distribution to testing and redundancy scenarios.

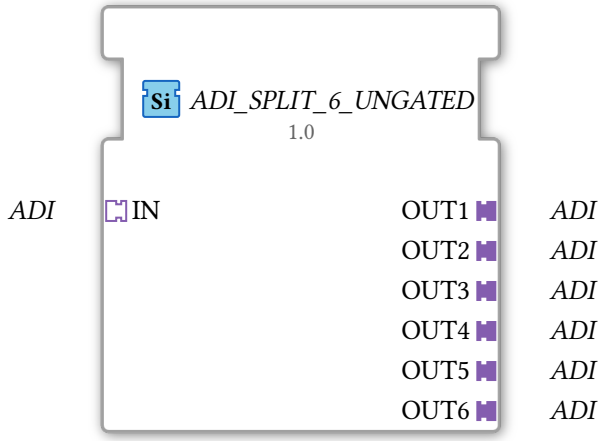
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ADI_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The ADI_SPLIT_6_UNGATED function block is used to distribute a single (unidirectional) ADI adapter input to six identical ADI adapter outputs. It is designed as a generic building block for the Eclipse 4diac framework and is typically used when an ADI signal needs to be forwarded to multiple downstream components.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

The module communicates exclusively via adapter interfaces:
Socket (Input):

- `IN`: Type `adapter::types::unidirectional::ADI` – receives a unidirectional ADI signal.

Plugs (Outputs):

- `OUT1` to `OUT6`: each type `adapter::types::unidirectional::ADI` – pass the incoming signal identically to six parallel outputs.

Functionality

The module performs a simple 1:6 distribution of the ADI adapter signal. All data and events arriving at socket `IN` are passed on unchanged and simultaneously to all six plugs `OUT1`…`OUT6`. No buffering, filtering, or delay occurs. The function block thus operates as a passive splitter at the adapter level.

Technical Features

- **Generic Implementation:** The function block is implemented as a generic block (`eclipse4diac::core::GenericClassName = 'GEN_ADI_SPLIT'`) and can be reused in various projects.
- **Unidirectional Adapters:** The ADI adapters used are unidirectional; feedback via the outputs is not provided.
- **No Internal States:** Since the function block does not perform any event or data processing, it does not have an internal state machine.

- **Performance:** Due to the direct forwarding without logic or memory, there is virtually no latency.

State Overview

This function block does not have a state machine or explicit operating states. It is always active and immediately passes the input signal to all outputs.

Application Scenarios

- **Signal Distribution in Control Systems:** An ADI signal provided by a sensor or central logic unit is to be sent in parallel to multiple actuators, displays, or downstream functions.
- **Test Environments:** During simulation or debugging, a single signal can be split across multiple receivers to test different components simultaneously.
- **Redundancy Mapping:** If a signal is needed multiple times (e.g., for monitoring and control paths), this splitter can provide clean decoupling of the outputs.

Comparison with Similar Function Blocks

- **SPLIT for Standard Data Types:** Many frameworks offer splitters for simple data types (e.g., `SPLIT_INT`). The `ADI_SPLIT_6_UNGATED` function block is specifically designed for the `ADI` adapter type and operates at a higher level of abstraction (adapters instead of individual data points).
- **ADI_MERGE / ADI_COMBINE:** While this function block distributes a signal, other function blocks combine multiple adapter inputs into a single output.
- **Other Splitter Variants:** Splitters may exist for other adapter types (e.g., bidirectional ADI). This function block is limited to unidirectional signals.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

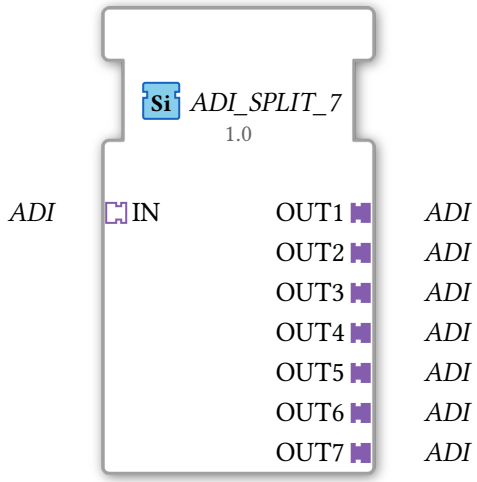
The `ADI_SPLIT_6_UNGATED` function block is a simple yet effective function block for multiplying unidirectional ADI adapter signals. Its generic design and low complexity make it ideally suited for the modular development of control applications in the Eclipse 4diac environment. Its applications range from simple signal distribution to testing and redundancy scenarios.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ADI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ADI_SPLIT_7** is used to split an incoming **ADI** adapter signal into seven identical outputs. It is implemented as a generic function block (Generic FB) and enables the unidirectional distribution of data via one adapter socket to seven adapter plugs.

Interface Structure

Event Inputs

None – the function block has no event inputs. Data is transmitted exclusively via the adapter interface.

Event Outputs

None – the function block has no event outputs. The output adapters are supplied directly with the received data.

Data Inputs

None – the data is not provided via separate data inputs, but rather via the adapter socket IN.

Data Outputs

None – the data is not output via separate data outputs, but rather via the adapter plugs OUT1 ... OUT7.

Adapters

Name	Direction	Type	Description
IN	Socket	adi (unidirectional)	Input adapter that receives the data to be distributed.
OUT1	Plug	adi (unidirectional)	First output adapter (identical to IN).
OUT2	Plug	adi (unidirectional)	Second output adapter (identical to IN).
OUT3	Plug	adi (unidirectional)	Third output adapter (identical to IN).
OUT4	Plug	adi (unidirectional)	Fourth output adapter (identical to IN).
OUT5	Plug	ADI (unidirectional)	Fifth output adapter (identical to IN).
OUT6	Plug	ADI (unidirectional)	Sixth output adapter (identical to IN).
OUT7	Plug	ADI (unidirectional)	Seventh output adapter (identical to IN).

Functionality

This component functions as a **1-to-7 splitter** for ADI adapter signals. When the signal is set at the input adapter **IN**, it is copied to all seven output adapters **OUT1** to **OUT7** without further processing or delay. The component has no internal logic or state machine – it simply passes the data on passively.

- **Generic component** – the type is instantiated as `GEN_ADI_SPLIT` and can be reused in various applications.
- **No Event Control** – Data transmission is purely data-driven without event inputs/outputs.
- **1:1 Copy** – The data at the input is transferred unchanged to all outputs.

The module has **no state machine** (ECC). It is stateless, meaning it does not react to events and has no internal states. The output data directly follows the input data.

- **Signal Distribution** – A single ADI signal (e.g., sensor value, control signal) must be distributed to multiple consumers (e.g., control blocks, visualizations).
- **Redundancy** – A signal is required for redundant processing paths.
- **Test and Simulation Environments** – an incoming signal is split for different test instances.

Function Block	Number of Outputs	Properties
ADI_SPLIT_2	2	Distributes 1 ADI to 2 outputs
ADI_SPLIT_4	4	Distributes 1 ADI to 4 outputs
ADI_SPLIT_7	7	Distributes 1 ADI to 7 outputs

ADI_MERGE	1 (Input: multiple)	Combines multiple ADI inputs into one output

Compared to manually connecting multiple ADI_Split function blocks in parallel, the `ADI_SPLIT_7` reduces configuration effort and improves network clarity.

The `ADI_SPLIT_7` is a simple yet powerful component for distributing a unidirectional ADI signal to seven identical outputs. Thanks to its generic implementation and the absence of event-driven control, it is ideally suited for pure distribution in automation and control environments based on the ADI adapter protocol.

- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

Change Detection

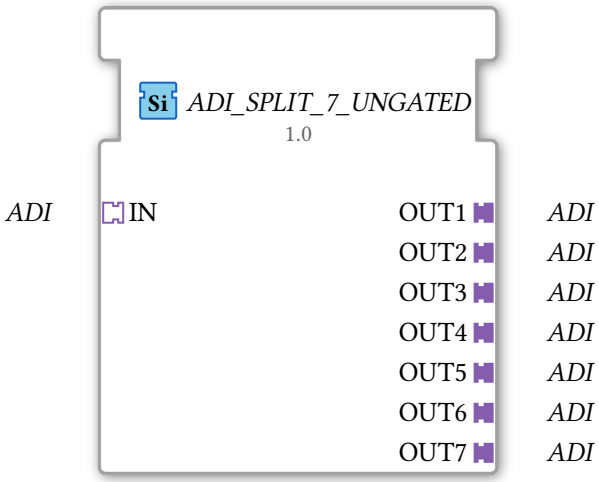
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

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

ADI_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ADI_SPLIT_7_UNGATED** is used to split an incoming **ADI** adapter signal into seven identical outputs. It is implemented as a generic function block (Generic FB) and enables the unidirectional distribution of data via one adapter socket to seven adapter plugs.

Interface Structure

Event Inputs

None – the function block has no event inputs. Data is transmitted exclusively via the adapter interface.

Event Outputs

None – the function block has no event outputs. The output adapters are supplied directly with the received data.

Data Inputs

None – the data is not provided via separate data inputs, but rather via the adapter socket IN.

Data Outputs

None – the data is not output via separate data outputs, but rather via the adapter plugs OUT1 ... OUT7.

Adapters

Name	Direction	Type	Description
IN	Socket	adi (unidirectional)	Input adapter that receives the data to be distributed.
OUT1	Plug	adi (unidirectional)	First output adapter (identical to IN).
OUT2	Plug	adi (unidirectional)	Second output adapter (identical to IN).
OUT3	Plug	adi (unidirectional)	Third output adapter (identical to IN).
OUT4	Plug	adi (unidirectional)	Fourth output adapter (identical to IN).
OUT5	Plug	ADI (unidirectional)	Fifth output adapter (identical to IN).
OUT6	Plug	ADI (unidirectional)	Sixth output adapter (identical to IN).
OUT7	Plug	ADI (unidirectional)	Seventh output adapter (identical to IN).

Functionality

This component functions as a **1-to-7 splitter** for ADI adapter signals. When the signal is set at the input adapter **IN**, it is copied to all seven output adapters **OUT1** to **OUT7** without further processing or delay. The component has no internal logic or state machine – it simply passes the data on passively.

- **Generic component** – the type is instantiated as `GEN_ADI_SPLIT` and can be reused in various applications.
- **No Event Control** – Data transmission is purely data-driven without event inputs/outputs.
- **1:1 Copy** – The data at the input is transferred unchanged to all outputs.

The module has **no state machine** (ECC). It is stateless, meaning it does not react to events and has no internal states. The output data directly follows the input data.

- **Signal Distribution** – A single ADI signal (e.g., sensor value, control signal) must be distributed to multiple consumers (e.g., control blocks, visualizations).
- **Redundancy** – A signal is required for redundant processing paths.
- **Test and Simulation Environments** – an incoming signal is split for different test instances.

Function Block	Number of Outputs	Properties
ADI_SPLIT_2	2	Distributes 1 ADI to 2 outputs
ADI_SPLIT_4	4	Distributes 1 ADI to 4 outputs
ADI_SPLIT_7_UNGATED	7	Distributes 1 ADI to 7 outputs
ADI_MERGE	1 (Input: multiple)	Combines multiple ADI inputs into one output

Compared to manually connecting multiple ADI_Split function blocks in parallel, the `ADI_SPLIT_7_UNGATED` reduces configuration effort and improves network clarity.

The `ADI_SPLIT_7_UNGATED` is a simple yet powerful component for distributing a unidirectional ADI signal to seven identical outputs. Thanks to its generic implementation and the absence of event-driven control, it is ideally suited for pure distribution in automation and control environments based on the ADI adapter protocol.

- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

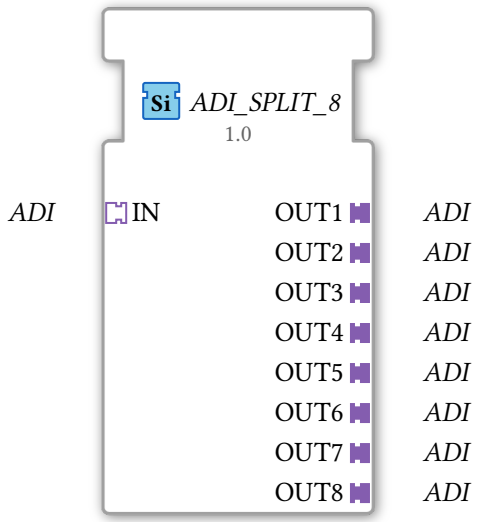
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

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

ADI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_SPLIT_8` is used to split a single unidirectional ADI (Application Data Interface) adapter into eight identical output adapters. It is implemented as a generic block that can be adapted to different ADI contexts by specifying a `GenericClassName` attribute. The function block simplifies signal distribution in control applications by distributing an incoming adapter path to multiple receivers without data loss.

Interface Structure

Event Inputs

There are no event inputs. Data is passed continuously and without events via a direct adapter connection.

Event Outputs

There are no event outputs.

Data Inputs

No data inputs in the conventional sense are defined. Information is transmitted exclusively via the adapter interfaces.

Data Outputs

No data outputs are available.

Adapters

Direction	Name	Type	Description
Socket	IN	ADI (unidirectional)	Input interface distributed to eight outputs.
Plug	OUT1	ADI (unidirectional)	First outgoing adapter (identical to IN).
Plug	OUT2	ADI (unidirectional)	Second outgoing adapter.
Plug	OUT3	ADI (unidirectional)	Third outgoing adapter.
Plug	OUT4	ADI (unidirectional)	Fourth outgoing adapter.
Plug	OUT5	ADI (unidirectional)	Fifth outgoing adapter.
Plug	OUT6	ADI (unidirectional)	Sixth outgoing adapter.
Plug	OUT7	ADI (unidirectional)	Seventh outgoing adapter.
Plug	OUT8	ADI (unidirectional)	Eighth outgoing adapter.

Functionality

The function block forwards the ADI adapter arriving via socket `IN` unchanged to all eight `OUT` plugs. Each output represents an independent, but identical copy of the incoming adapter. Distribution occurs directly – there is no buffering, delay, or transformation. Changes to the `IN` adapter are immediately propagated to all `OUT` adapters. The function block operates purely combinatorially and requires no clocking or initialization.

Technical Features

- **Generic Type**: The FB can be parameterized to various specific ADI implementations using the attribute `eclipse4diac::core::GenericClassName`. By default, the class name is set to `'GEN_ADI_SPLIT'`.
- **No State Management**: The FB has no internal memory or state machines; it acts as a simple signal distributor.
- **Unidirectionality**: All adapters are unidirectional – data flows only from the socket to the plugs. Feedback is not provided.
- **No Event Control**: Distribution occurs without events, which increases runtime determinism.

State Overview

The `ADI_SPLIT_8` FB has no internal states or an ECC (Execution Control Chart). It behaves statically and performs the same function at all times. Therefore, a state overview is unnecessary.

Application Scenarios

- **Signal Broadcasting**: Distributing a measured value or control signal (e.g., temperature, speed) to multiple parallel control loops.
- **Adapter Multiplexing**: Providing a central ADI path for multiple subsystems that require the same data stream.
- **Test and Simulation Environments**: Feeding a simulated signal to multiple receiver blocks simultaneously.
- **Modular Automation Architectures**: Creating redundant or parallel processing paths by simply duplicating adapter connections.

Comparison with Similar Components

- **SPLIT (Data-Event Variant)**: Classic split components (e.g., `E_SPLIT`) distribute events or data values, but not entire adapter paths. `ADI_SPLIT_8` operates at a higher level of abstraction (adapter level).
- **FORK FBs**: Some IEC 61499 implementations offer fork blocks for adapters, but these are often event-driven. This FB is distinguished by its absence of events and its fixed number of eight outputs.
- **Manual Wiring**: Without this FB, each destination connection would have to be implemented individually via coupling adapters or manual branching – more complex and prone to errors.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

ADI_SPLIT_8` is a powerful, generic split block for unidirectional ADI adapters in IEC 61499 applications. It significantly reduces wiring effort, improves clarity, and allows for the easy duplication of an adapter path. Thanks to its generic parameterization and event-free operation, it is particularly well-suited for data-driven automation systems that require reliable, low-latency signal distribution.

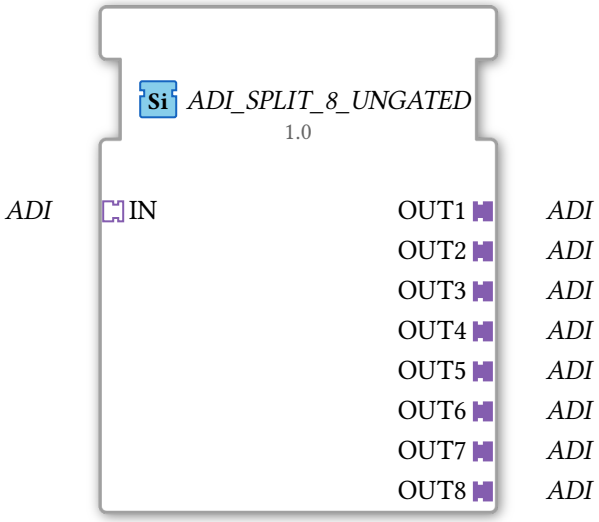
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ADI_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_SPLIT_8_UNGATED` is used to split a single unidirectional ADI (Application Data Interface) adapter into eight identical output adapters. It is implemented as a generic block that can be adapted to different ADI contexts by specifying a `GenericClassName` attribute. The function block simplifies signal distribution in control applications by distributing an incoming adapter path to multiple receivers without data loss.

Interface Structure

Event Inputs

There are no event inputs. Data is passed continuously and without events via a direct adapter connection.

Event Outputs

There are no event outputs.

Data Inputs

No data inputs in the conventional sense are defined. Information is transmitted exclusively via the adapter interfaces.

Data Outputs

No data outputs are available.

Adapters

Direction	Name	Type	Description
Socket	IN	ADI (unidirectional)	Input interface distributed to eight outputs.
Plug	OUT1	ADI (unidirectional)	First outgoing adapter (identical to IN).
Plug	OUT2	ADI (unidirectional)	Second outgoing adapter.
Plug	OUT3	ADI (unidirectional)	Third outgoing adapter.
Plug	OUT4	ADI (unidirectional)	Fourth outgoing adapter.
Plug	OUT5	ADI (unidirectional)	Fifth outgoing adapter.
Plug	OUT6	ADI (unidirectional)	Sixth outgoing adapter.
Plug	OUT7	ADI (unidirectional)	Seventh outgoing adapter.
Plug	OUT8	ADI (unidirectional)	Eighth outgoing adapter.

Functionality

The function block forwards the ADI adapter arriving via socket `IN` unchanged to all eight `OUT` plugs. Each output represents an independent, but identical copy of the incoming adapter. Distribution occurs directly – there is no buffering, delay, or transformation. Changes to the `IN` adapter are immediately propagated to all `OUT` adapters. The function block operates purely combinatorially and requires no clocking or initialization.

Technical Features

- **Generic Type**: The FB can be parameterized to various specific ADI implementations using the attribute `eclipse4diac::core::GenericClassName`. By default, the class name is set to `'GEN_ADI_SPLIT'`.
- **No State Management**: The FB has no internal memory or state machines; it acts as a simple signal distributor.
- **Unidirectionality**: All adapters are unidirectional – data flows only from the socket to the plugs. Feedback is not provided.
- **No Event Control**: Distribution occurs without events, which increases runtime determinism.

State Overview

The `ADI_SPLIT_8_UNGATED` FB has no internal states or an ECC (Execution Control Chart). It behaves statically and performs the same function at all times. Therefore, a state overview is unnecessary.

Application Scenarios

- **Signal Broadcasting**: Distributing a measured value or control signal (e.g., temperature, speed) to multiple parallel control loops.
- **Adapter Multiplexing**: Providing a central ADI path for multiple subsystems that require the same data stream.

- **Test and Simulation Environments**: Feeding a simulated signal to multiple receiver blocks simultaneously.
- **Modular Automation Architectures**: Creating redundant or parallel processing paths by simply duplicating adapter connections.

Comparison with Similar Components

- **SPLIT (Data-Event Variant)**: Classic split components (e.g., `E_SPLIT`) distribute events or data values, but not entire adapter paths. `ADI_SPLIT_8_UNGATED` operates at a higher level of abstraction (adapter level).
- **FORK FBs**: Some IEC 61499 implementations offer fork blocks for adapters, but these are often event-driven. This FB is distinguished by its absence of events and its fixed number of eight outputs.
- **Manual Wiring**: Without this FB, each destination connection would have to be implemented individually via coupling adapters or manual branching – more complex and prone to errors.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

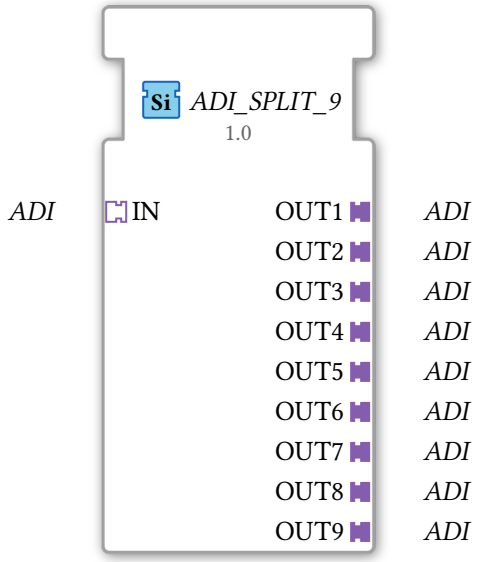
ADI_SPLIT_8_UNGATED` is a powerful, generic split block for unidirectional ADI adapters in IEC 61499 applications. It significantly reduces wiring effort, improves clarity, and allows for the easy duplication of an adapter path. Thanks to its generic parameterization and event-free operation, it is particularly well-suited for data-driven automation systems that require reliable, low-latency signal distribution.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ADI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_SPLIT_9` is a generic component that splits an incoming ADI signal (adapter) into nine identical output signals. It functions purely as a splitter at the adapter level.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapter
- **IN** (Socket): Adapter type `adapter::types::unidirectional::ADI` – Input signal.
- **OUT1** to **OUT9** (Plugs): Nine adapters of the same type – outputs that pass the input signal on unchanged.

Functionality
The function block forwards the ADI signal present at socket `IN` to all nine plugs `OUT1` to `OUT9`. No signal processing or conversion takes place; the distribution is purely structural via the adapter connections. Due to the unidirectional adapter type, the data flow direction from the socket to the plugs is fixed.

Technical Features
- The function block is defined as a generic function block (attribute `eclipse4diac::core::GenericClassName = 'GEN_ADI_SPLIT'`) and can therefore be reused in different projects.
- It has neither events nor data inputs/outputs; all functionality resides in the adapter topology.
- The type hash attribute is empty, indicating a simple, non-versioned execution.

State Overview
The function block has no state machines or algorithms. It is purely passive and does not react to events.

Application Scenarios
- Distributing an analog or digital sensor signal (as ADI) to multiple downstream function blocks.
- Splitting a control signal in an automation system to control identical actuators in parallel.
- Generating signal copies for monitoring, testing, or diagnostic purposes.

Comparison with Similar Function Blocks
- Unlike data- or event-based splitters, `ADI_SPLIT_9` operates at the higher adapter level, enabling clean architectural separation.
- Similar function blocks exist with a variable number of outputs (e.g., `ADI_SPLIT_M`), while this function block offers a fixed number of nine outputs.

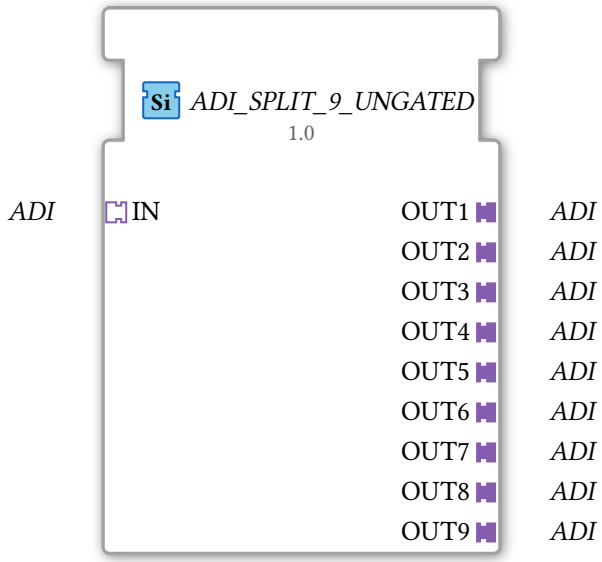
Change Detection
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion
ADI_SPLIT_9` is a simple, generic function block for 1-to-9 distribution of a unidirectional ADI signal. It is suitable for applications that require multiple, identical forwarding of adapter signals without additional logic.

🌐 Related topic subpages on ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

ADI_SPLIT_9_UNGATED

i UNGATED variant: This block is the ungated version of `ADI_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_SPLIT_9_UNGATED` is a generic component that splits an incoming ADI signal (adapter) into nine identical output signals. It functions purely as a splitter at the adapter level.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

- **IN** (Socket): Adapter type `adapter::types::unidirectional::ADI` – Input signal.
- **OUT1** to **OUT9** (Plugs): Nine adapters of the same type – outputs that pass the input signal on unchanged.

Functionality

The function block forwards the ADI signal present at socket `IN` to all nine plugs `OUT1` to `OUT9`. No signal processing or conversion takes place; the distribution is purely structural via the adapter connections. Due to the unidirectional adapter type, the data flow direction from the socket to the plugs is fixed.

Technical Features

- The function block is defined as a generic function block (attribute `eclipse4diac::core::GenericClassName = 'GEN_ADI_SPLIT'`) and can therefore be reused in different projects.
- It has neither events nor data inputs/outputs; all functionality resides in the adapter topology.
- The type hash attribute is empty, indicating a simple, non-versioned execution.

State Overview

The function block has no state machines or algorithms. It is purely passive and does not react to events.

Application Scenarios

- Distributing an analog or digital sensor signal (as ADI) to multiple downstream function blocks.
- Splitting a control signal in an automation system to control identical actuators in parallel.
- Generating signal copies for monitoring, testing, or diagnostic purposes.

Comparison with Similar Function Blocks

- Unlike data- or event-based splitters, `ADI_SPLIT_9_UNGATED` operates at the higher adapter level, enabling clean architectural separation.
- Similar function blocks exist with a variable number of outputs (e.g., `ADI_SPLIT_M`), while this function block offers a fixed number of nine outputs.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

ADI_SPLIT_9_UNGATED` is a simple, generic function block for 1-to-9 distribution of a unidirectional ADI signal. It is suitable for applications that require multiple, identical forwarding of adapter signals without additional logic.

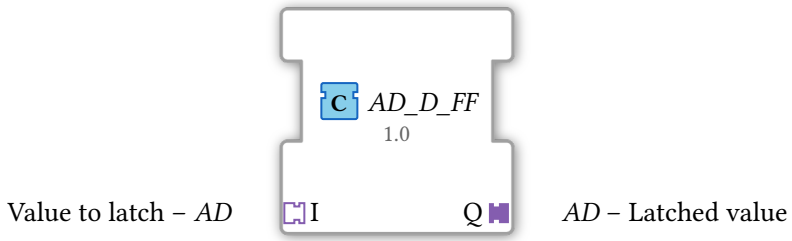
🌐 Related topic subpages on ms-muc-docs.de

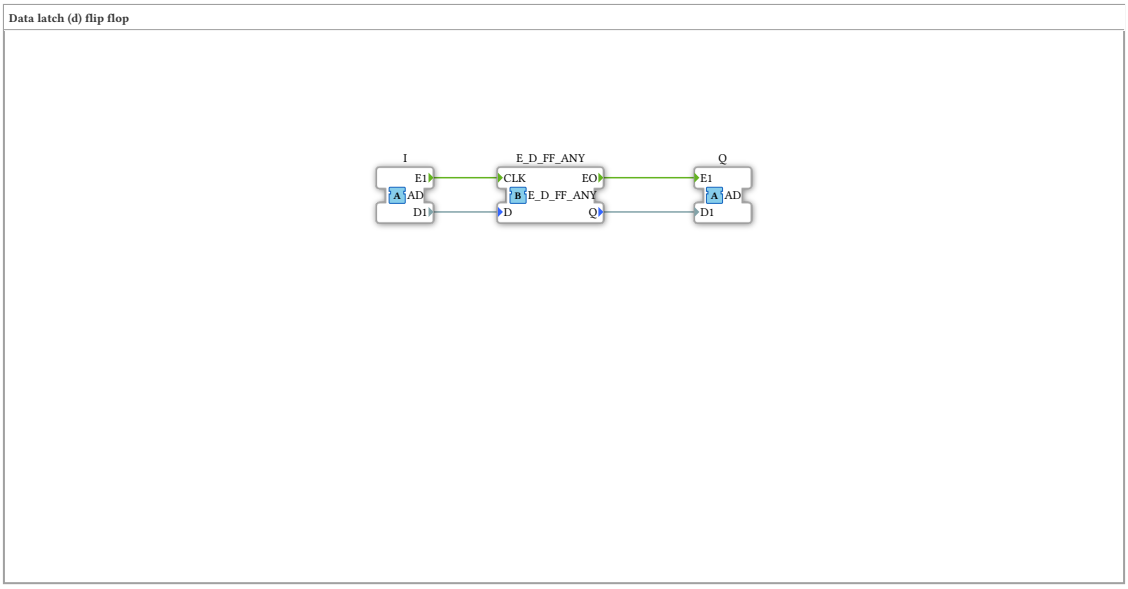
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AD_D_FF





• – ○ • – ○ • – ○ •

The function block **AD_D_FF** implements a **data latch (D flip-flop)**. It receives a data value upon an incoming event and holds this value until the next data transfer. The interface is defined exclusively via standardized **adapters**, which allows for flexible wiring in complex automation systems.

No direct event inputs. The function block uses only the **input adapter `I`**, which provides the clock signal via its internal event channel.

No direct event outputs. The FB uses only the **output adapter `Q`**, which signals via its internal event channel that a new data value is available.

No direct data inputs. The value to be latched is provided via the **data channel of the input adapter `I`**.

No direct data outputs. The latched value is provided via the **data channel of the output adapter `Q`**.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Adapter	Direction	Type	Description
`I` (Socket)	Input	`adapter::types::unidirectional::AD`	Provides the clock signal (E1) and the data value to be stored (D1).
`Q` (Plug)	Output	`adapter::types::unidirectional::AD`	Outputs the stored data value after the clock event (E1 as acknowledgment, D1 as data).

The adapters of type `unidirectional::AD` each define one event channel (`E1`) and one data channel (`D1`). The exact data types are defined by the adapter definition (usually `ANY`).

The adapters of type `unidirectional::AD` define one event channel (`E1`) and one data channel (`D1`). The exact data types are defined by the adapter definition (usually `ANY`).

Functionality

The function block (FB) internally uses the elementary block `E_D_FF_ANY` (from the IEC 61499 standard library). Its behavior is as follows:

1. **Upon each event at the clock input (CLK)**, the current data value at input `D` is transferred to output `Q`. This transfer is triggered by an edge (rising edge of the event).
2. Output `Q` remains unchanged until a new clock event arrives.

The connection in the FB network:

- The event `I.E1` is connected to `CLK`.
- The date `I.D1` is linked to `D`.
- The event output `EO` of the internal function block is forwarded to `Q.E1`.
- The data output `Q` of the internal function block is forwarded to `Q.D1`.

Thus, with each event at the input adapter, the incoming data value is received and immediately output via the output adapter. The value is retained until the next event.

- **Pure Adapter Interface:** The function block does not have traditional event/data ports but communicates exclusively via adapters. This enables its use in systems with standardized connections (e.g., via specific fieldbus adapter profiles).
- **Type Transparency:** Since the adapters are generic (`ANY`), the function block (FB) can work with different data types – provided that both the input and output adapters use the same data type.
- **No Internal State Machine:** The FB implements the flip-flop behavior through the pure data/event wiring of the internal `E_D_FF_ANY`. It does not have its own state machine.

The FB does not have an explicit state machine. The internal block `E_D_FF_ANY` internally stores the saved value (state). However, from the perspective of event processing, the FB is stateless: Every event at the input immediately triggers an output.

- **Signal Buffering:** A data value should be "frozen" upon a specific event and made available for subsequent processing steps.
- **Standardized data coupling:** In architectures that rely on adapter interfaces (e.g., in agricultural machinery), the function block can be used as a universal latch component.
- **RS flip-flop (AD_RS_FF):** Stores two states (Set/Reset) and requires two event/data channels. The AD_D_FF operates with only one clock signal and one data point – simpler and more deterministic.
- **E_D_FF_ANY (direct):** Offers the same functionality, but with classic event/data ports instead of adapters. The AD_D_FF encapsulates this logic in an adapter interface, which facilitates reuse in adapter-based environments.
- **Toggle FF (AD_T_FF)** or **JK FF:** More complex flip-flops with multiple inputs; the AD_D_FF is the simplest form of a data-controlled memory.

The **AD_D_FF** is a lightweight, adapter-based D flip-flop device for the 4diac IDE. It takes a data value on each clock event and outputs it. Thanks to its simple adapter interfaces, it integrates seamlessly into architectures that rely on standardized connections. Its straightforward functionality makes it the ideal choice for data buffering and synchronization tasks without the need for additional logic.

The **AD_D_FF** is a lightweight, adapter-based D flip-flop device for the 4diac IDE. —

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

State Overview

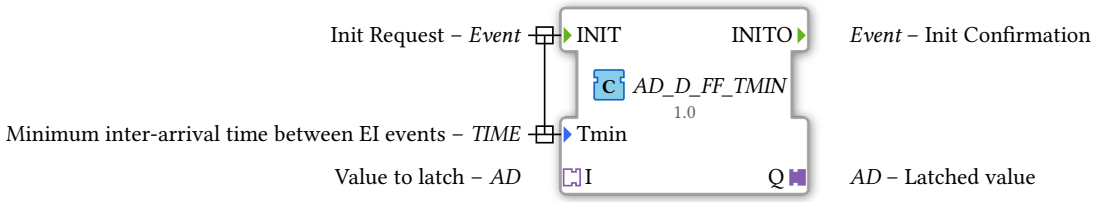
Application Scenarios

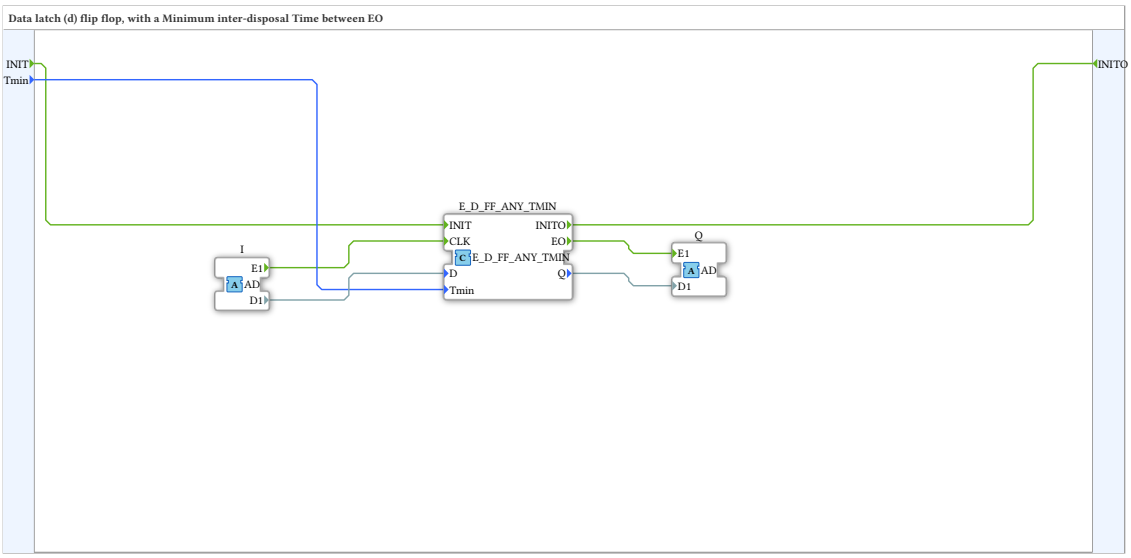
Comparison with Similar Function Blocks

Conclusion

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

AD_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AD_D_FF_TMIN** implements a data-retaining flip-flop (D-latch) based on adapters with a unidirectional interface. It features a configurable minimum inter-disposal time (MDR) between two consecutive events, making it suitable for time-critical signal processing.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request (Passing the Minimum Time)

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirmation

Data Inputs

Name	Data Type	Comment
Tmin	TIME	Minimum time between two consecutive events on socket *I*

Data Outputs

No direct data outputs are available; output is exclusively via the adapter *Q*.

Adapters

Type	Direction	Name	Comment
adapter::types::unidirectional::AD	**Plug** (Output)	Q	Returns the latched value
adapter::types::unidirectional::AD	**Socket** (Input)	I	Receives the value to be latched and the clock event

Functionality

This component encapsulates an internal instance of the function block `iec61499::events::E_D_FF_ANY_TMIN`. The socket *I* returns both the clock event (via the adapter event output `E1`) and the data value (via the adapter data output `D1`). At each clock cycle, the internal flip-flop updates the current value from `I.D1` to its output `Q`. This signal is output via the plug *Q* (event `E1` and data `D1`).

The minimum dwell time `Tmin` is passed to the internal block during initialization and limits the frequency of clock events: Events arriving before the elapsed time of `Tmin` since the last clock cycle are suppressed.

Technical Features

- **Adapter-based I/O**: The block uses unidirectional adapters (type `adapter::types::unidirectional::AD`) that combine events and data in a structured interface.
- **Event-driven minimum dwell time**: The configurable time `Tmin` prevents excessively fast clocking and protects downstream logic from overload.
- **Reuse**: The internal logic is implemented as a separate function block (`E_D_FF_ANY_TMIN`), which facilitates modular maintenance and testing.

State Overview

The function block itself does not have an explicit state machine (no ECC definition). Its behavior is entirely determined by the embedded function block `E_D_FF_ANY_TMIN`. This operates as a **D flip-flop with a time filter**:

- **Idle State**: Waiting for the next clock event.
- **Clock Processing**: Checking whether the time interval `Tmin` has elapsed since the last clock cycle. If yes → Data transfer and generation of an output event. If no → Event ignored.
- **Initialization State**: Only when `INIT` is `Tmin` set and the internal timer reset.

Application Scenarios

- **Data Acquisition with Minimum Interval**:

Sensor data should only be acquired if a specific time interval from the previous value is maintained (e.g., debouncing or sampling rate limiting).

- **Pulse Shaping / Trigger Management**:

The function block can be used as a "debounced" flip-flop in controllers to smooth signal edges and filter out interference pulses.

- **Adapter-Oriented Bus Architectures**:

Thanks to the adapter interface, the function block can be seamlessly integrated into existing unidirectional data flows, e.g., in sensor-actuator networks.

Comparison with Similar Function Blocks

Function Block	Feature	Difference
E_D_FF	Standard D Flip-Flop	No time constraint; every clock event is processed.
E_CTD / E_CTU	Counter	Different basic function; counts events.
AD_D_FF_TMIN	D Flip-Flop with *Minimum Inter-disposal Time*	Prevents excessively fast clock sequences through configurable lock time.

Conclusion

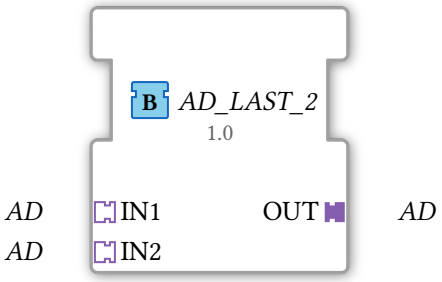
The **AD_D_FF_TMIN** extends the classic D Flip-Flop with a time-based filter function, which is configured via the parameter `Tmin`. Its adapter interface allows for clean encapsulation of events and data in a single connector, simplifying integration into complex automation systems. The module is ideal for applications requiring a minimal event distance.

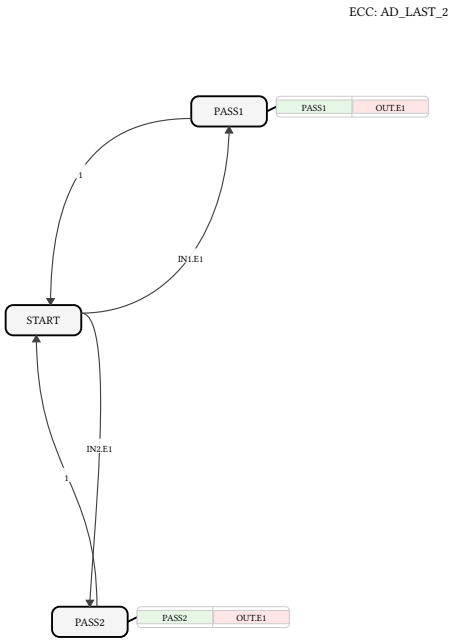
Related topic subpages on ms-muc-docs.de

- E_CTU Event Counter Module on ms-muc-docs.de

]

AD_LAST_2





Introduction

The AD_LAST_2 function block merges two same-type DWORD adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: DWORD adapter (unidirectional) - first signal source
- **IN2**: DWORD adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: DWORD adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

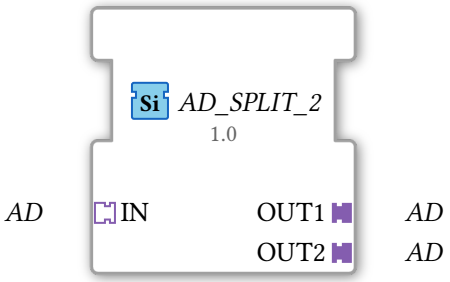
Application Scenarios

- Merging two equally-valid DWORD signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same DWORD value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AD_LAST_2 is the inverse of `AD_SPLIT_2`: AD_SPLIT_2 distributes 1 input to 2 outputs, AD_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

AD_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The AD_SPLIT_2 function block is used to split an incoming unidirectional adapter (AD) into two identical output adapters. It is implemented as a generic function block and allows for the simple branching of an adapter signal into two parallel paths.

Interface Structure

Event Inputs
None

Event Outputs
None

Data Inputs
None

Data Outputs
None

Adapter

Name	Type	Direction
IN	adapter::types::unidirectional::AD	Socket (Input)
OUT1	adapter::types::unidirectional::AD	Plug (Output)
OUT2	adapter::types::unidirectional::AD	Plug (Output)

Functionality
The function block receives a unidirectional adapter via socket `IN`. All data and events transmitted via this adapter are passed unchanged to both output adapters `OUT1` and `OUT2`. No processing or filtering takes place – the signals are simply duplicated. The distribution occurs without any time delay and without internal buffering.

Technical Features
- **Generic Implementation**: The function block uses the generics mechanisms of Eclipse 4diac (`GenericClassName`). This allows the specific adapter type to be parameterized at design time or runtime, as long as it conforms to the adapter schema `unidirectional::AD`.
- **Type purity**: The output adapters are of the same type as the input adapter; no type conversion takes place.
- **No state logic**: The function block has no internal state machine (ECC) and reacts to adapter communication in an event-driven manner.

State overview
The function block has no explicit states. It operates continuously and immediately forwards each incoming signal to both outputs.

Application scenarios
- Splitting a sensor data stream to two independent control or monitoring units.
- Parallelizing adapter signals in industrial automation systems (e.g., distributing a fieldbus adapter value across two control loops).
- Test and simulation setups where a signal must be sent to different devices simultaneously.

Comparison with Similar Function Blocks
- **AD_SPLIT_3, AD_SPLIT_4**: Extensions with multiple outputs, based on the same mechanism.
- **AD_MERGE**: Combines multiple adapter inputs into one output – the inverse function of AD_SPLIT_2.
- **AD_SELECT**: Selects one of several adapter inputs and passes it on, but requires control logic.

Change Detection
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AD_SPLIT_2 is a simple yet effective function block for signal distribution in adapter-based architectures. Its generic nature and lack of state logic make it easy to configure and resource-efficient. It is particularly suitable for modular automation projects where an adapter signal needs to be distributed to multiple downstream components.

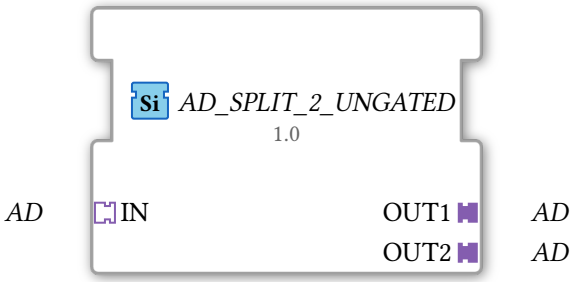
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AD_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The AD_SPLIT_2_UNGATED function block is used to split an incoming unidirectional adapter (AD) into two identical output adapters. It is implemented as a generic function block and allows for the simple branching of an adapter signal into two parallel paths.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Name	Type	Direction
IN	adapter::types::unidirectional::AD	Socket (Input)
OUT1	adapter::types::unidirectional::AD	Plug (Output)
OUT2	adapter::types::unidirectional::AD	Plug (Output)

Functionality

The function block receives a unidirectional adapter via socket `IN`. All data and events transmitted via this adapter are passed unchanged to both output adapters `OUT1` and `OUT2`. No processing or filtering takes place – the signals are simply duplicated. The distribution occurs without any time delay and without internal buffering.

Technical Features

- **Generic Implementation**: The function block uses the generics mechanisms of Eclipse 4diac (`GenericClassName`). This allows the specific adapter type to be parameterized at design time or runtime, as long as it conforms to the adapter schema `unidirectional::AD`.
- **Type purity**: The output adapters are of the same type as the input adapter; no type conversion takes place.
- **No state logic**: The function block has no internal state machine (ECC) and reacts to adapter communication in an event-driven manner.

State overview

The function block has no explicit states. It operates continuously and immediately forwards each incoming signal to both outputs.

Application scenarios

- Splitting a sensor data stream to two independent control or monitoring units.
- Parallelizing adapter signals in industrial automation systems (e.g., distributing a fieldbus adapter value across two control loops).
- Test and simulation setups where a signal must be sent to different devices simultaneously.

Comparison with Similar Function Blocks

- **AD_SPLIT_3, AD_SPLIT_4**: Extensions with multiple outputs, based on the same mechanism.
- **AD_MERGE**: Combines multiple adapter inputs into one output – the inverse function of AD_SPLIT_2_UNGATED.
- **AD_SELECT**: Selects one of several adapter inputs and passes it on, but requires control logic.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

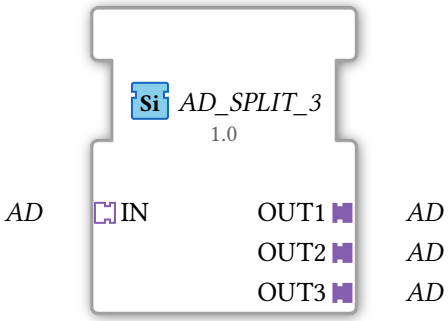
The AD_SPLIT_2_UNGATED is a simple yet effective function block for signal distribution in adapter-based architectures. Its generic nature and lack of state logic make it easy to configure and resource-efficient. It is particularly suitable for modular automation projects where an adapter signal needs to be distributed to multiple downstream components.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AD_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AD_SPLIT_3` serves as a generic splitter for a unidirectional adapter (AD). It receives an incoming AD and forwards it to three separate outputs. This allows multiple uses of the same data or signal path.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Direction	Name	Description
Socket (Input)	Input	IN	Unidirectional adapter input, which is distributed to the three outputs.
Plug (Output)	Output	OUT1	First output with duplicated adapter signal.
Plug (Output)	Output	OUT2	Second output with duplicated adapter signal.
Plug (Output)	Output	OUT3	Third output with duplicated adapter signal.

All adapters are of type `adapter::types::unidirectional::AD`.

Functionality

The module functions as a pure splitter: The adapter (data or event stream) connected to socket `IN` is identically duplicated to the three plugs `OUT1`, `OUT2`, and `OUT3`. Changes or events at the input affect all three outputs instantly. No transformation, buffering, or logic is performed – signal distribution is transparent.

Technical Features

- **Generic Type**: The function block is designed as a generic adapter splitter (`GenericClassName = 'GEN_AD_SPLIT'`) and can be used with any unidirectional adapter of type `adapter::types::unidirectional::AD`.
- **Pure Adapter Interface**: The function block has neither event nor data inputs/outputs in the traditional sense, but communicates exclusively via adapter plugs and sockets.
- **Unidirectional**: Communication occurs only in one direction (from the socket to the plugs), therefore no feedback or handshake mechanism is required.
- **No Runtime Dependence**: Distribution occurs instantaneously and without delay.

State Overview

The `AD_SPLIT_3` does not have its own state diagram (ECC). The outputs follow the input without any internal logic or memory behavior. Therefore, there is only a single, passive state in which the current input is constantly passed on to the outputs.

Application Scenarios

1. **Signal Fan-Out**: A sensor signal or a command (e.g., control word or status) is to be sent to multiple parallel consumers.
2. **Logging**: A data stream is processed once and then forwarded a second time for recording or monitoring.

3. **Redundancy**: A critical adapter path is distributed across multiple independent systems.
4. **Test Environments**: In simulation or test setups, a signal can be mirrored to multiple test points.

Comparison with Similar Function Blocks

- **AD_SPLIT_2**: Distributes the input to only two outputs. `AD_SPLIT_3` extends this to three outputs.
- **AD_MERGE**: Combines multiple adapter inputs into one output – functionally the opposite.
- **Specific Data Splitters**: Function blocks like `SPLIT_INT` or `SPLIT_BOOL` operate at the data level, not the adapter level. `AD_SPLIT_3` is designed for complete adapter structures (data plus events).

Change Detection

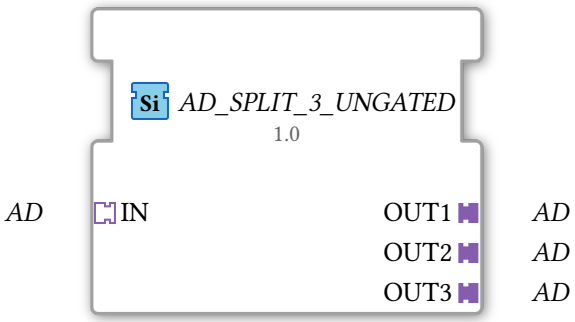
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AD_SPLIT_3` is a simple yet useful generic function block for multiplying unidirectional adapters. Due to its pure adapter interface and lack of logic, it is particularly suitable for modular architectures where a signal needs to be distributed to multiple receivers without introducing additional complexity.

AD_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AD_SPLIT_3_UNGATED` serves as a generic splitter for a unidirectional adapter (AD). It receives an incoming AD and forwards it to three separate outputs. This allows multiple uses of the same data or signal path.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Direction	Name	Description
Socket (Input)	Input	IN	Unidirectional adapter input, which is distributed to the three outputs.
Plug (Output)	Output	OUT1	First output with duplicated adapter signal.
Plug (Output)	Output	OUT2	Second output with duplicated adapter signal.
Plug (Output)	Output	OUT3	Third output with duplicated adapter signal.

All adapters are of type `adapter::types::unidirectional::AD`.

Functionality

The module functions as a pure splitter: The adapter (data or event stream) connected to socket `IN` is identically duplicated to the three plugs `OUT1`, `OUT2`, and `OUT3`. Changes or events at the input affect all three outputs instantly. No transformation, buffering, or logic is performed – signal distribution is transparent.

Technical Features

- **Generic Type**: The function block is designed as a generic adapter splitter (`GenericClassName = 'GEN_AD_SPLIT'`) and can be used with any unidirectional adapter of type `adapter::types::unidirectional::AD`.
- **Pure Adapter Interface**: The function block has neither event nor data inputs/outputs in the traditional sense, but communicates exclusively via adapter plugs and sockets.
- **Unidirectional**: Communication occurs only in one direction (from the socket to the plugs), therefore no feedback or handshake mechanism is required.
- **No Runtime Dependence**: Distribution occurs instantaneously and without delay.

State Overview

The `AD_SPLIT_3_UNGATED` does not have its own state diagram (ECC). The outputs follow the input without any internal logic or memory behavior. Therefore, there is only a single, passive state in which the current input is constantly passed on to the outputs.

Application Scenarios

1. **Signal Fan-Out**: A sensor signal or a command (e.g., control word or status) is to be sent to multiple parallel consumers.
2. **Logging**: A data stream is processed once and then forwarded a second time for recording or monitoring.
3. **Redundancy**: A critical adapter path is distributed across multiple independent systems.
4. **Test Environments**: In simulation or test setups, a signal can be mirrored to multiple test points.

Comparison with Similar Function Blocks

- **AD_SPLIT_2**: Distributes the input to only two outputs. `AD_SPLIT_3_UNGATED` extends this to three outputs.
- **AD_MERGE**: Combines multiple adapter inputs into one output – functionally the opposite.
- **Specific Data Splitters**: Function blocks like `SPLIT_INT` or `SPLIT_BOOL` operate at the data level, not the adapter level. `AD_SPLIT_3_UNGATED` is designed for complete adapter structures (data plus events).

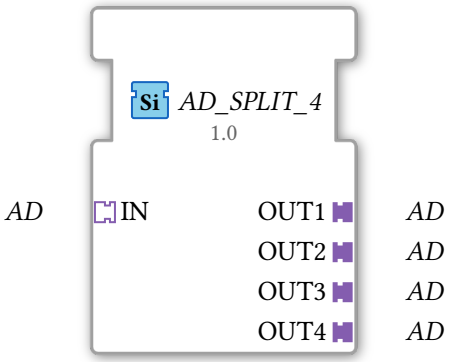
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AD_SPLIT_3_UNGATED` is a simple yet useful generic function block for multiplying unidirectional adapters. Due to its pure adapter interface and lack of logic, it is particularly suitable for modular architectures where a signal needs to be distributed to multiple receivers without introducing additional complexity.

AD_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_SPLIT_4** is used to distribute an incoming unidirectional AD adapter (adapter definition) to four identical outputs. It is a generic function block specifically designed for splitting a single AD signal into up to four separate paths.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available. Data transmission occurs exclusively via the incoming AD adapter.

Data Outputs

No data outputs available. Output occurs exclusively via the four outgoing AD adapters.

Adapters

This component uses only adapters of type `adapter::types::unidirectional::AD`:

- **Socket (Input)**

IN` – A unidirectional analog-to-digital input that receives the signal to be split.

- **Plugs (Outputs)**

OUT1, OUT2, OUT3, OUT4` – Four unidirectional analog-to-digital outputs that pass the signal received from the input onward identically.

Functionality

The **AD_SPLIT_4** function block copies the AD data stream arriving at its socket `IN` (without event handling) and makes it available in parallel at all four plugs `OUT1` to `OUT4`. Distribution occurs without delay or buffering – any change at the input is immediately propagated to all outputs. The function block is designed generically, so the specific data type of the AD adapter is only determined at runtime (GenericClassName = `'GEN_AD_SPLIT'`).

Technical Features

- **Generic Type**: The function block is implemented as a generic function block. The actual AD data type is determined by the attribute `eclipse4diac::core::GenericClassName` (`'GEN_AD_SPLIT'`).
- **Unidirectional Communication**: The adapter used, `adapter::types::unidirectional::AD`, only allows data transmission from the socket to the plugs – feedback is not supported.
- **No Event Control**: The module does not require events, as data transmission is purely data-driven (or by the underlying adapter logic).

State Overview

Since the module has no event inputs/outputs, there is no state machine. The behavior is deterministic: The outputs always reflect the current state of the input.

Application Scenarios

- **Signal Distribution**: When a single A/D signal (e.g., a measured value or a control signal) needs to be forwarded to multiple subsequent function blocks or hardware components.
- **Monitoring and Control**: Simultaneous monitoring of a signal in different logic units without requiring multiple copies of the signal source.
- **Redundancy Paths**: Splitting a signal for parallel, independent processing paths.

Comparison with Similar Function Blocks

- **AD_SPLIT_2**, **AD_SPLIT_8**: Related function blocks with identical functionality, but for two or eight outputs, respectively. The choice depends on the number of distributions required.
- **AD_MERGE_4**: A counterpart that combines multiple AD inputs into a single output, as opposed to splitting them.
- **Pure Data Split Function Blocks (e.g., D_SPLIT_4)**: These split individual data values (e.g., INT, REAL), while AD_SPLIT_4 is specifically designed for the complex AD adapter type.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AD_SPLIT_4** function block is a compact, generic tool for easily replicating a unidirectional AD signal. Its clean, event-free interface makes it particularly suitable for data flow-oriented applications in automation technology where a signal needs to be distributed to multiple receivers. Its generic design allows it to be flexibly adapted to various adapter data types.

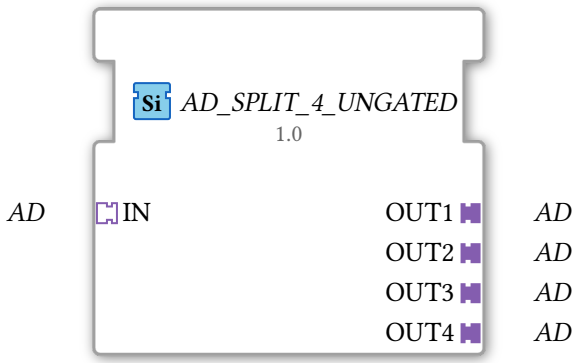
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AD_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_SPLIT_4_UNGATED** is used to distribute an incoming unidirectional AD adapter (adapter definition) to four identical outputs. It is a generic function block specifically designed for splitting a single AD signal into up to four separate paths.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available. Data transmission occurs exclusively via the incoming AD adapter.

Data Outputs

No data outputs available. Output occurs exclusively via the four outgoing AD adapters.

Adapters

This component uses only adapters of type `adapter::types::unidirectional::AD`:

- **Socket (Input)**

IN` – A unidirectional analog-to-digital input that receives the signal to be split.

- **Plugs (Outputs)**

OUT1, OUT2, OUT3, OUT4` – Four unidirectional analog-to-digital outputs that pass the signal received from the input onward identically.

Functionality

The **AD_SPLIT_4_UNGATED** function block copies the AD data stream arriving at its socket `IN` (without event handling) and makes it available in parallel at all four plugs `OUT1` to `OUT4`. Distribution occurs without delay or buffering – any change at the input is immediately propagated to all outputs. The function block is designed generically, so the specific data type of the AD adapter is only determined at runtime (GenericClassName = `'GEN_AD_SPLIT'`).

Technical Features

- **Generic Type**: The function block is implemented as a generic function block. The actual AD data type is determined by the attribute `eclipse4diac::core::GenericClassName` (`'GEN_AD_SPLIT'`).
- **Unidirectional Communication**: The adapter used, `adapter::types::unidirectional::AD`, only allows data transmission from the socket to the plugs – feedback is not supported.
- **No Event Control**: The module does not require events, as data transmission is purely data-driven (or by the underlying adapter logic).

State Overview

Since the module has no event inputs/outputs, there is no state machine. The behavior is deterministic: The outputs always reflect the current state of the input.

Application Scenarios

- **Signal Distribution**: When a single A/D signal (e.g., a measured value or a control signal) needs to be forwarded to multiple subsequent function blocks or hardware components.
- **Monitoring and Control**: Simultaneous monitoring of a signal in different logic units without requiring multiple copies of the signal source.
- **Redundancy Paths**: Splitting a signal for parallel, independent processing paths.

Comparison with Similar Function Blocks

- **AD_SPLIT_2**, **AD_SPLIT_8**: Related function blocks with identical functionality, but for two or eight outputs, respectively. The choice depends on the number of distributions required.

- **AD_MERGE_4**: A counterpart that combines multiple AD inputs into a single output, as opposed to splitting them.
- **Pure Data Split Function Blocks (e.g., D_SPLIT_4)**: These split individual data values (e.g., INT, REAL), while AD_SPLIT_4_UNGATED is specifically designed for the complex AD adapter type.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

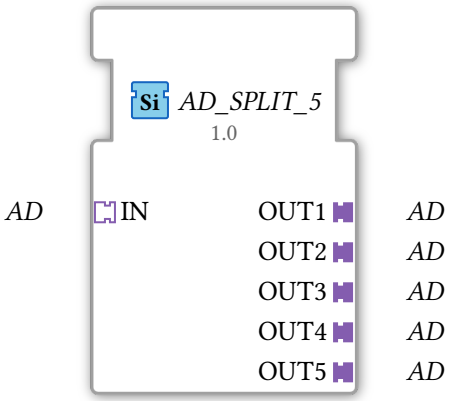
The **AD_SPLIT_4_UNGATED** function block is a compact, generic tool for easily replicating a unidirectional AD signal. Its clean, event-free interface makes it particularly suitable for data flow-oriented applications in automation technology where a signal needs to be distributed to multiple receivers. Its generic design allows it to be flexibly adapted to various adapter data types.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AD_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AD_SPLIT_5` is a generic function block that splits an incoming unidirectional adapter (type `adapter::types::unidirectional::AD`) into five identical output adapters. It serves to forward an adapter signal in parallel to multiple subsequent function blocks without requiring the input to be connected multiple times.

Interface Structure

Event Inputs

No direct event inputs are available. All communication takes place via the adapter socket.

Event Outputs

No direct event outputs are available. Outputs are exclusively via the adapter plugs.

Data Inputs

No direct data inputs are available. Data is received via the adapter socket.

Data Outputs

No direct data outputs are available. Data is provided via the adapter plugs.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AD`	`IN`	Socket (Input)	Incoming adapter signal, which is distributed to all outputs.
`adapter::types::unidirectional::AD`	`OUT1`	Plug (Output)	First outgoing adapter, identical to the input signal.
`adapter::types::unidirectional::AD`	`OUT2`	Plug (Output)	Second outgoing adapter.
`adapter::types::unidirectional::AD`	`OUT3`	Plug (Output)	Third outgoing adapter.
`adapter::types::unidirectional::AD`	`OUT4`	Plug (Output)	Fourth outgoing adapter.
`adapter::types::unidirectional::AD`	`OUT5`	Plug (Output)	Fifth outgoing adapter.

Functionality

This component does not have event or data ports in the conventional sense. Instead, an incoming adapter `IN` is passed through to all five outgoing adapters `OUT1` to `OUT5`. This means that changes or events transmitted via the input adapter are passed to all five outputs simultaneously. This multiplication occurs without any logical processing or delay.

- **Generic Type**: The function block is implemented as a generic component (attribute `eclipse4diac::core::GenericClassName = 'GEN_AD_SPLIT'`). This allows it to be used in various contexts with the adapter type `adapter::types::unidirectional::AD`.
- **No State Machine**: The block does not have an ECC (Execution Control Chart) and operates purely in a data flow-driven manner. It does not perform any data storage operations.
- **Unidirectional Adapters**: All adapters are unidirectional, meaning data flows in only one direction – from the socket to the plugs.

The function block has no internal states or steps. It is a pure splitter without its own behavior. Therefore, a state overview is not available.

- **Signal Distribution**: When an adapter signal is needed multiple times, e.g., to feed several parallel calculations or outputs.
- **Modular Chaining**: In complex automation systems where a sensor value (as an adapter) needs to be passed on to different subsystems.
- **Prototyping**: Rapid setup of test environments where a signal needs to reach multiple locations simultaneously.
- **AD_SPLIT_2, AD_SPLIT_3, AD_SPLIT_4**: The same functionality with two, three, or four outputs, respectively. `AD_SPLIT_5` is the five-output version.
- **Manual Wiring**: Without this block, the output of the source block would have to be connected multiple times, which is not directly possible in the 4diac IDE (a port can only have one connection). The splitter solves this problem.
- **Event-Based Splitters**: There are also splitters for events (e.g., `E_SPLIT`), but these are based on event and data ports. In contrast, `AD_SPLIT_5` operates exclusively at the adapter level.

The function block `AD_SPLIT_5` is a simple yet useful tool for multiplying adapter signals in the 4diac development environment. Its generic design and the absence of a state machine provide an efficient way to distribute an adapter signal across five parallel paths without requiring any code changes. It is ideal for modular and scalable automation applications.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

Change Detection

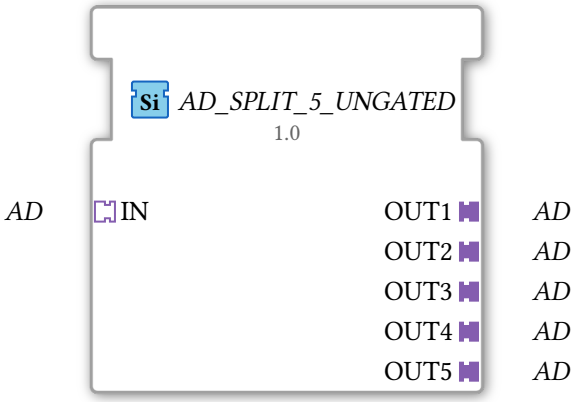
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

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

AD_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AD_SPLIT_5_UNGATED` is a generic function block that splits an incoming unidirectional adapter (type `adapter::types::unidirectional::AD`) into five identical output adapters. It serves to forward an adapter signal in parallel to multiple subsequent function blocks without requiring the input to be connected multiple times.

Interface Structure

Event Inputs

No direct event inputs are available. All communication takes place via the adapter socket.

Event Outputs

No direct event outputs are available. Outputs are exclusively via the adapter plugs.

Data Inputs

No direct data inputs are available. Data is received via the adapter socket.

Data Outputs

No direct data outputs are available. Data is provided via the adapter plugs.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AD`	`IN`	Socket (Input)	Incoming adapter signal, which is distributed to all outputs.
`adapter::types::unidirectional::AD`	`OUT1`	Plug (Output)	First outgoing adapter, identical to the input signal.
`adapter::types::unidirectional::AD`	`OUT2`	Plug (Output)	Second outgoing adapter.
`adapter::types::unidirectional::AD`	`OUT3`	Plug (Output)	Third outgoing adapter.
`adapter::types::unidirectional::AD`	`OUT4`	Plug (Output)	Fourth outgoing adapter.
`adapter::types::unidirectional::AD`	`OUT5`	Plug (Output)	Fifth outgoing adapter.

Functionality

This component does not have event or data ports in the conventional sense. Instead, an incoming adapter `IN` is passed through to all five outgoing adapters `OUT1` to `OUT5`. This means that changes or events transmitted via the input adapter are passed to all five outputs simultaneously. This multiplication occurs without any logical processing or delay.

- **Generic Type**: The function block is implemented as a generic component (attribute `eclipse4diac::core::GenericClassName = 'GEN_AD_SPLIT'`). This allows it to be used in various contexts with the adapter type `adapter::types::unidirectional::AD`.
- **No State Machine**: The block does not have an ECC (Execution Control Chart) and operates purely in a data flow-driven manner. It does not perform any data storage operations.
- **Unidirectional Adapters**: All adapters are unidirectional, meaning data flows in only one direction – from the socket to the plugs.

The function block has no internal states or steps. It is a pure splitter without its own behavior. Therefore, a state overview is not available.

- **Signal Distribution**: When an adapter signal is needed multiple times, e.g., to feed several parallel calculations or outputs.
- **Modular Chaining**: In complex automation systems where a sensor value (as an adapter) needs to be passed on to different subsystems.
- **Prototyping**: Rapid setup of test environments where a signal needs to reach multiple locations simultaneously.
- **AD_SPLIT_2, AD_SPLIT_3, AD_SPLIT_4**: The same functionality with two, three, or four outputs, respectively. `AD_SPLIT_5_UNGATED` is the five-output version.
- **Manual Wiring**: Without this block, the output of the source block would have to be connected multiple times, which is not directly possible in the 4diac IDE (a port can only have one connection). The splitter solves this problem.
- **Event-Based Splitters**: There are also splitters for events (e.g., `E_SPLIT`), but these are based on event and data ports. In contrast, `AD_SPLIT_5_UNGATED` operates exclusively at the adapter level.

The function block `AD_SPLIT_5_UNGATED` is a simple yet useful tool for multiplying adapter signals in the 4diac development environment. Its generic design and the absence of a state machine provide an efficient way to distribute an adapter signal across five parallel paths without requiring any code changes. It is ideal for modular and scalable automation applications.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

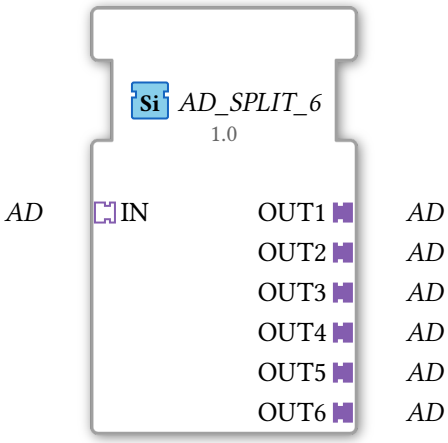
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

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

AD_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_SPLIT_6** distributes an incoming adapter of type `adapter::types::unidirectional::AD` to six separate output adapters of the same type. It is designed as a generic function block that implements a simple 1:6 split without additional logic or state management.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AD`	IN	Socket	Input adapter that distributes the signal to the six outputs.

`adapter::types::unidirectional::AD`	OUT1	Plug	First output adapter (identical to IN).
`adapter::types::unidirectional::AD`	OUT2	Plug	Second output adapter.
`adapter::types::unidirectional::AD`	OUT3	Plug	Third output adapter.
`adapter::types::unidirectional::AD`	OUT4	Plug	Fourth output adapter.
`adapter::types::unidirectional::AD`	OUT5	Plug	Fifth output adapter.
`adapter::types::unidirectional::AD`	OUT6	Plug	Sixth output adapter.

Functionality

The function block forwards the adapter received via socket **IN** unchanged to all six plugs (**OUT1** to **OUT6**). No data manipulation, filtering, or delay occurs. The adapter is multiplied by reference or by direct coupling (depending on the runtime environment) to the outputs. Thus, all connected subsequent function blocks receive the same adapter content.

Technical Features

- **Generic Function Block**: The attribute `eclipse4diac::core::GenericClassName` is set to `'GEN_AD_SPLIT'`, enabling reuse in various type configurations.
- **No Events or Data**: The function block operates exclusively via adapter interfaces; events or direct data connections are not required.
- **Simple Topology**: The 1:6 distribution is hard-coded and cannot be dynamically adjusted.

State Overview

The function block has no internal states or state machines. It behaves passively and continuously passes the input value to all outputs. There are no time constraints or mode switching.

Application Scenarios

- **Signal distribution** in control applications where an AD adapter interface needs to be distributed to multiple downstream components.
- **Test and simulation setups** for monitoring and processing a data stream in parallel.
- **Bus-like architectures** where multiple devices need to receive identical adapter information.

Comparison with Similar Components

Other split components such as `AD_SPLIT_2`, `AD_SPLIT_4`, or `AD_SPLIT_8` differ only in the number of outputs. `AD_SPLIT_6`, with six outputs, offers a middle ground between these variants. Components with additional logic (e.g., conditional distribution) also exist, but this is not the case here.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AD_SPLIT_6` is a simple yet useful generic function block for 1:6 distribution of an AD adapter. Its clear interface and lack of state logic make it easy to understand and efficient for distribution tasks in IEC 61499-based applications.

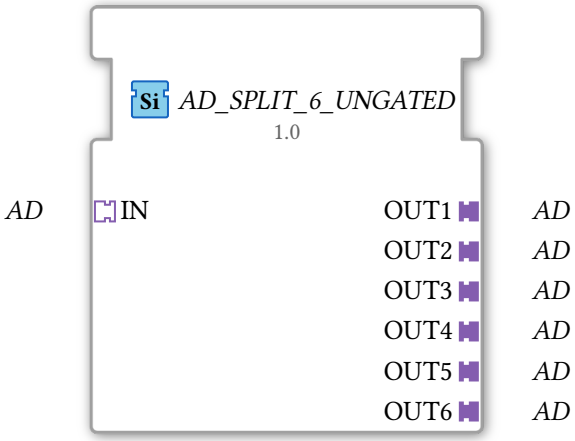
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AD_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/

frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_SPLIT_6_UNGATED** distributes an incoming adapter of type `adapter::types::unidirectional::AD` to six separate output adapters of the same type. It is designed as a generic function block that implements a simple 1:6 split without additional logic or state management.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AD`	IN	Socket	Input adapter that distributes the signal to the six outputs.
`adapter::types::unidirectional::AD`	OUT1	Plug	First output adapter (identical to IN).
`adapter::types::unidirectional::AD`	OUT2	Plug	Second output adapter.
`adapter::types::unidirectional::AD`	OUT3	Plug	Third output adapter.
`adapter::types::unidirectional::AD`	OUT4	Plug	Fourth output adapter.
`adapter::types::unidirectional::AD`	OUT5	Plug	Fifth output adapter.
`adapter::types::unidirectional::AD`	OUT6	Plug	Sixth output adapter.

Functionality

The function block forwards the adapter received via socket **IN** unchanged to all six plugs (**OUT1** to **OUT6**). No data manipulation, filtering, or delay occurs. The adapter is multiplied by reference or by

direct coupling (depending on the runtime environment) to the outputs. Thus, all connected subsequent function blocks receive the same adapter content.

Technical Features

- **Generic Function Block**: The attribute `eclipse4diac::core::GenericClassName` is set to `'GEN_AD_SPLIT'`, enabling reuse in various type configurations.
- **No Events or Data**: The function block operates exclusively via adapter interfaces; events or direct data connections are not required.
- **Simple Topology**: The 1:6 distribution is hard-coded and cannot be dynamically adjusted.

State Overview

The function block has no internal states or state machines. It behaves passively and continuously passes the input value to all outputs. There are no time constraints or mode switching.

Application Scenarios

- **Signal distribution** in control applications where an AD adapter interface needs to be distributed to multiple downstream components.
- **Test and simulation setups** for monitoring and processing a data stream in parallel.
- **Bus-like architectures** where multiple devices need to receive identical adapter information.

Comparison with Similar Components

Other split components such as `AD_SPLIT_2`, `AD_SPLIT_4`, or `AD_SPLIT_8` differ only in the number of outputs. `AD_SPLIT_6_UNGATED`, with six outputs, offers a middle ground between these variants. Components with additional logic (e.g., conditional distribution) also exist, but this is not the case here.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

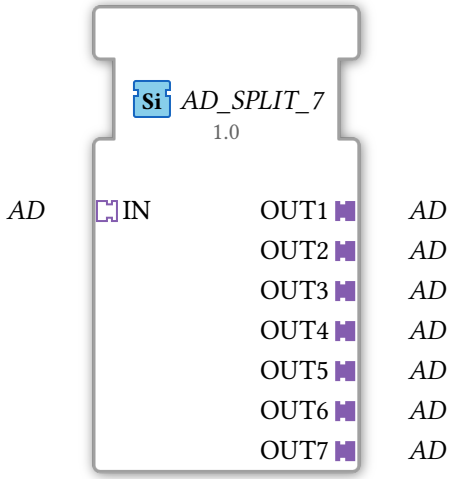
Conclusion

AD_SPLIT_6_UNGATED` is a simple yet useful generic function block for 1:6 distribution of an AD adapter. Its clear interface and lack of state logic make it easy to understand and efficient for distribution tasks in IEC 61499-based applications.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

AD_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_SPLIT_7** is a generic function block for splitting an incoming adapter of type `AD` (unidirectional) into seven separate output adapters. It allows a single adapter signal to be forwarded in parallel to multiple downstream components without modifying or buffering the data. The function block is specifically designed for use in control engineering (e.g., agricultural machinery) and is licensed under the Eclipse Public License 2.0.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AD`	Receives the adapter signal to be distributed.
Plug (Output) 1	`OUT1`	`adapter::types::unidirectional::AD`	First output – identical copy of the input signal.
Plug (Output) 2	`OUT2`	`adapter::types::unidirectional::AD`	Second output.
Plug (Output) 3	`OUT3`	`adapter::types::unidirectional::AD`	Third output.
Plug (Output) 4	`OUT4`	`adapter::types::unidirectional::AD`	Fourth output.
Plug (Output) 5	`OUT5`	`adapter::types::unidirectional::AD`	Fifth output.
Plug (Output) 6	`OUT6`	`adapter::types::unidirectional::AD`	Sixth output.
Plug (Output) 7	`OUT7`	`adapter::types::unidirectional::AD`	Seventh output.

Functionality

The function block forwards the adapter signal (type `AD`) present at socket `IN` unchanged to all seven plugs (`OUT1` … `OUT7`). No data manipulation, synchronization, or buffering takes place. The outputs are always simultaneous and identical to the input. The function block operates purely on a data flow basis without events or state logic.

–

Technical Features

- **Generic Function Block**: The function block is declared as a generic type (GenericClassName = `'GEN_AD_SPLIT'`), which allows reuse with different adapter variants.
- **Unidirectional Adapters**: All adapters used are unidirectional (direction from the socket to the plugs).
- **No Internal States**: There is no state machine – the outputs are continuously derived from the input.
- **Type Hash**: An empty type hash (`''`) is set, but can be used for version control if needed.

State Overview

The function block has no states or state machines. Its behavior is purely combinatorial: As soon as a valid signal is present at the input, it is immediately mirrored to all outputs.

Application Scenarios

- **Distributing Sensor Data**: A single sensor (e.g., speed, pressure) should supply several independent evaluation units in parallel.
- **Parallel Operation of Actuators**: A control signal is simultaneously transmitted to several identical actuators (e.g., valves, motors).
- **Redundancy Setup**: The same data stream is sent to multiple redundant processing paths.
- **Test and Simulation Environments**: A simulated adapter signal is distributed across various test modules.

Comparison with Similar Components

Component	Number of Outputs	Special Feature
`AD_SPLIT_2`	2	Simple 1:2 distribution
`AD_SPLIT_4`	4	Typical for 4-channel systems
`AD_SPLIT_7`	**7**	Extended distribution for seven parallel paths
`AD_MERGE`	–	Combines multiple inputs into one output

Unlike a merge function block, `AD_SPLIT_7` is used exclusively for duplicating a signal without combining data.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AD_SPLIT_7` is a simple yet useful function block for passive signal duplication in adapter-based 4diac applications. Its generic nature and independence from events make it ideal for pure dataflow architectures. For applications requiring a higher number of parallel outputs, the function block can be extended by adjusting the GenericClassName.

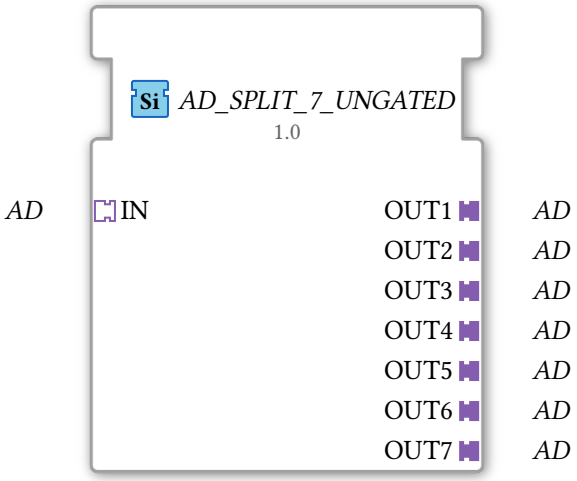
🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AD_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_SPLIT_7_UNGATED** is a generic function block for splitting an incoming adapter of type `AD` (unidirectional) into seven separate output adapters. It allows a single adapter signal to be forwarded in parallel to multiple downstream components without modifying or buffering the data. The function block is specifically designed for use in control engineering (e.g., agricultural machinery) and is licensed under the Eclipse Public License 2.0.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AD`	Receives the adapter signal to be distributed.
Plug (Output) 1	`OUT1`	`adapter::types::unidirectional::AD`	First output – identical copy of the input signal.
Plug (Output) 2	`OUT2`	`adapter::types::unidirectional::AD`	Second output.
Plug (Output) 3	`OUT3`	`adapter::types::unidirectional::AD`	Third output.
Plug (Output) 4	`OUT4`	`adapter::types::unidirectional::AD`	Fourth output.
Plug (Output) 5	`OUT5`	`adapter::types::unidirectional::AD`	Fifth output.
Plug (Output) 6	`OUT6`	`adapter::types::unidirectional::AD`	Sixth output.
Plug (Output) 7	`OUT7`	`adapter::types::unidirectional::AD`	Seventh output.

Functionality

The function block forwards the adapter signal (type `AD`) present at socket `IN` unchanged to all seven plugs (`OUT1` ... `OUT7`). No data manipulation, synchronization, or buffering takes place. The outputs are always simultaneous and identical to the input. The function block operates purely on a data flow basis without events or state logic.

–

Technical Features

- **Generic Function Block**: The function block is declared as a generic type (GenericClassName = `'GEN_AD_SPLIT'`), which allows reuse with different adapter variants.
- **Unidirectional Adapters**: All adapters used are unidirectional (direction from the socket to the plugs).
- **No Internal States**: There is no state machine – the outputs are continuously derived from the input.
- **Type Hash**: An empty type hash (`''`) is set, but can be used for version control if needed.

State Overview

The function block has no states or state machines. Its behavior is purely combinatorial: As soon as a valid signal is present at the input, it is immediately mirrored to all outputs.

Application Scenarios

- **Distributing Sensor Data**: A single sensor (e.g., speed, pressure) should supply several independent evaluation units in parallel.
- **Parallel Operation of Actuators**: A control signal is simultaneously transmitted to several identical actuators (e.g., valves, motors).
- **Redundancy Setup**: The same data stream is sent to multiple redundant processing paths.
- **Test and Simulation Environments**: A simulated adapter signal is distributed across various test modules.

Comparison with Similar Components

Component	Number of Outputs	Special Feature
`AD_SPLIT_2`	2	Simple 1:2 distribution
`AD_SPLIT_4`	4	Typical for 4-channel systems
`AD_SPLIT_7_UNGATED`	**7**	Extended distribution for seven parallel paths
`AD_MERGE`	–	Combines multiple inputs into one output

Unlike a merge function block, `AD_SPLIT_7_UNGATED` is used exclusively for duplicating a signal without combining data.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

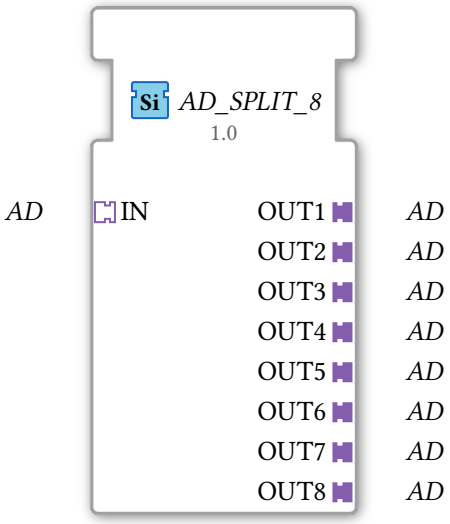
AD_SPLIT_7_UNGATED` is a simple yet useful function block for passive signal duplication in adapter-based 4diac applications. Its generic nature and independence from events make it ideal for pure dataflow architectures. For applications requiring a higher number of parallel outputs, the function block can be extended by adjusting the GenericClassName.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AD_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_SPLIT_8** is a generic function block for splitting an incoming unidirectional AD adapter signal into eight separate outputs. It enables the distribution of an analog or digital signal to multiple downstream components without altering the signal.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Direction	Name	Type	Description
Socket	**IN**	`adapter::types::unidirectional::AD`	Input Adapter – Signal to be Split
Plug	**OUT1**	`adapter::types::unidirectional::AD`	First Output
Plug	**OUT2**	`adapter::types::unidirectional::AD`	Second Output
Plug	**OUT3**	`adapter::types::unidirectional::AD`	Third Output
Plug	**OUT4**	`adapter::types::unidirectional::AD`	Fourth Output
Plug	**OUT5**	`adapter::types::unidirectional::AD`	Fifth Output
Plug	**OUT6**	`adapter::types::unidirectional::AD`	Sixth Output
Plug	**OUT7**	`adapter::types::unidirectional::AD`	Seventh Output
Plug	**OUT8**	`adapter::types::unidirectional::AD`	Eighth Output

Functionality

This function block forwards the signal present at socket **IN** unchanged to all eight plugs **OUT1** through **OUT8**. No signal processing, delay, or logical modification takes place. Distribution occurs continuously and without event triggering.

Technical Features

- **Generic Type**: Depending on the project definition, the adapter type `adapter::types::unidirectional::AD` can represent any unidirectional signal (e.g., analog value, byte, structure).
- **No Event Control**: The function block operates purely data-driven (adapter flow) – no INIT, RSP, or other events are required.
- **Compatibility**: Required import declarations for `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash` are included in the CompilerInfo.
- **Scalability**: By adjusting the generic parameters, similar function blocks with different output numbers can be created.

State Overview

This function block does not have a state machine (ECC). It operates statically and does not adapt its output behavior to different operating modes.

Application Scenarios

- **Sensor Distribution**: A single analog sensor value (e.g., temperature) is passed on to multiple control modules in parallel.
- **Signal Provision for Visualization and Control**: The same A/D signal is sent simultaneously to a higher-level controller and an operator panel.
- **Redundant Data Supply**: In safety-critical applications, the signal can be split across multiple independent paths.

Comparison with Similar Function Blocks

- **AD_SPLIT_2, AD_SPLIT_4** – Function blocks with similar functionality but fewer outputs.
- **AD_ROUTER** – A function block that selectively routes the incoming signal to one of several outputs.
- **AD_MULTIPLEXER** – Combines multiple inputs into one output (in reverse direction).

Unlike these function blocks, **AD_SPLIT_8** does not perform any selection – all outputs always receive the same signal.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AD_SPLIT_8 is a simple yet essential function block in IEC 61499 systems that implements signal distribution via adapter interfaces without additional logic or runtime costs. Its generic nature makes it flexible and facilitates the modular structuring of automation projects.

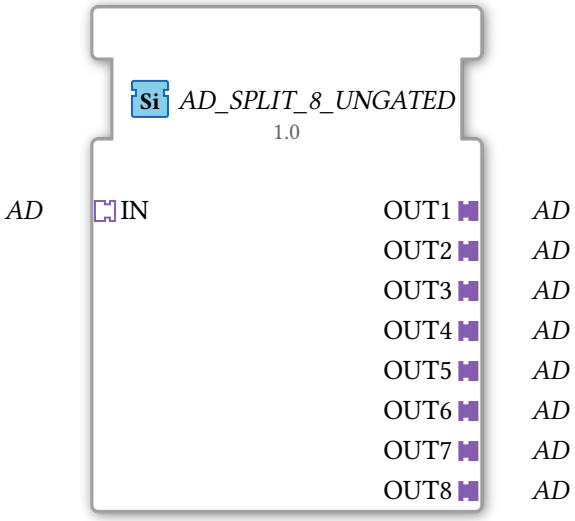
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AD_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_SPLIT_8_UNGATED** is a generic function block for splitting an incoming unidirectional AD adapter signal into eight separate outputs. It enables the distribution of an analog or digital signal to multiple downstream components without altering the signal.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Direction	Name	Type	Description
Socket	**IN**	`adapter::types::unidirectional::AD`	Input Adapter – Signal to be Split
Plug	**OUT1**	`adapter::types::unidirectional::AD`	First Output
Plug	**OUT2**	`adapter::types::unidirectional::AD`	Second Output
Plug	**OUT3**	`adapter::types::unidirectional::AD`	Third Output
Plug	**OUT4**	`adapter::types::unidirectional::AD`	Fourth Output
Plug	**OUT5**	`adapter::types::unidirectional::AD`	Fifth Output
Plug	**OUT6**	`adapter::types::unidirectional::AD`	Sixth Output
Plug	**OUT7**	`adapter::types::unidirectional::AD`	Seventh Output
Plug	**OUT8**	`adapter::types::unidirectional::AD`	Eighth Output

Functionality

This function block forwards the signal present at socket **IN** unchanged to all eight plugs **OUT1** through **OUT8**. No signal processing, delay, or logical modification takes place. Distribution occurs continuously and without event triggering.

Technical Features

- **Generic Type**: Depending on the project definition, the adapter type `adapter::types::unidirectional::AD` can represent any unidirectional signal (e.g., analog value, byte, structure).
- **No Event Control**: The function block operates purely data-driven (adapter flow) – no INIT, RSP, or other events are required.
- **Compatibility**: Required import declarations for `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash` are included in the CompilerInfo.
- **Scalability**: By adjusting the generic parameters, similar function blocks with different output numbers can be created.

State Overview

This function block does not have a state machine (ECC). It operates statically and does not adapt its output behavior to different operating modes.

Application Scenarios

- **Sensor Distribution**: A single analog sensor value (e.g., temperature) is passed on to multiple control modules in parallel.
- **Signal Provision for Visualization and Control**: The same A/D signal is sent simultaneously to a higher-level controller and an operator panel.
- **Redundant Data Supply**: In safety-critical applications, the signal can be split across multiple independent paths.

Comparison with Similar Function Blocks

- **AD_SPLIT_2, AD_SPLIT_4** – Function blocks with similar functionality but fewer outputs.
- **AD_ROUTER** – A function block that selectively routes the incoming signal to one of several outputs.
- **AD_MULTIPLEXER** – Combines multiple inputs into one output (in reverse direction).

Unlike these function blocks, **AD_SPLIT_8_UNGATED** does not perform any selection – all outputs always receive the same signal.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

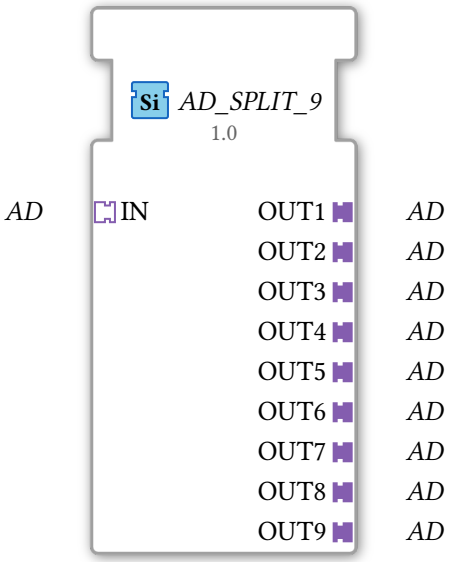
AD_SPLIT_8_UNGATED is a simple yet essential function block in IEC 61499 systems that implements signal distribution via adapter interfaces without additional logic or runtime costs. Its generic nature makes it flexible and facilitates the modular structuring of automation projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AD_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_SPLIT_9** distributes an incoming unidirectional adapter (AD) signal to nine identical outputs. It is implemented as a generic function block and is suitable for reusing an adapter signal in various downstream components.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Name	Type	Direction
IN	adapter::types::unidirectional::AD	Socket (Input)
OUT1	adapter::types::unidirectional::AD	Plug (Output)
OUT2	adapter::types::unidirectional::AD	Plug (Output)
OUT3	adapter::types::unidirectional::AD	Plug (Output)
OUT4	adapter::types::unidirectional::AD	Plug (Output)
OUT5	adapter::types::unidirectional::AD	Plug (Output)
OUT6	adapter::types::unidirectional::AD	Plug (Output)
OUT7	adapter::types::unidirectional::AD	Plug (Output)
OUT8	adapter::types::unidirectional::AD	Plug (Output)

OUT9	adapter::types::unidirectional::AD	Plug (Output)

Functionality

The function block (FB) performs a **1:n distribution** of the incoming adapter signal. Each of the nine output adapters (OUT1 … OUT9) provides exactly the same data as the input adapter (IN). Since the FB has neither event nor data inputs, signal transmission is purely structural: the connection between IN and all OUTs is defined at configuration time and transparently passed on at runtime.

Technical Features

- **Generic Block**: The FB carries the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AD_SPLIT'`, thus serving as a placeholder for type-specific instantiation.
- **Type Binding**: All adapters are of type `adapter::types::unidirectional::AD` – a standardized unidirectional adapter interface.
- **No Event Control**: Distribution occurs without time synchronization or events; changes to the input signal are immediately passed on to all outputs.

State Overview

The FB has no internal state machine and executes no sequential logic. Its behavior is **purely combinatorial**.

Application Scenarios

- **Signal Multiplexing**: An adapter provided by a sensor or controller is to be connected to multiple independent evaluation units (e.g., visualization, logging, alarm generation).
- **Modular Architecture**: In an IEC 61499-based application, a central data provider can distribute its information via `AD_SPLIT_9` to up to nine different submodules without duplicating the source logic.

Comparison with Similar Function Blocks

- **AD_SPLIT_2 / AD_SPLIT_4 / AD_SPLIT_8**: These function blocks offer the same functionality, but with fewer outputs (2, 4, 8). `AD_SPLIT_9` complements the portfolio for applications requiring exactly nine parallel outputs.
- **Other Split Function Blocks**: Data type-oriented split function blocks (e.g., for INTEGER or BOOL) split individual data values, while `AD_SPLIT_9` replicates complete adapter structures. The adapter can bundle multiple related data and events.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AD_SPLIT_9` is a simple yet useful generic function block for unidirectionally splitting an adapter signal into nine identical outputs. It helps avoid redundancy in system design and facilitates the modular structuring of IEC 61499 applications.

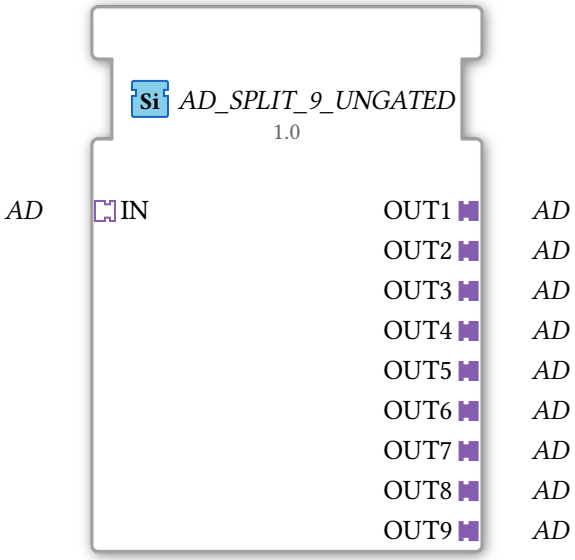
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AD_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_SPLIT_9_UNGATED** distributes an incoming unidirectional adapter (AD) signal to nine identical outputs. It is implemented as a generic function block and is suitable for reusing an adapter signal in various downstream components.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Name	Type	Direction
IN	adapter::types::unidirectional::AD	Socket (Input)
OUT1	adapter::types::unidirectional::AD	Plug (Output)
OUT2	adapter::types::unidirectional::AD	Plug (Output)
OUT3	adapter::types::unidirectional::AD	Plug (Output)
OUT4	adapter::types::unidirectional::AD	Plug (Output)
OUT5	adapter::types::unidirectional::AD	Plug (Output)
OUT6	adapter::types::unidirectional::AD	Plug (Output)
OUT7	adapter::types::unidirectional::AD	Plug (Output)
OUT8	adapter::types::unidirectional::AD	Plug (Output)

OUT9	adapter::types::unidirectional::AD	Plug (Output)

Functionality

The function block (FB) performs a **1:n distribution** of the incoming adapter signal. Each of the nine output adapters (OUT1 … OUT9) provides exactly the same data as the input adapter (IN). Since the FB has neither event nor data inputs, signal transmission is purely structural: the connection between IN and all OUTs is defined at configuration time and transparently passed on at runtime.

Technical Features

- **Generic Block**: The FB carries the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AD_SPLIT'`, thus serving as a placeholder for type-specific instantiation.
- **Type Binding**: All adapters are of type `adapter::types::unidirectional::AD` – a standardized unidirectional adapter interface.
- **No Event Control**: Distribution occurs without time synchronization or events; changes to the input signal are immediately passed on to all outputs.

State Overview

The FB has no internal state machine and executes no sequential logic. Its behavior is **purely combinatorial**.

Application Scenarios

- **Signal Multiplexing**: An adapter provided by a sensor or controller is to be connected to multiple independent evaluation units (e.g., visualization, logging, alarm generation).
- **Modular Architecture**: In an IEC 61499-based application, a central data provider can distribute its information via `AD_SPLIT_9_UNGATED` to up to nine different submodules without duplicating the source logic.

Comparison with Similar Function Blocks

- **AD_SPLIT_2 / AD_SPLIT_4 / AD_SPLIT_8**: These function blocks offer the same functionality, but with fewer outputs (2, 4, 8). `AD_SPLIT_9_UNGATED` complements the portfolio for applications requiring exactly nine parallel outputs.
- **Other Split Function Blocks**: Data type-oriented split function blocks (e.g., for INTEGER or BOOL) split individual data values, while `AD_SPLIT_9_UNGATED` replicates complete adapter structures. The adapter can bundle multiple related data and events.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AD_SPLIT_9_UNGATED` is a simple yet useful generic function block for unidirectionally splitting an adapter signal into nine identical outputs. It helps avoid redundancy in system design and facilitates the modular structuring of IEC 61499 applications.

🌐 Related topic subpages on ms-muc-docs.de

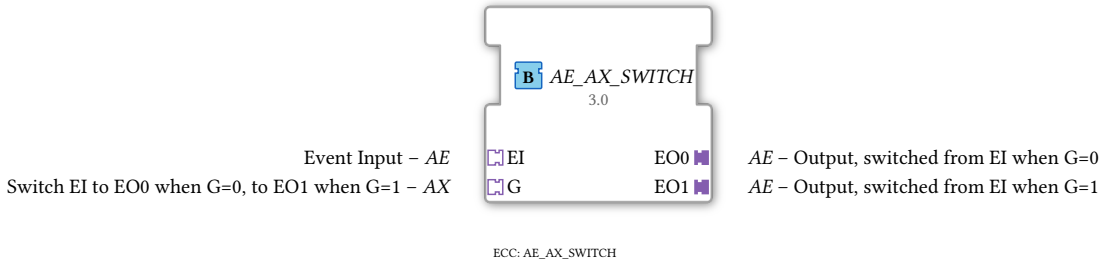
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

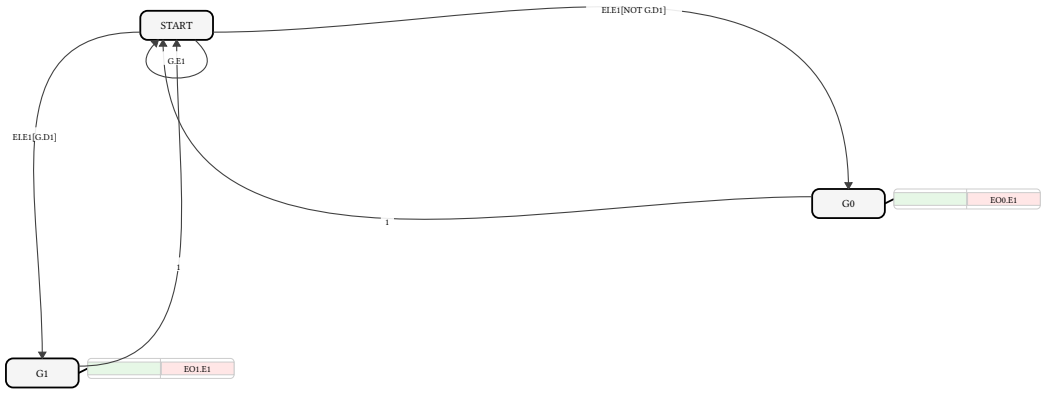
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AE_AX_SWITCH





• – ○ • – ○ • – ○ •

Introduction

The function block **AE_AX_SWITCH** implements event-driven switching (demultiplexing). An incoming event is forwarded to one of two event outputs – depending on the Boolean value of a separate data input.

Interface Structure

Event Inputs

The function block has no direct event inputs. Events are received via the **adapter `EI`** (type `adapter::types::unidirectional::AE`).

Event Outputs

The function block has no direct event outputs. Events are output via the **adapters `EO0` and `EO1`** (both type `adapter::types::unidirectional::AE`).

Data Inputs

- **`G`** (adapter type `adapter::types::unidirectional::AX`) – Boolean value that controls the switching.
- `G=0` ⟶ Event is routed to `EO0`
- `G=1` ⟶ Event is routed to `EO1`

Data Outputs

No direct data outputs. Output is exclusively via the event adapters.

Adapters

Name	Type	Direction	Description
EI	AE (Event Adapter)	Socket	Receives the event to be switched.
G	AX (Data Adapter)	Socket	Returns the switching state (boolean).
EO0	AE	Plug	Outputs the event if G=0 was present.
EO1	AE	Plug	Outputs the event if G=1 was present.

Functionality

The function block (FB) is always in state START. As soon as an event arrives at adapter EI (signal EI.E1), the current value of the Boolean adapter G.D1 is evaluated:

- **G.D1 = 0** → The FB switches to state G0 and sends an event via adapter EO0.E1. It then immediately returns to state START.
- **G.D1 = 1** → The function block (FB) switches to state G1 and sends an event via adapter EO1.E1. It then immediately returns to state START.

An additional event at adapter G (signal G.E1) does not trigger any action – the FB remains in state START. This prevents unwanted switching operations when the data value changes.

Technical Features

- **Adapter-based interface** – The FB uses loosely coupled adapters of types AE (event) and AX (data). This enables flexible integration with other function blocks in the 4diac framework.
- **Deterministic Switching** – The decision as to which output is activated is made in the same step as the event is received, thus eliminating any additional delay.
- **No Memory Behavior** – After outputting, the function block immediately returns to its initial state. The state G is not cached – it is read anew with each event.

State Overview

State	Description	Output Action
START	Waiting for an event at EI.	–
G0	Event from EI processed at G.D1=0.	EO0.E1
G1	Event from EI processed at G.D1=1.	EO1.E1

States G0 and G1 are exited immediately after output (automatic transition to START).

Application Scenarios

- **Event Routing** – Forwarding an event to different downstream logic branches, depending on a condition.
- **State-Dependent Control** – In automation systems, a sensor event can be sent to different processing blocks depending on the operating mode.
- **Multiplexing Reversal** – As a demultiplexer for asynchronous event streams, e.g., in communication protocols.

Comparison with Similar Function Blocks

- **E_SWITCH** – The standard function block E_SWITCH uses direct event inputs/outputs. AE_AX_SWITCH extends this concept with adapters, enabling looser coupling and reusability in different contexts.
- **E_DEMUX** – Other demultiplexer function blocks often use multiple inputs or states. AE_AX_SWITCH focuses on the simple, Boolean switching of a single event – without buffering or delay.

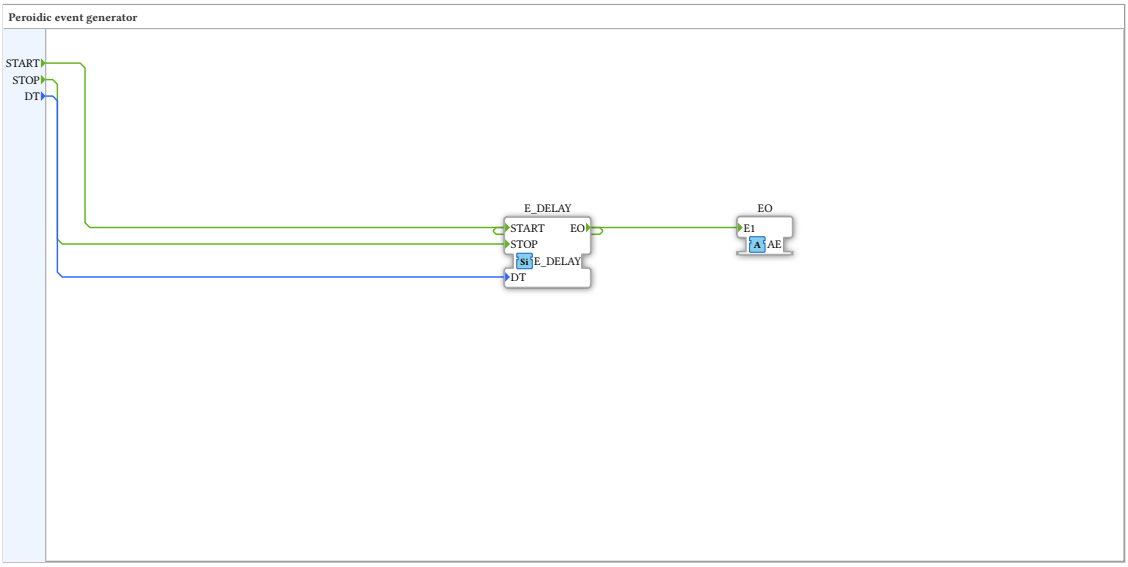
Conclusion

The AE_AX_SWITCH is a compact, efficient demultiplexer for events based on a Boolean switching signal. Its adapter-based interface allows for flexible integration into IEC 61499 systems, while its deterministic behavior ensures reliable routing. It is particularly suitable for applications where an event needs to be sent to different processing paths depending on the state of a condition.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

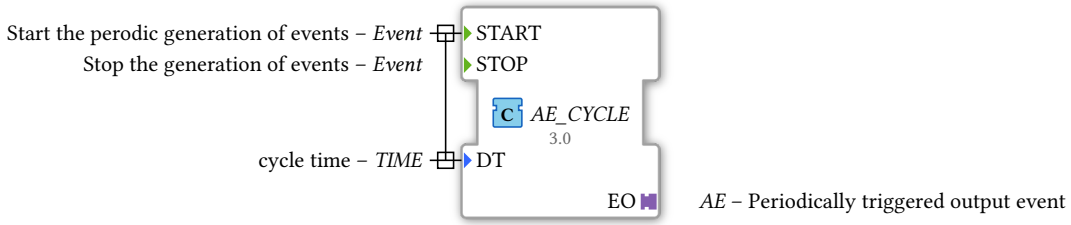
AE_CYCLE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_CYCLE** (Adapter Event Cycle) serves as a periodic event generator. It generates an output event at regular intervals, defined by a timer. Unlike the standard `E_CYCLE` block, this function block uses an adapter for signal output, making it particularly suitable for architectural patterns based on event adapters.



Interface Structure

Event Inputs

- **START**: Starts the periodic generation of events. When this event is triggered, the timer is activated.
- **STOP**: Stops the generation of events and interrupts the current cycle.

Event Outputs

- *No direct event outputs are available (see adapters).*

Data Inputs

- **DT** (TIME): The cycle time. This value determines the interval between two consecutive events.

Data Outputs

- *No data outputs available.*

Adapter

- **EO** (Plug): An adapter of type `adapter::types::unidirectional::AE`. The periodically generated event is output via this adapter (specifically via the adapter event `E1`).

Functionality

The **AE_CYCLE** is implemented as a Composite Function Block. Internally, its logic is based on the standard block `E_DELAY`.

1. **Start**: When the `START` event occurs, the internal `E_DELAY` block is started with a duration of `DT`.
2. **Delay & Trigger**: After the time specified by `DT` has elapsed, the internal module generates an output event.
3. **Feedback (Loop)**: This internal event performs two functions:

- It is forwarded to the adapter **EO** to output the signal.
- It is fed directly back to the `START` input of the internal `E_DELAY`.

1. **Cycle**: This feedback immediately restarts the timer, creating a continuous loop.
2. **Stop**: The `STOP` event immediately interrupts the internal `E_DELAY` module and ends the loop.

Technical Features

- **Encapsulation**: The function block encapsulates the logic of an oscillator circuit (feedback loop) and makes it available via a simplified interface.
- **Adapter Usage**: The use of the `AE` adapter (Adapter Event) enables clean separation and grouping of event flows, compatible with the `adapter::events::unidirectional` package.

State Overview

Since this is a Composite Function Block, it does not have its own state machine (ECC). The implicit state is determined by the internal `E_DELAY`:

- **Inactive**: After initialization or `STOP`.
- **Running**: After `START`, while the timer is running or being retriggered.

Application Scenarios

- **Heartbeat Signals**: Generating a status signal for higher-level systems or watchdogs.
- **Polling**: Periodically querying sensor data (trigger for `READ` operations).
- **Blinking**: Controlling indicator lights (in combination with toggle logic).
- **Timed Sequences**: Generating a clock signal for step sequences or periodic calculations.

⚖️ Comparison with Similar Function Blocks

- **E_CYCLE**: This is the direct standard counterpart from the IEC 61499 library. `E_CYCLE` offers the same functionality but uses a direct `EO` event output instead of an adapter.
- **E_DELAY**: `AE_CYCLE` is based on `E_DELAY`. While `E_DELAY` only forwards an event with a single delay, `AE_CYCLE`, due to its internal circuitry, ensures an endless repetition.
- **E_TRAIN**: Generates a limited number of pulses (pulse train), while `AE_CYCLE` runs indefinitely until stopped.

Conclusion

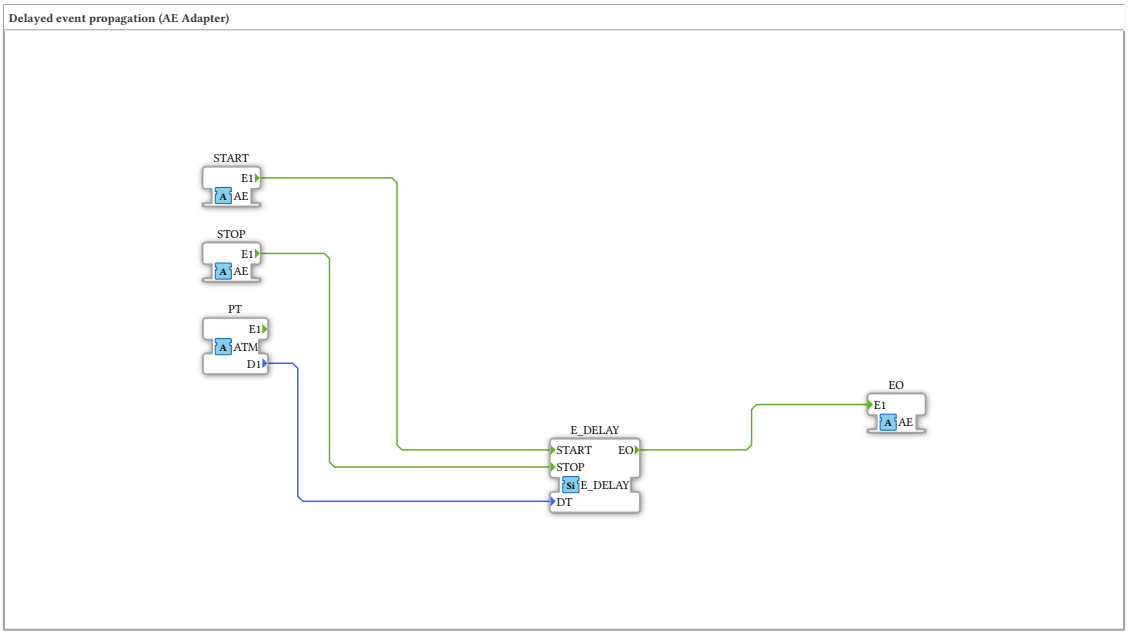
The **AE_CYCLE** is a useful utility block for developers working with event adapters within the 4diac environment. It abstracts the necessary feedback logic for generating periodic signals and integrates seamlessly into adapter-based software architectures.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

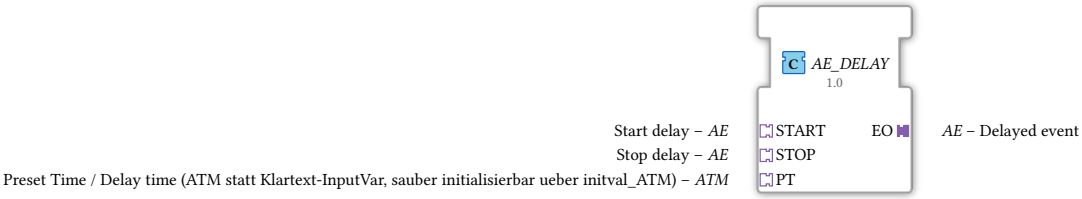
AE_DELAY



• — ○ • — ○ • — ○ •

Introduction

The **AE_DELAY** function block is a wrapper for the standard IEC 61499 function block `E_DELAY`, specifically designed for use with **Event Adapters (AE)**. It enables the delayed forwarding of events within an adapter-based architecture. Instead of conventional event pins, this function block uses adapter sockets and plugs to receive start and stop signals and output the delayed event.



Interface Structure

The function block encapsulates the functionality of an on-delay timer in adapter interfaces.

Event Inputs

This function block does not have direct event inputs. All input events are routed via the adapters `START` and `STOP`.

Event Outputs

This function block does not have direct event outputs at the main level. All output events are routed via the adapter `EO`.

Data Inputs

This function block does not have direct data inputs. The delay time is provided via the adapter `PT`.

Data Outputs

This function block does not have any data outputs.

Adapter

Name	Type	Kind	Comment

START	adapter::types::unidirectional::AE	Socket (Input)	Starts the time delay (internally triggers `START`).
STOP	adapter::types::unidirectional::AE	Socket (Input)	Stops/cancels the time delay (internally triggers `STOP`).
PT	adapter::types::unidirectional::ATM	Socket (Input)	**Preset Time**: Delay time (`PT.D1`, TIME).
EO	adapter::types::unidirectional::AE	Plug (Output)	**Event Output**: Outputs the event after the specified time elapses, `PT`.

Functionality

The **AE_DELAY** block acts as a bridge between the adapter world and the classic `E_DELAY` timer:

1. **Initialization:** The desired delay time is set via the adapter socket `PT` (data value `PT.D1`).
2. **Starting the Timer:** When an event (usually `E1`) arrives at the adapter socket **START**, it is internally forwarded to the `START` input of the embedded `E_DELAY` block. The timer then starts running.
3. **Time Expiration:** Once the time `PT` has elapsed, the internal timer generates an event. This is forwarded to the adapter plug **EO** (event `E1`).
4. **Stopping:** When an event arrives at the adapter socket **STOP**, the running timer is immediately stopped and reset. No event is generated at output `EO`.

Technical Details

- **Package Membership:** The function block is part of the package `adapter::events::unidirectional::timers`.
- **Encapsulation:** It contains an instance of `iec61499::events::E_DELAY`.
- **Adapter Type:** Uses the unidirectional event adapter `AE` (`adapter::types::unidirectional::AE`), which typically carries a single event (`E1`).

State Overview

The function block itself does not have an explicit state machine (ECC) because it is a composite network (FB). The behavior is entirely determined by the internal `E_DELAY`:

- **Idle:** Waiting for an event at the `START` adapter.
- **Timing:** After receiving a signal at `START`, the timer `PT` starts.
- **Triggered:** After `PT` expires, the signal is sent to `EO`, and the block returns to the idle state.
- **Reset:** A signal at `STOP` during the *Timing* phase immediately resets the block to the idle state.

Application Scenarios

- **Adapter-Based Controllers:** Ideal for systems that rely heavily on adapter connections to reduce the "spaghetti code" of individual connections.
- **Delayed Signal Chains:** Inserting delays into an event chain that is logically grouped by adapters.
- **Process Control:** Delayed start of units or processes controlled via standardized AE interfaces.

⚖️ Comparison with Similar Function Blocks

Function Block	Difference
E_DELAY	The standard function block with direct event pins (START, STOP, EO). **AE_DELAY** is the wrapper for this.
E_CYCLE	Generates periodic events, while **AE_DELAY** forwards an event only once with a delay.
AE_SPLIT	Used to split adapter events, but does not offer time delay.

Conclusion

The **AE_DELAY** is an indispensable tool for developers who want to leverage the advantages of 4diac adapters (reduced connection complexity, clear interface definitions) without sacrificing fundamental

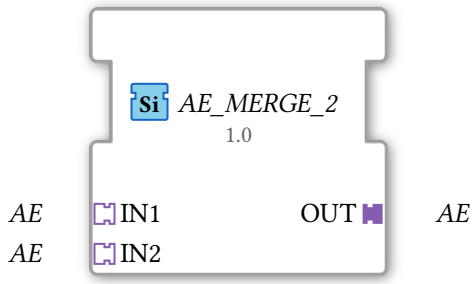
timing functions like delays. It seamlessly integrates the proven `E_DELAY` into modern, adapter-based application designs.

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AE_MERGE_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_MERGE_2** merges two unidirectional AE adapters (pure event, no payload data) into one common output adapter. It is the reverse of AE_SPLIT_2: instead of distributing one event to two outputs, it merges two independent event sources into a common output. The block is implemented as a generic FB (`GenericClassName: 'GEN_AE_MERGE'`).

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::AE`	First event source.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::AE`	Second event source.
Plug (Output)	`OUT`	`adapter::types::unidirectional::AE`	Merged event.

Functionality

Whenever an event arrives at `IN1` **or** `IN2`, it is forwarded unchanged to `OUT`. Since `AE` carries no data value, there is nothing to arbitrate when merging beyond the event itself – every single incoming event appears at the output immediately, regardless of which socket it came from. The order in which both sockets are checked for a simultaneous arrival is `IN1` before `IN2` in the implementation, which can matter if both events arrive within the same execution cycle.

Technical Features

- **Generic type**: Implemented via the generic base class `CGenUnidirectMergeBase` (N sockets, 1 plug) – the same C++ base underlying `GEN_ASR_MERGE` and `GEN_ASRT_MERGE`. The number of input sockets is fixed at compile time via the generic suffix (_2).
- **No states / algorithms**: Since `AE` carries no data value, there is no change detection and no ECC – the block is pure event forwarding.
- **No data loss on simultaneous events**: Unlike a plain data connection (which only allows one source), every event from `IN1` and `IN2` can be forwarded independently; no event is "lost" just because the other socket fired shortly before.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at `IN1` or `IN2` is forwarded to `OUT` immediately.

Application Scenarios

- **Event merging**: Two independent event sources (e.g. two buttons or two sensor triggers) should trigger the same follow-up action, without wiring the downstream logic twice.
- **Redundant trigger paths**: An event can be triggered via two independent sources (e.g. a manual and an automatic path) that both feed into the same output.
- **Simplifying networks** that would otherwise need two separate event connections to the same destination.

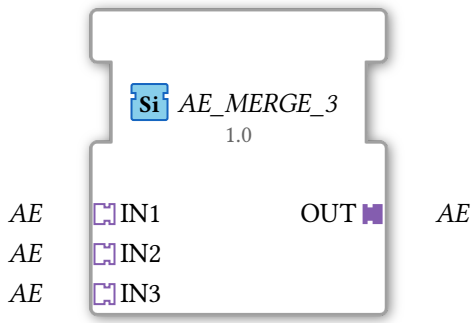
Comparison with Similar Components

- AE_SPLIT_2: the reverse direction – distributes one incoming event to two outputs, instead of merging two inputs.
- ASR_MERGE_2: the same merge logic for the set/reset event adapter `ASR` (2 events instead of 1).
- ASRT_MERGE_2: the same merge logic for the set/reset/toggle event adapter `ASRT` (3 events instead of 1).

Conclusion

AE_MERGE_2 is a minimal generic function block for merging two pure event sources into a common AE adapter output. Its lack of logic ensures minimal latency, while the generic implementation keeps it consistent with the other `GEN_*_MERGE` blocks in the family.

AE_MERGE_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_MERGE_3** merges 3 unidirectional AE adapters (pure event adapter, no payload data) into one common output adapter. It is the 3-input variant of AE_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_AE_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 3 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::AE`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::AE`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::AE`	Source 3.
Plug (Output)	`OUT`	`adapter::types::unidirectional::AE`	Merged signal.

Functionality

Whenever `E1` arrives at any of the 3 sockets (`IN1`, `IN2`, `IN3`), it is forwarded unchanged to `OUT` as the matching event type. All event types are handled independently, and all 3 sockets are treated equally — the order in which sockets are checked follows `IN1` … `IN3`, which only matters if several events arrive within the same execution cycle. Since `AE` carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class `CGenUnidirectMergeBase` (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (`_3`).
- **No states / algorithms**: Since `AE` carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers AE_MERGE_2 through AE_MERGE_7; for other input counts see `AE_MERGE_2`, `AE_MERGE_4`, `AE_MERGE_5`, `AE_MERGE_6`, `AE_MERGE_7`.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 3 sockets is forwarded to `OUT` immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 3 redundant or alternative trigger paths) into a common AE adapter output.
- **Simplifying networks** that would otherwise need 3 separate event connections to the same destination.

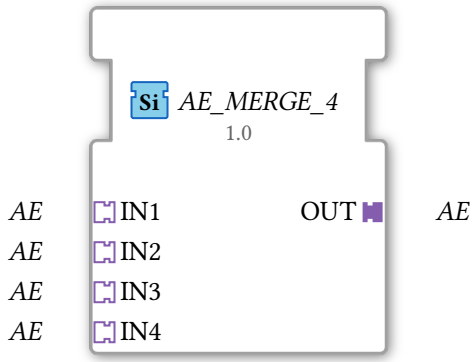
Comparison with Similar Components

- **AE_SPLIT_2**: the reverse direction for 2 outputs (for ASRT, AE_SPLIT_3 through AE_SPLIT_9 are also available).
- **`AE_MERGE_2`, `AE_MERGE_4`, `AE_MERGE_5`, `AE_MERGE_6`, `AE_MERGE_7`**: the same generic implementation with a different input count.
- ASR_MERGE_3, ASRT_MERGE_3: the same merge logic for the other unidirectional event adapters.

Conclusion

`AE_MERGE_3` provides a generically implemented merge of 3 `AE` event sources into a common adapter output, completing the `GEN_AE_MERGE` family with the 3-input variant.

AE_MERGE_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_MERGE_4** merges 4 unidirectional AE adapters (pure event adapter, no payload data) into one common output adapter. It is the 4-input variant of AE_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_AE_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 4 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::AE`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::AE`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::AE`	Source 3.
Socket (Input 4)	`IN4`	`adapter::types::unidirectional::AE`	Source 4.
Plug (Output)	`OUT`	`adapter::types::unidirectional::AE`	Merged signal.

Functionality

Whenever `E1` arrives at any of the 4 sockets (`IN1`, `IN2`, `IN3`, `IN4`), it is forwarded unchanged to `OUT` as the matching event type. All event types are handled independently, and all 4 sockets are treated equally — the order in which sockets are checked follows `IN1` … `IN4`, which only matters if several events arrive within the same execution cycle. Since `AE` carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class `CGenUnidirectMergeBase` (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (_4).
- **No states / algorithms**: Since `AE` carries no data value, there is no change detection and no ECC.

- **Arbitrary input count**: The same implementation covers AE_MERGE_2 through AE_MERGE_7; for other input counts see `AE_MERGE_2`, `AE_MERGE_3`, `AE_MERGE_5`, `AE_MERGE_6`, `AE_MERGE_7`.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 4 sockets is forwarded to `OUT` immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 4 redundant or alternative trigger paths) into a common AE adapter output.
- **Simplifying networks** that would otherwise need 4 separate event connections to the same destination.

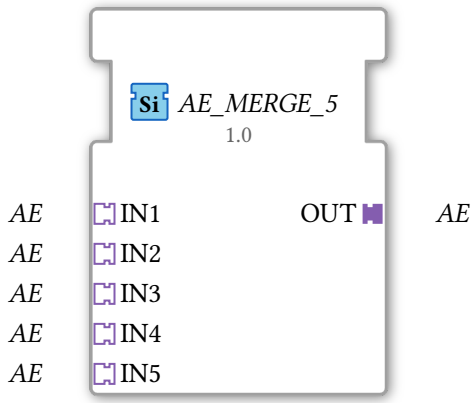
Comparison with Similar Components

- AE_SPLIT_2: the reverse direction for 2 outputs (for ASRT, AE_SPLIT_3 through AE_SPLIT_9 are also available).
- **`AE_MERGE_2`, `AE_MERGE_3`, `AE_MERGE_5`, `AE_MERGE_6`, `AE_MERGE_7`**: the same generic implementation with a different input count.
- ASR_MERGE_4, ASRT_MERGE_4: the same merge logic for the other unidirectional event adapters.

Conclusion

`AE_MERGE_4` provides a generically implemented merge of 4 `AE` event sources into a common adapter output, completing the `GEN_AE_MERGE` family with the 4-input variant.

AE_MERGE_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_MERGE_5** merges 5 unidirectional AE adapters (pure event adapter, no payload data) into one common output adapter. It is the 5-input variant of AE_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_AE_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 5 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	IN1	adapter::types::unidirectional::AE	Source 1.
Socket (Input 2)	IN2	adapter::types::unidirectional::AE	Source 2.
Socket (Input 3)	IN3	adapter::types::unidirectional::AE	Source 3.
Socket (Input 4)	IN4	adapter::types::unidirectional::AE	Source 4.
Socket (Input 5)	IN5	adapter::types::unidirectional::AE	Source 5.
Plug (Output)	OUT	adapter::types::unidirectional::AE	Merged signal.

Functionality

Whenever E1 arrives at any of the 5 sockets (IN1, IN2, IN3, IN4, IN5), it is forwarded unchanged to OUT as the matching event type. All event types are handled independently, and all 5 sockets are treated equally — the order in which sockets are checked follows IN1 … IN5, which only matters if several events arrive within the same execution cycle. Since AE carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectMergeBase (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (_5).
- **No states / algorithms**: Since AE carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers AE_MERGE_2 through AE_MERGE_7; for other input counts see AE_MERGE_2, AE_MERGE_3, AE_MERGE_4, AE_MERGE_6, AE_MERGE_7.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 5 sockets is forwarded to OUT immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 5 redundant or alternative trigger paths) into a common AE adapter output.
- **Simplifying networks** that would otherwise need 5 separate event connections to the same destination.

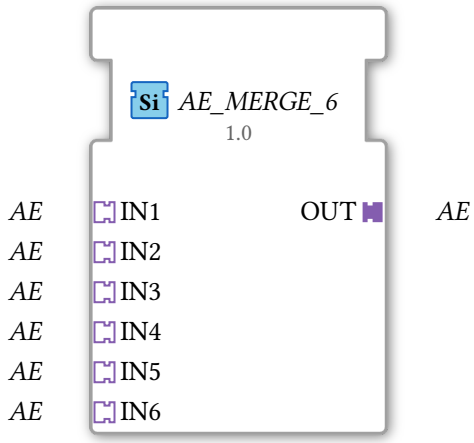
Comparison with Similar Components

- **AE_SPLIT_2**: the reverse direction for 2 outputs (for ASRT, AE_SPLIT_3 through AE_SPLIT_9 are also available).
- **AE_MERGE_2, AE_MERGE_3, AE_MERGE_4, AE_MERGE_6, AE_MERGE_7**: the same generic implementation with a different input count.
- ASR_MERGE_5, ASRT_MERGE_5: the same merge logic for the other unidirectional event adapters.

Conclusion

AE_MERGE_5 provides a generically implemented merge of 5 AE event sources into a common adapter output, completing the GEN_AE_MERGE family with the 5-input variant.

AE_MERGE_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_MERGE_6** merges 6 unidirectional AE adapters (pure event adapter, no payload data) into one common output adapter. It is the 6-input variant of AE_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_AE_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 6 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::AE`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::AE`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::AE`	Source 3.
Socket (Input 4)	`IN4`	`adapter::types::unidirectional::AE`	Source 4.
Socket (Input 5)	`IN5`	`adapter::types::unidirectional::AE`	Source 5.
Socket (Input 6)	`IN6`	`adapter::types::unidirectional::AE`	Source 6.
Plug (Output)	`OUT`	`adapter::types::unidirectional::AE`	Merged signal.

Functionality

Whenever `E1` arrives at any of the 6 sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`), it is forwarded unchanged to `OUT` as the matching event type. All event types are handled independently, and all 6 sockets are treated equally — the order in which sockets are checked follows `IN1` … `IN6`, which only matters if several events

arrive within the same execution cycle. Since AE carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectMergeBase (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (_6).
- **No states / algorithms**: Since AE carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers AE_MERGE_2 through AE_MERGE_7; for other input counts see AE_MERGE_2, AE_MERGE_3, AE_MERGE_4, AE_MERGE_5, AE_MERGE_7.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 6 sockets is forwarded to OUT immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 6 redundant or alternative trigger paths) into a common AE adapter output.
- **Simplifying networks** that would otherwise need 6 separate event connections to the same destination.

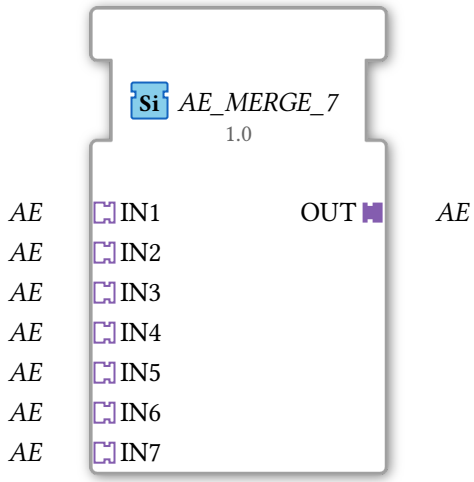
Comparison with Similar Components

- AE_SPLIT_2: the reverse direction for 2 outputs (for ASRT, AE_SPLIT_3 through AE_SPLIT_9 are also available).
- **AE_MERGE_2, AE_MERGE_3, AE_MERGE_4, AE_MERGE_5, AE_MERGE_7**: the same generic implementation with a different input count.
- ASR_MERGE_6, ASRT_MERGE_6: the same merge logic for the other unidirectional event adapters.

Conclusion

AE_MERGE_6 provides a generically implemented merge of 6 AE event sources into a common adapter output, completing the GEN_AE_MERGE family with the 6-input variant.

AE_MERGE_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_MERGE_7** merges 7 unidirectional AE adapters (pure event adapter, no payload data) into one common output adapter. It is the 7-input variant of AE_MERGE_2 and, like it, is implemented as a generic FB (GenericClassName: 'GEN_AE_MERGE') — the same C++ base (CGenUnidirectMergeBase), just with 7 instead of 2 sockets.

Interface Structure

Event Inputs
None. Events are received exclusively via the adapter sockets.

Event Outputs
None. Events are forwarded exclusively via the adapter plug.

Data Inputs
None.

Data Outputs
None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::AE`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::AE`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::AE`	Source 3.
Socket (Input 4)	`IN4`	`adapter::types::unidirectional::AE`	Source 4.
Socket (Input 5)	`IN5`	`adapter::types::unidirectional::AE`	Source 5.
Socket (Input 6)	`IN6`	`adapter::types::unidirectional::AE`	Source 6.
Socket (Input 7)	`IN7`	`adapter::types::unidirectional::AE`	Source 7.
Plug (Output)	`OUT`	`adapter::types::unidirectional::AE`	Merged signal.

Functionality
Whenever `E1` arrives at any of the 7 sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`), it is forwarded unchanged to `OUT` as the matching event type. All event types are handled independently, and all 7 sockets are treated equally — the order in which sockets are checked follows `IN1` … `IN7`, which only matters if several events arrive within the same execution cycle. Since `AE` carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features
- **Generic type**: Implemented via the generic base class `CGenUnidirectMergeBase` (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (`_7`).
- **No states / algorithms**: Since `AE` carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers AE_MERGE_2 through AE_MERGE_7; for other input counts see `AE_MERGE_2`, `AE_MERGE_3`, `AE_MERGE_4`, `AE_MERGE_5`, `AE_MERGE_6`.

State Overview
The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 7 sockets is forwarded to `OUT` immediately.

Application Scenarios
- **Merging several equivalent signal sources** (e.g. 7 redundant or alternative trigger paths) into a common AE adapter output.
- **Simplifying networks** that would otherwise need 7 separate event connections to the same destination.

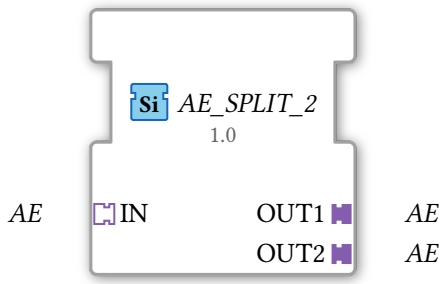
Comparison with Similar Components
- AE_SPLIT_2: the reverse direction for 2 outputs (for ASRT, AE_SPLIT_3 through AE_SPLIT_9 are also available).

- **AE_MERGE_2, AE_MERGE_3, AE_MERGE_4, AE_MERGE_5, AE_MERGE_6**: the same generic implementation with a different input count.
- ASR_MERGE_7, ASRT_MERGE_7: the same merge logic for the other unidirectional event adapters.

Conclusion

AE_MERGE_7 provides a generically implemented merge of 7 AE event sources into a common adapter output, completing the GEN_AE_MERGE family with the 7-input variant.

AE_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_SPLIT_2** serves as a generic distributor for an incoming adapter event (AE). It receives a unidirectional event via socket IN and forwards it without delay or modification to the two output plugs OUT1 and OUT2. The block is implemented as a generic FB (GenericClassName: 'GEN_AE_SPLIT'), so it can theoretically be used for various adapter types – provided they conform to the unidirectional AE protocol.

Interface Structure

Event Inputs

None. The FB does not have any traditional event inputs. Event forwarding occurs exclusively via the adapter socket.

Event Outputs

None. Event forwarding occurs via the adapter plugs.

Data Inputs

None.

Data Outputs

None.

Adapter

Role	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::AE	Receives a unidirectional event, which is distributed to both outputs.
Plug (Output 1)	OUT1	adapter::types::unidirectional::AE	First output for the duplicated event.
Plug (Output 2)	OUT2	adapter::types::unidirectional::AE	Second output for the duplicated event.

Functionality

As soon as an event arrives at the adapter socket IN, it is forwarded **immediately and in parallel** to both output plugs OUT1 and OUT2. The function block performs no logic, filtering, or delay – it functions purely

as a splitter at the adapter level. The distribution is event-based: Every single incoming event appears at both outputs.

Technical Features

- **Generic Type**: The function block is declared as a generic function block (`GenericClassName = 'GEN_AE_SPLIT'`). This allows adaptation to other user-defined adapter types, as long as they use the unidirectional event interface. In the current configuration, the specific type `adapter::types::unidirectional::AE` is defined.
- **No States / Algorithms**: Since no logic or memory behavior is implemented, the function block (FB) has no Execution Control Chart (ECC) and no internal states.
- **Latency**: Forwarding occurs instantaneously within one execution cycle.

State Overview

The FB has **no state machine**. Its behavior is purely combinatorial: Every incoming adapter event is immediately duplicated and output to both outputs.

Application Scenarios

- **Event Distribution**: A sensor or controller delivers an event (e.g., "Start" or "Alarm") that must be processed by two independent subsystems.
- **Parallel Circuit**: Splitting a signal to simultaneously control two actuators or for parallel data acquisition.
- **Prototyping**: Temporary branching of an event stream during the development and testing phase.

Comparison with Similar Components

- **AE_SPLIT_1**: A splitter with only one output (i.e., pure pass-through) – usually unnecessary.
- **E_SPLIT_2**: Classic event splitter for standard event inputs/outputs. Unlike `AE_SPLIT_2`, this one operates at the event level without adapter coupling.
- **AE_ROUTER**: An adapter-based distributor that can selectively route events to specific outputs (e.g., based on data values). `AE_SPLIT_2` is simpler and always distributes to all outputs.

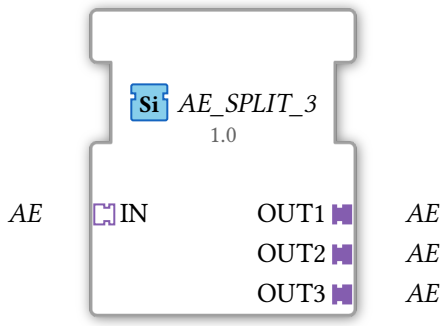
Conclusion

The **AE_SPLIT_2** is a minimal yet useful function block for easily distributing adapter events. Its generic nature makes it versatile, while the lack of logic ensures low latency and high reliability. It is ideal for all scenarios where an event needs to be forwarded to multiple receivers.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

AE_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_SPLIT_3** is used to distribute an incoming adapter data stream (of type `AE`) to three identical outputs. It is implemented as a generic function block and allows for the flexible use of different adapter types through design-time configuration.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapter

Direction	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::AE	Incoming adapter data stream, distributed to the three outputs.
Connector (Output)	OUT1	adapter::types::unidirectional::AE	First output – receives the same data as the input.
Connector (Output)	OUT2	adapter::types::unidirectional::AE	Second output – receives the same data as the input.
Connector (Output)	OUT3	adapter::types::unidirectional::AE	Third output – receives the same data as the input.

Functionality
The function block accepts a unidirectional adapter data stream of type `AE` via socket `IN`. Each incoming data packet or event is forwarded without modification to all three adapter plugs (`OUT1`, `OUT2`, `OUT3`). No processing or preparation of the data takes place – the function block acts purely as a **signal distributor** for adapter connections.

The function block accepts a unidirectional adapter data stream of type `AE` via socket `OUT1`, `OUT2`, and `OUT3`. Thanks to the generic design (`eclipse4diac::core::GenericClassName = 'GEN_AE_SPLIT'`), the specific adapter type (e.g., a user-defined AE subtype) can be defined at design time for `IN`, `OUT1`–`OUT3`. At runtime, all instances are identically typed.

Technical Features
- **Generic Component** – The adapter type is only defined during instantiation in the project, which increases reusability and type safety.
- **Unidirectional Communication** – Data flows only from the input to the outputs; reverse communication is not supported.
- **License** – The component is licensed under the **Eclipse Public License 2.0 (EPL-2.0)**, which permits free use, modification, and distribution.
- No event- or data-based inputs/outputs – all communication takes place exclusively via the adapter interfaces.

State Overview
The function block does not have an internal state diagram (ECC). It operates **combinatorically**: Any change at the input is immediately and without delay passed on to all outputs.

Application Scenarios
- **Signal Multiplication** – An adapter data stream provided by a sensor (e.g., `AE_Temperature`) is to be sent in parallel to three different evaluation units.
- **Distribution of Event Adapters** – In automation systems that rely on adapter-based communication, this allows multiple receivers to be served.
- **Test and Simulation Environments** – A data stream is distributed across multiple monitoring or logging modules.

Comparison with Similar Function Blocks

Function Block	Number of Outputs	Special Features
`AE_SPLIT_2`	2	Distributed to two outputs.
`AE_SPLIT_3`	3	Distributed to three outputs (existing FB).
`AE_SPLIT_N` (hypothetical)	variable	Flexible number via parameters – requires more configuration effort.

All variants have in common that they operate solely at the adapter level and do not manipulate any data.

Conclusion

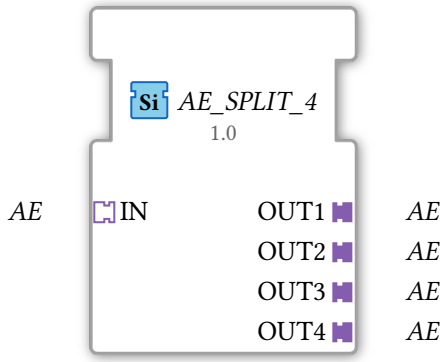
The **AE_SPLIT_3** is a simple yet indispensable function block for duplicating adapter connections in the 4diac IDE. Its generic design and clear, unidirectional structure make it a maintainable tool for signal distribution without additional logic. Thanks to the EPL 2.0 license, it can be used in your own projects without restrictions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AE_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_SPLIT_4** distributes an incoming event from a unidirectional **AE adapter** to four identical output adapters. It is executed as a generic function block (Generic FB) and can therefore be used for any data type of the adapter. The distribution occurs without delay or state change – the incoming event is forwarded to all four outputs simultaneously.

Interface Structure

Event Inputs

The function block has no explicit event inputs. Event control is handled exclusively via the **IN** socket (see section Adapter).

Event Outputs

The function block has no explicit event outputs. Event propagation is handled via the **OUT1** to **OUT4** plugs (see section Adapter).

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	**IN**	`adapter::types::unidirectional::AE`	Receives a unidirectional adapter event (event + associated data).
Plug (Output)	**OUT1**	`adapter::types::unidirectional::AE`	First output – receives the same event and data as the input.
Plug (Output)	**OUT2**	`adapter::types::unidirectional::AE`	Second output.
Plug (Output)	**OUT3**	`adapter::types::unidirectional::AE`	Third output.
Plug (Output)	**OUT4**	`adapter::types::unidirectional::AE`	Fourth output.

Functionality

As soon as an event from the AE adapter is received at socket **IN** (with all data associated with that event), this event is immediately and simultaneously passed on to all four plugs **OUT1**, **OUT2**, **OUT3**, and **OUT4**. The output is simultaneous – there are no priorities or sequences. Due to the generic design of the function block, the data types of the adapters are freely selectable (e.g., `INT`, `REAL`, `BOOL`, etc.), as long as all participating adapters use the same data type.

Technical Features

- **Generic Function Block** – The function block is implemented as a generic type (`GenericClassName = 'GEN_AE_SPLIT'`). This allows the specific data type of the adapter to be defined during the project planning phase.
- **Unidirectional Adapters** – The function block works exclusively with adapters of type `unidirectional::AE`. Return channels or bidirectional communication are not supported.
- **License** – The source code is subject to the **Eclipse Public License 2.0** (EPL-2.0).

State Overview

The **AE_SPLIT_4** is a purely combinational function block and has no internal states. The output signal depends solely on the current input signal – there is no storage or delay.

Application Scenarios

- **Sensor Distribution** – A single sensor (e.g., temperature, speed) provides data via an AE adapter; this data is to be distributed to several control functions operating in parallel.
- **Signal Propagation in a Control Chain** – An event must reach several downstream function blocks in parallel, e.g., for triggering alarms and logging.
- **Event Source Multiplexing** – (in conjunction with a preceding AE-MUX) different sources can be routed to the same four outputs.

Comparison with Similar Function Blocks

- **E_SPLIT** – Standard 4diac function block for distributing an event to multiple event outputs without data transmission. **AE_SPLIT_4** extends this functionality by also enabling the transmission of adapter data associated with the event.
- **F_SPLIT** – Data splitters (e.g., for IEC 61131-3 “MUX”) typically do not operate in an event-driven manner. **AE_SPLIT_4** is specifically optimized for event-based adapter communication.
- **AE_MERGE** – The reverse function block: combines multiple AE inputs into a single output.

Conclusion

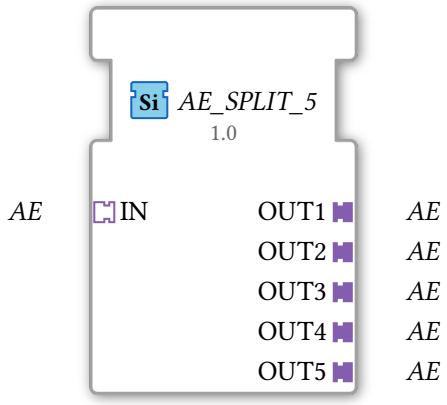
The **AE_SPLIT_4** is a simple yet powerful function block that enables the event-driven distribution of adapter data to up to four equal outputs. Thanks to its generic design, it can be used for any data type without modification and is ideally suited for modular, parallel control architectures.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AE_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_SPLIT_5** is used to duplicate an incoming adapter event. It receives an adapter event (AE) at its socket `IN` and forwards it identically to five separate output plugs (`OUT1` … `OUT5`). The block is implemented as a generic FB (GenericClassName `'GEN_AE_SPLIT'`), thus enabling easy reuse in different contexts.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AE`	A unidirectional adapter event whose content is forwarded to all outputs.
Plug (Output 1)	`OUT1`	`adapter::types::unidirectional::AE`	First output that provides the incoming event.
Plug (Output 2)	`OUT2`	`adapter::types::unidirectional::AE`	Second output providing the incoming event.
Plug (Output 3)	`OUT3`	`adapter::types::unidirectional::AE`	Third output providing the incoming event.
Plug (Output 4)	`OUT4`	`adapter::types::unidirectional::AE`	Fourth output providing the incoming event.
Plug (Output 5)	`OUT5`	`adapter::types::unidirectional::AE`	Fifth output providing the incoming event.

Functionality

The FB **AE_SPLIT_5** is a pure distribution block. As soon as an adapter event arrives at socket `IN`, this event is sent – without delay or data manipulation – in parallel to all five output plugs `OUT1` to `OUT5`. No internal storage or state modification occurs; the function block (FB) is entirely combinatorial.

Technical Features

- **Generic Implementation**: The function block is declared as a generic FB (`GenericClassName = 'GEN_AE_SPLIT'`), so it can be used with different adapter types if needed, provided they are of type `adapter::types::unidirectional::AE`.
- **Unidirectionality**: All adapters involved are unidirectional. There is no communication back to the sender.
- **No Additional Logic**: The FB has neither event nor data inputs/outputs. The forwarding is done entirely via adapters.

State Overview

This function block has no defined state machine or sequence control. Its behavior is static: Upon receiving an event at `IN`, all outputs are activated simultaneously. There is no latency or intermediate states.

Application Scenarios

- **Event Distribution in Control Systems**: A sensor event (e.g., "Position Reached") is to be sent to multiple independent control components.
- **Multicast in Modular Architectures**: In a distributed automation environment, a central function block can distribute its event to up to five downstream modules.
- **Prototyping and Test Environments**: For simultaneously testing multiple receivers with the same event.

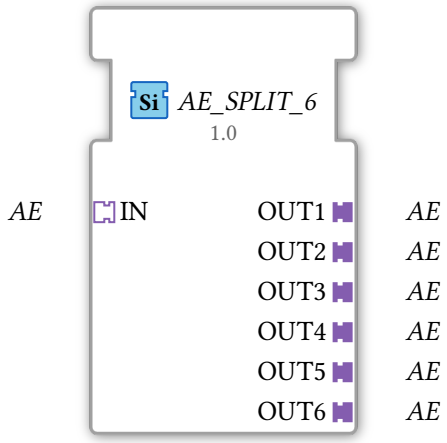
Comparison with Similar Function Blocks

- **Event Splitter (E_SPLIT)**: These function blocks distribute **events** (event inputs) to multiple event outputs. **AE_SPLIT_5**, on the other hand, distributes **adapter events** (complex data objects that communicate via adapters), thus offering a higher level of abstraction.
- **Data Splitter (F_SPLIT)**: Splits data streams but does not have an adapter interface. **AE_SPLIT_5** focuses on distributing adapter-based event messages.

Conclusion

The **AE_SPLIT_5** is a simple yet effective component for replicating a unidirectional adapter event to five outputs. Due to its generic nature and its pure adapter interface, it is particularly suitable for modular and scalable control applications where information needs to be sent to multiple receivers in parallel.

AE_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_SPLIT_6** distributes an incoming AE adapter (type `adapter::types::unidirectional::AE`) to six identical outputs. It is a generic function block that can be reused for various AE types. The name derives from the 1:6 split: one input is split into six outputs.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Name	Type	Description
Socket (Input)	IN	`adapter::types::unidirectional::AE`	Incoming AE adapter, whose signal is duplicated to all outputs.
Plug (Output)	OUT1	`adapter::types::unidirectional::AE`	First output, identical copy of the input signal.
Plug (Output)	OUT2	`adapter::types::unidirectional::AE`	Second output, identical copy of the input signal.
Plug (Output)	OUT3	`adapter::types::unidirectional::AE`	Third output, identical copy of the input signal.
Plug (Output)	OUT4	`adapter::types::unidirectional::AE`	Fourth output, identical copy of the input signal.
Plug (Output)	OUT5	`adapter::types::unidirectional::AE`	Fifth output, identical copy of the input signal.
Plug (Output)	OUT6	`adapter::types::unidirectional::AE`	Sixth output, identical copy of the input signal.

Functionality

The function block forwards the adapter signal present at socket **IN** unchanged to all six plugs (**OUT1** … **OUT6**). No logical processing, filtering, or delay takes place – the distribution is passive and instantaneous. Thus, the function block behaves like a pure 1:6 splitter for the unidirectional adapter type `AE`.

Technical Features

- **Generic Block**: The function block (FB) is marked with the attribute `eclipse4diac::core::GenericClassName` as `'GEN_AE_SPLIT'`. This allows it to be used in different projects with different, but structurally identical, AE adapter types – the specific type is only determined at project runtime.
- **Unidirectionality**: All adapters are declared as unidirectional, meaning that data/event transfer only occurs from the input (socket) to the outputs (plugs). Feedback is not provided.
- **No State Logic**: The function block has no internal states, no events, and no data inputs/outputs. It is entirely defined by its adapter interfaces.

State Overview

The FB **AE_SPLIT_6** is a purely combinational function block without a state machine. There are no internal steps, transitions, or actions – distribution is direct and permanent.

Application Scenarios

- **Signal Distribution**: Several downstream function blocks should receive the same AE signal, e.g., to start parallel calculations or parallel output.
- **Bus Structure**: In a control architecture where an event must be sent to multiple components (actuators, displays, loggers) without burdening the source.
- **Adapter Networks**: When an AE adapter is provided by a sender and multiple receivers need to access it independently.

Comparison with Similar Function Blocks

- **E_SPLIT Function Blocks**: These split an event signal (e.g., into **E_SPLIT** or **E_F_SPLIT**) across multiple event outputs. The **AE_SPLIT_6**, on the other hand, operates at the adapter level and distributes the entire adapter network, which can contain both events and data – depending on the specific adapter instantiation.
- **AE_SPLIT_2 / AE_SPLIT_4**: Variants with two or four outputs. The **AE_SPLIT_6** offers the maximum number of outputs in this family: six.
- **MUX/DEMUX blocks**: These select or combine signals. The **AE_SPLIT_6**, however, duplicates the signal without selection.

Conclusion

The **AE_SPLIT_6** is a simple yet useful generic function block for the 1:6 distribution of a unidirectional AE adapter. Its generic design allows it to be used in a wide variety of automation and control environments where a signal is required multiple times. It operates without delay and without its own logic, making it particularly efficient and reliable.

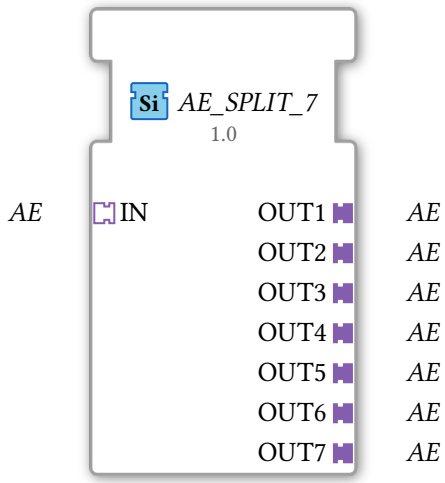
AE_SPLIT_6 –

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AE_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_SPLIT_7** is used to distribute a single incoming adapter event (AE) to seven identical output adapters. It is a generic function block designed for unidirectional adapters of type `adapter::types::unidirectional::AE`. The function block enables star-shaped routing of an event signal to multiple downstream function blocks without modifying or delaying the events.

Interface Structure

Event Inputs
None – communication occurs exclusively via adapters.

Event Outputs
None – output occurs via adapters.

Data Inputs
None.

Data Outputs
None.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AE`	**IN**	Socket	Input adapter that receives the event to be distributed.
`adapter::types::unidirectional::AE`	**OUT1**	Plug	First output adapter; the received event is forwarded here.
`adapter::types::unidirectional::AE`	**OUT2**	Plug	Second output adapter.
`adapter::types::unidirectional::AE`	**OUT3**	Plug	Third output adapter.
`adapter::types::unidirectional::AE`	**OUT4**	Plug	Fourth output adapter.
`adapter::types::unidirectional::AE`	**OUT5**	Plug	Fifth output adapter.
`adapter::types::unidirectional::AE`	**OUT6**	Plug	Sixth output adapter.
`adapter::types::unidirectional::AE`	**OUT7**	Plug	Seventh output adapter.

Functionality
This function block is passive and forwards an incoming event at the input adapter **IN** unchanged to all seven output adapters **OUT1** to **OUT7**. The forwarding occurs simultaneously (logically parallel), meaning that after an event is received, all outputs are supplied with the same event. No buffering, prioritization, or data manipulation takes place. Since these are unidirectional adapters, data flow is only possible in one direction (from the socket to the plug).

Technical Features
- **Generic Function Block:** The function block is declared as a generic type (`GEN_AE_SPLIT`) and can be used in various contexts with the corresponding adapter type. Type checking is performed at design time.
- **No Latency:** The event is passed through without delay; the runtime depends solely on the infrastructure of the IEC 61499 runtime system.
- **Type Identity:** All participating adapters must be of the same type (`AE` – unidirectional). Mixing different adapter types is not permitted.
- **No State Storage:** The function block (FB) has no internal memory – it only performs a one-to-many (1:n) forwarding.

State Overview
Since the FB does not have its own state machine and operates solely reactively to incoming events, there is only one implicit state: **Ready**. In this state, input events are immediately passed to all outputs. There are no error states or start/stop logic.

Application Scenarios
- **Event Distribution in Control Applications:** A single sensor or bus participant triggers an event that should simultaneously reach multiple actuators, alarms, or monitoring blocks.

- **Parallel Processing:** An event should be processed simultaneously in several independent function blocks, e.g., for redundancy or different evaluations.
- **Application Structuring:** Replaces multiple manual connection nodes and improves clarity in the network diagram.

Comparison with Similar Function Blocks

- **AE_SPLIT_2, AE_SPLIT_3, ... :** These function blocks offer the same functionality but with fewer outputs. The choice depends on the required number.
- **E_SPLIT (Event Splitter, Dataless):** A pure event splitter (without adapter encapsulation) distributes events but cannot handle adapter interfaces. The AE_SPLIT_7 is specifically designed for use with adapter types such as `AE`.
- **Adapter Merger (e.g., AE_MERGE):** Combines multiple adapter events into one – the opposite of this function block.

Conclusion

The **AE_SPLIT_7** is a simple yet useful function block for duplicating adapter events in IEC 61499 applications. Its generic nature, full event fidelity, and ease of use make it a fundamental tool for structured event distribution. It is ideally suited when an event needs to be forwarded to multiple similar target adapters and no data content is being transferred.

AE_SPLIT_8

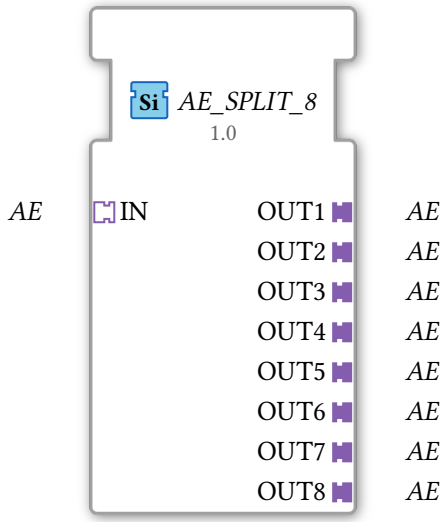


Image of the function block: [Insert here if available]

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AE_SPLIT_8** serves as a generic distributor for one unidirectional adapter input (AE) to eight identical adapter outputs. It allows the multiple use of an incoming signal without modifying it.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AE`	Incoming unidirectional AE adapter
Plug	`OUT1`	`adapter::types::unidirectional::AE`	First outgoing AE adapter
Plug	`OUT2`	`adapter::types::unidirectional::AE`	Second outgoing AE adapter
Plug	`OUT3`	`adapter::types::unidirectional::AE`	Third Outgoing AE Adapter
Plug	`OUT4`	`adapter::types::unidirectional::AE`	Fourth Outgoing AE Adapter
Plug	`OUT5`	`adapter::types::unidirectional::AE`	Fifth Outgoing AE Adapter
Plug	`OUT6`	`adapter::types::unidirectional::AE`	Sixth Outgoing AE Adapter
Plug	`OUT7`	`adapter::types::unidirectional::AE`	Seventh Outgoing AE Adapter
Plug	`OUT8`	`adapter::types::unidirectional::AE`	Eighth Outgoing AE Adapter

Functionality

This function block forwards the adapter signal present at socket `IN` to all eight plugs (`OUT1` to `OUT8`) without delay or modification. No logical or timing processing takes place – the incoming adapter is duplicated one-to-one.

Technical Features

- **Generic Type**: The function block is defined as a generic function block (`GenericClassName: GEN_AE_SPLIT`) and can be reused in various contexts.
- **Unidirectionality**: Both the input and output adapters are unidirectional. Feedback from the outputs to the input is not possible.
- **No Event Control**: The function block has no event inputs and operates purely passively. Forwarding occurs immediately upon connection establishment.
- **Scalability**: With a fixed number of eight outputs, the function block is optimized for applications requiring a precise 1:8 distribution.

State Overview

The function block has no internal states or sequence control. There is no ECC (Execution Control Chart) – functionality is purely combinatorial.

Application Scenarios

- **Signal Distribution in Control Networks**: An AE adapter signal from a sensor or a higher-level controller is to be simultaneously passed on to several downstream function blocks.
- **Test and Simulation Environments**: Multiple receivers are to receive the same adapter signal to observe parallel responses.
- **Redundant Connection**: The same signal can be processed in parallel on multiple paths, for example, for safety or monitoring functions.

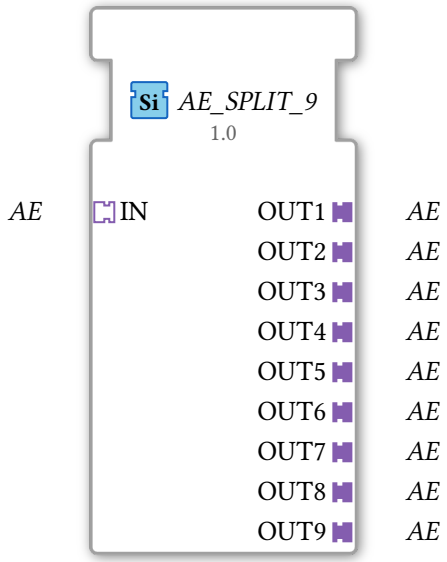
Comparison with Similar Function Blocks

- **AE_SPLIT_2/AE_SPLIT_4**: Function blocks with a smaller number of outputs (2 or 4, respectively) for smaller distributions. **AE_SPLIT_8** extends this to eight outputs.
- **Data Splitters (e.g., SPLIT_INT, SPLIT_BOOL)**: These distribute data values, while **AE_SPLIT_8** distributes adapter signals. Adapters can bundle complex composite interfaces (events and data).
- **Event Splitters (e.g., E_SPLIT)**: Distribute only events. **AE_SPLIT_8** transmits the entire adapter context, including all data and event connections it contains.

Conclusion

The **AE_SPLIT_8** is a simple and efficient function block for 1:8 distribution of unidirectional adapter signals. Due to its generic nature and the absence of event control, it is particularly suitable for applications where a signal needs to be forwarded to multiple receivers in parallel without delay.

AE_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The AE_SPLIT_9 function block is used to split an incoming unidirectional adapter event (AE) into nine identical outputs. It is implemented as a generic function block and allows events to be distributed to multiple downstream function blocks.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

- **IN** (Socket): Unidirectional AE adapter (input)
- **OUT1 ... OUT9** (Plugs): Unidirectional AE adapters (outputs)

Functionality

The function block receives an adapter event via the “IN” socket. Each incoming event is duplicated to all nine plug outputs (OUT1 to OUT9) without delay or further logic. Thus, the function block behaves like a 1:9 splitter for AE signals.

Technical Features

- The function block is implemented as a generic function block (GenericClassName = ‘GEN_AE_SPLIT’), which allows for flexible reuse in different contexts.
- No data is processed; only the adapter interface is passed through.
- The outputs are independent of each other; a fault on one output does not affect the others.

State Overview

The function block has no internal state. It operates purely combinatorially: An event at the input immediately triggers events at all outputs.

Application Scenarios

- Distribution of a sensor event to multiple control logics.
- Broadcast of an alarm to various monitoring units.
- Splitting a trigger signal for parallel processing paths.

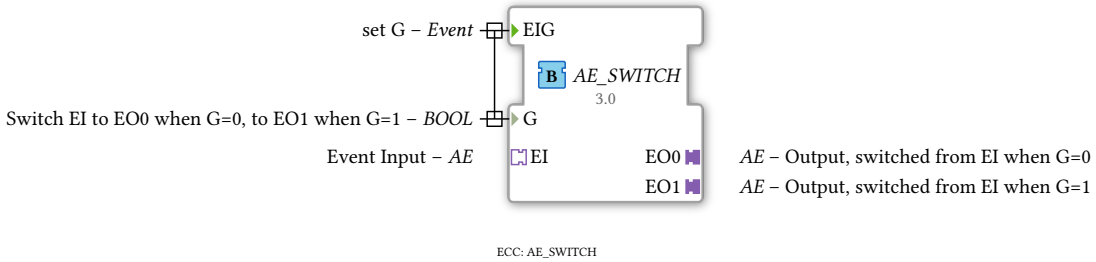
Comparison with Similar Function Blocks

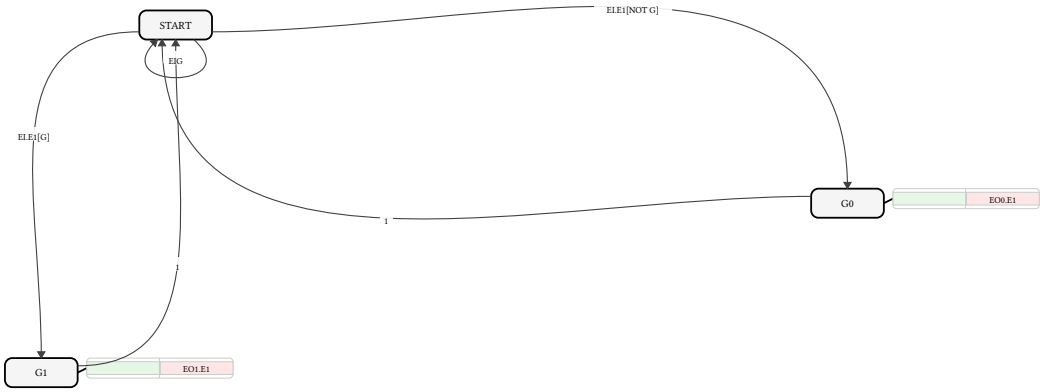
- AE_SPLIT_2, AE_SPLIT_4: same functionality with fewer outputs.
- Unlike data splitters (e.g., SPLIT_INT), this function block only replicates events (without data).

Conclusion

AE_SPLIT_9 is a simple yet useful function block for event distribution in IEC 61499 systems. Its generic nature and clear 1:9 split make it ideal for applications requiring a wide distribution of adapter events.

AE_SWITCH





• – ○ • – ○ • – ○ •

Introduction

The **AE_SWITCH** function block acts as an event demultiplexer. It forwards an incoming event to one of two unidirectional event adapter outputs, depending on the value of a Boolean input `G`.

Interface Structure

Event Inputs

- **EIG** (Event): Sets the value of `G`. When this event is triggered, the corresponding data input `G` is used – no forwarding to the outputs occurs.

Event Outputs

- No independent event outputs are available. Output is exclusively via the adapters `EO0` and `EO1`.

Data Inputs

Name	Type	Comment
`G`	BOOL	Switch: 0 ⟶ Output via `EO0`, 1 ⟶ Output via `EO1`

Data Outputs

- No data outputs.

Adapters

Direction	Name	Type (unidirectional)	Comment
Socket (Input)	`EI`	`adapter::types::unidirectional::AE`	Event input that is switched.
Plug (Output)	`EO0`	`adapter::types::unidirectional::AE`	Event output, active at `G = 0`
Plug (Output)	`EO1`	`adapter::types::unidirectional::AE`	Event output, active at `G = 1`

Functionality

The function block is initially in the **START** state.

1. **Setting the Switch**: An event at `EIG` inherits the current value from `G`. The block then remains in the **START** state.
2. **Forwarding the Input Event**: If an event arrives via the adapter input `EI`, the value of `G` is evaluated:

- If `G = 0` ⟶ state change to **G0**: The adapter output `EO0` is triggered with its event `E1`.
- If `G = 1` ⟶ State change to **G1**: The adapter output `EO1` is triggered by its event `E1`.

1. After the respective output is triggered, the function block automatically returns to the **START** state.

Technical Features

- Event forwarding occurs exclusively via **unidirectional adapters** of type `AE`. This ensures a clean separation of event sources and sinks.
- The function block implements **demultiplexing** (1-to-2) at the event level: An incoming event is routed to one of two outputs depending on a Boolean condition.
- The set input `EIG` allows the switching value to be changed independently of any pending event at input `EI`.

State Overview

State	Description	Action on Entry
START	Wait for an event at `EI` or `EIG`	–
G0	Event at `EI` with `G = 0` – forward to `EO0`	`EO0.E1`
G1	Event at `EI` with `G = 1` – forward to `EO1`	`EO1.E1`

Transitions:

- START ⟶ G0 : `EI.E1[NOT G]`
- START ⟶ G1 : `EI.E1[G]`
- G0 ⟶ START : `1` (immediate)
- G1 ⟶ START : `1` (immediate)
- START ⟶ START : `EIG` (only sets G, no forwarding)

Application Scenarios

- **Event Routing** in automation systems, where a central trigger (e.g., a sensor event) controls different actuators depending on the operating mode.
- **State-Dependent Processing**: An incoming event should only be processed in a specific operating mode; the other operating mode receives a separate event.
- **Test and Diagnostic Tasks**: Switching between normal operation and test mode by setting `G`.

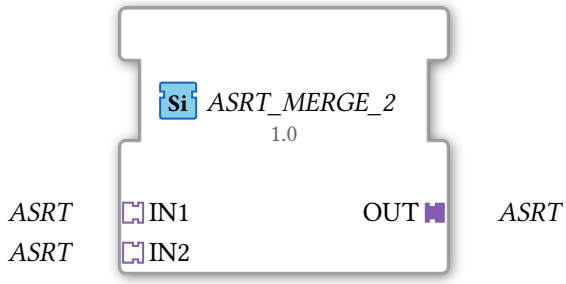
Comparison with Similar Function Blocks

- **E_SWITCH** (Standard 61499): Similar functionality, but via regular event outputs. `AE_SWITCH` uses adapters, which allows for better encapsulation and reusability in components with standardized interfaces.
- **E_MUX** (Multiplexer): Combines multiple events into one output – in the opposite direction. `AE_SWITCH` demultiplexes an input event to two outputs.

Conclusion

The `AE_SWITCH` offers a lean, adapter-based solution for conditional event forwarding. Its use of unidirectional adapters makes it ideal for modular and type-safe IEC 61499 applications. Its simple state machine makes it efficient and easy to understand.

ASRT_MERGE_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_MERGE_2** merges two unidirectional ASRT adapters (a set/reset/toggle event triple, no payload data) into one common output adapter. `ASRT` extends `ASR` with a third event `TOGGLE` (see `AX_T_FF_SR`). The block is the reverse of ASRT_SPLIT_2 and is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_MERGE'`).

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::ASRT`	First SET/RESET/TOGGLE source.

Socket (Input 2)	IN2	adapter::types::unidirectional::ASRT	Second SET/RESET/TOGGLE source.
Plug (Output)	OUT	adapter::types::unidirectional::ASRT	Merged SET/RESET/TOGGLE signal.

Functionality

Whenever a SET, RESET, or TOGGLE event arrives at IN1 **or** IN2, it is forwarded to OUT as the matching event type. All three event types are handled independently – which event arrives on which socket has no effect on the other two event types or on the other socket. Since ASRT carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectMergeBase (N sockets, 1 plug) – the same C++ base underlying GEN_AE_MERGE and GEN_ASR_MERGE.
- **Three independent event channels**: SET, RESET, and TOGGLE are checked and forwarded separately.
- **No states / algorithms**: Since ASRT carries no data value, there is no change detection and no ECC.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming SET, RESET, or TOGGLE event at IN1 or IN2 is forwarded to OUT immediately as the matching event type.

Application Scenarios

- **Merging redundant set/reset/toggle sources**: e.g. two operator stations that should both drive the same AX_T_FF_SR-controlled element via a common ASRT adapter.
- **Simplifying networks** that would otherwise need three separate event connection pairs (SET, RESET, TOGGLE) per source to the same destination.

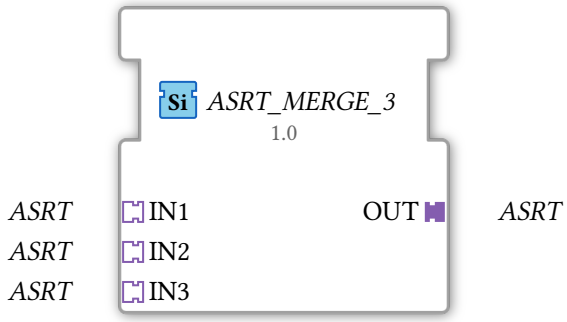
Comparison with Similar Components

- ASRT_SPLIT_2: the reverse direction – distributes one incoming set/reset/toggle signal to two outputs, instead of merging two inputs.
- ASR_MERGE_2: the same merge logic for the set/reset event adapter ASR (no TOGGLE).
- AE_MERGE_2: the same merge logic for the pure event adapter AE (1 event instead of SET/RESET/TOGGLE).

Conclusion

ASRT_MERGE_2 is a minimal generic function block for merging two set/reset/toggle event sources into a common ASRT adapter output, completing the GEN_*_MERGE family for all three unidirectional event adapters (AE, ASR, ASRT).

ASRT_MERGE_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_MERGE_3** merges 3 unidirectional ASRT adapters (set/reset/toggle event adapter, no payload data) into one common output adapter. It is the 3-input variant of ASRT_MERGE_2

and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 3 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::ASRT`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::ASRT`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::ASRT`	Source 3.
Plug (Output)	`OUT`	`adapter::types::unidirectional::ASRT`	Merged signal.

Functionality

Whenever `SET` or `RESET` or `TOGGLE` arrives at any of the 3 sockets (`IN1`, `IN2`, `IN3`), it is forwarded unchanged to `OUT` as the matching event type. All event types are handled independently, and all 3 sockets are treated equally — the order in which sockets are checked follows `IN1` … `IN3`, which only matters if several events arrive within the same execution cycle. Since `ASRT` carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class `CGenUnidirectMergeBase` (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (`_3`).
- **No states / algorithms**: Since `ASRT` carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers ASRT_MERGE_2 through ASRT_MERGE_7; for other input counts see `ASRT_MERGE_2`, `ASRT_MERGE_4`, `ASRT_MERGE_5`, `ASRT_MERGE_6`, `ASRT_MERGE_7`.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 3 sockets is forwarded to `OUT` immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 3 redundant or alternative trigger paths) into a common ASRT adapter output.
- **Simplifying networks** that would otherwise need 3 separate event connections to the same destination.

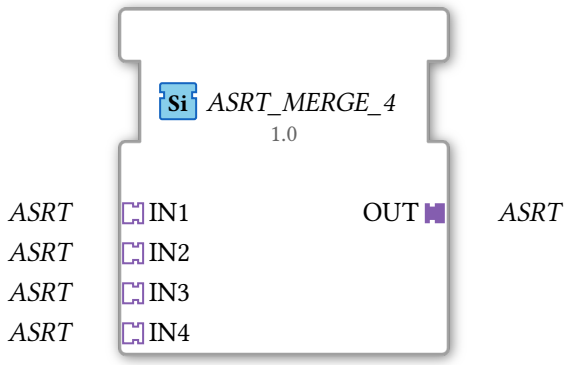
Comparison with Similar Components

- ASRT_SPLIT_2: the reverse direction for 2 outputs (for ASRT, ASRT_SPLIT_3 through ASRT_SPLIT_9 are also available).
- **`ASRT_MERGE_2`, `ASRT_MERGE_4`, `ASRT_MERGE_5`, `ASRT_MERGE_6`, `ASRT_MERGE_7`**: the same generic implementation with a different input count.
- AE_MERGE_3, ASR_MERGE_3: the same merge logic for the other unidirectional event adapters.

Conclusion

`ASRT_MERGE_3` provides a generically implemented merge of 3 `ASRT` event sources into a common adapter output, completing the `GEN_ASRT_MERGE` family with the 3-input variant.

ASRT_MERGE_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_MERGE_4** merges 4 unidirectional ASRT adapters (set/reset/toggle event adapter, no payload data) into one common output adapter. It is the 4-input variant of ASRT_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 4 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::ASRT`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::ASRT`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::ASRT`	Source 3.
Socket (Input 4)	`IN4`	`adapter::types::unidirectional::ASRT`	Source 4.
Plug (Output)	`OUT`	`adapter::types::unidirectional::ASRT`	Merged signal.

Functionality

Whenever `SET` or `RESET` or `TOGGLE` arrives at any of the 4 sockets (`IN1`, `IN2`, `IN3`, `IN4`), it is forwarded unchanged to `OUT` as the matching event type. All event types are handled independently, and all 4 sockets are treated equally — the order in which sockets are checked follows `IN1` … `IN4`, which only matters if several events arrive within the same execution cycle. Since `ASRT` carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class `CGenUnidirectMergeBase` (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (_4).
- **No states / algorithms**: Since `ASRT` carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers ASRT_MERGE_2 through ASRT_MERGE_7; for other input counts see `ASRT_MERGE_2`, `ASRT_MERGE_3`, `ASRT_MERGE_5`, `ASRT_MERGE_6`, `ASRT_MERGE_7`.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 4 sockets is forwarded to `OUT` immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 4 redundant or alternative trigger paths) into a common ASRT adapter output.
- **Simplifying networks** that would otherwise need 4 separate event connections to the same destination.

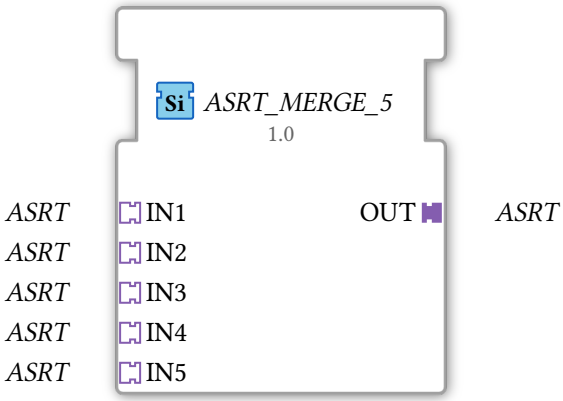
Comparison with Similar Components

- **ASRT_SPLIT_2**: the reverse direction for 2 outputs (for ASRT, ASRT_SPLIT_3 through ASRT_SPLIT_9 are also available).
- **`ASRT_MERGE_2`, `ASRT_MERGE_3`, `ASRT_MERGE_5`, `ASRT_MERGE_6`, `ASRT_MERGE_7`**: the same generic implementation with a different input count.
- AE_MERGE_4, ASR_MERGE_4: the same merge logic for the other unidirectional event adapters.

Conclusion

`ASRT_MERGE_4` provides a generically implemented merge of 4 `ASRT` event sources into a common adapter output, completing the `GEN_ASRT_MERGE` family with the 4-input variant.

ASRT_MERGE_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_MERGE_5** merges 5 unidirectional ASRT adapters (set/reset/toggle event adapter, no payload data) into one common output adapter. It is the 5-input variant of ASRT_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 5 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	IN1	adapter::types::unidirectional::ASRT	Source 1.
Socket (Input 2)	IN2	adapter::types::unidirectional::ASRT	Source 2.
Socket (Input 3)	IN3	adapter::types::unidirectional::ASRT	Source 3.
Socket (Input 4)	IN4	adapter::types::unidirectional::ASRT	Source 4.
Socket (Input 5)	IN5	adapter::types::unidirectional::ASRT	Source 5.
Plug (Output)	OUT	adapter::types::unidirectional::ASRT	Merged signal.

Functionality

Whenever SET or RESET or TOGGLE arrives at any of the 5 sockets (IN1, IN2, IN3, IN4, IN5), it is forwarded unchanged to OUT as the matching event type. All event types are handled independently, and all 5 sockets are treated equally — the order in which sockets are checked follows IN1 … IN5, which only matters if several events arrive within the same execution cycle. Since ASRT carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectMergeBase (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (_5).
- **No states / algorithms**: Since ASRT carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers ASRT_MERGE_2 through ASRT_MERGE_7; for other input counts see ASRT_MERGE_2, ASRT_MERGE_3, ASRT_MERGE_4, ASRT_MERGE_6, ASRT_MERGE_7.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 5 sockets is forwarded to OUT immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 5 redundant or alternative trigger paths) into a common ASRT adapter output.
- **Simplifying networks** that would otherwise need 5 separate event connections to the same destination.

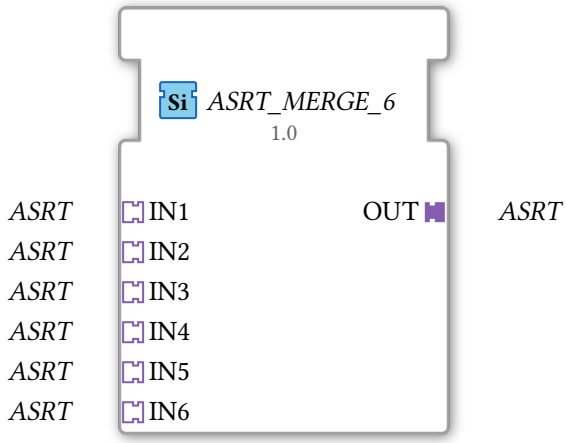
Comparison with Similar Components

- **ASRT_SPLIT_2**: the reverse direction for 2 outputs (for ASRT, ASRT_SPLIT_3 through ASRT_SPLIT_9 are also available).
- **ASRT_MERGE_2, ASRT_MERGE_3, ASRT_MERGE_4, ASRT_MERGE_6, ASRT_MERGE_7**: the same generic implementation with a different input count.
- AE_MERGE_5, ASR_MERGE_5: the same merge logic for the other unidirectional event adapters.

Conclusion

ASRT_MERGE_5 provides a generically implemented merge of 5 ASRT event sources into a common adapter output, completing the GEN_ASRT_MERGE family with the 5-input variant.

ASRT_MERGE_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_MERGE_6** merges 6 unidirectional ASRT adapters (set/reset/toggle event adapter, no payload data) into one common output adapter. It is the 6-input variant of ASRT_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 6 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::ASRT`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::ASRT`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::ASRT`	Source 3.
Socket (Input 4)	`IN4`	`adapter::types::unidirectional::ASRT`	Source 4.
Socket (Input 5)	`IN5`	`adapter::types::unidirectional::ASRT`	Source 5.
Socket (Input 6)	`IN6`	`adapter::types::unidirectional::ASRT`	Source 6.
Plug (Output)	`OUT`	`adapter::types::unidirectional::ASRT`	Merged signal.

Functionality

Whenever `SET` or `RESET` or `TOGGLE` arrives at any of the 6 sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`), it is forwarded unchanged to `OUT` as the matching event type. All event types are handled independently, and all 6 sockets are treated equally — the order in which sockets are checked follows `IN1` … `IN6`, which only

matters if several events arrive within the same execution cycle. Since ASRT carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectMergeBase (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (_6).
- **No states / algorithms**: Since ASRT carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers ASRT_MERGE_2 through ASRT_MERGE_7; for other input counts see ASRT_MERGE_2, ASRT_MERGE_3, ASRT_MERGE_4, ASRT_MERGE_5, ASRT_MERGE_7.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 6 sockets is forwarded to OUT immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 6 redundant or alternative trigger paths) into a common ASRT adapter output.
- **Simplifying networks** that would otherwise need 6 separate event connections to the same destination.

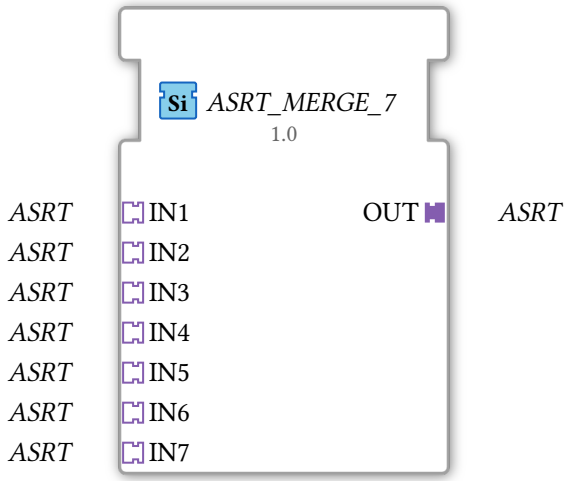
Comparison with Similar Components

- ASRT_SPLIT_2: the reverse direction for 2 outputs (for ASRT, ASRT_SPLIT_3 through ASRT_SPLIT_9 are also available).
- **ASRT_MERGE_2, ASRT_MERGE_3, ASRT_MERGE_4, ASRT_MERGE_5, ASRT_MERGE_7**: the same generic implementation with a different input count.
- AE_MERGE_6, ASR_MERGE_6: the same merge logic for the other unidirectional event adapters.

Conclusion

ASRT_MERGE_6 provides a generically implemented merge of 6 ASRT event sources into a common adapter output, completing the GEN_ASRT_MERGE family with the 6-input variant.

ASRT_MERGE_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_MERGE_7** merges 7 unidirectional ASRT adapters (set/reset/toggle event adapter, no payload data) into one common output adapter. It is the 7-input variant of ASRT_MERGE_2 and, like it, is implemented as a generic FB (GenericClassName: 'GEN_ASRT_MERGE') — the same C++ base (CGenUnidirectMergeBase), just with 7 instead of 2 sockets.

Interface Structure

Event Inputs
None. Events are received exclusively via the adapter sockets.

Event Outputs
None. Events are forwarded exclusively via the adapter plug.

Data Inputs
None.

Data Outputs
None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::ASRT`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::ASRT`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::ASRT`	Source 3.
Socket (Input 4)	`IN4`	`adapter::types::unidirectional::ASRT`	Source 4.
Socket (Input 5)	`IN5`	`adapter::types::unidirectional::ASRT`	Source 5.
Socket (Input 6)	`IN6`	`adapter::types::unidirectional::ASRT`	Source 6.
Socket (Input 7)	`IN7`	`adapter::types::unidirectional::ASRT`	Source 7.
Plug (Output)	`OUT`	`adapter::types::unidirectional::ASRT`	Merged signal.

Functionality
Whenever `SET` or `RESET` or `TOGGLE` arrives at any of the 7 sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`), it is forwarded unchanged to `OUT` as the matching event type. All event types are handled independently, and all 7 sockets are treated equally — the order in which sockets are checked follows `IN1` … `IN7`, which only matters if several events arrive within the same execution cycle. Since `ASRT` carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features
- **Generic type**: Implemented via the generic base class `CGenUnidirectMergeBase` (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (`_7`).
- **No states / algorithms**: Since `ASRT` carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers ASRT_MERGE_2 through ASRT_MERGE_7; for other input counts see `ASRT_MERGE_2`, `ASRT_MERGE_3`, `ASRT_MERGE_4`, `ASRT_MERGE_5`, `ASRT_MERGE_6`.

State Overview
The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 7 sockets is forwarded to `OUT` immediately.

Application Scenarios
- **Merging several equivalent signal sources** (e.g. 7 redundant or alternative trigger paths) into a common ASRT adapter output.
- **Simplifying networks** that would otherwise need 7 separate event connections to the same destination.

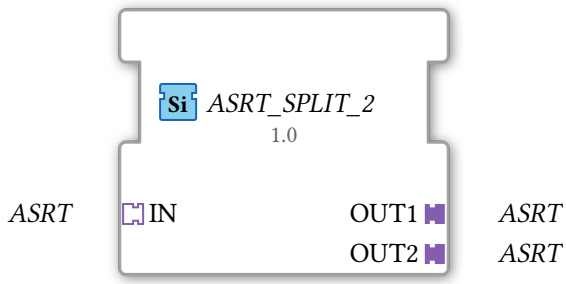
Comparison with Similar Components
- ASRT_SPLIT_2: the reverse direction for 2 outputs (for ASRT, ASRT_SPLIT_3 through ASRT_SPLIT_9 are also available).

- **ASRT_MERGE_2, ASRT_MERGE_3, ASRT_MERGE_4, ASRT_MERGE_5, ASRT_MERGE_6**: the same generic implementation with a different input count.
- AE_MERGE_7, ASR_MERGE_7: the same merge logic for the other unidirectional event adapters.

Conclusion

ASRT_MERGE_7 provides a generically implemented merge of 7 ASRT event sources into a common adapter output, completing the GEN_ASRT_MERGE family with the 7-input variant.

ASRT_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_SPLIT_2** distributes an incoming unidirectional ASRT adapter (a set/reset/toggle event triple, no payload data) to two identical output adapters. ASRT extends ASR with a third event TOGGLE (see AX_T_FF_SR). It is implemented as a generic FB (GenericClassName: 'GEN_ASRT_SPLIT') and completes the SPLIT family alongside AE_SPLIT_2 and ASR_SPLIT_2.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter socket.

Event Outputs

None. Events are forwarded exclusively via the adapter plugs.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::ASRT	Receives a SET, RESET, or TOGGLE event, which is distributed to both outputs.
Plug (Output 1)	OUT1	adapter::types::unidirectional::ASRT	First output for the duplicated event.
Plug (Output 2)	OUT2	adapter::types::unidirectional::ASRT	Second output for the duplicated event.

Functionality

As soon as a SET, RESET, or TOGGLE event arrives at adapter socket IN, it is forwarded **immediately and in parallel** as a matching event to both output plugs OUT1 and OUT2. The block performs no logic, filtering, or delay – it acts as a pure splitter at the adapter level, regardless of which of the three event types arrives.

Technical Features

- **Generic type**: Implemented via the generic base class `CGenUnidirectSplitBase` (1 socket, N plugs) – the same C++ base underlying `GEN_AE_SPLIT` and `GEN_ASR_SPLIT`.
- **Three independent event channels**: `SET`, `RESET`, and `TOGGLE` are checked separately and each distributed to both outputs.
- **No states / algorithms**: Since `ASRT` carries no data value, there is no change detection and no ECC.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming `SET`, `RESET`, or `TOGGLE` event at socket `IN` is duplicated to both outputs immediately.

Application Scenarios

- **Event distribution**: A set/reset/toggle signal (e.g. driven by `AX_T_FF_SR`) needs to be processed by two independent subsystems.
- **Parallel wiring**: Splitting an ASRT signal to simultaneously drive two actuators.

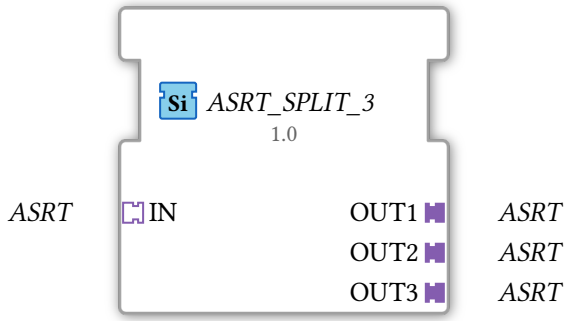
Comparison with Similar Components

- ASRT_MERGE_2: the reverse direction – merges two incoming set/reset/toggle signals into one common output, instead of distributing one signal.
- ASR_SPLIT_2: the same distribution logic for the set/reset event adapter `ASR` (no `TOGGLE`).
- AE_SPLIT_2: the same distribution logic for the pure event adapter `AE` (1 event instead of SET/RESET/TOGGLE).

Conclusion

ASRT_SPLIT_2 is a minimal generic function block for distributing a set/reset/toggle event to two ASRT adapter outputs, closing the last remaining gap in the `GEN_*_SPLIT` family for all three unidirectional event adapters (`AE`, `ASR`, `ASRT`).

ASRT_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_SPLIT_3** distributes an incoming unidirectional ASRT adapter (set/reset/toggle event adapter, no payload data) to 3 identical output adapters. It is the 3-output variant of ASRT_SPLIT_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_SPLIT'`) — the same C++ base (`CGenUnidirectSplitBase`), just with 3 instead of 2 plugs.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter socket.

Event Outputs

None. Events are forwarded exclusively via the adapter plugs.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::ASRT	Receives SET or RESET or TOGGLE, distributed to all 3 outputs.
Plug (Output 1)	OUT1	adapter::types::unidirectional::ASRT	Output 1 for the duplicated event.
Plug (Output 2)	OUT2	adapter::types::unidirectional::ASRT	Output 2 for the duplicated event.
Plug (Output 3)	OUT3	adapter::types::unidirectional::ASRT	Output 3 for the duplicated event.

Functionality

As soon as SET or RESET or TOGGLE arrives at adapter socket IN, it is forwarded **immediately and in parallel** as a matching event to all 3 output plugs (OUT1, OUT2, OUT3). The block performs no logic, filtering, or delay – it acts as a pure splitter at the adapter level.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectSplitBase (1 socket, N plugs); the number of output plugs is fixed at compile time via the generic suffix (_3).
- **No states / algorithms**: Since ASRT carries no data value, there is no change detection and no ECC.
- **Arbitrary output count**: The same implementation covers ASRT_SPLIT_2 through ASRT_SPLIT_9; for other output counts see ASRT_SPLIT_2, ASRT_SPLIT_4, ASRT_SPLIT_5, ASRT_SPLIT_6, ASRT_SPLIT_7, ASRT_SPLIT_8, ASRT_SPLIT_9.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at socket IN is duplicated to all 3 outputs immediately.

Application Scenarios

- **Event distribution**: A signal needs to be processed by 3 independent subsystems.
- **Parallel wiring**: Splitting a ASRT signal to simultaneously drive several actuators.

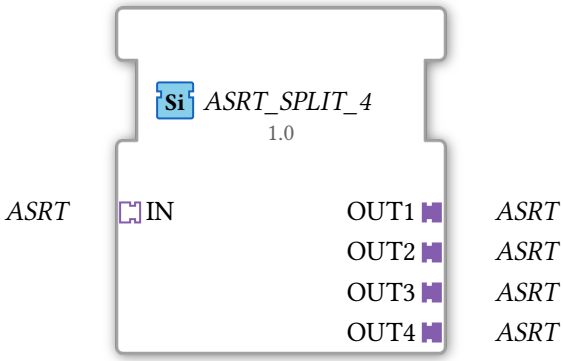
Comparison with Similar Components

- ASRT_MERGE_2: the reverse direction for 2 inputs (for ASRT, ASRT_MERGE_3 through ASRT_MERGE_7 are also available).
- **ASRT_SPLIT_2, ASRT_SPLIT_4, ASRT_SPLIT_5, ASRT_SPLIT_6, ASRT_SPLIT_7, ASRT_SPLIT_8, ASRT_SPLIT_9**: the same generic implementation with a different output count.

Conclusion

ASRT_SPLIT_3 provides a generically implemented distribution of a ASRT event to 3 adapter outputs, completing the GEN_ASRT_SPLIT family with the 3-output variant.

ASRT_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_SPLIT_4** distributes an incoming unidirectional ASRT adapter (set/reset/toggle event adapter, no payload data) to 4 identical output adapters. It is the 4-output variant of ASRT_SPLIT_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_SPLIT'`) — the same C++ base (`CGenUnidirectSplitBase`), just with 4 instead of 2 plugs.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter socket.

Event Outputs

None. Events are forwarded exclusively via the adapter plugs.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::ASRT`	Receives `SET` or `RESET` or `TOGGLE`, distributed to all 4 outputs.
Plug (Output 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	Output 1 for the duplicated event.
Plug (Output 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	Output 2 for the duplicated event.
Plug (Output 3)	`OUT3`	`adapter::types::unidirectional::ASRT`	Output 3 for the duplicated event.
Plug (Output 4)	`OUT4`	`adapter::types::unidirectional::ASRT`	Output 4 for the duplicated event.

Functionality

As soon as `SET` or `RESET` or `TOGGLE` arrives at adapter socket `IN`, it is forwarded **immediately and in parallel** as a matching event to all 4 output plugs (`OUT1`, `OUT2`, `OUT3`, `OUT4`). The block performs no logic, filtering, or delay – it acts as a pure splitter at the adapter level.

Technical Features

- **Generic type**: Implemented via the generic base class `CGenUnidirectSplitBase` (1 socket, N plugs); the number of output plugs is fixed at compile time via the generic suffix (_4).
- **No states / algorithms**: Since `ASRT` carries no data value, there is no change detection and no ECC.
- **Arbitrary output count**: The same implementation covers ASRT_SPLIT_2 through ASRT_SPLIT_9; for other output counts see `ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_5`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at socket `IN` is duplicated to all 4 outputs immediately.

Application Scenarios

- **Event distribution**: A signal needs to be processed by 4 independent subsystems.
- **Parallel wiring**: Splitting a ASRT signal to simultaneously drive several actuators.

Comparison with Similar Components

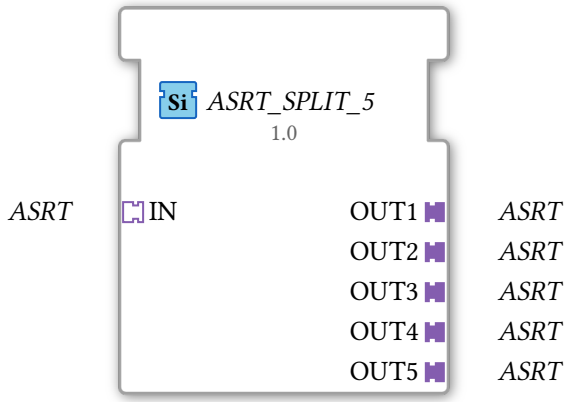
- **ASRT_MERGE_2**: the reverse direction for 2 inputs (for ASRT, ASRT_MERGE_3 through ASRT_MERGE_7 are also available).

- **ASRT_SPLIT_2, ASRT_SPLIT_3, ASRT_SPLIT_5, ASRT_SPLIT_6, ASRT_SPLIT_7, ASRT_SPLIT_8, ASRT_SPLIT_9**: the same generic implementation with a different output count.

Conclusion

`ASRT_SPLIT_4` provides a generically implemented distribution of a `ASRT` event to 4 adapter outputs, completing the `GEN_ASRT_SPLIT` family with the 4-output variant.

ASRT_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_SPLIT_5** distributes an incoming unidirectional ASRT adapter (set/reset/toggle event adapter, no payload data) to 5 identical output adapters. It is the 5-output variant of ASRT_SPLIT_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_SPLIT'`) — the same C++ base (`CGenUnidirectSplitBase`), just with 5 instead of 2 plugs.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter socket.

Event Outputs

None. Events are forwarded exclusively via the adapter plugs.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::ASRT`	Receives `SET` or `RESET` or `TOGGLE`, distributed to all 5 outputs.
Plug (Output 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	Output 1 for the duplicated event.
Plug (Output 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	Output 2 for the duplicated event.
Plug (Output 3)	`OUT3`	`adapter::types::unidirectional::ASRT`	Output 3 for the duplicated event.
Plug (Output 4)	`OUT4`	`adapter::types::unidirectional::ASRT`	Output 4 for the duplicated event.
Plug (Output 5)	`OUT5`	`adapter::types::unidirectional::ASRT`	Output 5 for the duplicated event.

Functionality

As soon as `SET` or `RESET` or `TOGGLE` arrives at adapter socket `IN`, it is forwarded **immediately and in parallel** as a matching event to all 5 output plugs (`OUT1`, `OUT2`, `OUT3`, `OUT4`, `OUT5`). The block performs no logic, filtering, or delay – it acts as a pure splitter at the adapter level.

Technical Features

- **Generic type**: Implemented via the generic base class `CGenUnidirectSplitBase` (1 socket, N plugs); the number of output plugs is fixed at compile time via the generic suffix (`_5`).
- **No states / algorithms**: Since `ASRT` carries no data value, there is no change detection and no ECC.
- **Arbitrary output count**: The same implementation covers ASRT_SPLIT_2 through ASRT_SPLIT_9; for other output counts see `ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at socket `IN` is duplicated to all 5 outputs immediately.

Application Scenarios

- **Event distribution**: A signal needs to be processed by 5 independent subsystems.
- **Parallel wiring**: Splitting a ASRT signal to simultaneously drive several actuators.

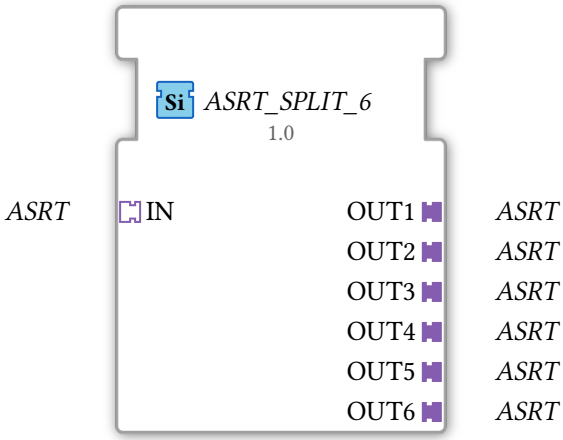
Comparison with Similar Components

- ASRT_MERGE_2: the reverse direction for 2 inputs (for ASRT, ASRT_MERGE_3 through ASRT_MERGE_7 are also available).
- **`ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`**: the same generic implementation with a different output count.

Conclusion

`ASRT_SPLIT_5` provides a generically implemented distribution of a `ASRT` event to 5 adapter outputs, completing the `GEN_ASRT_SPLIT` family with the 5-output variant.

ASRT_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_SPLIT_6** distributes an incoming unidirectional ASRT adapter (set/reset/toggle event adapter, no payload data) to 6 identical output adapters. It is the 6-output variant of ASRT_SPLIT_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_SPLIT'`) — the same C++ base (`CGenUnidirectSplitBase`), just with 6 instead of 2 plugs.

Interface Structure

Event Inputs
None. Events are received exclusively via the adapter socket.

Event Outputs
None. Events are forwarded exclusively via the adapter plugs.

Data Inputs
None.

Data Outputs
None.

Adapters

Role	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::ASRT`	Receives `SET` or `RESET` or `TOGGLE`, distributed to all 6 outputs.
Plug (Output 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	Output 1 for the duplicated event.
Plug (Output 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	Output 2 for the duplicated event.
Plug (Output 3)	`OUT3`	`adapter::types::unidirectional::ASRT`	Output 3 for the duplicated event.
Plug (Output 4)	`OUT4`	`adapter::types::unidirectional::ASRT`	Output 4 for the duplicated event.
Plug (Output 5)	`OUT5`	`adapter::types::unidirectional::ASRT`	Output 5 for the duplicated event.
Plug (Output 6)	`OUT6`	`adapter::types::unidirectional::ASRT`	Output 6 for the duplicated event.

Functionality
As soon as `SET` or `RESET` or `TOGGLE` arrives at adapter socket `IN`, it is forwarded **immediately and in parallel** as a matching event to all 6 output plugs (`OUT1`, `OUT2`, `OUT3`, `OUT4`, `OUT5`, `OUT6`). The block performs no logic, filtering, or delay – it acts as a pure splitter at the adapter level.

Technical Features
- **Generic type**: Implemented via the generic base class `CGenUnidirectSplitBase` (1 socket, N plugs); the number of output plugs is fixed at compile time via the generic suffix (`_6`).
- **No states / algorithms**: Since `ASRT` carries no data value, there is no change detection and no ECC.
- **Arbitrary output count**: The same implementation covers ASRT_SPLIT_2 through ASRT_SPLIT_9; for other output counts see `ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`.

State Overview
The function block has **no state machine**. Its behavior is purely combinational: every incoming event at socket `IN` is duplicated to all 6 outputs immediately.

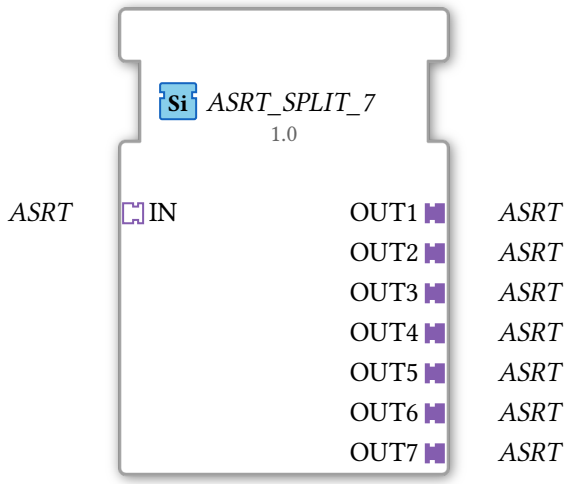
Application Scenarios
- **Event distribution**: A signal needs to be processed by 6 independent subsystems.
- **Parallel wiring**: Splitting a ASRT signal to simultaneously drive several actuators.

Comparison with Similar Components
- **ASRT_MERGE_2**: the reverse direction for 2 inputs (for ASRT, ASRT_MERGE_3 through ASRT_MERGE_7 are also available).
- **`ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_7`, `ASRT_SPLIT_8`, `ASRT_SPLIT_9`**: the same generic implementation with a different output count.

Conclusion

`ASRT_SPLIT_6` provides a generically implemented distribution of a `ASRT` event to 6 adapter outputs, completing the `GEN_ASRT_SPLIT` family with the 6-output variant.

ASRT_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_SPLIT_7** distributes an incoming unidirectional ASRT adapter (set/reset/toggle event adapter, no payload data) to 7 identical output adapters. It is the 7-output variant of ASRT_SPLIT_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_SPLIT'`) — the same C++ base (`CGenUnidirectSplitBase`), just with 7 instead of 2 plugs.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter socket.

Event Outputs

None. Events are forwarded exclusively via the adapter plugs.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::ASRT`	Receives `SET` or `RESET` or `TOGGLE`, distributed to all 7 outputs.
Plug (Output 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	Output 1 for the duplicated event.
Plug (Output 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	Output 2 for the duplicated event.
Plug (Output 3)	`OUT3`	`adapter::types::unidirectional::ASRT`	Output 3 for the duplicated event.
Plug (Output 4)	`OUT4`	`adapter::types::unidirectional::ASRT`	Output 4 for the duplicated event.
Plug (Output 5)	`OUT5`	`adapter::types::unidirectional::ASRT`	Output 5 for the duplicated event.
Plug (Output 6)	`OUT6`	`adapter::types::unidirectional::ASRT`	Output 6 for the duplicated event.

Plug (Output 7)	OUT7	adapter::types::unidirectional::ASRT	Output 7 for the duplicated event.

Functionality

As soon as SET or RESET or TOGGLE arrives at adapter socket IN, it is forwarded **immediately and in parallel** as a matching event to all 7 output plugs (OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7). The block performs no logic, filtering, or delay – it acts as a pure splitter at the adapter level.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectSplitBase (1 socket, N plugs); the number of output plugs is fixed at compile time via the generic suffix (_7).
- **No states / algorithms**: Since ASRT carries no data value, there is no change detection and no ECC.
- **Arbitrary output count**: The same implementation covers ASRT_SPLIT_2 through ASRT_SPLIT_9; for other output counts see ASRT_SPLIT_2, ASRT_SPLIT_3, ASRT_SPLIT_4, ASRT_SPLIT_5, ASRT_SPLIT_6, ASRT_SPLIT_8, ASRT_SPLIT_9.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at socket IN is duplicated to all 7 outputs immediately.

Application Scenarios

- **Event distribution**: A signal needs to be processed by 7 independent subsystems.
- **Parallel wiring**: Splitting a ASRT signal to simultaneously drive several actuators.

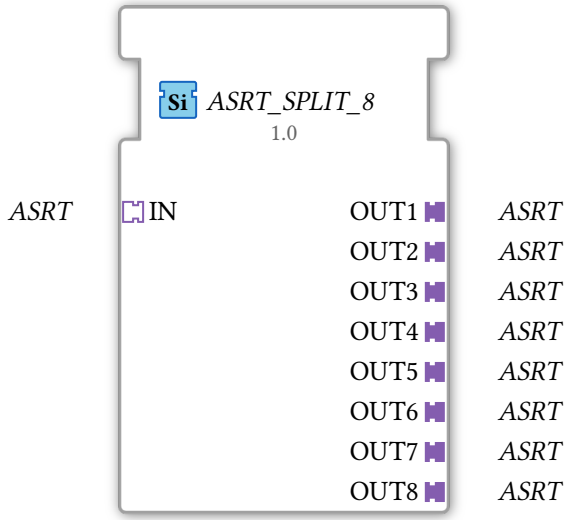
Comparison with Similar Components

- ASRT_MERGE_2: the reverse direction for 2 inputs (for ASRT, ASRT_MERGE_3 through ASRT_MERGE_7 are also available).
- **ASRT_SPLIT_2, ASRT_SPLIT_3, ASRT_SPLIT_4, ASRT_SPLIT_5, ASRT_SPLIT_6, ASRT_SPLIT_8, ASRT_SPLIT_9**: the same generic implementation with a different output count.

Conclusion

ASRT_SPLIT_7 provides a generically implemented distribution of a ASRT event to 7 adapter outputs, completing the GEN_ASRT_SPLIT family with the 7-output variant.

ASRT_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_SPLIT_8** distributes an incoming unidirectional ASRT adapter (set/reset/toggle event adapter, no payload data) to 8 identical output adapters. It is the 8-output variant of

ASRT_SPLIT_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_SPLIT'`) — the same C++ base (`CGenUnidirectSplitBase`), just with 8 instead of 2 plugs.

Interface Structure

Event Inputs
None. Events are received exclusively via the adapter socket.

Event Outputs
None. Events are forwarded exclusively via the adapter plugs.

Data Inputs
None.

Data Outputs
None.

Adapters

Role	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::ASRT`	Receives `SET` or `RESET` or `TOGGLE`, distributed to all 8 outputs.
Plug (Output 1)	`OUT1`	`adapter::types::unidirectional::ASRT`	Output 1 for the duplicated event.
Plug (Output 2)	`OUT2`	`adapter::types::unidirectional::ASRT`	Output 2 for the duplicated event.
Plug (Output 3)	`OUT3`	`adapter::types::unidirectional::ASRT`	Output 3 for the duplicated event.
Plug (Output 4)	`OUT4`	`adapter::types::unidirectional::ASRT`	Output 4 for the duplicated event.
Plug (Output 5)	`OUT5`	`adapter::types::unidirectional::ASRT`	Output 5 for the duplicated event.
Plug (Output 6)	`OUT6`	`adapter::types::unidirectional::ASRT`	Output 6 for the duplicated event.
Plug (Output 7)	`OUT7`	`adapter::types::unidirectional::ASRT`	Output 7 for the duplicated event.
Plug (Output 8)	`OUT8`	`adapter::types::unidirectional::ASRT`	Output 8 for the duplicated event.

Functionality
As soon as `SET` or `RESET` or `TOGGLE` arrives at adapter socket `IN`, it is forwarded **immediately and in parallel** as a matching event to all 8 output plugs (`OUT1`, `OUT2`, `OUT3`, `OUT4`, `OUT5`, `OUT6`, `OUT7`, `OUT8`). The block performs no logic, filtering, or delay – it acts as a pure splitter at the adapter level.

Technical Features
- **Generic type**: Implemented via the generic base class `CGenUnidirectSplitBase` (1 socket, N plugs); the number of output plugs is fixed at compile time via the generic suffix (`_8`).
- **No states / algorithms**: Since `ASRT` carries no data value, there is no change detection and no ECC.
- **Arbitrary output count**: The same implementation covers ASRT_SPLIT_2 through ASRT_SPLIT_9; for other output counts see `ASRT_SPLIT_2`, `ASRT_SPLIT_3`, `ASRT_SPLIT_4`, `ASRT_SPLIT_5`, `ASRT_SPLIT_6`, `ASRT_SPLIT_7`, `ASRT_SPLIT_9`.

State Overview
The function block has **no state machine**. Its behavior is purely combinational: every incoming event at socket `IN` is duplicated to all 8 outputs immediately.

Application Scenarios
- **Event distribution**: A signal needs to be processed by 8 independent subsystems.
- **Parallel wiring**: Splitting a ASRT signal to simultaneously drive several actuators.

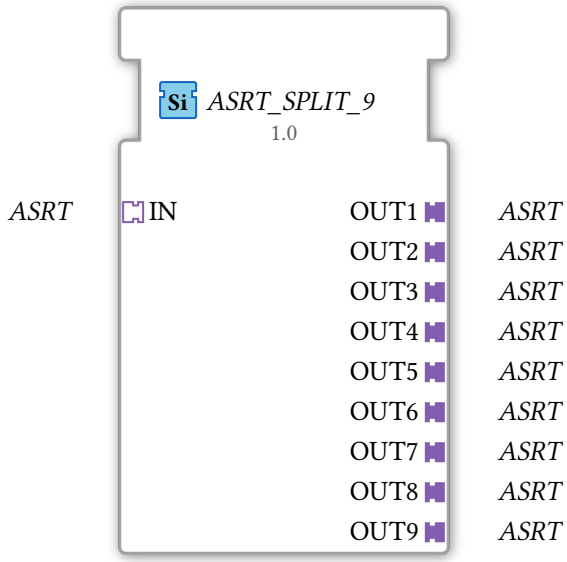
Comparison with Similar Components

- ASRT_MERGE_2: the reverse direction for 2 inputs (for ASRT, ASRT_MERGE_3 through ASRT_MERGE_7 are also available).
- **ASRT_SPLIT_2**, **ASRT_SPLIT_3**, **ASRT_SPLIT_4**, **ASRT_SPLIT_5**, **ASRT_SPLIT_6**, **ASRT_SPLIT_7**, **ASRT_SPLIT_9**: the same generic implementation with a different output count.

Conclusion

`ASRT_SPLIT_8` provides a generically implemented distribution of a `ASRT` event to 8 adapter outputs, completing the `GEN_ASRT_SPLIT` family with the 8-output variant.

ASRT_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASRT_SPLIT_9** distributes an incoming unidirectional ASRT adapter (set/reset/toggle event adapter, no payload data) to 9 identical output adapters. It is the 9-output variant of ASRT_SPLIT_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASRT_SPLIT'`) — the same C++ base (`CGenUnidirectSplitBase`), just with 9 instead of 2 plugs.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter socket.

Event Outputs

None. Events are forwarded exclusively via the adapter plugs.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::ASRT`	Receives `SET` or `RESET` or `TOGGLE`, distributed to all 9 outputs.

Plug (Output 1)	OUT1	adapter::types::unidirectional::ASRT Output 1 for the duplicated event.
Plug (Output 2)	OUT2	adapter::types::unidirectional::ASRT Output 2 for the duplicated event.
Plug (Output 3)	OUT3	adapter::types::unidirectional::ASRT Output 3 for the duplicated event.
Plug (Output 4)	OUT4	adapter::types::unidirectional::ASRT Output 4 for the duplicated event.
Plug (Output 5)	OUT5	adapter::types::unidirectional::ASRT Output 5 for the duplicated event.
Plug (Output 6)	OUT6	adapter::types::unidirectional::ASRT Output 6 for the duplicated event.
Plug (Output 7)	OUT7	adapter::types::unidirectional::ASRT Output 7 for the duplicated event.
Plug (Output 8)	OUT8	adapter::types::unidirectional::ASRT Output 8 for the duplicated event.
Plug (Output 9)	OUT9	adapter::types::unidirectional::ASRT Output 9 for the duplicated event.

Functionality

As soon as SET or RESET or TOGGLE arrives at adapter socket IN, it is forwarded **immediately and in parallel** as a matching event to all 9 output plugs (OUT1, OUT2, OUT3, OUT4, OUT5, OUT6, OUT7, OUT8, OUT9). The block performs no logic, filtering, or delay – it acts as a pure splitter at the adapter level.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectSplitBase (1 socket, N plugs); the number of output plugs is fixed at compile time via the generic suffix (_9).
- **No states / algorithms**: Since ASRT carries no data value, there is no change detection and no ECC.
- **Arbitrary output count**: The same implementation covers ASRT_SPLIT_2 through ASRT_SPLIT_9; for other output counts see ASRT_SPLIT_2, ASRT_SPLIT_3, ASRT_SPLIT_4, ASRT_SPLIT_5, ASRT_SPLIT_6, ASRT_SPLIT_7, ASRT_SPLIT_8.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at socket IN is duplicated to all 9 outputs immediately.

Application Scenarios

- **Event distribution**: A signal needs to be processed by 9 independent subsystems.
- **Parallel wiring**: Splitting a ASRT signal to simultaneously drive several actuators.

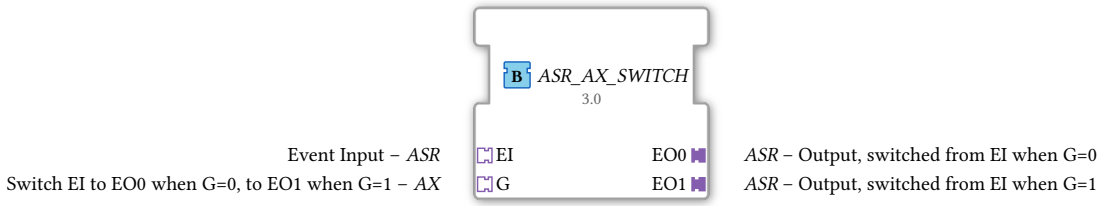
Comparison with Similar Components

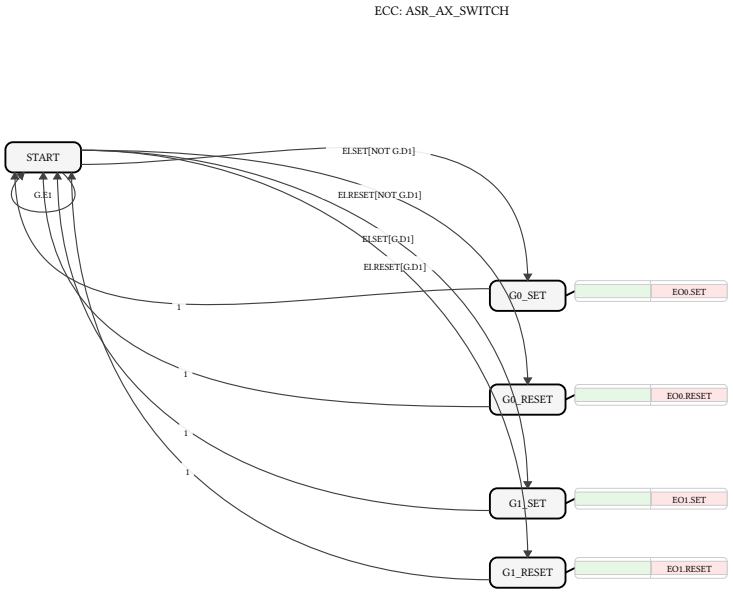
- ASRT_MERGE_2: the reverse direction for 2 inputs (for ASRT, ASRT_MERGE_3 through ASRT_MERGE_7 are also available).
- **ASRT_SPLIT_2, ASRT_SPLIT_3, ASRT_SPLIT_4, ASRT_SPLIT_5, ASRT_SPLIT_6, ASRT_SPLIT_7, ASRT_SPLIT_8**: the same generic implementation with a different output count.

Conclusion

ASRT_SPLIT_9 provides a generically implemented distribution of a ASRT event to 9 adapter outputs, completing the GEN_ASRT_SPLIT family with the 9-output variant.

ASR_AX_SWITCH





• – ○ • – ○ • – ○ •

Introduction

The function block **ASR_AX_SWITCH** serves as a demultiplexer for asynchronous set/reset events (ASR). It forwards an incoming SET or RESET event, based on the state of a binary control input **G**, to either one of two outputs (**EO0** or **EO1**). This block enables demand-based distribution of control signals in automation applications.

Interface Structure

Event Inputs

The function block does not have traditional event inputs via *EventInputs* – events are received via the **Socket Adapter `EI`** of type `ASR`. This adapter provides the two events `SET` and `RESET`.

Event Outputs

The event outputs are provided via the **plug adapters `EO0`** and **`EO1`** (both type `ASR`). Each of these adapters has the outputs `SET` and `RESET`:

Adapter	Event	Description
`EO0.SET`	SET	Triggered when a `EI.SET` event occurs and `G.D1 == 0`
`EO0.RESET`	RESET	Triggered when a `EI.RESET` event occurs and `G.D1 == 0`
`EO1.SET`	SET	Triggered when a `EI.SET` event occurs and `G.D1 == 1`
`EO1.RESET`	RESET	Triggered when a `EI.RESET` event occurs and `G.D1 == 1`

Data Inputs

The FB has no separate data inputs. The control variable **G** is obtained via the **Socket Adapter `G`** of type `AX` – the logical signal is present at the data output `D1` of the adapter.

Data Outputs

There are no explicit data outputs. Control information is transmitted exclusively via the event outputs of the ASR adapters.

Adapters

Name	Direction	Type	Description
EI	Socket (Input)	adapter::types::unidirectional::ASR	Receives the SET/RESET events to be distributed
G	Socket (Input)	adapter::types::unidirectional::AX	Provides the binary toggle signal via D1 (0 ⟶ Channel 0, 1 ⟶ Channel 1)
EO0	Plug (Output)	adapter::types::unidirectional::ASR	Output for Channel 0 (with G.D1 = 0)
EO1	Plug (Output)	adapter::types::unidirectional::ASR	Output for Channel 1 (with G.D1 = 1)

Functionality

The function block operates in an event-driven manner. After startup, it is in the **START** state and waits for incoming events via the EI adapter:

- **EI.SET** and **EI.RESET** are only processed if a valid binary value is simultaneously present at G.D1 via the G adapter.
- The condition for the transition is:
- EI.SET[NOT G.D1] ⟶ **G0_SET** ⟶ Output EO0.SET
- EI.SET[G.D1] ⟶ **G1_SET** ⟶ Output EO1.SET
- EI.RESET[NOT G.D1] ⟶ **G0_RESET** ⟶ Output EO0.RESET
- EI.RESET[G.D1] ⟶ **G1_RESET** ⟶ Output EO1.RESET
- After the initial event is triggered, the function block immediately returns to the **START** state (unconditional transition with 1).
- Additionally, there is a transition from **START** to **START** at **G.E1** – this allows for asynchronous updating of the control signal without immediate event propagation (serves, for example, for synchronization or initialization).

Processing is deterministic without intermediate event storage.

Technical Features

- **Adapter-based communication**: The function block uses only adapters (sockets/plugs) for event and data exchange – this enables flexible integration within the 4diac IDE.
- **No state storage**: The selection of the target channel is purely combinatorial at the time of the incoming event. No internal state is maintained.
- **Full synchronicity**: After each output event, the function block returns to its initial state – events cannot be stacked.
- **Embedded license**: The code is released under the Eclipse Public License 2.0.

Status Overview

Status	Description	Action
START	Waiting for incoming SET/RESET event	–
G0_SET	Receive EI.SET at G.D1=0	Output EO0.SET
G1_SET	Receive EI.SET at G.D1=1	Output EO1.SET
G0_RESET	Receive EI.RESET at G.D1=0	Output EO0.RESET
G1_RESET	Receive EI.RESET at G.D1=1	Output EO1.RESET

State transitions follow the ECC scheme: A condition leads from START to the action state, and from there, without further conditions, immediately back to START.

Application Scenarios

- **Channel Switching**: In a modular controller, a central set/reset command should be forwarded to different actuators (e.g., valves, motors) depending on the operating mode.
- **Safety Logic**: Depending on an enable signal (G), resets are only routed to the active channel.

- **Redundancy Control**: The signal `G` selects between the main and backup channels.
- **Sequential Processes**: Multiple processing stations are controlled sequentially by switching `G`.

Comparison with Similar Function Blocks

Property	ASR_AX_SWITCH	Simple event multiplexer (e.g., SELECT)
Event Type	SET/RESET (ASR Adapter)	Any single event
Control	Binary via AX adapter (with `D1`)	Boolean data input
Output Type	Two ASR adapters	Depending on the type, single or multiple events
Stateful	No (no storage)	Mostly stateless
Adapter interface	Yes	Often directly via event/data ports

The ASR_AX_SWITCH is specifically optimized for asynchronous set/reset signals and integrates seamlessly into the ASR adapter world of the 4diac library. Compared to generic multiplexers, there is no need to manually extract the adapter events.

Conclusion

The **ASR_AX_SWITCH** is a compact, efficient demultiplexer for the directed routing of ASR control events. Thanks to its adapter-based interface and clear decision logic, it is ideally suited for modular automation projects where a single control signal needs to be selectively routed to different paths. The simple state machine guarantees fast, deterministic response times without side effects.

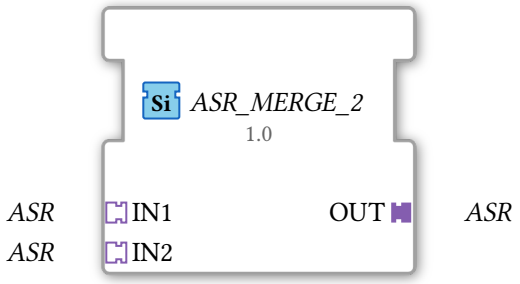
ASR_AX_SWITCH –

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ASR_MERGE_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_MERGE_2** merges two unidirectional ASR adapters (a set/reset event pair, no payload data) into one common output adapter. It is the reverse of ASR_SPLIT_2: instead of distributing a set/reset signal to two outputs, it merges two independent set/reset sources into a common output. The block is implemented as a generic FB (`GenericClassName: 'GEN_ASR_MERGE'`).

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	IN1	adapter::types::unidirectional::ASR	First SET/RESET source.
Socket (Input 2)	IN2	adapter::types::unidirectional::ASR	Second SET/RESET source.
Plug (Output)	OUT	adapter::types::unidirectional::ASR	Merged SET/RESET signal.

Functionality

Whenever a SET event arrives at IN1 **or** IN2, a SET is forwarded to OUT; likewise, a RESET at IN1/IN2 is forwarded to OUT as a RESET. Both event types are handled independently – a SET at IN1 has no effect on whether a RESET at IN2 is forwarded. Since ASR carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectMergeBase (N sockets, 1 plug) – the same C++ base underlying GEN_AE_MERGE and GEN_ASRT_MERGE.
- **Two independent event channels**: SET and RESET are checked and forwarded separately; one socket could, for example, only ever deliver SET events while the other only delivers RESET, without the two interfering with each other.
- **No states / algorithms**: Since ASR carries no data value, there is no change detection and no ECC.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming SET or RESET event at IN1 or IN2 is forwarded to OUT immediately as the matching event type.

Application Scenarios

- **Merging redundant set/reset sources**: Two independent operator stations or safety paths should both be able to set or reset the same ASR output.
- **Simplifying networks** that would otherwise need two separate SET/RESET event connection pairs to the same destination.

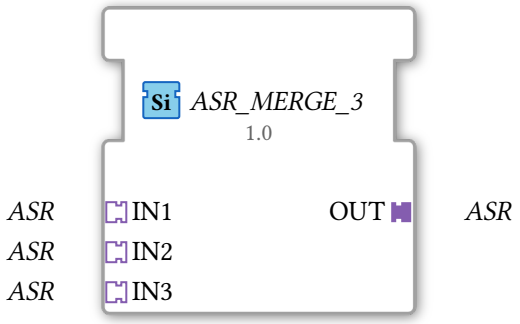
Comparison with Similar Components

- ASR_SPLIT_2: the reverse direction – distributes one incoming set/reset signal to two outputs, instead of merging two inputs.
- AE_MERGE_2: the same merge logic for the pure event adapter AE (1 event instead of SET/RESET).
- ASRT_MERGE_2: the same merge logic for the set/reset/toggle event adapter ASRT (adds TOGGLE).

Conclusion

ASR_MERGE_2 is a minimal generic function block for merging two set/reset event sources into a common ASR adapter output, closing the gap previously noted as "hypothetical" next to ASR_SPLIT_2.

ASR_MERGE_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_MERGE_3** merges 3 unidirectional ASR adapters (set/reset event adapter, no payload data) into one common output adapter. It is the 3-input variant of ASR_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASR_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 3 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::ASR`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::ASR`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::ASR`	Source 3.
Plug (Output)	`OUT`	`adapter::types::unidirectional::ASR`	Merged signal.

Functionality

Whenever `SET` or `RESET` arrives at any of the 3 sockets (`IN1`, `IN2`, `IN3`), it is forwarded unchanged to `OUT` as the matching event type. All event types are handled independently, and all 3 sockets are treated equally — the order in which sockets are checked follows `IN1` … `IN3`, which only matters if several events arrive within the same execution cycle. Since `ASR` carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class `CGenUnidirectMergeBase` (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (`_3`).
- **No states / algorithms**: Since `ASR` carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers ASR_MERGE_2 through ASR_MERGE_7; for other input counts see `ASR_MERGE_2`, `ASR_MERGE_4`, `ASR_MERGE_5`, `ASR_MERGE_6`, `ASR_MERGE_7`.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 3 sockets is forwarded to `OUT` immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 3 redundant or alternative trigger paths) into a common ASR adapter output.
- **Simplifying networks** that would otherwise need 3 separate event connections to the same destination.

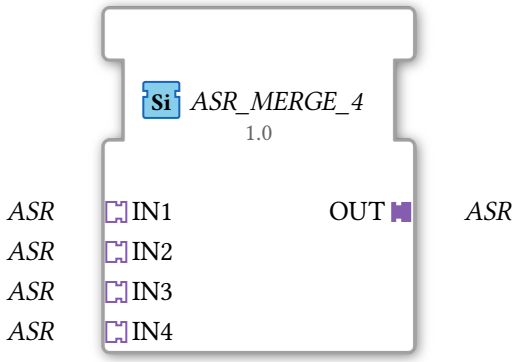
Comparison with Similar Components

- **ASR_SPLIT_2**: the reverse direction for 2 outputs (for ASRT, ASR_SPLIT_3 through ASR_SPLIT_9 are also available).
- **`ASR_MERGE_2`, `ASR_MERGE_4`, `ASR_MERGE_5`, `ASR_MERGE_6`, `ASR_MERGE_7`**: the same generic implementation with a different input count.
- AE_MERGE_3, ASRT_MERGE_3: the same merge logic for the other unidirectional event adapters.

Conclusion

`ASR_MERGE_3` provides a generically implemented merge of 3 `ASR` event sources into a common adapter output, completing the `GEN_ASR_MERGE` family with the 3-input variant.

ASR_MERGE_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_MERGE_4** merges 4 unidirectional ASR adapters (set/reset event adapter, no payload data) into one common output adapter. It is the 4-input variant of ASR_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASR_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 4 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	IN1	adapter::types::unidirectional::ASR	Source 1.
Socket (Input 2)	IN2	adapter::types::unidirectional::ASR	Source 2.
Socket (Input 3)	IN3	adapter::types::unidirectional::ASR	Source 3.
Socket (Input 4)	IN4	adapter::types::unidirectional::ASR	Source 4.
Plug (Output)	OUT	adapter::types::unidirectional::ASR	Merged signal.

Functionality

Whenever SET or RESET arrives at any of the 4 sockets (IN1, IN2, IN3, IN4), it is forwarded unchanged to OUT as the matching event type. All event types are handled independently, and all 4 sockets are treated equally — the order in which sockets are checked follows IN1 … IN4, which only matters if several events arrive within the same execution cycle. Since ASR carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectMergeBase (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (_4).
- **No states / algorithms**: Since ASR carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers ASR_MERGE_2 through ASR_MERGE_7; for other input counts see ASR_MERGE_2, ASR_MERGE_3, ASR_MERGE_5, ASR_MERGE_6, ASR_MERGE_7.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 4 sockets is forwarded to OUT immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 4 redundant or alternative trigger paths) into a common ASR adapter output.
- **Simplifying networks** that would otherwise need 4 separate event connections to the same destination.

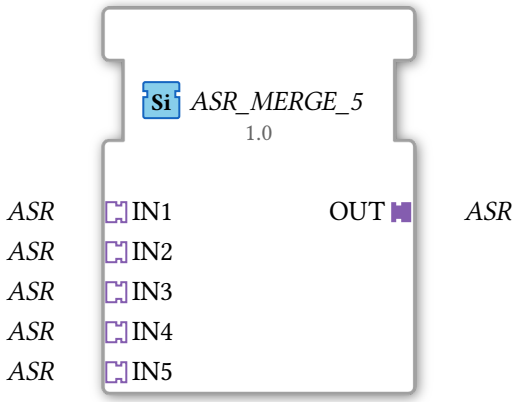
Comparison with Similar Components

- ASR_SPLIT_2: the reverse direction for 2 outputs (for ASRT, ASR_SPLIT_3 through ASR_SPLIT_9 are also available).
- **ASR_MERGE_2, ASR_MERGE_3, ASR_MERGE_5, ASR_MERGE_6, ASR_MERGE_7**: the same generic implementation with a different input count.
- AE_MERGE_4, ASRT_MERGE_4: the same merge logic for the other unidirectional event adapters.

Conclusion

ASR_MERGE_4 provides a generically implemented merge of 4 ASR event sources into a common adapter output, completing the GEN_ASR_MERGE family with the 4-input variant.

ASR_MERGE_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_MERGE_5** merges 5 unidirectional ASR adapters (set/reset event adapter, no payload data) into one common output adapter. It is the 5-input variant of ASR_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASR_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 5 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::ASR`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::ASR`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::ASR`	Source 3.
Socket (Input 4)	`IN4`	`adapter::types::unidirectional::ASR`	Source 4.
Socket (Input 5)	`IN5`	`adapter::types::unidirectional::ASR`	Source 5.
Plug (Output)	`OUT`	`adapter::types::unidirectional::ASR`	Merged signal.

Functionality

Whenever `SET` or `RESET` arrives at any of the 5 sockets (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`), it is forwarded unchanged to `OUT` as the matching event type. All event types are handled independently, and all 5 sockets are treated equally — the order in which sockets are checked follows `IN1` … `IN5`, which only matters if several events arrive within the same execution cycle. Since `ASR` carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class `CGenUnidirectMergeBase` (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (_5).
- **No states / algorithms**: Since `ASR` carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers ASR_MERGE_2 through ASR_MERGE_7; for other input counts see `ASR_MERGE_2`, `ASR_MERGE_3`, `ASR_MERGE_4`, `ASR_MERGE_6`, `ASR_MERGE_7`.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 5 sockets is forwarded to `OUT` immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 5 redundant or alternative trigger paths) into a common ASR adapter output.
- **Simplifying networks** that would otherwise need 5 separate event connections to the same destination.

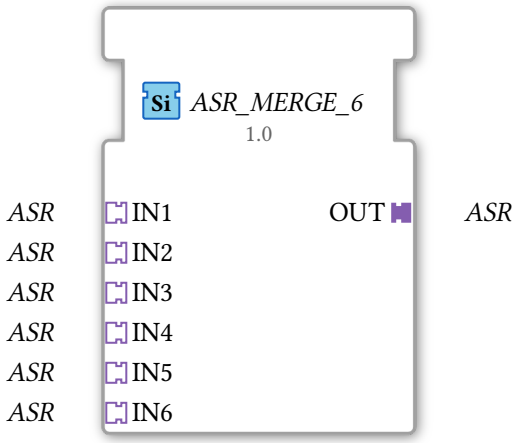
Comparison with Similar Components

- **ASR_SPLIT_2**: the reverse direction for 2 outputs (for ASRT, ASR_SPLIT_3 through ASR_SPLIT_9 are also available).
- **`ASR_MERGE_2, ASR_MERGE_3, ASR_MERGE_4, ASR_MERGE_6, ASR_MERGE_7`**: the same generic implementation with a different input count.
- AE_MERGE_5, ASRT_MERGE_5: the same merge logic for the other unidirectional event adapters.

Conclusion

`ASR_MERGE_5` provides a generically implemented merge of 5 `ASR` event sources into a common adapter output, completing the `GEN_ASR_MERGE` family with the 5-input variant.

ASR_MERGE_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_MERGE_6** merges 6 unidirectional ASR adapters (set/reset event adapter, no payload data) into one common output adapter. It is the 6-input variant of ASR_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASR_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 6 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	IN1	adapter::types::unidirectional::ASR	Source 1.
Socket (Input 2)	IN2	adapter::types::unidirectional::ASR	Source 2.
Socket (Input 3)	IN3	adapter::types::unidirectional::ASR	Source 3.
Socket (Input 4)	IN4	adapter::types::unidirectional::ASR	Source 4.
Socket (Input 5)	IN5	adapter::types::unidirectional::ASR	Source 5.
Socket (Input 6)	IN6	adapter::types::unidirectional::ASR	Source 6.
Plug (Output)	OUT	adapter::types::unidirectional::ASR	Merged signal.

Functionality

Whenever SET or RESET arrives at any of the 6 sockets (IN1, IN2, IN3, IN4, IN5, IN6), it is forwarded unchanged to OUT as the matching event type. All event types are handled independently, and all 6 sockets are treated equally — the order in which sockets are checked follows IN1 … IN6, which only matters if several events arrive within the same execution cycle. Since ASR carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectMergeBase (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (_6).
- **No states / algorithms**: Since ASR carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers ASR_MERGE_2 through ASR_MERGE_7; for other input counts see ASR_MERGE_2, ASR_MERGE_3, ASR_MERGE_4, ASR_MERGE_5, ASR_MERGE_7.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 6 sockets is forwarded to OUT immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 6 redundant or alternative trigger paths) into a common ASR adapter output.
- **Simplifying networks** that would otherwise need 6 separate event connections to the same destination.

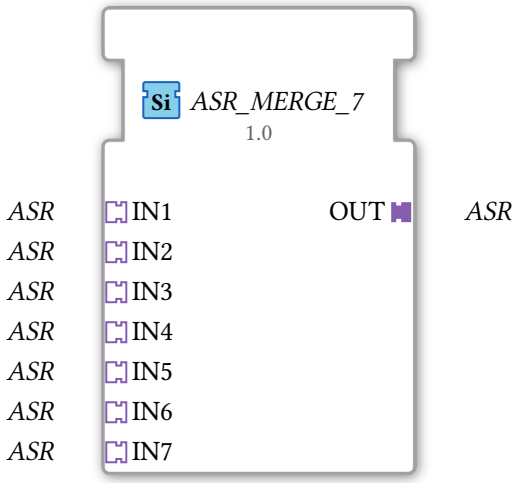
Comparison with Similar Components

- ASR_SPLIT_2: the reverse direction for 2 outputs (for ASRT, ASR_SPLIT_3 through ASR_SPLIT_9 are also available).
- **ASR_MERGE_2, ASR_MERGE_3, ASR_MERGE_4, ASR_MERGE_5, ASR_MERGE_7**: the same generic implementation with a different input count.
- AE_MERGE_6, ASRT_MERGE_6: the same merge logic for the other unidirectional event adapters.

Conclusion

ASR_MERGE_6 provides a generically implemented merge of 6 ASR event sources into a common adapter output, completing the GEN_ASR_MERGE family with the 6-input variant.

ASR_MERGE_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_MERGE_7** merges 7 unidirectional ASR adapters (set/reset event adapter, no payload data) into one common output adapter. It is the 7-input variant of ASR_MERGE_2 and, like it, is implemented as a generic FB (`GenericClassName: 'GEN_ASR_MERGE'`) — the same C++ base (`CGenUnidirectMergeBase`), just with 7 instead of 2 sockets.

Interface Structure

Event Inputs

None. Events are received exclusively via the adapter sockets.

Event Outputs

None. Events are forwarded exclusively via the adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Role	Name	Type	Description
Socket (Input 1)	`IN1`	`adapter::types::unidirectional::ASR`	Source 1.
Socket (Input 2)	`IN2`	`adapter::types::unidirectional::ASR`	Source 2.
Socket (Input 3)	`IN3`	`adapter::types::unidirectional::ASR`	Source 3.
Socket (Input 4)	`IN4`	`adapter::types::unidirectional::ASR`	Source 4.
Socket (Input 5)	`IN5`	`adapter::types::unidirectional::ASR`	Source 5.
Socket (Input 6)	`IN6`	`adapter::types::unidirectional::ASR`	Source 6.
Socket (Input 7)	`IN7`	`adapter::types::unidirectional::ASR`	Source 7.
Plug (Output)	`OUT`	`adapter::types::unidirectional::ASR`	Merged signal.

Functionality

Whenever SET or RESET arrives at any of the 7 sockets (IN1, IN2, IN3, IN4, IN5, IN6, IN7), it is forwarded unchanged to OUT as the matching event type. All event types are handled independently, and all 7 sockets are treated equally — the order in which sockets are checked follows IN1 ... IN7, which only matters if several events arrive within the same execution cycle. Since ASR carries no data value, there is nothing to arbitrate beyond the respective event type itself.

Technical Features

- **Generic type**: Implemented via the generic base class CGenUnidirectMergeBase (N sockets, 1 plug); the number of input sockets is fixed at compile time via the generic suffix (_7).
- **No states / algorithms**: Since ASR carries no data value, there is no change detection and no ECC.
- **Arbitrary input count**: The same implementation covers ASR_MERGE_2 through ASR_MERGE_7; for other input counts see ASR_MERGE_2, ASR_MERGE_3, ASR_MERGE_4, ASR_MERGE_5, ASR_MERGE_6.

State Overview

The function block has **no state machine**. Its behavior is purely combinational: every incoming event at any of the 7 sockets is forwarded to OUT immediately.

Application Scenarios

- **Merging several equivalent signal sources** (e.g. 7 redundant or alternative trigger paths) into a common ASR adapter output.
- **Simplifying networks** that would otherwise need 7 separate event connections to the same destination.

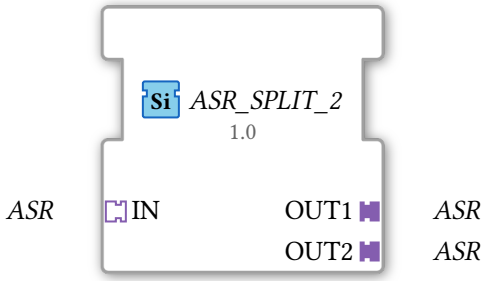
Comparison with Similar Components

- ASR_SPLIT_2: the reverse direction for 2 outputs (for ASRT, ASR_SPLIT_3 through ASR_SPLIT_9 are also available).
- **ASR_MERGE_2, ASR_MERGE_3, ASR_MERGE_4, ASR_MERGE_5, ASR_MERGE_6**: the same generic implementation with a different input count.
- AE_MERGE_7, ASRT_MERGE_7: the same merge logic for the other unidirectional event adapters.

Conclusion

ASR_MERGE_7 provides a generically implemented merge of 7 ASR event sources into a common adapter output, completing the GEN_ASR_MERGE family with the 7-input variant.

ASR_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_SPLIT_2** splits an incoming unidirectional ASR (Application Service Request) adapter into two identical output adapters. It is designed as a generic function block and allows an ASR signal to be forwarded to two separate consumers or subsequent function blocks without changing the logic.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Direction	Name	Description
`adapter::types::unidirectional::ASR`	Input (Socket)	**IN**	Incoming ASR signal
`adapter::types::unidirectional::ASR`	Output (Plug)	**OUT1**	First outgoing ASR path
`adapter::types::unidirectional::ASR`	Output (Plug)	**OUT2**	Second outgoing ASR path

Functionality

This module passes the ASR signal received via socket **IN** unchanged to both plugs **OUT1** and **OUT2**. No processing, filtering, or delay occurs – the signal is passively split into two paths. Each output provides the exact same information as the input.

Technical Features

- The function block is implemented as a **Generic Function Block (FB)**. The actual type instance is determined at runtime by configuring the generic class attribute (`eclipse4diac::core::GenericClassName = 'GEN_ASR_SPLIT'`).
- No event or data inputs/outputs are used – all communication takes place exclusively via the adapter interfaces.
- The outputs are independent of each other; an error or blockage at one output does not affect the other.

State Overview

The function block does not have its own state diagram, as it operates purely passively. Its behavior corresponds to a static branch.

Application Scenarios

- **Parallelization of Service Requests:** An ASR (e.g., a service request) should be sent to two independent components, e.g., to a logging component and simultaneously to an execution service.
- **Redundancy:** Distribution of a signal across two redundant paths to increase fault tolerance.
- **Adapter Bridge:** Used in systems where an ASR connection needs to be split across two adapter points without copying the logic.

Comparison with Similar Components

- ASR_MERGE_2: Merges two ASR inputs into one output (every SET/RESET event is forwarded unchanged, no priority arbitration). The **ASR_SPLIT_2** does the opposite.
- **SPLIT_1_to_N** (generic split components): A general splitter that operates with a configurable number of outputs. The **ASR_SPLIT_2** is a specialized version with two fixed outputs.

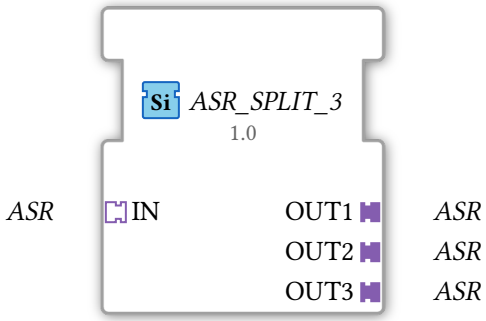
Conclusion

The **ASR_SPLIT_2** is a simple yet useful generic component for multiplying ASR signals. Thanks to its platform-independent adapters and purely passive operation, it is suitable for all scenarios where a service signal needs to be distributed across multiple paths. Integration requires no configuration and has no side effects on the transmitted data.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

ASR_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASR_SPLIT_3` is used to distribute an incoming ASR (Application-Specific Resource) adapter to three identical outgoing ASR adapters. It is designed as a generic block and enables simple signal distribution within an adapter-based communication structure.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Direction	Name	Description
Socket	Input	IN	Incoming ASR signal adapter.
Plug	Output	OUT1	First outgoing ASR signal adapter.
Plug	Output	OUT2	Second outgoing ASR signal adapter.
Plug	Output	OUT3	Third outgoing ASR signal adapter.

All adapters are of type `adapter::types::unidirectional::ASR`. ...

Functionality

This function block forwards all signals and events arriving via socket `IN` from the ASR adapter unchanged to the three plug channels `OUT1`, `OUT2`, and `OUT3`. No logical processing or buffering takes place – the distribution is passive and instantaneous. The function block acts as a simple splitter for ASR connections.

Technical Features

- The function block is generic (attribute `eclipse4diac::core::GenericClassName = 'GEN_ASR_SPLIT'`), allowing it to be reused for different instances of the ASR type.
- No Execution Control Chart (ECC) is required, as the forwarding occurs directly through the adapter coupling.
- All three outputs are separately addressable, allowing different target modules to be connected independently.

State Overview

Since the module does not process events and operates without a state, there is no state diagram. Its functionality is limited to passive signal pass-through.

Application Scenarios

- **Distributing control signals** to multiple parallel components in an automation system, e.g., to three drives or sensors.
- **Splitting ASR communication** across different subsystems that expect the same adapter interface.
- **Test and debugging setups** where a signal needs to be forwarded to multiple monitoring points simultaneously.

Comparison with Similar Modules

- `ASR_SPLIT_2` splits an ASR signal across two outputs; `ASR_SPLIT_3` extends this functionality to three outputs.
- `ASR_MERGE` (if present) combines multiple ASR signals, while `ASR_SPLIT_3` separates them.
- Unlike generic data splitters, this component operates directly at the adapter level and requires no additional data type conversions.

Conclusion

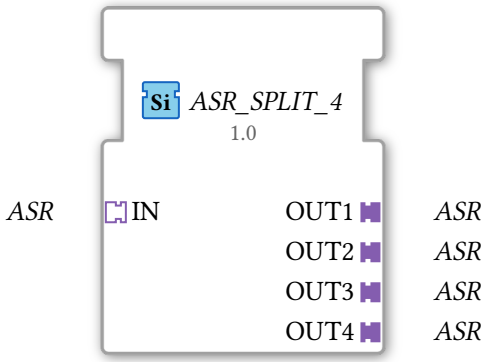
The `ASR_SPLIT_3` is a simple yet effective component for duplicating an ASR signal to three identical outputs. It is particularly well-suited for modular structures where a central signal is needed multiple times. Its generic design increases reusability and reduces configuration effort.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ASR_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_SPLIT_4** is a generic block that distributes an incoming unidirectional ASR adapter (actuator-control adapter) signal to four separate, identical ASR outputs. It enables the simultaneous control of up to four independent actuators or subsystems with the same signal without compromising signal integrity. The block is designed for use in distributed automation systems according to IEC 61499.

Interface Structure

Adapter Inputs (Sockets)

Name	Type	Description
IN	`adapter::types::unidirectional::ASR`	Input adapter providing the ASR connection to be distributed.

Adapter Outputs (Plugs)

Name	Type	Description
OUT1	`adapter::types::unidirectional::ASR`	First output adapter (identical copy of the input).
OUT2	`adapter::types::unidirectional::ASR`	Second output adapter.
OUT3	`adapter::types::unidirectional::ASR`	Third output adapter.
OUT4	`adapter::types::unidirectional::ASR`	Fourth output adapter.

Note: The specific structure of the ASR adapter (event/data elements) is not defined within the function block itself, but is determined by the associated adapter type `adapter::types::unidirectional::ASR`. Typically, this includes control and feedback signals for drives.

Functionality

The ASR_SPLIT_4 function block operates as a pure signal distributor: All ASR connection elements (events and data) received via the `IN` socket are replicated **1:1** to all four output plugs (OUT1–OUT4). No signal amplification, filtering, or logical processing takes place. As soon as the input adapter establishes an active connection, it is immediately passed through to all outputs. The function block does not require its own state logic – it operates entirely combinationally.

Technical Features

- **Generic Function Block:** The function block is declared as a generic type (`GEN_ASR_SPLIT`), which means that the interface definition of the ASR adapter is resolved at compile time. It can be used with various versions of the ASR adapter, as long as they conform to the unidirectional protocol.
- **Unidirectional Signal Direction:** The adapters are designed as unidirectional, meaning data flows only from the input to the outputs (no return channel from output to input).
- **No Latency Buffering:** All outputs are supplied with the same signal simultaneously; there is no staggered timing or buffering.

State Overview

The function block has **no explicit internal states**. The outputs always follow the input signals directly. There is no state machine. The behavior can be described as a single "switching" operation.

Application Scenarios

- **Control of Multiple Parallel Drives:** A central drive command transmitter (e.g., an AGV control block) distributes the speed or direction signal to four independent motor controllers.
- **Redundant Signal Distribution:** In safety-critical systems, the same command can be sent to multiple redundant actuators to ensure fail-safe operation.
- **Test and Simulation Environments:** A simulated ASR signal is simultaneously applied to multiple test instances.

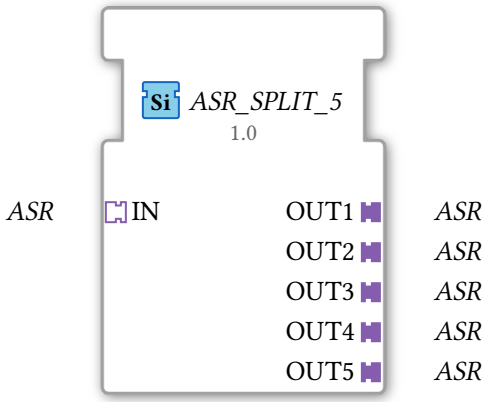
Comparison with Similar Function Blocks

- **ASR_SPLIT_2 / ASR_SPLIT_N:** These function blocks split the ASR input into two or any number of outputs, respectively. The `ASR_SPLIT_4` is a special, four-channel variant.
- **Signal Multiplexer:** Unlike a true multiplexer, which selects from multiple sources, the split function block distributes the *same* source to multiple sinks.
- **Adapter Coupler:** Some libraries offer coupling blocks for adapter routing; the Split block extends this functionality to include multiplication.

Conclusion

The **ASR_SPLIT_4** is a simple yet useful function block that enables the modular and reusable coupling of ASR adapter connections in IEC 61499 systems. Its generic implementation allows it to be used in various contexts where an input signal needs to be distributed to multiple identical output interfaces. Thanks to its adaptive, stateless design, it integrates seamlessly into event-driven automation architectures.

ASR_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_SPLIT_5** is used to split a single unidirectional ASR adapter (Agricultural Standardized Interface) into five identical ASR outputs. It is designed as a generic module, thus enabling the simple distribution of an ASR signal to multiple downstream components or controllers.

Interface Structure

The module does not have any traditional event or data inputs/outputs. All communication takes place via adapter interfaces of type `adapter::types::unidirectional::ASR`.

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Name	Type	Direction
`IN`	`adapter::types::unidirectional::ASR`	Socket (Input)
`OUT1`	`adapter::types::unidirectional::ASR`	Plug (Output)
`OUT2`	`adapter::types::unidirectional::ASR`	Plug (Output)
`OUT3`	`adapter::types::unidirectional::ASR`	Plug (Output)
`OUT4`	`adapter::types::unidirectional::ASR`	Plug (Output)
`OUT5`	`adapter::types::unidirectional::ASR`	Plug (Output)

Functionality

This component is a simple multiplexer at the adapter level. The ASR signal present at socket `IN` is passed through unchanged and without delay to all five output plugs (`OUT1` … `OUT5`). Thus, each connected receiver receives the identical signal from the input adapter. No active signal processing or conditioning takes place.

Technical Features

- **Generic Type**: The component is implemented as a generic function block (`GEN_ASR_SPLIT`) and can be automatically adapted to the specific data types of the ASR adapter used.
- **Unidirectional Communication**: All adapters are unidirectional (`adapter::types::unidirectional::ASR`), meaning data flows only from the input to the outputs; feedback is not provided.
- **No State Logic**: Since no events or algorithms are present, the module does not have an ECC state diagram and operates exclusively in a data flow-oriented manner.

State Overview

The functional block has no internal state machine. It behaves like a passive signal distributor and does not require state control.

Application Scenarios

- **Signal Distribution**: An ASR signal (e.g., sensor data, control commands) must be simultaneously transmitted to multiple consumers (e.g., multiple actuators or higher-level controllers).
- **Test and Simulation Environments**: Parallel injection of a test signal into multiple independent system paths.
- **Redundancy**: Supplying multiple identical control units with the same input information.

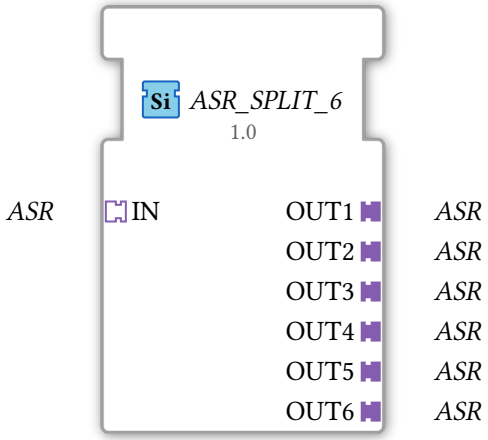
Comparison with Similar Function Blocks

- **ASR_SPLIT_2 / ASR_SPLIT_3 / ASR_SPLIT_4**: These function blocks split an ASR signal into two, three, or four outputs, respectively – the appropriate number can be selected as needed.
- **ASR_MERGE**: Unlike a splitter, a MERGE combines multiple ASR inputs into a single output.

Conclusion

The **ASR_SPLIT_5** is a simple yet useful routing function block for multiplying a unidirectional ASR adapter. Thanks to its generic implementation and pure adapter interface, it integrates seamlessly into IEC 61499 applications and facilitates structured signal distribution in complex automation systems.

ASR_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASR_SPLIT_6` distributes an incoming ASR signal (Generic Adapter Type) to six identical ASR outputs. It serves as a pure signal splitter and is provided as a generic function block in the 4diac IDE.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Direction	Type	Description
IN	Socket	adapter::types::unidirectional::ASR	Input adapter – the ASR signal to be distributed
OUT1 – OUT6	Plug	adapter::types::unidirectional::ASR	Six output adapters – identical copies of the input signal

Functionality

This component does not perform any data processing or state logic. It forwards the adapter (unidirectional ASR) connected to socket IN directly and without delay to all six plugs (OUT1 to OUT6). Thus, the same signal is present at each output as at the input. The distribution occurs purely at the adapter level, without the involvement of events or data inputs/outputs.

Technical Features

- **Generic Function Block**: The function block uses the generic type definition GEN_ASR_SPLIT (attribute eclipse4diac::core::GenericClassName). This allows for flexible reuse with different ASR adapter types.
- **Unidirectional Adapter**: Both the input and outputs are of type adapter::types::unidirectional::ASR, meaning the data flows only in one direction (from the socket to the plugs).
- **No State Logic**: The component has no ECC (Execution Control Chart) and no internal states – it is a pure connection without any temporal behavior.
- **Scalability**: Due to the generic concept, similar splitters with any number of outputs (e.g., 2, 4, 8) can be created.

State Overview

There is no state diagram (ECC) because the component does not execute any sequential logic. Signal distribution is static and continuous.

Application Scenarios

- **Distribution of a sensor signal** to multiple control units or subsystems.
- **Provision of the same ASR communication channel** for parallel processing branches.
- **Test and simulation environments** where an input signal is required multiple times.

Comparison with Similar Components

Component	Outputs	Special Features
ASR_SPLIT_2	2	Dual Split
ASR_SPLIT_4	4	Quad Split
ASR_SPLIT_6	**6**	**Six-Way Split (this module)**
Generic Split (e.g., via template)	Variable	Requires individual parameterization

The ASR_SPLIT_6 offers a fixed number of six outputs and is therefore specifically designed for applications that require exactly this number.

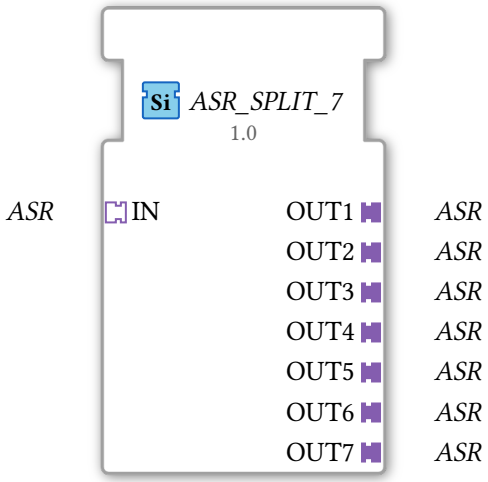
Conclusion

The `ASR_SPLIT_6` function block is a simple yet useful function block for signal duplication at the adapter level. Its generic nature and clear, uneventful operation make it ideal for modular control architectures that require multiple uses of an ASR signal.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

ASR_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_SPLIT_7** is a generic function block that distributes a single ASR adapter input to seven ASR adapter outputs. It acts as a 1:7 splitter for ASR data and enables the parallel transmission of a signal to multiple downstream components.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Direction	Name	Type	Description
Input (Socket)	**IN**	`adapter::types::unidirectional::ASR`	Source adapter whose data is distributed to all outputs.
Output (Plug)	**OUT1**	`adapter::types::unidirectional::ASR`	First output, receives the duplicated data from `IN`.
Output (Plug)	**OUT2**	`adapter::types::unidirectional::ASR`	Second output.

Output (Plug)	**OUT3**	`adapter::types::unidirectional::ASR`	Third output.
Output (Plug)	**OUT4**	`adapter::types::unidirectional::ASR`	Fourth output.
Output (Plug)	**OUT5**	`adapter::types::unidirectional::ASR`	Fifth output.
Output (Plug)	**OUT6**	`adapter::types::unidirectional::ASR`	Sixth output.
Output (Plug)	**OUT7**	`adapter::types::unidirectional::ASR`	Seventh output.

Functionality

The module receives an ASR signal via socket `IN`. This signal is internally duplicated to all seven plugs (`OUT1` ... `OUT7`) and forwarded without modification. No data manipulation or filtering takes place – the incoming signal is copied one-to-one to each output.

Technical Features

- **Generic Type**: The function block is declared as a generic block with the class name `GEN_ASR_SPLIT`. This allows it to be used in various contexts where an ASR splitter is required.
- **Pure Adapter Interface**: Only adapters (no events or data inputs/outputs) are used. Communication takes place via the unidirectional ASR adapter protocol.
- **No Internal State**: The block has no state management; signal distribution is continuous and instantaneous.

State Overview

The block has no explicit states. Its functionality is limited to the static duplication of the input signal.

Application Scenarios

- **Signal Distribution**: An ASR signal (e.g., a control command or a configuration message) is to be sent to multiple independent receivers.
- **Parallel Processing**: In an automated control system, information must reach multiple subsystems simultaneously, each designed for the ASR protocol.
- **Test Environments**: A test adapter is connected to `IN`, and the outputs are connected to various monitoring or logging modules.

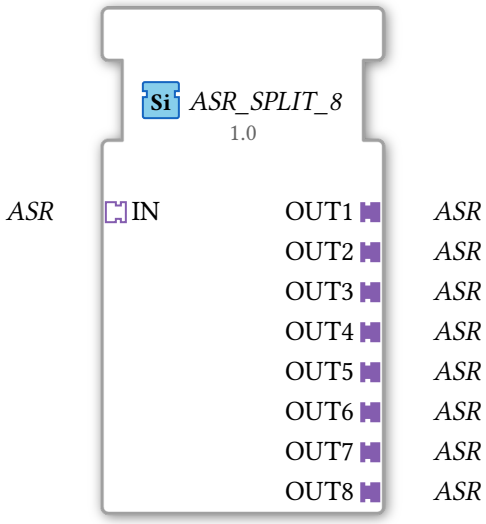
Comparison with Similar Modules

Other splitter modules exist, such as `ASR_SPLIT_2`, `ASR_SPLIT_4`, and `ASR_SPLIT_n`, which distribute the input signal to 2, 4, and `n` outputs, respectively. `ASR_SPLIT_7` represents a specific variant with seven outputs. All of them share the common feature of duplicating the incoming ASR signal without modification.

Conclusion

The `ASR_SPLIT_7` is a simple yet useful component for distributing an ASR signal across seven parallel paths. Its generic definition and the absence of events or states make it flexible and robust in operation.

ASR_SPLIT_8



(Insert image of function block here)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASR_SPLIT_8` is used to distribute a single incoming **ASR** adapter signal unchanged to eight identical outputs. It is defined as a generic function block and is particularly suitable for applications where a sensor or control signal is required multiple times – for example, in agricultural automation technology.

Interface Structure

Event Inputs

No event inputs are available. The function block operates purely on an adapter basis without event-driven execution.

Event Outputs

No event outputs are available.

Data Inputs

No data inputs are available. All signal transmissions take place via the **adapter** interfaces.

Data Outputs

No data outputs available.

Adapters

Direction	Name	Type	Description
Socket	IN	`adapter::types::unidirectional::ASR`	Incoming ASR signal distributed to all outputs.
Plug	OUT1	`adapter::types::unidirectional::ASR`	First output with the signal from IN.
Plug	OUT2	`adapter::types::unidirectional::ASR`	Second output with the signal from IN.
Plug	OUT3	`adapter::types::unidirectional::ASR`	Third output with the signal from IN.
Plug	OUT4	`adapter::types::unidirectional::ASR`	Fourth output with the signal from IN.
Plug	OUT5	`adapter::types::unidirectional::ASR`	Fifth output with the signal from IN.
Plug	OUT6	`adapter::types::unidirectional::ASR`	Sixth output with the signal from IN.

Plug	OUT7	`adapter::types::unidirectional::ASR`	Seventh output with the signal from IN.
Plug	OUT8	`adapter::types::unidirectional::ASR`	Eighth output with the signal from IN.

Functionality

The `ASR_SPLIT_8` block represents a **1-to-8 signal distribution** for the ASR adapter type. The ASR signal present at socket `IN` is forwarded to all eight plug outputs (`OUT1`...`OUT8`) without delay or state change. No signal conditioning, filtering, or logical processing takes place – the block operates purely passively and without any inherent behavior.

Since the adapter type `ASR` is defined as unidirectional, data flows only from the socket to the plugs. Changes to the input signal immediately affect all outputs.

Technical Features

- **Generic Type Parameterization**: The function block (FB) has the attributes `eclipse4diac::core::GenericClassName` (value `'GEN_ASR_SPLIT'`) and `eclipse4diac::core::TypeHash`, which enable the use of a generic ASR data type. This allows the block to be reused for various ASR subtypes.
- **No Event Control**: The block contains no event inputs or outputs; signal transmission occurs exclusively via the adapter interfaces. This simplifies integration into static data flows.
- **No Internal Behavior**: There is no state machine or algorithm – functionality is limited to the pure wiring logic of the 4diac IDE.

State Overview

The `ASR_SPLIT_8` has **no internal states**. The output is a direct function of the input signal at any given time. The block behaves like passive wiring (sterile splitter) without any memory or delay effects.

Application Scenarios

- **Distribution of a sensor signal** (e.g., speed, pressure, or position) to multiple parallel control loops or display units.
- **Dividing a control command** to multiple actuators in an agricultural machine, such as the simultaneous control of several valves or drives.
- **Monitoring and Logging**: A signal can be routed to a controller via one output and to a diagnostic system via a second output.

Comparison with Similar Blocks

Block	Outputs	Special Features
`ASR_SPLIT_8`	8	Provides the maximum possible number of ASR copies in one block.
`ASR_SPLIT_4`	4	More compact alternative for smaller distribution systems.
`ASR_SPLIT_2`	2	Minimal splitter for point-to-point transmission.

The **8-way version** is the most comprehensive and allows the connection of many devices without additional cascading. Compared to cascading multiple 2-way or 4-way splitters, `ASR_SPLIT_8` reduces the number of required blocks and simplifies engineering.

Conclusion

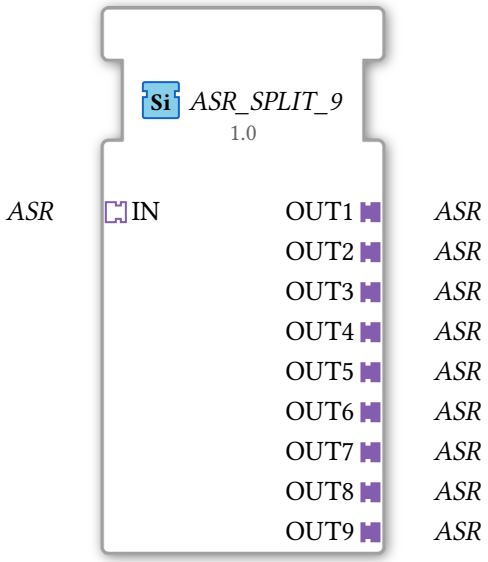
The `ASR_SPLIT_8` is a simple yet efficient component for distributing an ASR signal across eight independent paths. Its generic design and the absence of states and events make it robust, resource-efficient, and ideally suited for unidirectional data transmission in agricultural and industrial automation systems. It complements the ASR splitter family and addresses the need for a high number of parallel outputs.

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ASR_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ASR_SPLIT_9** is used to split a single incoming ASR adapter (Application Specific Resource) into nine separate ASR outputs. It is implemented as a generic function block (Generic FB) and allows the distribution of an ASR signal to up to nine independent receivers.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

No separate data inputs. All data exchange takes place via the ASR adapter **IN**.

Data Outputs

No separate data outputs. The data is output via the ASR adapters **OUT1** to **OUT9**.

Adapter

Socket (Input):

Name	Type	Direction
IN	`adapter::types::unidirectional::ASR`	Socket

Plugs (Outputs):

Name	Type	Direction
OUT1	`adapter::types::unidirectional::ASR`	Plug
OUT2	`adapter::types::unidirectional::ASR`	Plug
OUT3	`adapter::types::unidirectional::ASR`	Plug
OUT4	`adapter::types::unidirectional::ASR`	Plug
OUT5	`adapter::types::unidirectional::ASR`	Plug

OUT6	`adapter::types::unidirectional::ASR`	Plug
OUT7	`adapter::types::unidirectional::ASR`	Plug
OUT8	`adapter::types::unidirectional::ASR`	Plug
OUT9	`adapter::types::unidirectional::ASR`	Plug

Functionality

The module receives an ASR signal via the socket adapter **IN**. This signal is internally routed to all nine output adapters (**OUT1** to **OUT9**). Thus, the identical ASR signal is present at each output. No delay or buffering occurs. The module is purely passive and does not perform any data transformation.

Technical Features

- **Generic Function Block:** This type is designed as a generic FB (GenericClassName `'GEN_ASR_SPLIT'`), allowing it to be parameterized with different ASR types depending on the project context.
- **Unidirectional Adapters:** All adapters used are of type `unidirectional::ASR`, ensuring a clear data flow direction from the input to the outputs.
- **No Events:** The function block has no event inputs or outputs. Data is transmitted solely via the adapter interfaces.
- **Platform-Neutral:** By using the IEC 61499-2 standard, the function block can be used in various 4diac IDE projects.

State Overview

The function block does not have an internal state machine (ECC). It operates statically and passes the incoming signal to all outputs without any temporal or logical dependencies.

Application Scenarios

- **Distribution of an ASR signal** to multiple parallel-operating resources or functional units.
- **Monitoring or parallel control** when multiple components require the same ASR data (e.g., one sensor value to multiple evaluation modules).
- **Simple bus-like structures** in automated control systems, without complex multicast mechanisms.

Comparison with Similar Modules

- **ASR_SPLIT_4 / ASR_SPLIT_2:** Modules with a smaller number of outputs (4 or 2, respectively). `ASR_SPLIT_9` offers the maximum distribution across nine channels.
- **ASR_MERGE:** Combines multiple ASR signals into one, while `ASR_SPLIT_9` performs the exact opposite function.
- **Data Distributor:** Data type-oriented function blocks (e.g., `DATA_SPLIT`) tend to use simpler data types. `ASR_SPLIT_9` is specifically optimized for complex adapter structures.

Conclusion

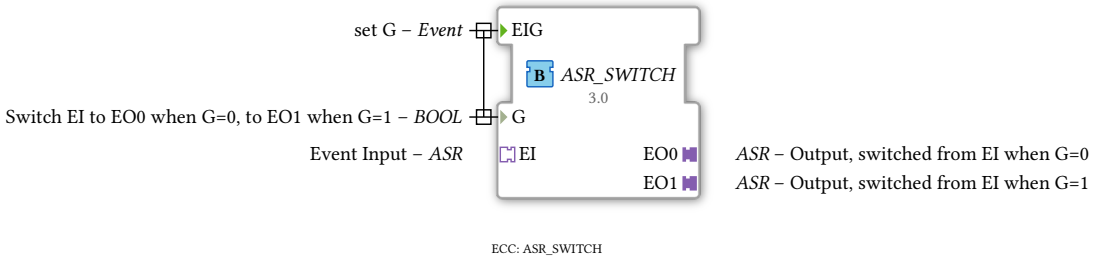
The **ASR_SPLIT_9** function block is a compact and efficient solution for distributing an ASR signal to nine identical outputs. Its generic nature and clear, event-free adapter coupling make it particularly suitable for modular, reusable automation systems based on IEC 61499.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ASR_SWITCH





• – ○ • – ○ • – ○ •

Introduction

The **ASR_SWITCH** function block acts as a demultiplexer for **ASR events** (SET/RESET). It forwards events received via the adapter socket `EI` to either the adapter plug `EO0` or `EO1`, depending on the Boolean value of the input `G`. This allows switching between two different signal paths in an automation application.

Interface Structure

Event Inputs

Event	Data Link	Description
`EIG`	`G`	Sets the switching value `G` to the Boolean value present at input `G`.

Event Outputs

This function block does not have any independent event outputs. Output is exclusively via the **adapter plugs**.

Data Inputs

Variable	Type	Description
`G`	`BOOL`	Switch position: `FALSE` ⟶ Events are forwarded to `EO0`, `TRUE` ⟶ to `EO1`.

Data Outputs

None.

Adapters

Adapter	Direction	Type	Description
EI	Socket	adapter::types::unidirectional::ASR	Receives incoming ASR events (SET/RESET).
EO0	Plug	adapter::types::unidirectional::ASR	Output for the case G = FALSE.
EO1	Plug	adapter::types::unidirectional::ASR	Output for the case G = TRUE.

Functionality

The **ASR_SWITCH** is activated by an incoming event from the adapter socket EI or by the event EIG. The processing takes place in **Event-driven Control (ECC)**:

1. **Switching G:**

When the event EIG occurs, the value of G is adopted. The function block remains in the start state START and waits for the next events.

1. **Forwarding EI.SET:**

- EI.SET and G = FALSE: Transition to the state G0_SET, where EO0.SET is output.
- EI.SET and G = TRUE: Transition to state G1_SET, where EO1.SET is output.

1. **Forwarding from EI.RESET:**

- EI.RESET and G = FALSE: Transition to state G0_RESET, where EO0.RESET is output.
- EI.RESET and G = TRUE: Transition to state G1_RESET, where EO1.RESET is output.

1. **Return:**

After the action is output, the ECC immediately (condition = 1) returns to the start state START to process the next event.

Technical Features

- The function block works exclusively with the **unidirectional adapter type ASR**, meaning it can transport both SET and RESET events via the same adapter.
- The switchover is **event-triggered** via EIG (non-cyclically).
- No data outputs are required – all signaling is handled via the adapter plugs.
- Due to the immediate reset to the start state, the module is ready to restart **deterministically and quickly**.

State Overview

The following table summarizes the ECC states and the actions performed:

State	Description	Action(s)
START	Waiting for events	–
G0_SET	EI.SET at G = FALSE	Output EO0.SET
G1_SET	EI.SET at G = TRUE	Output EO1.SET
G0_RESET	EI.RESET at G = FALSE	Output EO0.RESET
G1_RESET	EI.RESET at G = TRUE	Output EO1.RESET

All states (except START) return to START with an unconditional transition (1).

Application Scenarios

- **Control of two parallel process paths:** Depending on the position of a switch, SET/RESET signals are distributed to different parts of the system.
- **Operating mode switching:** For example, between automatic and manual operation – a logical variable G determines which events affect which logic.

- **Test environments:** A test frame can send the same events to different test objects by switching `G`.

Comparison with Similar Function Blocks

- **`SELECT` Function Block:** A classic multiplexer/demultiplexer usually works with data, not events. The **ASR_SWITCH** specializes in ASR event forwarding.
- **`F_TRIG` / `R_TRIG` / `SR`:** These function blocks set or reset a memory function, but do not forward events. The **ASR_SWITCH** does not create states itself, but only channels existing events.
- **Adapter-Based Solutions:** In the 4diac IDE, the use of adapters is typical for modular communication – the **ASR_SWITCH** is a prime example of the flexible coupling of function blocks.

**

Conclusion

The **ASR_SWITCH** is a simple yet effective function block for demultiplexing SET/RESET events. Its adapter interface allows for seamless integration into event-based automation solutions. Its clear state logic and minimal interface make it a reliable component for switching between two signal paths.

🌐 Related topic subpages on ms-muc-docs.de

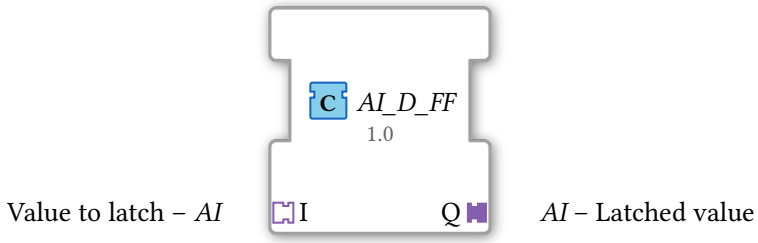
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

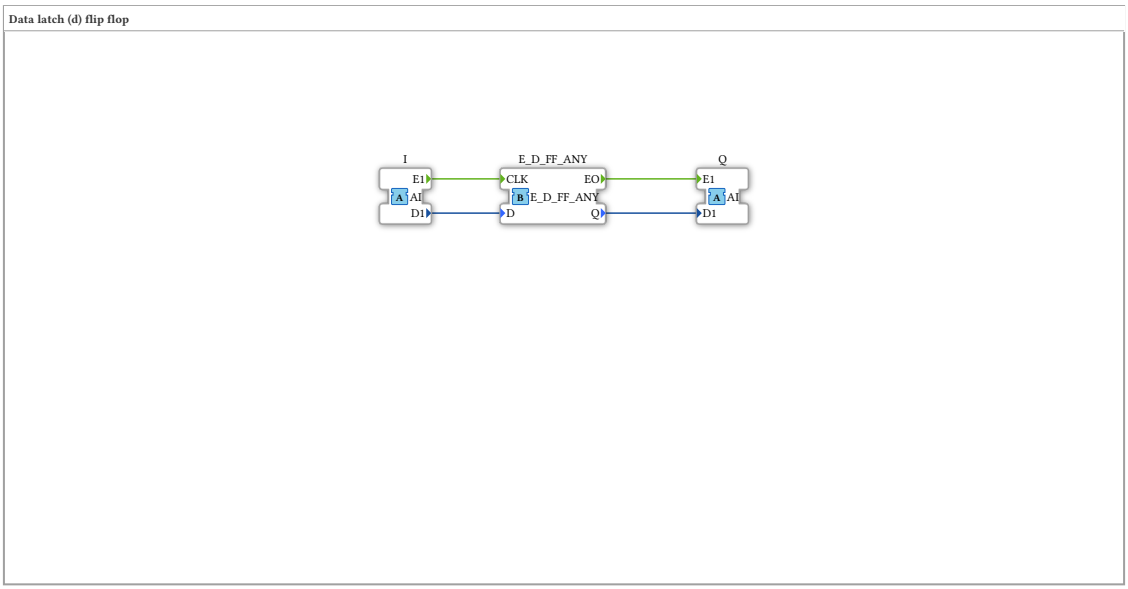
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AI_D_FF





• – ○ • – ○ • – ○ •

Introduction

The function block **AI_D_FF** implements a data latch (D flip-flop) for the temporary storage of a data value. It serves as an adapter-encapsulated implementation of an edge-triggered D flip-flop according to IEC 61499, which, upon an event at the input, takes the incoming data value and makes it available at the output.

Interface Structure

Event Inputs

No direct event inputs.

Event Outputs

No direct event outputs.

Data Inputs

No direct data inputs.

Data Outputs

No direct data outputs.

Adapter

Name	Direction	Type	Description
I	Socket	`adapter::types::unidirectional::AI`	Returns the input event (E1) and the data value to be latched (D1).
Q	Plug	`adapter::types::unidirectional::AI`	After the latch is closed, outputs the result event (E1) and the stored data value (D1).

The `AI` adapter is unidirectional, meaning that data and event flows only in one direction. Socket **I** serves as the input, and Plug **Q** as the output.

Functionality

The internal component is a `E_D_FF_ANY` function block (IEC 61499 D flip-flop). The wiring connects:

- `I.E1` ⟶ `E_D_FF_ANY.CLK` (clock edge)
- `I.D1` ⟶ `E_D_FF_ANY.D` (data input)
- `E_D_FF_ANY.EO` ⟶ `Q.E1` (output event)
- `E_D_FF_ANY.Q` ⟶ `Q.D1` (latching data value)

Process:

An event at `I.E1` triggers a rising edge at the clock input of the internal flip-flop. The current value of `I.D1` is transferred and is available at `Q.D1` after processing is complete. Simultaneously, an event is output to `Q.E1`, signaling the validity of the new data value.

Technical Features

- **Adapter Encapsulation:** The function block (FB) has no direct inputs/outputs, but only adapter interfaces. This facilitates reuse in modular architectures where multiple signals are bundled and transmitted.
- **Unidirectional Data Flow:** The adapters are designed as pure sources (plugs) or sinks (sockets), which clearly defines the signal direction.
- **Edge-Triggered:** The transfer occurs only upon an event (rising edge), not upon level changes. Therefore, the function block is suitable for clocked systems.
-

State Overview

The internal state of the flip-flop is determined by the stored value `Q`. Two stable states exist:

Clock (CLK)	D (Input)	Q (Output) before	Q (Output) after
No event	-	Q_old	Q_old

Event (rising edge)	0	X	0
Event (rising edge)	1	X	1

For each event at `I.E1`, the current D value is adopted, regardless of the previous Q value.

Application Scenarios

- **Analog Value Sampling:** A sensor (e.g., temperature, pressure) periodically delivers a measured value via the `I` adapter. The function block stores this value upon a trigger event and passes it on to the subsequent logic.
- **Cable Storage in Data Paths:** In conveyor belt or pipeline structures, `AI_D_FF` can be used as a data hold to buffer values for one clock cycle.
- **Debouncing / Edge Processing:** Combined with a clock generator, the function block can be used to process incoming signals only at specific times.

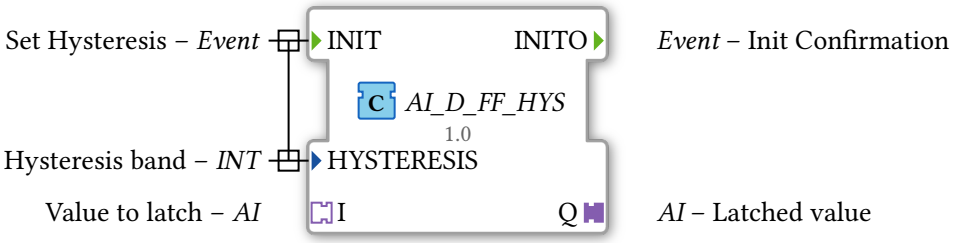
Comparison with Similar Function Blocks

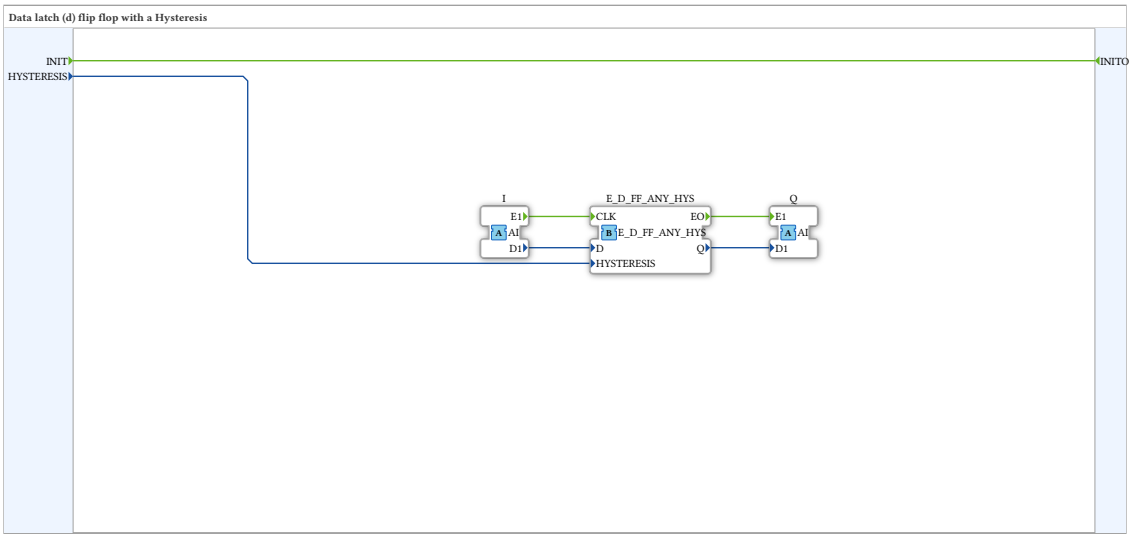
- **E_D_FF** (without adapter): Offers direct event and data inputs/outputs but is less abstract. `AI_D_FF`, on the other hand, encapsulates the signals in adapters, which simplifies composition in higher hierarchies.
- **SR Flip-Flop (S/R dominant):** Stores state via set/reset, while `AI_D_FF` only takes over on clock edges and has no reset function.
- **Latch (level-controlled):** Unlike a latch, `AI_D_FF` only reacts to edges, making it better suited for synchronized systems.

Conclusion

The **AI_D_FF** is a compact, adapter-based D flip-flop device that stores and outputs a data value on an edge-triggered basis. By using unidirectional adapters, it integrates seamlessly into modern, modular automation concepts and enables a clean separation of data and event signals. Its application is particularly suitable for clocked sampling and buffering tasks in IEC-61499 systems.

AI_D_FF_HYS





• – ○ • – ○ • – ○ •

Introduction

The function block **AI_D_FF_HYS** implements a clock-controlled D flip-flop (data latch) with configurable hysteresis. It serves to stably lock a binary or analog input signal and filter out disturbances in the signal waveform. The hysteresis prevents constant switching due to minor signal changes (bouncing, noise), thus ensuring a clean, latency-free output state.

Interface Structure

Event Inputs

Event	Type	With Variables	Description
INIT	EInit	HYSTERESIS	Initialization and Hysteresis Band Setting

Event Outputs

Event	Type	Description
INITO	EInit	Initialization Confirmation

Data Inputs

Name	Type	Description
HYSTERESIS	INT	Hysteresis Bandwidth (e.g., in digital steps)

Data Outputs

This function block does not have direct data outputs. The locked value is provided via the **Adapter Plug Q**.

Adapter

Name	Type	Direction	Description
I	AI	Socket	Input adapter: Clock signal (**E1**) and data (**D1**)
Q	AI	Plug	Output adapter: Latched value (**D1**) and associated event (**E1**)

The adapter type used, `adapter::types::unidirectional::AI`, is a unidirectional data and event channel. The clock (E1) and data (D1) values are received via socket **I** the latched value (D1) is output via plug **Q** along with an acknowledgment event (E1).

Functionality

1. **Initialization**

The module is activated by the **INIT** event. The passed value **HYSTERESIS** is stored as the hysteresis band. After successful initialization, **INITO** is triggered.

1. **Data Latching**

After initialization, the module waits for a clock signal at adapter **I** (event **I.E1**). With each clock event, the current data value **I.D1** is read.

The internal function block `E_D_FF_ANY_HYS` compares the incoming value with the last latched value, taking the hysteresis into account:

- If the new value exceeds the old value by more than half the hysteresis, the new value is adopted (set).
- If it falls short of the old value by more than half the hysteresis, the new value is adopted (reset).
- For smaller differences, the output remains unchanged.

The latched value determined in this way is provided at adapter **Q**: **Q.D1** contains the digital value, and event **Q.E1** signals the update.

Technical Features

- **Adapter-Based Communication**

All data and event exchange occurs via adapters. This enables loose coupling between the flip-flop and the connected components and facilitates reuse in different contexts.

- **Hysteresis for Noise Suppression**

Hysteresis prevents flip-flop behavior in the presence of noise or bouncing signals. The bandwidth is passed as an integer (INT) and can be reset at runtime via **INIT**.

- **No Direct Data Outputs**

The latched value is only accessible via the **Q** adapter. This enforces structured signal routing within the overall system.

- **Embedded Function Block**

The actual flip-flop with hysteresis is implemented by the internal function block `E_D_FF_ANY_HYS`, which encapsulates complexity and simplifies maintenance.

State Overview

The function block (FB) does not have explicit states in the sense of a finite automaton. After the **INIT** operation, it is in an operating state where it waits for clock events. A subsequent **INIT** call resets the hysteresis and initializes the internal logic.

State	Description
Initialized	Ready for processing after a successful **INIT**
Waiting for Clock	Expecting an **I.E1** event at the input adapter
Data Processing	Upon **I.E1**: Hysteresis comparison and, if necessary, acceptance

Application Scenarios

- **Debouncing of Switches and Sensors**

A mechanical contact produces bouncing signals. Hysteresis suppresses short-term changes and only passes on stable states.

- **Noise Reduction for Analog Signals**

An analog-to-digital converter (ADC) delivers noisy values. Hysteresis prevents constant switching between adjacent quantization levels.

- **State Memory in Controllers**

This component can be used as a simple D flip-flop to clock a digital value and hold it until the next clock signal.

- **Interface Between Asynchronous and Synchronous Systems**

This adapter allows the connection of components with different event clocks by using the flip-flop as a buffer.

Comparison with Similar Components

Compo-nent	Hys-teresis	Clock Input	Output Type	Special Feature					
AI_D_FF_HYS	Yes	Yes	Adapter (AI)	Flexible Adapter Cou-pling	... Stan-dard D-FF (61499)	No	Yes	Direct Data Outputs	No Noise Sup-pression
SR Flip-Flop	No	No	Direct	Set/Re-set, No Clock					
Schmitt Trigger	Yes	No	Analog Thresh-old	Thresh-old Only,					

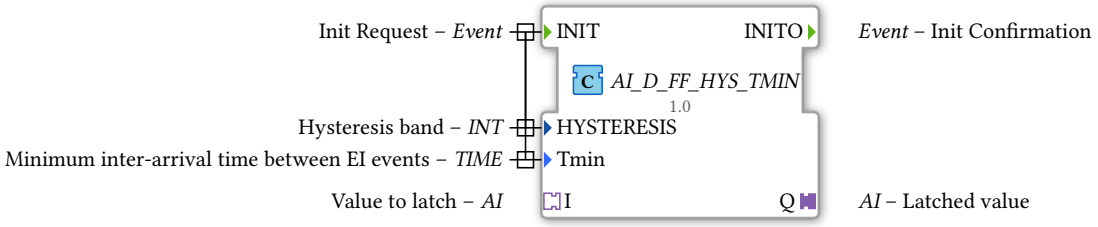
				No Memory					

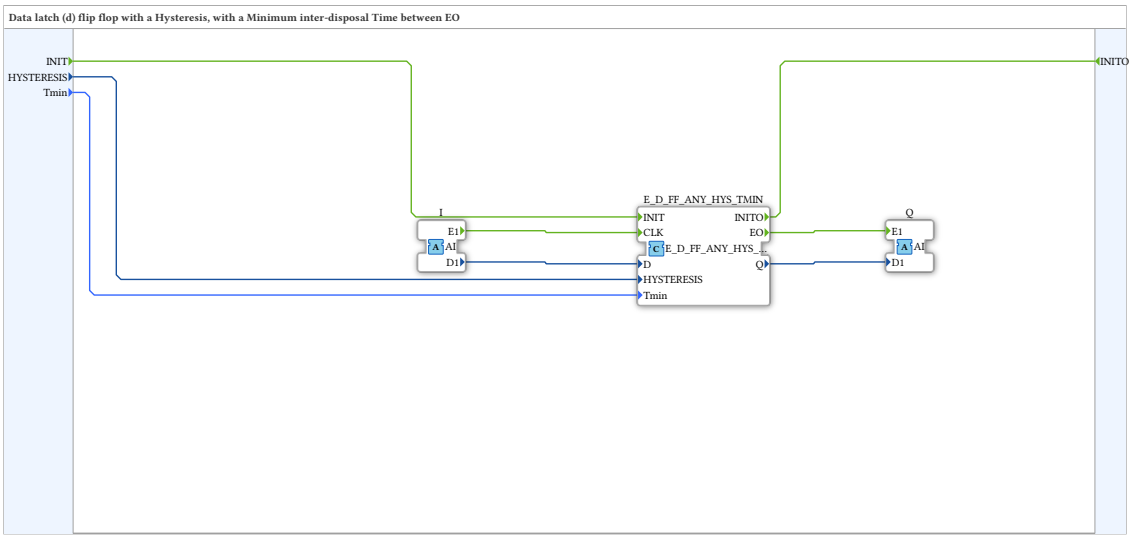
The AI_D_FF_HYS combines the features of a clocked D flip-flop with adjustable hysteresis and offers platform-independent integration into IEC 61499 systems via its adapter interface.

Conclusion

The **AI_D_FF_HYS** is a robust and flexible functional block for clocked signal locking with hysteresis. It is particularly suitable for applications where noise needs to be filtered out of the input signal without sacrificing the functionality of a simple clock flip-flop. The adapter-based interface promotes modular, reusable designs and makes the component a valuable element in automation technology and signal processing.

AI_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AI_D_FF_HYS_TMIN** implements a data latch (D flip-flop) with a hysteresis band and a minimum time between successive event outputs. It serves as a robust signal processing component that suppresses noise and fast edge transitions and only passes on stable state changes.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Initialization Request (sets hysteresis band and Tmin)

Event Outputs

Event	Type	Comment

INITO	EInit	Initialization Confirmation

Data Inputs

Variable	Type	Comment
HYSTERESIS	INT	Hysteresis Band (value in the same unit as the input signal)
Tmin	TIME	Minimum time between two event outputs (E1 on the Q adapter)

Data Outputs

This function block does not have its own data outputs. The latched value is provided via the output adapter **Q**.

Adapters

Adapter	Type	Comment
I	adapter::types::unidirectional::AI	Input adapter: provides the signal to be processed (via event E1 and data D1)
Q	adapter::types::unidirectional::AI	Output adapter: outputs the latched value (via event E1 and data D1)

Functionality

The function block internally uses the subroutine **E_D_FF_ANY_HYS_TMIN** from the *logiBUS::signal-processing::hysteresis* library. Overview of its functionality:

1. **Initialization**

The parameters `HYSTERESIS` and `Tmin` are received via the **INIT** event input. After successful initialization, **INITO** is sent.

1. **Signal Processing**

- A new value is received via the **I** adapter (event **E1** with the corresponding data value **D1**).
- The function block compares the new value with the currently stored (latched) value, taking the hysteresis band into account:
- A change is only accepted if the difference between the new value and the old value is greater than `HYSTERESIS`.
- Additionally, the time elapsed since the last event output must be at least `Tmin`. Only when both conditions are met is the new value accepted and output via the **Q** adapter (event **E1** and data **D1**).

1. **Output**

The currently valid latched value is provided via the **Q** adapter. An event is sent to **Q.E1** for every valid state change.

Technical Features

- **Hysteresis** – Suppresses noise and rapid fluctuations by only accepting a change if it exceeds the hysteresis band.
- **Minimum Time Between Events (Tmin)** – Prevents overly frequent outputs (e.g., due to high-frequency signals or bouncing). Events are only released after a period of `Tmin` has elapsed since the last event.
- **Adapter-Based Interface** – Input and output are handled via unidirectional adapters, allowing for flexible integration with other components.

State Overview

The component does not have explicitly named states. The internal flip-flop operates as a bistable element with two stable states (high/low or the last valid value). State transitions are controlled by the hysteresis and timing conditions.

Application Scenarios

- **Sensor Value Debouncing** – Smoothing of analog or digital input signals (e.g., temperature, pressure, level) when thresholds are exceeded.
- **Noise Peak Filtering** – Prevention of false triggers caused by short-term disturbances.
- **Time-controlled signal monitoring** – When changes should only be reported at specific intervals (e.g., for logging or update rates).
- **Interface decoupling** – Used as a stabilized intermediate between a fast signal transmitter and a slow-moving actuator.

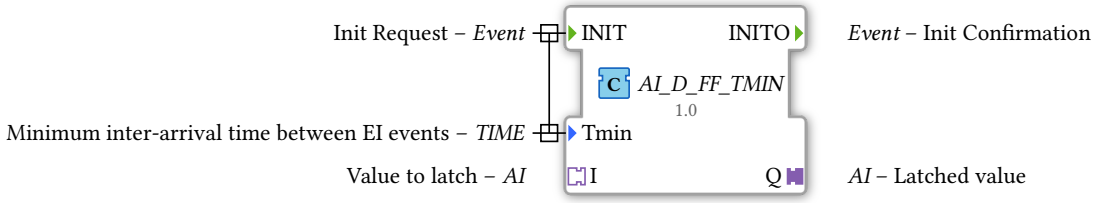
Comparison with similar components

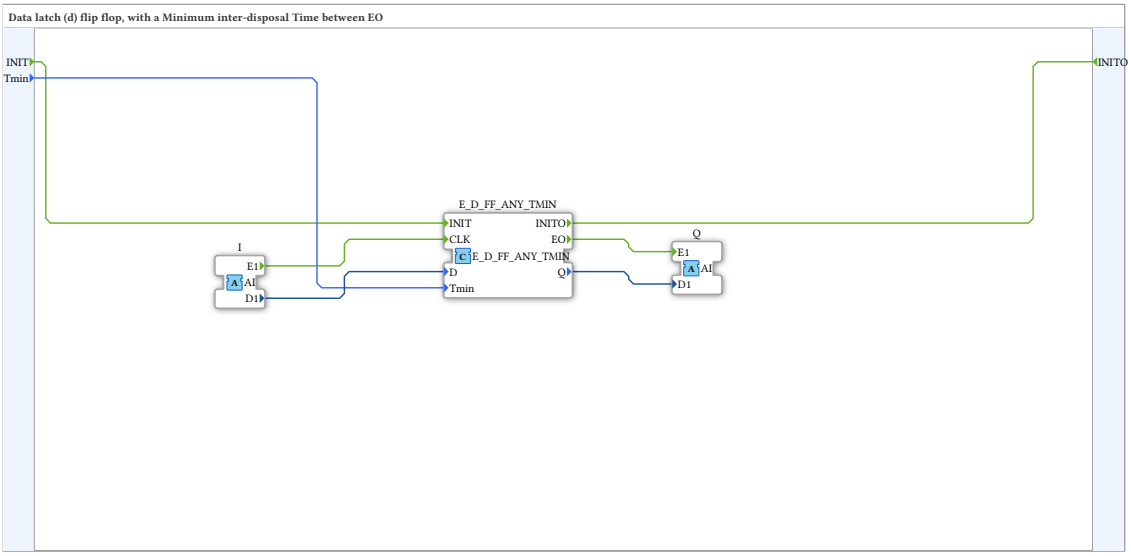
Component	Key difference
E_D_FF (simple D flip-flop)	No hysteresis band, no minimum time – accepts every change immediately.
AI_D_FF_HYS (without Tmin)	Includes hysteresis, but no time limit for the output.
AI_D_FF_TMIN (without hysteresis)	Includes only the minimum time, but no hysteresis.
AI_D_FF_HYS_TMIN (this one)	Combines both filters: hysteresis and time limiting.

Conclusion

The **AI_D_FF_HYS_TMIN** is a versatile function block for robust signal conditioning. Its combination of hysteresis and minimum interval makes it ideal for applications requiring both noise-resistant and time-controlled state changes. The adapter-based interface facilitates integration into modular automation systems.

AI_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AI_D_FF_TMIN** implements a D flip-flop (data latch) with a minimum inter-disposal time between successive output events. It serves as an adapter-encapsulated component that receives an

asynchronously incoming data value via a socket, temporarily stores it during an active event, and passes it on via a plug. Output is only released again when the configured minimum time (T_{\text{min}}) has elapsed since the last output.

Interface Structure

Event Inputs

Name	Type	With Variable	Comment
`INIT`	EInit	`Tmin`	Initialization Request

Event Outputs

Label	Type	With Variable	Comment	
`INITO`	EInit	–	Initialization Confirmation	

Data Inputs

Label	Type	Comment
`Tmin`	TIME	Minimum time interval between two EO events

Data Outputs

This function block does not have its own data outputs; output is exclusively via the adapter `Q`.

Adapter

Name	Type (Adapter)	Direction	Comment
`I`	`adapter::types::unidirectional::AI`	Socket	Input data for latching
`Q`	`adapter::types::unidirectional::AI`	Plug	Latched Output Value

Functionality

This function block encapsulates the internal function block `E_D_FF_ANY_TMIN`, which implements the actual D flip-flop with timing control. The interconnection is as follows:

- The initialization event `INIT` is passed to the internal function block along with the parameter `Tmin`.
- A data latch is triggered by an event on socket `I` (adapter event `I.E1`). This event serves as the clock signal (`CLK`) for the flip-flop.
- The current data value of `I.D1` is taken at each clock cycle and made available at the output of the internal function block.
- The internal function block generates an event at output `EO`, which is passed on via the adapter `Q` as `Q.E1`.
- There must be at least `EO` time between two consecutive events. If another clock cycle occurs during this lock time, the data value is latched, but the output is only released after the lock time has expired.
- The event `INITO` confirms the completion of the initialization process.

Technical Features

- **Adapter Interface:** Input and output are handled via unidirectional adapters of type `AI`. These enable loose coupling and easy reuse of the function block in different environments.
- **Minimum Output Time:** The parameter `Tmin` allows for targeted limitation of the event rate at the output. This prevents downstream components from being overloaded by excessively rapid event sequences.
- **Initialization Required:** Before first use, the function block must be initialized via `INIT`, as the internal state is set and `Tmin` is adopted during this phase.

State Overview

The function block does not have an explicitly visible state machine; the timing behavior is entirely controlled by the internal function block `E_D_FF_ANY_TMIN`. The function block essentially goes through the following phases:

1. **Initialization** – After receiving `INIT`, the internal function block is configured.
2. **Waiting for Clock** – The function block waits for an event at socket `I`.
3. **Latching** – Upon receiving a clock signal, the data value is acquired and an output event is generated.
4. **Lockout Phase** – After the output, any further output event is delayed for the duration of `Tmin`.
5. **Ready** – After the lockout period expires, another clock signal can be processed.

Application Scenarios

- **Sensor Data Acquisition:** An asynchronously operating sensor provides measured values that should only be transmitted to a control system at a defined minimum interval (e.g., temperature measurement with debouncing).
- **Buffering of Time-Critical Signals:** In automation applications where downstream processing modules tolerate a limited event rate, this module is used as a "rate limiter."
- **Synchronization:** This module can be used to synchronize cyclic processes by limiting the maximum update rate of an output signal.

Comparison with Similar Modules

Module	Description	Difference to AI_D_FF_TMIN
`E_D_FF`	Simple D flip-flop without a time limit	No minimum time between output events
`E_D_FF_ANY`	D flip-flop for arbitrary data types	Also without time control
`AI_D_FF`	D-Flip-Flop with Adapter Interface	Missing `Tmin` Option
`E_CYCLE`	Clock Generator with Adjustable Period	Generates Periodic Events, No Latch

The `AI_D_FF_TMIN` combines the functionality of an adapter-encapsulated flip-flop with a freely configurable output lock – a combination not found in standard flip-flops.

Conclusion

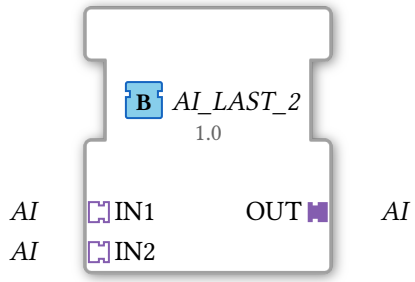
AI_D_FF_TMIN` offers a compact solution for buffering and time-controlled data transmission via adapter interfaces. The integrated minimum output time protects downstream components from overload and enables robust signal processing in industrial control systems. Thanks to its modular adapter interface, the component can be flexibly integrated into a wide variety of 4diac networks.

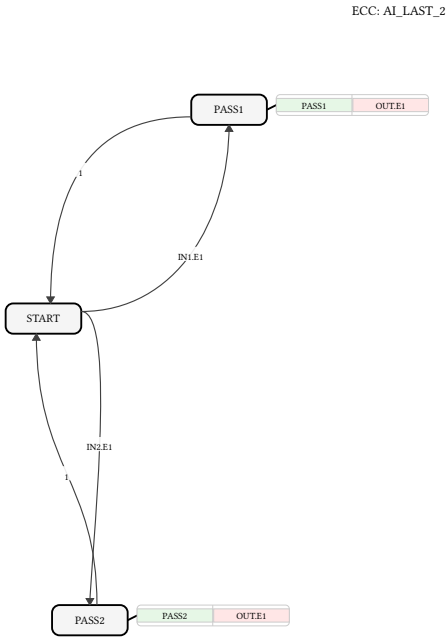
Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AI_LAST_2





Introduction

The AI_LAST_2 function block merges two same-type INT adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: INT adapter (unidirectional) - first signal source
- **IN2**: INT adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: INT adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

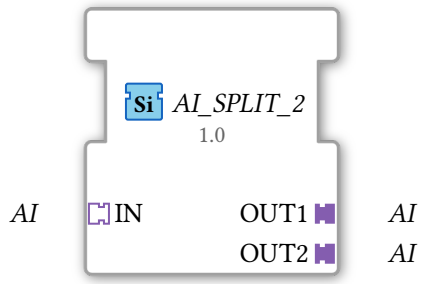
Application Scenarios

- Merging two equally-valid INT signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same INT value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AI_LAST_2 is the inverse of AI_SPLIT_2: AI_SPLIT_2 distributes 1 input to 2 outputs, AI_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use AX_AE_MERGE instead, which merges only the event without altering the data value.

AI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The AI_SPLIT_2 function block is used to distribute an analog input signal (AI) to two identical analog outputs. It is designed as a generic function block and allows for flexible further processing of the signal in two independent paths.

Interface Structure

The function block does not have event or data inputs/outputs in the traditional sense, but communicates exclusively via adapter interfaces of type `adapter::types::unidirectional::AI`.

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AI	Input (Socket)	Receives an analog input signal.
OUT1	adapter::types::unidirectional::AI	Output (Plug)	Passes the signal present at IN unchanged.
OUT2	adapter::types::unidirectional::AI	Output (Plug)	Passes the signal present at IN unchanged.

Functionality

The module passes the analog signal (AI) received via socket `IN` simultaneously to both outputs (`OUT1` and `OUT2`). No signal processing or modification takes place. The partitioning is purely passive and takes effect immediately. The adapters ensure the necessary interface compatibility.

Technical Features

- **Generic Structure**: The function block is implemented as a generic FB (`GEN_AI_SPLIT`) and can be used with various AI adapters that conform to the unidirectional AI protocol.

- **No State Logic**: There is no ECC (Execution Control Chart), therefore the behavior is purely combinatorial.
- **Platform Independence**: The function block is specified according to IEC 61499-2 and can be used in environments that support the adapter concept.

State Overview

The function block has no internal state behavior. The outputs follow the input signal directly and without delay.

Application Scenarios

- **Signal Distribution**: A single analog sensor (e.g., pressure sensor, temperature sensor) is to be evaluated by two independent control components.
- **Redundancy**: Splitting a signal for parallel monitoring and control paths.
- **Simulation**: Generating a second identical signal channel for testing or analysis purposes.

Comparison with similar function blocks

- **AI_SPLIT_3**: Extended to three outputs.
- **AI_SELECT**: Selects one of several inputs instead of distributing them.
- **AI_MERGE**: Combines multiple AI signals into one (e.g., average).

AI_SPLIT_2 is specialized for simple 1:2 distribution without logic or configuration.

Change Detection

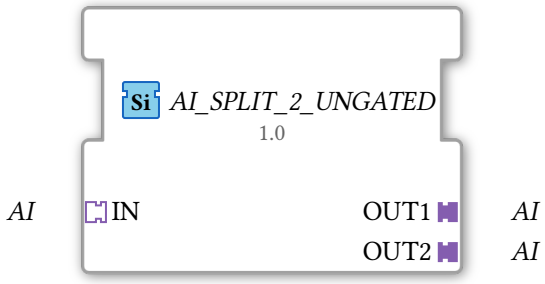
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AI_SPLIT_2 is a simple yet useful function block for passively splitting an analog input signal. Its generic adapter approach makes it flexible and facilitates the modular structuring of applications according to IEC 61499.

AI_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The AI_SPLIT_2_UNGATED function block is used to distribute an analog input signal (AI) to two identical analog outputs. It is designed as a generic function block and allows for flexible further processing of the signal in two independent paths.

Interface Structure

The function block does not have event or data inputs/outputs in the traditional sense, but communicates exclusively via adapter interfaces of type `adapter::types::unidirectional::AI`.

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AI	Input (Socket)	Receives an analog input signal.
OUT1	adapter::types::unidirectional::AI	Output (Plug)	Passes the signal present at IN unchanged.
OUT2	adapter::types::unidirectional::AI	Output (Plug)	Passes the signal present at IN unchanged.

Functionality

The module passes the analog signal (AI) received via socket `IN` simultaneously to both outputs (`OUT1` and `OUT2`). No signal processing or modification takes place. The partitioning is purely passive and takes effect immediately. The adapters ensure the necessary interface compatibility.

Technical Features

- **Generic Structure**: The function block is implemented as a generic FB (`GEN_AI_SPLIT`) and can be used with various AI adapters that conform to the unidirectional AI protocol.
- **No State Logic**: There is no ECC (Execution Control Chart), therefore the behavior is purely combinatorial.
- **Platform Independence**: The function block is specified according to IEC 61499-2 and can be used in environments that support the adapter concept.

State Overview

The function block has no internal state behavior. The outputs follow the input signal directly and without delay.

Application Scenarios

- **Signal Distribution**: A single analog sensor (e.g., pressure sensor, temperature sensor) is to be evaluated by two independent control components.
- **Redundancy**: Splitting a signal for parallel monitoring and control paths.
- **Simulation**: Generating a second identical signal channel for testing or analysis purposes.

Comparison with similar function blocks

- **AI_SPLIT_3**: Extended to three outputs.
- **AI_SELECT**: Selects one of several inputs instead of distributing them.
- **AI_MERGE**: Combines multiple AI signals into one (e.g., average).

AI_SPLIT_2_UNGATED is specialized for simple 1:2 distribution without logic or configuration.

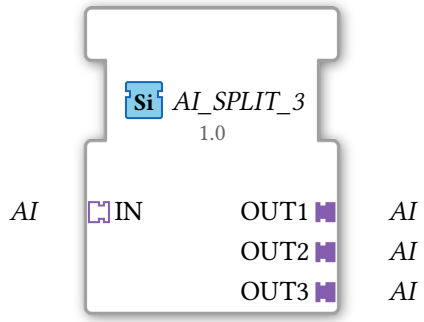
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AI_SPLIT_2_UNGATED is a simple yet useful function block for passively splitting an analog input signal. Its generic adapter approach makes it flexible and facilitates the modular structuring of applications according to IEC 61499.

AI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_3** is a generic function block for duplicating an analog input signal (AI). It splits an incoming AI signal into three separate, identical output signals. The function block is entirely defined via adapters of type `AI` and operates purely in a data flow-oriented manner without its own event or data point interfaces.

Interface Structure

The function block does not offer any direct event or data inputs/outputs. All communication takes place via **adapters**.

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

No direct data inputs – the AI signal is received via the socket adapter `IN`.

Data Outputs

No direct data outputs – the three identical AI signals are provided via the plug adapters `OUT1`, `OUT2`, and `OUT3`.

Adapters

Direction	Designation	Type	Description
Socket	`IN`	`ADAPTER_AI`	Incoming analog signal (source)
Plug	`OUT1`	`ADAPTER_AI`	First output – identical to the input signal
Plug	`OUT2`	`ADAPTER_AI`	Second output – identical to the input signal
Plug	`OUT3`	`ADAPTER_AI`	Third output – identical to the input signal

The adapters are of type `adapter::types::unidirectional::AI`. The component is generic, meaning that the specific data structure of the AI signal (e.g., value, unit, status) is only defined during instantiation by the connected adapters. ...|ADAPTER_AI` | qzmsdocs00009qz |

Functionality

The functional block forwards the analog signal received via socket `IN` unchanged and simultaneously to all three plug adapters `OUT1`, `OUT2`, and `OUT3`. No signal transformation, amplification, or delay takes place. The component functions purely as a **1-to-3 splitter**.

Since no events or state machines are present, the signal is passed purely via data flow. As soon as a new AI signal is present at `IN`, it is available at all three outputs.

Technical Features

- **Generic Function Block:** The function block is identified as `eclipse4diac::core::GenericClassName` using `'GEN_AI_SPLIT'`. This allows it to be used in various applications for different AI data types without requiring modification of the function block itself.
- **Pure Adapter Interface:** No direct event or data inputs/outputs are required. All data transmission occurs via the adapters, enabling flexible coupling with other function blocks.
- **No State Logic:** The function block does not have its own state machine (ECC) – its functionality is purely data flow-based and therefore very lightweight.

State Overview

The function block has **no** state machine or internal states. Signals are passed directly and without delay. Therefore, a state overview is not required.

Application Scenarios

- **Signal Distribution in Automation Systems:** An analog measured value (e.g., pressure sensor, temperature) is to be transmitted to multiple control or monitoring units.
- **Redundancy and Parallel Processing:** The same AI signal is required in parallel for different algorithms (e.g., control, alarming, logging).
- **Test Environments:** A simulated analog value can be distributed simultaneously to multiple components under test.

Comparison with Similar Components

- **AI_SPLIT_2:** Distributes the input signal to two outputs instead of three.
- **AI_SPLIT_4 / AI_SPLIT_N:** Extended versions with four or more outputs (if available).
- **AI_SELECT:** Selects one input from multiple inputs instead of duplicating a signal.
- **AI_MERGE:** Combines multiple AI inputs into one output (counterpart to the splitter).

All split blocks share the common feature of passing the input signal to all outputs without modification. The AI_SPLIT_3 represents a typical 3-way split.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

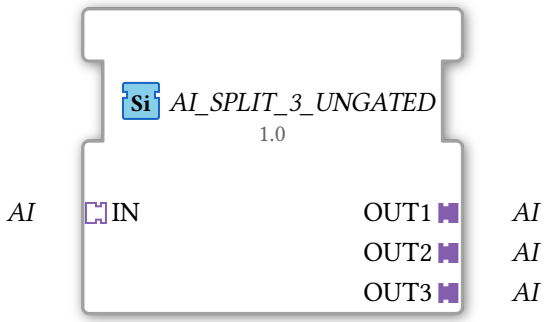
The **AI_SPLIT_3** is a simple yet effective block for multiplying an analog signal to three outputs. Its generic nature and the use of adapters allow it to integrate seamlessly into adapter-based 4diac applications. It is ideally suited for applications where an AI value needs to be distributed to multiple devices without requiring additional logic or configuration.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AI_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_3_UNGATED** is a generic function block for duplicating an analog input signal (AI). It splits an incoming AI signal into three separate, identical output signals. The function block is entirely defined via adapters of type `AI` and operates purely in a data flow-oriented manner without its own event or data point interfaces.

Interface Structure

The function block does not offer any direct event or data inputs/outputs. All communication takes place via **adapters**.

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

No direct data inputs – the AI signal is received via the socket adapter `IN`.

Data Outputs

No direct data outputs – the three identical AI signals are provided via the plug adapters `OUT1`, `OUT2`, and `OUT3`.

Adapters

Direction	Designation	Type	Description
Socket	`IN`	`ADAPTER_AI`	Incoming analog signal (source)
Plug	`OUT1`	`ADAPTER_AI`	First output – identical to the input signal
Plug	`OUT2`	`ADAPTER_AI`	Second output – identical to the input signal
Plug	`OUT3`	`ADAPTER_AI`	Third output – identical to the input signal

The adapters are of type `adapter::types::unidirectional::AI`. The component is generic, meaning that the specific data structure of the AI signal (e.g., value, unit, status) is only defined during instantiation by the connected adapters. …|ADAPTER_AI` | qzmsdocs00009qz |

Functionality

The functional block forwards the analog signal received via socket `IN` unchanged and simultaneously to all three plug adapters `OUT1`, `OUT2`, and `OUT3`. No signal transformation, amplification, or delay takes place. The component functions purely as a **1-to-3 splitter**.

Since no events or state machines are present, the signal is passed purely via data flow. As soon as a new AI signal is present at `IN`, it is available at all three outputs.

Technical Features

- **Generic Function Block:** The function block is identified as `eclipse4diac::core::GenericClassName` using `'GEN_AI_SPLIT'`. This allows it to be used in various applications for different AI data types without requiring modification of the function block itself.
- **Pure Adapter Interface:** No direct event or data inputs/outputs are required. All data transmission occurs via the adapters, enabling flexible coupling with other function blocks.
- **No State Logic:** The function block does not have its own state machine (ECC) – its functionality is purely data flow-based and therefore very lightweight.

State Overview

The function block has **no** state machine or internal states. Signals are passed directly and without delay. Therefore, a state overview is not required.

Application Scenarios

- **Signal Distribution in Automation Systems:** An analog measured value (e.g., pressure sensor, temperature) is to be transmitted to multiple control or monitoring units.
- **Redundancy and Parallel Processing:** The same AI signal is required in parallel for different algorithms (e.g., control, alarming, logging).
- **Test Environments:** A simulated analog value can be distributed simultaneously to multiple components under test.

Comparison with Similar Components

- **AI_SPLIT_2:** Distributes the input signal to two outputs instead of three.
- **AI_SPLIT_4 / AI_SPLIT_N:** Extended versions with four or more outputs (if available).
- **AI_SELECT:** Selects one input from multiple inputs instead of duplicating a signal.
- **AI_MERGE:** Combines multiple AI inputs into one output (counterpart to the splitter).

All split blocks share the common feature of passing the input signal to all outputs without modification. The AI_SPLIT_3_UNGATED represents a typical 3-way split.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

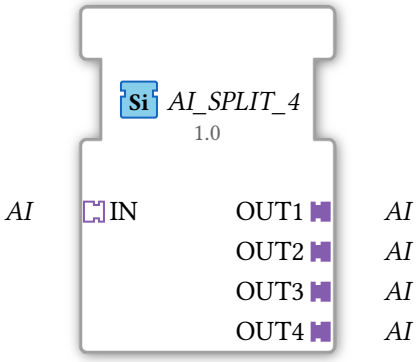
Conclusion

The **AI_SPLIT_3_UNGATED** is a simple yet effective block for multiplying an analog signal to three outputs. Its generic nature and the use of adapters allow it to integrate seamlessly into adapter-based 4diac applications. It is ideally suited for applications where an AI value needs to be distributed to multiple devices without requiring additional logic or configuration.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_4** is a generic block that splits a unidirectional analog input (AI) into four separate analog outputs. It serves as a signal distributor for analog signals and allows a single AI signal to be passed on to various subsequent blocks. The block is entirely adapter-based and has no built-in event or data interfaces. It is available under the Eclipse Public License 2.0.

Interface Structure

Event Inputs

- None

Event Outputs

- None

Data Inputs

- None

Data Outputs

- None

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AI`	Input of the analog signal to be distributed
Plug (Output)	`OUT1`	`adapter::types::unidirectional::AI`	First output (copy of the input signal)
Plug (Output)	`OUT2`	`adapter::types::unidirectional::AI`	Second output (copy of the input signal)
Plug (Output)	`OUT3`	`adapter::types::unidirectional::AI`	Third output (copy of the input signal)
Plug (output)	`OUT4`	`adapter::types::unidirectional::AI`	Fourth output (copy of the input signal)

Functionality

This module provides a simple 1:4 distribution of an analog signal. The signal present at adapter `IN` is passed unchanged to all four output adapters (`OUT1` … `OUT4`). No scaling, filtering, or conversion takes place – the signal is copied 1:1 at the fieldbus level. The number of channels is fixed at four.

Technical Features

- **Pure Adapter Block**: The function block contains no event or data interfaces; all communication takes place via adapters.
- **Generic Type**: The block is identified as generic via the attribute `GenericClassName` (`'GEN_AI_SPLIT'`). A specific type can be derived from this in the 4diac IDE.
- **Unidirectional**: The adapter type used, `unidirectional::AI`, only transmits data in one direction (from the socket to the plugs).
- **No State Machine**: Since there are no events, there is no ECC (Execution Control Chart). The function block is purely data flow-oriented.

State Overview

The function block does not have a state machine – it is event-free and continuously passes the input signal to all outputs. Therefore, a state overview is not available.

Application Scenarios

- **Analog Signal Distribution**: A single sensor value (e.g., temperature, pressure, fill level) is to be sent to multiple control logics or visualization components simultaneously.

- **Redundancy or Monitoring**: The signal is forwarded in parallel to a main controller and an independent monitoring module.
- **Decoupling of Subsystems**: In a modular control system, different subsystems can receive the same analog value without requiring multiple access to the sensor.

Comparison with Similar Modules

- **AI_SPLIT_2 / AI_SPLIT_8**: Similar modules with 2 or 8 outputs, respectively. The choice depends on the required number of parallel signals.
- **AI_SELECT** or **AI_MUX**: These modules select one input from several, while the AI_SPLIT multiplies the input signal.
- **Standard Adapter Split**: Some libraries offer generic split function blocks, but these often require additional event or data interfaces. The AI_SPLIT_4, with its pure adapter approach, is particularly easy to integrate into adapter chains.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AI_SPLIT_4** function block is a specialized yet easy-to-use function block for multiplying analog signals. Thanks to its pure adapter interface, it can be seamlessly integrated into 4diac projects without additional event logic. It is particularly suitable for modular and scalable automation solutions where a single measured value needs to be distributed to multiple devices.

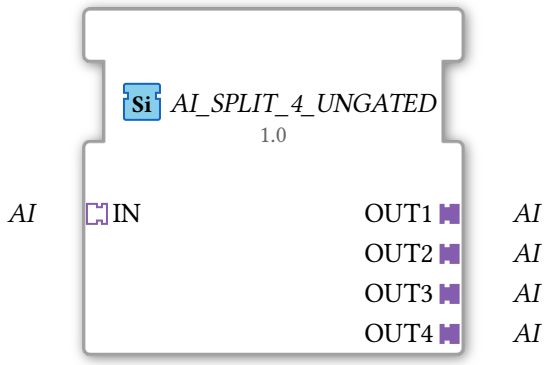
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AI_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_4_UNGATED** is a generic block that splits a unidirectional analog input (AI) into four separate analog outputs. It serves as a signal distributor for analog signals and allows a single AI signal to be passed on to various subsequent blocks. The block is entirely adapter-based and has no built-in event or data interfaces. It is available under the Eclipse Public License 2.0.

Interface Structure

Event Inputs
- None

Event Outputs
- None

Data Inputs
- None

Data Outputs
- None

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AI`	Input of the analog signal to be distributed
Plug (Output)	`OUT1`	`adapter::types::unidirectional::AI`	First output (copy of the input signal)
Plug (Output)	`OUT2`	`adapter::types::unidirectional::AI`	Second output (copy of the input signal)
Plug (Output)	`OUT3`	`adapter::types::unidirectional::AI`	Third output (copy of the input signal)
Plug (output)	`OUT4`	`adapter::types::unidirectional::AI`	Fourth output (copy of the input signal)

Functionality
This module provides a simple 1:4 distribution of an analog signal. The signal present at adapter `IN` is passed unchanged to all four output adapters (`OUT1` … `OUT4`). No scaling, filtering, or conversion takes place – the signal is copied 1:1 at the fieldbus level. The number of channels is fixed at four.

Technical Features
- **Pure Adapter Block**: The function block contains no event or data interfaces; all communication takes place via adapters.
- **Generic Type**: The block is identified as generic via the attribute `GenericClassName` (`'GEN_AI_SPLIT'`). A specific type can be derived from this in the 4diac IDE.
- **Unidirectional**: The adapter type used, `unidirectional::AI`, only transmits data in one direction (from the socket to the plugs).
- **No State Machine**: Since there are no events, there is no ECC (Execution Control Chart). The function block is purely data flow-oriented.

State Overview
The function block does not have a state machine – it is event-free and continuously passes the input signal to all outputs. Therefore, a state overview is not available.

Application Scenarios
- **Analog Signal Distribution**: A single sensor value (e.g., temperature, pressure, fill level) is to be sent to multiple control logics or visualization components simultaneously.
- **Redundancy or Monitoring**: The signal is forwarded in parallel to a main controller and an independent monitoring module.
- **Decoupling of Subsystems**: In a modular control system, different subsystems can receive the same analog value without requiring multiple access to the sensor.

Comparison with Similar Modules

- **AI_SPLIT_2 / AI_SPLIT_8**: Similar modules with 2 or 8 outputs, respectively. The choice depends on the required number of parallel signals.
- **AI_SELECT** or **AI_MUX**: These modules select one input from several, while the AI_SPLIT multiplies the input signal.
- **Standard Adapter Split**: Some libraries offer generic split function blocks, but these often require additional event or data interfaces. The AI_SPLIT_4_UNGATED, with its pure adapter approach, is particularly easy to integrate into adapter chains.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

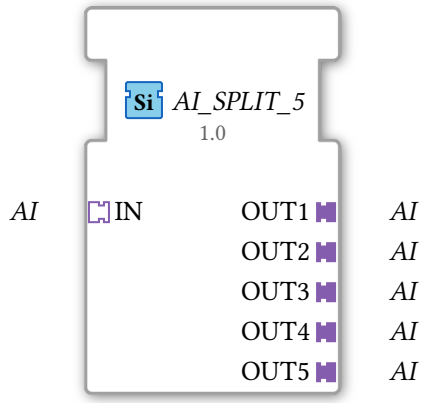
The **AI_SPLIT_4_UNGATED** function block is a specialized yet easy-to-use function block for multiplying analog signals. Thanks to its pure adapter interface, it can be seamlessly integrated into 4diac projects without additional event logic. It is particularly suitable for modular and scalable automation solutions where a single measured value needs to be distributed to multiple devices.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_5** is used to distribute an incoming analog signal (AI) to five identical outputs. It is a generic function block specifically designed for splitting an analog value into multiple parallel paths. It is implemented using unidirectional adapters of type `AI`.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available. Signal transmission occurs exclusively via the adapter input.

Data Outputs

No direct data outputs available. Signal transmission occurs exclusively via the adapter outputs.

Adapter

Direction	Name	Type	Description
Socket	IN	`adapter::types::unidirectional::AI`	Receives the incoming analog signal.
Plug	OUT1	`adapter::types::unidirectional::AI`	First output, provides the same signal as the input.
Plug	OUT2	`adapter::types::unidirectional::AI`	Second output, provides the same signal as the input.
Plug	OUT3	`adapter::types::unidirectional::AI`	Third output, provides the same signal as the input.
Plug	OUT4	`adapter::types::unidirectional::AI`	Fourth output provides the same signal as IN.
Plug	OUT5	`adapter::types::unidirectional::AI`	Fifth output provides the same signal as IN.

Functionality

The function block forwards the analog signal present at the adapter input *IN* unchanged to all five adapter outputs *OUT1* to *OUT5*. No signal processing or buffering takes place; the distribution is purely topological at the connection level. This makes the function block ideal for distributing a single analog signal to multiple subsequent functions or devices without requiring any duplication at the application level.

Technical Features

- **Generic Type**: The function block is implemented as a generic function block (`GEN_AI_SPLIT`) and can be adapted to different analog signal types by assigning appropriate type codes.
- **Unidirectional Adapters**: All adapters are unidirectional, meaning data flows only from the input to the outputs.
- **No Internal State Behavior**: The function block (FB) does not have its own state model (ECC) because it operates purely combinatorially.
- **Scalability**: The function block is designed for five outputs. For other numbers, similar function blocks (e.g., AI_SPLIT_2, AI_SPLIT_3) can be used or created by adapting the generic pattern.

State Overview

The FB does not have an explicit state diagram. Since no events are processed, the behavior is static: the signal at the input is permanently passed through to the outputs. Initialization or special behavior during commissioning is not required.

Application Scenarios

- **Distribution of a Measured Value**: An analog sensor (e.g., temperature, pressure) provides a value that must be transmitted in parallel to several control, monitoring, or visualization components.
- **Parallel Processing**: In a control application, the same analog value is processed differently in different branches (e.g., in both a PID controller and a limit value monitor).
- **Redundancy**: The signal is sent to multiple independent evaluation units to ensure fault tolerance.

Comparison with Similar Function Blocks

- **AI_SPLIT_2, AI_SPLIT_3, AI_SPLIT_N**: These function blocks split an analog signal to a different number of outputs. AI_SPLIT_5 is specifically optimized for five outputs.
- **Analog Splitters in Other Libraries**: Often, such functions are implemented using adapter coupling or data copiers. AI_SPLIT_5 offers a clean, reusable structure within the 4diac framework.
- **Difference to Data-Event-Based Splitters**: Since the function block does not use events, it is purely signal-flow oriented and requires no synchronization or sequence control.

AI_SPLIT_5 is not a data-event-based splitter.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AI_SPLIT_5 is a simple yet useful function block for duplicating an analog signal. Its generic nature and clear adapter interface make it an ideal building block for modular automation solutions where an analog value is needed multiple times. The lack of event-driven control reduces complexity and enables direct signal transmission without time delays.

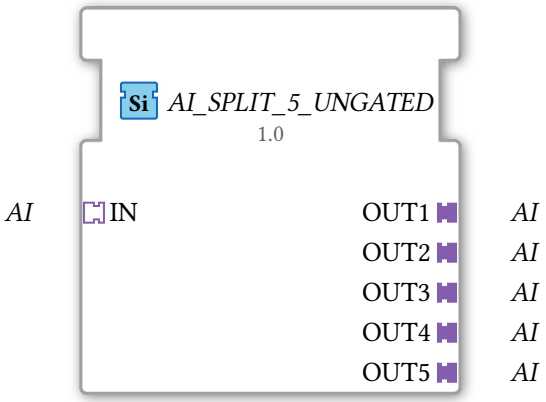
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AI_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_5_UNGATED** is used to distribute an incoming analog signal (AI) to five identical outputs. It is a generic function block specifically designed for splitting an analog value into multiple parallel paths. It is implemented using unidirectional adapters of type `AI`.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available. Signal transmission occurs exclusively via the adapter input.

Data Outputs

No direct data outputs available. Signal transmission occurs exclusively via the adapter outputs.

Adapter

Direction	Name	Type	Description
Socket	IN	`adapter::types::unidirectional::AI`	Receives the incoming analog signal.
Plug	OUT1	`adapter::types::unidirectional::AI`	First output, provides the same signal as the input.
Plug	OUT2	`adapter::types::unidirectional::AI`	Second output, provides the same signal as the input.
Plug	OUT3	`adapter::types::unidirectional::AI`	Third output, provides the same signal as the input.
Plug	OUT4	`adapter::types::unidirectional::AI`	Fourth output provides the same signal as IN.
Plug	OUT5	`adapter::types::unidirectional::AI`	Fifth output provides the same signal as IN.

Functionality

The function block forwards the analog signal present at the adapter input *IN* unchanged to all five adapter outputs *OUT1* to *OUT5*. No signal processing or buffering takes place; the distribution is purely topological at the connection level. This makes the function block ideal for distributing a single analog signal to multiple subsequent functions or devices without requiring any duplication at the application level.

Technical Features

- **Generic Type**: The function block is implemented as a generic function block (`GEN_AI_SPLIT`) and can be adapted to different analog signal types by assigning appropriate type codes.
- **Unidirectional Adapters**: All adapters are unidirectional, meaning data flows only from the input to the outputs.
- **No Internal State Behavior**: The function block (FB) does not have its own state model (ECC) because it operates purely combinatorially.
- **Scalability**: The function block is designed for five outputs. For other numbers, similar function blocks (e.g., AI_SPLIT_2, AI_SPLIT_3) can be used or created by adapting the generic pattern.

State Overview

The FB does not have an explicit state diagram. Since no events are processed, the behavior is static: the signal at the input is permanently passed through to the outputs. Initialization or special behavior during commissioning is not required.

Application Scenarios

- **Distribution of a Measured Value**: An analog sensor (e.g., temperature, pressure) provides a value that must be transmitted in parallel to several control, monitoring, or visualization components.
- **Parallel Processing**: In a control application, the same analog value is processed differently in different branches (e.g., in both a PID controller and a limit value monitor).
- **Redundancy**: The signal is sent to multiple independent evaluation units to ensure fault tolerance.

Comparison with Similar Function Blocks

- **AI_SPLIT_2, AI_SPLIT_3, AI_SPLIT_N**: These function blocks split an analog signal to a different number of outputs. AI_SPLIT_5_UNGATED is specifically optimized for five outputs.
- **Analog Splitters in Other Libraries**: Often, such functions are implemented using adapter coupling or data copiers. AI_SPLIT_5_UNGATED offers a clean, reusable structure within the 4diac framework.
- **Difference to Data-Event-Based Splitters**: Since the function block does not use events, it is purely signal-flow oriented and requires no synchronization or sequence control.

AI_SPLIT_5_UNGATED is not a data-event-based splitter.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AI_SPLIT_5_UNGATED is a simple yet useful function block for duplicating an analog signal. Its generic nature and clear adapter interface make it an ideal building block for modular automation solutions where an analog value is needed multiple times. The lack of event-driven control reduces complexity and enables direct signal transmission without time delays.

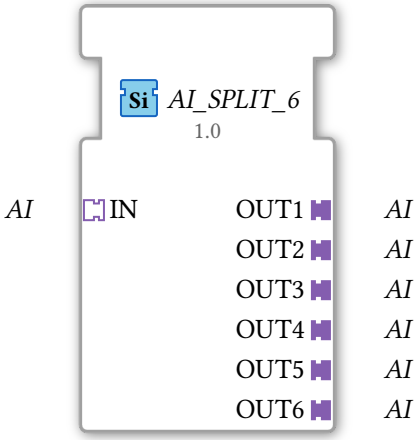
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AI_SPLIT_6

Introduction



The function block **AI_SPLIT_6** serves as a generic splitter that distributes an analog input value (AI) to six identical outputs. It implements a 1:6 distribution of an AI adapter without modifying any data or events. The block is defined as a generic FB (GenericClassName: `GEN_AI_SPLIT`) and operates purely on an adapter basis.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Name	Type	Description
Input (Socket)	IN	`adapter::types::unidirectional::AI`	Single Analog Input Adapter (Source)
Output (Plug)	OUT1	`adapter::types::unidirectional::AI`	First Output (1:1 pass-through from IN)
Output (Plug)	OUT2	`adapter::types::unidirectional::AI`	Second Output

Output (Plug)	OUT3	`adapter::types::unidirectional::AI`	Third Output
Output (Plug)	OUT4	`adapter::types::unidirectional::AI`	Fourth Output
Output (Plug)	OUT5	`adapter::types::unidirectional::AI`	Fifth Output
Output (Plug)	OUT6	`adapter::types::unidirectional::AI`	Sixth Output

Functionality

This function block has no internal logic or state machines. It functions as a passive splitter: The AI adapter connected to socket **IN** is duplicated unchanged to all six plug outputs **OUT1** to **OUT6**. Changes to the AI value (e.g., measured value or status) at the source are immediately and simultaneously visible at all outputs.

Technical Features

- **Generic Function Block**: The function block is marked as `eclipse4diac::core::GenericClassName`. This allows it to be parameterized or instantiated depending on the runtime environment without changing its basic splitter function.
- **Pure Adapter Coupling**: Neither events nor data port variables are used; only unidirectional adapters of type `adapter::types::unidirectional::AI` are employed. This reduces overhead and enables direct interconnection with other AI adapter blocks.
- **No States**: Since no ECC (Execution Control Chart) is present, the function block requires no algorithms or state transitions. The routing is purely structural.

State Overview

The function block does not have a state machine (ECC) and does not perform any active operations. There are no distinguishable operating states.

Application Scenarios

- **Distribution of an Analog Sensor Signal**: A single analog input (e.g., a 4-20 mA transmitter) is to be passed on in parallel to multiple control functions or display units.
- **Redundancy / Plausibility Check**: The same measured value is routed to several parallel evaluation blocks, which perform tasks such as averaging or error detection.
- **Simulation and Testing**: A simulated AI adapter is distributed across multiple test blocks to verify system behavior under load.

Comparison with Similar Blocks

- **AI_SPLIT_2 / AI_SPLIT_4**: These blocks split one AI input into two or four outputs, respectively. **AI_SPLIT_6** represents the variant with six outputs. All splitters function identically and differ only in the number of output adapters.
- **AI_DUPLICATE (hypothetical)**: A block that duplicates the value for copying purposes, but usually uses data ports. AI_SPLIT_6, on the other hand, uses adapters exclusively, allowing direct coupling without additional event control.
- `AI_SPLIT_6_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AI_SPLIT_6** is a minimal yet useful component for distributing analog adapter signals in the 4diac IDE. Its generic definition and pure adapter interface make it particularly suitable for architectures requiring loose coupling and easy extensibility. The distribution across six outputs enables flexible multiple use of a single analog input value.

Related topic subpages on ms-muc-docs.de

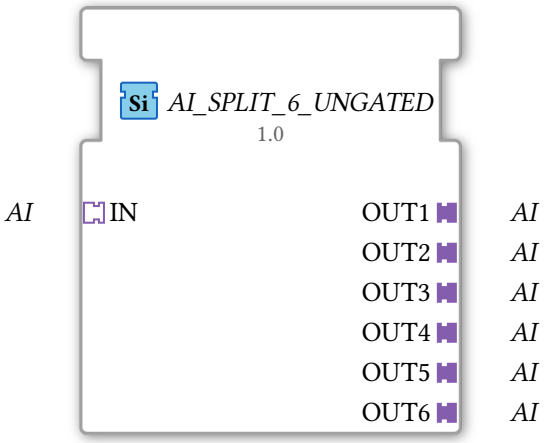
- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AI_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

Introduction



The function block **AI_SPLIT_6_UNGATED** serves as a generic splitter that distributes an analog input value (AI) to six identical outputs. It implements a 1:6 distribution of an AI adapter without modifying any data or events. The block is defined as a generic FB (GenericClassName: `GEN_AI_SPLIT`) and operates purely on an adapter basis.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Name	Type	Description
Input (Socket)	IN	`adapter::types::unidirectional::AI`	Single Analog Input Adapter (Source)
Output (Plug)	OUT1	`adapter::types::unidirectional::AI`	First Output (1:1 pass-through from IN)
Output (Plug)	OUT2	`adapter::types::unidirectional::AI`	Second Output

Output (Plug)	OUT3	`adapter::types::unidirectional::AI`	Third Output
Output (Plug)	OUT4	`adapter::types::unidirectional::AI`	Fourth Output
Output (Plug)	OUT5	`adapter::types::unidirectional::AI`	Fifth Output
Output (Plug)	OUT6	`adapter::types::unidirectional::AI`	Sixth Output

Functionality

This function block has no internal logic or state machines. It functions as a passive splitter: The AI adapter connected to socket **IN** is duplicated unchanged to all six plug outputs **OUT1** to **OUT6**. Changes to the AI value (e.g., measured value or status) at the source are immediately and simultaneously visible at all outputs.

Technical Features

- **Generic Function Block**: The function block is marked as `eclipse4diac::core::GenericClassName`. This allows it to be parameterized or instantiated depending on the runtime environment without changing its basic splitter function.
- **Pure Adapter Coupling**: Neither events nor data port variables are used; only unidirectional adapters of type `adapter::types::unidirectional::AI` are employed. This reduces overhead and enables direct interconnection with other AI adapter blocks.
- **No States**: Since no ECC (Execution Control Chart) is present, the function block requires no algorithms or state transitions. The routing is purely structural.

State Overview

The function block does not have a state machine (ECC) and does not perform any active operations. There are no distinguishable operating states.

Application Scenarios

- **Distribution of an Analog Sensor Signal**: A single analog input (e.g., a 4-20 mA transmitter) is to be passed on in parallel to multiple control functions or display units.
- **Redundancy / Plausibility Check**: The same measured value is routed to several parallel evaluation blocks, which perform tasks such as averaging or error detection.
- **Simulation and Testing**: A simulated AI adapter is distributed across multiple test blocks to verify system behavior under load.

Comparison with Similar Blocks

- **AI_SPLIT_2 / AI_SPLIT_4**: These blocks split one AI input into two or four outputs, respectively. **AI_SPLIT_6_UNGATED** represents the variant with six outputs. All splitters function identically and differ only in the number of output adapters.
- **AI_DUPLICATE (hypothetical)**: A block that duplicates the value for copying purposes, but usually uses data ports. AI_SPLIT_6_UNGATED, on the other hand, uses adapters exclusively, allowing direct coupling without additional event control.
- `AI_SPLIT_6`: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

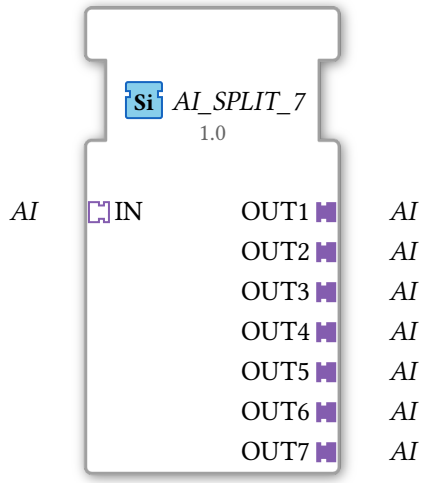
The **AI_SPLIT_6_UNGATED** is a minimal yet useful component for distributing analog adapter signals in the 4diac IDE. Its generic definition and pure adapter interface make it particularly suitable for architectures requiring loose coupling and easy extensibility. The distribution across six outputs enables flexible multiple use of a single analog input value.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_7** is used to distribute a single analog input (AI) to seven identical analog outputs. It is a generic function block that passes the incoming analog value unchanged to all seven output adapters. This allows multiple devices to be powered from a single signal source without having to acquire or duplicate the signal multiple times.

Interface Structure

Event Inputs

The function block has no event inputs. Data is passed exclusively via adapters.

Event Outputs

There are no event outputs.

Data Inputs

Direct data inputs are not defined. The analog value is read exclusively via the **IN** (socket) adapter.

Data Outputs

Direct data outputs are not available. The analog values are output via the seven adapters **OUT1** to **OUT7** (plugs).

Adapters

- **IN** (Socket): Type `adapter::types::unidirectional::AI` – Input for the analog signal.
- **OUT1** to **OUT7** (plugs): Type `adapter::types::unidirectional::AI` – Seven outputs that forward the incoming value identically.

Technical Features

AI_SPLIT_7 operates as a pure data distributor. An analog value present at the **IN** adapter is copied to all seven output adapters (**OUT1** to **OUT7**) without delay or transformation. This ensures that all connected components receive the same analog value. The internal logic is designed for unidirectional data transmission.

State Overview

- **Generic Function Block:** The function block is implemented as a generic type (`GEN_AI_SPLIT`) and can be flexibly integrated into various applications.
- **Adapter-Based:** The interfaces are implemented as IEC 61499 adapters, enabling loose coupling between function blocks.

- **No Event Control:** Since no events are used, data transmission occurs exclusively via the adapter connections and is driven by the runtime environment.

Application Scenarios

The function block has no explicit states because it contains no sequential or event-driven logic. It operates continuously as a pure signal distributor.

Comparison with Similar Function Blocks

- **Distributing a Sensor Signal:** An analog sensor (e.g., temperature, pressure) is to be transmitted to multiple control logics or displays.
- **Redundant Processing:** The same measured value is required in parallel by different algorithms or monitoring units.
- **AI_SPLIT_N:** Function blocks such as `AI_SPLIT_2` or `AI_SPLIT_3` differ only in the number of output adapters. `AI_SPLIT_7` offers the maximum distribution across seven channels.
- **Other splitters:** Unlike event-driven splitters (e.g., `E_SPLIT`), this function block operates purely analogously without event triggering. It is optimized for continuous analog signals.

Change Detection

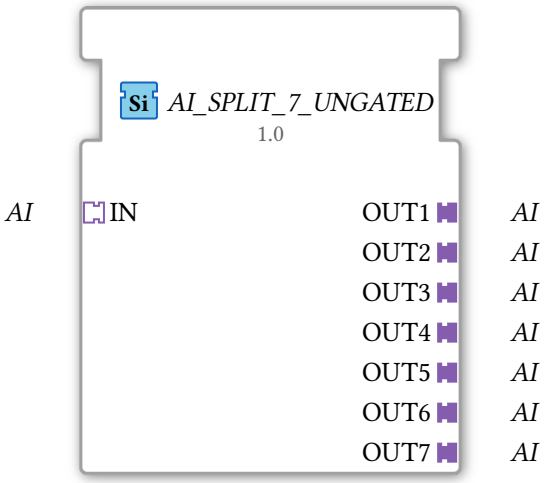
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AI_SPLIT_7 is a simple yet effective function block for multiplying analog signals in IEC 61499 systems. Through the use of adapters and its generic implementation, it can be easily integrated into various automation and control applications. Its clear, event-free structure makes it particularly suitable for unidirectional data transfer without additional control logic.

AI_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_7_UNGATED** is used to distribute a single analog input (AI) to seven identical analog outputs. It is a generic function block that passes the incoming analog value unchanged to

all seven output adapters. This allows multiple devices to be powered from a single signal source without having to acquire or duplicate the signal multiple times.

Interface Structure

Event Inputs
The function block has no event inputs. Data is passed exclusively via adapters.

Event Outputs
There are no event outputs.

Data Inputs
Direct data inputs are not defined. The analog value is read exclusively via the **IN** (socket) adapter.

Data Outputs
Direct data outputs are not available. The analog values are output via the seven adapters **OUT1** to **OUT7** (plugs).

Adapters
- **IN** (Socket): Type `adapter::types::unidirectional::AI` – Input for the analog signal.
- **OUT1** to **OUT7** (plugs): Type `adapter::types::unidirectional::AI` – Seven outputs that forward the incoming value identically.

Technical Features
AI_SPLIT_7_UNGATED operates as a pure data distributor. An analog value present at the **IN** adapter is copied to all seven output adapters (**OUT1** to **OUT7**) without delay or transformation. This ensures that all connected components receive the same analog value. The internal logic is designed for unidirectional data transmission.

State Overview
- **Generic Function Block:** The function block is implemented as a generic type (`GEN_AI_SPLIT`) and can be flexibly integrated into various applications.
- **Adapter-Based:** The interfaces are implemented as IEC 61499 adapters, enabling loose coupling between function blocks.
- **No Event Control:** Since no events are used, data transmission occurs exclusively via the adapter connections and is driven by the runtime environment.

Application Scenarios
The function block has no explicit states because it contains no sequential or event-driven logic. It operates continuously as a pure signal distributor.

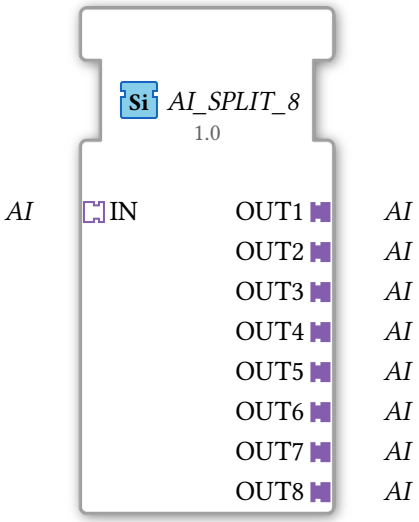
Comparison with Similar Function Blocks
- **Distributing a Sensor Signal:** An analog sensor (e.g., temperature, pressure) is to be transmitted to multiple control logics or displays.
- **Redundant Processing:** The same measured value is required in parallel by different algorithms or monitoring units.
- **AI_SPLIT_N:** Function blocks such as `AI_SPLIT_2` or `AI_SPLIT_3` differ only in the number of output adapters. `AI_SPLIT_7_UNGATED` offers the maximum distribution across seven channels.
- **Other splitters:** Unlike event-driven splitters (e.g., `E_SPLIT`), this function block operates purely analogously without event triggering. It is optimized for continuous analog signals.

Change Detection
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion
AI_SPLIT_7_UNGATED is a simple yet effective function block for multiplying analog signals in IEC 61499 systems. Through the use of adapters and its generic implementation, it can be easily integrated into

various automation and control applications. Its clear, event-free structure makes it particularly suitable for unidirectional data transfer without additional control logic.

AI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_8** is used to distribute an analog input signal (type *AI*) to eight identical analog output signals. The block is implemented as a generic FB (GenericClassName: `GEN_AI_SPLIT`) and is used in the 4diac IDE to make a single analog signal usable multiple times.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::AS`	Socket (Input)	Receives the analog signal, which is distributed to all outputs.
OUT1 – OUT8	`adapter::types::unidirectional::AP`	Plug (Output)	Eight identical outputs that provide the signal present at the input.

Functionality

The FB has no processing logic of its own (no state machine, no ECC). The analog signal present at the adapter socket **IN** is passed directly and without delay to all eight adapter plugs (**OUT1** to **OUT8**). No signal conversion or amplification takes place – the signal is duplicated one-to-one.

Technical Features

- **Generic Function Block:** The function block is declared as a generic type (`GEN_AI_SPLIT`) and can therefore be parameterized with a specific analog data type at runtime.
- **Unidirectional Adapters:** All adapters used are unidirectional, meaning that data flows exclusively from the socket to the plugs.
- **No Event Control:** Distribution is eventless; an incoming signal is immediately forwarded to all outputs.

State Overview

The function block does not have an internal state machine (ECC). There are no defined states – the function block operates continuously and without its own behavior.

Application Scenarios

- **Signal Distribution:** An analog sensor signal (e.g., temperature, pressure) is to be sent to multiple control modules or monitoring units simultaneously.
- **Redundancy:** Multiple parallel evaluations of the same signal in different parts of a system.
- **Prototyping:** Quickly deploy a signal to multiple test points in a development or simulation environment.

Comparison with similar function blocks

- **AI_SPLIT_2, AI_SPLIT_4:** These function blocks distribute an analog signal to two or four outputs, respectively. The present function block offers a higher number of parallel connections with eight outputs.
- **DIO_SPLIT (digital):** A similar splitter for digital signals, but based on DI/DO adapter types. The AI_SPLIT_8 is specifically designed for analog signals.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AI_SPLIT_8** is a simple, generic function block for multiplying an analog signal. Its clear structure—one input, eight outputs, no events—makes it ideal for straightforward signal distribution in automation applications. Its generic nature allows for flexible adaptation to various analog data types.

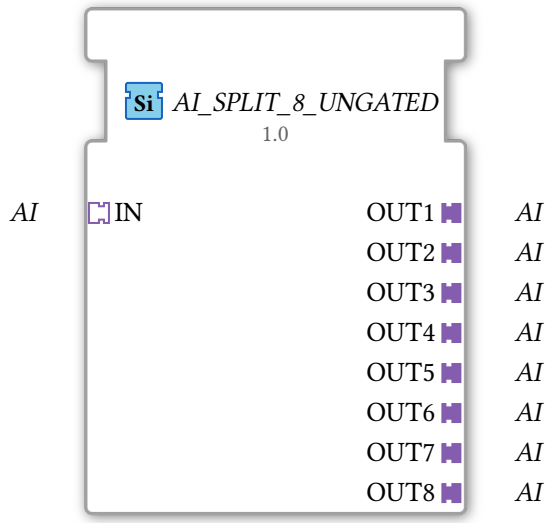
🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AI_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_8_UNGATED** is used to distribute an analog input signal (type *AI*) to eight identical analog output signals. The block is implemented as a generic FB (GenericClassName: `GEN_AI_SPLIT`) and is used in the 4diac IDE to make a single analog signal usable multiple times.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::AS`	Socket (Input)	Receives the analog signal, which is distributed to all outputs.
OUT1 – OUT8	`adapter::types::unidirectional::AP`	Plug (Output)	Eight identical outputs that provide the signal present at the input.

Functionality

The FB has no processing logic of its own (no state machine, no ECC). The analog signal present at the adapter socket **IN** is passed directly and without delay to all eight adapter plugs (**OUT1** to **OUT8**). No signal conversion or amplification takes place – the signal is duplicated one-to-one.

Technical Features

- **Generic Function Block:** The function block is declared as a generic type (`GEN_AI_SPLIT`) and can therefore be parameterized with a specific analog data type at runtime.

- **Unidirectional Adapters:** All adapters used are unidirectional, meaning that data flows exclusively from the socket to the plugs.
- **No Event Control:** Distribution is eventless; an incoming signal is immediately forwarded to all outputs.

State Overview

The function block does not have an internal state machine (ECC). There are no defined states – the function block operates continuously and without its own behavior.

Application Scenarios

- **Signal Distribution:** An analog sensor signal (e.g., temperature, pressure) is to be sent to multiple control modules or monitoring units simultaneously.
- **Redundancy:** Multiple parallel evaluations of the same signal in different parts of a system.
- **Prototyping:** Quickly deploy a signal to multiple test points in a development or simulation environment.

Comparison with similar function blocks

- **AI_SPLIT_2, AI_SPLIT_4:** These function blocks distribute an analog signal to two or four outputs, respectively. The present function block offers a higher number of parallel connections with eight outputs.
- **DIO_SPLIT (digital):** A similar splitter for digital signals, but based on DI/DO adapter types. The AI_SPLIT_8_UNGATED is specifically designed for analog signals.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

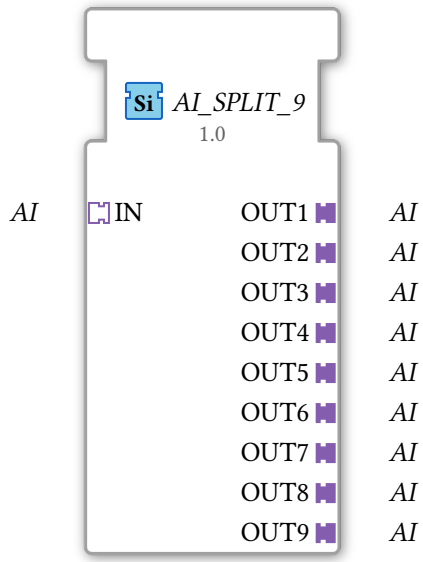
The **AI_SPLIT_8_UNGATED** is a simple, generic function block for multiplying an analog signal. Its clear structure—one input, eight outputs, no events—makes it ideal for straightforward signal distribution in automation applications. Its generic nature allows for flexible adaptation to various analog data types.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_9** is a generic component that distributes a single analog input signal (adapter type `AI`) to nine identical outputs (OUT1 to OUT9). It serves as a signal distributor, allowing an AI signal to be used multiple times in a control application without having to link the source multiple times.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Name	Type	Description
Input (Socket)	`IN`	`adapter::types::unidirectional::AI`	Single analog input distributed to all outputs.
Output (Plug)	`OUT1`	`adapter::types::unidirectional::AI`	First outgoing AI channel.
Output (Plug)	`OUT2`	`adapter::types::unidirectional::AI`	Second outgoing AI channel.
Output (Plug)	`OUT3`	`adapter::types::unidirectional::AI`	Third outgoing AI channel.
Output (Plug)	`OUT4`	`adapter::types::unidirectional::AI`	Fourth outgoing AI channel.
Output (Plug)	`OUT5`	`adapter::types::unidirectional::AI`	Fifth outgoing AI channel.
Output (Plug)	`OUT6`	`adapter::types::unidirectional::AI`	Sixth outgoing AI channel.
Output (Plug)	`OUT7`	`adapter::types::unidirectional::AI`	Seventh outgoing AI channel.
Output (Plug)	`OUT8`	`adapter::types::unidirectional::AI`	Eighth outgoing AI channel.
Output (Plug)	`OUT9`	`adapter::types::unidirectional::AI`	Ninth outgoing AI channel.

Functionality

The module operates as a pure signal distributor: The AI value present at adapter socket `IN` is passed on unchanged and simultaneously to all nine adapter plugs `OUT1` … `OUT9`. No signal processing, scaling, or other manipulation takes place. The distribution is continuous and without internal state behavior – the output follows the input without delay.

Technical Features

- **Generic Block:** The function block is typed as `GEN_AI_SPLIT` and utilizes the generics mechanisms of Eclipse 4diac (`GenericClassName`, `TypeHash`). This allows for flexible use in type hierarchies and libraries.
- **Adapter-Based Interface:** Instead of individual data ports, adapters are used that provide an encapsulated, reusable interface for analog inputs (`AI`). This simplifies the interconnection concept in 4diac applications.
- **Zero Latency:** Since there is no internal logic or processes, signal transmission is instantaneous.

State Overview

The block does not have an internal state machine (no ECC). It operates purely on a data flow basis: The input value is continuously passed through to the outputs.

Application Scenarios

- **Signal Multiplication:** A single analog sensor (e.g., temperature, pressure) is to be evaluated in parallel across multiple function blocks.
- **Redundant Processing:** An AI signal is routed to multiple independent paths to enable failover or plausibility checks.
- **Modular Distribution:** In a large control application, an analog input signal is routed to various modules (e.g., control, monitoring, visualization).

Comparison with Similar Components

- **AI_SPLIT_2 / AI_SPLIT_4 / ...**: Analog components with 2 or 4 outputs, respectively. `AI_SPLIT_9` extends the concept to nine outputs and is therefore suitable for applications with many receivers.
- **AI_MERGE**: Combines multiple AI inputs into a single output – the opposite operation.
- **Direct Adapter Linking**: In 4diac, a single adapter port can theoretically be connected multiple times, but the explicit splitting with this function block is more documented and clearly structured.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AI_SPLIT_9** is a simple yet useful function block for signal multiplication of analog inputs in IEC 61499 applications. Its generic structure and adapter interface allow it to integrate seamlessly into the 4diac ecosystem and facilitate the structured routing of signals. Its use is recommended wherever an AI signal needs to be simultaneously passed to multiple processing units.

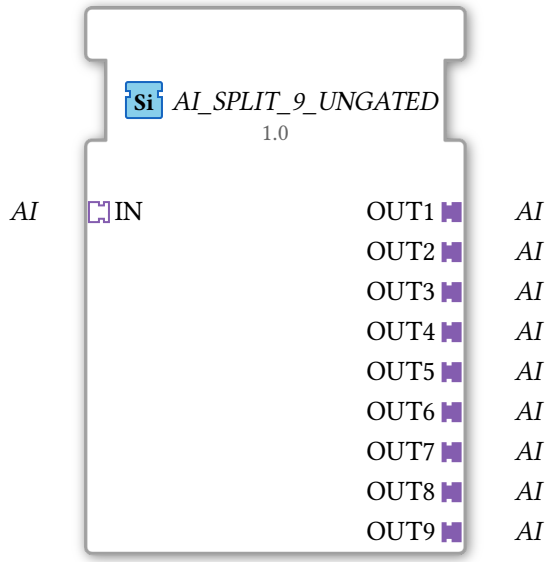
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AI_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_SPLIT_9_UNGATED** is a generic component that distributes a single analog input signal (adapter type `AI`) to nine identical outputs (OUT1 to OUT9). It serves as a signal distributor, allowing an AI signal to be used multiple times in a control application without having to link the source multiple times.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Name	Type	Description
Input (Socket)	`IN`	`adapter::types::unidirectional::AI`	Single analog input distributed to all outputs.
Output (Plug)	`OUT1`	`adapter::types::unidirectional::AI`	First outgoing AI channel.
Output (Plug)	`OUT2`	`adapter::types::unidirectional::AI`	Second outgoing AI channel.
Output (Plug)	`OUT3`	`adapter::types::unidirectional::AI`	Third outgoing AI channel.
Output (Plug)	`OUT4`	`adapter::types::unidirectional::AI`	Fourth outgoing AI channel.
Output (Plug)	`OUT5`	`adapter::types::unidirectional::AI`	Fifth outgoing AI channel.
Output (Plug)	`OUT6`	`adapter::types::unidirectional::AI`	Sixth outgoing AI channel.
Output (Plug)	`OUT7`	`adapter::types::unidirectional::AI`	Seventh outgoing AI channel.
Output (Plug)	`OUT8`	`adapter::types::unidirectional::AI`	Eighth outgoing AI channel.
Output (Plug)	`OUT9`	`adapter::types::unidirectional::AI`	Ninth outgoing AI channel.

Functionality

The module operates as a pure signal distributor: The AI value present at adapter socket `IN` is passed on unchanged and simultaneously to all nine adapter plugs `OUT1` … `OUT9`. No signal processing, scaling, or other manipulation takes place. The distribution is continuous and without internal state behavior – the output follows the input without delay.

Technical Features

- **Generic Block:** The function block is typed as `GEN_AI_SPLIT` and utilizes the generics mechanisms of Eclipse 4diac (`GenericClassName`, `TypeHash`). This allows for flexible use in type hierarchies and libraries.
- **Adapter-Based Interface:** Instead of individual data ports, adapters are used that provide an encapsulated, reusable interface for analog inputs (`AI`). This simplifies the interconnection concept in 4diac applications.
- **Zero Latency:** Since there is no internal logic or processes, signal transmission is instantaneous.

State Overview

The block does not have an internal state machine (no ECC). It operates purely on a data flow basis: The input value is continuously passed through to the outputs.

Application Scenarios

- **Signal Multiplication:** A single analog sensor (e.g., temperature, pressure) is to be evaluated in parallel across multiple function blocks.
- **Redundant Processing:** An AI signal is routed to multiple independent paths to enable failover or plausibility checks.
- **Modular Distribution:** In a large control application, an analog input signal is routed to various modules (e.g., control, monitoring, visualization).

Comparison with Similar Components

- **AI_SPLIT_2 / AI_SPLIT_4 / ...**: Analog components with 2 or 4 outputs, respectively. `AI_SPLIT_9_UNGATED` extends the concept to nine outputs and is therefore suitable for applications with many receivers.
- **AI_MERGE**: Combines multiple AI inputs into a single output – the opposite operation.
- **Direct Adapter Linking**: In 4diac, a single adapter port can theoretically be connected multiple times, but the explicit splitting with this function block is more documented and clearly structured.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AI_SPLIT_9_UNGATED** is a simple yet useful function block for signal multiplication of analog inputs in IEC 61499 applications. Its generic structure and adapter interface allow it to integrate seamlessly into the 4diac ecosystem and facilitate the structured routing of signals. Its use is recommended wherever an AI signal needs to be simultaneously passed to multiple processing units.

🌐 Related topic subpages on ms-muc-docs.de

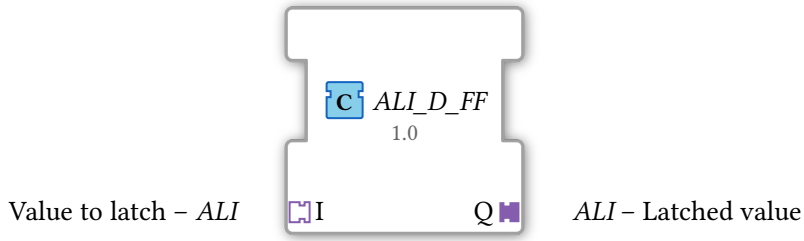
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

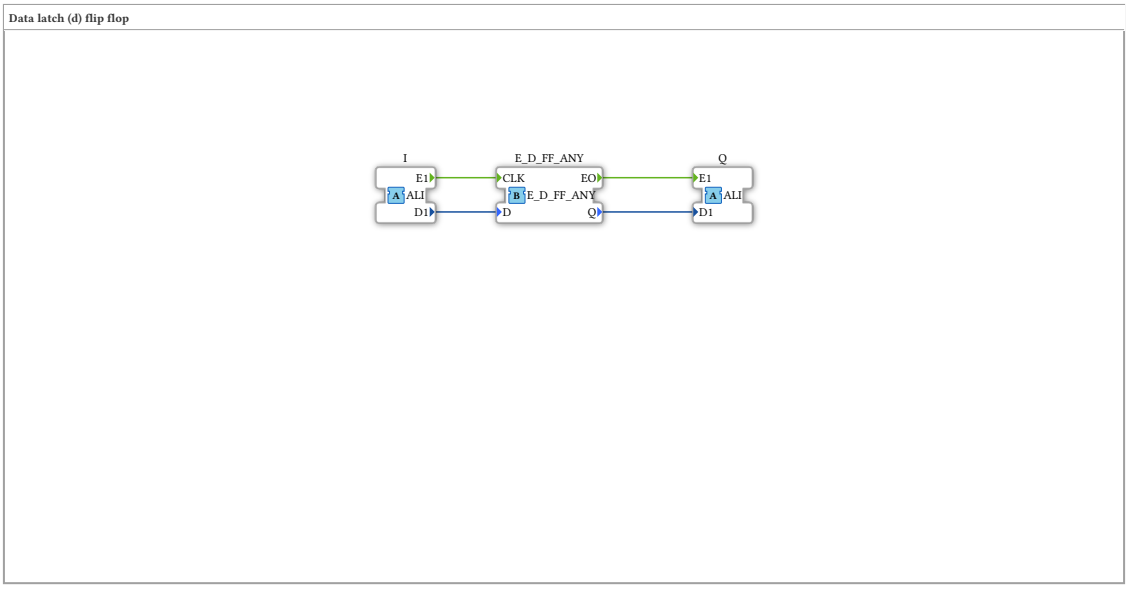
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

ALI_D_FF





• – ○ • – ○ • – ○ •

Introduction

The **ALI_D_FF** function block implements a clock-controlled D flip-flop (data latch) via adapter interfaces. It stores a data value on the rising edge of a clock signal and outputs it until the next clock edge. This function block is suitable for synchronous data storage in event-driven automation environments.

Interface Structure

Event Inputs

- **I.E1** (via adapter I): Clock input – the data is stored on a rising edge.

Event Outputs

- **Q.E1** (via adapter Q): Confirmation of data storage – output after successful edge detection and value assignment.

Data Inputs

- **I.D1** (via adapter I): Data input – the value to be stored (any data type, depending on the adapter type `ALI`).

Data Outputs

- **Q.D1** (via adapter Q): Data output – the currently stored (latched) value.

Adapters

Name	Type	Direction	Description
I	`adapter::types::unidirectional::ALI`	Socket	Provides event and data input.
Q	`adapter::types::unidirectional::ALI`	Plug	Provides event and data output.

Functionality

The function block internally uses the standard function `E_D_FF_ANY` from the IEC 61499 library.

- On a rising edge at the event input **I.E1**, the current value of **I.D1** is transferred to the internal state.
- Subsequently, an event is generated at the output **Q.E1**, signaling that the new value is present at **Q.D1**.
- The data output **Q.D1** retains the stored value until a new rising edge occurs.

Technical Features

- **Adapter-based interface**: The function block does not have direct event/data inputs but uses two adapters (`I` and `Q`). This allows for flexible encapsulation of the connections and facilitates reuse in complex networks.
- **Internal Implementation**: The core of the function block is the FB `E_D_FF_ANY`, which operates independently of data type. The data type is determined by the adapters used.
- **Event-driven**: State changes occur exclusively upon events, allowing integration into event-driven processes.

State Overview

The FB has two logical states:

- **State 0**: Output `Q.D1` contains the last latched value, which can be, for example, logical 0.
- **State 1**: Output `Q.D1` contains a valid value other than 0 (provided the data type allows this).

A state change occurs only on a rising edge of `I.E1`. Without a clock signal, the output remains stable.

Application Scenarios

- **Data Synchronization**: Buffering a sensor signal until the next clock cycle of a controller.
- **State Memory**: Maintaining a switching state in sequence controllers.
- **Feedback-Free Delay**: Temporarily storing values in data paths to avoid asynchronous concurrent execution.

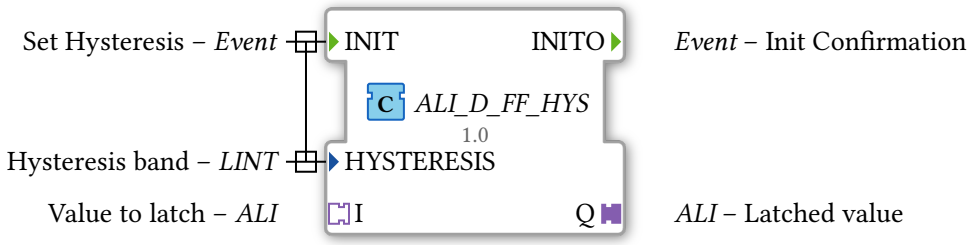
Comparison with Similar Components

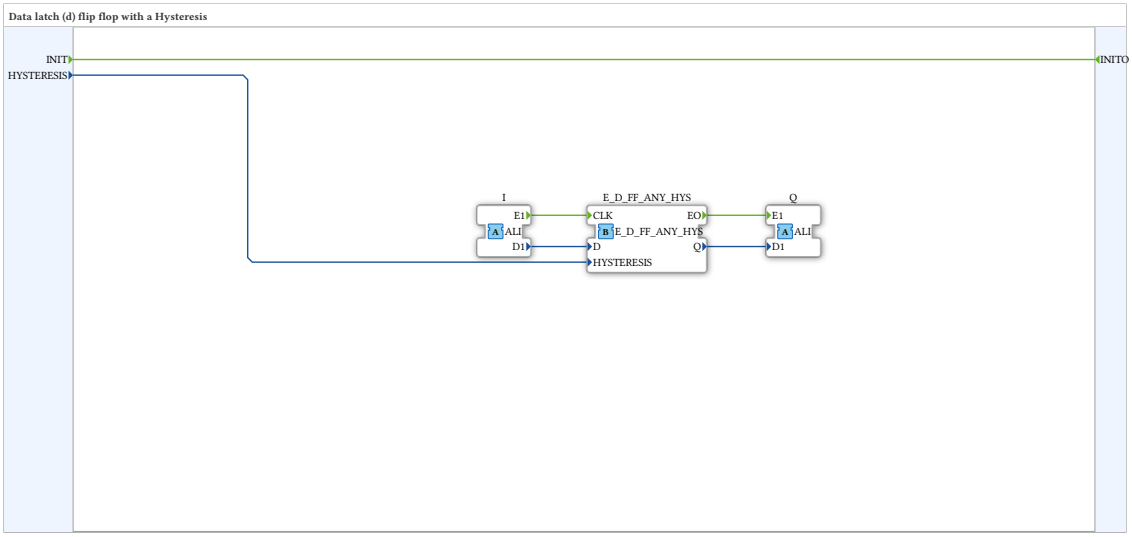
- **ALI_D_FF vs. Simple D-FF (e.g., `E_D_FF_ANY`)**: The ALI_D_FF encapsulates event and data connections in adapters, making it more modular and easier to integrate into predefined interfaces.
- **ALI_D_FF vs. RS Flip-Flop**: An RS flip-flop allows setting and resetting (asynchronously), while the D flip-flop operates purely clock-driven and does not offer direct set/reset inputs.
- **ALI_D_FF vs. T-Flip-Flop**: The T-FF toggles on every clock cycle, while the D-FF takes an external value.

Conclusion

The **ALI_D_FF** offers a clean, adapter-based solution for synchronous data storage. It combines the proven functionality of a D-flip-flop with the flexibility of IEC 61499 adapters. This makes it particularly suitable for modular, reusable automation components in distributed systems.

ALI_D_FF_HYS





• – ◦ • – ◦ • – ◦ •

Introduction

The **ALI_D_FF_HYS** function block implements a data latch (D flip-flop) with a hysteresis function. It serves for the stable and debounced transmission of a binary signal: An incoming data signal is received and output on a rising clock edge, with configurable hysteresis preventing small signal changes (noise) from causing unwanted output changes. The component is specifically designed for use with the unidirectional **ALI** adapter structure in IEC 61499 environments.

Interface Structure

Event Inputs

Event	Type	Comment
`INIT`	EInit	Sets the hysteresis value for the switching threshold.

Event Outputs

Event	Type	Comment
`INITO`	EInit	Confirmation of successful initialization.

Data Inputs

Variable	Type	Comment
`HYSTERESIS`	LINT	Hysteresis bandwidth (e.g., as an integer)

Data Outputs

No direct data outputs – output is via the adapter `Q`.

Adapters

Direction	Name	Type	Comment
Socket	`I`	`adapter::types::unidirectional::ALI`	Input value (data + clock)
Plug	`Q`	`adapter::types::unidirectional::ALI`	Output value (latched value)

The adapter `I` provides two signals:

- **`I.E1`** – Clock signal (event)
- **`I.D1`** – Data input (e.g., BOOL or numeric)

The adapter `Q` provides the latched value as **`Q.D1`**.

Functionality

The function block works internally with another function block **E_D_FF_ANY_HYS** (from the library `logiBUS::signalprocessing::hysteresis`), which implements the core logic. Overview of how it works:

1. **Initialization**

When the event `INIT` arrives, the value of `HYSTERESIS` is passed to the inner component, and the event is immediately acknowledged as `INITO`.

1. **Data Processing**

The event and data connections are directly connected between the adapters and the internal module:

- `I.E1` ⟶ **CLK** (Clock signal)
- `I.D1` ⟶ **D** (Input data)
- **Q** ⟶ `Q.D1` (Lagged output signal)

1. **Hysteresis Behavior**

At each clock cycle (`CLK`), the current data value `D` is evaluated. If `D` lies within the hysteresis band around the current output value `Q`, the output remains unchanged. If `D` exceeds the upper or lower threshold, the new value is adopted. This suppresses interference and ensures a clean, debounced signal.

Technical Features

- **Pure Adapter Communication**: The function block uses only unidirectional adapters (`ALI`) for event and data transmission. This makes it easy to integrate into modular, adapter-based architectures.
- **Indirect Hysteresis Implementation**: Hysteresis is not implemented directly in the function block body, but rather through the internal block `E_D_FF_ANY_HYS`. This allows for flexible reuse of the logic.
- **Immediate INIT Acknowledgement**: The `INIT` event is acknowledged immediately via a direct connection (`INIT → INITO`) without waiting for the hysteresis logic.
- **Hysteresis Parameter Type**: `HYSTERESIS` is defined as `LINT`, thus allowing large integer values.

State Overview

The function block itself has no visible states, as all state logic is encapsulated within the internal function block. Behavior of the D flip-flop with hysteresis:

- **Idle State**: Output values remain stable as long as no clock signal is applied.
- **Clock Edge (Rising)**:
- If D > Q + HYSTERESIS ⟶ Q = D (Set)
- If D < Q - HYSTERESIS ⟶ Q = D (Reset)
- Otherwise ⟶ Q remains unchanged (hysteresis range)

Application Scenarios

Typical applications for this function block include:

- **Debouncing of switch signals** in automation technology (e.g., limit switches, proximity sensors).
- **Signal conditioning in harsh environments** (agricultural technology, mechanical engineering) – the hysteresis prevents oscillations caused by vibrations or noise.
- **Basic building block for more complex logic** with adapter-based data transmission, e.g., as an input stage of a state machine.

Comparison with Similar Function Blocks

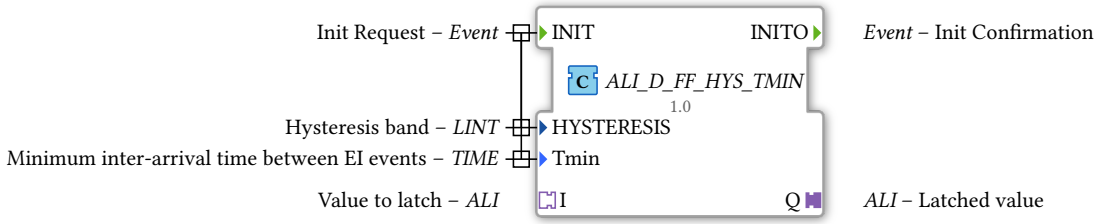
Function Block	Feature
ALI_D_FF (without hysteresis)	Simple D flip-flop without noise suppression
ALI_D_FF_HYS (this one)	D flip-flop with configurable hysteresis for noise filtering
Schmitt trigger function block	Similar function, but often with fixed thresholds
E_D_FF_ANY_HYS	Internal logic, generic type (no adapter format)

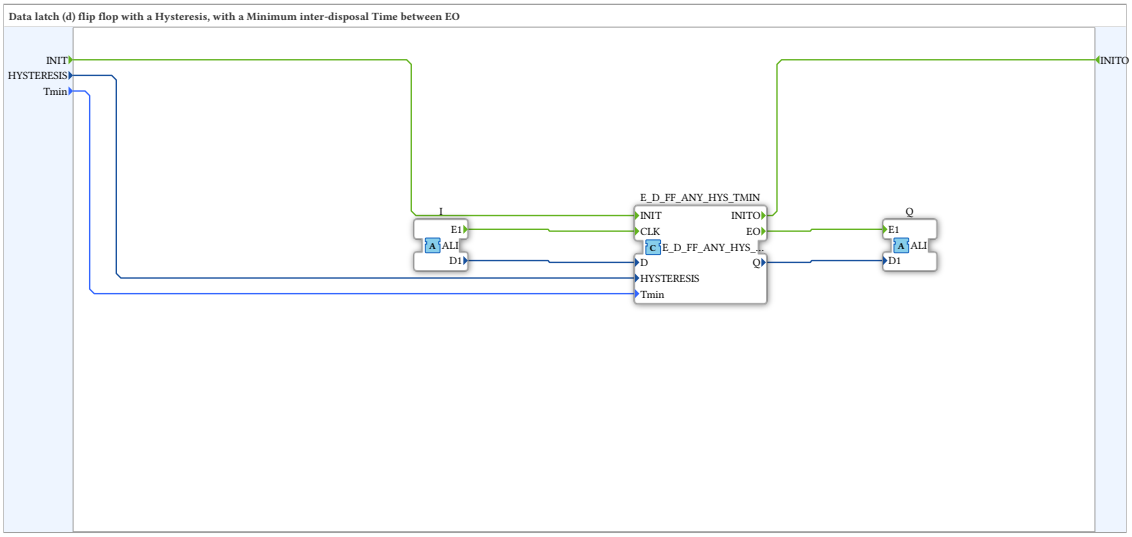
The advantage of **ALI_D_FF_HYS** lies in its direct adapter connection, enabling seamless integration into existing IEC 61499 adapter networks.

Conclusion

The **ALI_D_FF_HYS** function block offers a proven combination of a D flip-flop and adjustable hysteresis, packaged in a standards-compliant IEC 61499 adapter interface. It is particularly suitable for applications where signals must be processed reliably under real-world conditions (noise, bounce). The clear separation of initialization and data processing, as well as the use of a proven sub-component, makes it robust, maintainable and flexible in its application.

ALI_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **ALI_D_FF_HYS_TMIN** implements a data-driven D flip-flop with an adjustable hysteresis width (threshold band) and a minimum waiting time between two consecutive output events. It is used for debouncing and stabilizing signal transitions in industrial automation, particularly when a signal should only be considered valid after it has left a defined tolerance range and a minimum time has elapsed since the last valid edge.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization request – sets the hysteresis width and the minimum time.

Event Outputs

Name	Type	Comment
INITO	EInit	Confirmation of successful initialization.

Data Inputs

Name	Type	Comment
HYSTERESIS	LINT	Hysteresis band as an integer value (LINT).
Tmin	TIME	Minimum time between two output events (Inter-Disposal Time).

Data Outputs

No direct data outputs available.

The function block outputs its status via the adapter **Q** (see Adapters).

Adapters

| Type | Name | Direction | Comment | | `adapter::types::unidirectional::ALI` | I | Socket | Input adapter – provides the signal to be received (event and data value). | | `adapter::types::unidirectional::ALI` | Q | Plug | Output adapter – outputs the last valid data value and an event. |

The adapter type `ALI` includes an event (E1) and a data value (D1) and is used for unidirectional communication between function blocks.

Functionality

The function block **ALI_D_FF_HYS_TMIN** is implemented as a composition and internally uses the function block `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS_TMIN`. Its operation can be described as follows:

1. **Initialization** – The **INIT** event sets the **HYSTERESIS** and **Tmin** parameters. After successful initialization, **INITO** is triggered.
2. **Signal Acquisition** – Every event at socket **I** (event **I.E1**) is interpreted as a clock edge. The corresponding data value **I.D1** is only transferred to internal memory if it is outside the hysteresis band around the currently stored value (i.e., the change exceeds the hysteresis).
3. **Minimum Time Control** – After a valid value is acquired, output via plug **Q** (event **Q.E1** and value **Q.D1**) is only enabled after at least **Tmin** has elapsed since the last output event. This suppresses unwanted rapid signal changes.
4. **Signal Acquisition** – After a valid value is acquired, output via plug **Q** (event **Q.E1** and value **Q.D1**) is only enabled after at least **Tmin** has elapsed since the last output event. This suppresses unwanted rapid signal changes.
5. **Output** – Once the minimum time has elapsed, the new value is output via the output adapter **Q**. The function block then waits for the next input event.

Technical Features

- **Hysteresis Data Width** – The hysteresis is defined as a LINT (64-bit integer); this allows for the processing of very large value ranges.
- **Time Parameters** – The minimum inter-disposal time **Tmin** is specified as IEC-61499-`TIME` (e.g., `T#10ms`).
- **Adapter-Based Coupling** – The use of the adapters `ALI` allows loose coupling between signal-generating and signal-processing function blocks. The socket **I** receives the data, and the plug **Q** provides it.
- **Type of Dependency** – The function block is based on the internal FB `E_D_FF_ANY_HYS_TMIN`, which implements the actual hysteresis and timing logic. The outer function block merely encapsulates the adapter interface.

State Overview

The function block goes through the following states:

1. **INIT** – Waiting for initialization parameters.
2. **IDLE** – Ready; waiting for an event on socket **I**.
3. **SAMPLE** – An event has arrived; the data value is checked for exceeding the hysteresis.

- *Hysteresis violated:* The value is ignored, returning to **IDLE**.
- *Hysteresis satisfied:* Transition to state **WAIT_TMIN**.

1. **WAIT_TMIN** – The new value is saved; release is delayed by **Tmin**.

- After the time has elapsed, output is made via **Q**, and the data returns to **IDLE**.

A formal state machine is not visible via the XML interface, but it can be deduced from the behavior of the internal flip-flop.

Application Scenarios

- **Sensor Value Smoothing** – A digital sensor (e.g., rotary encoder) delivers values that fluctuate briefly due to vibrations or noise. Hysteresis prevents small fluctuations from being interpreted as valid changes.
- **Button Debouncing** – For mechanical contacts, **Tmin** ensures that bouncing events within a specific time do not result in multiple output events.
- **Measurement Acquisition with Dead Time** – In dosing or positioning systems, the minimum time between two valid measurements must be maintained to avoid overloading the system dynamics.

Comparison with Similar Components

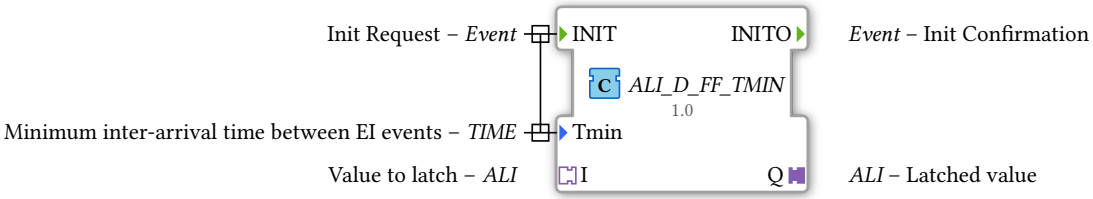
Component	Hysteresis	Minimum Time	Adapter Interface
`E_D_FF` (simple)	No	No	No (direct I/O)
`E_D_FF_HYS`	Yes	No	No
`ALI_D_FF_HYS_TMIN`	Yes	Yes	Yes (adapter usage)

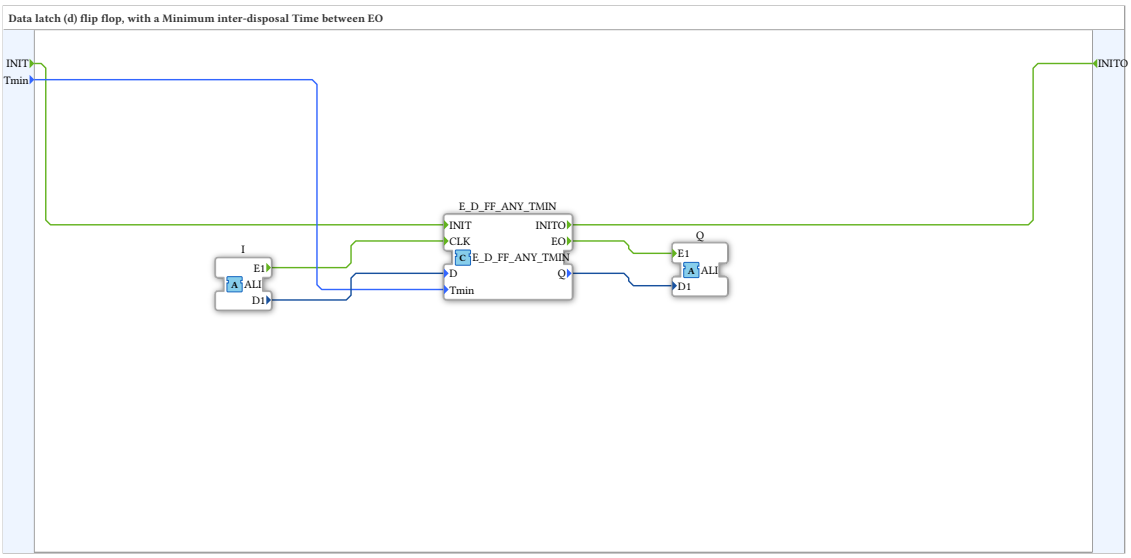
The function block described here combines the advantages of hysteresis and minimum time intervals and, thanks to its adapter interface, is particularly well-suited for a modular, event-driven architecture according to IEC 61499.

Conclusion

The **ALI_D_FF_HYS_TMIN** module provides a robust solution for the stable acquisition of signals in harsh environments. The combination of hysteresis and minimal output time effectively suppresses both amplitude fluctuations and time bounces. Its adapter-based input and output capabilities make it flexible for use in complex automation networks.

ALI_D_FF_TMIN





• – ○ • – ○ • – ○ •

The function block **ALI_D_FF_TMIN** implements a data storage (D flip-flop) with an adapter-based input and output and a configurable minimum dwell time between output events. It serves to accept an incoming data value on a rising edge at the input and release it after a minimum time to avoid cyclical conditions or timing issues in communication.

Event	Type	With Variables	Comment
INIT	EInit	Tmin	Initialization request, sets the minimum dwell time
Event	Type	With Variables	Comment
–––––	––	–––––	–––––
INITO	EInit	–	Initialization Confirmation
Variable	Type	Comment	
–––––	–––	–––––	
Tmin	TIME	Minimum time between two output events (EO)	

No direct data outputs. The latched value is provided via the **Q-Adapter**.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

| Label | Type | Direction | Comment | | I | ALI (unidirectional) | Socket (Input) | Value to be transferred | | Q | ALI (unidirectional) | Plug (Output) | Stored value |

Functionality

The **ALI_D_FF_TMIN** encapsulates an internal function block of type `E_D_FF_ANY_TMIN`. Its functionality:

- Upon an event at the input adapter **I.E1** (rising edge), the data value **I.D1** is transferred.
- However, the output event **EO** is only triggered if at least the time **Tmin** has elapsed since the last **EO**.
- Initialization (INIT) sets the internal timer and prepares the function block.

The function block operates asynchronously; timing is controlled via the internal function block.

- The logic is completely outsourced to the internal function block `E_D_FF_ANY_TMIN`, which has its own state machine and timer.
- The **Tmin** time is set only once during INIT and remains valid for the entire lifetime.
- Adapters **I** (input) and **Q** (output) are unidirectional of type `ALI`; they transmit both events and data.
- The function block does not implement its own top-level behavior but delegates all functions.

Since the function block uses `E_D_FF_ANY_TMIN` internally, it implicitly has the following states:

1. **Initialized**: After INIT, ready for the first edge.
2. **Edge Received**: Value has been stored; the output will be sent after **Tmin**.
3. **Wait for Minimum Time**: No new EO (Event End) is possible until Tmin has expired.
4. **Ready**: After the waiting time has elapsed, the next EO can be triggered.

The exact state machine is located in the internal function block and is not shown here.

- **Data Backup in Time-Controlled Bus Systems**: Buffering of values at a clock signal, which may only be passed on at defined intervals.

- **Signal Debouncing**: Prevents rapid switching on and off from resulting in numerous events at the logic level.
- **Sequential Processing with Minimum Interval**: Ensures that subsequent function blocks have sufficient processing time.

Function Block	Difference
`ALI_D_FF` (without TMIN)	No minimum time between output events, immediate forwarding on every edge.
`E_D_FF` (Standard-61499)	Direct event inputs and outputs, without adapters or timing.
`E_D_FF_ANY_TMIN`	Same functionality, but with dedicated data and event ports, not via adapters.

The **ALI_D_FF_TMIN** extends the adapter interface with convenient encapsulation and adds timing control, which is lacking in simple flip-flops.

The **ALI_D_FF_TMIN** is a useful component for applications that require delayed or timed data transmission via adapters. Its configurable minimum time between output events makes it particularly suitable for time-critical or debounced communication links. The internal implementation using a proven function block ensures robustness and reusability.

Technical Features

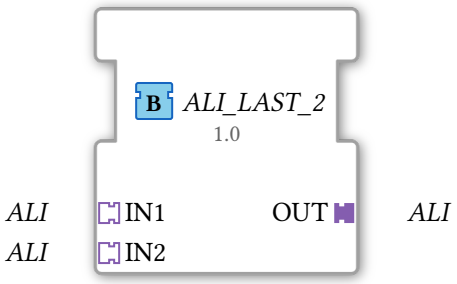
State Overview

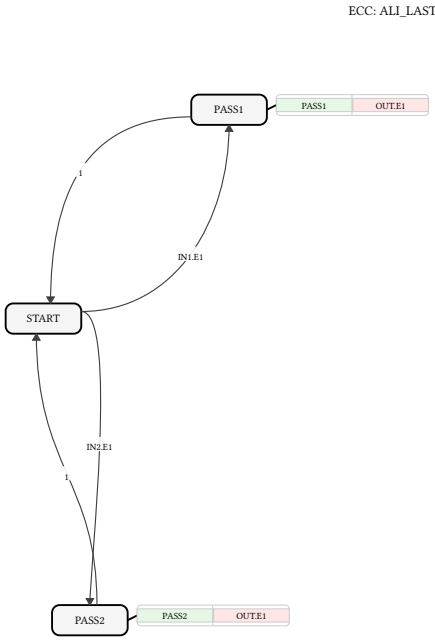
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

ALI_LAST_2





Introduction

The ALI_LAST_2 function block merges two same-type LINT adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: LINT adapter (unidirectional) - first signal source
- **IN2**: LINT adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: LINT adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

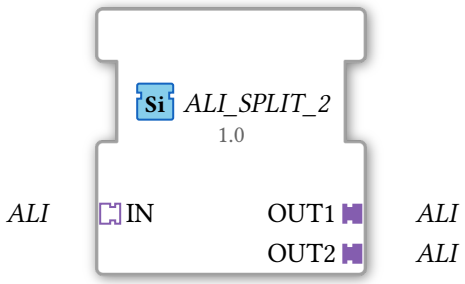
Application Scenarios

- Merging two equally-valid LINT signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same LINT value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

ALI_LAST_2 is the inverse of `ALI_SPLIT_2`: ALI_SPLIT_2 distributes 1 input to 2 outputs, ALI_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

ALI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The **ALI_SPLIT_2** function block serves as a generic distributor for a unidirectional ALI (Application Layer Interface) adapter. It copies all incoming data and events from a single input socket to two output plugs. This allows a signal or data source to be passed on to two independent receivers.

Interface Structure

Event Inputs

No event inputs are available. Data is passed solely via the adapter interface.

Event Outputs

No event outputs are available. Event control is handled via the adapter connections.

Data Inputs

No dedicated data inputs. All data is provided via the incoming adapter **IN**.

Data Outputs

No dedicated data outputs. The distributed data is provided via the two outgoing adapters **OUT1** and **OUT2**.

Adapters

Type	Name	Direction	Description
Socket	`IN`	Input	Unidirectional ALI adapter that receives incoming data/events.
Plug	`OUT1`	Output	First unidirectional ALI adapter to which the data/events are forwarded.
Plug	`OUT2`	Output	Second unidirectional ALI adapter to which the data/events are forwarded.

All adapters are of type `adapter::types::unidirectional::ALI`.

Functionality

The module implements a 1:2 split at the adapter level. As soon as data or events are present at socket **IN**, they are passed on unchanged and simultaneously to both plugs **OUT1** and **OUT2**. No buffering or processing of the data takes place – the module operates purely as a signal distributor.

Technical Features

- **Generic Module**: The attribute `eclipse4diac::core::GenericClassName` marks the module as a generic instance `GEN_ALI_SPLIT`. It can therefore be used with different ALI types or parameters, as long as the underlying adapter definition is compatible.
- **No State Management**: The module has no internal states or variables – its functionality is purely combinatorial.
- **Unidirectionality**: The adapters are defined as unidirectional, meaning data flows only from the input to the outputs; reverse communication is not supported.

State Overview

Since the module has no event inputs/outputs or internal states, there is no state machine. Data transmission is continuous and instantaneous.

Application Scenarios

- **Distributing a Sensor Signal**: An analog or digital ALI sensor is connected to two control units that need to access the measurement data independently.
- **Parallel Processing**: A data stream should be processed simultaneously by two different function blocks without requiring manual branching in the application.
- **Redundancy**: An application requires duplicate ALI communication to increase reliability.

Comparison with Similar Function Blocks

Function Block	Function	Difference
ALI_SPLIT_2	Distributes one ALI input to two ALI outputs.	–
ALI_SPLIT_3 (hypothetical)	Distributes to three outputs.	Higher number of outputs.
ALI_MERGE_2	Combines two ALI inputs into one ALI output.	Opposite function (Join).
ALI_FILTER	Passes on only specific data/events.	Contains filter logic.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

ALI_SPLIT_2 is a simple but essential function block for distributing unidirectional ALI interfaces. Its generic nature and lean implementation make it a universal tool in IEC 61499 component development when a signal or data stream needs to be passed to multiple subsequent blocks.

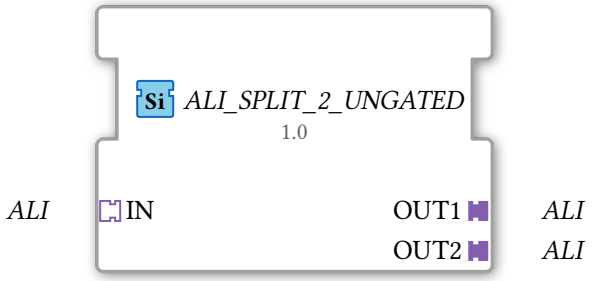
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ALI_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **ALI_SPLIT_2_UNGATED** function block serves as a generic distributor for a unidirectional ALI (Application Layer Interface) adapter. It copies all incoming data and events from a single input socket to two output plugs. This allows a signal or data source to be passed on to two independent receivers.

Interface Structure

Event Inputs

No event inputs are available. Data is passed solely via the adapter interface.

Event Outputs

No event outputs are available. Event control is handled via the adapter connections.

Data Inputs

No dedicated data inputs. All data is provided via the incoming adapter **IN**.

Data Outputs

No dedicated data outputs. The distributed data is provided via the two outgoing adapters **OUT1** and **OUT2**.

Adapters

Type	Name	Direction	Description
Socket	`IN`	Input	Unidirectional ALI adapter that receives incoming data/events.
Plug	`OUT1`	Output	First unidirectional ALI adapter to which the data/events are forwarded.
Plug	`OUT2`	Output	Second unidirectional ALI adapter to which the data/events are forwarded.

All adapters are of type `adapter::types::unidirectional::ALI`.

Functionality

The module implements a 1:2 split at the adapter level. As soon as data or events are present at socket **IN**, they are passed on unchanged and simultaneously to both plugs **OUT1** and **OUT2**. No buffering or processing of the data takes place – the module operates purely as a signal distributor.

Technical Features

- **Generic Module**: The attribute `eclipse4diac::core::GenericClassName` marks the module as a generic instance `GEN_ALI_SPLIT`. It can therefore be used with different ALI types or parameters, as long as the underlying adapter definition is compatible.
- **No State Management**: The module has no internal states or variables – its functionality is purely combinatorial.
- **Unidirectionality**: The adapters are defined as unidirectional, meaning data flows only from the input to the outputs; reverse communication is not supported.

State Overview

Since the module has no event inputs/outputs or internal states, there is no state machine. Data transmission is continuous and instantaneous.

Application Scenarios

- **Distributing a Sensor Signal**: An analog or digital ALI sensor is connected to two control units that need to access the measurement data independently.
- **Parallel Processing**: A data stream should be processed simultaneously by two different function blocks without requiring manual branching in the application.
- **Redundancy**: An application requires duplicate ALI communication to increase reliability.

Comparison with Similar Function Blocks

Function Block	Function	Difference
ALI_SPLIT_2_UNGATED	Distributes one ALI input to two ALI outputs.	–
ALI_SPLIT_3 (hypothetical)	Distributes to three outputs.	Higher number of outputs.
ALI_MERGE_2	Combines two ALI inputs into one ALI output.	Opposite function (Join).
ALI_FILTER	Passes on only specific data/events.	Contains filter logic.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

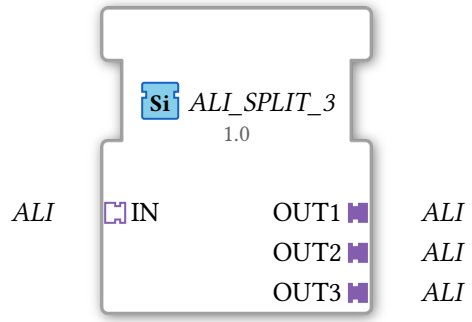
ALI_SPLIT_2_UNGATED is a simple but essential function block for distributing unidirectional ALI interfaces. Its generic nature and lean implementation make it a universal tool in IEC 61499 component development when a signal or data stream needs to be passed to multiple subsequent blocks.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ALI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_SPLIT_3** is used to distribute an incoming unidirectional adapter of type **ALI** (Application Layer Interface) to three identical output adapters of the same type. It is designed as a generic component and enables signal multiplication without data modification.

Interface Structure

Event Inputs
– None –

Event Outputs
– None –

Data Inputs
– None (all communication takes place via the adapter interfaces)

Adapter

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::ALI`	Receives the incoming ALI signal.
Plug (Output)	`OUT1`	`adapter::types::unidirectional::ALI`	First output copy of the signal.
Plug (Output)	`OUT2`	`adapter::types::unidirectional::ALI`	Second output copy of the signal.
Plug (Output)	`OUT3`	`adapter::types::unidirectional::ALI`	Third output copy of the signal.

Note: The adapter type `ALI` is a unidirectional interface adapter that typically encapsulates data and event flow in one direction. The exact internal structure is defined application-specifically.

Functionality
The function block performs a pure 1:3 distribution of the ALI adapter connected to socket `IN`. All information received at the input (e.g., data values, events, or states) is forwarded in parallel to the three outputs `OUT1`, `OUT2`, and `OUT3` without delay or transformation. No buffering or logic takes place – the function block operates purely synchronously with the data flow.

Technical Features
- **Generic Type:** The function block is declared as a generic function block (`eclipse4diac::core::GenericClassName = 'GEN_ALI_SPLIT'`). This allows the actual adapter type to be defined at configuration time, provided the underlying system supports this.
- **No Internal Dynamics:** The function block has no event or data inputs/outputs outside of the adapters. All communication takes place exclusively via the adapter interfaces.
- **Low Latency:** By eliminating internal processing, signal distribution is virtually delay-free.

State Overview
The function block has no state machines or state memory. Its behavior is deterministic and purely combinatorial: The current input state is immediately mapped to all outputs.

Application Scenarios
- **Signal Multiplication in Control Systems:** If an ALI signal (e.g., a sensor data stream) is required multiple times by different downstream function blocks, ALI_SPLIT_3 can handle the distribution.
- **Test and Simulation Environments:** Multiple components are supplied with the same input signal to observe parallel responses.
- **Redundancy:** A signal is split across three paths, which are processed independently (e.g., for comparison or fault detection).

Comparison with Similar Function Blocks
- **ALI_SPLIT_2:** Distributes an ALI signal to two outputs instead of three. ALI_SPLIT_3 extends this number to three.
- **ALI_MERGE (hypothetical):** Combines multiple ALI inputs into one output – functionally the opposite.
- **Event-Based Splitters (e.g., E_SPLIT):** Work with pure events, while ALI_SPLIT_3 distributes data and event components together via a single adapter.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

ALI_SPLIT_3 is a simple yet useful function block for multiplying ALI adapter connections. Its generic nature and pure signal transmission make it a flexible tool in automation and control engineering, especially when a signal is needed multiple times.

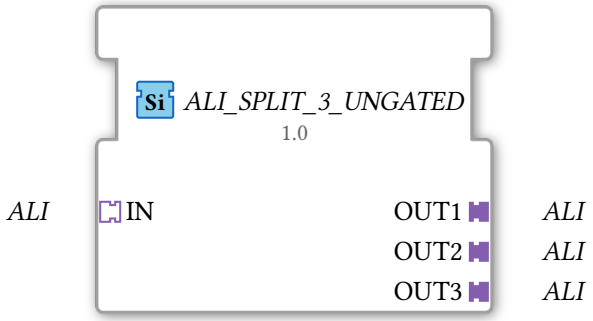
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALI_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_SPLIT_3_UNGATED** is used to distribute an incoming unidirectional adapter of type **ALI** (Application Layer Interface) to three identical output adapters of the same type. It is designed as a generic component and enables signal multiplication without data modification.

Interface Structure

Event Inputs

– None –

Event Outputs

– None –

Data Inputs

– None (all communication takes place via the adapter interfaces)

Adapter

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::ALI`	Receives the incoming ALI signal.

Plug (Output)	`OUT1`	`adapter::types::unidirectional::ALI`	First output copy of the signal.
Plug (Output)	`OUT2`	`adapter::types::unidirectional::ALI`	Second output copy of the signal.
Plug (Output)	`OUT3`	`adapter::types::unidirectional::ALI`	Third output copy of the signal.

Note: The adapter type `ALI` is a unidirectional interface adapter that typically encapsulates data and event flow in one direction. The exact internal structure is defined application-specifically.

Functionality

The function block performs a pure 1:3 distribution of the ALI adapter connected to socket `IN`. All information received at the input (e.g., data values, events, or states) is forwarded in parallel to the three outputs `OUT1`, `OUT2`, and `OUT3` without delay or transformation. No buffering or logic takes place – the function block operates purely synchronously with the data flow.

Technical Features

- **Generic Type:** The function block is declared as a generic function block (`eclipse4diac::core::GenericClassName = 'GEN_ALI_SPLIT'`). This allows the actual adapter type to be defined at configuration time, provided the underlying system supports this.
- **No Internal Dynamics:** The function block has no event or data inputs/outputs outside of the adapters. All communication takes place exclusively via the adapter interfaces.
- **Low Latency:** By eliminating internal processing, signal distribution is virtually delay-free.

State Overview

The function block has no state machines or state memory. Its behavior is deterministic and purely combinatorial: The current input state is immediately mapped to all outputs.

Application Scenarios

- **Signal Multiplication in Control Systems:** If an ALI signal (e.g., a sensor data stream) is required multiple times by different downstream function blocks, ALI_SPLIT_3_UNGATED can handle the distribution.
- **Test and Simulation Environments:** Multiple components are supplied with the same input signal to observe parallel responses.
- **Redundancy:** A signal is split across three paths, which are processed independently (e.g., for comparison or fault detection).

Comparison with Similar Function Blocks

- **ALI_SPLIT_2:** Distributes an ALI signal to two outputs instead of three. ALI_SPLIT_3_UNGATED extends this number to three.
- **ALI_MERGE (hypothetical):** Combines multiple ALI inputs into one output – functionally the opposite.
- **Event-Based Splitters (e.g., E_SPLIT):** Work with pure events, while ALI_SPLIT_3_UNGATED distributes data and event components together via a single adapter.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

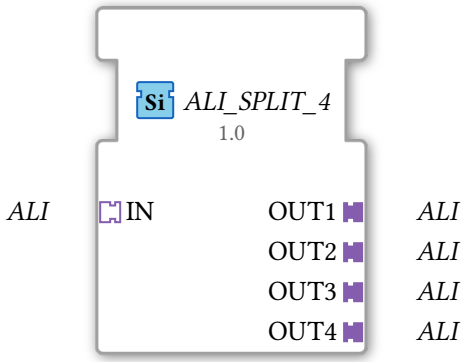
ALI_SPLIT_3_UNGATED is a simple yet useful function block for multiplying ALI adapter connections. Its generic nature and pure signal transmission make it a flexible tool in automation and control engineering, especially when a signal is needed multiple times.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_SPLIT_4** is used to distribute a single ALI adapter input (type: `adapter::types::unidirectional::ALI`) to four separate ALI adapter outputs. It is designed as a generic function block (FB) and enables unidirectional signal transmission from one source to up to four destinations.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Direction	Name	Type	Description
Socket	IN	adapter::types::unidirectional::ALI	Unidirectional ALI Input (Source)
Plug	OUT1	adapter::types::unidirectional::ALI	First ALI Output (Destination)
Plug	OUT2	adapter::types::unidirectional::ALI	Second ALI Output (Destination)
Plug	OUT3	adapter::types::unidirectional::ALI	Third ALI Output (Destination)
Plug	OUT4	adapter::types::unidirectional::ALI	Fourth ALI Output (Destination)

Functionality

The function block receives an ALI adapter connection via socket **IN** and forwards the incoming signals unchanged to all four plugs **OUT1** to **OUT4**. No data processing, filtering, or logical operation takes place – the distribution is purely passive. Due to its generic implementation, the function block can be used in various contexts where an ALI signal is required multiple times.

Technical Features

- The function block is declared as a generic function block (`GEN_ALI_SPLIT`), which allows for flexible type customization in the 4diac IDE development environment.
- No events or data points are required – all communication occurs exclusively via the adapter interfaces.
- The function block is unidirectional: feedback from the outputs to the input is not provided.

- The attributes `GenericClassName` and `TypeHash` serve for the unique identification and versioning of the generic type.

State Overview

The function block has no states or control logic of its own. Its behavior is deterministic and consists solely of forwarding the adapter signal. Therefore, a separate state machine is not required.

Application Scenarios

- **Signal Distribution**: A central ALI data stream (e.g., from a sensor or control unit) is to be sent in parallel to several downstream components.
- **Prototyping**: In early development phases, the function block can be used to quickly distribute a signal to multiple test units.
- **Redundancy**: Multiple identical outputs allow for the connection of backup systems or separate monitoring units.

Comparison with Similar Function Blocks

- **ALI_SPLIT_2**: Distributes the input signal to two outputs instead of four. The ALI_SPLIT_4 offers twice the number of output channels.
- **ALI_MERGE**: Combines multiple ALI inputs into one output – the inverse function of the split function.
- **ALI_SELECT**: Selects one of several inputs based on a control signal and routes it to an output (no parallel distribution).

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ALI_SPLIT_4** is a simple yet useful function block for passively distributing a unidirectional ALI signal to four identical outputs. Its generic design and the absence of event or data logic make it particularly suitable for rapid signal multiplication in automation technology without incurring additional processing load.

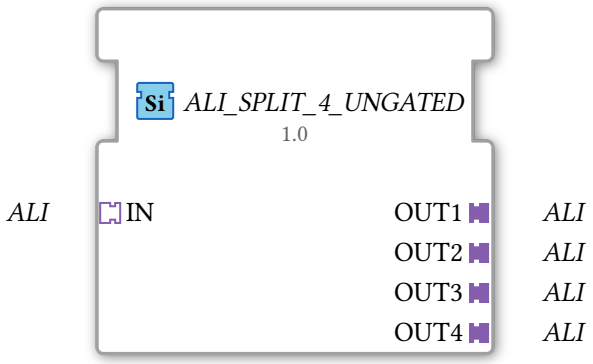
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALI_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_SPLIT_4_UNGATED** is used to distribute a single ALI adapter input (type: `adapter::types::unidirectional::ALI`) to four separate ALI adapter outputs. It is designed as a generic function block (FB) and enables unidirectional signal transmission from one source to up to four destinations.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Direction	Name	Type	Description
Socket	IN	adapter::types::unidirectional::ALI	Unidirectional ALI Input (Source)
Plug	OUT1	adapter::types::unidirectional::ALI	First ALI Output (Destination)
Plug	OUT2	adapter::types::unidirectional::ALI	Second ALI Output (Destination)
Plug	OUT3	adapter::types::unidirectional::ALI	Third ALI Output (Destination)
Plug	OUT4	adapter::types::unidirectional::ALI	Fourth ALI Output (Destination)

Functionality

The function block receives an ALI adapter connection via socket **IN** and forwards the incoming signals unchanged to all four plugs **OUT1** to **OUT4**. No data processing, filtering, or logical operation takes place – the distribution is purely passive. Due to its generic implementation, the function block can be used in various contexts where an ALI signal is required multiple times.

Technical Features

- The function block is declared as a generic function block (`GEN_ALI_SPLIT`), which allows for flexible type customization in the 4diac IDE development environment.
- No events or data points are required – all communication occurs exclusively via the adapter interfaces.
- The function block is unidirectional: feedback from the outputs to the input is not provided.

- The attributes `GenericClassName` and `TypeHash` serve for the unique identification and versioning of the generic type.

State Overview

The function block has no states or control logic of its own. Its behavior is deterministic and consists solely of forwarding the adapter signal. Therefore, a separate state machine is not required.

Application Scenarios

- **Signal Distribution**: A central ALI data stream (e.g., from a sensor or control unit) is to be sent in parallel to several downstream components.
- **Prototyping**: In early development phases, the function block can be used to quickly distribute a signal to multiple test units.
- **Redundancy**: Multiple identical outputs allow for the connection of backup systems or separate monitoring units.

Comparison with Similar Function Blocks

- **ALI_SPLIT_2**: Distributes the input signal to two outputs instead of four. The ALI_SPLIT_4_UNGATED offers twice the number of output channels.
- **ALI_MERGE**: Combines multiple ALI inputs into one output – the inverse function of the split function.
- **ALI_SELECT**: Selects one of several inputs based on a control signal and routes it to an output (no parallel distribution).

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

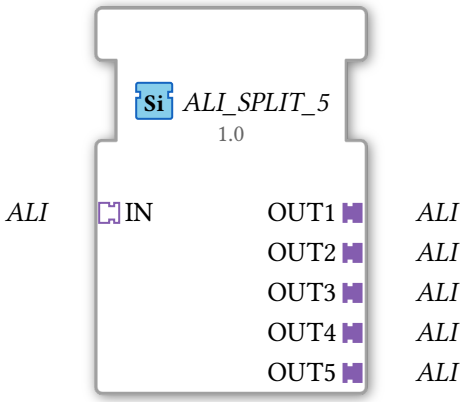
The **ALI_SPLIT_4_UNGATED** is a simple yet useful function block for passively distributing a unidirectional ALI signal to four identical outputs. Its generic design and the absence of event or data logic make it particularly suitable for rapid signal multiplication in automation technology without incurring additional processing load.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_SPLIT_5` is a generic adapter split block. It accepts a single incoming ALI adapter (`adapter::types::unidirectional::ALI`) and provides the same ALI signals at five outgoing

ALI adapter interfaces. This allows an ALI connection to be distributed to multiple subsequent blocks or devices without interrupting the logical signal chain.

Interface Structure

Event Inputs
- None

Event Outputs
- None

Data Inputs
- None

Data Outputs
- None

Adapters

Direction	Name	Type	Description
Input (Socket)	`IN`	`adapter::types::unidirectional::ALI`	Single input: contains the ALI data to be distributed to the five outputs.
Output (Plug)	`OUT1`	`adapter::types::unidirectional::ALI`	First output (identical signals as at the input).
Output (Plug)	`OUT2`	`adapter::types::unidirectional::ALI`	Second output.
Output (Plug)	`OUT3`	`adapter::types::unidirectional::ALI`	Third output.
Output (Plug)	`OUT4`	`adapter::types::unidirectional::ALI`	Fourth output.
Output (Plug)	`OUT5`	`adapter::types::unidirectional::ALI`	Fifth output.

Functionality

As soon as the module is connected to socket `IN` via a valid ALI connection, all data and event traffic from the incoming adapter is forwarded **unchanged** and **in parallel** to the five plug adapters `OUT1` to `OUT5`. No logical processing, filtering, or delay takes place – the function block acts as a passive distribution unit ("splitter") for the ALI channel.

The forwarding is bidirectional according to the ALI adapter definition: Both events and data (as defined in the ALI type) are made available synchronously at all outputs. The function block has no internal runtime logic or state.

Technical Features
- **Generic Function Block** – The specific implementation is controlled by the meta attributes `GenericClassName` and `TypeHash`. This allows the same function block definition to be reused for different ALI variants (e.g., with different data structures).
- **No Delay** – The signals are replicated to the outputs without any significant delay.
- **Easily Expandable** – The principle of this splitter can be transferred to other adapter types; the generic architecture eliminates the need to modify the basic logic.
- **Direct Coupling** – The output state always corresponds to the current input state; there is no buffering or intermediate storage.

State Overview

The component has **no internal state machine**. It operates directly and without delay as a pure connection switch ("wired split"). State changes occur only through the connected adapter partners, and these are passed on transparently.

Application Scenarios
- **Distributing a sensor signal** to multiple evaluation units or controllers that require the same ALI data stream.

- **Parallelizing ALI communication paths** in a machine controller to supply redundant or independent processing chains.
- **Test and debugging environments** where an ALI signal needs to be simultaneously monitored and recorded.

Comparison with similar components

Other split or fanout components usually offer a fixed number of outputs or are restricted to specific data types. The `ALI_SPLIT_5` is specifically designed for the unidirectional ALI adapter, but its generic template allows for flexible application. Unlike a **multiplexer** or **router**, this component always forwards the signal to all outputs **without switching**.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `ALI_SPLIT_5` is a simple yet essential component for multiplying an ALI signal. Due to its generic nature, it can be easily integrated into various 4diac projects and simplifies signal distribution in complex automation architectures. It requires little maintenance because it contains no internal logic and contributes to the modularization and reusability of ALI connections.

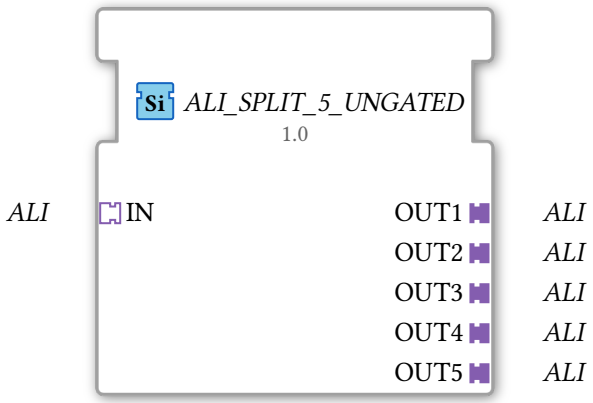
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALI_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_SPLIT_5_UNGATED` is a generic adapter split block. It accepts a single incoming ALI adapter (`adapter::types::unidirectional::ALI`) and provides the same ALI signals at five outgoing ALI adapter interfaces. This allows an ALI connection to be distributed to multiple subsequent blocks or devices without interrupting the logical signal chain.

Interface Structure

Event Inputs
- None

Event Outputs
- None

Data Inputs
- None

Data Outputs
- None

Adapters

Direction	Name	Type	Description
Input (Socket)	`IN`	`adapter::types::unidirectional::ALI`	Single input: contains the ALI data to be distributed to the five outputs.
Output (Plug)	`OUT1`	`adapter::types::unidirectional::ALI`	First output (identical signals as at the input).
Output (Plug)	`OUT2`	`adapter::types::unidirectional::ALI`	Second output.
Output (Plug)	`OUT3`	`adapter::types::unidirectional::ALI`	Third output.
Output (Plug)	`OUT4`	`adapter::types::unidirectional::ALI`	Fourth output.
Output (Plug)	`OUT5`	`adapter::types::unidirectional::ALI`	Fifth output.

Functionality
As soon as the module is connected to socket `IN` via a valid ALI connection, all data and event traffic from the incoming adapter is forwarded **unchanged** and **in parallel** to the five plug adapters `OUT1` to `OUT5`. No logical processing, filtering, or delay takes place – the function block acts as a passive distribution unit ("splitter") for the ALI channel.

The forwarding is bidirectional according to the ALI adapter definition: Both events and data (as defined in the ALI type) are made available synchronously at all outputs. The function block has no internal runtime logic or state.

Technical Features
- **Generic Function Block** – The specific implementation is controlled by the meta attributes `GenericClassName` and `TypeHash`. This allows the same function block definition to be reused for different ALI variants (e.g., with different data structures).
- **No Delay** – The signals are replicated to the outputs without any significant delay.
- **Easily Expandable** – The principle of this splitter can be transferred to other adapter types; the generic architecture eliminates the need to modify the basic logic.
- **Direct Coupling** – The output state always corresponds to the current input state; there is no buffering or intermediate storage.

State Overview
The component has **no internal state machine**. It operates directly and without delay as a pure connection switch ("wired split"). State changes occur only through the connected adapter partners, and these are passed on transparently.

Application Scenarios
- **Distributing a sensor signal** to multiple evaluation units or controllers that require the same ALI data stream.
- **Parallelizing ALI communication paths** in a machine controller to supply redundant or independent processing chains.

- **Test and debugging environments** where an ALI signal needs to be simultaneously monitored and recorded.

Comparison with similar components

Other split or fanout components usually offer a fixed number of outputs or are restricted to specific data types. The `ALI_SPLIT_5_UNGATED` is specifically designed for the unidirectional ALI adapter, but its generic template allows for flexible application. Unlike a **multiplexer** or **router**, this component always forwards the signal to all outputs **without switching**.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

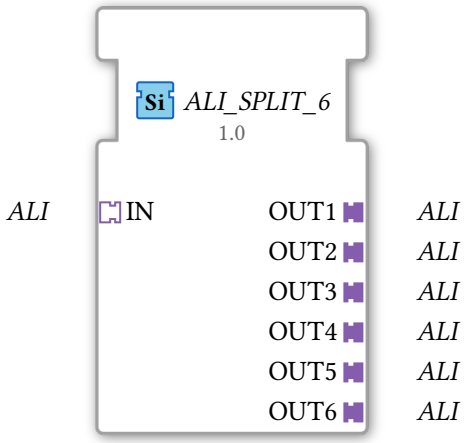
The `ALI_SPLIT_5_UNGATED` is a simple yet essential component for multiplying an ALI signal. Due to its generic nature, it can be easily integrated into various 4diac projects and simplifies signal distribution in complex automation architectures. It requires little maintenance because it contains no internal logic and contributes to the modularization and reusability of ALI connections.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALI_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The ALI_SPLIT_6 function block is used to distribute an incoming unidirectional ALI (Application Layer Interface) adapter signal to six identical outputs. It is designed as a generic function block and enables simple signal multiplication without additional logic.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Type	Name	Direction	Description
adapter::types::unidirectional::ALI	IN	Socket (Input)	An ALI adapter whose signal is distributed to all outputs.
adapter::types::unidirectional::ALI	OUT1	Plug (Output)	First output carrying the input signal.
adapter::types::unidirectional::ALI	OUT2	Plug (Output)	Second output carrying the input signal.
adapter::types::unidirectional::ALI	OUT3	Plug (Output)	Third output carrying the input signal.
adapter::types::unidirectional::ALI	OUT4	Plug (Output)	Fourth output with the input signal.
adapter::types::unidirectional::ALI	OUT5	Plug (Output)	Fifth output with the input signal.
adapter::types::unidirectional::ALI	OUT6	Plug (Output)	Sixth output with the input signal.

Functionality

The ALI_SPLIT_6 implements a simple 1:6 distribution. The ALI adapter signal received via socket `IN` is forwarded unchanged and simultaneously to all six plugs (`OUT1` to `OUT6`). No signal processing, delay, or state changes occur. The distribution is purely structural via the adapter connections.

Technical Features

- **Generic Function Block:** The FB is defined as a generic function block. The specific class name (e.g., `eclipse4diac::core::GenericClassName`) can be defined via the attribute `'GEN_ALI_SPLIT'`. The attribute `eclipse4diac::core::TypeHash` can be used to identify the specific configuration.
- **No State Machines:** The FB has no ECC (Execution Control Chart) and no internal logic. It operates purely passively.
- **Unidirectional ALI Adapter Interface:** Both inputs and outputs use the type `adapter::types::unidirectional::ALI`, which enables simple and standardized communication.

State Overview

The FB does not have its own states or behavior modes, as it only performs structural routing. Therefore, state analysis is not required.

Application Scenarios

- **Controlling Multiple Actuators:** An ALI signal, e.g., a enable or control signal, is to be simultaneously transmitted to six receivers (e.g., servo motors or valve manifolds).
- **Signal Distribution in Modular Systems:** In an industrial control system, a central ALI signal can be distributed to multiple decentralized units.
- **Replacing Manual Wiring:** The FB replaces the physical distribution of a signal with a software-defined, maintainable solution.

Comparison with Similar Function Blocks

Function Block	Outputs	Special Features
ALI_SPLIT_2	2	Dual Distribution
ALI_SPLIT_4	4	Quadruple Distribution
ALI_SPLIT_6	**6**	**Six-Way Distributor, Generic**
ALI_SPLIT_8	8	Eight-Way Distributor

The ALI_SPLIT_6 differs from other split variants only in the number of outputs. All function blocks in this family operate on the same passive distribution principle. The function block described here is implemented as a generic block, while other variants may be implemented as simple function block types without genericity.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

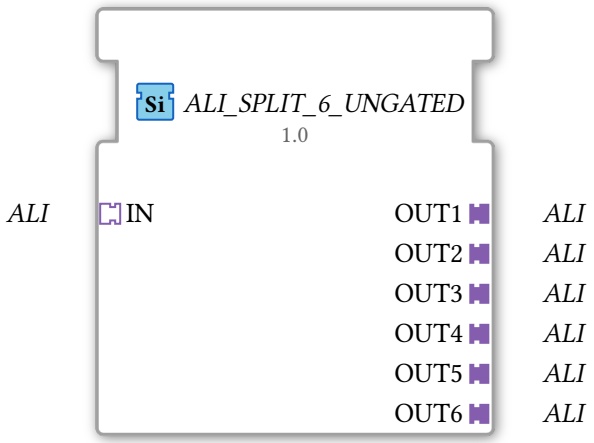
The ALI_SPLIT_6 is a simple yet useful function block for multiplying a unidirectional ALI adapter signal to six identical outputs. Thanks to its generic properties, it can be flexibly integrated into various control systems. Due to the lack of processing logic, it is particularly resource-efficient and easy to understand. It is ideally suited for applications where a signal needs to be provided to multiple receivers simultaneously.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

ALI_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The ALI_SPLIT_6_UNGATED function block is used to distribute an incoming unidirectional ALI (Application Layer Interface) adapter signal to six identical outputs. It is designed as a generic function block and enables simple signal multiplication without additional logic.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Type	Name	Direction	Description
adapter::types::unidirectional::ALI	IN	Socket (Input)	An ALI adapter whose signal is distributed to all outputs.
adapter::types::unidirectional::ALI	OUT1	Plug (Output)	First output carrying the input signal.
adapter::types::unidirectional::ALI	OUT2	Plug (Output)	Second output carrying the input signal.
adapter::types::unidirectional::ALI	OUT3	Plug (Output)	Third output carrying the input signal.
adapter::types::unidirectional::ALI	OUT4	Plug (Output)	Fourth output with the input signal.
adapter::types::unidirectional::ALI	OUT5	Plug (Output)	Fifth output with the input signal.
adapter::types::unidirectional::ALI	OUT6	Plug (Output)	Sixth output with the input signal.

Functionality

The ALI_SPLIT_6_UNGATED implements a simple 1:6 distribution. The ALI adapter signal received via socket `IN` is forwarded unchanged and simultaneously to all six plugs (`OUT1` to `OUT6`). No signal processing, delay, or state changes occur. The distribution is purely structural via the adapter connections.

Technical Features

- **Generic Function Block:** The FB is defined as a generic function block. The specific class name (e.g., `eclipse4diac::core::GenericClassName`) can be defined via the attribute `'GEN_ALI_SPLIT'`. The attribute `eclipse4diac::core::TypeHash` can be used to identify the specific configuration.
- **No State Machines:** The FB has no ECC (Execution Control Chart) and no internal logic. It operates purely passively.
- **Unidirectional ALI Adapter Interface:** Both inputs and outputs use the type `adapter::types::unidirectional::ALI`, which enables simple and standardized communication.

State Overview

The FB does not have its own states or behavior modes, as it only performs structural routing. Therefore, state analysis is not required.

Application Scenarios

- **Controlling Multiple Actuators:** An ALI signal, e.g., a enable or control signal, is to be simultaneously transmitted to six receivers (e.g., servo motors or valve manifolds).
- **Signal Distribution in Modular Systems:** In an industrial control system, a central ALI signal can be distributed to multiple decentralized units.
- **Replacing Manual Wiring:** The FB replaces the physical distribution of a signal with a software-defined, maintainable solution.

Comparison with Similar Function Blocks

Function Block	Outputs	Special Features
ALI_SPLIT_2	2	Dual Distribution
ALI_SPLIT_4	4	Quadruple Distribution
ALI_SPLIT_6_UNGATED	**6**	**Six-Way Distributor, Generic**

ALI_SPLIT_8	8	Eight-Way Distributor

The ALI_SPLIT_6_UNGATED differs from other split variants only in the number of outputs. All function blocks in this family operate on the same passive distribution principle. The function block described here is implemented as a generic block, while other variants may be implemented as simple function block types without genericity.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

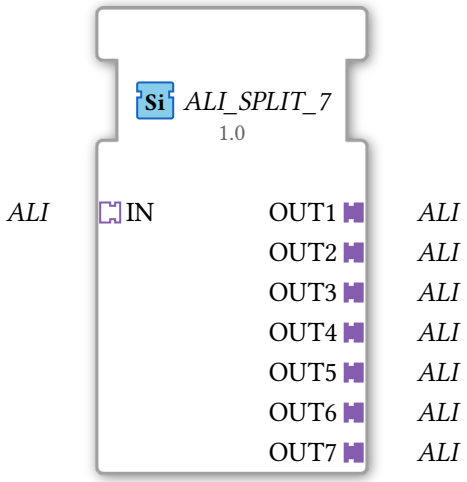
Conclusion

The ALI_SPLIT_6_UNGATED is a simple yet useful function block for multiplying a unidirectional ALI adapter signal to six identical outputs. Thanks to its generic properties, it can be flexibly integrated into various control systems. Due to the lack of processing logic, it is particularly resource-efficient and easy to understand. It is ideally suited for applications where a signal needs to be provided to multiple receivers simultaneously.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

ALI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_SPLIT_7** is a generic splitter that distributes one incoming ALI adapter (socket `IN`) to seven outgoing ALI adapters (plugs `OUT1`…`OUT7`). It is used for a 1:7 distribution of data and events within a unidirectional ALI communication path.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::ALI	Input adapter whose data and events are distributed.
Plug (Output)	OUT1	adapter::types::unidirectional::ALI	First output adapter.
Plug (Output)	OUT2	adapter::types::unidirectional::ALI	Second output adapter.
Plug (Output)	OUT3	adapter::types::unidirectional::ALI	Third output adapter.
Plug (Output)	OUT4	adapter::types::unidirectional::ALI	Fourth Output Adapter.
Plug (Output)	OUT5	adapter::types::unidirectional::ALI	Fifth Output Adapter.
Plug (Output)	OUT6	adapter::types::unidirectional::ALI	Sixth Output Adapter.
Plug (Output)	OUT7	adapter::types::unidirectional::ALI	Seventh Output Adapter.

Functionality

The function block forwards all incoming ALI signals (both data and events) via socket `IN` unchanged to all seven output adapters `OUT1`…`OUT7`. This results in a strict 1:7 parallel distribution without delay or buffering. The function block behaves purely combinatorially; no processing or filtering of the contents takes place.

Technical Features

- **Generic Type**: The function block is declared as a generic function block (`GenericClassName = 'GEN_ALI_SPLIT'`). This allows it to be used with different ALI adapter variants, provided they support the same interface protocol (unidirectional adapters).
- **No States**: The function block does not have a state machine (ECC) and does not store any data. Each iteration is deterministic and has no side effects.
- **Flexible Distribution**: With its fixed number of seven outputs, the FB is particularly suitable for applications requiring distribution across exactly seven paths.

State Overview

The FB has **no explicit states**. Its functionality is purely passive and data flow-driven.

Application Scenarios

- **Parallel Signal Distribution**: An ALI data stream sent by a higher-level control module is to be simultaneously passed on to several downstream modules (e.g., actuators, visualizations, or logging systems).
- **Redundancy or Load Balancing**: Several identical processing units receive the same ALI input data and operate independently.
- **Adapter Conversion**: In combination with specialized ALI adapters, the splitter can be used to multiply a data path across different logic or hardware interfaces.

Comparison with Similar Function Blocks

- **ALI_SPLIT_2 / ALI_SPLIT_4**: These function blocks offer splits to two or four outputs, respectively. `ALI_SPLIT_7` is the extension to seven outputs for specific requirements.
- **ALI_MERGE**: The counterpart that combines multiple ALI inputs into one output. A splitter like `ALI_SPLIT_7` operates in the opposite direction (fan-out).

Change Detection

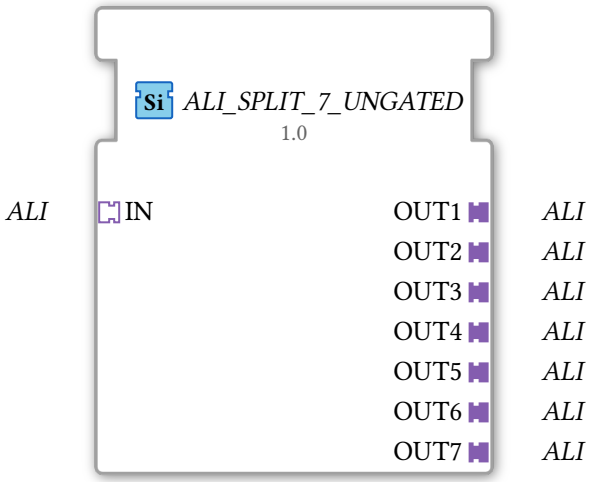
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ALI_SPLIT_7** is a specialized, generic function block for the clean and lossless 1:7 distribution of unidirectional ALI signals. Its simple structure, lack of state logic, and generic design make it ideal for modular control architectures where a signal needs to be passed to multiple receivers in parallel.

ALI_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_SPLIT_7_UNGATED** is a generic splitter that distributes one incoming ALI adapter (socket `IN`) to seven outgoing ALI adapters (plugs `OUT1`...`OUT7`). It is used for a 1:7 distribution of data and events within a unidirectional ALI communication path.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::ALI`	Input adapter whose data and events are distributed.
Plug (Output)	`OUT1`	`adapter::types::unidirectional::ALI`	First output adapter.

Plug (Output)	OUT2	adapter::types::unidirectional::ALI	Second output adapter.
Plug (Output)	OUT3	adapter::types::unidirectional::ALI	Third output adapter.
Plug (Output)	OUT4	adapter::types::unidirectional::ALI	Fourth Output Adapter.
Plug (Output)	OUT5	adapter::types::unidirectional::ALI	Fifth Output Adapter.
Plug (Output)	OUT6	adapter::types::unidirectional::ALI	Sixth Output Adapter.
Plug (Output)	OUT7	adapter::types::unidirectional::ALI	Seventh Output Adapter.

Functionality

The function block forwards all incoming ALI signals (both data and events) via socket `IN` unchanged to all seven output adapters `OUT1`...`OUT7`. This results in a strict 1:7 parallel distribution without delay or buffering. The function block behaves purely combinatorially; no processing or filtering of the contents takes place.

Technical Features

- **Generic Type**: The function block is declared as a generic function block (`GenericClassName = 'GEN_ALI_SPLIT'`). This allows it to be used with different ALI adapter variants, provided they support the same interface protocol (unidirectional adapters).
- **No States**: The function block does not have a state machine (ECC) and does not store any data. Each iteration is deterministic and has no side effects.
- **Flexible Distribution**: With its fixed number of seven outputs, the FB is particularly suitable for applications requiring distribution across exactly seven paths.

State Overview

The FB has **no explicit states**. Its functionality is purely passive and data flow-driven.

Application Scenarios

- **Parallel Signal Distribution**: An ALI data stream sent by a higher-level control module is to be simultaneously passed on to several downstream modules (e.g., actuators, visualizations, or logging systems).
- **Redundancy or Load Balancing**: Several identical processing units receive the same ALI input data and operate independently.
- **Adapter Conversion**: In combination with specialized ALI adapters, the splitter can be used to multiply a data path across different logic or hardware interfaces.

Comparison with Similar Function Blocks

- **ALI_SPLIT_2 / ALI_SPLIT_4**: These function blocks offer splits to two or four outputs, respectively. `ALI_SPLIT_7_UNGATED` is the extension to seven outputs for specific requirements.
- **ALI_MERGE**: The counterpart that combines multiple ALI inputs into one output. A splitter like `ALI_SPLIT_7_UNGATED` operates in the opposite direction (fan-out).

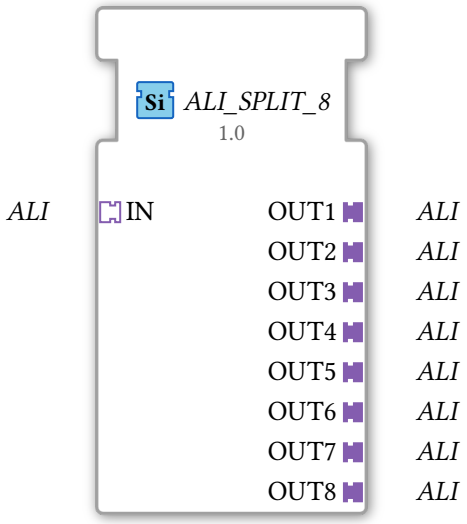
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **ALI_SPLIT_7_UNGATED** is a specialized, generic function block for the clean and lossless 1:7 distribution of unidirectional ALI signals. Its simple structure, lack of state logic, and generic design make it ideal for modular control architectures where a signal needs to be passed to multiple receivers in parallel.

ALI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_SPLIT_8` serves as a generic splitter for the **Agricultural Light Interface (ALI)**. It distributes an incoming ALI signal unchanged to eight parallel outputs. This allows multiple downstream devices or controllers to be supplied with the same signal without compromising signal integrity.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

- **Socket (Input):**

IN– `of type`adapter::types::unidirectional::ALI` Receives the incoming ALI signal.

- **Plugs (Outputs):**

OUT1…OUT8– `each of type`adapter::types::unidirectional::ALI` Provide the eight identical copies of the input signal.

Functionality

The module forwards the ALI signal present at socket `IN` unchanged and without delay to all eight plugs (`OUT1` … `OUT8`). Since this is purely signal distribution, no processing, buffering, or status changes occur. All events and data transmitted via the ALI adapter are replicated to each output.

Technical Features

- **Generic Function Block:** The function block is defined as a generic type (`GEN_ALI_SPLIT`) and can be used in various contexts within the ALI ecosystem.
- **No ECC:** There is no execution state machine (ECC); distribution is purely combinatorial.

- **Unidirectional:** The adapter type is unidirectional – the signal flows only from the socket to the plugs; feedback is not provided.
- **Scalability:** Due to its modular design, similar splitters for other output numbers (e.g., `ALI_SPLIT_2`) can be derived.

State Overview

The function block has no internal states or sequential logic. Signal distribution operates continuously and without delay.

Application Scenarios

- **Distribution Circuits:** An ALI signal (e.g., from a control unit) must be simultaneously transmitted to multiple actuators or sensors.
- **Redundancy:** Parallel control of identical components without requiring separate signal sources.
- **Bus Structures:** Construction of star-shaped ALI networks with a central splitter.

Comparison with Similar Components

Other splitter components such as `ALI_SPLIT_2`, `ALI_SPLIT_4`, or `ALI_SPLIT_N` differ only in the number of outputs. The `ALI_SPLIT_8` offers the maximum distribution in the standard family. Unlike a multiplexer (`ALI_MUX`) or a switch (`ALI_SWITCH`), here **each output is supplied with the same signal** – no selection or switching takes place.

Change Detection

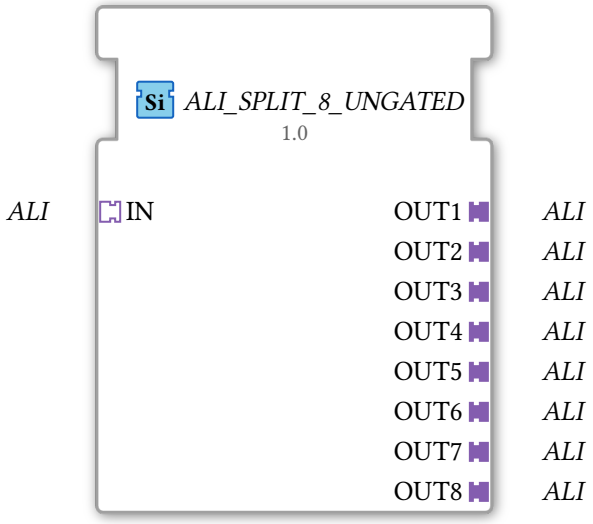
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `ALI_SPLIT_8` is a simple yet essential component for multiplying ALI signals. Its generic definition and the absence of state logic make it particularly suitable for robust, low-latency distributions in agricultural automation.

ALI_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_SPLIT_8_UNGATED` serves as a generic splitter for the **Agricultural Light Interface (ALI)**. It distributes an incoming ALI signal unchanged to eight parallel outputs. This allows multiple downstream devices or controllers to be supplied with the same signal without compromising signal integrity.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

- **Socket (Input):**

IN– `of type`adapter::types::unidirectional::ALI` Receives the incoming ALI signal.

- **Plugs (Outputs):**

OUT1…OUT8– `each of type`adapter::types::unidirectional::ALI` Provide the eight identical copies of the input signal.

Functionality

The module forwards the ALI signal present at socket `IN` unchanged and without delay to all eight plugs (`OUT1` … `OUT8`). Since this is purely signal distribution, no processing, buffering, or status changes occur. All events and data transmitted via the ALI adapter are replicated to each output.

Technical Features

- **Generic Function Block:** The function block is defined as a generic type (`GEN_ALI_SPLIT`) and can be used in various contexts within the ALI ecosystem.
- **No ECC:** There is no execution state machine (ECC); distribution is purely combinatorial.
- **Unidirectional:** The adapter type is unidirectional – the signal flows only from the socket to the plugs; feedback is not provided.
- **Scalability:** Due to its modular design, similar splitters for other output numbers (e.g., `ALI_SPLIT_2`) can be derived.

State Overview

The function block has no internal states or sequential logic. Signal distribution operates continuously and without delay.

Application Scenarios

- **Distribution Circuits:** An ALI signal (e.g., from a control unit) must be simultaneously transmitted to multiple actuators or sensors.
- **Redundancy:** Parallel control of identical components without requiring separate signal sources.
- **Bus Structures:** Construction of star-shaped ALI networks with a central splitter.

Comparison with Similar Components

Other splitter components such as `ALI_SPLIT_2`, `ALI_SPLIT_4`, or `ALI_SPLIT_N` differ only in the number of outputs. The `ALI_SPLIT_8_UNGATED` offers the maximum distribution in the standard family. Unlike a multiplexer (`ALI_MUX`) or a switch (`ALI_SWITCH`), here **each output is supplied with the same signal** – no selection or switching takes place.

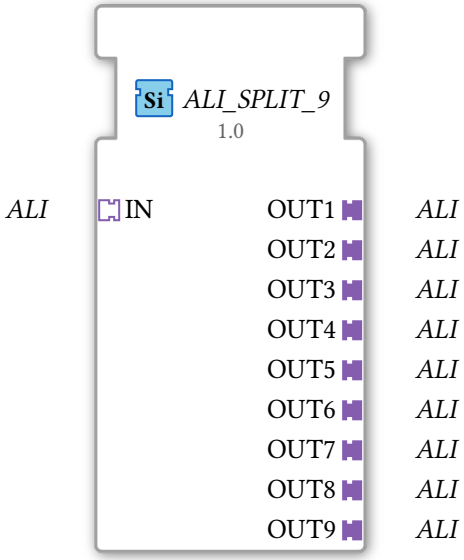
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALI_SPLIT_8_UNGATED` is a simple yet essential component for multiplying ALI signals. Its generic definition and the absence of state logic make it particularly suitable for robust, low-latency distributions in agricultural automation.

ALI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The **ALI_SPLIT_9** is a generic function block (FB) within the 4diac IDE, used to split an incoming **ALI** adapter signal (unidirectional) into up to nine separate output adapters. The block is implemented as a generic type ("Generic FB") and is typically customized using the type parameters `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`. The goal is flexible signal distribution without additional logic, purely at the adapter level.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Direction	Name	Type	Description
Socket (Input)	IN	`ALI` (unidirectional)	Input of the ALI signal to be distributed
Plug (Output 1)	OUT1	`ALI` (unidirectional)	First output – identical copy of the input signal

Plug (Output 2)	OUT2	`ALI` (unidirectional)	Second output
Plug (Output 3)	OUT3	`ALI` (unidirectional)	Third output
Plug (Output 4)	OUT4	`ALI` (unidirectional)	Fourth exit
Plug (output 5)	OUT5	`ALI` (unidirectional)	Fifth exit
Plug (output 6)	OUT6	`ALI` (unidirectional)	Sixth exit
Plug (output 7)	OUT7	`ALI` (unidirectional)	Seventh exit
Plug (output 8)	OUT8	`ALI` (unidirectional)	Eighth exit
Plug (output 9)	OUT9	`ALI` (unidirectional)	Ninth exit

Functionality

The function block operates passively as a pure **signal distributor**. The ALI adapter protocol applied to socket **IN** is duplicated unchanged to all nine plug outputs **OUT1…OUT9**. No data processing, event generation, or filtering takes place. Each output provides an identical logical and temporal copy of the incoming signal.

The internal implementation uses the generic type mechanisms of the 4diac IDE (`GenericClassName` and `TypeHash`), allowing the function block to be specifically adapted to the project requirements at runtime.

Technical Features

- **Generic Function Block**: The function block is declared as `GEN_ALI_SPLIT` (according to the attribute `eclipse4diac::core::GenericClassName`). Specific instances receive a unique type identifier via `TypeHash`.
- **No Events or Data**: The interface consists exclusively of adapters; there are no event inputs/outputs or data inputs/outputs.
- **Unidirectional ALI Adapters**: All adapters used are of type `adapter::types::unidirectional::ALI`, meaning communication is one-way (input → outputs). Feedback from the outputs to the input is not supported.
- **Simple Topology**: The module implements a 1:9 star distribution without additional buffers or synchronization.

State Overview

The module does not have its own state machine. Its behavior is determined solely by the adapter interfaces: As long as socket **IN** provides a signal, all nine plugs are active. As soon as **IN** becomes inactive (no signal), all outputs also provide no signal.

Application Scenarios

- **Distributing an ALI bus signal** to multiple downstream subsystems (e.g., sensor data to multiple controllers).
- **Splitting a master signal** for redundancy or parallel processing in plant control.
- **Test and debugging environments**: An ALI signal can be simultaneously routed to an analyzer and multiple target systems.
- **Generic interface extension**: When a system has only one ALI output but needs to serve multiple devices.

Comparison with similar components

- **ALI_SPLIT_2 / ALI_SPLIT_4 / ALI_SPLIT_8**: Variants with 2, 4, or 8 outputs. The component described here offers the maximum split to 9 channels.
- **ALI_MERGE** (hypothetical): Combines multiple ALI inputs into one output – the opposite function.
- **Event-based splitters**: Other components use events to control the signal flow; this adapter splitter operates in a signal-driven manner without event logic.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ALI_SPLIT_9** is a specialized, generic adapter function block (FB) that enables the simple and reliable distribution of a unidirectional ALI signal to up to nine separate receivers in industrial automation using the 4diac IDE. Its pure adapter interface makes it particularly resource-efficient and ideally suited for star network topologies in IEC 61499-based systems. Its generic nature allows for flexible adaptation to specific requirements without changing the block structure.

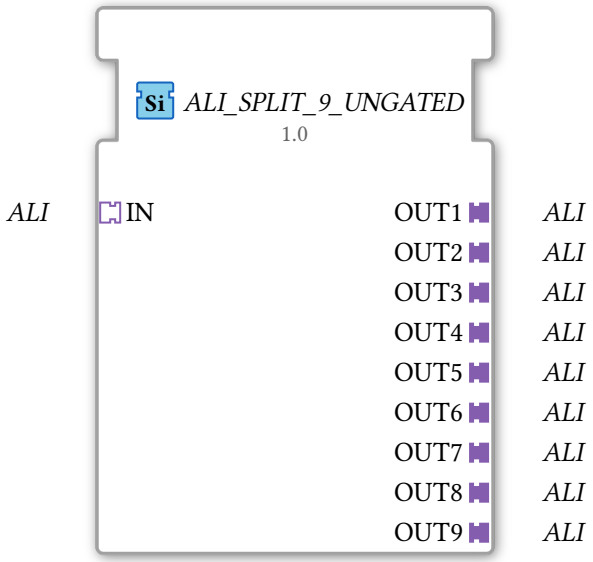
Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALI_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **ALI_SPLIT_9_UNGATED** is a generic function block (FB) within the 4diac IDE, used to split an incoming **ALI** adapter signal (unidirectional) into up to nine separate output adapters. The block is implemented as a generic type ("Generic FB") and is typically customized using the type parameters `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`. The goal is flexible signal distribution without additional logic, purely at the adapter level.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Direction	Name	Type	Description
Socket (Input)	IN	`ALI` (unidirectional)	Input of the ALI signal to be distributed
Plug (Output 1)	OUT1	`ALI` (unidirectional)	First output – identical copy of the input signal
Plug (Output 2)	OUT2	`ALI` (unidirectional)	Second output
Plug (Output 3)	OUT3	`ALI` (unidirectional)	Third output
Plug (Output 4)	OUT4	`ALI` (unidirectional)	Fourth exit
Plug (output 5)	OUT5	`ALI` (unidirectional)	Fifth exit
Plug (output 6)	OUT6	`ALI` (unidirectional)	Sixth exit
Plug (output 7)	OUT7	`ALI` (unidirectional)	Seventh exit
Plug (output 8)	OUT8	`ALI` (unidirectional)	Eighth exit
Plug (output 9)	OUT9	`ALI` (unidirectional)	Ninth exit

Functionality

The function block operates passively as a pure **signal distributor**. The ALI adapter protocol applied to socket **IN** is duplicated unchanged to all nine plug outputs **OUT1...OUT9**. No data processing, event generation, or filtering takes place. Each output provides an identical logical and temporal copy of the incoming signal.

The internal implementation uses the generic type mechanisms of the 4diac IDE (`GenericClassName` and `TypeHash`), allowing the function block to be specifically adapted to the project requirements at runtime.

Technical Features

- **Generic Function Block**: The function block is declared as `GEN_ALI_SPLIT` (according to the attribute `eclipse4diac::core::GenericClassName`). Specific instances receive a unique type identifier via `TypeHash`.
- **No Events or Data**: The interface consists exclusively of adapters; there are no event inputs/outputs or data inputs/outputs.
- **Unidirectional ALI Adapters**: All adapters used are of type `adapter::types::unidirectional::ALI`, meaning communication is one-way (input → outputs). Feedback from the outputs to the input is not supported.
- **Simple Topology**: The module implements a 1:9 star distribution without additional buffers or synchronization.

State Overview

The module does not have its own state machine. Its behavior is determined solely by the adapter interfaces: As long as socket **IN** provides a signal, all nine plugs are active. As soon as **IN** becomes inactive (no signal), all outputs also provide no signal.

Application Scenarios

- **Distributing an ALI bus signal** to multiple downstream subsystems (e.g., sensor data to multiple controllers).
- **Splitting a master signal** for redundancy or parallel processing in plant control.

- **Test and debugging environments**: An ALI signal can be simultaneously routed to an analyzer and multiple target systems.
- **Generic interface extension**: When a system has only one ALI output but needs to serve multiple devices.

Comparison with similar components

- **ALI_SPLIT_2 / ALI_SPLIT_4 / ALI_SPLIT_8**: Variants with 2, 4, or 8 outputs. The component described here offers the maximum split to 9 channels.
- **ALI_MERGE** (hypothetical): Combines multiple ALI inputs into one output – the opposite function.
- **Event-based splitters**: Other components use events to control the signal flow; this adapter splitter operates in a signal-driven manner without event logic.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **ALI_SPLIT_9_UNGATED** is a specialized, generic adapter function block (FB) that enables the simple and reliable distribution of a unidirectional ALI signal to up to nine separate receivers in industrial automation using the 4diac IDE. Its pure adapter interface makes it particularly resource-efficient and ideally suited for star network topologies in IEC 61499-based systems. Its generic nature allows for flexible adaptation to specific requirements without changing the block structure.

Related topic subpages on ms-muc-docs.de

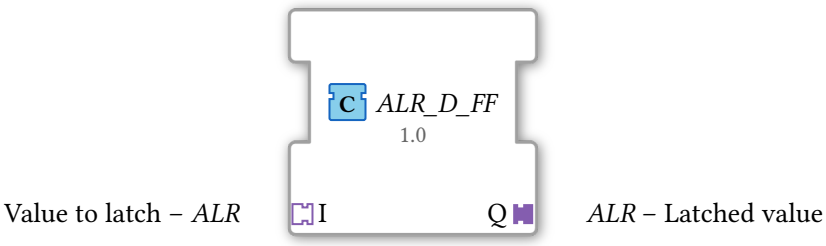
- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

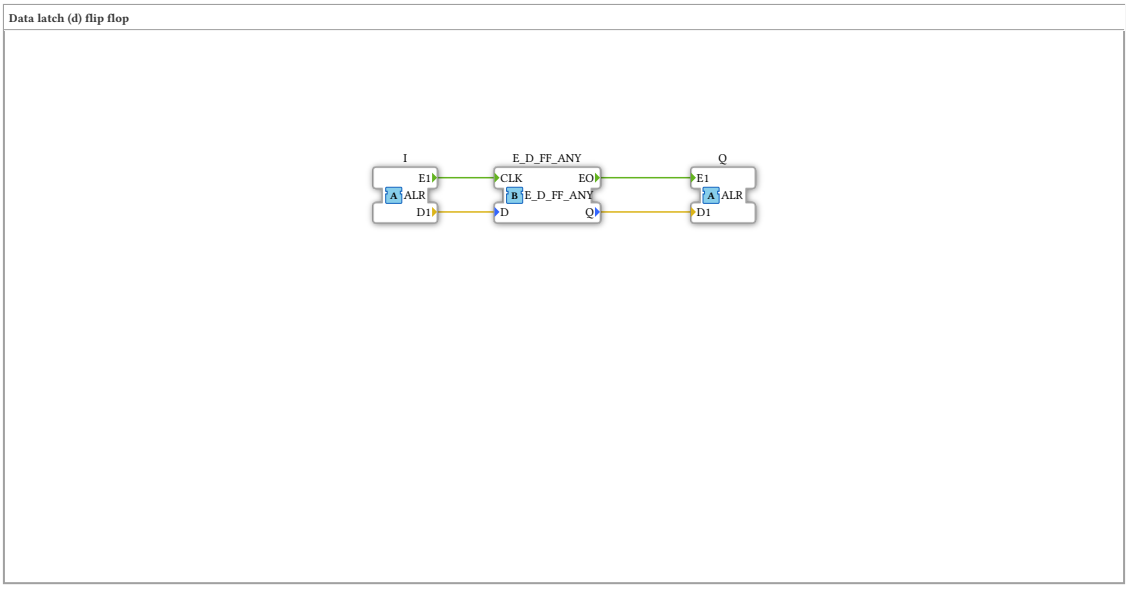
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALR_D_FF





• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_D_FF** implements a data-driven flip-flop (D-latch) with an adapter-based interface. It takes a data value on a clock event and makes it available at the output. The block encapsulates the IEC 61499 standard function `E_D_FF_ANY` and offers a unidirectional adapter connection for modular system integration.

Interface Structure

The function block has no direct event or data inputs/outputs. All communication takes place via two adapters:

Event Inputs

- **I.E1** (via socket `I`): Clock event – the current data value is taken on a rising edge.

Event Outputs

- **Q.E1** (via plug `Q`): Acknowledge event – sent after successful data value transfer.

Data Inputs

- **I.D1** (via socket `I`): Data input – the value to be latched (data type is defined by adapter `ALR`).

Data Outputs

- **Q.D1** (via plug `Q`): Data output – the stored, latched value.

Adapters

Adapter	Type	Direction	Description
`I`	adapter::types::unidirectional::ALR	Socket (Input)	Returns the clock event and the data value.
`Q`	adapter::types::unidirectional::ALR	Plug (Output)	Outputs the output event and the latched value.

Functionality

1. The function block expects a clock event at the adapter input `I.E1`.
2. Simultaneously, the data value from `I.D1` is read.
3. The internal function block `E_D_FF_ANY` receives the data value on the rising edge of the clock signal.
4. The received value is output unchanged to `Q.D1`.
5. An output event on `Q.E1` signals that the new value is ready.

This process repeats with each subsequent clock event. The output value remains stable between clock cycles (latch behavior).

Technical Features

- **Adapter-based interface**: The function block (FB) uses the unidirectional adapter `ALR`, which carries exactly one event and one data value. This enables loose coupling and easy interconnection with other adapter blocks.
- **Delegation to IEC 61499 standard**: The actual flip-flop logic is implemented by the standard FB `E_D_FF_ANY`, which is designed for any data type ("ANY"). `ALR_D_FF` therefore inherits its robustness and type neutrality.
- **No dedicated state machine**: The function block (FB) does not have an internal state machine – state management is handled entirely within the embedded `E_D_FF_ANY`.

State Overview

The function block has no explicit states. The internal logic (flip-flop) only recognizes two implicit states:

- **Waiting for clock event**: The output retains the last latched value.
- **Latch moment**: Upon a clock event, the input value is adopted and the output is updated.

Application Scenarios

- **Data storage**: Holding a digital signal (e.g., switch position) until the next query.
- **Synchronization**: Transferring asynchronous data into a clocked context.
- **Buffering**: Temporarily storing values in control sequences, e.g., in state machines or sequence controllers.

Comparison with Similar Components

- **`ALR_SR_FF`** (Set-Reset Flip-Flop): Offers two control inputs (Set/Reset) instead of a clock input. Suitable for priority-driven applications.
- **`ALR_JK_FF`** (JK Flip-Flop): More complex switching behavior with toggling and holding. More flexible, but more expensive.
- **`E_D_FF_ANY`** (Direct): Offers the same functionality, but with separate event and data inputs/outputs instead of an adapter. The `ALR_D_FF` simplifies coupling in adapter-based frameworks.

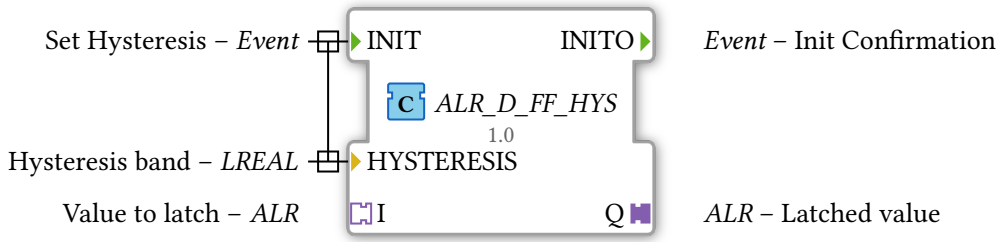
Conclusion

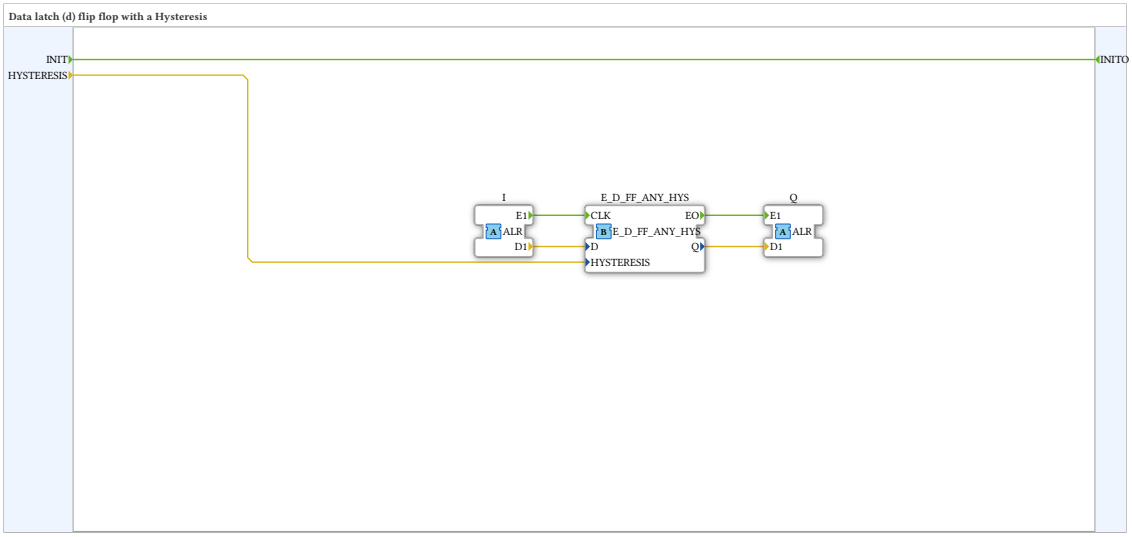
The `ALR_D_FF` is a lean yet reliable D flip-flop component for use in adapter-oriented 4diac applications. It combines proven IEC 61499 standard logic with a clearly structured adapter interface and is particularly suitable for modular, reusable control systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

ALR_D_FF_HYS





• – ◦ • – ◦ • – ◦ •

Introduction

The **ALR_D_FF_HYS** function block implements a data latch (D flip-flop) with adjustable hysteresis. It receives a data value via an adapter socket and outputs the latched value via an adapter plug. The hysteresis is set during the initialization event and serves to suppress noise or small fluctuations in the input signal.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Sets the hysteresis (with the data value `HYSTERESIS`) and starts the initialization.

Event Outputs

Event	Type	Description
INITO	EInit	Confirms successful initialization.

Data Inputs

Variable	Type	Description
HYSTERESIS	LREAL	Size of the hysteresis band (passed on the `INIT` event).

Data Outputs

The FB has no direct data outputs. The latched value is output exclusively via the adapter plug `Q`.

Adapter

Name	Type	Direction	Description
I (Socket)	`adapter::types::unidirectional::ALR`	Input	Receives the value to be latched via the event `I.E1` and the associated data value `I.D1`.
Q (Plug)	`adapter::types::unidirectional::ALR`	Output	Outputs the latched value via the event `Q.E1` and the data value `Q.D1`.

The adapter used, **ALR**, is a unidirectional data adapter with one event (`E1`) and one data value (`D1`).

Functionality

The function block (FB) works internally with the block `E_D_FF_ANY_HYS`, which implements the actual latch and hysteresis logic. The sequence of events is as follows:

1. **Initialization**: Upon arrival of `INIT`, the value of `HYSTERESIS` is forwarded to the internal block, and the output event `INITO` is immediately acknowledged.
2. **Latch Operation**: On each rising edge of the event `I.E1` (from the socket), the current data value `I.D1` is adopted. However, the latched value is only output when it leaves the band defined by the hysteresis. This suppresses small changes around the current latch value.
3. **Output**: As soon as the latched value leaves the hysteresis band, the internal function block generates an event on `E0`, which is output externally as `Q.E1`, and the new value is available on `Q.D1`.

Technical Features

- The hysteresis is set exclusively on the `INIT` event and remains constant during operation.
- The function block (FB) does not use its own state machine, but delegates all logic to the internal block `E_D_FF_ANY_HYS`.
- The initial state of the latch is undefined; initialization with `INIT` is mandatory before valid latch operations can occur.

State Overview

Since the FB implements the behavior via an internal block, there is no separate state diagram. The internal block `E_D_FF_ANY_HYS` typically operates with the following states:

- **Initial** – Waiting for the hysteresis setting.
- **Normal** – Latch operation with hysteresis band check (delayed output update).
- **Output** – Sending the newly latched value via the output adapter.

Application Scenarios

- **Signal Conditioning**: Latching of measured values with noise suppression in automation technology.
- **Debouncing**: Processing of switching signals where short pulses or bouncing need to be suppressed.
- **Threshold Monitoring**: Detection of analog values that only assume a new valid state above or below a hysteresis band.

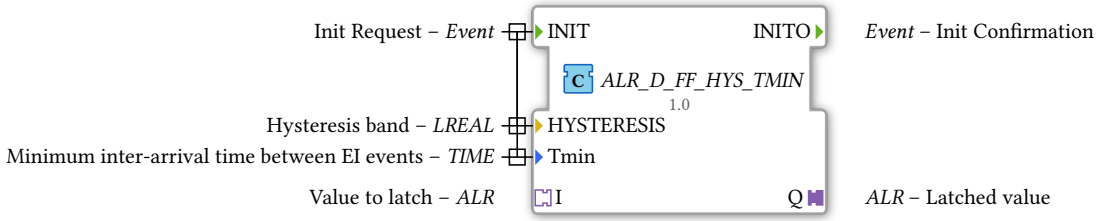
Comparison with Similar Components

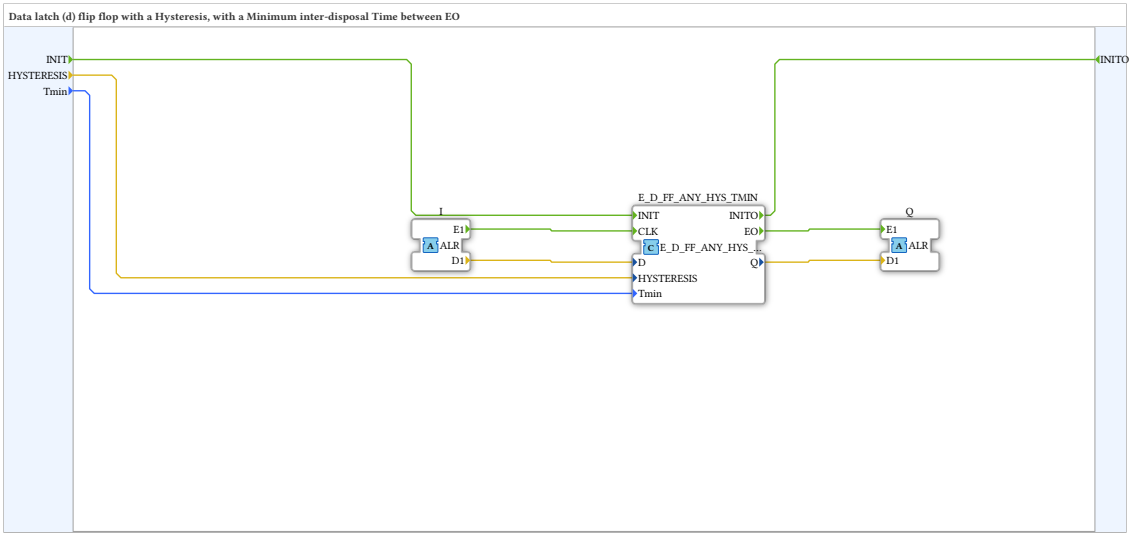
Component	Property
Simple D flip-flop (e.g., `E_D_FF`)	Latch without hysteresis – any change at the input is immediately registered.
D flip-flop with deadband (`E_D_FF_TOTBAND`)	It has a deadband that blocks changes as long as the value remains within the band.
ALR_D_FF_HYS	Hysteresis is only effective in one direction (threshold behavior), typically used to prevent flutter.

Conclusion

The ALR_D_FF_HYS is a specialized function block for applications where a data value needs to be latched and hysteresis applied simultaneously for noise suppression. Encapsulating the logic in an internal block keeps the interface simple and allows for flexible use via the adapter interface.

ALR_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_D_FF_HYS_TMIN** implements a clock-controlled D-latch (flip-flop) with adjustable hysteresis for data acquisition and a minimal time delay between successive output events. It processes an analog or Boolean value via an adapter input and outputs the buffered value via an adapter output. The hysteresis prevents unintended switching due to fluctuating input values, while the minimum time (Tmin) limits the event frequency at the output.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Initialization Request (sets hysteresis and Tmin)

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization Acknowledgement

Data Inputs

Name	Type	Comment
HYSTERESIS	LREAL	Hysteresis band for data transfer
Tmin	TIME	Minimum time between two output events

Data Outputs

The FB does not have its own data outputs. The latched value is provided via the adapter output `Q`.

Adapter

Direction	Name	Type (unidirectional)	Comment
Socket	I	ALR	Input value to be latched
Plug	Q	ALR	Output value (latched)

Functionality

The function block (FB) is implemented as a composite block that internally uses an FB of type `E_D_FF_ANY_HYS_TMIN` (from the library `logiBUS::signalprocessing::hysteresis`). The functionality in detail:

1. **Initialization (INIT)**: Upon the event `INIT`, the parameters `HYSTERESIS` and `Tmin` are passed to the internal FB. The internal FB is initialized and acknowledged with `INITO`.
2. **Data Transfer**: An event at the adapter input `I.E1` triggers the clock (`CLK`) of the internal flip-flop. Simultaneously, the data value `I.D1` is transferred as a new input `D`.
3. **Hysteresis**: The value is only transferred if the deviation from the currently latched value is greater than or equal to the set hysteresis. This prevents constant switching due to small fluctuations.
4. **Minimum Time (Tmin)**: After each output event (`EO`), a delay of `Tmin` is observed before another output event can be generated. Events arriving within this delay period are ignored.
5. **Output**: The latched value is provided via the adapter output `Q.D1`, and an event (`Q.E1`) signals the update.

Technical Features

- **Hysteresis**: The function block operates with an adjustable hysteresis band. The actual implementation (whether symmetrical or with thresholds) depends on the internal function block. A Schmitt trigger characteristic is common.
- **Time-Controlled Output**: `Tmin` limits the maximum output frequency. This is useful for reducing the load on downstream components or ensuring a minimum signal duration.
- **Adapter Interface**: The use of adapters (`ALR`) enables loose coupling between the data value and the event – an event implicitly carries the corresponding data value.
- **Reusability**: The FB is built from a standardized hysteresis flip-flop and can be easily adapted to other data formats.

State Overview

Since the function block consists of several internal states (e.g., waiting for clock signal, hysteresis check, waiting for minimum time), a simplified state description is given here:

1. **Initialized** (after INIT and INITO)
2. **Ready** – waits for an event at input `I.E1`
3. **Hysteresis Check** – compares the new value with the latched value

- If the hysteresis limit is exceeded: the new value is adopted, and the process transitions to **Waiting for Minimum Time**
- Otherwise: returns to **Ready** without output

1. **Waiting for Minimum Time** – after the minimum time has elapsed, the output event `Q.E1` is generated, and the latched value is displayed at `Q.D1`Output; then return to **Ready**

Application Scenarios

- **Analog Switching Thresholds**: Monitoring of sensors (e.g., temperature, pressure) with adjustable hysteresis and a minimum switching pause to suppress bounce or noise.
- **Digital Signal Conditioning**: Processing of square wave signals with noise, where hysteresis stabilizes the edges and Tmin enforces a defined pulse duration.
- **Stepper Motor Control**: Latching of position values with hysteresis to counter mechanical vibrations, accompanied by time-debouncing of the output commands.
- **Safety-Critical Systems**: Minimum time between output changes prevents excessively rapid state transitions and protects subsequent actuators.

Comparison with Similar Function Blocks

Function Block	Properties	Difference to ALR_D_FF_HYS_TMIN
Simple D flip-flop (D_FF)	No hysteresis, no timing	No signal stabilization or frequency limiting
Schmitt trigger (e.g., S_TRIG)	Hysteresis only, no timing, often only binary	Differs with analog values and forced pauses

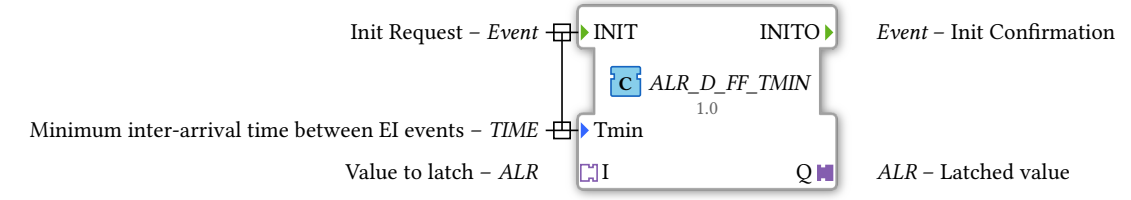
Edge-triggered latch with dead time	Timing (dead time) present, but usually without hysteresis	Combination of both effects (hysteresis + Tmin)
Filter block (e.g., FILTER)	Noise reduction, e.g., as a low-pass filter, no discrete state change	Explicit latch with event output and adjustable hysteresis

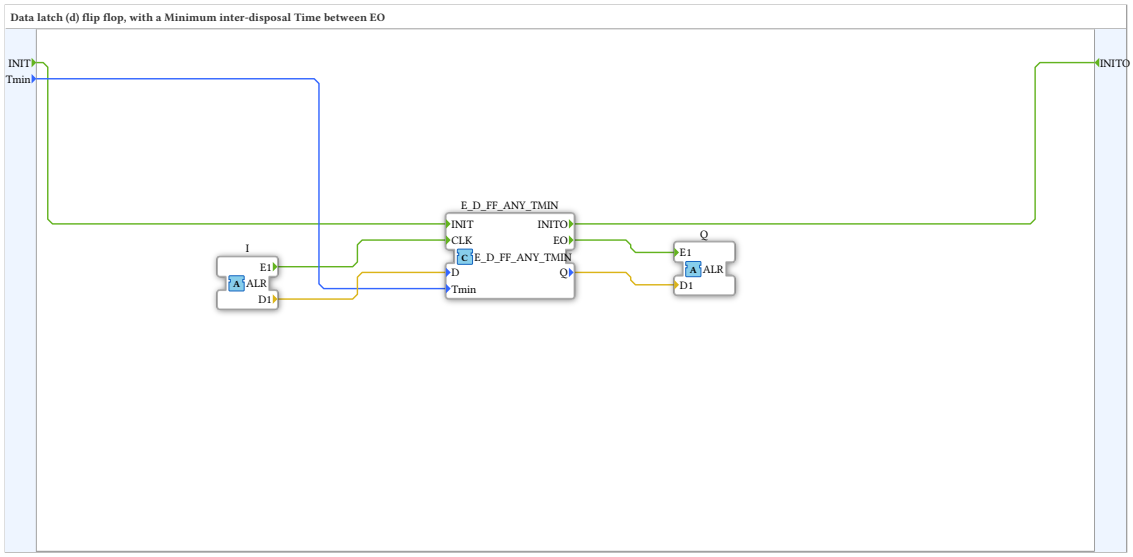
This block combines the advantages of hysteresis with a minimal event spacing function, making it particularly suitable for harsh industrial environments.

Conclusion

The **ALR_D_FF_HYS_TMIN** is a robust, configurable function block for reliable signal processing with hysteresis and time-based debouncing. It is suitable for all applications requiring a soft threshold and limiting the output event frequency. The adapter-based interface facilitates integration into complex control architectures.

ALR_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_D_FF_TMIN** implements a clock-edge-triggered data latch flip-flop (D flip-flop) with a minimum inter-disposal time (MIT) between two consecutive event outputs. It enables the latching of a data value upon receiving an event and ensures that the output event pulse does not occur faster than a definable time interval. This is particularly useful for protecting downstream systems from overload or for maintaining minimum time intervals in event communication.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Initialization Request; Sets the minimum time `Tmin`

Event Outputs

Event	Type	Comment
INITO	EInit	Confirmation of successful initialization

Data Inputs

Name	Type	Comment
Tmin	TIME	Minimum wait time between two EO events

Data Outputs

No dedicated data outputs available – the latched value is provided via the adapter output.

Adapter

| Adapter | Direction | Type | Comment | | Tmin | TIME | Minimum wait time between two EO events | | Adapter | Direction | Type | Comment | | Adapter | | ... |–––|–––––|––––––|–––––––––––––| | I | Socket | ALR (unidirectional) | Input value (`D1`) and clock (`E1`) | | Q | Plug | ALR (unidirectional) | Output value (`D1`) and acknowledgement (`E1`) |

The unidirectional adapter `ALR` has an event input `E1` and a data input `D1` (socket side) and an event output `E1` and a data output `D1` (plug side). The exact semantics of `E1` and `D1` are application-specific; here, `E1` serves as the clock signal and `D1` as the data value to be latched.

Functionality

After initialization (INIT), the function block operates as follows:

1. An **event at the input adapter `I.E1`** triggers the latch operation. The current value `I.D1` is internally adopted.
2. The latched value is immediately output at the output adapter `Q.D1`.
3. Simultaneously, an event `Q.E1` is triggered – but only after at least `Tmin` has elapsed since the last output event.

If another `I.E1` event arrives within the waiting time, it is ignored (or processed only after `Tmin` has elapsed). The minimum delay is implemented by the internal function block `E_D_FF_ANY_TMIN`.

The data value is updated on every valid clock cycle, but the event output is delayed.

Technical Features

- **Composite Function Block**: The function block is implemented as a composite function block and internally uses the function block `iec61499::events::E_D_FF_ANY_TMIN`. The logic of the actual flip-flop and the timing control are encapsulated there.
- **Universal Time Interval**: `Tmin` can be set at runtime via INIT and is not fixed.
- **Buffered Events**: Since the event output is delayed, incoming clock events can be lost during the waiting period unless the internal function block buffers them (not specified here). Typically, they are simply ignored.
- **Adapter-based interface**: The use of adapters enables flexible interconnection in modular control applications without the need for hardwiring.

State overview

The function block does not have explicit states but operates event-driven:

- **Idle**: Waiting for `I.E1` (after INIT).
- **Latching**: The data value is transferred upon `I.E1`, and `Q.E1` is triggered after `Tmin` has expired.
- **Blocked**: While `Tmin` has not yet expired, further `I.E1` are ignored.
- After the waiting period expires, the function block returns to the idle state.

Application Scenarios

- **Sensor Signal Debouncing**: A sensor delivers events faster than the subsequent controller can process them. `Tmin` defines the minimum interval.
- **Pulse Width Limiting**: Ensures that actuator controls are not switched too quickly in succession (e.g., valves, motors).

- **Clock Synchronization**: When a data value is latched with an asynchronous clock and only passed on after a defined dead time.
- **Safety Functions**: Prevents “rapid-fire” commands in critical applications.

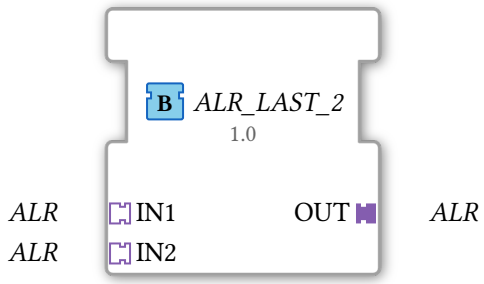
Comparison with Similar Function Blocks

- **E_D_FF_ANY_TMIN**: This is the internal function block – it offers the same functionality, but without the adapter interface. `ALR_D_FF_TMIN` encapsulates this in a more user-friendly, adapter-based form.
- **Standard IEC 61499 D flip-flops**: These often lack time-based control of output events. The function block described here extends this functionality with the `Tmin` delay.
- **Function block with TON/Timer**: A simple timer would only trigger after a delay but would not hold a data value. The combination with a latch is specific.

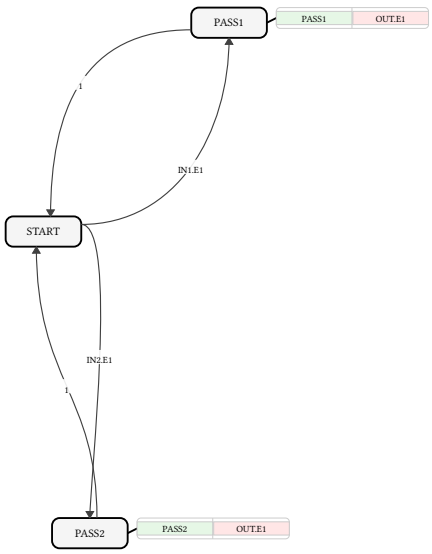
Conclusion

The function block `ALR_D_FF_TMIN` combines the features of a D flip-flop with an adjustable minimum time between event outputs. Thanks to its adapter technology and simple initialization, it can be flexibly deployed in IoT, automation, or control environments. It is particularly suitable for applications that require temporal decoupling between data acquisition and event transmission.

ALR_LAST_2



ECC: ALR_LAST_2



Introduction

The ALR_LAST_2 function block merges two same-type LREAL adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event

wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: LREAL adapter (unidirectional) - first signal source
- **IN2**: LREAL adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: LREAL adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

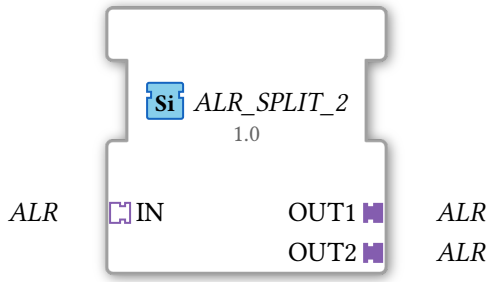
Application Scenarios

- Merging two equally-valid LREAL signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same LREAL value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

ALR_LAST_2 is the inverse of `ALR_SPLIT_2`: ALR_SPLIT_2 distributes 1 input to 2 outputs, ALR_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

ALR_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The ALR_SPLIT_2 function block splits an incoming ALR adapter signal into two identical output signals. It is designed generically (`GEN_ALR_SPLIT`) and is suitable for distributing alarm or lifecycle signals.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

- **IN** (Socket): Type `adapter::types::unidirectional::ALR` – Input adapter for the ALR signal.
- **OUT1** (Plug): Type `adapter::types::unidirectional::ALR` – first output adapter.
- **OUT2** (Plug): Type `adapter::types::unidirectional::ALR` – second output adapter.

Functionality

The function block forwards the ALR signal received via socket `IN` unchanged to both plugs `OUT1` and `OUT2`. No transformation or delay occurs. The signal is passively split.

Technical Features

- The function block is implemented as a generic function block (attribute `GenericClassName = 'GEN_ALR_SPLIT'`).
- Unidirectional adapters of type `ALR` are used, which only support one direction of data flow.
- No internal timers, states, or events – pure signal distribution.

State Overview

This function block has no state machines. Its behavior is determined solely by the adapter definition.

Application Scenarios

- Distribution of an ALR signal (e.g., alarm, temperature limit) to two different receivers.
- As part of a logic circuit that needs to evaluate an alarm signal multiple times.
- Use in control systems that forward a signal in parallel to two function blocks.

Comparison with Similar Function Blocks

- **ALR_SPLIT_2** splits an ALR signal to two outputs; comparable to event splitters like `E_SWITCH` or `E_SPLIT`, but for ALR adapters.
- Unlike data-based splitters (`MUX`, `F_SPLIT`), no data manipulation takes place here.
- Variants with more outputs may exist (e.g., `ALR_SPLIT_4`), allowing for a higher number of distributions.

Change Detection

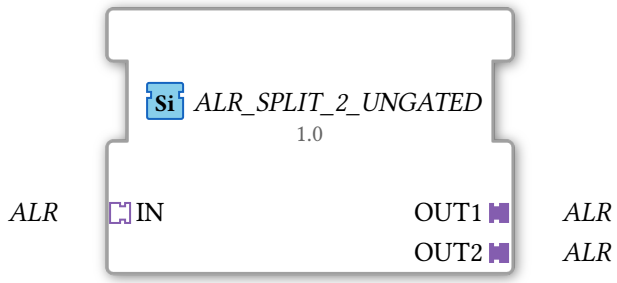
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

ALR_SPLIT_2` is a simple and effective component for splitting an incoming ALR adapter signal to two outputs. Due to its generic nature and lack of logic, it is ideally suited for clean signal distribution in IEC 61499 systems.

ALR_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of ALR_SPLIT_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The ALR_SPLIT_2_UNGATED function block splits an incoming ALR adapter signal into two identical output signals. It is designed generically (`GEN_ALR_SPLIT`) and is suitable for distributing alarm or lifecycle signals.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

- **IN** (Socket): Type `adapter::types::unidirectional::ALR` – Input adapter for the ALR signal.
- **OUT1** (Plug): Type `adapter::types::unidirectional::ALR` – first output adapter.
- **OUT2** (Plug): Type `adapter::types::unidirectional::ALR` – second output adapter.

Functionality

The function block forwards the ALR signal received via socket `IN` unchanged to both plugs `OUT1` and `OUT2`. No transformation or delay occurs. The signal is passively split.

Technical Features

- The function block is implemented as a generic function block (attribute `GenericClassName = 'GEN_ALR_SPLIT'`).
- Unidirectional adapters of type `ALR` are used, which only support one direction of data flow.
- No internal timers, states, or events – pure signal distribution.

State Overview

This function block has no state machines. Its behavior is determined solely by the adapter definition.

Application Scenarios

- Distribution of an ALR signal (e.g., alarm, temperature limit) to two different receivers.
- As part of a logic circuit that needs to evaluate an alarm signal multiple times.
- Use in control systems that forward a signal in parallel to two function blocks.

Comparison with Similar Function Blocks

- **ALR_SPLIT_2_UNGATED** splits an ALR signal to two outputs; comparable to event splitters like `E_SWITCH` or `E_SPLIT`, but for ALR adapters.
- Unlike data-based splitters (`MUX`, `F_SPLIT`), no data manipulation takes place here.

- Variants with more outputs may exist (e.g., `ALR_SPLIT_4`), allowing for a higher number of distributions.

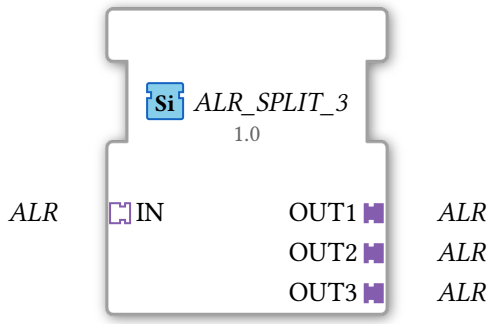
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

ALR_SPLIT_2_UNGATED` is a simple and effective component for splitting an incoming ALR adapter signal to two outputs. Due to its generic nature and lack of logic, it is ideally suited for clean signal distribution in IEC 61499 systems.

ALR_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALR_SPLIT_3** serves as a generic SPLIT function block that splits an incoming ALR signal (alarm or event signal) into three identical outputs. It is primarily used in automation systems where a single signal needs to be distributed in parallel to multiple downstream components without altering the signal quality or logic.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

- **IN** – *Socket* (Input) of type `adapter::types::unidirectional::ALR` Receives the ALR signal to be distributed.
- **OUT1** – *Plug* (Output 1) of type `adapter::types::unidirectional::ALR` Passes the incoming signal on unchanged.
- **OUT2** – *Plug* (Output 2) of type `adapter::types::unidirectional::ALR`
- **OUT3** – *Plug* (Output 3) of type `adapter::types::unidirectional::ALR`

Functionality

This module functions purely as a signal distributor. As soon as an ALR signal is present at the adapter socket **IN** (e.g., an alarm state, an event, or an activation), this signal is forwarded to all three adapter

plugs **OUT1**, **OUT2**, and **OUT3** without delay or manipulation. No logical processing or filtering takes place. Distribution is handled passively via the connection logic of the 4diac IDE.

Technical Features

- **Generic Function Block** – The block can be created as a generic instance using the attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`. This allows for easy reuse and parameterization in different projects.
- **No State Machine** – The block does not have an ECC (Execution Control Chart), as its behavior is defined solely by the adapter connectivity.
- **Unidirectional Adapters** – All ALR adapters used are of type `unidirectional`; the signal direction is fixed (input ⟶ outputs).

State Overview

The function block does not define any internal states. The output signal always corresponds instantaneously to the input signal. A separate state description is therefore not required.

Application Scenarios

- **Alarm Distribution** – A central alarm detector (e.g., a higher-level control system) is connected to three independent control units that are intended to react to the alarm in parallel.
- **Signal Multiplexing** – A Boolean or value-based signal from a sensor is distributed to multiple actuators or visualizations.
- **Redundancy Architectures** – The same signal is sent to two or three independent paths to achieve fault tolerance.

Comparison with Similar Function Blocks

- **ALR_SPLIT_2** – Distributes a signal to two outputs (identical function, but fewer outputs).
- **ALR_SPLIT_4** – Distributes one signal to four outputs.
- **Manual Coupling** – Without the SPLIT function block, the signal would have to be tapped multiple times in the application or distributed via a common line, which reduces clarity and maintainability.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ALR_SPLIT_3** is a simple yet useful generic function block for multiplying an ALR signal. It facilitates structured and reusable interconnection in automation solutions where a signal needs to be passed on to multiple receivers without additional logic or latency.

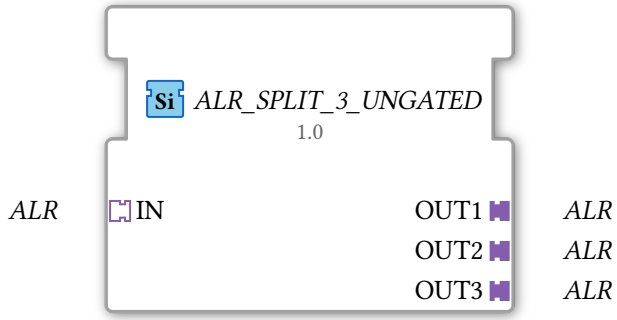
🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALR_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALR_SPLIT_3_UNGATED** serves as a generic SPLIT function block that splits an incoming ALR signal (alarm or event signal) into three identical outputs. It is primarily used in automation systems where a single signal needs to be distributed in parallel to multiple downstream components without altering the signal quality or logic.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

- **IN** – *Socket* (Input) of type `adapter::types::unidirectional::ALR` Receives the ALR signal to be distributed.
- **OUT1** – *Plug* (Output 1) of type `adapter::types::unidirectional::ALR` Passes the incoming signal on unchanged.
- **OUT2** – *Plug* (Output 2) of type `adapter::types::unidirectional::ALR`
- **OUT3** – *Plug* (Output 3) of type `adapter::types::unidirectional::ALR`

Functionality

This module functions purely as a signal distributor. As soon as an ALR signal is present at the adapter socket **IN** (e.g., an alarm state, an event, or an activation), this signal is forwarded to all three adapter plugs **OUT1**, **OUT2**, and **OUT3** without delay or manipulation. No logical processing or filtering takes place. Distribution is handled passively via the connection logic of the 4diac IDE.

Technical Features

- **Generic Function Block** – The block can be created as a generic instance using the attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`. This allows for easy reuse and parameterization in different projects.
- **No State Machine** – The block does not have an ECC (Execution Control Chart), as its behavior is defined solely by the adapter connectivity.
- **Unidirectional Adapters** – All ALR adapters used are of type `unidirectional`; the signal direction is fixed (input ⟶ outputs).

State Overview

The function block does not define any internal states. The output signal always corresponds instantaneously to the input signal. A separate state description is therefore not required.

Application Scenarios

- **Alarm Distribution** – A central alarm detector (e.g., a higher-level control system) is connected to three independent control units that are intended to react to the alarm in parallel.
- **Signal Multiplexing** – A Boolean or value-based signal from a sensor is distributed to multiple actuators or visualizations.
- **Redundancy Architectures** – The same signal is sent to two or three independent paths to achieve fault tolerance.

Comparison with Similar Function Blocks

- **ALR_SPLIT_2** – Distributes a signal to two outputs (identical function, but fewer outputs).
- **ALR_SPLIT_4** – Distributes one signal to four outputs.
- **Manual Coupling** – Without the SPLIT function block, the signal would have to be tapped multiple times in the application or distributed via a common line, which reduces clarity and maintainability.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

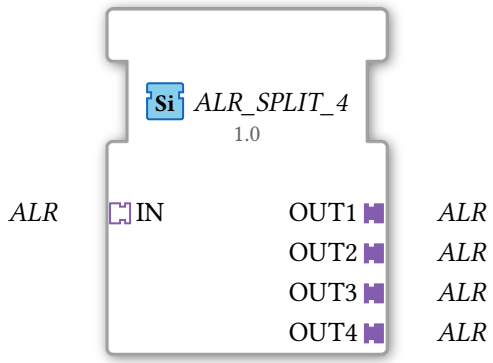
The **ALR_SPLIT_3_UNGATED** is a simple yet useful generic function block for multiplying an ALR signal. It facilitates structured and reusable interconnection in automation solutions where a signal needs to be passed on to multiple receivers without additional logic or latency.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALR_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALR_SPLIT_4** splits an incoming ALR adapter signal into four identical outputs. It is a generic function block (FB) that can be used with various ALR adapter types.

Interface Structure

Event Inputs

No event inputs are available. Event control is handled via the input adapter.

Event Outputs

No event outputs are available. Events are forwarded via the output adapters.

Data Inputs

No data inputs are available. Data is provided via the input adapter.

Data Outputs

No data outputs are available. Data is output via the output adapters.

Adapters

- **Socket (Input)**:
- **IN**: Unidirectional ALR adapter (type `adapter::types::unidirectional::ALR`). Receives the incoming signal.
- **Plugs (Outputs)**:
- **OUT1**, **OUT2**, **OUT3**, **OUT4**: Each a unidirectional ALR adapter (same type). Outputs the distributed signal.

Functionality

The module features direct pass-through: All events and data arriving at the input adapter **IN** are passed on unchanged and simultaneously to the four output adapters **OUT1** to **OUT4**. No processing, buffering, or filtering takes place. The module operates purely combinationally.

Technical Features

- **Generic Type**: The specific ALR adapter type can be defined at design time using the attribute `eclipse4diac::core::GenericClassName` (e.g., `'GEN_ALR_SPLIT'`).
- **Unidirectional**: The data flow direction is fixed from input to output. Backward communication is not supported.
- **No State Logic**: The function block does not contain an event-driven state machine (ECC) and does not include any time delays.

State Overview

The function block does not have its own states. Signal transmission is continuous and instantaneous.

Application Scenarios

- Distribution of an ALR signal (e.g., control commands, status messages) to multiple parallel modules or actuators.
- Creation of branches in adapter-based communication when multiple downstream components require the same information.
- Use in modular automation systems where a common signal needs to be split across multiple identical units.

Comparison with similar components

- **ALR_SPLIT_2 / ALR_SPLIT_N**: Split components with a different number of outputs (e.g., 2 or 8).
- **Event Splitters**: Split events but operate at the event level only. ALR_SPLIT_4, on the other hand, distributes complete adapter signals (events and data encapsulated).
- **Data Splitters**: Distribute individual data values, but without adapter encapsulation. ALR_SPLIT_4 is specifically optimized for use with unidirectional ALR adapter types.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

ALR_SPLIT_4 is a simple yet essential component for multiplying adapter connections. Its generic design allows it to be used with various ALR adapter types without requiring any modification to the block logic. It is particularly suitable for modular architectures where a signal needs to be distributed to multiple receivers in parallel.

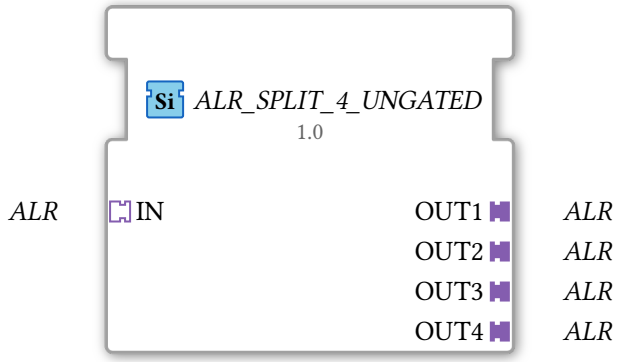
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ALR_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALR_SPLIT_4_UNGATED** splits an incoming ALR adapter signal into four identical outputs. It is a generic function block (FB) that can be used with various ALR adapter types.

Interface Structure

Event Inputs

No event inputs are available. Event control is handled via the input adapter.

Event Outputs

No event outputs are available. Events are forwarded via the output adapters.

Data Inputs

No data inputs are available. Data is provided via the input adapter.

Data Outputs

No data outputs are available. Data is output via the output adapters.

Adapters

- **Socket (Input)**:
- **IN**: Unidirectional ALR adapter (type `adapter::types::unidirectional::ALR`). Receives the incoming signal.
- **Plugs (Outputs)**:
- **OUT1**, **OUT2**, **OUT3**, **OUT4**: Each a unidirectional ALR adapter (same type). Outputs the distributed signal.

Functionality

The module features direct pass-through: All events and data arriving at the input adapter **IN** are passed on unchanged and simultaneously to the four output adapters **OUT1** to **OUT4**. No processing, buffering, or filtering takes place. The module operates purely combinationally.

Technical Features

- **Generic Type**: The specific ALR adapter type can be defined at design time using the attribute `eclipse4diac::core::GenericClassName` (e.g., `'GEN_ALR_SPLIT'`).
- **Unidirectional**: The data flow direction is fixed from input to output. Backward communication is not supported.
- **No State Logic**: The function block does not contain an event-driven state machine (ECC) and does not include any time delays.

State Overview

The function block does not have its own states. Signal transmission is continuous and instantaneous.

Application Scenarios

- Distribution of an ALR signal (e.g., control commands, status messages) to multiple parallel modules or actuators.
- Creation of branches in adapter-based communication when multiple downstream components require the same information.
- Use in modular automation systems where a common signal needs to be split across multiple identical units.

Comparison with similar components

- **ALR_SPLIT_2 / ALR_SPLIT_N**: Split components with a different number of outputs (e.g., 2 or 8).
- **Event Splitters**: Split events but operate at the event level only. ALR_SPLIT_4_UNGATED, on the other hand, distributes complete adapter signals (events and data encapsulated).
- **Data Splitters**: Distribute individual data values, but without adapter encapsulation. ALR_SPLIT_4_UNGATED is specifically optimized for use with unidirectional ALR adapter types.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

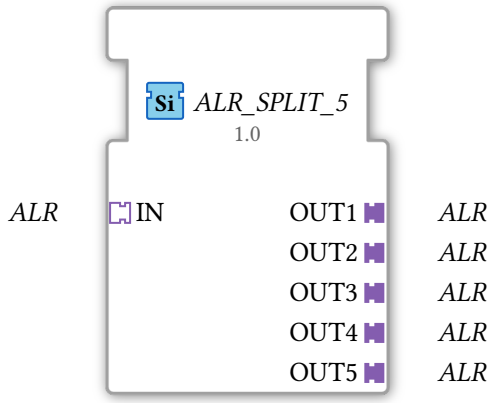
ALR_SPLIT_4_UNGATED is a simple yet essential component for multiplying adapter connections. Its generic design allows it to be used with various ALR adapter types without requiring any modification to the block logic. It is particularly suitable for modular architectures where a signal needs to be distributed to multiple receivers in parallel.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ALR_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALR_SPLIT_5** is used to distribute an incoming ALR signal to five identical outputs. It is implemented as a generic function block (Generic FB) and enables simple signal multiplication in control systems based on the adapter concept of IEC 61499.

Interface Structure

Event Inputs

No event inputs are available.

Event Outputs

No event outputs are available.

Data Inputs

No data inputs are available. All data transmission occurs exclusively via adapters.

Data Outputs

No data outputs are available.

Adapter

The module has one socket (input) and five plugs (outputs):

Direction	Name	Type	Description
Input	IN	`adapter::types::unidirectional::ALR`	Receives the ALR signal to be distributed
Output	OUT1	`adapter::types::unidirectional::ALR`	First output with the identical ALR signal
Output	OUT2	`adapter::types::unidirectional::ALR`	Second output with the identical ALR signal
Output	OUT3	`adapter::types::unidirectional::ALR`	Third output with the identical ALR signal
Output	OUT4	`adapter::types::unidirectional::ALR`	Fourth output with the identical ALR signal
Output	OUT5	`adapter::types::unidirectional::ALR`	Fifth output with the identical ALR signal

Functionality

This function block forwards the ALR signal received from socket `IN` unchanged to all five plugs `OUT1` to `OUT5`. This is a purely combinational forwarding process – no logic, delay, or data manipulation takes place. The block operates without requiring an event or state; as soon as a signal is present at the input, it is available at all outputs.

This function block forwards the ALR signal received from socket `IN` unchanged to all five plugs `OUT1` to `OUT5`.

- **Generic Function Block:** The function block is defined as a Generic FB and uses the Eclipse 4diac Generics mechanisms (`eclipse4diac::core::GenericClassName`). This allows it to be used in various contexts without modifying the core logic.
- **Pure Adapter Communication:** No traditional event or data inputs/outputs are used. All communication takes place via adapters of type `ALR`, which enable unidirectional data exchange.
- **Simple Structure:** No state machines or time dependencies – the function block is deterministic and resource-efficient.

The function block has no explicit state logic or state machine. It operates entirely combinatorially, meaning the output signal is derived directly from the input signal without delay.

- **Signal Distribution in Automation:** A sensor delivers an ALR signal that must be forwarded in parallel to several actuators (e.g., valves, drives).

- **Redundancy:** The signal can be distributed to various control units or monitoring modules.
- **Test and Simulation Environments:** A central ALR signal is split across multiple test points to enable parallel evaluations.
- **ALR_SPLIT_2, ALR_SPLIT_3, ALR_SPLIT_4:** These function blocks split an ALR signal across two, three, and four outputs, respectively. This function block extends this to five outputs.
- **General Split Function Blocks (e.g., DATA_SPLIT):** Similar function blocks exist for other data types. Their functionality is identical; only the adapter type used differs. The advantage of ALR_SPLIT_5 lies in its direct use of the ALR adapter protocol without additional type conversion.

The **ALR_SPLIT_5** is a simple yet valuable component for multiplying ALR signals in IEC 61499 systems. Its generic and combinatorial implementation makes it easy to integrate and resource-efficient. It is particularly suitable for applications where a signal needs to be distributed to multiple receivers without requiring additional logic or timing.

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

Change Detection

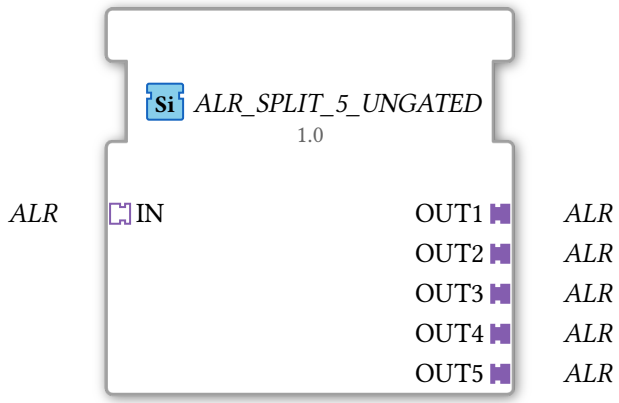
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

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

ALR_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALR_SPLIT_5_UNGATED** is used to distribute an incoming ALR signal to five identical outputs. It is implemented as a generic function block (Generic FB) and enables simple signal multiplication in control systems based on the adapter concept of IEC 61499.

Interface Structure

Event Inputs

No event inputs are available.

Event Outputs

No event outputs are available.

Data Inputs

No data inputs are available. All data transmission occurs exclusively via adapters.

Data Outputs

No data outputs are available.

Adapter

The module has one socket (input) and five plugs (outputs):

Direction	Name	Type	Description
Input	IN	`adapter::types::unidirectional::ALR`	Receives the ALR signal to be distributed
Output	OUT1	`adapter::types::unidirectional::ALR`	First output with the identical ALR signal
Output	OUT2	`adapter::types::unidirectional::ALR`	Second output with the identical ALR signal
Output	OUT3	`adapter::types::unidirectional::ALR`	Third output with the identical ALR signal
Output	OUT4	`adapter::types::unidirectional::ALR`	Fourth output with the identical ALR signal
Output	OUT5	`adapter::types::unidirectional::ALR`	Fifth output with the identical ALR signal

Functionality

This function block forwards the ALR signal received from socket `IN` unchanged to all five plugs `OUT1` to `OUT5`. This is a purely combinational forwarding process – no logic, delay, or data manipulation takes place. The block operates without requiring an event or state; as soon as a signal is present at the input, it is available at all outputs.

This function block forwards the ALR signal received from socket `IN` unchanged to all five plugs `OUT1` to `OUT5`.

- **Generic Function Block:** The function block is defined as a Generic FB and uses the Eclipse 4diac Generics mechanisms (`eclipse4diac::core::GenericClassName`). This allows it to be used in various contexts without modifying the core logic.
- **Pure Adapter Communication:** No traditional event or data inputs/outputs are used. All communication takes place via adapters of type `ALR`, which enable unidirectional data exchange.
- **Simple Structure:** No state machines or time dependencies – the function block is deterministic and resource-efficient.

The function block has no explicit state logic or state machine. It operates entirely combinatorially, meaning the output signal is derived directly from the input signal without delay.

- **Signal Distribution in Automation:** A sensor delivers an ALR signal that must be forwarded in parallel to several actuators (e.g., valves, drives).

- **Redundancy:** The signal can be distributed to various control units or monitoring modules.
- **Test and Simulation Environments:** A central ALR signal is split across multiple test points to enable parallel evaluations.
- **ALR_SPLIT_2, ALR_SPLIT_3, ALR_SPLIT_4:** These function blocks split an ALR signal across two, three, and four outputs, respectively. This function block extends this to five outputs.
- **General Split Function Blocks (e.g., DATA_SPLIT):** Similar function blocks exist for other data types. Their functionality is identical; only the adapter type used differs. The advantage of ALR_SPLIT_5_UNGATED lies in its direct use of the ALR adapter protocol without additional type conversion.

The **ALR_SPLIT_5_UNGATED** is a simple yet valuable component for multiplying ALR signals in IEC 61499 systems. Its generic and combinatorial implementation makes it easy to integrate and resource-efficient. It is particularly suitable for applications where a signal needs to be distributed to multiple receivers without requiring additional logic or timing.

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

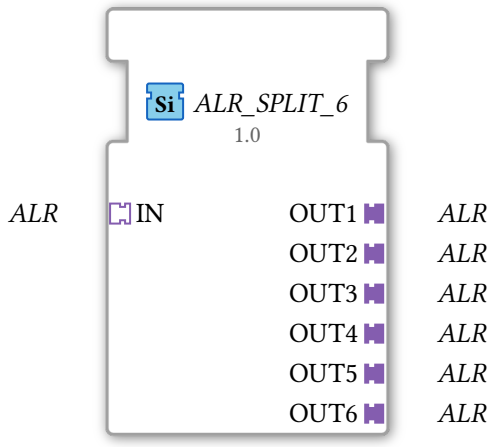
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

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

ALR_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The ALR_SPLIT_6 function block is a generic 1:6 splitter for the adapter type `adapter::types::unidirectional::ALR`. It distributes an incoming ALR signal unchanged to six outputs and serves as a passive signal distributor without data manipulation.

Interface Structure

Event Inputs
None

Event Outputs
None

Data Inputs
None

Data Outputs
None

Adapter

Name	Type	Direction	Description
IN	adapter::types::unidirectional::ALR	Socket (Input)	ALR input signal copied to all outputs
OUT1	adapter::types::unidirectional::ALR	Plug (Output)	First ALR output
OUT2	adapter::types::unidirectional::ALR	Plug (Output)	Second ALR output
OUT3	adapter::types::unidirectional::ALR	Plug (Output)	Third ALR output
OUT4	adapter::types::unidirectional::ALR	Plug (Output)	Fourth ALR output
OUT5	adapter::types::unidirectional::ALR	Plug (Output)	Fifth ALR output
OUT6	adapter::types::unidirectional::ALR	Plug (Output)	Sixth ALR Output

Functionality
This function block forwards the adapter signal present at Socket IN unchanged to all six plugs OUT1–OUT6. No transformation, delay, or selection takes place – all outputs always receive the identical signal copy. The function block is declared as a generic function block (`GEN_ALR_SPLIT`) and can be typed by specifying the concrete adapter type.

Technical Features
- Purely passive splitter without events, data ports, or state machines.
- Generic implementation via the metadata `eclipse4diac::core::GenericClassName = 'GEN_ALR_SPLIT'`.
- All adapter interfaces are unidirectional; there are no return channels.
- No runtime dependencies or initialization are required.

State Overview
The function block has no ECC (Execution Control Chart) and no internal states. It operates continuously and immediately forwards every incoming adapter signal to all outputs.

Application Scenarios
- Duplication of an ALR signal for parallel loads in control networks.
- Signal distribution in agricultural automation systems (based on the ALR adapter context).
- Easy provision of an identical reference for multiple downstream logic blocks.

Comparison with Similar Blocks
- **ALR_SPLIT_2 / ALR_SPLIT_3 / …** – same functionality, but with fewer outputs.
- **Event-based splitters (e.g., E_SPLIT)** – operate with event/data separation, while ALR_SPLIT_6 uses adapters exclusively.
- **Data multiplexers** – select one output; ALR_SPLIT_6 always activates all outputs.

- `ALR_SPLIT_6_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

ALR_SPLIT_6 is a minimalist and reliable block for signal duplication at the adapter level. Its generic design facilitates reuse in different projects, and the absence of complex logic keeps the data flow model clear and simple.

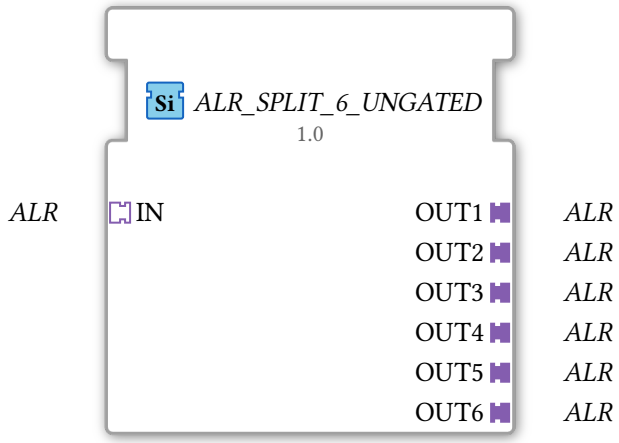
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALR_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The ALR_SPLIT_6_UNGATED function block is a generic 1:6 splitter for the adapter type `adapter::types::unidirectional::ALR`. It distributes an incoming ALR signal unchanged to six outputs and serves as a passive signal distributor without data manipulation.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Name	Type	Direction	Description
IN	adapter::types::unidirectional::ALR	Socket (Input)	ALR input signal copied to all outputs
OUT1	adapter::types::unidirectional::ALR	Plug (Output)	First ALR output
OUT2	adapter::types::unidirectional::ALR	Plug (Output)	Second ALR output
OUT3	adapter::types::unidirectional::ALR	Plug (Output)	Third ALR output
OUT4	adapter::types::unidirectional::ALR	Plug (Output)	Fourth ALR output
OUT5	adapter::types::unidirectional::ALR	Plug (Output)	Fifth ALR output
OUT6	adapter::types::unidirectional::ALR	Plug (Output)	Sixth ALR Output

Functionality

This function block forwards the adapter signal present at Socket IN unchanged to all six plugs OUT1–OUT6. No transformation, delay, or selection takes place – all outputs always receive the identical signal copy. The function block is declared as a generic function block (`GEN_ALR_SPLIT`) and can be typed by specifying the concrete adapter type.

Technical Features

- Purely passive splitter without events, data ports, or state machines.
- Generic implementation via the metadata `eclipse4diac::core::GenericClassName = 'GEN_ALR_SPLIT'`.
- All adapter interfaces are unidirectional; there are no return channels.
- No runtime dependencies or initialization are required.

State Overview

The function block has no ECC (Execution Control Chart) and no internal states. It operates continuously and immediately forwards every incoming adapter signal to all outputs.

Application Scenarios

- Duplication of an ALR signal for parallel loads in control networks.
- Signal distribution in agricultural automation systems (based on the ALR adapter context).
- Easy provision of an identical reference for multiple downstream logic blocks.

Comparison with Similar Blocks

- **ALR_SPLIT_2 / ALR_SPLIT_3 / …** – same functionality, but with fewer outputs.
- **Event-based splitters (e.g., E_SPLIT)** – operate with event/data separation, while ALR_SPLIT_6_UNGATED uses adapters exclusively.
- **Data multiplexers** – select one output; ALR_SPLIT_6_UNGATED always activates all outputs.
- **ALR_SPLIT_6**: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

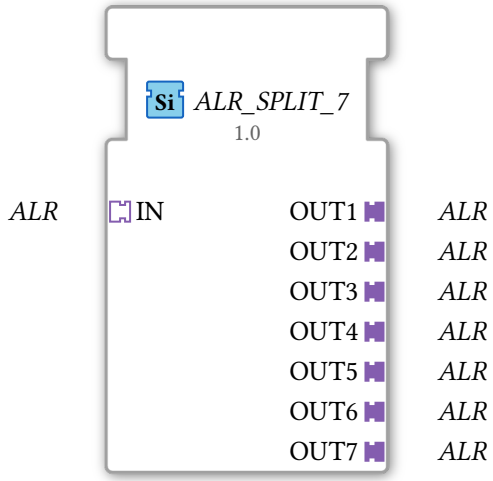
ALR_SPLIT_6_UNGATED is a minimalist and reliable block for signal duplication at the adapter level. Its generic design facilitates reuse in different projects, and the absence of complex logic keeps the data flow model clear and simple.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALR_SPLIT_7



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_SPLIT_7` is used to split an incoming ALR (Agricultural Logic Resource) signal into seven identical output signals. It is designed as a generic block and enables the parallel distribution of an alarm or control signal to up to seven downstream components or subsystems.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::ALR`	Socket (Input)	Incoming ALR signal, which is distributed to the outputs.
OUT1	`adapter::types::unidirectional::ALR`	Plug (Output)	First output of the split signal.
OUT2	`adapter::types::unidirectional::ALR`	Plug (Output)	Second output of the split signal.

OUT3	adapter::types::unidirectional::ALR	Plug (Output)	Third output of the split signal.
OUT4	adapter::types::unidirectional::ALR	Plug (Output)	Fourth output of the split signal.
OUT5	adapter::types::unidirectional::ALR	Plug (Output)	Fifth output of the split signal.
OUT6	adapter::types::unidirectional::ALR	Plug (Output)	Sixth output of the split signal.
OUT7	adapter::types::unidirectional::ALR	Plug (Output)	Seventh output of the split signal.

Functionality

The ALR_SPLIT_7 module forwards the ALR signal present at its socket IN unchanged to all seven plugs (OUT1 to OUT7). No transformation, delay, or filtering of the signal takes place. The module functions purely as a splitter for a unidirectional ALR data stream.

Technical Features

- **Generic FB:** The function block is implemented as a generic function block (GenericClassName = 'GEN_ALR_SPLIT') and can be parameterized or instantiated depending on the application.
- **Unidirectional:** All adapters are of type adapter::types::unidirectional::ALR, meaning data flows in only one direction – from the input to the outputs.
- **No Event Control:** The function block has no event inputs or outputs; signal transmission is continuous (data-driven).

State Overview

The function block does not have an internal state machine. There are no states in the sense of an ECC (Execution Control Chart). It operates purely combinationally and always outputs the input signal to all outputs.

Application Scenarios

- **Distributing an Alarm to Multiple Units:** A central alarm signal (e.g., a fault) is to be forwarded to seven different controllers or display elements.
- **Parallel Processing:** An ALR signal is required to trigger or monitor multiple independent subsystems simultaneously.
- **Redundancy and Testing:** The signal is sent to multiple test or logging components without interfering with each other.

Comparison with Similar Components

Component	Number of Outputs	Special Feature
ALR_SPLIT_2	2	Simple Dual Distribution
ALR_SPLIT_4	4	Quad Distribution
ALR_SPLIT_7	7	Seven-Way Distribution (this module)
ALR_SPLIT_N (generic)	Configurable	Flexible splitter with adjustable number of paths

All variations are based on the same principle: unidirectional distribution without signal modification.

Change Detection

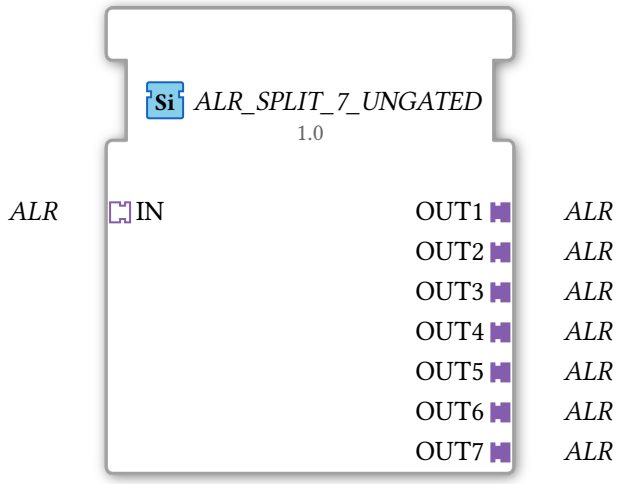
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The ALR_SPLIT_7 is a simple yet useful function block for splitting a single ALR signal into seven parallel paths. Its generic implementation allows it to be used in various industrial or agricultural automation scenarios where a signal is needed multiple times. It does not offer complex logic but reliably fulfills its function as a passive splitter.

ALR_SPLIT_7_UNGATED

i UNGATED variant: This block is the ungated version of `ALR_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_SPLIT_7_UNGATED` is used to split an incoming ALR (Agricultural Logic Resource) signal into seven identical output signals. It is designed as a generic block and enables the parallel distribution of an alarm or control signal to up to seven downstream components or subsystems.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::ALR`	Socket (Input)	Incoming ALR signal, which is distributed to the outputs.
OUT1	`adapter::types::unidirectional::ALR`	Plug (Output)	First output of the split signal.
OUT2	`adapter::types::unidirectional::ALR`	Plug (Output)	Second output of the split signal.
OUT3	`adapter::types::unidirectional::ALR`	Plug (Output)	Third output of the split signal.

OUT4	adapter::types::unidirectional::ALR	Plug (Output)	Fourth output of the split signal.
OUT5	adapter::types::unidirectional::ALR	Plug (Output)	Fifth output of the split signal.
OUT6	adapter::types::unidirectional::ALR	Plug (Output)	Sixth output of the split signal.
OUT7	adapter::types::unidirectional::ALR	Plug (Output)	Seventh output of the split signal.

Functionality

The ALR_SPLIT_7_UNGATED module forwards the ALR signal present at its socket IN unchanged to all seven plugs (OUT1 to OUT7). No transformation, delay, or filtering of the signal takes place. The module functions purely as a splitter for a unidirectional ALR data stream.

Technical Features

- **Generic FB:** The function block is implemented as a generic function block (GenericClassName = 'GEN_ALR_SPLIT') and can be parameterized or instantiated depending on the application.
- **Unidirectional:** All adapters are of type adapter::types::unidirectional::ALR, meaning data flows in only one direction – from the input to the outputs.
- **No Event Control:** The function block has no event inputs or outputs; signal transmission is continuous (data-driven).

State Overview

The function block does not have an internal state machine. There are no states in the sense of an ECC (Execution Control Chart). It operates purely combinationally and always outputs the input signal to all outputs.

Application Scenarios

- **Distributing an Alarm to Multiple Units:** A central alarm signal (e.g., a fault) is to be forwarded to seven different controllers or display elements.
- **Parallel Processing:** An ALR signal is required to trigger or monitor multiple independent subsystems simultaneously.
- **Redundancy and Testing:** The signal is sent to multiple test or logging components without interfering with each other.

Comparison with Similar Components

Component	Number of Outputs	Special Feature
ALR_SPLIT_2	2	Simple Dual Distribution
ALR_SPLIT_4	4	Quad Distribution
ALR_SPLIT_7_UNGATED	7	Seven-Way Distribution (this module)
ALR_SPLIT_N (generic)	Configurable	Flexible splitter with adjustable number of paths

All variations are based on the same principle: unidirectional distribution without signal modification.

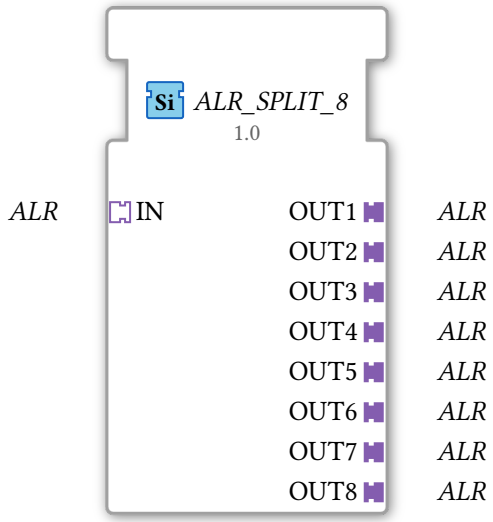
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The ALR_SPLIT_7_UNGATED is a simple yet useful function block for splitting a single ALR signal into seven parallel paths. Its generic implementation allows it to be used in various industrial or agricultural automation scenarios where a signal is needed multiple times. It does not offer complex logic but reliably fulfills its function as a passive splitter.

ALR_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALR_SPLIT_8** is used to split a single ALR signal (via a socket) into eight identical ALR outputs (plugs). This block is designed as a generic function block and enables signal distribution within an IEC 61499-based controller.

Interface Structure

Event Inputs

No direct event inputs are available. Control is integrated into the input adapter.

Event Outputs

No direct event outputs are available. Control is integrated into the output adapters.

Data Inputs

No direct data inputs are available. Data is received via the input adapter.

Data Outputs

No direct data outputs are available. Data is provided via the output adapters.

Adapter

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::ALR`	Socket	Input signal that is routed to all outputs
OUT1	`adapter::types::unidirectional::ALR`	Plug	Output 1 – duplicated signal
OUT2	`adapter::types::unidirectional::ALR`	Plug	Output 2 – duplicated signal
OUT3	`adapter::types::unidirectional::ALR`	Plug	Output 3 – duplicated signal
OUT4	`adapter::types::unidirectional::ALR`	Plug	Output 4 – duplicated signal
OUT5	`adapter::types::unidirectional::ALR`	Plug	Output 5 – duplicated signal
OUT6	`adapter::types::unidirectional::ALR`	Plug	Output 6 – Duplicated Signal
OUT7	`adapter::types::unidirectional::ALR`	Plug	Output 7 – Duplicated Signal
OUT8	`adapter::types::unidirectional::ALR`	Plug	Output 8 – Duplicated Signal

Functionality

The function block forwards the ALR signal received via socket **IN** unchanged to all eight output adapters **OUT1** to **OUT8**. No data processing or time delay occurs – the signals are passed through directly. This allows multiple subsequent function blocks to be supplied with the same signal in parallel.

Technical Features

- **Generic Type:** The function block is implemented as a generic function block (`GenericClassName = 'GEN_ALR_SPLIT'`), so the underlying ALR type can be specified during instantiation.
- **Adapter-Based:** All communication occurs via adapters, ensuring loose coupling and easy reuse.

State Overview

The function block has no events and no internal state machine. Execution is strictly data flow-driven: If the signal at the input changes, it is immediately propagated to all outputs.

Application Scenarios

- **Signal Splitting** in control architectures when an ALR signal is needed multiple times (e.g., for parallel evaluations or redundancy).
- **Interconnecting multiple subsystems** with identical control data, without multiple signal generation.
- **Test setups** for simultaneously stimulating multiple receivers with the same value.

Comparison with Similar Function Blocks

Other splitter function blocks (e.g., SPLIT_4 or SPLIT_16) differ only in the number of outputs. The ALR_SPLIT_8 is specifically designed for the ALR adapter type, while generic splitters (e.g., ANY_SPLIT) can be used for any data type. This component provides an optimized, type-safe solution for ALR distribution applications.

Change Detection

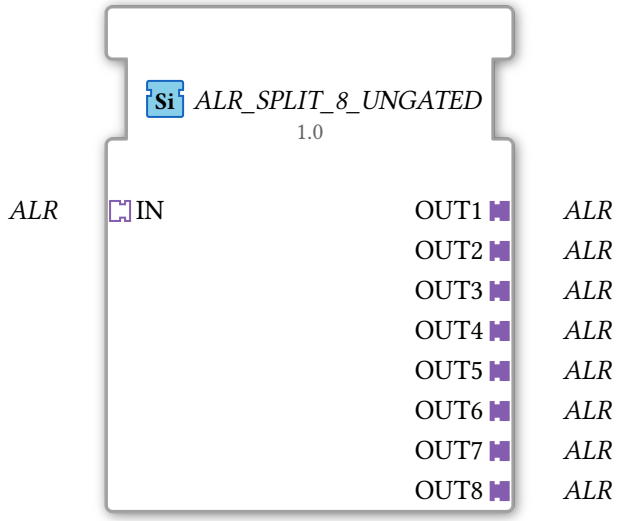
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ALR_SPLIT_8** is a simple yet effective functional block for multiplying an ALR signal to eight outputs. Its generic nature, adapter-based interface, and passive operation make it a robust and flexible component for IEC 61499 applications.

ALR_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of ALR_SPLIT_8. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALR_SPLIT_8_UNGATED** is used to split a single ALR signal (via a socket) into eight identical ALR outputs (plugs). This block is designed as a generic function block and enables signal distribution within an IEC 61499-based controller.

Interface Structure

Event Inputs

No direct event inputs are available. Control is integrated into the input adapter.

Event Outputs

No direct event outputs are available. Control is integrated into the output adapters.

Data Inputs

No direct data inputs are available. Data is received via the input adapter.

Data Outputs

No direct data outputs are available. Data is provided via the output adapters.

Adapter

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::ALR`	Socket	Input signal that is routed to all outputs
OUT1	`adapter::types::unidirectional::ALR`	Plug	Output 1 – duplicated signal
OUT2	`adapter::types::unidirectional::ALR`	Plug	Output 2 – duplicated signal
OUT3	`adapter::types::unidirectional::ALR`	Plug	Output 3 – duplicated signal
OUT4	`adapter::types::unidirectional::ALR`	Plug	Output 4 – duplicated signal
OUT5	`adapter::types::unidirectional::ALR`	Plug	Output 5 – duplicated signal
OUT6	`adapter::types::unidirectional::ALR`	Plug	Output 6 – Duplicated Signal
OUT7	`adapter::types::unidirectional::ALR`	Plug	Output 7 – Duplicated Signal
OUT8	`adapter::types::unidirectional::ALR`	Plug	Output 8 – Duplicated Signal

Functionality

The function block forwards the ALR signal received via socket **IN** unchanged to all eight output adapters **OUT1** to **OUT8**. No data processing or time delay occurs – the signals are passed through directly. This allows multiple subsequent function blocks to be supplied with the same signal in parallel.

Technical Features

- **Generic Type:** The function block is implemented as a generic function block (`GenericClassName = 'GEN_ALR_SPLIT'`), so the underlying ALR type can be specified during instantiation.
- **Adapter-Based:** All communication occurs via adapters, ensuring loose coupling and easy reuse.

State Overview

The function block has no events and no internal state machine. Execution is strictly data flow-driven: If the signal at the input changes, it is immediately propagated to all outputs.

Application Scenarios

- **Signal Splitting** in control architectures when an ALR signal is needed multiple times (e.g., for parallel evaluations or redundancy).
- **Interconnecting multiple subsystems** with identical control data, without multiple signal generation.
- **Test setups** for simultaneously stimulating multiple receivers with the same value.

Comparison with Similar Function Blocks

Other splitter function blocks (e.g., SPLIT_4 or SPLIT_16) differ only in the number of outputs. The ALR_SPLIT_8_UNGATED is specifically designed for the ALR adapter type, while generic splitters (e.g., ANY_SPLIT) can be used for any data type. This component provides an optimized, type-safe solution for ALR distribution applications.

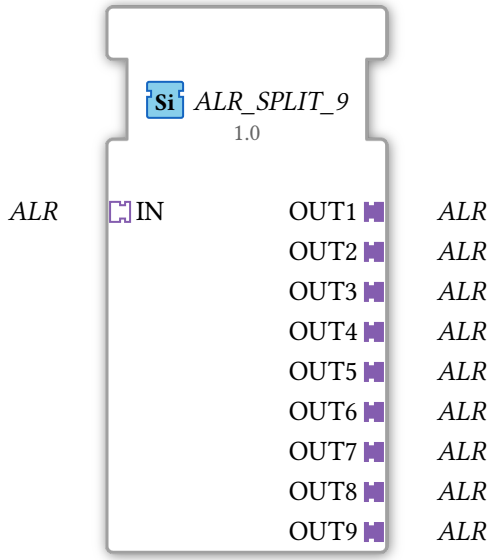
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **ALR_SPLIT_8_UNGATED** is a simple yet effective functional block for multiplying an ALR signal to eight outputs. Its generic nature, adapter-based interface, and passive operation make it a robust and flexible component for IEC 61499 applications.

ALR_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The **ALR_SPLIT_9** is a generic function block that splits an incoming ALR signal (unidirectional adapter) into exactly nine identical outputs. It functions as a 1:9 splitter for the adapter type `ALR` from the package `adapter::types::unidirectional`.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Name	Type
Socket	`IN`	`ALR` (unidirectional) – The input that is distributed to the nine outputs.
Plug	`OUT1` ... `OUT9`	`ALR` (unidirectional) – The nine outputs to which the incoming signal is provided.

Functionality

The module forwards the ALR signal present at socket `IN` unchanged and without delay to all nine plugs (`OUT1` to `OUT9`). No processing, conversion, or state logic takes place – the function block (FB) implements pure signal distribution at the adapter level.

Technical Features

- The FB is generic and has the attributes `GenericClassName` (value: `'GEN_ALR_SPLIT'`) and `TypeHash`.
- It uses only adapter interfaces; no events or input/output data are defined.
- Since no algorithm or state machine is implemented, the FB operates passively and efficiently.

State Overview

The FB has no state machine because it has no internal logic or memory behavior.

Application Scenarios

- **Signal Distribution:** An ALR event generated by a controller should reach multiple actuators or components simultaneously.
- **Redundant Control:** Several identical devices must receive the same output signal (e.g., parallel hydraulic valves in agricultural machinery).
- **Monitoring:** A signal should be passed to a monitoring unit and simultaneously to the actual device without interference.

Comparison with Similar Function Blocks

- **ALR_SPLIT_4 / ALR_SPLIT_8:** Analog function blocks with a different number of outputs.
- **Data Type Splitters (e.g., `SPLIT_INT`):** These operate at the data level, while `ALR_SPLIT_9` operates at the adapter level (signal/event forwarding).
- **Adapter Multiplexers/Demultiplexers:** Unlike these, `ALR_SPLIT_9` does not select or combine signals, but simply forwards the incoming signal 1:9.

Change Detection

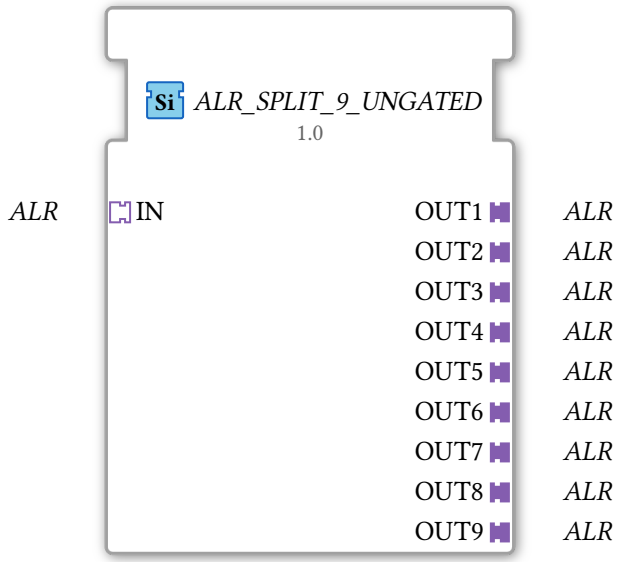
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ALR_SPLIT_9** is a simple yet practical function block for distributing a unidirectional ALR signal across nine parallel paths. It is easy to understand, requires no resources for logic or states, and can be directly integrated into control programs that require signal duplication.

ALR_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **ALR_SPLIT_9_UNGATED** is a generic function block that splits an incoming ALR signal (unidirectional adapter) into exactly nine identical outputs. It functions as a 1:9 splitter for the adapter type `ALR` from the package `adapter::types::unidirectional`.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Name	Type
Socket	`IN`	`ALR` (unidirectional) – The input that is distributed to the nine outputs.

Plug	`OUT1` ... `OUT9`	`ALR` (unidirectional) – The nine outputs to which the incoming signal is provided.

Functionality

The module forwards the ALR signal present at socket `IN` unchanged and without delay to all nine plugs (`OUT1` to `OUT9`). No processing, conversion, or state logic takes place – the function block (FB) implements pure signal distribution at the adapter level.

Technical Features

- The FB is generic and has the attributes `GenericClassName` (value: `'GEN_ALR_SPLIT'`) and `TypeHash`.
- It uses only adapter interfaces; no events or input/output data are defined.
- Since no algorithm or state machine is implemented, the FB operates passively and efficiently.

State Overview

The FB has no state machine because it has no internal logic or memory behavior.

Application Scenarios

- **Signal Distribution:** An ALR event generated by a controller should reach multiple actuators or components simultaneously.
- **Redundant Control:** Several identical devices must receive the same output signal (e.g., parallel hydraulic valves in agricultural machinery).
- **Monitoring:** A signal should be passed to a monitoring unit and simultaneously to the actual device without interference.

Comparison with Similar Function Blocks

- **ALR_SPLIT_4 / ALR_SPLIT_8:** Analog function blocks with a different number of outputs.
- **Data Type Splitters (e.g., `SPLIT_INT`):** These operate at the data level, while `ALR_SPLIT_9_UNGATED` operates at the adapter level (signal/event forwarding).
- **Adapter Multiplexers/Demultiplexers:** Unlike these, `ALR_SPLIT_9_UNGATED` does not select or combine signals, but simply forwards the incoming signal 1:9.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

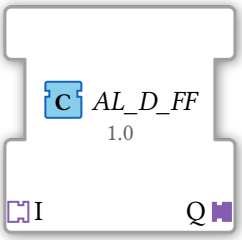
The **ALR_SPLIT_9_UNGATED** is a simple yet practical function block for distributing a unidirectional ALR signal across nine parallel paths. It is easy to understand, requires no resources for logic or states, and can be directly integrated into control programs that require signal duplication.

Numeric Index (unidirectional)

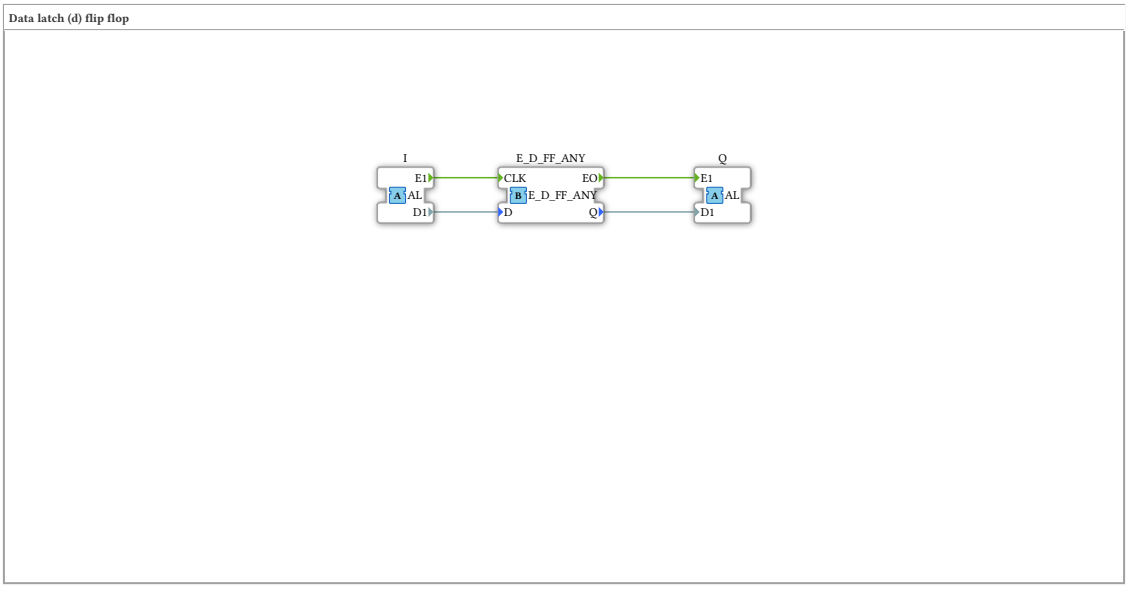
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AL_D_FF

Value to latch – *AL*



AL – Latched value



• – ○ • – ○ • – ○ •

Introduction

The function block **AL_D_FF** implements a data latch flip-flop (D flip-flop). It serves to store a digital data value upon each incoming event and make the stored value available via an adapter output. The block encapsulates the recurring task of a D flip-flop and provides a clean, typed interface for event and data exchange via adapters.

Interface Structure

Event Inputs

No direct event inputs. The event for transferring the data value is provided via the adapter socket **I** (E1).

Event Outputs

No direct event outputs. The output event is provided via the adapter plug **Q** (E1).

Data Inputs

No direct data inputs. The data value to be stored is supplied via the adapter socket **I** (D1).

Data Outputs

No direct data outputs. The stored data value is output via the adapter plug **Q** (D1).

Adapters

Name	Type	Direction	Comment
I	`adapter::types::unidirectional::AL`	Socket	Value to be latched
Q	`adapter::types::unidirectional::AL`	Plug	Latched value

The adapters are unidirectional and of type `AL`. Each adapter contains an event input/output (E1) and a data input/output (D1). Socket **I** serves as the input interface, and plug **Q** as the output interface.

Functionality

The function block internally uses the standard function block `iec61499::events::E_D_FF_ANY`.

Upon each event at the adapter input **I** (E1), the data value from **I.D1** is taken and stored at the adapter output **Q.D1**. Simultaneously, an event is generated at output **Q.E1** to signal the availability of the new value to the downstream logic.

The takeover is edge-triggered (rising edge) – the behavior corresponds to a clocked D-latch or D-flip-flop, depending on the definition of the internally used function block.

Technical Features

- **Adapter-based interface**: Instead of individual inputs/outputs, the entire signal group (event + data) is combined via an adapter. This simplifies the connection in the application and ensures clear semantic encapsulation.
- **Reusing Existing Logic**: The function block delegates the core function to the IEC 61499 standard function block `E_D_FF_ANY`, thus using a proven and portable implementation.
- **Type Safety**: By using a defined adapter type (`AL`), it is ensured that only compatible signals can be connected.

State Overview

The function block does not have its own ECC state machine. Its internal behavior is entirely determined by the embedded function block `E_D_FF_ANY`. This implements a D flip-flop with two logical states:

- **State 0**: The stored value is `false` (or the adapter's data value is 0).
- **State 1**: The stored value is `true` (or the adapter's data value is 1).

A state change occurs only upon an event at **I.E1** the value present at **I.D1** is then adopted.

Application Scenarios

- **Signal Delay and Synchronization**: A data value should only be passed on upon a defined event (e.g., clock edge).
- **Storage of a Binary State**: e.g., as part of a shift register, a state register, or marker logic.
- **Coupling with Adapter-Based Components**: Easy integration into applications that already use the adapter type `adapter::types::unidirectional::AL`.

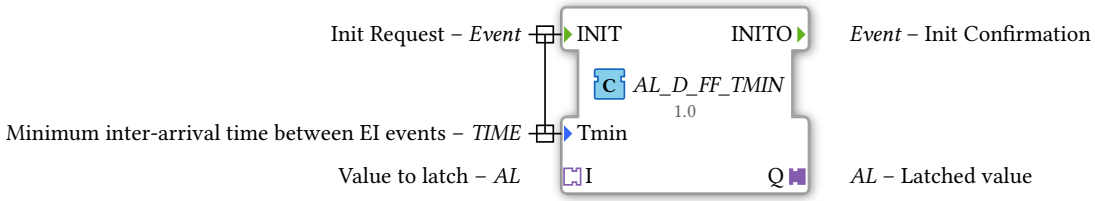
Comparison with Similar Components

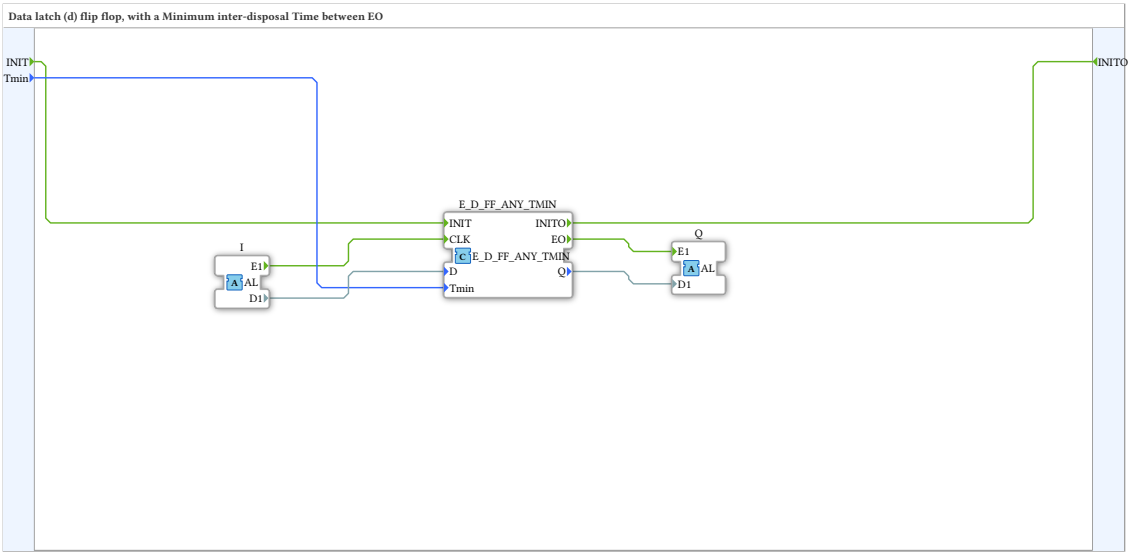
- **E_D_FF_ANY**: Standard D flip-flop with single event and data inputs. `AL_D_FF` encapsulates this function block and provides an adapter interface.
- **SR_FF or JK_FF**: Other flip-flop types with multiple control inputs. `AL_D_FF` is limited to basic D functionality.
- **AL_D_LATCH**: A simple latch without event control? (Not available). `AL_D_FF` operates on an edge and is therefore suitable for clock-synchronous systems.

Conclusion

The function block `AL_D_FF` offers an encapsulated, adapter-based solution for basic D flip-flop functionality. It simplifies the handling of signal and event groups in IEC 61499 applications, increases reusability, and remains reliable and traceable through traceability to a standardized core component.

AL_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The AL_D_FF_TMIN function block implements an edge-triggered D flip-flop (data latch) with an adjustable minimum inter-disposal time. It receives a data value upon an incoming event, stores it, and outputs it—along with its own event—only after the specified minimum time has elapsed. This limits the output event rate.

Interface Structure

Event Inputs

- **INIT**: Initialization request; sets the parameter Tmin for the minimum time between output events.

Event Outputs

- **INITO**: Confirmation of successful initialization.

Data Inputs

- **Tmin** (TIME): Minimum time interval that must be maintained between two consecutive output events (EO).

Data Outputs

- No direct data outputs – the latched values are transmitted exclusively via the output adapter Q.

Adapters

- **I** (Socket, Type: `adapter::types::unidirectional::AL`): Input adapter for the value to be latched. The latch pulse is received via the event interface (E1), and the data value to be latched is provided via the data interface (D1).
- **Q** (Plug, Type: `adapter::types::unidirectional::AL`): Output adapter for the stored value. An event (E1) is sent and the currently stored data value (D1) is output for each successful latch operation.

Functionality

The module internally uses the predefined flip-flop module `E_D_FF_ANY_TMIN`.

As soon as an event arrives at the input adapter (I.E1), the simultaneously present data value (I.D1) is received and stored internally. However, the output event (Q.E1) is only generated if at least the time interval specified by Tmin has elapsed since the last output event. Only then is the stored value passed on to the output adapter (Q.D1). Initialization with INIT sets the parameter Tmin and must be performed before first use.

Technical Features

- **Minimum time between output events**: The delay Tmin limits the frequency of output events – useful for preventing bus overloads or limiting the actuator control rate.
- **Adapter Interfaces**: Coupling is achieved via standardized, unidirectional adapters (`adapter::types::unidirectional::AL`). This allows the function block to be flexibly integrated into various applications.
- **Edge-Triggered**: The internal flip-flop reacts to the rising edge of the event at the input adapter (I.E1).

State Overview

The function block does not have an explicitly displayed state machine. Its behavior is determined by the internal D flip-flop with time control:

- After successful initialization (INIT ⟶ INITO), the function block waits for an event at the input adapter I.
- Upon an event, the incoming data value is latched. Immediately afterward, it is checked whether the time elapsed since the last output event has already exceeded Tmin.
- If yes: The output event is generated immediately.
- If no: The output is blocked until the waiting time has expired; then the output is generated.
- Events received during the wait time may be ignored or buffered (depending on the internal implementation of the underlying function block) – typically, only the most recent data value is considered for the next output.

Application Scenarios

- **Synchronization of time-critical controllers** where data must not be updated more frequently than at a predefined interval.
- **Debouncing of signals** or throttling of event sequences in communication networks.
- **Simple state machines** that react to external events and must maintain a minimum response time (e.g., in automation technology or the control of physical actuators).

Comparison with Similar Function Blocks

- **AL_D_FF**: A simple D flip-flop without a minimum time between output events – AL_D_FF_TMIN extends this with the Tmin control and adapter interfaces.

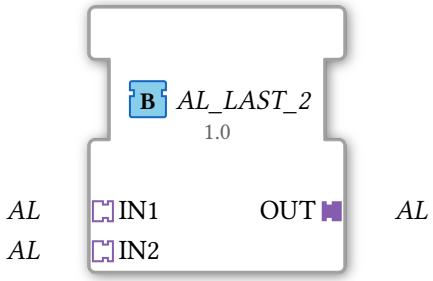
AL_D_FF_TMIN - **E_D_FF_ANY_TMIN**: The underlying generic flip-flop that can be used directly (without adapter encapsulation). AL_D_FF_TMIN encapsulates this in a plug-and-play adapter format.

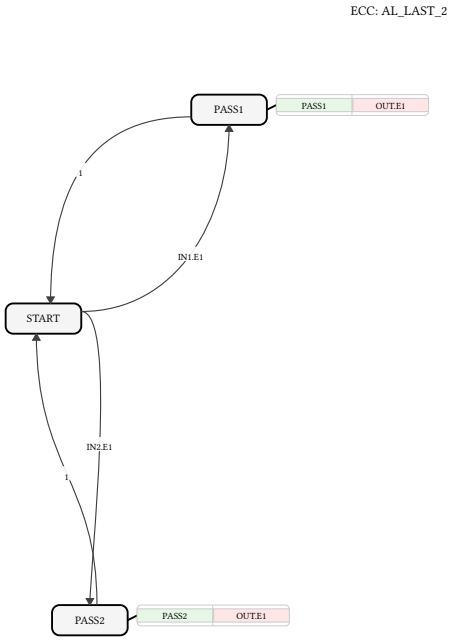
- **R_TRIG / F_TRIG**: Detect rising and falling edges, respectively, but do not store data values and do not offer an adjustable output delay.

Conclusion

The AL_D_FF_TMIN function block provides a reliable, edge-triggered D-latch function with an adjustable minimum time between output events. By using standardized adapter interfaces, it is easily integrated into IEC 61499 applications and is particularly suitable for applications that require limiting the event rate or time-controlled data release.

AL_LAST_2





Introduction

The AL_LAST_2 function block merges two same-type LWORD adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: LWORD adapter (unidirectional) - first signal source
- **IN2**: LWORD adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: LWORD adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

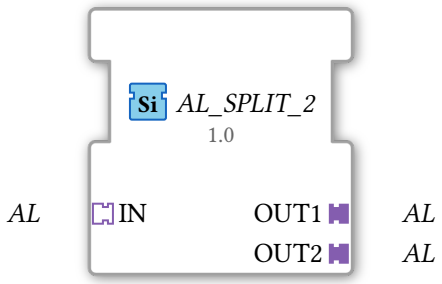
Application Scenarios

- Merging two equally-valid LWORD signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same LWORD value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AL_LAST_2 is the inverse of `AL_SPLIT_2`: AL_SPLIT_2 distributes 1 input to 2 outputs, AL_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

AL_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_2** is used to split an incoming unidirectional adapter signal into two identical outputs. It is designed as a generic function block, so the specific adapter type is only determined during use.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Type	Direction	Name	Description
`adapter::types::unidirectional::AL`	**Socket** (Input)	`IN`	Input signal distributed to both outputs
`adapter::types::unidirectional::AL`	**Plug** (Output)	`OUT1`	First output with the same adapter signal as `IN`
`adapter::types::unidirectional::AL`	**Plug** (Output)	`OUT2`	Second output with the same adapter signal as `IN`

Functionality

The function block forwards the unidirectional adapter signal present at socket `IN` unchanged to both plugs `OUT1` and `OUT2`. No data processing or transformation takes place – the function block operates purely passively and by passing the signal through. As soon as a signal is present at the input, it is simultaneously available at both outputs.

Technical Features

- **Generic Type**: The function block is defined as a generic function block (attributes `GenericClassName` and `TypeHash`). This allows it to be used with different unidirectional adapter instances without requiring any modifications to the source code.
- **Adapters instead of Data Points**: Signal transmission occurs exclusively via adapters, not via discrete event or data inputs/outputs. This enables structured and reusable encapsulation of communication interfaces.
- **No State Dependency**: The function block has no internal memory and no state machine. The output signals follow the input signal directly.

State Overview

The function block does not have its own state management. It can be considered a **combinatorial** block where the outputs are always assigned the current input value. There is no explicit state machine.

Application Scenarios

- **Signal Distribution**: When an adapter signal needs to be passed from one source to multiple independent receivers (e.g., a control command to two actuators).
- **Logging / Monitoring**: An adapter signal can be split into a path for actual processing and simultaneously into a monitoring or logging path.
- **Modular Architectures**: In complex systems that use adapters, the `AL_SPLIT_2` simplifies the topological distribution of signals without requiring manual connections or duplicates.

Comparison with Similar Components

The 4diac IDE offers various splitter components for different signal types:

Component	Signal Type	Outputs	Special Feature
`AL_SPLIT_2`	Adapter (unidirectional)	2	Generic, no data/events
`SPLIT`	Events	2	Distributes events without data
`E_SPLIT`	Events (with data)	2	Distributes events along with accompanying data
`F_SPLIT`	Data (any type)	n	Distributes a data value to multiple outputs

The `AL_SPLIT_2` is specifically optimized for adapter interfaces, providing a clean way to split complex, adapter-based connections without interfering with the data or event layer.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AL_SPLIT_2` is a simple yet useful generic function block for splitting unidirectional adapter signals. It enables flexible and structured signal distribution in modular automation solutions without additional logic or state dependencies. Its generic nature makes it universally applicable.

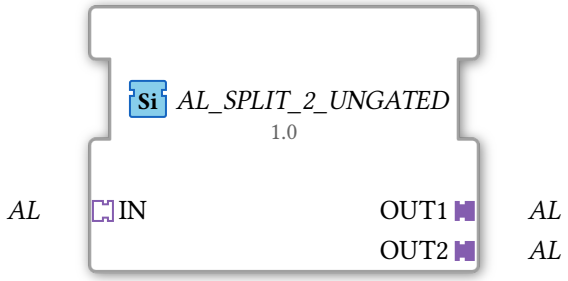
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AL_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_2_UNGATED** is used to split an incoming unidirectional adapter signal into two identical outputs. It is designed as a generic function block, so the specific adapter type is only determined during use.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Type	Direction	Name	Description
`adapter::types::unidirectional::AL`	**Socket** (Input)	`IN`	Input signal distributed to both outputs
`adapter::types::unidirectional::AL`	**Plug** (Output)	`OUT1`	First output with the same adapter signal as `IN`
`adapter::types::unidirectional::AL`	**Plug** (Output)	`OUT2`	Second output with the same adapter signal as `IN`

Functionality

The function block forwards the unidirectional adapter signal present at socket `IN` unchanged to both plugs `OUT1` and `OUT2`. No data processing or transformation takes place – the function block operates purely passively and by passing the signal through. As soon as a signal is present at the input, it is simultaneously available at both outputs.

Technical Features

- **Generic Type**: The function block is defined as a generic function block (attributes `GenericClassName` and `TypeHash`). This allows it to be used with different unidirectional adapter instances without requiring any modifications to the source code.
- **Adapters instead of Data Points**: Signal transmission occurs exclusively via adapters, not via discrete event or data inputs/outputs. This enables structured and reusable encapsulation of communication interfaces.
- **No State Dependency**: The function block has no internal memory and no state machine. The output signals follow the input signal directly.

State Overview

The function block does not have its own state management. It can be considered a **combinatorial** block where the outputs are always assigned the current input value. There is no explicit state machine.

Application Scenarios

- **Signal Distribution**: When an adapter signal needs to be passed from one source to multiple independent receivers (e.g., a control command to two actuators).
- **Logging / Monitoring**: An adapter signal can be split into a path for actual processing and simultaneously into a monitoring or logging path.
- **Modular Architectures**: In complex systems that use adapters, the `AL_SPLIT_2_UNGATED` simplifies the topological distribution of signals without requiring manual connections or duplicates.

Comparison with Similar Components

The 4diac IDE offers various splitter components for different signal types:

Component	Signal Type	Outputs	Special Feature
`AL_SPLIT_2_UNGATED`	Adapter (unidirectional)	2	Generic, no data/events
`SPLIT`	Events	2	Distributes events without data
`E_SPLIT`	Events (with data)	2	Distributes events along with accompanying data
`F_SPLIT`	Data (any type)	n	Distributes a data value to multiple outputs

The `AL_SPLIT_2_UNGATED` is specifically optimized for adapter interfaces, providing a clean way to split complex, adapter-based connections without interfering with the data or event layer.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

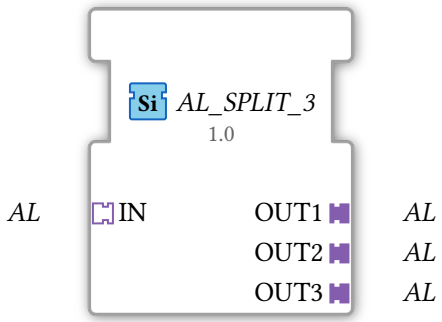
The `AL_SPLIT_2_UNGATED` is a simple yet useful generic function block for splitting unidirectional adapter signals. It enables flexible and structured signal distribution in modular automation solutions without additional logic or state dependencies. Its generic nature makes it universally applicable.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AL_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_3** is used to split an incoming unidirectional adapter signal (AL – Adapter Label) into three identical output adapters. It is designed as a generic building block and enables the multiple forwarding of an adapter-based data or event flow within an IEC 61499 application.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AL`	IN	Socket	Input adapter that provides the signal to be distributed.
`adapter::types::unidirectional::AL`	OUT1	Plug	First output adapter – identical copy of the input signal.
`adapter::types::unidirectional::AL`	OUT2	Plug	Second output adapter – identical copy of the input signal.
`adapter::types::unidirectional::AL`	OUT3	Plug	Third output adapter – identical copy of the input signal.

Functionality

The function block (FB) forwards the unidirectional adapter signal present at socket **IN unchanged and without delay** to the three plugs **OUT1**, **OUT2**, and **OUT3**. No logical or timing manipulation takes place – the distribution is purely structural. This behavior corresponds to passive wiring (“fan-out”) at the adapter level.

Technical Features

- **Generic Implementation:** The function block uses the generic class name `'GEN_AL_SPLIT'`, allowing it to be reused for various instances of the unidirectional AL type, as long as the specific type is defined at design time.
- **No State Logic:** There are no internal states, events, or data variables. The FB is completely passive and requires no active processing.
- **No Runtime Dependencies:** The functionality is implemented at design time by the interconnection system of the 4diac IDE.

State Overview

The function block (FB) has **no internal state** and no state machine. It does not execute any sequential processes. The output signal is always a direct copy of the input signal.

Application Scenarios

- **Multiple Use of an Adapter:** A unidirectional AL signal provided by a component is to be distributed to several subsequent function blocks (e.g., a sensor signal to different evaluation units).
- **Architectural Structuring:** Splitting a data stream for parallel processing or monitoring without the source component needing to know the number of sinks.
- **Testing and Diagnostic Purposes:** Connecting additional monitoring or logging blocks to an existing adapter connection.

Comparison with Similar Function Blocks

- **Event-Based Splitters (e.g., E_SPLIT):** These split events, while AL_SPLIT_3 distributes adapter objects (complex data structures). Event splitters often require handling of event sequences.
- **Adapter Merger (e.g., AL_MERGE):** Combines multiple adapter signals into one – the opposite of the function described here.
- **Individual AL Splitters:** Generic versions also exist for other output numbers (e.g., AL_SPLIT_2, AL_SPLIT_4), differing only in the number of plugs.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AL_SPLIT_3 is a simple yet useful generic function block for duplicating a unidirectional adapter signal to three independent outputs. Due to its passive nature, it is resource-efficient and ideally suited for the flexible design of IEC 61499 applications where a signal is needed multiple times without altering the source logic.

AL_SPLIT_3 –

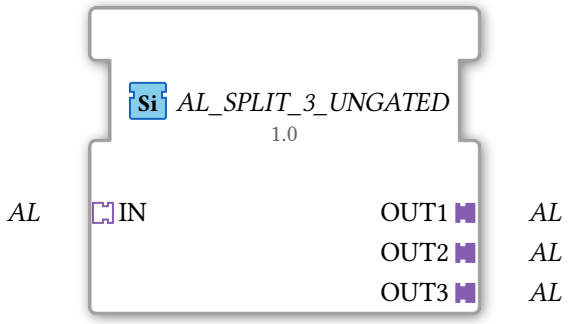
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AL_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_3_UNGATED** is used to split an incoming unidirectional adapter signal (AL – Adapter Label) into three identical output adapters. It is designed as a generic building block and enables the multiple forwarding of an adapter-based data or event flow within an IEC 61499 application.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AL`	IN	Socket	Input adapter that provides the signal to be distributed.
`adapter::types::unidirectional::AL`	OUT1	Plug	First output adapter – identical copy of the input signal.
`adapter::types::unidirectional::AL`	OUT2	Plug	Second output adapter – identical copy of the input signal.
`adapter::types::unidirectional::AL`	OUT3	Plug	Third output adapter – identical copy of the input signal.

Functionality

The function block (FB) forwards the unidirectional adapter signal present at socket **IN unchanged and without delay** to the three plugs **OUT1**, **OUT2**, and **OUT3**. No logical or timing manipulation takes place – the distribution is purely structural. This behavior corresponds to passive wiring ("fan-out") at the adapter level.

Technical Features

- **Generic Implementation:** The function block uses the generic class name `'GEN_AL_SPLIT'`, allowing it to be reused for various instances of the unidirectional AL type, as long as the specific type is defined at design time.
- **No State Logic:** There are no internal states, events, or data variables. The FB is completely passive and requires no active processing.
- **No Runtime Dependencies:** The functionality is implemented at design time by the interconnection system of the 4diac IDE.

State Overview

The function block (FB) has **no internal state** and no state machine. It does not execute any sequential processes. The output signal is always a direct copy of the input signal.

Application Scenarios

- **Multiple Use of an Adapter:** A unidirectional AL signal provided by a component is to be distributed to several subsequent function blocks (e.g., a sensor signal to different evaluation units).
- **Architectural Structuring:** Splitting a data stream for parallel processing or monitoring without the source component needing to know the number of sinks.
- **Testing and Diagnostic Purposes:** Connecting additional monitoring or logging blocks to an existing adapter connection.

Comparison with Similar Function Blocks

- **Event-Based Splitters (e.g., E_SPLIT):** These split events, while AL_SPLIT_3_UNGATED distributes adapter objects (complex data structures). Event splitters often require handling of event sequences.
- **Adapter Merger (e.g., AL_MERGE):** Combines multiple adapter signals into one – the opposite of the function described here.
- **Individual AL Splitters:** Generic versions also exist for other output numbers (e.g., AL_SPLIT_2, AL_SPLIT_4), differing only in the number of plugs.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AL_SPLIT_3_UNGATED is a simple yet useful generic function block for duplicating a unidirectional adapter signal to three independent outputs. Due to its passive nature, it is resource-efficient and ideally suited for the flexible design of IEC 61499 applications where a signal is needed multiple times without altering the source logic.

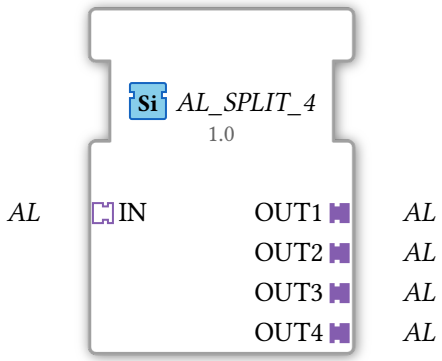
AL_SPLIT_3_UNGATED –

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AL_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_4** is a generic function block that splits a single adapter input (socket) of type `adapter::types::unidirectional::AL` into four separate outputs (plugs) of the same type. It serves to distribute an incoming adapter data stream to multiple parallel consumers without modifying the data itself.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AL`	**IN**	Socket (Input)	Incoming adapter data stream, distributed to all outputs.
`adapter::types::unidirectional::AL`	**OUT1**	Plug (Output)	First output – receives an identical copy of the input data stream.
`adapter::types::unidirectional::AL`	**OUT2**	Plug (Output)	Second output – receives an identical copy of the input data stream.

adapter::types::unidirectional::AL	**OUT3**	Plug (Output)	Third output – receives an identical copy of the input data stream.
adapter::types::unidirectional::AL	**OUT4**	Plug (Output)	Fourth output – receives an identical copy of the input data stream.

Functionality

This component is a pure signal distributor. It has internal logic that – as soon as the adapter input **IN** is connected to a data source – forwards the incoming data packets (events and data) directly and unchanged to all four output adapters **OUT1** to **OUT4**. No buffering, delay, or logical processing takes place. Each output receives exactly the same information as the input.

Technical Features

- **Generic Component:** The class `GEN_AL_SPLIT` is defined in the attribute `eclipse4diac::core::GenericClassName`, which means that the component can be reused as a template for various implementations of the adapter type `AL`.
- **No states or timing:** Since the function block operates exclusively on a line-based basis and has no processing logic, there are no internal states or concurrent execution sequences.
- **Adapter type:** The adapter used, `adapter::types::unidirectional::AL`, is a standardized, unidirectional adapter from the library `adapter::types::unidirectional`. Its exact interface (included events and data) is defined by the adapter definition and is not influenced by the split block itself.

State overview

The function block does not have a state machine. It operates statically and permanently distributes the input signal to all outputs.

Application scenarios

- **Distributing a control signal** to multiple parallel actuator units, e.g., in agricultural technology (as indicated in the copyright notice of HR Agrartechnik GmbH).
- **Splitting a measurement data source** across multiple analysis or logging modules.
- **Testing and Debugging:** An original adapter signal can be routed to multiple outputs to be sent to different test instances simultaneously.

Comparison with Similar Function Blocks

- **AL_SPLIT_2** – splits an AL signal across two outputs (fewer outputs, lower fan-out capacity).
- **AL_MERGE** – combines multiple AL inputs into one output (inverse function).
- **Event-based split function blocks (e.g., SPLIT_EVENT)** – split only events without duplicating the associated data adapters.

Unlike these function blocks, `AL_SPLIT_4` operates exclusively at the adapter level and copies complete adapter communication streams.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

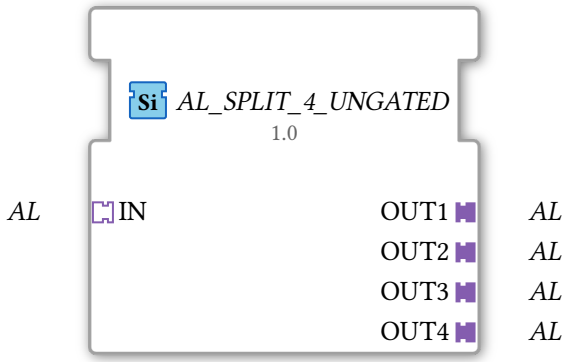
The **AL_SPLIT_4** function block is a simple, efficient signal fan for the `AL` adapter type. It is ideally suited for all applications where a single adapter data stream needs to be distributed to multiple consumers without additional logic or delays. Its generic nature allows its use in various automation contexts, particularly in agricultural machinery control, as intended by the manufacturer.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AL_SPLIT_4_UNGATED

i UNGATED variant: This block is the ungated version of `AL_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_4_UNGATED** is a generic function block that splits a single adapter input (socket) of type `adapter::types::unidirectional::AL` into four separate outputs (plugs) of the same type. It serves to distribute an incoming adapter data stream to multiple parallel consumers without modifying the data itself.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AL`	**IN**	Socket (Input)	Incoming adapter data stream, distributed to all outputs.
`adapter::types::unidirectional::AL`	**OUT1**	Plug (Output)	First output – receives an identical copy of the input data stream.
`adapter::types::unidirectional::AL`	**OUT2**	Plug (Output)	Second output – receives an identical copy of the input data stream.
`adapter::types::unidirectional::AL`	**OUT3**	Plug (Output)	Third output – receives an identical copy of the input data stream.

adapter::types::unidirectional::AL	**OUT4**	Plug (Output)	Fourth output – receives an identical copy of the input data stream.

Functionality

This component is a pure signal distributor. It has internal logic that – as soon as the adapter input **IN** is connected to a data source – forwards the incoming data packets (events and data) directly and unchanged to all four output adapters **OUT1** to **OUT4**. No buffering, delay, or logical processing takes place. Each output receives exactly the same information as the input.

Technical Features

- **Generic Component:** The class `GEN_AL_SPLIT` is defined in the attribute `eclipse4diac::core::GenericClassName`, which means that the component can be reused as a template for various implementations of the adapter type `AL`.
- **No states or timing:** Since the function block operates exclusively on a line-based basis and has no processing logic, there are no internal states or concurrent execution sequences.
- **Adapter type:** The adapter used, `adapter::types::unidirectional::AL`, is a standardized, unidirectional adapter from the library `adapter::types::unidirectional`. Its exact interface (included events and data) is defined by the adapter definition and is not influenced by the split block itself.

State overview

The function block does not have a state machine. It operates statically and permanently distributes the input signal to all outputs.

Application scenarios

- **Distributing a control signal** to multiple parallel actuator units, e.g., in agricultural technology (as indicated in the copyright notice of HR Agrartechnik GmbH).
- **Splitting a measurement data source** across multiple analysis or logging modules.
- **Testing and Debugging:** An original adapter signal can be routed to multiple outputs to be sent to different test instances simultaneously.

Comparison with Similar Function Blocks

- **AL_SPLIT_2** – splits an AL signal across two outputs (fewer outputs, lower fan-out capacity).
- **AL_MERGE** – combines multiple AL inputs into one output (inverse function).
- **Event-based split function blocks (e.g., SPLIT_EVENT)** – split only events without duplicating the associated data adapters.

Unlike these function blocks, `AL_SPLIT_4_UNGATED` operates exclusively at the adapter level and copies complete adapter communication streams.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

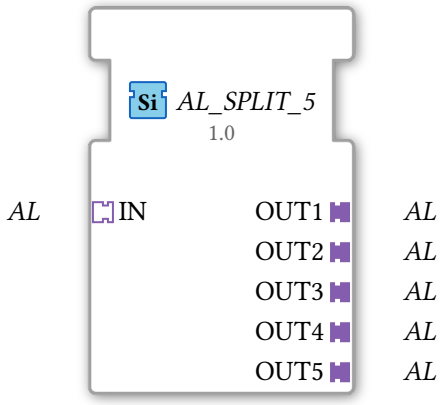
Conclusion

The **AL_SPLIT_4_UNGATED** function block is a simple, efficient signal fan for the `AL` adapter type. It is ideally suited for all applications where a single adapter data stream needs to be distributed to multiple consumers without additional logic or delays. Its generic nature allows its use in various automation contexts, particularly in agricultural machinery control, as intended by the manufacturer.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AL_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_5** serves as a generic splitter for a unidirectional adapter of type `AL`. It splits an incoming signal (via the `IN` socket) into five separate outputs (`OUT1` to `OUT5`). The block is marked as generic, which allows the specific adapter type to be defined when using it.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Type	Direction	Name	Description
Socket	Input	IN	Unidirectional `AL` adapter input
Plug	Output	OUT1	First output (unidirectional `AL`)
Plug	Output	OUT2	Second output (unidirectional `AL`)
Plug	Output	OUT3	Third output (unidirectional `AL`)
Plug	Output	OUT4	Fourth output (unidirectional `AL`)
Plug	Output	OUT5	Fifth output (unidirectional `AL`)

Functionality

The function block receives a signal of type `AL` via socket `IN` and forwards it unchanged to all five plugs `OUT1` to `OUT5`. No data processing or event handling takes place; the distribution occurs purely at the adapter level. This allows multiple downstream components to be supplied with the same adapter signal in parallel.

The function block is designed as a generic function block (`GEN_AL_SPLIT`). The specific adapter type `AL` can be defined via an attribute during instantiation.

Technical Features

- **No Events or Data**: The function block has only adapter interfaces, no event or data ports.
- **Generic Design**: The type of the `AL` adapter is configurable (generic class), allowing different implementations of the same adapter type to be used.
- **Pure Signal Distribution**: This is a pure 1:5 distribution without any logic, delay, or transformation.

State Overview

The function block does not have an internal state machine, as it does not process any events. It behaves statically and permanently passes the input signal to all outputs.

Application Scenarios

- **Signal Distribution**: If several components require the same adapter signal of a `AL` type, e.g., For example, to send a sensor value to different control logics.
- **Test and debugging environments**: An incoming adapter signal can be passed on to multiple monitor or logging blocks simultaneously.
- **Redundancy**: The signal can be distributed to parallel processing paths.

Comparison with similar blocks

Other splitter blocks exist in the 4diac ecosystem, such as `SPLIT` for event or data flows. The advantage of `AL_SPLIT_5` lies in its exclusive adapter interface. While a `SPLIT` block separates events or data, this block operates directly at the adapter level and requires no conversion between events and data. Similar function blocks with a different number of outputs (e.g., `AL_SPLIT_2`, `AL_SPLIT_3`) are conceivable, but are not included in this generic form by default.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AL_SPLIT_5** is a simple yet useful generic function block for duplicating a unidirectional adapter signal to five outputs. Due to its pure adapter interface and lack of event/data logic, it is particularly suitable for applications that need to distribute a signal to multiple receivers without processing. The generic design increases flexibility for reuse in different projects.

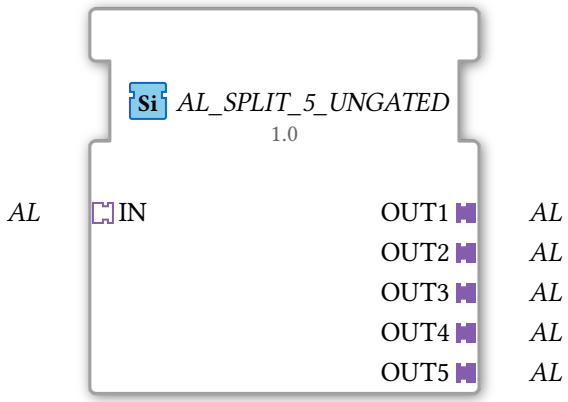
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AL_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_5_UNGATED** serves as a generic splitter for a unidirectional adapter of type `AL`. It splits an incoming signal (via the `IN` socket) into five separate outputs (`OUT1` to `OUT5`). The block is marked as generic, which allows the specific adapter type to be defined when using it.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Type	Direction	Name	Description
Socket	Input	IN	Unidirectional `AL` adapter input
Plug	Output	OUT1	First output (unidirectional `AL`)
Plug	Output	OUT2	Second output (unidirectional `AL`)
Plug	Output	OUT3	Third output (unidirectional `AL`)
Plug	Output	OUT4	Fourth output (unidirectional `AL`)
Plug	Output	OUT5	Fifth output (unidirectional `AL`)

Functionality

The function block receives a signal of type `AL` via socket `IN` and forwards it unchanged to all five plugs `OUT1` to `OUT5`. No data processing or event handling takes place; the distribution occurs purely at the adapter level. This allows multiple downstream components to be supplied with the same adapter signal in parallel.

The function block is designed as a generic function block (`GEN_AL_SPLIT`). The specific adapter type `AL` can be defined via an attribute during instantiation.

Technical Features

- **No Events or Data**: The function block has only adapter interfaces, no event or data ports.

- **Generic Design**: The type of the `AL` adapter is configurable (generic class), allowing different implementations of the same adapter type to be used.
- **Pure Signal Distribution**: This is a pure 1:5 distribution without any logic, delay, or transformation.

State Overview

The function block does not have an internal state machine, as it does not process any events. It behaves statically and permanently passes the input signal to all outputs.

Application Scenarios

- **Signal Distribution**: If several components require the same adapter signal of a `AL` type, e.g., For example, to send a sensor value to different control logics.
- **Test and debugging environments**: An incoming adapter signal can be passed on to multiple monitor or logging blocks simultaneously.
- **Redundancy**: The signal can be distributed to parallel processing paths.

Comparison with similar blocks

Other splitter blocks exist in the 4diac ecosystem, such as `SPLIT` for event or data flows. The advantage of `AL_SPLIT_5_UNGATED` lies in its exclusive adapter interface. While a `SPLIT` block separates events or data, this block operates directly at the adapter level and requires no conversion between events and data. Similar function blocks with a different number of outputs (e.g., `AL_SPLIT_2`, `AL_SPLIT_3`) are conceivable, but are not included in this generic form by default.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

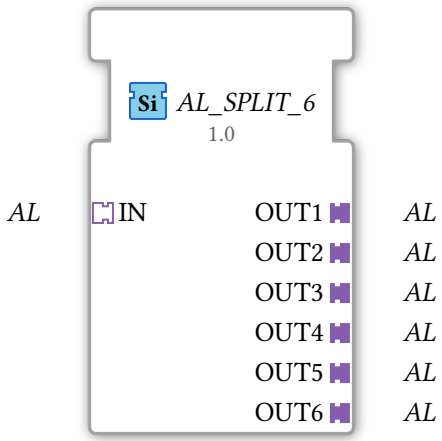
The **AL_SPLIT_5_UNGATED** is a simple yet useful generic function block for duplicating a unidirectional adapter signal to five outputs. Due to its pure adapter interface and lack of event/data logic, it is particularly suitable for applications that need to distribute a signal to multiple receivers without processing. The generic design increases flexibility for reuse in different projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AL_SPLIT_6



• − ◦ • − ◦ • − ◦ •

Introduction

The function block **AL_SPLIT_6** is used to split a single unidirectional AL adapter input (socket) into six identical AL adapter outputs (plugs). It enables simple signal distribution within a 4diac application without requiring additional logic or a state machine.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Direction	Identifier	Type	Description
In	IN	adapter::types::unidirectional::AL	Unidirectional AL adapter as input signal.
Out	OUT1	adapter::types::unidirectional::AL	First output channel (identical to IN).
Out	OUT2	adapter::types::unidirectional::AL	Second output channel.
Out	OUT3	adapter::types::unidirectional::AL	Third output channel.
Out	OUT4	adapter::types::unidirectional::AL	Fourth output channel.
Out	OUT5	adapter::types::unidirectional::AL	Fifth output channel.
Out	OUT6	adapter::types::unidirectional::AL	Sixth output channel.

Functionality

This function block has no internal logic or state machine. It forwards the AL signal arriving at socket **IN** unchanged to all six plugs **OUT1** … **OUT6**. Any change or event at the input is thus transmitted simultaneously to all outputs. The function block operates purely structurally as a splitter and does not require event activation.

Technical Features

- **Generic Function Block:** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `GEN_AL_SPLIT`. This allows it to be treated as a generic type in the 4diac IDE and enables clean type system integration.
- **No Data/Event Interface:** Distribution occurs exclusively via the adapter interface. No additional data or event ports are required.
- **No Runtime Dependencies:** Since no algorithms or states exist, the function block is resource-efficient and requires no specific initialization.

State Overview

The function block has no state machine (no ECC) and therefore no defined state machine. Its behavior is entirely determined by the interface structure and the simple forwarding of the adapter signal.

Application Scenarios

- **Signal Distribution in Agricultural Engineering or Automation:** A central AL adapter (e.g., a control command) is to be sent to several downstream components simultaneously.
- **Tests and Simulations:** A test signal can be distributed in parallel to different modules to verify system behavior.

- **Redundant Paths:** The same AL adapter can be routed to different processing units via multiple channels.

Comparison with Similar Function Blocks

- **AL_SPLIT_2, AL_SPLIT_4 … AL_SPLIT_N**: These function blocks differ only in the number of output channels. The functionality is identical – a simple 1:N split.
- **Other split blocks with events or data:** Unlike these, **AL_SPLIT_6** has no data or event interfaces, but works purely via adapters. This simplifies configuration but limits the variety of types.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AL_SPLIT_6** is a compact and efficient function block for multiplying a unidirectional AL adapter signal. Due to its generic nature and simple structure, it is ideally suited for all applications where an input signal needs to be distributed to multiple outputs without additional logic or event control.

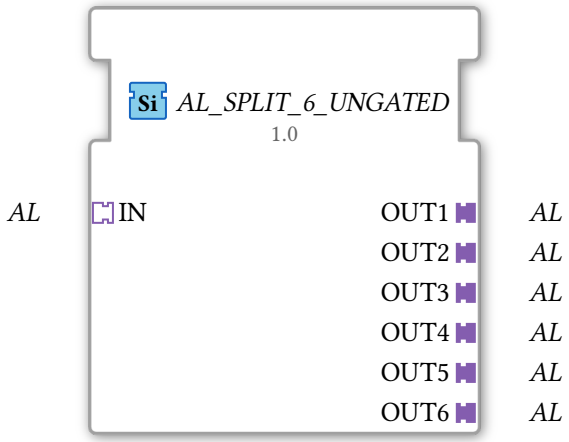
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AL_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_6_UNGATED** is used to split a single unidirectional AL adapter input (socket) into six identical AL adapter outputs (plugs). It enables simple signal distribution within a 4diac application without requiring additional logic or a state machine.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Direction	Identifier	Type	Description
In	IN	adapter::types::unidirectional::AL	Unidirectional AL adapter as input signal.
Out	OUT1	adapter::types::unidirectional::AL	First output channel (identical to IN).
Out	OUT2	adapter::types::unidirectional::AL	Second output channel.
Out	OUT3	adapter::types::unidirectional::AL	Third output channel.
Out	OUT4	adapter::types::unidirectional::AL	Fourth output channel.
Out	OUT5	adapter::types::unidirectional::AL	Fifth output channel.
Out	OUT6	adapter::types::unidirectional::AL	Sixth output channel.

Functionality

This function block has no internal logic or state machine. It forwards the AL signal arriving at socket **IN** unchanged to all six plugs **OUT1** … **OUT6**. Any change or event at the input is thus transmitted simultaneously to all outputs. The function block operates purely structurally as a splitter and does not require event activation.

Technical Features

- **Generic Function Block:** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `GEN_AL_SPLIT`. This allows it to be treated as a generic type in the 4diac IDE and enables clean type system integration.
- **No Data/Event Interface:** Distribution occurs exclusively via the adapter interface. No additional data or event ports are required.
- **No Runtime Dependencies:** Since no algorithms or states exist, the function block is resource-efficient and requires no specific initialization.

State Overview

The function block has no state machine (no ECC) and therefore no defined state machine. Its behavior is entirely determined by the interface structure and the simple forwarding of the adapter signal.

Application Scenarios

- **Signal Distribution in Agricultural Engineering or Automation:** A central AL adapter (e.g., a control command) is to be sent to several downstream components simultaneously.
- **Tests and Simulations:** A test signal can be distributed in parallel to different modules to verify system behavior.
- **Redundant Paths:** The same AL adapter can be routed to different processing units via multiple channels.

Comparison with Similar Function Blocks

- **AL_SPLIT_2, AL_SPLIT_4** … **AL_SPLIT_N**: These function blocks differ only in the number of output channels. The functionality is identical – a simple 1:N split.
- **Other split blocks with events or data:** Unlike these, **AL_SPLIT_6_UNGATED** has no data or event interfaces, but works purely via adapters. This simplifies configuration but limits the variety of types.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

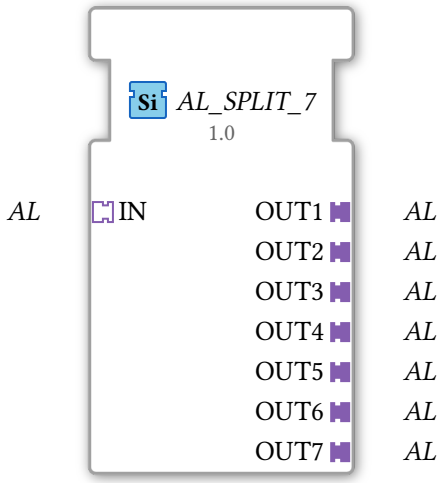
The **AL_SPLIT_6_UNGATED** is a compact and efficient function block for multiplying a unidirectional AL adapter signal. Due to its generic nature and simple structure, it is ideally suited for all applications where an input signal needs to be distributed to multiple outputs without additional logic or event control.

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AL_SPLIT_7



(Image of function block not available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_7** serves as a generic distributor (splitter) for a unidirectional adapter of type `AL`. It accepts a single adapter at its socket `IN` and forwards the incoming data and events unchanged to seven identical adapter outputs (`OUT1` … `OUT7`). The block is implemented as a generic type (`GEN_AL_SPLIT`), thus enabling flexible use in control and automation applications.

Interface Structure

Event Inputs

This function block has no explicit event inputs. All event control is handled via the connected adapter `IN`.

Event Outputs

There are no direct event outputs. Events are passed on via the seven adapter outputs (`OUT1` … `OUT7`).

Data Inputs

This function block has no independent data inputs. Data is transferred exclusively via the adapter `IN`.

Data Outputs

Accordingly, there are no direct data outputs. The data is replicated via the seven adapter outputs.

Adapters

Direction	Name	Type	Description

Socket	IN	adapter::types::unidirectional::AL	Adapter input that provides the AL connection to be distributed.
Plug	OUT1 … OUT7	adapter::types::unidirectional::AL	Seven adapter outputs that provide an identical copy of the input adapter.

Functionality

The **AL_SPLIT_7** is a pure distribution module. When an adapter of type AL is connected to socket IN, all data and events received via this adapter are forwarded unchanged to each of the seven plugs (OUT1 … OUT7). No logical or temporal processing takes place – the distribution is one-to-seven without delay or buffering. Due to its generic implementation, the function block can be used with different AL adapter instances that have the same data type and event schema.

Technical Features

- **Generic Implementation:** The function block is declared as a generic type (GEN_AL_SPLIT). This allows it to be used in different projects without modifying the source code, as long as the underlying adapter type AL corresponds to the expected interface.
- **Unidirectionality:** The module operates exclusively unidirectionally – data and events flow from the socket to the plugs. Feedback or bidirectional communication is not supported.
- **No State Logic:** The module does not have its own state machine; it is transparent.

State Overview

The module has no internal states. It is always ready to distribute incoming adapter connections. An active state results solely from the presence of a valid adapter at the IN socket.

Application Scenarios

- **Control Distribution:** A higher-level control adapter (e.g., for controlling an axis or sensor) should send identical signals to multiple parallel components.
- **Test Environments:** In simulation or test setups, one adapter can supply several similar subsystems.
- **Signal Bundling:** This function block can be used as a distributor in a hierarchy of adapters when a signal needs to be split across multiple downstream function blocks.

Comparison with Similar Function Blocks

Compared to an **AL_SPLIT_2** or **AL_SPLIT_4**, **AL_SPLIT_7** offers a higher number of outputs (seven instead of two or four). While smaller splitters are sufficient for low-demand applications, this function block enables direct distribution to seven receivers without the need to cascade multiple splitters. Its function is the exact opposite of an **AL_MERGE** (which combines multiple inputs).

Change Detection

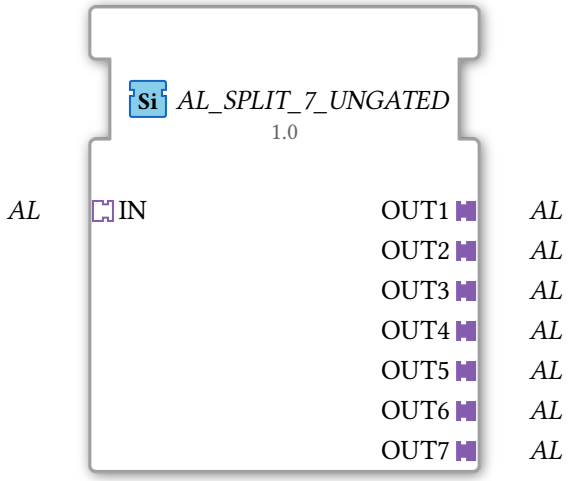
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AL_SPLIT_7** is a simple yet useful generic function block for replicating unidirectional AL adapter connections. Thanks to its generic nature and clear interface, it can be flexibly integrated into automation and control systems where a signal needs to be distributed across multiple paths. Its inherent transparency ensures lossless transmission without additional latency.

AL_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_SPLIT_7. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(Image of function block not available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_7_UNGATED** serves as a generic distributor (splitter) for a unidirectional adapter of type `AL`. It accepts a single adapter at its socket `IN` and forwards the incoming data and events unchanged to seven identical adapter outputs (`OUT1` ... `OUT7`). The block is implemented as a generic type (`GEN_AL_SPLIT`), thus enabling flexible use in control and automation applications.

Interface Structure

Event Inputs

This function block has no explicit event inputs. All event control is handled via the connected adapter `IN`.

Event Outputs

There are no direct event outputs. Events are passed on via the seven adapter outputs (`OUT1` ... `OUT7`).

Data Inputs

This function block has no independent data inputs. Data is transferred exclusively via the adapter `IN`.

Data Outputs

Accordingly, there are no direct data outputs. The data is replicated via the seven adapter outputs.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AL`	Adapter input that provides the AL connection to be distributed.
Plug	`OUT1` ... `OUT7`	`adapter::types::unidirectional::AL`	Seven adapter outputs that provide an identical copy of the input adapter.

Functionality

The **AL_SPLIT_7_UNGATED** is a pure distribution module. When an adapter of type `AL` is connected to socket `IN`, all data and events received via this adapter are forwarded unchanged to each of the seven plugs (`OUT1` ... `OUT7`). No logical or temporal processing takes place – the distribution is one-to-seven without delay or buffering. Due to its generic implementation, the function block can be used with different AL adapter instances that have the same data type and event schema.

Technical Features

- **Generic Implementation:** The function block is declared as a generic type (`GEN_AL_SPLIT`). This allows it to be used in different projects without modifying the source code, as long as the underlying adapter type `AL` corresponds to the expected interface.
- **Unidirectionality:** The module operates exclusively unidirectionally – data and events flow from the socket to the plugs. Feedback or bidirectional communication is not supported.
- **No State Logic:** The module does not have its own state machine; it is transparent.

State Overview

The module has no internal states. It is always ready to distribute incoming adapter connections. An active state results solely from the presence of a valid adapter at the `IN` socket.

Application Scenarios

- **Control Distribution:** A higher-level control adapter (e.g., for controlling an axis or sensor) should send identical signals to multiple parallel components.
- **Test Environments:** In simulation or test setups, one adapter can supply several similar subsystems.
- **Signal Bundling:** This function block can be used as a distributor in a hierarchy of adapters when a signal needs to be split across multiple downstream function blocks.

Comparison with Similar Function Blocks

Compared to an **AL_SPLIT_2** or **AL_SPLIT_4**, **AL_SPLIT_7_UNGATED** offers a higher number of outputs (seven instead of two or four). While smaller splitters are sufficient for low-demand applications, this function block enables direct distribution to seven receivers without the need to cascade multiple splitters. Its function is the exact opposite of an **AL_MERGE** (which combines multiple inputs).

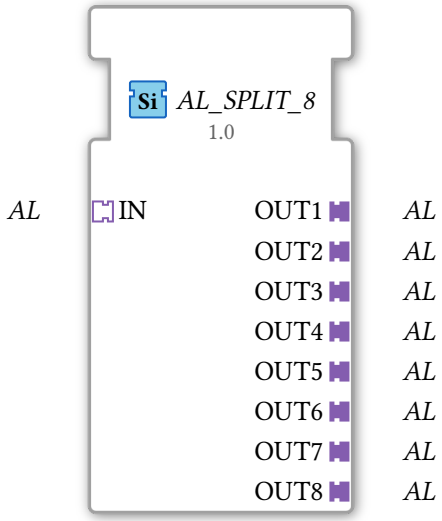
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AL_SPLIT_7_UNGATED** is a simple yet useful generic function block for replicating unidirectional AL adapter connections. Thanks to its generic nature and clear interface, it can be flexibly integrated into automation and control systems where a signal needs to be distributed across multiple paths. Its inherent transparency ensures lossless transmission without additional latency.

AL_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block AL_SPLIT_8 distributes an incoming unidirectional AL signal (socket IN) to eight identical outputs (plugs OUT1 to OUT8). It is declared as a generic block and serves for simple signal multiplication in 4diac applications.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Type	Name	Direction	Description
adapter::types::unidirectional::AL	IN	Socket (Input)	Incoming AL signal
adapter::types::unidirectional::AL	OUT1 ... OUT8	Plugs (Output)	Eight outgoing AL signals

Functionality

The component has no internal logic or state machine. An AL adapter signal applied to IN is forwarded unchanged and simultaneously to all eight output adapters. This multiplication is purely data-flow controlled, without delay or buffering.

Technical Features

- **Generic Type**: The function block is declared as a generic block (GenericClassName = 'GEN_AL_SPLIT'), but due to the fixed adapter definition, it can only be used with the type AL.
- **No Event Control**: There is neither an ECC nor an event interface; data is passed passively via the data flow.
- **Unidirectional**: The adapter is designed for only one direction (input → outputs).

State Overview

The function block has no internal states. It is completely stateless and performs no processing.

Application Scenarios

- **Alarm Distribution**: A central alarm signal is passed on to multiple subsystems (e.g., display, logging, control).
- **Redundant Signaling**: The same signal can be sent in parallel to multiple receivers to increase fault tolerance.
- **Controlling multiple actuators**: A sensor or control signal is split across multiple independent actuators.

Comparison with similar function blocks

- **Standard SPLIT function blocks** (e.g., SPLIT for simple data types) distribute individual values, while AL_SPLIT_8 is specifically designed for the unidirectional adapter AL.
- **Adapter mergers** (such as a hypothetical AL_MERGE) combine multiple signals into one; AL_SPLIT_8 implements the reverse 1:n functionality.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AL_SPLIT_8` is a simple yet important function block for multiplying AL adapter signals. Its passive, stateless operation allows it to integrate seamlessly into data-flow-oriented 4diac applications and facilitates the structured distribution of alarm or control signals.

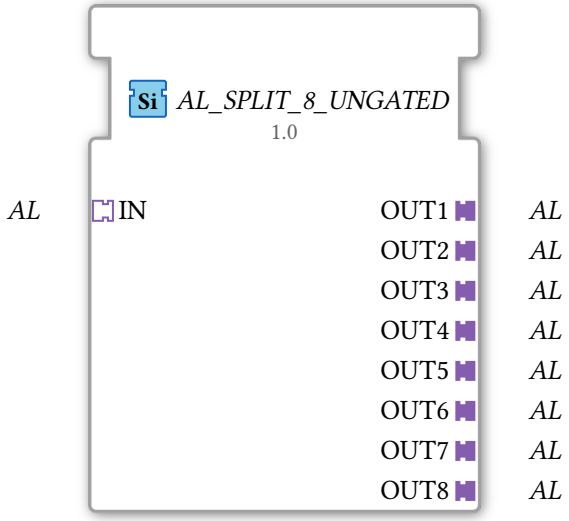
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AL_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AL_SPLIT_8_UNGATED` distributes an incoming unidirectional AL signal (socket `IN`) to eight identical outputs (plugs `OUT1` to `OUT8`). It is declared as a generic block and serves for simple signal multiplication in 4diac applications.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AL`	`IN`	Socket (Input)	Incoming AL signal
`adapter::types::unidirectional::AL`	`OUT1` … `OUT8`	Plugs (Output)	Eight outgoing AL signals

Functionality

The component has no internal logic or state machine. An AL adapter signal applied to `IN` is forwarded unchanged and simultaneously to all eight output adapters. This multiplication is purely data-flow controlled, without delay or buffering.

Technical Features

- **Generic Type**: The function block is declared as a generic block (`GenericClassName = 'GEN_AL_SPLIT'`), but due to the fixed adapter definition, it can only be used with the type `AL`.
- **No Event Control**: There is neither an ECC nor an event interface; data is passed passively via the data flow.
- **Unidirectional**: The adapter is designed for only one direction (input ⟶ outputs).

State Overview

The function block has no internal states. It is completely stateless and performs no processing.

Application Scenarios

- **Alarm Distribution**: A central alarm signal is passed on to multiple subsystems (e.g., display, logging, control).
- **Redundant Signaling**: The same signal can be sent in parallel to multiple receivers to increase fault tolerance.
- **Controlling multiple actuators**: A sensor or control signal is split across multiple independent actuators.

Comparison with similar function blocks

- **Standard SPLIT function blocks** (e.g., `SPLIT` for simple data types) distribute individual values, while `AL_SPLIT_8_UNGATED` is specifically designed for the unidirectional adapter `AL`.
- **Adapter mergers** (such as a hypothetical `AL_MERGE`) combine multiple signals into one; `AL_SPLIT_8_UNGATED` implements the reverse 1:n functionality.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

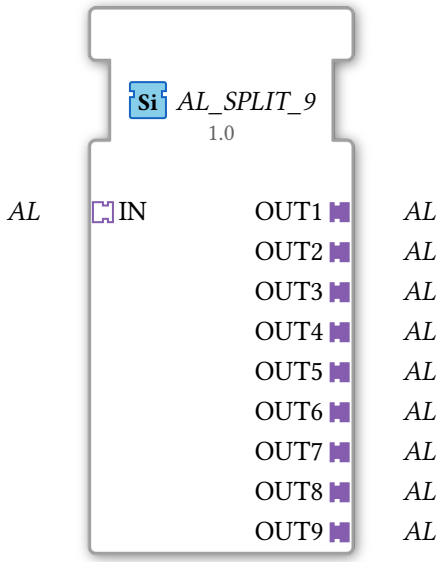
The `AL_SPLIT_8_UNGATED` is a simple yet important function block for multiplying AL adapter signals. Its passive, stateless operation allows it to integrate seamlessly into data-flow-oriented 4diac applications and facilitates the structured distribution of alarm or control signals.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AL_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_9** is a generic function block that splits an incoming adapter of type `AL` into nine identical outgoing adapters. It is used to distribute a signal or data flow arriving via a single adapter to multiple downstream components. The function block is defined as a generic type (`GEN_AL_SPLIT`) and must be bound to the specific adapter type before use.

Interface Structure

Event Inputs

The function block has no event inputs. Data is passed exclusively via adapter connections.

Event Outputs

There are no event outputs.

Data Inputs

No separate data inputs – the actual data is transmitted via the incoming adapter.

Data Outputs

No separate data outputs – the output data is provided via the nine outgoing adapters.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AL`	An incoming adapter that provides the signal or data to be distributed.
Plug (Output)	`OUT1` … `OUT9`	`adapter::types::unidirectional::AL`	Nine outgoing adapters, each outputting a copy of the input signal.

Functionality

The function block acts as a passive distributor. As soon as the incoming adapter `IN` receives a signal or data packet, the function block forwards it to all nine output adapters (`OUT1` … `OUT9`). This forwarding occurs without buffering or delay – it is a pure 1:9 distribution.

Since the function block is generic, a specific adapter type must be assigned to it before use. This is done in the development environment by specifying the adapter type, which defines the actual data fields and events.

Technical Features

- **Generic Function Block:** The function block is declared as `GEN_AL_SPLIT` and requires binding to a specific `AL` adapter type (e.g., `SimpleData_AL`). The binding is controlled by the attribute `eclipse4diac::core::GenericClassName`.
- **Type Safety:** All nine outputs use the same adapter type as the input. This ensures that the data structure remains identical.
- **Stateless:** The function block has no internal states or memory – distribution occurs instantaneously with each new data iteration.
- **No Event Control:** Since no event inputs are defined, distribution is triggered solely by data changes at the input adapter.

State Overview

The function block has no internal state machine. It operates statelessly and is purely data-flow driven.

Application Scenarios

- **Signal Distribution:** A sensor value (e.g., temperature, pressure) is to be sent simultaneously to multiple evaluation units or controllers.
- **Data Replication:** A control command is to be distributed to multiple actuators in parallel.
- **Modular Architectures:** Splitting a data stream in a pipeline to feed different processing branches.

Comparison with Similar Components

- **AL_SPLIT_2, AL_SPLIT_4:** Components with analog functionality that distribute the input signal to 2 or 4 outputs, respectively. `AL_SPLIT_9` covers a larger number of outputs, which is particularly useful in complex distribution structures.
- **Simple Coupling (e.g., Direct Connection):** Without the Split component, the sender would have to provide multiple adapter connections themselves. The Split component encapsulates this logic and simplifies the overall architecture.

AL_SPLIT_2, AL_SPLIT_4: Components with analog functionality that distribute the input signal to 2 or 4 outputs, respectively. `AL_SPLIT_9` covers a larger number of outputs, which is particularly useful in complex distribution structures.

Simple Coupling (e.g., Direct Connection): Without the Split component, the sender would have to provide multiple adapter connections themselves. The Split component encapsulates this logic and simplifies the overall architecture.

**

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AL_SPLIT_9` is a simple yet powerful generic distribution block for adapter interfaces of type `AL`. It enables clean, maintainable distribution of a data flow across nine independent paths. Thanks to its generic nature and stateless operation, it is ideally suited for modular and scalable automation solutions.

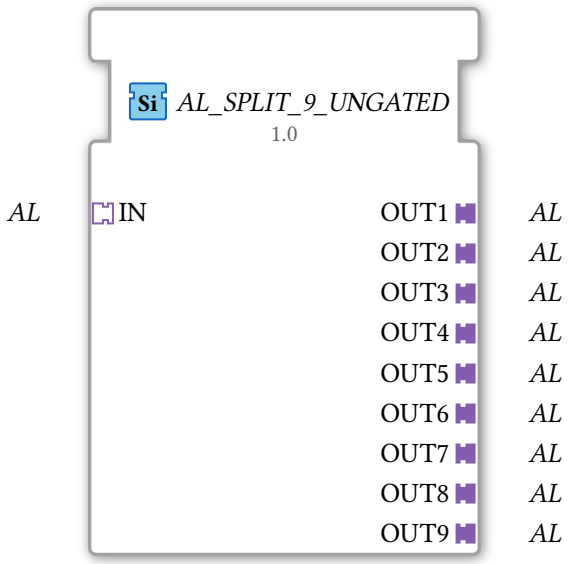
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AL_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/

frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_SPLIT_9_UNGATED** is a generic function block that splits an incoming adapter of type `AL` into nine identical outgoing adapters. It is used to distribute a signal or data flow arriving via a single adapter to multiple downstream components. The function block is defined as a generic type (`GEN_AL_SPLIT`) and must be bound to the specific adapter type before use.

Interface Structure

Event Inputs

The function block has no event inputs. Data is passed exclusively via adapter connections.

Event Outputs

There are no event outputs.

Data Inputs

No separate data inputs – the actual data is transmitted via the incoming adapter.

Data Outputs

No separate data outputs – the output data is provided via the nine outgoing adapters.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AL`	An incoming adapter that provides the signal or data to be distributed.
Plug (Output)	`OUT1` … `OUT9`	`adapter::types::unidirectional::AL`	Nine outgoing adapters, each outputting a copy of the input signal.

Functionality

The function block acts as a passive distributor. As soon as the incoming adapter `IN` receives a signal or data packet, the function block forwards it to all nine output adapters (`OUT1` … `OUT9`). This forwarding occurs without buffering or delay – it is a pure 1:9 distribution.

Since the function block is generic, a specific adapter type must be assigned to it before use. This is done in the development environment by specifying the adapter type, which defines the actual data fields and events.

Technical Features

- **Generic Function Block:** The function block is declared as `GEN_AL_SPLIT` and requires binding to a specific `AL` adapter type (e.g., `SimpleData_AL`). The binding is controlled by the attribute `eclipse4diac::core::GenericClassName`.
- **Type Safety:** All nine outputs use the same adapter type as the input. This ensures that the data structure remains identical.
- **Stateless:** The function block has no internal states or memory – distribution occurs instantaneously with each new data iteration.
- **No Event Control:** Since no event inputs are defined, distribution is triggered solely by data changes at the input adapter.

State Overview

The function block has no internal state machine. It operates statelessly and is purely data-flow driven.

Application Scenarios

- **Signal Distribution:** A sensor value (e.g., temperature, pressure) is to be sent simultaneously to multiple evaluation units or controllers.
- **Data Replication:** A control command is to be distributed to multiple actuators in parallel.
- **Modular Architectures:** Splitting a data stream in a pipeline to feed different processing branches.

Comparison with Similar Components

- **AL_SPLIT_2, AL_SPLIT_4:** Components with analog functionality that distribute the input signal to 2 or 4 outputs, respectively. `AL_SPLIT_9_UNGATED` covers a larger number of outputs, which is particularly useful in complex distribution structures.
- **Simple Coupling (e.g., Direct Connection):** Without the Split component, the sender would have to provide multiple adapter connections themselves. The Split component encapsulates this logic and simplifies the overall architecture.

AL_SPLIT_2, AL_SPLIT_4: Components with analog functionality that distribute the input signal to 2 or 4 outputs, respectively. `AL_SPLIT_9_UNGATED` covers a larger number of outputs, which is particularly useful in complex distribution structures.

Simple Coupling (e.g., Direct Connection): Without the Split component, the sender would have to provide multiple adapter connections themselves. The Split component encapsulates this logic and simplifies the overall architecture.

**

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AL_SPLIT_9_UNGATED` is a simple yet powerful generic distribution block for adapter interfaces of type `AL`. It enables clean, maintainable distribution of a data flow across nine independent paths. Thanks to its generic nature and stateless operation, it is ideally suited for modular and scalable automation solutions.

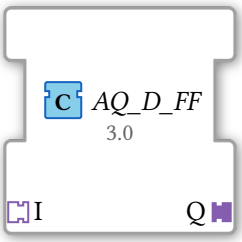
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numeric Index (unidirectional)

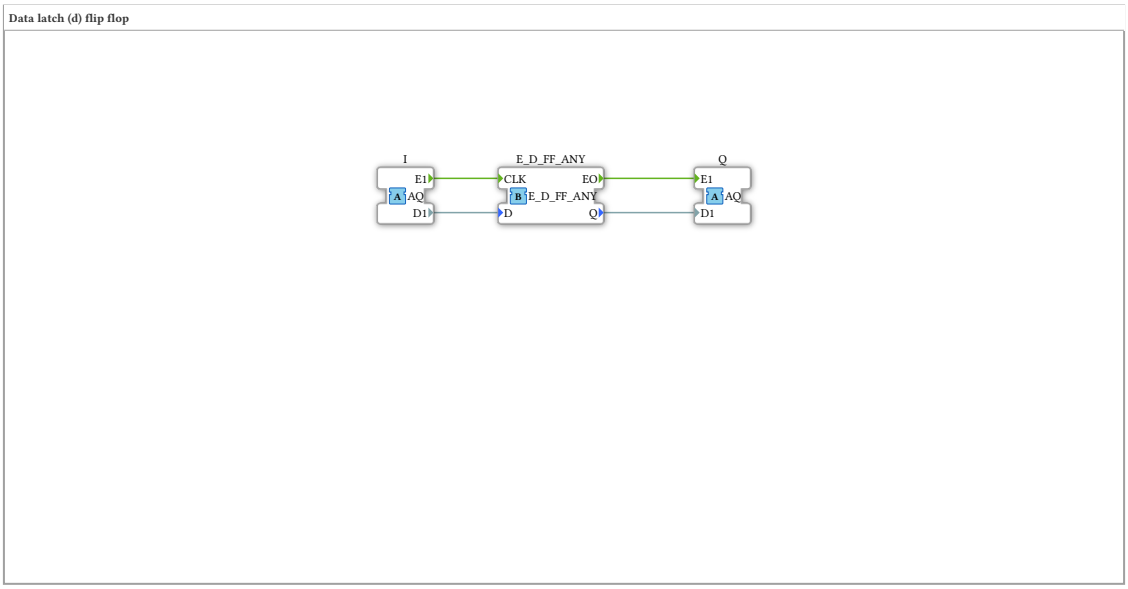
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AQ_D_FF



Value to latch – *AQ*

AQ – Latched value



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AQ_D_FF` implements a data storage element (D-latch) based on the adapter concept of IEC 61499. It receives an incoming data value upon an event and makes it available at the output until the next event. The block fully encapsulates the latch functionality in an adapter interface, thus enabling modular integration into system-wide data and event paths.

Interface Structure

Event Inputs

No direct event inputs are available. Event control is handled exclusively via the adapter `I`.

Event Outputs

No direct event outputs are available. Event output is handled exclusively via the adapter `Q`.

Data Inputs

No direct data inputs are available. The data value to be latched is provided via the adapter `I`.

Data Outputs

No direct data outputs are available. The latched data value is output via the adapter `Q`.

Adapters

Adapter	Direction	Type	Description
`I`	Socket (Input)	`adapter::types::unidirectional::AQ`	Provides the data value to be latched and the triggering event.

Q	Plug (Output)	adapter::types::unidirectional::AQ	Outputs the latched data value and an acknowledgment event.

The adapter type used, adapter::types::unidirectional::AQ, provides a unidirectional connection: One event channel and one data channel are transmitted from the socket to the plug.

Functionality

The AQ_D_FF operates on the principle of an edge-triggered D-latch. Internally, the function block iec61499::events::E_D_FF_ANY is used:

- As soon as an event arrives at the I adapter, the currently available data value, D1 (from the I adapter), is retrieved.
- The received value is output as D1 at adapter Q, and simultaneously, event E1 is triggered at the output adapter.
- The output value remains constant between two events. With each new event at the input, the output is updated to the then-current input value.

Technical Features

- **Pure Adapter Interface**: The module has no direct event or data inputs/outputs. All communication takes place via adapters I and Q.
- **Unidirectional Data Transmission**: The adapters are of type adapter::types::unidirectional::AQ, meaning that data and events flow only from input to output. Feedback or bidirectional communication is not supported.
- **Internal Implementation**: The latch function is implemented using the universal function block E_D_FF_ANY from the IEC 61499 event library, ensuring robust and standards-compliant processing.
- **No Initialization**: The initial state of the latched value is undefined until the first event arrives at the input.

State Overview

The AQ_D_FF does not have an explicit state machine. However, its behavior can be described by two logical states:

State	Description
Waiting for Event	The output Q holds the last latched value. No event is pending at I.
Accept/Output	When an event arrives at I, the new value is transferred from I, output to Q, and an event is generated on E1 by Q. The function block then returns to its wait state.

Application Scenarios

- **Cache Storage of Sensor Values**: A measured value is captured upon a trigger event (e.g., clock cycle or query) and is then available for further processing.
- **Synchronization of Asynchronous Data Streams**: If data arrives asynchronously, AQ_D_FF can hold the last valid value until a subsequent process retrieves it.
- **Event-driven data forwarding**: The function block can be used as a simple "data holder" in an adapter-based communication path to forward data only upon specific events.

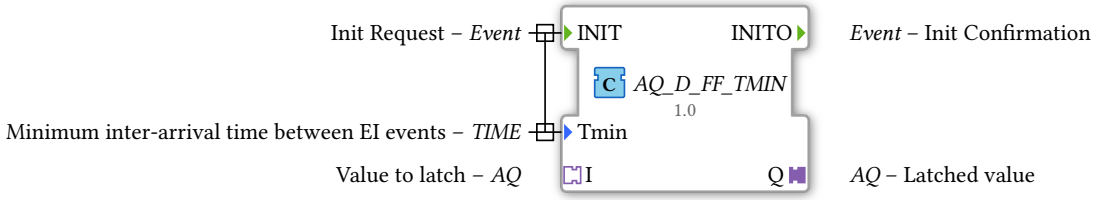
Comparison with similar function blocks

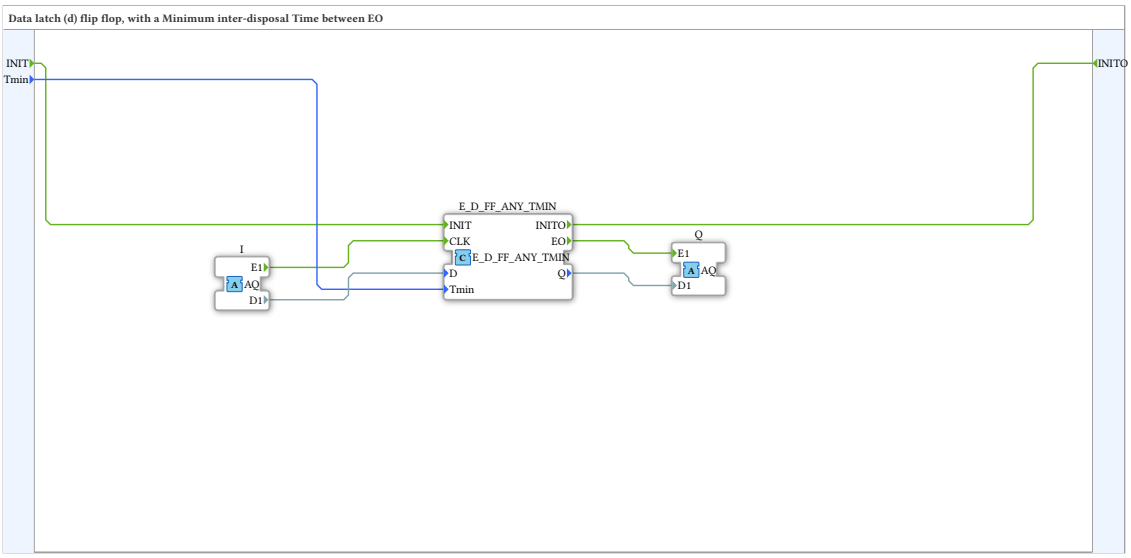
Function block	Properties
E_D_FF (direct I/Os)	Same latch functionality, but with separate event and data inputs/outputs. No adapters.
E_SR_FF	Set/reset flip-flop with two separate input events. Binary states, not data values.
AQ_D_FF	Enables embedding the latch function into existing adapter interfaces without additional wiring at the function block level.

Conclusion

The function block `AQ_D_FF` offers a compact, adapter-based implementation of a D-latch. It integrates seamlessly into system-wide adapter paths, thus reducing wiring effort at the function block level. Its operation is simple and reliable, especially for applications where a data value needs to be transferred upon an event and held until the next event.

AQ_D_FF_TMIN





• – ○ • – ○ • – ○ •

The function block **AQ_D_FF_TMIN** implements a data-locking D flip-flop with a minimum inter-disposal time (MIT) between two consecutive output events (EOs). It serves as an extension of a simple D flip-flop, adding time-based debouncing or minimum-distance control, and is connected to the signal transmitter and receiver via adapters.

Event	Type	With Variable	Comment
INIT	EInit	Tmin	Initialization request, sets the minimum MIT.
Event	Type	Comment	
–––––	––	–––––	
INITO	EInit	Initialization confirmation.	
Variable	Type	Comment	
–––––	––	–––––	
Tmin	TIME	Minimum waiting time between two output events (EO).	

No dedicated data outputs; the latched signal is output via the adapter `Q`.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

| Type | Label | Direction | Comment | | `adapter::types::unidirectional::AQ` | I | Socket (Input) | Latching value, containing one data signal (D1) and one event signal (E1). | | `adapter::types::unidirectional::AQ` | Q | Plug (Output) | Latched value, containing one data signal (D1) and one event signal (E1). |

Functionality

This function block encapsulates an internal function block of type `iec61499::events::E_D_FF_ANY_TMIN`. This functions like an edge-triggered D flip-flop, where the event `I.E1` serves as the clock signal and `I.D1` as the data input. The output event `EO` is only output at adapter `Q.E1` if at least the time specified in `Tmin` has elapsed since the last output event. The data value at output `Q.D1` corresponds to the value `I.D1` present at the time of the rising edge at `I.E1`.

The initialization (`INIT`) passes the parameter `Tmin` to the internal function block, thus defining the timing condition.

- **Minimum Interval Time (Tmin):** Prevents excessively rapid succession of output events, e.g., for debouncing or limiting the update rate.
- **Adapter-Based Communication:** The function block (FB) uses unidirectional adapters (`AQ`) that bundle both data and events, enabling loose coupling between the signal source and sink.
- **IEC 61499-Compliant:** Complies with the 61499-1 Annex A standard and utilizes the event/data flows of the standard.

The FB does not have an explicit public state machine. Its behavior is determined by the internal FB `E_D_FF_ANY_TMIN`, which has the last latched value as its state and a wait logic. Simplified representation:

State	Meaning
Initialized	Ready after INIT, Tmin is set
Ready	Waiting for clock (I.E1)
Latched	Data value has been accepted; output event will be released after Tmin has elapsed
Waiting	Tmin is running – output event is blocked

A detailed state representation can be found in the source code of the internal function block.

- **Signal Smoothing/Debouncing:** Preventing multiple triggers with mechanical pushbuttons or rapid signal changes.
- **Data Rate Limiting:** Limiting the output frequency of measured values in monitoring systems.
- **Synchronization of Asynchronous Signals:** Forwarding a data value with a defined minimum time interval.
- **Control of Slow Actuators:** Ensuring that actuators are not controlled faster than their mechanical response time allows.

Function Block	Special Feature
`AQ_D_FF` / `E_D_FF`	Simple D flip-flop without a time limit on the output.
AQ_D_FF_TMIN` (this FB)	Extended by a minimum delay Tmin between output events.
AQ_D_FF_TMAX` (hypothetical)	Limits the maximum time between two output events.

Compared to a standard D flip-flop, `AQ_D_FF_TMIN` offers additional protection against excessively rapid succession of output events, which is often necessary in practice.

The function block `AQ_D_FF_TMIN` combines the functionality of a D flip-flop with an adjustable minimum delay at the output. Thanks to its adapter interfaces and IEC 61499 compliance, it can be easily integrated into industrial control applications where both data acquisition and temporal decoupling of events are required. It offers a robust solution for signal processing and actuator control with defined temporal constraints.

Technical Features

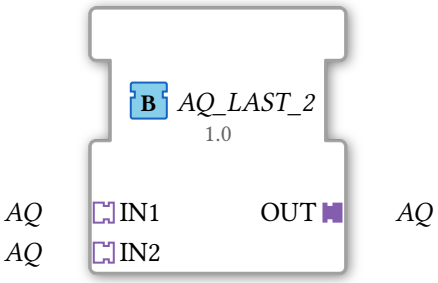
State Overview

Application Scenarios

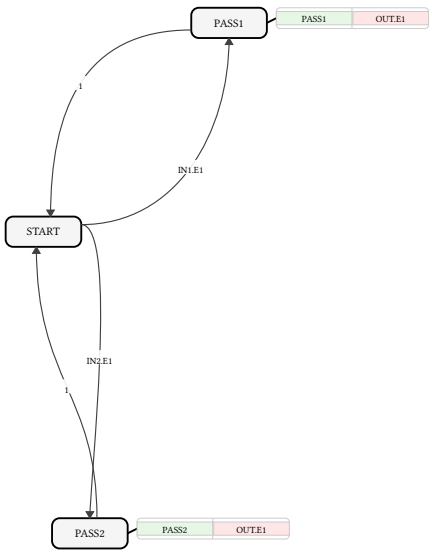
Comparison with Similar Function Blocks

Conclusion

AQ_LAST_2



ECC: AQ_LAST_2



Introduction

The AQ_LAST_2 function block merges two same-type QUARTER adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: QUARTER adapter (unidirectional) - first signal source
- **IN2**: QUARTER adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: QUARTER adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

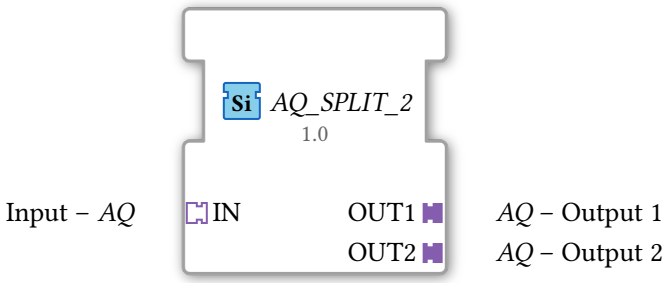
Application Scenarios

- Merging two equally-valid QUARTER signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same QUARTER value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AQ_LAST_2 is the inverse of `AQ_SPLIT_2`: AQ_SPLIT_2 distributes 1 input to 2 outputs, AQ_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

AQ_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AQ_SPLIT_2` serves as a fan-out block. It distributes an incoming AQ (Analog Quantity) signal unidirectionally to two identical outputs. It thus implements a simple 1:2 signal multiplication for analog quantities.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
adapter::types::unidirectional::AQ	IN	Socket (Input)	Input adapter for the AQ signal to be split.
adapter::types::unidirectional::AQ	OUT1	Plug (Output)	First output – provides the unaltered input signal.
adapter::types::unidirectional::AQ	OUT2	Plug (Output)	Second output – also provides the unaltered input signal.

Functionality

The module receives an analog signal via the adapter socket `IN`. This signal is passed internally without delay or conversion to both adapter plugs `OUT1` and `OUT2`. No signal processing takes place – the function block acts purely as a splitter.

Technical Features

- **Generic Implementation**: The function block is declared as the generic type `GEN_AQ_SPLIT` and can be used with any AQ adapter, as long as it conforms to the unidirectional AQ interface.
- **No Event Control**: The function block operates purely data-driven without events – a change in state at the input immediately affects both outputs.
- **Unidirectional**: The adapters are designed for one direction only (input ⟶ outputs).

State Overview

The function block has no internal states. It is stateless (combinatory) and outputs the currently present input signal at both outputs at any given time.

Application Scenarios

- **Signal Distribution** in Control Systems: An analog signal provided by a sensor is to be passed on in parallel to two different devices (e.g., two controllers or display units).
- **Test and Simulation Setups**: A reference signal is split across multiple test points without affecting the signal flow.
- **Redundant Transmission**: The signal is sent to two independent paths to increase reliability.

Comparison with Similar Components

- **`AQ_SPLIT_N`** (hypothetical): A comparable component with outputs like `N` – `AQ_SPLIT_2` is a specialized 1:2 version.
- **Event-Driven Split Components**: These often require trigger events and then perform signal transmission. In contrast, `AQ_SPLIT_2` operates purely signal-driven without events.
- **Bus-based splitters**: These distribute multiple signals over a shared bus (e.g., via multicast). `AQ_SPLIT_2` operates selectively and individually via adapters.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AQ_SPLIT_2` is a simple yet useful component for distributing an analog signal to two identical outputs. Thanks to its generic nature, it can be used in various 4diac projects without modification. Its stateless operation minimizes complexity and makes it ideal for real-time applications requiring immediate signal transmission.

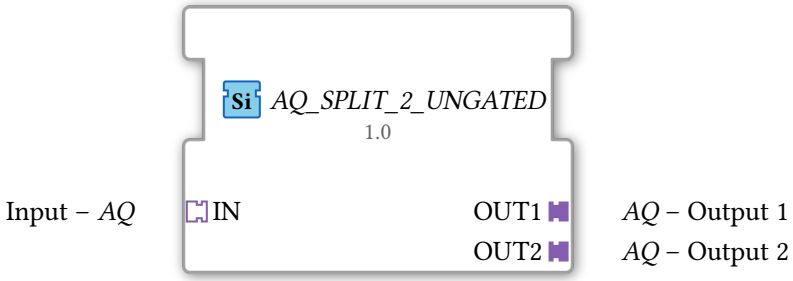
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AQ_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AQ_SPLIT_2_UNGATED` serves as a fan-out block. It distributes an incoming AQ (Analog Quantity) signal unidirectionally to two identical outputs. It thus implements a simple 1:2 signal multiplication for analog quantities.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AQ`	`IN`	Socket (Input)	Input adapter for the AQ signal to be split.
`adapter::types::unidirectional::AQ`	`OUT1`	Plug (Output)	First output – provides the unaltered input signal.
`adapter::types::unidirectional::AQ`	`OUT2`	Plug (Output)	Second output – also provides the unaltered input signal.

Functionality

The module receives an analog signal via the adapter socket `IN`. This signal is passed internally without delay or conversion to both adapter plugs `OUT1` and `OUT2`. No signal processing takes place – the function block acts purely as a splitter.

Technical Features

- **Generic Implementation**: The function block is declared as the generic type `GEN_AQ_SPLIT` and can be used with any AQ adapter, as long as it conforms to the unidirectional AQ interface.
- **No Event Control**: The function block operates purely data-driven without events – a change in state at the input immediately affects both outputs.
- **Unidirectional**: The adapters are designed for one direction only (input ⟶ outputs).

State Overview

The function block has no internal states. It is stateless (combinatory) and outputs the currently present input signal at both outputs at any given time.

Application Scenarios

- **Signal Distribution** in Control Systems: An analog signal provided by a sensor is to be passed on in parallel to two different devices (e.g., two controllers or display units).
- **Test and Simulation Setups**: A reference signal is split across multiple test points without affecting the signal flow.
- **Redundant Transmission**: The signal is sent to two independent paths to increase reliability.

Comparison with Similar Components

- **`AQ_SPLIT_N`** (hypothetical): A comparable component with outputs like `N` – `AQ_SPLIT_2_UNGATED` is a specialized 1:2 version.
- **Event-Driven Split Components**: These often require trigger events and then perform signal transmission. In contrast, `AQ_SPLIT_2_UNGATED` operates purely signal-driven without events.
- **Bus-based splitters**: These distribute multiple signals over a shared bus (e.g., via multicast). `AQ_SPLIT_2_UNGATED` operates selectively and individually via adapters.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

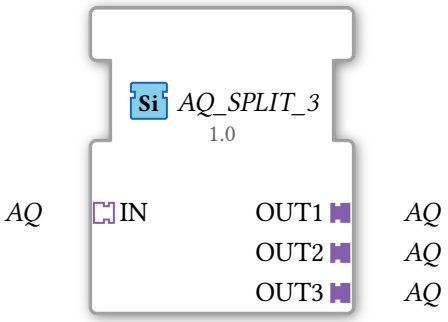
The `AQ_SPLIT_2_UNGATED` is a simple yet useful component for distributing an analog signal to two identical outputs. Thanks to its generic nature, it can be used in various 4diac projects without modification. Its stateless operation minimizes complexity and makes it ideal for real-time applications requiring immediate signal transmission.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AQ_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AQ_SPLIT_3` implements a 1-to-3 split (fan-out) of a unidirectional AQ adapter signal. An incoming AQ adapter signal is copied to three identical output adapters and made available in parallel.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Socket (Input)

Name	Type	Description
`IN`	`adapter::types::unidirectional::AQ`	Input adapter whose signal is distributed to three outputs

Plug (Outputs)

Name	Type	Description
`OUT1`	`adapter::types::unidirectional::AQ`	First output (copy of the input signal)
`OUT2`	`adapter::types::unidirectional::AQ`	Second Output (Copy of Input Signal)
`OUT3`	`adapter::types::unidirectional::AQ`	Third Output (Copy of Input Signal)

Functionality

The module forwards the AQ signal present at socket `IN` to all three plugs (`OUT1`, `OUT2`, `OUT3`) without delay or data manipulation. This is a pure signal distribution (fan-out) – the incoming adapter interface is mirrored to three identical output interfaces. Since no events or data outside the adapter are processed, the forwarding is passive and transparent.

Technical Features

- **Generic Type**: The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AQ_SPLIT'`. This allows it to be used as a generic AQ splitter in various contexts.
- **Unidirectional**: The adapters are defined as unidirectional AQ interfaces – data flows only from the input to the outputs.
- **No State Logic**: The function block has no event I/O, no internal state, and no data I/O. It functions purely as connection logic.

State Overview

The function block is stateless. It does not execute any time-dependent or sequential processes. The output signals always directly correspond to the current input signal.

Application Scenarios

- **Signal Distribution in Automation**: When an AQ signal (e.g., a quality value, a measurement, or status adapter) needs to be simultaneously distributed to multiple consumers.
- **Multicasting of Adapters**: In architectures where a service sends its result to multiple downstream components via an AQ adapter.

- **Generic Coupling**: Used as a generic splitter in libraries or frameworks based on the `GEN_AQ_SPLIT` type.

Comparison with Similar Components

- **AQ_SPLIT_2**: Distributes to only two outputs – `AQ_SPLIT_3` offers an additional third output.
- **Manual Triple Wiring**: Instead of a single component, the input signal could be manually connected to three sockets. However, the splitter improves the clarity and maintainability of the network.
- **Generic Splitters for Other Adapter Types**: Similar components exist, for example, for data or event signals (e.g., `DATA_SPLIT_3`). `AQ_SPLIT_3` is specifically designed for the AQ adapter type.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AQ_SPLIT_3` is a simple, functional fan-out component for unidirectional AQ adapters. Thanks to its generic design and clear 1:3 distribution, it is ideally suited for all applications that need to provide an AQ signal multiple times. The absence of event and data I/O makes it lightweight and usable in any process context.

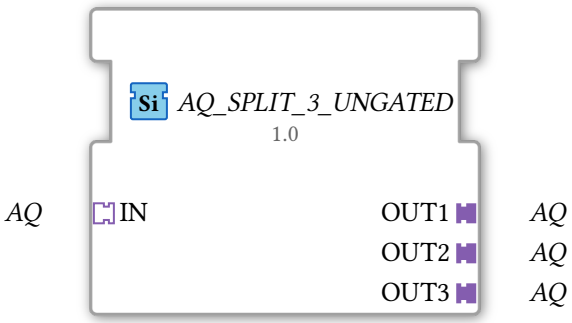
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AQ_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AQ_SPLIT_3_UNGATED` implements a 1-to-3 split (fan-out) of a unidirectional AQ adapter signal. An incoming AQ adapter signal is copied to three identical output adapters and made available in parallel.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Socket (Input)

Name	Type	Description
IN	adapter::types::unidirectional::AQ	Input adapter whose signal is distributed to three outputs

Plug (Outputs)

Name	Type	Description
OUT1	adapter::types::unidirectional::AQ	First output (copy of the input signal)
OUT2	adapter::types::unidirectional::AQ	Second Output (Copy of Input Signal)
OUT3	adapter::types::unidirectional::AQ	Third Output (Copy of Input Signal)

Functionality

The module forwards the AQ signal present at socket `IN` to all three plugs (`OUT1`, `OUT2`, `OUT3`) without delay or data manipulation. This is a pure signal distribution (fan-out) – the incoming adapter interface is mirrored to three identical output interfaces. Since no events or data outside the adapter are processed, the forwarding is passive and transparent.

Technical Features

- **Generic Type**: The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AQ_SPLIT'`. This allows it to be used as a generic AQ splitter in various contexts.
- **Unidirectional**: The adapters are defined as unidirectional AQ interfaces – data flows only from the input to the outputs.
- **No State Logic**: The function block has no event I/O, no internal state, and no data I/O. It functions purely as connection logic.

State Overview

The function block is stateless. It does not execute any time-dependent or sequential processes. The output signals always directly correspond to the current input signal.

Application Scenarios

- **Signal Distribution in Automation**: When an AQ signal (e.g., a quality value, a measurement, or status adapter) needs to be simultaneously distributed to multiple consumers.
- **Multicasting of Adapters**: In architectures where a service sends its result to multiple downstream components via an AQ adapter.
- **Generic Coupling**: Used as a generic splitter in libraries or frameworks based on the `GEN_AQ_SPLIT` type.

Comparison with Similar Components

- **AQ_SPLIT_2**: Distributes to only two outputs – `AQ_SPLIT_3_UNGATED` offers an additional third output.
- **Manual Triple Wiring**: Instead of a single component, the input signal could be manually connected to three sockets. However, the splitter improves the clarity and maintainability of the network.
- **Generic Splitters for Other Adapter Types**: Similar components exist, for example, for data or event signals (e.g., `DATA_SPLIT_3`). `AQ_SPLIT_3_UNGATED` is specifically designed for the AQ adapter type.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

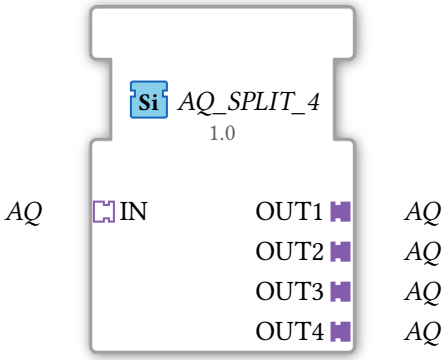
AQ_SPLIT_3_UNGATED` is a simple, functional fan-out component for unidirectional AQ adapters. Thanks to its generic design and clear 1:3 distribution, it is ideally suited for all applications that need to provide an AQ signal multiple times. The absence of event and data I/O makes it lightweight and usable in any process context.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AQ_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AQ_SPLIT_4` serves as a fan-out component for distributing an incoming AQ adapter to four identical output adapters. It is used to forward an analog signal or control variable (AQ – Analog Quantity) from a sender to multiple independent receivers without compromising signal integrity.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Type	Direction	Name	Description
`adapter::types::unidirectional::AQ`	Socket (Input)	`IN`	Incoming AQ adapter, which is distributed to the four outputs.
`adapter::types::unidirectional::AQ`	Plug (Output)	`OUT1`	First outgoing AQ adapter.
`adapter::types::unidirectional::AQ`	Plug (Output)	`OUT2`	Second outgoing AQ adapter.

`adapter::types::unidirectional::AQ`	Plug (Output)	`OUT3`	Third outgoing AQ adapter.
`adapter::types::unidirectional::AQ`	Plug (Output)	`OUT4`	Fourth outgoing AQ adapter.

Functionality

The `AQ_SPLIT_4` forwards all data and events arriving via socket `IN` from the AQ adapter unchanged to the four plug adapters `OUT1` to `OUT4`. No processing, filtering, or delay of the signals takes place. Distribution occurs purely at the connection level, so all outputs always have the same state as the input.

- **Generic Typing**: The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AQ_SPLIT'`. This allows it to be addressed as a generic function block in the 4diac IDE, increasing its reusability in various application contexts.
- **No State Logic**: The function block has no internal state machine (ECC) and no input/output events. It acts purely as a "wiring aid" for adapter connections.
- **Unidirectional Adapter**: All adapters used are of type `unidirectional::AQ`, meaning data flows only in one direction (from the socket to the plugs).

Technical Features

- **Generic Typing**: The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AQ_SPLIT'`. This allows it to be addressed as a generic function block in the 4diac IDE, increasing its reusability in various application contexts.
- **No State Logic**: The function block has no internal state machine (ECC) and no input/output events. It acts purely as a "wiring aid" for adapter connections.
- **Unidirectional Adapter**: All adapters used are of type `unidirectional::AQ`, meaning data flows only in one direction (from the socket to the plugs).

State Overview

The function block has no state machine. It is passive and does not perform any time- or event-driven operations. Signal transmission is continuous and without intermediate storage.

Application Scenarios

- **Distributing a Measured Value**: An analog sensor (e.g., temperature, pressure, flow rate) is connected via an AQ adapter and its signal needs to be sent to multiple control or monitoring units.
- **Multiple Use of a Control Signal**: An analog control signal output by a controller is transmitted in parallel to multiple actuators (e.g., valves, frequency converters).
- **Signal Monitoring**: The original signal is sent unchanged to the end devices, while an additional output is used for diagnostics or logging.

Comparison with Similar Function Blocks

- **AQ_SPLIT_2**: Distributes an AQ signal to two outputs instead of four – saving space and reducing requirements.
- **IQ_SPLIT_4**: Analog function block for digital adapters (e.g., `adapter::types::unidirectional::IQ`), otherwise identical functionality.
- **Manual Parallel Connection**: Theoretically, one could manually connect an AQ socket to multiple plugs, but the `AQ_SPLIT_4` offers a clear and reusable encapsulation.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AQ_SPLIT_4` is a simple yet useful function block for signal distribution in industrial automation with 4diac. Its generic design and lack of internal logic make it easy to understand, robust, and flexible. It contributes to the structuring and reusability of function block networks.

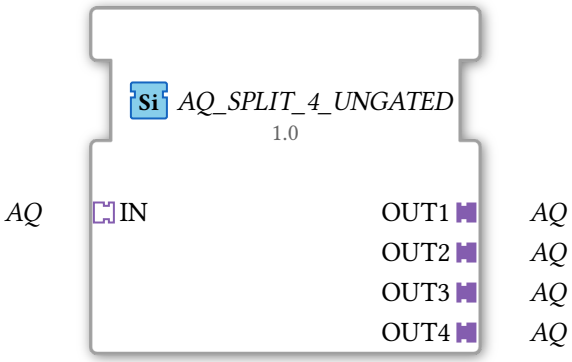
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

AQ_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AQ_SPLIT_4_UNGATED` serves as a fan-out component for distributing an incoming AQ adapter to four identical output adapters. It is used to forward an analog signal or control variable (AQ – Analog Quantity) from a sender to multiple independent receivers without compromising signal integrity.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Type	Direction	Name	Description
`adapter::types::unidirectional::AQ`	Socket (Input)	`IN`	Incoming AQ adapter, which is distributed to the four outputs.
`adapter::types::unidirectional::AQ`	Plug (Output)	`OUT1`	First outgoing AQ adapter.
`adapter::types::unidirectional::AQ`	Plug (Output)	`OUT2`	Second outgoing AQ adapter.
`adapter::types::unidirectional::AQ`	Plug (Output)	`OUT3`	Third outgoing AQ adapter.
`adapter::types::unidirectional::AQ`	Plug (Output)	`OUT4`	Fourth outgoing AQ adapter.

Functionality

The `AQ_SPLIT_4_UNGATED` forwards all data and events arriving via socket `IN` from the AQ adapter unchanged to the four plug adapters `OUT1` to `OUT4`. No processing, filtering, or delay of the signals takes place. Distribution occurs purely at the connection level, so all outputs always have the same state as the input.

- **Generic Typing**: The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AQ_SPLIT'`. This allows it to be addressed as a generic function block in the 4diac IDE, increasing its reusability in various application contexts.
- **No State Logic**: The function block has no internal state machine (ECC) and no input/output events. It acts purely as a "wiring aid" for adapter connections.
- **Unidirectional Adapter**: All adapters used are of type `unidirectional::AQ`, meaning data flows only in one direction (from the socket to the plugs).

Technical Features

- **Generic Typing**: The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AQ_SPLIT'`. This allows it to be addressed as a generic function block in the 4diac IDE, increasing its reusability in various application contexts.
- **No State Logic**: The function block has no internal state machine (ECC) and no input/output events. It acts purely as a "wiring aid" for adapter connections.
- **Unidirectional Adapter**: All adapters used are of type `unidirectional::AQ`, meaning data flows only in one direction (from the socket to the plugs).

State Overview

The function block has no state machine. It is passive and does not perform any time- or event-driven operations. Signal transmission is continuous and without intermediate storage.

Application Scenarios

- **Distributing a Measured Value**: An analog sensor (e.g., temperature, pressure, flow rate) is connected via an AQ adapter and its signal needs to be sent to multiple control or monitoring units.
- **Multiple Use of a Control Signal**: An analog control signal output by a controller is transmitted in parallel to multiple actuators (e.g., valves, frequency converters).
- **Signal Monitoring**: The original signal is sent unchanged to the end devices, while an additional output is used for diagnostics or logging.

Comparison with Similar Function Blocks

- **AQ_SPLIT_2**: Distributes an AQ signal to two outputs instead of four – saving space and reducing requirements.
- **IQ_SPLIT_4**: Analog function block for digital adapters (e.g., `adapter::types::unidirectional::IQ`), otherwise identical functionality.
- **Manual Parallel Connection**: Theoretically, one could manually connect an AQ socket to multiple plugs, but the `AQ_SPLIT_4_UNGATED` offers a clear and reusable encapsulation.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

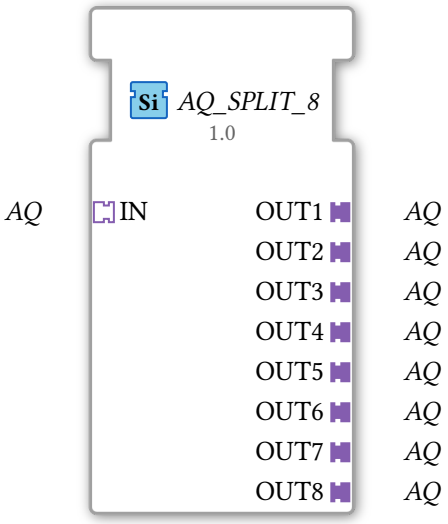
Conclusion

The `AQ_SPLIT_4_UNGATED` is a simple yet useful function block for signal distribution in industrial automation with 4diac. Its generic design and lack of internal logic make it easy to understand, robust, and flexible. It contributes to the structuring and reusability of function block networks.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

AQ_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AQ_SPLIT_8** is a fan-out function block that distributes a single AQ adapter input to eight identical AQ adapter outputs. It serves to multiply signals in analog output paths, thus enabling the parallel control of multiple identical actuators or subsystems. The implementation is based on the generic mechanism `GEN_AQ_SPLIT` and is designed for unidirectional AQ interfaces.

Interface Structure

Event Inputs

None – the function block operates purely on an adapter basis without event control.

Event Outputs

None – the function block has no event outputs.

Data Inputs

None – all signal transmission occurs via AQ adapters.

Data Outputs

None – see Data Inputs.

Adapters

Adapter	Type	Direction	Description
IN	`adapter::types::unidirectional::AQ`	Socket (Input)	Receives the AQ signal to be distributed.
OUT1 – OUT8	`adapter::types::unidirectional::AQ`	Plug (Output)	Identical outputs that forward the signal from `IN`.

Functionality

The module forwards the AQ signal present at socket `IN` to all eight plug outputs `OUT1` to `OUT8` via internal wiring. No signal processing or modification takes place – it is a simple 1:8 distribution (fan-out). The adapters are unidirectional, meaning data and events flow only from the input to the outputs.

Technical Features

- **No Clocked Logic**: The module does not require event I/Os, as the forwarding occurs directly via the adapter interface level.

- **Generic Type**: The module is declared as a generic instance (`GEN_AQ_SPLIT`) – the specific adapter type can be adapted depending on the system environment.
- **Expandability**: The 8 outputs are hardwired; dynamic numbering or configuration is not supported.

State Overview

The module does not have an explicit state machine (ECC). Its functionality is purely static: As soon as the `IN` adapter is active, all outputs output the same signal. There are no operating modes, error states, or lifecycle events.

Application Scenarios

- **Parallel Wiring**: Distribute an analog output signal simultaneously to multiple actuators or measuring points.
- **Test and Simulation Environments**: A signal generator feeds the signal via `IN`, and multiple simulated components are supplied via `OUT1`–`OUT8`.
- **Redundant Control**: For situations where multiple subsystems need to receive the same setpoint without requiring a bus or software branching.

Comparison with Similar Components

- **AQ_SPLIT_4**: Offers only 4 outputs – suitable for lower distribution requirements.
- **AQ_SELECT**: A multiplexer that selects one input from multiple inputs – the opposite of a split function.
- **DQ_SPLIT_8**: Analog component for digital signals (DQ type) – structurally identical, but for a different signal type.

Change Detection

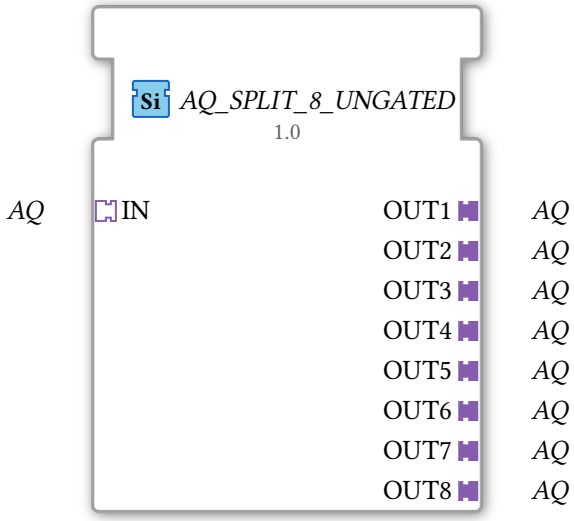
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AQ_SPLIT_8** is a simple, reliable fan-out component for the 1:8 distribution of analog output signals via adapter interfaces. It does not use event logic or state machines and is therefore particularly suitable for clearly defined, static signal distributions in automation systems. Its generic basis facilitates reuse in different projects.

AQ_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AQ_SPLIT_8_UNGATED** is a fan-out function block that distributes a single AQ adapter input to eight identical AQ adapter outputs. It serves to multiply signals in analog output paths, thus enabling the parallel control of multiple identical actuators or subsystems. The implementation is based on the generic mechanism `GEN_AQ_SPLIT` and is designed for unidirectional AQ interfaces.

Interface Structure

Event Inputs

None – the function block operates purely on an adapter basis without event control.

Event Outputs

None – the function block has no event outputs.

Data Inputs

None – all signal transmission occurs via AQ adapters.

Data Outputs

None – see Data Inputs.

Adapters

Adapter	Type	Direction	Description
IN	`adapter::types::unidirectional::AQ`	Socket (Input)	Receives the AQ signal to be distributed.
OUT1 – OUT8	`adapter::types::unidirectional::AQ`	Plug (Output)	Identical outputs that forward the signal from `IN`.

Functionality

The module forwards the AQ signal present at socket `IN` to all eight plug outputs `OUT1` to `OUT8` via internal wiring. No signal processing or modification takes place – it is a simple 1:8 distribution (fan-out). The adapters are unidirectional, meaning data and events flow only from the input to the outputs.

Technical Features

- **No Clocked Logic**: The module does not require event I/Os, as the forwarding occurs directly via the adapter interface level.
- **Generic Type**: The module is declared as a generic instance (`GEN_AQ_SPLIT`) – the specific adapter type can be adapted depending on the system environment.

- **Expandability**: The 8 outputs are hardwired; dynamic numbering or configuration is not supported.

State Overview

The module does not have an explicit state machine (ECC). Its functionality is purely static: As soon as the `IN` adapter is active, all outputs output the same signal. There are no operating modes, error states, or lifecycle events.

Application Scenarios

- **Parallel Wiring**: Distribute an analog output signal simultaneously to multiple actuators or measuring points.
- **Test and Simulation Environments**: A signal generator feeds the signal via `IN`, and multiple simulated components are supplied via `OUT1–OUT8`.
- **Redundant Control**: For situations where multiple subsystems need to receive the same setpoint without requiring a bus or software branching.

Comparison with Similar Components

- **AQ_SPLIT_4**: Offers only 4 outputs – suitable for lower distribution requirements.
- **AQ_SELECT**: A multiplexer that selects one input from multiple inputs – the opposite of a split function.
- **DQ_SPLIT_8**: Analog component for digital signals (DQ type) – structurally identical, but for a different signal type.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

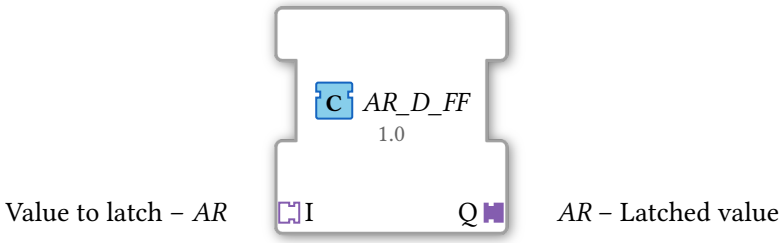
Conclusion

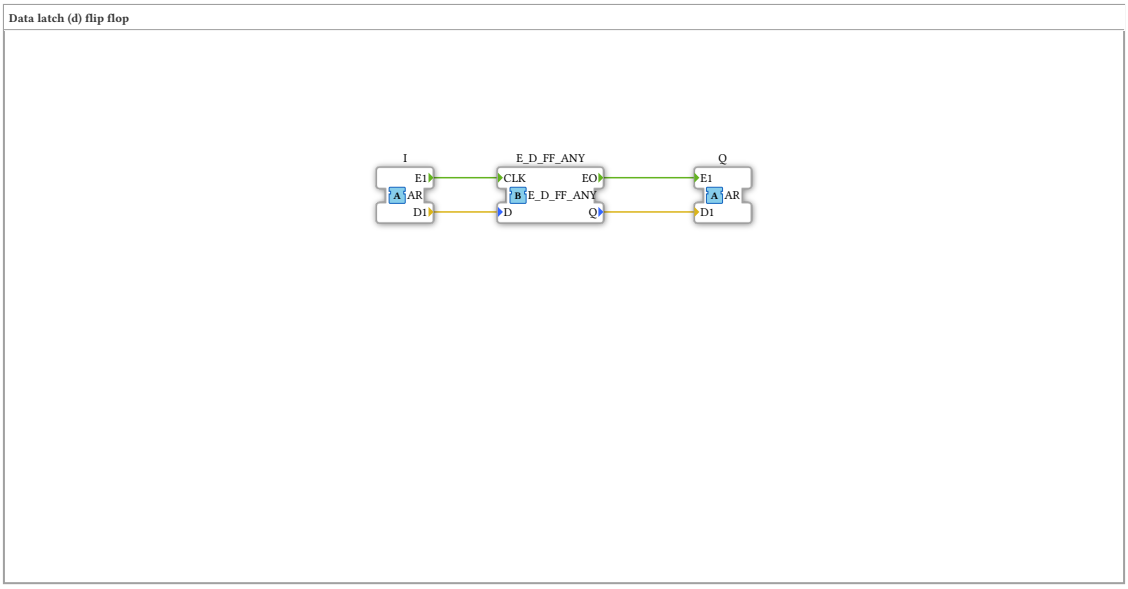
The **AQ_SPLIT_8_UNGATED** is a simple, reliable fan-out component for the 1:8 distribution of analog output signals via adapter interfaces. It does not use event logic or state machines and is therefore particularly suitable for clearly defined, static signal distributions in automation systems. Its generic basis facilitates reuse in different projects.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AR_D_FF





• – ○ • – ○ • – ○ •

Introduction

The function block **AR_D_FF** implements a data-controlled flip-flop (D-latch) according to the IEC 61499-1 Annex A specification. It serves as an encapsulation for the standard function block `iec61499::events::E_D_FF_ANY` and provides its functionality via two unidirectional adapters. The block accepts a data value on a rising edge of the clock signal and holds it until the next rising edge.

Interface Structure

The AR_D_FF has no direct event or data inputs/outputs; all communication takes place via its two adapters.

Event Inputs

- **No direct inputs.**

The clock event is provided via the input adapter **I** (see Adapter section).

Event Outputs

- **No direct outputs.**

The acknowledgment event is provided via the output adapter **Q** (see Adapters section).

Data Inputs

- **No direct outputs.**

The data value to be latched is provided via the input adapter **I** (see Adapters section).

Data Outputs

- **No direct outputs.**

The latched data value is provided via the output adapter **Q** (see Adapters section).

Adapters

Adapter	Direction / Type	Description	Included Elements
I	Socket (`adapter::types::unidirectional::AR`)	Value to be latched.	**E1** (Event Input) – Clock Signal (CLK) **D1** (Data Input) – Data Value
Q	Plug (`adapter::types::unidirectional::AR`)	Latched Value.	**E1** (Event Output) – Acknowledgement after Successful Latch

			D1 (Data Output) – Latched Data Value

Functionality

The component operates on the principle of an edge-triggered D flip-flop:

1. An event at the clock input **I.E1** activates the latch.
2. At the time of the rising edge, the current data value **I.D1** is stored internally.
3. The stored value is immediately output to **Q.D1**.
4. Simultaneously, an event is triggered on **Q.E1** to acknowledge the processing.

The internal state remains unchanged between two clock edges (memory behavior).

Technical Features

- **Adapter-based encapsulation:** The AR_D_FF uses only unidirectional adapters (type `AR`) for input and output. This enables modular, standardized integration into larger adapter networks.
- **Use of the standard component `E_D_FF_ANY`:** The actual flip-flop logic is implemented using the IEC 61499 standard component. Changes to the internal implementation do not affect the externally visible interface.
- **No reset input:** Unlike other flip-flop variants, this component has no reset function. The state can only be changed by a new data value and a clock pulse.

State overview

The component has a single internal memory (the latched data value). State transitions occur exclusively on a rising edge at the clock input **I.E1**:

Old State	Input D1 (current)	New State	Output D1	Event Q.E1
any	*any*	Value of D1	Value of D1	is triggered

There are no other internal states or branches.

Application Scenarios

- **Data Synchronization:** Transfer of a data word at a defined time, e.g., in a cyclic control system.
- **Buffering:** Intermediate storage of a value between two calculation steps.
- **Event-driven signal propagation:** Applications where a value may only be updated when a specific event occurs.

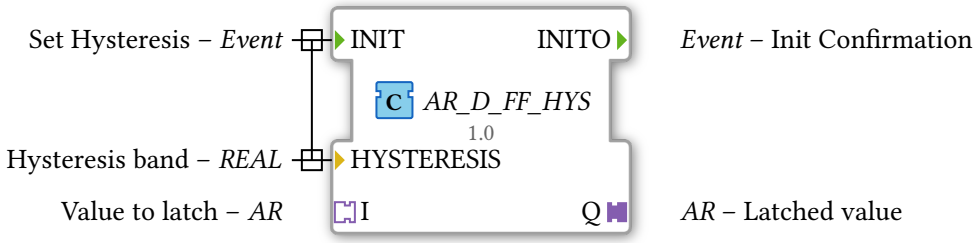
Comparison with similar components

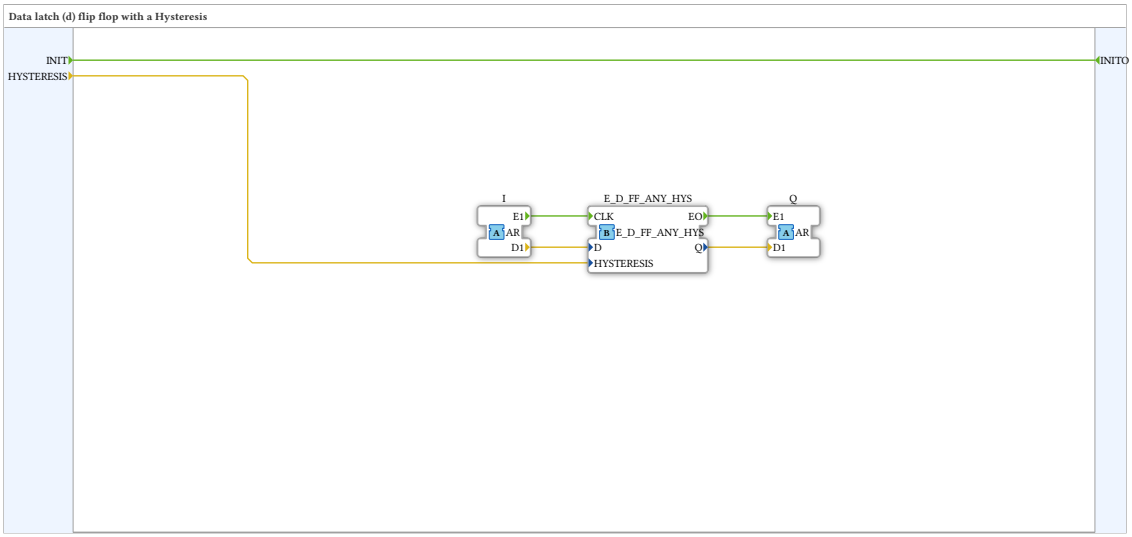
Component	Property	Difference to AR_D_FF
AR_SR_FF	Set-Reset Flip-Flop	Does not latch a data value, but is set/reset via Set/Reset. Has two event inputs.
E_D_FF_ANY	Standard D Flip-Flop without adapter	Same core logic, but direct event/data connections; no adapter encapsulation.
AR_T_FF	Toggle Flip-Flop	Toggles the output on each clock pulse; no data input.

Conclusion

The **AR_D_FF** is a simple, reliable D flip-flop that, thanks to its adapter interface, is particularly suitable for modular and reusable IEC 61499 applications. The encapsulation of this standard component increases its robustness and facilitates integration into larger systems. Specialized flip-flop variants are available for applications requiring a reset or other logic functions.

AR_D_FF_HYS





• – ○ • – ○ • – ○ •

Introduction

The function block **AR_D_FF_HYS** implements a data latch (D) flip-flop with hysteresis. It serves for the stable acquisition and transmission of an analog (real) value, with a hysteresis band suppressing unwanted fluctuations at the output. The block is designed for use in industrial signal processing.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initializes the block and sets the hysteresis band.

Event Outputs

Event	Type	Description
INITO	EInit	Confirms successful initialization.

Data Inputs

Name	Type	Description
HYSTERESIS	REAL	Hysteresis band size.

Data Outputs

(No direct data outputs; the latent value is output via adapter Q)

Adapter

Name	Type	Direction	Description
I	adapter::types::unidirectional::AR	Socket (Input)	Receives the value to be latched (trigger event on E1, data on D1).
Q	adapter::types::unidirectional::AR	Plug (Output)	Outputs the latched value (event on E1, data on D1).

Functionality

The function block internally contains a sub-FB `E_D_FF_ANY_HYS` from the library `logiBUS::signalprocessing::hysteresis`.

- On each rising edge at the event input `I.E1`, the current data value from `I.D1` is taken.
- The received value is only passed on to `Q.D1` if it differs from the last issued value by more than the configured hysteresis (`HYSTERESIS`).
- The event `I.E1` is synchronously forwarded as `Q.E1` to inform the receiver of a new value.
- Initialization (`INIT`) sets the hysteresis band and immediately confirms it with `INITO`. Changing `HYSTERESIS` during operation is not supported; the hysteresis is only applied during the INIT event.
- Initialization (`INIT`) sets the hysteresis band and immediately confirms it with `INITO`. Modifying `HYSTERESIS` during operation is not supported; the hysteresis is only applied during the INIT event.
-

Technical Features

- **Adapter-Based Interface**: Both the input and output use the generic adapter `adapter::types::unidirectional::AR`, allowing the block to be flexibly integrated into existing adapter chains.
- **Hysteresis Function**: Suppresses noise and oscillations at threshold values, ensuring a clean, stable output.
- **EPL 2.0**: The block is licensed under the Eclipse Public License 2.0 and can be freely used and modified.
- **Internal Implementation**: The actual hysteresis logic is executed by a specialized sub-block (`E_D_FF_ANY_HYS`), improving maintainability and reusability.

State Overview

The block has an implicit memory state for the last output value. Since this is a D flip-flop with hysteresis, three logical ranges can be distinguished:

- **Input ≥ (Output + Hysteresis/2)** ⟶ Output is set to the new input.
- **Input ≤ (Output – Hysteresis/2)** ⟶ Output is set to the new input.
- **Value within the hysteresis range** ⟶ Output remains unchanged (history).

There are no explicit state machines; the component operates purely event-driven.

Application Scenarios

- **Signal Smoothing**: Removal of high-frequency noise from analog measured values.
- **Threshold Switch**: Defined on and off points for binary decisions (e.g., temperature control).
- **Edge Detection with Hysteresis**: Reliable detection of rising or falling edges of an analog signal.
- **Level Adjustment**: Stable tracking of an output value to a fluctuating input.

Comparison with Similar Function Blocks

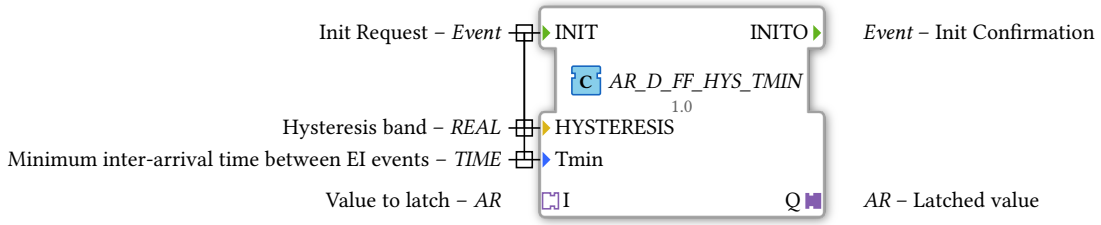
Function Block	Feature
AR_D_FF (without hysteresis)	Instantly adopts any new value; sensitive to noise.
AR_D_FF_HYS (this function block)	Uses a hysteresis band to suppress flicker.

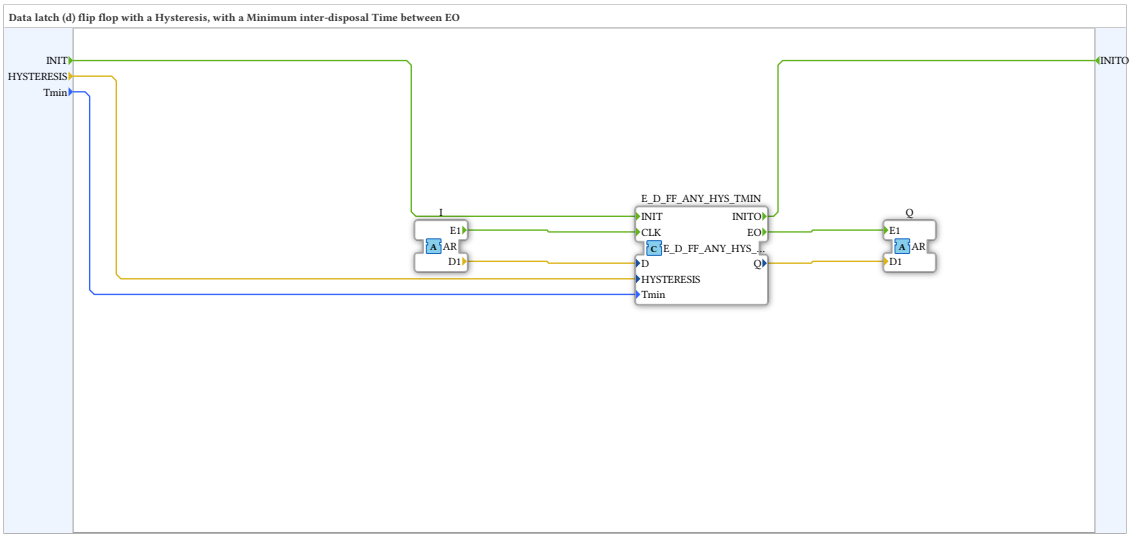
Schmitt Trigger	Implements similar hysteresis, but usually for binary (Boolean) signals. This function block operates with analog (REAL) values.

Conclusion

The **AR_D_FF_HYS** is a robust, adapter-based function block for the reliable transmission of analog values with adjustable hysteresis. It is particularly suitable for applications where interference occurs on the signal line or where defined switching behavior is required. Its simple initialization and clear interface structure facilitate integration into automation systems.

AR_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Introduction

The **AR_D_FF_HYS_TMIN** function block implements a data latch (D flip-flop) with a hysteresis function and a minimal delay between successive output events at the output adapter. It serves for the stable acquisition and transmission of analog values via unidirectional AR adapters, suppressing edge noise and limiting the output frequency.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initialization request; passes the parameters HYSTERESIS and Tmin to the internal logic.

Event Outputs

Event	Type	Description

INITO	EInit	Confirmation of successful initialization.

Data Inputs

Variable	Type	Description
HYSTERESIS	REAL	Hysteresis band: A change in the input value must exceed this amount for a new value to be latched.
Tmin	TIME	Minimum time between two output events (inter-disposal time). Prevents overly rapid switching.

Data Outputs

The FB does not have its own data outputs; the output data is provided via the **Q** adapter.

Adapter

Name	Type	Direction	Description
I	adapter::types::unidirectional::AR (Socket)	Input	Provides the value to be latched via **I.D1** and the clock signal via **I.E1**.
Q	adapter::types::unidirectional::AR (Plug)	Output	Provides the latched value via **Q.D1** and signals an update via **Q.E1**.

Functionality

The function block works internally with a sub-function block of type `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS_TMIN`.

1. **Initialization**: During the **INIT** event, the parameters **HYSTERESIS** and **Tmin** are passed to the internal logic. After successful initialization, **INITO** is output.
2. **Data Acquisition**: At each clock event on adapter **I.E1**, the currently present value **I.D1** is compared with the last stored value.

- A change is only made if the absolute difference between **I.D1** and the stored value is greater than **HYSTERESIS**.
- If the difference is smaller, the old value remains unchanged (hysteresis function).

1. **Time-Limited Output**: As soon as the new value has been acquired, the output signal **Q.D1** is updated. However, the corresponding event **Q.E1** is only sent if at least the time interval **Tmin** has elapsed since the last **Q.E1**. This limits the maximum output frequency.

- A change is only made if the time interval **Tmin** has elapsed since the last **Q.E1**. The entire behavior can be viewed as a clock-controlled, hysteresis-enabled signal-hold block with output blocking.

Technical Features

- **Hysteresis**: Suppresses noise and small fluctuations in the input signal. A new state is only assumed when the change exceeds the **HYSTERESIS** value.
- **Minimum Interval Time (Tmin)**: Prevents an excessively high event rate at the output. This is particularly advantageous for downstream, computationally intensive, or communication-limited systems.
- **Adapter-Based Interfaces**: Both input and output use the unidirectional AR adapter, which combines data and events. This simplifies the modular chaining of signal processing blocks.

State Overview

The function block (FB) does not have an explicit finite state machine (FSM) at the top level, as the state logic is encapsulated in the internal sub-FB. The logical behavior, however, corresponds to a D flip-flop with extended conditions:

- **Idle State**: Waiting for the next clock cycle (I.E1).
- **Check State**: Comparison of the new value with the stored value, taking hysteresis into account.
- **Output State**: Update of **Q.D1** and, if **Tmin** has expired, send of **Q.E1**.

A graphical state machine can be derived from the internal sub-FB if required.

Application Scenarios

- **Measurement Acquisition with Noise Suppression**: Smoothing of sensor signals (e.g., temperature, pressure) and transmission only when significant changes occur.
- **Time-Controlled Process Outputs**: Control of actuators that must not be driven faster than a predefined clock frequency.
- **Signal Conditioning in Fieldbus Systems**: Reduction of bus load by limiting the event rate without information loss due to hysteresis.
- **Cascading Flip-Flops**: Used as a stabilized intermediate stage in data chains with multiple sampling stages.

Comparison with Similar Components

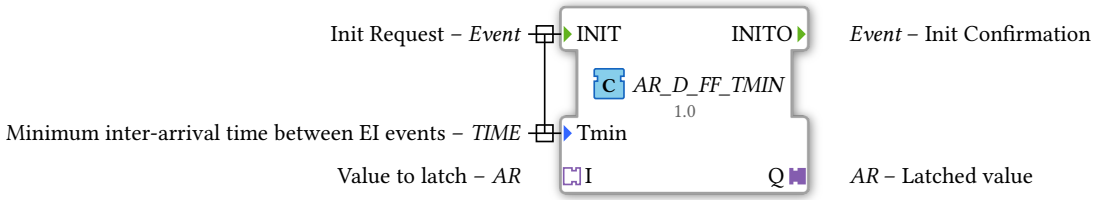
- **Simple D Flip-Flop (e.g., E_D_FF)**: Has neither hysteresis nor an output time limit. Changes its output on every clock cycle, regardless of the change magnitude or frequency.
- **Hysteresis Flip-Flop (e.g., E_D_FF_HYS)**: Includes hysteresis but no minimum interval time. Events can follow each other very quickly if the hysteresis is constantly exceeded.
- **Limited-Time Flip-Flop (No Hysteresis)**: Only the minimum output time is observed; small value changes are still processed immediately, which can lead to oscillations.

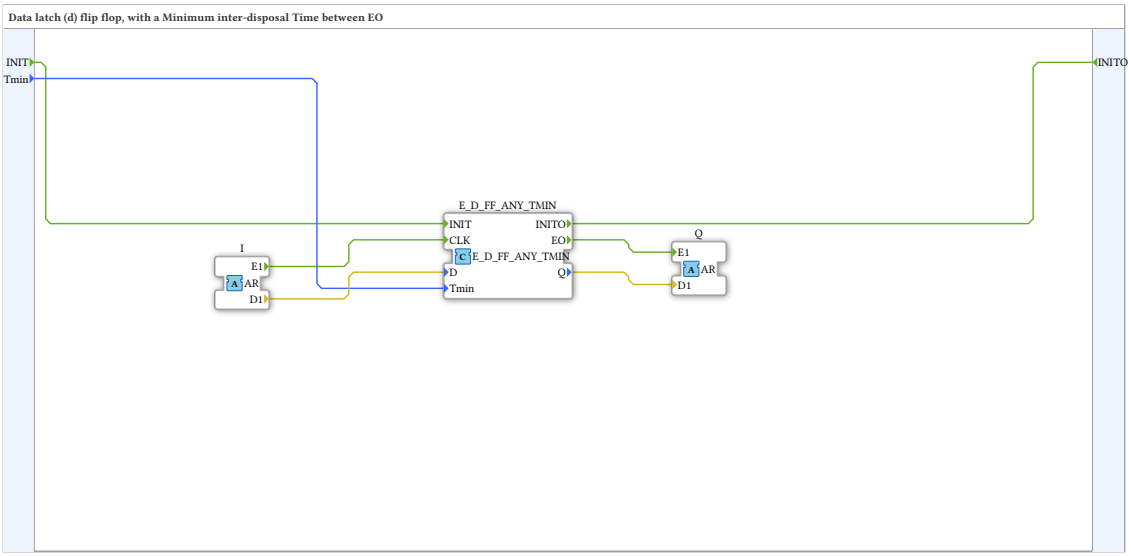
The **AR_D_FF_HYS_TMIN** combines both mechanisms, thus offering a robust solution for practical measurement and control tasks.

Conclusion

The **AR_D_FF_HYS_TMIN** function block is a versatile tool for reliable signal routing in automation systems. By combining hysteresis and minimal inter-disposal time, typical problems such as noise, flicker, and overload of downstream processing stages are effectively avoided. The adapter-based interface facilitates integration into existing modular architectures.

AR_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AR_D_FF_TMIN** implements a data latch (D flip-flop) with one input and one output adapter. It is characterized by the optional specification of a minimum time (`Tmin`) between successive

event outputs. The core of the function block is based on the standard function block `E_D_FF_ANY_TMIN`, which handles the actual flip-flop logic and the timing.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request (with parameter `Tmin`)

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Acknowledgement

Data Inputs

Name	Type	Comment
Tmin	TIME	Minimum Delay Between Two EO Events

Data Outputs

No standalone data outputs; the value is output via the `Q` adapter.

Adapter

Direction	Name	Type	Comment
Socket	I	`adapter::types::unidirectional::AR`	Value to be transferred
Plug	Q	`adapter::types::unidirectional::AR`	Latched value

The adapter `adapter::types::unidirectional::AR` combines one event (`E1`) and one data word (`D1`) per direction.

Functionality

The function block works internally with the function block `E_D_FF_ANY_TMIN`:

1. **Initialization**: An event at `INIT` passes the parameter `Tmin` to the internal function block. Then, `INITO` is output.

The adapter `INITO` is output. 2. **Flip-flop operation**: An event at the adapter input `I.E1` (socket) is forwarded to the **CLK** input of the internal flip-flop. Simultaneously, the data value `I.D1` is passed to the **D** input of the flip-flop.

1. **Output**: The internally generated state (`Q`) and the associated event (`EO`) are passed to the adapter output `Q` (plug) – after adhering to the set minimum time `Tmin`. The event `Q.E1` and the data value `Q.D1` represent the latched value.
2. **Timing**: The parameter `Tmin` defines the minimum time that must elapse between two consecutive `EO` events. If the frequency of input events exceeds this limit, the output is delayed or blocked accordingly.

Technical Features

- **Adapter-Based Communication**: The function block uses only unidirectional AR adapters, making it suitable for loosely coupled or distributed systems.
- **Reuse of a Standard Function Block**: The core logic is provided by the IEC 61499 standard function block `E_D_FF_ANY_TMIN` – thus, the functionality is standards-compliant and extensible.
- **Configurable Lockout Time**: The minimum time `Tmin` is passed for initialization and cannot be changed during runtime (except via a new INIT cycle).

State Overview

The FB itself does not have its own state diagram. Its behavior is entirely determined by the embedded standard function block `E_D_FF_ANY_TMIN`. After the INIT phase, this block is in a wait state and switches to an output state with each CLK event, provided the `Tmin` time has elapsed.

Application Scenarios

- **Signal Debouncing**: When a sensor signal (e.g., via `I`) should only be passed on with a minimum pause to suppress bounce or noise.
- **Event Rate Limitation**: In control or monitoring systems where downstream components must not be operated faster than a defined rate.
- **Secure Data Transfer**: The latched value remains stable until the next CLK event (at the adapter input) arrives.

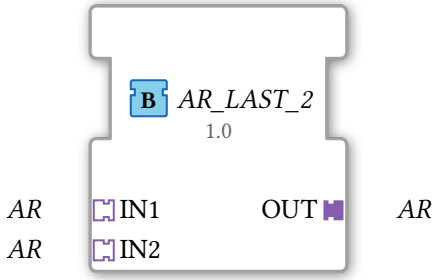
Comparison with Similar Function Blocks

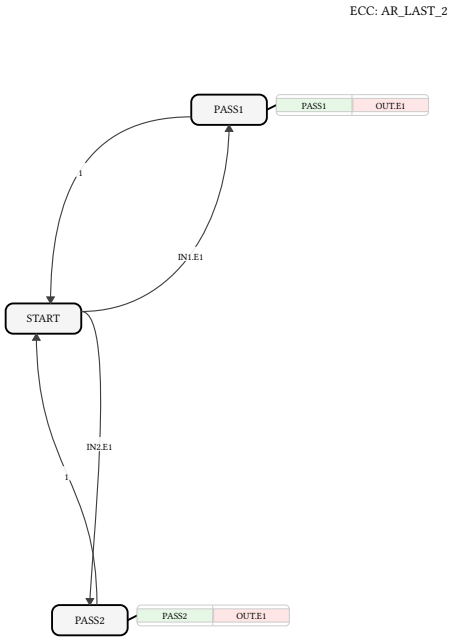
- **AR_D_FF** (without TMIN): A pure D flip-flop function block without a time limit on the output events. It is suitable for real-time applications without delay requirements.
- **AR_D_FF_TMIN**: Extends the basic function block with the ability to limit the event rate, which is often necessary in industrial control systems with time constraints.
- **AR_SR_FF**: Implements set-reset logic – unlike the D flip-flop implemented here, the output is not clock-controlled.

Conclusion

The `AR_D_FF_TMIN` is a functionally enhanced D flip-flop device that combines latch functionality with a configurable minimum time between output events. Its adapter-based interface and the use of a standardized core make it a flexible and standards-compliant component for time-critical control applications in the IEC 61499 environment.

AR_LAST_2





Introduction

The AR_LAST_2 function block merges two same-type REAL adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: REAL adapter (unidirectional) - first signal source
- **IN2**: REAL adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: REAL adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

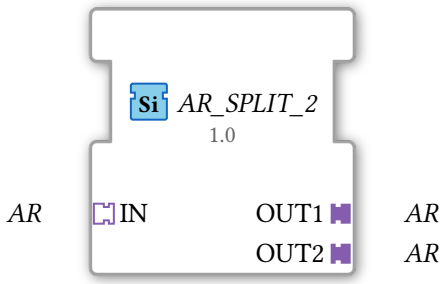
Application Scenarios

- Merging two equally-valid REAL signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same REAL value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AR_LAST_2 is the inverse of AR_SPLIT_2: AR_SPLIT_2 distributes 1 input to 2 outputs, AR_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use AX_AE_MERGE instead, which merges only the event without altering the data value.

AR_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The **AR_SPLIT_2** is a generic function block that splits an incoming adapter of type `adapter::types::unidirectional::AR` into two separate outputs (OUT1, OUT2). It serves to forward a signal to two subsequent function blocks without modifying the data. The function block is implemented as a generic type (`GEN_AR_SPLIT`) and is parameterized at runtime.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AR`	IN	Socket	Input signal (AR)
`adapter::types::unidirectional::AR`	OUT1	Plug	First output (identical to IN)
`adapter::types::unidirectional::AR`	OUT2	Plug	Second output (identical to IN)

Functionality

This module forwards the AR signal present at socket **IN** unchanged to both plugs **OUT1** and **OUT2**. No data manipulation, filtering, or delay occurs. The distribution is purely structural: Each output receives its own copy of the reference to the underlying AR object.

Technical Features

- **Generic Implementation**: The function block uses the generic class name attribute (`eclipse4diac::core::GenericClassName`) with the value `'GEN_AR_SPLIT'`, allowing for application-specific configuration.
- **Unidirectional Adapters**: All adapters are of type `adapter::types::unidirectional::AR` (forward direction only).

- **No State Automation**: The function block does not have an explicit ECC (Execution Control Chart); signal routing is direct and event-independent.

State Overview

The function block does not implement state automation. Its functionality is limited to passively routing the input signal to both outputs. Therefore, state visualization is not required.

Application Scenarios

- **Signal Distribution**: Splitting an AR-based sensor signal to two parallel-operating control components.
- **Redundancy**: Providing the same AR data stream for a primary and a redundant system.
- **Debugging**: Connecting an analysis or logging module in parallel to the existing path without interrupting the original signal chain.

Comparison with Similar Modules

Modules such as `AR_SPLIT_3` or `AR_SPLIT_N` offer similar functionality, distributing a signal to three or N outputs, respectively. The choice depends on the required number of outputs. Cascading multiple `AR_SPLIT_2` instances is also possible, but increases complexity compared to a dedicated multi-output splitter.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AR_SPLIT_2** is a simple yet essential module for multiplying an AR signal in IEC 61499-based control systems. Its generic design and clear interface make it the first choice when a signal needs to be passed between two independent target components.

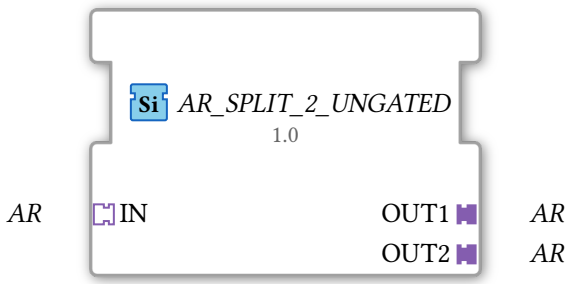
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AR_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **AR_SPLIT_2_UNGATED** is a generic function block that splits an incoming adapter of type `adapter::types::unidirectional::AR` into two separate outputs (OUT1, OUT2). It serves to forward a signal to two subsequent function blocks without modifying the data. The function block is implemented as a generic type (`GEN_AR_SPLIT`) and is parameterized at runtime.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AR`	IN	Socket	Input signal (AR)
`adapter::types::unidirectional::AR`	OUT1	Plug	First output (identical to IN)
`adapter::types::unidirectional::AR`	OUT2	Plug	Second output (identical to IN)

Functionality

This module forwards the AR signal present at socket **IN** unchanged to both plugs **OUT1** and **OUT2**. No data manipulation, filtering, or delay occurs. The distribution is purely structural: Each output receives its own copy of the reference to the underlying AR object.

Technical Features

- **Generic Implementation**: The function block uses the generic class name attribute (`eclipse4diac::core::GenericClassName`) with the value `'GEN_AR_SPLIT'`, allowing for application-specific configuration.
- **Unidirectional Adapters**: All adapters are of type `adapter::types::unidirectional::AR` (forward direction only).
- **No State Automation**: The function block does not have an explicit ECC (Execution Control Chart); signal routing is direct and event-independent.

State Overview

The function block does not implement state automation. Its functionality is limited to passively routing the input signal to both outputs. Therefore, state visualization is not required.

Application Scenarios

- **Signal Distribution**: Splitting an AR-based sensor signal to two parallel-operating control components.
- **Redundancy**: Providing the same AR data stream for a primary and a redundant system.
- **Debugging**: Connecting an analysis or logging module in parallel to the existing path without interrupting the original signal chain.

Comparison with Similar Modules

Modules such as `AR_SPLIT_3` or `AR_SPLIT_N` offer similar functionality, distributing a signal to three or N outputs, respectively. The choice depends on the required number of outputs. Cascading multiple `AR_SPLIT_2_UNGATED` instances is also possible, but increases complexity compared to a dedicated multi-output splitter.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

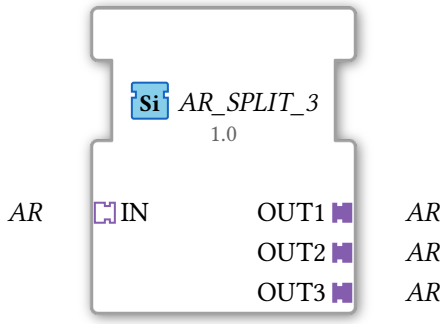
The **AR_SPLIT_2_UNGATED** is a simple yet essential module for multiplying an AR signal in IEC 61499-based control systems. Its generic design and clear interface make it the first choice when a signal needs to be passed between two independent target components.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AR_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_SPLIT_3** serves as a generic splitter for adapter connections. It distributes an incoming data and event stream (via a `AR` adapter) to three separate, identically structured output adapters. This enables efficient reuse of information in automation processes without duplicating the logic.

Interface Structure

Event Inputs

None – the block has only adapter interfaces. Events are transmitted implicitly via the socket `IN` (of type `AR`).

Event Outputs

None – Events are passed via the plugs `OUT1`, `OUT2`, and `OUT3` (type `AR`).

Data Inputs

None – All data is provided via the adapter `IN`.

Data Outputs

None – All data is output via the three output adapters (`OUT1`, `OUT2`, `OUT3`).

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AR`	Incoming adapter as the source of the data/events.
Plug	`OUT1`	`adapter::types::unidirectional::AR`	First outgoing adapter (Destination 1).
Plug	`OUT2`	`adapter::types::unidirectional::AR`	Second outgoing adapter (Destination 2).

Plug	OUT3	adapter::types::unidirectional::AR	Third outgoing adapter (Destination 3).

Functionality

This function block implements a **1:3 distribution** at the adapter level. A AR adapter (unidirectional) connected to IN is internally routed to the three output adapters OUT1, OUT2, and OUT3.

Every event or data packet received at IN is simultaneously forwarded to all three outputs. The distribution occurs without delay and without data modification. The function block is generic and can be used with any AR adapter profile – its internal logic automatically adapts to the structure of the adapter type used.

…

Technical Features

- **Generic Type**: The function block (FB) is designed as a generic block (GEN_AR_SPLIT) and can be used with different AR adapter definitions (unidirectional), provided the interfaces are compatible.
- **Unidirectional Adapters**: Both inputs and outputs are unidirectional, meaning the data/event direction is fixed (from sender to receiver). Feedback is not possible with this block.
- **Easy Scaling**: The outputs are split without additional configuration. For different numbers of outputs, variants such as AR_SPLIT_2 or a configurable splitter (AR_SPLIT_N) can be used.
- **No State Change**: The block has no internal memory and no state machine – every event is replicated immediately and completely.

State Overview

The function block is **stateless** (combinatorial). There are no internal states or sequential processes. Distribution occurs purely through the connection of the adapter signals.

Application Scenarios

- **Distributing sensor data** to multiple parallel processing logics (e.g., evaluation, visualization, logging).
- **Controlling multiple actuator groups** with an identical control signal (e.g., parallel valve or motor control).
- **Splitting a control event** into different function blocks without copying the source logic.
- **Debugging and Monitoring**: A signal can be routed to a physical receiver and, in parallel, to a monitoring component.

Comparison with Similar Function Blocks

Function Block	Description
AR_SPLIT_2	Distribution to two outputs – more compact for smaller applications.
AR_SPLIT_N	Distribution to a configurable number of outputs (generic).
AR_CONNECT	Simple 1:1 connection (no distribution).

Unlike dedicated splitters with data or event inputs/outputs, the AR_SPLIT_3 operates exclusively via adapters, making it particularly flexible and type-independent.

Change Detection

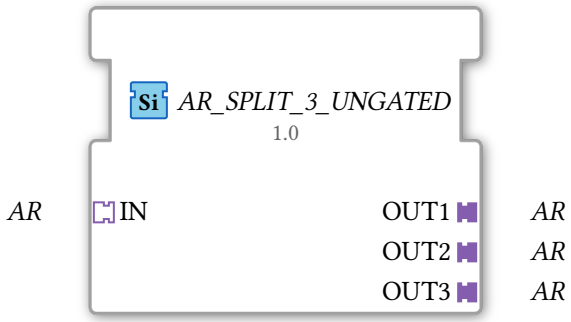
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AR_SPLIT_3 is a simple yet useful generic function block for distributing adapter signals to multiple outputs. It reduces the effort required for signal duplication in IEC 61499 applications and, due to its generic nature, contributes to the reusability and clarity of control logic. Thanks to its pure adapter approach, it can be integrated into a wide variety of environments without modification.

AR_SPLIT_3_UNGATED

i UNGATED variant: This block is the ungated version of AR_SPLIT_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_SPLIT_3_UNGATED** serves as a generic splitter for adapter connections. It distributes an incoming data and event stream (via a AR adapter) to three separate, identically structured output adapters. This enables efficient reuse of information in automation processes without duplicating the logic.

Interface Structure

Event Inputs

None – the block has only adapter interfaces. Events are transmitted implicitly via the socket IN (of type AR).

Event Outputs

None – Events are passed via the plugs OUT1, OUT2, and OUT3 (type AR).

Data Inputs

None – All data is provided via the adapter IN.

Data Outputs

None – All data is output via the three output adapters (OUT1, OUT2, OUT3).

Adapters

Direction	Name	Type	Description
Socket	IN	adapter::types::unidirectional::AR	Incoming adapter as the source of the data/events.
Plug	OUT1	adapter::types::unidirectional::AR	First outgoing adapter (Destination 1).
Plug	OUT2	adapter::types::unidirectional::AR	Second outgoing adapter (Destination 2).
Plug	OUT3	adapter::types::unidirectional::AR	Third outgoing adapter (Destination 3).

Functionality

This function block implements a **1:3 distribution** at the adapter level. A AR adapter (unidirectional) connected to IN is internally routed to the three output adapters OUT1, OUT2, and OUT3.

Every event or data packet received at IN is simultaneously forwarded to all three outputs. The distribution occurs without delay and without data modification. The function block is generic and can be used with any AR adapter profile – its internal logic automatically adapts to the structure of the adapter type used.

...

Technical Features

- **Generic Type**: The function block (FB) is designed as a generic block (GEN_AR_SPLIT) and can be used with different AR adapter definitions (unidirectional), provided the interfaces are compatible.
- **Unidirectional Adapters**: Both inputs and outputs are unidirectional, meaning the data/event direction is fixed (from sender to receiver). Feedback is not possible with this block.
- **Easy Scaling**: The outputs are split without additional configuration. For different numbers of outputs, variants such as AR_SPLIT_2 or a configurable splitter (AR_SPLIT_N) can be used.
- **No State Change**: The block has no internal memory and no state machine – every event is replicated immediately and completely.

State Overview

The function block is **stateless** (combinatorial). There are no internal states or sequential processes. Distribution occurs purely through the connection of the adapter signals.

Application Scenarios

- **Distributing sensor data** to multiple parallel processing logics (e.g., evaluation, visualization, logging).
- **Controlling multiple actuator groups** with an identical control signal (e.g., parallel valve or motor control).
- **Splitting a control event** into different function blocks without copying the source logic.
- **Debugging and Monitoring**: A signal can be routed to a physical receiver and, in parallel, to a monitoring component.

Comparison with Similar Function Blocks

Function Block	Description
AR_SPLIT_2	Distribution to two outputs – more compact for smaller applications.
AR_SPLIT_N	Distribution to a configurable number of outputs (generic).
AR_CONNECT	Simple 1:1 connection (no distribution).

Unlike dedicated splitters with data or event inputs/outputs, the AR_SPLIT_3_UNGATED operates exclusively via adapters, making it particularly flexible and type-independent.

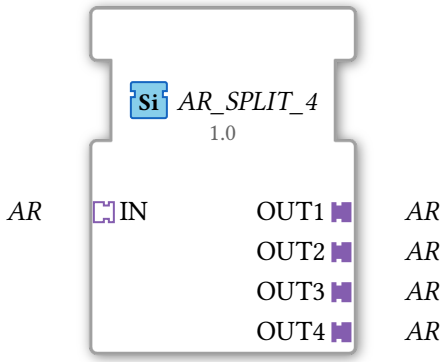
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AR_SPLIT_3_UNGATED is a simple yet useful generic function block for distributing adapter signals to multiple outputs. It reduces the effort required for signal duplication in IEC 61499 applications and, due to its generic nature, contributes to the reusability and clarity of control logic. Thanks to its pure adapter approach, it can be integrated into a wide variety of environments without modification.

AR_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_SPLIT_4** is used to split one incoming AR adapter (unidirectional) into four outgoing AR adapters. It is implemented as a generic function block (`GenericClassName = 'GEN_AR_SPLIT'`) and enables simple signal multiplication for adapter-based communication in control systems.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

Name	Type	Comment
(none)	–	–

Data Outputs

Name	Type	Comment
(none)	–	–

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AR`	**IN**	Socket (Input)	Incoming AR adapter signal, which is distributed to the four outputs.
`adapter::types::unidirectional::AR`	**OUT1**	Plug (Output)	First outgoing AR channel.
`adapter::types::unidirectional::AR`	**OUT2**	Plug (Output)	Second outgoing AR channel.
`adapter::types::unidirectional::AR`	**OUT3**	Plug (Output)	Third outgoing AR channel.
`adapter::types::unidirectional::AR`	**OUT4**	Plug (Output)	Fourth outgoing AR channel.

Functionality

The function block receives an AR adapter signal via socket **IN** and forwards it unchanged to all four plugs **OUT1** to **OUT4**. Distribution occurs in parallel – every event or data value received via **IN** is simultaneously available on all outputs. The function block itself does not perform any logical processing or transformation; it functions purely as a splitter for unidirectional AR adapters.

Technical Features

- **Generic Type**: The function block is defined as a generic function block (`GenericClassName`), which means that the specific AR adapter type is determined at compile time. This increases reusability in various applications.
- **No State Machine**: The function block (FB) has no Execution Control Chart (ECC) and no internal states. All operations are purely combinatorial or event-driven.
- **Unidirectional**: The adapter type is specified as unidirectional; feedback from the target block is not provided.

State Overview

Since the FB has no state machine, no explicit states exist. The behavior is deterministic: An incoming trigger at socket **IN** is immediately forwarded to all outputs.

Application Scenarios

- **Signal Multiplication**: A control signal (e.g., "Start" or "Reset") should reach several parallel function blocks simultaneously.
- **Monitoring and Parallel Processing**: A measured value or an alarm is distributed to several independent evaluation blocks.
- **Testing and Simulation**: During development, a signal can be recorded or simulated while simultaneously being sent to the actual target block.

Comparison with Similar Function Blocks

- **AR_SPLIT_2**: Splits an AR signal into two outputs – suitable for smaller distributions.
- **AR_SPLIT_N**: A non-standard, configurable splitter (e.g., via parameters) designed for any number of outputs.
- **AR_MERGE**: Merges multiple AR signals into one – the inverse function of AR_SPLIT_4.

Compared to these alternatives, AR_SPLIT_4 offers a fixed number of four outputs, enabling a compact and clear implementation without configuration overhead.

Change Detection

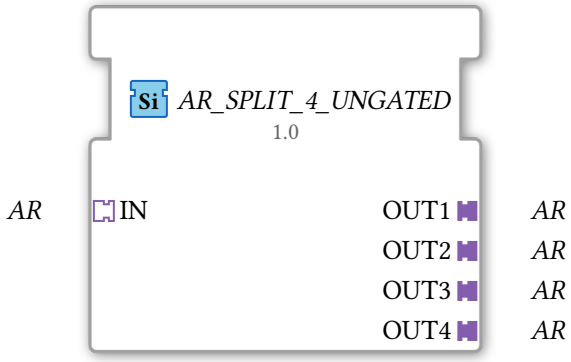
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AR_SPLIT_4** is a simple yet useful generic function block for distributing a unidirectional AR adapter signal across four parallel paths. Due to its streamlined design (no event/data interfaces, no state machine), it is ideally suited for all applications where a signal is needed multiple times without introducing additional logic.

AR_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_SPLIT_4_UNGATED** is used to split one incoming AR adapter (unidirectional) into four outgoing AR adapters. It is implemented as a generic function block (`GenericClassName = 'GEN_AR_SPLIT'`) and enables simple signal multiplication for adapter-based communication in control systems.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

Name	Type	Comment
(none)	–	–

Data Outputs

Name	Type	Comment
(none)	–	–

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AR`	**IN**	Socket (Input)	Incoming AR adapter signal, which is distributed to the four outputs.
`adapter::types::unidirectional::AR`	**OUT1**	Plug (Output)	First outgoing AR channel.
`adapter::types::unidirectional::AR`	**OUT2**	Plug (Output)	Second outgoing AR channel.
`adapter::types::unidirectional::AR`	**OUT3**	Plug (Output)	Third outgoing AR channel.
`adapter::types::unidirectional::AR`	**OUT4**	Plug (Output)	Fourth outgoing AR channel.

Functionality

The function block receives an AR adapter signal via socket **IN** and forwards it unchanged to all four plugs **OUT1** to **OUT4**. Distribution occurs in parallel – every event or data value received via **IN** is simultaneously available on all outputs. The function block itself does not perform any logical processing or transformation; it functions purely as a splitter for unidirectional AR adapters.

Technical Features

- **Generic Type**: The function block is defined as a generic function block (`GenericClassName`), which means that the specific AR adapter type is determined at compile time. This increases reusability in various applications.
- **No State Machine**: The function block (FB) has no Execution Control Chart (ECC) and no internal states. All operations are purely combinatorial or event-driven.
- **Unidirectional**: The adapter type is specified as unidirectional; feedback from the target block is not provided.

State Overview

Since the FB has no state machine, no explicit states exist. The behavior is deterministic: An incoming trigger at socket **IN** is immediately forwarded to all outputs.

Application Scenarios

- **Signal Multiplication**: A control signal (e.g., "Start" or "Reset") should reach several parallel function blocks simultaneously.
- **Monitoring and Parallel Processing**: A measured value or an alarm is distributed to several independent evaluation blocks.
- **Testing and Simulation**: During development, a signal can be recorded or simulated while simultaneously being sent to the actual target block.

Comparison with Similar Function Blocks

- **AR_SPLIT_2**: Splits an AR signal into two outputs – suitable for smaller distributions.
- **AR_SPLIT_N**: A non-standard, configurable splitter (e.g., via parameters) designed for any number of outputs.
- **AR_MERGE**: Merges multiple AR signals into one – the inverse function of AR_SPLIT_4_UNGATED.

Compared to these alternatives, AR_SPLIT_4_UNGATED offers a fixed number of four outputs, enabling a compact and clear implementation without configuration overhead.

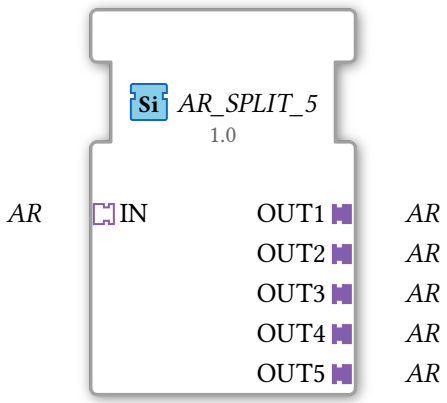
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AR_SPLIT_4_UNGATED** is a simple yet useful generic function block for distributing a unidirectional AR adapter signal across four parallel paths. Due to its streamlined design (no event/data interfaces, no state machine), it is ideally suited for all applications where a signal is needed multiple times without introducing additional logic.

AR_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The AR_SPLIT_5 function block is used to split an incoming AR adapter into five identical AR adapter outputs. It is a generic function block specifically designed for distributing unidirectional adapters (type: `adapter::types::unidirectional::AR`).

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapter

- **Socket (Input)**:
- `IN`: Type `adapter::types::unidirectional::AR` – Receives a single AR adapter.
- **Plugs (Outputs)**:
- `OUT1`: Type `adapter::types::unidirectional::AR`
- `OUT2`: Type `adapter::types::unidirectional::AR`
- `OUT3`: Type `adapter::types::unidirectional::AR`
- `OUT4`: Type `adapter::types::unidirectional::AR`
- `OUT5`: Type `adapter::types::unidirectional::AR`

All outputs are identical and pass the incoming AR adapter through unchanged.

Functionality

The function block operates as a passive distributor: As soon as an AR adapter is connected to the input `IN`, it is duplicated to the five outputs `OUT1` to `OUT5`. Each connection receives an independent but identical copy of the adapter. No data processing or modification takes place.

Technical Features

- **Unidirectionality**: The adapters of type `AR` are exclusively unidirectional, meaning signals flow only from the input to the outputs.
- **Generic Implementation**: The function block is implemented as a generic type (`GEN_AR_SPLIT`), which allows for easy reuse and adaptation to other adapter types.
- **No State Logic**: The function block has no internal states or events; The splitting is achieved purely through interconnection technology.

State Overview

The AR_SPLIT_5 does not have its own state machine. Its behavior is static and purely structural: The input is directly passed through to all outputs.

Application Scenarios

- **Signal Distribution**: When an AR adapter provides a control or data signal that needs to be passed on to several downstream components (e.g., actuators, sensors).
- **Module Cascading**: In complex automation systems where an AR signal needs to be split to different subsystems.

Comparison with Similar Function Blocks

- **AR_SPLIT_2, AR_SPLIT_3, AR_SPLIT_4**: These function blocks split an AR adapter into two, three, or four outputs, respectively. The AR_SPLIT_5 is the extension to five outputs.

- **AR_MERGE_***: Unlike split function blocks, merge function blocks combine multiple AR adapters into one.

Change Detection

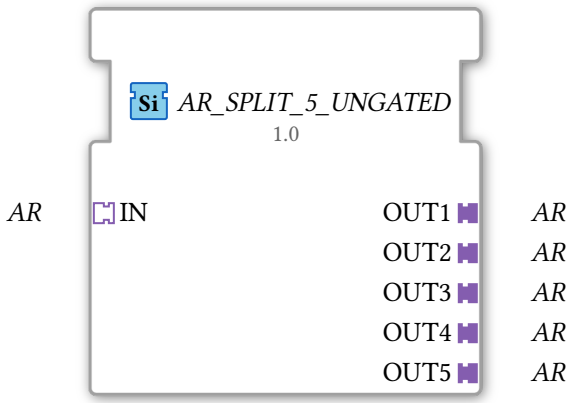
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AR_SPLIT_5 is a simple yet useful function block for distributing a unidirectional AR adapter across five independent outputs. Its generic and eventless structure makes it a fundamental tool for adapter-based communication in IEC 61499 systems.

AR_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The AR_SPLIT_5_UNGATED function block is used to split an incoming AR adapter into five identical AR adapter outputs. It is a generic function block specifically designed for distributing unidirectional adapters (type: `adapter::types::unidirectional::AR`).

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapter

- **Socket (Input)**:

- `IN`: Type `adapter::types::unidirectional::AR` – Receives a single AR adapter.
- **Plugs (Outputs)**:
- `OUT1`: Type `adapter::types::unidirectional::AR`
- `OUT2`: Type `adapter::types::unidirectional::AR`
- `OUT3`: Type `adapter::types::unidirectional::AR`
- `OUT4`: Type `adapter::types::unidirectional::AR`
- `OUT5`: Type `adapter::types::unidirectional::AR`

All outputs are identical and pass the incoming AR adapter through unchanged.

Functionality

The function block operates as a passive distributor: As soon as an AR adapter is connected to the input `IN`, it is duplicated to the five outputs `OUT1` to `OUT5`. Each connection receives an independent but identical copy of the adapter. No data processing or modification takes place.

Technical Features

- **Unidirectionality**: The adapters of type `AR` are exclusively unidirectional, meaning signals flow only from the input to the outputs.
- **Generic Implementation**: The function block is implemented as a generic type (`GEN_AR_SPLIT`), which allows for easy reuse and adaptation to other adapter types.
- **No State Logic**: The function block has no internal states or events; The splitting is achieved purely through interconnection technology.

State Overview

The AR_SPLIT_5_UNGATED does not have its own state machine. Its behavior is static and purely structural: The input is directly passed through to all outputs.

Application Scenarios

- **Signal Distribution**: When an AR adapter provides a control or data signal that needs to be passed on to several downstream components (e.g., actuators, sensors).
- **Module Cascading**: In complex automation systems where an AR signal needs to be split to different subsystems.

Comparison with Similar Function Blocks

- **AR_SPLIT_2, AR_SPLIT_3, AR_SPLIT_4**: These function blocks split an AR adapter into two, three, or four outputs, respectively. The AR_SPLIT_5_UNGATED is the extension to five outputs.
- **AR_MERGE_***: Unlike split function blocks, merge function blocks combine multiple AR adapters into one.

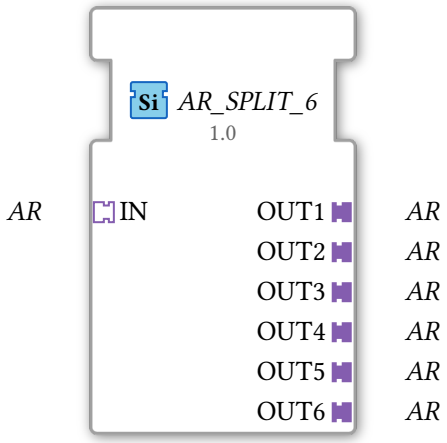
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AR_SPLIT_5_UNGATED is a simple yet useful function block for distributing a unidirectional AR adapter across five independent outputs. Its generic and eventless structure makes it a fundamental tool for adapter-based communication in IEC 61499 systems.

AR_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_SPLIT_6** is used to distribute a single AR adapter input to six separate AR adapter outputs. It is designed as a generic function block (Generic FB) and allows for the simple duplication of an AR interface without data loss or protocol-specific conversion.

Interface Structure

Event Inputs

No event inputs are available. Functionality is controlled solely via the adapter interface.

Event Outputs

No event outputs are available.

Data Inputs

No data inputs are available.

Data Outputs

No data outputs are available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AR`	**IN**	Socket (Input)	An incoming AR adapter that is distributed to all outputs.
`adapter::types::unidirectional::AR`	**OUT1**	Plug (Output)	First outgoing AR adapter (identical to IN).
`adapter::types::unidirectional::AR`	**OUT2**	Plug (Output)	Second outgoing AR adapter (identical to IN).
`adapter::types::unidirectional::AR`	**OUT3**	Plug (Output)	Third outgoing AR adapter (identical to IN).
`adapter::types::unidirectional::AR`	**OUT4**	Plug (Output)	Fourth outgoing AR adapter (identical to IN).
`adapter::types::unidirectional::AR`	**OUT5**	Plug (Output)	Fifth outgoing AR adapter (identical to IN).
`adapter::types::unidirectional::AR`	**OUT6**	Plug (Output)	Sixth outgoing AR adapter (identical to IN).

Functionality

The function block forwards the signals from the incoming AR adapter (socket **IN**) unchanged to all six output adapters (**OUT1** … **OUT6**). No logical or timing processing takes place – any change at the input is immediately visible on all outputs. The function block operates purely passively and does not require event control.

Technical Features

- **Generic Function Block (FB):** The FB is declared as a generic type (`eclipse4diac::core::GenericClassName = 'GEN_AR_SPLIT'`), allowing it to be reused in different contexts without having to redefine the underlying AR type.
- **Unidirectional Adapters:** All adapters used are of type `adapter::types::unidirectional::AR`, meaning data flows only from the socket to the plugs.
- **No State Machine:** Since the FB has no events or internal logic, there is no ECC (Execution Control Chart).

State Overview

This FB does not contain a state machine. It operates in a data-driven manner and establishes the input/output relationship continuously and without delay.

Application Scenarios

- **Distributing an AR signal to multiple devices:** If a sensor or actuator provides its data via an AR adapter, this function block (FB) can supply multiple downstream modules in parallel (e.g., several controllers or displays).
- **Test and simulation environments:** This FB is suitable for switching a single test signal to multiple evaluation units simultaneously.
- **Redundancy or parallelization:** In controllers that require multiple identical processing chains, the input is simply duplicated.

Comparison with similar function blocks

- **AR_SPLIT_2 / AR_SPLIT_4:** These function blocks split the input into two or four outputs, respectively, and differ only in the number of outputs. AR_SPLIT_6 extends this to six outputs.
- **General split function blocks:** Other split function blocks exist for data types (e.g., INT_SPLIT), but these are designed for specific data formats. This function block operates exclusively at the adapter level, making it more flexible when data exchange occurs via adapters.
- **Custom Implementation:** Alternatively, the distribution could also be achieved by manually wiring multiple adapter nodes, but this reduces clarity and maintainability.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AR_SPLIT_6** is a simple yet useful generic function block for multiplying an AR adapter signal into six parallel outputs. It is characterized by minimal complexity, a clear structure, and high reusability in automation projects based on the 4diac IDE and the IEC 61499 standard.

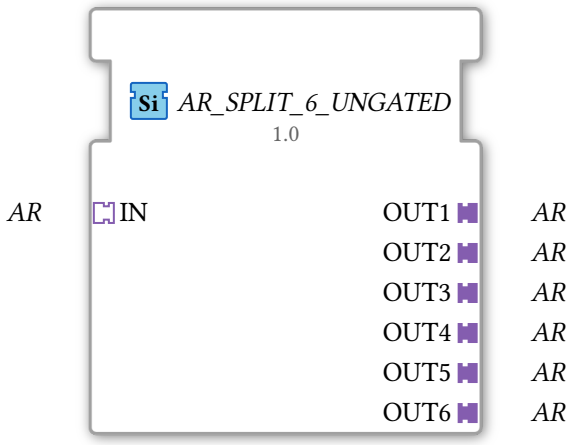
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AR_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_SPLIT_6_UNGATED** is used to distribute a single AR adapter input to six separate AR adapter outputs. It is designed as a generic function block (Generic FB) and allows for the simple duplication of an AR interface without data loss or protocol-specific conversion.

Interface Structure

Event Inputs

No event inputs are available. Functionality is controlled solely via the adapter interface.

Event Outputs

No event outputs are available.

Data Inputs

No data inputs are available.

Data Outputs

No data outputs are available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AR`	**IN**	Socket (Input)	An incoming AR adapter that is distributed to all outputs.
`adapter::types::unidirectional::AR`	**OUT1**	Plug (Output)	First outgoing AR adapter (identical to IN).
`adapter::types::unidirectional::AR`	**OUT2**	Plug (Output)	Second outgoing AR adapter (identical to IN).
`adapter::types::unidirectional::AR`	**OUT3**	Plug (Output)	Third outgoing AR adapter (identical to IN).
`adapter::types::unidirectional::AR`	**OUT4**	Plug (Output)	Fourth outgoing AR adapter (identical to IN).
`adapter::types::unidirectional::AR`	**OUT5**	Plug (Output)	Fifth outgoing AR adapter (identical to IN).
`adapter::types::unidirectional::AR`	**OUT6**	Plug (Output)	Sixth outgoing AR adapter (identical to IN).

Functionality

The function block forwards the signals from the incoming AR adapter (socket **IN**) unchanged to all six output adapters (**OUT1** ... **OUT6**). No logical or timing processing takes place – any change at the input is immediately visible on all outputs. The function block operates purely passively and does not require event control.

Technical Features

- **Generic Function Block (FB):** The FB is declared as a generic type (`eclipse4diac::core::GenericClassName = 'GEN_AR_SPLIT'`), allowing it to be reused in different contexts without having to redefine the underlying AR type.
- **Unidirectional Adapters:** All adapters used are of type `adapter::types::unidirectional::AR`, meaning data flows only from the socket to the plugs.
- **No State Machine:** Since the FB has no events or internal logic, there is no ECC (Execution Control Chart).

State Overview

This FB does not contain a state machine. It operates in a data-driven manner and establishes the input/output relationship continuously and without delay.

Application Scenarios

- **Distributing an AR signal to multiple devices:** If a sensor or actuator provides its data via an AR adapter, this function block (FB) can supply multiple downstream modules in parallel (e.g., several controllers or displays).
- **Test and simulation environments:** This FB is suitable for switching a single test signal to multiple evaluation units simultaneously.
- **Redundancy or parallelization:** In controllers that require multiple identical processing chains, the input is simply duplicated.

Comparison with similar function blocks

- **AR_SPLIT_2 / AR_SPLIT_4:** These function blocks split the input into two or four outputs, respectively, and differ only in the number of outputs. AR_SPLIT_6_UNGATED extends this to six outputs.
- **General split function blocks:** Other split function blocks exist for data types (e.g., INT_SPLIT), but these are designed for specific data formats. This function block operates exclusively at the adapter level, making it more flexible when data exchange occurs via adapters.
- **Custom Implementation:** Alternatively, the distribution could also be achieved by manually wiring multiple adapter nodes, but this reduces clarity and maintainability.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

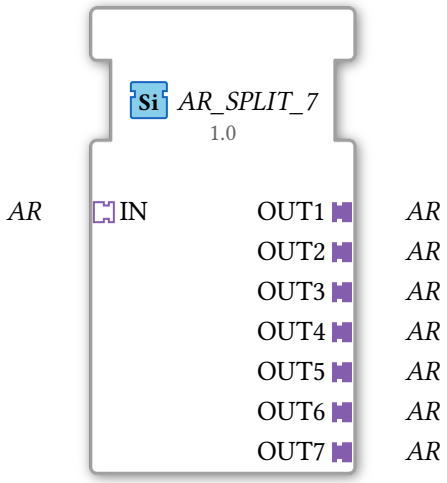
The **AR_SPLIT_6_UNGATED** is a simple yet useful generic function block for multiplying an AR adapter signal into six parallel outputs. It is characterized by minimal complexity, a clear structure, and high reusability in automation projects based on the 4diac IDE and the IEC 61499 standard.

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AR_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The **AR_SPLIT_7** is a generic function block that distributes an incoming AR adapter socket (of type `adapter::types::unidirectional::AR`) to seven separate AR adapter plugs. It serves to forward an AR signal to up to seven different receivers without requiring the data to be provided multiple times.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AR`	IN	Socket	Incoming AR adapter, which is distributed to the seven outputs.
`adapter::types::unidirectional::AR`	OUT1 ... OUT7	Plug	Seven output adapters to which the incoming AR adapter is passed on unchanged.

Functionality

The function block receives an AR adapter via socket `IN`. As soon as a connection to this socket is established, the function block forwards the complete adapter – including all data and events it contains – to all seven plug adapters (`OUT1` to `OUT7`). No data transformation, filtering, or buffering takes place; the AR adapter is mirrored **1:1 across all outputs**.

The function block (FB) is implemented as a **generic block** (see attribute `eclipse4diac::core::GenericClassName = 'GEN_AR_SPLIT'`), so it operates independently of the specific AR adapter parameters.

Technical Features

- **Generic Design** – The FB does not require specific type information from the AR adapter; it operates purely via the adapter interface.
- **No State Machine** – The FB does not have an ECC (Execution Control Chart) and is purely data flow-related; it reacts directly to adapter connections.
- **Copyright & License** – The function block is subject to the Eclipse Public License 2.0 (EPL-2.0) and was provided by HR Agrartechnik GmbH (Version 1.0, 2025-01-24).

State Overview

The function block (FB) does not have a state machine (ECC). Distribution is static and without runtime logic.

Application Scenarios

- **Signal Distribution in Automation Systems** – One sensor or control adapter (AR) must supply several downstream function blocks in parallel.
- **Monitor or Diagnostic Connection** – An AR signal is simultaneously passed on to a control branch and a monitoring branch.
- **Test and Simulation Environments** – One adapter is distributed to several test or mock function blocks without instantiating the source multiple times.

Comparison with Similar Function Blocks

- **AR_SPLIT_2, AR_SPLIT_3, ...** – These function blocks offer the same functionality with a different number of outputs (2, 3, ...). The `AR_SPLIT_7` is the variant with exactly seven outputs.
- **AR_MERGE** – Merging multiple AR adapters into one, i.e., the inverse operation.
- **AR_COPY** – Often used as a dedicated function block for a single 1:1 distribution, while `AR_SPLIT_7` handles multiple outputs at once.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AR_SPLIT_7** is a lean, generic function block for easily distributing one AR adapter to up to seven target adapters. Thanks to its generic nature, it can be used immediately without modifying the type information and is particularly suitable for loosely coupled, dataflow-oriented architectures in automation technology.

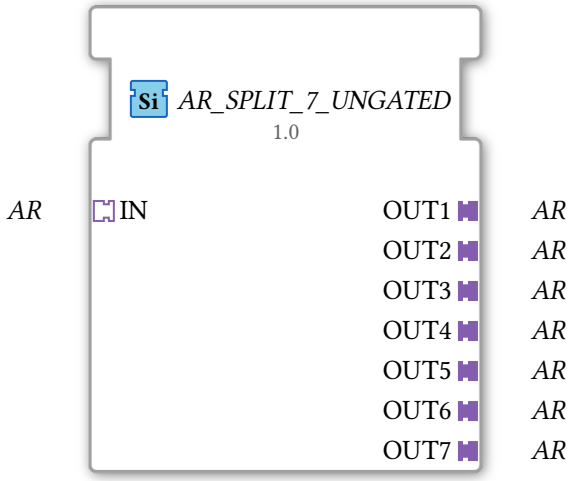
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AR_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **AR_SPLIT_7_UNGATED** is a generic function block that distributes an incoming AR adapter socket (of type `adapter::types::unidirectional::AR`) to seven separate AR adapter plugs. It serves to forward an AR signal to up to seven different receivers without requiring the data to be provided multiple times.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AR`	IN	Socket	Incoming AR adapter, which is distributed to the seven outputs.
`adapter::types::unidirectional::AR`	OUT1 ... OUT7	Plug	Seven output adapters to which the incoming AR adapter is passed on unchanged.

Functionality

The function block receives an AR adapter via socket `IN`. As soon as a connection to this socket is established, the function block forwards the complete adapter – including all data and events it contains – to all seven plug adapters (`OUT1` to `OUT7`). No data transformation, filtering, or buffering takes place; the AR adapter is mirrored **1:1 across all outputs**.

The function block (FB) is implemented as a **generic block** (see attribute `eclipse4diac::core::GenericClassName = 'GEN_AR_SPLIT'`), so it operates independently of the specific AR adapter parameters.

Technical Features

- **Generic Design** – The FB does not require specific type information from the AR adapter; it operates purely via the adapter interface.
- **No State Machine** – The FB does not have an ECC (Execution Control Chart) and is purely data flow-related; it reacts directly to adapter connections.
- **Copyright & License** – The function block is subject to the Eclipse Public License 2.0 (EPL-2.0) and was provided by HR Agrartechnik GmbH (Version 1.0, 2025-01-24).

State Overview

The function block (FB) does not have a state machine (ECC). Distribution is static and without runtime logic.

Application Scenarios

- **Signal Distribution in Automation Systems** – One sensor or control adapter (AR) must supply several downstream function blocks in parallel.
- **Monitor or Diagnostic Connection** – An AR signal is simultaneously passed on to a control branch and a monitoring branch.
- **Test and Simulation Environments** – One adapter is distributed to several test or mock function blocks without instantiating the source multiple times.

Comparison with Similar Function Blocks

- **AR_SPLIT_2, AR_SPLIT_3, ...** – These function blocks offer the same functionality with a different number of outputs (2, 3, ...). The `AR_SPLIT_7_UNGATED` is the variant with exactly seven outputs.
- **AR_MERGE** – Merging multiple AR adapters into one, i.e., the inverse operation.
- **AR_COPY** – Often used as a dedicated function block for a single 1:1 distribution, while `AR_SPLIT_7_UNGATED` handles multiple outputs at once.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

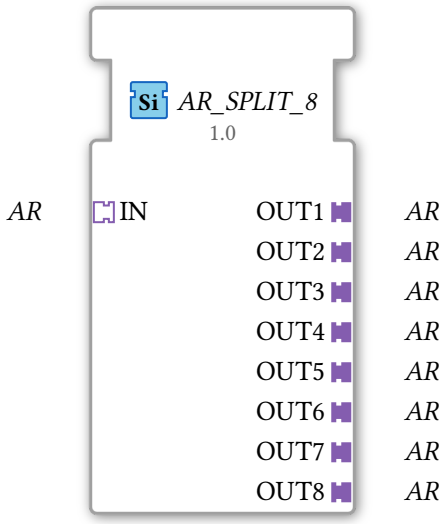
The **AR_SPLIT_7_UNGATED** is a lean, generic function block for easily distributing one AR adapter to up to seven target adapters. Thanks to its generic nature, it can be used immediately without modifying the type information and is particularly suitable for loosely coupled, dataflow-oriented architectures in automation technology.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AR_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The AR_SPLIT_8 function block is used to split a single unidirectional AR adapter signal into eight identical AR outputs. It is implemented as a generic function block and enables simple signal distribution without additional logic.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Name	Type	Direction
IN	`adapter::types::unidirectional::AR`	Socket (Input)
OUT1 – OUT8	`adapter::types::unidirectional::AR`	Plug (Output)

Functionality

The function block receives the complete interface of a unidirectional AR adapter via the **IN** socket. Events and data present at this interface are replicated to all eight output adapters (**OUT1** to **OUT8**) without delay or modification. Thus, a single data/event stream can be distributed in parallel to up to eight downstream components.

Technical Features

- The function block is generic (generic FB) and is identified as such by the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AR_SPLIT'`.
- There is no internal state diagram (ECC) – the forwarding is direct and stateless at all times.

- The implementation is designed for unidirectional AR adapters; use with bidirectional adapters is not supported.

State Overview

The function block has no explicit state diagram and operates statelessly. Adapter distribution occurs continuously, without any time dependencies or internal logic.

Application Scenarios

- **Distribution of Control Signals**: A central control algorithm (e.g., a PID controller) distributes its output via AR to several parallel actuators.
- **Broadcast in Star-Shaped Plant Sections**: Signals from a higher-level controller are distributed to eight identical substations.
- **Test and Simulation Environments**: A signal generator feeds several test objects simultaneously with the same adapter signal.

Comparison with Similar Function Blocks

- **AR_SPLIT_2, AR_SPLIT_4**: Analog function blocks with 2 and 4 outputs, respectively; AR_SPLIT_8 extends the number to eight outputs.
- **AR_MERGE_X**: Combines multiple AR signals into one – opposite function.
- SPLIT_ *for other adapter types**: Split blocks exist for other unidirectional and bidirectional adapter definitions that implement a similar splitting logic.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AR_SPLIT_8 is a simple yet extremely useful block for distributing unidirectional AR adapter signals across up to eight parallel paths. Its generic implementation allows for flexible use in various automation projects without requiring the implementation of additional signal multiplication logic.

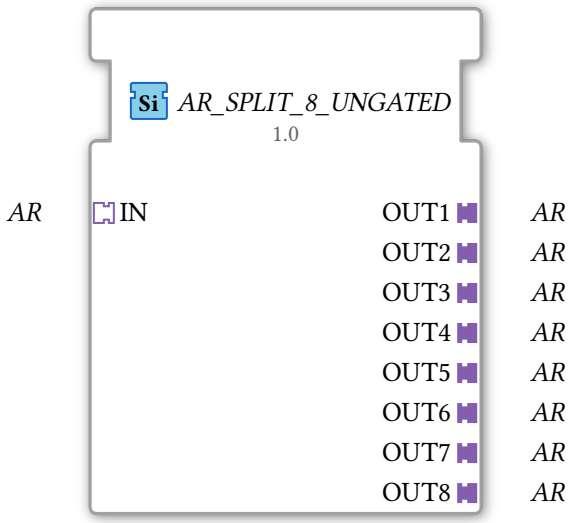
🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AR_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The AR_SPLIT_8_UNGATED function block is used to split a single unidirectional AR adapter signal into eight identical AR outputs. It is implemented as a generic function block and enables simple signal distribution without additional logic.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Name	Type	Direction
IN	`adapter::types::unidirectional::AR`	Socket (Input)
OUT1 – OUT8	`adapter::types::unidirectional::AR`	Plug (Output)

Functionality

The function block receives the complete interface of a unidirectional AR adapter via the **IN** socket. Events and data present at this interface are replicated to all eight output adapters (**OUT1** to **OUT8**) without delay or modification. Thus, a single data/event stream can be distributed in parallel to up to eight downstream components.

Technical Features

- The function block is generic (generic FB) and is identified as such by the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AR_SPLIT'`.
- There is no internal state diagram (ECC) – the forwarding is direct and stateless at all times.
- The implementation is designed for unidirectional AR adapters; use with bidirectional adapters is not supported.

State Overview

The function block has no explicit state diagram and operates statelessly. Adapter distribution occurs continuously, without any time dependencies or internal logic.

Application Scenarios

- **Distribution of Control Signals**: A central control algorithm (e.g., a PID controller) distributes its output via AR to several parallel actuators.
- **Broadcast in Star-Shaped Plant Sections**: Signals from a higher-level controller are distributed to eight identical substations.
- **Test and Simulation Environments**: A signal generator feeds several test objects simultaneously with the same adapter signal.

Comparison with Similar Function Blocks

- **AR_SPLIT_2, AR_SPLIT_4**: Analog function blocks with 2 and 4 outputs, respectively; AR_SPLIT_8_UNGATED extends the number to eight outputs.
- **AR_MERGE_X**: Combines multiple AR signals into one – opposite function.
- SPLIT_ *for other adapter types**: Split blocks exist for other unidirectional and bidirectional adapter definitions that implement a similar splitting logic.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

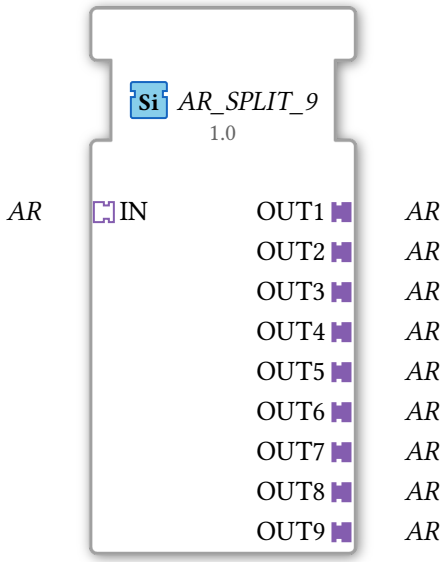
The AR_SPLIT_8_UNGATED is a simple yet extremely useful block for distributing unidirectional AR adapter signals across up to eight parallel paths. Its generic implementation allows for flexible use in various automation projects without requiring the implementation of additional signal multiplication logic.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AR_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_SPLIT_9** splits an incoming AR adapter (type `unidirectional`) into nine separate AR outputs. It serves as a distributor for an AR signal to multiple downstream components. The block is designed to be completely generic.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Adapter	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AR`	Incoming AR signal that is distributed.
Plug (Output)	`OUT1` ... `OUT9`	`adapter::types::unidirectional::AR`	Nine identical outputs that forward the incoming AR signal.

Functionality

The module forwards the AR signal present at the **IN** socket unchanged to all nine outputs **OUT1** to **OUT9**. No delay, filtering, or conversion takes place. Signal propagation is purely combinatorial.

Technical Features

- **Generic Function Block**: The function block uses a generic class name attribute (`eclipse4diac::core::GenericClassName` with the value `'GEN_AR_SPLIT'`) and a type hash attribute to enable flexible type adaptation in the IEC 61499 model.
- **Unidirectional Adapters**: Both the input and all outputs are defined as unidirectional AR adapters. Feedback or bidirectional communication is not provided.
- **No Events or Data**: All logic is handled exclusively via adapter interfaces – there are no event or data inputs/outputs.

State Overview

The function block has no internal state (no state machine). The output signal corresponds to the current input signal at any given time. There are no initialization or error states.

Application Scenarios

- **Distribution of a Start/Stop Signal**: A central AR command (e.g., "Start") is distributed to multiple actuators or machine parts.
- **Multicast Coupling**: An AR signal originating from a higher-level controller is simultaneously sent to multiple subordinate modules.
- **Parallel Control** of nine identical or different components using the same AR protocol.

Comparison with Similar Function Blocks

Function Block	Number of Outputs	Remarks
`AR_SPLIT_4`	4	Same functionality, but only four outputs.
`AR_SPLIT_9`	9	Presented function block (nine outputs).

AR_SPLIT_2	2	Minimal splitter for two paths.

The function blocks differ only in the number of outputs; the functionality and generic properties are identical.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AR_SPLIT_9** is a simple yet useful splitter function block for unidirectional AR signals. Due to its generic design, it can be flexibly used in different control architectures without requiring manual type adaptation. It is particularly suitable for scenarios where a signal needs to be duplicated to multiple paths simultaneously.

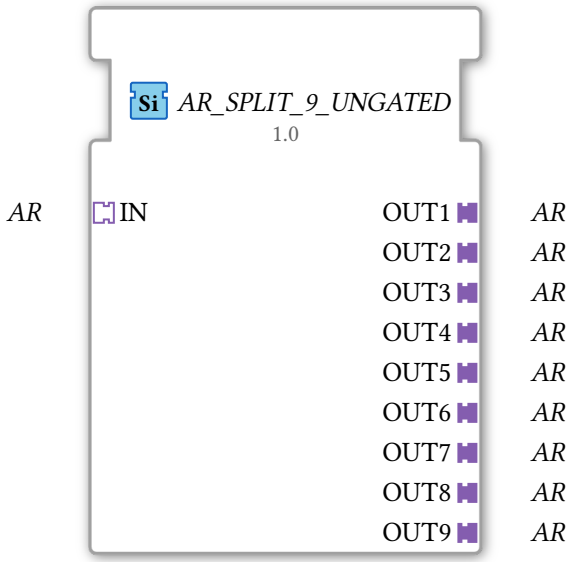
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AR_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_SPLIT_9_UNGATED** splits an incoming AR adapter (type `unidirectional`) into nine separate AR outputs. It serves as a distributor for an AR signal to multiple downstream components. The block is designed to be completely generic.

Interface Structure

Event Inputs
No event inputs available.

Event Outputs
No event outputs available.

Data Inputs
No data inputs available.

Data Outputs
No data outputs available.

Adapter

Direction	Adapter	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AR`	Incoming AR signal that is distributed.
Plug (Output)	`OUT1` ... `OUT9`	`adapter::types::unidirectional::AR`	Nine identical outputs that forward the incoming AR signal.

Functionality
The module forwards the AR signal present at the **IN** socket unchanged to all nine outputs **OUT1** to **OUT9**. No delay, filtering, or conversion takes place. Signal propagation is purely combinatorial.

Technical Features
- **Generic Function Block**: The function block uses a generic class name attribute (`eclipse4diac::core::GenericClassName` with the value `'GEN_AR_SPLIT'`) and a type hash attribute to enable flexible type adaptation in the IEC 61499 model.
- **Unidirectional Adapters**: Both the input and all outputs are defined as unidirectional AR adapters. Feedback or bidirectional communication is not provided.
- **No Events or Data**: All logic is handled exclusively via adapter interfaces – there are no event or data inputs/outputs.

State Overview
The function block has no internal state (no state machine). The output signal corresponds to the current input signal at any given time. There are no initialization or error states.

Application Scenarios
- **Distribution of a Start/Stop Signal**: A central AR command (e.g., “Start”) is distributed to multiple actuators or machine parts.
- **Multicast Coupling**: An AR signal originating from a higher-level controller is simultaneously sent to multiple subordinate modules.
- **Parallel Control** of nine identical or different components using the same AR protocol.

Comparison with Similar Function Blocks

Function Block	Number of Outputs	Remarks
`AR_SPLIT_4`	4	Same functionality, but only four outputs.
`AR_SPLIT_9_UNGATED`	9	Presented function block (nine outputs).
`AR_SPLIT_2`	2	Minimal splitter for two paths.

The function blocks differ only in the number of outputs; the functionality and generic properties are identical.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AR_SPLIT_9_UNGATED** is a simple yet useful splitter function block for unidirectional AR signals. Due to its generic design, it can be flexibly used in different control architectures without requiring manual type adaptation. It is particularly suitable for scenarios where a signal needs to be duplicated to multiple paths simultaneously.

–

🌐 Related topic subpages on ms-muc-docs.de

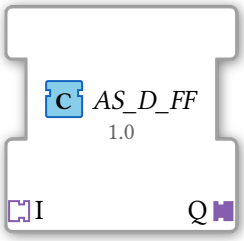
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numeric Index (unidirectional)

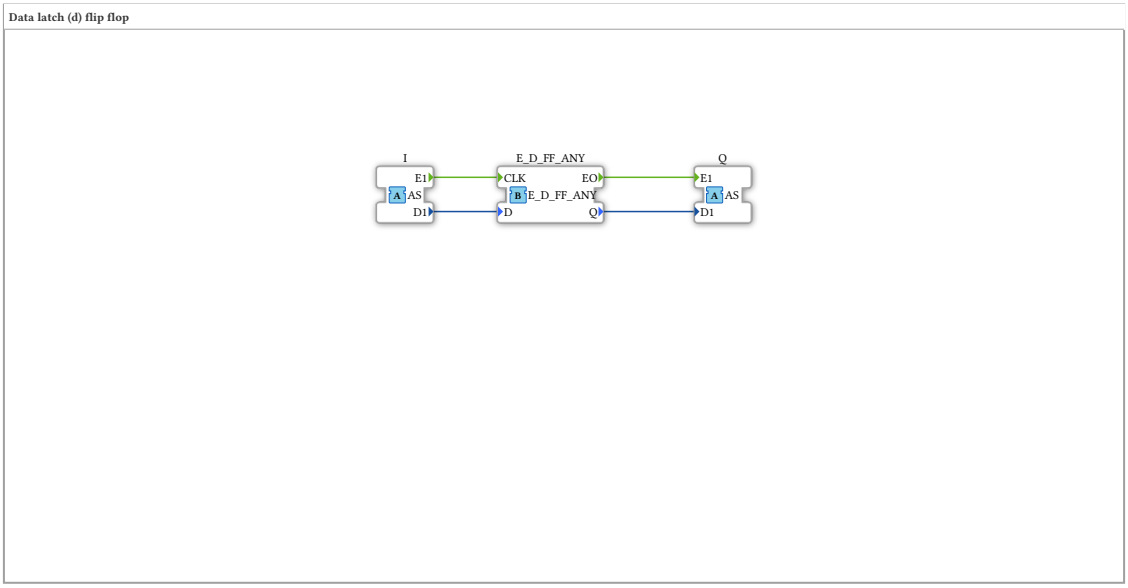
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AS_D_FF

Value to latch – *AS*



AS – Latched value



• – ○ • – ○ • – ○ •

Introduction

The function block **AS_D_FF** implements a simple data latch (D flip-flop) based on adapters. It stores an incoming data value with each incoming event and outputs the stored value as a signal. The block encapsulates the logic of the 4diac standard block `E_D_FF_ANY` and provides its interfaces exclusively via unidirectional adapters of type `adapter::types::unidirectional::AS`. This enables modular and standardized integration into more complex systems.

Interface Structure

Event Inputs

No direct event inputs. The block receives events exclusively via the adapter **I** (socket). The event is provided at the adapter input `I.E1`.

Event Outputs

No direct event outputs. The function block sends events exclusively via the **Q** (Plug) adapter. The output event appears at the adapter output `Q.E1`.

Data Inputs

No direct data inputs. The data value to be stored is provided via the **I** (Socket) adapter as `I.D1`.

Data Outputs

No direct data outputs. The stored data value is output via the **Q** (Plug) adapter as `Q.D1`.

Adapters

Adapter	Direction	Type	Description
I	Socket (Input)	`adapter::types::unidirectional::AS`	Provides the input event (`E1`) and the data value to be latched (`D1`).
Q	Plug (Output)	`adapter::types::unidirectional::AS`	Outputs an output event (`E1`) and the stored data value (`D1`).

Functionality

The **AS_D_FF** function block operates as an asynchronous data latch.

Upon each positive event on `I.E1`, the current value of `I.D1` is transferred to the block's internal memory. Immediately afterwards, an event is triggered on `Q.E1`, and the stored value is available on `Q.D1`.

This logic is implemented by the internally used component `E_D_FF_ANY`:

- `I.E1` ⟶ `E_D_FF_ANY.CLK`
- `I.D1` ⟶ `E_D_FF_ANY.D`
- `E_D_FF_ANY.EO` ⟶ `Q.E1`
- `E_D_FF_ANY.Q` ⟶ `Q.D1`

The component thus behaves like a classic D flip-flop, but with a standardized adapter interface instead of direct inputs/outputs.

…

Technical Features

- **Adapter-based encapsulation**: All data exchange occurs exclusively via unidirectional adapters of type `adapter::types::unidirectional::AS`. This facilitates reuse in different contexts and ensures a clear separation between event and data flows.
- **Internal use of `E_D_FF_ANY`**: The component utilizes the standard function block `iec61499::events::E_D_FF_ANY`, which offers a proven and efficient implementation of a D flip-flop.
- **No direct I/O pins**: The component itself does not define any event or data inputs/outputs; all communication is handled via the adapters.
- **Unidirectional adapters**: Both adapters (I and Q) are defined as unidirectional, meaning that event and data flow occurs only in one direction (from the socket to the plug).

State Overview

The component does not have an explicit state machine. Its behavior is determined by the internal D flip-flop, which has two logical states:

- **State 0**: The stored data value is 0 (or `FALSE`).
- **State 1**: The stored data value is 1 (or `TRUE`).

The state is switched to `I.E1` by an event (if the data value has changed) and remains in this state until the next event.

Application Scenarios

- **Signal Buffering**: Temporarily storing a digital signal that is to be retrieved at specific times (e.g., sample-and-hold).
- **Synchronization**: Adapting asynchronous data streams to a clock-driven sequence by holding a value at an event.
- **Modular Interconnection**: Used in systems based on adapter communication to maintain a uniform interface (e.g., in a chain of processing blocks).
- **Simple State Storage**: Implements simple memory functions in automation applications without relying on complex memory modules.

Comparison with Similar Modules

Module	Property	Difference from AS_D_FF
SR_FF	Set-Reset Flip-Flop	Has separate set and reset inputs, no clock signal.
E_D_FF_ANY	Pure D flip-flop with direct I/O	Does not offer an adapter interface; requires direct connections.
AS_FF_RS (hypothetical)	RS flip-flop with adapters	Uses two adapters for set and reset, no clock signal.

The **AS_D_FF** combines the clear logic of a D flip-flop with the flexibility of an adapter-based interface. It is particularly suitable for systems that prefer standardized, unidirectional communication.

Conclusion

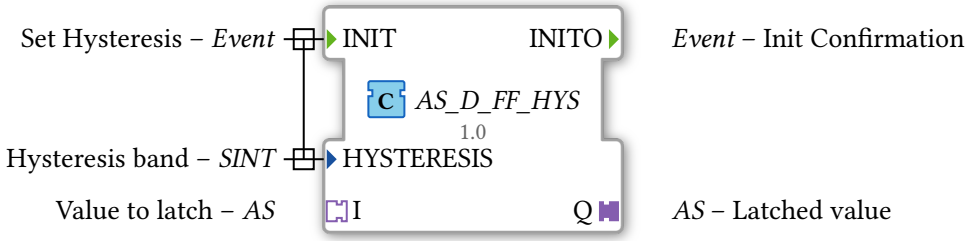
The **AS_D_FF** function block represents a practical encapsulation of a D flip-flop that communicates exclusively via adapters. Its internal behavior corresponds to a reliable data latch that responds to every incoming event. The use of adapters increases modularity and reusability, while the implementation is based on the proven `E_D_FF_ANY`. The block is ideal for all applications that require simple, event-driven data storage with a standardized interface.

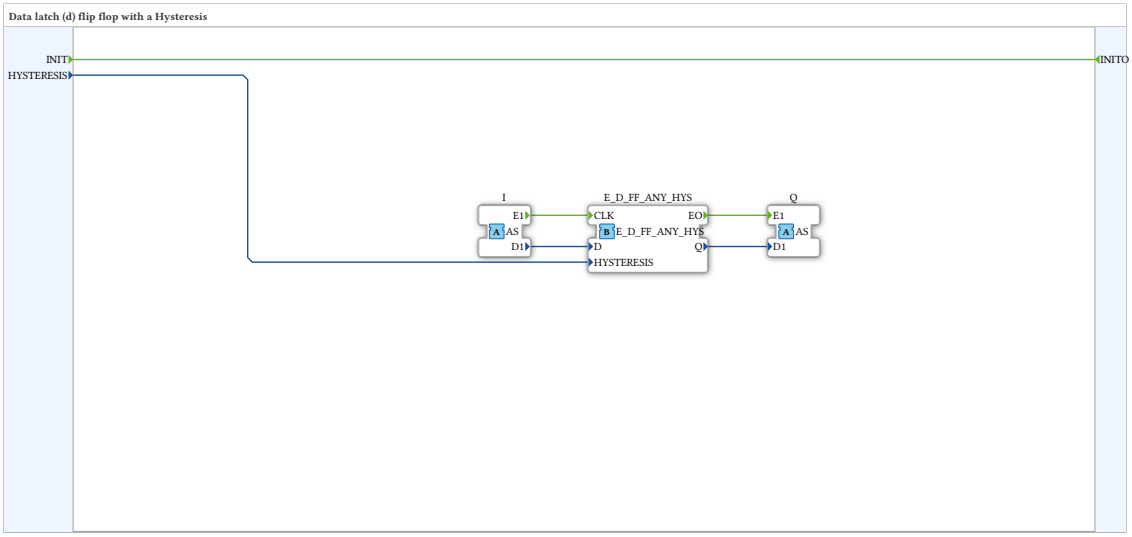
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AS_D_FF_HYS





• – ○ • – ○ • – ○ •

Introduction

The function block **AS_D_FF_HYS** implements a data-triggered flip-flop (D-latch) with **hysteresis** (switching hysteresis). This component is typically used in signal processing to smooth noisy or fluctuating input signals and create defined switching points. It features an event input `INIT` to set the hysteresis value, as well as two adapter interfaces: an input adapter `I` for the signal to be latched and an output adapter `Q` for the latched and hysteresis-filtered output signal.

Interface Structure

Event Inputs

Name	Type	Comment	With Data
`INIT`	EInit	Sets the hysteresis value	`HYSTERESIS`

Event Outputs

Name	Type	Comment
`INITO`	EInit	Confirmation of INIT processing

Data Inputs

Name	Type	Comment
`HYSTERESIS`	SINT	Hysteresis Bandwidth (Integer Value)

Data Outputs

No direct data outputs – the output value is provided via the `Q` adapter.

Adapters

Direction	Name	Type	Comment
Socket	`I`	`adapter::types::unidirectional::AS`	Input Signal (Value and Clock)
Plug	`Q`	`adapter::types::unidirectional::AS`	Output signal (latched with hysteresis)

The `AS` (unidirectional) adapters typically have an event port `E1` and a data port `D1`. The socket `I` provides the clock signal via `I.E1` and the data value to be latched via `I.D1`. The plug `Q` outputs an event via `Q.E1` and the filtered output value via `Q.D1`.

The socket `I` provides the clock signal via `I.E1` and the data value to be latched via `I.D1`.

Functionality

The FB receives a clock signal (event `I.E1`) and a data value (`I.D1`) via the adapter `I`. With each clock cycle, the current data value is compared to the last output value, taking the hysteresis band into account. The output `Q.D1` only changes if the new input value is outside the interval `[letzter Ausgangswert - HYSTERESIS , letzter Ausgangswert + HYSTERESIS]`. This suppresses small fluctuations (noise) around the set operating point.

The event `INIT` is used for the one-time configuration of the hysteresis value. The hysteresis value from the data input `HYSTERESIS` is used for this purpose. After successful initialization, the event `INITO` is triggered.

The function block (FB) internally uses the block `E_D_FF_ANY_HYS` from the library `logiBUS::signalprocessing::hysteresis`, which implements the actual flip-flop logic with hysteresis.

Technical Features

- The hysteresis value is defined as `SINT` (signed integer).
- The actual signal processing is implemented by a subordinate FB (`E_D_FF_ANY_HYS`), which is not directly accessible to the user.
- The adapter interface allows loose coupling of the function block to other components.
- The event `INIT` does not process a value – the hysteresis parameters can only be set before initial operation. Subsequent changes are only possible by resending `INIT`.

State Overview

The function block (FB) has **no explicit state machine** that is externally visible. Internally, the current output value is stored. Hysteresis creates a state memory: The output does not change with every input change, but only when the hysteresis thresholds are exceeded or fallen below.

Application Scenarios

- **Debouncing of Switch Contacts** – Suppression of bounce and short interference pulses.
- **Threshold Switches with Hysteresis** – e.g., temperature controllers that separate on and off points.
- **Signal Smoothing in Sensors** – Processing noisy analog values into discrete states.
- **Data Latch with Noise Suppression** – for slow but noisy data signals.

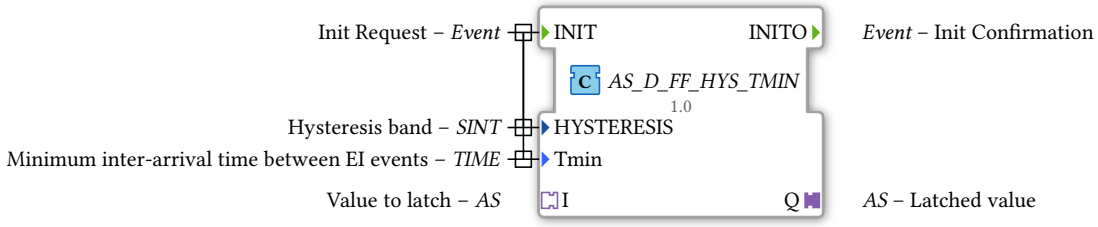
Comparison with Similar Function Blocks

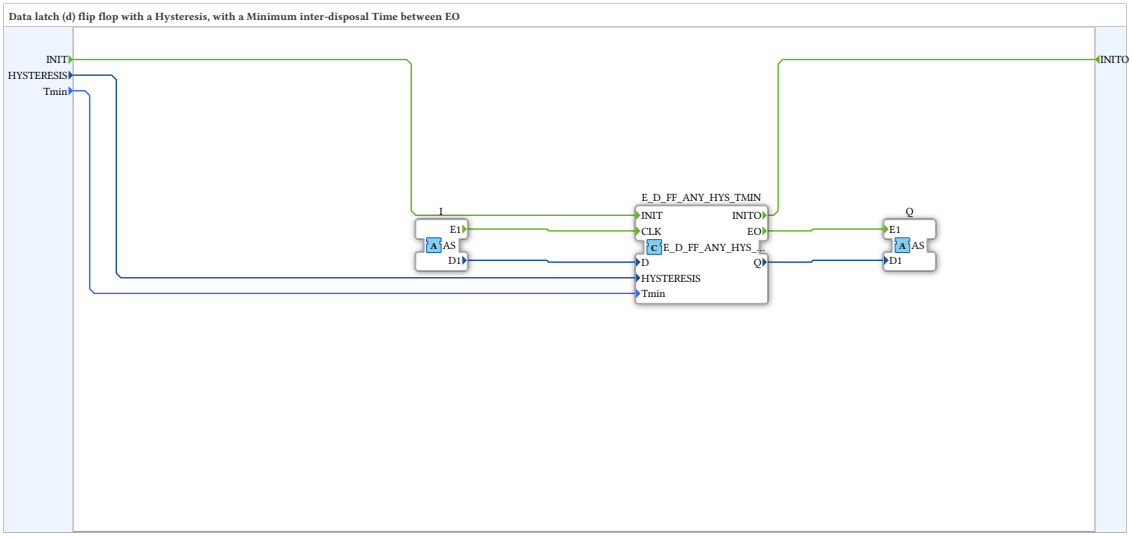
Function Block	Property	Difference to AS_D_FF_HYS
`E_D_FF`	Classic D flip-flop without hysteresis	No filtering; output follows each clock cycle.
`E_D_FF_HYS`	D flip-flop with hysteresis (same logic)	No dedicated `INIT` input; hysteresis is set via data if necessary.
`AS_FF_HYS`	Flip-flop with hysteresis and dedicated adapter interface	Possibly different coupling, similar function.

Conclusion

The `AS_D_FF_HYS` combines a D flip-flop with adjustable hysteresis and encapsulates the complex logic in an easy-to-use, adapter-based component. It is particularly suitable for applications where noisy discrete signals need to be processed cleanly and with defined switching points. Initial configuration via `INIT` allows for flexible adaptation to various noise environments.

AS_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AS_D_FF_HYS_TMIN** implements a data-triggered flip-flop (D flip-flop) with hysteresis and a minimal time delay between successive events. It serves for stable, debounced signal transmission in time-critical control sequences. The block uses adapters for unidirectional communication, with an input signal received via socket **I** and the latching output signal provided via plug **Q**.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization request; sets hysteresis and minimum time

Event Outputs

Name	Type	Comment
INITO	EInit	Confirmation of completed initialization

Data Inputs

Name	Type	Comment
HYSTERESIS	SINT	Hysteresis band (integer value, e.g., 8 bits)
Tmin	TIME	Minimum time between two events (e.g., for debouncing)

Data Outputs

No direct data outputs – the output signal is provided via the **Q** adapter.

Adapters

Type	Name	Direction	Comment
AS	I (Socket)	Input	Unidirectional adapter for the signal to be latched (e.g., data + event line)
AS	Q (Plug)	Output	Unidirectional adapter for the latching output signal

Functionality

The function block operates internally with a predefined function block **E_D_FF_ANY_HYS_TMIN**. Upon the **INIT** event, the parameters **HYSTERESIS** and **Tmin** are transferred. The incoming event (via the **I** adapter, line **E1**) triggers the flip-flop clock, while the corresponding data value (via **I.D1**) is transferred. Output **Q.D1** is updated only if:

- the new data value differs from the currently stored value by at least the **hysteresis**, and
- at least the time **Tmin** has elapsed since the last event.

The output event (via **Q.E1**) signals a valid update. The module thus suppresses noise and prevents excessively rapid edge transitions.

Technical Features

- **Adapter-Based Input/Output** – Signal transmission via adapters **I** and **Q** enables easy integration into complex adapter networks and increases reusability.
- **Hysteresis** – Prevents oscillations with fluctuating or noisy input signals by only registering significant changes.
- **Minimum Event Time (Tmin)** – Enforces a minimum pause between two triggers; serves as an additional debouncing mechanism.
- **Internal use of a proven flip-flop component** – The implementation delegates the core logic to **E_D_FF_ANY_HYS_TMIN**, which is designed for generic data types.

State Overview

The component does not have any explicit, externally visible states. Internally, the flip-flop maintains a stored value (last valid output) that is only updated when the hysteresis and timing conditions are met. After initialization (INIT/INITO), the stored value is undefined until the first valid signal arrives.

Application Scenarios

- **Debouncing of sensor signals** – e.g., limit switches or proximity sensors with mechanical bouncing.
- **Frequency limiting** – Preventing excessively fast switching operations in downstream actuators or communication interfaces.
- **Hysteresis-based threshold detection** – e.g., temperature or pressure control with a reset band.
- **Signal transmission in time-critical, interference-prone environments** (e.g., agricultural machinery controls).

Comparison with similar components

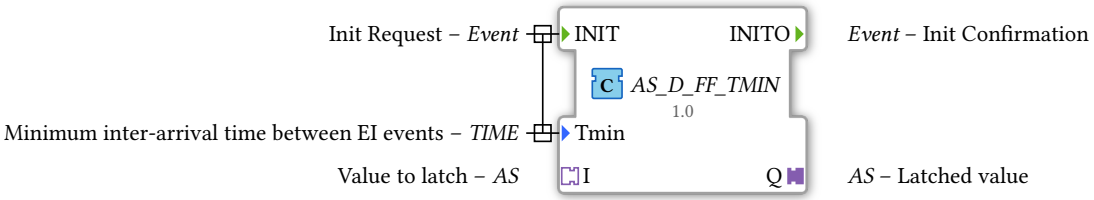
- **Simple D flip-flop (e.g., E_D_FF)** – without hysteresis or time delay; accepts every clock signal without filtering.
- **Schmitt trigger component** – also with hysteresis, but usually without asynchronous event control and without an adjustable minimum time.
- **Delay component (e.g., E_DELAY)** – only delays the signal, but has no hysteresis or latch function.

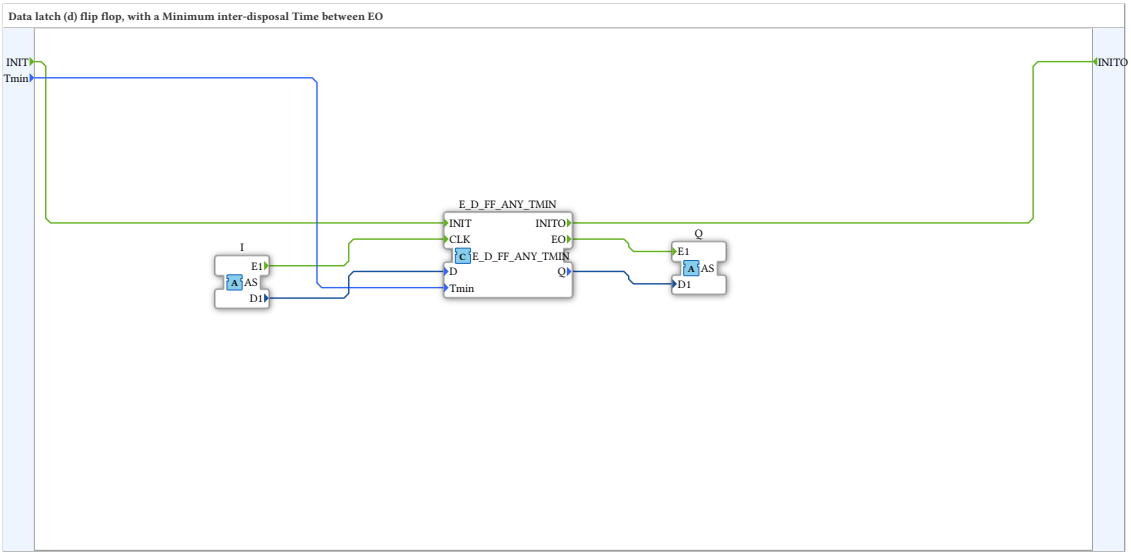
AS_D_FF_HYS_TMIN combines latch function, hysteresis, and timing in a single component and is therefore specifically optimized for robust, time-controlled signal processing.

Conclusion

The **AS_D_FF_HYS_TMIN** is a specialized function block for industrial controllers that require reliable, noise-insensitive signal transmission with adjustable dead time. The combination of a D flip-flop, hysteresis, and minimum time makes it a valuable tool for automation solutions, especially in environments with high levels of interference or tight timing requirements.

AS_D_FF_TMIN





• – ○ • – ○ • – ○ •

The function block **AS_D_FF_TMIN** implements a data-locking D flip-flop that takes over a data value supplied via an adapter upon the arrival of an event and outputs it via another adapter. Its special feature is a configurable minimum time (`Tmin`) between two consecutive take-over events. This suppresses excessively rapid event sequences, which can be used, for example, for debouncing or to enforce process constraints.

Event	Type	Comment
INIT	EInit	Initialization Request (reads `Tmin`)
Event	Type	Comment
–––––	–––	–––––––––––––––
INITO	EInit	Initialization Confirmation
Name	Type	Comment
––	––	–––––––––––––––––––––––
Tmin	TIME	Minimum time between two consecutive CLK events

This block has no explicit data outputs; output is exclusively via the **Q** adapter.

Daten-Ausgänge

Daten-Eingänge

Ereignis-Ausgänge

Ereignis-Eingänge

Interface Structure

Introduction

Adapters

| Direction | Adapter | Type | Comment | | Socket | I | adapter::types::unidirectional::AS | Input data (D1) and clock (E1) | | Plug | Q | adapter::types::unidirectional::AS | Output data (D1) and event (E1) |

The adapters of type `AS` each provide one event input/output (E1) and one data input/output (D1).

Functionality

After successful initialization (INIT with a valid `Tmin`), the block operates like a clocked D flip-flop with a time filter:

- An event at the adapter input **I.E1** is interpreted as a clock signal (CLK).
- The data value **I.D1** is adopted upon the arrival of I.E1, **provided the time interval to the last CLK event is at least `Tmin`**. Otherwise, the event is ignored.
- If the adoption is successful, the stored value is output at data output **Q.D1**, and an event **Q.E1** is triggered.

Time filtering is implemented by the internally used block `E_D_FF_ANY_TMIN`, which delays or discards the clock event if the minimum time is not met.

- **Adapter-based I/O**: Data and events are not exchanged as separate variables, but via unidirectional adapters (`AS`). This allows for modular connection with other blocks.
- **Parameterizable Minimum Time**: The time `Tmin` is set for initialization and cannot be changed during operation (another INIT call is required).
- **Event Suppression**: Clock events that occur too close together are silently ignored – no error message is generated.

The block does not have explicit state machines in the sense of a statechart. However, its behavior can be described in two phases:

1. **Initialization Phase**: After the INIT event, `Tmin` is set internally. Only then is the block ready to process CLK events.
2. **Operation Phase**: The block waits for CLK events. When one arrives, it checks whether the time lock (since the last CLK) has expired. If so, the data value is taken and output. If no, the event is discarded.

- **Digital Signal Debouncing**: A sensor delivers fast, bouncing signals; the minimum pulse width is set using `Tmin`.
- **Clock Synchronization in Time-Critical Systems**: Prevents data transfers faster than a predefined system clock threshold.
- **Rate Limiting for Communication Interfaces**: A new value is only accepted every `Tmin` seconds (e.g., for bus accesses).

Function Block	Time Filtering	Additional Features
AS_D_FF	No (immediate transfer)	Simple D flip-flop without time limit
AS_D_FF_TMIN (this one)	Yes, configurable via `Tmin`	Suppresses excessively fast clock sequences
AS_D_FF_TMAX	Yes, maximum time between clock cycles	Enforces regular takeovers

The **AS_D_FF_TMIN** function block extends a simple D flip-flop by adding an adjustable minimum time between two takeover events. Thanks to its adapter interface and the clear separation of initialization and operation, it is particularly suitable for industrial control systems and discrete-time processing applications where excessively fast switching must be avoided.

Technical Features

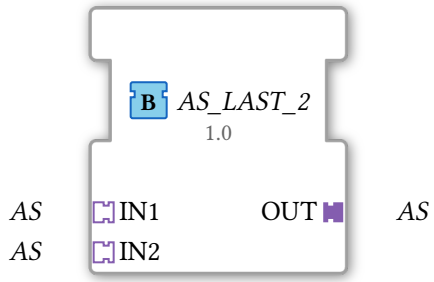
State Overview

Application Scenarios

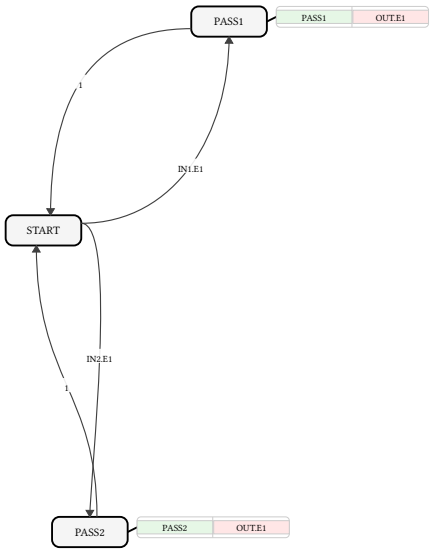
Comparison with Similar Function Blocks

Conclusion

AS_LAST_2



ECC: AS_LAST_2



Introduction

The AS_LAST_2 function block merges two same-type SINT adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: SINT adapter (unidirectional) - first signal source
- **IN2**: SINT adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: SINT adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

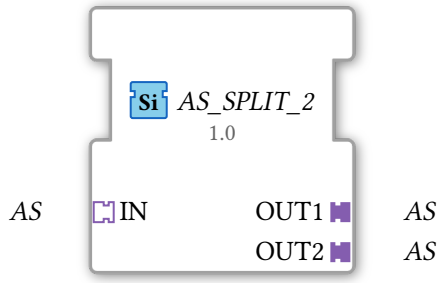
Application Scenarios

- Merging two equally-valid SINT signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same SINT value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AS_LAST_2 is the inverse of `AS_SPLIT_2`: AS_SPLIT_2 distributes 1 input to 2 outputs, AS_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

AS_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_2** splits an incoming adapter signal of type `adapter::types::unidirectional::AS` into two identical output adapters. It is designed as a generic function block and is suitable for applications where an adapter data stream needs to be distributed to multiple downstream components without altering the original signal.

Interface Structure

Event Inputs

None. The function block operates purely data-driven via the adapters.

Event Outputs

None.

Data Inputs

None. All data transmission occurs via the adapter input `IN`.

Data Outputs

None. The output data is provided via the adapters `OUT1` and `OUT2`.

Adapters

Direction	Name	Type	Description
Socket	IN	adapter::types::unidirectional::AS	Input adapter whose data is distributed to both outputs.
Plug	OUT1	adapter::types::unidirectional::AS	First output adapter – receives a copy of the incoming data.
Plug	OUT2	adapter::types::unidirectional::AS	Second output adapter – also receives a copy of the incoming data.

Functionality

The module functions as a passive distributor: An adapter data stream present at socket `IN` is forwarded unchanged to both plugs `OUT1` and `OUT2`. No data manipulation, buffering, or filtering takes place. The distribution is signal-accurate, so the data present at `OUT1` and `OUT2` corresponds exactly in time and content to the data at `IN`.

Technical Features

- **Generic Type:** The module is declared as a generic function block (`GenericClassName = GEN_AS_SPLIT`). This allows for use in various contexts, provided the connected adapters are of type `AS`.
- **No Event Control:** Data flows solely through the adapter interfaces. There are no event inputs or outputs, meaning the module has no independent sequence control.
- **No Data Buffer:** Since neither memory nor delay is implemented, the module is particularly suitable for low-latency, real-time applications.

State Overview

The module has no internal states or sequential logic. It behaves like a pure wiring component and operates continuously without an explicit state machine.

Application Scenarios

- **Signal Fan-out:** Distributing a sensor adapter to multiple evaluation units connected in parallel.
- **Redundancy:** Feeding a control signal into two independent actuator circuits.
- **Monitoring:** Connecting an analysis or logging adapter in parallel to the existing data path.

Comparison with Similar Function Blocks

Unlike a simple connection node (which only implements a 1:1 connection), `AS_SPLIT_2` enables clean, configurable distribution across two outputs. Compared to an **AS_MUX** or **AS_DEMUX**, this function block lacks selection or prioritization logic – it always distributes all incoming data to all outputs. Similar blocks like `AS_SPLIT_3` or `AS_SPLIT_N` expand the number of outputs accordingly.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AS_SPLIT_2** is a simple yet useful function block for duplicating adapter data streams. Its generic definition and the absence of complex logic make it ideally suited as a universal distribution element in 4diac-based automation systems where signal splitting without data modification is required.

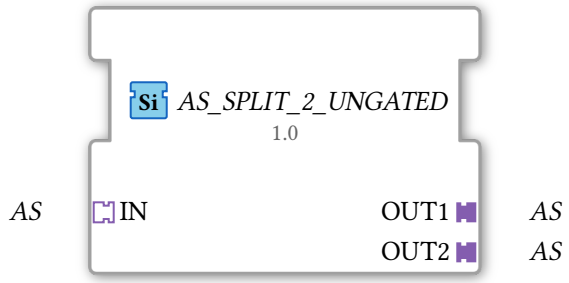
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AS_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change.

This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_2_UNGATED** splits an incoming adapter signal of type `adapter::types::unidirectional::AS` into two identical output adapters. It is designed as a generic function block and is suitable for applications where an adapter data stream needs to be distributed to multiple downstream components without altering the original signal.

Interface Structure

Event Inputs

None. The function block operates purely data-driven via the adapters.

Event Outputs

None.

Data Inputs

None. All data transmission occurs via the adapter input `IN`.

Data Outputs

None. The output data is provided via the adapters `OUT1` and `OUT2`.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AS`	Input adapter whose data is distributed to both outputs.
Plug	`OUT1`	`adapter::types::unidirectional::AS`	First output adapter – receives a copy of the incoming data.
Plug	`OUT2`	`adapter::types::unidirectional::AS`	Second output adapter – also receives a copy of the incoming data.

Functionality

The module functions as a passive distributor: An adapter data stream present at socket `IN` is forwarded unchanged to both plugs `OUT1` and `OUT2`. No data manipulation, buffering, or filtering takes place. The distribution is signal-accurate, so the data present at `OUT1` and `OUT2` corresponds exactly in time and content to the data at `IN`.

Technical Features

- **Generic Type:** The module is declared as a generic function block (`GenericClassName = GEN_AS_SPLIT`). This allows for use in various contexts, provided the connected adapters are of type `AS`.
- **No Event Control:** Data flows solely through the adapter interfaces. There are no event inputs or outputs, meaning the module has no independent sequence control.

- **No Data Buffer:** Since neither memory nor delay is implemented, the module is particularly suitable for low-latency, real-time applications.

State Overview

The module has no internal states or sequential logic. It behaves like a pure wiring component and operates continuously without an explicit state machine.

Application Scenarios

- **Signal Fan-out:** Distributing a sensor adapter to multiple evaluation units connected in parallel.
- **Redundancy:** Feeding a control signal into two independent actuator circuits.
- **Monitoring:** Connecting an analysis or logging adapter in parallel to the existing data path.

Comparison with Similar Function Blocks

Unlike a simple connection node (which only implements a 1:1 connection), `AS_SPLIT_2_UNGATED` enables clean, configurable distribution across two outputs. Compared to an **AS_MUX** or **AS_DEMUX**, this function block lacks selection or prioritization logic – it always distributes all incoming data to all outputs. Similar blocks like `AS_SPLIT_3` or `AS_SPLIT_N` expand the number of outputs accordingly.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

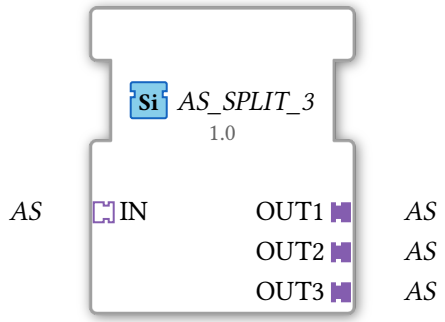
Conclusion

The **AS_SPLIT_2_UNGATED** is a simple yet useful function block for duplicating adapter data streams. Its generic definition and the absence of complex logic make it ideally suited as a universal distribution element in 4diac-based automation systems where signal splitting without data modification is required.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AS_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The AS_SPLIT_3 function block distributes an incoming adapter data stream to three identical outputs. It is implemented as a generic function block and designed for use with the `adapter::types::unidirectional::AS` adapter type.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::AS`	Socket	Input adapter – Data source distributed to three outputs.
OUT1	`adapter::types::unidirectional::AS`	Plug	First output – receives the unchanged data from IN.
OUT2	`adapter::types::unidirectional::AS`	Plug	Second output – receives the unchanged data from IN.
OUT3	`adapter::types::unidirectional::AS`	Plug	Third output – receives the unchanged data from IN.

Functionality

The module forwards the adapter data received via socket `IN` identically and without delay to all three adapter plugs `OUT1`, `OUT2`, and `OUT3`. No data manipulation, filtering, or buffering takes place. Distribution occurs in a pure pass-through mode.

Technical Features

- **Generic Type**: The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AS_SPLIT'` to enable generic instantiation. The specific adapter type (`adapter::types::unidirectional::AS`) is defined during project configuration.
- **Eventless**: Since no events (event inputs/outputs) are defined, data transmission is purely data-driven – changes to the input adapter are immediately propagated to all outputs.
- **Registered Copyright**: The function block is copyrighted under the Eclipse Public License 2.0.

State Overview

The function block does not have an internal state machine or an ECC (Execution Control Chart). The behavior is purely combinatorial: As soon as the input adapter data changes, the output data is updated accordingly.

Application Scenarios

- **Data Distribution in Control Systems**: Splitting a sensor data stream (e.g., AS-i bus data) across multiple parallel processing blocks.
- **Signal Copying for Diagnostic Purposes**: Connecting a separate monitor or logging path without affecting the main data.
- **Control Signal Multiplication**: Distributing a single instruction set to multiple actuators or subsystems.

Comparison with Similar Function Blocks

- **AS_SPLIT_2**: An analog function block with only two outputs. AS_SPLIT_3 extends the number to three outputs.
- **AS_MERGE (Theoretical)**: Unlike a merge function block, which combines multiple inputs into one output, AS_SPLIT_3 performs the reverse function (1 ⟶ N).
- **Generic Split Function Blocks**: Similar concepts exist for data inputs (e.g., SPLIT_INT), but these work with elementary data, while AS_SPLIT_3 processes adapters (complex data types).

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AS_SPLIT_3 is a simple yet useful function block for duplicating adapter data in the 4diac IDE. Its generic design and the absence of event traffic make it particularly lightweight and usable in many scenarios. The documented license (EPL 2.0) ensures that the block can be legally integrated into your own projects.

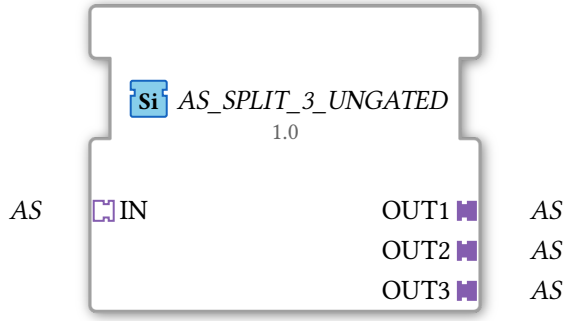
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AS_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The AS_SPLIT_3_UNGATED function block distributes an incoming adapter data stream to three identical outputs. It is implemented as a generic function block and designed for use with the `adapter::types::unidirectional::AS` adapter type.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::AS`	Socket	Input adapter – Data source distributed to three outputs.

OUT1	`adapter::types::unidirectional::AS`	Plug	First output – receives the unchanged data from IN.
OUT2	`adapter::types::unidirectional::AS`	Plug	Second output – receives the unchanged data from IN.
OUT3	`adapter::types::unidirectional::AS`	Plug	Third output – receives the unchanged data from IN.

Functionality

The module forwards the adapter data received via socket `IN` identically and without delay to all three adapter plugs `OUT1`, `OUT2`, and `OUT3`. No data manipulation, filtering, or buffering takes place. Distribution occurs in a pure pass-through mode.

Technical Features

- **Generic Type**: The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AS_SPLIT'` to enable generic instantiation. The specific adapter type (`adapter::types::unidirectional::AS`) is defined during project configuration.
- **Eventless**: Since no events (event inputs/outputs) are defined, data transmission is purely data-driven – changes to the input adapter are immediately propagated to all outputs.
- **Registered Copyright**: The function block is copyrighted under the Eclipse Public License 2.0.

State Overview

The function block does not have an internal state machine or an ECC (Execution Control Chart). The behavior is purely combinatorial: As soon as the input adapter data changes, the output data is updated accordingly.

Application Scenarios

- **Data Distribution in Control Systems**: Splitting a sensor data stream (e.g., AS-i bus data) across multiple parallel processing blocks.
- **Signal Copying for Diagnostic Purposes**: Connecting a separate monitor or logging path without affecting the main data.
- **Control Signal Multiplication**: Distributing a single instruction set to multiple actuators or subsystems.

Comparison with Similar Function Blocks

- **AS_SPLIT_2**: An analog function block with only two outputs. AS_SPLIT_3_UNGATED extends the number to three outputs.
- **AS_MERGE (Theoretical)**: Unlike a merge function block, which combines multiple inputs into one output, AS_SPLIT_3_UNGATED performs the reverse function (1 ⟶ N).
- **Generic Split Function Blocks**: Similar concepts exist for data inputs (e.g., SPLIT_INT), but these work with elementary data, while AS_SPLIT_3_UNGATED processes adapters (complex data types).

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

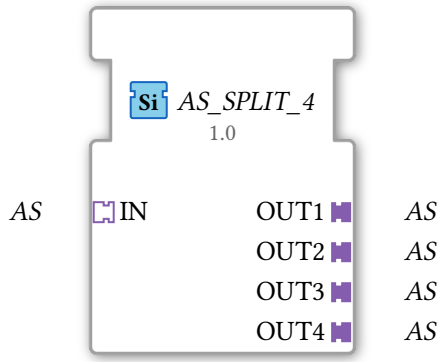
AS_SPLIT_3_UNGATED is a simple yet useful function block for duplicating adapter data in the 4diac IDE. Its generic design and the absence of event traffic make it particularly lightweight and usable in many scenarios. The documented license (EPL 2.0) ensures that the block can be legally integrated into your own projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AS_SPLIT_4



No image available.

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_4** splits an incoming unidirectional AS adapter into four identical outputs. It acts as a generic splitter, forwarding the signal received via socket `IN` unchanged to the four plugs `OUT1` to `OUT4`.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AS`	IN	Socket	Incoming AS signal path (input)
`adapter::types::unidirectional::AS`	OUT1	Plug	First output (identical to IN)
`adapter::types::unidirectional::AS`	OUT2	Plug	Second output (identical to IN)
`adapter::types::unidirectional::AS`	OUT3	Plug	Third output (identical to IN)
`adapter::types::unidirectional::AS`	OUT4	Plug	Fourth output (identical to IN)

Functionality

The function block (FB) operates purely passively: Every signal received via the `IN` adapter is forwarded to all four plugs (`OUT1`–`OUT4`) without delay or transformation. No data buffering, logical processing, or state changes occur. Distribution is deterministic and takes place at compile time or runtime according to the adapter contracts.

Technical Features

- **No Event/Data Interfaces:** All communication occurs exclusively via adapters.
- **Uniform Type:** All five adapters are of the same unidirectional type, `AS`.

- **Generic Parameterization:** The function block (FB) has the attributes `GenericClassName` (value: `'GEN_AS_SPLIT'`) and `TypeHash`, allowing it to be adapted to specific applications (e.g., through refactoring or model transformation).
- **Stateless:** No ECC, no internal variables – the behavior is completely defined by the adapter contracts.

State Overview

The FB has no state machines. Its behavior is timeless and constant: The input signal is always immediately duplicated to all outputs.

Application Scenarios

- **Signal Distribution in Controllers:** Distributing a speed or position command across multiple parallel axes.
- **Sensor Value Multiplexing:** Distributing a measured value to multiple evaluation or monitoring instances.
- **Modular Connection:** Used in hierarchical systems where an adapter signal is needed multiple times without instantiating the sender multiple times.

Comparison with Similar Components

- **AS_MERGE:** Combines multiple AS inputs into one output – the functional counterpart to Split.
- **AS_SPLIT_2:** Same functionality, but only two outputs. `AS_SPLIT_4` extends this to four.
- **Event-Based Splitters (e.g., E_SPLIT):** Work with event and data inputs/outputs and distribute events. `AS_SPLIT_4`, on the other hand, operates at the adapter level and forwards the entire adapter contract (including all events and data contained within it).

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

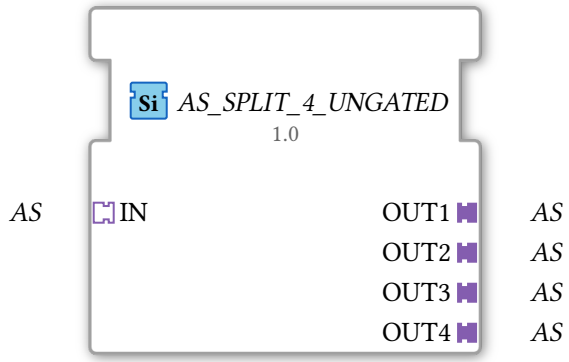
AS_SPLIT_4` is a simple yet essential building block for structurally replicating unidirectional AS adapter connections in 4diac applications. Its generic design and pure adapter interface make it ideal for modular and scalable control architectures.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AS_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



No image available.

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_4_UNGATED** splits an incoming unidirectional AS adapter into four identical outputs. It acts as a generic splitter, forwarding the signal received via socket `IN` unchanged to the four plugs `OUT1` to `OUT4`.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AS`	IN	Socket	Incoming AS signal path (input)
`adapter::types::unidirectional::AS`	OUT1	Plug	First output (identical to IN)
`adapter::types::unidirectional::AS`	OUT2	Plug	Second output (identical to IN)
`adapter::types::unidirectional::AS`	OUT3	Plug	Third output (identical to IN)
`adapter::types::unidirectional::AS`	OUT4	Plug	Fourth output (identical to IN)

Functionality

The function block (FB) operates purely passively: Every signal received via the `IN` adapter is forwarded to all four plugs (`OUT1`–`OUT4`) without delay or transformation. No data buffering, logical processing, or state changes occur. Distribution is deterministic and takes place at compile time or runtime according to the adapter contracts.

Technical Features

- **No Event/Data Interfaces:** All communication occurs exclusively via adapters.
- **Uniform Type:** All five adapters are of the same unidirectional type, `AS`.
- **Generic Parameterization:** The function block (FB) has the attributes `GenericClassName` (value: `'GEN_AS_SPLIT'`) and `TypeHash`, allowing it to be adapted to specific applications (e.g., through refactoring or model transformation).
- **Stateless:** No ECC, no internal variables – the behavior is completely defined by the adapter contracts.

State Overview

The FB has no state machines. Its behavior is timeless and constant: The input signal is always immediately duplicated to all outputs.

Application Scenarios

- **Signal Distribution in Controllers:** Distributing a speed or position command across multiple parallel axes.
- **Sensor Value Multiplexing:** Distributing a measured value to multiple evaluation or monitoring instances.
- **Modular Connection:** Used in hierarchical systems where an adapter signal is needed multiple times without instantiating the sender multiple times.

Comparison with Similar Components

- **AS_MERGE:** Combines multiple AS inputs into one output – the functional counterpart to Split.
- **AS_SPLIT_2:** Same functionality, but only two outputs. `AS_SPLIT_4_UNGATED` extends this to four.
- **Event-Based Splitters (e.g., E_SPLIT):** Work with event and data inputs/outputs and distribute events. `AS_SPLIT_4_UNGATED`, on the other hand, operates at the adapter level and forwards the entire adapter contract (including all events and data contained within it).

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

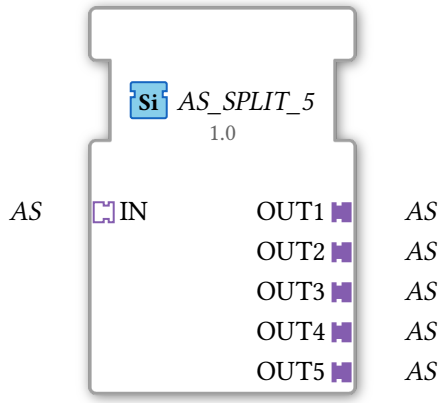
Conclusion

AS_SPLIT_4_UNGATED` is a simple yet essential building block for structurally replicating unidirectional AS adapter connections in 4diac applications. Its generic design and pure adapter interface make it ideal for modular and scalable control architectures.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AS_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The AS_SPLIT_5 function block is a generic function block that distributes one incoming adapter of type “AS” (unidirectional) to five separate output adapters of the same type. It is used for signal multiplication in control and automation systems, based on the IEC 61499-2 standard.

Interface Structure

Event Inputs

No event inputs defined.

Event Outputs

No event outputs defined.

Data Inputs

No data inputs defined.

Data Outputs

No data outputs defined.

Adapters

Role	Name	Type	Description
Socket	IN	adapter::types::unidirectional::AS	Input adapter that receives the AS signal to be distributed.
Plug	OUT1	adapter::types::unidirectional::AS	First output adapter for the distributed signal.
Plug	OUT2	adapter::types::unidirectional::AS	Second output adapter.
Plug	OUT3	adapter::types::unidirectional::AS	Third output adapter.
Plug	OUT4	adapter::types::unidirectional::AS	Fourth output adapter.
Plug	OUT5	adapter::types::unidirectional::AS	Fifth output adapter.

Functionality

The AS_SPLIT_5 block functions as a passive distributor. The AS signal present at socket "IN" is forwarded identically to all five output plugs (OUT1 to OUT5) without delay or modification. Distribution strictly follows the broadcast principle – each outgoing signal corresponds to the original signal. The implementation is purely adapter-based and requires no additional event or data port connections.

Technical Features

- The block is defined as a generic type (GenericClassName = 'GEN_AS_SPLIT') and can therefore be parameterized using the 4diac IDE.
- A type hash is used to ensure the block's uniqueness within the project.
- The number of outputs is fixed at five; dynamic configuration is not supported.
- The runtime environment must support the unidirectional adapter interface "adapter::types::unidirectional::AS".

State Overview

The block does not have an internal state machine. Signal distribution is continuous and without sequential logic. Therefore, a status overview is not required.

Application Scenarios

- **Control Wiring**: Splitting an AS signal from a higher-level controller to multiple substations or actuators.
- **Simulation and Testing**: Multiplying a test signal for parallel control of multiple simulated components.
- **Redundancy**: Providing the same signal on different paths to enable redundancy or load balancing.

Comparison with Similar Components

Since AS_SPLIT_5 is designed for a specific adapter architecture (unidirectional AS), similar components typically exist, such as "AS_SPLIT_2", "AS_SPLIT_3", or generalized splitters with a variable number of outputs. This component differs in its fixed number of five outputs and its pure adapter interface without data or event ports.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AS_SPLIT_5 is a simple yet effective function block for multiplying an AS adapter signal into five parallel outputs. Its generic implementation and clear interface make it well-suited for modular automation solutions according to IEC 61499.

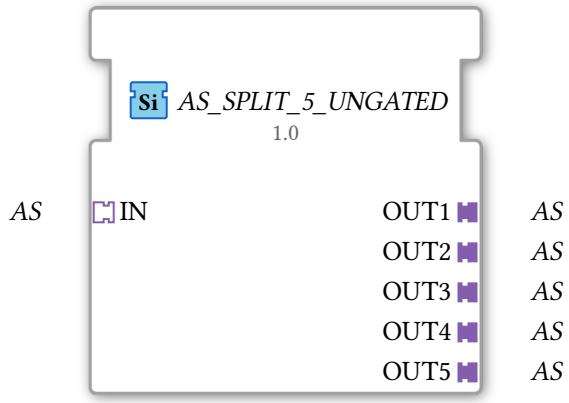
Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AS_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of AS_SPLIT_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The AS_SPLIT_5_UNGATED function block is a generic function block that distributes one incoming adapter of type "AS" (unidirectional) to five separate output adapters of the same type. It is used for signal multiplication in control and automation systems, based on the IEC 61499-2 standard.

Interface Structure

Event Inputs

No event inputs defined.

Event Outputs

No event outputs defined.

Data Inputs

No data inputs defined.

Data Outputs

No data outputs defined.

Adapters

Role	Name	Type	Description
Socket	IN	adapter::types::unidirectional::AS	Input adapter that receives the AS signal to be distributed.
Plug	OUT1	adapter::types::unidirectional::AS	First output adapter for the distributed signal.
Plug	OUT2	adapter::types::unidirectional::AS	Second output adapter.
Plug	OUT3	adapter::types::unidirectional::AS	Third output adapter.
Plug	OUT4	adapter::types::unidirectional::AS	Fourth output adapter.

Plug	OUT5	adapter::types::unidirectional::AS	Fifth output adapter.

Functionality

The AS_SPLIT_5_UNGATED block functions as a passive distributor. The AS signal present at socket “IN” is forwarded identically to all five output plugs (OUT1 to OUT5) without delay or modification. Distribution strictly follows the broadcast principle – each outgoing signal corresponds to the original signal. The implementation is purely adapter-based and requires no additional event or data port connections.

Technical Features

- The block is defined as a generic type (GenericClassName = ‘GEN_AS_SPLIT’) and can therefore be parameterized using the 4diac IDE.
- A type hash is used to ensure the block’s uniqueness within the project.
- The number of outputs is fixed at five; dynamic configuration is not supported.
- The runtime environment must support the unidirectional adapter interface “adapter::types::unidirectional::AS”.

State Overview

The block does not have an internal state machine. Signal distribution is continuous and without sequential logic. Therefore, a status overview is not required.

Application Scenarios

- **Control Wiring**: Splitting an AS signal from a higher-level controller to multiple substations or actuators.
- **Simulation and Testing**: Multiplying a test signal for parallel control of multiple simulated components.
- **Redundancy**: Providing the same signal on different paths to enable redundancy or load balancing.

Comparison with Similar Components

Since AS_SPLIT_5_UNGATED is designed for a specific adapter architecture (unidirectional AS), similar components typically exist, such as “AS_SPLIT_2”, “AS_SPLIT_3”, or generalized splitters with a variable number of outputs. This component differs in its fixed number of five outputs and its pure adapter interface without data or event ports.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

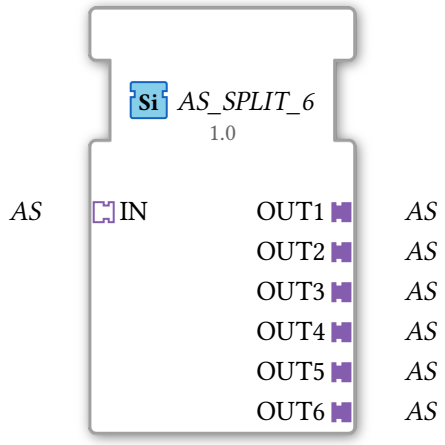
The AS_SPLIT_5_UNGATED is a simple yet effective function block for multiplying an AS adapter signal into five parallel outputs. Its generic implementation and clear interface make it well-suited for modular automation solutions according to IEC 61499.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AS_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_6** is used to split a single incoming AS adapter signal into six separate AS adapter outputs. It is implemented as a generic block and can be used with various unidirectional AS adapter types (e.g., sensor or actuator adapters).

Interface Structure

Event Inputs

– None –

Event Outputs

– None –

Data Inputs

– None –

Data Outputs

– None –

Adapter

Direction	Name	Type
SOCKET	IN	unidirectional::AS (generic)
PLUG	OUT1	unidirectional::AS (generic)
PLUG	OUT2	unidirectional::AS (generic)
PLUG	OUT3	unidirectional::AS (generic)
PLUG	OUT4	unidirectional::AS (generic)
PLUG	OUT5	unidirectional::AS (generic)
PLUG	OUT6	unidirectional::AS (generic)

Functionality

This component forwards all adapter signals present at the SOCKET **IN** unchanged to all six PLUGs (**OUT1** to **OUT6**). No data processing or conversion takes place – the distribution is purely structural. Therefore, all outputs behave identically to the inputs.

Technical Features

- **Generic Block:** The specific AS adapter type is only defined during instantiation (e.g., by specifying a GenericClassName such as `'GEN_AS_SPLIT'`). This allows the block to be used with various protocols or data structures.
- **No State Logic:** The block has no state machine, no algorithms, and no events – it operates purely passively and immediately.
- **Unidirectional:** The adapters are of type `unidirectional::AS`, meaning data flows only from the input to the outputs.

State Overview

The block does not have an End-Centre Conditioning (ECC) function, as no sequential processes or processing steps are defined. Signal distribution is static and continuous.

Application Scenarios

- **Signal Multiplication:** A single sensor adapter (e.g., temperature measurement) is provided to multiple processing units.
- **Data Bus Coupling:** A common signal is passed on to six parallel control modules or displays.
- **Prototyping and Testing:** Simplified generation of multiple identical interfaces from a single source.

Comparison with Similar Function Blocks

- **AS_SPLIT_2 / AS_SPLIT_4** – Identical functionality, but with only two or four outputs, respectively. This function block is suitable for applications with exactly six destinations.
- **AS_MERGE_...** – Combines multiple adapter inputs into one output – the opposite task.
- **Manual Wiring** – Could be implemented using multiple adapter couplings, but increases the configuration effort.

Change Detection

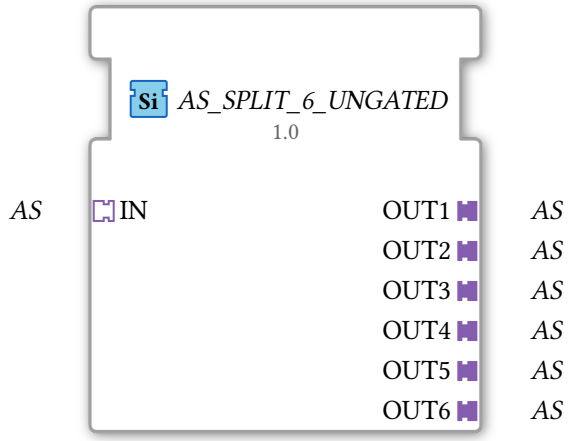
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AS_SPLIT_6** is a simple yet effective generic function block for splitting a unidirectional AS signal into six separate outputs. Its generic nature allows for flexible use in various automation environments where fast and clear signal duplication is required.

AS_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of AS_SPLIT_6. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_6_UNGATED** is used to split a single incoming AS adapter signal into six separate AS adapter outputs. It is implemented as a generic block and can be used with various unidirectional AS adapter types (e.g., sensor or actuator adapters).

Interface Structure

Event Inputs

– None –

Event Outputs

– None –

Data Inputs

– None –

Data Outputs

– None –

Adapter

Direction	Name	Type
SOCKET	IN	unidirectional::AS (generic)
PLUG	OUT1	unidirectional::AS (generic)
PLUG	OUT2	unidirectional::AS (generic)
PLUG	OUT3	unidirectional::AS (generic)
PLUG	OUT4	unidirectional::AS (generic)
PLUG	OUT5	unidirectional::AS (generic)
PLUG	OUT6	unidirectional::AS (generic)

Functionality

This component forwards all adapter signals present at the SOCKET **IN** unchanged to all six PLUGs (**OUT1** to **OUT6**). No data processing or conversion takes place – the distribution is purely structural. Therefore, all outputs behave identically to the inputs.

Technical Features

- **Generic Block:** The specific AS adapter type is only defined during instantiation (e.g., by specifying a GenericClassName such as `'GEN_AS_SPLIT'`). This allows the block to be used with various protocols or data structures.
- **No State Logic:** The block has no state machine, no algorithms, and no events – it operates purely passively and immediately.
- **Unidirectional:** The adapters are of type `unidirectional::AS`, meaning data flows only from the input to the outputs.

State Overview

The block does not have an End-Centre Conditioning (ECC) function, as no sequential processes or processing steps are defined. Signal distribution is static and continuous.

Application Scenarios

- **Signal Multiplication:** A single sensor adapter (e.g., temperature measurement) is provided to multiple processing units.
- **Data Bus Coupling:** A common signal is passed on to six parallel control modules or displays.
- **Prototyping and Testing:** Simplified generation of multiple identical interfaces from a single source.

Comparison with Similar Function Blocks

- **AS_SPLIT_2 / AS_SPLIT_4** – Identical functionality, but with only two or four outputs, respectively. This function block is suitable for applications with exactly six destinations.
- **AS_MERGE_...** – Combines multiple adapter inputs into one output – the opposite task.
- **Manual Wiring** – Could be implemented using multiple adapter couplings, but increases the configuration effort.

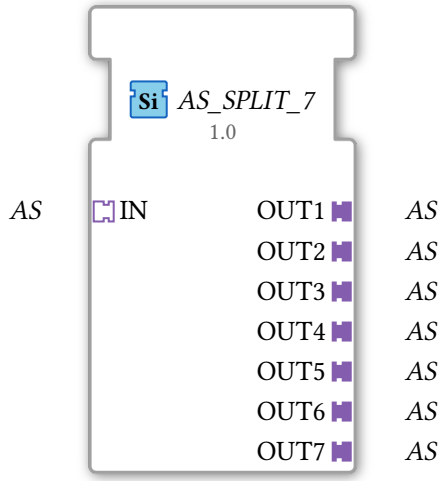
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AS_SPLIT_6_UNGATED** is a simple yet effective generic function block for splitting a unidirectional AS signal into six separate outputs. Its generic nature allows for flexible use in various automation environments where fast and clear signal duplication is required.

AS_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_7** is used to distribute an incoming unidirectional AS signal (e.g., an analog or binary automation signal) to seven identical outputs. The block is implemented as a generic FB and

is frequently used in automation technology to provide a signal to multiple subsequent consumers or process steps.

Interface Structure

Event Inputs
None

Event Outputs
None

Data Inputs
None (Signal transmission occurs exclusively via adapters.)

Data Outputs
None

Adapters

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::AS`	Socket	Receives the AS signal to be distributed.
OUT1	`adapter::types::unidirectional::AS`	Plug	First output – same signal as the input.
OUT2	`adapter::types::unidirectional::AS`	Plug	Second output – same signal as the input.
OUT3	`adapter::types::unidirectional::AS`	Plug	Third output – same signal as the input.
OUT4	`adapter::types::unidirectional::AS`	Plug	Fourth output – same signal as the input.
OUT5	`adapter::types::unidirectional::AS`	Plug	Fifth output – same signal as the input.
OUT6	`adapter::types::unidirectional::AS`	Plug	Sixth output – same signal as the input.
OUT7	`adapter::types::unidirectional::AS`	Plug	Seventh output – same signal as the input.

Functionality
The function block forwards the AS signal present at socket `IN` unchanged to all seven plug outputs. No signal conditioning, amplification, or processing takes place. The function block operates purely passively and without any time delay – as soon as a signal is present at the input, it is available at all outputs simultaneously.

Technical Features
- **Generic Function Block:** The function block is implemented as a generic type (`GEN_AS_SPLIT`) and can be used in various contexts by configuring its parameters.
- **Adapter-Based Interface:** Communication occurs exclusively via unidirectional adapters of type `adapter::types::unidirectional::AS`. This makes the function block particularly suitable for modular architectures where signals are exchanged via clearly defined interfaces.
- **No State Logic:** The function block has no internal states, events, or processing logic – distribution is purely combinatorial.
- **Easy Extensibility:** Due to the generic nature of the function block, analog splitter variants (e.g., `AS_SPLIT_2`, `AS_SPLIT_4`) can be easily derived or created by adjusting the parameters.

State Overview
The **AS_SPLIT_7** has no state machines, as it operates purely passively. There are no start, stop, or error states. Its behavior is determined solely by the current value of the input signal at all times.

Application Scenarios

- **Signal Distribution in Control Systems:** An analog measurement signal (e.g., pressure, temperature) is distributed to multiple displays, controllers, or data loggers.
- **Parallel Connection of Actuators:** A control signal (e.g., 0-10V) is intended to control multiple actuators simultaneously.
- **Redundant Signal Paths:** The same signal is provided to multiple independent processing units to create redundancy.
- **Test and Simulation Environments:** A generated test signal is distributed across multiple simulation components.

Comparison with Similar Function Blocks

The **AS_SPLIT_7** belongs to a family of splitter function blocks that differ only in the number of outputs (e.g., `AS_SPLIT_2`, `AS_SPLIT_3`, ..., `AS_SPLIT_N`). Compared to manual wiring, the function block offers the advantages of reusability, clear interface definition, and easy parameterization. Unlike active signal distributors (e.g., with amplification), the function block has no effect on the signal and is therefore ideal for passive applications.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AS_SPLIT_7** is a simple yet useful function block for passively distributing a unidirectional AS signal to seven outputs. Thanks to its generic implementation and adapter interface, it integrates seamlessly into modern, component-based automation solutions. For applications requiring simple signal multiplication without processing, it offers a reliable and easy-to-understand solution.

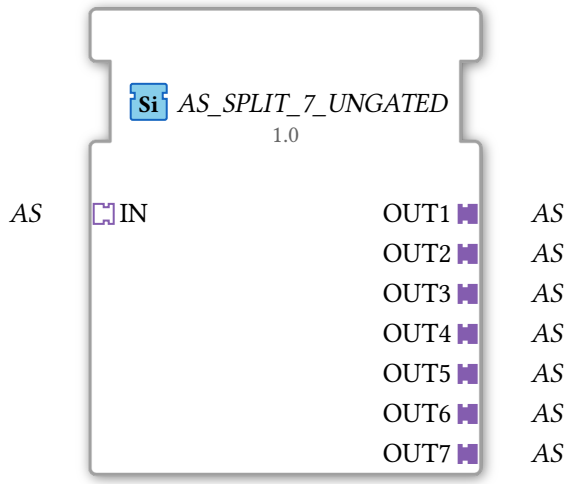
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

]

AS_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_7_UNGATED** is used to distribute an incoming unidirectional AS signal (e.g., an analog or binary automation signal) to seven identical outputs. The block is implemented as a generic FB and is frequently used in automation technology to provide a signal to multiple subsequent consumers or process steps.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None (Signal transmission occurs exclusively via adapters.)

Data Outputs

None

Adapters

Name	Type	Direction	Description
IN	`adapter::types::unidirectional::AS`	Socket	Receives the AS signal to be distributed.
OUT1	`adapter::types::unidirectional::AS`	Plug	First output – same signal as the input.
OUT2	`adapter::types::unidirectional::AS`	Plug	Second output – same signal as the input.
OUT3	`adapter::types::unidirectional::AS`	Plug	Third output – same signal as the input.
OUT4	`adapter::types::unidirectional::AS`	Plug	Fourth output – same signal as the input.
OUT5	`adapter::types::unidirectional::AS`	Plug	Fifth output – same signal as the input.
OUT6	`adapter::types::unidirectional::AS`	Plug	Sixth output – same signal as the input.
OUT7	`adapter::types::unidirectional::AS`	Plug	Seventh output – same signal as the input.

Functionality

The function block forwards the AS signal present at socket `IN` unchanged to all seven plug outputs. No signal conditioning, amplification, or processing takes place. The function block operates purely passively and without any time delay – as soon as a signal is present at the input, it is available at all outputs simultaneously.

Technical Features

- **Generic Function Block:** The function block is implemented as a generic type (`GEN_AS_SPLIT`) and can be used in various contexts by configuring its parameters.
- **Adapter-Based Interface:** Communication occurs exclusively via unidirectional adapters of type `adapter::types::unidirectional::AS`. This makes the function block particularly suitable for modular architectures where signals are exchanged via clearly defined interfaces.
- **No State Logic:** The function block has no internal states, events, or processing logic – distribution is purely combinatorial.

- **Easy Extensibility:** Due to the generic nature of the function block, analog splitter variants (e.g., `AS_SPLIT_2`, `AS_SPLIT_4`) can be easily derived or created by adjusting the parameters.

State Overview

The **AS_SPLIT_7_UNGATED** has no state machines, as it operates purely passively. There are no start, stop, or error states. Its behavior is determined solely by the current value of the input signal at all times.

Application Scenarios

- **Signal Distribution in Control Systems:** An analog measurement signal (e.g., pressure, temperature) is distributed to multiple displays, controllers, or data loggers.
- **Parallel Connection of Actuators:** A control signal (e.g., 0-10V) is intended to control multiple actuators simultaneously.
- **Redundant Signal Paths:** The same signal is provided to multiple independent processing units to create redundancy.
- **Test and Simulation Environments:** A generated test signal is distributed across multiple simulation components.

Comparison with Similar Function Blocks

The **AS_SPLIT_7_UNGATED** belongs to a family of splitter function blocks that differ only in the number of outputs (e.g., `AS_SPLIT_2`, `AS_SPLIT_3`, ..., `AS_SPLIT_N`). Compared to manual wiring, the function block offers the advantages of reusability, clear interface definition, and easy parameterization. Unlike active signal distributors (e.g., with amplification), the function block has no effect on the signal and is therefore ideal for passive applications.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

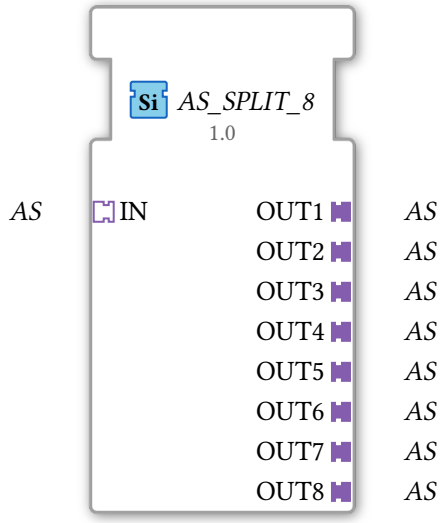
The **AS_SPLIT_7_UNGATED** is a simple yet useful function block for passively distributing a unidirectional AS signal to seven outputs. Thanks to its generic implementation and adapter interface, it integrates seamlessly into modern, component-based automation solutions. For applications requiring simple signal multiplication without processing, it offers a reliable and easy-to-understand solution.

Related topic subpages on ms-muc-docs.de

- Total resistance in series & parallel circuits on ms-muc-docs.de

]

AS_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_8** is used to split an incoming *Application Specific* (AS) adapter data stream into eight identical outputs. It is provided as a generic function block (generic FB) and is specifically designed for distributing adapter data within an IEC 61499-based control application.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Name	Direction	Description
adapter::types::unidirectional::AS`	IN	Socket (Input)	Receives the AS data stream to be distributed.
adapter::types::unidirectional::AS`	OUT1	Plug (Output)	First copy of the incoming AS data stream.
adapter::types::unidirectional::AS`	OUT2	Plug (Output)	Second copy of the incoming AS data stream.
adapter::types::unidirectional::AS`	OUT3	Plug (Output)	Third copy.
adapter::types::unidirectional::AS`	OUT4	Plug (Output)	Fourth copy.
adapter::types::unidirectional::AS`	OUT5	Plug (Output)	Fifth copy.
adapter::types::unidirectional::AS`	OUT6	Plug (Output)	Sixth Copy.
`adapter::types::unidirectional::AS`	OUT7	Plug (Output)	Seventh Copy.
`adapter::types::unidirectional::AS`	OUT8	Plug (Output)	Eighth Copy.

Functionality

The module operates purely on an adapter basis, without using events or data inputs/outputs. The adapter data stream present at socket `IN` is internally mirrored to all eight output plugs (`OUT1` … `OUT8`). Each output passes on exactly the same data that is present at the input. No delay, filtering, or modification takes place. Branching occurs passively and without active runtime logic (no state machine).

Technical Features

- **Pure Adapter Block**: The function block (FB) has neither event nor data interfaces in the traditional sense; all data transmission occurs via the adapter connections.
- **Generic Implementation**: By declaring it as a generic FB (`GenericClassName = 'GEN_AS_SPLIT'`), the block can be used for any AS adapter type, provided the underlying adapter type is `unidirectional::AS`.
- **No State Machine**: Due to the lack of events, an Execution Control Chart (ECC) is not required. Distribution is continuous.
- **Simple 1:8 Splitting**: Optimized for situations where a signal needs to be distributed to multiple subsequent modules.

State Overview

Not applicable – the block has no internal states or sequence control.

Application Scenarios

- **Parallel Distribution of Sensor Data**: A single adapter that… B. If a tank's fill level data is provided, it is distributed across eight monitoring or controlling functional units.
- **Signal Multicasting in Industry 4.0**: Distribution of a control command to multiple actuators or subsystems.
- **Test Environments** for simulating multiple receivers of a data stream.

Comparison with Similar Function Blocks

- **AS_SPLIT_2 / AS_SPLIT_4**: Analog function blocks with fewer outputs. AS_SPLIT_8 offers maximum distribution in one step with eight outputs.
- **Event-based splitters (e.g., E_SPLIT)**: These operate with events and distribute them according to a time-controlled process. In contrast, this function block continuously distributes all adapter data without event control.
- **Data-based splitters (e.g., ANY_DISTRIBUTE)**: Split data values but require additional events. AS_SPLIT_8 is optimized for simple adapter forwarding.

Change Detection

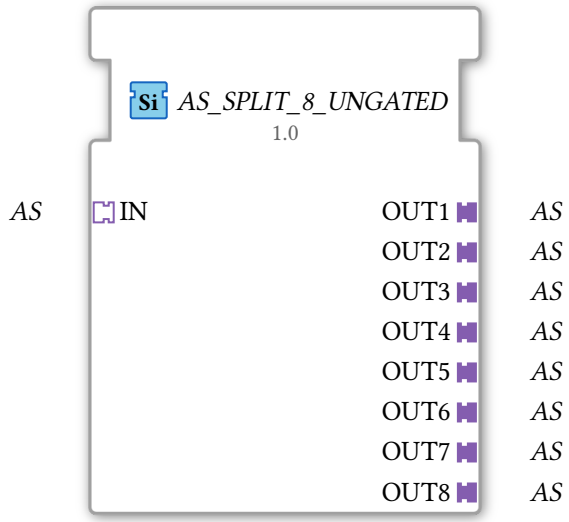
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AS_SPLIT_8** is a simple yet useful adapter distributor that duplicates an incoming AS data stream to eight outputs without requiring any additional logic. Its generic nature and clear structure make it a robust solution for scenarios where an adapter signal needs to be used multiple times. The module's documentation does not claim to be exhaustive regarding implementation details; the exact functionality may vary depending on the runtime environment used.

AS_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_8_UNGATED** is used to split an incoming *Application Specific* (AS) adapter data stream into eight identical outputs. It is provided as a generic function block (generic FB) and is specifically designed for distributing adapter data within an IEC 61499-based control application.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Name	Direction	Description
adapter::types::unidirectional::AS`	IN	Socket (Input)	Receives the AS data stream to be distributed.
adapter::types::unidirectional::AS`	OUT1	Plug (Output)	First copy of the incoming AS data stream.
adapter::types::unidirectional::AS`	OUT2	Plug (Output)	Second copy of the incoming AS data stream.
adapter::types::unidirectional::AS`	OUT3	Plug (Output)	Third copy.
adapter::types::unidirectional::AS`	OUT4	Plug (Output)	Fourth copy.
adapter::types::unidirectional::AS`	OUT5	Plug (Output)	Fifth copy.
adapter::types::unidirectional::AS`	OUT6	Plug (Output)	Sixth Copy.
`adapter::types::unidirectional::AS`	OUT7	Plug (Output)	Seventh Copy.
`adapter::types::unidirectional::AS`	OUT8	Plug (Output)	Eighth Copy.

Functionality

The module operates purely on an adapter basis, without using events or data inputs/outputs. The adapter data stream present at socket `IN` is internally mirrored to all eight output plugs (`OUT1` … `OUT8`). Each output passes on exactly the same data that is present at the input. No delay, filtering, or modification takes place. Branching occurs passively and without active runtime logic (no state machine).

Technical Features

- **Pure Adapter Block**: The function block (FB) has neither event nor data interfaces in the traditional sense; all data transmission occurs via the adapter connections.
- **Generic Implementation**: By declaring it as a generic FB (`GenericClassName = 'GEN_AS_SPLIT'`), the block can be used for any AS adapter type, provided the underlying adapter type is `unidirectional::AS`.
- **No State Machine**: Due to the lack of events, an Execution Control Chart (ECC) is not required. Distribution is continuous.
- **Simple 1:8 Splitting**: Optimized for situations where a signal needs to be distributed to multiple subsequent modules.

State Overview

Not applicable – the block has no internal states or sequence control.

Application Scenarios

- **Parallel Distribution of Sensor Data**: A single adapter that... B. If a tank's fill level data is provided, it is distributed across eight monitoring or controlling functional units.
- **Signal Multicasting in Industry 4.0**: Distribution of a control command to multiple actuators or subsystems.
- **Test Environments** for simulating multiple receivers of a data stream.

Comparison with Similar Function Blocks

- **AS_SPLIT_2 / AS_SPLIT_4**: Analog function blocks with fewer outputs. AS_SPLIT_8_UNGATED offers maximum distribution in one step with eight outputs.
- **Event-based splitters (e.g., E_SPLIT)**: These operate with events and distribute them according to a time-controlled process. In contrast, this function block continuously distributes all adapter data without event control.
- **Data-based splitters (e.g., ANY_DISTRIBUTE)**: Split data values but require additional events. AS_SPLIT_8_UNGATED is optimized for simple adapter forwarding.

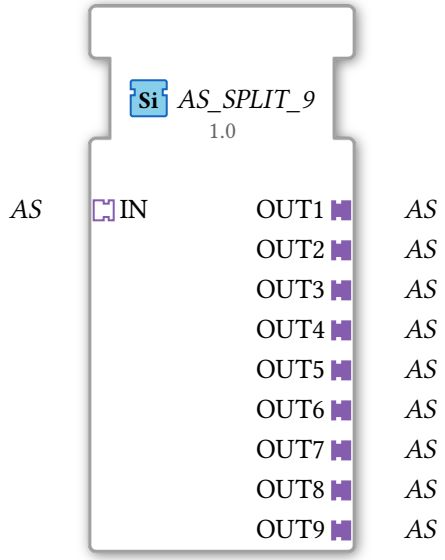
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AS_SPLIT_8_UNGATED** is a simple yet useful adapter distributor that duplicates an incoming AS data stream to eight outputs without requiring any additional logic. Its generic nature and clear structure make it a robust solution for scenarios where an adapter signal needs to be used multiple times. The module's documentation does not claim to be exhaustive regarding implementation details; the exact functionality may vary depending on the runtime environment used.

AS_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_9** is used to split an incoming **AS adapter** (unidirectional) into nine separate output adapters. It is implemented as a generic function block and distributes the AS signal present at socket `IN` identically to all nine plug outputs (`OUT1` ... `OUT9`). This makes it ideal for routing a single AS signal multiple times to different downstream components.

Interface Structure

Event Inputs
- None

Event Outputs
- None

Data Inputs
- None

Data Outputs
- None

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AS`	Input Adapter (AS Interface)
Plug	`OUT1`	`adapter::types::unidirectional::AS`	First Output Adapter
Plug	`OUT2`	`adapter::types::unidirectional::AS`	Second Output Adapter
Plug	`OUT3`	`adapter::types::unidirectional::AS`	Third Output Adapter
Plug	`OUT4`	`adapter::types::unidirectional::AS`	Fourth Output Adapter
Plug	`OUT5`	`adapter::types::unidirectional::AS`	Fifth Output Adapter
Plug	`OUT6`	`adapter::types::unidirectional::AS`	Sixth Output Adapter
Plug	`OUT7`	`adapter::types::unidirectional::AS`	Seventh Output Adapter
Plug	`OUT8`	`adapter::types::unidirectional::AS`	Eighth Output Adapter
Plug	`OUT9`	`adapter::types::unidirectional::AS`	Ninth Output Adapter

Functionality

The module receives an AS connection via socket `IN`. Internal logic forwards this incoming signal without modification to all nine plug outputs (`OUT1` … `OUT9`). This ensures that the same AS data and/or events are always available at all outputs. There is no separate state machine or internal processing – distribution occurs directly and without delay.

Technical Features

- **Generic Function Block** – The function block is classified as **GEN_AS_SPLIT** and can be parameterized as needed (e.g., for a variable number of outputs).
- **Unidirectional Adapters** – Both inputs and outputs use the type `adapter::types::unidirectional::AS`. This ensures that the data/event direction from IN to OUT is strictly maintained.
- No dedicated event or data inputs are required – the entire interface consists exclusively of adapters.

State Overview

The function block does not have an explicit state machine (ECC). Distribution occurs purely combinatorially as soon as the input adapter provides valid data or events.

Application Scenarios

- **Signal Distribution** – An AS sensor or controller must supply several actuators or monitored components simultaneously.
- **Parallel Operation** – Multiple devices are to be connected to an AS network without requiring active replication.
- **Test and Simulation Environments** – An outgoing AS signal is split across multiple test modules or analysis tools.

Comparison with Similar Components

- **AS_SPLIT_2**, **AS_SPLIT_4**, etc. – These components offer a smaller number of outputs (2, 4, etc.) and are optimized for smaller distributions.
- **AS_MUX** – A multiplexer that combines multiple inputs into one output; the opposite of a splitter.
- **AS_COPY** – Copies a signal to a second output; equivalent to a 1:2 split.
- **AS_SPLIT_9**, with nine outputs, is the largest standard variant and covers extensive distribution requirements.

Change Detection

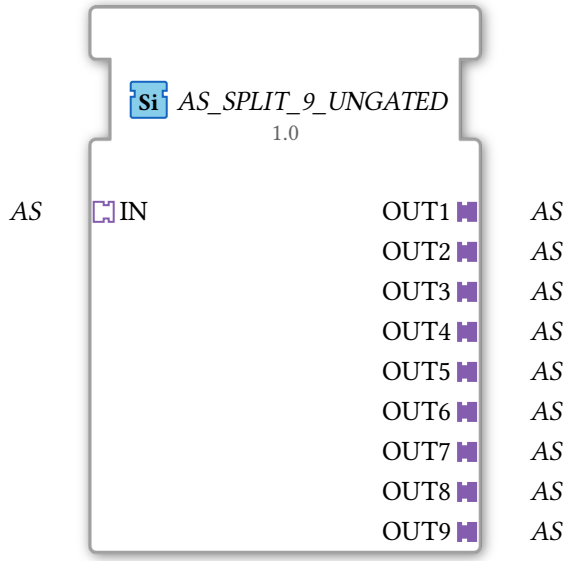
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AS_SPLIT_9** is a straightforward, reliable function block for duplicating an AS signal. Its simple adapter interface and lack of internal logic make it particularly suitable for applications where an output signal needs to be distributed to multiple receivers. Its generic nature also allows for easy adaptation to specific project requirements.

AS_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_SPLIT_9_UNGATED** is used to split an incoming **AS adapter** (unidirectional) into nine separate output adapters. It is implemented as a generic function block and distributes the AS signal present at socket `IN` identically to all nine plug outputs (`OUT1` … `OUT9`). This makes it ideal for routing a single AS signal multiple times to different downstream components.

Interface Structure

Event Inputs

- None

Event Outputs

- None

Data Inputs

- None

Data Outputs

- None

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AS`	Input Adapter (AS Interface)
Plug	`OUT1`	`adapter::types::unidirectional::AS`	First Output Adapter
Plug	`OUT2`	`adapter::types::unidirectional::AS`	Second Output Adapter
Plug	`OUT3`	`adapter::types::unidirectional::AS`	Third Output Adapter
Plug	`OUT4`	`adapter::types::unidirectional::AS`	Fourth Output Adapter
Plug	`OUT5`	`adapter::types::unidirectional::AS`	Fifth Output Adapter
Plug	`OUT6`	`adapter::types::unidirectional::AS`	Sixth Output Adapter
Plug	`OUT7`	`adapter::types::unidirectional::AS`	Seventh Output Adapter
Plug	`OUT8`	`adapter::types::unidirectional::AS`	Eighth Output Adapter
Plug	`OUT9`	`adapter::types::unidirectional::AS`	Ninth Output Adapter

Functionality

The module receives an AS connection via socket `IN`. Internal logic forwards this incoming signal without modification to all nine plug outputs (`OUT1` … `OUT9`). This ensures that the same AS data and/or events are always available at all outputs. There is no separate state machine or internal processing – distribution occurs directly and without delay.

Technical Features

- **Generic Function Block** – The function block is classified as **GEN_AS_SPLIT** and can be parameterized as needed (e.g., for a variable number of outputs).
- **Unidirectional Adapters** – Both inputs and outputs use the type `adapter::types::unidirectional::AS`. This ensures that the data/event direction from IN to OUT is strictly maintained.
- No dedicated event or data inputs are required – the entire interface consists exclusively of adapters.

State Overview

The function block does not have an explicit state machine (ECC). Distribution occurs purely combinatorially as soon as the input adapter provides valid data or events.

Application Scenarios

- **Signal Distribution** – An AS sensor or controller must supply several actuators or monitored components simultaneously.
- **Parallel Operation** – Multiple devices are to be connected to an AS network without requiring active replication.
- **Test and Simulation Environments** – An outgoing AS signal is split across multiple test modules or analysis tools.

Comparison with Similar Components

- **AS_SPLIT_2**, **AS_SPLIT_4**, etc. – These components offer a smaller number of outputs (2, 4, etc.) and are optimized for smaller distributions.
- **AS_MUX** – A multiplexer that combines multiple inputs into one output; the opposite of a splitter.

- **AS_COPY** – Copies a signal to a second output; equivalent to a 1:2 split.
- **AS_SPLIT_9_UNGATED**, with nine outputs, is the largest standard variant and covers extensive distribution requirements.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

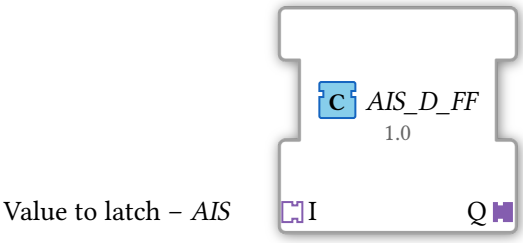
Conclusion

The **AS_SPLIT_9_UNGATED** is a straightforward, reliable function block for duplicating an AS signal. Its simple adapter interface and lack of internal logic make it particularly suitable for applications where an output signal needs to be distributed to multiple receivers. Its generic nature also allows for easy adaptation to specific project requirements.

Numeric Index (unidirectional)

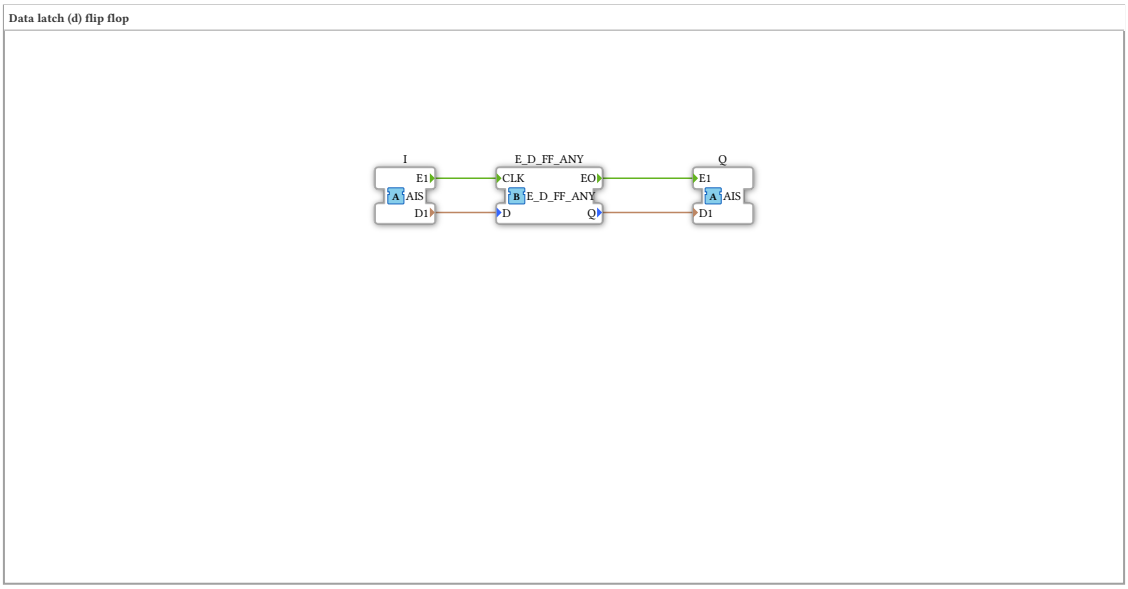
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AIS_D_FF



Value to latch – *AIS*

AIS – Latched value



• – ○ • – ○ • – ○ •

Introduction

The `AIS_D_FF` function block implements a clock-edge-triggered D flip-flop (data latch). It is used to store and output a data value upon receiving a clock event. Communication is exclusively via AIS adapters (unidirectional), allowing the function block to be integrated into adapter-based component architectures.

Interface Structure

Event Inputs

No direct event inputs. The clock signal is provided via the socket adapter **I** (event port `E1`).

Event Outputs

No direct event outputs. The output event is sent via the plug adapter **Q** (event port `E1`).

Data Inputs

No direct data inputs. The data value to be received is received via the socket adapter **I** (data port `D1`).

Data Outputs

No direct data outputs. The stored value is output via the plug adapter **Q** (data port `D1`).

Adapters

Adapter	Type	Direction	Description
I	`adapter::types::unidirectional::AIS`	Socket	Input adapter: Clock signal (event `E1`) and data input (data `D1`)
Q	`adapter::types::unidirectional::AIS`	Plug	Output adapter: Acknowledge event (event `E1`) and stored data value (data `D1`)

Functionality

This component encapsulates the IEC 61499 standard component `E_D_FF_ANY`. On a rising edge of the clock signal (event `I.E1`), the current value of `I.D1` is taken and stored internally in the flip-flop. An event is then triggered at output `Q.E1`, and the stored value is made available at port `Q.D1`. The output value remains until the next clock event arrives.

Technical Features

- **Pure Adapter Interface**: The module has no direct event or data ports but uses only AIS adapters. This allows for flexible integration with other adapter-based components.
- **Generic Data Type**: The internal `E_D_FF_ANY` module supports any data type. The data width and type are determined by the adapter implementation used.
- **Event-Driven Data Pass-Through**: The output is only updated upon an incoming clock event. Without a clock event, the output remains unchanged.

State Overview

Since the component implements the behavior of a D flip-flop, two states can be distinguished:

State	Description
Idle State	The stored value remains constant. No output event is generated.
Takeover	Upon the arrival of a clock event, the current input value is taken, the output is updated, and an output event is sent.

The flip-flop is edge-triggered (rising edge) and not level-triggered.

Application Scenarios

- **Data Buffer in Adapter Networks**: Temporarily storing values that are only valid for specific events (e.g., clock signals from sensors).
- **Synchronization**: Synchronizing data between components that communicate via asynchronous events.
- **Registers in Control Applications**: Construction of shift registers or simple memory cells in IEC 61499 systems using AIS adapters.

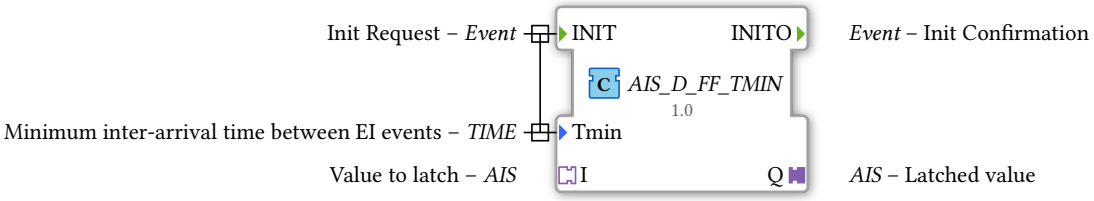
Comparison with Similar Devices

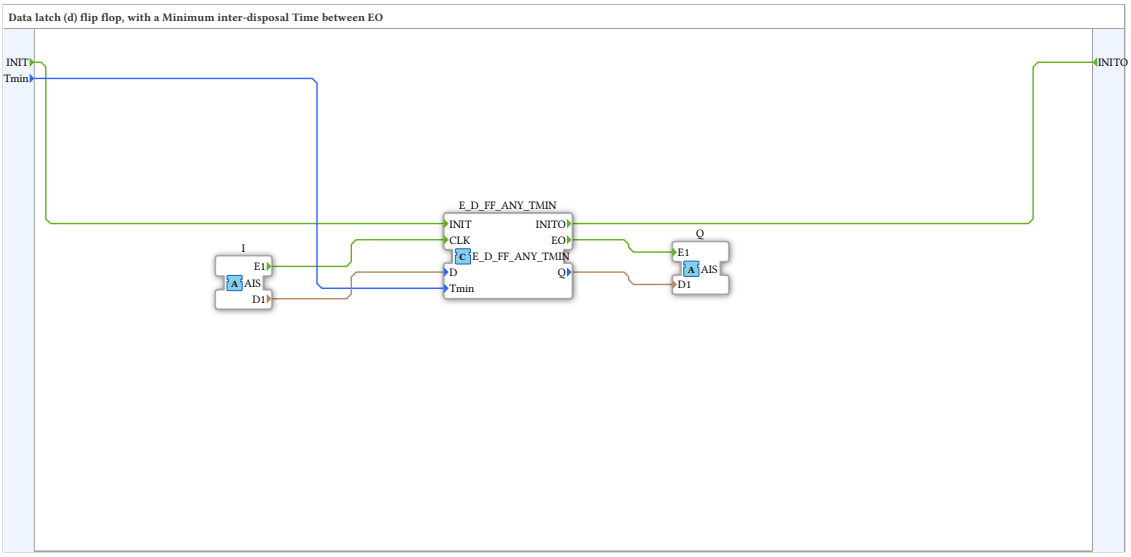
- **`E_D_FF_ANY`** – The direct function block equivalent with standard event and data ports. `AIS_D_FF` encapsulates this device in an adapter-based interface.
- **SR Flip-Flop** – Stores a binary state via set and reset inputs, while `AIS_D_FF` stores an arbitrary data value via a clock signal.
- **Latch (Level-Controlled)** – A latch holds the input value as long as the clock signal is active; In contrast, `AIS_D_FF` only reacts to an edge and is therefore less susceptible to noise and glitches.

Conclusion

AIS_D_FFis a useful component for implementing an edge-triggered D flip-flop in adapter-based architectures. By encapsulating the provenE_D_FF_ANY` functional block, it offers reliable memory functionality and can be seamlessly integrated into existing AIS communication structures.

AIS_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AIS_D_FF_TMIN** implements a data-dependent flip-flop (D flip-flop) with a minimum inter-disposal time (MIT) between two consecutive output events (EOs). It latches an incoming data value via an AIS adapter and outputs it only when the minimum time specified by the parameter `Tmin` has elapsed since the last EO. This suppresses excessively rapid signal changes and ensures a defined signal delay.

Interface Structure

Event Inputs

Event	Type	Comment
`INIT`	EInit	Initialization Request (with `Tmin`)

Event Outputs

Event	Type	Comment
`INITO`	EInit	Initialization Acknowledgement

Data Inputs

Variable	Type	Comment
`Tmin`	TIME	Minimum time interval between two EO events

Data Outputs

The FB has no data outputs of its own. Output is exclusively via the adapter plug (see below).

Adapter

Direction	Label	Adapter Type	Comment
Plug (Output)	`Q`	`adapter::types::unidirectional::AIS`	Latched output value
Socket (Input)	`I`	`adapter::types::unidirectional::AIS`	Input value to be latched

The adapter `AIS` is unidirectional and delivers the actual signals via the event/data interfaces `E1` and `D1`.

Functionality

Internally, the function block consists of a nested function block `E_D_FF_ANY_TMIN` (from the IEC 61499 library).

Processing proceeds as follows:

1. **Initialization**: Upon the event `INIT`, the parameter `Tmin` is forwarded to the internal function block, which is then initialized. Subsequently, `INITO` is triggered.
2. **Latch Operation**: An event `E1` arriving at **socket `I`** (connected to the internal `CLK`) transfers the simultaneously received data value `D1` (connected to `D`) into the flip-flop.
3. **Minimum Delay**: The internal function block checks whether at least the time `Tmin` has elapsed since the last output event (`EO`).

- If the condition is met, the latched data is output via the **plug `Q`** (event `E1` and data value `D1`).
- Otherwise, the value is held back until the required waiting time has elapsed.

This ensures that output events do not occur faster than specified by `Tmin`.

Technical Features

- **Adaptor-based I/O**: The signals are not routed as separate input/output variables, but rather via standardized unidirectional AIS adapters. This allows for flexible interconnection with other components or subsystems.
- **Maintaining a minimum delay**: The delay prevents "noise" or excessively rapid signal changes, which can lead to malfunctions in time-critical systems.
- **Internal IEC 61499 Standard Function Block (FB)**: The implementation uses the library's own `E_D_FF_ANY_TMIN`, ensuring high reliability and reusability.

State Overview

The FB itself does not have any explicitly modeled states in the XML definition. Its behavior is entirely determined by the internal FB `E_D_FF_ANY_TMIN`. This typically goes through the following phases (simplified):

- **Idle**: Waiting for a clock event at input `I.E1`.
- **Sampling**: Upon a clock event, the current data value `I.D1` is taken.
- **Delay**: After the takeover, the time elapsed since the last `EO` is compared to `Tmin`. If the time falls below the minimum, the system enters a wait state.
- **Output**: After the minimum time has elapsed, the latched data is output via `Q.E1` and `Q.D1`.

Application Scenarios

- **Signal Smoothing**: Suppression of bounces or short pulses at digital inputs.
- **Cycle Control**: Ensuring that an output signal is updated only at a defined minimum interval (e.g., every 100 ms).
- **Interface Buffering**: Buffering values between system components with different speeds, while maintaining a minimum update rate.
- **Automation Technology**: Control of actuators that require mechanical dead times or minimum switching intervals.

Comparison with Similar Function Blocks

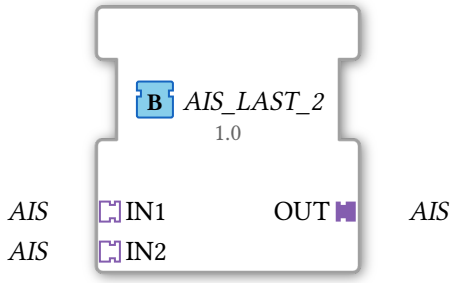
- **Simple D Flip-Flop (without timing)**: A standard D flip-flop outputs the latched value immediately with the next clock event – without a delay limit. `AIS_D_FF_TMIN` adds the additional timing condition.

- **Monostable Flip-Flop**: A monostable function block generates a pulse of fixed duration; the function block described here, however, does not maintain a pulse duration but rather a minimum interval between outputs.
- **IEC-61499 `E_D_FF_ANY`**: This is the basic version without a time limit. `E_D_FF_ANY_TMIN` extends this variant with the `Tmin` parameter.

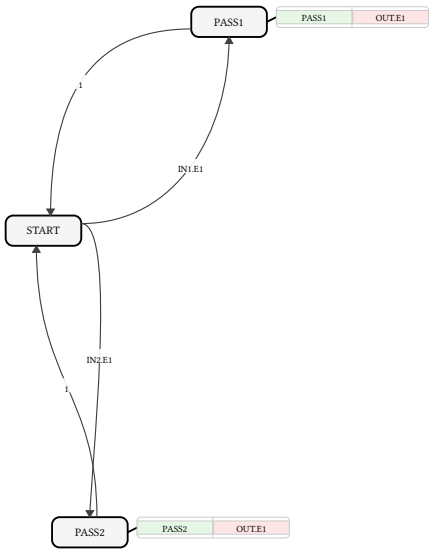
Conclusion

The `AIS_D_FF_TMIN` offers a robust way to latch digital signals with an adjustable minimum output delay. Thanks to the use of AIS adapters and internal IEC 61499 standard components, it is both flexible and reliable. It is particularly suitable for automation tasks where rapid signal changes need to be suppressed or defined minimum pauses between output events must be maintained.

AIS_LAST_2



ECC: AIS_LAST_2



Introduction

The AIS_LAST_2 function block merges two same-type STRING adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: STRING adapter (unidirectional) - first signal source
- **IN2**: STRING adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: STRING adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

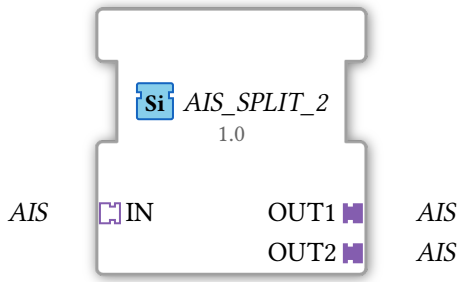
Application Scenarios

- Merging two equally-valid STRING signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same STRING value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AIS_LAST_2 is the inverse of `AIS_SPLIT_2`: AIS_SPLIT_2 distributes 1 input to 2 outputs, AIS_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

AIS_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_2** is a generic adapter module that splits an incoming unidirectional AIS adapter (Application Interface Socket) into two identical outgoing AIS adapters. It serves as a passive splitter without its own logic or event processing.

Interface Structure

Event Inputs

Not present.

Event Outputs

Not present.

Data Inputs

Not present.

Data Outputs

Not present.

Adapter

- **IN** (Socket): Type `adapter::types::unidirectional::AIS` – Input interface for the adapter to be distributed.
- **OUT1** (Plug): Type `adapter::types::unidirectional::AIS` – first output, receives all data/events from IN.
- **OUT2** (Plug): Type `adapter::types::unidirectional::AIS` – second output, receives identical data/events as OUT1.

Functionality

The function block forwards the adapter connected to socket **IN** unchanged and without delay to both plugs **OUT1** and **OUT2**. All events and data defined via the AIS adapter are copied to both outputs in parallel. The function block itself has no processing logic, states, or timing behavior.

Technical Features

- **Generic Parameterization**: The specific AIS adapter type is only defined during project configuration via the attribute `eclipse4diac::core::GenericClassName` (default value: `'GEN_AIS_SPLIT'`). This allows the function block to be reused for different AIS variants.
- **No Eigenstates**: Since there is no ECC state machine or data/event inputs/outputs, the function block is purely passive and behaves like a "wiring aid."
- **Unidirectional**: The adapters are declared as unidirectional – a data flow direction from IN to OUT1/OUT2 is implicit.

State Overview

The AIS_SPLIT_2 does not have its own state machine. The operating state is determined exclusively by the connected AIS adapters.

Application Scenarios

- **Parallel Processing**: An AIS signal is to be processed simultaneously by two subsequent function blocks (e.g., visualization and control).
- **Monitoring**: Branching off an AIS signal to a monitoring instance without affecting the original signal.
- **Generic Splitting**: Used in libraries where the exact AIS structure is only defined at runtime or configuration time.

Comparison with Similar Function Blocks

- **AIS_SPLIT_4**: Offers four outputs instead of two; otherwise, identical functionality.
- **Event/Data Splitter**: Other splitters work with simple events or data (e.g., F_SPLIT), while AIS_SPLIT_2 splits complete adapter interfaces.
- **AIS_MERGE**: The counterpart that combines two AIS inputs into one output.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AIS_SPLIT_2 is a minimalist, generically applicable adapter splitter that enables the flexible branching of AIS interfaces in 4diac projects. By configuring it via the GenericClassName attribute, it can be used for different AIS types without modifying the source code.

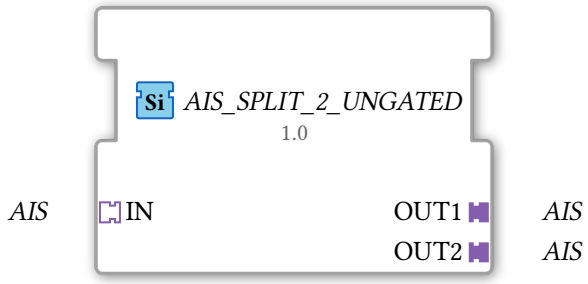
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIS_SPLIT_2_UNGATED

i UNGATED variant: This block is the ungated version of `AIS_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_2_UNGATED** is a generic adapter module that splits an incoming unidirectional AIS adapter (Application Interface Socket) into two identical outgoing AIS adapters. It serves as a passive splitter without its own logic or event processing.

Interface Structure

Event Inputs

Not present.

Event Outputs

Not present.

Data Inputs

Not present.

Data Outputs

Not present.

Adapter

- **IN** (Socket): Type `adapter::types::unidirectional::AIS` – Input interface for the adapter to be distributed.
- **OUT1** (Plug): Type `adapter::types::unidirectional::AIS` – first output, receives all data/events from IN.
- **OUT2** (Plug): Type `adapter::types::unidirectional::AIS` – second output, receives identical data/events as OUT1.

Functionality

The function block forwards the adapter connected to socket **IN** unchanged and without delay to both plugs **OUT1** and **OUT2**. All events and data defined via the AIS adapter are copied to both outputs in parallel. The function block itself has no processing logic, states, or timing behavior.

Technical Features

- **Generic Parameterization**: The specific AIS adapter type is only defined during project configuration via the attribute `eclipse4diac::core::GenericClassName` (default value: `'GEN_AIS_SPLIT'`). This allows the function block to be reused for different AIS variants.
- **No Eigenstates**: Since there is no ECC state machine or data/event inputs/outputs, the function block is purely passive and behaves like a “wiring aid.”

- **Unidirectional**: The adapters are declared as unidirectional – a data flow direction from IN to OUT1/ OUT2 is implicit.

State Overview

The AIS_SPLIT_2_UNGATED does not have its own state machine. The operating state is determined exclusively by the connected AIS adapters.

Application Scenarios

- **Parallel Processing**: An AIS signal is to be processed simultaneously by two subsequent function blocks (e.g., visualization and control).
- **Monitoring**: Branching off an AIS signal to a monitoring instance without affecting the original signal.
- **Generic Splitting**: Used in libraries where the exact AIS structure is only defined at runtime or configuration time.

Comparison with Similar Function Blocks

- **AIS_SPLIT_4**: Offers four outputs instead of two; otherwise, identical functionality.
- **Event/Data Splitter**: Other splitters work with simple events or data (e.g., F_SPLIT), while AIS_SPLIT_2_UNGATED splits complete adapter interfaces.
- **AIS_MERGE**: The counterpart that combines two AIS inputs into one output.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

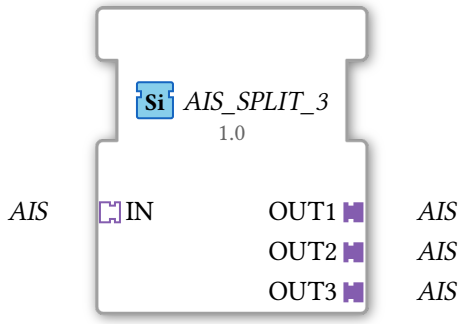
The AIS_SPLIT_2_UNGATED is a minimalist, generically applicable adapter splitter that enables the flexible branching of AIS interfaces in 4diac projects. By configuring it via the GenericClassName attribute, it can be used for different AIS types without modifying the source code.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIS_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_3** is used to distribute an incoming AIS data stream to three identical outputs. It is implemented as a generic function block and enables the simple duplication of AIS information in control systems according to IEC 61499.

Interface Structure

Event Inputs

Not present – the function block operates purely data- and adapter-driven.

Event Outputs

Not present.

Data Inputs

Not present – input data is provided exclusively via the adapter socket.

Data Outputs

Not present – output data is provided exclusively via the adapter plugs.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AIS`	IN	Socket (Input)	Receives an AIS data stream.
`adapter::types::unidirectional::AIS`	OUT1	Plug (Output)	First output channel – identical copy of the input.
`adapter::types::unidirectional::AIS`	OUT2	Plug (Output)	Second output channel – identical copy of the input.
`adapter::types::unidirectional::AIS`	OUT3	Plug (Output)	Third output channel – identical copy of the input.

Functionality

The module receives an AIS data stream via the **IN** socket and forwards it without delay or manipulation to the three output adapters **OUT1**, **OUT2**, and **OUT3**. Each output receives identical data. No buffering, filtering, or protocol conversion takes place – the module functions purely as a splitter at the adapter level.

Technical Features

- **Generic Type**: The module is assigned the generic class name `GEN_AIS_SPLIT`, allowing it to be reused in various contexts and with different AIS data types.
- **Unidirectional Adapters**: Both inputs and outputs use the unidirectional adapter `adapter::types::unidirectional::AIS`, ensuring a clear data flow direction.
- **No Event-Driven Synchronization**: Since there are no event inputs/outputs, data transmission occurs immediately and without triggering by external events.

State Overview

This function block does not have an explicit state machine. Its behavior is that of a continuously active pass-through connection: As long as the input adapter is supplying data, it is output on all three outputs. There are no internal states or modes.

Application Scenarios

- **Distributing AIS data to multiple consumers** – e.g., simultaneous visualization, logging, and further processing.
- **Redundant data provision** – feeding data into different subsystems without placing an additional load on the data source.
- **Test and diagnostic setups** – parallel monitoring of an AIS stream at multiple points.

Comparison with similar function blocks

Function block	Number of outputs	Special feature
AIS_SPLIT_2	2	Distributed across two channels.
AIS_SPLIT_3	**3**	**Standard splitter with three outputs.**
AIS_SPLIT_N	variable	Generic version with configurable number of outputs (where available).

The AIS_SPLIT_3 fills the gap between a simple 2-way splitter and a fully configurable splitter.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AIS_SPLIT_3** is a simple yet effective component for multiplying an AIS data stream. Thanks to its generic nature and clear, adapter-based interface, it can be easily integrated into larger 4diac projects. It offers a robust and maintainable solution for applications requiring multiple distributions without data modification.

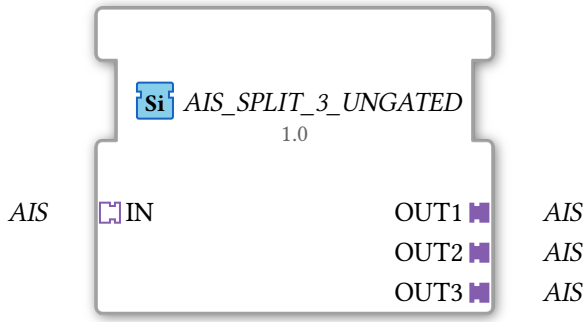
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIS_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_3_UNGATED** is used to distribute an incoming AIS data stream to three identical outputs. It is implemented as a generic function block and enables the simple duplication of AIS information in control systems according to IEC 61499.

Interface Structure

Event Inputs

Not present – the function block operates purely data- and adapter-driven.

Event Outputs

Not present.

Data Inputs

Not present – input data is provided exclusively via the adapter socket.

Data Outputs

Not present – output data is provided exclusively via the adapter plugs.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AIS`	IN	Socket (Input)	Receives an AIS data stream.
`adapter::types::unidirectional::AIS`	OUT1	Plug (Output)	First output channel – identical copy of the input.
`adapter::types::unidirectional::AIS`	OUT2	Plug (Output)	Second output channel – identical copy of the input.
`adapter::types::unidirectional::AIS`	OUT3	Plug (Output)	Third output channel – identical copy of the input.

Functionality

The module receives an AIS data stream via the **IN** socket and forwards it without delay or manipulation to the three output adapters **OUT1**, **OUT2**, and **OUT3**. Each output receives identical data. No buffering, filtering, or protocol conversion takes place – the module functions purely as a splitter at the adapter level.

Technical Features

- **Generic Type**: The module is assigned the generic class name `GEN_AIS_SPLIT`, allowing it to be reused in various contexts and with different AIS data types.
- **Unidirectional Adapters**: Both inputs and outputs use the unidirectional adapter `adapter::types::unidirectional::AIS`, ensuring a clear data flow direction.
- **No Event-Driven Synchronization**: Since there are no event inputs/outputs, data transmission occurs immediately and without triggering by external events.

State Overview

This function block does not have an explicit state machine. Its behavior is that of a continuously active pass-through connection: As long as the input adapter is supplying data, it is output on all three outputs. There are no internal states or modes.

Application Scenarios

- **Distributing AIS data to multiple consumers** – e.g., simultaneous visualization, logging, and further processing.
- **Redundant data provision** – feeding data into different subsystems without placing an additional load on the data source.
- **Test and diagnostic setups** – parallel monitoring of an AIS stream at multiple points.

Comparison with similar function blocks

Function block	Number of outputs	Special feature
AIS_SPLIT_2	2	Distributed across two channels.
AIS_SPLIT_3_UNGATED	**3**	**Standard splitter with three outputs.**
AIS_SPLIT_N	variable	Generic version with configurable number of outputs (where available).

The AIS_SPLIT_3_UNGATED fills the gap between a simple 2-way splitter and a fully configurable splitter.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

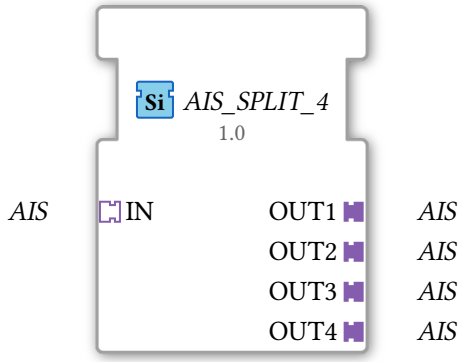
The **AIS_SPLIT_3_UNGATED** is a simple yet effective component for multiplying an AIS data stream. Thanks to its generic nature and clear, adapter-based interface, it can be easily integrated into larger 4diac projects. It offers a robust and maintainable solution for applications requiring multiple distributions without data modification.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIS_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_4** is used to distribute an incoming AIS adapter signal (type `adapter::types::unidirectional::AIS`) to four separate outputs. It implements a 1-to-4 split without modifying the data. The block is implemented generically and can be adapted to different signal types via the generic class name.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

- **Socket (Input):** `IN` – Type `adapter::types::unidirectional::AIS`
- **Plugs (Outputs):**
- `OUT1` – Type `adapter::types::unidirectional::AIS`
- `OUT2` – Type `adapter::types::unidirectional::AIS`
- `OUT3` – Type `adapter::types::unidirectional::AIS`
- `OUT4` – Type `adapter::types::unidirectional::AIS`

Functionality

The module forwards the adapter signal present at socket `IN` unchanged to all four output adapters. (`OUT1` … `OUT4`). No data manipulation, filtering, or buffering takes place. Each output corresponds exactly to the input signal. Distribution is asynchronous and without event control – the adapter interface handles data transmission according to its communication logic.

Technical Features

- **Generic Type:** The function block is implemented as a generic function block. The actual signal type can be defined at creation time via the attribute `eclipse4diac::core::GenericClassName` (value `'GEN_AIS_SPLIT'`).
- **Type Hash:** An empty string (`''`) for the attribute `eclipse4diac::core::TypeHash` indicates that the function block is dynamically configured as needed.
- **Compactness:** Since there are neither event nor data inputs/outputs, the module is purely adapter-based and free of timing dependencies.

State Overview

The module does not have a state machine – it operates continuously and without a state. There are no internal states or events that need to be switched. Its functionality is limited to passively passing through the adapter signal.

Application Scenarios

- **Signal Distribution in Control Architectures:** An AIS signal provided by a source (e.g., a sensor) is to be made available to several downstream processing units or displays.
- **Redundant Data Transmission:** In safety-critical systems, the input signal can be routed to multiple independent paths without duplicating it.
- **Test and Debugging Environments:** A signal is routed in parallel to the actual logic and to an analysis/logging tool.

Comparison with Similar Function Blocks

- **AIS_SPLIT_2 / AIS_SPLIT_8:** These function blocks represent variants with two and eight outputs, respectively. Their functionality is identical; only the number of output adapters differs.
- **Data-Based Splitters (e.g., SPLIT_INT):** Unlike adapter-based splitters, data-based splitters work with specific data types (e.g., integers) and usually require events for triggering. The AIS_SPLIT_4 is a pure adapter splitter and does not require explicit triggering.
- **Adapter Multiplexers (e.g., AIS_MUX):** A multiplexer selects one input from several, while the splitter distributes one input to multiple outputs.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AIS_SPLIT_4** is a simple yet practical function block for reliably distributing an AIS signal to four identical outputs. Its generic design and pure adapter logic make it flexible and eliminate the need for additional event control. It is ideally suited for applications where a signal is needed multiple times without data processing or synchronization.

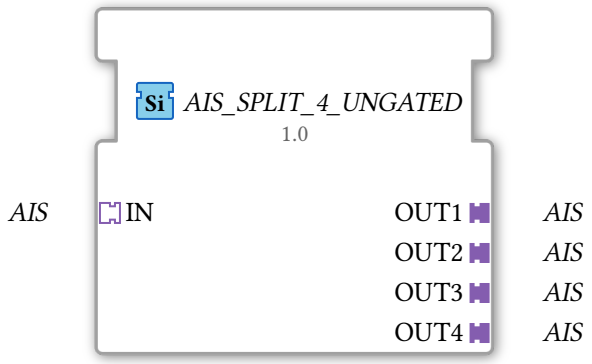
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIS_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_4_UNGATED** is used to distribute an incoming AIS adapter signal (type `adapter::types::unidirectional::AIS`) to four separate outputs. It implements a 1-to-4 split without modifying the data. The block is implemented generically and can be adapted to different signal types via the generic class name.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

- **Socket (Input):** `IN` – Type `adapter::types::unidirectional::AIS`
- **Plugs (Outputs):**
- `OUT1` – Type `adapter::types::unidirectional::AIS`
- `OUT2` – Type `adapter::types::unidirectional::AIS`
- `OUT3` – Type `adapter::types::unidirectional::AIS`
- `OUT4` – Type `adapter::types::unidirectional::AIS`

Functionality

The module forwards the adapter signal present at socket `IN` unchanged to all four output adapters. (`OUT1` … `OUT4`). No data manipulation, filtering, or buffering takes place. Each output corresponds exactly to the input signal. Distribution is asynchronous and without event control – the adapter interface handles data transmission according to its communication logic.

Technical Features

- **Generic Type:** The function block is implemented as a generic function block. The actual signal type can be defined at creation time via the attribute `eclipse4diac::core::GenericClassName` (value `'GEN_AIS_SPLIT'`).
- **Type Hash:** An empty string (`''`) for the attribute `eclipse4diac::core::TypeHash` indicates that the function block is dynamically configured as needed.
- **Compactness:** Since there are neither event nor data inputs/outputs, the module is purely adapter-based and free of timing dependencies.

State Overview

The module does not have a state machine – it operates continuously and without a state. There are no internal states or events that need to be switched. Its functionality is limited to passively passing through the adapter signal.

Application Scenarios

- **Signal Distribution in Control Architectures:** An AIS signal provided by a source (e.g., a sensor) is to be made available to several downstream processing units or displays.
- **Redundant Data Transmission:** In safety-critical systems, the input signal can be routed to multiple independent paths without duplicating it.
- **Test and Debugging Environments:** A signal is routed in parallel to the actual logic and to an analysis/logging tool.

Comparison with Similar Function Blocks

- **AIS_SPLIT_2 / AIS_SPLIT_8:** These function blocks represent variants with two and eight outputs, respectively. Their functionality is identical; only the number of output adapters differs.
- **Data-Based Splitters (e.g., SPLIT_INT):** Unlike adapter-based splitters, data-based splitters work with specific data types (e.g., integers) and usually require events for triggering. The AIS_SPLIT_4_UNGATED is a pure adapter splitter and does not require explicit triggering.
- **Adapter Multiplexers (e.g., AIS_MUX):** A multiplexer selects one input from several, while the splitter distributes one input to multiple outputs.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

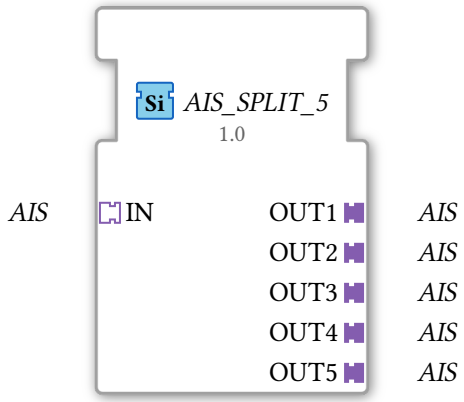
The **AIS_SPLIT_4_UNGATED** is a simple yet practical function block for reliably distributing an AIS signal to four identical outputs. Its generic design and pure adapter logic make it flexible and eliminate the need for additional event control. It is ideally suited for applications where a signal is needed multiple times without data processing or synchronization.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIS_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_5** serves as a generic splitter for adapters of type `AIS` (unidirectional). It receives an incoming AIS signal via a socket and forwards it to five separate AIS plugs. This allows a single adapter signal to be distributed to multiple target blocks.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapter

Direction	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::AIS	Incoming AIS adapter signal
Plug (Output 1)	OUT1	adapter::types::unidirectional::AIS	First outgoing AIS adapter port
Plug (Output 2)	OUT2	adapter::types::unidirectional::AIS	Second outgoing AIS adapter port
Plug (Output 3)	OUT3	adapter::types::unidirectional::AIS	Third outgoing AIS adapter port
Plug (Output 4)	OUT4	adapter::types::unidirectional::AIS	Fourth Outgoing AIS Adapter Port
Plug (Output 5)	OUT5	adapter::types::unidirectional::AIS	Fifth Outgoing AIS Adapter Port

Functionality
This function block operates as a pure broadcast without its own logic or states. As soon as an event or data change arrives via socket `IN`, this signal is passed unchanged to all five plugs (`OUT1`–`OUT5`). Each plug can be connected to a different receiver function block, allowing the original signal to be processed simultaneously by multiple components.

Technical Features
- **Generic Function Block**: The function block is marked as a generic function block (`eclipse4diac::core::GenericClassName = 'GEN_AIS_SPLIT'`). This allows it to be instantiated in various contexts with appropriate AIS adapter types.
- **No Event or Data Inputs/Outputs**: All communication takes place exclusively via the adapter interfaces. This makes it particularly simple and efficient.
- **Unidirectional**: The adapter type `AIS` is unidirectional, meaning the signal direction is fixed (from the socket to the plugs). Feedback is not provided.

State Overview
The module has no internal state logic (no ECC). There is no internal state machine or delay. Its operation is based on a rigid, combinational routing.

Application Scenarios
- **Signal Distribution in Control Systems**: An AIS signal supplied by a source (e.g., sensor or communication module) is to be passed on to several independent processing blocks.
- **Test and Simulation Environments**: A test signal can be sent simultaneously to various monitoring or analysis modules.
- **Redundant Processing**: The same signal is evaluated multiple times to detect errors or compare different algorithms.

Comparison with Similar Components
Other splitter components, such as `AIS_SPLIT_2`, `AIS_SPLIT_3`, and `AIS_SPLIT_4`, offer fewer outputs. All of them distribute a single AIS input signal to multiple outputs. The `AIS_SPLIT_5` offers the maximum

of five outputs in this family. The appropriate splitter can be selected based on requirements to avoid unnecessary cabling.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AIS_SPLIT_5` is a simple yet effective generic function block for multiplying a unidirectional AIS adapter signal. It reduces cabling effort in the 4diac IDE and enables clean, modular structuring of control applications. It represents an optimal solution for applications requiring the distribution of a signal to up to five receivers.

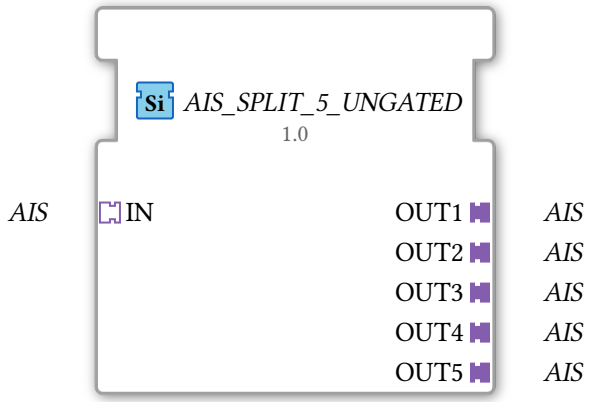
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIS_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_5_UNGATED** serves as a generic splitter for adapters of type `AIS` (unidirectional). It receives an incoming AIS signal via a socket and forwards it to five separate AIS plugs. This allows a single adapter signal to be distributed to multiple target blocks.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::AIS	Incoming AIS adapter signal
Plug (Output 1)	OUT1	adapter::types::unidirectional::AIS	First outgoing AIS adapter port
Plug (Output 2)	OUT2	adapter::types::unidirectional::AIS	Second outgoing AIS adapter port
Plug (Output 3)	OUT3	adapter::types::unidirectional::AIS	Third outgoing AIS adapter port
Plug (Output 4)	OUT4	adapter::types::unidirectional::AIS	Fourth Outgoing AIS Adapter Port
Plug (Output 5)	OUT5	adapter::types::unidirectional::AIS	Fifth Outgoing AIS Adapter Port

Functionality

This function block operates as a pure broadcast without its own logic or states. As soon as an event or data change arrives via socket `IN`, this signal is passed unchanged to all five plugs (`OUT1`–`OUT5`). Each plug can be connected to a different receiver function block, allowing the original signal to be processed simultaneously by multiple components.

Technical Features

- **Generic Function Block**: The function block is marked as a generic function block (`eclipse4diac::core::GenericClassName = 'GEN_AIS_SPLIT'`). This allows it to be instantiated in various contexts with appropriate AIS adapter types.
- **No Event or Data Inputs/Outputs**: All communication takes place exclusively via the adapter interfaces. This makes it particularly simple and efficient.
- **Unidirectional**: The adapter type `AIS` is unidirectional, meaning the signal direction is fixed (from the socket to the plugs). Feedback is not provided.

State Overview

The module has no internal state logic (no ECC). There is no internal state machine or delay. Its operation is based on a rigid, combinational routing.

Application Scenarios

- **Signal Distribution in Control Systems**: An AIS signal supplied by a source (e.g., sensor or communication module) is to be passed on to several independent processing blocks.
- **Test and Simulation Environments**: A test signal can be sent simultaneously to various monitoring or analysis modules.
- **Redundant Processing**: The same signal is evaluated multiple times to detect errors or compare different algorithms.

Comparison with Similar Components

Other splitter components, such as `AIS_SPLIT_2`, `AIS_SPLIT_3`, and `AIS_SPLIT_4`, offer fewer outputs. All of them distribute a single AIS input signal to multiple outputs. The `AIS_SPLIT_5_UNGATED` offers the maximum of five outputs in this family. The appropriate splitter can be selected based on requirements to avoid unnecessary cabling.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

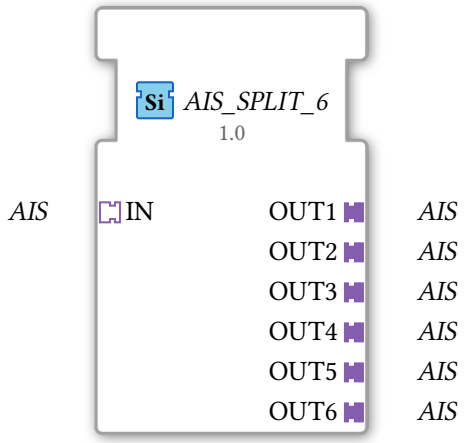
The `AIS_SPLIT_5_UNGATED` is a simple yet effective generic function block for multiplying a unidirectional AIS adapter signal. It reduces cabling effort in the 4diac IDE and enables clean, modular structuring of control applications. It represents an optimal solution for applications requiring the distribution of a signal to up to five receivers.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIS_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_6** is used to distribute an incoming AIS (Automation Interface Signal) signal to six separate AIS outputs. It acts as a generic splitter, forwarding the incoming signal to all connected outputs without delay or manipulation.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available (data transmission occurs exclusively via the adapter).

Data Outputs

No data outputs available (data transmission occurs exclusively via the adapter).

Adapter

Type	Direction	Name	Description
adapter::types::unidirectional::AIS	Input (Socket)	IN	Incoming AIS signal, which is distributed to all outputs.
adapter::types::unidirectional::AIS	Output (Plug)	OUT1	First output with the split AIS signal.
adapter::types::unidirectional::AIS	Output (Plug)	OUT2	Second output with the split AIS signal.
adapter::types::unidirectional::AIS	Output (Plug)	OUT3	Third output with the split AIS signal.
adapter::types::unidirectional::AIS	Output (Plug)	OUT4	Fourth output with the split AIS signal.
adapter::types::unidirectional::AIS	Output (Plug)	OUT5	Fifth output with the split AIS signal.

adapter::types::unidirectional::AIS	Output (Plug)	OUT6	Sixth output with the split AIS signal.

Functionality

The module receives an AIS signal via socket `IN` and forwards it simultaneously and identically to all six plugs `OUT1` to `OUT6`. No data processing, filtering, or intermediate storage takes place – the distribution is purely signal-based. The function block operates unidirectionally (only from the input to the outputs) and thus represents a simple 1:6 multiplication of the AIS signal.

Technical Features

- **Generic Type**: The function block is implemented as a generic function block (`GEN_AIS_SPLIT`), meaning it can be used with various AIS types as long as the adapter definition (`adapter::types::unidirectional::AIS`) matches.
- **Unidirectional**: Data exchange occurs only in one direction – from the input to the outputs. Feedback from the outputs to the input is not supported.
- **No Internal Logic**: Since there are neither events nor data inputs/outputs, the function block operates purely passively and requires no state machine or algorithms.
- **Easy Scalability**: Due to the fixed number of six outputs, the function block is optimized for typical applications requiring exactly six parallel AIS connections.

State Overview

This function block does not have its own state machine (ECC). Its behavior is deterministic and timeless: An incoming AIS signal is immediately passed on to all outputs. There are no internal states or steps.

Application Scenarios

- **Distributing Control Signals**: An AIS signal from a central controller (e.g., a higher-level function block) must be forwarded to several subordinate components.
- **Monitoring and Parallel Operation**: A signal should be used simultaneously for control, monitoring, and logging.
- **Test Setups**: A uniform test signal is distributed to several simulated or real modules.
- **Redundant Wiring**: Signal distribution via multiple paths to increase fault tolerance.

Comparison with Similar Function Blocks

Other split function blocks may exist in the 4diac framework that differ in the number of outputs or data type. `AIS_SPLIT_6` is specifically designed for the AIS adapter interface and six outputs. Other splitters might have additional events or configuration parameters (e.g., selective routing), while this module is intentionally simple and requires no configuration.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AIS_SPLIT_6` is a useful and minimalist module for signal splitters in AIS-based control systems. It reliably distributes an incoming signal to six outputs and is characterized by its simplicity, generic design, and clear interface separation. This makes it particularly suitable for modular, transparent control architectures.

–

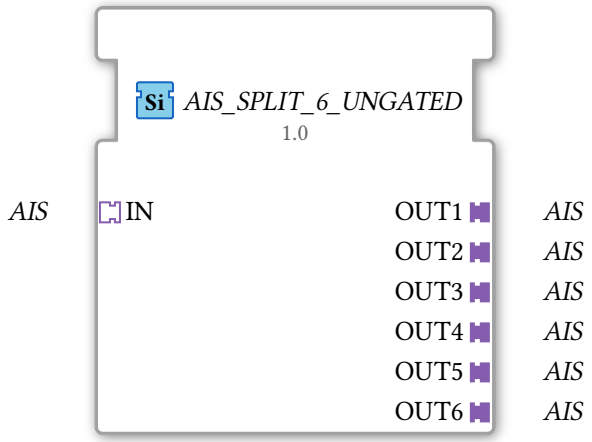
🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AIS_SPLIT_6_UNGATED

i UNGATED variant: This block is the ungated version of `AIS_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_6_UNGATED** is used to distribute an incoming AIS (Automation Interface Signal) signal to six separate AIS outputs. It acts as a generic splitter, forwarding the incoming signal to all connected outputs without delay or manipulation.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available (data transmission occurs exclusively via the adapter).

Data Outputs

No data outputs available (data transmission occurs exclusively via the adapter).

Adapter

Type	Direction	Name	Description
`adapter::types::unidirectional::AIS`	Input (Socket)	IN	Incoming AIS signal, which is distributed to all outputs.
`adapter::types::unidirectional::AIS`	Output (Plug)	OUT1	First output with the split AIS signal.
`adapter::types::unidirectional::AIS`	Output (Plug)	OUT2	Second output with the split AIS signal.
`adapter::types::unidirectional::AIS`	Output (Plug)	OUT3	Third output with the split AIS signal.

adapter::types::unidirectional::AIS	Output (Plug)	OUT4	Fourth output with the split AIS signal.
adapter::types::unidirectional::AIS	Output (Plug)	OUT5	Fifth output with the split AIS signal.
adapter::types::unidirectional::AIS	Output (Plug)	OUT6	Sixth output with the split AIS signal.

Functionality

The module receives an AIS signal via socket `IN` and forwards it simultaneously and identically to all six plugs `OUT1` to `OUT6`. No data processing, filtering, or intermediate storage takes place – the distribution is purely signal-based. The function block operates unidirectionally (only from the input to the outputs) and thus represents a simple 1:6 multiplication of the AIS signal.

Technical Features

- **Generic Type**: The function block is implemented as a generic function block (`GEN_AIS_SPLIT`), meaning it can be used with various AIS types as long as the adapter definition (`adapter::types::unidirectional::AIS`) matches.
- **Unidirectional**: Data exchange occurs only in one direction – from the input to the outputs. Feedback from the outputs to the input is not supported.
- **No Internal Logic**: Since there are neither events nor data inputs/outputs, the function block operates purely passively and requires no state machine or algorithms.
- **Easy Scalability**: Due to the fixed number of six outputs, the function block is optimized for typical applications requiring exactly six parallel AIS connections.

State Overview

This function block does not have its own state machine (ECC). Its behavior is deterministic and timeless: An incoming AIS signal is immediately passed on to all outputs. There are no internal states or steps.

Application Scenarios

- **Distributing Control Signals**: An AIS signal from a central controller (e.g., a higher-level function block) must be forwarded to several subordinate components.
- **Monitoring and Parallel Operation**: A signal should be used simultaneously for control, monitoring, and logging.
- **Test Setups**: A uniform test signal is distributed to several simulated or real modules.
- **Redundant Wiring**: Signal distribution via multiple paths to increase fault tolerance.

Comparison with Similar Function Blocks

Other split function blocks may exist in the 4diac framework that differ in the number of outputs or data type. `AIS_SPLIT_6_UNGATED` is specifically designed for the AIS adapter interface and six outputs. Other splitters might have additional events or configuration parameters (e.g., selective routing), while this module is intentionally simple and requires no configuration.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AIS_SPLIT_6_UNGATED` is a useful and minimalist module for signal splitters in AIS-based control systems. It reliably distributes an incoming signal to six outputs and is characterized by its simplicity, generic design, and clear interface separation. This makes it particularly suitable for modular, transparent control architectures.

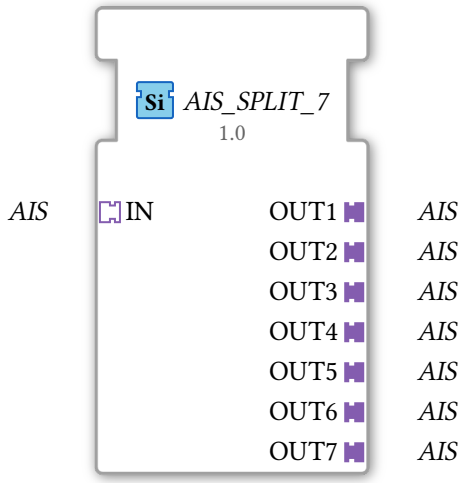
–

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AIS_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_7** is used to distribute an incoming AIS data stream to seven identical outputs. It is implemented as a generic block and allows the division of a unidirectional AIS adapter (Socket IN) into seven identical adapter plugs (OUT1 ... OUT7). The term "AIS" here refers to an application-specific, abstract data type in the adapter context of IEC 61499.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	IN	`adapter::types::unidirectional::AIS`	Incoming AIS data stream, distributed to all outputs.
Plug (Output)	OUT1 ... OUT7	`adapter::types::unidirectional::AIS`	Seven identical outputs, each forwarding the incoming AIS data stream unchanged.

Functionality

This module functions as a splitter for AIS adapter connections. Every incoming data or event at socket `IN` is replicated **without delay or processing** to all seven adapter plugs OUT1 to `OUT7`. The data is forwarded in parallel, meaning all outputs receive the same information simultaneously. No internal logic or state machine is required, as this is a passive, connection-oriented data transfer.

Technical Features

- **Generic Function Block** – The function block is declared as a generic instance (`GEN_AIS_SPLIT`) and can be parameterized for any AIS type.

- **Unidirectional Adapters** – Both inputs and outputs use only unidirectional adapters, ensuring a pure flow from input to output.
- **No Event or Data Inputs/Outputs** – All communication occurs exclusively via adapters, enabling clean encapsulation and reuse.
- **Scalability** – The principle can easily be adapted to different numbers of outputs (e.g., AIS_SPLIT_2, AIS_SPLIT_8).

State Overview

This function block does not have an explicit state machine (ECC) because it does not execute any sequential logic or delayed operations. Its function is reduced to static, continuous forwarding. Therefore, an operating state is not defined.

Application Scenarios

- **Distribution of AIS signals** to multiple parallel processing units, e.g., in automation systems, when a sensor value needs to be simultaneously passed to several control loops, displays, or analysis tools.
- **Redundant Signal Forwarding** – Used to feed an AIS data stream into multiple independent paths for testing or monitoring purposes.
- **Modular System Architecture** – As a component in a library of generic adapter function blocks to increase the reuse and interchangeability of connections.

Comparison with Similar Function Blocks

- **Data Splitters (e.g., SPLIT_1_to_N)** – These typically split simple data types (BOOL, INT, REAL). The AIS_SPLIT_7 is specifically designed for adapter interfaces and utilizes the adapter concept defined in IEC 61499 for complex, structured data streams.
- **Adapter Multiplexers (e.g., AIS_MERGE)** – While a splitter distributes one input to multiple outputs, a merger combines multiple inputs into one output. The AIS_SPLIT_7 is the counterpart to this.
- **Event Splitters** – These differ in their event processing capabilities; the AIS_SPLIT_7, however, operates purely on an adapter basis without separate event channels.

Change Detection

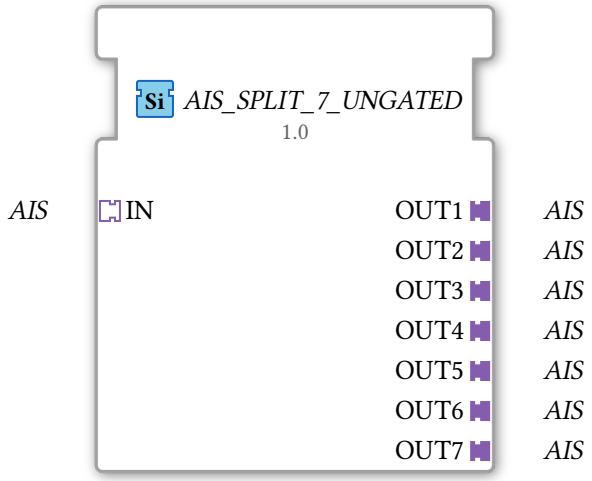
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AIS_SPLIT_7** is a simple yet effective functional block for distributing a unidirectional AIS data stream to seven parallel outputs. Its generic design and exclusive use of adapters make it a flexible tool in automation application development according to IEC 61499. The absence of state logic and event control makes it lightweight and ideally suited for pure distribution tasks.

AIS_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_7_UNGATED** is used to distribute an incoming AIS data stream to seven identical outputs. It is implemented as a generic block and allows the division of a unidirectional AIS adapter (Socket IN) into seven identical adapter plugs (OUT1 ... OUT7). The term “AIS” here refers to an application-specific, abstract data type in the adapter context of IEC 61499.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	IN	`adapter::types::unidirectional::AIS`	Incoming AIS data stream, distributed to all outputs.
Plug (Output)	OUT1 ... OUT7	`adapter::types::unidirectional::AIS`	Seven identical outputs, each forwarding the incoming AIS data stream unchanged.

Functionality

This module functions as a splitter for AIS adapter connections. Every incoming data or event at socket `IN` is replicated **without delay or processing** to all seven adapter plugs OUT1 to `OUT7`. The data is forwarded in parallel, meaning all outputs receive the same information simultaneously. No internal logic or state machine is required, as this is a passive, connection-oriented data transfer.

Technical Features

- **Generic Function Block** – The function block is declared as a generic instance (`GEN_AIS_SPLIT`) and can be parameterized for any AIS type.

- **Unidirectional Adapters** – Both inputs and outputs use only unidirectional adapters, ensuring a pure flow from input to output.
- **No Event or Data Inputs/Outputs** – All communication occurs exclusively via adapters, enabling clean encapsulation and reuse.
- **Scalability** – The principle can easily be adapted to different numbers of outputs (e.g., AIS_SPLIT_2, AIS_SPLIT_8).

State Overview

This function block does not have an explicit state machine (ECC) because it does not execute any sequential logic or delayed operations. Its function is reduced to static, continuous forwarding. Therefore, an operating state is not defined.

Application Scenarios

- **Distribution of AIS signals** to multiple parallel processing units, e.g., in automation systems, when a sensor value needs to be simultaneously passed to several control loops, displays, or analysis tools.
- **Redundant Signal Forwarding** – Used to feed an AIS data stream into multiple independent paths for testing or monitoring purposes.
- **Modular System Architecture** – As a component in a library of generic adapter function blocks to increase the reuse and interchangeability of connections.

Comparison with Similar Function Blocks

- **Data Splitters (e.g., SPLIT_1_to_N)** – These typically split simple data types (BOOL, INT, REAL). The AIS_SPLIT_7_UNGATED is specifically designed for adapter interfaces and utilizes the adapter concept defined in IEC 61499 for complex, structured data streams.
- **Adapter Multiplexers (e.g., AIS_MERGE)** – While a splitter distributes one input to multiple outputs, a merger combines multiple inputs into one output. The AIS_SPLIT_7_UNGATED is the counterpart to this.
- **Event Splitters** – These differ in their event processing capabilities; the AIS_SPLIT_7_UNGATED, however, operates purely on an adapter basis without separate event channels.

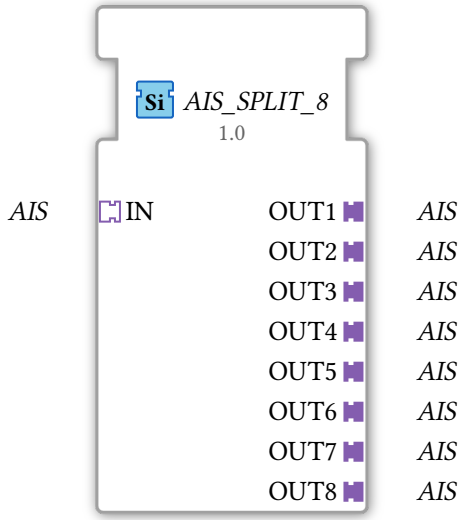
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AIS_SPLIT_7_UNGATED** is a simple yet effective functional block for distributing a unidirectional AIS data stream to seven parallel outputs. Its generic design and exclusive use of adapters make it a flexible tool in automation application development according to IEC 61499. The absence of state logic and event control makes it lightweight and ideally suited for pure distribution tasks.

AIS_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_8** distributes an incoming AIS (Application Interface Socket) signal to eight identical output adapters. It is designed as a generic block and allows the simultaneous transmission of an AIS signal to multiple subsequent function blocks without data loss or delay.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapters

Direction	Name	Type	Description
Input	IN	`adapter::types::unidirectional::AIS`	Incoming AIS signal to be distributed.
Output	OUT1	`adapter::types::unidirectional::AIS`	First output channel (identical to IN).
Output	OUT2	`adapter::types::unidirectional::AIS`	Second output channel (identical to IN).
Output	OUT3	`adapter::types::unidirectional::AIS`	Third output channel (identical to IN).
Output	OUT4	`adapter::types::unidirectional::AIS`	Fourth output channel (identical to IN).
Output	OUT5	`adapter::types::unidirectional::AIS`	Fifth output channel (identical to IN).
Output	OUT6	`adapter::types::unidirectional::AIS`	Sixth output channel (identical to IN).
Output	OUT7	`adapter::types::unidirectional::AIS`	Seventh output channel (identical to IN).
Output	OUT8	`adapter::types::unidirectional::AIS`	Eighth output channel (identical to IN).

Functionality

The module forwards the AIS signal present at adapter input `IN` unchanged and without delay to all eight output adapters `OUT1` to `OUT8`. No transformation, filtering, or buffering takes place. This behavior corresponds to a passive distribution (broadcast) of an AIS interface.

Technical Features

- **Generic Type** – The function block is declared as `GEN_AIS_SPLIT` and can be adapted to specific AIS implementations via type parameterization (e.g., for different data types or event structures).
- **No Runtime Logic** – All functionality is implemented solely through the interconnection of the adapters; there are no internal algorithms or state machines.
- **Minimal Resource Requirements** – Since no active processing takes place, the function block incurs neither CPU load nor memory consumption during execution.

State Overview

The function block has no internal states. The output signal is always a direct mapping of the input signal.

Application Scenarios

- **Parallel Processing** – Splitting a sensor signal across multiple independent evaluation blocks.

- **Redundancy** – Distributing a control signal to multiple actuators in safety-critical systems.
- **Test and Simulation Environments** – Providing an identical signal for different observers or loggers.

Comparison with Similar Modules

Unlike modules such as `AIS_MERGE_2` or `AIS_SELECT`, `AIS_SPLIT_8` does not have any selection or prioritization logic. Related variants differ only in the number of outputs (e.g., `AIS_SPLIT_2` or `AIS_SPLIT_4`). `AIS_SPLIT_8` is the maximum standard version for eight channels.

Change Detection

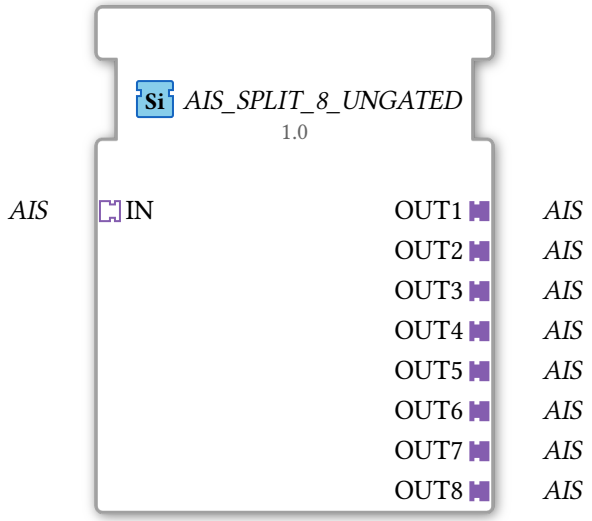
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AIS_SPLIT_8` is a simple and efficient module for replicating an AIS interface. Due to its generic nature and purely adapter-based implementation, it is ideally suited for building modular automation architectures that require distributed signal forwarding.

AIS_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_SPLIT_8_UNGATED** distributes an incoming AIS (Application Interface Socket) signal to eight identical output adapters. It is designed as a generic block and allows the simultaneous transmission of an AIS signal to multiple subsequent function blocks without data loss or delay.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapters

Direction	Name	Type	Description
Input	IN	`adapter::types::unidirectional::AIS`	Incoming AIS signal to be distributed.
Output	OUT1	`adapter::types::unidirectional::AIS`	First output channel (identical to IN).
Output	OUT2	`adapter::types::unidirectional::AIS`	Second output channel (identical to IN).
Output	OUT3	`adapter::types::unidirectional::AIS`	Third output channel (identical to IN).
Output	OUT4	`adapter::types::unidirectional::AIS`	Fourth output channel (identical to IN).
Output	OUT5	`adapter::types::unidirectional::AIS`	Fifth output channel (identical to IN).
Output	OUT6	`adapter::types::unidirectional::AIS`	Sixth output channel (identical to IN).
Output	OUT7	`adapter::types::unidirectional::AIS`	Seventh output channel (identical to IN).
Output	OUT8	`adapter::types::unidirectional::AIS`	Eighth output channel (identical to IN).

Functionality

The module forwards the AIS signal present at adapter input `IN` unchanged and without delay to all eight output adapters `OUT1` to `OUT8`. No transformation, filtering, or buffering takes place. This behavior corresponds to a passive distribution (broadcast) of an AIS interface.

Technical Features

- **Generic Type** – The function block is declared as `GEN_AIS_SPLIT` and can be adapted to specific AIS implementations via type parameterization (e.g., for different data types or event structures).
- **No Runtime Logic** – All functionality is implemented solely through the interconnection of the adapters; there are no internal algorithms or state machines.
- **Minimal Resource Requirements** – Since no active processing takes place, the function block incurs neither CPU load nor memory consumption during execution.

State Overview

The function block has no internal states. The output signal is always a direct mapping of the input signal.

Application Scenarios

- **Parallel Processing** – Splitting a sensor signal across multiple independent evaluation blocks.
- **Redundancy** – Distributing a control signal to multiple actuators in safety-critical systems.
- **Test and Simulation Environments** – Providing an identical signal for different observers or loggers.

Comparison with Similar Modules

Unlike modules such as `AIS_MERGE_2` or `AIS_SELECT`, `AIS_SPLIT_8_UNGATED` does not have any selection or prioritization logic. Related variants differ only in the number of outputs (e.g., `AIS_SPLIT_2` or `AIS_SPLIT_4`). `AIS_SPLIT_8_UNGATED` is the maximum standard version for eight channels.

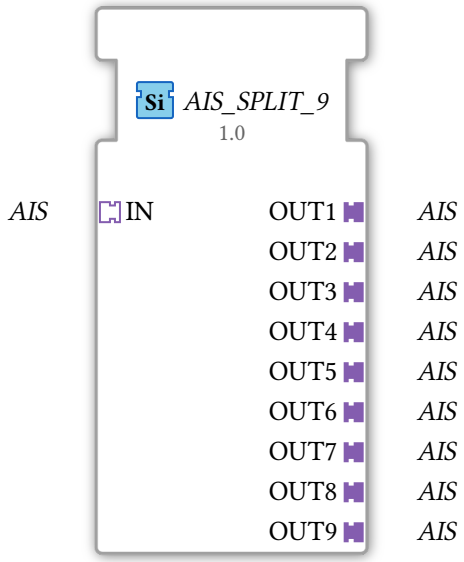
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AIS_SPLIT_8_UNGATED` is a simple and efficient module for replicating an AIS interface. Due to its generic nature and purely adapter-based implementation, it is ideally suited for building modular automation architectures that require distributed signal forwarding.

AIS_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The **AIS_SPLIT_9** function block is used to distribute an incoming unidirectional **AIS** adapter (Application Interface Socket) to nine identical outgoing **AIS** adapters (plugs). It is designed as a generic building block and enables the simple duplication of an AIS signal for up to nine downstream function blocks.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Name	Type	Direction	Description
IN	AIS (unidirectional)	Socket	Input adapter – the AIS signal to be distributed.
OUT1	AIS (unidirectional)	Plug	First output – identical copy of the input signal.
OUT2	AIS (unidirectional)	Plug	Second output – identical copy of the input signal.
OUT3	AIS (unidirectional)	Plug	Third output – identical copy of the input signal.
OUT4	AIS (unidirectional)	Plug	Fourth output – identical copy of the input signal.
OUT5	AIS (unidirectional)	Plug	Fifth output – identical copy of the input signal.

OUT6	AIS (unidirectional)	Plug	Sixth output – identical copy of the input signal.
OUT7	AIS (unidirectional)	Plug	Seventh output – identical copy of the input signal.
OUT8	AIS (unidirectional)	Plug	Eighth output – identical copy of the input signal.
OUT9	AIS (unidirectional)	Plug	Ninth output – identical copy of the input signal.

Functionality

The **AIS_SPLIT_9** is a purely combinational function block without its own state logic or event control. It receives an AIS signal at the **Socket IN** and forwards it unchanged to all nine **Plugs OUT1** to **OUT9**. Any change at the input is immediately transmitted to all outputs. The function block thus acts as a passive signal distributor ("broadcast") on the AIS interface.

Technical Features

- **Generic Function Block**: The function block is defined as a generic type (`GenericClassName = GEN_AIS_SPLIT`). This allows for flexible reuse in different AIS contexts without having to recreate the function block for each application.
- **Unidirectional Transmission**: Both inputs and outputs use the unidirectional **AIS** adapter type, meaning data flows in only one direction (from the input to the outputs).
- **No Data Buffering or Delay**: Due to the lack of event and data handling, the function block operates without buffering or clocking – it is strictly signal-passing.
- **Simple Interface**: The function block has neither events nor data inputs/outputs, but only adapters. This makes it particularly lightweight and suitable for pure signal coupling.

State Overview

The **AIS_SPLIT_9** has no internal states or sequence controls. Its behavior is deterministic and always determined by the input signal. There are no initializing or fault states.

Application Scenarios

- **Signal Multiplication**: An AIS-based control signal is to be passed on in parallel to several independent actuators, controllers, or displays.
- **Broadcast in Modular Systems**: In a distributed automation environment, a central measured value (e.g., temperature, pressure) is provided via AIS and must be consumed simultaneously by several function blocks.
- **Prototyping and Testing**: A test signal is to be distributed to multiple instances of the same function block type without requiring the source function block to be instantiated multiple times.

Comparison with Similar Components

- **AIS_SPLIT_2 / AIS_SPLIT_4 / AIS_SPLIT_8** etc.: These components distribute the input signal to 2, 4, or 8 outputs. The **AIS_SPLIT_9** is specifically designed for applications requiring exactly nine parallel connections. The functionality is identical – the only difference lies in the number of outputs.
- **AIS_MERGE**: Unlike the Split component, a Merge component combines multiple inputs into a single output. **AIS_SPLIT_9** implements the reverse data flow.
- **Direct Coupling**: Instead of a Split component, manual wiring (multiple OUT adapters) could also be performed in the 4diac IDE. However, the **AIS_SPLIT_9** simplifies the graphical representation and reduces the complexity of the network.

AIS_MERGE

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AIS_SPLIT_9** is a simple yet extremely useful function block for duplicating a unidirectional AIS signal to nine outputs. Its generic design, the absence of event or data logic, and its clear interface make it a reliable tool for modular automation solutions. It is particularly suitable for applications where a signal needs to be distributed to multiple receivers without delay or modification.

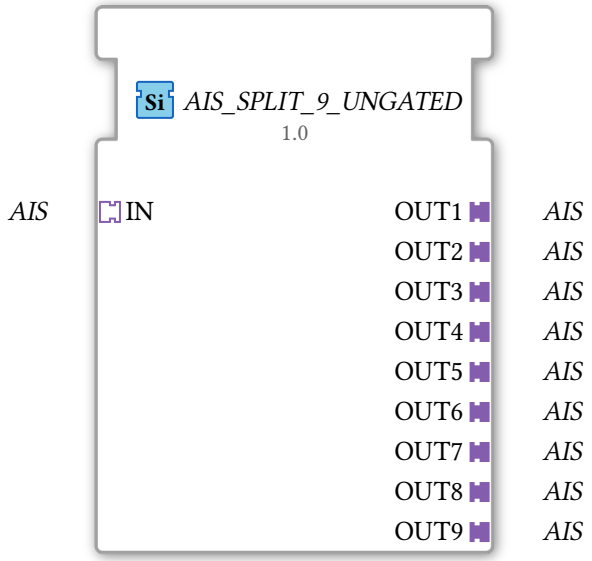
Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AIS_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **AIS_SPLIT_9_UNGATED** function block is used to distribute an incoming unidirectional **AIS** adapter (Application Interface Socket) to nine identical outgoing **AIS** adapters (plugs). It is designed as a generic building block and enables the simple duplication of an AIS signal for up to nine downstream function blocks.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Name	Type	Direction	Description

IN	AIS (unidirectional)	Socket	Input adapter – the AIS signal to be distributed.
OUT1	AIS (unidirectional)	Plug	First output – identical copy of the input signal.
OUT2	AIS (unidirectional)	Plug	Second output – identical copy of the input signal.
OUT3	AIS (unidirectional)	Plug	Third output – identical copy of the input signal.
OUT4	AIS (unidirectional)	Plug	Fourth output – identical copy of the input signal.
OUT5	AIS (unidirectional)	Plug	Fifth output – identical copy of the input signal.
OUT6	AIS (unidirectional)	Plug	Sixth output – identical copy of the input signal.
OUT7	AIS (unidirectional)	Plug	Seventh output – identical copy of the input signal.
OUT8	AIS (unidirectional)	Plug	Eighth output – identical copy of the input signal.
OUT9	AIS (unidirectional)	Plug	Ninth output – identical copy of the input signal.

Functionality

The **AIS_SPLIT_9_UNGATED** is a purely combinational function block without its own state logic or event control. It receives an AIS signal at the **Socket IN** and forwards it unchanged to all nine **Plugs OUT1** to **OUT9**. Any change at the input is immediately transmitted to all outputs. The function block thus acts as a passive signal distributor ("broadcast") on the AIS interface.

Technical Features

- **Generic Function Block**: The function block is defined as a generic type (`GenericClassName = GEN_AIS_SPLIT`). This allows for flexible reuse in different AIS contexts without having to recreate the function block for each application.
- **Unidirectional Transmission**: Both inputs and outputs use the unidirectional **AIS** adapter type, meaning data flows in only one direction (from the input to the outputs).
- **No Data Buffering or Delay**: Due to the lack of event and data handling, the function block operates without buffering or clocking – it is strictly signal-passing.
- **Simple Interface**: The function block has neither events nor data inputs/outputs, but only adapters. This makes it particularly lightweight and suitable for pure signal coupling.

State Overview

The **AIS_SPLIT_9_UNGATED** has no internal states or sequence controls. Its behavior is deterministic and always determined by the input signal. There are no initializing or fault states.

Application Scenarios

- **Signal Multiplication**: An AIS-based control signal is to be passed on in parallel to several independent actuators, controllers, or displays.
- **Broadcast in Modular Systems**: In a distributed automation environment, a central measured value (e.g., temperature, pressure) is provided via AIS and must be consumed simultaneously by several function blocks.
- **Prototyping and Testing**: A test signal is to be distributed to multiple instances of the same function block type without requiring the source function block to be instantiated multiple times.

Comparison with Similar Components

- **AIS_SPLIT_2 / AIS_SPLIT_4 / AIS_SPLIT_8** etc.: These components distribute the input signal to 2, 4, or 8 outputs. The **AIS_SPLIT_9_UNGATED** is specifically designed for applications requiring exactly nine parallel connections. The functionality is identical – the only difference lies in the number of outputs.
- **AIS_MERGE**: Unlike the Split component, a Merge component combines multiple inputs into a single output. **AIS_SPLIT_9_UNGATED** implements the reverse data flow.
- **Direct Coupling**: Instead of a Split component, manual wiring (multiple OUT adapters) could also be performed in the 4diac IDE. However, the **AIS_SPLIT_9_UNGATED** simplifies the graphical representation and reduces the complexity of the network.

AIS_MERGE

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AIS_SPLIT_9_UNGATED** is a simple yet extremely useful function block for duplicating a unidirectional AIS signal to nine outputs. Its generic design, the absence of event or data logic, and its clear interface make it a reliable tool for modular automation solutions. It is particularly suitable for applications where a signal needs to be distributed to multiple receivers without delay or modification.

🌐 Related topic subpages on ms-muc-docs.de

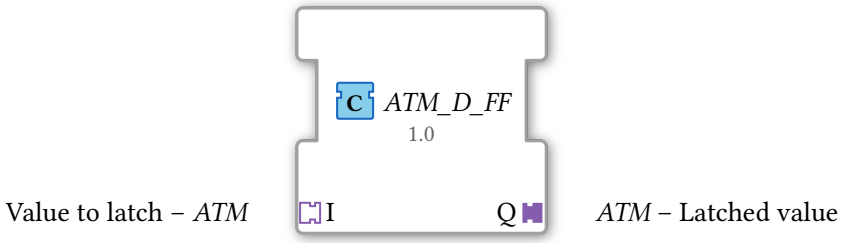
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

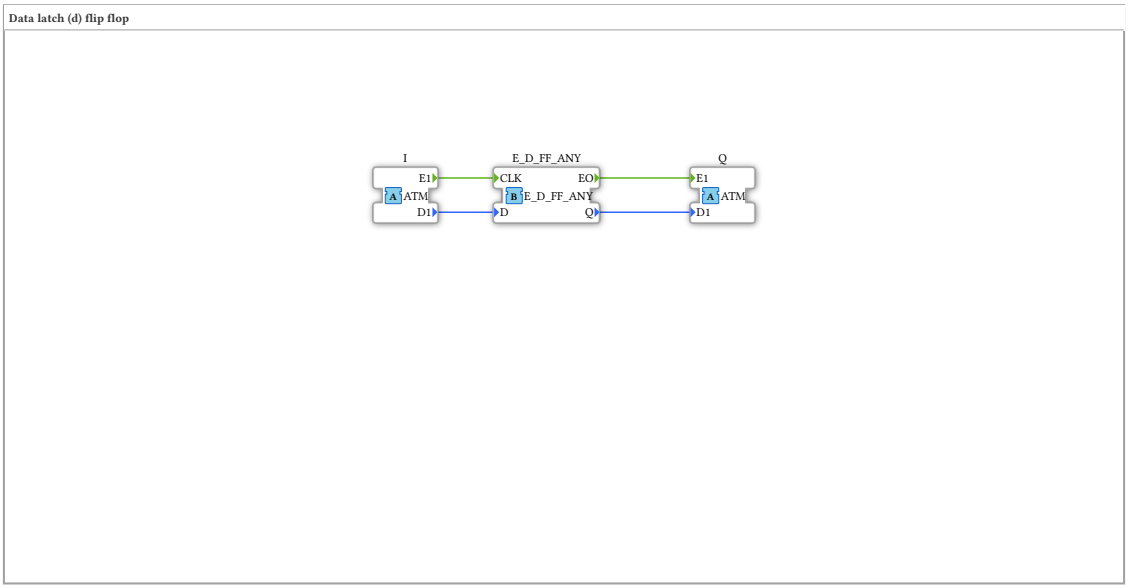
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ATM_D_FF





• – ○ • – ○ • – ○ •

Introduction

The function block `ATM_D_FF` implements a data latch (D flip-flop) for storing a single data value. Communication occurs exclusively via adapters of type `adapter::types::unidirectional::ATM`, so events and data are transmitted as a bundled interface. The function block is implemented according to IEC 61499-1 Annex A and is licensed under the Eclipse Public License 2.0.

Interface Structure

Event Inputs

The function block does not have direct event inputs. An event is received at the adapter interface *E1* via the socket adapter **I** (type: `adapter::types::unidirectional::ATM`).

Event Outputs

The FB has no direct event outputs. An event is output via the plug adapter **Q** (type: `adapter::types::unidirectional::ATM`) at the adapter interface *E1*.

Data Inputs

The FB has no direct data inputs. A data value is provided via the socket adapter **I** at the adapter interface *D1*.

Data Outputs

The FB has no direct data outputs. A data value is output via the plug adapter **Q** at the adapter interface *D1*.

Adapters

Adapter	Type	Direction	Comment
I	`adapter::types::unidirectional::ATM`	Socket (incoming)	Latch value
Q	`adapter::types::unidirectional::ATM`	Plug (outgoing)	Stored value

Functionality

The function block internally uses the standard block `iec61499::events::E_D_FF_ANY`. Upon an event at the input adapter **I.E1** (clock), the current data value is taken from **I.D1** and temporarily stored. Immediately afterward, an event is triggered at the output adapter **Q.E1**, and the stored value is output to **Q.D1**. This behavior corresponds to an edge-triggered D flip-flop.

Technical Features

- No direct event or data ports – all communication takes place via adapters, which enables flexible coupling in IEC 61499 systems.
- The function block is fully implemented by the internal `E_D_FF_ANY`; the state logic is handled by this block.
- Licensed under the Eclipse Public License 2.0; source code and metadata are stored in the function block type.

State Overview

The `ATM_D_FF` does not have its own state engine. Its behavior is purely data flow-driven:

- **Waiting for Event** – the last stored value is retained.
- **On Event at I.E1** – the value from I.D1 is taken and immediately output to Q.

Internally, edge-triggered takeover is implemented by `E_D_FF_ANY` (rising edge of the event).

Application Scenarios

- **Cache** for data that should be updated synchronously with an event.
- **State indicator** in automation processes (e.g., marker for "Value valid").
- **Data transfer** between asynchronous process steps when a simple latch is sufficient.
- **Adapter-based communication** in distributed systems where flat interfaces are preferred.

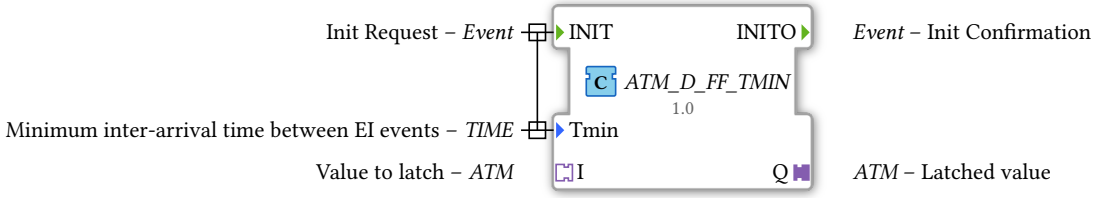
Comparison with similar components

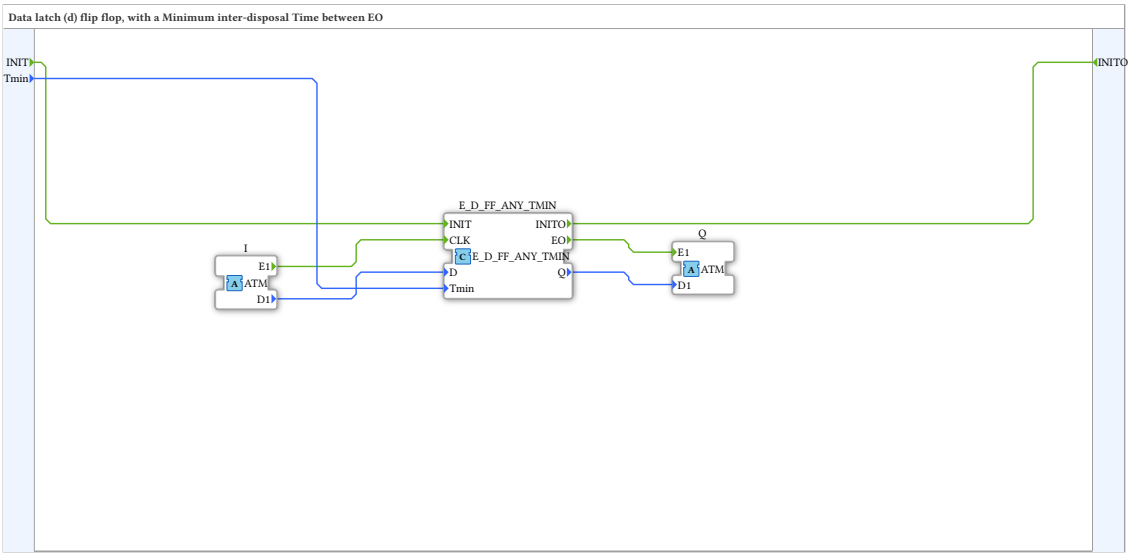
- **SR flip-flop** (`ATM_SR_FF`): Sets/resets state via two events, no clock-driven data input.
- **T flip-flop** (`ATM_T_FF`): Toggles the state on each event.
- **Direct `E_D_FF_ANY`**: Offers the same logic, but with explicit input/output ports instead of adapters. The `ATM_D_FF` encapsulates this logic and standardizes the interface to the unidirectional ATM adapter, facilitating reuse in adapter-based frameworks.

Conclusion

ATM_D_FF` is a compact, reliable functional block for implementing a D flip-flop in IEC 61499 environments. The use of adapters ensures a clean separation of event and data paths, which is particularly advantageous in modular, reusable automation solutions. The component is easy to integrate and fulfills the basic requirement for an edge-triggered data storage device.

ATM_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block `ATM_D_FF_TMIN` implements a clock-controlled D flip-flop (data latch) that takes the incoming data value at a clock event and outputs it as a stored value. A special feature is the adjustable minimum time (`Tmin`) between two consecutive output events, which guarantees a defined dead time.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request (with `Tmin`)

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirmation

Data Inputs

Name	Type	Comment
Tmin	TIME	Minimum time interval between two EO events

Data Outputs

The function block has no direct data outputs. The stored value is provided via the adapter output `Q.D1`.

Adapter

Direction	Name	Type	Comment
Input (Socket)	I	`adapter::types::unidirectional::ATM`	Value to be adopted
Output (Plug)	Q	`adapter::types::unidirectional::ATM`	Stored value

The adapter `ATM` provides one event channel and one data channel (`E1`, `D1` for input; `E1`, `D1` for output).

Functionality

1. **Initialization**: An event at `INIT` initializes the function block and adopts the parameter `Tmin`.
2. **Clocking**: An event at the adapter input `I.E1` (clock) causes the function block to take the current data value from `I.D1`. If the takeover is successful, the value is output at the adapter output `Q.D1`, and an event is simultaneously sent to `Q.E1`.
3. **Minimum Output Delay**: After each output event (`Q.E1`), an internal lock is activated. Only after the time specified in `Tmin` has elapsed can another clock event trigger a new output. If a new clock signal arrives within the lock time, it is ignored, or the output is delayed until `Tmin` expires (depending on the implementation of the internal function block `E_D_FF_ANY_TMIN`).

Technical Features

- **Dead Time Control**: The parameter `Tmin` forces a minimum pause between two output events – useful for protecting downstream components or debouncing signal edges.
- **Internal Implementation**: The function block internally uses the function block `iec61499::events::E_D_FF_ANY_TMIN`, which provides the actual flip-flop logic with timing. `ATM_D_FF_TMIN` acts as an interface adapter for the adapter `ATM`.
- **Internal Implementation**: The function block internally uses the function block `iec61499::events::E_D_FF_ANY_TMIN`, which provides the actual flip-flop logic with timing control. `ATM_D_FF_TMIN` acts as an interface adapter for the adapter `ATM`.

Default Time Control: - **Adapter Interface**: Both input and output data are exchanged via unidirectional adapters, enabling loose coupling in the application.

State Overview

Although no explicit state diagram is available, the following logical states can be derived:

1. **Ready**: Waiting for a clock event, no lock active.
2. **Set**: Value has been accepted and output; the lock `Tmin` is active.
3. **Locked**: During `Tmin`, incoming clock events are ignored or held back.

After the lock expires, the function block returns to the **Ready** state.

Application Scenarios

- **Signal conditioning** in automation technology, where a switching signal needs to be re-evaluated after a defined minimum pause (e.g., dead time during valve switching).
- **Debouncing** of binary sensors where a downstream output may only be generated after a stable time of `Tmin`.
- **Synchronization** of data streams where a minimum packet gap must be maintained.

Comparison with similar components

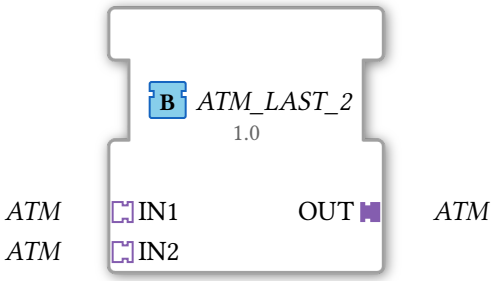
- **Standard D flip-flop** (e.g., `E_D_FF`): Provides an output immediately after each clock cycle – without any time constraint. The `ATM_D_FF_TMIN` extends this with the dead time `Tmin`.
- **Monostable flip-flops** (timers): Generate a pulse of defined length. The `ATM_D_FF_TMIN`, however, only disables the output without resetting the output level.

This component combines the properties of a flip-flop with an adjustable minimum delay time and is therefore particularly suitable for time-critical control tasks.

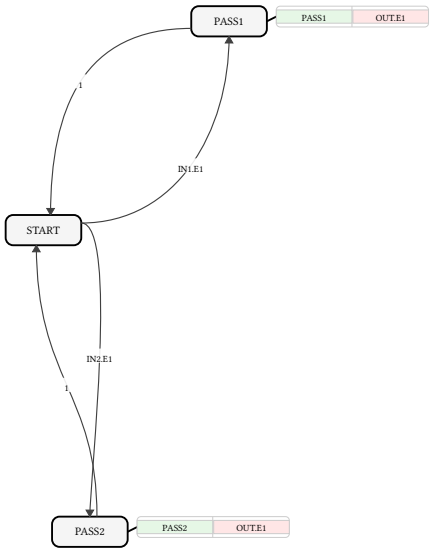
Conclusion

ATM_D_FF_TMIN` is a specialized D-type flip-flop component that guarantees the minimum dwell time between two output events. The use of adapters simplifies integration into modular control applications, while the adjustable delay time offers high flexibility for different physical constraints. It is a valuable addition to the IEC 61499 library for discrete-time signal processing.

ATM_LAST_2



ECC: ATM_LAST_2



Introduction

The ATM_LAST_2 function block merges two same-type TIME adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: TIME adapter (unidirectional) - first signal source
- **IN2**: TIME adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: TIME adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

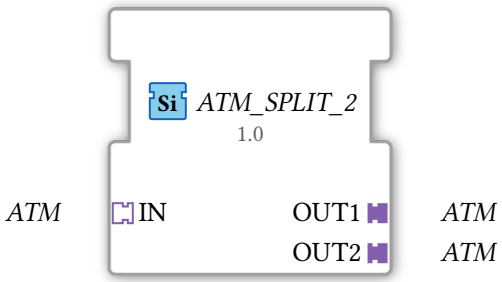
Application Scenarios

- Merging two equally-valid TIME signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same TIME value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

ATM_LAST_2 is the inverse of `ATM_SPLIT_2`: ATM_SPLIT_2 distributes 1 input to 2 outputs, ATM_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

ATM_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The **ATM_SPLIT_2** is a generic function block that splits an incoming adapter of type `adapter::types::unidirectional::ATM` into 2 separate outputs (OUT1, OUT2). It serves to forward a time signal to 2 subsequent function blocks without modifying the data. The function block is implemented as a generic type (`GEN_ATM_SPLIT`) and is parameterized at runtime.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::ATM`	IN	Socket	Input signal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Output 1 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Output 2 (identical to IN)

Functionality

This block forwards the ATM signal present at socket **IN** unchanged to all 2 plugs (OUT1, OUT2). No data manipulation, filtering, or delay occurs. The distribution is purely structural: each output receives its own copy of the reference to the underlying time value.

Technical Features

- **Generic Implementation**: The function block uses the generic class name attribute (`eclipse4diac::core::GenericClassName`) with the value `'GEN_ATM_SPLIT'`, so the same class covers the arities `ATM_SPLIT_2`, `ATM_SPLIT_3`, and `ATM_SPLIT_4` via the GenericClassName mechanism.
- **Unidirectional Adapters**: All adapters are of type `adapter::types::unidirectional::ATM` (forward direction only).
- **No State Automation**: The function block does not have an explicit ECC (Execution Control Chart); signal routing is direct and event-independent.

State Overview

The function block does not implement state automation. Its functionality is limited to passively routing the input signal to all 2 outputs. Therefore, state visualization is not required.

Application Scenarios

- **Signal Distribution**: Splitting an ATM-based time signal (e.g., a delay duration) to multiple parallel-operating control components.
- **Redundancy**: Providing the same time value for a primary and a redundant system.
- **Debugging**: Connecting an analysis or logging module in parallel to the existing path without interrupting the original signal chain.

Comparison with Similar Function Blocks

Similar functionality is offered by the other arities `ATM_SPLIT_2`/`ATM_SPLIT_3`/`ATM_SPLIT_4`, as well as the structurally identical `AR_SPLIT_2` for the `REAL` data type. The choice of arity depends on the required number of outputs.

Change Detection

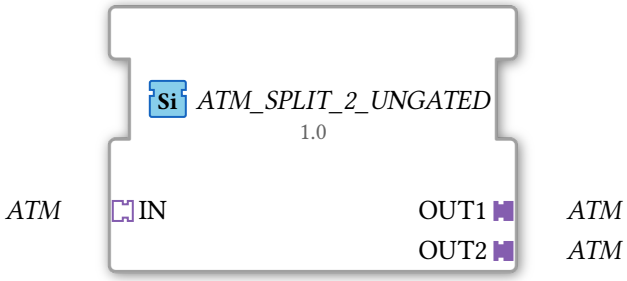
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ATM_SPLIT_2** is a simple yet essential module for multiplying a time signal in IEC 61499-based control systems. Its generic design and clear interface make it the first choice when a time value needs to be passed to multiple independent target components.

ATM_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ATM_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **ATM_SPLIT_2_UNGATED** is a generic function block that splits an incoming adapter of type `adapter::types::unidirectional::ATM` into 2 separate outputs (OUT1, OUT2). It serves to forward a time signal to 2 subsequent function blocks without modifying the data. The function block is implemented as a generic type (`GEN_ATM_SPLIT`) and is parameterized at runtime.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::ATM`	IN	Socket	Input signal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Output 1 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Output 2 (identical to IN)

Functionality

This block forwards the ATM signal present at socket **IN** unchanged to all 2 plugs (OUT1, OUT2). No data manipulation, filtering, or delay occurs. The distribution is purely structural: each output receives its own copy of the reference to the underlying time value.

Technical Features

- **Generic Implementation**: The function block uses the generic class name attribute (`eclipse4diac::core::GenericClassName`) with the value `'GEN_ATM_SPLIT'`, so the same class covers the arities `ATM_SPLIT_2_UNGATED`, `ATM_SPLIT_3`, and `ATM_SPLIT_4` via the GenericClassName mechanism.
- **Unidirectional Adapters**: All adapters are of type `adapter::types::unidirectional::ATM` (forward direction only).
- **No State Automation**: The function block does not have an explicit ECC (Execution Control Chart); signal routing is direct and event-independent.

State Overview

The function block does not implement state automation. Its functionality is limited to passively routing the input signal to all 2 outputs. Therefore, state visualization is not required.

Application Scenarios

- **Signal Distribution**: Splitting an ATM-based time signal (e.g., a delay duration) to multiple parallel-operating control components.
- **Redundancy**: Providing the same time value for a primary and a redundant system.
- **Debugging**: Connecting an analysis or logging module in parallel to the existing path without interrupting the original signal chain.

Comparison with Similar Function Blocks

Similar functionality is offered by the other arities `ATM_SPLIT_2_UNGATED`/`ATM_SPLIT_3`/`ATM_SPLIT_4`, as well as the structurally identical `AR_SPLIT_2` for the `REAL` data type. The choice of arity depends on the required number of outputs.

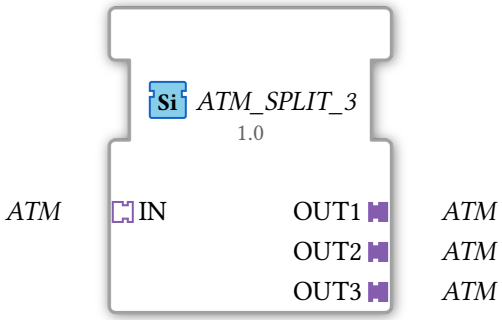
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **ATM_SPLIT_2_UNGATED** is a simple yet essential module for multiplying a time signal in IEC 61499-based control systems. Its generic design and clear interface make it the first choice when a time value needs to be passed to multiple independent target components.

ATM_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The **ATM_SPLIT_3** is a generic function block that splits an incoming adapter of type `adapter::types::unidirectional::ATM` into 3 separate outputs (OUT1, OUT2, OUT3). It serves to forward a time signal to 3 subsequent function blocks without modifying the data. The function block is implemented as a generic type (`GEN_ATM_SPLIT`) and is parameterized at runtime.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description

`adapter::types::unidirectional::ATM`	IN	Socket	Input signal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Output 1 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Output 2 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT3	Plug	Output 3 (identical to IN)

Functionality

This block forwards the ATM signal present at socket **IN** unchanged to all 3 plugs (OUT1, OUT2, OUT3). No data manipulation, filtering, or delay occurs. The distribution is purely structural: each output receives its own copy of the reference to the underlying time value.

Technical Features

- **Generic Implementation**: The function block uses the generic class name attribute (`eclipse4diac::core::GenericClassName`) with the value `'GEN_ATM_SPLIT'`, so the same class covers the arities `ATM_SPLIT_2`, `ATM_SPLIT_3`, and `ATM_SPLIT_4` via the GenericClassName mechanism.
- **Unidirectional Adapters**: All adapters are of type `adapter::types::unidirectional::ATM` (forward direction only).
- **No State Automation**: The function block does not have an explicit ECC (Execution Control Chart); signal routing is direct and event-independent.

State Overview

The function block does not implement state automation. Its functionality is limited to passively routing the input signal to all 3 outputs. Therefore, state visualization is not required.

Application Scenarios

- **Signal Distribution**: Splitting an ATM-based time signal (e.g., a delay duration) to multiple parallel-operating control components.
- **Redundancy**: Providing the same time value for a primary and a redundant system.
- **Debugging**: Connecting an analysis or logging module in parallel to the existing path without interrupting the original signal chain.

Comparison with Similar Function Blocks

Similar functionality is offered by the other arities `ATM_SPLIT_2`/`ATM_SPLIT_3`/`ATM_SPLIT_4`, as well as the structurally identical `AR_SPLIT_3` for the `REAL` data type. The choice of arity depends on the required number of outputs.

Change Detection

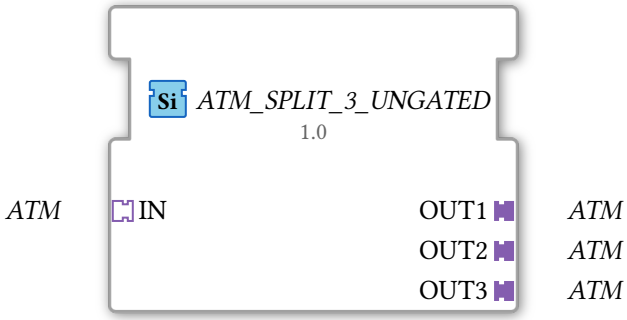
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ATM_SPLIT_3** is a simple yet essential module for multiplying a time signal in IEC 61499-based control systems. Its generic design and clear interface make it the first choice when a time value needs to be passed to multiple independent target components.

ATM_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ATM_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **ATM_SPLIT_3_UNGATED** is a generic function block that splits an incoming adapter of type `adapter::types::unidirectional::ATM` into 3 separate outputs (OUT1, OUT2, OUT3). It serves to forward a time signal to 3 subsequent function blocks without modifying the data. The function block is implemented as a generic type (`GEN_ATM_SPLIT`) and is parameterized at runtime.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::ATM`	IN	Socket	Input signal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Output 1 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Output 2 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT3	Plug	Output 3 (identical to IN)

Functionality

This block forwards the ATM signal present at socket **IN** unchanged to all 3 plugs (OUT1, OUT2, OUT3). No data manipulation, filtering, or delay occurs. The distribution is purely structural: each output receives its own copy of the reference to the underlying time value.

Technical Features

- **Generic Implementation**: The function block uses the generic class name attribute (`eclipse4diac::core::GenericClassName`) with the value `'GEN_ATM_SPLIT'`, so the same class covers the arities `ATM_SPLIT_2`, `ATM_SPLIT_3_UNGATED`, and `ATM_SPLIT_4` via the GenericClassName mechanism.
- **Unidirectional Adapters**: All adapters are of type `adapter::types::unidirectional::ATM` (forward direction only).
- **No State Automation**: The function block does not have an explicit ECC (Execution Control Chart); signal routing is direct and event-independent.

State Overview

The function block does not implement state automation. Its functionality is limited to passively routing the input signal to all 3 outputs. Therefore, state visualization is not required.

Application Scenarios

- **Signal Distribution**: Splitting an ATM-based time signal (e.g., a delay duration) to multiple parallel-operating control components.
- **Redundancy**: Providing the same time value for a primary and a redundant system.
- **Debugging**: Connecting an analysis or logging module in parallel to the existing path without interrupting the original signal chain.

Comparison with Similar Function Blocks

Similar functionality is offered by the other arities `ATM_SPLIT_2`/`ATM_SPLIT_3_UNGATED`/`ATM_SPLIT_4`, as well as the structurally identical `AR_SPLIT_3` for the `REAL` data type. The choice of arity depends on the required number of outputs.

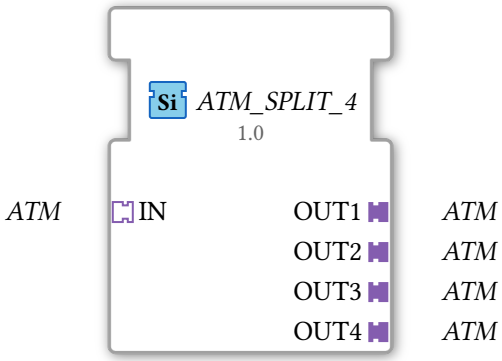
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **ATM_SPLIT_3_UNGATED** is a simple yet essential module for multiplying a time signal in IEC 61499-based control systems. Its generic design and clear interface make it the first choice when a time value needs to be passed to multiple independent target components.

ATM_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The **ATM_SPLIT_4** is a generic function block that splits an incoming adapter of type `adapter::types::unidirectional::ATM` into 4 separate outputs (OUT1, OUT2, OUT3, OUT4). It serves to forward a time signal to 4 subsequent function blocks without modifying the data. The function block is implemented as a generic type (`GEN_ATM_SPLIT`) and is parameterized at runtime.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::ATM`	IN	Socket	Input signal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Output 1 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Output 2 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT3	Plug	Output 3 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT4	Plug	Output 4 (identical to IN)

Functionality

This block forwards the ATM signal present at socket **IN** unchanged to all 4 plugs (OUT1, OUT2, OUT3, OUT4). No data manipulation, filtering, or delay occurs. The distribution is purely structural: each output receives its own copy of the reference to the underlying time value.

Technical Features

- **Generic Implementation**: The function block uses the generic class name attribute (`eclipse4diac::core::GenericClassName`) with the value `'GEN_ATM_SPLIT'`, so the same class covers the arities `ATM_SPLIT_2`, `ATM_SPLIT_3`, and `ATM_SPLIT_4` via the GenericClassName mechanism.
- **Unidirectional Adapters**: All adapters are of type `adapter::types::unidirectional::ATM` (forward direction only).
- **No State Automation**: The function block does not have an explicit ECC (Execution Control Chart); signal routing is direct and event-independent.

State Overview

The function block does not implement state automation. Its functionality is limited to passively routing the input signal to all 4 outputs. Therefore, state visualization is not required.

Application Scenarios

- **Signal Distribution**: Splitting an ATM-based time signal (e.g., a delay duration) to multiple parallel-operating control components.
- **Redundancy**: Providing the same time value for a primary and a redundant system.
- **Debugging**: Connecting an analysis or logging module in parallel to the existing path without interrupting the original signal chain.

Comparison with Similar Function Blocks

Similar functionality is offered by the other arities `ATM_SPLIT_2`/`ATM_SPLIT_3`/`ATM_SPLIT_4`, as well as the structurally identical `AR_SPLIT_4` for the `REAL` data type. The choice of arity depends on the required number of outputs.

Change Detection

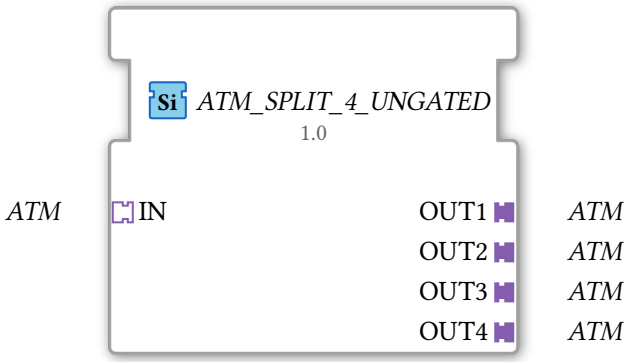
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **ATM_SPLIT_4** is a simple yet essential module for multiplying a time signal in IEC 61499-based control systems. Its generic design and clear interface make it the first choice when a time value needs to be passed to multiple independent target components.

ATM_SPLIT_4_UNGATED

i UNGATED variant: This block is the ungated version of ATM_SPLIT_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **ATM_SPLIT_4_UNGATED** is a generic function block that splits an incoming adapter of type `adapter::types::unidirectional::ATM` into 4 separate outputs (OUT1, OUT2, OUT3, OUT4). It serves to forward a time signal to 4 subsequent function blocks without modifying the data. The function block is implemented as a generic type (`GEN_ATM_SPLIT`) and is parameterized at runtime.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::ATM`	IN	Socket	Input signal (ATM)
`adapter::types::unidirectional::ATM`	OUT1	Plug	Output 1 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT2	Plug	Output 2 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT3	Plug	Output 3 (identical to IN)
`adapter::types::unidirectional::ATM`	OUT4	Plug	Output 4 (identical to IN)

Functionality

This block forwards the ATM signal present at socket **IN** unchanged to all 4 plugs (OUT1, OUT2, OUT3, OUT4). No data manipulation, filtering, or delay occurs. The distribution is purely structural: each output receives its own copy of the reference to the underlying time value.

Technical Features

- **Generic Implementation**: The function block uses the generic class name attribute (`eclipse4diac::core::GenericClassName`) with the value `'GEN_ATM_SPLIT'`, so the same class covers the arities `ATM_SPLIT_2`, `ATM_SPLIT_3`, and `ATM_SPLIT_4_UNGATED` via the GenericClassName mechanism.
- **Unidirectional Adapters**: All adapters are of type `adapter::types::unidirectional::ATM` (forward direction only).
- **No State Automation**: The function block does not have an explicit ECC (Execution Control Chart); signal routing is direct and event-independent.

State Overview

The function block does not implement state automation. Its functionality is limited to passively routing the input signal to all 4 outputs. Therefore, state visualization is not required.

Application Scenarios

- **Signal Distribution**: Splitting an ATM-based time signal (e.g., a delay duration) to multiple parallel-operating control components.
- **Redundancy**: Providing the same time value for a primary and a redundant system.
- **Debugging**: Connecting an analysis or logging module in parallel to the existing path without interrupting the original signal chain.

Comparison with Similar Function Blocks

Similar functionality is offered by the other arities `ATM_SPLIT_2`/`ATM_SPLIT_3`/`ATM_SPLIT_4_UNGATED`, as well as the structurally identical `AR_SPLIT_4` for the `REAL` data type. The choice of arity depends on the required number of outputs.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **ATM_SPLIT_4_UNGATED** is a simple yet essential module for multiplying a time signal in IEC 61499-based control systems. Its generic design and clear interface make it the first choice when a time value needs to be passed to multiple independent target components.

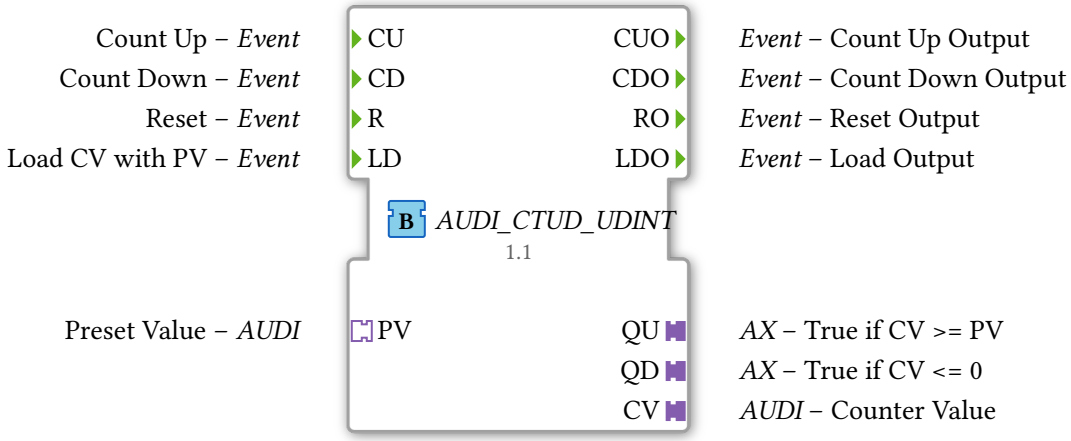
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

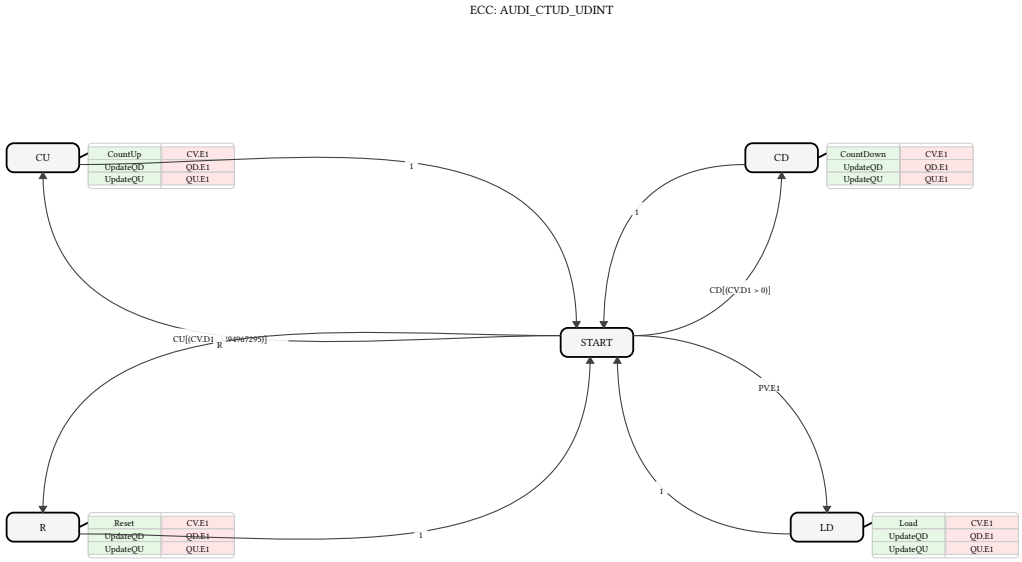
AUDI_CTUD_UDINT (Adapter-Based Up/Down Counter)

Introduction

The `AUDI_CTUD_UDINT` is an event-driven up/down counter for unsigned 32-bit integers (UDINT), specifically designed for integration into adapter-based systems. It utilizes the `AUDI` adapter for passing the counter value and the default value, enabling a clean separation of event and data flow.



Interface Structure



Event Inputs

- **CU**: `Event` - Increments the value by one (`Count Up`; only if `CV.D1` < 4294967295).
- **CD**: `Event` - Decrements the value by one (`Count Down`; only if `CV.D1` > 0).
- **R**: `Event` - Resets the counter to zero (`Reset`).
- **LD**: `Event` - Loads the counter value `CV.D1` with the preset value `PV.D1` (`Load CV with PV`).

Event Outputs

- **CUO**: `Event` - Confirmation of `CU` (`Count Up Output`).
- **CDO**: `Event` - Confirmation of `CD` (`Count Down Output`).
- **RO**: `Event` - Confirmation of `R` (`Reset Output`).
- **LDO**: `Event` - Confirmation of `LD` (`Load Output`).

Output Variables

No classic data outputs; the status signals `QU` and `QD` are provided as AX adapters (plugs), see the Adapter section.

Adapter

- **QU** (Plug): `AX - TRUE` when the counter value (`CV.D1`) is greater than or equal to the preset value (`PV.D1`). The event `QU.E1` is only fired when `QU` changes.
- **QD** (Plug): `AX - TRUE` if the counter value (`CV.D1`) is less than or equal to zero. The event `QD.E1` is only fired when `QD` changes.
- **CV** (Plug): `AUDI` - The current counter value (`Counter Value`).
- **PV** (Socket): `AUDI` - The default value (`Preset Value`) against which the counter is checked (for `QU`).

Functionality

The counter reacts to the event inputs `CU`, `CD`, `R` and `LD`. A `CU` event increments `CV.D1` by 1, a `CD` event decrements `CV.D1` by 1. A `R` event resets `CV.D1` to 0, a `LD` event sets `CV.D1` to `PV.D1`.

When an event (`PV.E1`) arrives at the `PV` adapter, only the status signals `QU`/`QD` are re-evaluated; the counter value `CV.D1` is left unchanged.

The counter value is output via the `CV` adapter as the `AUDI` signal. After each counting action, `CUO`, `CDO`, `RO` or `LDO` confirms the operation and `CV.E1` signals the value change; `QU.E1`/`QD.E1` are only fired when the respective status changes (on-change triggering).

Technical Features

✓ **Adapter-based**: Seamless integration into AX systems.
✓ **Event-driven**: No cyclic calls required.
✓ **UDINT-based**: Supports the full range of unsigned 32-bit integers.
✓ **On-change triggering**: `QU.E1` and `QD.E1` only fire when `QU` or `QD` changes; `PV.E1` re-evaluates both statuses.

Application Scenarios

- **Piece Counting**: Counting objects in production lines.
- **Operating Hour Counter**: Recording operating times (in combination with timers).
- **Position Monitoring**: Simple position counters in handling systems.

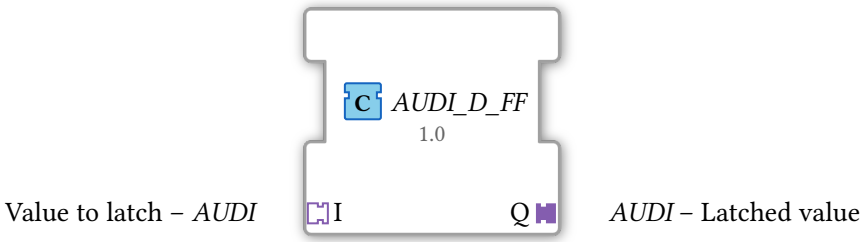
🛠️ **Related exercises**

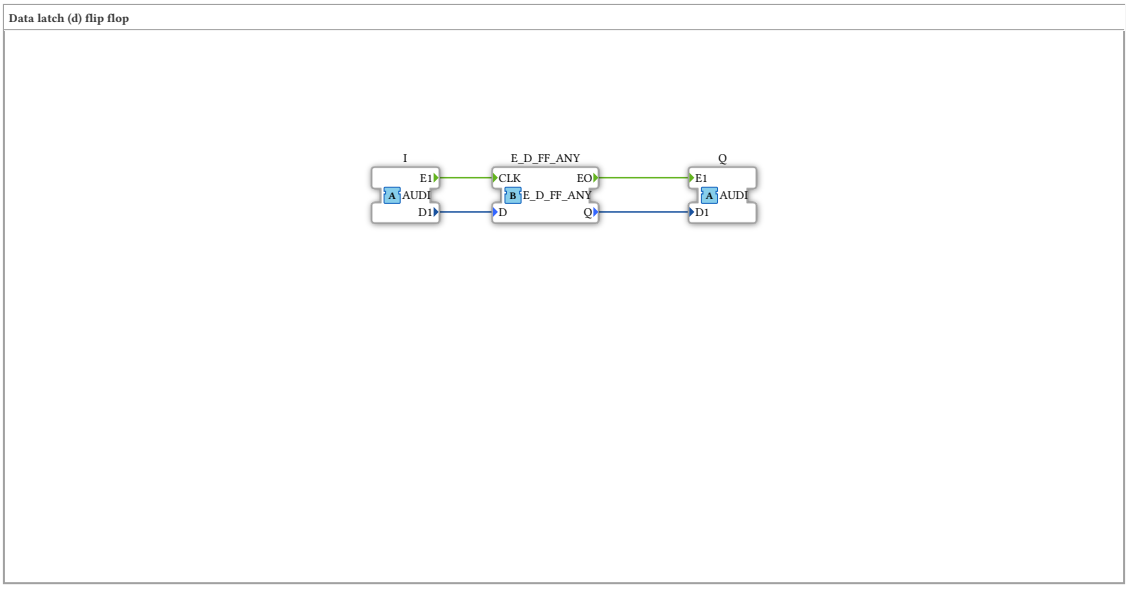
- Uebung_009_AX
- Uebung_083_AX

🌐 **Matching topic subpages on ms-muc-docs.de**

- 🌐 E_CTU Event Counter block on ms-muc-docs.de

AUDI_D_FF





• – ○ • – ○ • – ○ •

Introduction

The function block **AUDI_D_FF** implements a clock-controlled D flip-flop (data latch) based on IEC 61499 adapters. Upon an event at the input adapter, it receives the incoming data value, stores it, and outputs it via the output adapter. The block encapsulates all control logic in a compact, adapter-based interface, thus allowing for a clean separation of event and data communication.

Interface Structure

Event Inputs

None – event control is handled exclusively via the socket adapter.

Event Outputs

None – event output is handled exclusively via the plug adapter.

Data Inputs

None – data acquisition is handled exclusively via the socket adapter.

Data Outputs

None – data output is exclusively via the plug adapter.

Adapters

Name	Type	Direction	Description
I	`adapter::types::unidirectional::AUDI`	Socket (Input)	Provides the clock event (E1) and the data value to be received (D1).
Q	`adapter::types::unidirectional::AUDI`	Plug (Output)	Outputs the output event (E1) and the stored data value (D1).

Both adapters are of the same unidirectional type **AUDI**, which transports an event and a data value together. The socket **I** serves as the source, the plug **Q** as the sink.

Functionality

The function block operates according to the classic D flip-flop function:

1. **Waiting for Clock**: As long as no event is present at input **I.E1**, the internal memory retains its last value.
2. **Takeover on Clock Edge**: An event at **I.E1** (interpreted as a rising edge) causes the function block to take the current data value from **I.D1** and store it in the internal memory.

3. **Output**: After the takeover, an event is output at **Q.E1**, and the stored data value appears at **Q.D1**. This value remains stable until the next clock event arrives.

The internal implementation uses the IEC 61499 standard function block `E_D_FF_ANY`, which provides a type-independent D flip-flop function. Adapters **I** and **Q** are directly connected to its event and data ports.

Technical Features

- **Adapter-based interface**: The function block (FB) does not use individual event or data inputs/outputs, but communicates exclusively via unidirectional adapters. This simplifies connections in complex networks and supports modular type checking.
- **Type independence**: The internal block `E_D_FF_ANY` works with any data type (ANY). The actual data type is defined by the adapter definition of type `AUDI`.
- **Unidirectional communication**: The adapter type `adapter::types::unidirectional::AUDI` transmits events and data only in one direction – from the socket to the plug. Feedback is not provided.
- **No additional initialization state**: The FB starts without a defined data value; The internal state corresponds to the initial state of the `E_D_FF_ANY` function block used.

State Overview

Since this is a purely combinational, event-driven memory, its behavior can be described by two stable states:

- **State A (Waiting)**: No clock event is received – output **Q.D1** holds the last acquired value.
- **State B (Accepting)**: A clock event at **I.E1** triggers the acquisition of **I.D1**. After processing, the function block immediately returns to the waiting state (not a multi-stage state machine).

A graphical representation as a state machine is not required, as the logic is delegated to the internal `E_D_FF_ANY`, whose state diagram is defined in the IEC 61499 specification.

Application Scenarios

- **Data Buffer**: Acquires a sensor signal at a defined measurement time and forwards it to a controller.
- **Synchronization**: Simultaneous reading of multiple data sources via a common clock event.
- **Register Implementation**: Multiple `AUDI_D_FF` modules can be connected in parallel to create an n-bit register.
- **Interface to Event-Driven Communication Protocols**: The adapter is particularly suitable for systems that use standardized data-event pairs (e.g., OPC UA or MQTT connections).

Comparison with Similar Modules

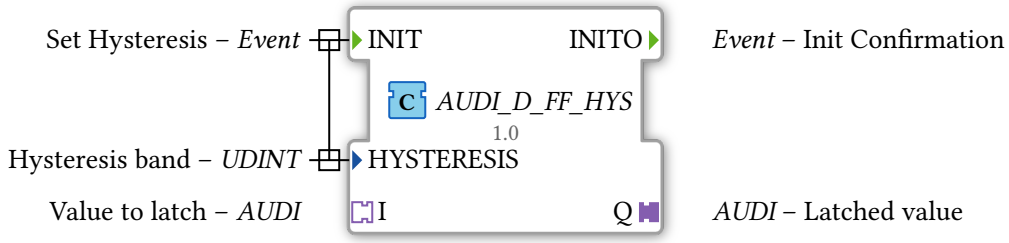
- **`E_D_FF_ANY`** (IEC 61499 standard module): Has separate event and data inputs/outputs, no adapter interface. The `AUDI_D_FF` encapsulates these interfaces in an adapter, thus simplifying their use in adapter-based networks.
- **SR Flip-Flop (Set-Reset)**: Has two data inputs (Set/Reset) and does not allow pure clock operation. The `AUDI_D_FF` is tailored to the simple D flip-flop application.
- **AUDI Adapter-Based Function Blocks (e.g., `AUDI_D_FF_2`)**: Other implementations might include additional logic such as asynchronous reset. This function block implements the minimal D flip-flop without additional control functions.

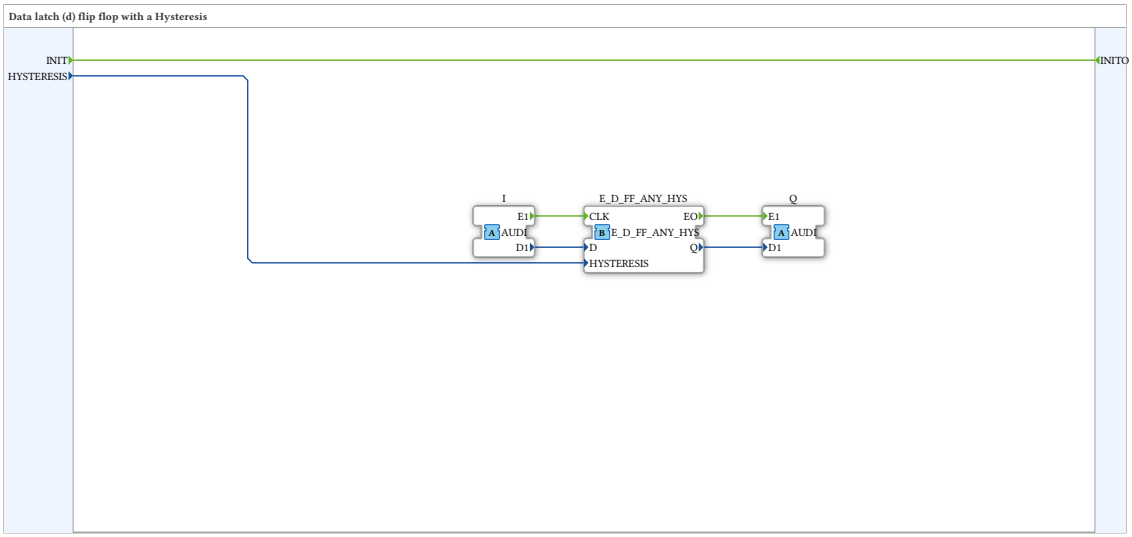
SR Flip-Flop (Set-Reset)

Conclusion

The **AUDI_D_FF** represents a cleanly encapsulated, adapter-based implementation of a D flip-flop. By using the unidirectional `AUDI` adapter type, the interface is reduced to the essentials – one event and one data value per side. The internal use of the proven `E_D_FF_ANY` function block ensures robust, standards-compliant functionality. The function block is particularly suitable for modular architectures that rely on adapter connectivity and facilitates reuse in various control projects.

AUDI_D_FF_HYS





• – ○ • – ○ • – ○ •

Introduction

The function block `AUDI_D_FF_HYS` implements a data-triggered flip-flop (D-latch) with a hysteresis function. It serves to accept an incoming data value upon an event, with the hysteresis enabling switching behavior with a defined deadband. The block encapsulates the logic of the internal block `E_D_FF_ANY_HYS` and provides the interfaces via two unidirectional adapters of type `AUDI`.

Interface Structure

Event Inputs

Event	Type	Comment	With Variable
`INIT`	EInit	Set Hysteresis	`HYSTERESIS`

Event Outputs

Event	Type	Comment
`INITO`	EInit	Init Confirmation

Data Inputs

Variable	Type	Comment
`HYSTERESIS`	UDINT	Hysteresis band

Data Outputs

No direct data outputs – output data is provided via the `Q` adapter.

Adapters

Role	Name	Type (Adapter)	Comment
Plug	Q	adapter::types::unidirectional::AUDI	Latched value
Socket	I	adapter::types::unidirectional::AUDI	Value to latch

The adapter AUDI contains one event channel and one data channel:

- I.E1 – Event that triggers the latch operation
- I.D1 – Data value to be latched
- Q.E1 – Confirmation event after successful latch
- Q.D1 – The latched (and hysteresis-affected) data value

Functionality

The module operates as a wrapper around the internal module E_D_FF_ANY_HYS. During the initialization event INIT, the hysteresis value (passed to HYSTERESIS) is transferred; The INITO event confirms initialization.

During operation, the current data value from I.D1 is taken at each event on the adapter input I.E1 and processed with the configured hysteresis. The result is output via the adapter output Q.D1, accompanied by an event Q.E1.

The hysteresis prevents constant switching with small signal changes: The output only changes its state when the input value leaves the hysteresis range.

Technical Features

- The function block uses the generic implementation E_D_FF_ANY_HYS (from the library logiBUS::signalprocessing::hysteresis).
- The hysteresis is specified as UDINT (unsigned double integer) – the value range is therefore 0… 4294967295.
- All input/output is handled exclusively via unidirectional adapters (AUDI), which promotes clear separation and reusability in complex networks.
- The INIT event is passed directly to INITO without intermediate processing – however, the hysteresis setting is configured before INITO (according to the data connection).

State Overview

Since the FB is a pure function block network (no dedicated ECC), the state is determined by the internal block E_D_FF_ANY_HYS. Simplified:

- **Reset**: Output Q.D1 is 0 or a defined initial value.
- **Set**: The output is set to 1 when the upper hysteresis threshold is exceeded.
- **Hold**: The output remains unchanged for values within the hysteresis range.
- **Reset**: The output is reset when the lower hysteresis threshold is undershot.

Application Scenarios

- **Sensor Signal Debouncing**: Prevents chatter in mechanical switches or proximity sensors.
- **Threshold Switch with Feedback Suppression**: E.g., temperature control with adjustable deadband.
- **Digital Input Buffering**: Adopts a stable value upon a defined event (e.g., clock edge).

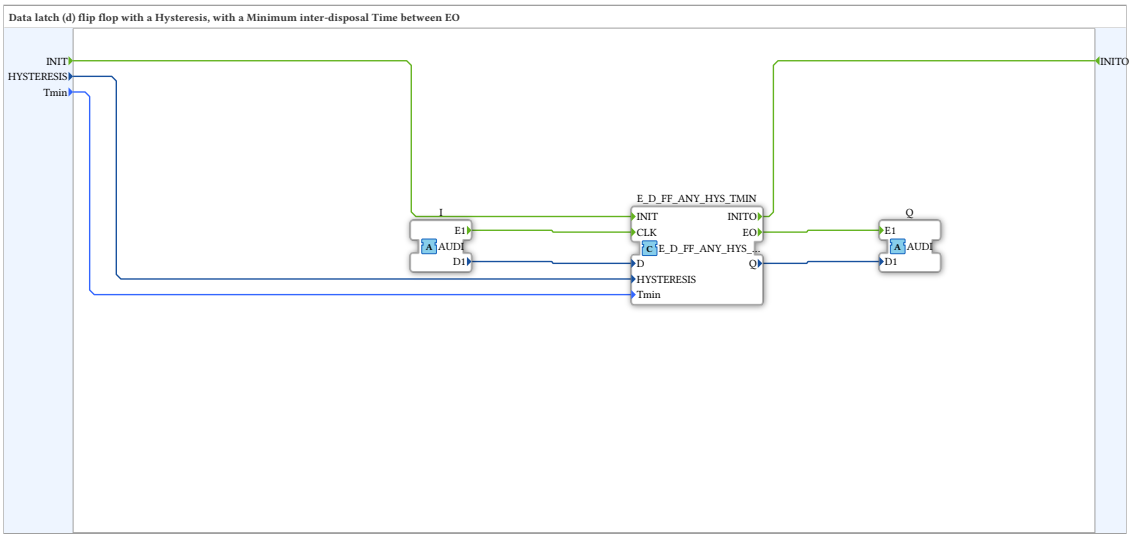
Comparison with Similar Components

- **Standard D Flip-Flop (without hysteresis)**: Responds instantly to any change – no noise or bounce suppression.
- **RS Flip-Flop with hysteresis (e.g., RS_FF_HYS)**: Controls via separate set/reset events, while AUDI_D_FF_HYS uses a single data/event input.
- **Schmitt Trigger Component**: Functionally similar, but usually without latch/store behavior – here, the value is latched at an event and held until the next event.

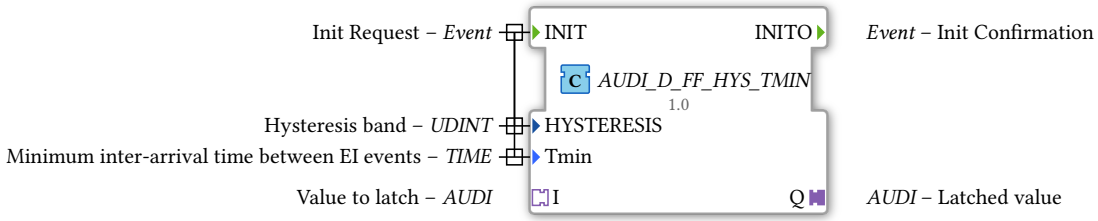
Conclusion

AUDI_D_FF_HYS` combines the properties of a data-triggered flip-flop with adjustable hysteresis. By being encapsulated as an adapter module, it can be easily integrated into modular automation networks and offers a robust solution for signal processing with debouncing or threshold logic.

AUDI_D_FF_HYS_TMIN



No image available



• – ○ • – ○ • – ○ •

Introduction

The function block **AUDI_D_FF_HYS_TMIN** implements a clock-edge-triggered data latch (D flip-flop) with a hysteresis function and a minimum dwell time between output events (EO). It serves to reliably and cleanly acquire a data value while simultaneously suppressing noise and undesirably fast switching operations. The function block encapsulates the logic of an internal component `E_D_FF_ANY_HYS_TMIN` and provides the inputs/outputs via standardized adapters.

Interface Structure

Event Inputs

| Event | Type | Comment | | INIT | EInit | Initialization request, sets hysteresis and minimum time |

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization confirmation

Data Inputs

Variable	Type	Comment
HYSTERESIS	UDINT	Hysteresis band (width of the dead zone for noise suppression)
Tmin	TIME	Minimum time between two consecutive output events

Data Outputs

No direct data outputs – the output data is provided via the **Q** adapter.

Adapter

Name	Type	Direction	Comment
I	`adapter::types::unidirectional::AUDI`	Input	Data value to be received
Q	`adapter::types::unidirectional::AUDI`	Output	Latched value

Functionality

1. **Initialization** – The parameters `HYSTERESIS` and `Tmin` are set via the event input `INIT`. After successful initialization, `INITO` is output.
2. **Data Latch** – Each event at the adapter input `I.E1` is interpreted as a clock signal (CLK). The current data value from `I.D1` is adopted.
3. **Hysteresis** – The output value only changes if the difference between the new and old values exceeds the hysteresis band (`HYSTERESIS`). This suppresses small fluctuations (e.g., noise).
4. **Minimum Time** – After an output event (`Q.E1`), no further output event is generated for the duration `Tmin`, even if new clock signals are present at the input. This prevents an excessively high output event rate.

The function block (FB) internally uses the block `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS_TMIN`, which implements the described logic.

Technical Features

- **Adapter-Based Input/Output** – The FB communicates via standardized unidirectional adapters of type `AUDI`. This enables type-safe and modular integration with other function blocks.
- **Configurable Hysteresis** – The hysteresis width can be adjusted at runtime using `HYSTERESIS`.
- **Minimum Time Between Events** – The minimum time `Tmin` between two output events protects downstream components from overload and defines a maximum update rate.
- **Initialization Required** – Before initial operation, the FB must be configured using `INIT`; otherwise, the internal parameters are undefined.

State Overview

The function block (FB) does not have an explicit state machine in the XML. Its behavior is entirely determined by the internal block `E_D_FF_ANY_HYS_TMIN`. The following states can be logically derived:

- **Initial** – After a reset, waits for `INIT`.
- **Ready** – After `INIT` and `INITO`, ready to receive data.
- **Waiting for Clock** – Awaits an event at `I.E1`.
- **Processing** – Data is being evaluated, hysteresis is being checked.
- **Output Locked** – After an output event, the `Tmin` timer starts running, during which no further output is possible.

Application Scenarios

- **Sensor Value Wiring** – Stabilization of analog or digital sensor signals with hysteresis (e.g., level, temperature, pressure) and update rate limitation.
- **Clock Edge-Triggered Data Acquisition** – Reliable reading of values from noisy environments when a minimum time between value changes is required.
- **Actuator Control** – Protection of actuators from excessively rapid switching by limiting the output event frequency.

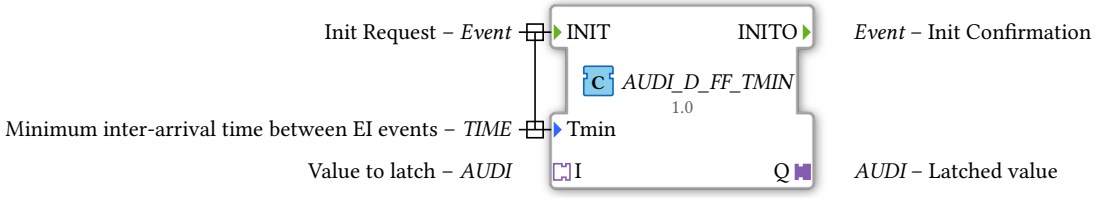
Comparison with Similar Components

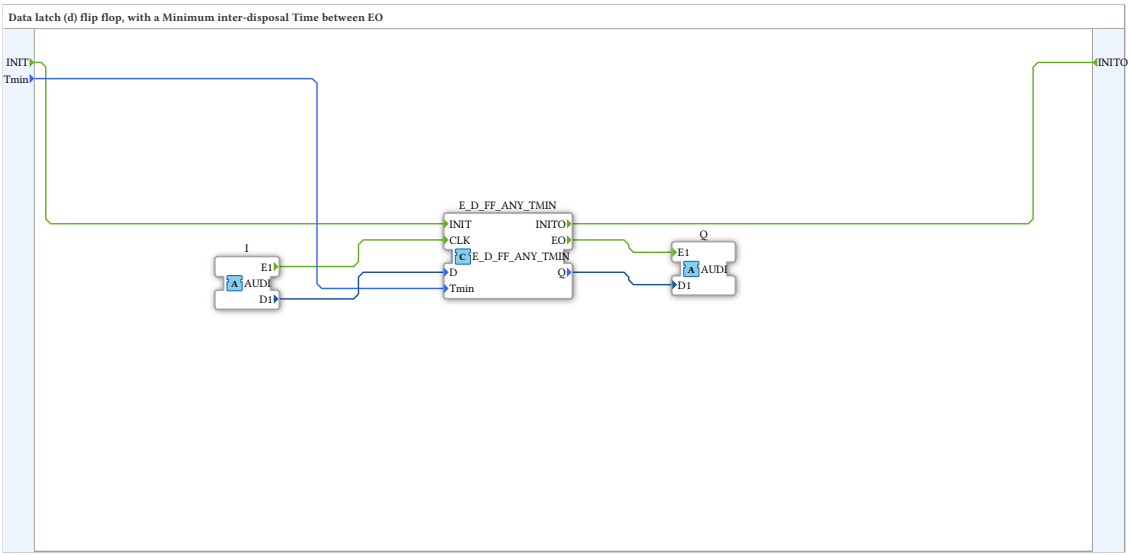
- **Standard D Flip-Flop (e.g., `E_D_FF`)** – No hysteresis and no time limit; responds to every clock edge without filtering.
- **D Flip-Flop with Hysteresis (e.g., `AUDI_D_FF_HYS`)** – Includes hysteresis, but no minimum time between output events.
- **AUDI_D_FF_HYS_TMIN** – Combines both features: hysteresis for noise suppression and `Tmin` for limiting the event rate.

Conclusion

The FB `AUDI_D_FF_HYS_TMIN` offers a robust, configurable solution for clock-driven data acquisition in industrial automation environments. By combining hysteresis and minimal inter-event time, it is particularly suitable for applications with noisy signals or limited output bandwidth. The adapter-based interface facilitates integration into existing systems and promotes reusability.

AUDI_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AUDI_D_FF_TMIN** implements a D flip-flop (data latch) with a minimum lock time between successive output events. It serves to receive a value received via one adapter and output it via a second adapter, with the propagation of the output event being time-controlled.

Interface Structure

Event Inputs

Name	Type	Description
INIT	EInit	Initializes the function block; sets the minimum time between two EO events.

Event Outputs

Name	Type	Description
INITO	EInit	Confirmation of completed initialization.

Data Inputs

Name	Type	Description
Tmin	TIME	Minimum time that must elapse between two consecutive EO events.

Data Outputs

No direct data outputs; the latched value is provided via the **Q** adapter.

Adapters

Name	Type (unidirectional)	Role
I	AUDI	Input adapter: provides the value to be latched (via D1) and the latch event (via E1).
Q	AUDI	Output adapter: outputs the latched value (via D1) and an output event (via E1).

Functionality

The function block internally contains an instance of the function block `E_D_FF_ANY_TMIN`, which implements the core logic of a time-limited D flip-flop.

1. **Initialization**

An INIT event sets the minimum delay time `Tmin` (via the With association) and is passed to the internal instance. After successful initialization, INITO is output.

1. **Latch Operation**

An event at the input `I.E1` (of the adapter) is passed as a clock signal (CLK) to the internal flip-flop. Simultaneously, the value from `I.D1` (D input) is taken. The latched value appears at output `Q.D1`, and the corresponding output event appears at the adapter's event output (`Q.E1`).

1. **Minimum Wait Time**

The internal function block enforces a minimum time interval between successive `Q.E1` events. If the time defined at `Tmin` is exceeded, the next output event is only triggered after this time interval has elapsed. This prevents overloads on the output side.

Technical Features

- **Adapter-Based Communication**

The function block uses two unidirectional AUDI adapters for input and output. This achieves loose coupling between data flow and event control.

- **Interval Control**

The minimum time between two EO events is configured via the parameter `Tmin`. The internal function block `E_D_FF_ANY_TMIN` ensures that this condition is met by delaying incoming clock events as needed.

- **No Explicit Data Outputs**

The latched value is provided exclusively via the adapter `Q`. This simplifies the connection to subsequent function blocks with the same adapter type.

State Overview

The function block has no externally visible states, as the state logic is encapsulated within the internal function block. The internal logic cycles through at least the following internal states:

- **IDLE**: Waiting for INIT or a clock event (I.E1).
- **LATCH**: Value is being stored; waiting time is being checked.
- **OUTPUT**: Outputs Q.E1 and Q.D1 when the minimum time since the last output event has elapsed.

An INIT event resets all states.

Application Scenarios

- **Time-Controlled Data Buffer**

Accepts measured values or control signals at fixed time intervals, e.g., to decouple a fast sensor from a slow actuator.

- **Event Filter**

Suppresses events that follow each other too quickly (application-level debouncing).

- **Synchronization**

Used as a basic building block for implementing delay elements or clock locks in IEC 61499 applications.

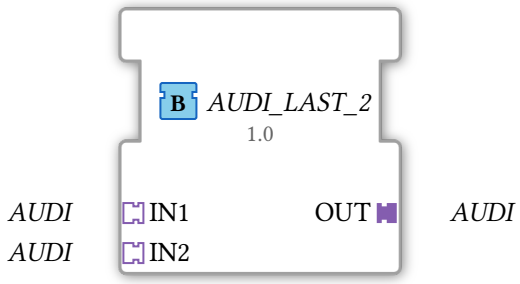
Comparison with Similar Building Blocks

Building Block	Special Feature
AUDI_D_FF	Pure D flip-flop without a time limit between EO events.
E_D_FF_ANY	Standard D flip-flop with universal data types, but without a minimum delay.
AUDI_D_FF_TMIN	Like AUDI_D_FF, but with the additional `Tmin` parameter to control the minimum pause between output events.

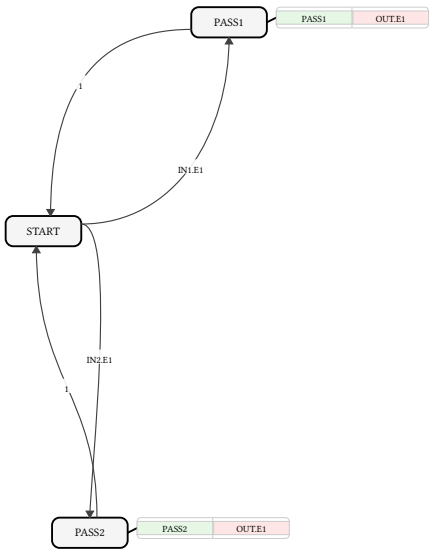
Conclusion

The **AUDI_D_FF_TMIN** is a specialized D flip-flop component that limits the output of events and values. Its adapter interface allows it to be seamlessly integrated into unidirectional communication structures. The combination of a latch function and adjustable lock time makes it a valuable tool for event-driven real-time processing in automation systems.

AUDI_LAST_2



ECC: AUDI_LAST_2



Introduction

The AUDI_LAST_2 function block merges two same-type UDINT adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event

wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: UDINT adapter (unidirectional) - first signal source
- **IN2**: UDINT adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: UDINT adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

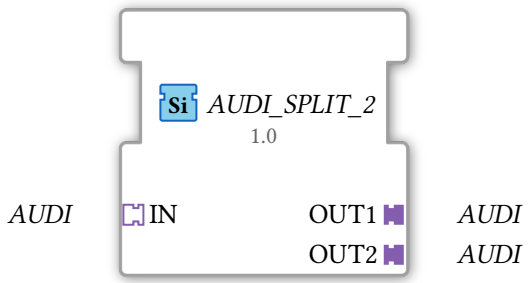
Application Scenarios

- Merging two equally-valid UDINT signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same UDINT value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AUDI_LAST_2 is the inverse of `AUDI_SPLIT_2`: AUDI_SPLIT_2 distributes 1 input to 2 outputs, AUDI_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

AUDI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_2** is a generic block for signal distribution. It receives a signal via a single **AUDI** adapter (unidirectional) and forwards it to two identical **AUDI** output adapters. The function block is declared as a **generic FB**, meaning the specific signal type can be defined during project configuration using the attribute `GenericClassName`. It was developed by HR Agrartechnik GmbH (Version 1.0, 2025-01-24).

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Incoming unidirectional AUDIO adapter (signal source).
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	First output – forwards the input signal.
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Second output – forwards the input signal.

Functionality
The module operates as a **passive splitter** without its own logic or states. The signal arriving via socket `IN` is passed through to both plugs `OUT1` and `OUT2`. The signal is forwarded unchanged and without buffering – any change at the input is immediately visible at both outputs.

Since this is a **unidirectional** adapter, data only flows from the socket to the plugs. No feedback from the outputs to the input is provided.

Technical Features
- **Generic Module**: The attribute `eclipse4diac::core::GenericClassName` allows adaptation to different AUDI types (e.g., `AUDI_Int`, `AUDI_Bool`). The actual type is only determined during instantiation.
- **No internal ECC**: The function block (FB) does not have an execution state machine because it does not process events. Data transfer is purely structural.
- **Unidirectional interface**: Both sockets and plugs are of type `adapter::types::unidirectional::AUDI`, meaning that the data flow direction is fixed.
- **Copyright**: The function block is licensed under the Eclipse Public License 2.0.

State overview
The FB does not have its own state diagrams because it does not contain any event-driven processes. Its function is limited to simple line branching.

Application scenarios
- **Signal distribution**: A central sensor or controller (e.g., a PID controller) sends a value that is to be passed on in parallel to two downstream actuators or monitoring units.
- **Redundant Monitoring**: A measured value is simultaneously sent to two independent evaluation blocks to implement comparison or safety functions.
- **Generic Adapter Architecture**: Particularly useful in systems where the specific AUDI type is only determined at runtime (e.g., in a configurable device platform).

Comparison with Similar Function Blocks
- **Non-generic Splitters** (e.g., `AUDI_SPLIT_2` with a fixed type): Do not allow type adaptation; the function block presented here is more flexible.
- **Other Splitters with More Outputs** (e.g., `AUDI_SPLIT_3`): Increase the number of branches but follow the same principle.

- **Event-based splitters** (e.g., `E_SPLIT`): These require event and data inputs/outputs and perform synchronized distribution—unlike the asynchronous data adapter split presented here.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AUDI_SPLIT_2** is a compact, generic adapter splitter for the 4diac IDE. It performs the simple task of splitting signals to two outputs without introducing additional latency or logic. Its generic design makes it suitable for a wide variety of AUDI data types and allows for flexible reuse in modular automation projects. Its simplicity and type safety make it a solid foundation for distributed control systems.

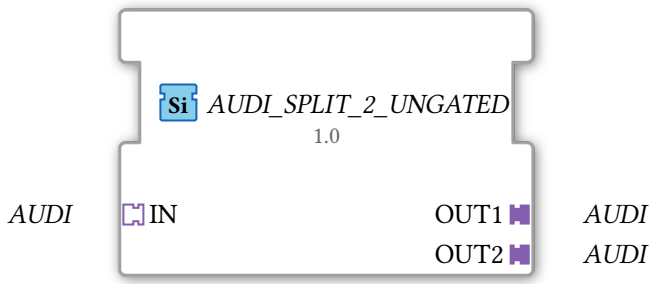
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUDI_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_2_UNGATED** is a generic block for signal distribution. It receives a signal via a single **AUDI** adapter (unidirectional) and forwards it to two identical **AUDI** output adapters. The function block is declared as a **generic FB**, meaning the specific signal type can be defined during project configuration using the attribute `GenericClassName`. It was developed by HR Agrartechnik GmbH (Version 1.0, 2025-01-24).

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Incoming unidirectional AUDIO adapter (signal source).
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	First output – forwards the input signal.
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Second output – forwards the input signal.

Functionality

The module operates as a **passive splitter** without its own logic or states. The signal arriving via socket `IN` is passed through to both plugs `OUT1` and `OUT2`. The signal is forwarded unchanged and without buffering – any change at the input is immediately visible at both outputs.

Since this is a **unidirectional** adapter, data only flows from the socket to the plugs. No feedback from the outputs to the input is provided.

Technical Features

- **Generic Module**: The attribute `eclipse4diac::core::GenericClassName` allows adaptation to different AUDI types (e.g., `AUDI_Int`, `AUDI_Bool`). The actual type is only determined during instantiation.
- **No internal ECC**: The function block (FB) does not have an execution state machine because it does not process events. Data transfer is purely structural.
- **Unidirectional interface**: Both sockets and plugs are of type `adapter::types::unidirectional::AUDI`, meaning that the data flow direction is fixed.
- **Copyright**: The function block is licensed under the Eclipse Public License 2.0.

State overview

The FB does not have its own state diagrams because it does not contain any event-driven processes. Its function is limited to simple line branching.

Application scenarios

- **Signal distribution**: A central sensor or controller (e.g., a PID controller) sends a value that is to be passed on in parallel to two downstream actuators or monitoring units.
- **Redundant Monitoring**: A measured value is simultaneously sent to two independent evaluation blocks to implement comparison or safety functions.
- **Generic Adapter Architecture**: Particularly useful in systems where the specific AUDI type is only determined at runtime (e.g., in a configurable device platform).

Comparison with Similar Function Blocks

- **Non-generic Splitters** (e.g., `AUDI_SPLIT_2_UNGATED` with a fixed type): Do not allow type adaptation; the function block presented here is more flexible.
- **Other Splitters with More Outputs** (e.g., `AUDI_SPLIT_3`): Increase the number of branches but follow the same principle.
- **Event-based splitters** (e.g., `E_SPLIT`): These require event and data inputs/outputs and perform synchronized distribution—unlike the asynchronous data adapter split presented here.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AUDI_SPLIT_2_UNGATED** is a compact, generic adapter splitter for the 4diac IDE. It performs the simple task of splitting signals to two outputs without introducing additional latency or logic. Its generic design makes it suitable for a wide variety of AUDI data types and allows for flexible reuse in

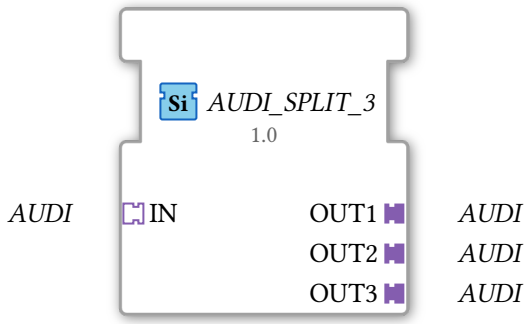
modular automation projects. Its simplicity and type safety make it a solid foundation for distributed control systems.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUDI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_3** is used to multiply an incoming **AUDI** adapter signal to three separate outputs. It is designed as a generic function block (Generic FB) and is particularly suitable for architectures where a signal (e.g., a control or data stream) needs to be distributed to multiple downstream components.

Interface Structure

Event Inputs

- None

Event Outputs

- None

Data Inputs

- None

Data Outputs

- None

Adapter

Type	Name	Description
Socket (adapter::types::unidirectional::AUDI)	**IN**	Input adapter for receiving the AUDI signal
Plug (adapter::types::unidirectional::AUDI)	**OUT1**	First output adapter
Plug (adapter::types::unidirectional::AUDI)	**OUT2**	Second output adapter
Plug (adapter::types::unidirectional::AUDI)	**OUT3**	Third output adapter

Functionality

This module forwards the AUDI signal present at the **IN** socket unchanged to all three **OUT** plugs. No conversion, filtering, or logical processing takes place – the distribution is transparent. The use of adapters of type `unidirectional::AUDI` ensures that the signal direction (input → outputs) is clearly defined.

Technical Features

- **Generic Implementation**: The function block is implemented as a generic FB (`eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`). This allows for easy reuse in different contexts without requiring the specific AUDI type to be fully defined at development time.
- **Type Hash**: The type hash can be set at runtime if needed to enable static type checking.
- **Pure Adapter Interface**: The FB has neither event nor data inputs/outputs in the traditional sense, but communicates exclusively via adapters.

State Overview

Since the FB has no event inputs, it does not have an explicit state machine. Signal transmission occurs continuously and asynchronously as soon as the IN adapter is connected.

Application Scenarios

- **Signal Multiplication**: Distribution of a common control or measurement signal to different function blocks (e.g., parallel connection of several actuators or sensors).
- **Test and Simulation Environments**: Injecting a simulated audio signal into multiple independent analysis or monitoring paths.
- **Architecture Coupling**: Used in modular systems where a data stream needs to be passed to different subsystems without degrading the original signal.

Comparison with Similar Blocks

- **AUDI_MERGE** (counterpart): A combining element that merges multiple audio inputs into one output – unlike the split block.
- **AUDI_SPLIT_2**: A splitter reduced to two outputs; this block extends this to three outputs.
- **Generic Splitters**: Many libraries contain split variants for other adapter types (e.g., `DATA_SPLIT`), but these are data-type specific.
- `AUDI_SPLIT_3_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

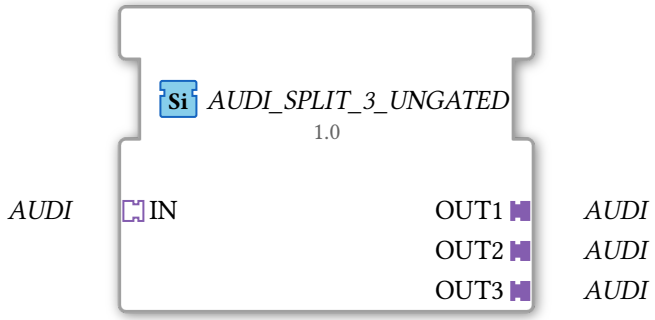
The **AUDI_SPLIT_3** is a simple yet useful function block for distributing a unidirectional AUDI adapter signal to three outputs. Its generic nature and pure adapter interface allow it to integrate seamlessly into event-driven and data-stream-oriented IEC 61499 applications. It provides a clean, maintainable solution for architectural signal distribution and reduces the need for manual signal routing.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Total Resistance in Series & Parallel Circuits on ms-muc-docs.de

AUDI_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_3_UNGATED** is used to multiply an incoming **AUDI** adapter signal to three separate outputs. It is designed as a generic function block (Generic FB) and is particularly suitable for architectures where a signal (e.g., a control or data stream) needs to be distributed to multiple downstream components.

Interface Structure

Event Inputs

- None

Event Outputs

- None

Data Inputs

- None

Data Outputs

- None

Adapter

Type	Name	Description
Socket (adapter::types::unidirectional::AUDI)	**IN**	Input adapter for receiving the AUDI signal
Plug (adapter::types::unidirectional::AUDI)	**OUT1**	First output adapter
Plug (adapter::types::unidirectional::AUDI)	**OUT2**	Second output adapter
Plug (adapter::types::unidirectional::AUDI)	**OUT3**	Third output adapter

Functionality

This module forwards the AUDI signal present at the **IN** socket unchanged to all three **OUT** plugs. No conversion, filtering, or logical processing takes place – the distribution is transparent. The use of adapters of type `unidirectional::AUDI` ensures that the signal direction (input ⟶ outputs) is clearly defined.

Technical Features

- **Generic Implementation**: The function block is implemented as a generic FB (`eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`). This allows for easy reuse in different contexts without requiring the specific AUDI type to be fully defined at development time.
- **Type Hash**: The type hash can be set at runtime if needed to enable static type checking.
- **Pure Adapter Interface**: The FB has neither event nor data inputs/outputs in the traditional sense, but communicates exclusively via adapters.

State Overview

Since the FB has no event inputs, it does not have an explicit state machine. Signal transmission occurs continuously and asynchronously as soon as the IN adapter is connected.

Application Scenarios

- **Signal Multiplication**: Distribution of a common control or measurement signal to different function blocks (e.g., parallel connection of several actuators or sensors).
- **Test and Simulation Environments**: Injecting a simulated audio signal into multiple independent analysis or monitoring paths.
- **Architecture Coupling**: Used in modular systems where a data stream needs to be passed to different subsystems without degrading the original signal.

Comparison with Similar Blocks

- **AUDI_MERGE** (counterpart): A combining element that merges multiple audio inputs into one output – unlike the split block.
- **AUDI_SPLIT_2**: A splitter reduced to two outputs; this block extends this to three outputs.
- **Generic Splitters**: Many libraries contain split variants for other adapter types (e.g., `DATA_SPLIT`), but these are data-type specific.
- `AUDI_SPLIT_3`: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

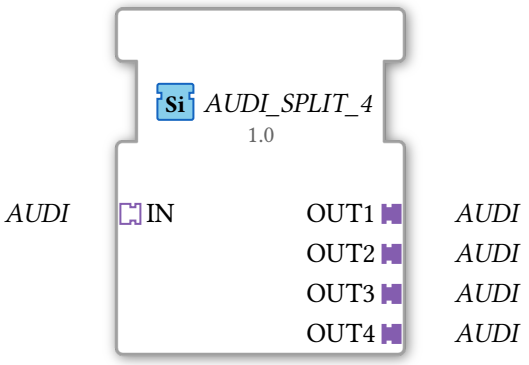
Conclusion

The **AUDI_SPLIT_3_UNGATED** is a simple yet useful function block for distributing a unidirectional AUDI adapter signal to three outputs. Its generic nature and pure adapter interface allow it to integrate seamlessly into event-driven and data-stream-oriented IEC 61499 applications. It provides a clean, maintainable solution for architectural signal distribution and reduces the need for manual signal routing.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Total Resistance in Series & Parallel Circuits on ms-muc-docs.de

AUDI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_4** is used to distribute an incoming unidirectional **AUDI** adapter signal to four identical outputs. It functions as a simple signal distributor (fan-out) for the generic adapter type **AUDI**. The block is implemented as a generic function block (Generic FB) and can be adapted to different adapter types via the class `GEN_AUDI_SPLIT`.

Interface Structure

Event Inputs
No event inputs available.

Event Outputs
No event outputs available.

Data Inputs
No data inputs available.

Data Outputs
No data outputs available.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Input adapter that provides the signal to be distributed.
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	First output, identical to the input signal.
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Second output, identical to the input signal.
Plug	`OUT3`	`adapter::types::unidirectional::AUDI`	Third output, identical to the input signal.
Plug	`OUT4`	`adapter::types::unidirectional::AUDI`	Fourth output, identical to the input signal.

Functionality
The module operates **eventlessly** and **continuously**:

As soon as socket `IN` is connected to an AUDI adapter, the data or signals transmitted via this adapter are forwarded unchanged to all four output adapters (`OUT1` to `OUT4`). No processing or conversion of the data takes place – the module functions solely as a passive distributor (splitter). Since no events or state changes are involved, the distribution occurs implicitly through the connection structure.

Technical Features
- **Generic Function Block**: The function block is declared as a generic type with the class name `'GEN_AUDI_SPLIT'`. This allows the specific adapter type (e.g., a specialized AUDI subtype) to be adapted at design time or through code generation.
- **Unidirectional Adapters**: All adapters used (`AUDI`) are unidirectional, meaning data flows only from the socket to the plugs.
- **No Event Control**: The function block has no event inputs or outputs. Signal distribution is data flow-driven and requires no explicit triggering.
- **Scalability**: The function block is specifically designed for 4 outputs. Variants exist for other numbers (e.g., `AUDI_SPLIT_2`, `AUDI_SPLIT_3`).

State Overview
This function block does **not have a state machine (ECC)**, as it does not process events or store an internal state. Its function is purely combinatorial: The outputs continuously reflect the input.

Application Scenarios
- **Signal Distribution in Control Systems**: An audio signal provided by a sensor or higher-level system is to be passed on in parallel to multiple consumers (actuators, visualizations, logic units).
- **Test and Simulation Environments**: A single test signal is split across multiple parallel test paths.
- **Prototypical Adapter Coupling**: When multiple downstream blocks require the same input signal without the need for logical replication.

Comparison with Similar Function Blocks

Function Block	Function	Special Feature
AUDI_SPLIT_2	Distributes an audio signal to two outputs.	Similar principle, but with two outputs.
AUDI_SPLIT_4`	Distributes an audio signal to four outputs.	This is a simple 1:4 splitter.
AUDI_MERGE`	Combines multiple audio inputs into one output.	Inverted splitter (1:1 mapping, but reversed).
Generic splitters	Can be applied to other adapters via type parameters.	Offer more flexibility than hardwired versions.

Change Detection

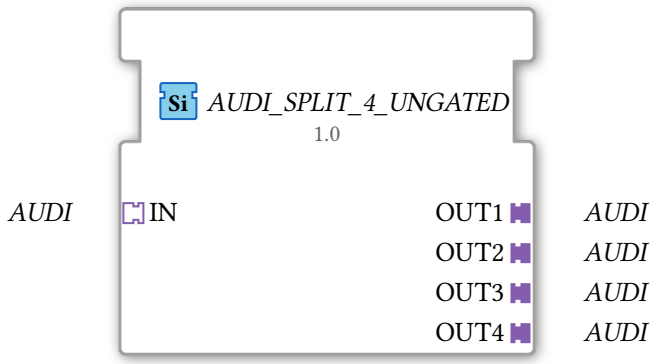
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The AUDI_SPLIT_4 is a simple yet useful function block for multiplying a unidirectional audio adapter signal to four parallel outputs. Its generic nature allows for adaptation to various adapter types, and its eventless operation makes it particularly suitable for pure data distribution tasks without additional control logic. The component contributes to the structuring and reusability of signals in complex automation solutions.

AUDI_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUDI_SPLIT_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_4_UNGATED** is used to distribute an incoming unidirectional **AUDI** adapter signal to four identical outputs. It functions as a simple signal distributor (fan-out) for the generic adapter type **AUDI**. The block is implemented as a generic function block (Generic FB) and can be adapted to different adapter types via the class GEN_AUDI_SPLIT.

Interface Structure

Event Inputs
No event inputs available.

Event Outputs
No event outputs available.

Data Inputs
No data inputs available.

Data Outputs
No data outputs available.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Input adapter that provides the signal to be distributed.
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	First output, identical to the input signal.
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Second output, identical to the input signal.
Plug	`OUT3`	`adapter::types::unidirectional::AUDI`	Third output, identical to the input signal.
Plug	`OUT4`	`adapter::types::unidirectional::AUDI`	Fourth output, identical to the input signal.

Functionality
The module operates **eventlessly** and **continuously**:

As soon as socket `IN` is connected to an AUDI adapter, the data or signals transmitted via this adapter are forwarded unchanged to all four output adapters (`OUT1` to `OUT4`). No processing or conversion of the data takes place – the module functions solely as a passive distributor (splitter). Since no events or state changes are involved, the distribution occurs implicitly through the connection structure.

Technical Features
- **Generic Function Block**: The function block is declared as a generic type with the class name `'GEN_AUDI_SPLIT'`. This allows the specific adapter type (e.g., a specialized AUDI subtype) to be adapted at design time or through code generation.
- **Unidirectional Adapters**: All adapters used (`AUDI`) are unidirectional, meaning data flows only from the socket to the plugs.
- **No Event Control**: The function block has no event inputs or outputs. Signal distribution is data flow-driven and requires no explicit triggering.
- **Scalability**: The function block is specifically designed for 4 outputs. Variants exist for other numbers (e.g., `AUDI_SPLIT_2`, `AUDI_SPLIT_3`).

State Overview
This function block does **not have a state machine (ECC)**, as it does not process events or store an internal state. Its function is purely combinatorial: The outputs continuously reflect the input.

Application Scenarios
- **Signal Distribution in Control Systems**: An audio signal provided by a sensor or higher-level system is to be passed on in parallel to multiple consumers (actuators, visualizations, logic units).
- **Test and Simulation Environments**: A single test signal is split across multiple parallel test paths.
- **Prototypical Adapter Coupling**: When multiple downstream blocks require the same input signal without the need for logical replication.

Comparison with Similar Function Blocks

Function Block	Function	Special Feature
`AUDI_SPLIT_2`	Distributes an audio signal to two outputs.	Similar principle, but with two outputs.
AUDI_SPLIT_4_UNGATED`	Distributes an audio signal to four outputs.	This is a simple 1:4 splitter.
AUDI_MERGE`	Combines multiple audio inputs into one output.	Inverted splitter (1:1 mapping, but reversed).
Generic splitters	Can be applied to other adapters via type parameters.	Offer more flexibility than hard-wired versions.

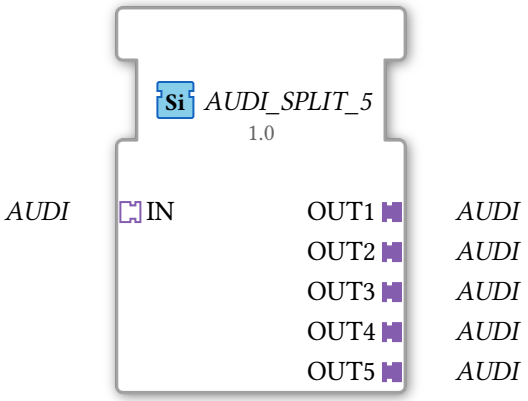
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUDI_SPLIT_4_UNGATED` is a simple yet useful function block for multiplying a unidirectional audio adapter signal to four parallel outputs. Its generic nature allows for adaptation to various adapter types, and its eventless operation makes it particularly suitable for pure data distribution tasks without additional control logic. The component contributes to the structuring and reusability of signals in complex automation solutions.

AUDI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_5** is used to distribute a single incoming AUDI signal to five identical AUDI outputs. It is designed as a generic adapter module and allows for flexible further processing of the signal along multiple paths.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapter

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Incoming AUDIO Signal
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	First Output
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Second Output
Plug	`OUT3`	`adapter::types::unidirectional::AUDI`	Third Output
Plug	`OUT4`	`adapter::types::unidirectional::AUDI`	Fourth Output
Plug	`OUT5`	`adapter::types::unidirectional::AUDI`	Fifth Output

Functionality

This function block forwards the AUDI signal present at socket `IN` unchanged and simultaneously to all five plugs `OUT1` to `OUT5`. No signal processing, conversion, or buffering takes place – distribution occurs purely at the adapter level as soon as a connection to the outputs is established.

Technical Features

- **Generic Function Block**: The function block is implemented as a generic type (`GEN_AUDI_SPLIT`). It can be instantiated for various data types via appropriate runtime environment mechanisms, as long as these data types conform to the unidirectional AUDI adapter protocol.
- **No State Logic**: The function block contains no event or data processing and no internal state machine.
- **Simple and High-Performance Signal Distribution**: Due to its pure adapter structure, it is particularly suitable for scenarios where a signal needs to be duplicated without latency.

State Overview

The function block does not have an explicit state machine. Its behavior is deterministic: As long as the function block is in operation, every signal arriving at `IN` is immediately passed on to all outputs.

Application Scenarios

- **Distributing control signals** in industrial automation, e.g., a start/stop command that is intended to reach several identical modules.
- **Duplicating a measured value** for parallel evaluations or visualizations.
- **Establishing redundancy paths** by using the same signal multiple times.

Comparison with Similar Function Blocks

Unlike function blocks such as `AUDI_MERGE_5` (combining multiple signals) or `AUDI_SPLIT_2` (only two outputs), `AUDI_SPLIT_5` offers a specific distribution across exactly five outputs. Generic split function blocks for other numbers (e.g., `AUDI_SPLIT_N`) often exist as templates, while this function block covers a fixed, but frequently required, configuration.

Change Detection

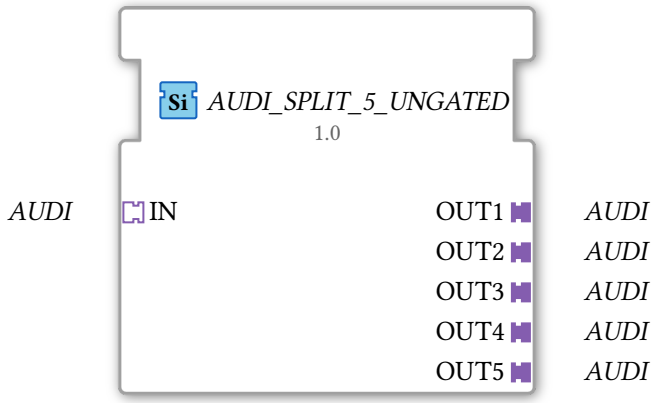
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AUDI_SPLIT_5` is a simple and efficient adapter function block for signal distribution. Its generic design and the absence of event/data logic make it lightweight and versatile. It is ideally suited for all applications where an audio signal needs to be split across five parallel paths.

AUDI_SPLIT_5_UNGATED

i UNGATED variant: This block is the ungated version of `AUDI_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_5_UNGATED** is used to distribute a single incoming AUDI signal to five identical AUDI outputs. It is designed as a generic adapter module and allows for flexible further processing of the signal along multiple paths.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapter

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AUDI`	Incoming AUDIO Signal
Plug	`OUT1`	`adapter::types::unidirectional::AUDI`	First Output
Plug	`OUT2`	`adapter::types::unidirectional::AUDI`	Second Output
Plug	`OUT3`	`adapter::types::unidirectional::AUDI`	Third Output
Plug	`OUT4`	`adapter::types::unidirectional::AUDI`	Fourth Output
Plug	`OUT5`	`adapter::types::unidirectional::AUDI`	Fifth Output

Functionality

This function block forwards the AUDI signal present at socket `IN` unchanged and simultaneously to all five plugs `OUT1` to `OUT5`. No signal processing, conversion, or buffering takes place – distribution occurs purely at the adapter level as soon as a connection to the outputs is established.

Technical Features

- **Generic Function Block**: The function block is implemented as a generic type (`GEN_AUDI_SPLIT`). It can be instantiated for various data types via appropriate runtime environment mechanisms, as long as these data types conform to the unidirectional AUDI adapter protocol.
- **No State Logic**: The function block contains no event or data processing and no internal state machine.
- **Simple and High-Performance Signal Distribution**: Due to its pure adapter structure, it is particularly suitable for scenarios where a signal needs to be duplicated without latency.

State Overview

The function block does not have an explicit state machine. Its behavior is deterministic: As long as the function block is in operation, every signal arriving at `IN` is immediately passed on to all outputs.

Application Scenarios

- **Distributing control signals** in industrial automation, e.g., a start/stop command that is intended to reach several identical modules.
- **Duplicating a measured value** for parallel evaluations or visualizations.
- **Establishing redundancy paths** by using the same signal multiple times.

Comparison with Similar Function Blocks

Unlike function blocks such as `AUDI_MERGE_5` (combining multiple signals) or `AUDI_SPLIT_2` (only two outputs), `AUDI_SPLIT_5_UNGATED` offers a specific distribution across exactly five outputs. Generic split function blocks for other numbers (e.g., `AUDI_SPLIT_N`) often exist as templates, while this function block covers a fixed, but frequently required, configuration.

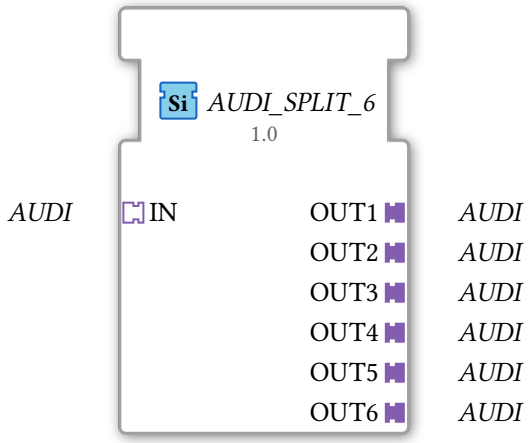
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AUDI_SPLIT_5_UNGATED` is a simple and efficient adapter function block for signal distribution. Its generic design and the absence of event/data logic make it lightweight and versatile. It is ideally suited for all applications where an audio signal needs to be split across five parallel paths.

AUDI_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_6** serves as a generic splitter for a unidirectional AUDI adapter. It receives an incoming adapter data stream via a socket and forwards it unchanged to six separate output adapters (plugs). The block is designed generically, so the specific adapter type can be configured via the attribute `GenericClassName`.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket	**IN**	`adapter::types::unidirectional::AUDI`	Incoming adapter for the data stream to be distributed.
Plug	**OUT1**	`adapter::types::unidirectional::AUDI`	First output adapter (copy of the input).
Plug	**OUT2**	`adapter::types::unidirectional::AUDI`	Second output adapter.
Plug	**OUT3**	`adapter::types::unidirectional::AUDI`	Third output adapter.
Plug	**OUT4**	`adapter::types::unidirectional::AUDI`	Fourth output adapter.
Plug	**OUT5**	`adapter::types::unidirectional::AUDI`	Fifth output adapter.
Plug	**OUT6**	`adapter::types::unidirectional::AUDI`	Sixth output adapter.

Functionality

This function block acts as a pure distributor without any data processing. The data stream arriving via the **IN** socket is forwarded unchanged and in parallel to all six output adapters (OUT1–OUT6). Distribution occurs continuously as soon as the input adapter provides data. Since the function block is generic, it can be adapted to a specific adapter type (e.g., a custom derived type) by setting the attribute `eclipse4diac::core::GenericClassName`.

Technical Features

- **Generic Design:** The function block uses the 4diac generic mechanisms (`GenericClassName`). This allows the adapter type to be set at runtime or during instantiation without recompiling the function block.
- **Unidirectional:** Both the input and output adapters are unidirectional. No feedback from the outputs to the input is provided.
- **No Event Control:** The function block has no event interface and no ECC. Data transmission is purely data-driven (as soon as data is present at the IN socket).
- **No Internal State:** No state change occurs; the function block is stateless.

State Overview

No state machine (ECC) exists. The function block does not execute any sequential processes.

Application Scenarios

- **Data Distribution:** An audio data stream provided by a sensor or another function block is to be sent to several subsequent components simultaneously, e.g., to a visualization, a logging component, and control logic.
- **Parallel Processing:** When the same data stream needs to be processed in parallel in different path segments, e.g., filtering, conditioning, and storage.
- **Generic Use:** Thanks to its configurable type assignment, the function block (FB) can be used in various projects with different adapter definitions without having to modify the source code.

Comparison with Similar Function Blocks

Other split function blocks exist (e.g., `AUDI_SPLIT_2`, `AUDI_SPLIT_4`) that differ only in the number of outputs. With six outputs, `AUDI_SPLIT_6` offers a higher distribution density. Unlike pure data splitters for basic data types, this FB works exclusively with AUDI adapters and supports unidirectional coupling.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AUDI_SPLIT_6` is a simple yet flexible function block for multiplying adapter connections in a 4diac environment. Its generic nature and lean implementation without event overhead make it ideal for scenarios where an incoming data stream needs to be distributed to multiple independent receivers.

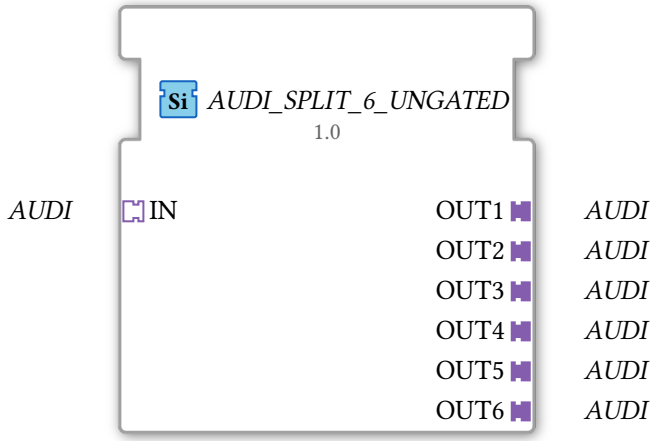
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUDI_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_6_UNGATED** serves as a generic splitter for a unidirectional AUDI adapter. It receives an incoming adapter data stream via a socket and forwards it unchanged to six separate

output adapters (plugs). The block is designed generically, so the specific adapter type can be configured via the attribute `GenericClassName`.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapters

Direction	Name	Type	Description
Socket	**IN**	`adapter::types::unidirectional::AUDI`	Incoming adapter for the data stream to be distributed.
Plug	**OUT1**	`adapter::types::unidirectional::AUDI`	First output adapter (copy of the input).
Plug	**OUT2**	`adapter::types::unidirectional::AUDI`	Second output adapter.
Plug	**OUT3**	`adapter::types::unidirectional::AUDI`	Third output adapter.
Plug	**OUT4**	`adapter::types::unidirectional::AUDI`	Fourth output adapter.
Plug	**OUT5**	`adapter::types::unidirectional::AUDI`	Fifth output adapter.
Plug	**OUT6**	`adapter::types::unidirectional::AUDI`	Sixth output adapter.

Functionality
This function block acts as a pure distributor without any data processing. The data stream arriving via the **IN** socket is forwarded unchanged and in parallel to all six output adapters (OUT1–OUT6). Distribution occurs continuously as soon as the input adapter provides data. Since the function block is generic, it can be adapted to a specific adapter type (e.g., a custom derived type) by setting the attribute `eclipse4diac::core::GenericClassName`.

Technical Features
- **Generic Design:** The function block uses the 4diac generic mechanisms (`GenericClassName`). This allows the adapter type to be set at runtime or during instantiation without recompiling the function block.
- **Unidirectional:** Both the input and output adapters are unidirectional. No feedback from the outputs to the input is provided.
- **No Event Control:** The function block has no event interface and no ECC. Data transmission is purely data-driven (as soon as data is present at the IN socket).
- **No Internal State:** No state change occurs; the function block is stateless.

State Overview
No state machine (ECC) exists. The function block does not execute any sequential processes.

Application Scenarios
- **Data Distribution:** An audio data stream provided by a sensor or another function block is to be sent to several subsequent components simultaneously, e.g., to a visualization, a logging component, and control logic.

- **Parallel Processing:** When the same data stream needs to be processed in parallel in different path segments, e.g., filtering, conditioning, and storage.
- **Generic Use:** Thanks to its configurable type assignment, the function block (FB) can be used in various projects with different adapter definitions without having to modify the source code.

Comparison with Similar Function Blocks

Other split function blocks exist (e.g., `AUDI_SPLIT_2`, `AUDI_SPLIT_4`) that differ only in the number of outputs. With six outputs, `AUDI_SPLIT_6_UNGATED` offers a higher distribution density. Unlike pure data splitters for basic data types, this FB works exclusively with AUDI adapters and supports unidirectional coupling.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

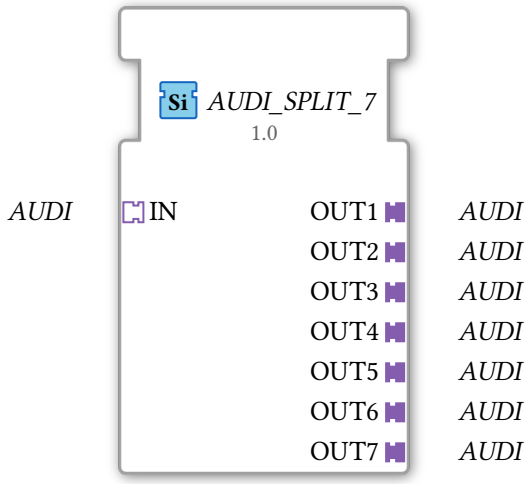
AUDI_SPLIT_6_UNGATED` is a simple yet flexible function block for multiplying adapter connections in a 4diac environment. Its generic nature and lean implementation without event overhead make it ideal for scenarios where an incoming data stream needs to be distributed to multiple independent receivers.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUDI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_SPLIT_7` is used to distribute a single AUDI input signal to seven identical AUDI output signals. It is designed as a generic building block and is suitable for all unidirectional AUDI adapter types.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Name	Type	Direction
IN	adapter::types::unidirectional::AUDI	Socket (Input)
OUT1	adapter::types::unidirectional::AUDI	Plug (Output)
OUT2	adapter::types::unidirectional::AUDI	Plug (Output)
OUT3	adapter::types::unidirectional::AUDI	Plug (Output)
OUT4	adapter::types::unidirectional::AUDI	Plug (Output)
OUT5	adapter::types::unidirectional::AUDI	Plug (Output)
OUT6	adapter::types::unidirectional::AUDI	Plug (Output)
OUT7	adapter::types::unidirectional::AUDI	Plug (Output)

All adapters are of the unidirectional type AUDI and transmit data exclusively from the socket to the plugs.

Functionality

The module copies the AUDIO data received via socket IN unchanged to all seven plug outputs OUT1 to OUT7. No processing or conversion of the data takes place – the function is limited to a simple 1-to-7 distribution (fan-out). Since there are no events or explicit data ports, the signal behavior is entirely defined by the connected adapters.

- **Generic Type**: The component is marked as generic via the attribute eclipse4diac::core::GenericClassName ('GEN_AUDI_SPLIT'). This allows it to be used for various specific implementations of the AUDI adapter without requiring a separate implementation.
- **No Runtime Dependency**: The component has no algorithm and no state diagram; signal transmission occurs purely structurally through the adapter wiring.
- **Type Hash**: The attribute eclipse4diac::core::TypeHash is empty, meaning that the type identity is not additionally secured at runtime.

State Overview

The component does not contain a state machine (ECC). Its behavior is completely deterministic and eventless – a state representation is therefore not required.

Application Scenarios

- **Signal Distribution**: An AUDI signal from a sensor or control unit must be forwarded in parallel to multiple consumers (actuators, displays, monitoring systems).
- **System Expansion**: Existing systems that provide a single AUDI signal are to be expanded with additional components without changing the source logic.
- **Test Setups**: A generated test signal is to be sent to multiple devices under test simultaneously.

Comparison with Similar Components

Component	Distribution	Adapter Type
AUDI_SPLIT_7	1 input ⟶ 7 outputs	Unidirectional AUDI
SPLIT_1_TO_2 (analog)	1 ⟶ 2	Any (generic)
AUDI_MERGE	Multiple Inputs ⟶ 1	Unidirectional AUDI

While AUDI_SPLIT_7 is designed for the specific AUDI adapter type, generic split modules exist for other data formats. Its limitation to a fixed number of seven outputs distinguishes it from flexible splitters with a configurable number of outputs.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

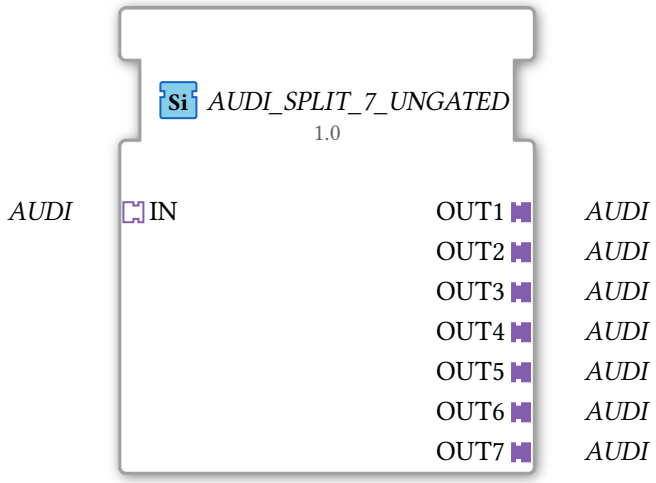
AUDI_SPLIT_7` is a simple yet effective module for distributing an AUDI signal to seven identical paths. Its generic design allows it to be reused in various contexts. The absence of events and internal logic makes it lightweight and reliable for pure signal distribution tasks.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AUDI_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_SPLIT_7_UNGATED` is used to distribute a single AUDI input signal to seven identical AUDI output signals. It is designed as a generic building block and is suitable for all unidirectional AUDI adapter types.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

Name	Type	Direction
IN	adapter::types::unidirectional::AUDI	Socket (Input)
OUT1	adapter::types::unidirectional::AUDI	Plug (Output)
OUT2	adapter::types::unidirectional::AUDI	Plug (Output)
OUT3	adapter::types::unidirectional::AUDI	Plug (Output)
OUT4	adapter::types::unidirectional::AUDI	Plug (Output)
OUT5	adapter::types::unidirectional::AUDI	Plug (Output)
OUT6	adapter::types::unidirectional::AUDI	Plug (Output)
OUT7	adapter::types::unidirectional::AUDI	Plug (Output)

All adapters are of the unidirectional type AUDI and transmit data exclusively from the socket to the plugs.

Functionality

The module copies the AUDIO data received via socket IN unchanged to all seven plug outputs OUT1 to OUT7. No processing or conversion of the data takes place – the function is limited to a simple 1-to-7 distribution (fan-out). Since there are no events or explicit data ports, the signal behavior is entirely defined by the connected adapters.

- **Generic Type**: The component is marked as generic via the attribute eclipse4diac::core::GenericClassName ('GEN_AUDI_SPLIT'). This allows it to be used for various specific implementations of the AUDI adapter without requiring a separate implementation.
- **No Runtime Dependency**: The component has no algorithm and no state diagram; signal transmission occurs purely structurally through the adapter wiring.
- **Type Hash**: The attribute eclipse4diac::core::TypeHash is empty, meaning that the type identity is not additionally secured at runtime.

State Overview

The component does not contain a state machine (ECC). Its behavior is completely deterministic and eventless – a state representation is therefore not required.

Application Scenarios

- **Signal Distribution**: An AUDI signal from a sensor or control unit must be forwarded in parallel to multiple consumers (actuators, displays, monitoring systems).
- **System Expansion**: Existing systems that provide a single AUDI signal are to be expanded with additional components without changing the source logic.
- **Test Setups**: A generated test signal is to be sent to multiple devices under test simultaneously.

Comparison with Similar Components

Component	Distribution	Adapter Type
AUDI_SPLIT_7_UNGATED	1 input ⟶ 7 outputs	Unidirectional AUDI
SPLIT_1_TO_2 (analog)	1 ⟶ 2	Any (generic)
AUDI_MERGE	Multiple Inputs ⟶ 1	Unidirectional AUDI

While AUDI_SPLIT_7_UNGATED is designed for the specific AUDI adapter type, generic split modules exist for other data formats. Its limitation to a fixed number of seven outputs distinguishes it from flexible splitters with a configurable number of outputs.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

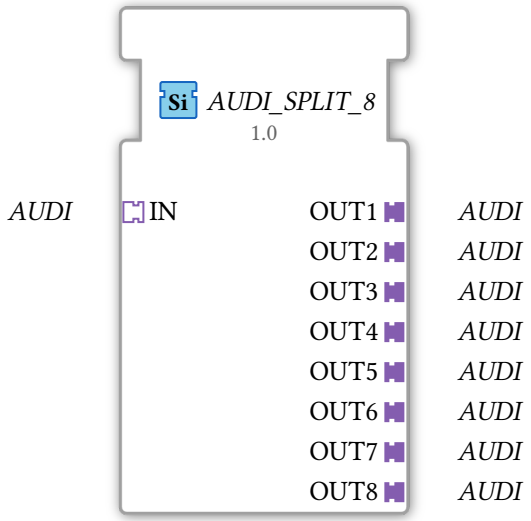
Conclusion

AUDI_SPLIT_7_UNGATED` is a simple yet effective module for distributing an AUDI signal to seven identical paths. Its generic design allows it to be reused in various contexts. The absence of events and internal logic makes it lightweight and reliable for pure signal distribution tasks.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AUDI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_8** is used to distribute an incoming unidirectional **AUDI** adapter signal to eight parallel outputs. It is implemented as a generic function block (FB) and allows for the simple forwarding of an adapter signal to up to eight receivers.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Direction	Name	Description
adapter::types::unidirectional::AUDI`	Socket (Input)	`IN`	Input adapter that receives the signal to be distributed.
adapter::types::unidirectional::AUDI`	Plug (Output)	`OUT1` .. `OUT8`	Eight output adapters, each providing the identical signal of the input.

Functionality

The module forwards the **AUDI** adapter signal present at socket `IN` unchanged to all eight plugs (`OUT1` to `OUT8`).

No processing, filtering, or buffering takes place – distribution is completely passive at the adapter level.

By using a generic type (see Technical Specifications), the function block can be flexibly used for various specific AUDI adapter implementations.

Technical Specifications

- **Generic Function Block**: The function block is declared as a generic type (`eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`).

This allows reuse for any AUDI adapter specialization without having to modify the function block itself.

- **No State Logic**: Since only adapters are used, the function block has neither events nor its own state machine.
- **Space Saving**: By combining eight outputs into a single function block, the network diagram becomes clearer than when using several simple split blocks.

State Overview

The function block has no internal state machine (no ECC).

The functionality is purely combinational: The signal at the input is permanently passed through to all outputs.

Application Scenarios

- **Signal broadcast** in control applications where a single AUDI adapter signal needs to be distributed to multiple independent receiver modules (e.g., smart sensors, actuators, or bus couplers).
- **Expanding the number of participants** in an existing AUDI wiring system without affecting the original signal.
- **Structured automation solutions** in agricultural machinery where adapters according to IEC 61499 are used for communication between components (e.g., in HR Agrartechnik GmbH systems).

Comparison with Similar Modules

- **Simple Split FB (e.g., SPLIT_2)**: Distributes a signal to two outputs – AUDI_SPLIT_8 offers eight outputs in a single module.
- **Data Distribution FB**: Some libraries provide generic distributors for data ports; this FB is specifically optimized for the AUDI adapter type and requires no additional data structure configuration.
- **Manual Parallel Connection**: Multiple AUDI socket-to-plug connections could achieve the same result, but are less organized and more prone to errors.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AUDI_SPLIT_8 is a compact, generic, specialized module for signal distribution at the adapter level.

It reduces modeling effort, improves the readability of control programs, and, due to its generic nature, can be flexibly deployed in various AUDI-based environments.

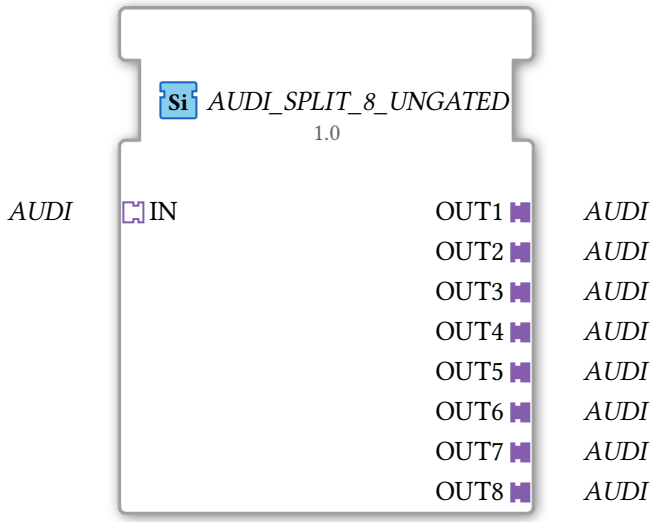
For applications requiring a 1:8 split of a unidirectional audio adapter, it represents an ideal and clean solution.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

AUDI_SPLIT_8_UNGATED

i UNGATED variant: This block is the ungated version of `AUDI_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_SPLIT_8_UNGATED** is used to distribute an incoming unidirectional **AUDI** adapter signal to eight parallel outputs. It is implemented as a generic function block (FB) and allows for the simple forwarding of an adapter signal to up to eight receivers.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Direction	Name	Description
adapter::types::unidirectional::AUDI`	Socket (Input)	`IN`	Input adapter that receives the signal to be distributed.
adapter::types::unidirectional::AUDI`	Plug (Output)	`OUT1` .. `OUT8`	Eight output adapters, each providing the identical signal of the input.

Functionality

The module forwards the **AUDI** adapter signal present at socket `IN` unchanged to all eight plugs (`OUT1` to `OUT8`).

No processing, filtering, or buffering takes place – distribution is completely passive at the adapter level.

By using a generic type (see Technical Specifications), the function block can be flexibly used for various specific AUDI adapter implementations.

Technical Specifications

- **Generic Function Block**: The function block is declared as a generic type (`eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`).

This allows reuse for any AUDI adapter specialization without having to modify the function block itself.

- **No State Logic**: Since only adapters are used, the function block has neither events nor its own state machine.
- **Space Saving**: By combining eight outputs into a single function block, the network diagram becomes clearer than when using several simple split blocks.

State Overview

The function block has no internal state machine (no ECC).

The functionality is purely combinational: The signal at the input is permanently passed through to all outputs.

Application Scenarios

- **Signal broadcast** in control applications where a single AUDI adapter signal needs to be distributed to multiple independent receiver modules (e.g., smart sensors, actuators, or bus couplers).
- **Expanding the number of participants** in an existing AUDI wiring system without affecting the original signal.
- **Structured automation solutions** in agricultural machinery where adapters according to IEC 61499 are used for communication between components (e.g., in HR Agrartechnik GmbH systems).

Comparison with Similar Modules

- **Simple Split FB (e.g., SPLIT_2)**: Distributes a signal to two outputs – AUDI_SPLIT_8_UNGATED offers eight outputs in a single module.
- **Data Distribution FB**: Some libraries provide generic distributors for data ports; this FB is specifically optimized for the AUDI adapter type and requires no additional data structure configuration.
- **Manual Parallel Connection**: Multiple AUDI socket-to-plug connections could achieve the same result, but are less organized and more prone to errors.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AUDI_SPLIT_8_UNGATED is a compact, generic, specialized module for signal distribution at the adapter level.

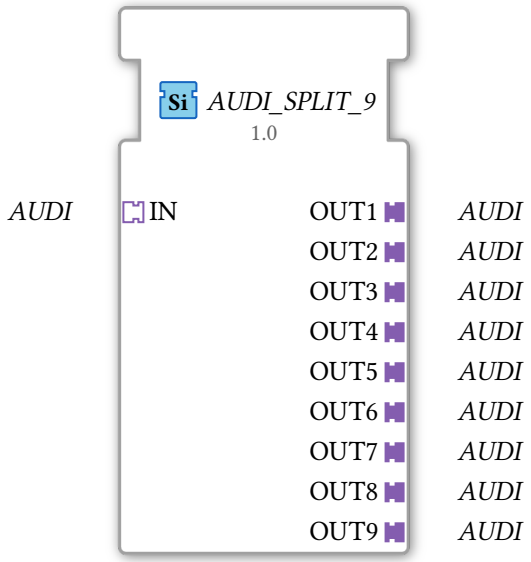
It reduces modeling effort, improves the readability of control programs, and, due to its generic nature, can be flexibly deployed in various AUDI-based environments.

For applications requiring a 1:8 split of a unidirectional audio adapter, it represents an ideal and clean solution.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

AUDI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_SPLIT_9` is used to distribute an incoming AUDI signal via a socket to nine identical AUDI plugs. It implements a 1-to-9 split functionality, where all outputs always carry the same value as the input. The block is defined as a generic FB (GenericClassName `GEN_AUDI_SPLIT`) and enables the easy duplication of AUDI data streams within a 4diac IDE application.

Interface Structure

Event Inputs

No event inputs are available. Data transmission occurs exclusively via the adapter interfaces.

Event Outputs

No event outputs are available. The output signals are provided directly via the adapter plugs.

Data Inputs

No direct data inputs are available. Input data is read via the socket adapter `IN`.

Data Outputs

No direct data outputs are available. Output data is provided via the nine plug adapters `OUT1` to `OUT9`.

Adapters

Adapter	Type	Direction	Description
`IN`	`adapter::types::unidirectional::AUDI`	Socket (Input)	Receives the AUDI signal to be distributed.
`OUT1` … `OUT9`	`adapter::types::unidirectional::AUDI`	Plug (Output)	Nine outputs that replicate the value of `IN`. Each output is identical and independent.

Functionality

This component establishes a passive coupling between the socket `IN` and the nine plugs `OUT1` to `OUT9`. Whenever the socket receives a valid AUDI data packet, it is forwarded unchanged to all nine plugs. No data modification, buffering, or time delay occurs. The component has no inherent behavior – it functions purely as a wiring aid for signal distribution.

Technical Features

- **Generic Implementation:** The function block (FB) carries the attribute `eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`, which allows it to be recognized as an instance of a generic split block and displayed correctly in tools with generic type support.
- **No Event Control:** Since no events are present, data transmission occurs purely via the adapter mechanisms of the target system (e.g., continuously or cyclically, depending on the runtime environment).
- **Scalability:** The number of outputs is fixed at nine. For other numbers, separate FB types (e.g., `AUDI_SPLIT_2`, `AUDI_SPLIT_4`, etc.) must be defined, or a parameterizable function block must be used.

State Overview

The function block has no internal state. Its behavior is completely static and defined solely by the forwarding of the signal. A state machine does not exist.

Application Scenarios

- **Distributing a measurement signal** to multiple parallel evaluation or control units.
- **Splitting a control command** to multiple actuators, provided they use the same AUDI protocol.
- **Test and simulation environments** where a generated signal needs to be sent to multiple observers (e.g., logging, visualization, control) simultaneously.
- **Replacing multiple copies** of a source function block when the source exists only once.

Comparison with Similar Function Blocks

Compared to a general-purpose `AUDI_MERGE` (which combines multiple inputs into one output) or a `AUDI_SELECT` (which switches between multiple inputs), the Split function block offers the exact opposite function: duplication instead of merging or selection. Compared to a custom, software-implemented split (e.g., by connecting an output multiple times in the application), the function block (FB) ensures a clean, typed, and reusable interface and avoids potential multiple connection errors in the editor.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AUDI_SPLIT_9` is a simple yet useful function block for duplicating audio signals. Its implementation as a generic type and the clear separation of input and nine outputs facilitate the development of modular automation solutions. Due to its static nature, it is particularly suitable for data flow-oriented applications where event control is not required.

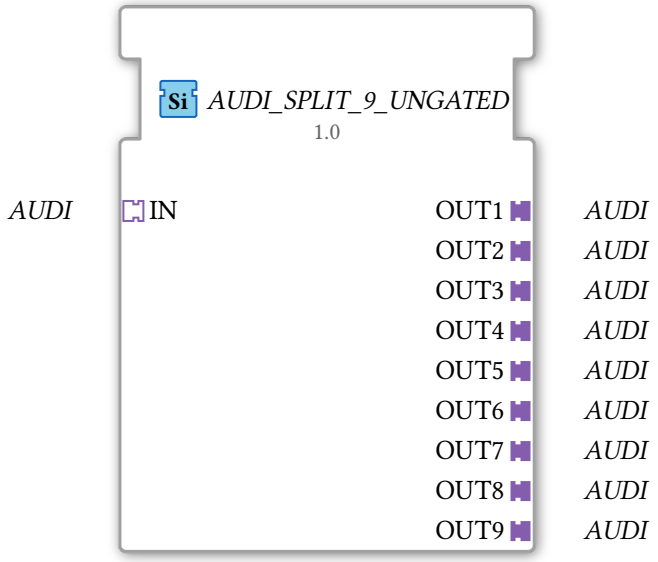
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AUDI_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_SPLIT_9_UNGATED` is used to distribute an incoming AUDI signal via a socket to nine identical AUDI plugs. It implements a 1-to-9 split functionality, where all outputs always carry the same value as the input. The block is defined as a generic FB (GenericClassName `GEN_AUDI_SPLIT`) and enables the easy duplication of AUDI data streams within a 4diac IDE application.

Interface Structure

Event Inputs

No event inputs are available. Data transmission occurs exclusively via the adapter interfaces.

Event Outputs

No event outputs are available. The output signals are provided directly via the adapter plugs.

Data Inputs

No direct data inputs are available. Input data is read via the socket adapter `IN`.

Data Outputs

No direct data outputs are available. Output data is provided via the nine plug adapters `OUT1` to `OUT9`.

Adapters

Adapter	Type	Direction	Description
`IN`	`adapter::types::unidirectional::AUDI`	Socket (Input)	Receives the AUDI signal to be distributed.
`OUT1` … `OUT9`	`adapter::types::unidirectional::AUDI`	Plug (Output)	Nine outputs that replicate the value of `IN`. Each output is identical and independent.

Functionality

This component establishes a passive coupling between the socket `IN` and the nine plugs `OUT1` to `OUT9`. Whenever the socket receives a valid AUDI data packet, it is forwarded unchanged to all nine plugs. No data modification, buffering, or time delay occurs. The component has no inherent behavior – it functions purely as a wiring aid for signal distribution.

Technical Features

- **Generic Implementation:** The function block (FB) carries the attribute `eclipse4diac::core::GenericClassName = 'GEN_AUDI_SPLIT'`, which allows it to be recognized as an instance of a generic split block and displayed correctly in tools with generic type support.
- **No Event Control:** Since no events are present, data transmission occurs purely via the adapter mechanisms of the target system (e.g., continuously or cyclically, depending on the runtime environment).
- **Scalability:** The number of outputs is fixed at nine. For other numbers, separate FB types (e.g., `AUDI_SPLIT_2`, `AUDI_SPLIT_4`, etc.) must be defined, or a parameterizable function block must be used.

State Overview

The function block has no internal state. Its behavior is completely static and defined solely by the forwarding of the signal. A state machine does not exist.

Application Scenarios

- **Distributing a measurement signal** to multiple parallel evaluation or control units.
- **Splitting a control command** to multiple actuators, provided they use the same AUDI protocol.
- **Test and simulation environments** where a generated signal needs to be sent to multiple observers (e.g., logging, visualization, control) simultaneously.
- **Replacing multiple copies** of a source function block when the source exists only once.

Comparison with Similar Function Blocks

Compared to a general-purpose `AUDI_MERGE` (which combines multiple inputs into one output) or a `AUDI_SELECT` (which switches between multiple inputs), the Split function block offers the exact opposite function: duplication instead of merging or selection. Compared to a custom, software-implemented split (e.g., by connecting an output multiple times in the application), the function block (FB) ensures a clean, typed, and reusable interface and avoids potential multiple connection errors in the editor.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUDI_SPLIT_9_UNGATED` is a simple yet useful function block for duplicating audio signals. Its implementation as a generic type and the clear separation of input and nine outputs facilitate the development of modular automation solutions. Due to its static nature, it is particularly suitable for data flow-oriented applications where event control is not required.

🌐 Related topic subpages on ms-muc-docs.de

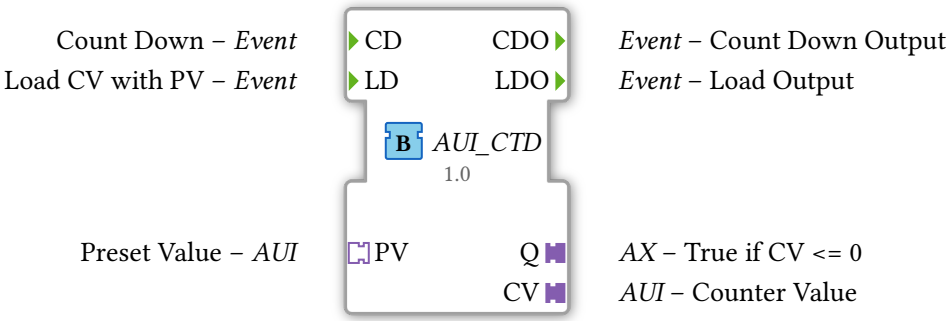
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

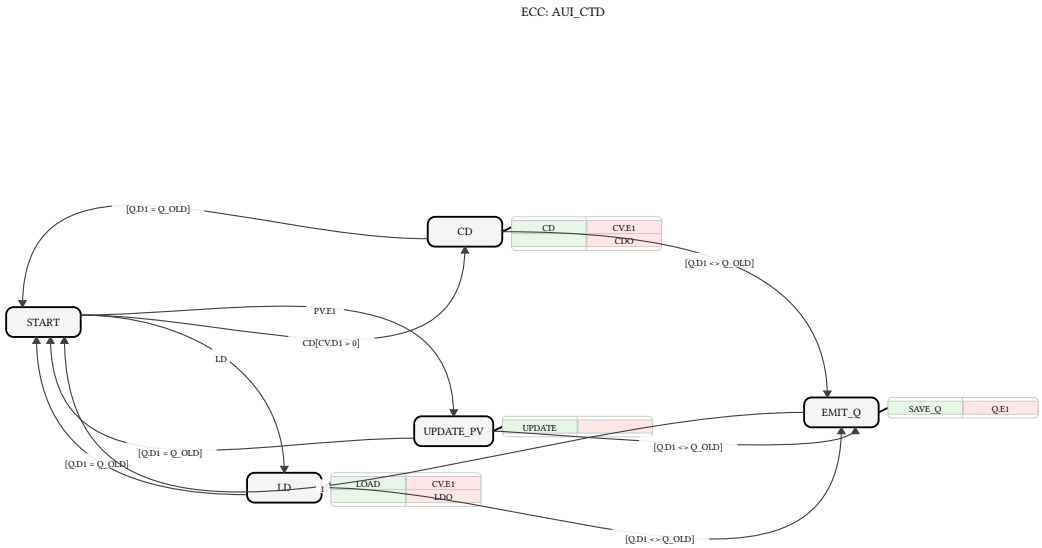
]

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUI_CTD





• — ○ • — ○ • — ○ •

Introduction

The **AUI_CTD** is an event-driven down counter with adapter interfaces. It features **On-Change triggering**: The output signal **Q** is only updated when its logical state actually changes. This avoids unnecessary events and increases efficiency in event-driven systems.

Interface Structure

Event Inputs

Event	Description
CD	Count Down – reduces the counter value by 1.
LD	Load – loads the counter value **CV** with the preset value **PV**.

Event Outputs

Event	Description
CDO	Count Down Output – output after each CD step.
LDO	Load Output – output after a successful load.

Data Inputs

The FB has no direct data inputs. Instead, values are provided via the adapter interface **PV**.

Data Outputs

The FB has no direct data outputs. The current values of **CV** and **Q** are output via the respective adapter ports.

Adapter

Port	Type	Direction	Description

Q	`adapter::types::unidirectional::AX` (BOOL)	Plug	Status output: `TRUE`, if **CV** ≤ **0**.
CV	`adapter::types::unidirectional::AUI` (INT)	Plug	Current counter value.
PV	`adapter::types::unidirectional::AUI` (INT)	Socket	Preset value loaded into **CV** during the LD event.

Functionality

1. **Count Down (CD)**

Each **CD** event decrements **CV** by 1 and sets **Q** to `TRUE` if **CV** ≤ **0**. **CDO** is then output.

1. **Load (LD)**

An **LD** event copies the value from **PV** to **CV**. **Q** is updated accordingly (TRUE if **CV** ≤ **0**). Then, **LDO** is output.

1. **On-Change Triggering**

The function block (FB) compares the current **Q** value with the stored predecessor value **Q_OLD**. The adapter event **Q.E1** is only triggered if **Q** has changed. Otherwise, the FB returns directly to its initial state. This prevents repeated identical messages.

1. **PV Changes**

When the **PV** adapter reports (event **PV.E1**), **Q** is recalculated (using the **UPDATE** algorithm). Here too, **Q** is only output when there is a state change.

Technical Features

- **On-Change Triggering** – Reduces event load by activating the **Q** output only when changes actually occur.
- **Adapter Interface** – Enables loose coupling and reuse in complex control structures.
- **Initial Value** – **Q_OLD** is initialized with `FALSE`, so that a change is detected on the first iteration if **CV** is already ≤ 0.

State Overview

The process is controlled by the following ECC states:

State	Description	Actions
START	Waits for incoming events (CD, LD, PV.E1).	–
CD	Executes the countdown algorithm.	`CD (CV--, Q aktualisieren), CV.E1, CDO`
LD	Loads **PV** into **CV**.	`LOAD (CV:=PV; Q aktualisieren), CV.E1, LDO`
UPDATE_PV	Recalculates **Q** when **PV** changes.	`UPDATE (Q aus CV berechnen)`
EMIT_Q	Saves the current **Q** value and triggers **Q.E1**.	`SAVE_Q (Q_OLD:=Q), Q.E1`

Transitions:

- After **CD**, **LD**, or **UPDATE_PV**, the system only switches to **EMIT_Q** if `Q.D1 ≠ Q_OLD` is present.
- If the values are equal (`Q.D1 = Q_OLD`), the system immediately returns to **START** without triggering **Q.E1**.
- From **EMIT_Q**, the system always switches back to **START**.

Application Scenarios

- **Material Counter** in Conveyor Systems: **CD** is triggered for each part that leaves the station. As soon as **CV** ≤ **0**, a replenishment request is sent.

- **Cycle Control** for Repeated Processes: The counter runs from a starting value (loaded via **LD**) to zero and then triggers an action.
- **Event-optimized systems** with many parallel counters: Thanks to on-change triggering, only relevant status changes are reported.

Comparison with similar function blocks

Function block	Special feature	Difference to AUI_CTD
E_CTD	Standard down counter without adapters	Uses direct data inputs/outputs and no adapters. No on-change triggering.
E_CTUD	Up/down counter	Offers both counting directions, but without adapters and without on-change.
AUI_CTU	Up counter with adapters	Counts upwards, analog adapter structure, also on-change.

The **AUI_CTD** fills the gap of a specialized down counter with more modern adapter concepts and reduced event volume.

Conclusion

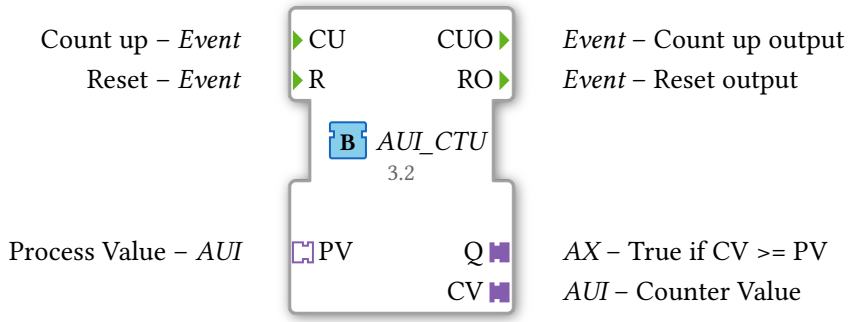
The **AUI_CTD** is a powerful down counter for IEC 61499-compliant controllers. Its combination of adapter interfaces and on-change triggering makes it particularly suitable for modular, resource-efficient automation solutions. Its behavior is deterministic and can be easily traced via the ECC. This makes it an ideal choice for applications requiring reliable and efficient counter functionality with minimized event communication.

Related topic subpages on ms-muc-docs.de

- E_CTU Event Counter block on ms-muc-docs.de

]

AUI_CTU



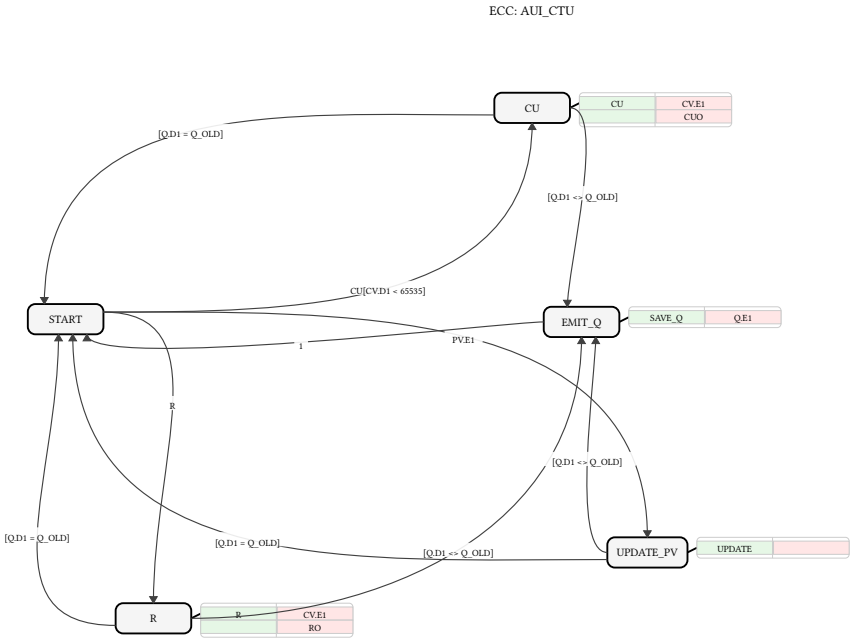


Image of function block not available

• — ○ • — ○ • — ○ •

Introduction

The AUI_CTU is an event-driven up counter with an adapter interface. It increments the counter on every positive edge event at input `CU` and outputs the count via adapter `CV`. The output `Q` indicates whether the counter value (`CV`) has reached or exceeded the set limit (`PV`). The special feature of this implementation is the “On-Change” triggering: The event on the adapter `Q.E1` is only triggered if the logical state of `Q` actually changes. This reduces unnecessary events in subsequent processing.

Interface Structure

Event Inputs

Name	Type	Comment
`CU`	Event	Count Up
`R`	Event	Reset

Event Outputs

Name	Type	Comment
`CUO`	Event	Output after successful increment
`RO`	Event	Output after successful reset

Data Inputs

Direct data inputs are not available. The limit value (`PV`) is provided via an adapter socket.

Data Outputs

Direct data outputs are not available. The current counter value (`CV`) and the comparison status (`Q`) are output via adapter plugs.

Adapters

Type	Direction	Name	Description
adapter::types::unidirectional::AX	Plug (Output)	Q	Outputs TRUE if CV >= PV, otherwise FALSE. The event Q.E1 is only sent on state changes.
adapter::types::unidirectional::AUI	Plug (Output)	CV	Returns the current counter value (unsigned integer). The event CV.E1 is triggered after each increment or reset.
adapter::types::unidirectional::AUI	Socket (Input)	PV	Receives the threshold value of type UINT. Changing this value automatically recalculates Q.

Functionality

This function block implements a finite state machine (ECC) with the following algorithms:

- **CU**: Increments the internal counter CV by 1 and then calculates Q := (CV >= PV).
- **R**: Sets CV to 0 and recalculates Q.
- **UPDATE**: Calculates Q from the current CV and the (modified) PV, without changing the counter.
- **SAVE_Q**: Stores the current value of Q in the internal variable Q_OLD for the next state change detection.

Process:

1. **Event CU** (and CV < 65535): Transition to state CU. The counter is incremented, Q is recalculated, and CV.E1 and the event CUO are output.

- If Q has changed compared to the last stored value (Q_OLD), the state changes to EMIT_Q.
- Otherwise, the function block returns to state START.

1. **Event R**: Transition to state R. The counter is reset, Q is recalculated, and CV.E1 and RO are output. Then, analogous to CU, a decision is made whether to reach EMIT_Q or return to START.
2. **Adapter event PV.E1** (limit change): Transition to state UPDATE_PV. The algorithm UPDATE recalculates Q. Here too, the state EMIT_Q is only traversed when Q changes.
3. **State EMIT_Q**: Executes the algorithm SAVE_Q (stores the new Q value in Q_OLD) and sends the event Q.E1. Afterward, the function block always returns to START.
4. **State EMIT_Q**: Executes the algorithm SAVE_Q (stores the new Q value in Q_OLD) and sends the event Q.E1.

The function block then always returns to START.

Technical Features

- **On-Change Triggering for Adapter Q**: The event Q.E1 is only triggered when the logical value of Q (true/false) changes. This is achieved through the internal status variable Q_OLD. This prevents unnecessary event flooding in the application.
- **Automatic Response to Threshold Change**: If the threshold value supplied via socket PV changes, Q is immediately recalculated and output upon change. The counter itself remains unaffected.
- **Overflow Protection**: The counter is only incremented if CV is less than 65535 (maximum value of UINT). Further incrementing beyond this value is not possible.
- **Adapter-Based**: Instead of classic direct inputs/outputs, all values are exchanged via standardized adapters. This allows for easy integration into existing 4diac adapter concepts.

State Overview

State	Description	Actions	Outgoing Transitions
START	Idle state, waiting for events	–	CU ⟶ CU, R ⟶ R, PV.E1 ⟶ UPDATE_PV
CU	Count up	CU algorithm, send CV.E1 and CUO	[Q != Q_OLD] ⟶ EMIT_Q, [Q == Q_OLD] ⟶ START

R	Reset	R algorithm, send CV.E1 and RO	[Q != Q_OLD] ⟶ EMIT_Q, [Q == Q_OLD] ⟶ START
UPDATE_PV	Recalculation after PV change	UPDATE algorithm	[Q != Q_OLD] ⟶ EMIT_Q, [Q == Q_OLD] ⟶ START
EMIT_Q	Emission of the Q event	SAVE_Q algorithm, send Q.E1	1 ⟶ START

The transitions are triggered by conditions:

- CU[CV.D1 < 65535]: CU event if the counter is not at its maximum.
- R: R event, always possible.
- PV.E1: Event from the PV adapter.
- [Q.D1 <> Q_OLD] / [Q.D1 = Q_OLD]Comparison of the current Q with the stored previous value.

Application Scenarios

- **Event Counter with Threshold Monitoring**: Counting pulses (e.g., workpieces on a conveyor belt) and triggering an action as soon as a specific threshold is reached.
- **Level Monitoring**: Used as a limit switch that transmits a status when a setpoint is exceeded.
- **State-Dependent Controls**: E.g., in batch processing where a process step should be triggered after a certain number of steps.
- **Systems with Dynamic Threshold**: The threshold can be changed during operation via the adapter PV; the function block reacts automatically.

Comparison with Similar Function Blocks

Feature	AUI_CTU	Standard CTU (IEC 61131-3)	CTUD (Up/Down Counter)
Interface	Adapter-based	Direct Inputs/Outputs	Direct Inputs/Outputs
Event on Q Change	Yes (On-Change)	No (Always on Counting Event)	No
Response to PV Change	Automatic	Not Provided	Not Provided
Overflow Protection	Yes (max. 65535)	Yes, via configuration	Yes
Down Count	No	No	Yes

The AUI_CTU is specifically optimized for event-driven systems with an adapter concept. On-change triggering and automatic PV recalculation are key differences compared to traditional meters.

Conclusion

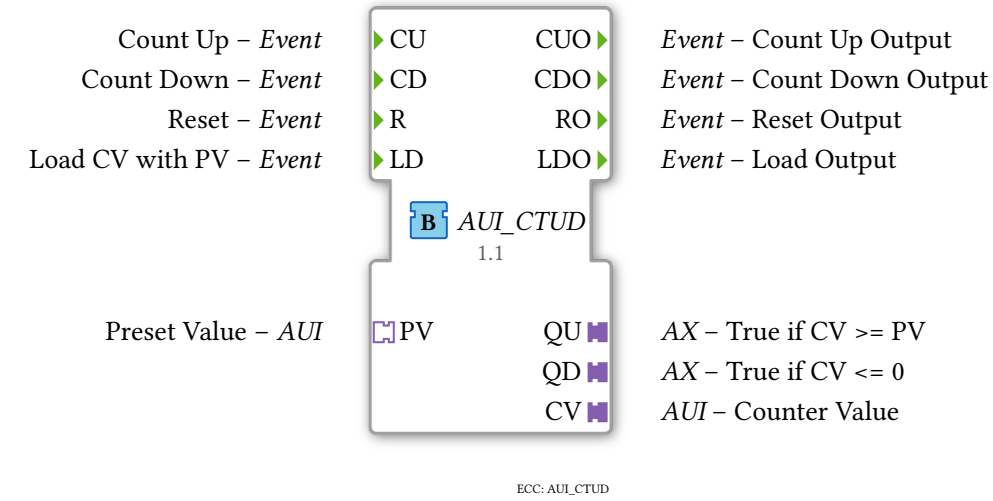
The AUI_CTU is a modern, adapter-based up-meter that stands out for its efficient event handling. Thanks to on-change triggering for the status Q, unnecessary messages in the system are avoided. Support for dynamic limits via the PV adapter increases flexibility in control applications. The module is robust against overflows and offers a clear, state-driven flow structure. It is particularly well-suited for distributed automation systems where loose coupling and event-driven communication are paramount.

🌐 Related topic subpages on ms-muc-docs.de

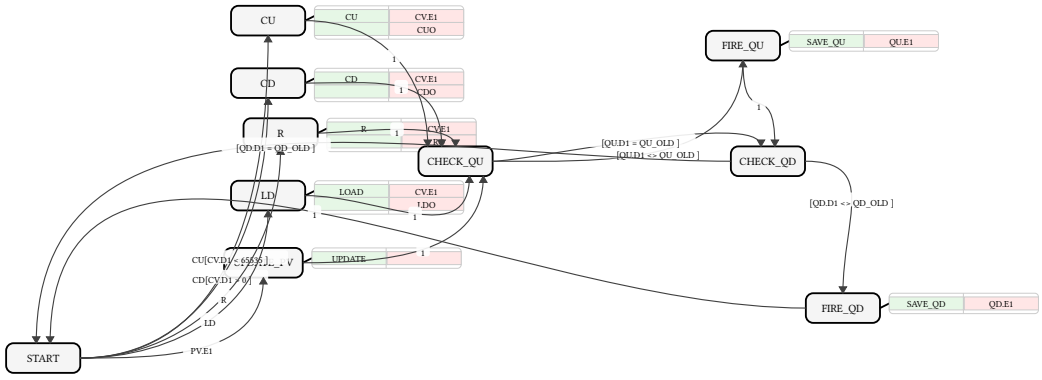
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUI_CTUD



ECC: AUI_CTUD



• – ○ • – ○ • – ○ •

Introduction

The function block **AUI_CTUD** is an event-driven up/down counter in the adapter version. It is based on the standard function block E_CTUD and extends it with “On-Change” triggering for the output states. The counter value (CV), the preset value (PV), and the comparison results (QU, QD) are exchanged with other function blocks via the adapter interfaces. The function block is intended for use in IEC 61499 applications where loose coupling via adapters is desired.

Interface Structure

Event Inputs

Event	Description
CU	Increment counter by 1 (Count Up)
CD	Decrease counter by 1 (Count Down)

R	Reset counter to 0 (Reset)
LD	Load counter with preset value (PV) (Load)

Event Outputs

Event	Description
CUO	Acknowledgement of successful count-up operation
CDO	Acknowledgement of successful count-down operation
RO	Acknowledgement of successful reset
LDO	Acknowledgement of successful load operation

Data Inputs

This function block does not have direct data inputs. The preset value (PV) is provided via a **socket adapter**:

Adapter (Socket)	Type	Description
PV	`adapter::types::unidirectional::AUI`	Preset value (preselection) for the counter

Data Outputs

This function block does not have direct data outputs. The results are output via **Plug Adapters**:

Adapter (Plug)	Type	Description
QU	`adapter::types::unidirectional::AX`	True if the current counter value is ≥ PV
QD	`adapter::types::unidirectional::AX`	True if the current counter value is ≤ 0
CV	`adapter::types::unidirectional::AUI`	Current Counter Value (Unsigned Integer)

Adapters

This function block defines three plug adapters (QU, QD, CV) and one socket adapter (PV). The adapters are of type `adapter::types::unidirectional` and enable loose coupling between the function blocks – values are exchanged via events (e.g., `CV.E1`, `QU.E1`).

Functionality

This function block operates as an event-driven counter with a resolution of 0 to 65535 (maximum value for an unsigned 16-bit integer). The following sequence of steps is executed:

1. An **input event** (CU, CD, R, or LD) triggers a transition from the start state to the corresponding processing state.
2. **Calculation**: In each algorithm, the internal counter (`CV.D1`) is updated, and the output adapters QU and QD are recalculated:

- **CU**: `CV.D1 := CV.D1 + 1` (only if the previous value < 65535)
- **CD**: `CV.D1 := CV.D1 - 1` (nur, wenn der Wert vorher > was 0)
- **R**: `CV.D1 := 0`
- **LD**: `CV.D1 := PV.D1`
- Subsequently, `QU.D1 := (CV.D1 >= PV.D1)` and `QD.D1 := (CV.D1 <= 0)` are set.

1. **Counter Value Output**: After each value change, the event `CV.E1` is sent to propagate the new counter value via the adapter.

CU: ** ... 4. **On-Change Triggering**:

- After each counting operation or after a change to the PV (event from the PV adapter), the state of QU and QD is checked.

- The internal variables `QU_OLD` and `QD_OLD` store the previous state. Only if the value has changed is the corresponding adapter event (`QU.E1` or `QD.E1`) triggered.
- This behavior prevents unnecessary event flooding with constant threshold values.

1. **Processing of PV Changes**: An incoming event on the PV adapter (`PV.E1`) results in the state `UPDATE_PV`, in which only the comparison results are recalculated (without changing the counter). The on-change check is also performed afterward.

Technical Features

- **On-Change Triggering** (Change Detection): The module sends events to the output adapters only if the logical state has actually changed compared to the last iteration. This reduces the bus load and prevents infinite loops in cyclic systems.
- **Adapter-Based Communication**: All input and output values are exchanged via adapters (plug/socket), enabling flexible interconnection without fixed data connections. The counter value (CV) and the comparison results (QU, QD) are provided as plug adapters, while the preset value (PV) is supplied as a socket from external modules.
- **State Overflow Protection**: During count-up, the module checks whether the current value is less than 65535; during count-down, it checks whether it is greater than 0. This prevents overflows and underflows.
- **Two internal state variables**: `QU_OLD` and `QD_OLD` serve as shadow registers for change detection.
- **Extended ECC**: After each counting operation, the function block cycles through a sequential chain of states (e.g., CU → CHECK_QU → CHECK_QD → START) to ensure change detection.

State Overview

State	Description
START	Waiting for an input event (CU, CD, R, LD) or a PV change
CU	Increment counter and output new value via CV adapter
CD	Decrement counter and output new value via CV adapter
R	Reset counter to 0 and output new value via CV adapter
LD	Load meter with PV and output new value via CV adapter
UPDATE_PV	Recalculate QU and QD after PV change (without meter change)
CHECK_QU	Check if QU has changed (comparison with QU_OLD)
FIRE_QU	If QU has changed: Update QU_OLD and send event to QU adapter
CHECK_QD	Check if QD has changed (comparison with QD_OLD)
FIRE_QD	If QD has changed: Update QD_OLD and send event to QD adapter

The state transitions are controlled by the events and conditions `[QU.D1 <> QU_OLD]` and `[QD.D1 <> QD_OLD]`. After change detection is complete, the function block returns to the START state.

Application Scenarios

- **Counters with External Preset**: In a system, the preset value (PV) can be dynamically set by a higher-level system or an HMI via the PV adapter. The function block then counts events and reports when the thresholds are reached or fallen below via the QU and QD adapters.
- **Direction-Dependent Counters**: By using CU and CD, a forward/reverse counter can be implemented, e.g., for position detection or inventory counting.
- **Event-Driven Limit Monitoring**: The on-change triggering of QU and QD is ideally suited for sending a discrete signal (e.g., "fill level reached") only when a state changes – similar to an edge detector.

– **Modular Control Architecture**: Thanks to the adapter interface, the function block can be integrated into an existing IEC 61499 application without fixed data port wiring. The logical connection is established solely via the adapter runtime environment.

Comparison with Similar Function Blocks

The standard function block **E_CTUD** (from the IEC 61499 library) also offers an up/down counter with event control, but with the following features:

Feature	**E_CTUD** (Standard)	**AUI_CTUD** (Adapter Version)
Interface	Fixed event and data ports (e.g., CV as BOOL)	Loose coupling via adapter (plug/socket)
Output: Counter value	Data port: CV (INT/UDINT)	Adapter CV (Type AUI)
Limit output	Bool ports QU, QD	Adapter QU, QD (Type AX)
Event on change	No change detection; QU/QD are calculated with each count	On-change triggering: Event only on state change
Feedback on actions	No dedicated feedback events	CUO, CDO, RO, LDO as confirmation events

The AUI_CTUD is therefore more flexible in its interconnection and reduces the event load in distributed systems. For simple applications without adapters, however, the standard E_CTUD may suffice.

Conclusion

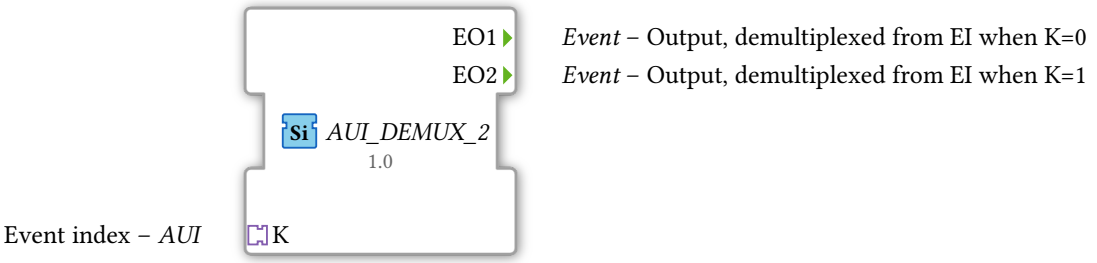
The **AUI_CTUD** function block represents a modern, adapter-based variant of an up/down counter. The on-change triggering of the limits and the feedback via dedicated event outputs are particularly noteworthy. The use of adapter interfaces enables loose coupling and reusability in a wide variety of control projects. Integrated overflow/underflow protection and the clean separation of counting and evaluation logic make it a robust component for industrial automation applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter component on ms-muc-docs.de

]

AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

AUI_DEMUX_2 is a generic event demultiplexer function block that routes an incoming event to one of two output event ports based on an index value received through an AUI (Adapter Unified Interface) adapter. It is the adapter-based variant of the standard E_DEMUX_2 block, replacing the conventional event input and data input pair with a single unidirectional AUI adapter. This design allows the event trigger and the selection index to be delivered together as a cohesive bundle, simplifying composition and promoting type-safe connections in IEC 61499 systems.

Interface Structure

The block features a minimal interface consisting solely of two event outputs and one adapter socket. No separate event inputs or data inputs/outputs are present, as all incoming information arrives via the AUI adapter.

Event Inputs

None. The triggering event is received through the AUI adapter socket “K”.

Event Outputs

Name	Type	Comment
EO1	Event	Output, demultiplexed from EI when K=0
EO2	Event	Output, demultiplexed from EI when K=1

Data Inputs

None. The selection value is conveyed through the adapter.

Data Outputs

None.

Adapters

Name	Type	Direction	Comment
K	adapter::types::unidirectional::AUI	Socket	Event index used for demultiplexing

Functionality

The block operates as a two-way event demultiplexer. When an event arrives through the AUI adapter K, the associated index value carried by the adapter is evaluated:

- If the index value equals 0, the event is propagated to output **EO1**.
- If the index value equals 1, the event is propagated to output **EO2**.

The AUI adapter bundles an event trigger with a data value, enabling the selection criterion to be delivered atomically with the event itself. This ensures that the demultiplexing decision is always based on the most current index value synchronized with the triggering event.

Technical Features

- **Generic FB**: Declared with the generic class name `GEN_E_DEMUX`, allowing the compiler/runtime to instantiate it with the appropriate underlying implementation.
- **Adapter-based input**: The single AUI socket replaces the traditional separate event input (EI) and data input (K) of the standard E_DEMUX_2 block.
- **Type-safe connections**: The adapter type enforces correct wiring at design time, preventing mismatched event/data pairs.
- **Two-way demultiplexing**: Supports exactly two distinct output paths based on a binary index.
- **Unidirectional adapter**: The AUI (adapter types::unidirectional::AUI) socket is of the unidirectional kind, meaning data flows in one direction only (into the FB).

State Overview

AUI_DEMUX_2 does not maintain internal state. It is a purely combinatorial event router:

- On adapter event arrival with index = 0 ⟶ EO1 fires.
- On adapter event arrival with index = 1 ⟶ EO2 fires.

No internal variables, timers, or memory are used; the behavior is fully deterministic and stateless.

Application Scenarios

- **Event routing in automation chains**: Distributing a single trigger to different processing branches depending on a mode or selector value.
- **Mode switching**: Selecting between two alternative algorithms or control strategies based on a command index.
- **Multiplexed communication**: Demultiplexing incoming events from a unidirectional AUI-based communication channel into separate functional paths.
- **Structured composition**: Used in systems where adapters are preferred over raw event/data pairs for cleaner, more maintainable type-safe connections.

Comparison with Similar Blocks

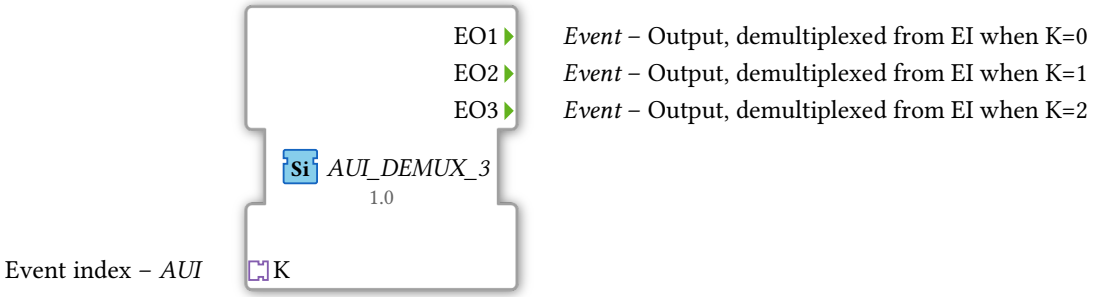
Feature	AUI_DEMUX_2	E_DEMUX_2 (standard)
Input style	Single AUI adapter socket	Separate EI (event) + K (data)
Selector	Via adapter data	Via data input K
Outputs	EO1, EO2	EO1, EO2
Generic nature	Yes (GEN_E_DEMUX)	No (fixed type)
Wiring complexity	Reduced (one connection)	Higher (two connections)
Type safety	Enhanced via adapter contract	Manual, type-checked per pin

The primary advantage of AUI_DEMUX_2 over E_DEMUX_2 is the consolidation of event and selector into a single adapter interface, which reduces wiring overhead and enforces a consistent data/event pairing at the type level.

Conclusion

AUI_DEMUX_2 provides a clean, adapter-based solution for two-way event demultiplexing in IEC 61499 systems. By leveraging the AUI adapter pattern, it simplifies interface design, improves type safety, and reduces connection count compared to its classical counterpart. It is particularly well suited for modern, modular automation architectures where adapters are the preferred means of interconnecting function blocks.

AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

The `AUI_DEMUX_3` is an event demultiplexer function block that routes an incoming event to one of three possible event outputs based on a select index. Unlike a conventional demultiplexer (e.g., `E_DEMUX`), the selection information is not provided through a separate data input but via an AUI (Adapter for Unified Interaction) socket. This allows the select value and the triggering event to be delivered together through a single adapter interface, simplifying wiring and enabling a generic, reusable pattern for event routing.

This block is designed for use in IEC 61499-1 Annex A compliant systems and is intended for scenarios where an event needs to be distributed to different downstream function blocks depending on a runtime-available code or identifier.

Interface Structure

Event Inputs

None. The demultiplexing trigger is received via the AUI adapter socket `K`. The adapter encapsulates both the event that initiates the demultiplexing and the index value that determines which output is activated.

Event Outputs

Output	Type	Comment
`EO1`	Event	Output event when the select value `K` equals `0`

EO2	Event	Output event when the select value K equals 1
EO3	Event	Output event when the select value K equals 2

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Comment
K	adapter::types::unidirectional::AUI	Socket	Provides the select index and the triggering event. The AUI adapter typically carries an event and an integer data value that is used to select one of the three outputs.

Functionality

The AUI_DEMUX_3 block waits for an event to arrive through the adapter socket K. When that event occurs, the block reads the integer value associated with the adapter. The value is used as an index to determine which of the three event outputs shall be issued:

- If the value is 0, EO1 is triggered.
- If the value is 1, EO2 is triggered.
- If the value is 2, EO3 is triggered.

If the value is not one of these three valid indices, no output event is generated (the event is ignored). The block operates in a purely event-driven manner; it does not store any internal state or maintain the selection between invocations.

Because the select index and the triggering event are delivered together via the adapter, the block can be used as a building block for more complex event-routing networks without requiring additional data connections or manual synchronization.

Technical Features

- **Adapter-Based Input**: The select index and event are packaged into an AUI adapter, promoting a clean interface and reducing the number of separate pins.
- **Generic Implementation**: The block is annotated as a generic function block (eclipse4diac::core::GenericClassName = 'GEN_E_DEMUX'). This indicates that the same logic can be instantiated with different adapters or select ranges, although the current façade is fixed to three outputs.
- **No Internal State**: The block is stateless; each invocation is independent and does not depend on previous calls.
- **Standard Compliance**: Conforms to the IEC 61499-1 Annex A event-driven model.

State Overview

AUI_DEMUX_3 does not maintain any internal state machine. Its behavior is entirely combinational in the event domain: a single event arrives through the adapter, and based on the accompanying data value, exactly one of the three output events is emitted. There are no idle⟶active transitions or persistent memory elements. This makes the block easy to reason about and suitable for high-throughput event-handling applications.

Application Scenarios

- **Event Routing**: Distributing a single event to one of several processing units based on a runtime-selected channel ID.
- **Mode Selection**: Triggering different control actions depending on a mode number provided by a supervisor or a higher-level orchestration block.

- **Protocol Decoding**: In communication stacks, an incoming packet can be demultiplexed to the appropriate handler based on a header field.
- **Test and Simulation**: As a generic demultiplexer, it can be used to simulate different branches of an application by injecting events with configurable indices.

Comparison with Similar Blocks

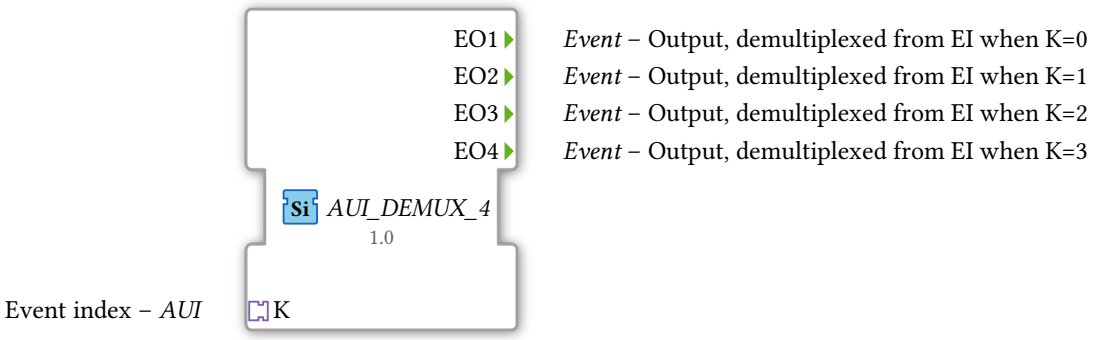
Block	Input Type	Number of Outputs	Select Mechanism
`E_DEMUX`	Separate event input `EI` and data input `K`	Variable (2, 3, ...)	Standard data input
`AUI_DEMUX_3`	Adapter socket `K` (combines event + data)	Fixed to 3	AUI adapter

The primary difference is the interface. `AUI_DEMUX_3` consolidates the trigger and the selection into a single adapter, which can simplify system wiring and improve modularity. It also imposes a fixed number of outputs (here, three), whereas a generic `E_DEMUX` might be parameterized. However, the underlying demultiplexing behavior—selecting exactly one output based on an index—is identical.

Conclusion

The `AUI_DEMUX_3` function block provides a concise, adapter-based solution for event demultiplexing in IEC 61499 applications. By integrating the select index and the triggering event into a single AUI socket, it reduces connection complexity and enhances reusability. Its stateless, event-driven design fits well into modern automation and distributed control scenarios, and its generic annotation ensures that it can be adapted to various use cases. For systems requiring a clean separation between event triggering and routing logic, `AUI_DEMUX_3` is a recommended building block.

AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_DEMUX_4** function block is an event demultiplexer that routes incoming event triggers to one of four distinct output events based on an index value. Unlike the conventional E_DEMUX_4 block, which uses a separate plain event input and a data input for the selector index, this variant receives both the event trigger and the selector value through a single AUI adapter socket. This design reduces wiring complexity and improves interface encapsulation in distributed IEC 61499 applications. The block is implemented as a generic FB (GEN_E_DEMUX) and is part of the `adapter::events::unidirectional` package.

Interface Structure

The block provides a minimal interface that contains only event outputs and one adapter socket. There are no direct event inputs, data inputs, or data outputs — all incoming information is carried by the AUI adapter.

Event Inputs

None. The triggering event is delivered via the **K** adapter socket instead of a dedicated event input.

Event Outputs

Name	Description

EO1	Demultiplexed output, triggered when the index value K = 0
EO2	Demultiplexed output, triggered when the index value K = 1
EO3	Demultiplexed output, triggered when the index value K = 2
EO4	Demultiplexed output, triggered when the index value K = 3

Data Inputs

None.

Data Outputs

None.

Adapters

Name	Type	Description
K	`adapter::types::unidirectional::AUI` (Socket)	Event index adapter. Carries an event trigger together with the selector value (0–3) that determines which output is activated.

Functionality

The **AUI_DEMUX_4** block implements a 1-to-4 event demultiplexing function. The AUI adapter **K** encapsulates both an event signal and an index value. When an event arrives through the adapter, the block evaluates the accompanying index value and forwards the event to the corresponding output:

- Index **0** ⟶ event emitted on **EO1**
- Index **1** ⟶ event emitted on **EO2**
- Index **2** ⟶ event emitted on **EO3**
- Index **3** ⟶ event emitted on **EO4**

Only one output event is triggered per incoming event. The block acts as a pure event router — no data is transformed or stored, and no additional processing is performed.

Technical Features

- **Generic FB implementation**: Declared with `GenericClassName = 'GEN_E_DEMUX'`, making it a specialization of the generic E_DEMUX pattern within the 4diac framework.
- **Adapter-based event transport**: The combined event-and-index interface simplifies connections and improves reusability across different application contexts.
- **Unidirectional adapter**: The AUI adapter operates in one direction (socket side), suitable for producer-consumer event flows.
- **Deterministic routing**: The index-to-output mapping is fixed and immediate; no memory or state retention is involved in the routing decision.
- **Compliance**: Conforms to IEC 61499-1 Annex A identification and is distributed under the Eclipse Public License 2.0.

State Overview

The **AUI_DEMUX_4** block is essentially stateless in terms of routing logic. Its behavior can be described by a simple state model:

- **Idle State**: The block waits for an event to arrive on the **K** adapter.
- **Routing State**: Upon receiving an event, the index value is captured and the event is dispatched to the matching output (EO1–EO4). After emission, the block returns to the Idle State.

Since the block processes each event independently, there is no internal state that persists between activations.

Application Scenarios

- **Event distribution in modular control systems**: Use the block to route events from a single source to different consumers (e.g., state machines, processing stages) based on a dynamically changing selector.

- **Command dispatch**: In a machine control application, the AUI adapter can carry a command code (0–3) along with an execution trigger, directing the command to one of four sub-controllers.
- **Communication protocol handling**: When a received telegram contains a channel or service identifier, this block can forward the associated event to the appropriate protocol handler.
- **Reduced wiring in distributed setups**: By bundling event and index into a single adapter, connections between function blocks become cleaner and more maintainable, especially in large IEC 61499 system configurations.

Comparison with Similar Blocks

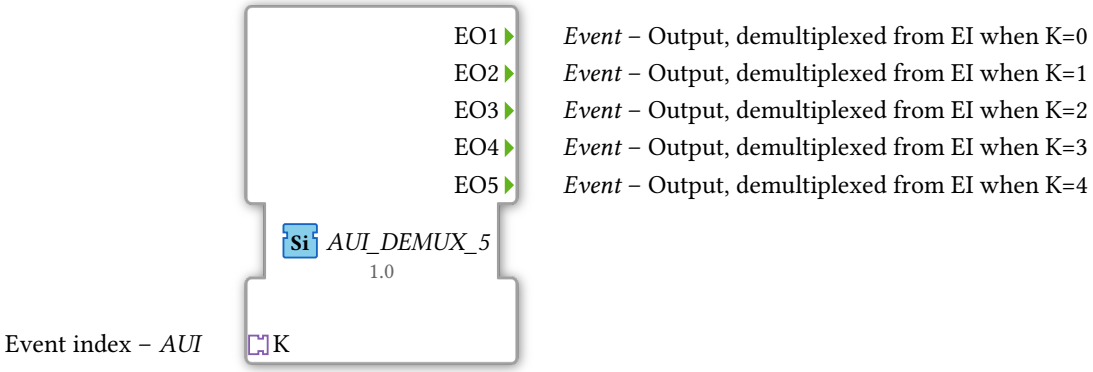
Feature	**AUI_DEMUX_4**	**E_DEMUX_4 (Standard)**
Event input	Via AUI adapter	Separate EI event input
Selector input	Part of adapter (index)	Separate K data input
Data inputs	None	1 (K)
Data outputs	None	None
Interface complexity	Reduced (fewer connections)	Higher (multiple separate pins)
Genericity	Generic FB (GEN_E_DEMUX)	Standard non-generic FB
Use case	Clean adapter-oriented designs	Classic flat wiring designs

The key advantage of the adapter variant is interface consolidation: event and index travel together, which improves modularity and reduces the risk of connection mismatches in complex applications.

Conclusion

The **AUI_DEMUX_4** function block provides a compact and elegant solution for event demultiplexing in IEC 61499-based systems. By integrating the event trigger and the selector index into a single AUI adapter, it simplifies the interface while preserving the deterministic routing behavior of a classic E_DEMUX. It is well suited for modular, adapter-centric control architectures where reducing wiring complexity and improving encapsulation are important design goals.

AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_DEMUX_5** function block is an event demultiplexer with a five-channel output structure. Instead of using a plain event input (EI) combined with a data selector (K), this block accepts its input through an **AUI adapter** (unidirectional application user interface). The block routes an incoming event to exactly one of five event outputs, based on the indexing value carried by the adapter. It is the adapter-based variant of the standard `E_DEMUX_5` block and is implemented as a generic FB (generic class name `GEN_E_DEMUX`).

Interface Structure

Event Inputs

There are no direct event inputs on this block. All event triggering and selection information is delivered through the **AUI adapter socket** named `K`. The adapter carries both the incoming event and the associated index value.

Event Outputs

Name	Type	Comment
EO1	Event	Output, demultiplexed from EI when K=0
EO2	Event	Output, demultiplexed from EI when K=1
EO3	Event	Output, demultiplexed from EI when K=2
EO4	Event	Output, demultiplexed from EI when K=3
EO5	Event	Output, demultiplexed from EI when K=4

Data Inputs

There are no direct data inputs. The selection value is embedded within the AUI adapter data channel.

Data Outputs

There are no data outputs.

Adapters

Name	Type	Direction	Comment
K	`adapter::types::unidirectional::AUI`	Socket	Event index (combines event input and selector data)

Functionality

The `AUI_DEMUX_5` block receives an event together with a selector index through the AUI adapter `K`. When an event arrives on the adapter's event channel, the block evaluates the accompanying index value and forwards the event to the corresponding output:

- If the index value is **0**, the event is emitted on `EO1`.
- If the index value is **1**, the event is emitted on `EO2`.
- If the index value is **2**, the event is emitted on `EO3`.
- If the index value is **3**, the event is emitted on `EO4`.
- If the index value is **4**, the event is emitted on `EO5`.

Exactly one output is triggered per incoming event; all other outputs remain inactive. This demultiplexing behavior enables the block to selectively activate different downstream logic branches based solely on the index provided through the adapter.

Technical Features

- **Adapter-based input:** Eliminates the need for a separate event input and data input pair, packaging both into a single `AUI` socket for cleaner interface design.
- **Unidirectional interface:** The adapter direction is unidirectional, meaning data and events flow only into the block.
- **Generic implementation:** Marked with the generic class name `GEN_E_DEMUX`, allowing the block to be instantiated with additional type information if required by the embedding application.
- **Five independent output channels:** Events are routed with a one-hot output pattern, which is straightforward to analyze and integrate into structured control logic.
- **No internal state:** The block operates purely combinatorially on the event transition; it retains no runtime data between events beyond the immediate routing decision.

State Overview

The AUI_DEMUX_5 block does not maintain any persistent internal states. Its behavior is instantaneous and deterministic:

1. **Idle** – waiting for an event on the adapter.
2. **Routing** – upon event arrival, the index is evaluated and the matching output is triggered in the same execution cycle.

Because the block is stateless, it is inherently safe to use in multithreaded or event-driven IEC 61499 runtime environments without requiring synchronization measures.

Application Scenarios

- **Event stream splitting:** Distributing events from a single producer to one of several consumers based on a routing key.
- **Mode-based control:** Selecting between different operational modes (e.g., startup, normal, maintenance, shutdown) by routing the control event to the appropriate handler.
- **Protocol adaptation:** In communication stacks, demultiplexing incoming frames or messages to dedicated processing functions depending on the message type identifier carried in the adapter.
- **Test harnesses:** Building structured test sequences where a single test trigger can be directed to various test case blocks based on an index.

Comparison with Similar Blocks

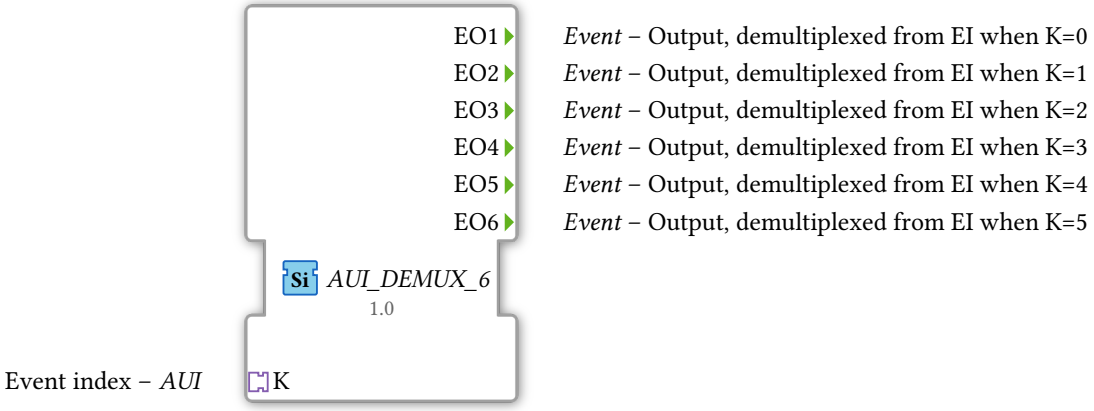
Feature	AUI_DEMUX_5	E_DEMUX_5
Input interface	Single AUI adapter socket (event + index combined)	Separate event input (EI) and data input (K)
Interface complexity	Reduced – adapter encapsulates both signals	Explicit – requires two distinct connections
Suitability for structured models	Better for adapter-based design patterns	Simpler for flat FB networks
Index range	0–4 (five outputs)	0–4 (five outputs)
Event routing logic	Identical	Identical

The core demultiplexing logic is identical to that of E_DEMUX_5. The difference lies exclusively in the input packaging – the adapter approach promotes reusability and reduces wiring clutter when an AUI-based connection pattern is already established in the system architecture.

Conclusion

AUI_DEMUX_5 provides a clean, adapter-driven event demultiplexing solution with five outputs. Its integration of the event and index signal into a single AUI socket simplifies component interfaces and aligns well with modern, adapter-based IEC 61499 design practices. Being stateless and generic, it is a versatile building block for event routing in distributed automation and control applications.

AUI_DEMUX_6



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_DEMUX_6** is an event demultiplexer function block that routes an incoming event to one of six possible output channels based on a selector index. Unlike the conventional `E_DEMUX_6` block, which requires separate Event Input (`EI`) and control data input (`K`) pins, this variant encapsulates both the event trigger and the selector index within a single **AUI adapter socket**. The adapter delivers the activation event and the selector value together, making the block highly suitable for adapter-based composition patterns and hierarchical application design.

The block is implemented as a generic function block (`GEN_E_DEMUX`), allowing flexible reuse across different data types for the selector index.

Interface Structure

The block exposes six event outputs and one adapter socket. There are no direct event inputs, data inputs, or data outputs — all inbound information flows through the AUI adapter.

Event Inputs

No explicit event inputs are declared. The triggering event is received via the AUI adapter socket.

Event Outputs

Output	Description
`E01`	Demultiplexed output, activated when selector index = 0
`E02`	Demultiplexed output, activated when selector index = 1
`E03`	Demultiplexed output, activated when selector index = 2
`E04`	Demultiplexed output, activated when selector index = 3
`E05`	Demultiplexed output, activated when selector index = 4
`E06`	Demultiplexed output, activated when selector index = 5

Data Inputs

No data inputs are present. The selector value is supplied through the AUI adapter.

Data Outputs

No data outputs are present.

Adapters

Adapter	Type	Direction	Description
`K`	`adapter::types::unidirectional::AUI`	Socket	Provides the triggering event and the selector index used to determine the active output channel. The selector value is carried as data alongside the event within the adapter connection.

Functionality

The block operates as a 1-to-6 event demultiplexer. Upon receiving an event via the `K` AUI adapter socket, the block reads the accompanying selector value (the index carried by the adapter). The selector determines which of the six event outputs is fired:

- Index `0` ⟶ `E01`
- Index `1` ⟶ `E02`
- Index `2` ⟶ `E03`
- Index `3` ⟶ `E04`
- Index `4` ⟶ `E05`
- Index `5` ⟶ `E06`

Only one output event is generated per incoming event. If the selector value is outside the valid range `0..5`, no output is triggered. Since the selector and event arrive together via the adapter, the block guarantees synchronous handling of control and event data, eliminating the risk of race conditions between separate input pins.

Technical Features

- **Generic implementation**: Declared as a generic function block (`GEN_E_DEMUX`), allowing the selector data type to be instantiated appropriately for the application context.
- **Adapter-based interface**: Replaces the conventional separate event input and data input with a single AUI socket, simplifying connection diagrams and improving encapsulation.
- **Six-way demultiplexing**: Supports up to six distinct output channels.
- **Unidirectional AUI**: The adapter socket `K` is of unidirectional type, meaning the block acts purely as a sink for the event and selector data.
- **Deterministic behavior**: Exactly one output event is generated per incoming event, based on the current selector value.
- **Zero internal state retention**: The block does not store the selector value between events; each incoming event is evaluated independently.

State Overview

The block follows the typical demultiplexer state machine found in IEC 61499 event demultiplexers:

- **Start state**: The block is idle, waiting for an event to arrive on the AUI adapter.
- **Dispatch state**: Upon receiving the event, the selector value is evaluated and the corresponding output event is triggered.
- **Return state**: The block returns to the idle state, ready to process the next event.

There is no persistent internal state; the behaviour is fully event-driven and combinatorial with respect to the selector value.

Application Scenarios

The `AUI_DEMUX_6` block is well suited for the following use cases:

- **Adapter-based control routing**: In applications where subsystems communicate through AUI adapters, this block can route events from one source to six different target components based on a control index carried in the same adapter connection.
- **Mode selection**: Selecting one of six operational modes by routing a “start” or “execute” event to the appropriate mode handler.
- **Data stream distribution**: Distributing data-triggered events (e.g. processing requests) to different processing units based on a priority or category index.
- **Hierarchical function block design**: Within composite FBs, the AUI socket keeps the internal interface clean and reduces the number of wires needed on the FB boundary.
- **Retrofitting existing designs**: The block provides a drop-in replacement for `E_DEMUX_6` in applications that already use AUI adapter architectures.

Comparison with Similar Blocks

Feature	`AUI_DEMUX_6`	`E_DEMUX_6`	`E_DEMUX` (generic)
Event input	Via AUI adapter	Separate `EI` pin	Separate `EI` pin
Selector input	Via AUI adapter (data channel)	Separate `K` data pin	Separate `K` data pin
Number of outputs	6 (`EO1`–`EO6`)	6 (`EO1`–`EO6`)	Configurable (3, 4, 6, etc.)
Adapter interface	Yes	No	No
Generic type	Yes (`GEN_E_DEMUX`)	No	Yes
Encapsulation	High — single connection	Low — two separate pins	Low — two separate pins

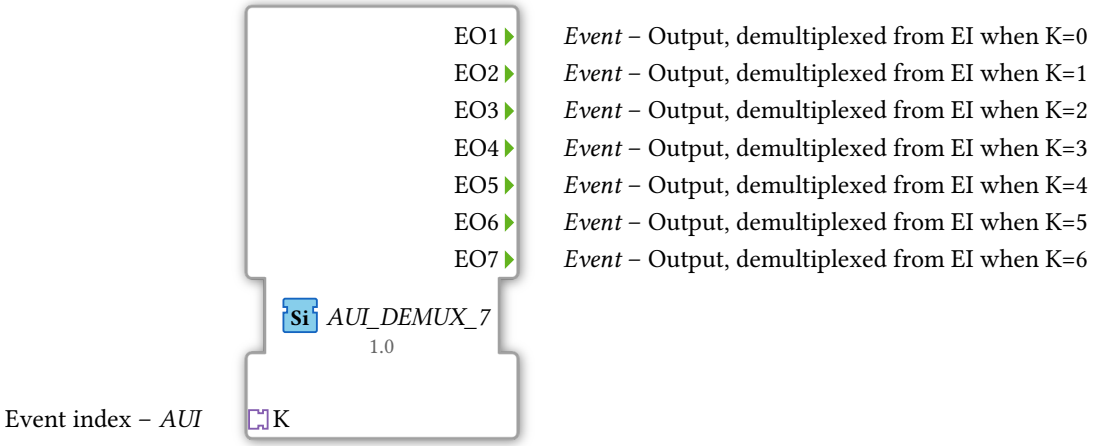
Suitability for adapter-based architectures	Excellent	Limited	Limited

Compared to the standard `E_DEMUX_6`, the AUI variant offers superior encapsulation by bundling event and selector data into a single connection. The trade-off is that it requires the surrounding system to support AUI adapter connections.

Conclusion

The **AUI_DEMUX_6** function block provides a clean, adapter-based approach to event demultiplexing in IEC 61499 applications. By integrating the event trigger and selector index into a single AUI socket, it simplifies interface design, enhances modularity, and supports generic instantiation. Its six output channels cover a wide range of routing and mode-selection use cases, making it a valuable building block for modern, adapter-oriented 4diac applications.

AUI_DEMUX_7



• – ◦ • – ◦ • – ◦ •

Introduction

AUI_DEMUX_7 is an event demultiplexer function block that routes a single incoming event to exactly one of seven mutually exclusive event outputs. It is an adapter-based variant of the standard `E_DEMUX_7` block: instead of exposing a separate event input (EI) and a data input (K), both the triggering event and the selector value are transported together through a unidirectional AUI adapter socket. This design reduces wiring complexity in modular IEC 61499 applications and enables clean encapsulation of event-triggered selection logic.

The block implements the generic class `GEN_E_DEMUX`, meaning that its behaviour follows the established demultiplexer pattern while its interface is expressed through the AUI adapter mechanism.

Interface Structure

The interface of AUI_DEMUX_7 consists exclusively of event outputs and one adapter socket. There are no plain data inputs or data outputs; all control information enters the block through the adapter.

Event Inputs

This block has no dedicated event inputs. The incoming event is carried by the AUI adapter `K`. The event arrives through the adapter's internal event channel and is demultiplexed internally.

Event Outputs

Name	Description
EO1	Output event, triggered when the selector K equals 0.
EO2	Output event, triggered when the selector K equals 1.
EO3	Output event, triggered when the selector K equals 2.

EO4	Output event, triggered when the selector K equals 3.
EO5	Output event, triggered when the selector K equals 4.
EO6	Output event, triggered when the selector K equals 5.
EO7	Output event, triggered when the selector K equals 6.

Data Inputs

None. All input data is encapsulated within the adapter.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Required)	K	`adapter::types::unidirectional::AUI`	Carries the input event and the demultiplexer selection value in one channel. The AUI adapter provides an event interface combined with associated data.

The adapter type `adapter::types::unidirectional::AUI` is part of the standard adapter library for unidirectional communication in 4diac. It bundles an event input together with the selector data, so the event and its associated select value arrive coherently.

Functionality

AUI_DEMUX_7 implements a 1-to-7 event demultiplexer. The functional behaviour is as follows:

1. An external module triggers an event on the AUI adapter connected to the socket `K`.
2. Simultaneously with that event, the adapter carries a selector value K (an integer in the range 0 to 6).
3. The block evaluates the selector value:
   - If K = 0 ⟶ the event is emitted on **EO1**.
   - If K = 1 ⟶ the event is emitted on **EO2**.
   - If K = 2 ⟶ the event is emitted on **EO3**.
   - If K = 3 ⟶ the event is emitted on **EO4**.
   - If K = 4 ⟶ the event is emitted on **EO5**.
   - If K = 5 ⟶ the event is emitted on **EO6**.
   - If K = 6 ⟶ the event is emitted on **EO7**.
4. Exactly one output is activated per input event. All other outputs remain inactive.

If the selector value is outside the valid range 0–6, no output is triggered (the event is dropped), or the behaviour follows the generic `GEN_E_DEMUX` specification.

The block is generic in the IEC 61499 sense: the attribute `eclipse4diac::core::GenericClassName` is set to `'GEN_E_DEMUX'`, which allows the 4diac runtime to treat this block as a specialisation of the generic demultiplexer implementation.

Technical Features

- **Adapter-based event transport:** The input event and selector travel together via a single AUI socket, avoiding the need for a separate event input and data input wiring.
- **Generic FB mechanism:** Declared as `GEN_E_DEMUX` through the `eclipse4diac::core::GenericClassName` attribute, enabling reuse of the generic demultiplexer behaviour.
- **Unidirectional adapter:** The AUI type is unidirectional, meaning the adapter only carries data from the plug to the socket (or vice versa, depending on orientation), simplifying the connection semantics.
- **Standard compliance:** The block is aligned with IEC 61499-1 (Annex A) identification and uses the standard 4diac adapter namespace `adapter::events::unidirectional`.
- **No external dependencies:** The block has no plain data outputs and no additional event outputs besides the seven demultiplexed channels.
- **License:** Distributed under the Eclipse Public License 2.0 (EPL-2.0).

State Overview

The internal execution control chart (ECC) of the generic `GEM_E_DEMUX` implementation follows a typical pattern for demultiplexers. Although this adapter variant does not expose its ECC in the Type definition (it relies on the generic class), the expected state behaviour is:

State	Description
IDLE	Waiting for an incoming event on the adapter's event channel. No output is active.
DECODE	The selector value K is read from the adapter and evaluated. This state is transient.
OUTPUT_1 … OUTPUT_7	One of these states is entered depending on the decoded K value. In each state, the corresponding event output (EO1 … EO7) is issued.

The transition from IDLE to DECODE is triggered by the adapter event. From DECODE, a conditional transition selects the matching output state based on K. After emitting the output event, the ECC returns to IDLE.

If K is outside the supported range, the ECC falls back to IDLE without emitting any event (or emits a default behaviour according to the generic implementation).

Application Scenarios

- **Event routing in modular automation systems:** When a single controller produces an event that must be forwarded to exactly one of seven downstream function blocks, AUI_DEMUX_7 provides a clean routing mechanism.
- **Mode-dependent processing chains:** The selector K can represent an operating mode (e.g., 0 = startup, 1 = normal, 2 = maintenance, …, 6 = error handling). The incoming event is then irrevocably channelled to the chain responsible for that mode.
- **Adapter-based subsystem interfaces:** In component-based designs where subsystems communicate via adapters, the AUI socket allows the demultiplexer to be plugged directly into a pre-existing adapter-based bus without additional event/data wiring.
- **Test and simulation environments:** Using the generic class mechanism, the block can be replaced or extended without changing its interface, which is useful for hardware-in-the-loop test benches.

Comparison with Similar Blocks

Feature	AUI_DEMUX_7	E_DEMUX_7 (standard)	E_DEMUX (generic, fewer outputs)
Input event	Via AUI adapter	Dedicated event input EI	Dedicated event input EI
Selector	Via AUI adapter (data channel)	Plain data input K	Plain data input K
Number of outputs	7	7	Variable (typically 2–N)
Interface style	Adapter-based, unidirectional	Event + data pins	Event + data pins
Wiring effort	Lower (single adapter connection)	Requires two connections (event + data)	Requires two connections
Reusability	High (generic mechanism)	Medium (fixed interface)	Medium (parameterised)

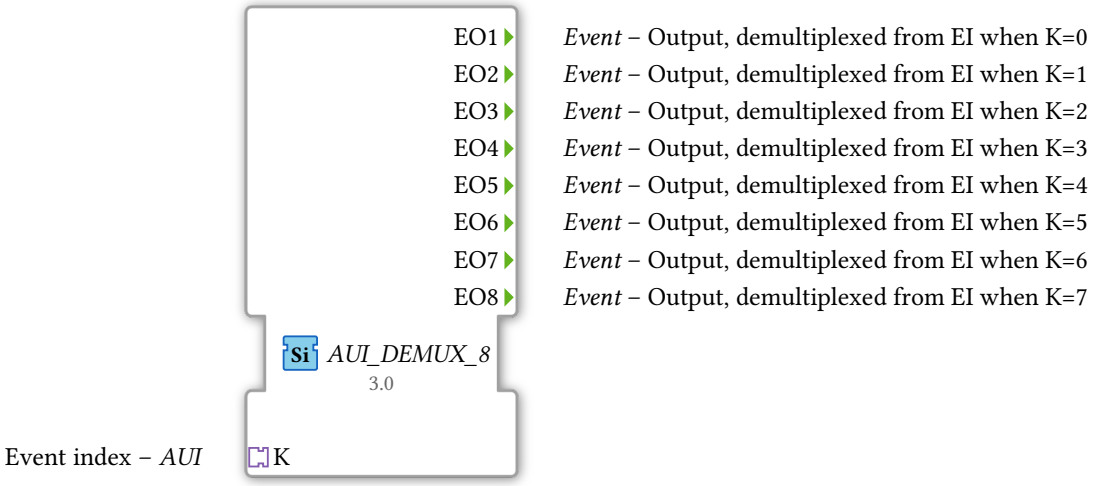
Compared to the classic `E_DEMUX_7`, the AUI variant reduces the external wiring footprint: only a single adapter connection is required at the input side. The disadvantage is that the selector value is no longer directly visible as a block pin, which can make debugging slightly less straightforward unless the adapter data is monitored.

Conclusion

AUI_DEMUX_7 is a compact and reusable event demultiplexer that combines an event input and a selector data input into a single AUI adapter connection. It is a generic, adapter-based specialisation of the well-

known E_DEMUX_7 and is suitable for IEC 61499 applications that favour modular, adapter-oriented design patterns. With seven mutually exclusive event outputs, it supports straightforward 1-to-N event routing, while its generic-class declaration ensures flexibility for different runtime implementations. The block is fully EPL-2.0 licensed and integrates seamlessly into the 4diac ecosystem.

AUI_DEMUX_8



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_DEMUX_8** is an event demultiplexer function block that routes an incoming event to one of eight possible output channels. Unlike a conventional E_DEMUX block, which requires a separate event input and a data input for the selection index, this block receives both the event trigger and the index value through a single unidirectional AUI adapter socket. This design simplifies wiring in distributed control applications by consolidating the event and its routing information into one adapter connection.

The block is implemented as a generic function block (GEN_E_DEMUX), allowing it to be parameterized and reused across different application contexts while maintaining a consistent demultiplexing behavior.

Interface Structure

Event Inputs

This function block has **no explicit event inputs**. The incoming event is received via the AUI adapter socket, which combines the event trigger and the routing index into a single connection.

Event Outputs

Name	Type	Comment
EO1	Event	Output, demultiplexed when K = 0
EO2	Event	Output, demultiplexed when K = 1
EO3	Event	Output, demultiplexed when K = 2
EO4	Event	Output, demultiplexed when K = 3
EO5	Event	Output, demultiplexed when K = 4
EO6	Event	Output, demultiplexed when K = 5
EO7	Event	Output, demultiplexed when K = 6
EO8	Event	Output, demultiplexed when K = 7

Each event output corresponds to one index value carried by the adapter, allowing the upstream event to be selectively forwarded to exactly one of the eight connected downstream blocks.

Data Inputs

This function block has **no data inputs**. The selection index is transmitted as part of the AUI adapter data, rather than through a standalone data port.

Data Outputs

This function block has **no data outputs**.

Adapters

Name	Type	Direction	Comment
K	adapter::types::unidirectional::AUI	Socket	Event index

The **K** socket is a unidirectional AUI adapter that provides the event trigger together with the index value (0–7). Because it is a socket, the block expects a matching plug from the partner component, typically an event source or a bus controller that supplies both the event and its destination index.

Functionality

The AUI_DEMUX_8 operates as a classic 1-to-N event demultiplexer, where N = 8. When the upstream component sends an event through the AUI adapter, the block extracts the accompanying index value and forwards the event to the corresponding output channel:

- If the index equals **0**, the event is emitted on **EO1**.
- If the index equals **1**, the event is emitted on **EO2**.
- ……
- If the index equals **7**, the event is emitted on **EO8**.

The demultiplexing decision is made solely from the index value carried by the adapter. The block requires no additional configuration parameters and does not retain state between events, making it a purely combinational routing element.

Because the block is a generic FB (GEN_E_DEMUX), the underlying implementation can be instantiated with different output counts or adapter variants depending on the application requirements, while the AUI_DEMUX_8 representation is fixed to eight outputs.

Technical Features

- **Eight output channels**: Supports routing to one of eight independent event destinations.
- **Adapter-based input**: The event and index are combined in a single unidirectional AUI adapter, reducing the number of connections and simplifying the interface.
- **Generic implementation**: The block is built on the GEN_E_DEMUX generic FB pattern, making it suitable for code generation and toolchain integration.
- **No internal state**: The routing decision is stateless, so the block does not require initialization or reset sequences.
- **Direct index mapping**: No ambiguity in routing — each index value has a dedicated output, and out-of-range indices are not generated by the upstream contract.

State Overview

The AUI_DEMUX_8 is a **stateless** function block. It does not contain an ECC (Execution Control Chart) state machine or maintain persistent internal variables. Every incoming event is processed independently:

1. An event arrives on the adapter K.
2. The index value is evaluated.
3. The corresponding output is triggered.
4. The block returns to its idle condition, ready for the next event.

This stateless behavior guarantees predictable and immediate response times, which is critical in time-sensitive control loops.

Application Scenarios

The AUI_DEMUX_8 is well suited for scenarios where a single event source must selectively activate one of several processing paths based on a routing index. Typical use cases include:

- **Multi-machine coordination**: Distributing a start command to one of eight machines or substations based on a machine identifier.

- **Mode-based routing**: Selecting one of eight operating modes, where the mode index determines which downstream FB receives the event.
- **Reconfigurable communication hubs**: Acting as a flexible event router in IEC 61499 distributed applications, especially when the index source is also an adapter-based block.
- **Reducing wiring complexity**: Replacing separate event and data connections with a single adapter link, which is particularly beneficial in modular or hot-pluggable systems.

Comparison with Similar Blocks

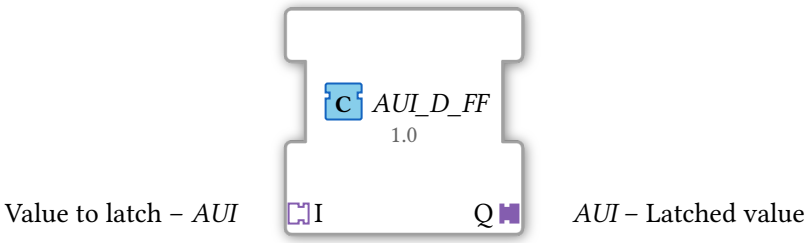
Feature	E_DEMUX (standard)	AUI_DEMUX_8
Event input	Separate EI port	Provided via AUI adapter
Selection index input	Separate K data port	Carried by AUI adapter
Number of outputs	Typically 2–8 (configurable)	Fixed to 8
Adapter-based connectivity	No	Yes (unidirectional AUI)
Genericity	Often generic	Generic (GEN_E_DEMUX)
Wiring complexity	Higher (two connections)	Reduced (one connection)

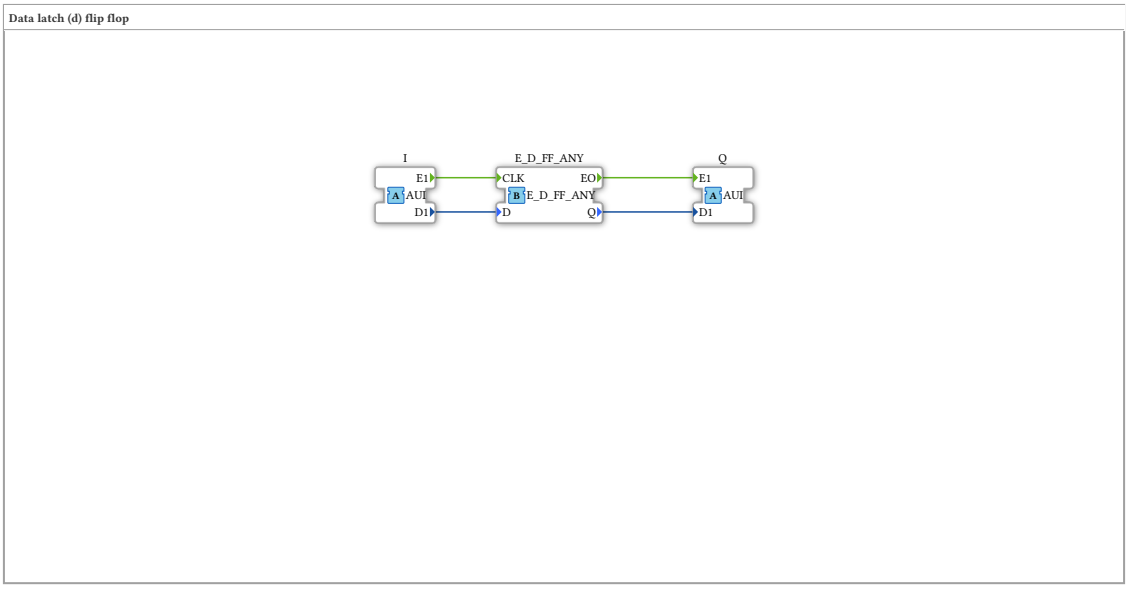
Compared to the traditional E_DEMUX, the AUI_DEMUX_8 provides a more compact interface by merging the event trigger and the routing index into one adapter. However, it requires both the source and sink to support the AUI adapter protocol, so it is most effective in systems already built around adapter-based communication.

Conclusion

The AUI_DEMUX_8 is a compact and efficient event demultiplexer that leverages a unidirectional AUI adapter to combine event triggering with index-based routing. Its eight outputs provide sufficient granularity for many industrial control and automation scenarios, while the stateless, generic implementation ensures reliable and predictable behavior. By reducing interface complexity and improving connection clarity, it is a valuable building block in modern IEC 61499-based distributed control systems.

AUI_D_FF





• – ○ • – ○ • – ○ •

Introduction

The function block **AUI_D_FF** implements a clock-controlled D flip-flop (data latch). It takes the value of a data input when an event occurs and outputs the stored value. The input and output interfaces are implemented as unidirectional **adapters**, enabling a flexible and standardized connection in the 4diac IDE.

Interface Structure

Event Inputs

None – event control is handled exclusively via the **I** adapter.

Event Outputs

None – event output is handled exclusively via the **Q** adapter.

Data Inputs

None – data transfer is handled exclusively via the **I** adapter.

Data Outputs

None – data output is exclusively via the **Q** adapter.

Adapters

Name	Type	Direction	Description
I	`adapter::types::unidirectional::AUI`	Socket (Input)	Input value (D) and clock event (CLK)
Q	`adapter::types::unidirectional::AUI`	Plug (Output)	Stored value (Q) and output event (EO)

The **AUI** adapter type provides a unidirectional channel that transmits exactly one event and one data value. Thus, the internal connections between the flip-flop and the outside world are implemented via adapters.

Functionality

The function block internally uses the function block `E_D_FF_ANY` from the IEC 61499 standard library.

- On each **clock event** on `I.E1`, the current data value is taken from `I.D1`.
- The taken value is made available at output `Q.D1`, and an **output event** is triggered simultaneously on `Q.E1`.
- The stored value is retained until a new clock event takes over a new data value.

Technical Features

- The interface consists exclusively of **adapters**, not separate event and data ports. This allows for loose coupling and reusability of the adapter type in different contexts.
- The adapter `AUI` is **unidirectional** – data and events flow in only one direction.
- The function block is type-safe: The data type is defined by the adapter (`ANY` by default).

State Overview

The function block has two logical states, corresponding to the internal flip-flop:

State	Description
Q = 0	Output value is LOW (logical 0).
Q = 1	Output value is HIGH (logical 1).

The state changes only on a rising edge of the clock event (provided the event is triggered in the adapter).

Application Scenarios

- **Synchronization**: Transfer of a data value at a precisely defined time (e.g., upon a sensor event).
- **Central Storage**: Buffering a value for further processing in subsequent components.
- **State Machines**: Storing states in discrete controllers when a simple D flip-flop function is sufficient.

Comparison with Similar Components

- **E_D_FF** (Standard IEC 61499): Offers the same flip-flop functionality, but with separate event and data ports. The `AUI_D_FF` encapsulates these ports in an adapter, enabling clean, adapter-based communication.
- **E_RS** or **E_SR** (Reset-Set Flip-Flops): Unlike the D flip-flop, these components require separate set/reset events and no clock signal. The `AUI_D_FF` is optimized for clock-driven applications.

Conclusion

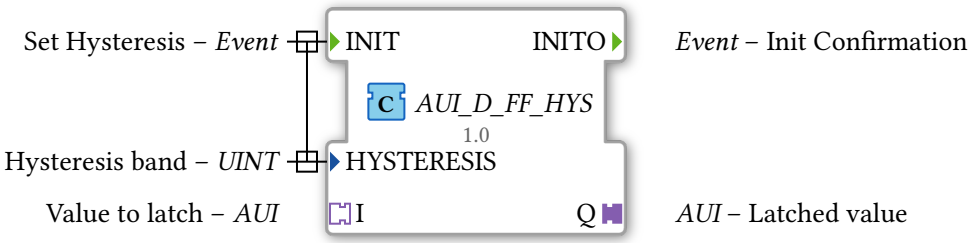
The `AUI_D_FF` is a standards-compliant D flip-flop in the 4diac IDE that offers a modular and reusable interface through the use of **adapters**. It is particularly suitable for applications where a uniform, unidirectional connection structure is desired without having to forgo the proven flip-flop logic.

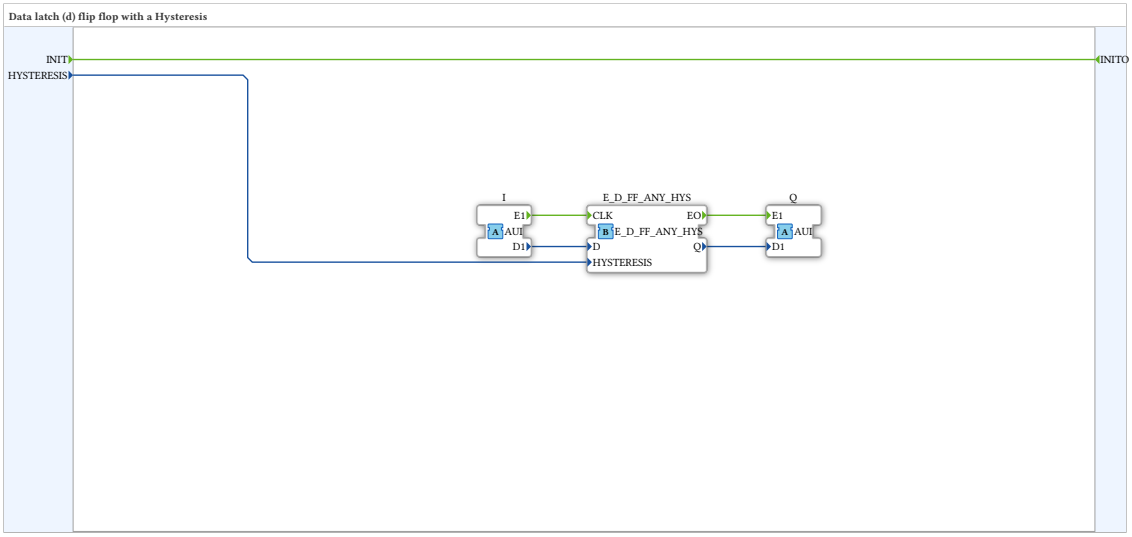
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AUI_D_FF_HYS





• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_D_FF_HYS` implements a data latch (D flip-flop) with a hysteresis function. It serves to reliably acquire an analog or digital value, suppressing small fluctuations (noise) through an adjustable hysteresis band. The block is designed for use in automation technology and complies with the IEC 61499-1 Annex A standard.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Sets the hysteresis width

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization Confirmation

Data Inputs

Data	Type	Comment
HYSTERESIS	UINT	Hysteresis Band Width

Data Outputs

No dedicated data outputs; the latched value is provided via the `Q` adapter.

Adapters

Name	Type	Direction	Comment
I	adapter::types::unidirectional::AUI	Socket	Latch Value
Q	adapter::types::unidirectional::AUI	Plug	Latched Value

The adapters `I` and `Q` of type `AUI` (unidirectional) transport both data (e.g., `D1`) and associated events (e.g., `E1`).

Functionality

The component internally contains the instance `E_D_FF_ANY_HYS` (type `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS`), which performs the actual latch with hysteresis. The signal flows are as follows:

- The event `I.E1` (from adapter `I`) is passed directly to the internal component as a clock signal (`CLK`).
- The data value `I.D1` is passed to the internal flip-flop as an input `D`.
- The hysteresis band is set via the parameter `HYSTERESIS`.
- The output of the internal component (`Q`) is routed to adapter `Q.D1`, and the output event (`EO`) is routed to `Q.E1`.
- The output of the internal component (`Q`) is routed to adapter `Q.D1`, and the output event (`EO`) is routed to adapter `Q.E1`.
- The data value `I.D1` is passed to the internal flip-flop as an input `D`.
- The output of the internal component (`Q`) is routed to adapter `EO`. - The event `INIT` is immediately passed through to `INITO` (no delay).

The hysteresis ensures that the latched value only jumps to the new input value if it differs from the currently stored value by more than the defined threshold (`HYSTERESIS`). This suppresses edge noise and unstable signal transitions.

Technical Features

- The function block is a pure **composite function block** that delegates all logic to an internal instance.
- The hysteresis is passed as an unsigned integer (`UINT`).
- The adapters are unidirectional – data and events flow in only one direction (socket ⟶ plug).
- The function block does not contain its own state machine; the state is maintained in the internal flip-flop.

State Overview

Since `AUI_D_FF_HYS` does not have its own ECC (Execution Control Chart), its behavior is determined solely by the internal component `E_D_FF_ANY_HYS`. A typical D flip-flop with hysteresis has two stable output states, which are switched by the applied clock signal and the hysteresis condition. A detailed state description can be found in the documentation for the internal component.

Application Scenarios

- **Sensor Data Debouncing**: Values from analog or digital sensors are stabilized with an adjustable hysteresis.
- **Threshold Switch**: The component can be used as a Schmitt trigger if an input value only changes its output above a defined upper threshold and below a defined lower threshold (determined by the hysteresis).
- **Signal Conditioning in Process Automation**: Noisy signals are smoothed before being passed to subsequent logic components.

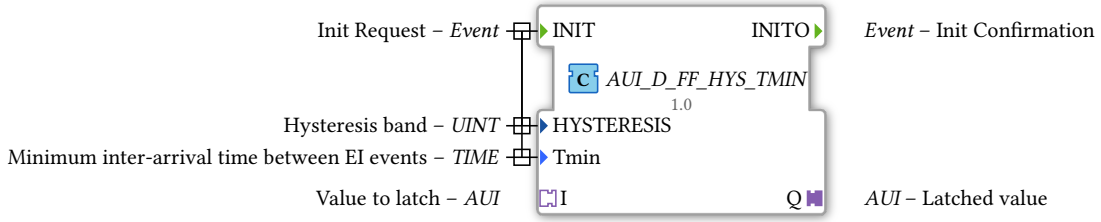
Comparison with Similar Components

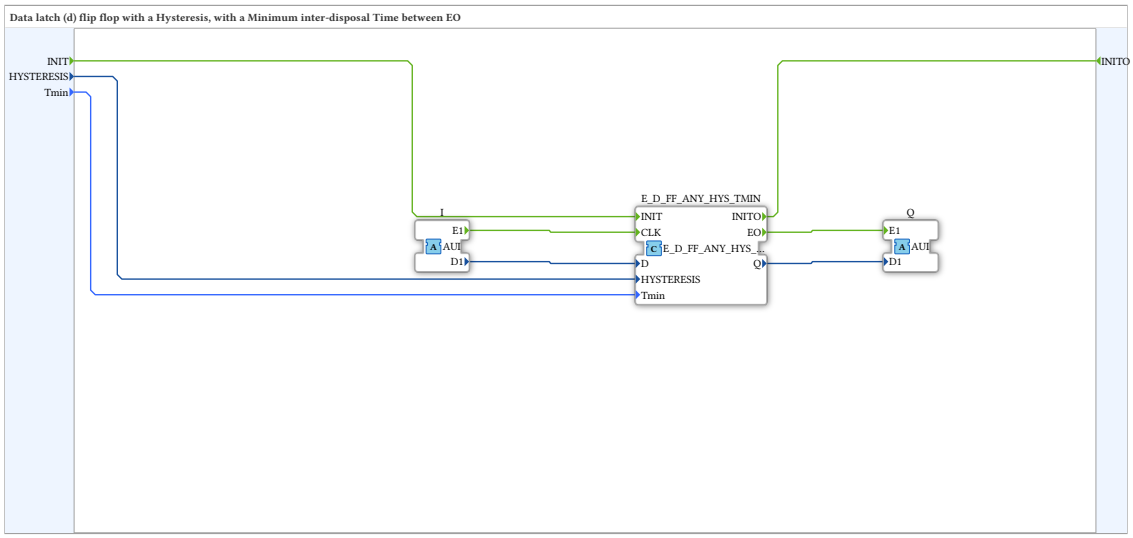
- **E_D_FF**: A pure D flip-flop without hysteresis. It adopts the input value on every clock edge, regardless of minimal fluctuations. `AUI_D_FF_HYS` offers improved noise immunity.
- **E_RS_BISTABLE**: A set-reset flip-flop without hysteresis and limited to binary states. `AUI_D_FF_HYS` operates with analog/integer values and allows for a dynamic threshold.
- **E_SPLIT_HYS**: A component that splits a signal with a fixed hysteresis. In contrast, the hysteresis in `AUI_D_FF_HYS` can be changed at runtime via the input `HYSTERESIS`.

Conclusion

The `AUI_D_FF_HYS` is a useful component for stable latching of values in environments with noise or signal instability. It combines the functionality of a D flip-flop with an adjustable hysteresis threshold and can be easily integrated into modular IEC 61499 controllers using adapters.

AUI_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AUI_D_FF_HYS_TMIN** is a data storage device (D flip-flop) with adjustable hysteresis and a minimum intermediate event time (`Tmin`). It is connected to other function blocks via adapter interfaces (socket `I` and plug `Q`). The internal core corresponds to function block `E_D_FF_ANY_HYS_TMIN` from the library `logiBUS::signalprocessing::hysteresis`. The initialization defines the parameters. Afterward, for each incoming event at input `I.E1`, the current value from `I.D1` is taken, filtered, and output via `Q.D1`.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Acknowledgement

Data Inputs

Name	Type	Comment
HYSTERESIS	UINT	Hysteresis Band
Tmin	TIME	Minimum Intermediate Arrival Time of EI Events

Data Outputs

No standalone data outputs – the output value is transmitted via the `Q` adapter.

Adapter

Type	Name	Comment
`adapter::types::unidirectional::AUI` (Socket)	I	Value to be stored
`adapter::types::unidirectional::AUI` (Plug)	Q	Stored Value

The adapter `AUI` is a unidirectional type that bundles an event and a data value (`E1` and `D1`).

Functionality

1. **Initialization**

Upon the event `INIT`, the parameters `HYSTERESIS` (as UINT) and `Tmin` (as TIME) are passed to the internal module. After successful initialization, `INITO` is triggered.

1. **Operation**

The internal function block `E_D_FF_ANY_HYS_TMIN` operates as a clocked D flip-flop with hysteresis and a time limit:

- An event at `I.E1` (Clock) takes the current data value from `I.D1`.
- The taken value is only passed on as a valid output `Q.D1` if the interval to the last valid event is greater than or equal to `Tmin`.
- Additionally, the value is smoothed with a hysteresis (band `HYSTERESIS`): Small changes below the hysteresis threshold are suppressed.
- The output is signaled via the event `Q.E1`.

1. **Result**

The output adapter `Q` returns the last assumed, filtered value.

Technical Features

- **Hysteresis** in conjunction with a **minimum time between two events** prevents both noise and excessively rapid state changes.
- The function block is implemented as an **adapter wrapper** – it uses standardized unidirectional AUI adapters for input and output.
- The implementation delegates all logic to the function block `E_D_FF_ANY_HYS_TMIN`, which is universally designed for any data type. Here, it is used in the context of the AUI adapter.

State Overview

The function block does not have an explicit internal state machine. The behavior is represented by the internal function block:

- **Idle:** Waits for the next event on `I.E1`.
- **Sampling:** The data value is captured upon arrival of `I.E1`.
- **Filtering:** Checks against hysteresis and minimum time.
- **Output:** If successful, `Q.E1` is triggered and the new value is assigned to `Q.D1`.

Application Scenarios

- **Sensor Value Acquisition** in agricultural technology (see Copyright HR Agrartechnik GmbH): Processing of analog or digital signals with debouncing and minimum sampling rate.
- **Control Components** in automation systems where noise suppression and a minimum pulse duration are required.
- **Data Logic** in distributed systems that must respond to asynchronous events with stability requirements.

Comparison with Similar Components

Component	Property
`E_D_FF`	Simple D flip-flop without hysteresis or time limit

`E_D_FF_HYS`	D flip-flop with hysteresis, but without `Tmin`
`E_D_FF_HYS_TMIN` (this FB)	Combines hysteresis *and* minimal time interval → more robust against noise and burst events

Conclusion

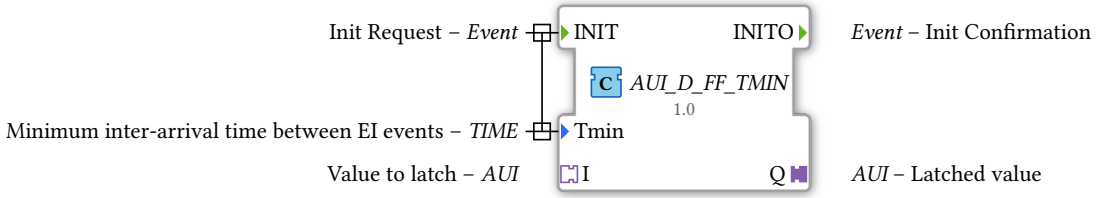
The `AUI_D_FF_HYS_TMIN` offers a compact and reliable solution for signal-processing data acquisition, suppressing unwanted interference pulses through hysteresis and time limiting. The adapter interface facilitates integration into existing 4diac projects using standardized, unidirectional AUI connections.

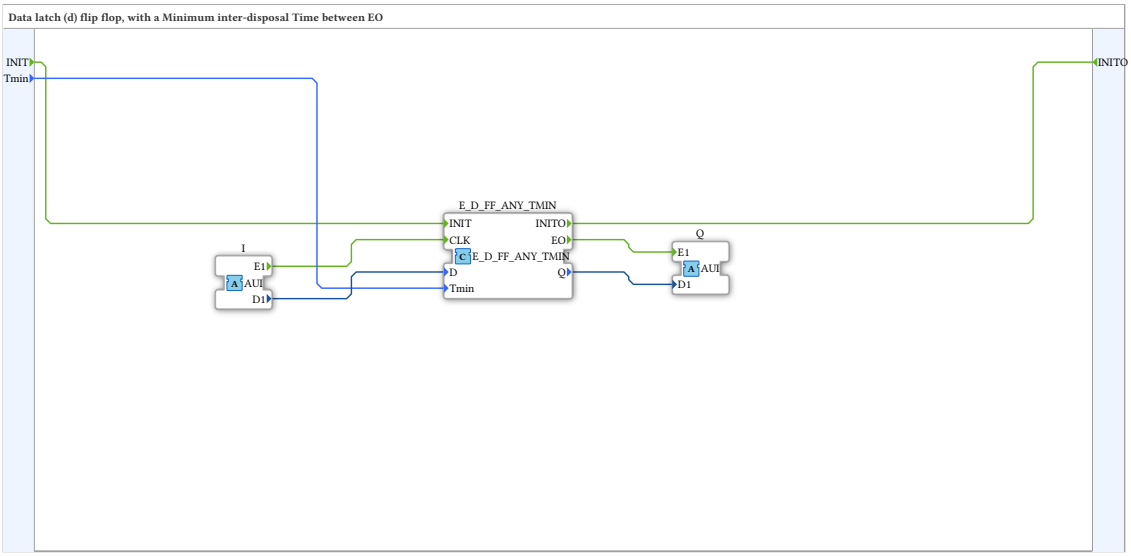
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUI_D_FF_TMIN





• – ○ • – ○ • – ○ • –

Introduction

The function block **AUI_D_FF_TMIN** implements a data-locking D flip-flop (data latch) with a minimum inter-disposal time between two consecutive output events. It serves to receive values via the unidirectional **AUI** adapter and pass them on with a defined minimum delay to prevent an excessively rapid sequence of events at the output.

Interface Structure

Event Inputs

Event	Comment
INIT	Initialization request; configures the block with the parameter `Tmin`.

Event Outputs

Event	Comment
INITO	Confirmation of successful initialization.

Data Inputs

Variable	Type	Comment
Tmin	TIME	Minimum time that must elapse between two output events (`EO`).

Data Outputs

No direct data outputs – the output values are provided via the `Q` adapter.

Adapter

Direction	Name	Type	Comment
Plug (Output)	**Q**	`adapter::types::unidirectional::AUI`	Provides the locked data value (event and data line).
Socket (Input)	**I**	`adapter::types::unidirectional::AUI`	Receives the data value to be locked (event and data line).

Functionality

1. **Initialization**

The block is initialized via the event input `INIT`, and the minimum time is set via `Tmin`.

1. **Data Transfer**

Every event arriving at the adapter input `I` via the event line `I.E1` transfers the corresponding data value `I.D1` to the flip-flop. The transferred value is immediately stored internally.

1. **Output with Minimum Time Delay**

The stored value is only output as an event on `Q.E1` and as a data value on `Q.D1` if at least `Tmin` of time has elapsed since the last output event. If a new transfer event occurs before this time has elapsed, the new value is stored, and the output is further delayed until the minimum time has been met.

1. **Output with Minimum Time Delay**

The output of the stored value occurs only as an event on `Q.E1` and as a data value on `Q.D1` if at least `Tmin` of time has passed since the last output event. If a new transfer event occurs before this time has elapsed, the new value is stored, and the output is further delayed until the minimum time has been met.

1. **State Protection**

The block behaves like an edge-triggered D flip-flop, where each valid clock cycle (event on `I.E1`) takes over the current data value, but the actual transmission is time-limited.

Technical Features

- **Adapter-Based Input/Output** – The block uses only the standardized unidirectional **AUI** adapter. This makes it easy to integrate into existing adapter-oriented architectures.
- **Reuse of Basic Components** – Internally, the IEC 61499 component `E_D_FF_ANY_TMIN` is used, which implements the time-limited throttling. This block encapsulates it and simplifies connection via adapters.
- **No Dedicated State Machines** – The behavior is entirely determined by the internal component; The outer function block is purely configurable and connective.

State Overview

The function block does not have an externally visible state machine (ECC). Internally, the used `E_D_FF_ANY_TMIN` can have the following implicit states:

State	Description
Idle	Waiting for an event at input `I`.
Timed	Data has been received; output is held back until `Tmin` expires.
Output	Minimum time has been met – output event is sent.

These states are transparent to the user.

Application Scenarios

- **Sensor Value Smoothing** – Prevents a sensor that delivers measurements at short intervals from flooding the subsequent processing block with events.
- **Actuator Control with Minimum Switching Time** – Ensures that an actuator (e.g., a valve) can only be switched after a defined minimum pause to reduce wear.
- **Data Flow Limiting in Communication Lines** – Throttles the output rate of a data stream to prevent overloading the receiver.

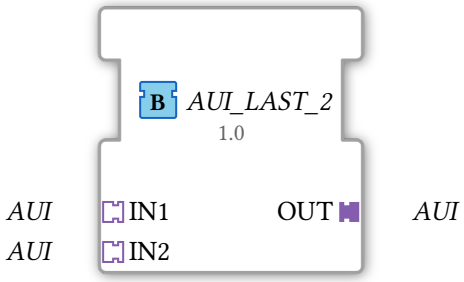
Comparison with Similar Components

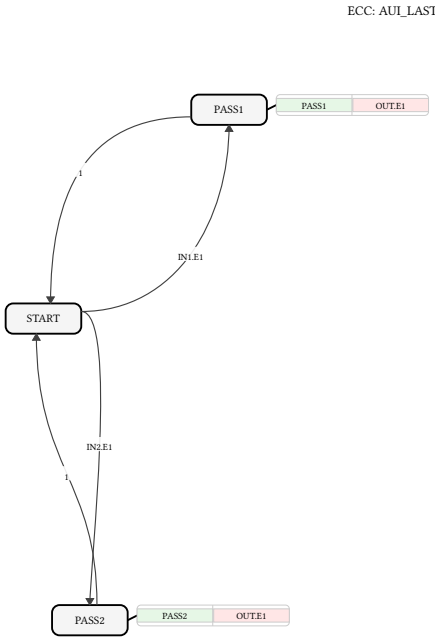
Component	Feature
E_D_FF (Standard D Flip-Flop)	No time limit; each event is output immediately.
E_D_FF_ANY_TMIN	Same behavior, but without adapter encapsulation; Direct event-/ data-based connection.
AUI_D_FF_TMIN (this block)	Combines D flip-flop functionality with time-based throttling and provides an adapter-based interface.

Conclusion

The **AUI_D_FF_TMIN** is a useful function block when data needs to be passed at time-controlled intervals. It simplifies the implementation of throttling logic in adapter-based IEC 61499 systems and is particularly suitable for applications in agricultural engineering and other time-critical control environments.

AUI_LAST_2





Introduction

The AUI_LAST_2 function block merges two same-type UINT adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: UINT adapter (unidirectional) - first signal source
- **IN2**: UINT adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: UINT adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

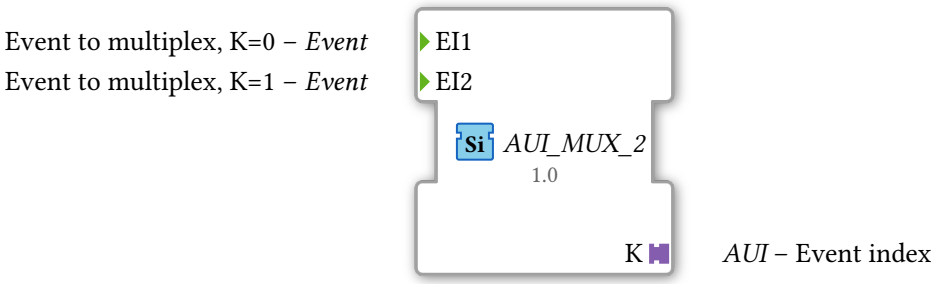
Application Scenarios

- Merging two equally-valid UINT signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same UINT value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AUI_LAST_2 is the inverse of AUI_SPLIT_2: AUI_SPLIT_2 distributes 1 input to 2 outputs, AUI_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use AX_AE_MERGE instead, which merges only the event without altering the data value.

AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

AUI_MUX_2 is an event multiplexer function block that forwards one of two incoming events (EI1 or EI2) based on an index value supplied through an AUI (Adapter User Interface) adapter. It is a specialized variant of the standard E_MUX_2 block, replacing the conventional combination of a data input K and a plain event output EO with a unidirectional AUI adapter connection. This design integrates the event-index selection and the multiplexed event delivery into a single adapter-based interface, making it suitable for architectures where adapter-based communication is preferred over direct event/data ports.

Interface Structure

Event Inputs

Name	Type	Description
EI1	Event	Event to be multiplexed when the index K selects channel 0.
EI2	Event	Event to be multiplexed when the index K selects channel 1.

Event Outputs

The block has no explicit event output ports. The multiplexed event is delivered through the AUI adapter connection (see Adapters below).

Data Inputs

No data inputs are provided. All selection information is carried via the adapter.

Data Outputs

No data outputs are provided. The adapter serves as the data and event output mechanism.

Adapters

Name	Type	Direction	Description
K	adapter::types::unidirectional::AUI	Plug (required)	Provides the event selection index and carries the multiplexed event output to a connected socket.

Functionality

The AUI_MUX_2 operates as a two-channel event selector. When an event arrives on EI1 or EI2, the block evaluates the current index value transmitted through the K adapter. If the index selects channel 0, the incoming event on EI1 is forwarded via the AUI adapter to the connected partner FB. If the index

selects channel 1, the event on EI2 is forwarded instead. Events arriving on a non-selected channel are ignored.

The adapter-based design encapsulates both the index value and the event output in one interface. This means the consuming logic does not need to observe a separate K data input or a dedicated event output; instead, it receives the selection context and the event through the same AUI connection. This reduces port clutter and enforces a cleaner separation between the multiplexing logic and the consuming application.

The block is declared as a generic FB (GEN_E_MUX) within the 4diac framework, meaning its behavior can be instantiated with a configurable number of inputs, although the fixed AUI_MUX_2 variant presented here exposes exactly two event input channels.

Technical Features

- **Adapter-based I/O:** All selection and output communication occur through a unidirectional AUI adapter, eliminating dedicated data input and event output ports.
- **Generic background:** Built on the generic GEN_E_MUX core, allowing conceptual extension to more than two channels in adapted versions.
- **Combinational event logic:** No internal storage or state machine is required; the routing decision is made purely based on the current adapter index value at the time an input event occurs.
- **Deterministic selection:** Each input event channel is strictly associated with one index value (EI1 ↔ index 0, EI2 ↔ index 1).
- **Standard compliance:** Developed according to IEC 61499-1 Annex A, ensuring interoperability within 4diac-based automation systems.

State Overview

This function block does not maintain persistent internal states. Its behavior is purely event-driven and combinational:

- When EI1 fires and the adapter-index is 0 ⟶ the selected event is emitted through the AUI adapter.
- When EI1 fires and the adapter-index is not 0 ⟶ the event is discarded.
- When EI2 fires and the adapter-index is 1 ⟶ the selected event is emitted through the AUI adapter.
- When EI2 fires and the adapter-index is not 1 ⟶ the event is discarded.

The block's reaction is instantaneous and does not depend on prior events, making it a deterministic routing element within a larger IEC 61499 application.

Application Scenarios

AUI_MUX_2 is particularly useful in system designs that already rely on AUI adapter connections for unidirectional communication between FBs. Typical use cases include:

- **Event routing in adapter-based architectures:** When a framework or application layer mandates that all event exchange between components occurs via adapters, AUI_MUX_2 allows event selection without breaking that convention.
- **Reducing port complexity:** In large FB networks, replacing multiple separate K data inputs and EO outputs with a single adapter simplifies wiring and improves readability.
- **Modular consumer integration:** The AUI socket on the consumer side can be designed to automatically interpret the multiplexed events with the accompanying index, enabling polymorphic event handling.
- **Migration of E_MUX_2 logic:** Applications previously built with E_MUX_2 but moving toward adapter-based communication can be migrated by swapping the plain output/data ports with the AUI adapter.

Comparison with Similar Blocks

Feature	E_MUX_2	AUI_MUX_2
Event inputs	EI1, EI2	EI1, EI2
Selection mechanism	Data input K (INT)	AUI adapter K
Event output	Plain event output EO	Via AUI adapter
Port count	3 (2 events + 1 data) + 1 event output	2 events + 1 adapter

Adapter support	None	Unidirectional AUI
Generic base	GEN_E_MUX	GEN_E_MUX

The key distinction is that AUI_MUX_2 consolidates the index selection and event output into one adapter interface. This makes it more suitable for adapter-oriented designs but requires that the connected consumer FB exposes a matching AUI socket. In contrast, E_MUX_2 uses the more traditional separate data input and event output, which is simpler to use in classic IEC 61499 applications.

Conclusion

AUI_MUX_2 provides a clean, adapter-based alternative to the classical E_MUX_2 event multiplexer. By routing the selection index and the multiplexed event through a single unidirectional AUI adapter, it aligns with modern IEC 61499 communication patterns that favor adapter encapsulation. It is ideal for systems where adapter-based event propagation is a design requirement, and it retains the deterministic, combinational behavior expected from a multiplexing element. While slightly more constrained than its plain counterpart due to the dependency on a matching AUI socket, it offers significant advantages in modularity and interface clarity for appropriate application contexts.

AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

The AUI_MUX_3 is an event multiplexer function block designed for selecting and forwarding one of three incoming event signals based on an index provided through an AUI (Adapter Unit Interface) adapter. Unlike conventional multiplexers that use separate event output and index data inputs, this block integrates the selector and output channel into a single unidirectional AUI adapter. This design reduces the number of interface elements and simplifies the connection to downstream components that expect a standardized adapter-based event interface.

As a generic function block (based on the GEN_E_MUX template), it can be instantiated with different numbers of inputs or adapted to various use cases, but the provided configuration supports exactly three event inputs (EI1, EI2, EI3). The block is particularly useful in event-driven systems where multiple sources need to be routed to a common destination, with the selection logic controlled by an external entity via the AUI adapter.

Interface Structure

Event Inputs

Name	Type	Comment
EI1	Event	Event to multiplex, K=0
EI2	Event	Event to multiplex, K=1
EI3	Event	Event to multiplex, K=2

These three event inputs represent the sources to be multiplexed. Only one of them is passed through to the output, depending on the value of the index provided through the AUI adapter.

Event Outputs

There are no explicit event outputs in this block. The selected event is transmitted via the AUI adapter connection, which acts as the output channel. This is a deliberate design choice to integrate the output event and the selection logic in a single adapter interface.

Data Inputs

There are no data inputs in this block. The multiplexer index is not provided as a separate data input but is instead carried within the AUI adapter connection.

Data Outputs

There are no data outputs in this block.

Adapters

Name	Type	Direction	Comment
K	adapter::types::unidirectional::AUI	Output (Plug)	Event index and output channel

The adapter `K` is a **plug** (output adapter) of type `adapter::types::unidirectional::AUI` (Adapter Unit Interface). It provides two functions:

- **Output for the selected event**: The multiplexed event is sent through this adapter to a connected socket.
- **Selector input**: The index value (0, 1, or 2) is supplied from the connected socket, indicating which of the three event inputs should be forwarded.

The AUI adapter encapsulates the index and the event output in a single connection, simplifying the wiring between the multiplexer and the receiving function block.

Functionality

The AUI_MUX_3 operates as a three-input event router. Whenever an event occurs on any of its inputs (EI1, EI2, EI3), the block checks the current value of the index available on the AUI adapter. Only if the index matches the input number (i.e., EI1 with index 0, EI2 with index 1, EI3 with index 2) will the event be forwarded through the AUI adapter to the connected downstream block. Events arriving on non-selected inputs are ignored.

The index value is continuously monitored and can be updated at any time via the AUI adapter. This allows dynamic switching between sources without needing to preconfigure the selection externally. The block does not have internal states; it simply reacts to incoming events based on the current index.

In a typical scenario, the AUI adapter is connected to a socket of another function block that both provides the selection index (e.g., from a controller) and receives the multiplexed event. This creates a bidirectional (or unidirectional, depending on the adapter definition) communication channel where the upstream block supplies the selection, and the downstream block receives the event.

Technical Features

- **Generic Implementation**: The block is based on the `GEN_E_MUX` generic template, allowing instantiation with different numbers of event inputs if needed, though the default configuration is fixed to three inputs.
- **AUI Adapter Integration**: Minimizes the number of separate interface points by combining the output event and the selector index into a single adapter connection. This reduces wiring complexity and improves modularity.
- **Unidirectional Adapter**: The `AUI` adapter is defined as unidirectional, meaning data and events flow only in one direction (from the multiplexer to the connected block). The index value is supplied externally to the adapter's socket and is read by the multiplexer.
- **No Internal State**: The block is stateless and reacts purely to input events and the current index value. This makes it suitable for time-critical applications where deterministic behavior is required.
- **Event Prioritization**: In case multiple events occur simultaneously, the block does not prioritize them; the first event processed according to the runtime order will be evaluated. This is consistent with the behavior of standard event multiplexers.

State Overview

This function block does not maintain any internal state. It has no state diagram because it does not store information between executions. The behavior is purely combinatorial:

- **Idle**: No event is present on any input.
- **Event Reception**: When an event arrives on one of the inputs, the block evaluates the current index from the AUI adapter.
- **Forwarding**: If the index matches the arriving input, the event is emitted through the adapter. Otherwise, the event is discarded.

The block is always ready to process the next event, and there is no transition delay except for the propagation of the adapter's index value (which may change independently).

Application Scenarios

- **Selective Event Routing**: In a system where multiple sensors or components generate events but only one should be processed at a time, AUI_MUX_3 can direct events from the currently active source to a central processing unit.
- **Configurable Input Selection**: When a control unit needs to switch between different data sources (e.g., different operating modes), the AUI adapter can be used to set the selection index dynamically, allowing the multiplexer to forward the appropriate event stream.
- **Adapter-Based Architectures**: In 4diac applications that heavily use adapters for decoupling and reusability, this block integrates seamlessly into a chain of adapters, avoiding the need for separate event and data lines.
- **Test and Simulation**: The block can be used in test harnesses to inject events from different input channels into a single output line, controlled by a test script that updates the adapter index.

Comparison with Similar Blocks

The AUI_MUX_3 is a variant of the standard `E_MUX` event multiplexer. The key differences are:

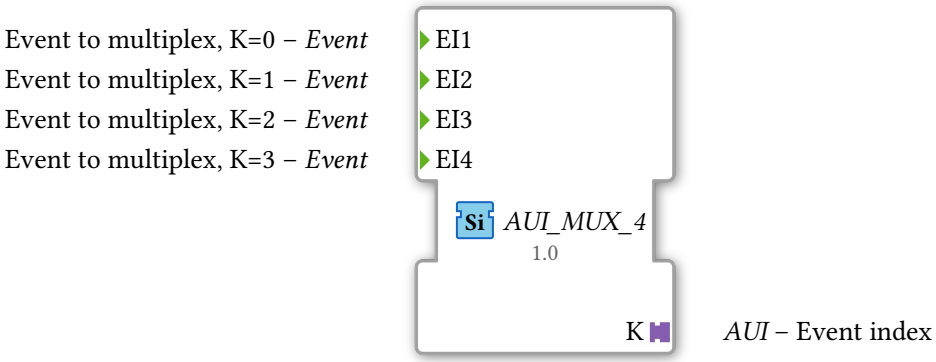
- **Interface Structure**: `E_MUX` typically has separate event output (EO) and a data input (K) of type INT to specify the active input. AUI_MUX_3 instead uses an AUI adapter that combines the index and the output event into one connection.
- **Flexibility**: The `E_MUX` may allow configuration of the number of inputs via a generic parameter, while AUI_MUX_3 is fixed to three inputs in this version, but can be extended following the same pattern.
- **Integration**: AUI_MUX_3 is more suited for adapter-based designs where the output is expected to be an adapter socket. It reduces the number of connections and can simplify the visual layout of the application.
- **Standard-Tool Compatibility**: While `E_MUX` is a well-known block in many IEC 61499 implementations, AUI_MUX_3 is a specialized adaptation that may be used in environments where adapter-based event communication is preferred.

Other multiplexers like `E_MUX2` or `E_MUX4` share the same concept but differ in input count. AUI_MUX_3 is unique in that it encapsulates the selector channel into an adapter, which is not typical in standard libraries.

Conclusion

AUI_MUX_3 provides a compact and elegant solution for event multiplexing in IEC 61499 applications that leverage adapter-based communication. By integrating the selection index and the output event into a single AUI adapter, it reduces interface clutter and promotes a cleaner system structure. Its stateless, event-driven nature ensures predictable behavior and easy integration into existing event processing chains. While it may not replace the standard `E_MUX` in all scenarios, it offers a valuable alternative for modern, adapter-centric designs. The block is a clear demonstration of how generic event processing can be adapted to meet the needs of evolving industrial automation architectures.

AUI_MUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_MUX_4** is an event multiplexer function block that selects one of four event inputs and forwards the received event through an **AUI adapter** interface. This is an adapter-based variant of the standard `E_MUX_4` block, replacing the plain event output plus data selector with a unidirectional AUI adapter plug. The block is configured as a generic function block (`GEN_E_MUX`), allowing the number of inputs to be adapted as needed.

Interface Structure

The AUI_MUX_4 features four event inputs and one AUI adapter plug. There are no conventional event outputs, data inputs, or data outputs; all selection and output functionality is carried through the adapter.

Event Inputs

Name	Type	Comment
EI1	Event	Event to multiplex, K=0
EI2	Event	Event to multiplex, K=1
EI3	Event	Event to multiplex, K=2
EI4	Event	Event to multiplex, K=3

Event Outputs

None. The multiplexed event is delivered via the AUI adapter plug.

Data Inputs

None.

Data Outputs

None.

Adapters

Name	Type	Direction	Comment
K	`adapter::types::unidirectional::AUI`	Plug (output)	Event index and event forwarding interface

The adapter `K` is of type `AUI` (unidirectional), carrying both the selection index and the event signal to the connected partner FB.

Functionality

The AUI_MUX_4 operates as an event selector. When an event occurs on one of the four inputs (EI1–EI4), the block recognizes the corresponding index (0–3) and forwards an event through the AUI adapter plug

K. The adapter carries the index value along with the event trigger, enabling the connected downstream block to identify which input caused the event.

The behavior is analogous to a classic multiplexer: exactly one input is selected and its event is propagated. The selection is implicit—based on which input fired—rather than set via a separate data input. This makes the block suitable for scenarios where event sources are known and need to be routed to a single consumer.

Technical Features

- **Generic FB instantiation**: Marked with `eclipse4diac::core::GenericClassName = 'GEN_E_MUX'`, enabling flexible adaptation of the number of event inputs.
- **Adapter-based output**: Uses the `AUI` adapter type (`adapter::types::unidirectional::AUI`) instead of a standard event output plus data output, offering a clean interface for unidirectional event communication.
- **Four input channels**: Supports multiplexing of up to four independent event sources.
- **No data handling**: Pure event-based operation; no data inputs or outputs are involved.
- **Package classification**: Belongs to the `adapter::events::unidirectional` package, reflecting its unidirectional event-oriented architecture.

State Overview

The block does not maintain internal state machines or persistent states. It operates purely combinatorially in response to incoming events. Each event on an input is immediately forwarded through the adapter without internal buffering or state retention. There are no ECC (Execution Control Chart) states to consider for this FB.

Application Scenarios

- **Event consolidation**: Merging multiple event sources into a single event stream where the adapter carries the origin index.
- **Inter-module event routing**: Connecting multiple sensors or controllers to a common processing unit via an AUI adapter, preserving the identity of the event source.
- **Generic event demultiplexing**: When used with a matching AUI recipient block, the index can be used to reconstruct the original event distribution on the consumer side.
- **Adapter-based system design**: Ideal for modular IEC 61499 systems that rely on adapter connections for clean, decoupled communication between function blocks.

Comparison with Similar Blocks

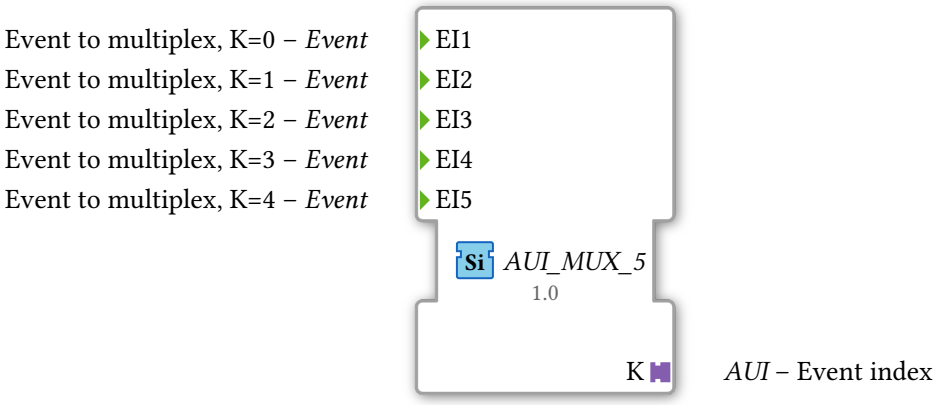
Feature	AUI_MUX_4	E_MUX_4 (standard)
Output type	AUI adapter plug	Event output (EO) + data output (K)
Selection mechanism	Implicit via fired input, index via adapter	Explicit via data input K
Event count	4 inputs	4 inputs
Data handling	None	Carries selector value K
Use case	Adapter-based architectures	Traditional ECC/event chains

The key difference is the output interface: AUI_MUX_4 delivers the event and selector via an adapter, promoting loose coupling and reusability, whereas E_MUX_4 uses traditional event and data outputs.

Conclusion

The AUI_MUX_4 is a specialized event multiplexer designed for adapter-centric IEC 61499 applications. By utilizing a unidirectional AUI adapter for both event forwarding and index signaling, it simplifies wiring in complex systems and enables clean separation of concerns. Its generic nature allows scaling to different input counts, making it a flexible building block for event routing in distributed automation systems.

AUI_MUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

AUI_MUX_5 is an event multiplexer with five event inputs. It is the adapter-based variant of a classic `E_MUX_5` block. Instead of providing a plain event output `EO` and a data output `K`, this block uses a unidirectional AUI adapter output called `K` that carries both the generated event and the selected event index.

The block belongs to the `adapter::events::unidirectional` package and is implemented using the generic event multiplexer class `GEN_E_MUX`.

Interface Structure

Event Inputs

Name	Type	Comment
`EI1`	Event	Event to multiplex, K=0
`EI2`	Event	Event to multiplex, K=1
`EI3`	Event	Event to multiplex, K=2
`EI4`	Event	Event to multiplex, K=3
`EI5`	Event	Event to multiplex, K=4

Event Outputs

None. The event output is embedded in the AUI adapter `K`.

Data Inputs

None.

Data Outputs

None. The data output is embedded in the AUI adapter `K`.

Adapters

Name	Direction	Type	Comment
`K`	Plug	`adapter::types::unidirectional::AUI`	Event index

The adapter plug `K` is the output side of the block. It provides a unidirectional connection to a consuming FB that has a matching AUI socket.

Functionality

AUI_MUX_5 works as an event selector. When one of the five event inputs `EI1` to `EI5` is triggered, the block:

1. Recognizes which event input received the event.
2. Maps that input to the corresponding zero-based index:
   - `EI1` ⟶ `K = 0`
   - `EI2` ⟶ `K = 1`
   - `EI3` ⟶ `K = 2`
   - `EI4` ⟶ `K = 3`
   - `EI5` ⟶ `K = 4`
3. Forwards the event through the AUI adapter `K`.
4. Provides the index value through the same adapter connection.

The consumer connected to the AUI socket receives both the event and the associated channel index in one structured interface.

Technical Features

- Five event inputs: `EI1` to `EI5`.
- One unidirectional AUI adapter plug named `K`.
- Generic implementation based on `GEN_E_MUX`.
- Uses the adapter type `adapter::types::unidirectional::AUI`.
- Replaces the separate `EO` and `K` outputs of a conventional event multiplexer.
- The adapter output is unidirectional, meaning events and index data flow from this block to the connected consumer.
- No separate event output, data input, or data output is exposed on the outer interface.

State Overview

AUI_MUX_5 does not expose an application-visible state machine. It behaves as an event-driven selector. After an event is processed, the selected index is available on the AUI adapter and the event is propagated to the connected socket.

The block is stateless between event occurrences from the user's perspective. No internal event history is required for normal operation.

Application Scenarios

AUI_MUX_5 is suitable for applications where several event sources need to be combined into one adapter-based output channel.

Typical use cases include:

- Merging event signals from multiple sensors into one common event handling path.
- Selecting one of several alarm or status sources and forwarding it to a central monitoring block.
- Connecting an event multiplexer to an AUI-compatible consumer without manually wiring an event output and an index data output.
- Replacing a conventional `E_MUX_5` in systems that already use unidirectional AUI adapters for communication between FBs.

Comparison with Similar Blocks

Compared with `E_MUX_5`, AUI_MUX_5 provides the same event multiplexing behavior but uses an AUI adapter output instead of a separate event output `EO` and data output `K`. This groups the related output information together and reduces connection clutter.

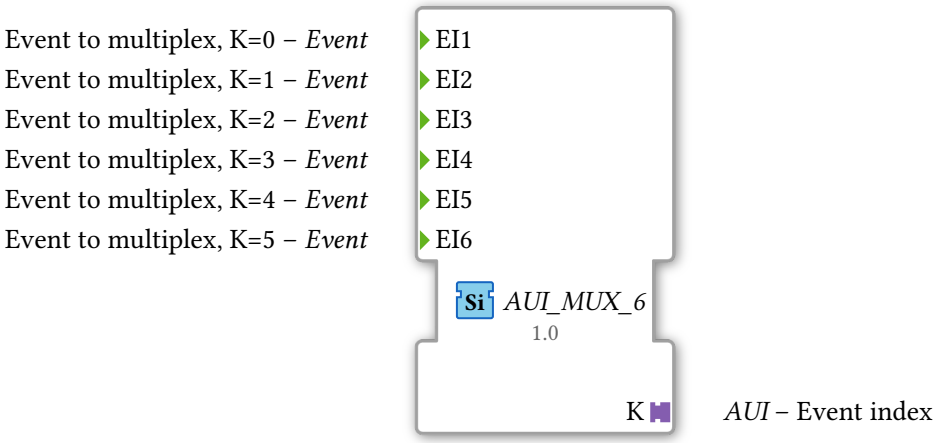
Compared with other `AUI_MUX_n` variants, AUI_MUX_5 is specialized for exactly five event inputs. The index mapping starts at `0` and ends at `4`.

Compared with the generic `GEN_E_MUX` block, AUI_MUX_5 is a concrete typed specialization with a fixed number of five event inputs and a predefined AUI adapter interface.

Conclusion

AUI_MUX_5 is a compact, adapter-based event multiplexer for five event inputs. It combines event selection and index information in one unidirectional AUI adapter connection. This makes it especially useful in component-oriented IEC 61499 applications where standardized AUI adapters are used for event and data exchange between function blocks.

AUI_MUX_6



• – ◦ • – ◦ • – ◦ •

Introduction

`AUI_MUX_6` is an event multiplexer function block with six event inputs. Instead of exposing a separate event output and a plain `K` data connection, it uses a unidirectional AUI adapter as its output interface. This combines the delivered event and the associated event index in a single adapter connection.

The block is a concrete configuration of the generic event multiplexer `GEN_E_MUX` and is intended for applications where multiple event sources must be merged into one unidirectional adapter-based event channel while preserving the identity of the triggering event source.

Interface Structure

The block provides six event inputs and one AUI adapter plug. It has no conventional event outputs, data inputs, or data outputs.

Event Inputs

Name	Type	Comment
`EI1`	Event	Event to multiplex, K=0
`EI2`	Event	Event to multiplex, K=1
`EI3`	Event	Event to multiplex, K=2
`EI4`	Event	Event to multiplex, K=3
`EI5`	Event	Event to multiplex, K=4
`EI6`	Event	Event to multiplex, K=5

**Event Outputs

None. The output event is carried through the AUI adapter `K`.

Data Inputs

None.

Data Outputs

None.

Adapters

Name	Type	Direction	Comment
`K`	`adapter::types::unidirectional::AUI`	Plug	Event index

The adapter K bundles the outgoing event with the corresponding event index. The index is zero-based and identifies which event input caused the output event.

Functionality

AUI_MUX_6 monitors its six event inputs and forwards a triggering event through the AUI adapter output. The event index carried by the adapter is derived from the input that was activated:

- An event on EI1 produces an adapter event with index 0.
- An event on EI2 produces an adapter event with index 1.
- An event on EI3 produces an adapter event with index 2.
- An event on EI4 produces an adapter event with index 3.
- An event on EI5 produces an adapter event with index 4.
- An event on EI6 produces an adapter event with index 5.

The adapter output is therefore both the multiplexed event signal and the source-identification signal. The receiving FB, connected through an AUI socket, can use the event to trigger its logic and the K value to determine which of the six original event inputs originated the event.

Technical Features

- Six fixed event input channels with zero-based indices.
- Single unidirectional AUI adapter output K.
- Compact interface: replaces separate event output and data output connections with one adapter.
- No external data inputs are required for multiplexing.
- The K value is delivered together with the output event, guaranteeing that the event and its index are observed consistently.
- Uses the generic implementation class GEN_E_MUX, allowing tooling and runtime environments to treat it as a generic event multiplexer.

State Overview

AUI_MUX_6 does not define a persistent internal state machine. It behaves statelessly from the application viewpoint: each event input is handled independently and immediately produces the corresponding adapter output event. After the event is emitted, the block returns to its idle condition and waits for the next input event.

There is no enabling, disabling, or selection state. All six event inputs are always ready to be forwarded.

Application Scenarios

- **Event source aggregation**: Combine multiple event sources into one adapter-based connection while preserving the source index.
- **Modular subapplication communication**: Transport events from several components of a subapplication to a central coordinator through a single unidirectional adapter.
- **Reduction of interface wiring**: Replace separate event and data lines with one AUI adapter connection, simplifying the interface of reusable FBs.
- **Source identification**: Use the K value on the receiving side to distinguish which of the six event inputs triggered the received event.
- **Generic event multiplexing**: Use the GEN_E_MUX generic behavior to build similar multiplexer variants with different numbers of event inputs.

Comparison with Similar Blocks

Block / Variant	Output Approach	Typical Use
Classic E_MUX style	Separate event output and K connection	Standard event multiplexing with direct event and data wiring
AUI_MUX_6	Event and index bundled in a unidirectional AUI adapter	Adapter-based multiplexing with source identification
Event demultiplexer	One input event routed to one of several outputs	Opposite operation: distributing one event stream to multiple destinations

The main advantage of AUI_MUX_6 over classic multiplexing blocks is the use of a single adapter connection instead of physically separate event and data interfaces. This keeps the event and its index together and simplifies the interface of the containing function block or subapplication.

Conclusion

AUI_MUX_6 provides a clean, adapter-based solution for multiplexing six event inputs into one unidirectional event channel. By combining the output event with a zero-based event index in the K AUI adapter, it preserves source information while reducing interface complexity. It is especially useful in modular 4diac applications where adapter-based communication is preferred over individual event and data connections.

AUI_MUX_7



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_MUX_7** is a generic event multiplexer function block that selects one out of seven incoming event inputs and routes it to an event output provided through an **AUI (Adapter Usage Interface)** adapter. This block is a variant of the standard **E_MUX_7**, where the selection index and the output event are bundled into a single unidirectional AUI adapter, making it suitable for modular, adapter-based communication patterns in distributed control applications.

Interface Structure

Event Inputs

Name	Type	Description
EI1	Event	Event to multiplex; selected when K = 0
EI2	Event	Event to multiplex; selected when K = 1
EI3	Event	Event to multiplex; selected when K = 2
EI4	Event	Event to multiplex; selected when K = 3
EI5	Event	Event to multiplex; selected when K = 4
EI6	Event	Event to multiplex; selected when K = 5
EI7	Event	Event to multiplex; selected when K = 6

Event Outputs

None. The multiplexed event is delivered through the AUI adapter's event channel.

Data Inputs

None. The selection index is provided via the AUI adapter.

Data Outputs

None.

Adapters

Name	Type	Direction	Description
K adapter	`::types::unidirectional::AUI`	Plug (output side)	Carries the event index (selection value) and provides the multiplexed event output to the connected socket.

Functionality

The **AUI_MUX_7** operates as a single-selection multiplexer controlled by the index value `K` supplied through the AUI adapter. When an event occurs on one of the seven inputs (EI1–EI7), the block checks the current value of `K`:

- If `K` matches the index associated with the triggered input, the event is forwarded to the output event channel of the AUI adapter.
- If `K` does not match, the event is discarded.

Only one event input is forwarded at a time, determined by the most recent value of `K`. This behavior allows a single consumer connected via the AUI socket to receive events from exactly one source (selected dynamically at runtime) without requiring separate event connections.

Technical Features

- **Generic FB implementation** – The block is based on the generic class `GEN_E_MUX`, allowing instantiation with a configurable number of inputs (here fixed to 7).
- **Adapter-based output** – The output event and selection index are encapsulated in a unidirectional AUI adapter, promoting loose coupling between producer and consumer.
- **Event-index correlation** – Each event input is mapped to a fixed index (EI1⟶0, EI2⟶1, ..., EI7⟶6), directly corresponding to the `K` value.
- **No data processing** – The block performs purely event-based routing; it does not manipulate or generate data values.
- **SPDX-Licensed** – Distributed under Eclipse Public License 2.0.

State Overview

The **AUI_MUX_7** is a stateless, combinational event router. It does not maintain internal state variables or sequential logic. The behavior can be summarized as:

Condition	Action
Event on EIn and K == (n-1)	Forward event to AUI output
Event on EIn and K != (n-1)	Drop event (no output)
No event on any input	No action

Since there is no state, the block is always ready to react to the next incoming event.

Application Scenarios

- **Dynamic source selection** – In systems where multiple sensors or controllers generate events but only one should be active at a time, the AUI_MUX_7 can route the active source's events to a downstream processing unit.
- **Adapter-based modular architectures** – When using AUI adapters to standardize connections between components, this block integrates seamlessly with unidirectional event-driven communication patterns.
- **Mode-switchable control panels** – A control application can switch between different operational modes (e.g., manual, automatic, emergency) by changing `K`, thereby selecting which event source (button, PLC signal, safety circuit) drives the output.
- **HMI and event aggregation** – In visualization systems, the block can multiplex multiple user-interface events to a single rendering engine, with the active widget determined by the index.

Comparison with Similar Blocks

Feature	AUI_MUX_7	E_MUX_7 (standard)
Output event	Via AUI adapter	Plain Event Output (EO)
Selection index	Via AUI adapter (K)	Separate data input (K)
Interface style	Adapter-based (unidirectional)	Direct event/data pins
Coupling	Loose (decoupled via adapter)	Tight (direct wiring)
Reusability	Higher – can connect to any matching AUI socket	Lower – requires explicit event and data connections
Typical use	Modular, distributed IEC 61499 systems	Simple, monolithic function block networks

The AUI_MUX_7 is essentially the adapter-wrapped equivalent of the standard E_MUX_7, trading a direct interface for greater modularity and interchangeability.

Conclusion

The **AUI_MUX_7** provides a clean, adapter-based solution for multiplexing seven event sources into a single output. By integrating the selection index and the output event into one unidirectional AUI adapter, it simplifies wiring, improves component interchangeability, and fits naturally into modern IEC 61499 distributed control architectures that rely on adapter-based communication. Its stateless nature and generic implementation make it a robust and flexible building block for event-driven applications.

AUI_MUX_8



• – ◦ • – ◦ • – ◦ •

Introduction

AUI_MUX_8 is an event multiplexer function block with a unidirectional AUI adapter output. It concentrates up to eight incoming event inputs on a single AUI adapter connection. Instead of forwarding the selected event through a plain event output and a separate index value, the block emits the event and the associated event index through the adapter `K`. It is a concrete specialization of the generic event multiplexer `GEN_E_MUX` and is intended for use in event-driven 4diac applications.

Interface Structure

The FB has eight event inputs and one AUI adapter Plug. There are no conventional event outputs, data inputs, or data outputs. All outgoing event and index information is transported through the `K` adapter.

Event Inputs

Input	Description
EI1	Event to multiplex, K = 0
EI2	Event to multiplex, K = 1
EI3	Event to multiplex, K = 2
EI4	Event to multiplex, K = 3
EI5	Event to multiplex, K = 4
EI6	Event to multiplex, K = 5
EI7	Event to multiplex, K = 6
EI8	Event to multiplex, K = 7

Event Outputs

None. The multiplexed event is sent through the AUI adapter `K`.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Type	Direction	Description
K	`adapter::types::unidirectional::AUI`	Plug	Output adapter carrying the multiplexed event and the index of the active event input (0–7).

Functionality

When an event occurs at one of the event inputs, the FB produces an event on the `K` adapter. The index value attached to this adapter output corresponds to the triggered input:

- EI1 ⟶ K = 0
- EI2 ⟶ K = 1
- EI3 ⟶ K = 2
- EI4 ⟶ K = 3
- EI5 ⟶ K = 4
- EI6 ⟶ K = 5
- EI7 ⟶ K = 6
- EI8 ⟶ K = 7

The receiving FB on the other side of the AUI adapter can use the event to trigger processing and the index to identify the source. Because the index is determined by the event input and not by an external selector, the block can be used as a compact event-source encoder.

Technical Features

- Supports up to eight event inputs with a fixed 0-based index mapping.
- Provides an AUI plug `K` using adapter type `adapter::types::unidirectional::AUI`.
- No separate event outputs or data pins; the AUI adapter encapsulates the output interface.
- Declared as a generic FB specialization with `GenericClassName = 'GEN_E_MUX'`.
- Stateless operation with no internal data persistence.
- Simplifies wiring by replacing explicit EO + K connections with an adapter-based connection.

State Overview

The type description does not define an explicit ECC state machine. AUI_MUX_8 is a stateless, event-activated block. At runtime, it remains in its ready state until an event input occurs; then it immediately emits the event and index through K and returns to the ready state. No state information is stored between invocations.

Application Scenarios

- Consolidating multiple event sources such as buttons, sensors, diagnostic flags, or mode triggers into one AUI-based connection to a central event handler.
- Supplying an event index to a processing block so the receiver can distinguish which source triggered the event.
- Reducing wiring complexity in 4diac applications by using AUI adapters instead of separate event output and index data connections.
- Serving as an eight-input specialization of a generic event multiplexer in modular automation logic.

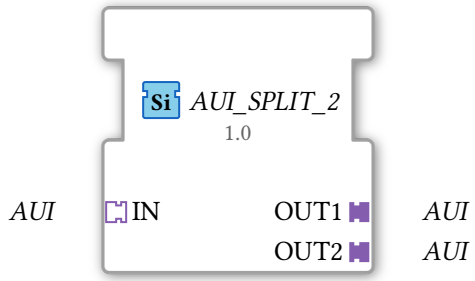
Comparison with Similar Blocks

- **Standard E_MUX:** Uses a plain event output and an external selector input. AUI_MUX_8 replaces this interface with an AUI adapter and derives the index directly from the triggering event input.
- **E_DEMUX:** Performs the inverse operation, routing one incoming event to one of several outputs according to a selector. AUI_MUX_8 merges several inputs into one adapter output.
- **Other AUI_MUX variants:** AUI_MUX_2, AUI_MUX_4, AUI_MUX_16, etc. follow the same principle with a different number of inputs and index range; AUI_MUX_8 is the eight-channel version.

Conclusion

AUI_MUX_8 is a practical event multiplexing block for applications that use AUI adapter connections. It combines event concentration and source identification in one interface, allowing up to eight event inputs to be forwarded as a single adapter-based event with a clear index. Its stateless behavior and generic design make it suitable for modular, event-driven control systems based on 4diac.

AUI_SPLIT_2



Introduction

The function block **AUI_SPLIT_2** distributes a signal received from one AUI adapter (socket) unchanged to two identical AUI adapters (plugs). It is delivered as a generic function block (GenericClassName = `'GEN_AUI_SPLIT'`) and enables the simple splitting of a unidirectional AUI data stream to two independent sinks.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available. Data is transported exclusively via the adapters.

Data Outputs

No direct data outputs available. Data is transmitted exclusively via the adapters.

Adapters

Name	Type	Direction / Role
`IN`	`adapter::types::unidirectional::AUI`	Socket (Input)
`OUT1`	`adapter::types::unidirectional::AUI`	Plug (Output 1)
`OUT2`	`adapter::types::unidirectional::AUI`	Plug (Output 2)

All three adapters are of type **AUI** and operate unidirectionally. Socket `IN` receives data, and plugs `OUT1` and `OUT2` forward the same data.

Functionality

This component forwards every incoming AUI signal at socket `IN` simultaneously to both plugs `OUT1` and `OUT2`. No processing, filtering, or buffering of the data takes place—the data is copied and sent **1:1**. The distribution is strictly deterministic and without any additional delay (apart from the propagation time of the internal connection).

Technical Features

- **Generic Type**: The function block is identified by the attribute `GenericClassName` as `'GEN_AUI_SPLIT'` and can be instantiated at runtime using type-hash mechanisms (`TypeHash` is currently empty).
- **No ECC (Execution Control Chart)**: Since there is no event-driven logic, the function block does not have a state machine. The routing is purely data-driven.
- **EPL 2.0 Licensed**: The function block is provided under the Eclipse Public License 2.0.

State Overview

The AUI_SPLIT_2 function block has **no state machine**. There are no internal states or transitions – the signal is continuously routed without clocking.

Application Scenarios

- **Signal Distribution**: An AUI data stream needs to be transmitted in parallel to two independent receivers or subsystems (e.g., two controllers, displays, or actuators).
- **Redundancy**: Identical data should be provided on two paths to support failover mechanisms.
- **Protocol Multicasting**: A unidirectional AUI stream is distributed to multiple consumers without requiring the sender to be addressed multiple times.

Comparison with Similar Components

- **AUI_SPLIT_N**: An extended variant that splits one input into N outputs (e.g., 1:3, 1:4). The AUI_SPLIT_2 is the simplest member of this family.
- **AUI_MERGE_2**: Combines two AUI inputs into one output (optionally with arbitration). The splitter operates in the opposite direction.
- **AUI_PASS**: A pure 1:1 pass-through adapter without branching.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

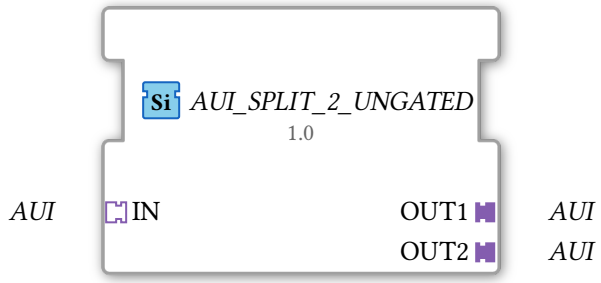
Conclusion

The **AUI_SPLIT_2** is a minimalist yet useful function block for splitting a unidirectional AUI signal into two outputs. Thanks to its generic implementation and the absence of complex logic, it is ideally suited for scenarios requiring clean, lossless duplication of data streams.

AUI_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change.

This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_2_UNGATED** distributes a signal received from one AUI adapter (socket) unchanged to two identical AUI adapters (plugs). It is delivered as a generic function block (GenericClassName = `'GEN_AUI_SPLIT'`) and enables the simple splitting of a unidirectional AUI data stream to two independent sinks.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available. Data is transported exclusively via the adapters.

Data Outputs

No direct data outputs available. Data is transmitted exclusively via the adapters.

Adapters

Name	Type	Direction / Role
`IN`	`adapter::types::unidirectional::AUI`	Socket (Input)
`OUT1`	`adapter::types::unidirectional::AUI`	Plug (Output 1)
`OUT2`	`adapter::types::unidirectional::AUI`	Plug (Output 2)

All three adapters are of type **AUI** and operate unidirectionally. Socket `IN` receives data, and plugs `OUT1` and `OUT2` forward the same data.

Functionality

This component forwards every incoming AUI signal at socket `IN` simultaneously to both plugs `OUT1` and `OUT2`. No processing, filtering, or buffering of the data takes place—the data is copied and sent **1:1**. The distribution is strictly deterministic and without any additional delay (apart from the propagation time of the internal connection).

Technical Features

- **Generic Type**: The function block is identified by the attribute `GenericClassName` as `'GEN_AUI_SPLIT'` and can be instantiated at runtime using type-hash mechanisms (`TypeHash` is currently empty).
- **No ECC (Execution Control Chart)**: Since there is no event-driven logic, the function block does not have a state machine. The routing is purely data-driven.
- **EPL 2.0 Licensed**: The function block is provided under the Eclipse Public License 2.0.

State Overview

The AUI_SPLIT_2_UNGATED function block has **no state machine**. There are no internal states or transitions – the signal is continuously routed without clocking.

Application Scenarios

- **Signal Distribution**: An AUI data stream needs to be transmitted in parallel to two independent receivers or subsystems (e.g., two controllers, displays, or actuators).
- **Redundancy**: Identical data should be provided on two paths to support failover mechanisms.
- **Protocol Multicasting**: A unidirectional AUI stream is distributed to multiple consumers without requiring the sender to be addressed multiple times.

Comparison with Similar Components

- **AUI_SPLIT_N**: An extended variant that splits one input into N outputs (e.g., 1:3, 1:4). The AUI_SPLIT_2_UNGATED is the simplest member of this family.
- **AUI_MERGE_2**: Combines two AUI inputs into one output (optionally with arbitration). The splitter operates in the opposite direction.
- **AUI_PASS**: A pure 1:1 pass-through adapter without branching.

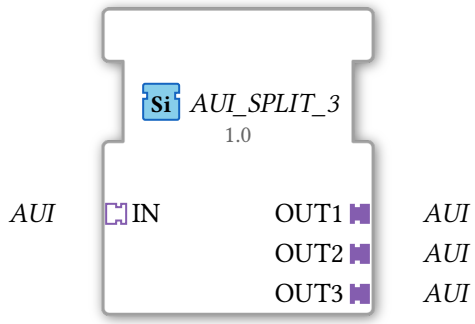
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AUI_SPLIT_2_UNGATED** is a minimalist yet useful function block for splitting a unidirectional AUI signal into two outputs. Thanks to its generic implementation and the absence of complex logic, it is ideally suited for scenarios requiring clean, lossless duplication of data streams.

AUI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_3** serves as a generic splitter that distributes a single incoming AUI interface to three identical outgoing AUI interfaces. It allows for the simple duplication of a unidirectional adapter (of type `adapter::types::unidirectional::AUI`) without altering the events or data it carries.

Interface Structure

Event Inputs

None – the function block has no independent event inputs. Event forwarding occurs exclusively via the AUI adapters.

Event Outputs

None – the function block has no independent event outputs. Event forwarding occurs exclusively via the AUI adapters.

Data Inputs

None – the FB has no independent data inputs. Data is transmitted via the AUI adapters.

Data Outputs

None – the FB has no independent data outputs. Data is transmitted via the AUI adapters.

Adapters

Direction	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::AUI	Incoming AUI interface distributed across three outputs.
Plug (Output)	OUT1	adapter::types::unidirectional::AUI	First outgoing AUI output (identical to the input signal).
Plug (Output)	OUT2	adapter::types::unidirectional::AUI	Second outgoing AUI output (identical to the input signal).
Plug (Output)	OUT3	adapter::types::unidirectional::AUI	Third outgoing AUI output (identical to the input signal).

Functionality

The module functions purely as a signal distributor. It receives an AUI interface via socket `IN` and forwards all incoming events and data unchanged to the three plugs `OUT1`, `OUT2`, and `OUT3`. No processing, delay, or state change takes place. The function block (FB) is therefore **stateless** and behaves like a simple replication of wiring.

Technical Features

- **Generic Structure**: The function block is labeled as a generic FB (`GEN_AUI_SPLIT`), allowing it to be used with various AUI adapter variants (with different event/data signatures).
- **Unidirectional**: The adapter type `unidirectional` means that data and event flows are only in one direction (from the socket to the plugs). Backward communication is not supported.
- **No Latency**: Due to the absence of internal logic, no measurable delay occurs.
- **No Configuration**: The FB requires no parameters – the number of outputs is fixed at three.

State Overview

The function block does not have an internal state machine. There is only one continuous operating state in which the input interface is permanently mirrored to the three output interfaces.

Application Scenarios

- **Signal Multiplication**: An AUI signal from a sensor or controller is to be forwarded to multiple independent receivers (e.g., actuators, displays, other controllers).
- **Monitoring**: An existing AUI data stream is copied to a monitoring unit without affecting the original signal paths.
- **Test and Simulation Environments**: A test stimulus is to be distributed in parallel to multiple components under test.

Comparison with Similar Components

Component	Outputs	Properties
`AUI_SPLIT_2`	2	Same functionality, but only two outputs.
`AUI_SPLIT_4`	4	Extended version with four outputs.
`AUI_MERGE`	–	Combines multiple AUI inputs into one output (opposite function).

The `AUI_SPLIT_3` represents a specific configuration that provides exactly three identical outputs. It can be replaced by other splitter variants if needed.

Change Detection

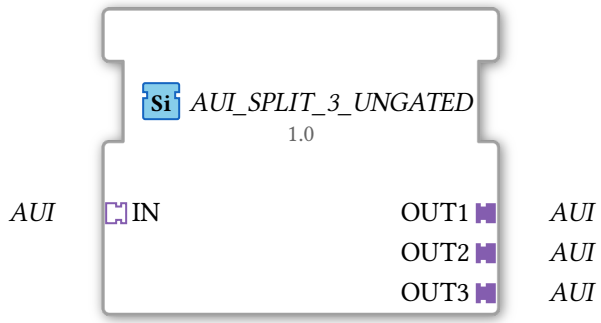
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AUI_SPLIT_3` is a simple yet useful generic function block for multiplying a unidirectional AUI interface. Thanks to its pure pass-through function and lack of state logic, it is reliable, performant, and easily integrated into existing automation solutions. It is particularly suitable for applications where a signal needs to be split across multiple paths.

AUI_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_3_UNGATED** serves as a generic splitter that distributes a single incoming AUI interface to three identical outgoing AUI interfaces. It allows for the simple duplication of a unidirectional adapter (of type `adapter::types::unidirectional::AUI`) without altering the events or data it carries.

Interface Structure

Event Inputs

None – the function block has no independent event inputs. Event forwarding occurs exclusively via the AUI adapters.

Event Outputs

None – the function block has no independent event outputs. Event forwarding occurs exclusively via the AUI adapters.

Data Inputs

None – the FB has no independent data inputs. Data is transmitted via the AUI adapters.

Data Outputs

None – the FB has no independent data outputs. Data is transmitted via the AUI adapters.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AUI`	Incoming AUI interface distributed across three outputs.
Plug (Output)	`OUT1`	`adapter::types::unidirectional::AUI`	First outgoing AUI output (identical to the input signal).

Plug (Output)	OUT2	adapter::types::unidirectional::AUI	Second outgoing AUI output (identical to the input signal).
Plug (Output)	OUT3	adapter::types::unidirectional::AUI	Third outgoing AUI output (identical to the input signal).

Functionality

The module functions purely as a signal distributor. It receives an AUI interface via socket `IN` and forwards all incoming events and data unchanged to the three plugs `OUT1`, `OUT2`, and `OUT3`. No processing, delay, or state change takes place. The function block (FB) is therefore **stateless** and behaves like a simple replication of wiring.

Technical Features

- **Generic Structure**: The function block is labeled as a generic FB (`GEN_AUI_SPLIT`), allowing it to be used with various AUI adapter variants (with different event/data signatures).
- **Unidirectional**: The adapter type `unidirectional` means that data and event flows are only in one direction (from the socket to the plugs). Backward communication is not supported.
- **No Latency**: Due to the absence of internal logic, no measurable delay occurs.
- **No Configuration**: The FB requires no parameters – the number of outputs is fixed at three.

State Overview

The function block does not have an internal state machine. There is only one continuous operating state in which the input interface is permanently mirrored to the three output interfaces.

Application Scenarios

- **Signal Multiplication**: An AUI signal from a sensor or controller is to be forwarded to multiple independent receivers (e.g., actuators, displays, other controllers).
- **Monitoring**: An existing AUI data stream is copied to a monitoring unit without affecting the original signal paths.
- **Test and Simulation Environments**: A test stimulus is to be distributed in parallel to multiple components under test.

Comparison with Similar Components

Component	Outputs	Properties
`AUI_SPLIT_2`	2	Same functionality, but only two outputs.
`AUI_SPLIT_4`	4	Extended version with four outputs.
`AUI_MERGE`	–	Combines multiple AUI inputs into one output (opposite function).

The `AUI_SPLIT_3_UNGATED` represents a specific configuration that provides exactly three identical outputs. It can be replaced by other splitter variants if needed.

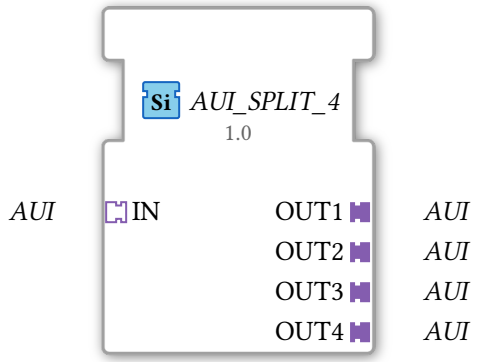
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUI_SPLIT_3_UNGATED` is a simple yet useful generic function block for multiplying a unidirectional AUI interface. Thanks to its pure pass-through function and lack of state logic, it is reliable, performant, and easily integrated into existing automation solutions. It is particularly suitable for applications where a signal needs to be split across multiple paths.

AUI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_4** serves as a generic distributor for AUI signals. It accepts a single AUI input (socket) and routes it to four separate AUI outputs (plugs). This block is typically used in automation technology when a unidirectional signal is required multiple times in parallel.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AUI`	Input signal (AUI)
Plug	`OUT1`	`adapter::types::unidirectional::AUI`	First output (identical to IN)
Plug	`OUT2`	`adapter::types::unidirectional::AUI`	Second output (identical to IN)
Plug	`OUT3`	`adapter::types::unidirectional::AUI`	Third output (identical to IN)
Plug	`OUT4`	`adapter::types::unidirectional::AUI`	Fourth output (identical to IN)

Functionality

This function block does not perform any data manipulation. It acts as a passive distribution unit: As soon as a signal is present via adapter `IN`, it is passed on to all four output adapters `OUT1`–`OUT4` without delay or modification. The signal direction is unidirectional, from the socket to the plugs.

Technical Features

- **Generic Type:** The function block is declared as a generic function block (`GenericClassName = 'GEN_AUI_SPLIT'`). It can therefore be used with various implementations of the AUI adapter type.
- **No States:** Since there are no events or algorithms, the function block does not have an internal state machine.
- **License:** The source code is available under the **Eclipse Public License 2.0**.

State Overview

This function block has no event inputs or outputs and does not execute any algorithm. There is no explicit state machine. The function block behaves like a simple signal distributor (wire connection).

Application Scenarios

- **Signal Cascading:** A sensor signal (e.g., an AUI-based clock signal) needs to be distributed to multiple actuators.
- **Test Environments:** A test signal is to be routed in parallel to multiple devices under test.
- **Redundancy:** A signal is split across multiple parallel paths to ensure fault tolerance.
- **Bus Extension:** As a passive splitter in an AUI communication system.

Comparison with Similar Function Blocks

- **AUI_SPLIT_2** – Splits one input into two outputs (analog principle, fewer outputs).
- **AUI_MERGE** – Combines multiple AUI inputs into a single output (counterpart).
- **AUI_SELECT** – Selects one of several inputs using a control signal (with selection function).
- Unlike these function blocks, `AUI_SPLIT_4` offers a pure 1:4 distribution without selection logic or merging.

Change Detection

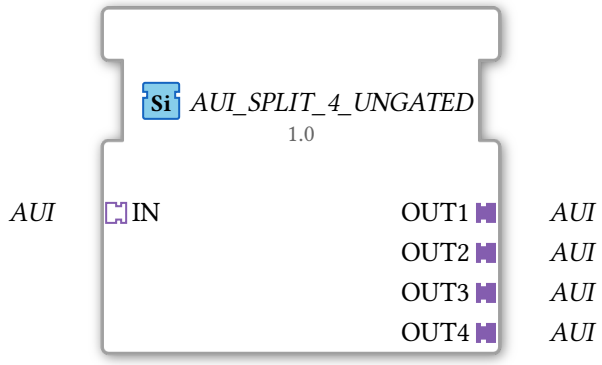
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AUI_SPLIT_4** function block is a simple yet useful component for passively multiplying AUI signals. Its generic nature and clear interface make it a flexible tool in automation development, especially when multiple identical receivers need to be supplied with the same signal.

AUI_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_4_UNGATED** serves as a generic distributor for AUI signals. It accepts a single AUI input (socket) and routes it to four separate AUI outputs (plugs). This block is typically used in automation technology when a unidirectional signal is required multiple times in parallel.

Interface Structure

Event Inputs
None available.

Event Outputs
None available.

Data Inputs
None available.

Data Outputs
None available.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AUI`	Input signal (AUI)
Plug	`OUT1`	`adapter::types::unidirectional::AUI`	First output (identical to IN)
Plug	`OUT2`	`adapter::types::unidirectional::AUI`	Second output (identical to IN)
Plug	`OUT3`	`adapter::types::unidirectional::AUI`	Third output (identical to IN)
Plug	`OUT4`	`adapter::types::unidirectional::AUI`	Fourth output (identical to IN)

Functionality
This function block does not perform any data manipulation. It acts as a passive distribution unit: As soon as a signal is present via adapter `IN`, it is passed on to all four output adapters `OUT1`–`OUT4` without delay or modification. The signal direction is unidirectional, from the socket to the plugs.

Technical Features
- **Generic Type:** The function block is declared as a generic function block (`GenericClassName = 'GEN_AUI_SPLIT'`). It can therefore be used with various implementations of the AUI adapter type.
- **No States:** Since there are no events or algorithms, the function block does not have an internal state machine.
- **License:** The source code is available under the **Eclipse Public License 2.0**.

State Overview
This function block has no event inputs or outputs and does not execute any algorithm. There is no explicit state machine. The function block behaves like a simple signal distributor (wire connection).

Application Scenarios
- **Signal Cascading:** A sensor signal (e.g., an AUI-based clock signal) needs to be distributed to multiple actuators.
- **Test Environments:** A test signal is to be routed in parallel to multiple devices under test.
- **Redundancy:** A signal is split across multiple parallel paths to ensure fault tolerance.
- **Bus Extension:** As a passive splitter in an AUI communication system.

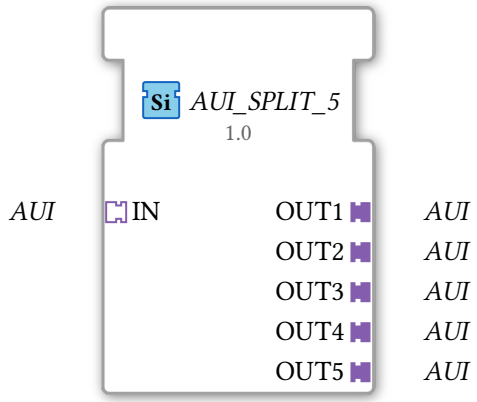
Comparison with Similar Function Blocks
- **AUI_SPLIT_2** – Splits one input into two outputs (analog principle, fewer outputs).
- **AUI_MERGE** – Combines multiple AUI inputs into a single output (counterpart).
- **AUI_SELECT** – Selects one of several inputs using a control signal (with selection function).
- Unlike these function blocks, `AUI_SPLIT_4_UNGATED` offers a pure 1:4 distribution without selection logic or merging.

Change Detection
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AUI_SPLIT_4_UNGATED** function block is a simple yet useful component for passively multiplying AUI signals. Its generic nature and clear interface make it a flexible tool in automation development, especially when multiple identical receivers need to be supplied with the same signal.

AUI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_5** is a generic building block for splitting an incoming AUI adapter (Automation Unit Interface) into five separate, identical AUI outputs. It implements a 1:5 distribution in a unidirectional data direction. Designed for use in automation technology, it enables the redundant or distributed forwarding of AUI signals.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available. Signals are transmitted exclusively via the adapter interfaces.

Data Outputs

No data outputs available.

Adapters

The module features six adapter interfaces of type `adapter::types::unidirectional::AUI`:

- **IN** (Socket) – Input adapter for the AUI signal to be distributed.
- **OUT1** to **OUT5** (Plugs) – Five output adapters that mirror the incoming signal.

All adapters are unidirectional and transmit data and, optionally, associated events according to the AUI specification.

Functionality

The module functions as a pure splitter without logic or buffering. An AUI signal present at socket **IN** is copied in parallel and unchanged to all five plugs **OUT1** to **OUT5**. The forwarding is instantaneous, i.e., without delay due to internal processing. The individual outputs are not galvanically or logically isolated – a change at the input immediately affects all outputs.

Technical Features

- **Generic Type**: The function block (FB) is declared as a generic block (attribute `eclipse4diac::core::GenericClassName = 'GEN_AUI_SPLIT'`). This allows for later type adaptation or instantiation with a specific AUI type.
- **No State Maintenance**: Since there are no events or data that react to changes, the FB is stateless. A state machine is not required.
- **Full Bandwidth**: The outputs provide the input signal without restriction – all data and event channels of the AUI adapter are copied 1:1.

State Overview

The function block does not have its own states or state machines. It behaves like passive wiring. A state overview is therefore not applicable.

Application Scenarios

- **Signal Distribution**: Splitting a central AUI data stream to multiple parallel consumers (e.g., displays, monitoring units, controllers).
- **Test and Diagnostic Setups**: Used in test benches to simultaneously send a signal to multiple measuring devices or logging systems.
- **Redundancy**: Provides identical signals for failover systems when AUI communication is unidirectional.
- **Bus Coupling**: Used as a passive splitter in AUI-based line or star topologies.

Comparison with Similar Components

- **AUI_SPLIT_2 / AUI_SPLIT_3 / AUI_SPLIT_4**: Analog components with fewer outputs; scaling is achieved solely by selecting the appropriate split type.
- **AUI_MERGE (e.g., AUI_MERGE_5)**: Combines multiple AUI inputs into a single output – the opposite function.
- **Pure Event Splitters (E_SPLIT)**: Share only events, not data adapters. The AUI_SPLIT_5, however, distributes complete adapter interfaces.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

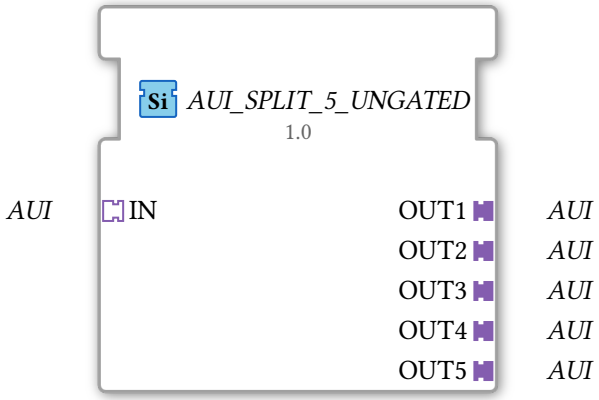
The **AUI_SPLIT_5** is a simple yet essential component for multiplying AUI signals. Its generic design and the adapter's complete passthrough make it highly versatile. It represents an optimal solution for applications requiring unidirectional 1:5 distribution while minimizing latency.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AUI_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_5_UNGATED** is a generic building block for splitting an incoming AUI adapter (Automation Unit Interface) into five separate, identical AUI outputs. It implements a 1:5 distribution in a unidirectional data direction. Designed for use in automation technology, it enables the redundant or distributed forwarding of AUI signals.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available. Signals are transmitted exclusively via the adapter interfaces.

Data Outputs

No data outputs available.

Adapters

The module features six adapter interfaces of type `adapter::types::unidirectional::AUI`:

- **IN** (Socket) – Input adapter for the AUI signal to be distributed.
- **OUT1** to **OUT5** (Plugs) – Five output adapters that mirror the incoming signal.

All adapters are unidirectional and transmit data and, optionally, associated events according to the AUI specification.

Functionality

The module functions as a pure splitter without logic or buffering. An AUI signal present at socket **IN** is copied in parallel and unchanged to all five plugs **OUT1** to **OUT5**. The forwarding is instantaneous, i.e., without delay due to internal processing. The individual outputs are not galvanically or logically isolated – a change at the input immediately affects all outputs.

Technical Features

- **Generic Type**: The function block (FB) is declared as a generic block (attribute `eclipse4diac::core::GenericClassName = 'GEN_AUI_SPLIT'`). This allows for later type adaptation or instantiation with a specific AUI type.
- **No State Maintenance**: Since there are no events or data that react to changes, the FB is stateless. A state machine is not required.
- **Full Bandwidth**: The outputs provide the input signal without restriction – all data and event channels of the AUI adapter are copied 1:1.

State Overview

The function block does not have its own states or state machines. It behaves like passive wiring. A state overview is therefore not applicable.

Application Scenarios

- **Signal Distribution**: Splitting a central AUI data stream to multiple parallel consumers (e.g., displays, monitoring units, controllers).
- **Test and Diagnostic Setups**: Used in test benches to simultaneously send a signal to multiple measuring devices or logging systems.
- **Redundancy**: Provides identical signals for failover systems when AUI communication is unidirectional.
- **Bus Coupling**: Used as a passive splitter in AUI-based line or star topologies.

Comparison with Similar Components

- **AUI_SPLIT_2 / AUI_SPLIT_3 / AUI_SPLIT_4**: Analog components with fewer outputs; scaling is achieved solely by selecting the appropriate split type.
- **AUI_MERGE (e.g., AUI_MERGE_5)**: Combines multiple AUI inputs into a single output – the opposite function.
- **Pure Event Splitters (E_SPLIT)**: Share only events, not data adapters. The AUI_SPLIT_5_UNGATED, however, distributes complete adapter interfaces.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

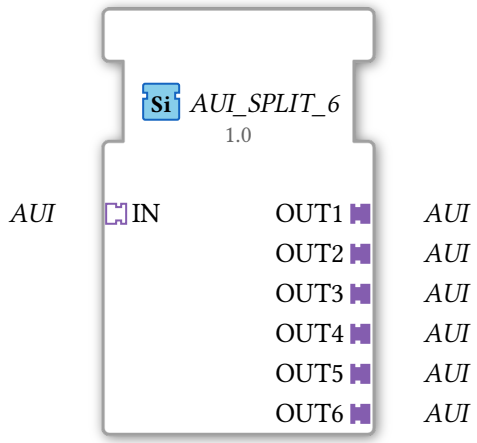
Conclusion

The **AUI_SPLIT_5_UNGATED** is a simple yet essential component for multiplying AUI signals. Its generic design and the adapter's complete passthrough make it highly versatile. It represents an optimal solution for applications requiring unidirectional 1:5 distribution while minimizing latency.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AUI_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_6** is used to split an incoming **AUI adapter** (Application Universal Interface) into six outgoing AUI adapters. It is designed as a generic function block capable of distributing the incoming AUI signal to all six outputs without any data processing or logic.

Interface Structure

Event Inputs
No event inputs available.

Event Outputs
No event outputs available.

Data Inputs
No data inputs available.

Data Outputs
No data outputs available.

Adapters
The function block uses adapters exclusively for communication:

Direction	Name	Type	Description
Input	**IN**	`adapter::types::unidirectional::AUI`	Incoming AUI adapter being split.
Output	**OUT1**	`adapter::types::unidirectional::AUI`	First outgoing AUI adapter.
Output	**OUT2**	`adapter::types::unidirectional::AUI`	Second outgoing AUI adapter.
Output	**OUT3**	`adapter::types::unidirectional::AUI`	Third outgoing AUI adapter.
Output	**OUT4**	`adapter::types::unidirectional::AUI`	Fourth outgoing AUI adapter.
Output	**OUT5**	`adapter::types::unidirectional::AUI`	Fifth outgoing AUI adapter.
Output	**OUT6**	`adapter::types::unidirectional::AUI`	Sixth outgoing AUI adapter.

Functionality
The function block forwards the AUI signal arriving at the adapter input **IN** to all six adapter outputs (**OUT1** to **OUT6**) without any delay or processing. No data manipulation takes place – the function block acts purely as a splitter for AUI signals.

Technical Features
- The function block is declared as a **generic function block**: The attribute `eclipse4diac::core::GenericClassName` is set to `'GEN_AUI_SPLIT'`.
- There is no state management or event logic – the function block is purely combinatorial.
- The outputs are independent of each other; A fault at one output does not affect the others.
- By using AUI adapters, the function block is optimized for unidirectional signal transmission.

State Overview
The function block has **no states**. It operates continuously without an internal state machine. The input signal is immediately and continuously mirrored to all outputs.

Application Scenarios
- **Signal Distribution in Control Engineering**: When an AUI signal (e.g., a measured value or control command) needs to be passed on to multiple devices.
- **Redundancy or Parallel Processing**: Distribution of a signal to multiple identical downstream functional units.
- **Test and Simulation Setups**: Splitting an AUI signal for simultaneous monitoring and processing.

Comparison with Similar Function Blocks
- **AUI_SPLIT_2 / AUI_SPLIT_4**: These function blocks split a signal across two or four outputs, respectively. **AUI_SPLIT_6** offers the largest number (6) of outputs, thus covering applications with many parallel destinations.
- **AUI_MERGE_***: Unlike splitters, merge blocks combine multiple AUI signals into one.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AUI_SPLIT_6** is a simple yet effective distribution block for AUI signals. It enables the star-shaped routing of an input signal to up to six independent outputs. Due to its generic nature and lack of logic, it is ideally suited for flexible and expandable automation solutions.

–

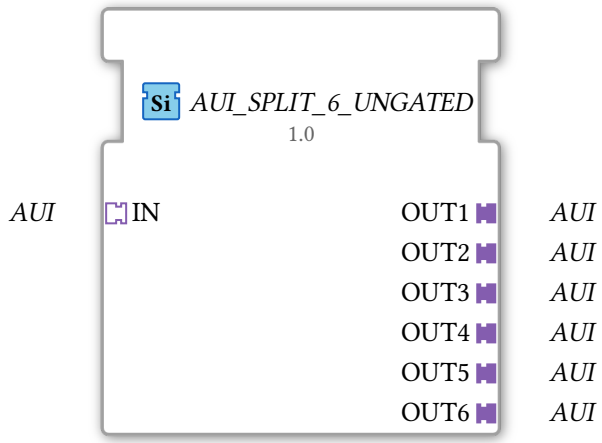
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AUI_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_6_UNGATED** is used to split an incoming **AUI adapter** (Application Universal Interface) into six outgoing AUI adapters. It is designed as a generic function block capable of distributing the incoming AUI signal to all six outputs without any data processing or logic.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

The function block uses adapters exclusively for communication:

Direction	Name	Type	Description
Input	**IN**	`adapter::types::unidirectional::AUI`	Incoming AUI adapter being split.
Output	**OUT1**	`adapter::types::unidirectional::AUI`	First outgoing AUI adapter.
Output	**OUT2**	`adapter::types::unidirectional::AUI`	Second outgoing AUI adapter.
Output	**OUT3**	`adapter::types::unidirectional::AUI`	Third outgoing AUI adapter.
Output	**OUT4**	`adapter::types::unidirectional::AUI`	Fourth outgoing AUI adapter.
Output	**OUT5**	`adapter::types::unidirectional::AUI`	Fifth outgoing AUI adapter.
Output	**OUT6**	`adapter::types::unidirectional::AUI`	Sixth outgoing AUI adapter.

Functionality

The function block forwards the AUI signal arriving at the adapter input **IN** to all six adapter outputs (**OUT1** to **OUT6**) without any delay or processing. No data manipulation takes place – the function block acts purely as a splitter for AUI signals.

Technical Features

- The function block is declared as a **generic function block**: The attribute `eclipse4diac::core::GenericClassName` is set to `'GEN_AUI_SPLIT'`.
- There is no state management or event logic – the function block is purely combinatorial.
- The outputs are independent of each other; A fault at one output does not affect the others.
- By using AUI adapters, the function block is optimized for unidirectional signal transmission.

State Overview

The function block has **no states**. It operates continuously without an internal state machine. The input signal is immediately and continuously mirrored to all outputs.

Application Scenarios

- **Signal Distribution in Control Engineering**: When an AUI signal (e.g., a measured value or control command) needs to be passed on to multiple devices.
- **Redundancy or Parallel Processing**: Distribution of a signal to multiple identical downstream functional units.
- **Test and Simulation Setups**: Splitting an AUI signal for simultaneous monitoring and processing.

Comparison with Similar Function Blocks

- **AUI_SPLIT_2 / AUI_SPLIT_4**: These function blocks split a signal across two or four outputs, respectively. **AUI_SPLIT_6_UNGATED** offers the largest number (6) of outputs, thus covering applications with many parallel destinations.
- **AUI_MERGE_***: Unlike splitters, merge blocks combine multiple AUI signals into one.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AUI_SPLIT_6_UNGATED** is a simple yet effective distribution block for AUI signals. It enables the star-shaped routing of an input signal to up to six independent outputs. Due to its generic nature and lack of logic, it is ideally suited for flexible and expandable automation solutions.

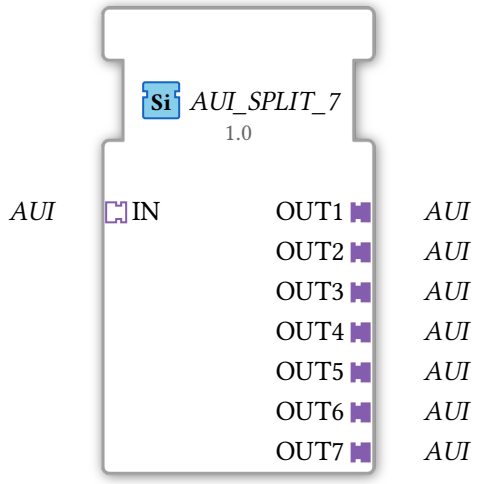
–

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AUI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_SPLIT_7` is used to distribute an incoming AUI (Application User Interface) signal to up to seven identical outputs. It is designed as a generic block and operates exclusively via adapter interfaces – without event or data inputs/outputs in the traditional sense. This makes it particularly suitable for pure signal distribution in adapter-based architectures.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapter

Type	Direction	Name
Socket (adapter::types::unidirectional::AUI)	Input	IN
Plug (adapter::types::unidirectional::AUI)	Output	OUT1
Plug (adapter::types::unidirectional::AUI)	Output	OUT2
Plug (adapter::types::unidirectional::AUI)	Output	OUT3
Plug (adapter::types::unidirectional::AUI)	Output	OUT4
Plug (adapter::types::unidirectional::AUI)	Output	OUT5
Plug (adapter::types::unidirectional::AUI)	Output	OUT6

Plug (adapter::types::unidirectional::AUI)	Output	OUT7

Functionality

This module represents a 1-to-7 distribution logic for AUI adapters. The signal received via socket `IN` is replicated to all seven plugs (`OUT1` … `OUT7`) without further processing or logical conditions. The forwarding is purely passive: each received signal is simultaneously made available at every output.

Technical Features

- **Pure Adapter Module**: The module contains neither event nor data inputs/outputs – all communication takes place via the AUI adapters.
- **Unidirectional**: The adapters are of type `unidirectional`, meaning that data flows only from the input (socket) to the outputs (plugs).
- **Generic Type**: The function block (FB) is implemented as a generic block (`GenericClassName = 'GEN_AUI_SPLIT'`) and can be reused for various AUI subtypes.
- **No State Logic**: Due to its pure distribution function, the block has no internal state machine – it operates continuously and without delay.

State Overview

There is no state machine. The block is purely data flow controlled and has only one static operating state.

Application Scenarios

- **Signal Distribution in Field Automation**: An AUI signal (e.g., sensor data or control commands) coming from a central controller is forwarded to several parallel actuators or display units.
- **Test and Simulation Environments**: Used as a splitter to simultaneously output an input signal to multiple simulation or diagnostic units.
- **Modular Systems**: In adapter-based architectures (e.g., according to IEC 61499), this function block is used to distribute a single interface across multiple downstream function blocks.

Comparison with Similar Function Blocks

- **AUI_SPLIT_2, AUI_SPLIT_4** – Function blocks with the same functionality, but reduced to 2 or 4 outputs, respectively. The `AUI_SPLIT_7` offers the largest number of outputs and thus covers more extensive distribution requirements.
- **AUI_MERGE** – the counterpart that combines multiple AUI inputs into a single output. In contrast, the `SPLIT` is purely passive and does not aggregate any data.

Change Detection

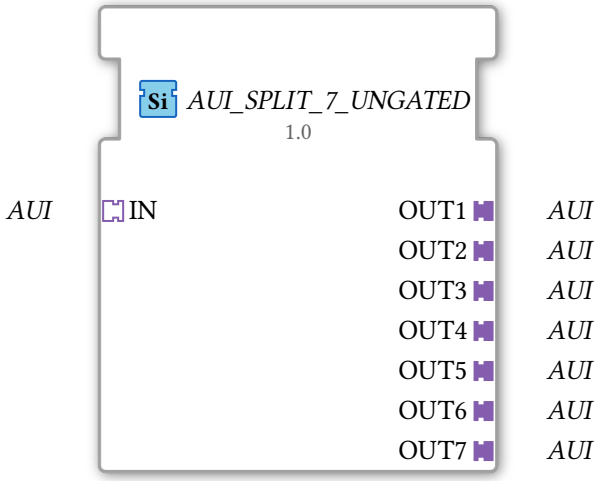
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AUI_SPLIT_7` is a simple yet essential function block for unidirectional signal distribution in adapter-based control systems. Thanks to its generic design and clear 1-to-N structure, it can be flexibly integrated into any AUI-based architecture. For applications requiring broad parallel distribution without side effects, it represents a robust and maintainable solution.

AUI_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_SPLIT_7_UNGATED` is used to distribute an incoming AUI (Application User Interface) signal to up to seven identical outputs. It is designed as a generic block and operates exclusively via adapter interfaces – without event or data inputs/outputs in the traditional sense. This makes it particularly suitable for pure signal distribution in adapter-based architectures.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapter

Type	Direction	Name
Socket (adapter::types::unidirectional::AUI)	Input	IN
Plug (adapter::types::unidirectional::AUI)	Output	OUT1
Plug (adapter::types::unidirectional::AUI)	Output	OUT2
Plug (adapter::types::unidirectional::AUI)	Output	OUT3
Plug (adapter::types::unidirectional::AUI)	Output	OUT4
Plug (adapter::types::unidirectional::AUI)	Output	OUT5
Plug (adapter::types::unidirectional::AUI)	Output	OUT6
Plug (adapter::types::unidirectional::AUI)	Output	OUT7

Functionality

This module represents a 1-to-7 distribution logic for AUI adapters. The signal received via socket `IN` is replicated to all seven plugs (`OUT1` ... `OUT7`) without further processing or logical conditions. The forwarding is purely passive: each received signal is simultaneously made available at every output.

Technical Features

- **Pure Adapter Module**: The module contains neither event nor data inputs/outputs – all communication takes place via the AUI adapters.
- **Unidirectional**: The adapters are of type `unidirectional`, meaning that data flows only from the input (socket) to the outputs (plugs).
- **Generic Type**: The function block (FB) is implemented as a generic block (`GenericClassName = 'GEN_AUI_SPLIT'`) and can be reused for various AUI subtypes.
- **No State Logic**: Due to its pure distribution function, the block has no internal state machine – it operates continuously and without delay.

State Overview

There is no state machine. The block is purely data flow controlled and has only one static operating state.

Application Scenarios

- **Signal Distribution in Field Automation**: An AUI signal (e.g., sensor data or control commands) coming from a central controller is forwarded to several parallel actuators or display units.
- **Test and Simulation Environments**: Used as a splitter to simultaneously output an input signal to multiple simulation or diagnostic units.
- **Modular Systems**: In adapter-based architectures (e.g., according to IEC 61499), this function block is used to distribute a single interface across multiple downstream function blocks.

Comparison with Similar Function Blocks

- **AUI_SPLIT_2, AUI_SPLIT_4** – Function blocks with the same functionality, but reduced to 2 or 4 outputs, respectively. The `AUI_SPLIT_7_UNGATED` offers the largest number of outputs and thus covers more extensive distribution requirements.
- **AUI_MERGE** – the counterpart that combines multiple AUI inputs into a single output. In contrast, the `SPLIT` is purely passive and does not aggregate any data.

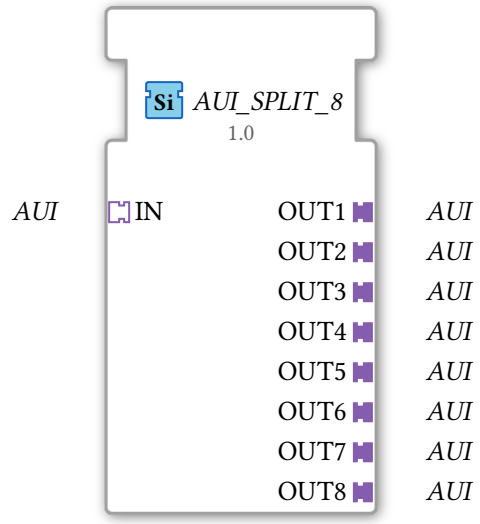
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUI_SPLIT_7_UNGATED` is a simple yet essential function block for unidirectional signal distribution in adapter-based control systems. Thanks to its generic design and clear 1-to-N structure, it can be flexibly integrated into any AUI-based architecture. For applications requiring broad parallel distribution without side effects, it represents a robust and maintainable solution.

AUI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_SPLIT_8` is used to distribute an incoming AUI adapter signal (unidirectional) to eight identical output adapters. It functions as a passive splitter and enables the simple distribution of a signal path to multiple downstream components. The function block is generic and uses the adapter pattern of IEC 61499.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Type	Direction
IN	`adapter::types::unidirectional::AUI`	Socket (Input)
OUT1	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT2	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT3	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT4	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT5	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT6	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT7	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT8	`adapter::types::unidirectional::AUI`	Plug (Output)

Functionality

This module forwards all data and events received via the `IN` socket of the unidirectional AUI adapter to all eight `OUT` adapters. No logical or temporal processing takes place; the forwarding is one-to-one and without latency. The adapters are of the same type (`AUI`), so the complete signature (data, events, and included interfaces) is passed through unchanged.

Technical Features

- **Generic Type**: The component is declared as generic using `GenericClassName` (`GEN_AUI_SPLIT`). It can therefore be used for various AUI adapter variants (e.g., different data widths), provided the underlying type is `adapter::types::unidirectional::AUI`.
- **Unidirectional**: The adapters are unidirectional – data transmission is only possible in one direction (from IN to OUT). Return channels are not included.
- **No State Logic**: The component has no internal states, algorithms, or event processing. It is purely structural (passive wiring).

State Overview

The component has no states. It behaves statically and always passes through all incoming signals.

Application Scenarios

- **Signal Distribution in Automation Networks**: A sensor or controller provides an AUI-compliant signal that needs to be distributed to multiple actuators or subsystems.
- **Test and Simulation Environments**: Distributing a test signal across multiple parallel test paths.
- **Redundant Signal Paths**: Simultaneously supplying multiple independent controllers with the same information.

Comparison with Similar Components

- **`AUI_MERGE_8`** (counterpart): Combines eight AUI inputs into one output (inverse operator).
- **`AUI_SPLIT_2`, `AUI_SPLIT_4`**: Same functionality with fewer outputs.
- **Event Splitters (e.g., `E_SPLIT`)**: Distribute only events without the accompanying data from an adapter. `AUI_SPLIT_8`, on the other hand, replicates the entire adapter content.

Change Detection

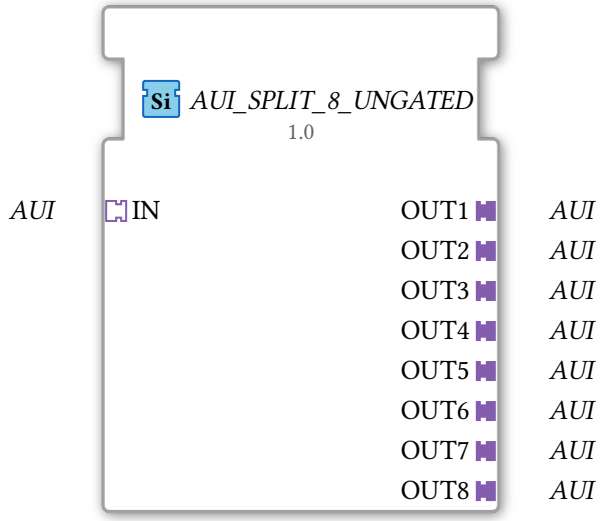
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AUI_SPLIT_8` is a simple yet essential component for replicating a unidirectional AUI signal path. Due to its generic design and purely passive structure, it is suitable for any AUI-based application where a signal needs to be distributed to up to eight receivers. The implementation is lean, error-free, and requires no configuration.

AUI_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_SPLIT_8_UNGATED` is used to distribute an incoming AUI adapter signal (unidirectional) to eight identical output adapters. It functions as a passive splitter and enables the simple distribution of a signal path to multiple downstream components. The function block is generic and uses the adapter pattern of IEC 61499.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Type	Direction
IN	`adapter::types::unidirectional::AUI`	Socket (Input)
OUT1	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT2	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT3	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT4	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT5	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT6	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT7	`adapter::types::unidirectional::AUI`	Plug (Output)
OUT8	`adapter::types::unidirectional::AUI`	Plug (Output)

Functionality

This module forwards all data and events received via the `IN` socket of the unidirectional AUI adapter to all eight `OUT` adapters. No logical or temporal processing takes place; the forwarding is one-to-one and without latency. The adapters are of the same type (`AUI`), so the complete signature (data, events, and included interfaces) is passed through unchanged.

Technical Features

- **Generic Type**: The component is declared as generic using `GenericClassName` (`GEN_AUI_SPLIT`). It can therefore be used for various AUI adapter variants (e.g., different data widths), provided the underlying type is `adapter::types::unidirectional::AUI`.
- **Unidirectional**: The adapters are unidirectional – data transmission is only possible in one direction (from IN to OUT). Return channels are not included.
- **No State Logic**: The component has no internal states, algorithms, or event processing. It is purely structural (passive wiring).

State Overview

The component has no states. It behaves statically and always passes through all incoming signals.

Application Scenarios

- **Signal Distribution in Automation Networks**: A sensor or controller provides an AUI-compliant signal that needs to be distributed to multiple actuators or subsystems.
- **Test and Simulation Environments**: Distributing a test signal across multiple parallel test paths.
- **Redundant Signal Paths**: Simultaneously supplying multiple independent controllers with the same information.

Comparison with Similar Components

- **`AUI_MERGE_8`** (counterpart): Combines eight AUI inputs into one output (inverse operator).
- **`AUI_SPLIT_2`, `AUI_SPLIT_4`**: Same functionality with fewer outputs.
- **Event Splitters (e.g., `E_SPLIT`)**: Distribute only events without the accompanying data from an adapter. `AUI_SPLIT_8_UNGATED`, on the other hand, replicates the entire adapter content.

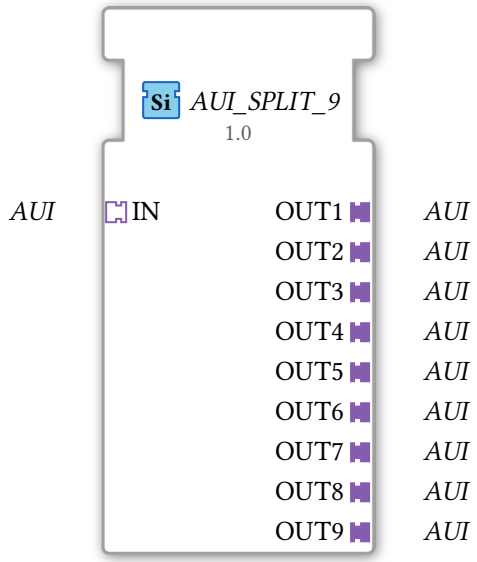
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUI_SPLIT_8_UNGATED` is a simple yet essential component for replicating a unidirectional AUI signal path. Due to its generic design and purely passive structure, it is suitable for any AUI-based application where a signal needs to be distributed to up to eight receivers. The implementation is lean, error-free, and requires no configuration.

AUI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_9** serves as a generic splitter for unidirectional AUI adapter connections. It receives an incoming AUI signal via socket `IN` and distributes it identically to nine separate output plugs (`OUT1` to `OUT9`). The block is typed as a generic function block (`GEN_AUI_SPLIT`) and can be used in various automation environments where an AUI signal needs to be passed on to multiple downstream components.

Interface Structure

Event Inputs

Not present. The function block operates purely on an adapter basis without event control.

Event Outputs

Not available.

Data Inputs

Not available.

Data Outputs

Not available.

Adapters

Socket (Input):

- `IN` – Type: `adapter::types::unidirectional::AUI` Receives the AUI signal to be distributed.

Plugs (Outputs):

- `OUT1` to `OUT9` – each Type: `adapter::types::unidirectional::AUI` Nine identical outputs that forward the incoming AUI signal unchanged.

The FB is a pure adapter splitter without additional logic and operates unidirectionally. All plugs are of the same type as the socket.

Functionality

The function block copies the AUI signal present at socket `IN` to all nine output plugs. No signal conditioning, delay, or filtering takes place – the outputs are exact replicas of the input. Distribution occurs in parallel and without data buffering. Since it is a generic function block, the specific signal type (e.g.,

fieldbus data, measured values, control commands) is not defined; it is determined by the AUI adapter definition.

Technical Features

- **Generic Type**: The function block is attributed as `GEN_AUI_SPLIT` with a type hash, which supports the dynamic creation of instances at runtime in the 4diac IDE.
- **No State Dependency**: The function block has no internal states or timing behavior – the outputs are always a direct switching of the input.
- **Extensibility**: The unidirectional adapter interface allows the FB to be integrated into any AUI-based communication chain.
- **Compatibility**: Requires the imports `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`, which must be available in the target 4diac environment.

State Overview

The FB does not have a state machine. Sequential processing does not occur; signal transmission is static and continuous. There is no explicit state.

Application Scenarios

- **Signal Distribution in Fieldbus Systems**: Splitting an AUI-based bus signal to multiple subsystems or actuators.
- **Test and Debugging Environments**: Connected downstream of a signal generator to supply multiple test points in parallel.
- **Redundancy or Parallel Operation**: When a signal needs to be sent to multiple controllers or monitoring units simultaneously.
- **Prototype Development**: Serves as a placeholder for later branching in the adapter logic.

Comparison with Similar Function Blocks

- **AUI_SPLIT_4 / AUI_SPLIT_16**: These function blocks offer the same functionality with 4 or 16 outputs, respectively. The choice depends on the required number of branches.
- **AUI_MERGE_***: A merger combines multiple AUI inputs into one output – the opposite function. A splitter, on the other hand, distributes the output.
- **Generic Splitters of Other Adapter Types (e.g., Data-Based)**: Unlike data-event splitters, AUI_SPLIT_9 operates purely at the adapter level and requires no event control, thus reducing implementation effort.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AUI_SPLIT_9** is a simple yet effective function block for distributing multiple unidirectional AUI signals. Its generic nature allows for flexible reuse in various projects. Due to its lack of inherent logic, this building block is resource-efficient and easy to understand. It is particularly well-suited for the rapid implementation of one-to-many connections in adapter-based automation architectures.

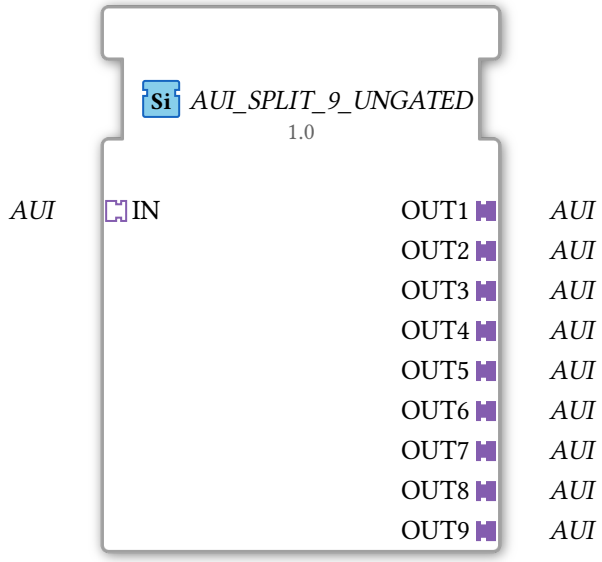
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AUI_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_SPLIT_9_UNGATED** serves as a generic splitter for unidirectional AUI adapter connections. It receives an incoming AUI signal via socket `IN` and distributes it identically to nine separate output plugs (`OUT1` to `OUT9`). The block is typed as a generic function block (`GEN_AUI_SPLIT`) and can be used in various automation environments where an AUI signal needs to be passed on to multiple downstream components.

Interface Structure

Event Inputs

Not present. The function block operates purely on an adapter basis without event control.

Event Outputs

Not available.

Data Inputs

Not available.

Data Outputs

Not available.

Adapters

Socket (Input):

- `IN` – Type: `adapter::types::unidirectional::AUI` Receives the AUI signal to be distributed.

Plugs (Outputs):

- `OUT1` to `OUT9` – each Type: `adapter::types::unidirectional::AUI` Nine identical outputs that forward the incoming AUI signal unchanged.

The FB is a pure adapter splitter without additional logic and operates unidirectionally. All plugs are of the same type as the socket.

Functionality

The function block copies the AUI signal present at socket `IN` to all nine output plugs. No signal conditioning, delay, or filtering takes place – the outputs are exact replicas of the input. Distribution occurs in parallel and without data buffering. Since it is a generic function block, the specific signal type (e.g., fieldbus data, measured values, control commands) is not defined; it is determined by the AUI adapter definition.

Technical Features

- **Generic Type**: The function block is attributed as `GEN_AUI_SPLIT` with a type hash, which supports the dynamic creation of instances at runtime in the 4diac IDE.
- **No State Dependency**: The function block has no internal states or timing behavior – the outputs are always a direct switching of the input.
- **Extensibility**: The unidirectional adapter interface allows the FB to be integrated into any AUI-based communication chain.
- **Compatibility**: Requires the imports `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`, which must be available in the target 4diac environment.

State Overview

The FB does not have a state machine. Sequential processing does not occur; signal transmission is static and continuous. There is no explicit state.

Application Scenarios

- **Signal Distribution in Fieldbus Systems**: Splitting an AUI-based bus signal to multiple subsystems or actuators.
- **Test and Debugging Environments**: Connected downstream of a signal generator to supply multiple test points in parallel.
- **Redundancy or Parallel Operation**: When a signal needs to be sent to multiple controllers or monitoring units simultaneously.
- **Prototype Development**: Serves as a placeholder for later branching in the adapter logic.

Comparison with Similar Function Blocks

- **AUI_SPLIT_4 / AUI_SPLIT_16**: These function blocks offer the same functionality with 4 or 16 outputs, respectively. The choice depends on the required number of branches.
- **AUI_MERGE_***: A merger combines multiple AUI inputs into one output – the opposite function. A splitter, on the other hand, distributes the output.
- **Generic Splitters of Other Adapter Types (e.g., Data-Based)**: Unlike data-event splitters, AUI_SPLIT_9_UNGATED operates purely at the adapter level and requires no event control, thus reducing implementation effort.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AUI_SPLIT_9_UNGATED** is a simple yet effective function block for distributing multiple unidirectional AUI signals. Its generic nature allows for flexible reuse in various projects. Due to its lack of inherent logic, this building block is resource-efficient and easy to understand. It is particularly well-suited for the rapid implementation of one-to-many connections in adapter-based automation architectures.

🌐 Related topic subpages on ms-muc-docs.de

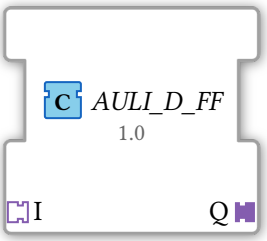
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

Numeric Index (unidirectional)

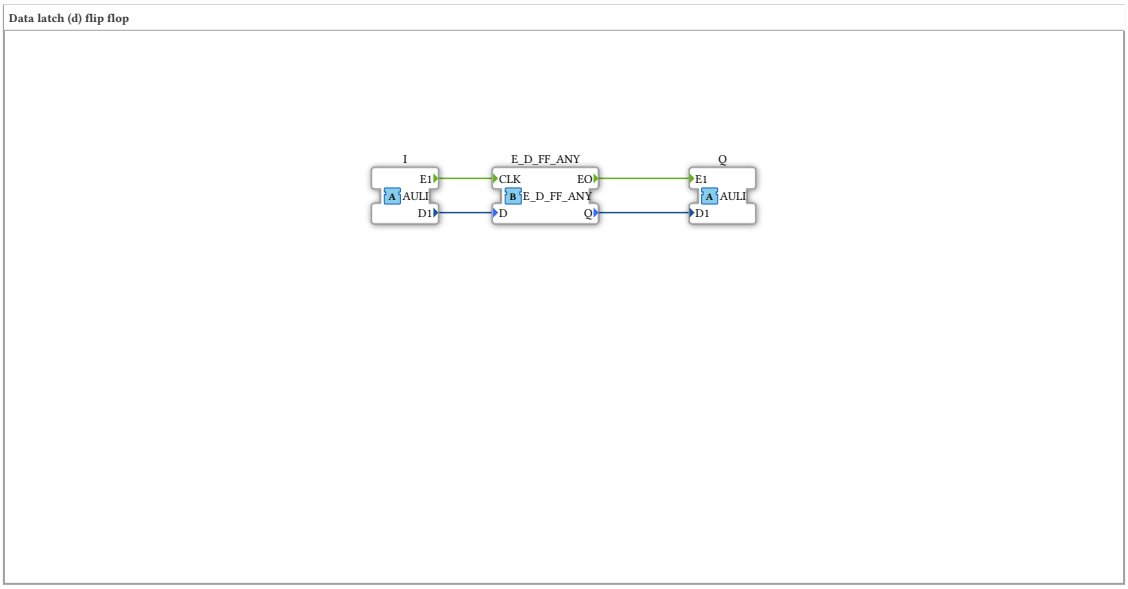
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AULI_D_FF



Value to latch – *AULI*

AULI – Latched value



• – ◦ • – ◦ • – ◦ •

Introduction

The AULI_D_FF function block implements a clock-edge-triggered D flip-flop (data latch). Upon receiving an event, it accepts the incoming data value and holds it until the next clock cycle. Input and output are handled via standardized unidirectional adapters, allowing for easy integration of the block into modular control architectures.

Interface Structure

Event Inputs

- **I.E1** – Clock event (rising edge): triggers the transfer of the data value

Event Outputs

- **Q.E1** – Output event: sent after successful transfer of the value

Data Inputs

- **I.D1** – Data value to be latched (any type, as the internal function block `E_D_FF_ANY` is used)

Data Outputs

- **Q.D1** – Latched data value (corresponds to the last value transferred from I.D1)

Adapters

The function block has two unidirectional adapters of type `adapter::types::unidirectional::AULI`:

- **Socket I** (input): receives the clock event and the data value.
- **Plug Q** (Output): Passes on the acknowledgment event and the stored data value.

Functionality

As soon as an event arrives at **I.E1** on input adapter I, the current value of **I.D1** is transferred to the internal flip-flop. Immediately afterward, the event **Q.E1** is sent at output adapter Q, and the new value is output to **Q.D1**. The stored value is retained until another clock event transfers a new value.

The function block operates as **clocked memory** and is particularly suitable for synchronizing data flows in event-driven automation systems.

Technical Features

- The function block is implemented as a **composition** and internally uses the standard function block `iec61499::events::E_D_FF_ANY`, which provides the actual flip-flop functionality.
- The use of adapters enables **uniform connectivity** to a wide variety of components that support the same adapter type.
- No further configuration parameters or initializations are required – the function block operates deterministically after the first clock edge.

State Overview

The internal state of the D flip-flop can be logical 0 or 1. Since the data value can be arbitrary, the state corresponds to the last latched value. An explicit state diagram is not required, as the behavior is fully described by the simple edge triggering.

Clock (I.E1)	Previous Q value	New Q value
Event	arbitrary	Value of I.D1
no event	unchanged	unchanged

Application Scenarios

- **Data Buffering** in production lines: A sensor value is acquired and buffered at each clock cycle.
- **State Machines**: Implementation of memory cells for step sequences where the next state depends on the current output.
- **Synchronization**: Adapting asynchronous data streams to a uniform clock in distributed control systems.
- **Signal Processing**: Holding a digital signal for the duration of a clock cycle.

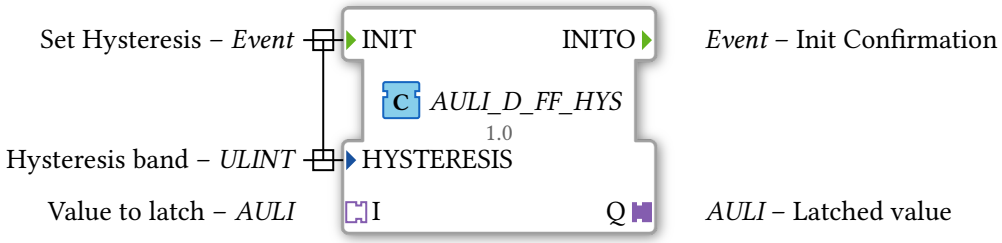
Comparison with Similar Components

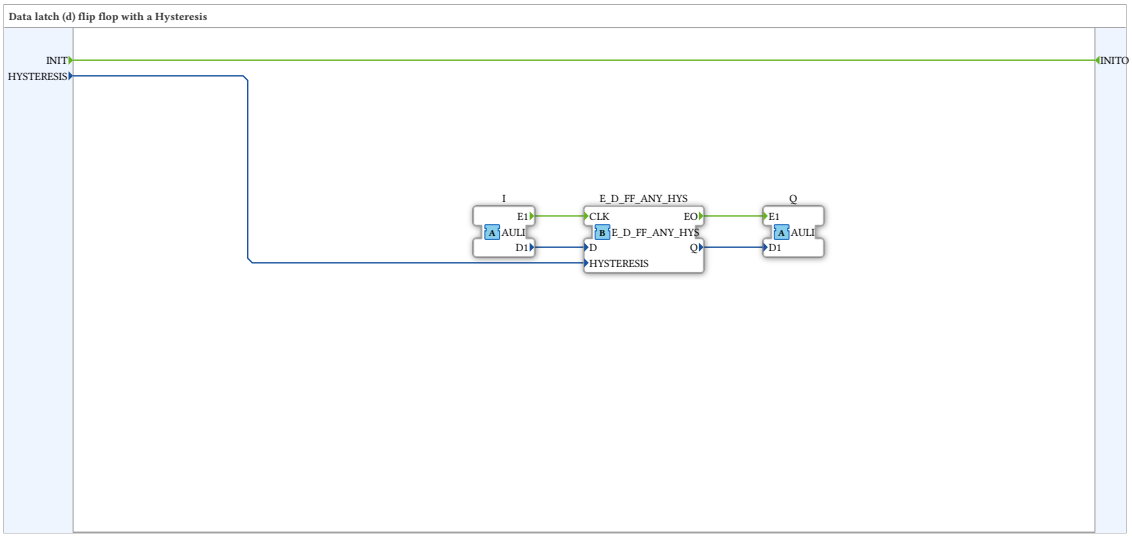
Component	Description	Difference from AULI_D_FF
SR-FF	Set-Reset Flip-Flop	Requires two separate inputs; can have illegal states.
JK-FF	Universal Flip-Flop	More complex, with toggle functionality; not needed here.
T-FF	Toggle Flip-Flop	Changes the state on each clock cycle; no data input.
AULI_D_FF	D Flip-Flop	Simple, no toggle, no illegal states, ideal for storing a value.

Conclusion

The AULI_D_FF is a compact, event-driven D flip-flop component that communicates via standardized adapters. It is ideally suited for intermediate data storage in discrete-event control environments. Its internal use of a proven standard function block ensures robust and deterministic behavior. The modular adapter interface facilitates integration into complex automation projects.

AULI_D_FF_HYS





• – ○ • – ○ • – ○ •

Introduction

The function block `AULI_D_FF_HYS` implements a data latch (D flip-flop) with adjustable hysteresis. It encapsulates the logic of such a flip-flop in an adapter-based interface and allows the transmission of data and events via unidirectional adapters of type `AULI`.

Interface Structure

Event Inputs

- `INIT` (EInit): Sets the hysteresis value and initializes the function block.

Event Outputs

- `INITO` (EInit): Confirms the successful initialization process (passed-through INIT).

Data Inputs

- `HYSTERESIS` (ULINT): Sets the hysteresis width. The value is taken during the INIT event.

Data Outputs

No direct data outputs are available; the output data is provided via the `Q` adapter.

Adapters

- **Socket `I`** (Type `adapter::types::unidirectional::AULI`): Receives the value to be latched and the associated clock signal.
- **Plug `Q`** (Type `adapter::types::unidirectional::AULI`): Passes on the latched value and the output event.

Functionality

The module internally uses the sub-module `E_D_FF_ANY_HYS`. The clock signal (`I.E1`) is fed to the CLK input and the data value (`I.D1`) to the D input of the sub-module via socket `I`. The data input `HYSTERESIS` is directly connected to the corresponding input of the sub-module. The output `Q` of the sub-module is

transferred to the data output of the plug `Q` (`Q.D1`), and the event `E0` of the sub-module is mapped to the output event of the plug `Q` (`Q.E1`).

The INIT event is passed through unchanged as INITO. The actual hysteresis initialization is performed by the sub-module upon receiving INIT, adopting the current value of `HYSTERESIS`.

The hysteresis prevents the output from switching immediately in response to small fluctuations in the input signal. Only when the input value exceeds or falls below the latched value by more than the hysteresis band is a new value adopted.

Technical Features

- The function block is defined as a pure interaction of adapters and a sub-function block (no dedicated ECC or SIFB logic).
- The use of the adapter type `AULI` (unidirectional) allows for a clear separation of data and events in a standardized interface.
- Hysteresis is set only during the INIT event; thereafter, the value remains constant for the duration of operation.
- The function block is available under the Eclipse Public License 2.0 and is defined in the IEC 61499 Annex A standard structure.

State Overview

The function block does not have its own state machine. The entire state logic is implemented in the sub-function block `E_D_FF_ANY_HYS`, which presumably implements two states (setting the latch, checking hysteresis).

Application Scenarios

- **Debouncing of Sensors or Switches**: A noisy digital signal is only accepted as a stable value if it exceeds the hysteresis band.
- **Signal Smoothing**: Small fluctuations in a measured value are suppressed, while significant changes are immediately passed through.
- **Data Backup in Asynchronous Systems**: The function block serves as a reliable buffer between a signal source and a subsequent processing unit.

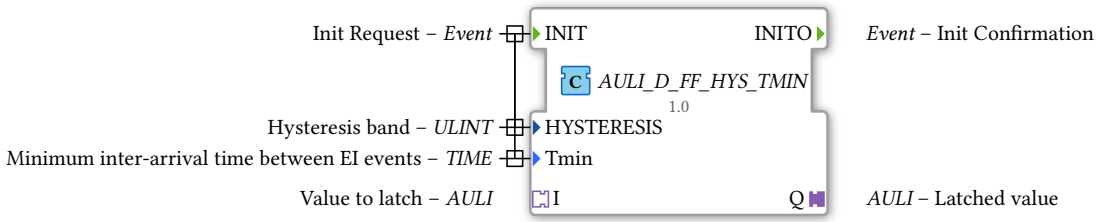
Comparison with Similar Function Blocks

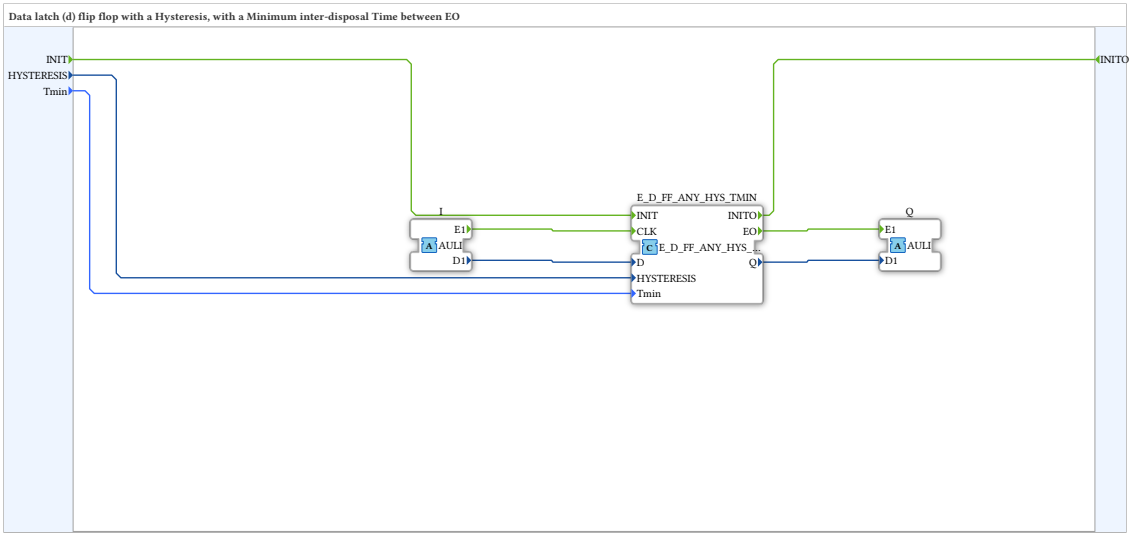
- A simple D flip-flop (e.g., `E_D_FF`) accepts the input value on every rising edge without hysteresis. The `AULI_D_FF_HYS` extends this functionality with a configurable switching threshold.
- Compared to a Schmitt trigger IC, the focus here is on the latching behavior with added hysteresis, not on level conversion.

Conclusion

The `AULI_D_FF_HYS` is a compact, adapter-based functional IC for reliable signal conditioning with hysteresis. By delegating the logic to a specialized sub-IC, the design remains maintainable and expandable. It is particularly suitable for applications where noise or unstable input signals need to be filtered without having to forgo a simple flip-flop.

AULI_D_FF_HYS_TMIN





• – ◦ • – ◦ • – ◦ •

The function block **AULI_D_FF_HYS_TMIN** implements a clock-edge-triggered D flip-flop (data latch) with hysteresis and a minimum dwell time between successive events. It is typically used to smooth noisy or fluctuating input signals and suppress unwanted rapid switching. The block communicates via standardized unidirectional adapters.

Event	Type	Comment
`INIT`	`EInit`	Initialization request. Sets the parameters hysteresis and minimum time.
Event	Type	Comment
`_____`	`__`	`____`
`INITO`	`EInit`	Confirmation of successful initialization.
Variable	Type	Comment
`_____`	`__`	`____`
`HYSTERESIS`	`ULINT`	Hysteresis band as an unsigned integer. The output only changes its value when the input exceeds the band.
`Tmin`	`TIME`	Minimum time between two triggered events at the output. Prevents excessively fast switching.

No direct data outputs. The latched information is provided via the `Q` adapter.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Adapter	Direction	Type	Comment

`I` (Socket)	**Input**	`adapter::types::unidirectional::AULI`	Receiving adapter that provides the signal to be latched. Contains the event and data signals `E1` and `D1`.
`Q` (Plug)	**Output**	`adapter::types::unidirectional::AULI`	Output adapter that provides the currently held value. Signal `E1` indicates a value change, `D1` contains the value.

Functionality

The function block internally uses the function block `E_D_FF_ANY_HYS_TMIN`, which implements the actual logic. The operation is as follows:

1. **Initialization:** An event `INIT` transfers the parameters `HYSTERESIS` and `Tmin` into the internal function block.
2. **Hysteresis:** The new output value is only adopted if the difference between the current input `D1` and the last valid output exceeds the hysteresis value.
3. **Minimum Time:** After each adoption (output of an event on `Q.E1`), no further adoption is permitted for the duration `Tmin`.
4. **Output:** On a valid edge, the new value is output via `Q.D1`, and an event is generated on `Q.E1`.

- **Hysteresis:** Suppresses noise or small fluctuations at the input by defining a deadband. The output only changes in the case of significant deviations.
- **Minimum Dwell Time (`Tmin`):** Enforces a pause between successive output events. This ensures compliance with minimum switching times due to mechanical or system requirements.
- **Adapter Interface:** The use of a unidirectional adapter type enables loose coupling and reusability in different environments.
- **Internal Function Block:** The actual logic is outsourced to a typed function block, which improves testability and maintainability.

The function block does not have its own explicit states, but delegates to the internal function block. However, the essential behaviors can be described as follows:

Phase	Description
Initial	After power-on, `HYSTERESIS` and `Tmin` are not yet applied. A `INIT` event must set the parameters.
Ready	After successful initialization, the function block waits for clock events at input `I.E1`.
Lock (Tmin active)	After an output event, any further takeover is blocked for the duration `Tmin`.
Takeover (on valid edge)	If the hysteresis is exceeded and no lock is present, the current input value is adopted and output.

- **Sensor Signal Debouncing:** A push button or proximity switch delivers a fluctuating signal. Hysteresis and a minimum time ensure a clean switching edge.
- **Data Logic with Signal Conditioning:** In automation technology, where a digital input value can be acquired but distorted by noise or bounce.
- **Time-Controlled Actuators:** If an actuator requires a minimum switching time (e.g., valves), `Tmin` prevents excessively fast actuation.

Function Block	Properties
Simple D-FF	No hysteresis, no delay – instantly responds to every clock edge.
D-FF with Hysteresis	Contains only the hysteresis band, no time limit between outputs.
D-FF with Tmin	Minimum dwell time only, without hysteresis – can still switch quickly in noisy environments.

AULI_D_FF_HYS_TMIN	Combines hysteresis and minimum dwell time – more robust against noise and protects downstream components from excessively fast switching.

The **AULI_D_FF_HYS_TMIN** offers a reliable combination of hysteresis and time limiting for clock-controlled signal takeovers. It is particularly suitable for industrial control systems where interference must be suppressed and actuators protected. The modular adapter interface facilitates integration into existing IEC 61499 applications.

Technical Features

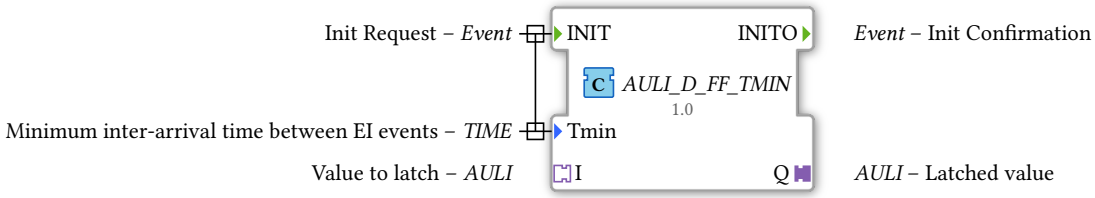
State Overview

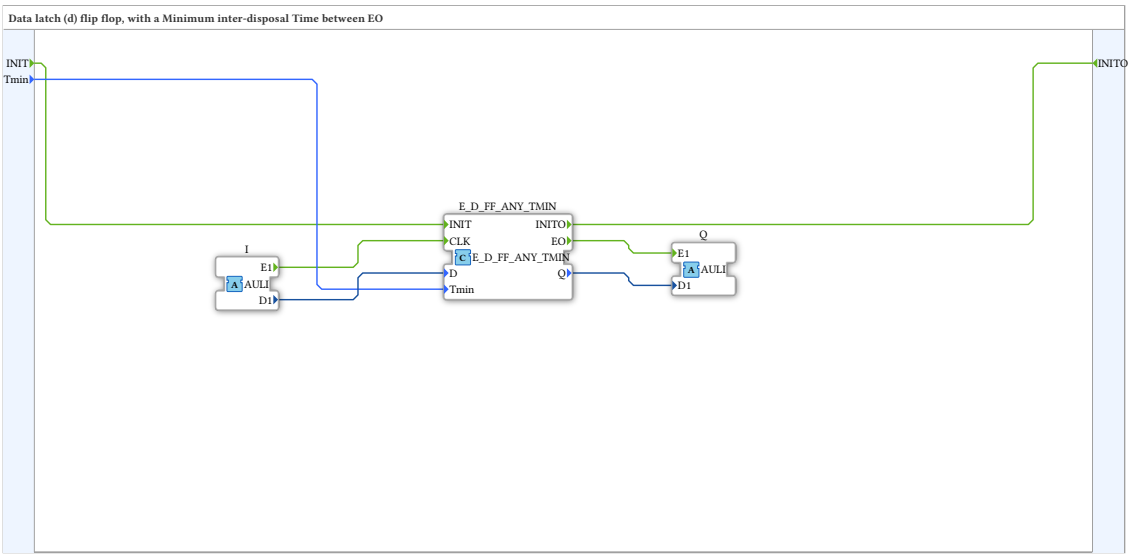
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AULI_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_D_FF_TMIN** implements a data latch (D flip-flop) with an adjustable minimum delay time between successive event outputs. It serves as a memory element for asynchronous event/data interfaces and ensures that a defined time interval elapses after a data transfer before another event is triggered. The block operates exclusively via adapter interfaces of type `AULI` (unidirectional) and is therefore particularly suitable for modular and distributed control architectures.

Interface Structure

Event Inputs

Name	Type	Comment

INIT	EInit	Initialization request; synchronized with `Tmin`

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization confirmation

Data Inputs

Name	Type	Comment
Tmin	TIME	Minimum time between two event outputs (EO) at the output adapter

Data Outputs

None

Adapters

Name	Role	Type	Comment
I	Socket	AULI	Input: value to latch (event E1, data D1)
Q	Plug	AULI	Output: latching value (event E1, data D1)

Functionality

This function block encapsulates an internal `E_D_FF_ANY_TMIN` function block from the IEC 61499 library. It operates as an edge-triggered D flip-flop with the following sequence:

1. **Initialization** – An event at the `INIT` input resets the internal state and assigns the parameter `Tmin`. The output `INITO` acknowledges the initialization.
2. **Event-driven data transfer** –

- An event at the adapter input `I.E1` (as a clock signal) transfers the current data value `I.D1` to the internal flip-flop.
- Simultaneously, the minimum time `Tmin` is started.
- After `Tmin` has elapsed, an event is triggered at the adapter output `Q.E1`, and the transferred data value `Q.D1` is output.

1. **Lockout time** – While `Tmin` is still active, a subsequent input event is ignored or buffered (depending on the internal implementation). Only after the specified time has elapsed can the next value be processed.

Technical Features

- **Unidirectional Adapters** – The function block exclusively uses the standardized adapter type `AULI` for events and data, enabling easy integration with other AULI-compatible function blocks.
- **Minimum Time (`Tmin`)** – Enforces a minimum pause between two consecutive output events. This prevents data loss or overloads in downstream components with limited processing speed.
- **No Dedicated Data Outputs** – Output data is provided exclusively via the plugin adapter `Q`, increasing modularity.

State Overview

The internal function block `E_D_FF_ANY_TMIN` can be simplified to three states:

- **IDLE** – Waiting for a clock event at input `I.E1`.
- **LATCHED** – Data has been transferred and the `Tmin` timer is running. No further transfer is possible.
- **OUTPUT** – `Tmin` has expired; the output event is triggered and the block returns to the IDLE state.

Initialization (`INIT`) resets the function block to the IDLE state.

Application Scenarios

- **Data synchronization** in time-critical control loops where a sensor value may only be sampled at a minimum interval.
- **Buffering** of asynchronous events to smooth out peak loads in the data stream.
- **Cascading** of flip-flops with different minimum times to implement time-staggered signal processing.
- **Interface between fast and slow components** (e.g., between fast logic and a slow actuator).

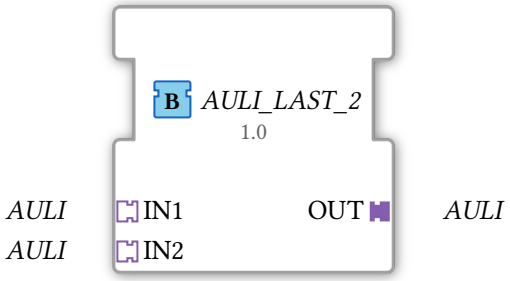
Comparison with similar components

- **E_D_FF** – Simple D flip-flop without timing; takes data immediately on every clock edge. The `AULI_D_FF_TMIN` extends this with an adjustable minimum time.
- **E_D_FF_TMIN** – Same internal core, but with direct event/data interfaces instead of adapters. The adapter version `AULI_D_FF_TMIN` is designed for systems that rely on standardized, reusable interfaces.
- **E_D_FF_ANY_TMIN** (directly used) – Offers the same functionality but is designed for arbitrary data types. `AULI_D_FF_TMIN` is specifically tailored for the AULI adapter.

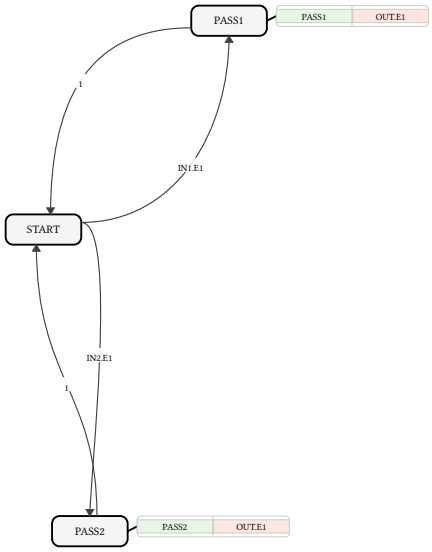
Conclusion

The `AULI_D_FF_TMIN` is a specialized D flip-flop device for event/data adapter interfaces. Its integrated minimum time (`Tmin`) ensures reliable and predictable data transfer in time-critical environments. Its modular design and compliance with IEC 61499 Annex A make it a robust component for industrial control applications.

AULI_LAST_2



ECC: AULI_LAST_2



Introduction

The AULI_LAST_2 function block merges two same-type ULINT adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: ULINT adapter (unidirectional) - first signal source
- **IN2**: ULINT adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: ULINT adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

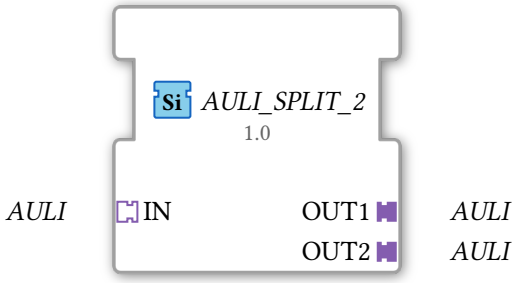
Application Scenarios

- Merging two equally-valid ULINT signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same ULINT value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AULI_LAST_2 is the inverse of `AULI_SPLIT_2`: AULI_SPLIT_2 distributes 1 input to 2 outputs, AULI_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

AULI_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_2** is used to distribute an incoming AULI adapter (unidirectional) to two identical outputs. It is implemented as a generic function block and enables simple signal multiplication in IEC 61499-based systems.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapter

Direction	Name	Type	Description
Socket (Input)	IN	AULI (unidirectional)	Incoming AULI data stream
Plug (Output)	OUT1	AULI (unidirectional)	First output, identical copy of IN
Plug (Output)	OUT2	AULI (unidirectional)	Second output, identical copy of IN

Functionality
The module forwards the AULI data received via the **IN** socket unchanged to both outputs, **OUT1** and **OUT2**. No data manipulation, buffering, or protocol conversion takes place. The distribution is passive and instantaneous.

Technical Features
- **Generic Type**: The function block is declared as `'GEN_AULI_SPLIT'` via the attribute `eclipse4diac::core::GenericClassName`, enabling flexible reuse in various engineering environments.
- **No State Management**: The function block has no internal state machine (ECC) and no memory behavior. It is completely passive.
- **Pure Adapter Interface**: Only adapters (`Plugs` and `Sockets`) are used; there are no data or event inputs.

State Overview
Since the function block contains no internal logic or state machine, there is no state description. Its behavior is limited to the constant forwarding of the incoming signal.

Application Scenarios
- **Signal Distribution**: Splitting an AULI data stream to two subsequent function blocks, e.g., for parallel processing or control.
- **Redundant Transmission**: Providing a second, identical data path for monitoring or security purposes.
- **Point-to-Multipoint Communication**: Enabling simple 1-to-2 wiring in adapter-based architectures.

Comparison with Similar Components

Component	Function
AULI_SPLIT_2	Split to two outputs (identical to IN)
AULI_SPLIT_N	Generalized variant with a configurable number of outputs
AULI_MERGE	Merging multiple inputs into one output

While **AULI_SPLIT_2** performs a fixed 1:2 split, generic splitters allow for a flexible number of outputs. Mergers like **AULI_MERGE** accomplish the opposite.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AULI_SPLIT_2** is a minimal and efficient component for multiplying signals from unidirectional AULI adapters. Due to its passive, stateless nature, it is ideally suited for real-time applications where copies of a data stream are needed without additional latency or logic. Its generic implementation facilitates its use in various development tools and libraries.

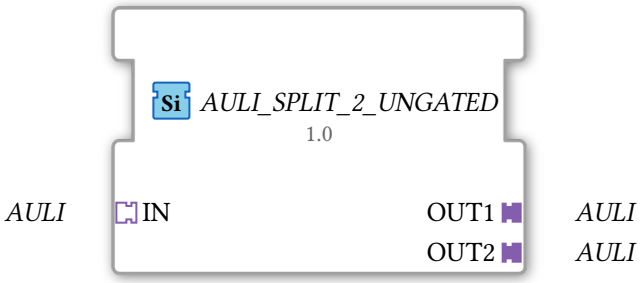
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AULI_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_2_UNGATED** is used to distribute an incoming AULI adapter (unidirectional) to two identical outputs. It is implemented as a generic function block and enables simple signal multiplication in IEC 61499-based systems.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Name	Type	Description
Socket (Input)	IN	AULI (unidirectional)	Incoming AULI data stream
Plug (Output)	OUT1	AULI (unidirectional)	First output, identical copy of IN
Plug (Output)	OUT2	AULI (unidirectional)	Second output, identical copy of IN

Functionality

The module forwards the AULI data received via the **IN** socket unchanged to both outputs, **OUT1** and **OUT2**. No data manipulation, buffering, or protocol conversion takes place. The distribution is passive and instantaneous.

Technical Features

- **Generic Type**: The function block is declared as `'GEN_AULI_SPLIT'` via the attribute `eclipse4diac::core::GenericClassName`, enabling flexible reuse in various engineering environments.
- **No State Management**: The function block has no internal state machine (ECC) and no memory behavior. It is completely passive.
- **Pure Adapter Interface**: Only adapters (`Plugs` and `Sockets`) are used; there are no data or event inputs.

State Overview

Since the function block contains no internal logic or state machine, there is no state description. Its behavior is limited to the constant forwarding of the incoming signal.

Application Scenarios

- **Signal Distribution**: Splitting an AULI data stream to two subsequent function blocks, e.g., for parallel processing or control.
- **Redundant Transmission**: Providing a second, identical data path for monitoring or security purposes.
- **Point-to-Multipoint Communication**: Enabling simple 1-to-2 wiring in adapter-based architectures.

Comparison with Similar Components

Component	Function
AULI_SPLIT_2_UNGATED	Split to two outputs (identical to IN)
AULI_SPLIT_N	Generalized variant with a configurable number of outputs
AULI_MERGE	Merging multiple inputs into one output

While **AULI_SPLIT_2_UNGATED** performs a fixed 1:2 split, generic splitters allow for a flexible number of outputs. Mergers like **AULI_MERGE** accomplish the opposite.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

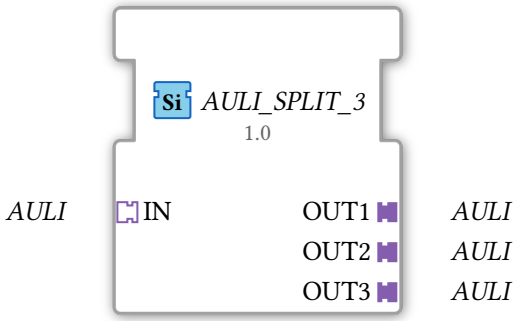
The **AULI_SPLIT_2_UNGATED** is a minimal and efficient component for multiplying signals from unidirectional AULI adapters. Due to its passive, stateless nature, it is ideally suited for real-time applications where copies of a data stream are needed without additional latency or logic. Its generic implementation facilitates its use in various development tools and libraries.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AULI_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_3** is used to distribute a single AULI adapter signal to three identical outputs. It acts as a simple splitter for unidirectional AULI connections and enables the broadcast of a signal to multiple downstream components. The block is designed generically and can be instantiated for various AULI types.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Name	Type	Description
IN	AULI (unidirectional)	Input adapter – receives the signal to be distributed.
OUT1	AULI (unidirectional)	First output adapter.
OUT2	AULI (unidirectional)	Second output adapter.
OUT3	AULI (unidirectional)	Third output adapter.

Functionality

The function block forwards all AULI information (both events and associated data) arriving at the input adapter **IN** directly and without delay to all three output adapters **OUT1**, **OUT2**, and **OUT3**. No filtering, transformation, or buffering takes place. The outputs are identical and operate in parallel, so the input signal is sent to three independent receivers simultaneously.

Technical Features

- **Generic Type:** The function block is declared as a generic function block (GenericClassName `'GEN_AULI_SPLIT'`). It can be used with various specific AULI adapter types, provided they conform to the unidirectional AULI interface.
- **Latency:** Due to the pure forwarding without internal logic, latency is minimal and limited to the runtime of the runtime system.
- **Copyright:** This function block is subject to the Eclipse Public License 2.0 (SPDX License Identifier: EPL-2.0).

State Overview

The function block does not have an internal state machine. Its functionality is purely combinational and continuously active – as soon as a signal is present at the input, it is passed on to all outputs.

Application Scenarios

- **Signal Distribution:** An AULI signal (e.g., control commands, measured values) is to be sent to several independent actuators or subsystems.
- **Test and Simulation Setups:** A test signal is recorded or sent from a source and distributed in parallel to several monitoring or analysis blocks.
- **Redundant Processing:** The same input value is required simultaneously by several parallel logic circuits.

Comparison with Similar Function Blocks

Unlike an **AULI_MERGE** (which combines multiple inputs into one output) or an **AULI_SELECT** (which selectively passes through one of several inputs), **AULI_SPLIT_3** offers a pure fan-out function. There is no decision mechanism, no prioritization, and no data modification. Similar splitters with a different number of outputs (e.g., SPLIT_2) differ only in the number of outputs.

Change Detection

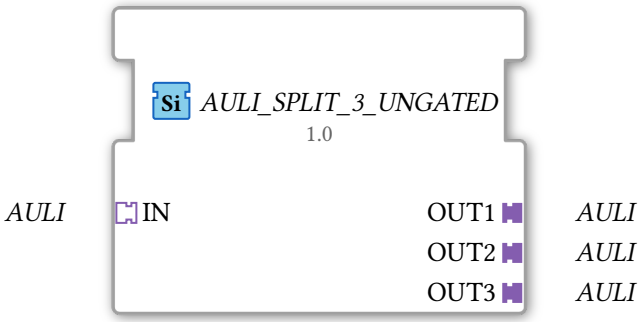
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AULI_SPLIT_3 is a simple yet essential function block for multiplying unidirectional AULI signals. Its generic design and delay-free transmission make it particularly suitable for broadcast scenarios in automation technology, where a signal must be transmitted to multiple devices simultaneously.

AULI_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_3_UNGATED** is used to distribute a single AULI adapter signal to three identical outputs. It acts as a simple splitter for unidirectional AULI connections and enables the broadcast of a signal to multiple downstream components. The block is designed generically and can be instantiated for various AULI types.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapter

Name	Type	Description
IN	AULI (unidirectional)	Input adapter – receives the signal to be distributed.
OUT1	AULI (unidirectional)	First output adapter.
OUT2	AULI (unidirectional)	Second output adapter.
OUT3	AULI (unidirectional)	Third output adapter.

Functionality
The function block forwards all AULI information (both events and associated data) arriving at the input adapter **IN** directly and without delay to all three output adapters **OUT1**, **OUT2**, and **OUT3**. No filtering, transformation, or buffering takes place. The outputs are identical and operate in parallel, so the input signal is sent to three independent receivers simultaneously.

Technical Features
- **Generic Type:** The function block is declared as a generic function block (GenericClassName `'GEN_AULI_SPLIT'`). It can be used with various specific AULI adapter types, provided they conform to the unidirectional AULI interface.
- **Latency:** Due to the pure forwarding without internal logic, latency is minimal and limited to the runtime of the runtime system.
- **Copyright:** This function block is subject to the Eclipse Public License 2.0 (SPDX License Identifier: EPL-2.0).

State Overview
The function block does not have an internal state machine. Its functionality is purely combinational and continuously active – as soon as a signal is present at the input, it is passed on to all outputs.

Application Scenarios
- **Signal Distribution:** An AULI signal (e.g., control commands, measured values) is to be sent to several independent actuators or subsystems.
- **Test and Simulation Setups:** A test signal is recorded or sent from a source and distributed in parallel to several monitoring or analysis blocks.
- **Redundant Processing:** The same input value is required simultaneously by several parallel logic circuits.

Comparison with Similar Function Blocks
Unlike an **AULI_MERGE** (which combines multiple inputs into one output) or an **AULI_SELECT** (which selectively passes through one of several inputs), **AULI_SPLIT_3_UNGATED** offers a pure fan-out function. There is no decision mechanism, no prioritization, and no data modification. Similar splitters with a different number of outputs (e.g., SPLIT_2) differ only in the number of outputs.

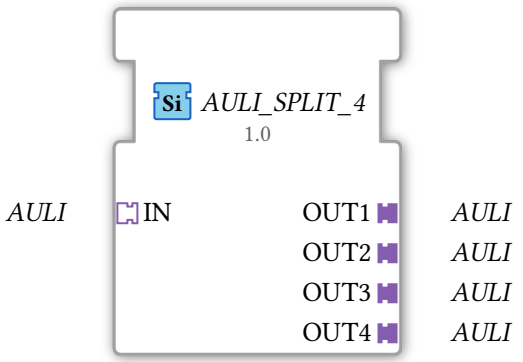
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AULI_SPLIT_3_UNGATED is a simple yet essential function block for multiplying unidirectional AULI signals. Its generic design and delay-free transmission make it particularly suitable for broadcast scenarios in automation technology, where a signal must be transmitted to multiple devices simultaneously.

AULI_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_4** is used to split an incoming **AULI** adapter into four separate, identical outputs. It is designed as a generic function block (Generic FB) and is distributed under the Eclipse Public License 2.0. It was developed for use in automation technology, particularly within the HR Agrartechnik GmbH environment.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AULI`	`IN`	Socket	Input adapter for the AULI signal to be distributed
`adapter::types::unidirectional::AULI`	`OUT1`	Plug	First output adapter (identical to the input)
`adapter::types::unidirectional::AULI`	`OUT2`	Plug	Second output adapter
`adapter::types::unidirectional::AULI`	`OUT3`	Plug	Third output adapter
`adapter::types::unidirectional::AULI`	`OUT4`	Plug	Fourth Output Adapter

Functionality

This function block operates as a passive distribution structure. The AULI adapter connected to `IN` is internally routed to the four outputs `OUT1` to `OUT4`. Since no active logic, processing, or buffering takes place, the data present at the input (e.g., signals, values, states) is passed on unchanged to all four outputs. The connection is unidirectional – changes to the outputs do not affect the input.

Technical Features

- **Generic Function Block**: The function block is declared as a generic function block with the generic class name `GEN_AULI_SPLIT`. This allows for flexible adaptation and reuse in various contexts (e.g., through parameterization during instantiation).
- **Unidirectional Adapters**: All adapters used are of type `adapter::types::unidirectional::AULI`, meaning they only support data flow in one direction.
- **License**: The block is licensed under the Eclipse Public License 2.0 and may be used and modified in accordance with the terms of this license.
- **No Own Events or Data**: All functionality is implemented exclusively via adapters. This reduces overhead and simplifies integration into existing 4diac architectures.

State Overview

The block does not have its own state machine (no ECC states). It behaves stateless: The outputs always reflect the current state of the input. Initialization or special state transitions are not required.

Application Scenarios

- **Signal Distribution**: An AULI-based signal (e.g., control command, measured value) is to be forwarded to multiple consumers or downstream function blocks.
- **Redundancy or Parallel Processing**: Several independent instances of a component require the same input value – the split enables simple distribution without additional logic.
- **Test and Simulation Environments**: In test setups, an AULI signal can be distributed to multiple observers or loggers.

Comparison with Similar Function Blocks

The **AULI_SPLIT_4** is a specialized split function block exclusively for the unidirectional AULI adapter type. Unlike general split function blocks (e.g., `SPLIT` for various data types), it is fixed to exactly one adapter interface, which increases type clarity and prevents misconfigurations. It differs from active distributors (e.g., `MUX` or `DEMUX`) in its passive, lossless distribution without switching logic or addressing.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

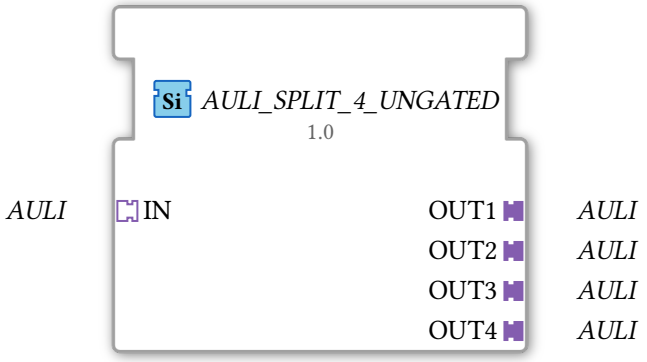
The **AULI_SPLIT_4** is a simple yet useful function block for multiplying an AULI adapter to four outputs. Its generic design and the absence of unnecessary logic allow it to integrate seamlessly into modular 4diac projects. It is particularly suitable for applications where a signal needs to be passed on to multiple receivers without requiring further processing or selection.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AULI_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_4_UNGATED** is used to split an incoming **AULI** adapter into four separate, identical outputs. It is designed as a generic function block (Generic FB) and is distributed under the Eclipse Public License 2.0. It was developed for use in automation technology, particularly within the HR Agrartechnik GmbH environment.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AULI`	`IN`	Socket	Input adapter for the AULI signal to be distributed
`adapter::types::unidirectional::AULI`	`OUT1`	Plug	First output adapter (identical to the input)
`adapter::types::unidirectional::AULI`	`OUT2`	Plug	Second output adapter
`adapter::types::unidirectional::AULI`	`OUT3`	Plug	Third output adapter
`adapter::types::unidirectional::AULI`	`OUT4`	Plug	Fourth Output Adapter

Functionality

This function block operates as a passive distribution structure. The AULI adapter connected to `IN` is internally routed to the four outputs `OUT1` to `OUT4`. Since no active logic, processing, or buffering takes place, the data present at the input (e.g., signals, values, states) is passed on unchanged to all four outputs. The connection is unidirectional – changes to the outputs do not affect the input.

Technical Features

- **Generic Function Block**: The function block is declared as a generic function block with the generic class name `GEN_AULI_SPLIT`. This allows for flexible adaptation and reuse in various contexts (e.g., through parameterization during instantiation).

- **Unidirectional Adapters**: All adapters used are of type `adapter::types::unidirectional::AULI`, meaning they only support data flow in one direction.
- **License**: The block is licensed under the Eclipse Public License 2.0 and may be used and modified in accordance with the terms of this license.
- **No Own Events or Data**: All functionality is implemented exclusively via adapters. This reduces overhead and simplifies integration into existing 4diac architectures.

State Overview

The block does not have its own state machine (no ECC states). It behaves stateless: The outputs always reflect the current state of the input. Initialization or special state transitions are not required.

Application Scenarios

- **Signal Distribution**: An AULI-based signal (e.g., control command, measured value) is to be forwarded to multiple consumers or downstream function blocks.
- **Redundancy or Parallel Processing**: Several independent instances of a component require the same input value – the split enables simple distribution without additional logic.
- **Test and Simulation Environments**: In test setups, an AULI signal can be distributed to multiple observers or loggers.

Comparison with Similar Function Blocks

The **AULI_SPLIT_4_UNGATED** is a specialized split function block exclusively for the unidirectional AULI adapter type. Unlike general split function blocks (e.g., `SPLIT` for various data types), it is fixed to exactly one adapter interface, which increases type clarity and prevents misconfigurations. It differs from active distributors (e.g., `MUX` or `DEMUX`) in its passive, lossless distribution without switching logic or addressing.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

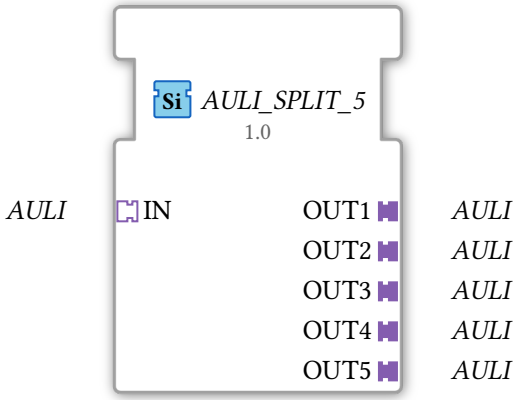
Conclusion

The **AULI_SPLIT_4_UNGATED** is a simple yet useful function block for multiplying an AULI adapter to four outputs. Its generic design and the absence of unnecessary logic allow it to integrate seamlessly into modular 4diac projects. It is particularly suitable for applications where a signal needs to be passed on to multiple receivers without requiring further processing or selection.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AULI_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_5** serves as a generic splitter for the unidirectional AULI adapter type. It generates five identical output signals from a single incoming AULI signal, which can then be distributed to separate devices. The block is implemented as a generic FB and is parameterized for the Eclipse 4diac Core Framework using the attributes `GenericClassName` and `TypeHash`.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AULI`	Incoming AULI adapter (source)
Plug	`OUT1`	`adapter::types::unidirectional::AULI`	First outgoing AULI adapter
Plug	`OUT2`	`adapter::types::unidirectional::AULI`	Second outgoing AULI adapter
Plug	`OUT3`	`adapter::types::unidirectional::AULI`	Third outgoing AULI adapter
Plug	`OUT4`	`adapter::types::unidirectional::AULI`	Fourth outgoing AULI adapter
Plug	`OUT5`	`adapter::types::unidirectional::AULI`	Fifth outgoing AULI adapter

The interface consists solely of one socket and five plugs of the same type. All adapters are unidirectional.

Functionality

The FB forwards the AULI signal arriving at socket `IN` to all five plugs (`OUT1` … `OUT5`) without delay or processing. A simple **1:5 replication** takes place. The function block has no logic, no states, and no internal buffer. Data is passed passively: As soon as the source adapter sends data, it is replicated to all outputs.

Technical Features

- **Generic Function Block**: The function block is declared as a generic type (`GenericClassName = 'GEN_AULI_SPLIT'`). The specific type used is only resolved at runtime by the Eclipse 4diac Core Framework.
- **Unidirectional Adapters**: All adapters use the type `adapter::types::unidirectional::AULI`, which allows data flow in only one direction, from the source to the destinations.
- **No Conditions or Filters**: The input signal is distributed to all outputs without quality control, conversion, or synchronization.
- **Compatibility**: The function block (FB) is designed for use in IEC 61499-2 compliant systems and requires the import of the packages `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`.

State Overview

The FB has **no internal state machine**. There are no states, no transitions, and no actions. Its functionality is limited to passive signal distribution.

Application Scenarios

- **Signal Distribution**: A sensor value or a control signal (e.g., an AULI-based data source) must be sent to multiple parallel processing units or actuators.
- **Test Environments**: In simulation or test systems, the same input signal can be applied simultaneously to multiple analysis blocks or log functions.
- **Redundant Outputs**: When multiple subsystems require the same value, but cascaded chaining is not desired.

Comparison with Similar Components

Component	Type	Outputs	Special Feature
`AULI_SPLIT_5`	Generic splitter	5	Specific to AULI adapters
`SPLIT_2` (non-standardized)	Generic splitter	2	Mostly for data types such as INT or BOOL
`F_SPLIT` (from IEC 61499 libraries)	Functional splitter	Variable	Based on ECC and events

The `AULI_SPLIT_5`, unlike event-based splitters, is **eventless** and purely adapter-based. It is therefore particularly suitable for use in data flow systems without a clock signal.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

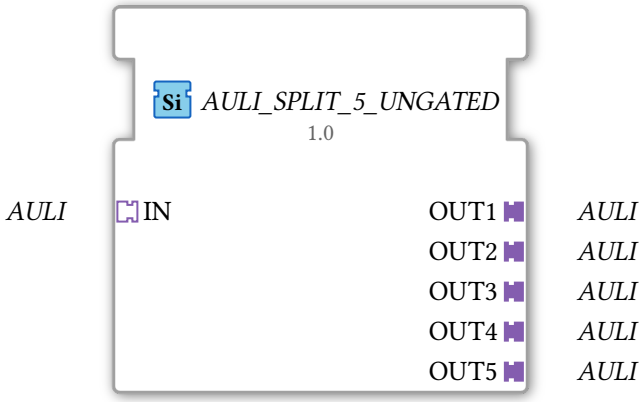
The `AULI_SPLIT_5` is a minimalist yet useful function block for multiplying a unidirectional AULI signal to five outputs. Its generic nature allows for flexible parameterization within the Eclipse 4diac framework. Due to the absence of a state machine and its simple pass-through functionality, it is extremely resource-efficient and suitable for high-speed data distribution.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AULI_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_5_UNGATED** serves as a generic splitter for the unidirectional AULI adapter type. It generates five identical output signals from a single incoming AULI signal, which can then be distributed to separate devices. The block is implemented as a generic FB and is parameterized for the Eclipse 4diac Core Framework using the attributes `GenericClassName` and `TypeHash`.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AULI`	Incoming AULI adapter (source)
Plug	`OUT1`	`adapter::types::unidirectional::AULI`	First outgoing AULI adapter
Plug	`OUT2`	`adapter::types::unidirectional::AULI`	Second outgoing AULI adapter
Plug	`OUT3`	`adapter::types::unidirectional::AULI`	Third outgoing AULI adapter
Plug	`OUT4`	`adapter::types::unidirectional::AULI`	Fourth outgoing AULI adapter
Plug	`OUT5`	`adapter::types::unidirectional::AULI`	Fifth outgoing AULI adapter

The interface consists solely of one socket and five plugs of the same type. All adapters are unidirectional.

Functionality

The FB forwards the AULI signal arriving at socket `IN` to all five plugs (`OUT1` … `OUT5`) without delay or processing. A simple **1:5 replication** takes place. The function block has no logic, no states, and no internal buffer. Data is passed passively: As soon as the source adapter sends data, it is replicated to all outputs.

Technical Features

- **Generic Function Block**: The function block is declared as a generic type (`GenericClassName = 'GEN_AULI_SPLIT'`). The specific type used is only resolved at runtime by the Eclipse 4diac Core Framework.
- **Unidirectional Adapters**: All adapters use the type `adapter::types::unidirectional::AULI`, which allows data flow in only one direction, from the source to the destinations.
- **No Conditions or Filters**: The input signal is distributed to all outputs without quality control, conversion, or synchronization.
- **Compatibility**: The function block (FB) is designed for use in IEC 61499-2 compliant systems and requires the import of the packages `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`.

State Overview

The FB has **no internal state machine**. There are no states, no transitions, and no actions. Its functionality is limited to passive signal distribution.

Application Scenarios

- **Signal Distribution**: A sensor value or a control signal (e.g., an AULI-based data source) must be sent to multiple parallel processing units or actuators.
- **Test Environments**: In simulation or test systems, the same input signal can be applied simultaneously to multiple analysis blocks or log functions.
- **Redundant Outputs**: When multiple subsystems require the same value, but cascaded chaining is not desired.

Comparison with Similar Components

Component	Type	Outputs	Special Feature
`AULI_SPLIT_5_UNGATED`	Generic splitter	5	Specific to AULI adapters
`SPLIT_2` (non-standardized)	Generic splitter	2	Mostly for data types such as INT or BOOL
`F_SPLIT` (from IEC 61499 libraries)	Functional splitter	Variable	Based on ECC and events

The `AULI_SPLIT_5_UNGATED`, unlike event-based splitters, is **eventless** and purely adapter-based. It is therefore particularly suitable for use in data flow systems without a clock signal.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

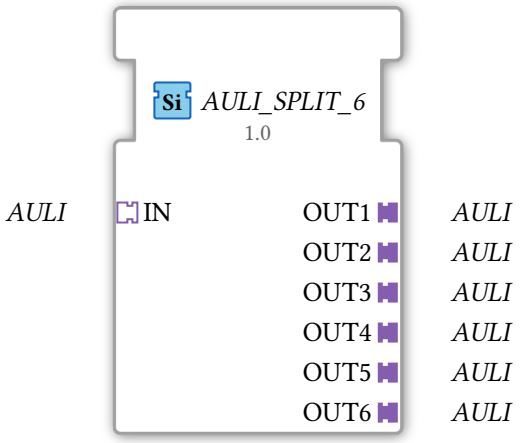
Conclusion

The `AULI_SPLIT_5_UNGATED` is a minimalist yet useful function block for multiplying a unidirectional AULI signal to five outputs. Its generic nature allows for flexible parameterization within the Eclipse 4diac framework. Due to the absence of a state machine and its simple pass-through functionality, it is extremely resource-efficient and suitable for high-speed data distribution.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AULI_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_6** is used to distribute a single AULI adapter input (unidirectional) to six identical AULI outputs. It provides a generic solution for forwarding an AULI signal to multiple subsequent blocks without compromising signal integrity.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapters

Adapter	Direction	Type	Description
IN	Socket (Input)	adapter::types::unidirectional::AULI	Receives the AULI signal to be distributed.
OUT1	Plug (Output)	adapter::types::unidirectional::AULI	First identical output of the distributed signal.
OUT2	Plug (Output)	adapter::types::unidirectional::AULI	Second identical output.
OUT3	Plug (Output)	adapter::types::unidirectional::AULI	Third identical output.
OUT4	Plug (Output)	adapter::types::unidirectional::AULI	Fourth identical output.
OUT5	Plug (Output)	adapter::types::unidirectional::AULI	Fifth identical output.
OUT6	Plug (Output)	adapter::types::unidirectional::AULI	Sixth identical output.

Functionality
This component functions as a pure signal distributor (splitter). An AULI signal applied to **IN** is duplicated to all six output adapters (**OUT1** to **OUT6**) without delay or data loss. No active processing or filtering takes place – data transmission occurs exclusively via the unidirectional AULI adapters.

Since there are no events or data inputs/outputs, communication is based entirely on the adapter interface. The component is event-free and is activated solely by connecting the adapters.

Technical Features
- **Generic Type:** The function block is implemented as a generic function block (`GEN_AULI_SPLIT`), allowing it to be reused for various AULI variants.
- **Unidirectional Adapters:** The adapters used are declared as unidirectional interfaces – they only support one direction of data flow (from the socket to the plugs).
- **No Data or Event Processing:** The function block contains no internal algorithms or states; it functions solely as a passive splitter.
- **Simple Scaling:** With a fixed number of six outputs, the function block is optimized for typical applications in agricultural technology (e.g., HR agricultural technology).

State Overview
Since the function block has no event control and manages no internal states, there is no state machine. The function block is purely combinational and passes the input signals unchanged to all outputs.

Application Scenarios

- **Distributing a Sensor Signal:** In agricultural control systems, a single AULI sensor (e.g., for crop flow) can be connected to multiple downstream evaluation modules.
- **Parallel Actuator Controls:** A control command (e.g., for hydraulic valves) is to be simultaneously forwarded to six identical actuators.
- **Redundancy Setup:** In safety-critical applications, the same signal can be routed multiple times to redundant control units.

Comparison with Similar Modules

Module	Number of Outputs	Special Feature
AULI_SPLIT_6	6	Fixed splitter for AULI adapter
AULI_SPLIT_2	2	Smaller version for two outputs
AULI_SPLIT_4	4	Medium-sized version for four outputs

Compared to smaller splitters, the **AULI_SPLIT_6** offers a higher number of outputs, reducing wiring complexity in complex systems while maintaining a clean signal structure.

Change Detection

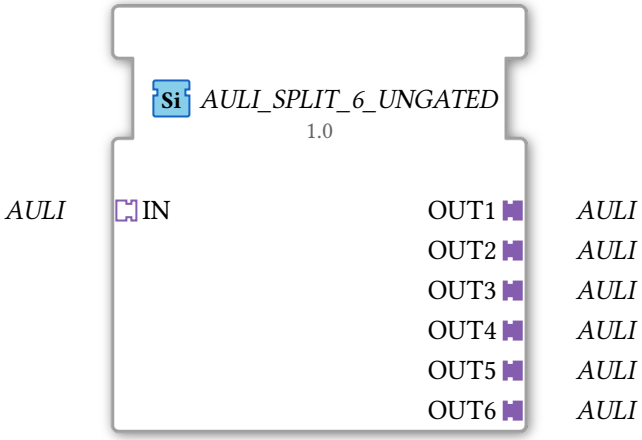
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AULI_SPLIT_6** is a simple yet effective function block for distributing an AULI adapter signal to six parallel outputs. Thanks to its generic implementation and pure adapter interface, it can be easily integrated into IEC 61499 applications in agricultural and automation technology. It is particularly suitable for scenarios where a signal is required multiple times without additional logic or state management.

AULI_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



Introduction

The function block **AULI_SPLIT_6_UNGATED** is used to distribute a single AULI adapter input (unidirectional) to six identical AULI outputs. It provides a generic solution for forwarding an AULI signal to multiple subsequent blocks without compromising signal integrity.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Adapter	Direction	Type	Description
IN	Socket (Input)	adapter::types::unidirectional::AULI	Receives the AULI signal to be distributed.
OUT1	Plug (Output)	adapter::types::unidirectional::AULI	First identical output of the distributed signal.
OUT2	Plug (Output)	adapter::types::unidirectional::AULI	Second identical output.
OUT3	Plug (Output)	adapter::types::unidirectional::AULI	Third identical output.
OUT4	Plug (Output)	adapter::types::unidirectional::AULI	Fourth identical output.
OUT5	Plug (Output)	adapter::types::unidirectional::AULI	Fifth identical output.
OUT6	Plug (Output)	adapter::types::unidirectional::AULI	Sixth identical output.

Functionality

This component functions as a pure signal distributor (splitter). An AULI signal applied to **IN** is duplicated to all six output adapters (**OUT1** to **OUT6**) without delay or data loss. No active processing or filtering takes place – data transmission occurs exclusively via the unidirectional AULI adapters.

Since there are no events or data inputs/outputs, communication is based entirely on the adapter interface. The component is event-free and is activated solely by connecting the adapters.

Technical Features

- **Generic Type:** The function block is implemented as a generic function block (`GEN_AULI_SPLIT`), allowing it to be reused for various AULI variants.
- **Unidirectional Adapters:** The adapters used are declared as unidirectional interfaces – they only support one direction of data flow (from the socket to the plugs).
- **No Data or Event Processing:** The function block contains no internal algorithms or states; it functions solely as a passive splitter.
- **Simple Scaling:** With a fixed number of six outputs, the function block is optimized for typical applications in agricultural technology (e.g., HR agricultural technology).

State Overview

Since the function block has no event control and manages no internal states, there is no state machine. The function block is purely combinational and passes the input signals unchanged to all outputs.

Application Scenarios

- **Distributing a Sensor Signal:** In agricultural control systems, a single AULI sensor (e.g., for crop flow) can be connected to multiple downstream evaluation modules.
- **Parallel Actuator Controls:** A control command (e.g., for hydraulic valves) is to be simultaneously forwarded to six identical actuators.
- **Redundancy Setup:** In safety-critical applications, the same signal can be routed multiple times to redundant control units.

Comparison with Similar Modules

Module	Number of Outputs	Special Feature
AULI_SPLIT_6_UNGATED	6	Fixed splitter for AULI adapter
AULI_SPLIT_2	2	Smaller version for two outputs
AULI_SPLIT_4	4	Medium-sized version for four outputs

Compared to smaller splitters, the **AULI_SPLIT_6_UNGATED** offers a higher number of outputs, reducing wiring complexity in complex systems while maintaining a clean signal structure.

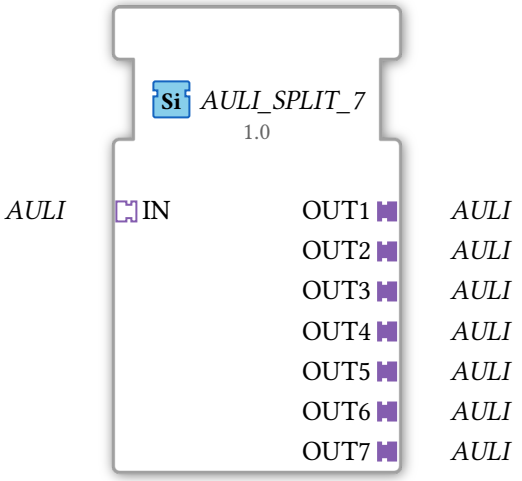
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AULI_SPLIT_6_UNGATED** is a simple yet effective function block for distributing an AULI adapter signal to six parallel outputs. Thanks to its generic implementation and pure adapter interface, it can be easily integrated into IEC 61499 applications in agricultural and automation technology. It is particularly suitable for scenarios where a signal is required multiple times without additional logic or state management.

AULI_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The **AULI_SPLIT_7** function block serves as a generic splitter that distributes an incoming AULI adapter signal (unidirectional) to seven separate AULI outputs. It is specifically designed for splitting a command or data stream (AULI protocol) and allows multiple downstream components to be supplied with the same signal simultaneously.

Interface Structure

Event Inputs
No event inputs available.

Event Outputs
No event outputs available.

Data Inputs
No data inputs available.

Data Outputs
No data outputs available.

Adapter

Direction	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::AULI	Receives the AULI signal to be split
Plug (Output)	OUT1	adapter::types::unidirectional::AULI	First output for the AULI signal
Plug (Output)	OUT2	adapter::types::unidirectional::AULI	Second output
Plug (Output)	OUT3	adapter::types::unidirectional::AULI	Third output
Plug (Output)	OUT4	adapter::types::unidirectional::AULI	Fourth output
Plug (Output)	OUT5	adapter::types::unidirectional::AULI	Fifth Output
Plug (Output)	OUT6	adapter::types::unidirectional::AULI	Sixth Output
Plug (Output)	OUT7	adapter::types::unidirectional::AULI	Seventh Output

Functionality
This module forwards the AULI signal present at socket **IN** unchanged to all seven plugs **OUT1** through **OUT7**. No signal processing, filtering, or delay takes place. Each output receives the same signal sequence simultaneously as the input. The distribution is handled purely by the circuitry and signal processing within the 4diac IDE; no calculations or state changes are required.

Technical Features
- **Generic Function Block:** The function block is declared as a generic type (`GEN_AULI_SPLIT`) and uses the attribute `eclipse4diac::core::GenericClassName` for type-safe adapter assignment.
- **Unidirectional Adapter:** Both input and output are based on the adapter type `adapter::types::unidirectional::AULI`, which supports only one direction of data flow.
- **No Event Control:** The function block does not require trigger events; signal propagation occurs automatically when the input signal changes.
- **No Internal States:** There is no state machine or memory behavior – the function block is purely combinational.

State Overview
Since there is no state machine (ECC), the function block has no internal states. The output signal follows the input signal immediately without delay or logic.

Application Scenarios
- **Parallel Supply of Multiple Actuators**: A common control signal (e.g., via an AULI adapter) is to be distributed simultaneously to multiple actuators or subsystems.
- **Signal Multicasting**: Distribution of a sensor signal or configuration message to multiple receivers in the automation system.
- **Test and Simulation Setups**: Splitting an input signal for simultaneous monitoring at different measuring points.

Comparison with Similar Function Blocks

The 4diac library contains splitter blocks for various output numbers, e.g., `AULI_SPLIT_3` or `AULI_SPLIT_5`. The `AULI_SPLIT_7` differs only in the number of outputs (7). Function blocks for splitting other adapter types (e.g., `BOOL_SPLIT`, `INT_SPLIT`) have similar logic but work with different data and adapter formats.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AULI_SPLIT_7` is a simple yet essential function block for signal distribution within the AULI adapter landscape. It allows for the clean and type-safe splitting of a unidirectional signal across up to seven paths without additional logic or delays. Thanks to its generic design, it can be flexibly used in IEC 61499-based automation projects.

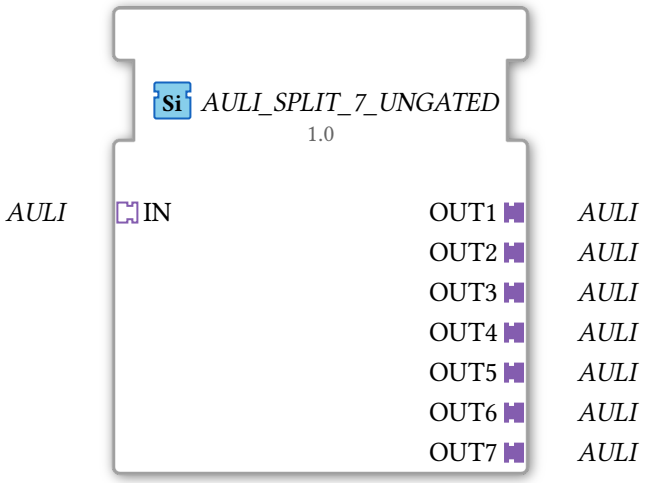
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AULI_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **AULI_SPLIT_7_UNGATED** function block serves as a generic splitter that distributes an incoming AULI adapter signal (unidirectional) to seven separate AULI outputs. It is specifically designed for splitting a command or data stream (AULI protocol) and allows multiple downstream components to be supplied with the same signal simultaneously.

Interface Structure

Event Inputs
No event inputs available.

Event Outputs
No event outputs available.

Data Inputs
No data inputs available.

Data Outputs
No data outputs available.

Adapter

Direction	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::AULI	Receives the AULI signal to be split
Plug (Output)	OUT1	adapter::types::unidirectional::AULI	First output for the AULI signal
Plug (Output)	OUT2	adapter::types::unidirectional::AULI	Second output
Plug (Output)	OUT3	adapter::types::unidirectional::AULI	Third output
Plug (Output)	OUT4	adapter::types::unidirectional::AULI	Fourth output
Plug (Output)	OUT5	adapter::types::unidirectional::AULI	Fifth Output
Plug (Output)	OUT6	adapter::types::unidirectional::AULI	Sixth Output
Plug (Output)	OUT7	adapter::types::unidirectional::AULI	Seventh Output

Functionality
This module forwards the AULI signal present at socket **IN** unchanged to all seven plugs **OUT1** through **OUT7**. No signal processing, filtering, or delay takes place. Each output receives the same signal sequence simultaneously as the input. The distribution is handled purely by the circuitry and signal processing within the 4diac IDE; no calculations or state changes are required.

Technical Features
- **Generic Function Block:** The function block is declared as a generic type (`GEN_AULI_SPLIT`) and uses the attribute `eclipse4diac::core::GenericClassName` for type-safe adapter assignment.
- **Unidirectional Adapter:** Both input and output are based on the adapter type `adapter::types::unidirectional::AULI`, which supports only one direction of data flow.
- **No Event Control:** The function block does not require trigger events; signal propagation occurs automatically when the input signal changes.
- **No Internal States:** There is no state machine or memory behavior – the function block is purely combinational.

State Overview
Since there is no state machine (ECC), the function block has no internal states. The output signal follows the input signal immediately without delay or logic.

Application Scenarios
- **Parallel Supply of Multiple Actuators**: A common control signal (e.g., via an AULI adapter) is to be distributed simultaneously to multiple actuators or subsystems.
- **Signal Multicasting**: Distribution of a sensor signal or configuration message to multiple receivers in the automation system.
- **Test and Simulation Setups**: Splitting an input signal for simultaneous monitoring at different measuring points.

Comparison with Similar Function Blocks

The 4diac library contains splitter blocks for various output numbers, e.g., `AULI_SPLIT_3` or `AULI_SPLIT_5`. The `AULI_SPLIT_7_UNGATED` differs only in the number of outputs (7). Function blocks for splitting other adapter types (e.g., `BOOL_SPLIT`, `INT_SPLIT`) have similar logic but work with different data and adapter formats.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

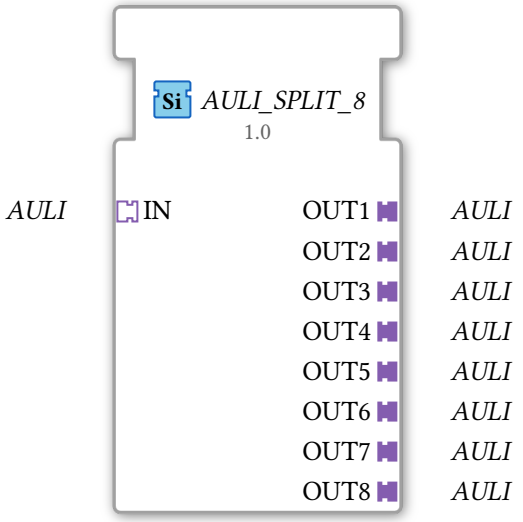
The `AULI_SPLIT_7_UNGATED` is a simple yet essential function block for signal distribution within the AULI adapter landscape. It allows for the clean and type-safe splitting of a unidirectional signal across up to seven paths without additional logic or delays. Thanks to its generic design, it can be flexibly used in IEC 61499-based automation projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AULI_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_SPLIT_8` is used to split an incoming unidirectional AULI adapter (socket `IN`) into eight identical AULI output adapters (`OUT1` to `OUT8`). It operates as a simple distributor – the incoming signal is passed on to all outputs without delay or logic modification. The block is designed generically and can be used with various AULI types (see attribute `GenericClassName`).

Interface Structure

Event Inputs

No event inputs are available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AULI`	Incoming AULI adapter, distributed to the eight outputs.
Plug (Output)	`OUT1` – `OUT8`	`adapter::types::unidirectional::AULI`	Eight parallel outputs that forward the identical signal of the input.

Functionality

The function block has no integrated control logic (no ECC) and no data processing algorithms. It connects the incoming adapter `IN` directly to all eight output adapters `OUT1` to `OUT8`. Any data exchange that occurs via the adapter `IN` (e.g., events, value changes) is passed on to all output adapters without modification. The distribution is combinatorial – there is no intermediate storage or time staggering.

Technical Features

- **Generic Structure:** The function block is declared as a generic function block with `eclipse4diac::core::GenericClassName = 'GEN_AULI_SPLIT'`. This allows it to be parameterized for different AULI adapter types (e.g., with different data types).
- **Pure Adapter Connection:** Neither event inputs/outputs nor data elements are required at the FB level – all communication takes place via the AULI adapters.
- **No State Logic:** The FB has no ECC; its behavior is entirely determined by the adapter wiring.

State Overview

The function block does not contain an internal state machine. There are no distinguishable operating modes or state changes – the behavior is purely combinatorial.

Application Scenarios

- **Signal Distribution:** An AULI sensor (e.g., temperature, pressure) should forward its signal to several independent devices (controllers, displays, data loggers) simultaneously.
- **Redundancy / Broadcasting:** A command or configuration is distributed from a central instance to up to eight actuators or subsystems.
- **Test Environments:** A simulated AULI value is sent in parallel to several FB instances for validation.

Comparison with Similar Blocks

The 4diac IDE often includes blocks such as `SPLIT_2`, `SPLIT_4`, or generic split blocks for event/data lines. `AULI_SPLIT_8` specializes in this for the AULI adapter and offers a compact 1:8 distribution without additional data type conversion. Compared to manually chaining simpler split blocks, it reduces wiring complexity and improves clarity.

- `AULI_SPLIT_8_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AULI_SPLIT_8` is a simple yet useful distribution block for unidirectional AULI adapters. It enables clean, generic signal distribution across up to eight outputs and is particularly suitable for broadcasting scenarios where multiple receivers require the same adapter value. Its generic nature makes it versatile and facilitates reuse in various projects.

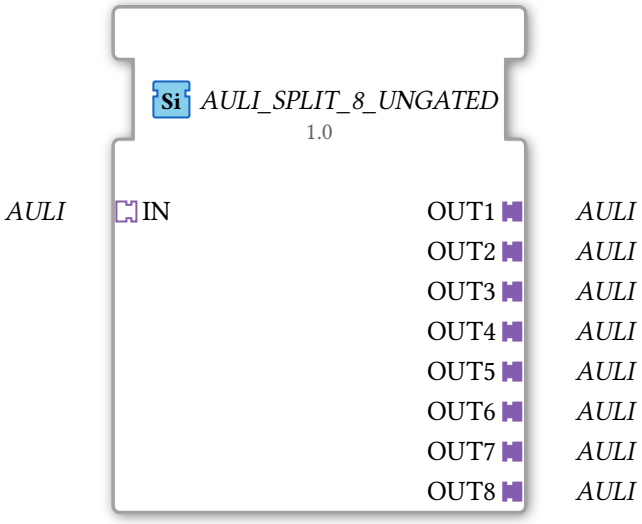
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AULI_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_SPLIT_8_UNGATED` is used to split an incoming unidirectional AULI adapter (socket `IN`) into eight identical AULI output adapters (`OUT1` to `OUT8`). It operates as a simple distributor – the incoming signal is passed on to all outputs without delay or logic modification. The block is designed generically and can be used with various AULI types (see attribute `GenericClassName`).

Interface Structure

Event Inputs

No event inputs are available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

Direction	Name	Type	Description

Socket (Input)	IN	adapter::types::unidirectional::AULI	Incoming AULI adapter, distributed to the eight outputs.
Plug (Output)	OUT1 – OUT8	adapter::types::unidirectional::AULI	Eight parallel outputs that forward the identical signal of the input.

Functionality

The function block has no integrated control logic (no ECC) and no data processing algorithms. It connects the incoming adapter `IN` directly to all eight output adapters `OUT1` to `OUT8`. Any data exchange that occurs via the adapter `IN` (e.g., events, value changes) is passed on to all output adapters without modification. The distribution is combinatorial – there is no intermediate storage or time staggering.

Technical Features

- **Generic Structure:** The function block is declared as a generic function block with `eclipse4diac::core::GenericClassName = 'GEN_AULI_SPLIT'`. This allows it to be parameterized for different AULI adapter types (e.g., with different data types).
- **Pure Adapter Connection:** Neither event inputs/outputs nor data elements are required at the FB level – all communication takes place via the AULI adapters.
- **No State Logic:** The FB has no ECC; its behavior is entirely determined by the adapter wiring.

State Overview

The function block does not contain an internal state machine. There are no distinguishable operating modes or state changes – the behavior is purely combinatorial.

Application Scenarios

- **Signal Distribution:** An AULI sensor (e.g., temperature, pressure) should forward its signal to several independent devices (controllers, displays, data loggers) simultaneously.
- **Redundancy / Broadcasting:** A command or configuration is distributed from a central instance to up to eight actuators or subsystems.
- **Test Environments:** A simulated AULI value is sent in parallel to several FB instances for validation.

Comparison with Similar Blocks

The 4diac IDE often includes blocks such as `SPLIT_2`, `SPLIT_4`, or generic split blocks for event/data lines. `AULI_SPLIT_8_UNGATED` specializes in this for the AULI adapter and offers a compact 1:8 distribution without additional data type conversion. Compared to manually chaining simpler split blocks, it reduces wiring complexity and improves clarity.

- **`AULI_SPLIT_8`**: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

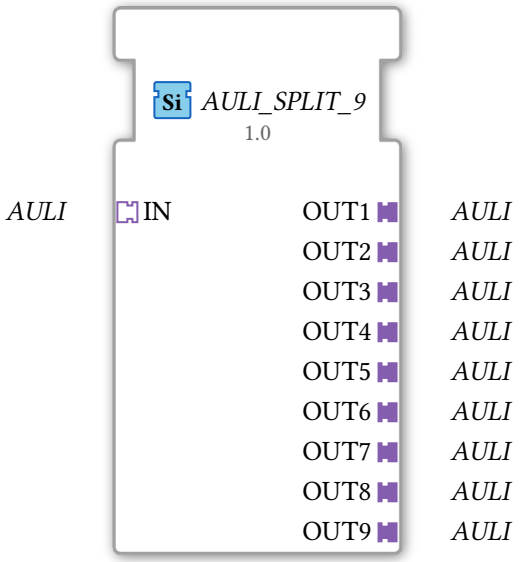
The `AULI_SPLIT_8_UNGATED` is a simple yet useful distribution block for unidirectional AULI adapters. It enables clean, generic signal distribution across up to eight outputs and is particularly suitable for broadcasting scenarios where multiple receivers require the same adapter value. Its generic nature makes it versatile and facilitates reuse in various projects.

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AULI_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_9** serves as a distributor for the unidirectional AULI adapter. It receives an incoming AULI data set via the **IN** socket and forwards it unchanged to all nine output adapters (**OUT1** to **OUT9**). The block is designed as a generic splitter and is suitable for applications where an AULI signal needs to be split among multiple devices.

Interface Structure

Event Inputs

No event inputs available.
The forwarding is purely data-driven.

Event Outputs

No event outputs available.
The block does not generate any control events.

Data Inputs

No data inputs available.
All data exchange occurs via the **IN** adapter.

Data Outputs

No data outputs are available.
Data output occurs exclusively via the **OUT1** to **OUT9** adapters.

Adapters

Type	Direction	Name	Description
Socket	Input	**IN**	Unidirectional AULI adapter – receives the data set to be distributed.
Plug	Output	**OUT1** ... **OUT9**	Nine unidirectional AULI adapters – output the identical input data set.

Functionality

The module operates as a pure 1:9 distribution stage. As soon as a data set is present at the **IN** socket, this data set is copied unchanged to all nine plugs **OUT1** to **OUT9**. Since there are no event-driven activations

or processing steps, the pass-through occurs implicitly through the runtime environment as soon as the input data record changes. The function block has no internal logic or state memory.

Technical Features

- **Generic Structure**: The function block is registered under the generic class name `GEN_AULI_SPLIT` and can be adapted depending on the runtime environment configuration.
- **Unidirectional Data Flow**: All involved adapters are of type `adapter::types::unidirectional::AULI`, which defines a clear data flow direction – from the socket IN to the plugs. No feedback is intended.
- **No State Maintenance**: The function block is stateless and requires no initialization or special control.

State Overview

The function block does not have an explicit state machine. It is always in active pass-through mode. The only "state" is the identity of the input data record being passed through.

Application Scenarios

- **Multiplication of a control/measurement signal** in automation technology, e.g., in agricultural or farm engineering (the origin of the function block indicates corresponding environments).
- **Division of an AULI-based protocol channel** across multiple parallel receivers without requiring active copying or signal amplification.
- **Simple star distribution** within a 4diac application when multiple subsequent function blocks require the same AULI data set.

Comparison with Similar Function Blocks

- **AULI_SPLIT_4 / AULI_SPLIT_8**: These variants differ only in the number of output channels. The function block described here offers a particularly high distribution density with 9 outputs.
- **Generic splitter function blocks for other adapter types**: In principle, analog splitters exist for... B. `AULI` adapter with a lower output count. All of them share stateless 1:n duplication.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AULI_SPLIT_9** is a simple yet effective distribution block for the unidirectional AULI adapter. Thanks to its generic nature and complete signal passthrough without latency or processing delay, it is ideally suited for scenarios where a single data set needs to be distributed to many receivers. The lack of event control and the pure adapter approach make it particularly lightweight and performant.

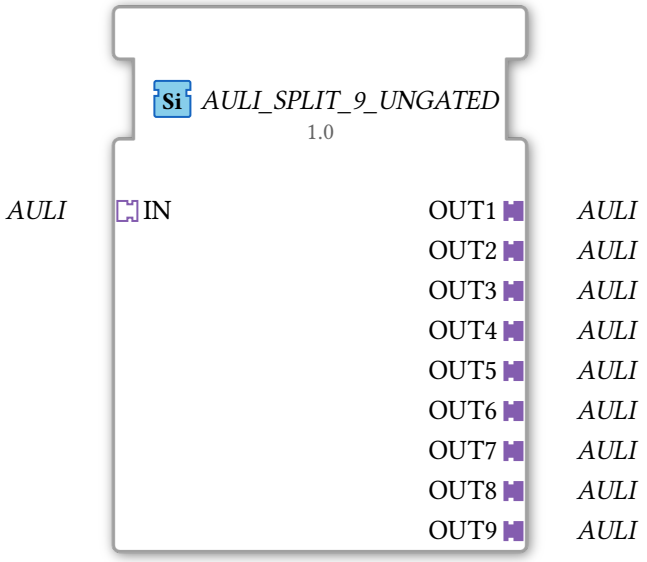
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AULI_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AULI_SPLIT_9_UNGATED** serves as a distributor for the unidirectional AULI adapter. It receives an incoming AULI data set via the **IN** socket and forwards it unchanged to all nine output adapters (**OUT1** to **OUT9**). The block is designed as a generic splitter and is suitable for applications where an AULI signal needs to be split among multiple devices.

Interface Structure

Event Inputs

No event inputs available.
The forwarding is purely data-driven.

Event Outputs

No event outputs available.
The block does not generate any control events.

Data Inputs

No data inputs available.
All data exchange occurs via the **IN** adapter.

Data Outputs

No data outputs are available.
Data output occurs exclusively via the **OUT1** to **OUT9** adapters.

Adapters

Type	Direction	Name	Description
Socket	Input	**IN**	Unidirectional AULI adapter – receives the data set to be distributed.
Plug	Output	**OUT1** ... **OUT9**	Nine unidirectional AULI adapters – output the identical input data set.

Functionality

The module operates as a pure 1:9 distribution stage. As soon as a data set is present at the **IN** socket, this data set is copied unchanged to all nine plugs **OUT1** to **OUT9**. Since there are no event-driven activations

or processing steps, the pass-through occurs implicitly through the runtime environment as soon as the input data record changes. The function block has no internal logic or state memory.

Technical Features

- **Generic Structure**: The function block is registered under the generic class name `GEN_AULI_SPLIT` and can be adapted depending on the runtime environment configuration.
- **Unidirectional Data Flow**: All involved adapters are of type `adapter::types::unidirectional::AULI`, which defines a clear data flow direction – from the socket IN to the plugs. No feedback is intended.
- **No State Maintenance**: The function block is stateless and requires no initialization or special control.

State Overview

The function block does not have an explicit state machine. It is always in active pass-through mode. The only "state" is the identity of the input data record being passed through.

Application Scenarios

- **Multiplication of a control/measurement signal** in automation technology, e.g., in agricultural or farm engineering (the origin of the function block indicates corresponding environments).
- **Division of an AULI-based protocol channel** across multiple parallel receivers without requiring active copying or signal amplification.
- **Simple star distribution** within a 4diac application when multiple subsequent function blocks require the same AULI data set.

Comparison with Similar Function Blocks

- **AULI_SPLIT_4 / AULI_SPLIT_8**: These variants differ only in the number of output channels. The function block described here offers a particularly high distribution density with 9 outputs.
- **Generic splitter function blocks for other adapter types**: In principle, analog splitters exist for... B. `AULI` adapter with a lower output count. All of them share stateless 1:n duplication.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AULI_SPLIT_9_UNGATED** is a simple yet effective distribution block for the unidirectional AULI adapter. Thanks to its generic nature and complete signal passthrough without latency or processing delay, it is ideally suited for scenarios where a single data set needs to be distributed to many receivers. The lack of event control and the pure adapter approach make it particularly lightweight and performant.
–

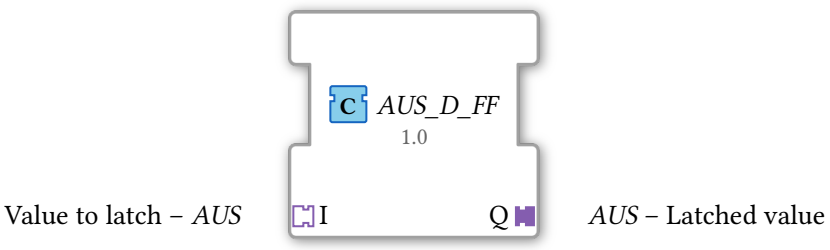
Related topic subpages on ms-muc-docs.de

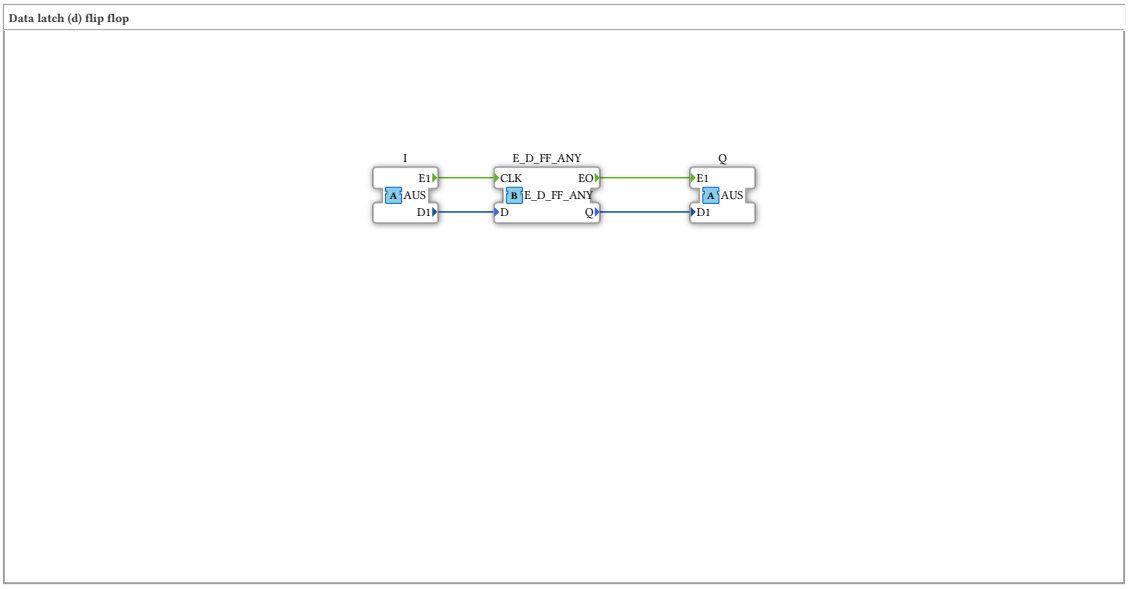
- Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AUS_D_FF





• – ○ • – ○ • – ○ •

The function block **AUS_D_FF** implements a data latch (D flip-flop) for the data type `AUS`. Upon receiving an event, it stores the incoming data value and outputs this stored value via an adapter output. Communication occurs exclusively via unidirectional adapters. No direct event inputs. Event-driven data transfer occurs via the adapter input **I** (see section Adapters).

No direct event outputs. The output of the stored value is signaled via the adapter output **Q** (see section Adapters).

No direct data inputs. The data to be stored is provided via the adapter input **I**.

No direct data outputs. The stored value is output via the adapter output **Q**.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Adapter	Direction	Type	Description
I	Socket	`adapter::types::unidirectional::AUS`	Provides the signal to be latched (event E1 and data D1).
Q	Plug	`adapter::types::unidirectional::AUS`	Outputs the latched value (event E1 and data D1).

The internally used module `E_D_FF_ANY` expects an event (inherited from **I.E1**) at its **CLK** input and the corresponding data value (from **I.D1**) at **D**. The output event **EO** is forwarded to **Q.E1**, and the stored data value **Q** to **Q.D1**.

Functionality

As soon as an event arrives at the adapter input **I** (pin **E1**), the simultaneously present data value (pin **D1**) is transferred to the internal memory. After successful transfer, an event is triggered at the adapter output **Q** (pin **E1**), and the stored value is available at **Q.D1**. The stored value is retained until a new event at **I.E1** transfers a new value.

- **Adapter-based interface:** The component uses only unidirectional adapters, enabling loose coupling and flexible interconnection.
- **No dedicated event/data pins:** All input and output is handled via the **I** and **Q** adapters.
- **Internal standard flip-flop:** The actual latch functionality is provided by the IEC 61499 component `E_D_FF_ANY`, which is specified to operate on any data type template.

The component has an implicit internal state – the stored data value (`Q` output). This only changes when an event occurs at **I.E1**. There is no explicit state machine in the FBNet; The state is managed by the internal `E_D_FF_ANY`.

- **Cache Storage:** A signal (value + event) is to be cached for later processing, e.g., in a control sequence where a sensor value is only to be acquired at a specific clock cycle.
- **Data Synchronization:** An asynchronously arriving data value is latched synchronously with an event and then used in the subsequent program flow.
- **State Maintenance:** The function block remembers the last valid value until a new value is explicitly acquired.
- **SR Flip-Flop:** Sets or resets the output via separate set/reset inputs; stores only Boolean or binary states. `AUS_D_FF`, on the other hand, stores general data values of type `AUS`.
- **Toggle Flip-Flop (T-FF):** Toggles the output on every clock cycle. `AUS_D_FF` takes the current data value and outputs it unchanged.
- **MUX-based Latch:** A multiplexer could also store a value, but requires additional reset logic. `AUS_D_FF` already encapsulates this logic.

The **AUS_D_FF** is an easy-to-use latching device that uses adapters to take a stored data-event pair and pass it on. Due to its adapter interface and the use of a standardized internal flip-flop, it is suitable for modular and reusable control designs in the IEC 61499 environment.

Technical Features

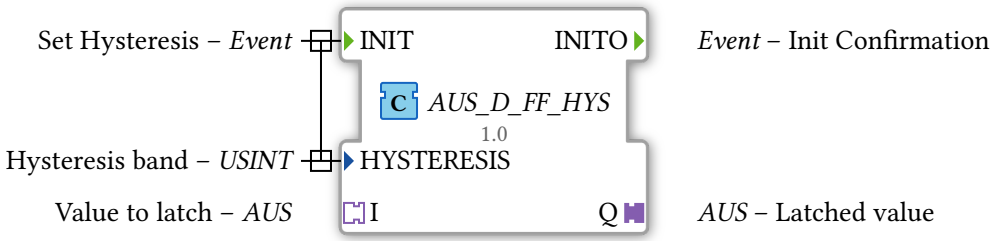
State Overview

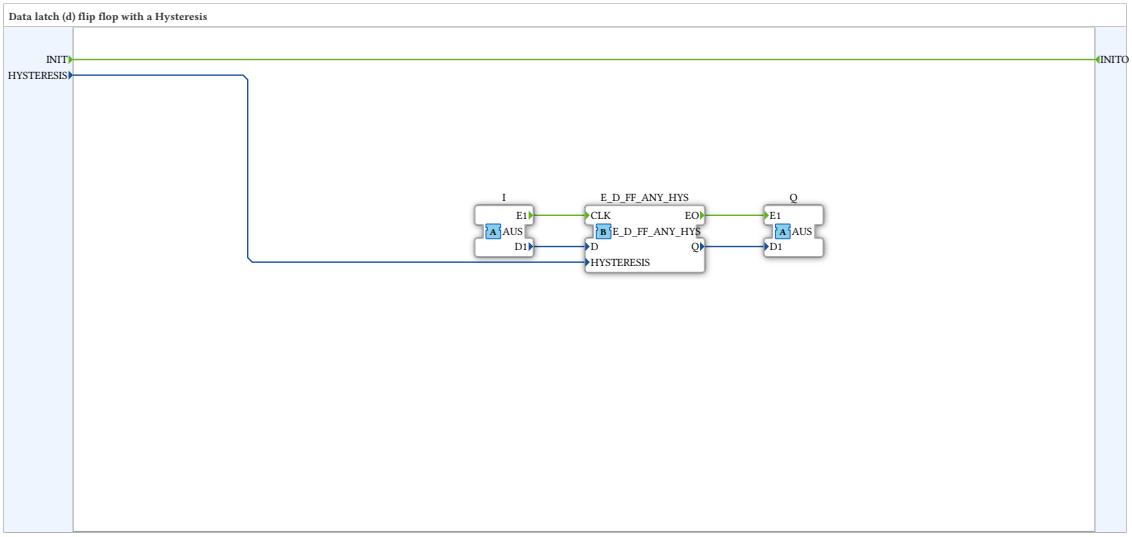
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AUS_D_FF_HYS





• – ○ • – ○ • – ○ •

Introduction

The function block `AUS_D_FF_HYS` implements a data latch (D flip-flop) with hysteresis (threshold band). It serves to receive an incoming data value on an edge-triggered basis and make it available at the output. The hysteresis ensures stable switching behavior even with noisy or fluctuating input signals. The block uses generic adapters (`AUS`) that enable unidirectional data transmission with event-driven control, making it usable with any data type.

Interface Structure

Event Inputs

Event	Type	Comment	With Parameter
`INIT`	EInit	Set Hysteresis	`HYSTERESIS`

Event Outputs

Event	Type	Comment
`INITO`	EInit	Init Confirmation

Data Inputs

Name	Type	Comment
`HYSTERESIS`	USINT	Hysteresis band

Data Outputs

The FB has no direct data outputs. The latched value is provided via the adapter output `Q`.

Adapters

Name	Type	Direction	Comment
`I`	`adapter::types::unidirectional::AUS`	Socket	Value to latch
`Q`	`adapter::types::unidirectional::AUS`	Plug	Latched value

The adapters are unidirectional and each contains an event (`E1`) and a data value (`D1`). The input value and its acceptance event are received via `I`, and the output value and an acknowledgment event are sent via `Q`.

Functionality

The function block encapsulates an internal block, `E_D_FF_ANY_HYS`, which implements the actual D flip-flop with hysteresis.

- **Initialization:** The hysteresis value (`HYSTERESIS`) is passed to the internal block via the event `INIT`. The initialization is acknowledged via the event `INITO`.

The FB encapsulates an internal block, `E_D_FF_ANY_HYS`, which implements the actual D flip-flop with hysteresis. - **Data Transfer:** An event on `I.E1` causes the internal component to take the current data value from `I.D1` and pass the latched value to `Q.D1`. Simultaneously, `Q.E1` is triggered.

- **Hysteresis Function:** The flip-flop output changes its state only when the input value exceeds an upper or lower threshold defined by the hysteresis value. This suppresses bounce or noise in the input signal (Schmitt trigger behavior).

The exact switching threshold depends on the data type used by the adapter (the `AUS` adapter supports any type). The hysteresis value (`USINT`) is internally scaled or interpreted to the corresponding numerical range.

Technical Features

- **Generic Adapter:** The use of the unidirectional adapter `AUS` allows the use of a wide variety of data types (e.g., `INT`, `REAL`, `BOOL`) without requiring the function block itself to be typed. Type conversion is handled by the internal function block.
- **Hysteresis as USINT:** Hysteresis is specified as an unsigned 8-bit value. The specific interpretation (e.g., as an absolute difference or percentage) depends on the implementation and is defined in the internal function block.
- **Event Passthrough:** The INIT event is passed directly to the INITO event. This enables a simple initialization chain in the network.
- **Event Passthrough:** - **No visible state machine:** The function block (FB) does not have its own Execution Control Chart (ECC); the logic is implemented entirely by the internal FB.

State Overview

Since the FB does not explicitly expose a state machine, its behavior is determined by the internal D flip-flop with hysteresis. This flip-flop has two stable output states:

- **State 0:** The output value corresponds to the logical “low” level (or the lower limit of the data range).
- **State 1:** The output value corresponds to the logical “high” level (or the upper limit).

State transitions occur only when the respective hysteresis thresholds are exceeded:

Transition	Condition (Input Value)
$0 \longrightarrow 1$	Input exceeds upper threshold
$1 \longrightarrow 0$	Input falls below lower threshold

The thresholds are offset by the hysteresis value relative to a mean or reference value (typically upper threshold = reference + hysteresis/2, lower threshold = reference - hysteresis/2).

Application Scenarios

- **Sensor Debouncing:** A digital or analog sensor delivers fluctuating values (e.g., due to mechanical bouncing). The function block smooths the signal and provides a stable output.
- **Threshold Switch with Reset Delay:** Monitoring of a process value that triggers a signal when an upper limit is exceeded and is only reset when a lower limit is undershot.
- **Signal Conditioning in Building Automation:** Suppression of short interference pulses from temperature, brightness, or level sensors.
- **Hysteresis in Position Controllers:** Prevents rapid switching on and off of actuators near a setpoint limit.

Comparison with Similar Function Blocks

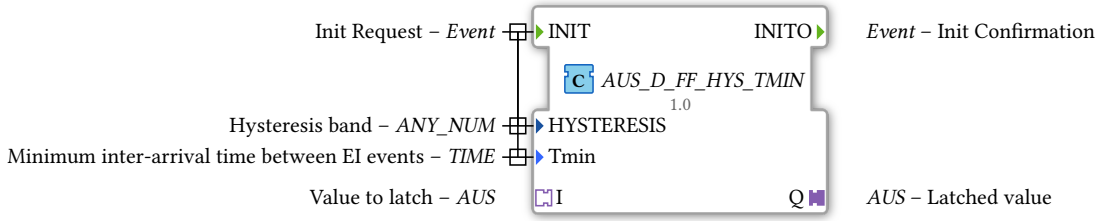
Function Block	Properties	Difference to `AUS_D_FF_HYS`

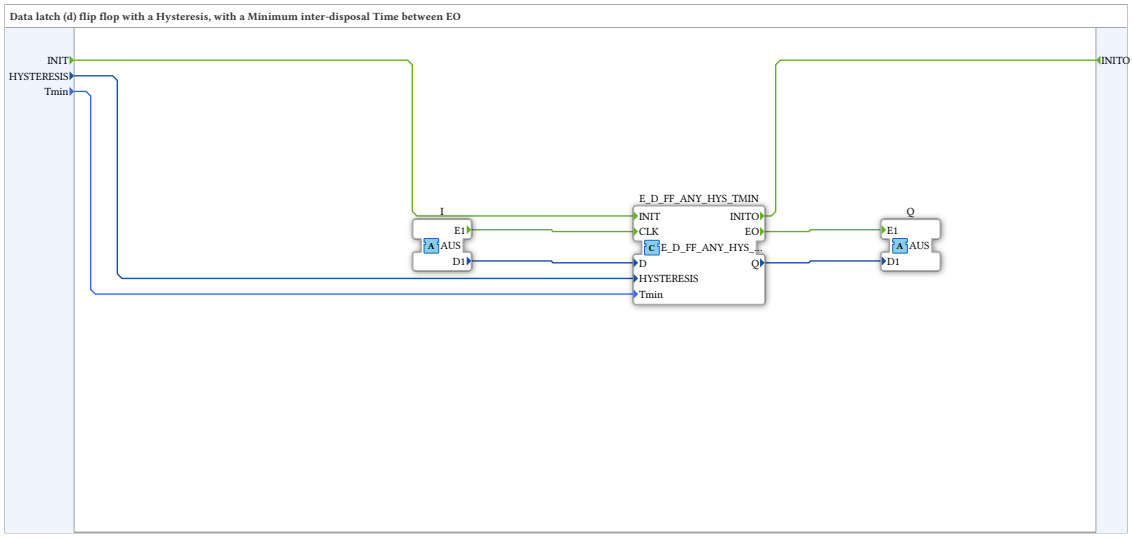
Standard D flip-flop (`E_D_FF`)	Transfers data on clock edge, no hysteresis	No noise suppression; state changes on every edge.
Schmitt trigger (`E_D_FF_ANY_HYS`)	Flip-flop with hysteresis, but usually data-type specific	`AUS_D_FF_HYS` encapsulates the generic adapter and provides a uniform interface for arbitrary types.
SR flip-flop (`E_SR_FF`)	Set/reset function without clock	No edge-triggered data transfer; Hysteresis not directly integrated.

Conclusion

AUS_D_FF_HYSis a versatile function block for applications requiring robust, edge-triggered data acquisition with hysteresis. The use of generic adapters makes it independent of the specific data type and facilitates reuse in different contexts. Particularly in automation technology, signal processing, and sensor evaluation, it offers reliable signal conditioning through noise and bounce suppression. Initialization viaINIT` and direct event pass-through simplify integration into existing control networks.

AUS_D_FF_HYS_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_D_FF_HYS_TMIN** implements a data-triggered flip-flop (D flip-flop) with hysteresis and a minimum inter-event time. It is structured as a sub-application (FB network) that delegates the actual logic to the inner block `logiBUS::signalprocessing::hysteresis::E_D_FF_ANY_HYS_TMIN`. This block serves to latch an analog or discrete value with a predefined hysteresis band and simultaneously limit the event rate (via the adapter input) to a configurable minimum interval.

Interface Structure

Event Inputs

Event	Type	With Variables	Description
INIT	EInit	HYSTERESIS, Tmin	Initialization request; sets hysteresis and minimum time.

Event Outputs

Event	Type	With Variables	Description
INITO	EInit	–	Confirmation of completed initialization.

Data Inputs

Name	Type	Description
HYSTERESIS	ANY_NUM	Hysteresis band – Threshold difference that must be exceeded/fallen below for a new value to be latched.
Tmin	TIME	Minimum time interval between two incoming events (inter-arrival time).

Data Outputs

– No direct data outputs are available. The latched value is output via the adapter plug Q.

Adapters

Adapter	Direction	Type	Description
I	Socket (In)	adapter::types::unidirectional::AUS	Receives the event to be latched and the corresponding data value.
Q	Plug (Out)	adapter::types::unidirectional::AUS	Outputs the processed event and the latched data value.

Functionality

This function block operates as a pure encapsulation of an internal flip-flop. After starting, INIT must first be called with the parameters HYSTERESIS and Tmin. Afterwards, events (via I.E1) and associated data values (via I.D1) can be received via the adapter socket I.

- Each event at input I.E1 is forwarded to the internal function block E_D_FF_ANY_HYS_TMIN.
- The data I.D1 is compared with the hysteresis band: The flip-flop is only updated if the new value lies outside the deadband defined by the hysteresis around the last latched value.
- Additionally, the time since the last valid event is monitored (Tmin). If an event occurs too soon (shorter than Tmin) after the previous one, it is ignored.
- A valid, latched event is output via the adapter plug Q as Q.E1, and the corresponding data value as Q.D1.

The internal function block handles all the logic; the outer block serves as an interface wrapper that connects the events and data accordingly.

Technical Features

- **Adapter Interface:** Communication takes place via the unidirectional adapters adapter::types::unidirectional::AUS. This enables a clear separation of event and data flow and easy reuse in different contexts.
- **Hysteresis:** Prevents constant switching due to noisy signals. The latched value only changes when the input leaves the hysteresis band.
- **Minimum Inter-Event Time (Tmin):** Debounces events and prevents bursts. Only events with a minimum interval of Tmin are accepted.

- **Compact Encapsulation:** The function block (FB) contains no independent behavior but forms a logical unit comprising initialization, event processing, and output – ideal for modular control architectures.

State Overview

Since the function block does not have its own state diagram (the logic resides within the internal FB), the state behavior results from the interaction of event control and the parameters:

1. **Initialization (`INIT` – `INITO`)**: After startup, the internal FB is configured. Only then can events be processed.
2. **Waiting for Incoming Event**: The function block is passive until an event arrives via `I.E1`.
3. **Event Check** (internal):

- Time Check: Is the interval to the last event ≥ `Tmin`?
- Hysteresis Check: Does the new value exceed the old value by more than `HYSTERESIS`?

1. **Output**: If the check is successful, `Q.E1` is triggered and the new data value is passed to `Q.D1`.
2. **Return to Waiting State**.

Application Scenarios

- **Sensor Evaluation with Noise:** An analog value (e.g., temperature, pressure) should only trigger a switching operation if it shows a significant change (hysteresis) and does not occur too rapidly (debouncing).
- **Frequency Limiting:** In communication protocols where an event must not be processed more frequently than a certain rate.
- **Safety-Critical Applications:** Preventing false triggers caused by short-duration interference pulses (minimum time between two edges).

Comparison with Similar Components

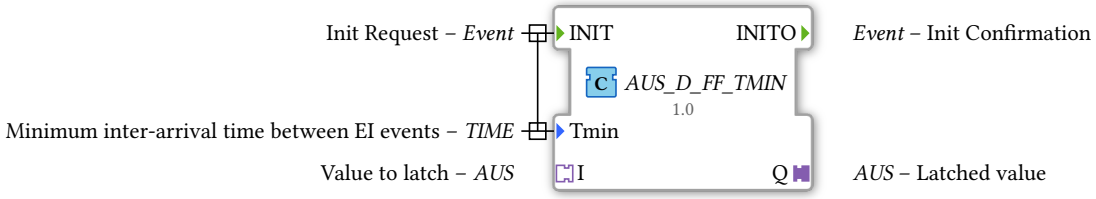
Component	Properties
Standard D Flip-Flop	Latches on every rising edge without hysteresis or time limit.
`E_D_FF_HYS` (without Tmin)	Hysteresis only, no time debouncing.
E_D_FF_TMIN` (without Hys)	Minimum time only, no hysteresis.
AUS_D_FF_HYS_TMIN`	Combines hysteresis **and** minimum time – more robust against noise and bursts.

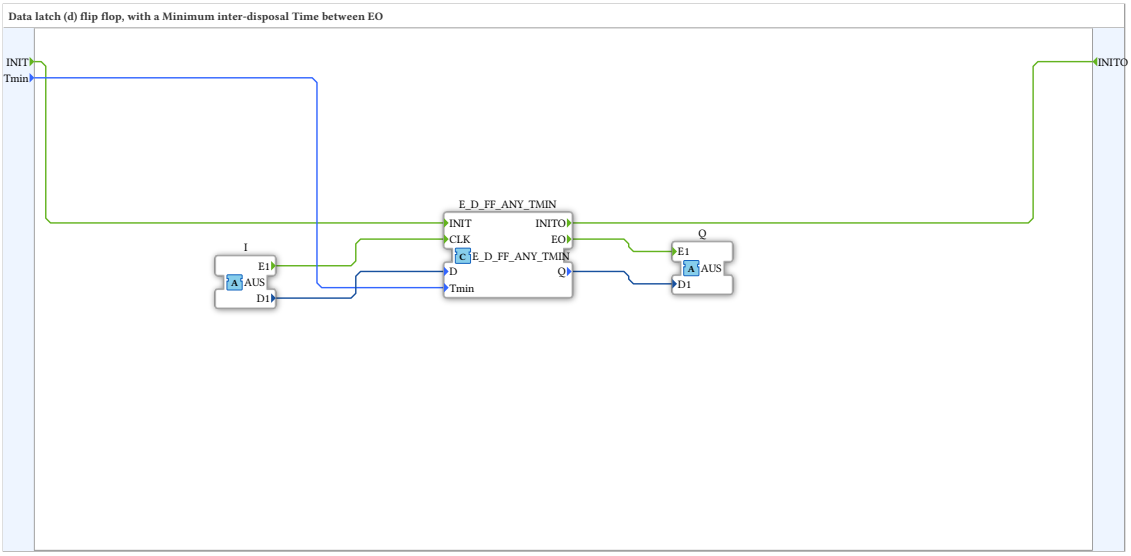
This component is therefore particularly suitable when both effects (signal tolerance and time limiting) are required simultaneously.

Conclusion

AUS_D_FF_HYS_TMINThis is a specialized D-flip-flop wrapper that offers reliable, debounced data latching through a combination of hysteresis and minimal event interval time. Its encapsulation as a sub-application with adapter interfaces makes it easy to integrate into larger control networks. Initialization via INIT` ensures a defined startup configuration. This component is an ideal solution for applications requiring robust threshold detection with noise suppression and frequency limiting.

AUS_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_D_FF_TMIN** implements a clock-controlled D-latch (data flip-flop) with a minimum inter-disposal time between two consecutive output events (EO). It is particularly suitable for applications where an incoming value (via an adapter socket) is received and may only be output again after a configurable minimum time has elapsed. Initialization is performed via the standard event input **INIT**.

Interface Structure

Event Inputs

Name	Type	Comment	With Variable
INIT	EInit	Init Request	Tmin

Event Outputs

Name	Type	Comment
INITO	EInit	Init Confirmation

Data Inputs

Name	Type	Comment
HYSTERESIS	ANY_NUM	Hysteresis band (currently not connected to the internal network)
Tmin	TIME	Minimum time interval between two EO events

Data Outputs

No direct data outputs – output data is provided exclusively via the **Adapter Plug Q**.

Adapter

Direction	Name	Type	Comment
Socket	I	adapter::types::unidirectional::OFF	Value to be transferred (contains event E1 and data D1)
Plug	Q	adapter::types::unidirectional::OFF	Transferred output value (contains event E1 and data D1)

Functionality

The function block internally contains a pre-built FB **E_D_FF_ANY_TMIN** (from the library `iec61499::events`). This function block takes the current data value from the Clock input upon each incoming event and passes it on to the output – but only if the time elapsed since the last output event is at least **Tmin**.

- During the **INIT** event, the parameter *Tmin* is passed to the internal function block.
- An event on **I.E1** (adapter socket) acts as the clock for the internal flip-flop.
- The latched data value appears on **Q.D1** (adapter plug) and is simultaneously acknowledged with an event on **Q.E1** – provided the timing condition is met.
- The **HYSTERESIS** parameter is declared but is not used in the current network. It can be used for future expansions or for manual wiring.

Technical Features

- **Minimum time between outputs**: The output of the latched value is only enabled when the time elapsed since the last **EO** is ≥ *Tmin*. This prevents an excessively high event rate at the output.
- **Adapter Interface**: Input and output are implemented as unidirectional adapters of type **OFF**, allowing the module to be easily integrated into existing adapter-based architectures.
- **Hysteresis Parameter**: Although not directly wired, the *HYSTERESIS* input is available – for example, to implement Schmitt-trigger-like behavior in later versions.

State Overview

The internal FB **E_D_FF_ANY_TMIN** has a memory for the last acquired data value. This state is updated with each clock event (I.E1). An explicit state machine is not present at the top level; the behavior results from the logic of the underlying flip-flop.

Application Scenarios

- **Sensor Value Debouncing**: On and off thresholds with a minimum repetition rate.
- **Data Buffer with Output Limit**: Prevents downstream consumers from being flooded with too many updates.
- **Timed Latch Function**: A value is only transferred if the last output was longer than *Tmin*.

Comparison with Similar Components

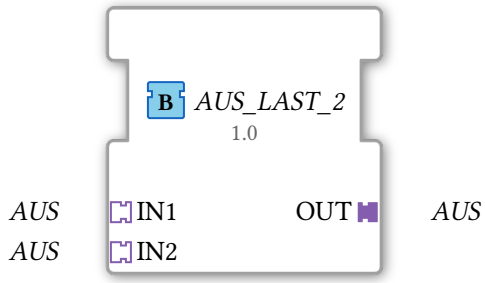
Component	Properties
E_D_FF (Simple D Flip-Flop)	Transfers a value immediately at each clock cycle, no time control.
AUS_D_FF_TMIN	Like **E_D_FF**, but with an additional minimum time between output events.
R_TRIG / F_TRIG	Detect edges, but do not store a data value.

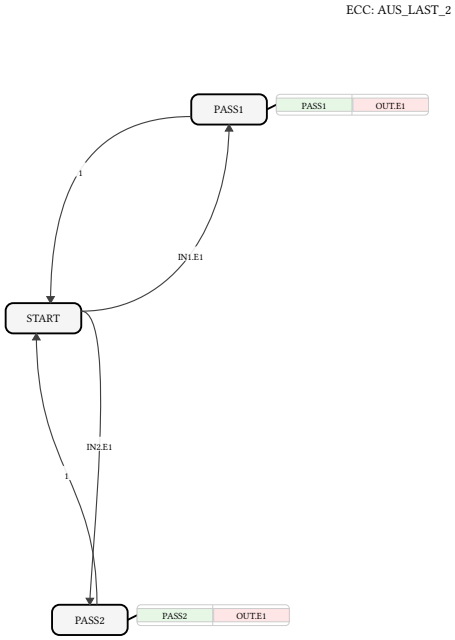
The **AUS_D_FF_TMIN** is therefore particularly suitable for time-critical applications where a minimum sampling or update rate must be maintained.

Conclusion

The **AUS_D_FF_TMIN** is a useful extension module for the D flip-flop, providing time-decoupling of the output. Thanks to its adapter interface and simple configuration via **Tmin**, it can be flexibly integrated into IEC 61499 systems. The existing but unused **HYSTERESIS** input offers potential for future functional extensions.

AUS_LAST_2





Introduction

The AUS_LAST_2 function block merges two same-type USINT adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: USINT adapter (unidirectional) - first signal source
- **IN2**: USINT adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: USINT adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

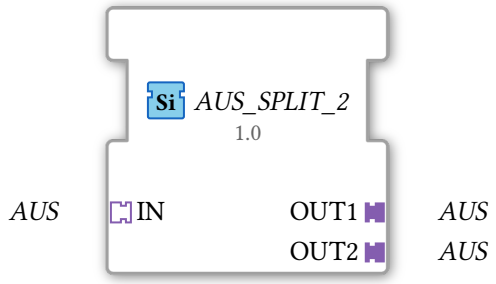
Application Scenarios

- Merging two equally-valid USINT signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same USINT value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AUS_LAST_2 is the inverse of `AUS_SPLIT_2`: AUS_SPLIT_2 distributes 1 input to 2 outputs, AUS_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

OFF_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **OFF_SPLIT_2** is used to distribute an incoming OFF signal to two identical outputs. It is implemented as a generic function block (FB) and is suitable for applications where a universal control signal is required multiple times. The block operates purely on an adapter basis and has no event or data inputs of its own.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Name	Type	Description
Sockets	IN	OFF (unidirectional)	Input adapter that receives the signal to be distributed
Plugs	OUT1	OFF (unidirectional)	First output adapter, identical to the input signal
Plugs	OUT2	OFF (unidirectional)	Second output adapter, identical to the input signal

Functionality

The module forwards the OFF signal present at socket **IN** unchanged to both plugs **OUT1** and **OUT2**. No processing or buffering takes place – the distribution is purely topological. The input signal is available at both outputs simultaneously and without delay. The connection is only activated when the socket is connected to a corresponding adapter.

Technical Features

- **Generic Type**: The function block is declared as `GEN_AUS_SPLIT` with the attribute `eclipse4diac::core::GenericClassName`. This allows it to be reused in different projects without type modification.
- **No State Dependency**: The function block operates statelessly – there is no internal behavior controlled by a state machine.
- **Adapter-Based**: All interfaces are implemented as adapters of type `adapter::types::unidirectional::AUS`. This enables flexible cabling in directional communication.
- **Copyright**: The function block originates from HR Agrartechnik GmbH and is subject to the Eclipse Public License 2.0.

State Overview

Since the function block has no state logic, there is no state machine. Its functionality is limited to simple signal transmission.

Application Scenarios

- **Signal Distribution in Control Engineering**: When a bus signal or a universal control signal needs to be distributed to multiple parallel modules.
- **Test and Simulation Setups**: To split a single test signal across two parallel paths.
- **Redundant Connection**: When a signal needs to be sent to two independent receivers for availability reasons.

Comparison with Similar Components

- **OFF_SPLIT_3 / OFF_SPLIT_N**: Analog components with three or more outputs. `AUS_SPLIT_2` is the simplest option for splitting a signal across two channels.
- **Event-Based Splitters**: Unlike components with event inputs (e.g., `E_SPLIT`), `AUS_SPLIT_2` operates exclusively via adapters and is therefore suitable for pure signal distribution without control logic.
- **Merge Blocks**: While splitters duplicate signals, merge blocks combine multiple signals into one (e.g., `AUS_MERGE_2`).

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AUS_SPLIT_2` is a minimalist yet useful function block for decentralized signal distribution in 4diac applications. Its generic nature and simple adapter interface make it universally applicable, especially when only unidirectional signal copying is required. More complex tasks involving control or processing logic require extended versions.

–

🌐 Related topic subpages on ms-muc-docs.de

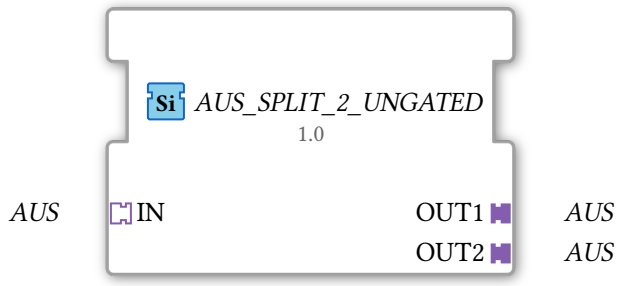
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

OFF_SPLIT_2

> **i UNGATED variant:** This block is the ungated version of `AUS_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/

frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **OFF_SPLIT_2** is used to distribute an incoming OFF signal to two identical outputs. It is implemented as a generic function block (FB) and is suitable for applications where a universal control signal is required multiple times. The block operates purely on an adapter basis and has no event or data inputs of its own.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Name	Type	Description
Sockets	IN	OFF (unidirectional)	Input adapter that receives the signal to be distributed
Plugs	OUT1	OFF (unidirectional)	First output adapter, identical to the input signal
Plugs	OUT2	OFF (unidirectional)	Second output adapter, identical to the input signal

Functionality

The module forwards the OFF signal present at socket **IN** unchanged to both plugs **OUT1** and **OUT2**. No processing or buffering takes place – the distribution is purely topological. The input signal is available at both outputs simultaneously and without delay. The connection is only activated when the socket is connected to a corresponding adapter.

Technical Features

- **Generic Type**: The function block is declared as `GEN_AUS_SPLIT` with the attribute `eclipse4diac::core::GenericClassName`. This allows it to be reused in different projects without type modification.
- **No State Dependency**: The function block operates statelessly – there is no internal behavior controlled by a state machine.
- **Adapter-Based**: All interfaces are implemented as adapters of type `adapter::types::unidirectional::AUS`. This enables flexible cabling in directional communication.

- **Copyright**: The function block originates from HR Agrartechnik GmbH and is subject to the Eclipse Public License 2.0.

State Overview

Since the function block has no state logic, there is no state machine. Its functionality is limited to simple signal transmission.

Application Scenarios

- **Signal Distribution in Control Engineering**: When a bus signal or a universal control signal needs to be distributed to multiple parallel modules.
- **Test and Simulation Setups**: To split a single test signal across two parallel paths.
- **Redundant Connection**: When a signal needs to be sent to two independent receivers for availability reasons.

Comparison with Similar Components

- **OFF_SPLIT_3 / OFF_SPLIT_N**: Analog components with three or more outputs. `AUS_SPLIT_2_UNGATED` is the simplest option for splitting a signal across two channels.
- **Event-Based Splitters**: Unlike components with event inputs (e.g., `E_SPLIT`), `AUS_SPLIT_2_UNGATED` operates exclusively via adapters and is therefore suitable for pure signal distribution without control logic.
- **Merge Blocks**: While splitters duplicate signals, merge blocks combine multiple signals into one (e.g., `AUS_MERGE_2`).

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUS_SPLIT_2_UNGATED` is a minimalist yet useful function block for decentralized signal distribution in 4diac applications. Its generic nature and simple adapter interface make it universally applicable, especially when only unidirectional signal copying is required. More complex tasks involving control or processing logic require extended versions.

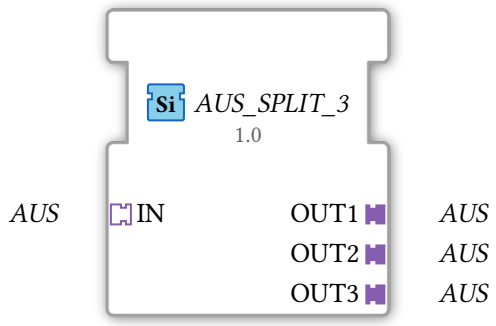
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

OFF_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **OFF_SPLIT_3** serves as a universal distributor for incoming adapter signals of type `adapter::types::unidirectional::AUS`. It receives such a signal and forwards it unchanged to three identical output adapters. This allows control, alarm, or status information to be easily distributed across multiple subsequent blocks.

Interface Structure

Event Inputs

None – signal transmission occurs via the adapter IN.

Event Outputs

None – output occurs via the adapter OUTs.

Data Inputs

None – the block has no dedicated data inputs; all data is transmitted via the adapter `IN`.

Data Outputs

None – the data is output via the adapters `OUT1`, `OUT2`, and `OUT3`.

Adapters

Name	Type	Direction	Description
`IN` ad	`apter::types::unidirectional::AUS`	Socket (incoming)	Receives an OFF signal.
`OUT1` ad	`apter::types::unidirectional::AUS`	Plug (outgoing)	First output of the incoming signal.
`OUT2` ad	`apter::types::unidirectional::AUS`	Plug (outgoing)	Second output of the incoming signal.
`OUT3` ad	`apter::types::unidirectional::AUS`	Plug (outgoing)	Third output of the incoming signal.

Functionality

The FB operates as a pure signal distributor. As soon as an OFF signal (consisting of an event and associated data) is active at socket `IN`, it is passed on identically to all three plugs `OUT1`, `OUT2`, and `OUT3` without delay or transformation. Each output receives the same event and the same data. No buffering, filtering, or logical processing takes place.

Technical Features

- **Generic Type**: The function block is declared as a generic FB (`eclipse4diac::core::GenericClassName = 'GEN_AUS_SPLIT'`). This allows it to be adapted to different OFF adapter types, provided they have the same interface structure.
- **Stateless**: The FB has no internal state diagram (ECC). Its response is purely data flow-driven and requires no initialization or reset logic.
- **License Notice**: The function block is provided under the Eclipse Public License 2.0.

State Overview

Since **AUS_SPLIT_3** has no sequence control or explicit states, a state machine is not required. The behavior is deterministic and immediate: One input signal synchronously generates three identical output signals.

Application Scenarios

- **Distribution of alarm signals** to multiple display or logic units.
- **Multiple use of a sensor signal** in different automation branches.
- **Splitting a control command** (e.g., “Shut down”) across multiple actuators or subsystems.
- **Signal cascading** in modular 4diac applications.

Comparison with similar function blocks

- **EVENT_SPLIT**: Distributes only events (without data). **AUS_SPLIT_3**, on the other hand, distributes complete adapter signals including all associated data.
- **DATA_SPLIT**: Distributes pure data channels and requires separate event control. The OFF splitter encapsulates the event and data together in the adapter.
- **Manual implementation**: Without this function block, each output would have to be connected to its own copy of the source adapter. **AUS_SPLIT_3** simplifies wiring and improves clarity.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AUS_SPLIT_3** is a simple yet useful function block for multiplying OFF adapter signals. Due to its generic design and stateless nature, it is ideally suited for the modular and maintainable structuring of 4diac applications.

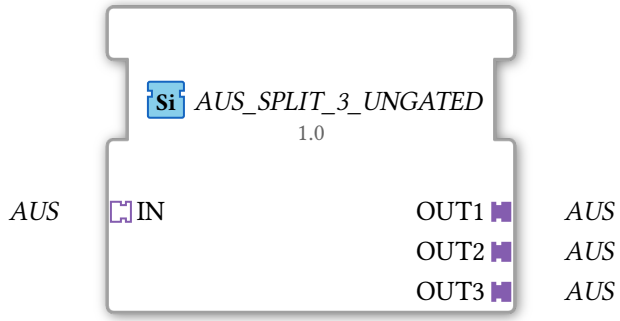
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

OFF_SPLIT_3

> **i UNGATED variant:** This block is the ungated version of `AUS_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **OFF_SPLIT_3** serves as a universal distributor for incoming adapter signals of type `adapter::types::unidirectional::AUS`. It receives such a signal and forwards it unchanged to three identical output adapters. This allows control, alarm, or status information to be easily distributed across multiple subsequent blocks.

Interface Structure

Event Inputs

None – signal transmission occurs via the adapter IN.

Event Outputs

None – output occurs via the adapter OUTs.

Data Inputs

None – the block has no dedicated data inputs; all data is transmitted via the adapter `IN`.

Data Outputs

None – the data is output via the adapters `OUT1`, `OUT2`, and `OUT3`.

Adapters

Name	Type	Direction	Description
IN	adapter::types::unidirectional::AUS	Socket (incoming)	Receives an OFF signal.
OUT1	adapter::types::unidirectional::AUS	Plug (outgoing)	First output of the incoming signal.
OUT2	adapter::types::unidirectional::AUS	Plug (outgoing)	Second output of the incoming signal.
OUT3	adapter::types::unidirectional::AUS	Plug (outgoing)	Third output of the incoming signal.

Functionality

The FB operates as a pure signal distributor. As soon as an OFF signal (consisting of an event and associated data) is active at socket `IN`, it is passed on identically to all three plugs `OUT1`, `OUT2`, and `OUT3` without delay or transformation. Each output receives the same event and the same data. No buffering, filtering, or logical processing takes place.

Technical Features

- **Generic Type**: The function block is declared as a generic FB (`eclipse4diac::core::GenericClassName = 'GEN_AUS_SPLIT'`). This allows it to be adapted to different OFF adapter types, provided they have the same interface structure.
- **Stateless**: The FB has no internal state diagram (ECC). Its response is purely data flow-driven and requires no initialization or reset logic.
- **License Notice**: The function block is provided under the Eclipse Public License 2.0.

State Overview

Since **AUS_SPLIT_3_UNGATED** has no sequence control or explicit states, a state machine is not required. The behavior is deterministic and immediate: One input signal synchronously generates three identical output signals.

Application Scenarios

- **Distribution of alarm signals** to multiple display or logic units.
- **Multiple use of a sensor signal** in different automation branches.
- **Splitting a control command** (e.g., “Shut down”) across multiple actuators or subsystems.
- **Signal cascading** in modular 4diac applications.

Comparison with similar function blocks

- **EVENT_SPLIT**: Distributes only events (without data). **AUS_SPLIT_3_UNGATED**, on the other hand, distributes complete adapter signals including all associated data.
- **DATA_SPLIT**: Distributes pure data channels and requires separate event control. The OFF splitter encapsulates the event and data together in the adapter.
- **Manual implementation**: Without this function block, each output would have to be connected to its own copy of the source adapter. **AUS_SPLIT_3_UNGATED** simplifies wiring and improves clarity.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

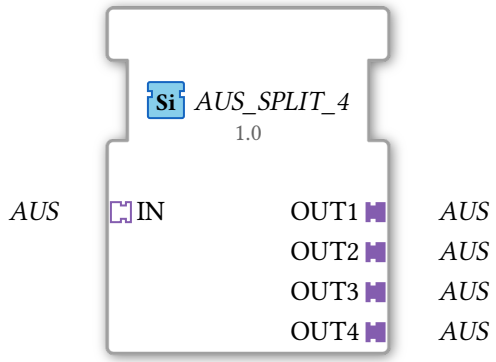
The **AUS_SPLIT_3_UNGATED** is a simple yet useful function block for multiplying OFF adapter signals. Due to its generic design and stateless nature, it is ideally suited for the modular and maintainable structuring of 4diac applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUS_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_SPLIT_4` distributes an incoming **OFF** signal (typically a binary control signal for "Off") to four identical output signals. It serves as a generic splitter and allows a single command or state to be forwarded to multiple downstream components without requiring separate branching logic.

Interface Structure

Event Inputs

No direct event inputs are available. Control is integrated into the input adapter.

Event Outputs

No direct event outputs are available. Control is integrated into the output adapters.

Data Inputs

No direct data inputs are available. Data is received via the input adapter.

Data Outputs

No direct data outputs are available. Data is provided via the output adapters.

Adapters

The function block communicates exclusively via adapters of type `adapter::types::unidirectional::AUS`. This is a unidirectional adapter that transmits a Boolean signal (OFF command).

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AUS`	Receives the original OFF signal.
Plug (Output)	`OUT1`	`adapter::types::unidirectional::AUS`	First outgoing OFF path.
Plug (Output)	`OUT2`	`adapter::types::unidirectional::AUS`	Second outgoing OFF path.
Plug (Output)	`OUT3`	`adapter::types::unidirectional::AUS`	Third outgoing OFF path.
Plug (Output)	`OUT4`	`adapter::types::unidirectional::AUS`	Fourth outgoing OFF path.

Functionality

The module operates as a pure signal distributor: As soon as an OFF signal is present at socket `IN`, it is passed on unchanged to all four plugs (`OUT1` to `OUT4`). No logical processing, delay, or state change takes place. The distribution occurs in parallel and immediately.

- **License**: The source code is subject to the **Eclipse Public License 2.0** (EPL-2.0).
- **Author**: Developed by **HR Agrartechnik GmbH**, Version 1.0, January 24, 2025.

- **Generic Implementation**: The function block is declared as a generic FB (`GenericClassName = 'GEN_AUS_SPLIT'`), which allows for easy adaptation to different signal types or configurations.
- **Adapter-Based**: Communication takes place exclusively via adapters, not via traditional event or data ports. This allows the function block to be seamlessly integrated into an adapter-based architecture.
- **No State Machine**: There is no internal state logic – the distribution is purely combinatorial.

Technical Features

The module does not have a state machine. The output signals follow the input signal directly. Therefore, a graphical state overview is not available.

State Overview

- **Feedback of OFF Signals**: A central "off" command from a controller should simultaneously switch off multiple actuators or subsystems.
- **Redundant Monitoring**: Distributing an OFF signal to multiple monitoring units that must react to the command independently.
- **Modular Machine Structure**: In a modular system, an OFF signal, once detected, is routed to multiple modules via buses or coupling elements. `AUS_SPLIT_4` replaces complex wiring or logical OR operations.
- **Testing and Simulation**: For simultaneously controlling multiple simulated components with the same signal.

Application Scenarios

Function Block	Function	Special Feature
`AUS_SPLIT_2`	Distributes an OFF signal to two outputs	Fewer ports, more compact.
`AUS_SPLIT_4`	Distributes to four outputs	This function block.
`AUS_SPLIT_N`	Configurable splitter (e.g., via generic adapter lists)	More flexible number, but more complex to configure.
`AUS_MERGE`	Combines multiple OFF inputs into one output	Counterpart to the splitter.

Comparison with Similar Function Blocks

The `AUS_SPLIT_4` sits between a simple 2-way splitter and a fully configurable splitter. It is ideal when exactly four outputs are needed – without any additional configuration.

Change Detection

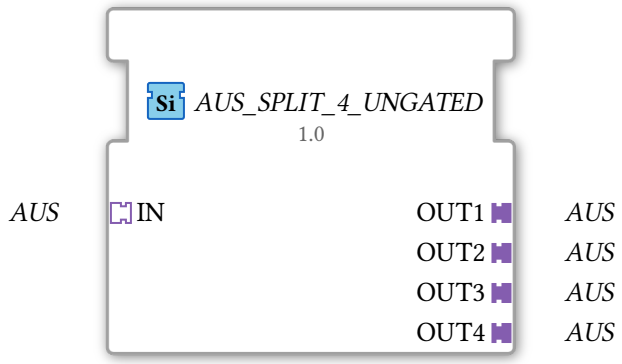
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AUS_SPLIT_4` is a simple yet useful generic function block for distributing an OFF signal to four parallel paths. Its adapter-based interface and clear separation of logic and signal transmission make it a robust component in IEC 61499-based automation. Thanks to its generic declaration, it can be easily integrated into various projects and, if necessary, adapted to custom signal types.

AUS_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_SPLIT_4_UNGATED` distributes an incoming **OFF** signal (typically a binary control signal for "Off") to four identical output signals. It serves as a generic splitter and allows a single command or state to be forwarded to multiple downstream components without requiring separate branching logic.

Interface Structure

Event Inputs

No direct event inputs are available. Control is integrated into the input adapter.

Event Outputs

No direct event outputs are available. Control is integrated into the output adapters.

Data Inputs

No direct data inputs are available. Data is received via the input adapter.

Data Outputs

No direct data outputs are available. Data is provided via the output adapters.

Adapters

The function block communicates exclusively via adapters of type `adapter::types::unidirectional::AUS`. This is a unidirectional adapter that transmits a Boolean signal (OFF command).

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AUS`	Receives the original OFF signal.
Plug (Output)	`OUT1`	`adapter::types::unidirectional::AUS`	First outgoing OFF path.
Plug (Output)	`OUT2`	`adapter::types::unidirectional::AUS`	Second outgoing OFF path.
Plug (Output)	`OUT3`	`adapter::types::unidirectional::AUS`	Third outgoing OFF path.
Plug (Output)	`OUT4`	`adapter::types::unidirectional::AUS`	Fourth outgoing OFF path.

Functionality

The module operates as a pure signal distributor: As soon as an OFF signal is present at socket `IN`, it is passed on unchanged to all four plugs (`OUT1` to `OUT4`). No logical processing, delay, or state change takes place. The distribution occurs in parallel and immediately.

- **License**: The source code is subject to the **Eclipse Public License 2.0** (EPL-2.0).
- **Author**: Developed by **HR Agrartechnik GmbH**, Version 1.0, January 24, 2025.
- **Generic Implementation**: The function block is declared as a generic FB (`GenericClassName = 'GEN_AUS_SPLIT'`), which allows for easy adaptation to different signal types or configurations.
- **Adapter-Based**: Communication takes place exclusively via adapters, not via traditional event or data ports. This allows the function block to be seamlessly integrated into an adapter-based architecture.

- **No State Machine**: There is no internal state logic – the distribution is purely combinatorial.

Technical Features

The module does not have a state machine. The output signals follow the input signal directly. Therefore, a graphical state overview is not available.

State Overview

- **Feedback of OFF Signals**: A central "off" command from a controller should simultaneously switch off multiple actuators or subsystems.
- **Redundant Monitoring**: Distributing an OFF signal to multiple monitoring units that must react to the command independently.
- **Modular Machine Structure**: In a modular system, an OFF signal, once detected, is routed to multiple modules via buses or coupling elements. `AUS_SPLIT_4_UNGATED` replaces complex wiring or logical OR operations.
- **Testing and Simulation**: For simultaneously controlling multiple simulated components with the same signal.

Application Scenarios

Function Block	Function	Special Feature
`AUS_SPLIT_2`	Distributes an OFF signal to two outputs	Fewer ports, more compact.
`AUS_SPLIT_4_UNGATED`	Distributes to four outputs	This function block.
`AUS_SPLIT_N`	Configurable splitter (e.g., via generic adapter lists)	More flexible number, but more complex to configure.
`AUS_MERGE`	Combines multiple OFF inputs into one output	Counterpart to the splitter.

Comparison with Similar Function Blocks

The `AUS_SPLIT_4_UNGATED` sits between a simple 2-way splitter and a fully configurable splitter. It is ideal when exactly four outputs are needed – without any additional configuration.

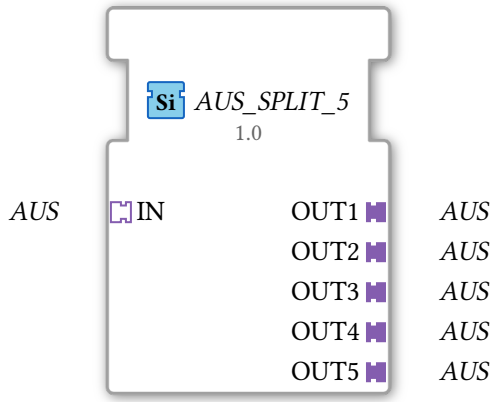
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUS_SPLIT_4_UNGATED` is a simple yet useful generic function block for distributing an OFF signal to four parallel paths. Its adapter-based interface and clear separation of logic and signal transmission make it a robust component in IEC 61499-based automation. Thanks to its generic declaration, it can be easily integrated into various projects and, if necessary, adapted to custom signal types.

OFF_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_SPLIT_5` distributes an incoming OFF event (e.g., a signal or a message) to five identical outputs. It is a generic function block that serves as a placeholder for an application-specific type (`GEN_AUS_SPLIT`) in the 4diac IDE.

Interface Structure

Event Inputs

None (signal exchange occurs exclusively via adapters).

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type
Socket (Input)	`IN`	`adapter::types::unidirectional::AUS`
Plug (Output)	`OUT1`	`adapter::types::unidirectional::AUS`
Plug (Output)	`OUT2`	`adapter::types::unidirectional::AUS`
Plug (Output)	`OUT3`	`adapter::types::unidirectional::AUS`
Plug (Output)	`OUT4`	`adapter::types::unidirectional::AUS`
Plug (Output)	`OUT5`	`adapter::types::unidirectional::AUS`

Functionality

The function block receives a unidirectional OFF event (e.g., a pulse or a logic 1) via socket `IN` and forwards it without delay or processing to all five connected plugs (`OUT1` to `OUT5`). Each outgoing event is an exact copy of the incoming event.

Technical Features

- **Generic Type**: The function block is defined as a generic placeholder by the attribute `GenericClassName = 'GEN_AUS_SPLIT'`. During instantiation, the actual adapter type (e.g., specific event signatures) can be replaced.
- **No States**: The component has no internal logic or state machine – routing is purely structural.
- **Unidirectional Communication**: All adapters are declared as `unidirectional::AUS`, meaning data flows only from the input to the outputs.

State Overview

The component has no internal state. It operates deterministically and entirely in a passive mode.

Application Scenarios

- **Signal Distribution**: A central event (e.g., “Start” or “Alarm”) is intended to control multiple components in parallel.
- **Bus Topologies**: Replacement for multiple manual wiring of a signal in the IEC 61499 environment.
- **Prototyping**: Rapid distribution of an event source across five target components during the development phase.

Comparison with Similar Function Blocks

- **OFF_SPLIT_2 / OFF_SPLIT_3** – Function blocks with the same functionality but a different number of outputs (2 or 3).
- **E_SPLIT** – A standard event split function block that uses event inputs and outputs instead of adapters. `AUS_SPLIT_5` is adapter-based and therefore more flexible in its reuse with different protocols.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output’s current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

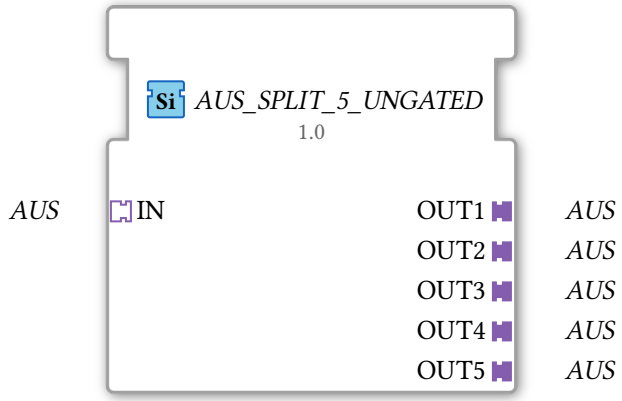
AUS_SPLIT_5` is a simple, generic distribution function block for unidirectional OFF signals. It simplifies modular control logic by implementing a 1:5 split without additional logic. Thanks to the adapter interface, it can be used in different contexts (e.g., event, data, or mixed streams).

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

OFF_SPLIT_5

> **i UNGATED variant:** This block is the ungated version of `AUS_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_SPLIT_5_UNGATED` distributes an incoming OFF event (e.g., a signal or a message) to five identical outputs. It is a generic function block that serves as a placeholder for an application-specific type (`GEN_AUS_SPLIT`) in the 4diac IDE.

Interface Structure

Event Inputs
None (signal exchange occurs exclusively via adapters).

Data Inputs
None.

Data Outputs
None.

Adapters

Direction	Name	Type
Socket (Input)	`IN`	`adapter::types::unidirectional::AUS`
Plug (Output)	`OUT1`	`adapter::types::unidirectional::AUS`
Plug (Output)	`OUT2`	`adapter::types::unidirectional::AUS`
Plug (Output)	`OUT3`	`adapter::types::unidirectional::AUS`
Plug (Output)	`OUT4`	`adapter::types::unidirectional::AUS`
Plug (Output)	`OUT5`	`adapter::types::unidirectional::AUS`

Functionality
The function block receives a unidirectional OFF event (e.g., a pulse or a logic 1) via socket `IN` and forwards it without delay or processing to all five connected plugs (`OUT1` to `OUT5`). Each outgoing event is an exact copy of the incoming event.

Technical Features
- **Generic Type**: The function block is defined as a generic placeholder by the attribute `GenericClassName = 'GEN_AUS_SPLIT'`. During instantiation, the actual adapter type (e.g., specific event signatures) can be replaced.
- **No States**: The component has no internal logic or state machine – routing is purely structural.
- **Unidirectional Communication**: All adapters are declared as `unidirectional::AUS`, meaning data flows only from the input to the outputs.

State Overview
The component has no internal state. It operates deterministically and entirely in a passive mode.

Application Scenarios
- **Signal Distribution**: A central event (e.g., "Start" or "Alarm") is intended to control multiple components in parallel.
- **Bus Topologies**: Replacement for multiple manual wiring of a signal in the IEC 61499 environment.
- **Prototyping**: Rapid distribution of an event source across five target components during the development phase.

Comparison with Similar Function Blocks
- **OFF_SPLIT_2 / OFF_SPLIT_3** – Function blocks with the same functionality but a different number of outputs (2 or 3).
- **E_SPLIT** – A standard event split function block that uses event inputs and outputs instead of adapters. `AUS_SPLIT_5_UNGATED` is adapter-based and therefore more flexible in its reuse with different protocols.

Change Detection
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

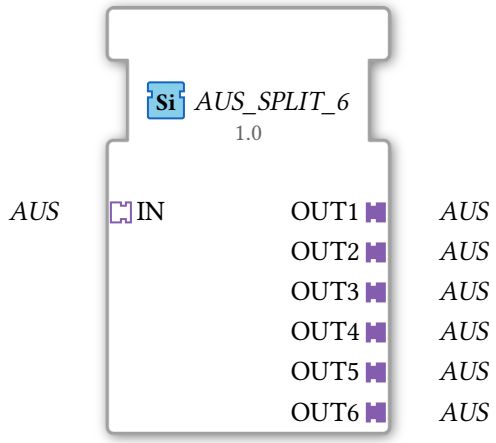
Conclusion

AUS_SPLIT_5_UNGATED` is a simple, generic distribution function block for unidirectional OFF signals. It simplifies modular control logic by implementing a 1:5 split without additional logic. Thanks to the adapter interface, it can be used in different contexts (e.g., event, data, or mixed streams).

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

OFF_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **OFF_SPLIT_6** is a generic block that distributes an incoming OFF signal (e.g., a uniform control signal or event) to six identical outputs. It serves as a splitter in IEC 61499 communication via adapters and enables simple multiple use of a signal without additional logic.

Interface Structure

Event Inputs

- No independent event inputs are available. All communication takes place via the integrated adapter socket.

Event Outputs

- No independent event outputs are available. Outputs are provided exclusively via the adapter plugs.

Data Inputs

- No direct data inputs. Data transfer is implemented indirectly via the adapter socket.

Data Outputs

- No direct data outputs. The output data is provided via the adapter plugs.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AUS`	**IN**	Socket	Incoming output signal distributed to all outputs.
`adapter::types::unidirectional::AUS`	**OUT1**	Plug	First output – identical to **IN**.
`adapter::types::unidirectional::AUS`	**OUT2**	Plug	Second output – identical to **IN**.
`adapter::types::unidirectional::AUS`	**OUT3**	Plug	Third output – identical to **IN**.

`adapter::types::unidirectional::AUS`	**OUT4**	Plug	Fourth output – identical to **IN**.
`adapter::types::unidirectional::AUS**	**OUT5**	Plug	Fifth output – identical to **IN**.
`adapter::types::unidirectional::AUS`	**OUT6**	Plug	Sixth output – identical to **IN**.

Note: The adapter type "AUS" is a unidirectional adapter that transmits a signal (event + data, if applicable) in one direction.

Functionality

The module functions as a passive distributor. The signal present at socket **IN** is copied unchanged and without delay to all six plugs **OUT1** to **OUT6**. No filtering, transformation, or buffering takes place. The function block behaves like a "fan-out" node for AUS adapter connections.

Technical Features

- **Generic Type**: The function block is implemented as a generic function block (GenericClassName = `'GEN_AUS_SPLIT'`). In the 4diac IDE, it can be instantiated depending on the specific AUS adapter type.
- **Dependencies**: Requires the adapter `adapter::types::unidirectional::AUS` from the package `adaper::events::unidirectional`. The implementation imports the Eclipse 4diac base types `GenericClassName` and `TypeHash`.
- **No State**: The function block does not have an internal state machine (no ECC). It operates purely combinatorially – any change at the input is immediately passed through to all outputs.
- **Unidirectional**: The adapters are designed for signal flow in only one direction. Feedback from the outputs to the input is not possible.

State Overview

The module has **no explicit states**. Its behavior is determined solely by the current configuration of the input adapter. State changes in the sense of an automaton do not occur.

Application Scenarios

- **Distributing an "OFF" Control Signal**: In a system, a single stop or shutdown signal is to be simultaneously forwarded to multiple actuators (motors, valves, etc.).
- **Redundant Signal Paths**: A signal is split across multiple parallel paths to trigger different actions in subsequent processing steps.
- **Test and Simulation Environments**: A generated test signal is sent unchanged to various monitoring or logic modules.

Comparison with Similar Function Blocks

- **Event Splitter (e.g., `E_SPLIT`)**: Processes discrete events at its own input/output events. In contrast, `AUS_SPLIT_6` operates at the adapter level and distributes both the event and its associated data in one step.
- **Data Splitter**: Pure data distributors (e.g., `F_MUX`, `F_DIST`) require separate data types. The present function block is specific to the adapter type `AUS` and encapsulates the signal structure.
- **Generic Capability**: Thanks to its generic declaration, the function block can be reused in various contexts with different output adapter implementations.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **OFF_SPLIT_6** is a simple yet useful generic function block for signal distribution in IEC 61499 applications. It reduces wiring complexity by converting a single OFF signal to six parallel outputs. Its generic nature makes it versatile, as long as the OFF adapter used adheres to the unidirectional contract.

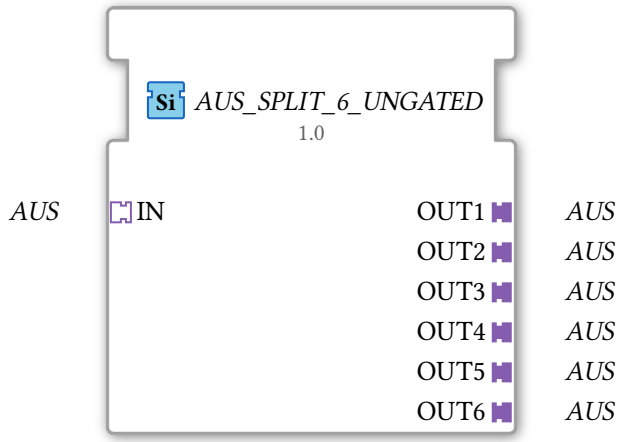
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

OFF_SPLIT_6

i UNGATED variant: This block is the ungated version of AUS_SPLIT_6. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **OFF_SPLIT_6** is a generic block that distributes an incoming OFF signal (e.g., a uniform control signal or event) to six identical outputs. It serves as a splitter in IEC 61499 communication via adapters and enables simple multiple use of a signal without additional logic.

Interface Structure

Event Inputs

- No independent event inputs are available. All communication takes place via the integrated adapter socket.

Event Outputs

- No independent event outputs are available. Outputs are provided exclusively via the adapter plugs.

Data Inputs

- No direct data inputs. Data transfer is implemented indirectly via the adapter socket.

Data Outputs

- No direct data outputs. The output data is provided via the adapter plugs.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AUS`	**IN**	Socket	Incoming output signal distributed to all outputs.
`adapter::types::unidirectional::AUS`	**OUT1**	Plug	First output – identical to **IN**.
`adapter::types::unidirectional::AUS`	**OUT2**	Plug	Second output – identical to **IN**.
`adapter::types::unidirectional::AUS`	**OUT3**	Plug	Third output – identical to **IN**.

`adapter::types::unidirectional::AUS`	**OUT4**	Plug	Fourth output – identical to **IN**.
`adapter::types::unidirectional::AUS**	**OUT5**	Plug	Fifth output – identical to **IN**.
`adapter::types::unidirectional::AUS`	**OUT6**	Plug	Sixth output – identical to **IN**.

Note: The adapter type "AUS" is a unidirectional adapter that transmits a signal (event + data, if applicable) in one direction.

Functionality

The module functions as a passive distributor. The signal present at socket **IN** is copied unchanged and without delay to all six plugs **OUT1** to **OUT6**. No filtering, transformation, or buffering takes place. The function block behaves like a "fan-out" node for AUS adapter connections.

Technical Features

- **Generic Type**: The function block is implemented as a generic function block (GenericClassName = `'GEN_AUS_SPLIT'`). In the 4diac IDE, it can be instantiated depending on the specific AUS adapter type.
- **Dependencies**: Requires the adapter `adapter::types::unidirectional::AUS` from the package `adaper::events::unidirectional`. The implementation imports the Eclipse 4diac base types `GenericClassName` and `TypeHash`.
- **No State**: The function block does not have an internal state machine (no ECC). It operates purely combinatorially – any change at the input is immediately passed through to all outputs.
- **Unidirectional**: The adapters are designed for signal flow in only one direction. Feedback from the outputs to the input is not possible.

State Overview

The module has **no explicit states**. Its behavior is determined solely by the current configuration of the input adapter. State changes in the sense of an automaton do not occur.

Application Scenarios

- **Distributing an "OFF" Control Signal**: In a system, a single stop or shutdown signal is to be simultaneously forwarded to multiple actuators (motors, valves, etc.).
- **Redundant Signal Paths**: A signal is split across multiple parallel paths to trigger different actions in subsequent processing steps.
- **Test and Simulation Environments**: A generated test signal is sent unchanged to various monitoring or logic modules.

Comparison with Similar Function Blocks

- **Event Splitter (e.g., `E_SPLIT`)**: Processes discrete events at its own input/output events. In contrast, `AUS_SPLIT_6_UNGATED` operates at the adapter level and distributes both the event and its associated data in one step.
- **Data Splitter**: Pure data distributors (e.g., `F_MUX`, `F_DIST`) require separate data types. The present function block is specific to the adapter type `AUS` and encapsulates the signal structure.
- **Generic Capability**: Thanks to its generic declaration, the function block can be reused in various contexts with different output adapter implementations.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

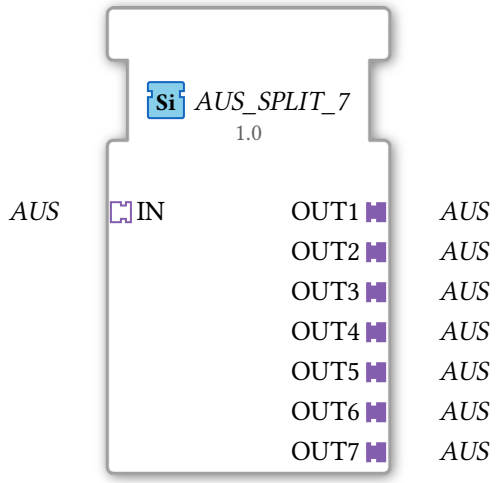
The **OFF_SPLIT_6** is a simple yet useful generic function block for signal distribution in IEC 61499 applications. It reduces wiring complexity by converting a single OFF signal to six parallel outputs. Its generic nature makes it versatile, as long as the OFF adapter used adheres to the unidirectional contract.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

OFF_SPLIT_7



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **OFF_SPLIT_7** is a generic block for distributing a single OFF adapter signal to seven identical OFF outputs. It serves as a signal multiplier in control applications based on the IEC 61499 standard. The block has no event- or data-based interfaces, but communicates exclusively via adapters of type `adapter::types::unidirectional::AUS`. The actual signal type is only determined at runtime via the generic attribute `GenericClassName`.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapter

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AUS`	Uniform OFF input signal
Plug (Output)	`OUT1` – `OUT7`	`adapter::types::unidirectional::AUS`	Seven parallel OFF signals

Functionality

The module receives an OFF signal via the adapter socket `IN`. This signal is passed on unchanged and without delay to all seven adapter plugs `OUT1` to `OUT7`. Since the function block is declared as a generic function block (`GenericClassName = 'GEN_AUS_SPLIT'`), the specific type of the OFF adapter can be overridden at design time or during system configuration. The behavior always remains the same: a one-to-many distribution.

Technical Features

- **Generic Function Block**: The actual adapter type is defined via the attribute `eclipse4diac::core::GenericClassName`. By default, `'GEN_AUS_SPLIT'` is set.

- **No ECC / State Machine**: Due to the absence of event inputs, the function block does not have an internal state machine. The outputs follow the input signal directly (combinatory behavior).
- **Pure Adapter Communication**: No IEC 61499 data types (BOOL, INT, etc.) are used; all data transmission occurs via unidirectional OFF adapters.
- **Compatibility**: The function block is designed for use in 4diac IDE and runtime environments that support adapters of type `adapter::types::unidirectional::AUS`.

State Overview

The function block does not have a state machine. The output signal of each of the seven OUT adapters always corresponds to the current value of the IN adapter. There is no time delay or state storage.

Application Scenarios

- **Signal Distribution in Redundant Systems**: A central OFF control signal (e.g., "emergency stop") is to be forwarded to several decentralized actuators or subsystems.
- **Modular Automation Structures**: An OFF signal received from a higher-level controller is split into seven parallel paths that control independent machine modules.
- **Test and Simulation Environments**: For easily replicating a test signal without changing the logic.

Comparison with Similar Function Blocks

- **AUS_SPLIT_N** (e.g., for 2, 4, or 8 outputs) exist in variants with different numbers of outputs. AUS_SPLIT_7 is a specific variant for exactly seven outputs.
- Unlike data splitters (e.g., `SPLIT_INT`, `SPLIT_BOOL`), this function block works exclusively with adapters and not with elementary data types. This makes it particularly suitable for protocol-based or type-safe connections.
- Function blocks with event interfaces (e.g., `E_SPLIT`) distribute events, not signals. AUS_SPLIT_7, on the other hand, distributes a continuous signal value.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AUS_SPLIT_7** is a simple yet effective generic function block for the unidirectional distribution of an OFF adapter signal to seven parallel outputs. Its purely adapter-based interface and lack of event logic make it suitable for all applications where a signal needs to be duplicated without delay or state storage. It offers a clean, modular solution for signal distribution in IEC 61499-based automation systems.

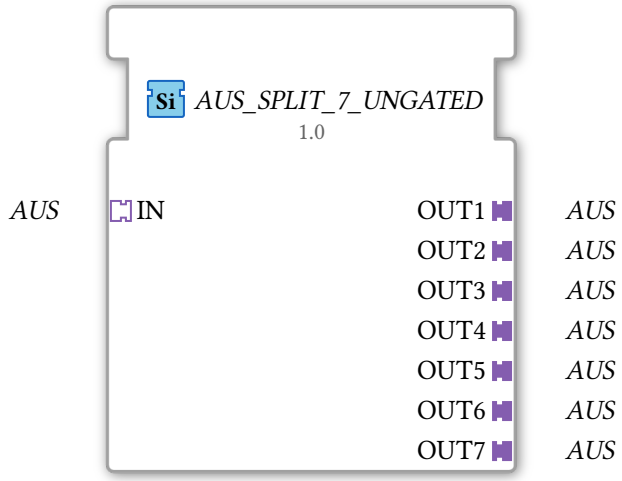
🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

OFF_SPLIT_7

> **i UNGATED variant:** This block is the ungated version of `AUS_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **OFF_SPLIT_7** is a generic block for distributing a single OFF adapter signal to seven identical OFF outputs. It serves as a signal multiplier in control applications based on the IEC 61499 standard. The block has no event- or data-based interfaces, but communicates exclusively via adapters of type `adapter::types::unidirectional::AUS`. The actual signal type is only determined at runtime via the generic attribute `GenericClassName`.

Interface Structure

Event Inputs

None available.

Event Outputs

None available.

Data Inputs

None available.

Data Outputs

None available.

Adapter

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AUS`	Uniform OFF input signal
Plug (Output)	`OUT1 – OUT7`	`adapter::types::unidirectional::AUS`	Seven parallel OFF signals

Functionality

The module receives an OFF signal via the adapter socket `IN`. This signal is passed on unchanged and without delay to all seven adapter plugs `OUT1` to `OUT7`. Since the function block is declared as a generic function block (`GenericClassName = 'GEN_AUS_SPLIT'`), the specific type of the OFF adapter can be overridden at design time or during system configuration. The behavior always remains the same: a one-to-many distribution.

Technical Features

- **Generic Function Block**: The actual adapter type is defined via the attribute `eclipse4diac::core::GenericClassName`. By default, `'GEN_AUS_SPLIT'` is set.

- **No ECC / State Machine**: Due to the absence of event inputs, the function block does not have an internal state machine. The outputs follow the input signal directly (combinatory behavior).
- **Pure Adapter Communication**: No IEC 61499 data types (BOOL, INT, etc.) are used; all data transmission occurs via unidirectional OFF adapters.
- **Compatibility**: The function block is designed for use in 4diac IDE and runtime environments that support adapters of type `adapter::types::unidirectional::AUS`.

State Overview

The function block does not have a state machine. The output signal of each of the seven OUT adapters always corresponds to the current value of the IN adapter. There is no time delay or state storage.

Application Scenarios

- **Signal Distribution in Redundant Systems**: A central OFF control signal (e.g., "emergency stop") is to be forwarded to several decentralized actuators or subsystems.
- **Modular Automation Structures**: An OFF signal received from a higher-level controller is split into seven parallel paths that control independent machine modules.
- **Test and Simulation Environments**: For easily replicating a test signal without changing the logic.

Comparison with Similar Function Blocks

- **AUS_SPLIT_N** (e.g., for 2, 4, or 8 outputs) exist in variants with different numbers of outputs. AUS_SPLIT_7_UNGATED is a specific variant for exactly seven outputs.
- Unlike data splitters (e.g., `SPLIT_INT`, `SPLIT_BOOL`), this function block works exclusively with adapters and not with elementary data types. This makes it particularly suitable for protocol-based or type-safe connections.
- Function blocks with event interfaces (e.g., `E_SPLIT`) distribute events, not signals. AUS_SPLIT_7_UNGATED, on the other hand, distributes a continuous signal value.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

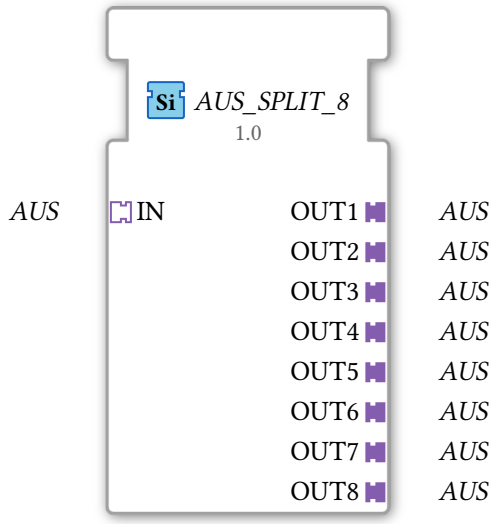
The **AUS_SPLIT_7_UNGATED** is a simple yet effective generic function block for the unidirectional distribution of an OFF adapter signal to seven parallel outputs. Its purely adapter-based interface and lack of event logic make it suitable for all applications where a signal needs to be duplicated without delay or state storage. It offers a clean, modular solution for signal distribution in IEC 61499-based automation systems.

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

OFF_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The **OFF_SPLIT_8** is a generic function block for distributing an incoming OFF signal to up to eight separate outputs. It serves as a simple "splitter" for signals of the type **adapter::types::unidirectional::OFF** and enables the parallel control of multiple devices or subsequent function blocks.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Interface	Type	Direction	Description
IN	`adapter::types::unidirectional::AUS`	Socket	Input – receives the OFF signal to be distributed
OUT1	`adapter::types::unidirectional::AUS`	Plug	Output 1 – forwards the incoming signal
OUT2	`adapter::types::unidirectional::AUS`	Plug	Output 2 – forwards the incoming signal
OUT3	`adapter::types::unidirectional::AUS`	Plug	Output 3 – forwards the incoming signal
OUT4	`adapter::types::unidirectional::AUS`	Plug	Output 4 – forwards the incoming signal
OUT5	`adapter::types::unidirectional::AUS`	Plug	Output 5 – forwards the incoming signal

OUT6	`adapter::types::unidirectional::AUS`	Plug	Output 6 – forwards the incoming signal
OUT7	`adapter::types::unidirectional::AUS`	Plug	Output 7 – forwards the incoming signal
OUT8	`adapter::types::unidirectional::AUS`	Plug	Output 8 – forwards the incoming signal

Functionality

This function block forwards the OFF signal present at socket **IN** unchanged to all eight plugs **OUT1** to **OUT8**. No delay, filtering, or logical processing takes place. The signal is available on all outputs simultaneously. The function corresponds to a 1:8 distribution.

Technical Features

- **Generic Function Block**: The function block is executed as a generic type (`GEN_AUS_SPLIT`). The specific type is determined at runtime via the attribute `eclipse4diac::core::GenericClassName`.
- **No State Machine**: Since the function block only performs direct signal forwarding without logic, it does not have its own execution control (ECC). Signal transmission occurs solely via the adapter interfaces.
- **Adapter Type**: All interfaces use the unidirectional `AUS` adapter type, which enables standardized signal transmission.

State Overview

The function block does not have a state machine. There are no internal states or transitions.

Application Scenarios

- **Distributing a Central Control Signal**: A single "off" signal (e.g., a switch-off command) is passed on to multiple actuators or sub-functions.
- **Parallel Control of Loads**: In agricultural or industrial control systems, when a signal needs to control multiple valves, motors, or lights simultaneously.
- **Splitting Bus Signals**: When an adapter-based signal path needs to be branched into multiple parallel paths.

Comparison with Similar Function Blocks

- **AUS_SPLIT_4**: Distributes a signal to four outputs. The AUS_SPLIT_8 offers higher fan-out capacity with eight outputs.
- **AUS_MERGE / AUS_JOIN**: Combine multiple signals – functionally the opposite.
- **AUS_ROUTER**: Can selectively route a signal to one of several outputs, while the splitter always activates all outputs.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AUS_SPLIT_8** is a simple yet useful function block for multiplying an OFF signal. Its lack of logic and states makes it efficient and easy to understand. It is particularly suitable for applications where a control signal needs to be distributed to multiple receivers without requiring selection or processing.

AUS_MERGE / AUS_JOIN —

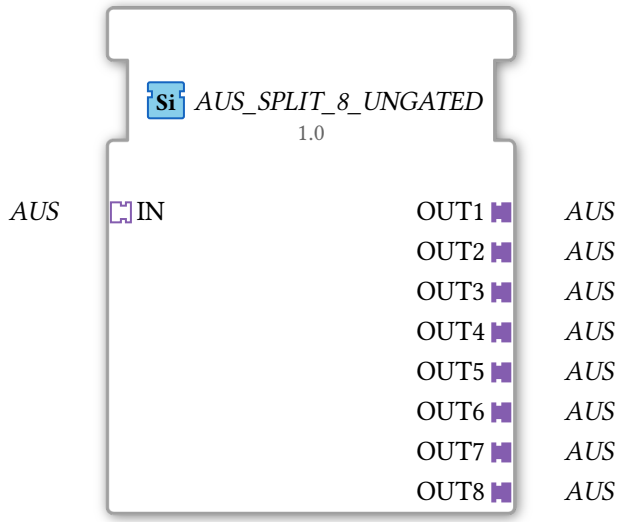
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

]

OFF_SPLIT_8

i UNGATED variant: This block is the ungated version of AUS_SPLIT_8. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **OFF_SPLIT_8** is a generic function block for distributing an incoming OFF signal to up to eight separate outputs. It serves as a simple "splitter" for signals of the type **adapter::types::unidirectional::OFF** and enables the parallel control of multiple devices or subsequent function blocks.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Interface	Type	Direction	Description
IN	`adapter::types::unidirectional::AUS`	Socket	Input – receives the OFF signal to be distributed
OUT1	`adapter::types::unidirectional::AUS`	Plug	Output 1 – forwards the incoming signal
OUT2	`adapter::types::unidirectional::AUS`	Plug	Output 2 – forwards the incoming signal

OUT3	adapter::types::unidirectional::AUS	Plug	Output 3 – forwards the incoming signal
OUT4	adapter::types::unidirectional::AUS	Plug	Output 4 – forwards the incoming signal
OUT5	adapter::types::unidirectional::AUS	Plug	Output 5 – forwards the incoming signal
OUT6	adapter::types::unidirectional::AUS	Plug	Output 6 – forwards the incoming signal
OUT7	adapter::types::unidirectional::AUS	Plug	Output 7 – forwards the incoming signal
OUT8	adapter::types::unidirectional::AUS	Plug	Output 8 – forwards the incoming signal

Functionality

This function block forwards the OFF signal present at socket **IN** unchanged to all eight plugs **OUT1** to **OUT8**. No delay, filtering, or logical processing takes place. The signal is available on all outputs simultaneously. The function corresponds to a 1:8 distribution.

Technical Features

- **Generic Function Block**: The function block is executed as a generic type (`GEN_AUS_SPLIT`). The specific type is determined at runtime via the attribute `eclipse4diac::core::GenericClassName`.
- **No State Machine**: Since the function block only performs direct signal forwarding without logic, it does not have its own execution control (ECC). Signal transmission occurs solely via the adapter interfaces.
- **Adapter Type**: All interfaces use the unidirectional `AUS` adapter type, which enables standardized signal transmission.

State Overview

The function block does not have a state machine. There are no internal states or transitions.

Application Scenarios

- **Distributing a Central Control Signal**: A single "off" signal (e.g., a switch-off command) is passed on to multiple actuators or sub-functions.
- **Parallel Control of Loads**: In agricultural or industrial control systems, when a signal needs to control multiple valves, motors, or lights simultaneously.
- **Splitting Bus Signals**: When an adapter-based signal path needs to be branched into multiple parallel paths.

Comparison with Similar Function Blocks

- **AUS_SPLIT_4**: Distributes a signal to four outputs. The AUS_SPLIT_8_UNGATED offers higher fan-out capacity with eight outputs.
- **AUS_MERGE / AUS_JOIN**: Combine multiple signals – functionally the opposite.
- **AUS_ROUTER**: Can selectively route a signal to one of several outputs, while the splitter always activates all outputs.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AUS_SPLIT_8_UNGATED** is a simple yet useful function block for multiplying an OFF signal. Its lack of logic and states makes it efficient and easy to understand. It is particularly suitable for applications where a control signal needs to be distributed to multiple receivers without requiring selection or processing.

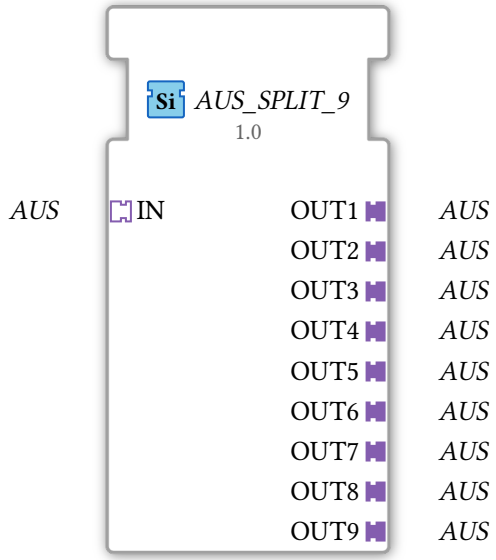
AUS_MERGE / AUS_JOIN –

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Total resistance in series & parallel circuits on ms-muc-docs.de

]

OFF_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The **OFF_SPLIT_9** function block is a generic component that distributes an incoming OFF signal to nine separate outputs. It serves as a fan-out for unidirectional adapter connections and is particularly suitable for applications where a signal needs to be forwarded to multiple receivers in parallel.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Name	Type	Description
SOCKET	IN	`adapter::types::unidirectional::AUS`	Incoming OFF signal
PLUG	OUT1 ... OUT9	`adapter::types::unidirectional::AUS`	Nine outgoing OFF signals

Functionality

This function block forwards the OFF signal arriving at the SOCKET *IN* unchanged to all nine PLUG outputs *OUT1* to *OUT9*. The forwarding occurs continuously without any additional logic or delay. Since

this is a generic function block, the specific data format of the OFF adapter type is not defined – it can be Boolean values, scalars, or complex structures, depending on the adapter definition used.

Technical Features

- **Generic Function Block**: The function block is declared as generic by the attribute `GenericClassName` (`GEN_AUS_SPLIT`) and can be parameterized for various OFF adapter instances.
- **Adapter-Based**: Communication occurs exclusively via adapters (sockets/plugs) – no event or data inputs are required.
- **Unidirectional**: The adapter type used, `adapter::types::unidirectional::AUS`, supports only one direction of data flow (from the socket to the plugs).

State Overview

The function block does not have its own states or event flow control (ECC). Signal transmission is purely data-driven via the adapter connections.

Application Scenarios

- **Signal Distribution**: A central control signal (e.g., "OFF" for a machine) is to be sent to multiple actuators or subsystems simultaneously.
- **Redundancy**: Parallel outputs for redundant control paths.
- **Test and Simulation Setups**: A simulation signal is distributed across multiple observers or logging blocks.

Comparison with Similar Function Blocks

Function Block	Type	Outputs	Special Feature
OFF_SPLIT_9	Adapter	9	Generic, for unidirectional OFF adapters
OFF_SPLIT_4	Adapter	4	Fewer outputs
SPLIT_1_2 (e.g., for data)	Data	any	Works with Data-Event Combinations

Unlike data-based split blocks, AUS_SPLIT_9 does not require event control, as the adapter connection handles data transmission implicitly.

Change Detection

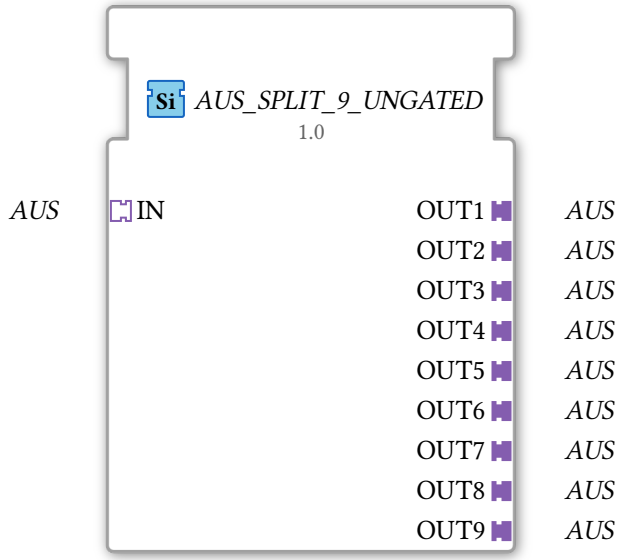
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AUS_SPLIT_9** is a simple yet effective function block for splitting a single OFF signal into nine parallel paths. Its generic nature and pure adapter interface make it flexible for use in IEC 61499-based control applications that require unidirectional signal duplication.

OFF_SPLIT_9

> **i UNGATED variant:** This block is the ungated version of AUS_SPLIT_9. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **OFF_SPLIT_9** function block is a generic component that distributes an incoming OFF signal to nine separate outputs. It serves as a fan-out for unidirectional adapter connections and is particularly suitable for applications where a signal needs to be forwarded to multiple receivers in parallel.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Direction	Name	Type	Description
SOCKET	IN	`adapter::types::unidirectional::AUS`	Incoming OFF signal
PLUG	OUT1 ... OUT9	`adapter::types::unidirectional::AUS`	Nine outgoing OFF signals

Functionality

This function block forwards the OFF signal arriving at the SOCKET *IN* unchanged to all nine PLUG outputs *OUT1* to *OUT9*. The forwarding occurs continuously without any additional logic or delay. Since this is a generic function block, the specific data format of the OFF adapter type is not defined – it can be Boolean values, scalars, or complex structures, depending on the adapter definition used.

Technical Features

- **Generic Function Block**: The function block is declared as generic by the attribute `GenericClassName` (`GEN_AUS_SPLIT`) and can be parameterized for various OFF adapter instances.
- **Adapter-Based**: Communication occurs exclusively via adapters (sockets/plugs) – no event or data inputs are required.

- **Unidirectional**: The adapter type used, `adapter::types::unidirectional::AUS`, supports only one direction of data flow (from the socket to the plugs).

State Overview

The function block does not have its own states or event flow control (ECC). Signal transmission is purely data-driven via the adapter connections.

Application Scenarios

- **Signal Distribution**: A central control signal (e.g., "OFF" for a machine) is to be sent to multiple actuators or subsystems simultaneously.
- **Redundancy**: Parallel outputs for redundant control paths.
- **Test and Simulation Setups**: A simulation signal is distributed across multiple observers or logging blocks.

Comparison with Similar Function Blocks

Function Block	Type	Outputs	Special Feature
OFF_SPLIT_9	Adapter	9	Generic, for unidirectional OFF adapters
OFF_SPLIT_4	Adapter	4	Fewer outputs
SPLIT_1_2 (e.g., for data)	Data	any	Works with Data-Event Combinations

Unlike data-based split blocks, AUS_SPLIT_9_UNGATED does not require event control, as the adapter connection handles data transmission implicitly.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

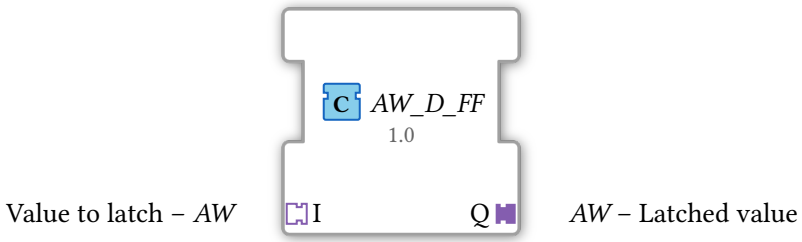
Conclusion

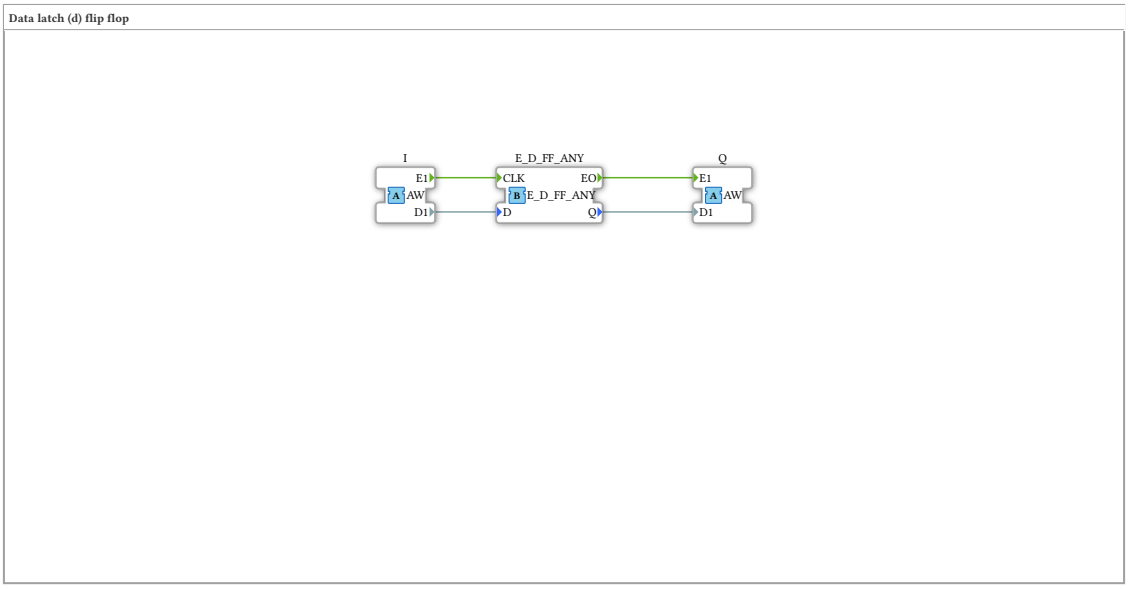
The **AUS_SPLIT_9_UNGATED** is a simple yet effective function block for splitting a single OFF signal into nine parallel paths. Its generic nature and pure adapter interface make it flexible for use in IEC 61499-based control applications that require unidirectional signal duplication.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AW_D_FF





• — ○ • — ○ • — ○ •

Introduction

The function block **AW_D_FF** implements a data-controlled flip-flop (D-latch) according to IEC 61499. It accepts a binary data value at its input and outputs it in a clock-controlled manner. The interface is exclusively via adapters, which enables modular and flexible integration into existing components.

Interface Structure

Event Inputs

No direct event inputs at the FB level. Event control is handled via the input adapter **I**.

Event Outputs

No direct event outputs at the FB level. Event output is handled via the output adapter **Q**.

Data Inputs

No direct data inputs at the FB level. Data is transferred via the input adapter **I**.

Data Outputs

No direct data outputs at the FB level. Data output is via the output adapter **Q**.

Adapters

Adapter	Type	Description
I (Socket)	`adapter::types::unidirectional::AW`	Input adapter – receives the data to be latched (D1) and the clock signal (E1).
Q (Plug)	`adapter::types::unidirectional::AW`	Output adapter – outputs the latched data (D1) and an acknowledgment event (E1).

Both adapters are of the same unidirectional type and each encapsulates one event and one data value.

Functionality

The component internally uses the IEC 61499 standard component `E_D_FF_ANY`.

On a rising edge of the clock signal (event `E1` at adapter **I**), the current data value (`D1`) is taken and made available at the output. The output value remains until a new rising clock edge takes over a new value.

The logic table (D flip-flop) is as follows:

Clock (E1)	Input D	Output Q (after clock)

↑	0	0
↑	1	1
otherwise	arbitrary	unchanged

Technical Features

- **Adapter-Based Interface**: The function block uses adapters exclusively for communication, enabling loose coupling and easy reuse in various contexts.
- **Standard Compliance**: Developed according to IEC 61499-1 Annex A and licensed under the Eclipse Public License 2.0 (EPL-2.0).
- **Internal Implementation**: Utilizes the predefined function block `E_D_FF_ANY`, which provides a general-purpose D flip-flop.
- **Unidirectional Data Flow**: The adapters are designed to be unidirectional, meaning that data and event flow is defined in only one direction.

State Overview

A D flip-flop has two stable states for the output Q (logic 0 or 1). The state change occurs exclusively on a rising edge of the clock signal:

- **State 0**: Output Q = 0 (as long as no clock signal with D=1 arrives)
- **State 1**: Output Q = 1 (after a clock signal with D=1, until a clock signal with D=0 arrives)

There is no other internal state machine; the functionality is purely storage-based.

Application Scenarios

- **Synchronization of Asynchronous Signals**: A stable data value is acquired at a defined time (clock signal).
- **State Register**: As a 1-bit memory in simple control logic circuits.
- **Data Buffer**: Short-term storage of a binary value for subsequent processing steps.
- **Clock-Controlled Threshold Decisions**: An input value is considered valid only on a clock edge.

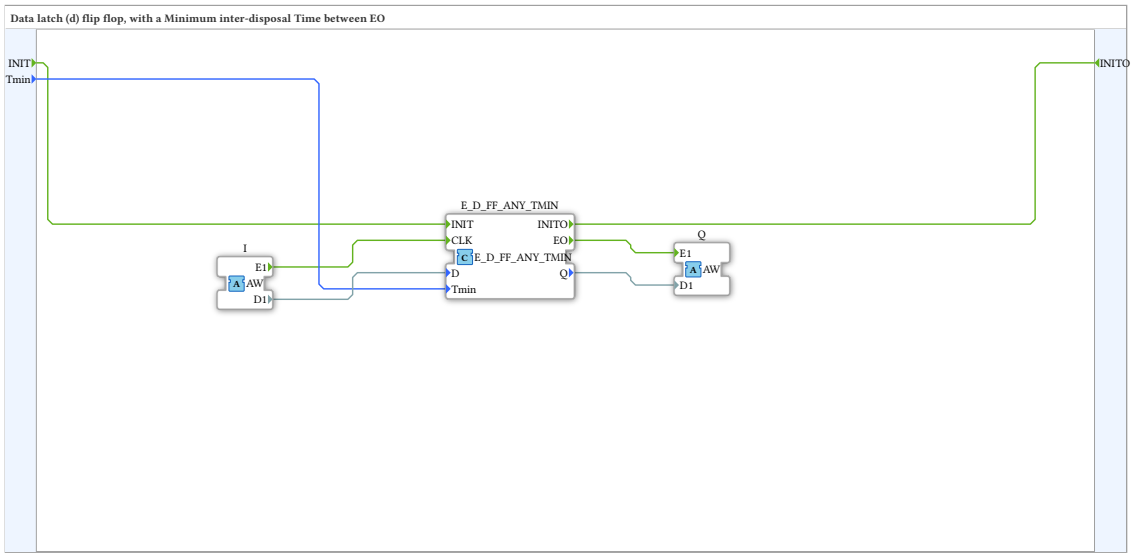
Comparison with Similar Components

Component	Property	Difference to AW_D_FF
RS Flip-Flop	Sets/resets asynchronously	AW_D_FF requires a clock signal, no separate set/reset inputs.
T Flip-Flop	Toggles on clock signal	AW_D_FF takes the value from D, no toggle function.
JK Flip-Flop	Universal (Set, Reset, Toggle)	AW_D_FF is simpler (only D input), no toggle mode.
E_D_FF_ANY	Directly usable with events/data	AW_D_FF encapsulates this function block and provides an adapter interface.

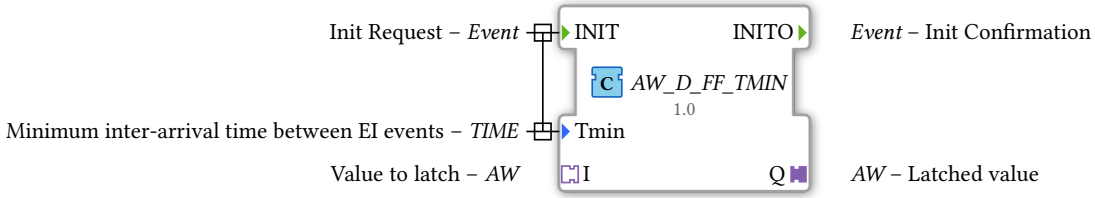
Conclusion

The **AW_D_FF** is a compact and standards-compliant D flip-flop device that, thanks to its adapter interface, is particularly flexible for use in modular IEC 61499 applications. It offers reliable clock-controlled data acquisition and is ideally suited for synchronized memory operations in industrial control systems.

AW_D_FF_TMIN



(Image of the function block – not included in the XML, therefore no image)



• – ○ • – ○ • – ○ •

Introduction

The **AW_D_FF_TMIN** is a composed IEC 61499 function block (a type of data latch flip-flop) that receives an incoming data value on an adapter, takes over upon an event, and passes it on to an output adapter with a minimal delay between two consecutive output events. It serves as a reliable buffer, extending the output events to a minimum time defined by the parameter `Tmin`.

Interface Structure

Event Inputs

Event	Type	Comment
`INIT`	EInit	Initialization request; sets the minimum inter-event time `Tmin`.

Event Outputs

Event	Type	Comment
`INITO`	EInit	Confirmation of successful initialization.

Data Inputs

Data	Type	Comment
`Tmin`	TIME	Minimum time (e.g., `T#100ms`) that must elapse between two consecutive output events (`Q.E1`).

Data Outputs

The function block (FB) has no direct data outputs; the latching value is provided via the adapter `Q`.

Adapters

Direction	Name	Type	Comment
Socket	I	adapter::types::unidirectional::AW	Value to be latched (input adapter). Contains the event and data interfaces I.E1 and I.D1.
Plug	Q	adapter::types::unidirectional::AW	Latched value (output adapter). Contains the event and data interfaces Q.E1 and Q.D1.

Functionality

The **AW_D_FF_TMIN** encapsulates the IEC 61499 standard function block E_D_FF_ANY_TMIN. The operation is as follows:

1. **Initialization**

The parameter Tmin is passed to the inner function block via the event input INIT. After successful initialization, INITO is output.

1. **Data Latching**

When an event arrives at the adapter input I.E1, the data value simultaneously present at I.D1 is adopted (edge-triggered D flip-flop).

1. **Output with Minimum Time**

The latched value is output at the adapter output Q.D1. Simultaneously, an event is generated on Q.E1 – but only if at least the time Tmin has elapsed since the last event on Q.E1. This ensures that the output events do not follow each other faster than allowed by Tmin.

1. **Further Cycles**

Each new event on I.E1 repeats the latching and the timed output.

Technical Features

- The function block is specified according to IEC 61499-1 Annex A and uses the adapter type definition adapter::types::unidirectional::AW.
- The implementation is based on the reproducible function block E_D_FF_ANY_TMIN, which provides the actual latch and timing logic.
- The minimum inter-event time Tmin is set at runtime via INIT and cannot be changed during operation.
- The function block is released under the Eclipse Public License 2.0 (EPL-2.0), version 1.0 (2026-05-15, HR Agrartechnik GmbH).

State Overview

The internal function block E_D_FF_ANY_TMIN has an implicit state machine:

- **Initialization State**: The configuration is applied after INIT; Upon success, INITO is sent.
- **Operating State**: After initialization, the function block waits for events at I.E1. With each event, the data value is latched, timer monitoring is started, and output is only released if the minimum time Tmin has elapsed since the last output event.
- **Wait State**: If an output event cannot yet be released (because the time since the last event is less than Tmin), the latched value is held until the required delay has elapsed. Only then is Q.E1 triggered.

Application Scenarios

- **Signal Debouncing**: Prevents mechanical or communication-related bounces from affecting a downstream process in rapid succession.
- **Clock Control with Minimum Pulse Duration**: Ensures that subsequent components have a minimum time between processing steps.
- **Data Stream Synchronization**: Bufferes a data value and passes it on with a defined minimum delay to smooth jitter.

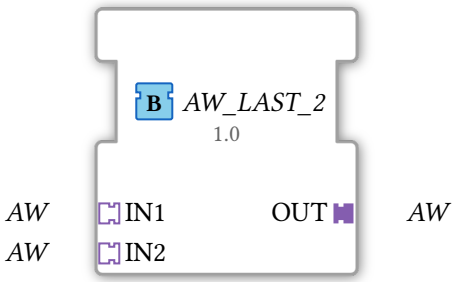
Comparison with Similar Components

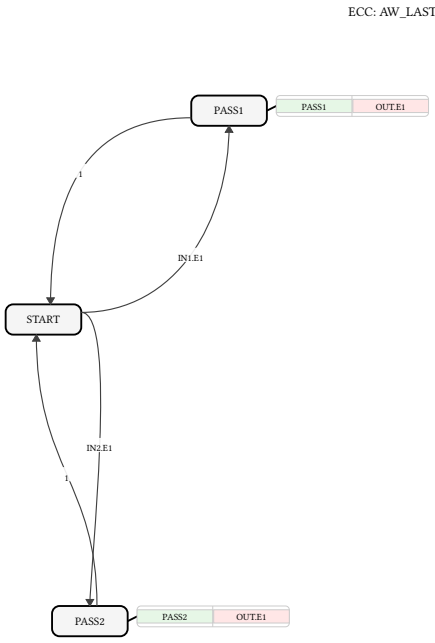
Component	Feature	Difference
`E_D_FF`	Simple D flip-flop without a time limit	No control over the maximum output frequency.
`SR_FF`	Set-Reset flip-flop	No edge-triggered data transfer.
`E_D_FF_ANY_TMIN` (direct)	Like AW_D_FF_TMIN, but a pure event/data function block	Does not have an adapter interface; the variant with an adapter increases interchangeability and encapsulation in pluggable components.

Conclusion

The **AW_D_FF_TMIN** is a maintenance-friendly, adapter-based D-latch function block that combines the acquisition of a data value with a configurable minimum time between output events. By using a standardized adapter type, it can be easily integrated into hierarchical modules and offers a robust solution for time-critical signal chains in IEC 61499 systems.

AW_LAST_2





Introduction

The AW_LAST_2 function block merges two same-type WORD adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: WORD adapter (unidirectional) - first signal source
- **IN2**: WORD adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: WORD adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

Application Scenarios

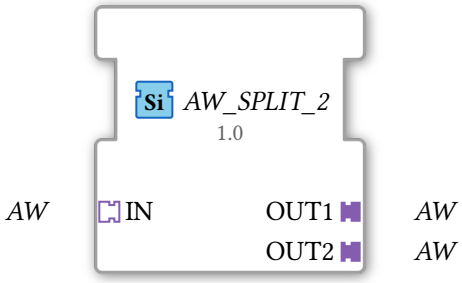
- Merging two equally-valid WORD signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same WORD value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AW_LAST_2 is the inverse of `AW_SPLIT_2`: AW_SPLIT_2 distributes 1 input to 2 outputs, AW_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use `AX_AE_MERGE` instead, which merges only the event without altering the data value.

AW_SPLIT_2

!Image of the function block (not available)



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AW_SPLIT_2** splits an incoming unidirectional **adapter (AW)** into two identical outgoing adapters. It is implemented as a generic block and is suitable for applications where a signal path or adapter connection needs to be split between two subsequent components.

Interface Structure

Event Inputs
No event input available.

Event Outputs
No event output available.

Data Inputs
No data input available.

Data Outputs
No data output available.

Adapters

Name	Direction	Type	Description
`IN`	Socket	`adapter::types::unidirectional::AW`	Input Adapter – the unidirectional adapter to be shared.
`OUT1`	Plug	`adapter::types::unidirectional::AW`	First Output – receives an identical copy of the incoming adapter.
`OUT2`	Plug	`adapter::types::unidirectional::AW`	Second Output – also receives an identical copy of the incoming adapter.

Functionality
The function block forwards the unidirectional adapter (AW) connected to **IN** unchanged to both outputs **OUT1** and **OUT2**. No data manipulation, event control, or state logic takes place – the function block functions solely as a passive distribution unit at the adapter level.

Since the adapter type is declared as generic (`GEN_AW_SPLIT`), the function block can be used with various specific implementations of the AW adapter, as long as they follow the same `unidirectional::AW` protocol.

Technical Features
- **Generic Function Block**: The function block is defined as a generic component (GenericClassName `GEN_AW_SPLIT`). It can be used in projects with different AW adapter types without requiring any modification of the function block itself.
- **No Runtime Logic**: The partitioning is purely structural – the function block (FB) has neither a state machine (ECC) nor does it process events or data. This results in no additional computational overhead at runtime.
- **License**: The function block is licensed under the **Eclipse Public License 2.0 (EPL-2.0)**.

State Overview
The function block has no internal state machine. The routing is static and occurs without time- or event-dependent control.

Application Scenarios
- **Distributing a measurement or status adapter path** to two independent processing units (e.g., a main controller and a monitoring component).
- **Building redundant processing chains**, where the same adapter stream is fed in parallel into two separate logic branches.
- **Test and diagnostic architectures** that split a data stream into both production logic and an analysis path.

Comparison with Similar Function Blocks
- **AW_MERGE_2** – combines two AW inputs into one output (counterpart to the Split function).
- **Data Split Function Blocks** (e.g., `SPLIT_INT`, `SPLIT_BOOL`) operate on data signals, not on adapters. `AW_SPLIT_2` is specifically designed for splitting adapter interfaces.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

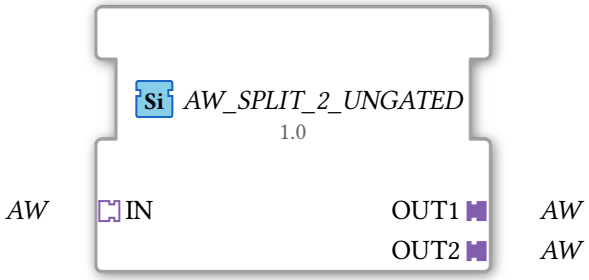
Conclusion

AW_SPLIT_2` is a simple yet essential generic function block for adapter distribution. It enables clean, reusable splitting of a unidirectional AW path without additional logic or runtime costs. Especially in modular, adapter-based control systems, it simplifies signal distribution and promotes component reusability.

AW_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

!Image of the function block (not available)



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AW_SPLIT_2_UNGATED** splits an incoming unidirectional **adapter (AW)** into two identical outgoing adapters. It is implemented as a generic block and is suitable for applications where a signal path or adapter connection needs to be split between two subsequent components.

Interface Structure

Event Inputs

No event input available.

Event Outputs

No event output available.

Data Inputs

No data input available.

Data Outputs

No data output available.

Adapters

Name	Direction	Type	Description
IN	Socket	`adapter::types::unidirectional::AW`	Input Adapter – the unidirectional adapter to be shared.

OUT1	Plug	adapter::types::unidirectional::AW	First Output – receives an identical copy of the incoming adapter.
OUT2	Plug	adapter::types::unidirectional::AW	Second Output – also receives an identical copy of the incoming adapter.

Functionality

The function block forwards the unidirectional adapter (AW) connected to **IN** unchanged to both outputs **OUT1** and **OUT2**. No data manipulation, event control, or state logic takes place – the function block functions solely as a passive distribution unit at the adapter level.

Since the adapter type is declared as generic (GEN_AW_SPLIT), the function block can be used with various specific implementations of the AW adapter, as long as they follow the same unidirectional::AW protocol.

Technical Features

- **Generic Function Block**: The function block is defined as a generic component (GenericClassName GEN_AW_SPLIT). It can be used in projects with different AW adapter types without requiring any modification of the function block itself.
- **No Runtime Logic**: The partitioning is purely structural – the function block (FB) has neither a state machine (ECC) nor does it process events or data. This results in no additional computational overhead at runtime.
- **License**: The function block is licensed under the **Eclipse Public License 2.0 (EPL-2.0)**.

State Overview

The function block has no internal state machine. The routing is static and occurs without time- or event-dependent control.

Application Scenarios

- **Distributing a measurement or status adapter path** to two independent processing units (e.g., a main controller and a monitoring component).
- **Building redundant processing chains**, where the same adapter stream is fed in parallel into two separate logic branches.
- **Test and diagnostic architectures** that split a data stream into both production logic and an analysis path.

Comparison with Similar Function Blocks

- **AW_MERGE_2** – combines two AW inputs into one output (counterpart to the Split function).
- **Data Split Function Blocks** (e.g., SPLIT_INT, SPLIT_BOOL) operate on data signals, not on adapters. AW_SPLIT_2_UNGATED is specifically designed for splitting adapter interfaces.

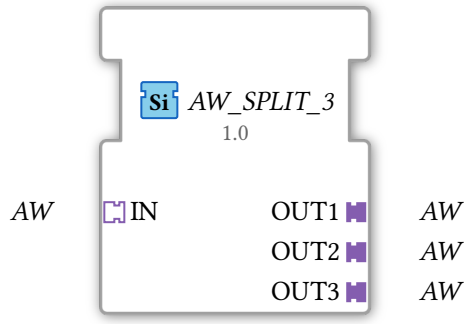
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AW_SPLIT_2_UNGATED` is a simple yet essential generic function block for adapter distribution. It enables clean, reusable splitting of a unidirectional AW path without additional logic or runtime costs. Especially in modular, adapter-based control systems, it simplifies signal distribution and promotes component reusability.

AW_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AW_SPLIT_3** serves as a generic distributor for a unidirectional adapter of type `AW`. It accepts a single adapter connection (socket `IN`) and provides it identically to three outputs (plugs `OUT1`, `OUT2`, `OUT3`). The block is designed as a generic function block (Generic FB), so the specific data type of the adapter `AW` can be defined at runtime. Its application lies in signal coupling or command forwarding to multiple downstream components.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AW`	Input adapter – Source of the signal to be distributed.
Plug	`OUT1`	`adapter::types::unidirectional::AW`	First output – Identical signal to `IN`.
Plug	`OUT2`	`adapter::types::unidirectional::AW`	Second output – identical signal to `IN`.
Plug	`OUT3`	`adapter::types::unidirectional::AW`	Third output – identical signal to `IN`.

Functionality

The module operates without its own internal state machine or data processing. It represents a passive distribution structure: The adapter connection arriving at socket `IN` is routed to the three plugs `OUT1`, `OUT2`, and `OUT3`. All outgoing adapters are logically and temporally identical to the incoming signal. No delay, buffering, or transformation occurs.

Since this is a generic function block, the adapter type `AW` must be parameterized with a specific, compatible type (e.g., `Analogwert` or `Steuerbefehl`) before commissioning. This is done by specifying the attribute `eclipse4diac::core::GenericClassName`. Type resolution occurs either at design time (IDE) or at system runtime.

Technical Features

- **Generic Approach**: The function block is implemented as a *Generic FB* (`GenericClassName = 'GEN_AW_SPLIT'`). This allows the same function block to be used for different adapter types without having to create a separate FB for each type.
- **Unidirectional**: The adapters are declared as `unidirectional`. This means that data flow is only from the socket to the plugs. Feedback from the outputs to the input is not provided.
- **No Runtime Logic**: The function block is purely structural (passive) – no events, states, or calculations are executed. This reduces resource consumption in real-time systems.
- **Use of the Eclipse 4diac Environment**: The function block relies on the mechanisms provided by 4diac for generic types and adapters.

State Overview

The function block does not have its own state machine (ECC). It is a pure connection block without any time-based behavior or state changes. Its functionality is limited to the static forwarding of the adapter.

Application Scenarios

- **Signal Fan-out in Controllers**: A measured value (e.g., temperature, pressure) should be sent in parallel to several controllers or monitoring units.
- **Distributing Control Commands**: A central command is passed on to multiple actuators or subsystems simultaneously.
- **Redundant Signal Paths**: In safety-critical applications, the same signal can be split across multiple paths to enable independent evaluation.
- **Prototype Development**: The generic nature of the function block allows for early implementation and later specification of the specific adapter type.

Comparison with Similar Function Blocks

- **AW_MERGE_3** (hypothetical): A merger of three AW inputs into one output. AW_SPLIT_3 is the logical inverse.
- **SPLIT_ALL_2**: A non-generic splitter for two outputs that uses specific data types. AW_SPLIT_3 offers greater flexibility due to its generic nature.
- **REPEATER**: A simple amplifier or repeater for adapters with only one output. AW_SPLIT_3 extends this to three parallel outputs.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AW_SPLIT_3** is a fundamental yet flexible distribution block in Eclipse 4diac. Its generic design makes it universally applicable to all unidirectional adapter types. The clear, passive structure without additional logic allows for efficient and reliable signal distribution in automation solutions. Especially when combined with the type variability and 4diac adapter mechanics, it represents a valuable tool for modular and reusable control architectures.

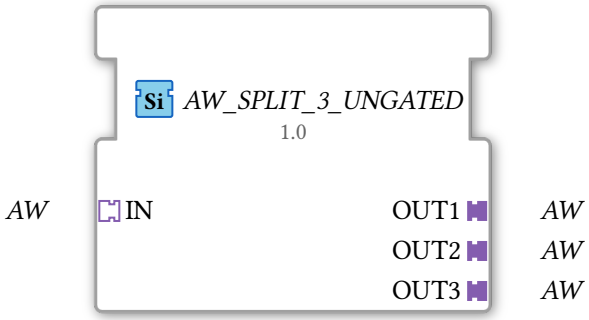
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AW_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AW_SPLIT_3_UNGATED** serves as a generic distributor for a unidirectional adapter of type `AW`. It accepts a single adapter connection (socket `IN`) and provides it identically to three outputs (plugs `OUT1`, `OUT2`, `OUT3`). The block is designed as a generic function block (Generic FB), so the specific data type of the adapter `AW` can be defined at runtime. Its application lies in signal coupling or command forwarding to multiple downstream components.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AW`	Input adapter – Source of the signal to be distributed.
Plug	`OUT1`	`adapter::types::unidirectional::AW`	First output – Identical signal to `IN`.
Plug	`OUT2`	`adapter::types::unidirectional::AW`	Second output – identical signal to `IN`.
Plug	`OUT3`	`adapter::types::unidirectional::AW`	Third output – identical signal to `IN`.

Functionality

The module operates without its own internal state machine or data processing. It represents a passive distribution structure: The adapter connection arriving at socket `IN` is routed to the three plugs `OUT1`, `OUT2`, and `OUT3`. All outgoing adapters are logically and temporally identical to the incoming signal. No delay, buffering, or transformation occurs.

Since this is a generic function block, the adapter type `AW` must be parameterized with a specific, compatible type (e.g., `Analogwert` or `Steuerbefehl`) before commissioning. This is done by specifying the attribute `eclipse4diac::core::GenericClassName`. Type resolution occurs either at design time (IDE) or at system runtime.

Technical Features

- **Generic Approach**: The function block is implemented as a *Generic FB* (`GenericClassName = 'GEN_AW_SPLIT'`). This allows the same function block to be used for different adapter types without having to create a separate FB for each type.
- **Unidirectional**: The adapters are declared as `unidirectional`. This means that data flow is only from the socket to the plugs. Feedback from the outputs to the input is not provided.
- **No Runtime Logic**: The function block is purely structural (passive) – no events, states, or calculations are executed. This reduces resource consumption in real-time systems.
- **Use of the Eclipse 4diac Environment**: The function block relies on the mechanisms provided by 4diac for generic types and adapters.

State Overview

The function block does not have its own state machine (ECC). It is a pure connection block without any time-based behavior or state changes. Its functionality is limited to the static forwarding of the adapter.

Application Scenarios

- **Signal Fan-out in Controllers**: A measured value (e.g., temperature, pressure) should be sent in parallel to several controllers or monitoring units.
- **Distributing Control Commands**: A central command is passed on to multiple actuators or subsystems simultaneously.
- **Redundant Signal Paths**: In safety-critical applications, the same signal can be split across multiple paths to enable independent evaluation.
- **Prototype Development**: The generic nature of the function block allows for early implementation and later specification of the specific adapter type.

Comparison with Similar Function Blocks

- **AW_MERGE_3** (hypothetical): A merger of three AW inputs into one output. AW_SPLIT_3_UNGATED is the logical inverse.
- **SPLIT_ALL_2**: A non-generic splitter for two outputs that uses specific data types. AW_SPLIT_3_UNGATED offers greater flexibility due to its generic nature.
- **REPEATER**: A simple amplifier or repeater for adapters with only one output. AW_SPLIT_3_UNGATED extends this to three parallel outputs.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

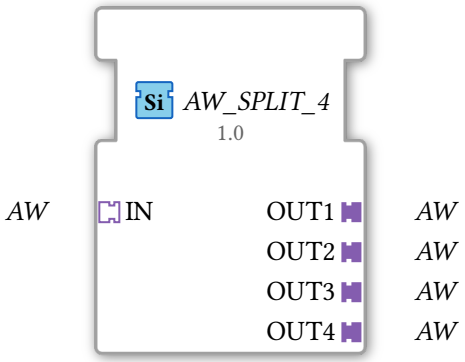
The **AW_SPLIT_3_UNGATED** is a fundamental yet flexible distribution block in Eclipse 4diac. Its generic design makes it universally applicable to all unidirectional adapter types. The clear, passive structure without additional logic allows for efficient and reliable signal distribution in automation solutions. Especially when combined with the type variability and 4diac adapter mechanics, it represents a valuable tool for modular and reusable control architectures.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AW_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AW_SPLIT_4` distributes an incoming AW adapter (unidirectional) to four identical AW output adapters. The block is designed generically and can be used with different AW types.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None (data transmission occurs exclusively via the adapter socket).

Data Outputs

None (output occurs exclusively via the adapter plugs).

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AW`	`IN`	Socket	Input for the AW data stream to be distributed
`adapter::types::unidirectional::AW`	`OUT1`	Plug	First output (copy of the input)
`adapter::types::unidirectional::AW`	`OUT2`	Plug	Second output
`adapter::types::unidirectional::AW`	`OUT3`	Plug	Third output
`adapter::types::unidirectional::AW`	`OUT4`	Plug	Fourth Output

Functionality

This function block acts as a passive distributor: Every value (or data packet) arriving via socket `IN` is forwarded unchanged to all four plugs `OUT1` to `OUT4`. No processing, buffering, or time synchronization takes place. The entire data flow is uneventful and purely adapter-based.

Technical Features

- **Generic Function Block**: The function block is declared with the generic name `GEN_AW_SPLIT`. This allows for late binding to specific AW adapter types (e.g., `AT_AW`, `CT_AW`, etc.) at compile or configuration time.
- **Unidirectional Communication**: The adapters used are of type `unidirectional::AW`, meaning data flows only in one direction. There are no return channels or acknowledgments.

- **No Dynamics**: The function block (FB) has no events, states, or time-dependent logic. It is purely combinatorial.

State Overview

Not applicable – the FB does not contain a state machine (ECC) and operates entirely in a data-driven manner.

Application Scenarios

- Distribution of a single sensor or actuator signal path across multiple parallel evaluations.
- Preparation of a data source for different consumers in a control application.
- Generation of test or simulation points by multiplying an AW data stream.

Comparison with similar function blocks

Unlike data- or event-based split function blocks (e.g., `SPLIT`, `F_SPLIT`), `AW_SPLIT_4` operates exclusively at the adapter level. The advantage lies in its loose coupling and reusability across different AW types. The disadvantage: Only AW data can be distributed, not events or general values.

Change Detection

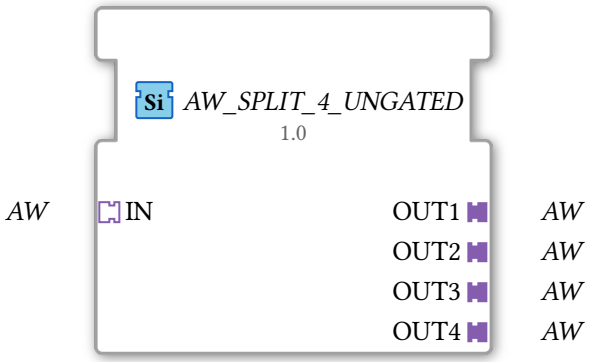
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AW_SPLIT_4` function block is a simple, generic distributor for unidirectional AW adapters. It is ideally suited to split a single AW signal into up to four outputs and, thanks to its generic nature, integrates flexibly into various IEC 61499 applications.

AW_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AW_SPLIT_4_UNGATED` distributes an incoming AW adapter (unidirectional) to four identical AW output adapters. The block is designed generically and can be used with different AW types.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None (data transmission occurs exclusively via the adapter socket).

Data Outputs

None (output occurs exclusively via the adapter plugs).

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AW`	`IN`	Socket	Input for the AW data stream to be distributed
`adapter::types::unidirectional::AW`	`OUT1`	Plug	First output (copy of the input)
`adapter::types::unidirectional::AW`	`OUT2`	Plug	Second output
`adapter::types::unidirectional::AW`	`OUT3`	Plug	Third output
`adapter::types::unidirectional::AW`	`OUT4`	Plug	Fourth Output

Functionality

This function block acts as a passive distributor: Every value (or data packet) arriving via socket `IN` is forwarded unchanged to all four plugs `OUT1` to `OUT4`. No processing, buffering, or time synchronization takes place. The entire data flow is uneventful and purely adapter-based.

Technical Features

- **Generic Function Block**: The function block is declared with the generic name `GEN_AW_SPLIT`. This allows for late binding to specific AW adapter types (e.g., `AT_AW`, `CT_AW`, etc.) at compile or configuration time.
- **Unidirectional Communication**: The adapters used are of type `unidirectional::AW`, meaning data flows only in one direction. There are no return channels or acknowledgments.
- **No Dynamics**: The function block (FB) has no events, states, or time-dependent logic. It is purely combinatorial.

State Overview

Not applicable – the FB does not contain a state machine (ECC) and operates entirely in a data-driven manner.

Application Scenarios

- Distribution of a single sensor or actuator signal path across multiple parallel evaluations.
- Preparation of a data source for different consumers in a control application.
- Generation of test or simulation points by multiplying an AW data stream.

Comparison with similar function blocks

Unlike data- or event-based split function blocks (e.g., `SPLIT`, `F_SPLIT`), `AW_SPLIT_4_UNGATED` operates exclusively at the adapter level. The advantage lies in its loose coupling and reusability across different AW types. The disadvantage: Only AW data can be distributed, not events or general values.

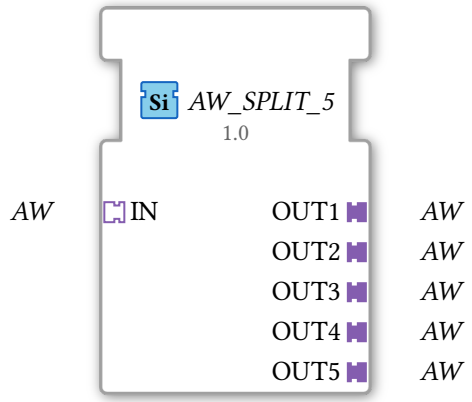
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AW_SPLIT_4_UNGATED` function block is a simple, generic distributor for unidirectional AW adapters. It is ideally suited to split a single AW signal into up to four outputs and, thanks to its generic nature, integrates flexibly into various IEC 61499 applications.

AW_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AW_SPLIT_5** is used to distribute an incoming AW adapter (“actuator/paths”) to five separate, identical AW outputs. It is designed as a generic function block and can be used in control architectures where a single signal or data connection needs to be split across multiple parallel paths.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapter

- **IN** (Socket) – Type: `adapter::types::unidirectional::AW`

Input interface for the original AW data stream.

- **OUT1** to **OUT5** (Plugs) – Type: `adapter::types::unidirectional::AW`

Five parallel output interfaces, each providing the same signal as the input.

Functionality

The function block passively forwards the incoming AW adapter to all five outputs. There is no active logic or state machine – the distribution is achieved structurally through the internal wiring of the function block. Changes to the input are immediately reflected at all outputs.

Technical Features

- **Generic Function Block** – The function block is declared as a generic type (`GEN_AW_SPLIT`) and can be reused in different contexts.
- **No Event Control** – Signal transmission occurs continuously without event-driven activation.
- **Unidirectional Adapters** – The adapter type used, `AW`, is unidirectional, meaning data flows only from the input to the outputs. Feedback from the outputs is not provided.

State Overview

The function block (FB) does not have an internal state machine (ECC). There are no defined states or state transitions – the functionality is purely data flow-oriented.

Application Scenarios

- **Distributing a sensor signal** to multiple downstream control units.
- **Parallel control** of multiple actuators that are to receive the same setpoint or command.
- **Test Setups** – Recording a signal or simultaneously forwarding it to various simulation and analysis components.

Comparison with Similar Function Blocks

- **AW_SPLIT_N** – Another generic split FB that can be extended to a variable number of outputs. `AW_SPLIT_5` is a specialized variant with five fixed outputs.
- **Pure Data Splitters** – These operate on data types like `INT` or `BOOL` and use different interface types. `AW_SPLIT_5` is specifically designed for the AW adapter.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AW_SPLIT_5` is a simple yet useful component for splitting an AW adapter signal into five parallel paths. Its generic design allows for flexible use in various control projects within the 4diac IDE. The absence of an event and data interface reduces complexity to the essentials – pure signal duplication.

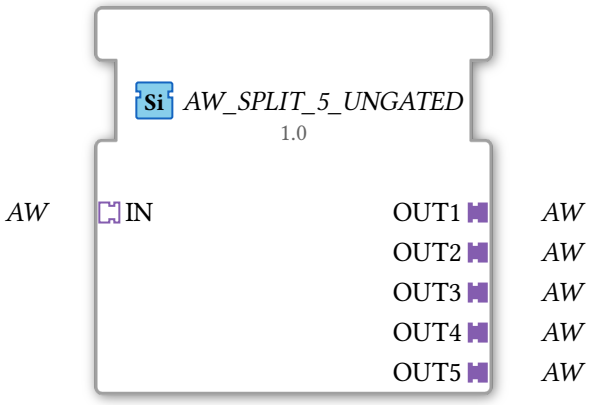
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AW_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AW_SPLIT_5_UNGATED** is used to distribute an incoming AW adapter ("actuator/paths") to five separate, identical AW outputs. It is designed as a generic function block and can be used in control architectures where a single signal or data connection needs to be split across multiple parallel paths.

Interface Structure

Event Inputs
None

Event Outputs
None

Data Inputs
None

Data Outputs
None

Adapter
- **IN** (Socket) – Type: `adapter::types::unidirectional::AW`

Input interface for the original AW data stream.

- **OUT1** to **OUT5** (Plugs) – Type: `adapter::types::unidirectional::AW`

Five parallel output interfaces, each providing the same signal as the input.

Functionality
The function block passively forwards the incoming AW adapter to all five outputs. There is no active logic or state machine – the distribution is achieved structurally through the internal wiring of the function block. Changes to the input are immediately reflected at all outputs.

Technical Features
- **Generic Function Block** – The function block is declared as a generic type (`GEN_AW_SPLIT`) and can be reused in different contexts.
- **No Event Control** – Signal transmission occurs continuously without event-driven activation.
- **Unidirectional Adapters** – The adapter type used, `AW`, is unidirectional, meaning data flows only from the input to the outputs. Feedback from the outputs is not provided.

State Overview
The function block (FB) does not have an internal state machine (ECC). There are no defined states or state transitions – the functionality is purely data flow-oriented.

Application Scenarios
- **Distributing a sensor signal** to multiple downstream control units.
- **Parallel control** of multiple actuators that are to receive the same setpoint or command.
- **Test Setups** – Recording a signal or simultaneously forwarding it to various simulation and analysis components.

Comparison with Similar Function Blocks
- **AW_SPLIT_N** – Another generic split FB that can be extended to a variable number of outputs. `AW_SPLIT_5_UNGATED` is a specialized variant with five fixed outputs.
- **Pure Data Splitters** – These operate on data types like `INT` or `BOOL` and use different interface types. `AW_SPLIT_5_UNGATED` is specifically designed for the AW adapter.

Change Detection
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

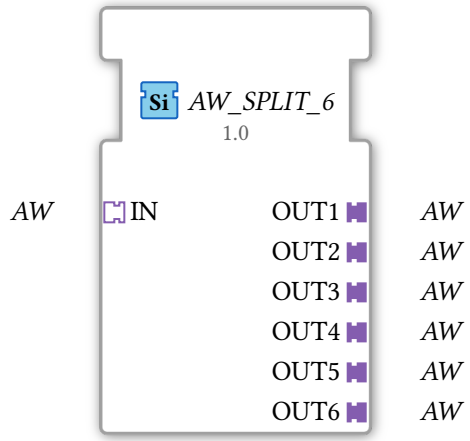
Conclusion
The `AW_SPLIT_5_UNGATED` is a simple yet useful component for splitting an AW adapter signal into five parallel paths. Its generic design allows for flexible use in various control projects within the 4diac IDE. The absence of an event and data interface reduces complexity to the essentials – pure signal duplication.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AW_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The **AW_SPLIT_6** function block is used to split an incoming unidirectional **AW** adapter into six identical output adapters. It is designed as a generic function block used in the 4diac IDE for distributing AW signals. The function block is intended for applications where a signal value—e.g., an analog or digital value—is required multiple times in parallel without the need for manual duplication of the connections.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AW`	IN	Socket	Input adapter for the AW signal to be distributed
`adapter::types::unidirectional::AW`	OUT1 – OUT6	Plugs	Six output adapters, each providing an identical copy of the input signal

Functionality

The module receives a unidirectional **AW** signal via the **IN** socket. This signal is internally duplicated to all six output plugs (**OUT1** to **OUT6**). The signal is passed through without delay or data manipulation – the value of the incoming adapter is made directly available at each of the six outputs. The module has no state logic or event control of its own; it operates purely passively and simply passes the signal through.

Technical Features

- **Generic Type**: The function block is implemented as a generic function block (`GEN_AW_SPLIT`), allowing for flexible reuse in different projects.
- **Type Hash**: The type hash property is empty, meaning that no automatic type identity check is performed during instantiation.
- **No Event or Data Ports**: All signal transmission occurs exclusively via adapters. Events are not processed; therefore, the function block is only suitable for pure data transmission without timing.

State Overview

The function block has no internal states. It is a pure split function block without sequential logic. Its behavior is deterministic and always identical: The input signal is copied to all outputs.

Application Scenarios

- **Distribution of an analog measured value** to multiple consumers (e.g., displays, controllers, recording systems)
- **Splitting a digital control signal** to parallel actuator or sensor interfaces
- **Star-shaped signal distribution** in modular control architectures where one source supplies multiple sinks

Comparison with Similar Function Blocks

Compared to an **AW_SPLIT_2** or **AW_SPLIT_4**, this function block offers a higher number of outputs (6). This makes it particularly suitable for systems with many parallel receivers. Alternatively, several smaller Split function blocks could be cascaded, but this can negatively impact clarity and performance. The **AW_SPLIT_6** is optimized as a complete solution for this requirement.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

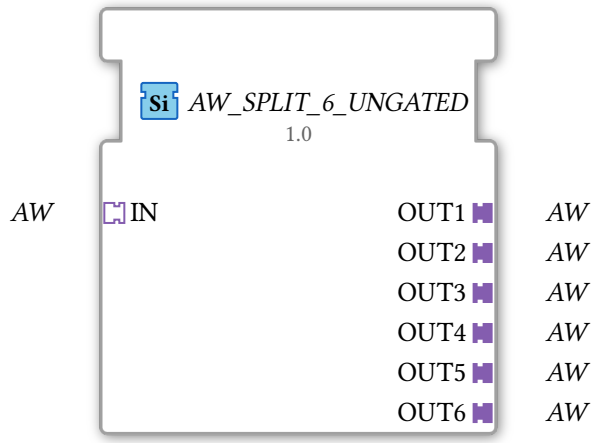
The **AW_SPLIT_6** is a simple yet useful function block for multiplying a unidirectional AW signal to six outputs. Due to its generic nature and adapter-based interface, it integrates seamlessly into a 4diac environment. It is recommended for all applications where a single signal is needed multiple times without any detours.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AW_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The **AW_SPLIT_6_UNGATED** function block is used to split an incoming unidirectional **AW** adapter into six identical output adapters. It is designed as a generic function block used in the 4diac IDE for distributing AW signals. The function block is intended for applications where a signal value—e.g., an analog or digital value—is required multiple times in parallel without the need for manual duplication of the connections.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AW`	IN	Socket	Input adapter for the AW signal to be distributed
`adapter::types::unidirectional::AW`	OUT1 – OUT6	Plugs	Six output adapters, each providing an identical copy of the input signal

Functionality

The module receives a unidirectional **AW** signal via the **IN** socket. This signal is internally duplicated to all six output plugs (**OUT1** to **OUT6**). The signal is passed through without delay or data manipulation – the value of the incoming adapter is made directly available at each of the six outputs. The module has no state logic or event control of its own; it operates purely passively and simply passes the signal through.

Technical Features

- **Generic Type**: The function block is implemented as a generic function block (`GEN_AW_SPLIT`), allowing for flexible reuse in different projects.

- **Type Hash**: The type hash property is empty, meaning that no automatic type identity check is performed during instantiation.
- **No Event or Data Ports**: All signal transmission occurs exclusively via adapters. Events are not processed; therefore, the function block is only suitable for pure data transmission without timing.

State Overview

The function block has no internal states. It is a pure split function block without sequential logic. Its behavior is deterministic and always identical: The input signal is copied to all outputs.

Application Scenarios

- **Distribution of an analog measured value** to multiple consumers (e.g., displays, controllers, recording systems)
- **Splitting a digital control signal** to parallel actuator or sensor interfaces
- **Star-shaped signal distribution** in modular control architectures where one source supplies multiple sinks

Comparison with Similar Function Blocks

Compared to an **AW_SPLIT_2** or **AW_SPLIT_4**, this function block offers a higher number of outputs (6). This makes it particularly suitable for systems with many parallel receivers. Alternatively, several smaller Split function blocks could be cascaded, but this can negatively impact clarity and performance. The **AW_SPLIT_6_UNGATED** is optimized as a complete solution for this requirement.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

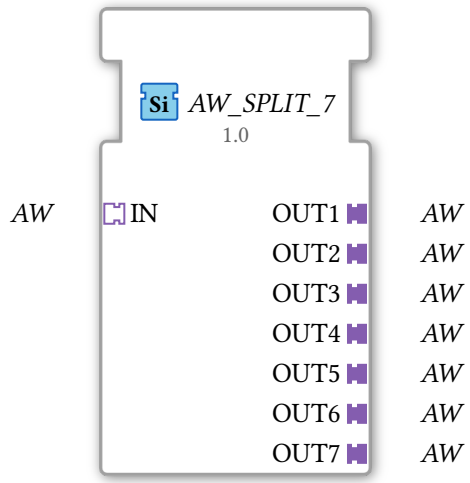
Conclusion

The **AW_SPLIT_6_UNGATED** is a simple yet useful function block for multiplying a unidirectional AW signal to six outputs. Due to its generic nature and adapter-based interface, it integrates seamlessly into a 4diac environment. It is recommended for all applications where a single signal is needed multiple times without any detours.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AW_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AW_SPLIT_7` is used to distribute a single incoming AW adapter (unidirectional) to seven identical AW adapter outputs. It is designed as a generic function block and enables the simple distribution of adapter connections without data processing.

Interface Structure

Event Inputs

- None

Event Outputs

- None

Data Inputs

- None

Data Outputs

- None

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AW`	IN	Socket (Input)	Incoming AW adapter, which is distributed to the outputs.
`adapter::types::unidirectional::AW`	OUT1 ... OUT7	Plug (Output)	Seven outgoing adapters, each providing the same data as the input.

Functionality

This module does not have its own event or data ports. It functions purely as an adapter duplication: The incoming AW adapter (`IN`) is internally passed through to all seven output adapters (`OUT1` to `OUT7`) without any processing. Each module connected to one of the outputs thus receives exactly the same instance of the AW adapter.

The implementation is generic (Generic FB), so the actual adapter type can be dynamically determined by parameters. This makes the function block suitable for any unidirectional AW interface.

Technical Features

- **Generic Type**: The function block uses the generic function block mechanism to determine the specific adapter type at runtime or during compilation. The placeholder `GEN_AW_SPLIT` is replaced by the actual adapter.
- **Pure Adapter Distribution**: No data copying or serialization takes place – the references to the AW adapter are passed on.
- **No Event Control Required**: The function block operates asynchronously and without time dependencies, as adapters are inherently data-specific.

State Overview

The `AW_SPLIT_7` function block does not have an internal state machine. Its behavior is deterministic and consists solely of passing through the adapter. A state machine is therefore not required.

Application Scenarios

- **Distributing sensor adapters** to multiple evaluation or control units in an automation axis.
- **Splitting a bus signal** into multiple parallel connections (e.g., for redundant monitoring).
- **Cascading adapters** in an IEC 61499 system when multiple function blocks need to access the same data stream.

Comparison with Similar Function Blocks

- **SPLIT Function Blocks with a Limited Number of Outputs**: Other split function blocks (e.g., `SPLIT_2`, `SPLIT_4`) distribute adapters across fewer channels – here, exactly seven outputs are provided.

- **Adapter Multiplexers**: Unlike multiplexers that select between multiple inputs, the `AW_SPLIT_7` distributes a single input without selection.
- **Data Split Function Blocks**: If the function block had data and event ports, it would have to copy data. Adapter-based splits are more efficient because only references are passed on.

Change Detection

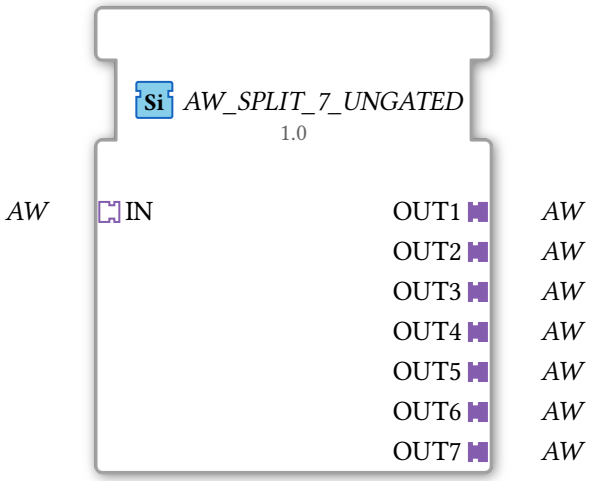
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AW_SPLIT_7` is a simple yet useful generic function block for duplicating a unidirectional AW adapter to seven outputs. Its strengths lie in its flexible type adaptation and efficient, event-free routing. It is particularly suitable for modular automation systems where an adapter needs to be used multiple times.

AW_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AW_SPLIT_7_UNGATED` is used to distribute a single incoming AW adapter (unidirectional) to seven identical AW adapter outputs. It is designed as a generic function block and enables the simple distribution of adapter connections without data processing.

Interface Structure

Event Inputs

- None

Event Outputs

- None

Data Inputs

- None

Data Outputs

- None

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AW`	IN	Socket (Input)	Incoming AW adapter, which is distributed to the outputs.
`adapter::types::unidirectional::AW`	OUT1 ... OUT7	Plug (Output)	Seven outgoing adapters, each providing the same data as the input.

Functionality

This module does not have its own event or data ports. It functions purely as an adapter duplication: The incoming AW adapter (`IN`) is internally passed through to all seven output adapters (`OUT1` to `OUT7`) without any processing. Each module connected to one of the outputs thus receives exactly the same instance of the AW adapter.

The implementation is generic (Generic FB), so the actual adapter type can be dynamically determined by parameters. This makes the function block suitable for any unidirectional AW interface.

Technical Features

- **Generic Type**: The function block uses the generic function block mechanism to determine the specific adapter type at runtime or during compilation. The placeholder `GEN_AW_SPLIT` is replaced by the actual adapter.
- **Pure Adapter Distribution**: No data copying or serialization takes place – the references to the AW adapter are passed on.
- **No Event Control Required**: The function block operates asynchronously and without time dependencies, as adapters are inherently data-specific.

State Overview

The `AW_SPLIT_7_UNGATED` function block does not have an internal state machine. Its behavior is deterministic and consists solely of passing through the adapter. A state machine is therefore not required.

Application Scenarios

- **Distributing sensor adapters** to multiple evaluation or control units in an automation axis.
- **Splitting a bus signal** into multiple parallel connections (e.g., for redundant monitoring).
- **Cascading adapters** in an IEC 61499 system when multiple function blocks need to access the same data stream.

Comparison with Similar Function Blocks

- **SPLIT Function Blocks with a Limited Number of Outputs**: Other split function blocks (e.g., `SPLIT_2`, `SPLIT_4`) distribute adapters across fewer channels – here, exactly seven outputs are provided.
- **Adapter Multiplexers**: Unlike multiplexers that select between multiple inputs, the `AW_SPLIT_7_UNGATED` distributes a single input without selection.
- **Data Split Function Blocks**: If the function block had data and event ports, it would have to copy data. Adapter-based splits are more efficient because only references are passed on.

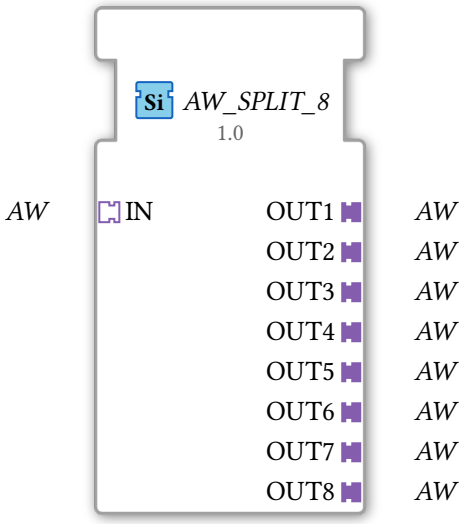
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AW_SPLIT_7_UNGATED` is a simple yet useful generic function block for duplicating a unidirectional AW adapter to seven outputs. Its strengths lie in its flexible type adaptation and efficient, event-free routing. It is particularly suitable for modular automation systems where an adapter needs to be used multiple times.

AW_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AW_SPLIT_8** is a generic block that splits an incoming unidirectional AW adapter signal into eight identical output signals. The block is implemented as a generic FB and can be assigned a specific type name and type check code at runtime. It serves as a 1:8 distributor for AW data streams and is typically used in automation technology when a signal needs to be forwarded to multiple sinks simultaneously.

Interface Structure

Event Inputs

No event inputs are available. Communication occurs exclusively via the adapter interfaces.

Event Outputs

No event outputs are available. Data is passed via the adapter plugs.

Data Inputs

No data inputs are available. All input data is provided via the socket adapter `IN`.

Data Outputs

No data outputs are available. Output data is provided via the plug adapters `OUT1` to `OUT8`.

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AW`	`IN`	Socket (Input)	Source adapter that provides the signal to be split
`adapter::types::unidirectional::AW`	`OUT1`	Plug (Output)	First output, identical to the IN signal
adapter::types::unidirectional::AW`	`OUT2`	Plug (Output)	Second output
adapter::types::unidirectional::AW`	`OUT3`	Plug (Output)	Third output
adapter::types::unidirectional::AW`	`OUT4`	Plug (Output)	Fourth output
adapter::types::unidirectional::AW`	`OUT5`	Plug (Output)	Fifth output
adapter::types::unidirectional::AW`	`OUT6`	Plug (Output)	Sixth Output

adapter::types::unidirectional::AW	OUT7	Plug (Output)	Seventh Output
adapter::types::unidirectional::AW	OUT8	Plug (Output)	Eighth Output

Functionality

The module behaves like a passive splitter. As soon as a valid AW adapter signal is present at socket `IN` (e.g., due to an event from the adapter protocol), this signal is simultaneously passed on to all eight plug adapters (`OUT1`–`OUT8`). No data modification, buffering, or delay occurs. The function block is event-driven in accordance with the underlying adapter protocol: Data transmission is triggered by the adapter mechanism of IEC 61499-2.

The outputs always provide the same data content as the input. If the input does not contain valid data, the outputs will also be silent.

Technical Features

- **Generic Function Block (FB):** The function block is declared as generic and carries the attributes `eclipse4diac::core::GenericClassName` (default: `'GEN_AW_SPLIT'`) and `eclipse4diac::core::TypeHash` (default: `''`). This allows for late binding to a specific function block type, for example, to create more type-safe instances.
- **No Own Events or Variables:** The function block has no explicit event inputs/outputs and no data inputs/outputs in the traditional sense. All communication is handled via the adapter interfaces, which reduces complexity and increases reusability.
- **Unidirectional Adapters:** Only unidirectional AW adapters are used. Feedback from the outputs to the input is not provided.

State Overview

The function block does not have its own state machine (e.g., ECC – Execution Control Chart). Its behavior is entirely determined by the behavior of the adapter type used, `adapter::types::unidirectional::AW`. In the simplest case, there are two implicit states:

- **Idle:** No valid signal at the input. All outputs are inactive.
- **Active:** A valid signal is present at the input and is replicated to all outputs.

A more detailed state description can be derived from the adapter definitions.

Application Scenarios

- **Parallel Supply of Multiple Consumers:** A sensor or control signal (e.g., speed setpoint) should be distributed simultaneously to multiple actuators or subsystems without requiring the programming of a separate distribution module.
- **Test and Simulation Environments:** A single test signal can be connected to multiple analysis or logging channels.
- **Data Multiplication in Hierarchical Control Systems:** In modular systems, a central control data stream can be distributed to multiple decentralized units (e.g., field devices).

Comparison with Similar Modules

- **AW_SPLIT_2, AW_SPLIT_4:** Analogous splitters with fewer outputs (2 or 4, respectively). The number of outputs is the only difference. For applications requiring exactly 8 outputs, `AW_SPLIT_8` is the appropriate choice.
- **General Splitters (e.g., for other adapter types):** Splitters exist for various adapter data types (e.g., `BOOL_SPLIT`, `INT_SPLIT`) that have an analog structure but different adapter interfaces.
- **Multicast Blocks:** More complex blocks can additionally offer filtering, prioritization, or buffering functions; `AW_SPLIT_8` is intentionally kept minimalist to avoid unwanted side effects.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AW_SPLIT_8` block is a simple yet useful generic 1:8 splitter for unidirectional AW adapters. It fulfills the basic requirement of distributing an incoming signal to eight outputs without delay or modification. Its

generic design allows it to be used in various contexts where type-safe replication of adapter data streams is required. Strict adherence to the IEC 61499-2 standard and the use of adapters facilitate integration into existing 4diac projects.

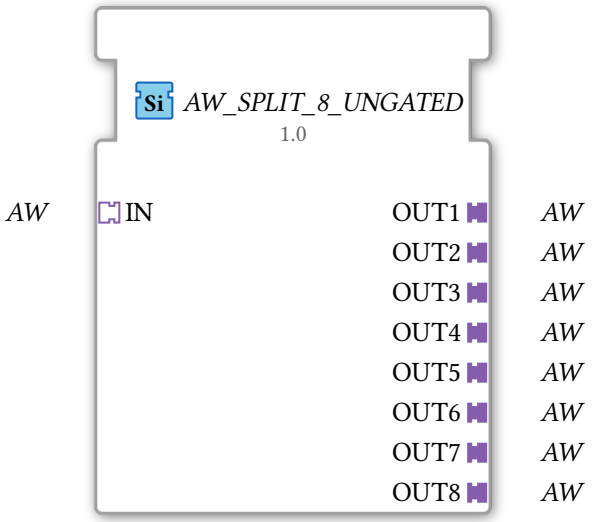
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AW_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AW_SPLIT_8_UNGATED** is a generic block that splits an incoming unidirectional AW adapter signal into eight identical output signals. The block is implemented as a generic FB and can be assigned a specific type name and type check code at runtime. It serves as a 1:8 distributor for AW data streams and is typically used in automation technology when a signal needs to be forwarded to multiple sinks simultaneously.

Interface Structure

Event Inputs

No event inputs are available. Communication occurs exclusively via the adapter interfaces.

Event Outputs

No event outputs are available. Data is passed via the adapter plugs.

Data Inputs

No data inputs are available. All input data is provided via the socket adapter `IN`.

Data Outputs

No data outputs are available. Output data is provided via the plug adapters `OUT1` to `OUT8`.

Adapters

Type	Name	Direction	Description
adapter::types::unidirectional::AW	IN	Socket (Input)	Source adapter that provides the signal to be split
adapter::types::unidirectional::AW	OUT1	Plug (Output)	First output, identical to the IN signal
adapter::types::unidirectional::AW`	OUT2	Plug (Output)	Second output
adapter::types::unidirectional::AW`	OUT3	Plug (Output)	Third output
adapter::types::unidirectional::AW`	OUT4	Plug (Output)	Fourth output
adapter::types::unidirectional::AW`	OUT5	Plug (Output)	Fifth output
adapter::types::unidirectional::AW`	OUT6	Plug (Output)	Sixth Output
adapter::types::unidirectional::AW	OUT7	Plug (Output)	Seventh Output
adapter::types::unidirectional::AW	OUT8	Plug (Output)	Eighth Output

Functionality

The module behaves like a passive splitter. As soon as a valid AW adapter signal is present at socket `IN` (e.g., due to an event from the adapter protocol), this signal is simultaneously passed on to all eight plug adapters (`OUT1`–`OUT8`). No data modification, buffering, or delay occurs. The function block is event-driven in accordance with the underlying adapter protocol: Data transmission is triggered by the adapter mechanism of IEC 61499-2.

The outputs always provide the same data content as the input. If the input does not contain valid data, the outputs will also be silent.

Technical Features

- **Generic Function Block (FB):** The function block is declared as generic and carries the attributes `eclipse4diac::core::GenericClassName` (default: `'GEN_AW_SPLIT'`) and `eclipse4diac::core::TypeHash` (default: `''`). This allows for late binding to a specific function block type, for example, to create more type-safe instances.
- **No Own Events or Variables:** The function block has no explicit event inputs/outputs and no data inputs/outputs in the traditional sense. All communication is handled via the adapter interfaces, which reduces complexity and increases reusability.
- **Unidirectional Adapters:** Only unidirectional AW adapters are used. Feedback from the outputs to the input is not provided.

State Overview

The function block does not have its own state machine (e.g., ECC – Execution Control Chart). Its behavior is entirely determined by the behavior of the adapter type used, `adapter::types::unidirectional::AW`. In the simplest case, there are two implicit states:

- **Idle:** No valid signal at the input. All outputs are inactive.
- **Active:** A valid signal is present at the input and is replicated to all outputs.

A more detailed state description can be derived from the adapter definitions.

Application Scenarios

- **Parallel Supply of Multiple Consumers:** A sensor or control signal (e.g., speed setpoint) should be distributed simultaneously to multiple actuators or subsystems without requiring the programming of a separate distribution module.
- **Test and Simulation Environments:** A single test signal can be connected to multiple analysis or logging channels.
- **Data Multiplication in Hierarchical Control Systems:** In modular systems, a central control data stream can be distributed to multiple decentralized units (e.g., field devices).

Comparison with Similar Modules

- **AW_SPLIT_2, AW_SPLIT_4:** Analogous splitters with fewer outputs (2 or 4, respectively). The number of outputs is the only difference. For applications requiring exactly 8 outputs, `AW_SPLIT_8_UNGATED` is the appropriate choice.
- **General Splitters (e.g., for other adapter types):** Splitters exist for various adapter data types (e.g., `BOOL_SPLIT`, `INT_SPLIT`) that have an analog structure but different adapter interfaces.
- **Multicast Blocks:** More complex blocks can additionally offer filtering, prioritization, or buffering functions; `AW_SPLIT_8_UNGATED` is intentionally kept minimalist to avoid unwanted side effects.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

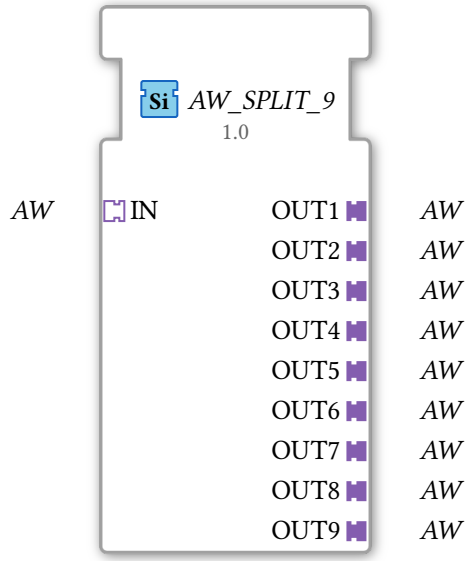
The `AW_SPLIT_8_UNGATED` block is a simple yet useful generic 1:8 splitter for unidirectional AW adapters. It fulfills the basic requirement of distributing an incoming signal to eight outputs without delay or modification. Its generic design allows it to be used in various contexts where type-safe replication of adapter data streams is required. Strict adherence to the IEC 61499-2 standard and the use of adapters facilitate integration into existing 4diac projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AW_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AW_SPLIT_9` serves as a generic distributor for one incoming adapter of type `AW` to nine identically configured output adapters. It enables the multiple use of a single adapter signal without duplicating the original data source. The designation “SPLIT 1 AW into 9 AW” describes the core function: One AW input is distributed to nine AW outputs.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Name	Direction	Description
adapter::types::unidirectional::AW	IN	Socket (Input)	Incoming AW adapter that is distributed to all outputs.
adapter::types::unidirectional::AW	OUT1	Plug (Output)	First outgoing AW adapter.
adapter::types::unidirectional::AW	OUT2	Plug (Output)	Second outgoing AW adapter.
adapter::types::unidirectional::AW	OUT3	Plug (Output)	Third outgoing AW adapter.
adapter::types::unidirectional::AW`	OUT4	Plug (Output)	Fourth outgoing AW adapter.
adapter::types::unidirectional::AW`	OUT5	Plug (Output)	Fifth outgoing AW adapter.
adapter::types::unidirectional::AW`	OUT6	Plug (Output)	Sixth outgoing AW adapter.
adapter::types::unidirectional::AW`	OUT7	Plug (Output)	Seventh outgoing AW adapter.
adapter::types::unidirectional::AW	OUT8	Plug (Output)	Eighth outgoing AW adapter.
adapter::types::unidirectional::AW	OUT9	Plug (Output)	Ninth outgoing AW adapter.

Functionality

The component has no processing logic whatsoever (no ECC, no algorithms). It functions purely as a wiring aid: The AW signal present at socket `IN` is structurally forwarded to all nine plug adapters `OUT1` to `OUT9`. In the IEC 61499 model, this means that all connections to the outputs receive the same signal present at the input. No timing or data manipulation takes place.

Technical Features

- **Generic Function Block**: The function block is declared as a generic class (`GEN_AW_SPLIT`). This allows reuse with different AW adapter types, provided the basic structure (unidirectional AW adapter) is maintained.
- **No Events or Data**: All communication occurs exclusively via adapter interfaces. This simplifies handling in purely adapter-based architectures.
- **No State Logic**: The function block has no inherent behavior; it is a purely "passive" element for signal copying.

State Overview

The function block does not have its own state machine, as no events or algorithms are defined. Its behavior is purely structural: As long as the input adapter is connected, all outputs are always active – no state change occurs.

Application Scenarios

- **Signal Fan**: A central output signal (e.g., a control command or sensor value) must be distributed to several downstream components.
- **Redundant Connections**: In safety-critical systems, the same output signal can be sent to multiple parallel evaluation units.
- **Wiring Simplification**: The function block replaces manual star wiring in the application and improves the clarity of the function diagram.

Comparison with Similar Function Blocks

- **AW_SPLIT_N**: A generic split function block with a variable number of outputs (e.g., `AW_SPLIT_4`, `AW_SPLIT_16`). `AW_SPLIT_9` is a special implementation for exactly nine outputs.
- **Data Split Blocks**: Blocks that split data inputs (e.g., `INT`, `REAL`). `AW_SPLIT_9` operates exclusively at the adapter level and not at individual data points.
- **Multiplexers (MUX)**: A multiplexer selects one of several inputs, while this block distributes one input to many outputs (fan-out).

Change Detection

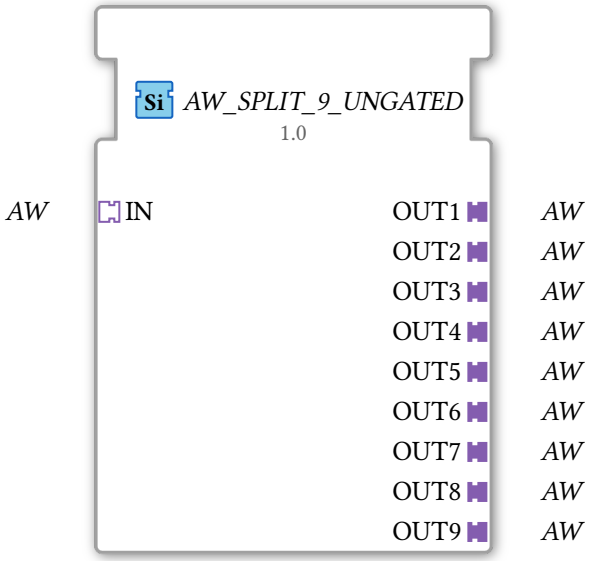
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AW_SPLIT_9` is a simple yet useful generic function block for splitting a unidirectional AW adapter signal into nine parallel outputs. Its structure is minimalist and purely passive, making it a reliable tool in adapter-oriented control architectures. It requires neither events nor data and can be integrated into existing applications without side effects.

AW_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AW_SPLIT_9_UNGATED` serves as a generic distributor for one incoming adapter of type `AW` to nine identically configured output adapters. It enables the multiple use of a single adapter signal without duplicating the original data source. The designation "SPLIT 1 AW into 9 AW" describes the core function: One AW input is distributed to nine AW outputs.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Type	Name	Direction	Description
adapter::types::unidirectional::AW	IN	Socket (Input)	Incoming AW adapter that is distributed to all outputs.
adapter::types::unidirectional::AW	OUT1	Plug (Output)	First outgoing AW adapter.
adapter::types::unidirectional::AW	OUT2	Plug (Output)	Second outgoing AW adapter.
adapter::types::unidirectional::AW	OUT3	Plug (Output)	Third outgoing AW adapter.
adapter::types::unidirectional::AW`	OUT4	Plug (Output)	Fourth outgoing AW adapter.
adapter::types::unidirectional::AW`	OUT5	Plug (Output)	Fifth outgoing AW adapter.
adapter::types::unidirectional::AW`	OUT6	Plug (Output)	Sixth outgoing AW adapter.
adapter::types::unidirectional::AW`	OUT7	Plug (Output)	Seventh outgoing AW adapter.
adapter::types::unidirectional::AW	OUT8	Plug (Output)	Eighth outgoing AW adapter.
adapter::types::unidirectional::AW	OUT9	Plug (Output)	Ninth outgoing AW adapter.

Functionality

The component has no processing logic whatsoever (no ECC, no algorithms). It functions purely as a wiring aid: The AW signal present at socket `IN` is structurally forwarded to all nine plug adapters `OUT1` to `OUT9`. In the IEC 61499 model, this means that all connections to the outputs receive the same signal present at the input. No timing or data manipulation takes place.

Technical Features

- **Generic Function Block**: The function block is declared as a generic class (`GEN_AW_SPLIT`). This allows reuse with different AW adapter types, provided the basic structure (unidirectional AW adapter) is maintained.
- **No Events or Data**: All communication occurs exclusively via adapter interfaces. This simplifies handling in purely adapter-based architectures.
- **No State Logic**: The function block has no inherent behavior; it is a purely "passive" element for signal copying.

State Overview

The function block does not have its own state machine, as no events or algorithms are defined. Its behavior is purely structural: As long as the input adapter is connected, all outputs are always active – no state change occurs.

Application Scenarios

- **Signal Fan**: A central output signal (e.g., a control command or sensor value) must be distributed to several downstream components.
- **Redundant Connections**: In safety-critical systems, the same output signal can be sent to multiple parallel evaluation units.
- **Wiring Simplification**: The function block replaces manual star wiring in the application and improves the clarity of the function diagram.

Comparison with Similar Function Blocks

- **AW_SPLIT_N**: A generic split function block with a variable number of outputs (e.g., `AW_SPLIT_4`, `AW_SPLIT_16`). `AW_SPLIT_9_UNGATED` is a special implementation for exactly nine outputs.
- **Data Split Blocks**: Blocks that split data inputs (e.g., `INT`, `REAL`). `AW_SPLIT_9_UNGATED` operates exclusively at the adapter level and not at individual data points.
- **Multiplexers (MUX)**: A multiplexer selects one of several inputs, while this block distributes one input to many outputs (fan-out).

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

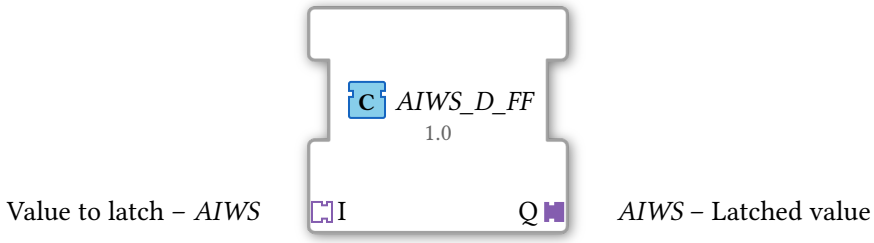
Conclusion

The `AW_SPLIT_9_UNGATED` is a simple yet useful generic function block for splitting a unidirectional AW adapter signal into nine parallel outputs. Its structure is minimalist and purely passive, making it a reliable tool in adapter-oriented control architectures. It requires neither events nor data and can be integrated into existing applications without side effects.

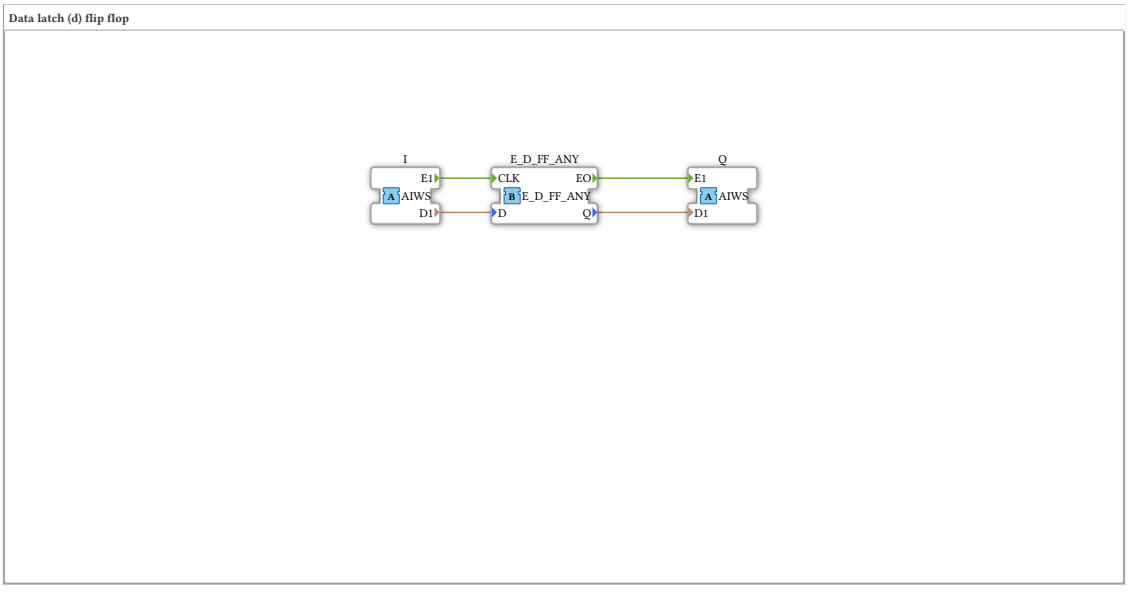
Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AIWS_D_FF



Value to latch – *AIWS* *AIWS* – Latched value



• – ○ • – ○ • – ○ •

Introduction

The AIWS_D_FF function block implements a data latch (D) flip-flop that stores an incoming data value and passes it to the output upon an event. Communication occurs exclusively via adapters, allowing the block to be seamlessly integrated into modular systems.

Interface Structure

Event Inputs

Name	Data Type	Description
I.E1	Event (via adapter)	Event that triggers the transfer of the data value

Event Outputs

Name	Data Type	Description
Q.E1	Event (via adapter)	Event that displays the stored data value

Data Inputs

Name	Data Type	Description
I.D1	ANY (via adapter)	Data value that is received upon arrival of I.E1

Data Outputs

Name	Data Type	Description
Q.D1	ANY (via adapter)	Stored data value that is output with Q.E1

Adapter

Name	Type	Direction	Description
I	`adapter::types::unidirectional::AIWS`	Socket (Input)	Provides the value to be latched and the triggering event
Q	`adapter::types::unidirectional::AIWS`	Plug (Output)	Outputs the stored value with an event

Functionality

The AIWS_D_FF encapsulates an internal `E_D_FF_ANY` module. The incoming adapter `I` provides an event via `I.E1`, which takes the value present at data port `I.D1`. The stored value is then output via `Q.D1`, accompanied by an event on `Q.E1`. Thus, the function block behaves like a classic clocked D flip-flop, but with a fully adapter-based interface.

Technical Features

- **Adapter-Based Interface:** All inputs and outputs are implemented via unidirectional adapters, enabling easy encapsulation and reuse in different contexts.
- **Internal Standard Function Block:** The implementation uses the IEC 61499 standard function block `E_D_FF_ANY`, which provides generic and proven latch functionality.
- **Unidirectional Data Flow:** The adapter type `AIWS` allows data flow in only one direction, improving clarity and preventing errors in networking.

State Overview

The AIWS_D_FF has no explicit internal states at the top level. The integrated `E_D_FF_ANY` goes through two phases:

1. **Waiting:** It waits for an event at the clock input.
2. **Accept & Output:** Upon a clock event, the data value is accepted and an output event is generated.

This sequence repeats with each incoming event.

Application Scenarios

- **Data Synchronization:** Buffering a sensor signal that is only to be passed on upon a specific trigger event.

- **Control Logic:** Storing a control signal as a function of a clock signal, e.g., for time-synchronized actuator control.
- **Modular Systems:** Integration into adapter-based architectures where functional blocks are connected via standardized interfaces.

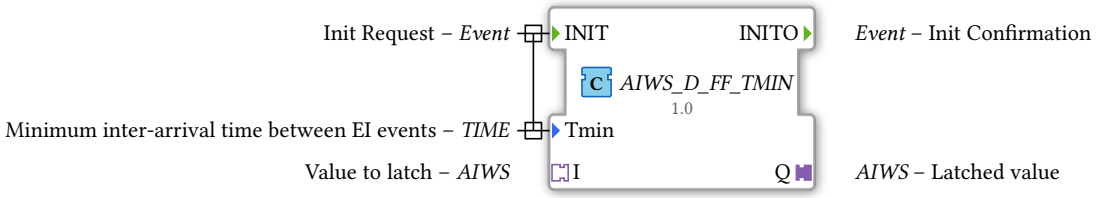
Comparison with Similar Components

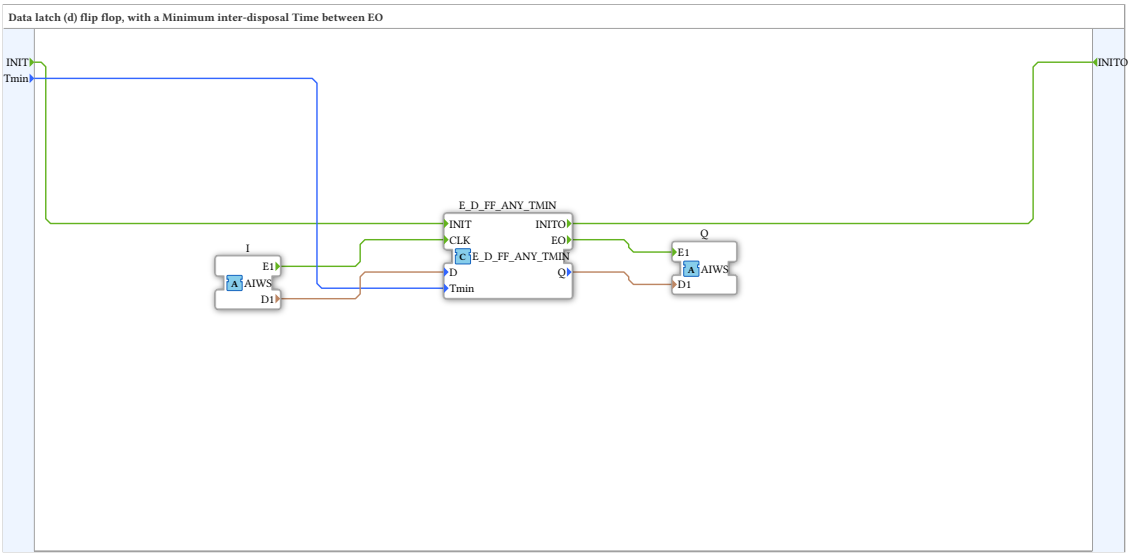
- **Simple D flip-flop (e.g., `E_D_FF`):** Has separate event and data ports, while the AIWS_D_FF encapsulates the same logic via adapters.
- **SR flip-flop:** Stores two states (Set/Reset), whereas the AIWS_D_FF only stores a single data value and does not offer a reset function.
- **Register components:** Can store multiple bits in parallel, while the AIWS_D_FF is designed for a single data value (ANY).

Conclusion

The AIWS_D_FF is a compact and modular functional block for implementing a D flip-flop in IEC 61499 systems. The exclusive use of adapters ensures high flexibility and reusability. It is particularly suitable for applications that require decoupled, event-driven data transfer.

AIWS_D_FF_TMIN





• – ○ • – ○ • – ○ •

Introduction

The function block **AIWS_D_FF_TMIN** implements a data latch (D flip-flop) with a minimum inter-disposal time between two consecutive output events. It is accessed via adapter interfaces (type: AIWS, unidirectional) that receive a value and output it as a latched value. Control is achieved via the input adapter's event (CLK edge), with output occurring only after a configurable dead time (Tmin) has elapsed.

Interface Structure

Event Inputs

Event	Type	Comment

INIT	EInit	Initialization Request (with Tmin parameter)

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization Acknowledgement

Data Inputs

Name	Type	Comment
Tmin	TIME	Minimum time between two output events (inter-disposal time)

Data Outputs

No dedicated data outputs – the latched value is provided exclusively via the **Q** adapter.

Adapters

Direction	Name	Type	Comment
Socket (Input)	I	adapter::types::unidirectional::AIWS	Value to be latched
Plug (Output)	Q	adapter::types::unidirectional::AIWS	Latched Value

Functionality

The function block internally uses the block `E_D_FF_ANY_TMIN` from the IEC 61499 library. The process is as follows:

1. At **INIT**, the minimum time `Tmin` is used, and the block is initialized (`INITO` confirms the successful start).
2. Each incoming event from adapter `I.E1` is interpreted as a clock edge (CLK).
3. The data value from adapter `I.D1` is transferred to the internal flip-flop on each CLK edge.
4. The latched value is provided on the output adapter `Q.D1`.
5. However, the corresponding event `Q.E1` is only sent after the time `Tmin` has elapsed since the last EO event (inter-disposal time). This limits the output rate.

The connection in the FB network ensures that events and data are correctly forwarded between the adapter `I`, the internal FB, and the adapter `Q`.

Technical Features

- **Adapter-Based Input/Output:** The function block uses only adapters of type `AIWS` (unidirectional) for signal processing. This allows for loose coupling and reuse of the interfaces.
- **Internal use of `E_D_FF_ANY_TMIN`:** The actual latch and timer logic is implemented by a specialized function block that already ensures a minimum time between two EO events.
- **Initialization required:** The function block must be initialized with `INIT` before its first use. This sets `Tmin`.

State overview

The function block essentially has two operating modes:

- **Initialization phase:** After receiving `INIT`, the internal timer and the flip-flop are reset. Upon completion, `INITO` is sent.
- **Normal operation:** After initialization, the function block waits for CLK events (`I.E1`). The data value is latched at each CLK (Current Control Point). An EO (Endpoint Output) event (`Q.E1`) is only triggered if the time elapsed since the last EO is ≥ **Tmin**. If this condition is not met, the EO is delayed.

A complete state machine is implemented in the internal FB (Function Block); the block described here transparently represents this.

Application Scenarios

- **Sensor Value Acquisition with Minimum Repetition Rate:** In automation, an analog or digital measured value should be passed on to a downstream controller at defined minimum intervals to avoid overload or unwanted switching cycles.
- **Edge Debouncing:** The latch can be used to keep an input value stable while the output is debounced by `Tmin`.
- **Time-controlled data buffer:** In communication protocols where data packets must be sent with a minimal pause.

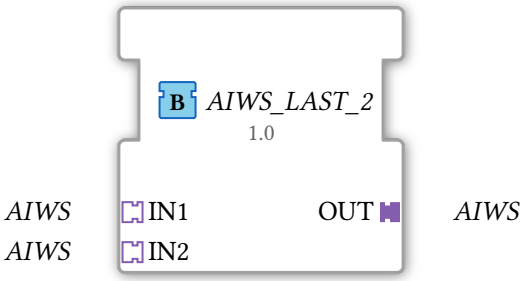
Comparison with similar function blocks

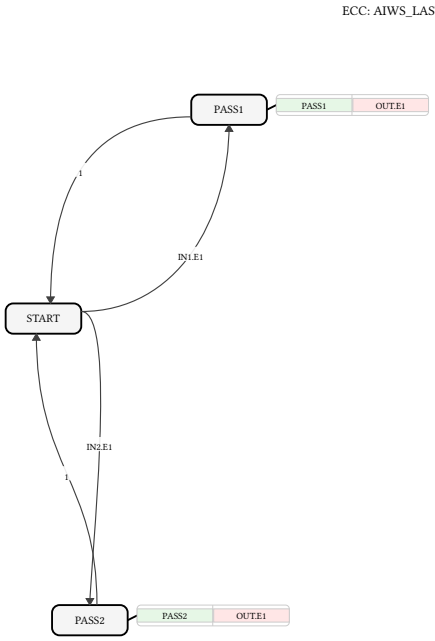
- **Standard D flip-flop (e.g., `E_D_FF`):** A standard D flip-flop accepts the input on every CLK edge and immediately outputs an event. It lacks a time limit on the output rate.
- **`AIWS_D_FF_TMIN`:** Extends the simple D flip-flop with a configurable minimum time between output events. This makes it particularly suitable for time-critical and rate-limited applications.
- **Timer-based latch function blocks (e.g., `E_D_FF_ANY_TMIN`):** The internal function block used here is the actual implementation; `AIWS_D_FF_TMIN` encapsulates this in an adapter-based interface, thus facilitating integration into an IEC 61499 architecture.

Conclusion

The `AIWS_D_FF_TMIN` function block combines the function of a data latch with a programmable minimum output delay. Thanks to its adapter-based interface definition, it can be flexibly deployed in distributed automation systems. It offers a simple and robust solution, particularly for applications where signal changes must occur only at specific time intervals.

AIWS_LAST_2





Introduction

The AIWS_LAST_2 function block merges two same-type WSTRING adapter signals (**IN1**, **IN2**) into one shared output (**OUT**), using a **last-writer-wins** rule: whichever of the two sockets last wrote an event wins, and its data value is passed through to OUT immediately and unchanged. This is explicitly not a naive merge (e.g. averaging or OR-ing) - it is pure arbitration by write order.

Interface Structure

Adapters

Input adapters (Sockets):

- **IN1**: WSTRING adapter (unidirectional) - first signal source
- **IN2**: WSTRING adapter (unidirectional) - second signal source

Output adapter (Plug):

- **OUT**: WSTRING adapter (unidirectional) - merged signal

How it works

The block is implemented as a simple Basic FB with a 3-state ECC (START, PASS1, PASS2). When an event arrives at IN1, the ECC moves to PASS1: IN1's current data value is copied to OUT and the adapter event is fired on OUT. When an event arrives at IN2, the same happens symmetrically via PASS2. After each pass-through the ECC immediately returns to START, ready for the next event from either input.

Technical Details

- Last-writer-wins semantics instead of a naive data combination
- Simple Basic FB (3-state ECC), no extra buffering needed
- No signal delay: the value is passed through in the same cycle the triggering event arrives
- Both inputs are equal - there is no fixed priority other than write order

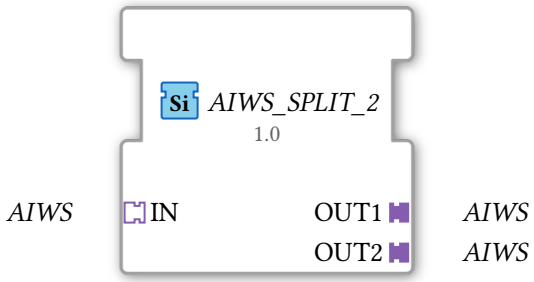
Application Scenarios

- Merging two equally-valid WSTRING signal sources into one downstream adapter path (e.g. two redundant sensors, or a manual and an automatic setpoint source)
- Arbitrating between two operator stations that may both write the same WSTRING value
- Simplifying networks that would otherwise need an inline `F_SEL`/`E_RS` construct

⚖️ Comparison with Similar Blocks

AIWS_LAST_2 is the inverse of AIWS_SPLIT_2: AIWS_SPLIT_2 distributes 1 input to 2 outputs, AIWS_LAST_2 merges 2 inputs into 1 output. For mixed operation between a data adapter and a pure event adapter (e.g. AX + AE), use AX_AE_MERGE instead, which merges only the event without altering the data value.

AIWS_SPLIT_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_SPLIT_2** is used to split an incoming AIWS adapter signal into two identical outputs. It is designed as a generic function block and allows multiple uses of an AIWS signal within the same application without requiring separate signal distribution programming.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Name	Type	Description
Socket (Input)	**IN**	adapter::types::unidirectional::AIWS	Incoming AIWS signal, which is distributed to both outputs.
Plug (Output)	**OUT1**	adapter::types::unidirectional::AIWS	First output – identical signal to the input.
Plug (Output)	**OUT2**	adapter::types::unidirectional::AIWS	Second output – identical signal to the input.

Functionality

The module passes the AIWS signal present at socket **IN** unchanged to both plugs **OUT1** and **OUT2**. No logical or timing processing takes place – the distribution occurs purely passively through the wiring within the functional block. This allows two independent adapter instances to be supplied with the same data simultaneously.

Technical Features

- **Generic Block**: The function block is declared as a generic type (`GEN_AIWS_SPLIT`), so it can be reused for different AIWS implementations.
- **Pure Adapter Logic**: Neither events nor data I/O are used; all communication takes place exclusively via the adapter interfaces.
- **No State Dependency**: The function block has no internal state machine – the output signal follows the input signal without delay or buffering.

State Overview

The function block is stateless. There are no internal states or transitions. The outputs are a direct mapping of the input at any given time.

Application Scenarios

- **Distributing a Sensor Signal**: A single AIWS sensor (e.g., analog value) is to be read simultaneously by two independent control or monitoring modules.
- **Redundant Processing**: In safety applications, the same signal can be passed to multiple parallel evaluation functions.
- **Logical Branching**: Splitting a data stream for different evaluations or visualizations in a 4diac application.

Comparison with Similar Modules

- **AIWS_SPLIT_3, AIWS_SPLIT_4**: Extended versions with three or four outputs, respectively – essentially the same functionality.
- **SPLIT Modules for Other Adapter Types**: 4diac offers analog split modules for other adapters (e.g., `DINT_SPLIT`) that distribute a data signal to multiple outputs – but at the data level, not via adapters.
- **Event Split Blocks**: Blocks like `E_SPLIT` distribute events, not data streams – AIWS_SPLIT_2 specifically addresses AIWS adapter communication.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AIWS_SPLIT_2** is a simple yet useful generic function block for duplicating an AIWS adapter signal. It reduces wiring complexity in applications that require the same analog or mixed signal multiple times and, thanks to its passive, stateless architecture, integrates seamlessly into existing workflows.

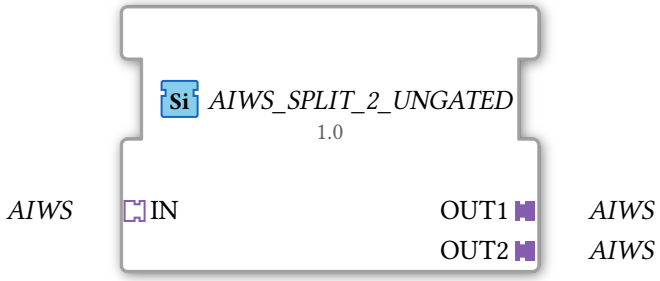
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AIWS_SPLIT_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_SPLIT_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_SPLIT_2_UNGATED** is used to split an incoming AIWS adapter signal into two identical outputs. It is designed as a generic function block and allows multiple uses of an AIWS signal within the same application without requiring separate signal distribution programming.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapter

Direction	Name	Type	Description
Socket (Input)	**IN**	`adapter::types::unidirectional::AIWS`	Incoming AIWS signal, which is distributed to both outputs.
Plug (Output)	**OUT1**	`adapter::types::unidirectional::AIWS`	First output – identical signal to the input.
Plug (Output)	**OUT2**	`adapter::types::unidirectional::AIWS`	Second output – identical signal to the input.

Functionality

The module passes the AIWS signal present at socket **IN** unchanged to both plugs **OUT1** and **OUT2**. No logical or timing processing takes place – the distribution occurs purely passively through the wiring within the functional block. This allows two independent adapter instances to be supplied with the same data simultaneously.

Technical Features

- **Generic Block**: The function block is declared as a generic type (`GEN_AIWS_SPLIT`), so it can be reused for different AIWS implementations.
- **Pure Adapter Logic**: Neither events nor data I/O are used; all communication takes place exclusively via the adapter interfaces.
- **No State Dependency**: The function block has no internal state machine – the output signal follows the input signal without delay or buffering.

State Overview

The function block is stateless. There are no internal states or transitions. The outputs are a direct mapping of the input at any given time.

Application Scenarios

- **Distributing a Sensor Signal**: A single AIWS sensor (e.g., analog value) is to be read simultaneously by two independent control or monitoring modules.
- **Redundant Processing**: In safety applications, the same signal can be passed to multiple parallel evaluation functions.
- **Logical Branching**: Splitting a data stream for different evaluations or visualizations in a 4diac application.

Comparison with Similar Modules

- **AIWS_SPLIT_3, AIWS_SPLIT_4**: Extended versions with three or four outputs, respectively – essentially the same functionality.
- **SPLIT Modules for Other Adapter Types**: 4diac offers analog split modules for other adapters (e.g., `DINT_SPLIT`) that distribute a data signal to multiple outputs – but at the data level, not via adapters.
- **Event Split Blocks**: Blocks like `E_SPLIT` distribute events, not data streams – AIWS_SPLIT_2_UNGATED specifically addresses AIWS adapter communication.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

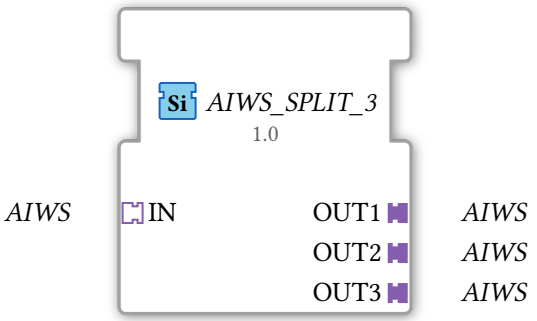
The **AIWS_SPLIT_2_UNGATED** is a simple yet useful generic function block for duplicating an AIWS adapter signal. It reduces wiring complexity in applications that require the same analog or mixed signal multiple times and, thanks to its passive, stateless architecture, integrates seamlessly into existing workflows.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AIWS_SPLIT_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_SPLIT_3** is used to split a single incoming AIWS adapter (type `adapter::types::unidirectional::AIWS`) into three identical output adapters. All data received via the input adapter is forwarded unchanged to all three outputs. The block is designed to be generic and can be used with various AIWS types.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapters

Direction	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::AIWS	Input adapter that receives the signal to be distributed.
Plug (Output)	OUT1	adapter::types::unidirectional::AIWS	First output adapter.
Plug (Output)	OUT2	adapter::types::unidirectional::AIWS	Second output adapter.
Plug (Output)	OUT3	adapter::types::unidirectional::AIWS	Third output adapter.

Functionality
The internal logic of the component is remarkably simple: As soon as the socket **IN** receives data from the connected adapter, this data is simultaneously passed on to the three plugs **OUT1**, **OUT2**, and **OUT3**. No transformation, filtering, or buffering takes place. Data is passed unidirectionally and without delay.

Technical Features
- **Generic Function Block:** The function block is implemented as a generic block (`eclipse4diac::core::GenericClassName = 'GEN_AIWS_SPLIT'`). This allows it to be used with various AIWS adapter specializations, as long as they adhere to the unidirectional communication protocol. The specific type is defined at design time in the 4diac IDE.
- **No State Automation:** The block has no events and no state machine (ECC). Data is passed purely flow-driven.
- **Scalability:** The block is specifically designed for a 1:3 split. Separate blocks (e.g., AIWS_SPLIT_2, AIWS_SPLIT_4) are available for other split ratios.

State Overview
This function block does not have a state machine (ECC), therefore there are no states or state transitions. Its function is purely combinatorial at the adapter level.

Application Scenarios
- **Signal Distribution:** An AIWS data stream from one source is to be forwarded in parallel to several subsequent processing blocks (e.g., monitoring, parallel filtering, or visualization).
- **Redundancy:** The same signal is used on different paths to achieve redundancy or comparability.
- **Testing and Simulation:** In test environments, the input signal can be made available to several test modules simultaneously without side effects.

Comparison with Similar Function Blocks
- **AIWS_SPLIT_2 / AIWS_SPLIT_N:** Function blocks with the same functionality but a different number of outputs (2 or n outputs).
- **ALL_SPLIT (for data):** A generic function block for splitting data inputs/outputs, not specific to adapters. AIWS_SPLIT_3, on the other hand, is specifically optimized for the AIWS adapter type and operates at the adapter level.
- **AIWS_JOIN_3:** The inverse function block that combines three AIWS inputs into one output.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AIWS_SPLIT_3** is a simple yet effective generic function block for duplicating an AIWS adapter signal to three parallel outputs. Thanks to its generic nature, it can be flexibly used in various automation and control applications where a data stream is required multiple times.

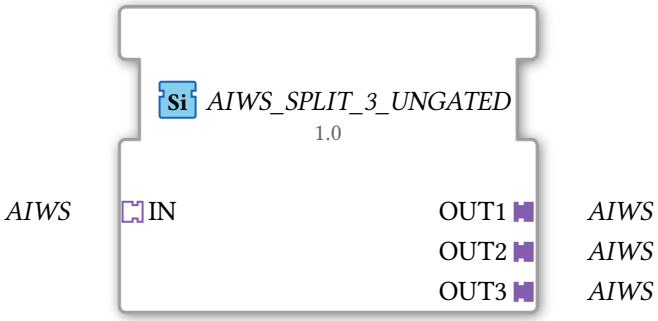
Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AIWS_SPLIT_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_SPLIT_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_SPLIT_3_UNGATED** is used to split a single incoming AIWS adapter (type `adapter::types::unidirectional::AIWS`) into three identical output adapters. All data received via the input adapter is forwarded unchanged to all three outputs. The block is designed to be generic and can be used with various AIWS types.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket (Input)	IN	adapter::types::unidirectional::AIWS	Input adapter that receives the signal to be distributed.
Plug (Output)	OUT1	adapter::types::unidirectional::AIWS	First output adapter.
Plug (Output)	OUT2	adapter::types::unidirectional::AIWS	Second output adapter.
Plug (Output)	OUT3	adapter::types::unidirectional::AIWS	Third output adapter.

Functionality

The internal logic of the component is remarkably simple: As soon as the socket **IN** receives data from the connected adapter, this data is simultaneously passed on to the three plugs **OUT1**, **OUT2**, and **OUT3**. No transformation, filtering, or buffering takes place. Data is passed unidirectionally and without delay.

Technical Features

- **Generic Function Block:** The function block is implemented as a generic block (`eclipse4diac::core::GenericClassName = 'GEN_AIWS_SPLIT'`). This allows it to be used with various AIWS adapter specializations, as long as they adhere to the unidirectional communication protocol. The specific type is defined at design time in the 4diac IDE.
- **No State Automation:** The block has no events and no state machine (ECC). Data is passed purely flow-driven.
- **Scalability:** The block is specifically designed for a 1:3 split. Separate blocks (e.g., AIWS_SPLIT_2, AIWS_SPLIT_4) are available for other split ratios.

State Overview

This function block does not have a state machine (ECC), therefore there are no states or state transitions. Its function is purely combinatorial at the adapter level.

Application Scenarios

- **Signal Distribution:** An AIWS data stream from one source is to be forwarded in parallel to several subsequent processing blocks (e.g., monitoring, parallel filtering, or visualization).
- **Redundancy:** The same signal is used on different paths to achieve redundancy or comparability.
- **Testing and Simulation:** In test environments, the input signal can be made available to several test modules simultaneously without side effects.

Comparison with Similar Function Blocks

- **AIWS_SPLIT_2 / AIWS_SPLIT_N:** Function blocks with the same functionality but a different number of outputs (2 or n outputs).
- **ALL_SPLIT (for data):** A generic function block for splitting data inputs/outputs, not specific to adapters. AIWS_SPLIT_3_UNGATED, on the other hand, is specifically optimized for the AIWS adapter type and operates at the adapter level.
- **AIWS_JOIN_3:** The inverse function block that combines three AIWS inputs into one output.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

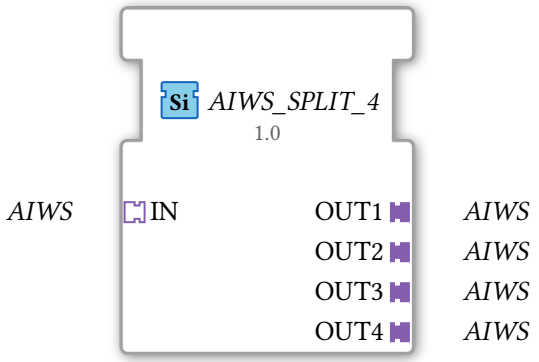
The **AIWS_SPLIT_3_UNGATED** is a simple yet effective generic function block for duplicating an AIWS adapter signal to three parallel outputs. Thanks to its generic nature, it can be flexibly used in various automation and control applications where a data stream is required multiple times.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AIWS_SPLIT_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AIWS_SPLIT_4` is used to split an incoming unidirectional **AIWS** adapter connection into four parallel outputs. It acts as a passive splitter, forwarding the values received via the socket to all four plugs without processing events or data itself.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Direction	Name	Description
`adapter::types::unidirectional::AIWS`	Socket	**IN**	Input adapter for the AIWS connection to be distributed
`adapter::types::unidirectional::AIWS`	Plug	**OUT1**	First output adapter
`adapter::types::unidirectional::AIWS`	Plug	**OUT2**	Second output adapter
`adapter::types::unidirectional::AIWS**	Plug	**OUT3**	Third output adapter
`adapter::types::unidirectional::AIWS`	Plug	**OUT4**	Fourth output adapter

Functionality

This component is a simple “splitter” for the unidirectional data type **AIWS**. It has no internal logic, no states, and no events of its own. All values arriving via the input adapter `IN` are copied identically to the four output adapters `OUT1` through `OUT4`. This allows multiple subsequent function blocks (FBs) to be supplied with the same data in parallel.

Since the FB is designed generically (`GenericClassName = 'GEN_AIWS_SPLIT'`), the specific data type **AIWS** must be defined when inserting it into a project.

Technical Features

- **Generic Implementation**: The FB is marked as a generic splitter via the attribute `eclipse4diac::core::GenericClassName`. The actual type is only determined during instantiation.
- **No Active Control**: The FB does not require any events to trigger – distribution occurs passively via the adapter connections.
- **Complete Transparency**: Changes to the adapter data are passed on to all outputs without delay.

State Overview

The function block (FB) has no states or sequences of its own. It uses purely combinatorial routing.

Application Scenarios

- **Distribution of Sensor Data** – A sensor value (e.g., temperature, pressure) is to be made available to multiple processing blocks simultaneously.
- **Redundant Data Backup** – A data stream is to be sent in parallel to two or more independent logic units.
- **Test and Simulation Setups** – A signal is split across multiple paths to test different response patterns.

Comparison with Similar Function Blocks

- **Manual Wiring**: Without a splitter, each subsequent FB would have its own connection to the source FB, which reduces clarity.
- **Event-Based Splitters**: Some splitters use event outputs (e.g., `SPLIT` for arbitrary events). `AIWS_SPLIT_4` is specifically designed for the unidirectional **AIWS** adapter and requires no events.
- **Multiple Outputs**: Variants with two (e.g., `AIWS_SPLIT_2`) or more outputs are available; `AIWS_SPLIT_4` provides four outputs by default.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The `AIWS_SPLIT_4` is a simple yet essential component for distributing unidirectional **AIWS** adapter connections. It reduces wiring complexity, ensures clear structures, and enables the parallel use of a data stream by multiple function blocks without additional logic.

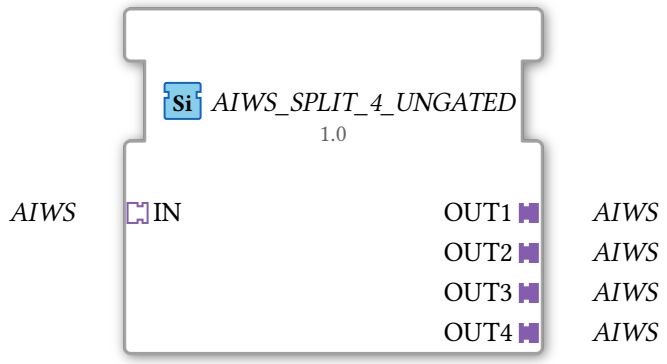
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIWS_SPLIT_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_SPLIT_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AIWS_SPLIT_4_UNGATED` is used to split an incoming unidirectional **AIWS** adapter connection into four parallel outputs. It acts as a passive splitter, forwarding the values received via the socket to all four plugs without processing events or data itself.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Direction	Name	Description
`adapter::types::unidirectional::AIWS`	Socket	**IN**	Input adapter for the AIWS connection to be distributed
`adapter::types::unidirectional::AIWS`	Plug	**OUT1**	First output adapter
`adapter::types::unidirectional::AIWS`	Plug	**OUT2**	Second output adapter
`adapter::types::unidirectional::AIWS**	Plug	**OUT3**	Third output adapter
`adapter::types::unidirectional::AIWS`	Plug	**OUT4**	Fourth output adapter

Functionality

This component is a simple “splitter” for the unidirectional data type **AIWS**. It has no internal logic, no states, and no events of its own. All values arriving via the input adapter `IN` are copied identically to the four output adapters `OUT1` through `OUT4`. This allows multiple subsequent function blocks (FBs) to be supplied with the same data in parallel.

Since the FB is designed generically (`GenericClassName = 'GEN_AIWS_SPLIT'`), the specific data type **AIWS** must be defined when inserting it into a project.

Technical Features

- **Generic Implementation**: The FB is marked as a generic splitter via the attribute `eclipse4diac::core::GenericClassName`. The actual type is only determined during instantiation.

- **No Active Control**: The FB does not require any events to trigger – distribution occurs passively via the adapter connections.
- **Complete Transparency**: Changes to the adapter data are passed on to all outputs without delay.

State Overview

The function block (FB) has no states or sequences of its own. It uses purely combinatorial routing.

Application Scenarios

- **Distribution of Sensor Data** – A sensor value (e.g., temperature, pressure) is to be made available to multiple processing blocks simultaneously.
- **Redundant Data Backup** – A data stream is to be sent in parallel to two or more independent logic units.
- **Test and Simulation Setups** – A signal is split across multiple paths to test different response patterns.

Comparison with Similar Function Blocks

- **Manual Wiring**: Without a splitter, each subsequent FB would have its own connection to the source FB, which reduces clarity.
- **Event-Based Splitters**: Some splitters use event outputs (e.g., `SPLIT` for arbitrary events). `AIWS_SPLIT_4_UNGATED` is specifically designed for the unidirectional **AIWS** adapter and requires no events.
- **Multiple Outputs**: Variants with two (e.g., `AIWS_SPLIT_2`) or more outputs are available; `AIWS_SPLIT_4_UNGATED` provides four outputs by default.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

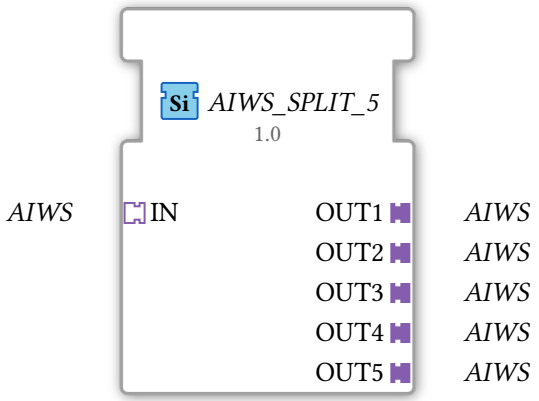
The `AIWS_SPLIT_4_UNGATED` is a simple yet essential component for distributing unidirectional **AIWS** adapter connections. It reduces wiring complexity, ensures clear structures, and enables the parallel use of a data stream by multiple function blocks without additional logic.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIWS_SPLIT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_SPLIT_5** is a generic block for splitting a single **AIWS** adapter into five identical **AIWS** adapters. It allows an AIWS signal provided by a socket to be forwarded to multiple independent adapter plugs without requiring event or data processing logic. This function block is

particularly suitable for signal distribution in industrial automation architectures based on the **IEC 61499** standard.

Interface Structure

Event Inputs
No event inputs are available. The block operates purely on a data-based or adapter-based basis.

Event Outputs
No event outputs are available. Forwarding is implicit via the adapter interfaces.

Data Inputs
No data inputs are available. All information is transmitted via the socket adapter.

Data Outputs
No data outputs are available. Output is via the plug adapters.

Adapters

Type	Label	Direction	Adapter Type
Socket (Input)	IN	Input	`adapter::types::unidirectional::AIWS`
Plug (Output)	OUT1	Output	`adapter::types::unidirectional::AIWS`
Plug (Output)	OUT2	Output	`adapter::types::unidirectional::AIWS`
Plug (Output)	OUT3	Output	`adapter::types::unidirectional::AIWS`
Plug (Output)	OUT4	Output	`adapter::types::unidirectional::AIWS`
Plug (Output)	OUT5	Output	`adapter::types::unidirectional::AIWS`

The function block contains a single **socket** (`IN`) and five **plugs** (`OUT1` ... `OUT5`), all of the same unidirectional **AIWS adapter type**.

Functionality
The **AIWS_SPLIT_5** forwards the adapter signal present at its socket `IN` unchanged to all five plug adapters `OUT1` to `OUT5`. No data transformation or filtering occurs – the function block acts purely as a **signal distributor** (fan-out). As soon as a connection is established via the socket, the output adapters are immediately available for communication. Event control or state-dependent logic is not required.

Technical Features
- **Generic Design**: The function block (FB) has the attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`, which allow for subsequent configuration or type adjustment. By default, the class name is set to `'GEN_AIWS_SPLIT'`; the type hash is empty.
- **Uniform Adapter Type**: All interfaces use the same unidirectional AIWS adapter. This ensures compatibility between inputs and outputs.
- **No Data or Event Ports**: All communication takes place exclusively via the adapter interfaces, which simplifies configuration and reduces the number of connections.

State Overview
The **AIWS_SPLIT_5** does not have an explicit state machine. Its functionality is purely combinatorial – as soon as the socket is connected, the same adapter signals are present at all outputs. Therefore, there are no internal states that influence the behavior.

Application Scenarios
- **Signal Distribution in Modular Systems**: An AIWS signal provided by a sensor or controller must be forwarded in parallel to multiple consumers (e.g., actuators, HMI systems, higher-level controllers).
- **Test and Simulation Environments**: A single AIWS signal is distributed to various test and analysis components for monitoring or recording.
- **Redundant Processing**: In safety-critical systems, the identical signal can be routed to multiple independent logic units to enable parallel evaluation.

Comparison with Similar Components

Component	Number of Outputs	Type	Special Feature
AIWS_SPLIT_5	5	AIWS Adapter	Generic, no events/data
AIWS_SPLIT_2	2	AIWS Adapter	Same functionality, minimal distribution
AIWS_SPLIT_3	3	AIWS Adapter	Medium number of outputs
AIWS_SPLIT_4	4	AIWS Adapter	Alternative to **5**

All split variants are based on the same principle and differ only in the number of output adapters. The **AIWS_SPLIT_5** covers a medium distribution requirement and is particularly advantageous when exactly five parallel outputs are needed.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AIWS_SPLIT_5** is a simple yet essential function block for duplicating AIWS adapter signals. Its generic design and the absence of events or data porting allow it to be integrated into existing IEC 61499 applications without additional configuration. Ideal for applications requiring precise distribution of an adapter across up to five independent paths.

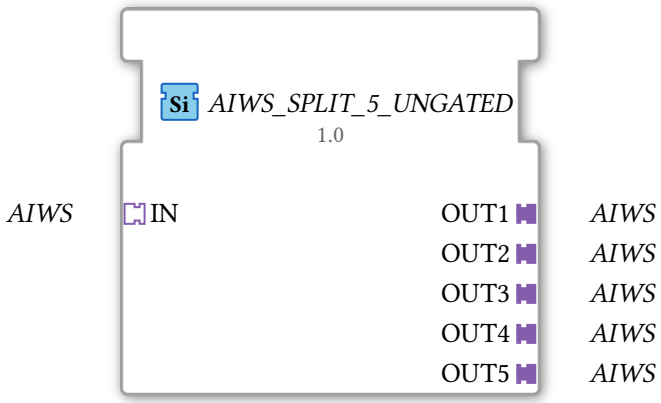
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIWS_SPLIT_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_SPLIT_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_SPLIT_5_UNGATED** is a generic block for splitting a single **AIWS** adapter into five identical **AIWS** adapters. It allows an AIWS signal provided by a socket to be forwarded to multiple independent adapter plugs without requiring event or data processing logic. This function block

is particularly suitable for signal distribution in industrial automation architectures based on the **IEC 61499** standard.

Interface Structure

Event Inputs
No event inputs are available. The block operates purely on a data-based or adapter-based basis.

Event Outputs
No event outputs are available. Forwarding is implicit via the adapter interfaces.

Data Inputs
No data inputs are available. All information is transmitted via the socket adapter.

Data Outputs
No data outputs are available. Output is via the plug adapters.

Adapters

Type	Label	Direction	Adapter Type
Socket (Input)	IN	Input	`adapter::types::unidirectional::AIWS`
Plug (Output)	OUT1	Output	`adapter::types::unidirectional::AIWS`
Plug (Output)	OUT2	Output	`adapter::types::unidirectional::AIWS`
Plug (Output)	OUT3	Output	`adapter::types::unidirectional::AIWS`
Plug (Output)	OUT4	Output	`adapter::types::unidirectional::AIWS`
Plug (Output)	OUT5	Output	`adapter::types::unidirectional::AIWS`

The function block contains a single **socket** (`IN`) and five **plugs** (`OUT1` ... `OUT5`), all of the same unidirectional **AIWS adapter type**.

Functionality
The **AIWS_SPLIT_5_UNGATED** forwards the adapter signal present at its socket `IN` unchanged to all five plug adapters `OUT1` to `OUT5`. No data transformation or filtering occurs – the function block acts purely as a **signal distributor** (fan-out). As soon as a connection is established via the socket, the output adapters are immediately available for communication. Event control or state-dependent logic is not required.

Technical Features
- **Generic Design**: The function block (FB) has the attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`, which allow for subsequent configuration or type adjustment. By default, the class name is set to `'GEN_AIWS_SPLIT'`; the type hash is empty.
- **Uniform Adapter Type**: All interfaces use the same unidirectional AIWS adapter. This ensures compatibility between inputs and outputs.
- **No Data or Event Ports**: All communication takes place exclusively via the adapter interfaces, which simplifies configuration and reduces the number of connections.

State Overview
The **AIWS_SPLIT_5_UNGATED** does not have an explicit state machine. Its functionality is purely combinatorial – as soon as the socket is connected, the same adapter signals are present at all outputs. Therefore, there are no internal states that influence the behavior.

Application Scenarios
- **Signal Distribution in Modular Systems**: An AIWS signal provided by a sensor or controller must be forwarded in parallel to multiple consumers (e.g., actuators, HMI systems, higher-level controllers).
- **Test and Simulation Environments**: A single AIWS signal is distributed to various test and analysis components for monitoring or recording.
- **Redundant Processing**: In safety-critical systems, the identical signal can be routed to multiple independent logic units to enable parallel evaluation.

Comparison with Similar Components

Component	Number of Outputs	Type	Special Feature
AIWS_SPLIT_5_UNGATED	5	AIWS Adapter	Generic, no events/data
AIWS_SPLIT_2	2	AIWS Adapter	Same functionality, minimal distribution
AIWS_SPLIT_3	3	AIWS Adapter	Medium number of outputs
AIWS_SPLIT_4	4	AIWS Adapter	Alternative to **5**

All split variants are based on the same principle and differ only in the number of output adapters. The **AIWS_SPLIT_5_UNGATED** covers a medium distribution requirement and is particularly advantageous when exactly five parallel outputs are needed.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

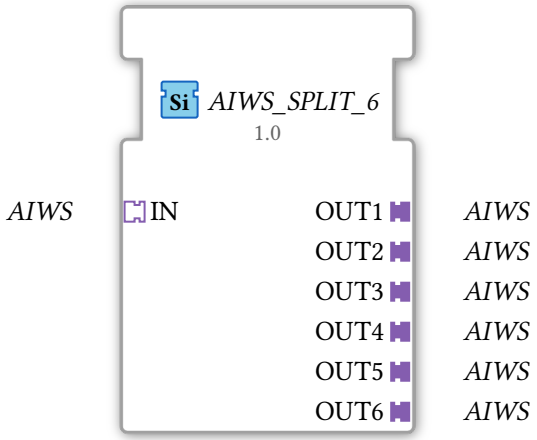
The **AIWS_SPLIT_5_UNGATED** is a simple yet essential function block for duplicating AIWS adapter signals. Its generic design and the absence of events or data porting allow it to be integrated into existing IEC 61499 applications without additional configuration. Ideal for applications requiring precise distribution of an adapter across up to five independent paths.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIWS_SPLIT_6



• – ◦ • – ◦ • – ◦ •

Introduction

The AIWS_SPLIT_6 function block is used to distribute an incoming AIWS signal (adapter type) to six identical outputs. It is designed as a generic function block and enables simple signal multiplication in control applications based on the 4diac IDE framework.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Socket (Input):

- `IN` (Type: `adapter::types::unidirectional::AIWS`): Input adapter for the AIWS signal to be distributed.

Plugs (Outputs):

- `OUT1` (Type: `adapter::types::unidirectional::AIWS`): First output with the split signal.
- `OUT2` (Type: `adapter::types::unidirectional::AIWS`): Second output.
- `OUT3` (Type: `adapter::types::unidirectional::AIWS`): Third output.
- `OUT4` (Type: `adapter::types::unidirectional::AIWS`): Fourth output.
- `OUT5` (Type: `adapter::types::unidirectional::AIWS`): Fifth output.
- `OUT6` (Type: `adapter::types::unidirectional::AIWS`): Sixth output.

Functionality

The AIWS_SPLIT_6 module receives the AIWS signal present at its socket `IN` and forwards it unchanged to all six plug outputs `OUT1` to `OUT6`. No signal processing or modification takes place; the module functions as a passive splitter (1:6). Communication occurs via adapters according to the unidirectional AIWS interface definition.

Technical Features

- The function block is implemented as a generic function block, identifiable by the attributes `GenericClassName` and `TypeHash`. This allows it to be created in various forms (e.g., AIWS_SPLIT_2, AIWS_SPLIT_4) as an instance of a generic type.
- There are no event or data inputs/outputs; all data transmission occurs exclusively via the adapter interfaces.
- The function block does not contain its own state logic or processing algorithms.

State Overview

The function block has no defined states, as it does not require event-driven execution. Signal propagation is continuous and passive.

Application Scenarios

- Distribution of an analog input signal (e.g., from a sensor) to multiple downstream function blocks.
- Provision of a reference signal for multiple parallel control loops.
- Use in modular automation solutions that require signal multiplication without additional hardware.

Comparison with similar function blocks

- **AIWS_SPLIT_2, AIWS_SPLIT_4:** These function blocks distribute the signal to two and four outputs, respectively. AIWS_SPLIT_6 extends the number to six.
- **AIWS_MERGE:** A compound function block that combines multiple AIWS inputs into one output – the functional inverse of a splitter.
- **Direct connection:** Instead of a splitter function block, multiple parallel connections could be made in the application model, but this reduces clarity and is more complex to modify.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion
The AIWS_SPLIT_6 is a simple yet useful function block for multiplying AIWS signals in IEC 61499-based controllers. Its generic nature and clear adapter interface facilitate reuse and integration into larger projects.

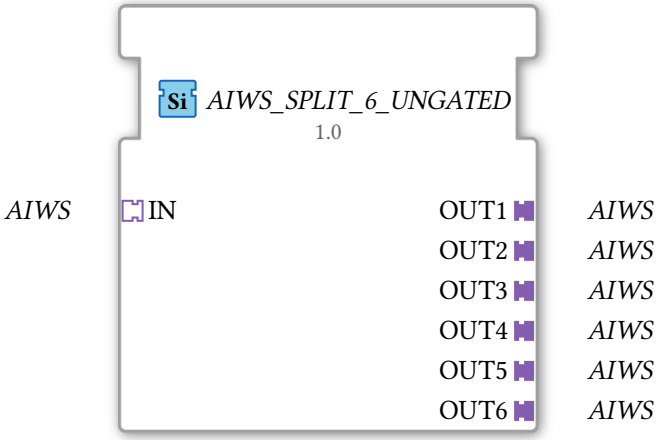
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIWS_SPLIT_6_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_SPLIT_6`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction
The AIWS_SPLIT_6_UNGATED function block is used to distribute an incoming AIWS signal (adapter type) to six identical outputs. It is designed as a generic function block and enables simple signal multiplication in control applications based on the 4diac IDE framework.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapters
Socket (Input):

- `IN` (Type: `adapter::types::unidirectional::AIWS`): Input adapter for the AIWS signal to be distributed.

Plugs (Outputs):

- `OUT1` (Type: `adapter::types::unidirectional::AIWS`): First output with the split signal.
- `OUT2` (Type: `adapter::types::unidirectional::AIWS`): Second output.
- `OUT3` (Type: `adapter::types::unidirectional::AIWS`): Third output.
- `OUT4` (Type: `adapter::types::unidirectional::AIWS`): Fourth output.
- `OUT5` (Type: `adapter::types::unidirectional::AIWS`): Fifth output.
- `OUT6` (Type: `adapter::types::unidirectional::AIWS`): Sixth output.

Functionality

The AIWS_SPLIT_6_UNGATED module receives the AIWS signal present at its socket `IN` and forwards it unchanged to all six plug outputs `OUT1` to `OUT6`. No signal processing or modification takes place; the module functions as a passive splitter (1:6). Communication occurs via adapters according to the unidirectional AIWS interface definition.

Technical Features

- The function block is implemented as a generic function block, identifiable by the attributes `GenericClassName` and `TypeHash`. This allows it to be created in various forms (e.g., AIWS_SPLIT_2, AIWS_SPLIT_4) as an instance of a generic type.
- There are no event or data inputs/outputs; all data transmission occurs exclusively via the adapter interfaces.
- The function block does not contain its own state logic or processing algorithms.

State Overview

The function block has no defined states, as it does not require event-driven execution. Signal propagation is continuous and passive.

Application Scenarios

- Distribution of an analog input signal (e.g., from a sensor) to multiple downstream function blocks.
- Provision of a reference signal for multiple parallel control loops.
- Use in modular automation solutions that require signal multiplication without additional hardware.

Comparison with similar function blocks

- **AIWS_SPLIT_2, AIWS_SPLIT_4:** These function blocks distribute the signal to two and four outputs, respectively. AIWS_SPLIT_6_UNGATED extends the number to six.
- **AIWS_MERGE:** A compound function block that combines multiple AIWS inputs into one output – the functional inverse of a splitter.
- **Direct connection:** Instead of a splitter function block, multiple parallel connections could be made in the application model, but this reduces clarity and is more complex to modify.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

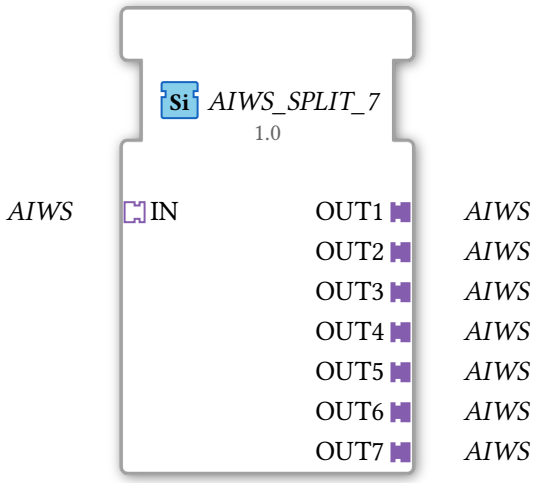
The AIWS_SPLIT_6_UNGATED is a simple yet useful function block for multiplying AIWS signals in IEC 61499-based controllers. Its generic nature and clear adapter interface facilitate reuse and integration into larger projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AIWS_SPLIT_7



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_SPLIT_7** is a generic component that splits a single incoming AIWS adapter (unidirectional) into seven identical outgoing AIWS adapters. It serves as a signal distributor for automation applications where a data stream or connection needs to be copied to multiple parallel paths.

Interface Structure

Event Inputs

None (the function block has no event inputs).

Event Outputs

None (the function block has no event outputs).

Data Inputs

None (the function block has no direct data inputs; data transmission occurs exclusively via the adapter interfaces).

Data Outputs

None (the function block has no direct data outputs; data transmission occurs exclusively via the adapter interfaces).

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AIWS`	**IN**	Socket (Input)	Receives the AIWS data stream to be distributed.
`adapter::types::unidirectional::AIWS`	**OUT1**	Plug (Output)	First identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT2**	Plug (Output)	Second identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT3**	Plug (Output)	Third identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT4**	Plug (Output)	Fourth identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT5**	Plug (Output)	Fifth identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT6**	Plug (Output)	Sixth identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT7**	Plug (Output)	Seventh identical output channel.

Functionality

The function block receives a unidirectional data stream of type `AIWS` via socket **IN**. This data stream is duplicated internally without modification and simultaneously passed to all seven plugs **OUT1** through **OUT7**. Each output channel receives exactly the same data as the input. There is neither delay nor data manipulation.

Technical Features

- **Generic Function Block** – The function block is defined as a generic component (attribute `GenericClassName = 'GEN_AIWS_SPLIT'`), which enables type-safe reuse in different contexts.
- **Unidirectional Adapters** – All adapters used are unidirectional, meaning data flows only from the socket to the plugs.
- **No Event Control** – The function block has no internal state machine and no event inputs/outputs; distribution is purely data-driven.
- **Scalable Architecture** – Thanks to the modular adapter interface, the function block can be used in complex hierarchies of function blocks.

State Overview

Since the function block does not have a state machine (Execution Control Chart), there are no explicit states. The output signals are always directly dependent on the input signals. The function block (FB) operates as a pure data distributor without delay or switching operations.

Application Scenarios

- **Parallel Processing** – Distribution of a measured value or control signal to multiple independent control or monitoring paths.
- **Signal Monitoring** – The same signal is routed to different diagnostic or logic blocks.
- **Redundancy** – Provision of identical data streams for different receivers (e.g., operation, visualization, and archiving).
- **Test Environments** – A test signal can be sent to multiple simulation blocks simultaneously.

Comparison with Similar splitters

- **AIWS_SPLIT_2, AIWS_SPLIT_4** – splitters with the same functionality but a different number of outputs (2 or 4). AIWS_SPLIT_7 offers the maximum number of outputs within this splitter family.
- **Generic 1:N Splitters** – Other IEC 61499 libraries often contain generic splitters for data types such as `ANY`. AIWS_SPLIT_7 is specifically designed for the adapter type `AIWS` and therefore requires no type conversion.

Change Detection

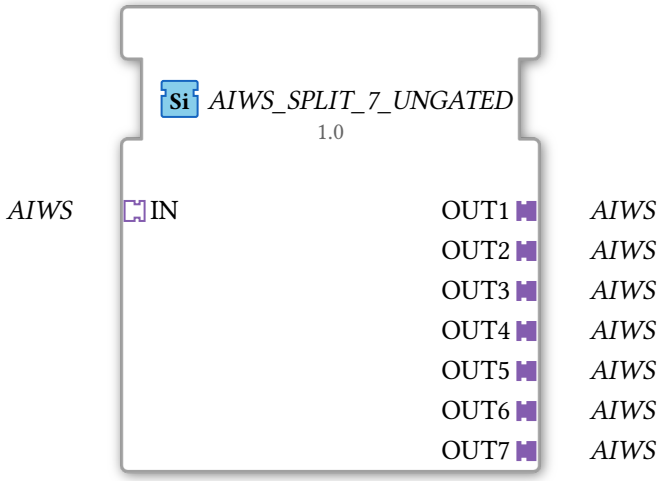
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

The **AIWS_SPLIT_7** is a simple yet powerful function block for multiplying a unidirectional AIWS data stream to seven parallel outputs. Its generic definition and purely adapter-based interface enable flexible integration into IEC 61499 applications, particularly for signal distribution and parallel processing tasks.

AIWS_SPLIT_7_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_SPLIT_7`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_SPLIT_7_UNGATED** is a generic component that splits a single incoming AIWS adapter (unidirectional) into seven identical outgoing AIWS adapters. It serves as a signal distributor for automation applications where a data stream or connection needs to be copied to multiple parallel paths.

Interface Structure

Event Inputs

None (the function block has no event inputs).

Event Outputs

None (the function block has no event outputs).

Data Inputs

None (the function block has no direct data inputs; data transmission occurs exclusively via the adapter interfaces).

Data Outputs

None (the function block has no direct data outputs; data transmission occurs exclusively via the adapter interfaces).

Adapters

Type	Name	Direction	Description
`adapter::types::unidirectional::AIWS`	**IN**	Socket (Input)	Receives the AIWS data stream to be distributed.
`adapter::types::unidirectional::AIWS`	**OUT1**	Plug (Output)	First identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT2**	Plug (Output)	Second identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT3**	Plug (Output)	Third identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT4**	Plug (Output)	Fourth identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT5**	Plug (Output)	Fifth identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT6**	Plug (Output)	Sixth identical output channel.
`adapter::types::unidirectional::AIWS`	**OUT7**	Plug (Output)	Seventh identical output channel.

Functionality

The function block receives a unidirectional data stream of type `AIWS` via socket **IN**. This data stream is duplicated internally without modification and simultaneously passed to all seven plugs **OUT1** through **OUT7**. Each output channel receives exactly the same data as the input. There is neither delay nor data manipulation.

Technical Features

- **Generic Function Block** – The function block is defined as a generic component (attribute `GenericClassName = 'GEN_AIWS_SPLIT'`), which enables type-safe reuse in different contexts.
- **Unidirectional Adapters** – All adapters used are unidirectional, meaning data flows only from the socket to the plugs.
- **No Event Control** – The function block has no internal state machine and no event inputs/outputs; distribution is purely data-driven.
- **Scalable Architecture** – Thanks to the modular adapter interface, the function block can be used in complex hierarchies of function blocks.

State Overview

Since the function block does not have a state machine (Execution Control Chart), there are no explicit states. The output signals are always directly dependent on the input signals. The function block (FB) operates as a pure data distributor without delay or switching operations.

Application Scenarios

- **Parallel Processing** – Distribution of a measured value or control signal to multiple independent control or monitoring paths.
- **Signal Monitoring** – The same signal is routed to different diagnostic or logic blocks.
- **Redundancy** – Provision of identical data streams for different receivers (e.g., operation, visualization, and archiving).
- **Test Environments** – A test signal can be sent to multiple simulation blocks simultaneously.

Comparison with Similar splitters

- **AIWS_SPLIT_2, AIWS_SPLIT_4** – splitters with the same functionality but a different number of outputs (2 or 4). AIWS_SPLIT_7_UNGATED offers the maximum number of outputs within this splitter family.
- **Generic 1:N Splitters** – Other IEC 61499 libraries often contain generic splitters for data types such as `ANY`. AIWS_SPLIT_7_UNGATED is specifically designed for the adapter type `AIWS` and therefore requires no type conversion.

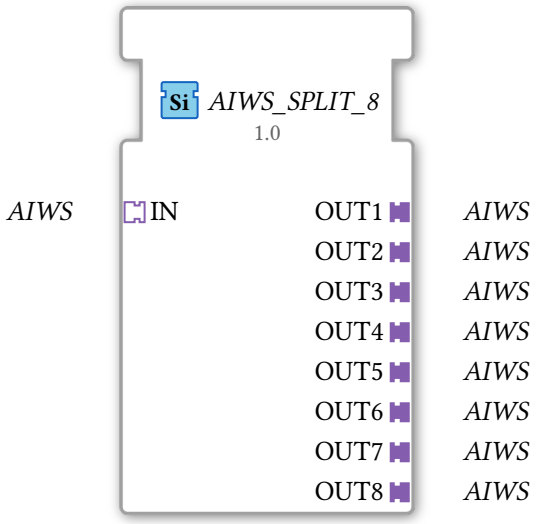
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AIWS_SPLIT_7_UNGATED** is a simple yet powerful function block for multiplying a unidirectional AIWS data stream to seven parallel outputs. Its generic definition and purely adapter-based interface enable flexible integration into IEC 61499 applications, particularly for signal distribution and parallel processing tasks.

AIWS_SPLIT_8



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_SPLIT_8** splits an incoming AIWS adapter signal into eight identical outputs. It functions as a generic splitter for unidirectional adapter connections and enables the parallel forwarding of a signal to multiple subsequent function blocks.

Interface Structure

The function block has neither event nor data inputs or outputs. All communication takes place exclusively via adapters.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Input (Socket)	IN	adaptor::types::unidirectional::AIWS	Incoming AIWS signal to be split.
Output (Plug)	OUT1	adaptor::types::unidirectional::AIWS	First parallel output of the signal.
Output (Plug)	OUT2	adaptor::types::unidirectional::AIWS	Second parallel output of the signal.
Output (Plug)	OUT3	adaptor::types::unidirectional::AIWS	Third parallel output of the signal.
Output (Plug)	OUT4	adaptor::types::unidirectional::AIWS	Fourth parallel output of the signal.

Output (Plug)	OUT5	adapater::types::unidirectional::AIWS	Fifth parallel output of the signal.
Output (Plug)	OUT6	adapater::types::unidirectional::AIWS	Sixth parallel output of the signal.
Output (Plug)	OUT7	adapater::types::unidirectional::AIWS	Seventh parallel output of the signal.
Output (Plug)	OUT8	adapater::types::unidirectional::AIWS	Eighth parallel output of the signal.

Functionality

The module receives an AIWS (adapter) signal via the **IN** socket. This signal is duplicated to all eight **OUT** plugs without processing or delay. Changes to the input signal are immediately reflected at all outputs.

Technical Features

- The module is generic and designed for a fixed number of eight outputs.
- There are no event-driven processes – signal transmission is purely adapter-based and continuous.
- The implementation is a pure splitter without internal logic or state management.

State Overview

The module does not have a state machine. There are no internal states or event sequences – the output directly follows the input.

Application Scenarios

- **Signal Distribution** in Automation Applications: An AIWS sensor value is to be made available to several independent control or monitoring blocks simultaneously.
- **Parallelization** of Algorithms: Subsequent modules require the same adapter data stream but operate in parallel on different resources.
- **Test and Simulation Environments**: An incoming signal is distributed across multiple evaluation instances to compare different processing paths.

Comparison with Similar Components

- **AIWS_SPLIT_4** – Splitter with four outputs.
- **AIWS_SPLIT_16** – Splitter with 16 outputs.
- **AIWS_MERGE** – Combines multiple AIWS adapters into a single output (counterpart).

All split components share the feature of passing the signal through unchanged; they differ only in the number of outputs.

Change Detection

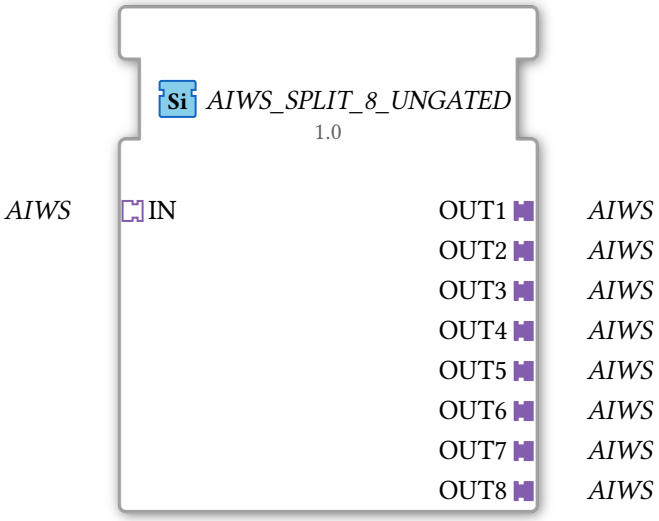
Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AIWS_SPLIT_8 is a simple yet essential component for multiplying an AIWS adapter signal across eight parallel paths. Its implementation without event-driven control or state logic makes it lightweight and reliable for use in distributed automation systems.

AIWS_SPLIT_8_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_SPLIT_8`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_SPLIT_8_UNGATED** splits an incoming AIWS adapter signal into eight identical outputs. It functions as a generic splitter for unidirectional adapter connections and enables the parallel forwarding of a signal to multiple subsequent function blocks.

Interface Structure

The function block has neither event nor data inputs or outputs. All communication takes place exclusively via adapters.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Input (Socket)	IN	adaptor::types::unidirectional::AIWS	Incoming AIWS signal to be split.
Output (Plug)	OUT1	adaptor::types::unidirectional::AIWS	First parallel output of the signal.
Output (Plug)	OUT2	adaptor::types::unidirectional::AIWS	Second parallel output of the signal.
Output (Plug)	OUT3	adaptor::types::unidirectional::AIWS	Third parallel output of the signal.
Output (Plug)	OUT4	adaptor::types::unidirectional::AIWS	Fourth parallel output of the signal.

Output (Plug)	OUT5	adapater::types::unidirectional::AIWS	Fifth parallel output of the signal.
Output (Plug)	OUT6	adapater::types::unidirectional::AIWS	Sixth parallel output of the signal.
Output (Plug)	OUT7	adapater::types::unidirectional::AIWS	Seventh parallel output of the signal.
Output (Plug)	OUT8	adapater::types::unidirectional::AIWS	Eighth parallel output of the signal.

Functionality

The module receives an AIWS (adapter) signal via the **IN** socket. This signal is duplicated to all eight **OUT** plugs without processing or delay. Changes to the input signal are immediately reflected at all outputs.

Technical Features

- The module is generic and designed for a fixed number of eight outputs.
- There are no event-driven processes – signal transmission is purely adapter-based and continuous.
- The implementation is a pure splitter without internal logic or state management.

State Overview

The module does not have a state machine. There are no internal states or event sequences – the output directly follows the input.

Application Scenarios

- **Signal Distribution** in Automation Applications: An AIWS sensor value is to be made available to several independent control or monitoring blocks simultaneously.
- **Parallelization** of Algorithms: Subsequent modules require the same adapter data stream but operate in parallel on different resources.
- **Test and Simulation Environments**: An incoming signal is distributed across multiple evaluation instances to compare different processing paths.

Comparison with Similar Components

- **AIWS_SPLIT_4** – Splitter with four outputs.
- **AIWS_SPLIT_16** – Splitter with 16 outputs.
- **AIWS_MERGE** – Combines multiple AIWS adapters into a single output (counterpart).

All split components share the feature of passing the signal through unchanged; they differ only in the number of outputs.

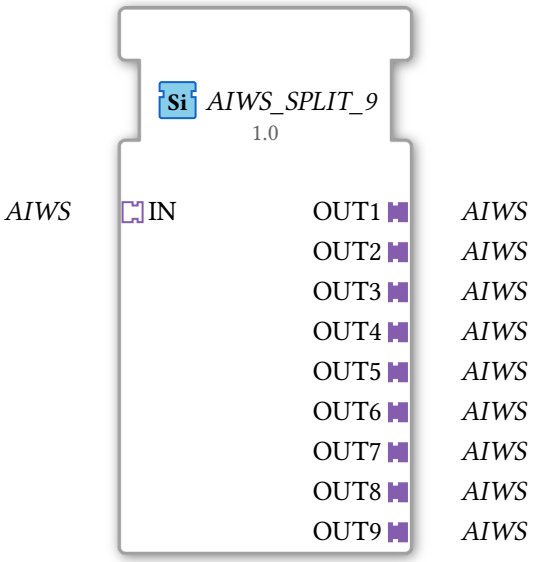
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AIWS_SPLIT_8_UNGATED is a simple yet essential component for multiplying an AIWS adapter signal across eight parallel paths. Its implementation without event-driven control or state logic makes it lightweight and reliable for use in distributed automation systems.

AIWS_SPLIT_9



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AIWS_SPLIT_9` splits an incoming `AIWS` adapter signal into nine separate, identically typed outputs. It is provided as a generic function block (FB) and enables the distribution of a single adapter data stream to multiple subsequent function units. The block is unidirectional and requires no event or data inputs, as data transmission occurs exclusively via the adapter interfaces.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AIWS`	Incoming AIWS signal, distributed to the nine outputs.
Plug	`OUT1`	`adapter::types::unidirectional::AIWS`	First AIWS outgoing output.
Plug	`OUT2`	`adapter::types::unidirectional::AIWS`	Second AIWS outgoing output.
Plug	`OUT3`	`adapter::types::unidirectional::AIWS`	Third AIWS outgoing output.
Plug	`OUT4`	`adapter::types::unidirectional::AIWS`	Fourth AIWS outgoing output.
Plug	`OUT5`	`adapter::types::unidirectional::AIWS`	Fifth AIWS outgoing output.
Plug	`OUT6`	`adapter::types::unidirectional::AIWS`	Sixth AIWS outgoing output.

Plug	`OUT7`	`adapter::types::unidirectional::AIWS`	Seventh AIWS outgoing output.
Plug	`OUT8`	`adapter::types::unidirectional::AIWS`	Eighth AIWS output.
Plug	`OUT9`	`adapter::types::unidirectional::AIWS`	Ninth AIWS output.

Functionality

The module receives an AIWS adapter signal at socket `IN` and forwards it unchanged to all nine plugs `OUT1` to `OUT9`. No data processing, conversion, or delay takes place – it is a pure signal multiplication (fan-out). The internal logic is implemented as a generic function block, which can be parameterized at runtime by specifying a generic class name (`eclipse4diac::core::GenericClassName`).

Technical Features

- **Generic Function Block**: The block is marked as generic `GEN_AIWS_SPLIT`. This allows it to be reused in different contexts by defining the specific adapter type at design time.
- **Unidirectionality**: Both the input and all outputs are unidirectional (`adapter::types::unidirectional::AIWS`), meaning data flows only from the socket to the plugs.
- **No Event Control**: The split block operates without events – data is passed implicitly via the adapter connections.
- **No State Machine**: There is no Execution Control Chart (ECC), therefore the block is stateless and requires no initialization.
- **Type Hash**: An optional attribute `eclipse4diac::core::TypeHash` can be set for identification and versioning purposes, but is empty in this case.

State Overview

The function block has no internal states. It is purely combinatorial and does not execute any sequential processes.

Application Scenarios

- **Signal Distribution in Automation Systems**: When a sensor or control signal (e.g., an AIWS-compliant temperature or pressure value) needs to be distributed to multiple independent modules.
- **Redundant Processing**: A signal is distributed to identical, parallel-operating algorithms or safety logic.
- **Test and Simulation Environments**: A single AIWS signal is to be split across multiple test or monitoring blocks.
- **Architecture Simplification**: Eliminates the need for manual wiring of multiple split blocks and reduces the complexity of the application diagram.

Comparison with Similar Function Blocks

- **AIWS_SPLIT_n (e.g., AIWS_SPLIT_4)**: Other variants exist with a different number of outputs (e.g., 2, 4, 8). This block specializes in splitting the signal into exactly nine strands.
- **Manual Split with Multiple Function Block Instances**: Without this function block, the AIWS signal would have to be implemented by cascading several 2- or 3-way split blocks, which reduces clarity.
- **Event-Based Data Distributor**: Function blocks controlled by events require additional event wiring and are less efficient for simple data forwarding.

Change Detection

Each output plug is updated independently: the incoming value is written to a given output, and its adapter event sent, only if it differs from that output's current value. Outputs that are already in sync stay quiet, while an output that was just connected (or has drifted out of sync) still receives the update it needs.

Conclusion

AIWS_SPLIT_9` is a simple yet useful generic function block for multiplying a unidirectional AIWS adapter signal to nine outputs. It avoids unnecessary complexity, requires no event control, and can be used directly in IEC 61499 applications without additional configuration. Thanks to its generic nature, it is flexibly adaptable to various adapter types and is particularly suitable for signal distribution in modular automation architectures.

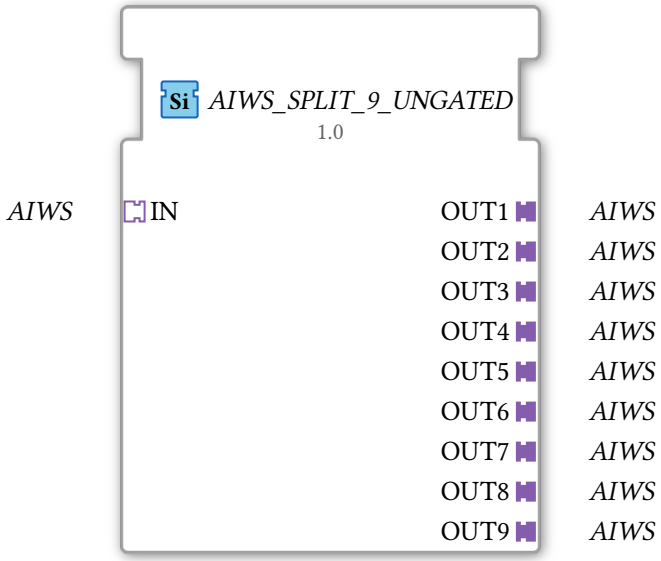
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AIWS_SPLIT_9_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_SPLIT_9`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AIWS_SPLIT_9_UNGATED` splits an incoming `AIWS` adapter signal into nine separate, identically typed outputs. It is provided as a generic function block (FB) and enables the distribution of a single adapter data stream to multiple subsequent function units. The block is unidirectional and requires no event or data inputs, as data transmission occurs exclusively via the adapter interfaces.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description

Socket	IN	adapter::types::unidirectional::AIWS	Incoming AIWS signal, distributed to the nine outputs.
Plug	OUT1	adapter::types::unidirectional::AIWS	First AIWS outgoing output.
Plug	OUT2	adapter::types::unidirectional::AIWS	Second AIWS outgoing output.
Plug	OUT3	adapter::types::unidirectional::AIWS	Third AIWS outgoing output.
Plug	OUT4	adapter::types::unidirectional::AIWS	Fourth AIWS outgoing output.
Plug	OUT5	adapter::types::unidirectional::AIWS	Fifth AIWS outgoing output.
Plug	OUT6	adapter::types::unidirectional::AIWS	Sixth AIWS outgoing output.
Plug	OUT7	adapter::types::unidirectional::AIWS	Seventh AIWS outgoing output.
Plug	OUT8	adapter::types::unidirectional::AIWS	Eighth AIWS output.
Plug	OUT9	adapter::types::unidirectional::AIWS	Ninth AIWS output.

Functionality

The module receives an AIWS adapter signal at socket `IN` and forwards it unchanged to all nine plugs `OUT1` to `OUT9`. No data processing, conversion, or delay takes place – it is a pure signal multiplication (fan-out). The internal logic is implemented as a generic function block, which can be parameterized at runtime by specifying a generic class name (`eclipse4diac::core::GenericClassName`).

Technical Features

- **Generic Function Block**: The block is marked as generic `GEN_AIWS_SPLIT`. This allows it to be reused in different contexts by defining the specific adapter type at design time.
- **Unidirectionality**: Both the input and all outputs are unidirectional (`adapter::types::unidirectional::AIWS`), meaning data flows only from the socket to the plugs.
- **No Event Control**: The split block operates without events – data is passed implicitly via the adapter connections.
- **No State Machine**: There is no Execution Control Chart (ECC), therefore the block is stateless and requires no initialization.
- **Type Hash**: An optional attribute `eclipse4diac::core::TypeHash` can be set for identification and versioning purposes, but is empty in this case.

State Overview

The function block has no internal states. It is purely combinatorial and does not execute any sequential processes.

Application Scenarios

- **Signal Distribution in Automation Systems**: When a sensor or control signal (e.g., an AIWS-compliant temperature or pressure value) needs to be distributed to multiple independent modules.
- **Redundant Processing**: A signal is distributed to identical, parallel-operating algorithms or safety logic.
- **Test and Simulation Environments**: A single AIWS signal is to be split across multiple test or monitoring blocks.
- **Architecture Simplification**: Eliminates the need for manual wiring of multiple split blocks and reduces the complexity of the application diagram.

Comparison with Similar Function Blocks

- **AIWS_SPLIT_n (e.g., AIWS_SPLIT_4)**: Other variants exist with a different number of outputs (e.g., 2, 4, 8). This block specializes in splitting the signal into exactly nine strands.
- **Manual Split with Multiple Function Block Instances**: Without this function block, the AIWS signal would have to be implemented by cascading several 2- or 3-way split blocks, which reduces clarity.
- **Event-Based Data Distributor**: Function blocks controlled by events require additional event wiring and are less efficient for simple data forwarding.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AIWS_SPLIT_9_UNGATED` is a simple yet useful generic function block for multiplying a unidirectional AIWS adapter signal to nine outputs. It avoids unnecessary complexity, requires no event control, and can be used directly in IEC 61499 applications without additional configuration. Thanks to its generic nature, it is flexibly adaptable to various adapter types and is particularly suitable for signal distribution in modular automation architectures.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

timers

2.1.11.10 iec61131

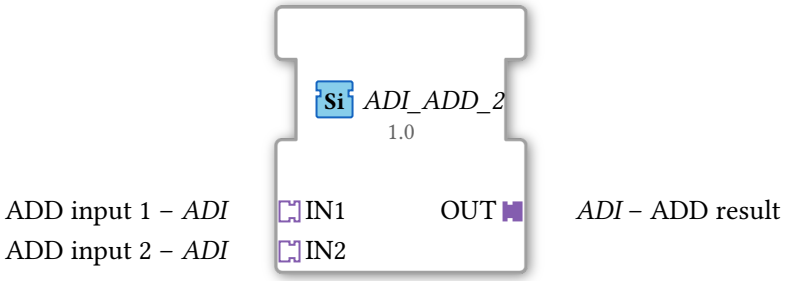
timers

2.1.11.11 Iec61131 3

Numeric Index (iec61131-3)

Here you will find all exercises from the `iec61131-3` workspace, listed in ascending order. …

ADI_ADD_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_ADD_2` is a generic function block for performing arithmetic addition operations. Unlike conventional mathematical function blocks, this block uses an adapter-based interface concept (unidirectional `ADI` adapters) to transmit data and associated control signals in a bundled manner. It enables the addition of two input values to produce one output value.

Interface Structure

Event Inputs

This function block does not have direct, dedicated event inputs. Event control is handled entirely via the adapters.

Event Outputs

This function block does not have direct, dedicated event outputs. Event forwarding is handled entirely via the output adapter.

Data Inputs

This function block does not have any standard, direct data inputs.

Data Outputs

This function block does not have any standard, direct data outputs.

Adapters

Sockets (Inputs)

- **IN1**: Type `adapter::types::unidirectional::ADI`
- *Description:* First input value for mathematical addition.
- **IN2**: Type `adapter::types::unidirectional::ADI`
- *Description:* Second input value for mathematical addition.

Plugs (Connectors - Outputs)

- **OUT**: Type `adapter::types::unidirectional::ADI`
- *Description:* Output that provides the result of the addition (`IN1 + IN2`).

Functionality

The function block `ADI_ADD_2` performs the arithmetic operation:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.4}$$

As soon as data changes at the input adapters `IN1` or `IN2`, or a corresponding transmission event is signaled via the adapter structure, the function block processes the values. The sum is calculated and forwarded to subsequent function blocks via the corresponding event/data bundle of the output adapter `OUT`.

Since this is a generic function block (`GEN_ADI_ADD`), the specific data type used depends on the specifications and instantiation of the `ADI` adapters used.

Technical Features

- **Generic Type:** The function block is declared as `GEN_ADI_ADD`. This allows for flexible handling of different numeric data types, provided they are supported by the adapters used.
- **Adapter-Based Design:** By using unidirectional adapters of type `ADI`, the number of explicit connection lines (wiring overhead in the 4diac IDE's Application Editor) is drastically reduced, as data and synchronization events are encapsulated in a single connection.

•

State Overview

The function block behaves like a stateless (or purely event-driven combinational) function block:

1. **Waiting for Data Update:** The function block remains in idle state until new values are signaled via `IN1` or `IN2`.
2. **Calculation:** When an event arrives at the inputs, the mathematical sum is calculated.
3. **Output:** The result is directly passed to the output adapter `OUT`, and the corresponding trigger event is triggered.

–

Application Scenarios

- **Modular Signal Processing:** Addition of measured values (e.g., Sensor 1 + Sensor 2 to determine a total value) in systems that are consistently based on an adapter architecture.
- **Cascaded Computations:** Easy integration into complex arithmetic computation networks through clean, structured adapter connections.

Comparison with Similar Components

Compared to a classic `ADD` component according to IEC 61499 (which typically has dedicated `REQ` and `CNF` event ports, as well as direct data inputs such as `IN1` and `IN2` as `ANY_NUM`), the `ADI_ADD_2` encapsulates these interfaces in adapters.

- **Standard ADD:** Requires manual wiring of a minimum of 2 events and 3 data lines (5 connections in total).

- **ADI_ADD_2:** Only requires connecting the three adapter lines (`IN1`, `IN2`, `OUT`), which significantly improves the readability of complex control diagrams.

Change Detection

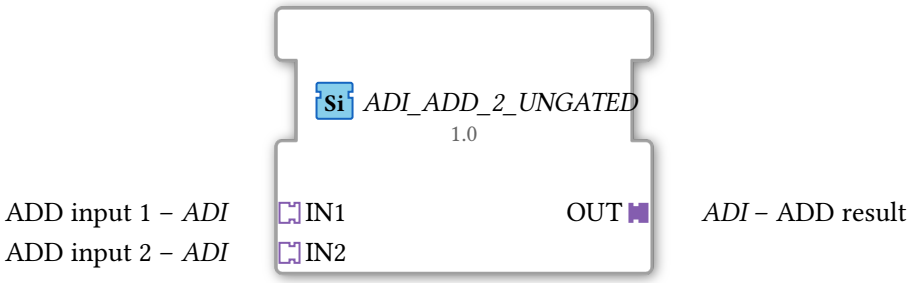
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ADI_ADD_2` is a highly efficient, clear, and modern component for arithmetic addition in the 4diac IDE. It is ideally suited for demanding architectures where clarity through the use of standardized adapters is paramount.

ADI_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_ADD_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_ADD_2_UNGATED` is a generic function block for performing arithmetic addition operations. Unlike conventional mathematical function blocks, this block uses an adapter-based interface concept (unidirectional `ADI` adapters) to transmit data and associated control signals in a bundled manner. It enables the addition of two input values to produce one output value.

Interface Structure

Event Inputs

This function block does not have direct, dedicated event inputs. Event control is handled entirely via the adapters.

Event Outputs

This function block does not have direct, dedicated event outputs. Event forwarding is handled entirely via the output adapter.

Data Inputs

This function block does not have any standard, direct data inputs.

Data Outputs

This function block does not have any standard, direct data outputs.

Adapters

Sockets (Inputs)

- **IN1**: Type `adapter::types::unidirectional::ADI`
- *Description:* First input value for mathematical addition.
- **IN2**: Type `adapter::types::unidirectional::ADI`
- *Description:* Second input value for mathematical addition.

Plugs (Connectors - Outputs)

- **OUT**: Type `adapter::types::unidirectional::ADI`
- *Description:* Output that provides the result of the addition (`IN1 + IN2`).

Functionality

The function block `ADI_ADD_2_UNGATED` performs the arithmetic operation:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.5}$$

As soon as data changes at the input adapters `IN1` or `IN2`, or a corresponding transmission event is signaled via the adapter structure, the function block processes the values. The sum is calculated and forwarded to subsequent function blocks via the corresponding event/data bundle of the output adapter `OUT`.

Since this is a generic function block (`GEN_ADI_ADD`), the specific data type used depends on the specifications and instantiation of the `ADI` adapters used.

Technical Features

- **Generic Type:** The function block is declared as `GEN_ADI_ADD`. This allows for flexible handling of different numeric data types, provided they are supported by the adapters used.
- **Adapter-Based Design:** By using unidirectional adapters of type `ADI`, the number of explicit connection lines (wiring overhead in the 4diac IDE's Application Editor) is drastically reduced, as data and synchronization events are encapsulated in a single connection.

•

State Overview

The function block behaves like a stateless (or purely event-driven combinational) function block:

1. **Waiting for Data Update:** The function block remains in idle state until new values are signaled via `IN1` or `IN2`.
2. **Calculation:** When an event arrives at the inputs, the mathematical sum is calculated.
3. **Output:** The result is directly passed to the output adapter `OUT`, and the corresponding trigger event is triggered.

–

Application Scenarios

- **Modular Signal Processing:** Addition of measured values (e.g., Sensor 1 + Sensor 2 to determine a total value) in systems that are consistently based on an adapter architecture.
- **Cascaded Computations:** Easy integration into complex arithmetic computation networks through clean, structured adapter connections.

Comparison with Similar Components

Compared to a classic `ADD` component according to IEC 61499 (which typically has dedicated `REQ` and `CNF` event ports, as well as direct data inputs such as `IN1` and `IN2` as `ANY_NUM`), the `ADI_ADD_2_UNGATED` encapsulates these interfaces in adapters.

- **Standard ADD:** Requires manual wiring of a minimum of 2 events and 3 data lines (5 connections in total).
- **ADI_ADD_2_UNGATED:** Only requires connecting the three adapter lines (`IN1`, `IN2`, `OUT`), which significantly improves the readability of complex control diagrams.

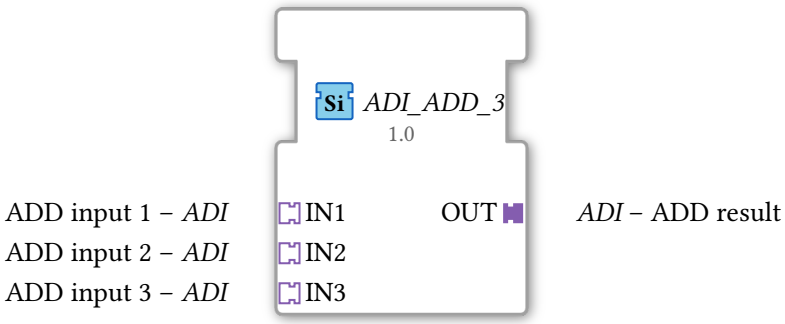
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ADI_ADD_2_UNGATED` is a highly efficient, clear, and modern component for arithmetic addition in the 4diac IDE. It is ideally suited for demanding architectures where clarity through the use of standardized adapters is paramount.

ADI_ADD_3



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_ADD_3` is a generic arithmetic block used to add three input values. It is based on the IEC 61499-2 standard and uses an adapter-based communication model. By using unidirectional adapters of type `ADI` (adapters for data integration/interaction), the block enables structured and clean signal and data routing within 4diac-ide.

Interface Structure

Event Inputs

There are no direct, standalone event inputs. Event control is implemented via the assigned adapter inputs (sockets).

Event Outputs

There are no direct, standalone event outputs. Event forwarding is implemented via the adapter output (plug).

Data Inputs

There are no direct data inputs. Data transmission occurs via the adapters.

Data Outputs

There are no direct data outputs. Data transmission occurs via the output adapter.

Adapters

Sockets (Input Interfaces)

- **IN1** (Type: `adapter::types::unidirectional::ADI`): First input summation for addition.
- **IN2** (Type: `adapter::types::unidirectional::ADI`): Second input summation.
- **IN3** (Type: `adapter::types::unidirectional::ADI`): Third input summand for addition.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::ADI`): Output for the calculated addition result ($OUT = IN1 + IN2 + IN3$).

Functionality

As soon as a new event or new data arrives at one of the input sockets (`IN1`, `IN2`, or `IN3`), the function block reads the values from the three adapters. It performs an arithmetic addition of the three values:

$$\text{Result} = \text{IN1} + \text{IN2} + \text{IN3} \tag{2.6}$$

The calculated result and the associated update event are then transmitted to the subsequent function blocks in the system via the output plug `OUT`.

Technical Features

- **Generic Function Block (FB):** Due to the attribute `eclipse4diac::core::GenericClassName` with the value `GEN_ADI_ADD`, this function block behaves polymorphically. It is not fixed to a specific data type (such as `INT` or `REAL`), but adapts flexibly to the data type defined in the adapter.
- **Adapter-Based Architecture:** Encapsulating data and events in the `ADI` adapters significantly reduces cable clutter in the 4diac IDE, as separate event and data lines are not required for each summand.

State Overview

The function block operates in a purely stateless (reactive) state:

1. **Idle State:** The block waits for trigger signals from the input sockets.
2. **Calculation & Forwarding:** Upon triggering at `IN1`, `IN2`, or `IN3`, the sum is calculated instantly and output via the `OUT` plug.

Application Scenarios

- **Sensor Consolidation:** Summing of three different physical quantities (e.g., three partial flows in a system to determine the total flow).
- **Setpoint Offset Calculation:** Combination of a main setpoint, a correction value, and an additional tolerance or offset value.
- **Process Value Calculation:** Balancing of inflows at three different measuring points in process engineering.
-

Comparison with Similar Components

- **Standard `ADD` (IEC 61131-3):** The classic ADD component uses direct pins for data and events. `ADI_ADD_3`, due to its use of unidirectional adapters, is better suited for modular and hierarchical architectures in IEC 61499.
- **Two-way adder (e.g., `ADI_ADD_2`):** While adding three values with standard function blocks would require cascading two blocks, `ADI_ADD_3` performs this task in a single step, saving resources and improving clarity.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

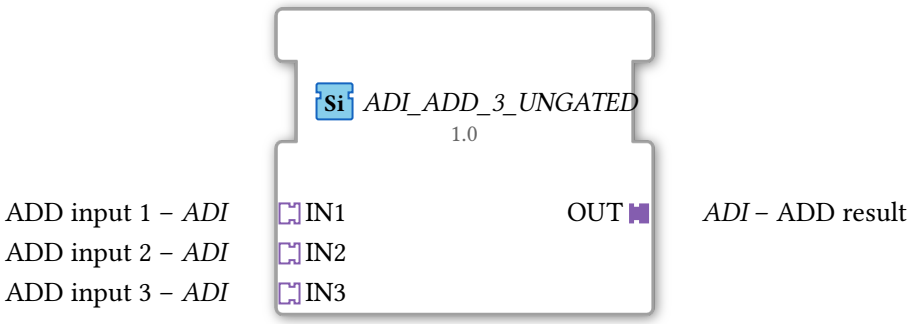
Conclusion

The `ADI_ADD_3` is a practical, generic auxiliary function block for arithmetic operations in modern IEC 61499 control applications. Its consistent use of adapters allows it to integrate seamlessly into clean software architectures and minimizes wiring effort in the development environment.

ADI_ADD_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/

frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_ADD_3_UNGATED` is a generic arithmetic block used to add three input values. It is based on the IEC 61499-2 standard and uses an adapter-based communication model. By using unidirectional adapters of type `ADI` (adapters for data integration/interaction), the block enables structured and clean signal and data routing within 4diac-ide.

Interface Structure

Event Inputs

There are no direct, standalone event inputs. Event control is implemented via the assigned adapter inputs (sockets).

Event Outputs

There are no direct, standalone event outputs. Event forwarding is implemented via the adapter output (plug).

Data Inputs

There are no direct data inputs. Data transmission occurs via the adapters.

Data Outputs

There are no direct data outputs. Data transmission occurs via the output adapter.

Adapters

Sockets (Input Interfaces)

- **IN1** (Type: `adapter::types::unidirectional::ADI`): First input summation for addition.
- **IN2** (Type: `adapter::types::unidirectional::ADI`): Second input summation.
- **IN3** (Type: `adapter::types::unidirectional::ADI`): Third input summand for addition.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::ADI`): Output for the calculated addition result ($OUT = IN1 + IN2 + IN3$).

Functionality

As soon as a new event or new data arrives at one of the input sockets (`IN1`, `IN2`, or `IN3`), the function block reads the values from the three adapters. It performs an arithmetic addition of the three values:

$$\text{Result} = \text{IN1} + \text{IN2} + \text{IN3} \tag{2.7}$$

The calculated result and the associated update event are then transmitted to the subsequent function blocks in the system via the output plug `OUT`.

Technical Features

- **Generic Function Block (FB):** Due to the attribute `eclipse4diac::core::GenericClassName` with the value `GEN_ADI_ADD`, this function block behaves polymorphically. It is not fixed to a specific data type (such as `INT` or `REAL`), but adapts flexibly to the data type defined in the adapter.
- **Adapter-Based Architecture:** Encapsulating data and events in the `ADI` adapters significantly reduces cable clutter in the 4diac IDE, as separate event and data lines are not required for each summand.

State Overview

The function block operates in a purely stateless (reactive) state:

1. **Idle State:** The block waits for trigger signals from the input sockets.
2. **Calculation & Forwarding:** Upon triggering at `IN1`, `IN2`, or `IN3`, the sum is calculated instantly and output via the `OUT` plug.

Application Scenarios

- **Sensor Consolidation:** Summing of three different physical quantities (e.g., three partial flows in a system to determine the total flow).
- **Setpoint Offset Calculation:** Combination of a main setpoint, a correction value, and an additional tolerance or offset value.
- **Process Value Calculation:** Balancing of inflows at three different measuring points in process engineering.
-

Comparison with Similar Components

- **Standard `ADD` (IEC 61131-3):** The classic ADD component uses direct pins for data and events. `ADI_ADD_3_UNGATED`, due to its use of unidirectional adapters, is better suited for modular and hierarchical architectures in IEC 61499.
- **Two-way adder (e.g., `ADI_ADD_2`):** While adding three values with standard function blocks would require cascading two blocks, `ADI_ADD_3_UNGATED` performs this task in a single step, saving resources and improving clarity.

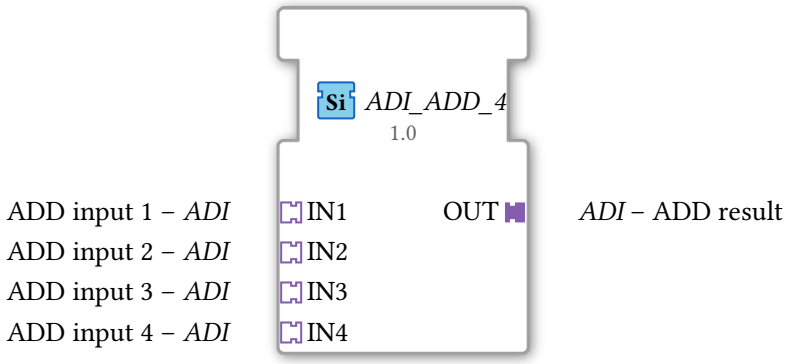
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ADI_ADD_3_UNGATED` is a practical, generic auxiliary function block for arithmetic operations in modern IEC 61499 control applications. Its consistent use of adapters allows it to integrate seamlessly into clean software architectures and minimizes wiring effort in the development environment.

ADI_ADD_4



(Image of function block not available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_ADD_4` performs arithmetic addition of up to four values. It is a generic function block (FB) based on the unidirectional adapter type `ADI`. It enables clear and modularized addition within IEC 61499 applications in the 4diac IDE.

Interface Structure

Since the block is entirely adapter-based, it does not have any traditional, separate event or data channels at the type's main level.

Event Inputs

No direct event inputs are available. Event control is implemented via the connected adapters.

Event Outputs

No direct event outputs are available. Event forwarding is implemented via the output adapter.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

- **Sockets (Input Adapters / Jacks):**
- `IN1` (Type: `adapter::types::unidirectional::ADI`): Input for the first summand.
- `IN2` (Type: `adapter::types::unidirectional::ADI`): Input for the second summand.
- `IN3` (Type: `adapter::types::unidirectional::ADI`): Input for the third addend.
- `IN4` (Type: `adapter::types::unidirectional::ADI`): Input for the fourth addend.
- **Plugs (Output Adapters):**
- `OUT` (Type: `adapter::types::unidirectional::ADI`): Output for the calculated total result of the addition.

Functionality

As soon as new data values or corresponding trigger events are present at the input adapters (`IN1` to `IN4`), the function block performs the addition of the values. The result is calculated using the formula:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.8}$$

The calculated result and the corresponding trigger event are provided directly via the output adapter `OUT`.

Technical Features

- **Generic Module:** By assigning the attribute `GenericClassName` with the value `'GEN_ADI_ADD'`, the module is flexibly structured. Depending on the implementation of the underlying `ADI` adapter, it can be used for various numeric data types.
- **Adapter-Based Design:** Encapsulating data and events in adapters (here of type `ADI`) drastically minimizes the wiring effort in the 4diac IDE function block diagram and ensures a clean, structured application design.
- **Unidirectional Coupling:** The use of unidirectional adapters ensures that data flow is strictly sequential from inputs to output.

State Overview

Since this is an adapter-based component for purely arithmetic data processing, the `ADI_ADD_4` does not have a complex internal state machine (ECC). Its operation is purely reactive:

- **Trigger on one of the inputs (`IN1` - `IN4`):** Calculates the sum of all four inputs and immediately forwards the result to `OUT`.

Application Scenarios

- **Sensor Data Aggregation:** Combines up to four analog measured values (e.g., four individual flow meters or temperature sensors) into a single total value.

- **Setpoint Calculation with Offsets:** Adds a primary setpoint with up to three different correction factors or offsets in control processes.
- **Structured Signal Processing:** Use in systems where the signal processing chain is consistently based on the `ADI` adapter standard to avoid unnecessary signal unpacking and repacking.

Comparison with Similar Function Blocks

- **Standard Adder (e.g., `F_ADD`):** A classic IEC 61131-3 adder block requires separate data lines and event connections. `ADI_ADD_4` bundles these in adapters, reducing visual complexity in the editor.
- **Dual Adder (`ADI_ADD_2`):** To add four values, three dual adders would have to be cascaded. `ADI_ADD_4` significantly saves space and execution time by combining the calculation into a single step.

Change Detection

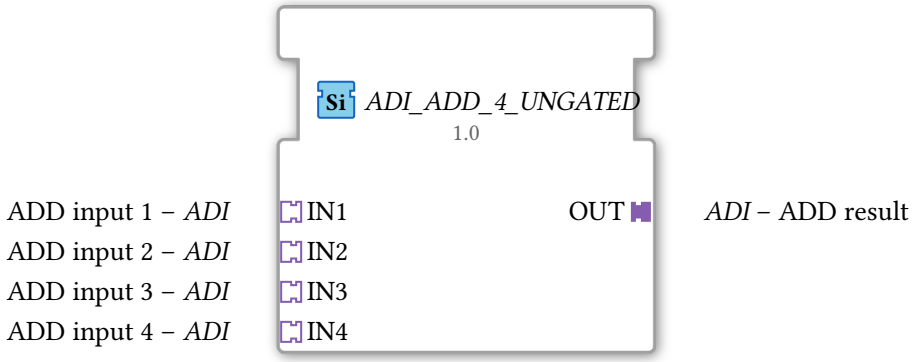
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ADI_ADD_4` is a high-performance and efficient function block for adding four numeric signals. By consistently using adapters, he reduces the number of physical connections in the control program and makes a significant contribution to a clear and maintenance-friendly software architecture.

ADI_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_ADD_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(Image of function block not available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_ADD_4_UNGATED` performs arithmetic addition of up to four values. It is a generic function block (FB) based on the unidirectional adapter type `ADI`. It enables clear and modularized addition within IEC 61499 applications in the 4diac IDE.

Interface Structure

Since the block is entirely adapter-based, it does not have any traditional, separate event or data channels at the type's main level.

Event Inputs

No direct event inputs are available. Event control is implemented via the connected adapters.

Event Outputs

No direct event outputs are available. Event forwarding is implemented via the output adapter.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

- **Sockets (Input Adapters / Jacks):**
- `IN1` (Type: `adapter::types::unidirectional::ADI`): Input for the first summand.
- `IN2` (Type: `adapter::types::unidirectional::ADI`): Input for the second summand.
- `IN3` (Type: `adapter::types::unidirectional::ADI`): Input for the third addend.
- `IN4` (Type: `adapter::types::unidirectional::ADI`): Input for the fourth addend.
- **Plugs (Output Adapters):**
- `OUT` (Type: `adapter::types::unidirectional::ADI`): Output for the calculated total result of the addition.

Functionality

As soon as new data values or corresponding trigger events are present at the input adapters (`IN1` to `IN4`), the function block performs the addition of the values. The result is calculated using the formula:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.9}$$

The calculated result and the corresponding trigger event are provided directly via the output adapter `OUT`.

Technical Features

- **Generic Module:** By assigning the attribute `GenericClassName` with the value `'GEN_ADI_ADD'`, the module is flexibly structured. Depending on the implementation of the underlying `ADI` adapter, it can be used for various numeric data types.
- **Adapter-Based Design:** Encapsulating data and events in adapters (here of type `ADI`) drastically minimizes the wiring effort in the 4diac IDE function block diagram and ensures a clean, structured application design.
- **Unidirectional Coupling:** The use of unidirectional adapters ensures that data flow is strictly sequential from inputs to output.

State Overview

Since this is an adapter-based component for purely arithmetic data processing, the `ADI_ADD_4_UNGATED` does not have a complex internal state machine (ECC). Its operation is purely reactive:

- **Trigger on one of the inputs (`IN1` - `IN4`):** Calculates the sum of all four inputs and immediately forwards the result to `OUT`.

Application Scenarios

- **Sensor Data Aggregation:** Combines up to four analog measured values (e.g., four individual flow meters or temperature sensors) into a single total value.
- **Setpoint Calculation with Offsets:** Adds a primary setpoint with up to three different correction factors or offsets in control processes.
- **Structured Signal Processing:** Use in systems where the signal processing chain is consistently based on the `ADI` adapter standard to avoid unnecessary signal unpacking and repacking.

Comparison with Similar Function Blocks

- **Standard Adder (e.g., `F_ADD`):** A classic IEC 61131-3 adder block requires separate data lines and event connections. `ADI_ADD_4_UNGATED` bundles these in adapters, reducing visual complexity in the editor.
- **Dual Adder (`ADI_ADD_2`):** To add four values, three dual adders would have to be cascaded. `ADI_ADD_4_UNGATED` significantly saves space and execution time by combining the calculation into a single step.

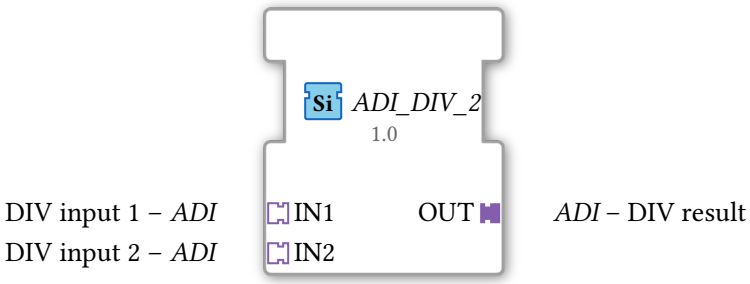
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ADI_ADD_4_UNGATED` is a high-performance and efficient function block for adding four numeric signals. By consistently using adapters, he reduces the number of physical connections in the control program and makes a significant contribution to a clear and maintenance-friendly software architecture.

ADI_DIV_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_DIV_2` performs an arithmetic division (DIV) of two input values via adapter connections. It is a generic function block (Generic FB) based on type `GEN_ADI_DIV`. The consistent use of unidirectional adapters of type `ADI` ensures clean encapsulation of data and events. This simplifies application design in the 4diac IDE, as the number of visible connection lines is significantly reduced.

Since this function block operates entirely on an adapter basis, it does not have traditional, discrete event or data connections at the top level. All interaction takes place via the adapter interfaces.

Interface Structure

Event Inputs

No direct event inputs are available. Event control is handled via the input adapters (`IN1`, `IN2`).

Event Outputs

No direct event outputs are available. Event output is handled via the output adapter (`OUT`).

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

- **Sockets (Input Adapters):**
- **`IN1`** (Type: `adapter::types::unidirectional::ADI`): First input value (dividend/counter) for division.
- **`IN2`** (Type: `adapter::types::unidirectional::ADI`): Second input value (divisor/denominator) for the division.
- **Plugs (Output Adapters):**
- **`OUT`** (Type: `adapter::types::unidirectional::ADI`): The calculated result (quotient) of the division.

Functionality

The function block divides the value received via adapter `IN1` by the value received via adapter `IN2`:

$$\text{Result (OUT)} = \frac{\text{Value of IN1}}{\text{Value of IN2}} \tag{2.10}$$

The calculation is triggered as soon as corresponding update events are received at the input adapters. After successful calculation, the new quotient, along with an output event, is passed via the adapter `OUT`. Due to the generic nature of the function block (`GEN_ADI_DIV`), its behavior adapts to the underlying data types of the `ADI` adapters used.

- **Generic Instantiation:** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_ADI_DIV'`. This allows for flexible typing at runtime, depending on the adapter subtype used.
- **Adapter Encapsulation:** Data and the associated trigger events are bundled in the adapter type `ADI`, which facilitates loose coupling of the software components.
- **Division by Zero:** During implementation and use, it is essential to ensure that the value at adapter `IN2` (divisor) is not zero. Division by zero must be handled in an application-oriented manner before the function block to prevent runtime errors.

The function block behaves in a stateless (i.e., purely reactive) manner:

Technical Features

1. **Wait State:** The function block waits for events at sockets `IN1` or `IN2`.
2. **Calculation:** After receiving valid data/events at the inputs, the division is performed.
3. **Update:** The result is passed to output plug `OUT`, and the corresponding send event is triggered.

State Overview

- **Measurement Scaling:** Normalization of analog sensor values (e.g., dividing a raw value by a scaling factor).
- **Calculation of Ratios:** Determination of mixing ratios or percentages in process engineering plants.
- **Average Calculations:** Division of a sum by the number of recorded values in simple mathematical computational networks.

Application Scenarios

Compared to a classic, standard `DIV` function block (according to IEC 61131-3), which uses discrete data pins (`IN1`, `IN2`, `OUT`) and explicit event pins (`REQ`, `CNF`):

Comparison with Similar Function Blocks

- **Advantages of `ADI_DIV_2`:** Significantly cleaner appearance in the 4diac function block diagram (FBD). Data consistency is directly enforced through the coupling in the adapter.
- **Disadvantages:** Direct use with standard data streams without adapter encapsulation is not possible. Dedicated converter modules must be used if the signal source does not support adapters.

Change Detection

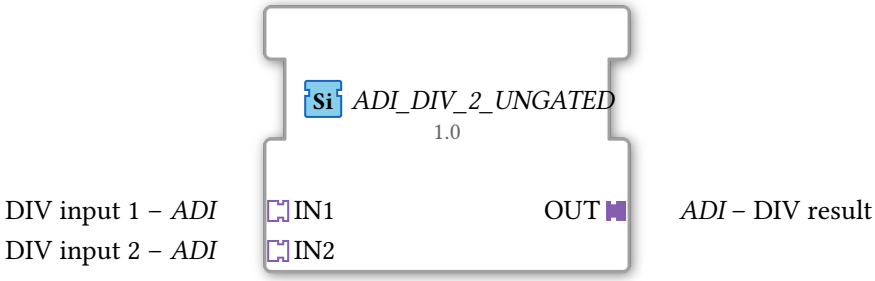
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

`ADI_DIV_2` is a specialized and efficient module for modern IEC 61499 architectures within 4diac. It is ideally suited for developers who rely on a clean, adapter-based, and modular software architecture and want to implement mathematical division in a clear and concise manner.

ADI_DIV_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_DIV_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_DIV_2_UNGATED` performs an arithmetic division (DIV) of two input values via adapter connections. It is a generic function block (Generic FB) based on type `GEN_ADI_DIV`. The consistent use of unidirectional adapters of type `ADI` ensures clean encapsulation of data and events. This simplifies application design in the 4diac IDE, as the number of visible connection lines is significantly reduced.

Since this function block operates entirely on an adapter basis, it does not have traditional, discrete event or data connections at the top level. All interaction takes place via the adapter interfaces.

Interface Structure

Event Inputs

No direct event inputs are available. Event control is handled via the input adapters (`IN1`, `IN2`).

Event Outputs

No direct event outputs are available. Event output is handled via the output adapter (`OUT`).

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

- **Sockets (Input Adapters):**
- **`IN1`** (Type: `adapter::types::unidirectional::ADI`): First input value (dividend/counter) for division.
- **`IN2`** (Type: `adapter::types::unidirectional::ADI`): Second input value (divisor/denominator) for the division.
- **Plugs (Output Adapters):**
- **`OUT`** (Type: `adapter::types::unidirectional::ADI`): The calculated result (quotient) of the division.

Functionality

The function block divides the value received via adapter `IN1` by the value received via adapter `IN2`:

$$\text{Result (OUT)} = \frac{\text{Value of IN1}}{\text{Value of IN2}} \tag{2.11}$$

The calculation is triggered as soon as corresponding update events are received at the input adapters. After successful calculation, the new quotient, along with an output event, is passed via the adapter `OUT`. Due to the generic nature of the function block (`GEN_ADI_DIV`), its behavior adapts to the underlying data types of the `ADI` adapters used.

- **Generic Instantiation:** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_ADI_DIV'`. This allows for flexible typing at runtime, depending on the adapter subtype used.
- **Adapter Encapsulation:** Data and the associated trigger events are bundled in the adapter type `ADI`, which facilitates loose coupling of the software components.

- **Division by Zero:** During implementation and use, it is essential to ensure that the value at adapter `IN2` (divisor) is not zero. Division by zero must be handled in an application-oriented manner before the function block to prevent runtime errors.

The function block behaves in a stateless (i.e., purely reactive) manner:

Technical Features

1. **Wait State:** The function block waits for events at sockets `IN1` or `IN2`.
2. **Calculation:** After receiving valid data/events at the inputs, the division is performed.
3. **Update:** The result is passed to output plug `OUT`, and the corresponding send event is triggered.

State Overview

- **Measurement Scaling:** Normalization of analog sensor values (e.g., dividing a raw value by a scaling factor).
- **Calculation of Ratios:** Determination of mixing ratios or percentages in process engineering plants.
- **Average Calculations:** Division of a sum by the number of recorded values in simple mathematical computational networks.

Application Scenarios

Compared to a classic, standard `DIV` function block (according to IEC 61131-3), which uses discrete data pins (`IN1`, `IN2`, `OUT`) and explicit event pins (`REQ`, `CNF`):

Comparison with Similar Function Blocks

- **Advantages of `ADI_DIV_2_UNGATED`:** Significantly cleaner appearance in the 4diac function block diagram (FBD). Data consistency is directly enforced through the coupling in the adapter.
- **Disadvantages:** Direct use with standard data streams without adapter encapsulation is not possible. Dedicated converter modules must be used if the signal source does not support adapters.

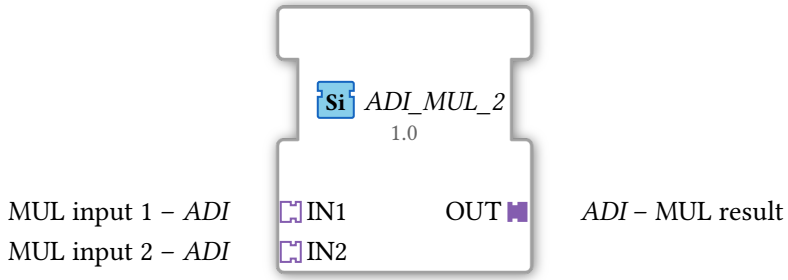
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

`ADI_DIV_2_UNGATED` is a specialized and efficient module for modern IEC 61499 architectures within 4diac. It is ideally suited for developers who rely on a clean, adapter-based, and modular software architecture and want to implement mathematical division in a clear and concise manner.

ADI_MUL_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_MUL_2` is a generic arithmetic multiplication block for the 4diac IDE, based on the IEC 61499 standard. It is used to multiply two input values. The unique feature of this block is the use of unidirectional adapters (`ADI`) for data transmission instead of traditional data and event connections. This enables modular and loosely coupled signal processing.

Interface Structure

Event Inputs

There are no direct event inputs. Control and synchronization are implicit via the adapters used.

Event Outputs

There are no direct event outputs. Event forwarding is coupled via the output adapter.

Data Inputs

The function block does not have traditional data inputs. Data acquisition is handled entirely via the adapter interfaces.

Data Outputs

The function block does not have traditional data outputs. Data output is handled via the output adapter.

Adapters

The function block uses adapters to encapsulate data and events:

- **Sockets (Input Adapters):**
- **IN1** (Type: `adapter::types::unidirectional::ADI`): First factor for multiplication (multiplicand).
- **IN2** (Type: `adapter::types::unidirectional::ADI`): Second factor for the multiplication (multiplier).
- **Plugs (Output Adapters):**
- **OUT** (Type: `adapter::types::unidirectional::ADI`): The result of the multiplication ($OUT = IN1 \times IN2$).

Functionality

As soon as valid values and the corresponding trigger events are received via the adapters `IN1` and `IN2`, the function block performs the arithmetic multiplication:

$$\$OUT = IN1 \times IN2$$

The result is immediately passed to the output adapter `OUT` and is available for subsequent function blocks. Since the function block is declared as a generic type (`GEN_ADI_MUL`), the exact data type width (e.g., INT, REAL, LREAL) is flexible and is determined during instantiation in the 4diac IDE based on the connected adapters.

Technical Features

- **Generic Implementation:** The `GenericClassName = 'GEN_ADI_MUL'` attribute makes the function block data type independent.
- **Adapter Encapsulation:** Using the unidirectional adapter `ADI` reduces the number of visible lines in the function block diagram because data and events are bundled and transferred in a single connection.
- **Compiler Context:** The function block is organized in the package `adapter::iec61131::arithmetic` and imports the class `eclipse4diac::core::GenericClassName`.

State Overview

The function block behaves in a purely stateless (or combinatorial) manner. There is no internal state machine (ECC). Each activation of the input adapters directly leads to the calculation of the output value and the updating of the output adapter `OUT`.

Application Scenarios

- **Modular Signal Processing:** Perfectly suited for control architectures that consistently rely on adapter connections to maintain the clarity of complex function plans.
- **Scalable Calculations:** Use in mathematical computing networks within IEC 61499 applications where different numerical data types need to be multiplied.

Comparison with Similar Function Blocks

Compared to a standard multiplication block (such as the classic `MUL` block from the IEC 61131-3 library), which uses explicit data pins (`IN1`, `IN2`, `OUT`) and events (`REQ`, `CNF`), the `ADI_MUL_2` eliminates the need for manual wiring of trigger events. This increases reusability and results in cleaner software design.

Change Detection

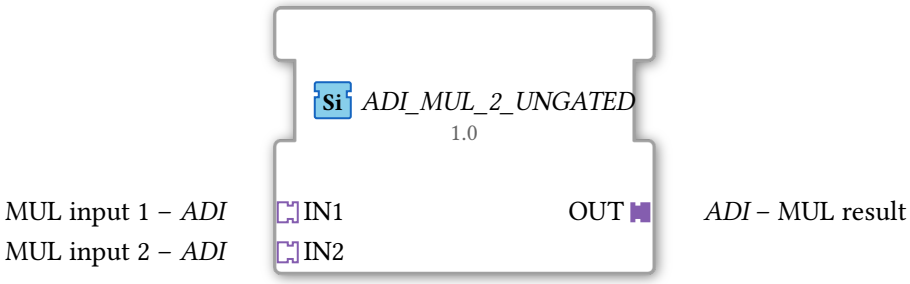
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ADI_MUL_2` is a modern, flexible, and easy-to-wire multiplication block. Through the consistent use of unidirectional adapters, it is ideally suited for service-oriented architectures in industrial automation and facilitates the creation of cleanly structured IEC 61499 applications.

ADI_MUL_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_MUL_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_MUL_2_UNGATED` is a generic arithmetic multiplication block for the 4diac IDE, based on the IEC 61499 standard. It is used to multiply two input values. The unique feature of this block is the use of unidirectional adapters (`ADI`) for data transmission instead of traditional data and event connections. This enables modular and loosely coupled signal processing.

Interface Structure

Event Inputs

There are no direct event inputs. Control and synchronization are implicit via the adapters used.

Event Outputs

There are no direct event outputs. Event forwarding is coupled via the output adapter.

Data Inputs

The function block does not have traditional data inputs. Data acquisition is handled entirely via the adapter interfaces.

Data Outputs

The function block does not have traditional data outputs. Data output is handled via the output adapter.

Adapters

The function block uses adapters to encapsulate data and events:

- **Sockets (Input Adapters):**
- **IN1** (Type: `adapter::types::unidirectional::ADI`): First factor for multiplication (multiplicand).
- **IN2** (Type: `adapter::types::unidirectional::ADI`): Second factor for the multiplication (multiplier).

- **Plugs (Output Adapters):**
- **OUT** (Type: `adapter::types::unidirectional::ADI`): The result of the multiplication ($OUT = IN1 \times IN2$).

Functionality

As soon as valid values and the corresponding trigger events are received via the adapters `IN1` and `IN2`, the function block performs the arithmetic multiplication:

$$OUT = IN1 \times IN2$$

The result is immediately passed to the output adapter `OUT` and is available for subsequent function blocks. Since the function block is declared as a generic type (`GEN_ADI_MUL`), the exact data type width (e.g., INT, REAL, LREAL) is flexible and is determined during instantiation in the 4diac IDE based on the connected adapters.

Technical Features

- **Generic Implementation:** The `GenericClassName = 'GEN_ADI_MUL'` attribute makes the function block data type independent.
- **Adapter Encapsulation:** Using the unidirectional adapter `ADI` reduces the number of visible lines in the function block diagram because data and events are bundled and transferred in a single connection.
- **Compiler Context:** The function block is organized in the package `adapter::iec61131::arithmetic` and imports the class `eclipse4diac::core::GenericClassName`.

State Overview

The function block behaves in a purely stateless (or combinatorial) manner. There is no internal state machine (ECC). Each activation of the input adapters directly leads to the calculation of the output value and the updating of the output adapter `OUT`.

Application Scenarios

- **Modular Signal Processing:** Perfectly suited for control architectures that consistently rely on adapter connections to maintain the clarity of complex function plans.
- **Scalable Calculations:** Use in mathematical computing networks within IEC 61499 applications where different numerical data types need to be multiplied.

Comparison with Similar Function Blocks

Compared to a standard multiplication block (such as the classic `MUL` block from the IEC 61131-3 library), which uses explicit data pins (`IN1`, `IN2`, `OUT`) and events (`REQ`, `CNF`), the `ADI_MUL_2_UNGATED` eliminates the need for manual wiring of trigger events. This increases reusability and results in cleaner software design.

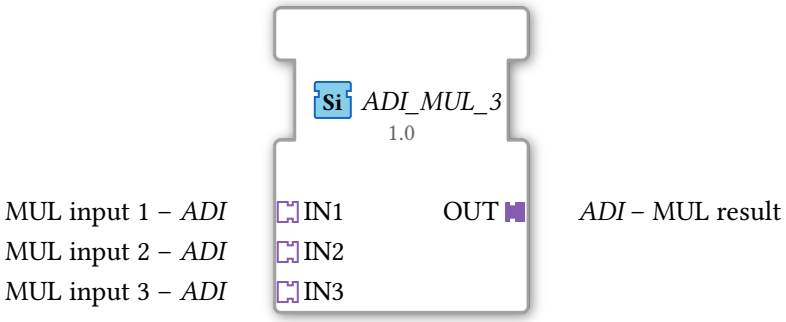
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ADI_MUL_2_UNGATED` is a modern, flexible, and easy-to-wire multiplication block. Through the consistent use of unidirectional adapters, it is ideally suited for service-oriented architectures in industrial automation and facilitates the creation of cleanly structured IEC 61499 applications.

ADI_MUL_3



No image available

• – ◦ • – ◦ • – ◦ •

Introduction

The `ADI_MUL_3` function block is a generic arithmetic function block for the 4diac IDE. It is used to multiply three numeric input values, which are transmitted via standardized, unidirectional adapters. By encapsulating the mathematical operation in an adapter interface, this function block is particularly suitable for modular and clean application architectures within the IEC 61499 standard.

Interface Structure

Because it is an adapter-based function block, the `ADI_MUL_3` does not have any direct, classic event or data pins at the top level. All communication is handled via the adapter interfaces.

Event Inputs

No direct event inputs available (control is handled via the adapters).

Event Outputs

No direct event outputs available (control is handled via the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ADI`)
- First multiplicand for the calculation operation.
- **IN2** (Type: `adapter::types::unidirectional::ADI`)
- Second multiplicand for the calculation operation.
- **IN3** (Type: `adapter::types::unidirectional::ADI`)
- Third multiplicand for the calculation operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ADI`)
- Output adapter that provides the mathematical product of the three input values.

Functionality

As soon as new values are signaled at the input adapters (`IN1`, `IN2`, `IN3`), the function block performs the multiplication. The result is calculated using the following formula:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.12}$$

The calculated result is immediately passed to the output adapter `OUT`, and a corresponding update event is forwarded via the adapter.

Technical Features

- **Generic Type:** The function block is based on the generic class `'GEN_ADI_MUL'`. This allows for flexible handling of different numeric data types (e.g., `INT`, `UINT`, `REAL`, `LREAL`), depending on how the underlying ADI adapters are implemented.
- **Generic Type:** The function block is based on the generic class `'GEN_ADI_MUL'`. This enables flexible handling of different numeric data types (e.g., `INT`, `UINT`, `REAL`, `LREAL`). * **Unidirectional Adapters:** Using the type `unidirectional::ADI` ensures a clear, one-way data flow, minimizing coupling between program components and increasing system stability.

State Overview

The function block is designed as a purely mathematical, stateless block. It does not store any internal history values. Every update to the inputs results in a direct recalculation of the output.

Application Scenarios

- **Volume Calculations:** Calculating the volume from three dimensions (length × width × height).
- **Three-Phase Measurements:** Power or energy calculations where several factors (e.g., current, voltage, and a scaling/correction factor) need to be multiplied together.
- **Multi-Level Scaling:** Reading a sensor value that needs to be multiplied by a calibration value and an additional gain factor.

Comparison with Similar Blocks

Compared to classic IEC 61131-3 `MUL` blocks, the `ADI_MUL_3` offers the advantage of encapsulating events and data logically in a single connection through the use of adapters. This significantly reduces wiring effort in the 4diac IDE. Compared to a dual multiplier (`ADI_MUL_2`), this block also eliminates the need to cascade blocks, which reduces execution time and makes the application diagram clearer.

- `ADI_MUL_3_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

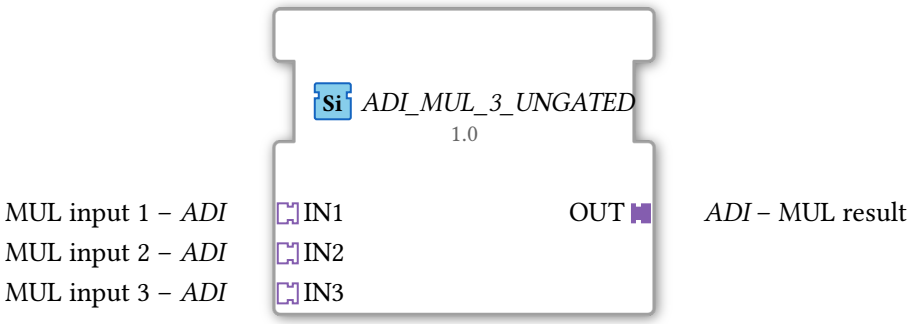
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ADI_MUL_3` is a practical auxiliary block for arithmetic calculations in IEC 61499 applications. Its consistent use of unidirectional adapters contributes to a clean, readable, and maintainable control design.

ADI_MUL_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_MUL_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



No image available

• – ◦ • – ◦ • – ◦ •

Introduction

The `ADI_MUL_3_UNGATED` function block is a generic arithmetic function block for the 4diac IDE. It is used to multiply three numeric input values, which are transmitted via standardized, unidirectional adapters. By encapsulating the mathematical operation in an adapter interface, this function block is particularly suitable for modular and clean application architectures within the IEC 61499 standard.

Interface Structure

Because it is an adapter-based function block, the `ADI_MUL_3_UNGATED` does not have any direct, classic event or data pins at the top level. All communication is handled via the adapter interfaces.

Event Inputs

No direct event inputs available (control is handled via the adapters).

Event Outputs

No direct event outputs available (control is handled via the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ADI`)
- First multiplicand for the calculation operation.
- **IN2** (Type: `adapter::types::unidirectional::ADI`)
- Second multiplicand for the calculation operation.
- **IN3** (Type: `adapter::types::unidirectional::ADI`)
- Third multiplicand for the calculation operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ADI`)
- Output adapter that provides the mathematical product of the three input values.

Functionality

As soon as new values are signaled at the input adapters (`IN1`, `IN2`, `IN3`), the function block performs the multiplication. The result is calculated using the following formula:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.13}$$

The calculated result is immediately passed to the output adapter `OUT`, and a corresponding update event is forwarded via the adapter.

Technical Features

- **Generic Type:** The function block is based on the generic class `'GEN_ADI_MUL'`. This allows for flexible handling of different numeric data types (e.g., `INT`, `UINT`, `REAL`, `LREAL`), depending on how the underlying ADI adapters are implemented.
- **Generic Type:** The function block is based on the generic class `'GEN_ADI_MUL'`. This enables flexible handling of different numeric data types (e.g., `INT`, `UINT`, `REAL`, `LREAL`). * **Unidirectional Adapters:** Using the type `unidirectional::ADI` ensures a clear, one-way data flow, minimizing coupling between program components and increasing system stability.

State Overview

The function block is designed as a purely mathematical, stateless block. It does not store any internal history values. Every update to the inputs results in a direct recalculation of the output.

Application Scenarios

- **Volume Calculations:** Calculating the volume from three dimensions (length × width × height).
- **Three-Phase Measurements:** Power or energy calculations where several factors (e.g., current, voltage, and a scaling/correction factor) need to be multiplied together.
- **Multi-Level Scaling:** Reading a sensor value that needs to be multiplied by a calibration value and an additional gain factor.

Comparison with Similar Blocks

Compared to classic IEC 61131-3 `MUL` blocks, the `ADI_MUL_3_UNGATED` offers the advantage of encapsulating events and data logically in a single connection through the use of adapters. This significantly reduces wiring effort in the 4diac IDE. Compared to a dual multiplier (`ADI_MUL_2`), this block also eliminates the need to cascade blocks, which reduces execution time and makes the application diagram clearer.

- **`ADI_MUL_3`**: The gated variant – updates the output only on an actual value change.

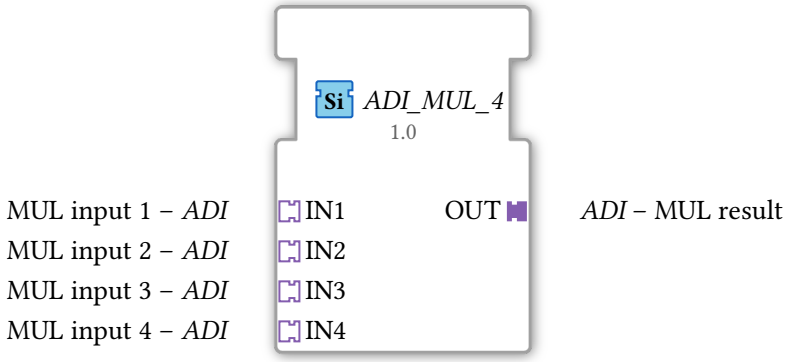
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ADI_MUL_3_UNGATED` is a practical auxiliary block for arithmetic calculations in IEC 61499 applications. Its consistent use of unidirectional adapters contributes to a clean, readable, and maintainable control design.

ADI_MUL_4



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_MUL_4` is a generic, adapter-based block for IEC 61499, used to perform arithmetic multiplications. It calculates the product of four input values provided via unidirectional ADI adapters (`adapter::types::unidirectional::ADI`) and outputs the result via a corresponding output adapter. By encapsulating signals in adapters, this block ensures a clearer application diagram in the 4diac IDE.

Interface Structure

Event Inputs

This block does not have direct event inputs. Event control is encapsulated within the adapters used.

Event Outputs

This block does not have direct event outputs. Event forwarding is encapsulated within the adapters used.

Data Inputs

This block does not have direct data inputs. Data is transferred via the sockets.

Data Outputs

This block does not have direct data outputs. Data is output via the plug.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ADI`): First factor for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::ADI`): Second factor for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::ADI`): Third factor for multiplication.

- **IN4** (Type: `adapter::types::unidirectional::ADI`): Fourth factor for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ADI`): Result of the multiplication ($OUT = IN1 \times IN2 \times IN3 \times IN4$).

Functionality

As soon as new values and their corresponding events arrive at the input adapters (`IN1` to `IN4`), the function block performs the multiplication of the four values:

$$OUT = \text{IN1} \cdot \text{IN2} \cdot \text{IN3} \cdot \text{IN4} \tag{2.14}$$

The calculated result, along with the corresponding trigger event, is forwarded via the output adapter `OUT` to the subsequent function blocks. Since this is a generic function block (`GEN_ADI_MUL`), the calculation behavior adapts to the data type defined in the adapter.

Technical Features

- **Generic Type (`GEN_ADI_MUL`):** The function block is internally declared as a generic function block. This allows for flexible handling of various data types (e.g., `INT`, `REAL`, `LREAL`), depending on the specifications of the connected ADI adapters.
- **Encapsulation by Adapters:** Since both data and events are routed via unidirectional adapters (`ADI`), the number of visible connection lines in the 4diac IDE is drastically reduced. This improves the clarity of complex control applications.

State Overview

The function block `ADI_MUL_4` is a **stateless** function block. It does not have an internal State Machine (ECC). Each activation by an event at the input adapters directly leads to a recalculation of the output based on the currently applied input values.

Application Scenarios

- **Physical Calculations:** Calculation of volumes (V = l × b × h) with an additional scaling or correction factor.
- **Measurement Scaling:** Multi-level weighting or scaling of analog sensor values across multiple factors.
- **Power Calculations:** Multiplication of various electrical or mechanical parameters to determine total power or efficiency in process automation.

Comparison with Similar Function Blocks

- **Standard `MUL` function block (IEC 61131-3):** Classic multipliers use direct data and event pins. `ADI_MUL_4`, on the other hand, uses adapters, which simplifies routing but requires a corresponding adapter infrastructure in the project. * **ADI_MUL_2 / ADI_MUL_3:** Similar function blocks with fewer inputs. The `ADI_MUL_4` is particularly suitable when exactly four factors need to be multiplied together without cascading multiple 2x multipliers.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

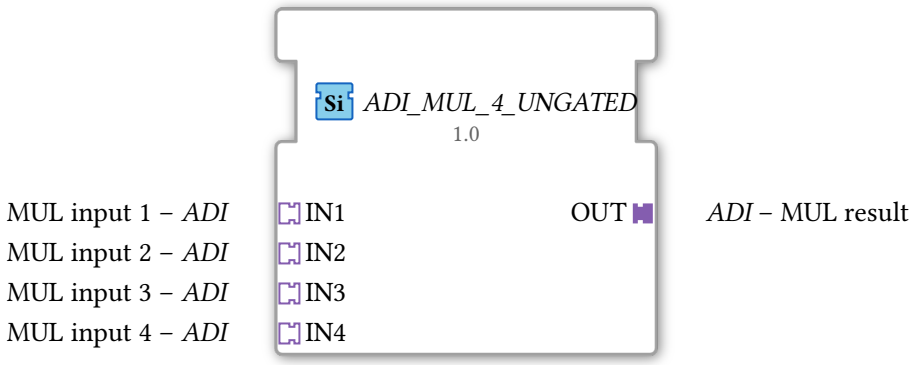
Conclusion

The `ADI_MUL_4` is a highly efficient auxiliary function block for mathematical signal processing in IEC 61499 systems. Through the consistent use of unidirectional adapters, it promotes clean, modular software design and reduces the complexity of graphical wiring within the 4diac IDE.

ADI_MUL_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_MUL_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/

frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_MUL_4_UNGATED` is a generic, adapter-based block for IEC 61499, used to perform arithmetic multiplications. It calculates the product of four input values provided via unidirectional ADI adapters (`adapter::types::unidirectional::ADI`) and outputs the result via a corresponding output adapter. By encapsulating signals in adapters, this block ensures a clearer application diagram in the 4diac IDE.

Interface Structure

Event Inputs

This block does not have direct event inputs. Event control is encapsulated within the adapters used.

Event Outputs

This block does not have direct event outputs. Event forwarding is encapsulated within the adapters used.

Data Inputs

This block does not have direct data inputs. Data is transferred via the sockets.

Data Outputs

This block does not have direct data outputs. Data is output via the plug.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ADI`): First factor for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::ADI`): Second factor for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::ADI`): Third factor for multiplication.
- **IN4** (Type: `adapter::types::unidirectional::ADI`): Fourth factor for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ADI`): Result of the multiplication ($OUT = IN1 \times IN2 \times IN3 \times IN4$).

Functionality

As soon as new values and their corresponding events arrive at the input adapters (`IN1` to `IN4`), the function block performs the multiplication of the four values:

$$OUT = \mathrm{IN1} \cdot \mathrm{IN2} \cdot \mathrm{IN3} \cdot \mathrm{IN4} \tag{2.15}$$

The calculated result, along with the corresponding trigger event, is forwarded via the output adapter `OUT` to the subsequent function blocks. Since this is a generic function block (`GEN_ADI_MUL`), the calculation behavior adapts to the data type defined in the adapter.

Technical Features

- **Generic Type (`GEN_ADI_MUL`):** The function block is internally declared as a generic function block. This allows for flexible handling of various data types (e.g., `INT`, `REAL`, `LREAL`), depending on the specifications of the connected ADI adapters.
- **Encapsulation by Adapters:** Since both data and events are routed via unidirectional adapters (`ADI`), the number of visible connection lines in the 4diac IDE is drastically reduced. This improves the clarity of complex control applications.

State Overview

The function block `ADI_MUL_4_UNGATED` is a **stateless** function block. It does not have an internal State Machine (ECC). Each activation by an event at the input adapters directly leads to a recalculation of the output based on the currently applied input values.

Application Scenarios

- **Physical Calculations:** Calculation of volumes (V = l × b × h) with an additional scaling or correction factor.
- **Measurement Scaling:** Multi-level weighting or scaling of analog sensor values across multiple factors.
- **Power Calculations:** Multiplication of various electrical or mechanical parameters to determine total power or efficiency in process automation.

Comparison with Similar Function Blocks

- **Standard `MUL` function block (IEC 61131-3):** Classic multipliers use direct data and event pins. `ADI_MUL_4_UNGATED`, on the other hand, uses adapters, which simplifies routing but requires a corresponding adapter infrastructure in the project. * **ADI_MUL_2 / ADI_MUL_3:** Similar function blocks with fewer inputs. The `ADI_MUL_4_UNGATED` is particularly suitable when exactly four factors need to be multiplied together without cascading multiple 2x multipliers.

Change Detection

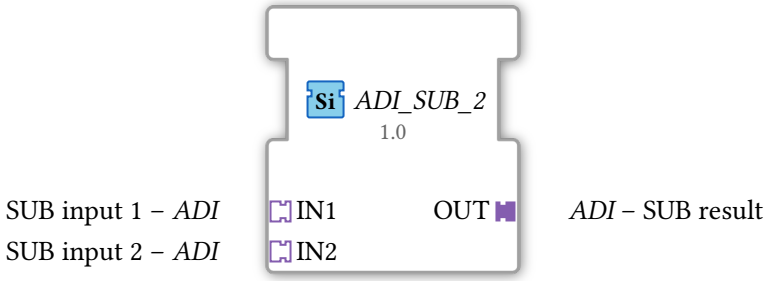
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ADI_MUL_4_UNGATED` is a highly efficient auxiliary function block for mathematical signal processing in IEC 61499 systems. Through the consistent use of unidirectional adapters, it promotes clean, modular software design and reduces the complexity of graphical wiring within the 4diac IDE.

ADI_SUB_2

```
    +-----------------------------------+
| ADI_SUB_2 | | | IN1 ==# [ADI] [ADI] #== OUT IN2 ==# [ADI] | +—————————————+
```




```
• – ◦ • – ◦ • – ◦ •
```

Introduction

The function block `ADI_SUB_2` is a generic function block (Generic FB) for performing arithmetic subtractions in IEC 61499 applications. The block uses unidirectional adapters of type `ADI` to receive input data and output the calculated result. Using adapters simplifies the wiring within the 4diac IDE and increases modularity.

Interface Structure

Event Inputs

The function block does not have direct event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

The function block does not have direct event outputs. Event forwarding is handled via the output adapter.

Data Inputs

No direct data inputs are defined. Data is passed via the input adapters.

Data Outputs

No direct data outputs are defined. The result is provided via the output adapter.

Adapters

Type	Name	Adapter Type	Description
Socket (Input)	`IN1`	`adapter::types::unidirectional::ADI`	First input value (minuend) for subtraction.
Socket (Input)	`IN2`	`adapter::types::unidirectional::ADI`	Second input value (subtrahend) for subtraction.
Plug (Output)	`OUT`	`adapter::types::unidirectional::ADI`	Result (difference) of the subtraction ($OUT = IN1 - IN2$).

Functionality

As soon as a corresponding calculation event is signaled at the input adapters `IN1` or `IN2`, the function block performs the subtraction:

$$\$OUT = IN1 - IN2$$

The result is passed to the output adapter `OUT`, and the corresponding output event of the adapter is triggered to inform subsequent function blocks of the new value.

- **Generic Type:** The function block is defined as a generic type (`eclipse4diac::core::GenericClassName` with the value `'GEN_ADI_SUB'`). This allows for flexible adaptation to different data types within the `ADI` adapter.
- **Unidirectional Adapters:** By using the unidirectional adapter structure (`adapter::types::unidirectional::ADI`), the data flow remains directed and cyclic dependencies are avoided.

Since this is a mathematical calculation function block, its behavior is essentially stateless and event-driven:

Technical Features

1. **Waiting:** The function block waits for a trigger event at one of the input adapters (`IN1` or `IN2`).
2. **Calculation:** Upon event input, the data values are read and subtracted.
3. **Output:** The calculated value is written to the adapter `OUT`, triggering the corresponding send event.

State Overview

- **Control Engineering:** Calculation of a control deviation ($e = w - x$), where the setpoint (w) and actual value (x) are transmitted via standardized ADI interfaces.
- **Measurement Processing:** Differential pressure measurement or offset correction of sensor values integrated into the application via adapters.

- **Signal Preprocessing:** Reduction of signal values by fixed or variable limits before forwarding them to actuators.

Application Scenarios

Compared to a standard subtraction block (such as the classic `SUB` block according to IEC 61131-3), which uses direct data and event pins, `ADI_SUB_2` offers the following advantages:

Comparison with Similar Function Blocks

- **Fewer connection lines:** Data and events are bundled in the `ADI` adapter, making the application diagram in 4diac clearer.
- **Standardized interfaces:** Facilitates the exchange of calculation and processing blocks, as only the adapters need to be connected.

Change Detection

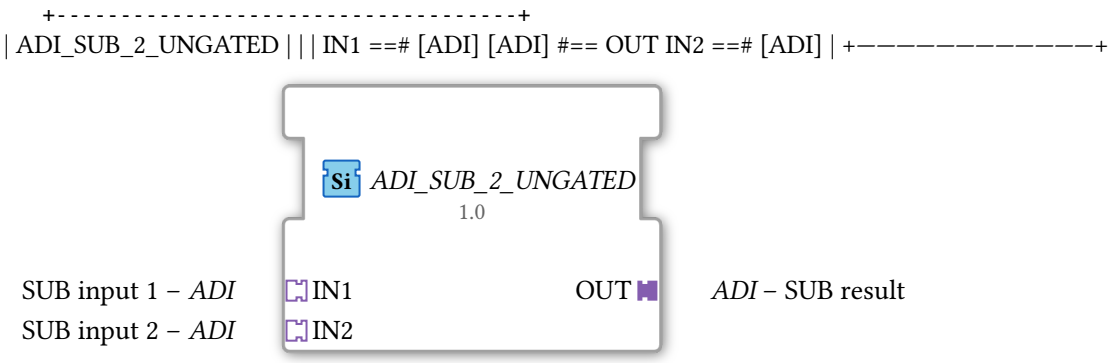
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ADI_SUB_2` is an efficient auxiliary block for structured IEC 61499 control applications. It is ideally suited for demanding architectures where clarity and standardized adapter interfaces are paramount.

ADI_SUB_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_SUB_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADI_SUB_2_UNGATED` is a generic function block (Generic FB) for performing arithmetic subtractions in IEC 61499 applications. The block uses unidirectional adapters of type `ADI` to receive input data and output the calculated result. Using adapters simplifies the wiring within the 4diac IDE and increases modularity.

Interface Structure

Event Inputs

The function block does not have direct event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

The function block does not have direct event outputs. Event forwarding is handled via the output adapter.

Data Inputs

No direct data inputs are defined. Data is passed via the input adapters.

Data Outputs

No direct data outputs are defined. The result is provided via the output adapter.

Adapters

Type	Name	Adapter Type	Description
Socket (Input)	`IN1`	`adapter::types::unidirectional::ADI`	First input value (minuend) for subtraction.
Socket (Input)	`IN2`	`adapter::types::unidirectional::ADI`	Second input value (subtrahend) for subtraction.
Plug (Output)	`OUT`	`adapter::types::unidirectional::ADI`	Result (difference) of the subtraction ($OUT = IN1 - IN2$).

Functionality

As soon as a corresponding calculation event is signaled at the input adapters `IN1` or `IN2`, the function block performs the subtraction:

$$OUT = IN1 - IN2$$

The result is passed to the output adapter `OUT`, and the corresponding output event of the adapter is triggered to inform subsequent function blocks of the new value.

- **Generic Type:** The function block is defined as a generic type (`eclipse4diac::core::GenericClassName` with the value `'GEN_ADI_SUB'`). This allows for flexible adaptation to different data types within the `ADI` adapter.
- **Unidirectional Adapters:** By using the unidirectional adapter structure (`adapter::types::unidirectional::ADI`), the data flow remains directed and cyclic dependencies are avoided.

Since this is a mathematical calculation function block, its behavior is essentially stateless and event-driven:

Technical Features

1. **Waiting:** The function block waits for a trigger event at one of the input adapters (`IN1` or `IN2`).
2. **Calculation:** Upon event input, the data values are read and subtracted.
3. **Output:** The calculated value is written to the adapter `OUT`, triggering the corresponding send event.

State Overview

- **Control Engineering:** Calculation of a control deviation ($e = w - x$), where the setpoint (w) and actual value (x) are transmitted via standardized ADI interfaces.
- **Measurement Processing:** Differential pressure measurement or offset correction of sensor values integrated into the application via adapters.
- **Signal Preprocessing:** Reduction of signal values by fixed or variable limits before forwarding them to actuators.

Application Scenarios

Compared to a standard subtraction block (such as the classic `SUB` block according to IEC 61131-3), which uses direct data and event pins, `ADI_SUB_2_UNGATED` offers the following advantages:

Comparison with Similar Function Blocks

- **Fewer connection lines:** Data and events are bundled in the `ADI` adapter, making the application diagram in 4diac clearer.
- **Standardized interfaces:** Facilitates the exchange of calculation and processing blocks, as only the adapters need to be connected.

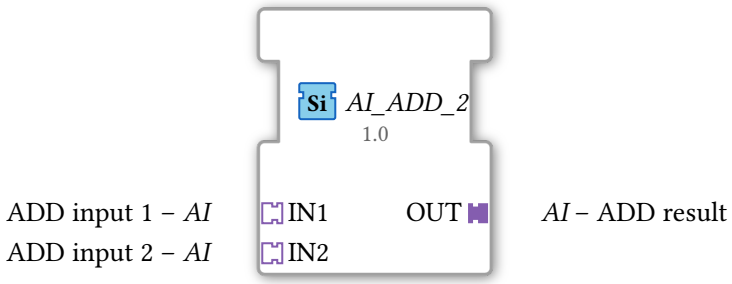
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ADI_SUB_2_UNGATED` is an efficient auxiliary block for structured IEC 61499 control applications. It is ideally suited for demanding architectures where clarity and standardized adapter interfaces are paramount.

AI_ADD_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) **AI_ADD_2** is a generic block for performing the arithmetic addition of two analog input values. It is based on the concept of generic function blocks (`GEN_AI_ADD`) and uses adapter interfaces instead of classic, loosely coupled data and event connections. This enables structured, clear, and modular signal processing within 4diac-ide-based control applications.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is encapsulated via the adapters used.

Event Outputs

There are no direct event outputs. Event control is encapsulated via the adapters used.

Data Inputs

There are no direct data inputs.

Data Outputs

There are no direct data outputs.

Adapters

Sockets (Input Interfaces)

- **IN1** (Type: `adapter::types::unidirectional::AI`): The first addend of the addition operation. Both the value and the corresponding update event are received via this adapter.
- **IN2** (Type: `adapter::types::unidirectional::AI`): The second addend of the addition operation.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::AI`): The result of the addition ($OUT = IN1 + IN2$). This adapter passes the result and the corresponding update event to subsequent function blocks.

Functionality

The function block **AI_ADD_2** functions as a mathematical adder. As soon as a new event is signaled at one of the input adapters (`IN1` or `IN2`) indicating that the input data has been updated, the function block performs the addition:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.16}$$

The calculated result is immediately passed to the output adapter `OUT`, and the corresponding update event is triggered to inform subsequent program parts of the new value.

Since this is a generic function block (`GEN_AI_ADD`), the underlying data type (e.g., `REAL`, `INT`, `LREAL`) is flexible and determined by the configuration of the connected adapters.

Technical Features

- **Generic Type (`GEN_AI_ADD`):** The function block is not tied to a specific data type, which increases its reusability for different analog signal ranges.
- **Unidirectional Adapters:** Using the `adapter::types::unidirectional::AI` type drastically reduces wiring complexity in the 4diac-ide, as values and events are bundled in a single connection.
- – **Clean Encapsulation:** No loose event and data lines on the surface of the function block.

State Overview

The function block operates in an event-driven and stateless manner. No internal state is maintained (as with a memory element). Each incoming event at the inputs immediately triggers recalculation and output at output `OUT`.

Application Scenarios

- **Sensor Data Aggregation:** Adding two flow rates (e.g., main flow and bypass flow) to obtain a total flow rate.
- **Setpoint Offsetting:** Adding a fixed or dynamic offset (e.g., calibration value) to an analog input signal.
- **Signal Mixing:** Simple mathematical combination of two analog control signals in process automation.

Comparison with Similar Function Blocks

Compared to the standard IEC 61131-3 add block (`ADD`), which uses classic data pins (e.g., `IN1`, `IN2` of type `ANY_NUM`) and explicit control events (`REQ` / `CNF`), **AI_ADD_2** offers a significantly cleaner visual representation. Encapsulation within adapters eliminates the need to manually draw separate event lines, minimizing the risk of errors during application development.

Change Detection

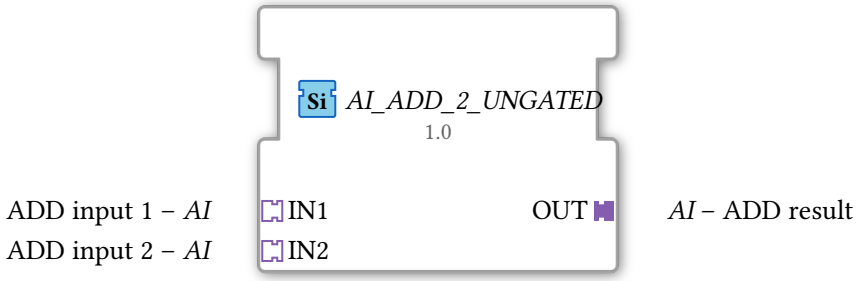
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The **AI_ADD_2** is a modern, highly efficient function block for the 4diac-ide. It combines the fundamental mathematical function of addition with the structural advantages of IEC 61499 adapters. This makes it the ideal choice for modular control architectures.

AI_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_ADD_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) **AI_ADD_2_UNGATED** is a generic block for performing the arithmetic addition of two analog input values. It is based on the concept of generic function blocks (`GEN_AI_ADD`) and uses adapter interfaces instead of classic, loosely coupled data and event connections. This enables structured, clear, and modular signal processing within 4diac-ide-based control applications.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is encapsulated via the adapters used.

Event Outputs

There are no direct event outputs. Event control is encapsulated via the adapters used.

Data Inputs

There are no direct data inputs.

Data Outputs

There are no direct data outputs.

Adapters

Sockets (Input Interfaces)

- **IN1** (Type: `adapter::types::unidirectional::AI`): The first addend of the addition operation. Both the value and the corresponding update event are received via this adapter.
- **IN2** (Type: `adapter::types::unidirectional::AI`): The second addend of the addition operation.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::AI`): The result of the addition ($OUT = IN1 + IN2$). This adapter passes the result and the corresponding update event to subsequent function blocks.

Functionality

The function block **AI_ADD_2_UNGATED** functions as a mathematical adder. As soon as a new event is signaled at one of the input adapters (`IN1` or `IN2`) indicating that the input data has been updated, the function block performs the addition:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.17}$$

The calculated result is immediately passed to the output adapter `OUT`, and the corresponding update event is triggered to inform subsequent program parts of the new value.

Since this is a generic function block (`GEN_AI_ADD`), the underlying data type (e.g., `REAL`, `INT`, `LREAL`) is flexible and determined by the configuration of the connected adapters.

Technical Features

- **Generic Type (`GEN_AI_ADD`):** The function block is not tied to a specific data type, which increases its reusability for different analog signal ranges.

- **Unidirectional Adapters:** Using the `adapter::types::unidirectional::AI` type drastically reduces wiring complexity in the 4diac-ide, as values and events are bundled in a single connection.
- – **Clean Encapsulation:** No loose event and data lines on the surface of the function block.

State Overview

The function block operates in an event-driven and stateless manner. No internal state is maintained (as with a memory element). Each incoming event at the inputs immediately triggers recalculation and output at output `OUT`.

Application Scenarios

- **Sensor Data Aggregation:** Adding two flow rates (e.g., main flow and bypass flow) to obtain a total flow rate.
- **Setpoint Offsetting:** Adding a fixed or dynamic offset (e.g., calibration value) to an analog input signal.
- **Signal Mixing:** Simple mathematical combination of two analog control signals in process automation.

Comparison with Similar Function Blocks

Compared to the standard IEC 61131-3 add block (`ADD`), which uses classic data pins (e.g., `IN1`, `IN2` of type `ANY_NUM`) and explicit control events (`REQ` / `CNF`), **AI_ADD_2_UNGATED** offers a significantly cleaner visual representation. Encapsulation within adapters eliminates the need to manually draw separate event lines, minimizing the risk of errors during application development.

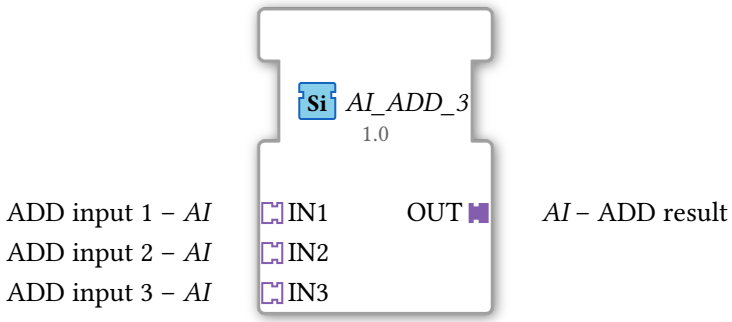
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The **AI_ADD_2_UNGATED** is a modern, highly efficient function block for the 4diac-ide. It combines the fundamental mathematical function of addition with the structural advantages of IEC 61499 adapters. This makes it the ideal choice for modular control architectures.

AI_ADD_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_ADD_3` is a generic, arithmetic addition block for the 4diac-ide development environment. It is used to add the values of three analog inputs (AI) and output the result to an analog output. The use of unidirectional adapters ensures clean encapsulation of data and associated events.

Interface Structure

The block uses an adapter-based interface design. It does not have direct, traditional event or data channels at the main level, but delegates these entirely to the adapters.

Event Inputs

No direct event inputs are available. (Event control is handled via the adapter interfaces).

Event Outputs

No direct event outputs available. (Event control is handled via the adapter interfaces).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AI`): First analog input value for the addition operation.
- **IN2** (Type: `adapter::types::unidirectional::AI`): Second analog input value for the addition operation.
- **IN3** (Type: `adapter::types::unidirectional::AI`): Third analog input value for the addition operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AI`): Output adapter that provides the calculated result of the addition.

Functionality

As soon as an update event arrives at one of the input adapters (`IN1`, `IN2`, or `IN3`), the function block reads the analog values of the three sockets.

The following mathematical operation is performed:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.18}$$

After successful calculation, the result value is passed to the output plug `OUT`, and a corresponding update event is triggered at the output.

Technical Features

- **Generic Function Block (FB):** The function block is internally typed as `GEN_AI_ADD`. This allows for flexible adaptation to different numeric data types (e.g., `INT`, `REAL`, `LREAL`), depending on the specific type defined by the connected adapters.
- **Adapter Coupling:** By using unidirectional adapters (`unidirectional::AI`), signal routing in the application diagram is greatly simplified. Data and trigger events are bundled in a single connection line.
- **Adapter Coupling:** The use of unidirectional adapters (`unidirectional::AI`) greatly simplifies signal routing in the application diagram. —

State Overview

The function block operates purely event-driven:

1. **Standby State (Idle):** The function block waits for incoming events at sockets `IN1`, `IN2`, or `IN3`.
2. **Calculation Phase:** An event triggers the addition of the incoming values.
3. **Update Phase:** The result is applied to `OUT`, and the output event is triggered. The function block immediately returns to standby state.

–

Application Scenarios

- **Sensor Data Fusion:** Combining and summing three analog measured values (e.g., determining a total volume flow from three individual partial flows).
- **Setpoint Calculation:** Calculation of an overall setpoint consisting of a base setpoint and two correction or offset values.
- **Mixing Processes:** Weighting and combining analog control signals in process automation.

Comparison with Similar Function Blocks

Compared to classic IEC 61131-3 adders (such as `F_ADD`) or standard IEC 61499 function blocks, `AI_ADD_3` eliminates the manual and error-prone wiring of separate `REQ` and `CNF` event lines.

While standard adders typically only support two inputs, the `AI_ADD_3` eliminates the need to cascade two separate modules when adding three values, thus optimizing execution time and improving system clarity.

Change Detection

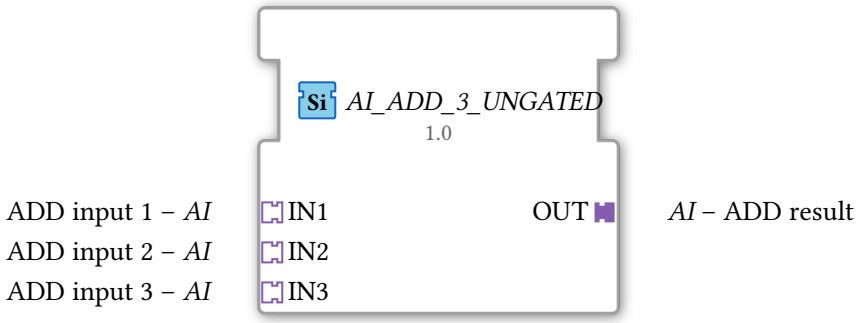
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AI_ADD_3` is a modular and user-friendly auxiliary module for efficient analog signal processing. Its consistent use of adapters significantly reduces the visual "spaghetti wiring" in 4diac applications and contributes to a high-performance, event-driven control architecture.

AI_ADD_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_ADD_3_UNGATED` is a generic, arithmetic addition block for the 4diac-ide development environment. It is used to add the values of three analog inputs (AI) and output the result to an analog output. The use of unidirectional adapters ensures clean encapsulation of data and associated events.

Interface Structure

The block uses an adapter-based interface design. It does not have direct, traditional event or data channels at the main level, but delegates these entirely to the adapters.

Event Inputs

No direct event inputs are available. (Event control is handled via the adapter interfaces).

Event Outputs

No direct event outputs available. (Event control is handled via the adapter interfaces).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AI`): First analog input value for the addition operation.
- **IN2** (Type: `adapter::types::unidirectional::AI`): Second analog input value for the addition operation.
- **IN3** (Type: `adapter::types::unidirectional::AI`): Third analog input value for the addition operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AI`): Output adapter that provides the calculated result of the addition.

Functionality

As soon as an update event arrives at one of the input adapters (`IN1`, `IN2`, or `IN3`), the function block reads the analog values of the three sockets.

The following mathematical operation is performed:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.19}$$

After successful calculation, the result value is passed to the output plug `OUT`, and a corresponding update event is triggered at the output.

Technical Features

- **Generic Function Block (FB):** The function block is internally typed as `GEN_AI_ADD`. This allows for flexible adaptation to different numeric data types (e.g., `INT`, `REAL`, `LREAL`), depending on the specific type defined by the connected adapters.
- **Adapter Coupling:** By using unidirectional adapters (`unidirectional::AI`), signal routing in the application diagram is greatly simplified. Data and trigger events are bundled in a single connection line.
- **Adapter Coupling:** The use of unidirectional adapters (`unidirectional::AI`) greatly simplifies signal routing in the application diagram. —

State Overview

The function block operates purely event-driven:

1. **Standby State (Idle):** The function block waits for incoming events at sockets `IN1`, `IN2`, or `IN3`.
2. **Calculation Phase:** An event triggers the addition of the incoming values.
3. **Update Phase:** The result is applied to `OUT`, and the output event is triggered. The function block immediately returns to standby state.

–

Application Scenarios

- **Sensor Data Fusion:** Combining and summing three analog measured values (e.g., determining a total volume flow from three individual partial flows).
- **Setpoint Calculation:** Calculation of an overall setpoint consisting of a base setpoint and two correction or offset values.
- **Mixing Processes:** Weighting and combining analog control signals in process automation.

Comparison with Similar Function Blocks

Compared to classic IEC 61131-3 adders (such as `F_ADD`) or standard IEC 61499 function blocks, `AI_ADD_3_UNGATED` eliminates the manual and error-prone wiring of separate `REQ` and `CNF` event lines.

While standard adders typically only support two inputs, the `AI_ADD_3_UNGATED` eliminates the need to cascade two separate modules when adding three values, thus optimizing execution time and improving system clarity.

Change Detection

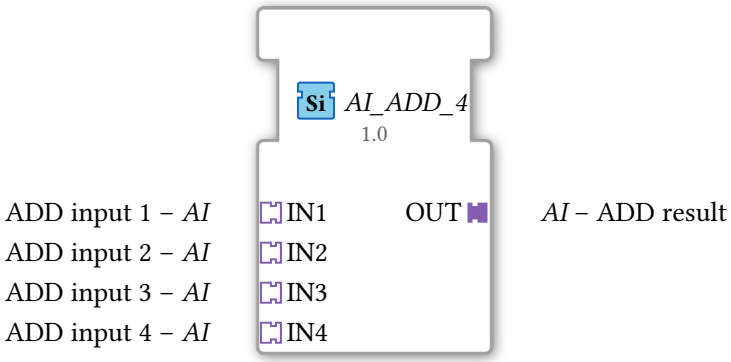
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AI_ADD_3_UNGATED` is a modular and user-friendly auxiliary module for efficient analog signal processing. Its consistent use of adapters significantly reduces the visual "spaghetti wiring" in 4diac applications and contributes to a high-performance, event-driven control architecture.

AI_ADD_4

```
     +------------------------+
IN1 =| [Socket] | IN2 =| [Socket] [Plug] |= OUT IN3 =| [Socket] | IN4 =| [Socket] | +——————————+
```



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_ADD_4` is a generic arithmetic block for the 4diac IDE, used to add four analog or numeric values. The unique feature of this block is the use of unidirectional adapters of type `AI` (Analog Input/Arithmetic Input) instead of traditional discrete data and event inputs. This enables a clean, modular, and clear structuring of signal flows in IEC 61499 applications.

Interface Structure

The block does not have any traditional event- or data-based inputs and outputs at the top level. All communication is handled via adapters.

Event Inputs

No direct event inputs are available. Event control is encapsulated within the adapters.

Event Outputs

No direct event outputs are available. Event control is encapsulated within the adapters.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AI`): First addend for the addition operation.
- **IN2** (Type: `adapter::types::unidirectional::AI`): Second addend for the addition operation.
- **IN3** (Type: `adapter::types::unidirectional::AI`): Third addend for the addition operation.
- **IN4** (Type: `adapter::types::unidirectional::AI`): Fourth addend for the addition operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AI`): Output adapter that provides the result of the addition (`IN1 + IN2 + IN3 + IN4`).

Functionality

As soon as a new event is signaled at one of the input adapters (`IN1` to `IN4`) indicating that a value has changed, the function block performs the following internal addition:

$$\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4} \tag{2.20}$$

The result is immediately passed to the output adapter `OUT`, and the corresponding update event of the output adapter is triggered. Since this is a generic function block (`GEN_AI_ADD`), the actual data type (e.g., `REAL`, `INT`, `LREAL`) depends on the specific implementation and instantiation of the adapter type used.

Technical Features

- **Generic Design:** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AI_ADD'`. This makes it highly flexible and allows it to be applied to different numeric data types, provided the adapters used support this.
- **Generic Design:** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AI_ADD'`. * **Unidirectional Adapters:** Using `unidirectional::AI` adapters drastically reduces wiring complexity in the function block diagram, as data and trigger events are bundled in a single connection.

State Overview

The function block behaves purely reactively (event-driven):

1. **Wait State (Idle):** The function block waits for an update event at one of the sockets (`IN1` to `IN4`).
2. **Calculation:** Upon receiving an event, the current values of all four inputs are read and added together.
3. **Output:** The calculated value is written to the plug `OUT`, triggering the corresponding output event. The function block immediately returns to the wait state.

Application Scenarios

- **Measurement Aggregation:** Summing of currents, powers, or flow rates from four different sources or sensors.
- **Average Preparation:** Combining four analog measurement values for subsequent division (e.g., for moving average calculation or redundancy checks).
- **Setpoint Offset Calculation:** Adding base setpoints, correction values, and user offsets in process engineering plants.

Comparison with Similar Function Blocks

Feature / Function Block	`AI_ADD_4`	Standard `ADD` (IEC 61131-3)
Interface Type	Adapter-based (`AI`)	Standard Data and Event Pins
Number of Inputs	Fixed (4)	Variable / Fixed (usually 2)
Wiring Effort	Very low (only 5 connections)	High (separate event and data lines)
Typing	Generic via adapter	Explicit via FB typing (e.g., `ADD_REAL`)

Change Detection

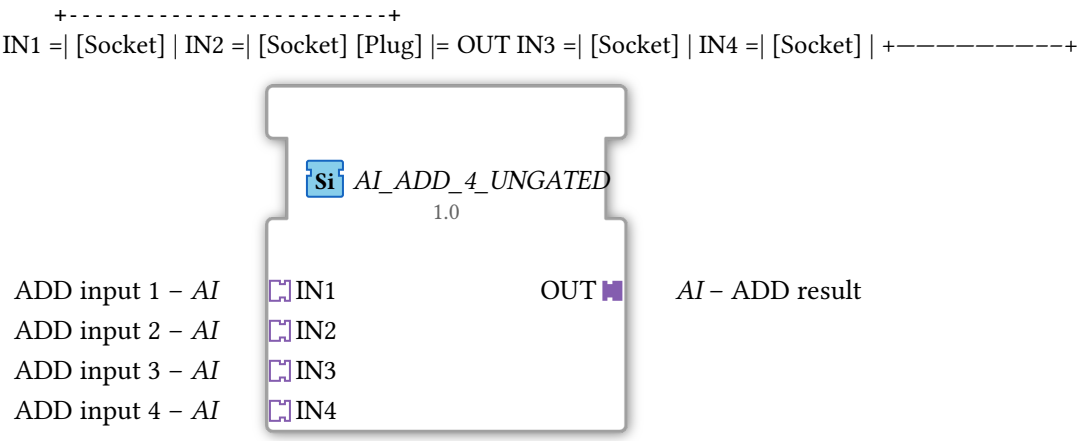
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AI_ADD_4` is a highly efficient and modern function block that fully utilizes the advantages of adapter-based coupling in IEC 61499. It is ideally suited for complex control architectures where clarity in the application diagram and a clear signal structure are paramount.

AI_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_ADD_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_ADD_4_UNGATED` is a generic arithmetic block for the 4diac IDE, used to add four analog or numeric values. The unique feature of this block is the use of unidirectional adapters of type `AI` (Analog Input/Arithmetic Input) instead of traditional discrete data and event inputs. This enables a clean, modular, and clear structuring of signal flows in IEC 61499 applications.

Interface Structure

The block does not have any traditional event- or data-based inputs and outputs at the top level. All communication is handled via adapters.

Event Inputs

No direct event inputs are available. Event control is encapsulated within the adapters.

Event Outputs

No direct event outputs are available. Event control is encapsulated within the adapters.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AI`): First addend for the addition operation.
- **IN2** (Type: `adapter::types::unidirectional::AI`): Second addend for the addition operation.
- **IN3** (Type: `adapter::types::unidirectional::AI`): Third addend for the addition operation.
- **IN4** (Type: `adapter::types::unidirectional::AI`): Fourth addend for the addition operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AI`): Output adapter that provides the result of the addition (`IN1 + IN2 + IN3 + IN4`).

Functionality

As soon as a new event is signaled at one of the input adapters (`IN1` to `IN4`) indicating that a value has changed, the function block performs the following internal addition:

$$\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4} \tag{2.21}$$

The result is immediately passed to the output adapter `OUT`, and the corresponding update event of the output adapter is triggered. Since this is a generic function block (`GEN_AI_ADD`), the actual data type (e.g., `REAL`, `INT`, `LREAL`) depends on the specific implementation and instantiation of the adapter type used.

Technical Features

- **Generic Design:** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AI_ADD'`. This makes it highly flexible and allows it to be applied to different numeric data types, provided the adapters used support this.
- **Generic Design:** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AI_ADD'`. * **Unidirectional Adapters:** Using `unidirectional::AI` adapters drastically reduces wiring complexity in the function block diagram, as data and trigger events are bundled in a single connection.

State Overview

The function block behaves purely reactively (event-driven):

1. **Wait State (Idle):** The function block waits for an update event at one of the sockets (`IN1` to `IN4`).
2. **Calculation:** Upon receiving an event, the current values of all four inputs are read and added together.
3. **Output:** The calculated value is written to the plug `OUT`, triggering the corresponding output event. The function block immediately returns to the wait state.

Application Scenarios

- **Measurement Aggregation:** Summing of currents, powers, or flow rates from four different sources or sensors.
- **Average Preparation:** Combining four analog measurement values for subsequent division (e.g., for moving average calculation or redundancy checks).
- **Setpoint Offset Calculation:** Adding base setpoints, correction values, and user offsets in process engineering plants.

Comparison with Similar Function Blocks

Feature / Function Block	`AI_ADD_4_UNGATED`	Standard `ADD` (IEC 61131-3)
Interface Type	Adapter-based (`AI`)	Standard Data and Event Pins
Number of Inputs	Fixed (4)	Variable / Fixed (usually 2)
Wiring Effort	Very low (only 5 connections)	High (separate event and data lines)
Typing	Generic via adapter	Explicit via FB typing (e.g., `ADD_REAL`)

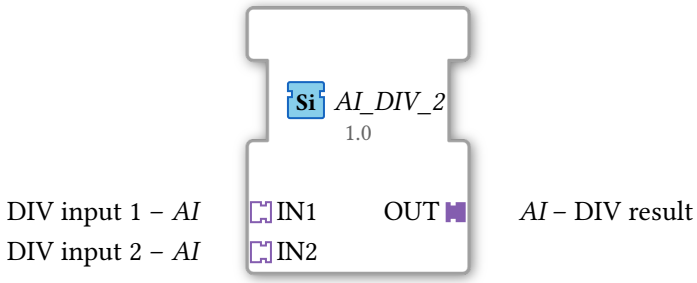
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AI_ADD_4_UNGATED` is a highly efficient and modern function block that fully utilizes the advantages of adapter-based coupling in IEC 61499. It is ideally suited for complex control architectures where clarity in the application diagram and a clear signal structure are paramount.

AI_DIV_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_DIV_2` is a generic arithmetic block for the 4diac platform. It is used to perform mathematical division (DIV) of two analog values. Unlike classic arithmetic blocks, this block uses standardized, unidirectional adapters of type `AI` (Analog Input) for its inputs and outputs. This enables structured, modular, and clear signal wiring within IEC 61499 applications.

Interface Structure

Event Inputs

The block does not have direct event inputs. Synchronization and event control are handled implicitly via the connected adapters.

Event Outputs

This function block has no direct event outputs.

Data Inputs

There are no direct data inputs. Data transmission is handled entirely via the adapter interfaces.

Data Outputs

There are no direct data outputs.

Adapter

Name	Type	Interface Type	Description
IN1	`adapter::types::unidirectional::AI`	Socket	First Input Value (Dividend)
IN2	`adapter::types::unidirectional::AI`	Socket	Second Input Value (Divisor)
OUT	`adapter::types::unidirectional::AI`	Plug	Result of Division (Quotient)

Functionality

The function block calculates the quotient of the values of the two input adapters:

$$OUT = \frac{IN1}{IN2} \tag{2.22}$$

As soon as the values at sockets `IN1` or `IN2` change and a corresponding update event is received via the adapter, the division is performed. The result and the associated update event are immediately forwarded to the output adapter `OUT` (plug).

Due to its generic type (`GEN_AI_DIV`), the function block is not limited to a specific data type (such as `REAL` or `INT`). The specific data type is determined flexibly during instantiation in the 4diac IDE.

Technical Features

- **Generic Typing:** The function block can be used polymorphically via the attribute `GenericClassName` with the value `GEN_AI_DIV` and can work with various compatible analog data types.
- **Adapter Encapsulation:** By using unidirectional adapters (`unidirectional::AI`), data and event lines are bundled into a single connection. This reduces the risk of wiring errors and improves clarity in the function block diagram (FBD).
- **Division by Zero:** Since this is a mathematical division, the system or upstream logic must ensure that the value at `IN2` is not zero to avoid calculation errors or division-by-zero exceptions at runtime.

State Overview

The function block `AI_DIV_2` is a stateless combination block. It does not have an internal Execution Control Chart (ECC). Every event at the input adapters directly triggers the calculation and update of the output adapter.

Application Scenarios

- **Ratio Calculations:** Determining ratios in process engineering plants (e.g., air-fuel ratio or mixing ratios of liquids).
- **Scaling and Normalization:** Dividing an analog raw value (e.g., sensor reading) by a scaling factor to convert it into physical units.
- **Measurement Averaging:** Used in computing networks where sum values need to be divided by a fixed or variable number.

Comparison with Similar Components

- **Standard Arithmetic Component (DIV):** A classic `DIV` component of IEC 61131-3 / IEC 61499 operates with direct PIN inputs (e.g., `IN1`, `IN2` as `REAL`) and separate event pins (`REQ`, `CNF`). `AI_DIV_2`, on the other hand, bundles these signals into adapters, enabling cleaner interface separation in complex signal paths.
- **AI_SUB_2 / AI_ADD_2:** These modules share the same adapter philosophy but perform subtractions and additions, respectively, instead of division.

Change Detection

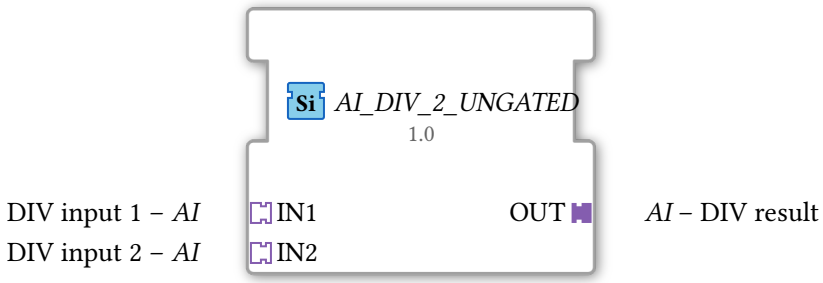
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AI_DIV_2` is a highly efficient and modern auxiliary module for analog signal processing in 4diac. Through the consistent use of adapters, it integrates seamlessly into service-oriented control architectures and minimizes the implementation and testing effort for basic mathematical operations.

AI_DIV_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_DIV_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_DIV_2_UNGATED` is a generic arithmetic block for the 4diac platform. It is used to perform mathematical division (DIV) of two analog values. Unlike classic arithmetic blocks, this block uses standardized, unidirectional adapters of type `AI` (Analog Input) for its inputs and outputs. This enables structured, modular, and clear signal wiring within IEC 61499 applications.

Interface Structure

Event Inputs

The block does not have direct event inputs. Synchronization and event control are handled implicitly via the connected adapters.

Event Outputs

This function block has no direct event outputs.

Data Inputs

There are no direct data inputs. Data transmission is handled entirely via the adapter interfaces.

Data Outputs

There are no direct data outputs.

Adapter

Name	Type	Interface Type	Description
IN1	`adapter::types::unidirectional::AI`	Socket	First Input Value (Dividend)
IN2	`adapter::types::unidirectional::AI`	Socket	Second Input Value (Divisor)
OUT	`adapter::types::unidirectional::AI`	Plug	Result of Division (Quotient)

Functionality

The function block calculates the quotient of the values of the two input adapters:

$$OUT = \frac{IN1}{IN2} \tag{2.23}$$

As soon as the values at sockets `IN1` or `IN2` change and a corresponding update event is received via the adapter, the division is performed. The result and the associated update event are immediately forwarded to the output adapter `OUT` (plug).

Due to its generic type (`GEN_AI_DIV`), the function block is not limited to a specific data type (such as `REAL` or `INT`). The specific data type is determined flexibly during instantiation in the 4diac IDE.

Technical Features

- **Generic Typing:** The function block can be used polymorphically via the attribute `GenericClassName` with the value `GEN_AI_DIV` and can work with various compatible analog data types.

- **Adapter Encapsulation:** By using unidirectional adapters (`unidirectional::AI`), data and event lines are bundled into a single connection. This reduces the risk of wiring errors and improves clarity in the function block diagram (FBD).
- **Division by Zero:** Since this is a mathematical division, the system or upstream logic must ensure that the value at `IN2` is not zero to avoid calculation errors or division-by-zero exceptions at runtime.

State Overview

The function block `AI_DIV_2_UNGATED` is a stateless combination block. It does not have an internal Execution Control Chart (ECC). Every event at the input adapters directly triggers the calculation and update of the output adapter.

Application Scenarios

- **Ratio Calculations:** Determining ratios in process engineering plants (e.g., air-fuel ratio or mixing ratios of liquids).
- **Scaling and Normalization:** Dividing an analog raw value (e.g., sensor reading) by a scaling factor to convert it into physical units.
- **Measurement Averaging:** Used in computing networks where sum values need to be divided by a fixed or variable number.

Comparison with Similar Components

- **Standard Arithmetic Component (DIV):** A classic `DIV` component of IEC 61131-3 / IEC 61499 operates with direct PIN inputs (e.g., `IN1`, `IN2` as `REAL`) and separate event pins (`REQ`, `CNF`). `AI_DIV_2_UNGATED`, on the other hand, bundles these signals into adapters, enabling cleaner interface separation in complex signal paths.
- **AI_SUB_2 / AI_ADD_2:** These modules share the same adapter philosophy but perform subtractions and additions, respectively, instead of division.

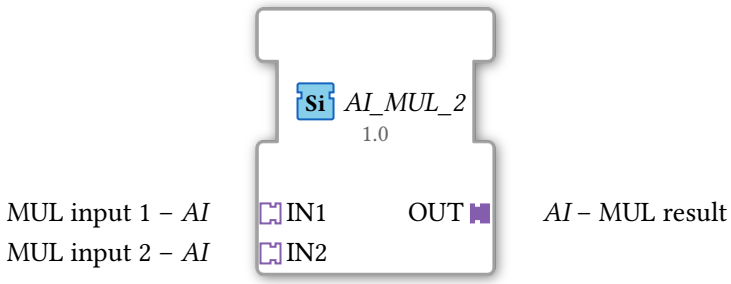
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AI_DIV_2_UNGATED` is a highly efficient and modern auxiliary module for analog signal processing in 4diac. Through the consistent use of adapters, it integrates seamlessly into service-oriented control architectures and minimizes the implementation and testing effort for basic mathematical operations.

AI_MUL_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_MUL_2` is a generic arithmetic function block for the 4diac IDE, compliant with the IEC 61131-3 standard. Its primary function is the multiplication of two input values provided via unidirectional adapters. The result of this multiplication is also output via a unidirectional adapter.

By using adapters instead of traditional discrete inputs and outputs, the complexity of the wiring in the control design is significantly reduced, as related data and events are bundled in a single connection.

The interface of this function block is entirely adapter-based. There are no direct, traditional event or data inputs and outputs on the block's interface.

Interface Structure

Event Inputs

No direct event inputs available (control is handled via the adapters).

Event Outputs

No direct event outputs available (control is handled via the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Name	Type	Direction (Mode)	Description
IN1	`adapter::types::unidirectional::AI`	Socket (Input)	First input value (multiplicand) for the arithmetic operation.
IN2	`adapter::types::unidirectional::AI`	Socket (Input)	Second input value (multiplier) for the arithmetic operation.
OUT	`adapter::types::unidirectional::AI`	Plug (Output)	Output for the calculated product of the two input values.

Functionality

The function block performs mathematical multiplication:

$$OUT = IN1 \times IN2 \tag{2.24}$$

Since the block is defined as a generic function block (`GEN_AI_MUL`), it can process different numeric data types (e.g., `REAL`, `LREAL`, `INT`, etc.) depending on the instantiation and typing of the adapters used.

As soon as an update event is received via the input adapters (`IN1` and/or `IN2`), the function block performs the multiplication internally and signals the update of the result via the output adapter `OUT`.

Technical Features

- **Generic Implementation:** The function block uses the class `GEN_AI_MUL`. This allows for high flexibility, as the specific data type is only determined when used in the system.
- **Unidirectional Adapters:** The interfaces use the type `adapter::types::unidirectional::AI`. This means that the information flow is strictly unidirectional, which increases system stability and performance.
- **Encapsulation:** The absence of individual signal pins keeps the application diagram clear and uncluttered, even with many mathematical operations.

State Overview

The module essentially behaves in a stateless manner (exhibiting analog characteristics):

Application Scenarios

- **Initialization / Idle State:** The module waits for incoming values via the adapters `IN1` and `IN2`.
- **Calculation:** Upon arrival of a new value or trigger signal at the sockets, the product is recalculated.
- **Output:** The result is immediately passed to the plug `OUT`, triggering a corresponding output event in the adapter.
- **Scaling of Sensor Values:** Multiplication of a raw analog value (e.g., from a 4–20 mA current input) by a scaling factor to convert it into a physical quantity.

- **Scaling of Sensor Values:** * **Calculation of physical quantities:** Calculation of power ($P = U \times I$) from measured voltage and current, provided these are supplied via appropriate adapter structures.
- **Amplifiers in control loops:** Use as a proportional gain factor (P-element) in software-based control.

Comparison with Similar Function Blocks

Compared to the standard IEC 61131-3 component `MUL`, the `AI_MUL_2` eliminates the need for manual wiring of trigger events (as with `REQ` and `CNF`) and individual data pins. While a classic `MUL` block requires separate lines for data and events for each connection, the `AI_MUL_2` bundles these using the `AI` adapters. This is particularly suitable for advanced, object-oriented, or modularized software architectures in 4diac.

Change Detection

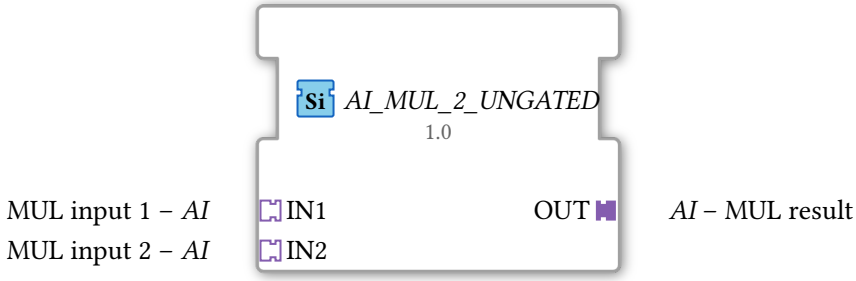
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AI_MUL_2` is a specialized, yet flexible, block for multiplying two values via adapter connections. It is ideally suited for clean, well-organized control architectures where analog signals need to be transmitted and processed in a standardized way using unidirectional adapters.

AI_MUL_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_MUL_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_MUL_2_UNGATED` is a generic arithmetic function block for the 4diac IDE, compliant with the IEC 61131-3 standard. Its primary function is the multiplication of two input values provided via unidirectional adapters. The result of this multiplication is also output via a unidirectional adapter.

By using adapters instead of traditional discrete inputs and outputs, the complexity of the wiring in the control design is significantly reduced, as related data and events are bundled in a single connection.

The interface of this function block is entirely adapter-based. There are no direct, traditional event or data inputs and outputs on the block's interface.

Interface Structure

Event Inputs

No direct event inputs available (control is handled via the adapters).

Event Outputs

No direct event outputs available (control is handled via the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Name	Type	Direction (Mode)	Description
IN1	`adapter::types::unidirectional::AI`	Socket (Input)	First input value (multiplicand) for the arithmetic operation.
IN2	`adapter::types::unidirectional::AI`	Socket (Input)	Second input value (multiplier) for the arithmetic operation.
OUT	`adapter::types::unidirectional::AI`	Plug (Output)	Output for the calculated product of the two input values.

Functionality

The function block performs mathematical multiplication:

$$OUT = IN1 \times IN2 \tag{2.25}$$

Since the block is defined as a generic function block (`GEN_AI_MUL`), it can process different numeric data types (e.g., `REAL`, `LREAL`, `INT`, etc.) depending on the instantiation and typing of the adapters used.

As soon as an update event is received via the input adapters (`IN1` and/or `IN2`), the function block performs the multiplication internally and signals the update of the result via the output adapter `OUT`.

Technical Features

- **Generic Implementation:** The function block uses the class `GEN_AI_MUL`. This allows for high flexibility, as the specific data type is only determined when used in the system.
- **Unidirectional Adapters:** The interfaces use the type `adapter::types::unidirectional::AI`. This means that the information flow is strictly unidirectional, which increases system stability and performance.
- **Encapsulation:** The absence of individual signal pins keeps the application diagram clear and uncluttered, even with many mathematical operations.

State Overview

The module essentially behaves in a stateless manner (exhibiting analog characteristics):

Application Scenarios

- **Initialization / Idle State:** The module waits for incoming values via the adapters `IN1` and `IN2`.
- **Calculation:** Upon arrival of a new value or trigger signal at the sockets, the product is recalculated.
- **Output:** The result is immediately passed to the plug `OUT`, triggering a corresponding output event in the adapter.
- **Scaling of Sensor Values:** Multiplication of a raw analog value (e.g., from a 4–20 mA current input) by a scaling factor to convert it into a physical quantity.
- **Scaling of Sensor Values:** * **Calculation of physical quantities:** Calculation of power ($P = U \times I$) from measured voltage and current, provided these are supplied via appropriate adapter structures.
- **Amplifiers in control loops:** Use as a proportional gain factor (P-element) in software-based control.

Comparison with Similar Function Blocks

Compared to the standard IEC 61131-3 component `MUL`, the `AI_MUL_2_UNGATED` eliminates the need for manual wiring of trigger events (as with `REQ` and `CNF`) and individual data pins. While a classic `MUL` block requires separate lines for data and events for each connection, the `AI_MUL_2_UNGATED` bundles these using the `AI` adapters. This is particularly suitable for advanced, object-oriented, or modularized software architectures in 4diac.

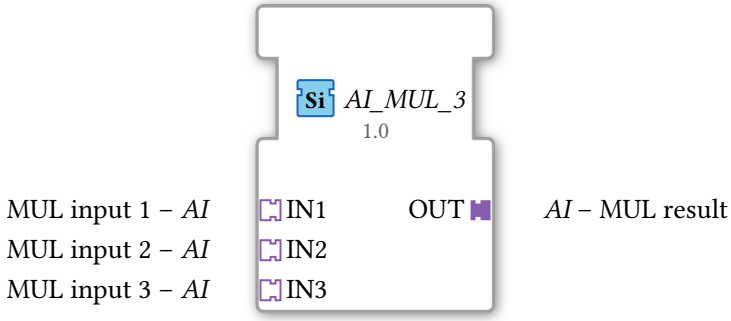
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AI_MUL_2_UNGATED` is a specialized, yet flexible, block for multiplying two values via adapter connections. It is ideally suited for clean, well-organized control architectures where analog signals need to be transmitted and processed in a standardized way using unidirectional adapters.

AI_MUL_3



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_MUL_3` is a generic arithmetic function block designed for multiplying three input values. It conforms to the classification for standard arithmetic functions according to IEC 61131-3 and utilizes the 4diac adapter concept. Encapsulating the signals in adapters significantly reduces the visual complexity of the function block diagram.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Event control is handled internally via the adapters.

Event Outputs

This function block does not have direct event outputs. Event control is handled internally via the adapters.

Data Inputs

This function block has no direct data inputs. Data is transferred via the input adapters.

Data Outputs

This function block has no direct data outputs. Data is output via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AI`): The first multiplicand (input value 1).
- **IN2** (Type: `adapter::types::unidirectional::AI`): The second multiplicand (input value 2).
- **IN3** (Type: `adapter::types::unidirectional::AI`): The third multiplicand (input value 3).

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AI`): The result of the multiplication.

Functionality

The function block performs a continuous or event-driven arithmetic multiplication of the three values connected via the sockets.

The mathematical formula is:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.26}$$

As soon as the values at the input adapters change or a corresponding update event is triggered via the adapters, the function block calculates the product and makes it available at the output adapter `OUT`.

Technical Features

- **Generic Nature:** The function block is declared as a generic type (`GenericClassName = 'GEN_AI_MUL'`). This means it can flexibly work with various numeric data types (such as REAL, LREAL, INT), provided these are supported by the adapter type `AI`.
- **Adapter Coupling:** By using unidirectional adapters (`unidirectional::AI`), data and its validation events are combined. This drastically simplifies the wiring in the 4diac IDE, as separate event and data lines are no longer required.

–

State Overview

The function block does not have a complex internal state diagram (stateless). It reacts purely to changes in values or trigger events at inputs `IN1`, `IN2`, and `IN3` and immediately forwards the result to `OUT`.

Application Scenarios

- **Measurement Scaling:** Multiplication of an analog raw value (`IN1`) by a calibration factor (`IN2`) and a further correction factor (`IN3`).
- **Physical Calculations:** Calculation of volumes (V = l × b × h) or power, where three factors must be multiplied directly together.
- – **Cascaded Amplifications:** Signal processing chains in which a signal passes through two amplification stages sequentially.

Comparison with Similar Components

- **Standard MUL Component (IEC 61131-3):** Classic multiplication components have direct data pins (e.g., `ANY_NUM`) and require explicit event connections (`REQ` / `CNF`). `AI_MUL_3` bundles this logic into adapters.
- **AI_MUL_2 (Dual Multiplier):** While multiplying three values with a standard dual multiplier requires two components cascaded, `AI_MUL_3` accomplishes this in a single step, saving resources and space in the control diagram.

Change Detection

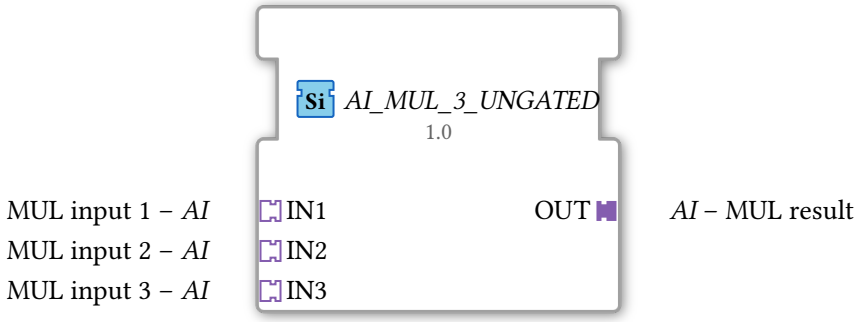
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AI_MUL_3` function block offers an efficient, clean, and high-performance way to implement triple multiplications within a 4diac application. Through the consistent use of the adapter concept, it significantly contributes to the clarity and maintainability of control software.

AI_MUL_3_UNGATED

i UNGATED variant: This block is the ungated version of `AI_MUL_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_MUL_3_UNGATED` is a generic arithmetic function block designed for multiplying three input values. It conforms to the classification for standard arithmetic functions according to IEC 61131-3 and utilizes the 4diac adapter concept. Encapsulating the signals in adapters significantly reduces the visual complexity of the function block diagram.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Event control is handled internally via the adapters.

Event Outputs

This function block does not have direct event outputs. Event control is handled internally via the adapters.

Data Inputs

This function block has no direct data inputs. Data is transferred via the input adapters.

Data Outputs

This function block has no direct data outputs. Data is output via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AI`): The first multiplicand (input value 1).
- **IN2** (Type: `adapter::types::unidirectional::AI`): The second multiplicand (input value 2).
- **IN3** (Type: `adapter::types::unidirectional::AI`): The third multiplicand (input value 3).

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AI`): The result of the multiplication.

Functionality

The function block performs a continuous or event-driven arithmetic multiplication of the three values connected via the sockets.

The mathematical formula is:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \tag{2.27}$$

As soon as the values at the input adapters change or a corresponding update event is triggered via the adapters, the function block calculates the product and makes it available at the output adapter `OUT`.

Technical Features

- **Generic Nature:** The function block is declared as a generic type (`GenericClassName = 'GEN_AI_MUL'`). This means it can flexibly work with various numeric data types (such as REAL, LREAL, INT), provided these are supported by the adapter type `AI`.

- **Adapter Coupling:** By using unidirectional adapters (`unidirectional::AI`), data and its validation events are combined. This drastically simplifies the wiring in the 4diac IDE, as separate event and data lines are no longer required.

–

State Overview

The function block does not have a complex internal state diagram (stateless). It reacts purely to changes in values or trigger events at inputs `IN1`, `IN2`, and `IN3` and immediately forwards the result to `OUT`.

Application Scenarios

- **Measurement Scaling:** Multiplication of an analog raw value (`IN1`) by a calibration factor (`IN2`) and a further correction factor (`IN3`).
- **Physical Calculations:** Calculation of volumes (V = l × b × h) or power, where three factors must be multiplied directly together.
- – **Cascaded Amplifications:** Signal processing chains in which a signal passes through two amplification stages sequentially.

Comparison with Similar Components

- **Standard MUL Component (IEC 61131-3):** Classic multiplication components have direct data pins (e.g., `ANY_NUM`) and require explicit event connections (`REQ` / `CNF`). `AI_MUL_3_UNGATED` bundles this logic into adapters.
- **AI_MUL_2 (Dual Multiplier):** While multiplying three values with a standard dual multiplier requires two components cascaded, `AI_MUL_3_UNGATED` accomplishes this in a single step, saving resources and space in the control diagram.

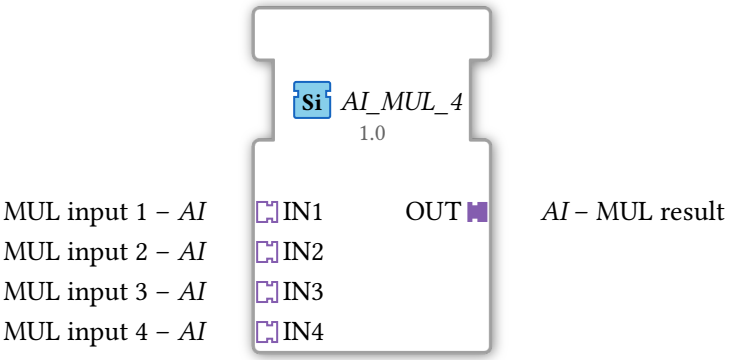
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AI_MUL_3_UNGATED` function block offers an efficient, clean, and high-performance way to implement triple multiplications within a 4diac application. Through the consistent use of the adapter concept, it significantly contributes to the clarity and maintainability of control software.

AI_MUL_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_MUL_4` is a generic arithmetic function block for 4diac-IDE, compliant with the IEC 61131-3 standard. Its primary function is to calculate the mathematical product of four analog input values. The block utilizes unidirectional adapters of type `AI` (Analog Input) to transmit both data and the

associated events in encapsulated form. This significantly reduces the wiring effort in the function block diagram and ensures a cleaner structure for the control logic.

Interface Structure

Event Inputs

The function block does not have direct, independent event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

This function block does not have direct, independent event outputs. Subsequent blocks are triggered via the output adapter.

Data Inputs

There are no direct data inputs. Values are provided via the adapter inputs.

Data Outputs

There are no direct data outputs. The calculated value is output via the adapter output.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AI`): First multiplicand.
- **IN2** (Type: `adapter::types::unidirectional::AI`): Second multiplicand.
- **IN3** (Type: `adapter::types::unidirectional::AI`): Third multiplicand.
- **IN4** (Type: `adapter::types::unidirectional::AI`): Fourth multiplicand.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AI`): Product of the multiplication of all four inputs ($OUT = IN1 \cdot IN2 \cdot IN3 \cdot IN4$).

Functionality

As soon as a new event is signaled at one of the input adapters (`IN1` to `IN4`) indicating that the data has been updated, the function block reads the analog values of the four adapters.

The calculation is performed according to the formula:

$$\text{Result} = \text{Value}\ (IN1) \cdot \text{Value}\ (IN2) \cdot \text{Value}\ (IN3) \cdot \text{Value}\ (IN4) \tag{2.28}$$

The result of this calculation is passed to the output adapter `OUT`, and the corresponding update event of the adapter is triggered to inform subsequent blocks in the control flow about the new value.

- **Generic Class:** The block is internally based on the generic class `GEN_AI_MUL`. This enables flexible processing of data streams independent of specific hardware implementations.
- **Unidirectional Adapters:** By using the type `adapter::types::unidirectional::AI`, data and trigger signals flow in only one direction (from sender to receiver). This simplifies signal design and avoids feedback loops.
- **Compact Structure:** Processing four inputs in a single function block eliminates the need to cascade multiple multiplication blocks.

Technical Features

Since `AI_MUL_4` is a purely mathematical, data- and event-driven function block, it does not have an internal state machine (ECC). Its execution is purely stateless:

State Overview

1. **Wait:** The function block waits for an update event at one of the sockets (`IN1` to `IN4`).
2. **Calculate:** Upon triggering, the current values of all four inputs are multiplied.
3. **Output:** The result is applied to the plug `OUT`, and the output event is triggered. The module immediately returns to standby mode.

Application Scenarios

- **Scaling and Calibration:** Calculation of compound correction factors for analog sensor values (e.g., sensor value \cdot $ calibration factor \cdot $ temperature compensation \cdot $ unit conversion).
- **Physical Calculations:** Calculation of values that depend on multiple analog factors, such as volumetric flow rates or electrical power, taking into account different efficiencies.
- **Cascaded Gains:** Use in control loops where multiple gains must be applied sequentially to an analog signal.

Comparison with Similar Function Blocks

- **Standard MUL (IEC 61131-3):** Classic multiplication function blocks typically only work with elementary data types (e.g., `REAL`, `INT`) and require separate event lines (`REQ` / `CNF`). Furthermore, they often only support two inputs by default. `AI_MUL_4`, on the other hand, directly processes four inputs and uses adapters for encapsulation, resulting in a clearer network layout.
- **AI_MUL_2:** A similar adapter-based function block, but only for two inputs. `AI_MUL_4` saves additional wiring effort and function block instances for more complex calculations with up to four factors.

Change Detection

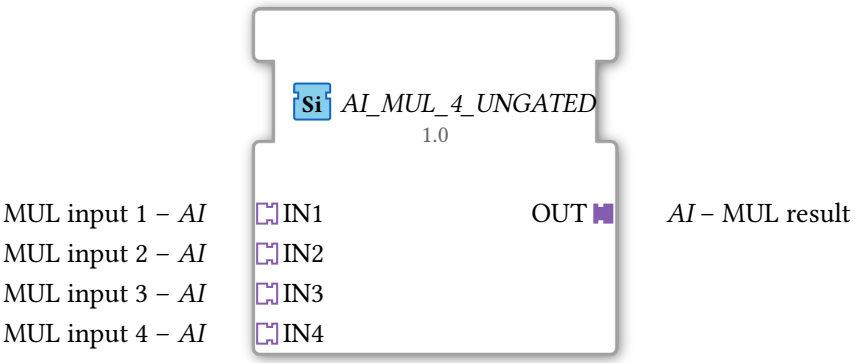
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AI_MUL_4` function block offers an efficient and clean way to implement complex multiplication tasks with up to four analog signals in 4diac-IDE. The consistent use of unidirectional adapters minimizes cabling effort and significantly improves the readability of the application design.

AI_MUL_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_MUL_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_MUL_4_UNGATED` is a generic arithmetic function block for 4diac-IDE, compliant with the IEC 61131-3 standard. Its primary function is to calculate the mathematical product of four analog input values. The block utilizes unidirectional adapters of type `AI` (Analog Input) to transmit both data and the associated events in encapsulated form. This significantly reduces the wiring effort in the function block diagram and ensures a cleaner structure for the control logic.

Interface Structure

Event Inputs

The function block does not have direct, independent event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

This function block does not have direct, independent event outputs. Subsequent blocks are triggered via the output adapter.

Data Inputs

There are no direct data inputs. Values are provided via the adapter inputs.

Data Outputs

There are no direct data outputs. The calculated value is output via the adapter output.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AI`): First multiplicand.
- **IN2** (Type: `adapter::types::unidirectional::AI`): Second multiplicand.
- **IN3** (Type: `adapter::types::unidirectional::AI`): Third multiplicand.
- **IN4** (Type: `adapter::types::unidirectional::AI`): Fourth multiplicand.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AI`): Product of the multiplication of all four inputs ($OUT = IN1 \cdot IN2 \cdot IN3 \cdot IN4$).

Functionality

As soon as a new event is signaled at one of the input adapters (`IN1` to `IN4`) indicating that the data has been updated, the function block reads the analog values of the four adapters.

The calculation is performed according to the formula:

$$\text{Result} = \text{Value}\ (IN1) \cdot \text{Value}\ (IN2) \cdot \text{Value}\ (IN3) \cdot \text{Value}\ (IN4) \tag{2.29}$$

The result of this calculation is passed to the output adapter `OUT`, and the corresponding update event of the adapter is triggered to inform subsequent blocks in the control flow about the new value.

- **Generic Class:** The block is internally based on the generic class `GEN_AI_MUL`. This enables flexible processing of data streams independent of specific hardware implementations.
- **Unidirectional Adapters:** By using the type `adapter::types::unidirectional::AI`, data and trigger signals flow in only one direction (from sender to receiver). This simplifies signal design and avoids feedback loops.
- **Compact Structure:** Processing four inputs in a single function block eliminates the need to cascade multiple multiplication blocks.

Technical Features

Since `AI_MUL_4_UNGATED` is a purely mathematical, data- and event-driven function block, it does not have an internal state machine (ECC). Its execution is purely stateless:

State Overview

1. **Wait:** The function block waits for an update event at one of the sockets (`IN1` to `IN4`).
2. **Calculate:** Upon triggering, the current values of all four inputs are multiplied.
3. **Output:** The result is applied to the plug `OUT`, and the output event is triggered. The module immediately returns to standby mode.

Application Scenarios

- **Scaling and Calibration:** Calculation of compound correction factors for analog sensor values (e.g., sensor value \cdot $ calibration factor \cdot $ temperature compensation \cdot $ unit conversion).

- **Physical Calculations:** Calculation of values that depend on multiple analog factors, such as volumetric flow rates or electrical power, taking into account different efficiencies.
- **Cascaded Gains:** Use in control loops where multiple gains must be applied sequentially to an analog signal.

Comparison with Similar Function Blocks

- **Standard MUL (IEC 61131-3):** Classic multiplication function blocks typically only work with elementary data types (e.g., `REAL`, `INT`) and require separate event lines (`REQ` / `CNF`). Furthermore, they often only support two inputs by default. `AI_MUL_4_UNGATED`, on the other hand, directly processes four inputs and uses adapters for encapsulation, resulting in a clearer network layout.
- **AI_MUL_2:** A similar adapter-based function block, but only for two inputs. `AI_MUL_4_UNGATED` saves additional wiring effort and function block instances for more complex calculations with up to four factors.

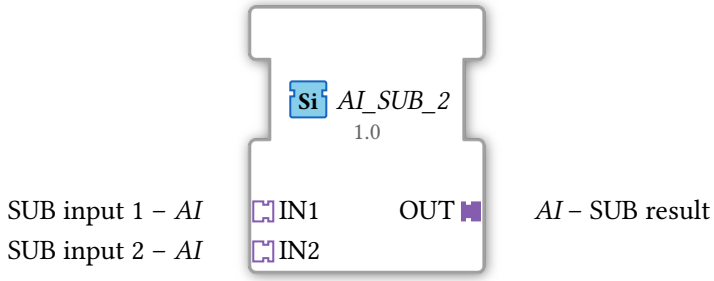
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AI_MUL_4_UNGATED` function block offers an efficient and clean way to implement complex multiplication tasks with up to four analog signals in 4diac-IDE. The consistent use of unidirectional adapters minimizes cabling effort and significantly improves the readability of the application design.

AI_SUB_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_SUB_2` is a generic arithmetic function block for performing subtraction. It is specifically designed for use with analog interface adapters. The block calculates the difference between two analog input values and outputs the result to an analog output adapter. Using adapters keeps the interface design in the 4diac-ide clear and concise.

Interface Structure

Since this function block is based entirely on an adapter architecture, it does not have any direct, traditional event or data channels on the block interface. All communication is handled via the adapters.

Event Inputs

No direct event inputs available (event control is encapsulated in the adapters).

Event Outputs

No direct event outputs available (event control is encapsulated in the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AI`): The first analog input value (minuend) for the subtraction operation.
- **IN2** (Type: `adapter::types::unidirectional::AI`): The second analog input value (subtrahend), which is subtracted from `IN1`.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AI`): The result of the subtraction (difference: `IN1 - IN2`).

Functionality

This function block performs basic arithmetic subtraction:

$$OUT = IN1 - IN2 \tag{2.30}$$

As soon as the input values at sockets `IN1` or `IN2` change, or a corresponding update event is received via the adapters, the function block calculates the difference internally. The result and the associated update event are immediately forwarded to output plug `OUT`.

Technical Features

- **Generic Block Type (`GEN_AI_SUB`)**: The block is internally defined as a generic type. This allows for flexible adaptation to the analog data types used by the adapters in the system.
- **Adapter Encapsulation**: The use of unidirectional analog adapters (`unidirectional::AI`) consolidates data and event lines. This minimizes the wiring effort in the FBD (Function Block Diagram) and significantly improves clarity.

State Overview

The block behaves like a purely functional block without internal memory (analogous to a classic combinational gate). It does not have a complex Execution Control Chart (ECC). Any activity on the input adapters directly triggers the calculation and updates the output adapter.

Application Scenarios

- **Differential Pressure and Differential Temperature Measurement**: Calculation of physical differential values in process engineering (e.g., determining the pressure drop across a filter or the temperature difference between the flow and return lines).
- **Control Deviation**: Calculation of the control deviation ($e = w - x$) in control loops where the setpoint and actual value are available as adapter structures.
- **Zero Point Correction / Offset Subtraction**: Subtraction of a calibration or offset value from a continuous measurement signal.

Comparison with Similar Components

Compared to standard subtraction components of IEC 61131-3 (such as a classic `SUB` component with discrete pins), the `AI_SUB_2` eliminates the need for manually wiring separate `REQ` and `CNF` event lines as well as individual analog data lines. All relevant information is cleanly transmitted in the `AI` adapters.

Change Detection

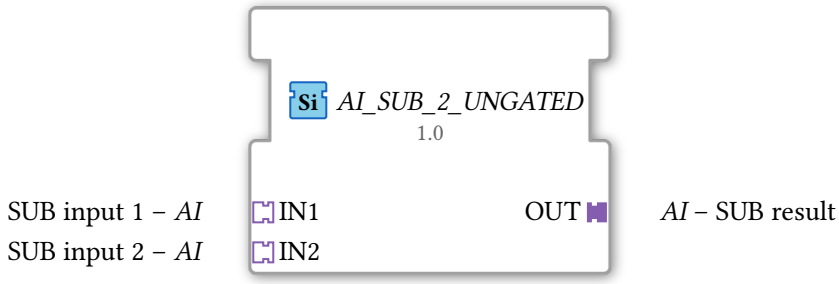
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AI_SUB_2` is a specialized and highly efficient component for modern, adapter-based programming in 4diac. It is ideally suited for clean software architectures in the field of analog signal preprocessing, where clarity and encapsulation are paramount.

AI_SUB_2_UNGATED

i UNGATED variant: This block is the ungated version of `AI_SUB_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AI_SUB_2_UNGATED` is a generic arithmetic function block for performing subtraction. It is specifically designed for use with analog interface adapters. The block calculates the difference between two analog input values and outputs the result to an analog output adapter. Using adapters keeps the interface design in the 4diac-ide clear and concise.

Interface Structure

Since this function block is based entirely on an adapter architecture, it does not have any direct, traditional event or data channels on the block interface. All communication is handled via the adapters.

Event Inputs

No direct event inputs available (event control is encapsulated in the adapters).

Event Outputs

No direct event outputs available (event control is encapsulated in the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AI`): The first analog input value (minuend) for the subtraction operation.
- **IN2** (Type: `adapter::types::unidirectional::AI`): The second analog input value (subtrahend), which is subtracted from `IN1`.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AI`): The result of the subtraction (difference: `IN1 - IN2`).

Functionality

This function block performs basic arithmetic subtraction:

$$OUT = IN1 - IN2 \tag{2.31}$$

As soon as the input values at sockets `IN1` or `IN2` change, or a corresponding update event is received via the adapters, the function block calculates the difference internally. The result and the associated update event are immediately forwarded to output plug `OUT`.

Technical Features

- **Generic Block Type (`GEN_AI_SUB`)**: The block is internally defined as a generic type. This allows for flexible adaptation to the analog data types used by the adapters in the system.
- **Adapter Encapsulation**: The use of unidirectional analog adapters (`unidirectional::AI`) consolidates data and event lines. This minimizes the wiring effort in the FBD (Function Block Diagram) and significantly improves clarity.

State Overview

The block behaves like a purely functional block without internal memory (analogous to a classic combinational gate). It does not have a complex Execution Control Chart (ECC). Any activity on the input adapters directly triggers the calculation and updates the output adapter.

Application Scenarios

- **Differential Pressure and Differential Temperature Measurement**: Calculation of physical differential values in process engineering (e.g., determining the pressure drop across a filter or the temperature difference between the flow and return lines).
- **Control Deviation**: Calculation of the control deviation ($e = w - x$) in control loops where the setpoint and actual value are available as adapter structures.
- **Zero Point Correction / Offset Subtraction**: Subtraction of a calibration or offset value from a continuous measurement signal.

Comparison with Similar Components

Compared to standard subtraction components of IEC 61131-3 (such as a classic `SUB` component with discrete pins), the `AI_SUB_2_UNGATED` eliminates the need for manually wiring separate `REQ` and `CNF` event lines as well as individual analog data lines. All relevant information is cleanly transmitted in the `AI` adapters.

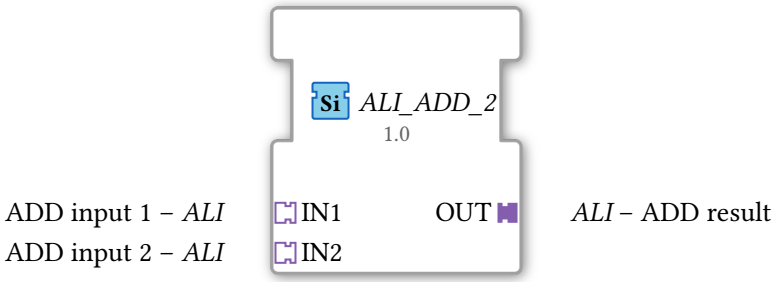
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AI_SUB_2_UNGATED` is a specialized and highly efficient component for modern, adapter-based programming in 4diac. It is ideally suited for clean software architectures in the field of analog signal preprocessing, where clarity and encapsulation are paramount.

ALI_ADD_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_ADD_2` is a generic arithmetic block for performing addition (ADD) based on the IEC 61131-3 standard. It is classified as a standard arithmetic function and serves to add two input values via an adapter interface and output the result.

By using adapters, this block enables highly modular and clear wiring within the 4diac IDE.

Interface Structure

Event Inputs
No direct event inputs are available. Control and synchronization are handled via the adapters.

Event Outputs
No direct event outputs are available. Control and synchronization are handled via the adapters.

Data Inputs
No direct data inputs available.

Data Outputs
No direct data outputs available.

Adapters
- **IN1 (Socket)**: Input adapter type `adapter::types::unidirectional::ALI`. Represents the first addend of the addition.
- **IN2 (Socket)**: Input adapter type `adapter::types::unidirectional::ALI`. Represents the second addend of the addition.
- **OUT (Plug)**: Output adapter type `adapter::types::unidirectional::ALI`. Provides the mathematical result of the addition (`IN1 + IN2`).

Functionality
The function block `ALI_ADD_2` performs a classic mathematical addition. As soon as the values at the input adapters `IN1` and/or `IN2` change, or a corresponding transfer event is triggered via the adapters, the block calculates the sum:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.32}$$

The result is immediately available via the output adapter `OUT`. Since the block is defined as a generic type (implemented via the generic class `GEN_ALI_ADD`), it is not tied to a specific data type but can adapt to various compatible numeric data types.

Technical Features
- **Generic Implementation**: The attribute `GenericClassName` with the value `'GEN_ALI_ADD'` allows the function block to be flexibly instantiated for different data types (e.g., INT, REAL, LREAL).
- **Adapter Coupling**: Unlike traditional function blocks that use discrete pins for data and events, this function block bundles the signals in unidirectional `ALI` adapters. This significantly reduces the number of wires in the function block diagram.

State Overview
The function block is stateless (combinatorial behavior). It does not have an internal state machine (ECC). Any change to the inputs directly updates the output.

Application Scenarios
- **Signal Processing**: Summing two sensor values or physical quantities that are already distributed within the application via an adapter system.
- **Setpoint Generation**: Adding a base setpoint and a correction offset.
- **Modular Plant Engineering**: Use in systems where interfaces are consistently mapped via adapters to maximize code reusability and readability.

Comparison with Similar Components
Classic `ADD` components according to IEC 61131-3 typically have direct data inputs (`IN1`, `IN2`...) and event pins (`REQ`, `CNF`). The `ALI_ADD_2` differs in that it encapsulates these channels in adapters (`ALI`). This results in better encapsulation and cleaner structuring in complex 4diac applications, but requires that the connected components also support the `ALI` adapter protocol.

Change Detection

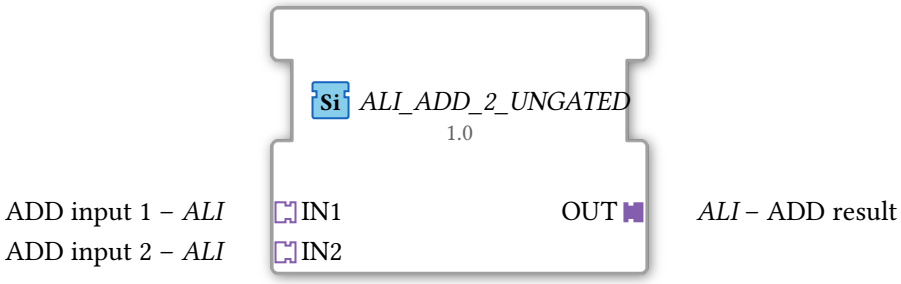
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALI_ADD_2` is a specialized, highly efficient addition component for adapter-based architectures in the 4diac IDE. It combines the standard mathematical functionality of addition with the structural advantages of modern, adapter-based communication paradigms.

ALI_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_ADD_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_ADD_2_UNGATED` is a generic arithmetic block for performing addition (ADD) based on the IEC 61131-3 standard. It is classified as a standard arithmetic function and serves to add two input values via an adapter interface and output the result.

By using adapters, this block enables highly modular and clear wiring within the 4diac IDE.

Interface Structure

Event Inputs

No direct event inputs are available. Control and synchronization are handled via the adapters.

Event Outputs

No direct event outputs are available. Control and synchronization are handled via the adapters.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

- **IN1 (Socket)**: Input adapter type `adapter::types::unidirectional::ALI`. Represents the first addend of the addition.
- **IN2 (Socket)**: Input adapter type `adapter::types::unidirectional::ALI`. Represents the second addend of the addition.
- **OUT (Plug)**: Output adapter type `adapter::types::unidirectional::ALI`. Provides the mathematical result of the addition (`IN1 + IN2`).

Functionality

The function block `ALI_ADD_2_UNGATED` performs a classic mathematical addition. As soon as the values at the input adapters `IN1` and/or `IN2` change, or a corresponding transfer event is triggered via the adapters, the block calculates the sum:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.33}$$

The result is immediately available via the output adapter `OUT`. Since the block is defined as a generic type (implemented via the generic class `GEN_ALI_ADD`), it is not tied to a specific data type but can adapt to various compatible numeric data types.

Technical Features

- **Generic Implementation**: The attribute `GenericClassName` with the value `'GEN_ALI_ADD'` allows the function block to be flexibly instantiated for different data types (e.g., INT, REAL, LREAL).
- **Adapter Coupling**: Unlike traditional function blocks that use discrete pins for data and events, this function block bundles the signals in unidirectional `ALI` adapters. This significantly reduces the number of wires in the function block diagram.

State Overview

The function block is stateless (combinatorial behavior). It does not have an internal state machine (ECC). Any change to the inputs directly updates the output.

Application Scenarios

- **Signal Processing**: Summing two sensor values or physical quantities that are already distributed within the application via an adapter system.
- **Setpoint Generation**: Adding a base setpoint and a correction offset.
- **Modular Plant Engineering**: Use in systems where interfaces are consistently mapped via adapters to maximize code reusability and readability.

Comparison with Similar Components

Classic `ADD` components according to IEC 61131-3 typically have direct data inputs (`IN1`, `IN2`...) and event pins (`REQ`, `CNF`). The `ALI_ADD_2_UNGATED` differs in that it encapsulates these channels in adapters (`ALI`). This results in better encapsulation and cleaner structuring in complex 4diac applications, but requires that the connected components also support the `ALI` adapter protocol.

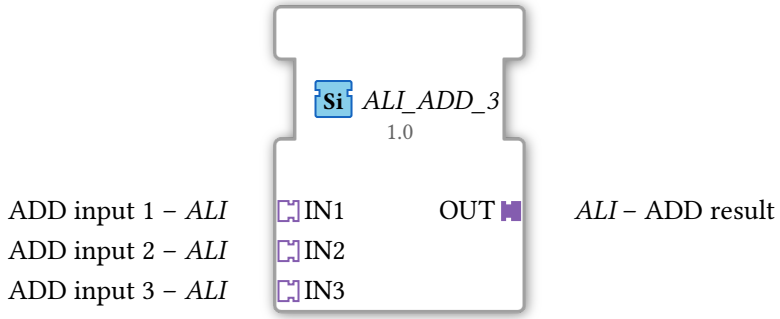
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALI_ADD_2_UNGATED` is a specialized, highly efficient addition component for adapter-based architectures in the 4diac IDE. It combines the standard mathematical functionality of addition with the structural advantages of modern, adapter-based communication paradigms.

ALI_ADD_3



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_ADD_3` is a generic arithmetic block for adding three values. It is based on the IEC 61131-3 standard for arithmetic functions. Instead of classic, separate data and event connections, this block uses unidirectional adapters of type `ALI` to enable structured and clear signal transmission in 4diac-ide.

Interface Structure

Event Inputs

No direct event inputs are available. Control and triggering are implicit via the connected adapters.

Event Outputs

No direct event outputs are available. Events are forwarded via the output adapter.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

The module communicates exclusively via adapter interfaces:

- **Sockets (Input Interfaces):**
- `IN1` (Type: `adapter::types::unidirectional::ALI`): Interface for the first summand (input value 1).
- `IN2` (Type: `adapter::types::unidirectional::ALI`): Interface for the second summand (input value 2).
- `IN3` (Type: `adapter::types::unidirectional::ALI`): Interface for the third addend (input value 3).
- **Plugs (Output Interfaces):**
- `OUT` (Type: `adapter::types::unidirectional::ALI`): Interface for outputting the calculated result.

Functionality

The function block `ALI_ADD_3` performs a standard arithmetic addition. As soon as new values are signaled at the input adapters (`IN1`, `IN2`, `IN3`), the function block calculates the sum using the following formula:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.34}$$

The result and the corresponding update event are then passed on to the subsequent function blocks via the output adapter `OUT`.

Technical Features

- **Generic Implementation:** The function block is declared as a generic type (`GEN_ALI_ADD`). This allows for flexible adaptation to various data types supported by the `ALI` adapter.
- **Use of Adapters:** By encapsulating data and events in unidirectional adapters (`ALI`), the number of necessary connection lines in the 4diac-ide application editor is drastically reduced, resulting in improved clarity.

State Overview

The function block operates purely stateless. There is no internal state machine (ECC). The calculation is purely data- or event-driven: An incoming event at one of the input adapters directly triggers the addition and updating of the output adapter.

Application Scenarios

- **Measurement Summing:** Summarizing three individual analog measurements (e.g., power consumption of three phases, flow rates from three pipes) into a total value.
- **Average Preparation:** Summing three values for subsequent division by 3 in a subsequent function block.
- **Structured Signal Processing:** Used in complex projects where analog signals need to be standardized and routed clearly via adapter channels.

Comparison with Similar Components

- **Standard ADD (IEC 61131-3):** A standard ADD component uses classic data and event connections. `ALI_ADD_3`, on the other hand, encapsulates these interfaces in adapters, reducing visual complexity in system design.
- **Cascaded Dual Adders:** To add three values with conventional dual adders, two components would have to be connected in series. `ALI_ADD_3` eliminates the need for one component as well as the intermediate instantiation and wiring.

Change Detection

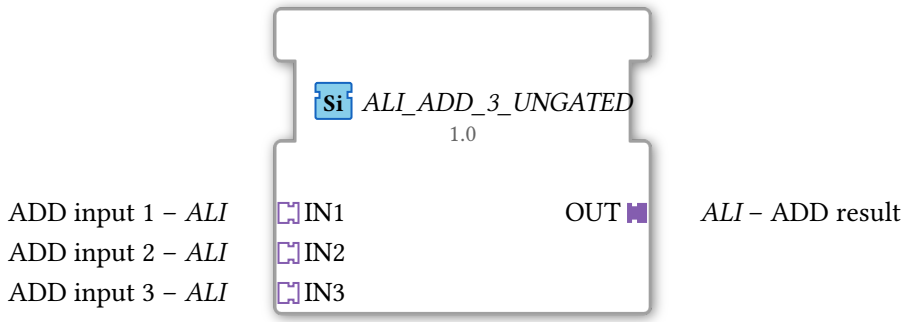
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

`ALI_ADD_3` is a compact and efficient auxiliary component for 4diac-ide. Through the consistent use of unidirectional adapters, he makes a significant contribution to the creation of clean, modularized and easily readable control code for processing analog signals.

ALI_ADD_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_ADD_3_UNGATED` is a generic arithmetic block for adding three values. It is based on the IEC 61131-3 standard for arithmetic functions. Instead of classic, separate data and event connections, this block uses unidirectional adapters of type `ALI` to enable structured and clear signal transmission in 4diac-ide.

Interface Structure

Event Inputs

No direct event inputs are available. Control and triggering are implicit via the connected adapters.

Event Outputs

No direct event outputs are available. Events are forwarded via the output adapter.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

The module communicates exclusively via adapter interfaces:

- **Sockets (Input Interfaces):**
- `IN1` (Type: `adapter::types::unidirectional::ALI`): Interface for the first summand (input value 1).
- `IN2` (Type: `adapter::types::unidirectional::ALI`): Interface for the second summand (input value 2).
- `IN3` (Type: `adapter::types::unidirectional::ALI`): Interface for the third addend (input value 3).
- **Plugs (Output Interfaces):**
- `OUT` (Type: `adapter::types::unidirectional::ALI`): Interface for outputting the calculated result.

Functionality

The function block `ALI_ADD_3_UNGATED` performs a standard arithmetic addition. As soon as new values are signaled at the input adapters (`IN1`, `IN2`, `IN3`), the function block calculates the sum using the following formula:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.35}$$

The result and the corresponding update event are then passed on to the subsequent function blocks via the output adapter `OUT`.

Technical Features

- **Generic Implementation:** The function block is declared as a generic type (`GEN_ALI_ADD`). This allows for flexible adaptation to various data types supported by the `ALI` adapter.
- **Use of Adapters:** By encapsulating data and events in unidirectional adapters (`ALI`), the number of necessary connection lines in the 4diac-ide application editor is drastically reduced, resulting in improved clarity.

State Overview

The function block operates purely stateless. There is no internal state machine (ECC). The calculation is purely data- or event-driven: An incoming event at one of the input adapters directly triggers the addition and updating of the output adapter.

Application Scenarios

- **Measurement Summing:** Summarizing three individual analog measurements (e.g., power consumption of three phases, flow rates from three pipes) into a total value.
- **Average Preparation:** Summing three values for subsequent division by 3 in a subsequent function block.
- **Structured Signal Processing:** Used in complex projects where analog signals need to be standardized and routed clearly via adapter channels.

Comparison with Similar Components

- **Standard ADD (IEC 61131-3):** A standard ADD component uses classic data and event connections. `ALI_ADD_3_UNGATED`, on the other hand, encapsulates these interfaces in adapters, reducing visual complexity in system design.
- **Cascaded Dual Adders:** To add three values with conventional dual adders, two components would have to be connected in series. `ALI_ADD_3_UNGATED` eliminates the need for one component as well as the intermediate instantiation and wiring.

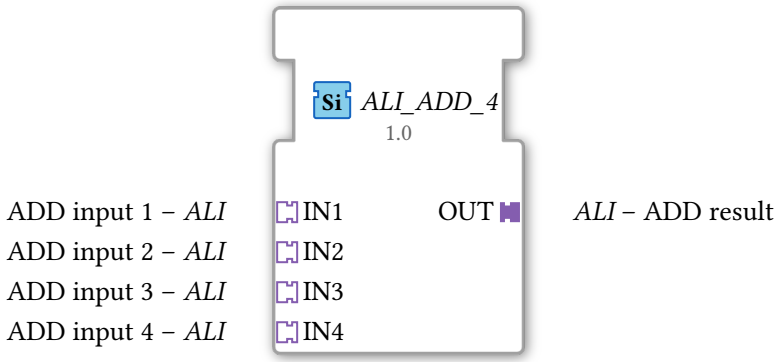
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

`ALI_ADD_3_UNGATED` is a compact and efficient auxiliary component for 4diac-ide. Through the consistent use of unidirectional adapters, he makes a significant contribution to the creation of clean, modularized and easily readable control code for processing analog signals.

ALI_ADD_4



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_ADD_4` is a generic arithmetic function block according to the IEC 61131-3 standard. Its main task is to calculate the sum of four input values. Unlike classic arithmetic blocks that work with elementary data types and standard event inputs/outputs, this block uses unidirectional adapter connections of type `ALI` (Analog Line Interface). This enables encapsulated and clear signal transmission of data and the associated trigger events within the 4diac-ide.

Interface Structure

Event Inputs

The block does not have direct event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Events are forwarded via the output adapter.

Data Inputs

This function block does not have direct data inputs. Data is transferred via the input adapters (sockets).

Data Outputs

This function block does not have direct data outputs. Data is output via the output adapter (plug).

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALI`): First addend for addition.
- **IN2** (Type: `adapter::types::unidirectional::ALI`): Second addend for addition.
- **IN3** (Type: `adapter::types::unidirectional::ALI`): Third addend for the addition.
- **IN4** (Type: `adapter::types::unidirectional::ALI`): Fourth addend for the addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALI`): Output of the calculated addition result.

Functionality

As soon as a new event with an updated data value is received at one of the input adapters (`IN1` to `IN4`), the function block performs the mathematical addition of the four values:

$$\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4} \tag{2.36}$$

The result of the calculation is immediately passed to the output adapter `OUT`, and a corresponding output event is triggered to inform subsequent function blocks in the control flow.

–

Technical Features

- **Generic Class:** The function block is based on the generic base class `GEN_ALI_ADD` (attribute: `eclipse4diac::core::GenericClassName`).
- **Adapter Coupling:** By using unidirectional adapters (`ALI`), data and events are bundled into a single connection. This drastically reduces the visual complexity (the "cable clutter") in the 4diac-ide Function Block Diagram (FBD) editor.
- **IEC 61131-3 Compliance:** Complies with the guidelines for standard arithmetic functions.

State Overview

The `ALI_ADD_4` function block is a stateless (combinatorial) function block. It does not store any internal states or historical values between calculation cycles. Each calculation is based solely on the values currently present at the input adapters at the time of the trigger event.

Application Scenarios

- **Measurement Consolidation:** Summing four analog measurements (e.g., partial load currents, flow rates, or individual power outputs) into a single total value.
- **Mixing Processes:** Calculating total weights or volumes in recipe systems where four components are combined.
- **Signal Processing:** Structured preprocessing of sensor signals in modular control architectures that consistently rely on adapter connections.

–

Comparison with Similar Components

- **Standard ADD Component (IEC 61131-3):** A standard ADD component uses elementary data types (e.g., `REAL`, `INT`) and separate event ports (`REQ` / `CNF`). `ALI_ADD_4`, on the other hand, encapsulates these interfaces in adapters, which increases reusability and clarity.
- **ALI_ADD_2 / ALI_ADD_3:** These function blocks operate on the same principle but are designed for adding only two or three input signals, respectively.

Change Detection

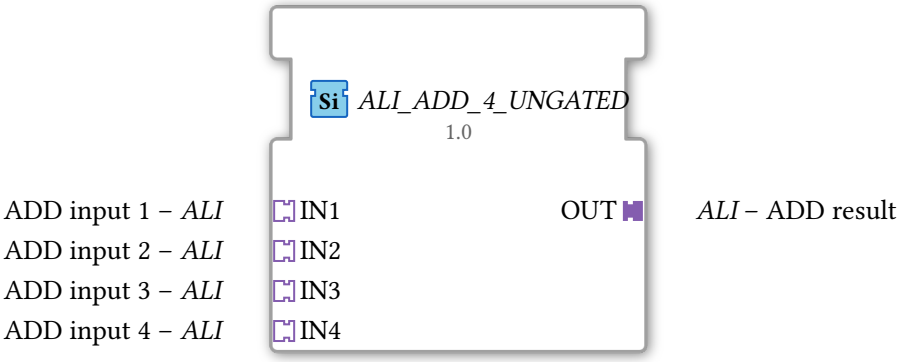
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALI_ADD_4` function block offers an elegant and clean way to add four analog values within an adapter-based 4diac application. By encapsulating data and event channels in `ALI` adapters, it supports the creation of modular, clear, and easily maintainable control software.

ALI_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_ADD_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_ADD_4_UNGATED` is a generic arithmetic function block according to the IEC 61131-3 standard. Its main task is to calculate the sum of four input values. Unlike classic arithmetic blocks that work with elementary data types and standard event inputs/outputs, this block uses unidirectional adapter connections of type `ALI` (Analog Line Interface). This enables encapsulated and clear signal transmission of data and the associated trigger events within the 4diac-ide.

Interface Structure

Event Inputs

The block does not have direct event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Events are forwarded via the output adapter.

Data Inputs

This function block does not have direct data inputs. Data is transferred via the input adapters (sockets).

Data Outputs

This function block does not have direct data outputs. Data is output via the output adapter (plug).

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALI`): First addend for addition.
- **IN2** (Type: `adapter::types::unidirectional::ALI`): Second addend for addition.
- **IN3** (Type: `adapter::types::unidirectional::ALI`): Third addend for the addition.
- **IN4** (Type: `adapter::types::unidirectional::ALI`): Fourth addend for the addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALI`): Output of the calculated addition result.

Functionality

As soon as a new event with an updated data value is received at one of the input adapters (`IN1` to `IN4`), the function block performs the mathematical addition of the four values:

$$\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4} \tag{2.37}$$

The result of the calculation is immediately passed to the output adapter `OUT`, and a corresponding output event is triggered to inform subsequent function blocks in the control flow.

–

Technical Features

- **Generic Class:** The function block is based on the generic base class `GEN_ALI_ADD` (attribute: `eclipse4diac::core::GenericClassName`).
- **Adapter Coupling:** By using unidirectional adapters (`ALI`), data and events are bundled into a single connection. This drastically reduces the visual complexity (the "cable clutter") in the 4diac-ide Function Block Diagram (FBD) editor.
- **IEC 61131-3 Compliance:** Complies with the guidelines for standard arithmetic functions.

State Overview

The `ALI_ADD_4_UNGATED` function block is a stateless (combinatorial) function block. It does not store any internal states or historical values between calculation cycles. Each calculation is based solely on the values currently present at the input adapters at the time of the trigger event.

Application Scenarios

- **Measurement Consolidation:** Summing four analog measurements (e.g., partial load currents, flow rates, or individual power outputs) into a single total value.
- **Mixing Processes:** Calculating total weights or volumes in recipe systems where four components are combined.
- **Signal Processing:** Structured preprocessing of sensor signals in modular control architectures that consistently rely on adapter connections.

–

Comparison with Similar Components

- **Standard ADD Component (IEC 61131-3):** A standard ADD component uses elementary data types (e.g., `REAL`, `INT`) and separate event ports (`REQ` / `CNF`). `ALI_ADD_4_UNGATED`, on the other hand, encapsulates these interfaces in adapters, which increases reusability and clarity.
- **ALI_ADD_2 / ALI_ADD_3:** These function blocks operate on the same principle but are designed for adding only two or three input signals, respectively.

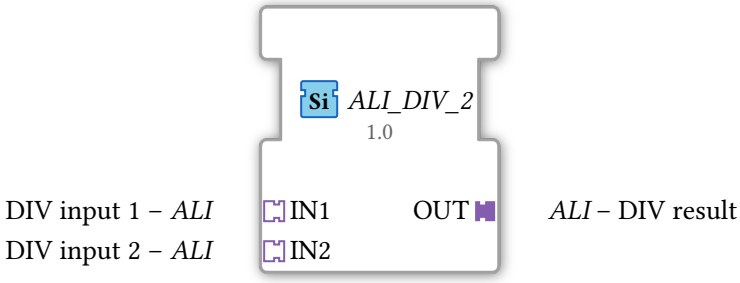
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALI_ADD_4_UNGATED` function block offers an elegant and clean way to add four analog values within an adapter-based 4diac application. By encapsulating data and event channels in `ALI` adapters, it supports the creation of modular, clear, and easily maintainable control software.

ALI_DIV_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_DIV_2` is a generic arithmetic function block for IEC 61499, designed for performing division operations. The unique feature of this block is the use of unidirectional adapters (`ALI`) instead of traditional data and event interfaces. This enables a modular and clear structuring of data flows within the 4diac IDE.

Interface Structure

Event Inputs
This function block does not have direct event inputs. Execution is controlled via the connected adapters.

Event Outputs
This function block does not have direct event outputs. Events are forwarded via the output adapter.

Data Inputs
This function block has no direct data inputs.

Data Outputs
This function block has no direct data outputs.

Adapters

Sockets (Input Adapters)
- **IN1** (Type: `adapter::types::unidirectional::ALI`): The dividend (numerator) of the division.
- **IN2** (Type: `adapter::types::unidirectional::ALI`): The divisor (denominator) of the division.

Plugs (Output Adapters)
- **OUT** (Type: `adapter::types::unidirectional::ALI`): The result of the mathematical division ($OUT = IN1/IN2$).

Functionality
The function block `ALI_DIV_2` calculates the quotient of the values provided via the input adapters `IN1` and `IN2`. The result is passed to the output adapter `OUT`.

Since it is a generic function block (`GEN_ALI_DIV`), the block can flexibly work with various data types, provided the `ALI` adapters support them (e.g., integers or floating-point numbers). The actual calculation is triggered by the events defined in the adapter. As soon as new data is present at the inputs and a corresponding trigger signal is received via the adapter interface, the division is performed and the result is provided at the output.

Because it is a generic function block (`GEN_ALI_DIV`), the block can flexibly work with different data types, provided the `ALI` adapters support them (e.g., integers or floating-point numbers). The actual calculation is triggered by the events defined in the adapter. As soon as new data is present at the inputs and a corresponding trigger signal is received via the adapter interface, the division is performed and the result is provided at the output.

... ## Technical Features

- **Generic Implementation:** By assigning it to the generic class `GEN_ALI_DIV`, the function block is not limited to a fixed data type (such as `REAL` or `INT`). It dynamically adapts to the type definitions of the connected adapters.
- **Adapter Encapsulation:** The use of unidirectional `ALI` adapters drastically reduces cabling in the 4diac IDE, as data and associated trigger events are bundled and transmitted in a single connection.
- – **Division by Zero:** During implementation and in the runtime environment, care must be taken to handle the value of `0` at input `IN2` to prevent system crashes (e.g., by default returning `NaN`, infinity, or by intercepting the error via adapter error handling).

State Overview
Since `ALI_DIV_2` is a data- and event-driven block, it does not have a complex internal state machine (no classic ECC). It behaves like a functional block:

1. **Waiting for Trigger:** The block waits for an update event at input adapters `IN1` or `IN2`.
2. **Calculation:** Upon signal reception, the data is read and divided
3. **Output:** The result is assigned to `OUT`, and the corresponding event is signaled via the output adapter.

Application Scenarios
- **Signal Normalization and Scaling:** Division of sensor values by constant factors for unit conversion within an adapter-based control architecture.

- **Ratio Calculation:** Determination of ratios (e.g., fuel-air mixture, flow ratios) in process engineering plants.
- **Distributed Control Systems:** Use in complex IEC 61499 applications where a clean, modular signal flow is enforced via adapters.

Comparison with Similar Function Blocks

Compared to a standard `DIV` function block according to IEC 61131-3 or IEC 61499, which uses classic data pins (`IN1`, `IN2`, `OUT`) and separate event pins (`REQ`, `CNF`), the `ALI_DIV_2` offers a significantly cleaner interface in the function block diagram through the use of adapters. It is particularly suitable for advanced software architectures that consistently leverage the advantages of adapters for structuring.

Change Detection

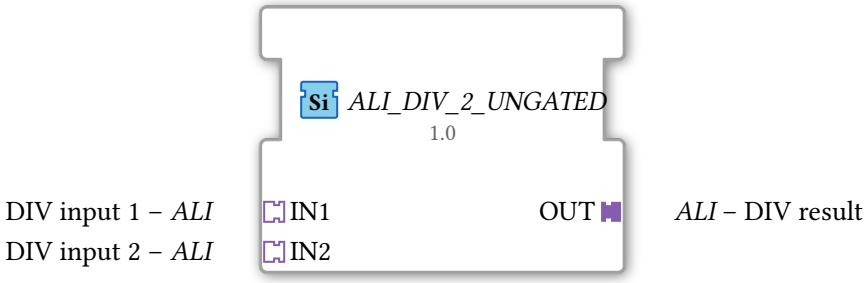
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALI_DIV_2` is a highly efficient and flexible division block for modern IEC 61499 applications. Through the consistent use of generic types and the `ALI` adapter concept, it significantly contributes to the clarity, reusability, and maintainability of control software in the 4diac IDE.

ALI_DIV_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_DIV_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_DIV_2_UNGATED` is a generic arithmetic function block for IEC 61499, designed for performing division operations. The unique feature of this block is the use of unidirectional adapters (`ALI`) instead of traditional data and event interfaces. This enables a modular and clear structuring of data flows within the 4diac IDE.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Execution is controlled via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Events are forwarded via the output adapter.

Data Inputs

This function block has no direct data inputs.

Data Outputs

This function block has no direct data outputs.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALI`): The dividend (numerator) of the division.
- **IN2** (Type: `adapter::types::unidirectional::ALI`): The divisor (denominator) of the division.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALI`): The result of the mathematical division ($OUT = IN1/IN2$).

Functionality

The function block `ALI_DIV_2_UNGATED` calculates the quotient of the values provided via the input adapters `IN1` and `IN2`. The result is passed to the output adapter `OUT`.

Since it is a generic function block (`GEN_ALI_DIV`), the block can flexibly work with various data types, provided the `ALI` adapters support them (e.g., integers or floating-point numbers). The actual calculation is triggered by the events defined in the adapter. As soon as new data is present at the inputs and a corresponding trigger signal is received via the adapter interface, the division is performed and the result is provided at the output.

Because it is a generic function block (`GEN_ALI_DIV`), the block can flexibly work with different data types, provided the `ALI` adapters support them (e.g., integers or floating-point numbers). The actual calculation is triggered by the events defined in the adapter. As soon as new data is present at the inputs and a corresponding trigger signal is received via the adapter interface, the division is performed and the result is provided at the output.

... ## Technical Features

- **Generic Implementation:** By assigning it to the generic class `GEN_ALI_DIV`, the function block is not limited to a fixed data type (such as `REAL` or `INT`). It dynamically adapts to the type definitions of the connected adapters.
- **Adapter Encapsulation:** The use of unidirectional `ALI` adapters drastically reduces cabling in the 4diac IDE, as data and associated trigger events are bundled and transmitted in a single connection.
- – **Division by Zero:** During implementation and in the runtime environment, care must be taken to handle the value of `0` at input `IN2` to prevent system crashes (e.g., by default returning `NaN`, infinity, or by intercepting the error via adapter error handling).

State Overview

Since `ALI_DIV_2_UNGATED` is a data- and event-driven block, it does not have a complex internal state machine (no classic ECC). It behaves like a functional block:

1. **Waiting for Trigger:** The block waits for an update event at input adapters `IN1` or `IN2`.
2. **Calculation:** Upon signal reception, the data is read and divided
3. **Output:** The result is assigned to `OUT`, and the corresponding event is signaled via the output adapter.

Application Scenarios

- **Signal Normalization and Scaling:** Division of sensor values by constant factors for unit conversion within an adapter-based control architecture.
- **Ratio Calculation:** Determination of ratios (e.g., fuel-air mixture, flow ratios) in process engineering plants.
- **Distributed Control Systems:** Use in complex IEC 61499 applications where a clean, modular signal flow is enforced via adapters.

Comparison with Similar Function Blocks

Compared to a standard `DIV` function block according to IEC 61131-3 or IEC 61499, which uses classic data pins (`IN1`, `IN2`, `OUT`) and separate event pins (`REQ`, `CNF`), the `ALI_DIV_2_UNGATED` offers a significantly cleaner interface in the function block diagram through the use of adapters. It is particularly suitable for advanced software architectures that consistently leverage the advantages of adapters for structuring.

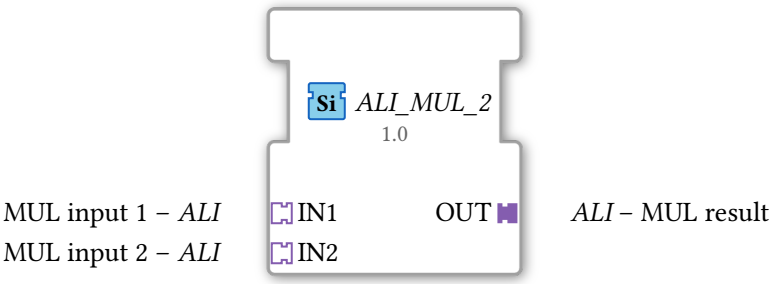
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALI_DIV_2_UNGATED` is a highly efficient and flexible division block for modern IEC 61499 applications. Through the consistent use of generic types and the `ALI` adapter concept, it significantly contributes to the clarity, reusability, and maintainability of control software in the 4diac IDE.

ALI_MUL_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_MUL_2` is used to perform an arithmetic multiplication of two values. It is a generic function block designed for a clean and modular architecture in IEC 61499 systems. Instead of classic discrete data and event ports, this block uses unidirectional adapters to transmit control events and data values in a bundled manner.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is integrated into the input adapters.

Event Outputs

There are no direct event outputs. Event control is integrated into the output adapter.

Data Inputs

There are no direct data inputs. Data transmission occurs via the adapters.

Data Outputs

There are no direct data outputs. Data transmission occurs via the output adapter.

Adapters

- **Sockets (Input Adapters):**
- `IN1` (Type: `adapter::types::unidirectional::ALI`): First input for multiplication (multiplicand).
- `IN2` (Type: `adapter::types::unidirectional::ALI`): Second input for multiplication (multiplier).
- **Plugs (Output Adapters):**
- `OUT` (Type: `adapter::types::unidirectional::ALI`): Output for the result of the multiplication (product).

Functionality

The function block `ALI_MUL_2` multiplies the values provided via the adapter sockets `IN1` and `IN2`. As soon as a corresponding trigger event is signaled via the input adapters, the block performs the calculation:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \tag{2.38}$$

The calculated result and the associated output event are then forwarded to subsequent blocks via the output plug `OUT`.

- **Generic Nature:** The function block is declared as `GEN_ALI_MUL` via the attribute `GenericClassName`. This allows for flexible handling of various data types specified by the higher-level system or the adapter type used.
- **Adapter Coupling:** By using the unidirectional adapter type `ALI`, data and trigger events are bundled in a single connection, significantly minimizing wiring effort in the development environment.

Technical Features

Since this is a purely data-flow and adapter-based function block, it does not have a complex internal state machine (Execution Control Chart - ECC). Execution is event-driven and directly coupled to the triggers received via the adapters.

State Overview

Adapter Coupling: ## Application Scenarios

Application Scenarios

- **Measurement Scaling:** Multiplication of sensor values by calibration or scaling factors.
- **Power Calculation:** Determination of physical quantities such as electrical power ($P = U \times I$) in adapter-based control networks.
- **Compact Computing Networks:** Use in complex mathematical calculations where the visual clarity of the control diagram must be maintained through the use of adapters.

Comparison with Similar Function Blocks

Compared to classic mathematical components of IEC 61131-3 (such as a standard `MUL` component with discrete `REQ` / `CNF` events and separate data lines), the `ALI_MUL_2` eliminates the need for manual linking of event and data paths. This prevents errors in event data mapping and results in a significantly cleaner and more readable application diagram in 4diac-ide.

Change Detection

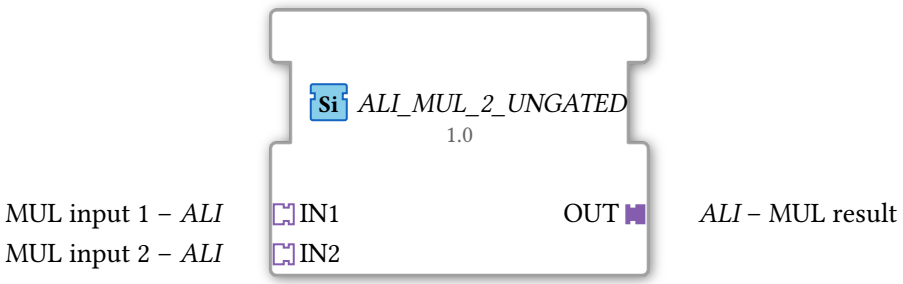
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALI_MUL_2` is a modern and efficient function block for multiplication. Thanks to the consistent use of unidirectional adapters, it is ideally suited for service-oriented and modular control systems where clarity and ease of maintenance are paramount.

ALI_MUL_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_MUL_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_MUL_2_UNGATED` is used to perform an arithmetic multiplication of two values. It is a generic function block designed for a clean and modular architecture in IEC 61499 systems. Instead of classic discrete data and event ports, this block uses unidirectional adapters to transmit control events and data values in a bundled manner.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is integrated into the input adapters.

Event Outputs

There are no direct event outputs. Event control is integrated into the output adapter.

Data Inputs

There are no direct data inputs. Data transmission occurs via the adapters.

Data Outputs

There are no direct data outputs. Data transmission occurs via the output adapter.

Adapters

- **Sockets (Input Adapters):**
- `IN1` (Type: `adapter::types::unidirectional::ALI`): First input for multiplication (multiplicand).
- `IN2` (Type: `adapter::types::unidirectional::ALI`): Second input for multiplication (multiplier).
- **Plugs (Output Adapters):**
- `OUT` (Type: `adapter::types::unidirectional::ALI`): Output for the result of the multiplication (product).

Functionality

The function block `ALI_MUL_2_UNGATED` multiplies the values provided via the adapter sockets `IN1` and `IN2`. As soon as a corresponding trigger event is signaled via the input adapters, the block performs the calculation:

$$\text{OUT} = \text{IN1} \times \text{IN2} \tag{2.39}$$

The calculated result and the associated output event are then forwarded to subsequent blocks via the output plug `OUT`.

- **Generic Nature:** The function block is declared as `GEN_ALI_MUL` via the attribute `GenericClassName`. This allows for flexible handling of various data types specified by the higher-level system or the adapter type used.
- **Adapter Coupling:** By using the unidirectional adapter type `ALI`, data and trigger events are bundled in a single connection, significantly minimizing wiring effort in the development environment.

Technical Features

Since this is a purely data-flow and adapter-based function block, it does not have a complex internal state machine (Execution Control Chart - ECC). Execution is event-driven and directly coupled to the triggers received via the adapters.

State Overview

Adapter Coupling: ## Application Scenarios

Application Scenarios

- **Measurement Scaling:** Multiplication of sensor values by calibration or scaling factors.
- **Power Calculation:** Determination of physical quantities such as electrical power ($P = U \times I$) in adapter-based control networks.
- **Compact Computing Networks:** Use in complex mathematical calculations where the visual clarity of the control diagram must be maintained through the use of adapters.

Comparison with Similar Function Blocks

Compared to classic mathematical components of IEC 61131-3 (such as a standard `MUL` component with discrete `REQ` / `CNF` events and separate data lines), the `ALI_MUL_2_UNGATED` eliminates the need for manual linking of event and data paths. This prevents errors in event data mapping and results in a significantly cleaner and more readable application diagram in 4diac-ide.

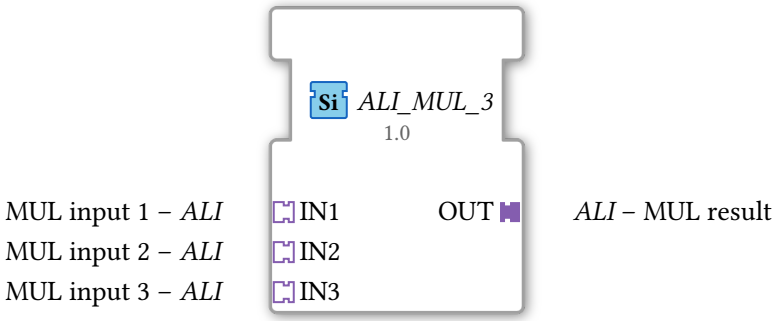
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALI_MUL_2_UNGATED` is a modern and efficient function block for multiplication. Thanks to the consistent use of unidirectional adapters, it is ideally suited for service-oriented and modular control systems where clarity and ease of maintenance are paramount.

ALI_MUL_3



No image available. The function block is graphically represented with three input sockets (IN1, IN2, IN3) and one output plug (OUT).

• – ◦ • – ◦ • – ◦ •

Introduction

The `ALI_MUL_3` function block is a generic arithmetic block for the 4diac IDE, used for multiplying three input values. It is characterized by the fact that the inputs and outputs are not implemented via classic data and event pins, but entirely via unidirectional adapters of type `ALI`. This enables structured, modular, and clear encapsulation of signal and data flows in IEC 61499 applications.

Since this functional block is entirely based on adapter technology, it has no direct, traditional event or data channels at the main level.

Interface Structure

Event Inputs

No direct event inputs are available (signal processing is handled by the adapters).

Event Outputs

No direct event outputs are available (signal forwarding is handled by the output adapter).

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

Sockets (Inputs)

- **IN1** (Type: `adapter::types::unidirectional::ALI`): The first factor (multiplicand 1) for arithmetic multiplication.

- **IN3** (Type: `adapter::types::unidirectional::ALI`): The third factor (multiplicand 3) for arithmetic multiplication.

Plugs (Outputs)

- **OUT** (Type: `adapter::types::unidirectional::ALI`): The result of the multiplication (product).

Functionality

The primary task of `ALI_MUL_3` is to calculate the product of three values:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \tag{2.40}$$

The calculation timing and the transfer of values are controlled by the `ALI` adapters. As soon as a corresponding trigger event is signaled via the input adapters, the function block reads the values from sockets `IN1`, `IN2`, and `IN3`, calculates the mathematical product, and provides the result at plug `OUT`, whereupon a corresponding output event is initiated.

- **Generic Building Block:** The attribute `GenericClassName` with the value `'GEN_ALI_MUL'` makes the function block data-type independent. It can be used for various numeric data types (e.g., `INT`, `REAL`, `LREAL`) depending on the specific `ALI` adapter used.
- **Clean Design:** The use of adapters significantly reduces wiring complexity in the function block diagram (FBD) because control and data flow are combined in a single connection.
- – **Unidirectional Adapters:** Since this is the `adapter::types::unidirectional` package, data and triggers flow directly from the sockets (`IN1`-`IN3`) to the plug (`OUT`).

The function block behaves like a classic, stateless mathematical operator (combinatorial behavior):

- **Readiness:** The block waits for incoming update events at the adapters `IN1`, `IN2`, or `IN3`.
- **Calculation:** When triggered, the multiplication is performed.
- – **Output:** The result is directly passed to output `OUT`, triggering the subsequent function block.

- **Calculation of Physical Quantities:** Determining volumes (length × width × height) or three-dimensional scales in factory automation.
- **Cascaded Gain Factors:** Multiplication of three different gain factors or correction values in control loops (e.g., sensor value × calibration factor × scaling factor).
- **Signal Processing:** Combining multiple weighting factors in process monitoring.
-

Compared to a standard IEC 61131-3 `MUL` component:

- **Classic `MUL` block:** Requires explicit `REQ` and `CNF` events, as well as individual data pins for each operand. This quickly leads to complex schematics with multiple inputs.
- **`ALI_MUL_3`:** Consolidates the interfaces into four clearly arranged adapter connections. Furthermore, it is designed for exactly three inputs, eliminating the need for an intermediate stage (cascading two 2x multipliers).

The `ALI_MUL_3` is a modern and highly efficient function block for mathematical operations in the 4diac environment. Through the consistent use of unidirectional adapters, it promotes clean, modularized software design and is ideally suited for complex calculations with three input variables.

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

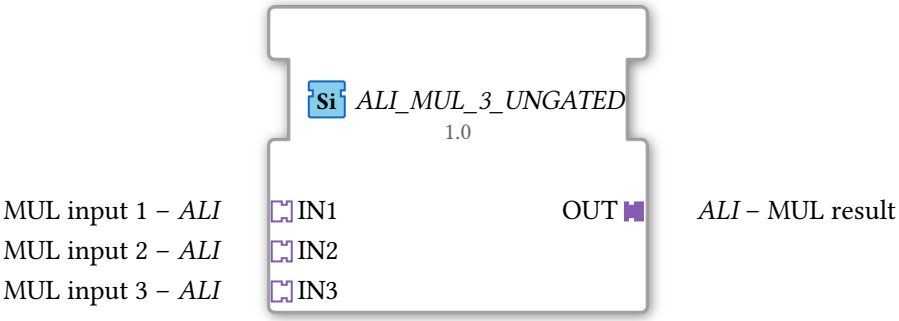
Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

ALI_MUL_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_MUL_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



No image available. The function block is graphically represented with three input sockets (IN1, IN2, IN3) and one output plug (OUT).

• – ◦ • – ◦ • – ◦ •

Introduction

The `ALI_MUL_3_UNGATED` function block is a generic arithmetic block for the 4diac IDE, used for multiplying three input values. It is characterized by the fact that the inputs and outputs are not implemented via classic data and event pins, but entirely via unidirectional adapters of type `ALI`. This enables structured, modular, and clear encapsulation of signal and data flows in IEC 61499 applications.

Since this functional block is entirely based on adapter technology, it has no direct, traditional event or data channels at the main level.

Interface Structure

Event Inputs

No direct event inputs are available (signal processing is handled by the adapters).

Event Outputs

No direct event outputs are available (signal forwarding is handled by the output adapter).

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

Sockets (Inputs)

- **IN1** (Type: `adapter::types::unidirectional::ALI`): The first factor (multiplicand 1) for arithmetic multiplication.
- **IN3** (Type: `adapter::types::unidirectional::ALI`): The third factor (multiplicand 3) for arithmetic multiplication.

Plugs (Outputs)

- **OUT** (Type: `adapter::types::unidirectional::ALI`): The result of the multiplication (product).

Functionality

The primary task of `ALI_MUL_3_UNGATED` is to calculate the product of three values:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \tag{2.41}$$

The calculation timing and the transfer of values are controlled by the `ALI` adapters. As soon as a corresponding trigger event is signaled via the input adapters, the function block reads the values from sockets `IN1`, `IN2`, and `IN3`, calculates the mathematical product, and provides the result at plug `OUT`, whereupon a corresponding output event is initiated.

- **Generic Building Block:** The attribute `GenericClassName` with the value `'GEN_ALI_MUL'` makes the function block data-type independent. It can be used for various numeric data types (e.g., `INT`, `REAL`, `LREAL`) depending on the specific `ALI` adapter used.
- **Clean Design:** The use of adapters significantly reduces wiring complexity in the function block diagram (FBD) because control and data flow are combined in a single connection.
- – **Unidirectional Adapters:** Since this is the `adapter::types::unidirectional` package, data and triggers flow directly from the sockets (`IN1`-`IN3`) to the plug (`OUT`).

The function block behaves like a classic, stateless mathematical operator (combinatorial behavior):

- **Readiness:** The block waits for incoming update events at the adapters `IN1`, `IN2`, or `IN3`.
- **Calculation:** When triggered, the multiplication is performed.
- – **Output:** The result is directly passed to output `OUT`, triggering the subsequent function block.

- **Calculation of Physical Quantities:** Determining volumes (length × width × height) or three-dimensional scales in factory automation.
- **Cascaded Gain Factors:** Multiplication of three different gain factors or correction values in control loops (e.g., sensor value × calibration factor × scaling factor).
- **Signal Processing:** Combining multiple weighting factors in process monitoring.
-

Compared to a standard IEC 61131-3 `MUL` component:

- **Classic `MUL` block:** Requires explicit `REQ` and `CNF` events, as well as individual data pins for each operand. This quickly leads to complex schematics with multiple inputs.
- **`ALI_MUL_3_UNGATED`:** Consolidates the interfaces into four clearly arranged adapter connections. Furthermore, it is designed for exactly three inputs, eliminating the need for an intermediate stage (cascading two 2x multipliers).

The `ALI_MUL_3_UNGATED` is a modern and highly efficient function block for mathematical operations in the 4diac environment. Through the consistent use of unidirectional adapters, it promotes clean, modularized software design and is ideally suited for complex calculations with three input variables.

Technical Features

State Overview

Application Scenarios

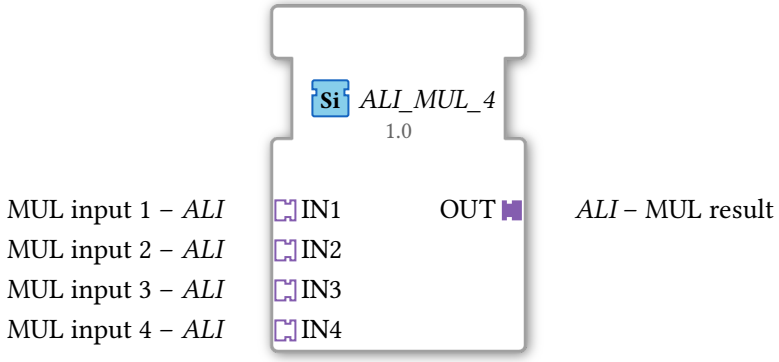
Comparison with Similar Function Blocks

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

ALI_MUL_4



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_MUL_4` is used to perform an arithmetic multiplication of four input values. It is a generic function block (Generic FB) based on the use of unidirectional adapters (`ALI`). By using adapters instead of classic data and event pins, the wiring in 4diac-ide is made clearer and more modular.

Interface Structure

Event Inputs

This function block has no direct event inputs. Control and triggering are handled via the connected adapters.

Event Outputs

This function block has no direct event outputs. Events are forwarded via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

The sockets serve as inputs for the values to be multiplied.

- **IN1** (Type: `adapter::types::unidirectional::ALI`): First factor for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::ALI`): Second factor for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::ALI`): Third factor for multiplication.
- **IN4** (Type: `adapter::types::unidirectional::ALI`): Fourth factor for multiplication.

Plugs (Output Adapters)

The plug outputs the calculated result.

- **OUT** (Type: `adapter::types::unidirectional::ALI`): The calculated product of the four input adapters.

Functionality

As soon as the input values at adapters `IN1` to `IN4` change, or a corresponding update event is triggered via the adapters, the function block internally multiplies the four values according to the following mathematical principle:

$$OUT = \text{IN1} \times \text{IN2} \times \text{IN3} \times \text{IN4} \tag{2.42}$$

The result and the associated update event are then output via the output adapter `OUT`.

–

Technical Features

- **Generic Function Block:** The attribute `GenericClassName` with the value `GEN_ALI_MUL` makes the function block data type-independent. Depending on the implementation of the `ALI` adapters, it can process various numeric data types (e.g., `INT`, `REAL`, `LREAL`).
- **Adapter Coupling:** By using unidirectional `ALI` adapters, clean encapsulation of data and trigger events is achieved, reducing complexity in system design.

State Overview

The function block is stateless (stateless, purely combinatorial processing). There are no internal state machines (Execution Control Chart / ECC). The output values depend directly on the values applied to the input adapters.

Application Scenarios

- **Scaling and Weighting:** Calculation of composite scaling factors in process automation.
- **Volume and Mass Calculation:** Continuous calculation of physical quantities (e.g., length × width × height × density).
- **Multi-Stage Gain Control:** Cascaded signal amplification in measurement and control technology.

–

Comparison with Similar Function Blocks

Compared to a standard multiplication function block (`MUL`) according to IEC 61131-3, which typically has only two direct data inputs, `ALI_MUL_4` offers:

1. Direct multiplication of **four** factors in a single step (fewer function blocks required on the control canvas).
2. The use of adapters instead of individual wiring significantly improves the overall clarity of the program.

Change Detection

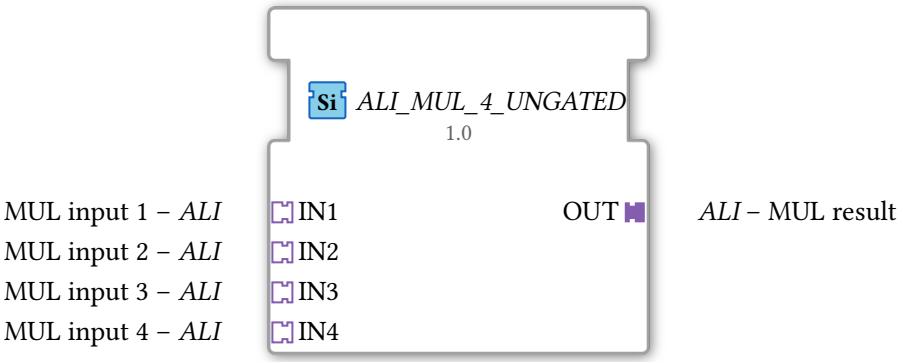
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALI_MUL_4` is an efficient auxiliary module for arithmetic calculations in complex 4diac systems. It is particularly suitable for applications where multiple values need to be multiplied compactly and systematically without cluttering the user interface with countless connection lines.

ALI_MUL_4_UNGATED

i UNGATED variant: This block is the ungated version of `ALI_MUL_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALI_MUL_4_UNGATED` is used to perform an arithmetic multiplication of four input values. It is a generic function block (Generic FB) based on the use of unidirectional adapters (`ALI`). By using adapters instead of classic data and event pins, the wiring in 4diac-ide is made clearer and more modular.

Interface Structure

Event Inputs

This function block has no direct event inputs. Control and triggering are handled via the connected adapters.

Event Outputs

This function block has no direct event outputs. Events are forwarded via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

The sockets serve as inputs for the values to be multiplied.

- **IN1** (Type: `adapter::types::unidirectional::ALI`): First factor for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::ALI`): Second factor for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::ALI`): Third factor for multiplication.
- **IN4** (Type: `adapter::types::unidirectional::ALI`): Fourth factor for multiplication.

Plugs (Output Adapters)

The plug outputs the calculated result.

- **OUT** (Type: `adapter::types::unidirectional::ALI`): The calculated product of the four input adapters.

Functionality

As soon as the input values at adapters `IN1` to `IN4` change, or a corresponding update event is triggered via the adapters, the function block internally multiplies the four values according to the following mathematical principle:

$$OUT = \text{IN1} \times \text{IN2} \times \text{IN3} \times \text{IN4} \tag{2.43}$$

The result and the associated update event are then output via the output adapter `OUT`.

–

Technical Features

- **Generic Function Block:** The attribute `GenericClassName` with the value `GEN_ALI_MUL` makes the function block data type-independent. Depending on the implementation of the `ALI` adapters, it can process various numeric data types (e.g., `INT`, `REAL`, `LREAL`).
- **Adapter Coupling:** By using unidirectional `ALI` adapters, clean encapsulation of data and trigger events is achieved, reducing complexity in system design.

State Overview

The function block is stateless (stateless, purely combinatorial processing). There are no internal state machines (Execution Control Chart / ECC). The output values depend directly on the values applied to the input adapters.

Application Scenarios

- **Scaling and Weighting:** Calculation of composite scaling factors in process automation.
- **Volume and Mass Calculation:** Continuous calculation of physical quantities (e.g., length × width × height × density).
- **Multi-Stage Gain Control:** Cascaded signal amplification in measurement and control technology.

–

Comparison with Similar Function Blocks

Compared to a standard multiplication function block (`MUL`) according to IEC 61131-3, which typically has only two direct data inputs, `ALI_MUL_4_UNGATED` offers:

1. Direct multiplication of **four** factors in a single step (fewer function blocks required on the control canvas).
2. The use of adapters instead of individual wiring significantly improves the overall clarity of the program.

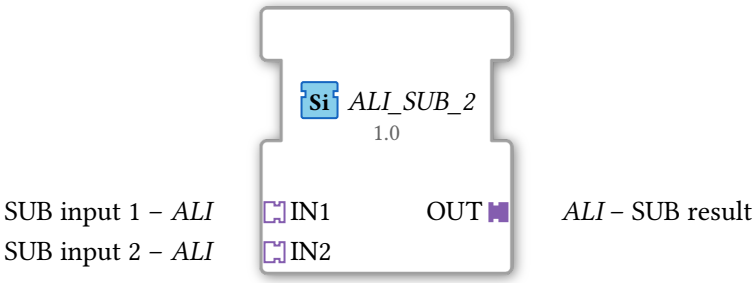
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALI_MUL_4_UNGATED` is an efficient auxiliary module for arithmetic calculations in complex 4diac systems. It is particularly suitable for applications where multiple values need to be multiplied compactly and systematically without cluttering the user interface with countless connection lines.

ALI_SUB_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) `ALI_SUB_2` is a generic arithmetic function block for performing a subtraction. It conforms to the IEC 61131-3 standard for standard arithmetic functions. The block uses adapter connections (`ALI` - Arithmetic Logical Interface) to receive input data and output the result of the subtraction.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Control and event processing are implemented via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Event forwarding occurs via the output adapters.

Data Inputs

There are no direct data inputs. Data transmission is encapsulated via the input adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALI`): The first input (minuend) for subtraction calculation.
- **IN2** (Type: `adapter::types::unidirectional::ALI`): The second input (subtrahend) for subtraction calculation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALI`): The output (difference) of the calculation ($OUT = IN1 - IN2$).

Functionality

The function block `ALI_SUB_2` performs a mathematical subtraction. As soon as values are received or updated via the input adapters, the function block calculates the difference:

$$\text{OUT} = \text{IN1} - \text{IN2} \tag{2.44}$$

The result is immediately passed to the output adapter `OUT`. Since this is a generic function block based on the internal class `GEN_ALI_SUB`, the function block can flexibly adapt to the data types defined in the adapter.

Technical Features

- **Generic Implementation:** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_ALI_SUB'`. This allows for flexible handling of various data types defined by the adapter interfaces.
- **Encapsulation by Adapters:** By using unidirectional `ALI` adapters, data and the associated control events (such as update events) are bundled in a single connection. This reduces the number of visible connection lines in the 4diac application editor and improves clarity.

–

State Overview

The function block behaves in a purely stateless and reactive manner. There is no internal state machine (Execution Control Chart - ECC). Processing is triggered directly by incoming data changes or events on the input adapters `IN1` and `IN2`.

Application Scenarios

- **Signal Processing and Scaling:** Calculation of deviations, such as the difference between a setpoint and an actual value (control deviation) in control applications.
- **Offset Compensation:** Subtraction of zero-point errors or offsets from sensor data fed into the system via adapter structures.
- **Modular System Architectures:** Use in complex, distributed control systems where a structured and transparent data flow is required through the use of adapters.
-

Comparison with Similar Function Blocks

Compared to the standard subtraction function block `SUB` from the IEC 61131-3 library, `ALI_SUB_2` does not require explicit wiring of data types (like `INT`, `REAL`) and control events (like `REQ` / `CNF`). While the standard `SUB` function block has discrete inputs and outputs, `ALI_SUB_2` combines these in `ALI` adapters, which significantly improves reusability and clarity in system design.

Change Detection

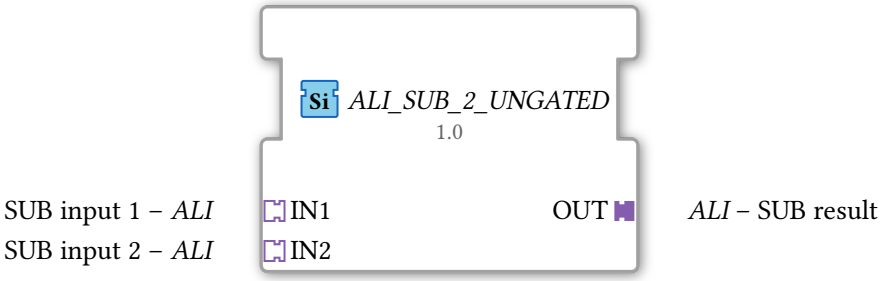
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALI_SUB_2` is a versatile and easy-to-maintain function block for arithmetic subtraction in 4diac. Thanks to the consistent use of adapters, it integrates perfectly into service-oriented and modular control architectures and minimizes wiring effort in the development tool.

ALI_SUB_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_SUB_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) `ALI_SUB_2_UNGATED` is a generic arithmetic function block for performing a subtraction. It conforms to the IEC 61131-3 standard for standard arithmetic functions. The block uses adapter connections (`ALI` - Arithmetic Logical Interface) to receive input data and output the result of the subtraction.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Control and event processing are implemented via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Event forwarding occurs via the output adapters.

Data Inputs

There are no direct data inputs. Data transmission is encapsulated via the input adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALI`): The first input (minuend) for subtraction calculation.
- **IN2** (Type: `adapter::types::unidirectional::ALI`): The second input (subtrahend) for subtraction calculation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALI`): The output (difference) of the calculation ($OUT = IN1 - IN2$).

Functionality

The function block `ALI_SUB_2_UNGATED` performs a mathematical subtraction. As soon as values are received or updated via the input adapters, the function block calculates the difference:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.45}$$

The result is immediately passed to the output adapter `OUT`. Since this is a generic function block based on the internal class `GEN_ALI_SUB`, the function block can flexibly adapt to the data types defined in the adapter.

Technical Features

- **Generic Implementation:** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_ALI_SUB'`. This allows for flexible handling of various data types defined by the adapter interfaces.
- **Encapsulation by Adapters:** By using unidirectional `ALI` adapters, data and the associated control events (such as update events) are bundled in a single connection. This reduces the number of visible connection lines in the 4diac application editor and improves clarity.

–

State Overview

The function block behaves in a purely stateless and reactive manner. There is no internal state machine (Execution Control Chart - ECC). Processing is triggered directly by incoming data changes or events on the input adapters `IN1` and `IN2`.

Application Scenarios

- **Signal Processing and Scaling:** Calculation of deviations, such as the difference between a setpoint and an actual value (control deviation) in control applications.
- **Offset Compensation:** Subtraction of zero-point errors or offsets from sensor data fed into the system via adapter structures.
- **Modular System Architectures:** Use in complex, distributed control systems where a structured and transparent data flow is required through the use of adapters.
-

Comparison with Similar Function Blocks

Compared to the standard subtraction function block `SUB` from the IEC 61131-3 library, `ALI_SUB_2_UNGATED` does not require explicit wiring of data types (like `INT`, `REAL`) and control events (like `REQ` / `CNF`). While the standard `SUB` function block has discrete inputs and outputs, `ALI_SUB_2_UNGATED` combines these in `ALI` adapters, which significantly improves reusability and clarity in system design.

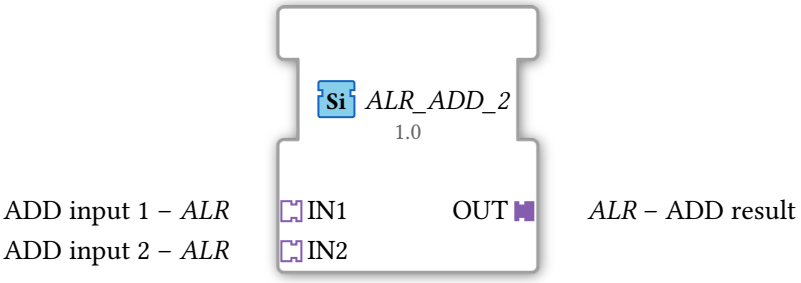
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALI_SUB_2_UNGATED` is a versatile and easy-to-maintain function block for arithmetic subtraction in 4diac. Thanks to the consistent use of adapters, it integrates perfectly into service-oriented and modular control architectures and minimizes wiring effort in the development tool.

ALR_ADD_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_ADD_2` is a generic, arithmetic addition block for the 4diac IDE, compliant with the IEC 61131-3 standard. It is used to mathematically add two input values. The special feature of this block lies in the use of unidirectional adapters (`ALR`), which encapsulate both the data and the associated control logic to enable clean and modular signal processing.

Interface Structure

Event Inputs
This function block does not have direct, dedicated event inputs. Event control is handled via the adapters.

Event Outputs
This function block does not have direct, dedicated event outputs. Event control is handled via the adapters.

Data Inputs
No direct data inputs are defined, as data transfer is handled entirely via the adapters.

Data Outputs
No direct data outputs are defined, as the result is provided via the output adapter.

Adapters

Sockets (Input Adapters)
- **IN1** (Type: `adapter::types::unidirectional::ALR`): The first addend of the addition operation.
- **IN2** (Type: `adapter::types::unidirectional::ALR`): The second addend of the addition operation.

Plugs (Output Adapters)
- **OUT** (Type: `adapter::types::unidirectional::ALR`): The result of the addition ($OUT = IN1 + IN2$).

Functionality
The `ALR_ADD_2` reads the values via the two input adapters `IN1` and `IN2`. As soon as a corresponding event signals via the adapters that new data is available, the function block performs the addition:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.46}$$

The calculated result is then forwarded via the output adapter `OUT`, along with the corresponding output event, to subsequent function blocks. Since it is a generic function block (`GEN_ALR_ADD`), it can flexibly work with various numeric data types supported by the ALR adapter interface.

Technical Features
- **Generic Type (`GEN_ALR_ADD`):** The function block is implemented in a data-type-independent manner and adapts to the data types defined in the ALR adapter.
- **Adapter Encapsulation:** By using unidirectional adapters, the number of visible connections in the 4diac IDE's control flow diagram is drastically reduced, thus improving the clarity and maintainability of the overall system.
- **Compliance:** Complies with the mathematical standards according to IEC 61131-3 for standard arithmetic functions.
-

State Overview
Since the logic of this function block operates purely data- and event-driven via the adapters, the block does not have a complex internal state machine (ECC). It reacts immediately to incoming data changes/ events at sockets `IN1` and `IN2` and forwards the result to plug `OUT`.

–

Application Scenarios
- **Signal Combining:** Addition of two analog sensor values (e.g., determining a total volume flow rate from two individual flow meters).
- **Setpoint Offset Calculation:** Applying a correction offset to a base setpoint in process engineering plants.
- **Compact Programming:** Use in complex control loops where clear routing via adapters is preferred.

Comparison with Similar Function Blocks

Compared to a standard `ADD` function block (which has separate event inputs like `REQ`, event outputs like `CNF`, and classic data pins), the `ALR_ADD_2` completely omits these classic pins. It replaces them with standardized `ALR` adapters. This significantly simplifies wiring in the 4diac IDE, as event and data are bundled in a single connection.

–

Change Detection

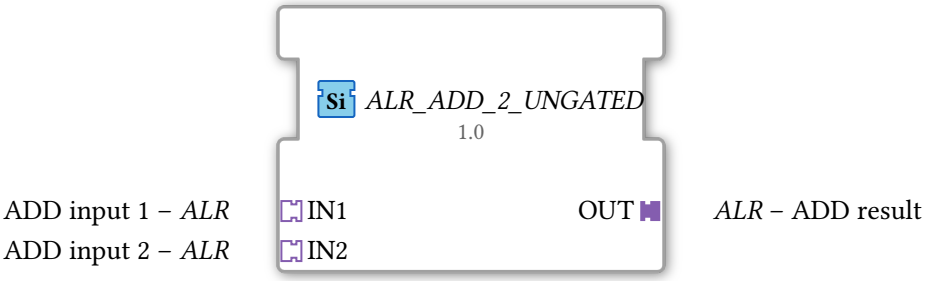
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALR_ADD_2` is a modern and highly efficient variant of the classic addition function block. It is ideally suited for service-oriented architectures and distributed control systems within the 4diac environment, where clarity and standardized adapter interfaces are paramount.

ALR_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_ADD_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_ADD_2_UNGATED` is a generic, arithmetic addition block for the 4diac IDE, compliant with the IEC 61131-3 standard. It is used to mathematically add two input values. The special feature of this block lies in the use of unidirectional adapters (`ALR`), which encapsulate both the data and the associated control logic to enable clean and modular signal processing.

Interface Structure

Event Inputs

This function block does not have direct, dedicated event inputs. Event control is handled via the adapters.

Event Outputs

This function block does not have direct, dedicated event outputs. Event control is handled via the adapters.

Data Inputs

No direct data inputs are defined, as data transfer is handled entirely via the adapters.

Data Outputs

No direct data outputs are defined, as the result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALR`): The first addend of the addition operation.
- **IN2** (Type: `adapter::types::unidirectional::ALR`): The second addend of the addition operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALR`): The result of the addition ($OUT = IN1 + IN2$).

Functionality

The `ALR_ADD_2_UNGATED` reads the values via the two input adapters `IN1` and `IN2`. As soon as a corresponding event signals via the adapters that new data is available, the function block performs the addition:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.47}$$

The calculated result is then forwarded via the output adapter `OUT`, along with the corresponding output event, to subsequent function blocks. Since it is a generic function block (`GEN_ALR_ADD`), it can flexibly work with various numeric data types supported by the ALR adapter interface.

Technical Features

- **Generic Type (`GEN_ALR_ADD`):** The function block is implemented in a data-type-independent manner and adapts to the data types defined in the ALR adapter.
- **Adapter Encapsulation:** By using unidirectional adapters, the number of visible connections in the 4diac IDE's control flow diagram is drastically reduced, thus improving the clarity and maintainability of the overall system.
- **Compliance:** Complies with the mathematical standards according to IEC 61131-3 for standard arithmetic functions.
-

State Overview

Since the logic of this function block operates purely data- and event-driven via the adapters, the block does not have a complex internal state machine (ECC). It reacts immediately to incoming data changes/events at sockets `IN1` and `IN2` and forwards the result to plug `OUT`.

–

Application Scenarios

- **Signal Combining:** Addition of two analog sensor values (e.g., determining a total volume flow rate from two individual flow meters).
- **Setpoint Offset Calculation:** Applying a correction offset to a base setpoint in process engineering plants.
- **Compact Programming:** Use in complex control loops where clear routing via adapters is preferred.

Comparison with Similar Function Blocks

Compared to a standard `ADD` function block (which has separate event inputs like `REQ`, event outputs like `CNF`, and classic data pins), the `ALR_ADD_2_UNGATED` completely omits these classic pins. It replaces them with standardized `ALR` adapters. This significantly simplifies wiring in the 4diac IDE, as event and data are bundled in a single connection.

–

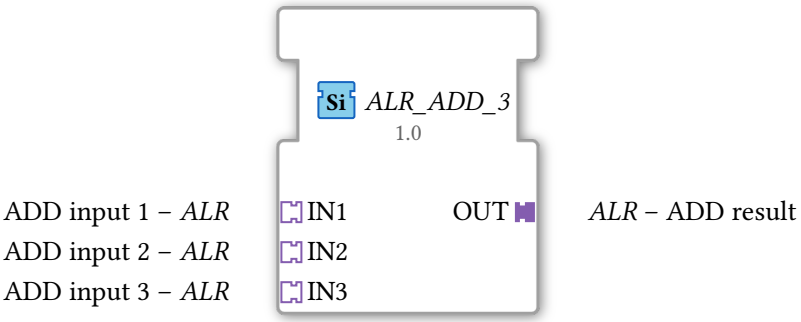
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALR_ADD_2_UNGATED` is a modern and highly efficient variant of the classic addition function block. It is ideally suited for service-oriented architectures and distributed control systems within the 4diac environment, where clarity and standardized adapter interfaces are paramount.

ALR_ADD_3



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_ADD_3` is a generic arithmetic block for the 4diac IDE, used to add three input values. It conforms to the standard arithmetic functions according to IEC 61131-3, but uses an adapter-based interface design. Instead of individual data and event pins, this block uses unidirectional adapters to enable clean encapsulation and clearer wiring within the application.

Interface Structure

Event Inputs

No direct event inputs are available. Event-based control is handled internally via the adapters.

Event Outputs

No direct event outputs are available. Event forwarding is handled via the output adapter.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Interfaces)

- **IN1** (Type: `adapter::types::unidirectional::ALR`): First addend of the addition operation.
- **IN2** (Type: `adapter::types::unidirectional::ALR`): Second addend of the addition operation.
- **IN3** (Type: `adapter::types::unidirectional::ALR`): Third addend of the addition operation.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::ALR`): Output adapter that provides the result of the addition (`IN1 + IN2 + IN3`) and the associated trigger event.

Functionality

This function block adds the values provided via the three input adapters (`IN1`, `IN2`, and `IN3`).

As soon as a new event or value is registered at one of the input adapters, the function block performs the calculation according to the following scheme:

$$OUT = IN1 + IN2 + IN3 \tag{2.48}$$

The result is immediately passed to the output adapter `OUT`, and a corresponding update event is triggered to initiate subsequent function blocks in the data flow.

Technical Features

- **Generic Function Block:** The attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_ALR_ADD'` makes the function block data type-independent (generic). It can therefore work flexibly with various numeric data types (e.g., INT, REAL, LREAL), provided the adapter type used, `ALR`, supports them.
- **Adapter Coupling:** Using adapters instead of traditional data and event pins drastically reduces the number of visible connection lines in the 4diac Application Editor. This significantly improves the readability of complex control diagrams.

State Overview

Since `ALR_ADD_3` is a mathematical, data-flow-oriented function block, it does not have a complex internal state machine (ECC). Execution is purely reactive (event-driven via the input adapters).

Application Scenarios

- **Measurement Summing:** Adding three analog partial flows to determine the total flow in an energy distribution system.
- **Mixing Calculations:** Combining three flow rates in a process plant to calculate the total inflow.
- **Setpoint Generation:** Calculating an overall setpoint from a base value, a correction value, and an offset.

Comparison with Similar Components

- **Standard `ADD` (IEC 61131-3):** The classic `ADD` component uses separate pins for data and events (REQ/CNF). In contrast, `ALR_ADD_3` combines these signals in adapters, increasing modularity and reusability in cross-project designs.
- **`ALR_ADD_2`:** A similar adapter-based component, but it only supports two inputs. `ALR_ADD_3` eliminates cascading when adding exactly three values, thus saving additional memory and computational effort.

Change Detection

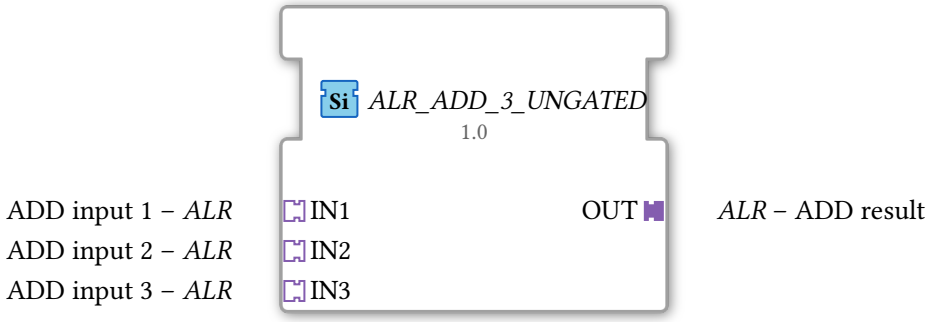
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALR_ADD_3` is a flexible and high-performance component for adding three values in 4diac environments. Through its generic design and the consistent use of unidirectional adapters, it makes a significant contribution to the clarity, structuring and maintainability of modern, distributed control programs.

ALR_ADD_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_ADD_3_UNGATED` is a generic arithmetic block for the 4diac IDE, used to add three input values. It conforms to the standard arithmetic functions according to IEC 61131-3, but uses an adapter-based interface design. Instead of individual data and event pins, this block uses unidirectional adapters to enable clean encapsulation and clearer wiring within the application.

Interface Structure

Event Inputs

No direct event inputs are available. Event-based control is handled internally via the adapters.

Event Outputs

No direct event outputs are available. Event forwarding is handled via the output adapter.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Interfaces)

- **IN1** (Type: `adapter::types::unidirectional::ALR`): First addend of the addition operation.
- **IN2** (Type: `adapter::types::unidirectional::ALR`): Second addend of the addition operation.
- **IN3** (Type: `adapter::types::unidirectional::ALR`): Third addend of the addition operation.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::ALR`): Output adapter that provides the result of the addition (`IN1 + IN2 + IN3`) and the associated trigger event.

Functionality

This function block adds the values provided via the three input adapters (`IN1`, `IN2`, and `IN3`).

As soon as a new event or value is registered at one of the input adapters, the function block performs the calculation according to the following scheme:

$$OUT = IN1 + IN2 + IN3 \tag{2.49}$$

The result is immediately passed to the output adapter `OUT`, and a corresponding update event is triggered to initiate subsequent function blocks in the data flow.

Technical Features

- **Generic Function Block:** The attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_ALR_ADD'` makes the function block data type-independent (generic). It can therefore work flexibly

with various numeric data types (e.g., INT, REAL, LREAL), provided the adapter type used, ALR, supports them.
- **Adapter Coupling:** Using adapters instead of traditional data and event pins drastically reduces the number of visible connection lines in the 4diac Application Editor. This significantly improves the readability of complex control diagrams.

State Overview

Since ALR_ADD_3_UNGATED is a mathematical, data-flow-oriented function block, it does not have a complex internal state machine (ECC). Execution is purely reactive (event-driven via the input adapters).

Application Scenarios

- **Measurement Summing:** Adding three analog partial flows to determine the total flow in an energy distribution system.
- **Mixing Calculations:** Combining three flow rates in a process plant to calculate the total inflow.
- **Setpoint Generation:** Calculating an overall setpoint from a base value, a correction value, and an offset.

Comparison with Similar Components

- **Standard ADD (IEC 61131-3):** The classic ADD component uses separate pins for data and events (REQ/CNF). In contrast, ALR_ADD_3_UNGATED combines these signals in adapters, increasing modularity and reusability in cross-project designs.
- **ALR_ADD_2:** A similar adapter-based component, but it only supports two inputs. ALR_ADD_3_UNGATED eliminates cascading when adding exactly three values, thus saving additional memory and computational effort.

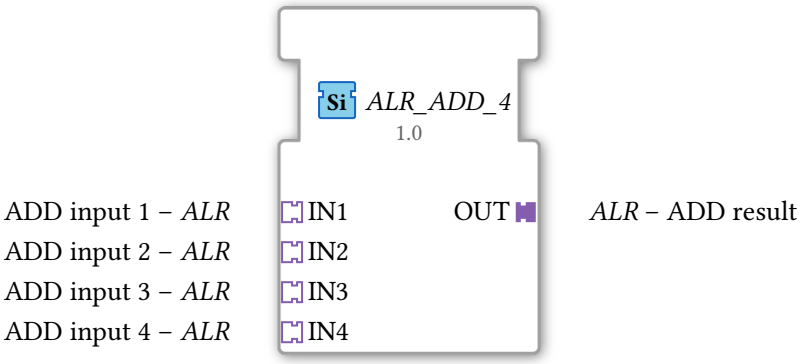
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The ALR_ADD_3_UNGATED is a flexible and high-performance component for adding three values in 4diac environments. Through its generic design and the consistent use of unidirectional adapters, it makes a significant contribution to the clarity, structuring and maintainability of modern, distributed control programs.

ALR_ADD_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) **ALR_ADD_4** is a generic arithmetic block for performing an addition operation. It is classified as a standard arithmetic function according to the IEC 61131-3 standard. The block allows up to four input values, provided via unidirectional adapters of type ALR, to be added and the result passed to a corresponding output adapter.

Interface Structure

Event Inputs

No direct event inputs are available. Synchronization and event control are handled implicitly via the adapters used.

Event Outputs

No direct event outputs are available. Event forwarding is implicit via the output adapter.

Data Inputs

No direct data inputs are available. Data is transferred entirely via the adapter interfaces.

Data Outputs

No direct data outputs are available. Data is transferred entirely via the adapter interface.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALR`): First input value (addend 1) for addition.
- **IN2** (Type: `adapter::types::unidirectional::ALR`): Second input value (addend 2) for addition.
- **IN3** (Type: `adapter::types::unidirectional::ALR`): Third input value (addend 3) for addition.
- **IN4** (Type: `adapter::types::unidirectional::ALR`): Fourth input value (addend 4) for addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALR`): The calculated result of the addition (sum).

Functionality

The function block `ALR_ADD_4` functions internally as a generic adder (`GEN_ALR_ADD`). As soon as the values at the input adapters `IN1` to `IN4` change, the function block calculates the sum of the incoming values:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.50}$$

The result is immediately passed to the output plug `OUT`. Since this is a generic implementation, the specific data type depends on the specification of the `ALR` adapter type used.

Technical Features

- **Generic Class:** The function block is linked to the value `'GEN_ALR_ADD'` via the attribute `GenericClassName`, giving it high flexibility in processing various compatible data types within the adapter structures.
- **Adapter-Based Coupling:** By exclusively using adapters, the number of connection lines in the 4diac IDE is drastically reduced, resulting in clearer application diagrams.

State Overview

Since this function block performs a purely combinatorial, mathematical operation, it does not have an internal state machine (Execution Control Chart, ECC). The output directly follows changes in the inputs.

Application Scenarios

- **Measurement Aggregation:** Summing up to four energy, flow, or power values transmitted via standardized unidirectional adapters.
- **Signal Merging:** Structured bundling of sensor signals in complex control architectures prior to further data processing.

Comparison with Similar Components

- **Standard Add-in Component:** A classic IEC 61131-3 `ADD` component uses direct data inputs (e.g., `REAL`, `INT`) and often requires explicit event connections (`REQ` / `CNF`). `ALR_ADD_4` encapsulates this data and these events in adapters, simplifying project design.
- **Two-Way Adder (e.g., ALR_ADD_2):** Offers only two inputs. The `ALR_ADD_4` block eliminates the need to cascade multiple smaller adder blocks when summing three or four values.

Change Detection

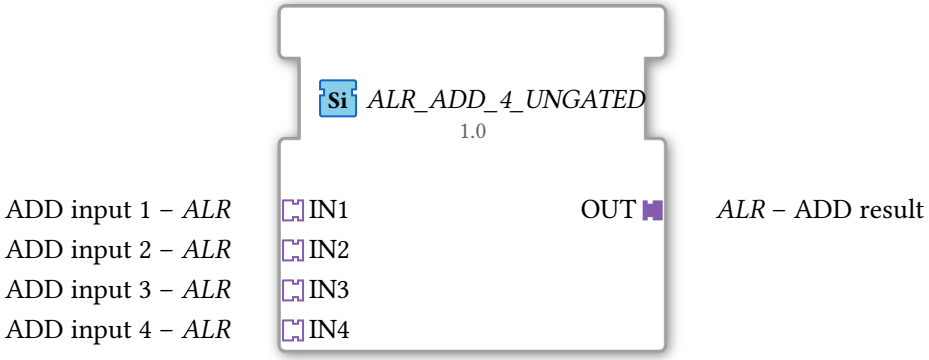
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALR_ADD_4` block is an efficient and clean solution for arithmetic addition in component- and adapter-based system architectures within 4diac. It promotes reusability and ensures a streamlined system design.

ALR_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_ADD_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) **ALR_ADD_4_UNGATED** is a generic arithmetic block for performing an addition operation. It is classified as a standard arithmetic function according to the IEC 61131-3 standard. The block allows up to four input values, provided via unidirectional adapters of type `ALR`, to be added and the result passed to a corresponding output adapter.

Interface Structure

Event Inputs

No direct event inputs are available. Synchronization and event control are handled implicitly via the adapters used.

Event Outputs

No direct event outputs are available. Event forwarding is implicit via the output adapter.

Data Inputs

No direct data inputs are available. Data is transferred entirely via the adapter interfaces.

Data Outputs

No direct data outputs are available. Data is transferred entirely via the adapter interface.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALR`): First input value (addend 1) for addition.
- **IN2** (Type: `adapter::types::unidirectional::ALR`): Second input value (addend 2) for addition.

- **IN3** (Type: `adapter::types::unidirectional::ALR`): Third input value (addend 3) for addition.
- **IN4** (Type: `adapter::types::unidirectional::ALR`): Fourth input value (addend 4) for addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALR`): The calculated result of the addition (sum).

Functionality

The function block `ALR_ADD_4_UNGATED` functions internally as a generic adder (`GEN_ALR_ADD`). As soon as the values at the input adapters `IN1` to `IN4` change, the function block calculates the sum of the incoming values:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.51}$$

The result is immediately passed to the output plug `OUT`. Since this is a generic implementation, the specific data type depends on the specification of the `ALR` adapter type used.

Technical Features

- **Generic Class:** The function block is linked to the value `'GEN_ALR_ADD'` via the attribute `GenericClassName`, giving it high flexibility in processing various compatible data types within the adapter structures.
- **Adapter-Based Coupling:** By exclusively using adapters, the number of connection lines in the 4diac IDE is drastically reduced, resulting in clearer application diagrams.

State Overview

Since this function block performs a purely combinatorial, mathematical operation, it does not have an internal state machine (Execution Control Chart, ECC). The output directly follows changes in the inputs.

Application Scenarios

- **Measurement Aggregation:** Summing up to four energy, flow, or power values transmitted via standardized unidirectional adapters.
- **Signal Merging:** Structured bundling of sensor signals in complex control architectures prior to further data processing.

Comparison with Similar Components

- **Standard Add-in Component:** A classic IEC 61131-3 `ADD` component uses direct data inputs (e.g., `REAL`, `INT`) and often requires explicit event connections (`REQ` / `CNF`). `ALR_ADD_4_UNGATED` encapsulates this data and these events in adapters, simplifying project design.
- **Two-Way Adder (e.g., ALR_ADD_2):** Offers only two inputs. The `ALR_ADD_4_UNGATED` block eliminates the need to cascade multiple smaller adder blocks when summing three or four values.

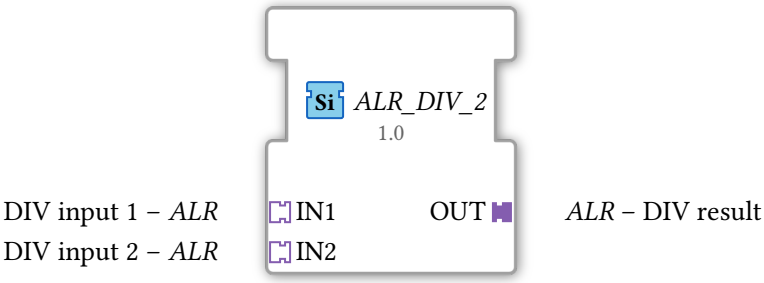
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALR_ADD_4_UNGATED` block is an efficient and clean solution for arithmetic addition in component- and adapter-based system architectures within 4diac. It promotes reusability and ensures a streamlined system design.

ALR_DIV_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_DIV_2` is a generic function block (Generic FB) for performing arithmetic divisions. It is defined in the package `adapter::iec61131::arithmetic` and is based on the generic class `GEN_ALR_DIV`.

Unlike classic arithmetic function blocks that work with direct data and event inputs, this block uses adapters of type `ALR` (unidirectional). This enables significantly simplified and structured wiring within 4diac applications, as signals and their associated control flows are bundled in adapters.

Interface Structure

Event Inputs

This block has no direct event inputs. Control and triggering are implicit via the connected adapters.

Event Outputs

This block has no direct event outputs.

Data Inputs

This block has no direct data inputs.

Data Outputs

This block has no direct data outputs.

Adapters

Type	Name	Adapter Type	Description
Socket (Input)	`IN1`	`adapter::types::unidirectional::ALR`	First Operand of Division (Dividend)
Socket (Input)	`IN2`	`adapter::types::unidirectional::ALR`	Second operand of the division (divisor)
Plug (output)	`OUT`	`adapter::types::unidirectional::ALR`	Result of the division (quotient)

Functionality

The function block `ALR_DIV_2` performs a mathematical division of the values provided via the adapters. The calculation follows the formula:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.52}$$

Since this is a generic function block, the exact data type of the mathematical operation is determined at runtime or during instantiation by the specific data type used by the underlying `ALR` adapter. Data transmission is unidirectional, from the sockets (`IN1` and `IN2`) to the plug (`OUT`).

Technical Features

- **Generic Block:** The property `GenericClassName = "GEN_ALR_DIV"` allows the block to be used flexibly for various data types, provided they are supported by the adapter type.
- **Adapter Coupling:** Using adapters instead of individual pins minimizes the graphical wiring effort in the 4diac IDE and improves clarity in the application editor.
- **Division by Zero:** During implementation and use, ensure that the value on the adapter `IN2` is not zero to avoid runtime errors or undefined system behavior.

State Overview

This module does not have a complex internal state diagram (ECC). It operates as a purely functional processing module that calculates the result when values change at the input adapters (`IN1` or `IN2`) and forwards it via the output adapter (`OUT`).

Application Scenarios

- **Signal Scaling:** Division of analog measured values by constant factors for unit conversion within an adapter-based signal processing chain.
- **Averaging and Ratio Control:** Calculation of ratios (e.g., air-fuel ratio) in control systems where signal paths are implemented modularly via adapters.
- **Standardization:** Use in complex control architectures to reduce the number of interconnects.

Comparison with Similar Function Blocks

Compared to the standard IEC 61131-3 compact component `DIV`, the `ALR_DIV_2` has no direct data pins (unlike `IN1`, `IN2`, and `ANY_NUM`). While the standard `DIV` component is optimized for simple, point-to-point calculations, the `ALR_DIV_2` is primarily suited for service-oriented architectures and applications that consistently rely on loose coupling via adapter structures.

Change Detection

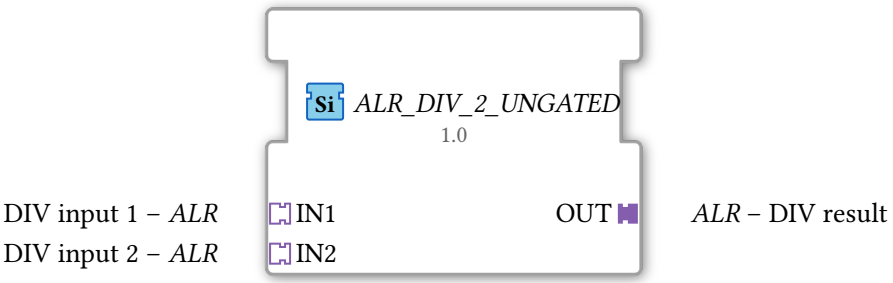
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALR_DIV_2` is a specialized yet flexible division component for the 4diac IDE. It is ideally suited for demanding projects that value clean encapsulation of data flows using the unidirectional `ALR` adapter.

ALR_DIV_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_DIV_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_DIV_2_UNGATED` is a generic function block (Generic FB) for performing arithmetic divisions. It is defined in the package `adapter::iec61131::arithmetic` and is based on the generic class `GEN_ALR_DIV`.

Unlike classic arithmetic function blocks that work with direct data and event inputs, this block uses adapters of type `ALR` (unidirectional). This enables significantly simplified and structured wiring within 4diac applications, as signals and their associated control flows are bundled in adapters.

Interface Structure

Event Inputs

This block has no direct event inputs. Control and triggering are implicit via the connected adapters.

Event Outputs

This block has no direct event outputs.

Data Inputs

This block has no direct data inputs.

Data Outputs

This block has no direct data outputs.

Adapters

Type	Name	Adapter Type	Description
Socket (Input)	`IN1`	`adapter::types::unidirectional::ALR`	First Operand of Division (Dividend)
Socket (Input)	`IN2`	`adapter::types::unidirectional::ALR`	Second operand of the division (divisor)
Plug (output)	`OUT`	`adapter::types::unidirectional::ALR`	Result of the division (quotient)

Functionality

The function block `ALR_DIV_2_UNGATED` performs a mathematical division of the values provided via the adapters. The calculation follows the formula:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.53}$$

Since this is a generic function block, the exact data type of the mathematical operation is determined at runtime or during instantiation by the specific data type used by the underlying `ALR` adapter. Data transmission is unidirectional, from the sockets (`IN1` and `IN2`) to the plug (`OUT`).

Technical Features

- **Generic Block:** The property `GenericClassName = "GEN_ALR_DIV"` allows the block to be used flexibly for various data types, provided they are supported by the adapter type.
- **Adapter Coupling:** Using adapters instead of individual pins minimizes the graphical wiring effort in the 4diac IDE and improves clarity in the application editor.
- **Division by Zero:** During implementation and use, ensure that the value on the adapter `IN2` is not zero to avoid runtime errors or undefined system behavior.

State Overview

This module does not have a complex internal state diagram (ECC). It operates as a purely functional processing module that calculates the result when values change at the input adapters (`IN1` or `IN2`) and forwards it via the output adapter (`OUT`).

Application Scenarios

- **Signal Scaling:** Division of analog measured values by constant factors for unit conversion within an adapter-based signal processing chain.

- **Averaging and Ratio Control:** Calculation of ratios (e.g., air-fuel ratio) in control systems where signal paths are implemented modularly via adapters.
- **Standardization:** Use in complex control architectures to reduce the number of interconnects.

Comparison with Similar Function Blocks

Compared to the standard IEC 61131-3 compact component `DIV`, the `ALR_DIV_2_UNGATED` has no direct data pins (unlike `IN1`, `IN2`, and `ANY_NUM`). While the standard `DIV` component is optimized for simple, point-to-point calculations, the `ALR_DIV_2_UNGATED` is primarily suited for service-oriented architectures and applications that consistently rely on loose coupling via adapter structures.

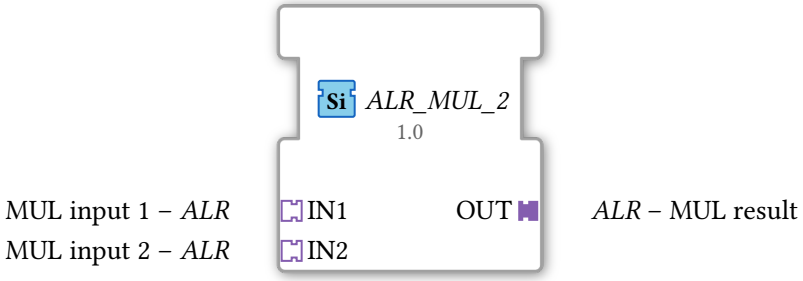
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALR_DIV_2_UNGATED` is a specialized yet flexible division component for the 4diac IDE. It is ideally suited for demanding projects that value clean encapsulation of data flows using the unidirectional `ALR` adapter.

ALR_MUL_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_MUL_2` is a generic arithmetic function block designed for multiplying two values. It is based on the IEC 61499 architecture and uses adapter interfaces to encapsulate and transmit data and events. This enables clean structuring and reduces wiring effort in 4diac IDE applications, as data and associated control flows are bundled in common connections (adapters).

Interface Structure

Event Inputs

This function block does not have direct event inputs. Control flow is managed implicitly via the adapters used.

Event Outputs

This function block does not have direct event outputs. Events are forwarded implicitly via the output adapter.

Data Inputs

This function block has no direct data inputs. The input data is provided via the adapter interfaces.

Data Outputs

This function block has no direct data outputs. The calculation result is provided via the output adapter.

Adapters

The function block communicates exclusively via adapter connections of type `ALR` (unidirectional):

- **IN1 (Socket)**:
- **Type**: `adapter::types::unidirectional::ALR`
- **Description**: First input (multiplicand 1) for multiplication.
- **IN2 (Socket)**:

Type: `adapter::types::unidirectional::ALR`
Description: Second input (multiplicand 2) for multiplication.
OUT (Plug):
Type: `adapter::types::unidirectional::ALR`
Description: Output for the calculated multiplication result.

Functionality

The function block `ALR_MUL_2` performs a mathematical multiplication according to the following principle:

$$OUT = IN1 \times IN2 \tag{2.54}$$

As soon as a new event is signaled at the input adapters `IN1` and/or `IN2`, indicating a data update, the function block receives the corresponding values, multiplies them, and outputs the result along with a corresponding update event via the output adapter `OUT`.

Since this is a generic function block (`GEN_ALR_MUL`), the underlying calculation adapts to the data type defined in the adapter.

Technical Features

- **Generic Implementation**: The function block is classified as `GEN_ALR_MUL`. This allows for flexible use with various numeric data types defined by the adapter profile `ALR`.
- **Adapter Encapsulation**: By using unidirectional adapters, the control flow (events) is directly linked to the data flow (values). This prevents synchronization errors between data and events.
- **Compiler Package**: The function block is declared in the package `adapter::iec61131::arithmetic` and uses the core library `eclipse4diac::core::GenericClassName`.

–

State Overview

The function block behaves like a classic, stateless (or purely functional) function block. There are no internal states that are stored beyond a single calculation cycle:

1. **Wait State**: The function block waits for an event at `IN1` or `IN2`.
2. **Calculation**: Upon arrival of an event, the values from the adapters are read and multiplied.
3. **Output**: The result is applied to `OUT`, and the output event is triggered. The function block immediately returns to the wait state.

Application Scenarios

- **Signal Processing in Distributed Systems**: When sensor values are already available as adapter structures and need to be scaled or calculated together (e.g., calculating power from current and voltage).
- **Modular Control Design**: Reduction of line crossings in the function block diagram of 4diac applications by using adapters instead of separate event and data lines. * **Cascaded Calculations**: Simple chaining of multiple mathematical operations without additional overhead for event wiring.

Comparison with Similar Function Blocks

- **Standard `MUL` Function Block (IEC 61131-3)**: The classic `MUL` function block uses separate event inputs (e.g., `REQ`) and data inputs (e.g., `IN1`, `IN2` as REAL/INT). `ALR_MUL_2` simplifies this by combining these channels in adapters.
- **ALR_ADD_2 / ALR_SUB_2**: These function blocks operate on the same adapter principle, but perform additions or subtractions instead of multiplications.

Change Detection

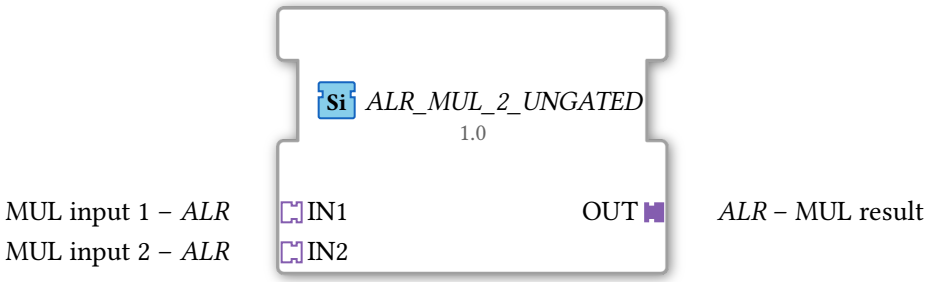
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

ALR_MUL_2 is a specialized, highly efficient function block for modern IEC 61499 architectures. Through its consistent use of adapters, it promotes clean, object-oriented software design in industrial automation and is ideally suited for complex, distributed control tasks.

ALR_MUL_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of ALR_MUL_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block ALR_MUL_2_UNGATED is a generic arithmetic function block designed for multiplying two values. It is based on the IEC 61499 architecture and uses adapter interfaces to encapsulate and transmit data and events. This enables clean structuring and reduces wiring effort in 4diac IDE applications, as data and associated control flows are bundled in common connections (adapters).

Interface Structure

Event Inputs

This function block does not have direct event inputs. Control flow is managed implicitly via the adapters used.

Event Outputs

This function block does not have direct event outputs. Events are forwarded implicitly via the output adapter.

Data Inputs

This function block has no direct data inputs. The input data is provided via the adapter interfaces.

Data Outputs

This function block has no direct data outputs. The calculation result is provided via the output adapter.

Adapters

The function block communicates exclusively via adapter connections of type ALR (unidirectional):

- **IN1 (Socket)**:
- **Type**: adapter::types::unidirectional::ALR
- **Description**: First input (multiplicand 1) for multiplication.
- **IN2 (Socket)**:

Type: adapter::types::unidirectional::ALR
Description: Second input (multiplicand 2) for multiplication.
OUT (Plug):
Type: adapter::types::unidirectional::ALR
Description: Output for the calculated multiplication result.

Functionality

The function block `ALR_MUL_2_UNGATED` performs a mathematical multiplication according to the following principle:

$$OUT = IN1 \times IN2 \tag{2.55}$$

As soon as a new event is signaled at the input adapters `IN1` and/or `IN2`, indicating a data update, the function block receives the corresponding values, multiplies them, and outputs the result along with a corresponding update event via the output adapter `OUT`.

Since this is a generic function block (`GEN_ALR_MUL`), the underlying calculation adapts to the data type defined in the adapter.

Technical Features

- **Generic Implementation**: The function block is classified as `GEN_ALR_MUL`. This allows for flexible use with various numeric data types defined by the adapter profile `ALR`.
- **Adapter Encapsulation**: By using unidirectional adapters, the control flow (events) is directly linked to the data flow (values). This prevents synchronization errors between data and events.
- **Compiler Package**: The function block is declared in the package `adapter::iec61131::arithmetic` and uses the core library `eclipse4diac::core::GenericClassName`.

–

State Overview

The function block behaves like a classic, stateless (or purely functional) function block. There are no internal states that are stored beyond a single calculation cycle:

1. **Wait State**: The function block waits for an event at `IN1` or `IN2`.
2. **Calculation**: Upon arrival of an event, the values from the adapters are read and multiplied.
3. **Output**: The result is applied to `OUT`, and the output event is triggered. The function block immediately returns to the wait state.

Application Scenarios

- **Signal Processing in Distributed Systems**: When sensor values are already available as adapter structures and need to be scaled or calculated together (e.g., calculating power from current and voltage).
- **Modular Control Design**: Reduction of line crossings in the function block diagram of 4diac applications by using adapters instead of separate event and data lines. * **Cascaded Calculations**: Simple chaining of multiple mathematical operations without additional overhead for event wiring.

Comparison with Similar Function Blocks

- **Standard `MUL` Function Block (IEC 61131-3)**: The classic `MUL` function block uses separate event inputs (e.g., `REQ`) and data inputs (e.g., `IN1`, `IN2` as REAL/INT). `ALR_MUL_2_UNGATED` simplifies this by combining these channels in adapters.
- **ALR_ADD_2 / ALR_SUB_2**: These function blocks operate on the same adapter principle, but perform additions or subtractions instead of multiplications.

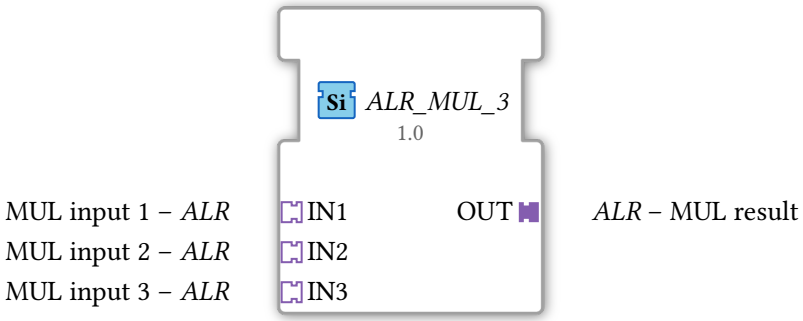
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

`ALR_MUL_2_UNGATED` is a specialized, highly efficient function block for modern IEC 61499 architectures. Through its consistent use of adapters, it promotes clean, object-oriented software design in industrial automation and is ideally suited for complex, distributed control tasks.

ALR_MUL_3



(Symbolic representation of the function block)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_MUL_3` is a generic block from the `adapter::iec61131::arithmetic` library, designed for the arithmetic multiplication of three input values. Instead of classic, discrete data inputs, this block uses unidirectional adapters of type `ALR` to encapsulate and transmit data and control events. This enables structured, modular, and clear wiring within the 4diac IDE.

Interface Structure

Event Inputs

No direct event inputs are available. Event-based control is handled entirely via the integrated adapters.

Event Outputs

No direct event outputs are available. Event-based control is handled entirely via the integrated adapters.

Data Inputs

No direct data inputs are available. The data values are encapsulated in the input adapters.

Data Outputs

No direct data outputs are available. The result is encapsulated in the output adapter.

Adapters

Sockets (Input Interfaces)

- **IN1** (Type: `adapter::types::unidirectional::ALR`): The first factor for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::ALR`): The second factor for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::ALR`): The third factor for the multiplication.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::ALR`): The calculated product of the three input values.

Functionality

The function block `ALR_MUL_3` performs the mathematical multiplication of three input variables. As soon as new values or trigger events are present at the sockets, the function block executes the calculation according to the following formula:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.56}$$

The result of the calculation and the corresponding update event are output via the plug `OUT`.

Since this is a generic function block (Generic class: `GEN_ALR_MUL`), the actual data type (e.g., `REAL`, `LREAL`, `INT`) depends on the definition of the `ALR` adapter used.

Technical Features

- **Generic Design:** By assigning the attribute `GenericClassName = 'GEN_ALR_MUL'`, the function block can be used flexibly for various data types, provided the adapters used support them.
- **Adapter-Based Coupling:** Using adapters instead of loose event/data connections drastically reduces the wiring effort (routing) within the 4diac application and improves clarity.
- – **Unidirectionality:** The `ALR` adapters used are defined as unidirectional, ensuring a clear and feedback-free data flow from the sources (sockets) to the sink (plug).

State Overview

The component does not have a complex internal state machine (ECC). It operates as a purely combinatorial/mathematical component that reacts directly to events at the input adapters and immediately forwards the result to the output.

Application Scenarios

- **Sensor Scaling and Correction:** Multiplication of a raw value (IN1) by a calibration factor (IN2) and an application-specific weighting factor (IN3).
- **Three-Dimensional Calculations:** Calculation of volumes or throughputs where three physical variables need to be multiplied together.
- **Cascaded Gainers:** Calculation of combined gains in control loops.

Comparison with Similar Function Blocks

- **Standard MUL Function Block (IEC 61131-3):** Classic multipliers work with direct elementary data types (e.g., `REAL`). `ALR_MUL_3`, on the other hand, bundles data and events in adapters, which improves modularity and reusability in distributed systems according to IEC 61499.
- **ALR_MUL_2 (Dual Multiplier):** For multiplying only two values, a corresponding dual function block is preferred. `ALR_MUL_3` eliminates the need to cascade two separate function blocks when multiplying three factors.

–

Change Detection

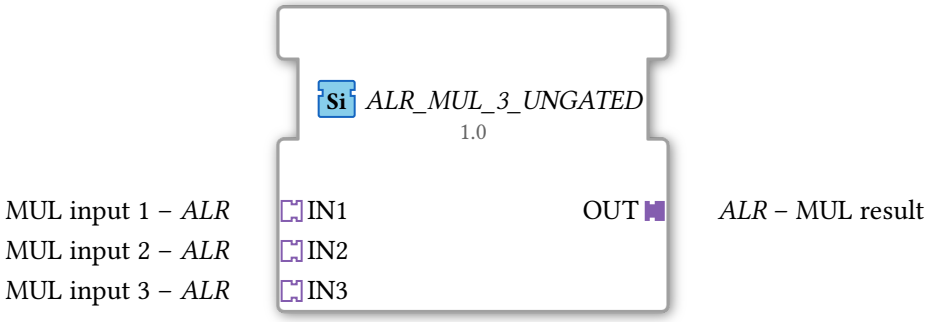
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALR_MUL_3` function block is an efficient and clean solution for triple multiplication tasks in IEC 61499 applications. By consistently using unidirectional adapters, he promotes clean software design and ensures well-structured data flows in the 4diac IDE.

ALR_MUL_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_MUL_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(Symbolic representation of the function block)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_MUL_3_UNGATED` is a generic block from the `adapter::iec61131::arithmetic` library, designed for the arithmetic multiplication of three input values. Instead of classic, discrete data inputs, this block uses unidirectional adapters of type `ALR` to encapsulate and transmit data and control events. This enables structured, modular, and clear wiring within the 4diac IDE.

Interface Structure

Event Inputs

No direct event inputs are available. Event-based control is handled entirely via the integrated adapters.

Event Outputs

No direct event outputs are available. Event-based control is handled entirely via the integrated adapters.

Data Inputs

No direct data inputs are available. The data values are encapsulated in the input adapters.

Data Outputs

No direct data outputs are available. The result is encapsulated in the output adapter.

Adapters

Sockets (Input Interfaces)

- **IN1** (Type: `adapter::types::unidirectional::ALR`): The first factor for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::ALR`): The second factor for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::ALR`): The third factor for the multiplication.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::ALR`): The calculated product of the three input values.

Functionality

The function block `ALR_MUL_3_UNGATED` performs the mathematical multiplication of three input variables. As soon as new values or trigger events are present at the sockets, the function block executes the calculation according to the following formula:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.57}$$

The result of the calculation and the corresponding update event are output via the plug `OUT`.

Since this is a generic function block (Generic class: `GEN_ALR_MUL`), the actual data type (e.g., `REAL`, `LREAL`, `INT`) depends on the definition of the `ALR` adapter used.

Technical Features

- **Generic Design:** By assigning the attribute `GenericClassName = 'GEN_ALR_MUL'`, the function block can be used flexibly for various data types, provided the adapters used support them.
- **Adapter-Based Coupling:** Using adapters instead of loose event/data connections drastically reduces the wiring effort (routing) within the 4diac application and improves clarity.
- – **Unidirectionality:** The `ALR` adapters used are defined as unidirectional, ensuring a clear and feedback-free data flow from the sources (sockets) to the sink (plug).

State Overview

The component does not have a complex internal state machine (ECC). It operates as a purely combinatorial/mathematical component that reacts directly to events at the input adapters and immediately forwards the result to the output.

Application Scenarios

- **Sensor Scaling and Correction:** Multiplication of a raw value (IN1) by a calibration factor (IN2) and an application-specific weighting factor (IN3).
- **Three-Dimensional Calculations:** Calculation of volumes or throughputs where three physical variables need to be multiplied together.
- **Cascaded Gainers:** Calculation of combined gains in control loops.

Comparison with Similar Function Blocks

- **Standard MUL Function Block (IEC 61131-3):** Classic multipliers work with direct elementary data types (e.g., `REAL`). `ALR_MUL_3_UNGATED`, on the other hand, bundles data and events in adapters, which improves modularity and reusability in distributed systems according to IEC 61499.
- **ALR_MUL_2 (Dual Multiplier):** For multiplying only two values, a corresponding dual function block is preferred. `ALR_MUL_3_UNGATED` eliminates the need to cascade two separate function blocks when multiplying three factors.

–

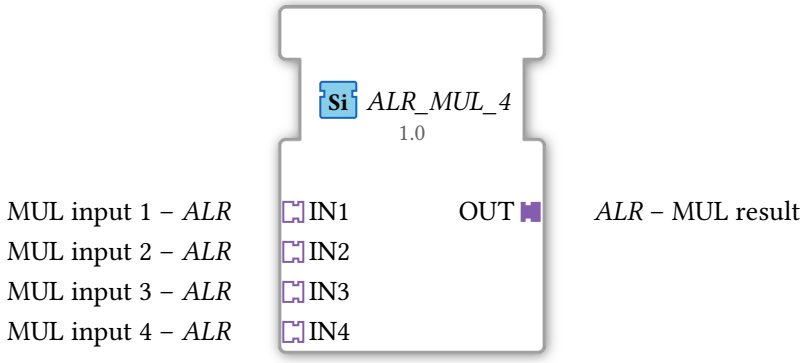
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALR_MUL_3_UNGATED` function block is an efficient and clean solution for triple multiplication tasks in IEC 61499 applications. By consistently using unidirectional adapters, he promotes clean software design and ensures well-structured data flows in the 4diac IDE.

ALR_MUL_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_MUL_4` performs an arithmetic multiplication of four input values. It is a generic function block (FB) for the 4diac IDE, based on the IEC 61499 standard. Signal transmission and processing are handled via special unidirectional analog adapters, enabling clean encapsulation of the data streams.

Interface Structure

Event Inputs

No direct event inputs are defined for this function block, as control and updates are handled via adapters.

Event Outputs

No direct event outputs are defined for this function block.

Data Inputs

No direct data inputs are defined for this function block.

Data Outputs

No direct data outputs are defined for this function block.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALR`): First input value (multiplicand 1) for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::ALR`): Second input value (multiplicand 2) for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::ALR`): Third input value (multiplicand 3) for multiplication.
- **IN4** (Type: `adapter::types::unidirectional::ALR`): Fourth input value (multiplicand 4) for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALR`): Output for the calculated product of the four input values.

Functionality

The `ALR_MUL_4` module multiplies the analog values provided by the four input adapters (`IN1` to `IN4`). The mathematical result is output via the `OUT` output adapter.

The underlying calculation formula is:

$$OUT = IN1 \times IN2 \times IN3 \times IN4 \tag{2.58}$$

Since this is a generic function block that uses the class `GEN_ALR_MUL`, the implementation is flexible with respect to the data types used in the adapter (e.g., `REAL` or `LREAL`).

Technical Features

- **Generic Function Block**: The attribute `GenericClassName` with the value `GEN_ALR_MUL` allows the function block to adapt flexibly to the specific data type implementation of the adapter used at runtime or during compilation.
- **Encapsulation via Adapter**: The use of the unidirectional adapter `ALR` (`adapter::types::unidirectional::ALR`) ensures that data values and any associated status or trigger events are bundled compactly in a single connection. This significantly reduces the wiring effort in the function block diagram.

State Overview

The function block has no internal state behavior (no state machine/ECC) and operates purely in a data flow-oriented manner. As soon as the values at the input adapters change or a corresponding update event is triggered via the adapters, the multiplication is performed and the result is provided at the output `OUT`.

Application Scenarios

- **Physical Calculations**: Calculation of more complex quantities that represent the product of several variables (e.g., power, energy calculations, or volumetric flow rates taking correction factors into account).
- **Cascaded Scaling**: Applying multiple scaling or correction factors to an analog input signal in a single step.
- **Signal Processing**: Preprocessing sensor values in control systems before the data is passed on to visualizations or actuators.

Comparison with Similar Function Blocks

- **Standard MUL Function Blocks (IEC 61131-3)**: Classic multiplication function blocks typically work directly with standard data types (e.g., `REAL`) and require separate event connections (`REQ`/`CNF`). `ALR_MUL_4` simplifies this through the use of adapters.
- **ALR_MUL_2 / ALR_MUL_3**: These function blocks are designed for multiplying only two or three values. The `ALR_MUL_4` function block eliminates the need for additional intermediate steps and block cascades when four multiplicands are required, thus improving the performance and clarity of the control program.

Change Detection

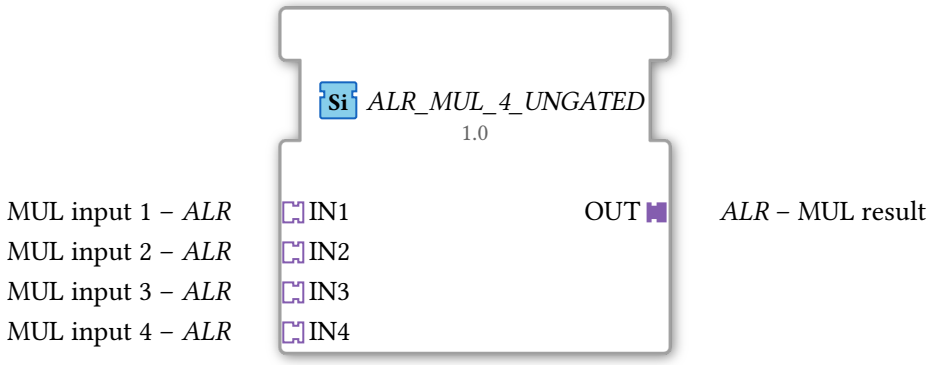
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALR_MUL_4` function block is a practical auxiliary block for mathematical signal processing in 4diac. By consistently utilizing the adapter concept, it significantly reduces the number of connecting lines in the application diagram while simultaneously offering the flexibility of a generic function block.

ALR_MUL_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_MUL_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_MUL_4_UNGATED` performs an arithmetic multiplication of four input values. It is a generic function block (FB) for the 4diac IDE, based on the IEC 61499 standard. Signal transmission and processing are handled via special unidirectional analog adapters, enabling clean encapsulation of the data streams.

Interface Structure

Event Inputs
No direct event inputs are defined for this function block, as control and updates are handled via adapters.

Event Outputs
No direct event outputs are defined for this function block.

Data Inputs
No direct data inputs are defined for this function block.

Data Outputs
No direct data outputs are defined for this function block.

Adapters

Sockets (Input Adapters)
- **IN1** (Type: `adapter::types::unidirectional::ALR`): First input value (multiplicand 1) for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::ALR`): Second input value (multiplicand 2) for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::ALR`): Third input value (multiplicand 3) for multiplication.
- **IN4** (Type: `adapter::types::unidirectional::ALR`): Fourth input value (multiplicand 4) for multiplication.

Plugs (Output Adapters)
- **OUT** (Type: `adapter::types::unidirectional::ALR`): Output for the calculated product of the four input values.

Functionality
The `ALR_MUL_4_UNGATED` module multiplies the analog values provided by the four input adapters (`IN1` to `IN4`). The mathematical result is output via the `OUT` output adapter.

The underlying calculation formula is:

$$OUT = IN1 \times IN2 \times IN3 \times IN4 \tag{2.59}$$

Since this is a generic function block that uses the class `GEN_ALR_MUL`, the implementation is flexible with respect to the data types used in the adapter (e.g., `REAL` or `LREAL`).

Technical Features
- **Generic Function Block**: The attribute `GenericClassName` with the value `GEN_ALR_MUL` allows the function block to adapt flexibly to the specific data type implementation of the adapter used at runtime or during compilation.
- **Encapsulation via Adapter**: The use of the unidirectional adapter `ALR` (`adapter::types::unidirectional::ALR`) ensures that data values and any associated status or trigger events are bundled compactly in a single connection. This significantly reduces the wiring effort in the function block diagram.

State Overview
The function block has no internal state behavior (no state machine/ECC) and operates purely in a data flow-oriented manner. As soon as the values at the input adapters change or a corresponding update event is triggered via the adapters, the multiplication is performed and the result is provided at the output `OUT`.

Application Scenarios
- **Physical Calculations**: Calculation of more complex quantities that represent the product of several variables (e.g., power, energy calculations, or volumetric flow rates taking correction factors into account).
- **Cascaded Scaling**: Applying multiple scaling or correction factors to an analog input signal in a single step.

- **Signal Processing**: Preprocessing sensor values in control systems before the data is passed on to visualizations or actuators.

Comparison with Similar Function Blocks

- **Standard MUL Function Blocks (IEC 61131-3)**: Classic multiplication function blocks typically work directly with standard data types (e.g., `REAL`) and require separate event connections (`REQ`/`CNF`). `ALR_MUL_4_UNGATED` simplifies this through the use of adapters.
- **ALR_MUL_2 / ALR_MUL_3**: These function blocks are designed for multiplying only two or three values. The `ALR_MUL_4_UNGATED` function block eliminates the need for additional intermediate steps and block cascades when four multiplicands are required, thus improving the performance and clarity of the control program.

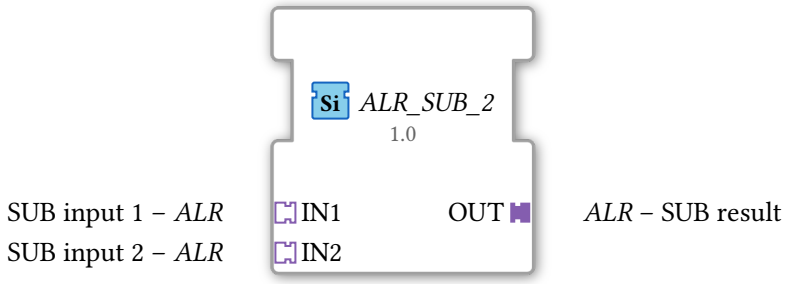
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALR_MUL_4_UNGATED` function block is a practical auxiliary block for mathematical signal processing in 4diac. By consistently utilizing the adapter concept, it significantly reduces the number of connecting lines in the application diagram while simultaneously offering the flexibility of a generic function block.

ALR_SUB_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_SUB_2` is a generic block for performing mathematical subtractions within an IEC 61499 development environment (such as 4diac IDE). Instead of using direct data and event pins, this block is based on an adapter-driven architecture. This enables structured and clean signal encapsulation and simplifies wiring in complex control applications.

Interface Structure

Event Inputs

This block does not have direct event inputs. Control and triggering are handled via the connected adapters.

Event Outputs

This block does not have direct event outputs. Event forwarding is handled via the output adapter.

Data Inputs

There are no direct data inputs.

Data Outputs

There are no direct data outputs.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALR`):

The first input value (minuend) of the subtraction. This adapter receives both the value and the triggering event.

- **IN2** (Type: `adapter::types::unidirectional::ALR`):

The second input value (subtrahend) to be subtracted from the first value.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALR`):

The result of the subtraction (difference). This adapter outputs the calculated result along with the corresponding output event.

Functionality

The module's internal logic calculates the difference between the two values provided via the sockets. The mathematical formula is:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.60}$$

As soon as the values at the input adapters change or a corresponding trigger event is received at `IN1` or `IN2`, the module performs the calculation and updates the value at the output plug `OUT`, followed by an output event.

Technical Features

- **Generic Behavior**: The module is defined as a generic type (`GEN_ALR_SUB`). It is therefore not rigidly bound to a single data type (such as only `REAL` or only `INT`), but adapts flexibly to the data types defined in the adapter `ALR`.
- **Unidirectional Adapters**: By using unidirectional adapters (`unidirectional::ALR`), the data flow is clearly directed in one direction, which increases system stability and minimizes unnecessary communication overhead.

State Overview

Since `ALR_SUB_2` is a purely mathematical, data-flow-oriented function block, it does not have an internal state machine (ECC - Execution Control Chart). Execution is purely event- and data-driven.

Application Scenarios

- **Control Engineering (Calculation of Control Deviation)**:

Determining the difference between a setpoint (reference input at `IN1`) and an actual value (controlled variable at `IN2`) to provide the control deviation at output `OUT` for a downstream PID controller.

- **Difference Measurements**:

Calculation of physical difference values, for example, to determine the pressure drop across a filter (pressure before the filter at `IN1` minus pressure after the filter at `IN2`).

- **Offset Correction**:

Subtracting a constant or variable correction value from a raw analog signal.

Comparison with Similar Components

Compared to classic IEC 61131-3 subtraction blocks (such as `SUB`), which use standard data types and direct pins, the `ALR_SUB_2` offers significant advantages in program code clarity thanks to its encapsulation in adapters. Complex signal paths (data + event) are bundled in a single adapter connection.

Compared to bidirectional calculation blocks, the `ALR_SUB_2` features deterministic, non-reactive signal processing.

Change Detection

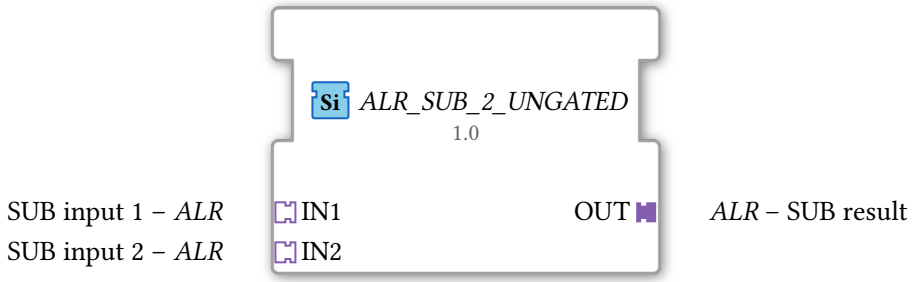
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `ALR_SUB_2` is an efficient and flexible standard component for arithmetic calculations in modern, service-oriented control systems. By utilizing adapter technology, it supports a modular and maintainable software design according to the IEC 61499 standard.

ALR_SUB_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_SUB_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ALR_SUB_2_UNGATED` is a generic block for performing mathematical subtractions within an IEC 61499 development environment (such as 4diac IDE). Instead of using direct data and event pins, this block is based on an adapter-driven architecture. This enables structured and clean signal encapsulation and simplifies wiring in complex control applications.

Interface Structure

Event Inputs

This block does not have direct event inputs. Control and triggering are handled via the connected adapters.

Event Outputs

This block does not have direct event outputs. Event forwarding is handled via the output adapter.

Data Inputs

There are no direct data inputs.

Data Outputs

There are no direct data outputs.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::ALR`):

The first input value (minuend) of the subtraction. This adapter receives both the value and the triggering event.

- **IN2** (Type: `adapter::types::unidirectional::ALR`):

The second input value (subtrahend) to be subtracted from the first value.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::ALR`):

The result of the subtraction (difference). This adapter outputs the calculated result along with the corresponding output event.

Functionality

The module's internal logic calculates the difference between the two values provided via the sockets. The mathematical formula is:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.61}$$

As soon as the values at the input adapters change or a corresponding trigger event is received at `IN1` or `IN2`, the module performs the calculation and updates the value at the output plug `OUT`, followed by an output event.

Technical Features

- **Generic Behavior**: The module is defined as a generic type (`GEN_ALR_SUB`). It is therefore not rigidly bound to a single data type (such as only `REAL` or only `INT`), but adapts flexibly to the data types defined in the adapter `ALR`.
- **Unidirectional Adapters**: By using unidirectional adapters (`unidirectional::ALR`), the data flow is clearly directed in one direction, which increases system stability and minimizes unnecessary communication overhead.

State Overview

Since `ALR_SUB_2_UNGATED` is a purely mathematical, data-flow-oriented function block, it does not have an internal state machine (ECC - Execution Control Chart). Execution is purely event- and data-driven.

Application Scenarios

- **Control Engineering (Calculation of Control Deviation)**:

Determining the difference between a setpoint (reference input at `IN1`) and an actual value (controlled variable at `IN2`) to provide the control deviation at output `OUT` for a downstream PID controller.

- **Difference Measurements**:

Calculation of physical difference values, for example, to determine the pressure drop across a filter (pressure before the filter at `IN1` minus pressure after the filter at `IN2`).

- **Offset Correction**:

Subtracting a constant or variable correction value from a raw analog signal.

Comparison with Similar Components

Compared to classic IEC 61131-3 subtraction blocks (such as `SUB`), which use standard data types and direct pins, the `ALR_SUB_2_UNGATED` offers significant advantages in program code clarity thanks to its encapsulation in adapters. Complex signal paths (data + event) are bundled in a single adapter connection.

Compared to bidirectional calculation blocks, the `ALR_SUB_2_UNGATED` features deterministic, non-reactive signal processing.

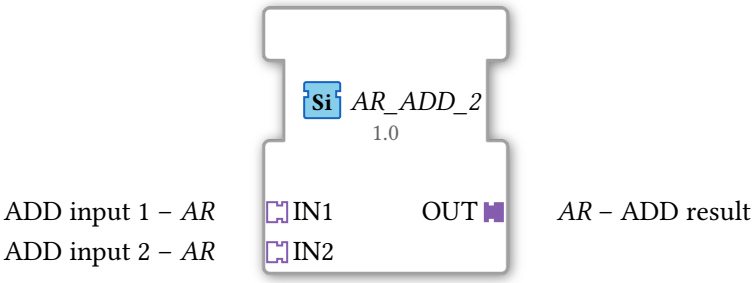
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `ALR_SUB_2_UNGATED` is an efficient and flexible standard component for arithmetic calculations in modern, service-oriented control systems. By utilizing adapter technology, it supports a modular and maintainable software design according to the IEC 61499 standard.

AR_ADD_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_ADD_2** is used to calculate the arithmetic addition of two values. It is designed as a generic function block and implements the basic arithmetic operation via adapter interfaces. The block is platform-independent and complies with the IEC 61499 standard.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Name	Direction	Type	Comment
`IN1`	Socket	`adapter::types::unidirectional::AR`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::AR`	ADD input 2
`OUT`	Plug	`adapter::types::unidirectional::AR`	ADD result

The adapters are of type `unidirectional::AR` and enable type-safe connections with other function blocks that support the same adapter type.

Functionality

The function block adds the two values received via adapters `IN1` and `IN2`. The result is provided at adapter `OUT`. The precise data interpretation (data type, value range) is determined by the adapter type used; the function block itself operates generically and delegates the type information to the underlying framework.

The actual calculation takes place in the background via the configured runtime environments – the function block merely defines the interfaces and the semantics of the addition.

Technical Features

- **Generic Structure** – The function block uses a generic class name (`GEN_AR_ADD`) at runtime, which is defined by the attribute `eclipse4diac::core::GenericClassName`. This allows for later specialization for specific data types (e.g., integers or floating-point numbers).
- **Adapter-Based** – Instead of individual data inputs and outputs, all signals are routed via unidirectional adapters. This simplifies encapsulation and reuse in complex systems.
- **Package Information** – The function block is organized in the package `adapter::iec61131::arithmetic`, indicating its origin from an IEC 61131 adapter library.

- **No State Logic** – The addition is stateless; there is no internal state machine.

State Overview

The function block does not have a state machine. The calculation is purely data-driven – if the input adapters change, the result is recalculated.

Application-Specific Scenarios

- **Signal Processing** – Combining two analog or digital measured values into a sum.
- **Control Engineering** – Summing counter readings or time values.
- **Generic Library Function Blocks** – Use as a basis for application-specific, typed addition function blocks (e.g., `AR_ADD_INT`, `AR_ADD_REAL`).

Comparison with Similar Function Blocks

Compared to classic, data-input-based addition blocks (e.g., `ADD`), `AR_ADD_2` is distinguished by its adapter interfaces. While standard adders often operate with single inputs (`IN1`, `IN2`) and one output (`OUT`), adapters enable direct connection to more complex data structures and greater reusability through type safety. The block is intentionally generic, allowing for dynamic type adjustment at runtime.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AR_ADD_2` is a compact, generic function block for adding two values using adapter interfaces. It is particularly suitable for cross-system applications where type flexibility and loose coupling via adapters are required. Its simple, stateless architecture makes it robust and easy to integrate into existing automation projects.

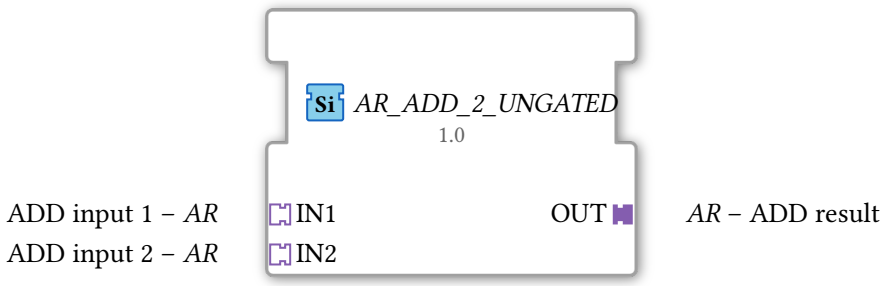
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AR_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_ADD_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_ADD_2_UNGATED** is used to calculate the arithmetic addition of two values. It is designed as a generic function block and implements the basic arithmetic operation via adapter interfaces. The block is platform-independent and complies with the IEC 61499 standard.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Name	Direction	Type	Comment
`IN1`	Socket	`adapter::types::unidirectional::AR`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::AR`	ADD input 2
`OUT`	Plug	`adapter::types::unidirectional::AR`	ADD result

The adapters are of type `unidirectional::AR` and enable type-safe connections with other function blocks that support the same adapter type.

Functionality

The function block adds the two values received via adapters `IN1` and `IN2`. The result is provided at adapter `OUT`. The precise data interpretation (data type, value range) is determined by the adapter type used; the function block itself operates generically and delegates the type information to the underlying framework.

The actual calculation takes place in the background via the configured runtime environments – the function block merely defines the interfaces and the semantics of the addition.

Technical Features

- **Generic Structure** – The function block uses a generic class name (`GEN_AR_ADD`) at runtime, which is defined by the attribute `eclipse4diac::core::GenericClassName`. This allows for later specialization for specific data types (e.g., integers or floating-point numbers).
- **Adapter-Based** – Instead of individual data inputs and outputs, all signals are routed via unidirectional adapters. This simplifies encapsulation and reuse in complex systems.
- **Package Information** – The function block is organized in the package `adapter::iec61131::arithmetic`, indicating its origin from an IEC 61131 adapter library.
- **No State Logic** – The addition is stateless; there is no internal state machine.

State Overview

The function block does not have a state machine. The calculation is purely data-driven – if the input adapters change, the result is recalculated.

Application-Specific Scenarios

- **Signal Processing** – Combining two analog or digital measured values into a sum.
- **Control Engineering** – Summing counter readings or time values.
- **Generic Library Function Blocks** – Use as a basis for application-specific, typed addition function blocks (e.g., `AR_ADD_INT`, `AR_ADD_REAL`).

Comparison with Similar Function Blocks

Compared to classic, data-input-based addition blocks (e.g., `ADD`), `AR_ADD_2_UNGATED` is distinguished by its adapter interfaces. While standard adders often operate with single inputs (`IN1`, `IN2`) and one output (`OUT`), adapters enable direct connection to more complex data structures and greater reusability through type safety. The block is intentionally generic, allowing for dynamic type adjustment at runtime.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

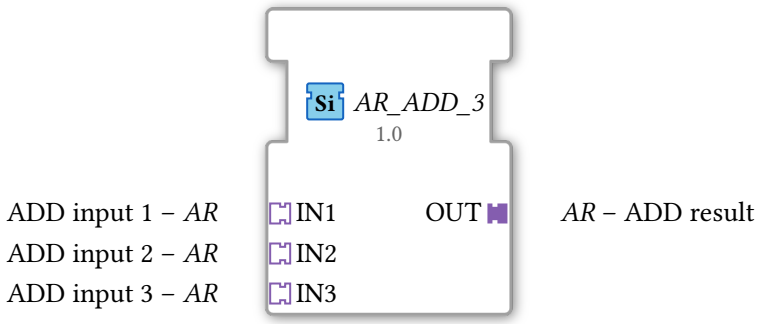
AR_ADD_2_UNGATED` is a compact, generic function block for adding two values using adapter interfaces. It is particularly suitable for cross-system applications where type flexibility and loose coupling via adapters are required. Its simple, stateless architecture makes it robust and easy to integrate into existing automation projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AR_ADD_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AR_ADD_3` is used to calculate the arithmetic addition of three values. It is a generic function block (FB) according to IEC 61499-2, which operates via adapter interfaces. The block is designed to be used in conjunction with the adapter `adapter::types::unidirectional::AR` from the library `adapter::iec61131::arithmetic`.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

No direct data inputs. Input values are provided via adapter interfaces.

Data Outputs

No direct data outputs. The result is provided via an adapter output.

Adapter

Direction	Name	Type	Comment

Sockets	IN1	`adapter::types::unidirectional::AR`	ADD input 1
Sockets	IN2	`adapter::types::unidirectional::AR`	ADD input 2
Sockets	IN3	`adapter::types::unidirectional::AR`	ADD input 3
Plugs	OUT	`adapter::types::unidirectional::AR`	ADD result

Functionality

The function block adds the three values received via the adapter sockets `IN1`, `IN2`, and `IN3` and provides the result via the adapter plug `OUT`. The exact semantics of the addition (e.g., supported data types, overflow behavior) are determined by the implementation of the adapter type used, `AR`. The function block is generic, meaning it can be parameterized for different data types (e.g., by type hash or generic class name).

Technical Features

- **Generic Block:** The actual data type handling is controlled via the attributes `eclipse4diac::core::GenericClassName` (value: `'GEN_AR_ADD'`) and `eclipse4diac::core::TypeHash`.
- **Adapter-Based Communication:** The block uses adapters (`adapter::types::unidirectional::AR`) exclusively for input and output. This enables loose coupling and reusability in different contexts.
- **License:** Released under the Eclipse Public License 2.0 (EPL-2.0).
- **Compatibility:** Developed for the 4diac IDE and the Eclipse 4diac runtime environment.

State Overview

The block does not have an explicit state diagram (ECC). It is a purely combinatorial function: The output is always the sum of the three inputs. There are no internal states or time dependencies.

Application Scenarios

- **Multiple Addition in Control Systems:** When three analog or digital values (e.g., sensor data, setpoints) need to be summed.
- **Generic Library Blocks:** Used in reusable function block libraries where the data types are defined at configuration time.
- **Adapter-Based Architectures:** In systems that rely on standardized adapter interfaces to enable flexible chaining of functions.

Comparison with Similar Blocks

AR_ADD_2 / AR_ADD_4 etc.: These blocks differ only in the number of inputs (two or four). `AR_ADD_3` is specifically optimized for adding exactly three values. The generic approach allows all these variants to be implemented, if needed, by parameterizing a common base type.

- **`AR_ADD_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

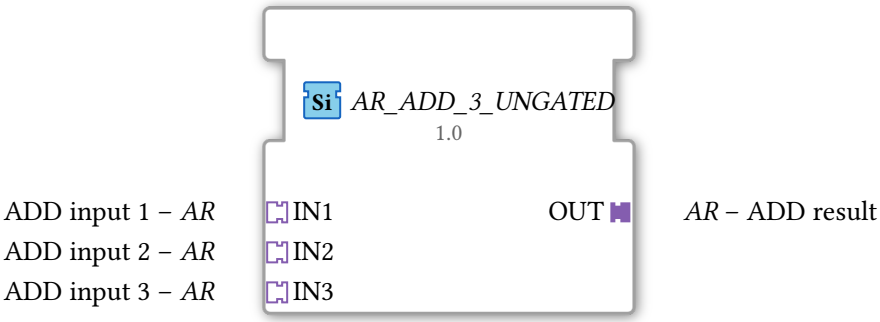
AR_ADD_3` is a clear, generic function block for adding three values using adapters. Its abstraction via generic attributes and adapters makes it well-suited for IEC 61499 applications that require flexible and reusable signal processing.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AR_ADD_3_UNGATED

i UNGATED variant: This block is the ungated version of `AR_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AR_ADD_3_UNGATED` is used to calculate the arithmetic addition of three values. It is a generic function block (FB) according to IEC 61499-2, which operates via adapter interfaces. The block is designed to be used in conjunction with the adapter `adapter::types::unidirectional::AR` from the library `adapter::iec61131::arithmetic`.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

No direct data inputs. Input values are provided via adapter interfaces.

Data Outputs

No direct data outputs. The result is provided via an adapter output.

Adapter

Direction	Name	Type	Comment
Sockets	IN1	`adapter::types::unidirectional::AR`	ADD input 1
Sockets	IN2	`adapter::types::unidirectional::AR`	ADD input 2
Sockets	IN3	`adapter::types::unidirectional::AR`	ADD input 3
Plugs	OUT	`adapter::types::unidirectional::AR`	ADD result

Functionality

The function block adds the three values received via the adapter sockets `IN1`, `IN2`, and `IN3` and provides the result via the adapter plug `OUT`. The exact semantics of the addition (e.g., supported data types, overflow behavior) are determined by the implementation of the adapter type used, `AR`. The function block is generic, meaning it can be parameterized for different data types (e.g., by type hash or generic class name).

Technical Features

- **Generic Block:** The actual data type handling is controlled via the attributes `eclipse4diac::core::GenericClassName` (value: `'GEN_AR_ADD'`) and `eclipse4diac::core::TypeHash`.
- **Adapter-Based Communication:** The block uses adapters (`adapter::types::unidirectional::AR`) exclusively for input and output. This enables loose coupling and reusability in different contexts.
- **License:** Released under the Eclipse Public License 2.0 (EPL-2.0).
- **Compatibility:** Developed for the 4diac IDE and the Eclipse 4diac runtime environment.

State Overview

The block does not have an explicit state diagram (ECC). It is a purely combinatorial function: The output is always the sum of the three inputs. There are no internal states or time dependencies.

Application Scenarios

- **Multiple Addition in Control Systems:** When three analog or digital values (e.g., sensor data, setpoints) need to be summed.
- **Generic Library Blocks:** Used in reusable function block libraries where the data types are defined at configuration time.
- **Adapter-Based Architectures:** In systems that rely on standardized adapter interfaces to enable flexible chaining of functions.

Comparison with Similar Blocks

AR_ADD_2 / AR_ADD_4 etc.: These blocks differ only in the number of inputs (two or four). `AR_ADD_3_UNGATED` is specifically optimized for adding exactly three values. The generic approach allows all these variants to be implemented, if needed, by parameterizing a common base type.

- `AR_ADD_3`: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

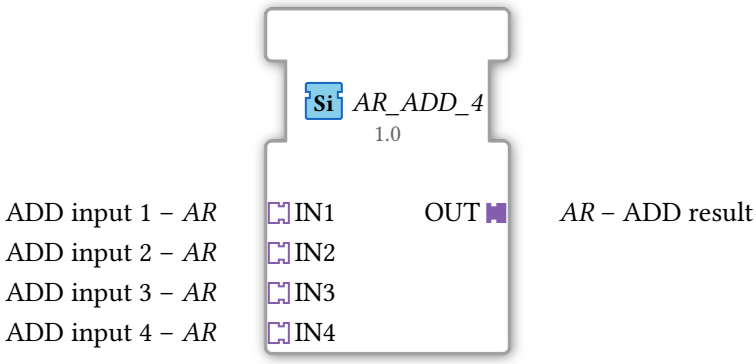
Conclusion

AR_ADD_3_UNGATED` is a clear, generic function block for adding three values using adapters. Its abstraction via generic attributes and adapters makes it well-suited for IEC 61499 applications that require flexible and reusable signal processing.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AR_ADD_4



Note: A graphical symbol for the function block is not available.

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_ADD_4** is a generic function block for the arithmetic addition of multiple values. It is designed according to the IEC 61499-2 standard and allows the flexible processing of up to four additive operands via adapter interfaces. The function block is typically used in automation systems where numerical summation of multiple signals is required.

Interface Structure

Event Inputs

- None (The function block operates without explicit event control.)

Event Outputs

- None

Data Inputs

- None (All input operands are provided via adapters.)

Data Outputs

- None (The result is output via an adapter.)

Adapter

Name	Type	Direction	Description
IN1	`adapter::types::unidirectional::AR`	Socket (Input)	First Addend of the Addition
IN2	`adapter::types::unidirectional::AR`	Socket (Input)	Second Addend
IN3	`adapter::types::unidirectional::AR`	Socket (Input)	Third Addend
IN4	`adapter::types::unidirectional::AR`	Socket (Input)	Fourth Addend
OUT	`adapter::types::unidirectional::AR`	Plug (Output)	Result of the Addition (Sum of Inputs)

The adapters are of type `unidirectional::AR`, indicating that they provide or process an arithmetic value (e.g., a numerical value) as a directed connection.

Functionality

The function block **AR_ADD_4** performs the summation of the four values applied to the input adapters and outputs the result to the output adapter **OUT**. The calculation is performed according to the general formula:

[\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4}
]

Since the function block is declared as a generic type (`eclipse4diac::core::GenericClassName = 'GEN_AR_ADD'`), the underlying data type (e.g., Integer, Real, or user-defined arithmetic types) is only determined at runtime through the specific configuration. Processing is data-driven – as soon as valid values are present at all four inputs, the sum is calculated and updated at the output.

Technical Features

- **Generic Function Block**: The function block is marked as generic (`eclipse4diac::core::GenericClassName`). This allows the specific arithmetic data type to be defined only when used in the project, enabling high reusability.
- **Adapter-Based Communication**: Instead of classic data inputs/outputs, all values are exchanged via adapters. This allows for loose coupling with other components and promotes modular structures.
- **No Event Control**: The component has no event inputs or outputs. Calculation and data transmission occur automatically as soon as all input values are available (similar to a continuous function).
- **Typical Package Structure**: The component is organized in the package `adapter::iec61131::arithmetic`, indicating an adapter implementation close to IEC 61131.

State Overview

Due to its purely data-driven and eventless operation, the component has **no internal states**. There are no sequential processes or state machines. The output is always the current sum of the four inputs.

Application Scenarios

1. **Calculating the sum of multiple process values** (e.g., adding flow signals from multiple sensors).
2. **Scaling and Summation** – In combination with scalable adapters, the function block can be used for weighted summation.
3. **Calculation of total consumption or total energy** from multiple partial measurements.
4. **Generic summation in modular automation solutions** where the number of summands is fixed, but the data type is variable.

Comparison with similar function blocks

Function block	Number of inputs	Special feature
AR_ADD_4	4	Adapter-based, generic, no events
AR_ADD_2 (hypothetical)	2	Reduced inputs, same concept
F_ADD (from IEC 61131)	2	Standard data types, event-driven (via ENABLE/ ENO)
AR_SUM (hypothetical)	variable	More flexible number, but more complex

The **AR_ADD_4** function block stands out from classic IEC 61499 arithmetic function blocks due to its pure adapter communication and generic design. It is particularly suitable for systems that already rely on adapter technology and do not require explicit event control.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The **AR_ADD_4** is a compact, generic function block for adding four values via adapter interfaces. Its simple and robust data-driven operation makes it a useful basic building block for modular automation applications. The absence of events and the generic typing enable flexible integration into heterogeneous systems. For applications with more or fewer than four addends, adapted variants or extended function blocks must be used.

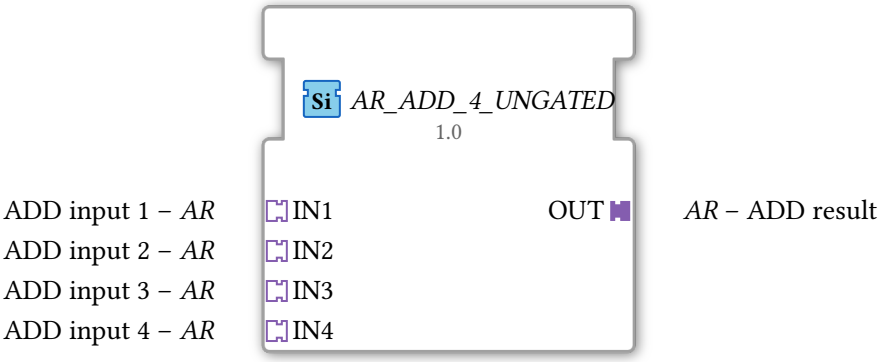
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AR_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_ADD_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



Note: A graphical symbol for the function block is not available.

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_ADD_4_UNGATED** is a generic function block for the arithmetic addition of multiple values. It is designed according to the IEC 61499-2 standard and allows the flexible processing of up to four additive operands via adapter interfaces. The function block is typically used in automation systems where numerical summation of multiple signals is required.

Interface Structure

Event Inputs

- None (The function block operates without explicit event control.)

Event Outputs

- None

Data Inputs

- None (All input operands are provided via adapters.)

Data Outputs

- None (The result is output via an adapter.)

Adapter

Name	Type	Direction	Description
IN1	`adapter::types::unidirectional::AR`	Socket (Input)	First Addend of the Addition
IN2	`adapter::types::unidirectional::AR`	Socket (Input)	Second Addend
IN3	`adapter::types::unidirectional::AR`	Socket (Input)	Third Addend
IN4	`adapter::types::unidirectional::AR`	Socket (Input)	Fourth Addend
OUT	`adapter::types::unidirectional::AR`	Plug (Output)	Result of the Addition (Sum of Inputs)

The adapters are of type `unidirectional::AR`, indicating that they provide or process an arithmetic value (e.g., a numerical value) as a directed connection.

Functionality

The function block **AR_ADD_4_UNGATED** performs the summation of the four values applied to the input adapters and outputs the result to the output adapter **OUT**. The calculation is performed according to the general formula:

[\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4}
]

Since the function block is declared as a generic type (`eclipse4diac::core::GenericClassName = 'GEN_AR_ADD'`), the underlying data type (e.g., Integer, Real, or user-defined arithmetic types) is only

determined at runtime through the specific configuration. Processing is data-driven – as soon as valid values are present at all four inputs, the sum is calculated and updated at the output.

Technical Features

- **Generic Function Block**: The function block is marked as generic (`eclipse4diac::core::GenericClassName`). This allows the specific arithmetic data type to be defined only when used in the project, enabling high reusability.
- **Adapter-Based Communication**: Instead of classic data inputs/outputs, all values are exchanged via adapters. This allows for loose coupling with other components and promotes modular structures.
- **No Event Control**: The component has no event inputs or outputs. Calculation and data transmission occur automatically as soon as all input values are available (similar to a continuous function).
- **Typical Package Structure**: The component is organized in the package `adapter::iec61131::arithmetic`, indicating an adapter implementation close to IEC 61131.

State Overview

Due to its purely data-driven and eventless operation, the component has **no internal states**. There are no sequential processes or state machines. The output is always the current sum of the four inputs.

Application Scenarios

1. **Calculating the sum of multiple process values** (e.g., adding flow signals from multiple sensors).
2. **Scaling and Summation** – In combination with scalable adapters, the function block can be used for weighted summation.
3. **Calculation of total consumption or total energy** from multiple partial measurements.
4. **Generic summation in modular automation solutions** where the number of summands is fixed, but the data type is variable.

Comparison with similar function blocks

Function block	Number of inputs	Special feature
AR_ADD_4_UNGATED	4	Adapter-based, generic, no events
AR_ADD_2 (hypothetical)	2	Reduced inputs, same concept
F_ADD (from IEC 61131)	2	Standard data types, event-driven (via ENABLE/ENO)
AR_SUM (hypothetical)	variable	More flexible number, but more complex

The **AR_ADD_4_UNGATED** function block stands out from classic IEC 61499 arithmetic function blocks due to its pure adapter communication and generic design. It is particularly suitable for systems that already rely on adapter technology and do not require explicit event control.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

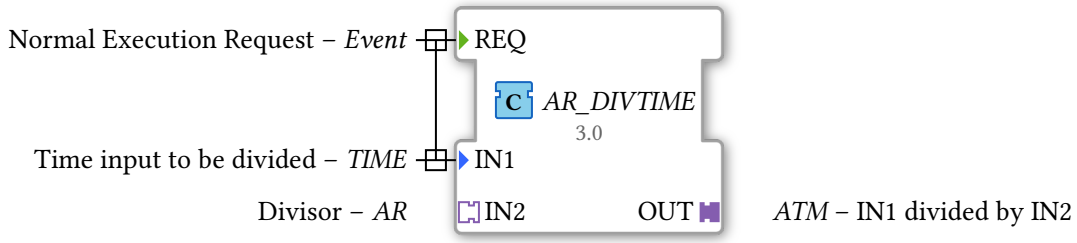
The **AR_ADD_4_UNGATED** is a compact, generic function block for adding four values via adapter interfaces. Its simple and robust data-driven operation makes it a useful basic building block for modular automation applications. The absence of events and the generic typing enable flexible integration into heterogeneous systems. For applications with more or fewer than four addends, adapted variants or extended function blocks must be used.

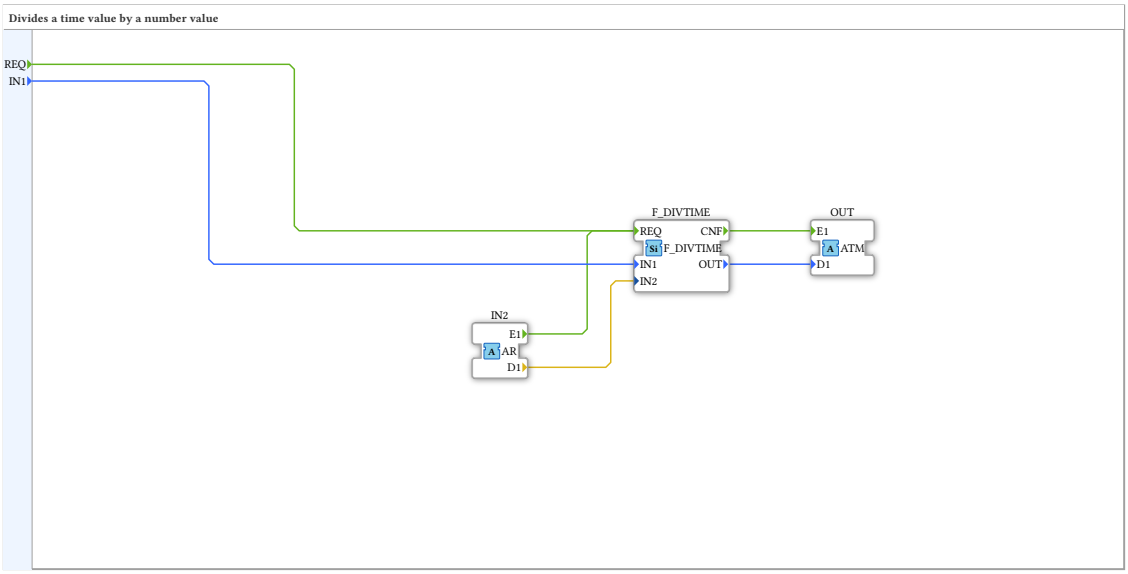
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AR_DIVTIME





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_DIVTIME** is used to divide a time value (type TIME) by a numeric value. The divisor is supplied via an adapter, allowing the block to be used flexibly in different contexts. The result is likewise output via an adapter. The block is classified as a standard arithmetic function according to IEC 61131-3.

Interface Structure

Event Inputs

- **REQ** (Event): Normal execution request. An event at REQ triggers the division. Input IN1 must be connected to the REQ event (WITH).

Event Outputs

The block has no explicit event outputs in its interface list. However, the result is provided via the output adapter **OUT**, which sends an event (E1) once the calculation is complete (as shown by the internal wiring).

Data Inputs

- **IN1** (TIME): Time value to be divided (dividend).

Data Outputs

The block has no direct data outputs. The division result is output via the output adapter **OUT** as a data value (D1).

Adapters

- **IN2** (Adapter, type `adapter::types::unidirectional::AR`): Divisor. This adapter provides both a triggering event (E1) and the data value (D1).

- **OUT** (Adapter, type `adapter::types::unidirectional::ATM`): Output for the division result (TIME). Provides both an event (E1) and data (D1).

Functionality

The function block AR_DIVTIME encapsulates the functionality of the internal function block **F_DIVTIME** (type `iec61131::arithmetic::F_DIVTIME`). An event at input **REQ**, or alternatively at adapter input **IN2.E1**, triggers the division. The internal function block divides **IN1** (TIME) by **IN2.D1** (numeric value) and outputs the result to **OUT.D1**. Once the calculation is complete, an event is sent to **OUT.E1**, signaling to the downstream receiver that the result is valid.

Technical Features

- **Adapter-based interface:** Instead of classic data inputs/outputs, adapters are used, allowing for flexible interconnection in complex networks. The adapters are defined as unidirectional and require compatible counterparts.
- **Standard compliance:** The function block meets the requirements of IEC 61131-3 (classification: standard arithmetic function).
- **Internal encapsulation:** The actual calculation is performed by the reusable function block F_DIVTIME, so AR_DIVTIME acts as a wrapper with adapter-based communication.
- **Mixed interface:** Unlike fully adapter-based blocks, AR_DIVTIME receives the dividend (IN1) as a classic data input, not via an adapter – see `ATM_AR_DIVTIME` for the fully adapter-based variant.

State Overview

The function block itself does not have an explicit state machine. Event-driven processing is deterministic:

- Waiting for an event at **REQ** or **IN2.E1**
- Executing the division
- Sending the result to **OUT.E1** and **OUT.D1**
- Returning to the wait state

Application Scenarios

- Splitting a time value into equal segments, e.g., converting a total runtime into individual cycles.
- In control applications where a time parameter must be dynamically divided by a variable factor.
- As part of libraries for IEC 61131-3 compliant automatic control, especially when adapters are used to decouple the signal paths.

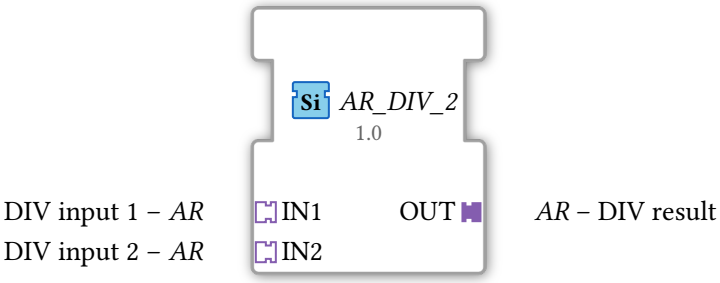
Comparison with Similar Function Blocks

- **F_DIVTIME:** The direct, adapter-free function block for dividing TIME by numeric values. It offers classic event and data interfaces. AR_DIVTIME encapsulates this function block and provides the divisor via an adapter.
- **AR_MULTIME:** Multiplies instead of dividing, otherwise structurally identical (mixed interface: IN1 as a data input, IN2 as an adapter).
- **ATM_AR_DIVTIME:** The fully adapter-based variant – here IN1 is also an adapter socket (type `ATM`) instead of a fixed data input.

Conclusion

The **AR_DIVTIME** function block enables the division of a time value by a numeric factor via an adapter-based interface for the divisor. It combines proven IEC 61131-3 arithmetic with flexible connectivity and is particularly suitable for modular automation systems. The internal use of the F_DIVTIME function block guarantees reliable and efficient calculations.

AR_DIV_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AR_DIV_2` is a generic arithmetic block designed for performing division operations (`DIV`) in IEC 61499-based systems. Unlike traditional function blocks, this block uses adapters (`unidirectional::AR`) to bundle data and events. This results in clearer and more modular visual programming within the 4diac IDE.

Since it is a purely adapter-based function block, `AR_DIV_2` has no direct, traditional data or event inputs and outputs at the main level. All communication is handled via adapters.

Interface Structure

Event Inputs

No direct event inputs available. (Event control is implicit via the input adapters).

Event Outputs

No direct event outputs available. (Event control is implicit via the output adapter).

Data Inputs

No direct data inputs available. (Data is transferred via the input adapters).

Data Outputs

No direct data outputs available. (Data is transferred via the output adapter).

Adapters

Type	Name	Adapter Type	Description
Socket (Input)	`IN1`	`adapter::types::unidirectional::AR`	First input value (dividend) for the division.
Socket (Input)	`IN2`	`adapter::types::unidirectional::AR`	Second input value (divisor) for the division.
Plug (Output)	`OUT`	`adapter::types::unidirectional::AR`	The result of the division (quotient).

Functionality

This function block performs a mathematical division of the values provided via the adapters. As soon as an update event at the input adapters (`IN1` or `IN2`) signals that new data is available, the calculation is triggered:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.62}$$

The calculated result is then passed to the output adapter `OUT`, and the corresponding send event of the adapter is triggered to inform subsequent function blocks about the new value.

Technical Features

- **Generic Type (`GEN_AR_DIV`):** The function block is internally declared as a generic type. This allows for flexible adaptation to various data types (e.g., `REAL`, `LREAL`, `INT`), depending on how the underlying `AR` adapter type is defined.
- **Adapter Coupling:** Using unidirectional adapters (`unidirectional::AR`) drastically reduces the number of connection lines in the function block diagram, improving the readability of complex mathematical calculations.
- **Protection Against Division by Zero:** System-integrated protection measures depend on the specific runtime environment. When using the function block, ensure that the divisor (`IN2`) is not zero to avoid calculation errors or runtime crashes.

State Overview

The function block is essentially stateless and reacts in an event-driven manner:

Application Scenarios

1. **Wait State (Idle):** The function block waits for an update event at the adapters `IN1` or `IN2`.
2. **Calculation:** After an event arrives, the division is performed.
3. **Update:** The result is applied to `OUT`, and the output event is triggered. The function block immediately returns to the wait state.

Comparison with Similar Function Blocks

- **Measurement Scaling:** Division of raw sensor data by a fixed scaling factor for conversion into physical units.
- **Average Calculation:** Division of a sum by the number of measured values in more complex computational networks.
- **Ratio Calculations:** Determining percentages or ratios in process engineering plants.
- **Classic `DIV` Module:** The classic IEC 61131-3 `DIV` module has dedicated pins for `IN1`, `IN2`, and `OUT`, as well as the corresponding `REQ` and `CNF` events. `AR_DIV_2` bundles these signals in adapters, simplifying wiring in large projects.
- **`AR_MUL_2` / `AR_ADD_2`:** These function blocks share the same adapter philosophy but perform multiplications and additions, respectively. They can be seamlessly chained with `AR_DIV_2` to form mathematical calculation trees.

Change Detection

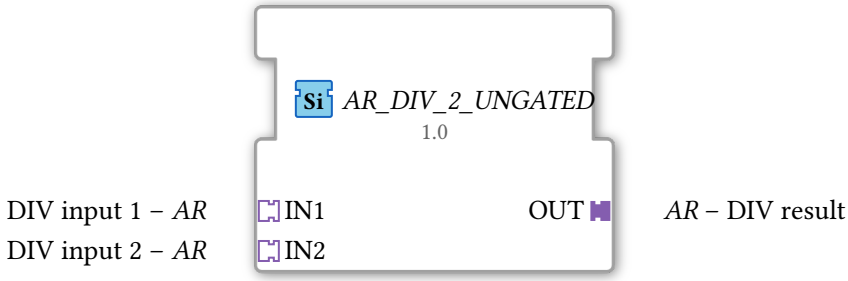
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AR_DIV_2` is a highly efficient, modular function block for arithmetic division. Through the consistent use of adapters, it is ideally suited for clean, well-organized software architectures in distributed automation according to the IEC 61499 standard.

AR_DIV_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_DIV_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AR_DIV_2_UNGATED` is a generic arithmetic block designed for performing division operations (`DIV`) in IEC 61499-based systems. Unlike traditional function blocks, this block uses adapters (`unidirectional::AR`) to bundle data and events. This results in clearer and more modular visual programming within the 4diac IDE.

Since it is a purely adapter-based function block, `AR_DIV_2_UNGATED` has no direct, traditional data or event inputs and outputs at the main level. All communication is handled via adapters.

Interface Structure

Event Inputs

No direct event inputs available. (Event control is implicit via the input adapters).

Event Outputs

No direct event outputs available. (Event control is implicit via the output adapter).

Data Inputs

No direct data inputs available. (Data is transferred via the input adapters).

Data Outputs

No direct data outputs available. (Data is transferred via the output adapter).

Adapters

Type	Name	Adapter Type	Description
Socket (Input)	`IN1`	`adapter::types::unidirectional::AR`	First input value (dividend) for the division.
Socket (Input)	`IN2`	`adapter::types::unidirectional::AR`	Second input value (divisor) for the division.
Plug (Output)	`OUT`	`adapter::types::unidirectional::AR`	The result of the division (quotient).

Functionality

This function block performs a mathematical division of the values provided via the adapters. As soon as an update event at the input adapters (`IN1` or `IN2`) signals that new data is available, the calculation is triggered:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.63}$$

The calculated result is then passed to the output adapter `OUT`, and the corresponding send event of the adapter is triggered to inform subsequent function blocks about the new value.

Technical Features

- **Generic Type (`GEN_AR_DIV`):** The function block is internally declared as a generic type. This allows for flexible adaptation to various data types (e.g., `REAL`, `LREAL`, `INT`), depending on how the underlying `AR` adapter type is defined.
- **Adapter Coupling:** Using unidirectional adapters (`unidirectional::AR`) drastically reduces the number of connection lines in the function block diagram, improving the readability of complex mathematical calculations.
- **Protection Against Division by Zero:** System-integrated protection measures depend on the specific runtime environment. When using the function block, ensure that the divisor (`IN2`) is not zero to avoid calculation errors or runtime crashes.

State Overview

The function block is essentially stateless and reacts in an event-driven manner:

Application Scenarios

1. **Wait State (Idle):** The function block waits for an update event at the adapters `IN1` or `IN2`.
2. **Calculation:** After an event arrives, the division is performed.
3. **Update:** The result is applied to `OUT`, and the output event is triggered. The function block immediately returns to the wait state.

Comparison with Similar Function Blocks

- **Measurement Scaling:** Division of raw sensor data by a fixed scaling factor for conversion into physical units.
- **Average Calculation:** Division of a sum by the number of measured values in more complex computational networks.
- **Ratio Calculations:** Determining percentages or ratios in process engineering plants.
- **Classic `DIV` Module:** The classic IEC 61131-3 `DIV` module has dedicated pins for `IN1`, `IN2`, and `OUT`, as well as the corresponding `REQ` and `CNF` events. `AR_DIV_2_UNGATED` bundles these signals in adapters, simplifying wiring in large projects.
- **`AR_MUL_2` / `AR_ADD_2`:** These function blocks share the same adapter philosophy but perform multiplications and additions, respectively. They can be seamlessly chained with `AR_DIV_2_UNGATED` to form mathematical calculation trees.

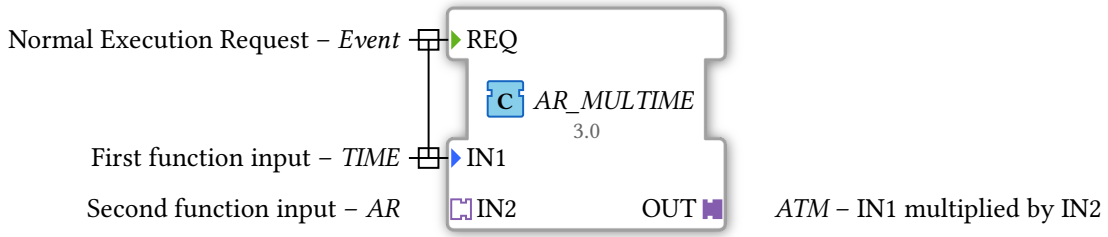
Change Detection

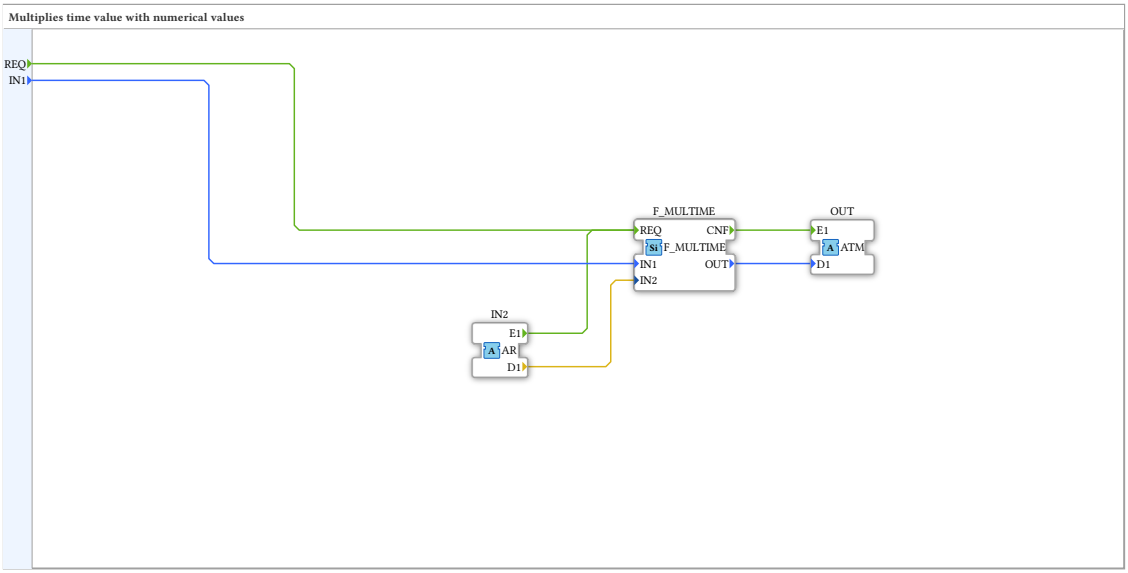
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AR_DIV_2_UNGATED` is a highly efficient, modular function block for arithmetic division. Through the consistent use of adapters, it is ideally suited for clean, well-organized software architectures in distributed automation according to the IEC 61499 standard.

AR_MULTIME





• – ○ • – ○ • – ○ •

Introduction

The function block **AR_MULTIME** is used to multiply a time value (type TIME) by a numeric value. The numeric factor is input via an adapter, allowing the function block to be used flexibly in various contexts. The result is also output via an adapter. The function block is classified as a standard arithmetic function according to IEC 61131-3.

Interface Structure

Event Inputs

- **REQ** (Event): Normal execution trigger. The multiplication is performed when an event occurs at REQ. The IN1 input must be connected to the REQ event (WITH).

Event Outputs

The function block does not have any explicit event outputs in the interface list. The result is provided via the adapter output **OUT**, which sends an event (E1) once the calculation is complete (as can be seen from the internal wiring).

Data Inputs

- **IN1** (TIME): First function input – the time value to be multiplied.

Data Outputs

The function block has no direct data outputs. The multiplication result is output as a data value (D1) via the adapter output **OUT**.

Adapters

- **IN2** (Adapter, type `adapter::types::unidirectional::AR`): Second function input – the numerical multiplier. This adapter provides an event (E1) for triggering the calculation as well as the data value (D1).
- **OUT** (Adapter, type `adapter::types::unidirectional::ATM`): Output for the multiplied result (TIME). Provides the event (E1) and data (D1).

Functionality

The function block AR_MULTIME encapsulates the functionality of the internal function block **F_MULTIME** (type `iec61131::arithmetic::F_MULTIME`).

When an event occurs at the **REQ** input or, alternatively, at the adapter input **IN2.E1**, the multiplication is initiated. The internal function block multiplies **IN1** (TIME) by **IN2.D1** (numeric value) and outputs the result to **OUT.D1**.

Once the calculation is complete, an event is sent to **OUT.E1**, signaling to the subsequent receiver that the result is valid.

Technical Features

- **Adapter-based interface:** Instead of classic data inputs/outputs, adapters are used, allowing for flexible interconnection in complex networks. The adapters are defined as unidirectional and require compatible counterparts.
- **Standard compliance:** The function block meets the requirements of IEC 61131-3 (classification: standard arithmetic function).
- **Internal Encapsulation:** The actual calculation is performed by the reusable function block F_MULTIME, so AR_MULTIME acts as a wrapper with adapter-based communication.

State Overview

The function block itself does not have an explicit state machine. Event-driven processing is deterministic:

- Waiting for an event at **REQ** or **IN2.E1**
- Executing the multiplication
- Sending the result to **OUT.E1** and **OUT.D1**
- Returning to the wait state

Application Scenarios

- Scaling time values by a constant or variable factor, e.g., converting seconds to milliseconds or extending delay times.
- In control applications where time parameters need to be dynamically adjusted, for example, in conjunction with PID controllers or scheduling.
- As part of libraries for IEC 61131-3 compliant automatic control, especially when adapters are used to decouple the signal paths.

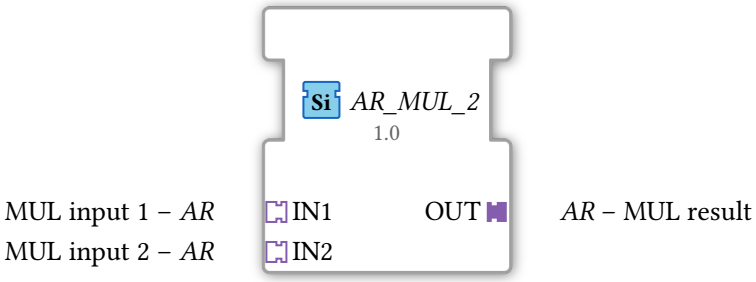
Comparison with similar function blocks

- **F_MULTIME:** The direct, adapter-free function block for multiplying TIME by numeric values. It offers classic event and data interfaces. AR_MULTIME, on the other hand, encapsulates this function block and provides the interfaces via adapters, which increases reusability in adapter-based architectures.
- **AR_MUL:** A general multiplication function block with adapters that could be designed for any numeric type. AR_MULTIME is specifically specialized for TIME values.

Conclusion

The **AR_MULTIME** function block enables the multiplication of a time value by a numeric factor via a modern, adapter-based interface. It combines proven IEC 61131-3 arithmetic with flexible connectivity and is particularly suitable for modular automation systems that rely on standardized adapters. The internal use of the F_MULTIME function block guarantees reliable and efficient calculations.

AR_MUL_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) **AR_MUL_2** is a generic arithmetic block for the 4diac IDE. It is used to multiply two values that are transmitted via unidirectional adapters. By encapsulating the mathematical operation

in an adapter interface, this block enables clean, modular, and clear modeling of calculations within IEC 61499 applications.

Interface Structure

The block does not have any classic event- or data-based inputs and outputs. All communication is handled via adapters.

Event Inputs

No direct event inputs available. (Event control is implemented via the adapters.)

Event Outputs

No direct event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AR`): First input value (multiplicand 1) for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::AR`): Second input value (multiplicand 2) for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AR`): Output for the calculated product of the multiplication ($OUT = IN1 \times IN2$).

Functionality

As soon as new values or corresponding trigger events arrive via the input adapters `IN1` and `IN2`, the function block multiplies the values of the two inputs. The result of the mathematical operation ($IN1 \cdot IN2$) is immediately passed to the output adapter `OUT` and made available for subsequent function blocks.

Since this is a generic function block (`GEN_AR_MUL`), its implementation is flexible with respect to the specific data types used within the `AR` adapter structure.

Technical Features

- **Generic Function Block:** The attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AR_MUL'` allows the function block to be used for various numeric data types, provided the adapters support them.
- **Unidirectional Adapters:** Using the type `unidirectional::AR` ensures a clean data flow without feedback loops, which increases application stability and minimizes latency.
- **Compiler Affiliation:** The function block is assigned to the package `adapter::iec61131::arithmetic` and conforms to the IEC 61499-2 standard.

State Overview

Since this function block primarily operates via adapters using data flow control, it does not have a complex internal state diagram (ECC). It acts as a stateless transfer element:

1. **Waiting:** The block waits for data updates at the adapters `IN1` and/or `IN2`.
2. **Calculation:** When a value changes, the product is calculated.
3. **Output:** The result is directly forwarded to the output `OUT`.

–

Application Scenarios

- **Measurement Scaling:** Multiplication of an analog sensor value (e.g., a voltage) by a scaling factor to convert it into a physical unit.
- **Signal Amplification:** Proportional amplification of control signals in process automation.
- **Mathematical Calculations:** Use as a modular component in more complex computational networks within 4diac applications.

Comparison with Similar Components

Compared to standard multiplication components (such as the classic `MUL` component according to IEC 61131-3), which work with elementary data types (e.g., `INT`, `REAL`), `AR_MUL_2` uses structured **adapters**. This offers the following advantages:

- **Reduced Wiring Effort:** Signals and associated control events are bundled in a single adapter connection.
- **Improved Readability:** Complex control diagrams remain clear because fewer individual connections need to be drawn on the workspace.

Change Detection

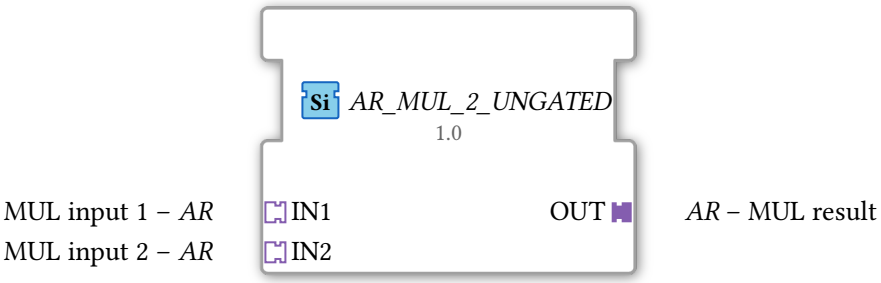
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AR_MUL_2` function block is an efficient tool for arithmetic operations in modern, adapter-based control programs. Its encapsulation in unidirectional interfaces offers a high degree of reusability and ensures a clean and structured design within the 4diac environment.

AR_MUL_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_MUL_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) **AR_MUL_2_UNGATED** is a generic arithmetic block for the 4diac IDE. It is used to multiply two values that are transmitted via unidirectional adapters. By encapsulating the mathematical operation in an adapter interface, this block enables clean, modular, and clear modeling of calculations within IEC 61499 applications.

Interface Structure

The block does not have any classic event- or data-based inputs and outputs. All communication is handled via adapters.

Event Inputs

No direct event inputs available. (Event control is implemented via the adapters.)

Event Outputs

No direct event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AR`): First input value (multiplicand 1) for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::AR`): Second input value (multiplicand 2) for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AR`): Output for the calculated product of the multiplication ($OUT = IN1 \times IN2$).

Functionality

As soon as new values or corresponding trigger events arrive via the input adapters `IN1` and `IN2`, the function block multiplies the values of the two inputs. The result of the mathematical operation ($IN1 \cdot IN2$) is immediately passed to the output adapter `OUT` and made available for subsequent function blocks.

Since this is a generic function block (`GEN_AR_MUL`), its implementation is flexible with respect to the specific data types used within the `AR` adapter structure.

Technical Features

- **Generic Function Block:** The attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AR_MUL'` allows the function block to be used for various numeric data types, provided the adapters support them.
- **Unidirectional Adapters:** Using the type `unidirectional::AR` ensures a clean data flow without feedback loops, which increases application stability and minimizes latency.
- **Compiler Affiliation:** The function block is assigned to the package `adapter::iec61131::arithmetic` and conforms to the IEC 61499-2 standard.

•

State Overview

Since this function block primarily operates via adapters using data flow control, it does not have a complex internal state diagram (ECC). It acts as a stateless transfer element:

1. **Waiting:** The block waits for data updates at the adapters `IN1` and/or `IN2`.
2. **Calculation:** When a value changes, the product is calculated.
3. **Output:** The result is directly forwarded to the output `OUT`.

–

Application Scenarios

- **Measurement Scaling:** Multiplication of an analog sensor value (e.g., a voltage) by a scaling factor to convert it into a physical unit.

- **Signal Amplification:** Proportional amplification of control signals in process automation.
- **Mathematical Calculations:** Use as a modular component in more complex computational networks within 4diac applications.

Comparison with Similar Components

Compared to standard multiplication components (such as the classic `MUL` component according to IEC 61131-3), which work with elementary data types (e.g., `INT`, `REAL`), `AR_MUL_2_UNGATED` uses structured **adapters**. This offers the following advantages:

- **Reduced Wiring Effort:** Signals and associated control events are bundled in a single adapter connection.
- **Improved Readability:** Complex control diagrams remain clear because fewer individual connections need to be drawn on the workspace.

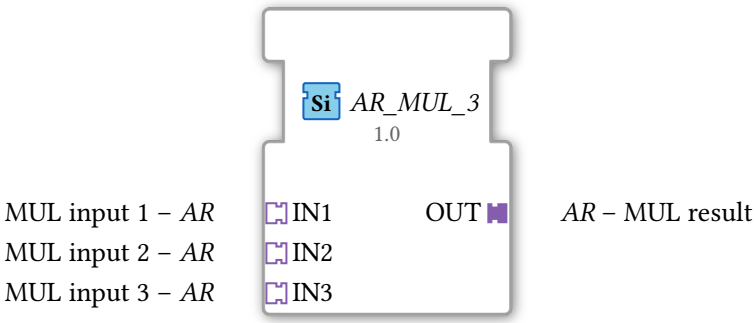
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AR_MUL_2_UNGATED` function block is an efficient tool for arithmetic operations in modern, adapter-based control programs. Its encapsulation in unidirectional interfaces offers a high degree of reusability and ensures a clean and structured design within the 4diac environment.

AR_MUL_3



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AR_MUL_3` is a generic arithmetic block used to multiply three input values. It is based on the use of unidirectional adapters of type `AR` (Arithmetic), which enables structured and clear signal transmission within the 4diac IDE. As it is a generic block, it can be used flexibly with various numeric data types.

Interface Structure

The interfaces of this function block are implemented entirely via adapters. There are no directly accessible standard event or data channels.

Event Inputs

There are no direct event inputs. Event control is handled via the input adapters.

Event Outputs

There are no direct event outputs. Event control is handled via the output adapter.

Data Inputs

There are no direct data inputs. Data transmission occurs via the adapters.

Data Outputs

There are no direct data outputs. Data transmission occurs via the adapters.

Adapters

- **Sockets (Input Adapters):**
- `IN1` (Type: `adapter::types::unidirectional::AR`): First multiplicand (Input 1).
- `IN2` (Type: `adapter::types::unidirectional::AR`): Second multiplicand (Input 2).
- `IN3` (Type: `adapter::types::unidirectional::AR`): Third multiplicand (Input 3).
- **Plugs (Output Adapters):**
- `OUT` (Type: `adapter::types::unidirectional::AR`): Result of the multiplication ($OUT = IN1 \cdot IN2 \cdot IN3$).

Functionality

The function block performs a mathematical multiplication of the values present at the adapters `IN1`, `IN2`, and `IN3`:

$$OUT = IN1 \times IN2 \times IN3 \tag{2.64}$$

As soon as a calculation event (e.g., a value update) is signaled at the input adapters, the function block reads the current values, calculates the product, and outputs the result along with a corresponding update event via the output adapter `OUT`.

Technical Features

- **Generic Behavior (`GEN_AR_MUL`):** The function block is declared as a generic type. This allows it to be applied to various numeric data types (e.g., `INT`, `REAL`, `LREAL`) during development or runtime, provided the adapters used support these data types.
- **Adapter Structure:** Using `unidirectional::AR` adapters drastically reduces the number of visible connection lines in the function block diagram (FBD) because data and control events are bundled in a single connection.

State Overview

The function block behaves purely functionally and essentially has the following logical states:

- **Waiting (Idle):** The function block waits for a trigger event via the input adapters.
- **Evaluating:** After an event arrives, the input data is read and multiplied.
- **Update:** The calculated product is applied to output `OUT`, and the output event is triggered.

Application Scenarios

- **Volume Calculations:** Multiplying three dimensions (length × width × height) to determine a volume.
- **Scaling and Weighting:** Applying two consecutive scaling factors to a raw value (e.g., sensor value × calibration factor × unit conversion).
- **Physical Formulas:** Calculating quantities that depend directly on three variables (e.g., power $P = U \times I \times cos(\varphi)$ in a simplified view).

Comparison with Similar Function Blocks

- **Standard `MUL` (IEC 61131-3):** Classic multiplication blocks operate with discrete data and event pins. `AR_MUL_3` uses adapters instead, which makes the design clearer.
- **`AR_MUL_2`:** Multiplies only two values. `AR_MUL_3` eliminates the need for an additional cascading block when three variables need to be multiplied, thus optimizing performance and clarity.

Change Detection

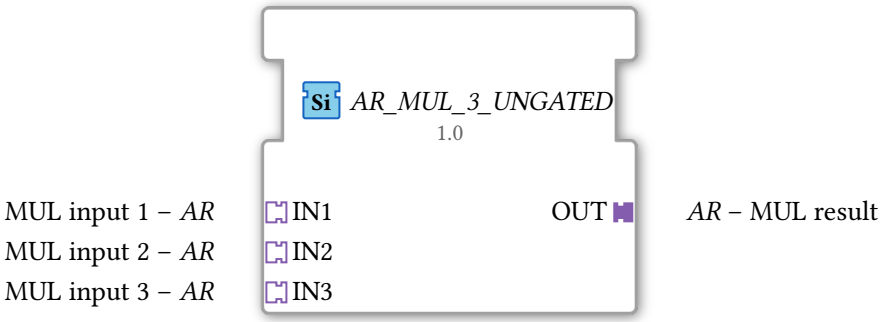
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AR_MUL_3` is a practical and reusable function block for modern IEC 61499 development in 4diac. By encapsulating the mathematical logic in an adapter-based structure, it significantly contributes to the clarity of complex control applications.

AR_MUL_3_UNGATED

i UNGATED variant: This block is the ungated version of AR_MUL_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block AR_MUL_3_UNGATED is a generic arithmetic block used to multiply three input values. It is based on the use of unidirectional adapters of type AR (Arithmetic), which enables structured and clear signal transmission within the 4diac IDE. As it is a generic block, it can be used flexibly with various numeric data types.

Interface Structure

The interfaces of this function block are implemented entirely via adapters. There are no directly accessible standard event or data channels.

Event Inputs

There are no direct event inputs. Event control is handled via the input adapters.

Event Outputs

There are no direct event outputs. Event control is handled via the output adapter.

Data Inputs

There are no direct data inputs. Data transmission occurs via the adapters.

Data Outputs

There are no direct data outputs. Data transmission occurs via the adapters.

Adapters

- **Sockets (Input Adapters):**
- IN1 (Type: adapter::types::unidirectional::AR): First multiplicand (Input 1).
- IN2 (Type: adapter::types::unidirectional::AR): Second multiplicand (Input 2).
- IN3 (Type: adapter::types::unidirectional::AR): Third multiplicand (Input 3).
- **Plugs (Output Adapters):**
- OUT (Type: adapter::types::unidirectional::AR): Result of the multiplication ($OUT = IN1 \cdot IN2 \cdot IN3$).

Functionality

The function block performs a mathematical multiplication of the values present at the adapters IN1, IN2, and IN3:

$$OUT = IN1 \times IN2 \times IN3 \tag{2.65}$$

As soon as a calculation event (e.g., a value update) is signaled at the input adapters, the function block reads the current values, calculates the product, and outputs the result along with a corresponding update event via the output adapter `OUT`.

Technical Features

- **Generic Behavior (`GEN_AR_MUL`):** The function block is declared as a generic type. This allows it to be applied to various numeric data types (e.g., `INT`, `REAL`, `LREAL`) during development or runtime, provided the adapters used support these data types.
- **Adapter Structure:** Using `unidirectional::AR` adapters drastically reduces the number of visible connection lines in the function block diagram (FBD) because data and control events are bundled in a single connection.

State Overview

The function block behaves purely functionally and essentially has the following logical states:

- **Waiting (Idle):** The function block waits for a trigger event via the input adapters.
- **Evaluating:** After an event arrives, the input data is read and multiplied.
- **Update:** The calculated product is applied to output `OUT`, and the output event is triggered.

Application Scenarios

- **Volume Calculations:** Multiplying three dimensions (length × width × height) to determine a volume.
- **Scaling and Weighting:** Applying two consecutive scaling factors to a raw value (e.g., sensor value × calibration factor × unit conversion).
- **Physical Formulas:** Calculating quantities that depend directly on three variables (e.g., power $P = U \times I \times cos(\varphi)$ in a simplified view).

Comparison with Similar Function Blocks

- **Standard `MUL` (IEC 61131-3):** Classic multiplication blocks operate with discrete data and event pins. `AR_MUL_3_UNGATED` uses adapters instead, which makes the design clearer.
- **`AR_MUL_2`:** Multiplies only two values. `AR_MUL_3_UNGATED` eliminates the need for an additional cascading block when three variables need to be multiplied, thus optimizing performance and clarity.

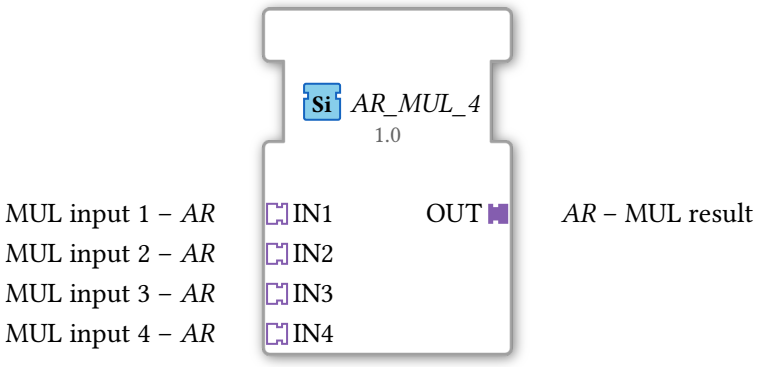
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AR_MUL_3_UNGATED` is a practical and reusable function block for modern IEC 61499 development in 4diac. By encapsulating the mathematical logic in an adapter-based structure, it significantly contributes to the clarity of complex control applications.

AR_MUL_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block AR_MUL_4 is a generic arithmetic block used to multiply four input values. It is based on an adapter interface that enables clean encapsulation and structuring of data and associated events. Due to its generic nature (GEN_AR_MUL), the block can flexibly adapt to various numeric data types.

Interface Structure

Event Inputs

The block does not have direct, dedicated event inputs. Synchronization and triggering of the calculation are implicit via the connected input adapters.

Event Outputs

The block does not have direct, dedicated event outputs. Calculation and update events are forwarded via the output adapter.

Data Inputs

There are no direct data inputs. The values for multiplication are read via the adapter interfaces.

Data Outputs

There are no direct data outputs. The result of the multiplication is provided via the output adapter.

Adapters

Communication with other function blocks is exclusively via adapters of type adapter::types::unidirectional::AR:

- **Sockets (Input Adapters):**
- IN1: First multiplicand (Input 1).
- IN2: Second multiplicand (Input 2).
- IN3: Third multiplicand (Input 3).
- IN4: Fourth multiplicand (Input 4).
- **Plugs (Output Adapters):**
- OUT: The calculated product of the four inputs.

Functionality

As soon as new values and the corresponding trigger events arrive via the input adapters (IN1 to IN4), the function block multiplies the values together. The mathematical behavior corresponds to the formula:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.66}$$

The result is output via the output adapter OUT, along with a corresponding update event.

Technical Features

- **Generic Behavior:** The attribute eclipse4diac::core::GenericClassName with the value GEN_AR_MUL makes the function block data type-independent. Depending on the application, it can work with various numeric data types (e.g., INT, REAL, LREAL) supported by the adapter type AR.
- **Adapter Coupling:** The use of unidirectional adapters (AR) significantly reduces wiring effort in the 4diac IDE, as data and control events are bundled in a single connection.

State Overview

Since this is a purely mathematical calculation function block, it is stateless. Each activation directly calculates the current product based on the input adapter values.

Application Scenarios

- **Signal Scaling:** Calibration or adjustment of sensor values where multiple factors (e.g., base value, gain, correction factor, unit conversion) need to be multiplied.
- **Physical Calculations:** Calculation of more complex quantities such as volumetric flow rates, electrical power, or energy flows, involving multiple measured variables and constant factors.
- **Cascade Avoidance:** Consolidating multiple multiplication steps into a single block to improve clarity in the application diagram.

Comparison with Similar Function Blocks

Compared to conventional IEC 61131-3 `MUL` function blocks, which often only have two inputs and use direct data and event pins, the `AR_MUL_4` offers a significantly cleaner visual representation in the control program by combining four inputs and using adapters. It eliminates the need to cascade multiple multipliers.

Change Detection

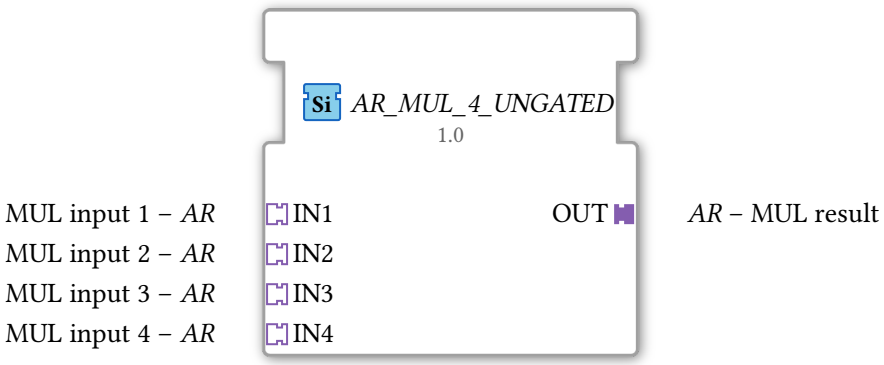
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AR_MUL_4` is a practical auxiliary function block for arithmetic operations in the IEC 61499 environment. Through the consistent use of adapters, it significantly contributes to the clarity and modularity of control programs, especially when processing more complex mathematical formulas.

AR_MUL_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_MUL_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AR_MUL_4_UNGATED` is a generic arithmetic block used to multiply four input values. It is based on an adapter interface that enables clean encapsulation and structuring of data and associated events. Due to its generic nature (`GEN_AR_MUL`), the block can flexibly adapt to various numeric data types.

Interface Structure

Event Inputs

The block does not have direct, dedicated event inputs. Synchronization and triggering of the calculation are implicit via the connected input adapters.

Event Outputs

The block does not have direct, dedicated event outputs. Calculation and update events are forwarded via the output adapter.

Data Inputs

There are no direct data inputs. The values for multiplication are read via the adapter interfaces.

Data Outputs

There are no direct data outputs. The result of the multiplication is provided via the output adapter.

Adapters

Communication with other function blocks is exclusively via adapters of type `adapter::types::unidirectional::AR`:

- **Sockets (Input Adapters):**
- `IN1`: First multiplicand (Input 1).
- `IN2`: Second multiplicand (Input 2).
- `IN3`: Third multiplicand (Input 3).
- `IN4`: Fourth multiplicand (Input 4).
- **Plugs (Output Adapters):**
- `OUT`: The calculated product of the four inputs.

Functionality

As soon as new values and the corresponding trigger events arrive via the input adapters (`IN1` to `IN4`), the function block multiplies the values together. The mathematical behavior corresponds to the formula:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.67}$$

The result is output via the output adapter `OUT`, along with a corresponding update event.

Technical Features

- **Generic Behavior:** The attribute `eclipse4diac::core::GenericClassName` with the value `GEN_AR_MUL` makes the function block data type-independent. Depending on the application, it can work with various numeric data types (e.g., `INT`, `REAL`, `LREAL`) supported by the adapter type `AR`.
- **Adapter Coupling:** The use of unidirectional adapters (`AR`) significantly reduces wiring effort in the 4diac IDE, as data and control events are bundled in a single connection.

State Overview

Since this is a purely mathematical calculation function block, it is stateless. Each activation directly calculates the current product based on the input adapter values.

Application Scenarios

- **Signal Scaling:** Calibration or adjustment of sensor values where multiple factors (e.g., base value, gain, correction factor, unit conversion) need to be multiplied.
- **Physical Calculations:** Calculation of more complex quantities such as volumetric flow rates, electrical power, or energy flows, involving multiple measured variables and constant factors.
- **Cascade Avoidance:** Consolidating multiple multiplication steps into a single block to improve clarity in the application diagram.

Comparison with Similar Function Blocks

Compared to conventional IEC 61131-3 `MUL` function blocks, which often only have two inputs and use direct data and event pins, the `AR_MUL_4_UNGATED` offers a significantly cleaner visual representation in the control program by combining four inputs and using adapters. It eliminates the need to cascade multiple multipliers.

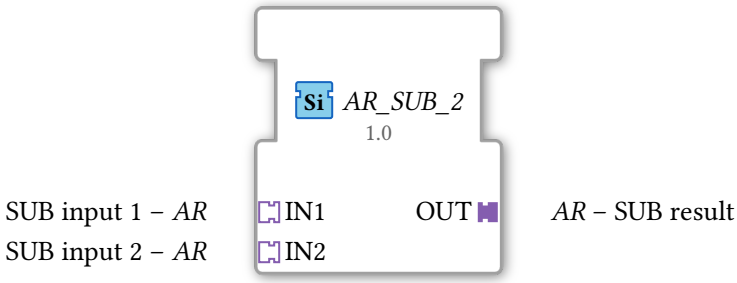
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AR_MUL_4_UNGATED` is a practical auxiliary function block for arithmetic operations in the IEC 61499 environment. Through the consistent use of adapters, it significantly contributes to the clarity and modularity of control programs, especially when processing more complex mathematical formulas.

AR_SUB_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AR_SUB_2` is used to perform an arithmetic subtraction (subtrahend of minuend). It is a generic function block (`GEN_AR_SUB`) that does not use classic discrete inputs and outputs for the calculation, but is based entirely on an adapter-based interface structure. The values and the associated control logic are transferred via unidirectional adapters of type `AR`.

Interface Structure

Event Inputs

This function block does not have direct, discrete event inputs. Event control is implemented via the connected adapters.

Event Outputs

This function block does not have direct, discrete event outputs. Event forwarding is encapsulated via the output adapter.

Data Inputs

There are no direct data inputs. Input data is received via the adapter inputs.

Data Outputs

There are no direct data outputs. The calculation result is provided via the adapter output.

Adapters

- **Sockets (Input Adapters):**
- **IN1** (Type: `adapter::types::unidirectional::AR`): First input for subtraction (minuend).
- **IN2** (Type: `adapter::types::unidirectional::AR`): Second input for subtraction (subtrahend).
- **Plugs (Output Adapters):**
- **OUT** (Type: `adapter::types::unidirectional::AR`): Output for the result of the subtraction ($OUT = IN1 - IN2$).

Functionality

The function block subtracts the value at adapter `IN2` from the value at adapter `IN1`. The result of this arithmetic operation is passed to the output plug `OUT`.

Since these are unidirectional adapters, incoming events on sockets `IN1` or `IN2` trigger the internal calculation. After the subtraction is complete, the corresponding update event is signaled to the subsequent function blocks via plug `OUT`.

Technical Features

- **Generic Type (`GEN_AR_SUB`)**: The function block is implemented generically. This allows for flexible handling of various numeric data types, provided they are permitted within the definition of the `AR` adapter structure.
- **Encapsulation**: By using adapters, data and event lines are bundled. This significantly reduces the number of visible connection lines in the 4diac IDE function block diagram.

State Overview

The function block does not have a complex internal state machine (no Execution Control Chart - ECC) because it is a purely data-flow-oriented computation block. The processing behavior is as follows:

1. **Waiting for Event**: The function block waits for an update event at `IN1` or `IN2`.
2. **Calculation**: Upon receiving an event, the current values from `IN1` and `IN2` are read and subtracted from each other.
3. **Output**: The result is written to `OUT`, and an output event is triggered at plug `OUT`.

Application Scenarios

- **Target-Actual Value Comparison**: Calculation of the control deviation ($e = w - x$) in control loops where the signals are already available as adapter structures.
- **Offset Compensation**: Subtraction of a zero-point error or offset from an analog sensor value.
- **Structured Signal Processing**: Mathematical calculations in complex, distributed control systems to maintain a clear software architecture.

Comparison with Similar Components

Compared to the standard subtraction block `SUB` (based on classic IEC 61131-3 elements), which uses discrete inputs like `IN1` and `IN2` as well as explicit events (`REQ` / `CNF`), `AR_SUB_2` offers a significantly higher level of abstraction through the use of adapters. This saves development time when coupling complex signal elements, but requires consistent use of adapters throughout the entire project.

Change Detection

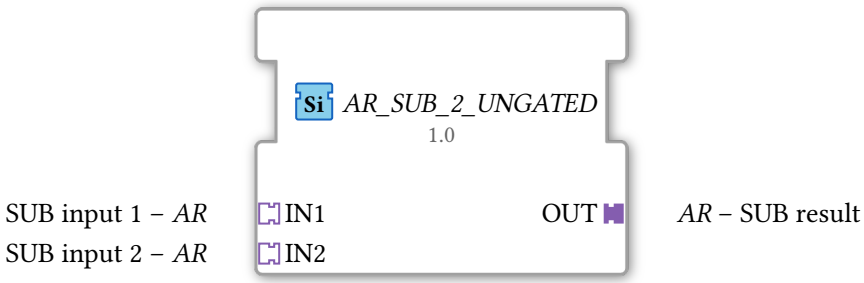
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AR_SUB_2` is a specialized and modern computing component for the 4diac IDE. It is ideally suited for service-oriented architectures within IEC 61499, where clarity and standardized adapter interfaces are paramount.

AR_SUB_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_SUB_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AR_SUB_2_UNGATED` is used to perform an arithmetic subtraction (subtrahend of minuend). It is a generic function block (`GEN_AR_SUB`) that does not use classic discrete inputs and outputs for the calculation, but is based entirely on an adapter-based interface structure. The values and the associated control logic are transferred via unidirectional adapters of type `AR`.

Interface Structure

Event Inputs

This function block does not have direct, discrete event inputs. Event control is implemented via the connected adapters.

Event Outputs

This function block does not have direct, discrete event outputs. Event forwarding is encapsulated via the output adapter.

Data Inputs

There are no direct data inputs. Input data is received via the adapter inputs.

Data Outputs

There are no direct data outputs. The calculation result is provided via the adapter output.

Adapters

- **Sockets (Input Adapters):**
- **IN1** (Type: `adapter::types::unidirectional::AR`): First input for subtraction (minuend).
- **IN2** (Type: `adapter::types::unidirectional::AR`): Second input for subtraction (subtrahend).
- **Plugs (Output Adapters):**
- **OUT** (Type: `adapter::types::unidirectional::AR`): Output for the result of the subtraction ($OUT = IN1 - IN2$).

Functionality

The function block subtracts the value at adapter `IN2` from the value at adapter `IN1`. The result of this arithmetic operation is passed to the output plug `OUT`.

Since these are unidirectional adapters, incoming events on sockets `IN1` or `IN2` trigger the internal calculation. After the subtraction is complete, the corresponding update event is signaled to the subsequent function blocks via plug `OUT`.

Technical Features

- **Generic Type (`GEN_AR_SUB`)**: The function block is implemented generically. This allows for flexible handling of various numeric data types, provided they are permitted within the definition of the `AR` adapter structure.
- **Encapsulation**: By using adapters, data and event lines are bundled. This significantly reduces the number of visible connection lines in the 4diac IDE function block diagram.

State Overview

The function block does not have a complex internal state machine (no Execution Control Chart - ECC) because it is a purely data-flow-oriented computation block. The processing behavior is as follows:

1. **Waiting for Event**: The function block waits for an update event at `IN1` or `IN2`.
2. **Calculation**: Upon receiving an event, the current values from `IN1` and `IN2` are read and subtracted from each other.
3. **Output**: The result is written to `OUT`, and an output event is triggered at plug `OUT`.

Application Scenarios

- **Target-Actual Value Comparison**: Calculation of the control deviation ($e = w - x$) in control loops where the signals are already available as adapter structures.
- **Offset Compensation**: Subtraction of a zero-point error or offset from an analog sensor value.
- **Structured Signal Processing**: Mathematical calculations in complex, distributed control systems to maintain a clear software architecture.

Comparison with Similar Components

Compared to the standard subtraction block `SUB` (based on classic IEC 61131-3 elements), which uses discrete inputs like `IN1` and `IN2` as well as explicit events (`REQ` / `CNF`), `AR_SUB_2_UNGATED` offers a significantly higher level of abstraction through the use of adapters. This saves development time when coupling complex signal elements, but requires consistent use of adapters throughout the entire project.

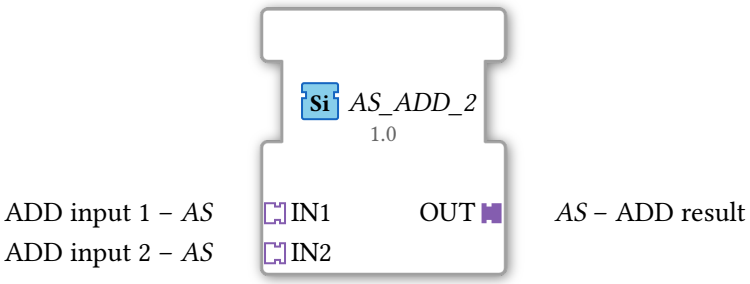
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AR_SUB_2_UNGATED` is a specialized and modern computing component for the 4diac IDE. It is ideally suited for service-oriented architectures within IEC 61499, where clarity and standardized adapter interfaces are paramount.

AS_ADD_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AS_ADD_2` is a generic arithmetic block designed for adding two values. Unlike classic mathematical function blocks, this block uses an adapter-based interface concept. The use of unidirectional adapters allows data and the associated control events to be transmitted in encapsulated form, contributing to a clean and modular application design in the IEC 61499 development environment (such as 4diac IDE).

Interface Structure

Event Inputs

No direct event inputs are available. Event control is integrated into the input adapters.

Event Outputs

No direct event outputs are available. Event control is integrated into the output adapter.

Data Inputs

No direct data inputs are available. Data transmission is encapsulated via the input adapters.

Data Outputs

No direct data outputs are available. Data transmission is encapsulated via the output adapter.

Adapters

- **IN1 (Socket):** Type `adapter::types::unidirectional::AS`
- Interface for the first addend of the addition.
- **IN2 (Socket):** Type `adapter::types::unidirectional::AS`
- Interface for the second addend of the addition.
- **OUT (Plug):** Type `adapter::types::unidirectional::AS`
- Interface for outputting the calculated addition result.

Functionality

The function block `AS_ADD_2` performs the mathematical operation:

$$OUT = IN1 + IN2 \tag{2.68}$$

As soon as new data is signaled at the input adapters (`IN1` or `IN2`), the function block internally triggers the addition of the transmitted values. The result is then immediately calculated and passed on to the subsequent logic elements via the output adapter `OUT`, along with a corresponding update event.

Due to its generic nature (GEN_AS_ADD), the function block can flexibly work with various numeric data types, provided the underlying adapter type AS supports them.

- **Generic Implementation:** The function block is based on the generic class GEN_AS_ADD, making it reusable for different data types.
- **Adapter Coupling:** Encapsulating signals in adapters drastically reduces the number of visible connection lines in the function block diagram (FBD) and improves the readability of complex applications.
- **Unidirectional Data Flow:** Using the type unidirectional::AS ensures that the information flow is clearly defined, running from the signal sources (sockets) to the signal sink (plug).

Technical Features

As a purely mathematical combination block, AS_ADD_2 does not have a complex internal state diagram (ECC). Its behavior can be divided into three cyclical steps:

State Overview

1. **Waiting (Idle):** The module waits for an event at one of the input adapters (IN1 or IN2).
2. **Calculate:** Upon receiving an event, the current values from both adapters are read and added together.
3. **Send:** The result of the addition is written to the output adapter OUT, triggering an output event.

Application Scenarios

- **Measurement Offset Calculation:** Adding a calibration or correction value (offset) to an analog sensor value within an adapter-based signal processing chain.
- **Signal Combining:** Summing two independently measured physical quantities (e.g., two partial currents to determine the total current).
- **Cascaded Calculations:** Easy expansion for more than two summands by cascading multiple AS_ADD_2 blocks.
- **Standard ADD (e.g., F_ADD):** The classic IEC 61131-3 or IEC 61499 ADD block works with discrete variables (e.g., ANY_NUM) and separate event ports (REQ / CNF). AS_ADD_2, on the other hand, bundles these signals in adapters, which simplifies wiring but requires the use of the specific adapter type AS.
- **Multi-Adder (e.g., ADD_3):** Enables the addition of three or more values in a single function block, but is often more cumbersome when data structures need to be consistently transported via adapters.

Comparison with Similar Function Blocks

The AS_ADD_2 is a specialized auxiliary function block for modern, modular IEC 61499 control programs. Through the consistent use of adapters, it integrates seamlessly into service-oriented architectures and minimizes design and wiring effort in the 4diac IDE.

Change Detection

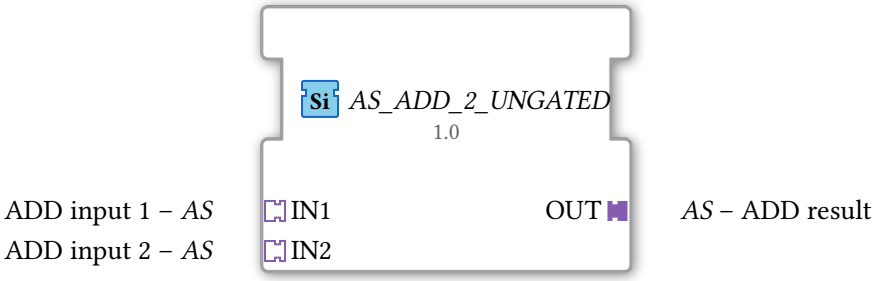
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AS_ADD_2 is a clean, adapter-based addition block that keeps a two-input arithmetic operation consistent with the rest of the adapter-based AS family, making it easy to wire and reuse across IEC 61499 applications.

AS_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AS_ADD_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AS_ADD_2_UNGATED` is a generic arithmetic block designed for adding two values. Unlike classic mathematical function blocks, this block uses an adapter-based interface concept. The use of unidirectional adapters allows data and the associated control events to be transmitted in encapsulated form, contributing to a clean and modular application design in the IEC 61499 development environment (such as 4diac IDE).

Interface Structure

Event Inputs

No direct event inputs are available. Event control is integrated into the input adapters.

Event Outputs

No direct event outputs are available. Event control is integrated into the output adapter.

Data Inputs

No direct data inputs are available. Data transmission is encapsulated via the input adapters.

Data Outputs

No direct data outputs are available. Data transmission is encapsulated via the output adapter.

Adapters

- **IN1 (Socket):** Type `adapter::types::unidirectional::AS`
- Interface for the first addend of the addition.
- **IN2 (Socket):** Type `adapter::types::unidirectional::AS`
- Interface for the second addend of the addition.
- **OUT (Plug):** Type `adapter::types::unidirectional::AS`
- Interface for outputting the calculated addition result.

Functionality

The function block `AS_ADD_2_UNGATED` performs the mathematical operation:

$$OUT = IN1 + IN2 \tag{2.69}$$

As soon as new data is signaled at the input adapters (`IN1` or `IN2`), the function block internally triggers the addition of the transmitted values. The result is then immediately calculated and passed on to the subsequent logic elements via the output adapter `OUT`, along with a corresponding update event.

Due to its generic nature (`GEN_AS_ADD`), the function block can flexibly work with various numeric data types, provided the underlying adapter type `AS` supports them.

- **Generic Implementation:** The function block is based on the generic class `GEN_AS_ADD`, making it reusable for different data types.
- **Adapter Coupling:** Encapsulating signals in adapters drastically reduces the number of visible connection lines in the function block diagram (FBD) and improves the readability of complex applications.
- **Unidirectional Data Flow:** Using the type `unidirectional::AS` ensures that the information flow is clearly defined, running from the signal sources (sockets) to the signal sink (plug).

Technical Features

As a purely mathematical combination block, `AS_ADD_2_UNGATED` does not have a complex internal state diagram (ECC). Its behavior can be divided into three cyclical steps:

State Overview

1. **Waiting (Idle):** The module waits for an event at one of the input adapters (`IN1` or `IN2`).
2. **Calculate:** Upon receiving an event, the current values from both adapters are read and added together.
3. **Send:** The result of the addition is written to the output adapter `OUT`, triggering an output event.

Application Scenarios

- **Measurement Offset Calculation:** Adding a calibration or correction value (offset) to an analog sensor value within an adapter-based signal processing chain.
- **Signal Combining:** Summing two independently measured physical quantities (e.g., two partial currents to determine the total current).
- **Cascaded Calculations:** Easy expansion for more than two summands by cascading multiple `AS_ADD_2_UNGATED` blocks.
- **Standard ADD (e.g., F_ADD):** The classic IEC 61131-3 or IEC 61499 ADD block works with discrete variables (e.g., `ANY_NUM`) and separate event ports (`REQ` / `CNF`). `AS_ADD_2_UNGATED`, on the other hand, bundles these signals in adapters, which simplifies wiring but requires the use of the specific adapter type `AS`.
- **Multi-Adder (e.g., ADD_3):** Enables the addition of three or more values in a single function block, but is often more cumbersome when data structures need to be consistently transported via adapters.

Comparison with Similar Function Blocks

The `AS_ADD_2_UNGATED` is a specialized auxiliary function block for modern, modular IEC 61499 control programs. Through the consistent use of adapters, it integrates seamlessly into service-oriented architectures and minimizes design and wiring effort in the 4diac IDE.

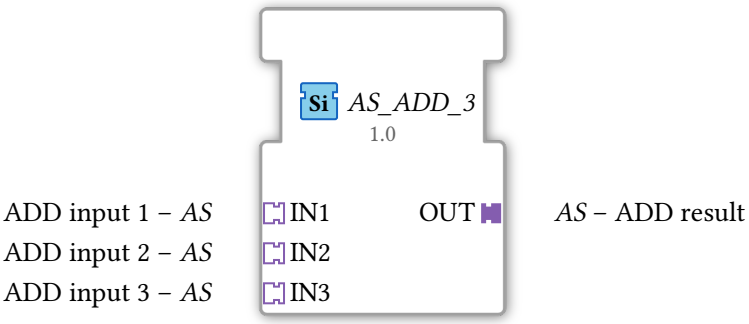
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AS_ADD_2_UNGATED` is a clean, adapter-based addition block that keeps a two-input arithmetic operation consistent with the rest of the adapter-based `AS` family, making it easy to wire and reuse across IEC 61499 applications.

AS_ADD_3



Introduction

The function block `AS_ADD_3` is a generic function block (FB) for IEC 61499, designed to perform arithmetic addition of three input values. A key feature of this block is that it has no direct data or event inputs/outputs, but relies entirely on the use of unidirectional adapters. This significantly simplifies modeling and wiring in more complex systems.

Interface Structure

Event Inputs
This function block has no direct event inputs. Event control is handled via the connected adapters.

Event Outputs
This function block has no direct event outputs. Events are forwarded via the output adapter.

Data Inputs
There are no direct data inputs.

Data Outputs
There are no direct data outputs.

Adapters
The module communicates exclusively via adapter interfaces.

Sockets (Inputs):
- **IN1** (Type: `adapter::types::unidirectional::AS`): First input for addition.
- **IN2** (Type: `adapter::types::unidirectional::AS`): Second input for addition.
- **IN3** (Type: `adapter::types::unidirectional::AS`): Third input for addition.

Plugs (Outputs):
- **OUT** (Type: `adapter::types::unidirectional::AS`): Output providing the result of the addition (`IN1 + IN2 + IN3`).

Functionality
The function block `AS_ADD_3` performs a classic addition:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.70}$$

Since the block is defined as a generic function block (`GEN_AS_ADD`), it can process various numeric data types (e.g., `INT`, `REAL`, `DINT`) depending on the specific instantiation and the data type used in the adapter.

As soon as new values and their corresponding events are signaled at the input adapters (`IN1`, `IN2`, `IN3`), the function block calculates the sum and triggers the corresponding output event at the `OUT` adapter to pass the new value on to subsequent function blocks.

Technical Features
- **Generic Type (Generic FB):** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AS_ADD'`. This allows for flexible adaptation to different data types without having to create a separate function block for each type.
- **Encapsulation via Adapters:** By using unidirectional adapters (`AS`), data and the associated trigger events are encapsulated in a single connection. This ensures a clean and uncluttered user interface design in the 4diac IDE.

State Overview
The function block operates in an event-driven manner and does not have a complex internal state diagram (ECC). Its execution is triggered by the adapter events received at the sockets:

1. **Waiting:** The function block waits for an update event at the adapters `IN1`, `IN2`, or `IN3`.
2. **Calculation:** Upon receiving an event, the current values of the three inputs are added together.
3. **Output:** The result is created in the plug `OUT`, and an update event is sent via the adapter.

Application Scenarios
- **Measurement Summing:** Addition of three analog sensor values (e.g., three flow meters to calculate a total flow rate).

- **Setpoint Offset Calculation:** Calculation of an overall setpoint based on a base setpoint, a user offset, and a correction value from a higher-level controller.
- **Signal Source Combination:** Combining three partial signals in process automation using standardized adapter structures.
-

Comparison with Similar Components

- **Standard ADD Components (IEC 61131-3):** Classic ADD components use dedicated data pins (e.g., `IN1`, `IN2`) and separate event pins (`REQ`, `CNF`). `AS_ADD_3` minimizes the wiring effort in the FBD (Function Block Diagram) by using adapters.
- **AS_ADD_2 (Two-Way Adder):** Compared to a variant with only two inputs, the `AS_ADD_3` saves an entire component and the associated intermediate wiring when adding three values.

Change Detection

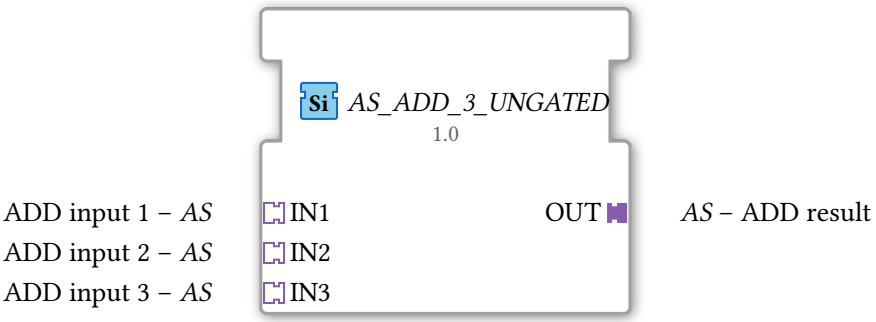
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AS_ADD_3` is a highly efficient, modular component for adding three values. Thanks to the consistent use of unidirectional adapters, it integrates perfectly into modern, service-oriented, and well-structured IEC 61499 control applications.

AS_ADD_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



Introduction

The function block `AS_ADD_3_UNGATED` is a generic function block (FB) for IEC 61499, designed to perform arithmetic addition of three input values. A key feature of this block is that it has no direct data or event inputs/outputs, but relies entirely on the use of unidirectional adapters. This significantly simplifies modeling and wiring in more complex systems.

Interface Structure

Event Inputs

This function block has no direct event inputs. Event control is handled via the connected adapters.

Event Outputs

This function block has no direct event outputs. Events are forwarded via the output adapter.

Data Inputs

There are no direct data inputs.

Data Outputs

There are no direct data outputs.

Adapters

The module communicates exclusively via adapter interfaces.

Sockets (Inputs):

- **IN1** (Type: `adapter::types::unidirectional::AS`): First input for addition.
- **IN2** (Type: `adapter::types::unidirectional::AS`): Second input for addition.
- **IN3** (Type: `adapter::types::unidirectional::AS`): Third input for addition.

Plugs (Outputs):

- **OUT** (Type: `adapter::types::unidirectional::AS`): Output providing the result of the addition (`IN1 + IN2 + IN3`).

Functionality

The function block `AS_ADD_3_UNGATED` performs a classic addition:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.71}$$

Since the block is defined as a generic function block (`GEN_AS_ADD`), it can process various numeric data types (e.g., `INT`, `REAL`, `DINT`) depending on the specific instantiation and the data type used in the adapter.

As soon as new values and their corresponding events are signaled at the input adapters (`IN1`, `IN2`, `IN3`), the function block calculates the sum and triggers the corresponding output event at the `OUT` adapter to pass the new value on to subsequent function blocks.

Technical Features

- **Generic Type (Generic FB):** The function block uses the attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AS_ADD'`. This allows for flexible adaptation to different data types without having to create a separate function block for each type.
- **Encapsulation via Adapters:** By using unidirectional adapters (`AS`), data and the associated trigger events are encapsulated in a single connection. This ensures a clean and uncluttered user interface design in the 4diac IDE.

State Overview

The function block operates in an event-driven manner and does not have a complex internal state diagram (ECC). Its execution is triggered by the adapter events received at the sockets:

1. **Waiting:** The function block waits for an update event at the adapters `IN1`, `IN2`, or `IN3`.
2. **Calculation:** Upon receiving an event, the current values of the three inputs are added together.
3. **Output:** The result is created in the plug `OUT`, and an update event is sent via the adapter.

Application Scenarios

- **Measurement Summing:** Addition of three analog sensor values (e.g., three flow meters to calculate a total flow rate).
- **Setpoint Offset Calculation:** Calculation of an overall setpoint based on a base setpoint, a user offset, and a correction value from a higher-level controller.
- **Signal Source Combination:** Combining three partial signals in process automation using standardized adapter structures.

•

Comparison with Similar Components

- **Standard ADD Components (IEC 61131-3):** Classic ADD components use dedicated data pins (e.g., `IN1`, `IN2`) and separate event pins (`REQ`, `CNF`). `AS_ADD_3_UNGATED` minimizes the wiring effort in the FBD (Function Block Diagram) by using adapters.
- **AS_ADD_2 (Two-Way Adder):** Compared to a variant with only two inputs, the `AS_ADD_3_UNGATED` saves an entire component and the associated intermediate wiring when adding three values.

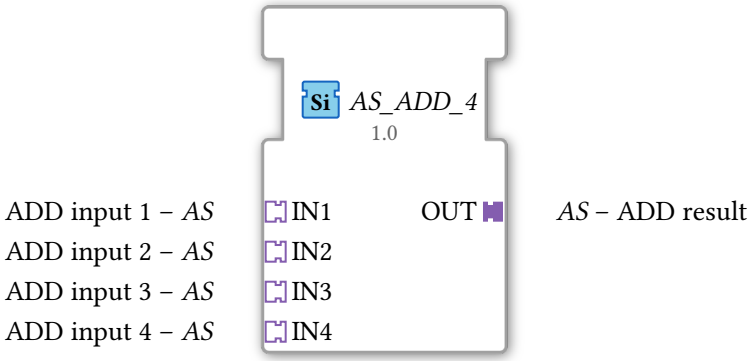
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AS_ADD_3_UNGATED` is a highly efficient, modular component for adding three values. Thanks to the consistent use of unidirectional adapters, it integrates perfectly into modern, service-oriented, and well-structured IEC 61499 control applications.

AS_ADD_4



(Image of function block not available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AS_ADD_4` is a generic function block (FB) designed to perform an arithmetic addition of four input values. The unique feature of this block lies in the use of unidirectional adapters (`AS`) for the inputs and outputs, enabling structured and clean encapsulation of data and associated events in IEC 61499 applications.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is implicit via the adapter interfaces.

Event Outputs

There are no direct event outputs. Event forwarding is implicit via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AS`): First input value (addend 1) for addition.
- **IN2** (Type: `adapter::types::unidirectional::AS`): Second input value (addend 2) for addition.

- **IN3** (Type: `adapter::types::unidirectional::AS`): Third input value (addend 3) for addition.
- **IN4** (Type: `adapter::types::unidirectional::AS`): Fourth input value (addend 4) for the addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AS`): Output adapter for displaying the calculated addition result.

Functionality

This function block performs a mathematical addition of the values received via the four input adapters (`IN1` to `IN4`). The mathematical formula is:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.72}$$

The calculation is typically triggered as soon as a corresponding update event arrives at one or more of the input adapters. After successful calculation, the result, along with a trigger event, is forwarded via the output adapter `OUT`.

Technical Features

- **Generic Nature:** The function block is declared as a generic function block (`GEN_AS_ADD`). This allows for flexible handling of different data types (e.g., `INT`, `REAL`, `DINT`), depending on how the underlying adapter types are defined.
- **Adapter Coupling:** By using unidirectional adapters, the risk of incorrect pin wiring in the 4diac IDE is reduced, as data and events are transmitted together in a single connection.
-

State Overview

Since this is an algorithmic function block for calculations, the FB operates statelessly or event-driven:

1. **Waiting:** The function block waits for an update event at input adapters `IN1` to `IN4`.
2. **Calculation:** Upon arrival of an event, the current values of the inputs are added together.
3. **Output:** The new total value is written to adapter `OUT`, and the corresponding output event is triggered.

–

Application Scenarios

- **Measurement Value Summation:** Summarizing four analog sensor values (e.g., flow rates or power) into a single total value.
- **Setpoint Generation:** Adding base setpoints with various correction or offset values in a cascade control system.
- **Structured Signal Processing:** Used in complex control architectures where a strict separation of logic and data flow is required through adapter structures.

Comparison with Similar Components

Compared to a classic, standard `ADD` component (according to IEC 61131-3):

- **Advantage:** Reduced wiring effort in the graphical user interface, as separate event and data lines do not need to be drawn.
- **Disadvantage:** Requires the definition and use of the specific adapter type `adapter::types::unidirectional::AS`. Direct connection of standard data types (such as simple `REAL` or `INT` variables) without an adapter is not possible.

Change Detection

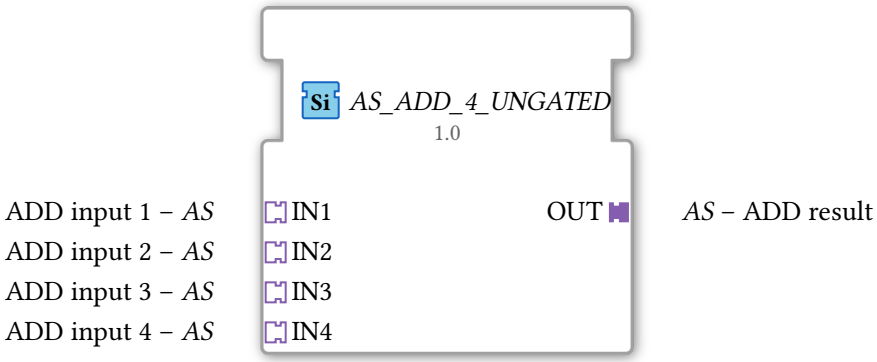
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AS_ADD_4 is a specialized, highly efficient function block for modern, adapter-based programming in the 4diac IDE. It is ideally suited for applications that require a high degree of modularity and clarity in the signal flow and need to add four summands cleanly and efficiently.

AS_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AS_ADD_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(Image of function block not available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block AS_ADD_4_UNGATED is a generic function block (FB) designed to perform an arithmetic addition of four input values. The unique feature of this block lies in the use of unidirectional adapters (AS) for the inputs and outputs, enabling structured and clean encapsulation of data and associated events in IEC 61499 applications.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is implicit via the adapter interfaces.

Event Outputs

There are no direct event outputs. Event forwarding is implicit via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: adapter::types::unidirectional::AS): First input value (addend 1) for addition.
- **IN2** (Type: adapter::types::unidirectional::AS): Second input value (addend 2) for addition.
- **IN3** (Type: adapter::types::unidirectional::AS): Third input value (addend 3) for addition.
- **IN4** (Type: adapter::types::unidirectional::AS): Fourth input value (addend 4) for the addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AS`): Output adapter for displaying the calculated addition result.

Functionality

This function block performs a mathematical addition of the values received via the four input adapters (`IN1` to `IN4`). The mathematical formula is:

$$\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4} \tag{2.73}$$

The calculation is typically triggered as soon as a corresponding update event arrives at one or more of the input adapters. After successful calculation, the result, along with a trigger event, is forwarded via the output adapter `OUT`.

Technical Features

- **Generic Nature:** The function block is declared as a generic function block (`GEN_AS_ADD`). This allows for flexible handling of different data types (e.g., `INT`, `REAL`, `DINT`), depending on how the underlying adapter types are defined.
- **Adapter Coupling:** By using unidirectional adapters, the risk of incorrect pin wiring in the 4diac IDE is reduced, as data and events are transmitted together in a single connection.
-

State Overview

Since this is an algorithmic function block for calculations, the FB operates statelessly or event-driven:

1. **Waiting:** The function block waits for an update event at input adapters `IN1` to `IN4`.
2. **Calculation:** Upon arrival of an event, the current values of the inputs are added together.
3. **Output:** The new total value is written to adapter `OUT`, and the corresponding output event is triggered.

–

Application Scenarios

- **Measurement Value Summation:** Summarizing four analog sensor values (e.g., flow rates or power) into a single total value.
- **Setpoint Generation:** Adding base setpoints with various correction or offset values in a cascade control system.
- **Structured Signal Processing:** Used in complex control architectures where a strict separation of logic and data flow is required through adapter structures.

Comparison with Similar Components

Compared to a classic, standard `ADD` component (according to IEC 61131-3):

- **Advantage:** Reduced wiring effort in the graphical user interface, as separate event and data lines do not need to be drawn.
- **Disadvantage:** Requires the definition and use of the specific adapter type `adapter::types::unidirectional::AS`. Direct connection of standard data types (such as simple `REAL` or `INT` variables) without an adapter is not possible.

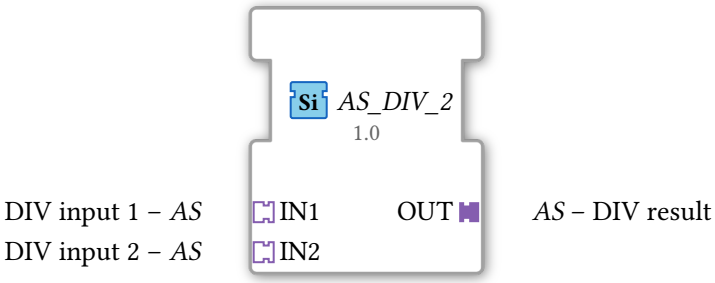
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AS_ADD_4_UNGATED` is a specialized, highly efficient function block for modern, adapter-based programming in the 4diac IDE. It is ideally suited for applications that require a high degree of modularity and clarity in the signal flow and need to add four summands cleanly and efficiently.

AS_DIV_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) `AS_DIV_2` is a generic block for performing arithmetic division within the context of the IEC 61499 architecture. It utilizes an adapter-based data transmission concept and enables clean and structured signal processing by receiving input data via standardized adapters, calculating the division, and passing the result to an output adapter.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Control and triggering are implicit via the connected adapters.

Event Outputs

This function block does not have direct event outputs.

Data Inputs

This function block does not have direct data inputs.

Data Outputs

This function block does not have direct data outputs.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AS`):
- Represents the first input value of the division (dividend).
- **IN2** (Type: `adapter::types::unidirectional::AS`):
- Represents the second input value of the division (divisor).

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AS`):
- Outputs the result of the division (quotient).

Functionality

The function block `AS_DIV_2` performs a mathematical division of the passed input values. The basic formula is:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.74}$$

As soon as the values at the input adapters `IN1` or `IN2` change, or a corresponding event is transmitted via the adapters, the division is performed and the result is provided via the output adapter `OUT`.

Technical Features

- **Generic Block:** The attribute `GenericClassName = 'GEN_AS_DIV'` defines the block as a generic type. This allows for high flexibility when processing different numeric data types.
- **Adapter Encapsulation:** Instead of using loose data and event lines, this block exclusively uses unidirectional adapters of type `AS`. This ensures a clean and clear application diagram in the 4diac IDE.
- **Division by Zero:** During implementation and use, it is important to ensure that the divisor (`IN2`) is not equal to zero to avoid a runtime error (division by zero).

State Overview

Since it is a purely mathematical/arithmetic function block, the `AS_DIV_2` does not have a complex internal state machine (ECC). The output values are calculated directly based on the input signals applied to the adapters.

–

Application Scenarios

- **Measurement Scaling:** Conversion and normalization of analog sensor values (e.g., dividing a raw value to convert it into a physical unit).
- **Percentage Calculations:** Determination of ratios or percentages in industrial processes.
- **Signal Preprocessing:** Mathematical reduction of process data directly at the field level before forwarding to higher-level systems.

Comparison with Similar Components

Compared to classic IEC 61131-3 `DIV` components, which use direct data pins, the `AS_DIV_2` offers significantly improved encapsulation through the use of adapters. The number of connection lines in the CFC (Continuous Function Chart) is significantly reduced, which improves maintainability, especially for large and complex control programs.

–

Change Detection

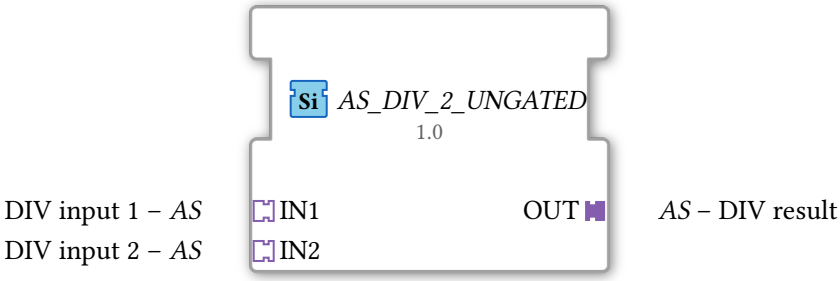
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AS_DIV_2` is a robust and versatile component for arithmetic division in distributed control systems. Its consistent use of the adapter concept supports modern, modular software design according to the IEC 61499 standard.

AS_DIV_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_DIV_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) `AS_DIV_2_UNGATED` is a generic block for performing arithmetic division within the context of the IEC 61499 architecture. It utilizes an adapter-based data transmission concept and enables clean and structured signal processing by receiving input data via standardized adapters, calculating the division, and passing the result to an output adapter.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Control and triggering are implicit via the connected adapters.

Event Outputs

This function block does not have direct event outputs.

Data Inputs

This function block does not have direct data inputs.

Data Outputs

This function block does not have direct data outputs.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AS`):
- Represents the first input value of the division (dividend).
- **IN2** (Type: `adapter::types::unidirectional::AS`):
- Represents the second input value of the division (divisor).

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AS`):
- Outputs the result of the division (quotient).

Functionality

The function block `AS_DIV_2_UNGATED` performs a mathematical division of the passed input values. The basic formula is:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.75}$$

As soon as the values at the input adapters `IN1` or `IN2` change, or a corresponding event is transmitted via the adapters, the division is performed and the result is provided via the output adapter `OUT`.

Technical Features

- **Generic Block:** The attribute `GenericClassName = 'GEN_AS_DIV'` defines the block as a generic type. This allows for high flexibility when processing different numeric data types.
- **Adapter Encapsulation:** Instead of using loose data and event lines, this block exclusively uses unidirectional adapters of type `AS`. This ensures a clean and clear application diagram in the 4diac IDE.
- **Division by Zero:** During implementation and use, it is important to ensure that the divisor (`IN2`) is not equal to zero to avoid a runtime error (division by zero).

State Overview

Since it is a purely mathematical/arithmetic function block, the `AS_DIV_2_UNGATED` does not have a complex internal state machine (ECC). The output values are calculated directly based on the input signals applied to the adapters.

–

Application Scenarios

- **Measurement Scaling:** Conversion and normalization of analog sensor values (e.g., dividing a raw value to convert it into a physical unit).
- **Percentage Calculations:** Determination of ratios or percentages in industrial processes.
- **Signal Preprocessing:** Mathematical reduction of process data directly at the field level before forwarding to higher-level systems.

Comparison with Similar Components

Compared to classic IEC 61131-3 `DIV` components, which use direct data pins, the `AS_DIV_2_UNGATED` offers significantly improved encapsulation through the use of adapters. The number of connection lines in the CFC (Continuous Function Chart) is significantly reduced, which improves maintainability, especially for large and complex control programs.

–

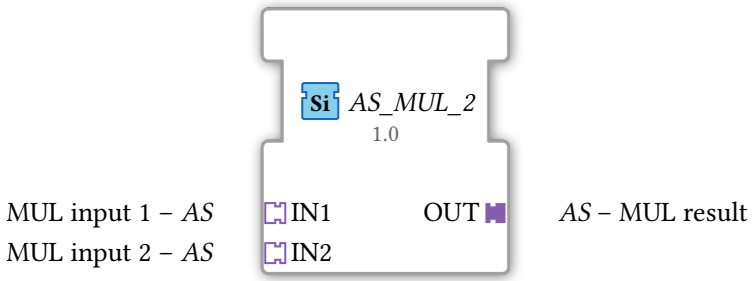
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AS_DIV_2_UNGATED` is a robust and versatile component for arithmetic division in distributed control systems. Its consistent use of the adapter concept supports modern, modular software design according to the IEC 61499 standard.

AS_MUL_2



– ◦ – ◦ – ◦

Introduction

The function block `AS_MUL_2` is a generic, arithmetic multiplication block for IEC 61499 applications in the 4diac IDE. It is used to multiply two input values. The special feature of this block lies in the use of

adapter interfaces instead of classic, discrete event and data channels. This enables structured and clear signal transmission.

Interface Structure

The block does not have any directly accessible standard event or data pins, but rather encapsulates these completely in adapters.

Event Inputs

No direct event inputs available (events are received via the adapters).

Event Outputs

No direct event outputs available (events are sent via the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

- **Sockets (Input Interfaces):**
- `IN1` (Type: `adapter::types::unidirectional::AS`): The first input (multiplicand) for calculation.
- `IN2` (Type: `adapter::types::unidirectional::AS`): The second input (multiplier) for calculation.
- **Plugs (Output Interfaces):**
- `OUT` (Type: `adapter::types::unidirectional::AS`): The result of the multiplication ($OUT = IN1 \times IN2$).

Functionality

As soon as data and a triggering event are received via the input adapters `IN1` and `IN2`, the function block performs the multiplication. The mathematical product of the two input values is calculated and, together with the corresponding trigger event, is made available for further use in the system via the output adapter `OUT`.

Since the function block is designed generically (`GenericClassName = 'GEN_AS_MUL'`), it adapts flexibly to the underlying data types of the connected adapters (e.g., `INT`, `REAL`, `DINT`).

Technical Features

- **Generic Function Block:** By defining it as `GEN_AS_MUL`, the function block is not tied to a specific data type.
- **Unidirectional Adapter Coupling:** Using the adapter type `adapter::types::unidirectional::AS` ensures a clear, directed flow of data and signals, thus preventing feedback loops.
- **Reduced Editor Complexity:** Encapsulating data and events in adapters minimizes the visual "spaghetti code" problem (too many connecting lines) in the 4diac IDE.

State Overview

The function block behaves purely event-driven:

1. **Waiting State:** The function block waits for an event at sockets `IN1` or `IN2`.
2. **Calculation:** Upon receipt of an event, the current values from `IN1` and `IN2` are read and multiplied.
3. **Output:** The result is applied to `OUT`, and an output event is emitted via plug `OUT`.

Application Scenarios

- **Signal Scaling:** Multiplication of analog sensor values by a scaling factor supplied via an adapter.
- **Modular Calculations:** Used in more complex mathematical computing networks where structured data streams need to be transported via adapter buses.
- **Power Calculation:** Multiplication of current and voltage values to determine electrical power in real-time systems.

Comparison with Similar Components

Compared to a standard IEC 61131-3 / IEC 61499 `MUL` component, which uses individual pins for `REQ`, `CNF`, `IN1`, `IN2`, and `OUT`, `AS_MUL_2` combines these signals into a single logical channel per pin. This increases reusability and ensures a cleaner application design, but requires the definition and use of appropriate adapter types in the project.

Change Detection

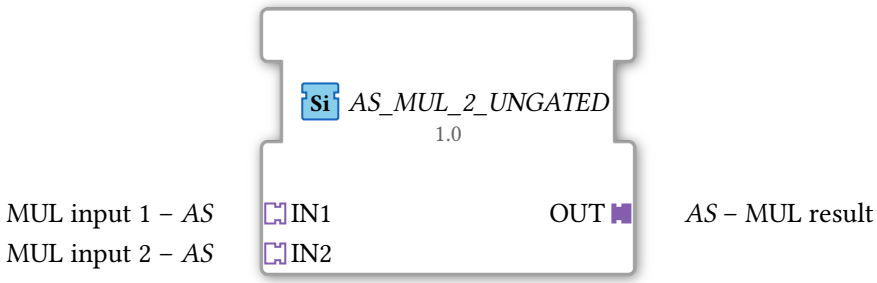
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AS_MUL_2` is a modern and flexible function block for arithmetic operations in IEC 61499 control programs. Thanks to its generic structure and consistent use of adapters, it is ideally suited for clean, modular, and easily maintainable software architectures in industrial automation.

AS_MUL_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_MUL_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AS_MUL_2_UNGATED` is a generic, arithmetic multiplication block for IEC 61499 applications in the 4diac IDE. It is used to multiply two input values. The special feature of this block lies in the use of adapter interfaces instead of classic, discrete event and data channels. This enables structured and clear signal transmission.

Interface Structure

The block does not have any directly accessible standard event or data pins, but rather encapsulates these completely in adapters.

Event Inputs

No direct event inputs available (events are received via the adapters).

Event Outputs

No direct event outputs available (events are sent via the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

- **Sockets (Input Interfaces):**
- `IN1` (Type: `adapter::types::unidirectional::AS`): The first input (multiplicand) for calculation.
- `IN2` (Type: `adapter::types::unidirectional::AS`): The second input (multiplier) for calculation.
- **Plugs (Output Interfaces):**
- `OUT` (Type: `adapter::types::unidirectional::AS`): The result of the multiplication ($OUT = IN1 \times IN2$).

Functionality

As soon as data and a triggering event are received via the input adapters `IN1` and `IN2`, the function block performs the multiplication. The mathematical product of the two input values is calculated and, together with the corresponding trigger event, is made available for further use in the system via the output adapter `OUT`.

Since the function block is designed generically (`GenericClassName = 'GEN_AS_MUL'`), it adapts flexibly to the underlying data types of the connected adapters (e.g., `INT`, `REAL`, `DINT`).

Technical Features

- **Generic Function Block:** By defining it as `GEN_AS_MUL`, the function block is not tied to a specific data type.
- **Unidirectional Adapter Coupling:** Using the adapter type `adapter::types::unidirectional::AS` ensures a clear, directed flow of data and signals, thus preventing feedback loops.
- **Reduced Editor Complexity:** Encapsulating data and events in adapters minimizes the visual "spaghetti code" problem (too many connecting lines) in the 4diac IDE.

State Overview

The function block behaves purely event-driven:

1. **Waiting State:** The function block waits for an event at sockets `IN1` or `IN2`.
2. **Calculation:** Upon receipt of an event, the current values from `IN1` and `IN2` are read and multiplied.
3. **Output:** The result is applied to `OUT`, and an output event is emitted via plug `OUT`.

Application Scenarios

- **Signal Scaling:** Multiplication of analog sensor values by a scaling factor supplied via an adapter.
- **Modular Calculations:** Used in more complex mathematical computing networks where structured data streams need to be transported via adapter buses.
- **Power Calculation:** Multiplication of current and voltage values to determine electrical power in real-time systems.

Comparison with Similar Components

Compared to a standard IEC 61131-3 / IEC 61499 `MUL` component, which uses individual pins for `REQ`, `CNF`, `IN1`, `IN2`, and `OUT`, `AS_MUL_2_UNGATED` combines these signals into a single logical channel per pin. This increases reusability and ensures a cleaner application design, but requires the definition and use of appropriate adapter types in the project.

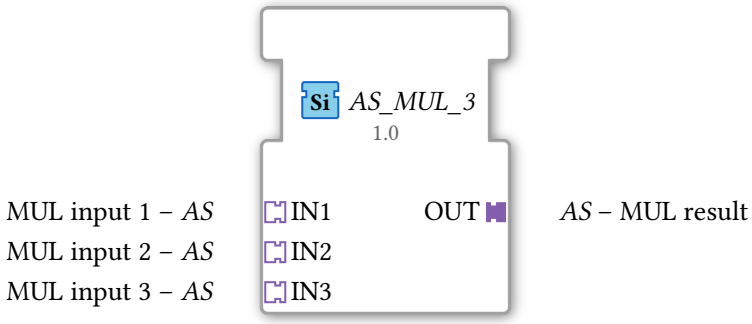
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AS_MUL_2_UNGATED` is a modern and flexible function block for arithmetic operations in IEC 61499 control programs. Thanks to its generic structure and consistent use of adapters, it is ideally suited for clean, modular, and easily maintainable software architectures in industrial automation.

AS_MUL_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AS_MUL_3` is a generic function block (FB) for the 4diac IDE, used to perform arithmetic multiplication with three input values. It is based on the use of unidirectional adapters (`adapter::types::unidirectional::AS`), which enables clean structuring and encapsulation of data and control flows in IEC 61499 applications.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Control and event processing are handled implicitly via the adapter interfaces.

Event Outputs

This function block does not have direct event outputs. Event forwarding is implicit via the adapter interfaces.

Data Inputs

This function block has no direct data inputs. Data is transferred via the adapter interface sockets.

Data Outputs

This function block has no direct data outputs. Data output is via the adapter interface plug.

Adapters

All communication for this function block is handled via adapters:

- **Sockets (Input Adapters):**
- **IN1** (Type: `adapter::types::unidirectional::AS`): The first input value (multiplicand 1) for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::AS`): The second input value (multiplicand 2) for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::AS`): The third input value (multiplicand 3) for the multiplication.
- **Plugs (Output Adapters):**
- **OUT** (Type: `adapter::types::unidirectional::AS`): The output adapter that provides the result of the multiplication.

Functionality

The function block `AS_MUL_3` multiplies the values received via the input adapters `IN1`, `IN2`, and `IN3`. As soon as values at the inputs change or a corresponding trigger event is received via the adapters, the calculation is performed and the result is passed to the output adapter `OUT`.

The mathematical formula is:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \tag{2.76}$$

Technical Features

- **Generic Type:** The component is defined as a generic type (`GEN_AS_MUL`). This allows for flexible adaptation to different numeric data types (e.g., `INT`, `REAL`, `LREAL`) supported by the underlying adapter type.
- **Adapter-Based Architecture:** By using unidirectional adapters (`AS`), the cabling effort in the 4diac IDE is drastically reduced, as event and data lines are bundled into a single connection.
- **Adapter-Based Architecture:** —

State Overview

This function block behaves like a classic stateless computation block. Execution is triggered by incoming events on the sockets (`IN1`, `IN2`, `IN3`). After successful calculation of the product, the output event is emitted directly via the plug `OUT` along with the calculated value.

–

Application Scenarios

- **Sensor Scaling and Calibration:** Calculation of corrected measured values where a raw value must be multiplied by two different correction factors.
- **Volume Calculations:** Multiplication of three dimensions (length × width × height) to determine a volume in process engineering.
- **Multi-Stage Gain Control:** Cascaded signal amplification in control engineering.

Comparison with Similar Function Blocks

Compared to standard multiplication function blocks (such as `MUL`), which use separate event and data lines (e.g., `REQ`/`CNF` and standard data types), `AS_MUL_3`, thanks to its encapsulation in adapters, offers a significantly cleaner visual representation in the 4diac IDE's Application Editor. Manually linking "With" connections between events and data is no longer necessary.

–

Change Detection

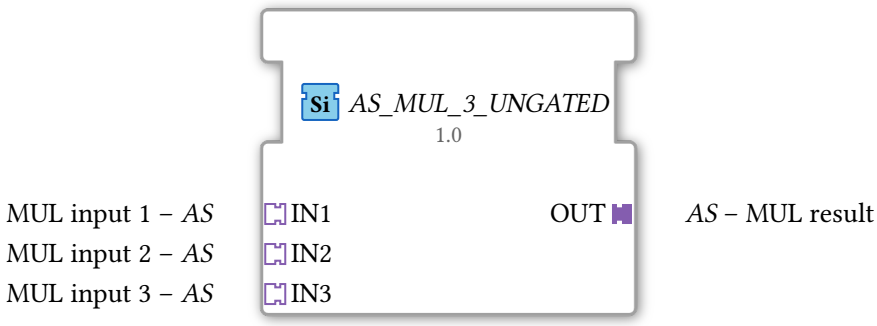
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

`AS_MUL_3` is an efficient, modular, and reusable function block for arithmetic triple multiplication. It is ideally suited for modern, adapter-based software architectures within IEC 61499 and contributes to the clarity of complex control applications.

AS_MUL_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_MUL_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AS_MUL_3_UNGATED` is a generic function block (FB) for the 4diac IDE, used to perform arithmetic multiplication with three input values. It is based on the use of unidirectional adapters (`adapter::types::unidirectional::AS`), which enables clean structuring and encapsulation of data and control flows in IEC 61499 applications.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Control and event processing are handled implicitly via the adapter interfaces.

Event Outputs

This function block does not have direct event outputs. Event forwarding is implicit via the adapter interfaces.

Data Inputs

This function block has no direct data inputs. Data is transferred via the adapter interface sockets.

Data Outputs

This function block has no direct data outputs. Data output is via the adapter interface plug.

Adapters

All communication for this function block is handled via adapters:

- **Sockets (Input Adapters):**
- **IN1** (Type: `adapter::types::unidirectional::AS`): The first input value (multiplicand 1) for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::AS`): The second input value (multiplicand 2) for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::AS`): The third input value (multiplicand 3) for the multiplication.
- **Plugs (Output Adapters):**
- **OUT** (Type: `adapter::types::unidirectional::AS`): The output adapter that provides the result of the multiplication.

Functionality

The function block `AS_MUL_3_UNGATED` multiplies the values received via the input adapters `IN1`, `IN2`, and `IN3`. As soon as values at the inputs change or a corresponding trigger event is received via the adapters, the calculation is performed and the result is passed to the output adapter `OUT`.

The mathematical formula is:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.77}$$

Technical Features

- **Generic Type:** The component is defined as a generic type (`GEN_AS_MUL`). This allows for flexible adaptation to different numeric data types (e.g., `INT`, `REAL`, `LREAL`) supported by the underlying adapter type.
- **Adapter-Based Architecture:** By using unidirectional adapters (`AS`), the cabling effort in the 4diac IDE is drastically reduced, as event and data lines are bundled into a single connection.
- **Adapter-Based Architecture:** –

State Overview

This function block behaves like a classic stateless computation block. Execution is triggered by incoming events on the sockets (`IN1`, `IN2`, `IN3`). After successful calculation of the product, the output event is emitted directly via the plug `OUT` along with the calculated value.

–

Application Scenarios

- **Sensor Scaling and Calibration:** Calculation of corrected measured values where a raw value must be multiplied by two different correction factors.
- **Volume Calculations:** Multiplication of three dimensions (length × width × height) to determine a volume in process engineering.
- **Multi-Stage Gain Control:** Cascaded signal amplification in control engineering.

Comparison with Similar Function Blocks

Compared to standard multiplication function blocks (such as `MUL`), which use separate event and data lines (e.g., `REQ`/`CNF` and standard data types), `AS_MUL_3_UNGATED`, thanks to its encapsulation in adapters, offers a significantly cleaner visual representation in the 4diac IDE's Application Editor. Manually linking "With" connections between events and data is no longer necessary.

–

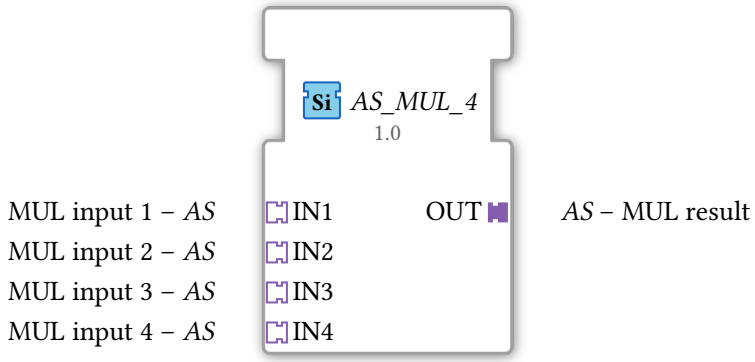
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

`AS_MUL_3_UNGATED` is an efficient, modular, and reusable function block for arithmetic triple multiplication. It is ideally suited for modern, adapter-based software architectures within IEC 61499 and contributes to the clarity of complex control applications.

AS_MUL_4



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AS_MUL_4` is a generic function block (Generic FB) for IEC 61499, designed for the arithmetic multiplication of four input values. Instead of traditional data and event pins, this block uses adapter connections (`AS` - unidirectional) to transmit data and the associated execution control in encapsulated form. This results in a cleaner design in the 4diac IDE, as the number of visible connection lines is drastically reduced.

Interface Structure

Event Inputs

This function block does not have direct, independent event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

This function block does not have direct, independent event outputs. Event control for subsequent function blocks is transmitted via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

This function block is based entirely on adapter-based communication.

Sockets (Inputs)

- **IN1** (Type: `adapter::types::unidirectional::AS`): First multiplicand (Input 1).
- **IN2** (Type: `adapter::types::unidirectional::AS`): Second multiplicand (Input 2).
- **IN3** (Type: `adapter::types::unidirectional::AS`): Third multiplicand (Input 3).
- **IN4** (Type: `adapter::types::unidirectional::AS`): Fourth multiplicand (Input 4).

Plugs (Connectors / Outputs)

- **OUT** (Type: `adapter::types::unidirectional::AS`): Result of the multiplication (`IN1 * IN2 * IN3 * IN4`).

Functionality

As soon as new values or a corresponding trigger event are received via the adapter sockets, the function block performs the arithmetic multiplication of the four input values:

$$ OUT = IN1 × IN2 × IN3 × IN4

The result and the corresponding output event are then passed on to the subsequent logic elements via the adapter plug `OUT`.

Technical Features

- **Generic Character:** By assigning it to the generic class `GEN_AS_MUL`, the function block can flexibly adapt to various numeric data types (e.g., `INT`, `REAL`, `LREAL`), provided the connected adapters of type `AS` use the same data type.
- **Unidirectional Adapters:** Using the type `adapter::types::unidirectional::AS` ensures that the data flow is clear and direct from the inputs to the output.
- **Compact Design:** Encapsulating data and event channels in adapters keeps the application diagram in 4diac clear and concise.
-

State Overview

Since `AS_MUL_4` is a mathematical, stateless function block, no complex internal state diagram (ECC) exists.

- **Ready / Idle:** The block waits for incoming events at sockets `IN1` to `IN4`.
- **Calculation:** Upon receiving a trigger event, the values are updated, multiplied, and directly passed to `OUT`.

–

Application Scenarios

- **Scaling of Measured Values:** Multiplying a sensor value by several calibration and correction factors.
- **Volume Calculations:** Calculating a volume from three dimensions (length × width × height) multiplied by a density factor.
- **Cascaded Gainers:** Calculation of total loop gains in control engineering where four different gain factors need to be multiplied.

Comparison with Similar Components

- **Standard `MUL` component (IEC 61131-3):** This requires separate data lines for each value and at least one trigger event (REQ/CNF). `AS_MUL_4` simplifies this by using adapters, requiring only five connections (4 input adapters, 1 output adapter).

- **AS_MUL_2 / AS_MUL_3:** Offer the same adapter-based functionality but are limited to two or three input values, respectively. AS_MUL_4 eliminates the need for additional intermediate steps and temporary auxiliary variables when combining four factors.
-

Change Detection

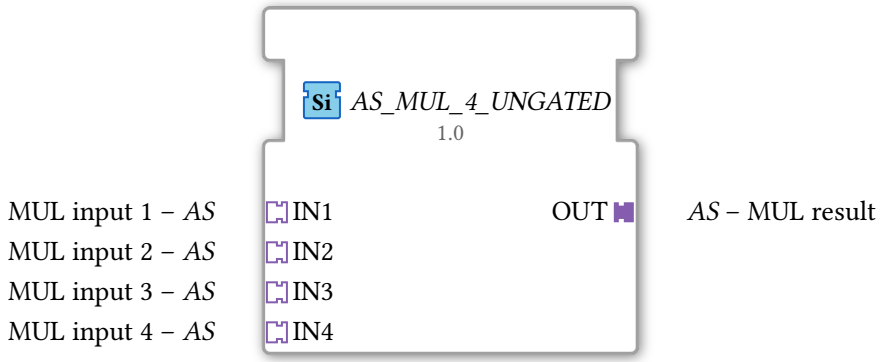
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AS_MUL_4 is a highly efficient, clear, and modern function block for arithmetic calculations in IEC 61499. The consistent use of unidirectional adapters keeps the application design uncluttered while maintaining the full flexibility of a generic data type.

AS_MUL_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AS_MUL_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block AS_MUL_4_UNGATED is a generic function block (Generic FB) for IEC 61499, designed for the arithmetic multiplication of four input values. Instead of traditional data and event pins, this block uses adapter connections (AS - unidirectional) to transmit data and the associated execution control in encapsulated form. This results in a cleaner design in the 4diac IDE, as the number of visible connection lines is drastically reduced.

Interface Structure

Event Inputs

This function block does not have direct, independent event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

This function block does not have direct, independent event outputs. Event control for subsequent function blocks is transmitted via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

This function block is based entirely on adapter-based communication.

Sockets (Inputs)

- **IN1** (Type: `adapter::types::unidirectional::AS`): First multiplicand (Input 1).
- **IN2** (Type: `adapter::types::unidirectional::AS`): Second multiplicand (Input 2).
- **IN3** (Type: `adapter::types::unidirectional::AS`): Third multiplicand (Input 3).
- **IN4** (Type: `adapter::types::unidirectional::AS`): Fourth multiplicand (Input 4).

Plugs (Connectors / Outputs)

- **OUT** (Type: `adapter::types::unidirectional::AS`): Result of the multiplication (`IN1 * IN2 * IN3 * IN4`).

Functionality

As soon as new values or a corresponding trigger event are received via the adapter sockets, the function block performs the arithmetic multiplication of the four input values:

$$ OUT = IN1 × IN2 × IN3 × IN4

The result and the corresponding output event are then passed on to the subsequent logic elements via the adapter plug `OUT`.

Technical Features

- **Generic Character:** By assigning it to the generic class `GEN_AS_MUL`, the function block can flexibly adapt to various numeric data types (e.g., `INT`, `REAL`, `LREAL`), provided the connected adapters of type `AS` use the same data type.
- **Unidirectional Adapters:** Using the type `adapter::types::unidirectional::AS` ensures that the data flow is clear and direct from the inputs to the output.
- **Compact Design:** Encapsulating data and event channels in adapters keeps the application diagram in 4diac clear and concise.

•

State Overview

Since `AS_MUL_4_UNGATED` is a mathematical, stateless function block, no complex internal state diagram (ECC) exists.

- **Ready / Idle:** The block waits for incoming events at sockets `IN1` to `IN4`.
- **Calculation:** Upon receiving a trigger event, the values are updated, multiplied, and directly passed to `OUT`.

–

Application Scenarios

- **Scaling of Measured Values:** Multiplying a sensor value by several calibration and correction factors.
- **Volume Calculations:** Calculating a volume from three dimensions (length × width × height) multiplied by a density factor.
- **Cascaded Gainers:** Calculation of total loop gains in control engineering where four different gain factors need to be multiplied.

Comparison with Similar Components

- **Standard `MUL` component (IEC 61131-3):** This requires separate data lines for each value and at least one trigger event (REQ/CNF). `AS_MUL_4_UNGATED` simplifies this by using adapters, requiring only five connections (4 input adapters, 1 output adapter).

- **AS_MUL_2 / AS_MUL_3:** Offer the same adapter-based functionality but are limited to two or three input values, respectively. AS_MUL_4_UNGATED eliminates the need for additional intermediate steps and temporary auxiliary variables when combining four factors.

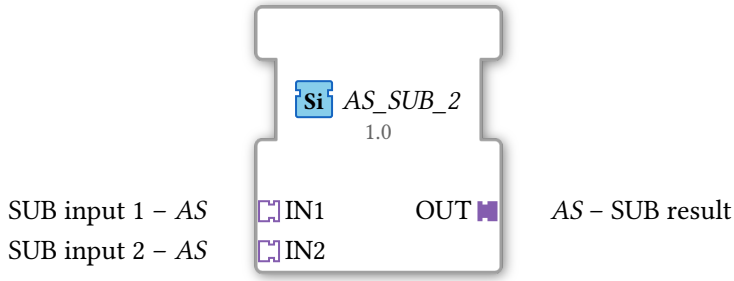
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AS_MUL_4_UNGATED is a highly efficient, clear, and modern function block for arithmetic calculations in IEC 61499. The consistent use of unidirectional adapters keeps the application design uncluttered while maintaining the full flexibility of a generic data type.

AS_SUB_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block AS_SUB_2 is a generic function block (Generic FB) for the 4diac-ide development environment. It performs an arithmetic subtraction ($OUT = IN1 - IN2$).

Unlike classic arithmetic blocks that work directly with standard data and event interfaces, this block uses unidirectional adapter connections to encapsulate data and control events. This enables structured and clear wiring in complex IEC 61499 applications.

Interface Structure

Event Inputs

The function block does not have direct, independent event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

The function block does not have direct, independent event outputs. Events are forwarded via the output adapter.

Data Inputs

No direct data inputs are defined on the function block. Input data is provided via the adapter interfaces.

Data Outputs

No direct data outputs are defined on the function block. The result is passed via the output adapter.

Adapters

All communication of the function block is implemented via adapters:

- **Sockets (Input Adapters):**
- IN1 (Type: adapter::types::unidirectional::AS): The first input (minuend) for subtraction.
- IN2 (Type: adapter::types::unidirectional::AS): The second input (subtrahend) for subtraction.
- **Plugs (Output Adapters):**

- `OUT` (Type: `adapter::types::unidirectional::AS`): The result (difference) of the arithmetic subtraction.

Functionality

As soon as valid values and the corresponding trigger events are present at the input adapters `IN1` and `IN2`, the function block performs the calculation.

The mathematical operation is:

$$\text{Result} = \text{Value of } \ IN1 - \text{Value of } \ IN2 \tag{2.78}$$

The result of the calculation and the corresponding output event are then forwarded to subsequent function blocks via the output adapter `OUT`. Since it is a generic function block (`GenericClassName="GEN_AS_SUB"`), it is flexible with respect to the underlying data types, provided these are supported by the adapter type.

Technical Features

- **Generic Behavior:** By declaring it as a generic function block, it can be used for various numeric data types (e.g., INT, REAL, LREAL), depending on the definition of the adapter used, `adapter::types::unidirectional::AS`.
- **Compact Structure:** The use of adapters significantly reduces visual complexity in the 4diac Application Editor, as event and data lines are bundled into a single adapter connection.

Unidirectional Data Flow: The adapters used have a strictly unidirectional signal flow (from the sockets to the plug).

State Overview

The execution logic is typically triggered by events on the input adapters:

1. **Idle State:** The block waits for an event on `IN1` or `IN2`.
2. **Calculation:** Upon receiving a trigger, the data is read from the adapters, and the difference is calculated.
3. **Dispatch:** The result is written to the output adapter, and the output event on `OUT` is triggered. The module returns to standby mode.

Application Scenarios

- **Measurement Correction:** Subtraction of zero-point errors or offsets in analog sensor values.
- **Setpoint-Actual Comparison:** Calculation of the control deviation ($e = w - x$) in control loops where setpoint and actual values are transferred via standardized adapter structures.
- **Level and Quantity Calculation:** Determination of differential quantities in process plants.

Comparison with Similar Function Blocks

Compared to a standard subtraction function block (e.g., `SUB` from the IEC 61131-3 library), which uses explicit inputs like `REQ`, `IN1`, and `IN2`, the `AS_SUB_2` offers significantly higher modularity due to its encapsulation in adapters. It is particularly suitable for advanced software architectures in IEC 61499, where data and signal flows are to be combined as logical units (interfaces).

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

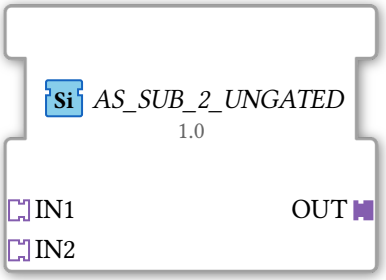
Conclusion

The `AS_SUB_2` is a highly efficient, reusable, and clearly structured function block for implementing subtraction tasks. Its strength lies in the consistent use of adapters, which leads to cleaner software designs and improved maintainability within the 4diac-ide.

AS_SUB_2_UNGATED

i UNGATED variant: This block is the ungated version of AS_SUB_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change.

This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AS_SUB_2_UNGATED` is a generic function block (Generic FB) for the 4diac-ide development environment. It performs an arithmetic subtraction ($OUT = IN1 - IN2$).

Unlike classic arithmetic blocks that work directly with standard data and event interfaces, this block uses unidirectional adapter connections to encapsulate data and control events. This enables structured and clear wiring in complex IEC 61499 applications.

Interface Structure

Event Inputs

The function block does not have direct, independent event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

The function block does not have direct, independent event outputs. Events are forwarded via the output adapter.

Data Inputs

No direct data inputs are defined on the function block. Input data is provided via the adapter interfaces.

Data Outputs

No direct data outputs are defined on the function block. The result is passed via the output adapter.

Adapters

All communication of the function block is implemented via adapters:

- **Sockets (Input Adapters):**
- `IN1` (Type: `adapter::types::unidirectional::AS`): The first input (minuend) for subtraction.
- `IN2` (Type: `adapter::types::unidirectional::AS`): The second input (subtrahend) for subtraction.
- **Plugs (Output Adapters):**
- `OUT` (Type: `adapter::types::unidirectional::AS`): The result (difference) of the arithmetic subtraction.

Functionality

As soon as valid values and the corresponding trigger events are present at the input adapters `IN1` and `IN2`, the function block performs the calculation.

The mathematical operation is:

$$\text{Result} = \text{Value of } \ IN1 - \text{Value of } \ IN2 \tag{2.79}$$

The result of the calculation and the corresponding output event are then forwarded to subsequent function blocks via the output adapter `OUT`. Since it is a generic function block

(`GenericClassName="GEN_AS_SUB"`), it is flexible with respect to the underlying data types, provided these are supported by the adapter type.

Technical Features

- **Generic Behavior:** By declaring it as a generic function block, it can be used for various numeric data types (e.g., INT, REAL, LREAL), depending on the definition of the adapter used, `adapter::types::unidirectional::AS`.
- **Compact Structure:** The use of adapters significantly reduces visual complexity in the 4diac Application Editor, as event and data lines are bundled into a single adapter connection.

Unidirectional Data Flow: The adapters used have a strictly unidirectional signal flow (from the sockets to the plug).

State Overview

The execution logic is typically triggered by events on the input adapters:

1. **Idle State:** The block waits for an event on `IN1` or `IN2`.
2. **Calculation:** Upon receiving a trigger, the data is read from the adapters, and the difference is calculated.
3. **Dispatch:** The result is written to the output adapter, and the output event on `OUT` is triggered. The module returns to standby mode.

Application Scenarios

- **Measurement Correction:** Subtraction of zero-point errors or offsets in analog sensor values.
- **Setpoint-Actual Comparison:** Calculation of the control deviation ($e = w - x$) in control loops where setpoint and actual values are transferred via standardized adapter structures.
- **Level and Quantity Calculation:** Determination of differential quantities in process plants.

Comparison with Similar Function Blocks

Compared to a standard subtraction function block (e.g., `SUB` from the IEC 61131-3 library), which uses explicit inputs like `REQ`, `IN1`, and `IN2`, the `AS_SUB_2_UNGATED` offers significantly higher modularity due to its encapsulation in adapters. It is particularly suitable for advanced software architectures in IEC 61499, where data and signal flows are to be combined as logical units (interfaces).

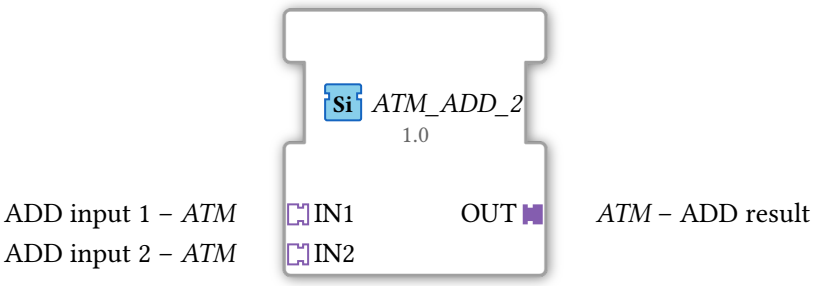
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AS_SUB_2_UNGATED` is a highly efficient, reusable, and clearly structured function block for implementing subtraction tasks. Its strength lies in the consistent use of adapters, which leads to cleaner software designs and improved maintainability within the 4diac-ide.

ATM_ADD_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_ADD_2** is used to calculate the arithmetic addition of two time values (type `TIME`). It is designed as a generic function block and implements the basic arithmetic operation via adapter interfaces. The block is platform-independent and complies with the IEC 61499 standard.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Name	Direction	Type	Comment
`IN1`	Socket	`adapter::types::unidirectional::ATM`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::ATM`	ADD input 2
`OUT`	Plug	`adapter::types::unidirectional::ATM`	ADD result

The adapters are of type `unidirectional::ATM` and enable type-safe connections with other function blocks that support the same adapter type.

Functionality

The function block sums the time values received via adapters `IN1` and `IN2` and provides the result at adapter `OUT`. The starting point of the summation is the identity value `TIME#0s`, so that with only one input actually connected, its value is passed through unchanged. The function block operates generically – the calculation is re-executed on every event at one of the input adapters.

Technical Features

- **Generic Structure** – The function block uses a generic class name (`GEN_ATM_ADD`) at runtime, defined by the attribute `eclipse4diac::core::GenericClassName`. The same class covers the arities `ATM_ADD_2`, `ATM_ADD_3`, and `ATM_ADD_4` via the GenericClassName mechanism.
- **Adapter-Based** – Instead of individual data inputs and outputs, all signals are routed via unidirectional adapters.
- **Package Information** – The function block is organized in the package `adapter::iec61131::arithmetic`.
- **No State Logic** – The addition is stateless; there is no internal state machine.

State Overview

The function block does not have a state machine. The calculation is purely event-driven – on any event at one of the input adapters, the sum is recalculated.

Application-Specific Scenarios

- **Time accumulation** – Summing multiple delay or runtime values into a total duration.
- **Control Engineering** – Combining two dynamically determined time values, e.g., base time plus a surcharge.
- **Generic Library Function Blocks** – Use as the adapter-based counterpart to the classic `ADD` function, specialized for `TIME`.

Comparison with Similar Function Blocks

Compared to `AR_ADD_2` (addition of two `REAL` values via adapters), `ATM_ADD_2` is specialized for the `TIME` data type. Unlike `AR_MULTIME/ATM_AR_MULTIME`, which multiply a time value by a numeric factor, `ATM_ADD_2` adds two values of the same kind.

Change Detection

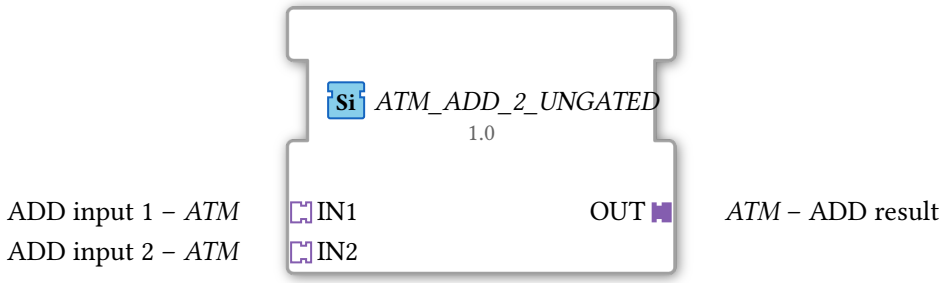
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

`ATM_ADD_2` is a compact, generic function block for adding two time values using adapter interfaces. It is particularly suitable for applications where multiple durations need to be combined into a total duration.

ATM_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ATM_ADD_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_ADD_2_UNGATED** is used to calculate the arithmetic addition of two time values (type `TIME`). It is designed as a generic function block and implements the basic arithmetic operation via adapter interfaces. The block is platform-independent and complies with the IEC 61499 standard.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Name	Direction	Type	Comment
`IN1`	Socket	`adapter::types::unidirectional::ATM`	ADD input 1

IN2	Socket	adapter::types::unidirectional::ATM	ADD input 2
OUT	Plug	adapter::types::unidirectional::ATM	ADD result

The adapters are of type unidirectional::ATM and enable type-safe connections with other function blocks that support the same adapter type.

Functionality

The function block sums the time values received via adapters IN1 and IN2 and provides the result at adapter OUT. The starting point of the summation is the identity value TIME#0s, so that with only one input actually connected, its value is passed through unchanged. The function block operates generically – the calculation is re-executed on every event at one of the input adapters.

Technical Features

- **Generic Structure** – The function block uses a generic class name (GEN_ATM_ADD) at runtime, defined by the attribute eclipse4diac::core::GenericClassName. The same class covers the arities ATM_ADD_2_UNGATED, ATM_ADD_3, and ATM_ADD_4 via the GenericClassName mechanism.
- **Adapter-Based** – Instead of individual data inputs and outputs, all signals are routed via unidirectional adapters.
- **Package Information** – The function block is organized in the package adapter::iec61131::arithmetic.
- **No State Logic** – The addition is stateless; there is no internal state machine.

State Overview

The function block does not have a state machine. The calculation is purely event-driven – on any event at one of the input adapters, the sum is recalculated.

Application-Specific Scenarios

- **Time accumulation** – Summing multiple delay or runtime values into a total duration.
- **Control Engineering** – Combining two dynamically determined time values, e.g., base time plus a surcharge.
- **Generic Library Function Blocks** – Use as the adapter-based counterpart to the classic ADD function, specialized for TIME.

Comparison with Similar Function Blocks

Compared to AR_ADD_2 (addition of two REAL values via adapters), ATM_ADD_2_UNGATED is specialized for the TIME data type. Unlike AR_MULTIME/ATM_AR_MULTIME, which multiply a time value by a numeric factor, ATM_ADD_2_UNGATED adds two values of the same kind.

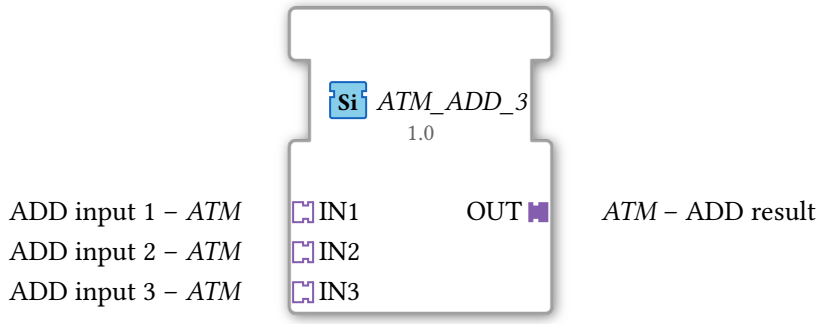
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

ATM_ADD_2_UNGATED is a compact, generic function block for adding two time values using adapter interfaces. It is particularly suitable for applications where multiple durations need to be combined into a total duration.

ATM_ADD_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_ADD_3** is used to calculate the arithmetic addition of two time values (type `TIME`). It is designed as a generic function block and implements the basic arithmetic operation via adapter interfaces. The block is platform-independent and complies with the IEC 61499 standard.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Name	Direction	Type	Comment
`IN1`	Socket	`adapter::types::unidirectional::ATM`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::ATM`	ADD input 2
`IN3`	Socket	`adapter::types::unidirectional::ATM`	ADD input 3
`OUT`	Plug	`adapter::types::unidirectional::ATM`	ADD result

The adapters are of type `unidirectional::ATM` and enable type-safe connections with other function blocks that support the same adapter type.

Functionality

The function block sums the time values received via adapters `IN1` and `IN2` and provides the result at adapter `OUT`. The starting point of the summation is the identity value `TIME#0s`, so that with only one input actually connected, its value is passed through unchanged. The function block operates generically – the calculation is re-executed on every event at one of the input adapters.

Technical Features

- **Generic Structure** – The function block uses a generic class name (`GEN_ATM_ADD`) at runtime, defined by the attribute `eclipse4diac::core::GenericClassName`. The same class covers the arities `ATM_ADD_3`, `ATM_ADD_3`, and `ATM_ADD_4` via the GenericClassName mechanism.
- **Adapter-Based** – Instead of individual data inputs and outputs, all signals are routed via unidirectional adapters.

- **Package Information** – The function block is organized in the package `adapter::iec61131::arithmetic`.
- **No State Logic** – The addition is stateless; there is no internal state machine.

State Overview

The function block does not have a state machine. The calculation is purely event-driven – on any event at one of the input adapters, the sum is recalculated.

Application-Specific Scenarios

- **Time accumulation** – Summing multiple delay or runtime values into a total duration.
- **Control Engineering** – Combining two dynamically determined time values, e.g., base time plus a surcharge.
- **Generic Library Function Blocks** – Use as the adapter-based counterpart to the classic `ADD` function, specialized for `TIME`.

Comparison with Similar Function Blocks

Compared to `AR_ADD_2` (addition of two `REAL` values via adapters), `ATM_ADD_3` is specialized for the `TIME` data type. Unlike `AR_MULTIME`/`ATM_AR_MULTIME`, which multiply a time value by a numeric factor, `ATM_ADD_3` adds two values of the same kind.

Change Detection

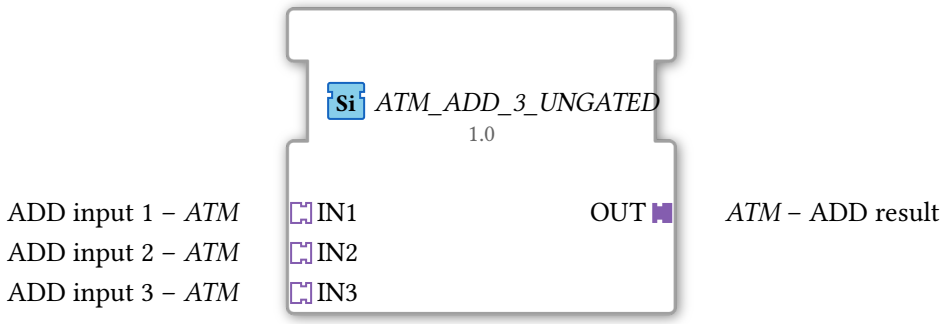
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

`ATM_ADD_3` is a compact, generic function block for adding two time values using adapter interfaces. It is particularly suitable for applications where multiple durations need to be combined into a total duration.

ATM_ADD_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ATM_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_ADD_3_UNGATED** is used to calculate the arithmetic addition of two time values (type `TIME`). It is designed as a generic function block and implements the basic arithmetic operation via adapter interfaces. The block is platform-independent and complies with the IEC 61499 standard.

Interface Structure

Event Inputs
None

Event Outputs
None

Data Inputs
None

Data Outputs
None

Adapters

Name	Direction	Type	Comment
IN1	Socket	adapter::types::unidirectional::ATM	ADD input 1
IN2	Socket	adapter::types::unidirectional::ATM	ADD input 2
IN3	Socket	adapter::types::unidirectional::ATM	ADD input 3
OUT	Plug	adapter::types::unidirectional::ATM	ADD result

The adapters are of type `unidirectional::ATM` and enable type-safe connections with other function blocks that support the same adapter type.

Functionality
The function block sums the time values received via adapters `IN1` and `IN2` and provides the result at adapter `OUT`. The starting point of the summation is the identity value `TIME#0s`, so that with only one input actually connected, its value is passed through unchanged. The function block operates generically – the calculation is re-executed on every event at one of the input adapters.

Technical Features
- **Generic Structure** – The function block uses a generic class name (`GEN_ATM_ADD`) at runtime, defined by the attribute `eclipse4diac::core::GenericClassName`. The same class covers the arities `ATM_ADD_3_UNGATED`, `ATM_ADD_3_UNGATED`, and `ATM_ADD_4` via the GenericClassName mechanism.
- **Adapter-Based** – Instead of individual data inputs and outputs, all signals are routed via unidirectional adapters.
- **Package Information** – The function block is organized in the package `adapter::iec61131::arithmetic`.
- **No State Logic** – The addition is stateless; there is no internal state machine.

State Overview
The function block does not have a state machine. The calculation is purely event-driven – on any event at one of the input adapters, the sum is recalculated.

Application-Specific Scenarios
- **Time accumulation** – Summing multiple delay or runtime values into a total duration.
- **Control Engineering** – Combining two dynamically determined time values, e.g., base time plus a surcharge.
- **Generic Library Function Blocks** – Use as the adapter-based counterpart to the classic `ADD` function, specialized for `TIME`.

Comparison with Similar Function Blocks
Compared to `AR_ADD_2` (addition of two `REAL` values via adapters), `ATM_ADD_3_UNGATED` is specialized for the `TIME` data type. Unlike `AR_MULTIME`/`ATM_AR_MULTIME`, which multiply a time value by a numeric factor, `ATM_ADD_3_UNGATED` adds two values of the same kind.

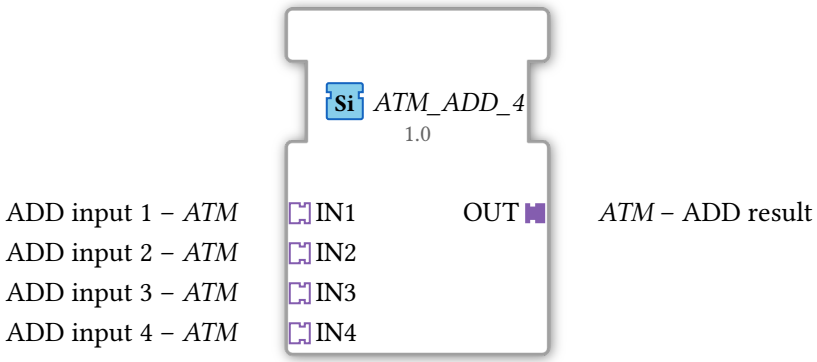
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

`ATM_ADD_3_UNGATED` is a compact, generic function block for adding two time values using adapter interfaces. It is particularly suitable for applications where multiple durations need to be combined into a total duration.

ATM_ADD_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_ADD_4** is used to calculate the arithmetic addition of two time values (type `TIME`). It is designed as a generic function block and implements the basic arithmetic operation via adapter interfaces. The block is platform-independent and complies with the IEC 61499 standard.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Name	Direction	Type	Comment
`IN1`	Socket	`adapter::types::unidirectional::ATM`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::ATM`	ADD input 2
`IN3`	Socket	`adapter::types::unidirectional::ATM`	ADD input 3
`IN4`	Socket	`adapter::types::unidirectional::ATM`	ADD input 4
`OUT`	Plug	`adapter::types::unidirectional::ATM`	ADD result

The adapters are of type `unidirectional::ATM` and enable type-safe connections with other function blocks that support the same adapter type.

Functionality

The function block sums the time values received via adapters `IN1` and `IN2` and provides the result at adapter `OUT`. The starting point of the summation is the identity value `TIME#0s`, so that with only one input actually connected, its value is passed through unchanged. The function block operates generically – the calculation is re-executed on every event at one of the input adapters.

Technical Features

- **Generic Structure** – The function block uses a generic class name (`GEN_ATM_ADD`) at runtime, defined by the attribute `eclipse4diac::core::GenericClassName`. The same class covers the arities `ATM_ADD_4`, `ATM_ADD_3`, and `ATM_ADD_4` via the GenericClassName mechanism.
- **Adapter-Based** – Instead of individual data inputs and outputs, all signals are routed via unidirectional adapters.
- **Package Information** – The function block is organized in the package `adapter::iec61131::arithmetic`.
- **No State Logic** – The addition is stateless; there is no internal state machine.

State Overview

The function block does not have a state machine. The calculation is purely event-driven – on any event at one of the input adapters, the sum is recalculated.

Application-Specific Scenarios

- **Time accumulation** – Summing multiple delay or runtime values into a total duration.
- **Control Engineering** – Combining two dynamically determined time values, e.g., base time plus a surcharge.
- **Generic Library Function Blocks** – Use as the adapter-based counterpart to the classic `ADD` function, specialized for `TIME`.

Comparison with Similar Function Blocks

Compared to `AR_ADD_2` (addition of two `REAL` values via adapters), `ATM_ADD_4` is specialized for the `TIME` data type. Unlike `AR_MULTIME`/`ATM_AR_MULTIME`, which multiply a time value by a numeric factor, `ATM_ADD_4` adds two values of the same kind.

Change Detection

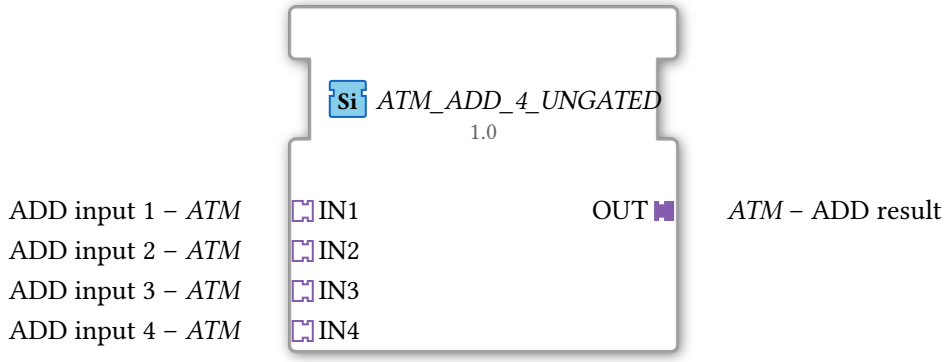
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

`ATM_ADD_4` is a compact, generic function block for adding two time values using adapter interfaces. It is particularly suitable for applications where multiple durations need to be combined into a total duration.

ATM_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ATM_ADD_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_ADD_4_UNGATED** is used to calculate the arithmetic addition of two time values (type `TIME`). It is designed as a generic function block and implements the basic arithmetic operation via adapter interfaces. The block is platform-independent and complies with the IEC 61499 standard.

Interface Structure

Event Inputs

None

Event Outputs

None

Data Inputs

None

Data Outputs

None

Adapters

Name	Direction	Type	Comment
`IN1`	Socket	`adapter::types::unidirectional::ATM`	ADD input 1
`IN2`	Socket	`adapter::types::unidirectional::ATM`	ADD input 2
`IN3`	Socket	`adapter::types::unidirectional::ATM`	ADD input 3
`IN4`	Socket	`adapter::types::unidirectional::ATM`	ADD input 4
`OUT`	Plug	`adapter::types::unidirectional::ATM`	ADD result

The adapters are of type `unidirectional::ATM` and enable type-safe connections with other function blocks that support the same adapter type.

Functionality

The function block sums the time values received via adapters `IN1` and `IN2` and provides the result at adapter `OUT`. The starting point of the summation is the identity value `TIME#0s`, so that with only one input actually connected, its value is passed through unchanged. The function block operates generically – the calculation is re-executed on every event at one of the input adapters.

Technical Features

- **Generic Structure** – The function block uses a generic class name (`GEN_ATM_ADD`) at runtime, defined by the attribute `eclipse4diac::core::GenericClassName`. The same class covers the arities `ATM_ADD_4_UNGATED`, `ATM_ADD_3`, and `ATM_ADD_4_UNGATED` via the GenericClassName mechanism.

- **Adapter-Based** – Instead of individual data inputs and outputs, all signals are routed via unidirectional adapters.
- **Package Information** – The function block is organized in the package `adapter::iec61131::arithmetic`.
- **No State Logic** – The addition is stateless; there is no internal state machine.

State Overview

The function block does not have a state machine. The calculation is purely event-driven – on any event at one of the input adapters, the sum is recalculated.

Application-Specific Scenarios

- **Time accumulation** – Summing multiple delay or runtime values into a total duration.
- **Control Engineering** – Combining two dynamically determined time values, e.g., base time plus a surcharge.
- **Generic Library Function Blocks** – Use as the adapter-based counterpart to the classic `ADD` function, specialized for `TIME`.

Comparison with Similar Function Blocks

Compared to `AR_ADD_2` (addition of two `REAL` values via adapters), `ATM_ADD_4_UNGATED` is specialized for the `TIME` data type. Unlike `AR_MULTIME`/`ATM_AR_MULTIME`, which multiply a time value by a numeric factor, `ATM_ADD_4_UNGATED` adds two values of the same kind.

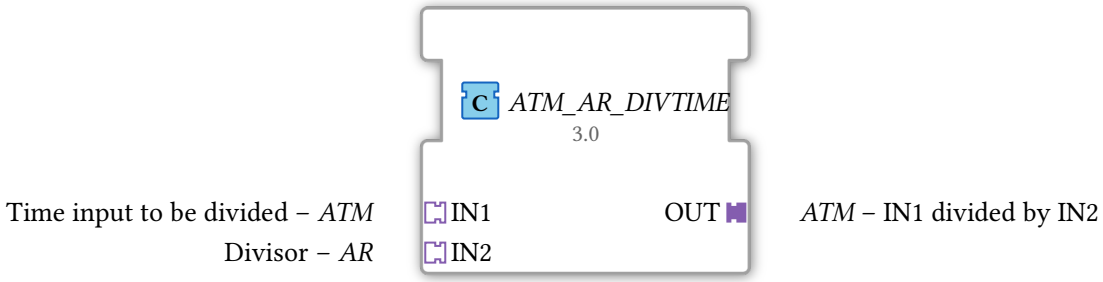
Change Detection

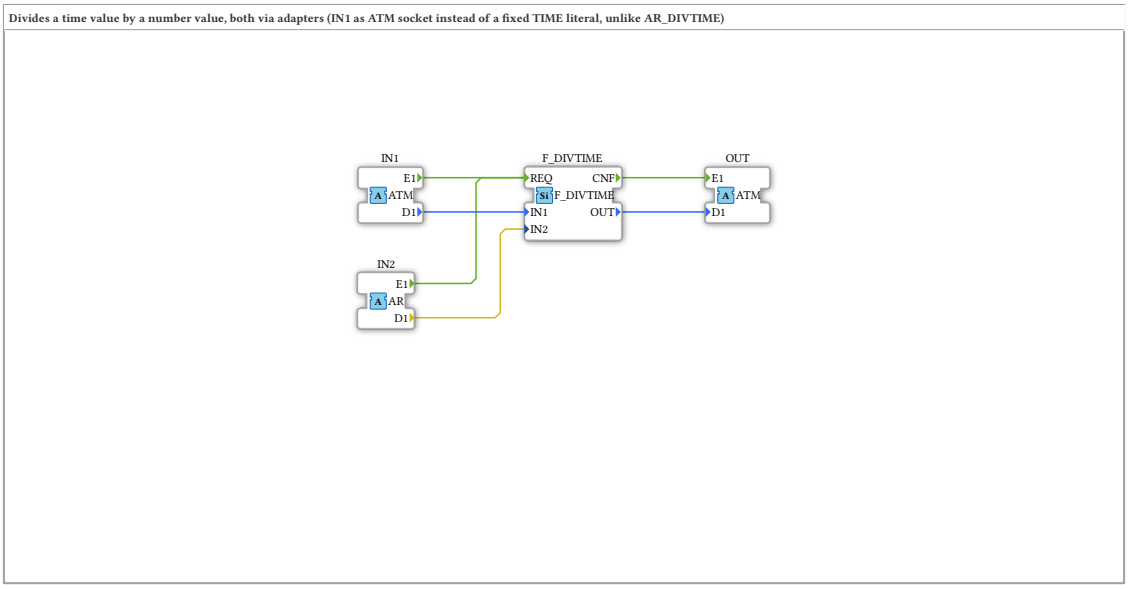
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

`ATM_ADD_4_UNGATED` is a compact, generic function block for adding two time values using adapter interfaces. It is particularly suitable for applications where multiple durations need to be combined into a total duration.

ATM_AR_DIVTIME





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_AR_DIVTIME** divides a time value (type TIME) by a numeric value – the fully adapter-based variant of AR_DIVTIME. Unlike AR_DIVTIME, the dividend (IN1) is also supplied via an adapter (type `ATM`) instead of a fixed data input. The block is classified as a standard arithmetic function according to IEC 61131-3.

Interface Structure

Event Inputs

None – the block has no classic event inputs; triggering happens exclusively via the adapter sockets.

Event Outputs

No explicit event outputs in the interface list. The result is provided via the output adapter **OUT**, which sends an event (E1) once the calculation is complete.

Data Inputs

None.

Data Outputs

None. The division result is output via the output adapter **OUT** as a data value (D1).

Adapters

- **IN1** (Adapter, type `adapter::types::unidirectional::ATM`): Time value to be divided (dividend).
- **IN2** (Adapter, type `adapter::types::unidirectional::AR`): Divisor.
- **OUT** (Adapter, type `adapter::types::unidirectional::ATM`): Output for the division result (TIME).

Functionality

The function block encapsulates the functionality of the internal function block **F_DIVTIME** (type `iec61131::arithmetic::F_DIVTIME`). An event at **IN1.E1** or **IN2.E1** triggers the division: the internal function block divides **IN1.D1** (TIME) by **IN2.D1** (numeric value) and outputs the result to **OUT.D1**. An event is then sent to **OUT.E1**.

Technical Features

- **Fully adapter-based:** Unlike `AR_DIVTIME`, which expects IN1 as a classic data input, `ATM_AR_DIVTIME` implements both operands as adapter sockets (IN1 of type `ATM`, IN2 of type `AR`).
- **Standard compliance:** The function block meets the requirements of IEC 61131-3 (classification: standard arithmetic function).

- **Internal encapsulation:** The actual calculation is performed by the reusable function block F_DIVTIME.

State Overview

The function block does not have an explicit state machine. Event-driven processing is deterministic: waiting for an event at IN1.E1 or IN2.E1, executing the division, sending the result.

Application Scenarios

- Splitting time values where both the time value and the divisor are already available as adapter connections from upstream blocks.
- Modular networks that consistently rely on adapters instead of classic data wiring.

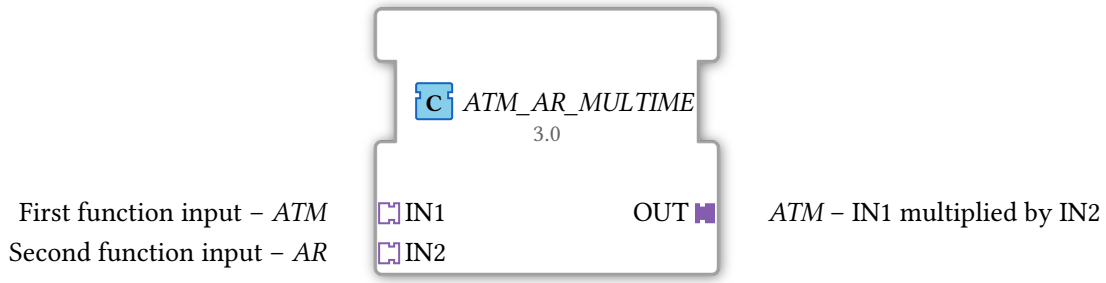
Comparison with Similar Function Blocks

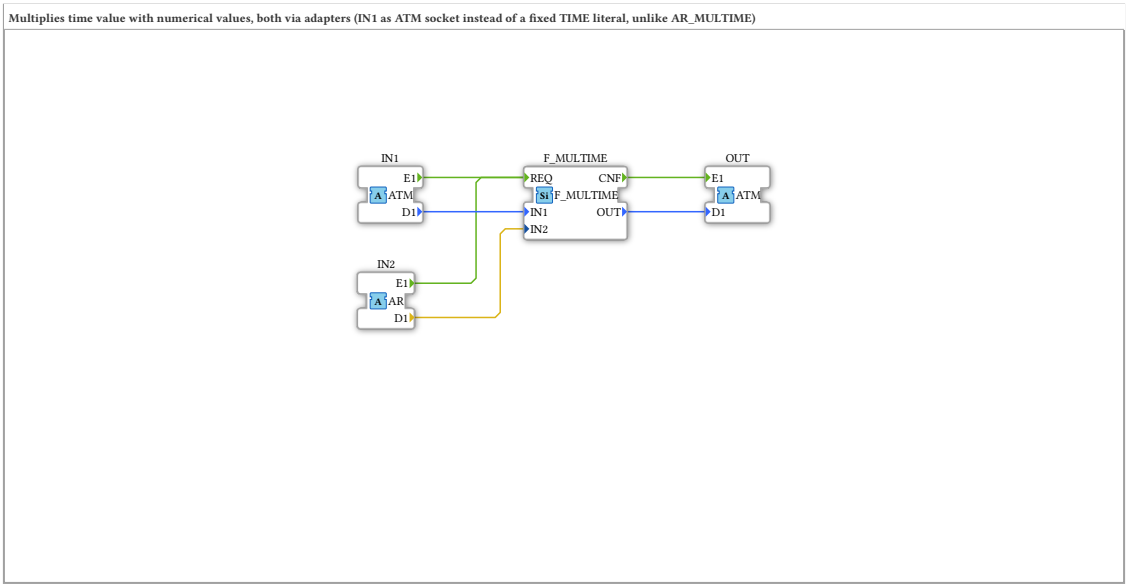
- **AR_DIVTIME:** The mixed variant with IN1 as a classic data input instead of an adapter socket.
- **ATM_AR_MULTIME:** Multiplies instead of dividing, otherwise structurally identical.

Conclusion

`ATM_AR_DIVTIME` provides a fully adapter-based division of time values by numeric factors and is particularly suitable for networks that are built consistently around adapter connections.

ATM_AR_MULTIME





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_AR_MULTIME** multiplies a time value (type TIME) by a numeric value – the fully adapter-based variant of AR_MULTIME. Unlike AR_MULTIME, the time value (IN1) is also supplied via an adapter (type `ATM`) instead of a fixed data input. The block is classified as a standard arithmetic function according to IEC 61131-3.

Interface Structure

Event Inputs

None – the block has no classic event inputs; triggering happens exclusively via the adapter sockets.

Event Outputs

No explicit event outputs in the interface list. The result is provided via the output adapter **OUT**, which sends an event (E1) once the calculation is complete.

Data Inputs

None.

Data Outputs

None. The multiplication result is output via the output adapter **OUT** as a data value (D1).

Adapters

- **IN1** (Adapter, type `adapter::types::unidirectional::ATM`): First function input – the time value to be multiplied.
- **IN2** (Adapter, type `adapter::types::unidirectional::AR`): Second function input – the numeric multiplier.
- **OUT** (Adapter, type `adapter::types::unidirectional::ATM`): Output for the multiplied result (TIME).

Functionality

The function block encapsulates the functionality of the internal function block **F_MULTIME** (type `iec61131::arithmetic::F_MULTIME`). An event at **IN1.E1** or **IN2.E1** triggers the multiplication: the internal function block multiplies **IN1.D1** (TIME) by **IN2.D1** (numeric value) and outputs the result to **OUT.D1**. An event is then sent to **OUT.E1**.

Technical Features

- **Fully adapter-based:** Unlike `AR_MULTIME`, which expects IN1 as a classic data input, `ATM_AR_MULTIME` implements both operands as adapter sockets (IN1 of type `ATM`, IN2 of type `AR`).
- **Standard compliance:** The function block meets the requirements of IEC 61131-3 (classification: standard arithmetic function).
- **Internal encapsulation:** The actual calculation is performed by the reusable function block F_MULTIME.

State Overview

The function block does not have an explicit state machine. Event-driven processing is deterministic: waiting for an event at IN1.E1 or IN2.E1, executing the multiplication, sending the result.

Application Scenarios

- Scaling time values where both the time value and the factor are already available as adapter connections from upstream blocks.
- Modular networks that consistently rely on adapters instead of classic data wiring.

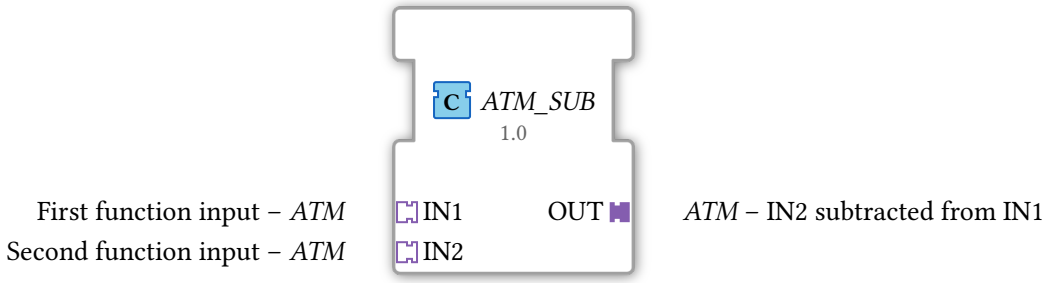
Comparison with Similar Function Blocks

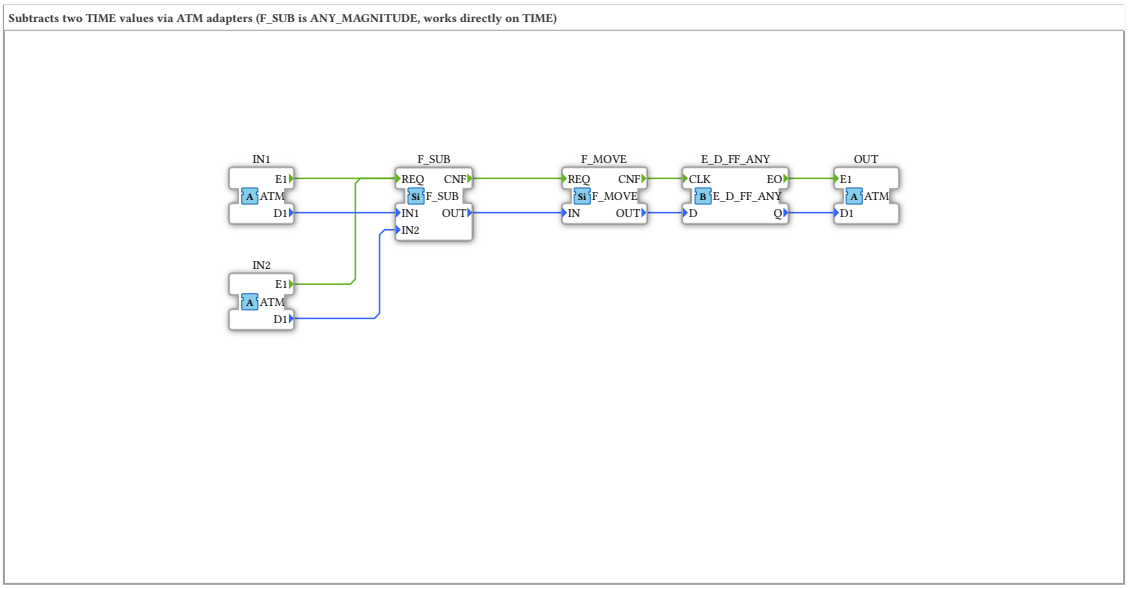
- **AR_MULTIME:** The mixed variant with IN1 as a classic data input instead of an adapter socket.
- **ATM_AR_DIVTIME:** Divides instead of multiplying, otherwise structurally identical.

Conclusion

`ATM_AR_MULTIME` provides a fully adapter-based multiplication of time values by numeric factors and is particularly suitable for networks that are built consistently around adapter connections.

ATM_SUB





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_SUB** subtracts two time values (type TIME) via fully adapter-based interfaces: `IN2` is subtracted from `IN1`, and the result is written to the output adapter only on an actual value change. Unlike the generic `ATM_ADD_*` blocks, ATM_SUB is implemented as a hand-wired composite FB that maps change detection explicitly through its own sub-blocks.

Interface Structure

Event Inputs

None – triggering happens exclusively via the adapter sockets `IN1`/`IN2`.

Event Outputs

No explicit event outputs in the interface list. The result is provided via the output adapter **OUT**, whose event (E1) is only sent on an actual value change.

Data Inputs

None.

Data Outputs

None. The subtraction result is output via the output adapter **OUT** as a data value (D1).

Adapters

- **IN1** (Adapter, type `adapter::types::unidirectional::ATM`): Minuend.
- **IN2** (Adapter, type `adapter::types::unidirectional::ATM`): Subtrahend.
- **OUT** (Adapter, type `adapter::types::unidirectional::ATM`): Result of `IN1 - IN2`.

Functionality

An event at **IN1.E1** or **IN2.E1** internally triggers the function block **F_SUB** (type `iec61131::arithmetic::F_SUB`), which computes `IN1.D1 - IN2.D1`. The result then passes through **F_MOVE** (type `iec61131::selection::F_MOVE`, hardwired with `DataType="TIME"`) into an **E_D_FF_ANY** block (type `iec61499::events::E_D_FF_ANY`). This block compares the new value with the last stored one and only lets the event through to **OUT.E1** if the value has actually changed – the new value is written to **OUT.D1** in every case.

Technical Features

- **Explicit change detection as a composite structure:** Unlike the generic `GEN_ATM_*` blocks, which implement change detection in C++, ATM_SUB maps it via the `F_SUB → F_MOVE → E_D_FF_ANY` chain inside the FBNetwork.
- **F_MOVE as a type anchor:** The `F_MOVE` block carries the `DataType="TIME"` attribute, pinning the generic `ANY` data path to `TIME` before it is fed into `E_D_FF_ANY`.
- **Fully adapter-based:** Both operands and the result are implemented as `ATM` adapters; there are no classic data inputs/outputs.

State Overview

The block implicitly holds the last output value in the internal `E_D_FF_ANY` storage:

- **Wait** for an event at IN1.E1 or IN2.E1
- **Compute** IN1 – IN2 via F_SUB
- **Compare** against the value stored in E_D_FF_ANY
- **Output**: OUT.D1 is always updated, OUT.E1 is only sent on an actual change

Application Scenarios

- Calculating remaining durations or differences between two time values, e.g., target minus actual time.
- Reducing downstream event load when the result does not change across multiple cycles.

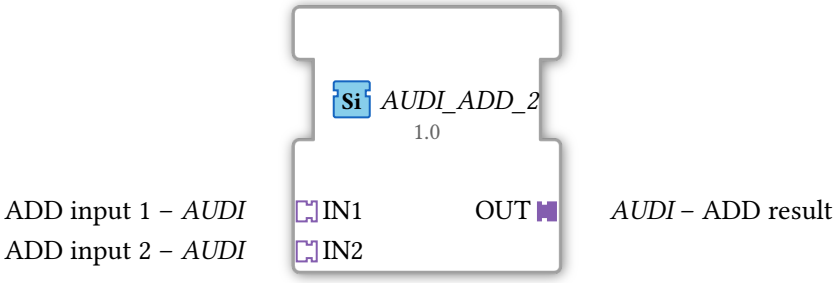
Comparison with Similar Function Blocks

- **ATM_ADD_2/3/4:** Add instead of subtracting; implemented as generic `GEN_ATM_ADD` blocks (change detection in C++, not a visible FBNetwork).
- **F_SUB:** The direct, adapter-free subtraction block without its own change detection, used internally by ATM_SUB.

Conclusion

`ATM_SUB` provides a fully adapter-based subtraction of two time values with explicit, FBNetwork-visible change detection. It is particularly suitable wherever differences between time values are needed and unnecessary follow-up events for an unchanged result should be avoided.

AUDI_ADD_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_ADD_2** performs an arithmetic addition of two input values and outputs the result. The block is generic and uses a standardized adapter interface of type `AUDI`. This allows it to be used for different data types (e.g., integers, floating-point numbers) without requiring modification of the function block itself.

Interface Structure

Event Inputs
None.

Event Outputs
None.

Data Inputs
None.

Data Outputs
None.

Adapter

Type	Direction	Name	Comment
`adapter::types::unidirectional::AUDI`	Socket (Input)	**IN1**	ADD input 1
`adapter::types::unidirectional::AUDI`	Socket (Input)	**IN2**	ADD input 2
`adapter::types::unidirectional::AUDI`	Plug (Output)	**OUT**	ADD result

The three adapters are all of the same type, `AUDI`. This adapter type encapsulates the actual data value (e.g., a numerical value) and enables unidirectional data transfer from the socket to the plug. The addition is performed between the values of `IN1` and `IN2`; The result is provided to `OUT`.

Functionality
The function block **AUDI_ADD_2** does not have traditional data inputs or events. Instead, it communicates exclusively via adapters. As soon as both input adapters (`IN1` and `IN2`) contain valid values, the sum `IN1 + IN2` is calculated internally, and the resulting value is output via the output adapter `OUT`.

The function block is generic: The actual data type used is determined by the adapter configuration and the generic attributes (`GenericClassName`, `TypeHash`). This allows the use of different numeric data types (e.g., `INT`, `REAL`, `LREAL`) without requiring multiple instantiations of the function block itself.

Technical Features
- **Pure Adapter Interface**: No events or direct data inputs/outputs – all communication occurs via adapters.
- **Generic Execution**: The attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash` allow the function block to be specialized for different data types at runtime.
- **Unidirectional Adapters**: The adapters used, of type `AUDI`, are unidirectional, meaning they only transmit values in one direction (socket → plug).
- – **No State Control**: The function block (FB) contains no Execution Control Chart (ECC) and operates without events; the calculation is performed immediately upon receipt of the input values.

State Overview
The **AUDI_ADD_2** function block has no states (no ECC). The addition is performed without delay when valid input values are present.

Application Scenarios
- **Arithmetic Processing in Adapter-Based Systems**: If an architecture consistently relies on adapters, this FB offers a clean way to add two numerical values.
- **Generic Calculations**: Thanks to its generic design, the FB can be used in algorithms that work with different data types (e.g., mixed integer and floating-point environments).
- **Data Flow Systems**: In purely data flow-oriented applications where no events are required, the FB integrates seamlessly.

Comparison with Similar Function Blocks

- **Classic ADD Function Blocks (e.g., `ADD` from IEC 61131-3)**: These typically have concrete data inputs and outputs as well as event control. The `AUDI_ADD_2` replaces this with an adapter interface, enabling a higher level of abstraction and encapsulation.
- **Generic Adders (e.g., `GEN_ADD`)**: Such function blocks are also generic but often use direct data ports. The `AUDI_ADD_2` is specifically optimized for adapter-based interfaces.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The **AUDI_ADD_2** is a specialized, generic addition function block for adapter-based 4diac IDE environments. By eliminating traditional ports and utilizing the `AUDI` adapter type, it is particularly well-suited for data flow-oriented and highly modularized control applications. Its generic design increases reusability across different data types.

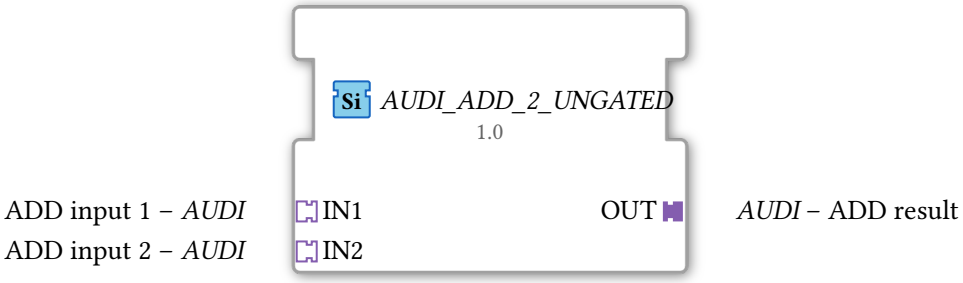
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AUDI_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_ADD_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_ADD_2_UNGATED** performs an arithmetic addition of two input values and outputs the result. The block is generic and uses a standardized adapter interface of type `AUDI`. This allows it to be used for different data types (e.g., integers, floating-point numbers) without requiring modification of the function block itself.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapter

Type	Direction	Name	Comment
`adapter::types::unidirectional::AUDI`	Socket (Input)	**IN1**	ADD input 1
`adapter::types::unidirectional::AUDI`	Socket (Input)	**IN2**	ADD input 2
`adapter::types::unidirectional::AUDI`	Plug (Output)	**OUT**	ADD result

The three adapters are all of the same type, `AUDI`. This adapter type encapsulates the actual data value (e.g., a numerical value) and enables unidirectional data transfer from the socket to the plug. The addition is performed between the values of `IN1` and `IN2`; The result is provided to `OUT`.

Functionality

The function block **AUDI_ADD_2_UNGATED** does not have traditional data inputs or events. Instead, it communicates exclusively via adapters. As soon as both input adapters (`IN1` and `IN2`) contain valid values, the sum `IN1 + IN2` is calculated internally, and the resulting value is output via the output adapter `OUT`.

The function block is generic: The actual data type used is determined by the adapter configuration and the generic attributes (`GenericClassName`, `TypeHash`). This allows the use of different numeric data types (e.g., `INT`, `REAL`, `LREAL`) without requiring multiple instantiations of the function block itself.

Technical Features

- **Pure Adapter Interface**: No events or direct data inputs/outputs – all communication occurs via adapters.
- **Generic Execution**: The attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash` allow the function block to be specialized for different data types at runtime.
- **Unidirectional Adapters**: The adapters used, of type `AUDI`, are unidirectional, meaning they only transmit values in one direction (socket → plug).
- – **No State Control**: The function block (FB) contains no Execution Control Chart (ECC) and operates without events; the calculation is performed immediately upon receipt of the input values.

State Overview

The **AUDI_ADD_2_UNGATED** function block has no states (no ECC). The addition is performed without delay when valid input values are present.

Application Scenarios

- **Arithmetic Processing in Adapter-Based Systems**: If an architecture consistently relies on adapters, this FB offers a clean way to add two numerical values.
- **Generic Calculations**: Thanks to its generic design, the FB can be used in algorithms that work with different data types (e.g., mixed integer and floating-point environments).
- **Data Flow Systems**: In purely data flow-oriented applications where no events are required, the FB integrates seamlessly.

Comparison with Similar Function Blocks

- **Classic ADD Function Blocks (e.g., `ADD` from IEC 61131-3)**: These typically have concrete data inputs and outputs as well as event control. The `AUDI_ADD_2_UNGATED` replaces this with an adapter interface, enabling a higher level of abstraction and encapsulation.
- **Generic Adders (e.g., `GEN_ADD`)**: Such function blocks are also generic but often use direct data ports. The `AUDI_ADD_2_UNGATED` is specifically optimized for adapter-based interfaces.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

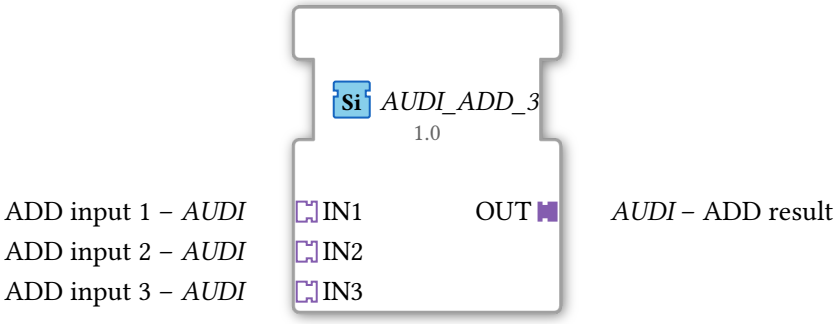
The **AUDI_ADD_2_UNGATED** is a specialized, generic addition function block for adapter-based 4diac IDE environments. By eliminating traditional ports and utilizing the `AUDI` adapter type, it is particularly well-suited for data flow-oriented and highly modularized control applications. Its generic design increases reusability across different data types.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AUDI_ADD_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_ADD_3` performs an arithmetic addition of three input values. It is a generic block that receives the values via adapter interfaces and outputs the result via another adapter. The function block is designed as part of the 4diac IDE for IEC 61499-2.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available. The input values are provided via the adapter interfaces.

Data Outputs

No direct data outputs available. The result is provided via the adapter output.

Adapter

Name	Type	Direction	Comment
IN1	`adapter::types::unidirectional::AUDI`	Socket (Input)	ADD input 1
IN2	`adapter::types::unidirectional::AUDI`	Socket (Input)	ADD input 2
IN3	`adapter::types::unidirectional::AUDI`	Socket (Input)	ADD input 3
OUT	`adapter::types::unidirectional::AUDI`	Plug (Output)	ADD result

Functionality

The function block adds the three values received via the adapters `IN1`, `IN2`, and `IN3`. The result is continuously output via the adapter `OUT`. Since this is a generic function block, the specific data type of the addition is determined at runtime by the attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`.

Technical Features

- The function block is stateless (purely combinatorial) – no ECC is required.
- It uses the Eclipse 4diac generic type system to enable flexible reusability for various arithmetic data types (e.g., INT, REAL, LREAL).
- The connection is established exclusively via adapters of type `adapter::types::unidirectional::AUDI`, which provide a standardized unidirectional interface for arithmetic operations.

State Overview

The function block (FB) does not have a state machine (no ECC elements in the XML). Its behavior is purely data-driven: The output is recalculated whenever an input value changes.

Application Scenarios

- **Summation of Measured Values:** Summing three analog sensor values (e.g., temperatures).
- **Mixing Calculation:** Adding three quantities or volume flows in process engineering.
- **Generic Addition:** Used in libraries where the data type is defined during project planning.

Comparison with Similar Function Blocks

- **ADD_2:** Performs an addition of two values – `AUDI_ADD_3` extends this to three inputs.
- **ADD_4:** Adds four values – `AUDI_ADD_3` specifies three inputs.
- **MUL_3:** Multiplies three values – unlike addition.

All of the above blocks use the same adapter concept and are generic.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUDI_ADD_3` is a flexible, generic function block for adding three values. Its pure adapter interface and stateless nature make it easy to use and ideal for modular automation solutions based on IEC 61499.

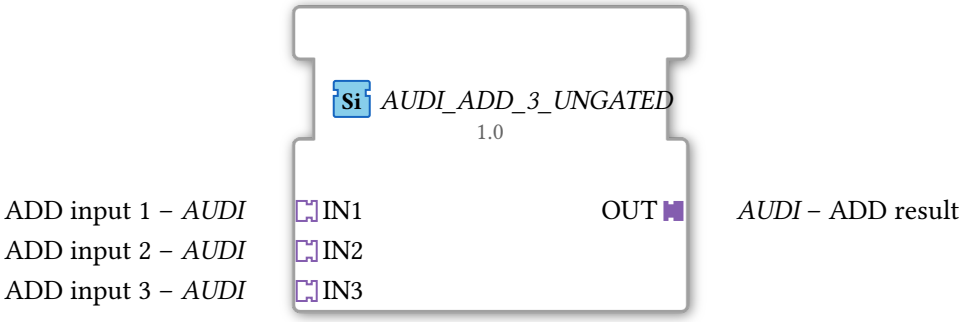
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AUDI_ADD_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_ADD_3_UNGATED` performs an arithmetic addition of three input values. It is a generic block that receives the values via adapter interfaces and outputs the result via another adapter. The function block is designed as part of the 4diac IDE for IEC 61499-2.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No direct data inputs available. The input values are provided via the adapter interfaces.

Data Outputs

No direct data outputs available. The result is provided via the adapter output.

Adapter

Name	Type	Direction	Comment
IN1	`adapter::types::unidirectional::AUDI`	Socket (Input)	ADD input 1
IN2	`adapter::types::unidirectional::AUDI`	Socket (Input)	ADD input 2
IN3	`adapter::types::unidirectional::AUDI`	Socket (Input)	ADD input 3
OUT	`adapter::types::unidirectional::AUDI`	Plug (Output)	ADD result

Functionality

The function block adds the three values received via the adapters `IN1`, `IN2`, and `IN3`. The result is continuously output via the adapter `OUT`. Since this is a generic function block, the specific data type of the addition is determined at runtime by the attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash`.

Technical Features

- The function block is stateless (purely combinatorial) – no ECC is required.
- It uses the Eclipse 4diac generic type system to enable flexible reusability for various arithmetic data types (e.g., INT, REAL, LREAL).
- The connection is established exclusively via adapters of type `adapter::types::unidirectional::AUDI`, which provide a standardized unidirectional interface for arithmetic operations.

State Overview

The function block (FB) does not have a state machine (no ECC elements in the XML). Its behavior is purely data-driven: The output is recalculated whenever an input value changes.

Application Scenarios

- **Summation of Measured Values:** Summing three analog sensor values (e.g., temperatures).
- **Mixing Calculation:** Adding three quantities or volume flows in process engineering.
- **Generic Addition:** Used in libraries where the data type is defined during project planning.

Comparison with Similar Function Blocks

- **ADD_2:** Performs an addition of two values – `AUDI_ADD_3_UNGATED` extends this to three inputs.
- **ADD_4:** Adds four values – `AUDI_ADD_3_UNGATED` specifies three inputs.
- **MUL_3:** Multiplies three values – unlike addition.

All of the above blocks use the same adapter concept and are generic.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

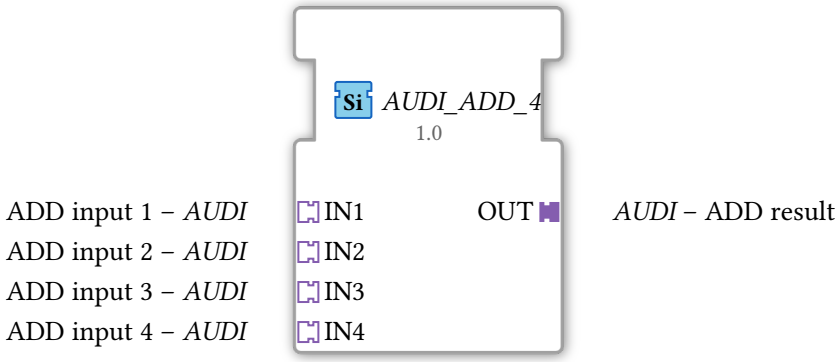
The `AUDI_ADD_3_UNGATED` is a flexible, generic function block for adding three values. Its pure adapter interface and stateless nature make it easy to use and ideal for modular automation solutions based on IEC 61499.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AUDI_ADD_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_ADD_4` is a generic function block for calculating the arithmetic addition of four input values. It is implemented as a purely adapter-based function block without event or data interfaces. The use of adapters enables flexible coupling with other function blocks that use the same adapter type, `adapter::types::unidirectional::AUDI`.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

The function block has no direct data inputs. The values to be added are provided via adapter inputs.

Data Outputs

The function block has no direct data outputs. The result is passed on via an adapter output.

Adapter

Name	Direction	Type	Description
`IN1`	Socket (Input)	`adapter::types::unidirectional::AUDI`	First Addend
`IN2`	Socket (Input)	`adapter::types::unidirectional::AUDI`	Second Addend
`IN3`	Socket (Input)	`adapter::types::unidirectional::AUDI`	Third Addend
`IN4`	Socket (Input)	`adapter::types::unidirectional::AUDI`	Fourth Addend
`OUT`	Plug (Output)	`adapter::types::unidirectional::AUDI`	Result of the Addition

Functionality

The function block waits for valid values at the adapter inputs `IN1` through `IN4`. As soon as all four inputs provide a value, the sum `IN1 + IN2 + IN3 + IN4` is calculated and output via the `OUT` adapter. The actual data type specification is defined by the generic attribute `eclipse4diac::core::GenericClassName`, which is set to `'GEN_AUDI_ADD'`. This allows the function block to be instantiated for various numeric data types (e.g., INT, REAL, LREAL), provided the adapter type used, `AUDI`, supports them.

Technical Features

- **Pure Adapter Function Block**: The function block does not exchange events. Synchronization occurs implicitly through the connected adapter links.
- **Generic Data Type**: The specific data type is defined at runtime via the attributes `GenericClassName` and `TypeHash`. This enables a reusable implementation without changing the logic.
- **No State Machines**: The function block does not contain an ECC (Execution Control Chart) – addition is performed continuously or on demand by the data flow.

State Overview

The function block has no explicit states. The processing is data-driven: As soon as all four input values are available, the result is calculated and output.

Application Scenarios

- **Averaging**: In combination with a downstream division block, the sum can be used to calculate an average.
- **Summation of Measured Values**: For summing four analog input signals (e.g., temperature, pressure, flow rate) in an automation solution.
- **Cascaded Addition**: Multiple `AUDI_ADD_4` blocks can be cascaded to process a larger number of summands.

Comparison with Similar Blocks

- **`ADD` (Standard 61499)**: A typical ADD block has event inputs and data inputs/outputs. The `AUDI_ADD_4`, on the other hand, is entirely adapter-based and has no events, which necessitates stronger coupling between function blocks via adapters.
- **`AUDI_ADD_2`**: A hypothetical function block with only two adapter inputs – `AUDI_ADD_4` extends this to four summands.
- **Generic Function Blocks**: The attribute `GenericClassName` makes this function block similar to the generic approach of IEC 61499, where the data type is only determined at runtime.
- **`AUDI_ADD_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUDI_ADD_4` is a flexible, pure adapter function block for adding four values. Thanks to its generic design, it is suitable for various numeric data types and can be used in modular automation projects that rely on adapter-based communication. Eliminating events simplifies handling in data-driven systems, but requires correct adapter cabling.

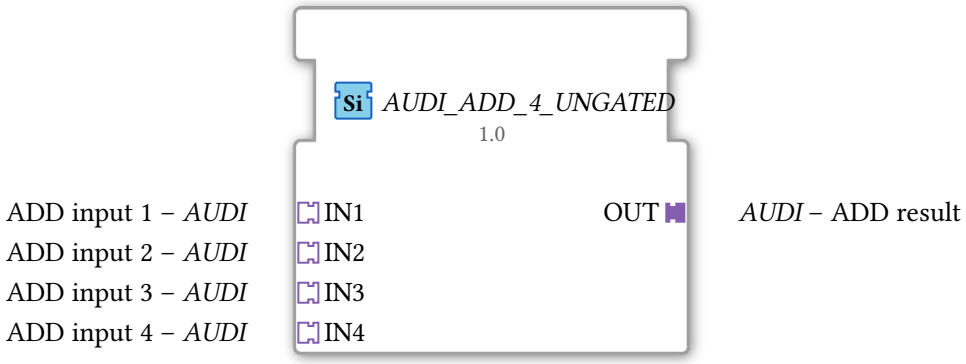
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUDI_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_ADD_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_ADD_4_UNGATED` is a generic function block for calculating the arithmetic addition of four input values. It is implemented as a purely adapter-based function block without event or data interfaces. The use of adapters enables flexible coupling with other function blocks that use the same adapter type, `adapter::types::unidirectional::AUDI`.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

The function block has no direct data inputs. The values to be added are provided via adapter inputs.

Data Outputs

The function block has no direct data outputs. The result is passed on via an adapter output.

Adapter

Name	Direction	Type	Description
IN1	Socket (Input)	adapter::types::unidirectional::AUDI	First Addend
IN2	Socket (Input)	adapter::types::unidirectional::AUDI	Second Addend
IN3	Socket (Input)	adapter::types::unidirectional::AUDI	Third Addend
IN4	Socket (Input)	adapter::types::unidirectional::AUDI	Fourth Addend
OUT	Plug (Output)	adapter::types::unidirectional::AUDI	Result of the Addition

Functionality

The function block waits for valid values at the adapter inputs IN1 through IN4. As soon as all four inputs provide a value, the sum IN1 + IN2 + IN3 + IN4 is calculated and output via the OUT adapter. The actual data type specification is defined by the generic attribute eclipse4diac::core::GenericClassName, which is set to 'GEN_AUDI_ADD'. This allows the function block to be instantiated for various numeric data types (e.g., INT, REAL, LREAL), provided the adapter type used, AUDI, supports them.

Technical Features

- **Pure Adapter Function Block**: The function block does not exchange events. Synchronization occurs implicitly through the connected adapter links.
- **Generic Data Type**: The specific data type is defined at runtime via the attributes GenericClassName and TypeHash. This enables a reusable implementation without changing the logic.
- **No State Machines**: The function block does not contain an ECC (Execution Control Chart) – addition is performed continuously or on demand by the data flow.

State Overview

The function block has no explicit states. The processing is data-driven: As soon as all four input values are available, the result is calculated and output.

Application Scenarios

- **Averaging**: In combination with a downstream division block, the sum can be used to calculate an average.
- **Summation of Measured Values**: For summing four analog input signals (e.g., temperature, pressure, flow rate) in an automation solution.
- **Cascaded Addition**: Multiple AUDI_ADD_4_UNGATED blocks can be cascaded to process a larger number of summands.

Comparison with Similar Blocks

- **ADD (Standard 61499)**: A typical ADD block has event inputs and data inputs/outputs. The AUDI_ADD_4_UNGATED, on the other hand, is entirely adapter-based and has no events, which necessitates stronger coupling between function blocks via adapters.
- **AUDI_ADD_2**: A hypothetical function block with only two adapter inputs – AUDI_ADD_4_UNGATED extends this to four summands.
- **Generic Function Blocks**: The attribute GenericClassName makes this function block similar to the generic approach of IEC 61499, where the data type is only determined at runtime.
- **AUDI_ADD_4**: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AUDI_ADD_4_UNGATED is a flexible, pure adapter function block for adding four values. Thanks to its generic design, it is suitable for various numeric data types and can be used in modular automation

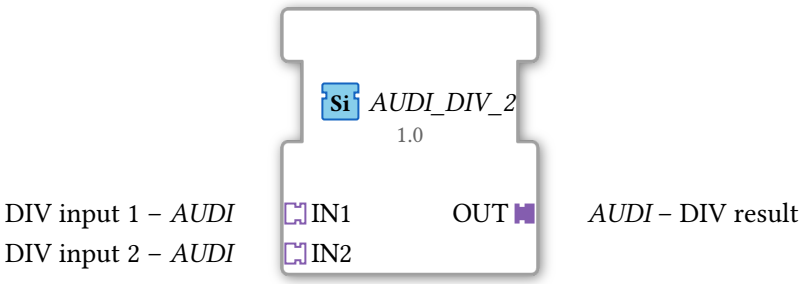
projects that rely on adapter-based communication. Eliminating events simplifies handling in data-driven systems, but requires correct adapter cabling.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUDI_DIV_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) **AUDI_DIV_2** is used to perform arithmetic division. It is a generic function block (`GEN_AUDI_DIV`) that can flexibly work with various data types within the adapter structure. Unlike classic mathematical function blocks, which have dedicated data inputs, this block uses adapter connections to encapsulate signals and control flows.

Interface Structure

The function block does not have any direct, classic event or data inputs on its surface. All communication is handled via adapters.

```
      +--------------------------------------+
| AUDI_DIV_2 | | | [IN1] =|=======================================>|=OUT | (unidirectional::AUDI) (unidirect..) | [IN2] =|=======================================>| | (unidirectional::AUDI) |
```

Event Inputs

No direct event inputs are available. Event control is handled via the adapters.

Event Outputs

No direct event outputs are available. Event control is handled via the adapters.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

Type	Name	Adapter Type	Description
Socket (Female)	`IN1`	`adapter::types::unidirectional::AUDI`	First input value for division (dividend).
Socket (Female)	`IN2`	`adapter::types::unidirectional::AUDI`	Second input value for the division (divisor).
Plug (connector)	`OUT`	`adapter::types::unidirectional::AUDI`	Result of the division (quotient).

Functionality

The function block calculates the quotient of the values provided via the adapters `IN1` and `IN2`:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.80}$$

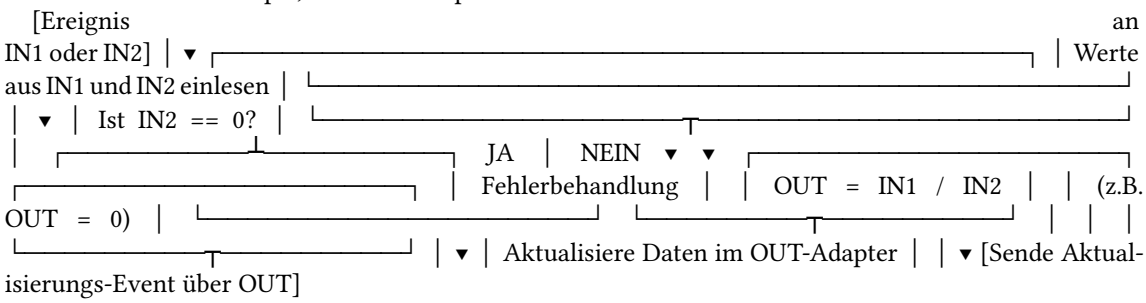
Since the interfaces are implemented as unidirectional `AUDI` adapters, an update of the input values at `IN1` or `IN2` (via the event channels integrated into the adapter) triggers the calculation in the function block. The result is then immediately signaled via the event encapsulated in the `OUT` adapter and passed on to subsequent function blocks.

Technical Features

- **Generic Type:** The function block is declared as `GEN_AUDI_DIV` via the attribute `eclipse4diac::core::GenericClassName`. This enables automatic type adaptation to the specific data structures used within the `AUDI` adapters.
- **Clean Wiring:** Encapsulating data and event signals in adapters minimizes the visual "spaghetti code" problem in 4diac-ide.
- **Error Handling (Division by Zero):** Since this is a generic division, the underlying runtime environment (e.g., forte) catches division by zero to prevent the controller from crashing. In such a case, the output is typically set to a default value (e.g., `0` or `NaN`/`Infinity`, depending on the data type).

State Overview

Execution follows a simple, event-driven pattern:



Application Scenarios

- **Scaling and Normalization Tasks:** Division of measured values transmitted in standardized `AUDI` data structures.
- **Ratio Calculations:** Determining ratios in the process industry (e.g., flow ratios of two media).
- **Signal Processing:** Modular use in control architectures where strict separation through adapter structures is required.

Comparison with Similar Components

Component Type	Interface Type	Wiring Effort	Flexibility
Standard `DIV`	Individual Pins (REQ, IN1, IN2, OUT, CNF)	High (many lines)	Low (fixed data types per instance)
`AUDI_DIV_2`	Adapter (`IN1`, `IN2`, `OUT`)	Very low	High (generic adaptation via adapter)

Change Detection

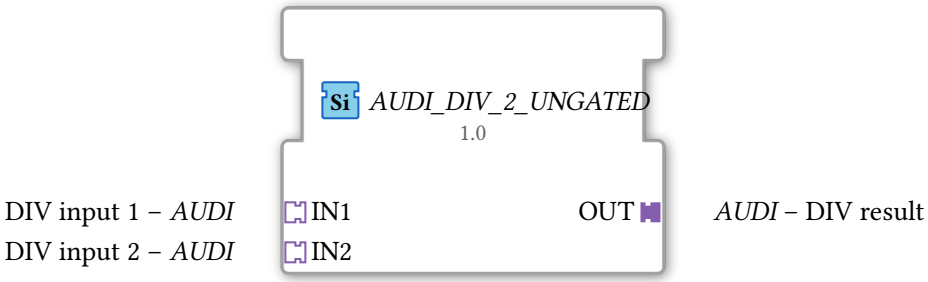
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUDI_DIV_2` is a modern, adapter-based function block for IEC 61499. It is ideally suited for complex projects in the 4diac-ide where clarity, encapsulation, and easy maintainability of the application diagram are paramount.

AUDI_DIV_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_DIV_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) **AUDI_DIV_2_UNGATED** is used to perform arithmetic division. It is a generic function block (`GEN_AUDI_DIV`) that can flexibly work with various data types within the adapter structure. Unlike classic mathematical function blocks, which have dedicated data inputs, this block uses adapter connections to encapsulate signals and control flows.

Interface Structure

The function block does not have any direct, classic event or data inputs on its surface. All communication is handled via adapters.

+----------+
| AUDI_DIV_2_UNGATED | | | [IN1] =|=========>|=OUT | (unidirectional::AUDI) (unidirect..) | [IN2] =|=========>| | (unidirectional::AUDI) |

Event Inputs

No direct event inputs are available. Event control is handled via the adapters.

Event Outputs

No direct event outputs are available. Event control is handled via the adapters.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

Type	Name	Adapter Type	Description
Socket (Female)	`IN1`	`adapter::types::unidirectional::AUDI`	AUDI input value for division (dividend).

Socket (Female)	`IN2`	`adapter::types::unidirectional::AUDI`	Second input value for the division (divisor).
Plug (connector)	`OUT`	`adapter::types::unidirectional::AUDI`	Result of the division (quotient).

Functionality

The function block calculates the quotient of the values provided via the adapters `IN1` and `IN2`:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.81}$$

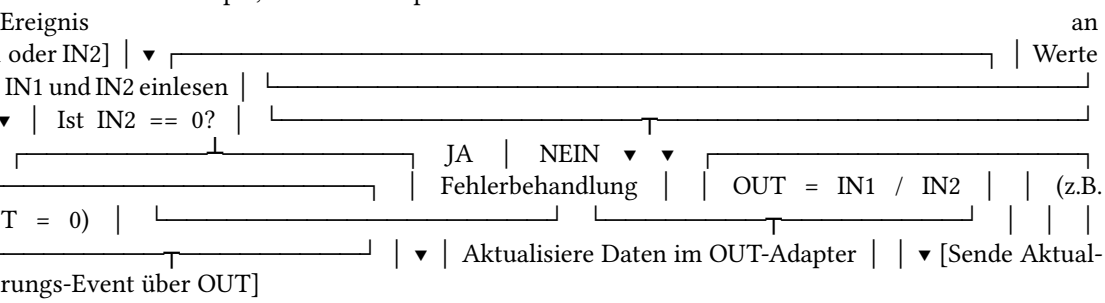
Since the interfaces are implemented as unidirectional `AUDI` adapters, an update of the input values at `IN1` or `IN2` (via the event channels integrated into the adapter) triggers the calculation in the function block. The result is then immediately signaled via the event encapsulated in the `OUT` adapter and passed on to subsequent function blocks.

Technical Features

- **Generic Type:** The function block is declared as `GEN_AUDI_DIV` via the attribute `eclipse4diac::core::GenericClassName`. This enables automatic type adaptation to the specific data structures used within the `AUDI` adapters.
- **Clean Wiring:** Encapsulating data and event signals in adapters minimizes the visual "spaghetti code" problem in 4diac-ide.
- **Error Handling (Division by Zero):** Since this is a generic division, the underlying runtime environment (e.g., forte) catches division by zero to prevent the controller from crashing. In such a case, the output is typically set to a default value (e.g., `0` or `NaN`/`Infinity`, depending on the data type).

State Overview

Execution follows a simple, event-driven pattern:



Application Scenarios

- **Scaling and Normalization Tasks:** Division of measured values transmitted in standardized `AUDI` data structures.
- **Ratio Calculations:** Determining ratios in the process industry (e.g., flow ratios of two media).
- **Signal Processing:** Modular use in control architectures where strict separation through adapter structures is required.

Comparison with Similar Components

Component Type	Interface Type	Wiring Effort	Flexibility
Standard `DIV`	Individual Pins (REQ, IN1, IN2, OUT, CNF)	High (many lines)	Low (fixed data types per instance)
`AUDI_DIV_2_UNGATED`	Adapter (`IN1`, `IN2`, `OUT`)	Very low	High (generic adaptation via adapter)

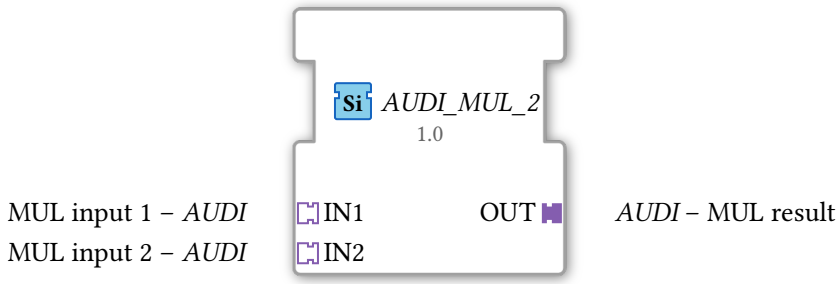
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUDI_DIV_2_UNGATED` is a modern, adapter-based function block for IEC 61499. It is ideally suited for complex projects in the 4diac-ide where clarity, encapsulation, and easy maintainability of the application diagram are paramount.

AUDI_MUL_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_MUL_2` is a generic function block for calculating the arithmetic product of two numeric values. It receives its operands via two input adapters and outputs the result via one output adapter. Its generic design allows the use of different data types (e.g., integers, floating-point numbers), which are defined by the specific adapter type.

Interface Structure

Event Inputs

No direct event inputs are available. Control is integrated into the input adapters `IN1` and `IN2`.

Event Outputs

No direct event outputs are available. Control is integrated into the output adapter `OUT`.

Data Inputs

No direct data ports. The input values are transmitted via the adapters `IN1` and `IN2`.

Data Outputs

No direct data ports. The result is output via the adapter `OUT`.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN1`	`adapter::types::unidirectional::AUDI`	First Multiplication Operand
Socket (Input)	`IN2`	`adapter::types::unidirectional::AUDI`	Second Multiplication Operand
Plug (Output)	`OUT`	`adapter::types::unidirectional::AUDI`	Product of the two input values

Note: The adapter type `AUDI` is a unidirectional, typed adapter that encapsulates the actual numerical value.

Functionality

The function block performs the multiplication `OUT = IN1 * IN2` as soon as both input values are available via the adapters. Since no events are present, the processing is purely data-driven: As soon as the socket adapters `IN1` and `IN2` provide valid values, the product is calculated and output via the plug adapter `OUT`.

The function block is designed generically. The specific data type (e.g., `REAL`, `INT`, `LINT`) is determined by the adapter type used. Multiplication is performed using the operation defined for that type.

- **Adapter-Based Communication:** The function block does not use traditional event/data ports, but exclusively adapters. This enables loose coupling and reuse in different contexts.
- **Generic Design:** The attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash` identify the function block as generic. The actual data type is only defined in the specific project, e.g., by typing the adapter.
- **No Execution Control Chart (ECC):** Since no events exist, a state machine is not required. Processing occurs continuously with valid data.

This function block has no states or ECC state diagrams. Execution is determined solely by the data flow.

- Arithmetic calculations in industrial control applications where the multiplication of two values is required (e.g., scaling, unit conversion).
- Use in complex function blocks where operands are obtained from other components via adapters.
- Replacement for classic MUL function blocks when loose coupling via adapters is desired.
- **Classic MUL FB** (with event/data ports): Operates in an event-driven manner and requires explicit data inputs/outputs. `AUDI_MUL_2`, on the other hand, uses adapters, which allows for more flexible interconnection.
- **Other generic arithmetic function blocks** (e.g., `AUDI_ADD_2`): Same structure, but with a different operation. The adapter interface remains identical, ensuring easy interchangeability.

AUDI_MUL_2` is a generic, adapter-based multiplication function block for IEC 61499 applications. By eliminating events and using typed adapters, it is particularly well-suited for dataflow-oriented and loosely coupled systems. Its generic design allows it to be used with various numeric data types without requiring modification of the function block itself.

AUDI_MUL_2` is a generic, adapter-based multiplication function block for IEC 61499 applications. —

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

Change Detection

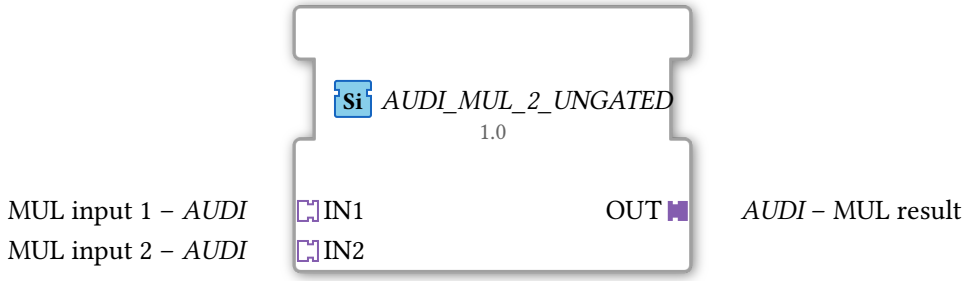
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

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

AUDI_MUL_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_MUL_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUDI_MUL_2_UNGATED` is a generic function block for calculating the arithmetic product of two numeric values. It receives its operands via two input adapters and outputs the result via one output adapter. Its generic design allows the use of different data types (e.g., integers, floating-point numbers), which are defined by the specific adapter type.

Interface Structure

Event Inputs

No direct event inputs are available. Control is integrated into the input adapters `IN1` and `IN2`.

Event Outputs

No direct event outputs are available. Control is integrated into the output adapter `OUT`.

Data Inputs

No direct data ports. The input values are transmitted via the adapters `IN1` and `IN2`.

Data Outputs

No direct data ports. The result is output via the adapter `OUT`.

Adapters

Direction	Name	Type	Description
Socket (Input)	`IN1`	`adapter::types::unidirectional::AUDI`	First Multiplication Operand
Socket (Input)	`IN2`	`adapter::types::unidirectional::AUDI`	Second Multiplication Operand
Plug (Output)	`OUT`	`adapter::types::unidirectional::AUDI`	Product of the two input values

Note: The adapter type `AUDI` is a unidirectional, typed adapter that encapsulates the actual numerical value.

Functionality

The function block performs the multiplication `OUT = IN1 * IN2` as soon as both input values are available via the adapters. Since no events are present, the processing is purely data-driven: As soon as the socket adapters `IN1` and `IN2` provide valid values, the product is calculated and output via the plug adapter `OUT`.

The function block is designed generically. The specific data type (e.g., `REAL`, `INT`, `LINT`) is determined by the adapter type used. Multiplication is performed using the operation defined for that type.

- **Adapter-Based Communication:** The function block does not use traditional event/data ports, but exclusively adapters. This enables loose coupling and reuse in different contexts.
- **Generic Design:** The attributes `eclipse4diac::core::GenericClassName` and `eclipse4diac::core::TypeHash` identify the function block as generic. The actual data type is only defined in the specific project, e.g., by typing the adapter.
- **No Execution Control Chart (ECC):** Since no events exist, a state machine is not required. Processing occurs continuously with valid data.

This function block has no states or ECC state diagrams. Execution is determined solely by the data flow.

- Arithmetic calculations in industrial control applications where the multiplication of two values is required (e.g., scaling, unit conversion).
- Use in complex function blocks where operands are obtained from other components via adapters.
- Replacement for classic MUL function blocks when loose coupling via adapters is desired.
- **Classic MUL FB** (with event/data ports): Operates in an event-driven manner and requires explicit data inputs/outputs. `AUDI_MUL_2_UNGATED`, on the other hand, uses adapters, which allows for more flexible interconnection.
- **Other generic arithmetic function blocks** (e.g., `AUDI_ADD_2`): Same structure, but with a different operation. The adapter interface remains identical, ensuring easy interchangeability.

AUDI_MUL_2_UNGATED` is a generic, adapter-based multiplication function block for IEC 61499 applications. By eliminating events and using typed adapters, it is particularly well-suited for dataflow-oriented and loosely coupled systems. Its generic design allows it to be used with various numeric data types without requiring modification of the function block itself.

AUDI_MUL_2_UNGATED` is a generic, adapter-based multiplication function block for IEC 61499 applications. —

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

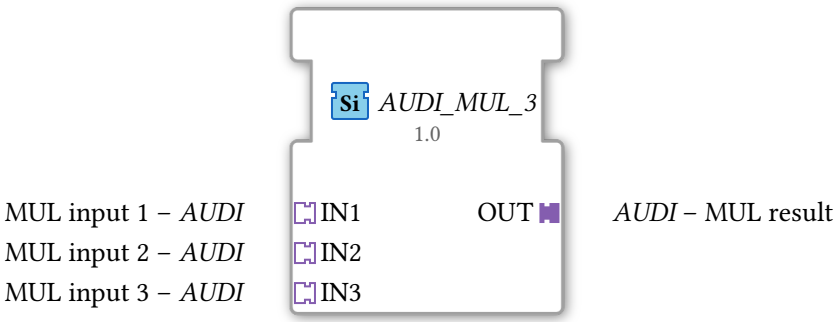
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

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

AUDI_MUL_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_MUL_3** is a generic function block for multiplying three input values. The values are provided via unidirectional adapters of type **AUDI**, and the result is output via a similar adapter. The function block is designed for use in IEC 61499-based control systems.

Interface Structure

Event Inputs

Not present – all data transmission is adapter-based without explicit event control.

Event Outputs

Not present.

Data Inputs

Not present – the input values are provided via **Sockets IN1, IN2, and IN3** as adapters.

Data Outputs

Not present – the result is output via **Plug OUT** as an adapter.

Adapter

Name	Direction	Type	Description
IN1	Socket (Input)	`adapter::types::unidirectional::AUDI`	First multiplication factor.
IN2	Socket (Input)	`adapter::types::unidirectional::AUDI`	Second multiplication factor.
IN3	Socket (Input)	`adapter::types::unidirectional::AUDI`	Third multiplication factor.
OUT	Plug (Output)	`adapter::types::unidirectional::AUDI`	Result of multiplication `IN1 * IN2 * IN3`.

Functionality

The function block **AUDI_MUL_3** performs an arithmetic multiplication of the three values provided via the adapters **IN1, IN2, and IN3**. The product is calculated continuously and made available via the adapter **OUT**. The underlying data type **AUDI** is defined generically (via the attribute `eclipse4diac::core::GenericClassName`) and can be replaced by a specific numeric type (e.g., INT, REAL) at configuration time. Explicit triggering by events is not required, as the adapters handle the data transfer independently.

Technical Features

- **Generic Data Type**: The function block is designed as a generic block (`GenericClassName = 'GEN_AUDI_MUL'`). The adapter type used, **AUDI**, can be replaced by any numeric data type, as long as the same type is used on all adapters.
- **Adapter-based interface**: Instead of discrete data and event ports, the function block communicates exclusively via unidirectional adapters. This enables flexible chaining with other function blocks of the same adapter type.
- **No explicit state logic**: The function block does not require an internal state machine – multiplication is purely data flow-driven.

State overview

The **AUDI_MUL_3** does not have an explicit state machine. The output **OUT** always corresponds to the current product of the three inputs. As soon as valid data is present at all sockets, the result is calculated without delay.

Application scenarios

- **Multidimensional calculations**: Combining several physical quantities that need to be multiplied (e.g., volume from length, width, and height).
- **Signal scaling**: Multiplying a sensor value by two calibration factors.
- **Nested arithmetic**: Use as part of a larger calculation chain where the product of three inputs is required.

Comparison with Similar Function Blocks

- **AUDI_MUL_2**: Performs the multiplication of only two inputs. **AUDI_MUL_3** extends this functionality to three factors.
- **AUDI_ADD_3**: Performs the addition of three values, while **AUDI_MUL_3** performs multiplication – both use the same adapter structure.
- **Conventional IEC 61499 Function Blocks**: Typical multifunction blocks often use separate data and event ports. The adapter-based approach simplifies typing and reusability in generic frameworks.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AUDI_MUL_3 is a compact, generic function block for multiplying three values. Its adapter-driven interface and generic typing make it ideal for modular, reusable control applications. Its simple, data-flow-oriented approach without state machines makes it intuitive to use and extensible.

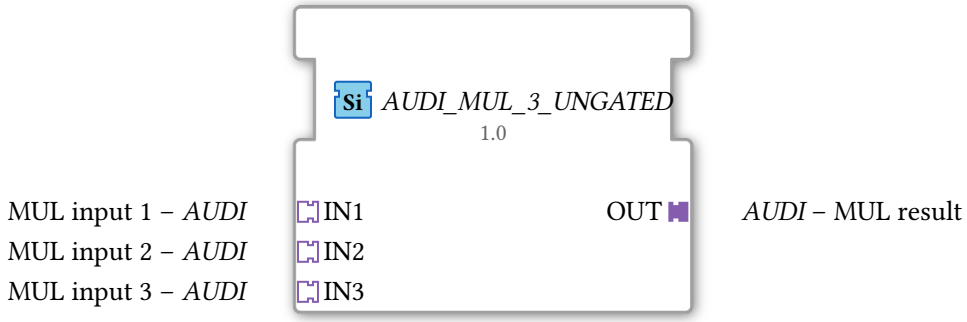
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUDI_MUL_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_MUL_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_MUL_3_UNGATED** is a generic function block for multiplying three input values. The values are provided via unidirectional adapters of type **AUDI**, and the result is output via a similar adapter. The function block is designed for use in IEC 61499-based control systems.

Interface Structure

Event Inputs

Not present – all data transmission is adapter-based without explicit event control.

Event Outputs

Not present.

Data Inputs

Not present – the input values are provided via **Sockets IN1, IN2, and IN3** as adapters.

Data Outputs

Not present – the result is output via **Plug OUT** as an adapter.

Adapter

Name	Direction	Type	Description
IN1	Socket (Input)	`adapter::types::unidirectional::AUDI`	First multiplication factor.
IN2	Socket (Input)	`adapter::types::unidirectional::AUDI`	Second multiplication factor.
IN3	Socket (Input)	`adapter::types::unidirectional::AUDI`	Third multiplication factor.
OUT	Plug (Output)	`adapter::types::unidirectional::AUDI`	Result of multiplication `IN1 * IN2 * IN3`.

Functionality

The function block **AUDI_MUL_3_UNGATED** performs an arithmetic multiplication of the three values provided via the adapters **IN1, IN2, and IN3**. The product is calculated continuously and made available via the adapter **OUT**. The underlying data type **AUDI** is defined generically (via the attribute `eclipse4diac::core::GenericClassName`) and can be replaced by a specific numeric type (e.g., INT, REAL) at configuration time. Explicit triggering by events is not required, as the adapters handle the data transfer independently.

Technical Features

- **Generic Data Type**: The function block is designed as a generic block (`GenericClassName = 'GEN_AUDI_MUL'`). The adapter type used, **AUDI**, can be replaced by any numeric data type, as long as the same type is used on all adapters.
- **Adapter-based interface**: Instead of discrete data and event ports, the function block communicates exclusively via unidirectional adapters. This enables flexible chaining with other function blocks of the same adapter type.
- **No explicit state logic**: The function block does not require an internal state machine – multiplication is purely data flow-driven.

State overview

The **AUDI_MUL_3_UNGATED** does not have an explicit state machine. The output **OUT** always corresponds to the current product of the three inputs. As soon as valid data is present at all sockets, the result is calculated without delay.

Application scenarios

- **Multidimensional calculations**: Combining several physical quantities that need to be multiplied (e.g., volume from length, width, and height).
- **Signal scaling**: Multiplying a sensor value by two calibration factors.
- **Nested arithmetic**: Use as part of a larger calculation chain where the product of three inputs is required.

Comparison with Similar Function Blocks

- **AUDI_MUL_2**: Performs the multiplication of only two inputs. **AUDI_MUL_3_UNGATED** extends this functionality to three factors.
- **AUDI_ADD_3**: Performs the addition of three values, while **AUDI_MUL_3_UNGATED** performs multiplication – both use the same adapter structure.
- **Conventional IEC 61499 Function Blocks**: Typical multifunction blocks often use separate data and event ports. The adapter-based approach simplifies typing and reusability in generic frameworks.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

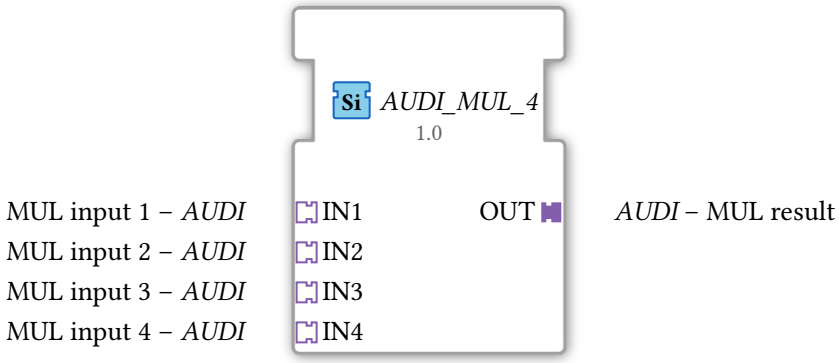
AUDI_MUL_3_UNGATED is a compact, generic function block for multiplying three values. Its adapter-driven interface and generic typing make it ideal for modular, reusable control applications. Its simple, data-flow-oriented approach without state machines makes it intuitive to use and extensible.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AUDI_MUL_4



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_MUL_4** performs an arithmetic multiplication based on four input values. It is a generic function block (FB) that operates via adapter interfaces with a unidirectional data structure of type `adapter::types::unidirectional::AUDI`. The block is specified according to IEC 61499-2 and is used particularly in automation solutions where scalable multiplication of multiple input variables is required.

Interface Structure

Event Inputs

The block does not have traditional event inputs. Control is integrated into the input adapters.

Event Outputs

The block does not have traditional event outputs. Control is integrated into the output adapter.

Data Inputs

The block does not have any traditional data inputs. All input values are provided via adapter interfaces.

Data Outputs

This function block does not have traditional data outputs. The multiplication result is output via an adapter plug.

Adapter

Type	Name	Direction	Comment
`adapter::types::unidirectional::AUDI`	IN1	Input (Socket)	MUL input 1
`adapter::types::unidirectional::AUDI`	IN2	Input (Socket)	MUL input 2
`adapter::types::unidirectional::AUDI`	IN3	Input (Socket)	MUL input 3
`adapter::types::unidirectional::AUDI`	IN4	Input (Socket)	MUL input 4
`adapter::types::unidirectional::AUDI`	OUT	Output (Plug)	MUL result

The adapter `AUDI` is defined as a unidirectional data type and transports the respective numerical values as well as the result.

Functionality

The function block (FB) multiplies the values applied to the four adapters **IN1**, **IN2**, **IN3**, and **IN4** and provides the product at the output adapter **OUT**. Processing is asynchronous, occurring as soon as all required input data is available. The function block is designed as a generic FB, meaning the data type used can be specified at runtime or during deployment via the attribute `eclipse4diac::core::GenericClassName`. By default, the name `'GEN_AUDI_MUL'` is used.

- **Generic Implementation**: The function block (FB) is generic and can be adapted to various numeric data types (e.g., INT, REAL, LREAL). Specific implementation is achieved via the class attribute.
- **Adapter-Based Communication**: Adapters are used instead of direct inputs/outputs. This allows for flexible data encapsulation and enables reuse in different contexts.
- **Type Hash**: The attribute value `eclipse4diac::core::TypeHash` is empty and can be supplemented at runtime with the actual hash of the data type used.
- **No State Machines**: The FB does not have an event-driven state machine (ECM), as multiplication is purely data-driven.

Not applicable – the function block operates without a state machine. Multiplication is triggered by the availability of input data.

- **Process Automation**: Multiplication of four analog measurements (e.g., flow rate, pressure, temperature, density) to calculate a mass flow correction.
- **Machine Control**: Calculation of total loads or torques from multiple individual variables.
- **Data Preprocessing**: Scaling or weighting of sensor data in a higher-level control system.
- **Image Processing**: Combination of multiple channels (e.g., red, green, blue, intensity) through multiplication.

Unlike classic IEC 61131 multiplication function blocks (e.g., `MUL` with fixed data types), `AUDI_MUL_4` offers a generic, adapter-based interface that enables flexible reuse. Other multipliers, such as `MUL_2` or `AUDI_MUL_2`, process only two inputs, while `AUDI_MUL_4` multiplies four inputs simultaneously. Function blocks with event-driven processing (e.g., `MUL_E`) require additional triggers, whereas this function block operates purely data-driven.

The function block **AUDI_MUL_4** is a specialized, generic multiplication block for four input values. Its use of adapters and generic design make it particularly suitable for modular, reusable automation solutions where flexibility and scalability are paramount. The simple, event-free interface facilitates integration into data-driven architectures.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

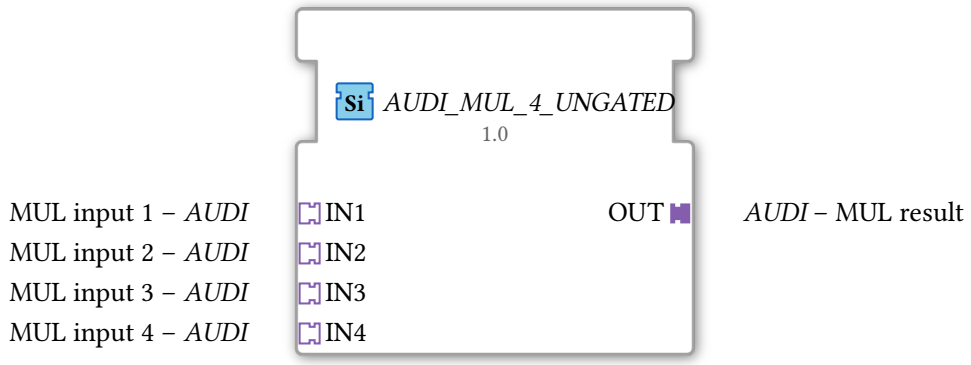
Conclusion

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

AUDI_MUL_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_MUL_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/

frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_MUL_4_UNGATED** performs an arithmetic multiplication based on four input values. It is a generic function block (FB) that operates via adapter interfaces with a unidirectional data structure of type `adapter::types::unidirectional::AUDI`. The block is specified according to IEC 61499-2 and is used particularly in automation solutions where scalable multiplication of multiple input variables is required.

Interface Structure

Event Inputs

The block does not have traditional event inputs. Control is integrated into the input adapters.

Event Outputs

The block does not have traditional event outputs. Control is integrated into the output adapter.

Data Inputs

The block does not have any traditional data inputs. All input values are provided via adapter interfaces.

Data Outputs

This function block does not have traditional data outputs. The multiplication result is output via an adapter plug.

Adapter

Type	Name	Direction	Comment
`adapter::types::unidirectional::AUDI`	IN1	Input (Socket)	MUL input 1
`adapter::types::unidirectional::AUDI`	IN2	Input (Socket)	MUL input 2
`adapter::types::unidirectional::AUDI`	IN3	Input (Socket)	MUL input 3
`adapter::types::unidirectional::AUDI`	IN4	Input (Socket)	MUL input 4
`adapter::types::unidirectional::AUDI`	OUT	Output (Plug)	MUL result

The adapter `AUDI` is defined as a unidirectional data type and transports the respective numerical values as well as the result.

Functionality

The function block (FB) multiplies the values applied to the four adapters **IN1**, **IN2**, **IN3**, and **IN4** and provides the product at the output adapter **OUT**. Processing is asynchronous, occurring as soon as all required input data is available. The function block is designed as a generic FB, meaning the data type used can be

specified at runtime or during deployment via the attribute `eclipse4diac::core::GenericClassName`. By default, the name `'GEN_AUDI_MUL'` is used.

- **Generic Implementation**: The function block (FB) is generic and can be adapted to various numeric data types (e.g., INT, REAL, LREAL). Specific implementation is achieved via the class attribute.
- **Adapter-Based Communication**: Adapters are used instead of direct inputs/outputs. This allows for flexible data encapsulation and enables reuse in different contexts.
- **Type Hash**: The attribute value `eclipse4diac::core::TypeHash` is empty and can be supplemented at runtime with the actual hash of the data type used.
- **No State Machines**: The FB does not have an event-driven state machine (ECM), as multiplication is purely data-driven.

Not applicable – the function block operates without a state machine. Multiplication is triggered by the availability of input data.

- **Process Automation**: Multiplication of four analog measurements (e.g., flow rate, pressure, temperature, density) to calculate a mass flow correction.
- **Machine Control**: Calculation of total loads or torques from multiple individual variables.
- **Data Preprocessing**: Scaling or weighting of sensor data in a higher-level control system.
- **Image Processing**: Combination of multiple channels (e.g., red, green, blue, intensity) through multiplication.

Unlike classic IEC 61131 multiplication function blocks (e.g., `MUL` with fixed data types), `AUDI_MUL_4_UNGATED` offers a generic, adapter-based interface that enables flexible reuse. Other multipliers, such as `MUL_2` or `AUDI_MUL_2`, process only two inputs, while `AUDI_MUL_4_UNGATED` multiplies four inputs simultaneously. Function blocks with event-driven processing (e.g., `MUL_E`) require additional triggers, whereas this function block operates purely data-driven.

The function block **AUDI_MUL_4_UNGATED** is a specialized, generic multiplication block for four input values. Its use of adapters and generic design make it particularly suitable for modular, reusable automation solutions where flexibility and scalability are paramount. The simple, event-free interface facilitates integration into data-driven architectures.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

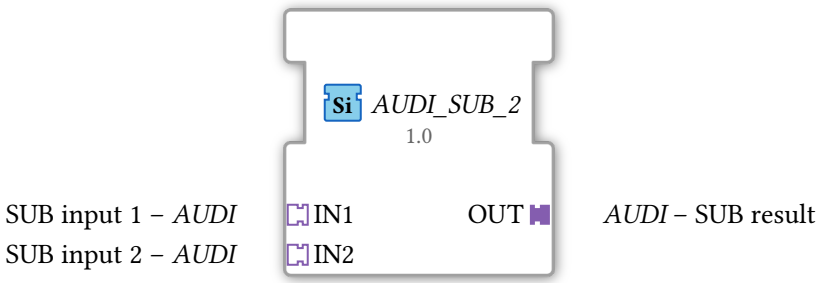
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

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

AUDI_SUB_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) `AUDI_SUB_2` is a generic function block for performing arithmetic subtraction. It is based on the generic class `GEN_AUDI_SUB` and uses unidirectional adapters of type `AUDI` for structured and event-driven data transmission. This achieves clean encapsulation of data and events, reducing the complexity of wiring in IEC 61499 applications.

Interface Structure

Since the block is entirely based on adapters, it does not have any traditional, directly visible event or data pins at the top level. All communication is handled via the adapters.

Event Inputs

No direct event inputs are available. Event control is encapsulated within the adapters.

Event Outputs

No direct event outputs are available. Event control is encapsulated within the adapters.

Data Inputs

No direct data inputs are available. Data transmission occurs via the adapter interfaces.

Data Outputs

No direct data outputs are available. Data transmission occurs via the adapter interfaces.

Adapters

Sockets (Input Adapters / Sockets)

- **IN1** (Type: `adapter::types::unidirectional::AUDI`):
- First input for subtraction (minuend).
- **IN2** (Type: `adapter::types::unidirectional::AUDI`):
- Second input for subtraction (subtrahend).

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUDI`):
- Output of the function block that provides the result of the subtraction (difference).

–

Functionality

The function block calculates the mathematical difference between the two values provided via the input adapters according to the following principle:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.82}$$

As soon as values at the input adapters `IN1` or `IN2` change and a corresponding trigger event is received via the adapter, the calculation is performed internally. The result and the associated update event are then forwarded via the output adapter `OUT`.

Technical Features

- **Generic Implementation:** The `eclipse4diac::core::GenericClassName = 'GEN_AUDI_SUB'` attribute makes the function block flexible. Depending on the implementation of the underlying adapter, it can support various data types.
- **Encapsulation by Adapter:** Using the unidirectional adapter `AUDI` bundles data and event lines. This significantly improves clarity in the 4diac application editor, as fewer lines need to be drawn.

–

State Overview

The function block behaves like a stateless mathematical operator. There is no internal state loop (ECC states in the classical sense) that is maintained for an extended period. Each activation by an event at the inputs leads to a direct calculation and update of the output.

Application Scenarios

- **Signal Processing:** Subtraction of sensor values (e.g., offset compensation or calculation of differential pressures/temperatures) in systems that are consistently based on the `AUDI` adapter architecture.
- **Control Engineering:** Calculation of the control error ($e = w - x$) by subtracting the actual value from the setpoint.

Comparison with Similar Components

- **Standard SUB Component (IEC 61131-3 / IEC 61499):** A classic `SUB` component has dedicated data inputs (`IN1`, `IN2`), a data output (`OUT`), and event inputs and outputs (e.g., `REQ` / `CNF`). The `AUDI_SUB_2`, on the other hand, bundles these signals in adapters, which improves reusability and modularity in complex architectures.
- **AUDI_ADD_2:** The counterpart for addition. Uses the same adapter interface, but adds the input values ($IN1 + IN2$).

Change Detection

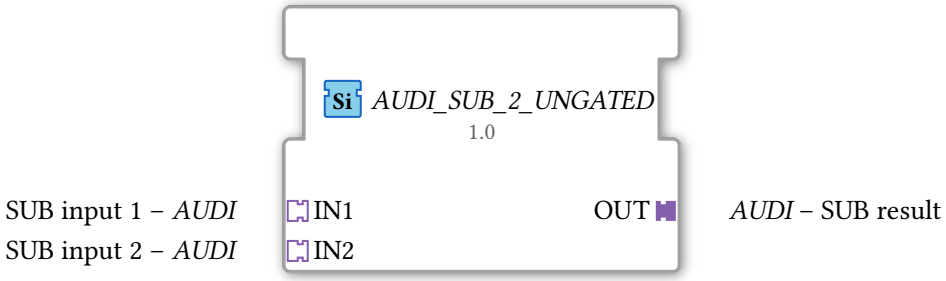
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUDI_SUB_2` is a specialized, yet flexible, function block for subtraction due to its generic nature. It is ideally suited for service-oriented architectures within 4diac, where a unified interface concept using unidirectional adapters is paramount.

AUDI_SUB_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_SUB_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) `AUDI_SUB_2_UNGATED` is a generic function block for performing arithmetic subtraction. It is based on the generic class `GEN_AUDI_SUB` and uses unidirectional adapters of type `AUDI` for structured and event-driven data transmission. This achieves clean encapsulation of data and events, reducing the complexity of wiring in IEC 61499 applications.

Interface Structure

Since the block is entirely based on adapters, it does not have any traditional, directly visible event or data pins at the top level. All communication is handled via the adapters.

Event Inputs

No direct event inputs are available. Event control is encapsulated within the adapters.

Event Outputs

No direct event outputs are available. Event control is encapsulated within the adapters.

Data Inputs

No direct data inputs are available. Data transmission occurs via the adapter interfaces.

Data Outputs

No direct data outputs are available. Data transmission occurs via the adapter interfaces.

Adapters

Sockets (Input Adapters / Sockets)

- **IN1** (Type: `adapter::types::unidirectional::AUDI`):
- First input for subtraction (minuend).
- **IN2** (Type: `adapter::types::unidirectional::AUDI`):
- Second input for subtraction (subtrahend).

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUDI`):
- Output of the function block that provides the result of the subtraction (difference).

–

Functionality

The function block calculates the mathematical difference between the two values provided via the input adapters according to the following principle:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.83}$$

As soon as values at the input adapters `IN1` or `IN2` change and a corresponding trigger event is received via the adapter, the calculation is performed internally. The result and the associated update event are then forwarded via the output adapter `OUT`.

Technical Features

- **Generic Implementation:** The `eclipse4diac::core::GenericClassName = 'GEN_AUDI_SUB'` attribute makes the function block flexible. Depending on the implementation of the underlying adapter, it can support various data types.
- **Encapsulation by Adapter:** Using the unidirectional adapter `AUDI` bundles data and event lines. This significantly improves clarity in the 4diac application editor, as fewer lines need to be drawn.

–

State Overview

The function block behaves like a stateless mathematical operator. There is no internal state loop (ECC states in the classical sense) that is maintained for an extended period. Each activation by an event at the inputs leads to a direct calculation and update of the output.

Application Scenarios

- **Signal Processing:** Subtraction of sensor values (e.g., offset compensation or calculation of differential pressures/temperatures) in systems that are consistently based on the `AUDI` adapter architecture.
- **Control Engineering:** Calculation of the control error ($e = w - x$) by subtracting the actual value from the setpoint.

Comparison with Similar Components

- **Standard SUB Component (IEC 61131-3 / IEC 61499):** A classic `SUB` component has dedicated data inputs (`IN1`, `IN2`), a data output (`OUT`), and event inputs and outputs (e.g., `REQ` / `CNF`). The

AUDI_SUB_2_UNGATED, on the other hand, bundles these signals in adapters, which improves reusability and modularity in complex architectures.
- **AUDI_ADD_2:** The counterpart for addition. Uses the same adapter interface, but adds the input values ($IN1 + IN2$).

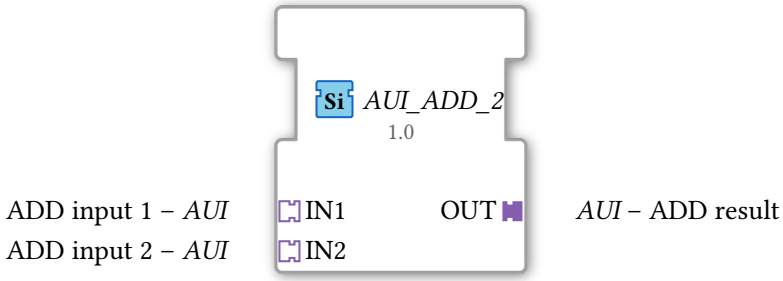
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The AUDI_SUB_2_UNGATED is a specialized, yet flexible, function block for subtraction due to its generic nature. It is ideally suited for service-oriented architectures within 4diac, where a unified interface concept using unidirectional adapters is paramount.

AUI_ADD_2



No image available

• – ◦ • – ◦ • – ◦ •

Introduction

The function block AUI_ADD_2 is a generic function block (Generic FB) designed to perform arithmetic addition. It uses unidirectional adapters of type AUI (Adapter Unidirectional Interface) to add two input values and output the result. Because it is a generic block based on the class GEN_AUI_ADD, it can be flexibly used for various data types supported by the corresponding adapters.

Interface Structure

Since this function block is entirely based on adapters, it has no direct, discrete data or event channels at the top level. All communication and control are handled via the adapter interfaces.

Event Inputs

No direct event inputs available (control is handled via the connected adapters).

Event Outputs

No direct event outputs available (forwarding is handled via the connected adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

- **Sockets (Input Adapters):**
- **IN1** (Type: adapter::types::unidirectional::AUI): First input (addend) for arithmetic addition.
- **IN2** (Type: adapter::types::unidirectional::AUI): Second input (addend) for arithmetic addition.
- **Plugs (Output Adapters):**
- **OUT** (Type: adapter::types::unidirectional::AUI): Output for the result of the addition (IN1 + IN2).

Functionality

As soon as values are present at the input adapters `IN1` and `IN2`, or a corresponding update event is triggered via the adapters, the function block performs the addition.

The mathematical operation is:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.84}$$

The result and the associated output event are immediately passed on to the subsequent function blocks via the output adapter `OUT`.

By defining it as a generic building block (`GEN_AUI_ADD`), the internal addition logic dynamically adapts to the data types of the connected adapters (e.g., integer or floating-point types).

Technical Features

- **Generic Design:** The building block is classified as `GEN_AUI_ADD` via the attribute `eclipse4diac::core::GenericClassName`. This enables high reusability for different numeric data types.
- **Adapter Encapsulation:** The use of the unidirectional adapter `AUI` ensures a very clean design in the 4diac application editor, as event and data flows are bundled in a single connection.
- **Unidirectional Flow:** The interfaces are designed as purely unidirectional connections, which dictates a clear direction of signal and data flow from the sockets (`IN1`, `IN2`) to the plug (`OUT`).

State Overview

The module does not have a complex internal state machine (ECC). Processing is event-driven and stateless (combinatorial) based on the triggers received via the adapters.

Application Scenarios

- **Signal Addition:** Combining two sensor values (e.g., to calculate a total volumetric flow rate from two individual flow meters).
- **Setpoint Offset:** Applying a constant or variable correction value (offset) to an existing control parameter.
- **Structured Signal Processing:** For use in more complex control loops within IEC 61499 where a clean, adapter-based wiring style is preferred.

Comparison with Similar Components

Compared to standard IEC 61131-3 addition components (such as a classic `ADD` component with discrete inputs), the `AUI_ADD_2` eliminates the need for separate wiring of REQ/CNF events and the actual data lines. This reduces the potential for errors when manually creating event-data associations (WITH connections).

Change Detection

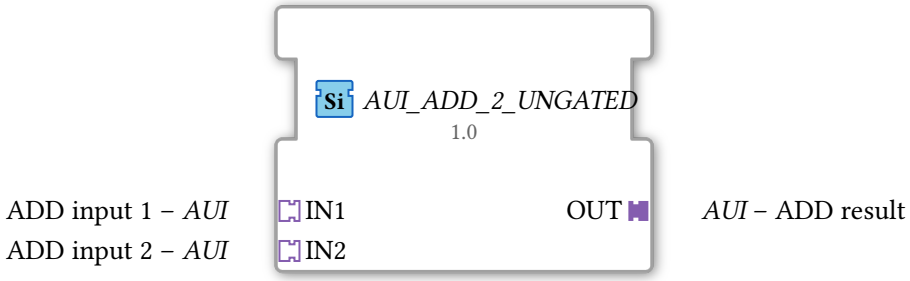
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUI_ADD_2` is a highly efficient, modular component for mathematical addition in modern IEC 61499 systems. Through the consistent use of adapters, he promotes a clear, error-resistant and easily maintainable application design in the 4diac IDE.

AUI_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_ADD_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



No image available

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_ADD_2_UNGATED` is a generic function block (Generic FB) designed to perform arithmetic addition. It uses unidirectional adapters of type `AUI` (Adapter Unidirectional Interface) to add two input values and output the result. Because it is a generic block based on the class `GEN_AUI_ADD`, it can be flexibly used for various data types supported by the corresponding adapters.

Interface Structure

Since this function block is entirely based on adapters, it has no direct, discrete data or event channels at the top level. All communication and control are handled via the adapter interfaces.

Event Inputs

No direct event inputs available (control is handled via the connected adapters).

Event Outputs

No direct event outputs available (forwarding is handled via the connected adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

- **Sockets (Input Adapters):**
- **IN1** (Type: `adapter::types::unidirectional::AUI`): First input (addend) for arithmetic addition.
- **IN2** (Type: `adapter::types::unidirectional::AUI`): Second input (addend) for arithmetic addition.
- **Plugs (Output Adapters):**
- **OUT** (Type: `adapter::types::unidirectional::AUI`): Output for the result of the addition (`IN1 + IN2`).

Functionality

As soon as values are present at the input adapters `IN1` and `IN2`, or a corresponding update event is triggered via the adapters, the function block performs the addition.

The mathematical operation is:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} \tag{2.85}$$

The result and the associated output event are immediately passed on to the subsequent function blocks via the output adapter `OUT`.

By defining it as a generic building block (`GEN_AUI_ADD`), the internal addition logic dynamically adapts to the data types of the connected adapters (e.g., integer or floating-point types).

Technical Features

- **Generic Design:** The building block is classified as `GEN_AUI_ADD` via the attribute `eclipse4diac::core::GenericClassName`. This enables high reusability for different numeric data types.

- **Adapter Encapsulation:** The use of the unidirectional adapter `AUI` ensures a very clean design in the 4diac application editor, as event and data flows are bundled in a single connection.
- **Unidirectional Flow:** The interfaces are designed as purely unidirectional connections, which dictates a clear direction of signal and data flow from the sockets (`IN1`, `IN2`) to the plug (`OUT`).

State Overview

The module does not have a complex internal state machine (ECC). Processing is event-driven and stateless (combinatorial) based on the triggers received via the adapters.

Application Scenarios

- **Signal Addition:** Combining two sensor values (e.g., to calculate a total volumetric flow rate from two individual flow meters).
- **Setpoint Offset:** Applying a constant or variable correction value (offset) to an existing control parameter.
- **Structured Signal Processing:** For use in more complex control loops within IEC 61499 where a clean, adapter-based wiring style is preferred.

Comparison with Similar Components

Compared to standard IEC 61131-3 addition components (such as a classic `ADD` component with discrete inputs), the `AUI_ADD_2_UNGATED` eliminates the need for separate wiring of REQ/CNF events and the actual data lines. This reduces the potential for errors when manually creating event-data associations (WITH connections).

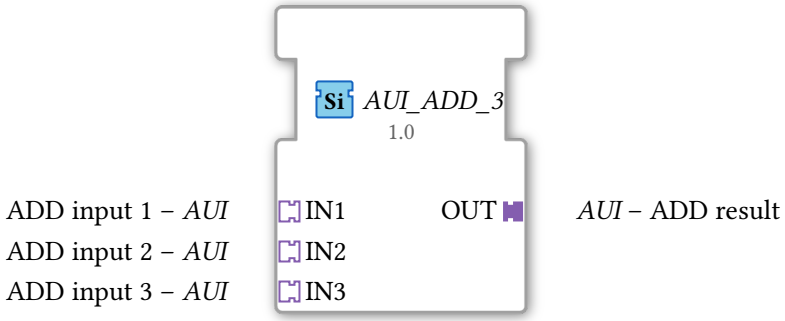
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUI_ADD_2_UNGATED` is a highly efficient, modular component for mathematical addition in modern IEC 61499 systems. Through the consistent use of adapters, he promotes a clear, error-resistant and easily maintainable application design in the 4diac IDE.

AUI_ADD_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_ADD_3` is a generic arithmetic function block for the IEC 61499 development environment (4diac-ide). Its primary function is to add three input values, which are passed via standardized, unidirectional adapter interfaces. The result of the addition is also output via a corresponding adapter. By encapsulating data and control events in adapters, wiring in the application editor is significantly simplified, and the clarity of the system design is improved.

Interface Structure

The interfaces of `AUI_ADD_3` are based entirely on adapters to ensure a clean signal structure.

Event Inputs

No direct event inputs are defined. Event control is handled implicitly via the input adapters (sockets).

Event Outputs

No direct event outputs are defined. Event forwarding is handled implicitly via the output adapter (plug).

Data Inputs

No direct data inputs are defined. Data is provided via the input adapters.

Data Outputs

No direct data outputs are defined. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUI`): First input value (addend 1) for arithmetic addition.
- **IN2** (Type: `adapter::types::unidirectional::AUI`): Second input value (addend 2) for arithmetic addition.
- **IN3** (Type: `adapter::types::unidirectional::AUI`): Third input value (addend 3) for arithmetic addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUI`): Output for the calculated result (sum of `IN1 + IN2 + IN3`).

Functionality

The function block performs three-digit addition according to the following mathematical formula:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.86}$$

As soon as an update event arrives at one of the input adapters (`IN1`, `IN2`, or `IN3`), the function block reads the current values of all three inputs. It then calculates the sum and provides the result at the output adapter `OUT`. Simultaneously, the corresponding update event is passed on via the `OUT` adapter to inform subsequent function blocks of the new value.

Since this is a generic function block (`GEN_AUI_ADD`), the supported data type (e.g., `INT`, `REAL`, `LREAL`) depends on the specific implementation and definition of the adapter type used, `AUI`.

Technical Features

- **Generic Function Block:** The attribute `eclipse4diac::core::GenericClassName = 'GEN_AUI_ADD'` allows the function block to be used flexibly for various numeric data types, provided the adapters used support this.
- **Compact Design:** By using adapters instead of separate event and data ports, the visual "spaghetti code" problem in 4diac systems is minimized.

- **Generic Function Block:** * **Unidirectional Communication:** The use of unidirectional adapters (`unidirectional::AUI`) ensures that the data flow is clearly defined from the inputs to the output.

State Overview

The function block operates in an event-driven and stateless (i.e., purely reactive) manner:

1. **Wait State:** The function block waits for a trigger event at one of the sockets (`IN1`, `IN2`, `IN3`).
2. **Calculation:** Upon receiving an event, the data values are read and added.
3. **Output:** The calculated value is written to the plug `OUT`, and the trigger event is fired.
4. **Return:** The block immediately returns to standby mode.

Application Scenarios

- **Sensor Value Aggregation:** Addition of three analog measured values (e.g., three temperature sensors to determine total heat or three flow meters).

- **Setpoint Generation:** Combining the base setpoint, manual offset, and automatic correction value in process engineering.
- **Power Calculation:** Summing the active power of three individual phases of an electrical network to obtain a total power value.

Comparison with Similar Blocks

- **Standard ADD Block (IEC 61131-3):** A classic `ADD` block has discrete inputs for data and events. `AUI_ADD_3` logically groups these in adapters, which increases reusability and clarity.
- **AUI_ADD_2 (Dual Adder):** While a dual adder would require cascading two function blocks to sum three values, `AUI_ADD_3` accomplishes this in a single step. This saves resources and reduces system latency.

- `AUI_ADD_3_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Change Detection

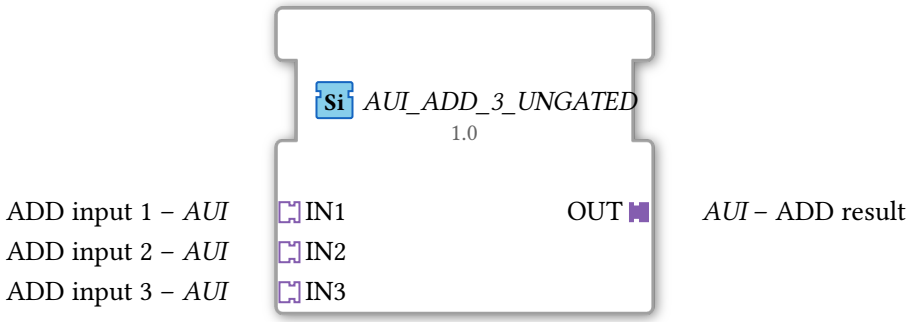
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUI_ADD_3` is an efficient auxiliary function block for arithmetic operations in modern IEC 61499 control applications. Its consistent use of the adapter concept allows it to integrate seamlessly into service-oriented and modular software architectures, contributing to reduced complexity in graphical programming.

AUI_ADD_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_ADD_3_UNGATED` is a generic arithmetic function block for the IEC 61499 development environment (4diac-ide). Its primary function is to add three input values, which are passed via standardized, unidirectional adapter interfaces. The result of the addition is also output via a corresponding adapter. By encapsulating data and control events in adapters, wiring in the application editor is significantly simplified, and the clarity of the system design is improved.

Interface Structure

The interfaces of `AUI_ADD_3_UNGATED` are based entirely on adapters to ensure a clean signal structure.

Event Inputs

No direct event inputs are defined. Event control is handled implicitly via the input adapters (sockets).

Event Outputs

No direct event outputs are defined. Event forwarding is handled implicitly via the output adapter (plug).

Data Inputs

No direct data inputs are defined. Data is provided via the input adapters.

Data Outputs

No direct data outputs are defined. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUI`): First input value (addend 1) for arithmetic addition.
- **IN2** (Type: `adapter::types::unidirectional::AUI`): Second input value (addend 2) for arithmetic addition.
- **IN3** (Type: `adapter::types::unidirectional::AUI`): Third input value (addend 3) for arithmetic addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUI`): Output for the calculated result (sum of `IN1 + IN2 + IN3`).

Functionality

The function block performs three-digit addition according to the following mathematical formula:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.87}$$

As soon as an update event arrives at one of the input adapters (`IN1`, `IN2`, or `IN3`), the function block reads the current values of all three inputs. It then calculates the sum and provides the result at the output adapter `OUT`. Simultaneously, the corresponding update event is passed on via the `OUT` adapter to inform subsequent function blocks of the new value.

Since this is a generic function block (`GEN_AUI_ADD`), the supported data type (e.g., `INT`, `REAL`, `LREAL`) depends on the specific implementation and definition of the adapter type used, `AUI`.

Technical Features

- **Generic Function Block:** The attribute `eclipse4diac::core::GenericClassName = 'GEN_AUI_ADD'` allows the function block to be used flexibly for various numeric data types, provided the adapters used support this.
- **Compact Design:** By using adapters instead of separate event and data ports, the visual "spaghetti code" problem in 4diac systems is minimized.

- **Generic Function Block:** * **Unidirectional Communication:** The use of unidirectional adapters (`unidirectional::AUI`) ensures that the data flow is clearly defined from the inputs to the output.

State Overview

The function block operates in an event-driven and stateless (i.e., purely reactive) manner:

1. **Wait State:** The function block waits for a trigger event at one of the sockets (`IN1`, `IN2`, `IN3`).
2. **Calculation:** Upon receiving an event, the data values are read and added.
3. **Output:** The calculated value is written to the plug `OUT`, and the trigger event is fired.
4. **Return:** The block immediately returns to standby mode.

Application Scenarios

- **Sensor Value Aggregation:** Addition of three analog measured values (e.g., three temperature sensors to determine total heat or three flow meters).
- **Setpoint Generation:** Combining the base setpoint, manual offset, and automatic correction value in process engineering.
- **Power Calculation:** Summing the active power of three individual phases of an electrical network to obtain a total power value.

Comparison with Similar Blocks

- **Standard ADD Block (IEC 61131-3):** A classic `ADD` block has discrete inputs for data and events. `AUI_ADD_3_UNGATED` logically groups these in adapters, which increases reusability and clarity.
- **AUI_ADD_2 (Dual Adder):** While a dual adder would require cascading two function blocks to sum three values, `AUI_ADD_3_UNGATED` accomplishes this in a single step. This saves resources and reduces system latency.

- **`AUI_ADD_3`**: The gated variant – updates the output only on an actual value change.

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUI_ADD_3_UNGATED` is an efficient auxiliary function block for arithmetic operations in modern IEC 61499 control applications. Its consistent use of the adapter concept allows it to integrate seamlessly into service-oriented and modular software architectures, contributing to reduced complexity in graphical programming.

AUI_ADD_4

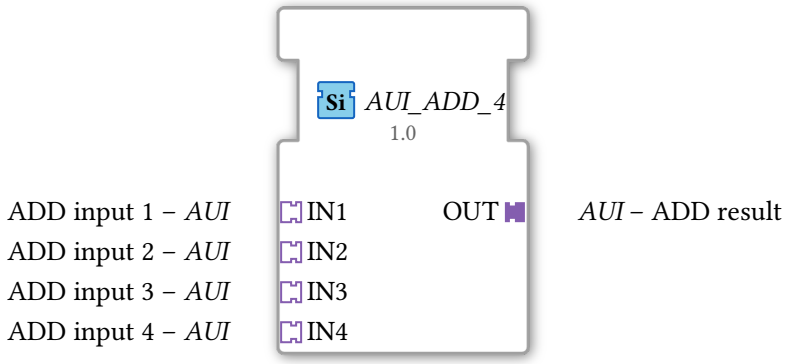
+———+ Trigger (Event an IN1..IN4) +—————+ | | ————————————> | | | IDLE | | BERECHNUNG & | | (Wartend) | <———————————— | AKTUALISIERUNG | +———+ Berechnung abgeschlossen +—————+

1. **IDLE (Default State):** The function block passively waits for incoming data changes at the sockets.
2. **CALCULATION:** An event at one of the inputs triggers the summation. The result is calculated, passed to `OUT`, and an output event is generated. The function block then immediately returns to the state `IDLE`.

- **Sensor Data Fusion:** Summing up to four analog measured values (e.g., determining the total volume flow from four individual partial flows).
- **Power Calculation:** Adding the active power of four separate loads to determine the total power in an energy monitoring system.
- **Setpoint Generation:** Combining a base setpoint with up to three correction or offset values in control systems.
- **Comparison with Standard `ADD` (IEC 61131-3):** Classic `ADD` function blocks use explicit data and event pins. `AUI_ADD_4` bundles these signals in adapters, simplifying and standardizing wiring.
- **Comparison with `AUI_ADD_2`:** Adding three or four values with a dual adder would require cascading multiple function blocks. `AUI_ADD_4` saves system resources, reduces latency, and results in a cleaner application layout.

The `AUI_ADD_4` is a practical utility function block for advanced control projects in the 4diac IDE. It combines the basic mathematical function of addition with the structural advantages of modern adapter connections and is ideally suited for clean, clear signal processing chains.

Introduction



Interface Structure

Event Inputs

Event Outputs

Data Inputs

Data Outputs

Adapter

Sockets (Eingangs-Schnittstellen)

Plugs (Ausgangs-Schnittstellen)

Functionality

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AUI_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_ADD_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

+——————+ Trigger (Event an IN1..IN4) +————————+ | | ————————————————————> | | | IDLE | | BERECHNUNG & | | (Wartend) | <———————————————— | AKTUALISIERUNG | +——————+ Berechnung abgeschlossen +————————+

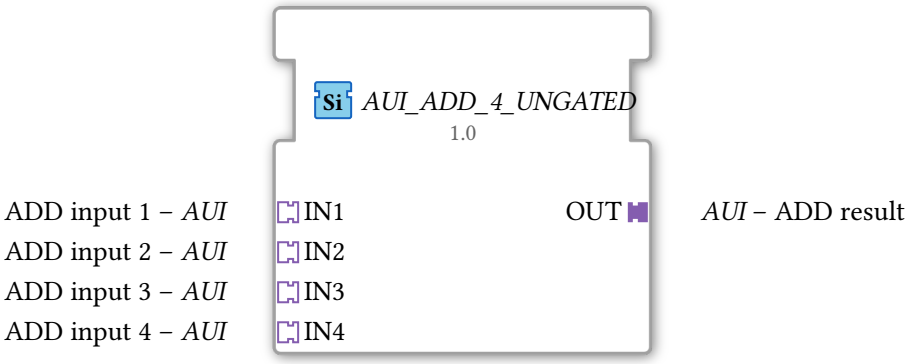
1. **IDLE (Default State):** The function block passively waits for incoming data changes at the sockets.

2. **CALCULATION:** An event at one of the inputs triggers the summation. The result is calculated, passed to `OUT`, and an output event is generated. The function block then immediately returns to the state `IDLE`.

- **Sensor Data Fusion:** Summing up to four analog measured values (e.g., determining the total volume flow from four individual partial flows).
- **Power Calculation:** Adding the active power of four separate loads to determine the total power in an energy monitoring system.
- **Setpoint Generation:** Combining a base setpoint with up to three correction or offset values in control systems.
- **Comparison with Standard `ADD` (IEC 61131-3):** Classic `ADD` function blocks use explicit data and event pins. `AUI_ADD_4_UNGATED` bundles these signals in adapters, simplifying and standardizing wiring.
- **Comparison with `AUI_ADD_2`:** Adding three or four values with a dual adder would require cascading multiple function blocks. `AUI_ADD_4_UNGATED` saves system resources, reduces latency, and results in a cleaner application layout.

The `AUI_ADD_4_UNGATED` is a practical utility function block for advanced control projects in the 4diac IDE. It combines the basic mathematical function of addition with the structural advantages of modern adapter connections and is ideally suited for clean, clear signal processing chains.

Introduction



Interface Structure

Event Inputs

Event Outputs

Data Inputs

Data Outputs

Adapter

Sockets (Eingangs-Schnittstellen)

Plugs (Ausgangs-Schnittstellen)

Functionality

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

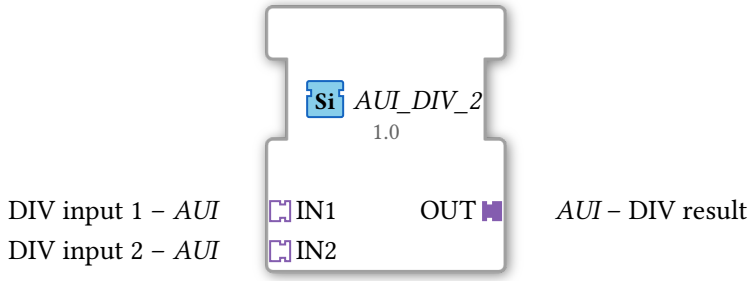
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AUI_DIV_2

```
  +----------------------------------+
| AUI_DIV_2 | +—————————————+ =====>| [AUI] IN1 OUT [AUI]|=====> =====>| [AUI] IN2 | +
—————————————+
```



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_DIV_2` is a generic function block for performing an arithmetic division of two input values. The block uses an adapter-based concept, where data and event streams are bundled in unidirectional adapters. This enables clean, modularized, and clear modeling within IEC 61499 applications in 4diac.

Interface Structure

Event Inputs

No direct event inputs are defined. Event control is implicit via the adapter interfaces.

Event Outputs

No direct event outputs are defined. Event control is implicit via the adapter interfaces.

Data Inputs

No direct data inputs are defined. Data is transferred via the input adapters.

Data Outputs

No direct data outputs are defined. Data is transferred via the output adapter.

Adapters

- **IN1** (Type: `adapter::types::unidirectional::AUI`):

The first input adapter. It provides the dividend (numerator) for the division operation and the associated trigger events.

- **IN2** (Type: `adapter::types::unidirectional::AUI`):

The second input adapter. It provides the divisor (denominator) for the division operation and the associated trigger events.

- **OUT** (Type: `adapter::types::unidirectional::AUI`):

The output adapter. It outputs the result of the division (quotient) and the corresponding calculation event.

Plugs (Ausgangs-Schnittstellen)

Sockets (Eingangs-Schnittstellen)

Functionality

The function block performs the mathematical operation `OUT = IN1 / IN2`.

As soon as a corresponding trigger event is registered at the input adapters (`IN1` or `IN2`), the function block reads the current values from the adapters, calculates the quotient, and outputs it along with an output event via the adapter `OUT`.

Since it is a generic function block (GEN_AUI_DIV), the function block can adapt to different numeric data types (e.g., INT, REAL, LREAL) depending on the implementation and system configuration, provided the underlying adapters support the same data type.

- **Generic Type:** The attribute GenericClassName with the value 'GEN_AUI_DIV' allows the function block to be used flexibly and is not restricted to a specific data type.
- **Interface Encapsulation:** The use of unidirectional adapters (AUI) significantly reduces wiring effort in the 4diac IDE, as event and data lines are bundled into a single connection.
- **Division by Zero:** When using this function block, it must be ensured that the value of IN2 is not zero to avoid runtime errors or infinite values (NaN/Inf) in the control system.

2.2 · Status Overview

1. **Waiting State (IDLE):** The function block waits for incoming events via the adapters IN1 or IN2.
2. **Calculation (EVALUATE):** After receiving an event, the data from the adapters is read and the division is performed.
3. **Output (DISPATCH):** The result and the associated event are provided via the adapter OUT.

- **Scaling of Sensor Values:** Division of raw values by scaling factors in modular control architectures.
- **Calculation of Averages:** Use in mathematical computing networks where data streams are transmitted in a structured manner via adapters.
- **Signal Processing:** Reduction of signal strength or frequency divisions in distributed systems.

Compared to a classic standard DIV function block (e.g., from the IEC 61131-3 standard library), which uses individual pins for REQ, CNF, IN1, IN2, and OUT, the AUI_DIV_2 offers a significantly cleaner visual representation in the function block diagram. Encapsulation in adapters minimizes the potential for errors when manually wiring event-to-data mappings.

The AUI_DIV_2 is a modern, modular, and reusable function block for division. Through the consistent use of unidirectional adapters, it is ideally suited for complex, distributed control applications in 4diac where clarity and easy maintainability of the code are paramount.

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

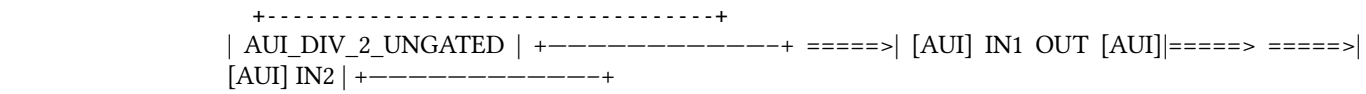
Change Detection

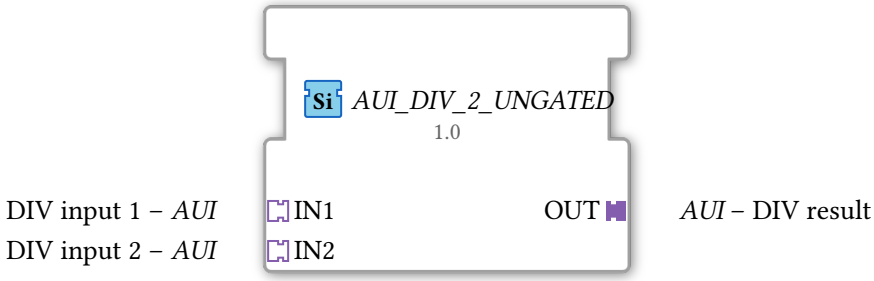
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AUI_DIV_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUI_DIV_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.





• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_DIV_2_UNGATED` is a generic function block for performing an arithmetic division of two input values. The block uses an adapter-based concept, where data and event streams are bundled in unidirectional adapters. This enables clean, modularized, and clear modeling within IEC 61499 applications in 4diac.

Interface Structure

Event Inputs

No direct event inputs are defined. Event control is implicit via the adapter interfaces.

Event Outputs

No direct event outputs are defined. Event control is implicit via the adapter interfaces.

Data Inputs

No direct data inputs are defined. Data is transferred via the input adapters.

Data Outputs

No direct data outputs are defined. Data is transferred via the output adapter.

Adapters

- **IN1** (Type: `adapter::types::unidirectional::AUI`):

The first input adapter. It provides the dividend (numerator) for the division operation and the associated trigger events.

- **IN2** (Type: `adapter::types::unidirectional::AUI`):

The second input adapter. It provides the divisor (denominator) for the division operation and the associated trigger events.

- **OUT** (Type: `adapter::types::unidirectional::AUI`):

The output adapter. It outputs the result of the division (quotient) and the corresponding calculation event.

Plugs (Ausgangs-Schnittstellen)

Sockets (Eingangs-Schnittstellen)

Functionality

The function block performs the mathematical operation `OUT = IN1 / IN2`.

As soon as a corresponding trigger event is registered at the input adapters (`IN1` or `IN2`), the function block reads the current values from the adapters, calculates the quotient, and outputs it along with an output event via the adapter `OUT`.

Since it is a generic function block (`GEN_AUI_DIV`), the function block can adapt to different numeric data types (e.g., INT, REAL, LREAL) depending on the implementation and system configuration, provided the underlying adapters support the same data type.

- **Generic Type:** The attribute `GenericClassName` with the value `'GEN_AUI_DIV'` allows the function block to be used flexibly and is not restricted to a specific data type.

- **Interface Encapsulation:** The use of unidirectional adapters (`AUI`) significantly reduces wiring effort in the 4diac IDE, as event and data lines are bundled into a single connection.
- **Division by Zero:** When using this function block, it must be ensured that the value of `IN2` is not zero to avoid runtime errors or infinite values (NaN/Inf) in the control system.

2.3 · Status Overview

1. **Waiting State (IDLE):** The function block waits for incoming events via the adapters `IN1` or `IN2`.
2. **Calculation (EVALUATE):** After receiving an event, the data from the adapters is read and the division is performed.
3. **Output (DISPATCH):** The result and the associated event are provided via the adapter `OUT`.

- **Scaling of Sensor Values:** Division of raw values by scaling factors in modular control architectures.
- **Calculation of Averages:** Use in mathematical computing networks where data streams are transmitted in a structured manner via adapters.
- **Signal Processing:** Reduction of signal strength or frequency divisions in distributed systems.

Compared to a classic standard DIV function block (e.g., from the IEC 61131-3 standard library), which uses individual pins for `REQ`, `CNF`, `IN1`, `IN2`, and `OUT`, the `AUI_DIV_2_UNGATED` offers a significantly cleaner visual representation in the function block diagram. Encapsulation in adapters minimizes the potential for errors when manually wiring event-to-data mappings.

The `AUI_DIV_2_UNGATED` is a modern, modular, and reusable function block for division. Through the consistent use of unidirectional adapters, it is ideally suited for complex, distributed control applications in 4diac where clarity and easy maintainability of the code are paramount.

Technical Features

State Overview

Application Scenarios

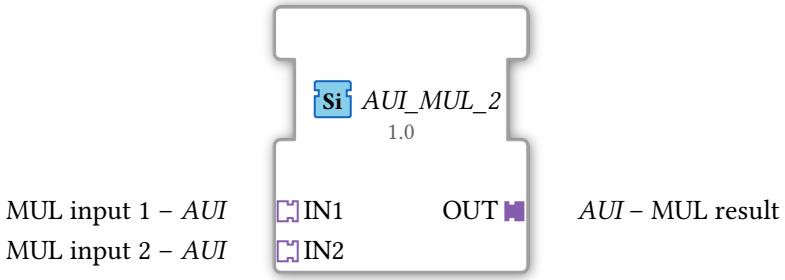
Comparison with Similar Function Blocks

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AUI_MUL_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_MUL_2` is a generic arithmetic function block for the 4diac-ide development environment, used to perform the multiplication of two input values.

The special feature of this block lies in its adapter-based architecture. Instead of classic, individual data and event lines, this block uses unidirectional adapters (`AUI`) to transmit data and the associated control events in encapsulated form. This significantly reduces the visual complexity of control applications.

Interface Structure

Event Inputs
No direct event inputs are available. Event control is handled entirely via the input adapters.

Event Outputs
No direct event outputs are available. Event forwarding is handled entirely via the output adapter.

Data Inputs
No direct data inputs available.

Data Outputs
No direct data outputs available.

Adapters
- **`IN1` (Type: `adapter::types::unidirectional::AUI` - Socket):**

The first input adapter, providing the first multiplication factor.

- **`IN2` (Type: `adapter::types::unidirectional::AUI` - Socket):**

The second input adapter, providing the second multiplication factor.

- **`OUT` (Type: `adapter::types::unidirectional::AUI` - Plug):**

The output adapter that outputs the calculated product of the two input values along with the corresponding update event.

Functionality
As soon as an event is triggered at one of the input adapters (`IN1` or `IN2`) indicating the presence of a new value, the function block reads the current values from both adapters.

The calculation is performed using the classic mathematical formula:
`OUT = IN1 * IN2`
After successful calculation, the result is passed to the output adapter `OUT`, and an output event is simultaneously triggered to inform subsequent function blocks about the presence of a new value.

Technical Features
- **Generic Behavior:** The function block is linked to the value `'GEN_AUI_MUL'` via the compiler attribute `GenericClassName`. This means that the function block can operate independently of the data type (e.g., for `INT`, `REAL`, `LREAL`, etc.), depending on the specific data type defined by the connected adapters.
- **Efficient Wiring:** By encapsulating the function block in `AUI` adapters (Unidirectional Interface Adapters), the number of connections in the function plan is drastically reduced, as data and event flows are combined.
-

State Overview
The function block operates purely event-driven and stateless (reactively):

1. **Standby State:** The block waits for incoming events via `IN1` or `IN2`.
2. **Calculation Phase:** An incoming event triggers the internal multiplication of the current adapter data.
3. **Output Phase:** The calculated product is applied to `OUT`, and the output event is triggered. The block immediately returns to the standby state.

–

Application Scenarios
- **Measurement Scaling:** Multiplication of an analog sensor value (e.g., voltage) by a scaling factor to calculate the actual physical value.
- **Power Calculation:** Continuous multiplication of current (I) and voltage (U) to determine the current active power (P). * **Signal Amplification:** Application in signal processing for simple level adjustment.

Comparison with Similar Components

- **Standard `MUL` component (IEC 61131-3):** A standard multiplication component requires separate pins for each data signal as well as for the execution control (`REQ`/`CNF`). `AUI_MUL_2` solves this more elegantly and compactly using adapter connections.
- **Multi-Input Multipliers (`MUL_3`, `MUL_4`):** While function blocks like `AUI_MUL_2` are fixed to two inputs, multi-input function blocks allow the multiplication of three or more factors in a single step, but are more complex.

Change Detection

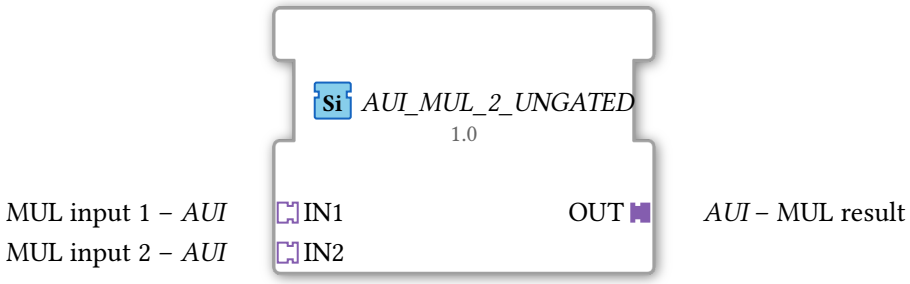
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUI_MUL_2` is a highly modular and modern function block for the 4diac-ide. Through the consistent use of unidirectional adapters, it promotes clean software design in distributed control systems according to IEC 61499, without sacrificing flexibility in the data types it can process.

AUI_MUL_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_MUL_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_MUL_2_UNGATED` is a generic arithmetic function block for the 4diac-ide development environment, used to perform the multiplication of two input values.

The special feature of this block lies in its adapter-based architecture. Instead of classic, individual data and event lines, this block uses unidirectional adapters (`AUI`) to transmit data and the associated control events in encapsulated form. This significantly reduces the visual complexity of control applications.

Interface Structure

Event Inputs

No direct event inputs are available. Event control is handled entirely via the input adapters.

Event Outputs

No direct event outputs are available. Event forwarding is handled entirely via the output adapter.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

- **`IN1` (Type: `adapter::types::unidirectional::AUI` - Socket):**

The first input adapter, providing the first multiplication factor.

- **`IN2` (Type: `adapter::types::unidirectional::AUI` - Socket):**

The second input adapter, providing the second multiplication factor.

- **`OUT` (Type: `adapter::types::unidirectional::AUI` - Plug):**

The output adapter that outputs the calculated product of the two input values along with the corresponding update event.

Functionality

As soon as an event is triggered at one of the input adapters (`IN1` or `IN2`) indicating the presence of a new value, the function block reads the current values from both adapters.

The calculation is performed using the classic mathematical formula:

```
OUT = IN1 * IN2
```

After successful calculation, the result is passed to the output adapter `OUT`, and an output event is simultaneously triggered to inform subsequent function blocks about the presence of a new value.

Technical Features

- **Generic Behavior:** The function block is linked to the value `'GEN_AUI_MUL'` via the compiler attribute `GenericClassName`. This means that the function block can operate independently of the data type (e.g., for `INT`, `REAL`, `LREAL`, etc.), depending on the specific data type defined by the connected adapters.
- **Efficient Wiring:** By encapsulating the function block in `AUI` adapters (Unidirectional Interface Adapters), the number of connections in the function plan is drastically reduced, as data and event flows are combined.

-

State Overview

The function block operates purely event-driven and stateless (reactively):

1. **Standby State:** The block waits for incoming events via `IN1` or `IN2`.
2. **Calculation Phase:** An incoming event triggers the internal multiplication of the current adapter data.
3. **Output Phase:** The calculated product is applied to `OUT`, and the output event is triggered. The block immediately returns to the standby state.

–

Application Scenarios

- **Measurement Scaling:** Multiplication of an analog sensor value (e.g., voltage) by a scaling factor to calculate the actual physical value.
- **Power Calculation:** Continuous multiplication of current (I) and voltage (U) to determine the current active power (P). * **Signal Amplification:** Application in signal processing for simple level adjustment.

Comparison with Similar Components

- **Standard `MUL` component (IEC 61131-3):** A standard multiplication component requires separate pins for each data signal as well as for the execution control (`REQ`/`CNF`). `AUI_MUL_2_UNGATED` solves this more elegantly and compactly using adapter connections.
- **Multi-Input Multipliers (`MUL_3`, `MUL_4`):** While function blocks like `AUI_MUL_2_UNGATED` are fixed to two inputs, multi-input function blocks allow the multiplication of three or more factors in a single step, but are more complex.

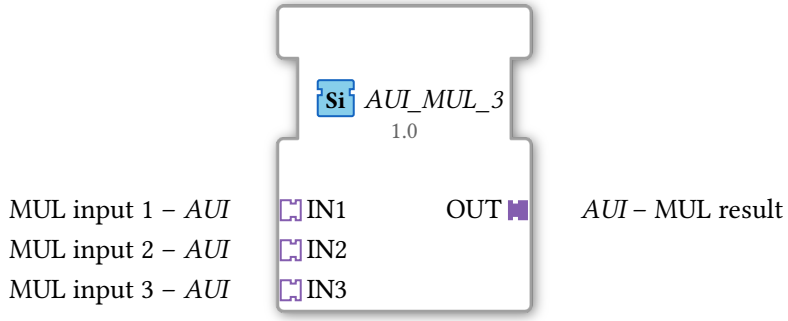
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUI_MUL_2_UNGATED` is a highly modular and modern function block for the 4diac-ide. Through the consistent use of unidirectional adapters, it promotes clean software design in distributed control systems according to IEC 61499, without sacrificing flexibility in the data types it can process.

AUI_MUL_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_MUL_3` is used for the mathematical multiplication of three input values. It is a generic function block (Generic FB) based on the higher-level class `GEN_AUI_MUL`. The special feature of this block lies in the use of unidirectional adapters of type `AUI` (Adapter Unidirectional Interface), which encapsulate both the data and the associated events. This ensures a clear and modular structure within 4diac applications.

Interface Structure

Event Inputs

This function block does not have direct, dedicated event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

This function block does not have direct, dedicated event outputs. Event forwarding is handled via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. Data output is handled via the output adapter.

Adapters

Sockets (Input Interfaces)

- **IN1** (Type: `adapter::types::unidirectional::AUI`): First multiplicand.
- **IN2** (Type: `adapter::types::unidirectional::AUI`): Second multiplicand.
- **IN3** (Type: `adapter::types::unidirectional::AUI`): Third multiplicand.

Plugs (Output Interface)

- **OUT** (Type: `adapter::types::unidirectional::AUI`): Result of the multiplication ($OUT = IN1 \cdot IN2 \cdot IN3$).

Functionality

As soon as a new event is signaled at one of the input adapters (`IN1`, `IN2`, or `IN3`), the function block reads the current values of the three inputs.

The arithmetic multiplication is calculated as follows:

$$Result = Value(IN1) \cdot Value(IN2) \cdot Value(IN3) \tag{2.88}$$

The result is passed to the output adapter `OUT`, and a corresponding output event is triggered via this adapter to inform subsequent function blocks of the update.

Since the function block is declared as generic (`GEN_AUI_MUL`), it adapts flexibly to the data types of the connected adapters (e.g., `INT`, `REAL`, `LREAL`), as long as these are supported by the adapter type used.

Technical Features

- **Generic Type:** By defining it as a generic function block, it can be flexibly used for various numeric data types without having to create a separate function block for each type.
- **Encapsulation by Adapters:** The use of the unidirectional `AUI` adapters significantly reduces the number of visible connection lines in the 4diac editor, as data and trigger events are bundled in a single connection.

–

State Overview

The function block operates purely event-driven:

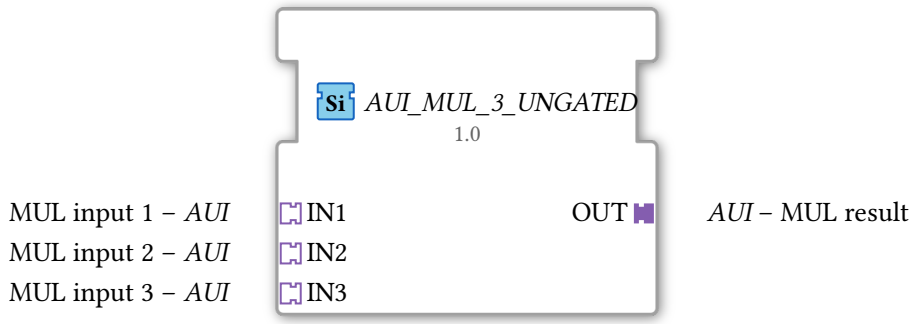
1. **Wait State:** The function block waits for a trigger event on one of the sockets (`IN1`, `IN2`, `IN3`).
2. **Calculation:** After an event occurs, the data is read and multiplied.

– 3. **Output:** The result is applied to the plug `OUT`, the plug's send event is triggered, and the function block returns to standby mode.

Application Scenarios

- **Volume Calculation:** Calculation of a three-dimensional volume (length \cdot \cdot \bdot \cdot \bdot \bdot \b

frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_MUL_3_UNGATED` is used for the mathematical multiplication of three input values. It is a generic function block (Generic FB) based on the higher-level class `GEN_AUI_MUL`. The special feature of this block lies in the use of unidirectional adapters of type `AUI` (Adapter Unidirectional Interface), which encapsulate both the data and the associated events. This ensures a clear and modular structure within 4diac applications.

Interface Structure

Event Inputs

This function block does not have direct, dedicated event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

This function block does not have direct, dedicated event outputs. Event forwarding is handled via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. Data output is handled via the output adapter.

Adapters

Sockets (Input Interfaces)

- **IN1** (Type: `adapter::types::unidirectional::AUI`): First multiplicand.
- **IN2** (Type: `adapter::types::unidirectional::AUI`): Second multiplicand.
- **IN3** (Type: `adapter::types::unidirectional::AUI`): Third multiplicand.

Plugs (Output Interface)

- **OUT** (Type: `adapter::types::unidirectional::AUI`): Result of the multiplication ($OUT = IN1 \cdot IN2 \cdot IN3$).

Functionality

As soon as a new event is signaled at one of the input adapters (`IN1`, `IN2`, or `IN3`), the function block reads the current values of the three inputs.

The arithmetic multiplication is calculated as follows:

$$Result = Value(IN1) \cdot Value(IN2) \cdot Value(IN3) \tag{2.89}$$

The result is passed to the output adapter `OUT`, and a corresponding output event is triggered via this adapter to inform subsequent function blocks of the update.

Since the function block is declared as generic (`GEN_AUI_MUL`), it adapts flexibly to the data types of the connected adapters (e.g., `INT`, `REAL`, `LREAL`), as long as these are supported by the adapter type used.

Technical Features

- **Generic Type:** By defining it as a generic function block, it can be flexibly used for various numeric data types without having to create a separate function block for each type.
- **Encapsulation by Adapters:** The use of the unidirectional `AUI` adapters significantly reduces the number of visible connection lines in the 4diac editor, as data and trigger events are bundled in a single connection.

–

State Overview

The function block operates purely event-driven:

1. **Wait State:** The function block waits for a trigger event on one of the sockets (`IN1`, `IN2`, `IN3`).
2. **Calculation:** After an event occurs, the data is read and multiplied.

– 3. **Output:** The result is applied to the plug `OUT`, the plug's send event is triggered, and the function block returns to standby mode.

Application Scenarios

- **Volume Calculation:** Calculation of a three-dimensional volume (length \cdot \cdot \bdot \cdot \bdot \bdot \b

No image available

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_MUL_4` is a generic block for performing an arithmetic multiplication of four input values. It uses unidirectional adapters of type `AUI` (Adapter Unit Interface) to transmit data and associated events in a bundled manner. This enables clean and clear modeling in the 4diac-ide.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is handled via the adapter interfaces.

Event Outputs

There are no direct event outputs. Event control is handled via the adapter interfaces.

Data Inputs

There are no direct data inputs.

Data Outputs

There are no direct data outputs.

Adapters

Sockets (Inputs)

- **IN1** (Type: `adapter::types::unidirectional::AUI`): First multiplicand.
- **IN2** (Type: `adapter::types::unidirectional::AUI`): Second multiplicand.
- **IN3** (Type: `adapter::types::unidirectional::AUI`): Third multiplicand.
- **IN4** (Type: `adapter::types::unidirectional::AUI`): Fourth multiplicand.

Plugs (Outputs)

- **OUT** (Type: `adapter::types::unidirectional::AUI`): Output for the result of the multiplication ($OUT = IN1 \cdot IN2 \cdot IN3 \cdot IN4$).

Functionality

As soon as a new event is signaled at one of the input adapters (`IN1` to `IN4`), the function block reads the current values of the four inputs. These values are multiplied together:

$$\text{Result} = \text{Value of IN1} \cdot \text{Value of IN2} \cdot \text{Value of IN3} \cdot \text{Value of IN4} \tag{2.90}$$

The calculated result is passed to the output adapter `OUT`, and a corresponding update event is output via this plug.

Technical Features

- **Generic Module:** The module is based on the generic class `GEN_AUI_MUL`. This allows for flexible adaptation to different data types within the adapter type used.
- **Using Adapters:** By encapsulating data and events in unidirectional adapters (`AUI`), the number of connection lines in the 4diac-ide application editor is drastically reduced, significantly improving the clarity of complex applications.

State Overview

The function block operates purely event-driven and has no internal state that persists beyond the duration of a calculation cycle (stateless/static function block).

- **Wait State:** The function block waits for an update event at one of the sockets (`IN1` to `IN4`).
- **Calculation:** After an event arrives, the multiplication is performed.
- **Output:** The result is created at `OUT`, an event is triggered at the plug, and the function block immediately returns to the wait state.

Application Scenarios

- **Signal Scaling:** Cascaded scaling of measured values where a raw value must be multiplied by several correction and calibration factors.
- **Signal Scaling:** * **Power Calculation:** Multiplication of various physical quantities to determine a target value (e.g., calculating total power from different factors).
- **Modular Control Concepts:** Applications where structured data transfers are implemented using standardized adapter interfaces.

Comparison with Similar Function Blocks

- **Standard MUL Function Blocks (IEC 61131-3):** Classic multipliers work with direct elementary data types (such as `REAL` or `INT`) and require separate event inputs (`REQ`) and event outputs (`CNF`). `AUI_MUL_4` simplifies this by encapsulating the data in adapters.
- **AUI_MUL_2 / AUI_MUL_3:** Similar function blocks with fewer inputs. The function block `AUI_MUL_4` is particularly suitable when exactly four factors need to be multiplied without having to chain multiple multiplication blocks together (which saves resources and latency in the execution model).

Change Detection

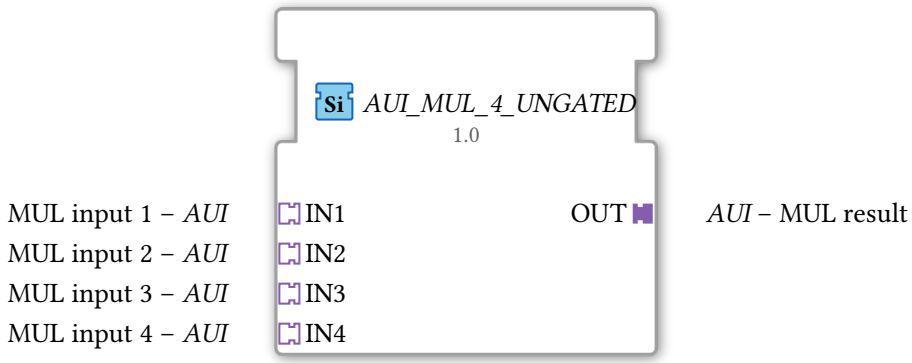
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The function block `AUI_MUL_4` represents an efficient and clean solution for arithmetic multiplications of four factors within an adapter-based 4diac architecture. It promotes the modular design and clarity of the IEC 61499 application.

AUI_MUL_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_MUL_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



No image available

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_MUL_4_UNGATED` is a generic block for performing an arithmetic multiplication of four input values. It uses unidirectional adapters of type `AUI` (Adapter Unit Interface) to transmit data and associated events in a bundled manner. This enables clean and clear modeling in the 4diac-ide.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is handled via the adapter interfaces.

Event Outputs

There are no direct event outputs. Event control is handled via the adapter interfaces.

Data Inputs

There are no direct data inputs.

Data Outputs

There are no direct data outputs.

Adapters

Sockets (Inputs)

- **IN1** (Type: `adapter::types::unidirectional::AUI`): First multiplicand.
- **IN2** (Type: `adapter::types::unidirectional::AUI`): Second multiplicand.
- **IN3** (Type: `adapter::types::unidirectional::AUI`): Third multiplicand.
- **IN4** (Type: `adapter::types::unidirectional::AUI`): Fourth multiplicand.

Plugs (Outputs)

- **OUT** (Type: `adapter::types::unidirectional::AUI`): Output for the result of the multiplication ($OUT = IN1 \cdot IN2 \cdot IN3 \cdot IN4$).

Functionality

As soon as a new event is signaled at one of the input adapters (`IN1` to `IN4`), the function block reads the current values of the four inputs. These values are multiplied together:

$$\text{Result} = \text{Value of IN1} \cdot \text{Value of IN2} \cdot \text{Value of IN3} \cdot \text{Value of IN4} \tag{2.91}$$

The calculated result is passed to the output adapter `OUT`, and a corresponding update event is output via this plug.

Technical Features

- **Generic Module:** The module is based on the generic class `GEN_AUI_MUL`. This allows for flexible adaptation to different data types within the adapter type used.
- **Using Adapters:** By encapsulating data and events in unidirectional adapters (`AUI`), the number of connection lines in the 4diac-ide application editor is drastically reduced, significantly improving the clarity of complex applications.

State Overview

The function block operates purely event-driven and has no internal state that persists beyond the duration of a calculation cycle (stateless/static function block).

- **Wait State:** The function block waits for an update event at one of the sockets (`IN1` to `IN4`).
- **Calculation:** After an event arrives, the multiplication is performed.
- **Output:** The result is created at `OUT`, an event is triggered at the plug, and the function block immediately returns to the wait state.

Application Scenarios

- **Signal Scaling:** Cascaded scaling of measured values where a raw value must be multiplied by several correction and calibration factors.
- **Signal Scaling:** * **Power Calculation:** Multiplication of various physical quantities to determine a target value (e.g., calculating total power from different factors).
- **Modular Control Concepts:** Applications where structured data transfers are implemented using standardized adapter interfaces.

Comparison with Similar Function Blocks

- **Standard MUL Function Blocks (IEC 61131-3):** Classic multipliers work with direct elementary data types (such as `REAL` or `INT`) and require separate event inputs (`REQ`) and event outputs (`CNF`). `AUI_MUL_4_UNGATED` simplifies this by encapsulating the data in adapters.
- **AUI_MUL_2 / AUI_MUL_3:** Similar function blocks with fewer inputs. The function block `AUI_MUL_4_UNGATED` is particularly suitable when exactly four factors need to be multiplied without having to chain multiple multiplication blocks together (which saves resources and latency in the execution model).

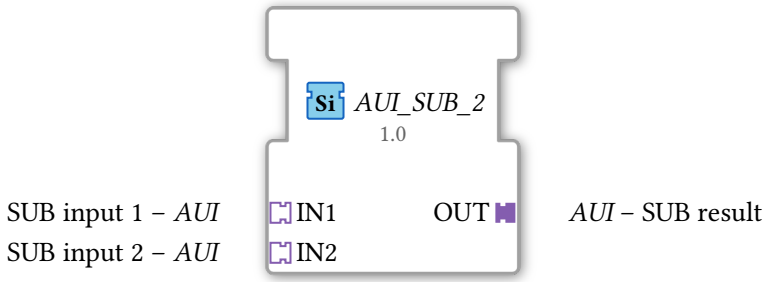
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The function block `AUI_MUL_4_UNGATED` represents an efficient and clean solution for arithmetic multiplications of four factors within an adapter-based 4diac architecture. It promotes the modular design and clarity of the IEC 61499 application.

AUI_SUB_2



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_SUB_2` is used to perform arithmetic subtraction within an IEC 61499 application. It is a generic function block (FB) specifically designed for interaction with adapter interfaces. It allows subtraction operations to be implemented modularly and in a structured manner via standardized adapter connections, instead of using discrete individual wires for data and events.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Event control is handled internally via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Event forwarding is encapsulated via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

Sockets (Inputs)

- **IN1** (Type: `adapter::types::unidirectional::AUI`):

First input adapter for subtraction. This represents the minuend (value from which the subtraction is made).

- **IN2** (Type: `adapter::types::unidirectional::AUI`):

Second input adapter for subtraction. This represents the subtrahend (value being subtracted).

Plugs (Inputs / Outputs)

- **OUT** (Type: `adapter::types::unidirectional::AUI`):

Output adapter that provides the result of the subtraction (the difference) for further use.

–

Functionality

The function block performs the basic mathematical operation of subtraction on the values received via the adapters:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.92}$$

As soon as a change in value is signaled at the input adapters `IN1` or `IN2` (triggered by the event-driven nature of the underlying `AUI` adapter), the function block calculates the difference and outputs the result, along with a corresponding update event, via the output adapter `OUT`.

Technical Features

- **Generic Type:** The function block is based on the generic class `GEN_AUI_SUB`. This allows for flexible handling of different data types, provided they are supported by the underlying adapter type `AUI` (Analog User Interface / Unidirectional).
- **Encapsulation:** By using unidirectional adapters, signal flows and their associated trigger events are neatly bundled. This significantly simplifies application design in the 4diac IDE and reduces the number of visible connection lines.

–

State Overview

The function block itself does not manage a complex internal state (stateless in the sense of a state machine). Its execution is purely reactive:

1. **Waiting:** The function block waits for an update event at `IN1` or `IN2`.

Encapsulation: 2. **Calculation:** After an event occurs, the data values are read and the subtraction is performed.

1. **Output:** The new difference value is applied to `OUT`, and the adapter's output event is triggered.

Application Scenarios

- **Differential Pressure/Differential Temperature Measurement:** Calculation of the deviation between two analog sensors whose values are already available as structured adapter signals.
- **Setpoint-Actual Value Comparison:** Subtraction of an actual value from a setpoint to determine the control deviation in control loops.
- **Zero Point Compensation (Offset Calculation):** Subtraction of a static or dynamic offset value (via `IN2`) from a raw signal (via `IN1`).
-

Comparison with Similar Building Blocks

Compared to a standard subtraction block (such as the classic `SUB` block from the IEC 61131-3 library), the `AUI_SUB_2` does not require explicit wiring of separate event and data ports (e.g., `REQ`, `IN1`, and `IN2`). All the logic for value transfer and updates is encapsulated in the `AUI` adapter. This makes the `AUI_SUB_2` easier to maintain and more robust against wiring errors in complex system architectures.

Change Detection

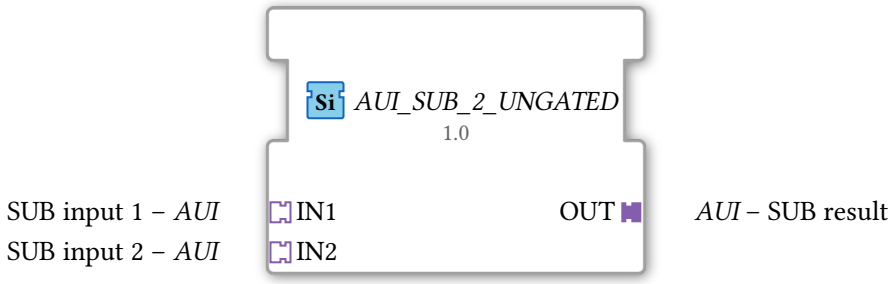
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUI_SUB_2` is a highly efficient, clear, and modern function block for performing subtractions. Through the consistent use of adapters, it integrates seamlessly into service-oriented and modularized control concepts and contributes to a clean software design in the 4diac IDE.

AUI_SUB_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_SUB_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUI_SUB_2_UNGATED` is used to perform arithmetic subtraction within an IEC 61499 application. It is a generic function block (FB) specifically designed for interaction with adapter interfaces. It allows subtraction operations to be implemented modularly and in a structured manner via standardized adapter connections, instead of using discrete individual wires for data and events.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Event control is handled internally via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Event forwarding is encapsulated via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

Sockets (Inputs)

- **IN1** (Type: `adapter::types::unidirectional::AUI`):

First input adapter for subtraction. This represents the minuend (value from which the subtraction is made).

- **IN2** (Type: `adapter::types::unidirectional::AUI`):

Second input adapter for subtraction. This represents the subtrahend (value being subtracted).

Plugs (Inputs / Outputs)

- **OUT** (Type: `adapter::types::unidirectional::AUI`):

Output adapter that provides the result of the subtraction (the difference) for further use.
–

Functionality

The function block performs the basic mathematical operation of subtraction on the values received via the adapters:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.93}$$

As soon as a change in value is signaled at the input adapters `IN1` or `IN2` (triggered by the event-driven nature of the underlying `AUI` adapter), the function block calculates the difference and outputs the result, along with a corresponding update event, via the output adapter `OUT`.

Technical Features

- **Generic Type:** The function block is based on the generic class `GEN_AUI_SUB`. This allows for flexible handling of different data types, provided they are supported by the underlying adapter type `AUI` (Analog User Interface / Unidirectional).
- **Encapsulation:** By using unidirectional adapters, signal flows and their associated trigger events are neatly bundled. This significantly simplifies application design in the 4diac IDE and reduces the number of visible connection lines.

–

State Overview

The function block itself does not manage a complex internal state (stateless in the sense of a state machine). Its execution is purely reactive:

1. **Waiting:** The function block waits for an update event at `IN1` or `IN2`.

Encapsulation: 2. **Calculation:** After an event occurs, the data values are read and the subtraction is performed.

1. **Output:** The new difference value is applied to `OUT`, and the adapter's output event is triggered.

Application Scenarios

- **Differential Pressure/Differential Temperature Measurement:** Calculation of the deviation between two analog sensors whose values are already available as structured adapter signals.
- **Setpoint-Actual Value Comparison:** Subtraction of an actual value from a setpoint to determine the control deviation in control loops.
- **Zero Point Compensation (Offset Calculation):** Subtraction of a static or dynamic offset value (via `IN2`) from a raw signal (via `IN1`).
-

Comparison with Similar Building Blocks

Compared to a standard subtraction block (such as the classic `SUB` block from the IEC 61131-3 library), the `AUI_SUB_2_UNGATED` does not require explicit wiring of separate event and data ports (e.g., `REQ`, `IN1`, and `IN2`). All the logic for value transfer and updates is encapsulated in the `AUI` adapter. This makes the `AUI_SUB_2_UNGATED` easier to maintain and more robust against wiring errors in complex system architectures.

Change Detection

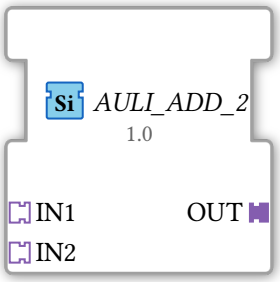
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUI_SUB_2_UNGATED` is a highly efficient, clear, and modern function block for performing subtractions. Through the consistent use of adapters, it integrates seamlessly into service-oriented and modularized control concepts and contributes to a clean software design in the 4diac IDE.

AULI_ADD_2

```
    +------------------------------+
| AULI_ADD_2 | | | IN1 ==| (Socket) (Plug) |== OUT | | IN2 ==| (Socket) | +————————————+
```



ADD input 1 – *AULI*
ADD input 2 – *AULI*

AULI – ADD result

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_ADD_2` is a generic arithmetic block for the IEC 61499 environment (4diac-ide). It performs the mathematical addition of two input values transmitted via standardized, unidirectional adapters. The calculated result is passed to a corresponding output adapter.

The block does not have any classic, discrete event or data inputs at the top level. All communication and value transmissions are handled via adapters.

Interface Structure

Event Inputs

No direct event inputs are available. (Event control is encapsulated within the adapters).

Event Outputs

No direct event outputs are available. (Event control is encapsulated within the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AULI`):

The first addend for the addition operation.

- **IN2** (Type: `adapter::types::unidirectional::AULI`):

The second addend for the addition operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AULI`):

The result of the addition ($OUT = IN1 + IN2$).

Functionality

The function block `AULI_ADD_2` performs a classic addition operation. As soon as values change at the input adapters `IN1` or `IN2`, or a corresponding update event is triggered via the adapters, the function block calculates the sum of the transmitted values:

$$Result = Value(IN1) + Value(IN2) \tag{2.94}$$

This result, along with the associated update event, is then forwarded to subsequent function blocks via the output plug `OUT`.

- **Generic Nature:** The function block is declared as qzmsdocs0000010qz via the attribute `GenericClassName`. This allows for flexible typing depending on the specific implementation of the `AULI` adapter data used.
- **Adapter Encapsulation:** Using adapters of type `unidirectional::AULI` reduces the complexity of control circuit diagrams because data and associated trigger events are bundled in a single connection.

Since this function block is a purely functional/mathematical coupler, it typically does not have a complex internal state machine (ECC). Its execution is purely data- or event-driven based on the incoming adapter signals:

1. **Idle State:** The function block waits for signal updates at `IN1` or `IN2`.
2. **Evaluate:** When an event is received at one of the sockets, the data values are added.
3. **Output (Propagate):** The added value is output via the plug `OUT`, triggering the corresponding event.

- **Measurement Aggregation:** Addition of two analog process values (e.g., two parallel flow sensors to determine the total flow rate).
- **Setpoint Offset:** Applying a correction value or offset (via `IN2`) to a primary setpoint (via `IN1`).
- **Signal Bundling:** Structured mathematical signal processing in complex plant control systems using standardized adapter interfaces to reduce wiring effort.

2.4 · Comparison with Similar Components

Compared to a standard add-in component (such as the IEC 61131-3 compliant `ADD` component), which uses discrete pins for `REQ`, `CNF`, `IN1`, `IN2`, and `OUT`, the `AULI_ADD_2` eliminates the need for explicit wiring of control flows (events) and data lines. This significantly improves clarity in the 4diac development environment, as only a single adapter line needs to be drawn per signal source.

The `AULI_ADD_2` is a highly efficient, modular auxiliary component for analog signal processing. Through the consistent use of unidirectional adapters, it facilitates the design of cleanly structured and maintainable control applications according to the IEC 61499 standard.

Technical Features

State Overview

Application Scenarios

Comparison with Similar Function Blocks

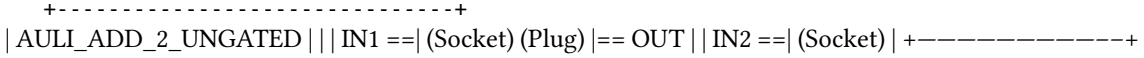
Change Detection

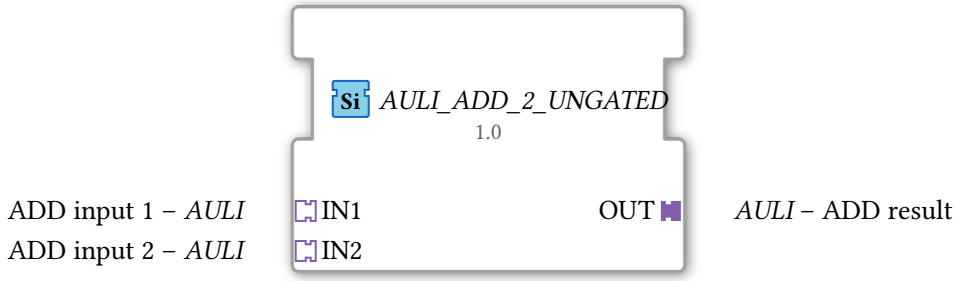
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AULI_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_ADD_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.





• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_ADD_2_UNGATED` is a generic arithmetic block for the IEC 61499 environment (4diac-ide). It performs the mathematical addition of two input values transmitted via standardized, unidirectional adapters. The calculated result is passed to a corresponding output adapter.

The block does not have any classic, discrete event or data inputs at the top level. All communication and value transmissions are handled via adapters.

Interface Structure

Event Inputs

No direct event inputs are available. (Event control is encapsulated within the adapters).

Event Outputs

No direct event outputs are available. (Event control is encapsulated within the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AULI`):

The first addend for the addition operation.

- **IN2** (Type: `adapter::types::unidirectional::AULI`):

The second addend for the addition operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AULI`):

The result of the addition ($OUT = IN1 + IN2$).

Functionality

The function block `AULI_ADD_2_UNGATED` performs a classic addition operation. As soon as values change at the input adapters `IN1` or `IN2`, or a corresponding update event is triggered via the adapters, the function block calculates the sum of the transmitted values:

$$Result = Value(IN1) + Value(IN2) \tag{2.95}$$

This result, along with the associated update event, is then forwarded to subsequent function blocks via the output plug `OUT`.

- **Generic Nature:** The function block is declared as qzmsdocs0000010qz via the attribute `GenericClassName`. This allows for flexible typing depending on the specific implementation of the `AULI` adapter data used.

- **Adapter Encapsulation:** Using adapters of type `unidirectional::AULI` reduces the complexity of control circuit diagrams because data and associated trigger events are bundled in a single connection.

Since this function block is a purely functional/mathematical coupler, it typically does not have a complex internal state machine (ECC). Its execution is purely data- or event-driven based on the incoming adapter signals:

1. **Idle State:** The function block waits for signal updates at `IN1` or `IN2`.
2. **Evaluate:** When an event is received at one of the sockets, the data values are added.
3. **Output (Propagate):** The added value is output via the plug `OUT`, triggering the corresponding event.

- **Measurement Aggregation:** Addition of two analog process values (e.g., two parallel flow sensors to determine the total flow rate).
- **Setpoint Offset:** Applying a correction value or offset (via `IN2`) to a primary setpoint (via `IN1`).
- **Signal Bundling:** Structured mathematical signal processing in complex plant control systems using standardized adapter interfaces to reduce wiring effort.

2.5 · Comparison with Similar Components

Compared to a standard add-in component (such as the IEC 61131-3 compliant `ADD` component), which uses discrete pins for `REQ`, `CNF`, `IN1`, `IN2`, and `OUT`, the `AULI_ADD_2_UNGATED` eliminates the need for explicit wiring of control flows (events) and data lines. This significantly improves clarity in the 4diac development environment, as only a single adapter line needs to be drawn per signal source.

The `AULI_ADD_2_UNGATED` is a highly efficient, modular auxiliary component for analog signal processing. Through the consistent use of unidirectional adapters, it facilitates the design of cleanly structured and maintainable control applications according to the IEC 61499 standard.

Technical Features

State Overview

Application Scenarios

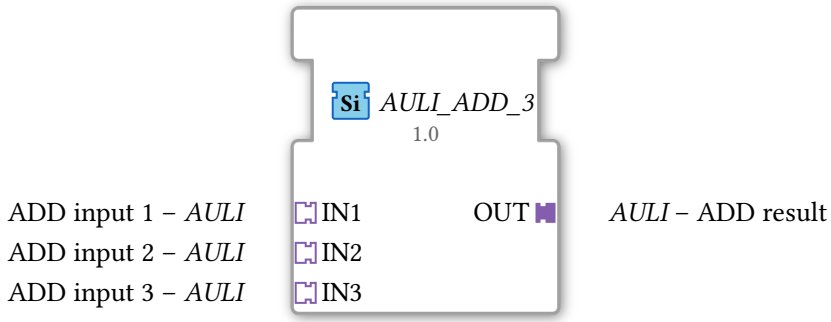
Comparison with Similar Function Blocks

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

AULI_ADD_3



(Image of the function block temporarily unavailable)

• – ◦ • – ◦ • – ◦ •

Introduction

The `AULI_ADD_3` function block is a generic arithmetic function block for IEC 61499 systems, specifically designed for adding three input values. Unlike traditional mathematical function blocks, this block uses `AULI` (unidirectional) adapter connections to transmit data and associated control signals in a bundled manner. This significantly reduces wiring complexity within the development environment (4diac IDE).

Interface Structure

Event Inputs

No direct event inputs are defined. Event-based control is handled implicitly via the connected adapters (`Sockets`).

Event Outputs

No direct event outputs are defined. Event forwarding is encapsulated via the output adapter (`Plug`).

Data Inputs

No direct data inputs are available. Data transfer occurs via the adapter interfaces.

Data Outputs

No direct data outputs are available. The calculation result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

The sockets serve as receivers for the values to be added.

- **IN1** (Type: `adapter::types::unidirectional::AULI`): First addend of the arithmetic addition.
- **IN2** (Type: `adapter::types::unidirectional::AULI`): Second addend of the arithmetic addition.
- **IN3** (Type: `adapter::types::unidirectional::AULI`): Third addend of the arithmetic addition.

Plugs (Output Adapters)

The plug serves as a transmitter for the calculated result.

- **OUT** (Type: `adapter::types::unidirectional::AULI`): Output for the result of the addition ($IN1 + IN2 + IN3$).

Functionality

As soon as new data is signaled at the input adapters (`IN1`, `IN2`, `IN3`), the function block performs the mathematical addition.

The underlying formula is:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.96}$$

The calculated result is immediately passed to the output adapter `OUT` and is available to subsequent function blocks. Since these are unidirectional adapters, the data stream flows exclusively from the sockets to the plug.

–

Technical Features

- **Generic Function Block:** The function block is declared as a generic type in the 4diac IDE (`GenericClassName = 'GEN_AULI_ADD'`). This enables flexible processing of various data types (e.g., INT, REAL, LREAL), supported by the definition of the `AULI` adapter.
- **Adapter Encapsulation:** Using adapters instead of standard data and event channels makes the system design more modular and transparent.
- **Resource Efficiency:** Computations are performed directly based on adapter events, ensuring high-performance execution.

State Overview

The function block operates as a stateless (or purely event-driven) arithmetic block:

1. **Ready (Idle):** The block waits for value changes or trigger signals at the input adapters `IN1`, `IN2`, or `IN3`.

Resource Efficiency: Computations are performed directly based on adapter events, ensuring high performance.

The function block operates as a stateless (or purely event-driven) arithmetic block:

Ready (Idle): The block waits for value changes or trigger signals at the input adapters `IN1`, `IN2`, or `IN3`. 2. **Evaluation:** After a signal arrives, the values are read and summed.

1. **Output:** The sum is passed to `OUT`, and the corresponding event in the adapter is triggered. The function block immediately returns to the *Ready* state.

Application Scenarios

- **Measurement Summing:** Summarizing three analog measurements (e.g., three partial currents to determine the total current) in process automation.
- **Setpoint Generation:** Adding base setpoints with two different correction or offset values.
- **Structured Data Processing:** Used in complex IEC 61499 control applications where consistent use of adapters is required to improve clarity.

Comparison with Similar Components

- **Standard Adder (IEC 61131-3):** The classic `ADD` component requires explicit `REQ` and `CNF` event lines, as well as individual data lines. `AULI_ADD_3` replaces these with just four adapter connections, which simplifies the visual programming interface.
- **Cascaded Adders (e.g., 2x `AULI_ADD_2`):** To add three values with classic two-way adders, two components would have to be connected in series. `AULI_ADD_3` eliminates the need for an entire function block and the associated intermediate wiring.

•

Change Detection

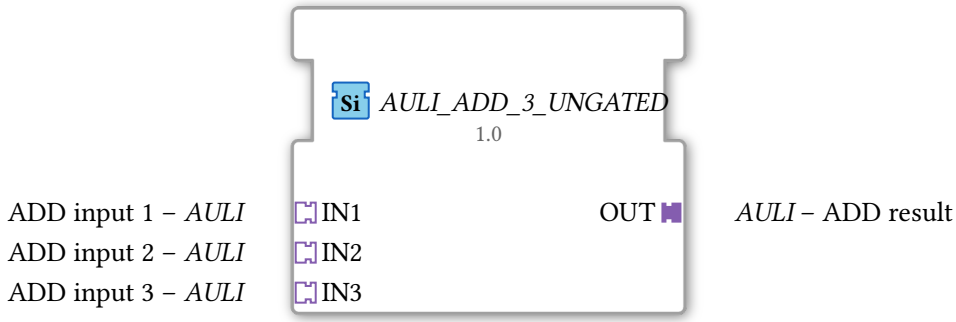
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AULI_ADD_3` is a practical and modern function block for structured programming in 4diac. Thanks to the consistent use of the unidirectional `AULI` adapter, it is ideally suited for clean, clear, and maintainable architectures in distributed control systems according to the IEC 61499 standard.

AULI_ADD_3_UNGATED

i UNGATED variant: This block is the ungated version of `AULI_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(Image of the function block temporarily unavailable)

• – ◦ • – ◦ • – ◦ •

Introduction

The `AULI_ADD_3_UNGATED` function block is a generic arithmetic function block for IEC 61499 systems, specifically designed for adding three input values. Unlike traditional mathematical function blocks, this block uses `AULI` (unidirectional) adapter connections to transmit data and associated control signals in a bundled manner. This significantly reduces wiring complexity within the development environment (4diac IDE).

Interface Structure

Event Inputs

No direct event inputs are defined. Event-based control is handled implicitly via the connected adapters (`Sockets`).

Event Outputs

No direct event outputs are defined. Event forwarding is encapsulated via the output adapter (`Plug`).

Data Inputs

No direct data inputs are available. Data transfer occurs via the adapter interfaces.

Data Outputs

No direct data outputs are available. The calculation result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

The sockets serve as receivers for the values to be added.

- **IN1** (Type: `adapter::types::unidirectional::AULI`): First addend of the arithmetic addition.
- **IN2** (Type: `adapter::types::unidirectional::AULI`): Second addend of the arithmetic addition.
- **IN3** (Type: `adapter::types::unidirectional::AULI`): Third addend of the arithmetic addition.

Plugs (Output Adapters)

The plug serves as a transmitter for the calculated result.

- **OUT** (Type: `adapter::types::unidirectional::AULI`): Output for the result of the addition ($IN1 + IN2 + IN3$).

Functionality

As soon as new data is signaled at the input adapters (`IN1`, `IN2`, `IN3`), the function block performs the mathematical addition.

The underlying formula is:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.97}$$

The calculated result is immediately passed to the output adapter `OUT` and is available to subsequent function blocks. Since these are unidirectional adapters, the data stream flows exclusively from the sockets to the plug.

–

Technical Features

- **Generic Function Block:** The function block is declared as a generic type in the 4diac IDE (`GenericClassName = 'GEN_AULI_ADD'`). This enables flexible processing of various data types (e.g., INT, REAL, LREAL), supported by the definition of the `AULI` adapter.
- **Adapter Encapsulation:** Using adapters instead of standard data and event channels makes the system design more modular and transparent.
- **Resource Efficiency:** Computations are performed directly based on adapter events, ensuring high-performance execution.

State Overview

The function block operates as a stateless (or purely event-driven) arithmetic block:

1. **Ready (Idle):** The block waits for value changes or trigger signals at the input adapters `IN1`, `IN2`, or `IN3`.

Resource Efficiency: Computations are performed directly based on adapter events, ensuring high performance.

The function block operates as a stateless (or purely event-driven) arithmetic block:
Ready (Idle): The block waits for value changes or trigger signals at the input adapters `IN1`, `IN2`, or `IN3`. 2. **Evaluation:** After a signal arrives, the values are read and summed.

1. **Output:** The sum is passed to `OUT`, and the corresponding event in the adapter is triggered. The function block immediately returns to the *Ready* state.

Application Scenarios

- **Measurement Summing:** Summarizing three analog measurements (e.g., three partial currents to determine the total current) in process automation.
- **Setpoint Generation:** Adding base setpoints with two different correction or offset values.
- **Structured Data Processing:** Used in complex IEC 61499 control applications where consistent use of adapters is required to improve clarity.

Comparison with Similar Components

- **Standard Adder (IEC 61131-3):** The classic `ADD` component requires explicit `REQ` and `CNF` event lines, as well as individual data lines. `AULI_ADD_3_UNGATED` replaces these with just four adapter connections, which simplifies the visual programming interface.
- **Cascaded Adders (e.g., 2x `AULI_ADD_2`):** To add three values with classic two-way adders, two components would have to be connected in series. `AULI_ADD_3_UNGATED` eliminates the need for an entire function block and the associated intermediate wiring.

•

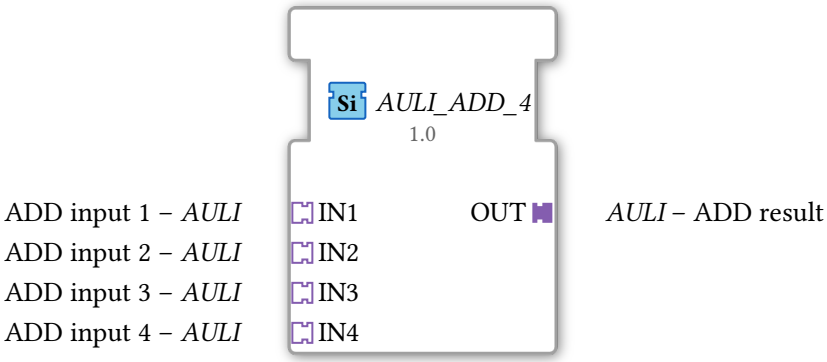
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AULI_ADD_3_UNGATED` is a practical and modern function block for structured programming in 4diac. Thanks to the consistent use of the unidirectional `AULI` adapter, it is ideally suited for clean, clear, and maintainable architectures in distributed control systems according to the IEC 61499 standard.

AULI_ADD_4



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_ADD_4` is a generic function block (FB) designed for the mathematical addition of four values. The unique feature of this block lies in its use of adapters of type `AULI` (unidirectional). By encapsulating data and event transmission within adapters, the block enables clear and modular wiring within IEC 61499 applications in the 4diac IDE.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is implicit via the connected adapter sockets.

Event Outputs

There are no direct event outputs. Event control is implicit via the connected adapter plug.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. Data is output via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AULI`): First addend for the addition operation.
- **IN2** (Type: `adapter::types::unidirectional::AULI`): Second addend for the addition operation.
- **IN3** (Type: `adapter::types::unidirectional::AULI`): Third addend for the addition operation.
- **IN4** (Type: `adapter::types::unidirectional::AULI`): Fourth addend for the addition operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AULI`): Contains the result of the addition (`IN1 + IN2 + IN3 + IN4`) and the corresponding output event.

Functionality

As soon as a new value event is signaled at one or more of the input adapters (`IN1` to `IN4`), the function block reads the current values from all four adapters. These values are added mathematically:

$$\text{Result} = \text{Value}_{IN1} + \text{Value}_{IN2} + \text{Value}_{IN3} + \text{Value}_{IN4} \tag{2.98}$$

The calculated result is passed to the output adapter `OUT`, and simultaneously, the corresponding event is triggered at the output plug to inform subsequent function blocks about the value update.

Technical Features

- **Generic Function Block:** The function block is based on the generic class `GEN_AULI_ADD`. This means that it can flexibly respond to different data types within the `AULI` adapter structure.
- **Unidirectional Adapters:** Unidirectional adapters (`unidirectional::AULI`) are used. This simplifies data flow, as information and events flow exclusively in one direction (from inputs to output).
- **Reduced wiring effort:** By using adapters, event and data lines do not need to be run separately. A single adapter channel bundles all relevant signals.

State overview

The function block behaves in an event-driven and stateless manner (i.e., it has no internal memory for historical values):

1. **Wait state:** The function block waits for an event at one of the inputs (`IN1` to `IN4`).
2. **Calculation:** When an event arrives, the data from all four inputs are summed.
3. **Output:** The calculated value is applied to `OUT`, and the output event is triggered. The function block immediately returns to the wait state.

Application Scenarios

- **Measurement Summing:** Summarizing four individual sensor values (e.g., energy measurements from four consumers, flow rates from four pipes) into a single total value.
- **Average Preparation:** Pre-summing four data points before subsequent division to calculate the average.
- **Modular Control Architectures:** Use in complex systems where signals are already standardized as `AULI` adapters.

Comparison with Similar Function Blocks

- **Standard ADD Function Blocks (e.g., `ADD` from IEC 61131-3):** These use direct data and event ports (such as `REQ`, `CNF`, `IN1`, `IN2`). `AULI_ADD_4`, on the other hand, completely encapsulates these interfaces in adapters, resulting in a cleaner control flow diagram.
- **AULI_ADD_2:** A similar function block, but with only two inputs. `AULI_ADD_4` eliminates the need to cascade multiple individual addition function blocks when adding four values.

Change Detection

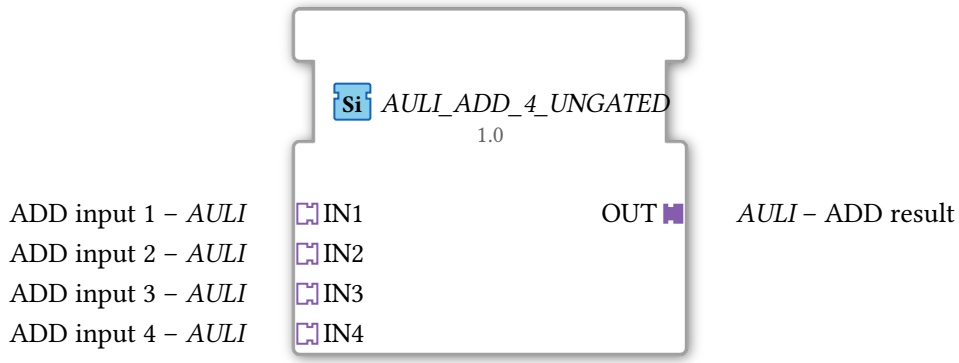
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AULI_ADD_4` function block is a highly efficient, clear, and modern block for arithmetic operations in IEC 61499. Through the consistent use of the `AULI` adapter structure, it significantly reduces the number of connection lines in the 4diac IDE application editor and is ideally suited for modular automation solutions.

AULI_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_ADD_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_ADD_4_UNGATED` is a generic function block (FB) designed for the mathematical addition of four values. The unique feature of this block lies in its use of adapters of type `AULI` (unidirectional). By encapsulating data and event transmission within adapters, the block enables clear and modular wiring within IEC 61499 applications in the 4diac IDE.

Interface Structure

Event Inputs
There are no direct event inputs. Event control is implicit via the connected adapter sockets.

Event Outputs
There are no direct event outputs. Event control is implicit via the connected adapter plug.

Data Inputs
There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs
There are no direct data outputs. Data is output via the output adapter.

Adapters

Sockets (Input Adapters)
- **IN1** (Type: `adapter::types::unidirectional::AULI`): First addend for the addition operation.
- **IN2** (Type: `adapter::types::unidirectional::AULI`): Second addend for the addition operation.
- **IN3** (Type: `adapter::types::unidirectional::AULI`): Third addend for the addition operation.
- **IN4** (Type: `adapter::types::unidirectional::AULI`): Fourth addend for the addition operation.

Plugs (Output Adapters)
- **OUT** (Type: `adapter::types::unidirectional::AULI`): Contains the result of the addition (`IN1 + IN2 + IN3 + IN4`) and the corresponding output event.

Functionality
As soon as a new value event is signaled at one or more of the input adapters (`IN1` to `IN4`), the function block reads the current values from all four adapters. These values are added mathematically:

$$\text{Result} = \text{Value}_{IN1} + \text{Value}_{IN2} + \text{Value}_{IN3} + \text{Value}_{IN4} \tag{2.99}$$

The calculated result is passed to the output adapter `OUT`, and simultaneously, the corresponding event is triggered at the output plug to inform subsequent function blocks about the value update.

Technical Features
- **Generic Function Block:** The function block is based on the generic class `GEN_AULI_ADD`. This means that it can flexibly respond to different data types within the `AULI` adapter structure.
- **Unidirectional Adapters:** Unidirectional adapters (`unidirectional::AULI`) are used. This simplifies data flow, as information and events flow exclusively in one direction (from inputs to output).
- **Reduced wiring effort:** By using adapters, event and data lines do not need to be run separately. A single adapter channel bundles all relevant signals.

State overview
The function block behaves in an event-driven and stateless manner (i.e., it has no internal memory for historical values):

1. **Wait state:** The function block waits for an event at one of the inputs (`IN1` to `IN4`).
2. **Calculation:** When an event arrives, the data from all four inputs are summed.
3. **Output:** The calculated value is applied to `OUT`, and the output event is triggered. The function block immediately returns to the wait state.

Application Scenarios
- **Measurement Summing:** Summarizing four individual sensor values (e.g., energy measurements from four consumers, flow rates from four pipes) into a single total value.
- **Average Preparation:** Pre-summing four data points before subsequent division to calculate the average.
- **Modular Control Architectures:** Use in complex systems where signals are already standardized as `AULI` adapters.

Comparison with Similar Function Blocks

- **Standard ADD Function Blocks (e.g., `ADD` from IEC 61131-3):** These use direct data and event ports (such as `REQ`, `CNF`, `IN1`, `IN2`). `AULI_ADD_4_UNGATED`, on the other hand, completely encapsulates these interfaces in adapters, resulting in a cleaner control flow diagram.
- **AULI_ADD_2:** A similar function block, but with only two inputs. `AULI_ADD_4_UNGATED` eliminates the need to cascade multiple individual addition function blocks when adding four values.

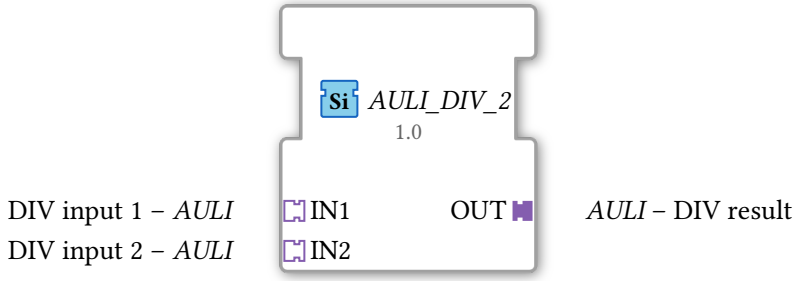
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AULI_ADD_4_UNGATED` function block is a highly efficient, clear, and modern block for arithmetic operations in IEC 61499. Through the consistent use of the `AULI` adapter structure, it significantly reduces the number of connection lines in the 4diac IDE application editor and is ideally suited for modular automation solutions.

AULI_DIV_2



No image available

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_DIV_2` is a generic function block (Generic FB) designed to perform arithmetic division (DIV). It is based on the IEC 61499 standard and utilizes the concept of adapters to efficiently transfer data and the associated control logic. The block is defined in the namespace `adapter::iec61131::arithmetic` and uses the generic class `GEN_AULI_DIV`.

Interface Structure

Event Inputs

This function block does not have direct, dedicated event inputs. Signal flow control and event processing are handled entirely via the connected adapters.

Event Outputs

This function block does not have direct, dedicated event outputs. Event forwarding is handled via the output adapter.

Data Inputs

There are no direct data inputs. The values for the calculation are provided via the input adapters.

Data Outputs

There are no direct data outputs. The calculation result is provided via the output adapter.

Adapters

All communication for this function block is implemented via adapters of type `AULI` (unidirectional):

- **IN1 (Socket)**:
- **Type**: `adapter::types::unidirectional::AULI`
- **Description**: First input of the division (dividend/counter).

- **IN2 (Socket)**:

Type: `adapter::types::unidirectional::AULI`
Description: Second input for division (divisor/denominator).
OUT (Plug):
Type: `adapter::types::unidirectional::AULI`
Description: Output that provides the result of the division (quotient).

Functionality

This module performs a mathematical division of the input values provided via the adapters. The formula is:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.100}$$

Since this is a generic function block (`GEN_AULI_DIV`), it can adapt to various numeric data types at runtime or during instantiation in the 4diac IDE, provided these are supported by type `AULI`. The calculation is typically triggered by an update event on one of the input adapters (`IN1` or `IN2`). The result is then provided at the output adapter `OUT` along with a corresponding output event.

- **Generic Nature**: The attribute `GenericClassName` with the value `'GEN_AULI_DIV'` makes the function block flexible and not rigidly tied to a single data type (such as only `REAL` or `INT`).
- **Encapsulation via Adapters**: Since no traditional event and data ports are used, the visual design in the function block diagram (FBD) remains very clear. The logical relationship between data and triggers is encapsulated in the `AULI` adapter.
- **Unidirectional Adapters**: The use of `unidirectional::AULI` indicates that the information flow is strictly unidirectional (from the sockets to the function block and from the function block to the plug).

Since this function block is a basic arithmetic function block, it typically exhibits simple, stateless behavior (i.e., it behaves like a classic Execution Control Chart (ECC) with a default state that reacts to input events):

Technical Features

1. **Waiting for Event**: The function block waits for an update event at `IN1` or `IN2`.
2. **Calculation**: Upon receiving an event, the division is performed.
3. **Result Transfer**: The calculated value is assigned to `OUT`, and the corresponding output event is triggered at the plug.
4. **Return**: The function block returns to the wait state.

State Overview

Note on Division by Zero: Depending on the implementation of the runtime environment for the generic type `GEN_AULI_DIV`, mechanisms to prevent division by zero (e.g., outputting `NaN`, infinity, or intercepting via the adapter status) should be considered.

Application Scenarios

- **Sensor Value Scaling**: Division of raw analog values by a scaling factor for conversion to physical units.
- **Ratio Measurement**: Calculation of ratios in process automation (e.g., flow rate ratio of two media).
- **Averaging**: Use in computational networks where total values need to be divided by a number.
- **Structured Architectures**: Use in systems that consistently rely on adapter-based communication to minimize the number of visible connection lines in the system model.

Comparison with Similar Function Blocks

- **Standard `DIV` component**: A classic IEC 61131-3 / IEC 61499 `DIV` component uses direct data inputs (`IN1`, `IN2`) and event ports (`REQ`, `CNF`). The `AULI_DIV_2`, on the other hand, bundles these signals in adapters, which significantly simplifies wiring for complex signal structures.
- **Standard `DIV` component**: A classic IEC 61131-3 / IEC 61499 `DIV` component uses direct data inputs (`IN1`, `IN2`) and event ports (`REQ`, `CNF`). The `AULI_DIV_2`, however, bundles these signals in adapters, which significantly simplifies wiring for complex signal structures.
- – **Other AULI Arithmetic Blocks** (e.g., `AULI_ADD`, `AULI_SUB`): These use the same interface philosophy with sockets and plugs, but perform different mathematical operations.

Change Detection

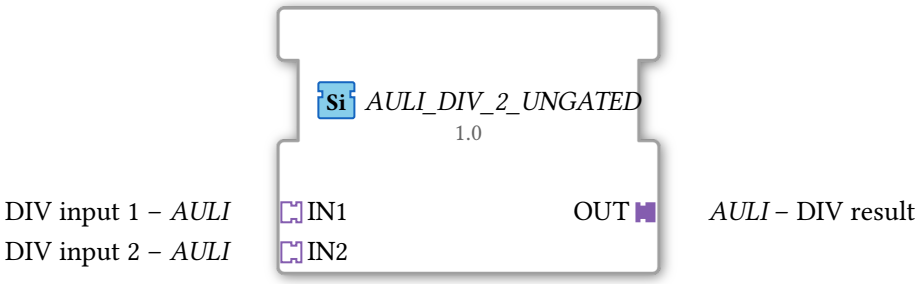
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AULI_DIV_2` is a modern, modular function block for arithmetic division in Eclipse 4diac. Through the consistent use of unidirectional adapters, it promotes clean and well-organized software design in distributed control systems according to IEC 61499 by elegantly bundling data and control events.

AULI_DIV_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_DIV_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



No image available

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_DIV_2_UNGATED` is a generic function block (Generic FB) designed to perform arithmetic division (DIV). It is based on the IEC 61499 standard and utilizes the concept of adapters to efficiently transfer data and the associated control logic. The block is defined in the namespace `adapter::iec61131::arithmetic` and uses the generic class `GEN_AULI_DIV`.

Interface Structure

Event Inputs

This function block does not have direct, dedicated event inputs. Signal flow control and event processing are handled entirely via the connected adapters.

Event Outputs

This function block does not have direct, dedicated event outputs. Event forwarding is handled via the output adapter.

Data Inputs

There are no direct data inputs. The values for the calculation are provided via the input adapters.

Data Outputs

There are no direct data outputs. The calculation result is provided via the output adapter.

Adapters

All communication for this function block is implemented via adapters of type `AULI` (unidirectional):

- **IN1 (Socket)**:
- **Type**: `adapter::types::unidirectional::AULI`

- **Description**: First input of the division (dividend/counter).
- **IN2 (Socket)**:

Type: `adapter::types::unidirectional::AULI`
Description: Second input for division (divisor/denominator).
OUT (Plug):
Type: `adapter::types::unidirectional::AULI`
Description: Output that provides the result of the division (quotient).

Functionality

This module performs a mathematical division of the input values provided via the adapters. The formula is:

$$\text{OUT} = \frac{\text{IN1}}{\text{IN2}} \tag{2.101}$$

Since this is a generic function block (`GEN_AULI_DIV`), it can adapt to various numeric data types at runtime or during instantiation in the 4diac IDE, provided these are supported by type `AULI`. The calculation is typically triggered by an update event on one of the input adapters (`IN1` or `IN2`). The result is then provided at the output adapter `OUT` along with a corresponding output event.

- **Generic Nature**: The attribute `GenericClassName` with the value `'GEN_AULI_DIV'` makes the function block flexible and not rigidly tied to a single data type (such as only `REAL` or `INT`).
- **Encapsulation via Adapters**: Since no traditional event and data ports are used, the visual design in the function block diagram (FBD) remains very clear. The logical relationship between data and triggers is encapsulated in the `AULI` adapter.
- **Unidirectional Adapters**: The use of `unidirectional::AULI` indicates that the information flow is strictly unidirectional (from the sockets to the function block and from the function block to the plug).

Since this function block is a basic arithmetic function block, it typically exhibits simple, stateless behavior (i.e., it behaves like a classic Execution Control Chart (ECC) with a default state that reacts to input events):

Technical Features

1. **Waiting for Event**: The function block waits for an update event at `IN1` or `IN2`.
2. **Calculation**: Upon receiving an event, the division is performed.
3. **Result Transfer**: The calculated value is assigned to `OUT`, and the corresponding output event is triggered at the plug.
4. **Return**: The function block returns to the wait state.

State Overview

Note on Division by Zero: Depending on the implementation of the runtime environment for the generic type `GEN_AULI_DIV`, mechanisms to prevent division by zero (e.g., outputting `NaN`, infinity, or intercepting via the adapter status) should be considered.

Application Scenarios

- **Sensor Value Scaling**: Division of raw analog values by a scaling factor for conversion to physical units.
- **Ratio Measurement**: Calculation of ratios in process automation (e.g., flow rate ratio of two media).
- **Averaging**: Use in computational networks where total values need to be divided by a number.
- **Structured Architectures**: Use in systems that consistently rely on adapter-based communication to minimize the number of visible connection lines in the system model.

Comparison with Similar Function Blocks

- **Standard `DIV` component**: A classic IEC 61131-3 / IEC 61499 `DIV` component uses direct data inputs (`IN1`, `IN2`) and event ports (`REQ`, `CNF`). The `AULI_DIV_2_UNGATED`, on the other hand, bundles these signals in adapters, which significantly simplifies wiring for complex signal structures.
- **Standard `DIV` component**: A classic IEC 61131-3 / IEC 61499 `DIV` component uses direct data inputs (`IN1`, `IN2`) and event ports (`REQ`, `CNF`). The `AULI_DIV_2_UNGATED`, however, bundles these signals in adapters, which significantly simplifies wiring for complex signal structures.
- – **Other AULI Arithmetic Blocks** (e.g., `AULI_ADD`, `AULI_SUB`): These use the same interface philosophy with sockets and plugs, but perform different mathematical operations.

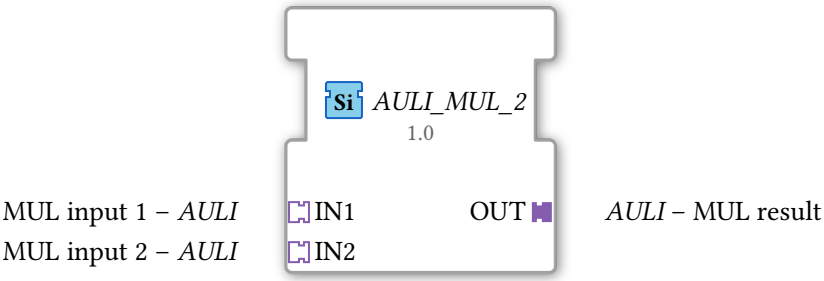
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AULI_DIV_2_UNGATED` is a modern, modular function block for arithmetic division in Eclipse 4diac. Through the consistent use of unidirectional adapters, it promotes clean and well-organized software design in distributed control systems according to IEC 61499 by elegantly bundling data and control events.

AULI_MUL_2



No image available

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_MUL_2` is used to perform an arithmetic multiplication of two values. It is a generic function block (`GEN_AULI_MUL`) whose interfaces are implemented entirely via adapters of type `AULI` (unidirectional). This allows for clean encapsulation of data and events and reduces the wiring effort within the 4diac IDE.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is handled implicitly via the adapter interfaces.

Event Outputs

There are no direct event outputs. Event control is implicit via the adapter interfaces.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. Data is output via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AULI`):
- First input value (multiplicand) for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::AULI`):
- Second input value (multiplier) for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AULI`):
- Result of the multiplication ($OUT = IN1 \times IN2$).

Functionality

As soon as new data is signaled at the input adapters `IN1` or `IN2`, the function block performs the multiplication of the two values:

$$OUT = IN1 \times IN2$$

The result and the corresponding update event are then output via the output adapter `OUT`. Since this is a generic function block, the exact data type depends on the definition of the `AULI` adapter used (e.g., REAL, INT, LREAL).

Technical Features

- **Generic Function Block:** The type is declared as `GEN_AULI_MUL`, which allows for flexible reusability for various numeric data types, provided they are supported by the adapter type.
- **Adapter Encapsulation:** By using unidirectional adapters (`AULI`), data and the associated trigger event are bundled in a single connection. This prevents "spaghetti code" and loose event-data pairs in the function plan.
- **Package Membership:** The function block is organized in the `adapter::iec61131::arithmetic` namespace.
-

State Overview

Since this is a purely computational function block (arithmetic function), it does not have an internal state machine (ECC) in the classical sense.

1. **Waiting:** The function block waits for an update event on the adapters `IN1` or `IN2`.
2. **Calculation:** Upon receiving an event, the values from `IN1` and `IN2` are read and multiplied.
3. **Output:** The result is created on `OUT`, and the output event of the adapter is triggered. The function block immediately returns to the *Waiting* state.

Application Scenarios

- **Signal Processing:** Scaling of sensor values where a measured value needs to be multiplied by a constant or dynamic factor.
- **Control Engineering:** Calculation of manipulated variables in control loops (e.g., the proportional component of a PID controller).
- **Structured Control Architectures:** Use in complex systems where adapters are used uniformly to maintain the readability of the overall system.

–

Comparison with Similar Function Blocks

Compared to a standard multiplication function block (such as the IEC 61199 `MUL` function block), the `AULI_MUL_2` offers the following advantages and disadvantages:

Feature	Standard `MUL` function block	`AULI_MUL_2`
Interfaces	Individual event and data pins	Bundled adapters (`AULI`)
Wiring Effort	High (separate event and data lines)	Minimal (only one adapter line per signal)
Flexibility	High (direct mapping of variables)	Requires the use of the `AULI` adapter type
Readability	Can become cluttered with many function blocks	Very clean and modular

---

Change Detection

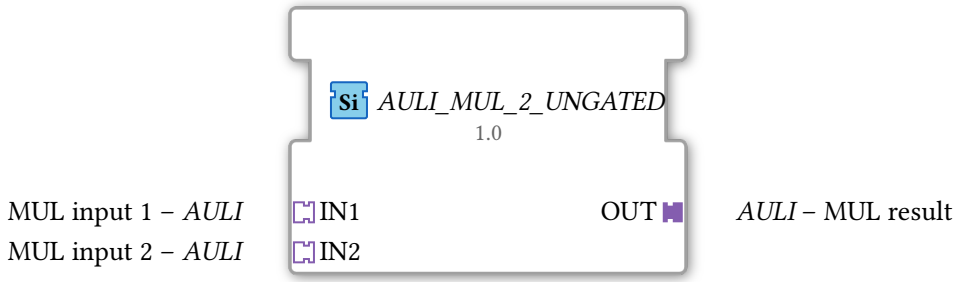
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AULI_MUL_2 is a specialized and modern variant of a multiplication function block for IEC 61499. Due to the consistent use of adapters, it is ideally suited for demanding, modular applications where clarity and standardization are paramount.

AULI_MUL_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AULI_MUL_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



No image available

• – ◦ • – ◦ • – ◦ •

Introduction

The function block AULI_MUL_2_UNGATED is used to perform an arithmetic multiplication of two values. It is a generic function block (GEN_AULI_MUL) whose interfaces are implemented entirely via adapters of type AULI (unidirectional). This allows for clean encapsulation of data and events and reduces the wiring effort within the 4diac IDE.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is handled implicitly via the adapter interfaces.

Event Outputs

There are no direct event outputs. Event control is implicit via the adapter interfaces.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. Data is output via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: adapter::types::unidirectional::AULI):
- First input value (multiplicand) for multiplication.
- **IN2** (Type: adapter::types::unidirectional::AULI):
- Second input value (multiplier) for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: adapter::types::unidirectional::AULI):
- Result of the multiplication ($OUT = IN1 \times IN2$).

Functionality

As soon as new data is signaled at the input adapters `IN1` or `IN2`, the function block performs the multiplication of the two values:

$$OUT = IN1 \times IN2$$

The result and the corresponding update event are then output via the output adapter `OUT`. Since this is a generic function block, the exact data type depends on the definition of the `AULI` adapter used (e.g., REAL, INT, LREAL).

Technical Features

- **Generic Function Block:** The type is declared as `GEN_AULI_MUL`, which allows for flexible reusability for various numeric data types, provided they are supported by the adapter type.
- **Adapter Encapsulation:** By using unidirectional adapters (`AULI`), data and the associated trigger event are bundled in a single connection. This prevents "spaghetti code" and loose event-data pairs in the function plan.
- **Package Membership:** The function block is organized in the `adapter::iec61131::arithmetic` namespace.
-

State Overview

Since this is a purely computational function block (arithmetic function), it does not have an internal state machine (ECC) in the classical sense.

1. **Waiting:** The function block waits for an update event on the adapters `IN1` or `IN2`.
2. **Calculation:** Upon receiving an event, the values from `IN1` and `IN2` are read and multiplied.
3. **Output:** The result is created on `OUT`, and the output event of the adapter is triggered. The function block immediately returns to the *Waiting* state.

Application Scenarios

- **Signal Processing:** Scaling of sensor values where a measured value needs to be multiplied by a constant or dynamic factor.
- **Control Engineering:** Calculation of manipulated variables in control loops (e.g., the proportional component of a PID controller).
- **Structured Control Architectures:** Use in complex systems where adapters are used uniformly to maintain the readability of the overall system.

–

Comparison with Similar Function Blocks

Compared to a standard multiplication function block (such as the IEC 61199 `MUL` function block), the `AULI_MUL_2_UNGATED` offers the following advantages and disadvantages:

Feature	Standard `MUL` function block	`AULI_MUL_2_UNGATED`
Interfaces	Individual event and data pins	Bundled adapters (`AULI`)
Wiring Effort	High (separate event and data lines)	Minimal (only one adapter line per signal)
Flexibility	High (direct mapping of variables)	Requires the use of the `AULI` adapter type
Readability	Can become cluttered with many function blocks	Very clean and modular

---

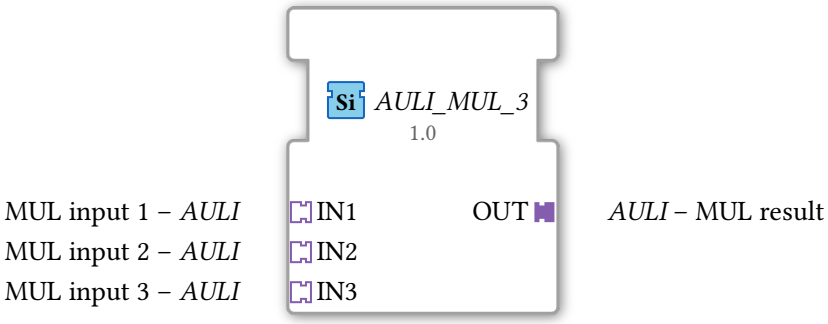
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AULI_MUL_2_UNGATED` is a specialized and modern variant of a multiplication function block for IEC 61499. Due to the consistent use of adapters, it is ideally suited for demanding, modular applications where clarity and standardization are paramount.

AULI_MUL_3



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_MUL_3` is a generic function block (FB) designed for the arithmetic multiplication of three input values via unidirectional adapters. It is defined in the package `adapter::iec61131::arithmetic` and enables clean, adapter-based data processing within the 4diac environment.

Interface Structure

Event Inputs

This function block has no direct event inputs. Control and event processing are handled via the connected adapters.

Event Outputs

This function block has no direct event outputs. Events are forwarded via the output adapter.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AULI`): First input value (multiplicand 1) for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::AULI`): Second input value (multiplicand 2) for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::AULI`): Third input value (multiplicand 3) for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AULI`): Output for the calculated product of the three input values.

Functionality

The AULI_MUL_3 function block performs a classic multiplication of the values provided by the three input adapters (IN1, IN2, and IN3). The calculation follows the mathematical formula:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.102}$$

Since this is a generic function block (GEN_AULI_MUL) based on unidirectional adapters (AULI), the values and their corresponding update events are transmitted within the adapter connections. As soon as new data is received and signaled at the inputs, the calculation is performed, and the result is output via the output adapter OUT.

Technical Features

- **Generic Type:** The function block is declared as a generic type (GenericClassName = "GEN_AULI_MUL"). This allows for flexible handling depending on the underlying data type of the adapters used.
- **Adapter Encapsulation:** By using adapters of type AULI, data and associated control flows (events) are bundled into a single connection, significantly reducing wiring complexity in the function block diagram.

Unidirectionality: The interfaces utilize a unidirectional profile, ensuring a clear flow of information from the inputs to the output.

State Overview

The function block operates in an event-driven manner and has no internal, persistent state (stateless/ combinatorial).

1. **Waiting for Update:** The function block waits for an update event at the input sockets (IN1, IN2, or IN3).
2. **Calculation:** Upon receiving an event, the current values are read and multiplied.

- ... 3. **Output:** The result is passed to the OUT plug and the corresponding output event is triggered.

Application Scenarios

- **Volume Calculation:** Calculation of three-dimensional quantities (e.g., length × width × height) in process automation.
- **Scaling and Weighting:** Application of two consecutive scaling factors to a measured value (e.g., sensor value × calibration factor × unit conversion).
- **Power Calculation:** Three-phase or multi-stage calculations where three physical quantities must be combined multiplicatively.

Comparison with Similar Blocks

- **Standard MUL (IEC 61131-3):** The classic MUL block uses direct data and event pins. AULI_MUL_3 differs in that it has exactly three inputs, which are completely encapsulated by adapters (AULI), resulting in a more streamlined architecture.
- **AULI_MUL_2 (if present):** A hypothetical component with only two inputs. AULI_MUL_3 eliminates the need for an entire cascading component when multiplying three values.

- **AULI_MUL_3_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Change Detection

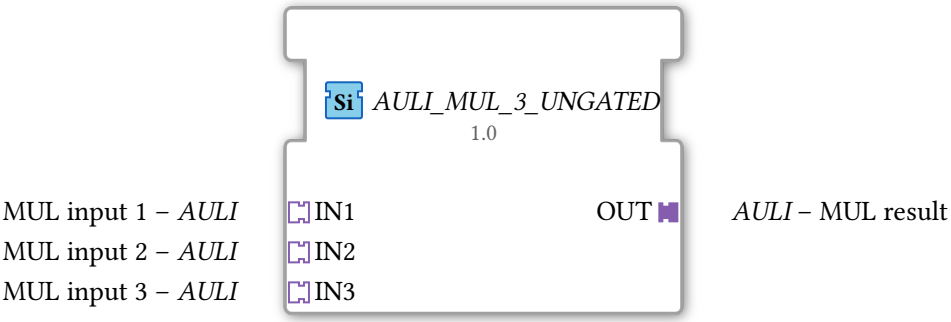
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

AULI_MUL_3 is an efficient auxiliary component for applications requiring a high degree of modularity. Through the consistent use of unidirectional adapters, it helps reduce "signal spaghetti" in complex 4diac applications and simplifies the multiplication of three variables.

AULI_MUL_3_UNGATED

i UNGATED variant: This block is the ungated version of AULI_MUL_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block AULI_MUL_3_UNGATED is a generic function block (FB) designed for the arithmetic multiplication of three input values via unidirectional adapters. It is defined in the package adapter::iec61131::arithmetic and enables clean, adapter-based data processing within the 4diac environment.

Interface Structure

Event Inputs

This function block has no direct event inputs. Control and event processing are handled via the connected adapters.

Event Outputs

This function block has no direct event outputs. Events are forwarded via the output adapter.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: adapter::types::unidirectional::AULI): First input value (multiplicand 1) for multiplication.
- **IN2** (Type: adapter::types::unidirectional::AULI): Second input value (multiplicand 2) for multiplication.
- **IN3** (Type: adapter::types::unidirectional::AULI): Third input value (multiplicand 3) for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: adapter::types::unidirectional::AULI): Output for the calculated product of the three input values.

Functionality

The `AULI_MUL_3_UNGATED` function block performs a classic multiplication of the values provided by the three input adapters (`IN1`, `IN2`, and `IN3`). The calculation follows the mathematical formula:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \tag{2.103}$$

Since this is a generic function block (`GEN_AULI_MUL`) based on unidirectional adapters (`AULI`), the values and their corresponding update events are transmitted within the adapter connections. As soon as new data is received and signaled at the inputs, the calculation is performed, and the result is output via the output adapter `OUT`.

Technical Features

- **Generic Type:** The function block is declared as a generic type (`GenericClassName = "GEN_AULI_MUL"`). This allows for flexible handling depending on the underlying data type of the adapters used.
- **Adapter Encapsulation:** By using adapters of type `AULI`, data and associated control flows (events) are bundled into a single connection, significantly reducing wiring complexity in the function block diagram.

Unidirectionality: The interfaces utilize a unidirectional profile, ensuring a clear flow of information from the inputs to the output.

State Overview

The function block operates in an event-driven manner and has no internal, persistent state (stateless/ combinatorial).

1. **Waiting for Update:** The function block waits for an update event at the input sockets (`IN1`, `IN2`, or `IN3`).
2. **Calculation:** Upon receiving an event, the current values are read and multiplied.

- ... 3. **Output:** The result is passed to the `OUT` plug and the corresponding output event is triggered.

Application Scenarios

- **Volume Calculation:** Calculation of three-dimensional quantities (e.g., length × width × height) in process automation.
- **Scaling and Weighting:** Application of two consecutive scaling factors to a measured value (e.g., sensor value × calibration factor × unit conversion).
- **Power Calculation:** Three-phase or multi-stage calculations where three physical quantities must be combined multiplicatively.

Comparison with Similar Blocks

- **Standard MUL (IEC 61131-3):** The classic `MUL` block uses direct data and event pins. `AULI_MUL_3_UNGATED` differs in that it has exactly three inputs, which are completely encapsulated by adapters (`AULI`), resulting in a more streamlined architecture.
- **AULI_MUL_2 (if present):** A hypothetical component with only two inputs. `AULI_MUL_3_UNGATED` eliminates the need for an entire cascading component when multiplying three values.

- `AULI_MUL_3`: The gated variant – updates the output only on an actual value change.

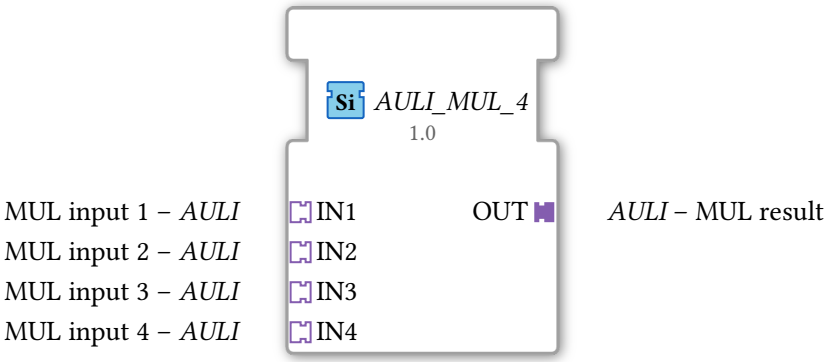
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

`AULI_MUL_3_UNGATED` is an efficient auxiliary component for applications requiring a high degree of modularity. Through the consistent use of unidirectional adapters, it helps reduce “signal spaghetti” in complex 4diac applications and simplifies the multiplication of three variables.

AULI_MUL_4



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_MUL_4` is a generic arithmetic block for the 4diac-ide. It is used to multiply four input values. The block uses an adapter-based interface concept to minimize the number of individual event and data connections in the application diagram and to ensure clean encapsulation.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is handled entirely via the adapters.

Event Outputs

There are no direct event outputs. Event control is handled entirely via the adapters.

Data Inputs

There are no direct data inputs. Data transmission is encapsulated via the input adapters.

Data Outputs

There are no direct data outputs. Data transmission is encapsulated via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AULI`): First multiplicand.
- **IN2** (Type: `adapter::types::unidirectional::AULI`): Second multiplicand.
- **IN3** (Type: `adapter::types::unidirectional::AULI`): Third multiplicand.
- **IN4** (Type: `adapter::types::unidirectional::AULI`): Fourth multiplicand.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AULI`): Result of the multiplication.

Functionality

As soon as new values are signaled at the input adapters (`IN1` to `IN4`), the module performs the multiplication of the four values.

The mathematical formula is:

$$\text{OUT} = \text{IN1} \times \text{IN2} \times \text{IN3} \times \text{IN4} \tag{2.104}$$

Technical Features

The result and the associated processing event are then output via the output adapter `OUT`. Since these are unidirectional adapters of type `AULI`, data and trigger signals flow directly from the sockets to the plug.

State Overview

- **Generic Block:** The block is declared as a generic type (`GenericClassName = 'GEN_AULI_MUL'`). This allows for flexible adaptation to different numeric data types supported by the underlying `AULI` adapter type.
- **Adapter Encapsulation:** Using adapters instead of standard event/data pins greatly simplifies the system design (avoiding "spaghetti wiring" in the control flow).

Application Scenarios

The function block is essentially stateless. Computation is purely reactive, based on the values and events present at the input adapters. No internal historical states are stored.

Comparison with Similar Function Blocks

- **Sensor Value Scaling:** Calculation of corrected measured values where a raw value must be multiplied by several calibration, correction, or conversion factors.
- **Volume and Mass Calculation:** Physical calculations in process engineering that require the product of several variables (e.g., $V = l \times b \times h$ taking into account an additional density factor).
- **Structured Signal Processing:** Use in more complex control applications where data is distributed modularly via adapter structures.
- **Structured Signal Processing:** Use in more complex control applications where data is distributed modularly via adapter structures. ## Comparison with Similar Components
- **Standard MUL Component (IEC 61131-3 / IEC 61499):** Standard multipliers typically have dedicated pins, such as `REQ`, `CNF`, as well as standard data inputs (e.g., `IN1`, `IN2`). `AULI_MUL_4` significantly simplifies interface design through the use of adapters.
- **Cascaded Dual Multipliers:** To multiply four values using standard components, three conventional `MUL` components would need to be cascaded. `AULI_MUL_4` consolidates this logic into a single function block, saving resources and improving clarity.

Change Detection

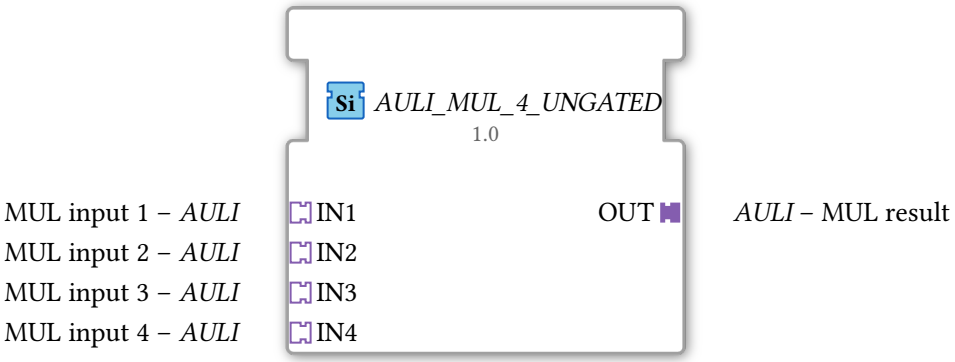
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AULI_MUL_4` is a practical and modern function block for multiplying four numeric values using the IEC 61499 adapter concept. It is ideally suited for cleanly structured, readable, and maintainable control applications in the 4diac IDE.

AULI_MUL_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_MUL_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_MUL_4_UNGATED` is a generic arithmetic block for the 4diac-ide. It is used to multiply four input values. The block uses an adapter-based interface concept to minimize the number of individual event and data connections in the application diagram and to ensure clean encapsulation.

Interface Structure

Event Inputs

There are no direct event inputs. Event control is handled entirely via the adapters.

Event Outputs

There are no direct event outputs. Event control is handled entirely via the adapters.

Data Inputs

There are no direct data inputs. Data transmission is encapsulated via the input adapters.

Data Outputs

There are no direct data outputs. Data transmission is encapsulated via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AULI`): First multiplicand.
- **IN2** (Type: `adapter::types::unidirectional::AULI`): Second multiplicand.
- **IN3** (Type: `adapter::types::unidirectional::AULI`): Third multiplicand.
- **IN4** (Type: `adapter::types::unidirectional::AULI`): Fourth multiplicand.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AULI`): Result of the multiplication.

Functionality

As soon as new values are signaled at the input adapters (`IN1` to `IN4`), the module performs the multiplication of the four values.

The mathematical formula is:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.105}$$

Technical Features

The result and the associated processing event are then output via the output adapter `OUT`. Since these are unidirectional adapters of type `AULI`, data and trigger signals flow directly from the sockets to the plug.

State Overview

- **Generic Block:** The block is declared as a generic type (`GenericClassName = 'GEN_AULI_MUL'`). This allows for flexible adaptation to different numeric data types supported by the underlying `AULI` adapter type.
- **Adapter Encapsulation:** Using adapters instead of standard event/data pins greatly simplifies the system design (avoiding “spaghetti wiring” in the control flow).

Application Scenarios

The function block is essentially stateless. Computation is purely reactive, based on the values and events present at the input adapters. No internal historical states are stored.

Comparison with Similar Function Blocks

- **Sensor Value Scaling:** Calculation of corrected measured values where a raw value must be multiplied by several calibration, correction, or conversion factors.

- **Volume and Mass Calculation:** Physical calculations in process engineering that require the product of several variables (e.g., $V = l \times b \times h$ taking into account an additional density factor).
- **Structured Signal Processing:** Use in more complex control applications where data is distributed modularly via adapter structures.
- **Structured Signal Processing:** Use in more complex control applications where data is distributed modularly via adapter structures. ## Comparison with Similar Components
- **Standard MUL Component (IEC 61131-3 / IEC 61499):** Standard multipliers typically have dedicated pins, such as `REQ`, `CNF`, as well as standard data inputs (e.g., `IN1`, `IN2`). `AULI_MUL_4_UNGATED` significantly simplifies interface design through the use of adapters.
- **Cascaded Dual Multipliers:** To multiply four values using standard components, three conventional `MUL` components would need to be cascaded. `AULI_MUL_4_UNGATED` consolidates this logic into a single function block, saving resources and improving clarity.

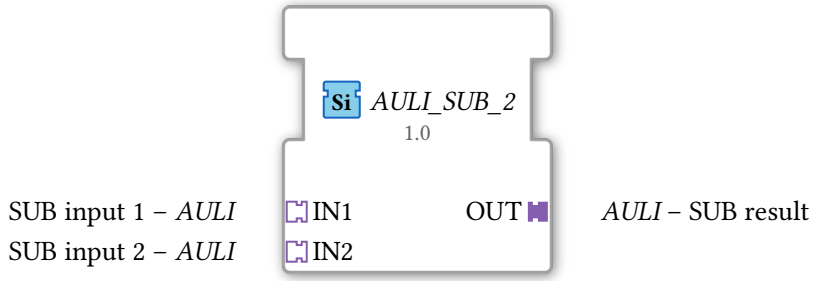
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AULI_MUL_4_UNGATED` is a practical and modern function block for multiplying four numeric values using the IEC 61499 adapter concept. It is ideally suited for cleanly structured, readable, and maintainable control applications in the 4diac IDE.

AULI_SUB_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_SUB_2` is a generic function block (Generic FB) designed to perform arithmetic subtraction (`OUT = IN1 - IN2`). Instead of traditional data and event pins, this block uses adapter interfaces of type `unidirectional::AULI`. This enables structured and modularized signal transmission in IEC 61499 applications.

Interface Structure

Event Inputs

No direct event inputs are available. Event control is handled entirely via the adapters.

Event Outputs

No direct event outputs are available. Event control is handled entirely via the adapters.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AULI`):
- Description: The first input (minuend) for subtraction.
- **IN2** (Type: `adapter::types::unidirectional::AULI`):
- Description: The second input (subtrahend) for subtraction.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AULI`):
- Description: The output (difference) of arithmetic subtraction.

Functionality

As soon as values and their corresponding events arrive via the input adapters `IN1` and `IN2`, the function block executes the mathematical operation `OUT = IN1 - IN2`.

Since this is a generic function block (`GEN_AULI_SUB`), the specific data type processing can be determined by the data types defined in the adapter (e.g., integers or floating-point numbers). The result, along with the corresponding trigger event, is forwarded via the output adapter `OUT`.

` ... —

Technical Features

- **Generic Block:** The attribute `GenericClassName` with the value `'GEN_AULI_SUB'` allows the block to be used flexibly for various data types, provided the adapters used support this.
- **Unidirectional Adapters:** Using the type `AULI` ensures a clear separation of signal flow directions (unidirectional) and drastically reduces the wiring effort in the 4diac editor.

–

State Overview

The block behaves purely combinatorially or event-driven:

1. **Waiting for Input:** The block remains in idle state until an event is signaled at one of the input adapters (`IN1` or `IN2`).

Unidirectional Adapters: 2. **Calculation:** Subtraction is performed upon signal change/event.

1. **Output:** The result is immediately transferred to the plug `OUT`, triggering the output event.

Application Scenarios

- **Target-Actual Value Comparison:** Calculation of control deviations in process engineering (e.g., `Regeldifferenz = Sollwert - Istwert`).
- **Offset Compensation:** Subtraction of zero-point errors or offsets from analog sensor values.
- **Cascaded Calculations:** Mathematical calculations in modular plant control systems where measured values are transported in a structured manner via adapters.

Comparison with Similar Function Blocks

Compared to a standard subtraction function block (e.g., `SUB` from the IEC 61131-3 standard library), `AULI_SUB_2` offers the following advantages:

- **Reduced Complexity:** Instead of separate lines for event (REQ/CNF) and data (IN1, IN2, OUT), the `AULI` adapters bundle all relevant information into a single connection.
- **Modularity:** The design is ideally suited for service-oriented architectures in distributed systems.

–

Change Detection

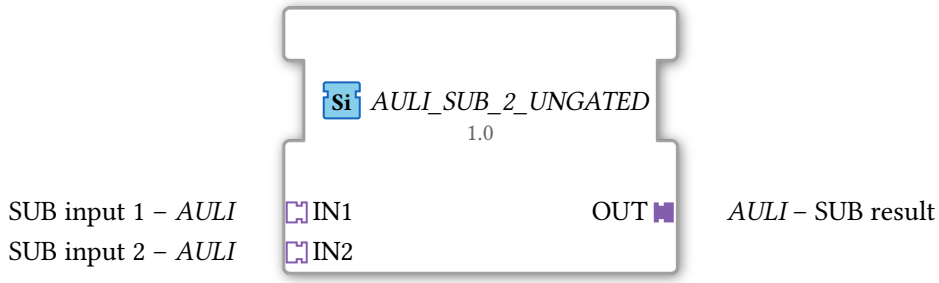
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AULI_SUB_2` is a robust and reusable function block for subtraction. Through the consistent use of adapters, it promotes a clean, well-organized, and maintainable application design in the 4diac IDE.

AULI_SUB_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_SUB_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/ frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AULI_SUB_2_UNGATED` is a generic function block (Generic FB) designed to perform arithmetic subtraction (`OUT = IN1 - IN2`). Instead of traditional data and event pins, this block uses adapter interfaces of type `unidirectional::AULI`. This enables structured and modularized signal transmission in IEC 61499 applications.

Interface Structure

Event Inputs

No direct event inputs are available. Event control is handled entirely via the adapters.

Event Outputs

No direct event outputs are available. Event control is handled entirely via the adapters.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AULI`):
- Description: The first input (minuend) for subtraction.
- **IN2** (Type: `adapter::types::unidirectional::AULI`):
- Description: The second input (subtrahend) for subtraction.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AULI`):
- Description: The output (difference) of arithmetic subtraction.

Functionality

As soon as values and their corresponding events arrive via the input adapters `IN1` and `IN2`, the function block executes the mathematical operation `OUT = IN1 - IN2`.

Since this is a generic function block (`GEN_AULI_SUB`), the specific data type processing can be determined by the data types defined in the adapter (e.g., integers or floating-point numbers). The result, along with the corresponding trigger event, is forwarded via the output adapter `OUT`.

` ... —

Technical Features

- **Generic Block:** The attribute `GenericClassName` with the value `'GEN_AULI_SUB'` allows the block to be used flexibly for various data types, provided the adapters used support this.
- **Unidirectional Adapters:** Using the type `AULI` ensures a clear separation of signal flow directions (unidirectional) and drastically reduces the wiring effort in the 4diac editor.

–

State Overview

The block behaves purely combinatorially or event-driven:

1. **Waiting for Input:** The block remains in idle state until an event is signaled at one of the input adapters (`IN1` or `IN2`).

Unidirectional Adapters: 2. **Calculation:** Subtraction is performed upon signal change/event.

1. **Output:** The result is immediately transferred to the plug `OUT`, triggering the output event.

Application Scenarios

- **Target-Actual Value Comparison:** Calculation of control deviations in process engineering (e.g., `Regeldifferenz = Sollwert - Istwert`).
- **Offset Compensation:** Subtraction of zero-point errors or offsets from analog sensor values.
- **Cascaded Calculations:** Mathematical calculations in modular plant control systems where measured values are transported in a structured manner via adapters.

Comparison with Similar Function Blocks

Compared to a standard subtraction function block (e.g., `SUB` from the IEC 61131-3 standard library), `AULI_SUB_2_UNGATED` offers the following advantages:

- **Reduced Complexity:** Instead of separate lines for event (REQ/CNF) and data (IN1, IN2, OUT), the `AULI` adapters bundle all relevant information into a single connection.
- **Modularity:** The design is ideally suited for service-oriented architectures in distributed systems.

–

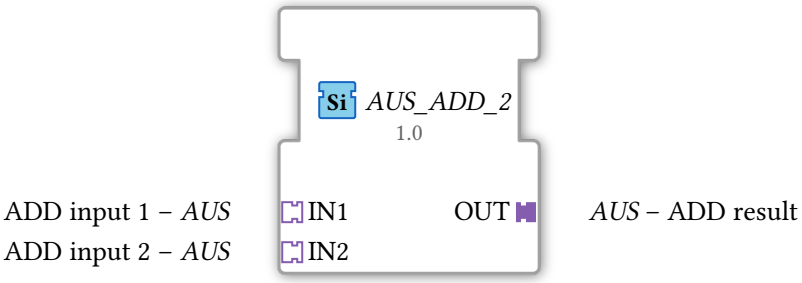
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AULI_SUB_2_UNGATED` is a robust and reusable function block for subtraction. Through the consistent use of adapters, it promotes a clean, well-organized, and maintainable application design in the 4diac IDE.

AUS_ADD_2



(Graphical representation of the function block not available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_ADD_2` is a generic function block (Generic FB) for performing arithmetic addition. It is used to add two input signals and output the result. The special feature of this block lies in the use of standardized, unidirectional adapters (`AUS`), which greatly simplifies and encapsulates the signal and data wiring within Eclipse 4diac.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Event control is handled via the adapters.

Event Outputs

This function block does not have direct event outputs. Event control is handled via the adapters.

Data Inputs

This function block does not have direct data inputs.

Data Outputs

This function block does not have direct data outputs.

Adapters

Sockets (Input Adapters / Jacks)

- **IN1** (Type: `adapter::types::unidirectional::AUS`): The first addend of the addition.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): The second addend of the addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`): The result of the addition (sum of `IN1` and `IN2`).

Functionality

The `AUS_ADD_2` module functions as an arithmetic adder based on adapters. As soon as a new event with a data value arrives at the input adapters `IN1` or `IN2`, the function block adds the two values:

$$OUT = IN1 + IN2 \tag{2.106}$$

The result, along with the corresponding trigger event, is output via the output adapter `OUT`. Since these are unidirectional adapters, the data and control flow is exclusively from the inputs to the output.

Technical Features

- **Generic Function Block:** The attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AUS_ADD'` defines the function block as a generic type. This allows for flexible adaptation to different data types defined by the adapter structures.
- **Encapsulation:** By using adapters instead of individual event and data pins, the system design becomes clearer and less prone to wiring errors.

State Overview

Since this component is essentially a mathematical operation, it does not have a complex internal state diagram (ECC). It reacts to incoming data at adapters `IN1` and `IN2`, calculates the sum, and forwards the result directly to `OUT`.

Application Scenarios

- **Measurement Processing:** Adding two sensor values or applying an offset to a measured value, which is supplied via an adapter.
- **Setpoint Calculation:** Combining base setpoints and correction values in process automation.
- – **Structured Programming:** Used in complex control systems where data and event lines are bundled in adapters to improve clarity.

Comparison with Similar Function Blocks

Compared to a standard addition function block (such as `ADD` from the IEC 61131-3 library), which has discrete data and event pins (such as `REQ`, `CNF`, `IN1`, `IN2`, `OUT`), `AUS_ADD_2` completely encapsulates these interfaces in adapters. This drastically reduces the number of visible connection lines in the 4diac editor and promotes object-oriented design.

Change Detection

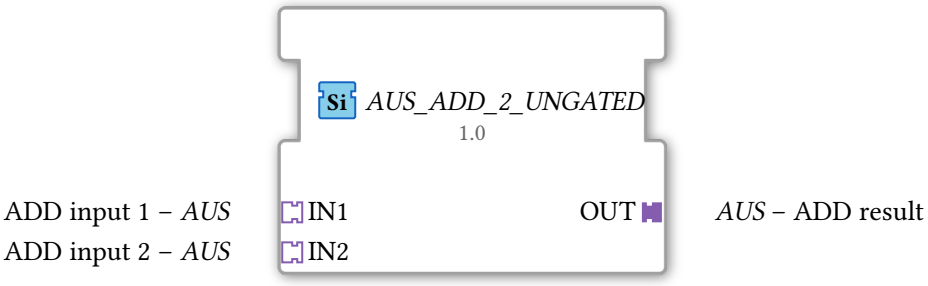
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

`AUS_ADD_2` is an efficient, generic auxiliary block for arithmetic addition in IEC 61499 applications. Its consistent use of unidirectional adapters makes it ideally suited for modern, modular software architectures in industrial automation.

AUS_ADD_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_ADD_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(Graphical representation of the function block not available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_ADD_2_UNGATED` is a generic function block (Generic FB) for performing arithmetic addition. It is used to add two input signals and output the result. The special feature of this block lies in the use of standardized, unidirectional adapters (`AUS`), which greatly simplifies and encapsulates the signal and data wiring within Eclipse 4diac.

Interface Structure

Event Inputs
This function block does not have direct event inputs. Event control is handled via the adapters.

Event Outputs
This function block does not have direct event outputs. Event control is handled via the adapters.

Data Inputs
This function block does not have direct data inputs.

Data Outputs
This function block does not have direct data outputs.

Adapters

Sockets (Input Adapters / Jacks)
- **IN1** (Type: `adapter::types::unidirectional::AUS`): The first addend of the addition.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): The second addend of the addition.

Plugs (Output Adapters)
- **OUT** (Type: `adapter::types::unidirectional::AUS`): The result of the addition (sum of `IN1` and `IN2`).

Functionality
The `AUS_ADD_2_UNGATED` module functions as an arithmetic adder based on adapters. As soon as a new event with a data value arrives at the input adapters `IN1` or `IN2`, the function block adds the two values:

$$OUT = IN1 + IN2 \tag{2.107}$$

The result, along with the corresponding trigger event, is output via the output adapter `OUT`. Since these are unidirectional adapters, the data and control flow is exclusively from the inputs to the output.

Technical Features
- **Generic Function Block:** The attribute `eclipse4diac::core::GenericClassName` with the value `'GEN_AUS_ADD'` defines the function block as a generic type. This allows for flexible adaptation to different data types defined by the adapter structures.
- **Encapsulation:** By using adapters instead of individual event and data pins, the system design becomes clearer and less prone to wiring errors.

State Overview
Since this component is essentially a mathematical operation, it does not have a complex internal state diagram (ECC). It reacts to incoming data at adapters `IN1` and `IN2`, calculates the sum, and forwards the result directly to `OUT`.

Application Scenarios
- **Measurement Processing:** Adding two sensor values or applying an offset to a measured value, which is supplied via an adapter.
- **Setpoint Calculation:** Combining base setpoints and correction values in process automation.
- – **Structured Programming:** Used in complex control systems where data and event lines are bundled in adapters to improve clarity.

Comparison with Similar Function Blocks
Compared to a standard addition function block (such as `ADD` from the IEC 61131-3 library), which has discrete data and event pins (such as `REQ`, `CNF`, `IN1`, `IN2`, `OUT`), `AUS_ADD_2_UNGATED` completely encapsulates these interfaces in adapters. This drastically reduces the number of visible connection lines in the 4diac editor and promotes object-oriented design.

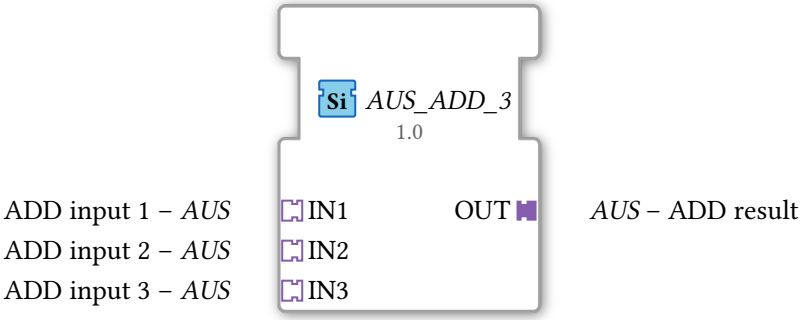
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

`AUS_ADD_2_UNGATED` is an efficient, generic auxiliary block for arithmetic addition in IEC 61499 applications. Its consistent use of unidirectional adapters makes it ideally suited for modern, modular software architectures in industrial automation.

AUS_ADD_3



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_ADD_3` is a generic arithmetic addition block for the 4diac IDE. It is used to add the values from three input adapters (`IN1`, `IN2`, and `IN3`) and output the result via an output adapter (`OUT`). The block uses unidirectional adapters of type `AUS` for this purpose.

Interface Structure

Event Inputs

No direct event inputs are available. Control and triggering are handled via the adapter interfaces.

Event Outputs

No direct event outputs are available. Event forwarding is handled via the adapter interfaces.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

The module communicates exclusively via adapter interfaces:
Sockets (Input Adapters):

- **IN1** (Type: `adapter::types::unidirectional::AUS`): First addend for addition.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): Second addend for addition.
- **IN3** (Type: `adapter::types::unidirectional::AUS`): Third addend for addition.

Plugs (Output Adapters):

- **OUT** (Type: `adapter::types::unidirectional::AUS`): Output adapter that carries the result of the addition (`IN1 + IN2 + IN3`).

Functionality

As soon as a new event or a changed data value is signaled at one of the input adapters (`IN1`, `IN2`, or `IN3`), the function block internally performs an addition of the values.

The calculation follows the mathematical formula:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.108}$$

After successful calculation, the result and the associated update event are forwarded to subsequent function blocks via the output adapter `OUT`.

Technical Features

- **Generic Function Block:** The function block is based on the generic class `GEN_AUS_ADD` (attribute `eclipse4diac::core::GenericClassName`). This allows for flexible adaptation to different data types within the `AUS` adapter.
- **Adapter-Based Architecture:** By using adapters instead of standard event/data connections, the wiring effort in the function block diagram is drastically reduced, as events and data are bundled in a single channel (the adapter).
- **Unidirectional Data Flow:** The adapters used are unidirectional, which defines a clear direction of data processing from inputs to output.

State Overview

Since it is a combinational (or stateless) function block, `AUS_ADD_3` has no complex internal states (no state machine). The processing is purely event-driven:

1. **Wait State:** The function block waits for an event at one of the inputs (`IN1`, `IN2`, `IN3`).
2. **Calculation:** Upon arrival of an event, the data is read and summed.
3. **Update:** The result is written to `OUT`, and the corresponding output event is triggered on the adapter.

Application Scenarios

- **Measurement Unit:** Combining and summing three analog sensor values (e.g., three flow meters, temperature sensors, or current consumers) distributed throughout the system via adapter interfaces.
- **Setpoint Generation:** Adding base setpoints, correction values, and offsets in control systems.

Comparison with Similar Function Blocks

- **Standard ADD (IEC 61131-3):** Classic `ADD` function blocks use dedicated event and data lines. `AUS_ADD_3`, on the other hand, encapsulates these in adapters, resulting in cleaner software architectures.
- **OFF_ADD_2 / OFF_ADD_4:** Compared to variants with two or four inputs, this function block is specifically optimized for exactly three input channels to avoid unused interfaces in the program code.

Change Detection

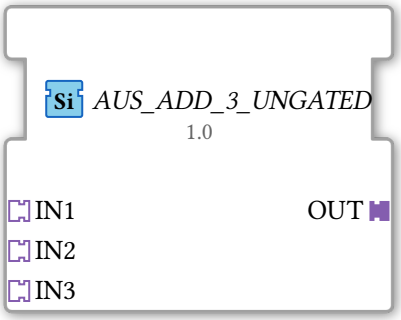
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUS_ADD_3` is an efficient and reusable function block for adding three signals. Through the consistent use of unidirectional adapters, it promotes a modular, clear, and maintainable application design within the IEC 61499 development environment.

AUS_ADD_3_UNGATED

i UNGATED variant: This block is the ungated version of `AUS_ADD_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



ADD input 1 – *AUS*
ADD input 2 – *AUS*
ADD input 3 – *AUS*

AUS – ADD result

(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_ADD_3_UNGATED` is a generic arithmetic addition block for the 4diac IDE. It is used to add the values from three input adapters (`IN1`, `IN2`, and `IN3`) and output the result via an output adapter (`OUT`). The block uses unidirectional adapters of type `AUS` for this purpose.

Interface Structure

Event Inputs

No direct event inputs are available. Control and triggering are handled via the adapter interfaces.

Event Outputs

No direct event outputs are available. Event forwarding is handled via the adapter interfaces.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

The module communicates exclusively via adapter interfaces:

Sockets (Input Adapters):

- **IN1** (Type: `adapter::types::unidirectional::AUS`): First addend for addition.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): Second addend for addition.
- **IN3** (Type: `adapter::types::unidirectional::AUS`): Third addend for addition.

Plugs (Output Adapters):

- **OUT** (Type: `adapter::types::unidirectional::AUS`): Output adapter that carries the result of the addition (`IN1 + IN2 + IN3`).

Functionality

As soon as a new event or a changed data value is signaled at one of the input adapters (`IN1`, `IN2`, or `IN3`), the function block internally performs an addition of the values.

The calculation follows the mathematical formula:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} \tag{2.109}$$

After successful calculation, the result and the associated update event are forwarded to subsequent function blocks via the output adapter `OUT`.

Technical Features

- **Generic Function Block:** The function block is based on the generic class `GEN_AUS_ADD` (attribute `eclipse4diac::core::GenericClassName`). This allows for flexible adaptation to different data types within the `AUS` adapter.

- **Adapter-Based Architecture:** By using adapters instead of standard event/data connections, the wiring effort in the function block diagram is drastically reduced, as events and data are bundled in a single channel (the adapter).
- **Unidirectional Data Flow:** The adapters used are unidirectional, which defines a clear direction of data processing from inputs to output.

State Overview

Since it is a combinational (or stateless) function block, `AUS_ADD_3_UNGATED` has no complex internal states (no state machine). The processing is purely event-driven:

1. **Wait State:** The function block waits for an event at one of the inputs (`IN1`, `IN2`, `IN3`).
2. **Calculation:** Upon arrival of an event, the data is read and summed.
3. **Update:** The result is written to `OUT`, and the corresponding output event is triggered on the adapter.

Application Scenarios

- **Measurement Unit:** Combining and summing three analog sensor values (e.g., three flow meters, temperature sensors, or current consumers) distributed throughout the system via adapter interfaces.
- **Setpoint Generation:** Adding base setpoints, correction values, and offsets in control systems.

Comparison with Similar Function Blocks

- **Standard ADD (IEC 61131-3):** Classic `ADD` function blocks use dedicated event and data lines. `AUS_ADD_3_UNGATED`, on the other hand, encapsulates these in adapters, resulting in cleaner software architectures.
- **OFF_ADD_2 / OFF_ADD_4:** Compared to variants with two or four inputs, this function block is specifically optimized for exactly three input channels to avoid unused interfaces in the program code.

Change Detection

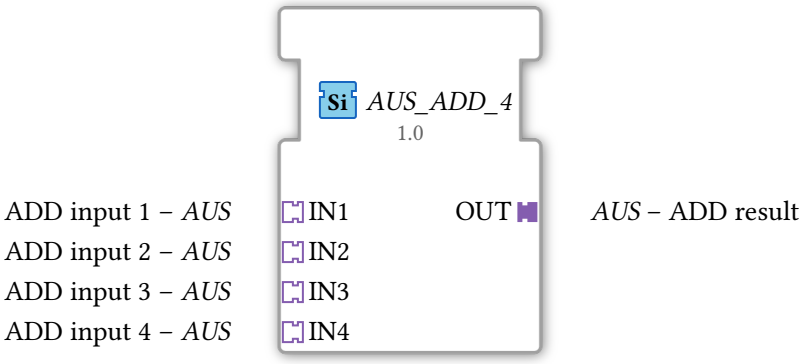
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUS_ADD_3_UNGATED` is an efficient and reusable function block for adding three signals. Through the consistent use of unidirectional adapters, it promotes a modular, clear, and maintainable application design within the IEC 61499 development environment.

AUS_ADD_4

+--+
| AUS_ADD_4 | IN1 ==# (Socket) (Plug) #== OUT IN2 ==# (Socket) | IN3 ==# (Socket) | IN4 ==# (Socket) |



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_ADD_4` performs the arithmetic addition of four input values. It is a generic function block (FB) specifically designed for use with unidirectional adapters. By encapsulating the signals within adapters, the block enables a clean and clear structuring of data flows within IEC 61499 applications.

Interface Structure

Event Inputs

This function block does not have direct, traditional event inputs. Event control is handled implicitly via the adapter interfaces.

Event Outputs

This function block does not have direct, traditional event outputs. Event forwarding occurs via the adapter interfaces.

Data Inputs

There are no direct data inputs. Data transmission occurs exclusively via the assigned adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUS`): First addend of the addition.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): Second addend of the addition.
- **IN3** (Type: `adapter::types::unidirectional::AUS`): Third addend of the addition.
- **IN4** (Type: `adapter::types::unidirectional::AUS`): Fourth addend of the addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`): Interface for outputting the calculated addition result.

Functionality

The function block `AUS_ADD_4` receives numerical values continuously or event-driven via its four input adapters (`IN1` to `IN4`). As soon as data is present at the sockets or an update event arrives, the function block performs the arithmetic addition of the four values:

$$\text{OUT} = \text{IN1} + \text{IN2} + \text{IN3} + \text{IN4} \tag{2.110}$$

The result of this calculation is immediately passed to the output plug `OUT` and made available for subsequent program components. Since these are unidirectional connections, data flows strictly in a directed direction from the inputs to the output.

Technical Features

- **Generic Block:** By being assigned to the generic class `GEN_AUS_ADD`, the block is highly reusable and adapts flexibly to the underlying data types of the adapters.
- **Reduced Routing Complexity:** By using adapters instead of individual event and data lines, the number of connection lines in the 4diac IDE is drastically reduced, which significantly improves the readability of large application diagrams.
- **Compiler Membership:** The block is organized in the package `adapter::iec61131::arithmetic`.

State Overview

The block behaves like a combinational component (or a stateless function block). It does not store historical values between calculation cycles. Each update to one of the input adapters directly triggers the calculation and updates the output.

Application Scenarios

- **Signal Summation:** Combining and adding four analog sensor values (e.g., determining the total flow rate from four individual flow meters or the total power consumption of four electrical devices).
- **Average Preparation:** Summing four measurement points as a preparatory step for subsequent division to calculate the average.
- **Combinatorial Control Logic:** Aggregating weighted control signals in more complex distributed systems.

Comparison with Similar Function Blocks

Compared to a standard adder (e.g., the classic `ADD` function block from the IEC 61131-3 library), which works directly with elementary data types (such as `REAL` or `INT`), `AUS_ADD_4` relies entirely on adapter coupling. This saves time during instantiation and wiring, but requires that the source and destination signals are encapsulated in the adapter type `AUS`.

Change Detection

The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

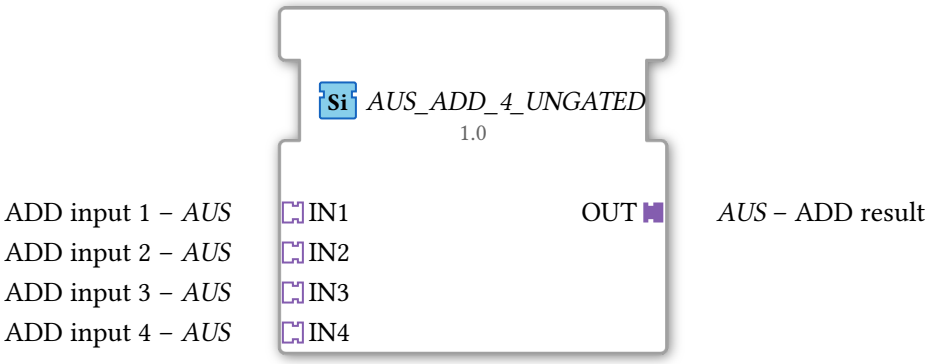
Conclusion

The `AUS_ADD_4` is a specialized and efficient auxiliary function block for structured application development in 4diac. It is ideally suited for cleanly structured control applications where signal processing is to be consistently implemented via adapter pipelines.

AUS_ADD_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_ADD_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.

+--+
| AUS_ADD_4_UNGATED | IN1 ==# (Socket) (Plug) #== OUT IN2 ==# (Socket) | IN3 ==# (Socket) | IN4 ==# (Socket) |



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_ADD_4_UNGATED` performs the arithmetic addition of four input values. It is a generic function block (FB) specifically designed for use with unidirectional adapters. By encapsulating the signals within adapters, the block enables a clean and clear structuring of data flows within IEC 61499 applications.

Interface Structure

Event Inputs

This function block does not have direct, traditional event inputs. Event control is handled implicitly via the adapter interfaces.

Event Outputs

This function block does not have direct, traditional event outputs. Event forwarding occurs via the adapter interfaces.

Data Inputs

There are no direct data inputs. Data transmission occurs exclusively via the assigned adapters.

Data Outputs

There are no direct data outputs. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUS`): First addend of the addition.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): Second addend of the addition.
- **IN3** (Type: `adapter::types::unidirectional::AUS`): Third addend of the addition.
- **IN4** (Type: `adapter::types::unidirectional::AUS`): Fourth addend of the addition.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`): Interface for outputting the calculated addition result.

Functionality

The function block `AUS_ADD_4_UNGATED` receives numerical values continuously or event-driven via its four input adapters (`IN1` to `IN4`). As soon as data is present at the sockets or an update event arrives, the function block performs the arithmetic addition of the four values:

$$\mathrm{OUT} = \mathrm{IN1} + \mathrm{IN2} + \mathrm{IN3} + \mathrm{IN4} \tag{2.111}$$

The result of this calculation is immediately passed to the output plug `OUT` and made available for subsequent program components. Since these are unidirectional connections, data flows strictly in a directed direction from the inputs to the output.

Technical Features

- **Generic Block:** By being assigned to the generic class `GEN_AUS_ADD`, the block is highly reusable and adapts flexibly to the underlying data types of the adapters.
- **Reduced Routing Complexity:** By using adapters instead of individual event and data lines, the number of connection lines in the 4diac IDE is drastically reduced, which significantly improves the readability of large application diagrams.
- **Compiler Membership:** The block is organized in the package `adapter::iec61131::arithmetic`.

State Overview

The block behaves like a combinational component (or a stateless function block). It does not store historical values between calculation cycles. Each update to one of the input adapters directly triggers the calculation and updates the output.

Application Scenarios

- **Signal Summation:** Combining and adding four analog sensor values (e.g., determining the total flow rate from four individual flow meters or the total power consumption of four electrical devices).
- **Average Preparation:** Summing four measurement points as a preparatory step for subsequent division to calculate the average.
- **Combinatorial Control Logic:** Aggregating weighted control signals in more complex distributed systems.

Comparison with Similar Function Blocks

Compared to a standard adder (e.g., the classic `ADD` function block from the IEC 61131-3 library), which works directly with elementary data types (such as `REAL` or `INT`), `AUS_ADD_4_UNGATED` relies entirely on adapter coupling. This saves time during instantiation and wiring, but requires that the source and destination signals are encapsulated in the adapter type `AUS`.

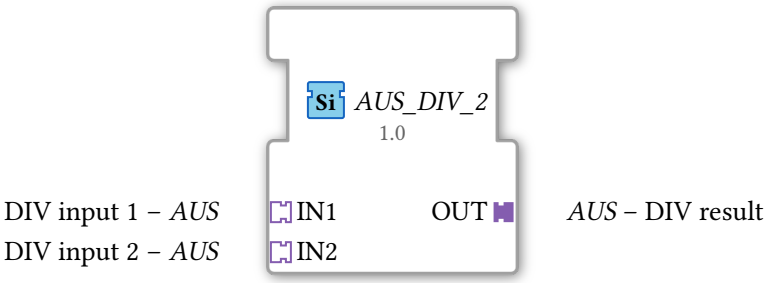
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUS_ADD_4_UNGATED` is a specialized and efficient auxiliary function block for structured application development in 4diac. It is ideally suited for cleanly structured control applications where signal processing is to be consistently implemented via adapter pipelines.

AUS_DIV_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) **AUS_DIV_2** is a generic block for performing arithmetic division (DIV). It is designed for use in IEC 61499-compliant systems (such as Eclipse 4diac). The block uses unidirectional adapters of type `AUS` to receive the input data, perform the division, and forward the result via a corresponding output adapter. Encapsulating the signals in adapters significantly simplifies the wiring within the development environment.

Interface Structure

Event Inputs

There are no direct event inputs on the block interface. Event-based control and triggering of the calculation are handled internally via the connected input adapters.

Event Outputs

There are no direct event outputs on the block interface. Calculation and update events are forwarded via the output adapter.

Data Inputs

The block has no direct elementary data inputs. All input data is passed via the adapter interfaces.

Data Outputs

The block has no direct elementary data outputs. The calculation result is provided via the output adapter.

Adapters

Type	Name	Adapter Type	Description
Socket (Input)	`IN1`	`adapter::types::unidirectional::AUS`	First input value for division (Dividend).
Socket (Input)	`IN2`	`adapter::types::unidirectional::AUS`	Second input value for the division (divisor).
Plug (Output)	`OUT`	`adapter::types::unidirectional::AUS`	Output for the result of the division (quotient).

Functionality

The function block `AUS_DIV_2` performs a classic division according to the following mathematical principle:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.112}$$

Since it is a generic function block (identified by the attribute `GEN_AUS_DIV`), the block is not fixed to a specific data type (such as `INT` or `REAL`). The actual data type is resolved during instantiation in the 4diac IDE based on the connected adapters.

As soon as a new data event is signaled at the input adapters `IN1` or `IN2`, the function block performs the division and updates the value at the output adapter `OUT`, followed by a corresponding release event via the output plug.

- **Generic Type (`GEN_AUS_DIV`):** Enables flexible reuse for different numeric data types without the need to create separate function blocks for `INT`, `REAL`, or `LREAL`.
- **Unidirectional Adapters:** Using the `adapter::types::unidirectional::AUS` type ensures clean data and event encapsulation. This reduces visual complexity in system design (less "spaghetti code" due to loosely connected event and data lines).
- **Division by Zero:** When implementing on the target platform (runtime), it is important to consider how the function block reacts to a divisor of `0` (e.g., outputting `NaN`/`INF` for floating-point numbers or a system error for integers).

The function block does not have a complex internal state diagram (ECC). Its execution is purely transactional:

Technical Features

1. **Wait State:** The function block waits for an update event at sockets `IN1` or `IN2`.
2. **Calculation:** Upon receiving an event, the values are read and divided.
3. **Output:** The result is written to the plug `OUT` and the output event is triggered. The function block immediately returns to the wait state.

State Overview

- **Measurement Scaling:** Division of sensor values by constant factors distributed across the system via adapter structures.
- **Average Calculation:** Use in mathematical computation networks within distributed control systems.
- **Ratio Control:** Calculation of ratios (e.g., air-fuel ratio in burner controls) where the input signals are already available as standardized `AUS` adapters.

Application Scenarios

Compared to a standard integer division function block (such as the IEC 61131-3 `DIV` block), `AUS_DIV_2` offers the following advantages:

Comparison with Similar Function Blocks

- **Adapter-Based Coupling:** Standard function blocks require separate pins for data and events (REQ/CNF). `AUS_DIV_2` logically combines these in the `AUS` adapters.
- **Data Type Flexibility:** While classic function blocks are often rigidly defined for, e.g., `REAL` (e.g., `R_DIV`), this generic function block adapts to the adapter type used.
- **Adapter-Based Coupling:** Standard function blocks require separate pins for data and events (REQ/CNF). `AUS_DIV_2` logically combines these in the `AUS` adapters.
- **Data Type Flexibility:** While classic function blocks are often fixed for, e.g., `REAL` (e.g., `R_DIV`), this generic function block adapts to the adapter type used.

Change Detection

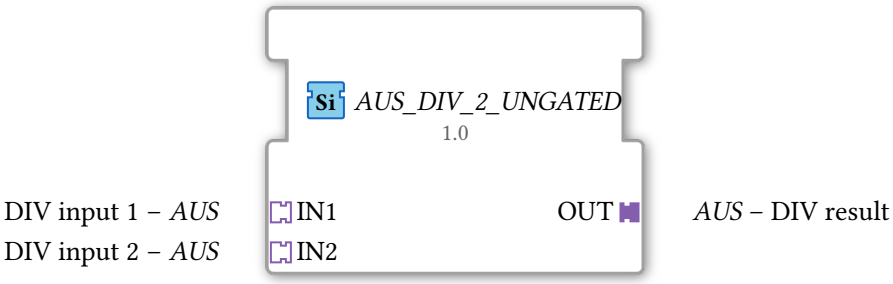
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The function block `AUS_DIV_2` is a highly efficient, reusable auxiliary block for arithmetic calculations in modern, adapter-based IEC 61499 architectures. It significantly improves the clarity of application diagrams by consolidating the signal and event flows for mathematical divisions in standardized adapters.

AUS_DIV_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUS_DIV_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) **AUS_DIV_2_UNGATED** is a generic block for performing arithmetic division (DIV). It is designed for use in IEC 61499-compliant systems (such as Eclipse 4diac). The block uses unidirectional adapters of type `AUS` to receive the input data, perform the division, and forward the result via a corresponding output adapter. Encapsulating the signals in adapters significantly simplifies the wiring within the development environment.

Interface Structure

Event Inputs

There are no direct event inputs on the block interface. Event-based control and triggering of the calculation are handled internally via the connected input adapters.

Event Outputs

There are no direct event outputs on the block interface. Calculation and update events are forwarded via the output adapter.

Data Inputs

The block has no direct elementary data inputs. All input data is passed via the adapter interfaces.

Data Outputs

The block has no direct elementary data outputs. The calculation result is provided via the output adapter.

Adapters

Type	Name	Adapter Type	Description
Socket (Input)	`IN1`	`adapter::types::unidirectional::AUS`	First input value for division (Dividend).
Socket (Input)	`IN2`	`adapter::types::unidirectional::AUS`	Second input value for the division (divisor).
Plug (Output)	`OUT`	`adapter::types::unidirectional::AUS`	Output for the result of the division (quotient).

Functionality

The function block AUS_DIV_2_UNGATED performs a classic division according to the following mathematical principle:

$$\mathrm{OUT} = \frac{\mathrm{IN1}}{\mathrm{IN2}} \tag{2.113}$$

Since it is a generic function block (identified by the attribute GEN_AUS_DIV), the block is not fixed to a specific data type (such as INT or REAL). The actual data type is resolved during instantiation in the 4diac IDE based on the connected adapters.

As soon as a new data event is signaled at the input adapters IN1 or IN2, the function block performs the division and updates the value at the output adapter OUT, followed by a corresponding release event via the output plug.

- **Generic Type (GEN_AUS_DIV):** Enables flexible reuse for different numeric data types without the need to create separate function blocks for INT, REAL, or LREAL.
- **Unidirectional Adapters:** Using the adapter::types::unidirectional::AUS type ensures clean data and event encapsulation. This reduces visual complexity in system design (less "spaghetti code" due to loosely connected event and data lines).
- **Division by Zero:** When implementing on the target platform (runtime), it is important to consider how the function block reacts to a divisor of 0 (e.g., outputting NaN/INF for floating-point numbers or a system error for integers).

The function block does not have a complex internal state diagram (ECC). Its execution is purely transactional:

Technical Features

1. **Wait State:** The function block waits for an update event at sockets IN1 or IN2.
2. **Calculation:** Upon receiving an event, the values are read and divided.
3. **Output:** The result is written to the plug OUT and the output event is triggered. The function block immediately returns to the wait state.

State Overview

- **Measurement Scaling:** Division of sensor values by constant factors distributed across the system via adapter structures.
- **Average Calculation:** Use in mathematical computation networks within distributed control systems.
- **Ratio Control:** Calculation of ratios (e.g., air-fuel ratio in burner controls) where the input signals are already available as standardized AUS adapters.

Application Scenarios

Compared to a standard integer division function block (such as the IEC 61131-3 DIV block), AUS_DIV_2_UNGATED offers the following advantages:

Comparison with Similar Function Blocks

- **Adapter-Based Coupling:** Standard function blocks require separate pins for data and events (REQ/CNF). AUS_DIV_2_UNGATED logically combines these in the AUS adapters.
- **Data Type Flexibility:** While classic function blocks are often rigidly defined for, e.g., REAL (e.g., R_DIV), this generic function block adapts to the adapter type used.
- **Adapter-Based Coupling:** Standard function blocks require separate pins for data and events (REQ/CNF). AUS_DIV_2_UNGATED logically combines these in the AUS adapters.
- **Data Type Flexibility:** While classic function blocks are often fixed for, e.g., REAL (e.g., R_DIV), this generic function block adapts to the adapter type used.

Change Detection

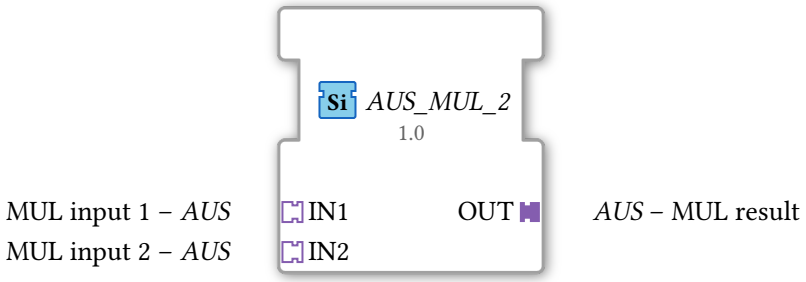
This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The function block AUS_DIV_2_UNGATED is a highly efficient, reusable auxiliary block for arithmetic calculations in modern, adapter-based IEC 61499 architectures. It significantly improves the clarity of

application diagrams by consolidating the signal and event flows for mathematical divisions in standardized adapters.

AUS_MUL_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_MUL_2` is a generic arithmetic block for the 4diac IDE, used for multiplying two values. Unlike classic mathematical function blocks, this block uses an adapter-based concept for data transmission. This enables structured, modular, and clear signal wiring in complex IEC 61499 applications.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Event control is handled via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Event forwarding is handled via the output adapter.

Data Inputs

This function block has no direct data inputs.

Data Outputs

This function block has no direct data outputs.

Adapters

All communication (data and associated trigger events) is implemented via unidirectional adapters of type `AUS`.

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUS`): The first multiplicand (input 1).
- **IN2** (Type: `adapter::types::unidirectional::AUS`): The second multiplicand (input 2).

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`): The result of the multiplication ($IN1 \times IN2$).

–

Functionality

As soon as a new event indicating new data is signaled at the input adapters `IN1` or `IN2`, the function block internally performs the multiplication:

$$OUT = \text{IN1} \times \text{IN2} \tag{2.114}$$

The result of the calculation and the corresponding confirmation event are then passed on to subsequent function blocks via the output plug `OUT`. Since this is a generic function block (`GEN_AUS_MUL`), the internal processing adapts flexibly to the data type defined in the adapter.

Technical Features

- **Generic Nature:** The function block is declared as a generic type (`GenericClassName = 'GEN_AUS_MUL'`). This allows for flexible handling of various numeric data types (e.g., `INT`, `REAL`, `LREAL`) defined by the adapter structure.
- **Adapter Focus:** Reducing the number of traditional pins to adapters significantly minimizes the wiring effort in the function block diagram and ensures a clean, object-oriented design.
-

State Overview

The function block operates in an event-driven manner based on the state changes of the adapters:

1. **IDLE (Standby):** The function block waits for incoming values/events at sockets `IN1` and `IN2`.
2. **CALCULATION:** Upon receiving a trigger, the data values contained in the adapters are multiplied.
3. **OUTPUT:** The product is written to adapter `OUT`, and a send event is triggered. The function block returns to the *IDLE* state.

–

Application Scenarios

- **Measurement Scaling:** Multiplication of a raw value (e.g., from a sensor adapter) by a calibration factor.
- **Power Calculation:** Multiplication of current and voltage values read in via standardized adapter interfaces.
- **Modular Signal Processing:** Use in complex control loops where signal chains are neatly encapsulated by adapters to maintain clarity in the control diagram.

Comparison with Similar Components

Compared to the standard IEC 61131-3 multiplication component (`MUL`), which uses individual pins for `REQ`, `CNF`, and direct data inputs and outputs, `AUS_MUL_2` bundles all these signals into adapters. This prevents "cable clutter" in the function block diagram but requires that the connected signals are already in the `AUS` adapter format.

Change Detection

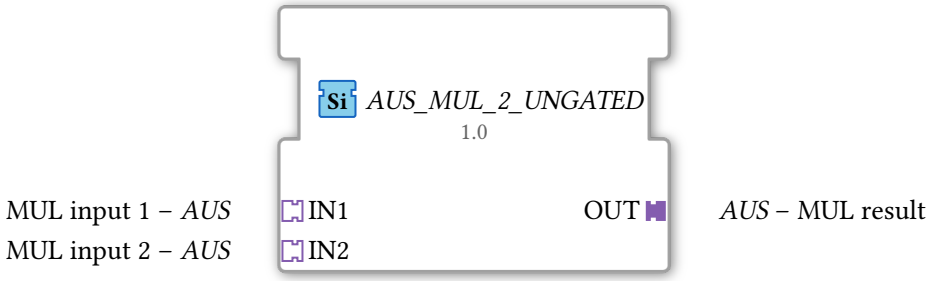
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUS_MUL_2` is a modern, robust, and reusable function block for multiplication in the 4diac IDE. Its consistent use of unidirectional adapters makes it ideally suited for service-oriented architectures and structured application designs in industrial environments.

AUS_MUL_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_MUL_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_MUL_2_UNGATED` is a generic arithmetic block for the 4diac IDE, used for multiplying two values. Unlike classic mathematical function blocks, this block uses an adapter-based concept for data transmission. This enables structured, modular, and clear signal wiring in complex IEC 61499 applications.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Event control is handled via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Event forwarding is handled via the output adapter.

Data Inputs

This function block has no direct data inputs.

Data Outputs

This function block has no direct data outputs.

Adapters

All communication (data and associated trigger events) is implemented via unidirectional adapters of type `AUS`.

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUS`): The first multiplicand (input 1).
- **IN2** (Type: `adapter::types::unidirectional::AUS`): The second multiplicand (input 2).

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`): The result of the multiplication ($IN1 \times IN2$).

–

Functionality

As soon as a new event indicating new data is signaled at the input adapters `IN1` or `IN2`, the function block internally performs the multiplication:

$$OUT = \text{IN1} \times \text{IN2} \tag{2.115}$$

The result of the calculation and the corresponding confirmation event are then passed on to subsequent function blocks via the output plug `OUT`. Since this is a generic function block (`GEN_AUS_MUL`), the internal processing adapts flexibly to the data type defined in the adapter.

Technical Features

- **Generic Nature:** The function block is declared as a generic type (`GenericClassName = 'GEN_AUS_MUL'`). This allows for flexible handling of various numeric data types (e.g., `INT`, `REAL`, `LREAL`) defined by the adapter structure.
- **Adapter Focus:** Reducing the number of traditional pins to adapters significantly minimizes the wiring effort in the function block diagram and ensures a clean, object-oriented design.
-

State Overview

The function block operates in an event-driven manner based on the state changes of the adapters:

1. **IDLE (Standby):** The function block waits for incoming values/events at sockets `IN1` and `IN2`.
2. **CALCULATION:** Upon receiving a trigger, the data values contained in the adapters are multiplied.
3. **OUTPUT:** The product is written to adapter `OUT`, and a send event is triggered. The function block returns to the *IDLE* state.

–

Application Scenarios

- **Measurement Scaling:** Multiplication of a raw value (e.g., from a sensor adapter) by a calibration factor.
- **Power Calculation:** Multiplication of current and voltage values read in via standardized adapter interfaces.
- **Modular Signal Processing:** Use in complex control loops where signal chains are neatly encapsulated by adapters to maintain clarity in the control diagram.

Comparison with Similar Components

Compared to the standard IEC 61131-3 multiplication component (`MUL`), which uses individual pins for `REQ`, `CNF`, and direct data inputs and outputs, `AUS_MUL_2_UNGATED` bundles all these signals into adapters. This prevents "cable clutter" in the function block diagram but requires that the connected signals are already in the `AUS` adapter format.

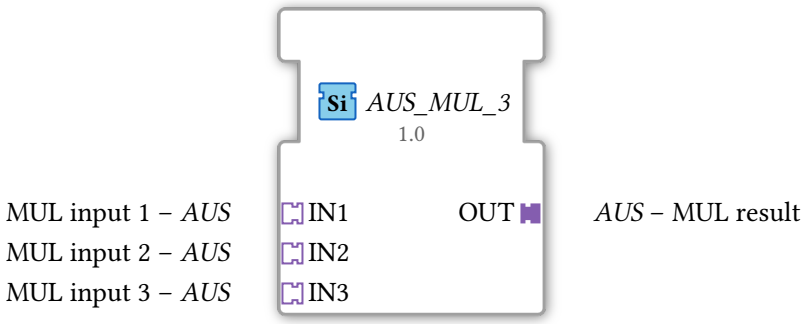
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUS_MUL_2_UNGATED` is a modern, robust, and reusable function block for multiplication in the 4diac IDE. Its consistent use of unidirectional adapters makes it ideally suited for service-oriented architectures and structured application designs in industrial environments.

AUS_MUL_3



• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) `AUS_MUL_3` is used for the arithmetic multiplication of three input values. It is a generic function block (`Generic FB`) based on the use of unidirectional adapters of type `AUS`. The block calculates the product of three input values and provides the result at an output adapter.

Since this function block is entirely based on adapters, it has no direct, traditional event or data interfaces at the block level. All communication and synchronization takes place via the declared adapters.

Interface Structure

Event Inputs

No direct event inputs available. (Event control is handled via the adapter sockets).

Event Outputs

No direct event outputs available. (Event control is handled via the adapter plug).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUS`): The first multiplicand for the arithmetic operation.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): The second multiplicand for the arithmetic operation.
- **IN3** (Type: `adapter::types::unidirectional::AUS`): The third multiplicand for the arithmetic operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`): The calculated result of the multiplication ($OUT = IN1 \times IN2 \times IN3$).

Functionality

As soon as new values are present at the input adapters (`IN1`, `IN2`, `IN3`) and a corresponding trigger event is transmitted via the adapters, the function block executes the multiplication.

The mathematical formula is:

$$\text{Result} = \text{Value of IN1} \times \text{Value of IN2} \times \text{Value of IN3} \tag{2.116}$$

The calculated result, along with a corresponding confirmation event, is passed to subsequent program components via the output plug `OUT`.

Technical Features

- **Generic Module:** The attribute `GenericClassName` with the value `'GEN_AUS_MUL'` allows for flexible use of the module. Depending on the specific implementation of the adapter type `AUS`, the function block can adapt to various data types (e.g., `INT`, `REAL`, `LREAL`).
- **Encapsulation by Adapter:** The use of unidirectional adapters (`unidirectional::AUS`) drastically reduces the routing of individual event and data lines in the 4diac IDE. This results in a cleaner and more organized application design.

State Overview

The module behaves in a purely stateless manner (i.e., it reacts to incoming adapter signals in an event-driven manner):

Application Scenarios

1. **Wait State:** The module waits for an update event at sockets `IN1`, `IN2`, or `IN3`.

2. **Calculation:** Upon receiving an event, the data values of the three input adapters are read and multiplied together.
3. **Output:** The result is written to plug `OUT`, and the corresponding send event of the plug is triggered.

Comparison with Similar Function Blocks

- **Scaling and Weighting:** Multiplication of a sensor value by two consecutive scaling factors (e.g., physical conversion and calibration factor).
- **Volume Calculation:** Calculating three-dimensional volume or flow rate where three parameters (e.g., length × width × height or three different rates) must be directly multiplied together.
- **Structured Signal Processing:** Used in more complex control loops that require a clean signal architecture using adapters.
- **Standard MUL Component (IEC 61131-3):** The classic `MUL` component uses direct data and event pins. The `AUS_MUL_3`, on the other hand, bundles these signals in adapters, simplifying wiring in large projects.
- **AUS_MUL_2 (Two-Input Variant):** While a hypothetical two-input variant can only multiply two factors, `AUS_MUL_3` eliminates the need for an additional cascading block when multiplying three values.

Change Detection

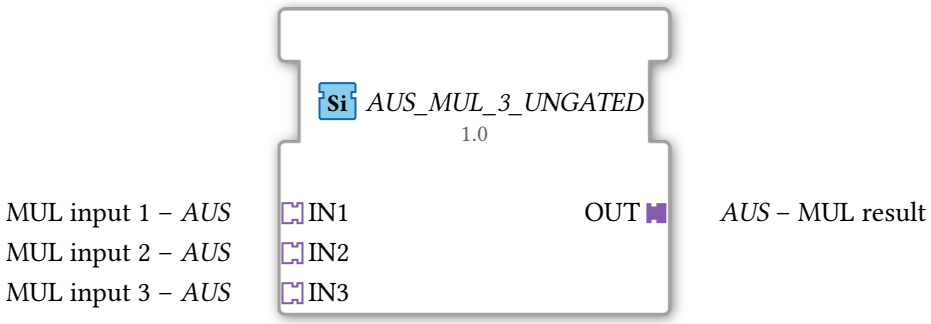
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUS_MUL_3` is a highly efficient, generic function block for multiplying three values. Its consistent use of unidirectional adapters makes it ideally suited for modern, adapter-based software architectures in the 4diac environment, as it minimizes the visual complexity of control diagrams.

AUS_MUL_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_MUL_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) `AUS_MUL_3_UNGATED` is used for the arithmetic multiplication of three input values. It is a generic function block (`Generic FB`) based on the use of unidirectional adapters of type `AUS`. The block calculates the product of three input values and provides the result at an output adapter.

Since this function block is entirely based on adapters, it has no direct, traditional event or data interfaces at the block level. All communication and synchronization takes place via the declared adapters.

Interface Structure

Event Inputs

No direct event inputs available. (Event control is handled via the adapter sockets).

Event Outputs

No direct event outputs available. (Event control is handled via the adapter plug).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUS`): The first multiplicand for the arithmetic operation.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): The second multiplicand for the arithmetic operation.
- **IN3** (Type: `adapter::types::unidirectional::AUS`): The third multiplicand for the arithmetic operation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`): The calculated result of the multiplication ($OUT = IN1 \times IN2 \times IN3$).

Functionality

As soon as new values are present at the input adapters (`IN1`, `IN2`, `IN3`) and a corresponding trigger event is transmitted via the adapters, the function block executes the multiplication.

The mathematical formula is:

$$\text{Result} = \text{Value of IN1} \times \text{Value of IN2} \times \text{Value of IN3} \tag{2.117}$$

The calculated result, along with a corresponding confirmation event, is passed to subsequent program components via the output plug `OUT`.

Technical Features

- **Generic Module:** The attribute `GenericClassName` with the value `'GEN_AUS_MUL'` allows for flexible use of the module. Depending on the specific implementation of the adapter type `AUS`, the function block can adapt to various data types (e.g., `INT`, `REAL`, `LREAL`).
- **Encapsulation by Adapter:** The use of unidirectional adapters (`unidirectional::AUS`) drastically reduces the routing of individual event and data lines in the 4diac IDE. This results in a cleaner and more organized application design.

State Overview

The module behaves in a purely stateless manner (i.e., it reacts to incoming adapter signals in an event-driven manner):

Application Scenarios

1. **Wait State:** The module waits for an update event at sockets `IN1`, `IN2`, or `IN3`.
2. **Calculation:** Upon receiving an event, the data values of the three input adapters are read and multiplied together.
3. **Output:** The result is written to plug `OUT`, and the corresponding send event of the plug is triggered.

Comparison with Similar Function Blocks

- **Scaling and Weighting:** Multiplication of a sensor value by two consecutive scaling factors (e.g., physical conversion and calibration factor).

- **Volume Calculation:** Calculating three-dimensional volume or flow rate where three parameters (e.g., length × width × height or three different rates) must be directly multiplied together.
- **Structured Signal Processing:** Used in more complex control loops that require a clean signal architecture using adapters.
- **Standard MUL Component (IEC 61131-3):** The classic `MUL` component uses direct data and event pins. The `AUS_MUL_3_UNGATED`, on the other hand, bundles these signals in adapters, simplifying wiring in large projects.
- **AUS_MUL_2 (Two-Input Variant):** While a hypothetical two-input variant can only multiply two factors, `AUS_MUL_3_UNGATED` eliminates the need for an additional cascading block when multiplying three values.

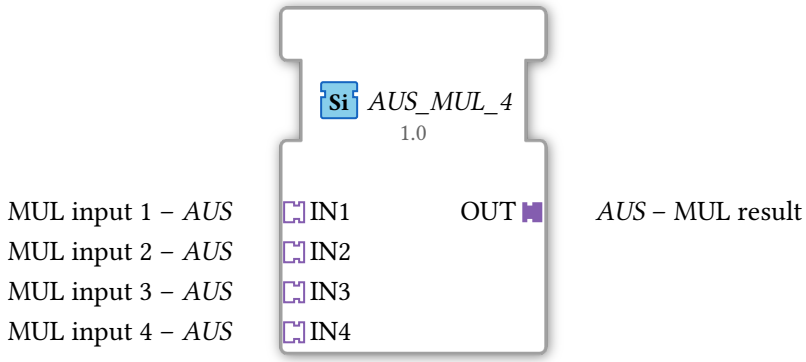
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUS_MUL_3_UNGATED` is a highly efficient, generic function block for multiplying three values. Its consistent use of unidirectional adapters makes it ideally suited for modern, adapter-based software architectures in the 4diac environment, as it minimizes the visual complexity of control diagrams.

AUS_MUL_4



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_MUL_4` is a generic function block (Generic FB) for performing an arithmetic multiplication (`MUL`) with up to four input values. It is based on the IEC 61499 architecture and uses unidirectional adapters to transmit data and the associated control events in a clean, encapsulated manner.

Due to its generic nature (`GEN_AUS_MUL`), the function block is flexible and can be applied to various numeric data types.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Event control is handled entirely via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. Data is transferred via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUS`): First input value (multiplicand 1) for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): Second input value (multiplicand 2) for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::AUS`): Third input value (multiplicand 3) for multiplication.
- **IN4** (Type: `adapter::types::unidirectional::AUS`): Fourth input value (multiplicand 4) for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`): The calculated product (result of the multiplication).

Functionality

As soon as new data values and the corresponding trigger events are present at the input adapters (`IN1` to `IN4`), the function block performs the multiplication.

The mathematical calculation follows the formula:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.118}$$

The calculated result, along with a corresponding output event, is provided via the output adapter `OUT`. Since this is a generic function block, the actual data type (e.g., `INT`, `REAL`, `LREAL`) adapts to the instantiation in the 4diac IDE.

Technical Features

- **Generic Function Block:** By defining it as `GEN_AUS_MUL`, the function block is not tied to a specific data type, which drastically increases its reusability across different projects.
- **Adapter-Based Communication:** The use of unidirectional `AUS` adapters ensures a clear application diagram, as data and event lines are bundled into a single connection. This reduces wiring errors in the development environment.

–

State Overview

The function block behaves like a classic, stateless mathematical operator (combinatorial logic). It does not store any internal states between calculation cycles. Each update of the input values via the adapters triggers a recalculation and update of the output.

Application Scenarios

- **Scaling and Calibration:** Calculation of more complex physical values where multiple factors (e.g., sensor value × calibration factor × temperature compensation × unit conversion) need to be multiplied.
- **Power Calculation:** Combination of multiple measured variables in process engineering.
- **Multi-Stage Gain Control:** Cascading of gain factors in control algorithms.

–

Comparison with Similar Components

Compared to standard multiplication components (such as the classic `MUL` of IEC 61131-3), which use separate pins for events and data, the `AUS_MUL_4` offers the following advantages:

- **Fewer Connections:** Four separate inputs and one output would require at least 10 connections (5 events, 5 data lines) in the classic model. The adapter technology reduces this to 5 adapter connections.
- **Compactness:** Cascading multiple 2x multipliers is no longer necessary, as four operands can be processed directly in a single function block.

Change Detection

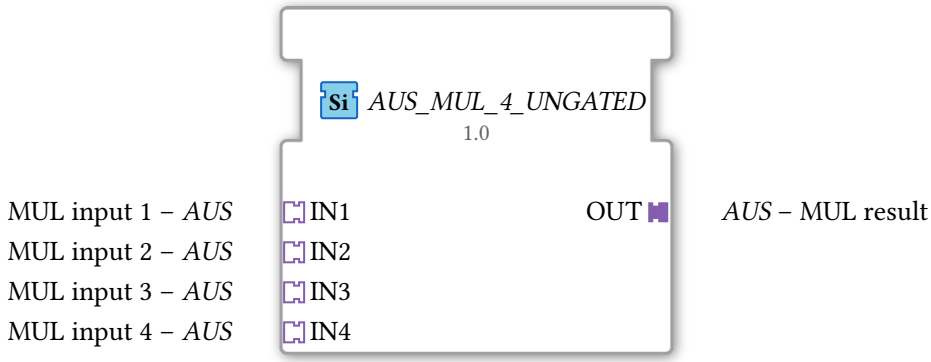
The result is only written to the output plug (`OUT`) and its adapter event only sent if the newly computed value differs from the value currently held on `OUT`. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The `AUS_MUL_4` is a highly efficient and modern function block for the 4diac IDE. It combines the fundamental mathematical function of multiplication with the modern structuring advantages of IEC 61499 adapters, resulting in cleaner, more maintainable, and more transparent control applications.

AUS_MUL_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_MUL_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AUS_MUL_4_UNGATED` is a generic function block (Generic FB) for performing an arithmetic multiplication (`MUL`) with up to four input values. It is based on the IEC 61499 architecture and uses unidirectional adapters to transmit data and the associated control events in a clean, encapsulated manner.

Due to its generic nature (`GEN_AUS_MUL`), the function block is flexible and can be applied to various numeric data types.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Event control is handled entirely via the connected adapters.

Event Outputs

This function block does not have direct event outputs. Event control is handled entirely via the output adapter.

Data Inputs

There are no direct data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct data outputs. Data is transferred via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUS`): First input value (multiplicand 1) for multiplication.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): Second input value (multiplicand 2) for multiplication.
- **IN3** (Type: `adapter::types::unidirectional::AUS`): Third input value (multiplicand 3) for multiplication.
- **IN4** (Type: `adapter::types::unidirectional::AUS`): Fourth input value (multiplicand 4) for multiplication.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`): The calculated product (result of the multiplication).

Functionality

As soon as new data values and the corresponding trigger events are present at the input adapters (`IN1` to `IN4`), the function block performs the multiplication.

The mathematical calculation follows the formula:

$$\mathrm{OUT} = \mathrm{IN1} \times \mathrm{IN2} \times \mathrm{IN3} \times \mathrm{IN4} \tag{2.119}$$

The calculated result, along with a corresponding output event, is provided via the output adapter `OUT`. Since this is a generic function block, the actual data type (e.g., `INT`, `REAL`, `LREAL`) adapts to the instantiation in the 4diac IDE.

Technical Features

- **Generic Function Block:** By defining it as `GEN_AUS_MUL`, the function block is not tied to a specific data type, which drastically increases its reusability across different projects.
- **Adapter-Based Communication:** The use of unidirectional `AUS` adapters ensures a clear application diagram, as data and event lines are bundled into a single connection. This reduces wiring errors in the development environment.

–

State Overview

The function block behaves like a classic, stateless mathematical operator (combinatorial logic). It does not store any internal states between calculation cycles. Each update of the input values via the adapters triggers a recalculation and update of the output.

Application Scenarios

- **Scaling and Calibration:** Calculation of more complex physical values where multiple factors (e.g., sensor value × calibration factor × temperature compensation × unit conversion) need to be multiplied.
- **Power Calculation:** Combination of multiple measured variables in process engineering.
- **Multi-Stage Gain Control:** Cascading of gain factors in control algorithms.

–

Comparison with Similar Components

Compared to standard multiplication components (such as the classic `MUL` of IEC 61131-3), which use separate pins for events and data, the `AUS_MUL_4_UNGATED` offers the following advantages:

- **Fewer Connections:** Four separate inputs and one output would require at least 10 connections (5 events, 5 data lines) in the classic model. The adapter technology reduces this to 5 adapter connections.
- **Compactness:** Cascading multiple 2x multipliers is no longer necessary, as four operands can be processed directly in a single function block.

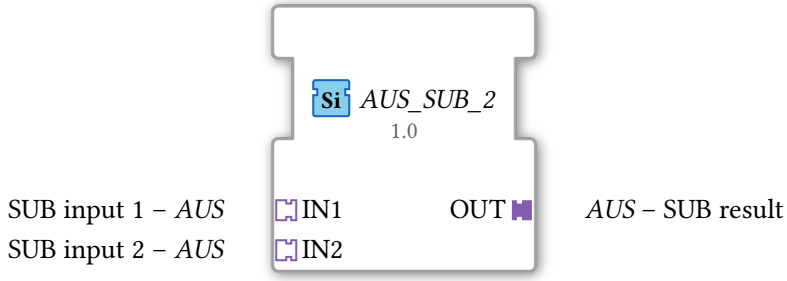
Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

Conclusion

The `AUS_MUL_4_UNGATED` is a highly efficient and modern function block for the 4diac IDE. It combines the fundamental mathematical function of multiplication with the modern structuring advantages of IEC 61499 adapters, resulting in cleaner, more maintainable, and more transparent control applications.

AUS_SUB_2



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) `AUS_SUB_2` is a generic block for performing arithmetic subtraction operations within 4diac-based control systems. The block uses unidirectional adapters of type `AUS` to encapsulate and transmit input and output signals. Due to its generic design (compilation class `GEN_AUS_SUB`), it can be flexibly used in various control scenarios to calculate the difference between two values.

Interface Structure

Event Inputs

This function block does not have direct, dedicated event inputs. Event control is handled entirely via the adapters used.

Event Outputs

This function block does not have direct, dedicated event outputs. Event control is handled entirely via the adapters used.

Data Inputs

There are no direct, basic data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct, basic data outputs. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: `adapter::types::unidirectional::AUS`): The first input (minuend) for subtraction calculation.
- **IN2** (Type: `adapter::types::unidirectional::AUS`): The second input (subtrahend) for subtraction calculation.

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`): The output (difference) that provides the calculated result of the subtraction.

Functionality

The primary function of AUS_SUB_2 is arithmetic subtraction:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.120}$$

Since the function block is based on adapters, it receives the values and the corresponding trigger events via the sockets IN1 and IN2. As soon as relevant data arrives via the input adapters, the mathematical operation is executed. The result of the calculation and the corresponding output event are then forwarded to the subsequent function blocks via the plug OUT.

Since this is a generic function block (GEN_AUS_SUB), the mathematical processing dynamically adapts to the data types defined in the AUS adapter.

Technical Features

- **Generic Type (GEN_AUS_SUB):** The function block is not limited to a single data type (such as only INT or REAL), but supports the data types defined by the adapter structure.
- **Unidirectional Adapters:** Using the unidirectional::AUS type ensures a clear, one-way data and signal flow. This minimizes the complexity of signal tracing within the system.
- – **Encapsulation:** By eliminating traditional event and data pins, the visual layout in the 4diac-ide Application Editor remains extremely compact and clear.

State Overview

Since this module is designed as a purely functional, stateless calculation block, it does not have a complex internal Execution Control Chart (ECC). Its execution is purely data- and event-driven, based on the interactions of the connected adapters.

Application Scenarios

- **Process Value Correction:** Subtraction of zero-point offsets or calibration values from a measured sensor value.
- **Target-Actual Value Comparison:** Calculation of the control deviation ($e = w - x$) in control loops where the signals are already available as structured adapter channels.
- **Level Calculation:** Determination of differential quantities in containers or systems by subtracting the outflow from the inflow.

Comparison with Similar Function Blocks

Compared to a classic IEC 61131-3 SUB function block, which works directly with elementary data types, the AUS_SUB_2 offers the following advantages:

- **Reduced Wiring Effort:** Event and data lines do not need to be run separately, as they are bundled in the AUS adapter.
- **Increased Modularity:** It is ideally suited for service-oriented architectures in IEC 61499, where subsystems typically communicate via adapters.

–

Change Detection

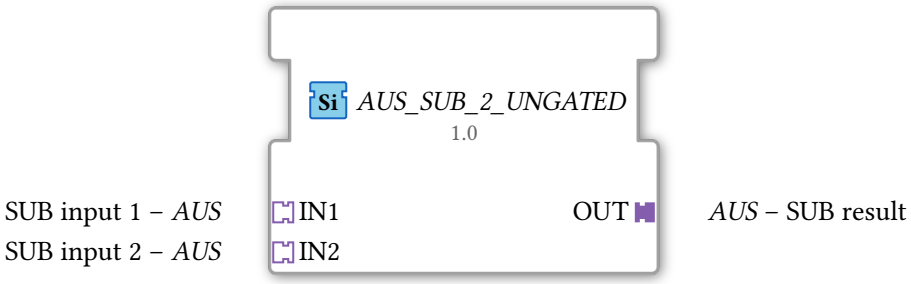
The result is only written to the output plug (OUT) and its adapter event only sent if the newly computed value differs from the value currently held on OUT. If the result is unchanged, no adapter event is sent, avoiding redundant updates on downstream peers.

Conclusion

The AUS_SUB_2 is a specialized yet flexible subtraction function block. Through the consistent use of unidirectional adapters, it promotes a clean, structured, and clear design of control programs in the 4diac IDE.

AUS_SUB_2_UNGATED

i UNGATED variant: This block is the ungated version of AUS_SUB_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



(No image available)

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) AUS_SUB_2_UNGATED is a generic block for performing arithmetic subtraction operations within 4diac-based control systems. The block uses unidirectional adapters of type AUS to encapsulate and transmit input and output signals. Due to its generic design (compilation class GEN_AUS_SUB), it can be flexibly used in various control scenarios to calculate the difference between two values.

Interface Structure

Event Inputs

This function block does not have direct, dedicated event inputs. Event control is handled entirely via the adapters used.

Event Outputs

This function block does not have direct, dedicated event outputs. Event control is handled entirely via the adapters used.

Data Inputs

There are no direct, basic data inputs. Data is transferred via the input adapters.

Data Outputs

There are no direct, basic data outputs. The result is provided via the output adapter.

Adapters

Sockets (Input Adapters)

- **IN1** (Type: adapter::types::unidirectional::AUS): The first input (minuend) for subtraction calculation.
- **IN2** (Type: adapter::types::unidirectional::AUS): The second input (subtrahend) for subtraction calculation.

Plugs (Output Adapters)

- **OUT** (Type: adapter::types::unidirectional::AUS): The output (difference) that provides the calculated result of the subtraction.

Functionality

The primary function of AUS_SUB_2_UNGATED is arithmetic subtraction:

$$\mathrm{OUT} = \mathrm{IN1} - \mathrm{IN2} \tag{2.121}$$

Since the function block is based on adapters, it receives the values and the corresponding trigger events via the sockets IN1 and IN2. As soon as relevant data arrives via the input adapters, the mathematical operation is executed. The result of the calculation and the corresponding output event are then forwarded to the subsequent function blocks via the plug OUT.

Since this is a generic function block (GEN_AUS_SUB), the mathematical processing dynamically adapts to the data types defined in the AUS adapter.

Technical Features

- **Generic Type (GEN_AUS_SUB):** The function block is not limited to a single data type (such as only INT or REAL), but supports the data types defined by the adapter structure.
- **Unidirectional Adapters:** Using the unidirectional::AUS type ensures a clear, one-way data and signal flow. This minimizes the complexity of signal tracing within the system.
- – **Encapsulation:** By eliminating traditional event and data pins, the visual layout in the 4diac-ide Application Editor remains extremely compact and clear.

State Overview

Since this module is designed as a purely functional, stateless calculation block, it does not have a complex internal Execution Control Chart (ECC). Its execution is purely data- and event-driven, based on the interactions of the connected adapters.

Application Scenarios

- **Process Value Correction:** Subtraction of zero-point offsets or calibration values from a measured sensor value.
- **Target-Actual Value Comparison:** Calculation of the control deviation ($e = w - x$) in control loops where the signals are already available as structured adapter channels.
- **Level Calculation:** Determination of differential quantities in containers or systems by subtracting the outflow from the inflow.

Comparison with Similar Function Blocks

Compared to a classic IEC 61131-3 SUB function block, which works directly with elementary data types, the AUS_SUB_2_UNGATED offers the following advantages:

- **Reduced Wiring Effort:** Event and data lines do not need to be run separately, as they are bundled in the AUS adapter.
- **Increased Modularity:** It is ideally suited for service-oriented architectures in IEC 61499, where subsystems typically communicate via adapters.

–

Change Detection

This block performs **no** change detection. Every newly computed result is written to the output and its adapter event fired unconditionally, regardless of whether the value differs from the previous run.

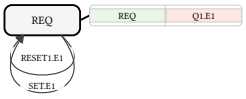
Conclusion

The AUS_SUB_2_UNGATED is a specialized yet flexible subtraction function block. Through the consistent use of unidirectional adapters, it promotes a clean, structured, and clear design of control programs in the 4diac IDE.

Numerical Index (iec61131-3)

Here you will find all exercises from the iec61131-3 workspace in ascending order.

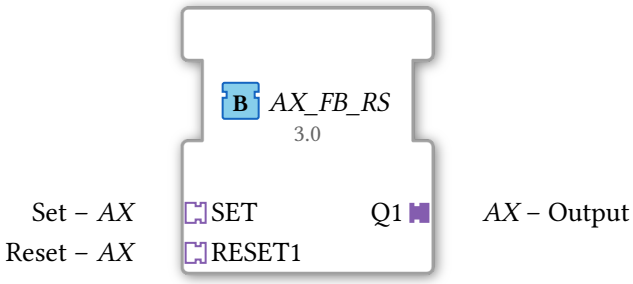
AX_FB_RS



• – ○ • – ○ • – ○ •

Introduction

The function block **AX_FB_RS** implements a bistable flip-flop with reset dominance, based on the IEC 61131-3 standard. Unlike the standard RS block, this function block uses adapter interfaces (AX) to transmit events and data in a bundled manner. This allows for a more compact representation in control diagrams, as data and event connections are combined in a single line.



Interface Structure

Since this block is based exclusively on adapters, the classic event and data inputs/outputs are empty. Communication takes place entirely via the defined plugs and sockets.

Event Inputs

No direct event inputs available.

Event Outputs

No direct event outputs available.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

This component uses the adapter type `adapter::types::unidirectional::AX`. This adapter type typically bundles a Boolean signal (`D1`) and an associated event (`E1`).

Name	Type	Direction	Description
Q1	AX	Plug (Output)	The output of the flip-flop (status).
SET	AX	Socket (Input)	The set input. When active, the output `Q1` is set to TRUE (unless `RESET1` is active).
RESET1	AX	Socket (Input)	The reset input. When active, the output `Q1` is set to FALSE. This input has priority over `SET`.

Functionality

The **AX_FB_RS** implements the logic of an RS flip-flop with reset dominance. The internal state and the output `Q1` are calculated based on the input values of the adapters `SET` and `RESET1`.

The logical function is:
Q1 := (NOT RESET1) AND (SET OR Q1)`
The behavior can be summarized as follows:

1. **Reset:** If `RESET1.D1` is TRUE, `Q1.D1` immediately becomes FALSE, regardless of the state of `SET.D1`.
2. **Set:** If `SET.D1` is TRUE and `RESET1.D1` is FALSE, `Q1.D1` becomes TRUE.
3. **Save:** If both `SET.D1` and `RESET1.D1` are FALSE, `Q1.D1` retains its previous state.

As soon as an event (`E1`) is received at one of the input adapters (`SET` or `RESET1`), the module executes the algorithm and then fires an event at the output adapter `Q1.E1`.

Technical Features

- **Adapter Technology:** By using `AX` adapters, the module reduces visual complexity in 4diac applications, as only one connecting line needs to be drawn for the event-boolean value pair.
- **Event-Driven:** The algorithm is triggered when an event arrives via the `SET` or `RESET1` adapter.
- **Reset Dominance:** This is a safety-relevant feature. In case of a conflict (both inputs TRUE), the output safely shuts down (FALSE).

State Overview

The function block is based on a simple ECC (Execution Control Chart) with a single main state.

- **State Name:** `REQ`
- **Action:** Executes the algorithm `REQ` and sends an event via `Q1.E1`.
- **Transitions:**
- The state `REQ` has self-transitions (self-loops) that are triggered by the arrival of events at the adapters `SET.E1` or `RESET1.E1`.

This means that the function block is always ready to process input signals and respond to them immediately.

Application Scenarios

- **Motor Controls (Start/Stop):** Typical application for start (Set) and stop (Reset) buttons, where the stop button takes precedence for safety reasons (emergency stop logic).
- **Alarm Memory:** An alarm is set and remains active until it is explicitly reset (acknowledged).
- **State Markers:** Storing binary states within an adapter-based control logic.
- **Modular Systems:** Used in systems that consistently rely on adapter interfaces to avoid “spaghetti code” in the graphical representation.

⚖️ Comparison with Similar Function Blocks

Feature	AX_FB_RS	AX_FB_SR	RS (Standard)

Dominance	**Reset Dominant** (Output FALSE at S=1, R=1)	Set Dominant (Output TRUE at S=1, R=1)	Reset Dominant
Interface	Adapter (AX)	Adapter (AX)	Boolean Event/Data Ports
Wiring	Minimalist (bundled)	Minimalist (bundled)	Classic (separate lines)

Conclusion

The **AX_FB_RS** is a modern, adapter-based implementation of the classic RS flip-flop. It is ideally suited for complex control applications in 4diac where clarity and signal encapsulation are paramount. Thanks to its integrated reset dominance, it is particularly well-suited for applications where safe shutdown takes precedence over shutdown.

Wiring | Minimalist (bundled) | Classic (separate lines) |

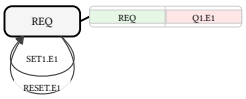
The **AX_FB_RS** is a modern, adapter-based implementation of the classic RS flip-flop. It is ideally suited for complex control applications in 4diac where clarity and signal encapsulation are essential. Its integrated reset dominance makes it especially suitable for applications where safe shutdown must take precedence over shutdown.

** _

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AX_FB_SR

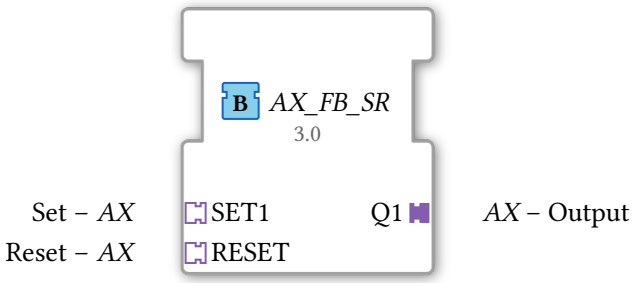


• – ○ • – ○ • – ○ •

Introduction

The **AX_FB_SR** function block implements a bistable latch with **set dominance** (SR flip-flop). Unlike conventional IEC 61131-3 function blocks, this block does not use discrete event and data connections, but rather **adapters** for signal transmission. This allows for a more compact representation and simplified wiring in complex control applications based on the adapter concept.

The block stores a binary state. If both the set (SET) and reset (RESET) signals are active simultaneously, the set signal takes precedence (set dominance).



Interface Structure

The block communicates exclusively via adapter interfaces. Traditional discrete inputs and outputs are not present, as these are encapsulated within the adapters.

Event Inputs

No direct event inputs available (these are contained in sockets `SET1` and `RESET`).

Event Outputs

No direct event outputs available (these are contained in plug `Q1`).

Data Inputs

No direct data inputs available (these are contained in sockets `SET1` and `RESET`).

Data Outputs

No direct data outputs available (these are contained in plug `Q1`).

Adapter

Name	Type	Direction	Description
Q1	`adapter::types::unidirectional::AX`	Plug (Output)	The output adapter that provides the current state (Q) and the associated event.
SET1	`adapter::types::unidirectional::AX`	Socket (Input)	The set input adapter. A `TRUE` signal here sets the output to `TRUE`.
RESET	`adapter::types::unidirectional::AX`	Socket (Input)	The reset input adapter. A `TRUE` signal here sets the output to `FALSE` (if SET1 is not active).

Note: The adapter type `AX` typically implies a combination of a Boolean data point (`D1`) and an event (`E1`).

Functionality

The internal algorithm implements the logic of an SR flip-flop with set precedence. The logic is executed as soon as an event is received at one of the adapter inputs (`SET1.E1` or `RESET.E1`).

The logical equation is (in ST syntax):

```
Q1.D1 := SET1.D1 OR ((NOT RESET.D1) AND Q1.D1);
```

Technical Features

- **Adapter-based:** The main advantage of this function block is the reduction of connection lines in the function plan, as data and events are bundled via a single adapter connection.
- **Event-driven:** The function block reacts immediately to incoming events at the `SET1` or `RESET` adapters and forwards the result immediately.
- **State machine (ECC):** The Execution Control Chart (ECC) is minimal. It consists of a single state, `REQ`, which waits for input events and executes the algorithm `REQ`.

State Overview

The following truth table describes the behavior of the function block (where `Q_n` is the old state and `Q_n+1` is the new state):

SET1.D1	RESET.D1	Q_n (Old State)	Q_n+1 (New State)	Description
0	0	0	0	Hold State
0	0	1	1	Hold State
0	1	X	0	Reset
1	0	X	1	Set
1	1	X	1	**Set (Dominant)**

Application Scenarios

- **Adapter-Based Control Systems:** Ideal for projects that consistently utilize 4diac's adapter technology to avoid the "spaghetti code" of parallel event and data lines.
- **Start/Stop Logic:** Implementing machine controls where the "on" signal (set) has priority or where continuous activation must be ensured as long as the button is pressed.
- **Alarm Handling:** Storing alarm states that must be manually acknowledged (reset).

⚖️ **Comparison with similar components**

- **SR (Standard):** The `AX_FB_SR` is logically equivalent to the standard `SR` component of IEC 61131-3, but differs in that it uses `AX` adapters instead of discrete `BOOL` and `EVENT` pins.
- **AX_FB_RS:** This is the counterpart with **reset dominance**. With the `AX_FB_RS`, if SET and RESET are applied simultaneously, the output would go to `FALSE`.
- **E_SR:** An event-driven standard SR flip-flop without adapters.

Conclusion

The **AX_FB_SR** is a specialized component for structured, adapter-based programming in 4diac. It offers the classic functionality of a set-dominant memory element, but optimizes the graphical representation and handling by encapsulating signals in adapters. It is indispensable in applications based on modular and clearly structured interface definitions.

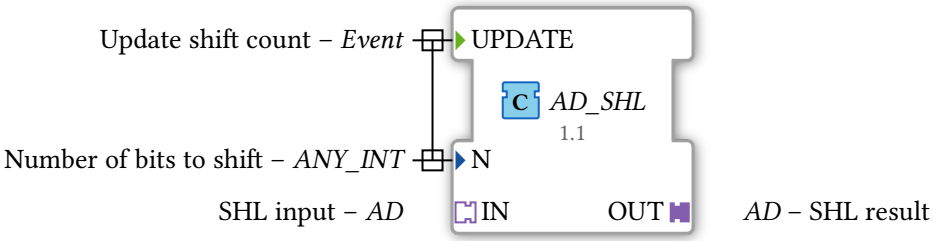
🌐 **Related topic subpages on ms-muc-docs.de**

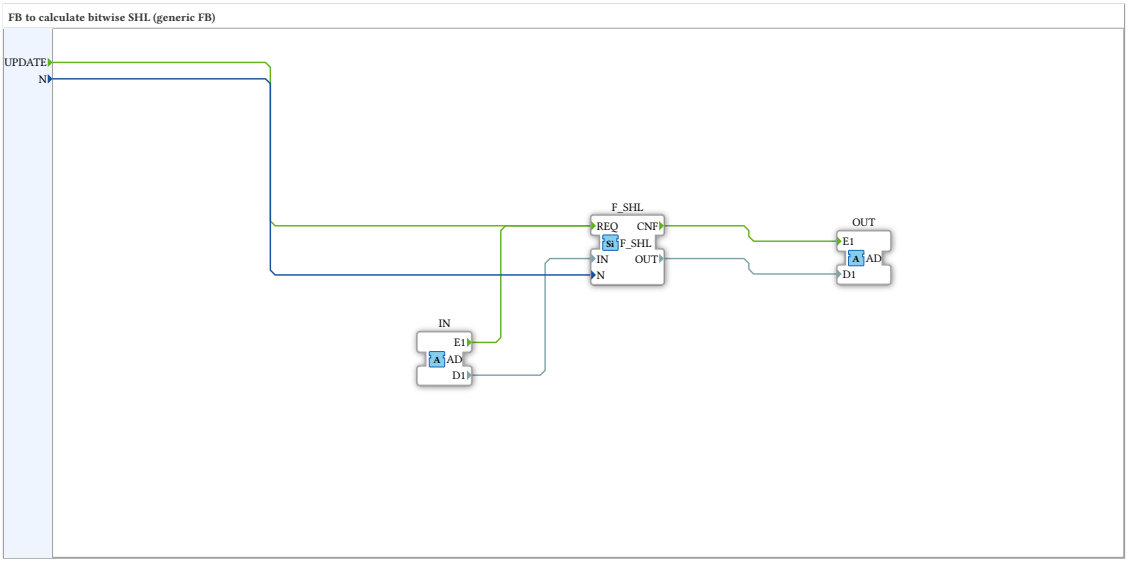
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Numeric Index (iec61131-3)

Here you will find all exercises from the `iec61131-3` workspace, listed in ascending order. ...

AD_SHL





• – ○ • – ○ • – ○ •

Introduction

The function block **AD_SHL** performs a bitwise left shift of an input value. It is designed as a generic function block that communicates via adapter interfaces and supports any integer data type (ANY_INT). The actual shift operation is implemented internally by a predefined function block `F_SHL`.

Interface Structure

Event Inputs

Event	Description	With Parameters
`UPDATE`	Updates the shift count and starts the calculation.	`N`

Event Outputs

No direct event outputs are available. The event for passing the result is handled via the adapter `OUT` (event `OUT.E1`).

Data Inputs

Name	Type	Description
`N`	`ANY_INT`	Number of bit positions to shift.

Data Outputs

No direct data outputs. The result of the shift is provided via the adapter `OUT` (data `OUT.D1`).

Adapter

Direction	Name	Type	Description
Socket	`IN`	`adapter::types::unidirectional::AD`	Receives the value to be shifted via the adapter channel.
Plug	`OUT`	`adapter::types::unidirectional::AD`	Provides the result of the left shift.

Functionality

The FB operates as a composite function block that delegates the actual operation to the block `F_SHL`. Internally, the following logic is implemented:

1. **Event Triggering**

- An incoming event from UPDATE or from the adapter IN.E1 triggers the input REQ of the internal block F_SHL.
- The value of N is directly passed to the input F_SHL.N.
- The data value provided via the adapter IN (IN.D1) is forwarded to the F_SHL.IN input.

1. **Calculation**

The internal function block F_SHL (type iec61131::bitwiseOperators::F_SHL) performs a bitwise left shift of the input value by the number of times specified in N.

1. **Output**

The result of the calculation (F_SHL.OUT) is transferred to the data output of the plug adapter OUT.D1. Simultaneously, the event F_SHL.CNF signals the availability of the new result to the plug adapter (OUT.E1).

The function block is event-driven: Each arrival of a valid event (either externally via UPDATE or from the socket adapter IN) triggers a complete calculation.

Technical Features

- **Adapter Interface**

Instead of fixed inputs and outputs, the function block uses adapters (IN as a socket, OUT as a plug), which enables flexible integration in various architectural contexts.

- **Generic Data Type**

The parameter N is declared as ANY_INT, allowing the function block to work with different integer types (e.g., INT, DINT, LINT), as long as the internal F_SHL supports them.

- **Combined Event Control**

Both the external event UPDATE and the event received from the socket adapter, IN.E1, can trigger the calculation. This enables both cyclic and data-driven updates.

- **Licensing**

This function block is licensed under the Eclipse Public License 2.0 and designed as a generic function block.

State Overview

This function block is implemented as a **composite network** and does not have its own internal state machine (ECC). The sequence control is entirely determined by the event connections within the network and the F_SHL function block used. An explicit state machine is not required.

Application Scenarios

- **Bit Manipulation in Control Engineering**

Shifting bit patterns for masking, scaling, or encoding signals.

- **Data Preparation in Communication Protocols**

Adapting data words by left shifting, e.g., for serial interfaces or bus systems.

- **Arithmetic Calculations**

Multiplication by powers of two by left shifting, especially in computationally critical environments.

Comparison with Similar Blocks

- **Direct F_SHL Block**

The internal F_SHL offers the same functionality, but without an adapter interface. AD_SHL encapsulates this block and extends it with an adapter-based connection, which facilitates reuse in modular systems.

- **Adapter-Based Arithmetic Blocks**

Similar concepts exist for other bitwise operations (e.g., AD_SHR, AD_AND). AD_SHL specifies the left shift and can be used in a library of adapter-coupled logic blocks.

Conclusion

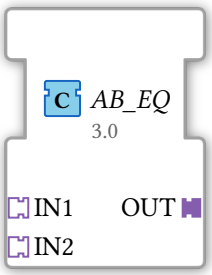
The function block `AD_SHL` implements a bitwise left shift using an encapsulated `F_SHL` block and provides all functionality via a standardized adapter interface. Its generic design and flexible event control make it suitable for a wide range of applications in IEC 61499-based automation technology. Its composite structure makes it maintainable and easily extensible.

Numeric Index (iec61131-3)

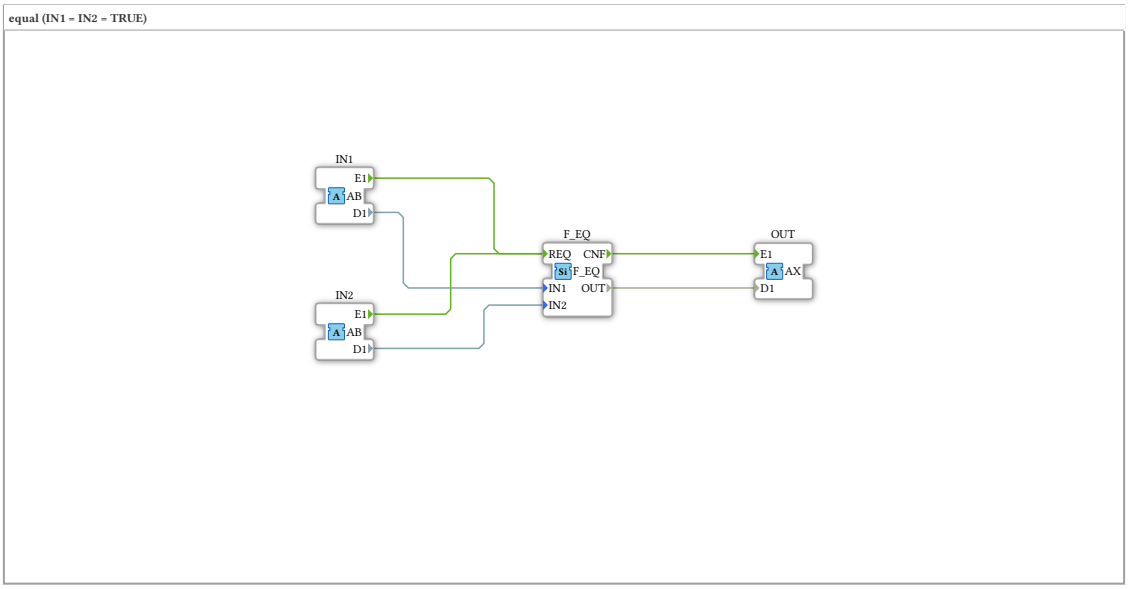
Here you will find all exercises from the `iec61131-3` workspace, listed in ascending order. ...

AB_EQ

Input value 1 -- left operand (IN1 = IN2) – *AB*
Input value 2 -- right operand (IN1 = IN2) – *AB*



AX – IN1 = IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_EQ** compares two values of type `BYTE` (8-bit bit pattern (byte)) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type BYTE, 8-bit bit pattern (byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – TRUE if IN1 = IN2, otherwise FALSE.

Adapters

Adapter	Type	Direction	Description
IN1	AB (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AB (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AB and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_EQ (type iec61131::comparison::F_EQ). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AB/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 ("equal to") for the BYTE data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status register against a reference or threshold value of type BYTE.
- **Control Logic**: Triggering an action once two values of type BYTE satisfy the "equal to" condition.
- **Data Filtering**: Selecting data records or signals that satisfy a BYTE-based comparison condition.

Comparison with Similar Function Blocks

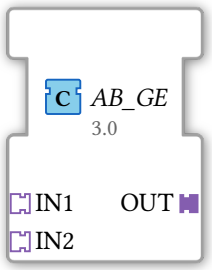
- **AB_NE**: Checks for "not equal to" – the inverse function of AB_EQ.
- **AB_NE**: Checks for inequality of values.

AB_EQ is part of the complete AB comparison family (EQ/NE/GT/GE/LT/LE) for the BYTE data type.

Conclusion

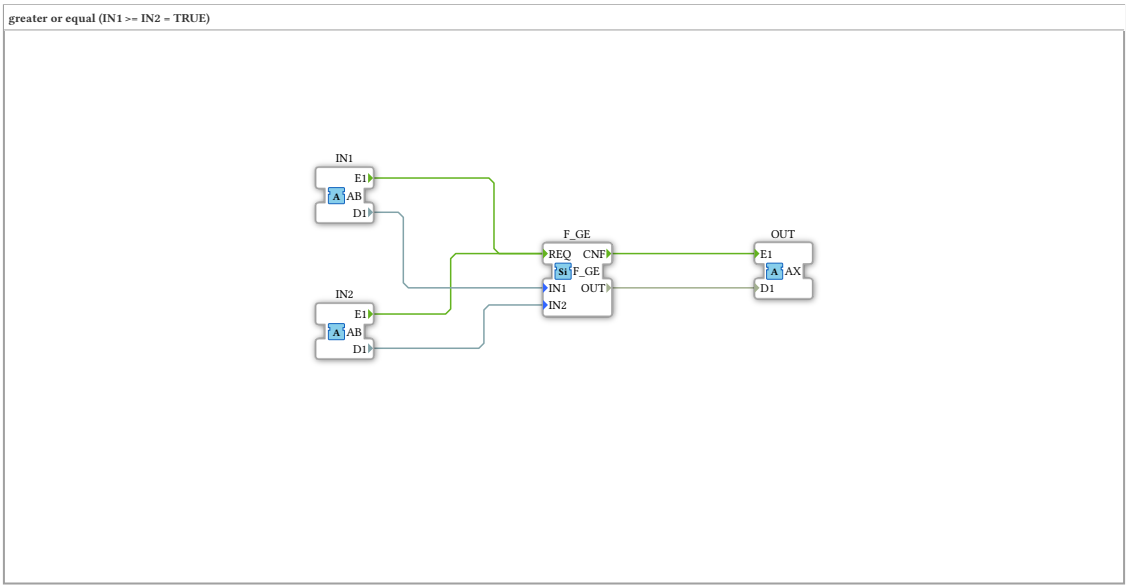
AB_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the BYTE data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AB_GE



Input value 1 -- left operand (IN1 >= IN2) – *AB*
Input value 2 -- right operand (IN1 >= IN2) – *AB*

AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_GE** compares two values of type `BYTE` (8-bit bit pattern (byte)) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BYTE`, 8-bit bit pattern (byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AB (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AB (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AB and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GE (type iec61131::comparison::F_GE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AB/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the BYTE data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status register against a reference or threshold value of type BYTE.
- **Control Logic**: Triggering an action once two values of type BYTE satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a BYTE-based comparison condition.

Comparison with Similar Function Blocks

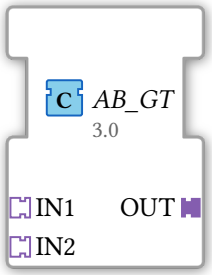
- **AB_LE**: Checks for “less than or equal to” – the inverse function of AB_GE.
- **AB_EQ**: Checks for equality of values.
- **AB_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AB_GE is the extended version.

AB_GE completes the AB comparison family with the boundary case of equality.

Conclusion

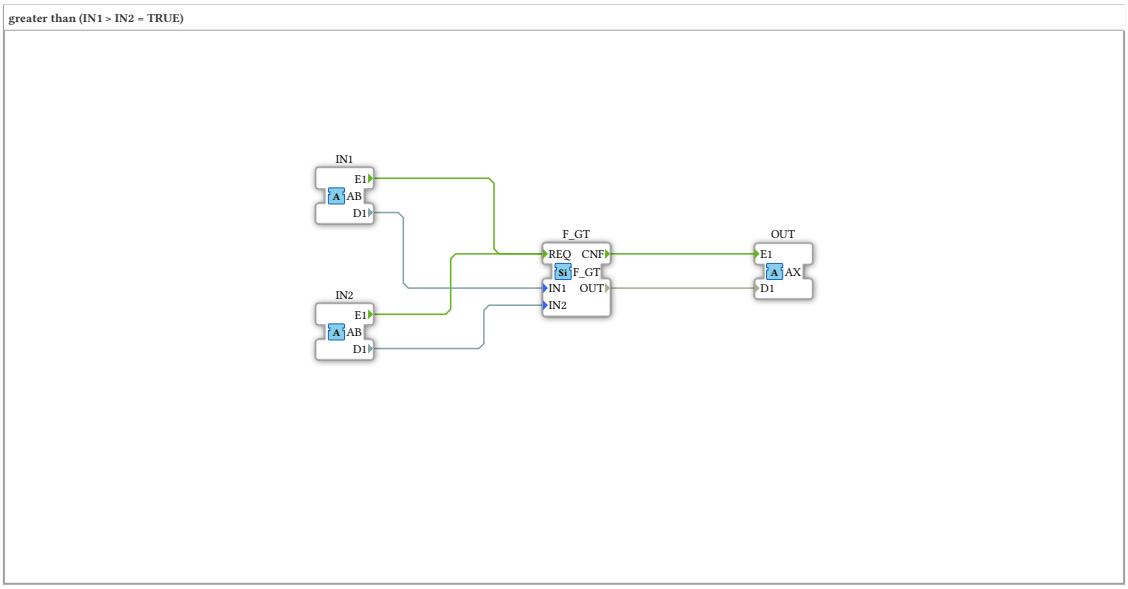
AB_GE is a simple yet important comparison block for IEC 61131-3 control applications using the BYTE data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AB_GT



Input value 1 -- left operand (IN1 > IN2) – *AB*
Input value 2 -- right operand (IN1 > IN2) – *AB*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AB_GT** compares two values of type `BYTE` (8-bit bit pattern (byte)) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BYTE`, 8-bit bit pattern (byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AB (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AB (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AB and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GT (type iec61131::comparison::F_GT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AB/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the BYTE data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status register against a reference or threshold value of type BYTE.
- **Control Logic**: Triggering an action once two values of type BYTE satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a BYTE-based comparison condition.

Comparison with Similar Function Blocks

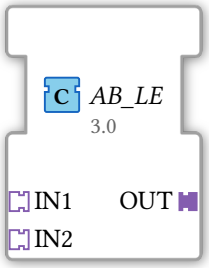
- **AB_LT**: Checks for “less than” – the inverse function of AB_GT.
- **AB_EQ**: Checks for equality of values.
- **AB_GE**: Performs the “greater than or equal to” comparison – the extended version of AB_GT that includes the equality case.

AB_GT is the basic “greater than” component without the equality case, complemented by AB_GE.

Conclusion

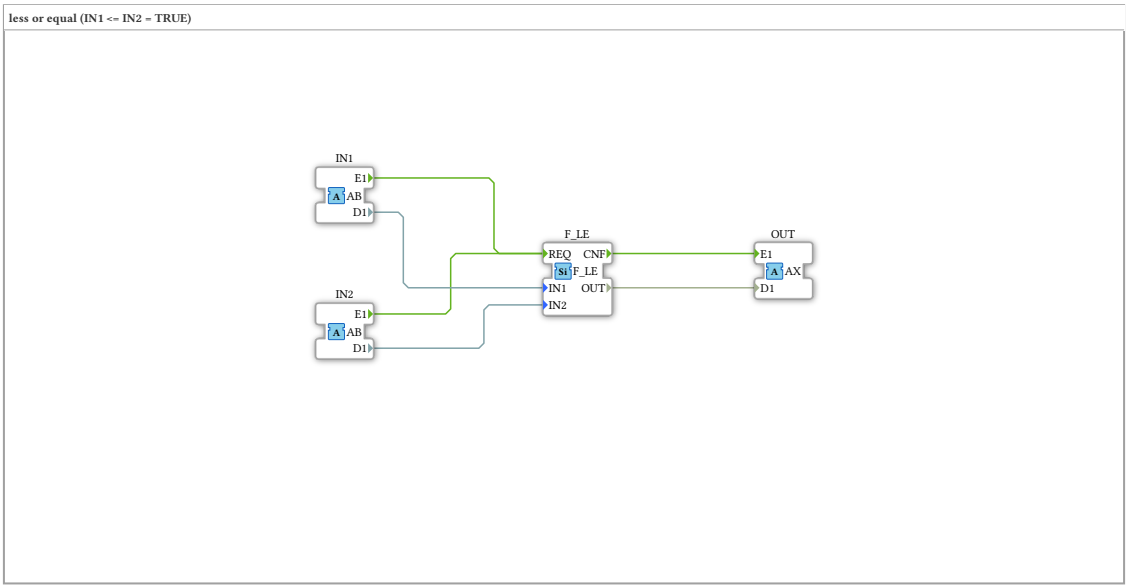
AB_GT is a simple yet important comparison block for IEC 61131-3 control applications using the BYTE data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AB_LE



Input value 1 -- left operand (IN1 <= IN2) – *AB*
Input value 2 -- right operand (IN1 <= IN2) – *AB*

AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_LE** compares two values of type `BYTE` (8-bit bit pattern (byte)) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BYTE`, 8-bit bit pattern (byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AB (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AB (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AB and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LE (type iec61131::comparison::F_LE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AB/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the BYTE data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status register against a reference or threshold value of type BYTE.
- **Control Logic**: Triggering an action once two values of type BYTE satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a BYTE-based comparison condition.

Comparison with Similar Function Blocks

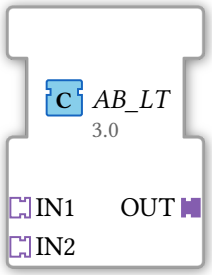
- **AB_GE**: Checks for “greater than or equal to” – the inverse function of AB_LE.
- **AB_EQ**: Checks for equality of values.
- **AB_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AB_LE is the extended version.

AB_LE completes the AB comparison family with the boundary case of equality.

Conclusion

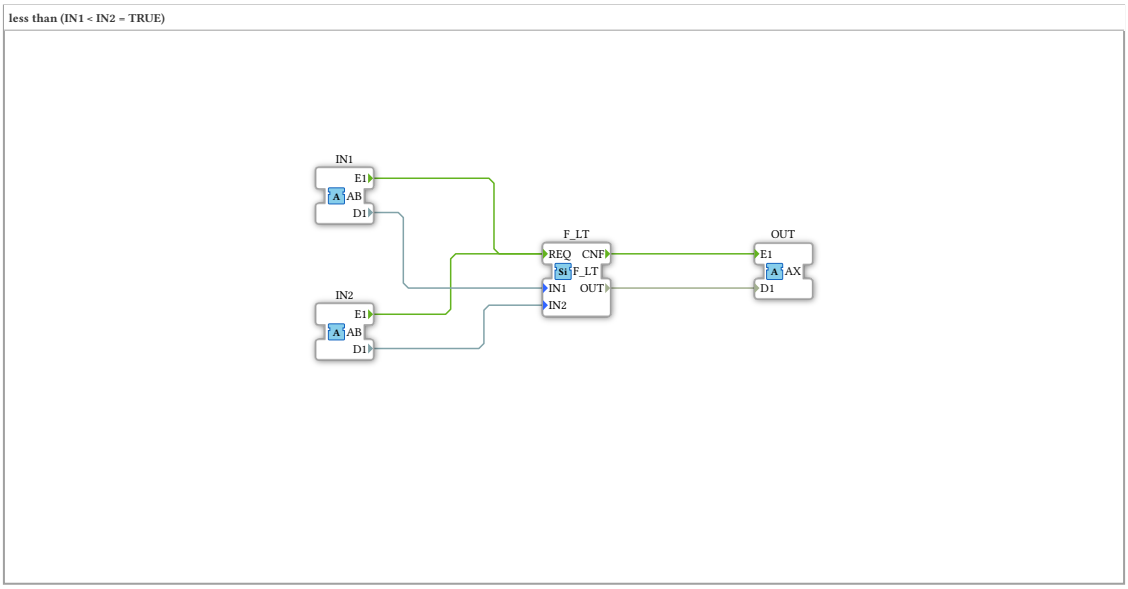
AB_LE is a simple yet important comparison block for IEC 61131-3 control applications using the BYTE data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AB_LT



Input value 1 -- left operand (IN1 < IN2) – *AB*
Input value 2 -- right operand (IN1 < IN2) – *AB*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AB_LT** compares two values of type `BYTE` (8-bit bit pattern (byte)) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BYTE`, 8-bit bit pattern (byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AB (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AB (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AB and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LT (type iec61131::comparison::F_LT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AB/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the BYTE data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status register against a reference or threshold value of type BYTE.
- **Control Logic**: Triggering an action once two values of type BYTE satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a BYTE-based comparison condition.

Comparison with Similar Function Blocks

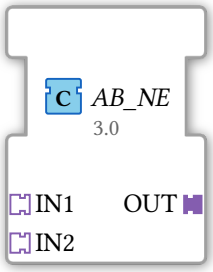
- **AB_GT**: Checks for “greater than” – the inverse function of AB_LT.
- **AB_EQ**: Checks for equality of values.
- **AB_LE**: Performs the “less than or equal to” comparison – the extended version of AB_LT that includes the equality case.

AB_LT is the basic “less than” component without the equality case, complemented by AB_LE.

Conclusion

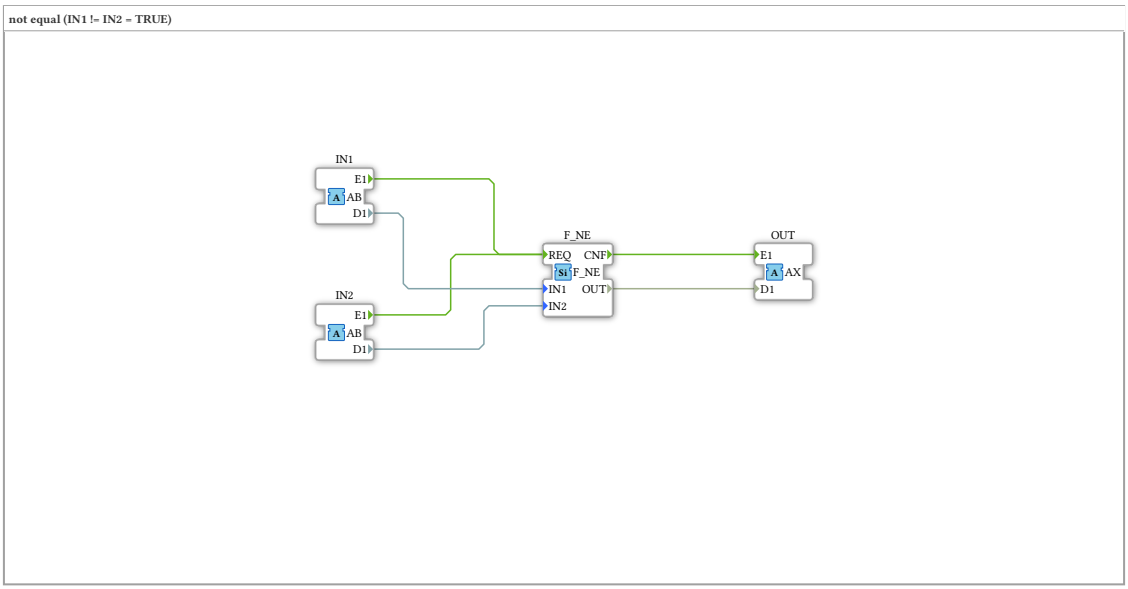
AB_LT is a simple yet important comparison block for IEC 61131-3 control applications using the BYTE data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AB_NE



Input value 1 -- left operand (IN1 != IN2) – *AB*
Input value 2 -- right operand (IN1 != IN2) – *AB*

AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_NE** compares two values of type `BYTE` (8-bit bit pattern (byte)) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BYTE`, 8-bit bit pattern (byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AB (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AB (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AB and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_NE (type iec61131::comparison::F_NE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AB/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the BYTE data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status register against a reference or threshold value of type BYTE.
- **Control Logic**: Triggering an action once two values of type BYTE satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a BYTE-based comparison condition.

Comparison with Similar Function Blocks

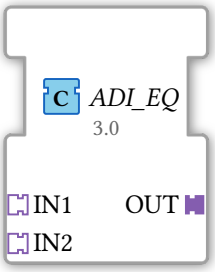
- **AB_EQ**: Checks for “equal to” – the inverse function of AB_NE.
- **AB_EQ**: Checks for equality of values.

AB_NE is part of the complete AB comparison family (EQ/NE/GT/GE/LT/LE) for the BYTE data type.

Conclusion

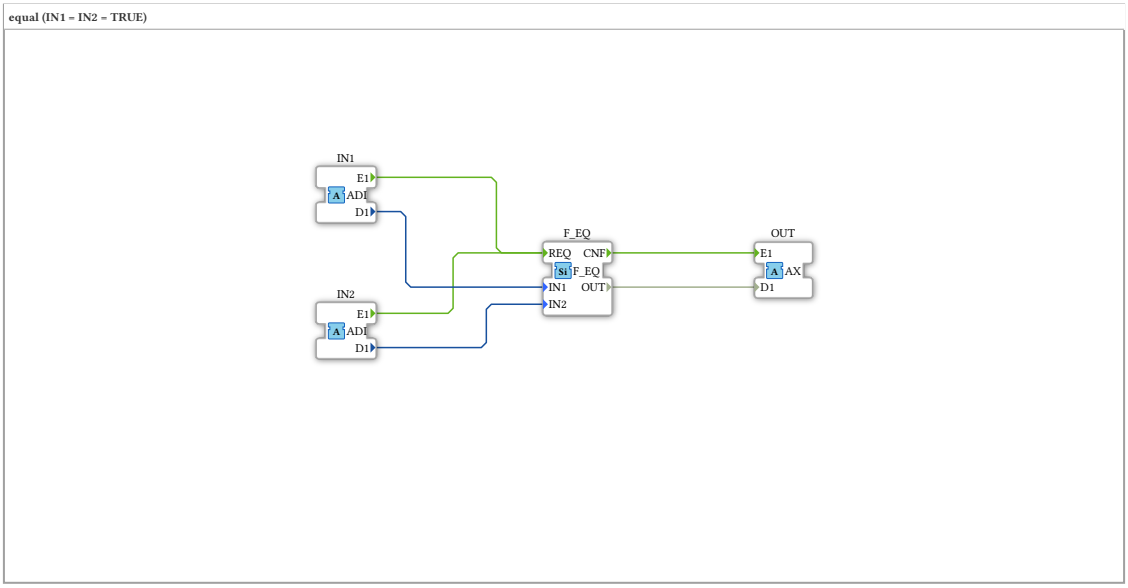
AB_NE is a simple yet important comparison block for IEC 61131-3 control applications using the BYTE data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ADI_EQ



Input value 1 -- left operand (IN1 = IN2) – *ADI*
Input value 2 -- right operand (IN1 = IN2) – *ADI*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_EQ** compares two values of type `DINT` (32-bit signed integer) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DINT`, 32-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ADI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ADI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ADI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_EQ` (type `iec61131::comparison::F_EQ`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ADI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the `DINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter and position values against a reference or threshold value of type `DINT`.
- **Control Logic**: Triggering an action once two values of type `DINT` satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `DINT`-based comparison condition.

Comparison with Similar Function Blocks

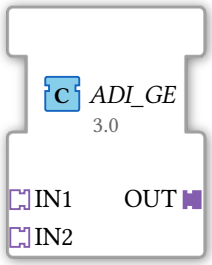
- **ADI_NE**: Checks for “not equal to” – the inverse function of ADI_EQ.
- **ADI_NE**: Checks for inequality of values.

ADI_EQ is part of the complete ADI comparison family (EQ/NE/GT/GE/LT/LE) for the `DINT` data type.

Conclusion

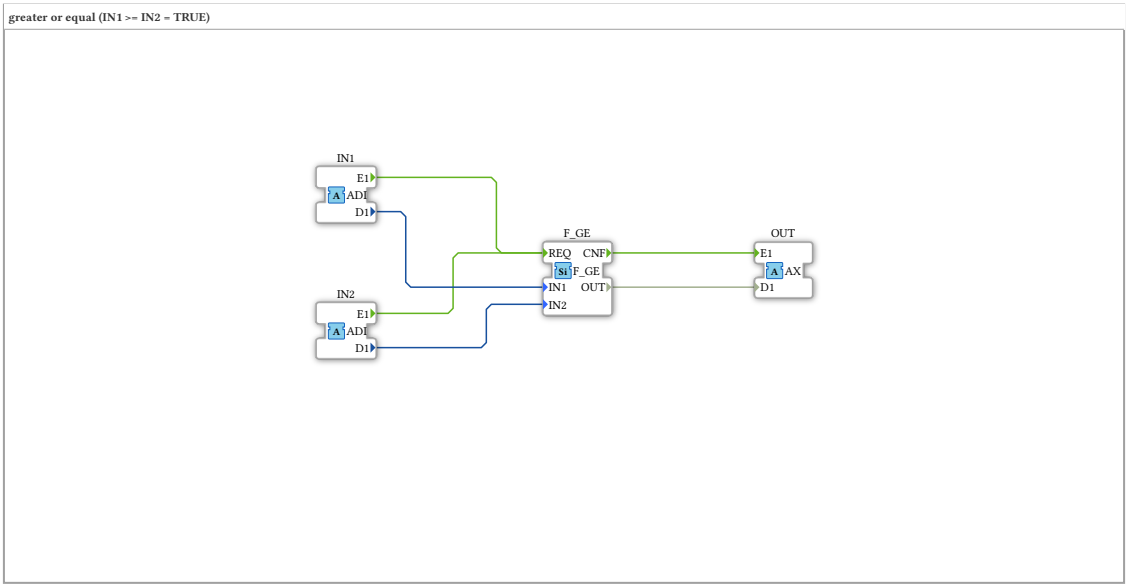
ADI_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the `DINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ADI_GE



Input value 1 -- left operand (IN1 >= IN2) – *ADI*
Input value 2 -- right operand (IN1 >= IN2) – *ADI*

AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_GE** compares two values of type `DINT` (32-bit signed integer) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DINT`, 32-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ADI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ADI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ADI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GE` (type `iec61131::comparison::F_GE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ADI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the `DINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter and position values against a reference or threshold value of type `DINT`.
- **Control Logic**: Triggering an action once two values of type `DINT` satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `DINT`-based comparison condition.

Comparison with Similar Function Blocks

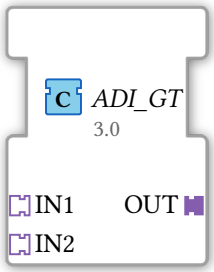
- **ADI_LE**: Checks for “less than or equal to” – the inverse function of ADI_GE.
- **ADI_EQ**: Checks for equality of values.
- **ADI_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which ADI_GE is the extended version.

ADI_GE completes the ADI comparison family with the boundary case of equality.

Conclusion

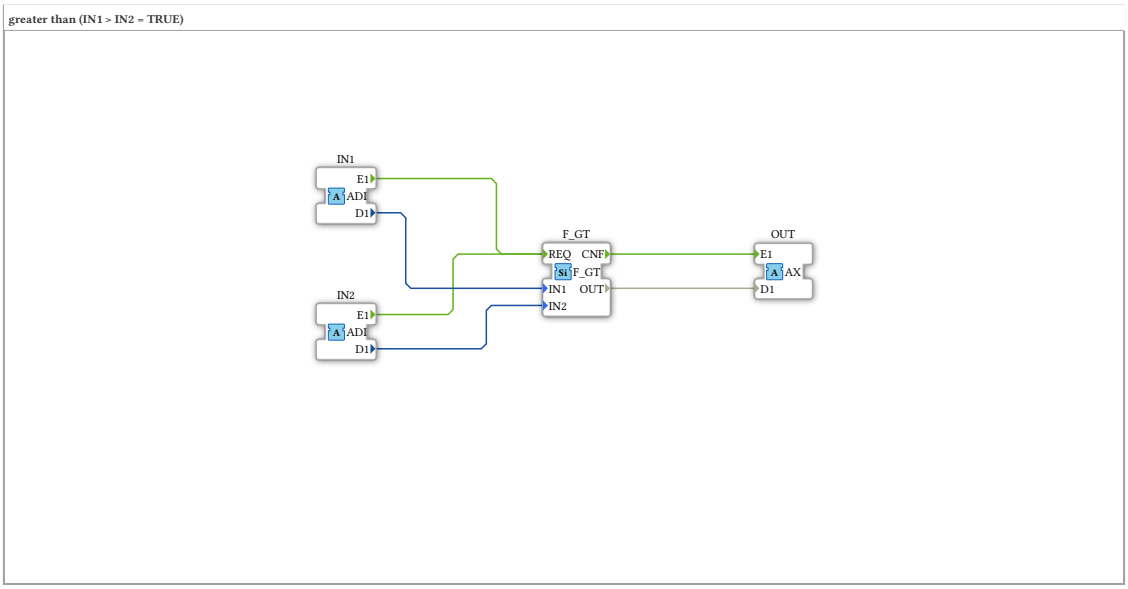
ADI_GE is a simple yet important comparison block for IEC 61131-3 control applications using the `DINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ADI_GT



Input value 1 -- left operand (IN1 > IN2) – *ADI*
Input value 2 -- right operand (IN1 > IN2) – *ADI*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_GT** compares two values of type `DINT` (32-bit signed integer) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DINT`, 32-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ADI (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ADI (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ADI and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GT (type iec61131::comparison::F_GT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ADI/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the DINT data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter and position values against a reference or threshold value of type DINT.
- **Control Logic**: Triggering an action once two values of type DINT satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a DINT-based comparison condition.

Comparison with Similar Function Blocks

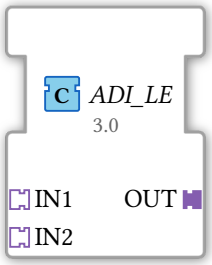
- **ADI_LT**: Checks for “less than” – the inverse function of ADI_GT.
- **ADI_EQ**: Checks for equality of values.
- **ADI_GE**: Performs the “greater than or equal to” comparison – the extended version of ADI_GT that includes the equality case.

ADI_GT is the basic “greater than” component without the equality case, complemented by ADI_GE.

Conclusion

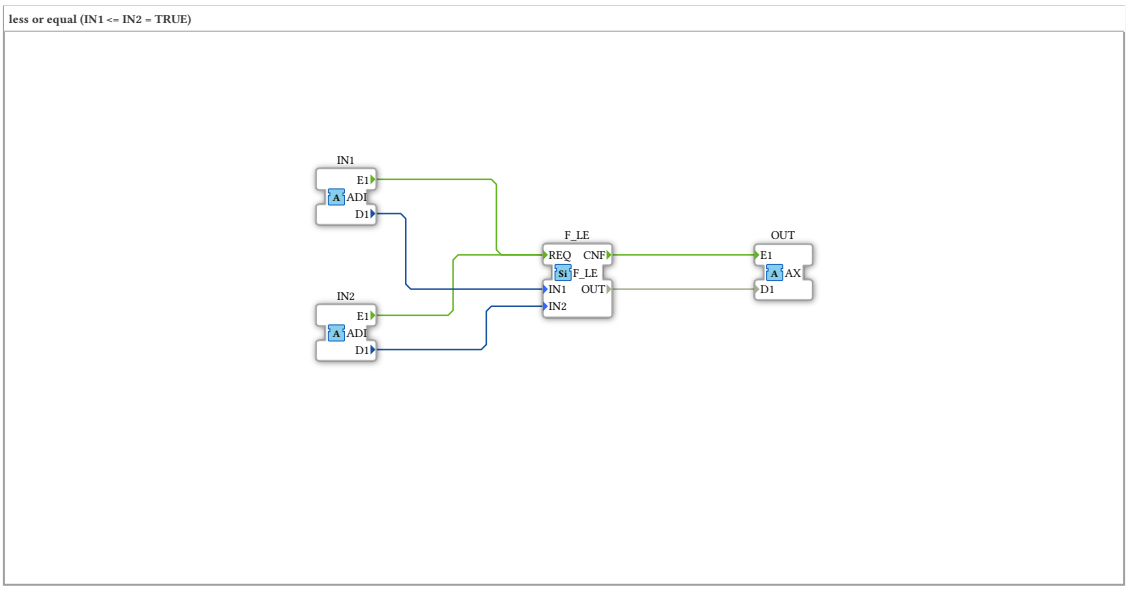
ADI_GT is a simple yet important comparison block for IEC 61131-3 control applications using the DINT data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ADI_LE



Input value 1 -- left operand (IN1 <= IN2) – *ADI*
Input value 2 -- right operand (IN1 <= IN2) – *ADI*

AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_LE** compares two values of type `DINT` (32-bit signed integer) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DINT`, 32-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ADI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ADI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ADI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LE` (type `iec61131::comparison::F_LE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ADI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the `DINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter and position values against a reference or threshold value of type `DINT`.
- **Control Logic**: Triggering an action once two values of type `DINT` satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `DINT`-based comparison condition.

Comparison with Similar Function Blocks

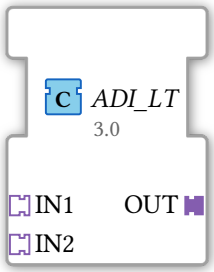
- **ADI_GE**: Checks for “greater than or equal to” – the inverse function of ADI_LE.
- **ADI_EQ**: Checks for equality of values.
- **ADI_LT**: Performs the “less than” comparison – the base variant without the equality case, of which ADI_LE is the extended version.

ADI_LE completes the ADI comparison family with the boundary case of equality.

Conclusion

ADI_LE is a simple yet important comparison block for IEC 61131-3 control applications using the `DINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

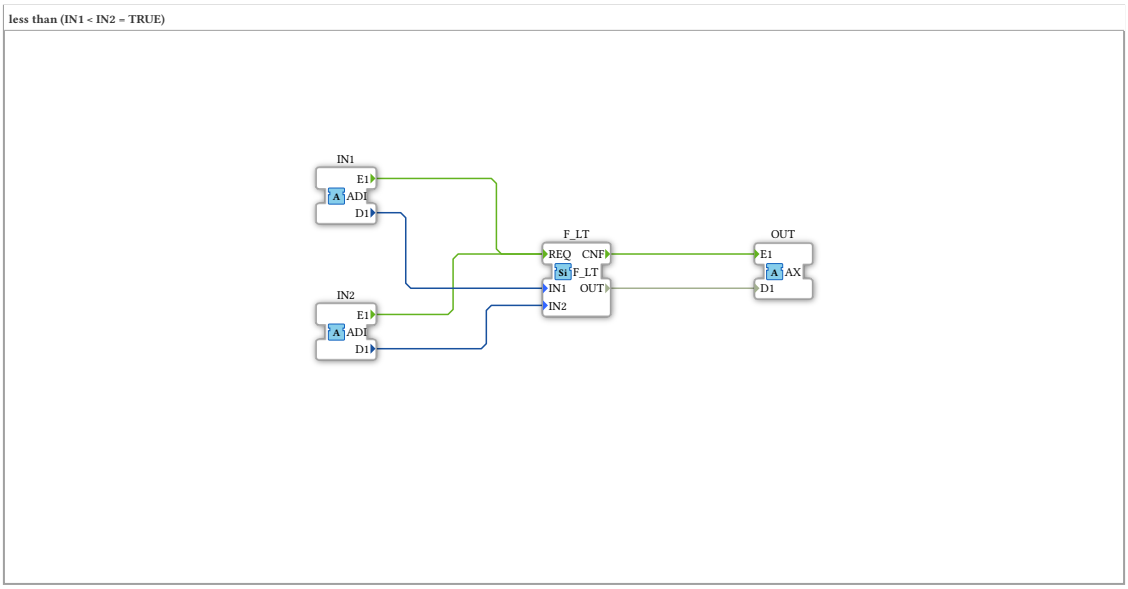
ADI_LT



Input value 1 -- left operand (IN1 < IN2) – *ADI*

Input value 2 -- right operand (IN1 < IN2) – *ADI*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_LT** compares two values of type `DINT` (32-bit signed integer) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DINT`, 32-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ADI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ADI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ADI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LT` (type `iec61131::comparison::F_LT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ADI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the `DINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter and position values against a reference or threshold value of type `DINT`.
- **Control Logic**: Triggering an action once two values of type `DINT` satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `DINT`-based comparison condition.

Comparison with Similar Function Blocks

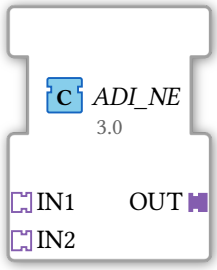
- **ADI_GT**: Checks for “greater than” – the inverse function of ADI_LT.
- **ADI_EQ**: Checks for equality of values.
- **ADI_LE**: Performs the “less than or equal to” comparison – the extended version of ADI_LT that includes the equality case.

ADI_LT is the basic “less than” component without the equality case, complemented by ADI_LE.

Conclusion

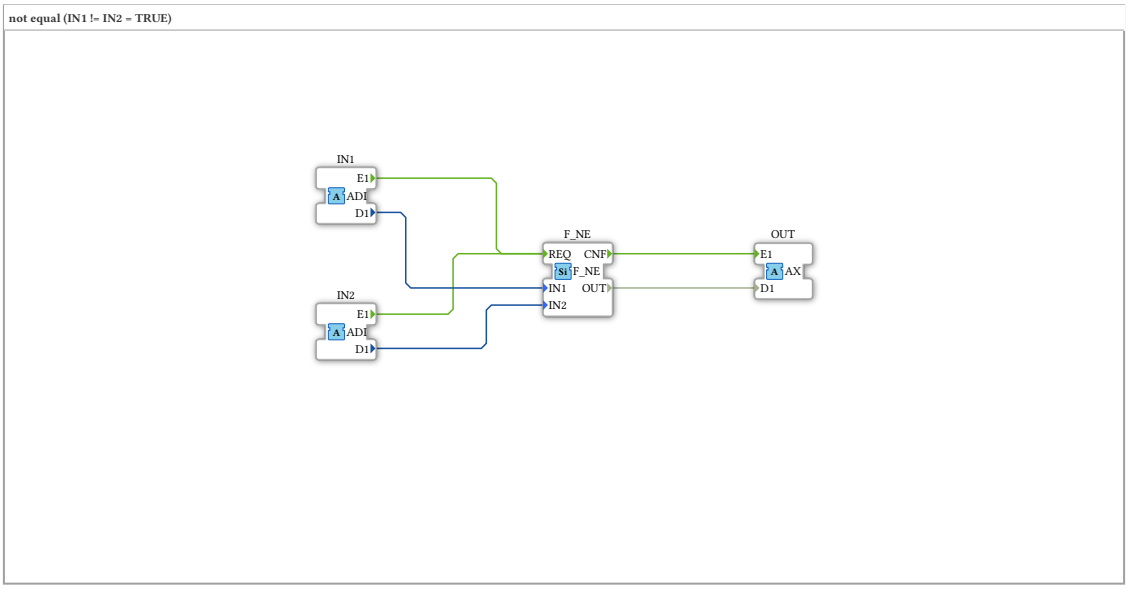
ADI_LT is a simple yet important comparison block for IEC 61131-3 control applications using the `DINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ADI_NE



Input value 1 -- left operand (IN1 != IN2) – *ADI*
Input value 2 -- right operand (IN1 != IN2) – *ADI*

AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ADI_NE** compares two values of type `DINT` (32-bit signed integer) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DINT`, 32-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ADI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ADI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ADI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_NE` (type `iec61131::comparison::F_NE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ADI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the `DINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter and position values against a reference or threshold value of type `DINT`.
- **Control Logic**: Triggering an action once two values of type `DINT` satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `DINT`-based comparison condition.

Comparison with Similar Function Blocks

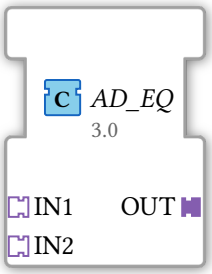
- **ADI_EQ**: Checks for “equal to” – the inverse function of ADI_NE.
- **ADI_EQ**: Checks for equality of values.

ADI_NE is part of the complete ADI comparison family (EQ/NE/GT/GE/LT/LE) for the `DINT` data type.

Conclusion

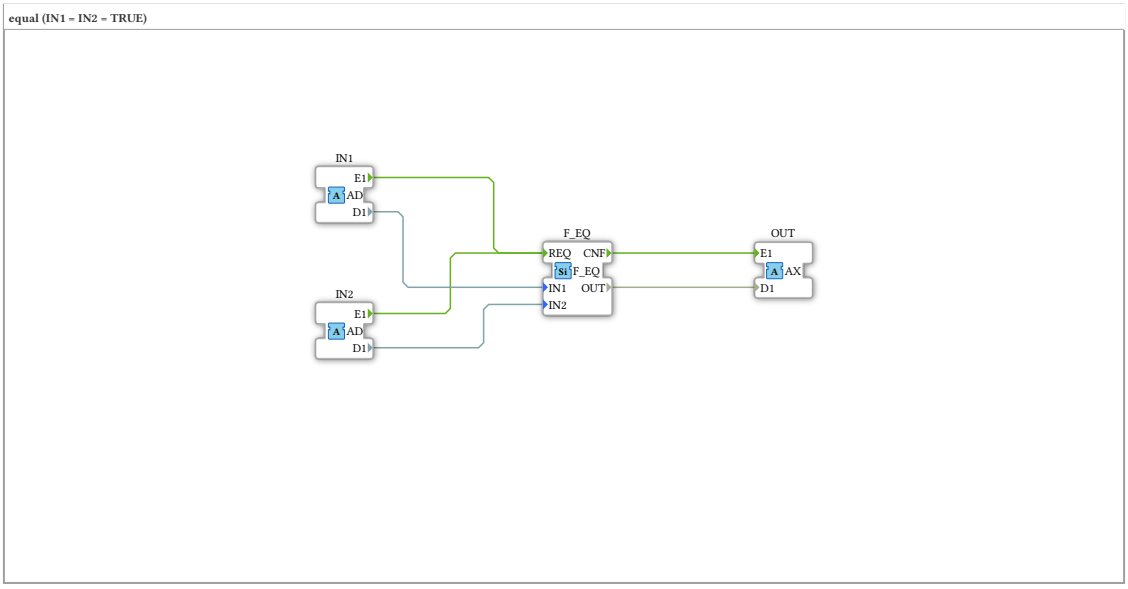
ADI_NE is a simple yet important comparison block for IEC 61131-3 control applications using the `DINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AD_EQ



Input value 1 -- left operand (IN1 = IN2) – *AD*
Input value 2 -- right operand (IN1 = IN2) – *AD*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AD_EQ** compares two values of type `DWORD` (32-bit bit pattern (double word)) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DWORD`, 32-bit bit pattern (double word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AD (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AD (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AD and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_EQ (type iec61131::comparison::F_EQ). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AD/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the DWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking bit masks against a reference or threshold value of type DWORD.
- **Control Logic**: Triggering an action once two values of type DWORD satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a DWORD-based comparison condition.

Comparison with Similar Function Blocks

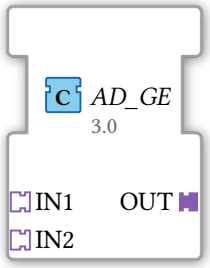
- **AD_NE**: Checks for “not equal to” – the inverse function of AD_EQ.
- **AD_NE**: Checks for inequality of values.

AD_EQ is part of the complete AD comparison family (EQ/NE/GT/GE/LT/LE) for the DWORD data type.

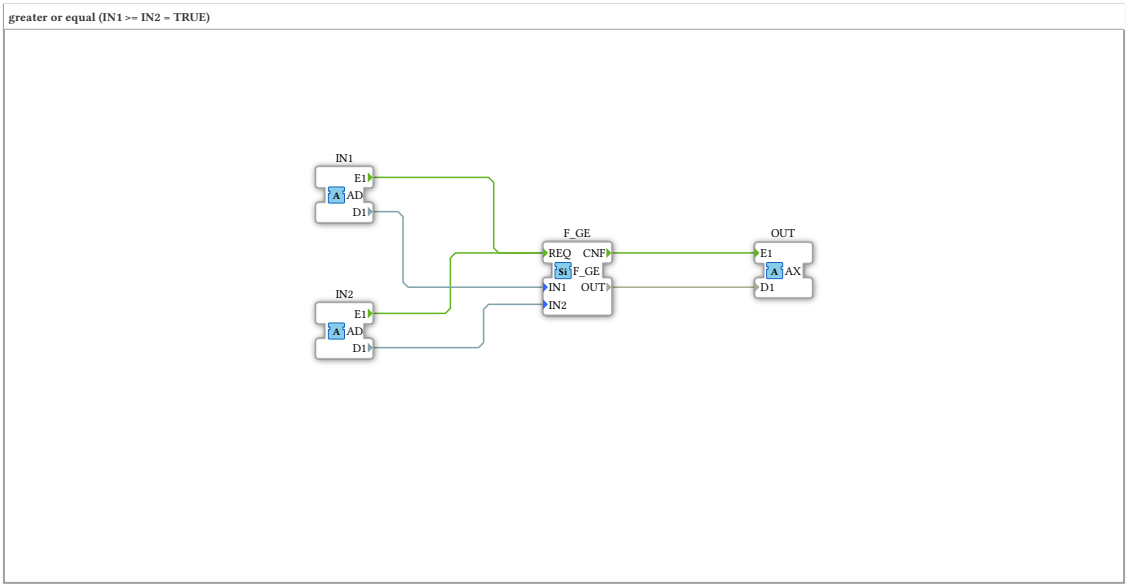
Conclusion

AD_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the DWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AD_GE



Input value 1 -- left operand (IN1 >= IN2) – *AD*
Input value 2 -- right operand (IN1 >= IN2) – *AD*
AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_GE** compares two values of type `DWORD` (32-bit bit pattern (double word)) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DWORD`, 32-bit bit pattern (double word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AD (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AD (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AD and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GE (type iec61131::comparison::F_GE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AD/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the DWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking bit masks against a reference or threshold value of type DWORD.
- **Control Logic**: Triggering an action once two values of type DWORD satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a DWORD-based comparison condition.

Comparison with Similar Function Blocks

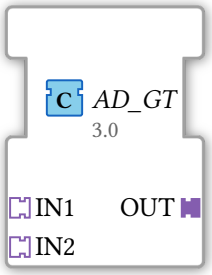
- **AD_LE**: Checks for “less than or equal to” – the inverse function of AD_GE.
- **AD_EQ**: Checks for equality of values.
- **AD_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AD_GE is the extended version.

AD_GE completes the AD comparison family with the boundary case of equality.

Conclusion

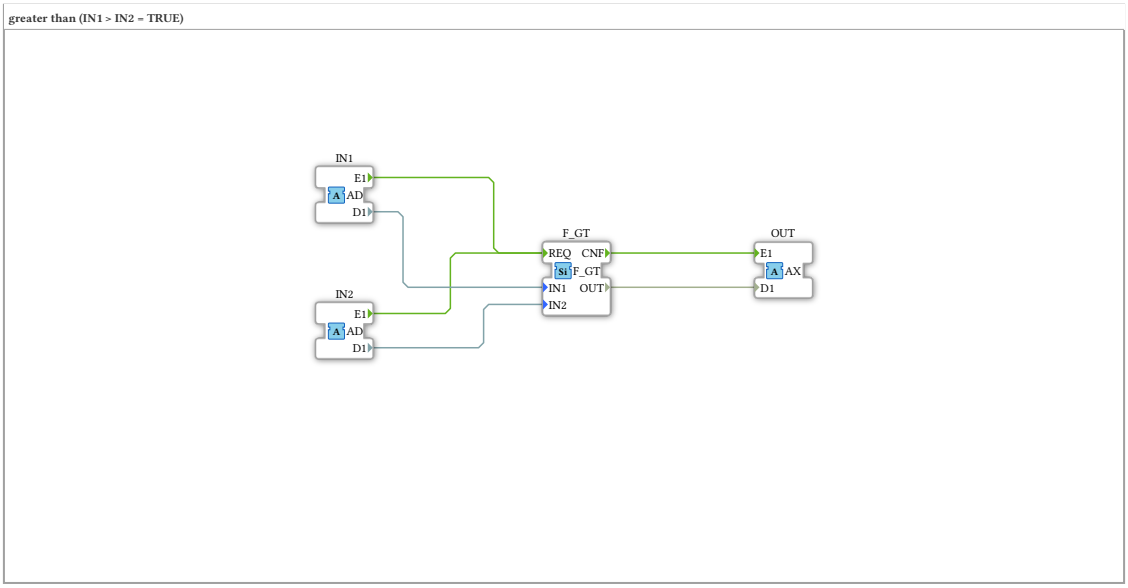
AD_GE is a simple yet important comparison block for IEC 61131-3 control applications using the DWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AD_GT



Input value 1 -- left operand (IN1 > IN2) – *AD*
Input value 2 -- right operand (IN1 > IN2) – *AD*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AD_GT** compares two values of type `DWORD` (32-bit bit pattern (double word)) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DWORD`, 32-bit bit pattern (double word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AD (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AD (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AD and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GT (type iec61131::comparison::F_GT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AD/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the DWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking bit masks against a reference or threshold value of type DWORD.
- **Control Logic**: Triggering an action once two values of type DWORD satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a DWORD-based comparison condition.

Comparison with Similar Function Blocks

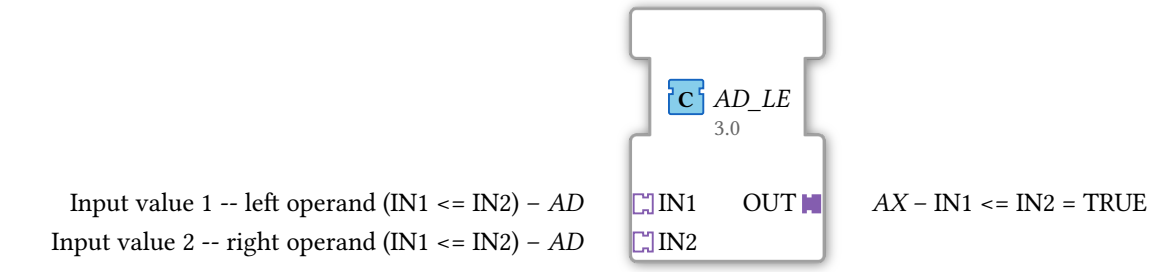
- **AD_LT**: Checks for “less than” – the inverse function of AD_GT.
- **AD_EQ**: Checks for equality of values.
- **AD_GE**: Performs the “greater than or equal to” comparison – the extended version of AD_GT that includes the equality case.

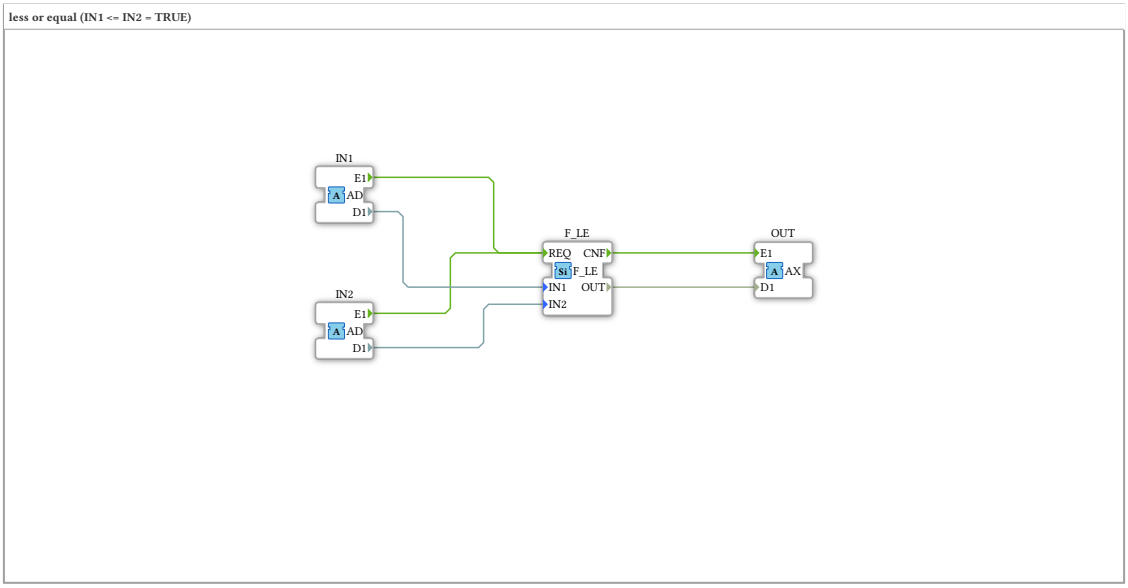
AD_GT is the basic “greater than” component without the equality case, complemented by AD_GE.

Conclusion

AD_GT is a simple yet important comparison block for IEC 61131-3 control applications using the DWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AD_LE





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_LE** compares two values of type `DWORD` (32-bit bit pattern (double word)) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DWORD`, 32-bit bit pattern (double word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AD (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AD (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AD and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LE (type iec61131::comparison::F_LE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AD/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the DWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking bit masks against a reference or threshold value of type DWORD.
- **Control Logic**: Triggering an action once two values of type DWORD satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a DWORD-based comparison condition.

Comparison with Similar Function Blocks

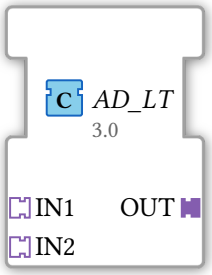
- **AD_GE**: Checks for “greater than or equal to” – the inverse function of AD_LE.
- **AD_EQ**: Checks for equality of values.
- **AD_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AD_LE is the extended version.

AD_LE completes the AD comparison family with the boundary case of equality.

Conclusion

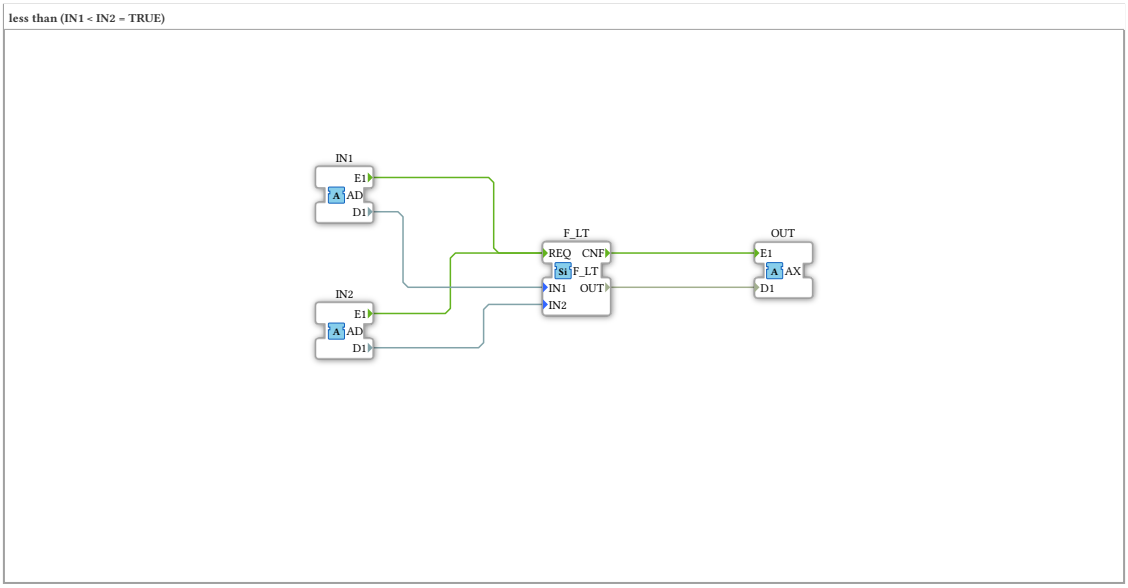
AD_LE is a simple yet important comparison block for IEC 61131-3 control applications using the DWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AD_LT



Input value 1 -- left operand (IN1 < IN2) – *AD*
Input value 2 -- right operand (IN1 < IN2) – *AD*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AD_LT** compares two values of type `DWORD` (32-bit bit pattern (double word)) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DWORD`, 32-bit bit pattern (double word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AD (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AD (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AD and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LT (type iec61131::comparison::F_LT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AD/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the DWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking bit masks against a reference or threshold value of type DWORD.
- **Control Logic**: Triggering an action once two values of type DWORD satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a DWORD-based comparison condition.

Comparison with Similar Function Blocks

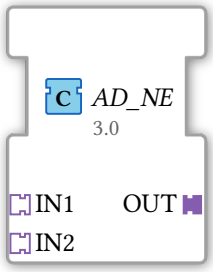
- **AD_GT**: Checks for “greater than” – the inverse function of AD_LT.
- **AD_EQ**: Checks for equality of values.
- **AD_LE**: Performs the “less than or equal to” comparison – the extended version of AD_LT that includes the equality case.

AD_LT is the basic “less than” component without the equality case, complemented by AD_LE.

Conclusion

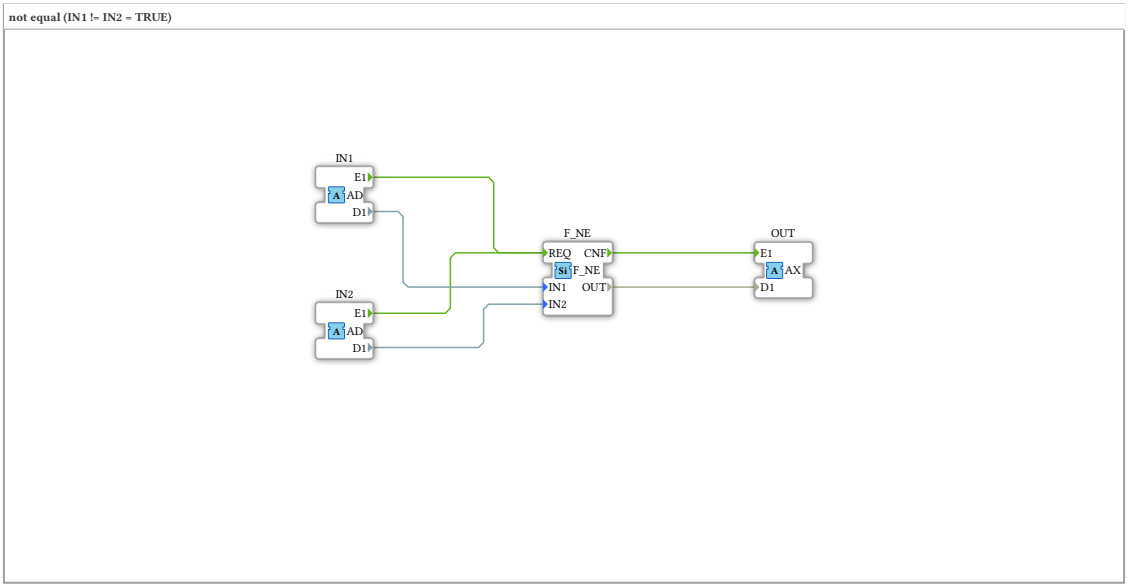
AD_LT is a simple yet important comparison block for IEC 61131-3 control applications using the DWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AD_NE



Input value 1 -- left operand (IN1 != IN2) – *AD*
Input value 2 -- right operand (IN1 != IN2) – *AD*

AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AD_NE** compares two values of type `DWORD` (32-bit bit pattern (double word)) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `DWORD`, 32-bit bit pattern (double word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AD (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AD (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AD and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_NE (type iec61131::comparison::F_NE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AD/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the DWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking bit masks against a reference or threshold value of type DWORD.
- **Control Logic**: Triggering an action once two values of type DWORD satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a DWORD-based comparison condition.

Comparison with Similar Function Blocks

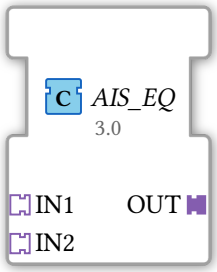
- **AD_EQ**: Checks for “equal to” – the inverse function of AD_NE.
- **AD_EQ**: Checks for equality of values.

AD_NE is part of the complete AD comparison family (EQ/NE/GT/GE/LT/LE) for the DWORD data type.

Conclusion

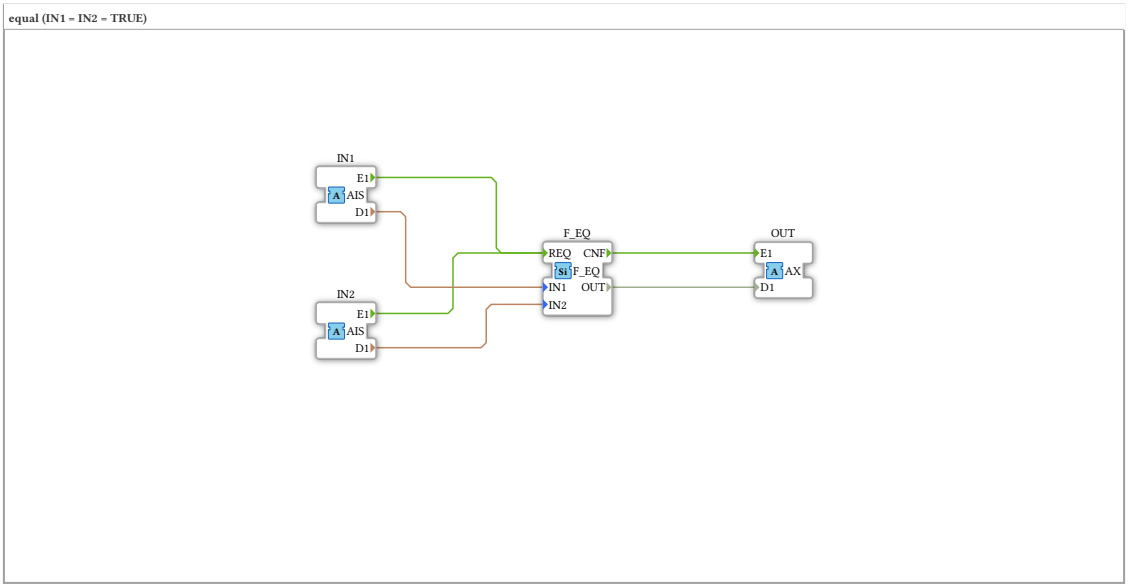
AD_NE is a simple yet important comparison block for IEC 61131-3 control applications using the DWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIS_EQ



Input value 1 -- left operand (IN1 = IN2) – *AIS*
Input value 2 -- right operand (IN1 = IN2) – *AIS*

AX – IN1 = IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_EQ** compares two values of type `STRING` (character string) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `STRING`, character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AIS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AIS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AIS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_EQ` (type `iec61131::comparison::F_EQ`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AIS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the `STRING` data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking recipe and text identifiers against a reference or threshold value of type `STRING`.
- **Control Logic**: Triggering an action once two values of type `STRING` satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `STRING`-based comparison condition.

Comparison with Similar Function Blocks

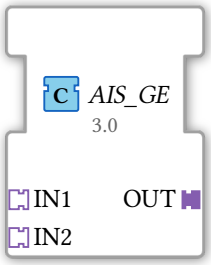
- **AIS_NE**: Checks for “not equal to” – the inverse function of AIS_EQ.
- **AIS_NE**: Checks for inequality of values.

AIS_EQ is part of the complete AIS comparison family (EQ/NE/GT/GE/LT/LE) for the `STRING` data type.

Conclusion

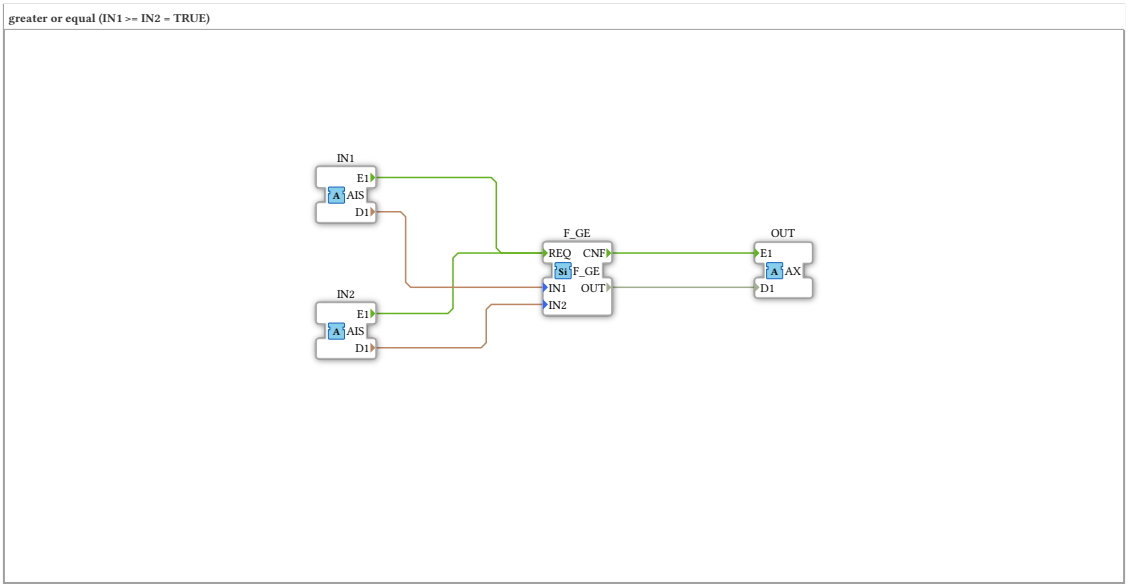
AIS_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the `STRING` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIS_GE



Input value 1 -- left operand (IN1 >= IN2) – *AIS*
Input value 2 -- right operand (IN1 >= IN2) – *AIS*

AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AIS_GE** compares two values of type `STRING` (character string) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `STRING`, character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AIS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AIS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AIS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GE` (type `iec61131::comparison::F_GE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AIS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the `STRING` data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking recipe and text identifiers against a reference or threshold value of type `STRING`.
- **Control Logic**: Triggering an action once two values of type `STRING` satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `STRING`-based comparison condition.

Comparison with Similar Function Blocks

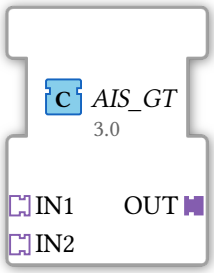
- **AIS_LE**: Checks for “less than or equal to” – the inverse function of AIS_GE.
- **AIS_EQ**: Checks for equality of values.
- **AIS_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AIS_GE is the extended version.

AIS_GE completes the AIS comparison family with the boundary case of equality.

Conclusion

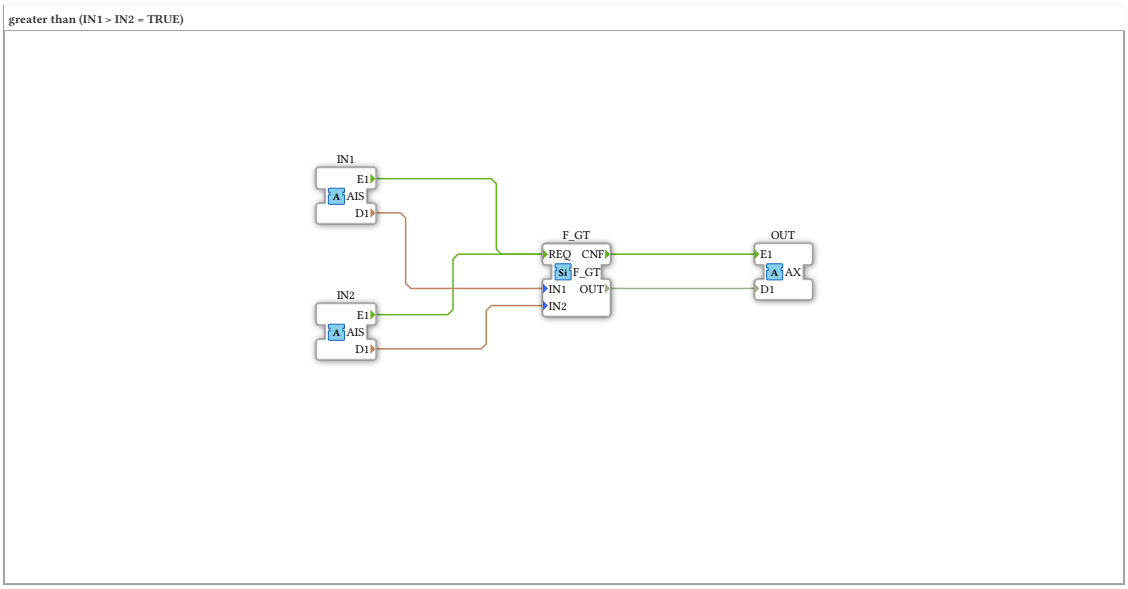
AIS_GE is a simple yet important comparison block for IEC 61131-3 control applications using the `STRING` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIS_GT



Input value 1 -- left operand (IN1 > IN2) – *AIS*
Input value 2 -- right operand (IN1 > IN2) – *AIS*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AIS_GT** compares two values of type `STRING` (character string) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `STRING`, character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AIS (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AIS (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AIS and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GT (type iec61131::comparison::F_GT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AIS/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 ("greater than") for the STRING data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking recipe and text identifiers against a reference or threshold value of type STRING.
- **Control Logic**: Triggering an action once two values of type STRING satisfy the "greater than" condition.
- **Data Filtering**: Selecting data records or signals that satisfy a STRING-based comparison condition.

Comparison with Similar Function Blocks

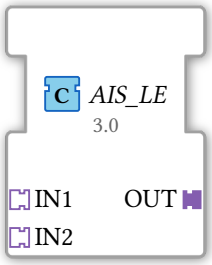
- **AIS_LT**: Checks for "less than" – the inverse function of AIS_GT.
- **AIS_EQ**: Checks for equality of values.
- **AIS_GE**: Performs the "greater than or equal to" comparison – the extended version of AIS_GT that includes the equality case.

AIS_GT is the basic "greater than" component without the equality case, complemented by AIS_GE.

Conclusion

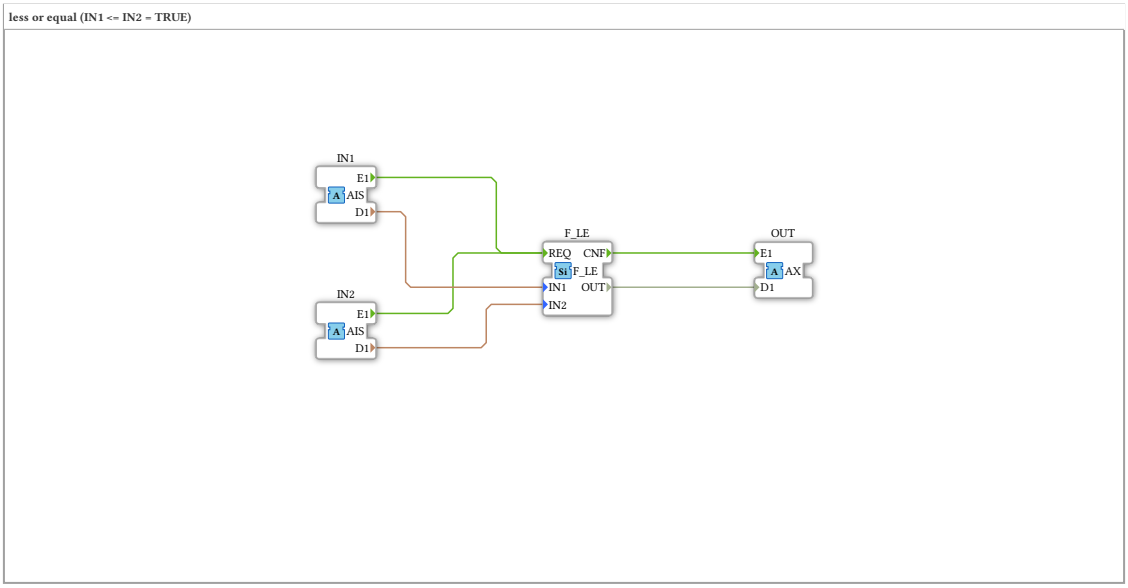
AIS_GT is a simple yet important comparison block for IEC 61131-3 control applications using the STRING data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIS_LE



Input value 1 -- left operand (IN1 <= IN2) – *AIS*
Input value 2 -- right operand (IN1 <= IN2) – *AIS*

AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AIS_LE** compares two values of type `STRING` (character string) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `STRING`, character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AIS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AIS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AIS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LE` (type `iec61131::comparison::F_LE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AIS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the `STRING` data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking recipe and text identifiers against a reference or threshold value of type `STRING`.
- **Control Logic**: Triggering an action once two values of type `STRING` satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `STRING`-based comparison condition.

Comparison with Similar Function Blocks

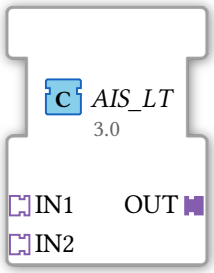
- **AIS_GE**: Checks for “greater than or equal to” – the inverse function of AIS_LE.
- **AIS_EQ**: Checks for equality of values.
- **AIS_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AIS_LE is the extended version.

AIS_LE completes the AIS comparison family with the boundary case of equality.

Conclusion

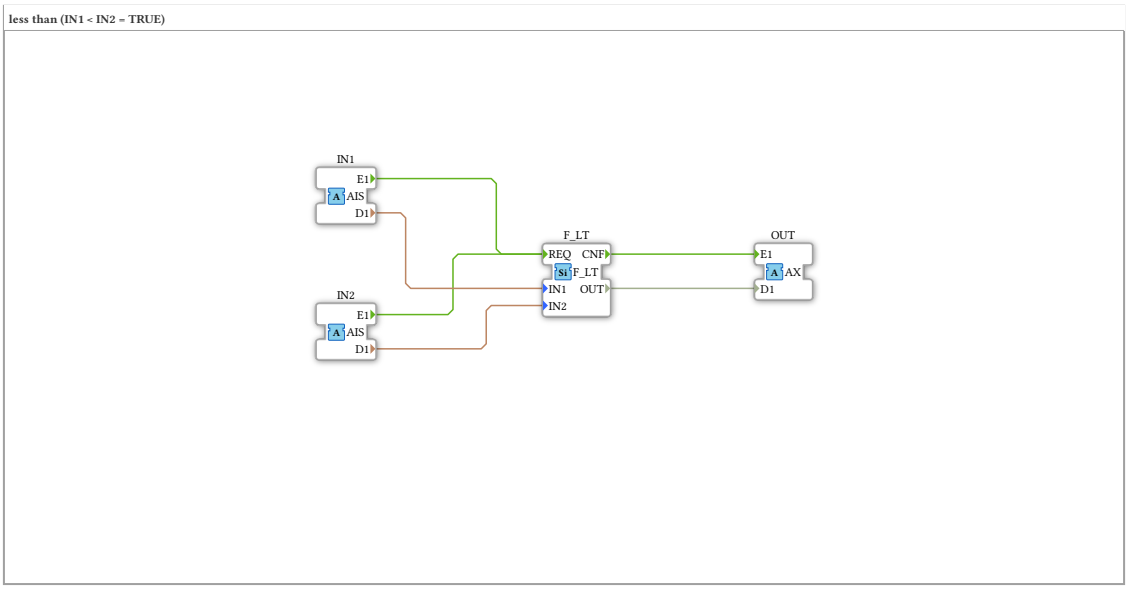
AIS_LE is a simple yet important comparison block for IEC 61131-3 control applications using the `STRING` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIS_LT



Input value 1 -- left operand (IN1 < IN2) – *AIS*
Input value 2 -- right operand (IN1 < IN2) – *AIS*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AIS_LT** compares two values of type `STRING` (character string) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `STRING`, character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AIS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AIS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AIS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LT` (type `iec61131::comparison::F_LT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AIS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the `STRING` data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking recipe and text identifiers against a reference or threshold value of type `STRING`.
- **Control Logic**: Triggering an action once two values of type `STRING` satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `STRING`-based comparison condition.

Comparison with Similar Function Blocks

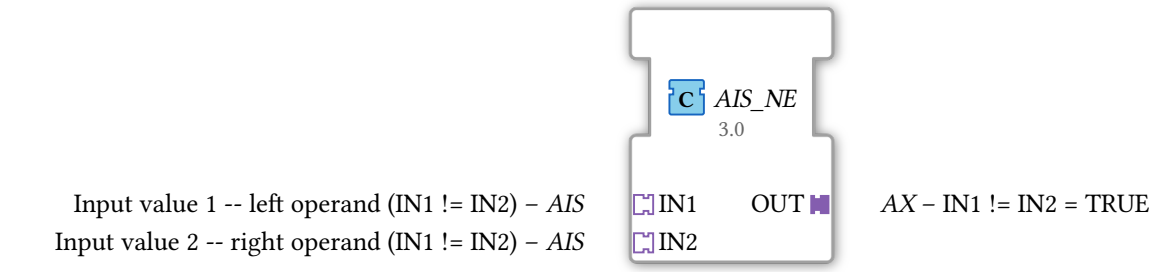
- **AIS_GT**: Checks for “greater than” – the inverse function of AIS_LT.
- **AIS_EQ**: Checks for equality of values.
- **AIS_LE**: Performs the “less than or equal to” comparison – the extended version of AIS_LT that includes the equality case.

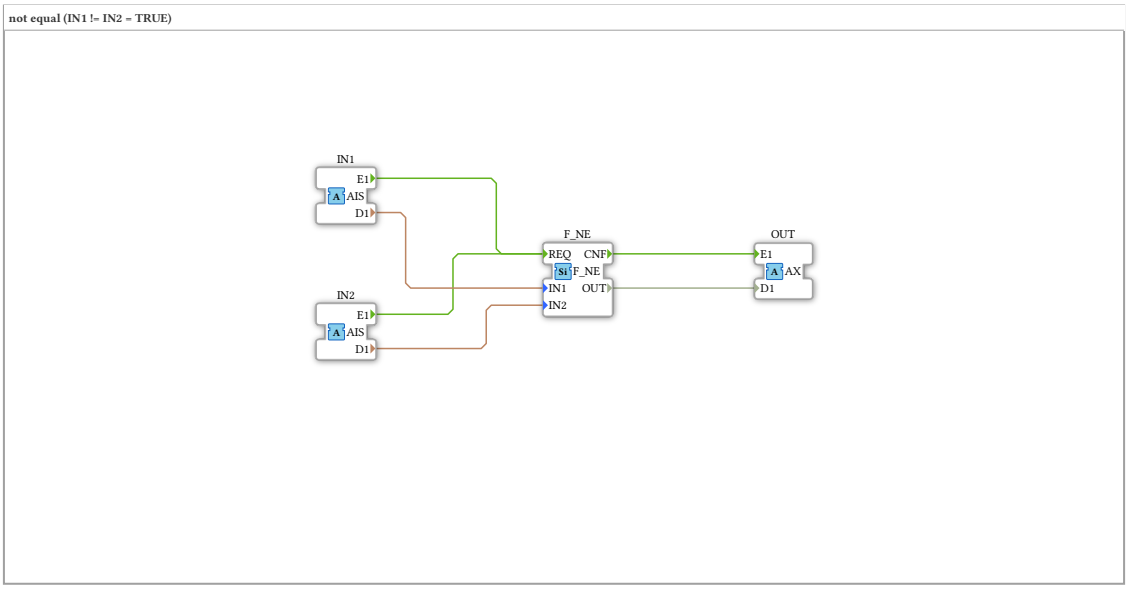
AIS_LT is the basic “less than” component without the equality case, complemented by AIS_LE.

Conclusion

AIS_LT is a simple yet important comparison block for IEC 61131-3 control applications using the `STRING` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIS_NE





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIS_NE** compares two values of type `STRING` (character string) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `STRING`, character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AIS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AIS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AIS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_NE` (type `iec61131::comparison::F_NE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AIS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the `STRING` data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking recipe and text identifiers against a reference or threshold value of type `STRING`.
- **Control Logic**: Triggering an action once two values of type `STRING` satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `STRING`-based comparison condition.

Comparison with Similar Function Blocks

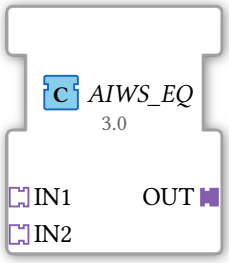
- **AIS_EQ**: Checks for “equal to” – the inverse function of AIS_NE.
- **AIS_EQ**: Checks for equality of values.

AIS_NE is part of the complete AIS comparison family (EQ/NE/GT/GE/LT/LE) for the `STRING` data type.

Conclusion

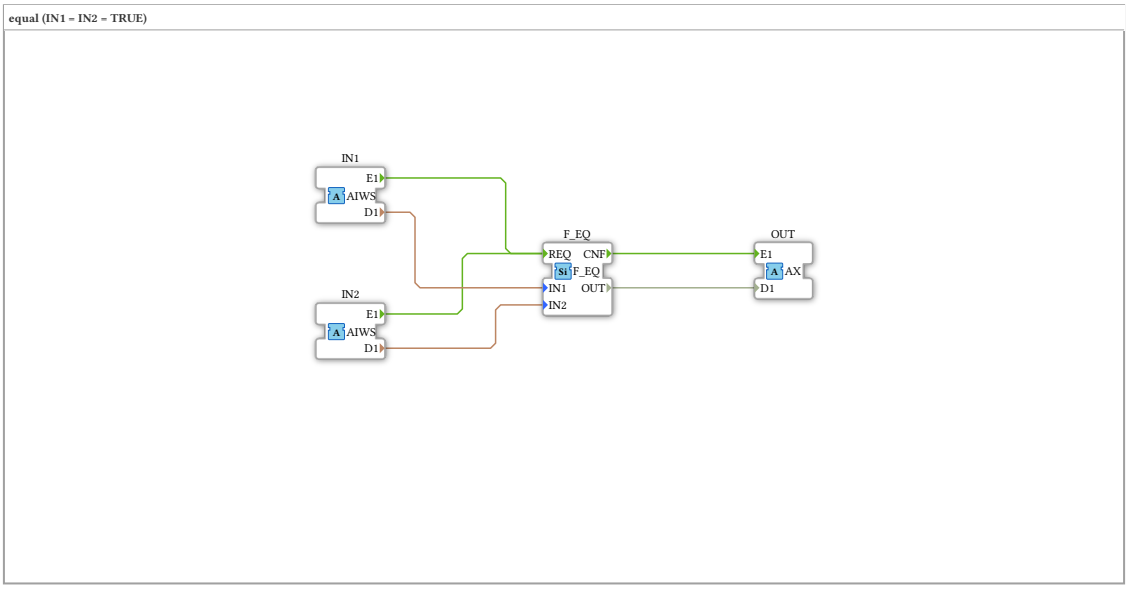
AIS_NE is a simple yet important comparison block for IEC 61131-3 control applications using the `STRING` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIWS_EQ



Input value 1 -- left operand (IN1 = IN2) – *AIWS*
Input value 2 -- right operand (IN1 = IN2) – *AIWS*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AIWS_EQ** compares two values of type `WSTRING` (wide (Unicode) character string) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WSTRING`, wide (Unicode) character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AIWS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AIWS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AIWS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_EQ` (type `iec61131::comparison::F_EQ`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AIWS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the `WSTRING` data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking multilingual text identifiers against a reference or threshold value of type `WSTRING`.
- **Control Logic**: Triggering an action once two values of type `WSTRING` satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WSTRING`-based comparison condition.

Comparison with Similar Function Blocks

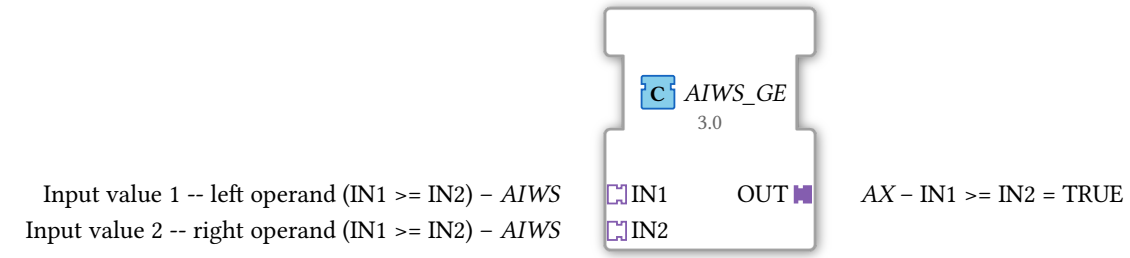
- **AIWS_NE**: Checks for “not equal to” – the inverse function of AIWS_EQ.
- **AIWS_NE**: Checks for inequality of values.

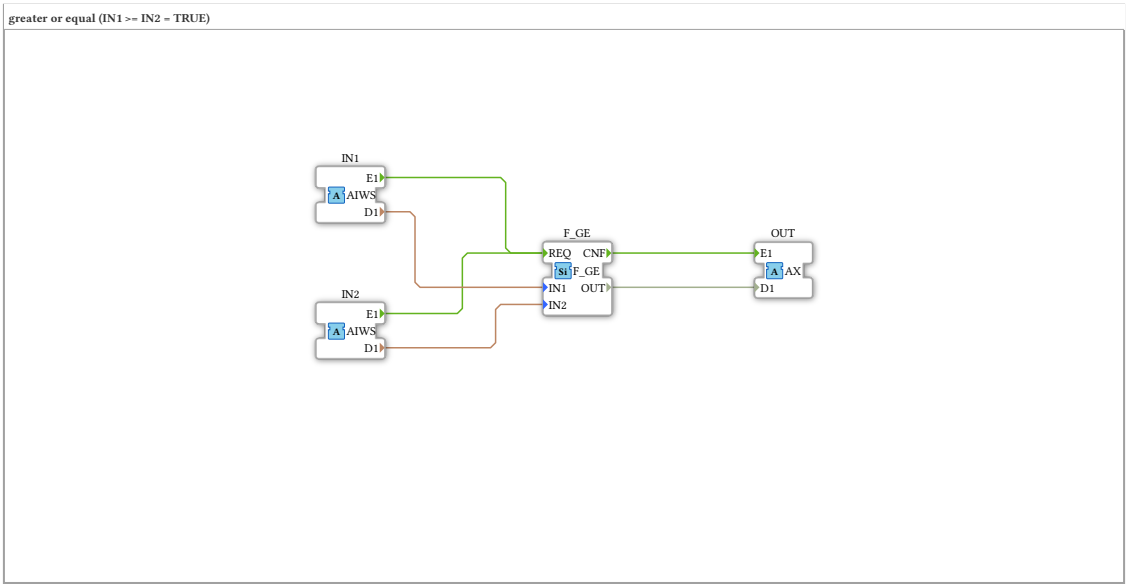
AIWS_EQ is part of the complete AIWS comparison family (EQ/NE/GT/GE/LT/LE) for the `WSTRING` data type.

Conclusion

AIWS_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the `WSTRING` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIWS_GE





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_GE** compares two values of type `WSTRING` (wide (Unicode) character string) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WSTRING`, wide (Unicode) character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AIWS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AIWS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AIWS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GE` (type `iec61131::comparison::F_GE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AIWS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the `WSTRING` data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking multilingual text identifiers against a reference or threshold value of type `WSTRING`.
- **Control Logic**: Triggering an action once two values of type `WSTRING` satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WSTRING`-based comparison condition.

Comparison with Similar Function Blocks

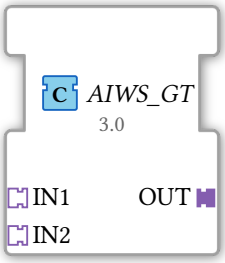
- **AIWS_LE**: Checks for “less than or equal to” – the inverse function of AIWS_GE.
- **AIWS_EQ**: Checks for equality of values.
- **AIWS_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AIWS_GE is the extended version.

AIWS_GE completes the AIWS comparison family with the boundary case of equality.

Conclusion

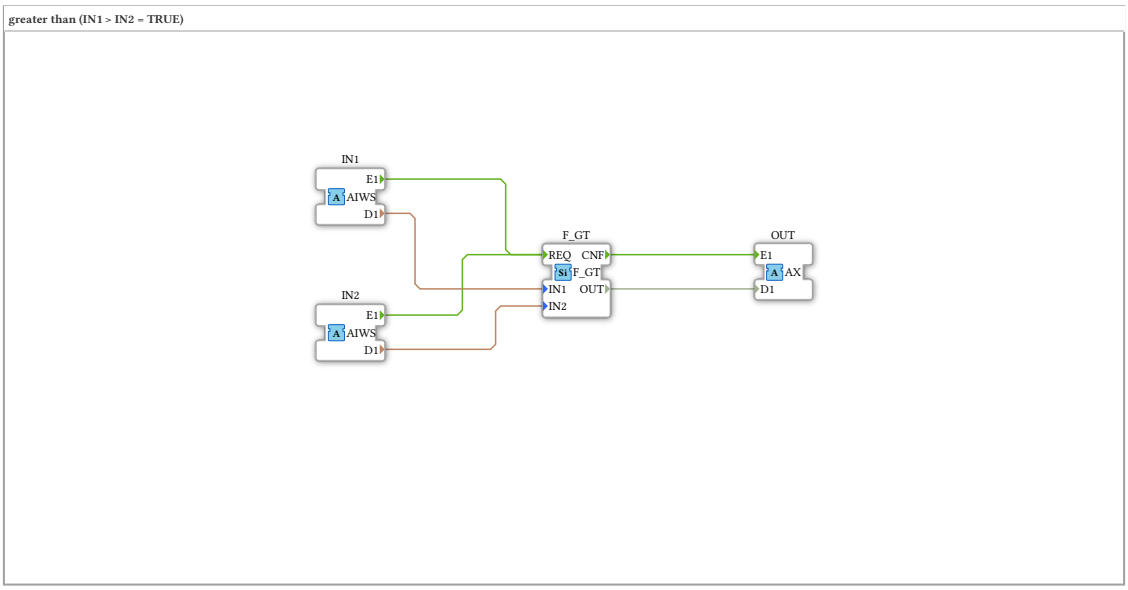
AIWS_GE is a simple yet important comparison block for IEC 61131-3 control applications using the `WSTRING` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIWS_GT



Input value 1 -- left operand (IN1 > IN2) – *AIWS*
Input value 2 -- right operand (IN1 > IN2) – *AIWS*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AIWS_GT** compares two values of type `WSTRING` (wide (Unicode) character string) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WSTRING`, wide (Unicode) character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AIWS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AIWS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AIWS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GT` (type `iec61131::comparison::F_GT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AIWS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the `WSTRING` data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking multilingual text identifiers against a reference or threshold value of type `WSTRING`.
- **Control Logic**: Triggering an action once two values of type `WSTRING` satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WSTRING`-based comparison condition.

Comparison with Similar Function Blocks

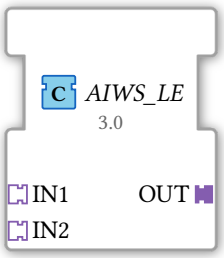
- **AIWS_LT**: Checks for “less than” – the inverse function of AIWS_GT.
- **AIWS_EQ**: Checks for equality of values.
- **AIWS_GE**: Performs the “greater than or equal to” comparison – the extended version of AIWS_GT that includes the equality case.

AIWS_GT is the basic “greater than” component without the equality case, complemented by AIWS_GE.

Conclusion

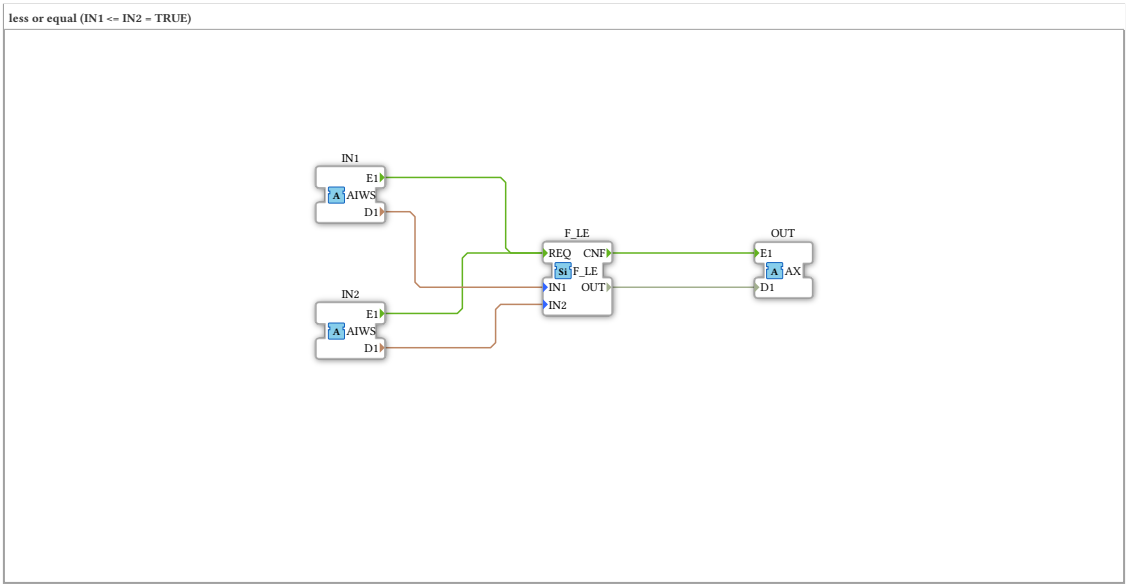
AIWS_GT is a simple yet important comparison block for IEC 61131-3 control applications using the `WSTRING` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIWS_LE



Input value 1 -- left operand (IN1 <= IN2) – *AIWS*
Input value 2 -- right operand (IN1 <= IN2) – *AIWS*

AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AIWS_LE** compares two values of type `WSTRING` (wide (Unicode) character string) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WSTRING`, wide (Unicode) character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AIWS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AIWS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AIWS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LE` (type `iec61131::comparison::F_LE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AIWS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the `WSTRING` data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking multilingual text identifiers against a reference or threshold value of type `WSTRING`.
- **Control Logic**: Triggering an action once two values of type `WSTRING` satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WSTRING`-based comparison condition.

Comparison with Similar Function Blocks

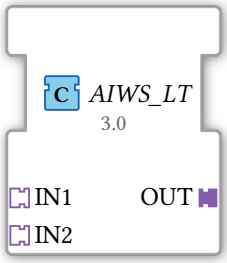
- **AIWS_GE**: Checks for “greater than or equal to” – the inverse function of AIWS_LE.
- **AIWS_EQ**: Checks for equality of values.
- **AIWS_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AIWS_LE is the extended version.

AIWS_LE completes the AIWS comparison family with the boundary case of equality.

Conclusion

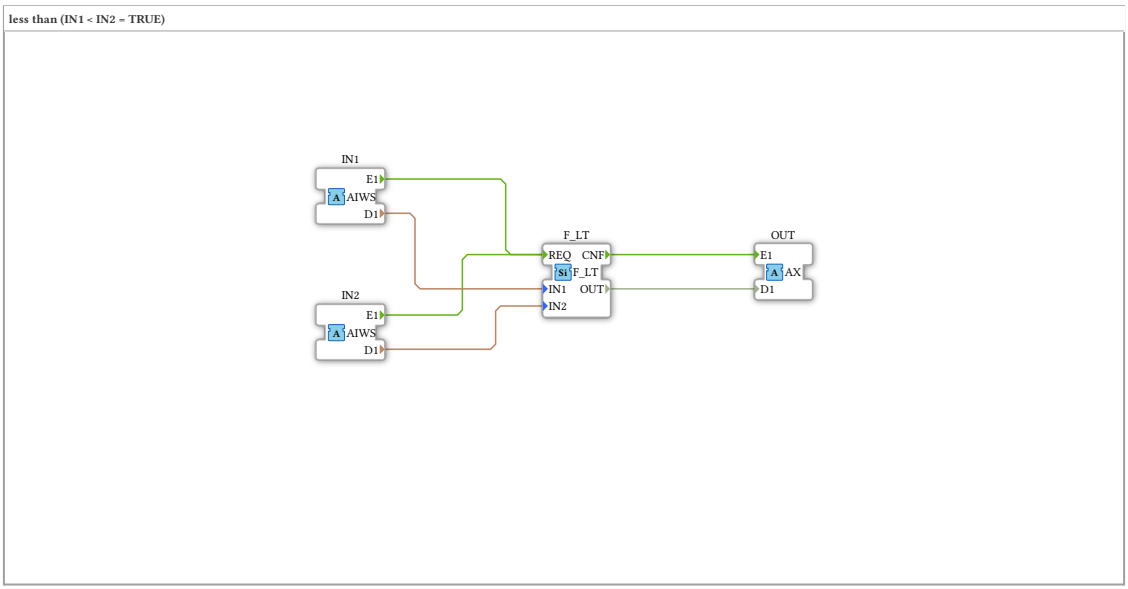
AIWS_LE is a simple yet important comparison block for IEC 61131-3 control applications using the `WSTRING` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIWS_LT



Input value 1 -- left operand (IN1 < IN2) – *AIWS*
Input value 2 -- right operand (IN1 < IN2) – *AIWS*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AIWS_LT** compares two values of type `WSTRING` (wide (Unicode) character string) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WSTRING`, wide (Unicode) character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AIWS (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AIWS (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AIWS and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LT (type iec61131::comparison::F_LT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AIWS/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the WSTRING data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking multilingual text identifiers against a reference or threshold value of type WSTRING.
- **Control Logic**: Triggering an action once two values of type WSTRING satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a WSTRING-based comparison condition.

Comparison with Similar Function Blocks

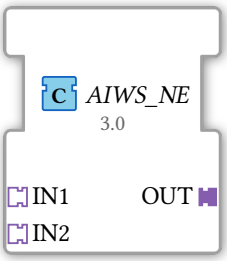
- **AIWS_GT**: Checks for “greater than” – the inverse function of AIWS_LT.
- **AIWS_EQ**: Checks for equality of values.
- **AIWS_LE**: Performs the “less than or equal to” comparison – the extended version of AIWS_LT that includes the equality case.

AIWS_LT is the basic “less than” component without the equality case, complemented by AIWS_LE.

Conclusion

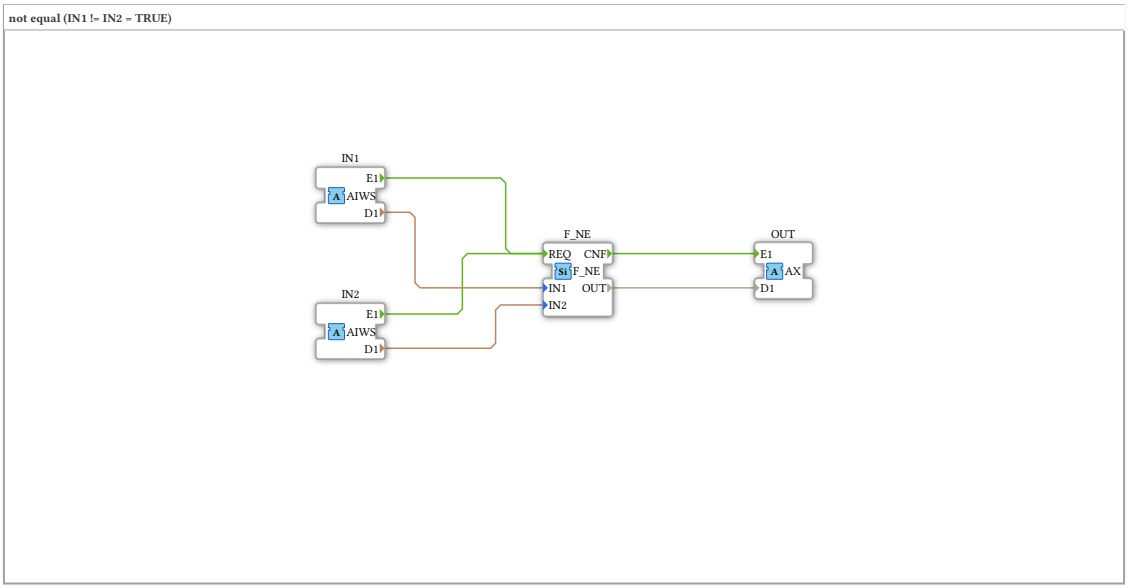
AIWS_LT is a simple yet important comparison block for IEC 61131-3 control applications using the WSTRING data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AIWS_NE



Input value 1 -- left operand (IN1 != IN2) – *AIWS*
Input value 2 -- right operand (IN1 != IN2) – *AIWS*

AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AIWS_NE** compares two values of type `WSTRING` (wide (Unicode) character string) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison is performed lexicographically (character by character) according to the character encoding.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WSTRING`, wide (Unicode) character string).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AIWS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AIWS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AIWS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_NE` (type `iec61131::comparison::F_NE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AIWS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the `WSTRING` data type.
- **Comparison logic**: The comparison is performed lexicographically (character by character) according to the character encoding.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking multilingual text identifiers against a reference or threshold value of type `WSTRING`.
- **Control Logic**: Triggering an action once two values of type `WSTRING` satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WSTRING`-based comparison condition.

Comparison with Similar Function Blocks

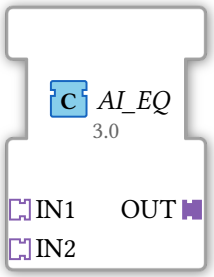
- **AIWS_EQ**: Checks for “equal to” – the inverse function of AIWS_NE.
- **AIWS_EQ**: Checks for equality of values.

AIWS_NE is part of the complete AIWS comparison family (EQ/NE/GT/GE/LT/LE) for the `WSTRING` data type.

Conclusion

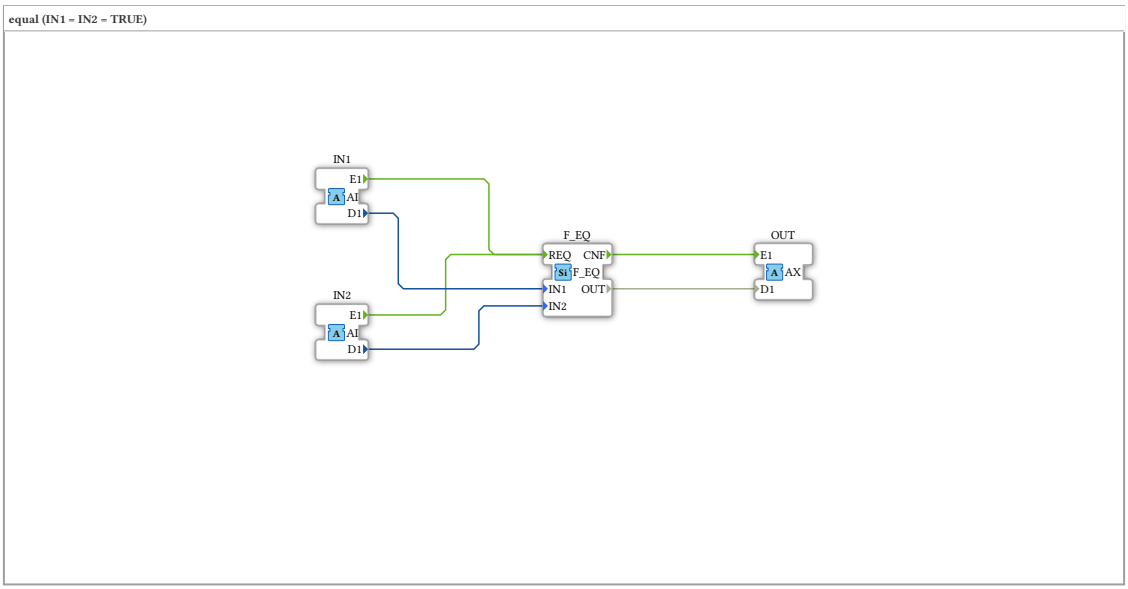
AIWS_NE is a simple yet important comparison block for IEC 61131-3 control applications using the `WSTRING` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AI_EQ



Input value 1 -- left operand (IN1 = IN2) – *AI*
Input value 2 -- right operand (IN1 = IN2) – *AI*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AI_EQ** compares two values of type `INT` (16-bit signed integer) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `INT`, 16-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_EQ` (type `iec61131::comparison::F_EQ`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the `INT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter readings against a reference or threshold value of type `INT`.
- **Control Logic**: Triggering an action once two values of type `INT` satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `INT`-based comparison condition.

Comparison with Similar Function Blocks

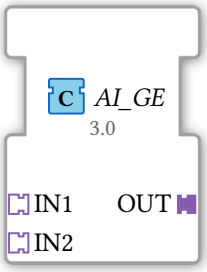
- **AI_NE**: Checks for “not equal to” – the inverse function of AI_EQ.
- **AI_NE**: Checks for inequality of values.

AI_EQ is part of the complete AI comparison family (EQ/NE/GT/GE/LT/LE) for the `INT` data type.

Conclusion

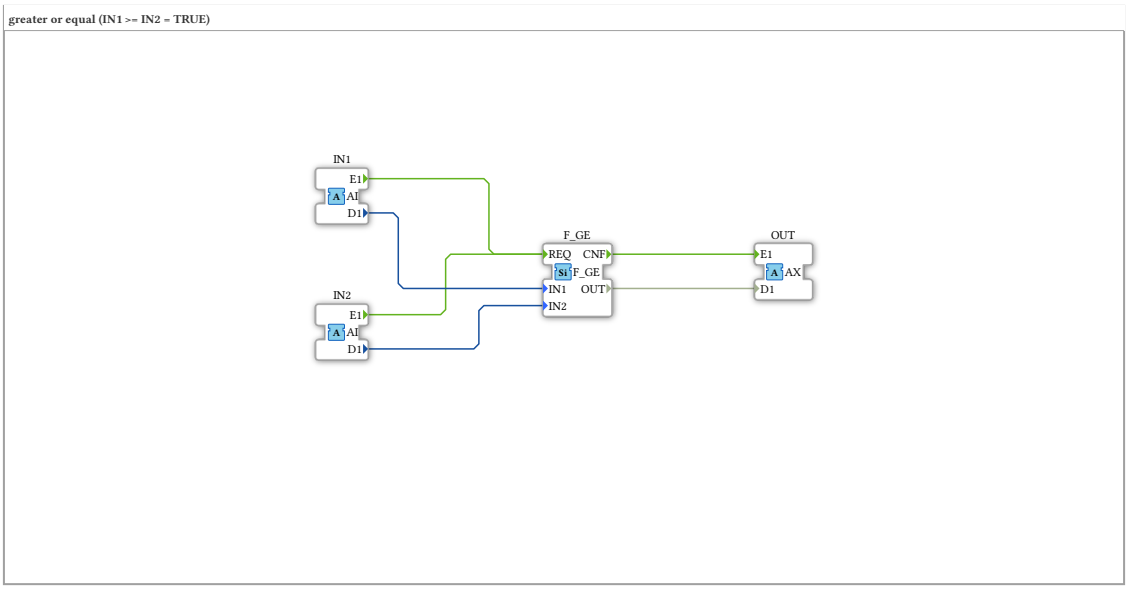
AI_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the `INT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AI_GE



Input value 1 -- left operand (IN1 >= IN2) – *AI*
Input value 2 -- right operand (IN1 >= IN2) – *AI*

AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_GE** compares two values of type `INT` (16-bit signed integer) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `INT`, 16-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GE` (type `iec61131::comparison::F_GE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the `INT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter readings against a reference or threshold value of type `INT`.
- **Control Logic**: Triggering an action once two values of type `INT` satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `INT`-based comparison condition.

Comparison with Similar Function Blocks

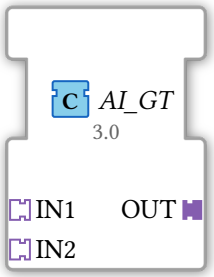
- **AI_LE**: Checks for “less than or equal to” – the inverse function of AI_GE.
- **AI_EQ**: Checks for equality of values.
- **AI_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AI_GE is the extended version.

AI_GE completes the AI comparison family with the boundary case of equality.

Conclusion

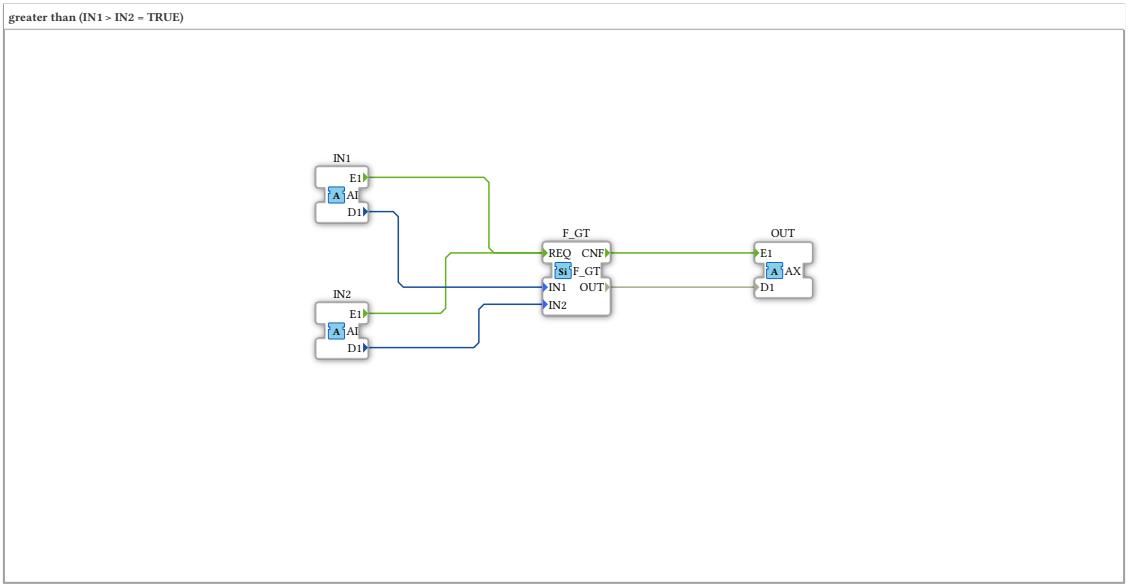
AI_GE is a simple yet important comparison block for IEC 61131-3 control applications using the `INT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AI_GT



Input value 1 -- left operand (IN1 > IN2) – *AI*
Input value 2 -- right operand (IN1 > IN2) – *AI*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AI_GT** compares two values of type `INT` (16-bit signed integer) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `INT`, 16-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GT` (type `iec61131::comparison::F_GT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the `INT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter readings against a reference or threshold value of type `INT`.
- **Control Logic**: Triggering an action once two values of type `INT` satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `INT`-based comparison condition.

Comparison with Similar Function Blocks

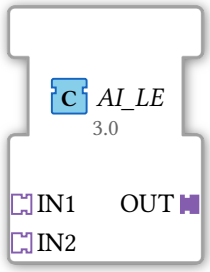
- **AI_LT**: Checks for “less than” – the inverse function of AI_GT.
- **AI_EQ**: Checks for equality of values.
- **AI_GE**: Performs the “greater than or equal to” comparison – the extended version of AI_GT that includes the equality case.

AI_GT is the basic “greater than” component without the equality case, complemented by AI_GE.

Conclusion

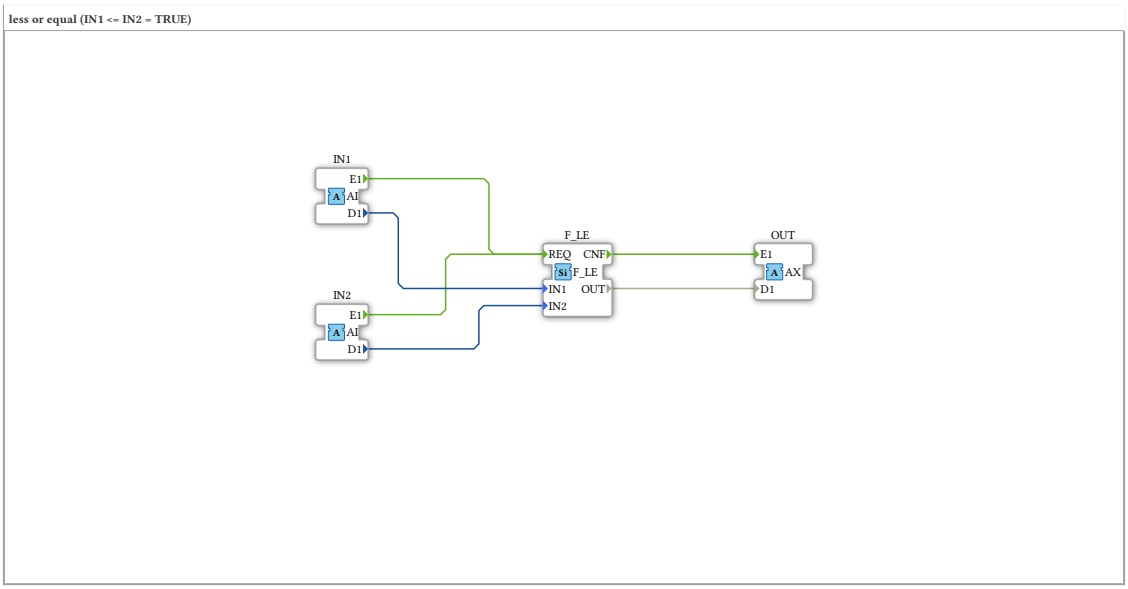
AI_GT is a simple yet important comparison block for IEC 61131-3 control applications using the `INT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AI_LE



Input value 1 -- left operand (IN1 <= IN2) – *AI*
Input value 2 -- right operand (IN1 <= IN2) – *AI*

AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_LE** compares two values of type `INT` (16-bit signed integer) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `INT`, 16-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LE` (type `iec61131::comparison::F_LE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the `INT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter readings against a reference or threshold value of type `INT`.
- **Control Logic**: Triggering an action once two values of type `INT` satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `INT`-based comparison condition.

Comparison with Similar Function Blocks

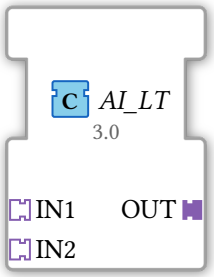
- **AI_GE**: Checks for “greater than or equal to” – the inverse function of AI_LE.
- **AI_EQ**: Checks for equality of values.
- **AI_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AI_LE is the extended version.

AI_LE completes the AI comparison family with the boundary case of equality.

Conclusion

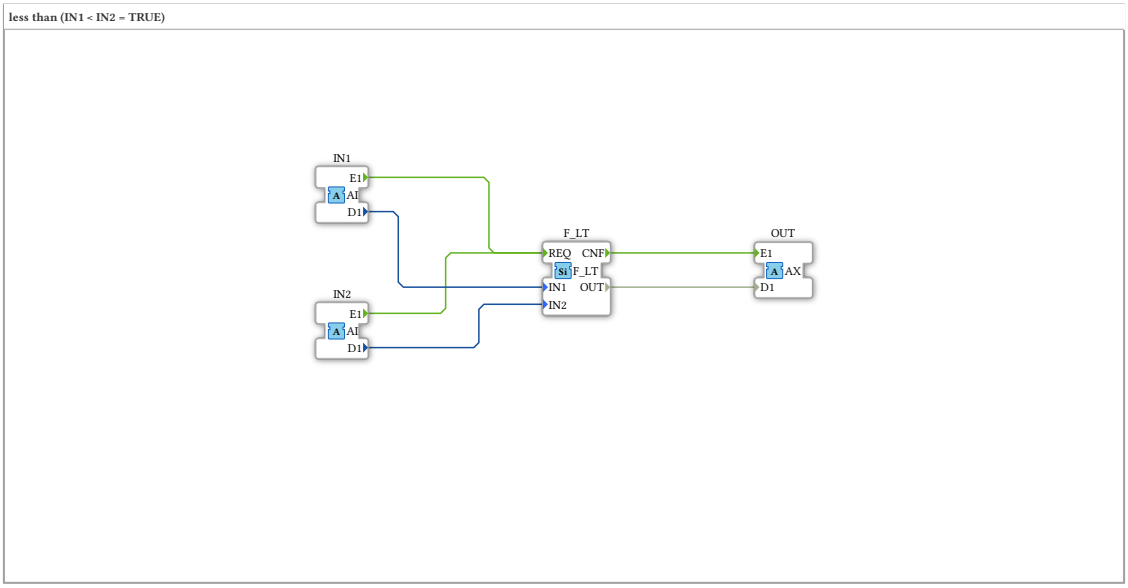
AI_LE is a simple yet important comparison block for IEC 61131-3 control applications using the `INT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AI_LT



Input value 1 -- left operand (IN1 < IN2) – *AI*
Input value 2 -- right operand (IN1 < IN2) – *AI*

AX – IN1 < IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AI_LT** compares two values of type `INT` (16-bit signed integer) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `INT`, 16-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LT` (type `iec61131::comparison::F_LT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the `INT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter readings against a reference or threshold value of type `INT`.
- **Control Logic**: Triggering an action once two values of type `INT` satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `INT`-based comparison condition.

Comparison with Similar Function Blocks

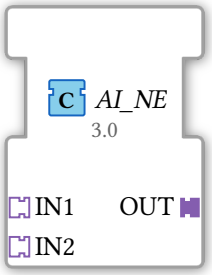
- **AI_GT**: Checks for “greater than” – the inverse function of AI_LT.
- **AI_EQ**: Checks for equality of values.
- **AI_LE**: Performs the “less than or equal to” comparison – the extended version of AI_LT that includes the equality case.

AI_LT is the basic “less than” component without the equality case, complemented by AI_LE.

Conclusion

AI_LT is a simple yet important comparison block for IEC 61131-3 control applications using the `INT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

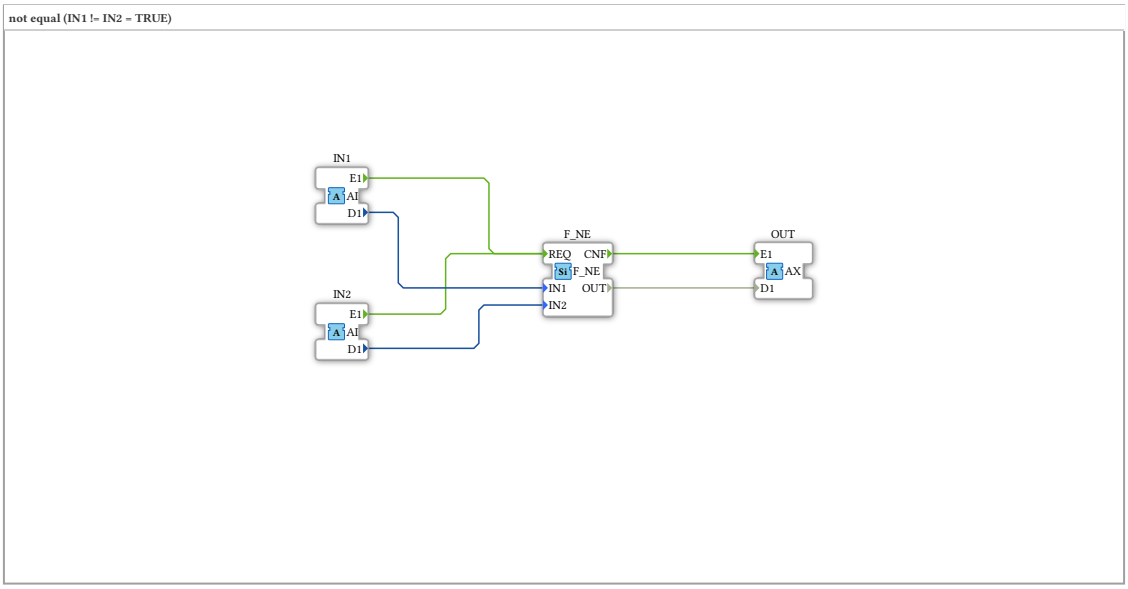
AI_NE



Input value 1 -- left operand (IN1 != IN2) – *AI*

Input value 2 -- right operand (IN1 != IN2) – *AI*

AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AI_NE** compares two values of type `INT` (16-bit signed integer) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `INT`, 16-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_NE` (type `iec61131::comparison::F_NE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the `INT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking counter readings against a reference or threshold value of type `INT`.
- **Control Logic**: Triggering an action once two values of type `INT` satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `INT`-based comparison condition.

Comparison with Similar Function Blocks

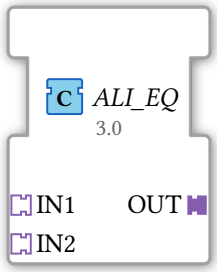
- **AI_EQ**: Checks for “equal to” – the inverse function of AI_NE.
- **AI_EQ**: Checks for equality of values.

AI_NE is part of the complete AI comparison family (EQ/NE/GT/GE/LT/LE) for the `INT` data type.

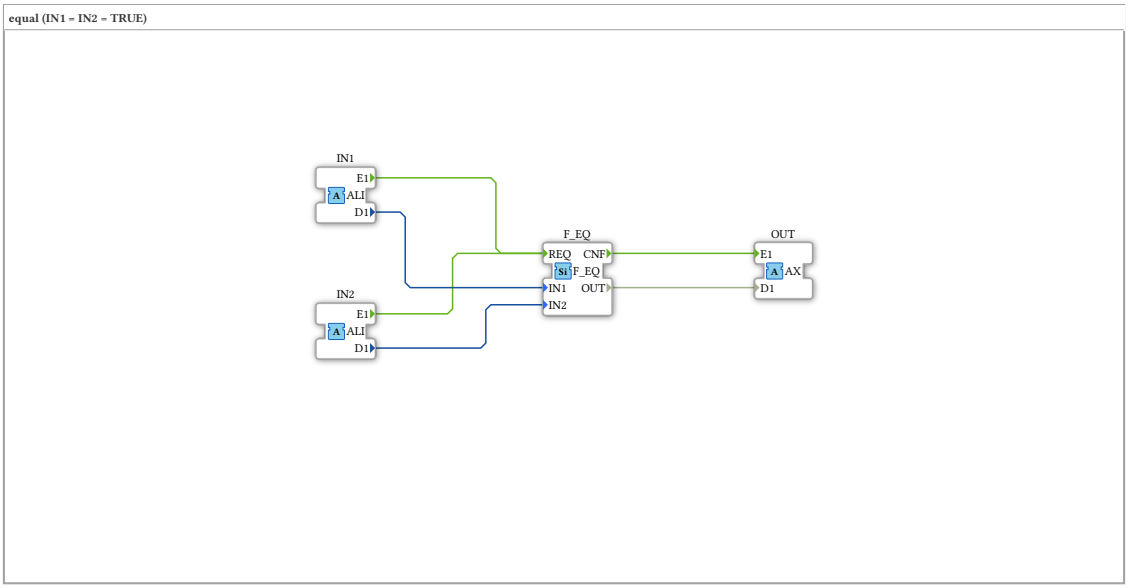
Conclusion

AI_NE is a simple yet important comparison block for IEC 61131-3 control applications using the `INT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALI_EQ



Input value 1 -- left operand (IN1 = IN2) – *ALI*
Input value 2 -- right operand (IN1 = IN2) – *ALI*
AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ALI_EQ** compares two values of type `LINT` (64-bit signed integer) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LINT`, 64-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ALI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ALI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ALI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_EQ` (type `iec61131::comparison::F_EQ`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ALI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the `LINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and total values against a reference or threshold value of type `LINT`.
- **Control Logic**: Triggering an action once two values of type `LINT` satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `LINT`-based comparison condition.

Comparison with Similar Function Blocks

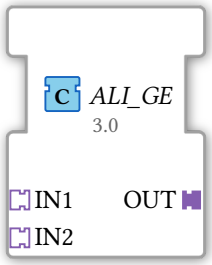
- **ALI_NE**: Checks for “not equal to” – the inverse function of ALI_EQ.
- **ALI_NE**: Checks for inequality of values.

ALI_EQ is part of the complete ALI comparison family (EQ/NE/GT/GE/LT/LE) for the `LINT` data type.

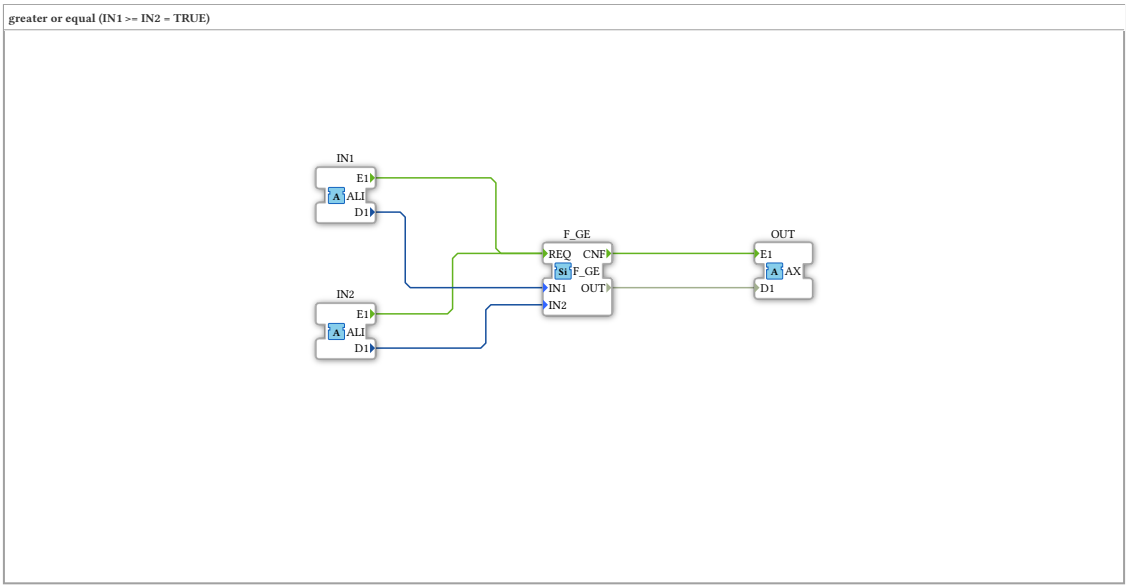
Conclusion

ALI_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the `LINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALI_GE



Input value 1 -- left operand (IN1 >= IN2) – *ALI*
Input value 2 -- right operand (IN1 >= IN2) – *ALI*
AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ALI_GE** compares two values of type `LINT` (64-bit signed integer) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LINT`, 64-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ALI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ALI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ALI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GE` (type `iec61131::comparison::F_GE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ALI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the `LINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and total values against a reference or threshold value of type `LINT`.
- **Control Logic**: Triggering an action once two values of type `LINT` satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `LINT`-based comparison condition.

Comparison with Similar Function Blocks

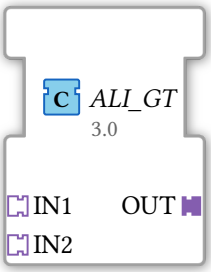
- **ALI_LE**: Checks for “less than or equal to” – the inverse function of ALI_GE.
- **ALI_EQ**: Checks for equality of values.
- **ALI_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which ALI_GE is the extended version.

ALI_GE completes the ALI comparison family with the boundary case of equality.

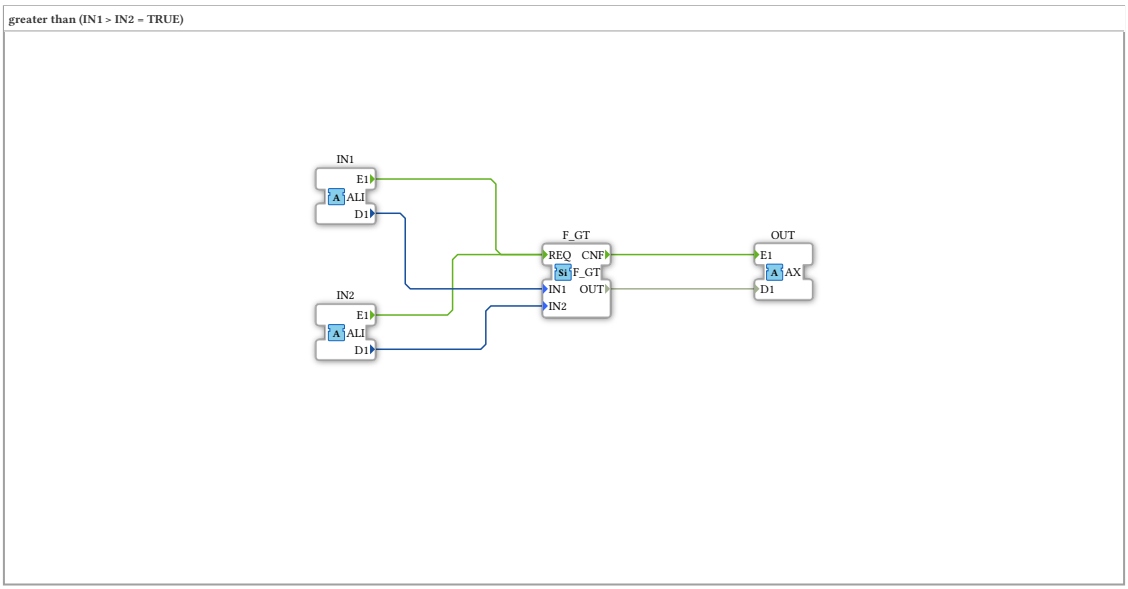
Conclusion

ALI_GE is a simple yet important comparison block for IEC 61131-3 control applications using the `LINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALI_GT



Input value 1 -- left operand (IN1 > IN2) – *ALI*
Input value 2 -- right operand (IN1 > IN2) – *ALI*
AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_GT** compares two values of type `LINT` (64-bit signed integer) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LINT`, 64-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ALI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ALI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ALI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GT` (type `iec61131::comparison::F_GT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ALI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the `LINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and total values against a reference or threshold value of type `LINT`.
- **Control Logic**: Triggering an action once two values of type `LINT` satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `LINT`-based comparison condition.

Comparison with Similar Function Blocks

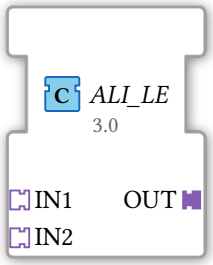
- **ALI_LT**: Checks for “less than” – the inverse function of ALI_GT.
- **ALI_EQ**: Checks for equality of values.
- **ALI_GE**: Performs the “greater than or equal to” comparison – the extended version of ALI_GT that includes the equality case.

ALI_GT is the basic “greater than” component without the equality case, complemented by ALI_GE.

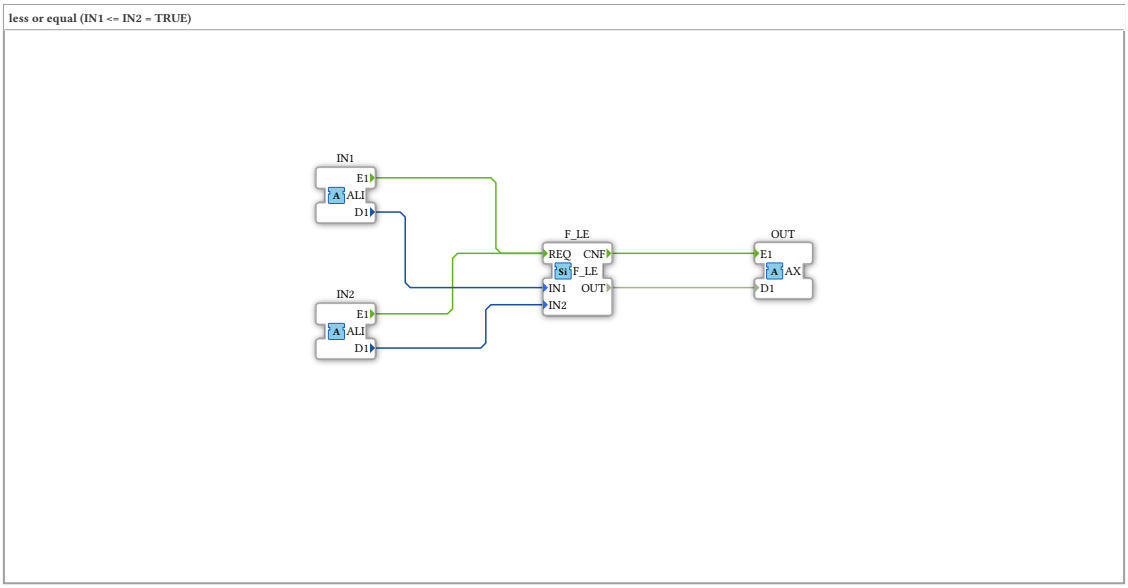
Conclusion

ALI_GT is a simple yet important comparison block for IEC 61131-3 control applications using the `LINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALI_LE



Input value 1 -- left operand (IN1 <= IN2) – *ALI*
Input value 2 -- right operand (IN1 <= IN2) – *ALI*
AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ALI_LE** compares two values of type `LINT` (64-bit signed integer) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LINT`, 64-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ALI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ALI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ALI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LE` (type `iec61131::comparison::F_LE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ALI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the `LINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and total values against a reference or threshold value of type `LINT`.
- **Control Logic**: Triggering an action once two values of type `LINT` satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `LINT`-based comparison condition.

Comparison with Similar Function Blocks

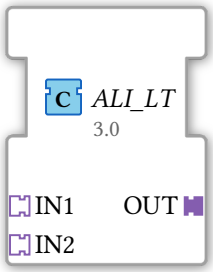
- **ALI_GE**: Checks for “greater than or equal to” – the inverse function of ALI_LE.
- **ALI_EQ**: Checks for equality of values.
- **ALI_LT**: Performs the “less than” comparison – the base variant without the equality case, of which ALI_LE is the extended version.

ALI_LE completes the ALI comparison family with the boundary case of equality.

Conclusion

ALI_LE is a simple yet important comparison block for IEC 61131-3 control applications using the `LINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

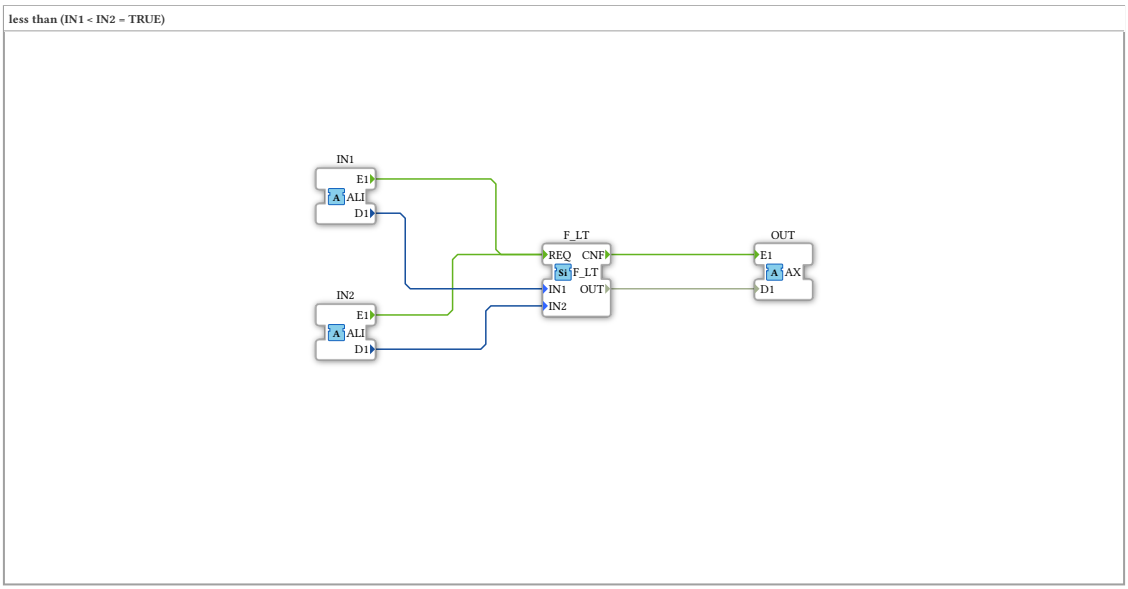
ALI_LT



Input value 1 -- left operand (IN1 < IN2) – *ALI*

Input value 2 -- right operand (IN1 < IN2) – *ALI*

AX – IN1 < IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_LT** compares two values of type `LINT` (64-bit signed integer) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LINT`, 64-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ALI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ALI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ALI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LT` (type `iec61131::comparison::F_LT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ALI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the `LINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and total values against a reference or threshold value of type `LINT`.
- **Control Logic**: Triggering an action once two values of type `LINT` satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `LINT`-based comparison condition.

Comparison with Similar Function Blocks

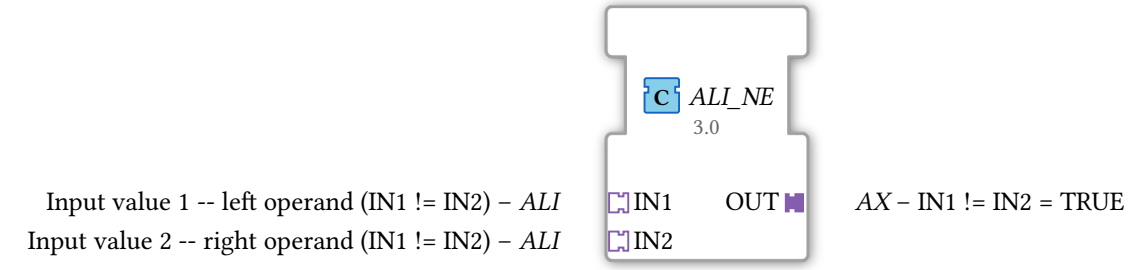
- **ALI_GT**: Checks for “greater than” – the inverse function of ALI_LT.
- **ALI_EQ**: Checks for equality of values.
- **ALI_LE**: Performs the “less than or equal to” comparison – the extended version of ALI_LT that includes the equality case.

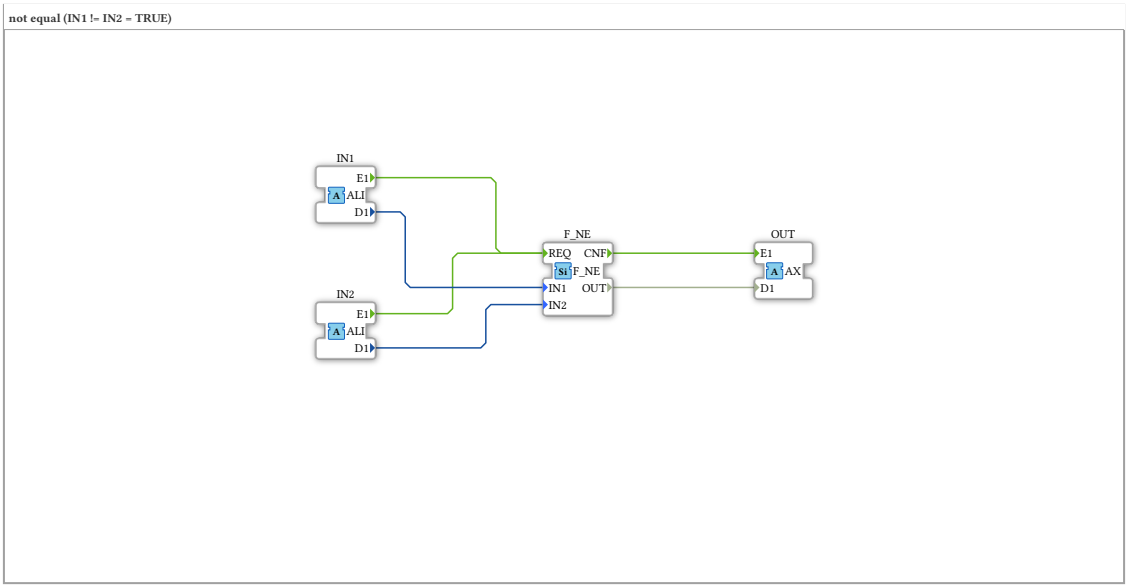
ALI_LT is the basic “less than” component without the equality case, complemented by ALI_LE.

Conclusion

ALI_LT is a simple yet important comparison block for IEC 61131-3 control applications using the `LINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALI_NE





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ALI_NE** compares two values of type `LINT` (64-bit signed integer) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LINT`, 64-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ALI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ALI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ALI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_NE` (type `iec61131::comparison::F_NE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ALI`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the `LINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and total values against a reference or threshold value of type `LINT`.
- **Control Logic**: Triggering an action once two values of type `LINT` satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `LINT`-based comparison condition.

Comparison with Similar Function Blocks

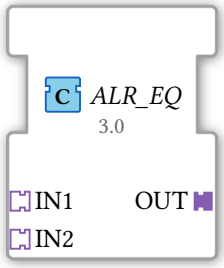
- **ALI_EQ**: Checks for “equal to” – the inverse function of ALI_NE.
- **ALI_EQ**: Checks for equality of values.

ALI_NE is part of the complete ALI comparison family (EQ/NE/GT/GE/LT/LE) for the `LINT` data type.

Conclusion

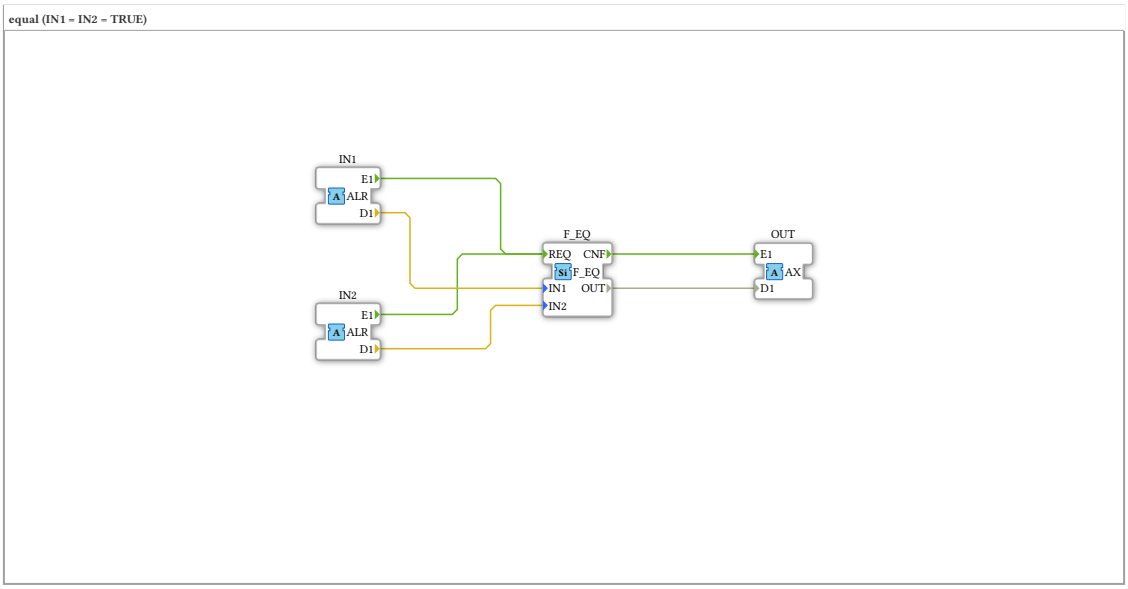
ALI_NE is a simple yet important comparison block for IEC 61131-3 control applications using the `LINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALR_EQ



Input value 1 -- left operand (IN1 = IN2) – *ALR*
Input value 2 -- right operand (IN1 = IN2) – *ALR*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_EQ** compares two values of type `LREAL` (64-bit floating-point number (double precision)) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LREAL`, 64-bit floating-point number (double precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ALR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ALR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ALR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_EQ (type iec61131::comparison::F_EQ). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ALR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the LREAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking high-precision measurements against a reference or threshold value of type LREAL.
- **Control Logic**: Triggering an action once two values of type LREAL satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LREAL-based comparison condition.

Comparison with Similar Function Blocks

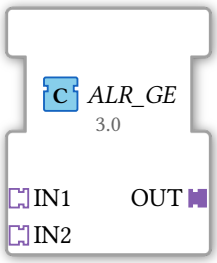
- **ALR_NE**: Checks for “not equal to” – the inverse function of ALR_EQ.
- **ALR_NE**: Checks for inequality of values.

ALR_EQ is part of the complete ALR comparison family (EQ/NE/GT/GE/LT/LE) for the LREAL data type.

Conclusion

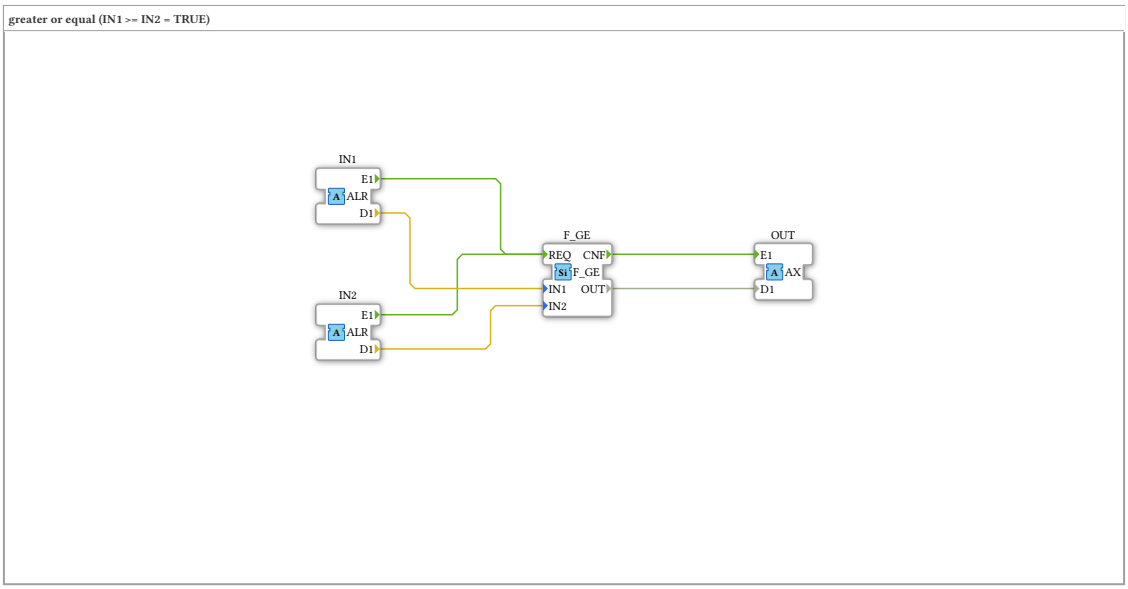
ALR_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the LREAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALR_GE



Input value 1 -- left operand (IN1 >= IN2) – *ALR*
Input value 2 -- right operand (IN1 >= IN2) – *ALR*

AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_GE** compares two values of type `LREAL` (64-bit floating-point number (double precision)) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LREAL`, 64-bit floating-point number (double precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ALR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ALR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ALR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GE (type iec61131::comparison::F_GE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ALR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the LREAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking high-precision measurements against a reference or threshold value of type LREAL.
- **Control Logic**: Triggering an action once two values of type LREAL satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LREAL-based comparison condition.

Comparison with Similar Function Blocks

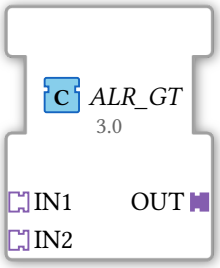
- **ALR_LE**: Checks for “less than or equal to” – the inverse function of ALR_GE.
- **ALR_EQ**: Checks for equality of values.
- **ALR_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which ALR_GE is the extended version.

ALR_GE completes the ALR comparison family with the boundary case of equality.

Conclusion

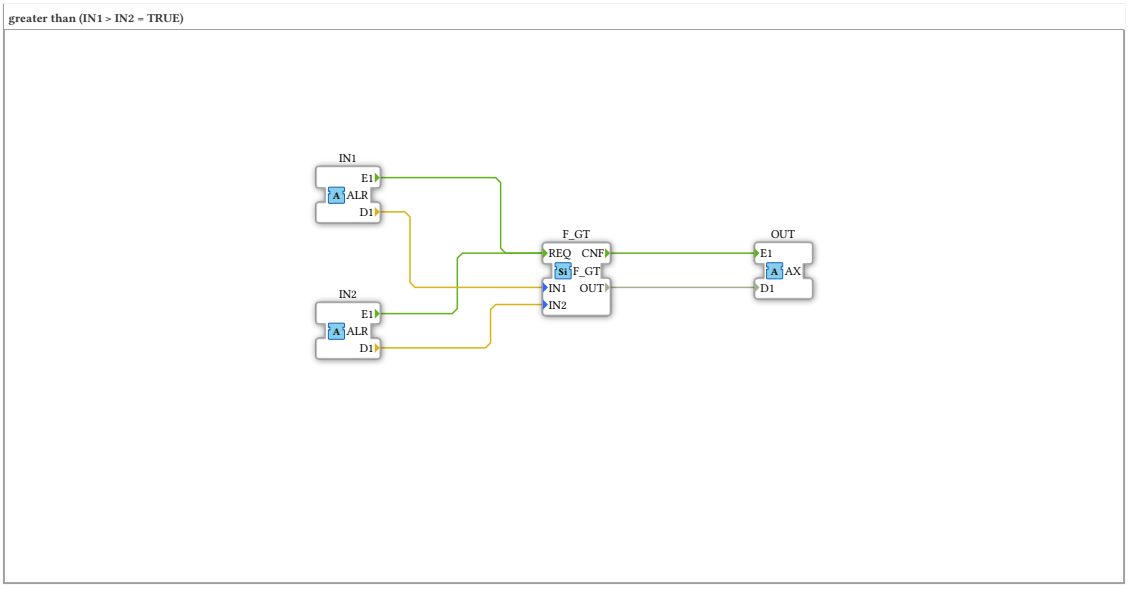
ALR_GE is a simple yet important comparison block for IEC 61131-3 control applications using the LREAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALR_GT



Input value 1 -- left operand (IN1 > IN2) – *ALR*
Input value 2 -- right operand (IN1 > IN2) – *ALR*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_GT** compares two values of type `LREAL` (64-bit floating-point number (double precision)) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LREAL`, 64-bit floating-point number (double precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ALR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ALR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ALR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GT (type iec61131::comparison::F_GT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ALR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the LREAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking high-precision measurements against a reference or threshold value of type LREAL.
- **Control Logic**: Triggering an action once two values of type LREAL satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LREAL-based comparison condition.

Comparison with Similar Function Blocks

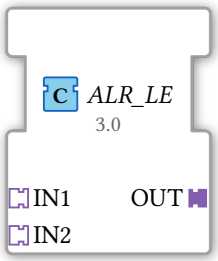
- **ALR_LT**: Checks for “less than” – the inverse function of ALR_GT.
- **ALR_EQ**: Checks for equality of values.
- **ALR_GE**: Performs the “greater than or equal to” comparison – the extended version of ALR_GT that includes the equality case.

ALR_GT is the basic “greater than” component without the equality case, complemented by ALR_GE.

Conclusion

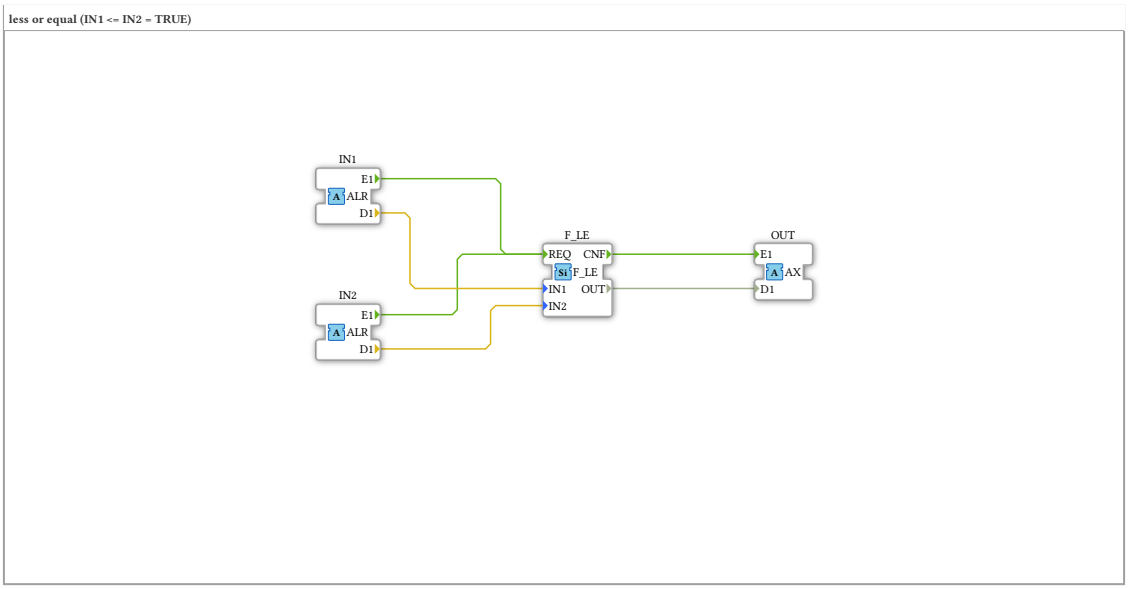
ALR_GT is a simple yet important comparison block for IEC 61131-3 control applications using the LREAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALR_LE



Input value 1 -- left operand (IN1 <= IN2) – *ALR*
Input value 2 -- right operand (IN1 <= IN2) – *ALR*

AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_LE** compares two values of type `LREAL` (64-bit floating-point number (double precision)) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LREAL`, 64-bit floating-point number (double precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ALR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ALR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ALR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LE (type iec61131::comparison::F_LE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ALR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the LREAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking high-precision measurements against a reference or threshold value of type LREAL.
- **Control Logic**: Triggering an action once two values of type LREAL satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LREAL-based comparison condition.

Comparison with Similar Function Blocks

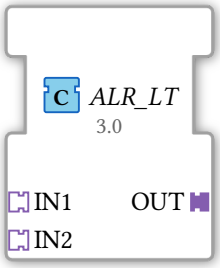
- **ALR_GE**: Checks for “greater than or equal to” – the inverse function of ALR_LE.
- **ALR_EQ**: Checks for equality of values.
- **ALR_LT**: Performs the “less than” comparison – the base variant without the equality case, of which ALR_LE is the extended version.

ALR_LE completes the ALR comparison family with the boundary case of equality.

Conclusion

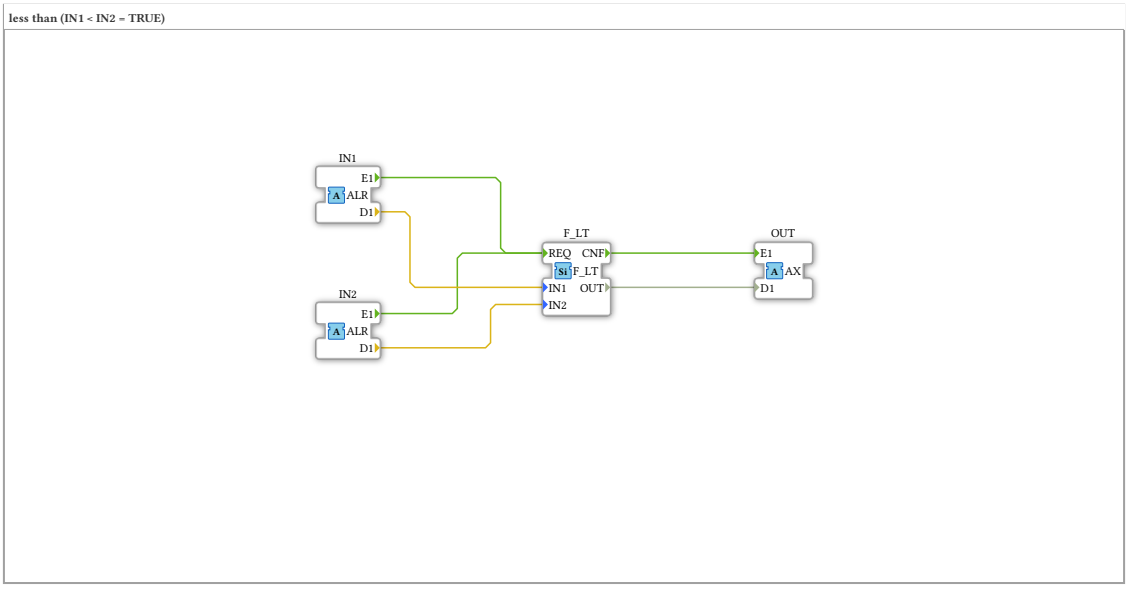
ALR_LE is a simple yet important comparison block for IEC 61131-3 control applications using the LREAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALR_LT



Input value 1 -- left operand (IN1 < IN2) – *ALR*
Input value 2 -- right operand (IN1 < IN2) – *ALR*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_LT** compares two values of type `LREAL` (64-bit floating-point number (double precision)) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LREAL`, 64-bit floating-point number (double precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ALR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ALR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ALR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LT (type iec61131::comparison::F_LT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ALR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the LREAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking high-precision measurements against a reference or threshold value of type LREAL.
- **Control Logic**: Triggering an action once two values of type LREAL satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LREAL-based comparison condition.

Comparison with Similar Function Blocks

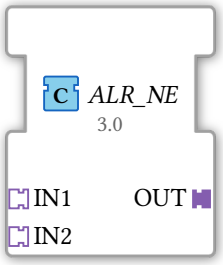
- **ALR_GT**: Checks for “greater than” – the inverse function of ALR_LT.
- **ALR_EQ**: Checks for equality of values.
- **ALR_LE**: Performs the “less than or equal to” comparison – the extended version of ALR_LT that includes the equality case.

ALR_LT is the basic “less than” component without the equality case, complemented by ALR_LE.

Conclusion

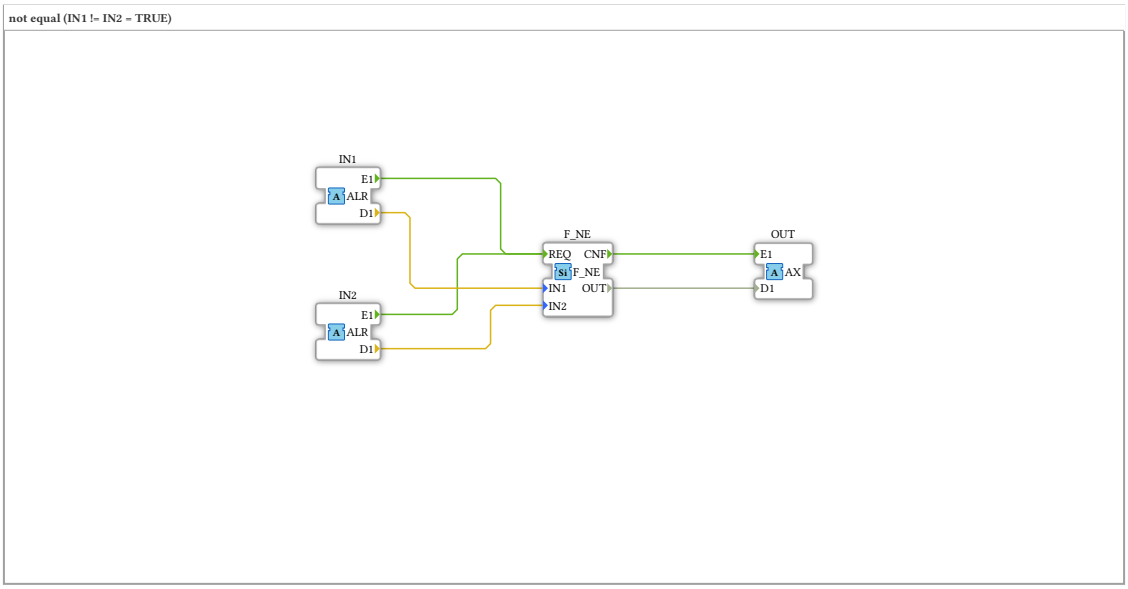
ALR_LT is a simple yet important comparison block for IEC 61131-3 control applications using the LREAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ALR_NE



Input value 1 -- left operand (IN1 != IN2) – *ALR*
Input value 2 -- right operand (IN1 != IN2) – *ALR*

AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_NE** compares two values of type `LREAL` (64-bit floating-point number (double precision)) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LREAL`, 64-bit floating-point number (double precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ALR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ALR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ALR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_NE (type iec61131::comparison::F_NE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ALR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the LREAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking high-precision measurements against a reference or threshold value of type LREAL.
- **Control Logic**: Triggering an action once two values of type LREAL satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LREAL-based comparison condition.

Comparison with Similar Function Blocks

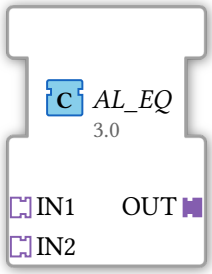
- **ALR_EQ**: Checks for “equal to” – the inverse function of ALR_NE.
- **ALR_EQ**: Checks for equality of values.

ALR_NE is part of the complete ALR comparison family (EQ/NE/GT/GE/LT/LE) for the LREAL data type.

Conclusion

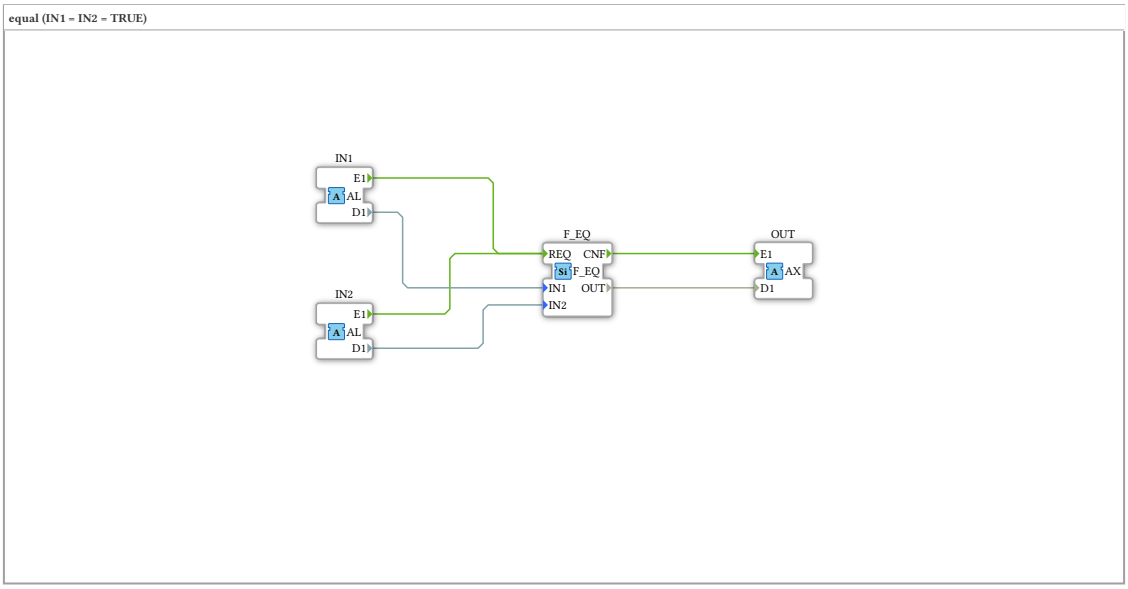
ALR_NE is a simple yet important comparison block for IEC 61131-3 control applications using the LREAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AL_EQ



Input value 1 -- left operand (IN1 = IN2) – *AL*
Input value 2 -- right operand (IN1 = IN2) – *AL*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AL_EQ** compares two values of type `LWORD` (64-bit bit pattern (long word)) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LWORD`, 64-bit bit pattern (long word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AL (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AL (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AL and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_EQ (type iec61131::comparison::F_EQ). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AL/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the LWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking extended bit masks against a reference or threshold value of type LWORD.
- **Control Logic**: Triggering an action once two values of type LWORD satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LWORD-based comparison condition.

Comparison with Similar Function Blocks

- **AL_NE**: Checks for “not equal to” – the inverse function of AL_EQ.
- **AL_NE**: Checks for inequality of values.

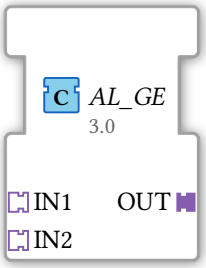
AL_EQ is part of the complete AL comparison family (EQ/NE/GT/GE/LT/LE) for the LWORD data type.

Conclusion

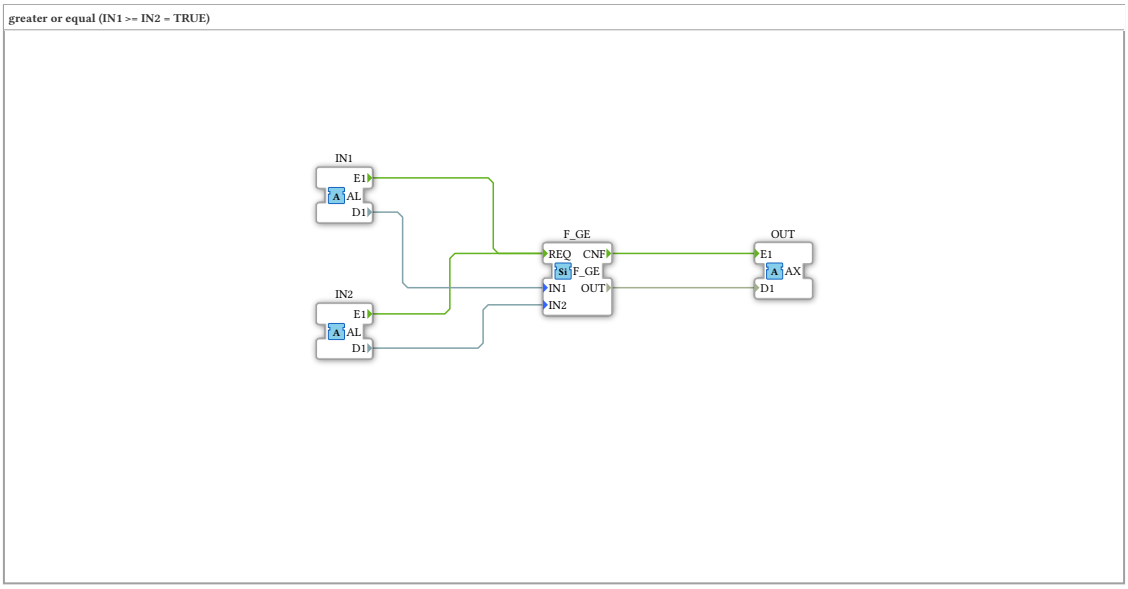
AL_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the LWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AL_GE

Input value 1 -- left operand (IN1 >= IN2) – *AL*
Input value 2 -- right operand (IN1 >= IN2) – *AL*



AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_GE** compares two values of type `LWORD` (64-bit bit pattern (long word)) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LWORD`, 64-bit bit pattern (long word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AL (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AL (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AL and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GE (type iec61131::comparison::F_GE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AL/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the LWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking extended bit masks against a reference or threshold value of type LWORD.
- **Control Logic**: Triggering an action once two values of type LWORD satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LWORD-based comparison condition.

Comparison with Similar Function Blocks

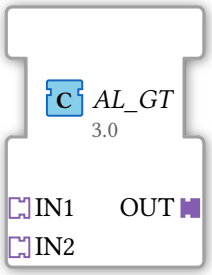
- **AL_LE**: Checks for “less than or equal to” – the inverse function of AL_GE.
- **AL_EQ**: Checks for equality of values.
- **AL_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AL_GE is the extended version.

AL_GE completes the AL comparison family with the boundary case of equality.

Conclusion

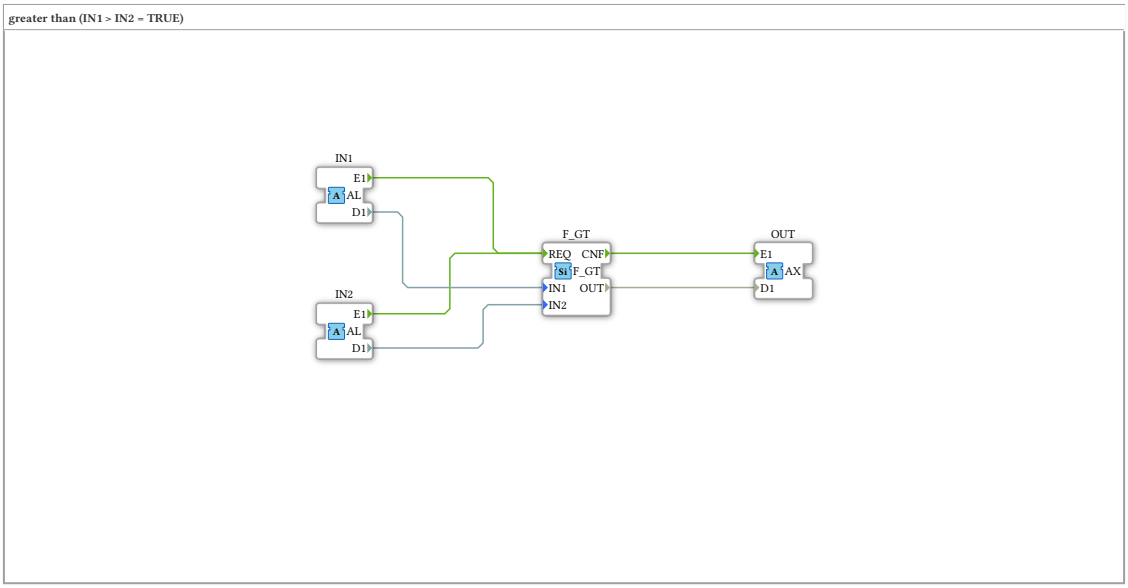
AL_GE is a simple yet important comparison block for IEC 61131-3 control applications using the LWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AL_GT



Input value 1 -- left operand (IN1 > IN2) – *AL*
Input value 2 -- right operand (IN1 > IN2) – *AL*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AL_GT** compares two values of type `LWORD` (64-bit bit pattern (long word)) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LWORD`, 64-bit bit pattern (long word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AL (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AL (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AL and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GT (type iec61131::comparison::F_GT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AL/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the LWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking extended bit masks against a reference or threshold value of type LWORD.
- **Control Logic**: Triggering an action once two values of type LWORD satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LWORD-based comparison condition.

Comparison with Similar Function Blocks

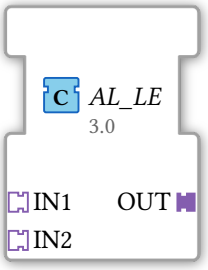
- **AL_LT**: Checks for “less than” – the inverse function of AL_GT.
- **AL_EQ**: Checks for equality of values.
- **AL_GE**: Performs the “greater than or equal to” comparison – the extended version of AL_GT that includes the equality case.

AL_GT is the basic “greater than” component without the equality case, complemented by AL_GE.

Conclusion

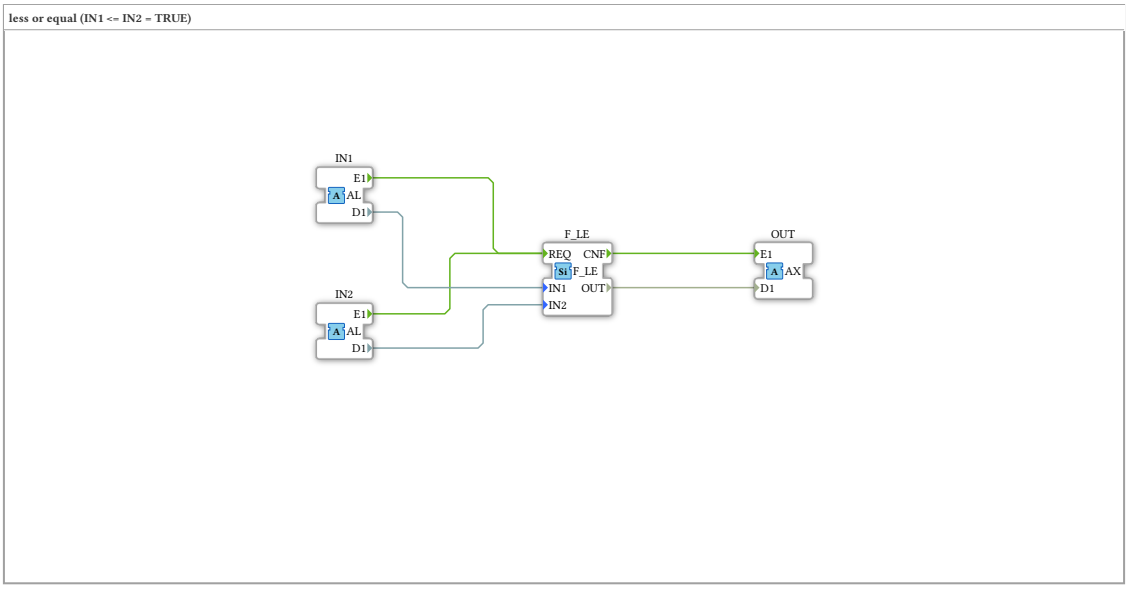
AL_GT is a simple yet important comparison block for IEC 61131-3 control applications using the LWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AL_LE



Input value 1 -- left operand (IN1 <= IN2) – *AL*
Input value 2 -- right operand (IN1 <= IN2) – *AL*

AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AL_LE** compares two values of type `LWORD` (64-bit bit pattern (long word)) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LWORD`, 64-bit bit pattern (long word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AL (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AL (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AL and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LE (type iec61131::comparison::F_LE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AL/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the LWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking extended bit masks against a reference or threshold value of type LWORD.
- **Control Logic**: Triggering an action once two values of type LWORD satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LWORD-based comparison condition.

Comparison with Similar Function Blocks

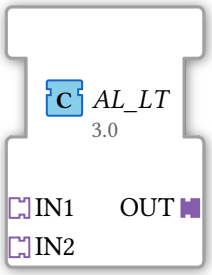
- **AL_GE**: Checks for “greater than or equal to” – the inverse function of AL_LE.
- **AL_EQ**: Checks for equality of values.
- **AL_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AL_LE is the extended version.

AL_LE completes the AL comparison family with the boundary case of equality.

Conclusion

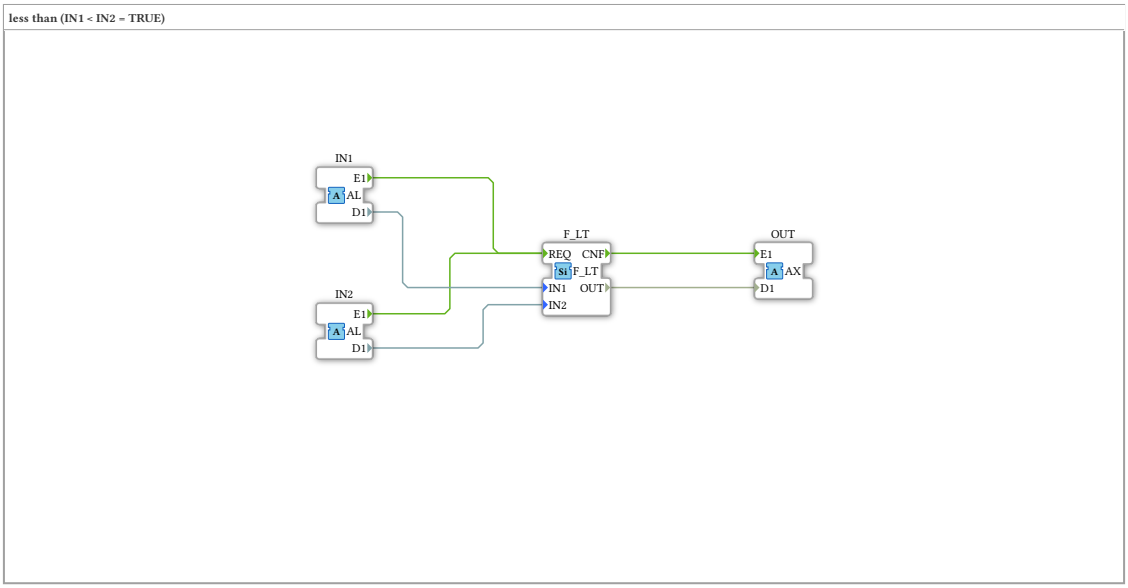
AL_LE is a simple yet important comparison block for IEC 61131-3 control applications using the LWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AL_LT



Input value 1 -- left operand (IN1 < IN2) – *AL*
Input value 2 -- right operand (IN1 < IN2) – *AL*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AL_LT** compares two values of type `LWORD` (64-bit bit pattern (long word)) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LWORD`, 64-bit bit pattern (long word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AL (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AL (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AL and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LT (type iec61131::comparison::F_LT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AL/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the LWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking extended bit masks against a reference or threshold value of type LWORD.
- **Control Logic**: Triggering an action once two values of type LWORD satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LWORD-based comparison condition.

Comparison with Similar Function Blocks

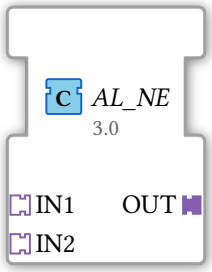
- **AL_GT**: Checks for “greater than” – the inverse function of AL_LT.
- **AL_EQ**: Checks for equality of values.
- **AL_LE**: Performs the “less than or equal to” comparison – the extended version of AL_LT that includes the equality case.

AL_LT is the basic “less than” component without the equality case, complemented by AL_LE.

Conclusion

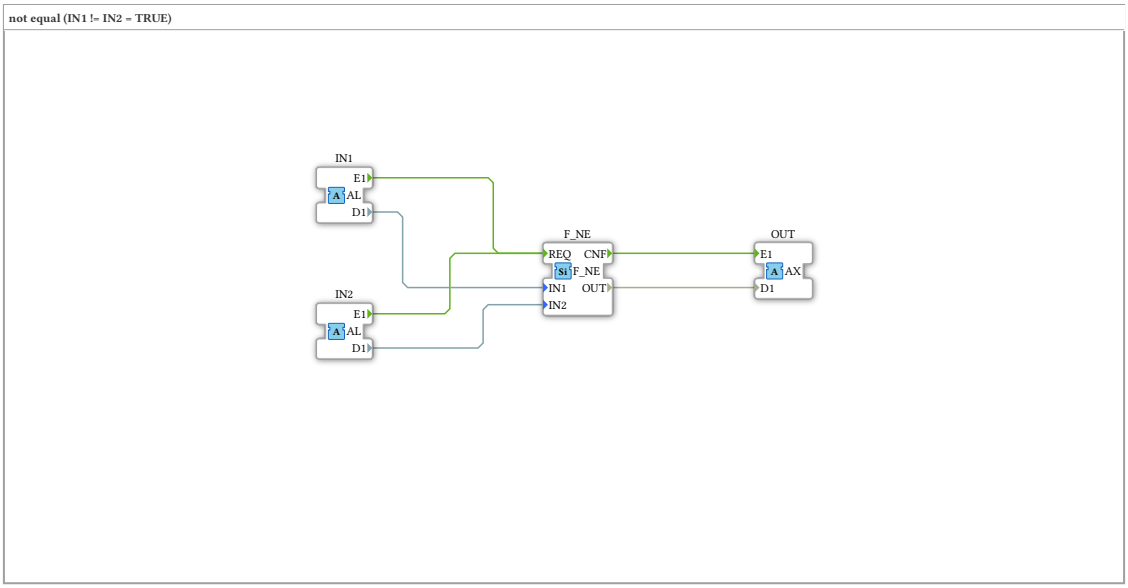
AL_LT is a simple yet important comparison block for IEC 61131-3 control applications using the LWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AL_NE



Input value 1 -- left operand (IN1 != IN2) – *AL*
Input value 2 -- right operand (IN1 != IN2) – *AL*

AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AL_NE** compares two values of type `LWORD` (64-bit bit pattern (long word)) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `LWORD`, 64-bit bit pattern (long word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AL (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AL (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AL and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_NE (type iec61131::comparison::F_NE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AL/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the LWORD data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking extended bit masks against a reference or threshold value of type LWORD.
- **Control Logic**: Triggering an action once two values of type LWORD satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a LWORD-based comparison condition.

Comparison with Similar Function Blocks

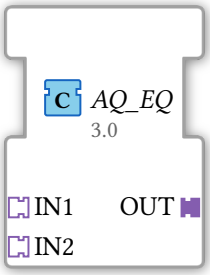
- **AL_EQ**: Checks for “equal to” – the inverse function of AL_NE.
- **AL_EQ**: Checks for equality of values.

AL_NE is part of the complete AL comparison family (EQ/NE/GT/GE/LT/LE) for the LWORD data type.

Conclusion

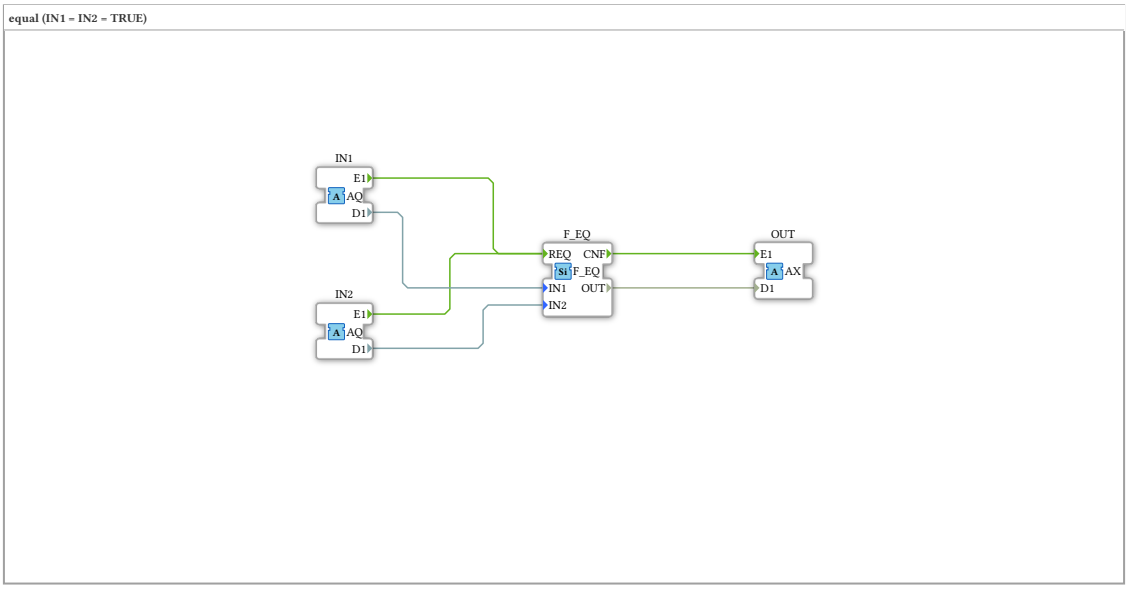
AL_NE is a simple yet important comparison block for IEC 61131-3 control applications using the LWORD data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AQ_EQ



Input value 1 -- left operand (IN1 = IN2) – *AQ*
Input value 2 -- right operand (IN1 = IN2) – *AQ*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AQ_EQ** compares two values of type `QUARTER` (2-bit value (quarter byte)) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `QUARTER`, 2-bit value (quarter byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AQ (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AQ (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AQ and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_EQ (type iec61131::comparison::F_EQ). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AQ/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the QUARTER data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking compact status fields against a reference or threshold value of type QUARTER.
- **Control Logic**: Triggering an action once two values of type QUARTER satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a QUARTER-based comparison condition.

Comparison with Similar Function Blocks

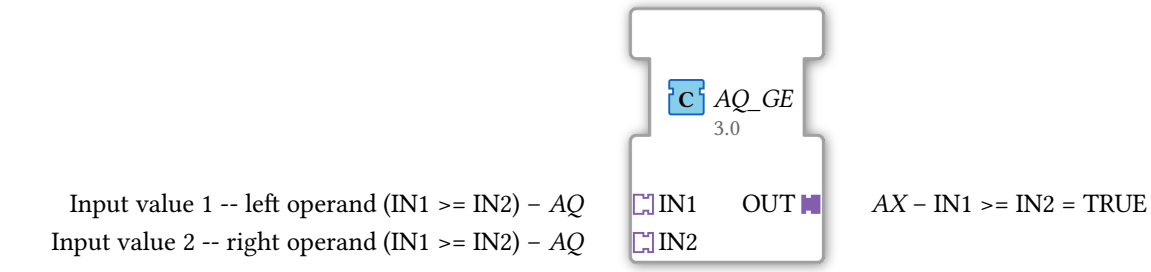
- **AQ_NE**: Checks for “not equal to” – the inverse function of AQ_EQ.
- **AQ_NE**: Checks for inequality of values.

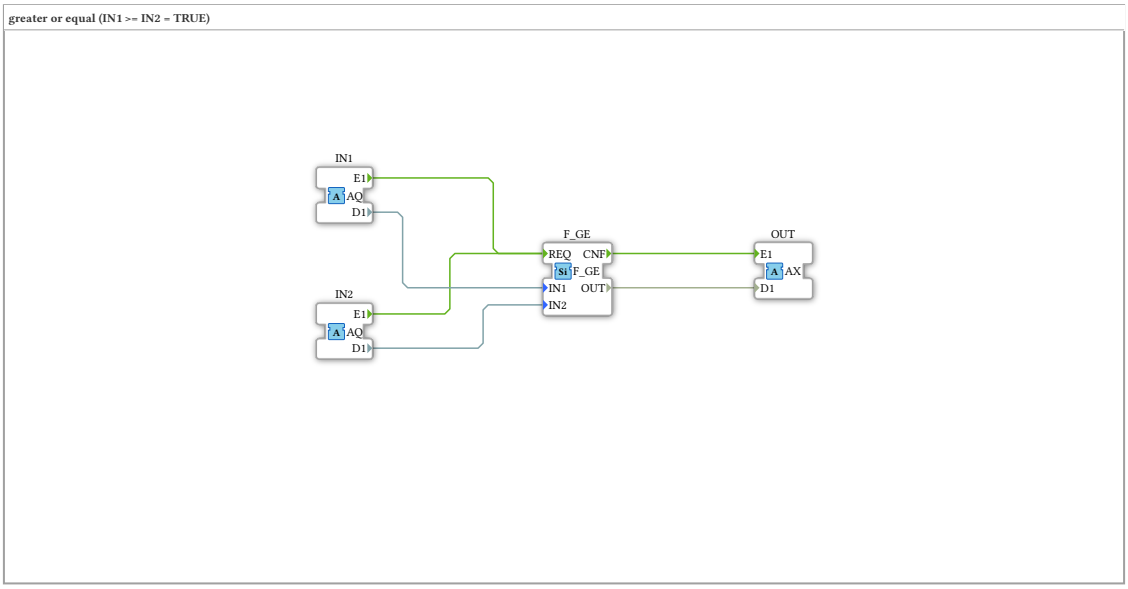
AQ_EQ is part of the complete AQ comparison family (EQ/NE/GT/GE/LT/LE) for the QUARTER data type.

Conclusion

AQ_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the QUARTER data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AQ_GE





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AQ_GE** compares two values of type `QUARTER` (2-bit value (quarter byte)) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `QUARTER`, 2-bit value (quarter byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AQ (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AQ (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AQ and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GE (type iec61131::comparison::F_GE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AQ/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 ("greater than or equal to") for the QUARTER data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking compact status fields against a reference or threshold value of type QUARTER.
- **Control Logic**: Triggering an action once two values of type QUARTER satisfy the "greater than or equal to" condition.
- **Data Filtering**: Selecting data records or signals that satisfy a QUARTER-based comparison condition.

Comparison with Similar Function Blocks

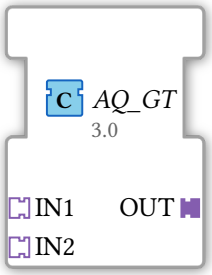
- **AQ_LE**: Checks for "less than or equal to" – the inverse function of AQ_GE.
- **AQ_EQ**: Checks for equality of values.
- **AQ_GT**: Performs the "greater than" comparison – the base variant without the equality case, of which AQ_GE is the extended version.

AQ_GE completes the AQ comparison family with the boundary case of equality.

Conclusion

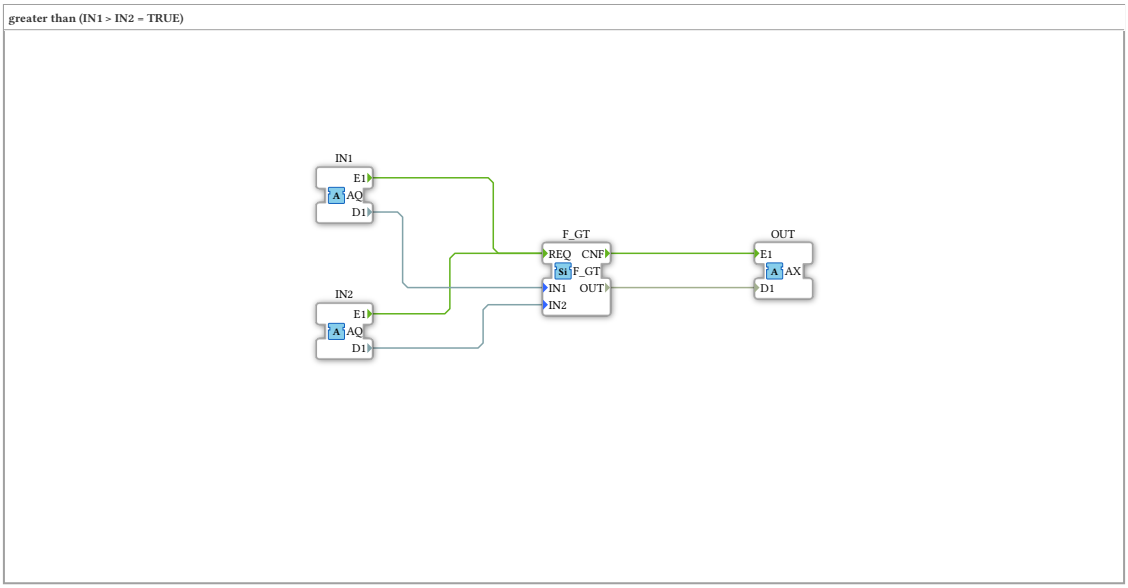
AQ_GE is a simple yet important comparison block for IEC 61131-3 control applications using the QUARTER data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AQ_GT



Input value 1 -- left operand (IN1 > IN2) – *AQ*
Input value 2 -- right operand (IN1 > IN2) – *AQ*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AQ_GT** compares two values of type `QUARTER` (2-bit value (quarter byte)) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `QUARTER`, 2-bit value (quarter byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AQ (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AQ (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AQ and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GT (type iec61131::comparison::F_GT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AQ/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 ("greater than") for the QUARTER data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking compact status fields against a reference or threshold value of type QUARTER.
- **Control Logic**: Triggering an action once two values of type QUARTER satisfy the "greater than" condition.
- **Data Filtering**: Selecting data records or signals that satisfy a QUARTER-based comparison condition.

Comparison with Similar Function Blocks

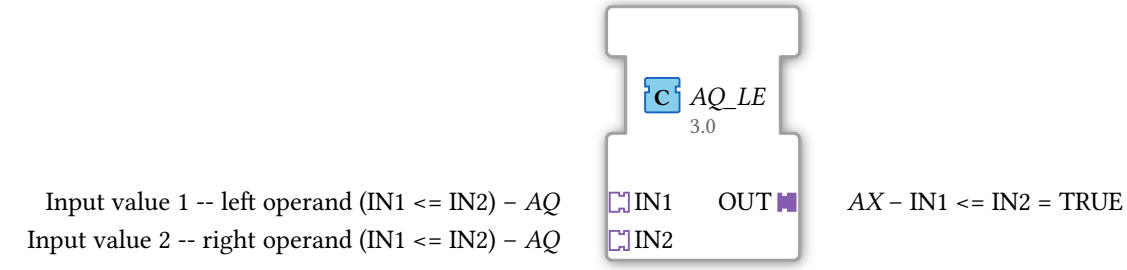
- **AQ_LT**: Checks for "less than" – the inverse function of AQ_GT.
- **AQ_EQ**: Checks for equality of values.
- **AQ_GE**: Performs the "greater than or equal to" comparison – the extended version of AQ_GT that includes the equality case.

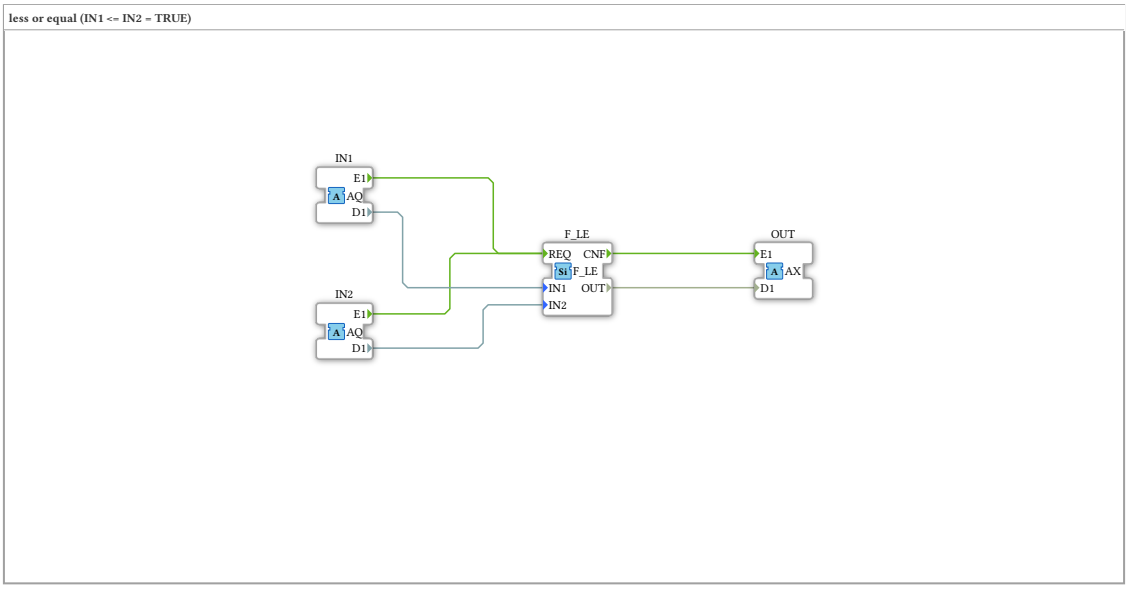
AQ_GT is the basic "greater than" component without the equality case, complemented by AQ_GE.

Conclusion

AQ_GT is a simple yet important comparison block for IEC 61131-3 control applications using the QUARTER data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AQ_LE





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AQ_LE** compares two values of type `QUARTER` (2-bit value (quarter byte)) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `QUARTER`, 2-bit value (quarter byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AQ (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AQ (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AQ and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LE (type iec61131::comparison::F_LE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AQ/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 ("less than or equal to") for the QUARTER data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking compact status fields against a reference or threshold value of type QUARTER.
- **Control Logic**: Triggering an action once two values of type QUARTER satisfy the "less than or equal to" condition.
- **Data Filtering**: Selecting data records or signals that satisfy a QUARTER-based comparison condition.

Comparison with Similar Function Blocks

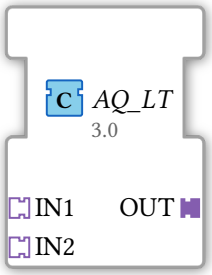
- **AQ_GE**: Checks for "greater than or equal to" – the inverse function of AQ_LE.
- **AQ_EQ**: Checks for equality of values.
- **AQ_LT**: Performs the "less than" comparison – the base variant without the equality case, of which AQ_LE is the extended version.

AQ_LE completes the AQ comparison family with the boundary case of equality.

Conclusion

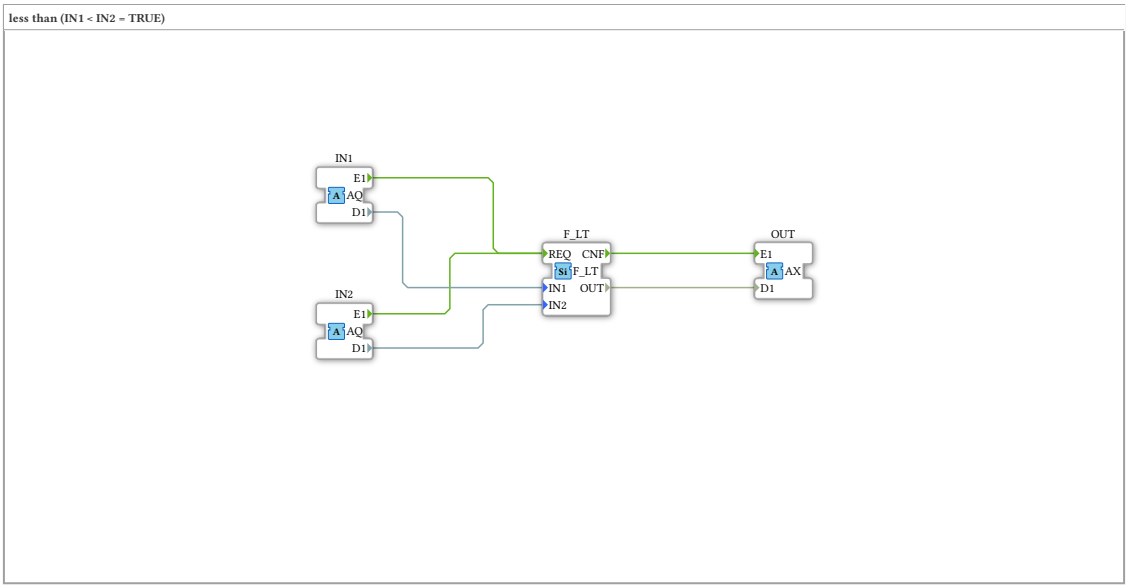
AQ_LE is a simple yet important comparison block for IEC 61131-3 control applications using the QUARTER data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AQ_LT



Input value 1 -- left operand (IN1 < IN2) – *AQ*
Input value 2 -- right operand (IN1 < IN2) – *AQ*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AQ_LT** compares two values of type `QUARTER` (2-bit value (quarter byte)) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `QUARTER`, 2-bit value (quarter byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AQ (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AQ (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AQ and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LT (type iec61131::comparison::F_LT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AQ/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the QUARTER data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking compact status fields against a reference or threshold value of type QUARTER.
- **Control Logic**: Triggering an action once two values of type QUARTER satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a QUARTER-based comparison condition.

Comparison with Similar Function Blocks

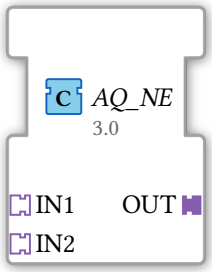
- **AQ_GT**: Checks for “greater than” – the inverse function of AQ_LT.
- **AQ_EQ**: Checks for equality of values.
- **AQ_LE**: Performs the “less than or equal to” comparison – the extended version of AQ_LT that includes the equality case.

AQ_LT is the basic “less than” component without the equality case, complemented by AQ_LE.

Conclusion

AQ_LT is a simple yet important comparison block for IEC 61131-3 control applications using the QUARTER data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

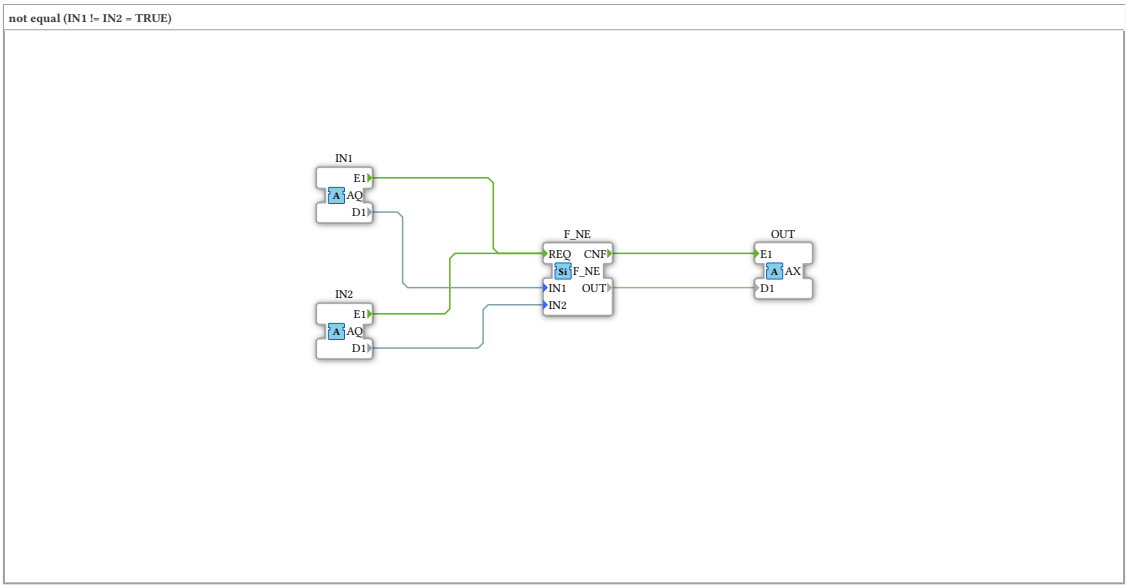
AQ_NE



Input value 1 -- left operand (IN1 != IN2) – *AQ*

Input value 2 -- right operand (IN1 != IN2) – *AQ*

AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AQ_NE** compares two values of type `QUARTER` (2-bit value (quarter byte)) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `QUARTER`, 2-bit value (quarter byte)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AQ (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AQ (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AQ and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_NE (type iec61131::comparison::F_NE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AQ/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the QUARTER data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking compact status fields against a reference or threshold value of type QUARTER.
- **Control Logic**: Triggering an action once two values of type QUARTER satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a QUARTER-based comparison condition.

Comparison with Similar Function Blocks

- **AQ_EQ**: Checks for “equal to” – the inverse function of AQ_NE.
- **AQ_EQ**: Checks for equality of values.

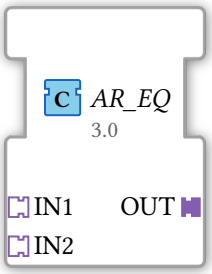
AQ_NE is part of the complete AQ comparison family (EQ/NE/GT/GE/LT/LE) for the QUARTER data type.

Conclusion

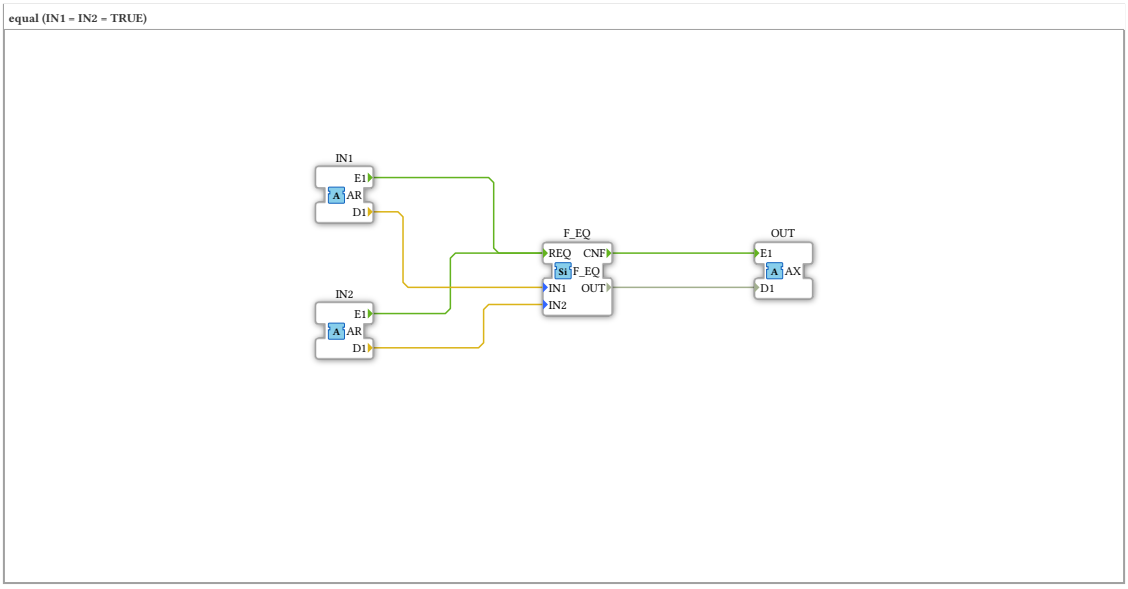
AQ_NE is a simple yet important comparison block for IEC 61131-3 control applications using the QUARTER data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AR_EQ

Input value 1 -- left operand (IN1 = IN2) – *AR*
Input value 2 -- right operand (IN1 = IN2) – *AR*



AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AR_EQ** compares two values of type `REAL` (32-bit floating-point number (single precision)) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `REAL`, 32-bit floating-point number (single precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_EQ (type iec61131::comparison::F_EQ). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the REAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking measured values against a reference or threshold value of type REAL.
- **Control Logic**: Triggering an action once two values of type REAL satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a REAL-based comparison condition.

Comparison with Similar Function Blocks

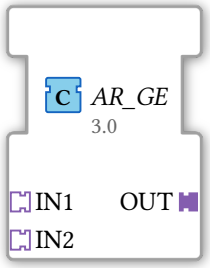
- **AR_NE**: Checks for “not equal to” – the inverse function of AR_EQ.
- **AR_NE**: Checks for inequality of values.

AR_EQ is part of the complete AR comparison family (EQ/NE/GT/GE/LT/LE) for the REAL data type.

Conclusion

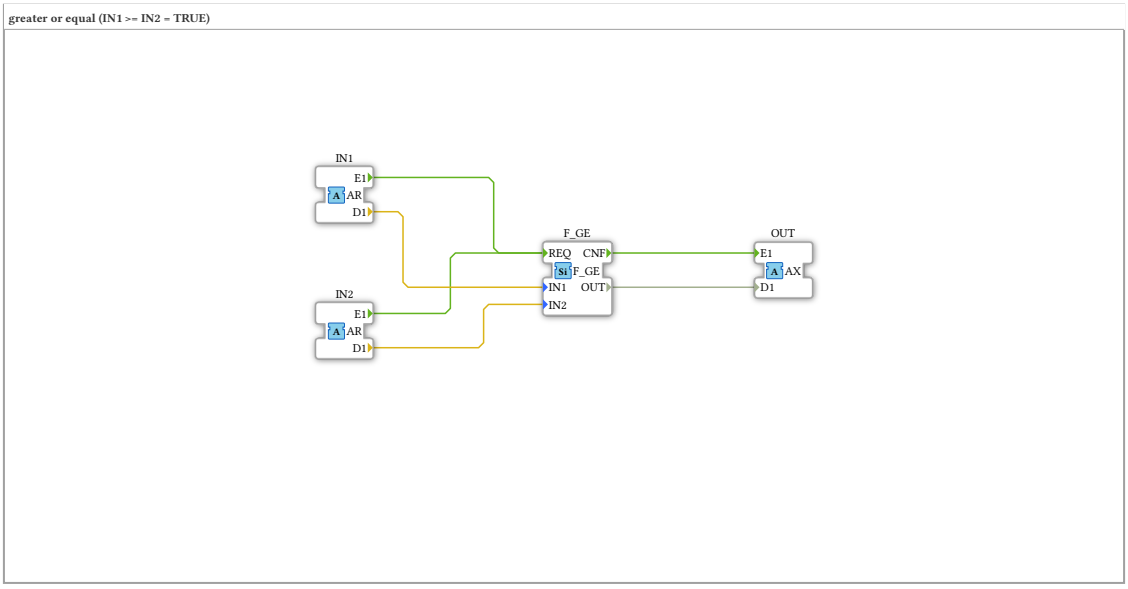
AR_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the REAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AR_GE



Input value 1 -- left operand (IN1 >= IN2) – *AR*
Input value 2 -- right operand (IN1 >= IN2) – *AR*

AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_GE** compares two values of type `REAL` (32-bit floating-point number (single precision)) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `REAL`, 32-bit floating-point number (single precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GE (type iec61131::comparison::F_GE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the REAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking measured values against a reference or threshold value of type REAL.
- **Control Logic**: Triggering an action once two values of type REAL satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a REAL-based comparison condition.

Comparison with Similar Function Blocks

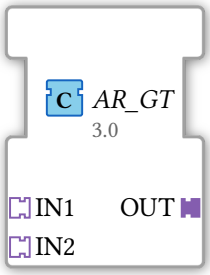
- **AR_LE**: Checks for “less than or equal to” – the inverse function of AR_GE.
- **AR_EQ**: Checks for equality of values.
- **AR_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AR_GE is the extended version.

AR_GE completes the AR comparison family with the boundary case of equality.

Conclusion

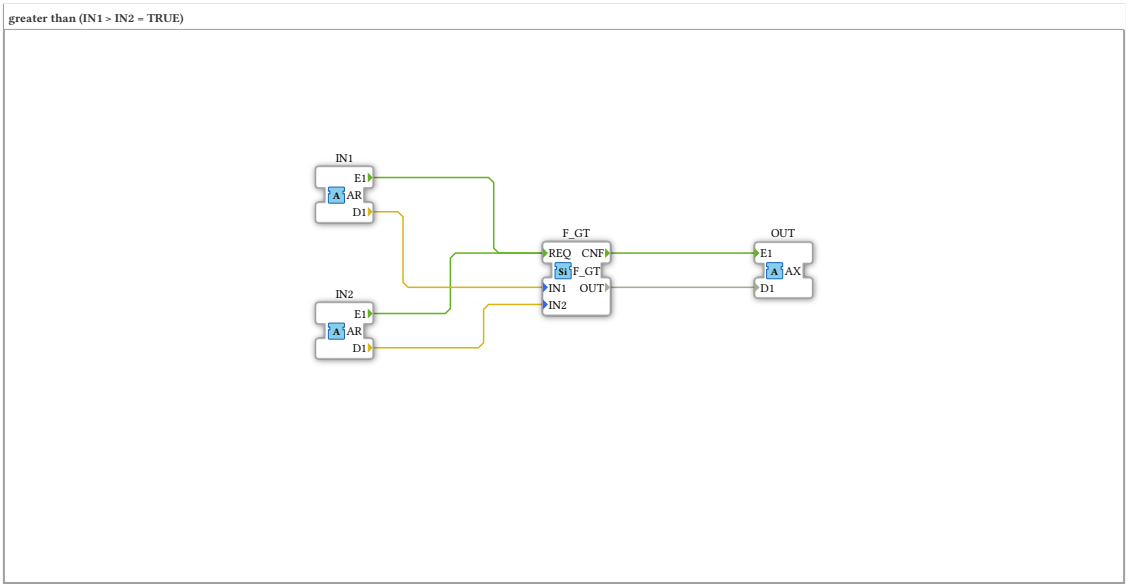
AR_GE is a simple yet important comparison block for IEC 61131-3 control applications using the REAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AR_GT



Input value 1 -- left operand (IN1 > IN2) – *AR*
Input value 2 -- right operand (IN1 > IN2) – *AR*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AR_GT** compares two values of type `REAL` (32-bit floating-point number (single precision)) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `REAL`, 32-bit floating-point number (single precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GT (type iec61131::comparison::F_GT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the REAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking measured values against a reference or threshold value of type REAL.
- **Control Logic**: Triggering an action once two values of type REAL satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a REAL-based comparison condition.

Comparison with Similar Function Blocks

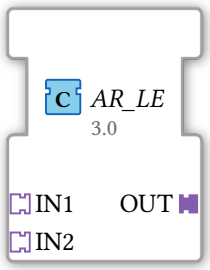
- **AR_LT**: Checks for “less than” – the inverse function of AR_GT.
- **AR_EQ**: Checks for equality of values.
- **AR_GE**: Performs the “greater than or equal to” comparison – the extended version of AR_GT that includes the equality case.

AR_GT is the basic “greater than” component without the equality case, complemented by AR_GE.

Conclusion

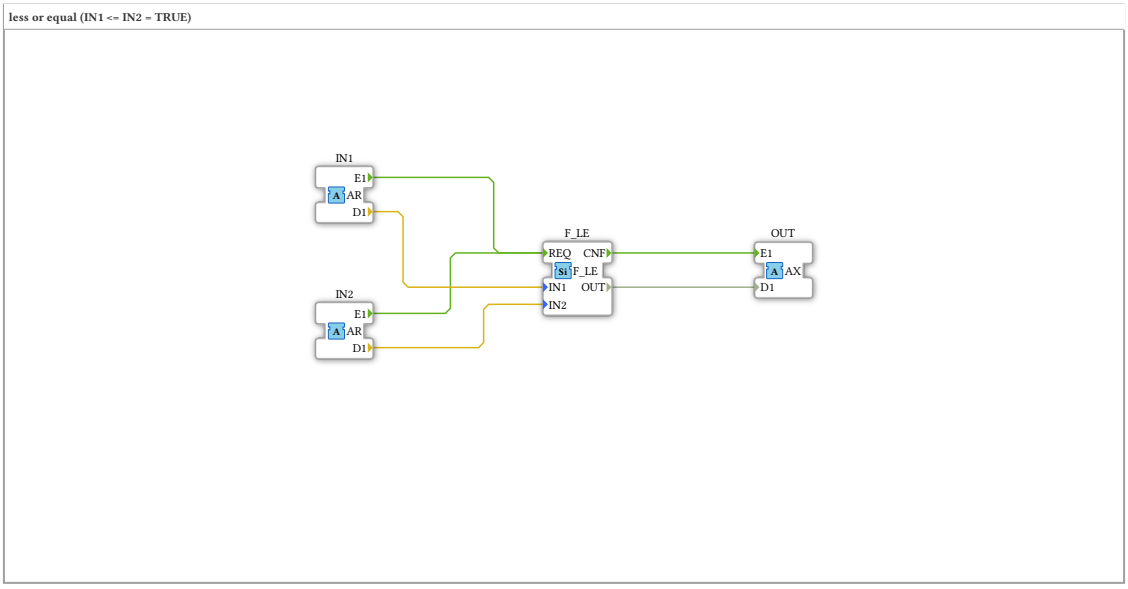
AR_GT is a simple yet important comparison block for IEC 61131-3 control applications using the REAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AR_LE



Input value 1 -- left operand (IN1 <= IN2) – *AR*
Input value 2 -- right operand (IN1 <= IN2) – *AR*

AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AR_LE** compares two values of type `REAL` (32-bit floating-point number (single precision)) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `REAL`, 32-bit floating-point number (single precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LE (type iec61131::comparison::F_LE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the REAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking measured values against a reference or threshold value of type REAL.
- **Control Logic**: Triggering an action once two values of type REAL satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a REAL-based comparison condition.

Comparison with Similar Function Blocks

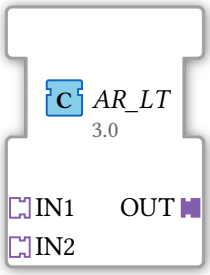
- **AR_GE**: Checks for “greater than or equal to” – the inverse function of AR_LE.
- **AR_EQ**: Checks for equality of values.
- **AR_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AR_LE is the extended version.

AR_LE completes the AR comparison family with the boundary case of equality.

Conclusion

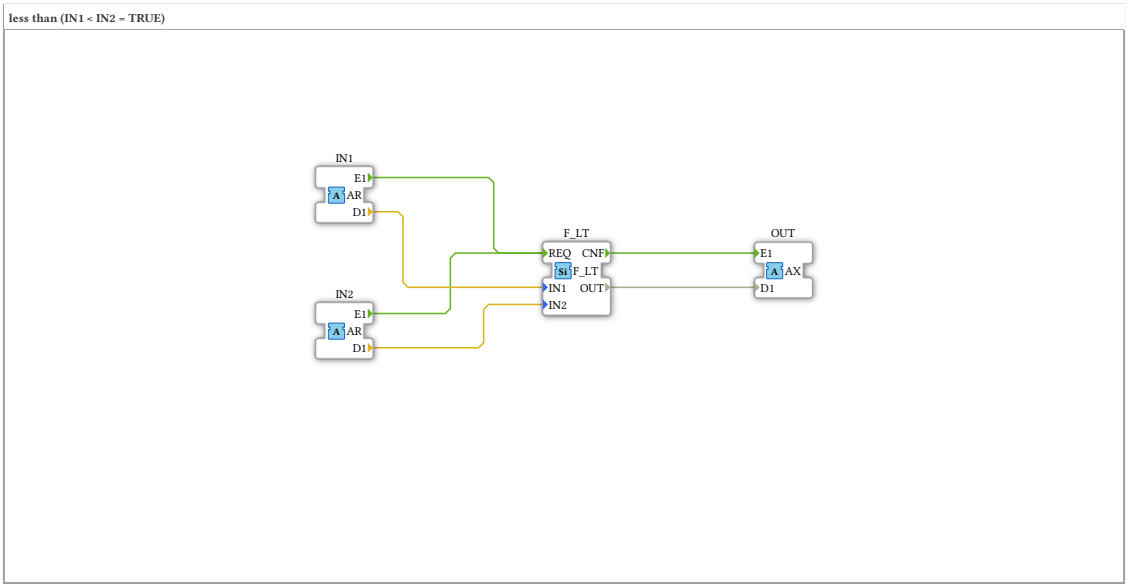
AR_LE is a simple yet important comparison block for IEC 61131-3 control applications using the REAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AR_LT



Input value 1 -- left operand (IN1 < IN2) – *AR*
Input value 2 -- right operand (IN1 < IN2) – *AR*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AR_LT** compares two values of type `REAL` (32-bit floating-point number (single precision)) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `REAL`, 32-bit floating-point number (single precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LT (type iec61131::comparison::F_LT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the REAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking measured values against a reference or threshold value of type REAL.
- **Control Logic**: Triggering an action once two values of type REAL satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a REAL-based comparison condition.

Comparison with Similar Function Blocks

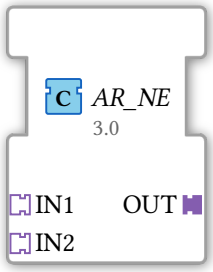
- **AR_GT**: Checks for “greater than” – the inverse function of AR_LT.
- **AR_EQ**: Checks for equality of values.
- **AR_LE**: Performs the “less than or equal to” comparison – the extended version of AR_LT that includes the equality case.

AR_LT is the basic “less than” component without the equality case, complemented by AR_LE.

Conclusion

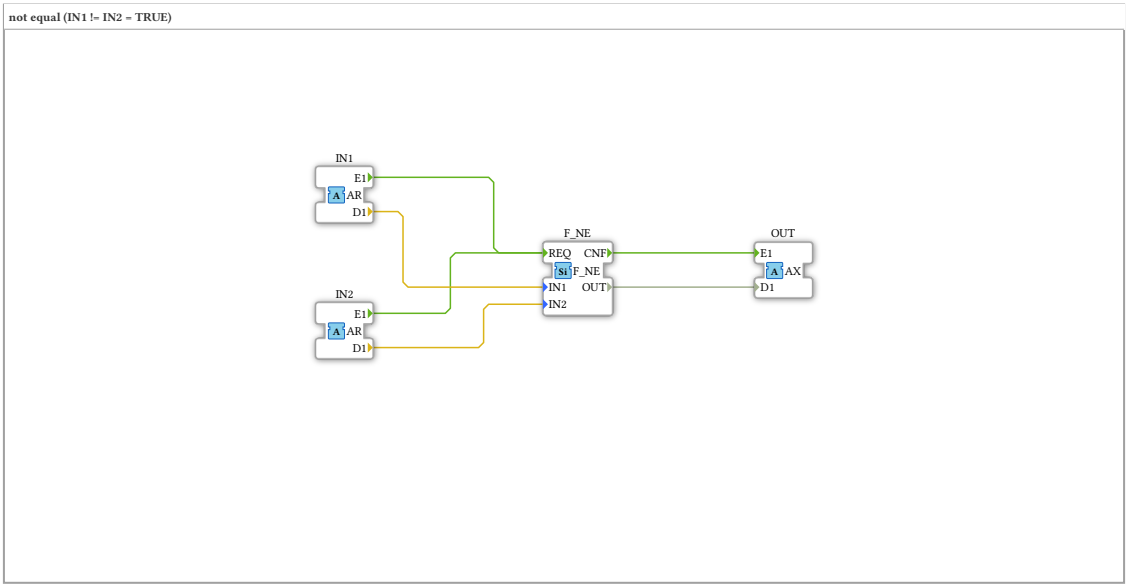
AR_LT is a simple yet important comparison block for IEC 61131-3 control applications using the REAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AR_NE



Input value 1 -- left operand (IN1 != IN2) – *AR*
Input value 2 -- right operand (IN1 != IN2) – *AR*

AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AR_NE** compares two values of type `REAL` (32-bit floating-point number (single precision)) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `REAL`, 32-bit floating-point number (single precision)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AR (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AR (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AR and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_NE (type iec61131::comparison::F_NE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AR/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the REAL data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking measured values against a reference or threshold value of type REAL.
- **Control Logic**: Triggering an action once two values of type REAL satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a REAL-based comparison condition.

Comparison with Similar Function Blocks

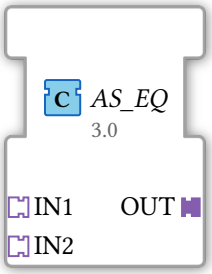
- **AR_EQ**: Checks for “equal to” – the inverse function of AR_NE.
- **AR_EQ**: Checks for equality of values.

AR_NE is part of the complete AR comparison family (EQ/NE/GT/GE/LT/LE) for the REAL data type.

Conclusion

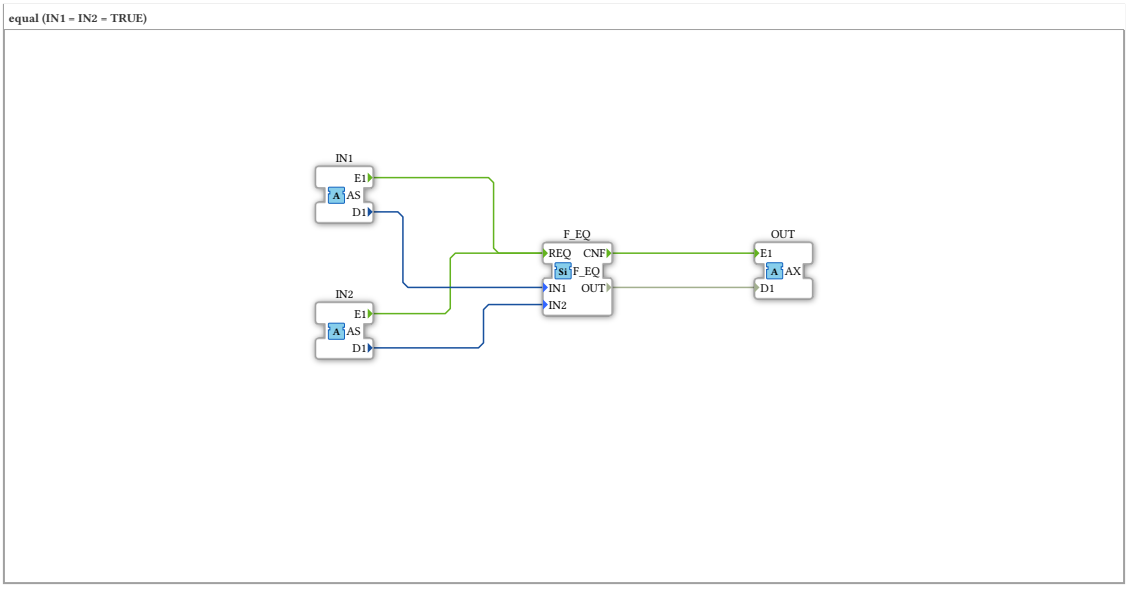
AR_NE is a simple yet important comparison block for IEC 61131-3 control applications using the REAL data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AS_EQ



Input value 1 -- left operand (IN1 = IN2) – *AS*
Input value 2 -- right operand (IN1 = IN2) – *AS*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AS_EQ** compares two values of type `SINT` (8-bit signed integer) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `SINT`, 8-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AS (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AS (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AS and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_EQ (type iec61131::comparison::F_EQ). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AS/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the SINT data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type SINT.
- **Control Logic**: Triggering an action once two values of type SINT satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a SINT-based comparison condition.

Comparison with Similar Function Blocks

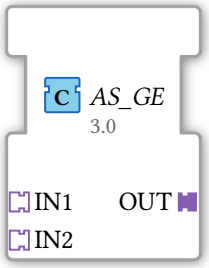
- **AS_NE**: Checks for “not equal to” – the inverse function of AS_EQ.
- **AS_NE**: Checks for inequality of values.

AS_EQ is part of the complete AS comparison family (EQ/NE/GT/GE/LT/LE) for the SINT data type.

Conclusion

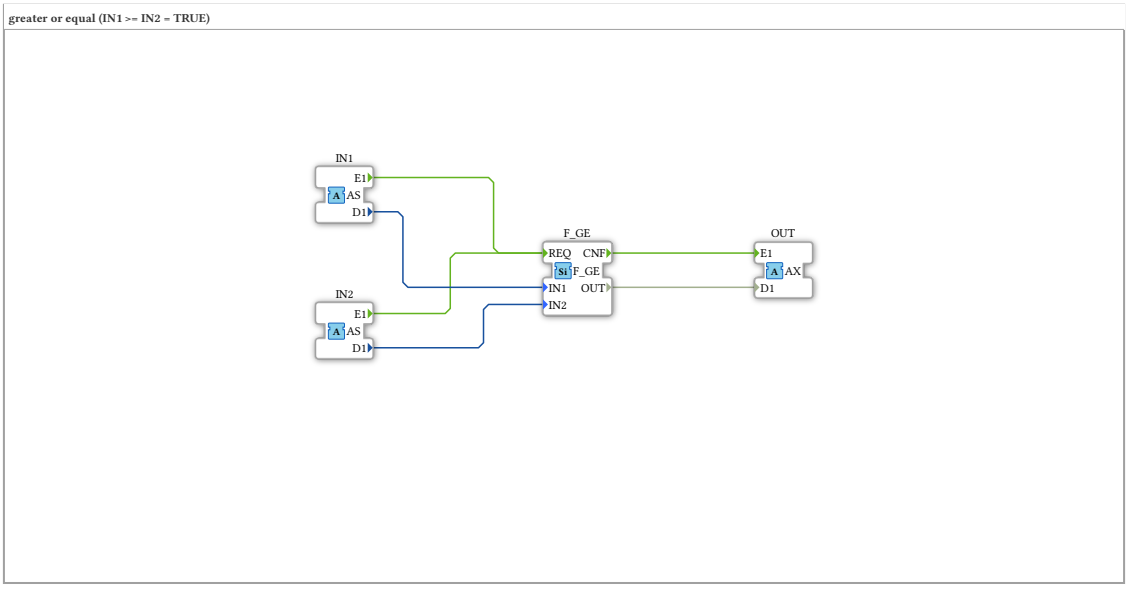
AS_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the SINT data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AS_GE



Input value 1 -- left operand (IN1 >= IN2) – *AS*
Input value 2 -- right operand (IN1 >= IN2) – *AS*

AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_GE** compares two values of type `SINT` (8-bit signed integer) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `SINT`, 8-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AS (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AS (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AS and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GE (type iec61131::comparison::F_GE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AS/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 ("greater than or equal to") for the SINT data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type SINT.
- **Control Logic**: Triggering an action once two values of type SINT satisfy the "greater than or equal to" condition.
- **Data Filtering**: Selecting data records or signals that satisfy a SINT-based comparison condition.

Comparison with Similar Function Blocks

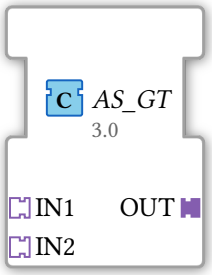
- **AS_LE**: Checks for "less than or equal to" – the inverse function of AS_GE.
- **AS_EQ**: Checks for equality of values.
- **AS_GT**: Performs the "greater than" comparison – the base variant without the equality case, of which AS_GE is the extended version.

AS_GE completes the AS comparison family with the boundary case of equality.

Conclusion

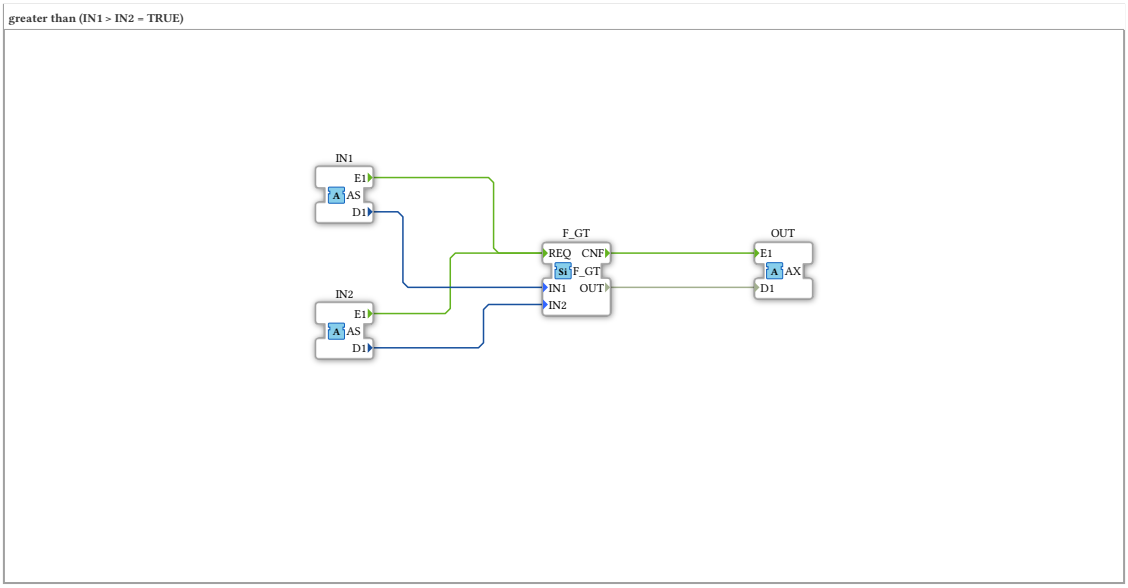
AS_GE is a simple yet important comparison block for IEC 61131-3 control applications using the SINT data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AS_GT



Input value 1 -- left operand (IN1 > IN2) – *AS*
Input value 2 -- right operand (IN1 > IN2) – *AS*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AS_GT** compares two values of type `SINT` (8-bit signed integer) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `SINT`, 8-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AS (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AS (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AS and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GT (type iec61131::comparison::F_GT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AS/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the SINT data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type SINT.
- **Control Logic**: Triggering an action once two values of type SINT satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a SINT-based comparison condition.

Comparison with Similar Function Blocks

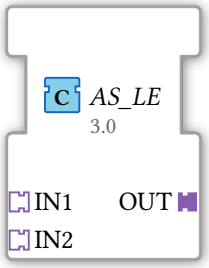
- **AS_LT**: Checks for “less than” – the inverse function of AS_GT.
- **AS_EQ**: Checks for equality of values.
- **AS_GE**: Performs the “greater than or equal to” comparison – the extended version of AS_GT that includes the equality case.

AS_GT is the basic “greater than” component without the equality case, complemented by AS_GE.

Conclusion

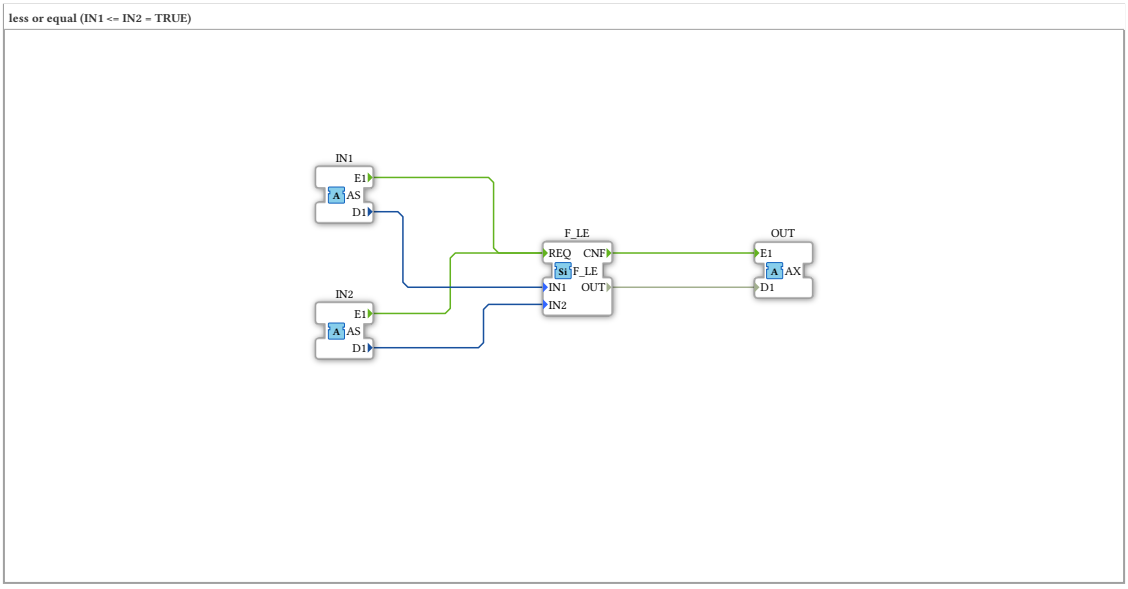
AS_GT is a simple yet important comparison block for IEC 61131-3 control applications using the SINT data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AS_LE



Input value 1 -- left operand (IN1 <= IN2) – *AS*
Input value 2 -- right operand (IN1 <= IN2) – *AS*

AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_LE** compares two values of type `SINT` (8-bit signed integer) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `SINT`, 8-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AS (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AS (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AS and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LE (type iec61131::comparison::F_LE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AS/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 ("less than or equal to") for the SINT data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type SINT.
- **Control Logic**: Triggering an action once two values of type SINT satisfy the "less than or equal to" condition.
- **Data Filtering**: Selecting data records or signals that satisfy a SINT-based comparison condition.

Comparison with Similar Function Blocks

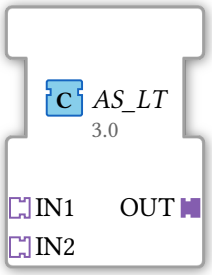
- **AS_GE**: Checks for "greater than or equal to" – the inverse function of AS_LE.
- **AS_EQ**: Checks for equality of values.
- **AS_LT**: Performs the "less than" comparison – the base variant without the equality case, of which AS_LE is the extended version.

AS_LE completes the AS comparison family with the boundary case of equality.

Conclusion

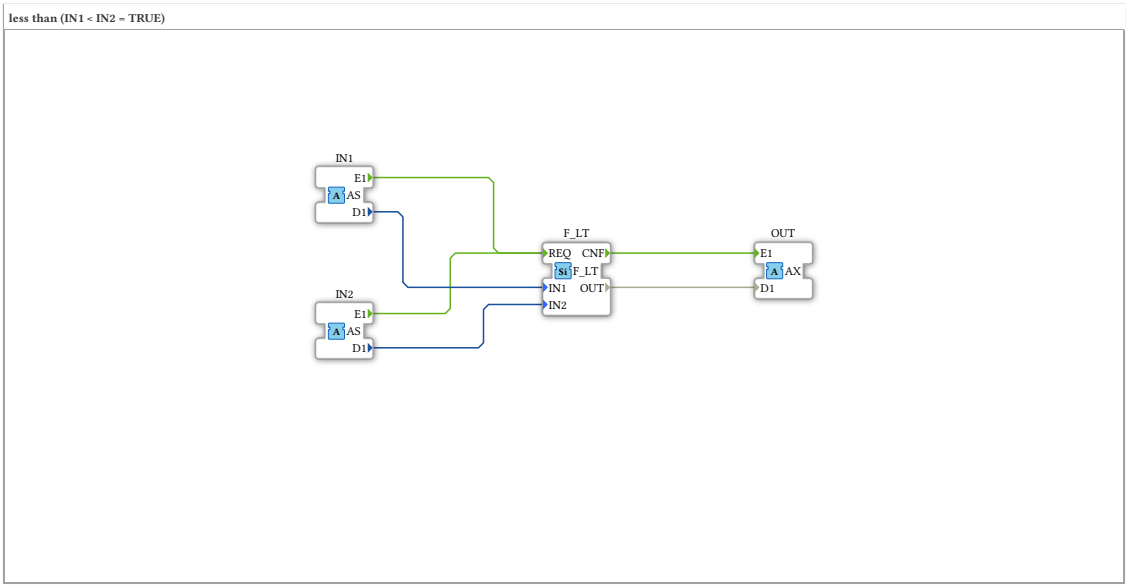
AS_LE is a simple yet important comparison block for IEC 61131-3 control applications using the SINT data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AS_LT



Input value 1 -- left operand (IN1 < IN2) – *AS*
Input value 2 -- right operand (IN1 < IN2) – *AS*

AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AS_LT** compares two values of type `SINT` (8-bit signed integer) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `SINT`, 8-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AS (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AS (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AS and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LT (type iec61131::comparison::F_LT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AS/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the SINT data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type SINT.
- **Control Logic**: Triggering an action once two values of type SINT satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a SINT-based comparison condition.

Comparison with Similar Function Blocks

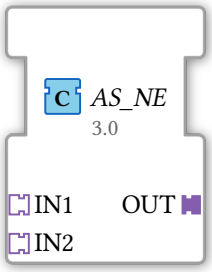
- **AS_GT**: Checks for “greater than” – the inverse function of AS_LT.
- **AS_EQ**: Checks for equality of values.
- **AS_LE**: Performs the “less than or equal to” comparison – the extended version of AS_LT that includes the equality case.

AS_LT is the basic “less than” component without the equality case, complemented by AS_LE.

Conclusion

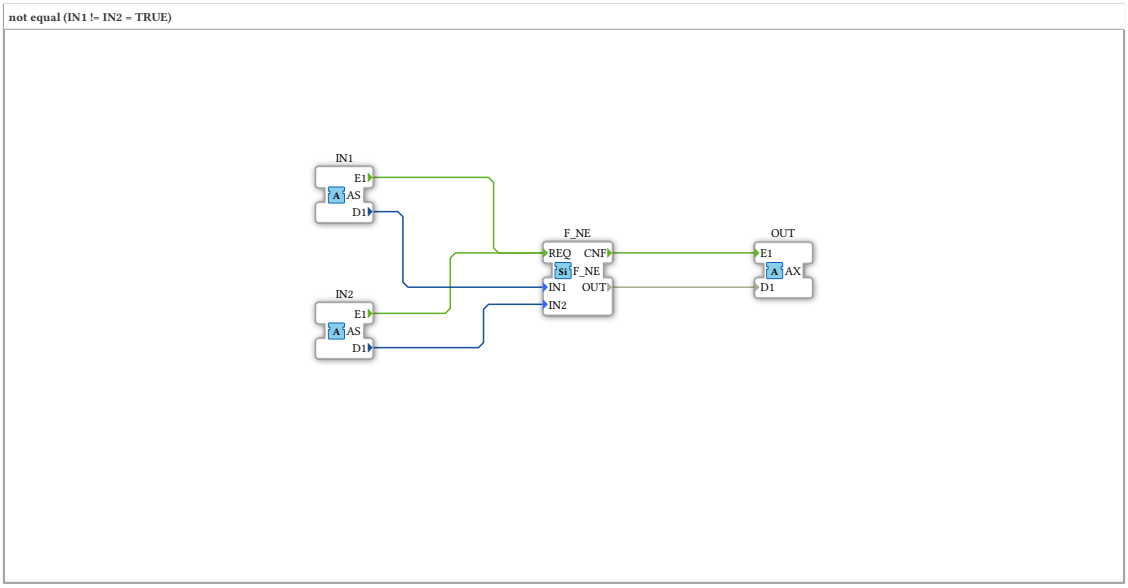
AS_LT is a simple yet important comparison block for IEC 61131-3 control applications using the SINT data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AS_NE



Input value 1 -- left operand (IN1 != IN2) – *AS*
Input value 2 -- right operand (IN1 != IN2) – *AS*

AX – IN1 != IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AS_NE** compares two values of type `SINT` (8-bit signed integer) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `SINT`, 8-bit signed integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	AS (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	AS (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters AS and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_NE (type iec61131::comparison::F_NE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (AS/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the SINT data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type SINT.
- **Control Logic**: Triggering an action once two values of type SINT satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a SINT-based comparison condition.

Comparison with Similar Function Blocks

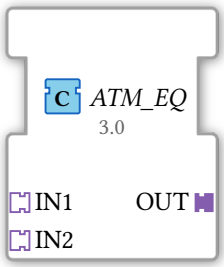
- **AS_EQ**: Checks for “equal to” – the inverse function of AS_NE.
- **AS_EQ**: Checks for equality of values.

AS_NE is part of the complete AS comparison family (EQ/NE/GT/GE/LT/LE) for the SINT data type.

Conclusion

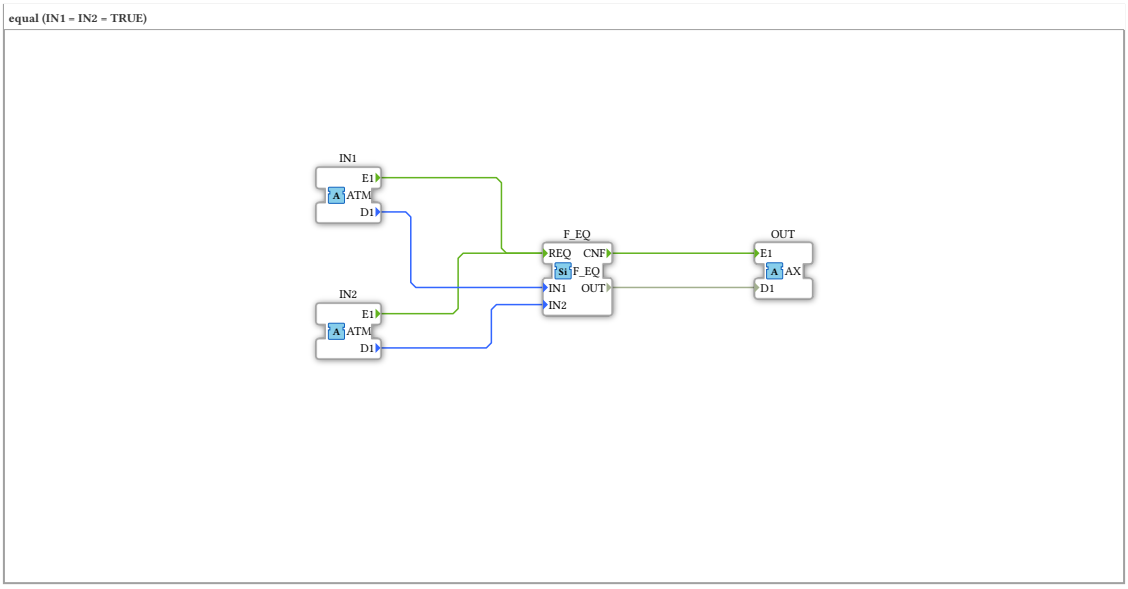
AS_NE is a simple yet important comparison block for IEC 61131-3 control applications using the SINT data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ATM_EQ



Input value 1 -- left operand (IN1 = IN2) – *ATM*
Input value 2 -- right operand (IN1 = IN2) – *ATM*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_EQ** compares two values of type `TIME` (duration) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The duration of the two time values is compared.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `TIME`, duration).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ATM (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ATM (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ATM and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_EQ (type iec61131::comparison::F_EQ). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ATM/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the TIME data type.
- **Comparison logic**: The duration of the two time values is compared.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking timing and delay values against a reference or threshold value of type TIME.
- **Control Logic**: Triggering an action once two values of type TIME satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a TIME-based comparison condition.

Comparison with Similar Function Blocks

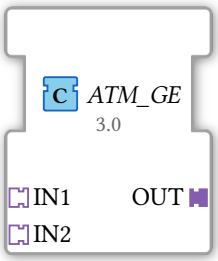
- **ATM_NE**: Checks for “not equal to” – the inverse function of ATM_EQ.
- **ATM_NE**: Checks for inequality of values.

ATM_EQ is part of the complete ATM comparison family (EQ/NE/GT/GE/LT/LE) for the TIME data type.

Conclusion

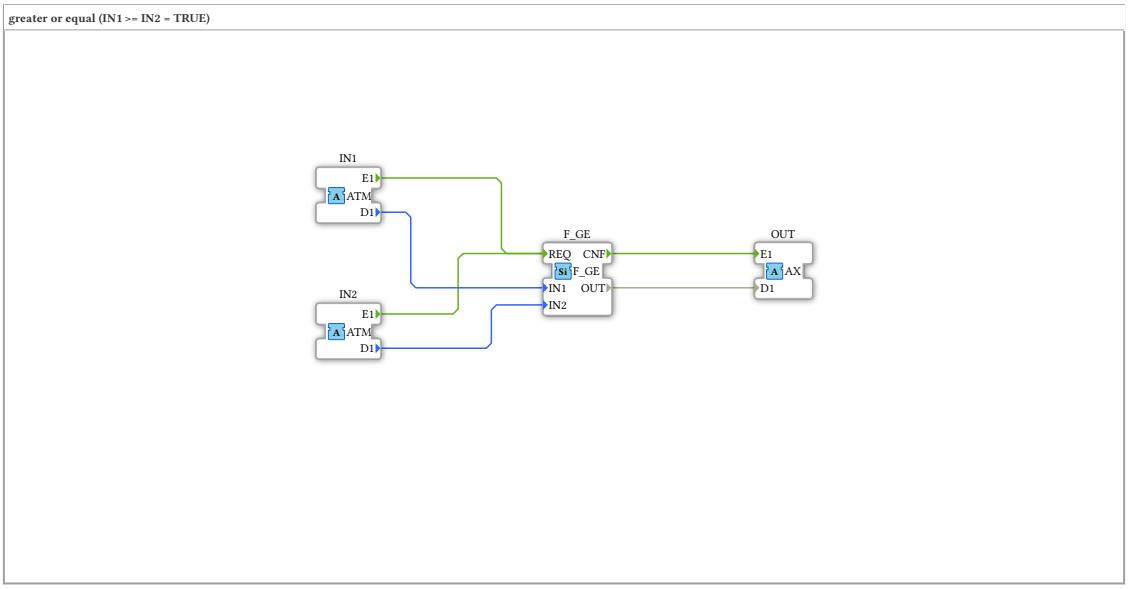
ATM_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the TIME data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ATM_GE



Input value 1 -- left operand (IN1 >= IN2) – *ATM*
Input value 2 -- right operand (IN1 >= IN2) – *ATM*

AX – IN1 >= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ATM_GE** compares two values of type `TIME` (duration) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The duration of the two time values is compared.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `TIME`, duration).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ATM (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ATM (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ATM and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GE (type iec61131::comparison::F_GE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ATM/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the TIME data type.
- **Comparison logic**: The duration of the two time values is compared.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking timing and delay values against a reference or threshold value of type TIME.
- **Control Logic**: Triggering an action once two values of type TIME satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a TIME-based comparison condition.

Comparison with Similar Function Blocks

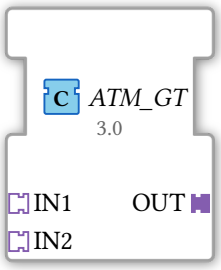
- **ATM_LE**: Checks for “less than or equal to” – the inverse function of ATM_GE.
- **ATM_EQ**: Checks for equality of values.
- **ATM_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which ATM_GE is the extended version.

ATM_GE completes the ATM comparison family with the boundary case of equality.

Conclusion

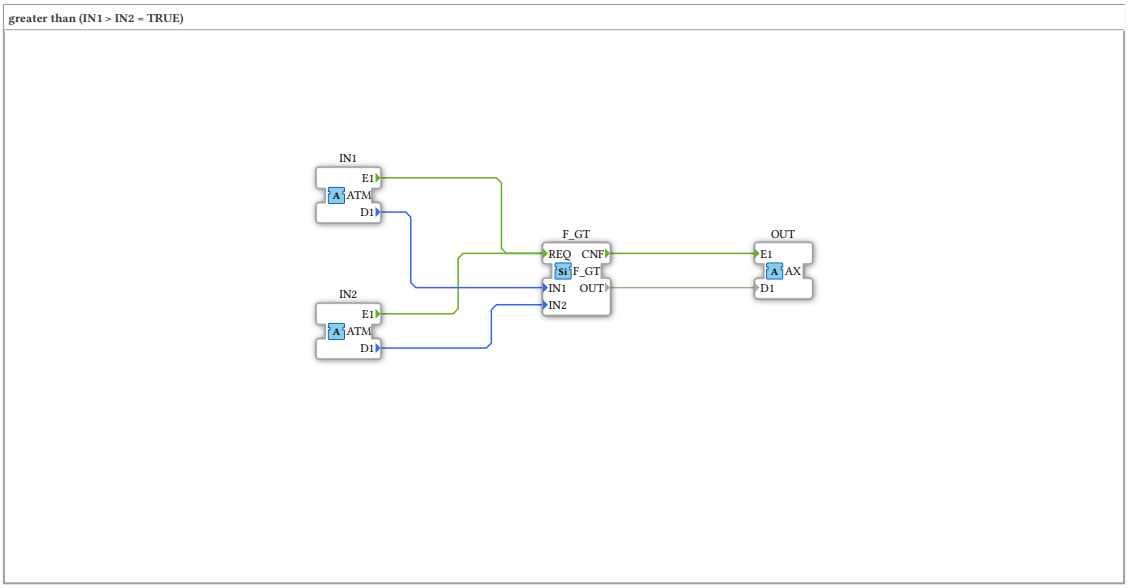
ATM_GE is a simple yet important comparison block for IEC 61131-3 control applications using the TIME data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ATM_GT



Input value 1 -- left operand (IN1 > IN2) – *ATM*
Input value 2 -- right operand (IN1 > IN2) – *ATM*

AX – IN1 > IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_GT** compares two values of type `TIME` (duration) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The duration of the two time values is compared.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `TIME`, duration).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ATM (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ATM (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ATM and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_GT (type iec61131::comparison::F_GT). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ATM/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the TIME data type.
- **Comparison logic**: The duration of the two time values is compared.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking timing and delay values against a reference or threshold value of type TIME.
- **Control Logic**: Triggering an action once two values of type TIME satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a TIME-based comparison condition.

Comparison with Similar Function Blocks

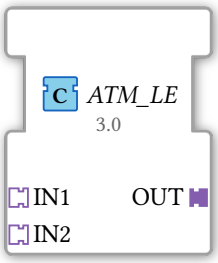
- **ATM_LT**: Checks for “less than” – the inverse function of ATM_GT.
- **ATM_EQ**: Checks for equality of values.
- **ATM_GE**: Performs the “greater than or equal to” comparison – the extended version of ATM_GT that includes the equality case.

ATM_GT is the basic “greater than” component without the equality case, complemented by ATM_GE.

Conclusion

ATM_GT is a simple yet important comparison block for IEC 61131-3 control applications using the TIME data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

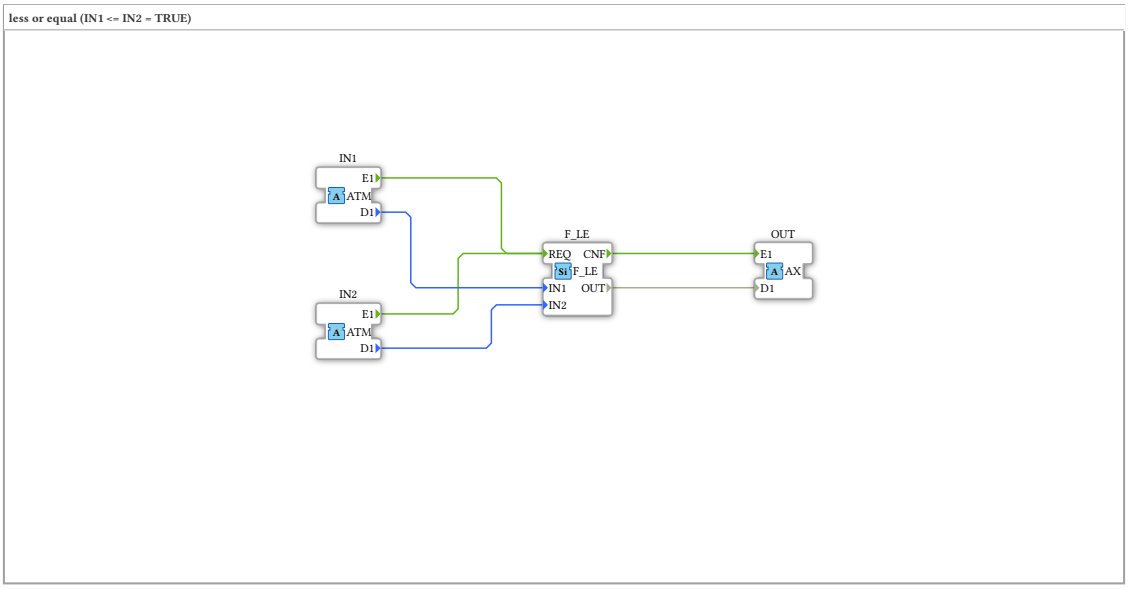
ATM_LE



Input value 1 -- left operand (IN1 <= IN2) – *ATM*

Input value 2 -- right operand (IN1 <= IN2) – *ATM*

AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_LE** compares two values of type `TIME` (duration) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The duration of the two time values is compared.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `TIME`, duration).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ATM (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ATM (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ATM and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_LE (type iec61131::comparison::F_LE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ATM/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the TIME data type.
- **Comparison logic**: The duration of the two time values is compared.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking timing and delay values against a reference or threshold value of type TIME.
- **Control Logic**: Triggering an action once two values of type TIME satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a TIME-based comparison condition.

Comparison with Similar Function Blocks

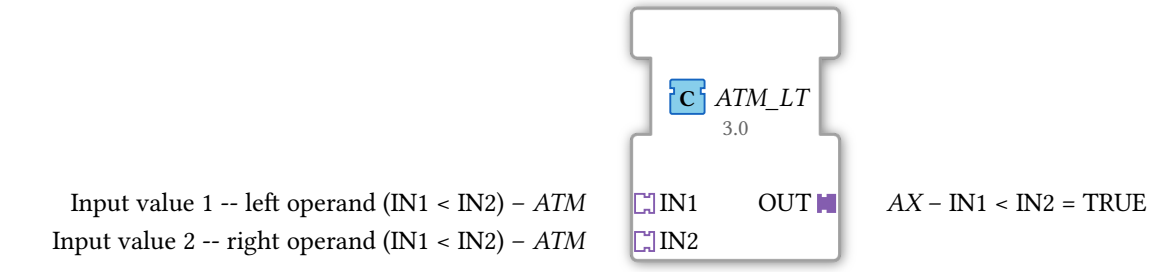
- **ATM_GE**: Checks for “greater than or equal to” – the inverse function of ATM_LE.
- **ATM_EQ**: Checks for equality of values.
- **ATM_LT**: Performs the “less than” comparison – the base variant without the equality case, of which ATM_LE is the extended version.

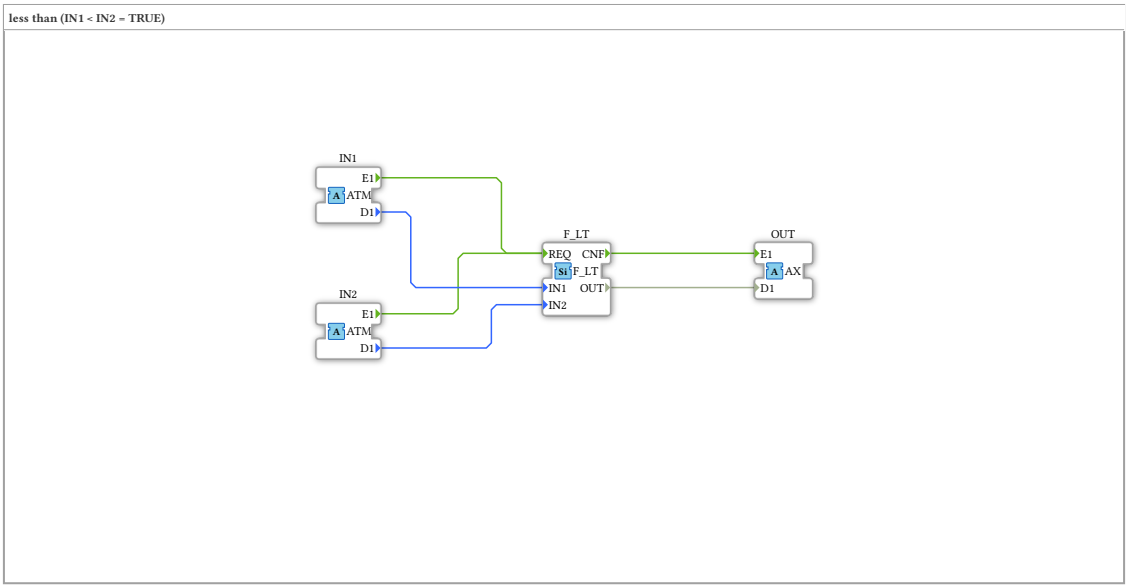
ATM_LE completes the ATM comparison family with the boundary case of equality.

Conclusion

ATM_LE is a simple yet important comparison block for IEC 61131-3 control applications using the TIME data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ATM_LT





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_LT** compares two values of type `TIME` (duration) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The duration of the two time values is compared.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `TIME`, duration).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`ATM` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`ATM` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `ATM` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LT` (type `iec61131::comparison::F_LT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`ATM`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the `TIME` data type.
- **Comparison logic**: The duration of the two time values is compared.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking timing and delay values against a reference or threshold value of type `TIME`.
- **Control Logic**: Triggering an action once two values of type `TIME` satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `TIME`-based comparison condition.

Comparison with Similar Function Blocks

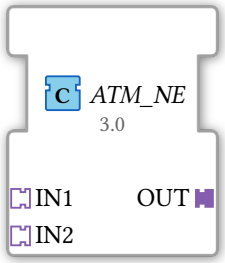
- **ATM_GT**: Checks for “greater than” – the inverse function of ATM_LT.
- **ATM_EQ**: Checks for equality of values.
- **ATM_LE**: Performs the “less than or equal to” comparison – the extended version of ATM_LT that includes the equality case.

ATM_LT is the basic “less than” component without the equality case, complemented by ATM_LE.

Conclusion

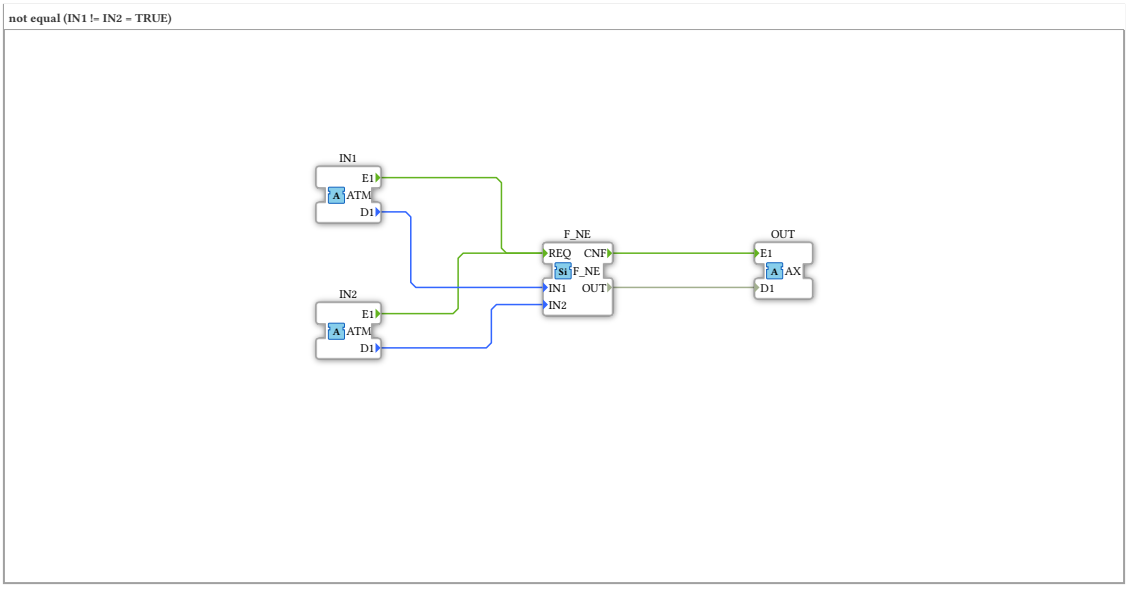
ATM_LT is a simple yet important comparison block for IEC 61131-3 control applications using the `TIME` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

ATM_NE



Input value 1 -- left operand (IN1 != IN2) – *ATM*
Input value 2 -- right operand (IN1 != IN2) – *ATM*

AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_NE** compares two values of type `TIME` (duration) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The duration of the two time values is compared.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `TIME`, duration).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	ATM (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	ATM (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	AX (Plug)	Output	Adapter for the comparison result and the output event

The adapters ATM and AX are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block F_NE (type iec61131::comparison::F_NE). There, the current values of IN1.D1 and IN2.D1 are compared.
3. The internal function block generates an output event (CNF) and outputs the Boolean result (OUT).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value OUT.D1 is set to TRUE or FALSE.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (ATM/AX), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the TIME data type.
- **Comparison logic**: The duration of the two time values is compared.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking timing and delay values against a reference or threshold value of type TIME.
- **Control Logic**: Triggering an action once two values of type TIME satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a TIME-based comparison condition.

Comparison with Similar Function Blocks

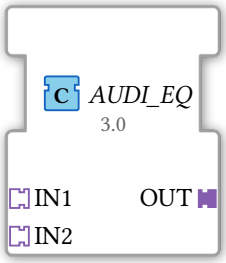
- **ATM_EQ**: Checks for “equal to” – the inverse function of ATM_NE.
- **ATM_EQ**: Checks for equality of values.

ATM_NE is part of the complete ATM comparison family (EQ/NE/GT/GE/LT/LE) for the TIME data type.

Conclusion

ATM_NE is a simple yet important comparison block for IEC 61131-3 control applications using the TIME data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

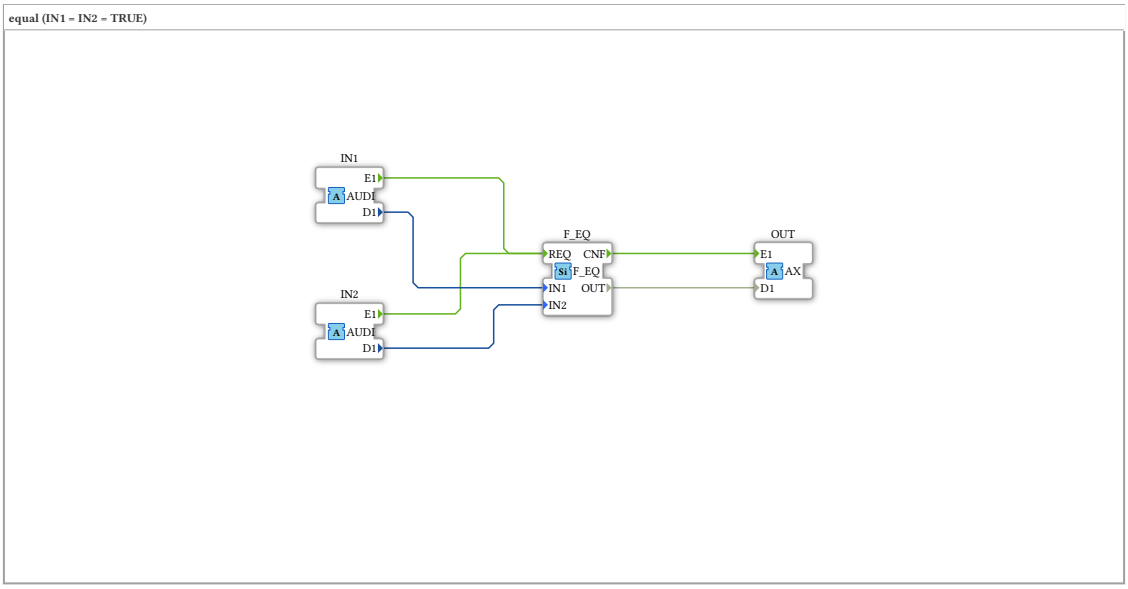
AUDI_EQ



Input value 1 -- left operand (IN1 = IN2) – *AUDI*

Input value 2 -- right operand (IN1 = IN2) – *AUDI*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AUDI_EQ** compares two analog values for equality. It is implemented as a composite function block (FB) and uses adapters to communicate with the environment. The result is a Boolean value, which is output via another adapter. The function block complies with the IEC 61131-3 classification “Standard comparison function”.

Interface Structure

The FB has only adapter interfaces. No direct event or data inputs/outputs are defined. The actual inputs/outputs are implemented via the adapter types.

Event Inputs

- **IN1.E1** (via adapter `IN1`): Event input of the first value to be compared.
- **IN2.E1** (via adapter `IN2`): Event input of the second value to be compared.

Event Outputs

- **OUT.E1** (via adapter `OUT`): Event output – activated once the comparison is complete and the result is valid.

Data Inputs

- **IN1.D1** (via adapter `IN1`): First comparison value (analog).
- **IN2.D1** (via adapter `IN2`): Second comparison value (analog).

Data Outputs

- **OUT.D1** (via adapter `OUT`): Boolean result of the comparison (`TRUE` if both values are equal, otherwise `FALSE`).

Adapters

Adapter	Direction	Type	Short Description
`IN1`	Socket	`adapter::types::unidirectional::AUDI`	Returns the first analog value including event.
`IN2`	Socket	`adapter::types::unidirectional::AUDI`	Returns the second analog value including event.
`OUT`	Plug	`adapter::types::unidirectional::AX`	Outputs the comparison result (Boolean) with an event.

Functionality

The function block **AUDI_EQ** internally contains a pre-existing function block `iec61131::comparison::F_EQ` for simple equality checking.

- As soon as an event arrives at one of the two input adapters (`IN1` or `IN2`), the internal function block `F_EQ` is called.
- The current data values from `IN1.D1` and `IN2.D1` are passed to `F_EQ`.
- The internal function block compares both values and returns a Boolean result.
- After processing is complete, the internal function block (FB) outputs an event that is forwarded via the output adapter `OUT`.
- Simultaneously, the comparison result is made available at the data output `OUT.D1`.

The function block operates purely **combinatorially** – it does not store any states and executes the comparison logic anew with each activation.

Technical Features

- **Composite FB**: The function block is implemented as a network of sub-FBs, enabling a modular and reusable structure.
- **Adapter-Based Interface**: The use of the unidirectional adapters `AUDI` and `AX` allows for flexible encapsulation of event and data channels. The adapters define the actual inputs and outputs.
- **Standard Component**: The internal FB `F_EQ` is part of the IEC 61131-3 library and guarantees platform-independent comparison logic.
- **No State Machine**: The function block does not have an internal state diagram but delegates the logic entirely to the embedded FB.

State Overview

The FB **AUDI_EQ** has **no independent states**. The internal logic is purely event-driven and results directly in the output. Therefore, a state overview is not available.

Application Scenarios

- **Monitoring Process Variables**: Checking two analog values (e.g., pressure sensors, temperature sensors) for equality.
- **Redundancy Check**: Comparing two independently measured values to detect sensor failures.
- **Signal Matching**: In control systems where it must be ensured that two input signals have the same value before an action is triggered.

Comparison with Similar Function Blocks

- **`F_EQ` (directly used)** – The internal function block `F_EQ` offers the same comparison logic, but without adapters. It requires explicit event and data inputs.
- **`F_NE` (inequality)** – Performs an inequality comparison; structurally similar, but the result is inverted.
- **`AUDI_GT` (greater than)** – Analogous structure with adapters, but a different comparison operator.

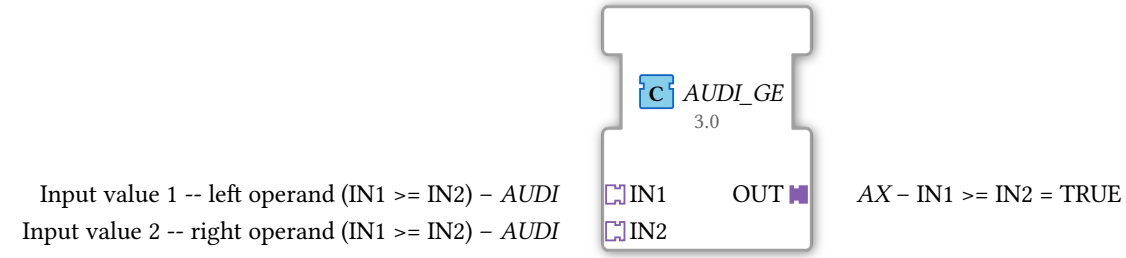
The **AUDI_EQ** differs in its pre-built adapter interface, which enables easy integration into existing adapter networks. It is specifically designed for applications that require standardized input/output via unidirectional adapters.

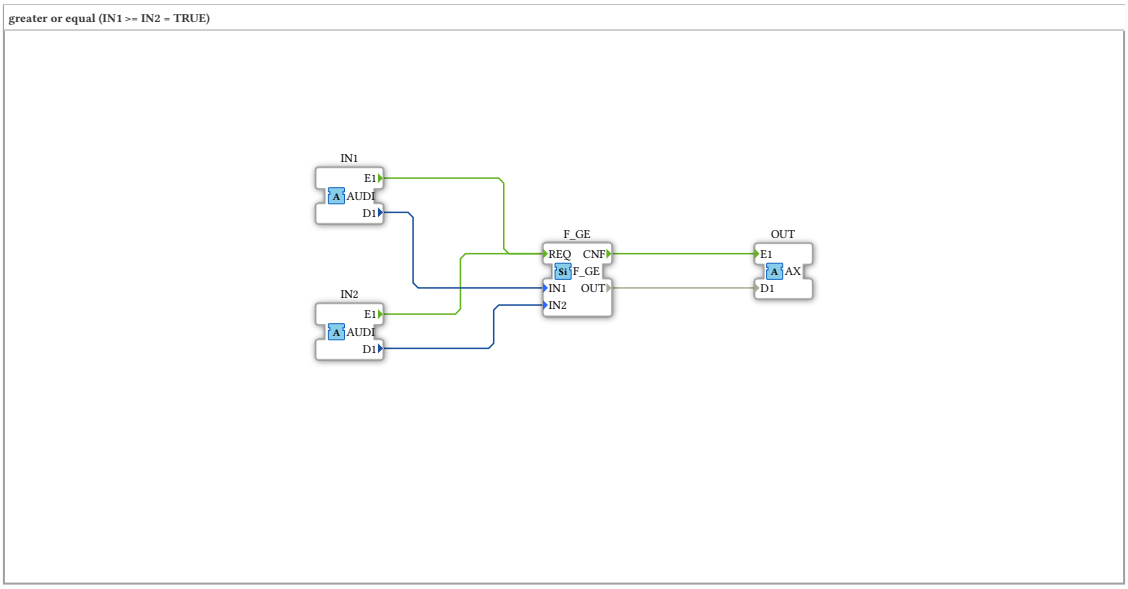
Conclusion

The **AUDI_EQ** is an easy-to-use, composite function block for equality comparisons with two analog values. Its adapter-based interface allows for seamless integration into modular control systems. The

internal use of standardized IEC function blocks ensures reliable functionality defined according to IEC 61131-3. This function block is particularly suitable for applications requiring a clear separation of event and data flows via adapters.

AUDI_GE





• – ○ • – ○ • – ○ •

Introduction

The function block `AUDI_GE` performs a “greater than or equal to” comparison of two values. It provides the result as a Boolean value. The block is implemented as a composition and internally uses the standard comparison block `F_GE` from the IEC 61131-3 library. Input and output are exclusively via unidirectional adapters.

Interface Structure

Event Inputs

- **No direct event inputs**

The comparison is triggered via the **socket adapters** `IN1` and `IN2`. Each of these adapters sends an event to the internal module as soon as valid data is available.

Event Outputs

- **No direct event outputs**

The result of the comparison is passed on as an event via the **Plug Adapter** `OUT` as soon as the evaluation is complete.

Data Inputs

The data values to be compared are obtained via the following adapters:

- **Adapter `IN1`** – provides the first comparison value (type corresponding to adapter `AUDI`).

- **Adapter `IN2`** – provides the second comparison value (type corresponding to adapter `AUDI`).

Data Outputs

- **Adapter `OUT`** – outputs the Boolean value `TRUE` if `IN1 ≥ IN2` is present, otherwise `FALSE` (type corresponds to adapter `AX`).

Adapters

Adapter	Direction	Type	Description
`IN1`	Socket	`adapter::types::unidirectional::AUDI`	First Input Value
`IN2`	Socket	`adapter::types::unidirectional::AUDI`	Second Input Value
`OUT`	Plug	`adapter::types::unidirectional::AX`	Output: `IN1 >= IN2` (boolean)

Functionality

1. The events from `IN1.E1` and `IN2.E1` are routed to the internal `F_GE` module (via an OR operator – both events trigger the same `REQ` input).
2. Simultaneously, the data `IN1.D1` and `IN2.D1` are transferred to the corresponding inputs `F_GE.IN1` and `F_GE.IN2`.
3. The internal function block `F_GE` calculates `IN1 ≥ IN2`.
4. The result appears at output `F_GE.OUT` and is passed on to adapter `OUT.D1`.
5. The event `F_GE.CNF` triggers `OUT.E1` to signal to the environment that a new result is available.

Technical Features

- The function block is entirely adapter-based – it has no traditional event or data inputs at the top level.
- The adapters `AUDI` and `AX` are unidirectional and must be defined in the corresponding project.
- Execution is asynchronous: As soon as one of the two input adapters provides an event, the comparison is recalculated. Synchronization of the values is not required, as both data must be available simultaneously for the result to be meaningful.
- The function block configuration complies with the IEC 61131-3 standard and the Eclipse 4diac environment.

State Overview

The function block itself does not have an explicit state machine – all control logic resides in the internal `F_GE`. Its behavior is deterministic:

- **Waiting for Trigger**: As long as no event arrives at `IN1` or `IN2`, the output `OUT` remains unchanged.
- **Calculation**: When an event arrives, the comparison is performed.
- **Result Output**: The result is provided at the output, and an event is sent to the `OUT` adapter.

Application Scenarios

- **Status Monitoring**: Checking whether a measured value reaches or exceeds a threshold.
- **Actuator Control**: Activating an output as soon as a condition `Wert1 ≥ Wert2` is met.
- **Value Comparison in Production Plants**: Comparing target/actual values in continuous processes.
- **Simple Logical Interconnections**: As part of more complex safety or control logic.

Comparison with Similar Function Blocks

- **AUDI_GT** (greater than) – compares against `>`, returns `TRUE` only if the first value is strictly greater.
- **AUDI_EQ** (equal) – compares for equality.
- **AUDI_LE** (less or equal) – the counterpart to ≤.
- **AUDI_GE** is the direct counterpart to `AUDI_LE` and covers the case of "at least equal". Compared to a single function block like `F_GE`, `AUDI_GE` offers adapter-based encapsulation, which facilitates reuse in different projects.

Conclusion

AUDI_GEis a specialized comparison block that enables flexible and clean interface definition through the use of adapters. It is particularly suitable for modular automation

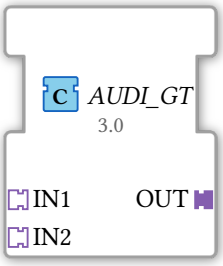
`projects where values are exchanged via standardized interfaces. Internal use of the provenF_GE` block guarantees correct and efficient comparison operations.

–

🌐 Related topic subpages on ms-muc-docs.de

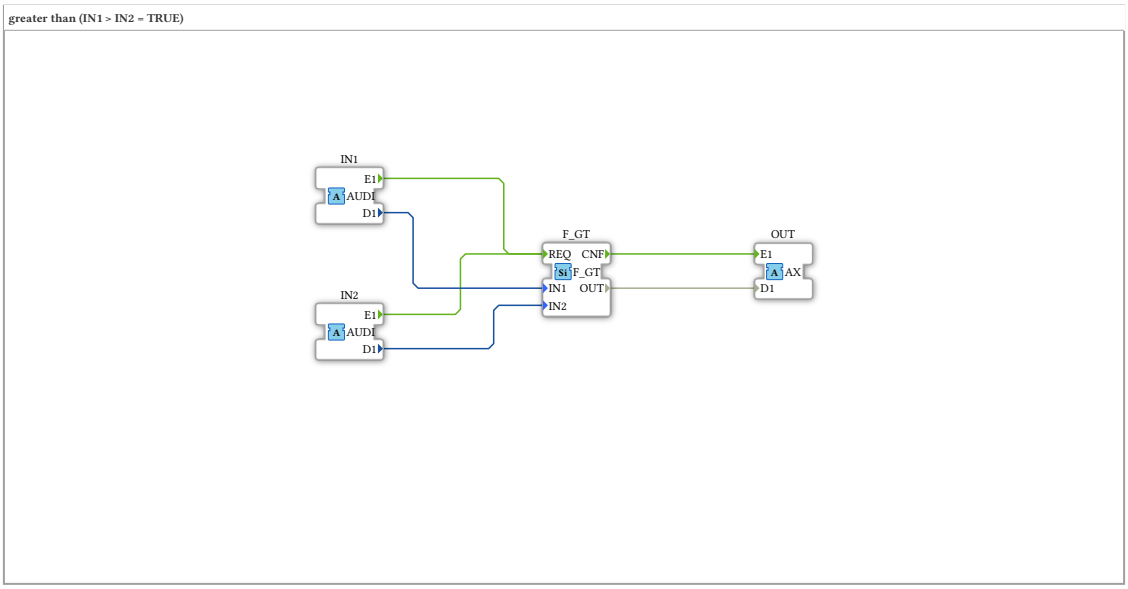
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AUDI_GT



Input value 1 -- left operand (IN1 > IN2) – *AUDI*
Input value 2 -- right operand (IN1 > IN2) – *AUDI*

AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The AUDI_GT function block performs a “greater than” comparison of two values. It is designed as an IEC 61131-3 compliant comparison block and uses an adapter-based interface to transmit values and events. The block is part of the iec61131::comparison library.

Interface Structure

Event Inputs

- **IN1.E1** – Event input for the first value (via adapter IN1)
- **IN2.E1** – Event input for the second value (via adapter IN2) Both events are connected to the internal REQ input of the F_GT block.

Event Outputs

- **OUT.E1** – Event output that is activated after the comparison is complete (via adapter OUT). The result is then available at the data output.

Data Inputs

- **IN1.D1** – First input value (via adapter IN1, type depends on adapter AUDI)
- **IN2.D1** – Second input value (via adapter IN2, type depends on adapter AUDI)

Data Outputs

- **OUT.D1** – Comparison result: TRUE if IN1 > IN2, otherwise FALSE (via adapter OUT, type depends on adapter AX)

Adapters

- **Socket IN1** (Type: `adapter::types::unidirectional::AUDI`) – Connection for the first value to be compared.
- **Socket IN2** (Type: `adapter::types::unidirectional::AUDI`) – Connection for the second value to be compared.
- **Plug OUT** (Type: `adapter::types::unidirectional::AX`) – Output of the comparison result.

The adapters used are unidirectional and each provides one event port and one data port.

Functionality

The AUDI_GT function block encapsulates the basic comparison function block F_GT from the IEC 61131-3 library. The input adapters IN1 and IN2 each provide a value (D1) and a start signal (E1). Both events are connected to the REQ input of F_GT, so that the comparison is triggered as soon as at least one of the events occurs. Internally, F_GT performs the comparison of the two values. The result (BOOL) is provided via the output adapter OUT, simultaneously generating an event (E1).

Technical Features

- The function block uses adapters instead of direct input/output ports, which allows for flexible encapsulation and reuse of the interfaces.
- The events from both inputs are combined on the same REQ input of the internal F_GT. This means that the function block re-performs the comparison with each event received at IN1 or IN2.
- The function block outputs the result as a BOOL value (TRUE/FALSE). The exact data types of the adapters (AUDI, AX) are application-specific and may vary depending on the context.
- The function block is identified as part of the `iec61131::comparison` library and complies with the IEC 61131-3 standard.

State Overview

The AUDI_GT function block does not have its own state machine. All logic is implemented by the internal F_GT, which performs a simple combinational comparison. There are no internal states or memory behaviors; the result is recalculated with each activation.

Application Scenarios

- **Process Monitoring**: Comparison of measured values (e.g., temperature, pressure) with a threshold value. If Value1 > Value2, an alarm is triggered.
- **Logical Decisions**: In automation processes, e.g., sorting logic or priority controls.
- **Safety Functions**: Checking whether a current value exceeds a limit.
- **Adapter-Based Systems**: Wherever modular, reusable interfaces via adapters are desired.

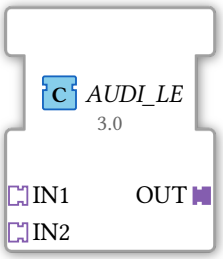
Comparison with Similar Function Blocks

- **GT (Greater Than)**: Simple comparators like `F_GT` (IEC 61131-3) have direct inputs. `AUDI_GT` extends this to an adapter-based interface, which facilitates integration into heterogeneous systems.
- **GE (Greater or Equal)**, **LT (Less Than)**, **EQ (Equal)**: Analogous function blocks for other comparison operations. This function block only supports "Greater Than".
- **Comparison Cascades**: Multiple AUDI_GTs can be linked to create more complex expressions by connecting their outputs to the inputs of other function blocks.

Conclusion

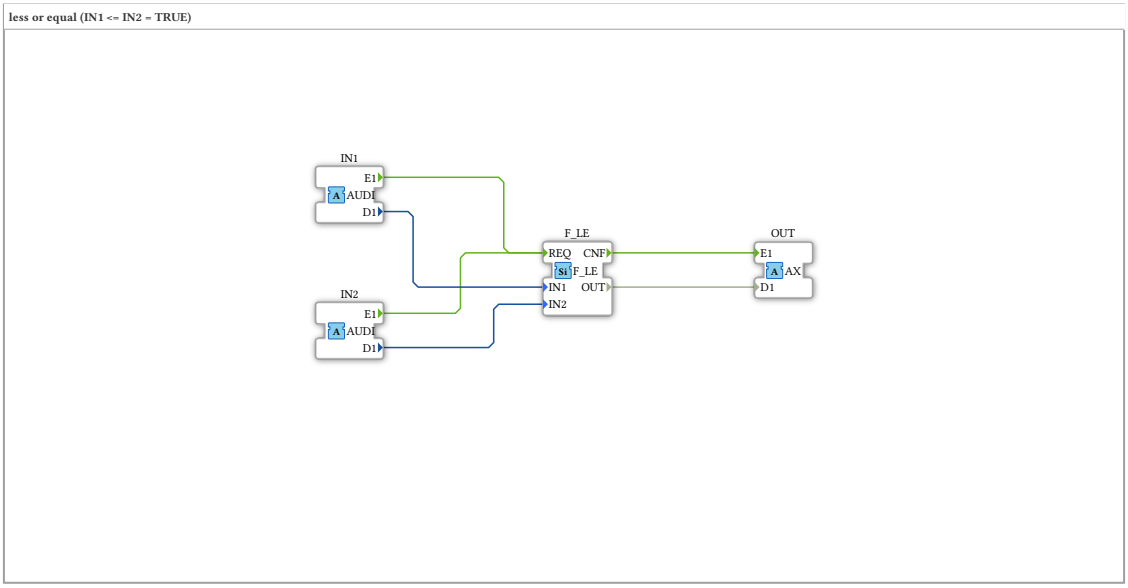
The AUDI_GT is a dedicated comparison function block for the "greater than" function with adapter-based inputs and outputs. It offers a standardized, modular interface according to IEC 61131-3 and is particularly suitable for use in distributed automation systems that require loose coupling of components. Thanks to its adapter technology, the function block can be flexibly embedded in various contexts.

AUDI_LE



Input value 1 -- left operand (IN1 <= IN2) – *AUDI*
Input value 2 -- right operand (IN1 <= IN2) – *AUDI*

AX – IN1 <= IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUDI_LE** performs a less-than-equal-to comparison of two values. It is built as a composite function block based on the standardized comparison block `F_LE` from the IEC 61131 classification. Communication with the environment occurs exclusively via adapter interfaces, allowing the block to be integrated modularly into event-driven processes.

Interface Structure

Event Inputs

Name	Data Type	Description
`IN1.E1`	EVENT	Event input of adapter IN1 – triggers processing of the first value.
`IN2.E1`	EVENT	Event input of adapter IN2 – triggers processing of the second value.

Note: Both events must be received for a comparison to take place (logical AND operation in the internal network).

Event Outputs

Label	Data Type	Description
`OUT.E1`	EVENT	Event output of adapter OUT – is activated as soon as the comparison result is available.

Data Inputs

Label	Data Type	Description

IN1.D1	ANY	First value to be compared (via adapter IN1).
IN2.D1	ANY	Second value to be compared (via adapter IN2).

Data Outputs

Label	Data Type	Description
OUT.D1	BOOL	Comparison result: TRUE if IN1 ≤ IN2; otherwise FALSE.

Adapter

Name	Type	Direction	Description
IN1	adapter::types::unidirectional::AUDI	Socket (Input)	Capture the first value (event + data).
IN2	adapter::types::unidirectional::AUDI	Socket (Input)	Capture the second value (event + data).
OUT	adapter::types::unidirectional::AX	Plug (Output)	Output the comparison result (event + data).

Functionality

1. The function block (FB) expects one event and one data value at each of its two input adapters (IN1 and IN2).
2. Internally, the two events are combined (via an implicit AND condition in the network), and the embedded function block F_LE is started with the data values.
3. F_LE performs the comparison IN1 ≤ IN2 and places the result (Boolean value) at its data output.
4. As soon as the result is available, the event at the output adapter OUT is triggered, and the result value is made available via OUT.D1.

Technical Features

- **Adapter-Based Interfaces:** The function block (FB) does not have traditional individual ports but uses standardized unidirectional adapters. This facilitates reuse and integration into complex communication structures.
- **Composite Structure:** All logic is implemented via an internal FB (F_LE). This makes the function block easy to understand and extend.
- **Event Synchronization:** Both input events must arrive before processing starts. This ensures that current value pairs are always compared.

State Overview

The FB does not have its own directed state machine, as it operates purely combinatorially. The internal F_LE is also stateless. The only relevant logic is the AND operation of the events – after receiving both events, the result is output once.

Application Scenarios

- **Limit Monitoring:** Check whether a measured value exceeds a threshold.
- **Enabling Logic:** Activate an action only if the current requirement is less than or equal to a maximum allowable value.
- **Comparison Chains:** Combine multiple AUDI_LE function blocks to implement complex conditions (e.g., x ≤ y ≤ z).

Comparison with Similar Function Blocks

FB	Function	Difference to AUDI_LE
AUDI_LT	Less than (<)	Not including equality.
AUDI_EQ	Equality (=)	Checks for exact matches only.
AUDI_GE	Greater Than or Equal To (≥)	Reverse comparison direction.

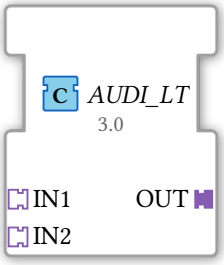
AUDI_AND	Logical AND	Works with Boolean values, not numerical comparisons.

Conclusion

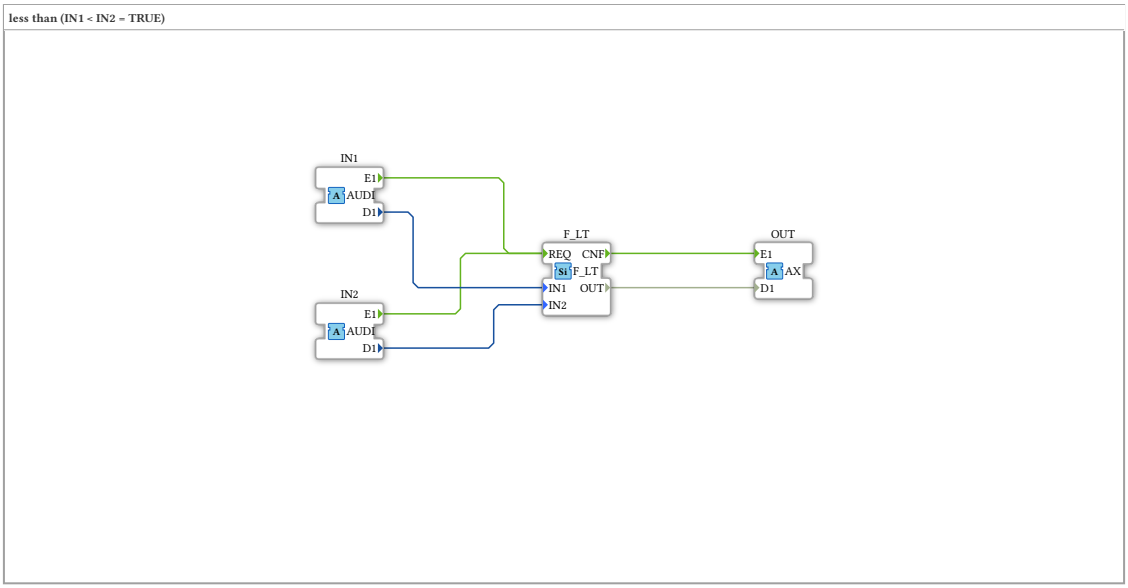
AUDI_LE` is a specialized comparison block that, thanks to its adapter interfaces, is particularly well-suited for modular, event-driven automation solutions. It offers clear, standardized logic for the “less than or equal to” condition and, due to its composite structure, can be easily integrated into custom applications.

AUDI_LT

Input value 1 -- left operand (IN1 < IN2) – *AUDI*
Input value 2 -- right operand (IN1 < IN2) – *AUDI*



AX – IN1 < IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block `AUDI_LT` performs a “less than” comparison between two input values. It is frequently used in automation technology to monitor thresholds or make logical decisions. The block is classified as a “standard comparison function” according to IEC 61131-3 and uses adapters for flexible connection of different data types.

Interface Structure

Event Inputs

Events are forwarded to the function block from external sources via the adapter sockets:

- `IN1.E1` – Event that triggers the comparison operation as soon as a new value is present at `IN1`.
- `IN2.E1` – Event that also triggers the comparison operation as soon as a new value is present at `IN2`.

Event Outputs

- `OUT.E1` – Event that is output after the comparison is complete and signals that the result at data output `OUT.D1` is valid.

Data Inputs

- `IN1.D1` – First value to be compared (data of the adapter type `AUDI`).
- `IN2.D1` – Second value to be compared (data of the adapter type `AUDI`).

Data Outputs

- `OUT.D1` – Comparison result: `TRUE` if `IN1 < IN2`, otherwise `FALSE` (data of the adapter type `AX`).

Adapters

- `IN1` – Socket (input) of type `adapter::types::unidirectional::AUDI`. Sends an event (`E1`) and a data value (`D1`) to the FB.
- `IN2` – Socket (input) of the same type as `AUDI`. Provides the second input value.
- `OUT` – Plug (output) of type `adapter::types::unidirectional::AX`. Receives the result event (`E1`) and the result data value (`D1`) for passing to subsequent function blocks.

Functionality

This function block performs a numerical comparison of the two data points at `IN1` and `IN2`. As soon as one of the two events (`IN1.E1` or `IN2.E1`) occurs, the internal function block `F_LT` (Less Than) is called. This function reads the current values of `IN1` and `IN2` and calculates their relative size. The result (`TRUE` / `FALSE`) is provided at the data output `OUT.D1`, and the event `OUT.E1` is triggered simultaneously. Since both event inputs are connected to the same `REQ` input of the internal function block, the comparison is performed again whenever either value changes.

Technical Features

- **Adapter-Based Interfaces** – The use of the adapters `AUDI` and `AX` enables universal coupling to various data types without requiring the function block itself to be typed. The actual type adaptation takes place in the adapter definitions.
- **Asynchronous Triggering** – The comparison is always started when an event arrives at `IN1` or `IN2`. This allows the function block to react immediately to changes in either of the two inputs.
- **Stateless** – The function block has no internal state; the output signal depends solely on the current input values.

State Overview

The function block is purely functional and does not require its own state machine. It performs a one-time comparison operation after each event and then returns to an empty wait state.

Application Scenarios

- **Limit Monitoring** – e.g., checking whether a temperature falls below a threshold.
- **Range Check** – Combination with other comparison blocks (e.g., `AUDI_GT`) to implement “inside” or “outside” logic.
- **Control Logic** – Boolean decisions depending on two analog or digital variables.

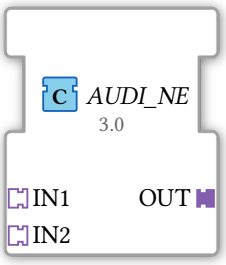
Comparison with Similar Blocks

- **`AUDI_EQ`** – Checks for equality (`IN1 = IN2`).
- **`AUDI_GT`** – Checks for greater than (`IN1 > IN2`).
- **`AUDI_LT`** – Checks for less than (`IN1 < IN2`). All three blocks have identical interfaces and differ only in their comparison operation. They can be easily interchanged to adapt the logic.

Conclusion

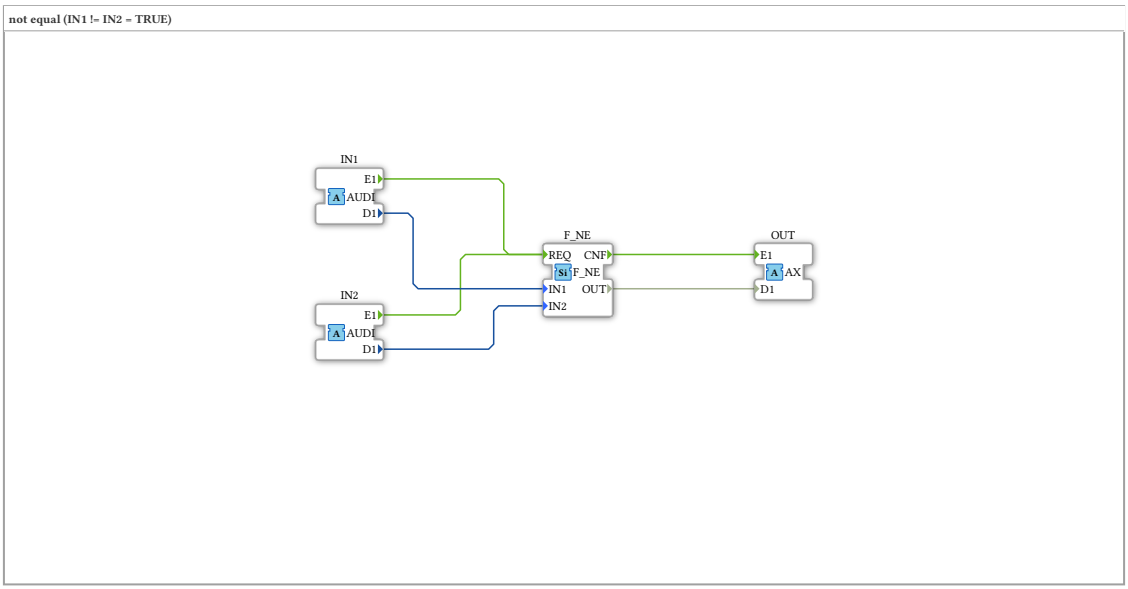
The `AUDI_LT` is a simple yet robust comparison module that can be flexibly integrated into a wide variety of automation environments thanks to its adapter interfaces. Its asynchronous operation and the clear separation of events and data make it a fundamental tool for value comparison tasks in control engineering.

AUDI_NE



Input value 1 -- left operand (IN1 != IN2) – *AUDI*
Input value 2 -- right operand (IN1 != IN2) – *AUDI*

AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AUDI_NE** performs a not equal check. It compares two input values provided via adapters and outputs the comparison result as a Boolean value via an output adapter. This block is primarily used in automation systems that support adapter-based, event-driven communication.

Interface Structure

Event Inputs

The function block does not have separate event inputs on the interface itself. Events are transmitted via the adapter sockets **IN1** and **IN2**. Each of these sockets contains an event input (E1), the arrival of which triggers the internal comparison.

Event Outputs

The output adapter **OUT** provides an event output (E1). This signals the completion of the comparison and indicates that the result at the data output is valid.

Data Inputs

- **IN1.D1** – First value to be compared (data type depends on the adapter definition `adapter::types::unidirectional::AUDI`).
- **IN2.D1** – Second value to be compared (same data type as IN1).

Data Outputs

- **OUT.D1** – Boolean result of the inequality check: `TRUE` if IN1 != IN2, otherwise `FALSE`.

Adapter

Adapter	Type	Direction	Description
IN1	`adapter::types::unidirectional::AUDI`	Socket	Captures the first input value
IN2	`adapter::types::unidirectional::AUDI`	Socket	Receives the second input value
OUT	`adapter::types::unidirectional::AX`	Plug	Outputs the comparison result

Functionality

The **AUDI_NE** is a composite function block. Internally, it uses the standardized comparison block **F_NE (Not Equal)** from the IEC 61131 library.

- If an event occurs at **IN1.E1** or **IN2.E1**, the signal is forwarded to the internal block **F_NE.REQ**.
- Simultaneously, the current data values from **IN1.D1** and **IN2.D1** are transferred to **F_NE.IN1** and **F_NE.IN2**.
- **F_NE** performs the inequality comparison and outputs the result via **F_NE.OUT** and an acknowledgment event via **F_NE.CNF**.
- The acknowledgment event is passed to **OUT.E1**, and the result to **OUT.D1**.

The function block (FB) thus operates in an event-driven manner: Each event at either of the two inputs triggers a new comparison, and the output is updated immediately.

Technical Features

- **Adapter-based interface**: Adapters are used instead of discrete inputs/outputs. This enables a type-safe and modular connection with other function blocks that use the same adapter protocol.
- **Two independent event sources**: Both IN1 and IN2 can provide an event. The internal behavior is designed so that a new comparison is started after each event – regardless of the source.
- **Reuse of Standardized Logic**: The actual comparison function is implemented by the embedded FB **F_NE**, ensuring high reliability and compliance with IEC 61131.

State Overview

The FB has no explicit states (no ECC). It is a combinational block without memory. The output is determined solely by the current input values and the triggering event.

Application Scenarios

- **Deviation Monitoring**: Checks whether two measured values (e.g., setpoint and actual values) differ. The result can be used for alarming or control.
- **Redundancy Check**: In safety-critical systems, the discrepancy between two redundant sensors can indicate a fault.
- **State Comparisons**: Compares binary states or count values to detect changes or inconsistencies.

Comparison with Similar Function Blocks

- **AUDI_EQ (Equal)**: The counterpart for equality checking. Same interface, but inverted logic: Output `TRUE` on equality.
- **Direct Use of F_NE**: Without the adapter wrapper, F_NE could be used directly. However, **AUDI_NE** offers a standardized adapter interface that improves integration and type safety in frameworks like 4diac.

Conclusion

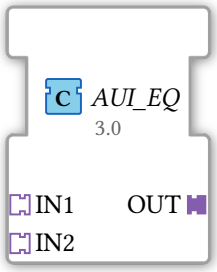
The **AUDI_NE** is a practical function block for non-equal comparison in adapter-based IEC 61131 systems. By using adapters and embedding the proven **F_NE** function block, it combines type safety with established functionality. It is particularly suitable for event-driven automation solutions where modular and reusable components are required.

AUDI_NE —

🌐 Related topic subpages on ms-muc-docs.de

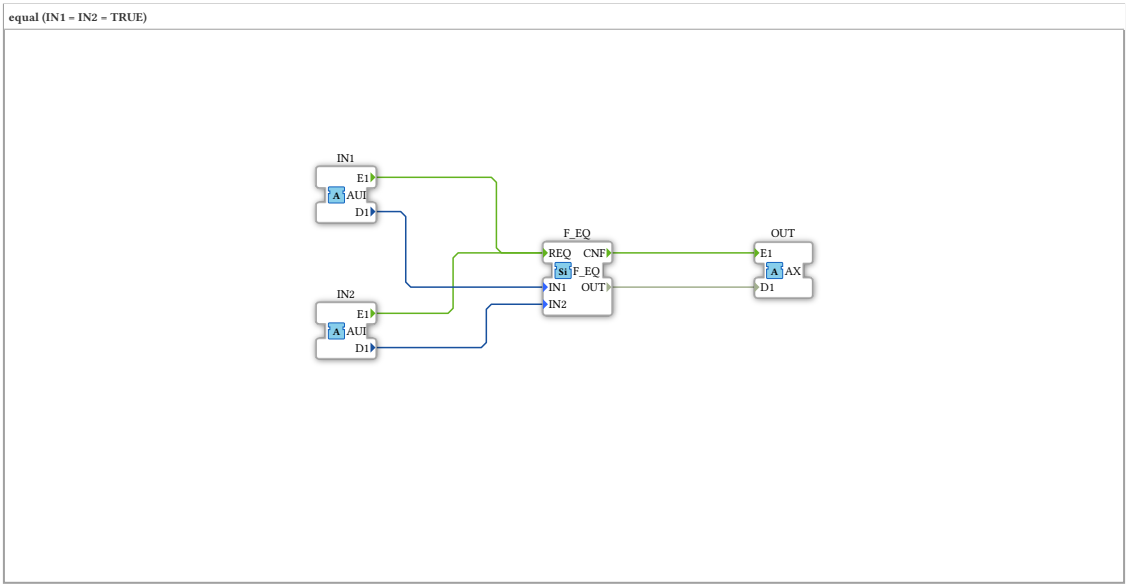
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AUI_EQ



Input value 1 -- left operand (IN1 = IN2) – *AUI*
Input value 2 -- right operand (IN1 = IN2) – *AUI*

AX – IN1 = IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The AUI_EQ function block performs an equality comparison between two analog values. It is classified as a standard comparison function according to IEC 61131-3 and uses adapters for type-safe communication with the environment. The result is output as a Boolean value, which is `true` if and only if both input values are equal.

Interface Structure

Event Inputs

- **IN1.E1** – Trigger for the comparison (sent when the first value is updated).
- **IN2.E1** – Alternative trigger for the comparison (sent when the second value is updated).

Both events are associated with the same internal processing event; the comparison is triggered when either event occurs.

Event Outputs

- **OUT.E1** – Confirmation that the comparison is complete and the result is available at the data output.

Data Inputs

- **IN1.D1** – First analog value to be compared.
- **IN2.D1** – Second analog value to be compared.

Data Outputs

- **OUT.D1** – Comparison result: `true` if `IN1.D1 == IN2.D1`, otherwise `false`.

Adapters

- **IN1** (Socket) – Adapter type `adapter::types::unidirectional::AUI`, encapsulates the first input value and the associated event.
- **IN2** (Socket) – Adapter of type `adapter::types::unidirectional::AUI`, encapsulates the second input value and the associated event.
- **OUT** (Plug) – Adapter of type `adapter::types::unidirectional::AX`, encapsulates the result and the output event.

The adapters enable loose coupling between the function block and its environment, simplifying reuse in different applications.

Functionality

AUI_EQ delegates the actual comparison logic to the internal function block `F_EQ` (type `iec61131::comparison::F_EQ`). As soon as an event arrives at one of the input adapters (IN1 or IN2), the calculation is started. The internal `F_EQ` compares the two passed values and returns the Boolean result. After the calculation is complete, the result is output via the output adapter OUT, and a corresponding event is sent.

The function block is event-driven – it reacts to input events and does not require its own cyclic call.

Technical Features

- The function block uses only adapters for communication, which offers high flexibility when connecting to various data sources and sinks.
- Internally, it is based on the standard function block `F_EQ` from the IEC 61131 library, which provides a robust and tested comparison function.
- The input events are logically ORed: Each of the two events triggers a recalculation. This simplifies control, as only the currently updated signal needs to be sent.
- The function block has been further developed in several versions and was last updated in 2025.

State Overview

AUI_EQ does not have an explicit state machine. Its behavior is entirely determined by the event processing of the internal `F_EQ`. Upon the arrival of an event, the function block performs a calculation and sends the result – there are no waiting or blocking states.

Application Scenarios

- **Equality Monitoring**: Checking whether two sensor readings (e.g., temperature, pressure) have the same value.
- **Alarming**: Triggering an alarm if two redundant measurements differ (logical negation of the output).
- **Control**: Switching actuators when a setpoint matches an actual value.
- **Plausibility Check**: Comparing analog values from different sources for error detection.

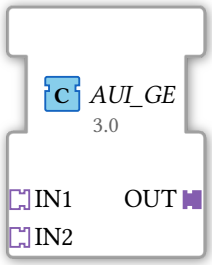
Comparison with Similar Function Blocks

In addition to `AUI_EQ`, the IEC 61131 library contains other comparison blocks such as `AUI_GT` (greater than), `AUI_LT` (less than), and `AUI_NE` (not equal to). These share the same adapter interface but differ in their comparison operation. `AUI_EQ` is specifically optimized for equality testing and does not offer any other comparison logic. For more comprehensive comparisons (e.g., greater than or equal to), combinations with logic blocks are required.

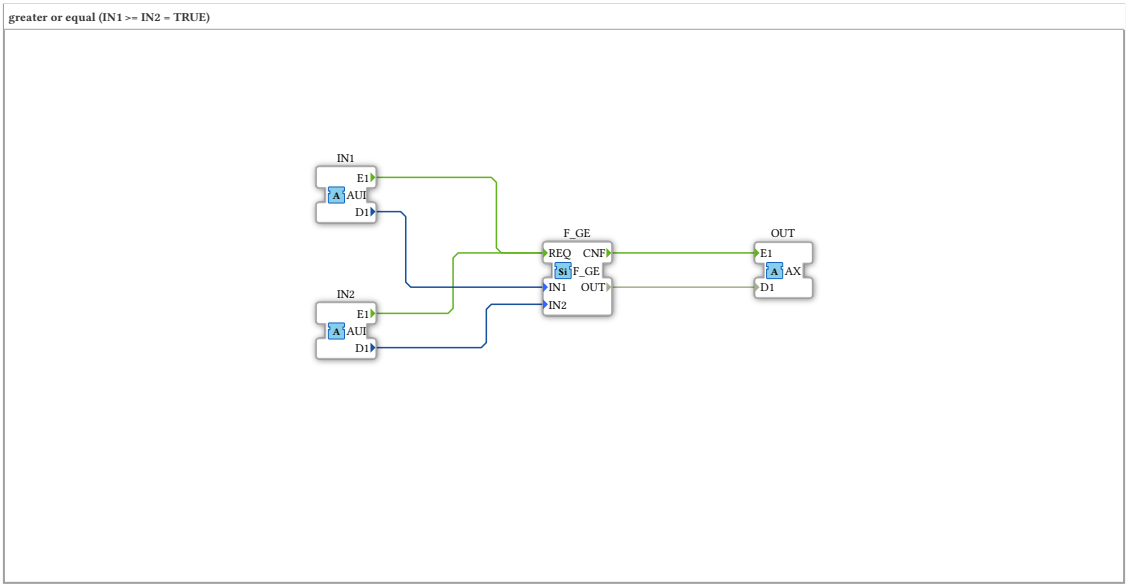
Conclusion

The AUI_EQ function block implements a simple and reliable equality check for analog values. The use of adapters makes it highly adaptable to various automation environments. Its event-driven operation and the internal use of a standardized comparison module ensure consistent and high-performance execution. AUI_EQ is therefore a fundamental building block for monitoring and control tasks where value equality must be determined.

AUI_GE



Input value 1 -- left operand (IN1 >= IN2) – *AUI*
Input value 2 -- right operand (IN1 >= IN2) – *AUI*
AX – IN1 >= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AUI_GE** compares two values and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type depends on the adapter type `AUI`, usually a numeric value).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AUI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AUI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AUI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GE` (type `iec61131::comparison::F_GE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters, allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”).
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checks whether a measured value (e.g., fill level, rotational speed) has reached or exceeded a predefined threshold.
- **Control Logic**: Activates an action (e.g., stop a pump) when an actual value reaches or exceeds a setpoint.
- **Data Filtering**: Selects data records or signals that meet a minimum condition.

Comparison with Similar Function Blocks

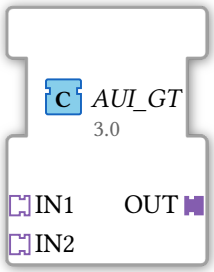
- **AUI_LE**: Checks for “less than or equal to” – the inverse function of AUI_GE.
- **AUI_EQ**: Checks for equality of values.
- **AUI_GT**: Checks for “greater than” – the base variant without the equality case, of which AUI_GE is the extended version.

AUI_GE completes the AUI comparison family with the boundary case of equality, making it the right choice when a threshold should be inclusive.

Conclusion

AUI_GE is a simple yet important comparison block for IEC 61131-3 control applications. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

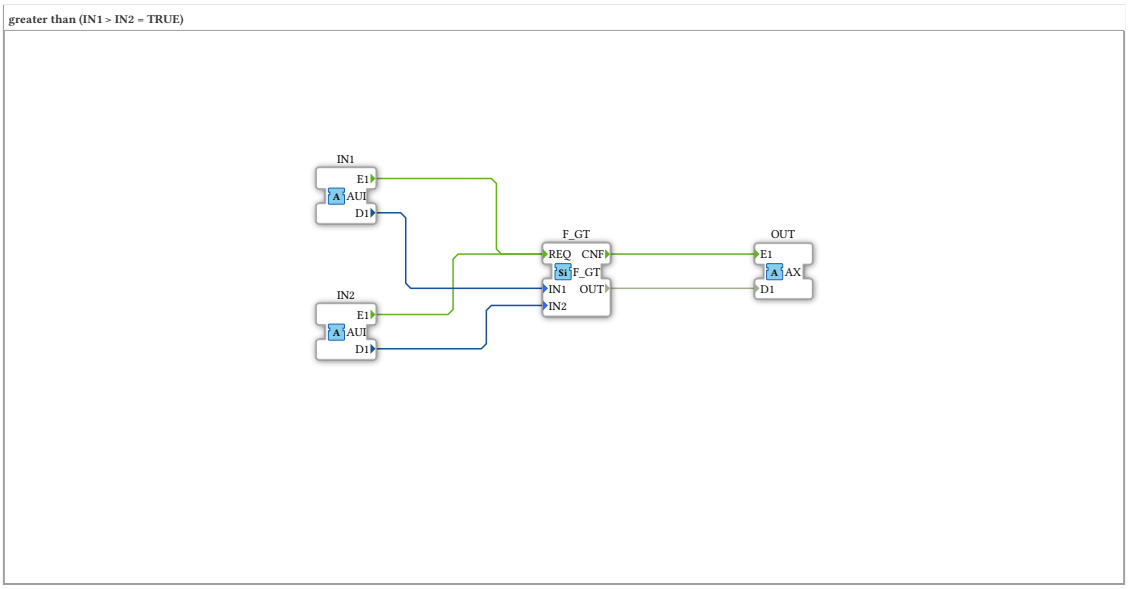
AUI_GT



Input value 1 -- left operand (IN1 > IN2) – *AUI*

Input value 2 -- right operand (IN1 > IN2) – *AUI*

AX – IN1 > IN2 = TRUE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_GT** compares two values and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type depends on the adapter type `AUI`, usually a numeric value).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AUI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AUI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AUI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GT` (type `iec61131::comparison::F_GT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters, allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”).
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checks whether a measured value (e.g., temperature, pressure) exceeds a predefined threshold.
- **Control Logic**: Activates an action (e.g., open a valve) when a setpoint is greater than the actual value.
- **Data Filtering**: Selects data records or signals that meet a specific condition.

Comparison with Similar Function Blocks

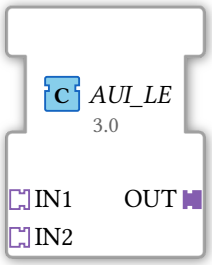
- **AUI_LT**: Checks for “less than” values – the inverse function of AUI_GT.
- **AUI_EQ**: Checks for equality of values.
- **AUI_GE** / **AUI_LE**: Perform “greater than or equal to” and “less than or equal to” comparisons, respectively – they are extended versions of AUI_GT and AUI_LT.

AUI_GT is the basic “greater than” component, enabling easy integration into adapter-based networks via the two socket adapters.

Conclusion

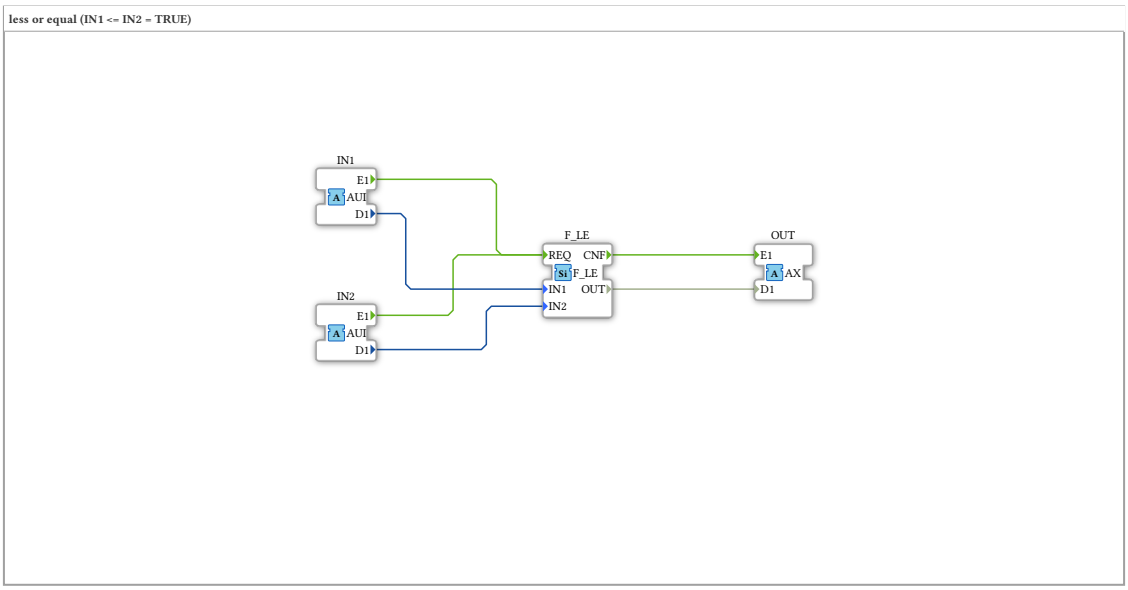
AUI_GT is a simple yet important comparison block for IEC 61131-3 control applications. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AUI_LE



Input value 1 -- left operand (IN1 <= IN2) – *AUI*
Input value 2 -- right operand (IN1 <= IN2) – *AUI*

AX – IN1 <= IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AUI_LE** performs a less-than-equal-to comparison of two analog input values. It is implemented as a composite block that internally uses the IEC 61131-3 block `F_LE`. The result (a BOOL value) is provided via an output adapter. The function block is suitable for event-driven limit monitoring in automation systems.

Interface Structure

Event Inputs

- **IN1.E1** (via adapter IN1) – Event that triggers the comparison as soon as the first input value is available.
- **IN2.E1** (via adapter IN2) – Event that triggers the comparison as soon as the second input value is available.

Event Outputs

- **OUT.E1** (via adapter OUT) – Event output after a successful comparison, indicating that the result is valid.

Data Inputs

- **IN1.D1** (via adapter IN1) – First input value (analog, type according to adapter `AUI`).
- **IN2.D1** (via adapter IN2) – Second input value (analog, type according to adapter `AUI`).

Data Outputs

- **OUT.D1** (via adapter OUT) – Comparison result (BOOL): `TRUE` if IN1 ≤ IN2, otherwise `FALSE`.

Adapters

- **IN1** – Socket (Type: `adapter::types::unidirectional::AUI`) for the first analog value.

- **IN2** – Socket (Type: `adapter::types::unidirectional::AUI`) for the second analog value.
- **OUT** – Plug (Type: `adapter::types::unidirectional::AX`) for the comparison result.

Functionality

The module contains an internal network with a `F_LE` module. The events of the two input adapters (`IN1.E1` and `IN2.E1`) are connected to the `REQ` input of the `F_LE` module. Each incoming event (regardless of the input) triggers a comparison of the current data values `IN1.D1` and `IN2.D1`. The internal `F_LE` calculates `(IN1 ≤ IN2)` and stores the result on its output `OUT`. This result is output via the adapter `OUT.D1`, and simultaneously, an event is sent via `OUT.E1`.

Technical Features

- **Adapter-based interface**: The inputs and outputs are implemented as unidirectional adapters, enabling flexible integration with other components.
- **Event triggering**: The comparison is triggered by any event present at either input – simultaneous events at both inputs are not required.
- **Internal Standard Block**: The comparison logic uses the tested IEC 61131-3 block `F_LE`.
- **License**: This block is licensed under the Eclipse Public License 2.0 (EPL-2.0), Copyright (c) 2013 TU Wien ACIN.

State Overview

The block does not have an explicit state machine. It operates in an event-driven manner, and its output is the direct result of the internal combinatorial comparison. After event processing, the result remains stable until the next event.

Application Scenarios

- **Limit Monitoring**: Checking whether a measured value (e.g., temperature, pressure) exceeds a set threshold.
- **Control Logic**: Triggering an action when two analog signals are in a defined relationship (e.g., fill level ≤ maximum fill level).
- **Safety Functions**: Compares redundant sensor values for "less than or equal to" for fault detection.

Comparison with Similar Function Blocks

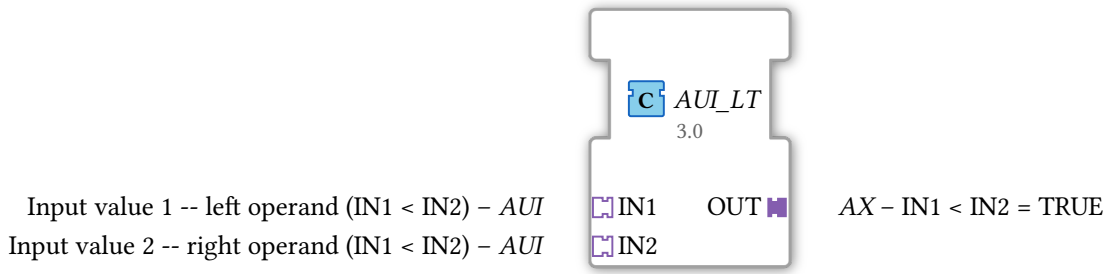
- **AUI_LT**: Performs a "less than" (strict less) comparison.
- **AUI_EQ**: Checks for equality between the two values.
- **AUI_GT**: Checks for "greater than".
- **AUI_GE**: Checks for "greater than or equal to".

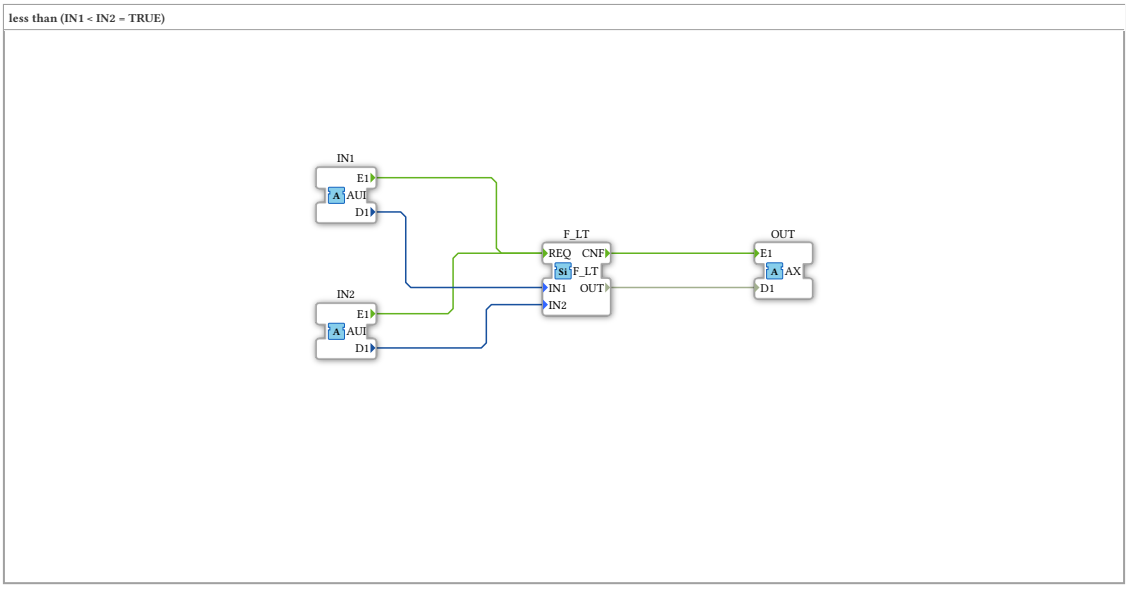
All these function blocks have the same adapter interface (AUI for inputs, AX for outputs) and differ only in their comparison operation.

Conclusion

The `AUI_LE` function block offers a simple and reliable way to check analog signals for "less than or equal to". Thanks to its adapter-based interface and event-driven execution, it can be seamlessly integrated into automation projects that require standardized comparison functions.

AUI_LT





• – ○ • – ○ • – ○ •

Introduction

The function block `AUI_LT` performs a less-than comparison (<) between two input values. It is accessed via adapter interfaces and is suitable for use in component-based automation systems according to IEC 61131-3. The block performs the comparison asynchronously and outputs the result as a Boolean value via an output adapter.

Interface Structure

Event Inputs

Name	Description
IN1.E1	Event at socket adapter IN1; triggers the comparison as soon as a new value is present at IN1.
IN2.E1	Event at socket adapter IN2; triggers the comparison as soon as a new value is present at IN2.

Event Outputs

Name	Description
OUT.E1	Event at plug adapter OUT; signals that a valid result is present at OUT.D1 after a comparison.

Data Inputs

Name	Description
IN1.D1	First value of the comparison (ANY_BIT compatible).
IN2.D1	Second value of the comparison (ANY_BIT compatible).

Data Outputs

Name	Description
OUT.D1	Comparison result: `true`, if IN1.D1 < IN2.D1; sonst `false`.

Adapter

Bezeichnung	Typ	Richtung	Beschreibung
IN1	AUI (unidirectional)	Socket	Eingangsadapter für den ersten Wert und dessen Ereignis.
IN2	AUI (unidirectional)	Socket	Eingangsadapter für den zweiten Wert und dessen Ereignis.
OUT	AX (unidirectional)	Plug	Ausgangsadapter für das Vergleichsergebnis und das zugehörige Ereignis.

Funktionsweise

Der `AUI_LT`-Baustein ist als Composite-FB realisiert. Intern wird ein einzelner `F_LT`-Funktionsblock (aus der IEC 61131-3 Vergleichsbibliothek) verwendet.

- Die **Ereignis-Eingänge** `IN1.E1` und `IN2.E1` sind beide mit dem `REQ`-Eingang des internen `F_LT` verbunden. Sobald an einem der beiden Adapter ein Ereignis eintrifft, wird der Vergleich angestoßen.
- Die **Datenwerte** `IN1.D1` und `IN2.D1` werden direkt an die entsprechenden Eingänge `IN1` und `IN2` des `F_LT` weitergeleitet.
- Nach Abschluss der Berechnung signalisiert der `F_LT` über seinen Ausgang `CNF` das Ereignis `OUT.E1` und stellt das Vergleichsergebnis am Datenausgang `OUT.D1` bereit.

Der Vergleich erfolgt nach der Logik: **IN1.D1 < IN2.D1 ⟹ OUT.D1 = true**, andernfalls `false`.

Technische Besonderheiten

- **Adapterbasierte Kommunikation**: Alle Ein- und Ausgänge erfolgen über unidirektionale Adapter (Typen `AUI` und `AX`). Dies ermöglicht eine lose Kopplung und Wiederverwendung in verschiedenen Umgebungen.
- **Asynchrone Auslösung**: Da beide Ereigniseingänge auf den gleichen `REQ` des internen `F_LT` führen, kann der Vergleich durch ein Ereignis an **einem** der beiden Eingänge gestartet werden. Es ist nicht erforderlich, dass beide Ereignisse gleichzeitig eintreffen.
- **Typkompatibilität**: Der Baustein arbeitet mit beliebigen `ANY_BIT`-Datentypen (z. B. `BOOL`, `BYTE`, `WORD`, `DWORD`, `LWORD`), sofern die angeschlossenen Adapter dies unterstützen.
- **Keine Zustandsspeicherung**: Der FB ist rein kombinatorisch; er speichert keine historischen Werte.

Zustandsübersicht

Da der `AUI_LT` keine explizite Zustandsmaschine besitzt, existiert lediglich ein **impliziter Zustand**:

Zustand	Beschreibung
Bereit	Wartet auf ein Ereignis an IN1.E1 oder IN2.E1. Nach Auslösung wird der Vergleich durchgeführt und das Ergebnis über OUT ausgegeben. Der FB kehrt sofort in den Bereit-Zustand zurück.

Anwendungsszenarien

- **Grenzwertüberwachung**: Prüfen, ob ein Messwert unter einem Schwellwert liegt, z. B. `Temperatur < Grenzwert`.
- **Steuerungslogik**: Bedingte Aktivierung von Ausgängen, z. B. Ventil öffnen, wenn `Druck < Sollwert`.
- **Interlock-Bedingungen**: Sicherheitsabfragen, bei denen ein Betrieb nur erlaubt ist, wenn eine Größe kleiner als eine andere ist.
- **Vergleichsketten**: Kombination mit anderen Vergleichsbausteinen (z. B. `AUI_EQ`, `AUI_GT`) zur Realisierung komplexer logischer Bedingungen.

Vergleich mit ähnlichen Bausteinen

Baustein	Funktion	Schnittstelle	Besonderheit
AUI_LT	Kleiner als (<)	Adapter (AUI, AX)	Asynchron, zwei separate Ereigniseingänge

F_LT	Kleiner als (<)	Standard-Ports (REQ/CNF, ANY_BIT)	Direkte Ereignis-/Datenports, kein Adapter
AUI_EQ	Gleich (=)	Adapter (AUI, AX)	Gleiche Struktur, anderer Vergleichsoperator
AUI_GT	Größer als (>)	Adapter (AUI, AX)	Same structure, different comparison operator

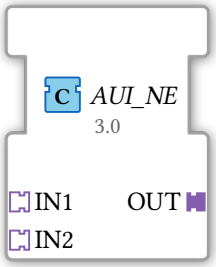
The `AUI_LT` offers particularly flexible integration into IEC 61131-3 compliant systems via its adapter interfaces, while the internal `F_LT` provides the standardized comparison logic.

Conclusion

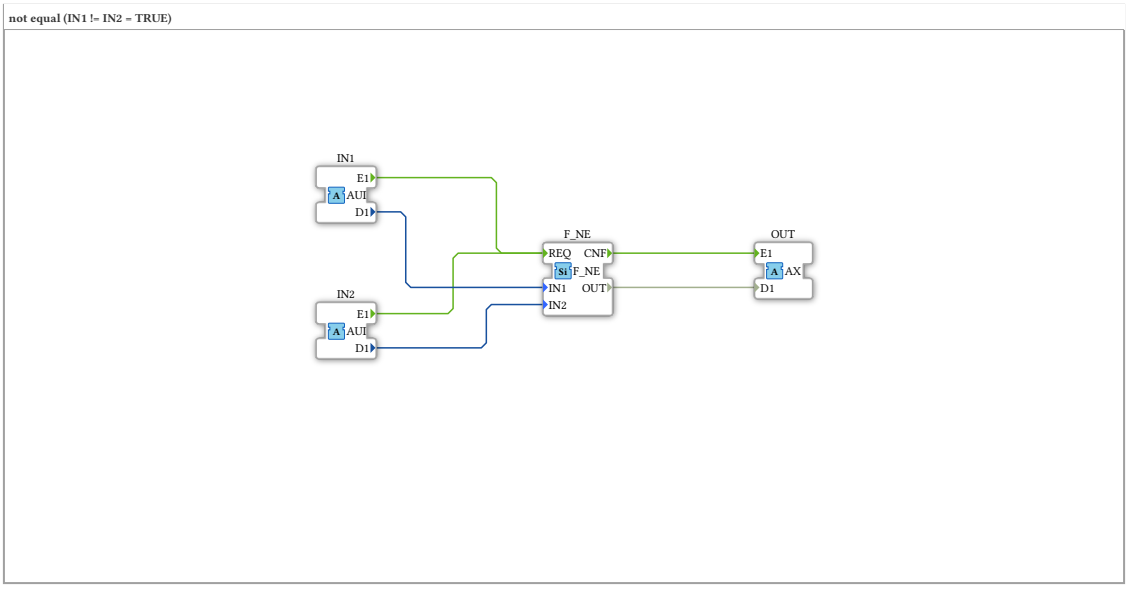
The `AUI_LT` is a compact and reliable function block for less-than comparisons in adapter-based automation environments. Its asynchronous event processing and loose coupling via adapters make it ideal for modular control applications where simple conditions need to be evaluated decentrally and in a resource-efficient manner.

AUI_NE

Input value 1 -- left operand (IN1 != IN2) – *AUI*
Input value 2 -- right operand (IN1 != IN2) – *AUI*



AX – IN1 != IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block `AUI_NE` performs a not equal comparison for two input values. It is built as a composite function block (FB) from the IEC 61131-3 comparison block `F_NE`. Communication with the environment is via standardized adapter interfaces (`AUI` for the inputs, `AX` for the output). This allows it to be seamlessly integrated into adapter-based architectures of the 4diac IDE.

Interface Structure

Event Inputs

The FB does not have direct event inputs. Events are received via the **socket adapters** `IN1` and `IN2`:

Adapter	Event	Description
`IN1`	Event via adapter input E1	Starts the comparison when the value is available at `IN1`.
`IN2`	Event via adapter input E1	Starts the comparison when the value is available at `IN2`.

Both events trigger the activation of the internal `F_NE` module.

Event Outputs

The FB has no direct event outputs. The result is output via the **Plug Adapter** `OUT`:

Adapter	Event	Description
`OUT`	Event via Adapter Output E1	Indicates that the comparison is complete and the result value is valid.

Data Inputs

Data is also provided via the socket adapters:

Adapter	Data Point	Type	Description
`IN1`	D1	ANY (via AUI)	First comparison value
`IN2`	D2	ANY (via AUI)	Second comparison value

The exact data types are determined by the adapter instance used (`adapter::types::unidirectional::AUI`).

Data Outputs

The result value is output via the plug adapter:

Adapter	Data Point	Type	Description
`OUT`	D1	BOOL (via AX)	`TRUE`, if IN1 ≠ IN2; `FALSE` other

Adapter

Adapter	Direction	Type	Description
`IN1`	Socket	`adapter::types::unidirectional::AUI`	Receiver adapter for the first input value.
`IN2`	Socket	`adapter::types::unidirectional::AUI`	Receiver adapter for the second input value.
`OUT`	Plug	`adapter::types::unidirectional::AX`	Output adapter for the comparison result.

Functionality

Each incoming event via `IN1.E1` or `IN2.E1` triggers the internal function block `F_NE` (IEC 61131-3: `NOT EQUAL`). `F_NE` compares the values at data points `IN1.D1` and `IN2.D2`. The result (BOOL) is forwarded to the output adapter `OUT.D1`, and simultaneously an event is sent to `OUT.E1` to signal the result retrieval.

The function block operates **event-driven** – a comparison is only recalculated if at least one of the two input adapters receives an event. This avoids unnecessary processing load.

Technical Features

- **Adapter Encapsulation**: All input/output is handled via standardized adapters, allowing the function block to be integrated into complex adapter networks.

- **Reuse**: Internally, the proven IEC 61131-3 block `F_NE` is used, which implements an efficient, type-independent non-equal comparison.
- **Unidirectional Adapters**: The adapters used (`AUI`, `AX`) are unidirectional – they each support only one data and event direction, which increases the clarity of data flows.
- **No State Storage**: The function block is stateless; each event triggers a recalculation.

State Overview

The FB `AUI_NE` itself does not have its own state machine. Its behavior is entirely determined by the internal `F_NE` block, which implements a simple functional sequence without states. The event-driven control (input ⟶ calculation ⟶ output) is strictly causal.

Application Scenarios

- **Deviation Monitoring**: Check whether two measured values (e.g., temperature, pressure) differ from each other.
- **Threshold Violation**: Combine with other blocks to use inequality as part of a plausibility check.
- **Data Validation**: Detect whether two redundant signals provide different values (comparison of copies).
- **Event-Driven Comparisons**: Embedding in applications that only require a comparison when one of the inputs changes.

Comparison with Similar Blocks

Block	Comparison Function	Special Feature
`AUI_EQ`	Equal	Result `TRUE` for equal values
`AUI_NE`	Not equal	Result `TRUE` for different values
`AUI_GT`	Greater than	Result `TRUE` if IN1 > IN2
`AUI_LT`	Less than	Result `TRUE` if IN1 < IN2

All the aforementioned components share the same adapter interface and can therefore be easily exchanged or combined.

Conclusion

The `AUI_NE` is a simple yet reliable function block for comparing the difference between two values within an adapter-based 4diac environment. Its strengths lie in its clean encapsulation via adapters, event-driven execution, and high reusability thanks to the underlying IEC 61131-3 component. It is particularly well-suited for modular, expandable automation solutions.

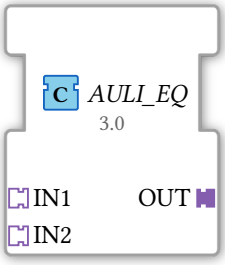
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

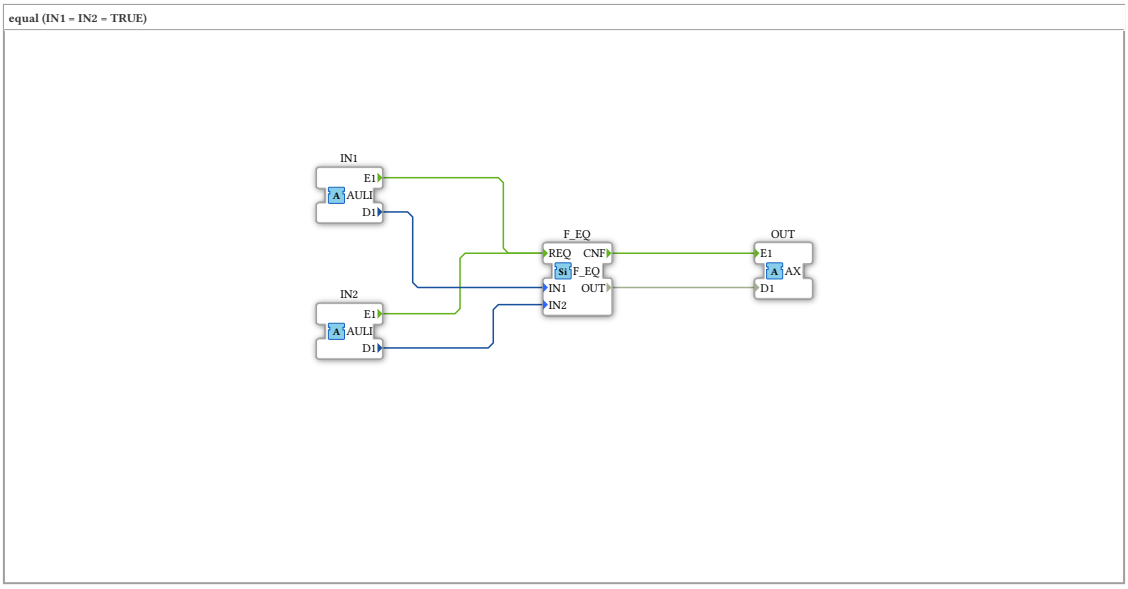
AULI_EQ



Input value 1 -- left operand (IN1 = IN2) – *AULI*

Input value 2 -- right operand (IN1 = IN2) – *AULI*

AX – IN1 = IN2 = TRUE



• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_EQ** compares two values of type `ULINT` (64-bit unsigned integer) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `ULINT`, 64-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AULI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AULI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AULI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_EQ` (type `iec61131::comparison::F_EQ`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AULI/AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the `ULINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and quantity values against a reference or threshold value of type `ULINT`.
- **Control Logic**: Triggering an action once two values of type `ULINT` satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `ULINT`-based comparison condition.

Comparison with Similar Function Blocks

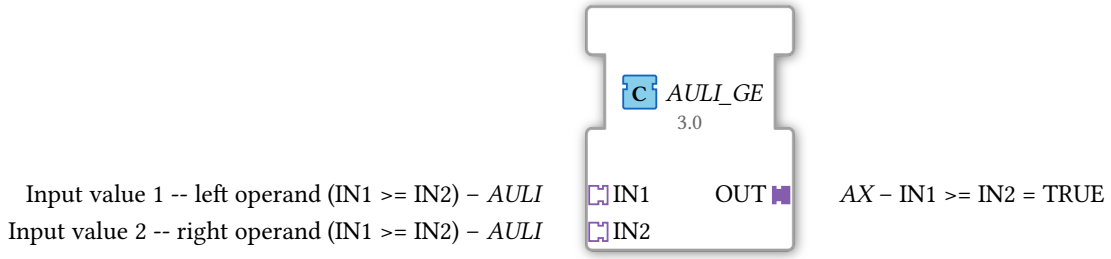
- **AULI_NE**: Checks for “not equal to” – the inverse function of AULI_EQ.
- **AULI_NE**: Checks for inequality of values.

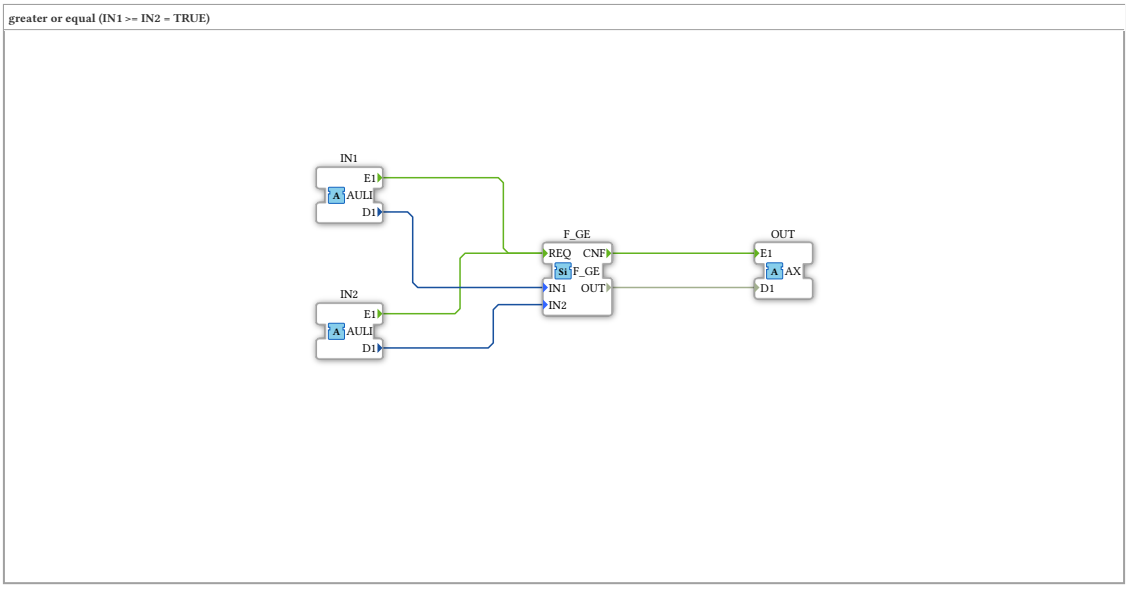
AULI_EQ is part of the complete AULI comparison family (EQ/NE/GT/GE/LT/LE) for the `ULINT` data type.

Conclusion

AULI_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the `ULINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AULI_GE





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_GE** compares two values of type `ULINT` (64-bit unsigned integer) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `ULINT`, 64-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AULI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AULI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AULI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GE` (type `iec61131::comparison::F_GE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AULI/AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the `ULINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and quantity values against a reference or threshold value of type `ULINT`.
- **Control Logic**: Triggering an action once two values of type `ULINT` satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `ULINT`-based comparison condition.

Comparison with Similar Function Blocks

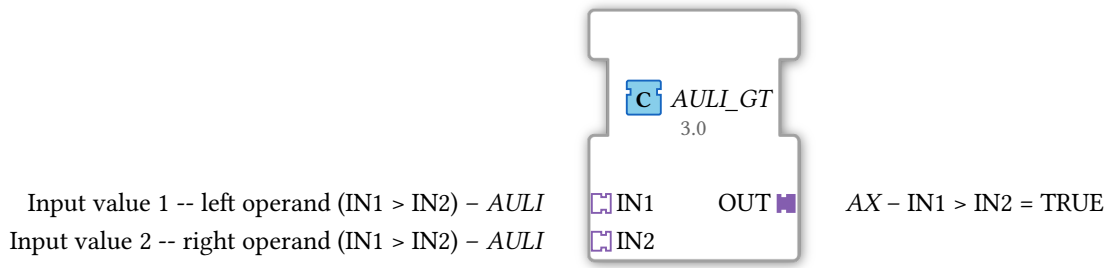
- **AULI_LE**: Checks for “less than or equal to” – the inverse function of AULI_GE.
- **AULI_EQ**: Checks for equality of values.
- **AULI_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AULI_GE is the extended version.

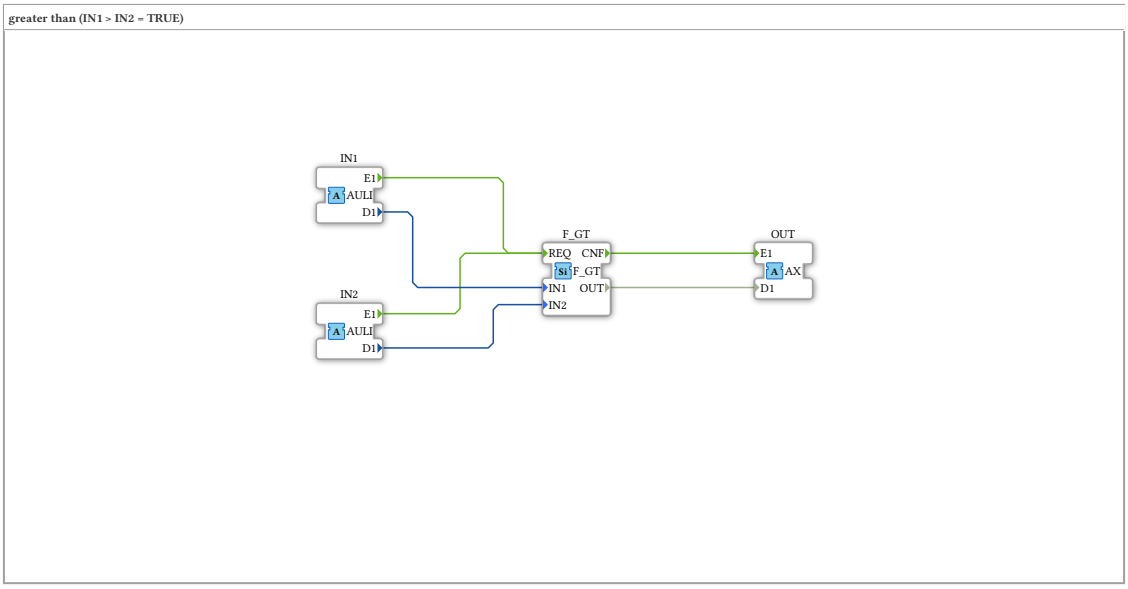
AULI_GE completes the AULI comparison family with the boundary case of equality.

Conclusion

AULI_GE is a simple yet important comparison block for IEC 61131-3 control applications using the `ULINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AULI_GT





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_GT** compares two values of type `ULINT` (64-bit unsigned integer) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `ULINT`, 64-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AULI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AULI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AULI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GT` (type `iec61131::comparison::F_GT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AULI/AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the `ULINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and quantity values against a reference or threshold value of type `ULINT`.
- **Control Logic**: Triggering an action once two values of type `ULINT` satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `ULINT`-based comparison condition.

Comparison with Similar Function Blocks

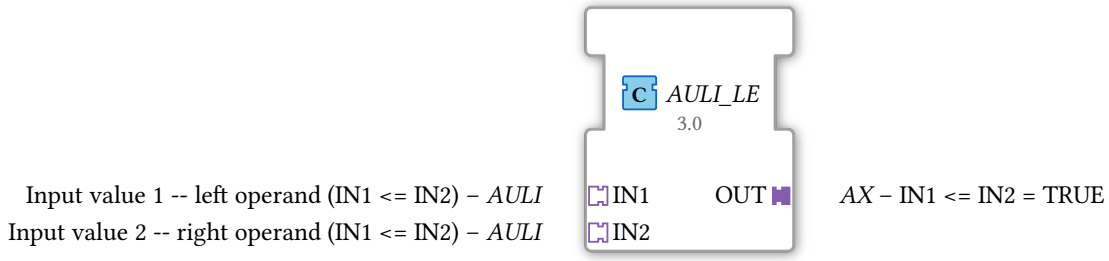
- **AULI_LT**: Checks for “less than” – the inverse function of AULI_GT.
- **AULI_EQ**: Checks for equality of values.
- **AULI_GE**: Performs the “greater than or equal to” comparison – the extended version of AULI_GT that includes the equality case.

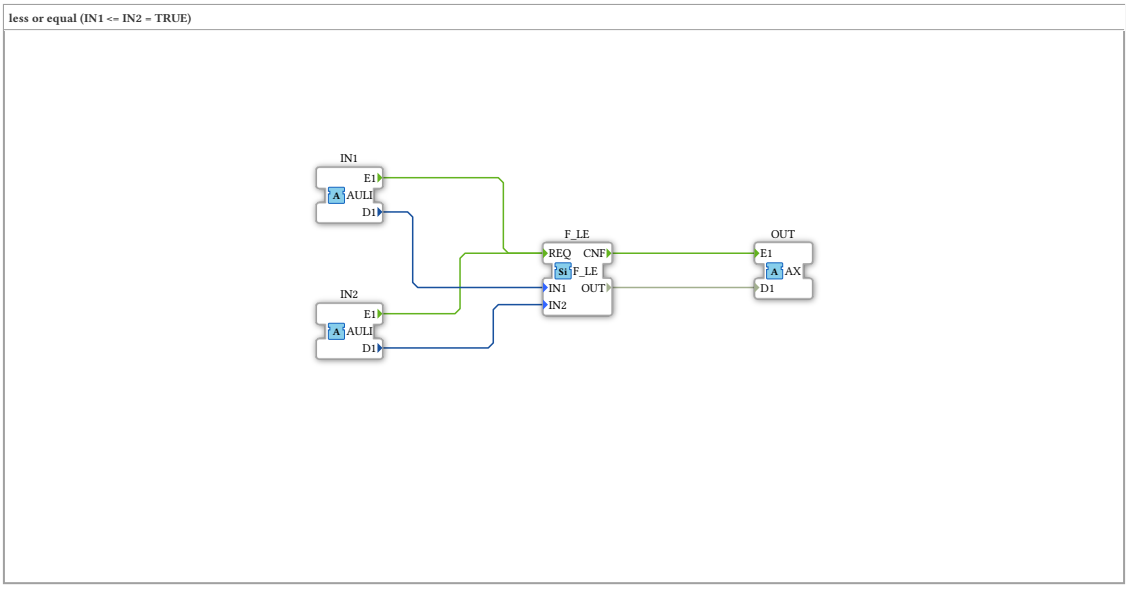
AULI_GT is the basic “greater than” component without the equality case, complemented by AULI_GE.

Conclusion

AULI_GT is a simple yet important comparison block for IEC 61131-3 control applications using the `ULINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AULI_LE





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_LE** compares two values of type `ULINT` (64-bit unsigned integer) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `ULINT`, 64-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AULI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AULI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AULI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LE` (type `iec61131::comparison::F_LE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AULI/AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the `ULINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and quantity values against a reference or threshold value of type `ULINT`.
- **Control Logic**: Triggering an action once two values of type `ULINT` satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `ULINT`-based comparison condition.

Comparison with Similar Function Blocks

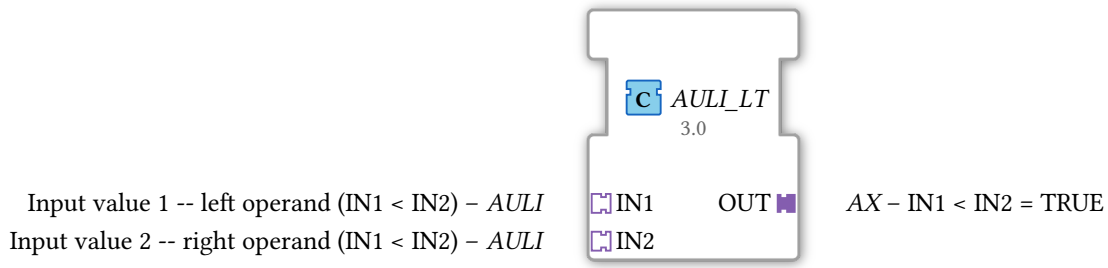
- **AULI_GE**: Checks for “greater than or equal to” – the inverse function of AULI_LE.
- **AULI_EQ**: Checks for equality of values.
- **AULI_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AULI_LE is the extended version.

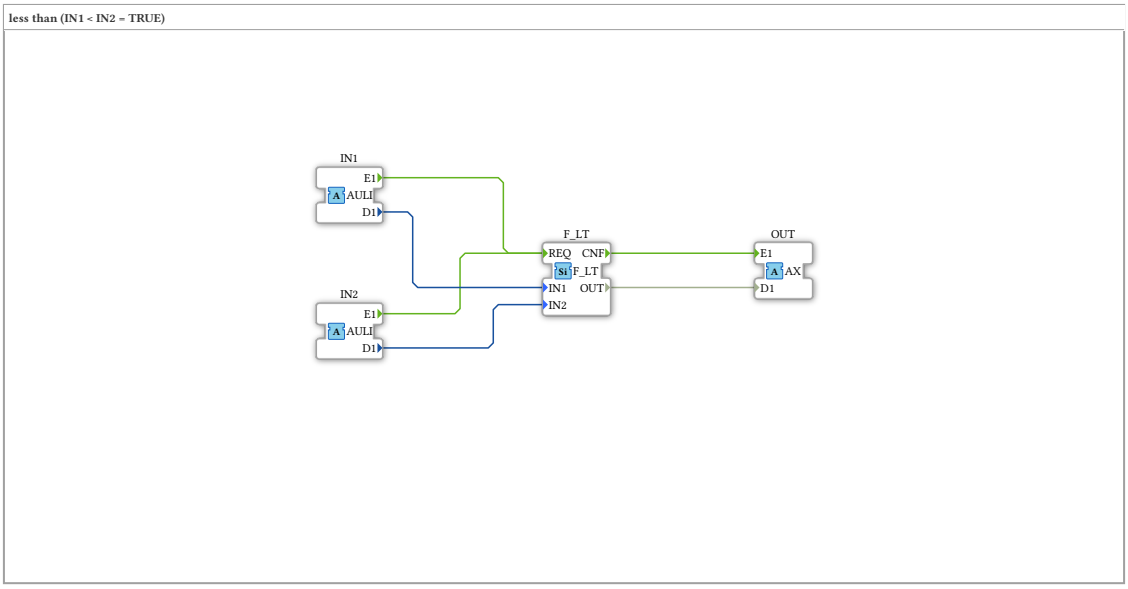
AULI_LE completes the AULI comparison family with the boundary case of equality.

Conclusion

AULI_LE is a simple yet important comparison block for IEC 61131-3 control applications using the `ULINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AULI_LT





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_LT** compares two values of type `ULINT` (64-bit unsigned integer) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `ULINT`, 64-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AULI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AULI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AULI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LT` (type `iec61131::comparison::F_LT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AULI/AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the `ULINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and quantity values against a reference or threshold value of type `ULINT`.
- **Control Logic**: Triggering an action once two values of type `ULINT` satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `ULINT`-based comparison condition.

Comparison with Similar Function Blocks

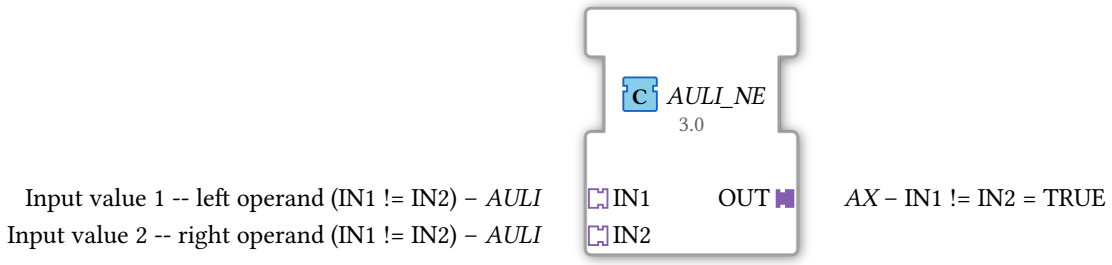
- **AULI_GT**: Checks for “greater than” – the inverse function of AULI_LT.
- **AULI_EQ**: Checks for equality of values.
- **AULI_LE**: Performs the “less than or equal to” comparison – the extended version of AULI_LT that includes the equality case.

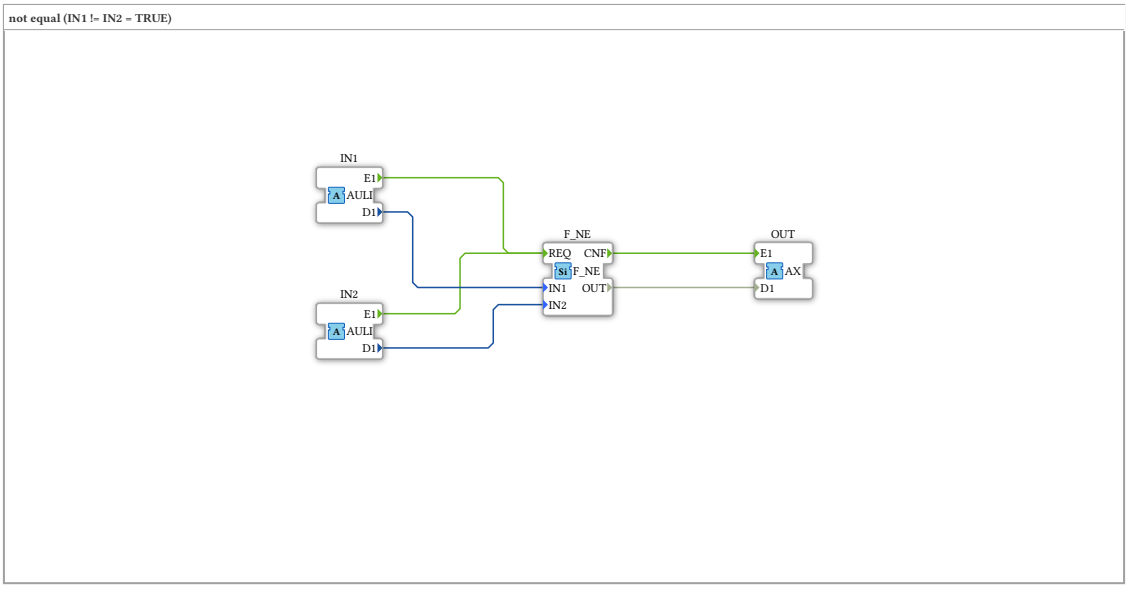
AULI_LT is the basic “less than” component without the equality case, complemented by AULI_LE.

Conclusion

AULI_LT is a simple yet important comparison block for IEC 61131-3 control applications using the `ULINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AULI_NE





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_NE** compares two values of type `ULINT` (64-bit unsigned integer) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `ULINT`, 64-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AULI` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AULI` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AULI` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_NE` (type `iec61131::comparison::F_NE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AULI/AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the `ULINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking large counter and quantity values against a reference or threshold value of type `ULINT`.
- **Control Logic**: Triggering an action once two values of type `ULINT` satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `ULINT`-based comparison condition.

Comparison with Similar Function Blocks

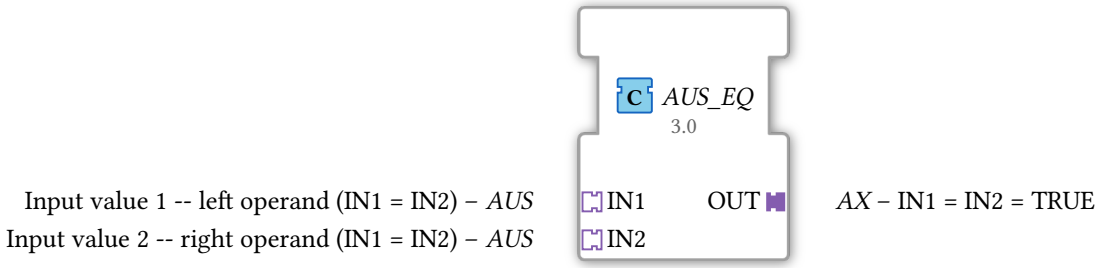
- **AULI_EQ**: Checks for “equal to” – the inverse function of AULI_NE.
- **AULI_EQ**: Checks for equality of values.

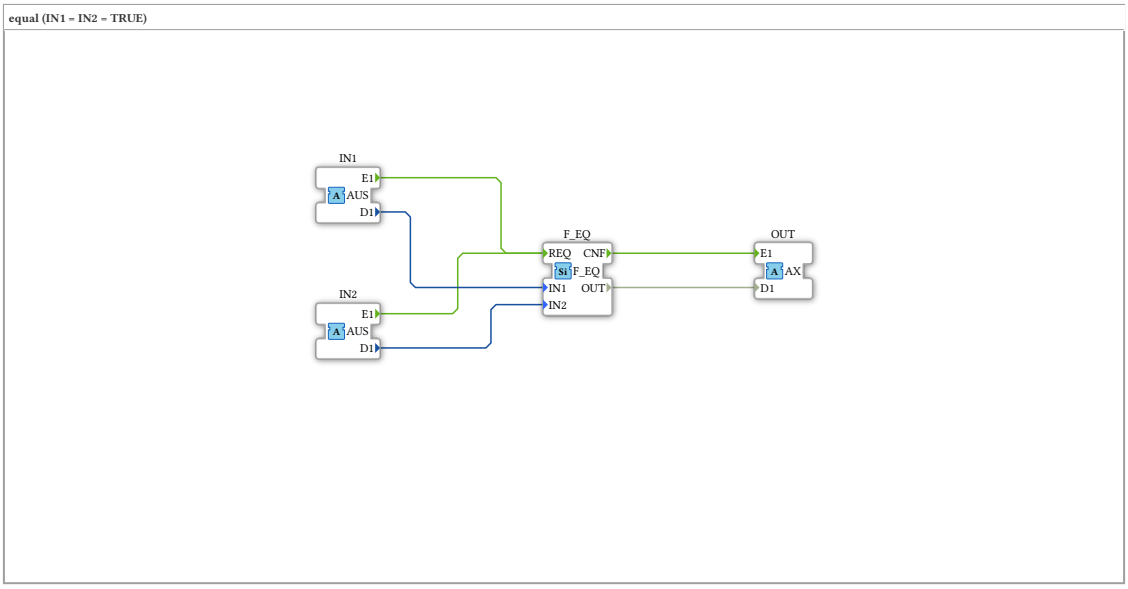
AULI_NE is part of the complete AULI comparison family (EQ/NE/GT/GE/LT/LE) for the `ULINT` data type.

Conclusion

AULI_NE is a simple yet important comparison block for IEC 61131-3 control applications using the `ULINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AUS_EQ





• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_EQ** compares two values of type `USINT` (8-bit unsigned integer) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `USINT`, 8-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AUS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AUS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AUS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_EQ` (type `iec61131::comparison::F_EQ`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AUS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the `USINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type `USINT`.
- **Control Logic**: Triggering an action once two values of type `USINT` satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `USINT`-based comparison condition.

Comparison with Similar Function Blocks

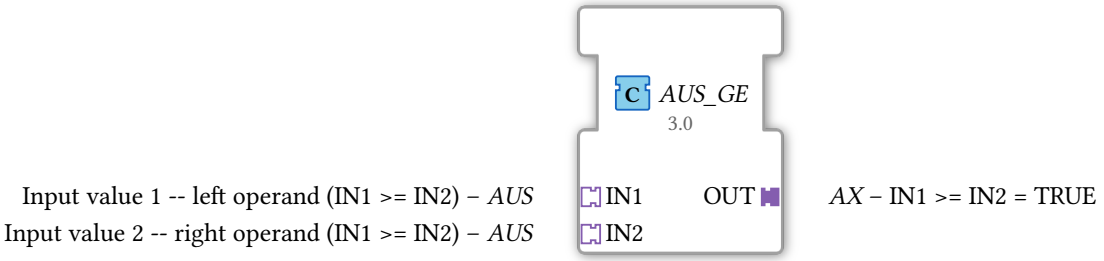
- **AUS_NE**: Checks for “not equal to” – the inverse function of AUS_EQ.
- **AUS_NE**: Checks for inequality of values.

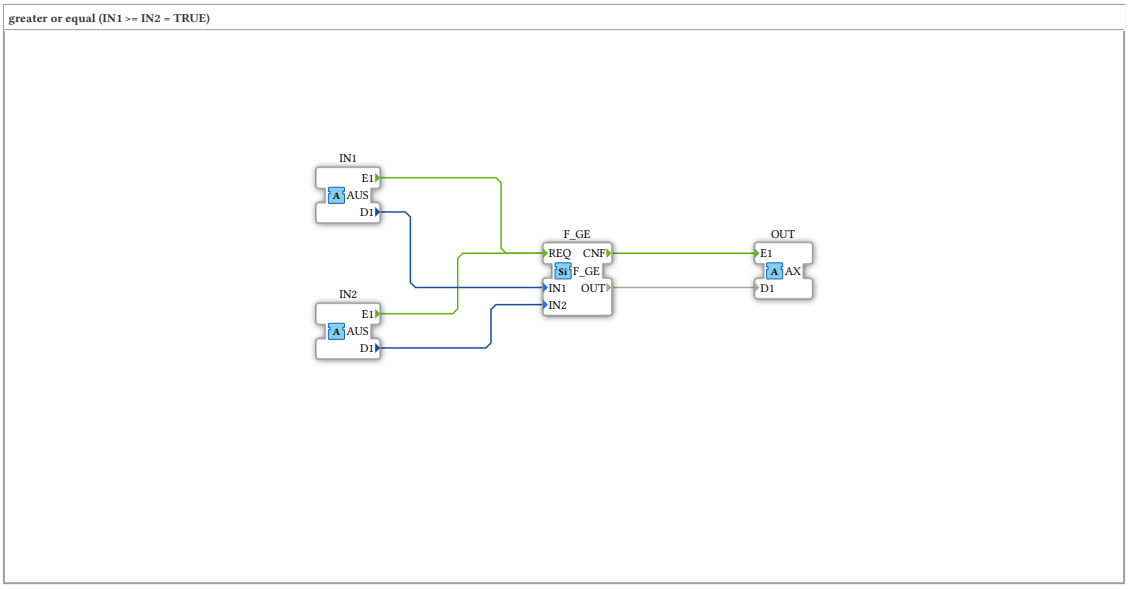
AUS_EQ is part of the complete AUS comparison family (EQ/NE/GT/GE/LT/LE) for the `USINT` data type.

Conclusion

AUS_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the `USINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AUS_GE





• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_GE** compares two values of type `USINT` (8-bit unsigned integer) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `USINT`, 8-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AUS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AUS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AUS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GE` (type `iec61131::comparison::F_GE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AUS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the `USINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type `USINT`.
- **Control Logic**: Triggering an action once two values of type `USINT` satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `USINT`-based comparison condition.

Comparison with Similar Function Blocks

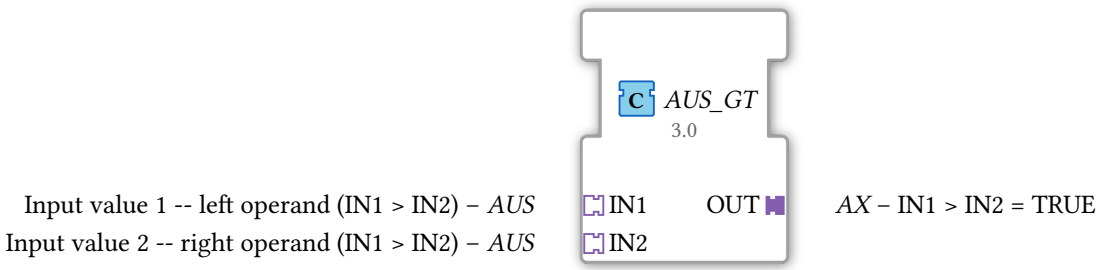
- **AUS_LE**: Checks for “less than or equal to” – the inverse function of AUS_GE.
- **AUS_EQ**: Checks for equality of values.
- **AUS_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AUS_GE is the extended version.

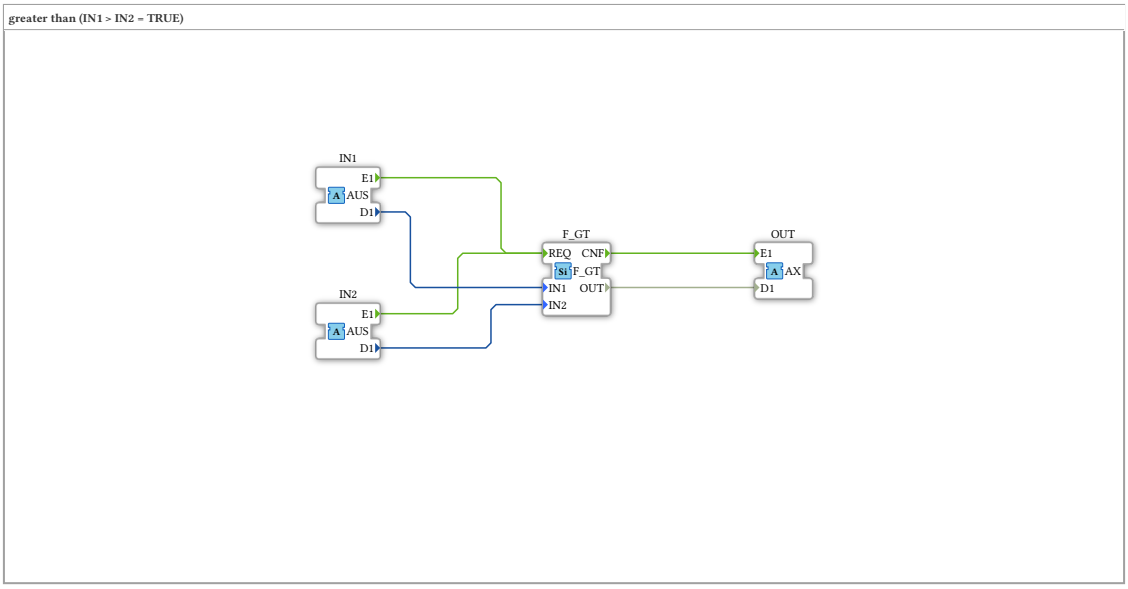
AUS_GE completes the AUS comparison family with the boundary case of equality.

Conclusion

AUS_GE is a simple yet important comparison block for IEC 61131-3 control applications using the `USINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AUS_GT





• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_GT** compares two values of type `USINT` (8-bit unsigned integer) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `USINT`, 8-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AUS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AUS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AUS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GT` (type `iec61131::comparison::F_GT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AUS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the `USINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type `USINT`.
- **Control Logic**: Triggering an action once two values of type `USINT` satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `USINT`-based comparison condition.

Comparison with Similar Function Blocks

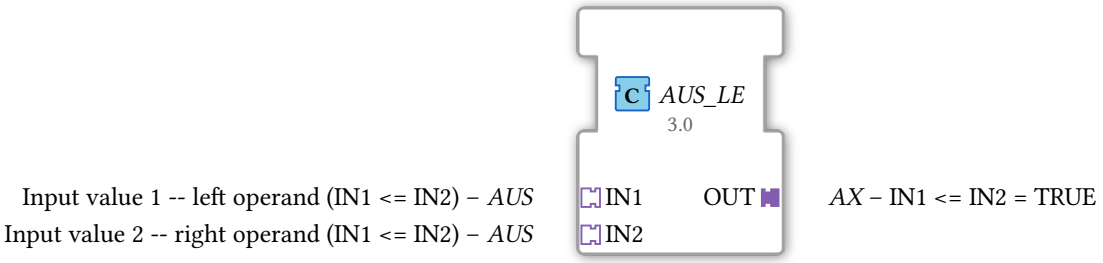
- **AUS_LT**: Checks for “less than” – the inverse function of AUS_GT.
- **AUS_EQ**: Checks for equality of values.
- **AUS_GE**: Performs the “greater than or equal to” comparison – the extended version of AUS_GT that includes the equality case.

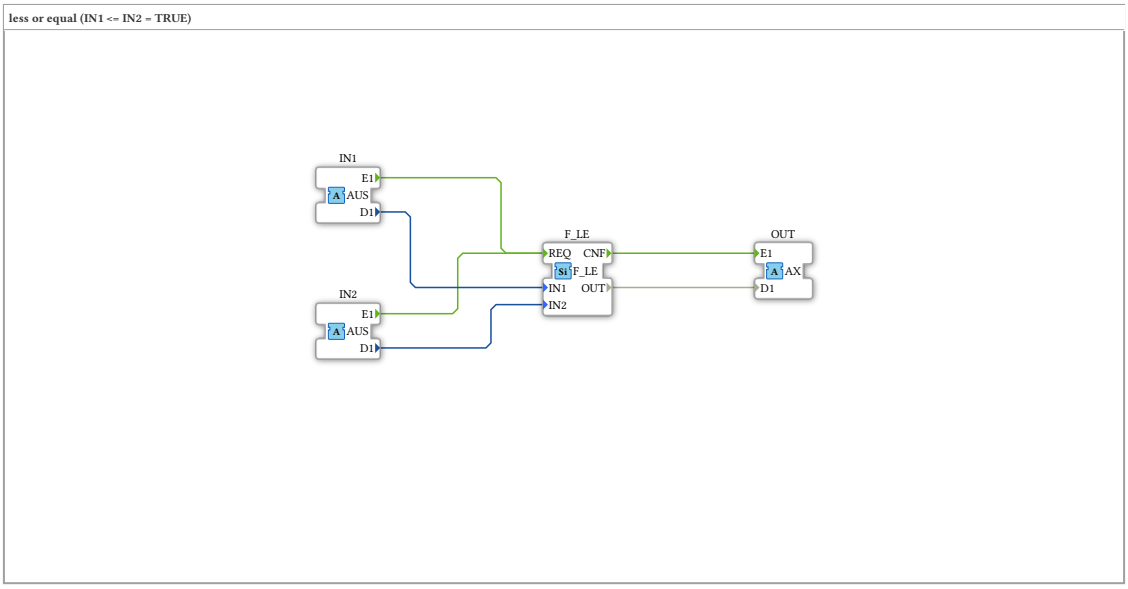
AUS_GT is the basic “greater than” component without the equality case, complemented by AUS_GE.

Conclusion

AUS_GT is a simple yet important comparison block for IEC 61131-3 control applications using the `USINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AUS_LE





• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_LE** compares two values of type `USINT` (8-bit unsigned integer) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `USINT`, 8-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AUS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AUS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AUS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LE` (type `iec61131::comparison::F_LE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AUS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the `USINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type `USINT`.
- **Control Logic**: Triggering an action once two values of type `USINT` satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `USINT`-based comparison condition.

Comparison with Similar Function Blocks

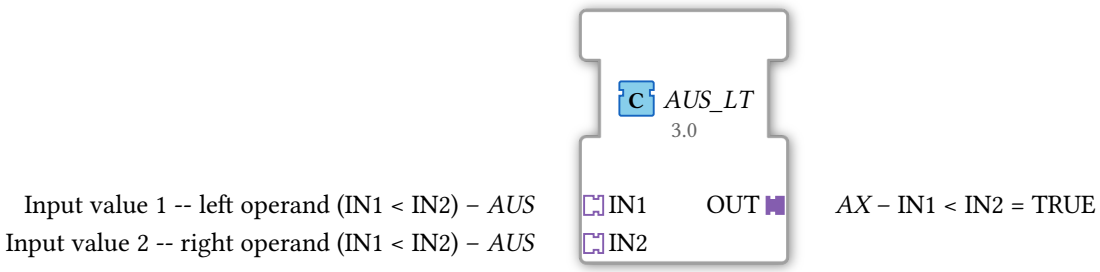
- **AUS_GE**: Checks for “greater than or equal to” – the inverse function of AUS_LE.
- **AUS_EQ**: Checks for equality of values.
- **AUS_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AUS_LE is the extended version.

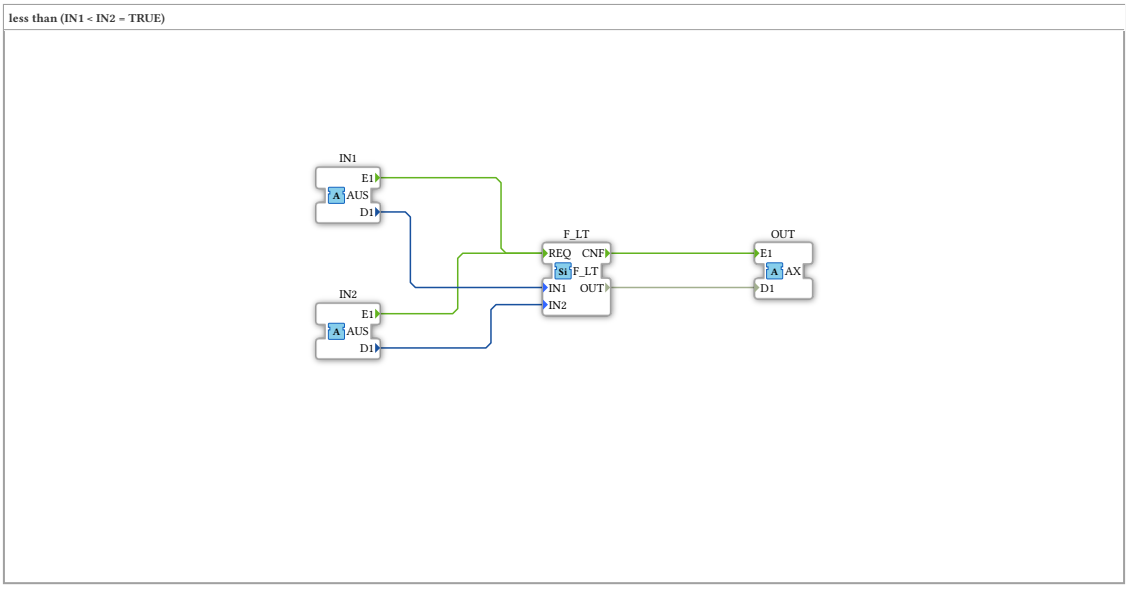
AUS_LE completes the AUS comparison family with the boundary case of equality.

Conclusion

AUS_LE is a simple yet important comparison block for IEC 61131-3 control applications using the `USINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AUS_LT





• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_LT** compares two values of type `USINT` (8-bit unsigned integer) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `USINT`, 8-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AUS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AUS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AUS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LT` (type `iec61131::comparison::F_LT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AUS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the `USINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type `USINT`.
- **Control Logic**: Triggering an action once two values of type `USINT` satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `USINT`-based comparison condition.

Comparison with Similar Function Blocks

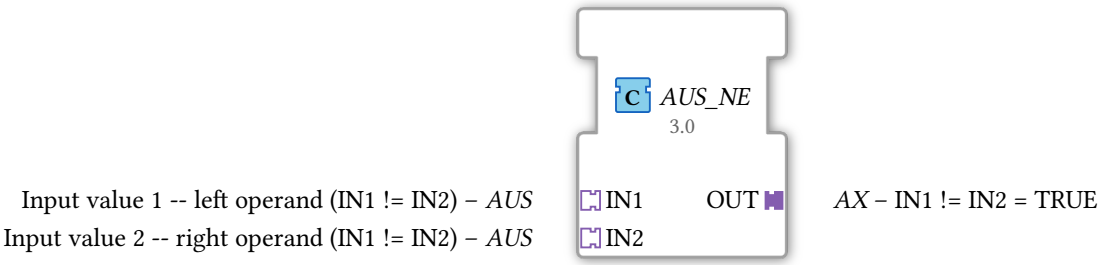
- **AUS_GT**: Checks for “greater than” – the inverse function of AUS_LT.
- **AUS_EQ**: Checks for equality of values.
- **AUS_LE**: Performs the “less than or equal to” comparison – the extended version of AUS_LT that includes the equality case.

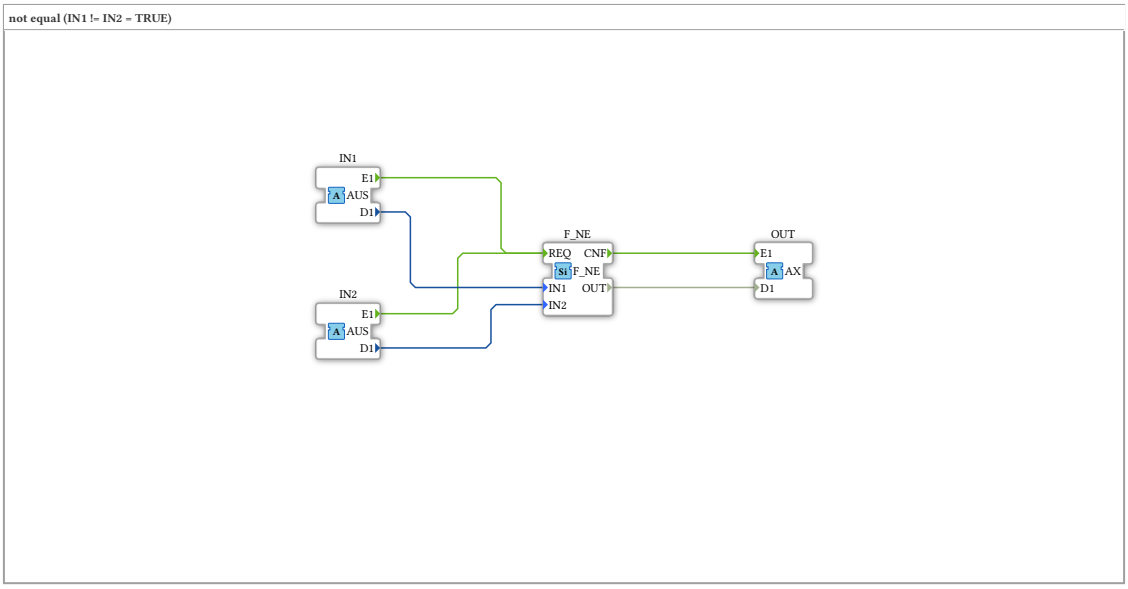
AUS_LT is the basic “less than” component without the equality case, complemented by AUS_LE.

Conclusion

AUS_LT is a simple yet important comparison block for IEC 61131-3 control applications using the `USINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AUS_NE





• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_NE** compares two values of type `USINT` (8-bit unsigned integer) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The comparison follows the usual numeric ordering.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `USINT`, 8-bit unsigned integer).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AUS` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AUS` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AUS` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_NE` (type `iec61131::comparison::F_NE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AUS`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the `USINT` data type.
- **Comparison logic**: The comparison follows the usual numeric ordering.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking small counter and index values against a reference or threshold value of type `USINT`.
- **Control Logic**: Triggering an action once two values of type `USINT` satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `USINT`-based comparison condition.

Comparison with Similar Function Blocks

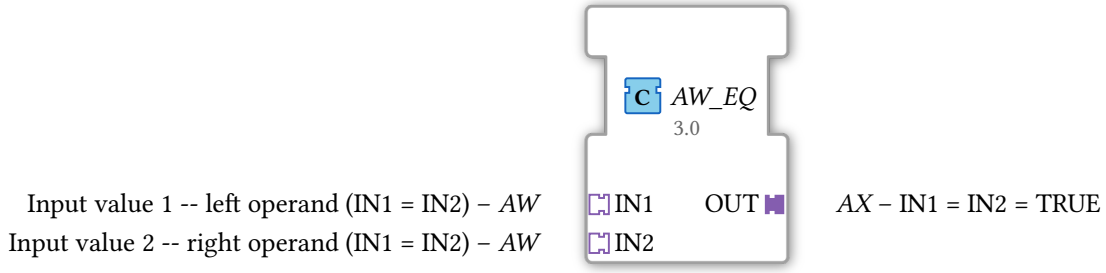
- **AUS_EQ**: Checks for “equal to” – the inverse function of AUS_NE.
- **AUS_EQ**: Checks for equality of values.

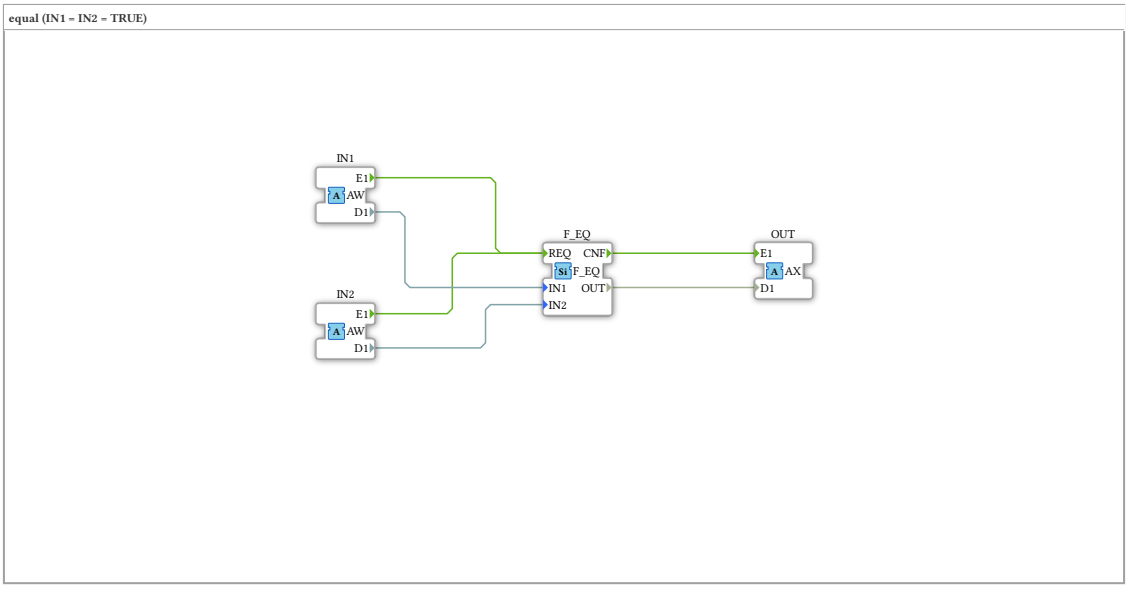
AUS_NE is part of the complete AUS comparison family (EQ/NE/GT/GE/LT/LE) for the `USINT` data type.

Conclusion

AUS_NE is a simple yet important comparison block for IEC 61131-3 control applications using the `USINT` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AW_EQ





• – ○ • – ○ • – ○ •

Introduction

The function block **AW_EQ** compares two values of type `WORD` (16-bit bit pattern (word)) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WORD`, 16-bit bit pattern (word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AW` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AW` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AW` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_EQ` (type `iec61131::comparison::F_EQ`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AW`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“equal to”) for the `WORD` data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status words against a reference or threshold value of type `WORD`.
- **Control Logic**: Triggering an action once two values of type `WORD` satisfy the “equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WORD`-based comparison condition.

Comparison with Similar Function Blocks

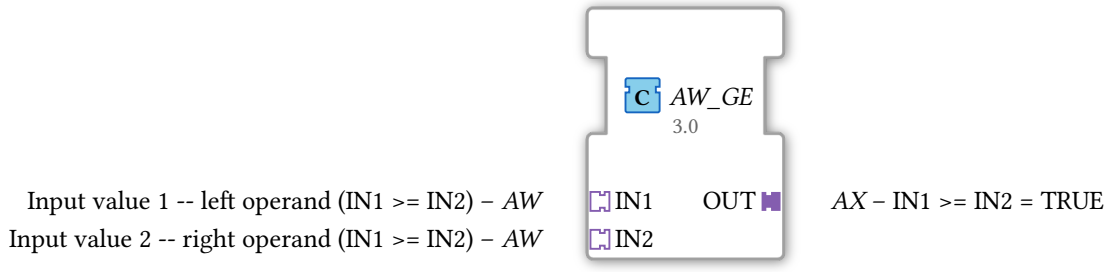
- **AW_NE**: Checks for “not equal to” – the inverse function of AW_EQ.
- **AW_NE**: Checks for inequality of values.

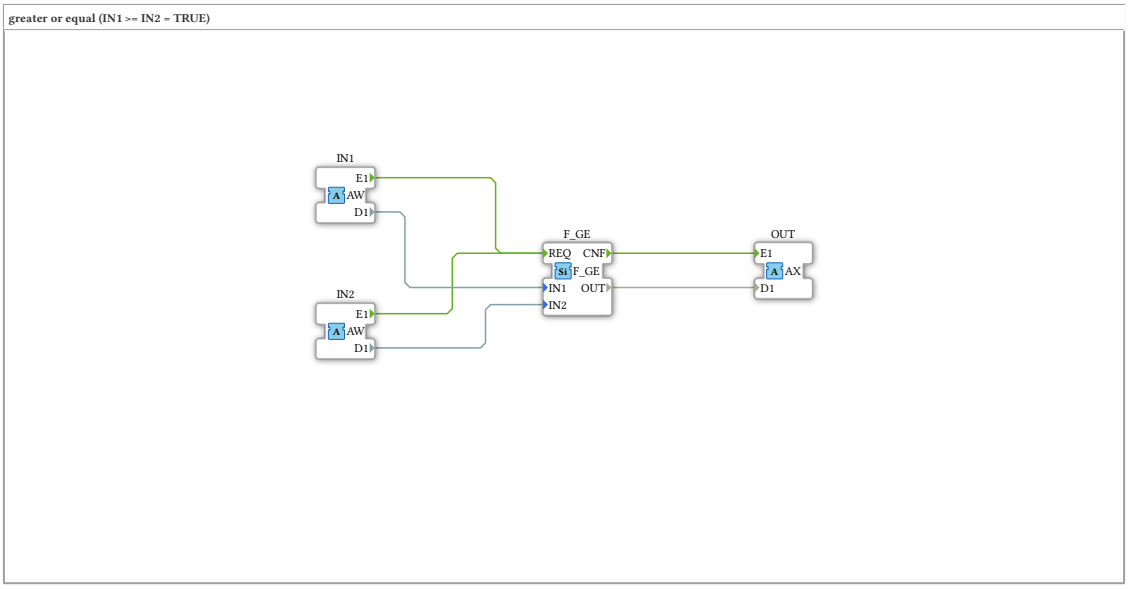
AW_EQ is part of the complete AW comparison family (EQ/NE/GT/GE/LT/LE) for the `WORD` data type.

Conclusion

AW_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the `WORD` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AW_GE





• – ○ • – ○ • – ○ •

Introduction

The function block **AW_GE** compares two values of type `WORD` (16-bit bit pattern (word)) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WORD`, 16-bit bit pattern (word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AW` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AW` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AW` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GE` (type `iec61131::comparison::F_GE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AW`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the `WORD` data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status words against a reference or threshold value of type `WORD`.
- **Control Logic**: Triggering an action once two values of type `WORD` satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WORD`-based comparison condition.

Comparison with Similar Function Blocks

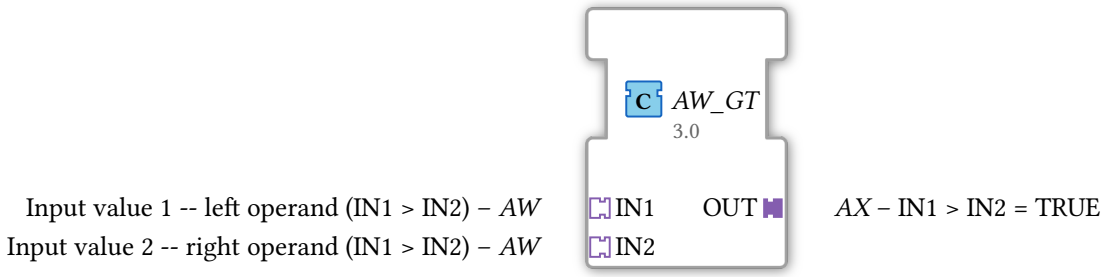
- **AW_LE**: Checks for “less than or equal to” – the inverse function of AW_GE.
- **AW_EQ**: Checks for equality of values.
- **AW_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AW_GE is the extended version.

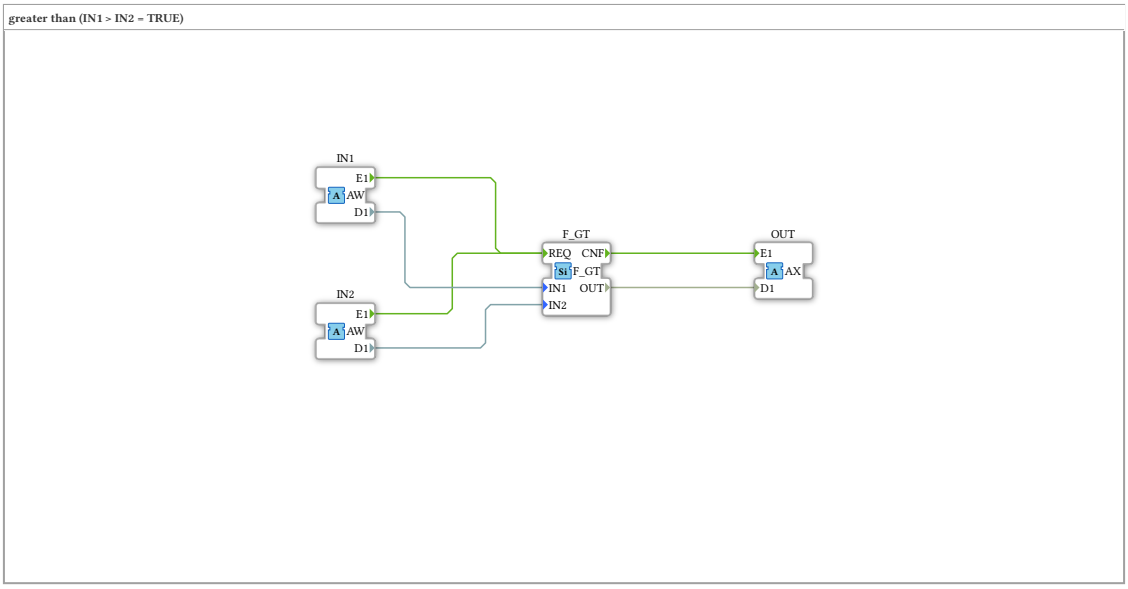
AW_GE completes the AW comparison family with the boundary case of equality.

Conclusion

AW_GE is a simple yet important comparison block for IEC 61131-3 control applications using the `WORD` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AW_GT





• – ○ • – ○ • – ○ •

Introduction

The function block **AW_GT** compares two values of type `WORD` (16-bit bit pattern (word)) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WORD`, 16-bit bit pattern (word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AW` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AW` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AW` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GT` (type `iec61131::comparison::F_GT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AW`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the `WORD` data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status words against a reference or threshold value of type `WORD`.
- **Control Logic**: Triggering an action once two values of type `WORD` satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WORD`-based comparison condition.

Comparison with Similar Function Blocks

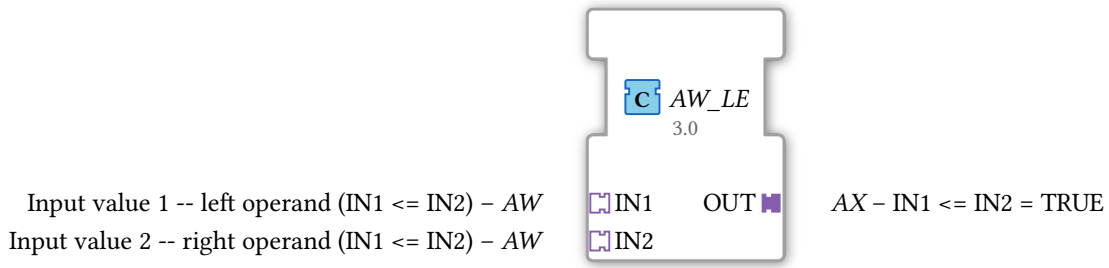
- **AW_LT**: Checks for “less than” – the inverse function of AW_GT.
- **AW_EQ**: Checks for equality of values.
- **AW_GE**: Performs the “greater than or equal to” comparison – the extended version of AW_GT that includes the equality case.

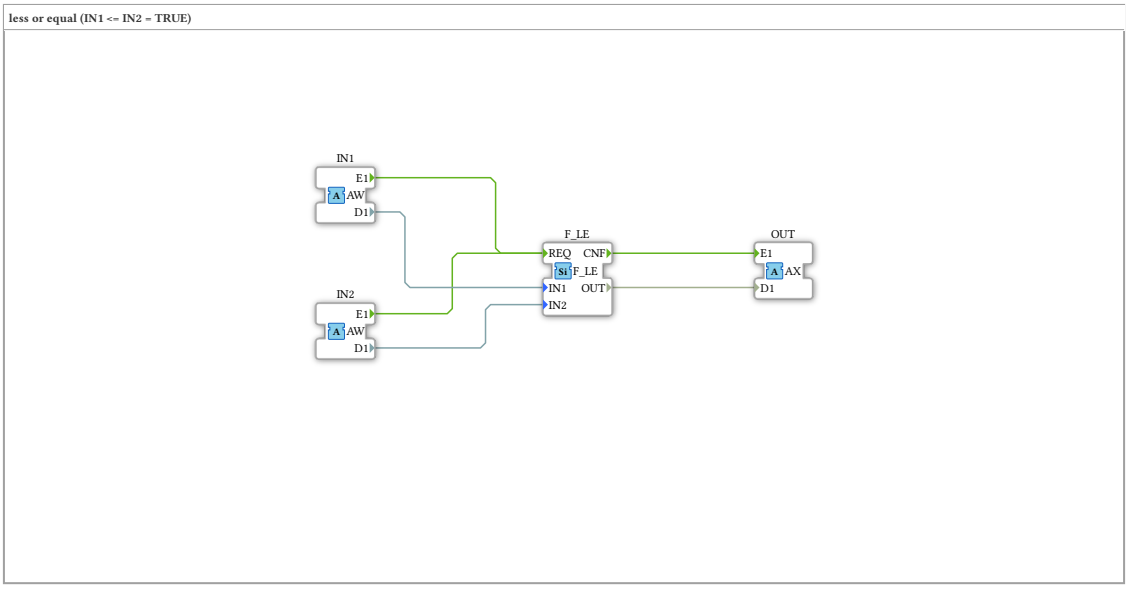
AW_GT is the basic “greater than” component without the equality case, complemented by AW_GE.

Conclusion

AW_GT is a simple yet important comparison block for IEC 61131-3 control applications using the `WORD` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AW_LE





• – ○ • – ○ • – ○ •

Introduction

The function block **AW_LE** compares two values of type `WORD` (16-bit bit pattern (word)) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WORD`, 16-bit bit pattern (word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AW` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AW` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AW` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LE` (type `iec61131::comparison::F_LE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AW`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the `WORD` data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status words against a reference or threshold value of type `WORD`.
- **Control Logic**: Triggering an action once two values of type `WORD` satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WORD`-based comparison condition.

Comparison with Similar Function Blocks

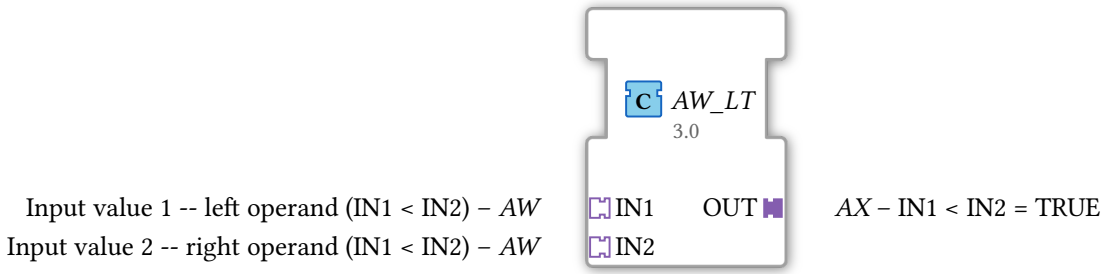
- **AW_GE**: Checks for “greater than or equal to” – the inverse function of AW_LE.
- **AW_EQ**: Checks for equality of values.
- **AW_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AW_LE is the extended version.

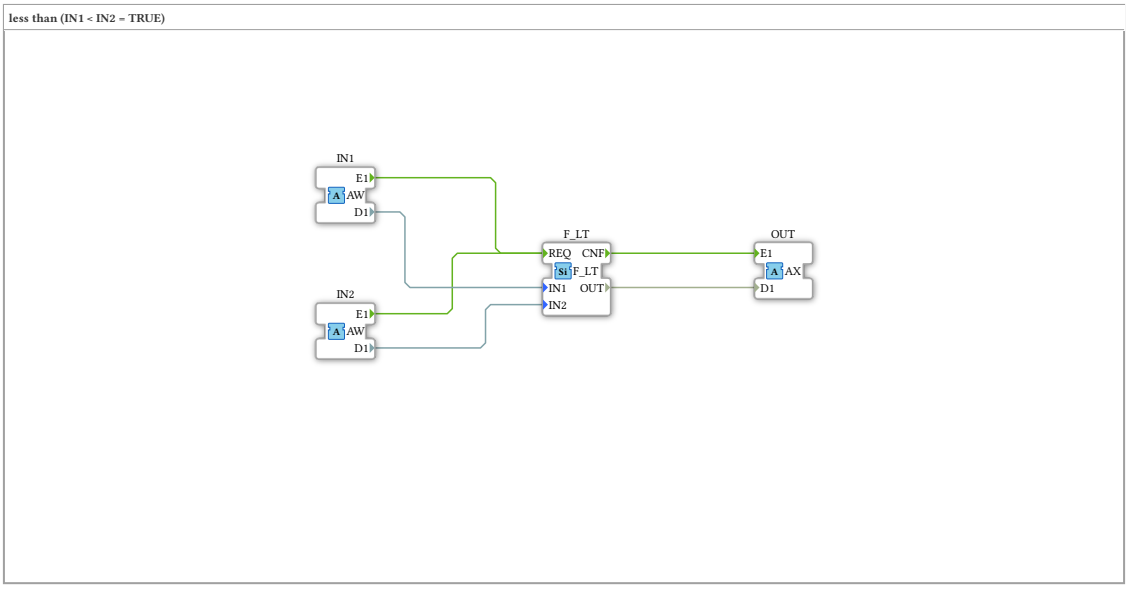
AW_LE completes the AW comparison family with the boundary case of equality.

Conclusion

AW_LE is a simple yet important comparison block for IEC 61131-3 control applications using the `WORD` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AW_LT





• – ○ • – ○ • – ○ •

Introduction

The function block **AW_LT** compares two values of type `WORD` (16-bit bit pattern (word)) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WORD`, 16-bit bit pattern (word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AW` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AW` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AW` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LT` (type `iec61131::comparison::F_LT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AW`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the `WORD` data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status words against a reference or threshold value of type `WORD`.
- **Control Logic**: Triggering an action once two values of type `WORD` satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WORD`-based comparison condition.

Comparison with Similar Function Blocks

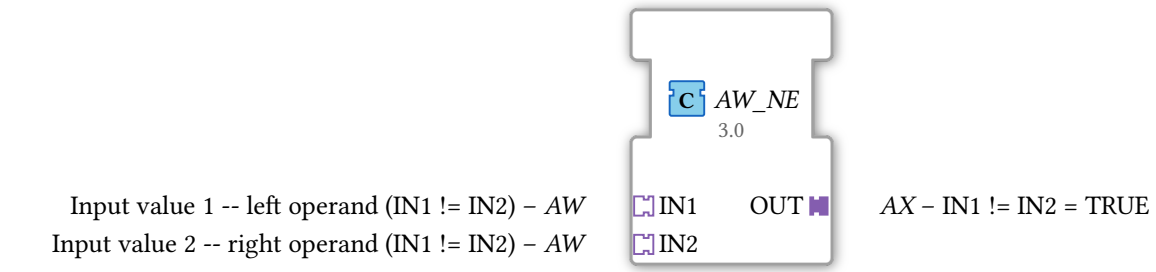
- **AW_GT**: Checks for “greater than” – the inverse function of AW_LT.
- **AW_EQ**: Checks for equality of values.
- **AW_LE**: Performs the “less than or equal to” comparison – the extended version of AW_LT that includes the equality case.

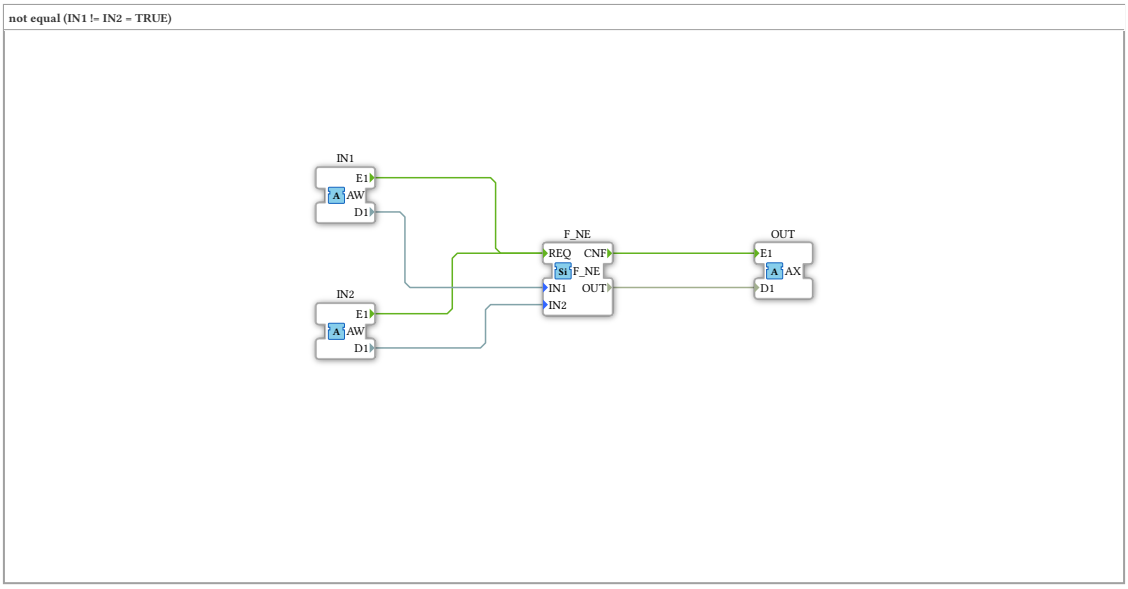
AW_LT is the basic “less than” component without the equality case, complemented by AW_LE.

Conclusion

AW_LT is a simple yet important comparison block for IEC 61131-3 control applications using the `WORD` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AW_NE





• – ○ • – ○ • – ○ •

Introduction

The function block **AW_NE** compares two values of type `WORD` (16-bit bit pattern (word)) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. The bit patterns are interpreted as unsigned integers and compared numerically.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `WORD`, 16-bit bit pattern (word)).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AW` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AW` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AW` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_NE` (type `iec61131::comparison::F_NE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AW`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the `WORD` data type.
- **Comparison logic**: The bit patterns are interpreted as unsigned integers and compared numerically.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking status words against a reference or threshold value of type `WORD`.
- **Control Logic**: Triggering an action once two values of type `WORD` satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `WORD`-based comparison condition.

Comparison with Similar Function Blocks

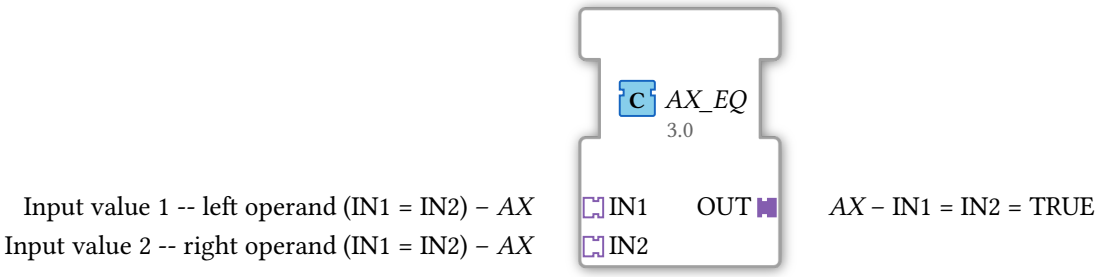
- **AW_EQ**: Checks for “equal to” – the inverse function of AW_NE.
- **AW_EQ**: Checks for equality of values.

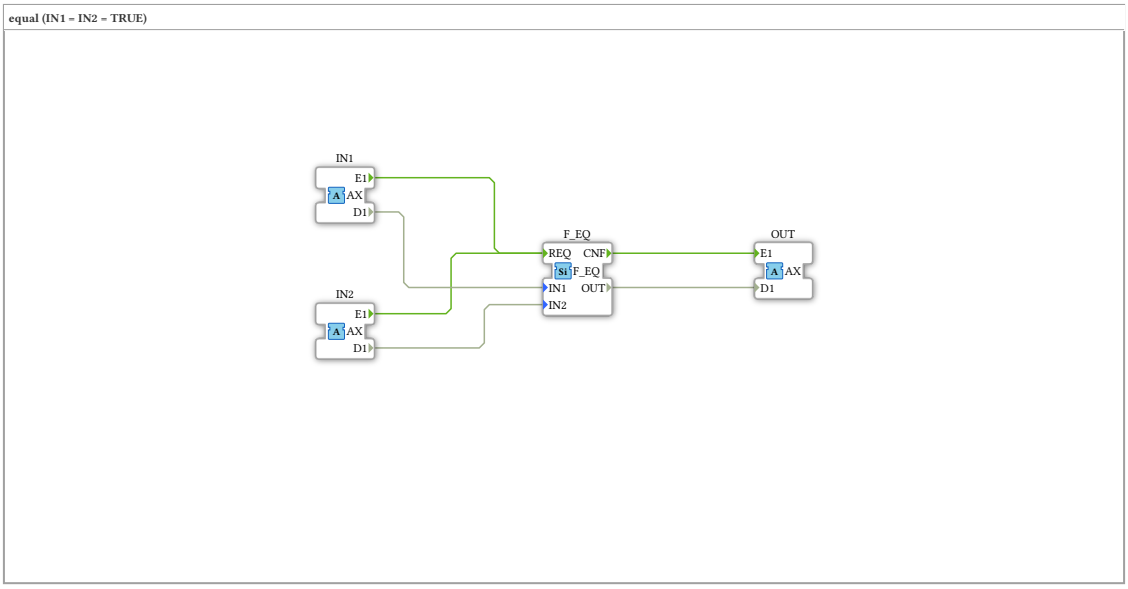
AW_NE is part of the complete AW comparison family (EQ/NE/GT/GE/LT/LE) for the `WORD` data type.

Conclusion

AW_NE is a simple yet important comparison block for IEC 61131-3 control applications using the `WORD` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AX_EQ





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_EQ** compares two values of type `BOOL` (Boolean value) and checks whether the first value (IN1) is equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. `FALSE` is considered the smaller and `TRUE` the greater value.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BOOL`, Boolean value).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 = IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AX` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AX` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AX` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_EQ` (type `iec61131::comparison::F_EQ`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AX`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 ("equal to") for the `BOOL` data type.
- **Comparison logic**: `FALSE` is considered the smaller and `TRUE` the greater value.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking enable and state bits against a reference or threshold value of type `BOOL`.
- **Control Logic**: Triggering an action once two values of type `BOOL` satisfy the "equal to" condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `BOOL`-based comparison condition.

Comparison with Similar Function Blocks

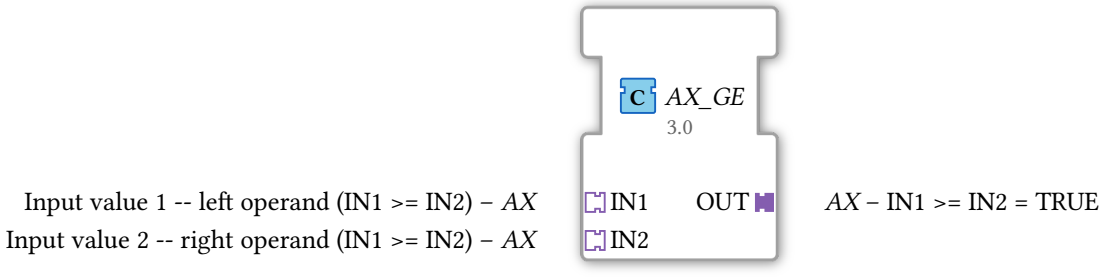
- **AX_NE**: Checks for "not equal to" – the inverse function of AX_EQ.
- **AX_NE**: Checks for inequality of values.

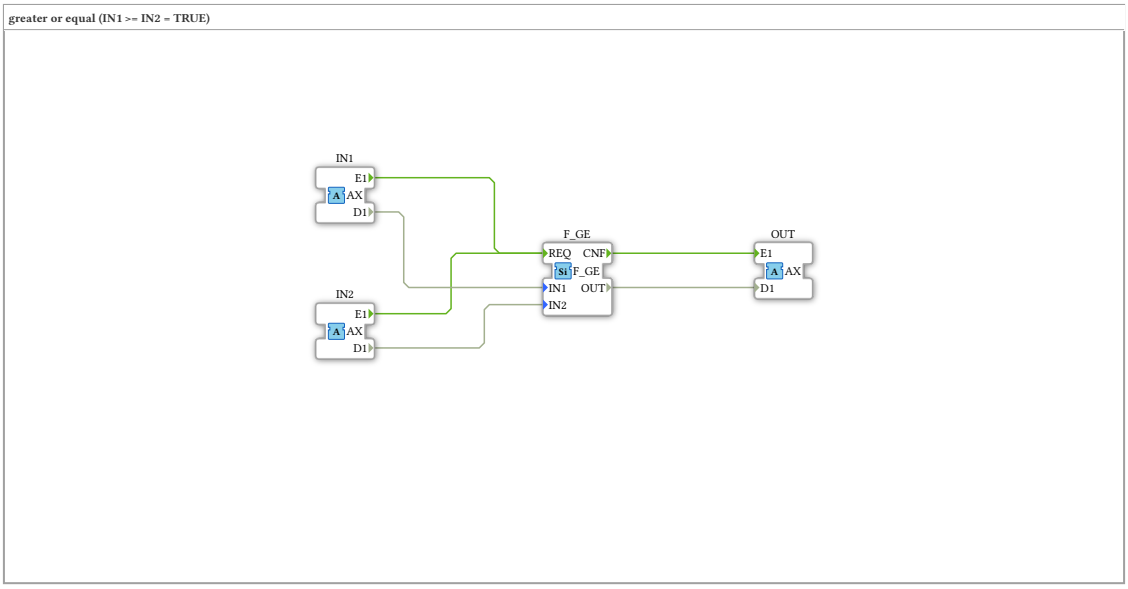
AX_EQ is part of the complete AX comparison family (EQ/NE/GT/GE/LT/LE) for the `BOOL` data type.

Conclusion

AX_EQ is a simple yet important comparison block for IEC 61131-3 control applications using the `BOOL` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AX_GE





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_GE** compares two values of type `BOOL` (Boolean value) and checks whether the first value (IN1) is greater than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. `FALSE` is considered the smaller and `TRUE` the greater value.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BOOL`, Boolean value).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 >= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AX` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AX` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AX` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GE` (type `iec61131::comparison::F_GE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AX`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than or equal to”) for the `BOOL` data type.
- **Comparison logic**: `FALSE` is considered the smaller and `TRUE` the greater value.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking enable and state bits against a reference or threshold value of type `BOOL`.
- **Control Logic**: Triggering an action once two values of type `BOOL` satisfy the “greater than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `BOOL`-based comparison condition.

Comparison with Similar Function Blocks

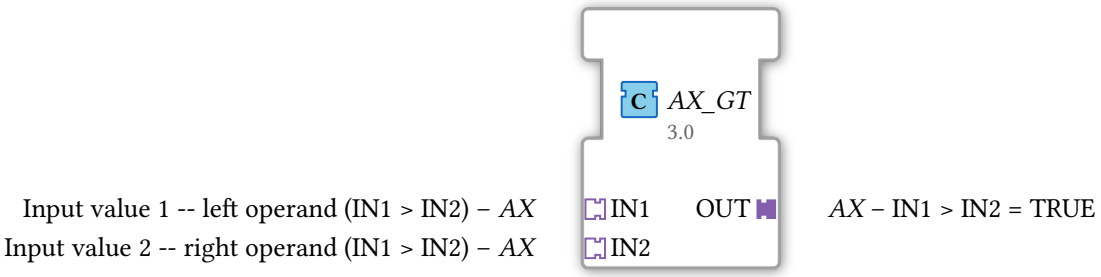
- **AX_LE**: Checks for “less than or equal to” – the inverse function of AX_GE.
- **AX_EQ**: Checks for equality of values.
- **AX_GT**: Performs the “greater than” comparison – the base variant without the equality case, of which AX_GE is the extended version.

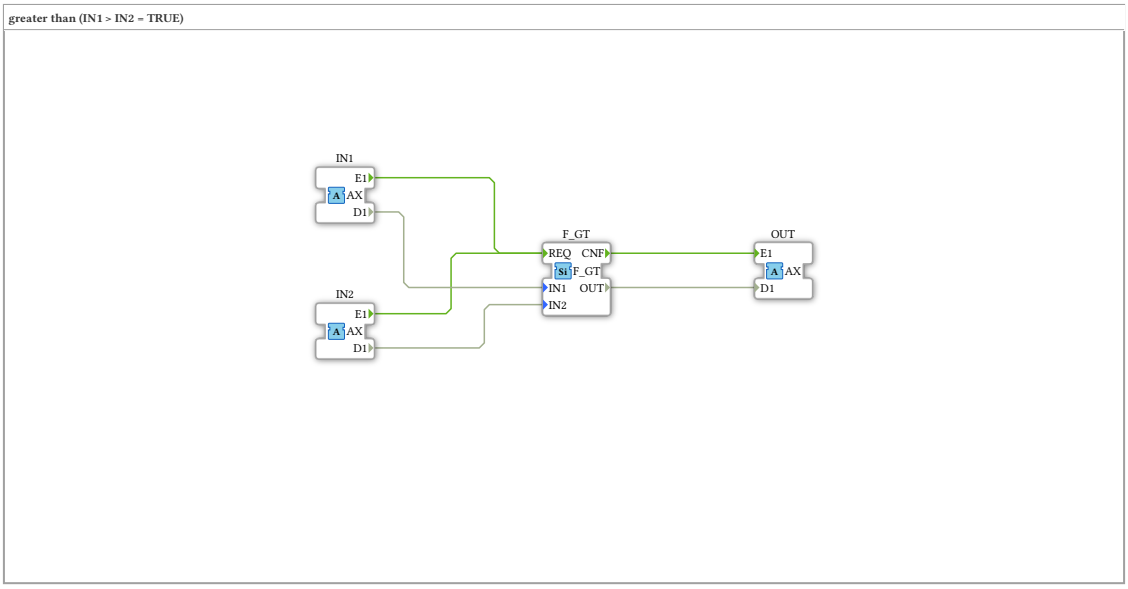
AX_GE completes the AX comparison family with the boundary case of equality.

Conclusion

AX_GE is a simple yet important comparison block for IEC 61131-3 control applications using the `BOOL` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AX_GT





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_GT** compares two values of type `BOOL` (Boolean value) and checks whether the first value (IN1) is greater than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. `FALSE` is considered the smaller and `TRUE` the greater value.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BOOL`, Boolean value).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 > IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AX` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AX` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AX` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_GT` (type `iec61131::comparison::F_GT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AX`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“greater than”) for the `BOOL` data type.
- **Comparison logic**: `FALSE` is considered the smaller and `TRUE` the greater value.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking enable and state bits against a reference or threshold value of type `BOOL`.
- **Control Logic**: Triggering an action once two values of type `BOOL` satisfy the “greater than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `BOOL`-based comparison condition.

Comparison with Similar Function Blocks

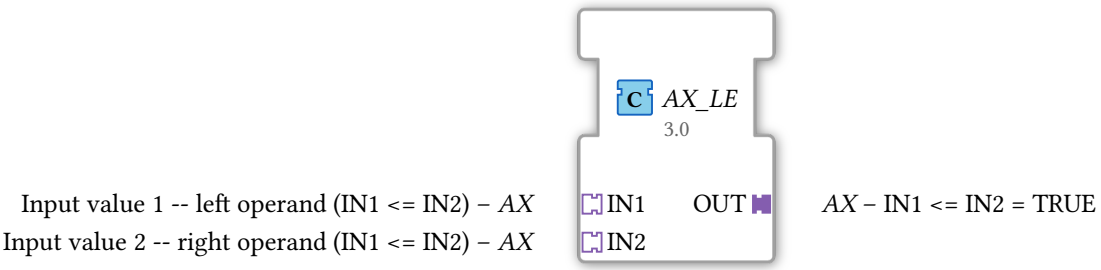
- **AX_LT**: Checks for “less than” – the inverse function of AX_GT.
- **AX_EQ**: Checks for equality of values.
- **AX_GE**: Performs the “greater than or equal to” comparison – the extended version of AX_GT that includes the equality case.

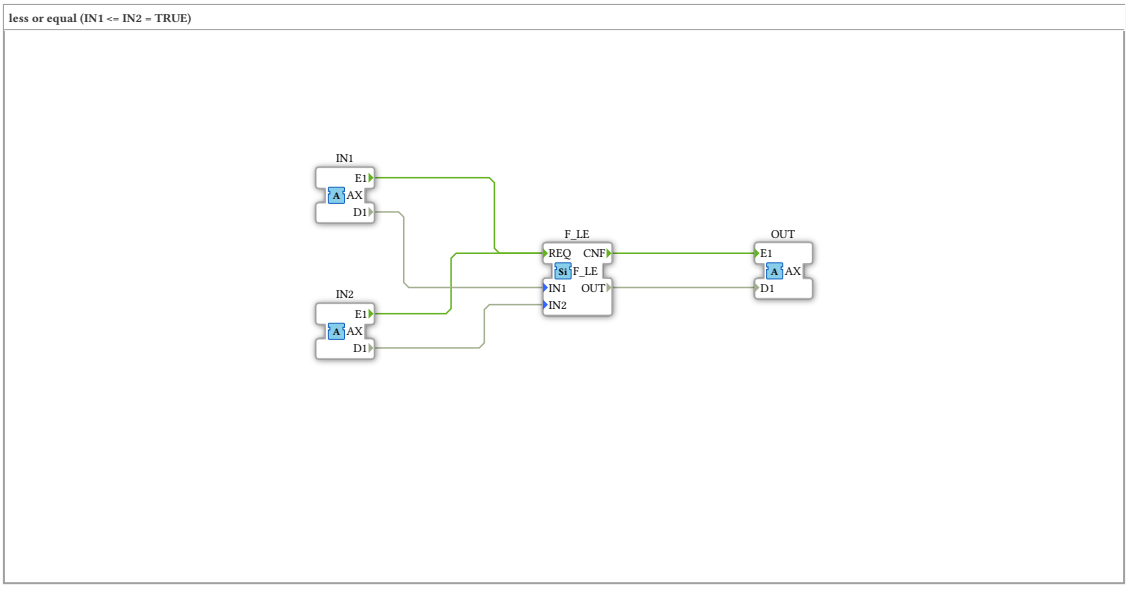
AX_GT is the basic “greater than” component without the equality case, complemented by AX_GE.

Conclusion

AX_GT is a simple yet important comparison block for IEC 61131-3 control applications using the `BOOL` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AX_LE





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_LE** compares two values of type `BOOL` (Boolean value) and checks whether the first value (IN1) is less than or equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. `FALSE` is considered the smaller and `TRUE` the greater value.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BOOL`, Boolean value).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 <= IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AX` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AX` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AX` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LE` (type `iec61131::comparison::F_LE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AX`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than or equal to”) for the `BOOL` data type.
- **Comparison logic**: `FALSE` is considered the smaller and `TRUE` the greater value.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking enable and state bits against a reference or threshold value of type `BOOL`.
- **Control Logic**: Triggering an action once two values of type `BOOL` satisfy the “less than or equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `BOOL`-based comparison condition.

Comparison with Similar Function Blocks

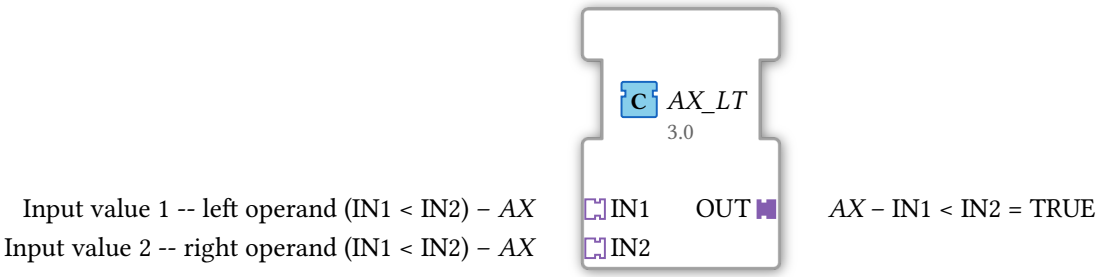
- **AX_GE**: Checks for “greater than or equal to” – the inverse function of AX_LE.
- **AX_EQ**: Checks for equality of values.
- **AX_LT**: Performs the “less than” comparison – the base variant without the equality case, of which AX_LE is the extended version.

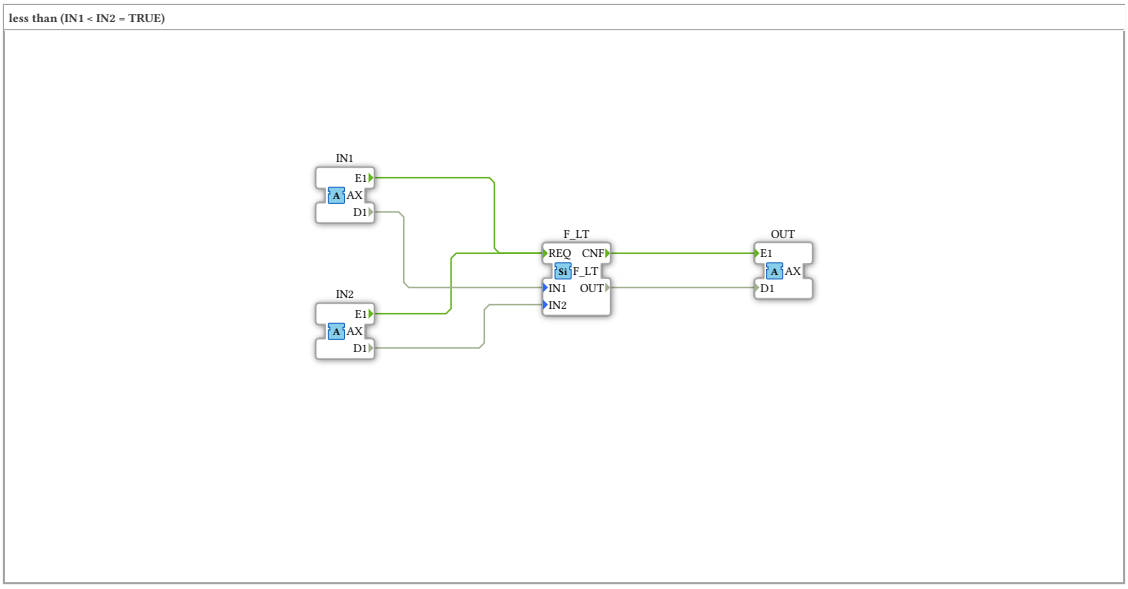
AX_LE completes the AX comparison family with the boundary case of equality.

Conclusion

AX_LE is a simple yet important comparison block for IEC 61131-3 control applications using the `BOOL` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AX_LT





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_LT** compares two values of type `BOOL` (Boolean value) and checks whether the first value (IN1) is less than the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. `FALSE` is considered the smaller and `TRUE` the greater value.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BOOL`, Boolean value).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 < IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AX` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AX` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AX` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_LT` (type `iec61131::comparison::F_LT`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AX`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“less than”) for the `BOOL` data type.
- **Comparison logic**: `FALSE` is considered the smaller and `TRUE` the greater value.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking enable and state bits against a reference or threshold value of type `BOOL`.
- **Control Logic**: Triggering an action once two values of type `BOOL` satisfy the “less than” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `BOOL`-based comparison condition.

Comparison with Similar Function Blocks

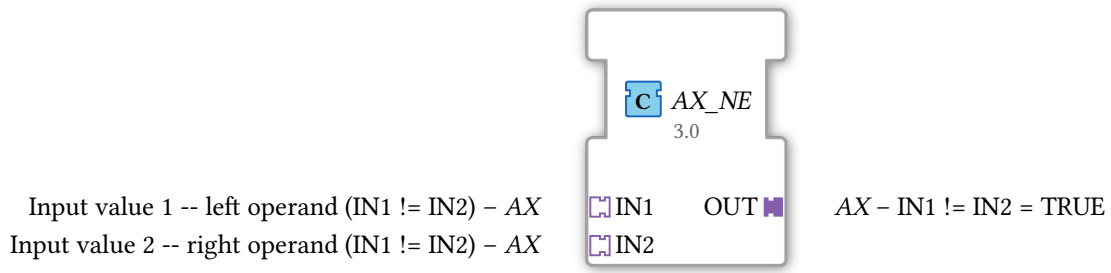
- **AX_GT**: Checks for “greater than” – the inverse function of AX_LT.
- **AX_EQ**: Checks for equality of values.
- **AX_LE**: Performs the “less than or equal to” comparison – the extended version of AX_LT that includes the equality case.

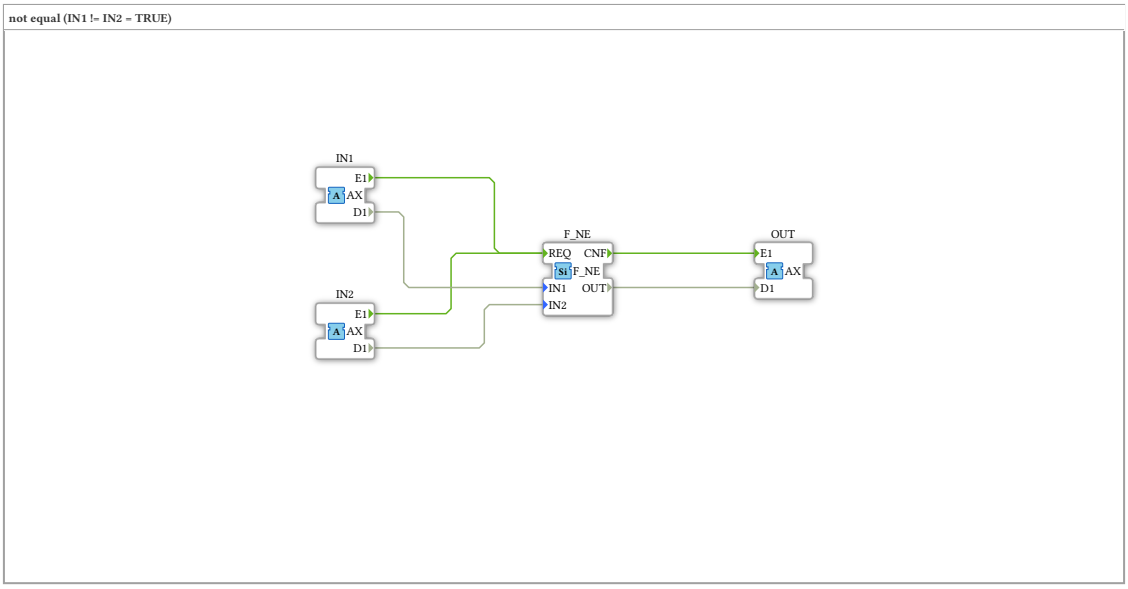
AX_LT is the basic “less than” component without the equality case, complemented by AX_LE.

Conclusion

AX_LT is a simple yet important comparison block for IEC 61131-3 control applications using the `BOOL` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

AX_NE





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_NE** compares two values of type `BOOL` (Boolean value) and checks whether the first value (IN1) is not equal to the second value (IN2). The result is provided as a Boolean value via the output adapter. The function block is event-driven and operates asynchronously. `FALSE` is considered the smaller and `TRUE` the greater value.

Interface Structure

The function block has no direct event or data inputs and outputs, but communicates exclusively via adapters.

Event Inputs

- **IN1.E1**: Event input of socket adapter IN1 – triggers the comparison.
- **IN2.E1**: Event input of socket adapter IN2 – also triggers the comparison.

Event Outputs

- **OUT.E1**: Event output of the plug adapter OUT – indicates that the comparison result is available at the data output.

Data Inputs

- **IN1.D1**: First value to be compared (data type `BOOL`, Boolean value).
- **IN2.D1**: Second value to be compared (data type same as IN1).

Data Outputs

- **OUT.D1**: Boolean result of the comparison – `TRUE` if `IN1 != IN2`, otherwise `FALSE`.

Adapters

Adapter	Type	Direction	Description
IN1	`AX` (Socket)	Input	Adapter for the first comparison value and its associated event
IN2	`AX` (Socket)	Input	Adapter for the second comparison value and its associated event
OUT	`AX` (Plug)	Output	Adapter for the comparison result and the output event

The adapters `AX` and `AX` are unidirectional interfaces, each providing one event channel and one data channel.

Functionality

1. The function block waits for an event at one of the socket adapters **IN1.E1** or **IN2.E1**.
2. As soon as such an event arrives, it is forwarded to the internal function block `F_NE` (type `iec61131::comparison::F_NE`). There, the current values of `IN1.D1` and `IN2.D1` are compared.
3. The internal function block generates an output event (`CNF`) and outputs the Boolean result (`OUT`).
4. This event is propagated to the plug adapter **OUT.E1**, and the data value `OUT.D1` is set to `TRUE` or `FALSE`.
5. Output occurs only when the input values change or when another event occurs.

Technical Features

- **Adapter-based interface**: The function block uses only adapters (`AX`/`AX`), allowing for flexible integration into various communication contexts.
- **Asynchronous processing**: The comparison operation is triggered by an event at one of the input adapters; the output signal is delivered in response to the event.
- **Standard compliance**: The function block implements the comparison function from IEC 61131-3 (“not equal to”) for the `BOOL` data type.
- **Comparison logic**: `FALSE` is considered the smaller and `TRUE` the greater value.
- **No state machines**: The function block is purely combinational – the result is generated immediately after the event; there is no internal state.

State Overview

This function block does not have an explicit state machine (ECC). Its internal logic consists of a direct connection between the adapters and a standardized comparison function block. Output occurs immediately upon event input.

Application Scenarios

- **Limit Monitoring**: Checking enable and state bits against a reference or threshold value of type `BOOL`.
- **Control Logic**: Triggering an action once two values of type `BOOL` satisfy the “not equal to” condition.
- **Data Filtering**: Selecting data records or signals that satisfy a `BOOL`-based comparison condition.

Comparison with Similar Function Blocks

- **AX_EQ**: Checks for “equal to” – the inverse function of AX_NE.
- **AX_EQ**: Checks for equality of values.

AX_NE is part of the complete AX comparison family (EQ/NE/GT/GE/LT/LE) for the `BOOL` data type.

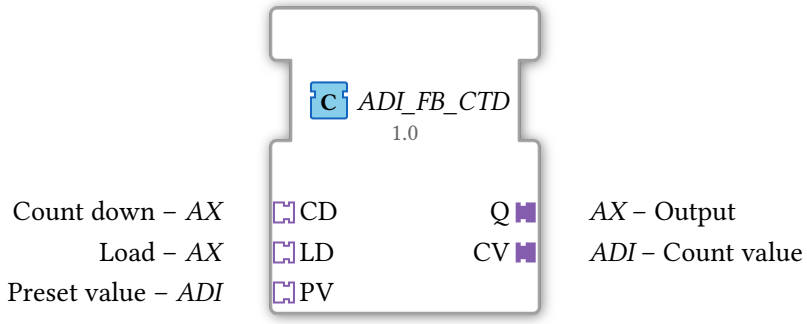
Conclusion

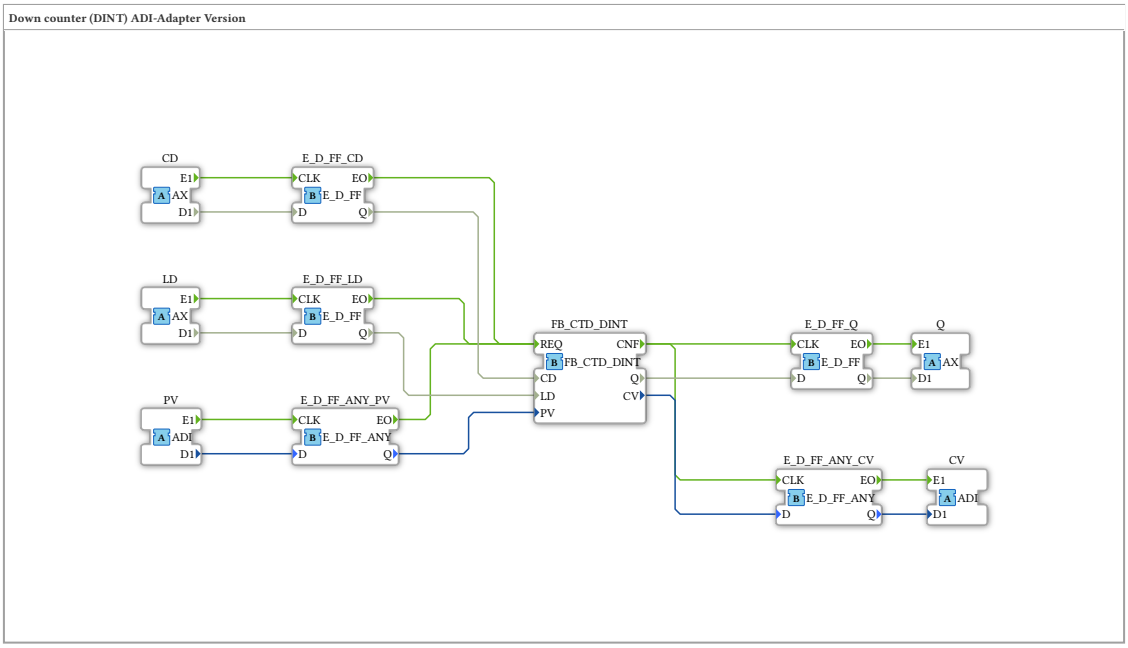
AX_NE is a simple yet important comparison block for IEC 61131-3 control applications using the `BOOL` data type. Its adapter-oriented interface makes it versatile and allows for a clear separation of event and data paths. Due to its event-driven operation, it is particularly suitable for real-time systems where conditions must react immediately to changes in values.

Numeric Index (iec61131-3)

Here you will find all exercises from the `iec61131-3` workspace, listed in ascending order. …

ADI_FB_CTD





• – ○ • – ○ • – ○ •

Introduction

The **ADI_FB_CTD** is a down counter for values of type `DINT` that communicates exclusively via ADI adapters. It encapsulates an internal standard counter and enables counting down, loading a default value, and outputting the current counter value – all via standardized adapter interfaces.

Interface Structure

Event Inputs

The FB does not have direct event inputs. Events are provided via the adapter sockets **CD**, **LD**, and **PV**.

Event Outputs

Name	Type	Description
CNF	Event	Acknowledgement signal output after each processing of an incoming event.

Data Inputs

No direct data inputs; data is provided via the adapter sockets.

Data Outputs

No direct data outputs; data is provided via the adapter plugs.

Adapter

Name	Type	Direction	Description
CD	`adapter::types::unidirectional::AX`	Socket (Input)	Countdown event; the data value (D1) is passed to the internal counter as a CD pulse.
LD	`adapter::types::unidirectional::AX`	Socket (Input)	Load the counter with the current value of **PV** The data value (D1) signals the charging process.
PV	`adapter::types::unidirectional::ADI`	Socket (Input)	Preset value as `DINT`; the event (E1) triggers an update of the internal PV register.

Q	adapter::types::unidirectional::AX	Plug (Output)	Output signal indicating whether the meter reading has reached zero (e.g., TRUE for zero).
CV	adapter::types::unidirectional::ADI	Plug (Output)	Current meter reading as DINT; output with each update.

Functionality

The ADI_FB_CTD uses an internal IEC 61131 counter (FB_CTD_DINT). All communication takes place via ADI adapters that combine an event (E1) and a data value (D1):

- An event at the **CD** socket causes the internal counter to decrement its current value by **1**.
- An event at the **LD** socket loads the counter to the value last provided via **PV**.
- An event at the **PV** socket updates the default value in the internal register; the counter itself is not changed. Only a subsequent LD event takes over the new value.

All incoming events (CD, LD, and PV) are routed to the common REQ input of the internal counter. The internal function block distinguishes between counting and loading based on the data values. Upon completion of the action, the event CNF is triggered, simultaneously operating the output adapters **Q** and **CV** and setting the event output CNF.

Technical Features

- **Always Active Output:** The function block outputs an event on the **Q** adapter with **every** update (via CD, LD, or PV) – even if the corresponding data value does not change. If a triggering output is only desired upon an actual state change, a filter block such as AX_D_FF must be placed upstream.
- **Adapter-Based Interface:** All inputs and outputs are implemented as ADI adapters. This enables loose coupling and easy reusability in modular control architectures.
- **Internal Standard Counter:** The counting logic is based on an established IEC 61131 component and is therefore robust and reliable.

State Overview

The FB does not have its own state machine. Its behavior is determined by the internal FB_CTD_DINT, which can operate in the following modes:

- **Loading:** When an event occurs at the **LD** adapter, the counter is set to the current PV value.
- **Counting:** When an event occurs at the **CD** adapter, the counter is decremented.
- **Waiting:** No event is present; the counter retains its current value.

After each processing step, the FB is immediately ready for the next event (no blocking).

Application Scenarios

- **Workpiece Counting:** A sensor reports each passing workpiece as an event on **CD**. The counter decrements from a predefined starting value (**PV**) to zero. When the counter reaches zero, a signal (e.g., "Container full") can be output via **Q**.
- **Time Delay:** In combination with a clock, the counter functions as a down timer, the remaining time of which is read via **CV**.
- **Control of Processes:** The current counter value (**CV**) serves as a parameter for subsequent steps (e.g., positioning, quantity control).

Comparison with Similar Function Blocks

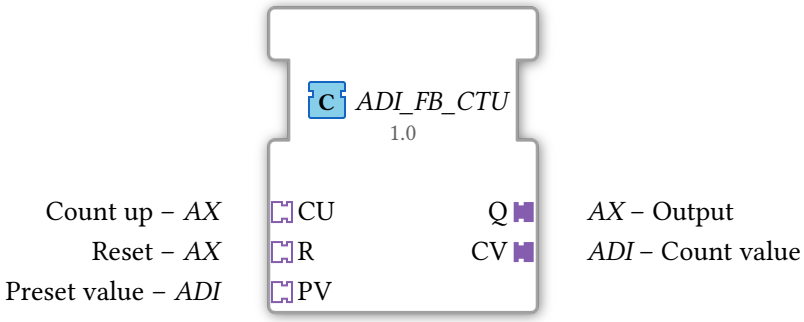
- **CTU (Up Counter)** – counts only upwards; ADI_FB_CTD only downwards.
- **CTUD (Up/Down Counter)** – can count in both directions; ADI_FB_CTD is specialized for one direction.
- **Standard IEC 61499 Counters** – use separate event and data ports, while ADI_FB_CTD unifies the interfaces via adapters.
- **ADI_FB_CTU** – structurally identical, but with upcount logic (also available as an adapter version).

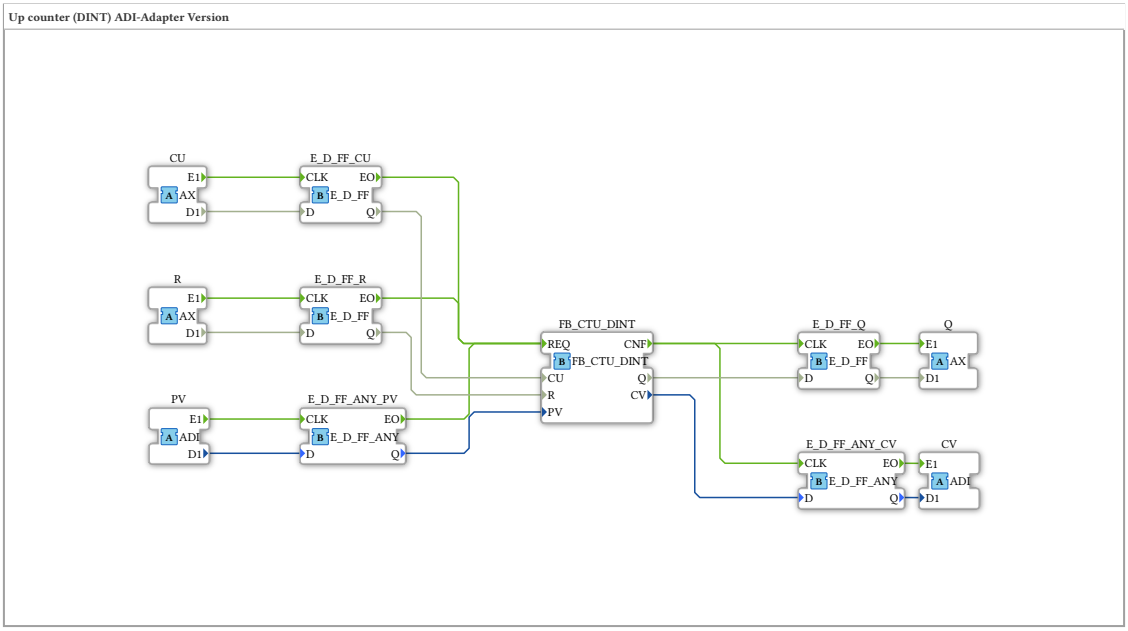
The advantage of the adapter version lies in its more modular, reusable interface; the disadvantage is the higher level of abstraction and the need for filtering when required.

Conclusion

The ADI_FB_CTD is a reliable downcounter for `DINT` values, which, thanks to its consistently adapter-based interface, integrates ideally into loosely coupled control architectures. The fact that the **Q** output triggers with every update must be considered during system design. Thanks to the internal standard counter, the counting logic is robust and proven – a reliable component for downcounting scenarios.

ADI_FB_CTU





• – ◦ • – ◦ • – ◦ •

Introduction

The ADI_FB_CTU is an up-counter for DINT integers, whose inputs and outputs are provided via standardized adapters (AX and ADI). It encapsulates the standard function block `FB_CTU_DINT` and enables its integration into modular, adapter-based systems. This function block is suitable for general counting tasks in automation technology.

Interface Structure

The function block does not have direct event or data interfaces, but only adapters for connection. The following table explains the available adapters, their type, and their function.

Direction	Name	Adapter Type	Description
Input (Socket)	`CU`	`AX`	Count pulse input (event + data)
Input (Socket)	`R`	`AX`	Reset input (event + data)
Input (Socket)	`PV`	`ADI`	Preset value for comparison

Output (Plug)	Q	AX	Output signal – active when meter reading ≥ PV
Output (Plug)	CV	ADI	Current meter reading

The adapters AX (event adapter) and ADI (data adapter) are unidirectional. Both events and their associated data values are transmitted via these adapters.

Event Inputs

No direct event inputs. Events are supplied via the adapters CU and R (of type AX).

Event Outputs

A direct event output, CNF, signals confirmation of processing. Additionally, an output event is sent via the adapter Q (type AX) with each update.

Data Inputs

No direct data inputs. The default value is provided via the adapter PV (type ADI).

Data Outputs

No direct data outputs. The current counter reading is output via the adapter CV (type ADI).

Adapters

The function block uses three sockets (input adapters) and two plugs (output adapters):

- **CU (socket, AX)**: Count pulse – the internal counter is incremented with each event.
- **R (socket, AX)**: Reset – resets the counter to zero.
- **PV (Socket, ADI)**: Preset Value – sets the threshold at which output Q becomes active.
- **Q (Plug, AX)**: Output – becomes active as soon as the counter value reaches or exceeds the value of PV.
- **CV (Plug, ADI)**: Current counter value – can be read by downstream function blocks.

Functionality

The ADI_FB_CTU implements a simple increment counter with preset value comparison.

- Each event at input CU increments the internal counter by 1.

An event at input R resets the counter to 0.

The default value PV is updated when an event arrives at input PV.

After each processing operation (regardless of whether it was triggered by CU, R, or PV), the acknowledgment event CNF is output. Simultaneously, output adapter Q is also updated with an event, and the current counter value is provided via adapter CV.

Internally, a standard function block FB_CTU_DINT is used, with its inputs and outputs wired via the adapters. The counter value is of type DINT (32-bit integer).

Technical Features

- **Adapter-based interface** – enables loose coupling and easy integration into adapter-based architectures (e.g., according to IEC 61499).
- **Unidirectional Adapters** – the adapters AX and ADI each transmit in only one direction.
- **Acknowledgement Event CNF** – any event at an input triggers an immediate acknowledgement.
- **No Edge Detection** – the function block reacts to any event, not to the rising or falling edges of a digital signal.
- **Note in Source Code** – the frequent output of the Q.E1 event can lead to unnecessary load in time-critical applications; filtering may be necessary.

State Overview

The function block has only one internal state: the current counter value (initial value = 0). Depending on the incoming events, the following state transitions occur:

| Event | Condition | New State (Counter) | Output |
** ... |-----|----|---------|---|
CU	–	Counter + 1	CNF, Q.E1, CV
R	–	0	CNF, Q.E1, CV
PV	–	unchanged	CNF, Q.E1, CV (PV value is stored internally)

Output `Q` (via the adapter) is set as soon as `Zähler ≥ PV` is reached. The current value of `Q` is included with every output.

Application Scenarios

- **Event Counting** – Counting pulses, e.g., part passage, machine cycles.
- **Fill Level Monitoring** – Recording the number of containers or batches.
- **Production Control** – Triggering an action when a specific quantity is reached.
- **Time Measurement** – In combination with a pulse generator, the number of pulses can be used as a measure of time.
- **Adapter-based automation systems** – wherever a standardized adapter interface is required.

Comparison with similar components

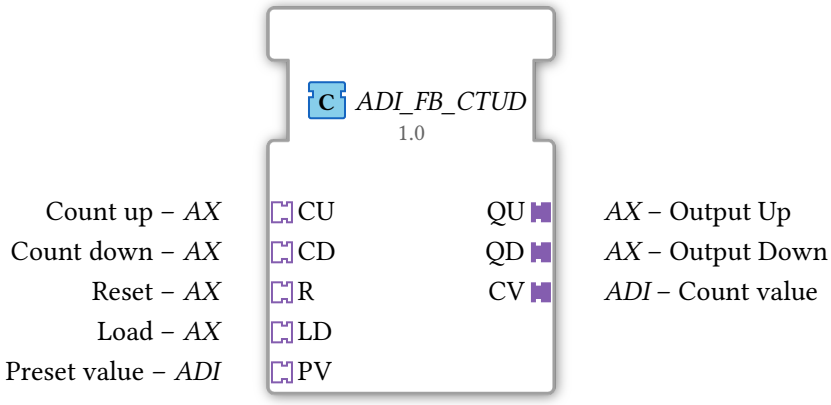
Component	Properties
`CTU` (Standard, without adapter)	Same counting function, but with direct event and data inputs/outputs. Easier to use in classic IEC 61499 networks.
`ADI_FB_CTUD`	Up/down counter, also adapter-based. Additionally offers a down count input.
`FB_CTU_DINT`(Internal)	Same counting logic, but without adapter encapsulation. The adapter version offers a uniform, modular interface.
`CTU` with AX filter	If edge detection is required, pure change detection can be achieved by preceding it with an AX_D_FF.

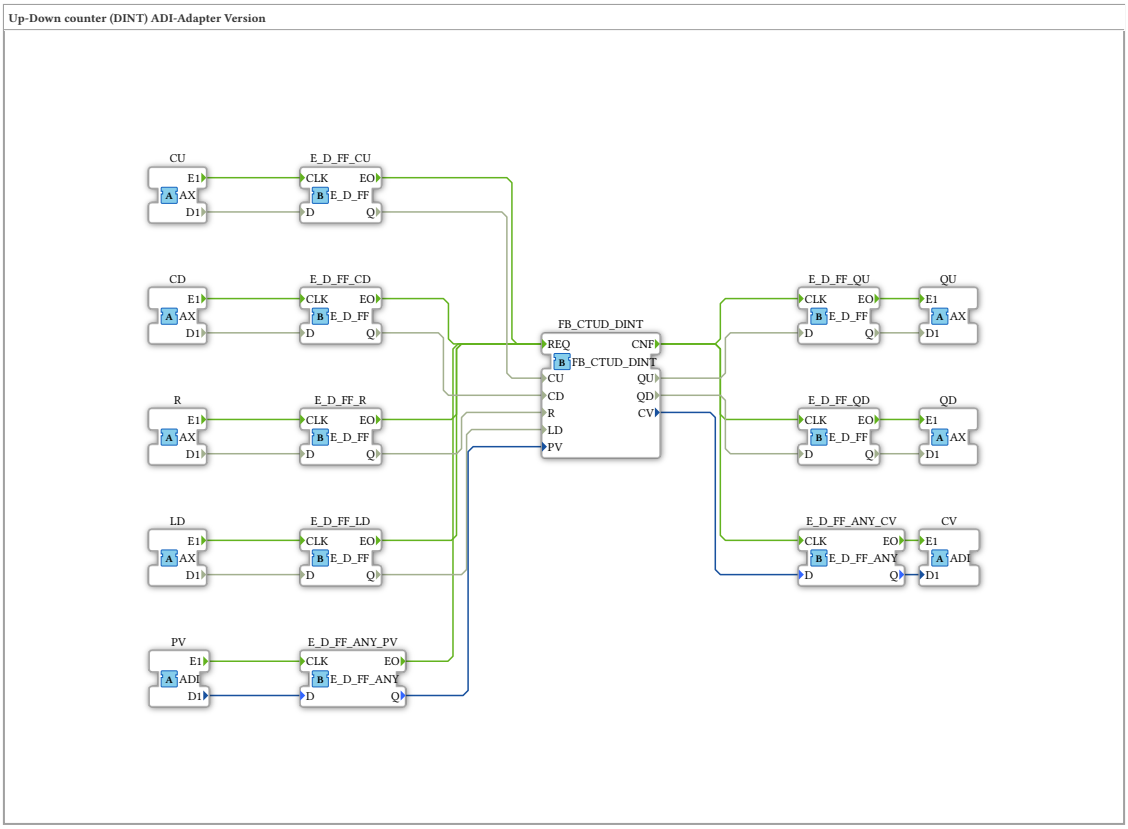
The ADI_FB_CTU is designed as a wrapper for the standard counter and facilitates reuse in adapter-based frameworks.

Conclusion

The ADI_FB_CTU is a flexible, adapter-based increment counter for DINT values. It encapsulates the proven counting logic of `FB_CTU_DINT` and makes it available via standardized adapters (AX and ADI). The fact that output events are generated with each update should be considered during system design. The function block is ideally suited for modular, expandable automation solutions that require a uniform adapter interface.

ADI_FB_CTUD





• – ○ • – ○ • – ○ •

Introduction

The **ADI_FB_CTUD** is an up/down counting function block for integer values (DINT). It implements a forward/backward counter that is controlled via standardized **adapter interfaces**. The block encapsulates the IEC 61131-3 component `FB_CTUD_DINT` and provides its functionality via event-driven adapters.

Interface Structure

Event Inputs

The block does not have direct event inputs. Control is exclusively via the **sockets**:

- **CU (Count Up)** – Event to increment the counter value
- **CD (Count Down)** – Event to decrement the counter value
- **R (Reset)** – Event to reset the counter value to zero
- **LD (Load)** – Event to load the preset value
- **PV (Preset Value)** – Event to set the preset value (data value)

These adapters provide both an event and a data value (for AX adapters, via the data output `D1`).

Event Outputs

- **CNF** – Acknowledge event, which is output after each processing of one of the input events.

Data Inputs

The data values of the input adapters are processed via the connections of the internal function block `FB_CTUD_DINT`. The adapters provide the following data:

- **CU.D1** – Counting direction (Bool, 1 = Count up)
- **CD.D1** – Counting direction (Bool, 1 = Count down)

(Note: In standard CTUD-DINT, only one counting event with direction selection is typically used; here, CU/CD are treated as separate events.)

- **R.D1** – Reset signal (Bool)

- **LD.D1** – Load signal (Bool)
- **PV.D1** – Preset value (DINT)

Data Outputs

The output data is provided via the plug adapters:

- **QU.D1** – Counter has incremented (Bool, after CU event)
- **QD.D1** – Counter has decremented (Bool, after CD event)
- **CV.D1** – Current count value (DINT)

Adapters

The function block uses three different adapter types:

- **unidirectional::AX** – for event and Boolean data (CU, CD, R, LD as sockets; QU, QD as plugs)
- **unidirectional::ADI** – for value transfer (PV as a socket, CV as a plug)

The adapters are designed to transfer both the event and the associated data value in one operation.

Functionality

The internal function block `FB_CTUD_DINT` implements the classic up/down counter logic:

- When an event occurs on **CU**, the counter value is incremented by 1, provided `CU.D1` = TRUE (or the event alone is considered an up pulse).
- When an event occurs on **CD**, the counter value is decremented by 1, provided `CD.D1` = TRUE.
- An event on **R** resets the counter value to 0.
- An event on **LD** loads the current counter value with the value passed via **PV**.
- After each processing step, **CNF** is output, as well as the events **QU.E1** and **QD.E1** via the corresponding adapters if the counter value has changed.

Important: The function block fires the output events (QU.E1, QD.E1) on *every* update (CU, CD, R, LD, PV) – not just on an actual value change. If only edge detection (on-change) is required, the use of an **AX_D_FF** as a filter is recommended.

Technical Features

- **Adapter-based interface:** All inputs and outputs are via standardized unidirectional adapters, which increases reusability and encapsulation.
- **IEC 61131-3 Compatibility:** The internal counter complies with the standard and allows for easy migration between different control systems.
- **Always Active Events:** As described above, output events are generated with every input event – this can lead to high bus load in time-critical applications.
- **No Top-Level State Machine:** The function block (FB) does not have its own ECC; all logic is executed by the internal FB.

State Overview

The function block does not have an explicit state machine. Processing is strictly event-driven:

1. Wait for an event at one of the sockets.
2. Process the associated data value (if available) and the internal counter state.
3. Output **CNF** and, if applicable, **QU.E1** and **QD.E1**.
4. Return to step 1.

Application Scenarios

- **Counting pulses** in manufacturing systems (e.g., workpiece counters).
- **Position detection** with incremental encoders (counting up/down).
- **Bill of materials and inventory counting** with reset capability.
- **Event-driven controllers** in automation technology, where counter readings are exchanged via adapters.

Comparison with similar function blocks

Property	ADI_FB_CTUD	Standard CTUD (IEC 61131-3)
Interface	Adapter-based (AX/ADI)	Direct inputs/outputs (events, data)

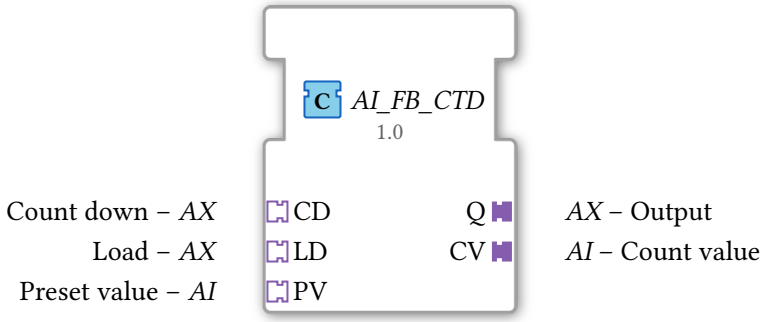
Event behavior	Output on every update	Mostly edge- or value-dependent
Reusability	Easy due to adapter encapsulation	Dependent on the specific controller
Compatibility	IEC 61131-3 compliant (internal)	Direct standard implementation
Filtering	Recommendation: AX_D_FF	Partially integrated

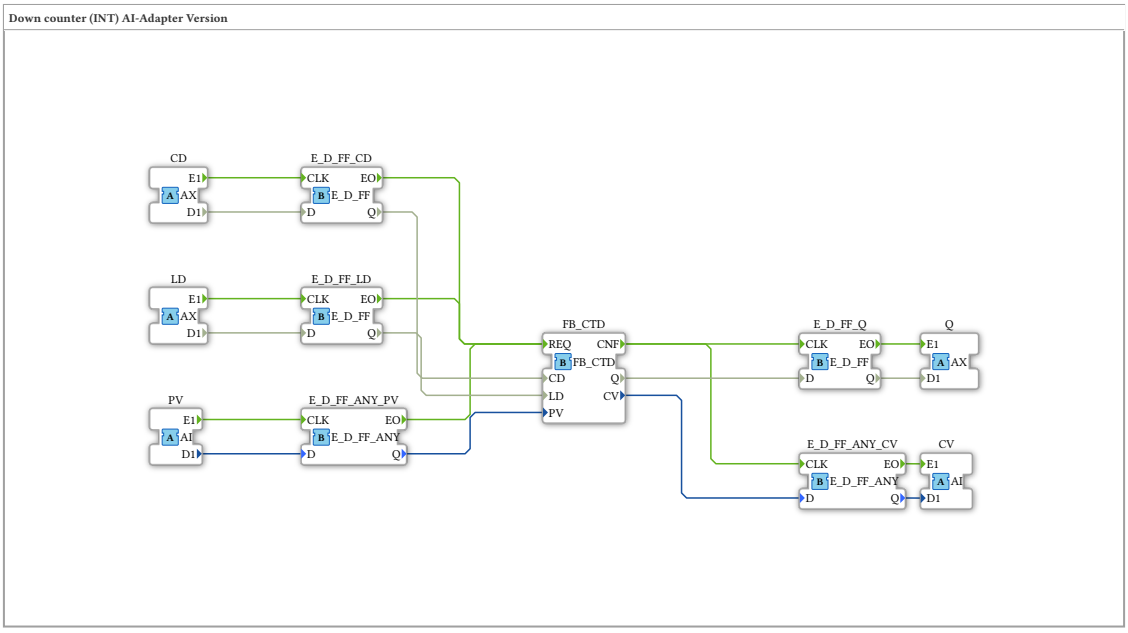
The ADI_FB_CTUD offers clean, adapter-based encapsulation, but requires additional measures for on-change triggering if needed.

Conclusion

The **ADI_FB_CTUD** is a flexible and standards-compliant up/down counter with a DINT value range, distinguished by its modern adapter interface. It is particularly suitable for modular control architectures where components communicate via standardized adapters. The always-active event output should be considered when integrating it into real-time systems.

AI_FB_CTD





• – ○ • – ○ • – ○ •

Introduction

The function block **AI_FB_CTD** is a down counter for integer values, standardized according to IEC 61499-2. It consistently uses adapter interfaces (AX, AI) to transport events and data. The block is implemented as a wrapper around the classic IEC 61131 function block `FB_CTD` and enables a modular, adapter-based connection in 4diac IDE networks. Every incoming event via the counter controllers (CD, LD, PV) updates the internal counter and triggers an output event.

Interface Structure

Event Inputs

The function block does not have direct, separate event inputs. The necessary events are provided via the **Socket Adapters** (CD, LD, PV) – each of these modules carries an event (E1) that triggers the counter logic.

Socket	Event (in the adapter)	Description
CD	E1	Counting event (down counter)
LD	E1	Load event (set to PV)
PV	E1	Default update

Event Outputs

Name	Type	Description
CNF	Event	Acknowledge event after each counter update (CD, LD, PV)

Additionally, the output events are also transmitted via the **Plug Adapters** Q and CV (E1 each).

Data Inputs

There are no direct data inputs. All data values are transmitted via the socket adapters:

- **CD** provides the down count pulse (D1) – a logical value that decrements the counter.
- **LD** provides the charge signal (D1) – a logical value that sets the counter to the current PV value.
- **PV** provides the setpoint value (D1) – an integer that is loaded as the new meter reading.

Data Outputs

There are no direct data outputs. The output data is provided via the plug adapters:

- **Q** provides the meter status (D1) – BOOL, TRUE if the meter reading is ≤ 0.
- **CV** provides the current meter reading (D1) – INT.

Adapters

Label	Direction	Type	Description
CD	Socket	`adapter::types::unidirectional::AX`	Counter input (event + data) for counting down
LD	Socket	`adapter::types::unidirectional::AX`	Load input (event + data) for setting to PV
PV	Socket	`adapter::types::unidirectional::AI`	Default input (event + data) for the preset value
Q	Plug	`adapter::types::unidirectional::AX`	Output signal (event + data) – counter ≤ 0
CV	Plug	`adapter::types::unidirectional::AI`	Output counter value (event + current counter reading)

Functionality

The function block internally contains a function block `iec61131::counters::FB_CTD`, which implements the actual counting logic. The function is as follows:

1. An event at **CD** (via `CD.E1`), **LD** (`LD.E1`), or **PV** (`PV.E1`) triggers the internal function block via its event input `REQ`.
2. The internal function block evaluates the associated data:

- On a **CD** event, the counter is decremented by 1 (if the associated date is TRUE).
- On a **LD** event, the counter is set to the current value of **PV** (if the load date is TRUE).
- In the event of a **PV** event, the default value is updated internally (the counter remains unchanged).

1. After processing, the internal FB generates its `CNF` event. This is distributed to the external outputs:

- `CNF` (direct event output)
- `Q.E1` (event of plug adapter Q)
- `CV.E1` (event of plug adapter CV)

1. Simultaneously, the following data is transferred:

- `FB_CTD.Q` (BOOL) ⟶ `Q.D1`
- `FB_CTD.CV` (INT) ⟶ `CV.D1`

` Note: The output event is triggered **with every update** (i.e., on CD, LD, or PV). A simple value change without an event (On-Change) is not implemented.

Technical Features

- **License and Origin**: This function block is licensed under the **Eclipse Public License 2.0** (EPL 2.0). It was developed as teaching material for the master craftsman schools at Ostbahnhof.
- **Adapter-Based Communication**: Unlike classic FB interfaces, all inputs and outputs are implemented via adapters (AX, AI). This enables flexible, hierarchical wiring and reuse in complex control architectures.
- **On-Change Filter**: The function block fires the output event on **every** input event, regardless of an actual value change. If change detection is required, the use of an **AX_D_FF** (Differentiator Flip-Flop) as a pre-filter is recommended (see also the copyright notice in the XML).
- **Internal FB**: The counting logic is based on the standard module `FB_CTD`From IEC 61131 (adapted for 4diac). This ensures deterministic behavior and compatibility with PLC programs.

State Overview

The function block does not have an explicit state machine. The only internal variable is the **current counter value** (integer). This value is modified by the following events:

- Initial value: 0 (or set to PV via LD)
- CD event: Counter = Counter - 1 (if the associated date is TRUE, otherwise no change)
- LD event: Counter = PV (if the loading date is TRUE)
- PV event: PV value is stored internally; the counter remains unchanged.

Therefore, a graphical state machine is not required.

Application Scenarios

- **Inventory Counter**: Decrementing an inventory level upon removal (CD), resetting via LD after replenishment.
- **Event Counter with Threshold**: Counts events until 0 is reached, then triggers a signal (Q).
- **Time/Pulse Control**: Used as a pulse counter in conjunction with a clock (CD) and specifying the desired number of events (PV).
- **Adapter Integration**: In modular control systems where events and data are to be encapsulated via standardized adapter interfaces.

Comparison with Similar Components

Component	Counting Direction	Interfaces	Special Feature
AI_FB_CTD	Downward	Adapters only (AX, AI)	Output on every update
AI_FB_CTU	Upward	Adapters only (AX, AI)	Upward counter, analog structure
FB_CTD (Standard)	Downward	Direct Events/Data	Classic Variable Connection, without Adapter
FB_CTUD	Both	Direct	Combined Increment/Decrement Counter, Extended Logic

The **AI_FB_CTD** stands out due to its consistent use of adapters, making it ideal for hierarchical and reusable control models. Unlike the standard FB_CTD, separate data and event inputs are eliminated – everything is handled via the adapters.

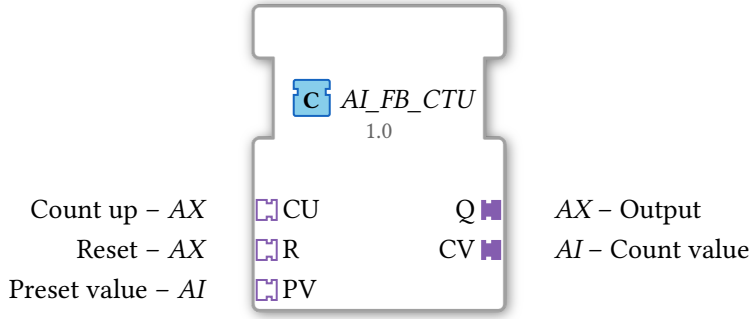
Conclusion

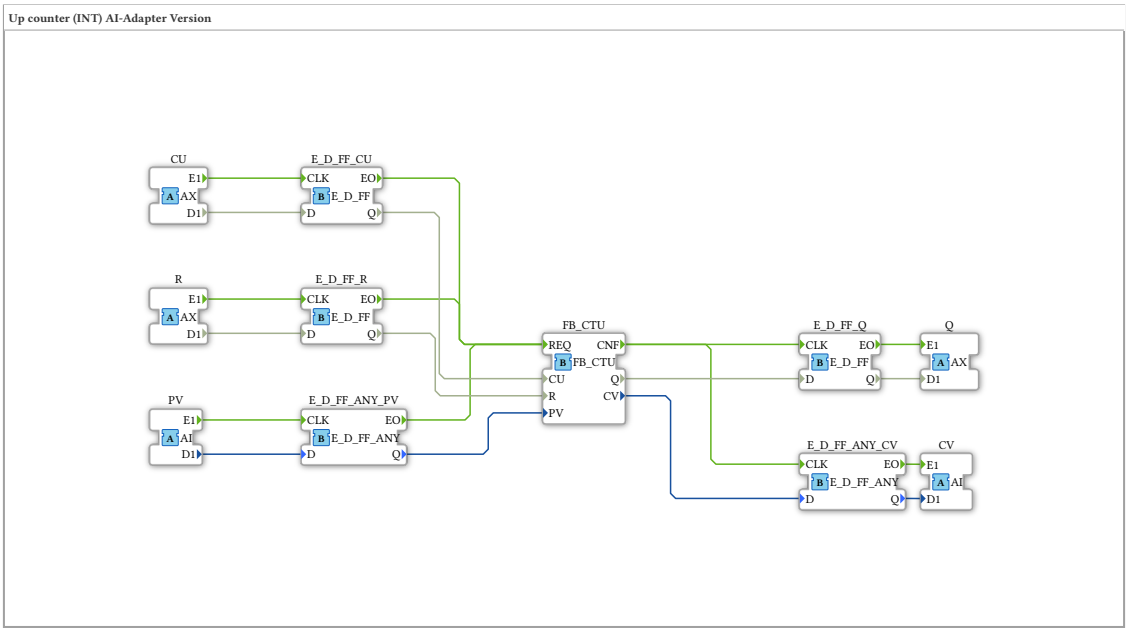
The `AI_FB_CTD` is a flexible, adapter-based down counter for integer values that translates the proven counting logic from IEC 61131 into a modern, modular interface landscape. It is ideally suited for use in distributed control systems according to IEC 61499, where encapsulation and reusability are paramount. Its ease of use (three input events, two output adapters) and clear documentation make it a solid foundation for counter applications. For change detection, it is recommended to use a `AX_D_FF` upstream.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AI_FB_CTU





• – ○ • – ○ • – ○ •

Introduction

The **AI_FB_CTU** is an up counter for integers (INT) that encapsulates the IEC 61131-3 functionality of a CTU (Counter Up) in an adapter-based form factor. It is specifically designed for use in the 4diac IDE and allows for modular connection via unidirectional adapter interfaces. The block fires an acknowledgment event with each update of its inputs (CU, R, PV), making it suitable for time-controlled or event-driven counting tasks.

Interface Structure

Event Inputs

The block does not have direct event inputs. Event control is handled exclusively via the adapter sockets **CU**, **R**, and **PV**. Each of these sockets provides an event (E1) that triggers the internal process.

- **CU.E1** – Counting pulse (event from the counting-up adapter)
- **R.E1** – Reset (event from the reset adapter)
- **PV.E1** – Set preset value (event from the preset-value adapter)

Event Outputs

- **CNF** (Type: Event) – Confirmation event triggered after each successful processing of all three possible events.

The output adapters **Q** and **CV** are also served with the same event:

- **Q.E1** – Event for the output adapter (counter reading reaches or exceeds the preset value)
- **CV.E1** – Event for the current count value

Data Inputs

All data inputs are provided via the adapter sockets:

Adapter	Data Input	Type	Description
CU.D1	CU	INT	Count Up – Count pulse (increments with each event)
R.D1	R	INT	Reset – Value to which the counter is reset (typically 0)
PV.D1	PV	INT	Preset Value – Threshold at which output Q becomes active

Data Outputs

- **Q.D1** (via adapter Q, type AX) – Output signal (BOOL), becomes TRUE when the counter value is ≥ PV.
- **CV.D1** (via adapter CV, type AI) – Current counter value (INT).

Adapters

Name	Type	Direction	Description
CU	adapter::types::unidirectional::AX	Socket (Input)	Event and data adapter for the counter pulse
R	adapter::types::unidirectional::AX	Socket (Input)	Event and data adapter for the reset
PV	adapter::types::unidirectional::AI	Socket (Input)	Data adapter (value only, no event) for the preset value
Q	adapter::types::unidirectional::AX	Plug (Output)	Event and data adapter for the counter output
CV	adapter::types::unidirectional::AI	Plug (Output)	Data adapter (value only) for the current counter reading

Functionality

The **AI_FB_CTU** internally uses a standardized IEC 61131-3 CTU block (`iec61131::counters::FB_CTU`). Upon each incoming event (CU.E1, R.E1, or PV.E1), the internal CTU is processed by calling its REQ input. The data from the adapters (CU.D1, R.D1, PV.D1) is forwarded directly to the corresponding inputs of the internal CTU.

After processing, the result (current counter reading CV and output Q) is sent to the output adapters, and the acknowledgment event CNF is triggered simultaneously. Important: **The block performs a complete pass for each of the three events**, meaning that CU, R, and PV are always evaluated together. This behavior can lead to unexpected counting pulses if not all inputs are relevant at the same time. For change-only triggering, the use of an AX_D_FF (D flip-flop) as a filter is recommended.

Technical Features

- **Adapter-Based Interface**: All inputs and outputs are implemented as adapters, enabling flexible interconnection in composite function blocks or sub-applications.
- **IEC 61131-3 Encapsulation**: The module encapsulates the proven counter logic from IEC 61131-3 in a 4diac-compliant component.
- **Simultaneous Triggering**: Every event (CU, R, PV) triggers a complete recalculation – even if only one parameter has changed.
- **License**: Released under the Eclipse Public License 2.0.

State Overview

The internal state is determined by the IEC 61131-3 CTU:

- **CV** (Current Meter Reading) – Integer value that is incremented with each CU event (unless a reset occurs).
- **Q** (Output) – Boolean value that becomes TRUE as soon as CV >= PV.
- On a **Reset** (R), CV is set to the value of R.D1 (usually 0) and Q is reset.
- On a new **PV**, only the threshold is updated; Q can change immediately if CV >= new PV.

The function block has no sequential states beyond these data dependencies.

Application Scenarios

- **Production Counting**: Recording of workpieces on a conveyor belt (CU = pulse generator, PV = batch size, Q = batch end).
- **Event Counter**: Counting sensor signals in combination with time-based evaluation.
- **Batch Processes**: Control of dosing or filling processes with an adjustable setpoint (PV).
- **Modular Automation**: Integration into larger function blocks via standardized adapter interfaces (AX/AI).

Comparison with Similar Function Blocks

Function Block	Properties
AI_FB_CTU	Adapter-based, uses IEC 61131-3 CTU, triggers on every input
Standard CTU (IEC 61131)	Inputs as events + data, no adapter concept, often directly bound to hardware
CTUD (Up/Down Counter)	Offers additional down counting, has a more complex interface
AX_CTU (hypothetical)	Could offer optimized event triggering (only on change)

The **AI_FB_CTU** impresses with its simple adapter connection, but may require an external filter to avoid unnecessary calls.

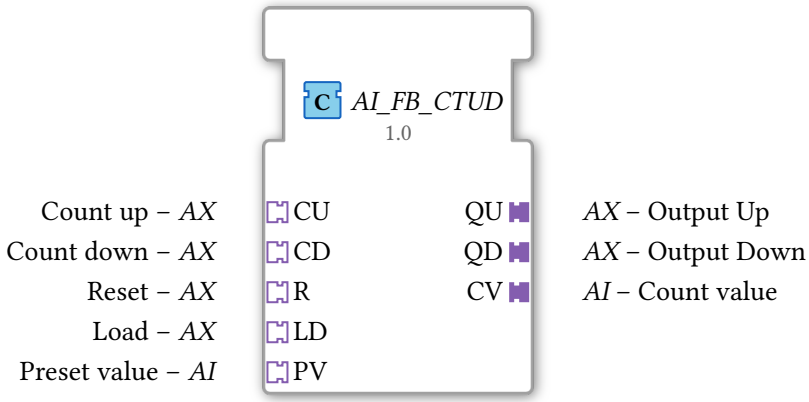
Conclusion

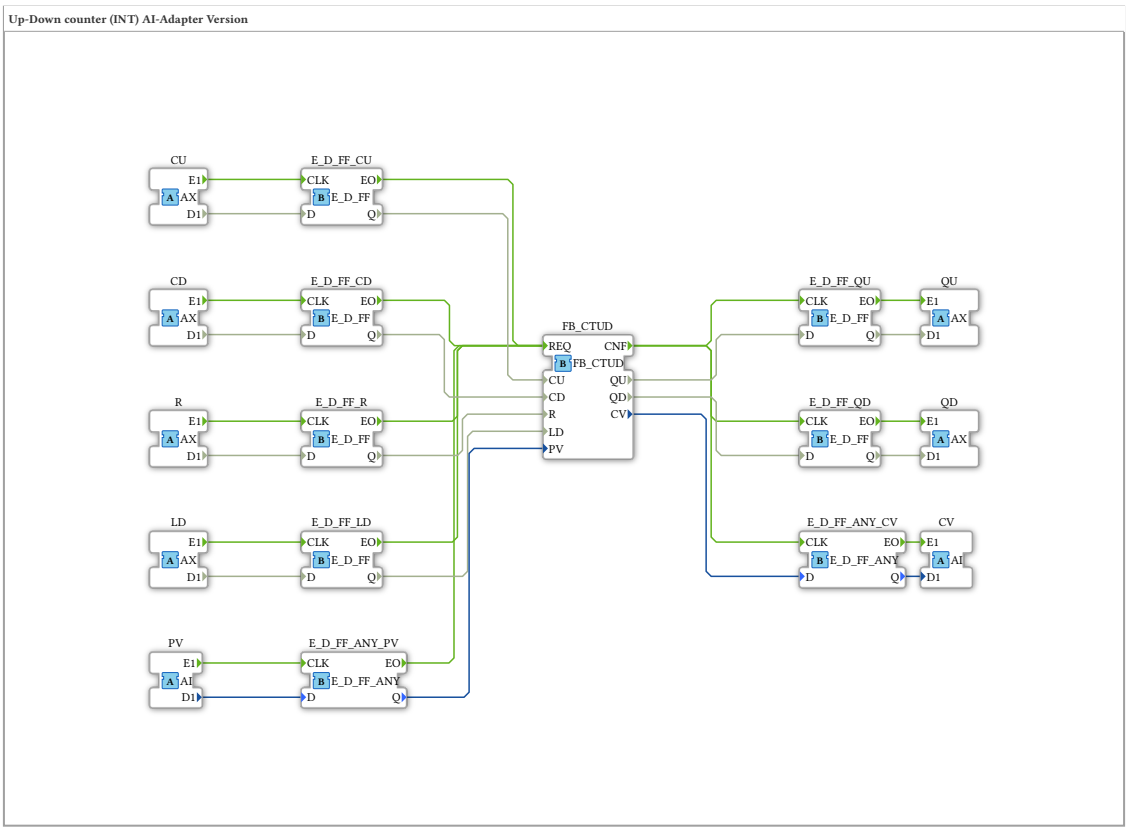
The **AI_FB_CTU** is a practical counter module for adapter-based automation with 4diac. It combines proven IEC 61131-3 logic with modern, modular interface technology. Its simple structure and clear functionality make it the first choice for all incrementing tasks where loose coupling via adapters is desired. Users should, however, be aware of the triggering on each event and implement differential filtering if necessary.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

AI_FB_CTUD





• – ○ • – ○ • – ○ •

Introduction

The function block **AI_FB_CTUD** implements an up/down counter of data type **INT** in an adapter-based variant. It encapsulates the IEC 61131-3 standard function block `FB_CTUD` and makes its functionality available to the 4diac IDE via the adapter interfaces (`AX` and `AI`). The function block is particularly suitable for use in industrial control systems that rely on event-driven communication.

Interface Structure

Event Inputs

The function block does not have separate event inputs in the traditional sense. Instead, events are received via the **Sockets** (adapters):

- **CU** (AX) – Count Up event
- **CD** (AX) – Count Down event
- **R** (AX) – Reset

- **LD** (AX) – Load preset value
- **PV** (AI) – Pass preset value

Each incoming event triggers an update of the internal counter.

Event Outputs

- **CNF** – Execution Confirmation, output with each update.
- **QU** (AX) – Up counter output
- **QD** (AX) – Down counter output
- **CV** (AI) – Current count value output

Data Inputs

All data is provided via the **sockets**:

- **CU.D** (BOOL) – Up counter pulse
- **CD.D** (BOOL) – Down counter pulse
- **R.D** (BOOL) – Reset signal
- **LD.D** (BOOL) – Enable preset value loading
- **PV.D** (INT) – Preset value

Data Outputs

- **QU.D** (BOOL) – Up counter flag (TRUE when counter reaches PV value)
- **QD.D** (BOOL) – Down counter flag (TRUE when counter reaches PV value) (has reached the value 0)
- **CV.D** (INT) – Current counter reading

Adapters

- **Sockets (Inputs):**
- `CU` (Type `adapter::types::unidirectional::AX`)
- `CD` (Type `adapter::types::unidirectional::AX`)
- `R` (Type `adapter::types::unidirectional::AX`)
- `LD` (Type `adapter::types::unidirectional::AX`)
- `PV` (Type `adapter::types::unidirectional::AI`)
- **Plugs (Outputs):**
- `QU` (Type `adapter::types::unidirectional::AX`)
- `QD` (Type `adapter::types::unidirectional::AX`)
- `CV` (Type `adapter::types::unidirectional::AI`)

Functionality

This function block operates as a pure **adapter wrapper** for the internal IEC 61131-3 FB `FB_CTUD`. Incoming adapter events (via the sockets) are aggregated at the event input `FB_CTUD.REQ`, while the corresponding data values (e.g., `CU.D`, `PV.D`) are forwarded to the corresponding inputs of the internal counter. During each execution of the internal function block, its outputs (`CV`, `QU`, `QD`) are read and output via the adapter plugs. Simultaneously, the event `CNF` is generated.

The internal counter behaves like a classic IEC 61131-3 up-down counter:

- A rising edge at `CU` increments the counter value by 1.
- A rising edge at `CD` decrements the counter value by 1.
- A rising edge at `R` resets the counter value to 0.
- A rising edge at `CU` increments the counter value by 1.
- A rising edge at `R` resets the counter value to 0. - The combination of `LD = TRUE` and an event at `PV` loads the current counter value to the value of `PV`.
- `QU` becomes TRUE when the counter value reaches or exceeds the preset value.
- `QD` becomes TRUE when the counter value reaches or falls below 0.

Technical Features

- **No Change Detection:** The function block outputs `CNF`, `QU`, `QD`, and `CV` **for every received event** (regardless of an actual value change). If a behavior is desired where the outputs only fire upon a change,

a preceding **AX_D_FF** (differentiator/filter) should be used. This is also explicitly recommended in the function block's source code.
- **Adapter Usage:** All communication takes place via unidirectional adapters (`AX` for Boolean events, `AI` for analog values). This enables loose coupling and reusability in different system topologies.
- **Unified Triggering:** All five event sources (CU, CD, R, LD, PV) trigger the execution of the internal counter via the common `REQ` input – there is no separate processing logic for each event.

State Overview

The component itself does not have an explicit state machine, but rather reflects the state of the internal `FB_CTUD`:

- **Counter Reading (INT):** Initial 0, changes via CU, CD, R, or LD.
- **Flags:** `QU` and `QD` are set based on a comparison of the counter reading and the preset value (or 0).
- **Preset Value:** Stored via the `PV` adapter and loaded with each `LD` event.

A detailed state machine description of the IEC 61131-3 counter can be found in the corresponding standard.

Application Scenarios

- **Bidirectional Event Counting:** E.g., detecting workpieces on a conveyor belt using sensors for infeed (CU) and outfeed (CD).
- **Inventory Management:** Adding and removing materials from a buffer.
- **Preset Positioning:** A system can be set to a predefined position (PV) and then tracked using CU/CD.
- **Resets After Cycle End:** Resetting the counter after reaching a specific quantity.

Comparison with Similar Function Blocks

- **FB_CTUD (Standard IEC 61131-3):** The `AI_FB_CTUD` is an **adapter version** of the same logic block. While the standard FB uses classic input/output pins, the adapter version communicates via event and data adapters. This simplifies integration into 4diac systems with adapter-based communication.
- **AI_FB_CTU / AI_FB_CTD:** Simple up and down counters without reverse direction. `AI_FB_CTUD` offers both counting directions in a single block.
- **AX_D_FF:** A differentiator/filter that fires only on changes. Can be used as a pre-function block to optimize the output behavior of `AI_FB_CTUD`.

Conclusion

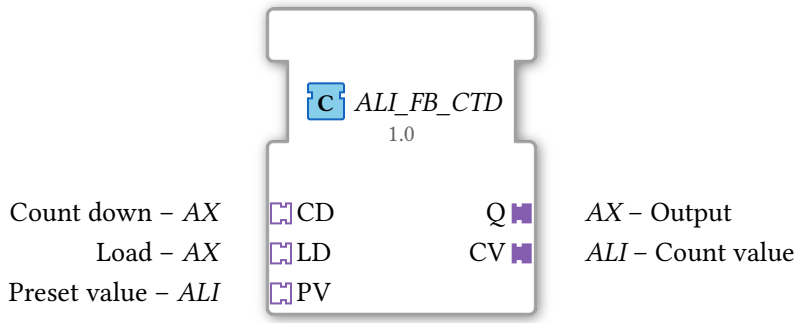
The `AI_FB_CTUD` is a powerful, adapter-based up/down counter that transfers the established IEC 61131-3 functionality into the event-driven world of 4diac. Its clean encapsulation and use of unidirectional adapters make it flexible and easy to integrate into modular automation solutions. The explicit recommendation to use a `AX_D_FF`For change-driven output, this shows that the component was deliberately kept simple – a strength for transparent and predictable systems.

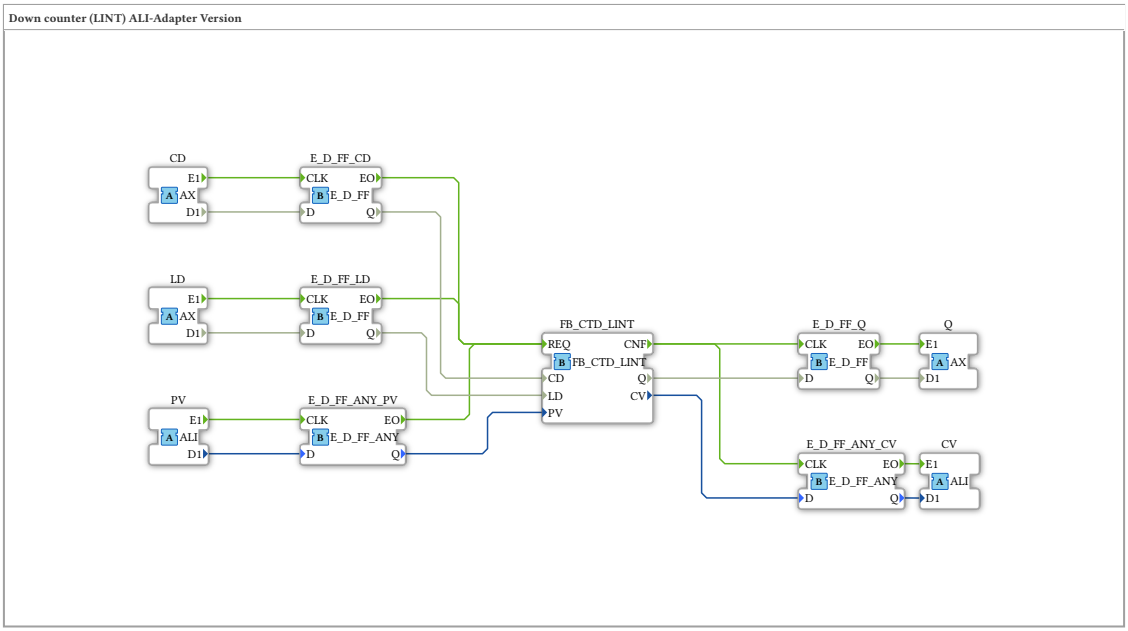
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ALI_FB_CTD





• – ◦ • – ◦ • – ◦ •

The **ALI_FB_CTD** is a down counter for LINT values (64-bit integers) with adapter interfaces according to IEC 61499. It is specifically designed for use with the unidirectional adapters `AX` (event/boolean carrier) and `ALI` (LINT value carrier). The function block encapsulates the standard FB `FB_CTD_LINT` from the library `iec61131::counters` and extends it with an adapter-based connection that enables loose coupling between the function blocks. The function block fires its event outputs with every update – regardless of whether the counter value actually changes. For change-triggered processing, it is recommended to use a `AX_D_FF` as a filter.

None – all event and data information is transmitted via adapter sockets (CD, LD, PV).

- **CNF** (Event): Acknowledgement of the executed operation (sent after every update).

There are no separate data inputs. The input data is provided via the following adapter sockets:

- **CD** (Socket, type `adapter::types::unidirectional::AX`): Down count pulse (activates the counting process).
- **PV** (Socket, type `adapter::types::unidirectional::ALI`): Preset value (LINT); the value is adopted during loading (LD).

There are no separate data outputs. The output data is provided via the following adapter plugs:

- **Q** (Plug, type `adapter::types::unidirectional::AX`): Output signal (Bool) – indicates whether the counter reading has reached **zero**.
- **CV** (Plug, type `adapter::types::unidirectional::ALI`): Current counter value (LINT).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Name	Direction	Type	Description
CD	Socket	AX	Down counter pulse (Event + Bool)

LD	Socket	AX	Charge pulse (Event + Bool)
PV	Socket	ALI	Preset value (LINT)
Q	Plug	AX	Output signal when counter value = 0
CV	Plug	ALI	Current counter value (LINT)

Functionality

The ALI_FB_CTD implements a down counter with charge function. The internal logic is implemented by the function block `FB_CTD_LINT`:

1. **Load**

An event at socket **LD** (via `LD.E1`) transfers the data value from **PV** (via `PV.D1`) to the internal counter. The counter value is then set to the preset value.

1. **Count Down**

An event at socket **CD** (via `CD.E1`) decrements the current counter value by 1, provided it is greater than 0.

1. **Output Signaling**

The current counter value is always output via plug **CV** (as a LINT).

- The plug **Q** returns `true` when the counter reaches **zero** (Boolean value via the AX adapter).
- The **CNF** event is sent after **every** processing operation (CD, LD, and PV).

1. **Event Passing**

The incoming events from CD, LD, and PV are all combined at the **REQ** input of the internal function block. Its **CNF** output is then distributed to the output adapters (Q.E1, CV.E1) and to the external event output CNF.

- **Always Triggering**: The function block sends the output events (Q.E1, CV.E1, CNF) with **every** update – even if the counter value does not change (e.g., after repeated CD resets to 0). For change-based triggering, a `AX_D_FF` (edge detector) must be interposed.
- **Adapter Coupling**: The use of unidirectional adapters (AX, ALI) enables flexible interconnection in IEC 61499 networks without rigid data and event connections.
- **LINT Values**: The counter operates with 64-bit integers, suitable for large counting ranges.
- **Internal Structure**: The function block delegates all counting logic to the proven standard FB `FB_CTD_LINT` from the `iec61131::counters` library.

The ALI_FB_CTD does **not have an explicit state machine** in the FB network. Its behavior is purely event-driven and combinatorial:

Input event	Action
LD activated	Counter reading = Preset value
CD activated	Decrement counter reading (if > 0)
PV activated	No counter change, but a CNF event is sent. The data value from PV is not directly adopted – an LD is required.

The current counter reading and the Boolean output are updated after each step.

- **Piece Counter in Manufacturing**: At the end of a production step, a pulse is sent to CD; the counter counts down from a predefined quantity (PV). At zero, Q is activated and signals, for example, “Batch complete”.
- **Time Control with Pulse Counting**: A clock generator sends pulses to CD; after a predefined number (PV), an event is triggered.
- **Maintenance Interval**: The counter serves as a countdown in maintenance cycles; an alarm is triggered when it reaches 0.

Function Block	Special Feature

ALI_FB_CTD (this function block)	Uses AX and ALI adapters; always triggers; suitable for adapter-based architectures.
FB_CTD_LINT (Standard)	Pure data/event function block without adapters; event triggering directly via the interfaces.
CTU (Upward Counter)	Counts upwards instead of downwards; different application logic.
Counter Function Blocks with Edge Suppression	They have their own filter logic (e.g., AX_D_FF) and only trigger on actual value changes.

The ALI_FB_CTD is particularly suitable for adapter-based environments where simple, reliable downcounting without additional filters is desired.

The **ALI_FB_CTD** is a robust downcounter for LINT values, specifically designed for adapter technology in IEC 61499 systems. It offers a clear, adapter-based interface and is reliable and tested thanks to the internal use of `FB_CTD_LINT`. Its "always-trigger" behavior makes it suitable for applications where every state change at the input should be immediately forwarded. Where change-based triggering is required, a simple filter block can be added. Overall, it represents a valuable addition to adapter-oriented automation solutions.

Technical Features

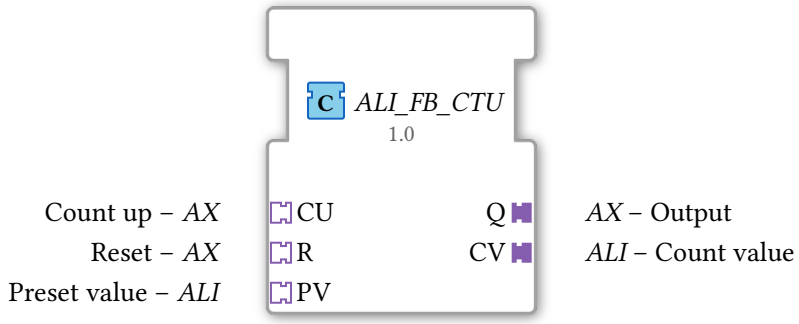
State Overview

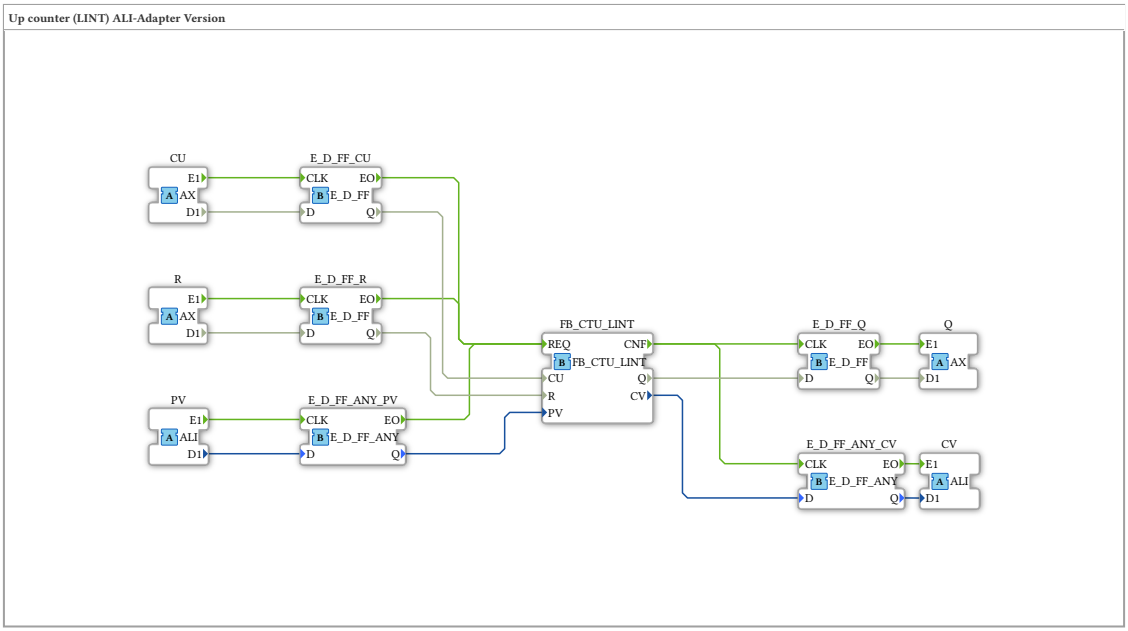
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

ALI_FB_CTU





• – ○ • – ○ • – ○ •

Introduction

The **ALI_FB_CTU** is a 64-bit integer counter (LINT). It serves as an adapter wrapper for the IEC 61131 counter module *FB_CTU_LINT* and provides all input and output signals via standardized adapter interfaces (AX and ALI). This ensures a clear separation of event and data flows and facilitates reuse in different project environments.

Interface Structure

Event Inputs

The module does not have separate, discrete event inputs. The triggering events are received via the adapters **CU**, **R**, and **PV**. Each of these adapters provides an event (E1) and a corresponding data value (D1). An incoming event at any of these adapters initiates processing.

Event Outputs

- **CNF** (Execution Confirmation): Outputs after each complete processing run, regardless of whether the CU, R, or PV triggered the event.
- **Q** (Adapter AX): Provides the output event (E1), which is output in parallel with the CNF event.
- **CV** (Adapter ALI): Also outputs an event (E1), synchronized with the CNF event.

Data Inputs

The module does not have separate, discrete data inputs. All input data is provided via the adapter interfaces:

- **CU.D1** – Count-Up Signal (BOOL)
- **R.D1** – Reset Signal (BOOL)
- **PV.D1** – Preset Value (LINT)

Data Outputs

The module does not have separate, discrete data outputs. The output data is provided via the adapter interfaces:

- **Q.D1** – Counter Output (BOOL): becomes TRUE as soon as the current counter reading is ≥ PV.
- **CV.D1** – Current Counter Reading (LINT)

Adapters

Direction	Name	Type	Description

Sockets (Inputs)	CU	AX	Count-Up: Event + BOOL signal
	R	AX	Reset: Event + BOOL signal
	PV	ALI	Preset value: Event + LINT value
Plugs (Outputs)	Q	AX	Output signal: Event + BOOL value
	CV	ALI	Counter reading: Event + LINT value

Functionality

The **ALI_FB_CTU** encapsulates an internal *FB_CTU_LINT*. Every incoming event at one of the three adapters **CU**, **R**, or **PV** is forwarded to the REQ event of the internal block. Simultaneously, the respective data values are passed to the corresponding inputs (*CU*, *R*, *PV*). The internal counter operates according to the IEC 61131 definition of a CTU:

- **On event at CU**: The counter reading is incremented by 1, provided the corresponding data bit is TRUE.
- **On event at R**: The counter reading is reset to 0.
- **On event at PV**: The passed value is stored as the new preset value.

After each processing operation, the internal block outputs a CNF event. This is passed to all three output adapters (CNF, Q, CV) so that all outputs are updated synchronously. The output **Q** becomes TRUE if and only if the current counter reading is greater than or equal to the preset value. The current counter reading can be read at any time via **CV**.

Technical Features

- **Adapter-based interface**: All signals are exchanged via standardized adapters (AX for binary, ALI for numeric values). This simplifies integration with heterogeneous systems and promotes reusability.
- **64-bit resolution**: The counter uses the LINT (Signed 64-bit Integer) data type, enabling very large counting ranges.
- **Shared trigger**: CU, R, and PV all trigger the same internal REQ. If multiple adapters are active simultaneously, all associated data values are processed in one step – the internal block evaluates the signals in parallel.
- **Note on output behavior**: The block fires the AX output event (Q.E1) with *every* update – even if the output state remains unchanged. If only "on-change" triggering is required, an AX_D_FF (turn-on delay/event filter) must be connected to the output.

State Overview

The internal state is determined by the counter reading (64-bit integer) and the Boolean output Q. There is no explicit state machine; the function block operates in an event-driven manner:

State Component	Possible Values	Description
Counter Reading (CV)	0 … 2^{63}-1	Current Count Value
Output Q	FALSE / TRUE	TRUE if CV ≥ PV

Application Scenarios

- **Event Counting**: Counting pulses in production facilities, conveyor belts, or energy meters.
- **Quantity Monitoring**: Recording the number of units produced and triggering a signal when a threshold is reached.
- **Reset-controlled counters**: Resetting the counter to zero after a manual or automatic reset.
- **Preset-based control**: Dynamically changing the setpoint during operation (e.g., batch changes).

Comparison with similar function blocks

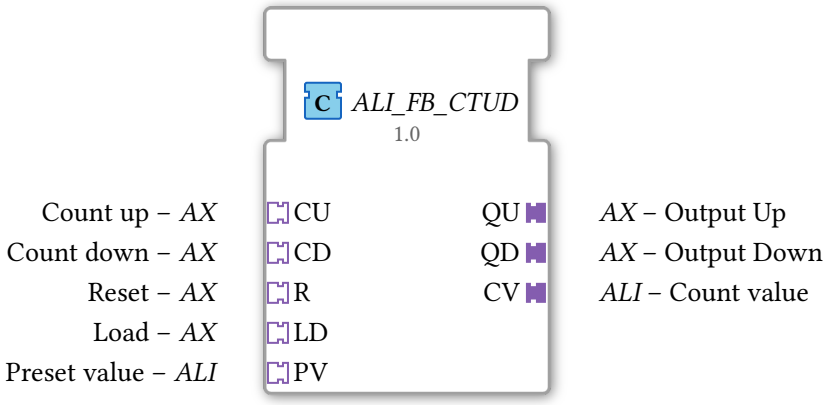
- **Compared to a direct FB_CTU_LINT**: The ALI_FB_CTU abstracts the interfaces at the adapter level. This allows the signals to be connected to other function blocks that also use AX/ALI adapters, regardless of the specific implementation.
- **Compared to a standard CTU with BOOL inputs**: The ALI_FB_CTU allows the connection of adapters that bundle both events and data. This reduces the number of separate lines and makes the design clearer.

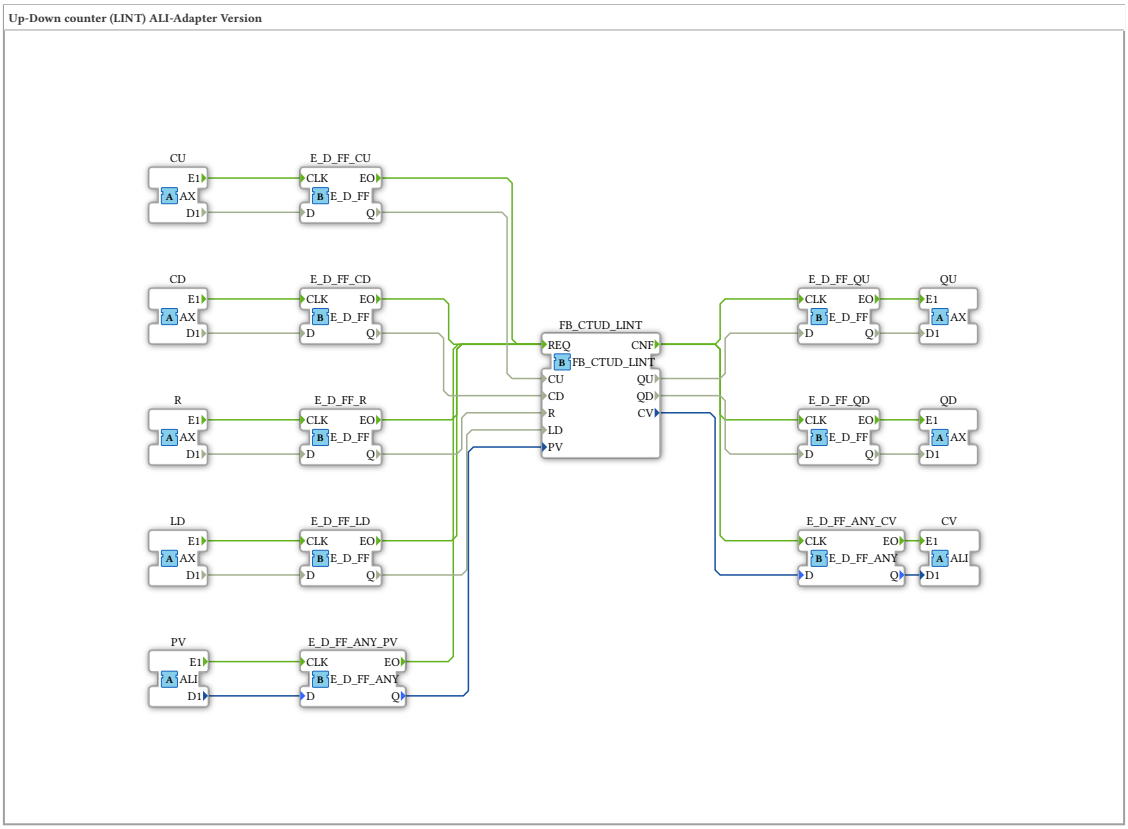
- **Compared to a CTU with INT/DINT**: The use of LINT covers a significantly larger value range, which is advantageous for high-resolution counting tasks or long-term monitoring.

Conclusion

The **ALI_FB_CTU** is a powerful, adapter-based up-counter for industrial automation environments. It combines proven IEC 61131 counter logic with modern adapter interfaces and offers simple, readable connectivity to other components. Its 64-bit resolution, shared event processing of all inputs, and the ability to dynamically set thresholds make it a flexible solution for a wide range of counting and monitoring tasks.

ALI_FB_CTUD





• – ○ • – ○ • – ○ •

Introduction

The function block **ALI_FB_CTUD** implements an up/down counter with a value range of type **LINT** (64-bit integer). It is specifically designed for use with **ALI adapters** and encapsulates a standard-

compliant counter according to IEC 61131-3 (FB_CTUD_LINT). Control and output are exclusively via adapter interfaces, allowing for flexible integration into adapter-based architectures.

Interface Structure

Event Inputs

The function block does not have direct event inputs. All control events are received via the **sockets** (adapters):

- **CU.E1** – Count Up event
- **CD.E1** – Count Down event
- **R.E1** – Reset event
- **LD.E1** – Load preset value event
- **PV.E1** – Apply preset value event

Note: All five events trigger a common internal processing cycle.
*Note: **CU.E1** – Count Up event *Note: **CU.E1** – Count Down event *CT ...

Event Outputs

- **CNF** (Event) – Execution Confirmation

Additionally, the following events are output via the **Plugs** (Output Adapters):

- **QU.E1** – Event when the counter value increases (Output Up)
- **QD.E1** – Event when the counter value decreases (Output Down)
- **CV.E1** – Event when the counter value changes (Count Value)

Special Note: These events are triggered with **every** update (regardless of the input event). For change-triggered triggering, it is recommended to use an AX_D_FF block beforehand.

Data Inputs

The data values are provided via the **sockets**:

- **CU.D1** (BOOL) – Count Up Enable
- **CD.D1** (BOOL) – Count Down Enable
- **R.D1** (BOOL) – Reset Signal
- **LD.D1** (BOOL) – Load Signal
- **PV.D1** (LINT) – Preset Value

Data Outputs

The results data are provided via the **plugs**:

- **QU.D1** (BOOL) – Signal: Counter value > 0 (e.g., for the “Up” output)
- **QD.D1** (BOOL) – Signal: Counter value < 0 (für „Down“-Ausgang, abhängig von interner Logik)
- **CV.D1** (LINT) – aktueller Zählerwert

Adapter

Typ	Richtung	Adapter-Typ	Beschreibung
CU	Socket	`adapter::types::unidirectional::AX`	Aufwärtszähl-Eingang (Ereignis+BOOL)
CD	Socket	`adapter::types::unidirectional::AX`	Abwärtszähl-Eingang (Ereignis+BOOL)
R	Socket	`adapter::types::unidirectional::AX`	Reset-Eingang (Ereignis+BOOL)
LD	Socket	`adapter::types::unidirectional::AX`	Load-Eingang (Ereignis+BOOL)
PV	Socket	`adapter::types::unidirectional::ALI`	Preset-Wert-Eingang (Ereignis+LINT)
QU	Plug	`adapter::types::unidirectional::AX`	Aufwärts-Ausgang (Ereignis+BOOL)
QD	Plug	`adapter::types::unidirectional::AX`	Abwärts-Ausgang (Ereignis+BOOL)
CV	Plug	`adapter::types::unidirectional::ALI`	Zählerwert-Ausgang (Ereignis+LINT)

Funktionsweise

Der Baustein enthält einen internen Funktionsblock **FB_CTUD_LINT**, der die eigentliche Zählerlogik nach IEC 61131-3 implementiert. Jedes Ereignis an einem der fünf Sockets (CU.E1, CD.E1, R.E1, LD.E1, PV.E1) löst eine Verarbeitung aus: Die zugehörigen booleschen Daten (CU.D1, CD.D1, R.D1, LD.D1) und der Preset-Wert (PV.D1) werden an den internen Baustein weitergeleitet und dort synchron ausgewertet. Der interne Baustein berechnet daraufhin den neuen Zählerstand und die Ausgangssignale QU, QD und CV. Nach Abschluss der Berechnung wird das Ausgangsereignis **CNF** gesendet und gleichzeitig werden die Ereignisse **QU.E1**, **QD.E1** und **CV.E1** auf den entsprechenden Plugs ausgegeben.

Die Verwendung von Adaptern ermöglicht eine lose Kopplung: Die eigentlichen Signalquellen (z. B. Sensoren oder Bedienelemente) und -senken (z. B. Aktoren oder Anzeigen) werden über Adapterverbindungen angebunden, ohne dass die internen Daten- und Ereignisleitungen direkt sichtbar sind.

Technische Besonderheiten

- **Datenbereich**: 64-Bit vorzeichenbehaftete Ganzzahl (LINT), geeignet für große Zählerstände.
- **Adapter-getriebene Schnittstelle**: Keine direkten Ereignis- oder Dateneingänge; alle Steuerungen erfolgen über AX- und ALI-Adapter.
- **Immerwährende Ereignisausgabe**: Die Ausgangsereignisse (QU.E1, QD.E1, CV.E1) werden bei **jedem** Verarbeitungszyklus ausgegeben, unabhängig davon, ob sich der Zählerstand tatsächlich geändert hat. Dieses Verhalten ist im Quellcode explizit dokumentiert – für eine änderungsbasierte Auslösung wird die Verwendung eines AX_D_FF-Filters empfohlen.
- **Interner Standard-Baustein**: Der FB_CTUD_LINT entspricht der IEC 61131-3-Definition eines Auf-/Abwärtszählers.

Zustandsübersicht

Eine formale Zustandsmaschine ist nicht extern sichtbar. Der interne Zustand besteht aus dem aktuellen **Zählerwert** (CV) und den internen Flags **QU** (z. B. „Wert > 0“) and **QD** (e.g., “Value < 0”). The state is updated with each input event according to the following priority:

1. **Reset (R)**: sets CV to 0.
2. **Load (LD)**: adopts PV as the new CV.
3. **Count Up (CU)**: increments CV by 1 if CU.D1 = TRUE.
4. **Count Down (CD)**: decrements CV by 1 if CD.D1 = TRUE.

The flags QU and QD are then calculated from the new CV.

Application Scenarios

- **Production Counting**: Recording workpieces or cycles – counting up on entry, counting down on exit.
- **Position Monitoring**: Counting up/down steps in a linear drive.
- **Inventory Management**: Counting storage units with manual correction via Load/Reset.
- **Adapter-based control systems**: Integration into architectures that rely on unidirectional adapters (AX/ALI), e.g., distributed automation nodes.

Comparison with similar function blocks

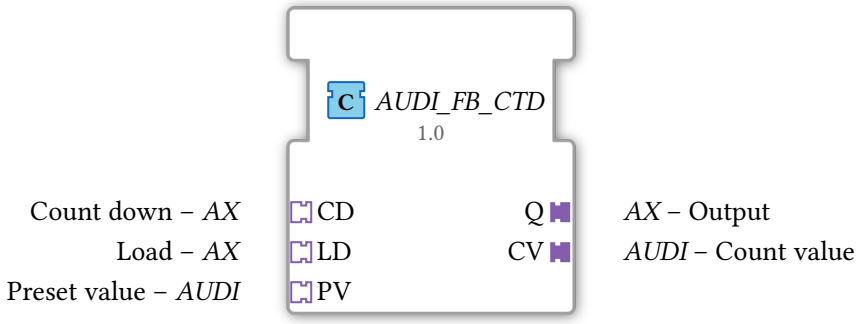
The standard IEC 61131-3 function block **CTUD** (e.g., `FB_CTUD_INT`) typically works with smaller data types (INT, DINT) and offers direct event and data inputs. The **ALI_FB_CTUD** extends this concept with:

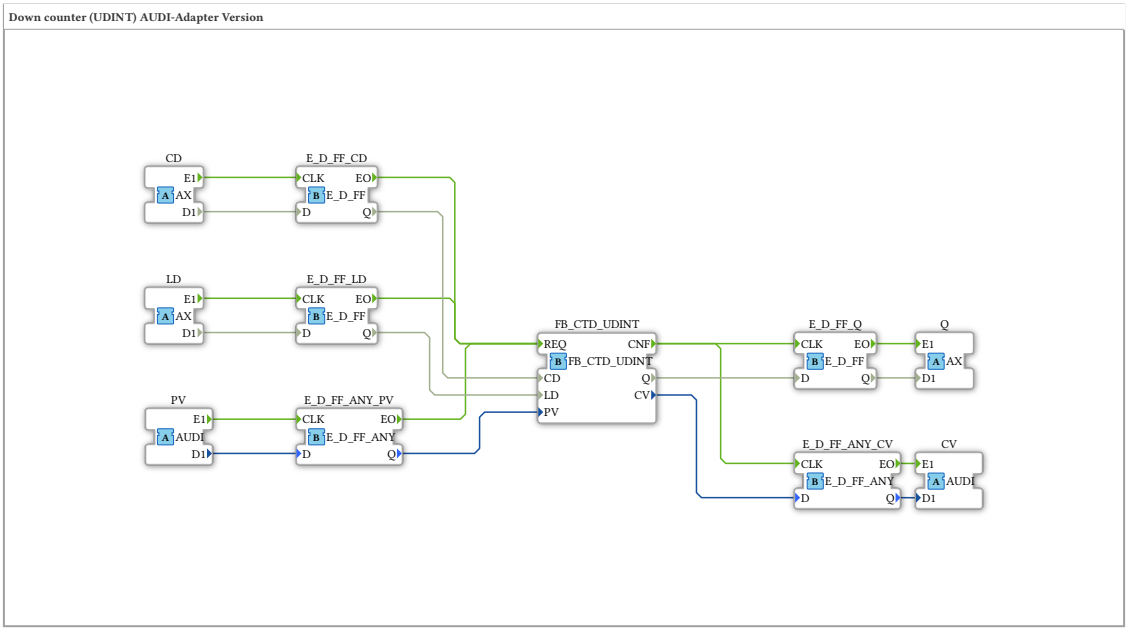
- **LINT data type** for very large counter ranges.
- **Adapter interfaces** (AX/ALI) instead of free inputs/outputs.
- **Continuous event output** as opposed to purely change-driven output.

Conclusion

The **ALI_FB_CTUD** is a powerful up/down counter for 64-bit numbers that integrates seamlessly into adapter-based automation environments. Its structure allows for a clear separation of control and data flows and facilitates reuse in modular control systems. The continuous event output should be considered during project planning – a downstream filter block may be required to prevent unwanted edge counts.

AUDI_FB_CTD





• – ○ • – ○ • – ○ •

Introduction

The function block **AUDI_FB_CTD** implements a down counter based on the UDINT data type. It is specifically designed for use with adapter interfaces and uses the types `AX` and `AUDI` for event-driven communication. The actual counting logic is executed by an internal function block of type `iec61131::counters::FB_CTD_UDINT`. This function block fires an event on the output channel `CD`, `LD`, or `PV` with every update—regardless of whether the input was triggered by `CD`, `LD`, or `PV`. For change-based triggering, the use of a downstream `AX_D_FF` filter is recommended.

Interface Structure

Event Inputs

This function block does not have direct event inputs. All events are supplied via the connected **sockets** (adapter inputs).

Socket (Adapter)	Type	Description
`CD`	`AX`	Countdown Event
LD`	`AX`	Load Event
PV`	`AUDI`	Preset Value Event

Event Outputs

Name	Type	Description
CNF`	Event	Acknowledge event after each processing step

Data Inputs

Data is transmitted via sockets as part of the adapter protocol.

Socket (Adapter)	Data Type	Meaning
CD.D1	BOOL	Countdown command (TRUE = count)
LD.D1	BOOL	Load preset value command
PV.D1	UDINT	Preset counter value

Data Outputs

The output data is displayed via the plugs.

Plug (Adapter)	Data Type	Meaning
Q.D1	BOOL	Counter value = 0 (TRUE when reached)
CV.D1	UDINT	Current Counter Value

Adapter

Direction	Name	Type	Description
Socket	CD	AX	Event-Driven Count Pulse (Down)
Socket	LD	AX	Event-Driven Preset Loading
Socket	PV	AUDI	Event-Driven Preset Selection
Plug	Q	AX	Output signal: Counter reading = 0
Plug	CV	AUDI	Output signal: Current counter value

Functionality

Internally, a standardized IEC 61131 down counter (FB_CTD_UDINT) is used. The adapter events CD.E1, LD.E1, and PV.E1 are combined into a single REQ event of the internal module. This causes the internal counter to be recalculated with every external event.

- **CD (Count Down)**: If the data signal CD.D1 = TRUE is present, the current counter value is decremented by 1. With FALSE, the value remains unchanged.
- **LD (Load)**: If LD.D1 = TRUE is present, the counter is set to the value of PV.D1.
- **PV (Preset Value)**: This event updates the stored preset value; a subsequent LD event adopts this value.

After each calculation, the output data CV (current counter value) and Q (TRUE if CV = 0) are output via the corresponding plugs. Simultaneously, the acknowledgment event CNF is triggered.

Technical Features

- **Event Behavior**: For every input event (CD, LD, or PV), the output channel Q is activated. This means that a Q.E1 event is sent even if the counter value or the Q state has not changed. This can lead to an unnecessarily high number of events in the subsequent logic.
- **Filtering Recommendation**: If the output event should only be triggered when Q actually changes state, an **AX_D_FF** (flip-flop) should be used as a filter. This suppresses repetitions.
- **Adapter-Based Connection**: All inputs and outputs are made via standardized adapters (AX, AUDI). This simplifies modular wiring and reuse in different environments.

State Overview

The component has no explicit states. The internal counter operates according to the rules of an IEC 61131 CTD component. The current counter value is determined by the data signals.

Application Scenarios

- **Event-driven Down Count**: Used in automation systems where a counter value is decremented by external events (e.g., number of workpieces processed).
- **Modular Control Components**: Integration into adapter-based architectures where standardized interfaces facilitate data exchange between different functional blocks.
- **Timer Simulation**: Combined with a clock generator to count a defined number of cycles and trigger an action when zero is reached.

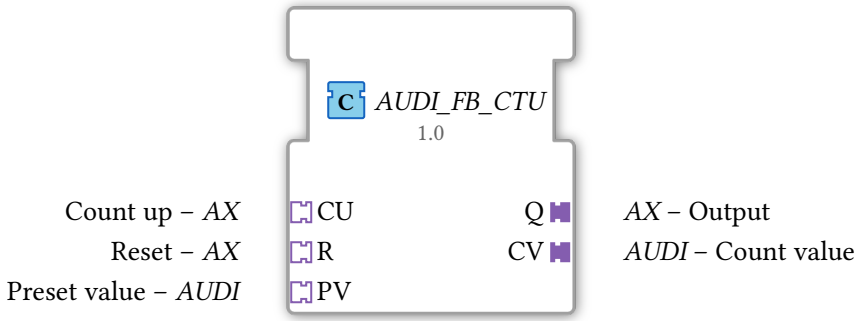
Comparison with Similar Function Blocks

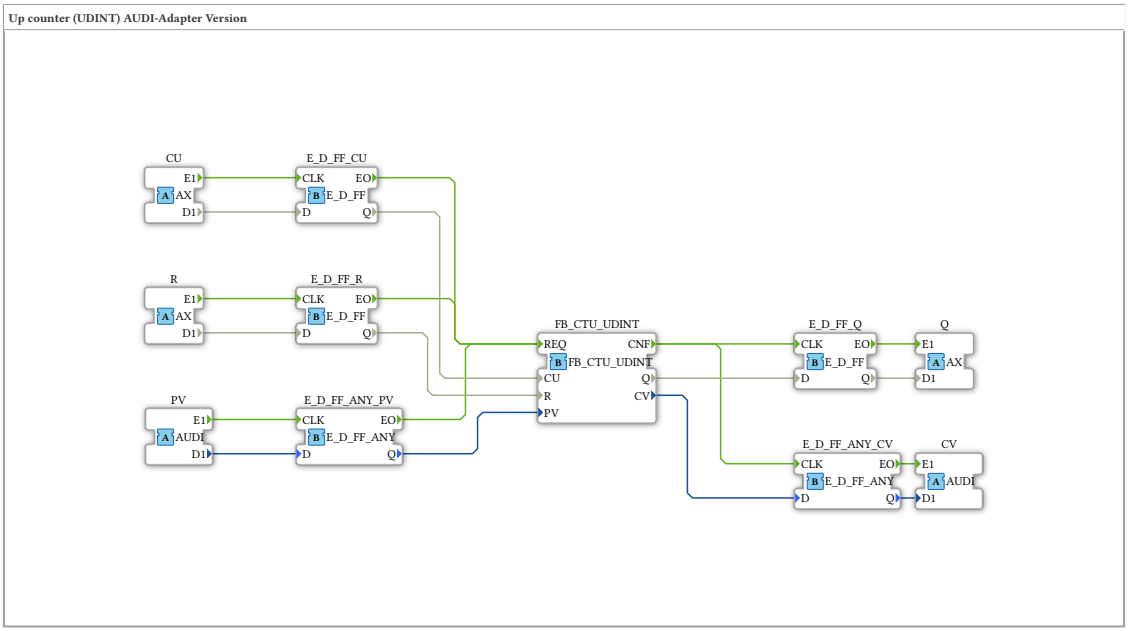
- **Standard CTD** (e.g., `FB_CTD_UDINT`): Has direct input/output variables, no adapters. The `AUDI_FB_CTD`, on the other hand, encapsulates the logic and provides the signals via adapters, enabling a clean separation of event and data channels.
- **Upward Counting Unit (CTU)**: Unlike the `AUDI_FB_CTD`, a CTU counts upwards. Both can be implemented using adapters, but are designed for different counting directions.
- **Always fire feature**: Many standard counters only output a signal when the initial state changes. The `AUDI_FB_CTD` does this with every processing step, which is a unique feature and can be an advantage or disadvantage depending on the application.

Conclusion

The `AUDI_FB_CTD`This is a flexible, adapter-based down counter for UDINT values. Its strength lies in its modular connectivity via standardized interfaces. However, developers should be aware of its unique event behavior (output pulse with each update) and correct it with a filter if necessary. This function block is well-suited for event-driven counting tasks in heterogeneous control environments.

AUDI_FB_CTU





• – ○ • – ○ • – ○ •

Introduction

The **AUDI_FB_CTU** is an up counter for the **UDINT** data type (32-bit unsigned) that connects via standard **AUDI adapters**. The function block encapsulates the IEC 61131 counting function in an adapter-based architecture and allows triggering of count pulses, reset, and preset operations via adapter events. Every counting-relevant operation (count pulse, reset, setting the preset value) necessarily triggers an output event – regardless of whether the counter value or the output value changes. If only change-based triggering is required, a downstream AX_D_FF function block must be used.

Interface Structure

Event Inputs

The module does not have traditional event inputs. Instead, the triggering events are introduced **via the adapter sockets**:

- **CU.E1** – Event from the *Count up* adapter: a rising pulse increments the counter by 1.
- **R.E1** – Event from the *Reset* adapter: resets the counter to 0.
- **PV.E1** – Event from the *Preset value* adapter: adopts the current value at the preset input as the new preset value (takes effect immediately, without changing the counter).

All three events are internally multiplexed onto the **REQ** event of the embedded IEC function block `FB_CTU_UDINT`. This means that each of these external pulses triggers processing.

Event Outputs

- **CNF** – Confirmation Event (EventOutput): This event is output after each processing of an incoming event. It is connected to the output adapters so that the connected components are informed about a completed counting operation.

In addition, the following **adapter events** are triggered (via internal wiring):

- **Q.E1** – Event of the *Output* adapter (type AX): Signals the availability of a new output value Q.
- **CV.E1** – Event of the *Count value* adapter (type AUDI): Signals the availability of a new counter reading CV.

These adapter events fire **simultaneously with CNF**, meaning that all three output events are activated simultaneously for every counting-relevant event.

Data Inputs

Data is also provided **via the adapter sockets**:

- **CU.D1** (UDINT, via adapter CU) – Count pulse: Only the value 0 or 1 is evaluated; a positive edge transition (0⟶1) results in an increment. (The adapter provides the event itself; additionally, the data value can be used, for example, as an edge-weighted signal.)
- **R.D1** (UDINT, via adapter R) – Reset signal: A value ≠ 0 resets the counter.
- **PV.D1** (UDINT, via adapter PV) – Preset value: The value present here is adopted as the new preset value upon arrival of the PV.E1 event. (The data value is only read in the event of an explicit PV event.)

Data Outputs

- **Q.D1** (BOOL, via adapter Q) – Output signal: `TRUE` when the current meter reading is ≥ **PV** (Preset Output).
- **CV.D1** (UDINT, via adapter CV) – Current meter reading (Count Value).

Adapters

Adapter Type	Direction	Name	Short Description
`AX` (unidirectional)	Socket (Input)	CU	Provides the count pulse event and an edge-weighted data value.
`AX` (unidirectional)	Socket (Input)	R	Provides the reset event and reset switch.
`AUDI` (unidirectional)	Socket (Input)	PV	Provides the preset event and preset numeric value.
`AX` (unidirectional)	Plug (Output)	Q	Provides the output binary (Q) as an event and data value.
`AUDI` (unidirectional)	Plug (Output)	CV	Provides the current counter reading as an event and data value.

Functionality

The **AUDI_FB_CTU** delegates all its counting logic to the internal IEC block `FB_CTU_UDINT`. For each incoming event (CU.E1, R.E1, or PV.E1), its **REQ** input is activated. The internal counter then executes the following logic:

1. **Reset first**: If **R** ≠ 0, the counter is set to 0.
2. **Count pulse**: If **CU** ≠ 0 and a rising edge transition has occurred since the last call, CV is incremented by 1.
3. **Preset takeover**: If **PV.E1** has been triggered, the value at PV is stored as the new threshold (PV).
4. **Comparison**: If CV ≥ PV (stored preset value), **Q** = TRUE; otherwise, it is FALSE.

After processing, the events **CNF**, **Q.E1**, and **CV.E1** are output simultaneously, regardless of whether Q or CV has changed.

Technical Features

- **Adapter-Based**: All inputs and outputs are implemented as standardized unidirectional adapters (`AX` for binary, `AUDI` for numeric values). This allows for flexible interconnection in adapter-oriented architectures.
- **Event Multiplexing**: The three input events (CU, R, PV) are multiplexed onto a common REQ. This simplifies internal wiring but means that a reset or preset operation also triggers the output event.
- **No On-Change Filtering**: As described in the header comment, the function block fires **always** on every processing operation. For change-only triggering, an AX_D_FF must be added (e.g., on Q and CV).
- **Data Width**: All counter and preset values are implemented as **UDINT** (32-bit unsigned).
- **State Maintenance**: The internal counter stores the current counter value and the preset threshold across multiple calls.

State Overview

The function block does **not have its own state machine** (ECC). The state logic is entirely contained within the embedded IEC function block `FB_CTU_UDINT`. Its essential states are:

- **Idle**: Waiting for an event (CU.E1, R.E1, or PV.E1).
- **Processing** (during internal processing): Evaluating the input data (CU, R, PV), updating the counter and the Q and CV outputs.
- **Emitting**: Outputting the CNF, Q.E1, and CV.E1 events.

An explicit error state is not provided.

Application Scenarios

- **Event counting in adapter-based automation systems** – e.g., counting workpieces on a conveyor belt, where the sensor is connected via an AX adapter.
- **Time or pulse counter with reset** – The reset adapter can, for example, be a manual push button.
- **Reaching a threshold** – The PV input allows for dynamic threshold setting; as soon as the counter reading reaches or exceeds this value, Q becomes active.
- **Integration into modular systems** – Thanks to the standardized adapters (AUDI, AX), the module can be easily integrated into existing adapter-based libraries.

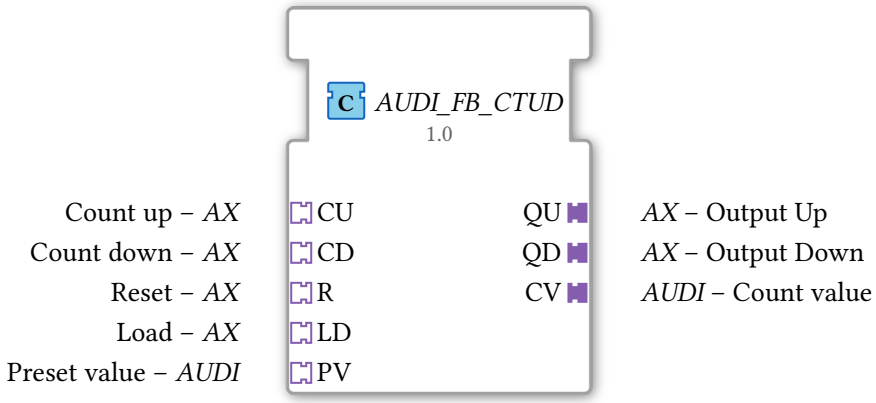
Comparison with similar modules

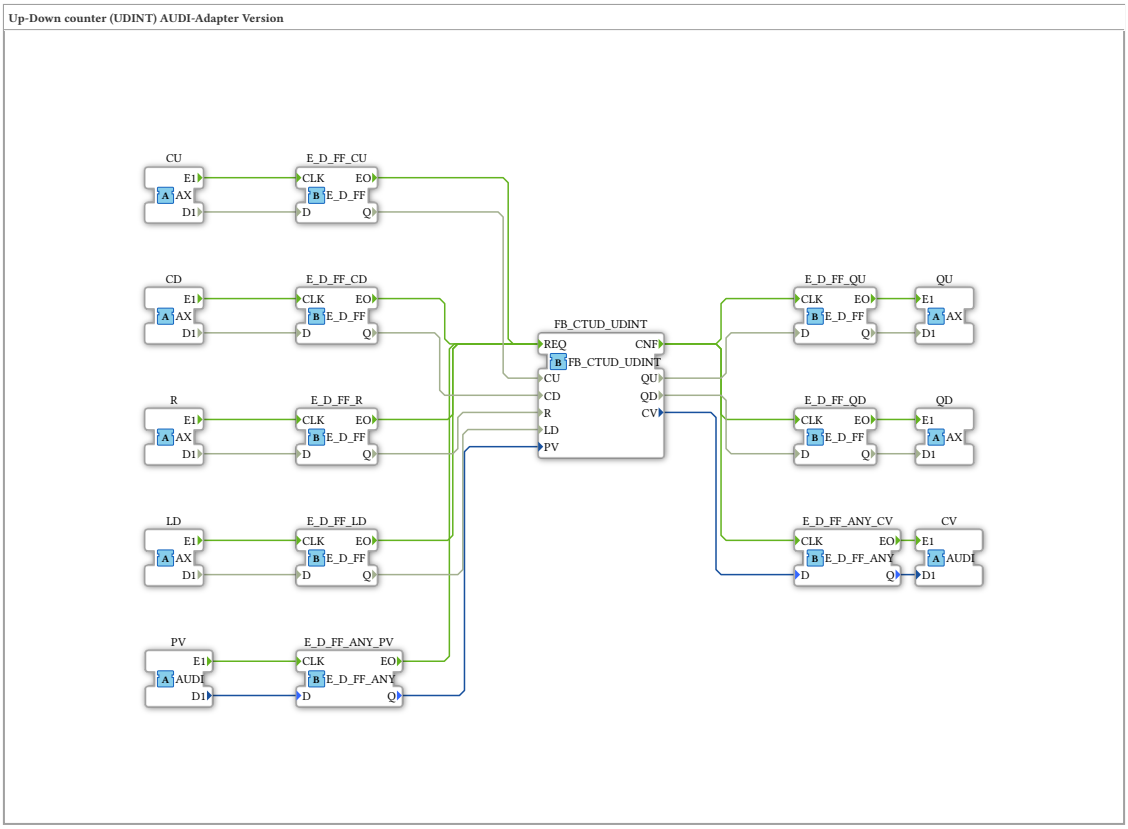
- **Unlike classic IEC CTU modules** (e.g.,`CTU_UDINT` (without adapter): These have direct event and data inputs but do not require adapter wrappers. The AUDI_FB_CTU offers the advantage of a standardized interface but requires specific adapter mapping.
- **Unlike CTU blocks with on-change triggering**: Blocks that only fire when there are actual value changes avoid unnecessary event floods. The AUDI_FB_CTU always fires, which can be compensated for by adding an AX_D_FF.
- **Comparison to the AUDI_FB_CTD** (down counter): The internal IEC block would then be `FB_CTD_UDINT`; interfaces and behavior are analogous, only the counting direction and the condition for Q differ (Q is active when CV ≤ 0).

Conclusion

The **AUDI_FB_CTU** is a robust, adapter-based up counter that can be seamlessly integrated into adapter-oriented automation solutions. High interoperability is achieved through the encapsulation of a standardized IEC counter and the use of AUDI/AX adapters. The characteristic of firing on every event is acceptable for many applications or can be corrected by downstream filtering. This component is particularly suitable for modular, expandable control architectures based on adapter protocols.

AUDI_FB_CTUD





• – ○ • – ○ • – ○ •

Introduction

The **AUDI_FB_CTUD** is an up/down counter with a value range for unsigned 32-bit integers (UDINT). It is implemented as a pure adapter version, meaning all inputs and outputs are provided via adapter interfaces. The function block encapsulates the standardized IEC 61131 function block `FB_CTUD_UDINT` and extends its functionality with an adapter-based interface. All output adapters are cyclically updated upon each triggering event (CU, CD, R, LD, or PV). If change-triggered (edge-sensitive) triggering is required, the use of a `AX_D_FF` as a filter block is recommended.

Interface Structure

Event Inputs

The function block has no discrete event inputs at the top level. All events are transmitted via the incoming adapter interfaces (sockets) of type `AX`.

Event Outputs

Name	Type	Comment
`CNF`	`Event`	Execution acknowledgment – is output each time an incoming event is processed.

Events are also sent via the outgoing adapters (plugs).

Data Inputs

There are no direct data inputs at the top level. All data values are provided via sockets `CU.D1`, `CD.D1`, `R.D1`, `LD.D1`, and `PV.D1`.

Data Outputs

There are no direct data outputs at the top level. The result data is output via plugs `QU.D1`, `QD.D1`, and `CV.D1`.

Adapters

Direction	Name	Type	Comment
Socket (Input)	`CU`	`adapter::types::unidirectional::AX`	Count up – Event and data (BOOL) for counting up
Socket (Input)	`CD`	`adapter::types::unidirectional::AX`	Count down – Event and data (BOOL) for counting down
Socket (Input)	`R`	`adapter::types::unidirectional::AX`	Reset – Event and data (BOOL) for resetting the counter
Socket (Input)	`LD`	`adapter::types::unidirectional::AX`	Load – Event and data (BOOL) for loading the preset value
Socket (Input)	`PV`	`adapter::types::unidirectional::AUDI`	Preset value – Event and data (UDINT) for the default value
Plug (Output)	`QU`	`adapter::types::unidirectional::AX`	Output Up – Event and data (BOOL), active when the current count reaches 0
Plug (Output)	`QD`	`adapter::types::unidirectional::AX`	Output Down – Event and data (BOOL), active when the current count reaches >= PV
Plug (Output)	`CV`	`adapter::types::unidirectional::AUDI`	Count value – Event and data (UDINT) with the current count value

Functionality

The `AUDI_FB_CTUD` internally uses the `FB_CTUD_UDINT` function block from the IEC 61131 library. All events received via the sockets (from `CU`, `CD`, `R`, `LD`, and `PV`) are routed to the `REQ` input of the internal function block. The corresponding data is routed to the appropriate inputs (`CU`, `CD`, `R`, `LD`, `PV`). The internal function block processes the request and updates its outputs. Upon successful processing, the event `CNF` is triggered, and simultaneously, the output adapters `QU`, `QD`, and `CV` are supplied with their new data and events.

The internal function block processes the request and updates its outputs. **Important:** This function block fires the output events (`QU.E1`, `QD.E1`, `CV.E1`) during **every** processing of an incoming event. No edge detection is performed.

Technical Features

- **Pure Adapter Interface:** This function block has no traditional inputs/outputs, but only adapters (sockets and plugs). This enables flexible integration into adapter-based automation architectures.
- **Use of Unidirectional Adapters:** The adapters `AX` and `AUDI` are defined as unidirectional. This ensures a clear direction for data and event flows.
- **Encapsulation of the IEC 61131 standard:** The counting logic conforms to the standardized `CTUD` function block, which increases portability and maintainability.
- **Range:** The counter value is implemented as `UDINT` (unsigned, 0…2^{32}–1).

State Overview

The internal `FB_CTUD_UDINT` cycles through the following states, depending on the incoming commands:

Condition	Action
`CU` = TRUE (rising edge) and `CD` = FALSE	Counter increments by 1
`CD` = TRUE (rising edge) and `CU` = FALSE	Counter decreases by 1
`R` = TRUE (rising edge)	Counter is reset to 0
`LD` = TRUE (rising edge)	Counter is set to the value of `PV`
Simultaneous `CU` and `CD` edges	Counter value remains unchanged (priority: none)

QU = TRUE	Current value = 0
QD = TRUE	Current value ≥ PV

Application Scenarios

- **Piece Counter in Production Plants:** Recording of products on a conveyor line, using CU and CD for accurate counting during forward and reverse movements.
- **Inventory Management:** Counting inbound and outbound shipments, with reset during inventory and load for target values.
- **Event Counter in Building Automation:** Recording of passenger flows (e.g., in elevators) with an additional reset function.
- **Adapter-based systems:** Integration into distributed control environments based on the adapter concept of the 4diac IDE (e.g., AUTOSAR-like communication).

Comparison with similar components

Feature	AUDI_FB_CTUD	Standard CTUD (IEC 61131)
Interface	Pure adapters (sockets/plugs)	Direct event and data inputs/outputs
Data format	UDINT (via AUDI adapter)	UDINT (direct)
Event behavior	Every incoming event triggers all outputs	Event is only output when relevant changes occur
Flexibility	Higher through adapter-based cabling	Lower, but simpler in classic PLC environments
Recommended filter	Additional AX_D_FF for edge-triggered output	No filter needed, as it is edge-sensitive

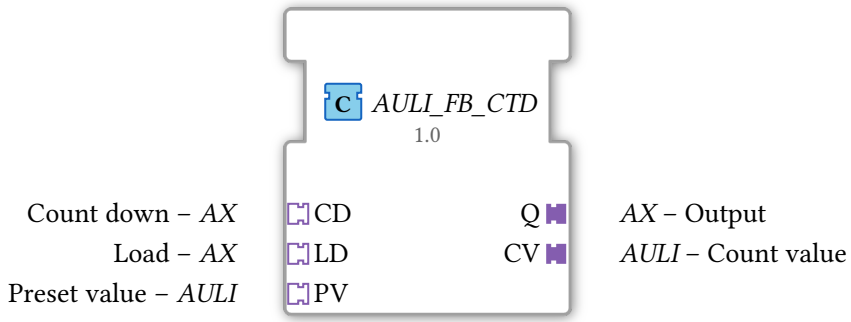
Conclusion

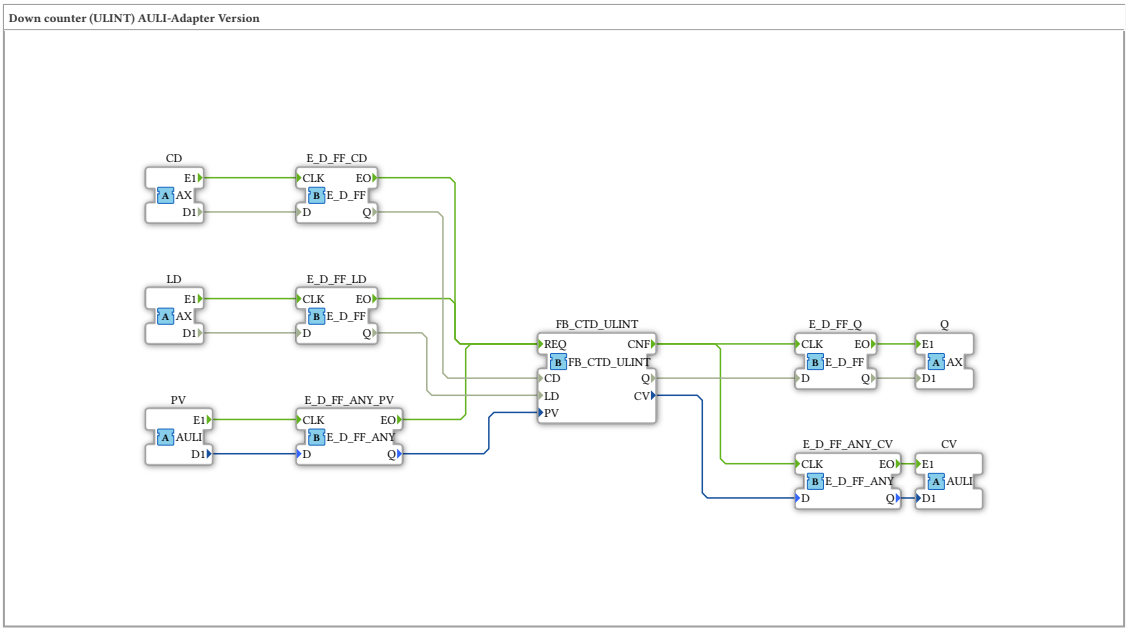
The AUDI_FB_CTUDis a high-performance up/down counter whose adapter-based interface makes it particularly suitable for modern, modular automation solutions. It combines the proven IEC 61131 counting logic with the flexibility of 4diac adapter technology. Users should note, however, that output events are triggered with every processing operation, not just changes – a downstream edge filter may be necessary. With its clear structure and standardized data types, it represents a robust component for counting tasks in industrial control systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AULI_FB_CTD





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_FB_CTD** implements a **down counter** based on the data type `ULINT` (unsigned long integer). It is implemented as an **adapter version** and encapsulates the standard function block `FB_CTD_ULINT` from the IEC 61131-3 library. The block allows modular connection via the adapter interfaces CD (Count Down), LD (Load), and PV (Preset Value), as well as the output of the current count value (CV) and a binary signal (Q) via corresponding plug adapters.

Interface Structure

Event Inputs

The function block has **no direct event inputs**. Triggering occurs exclusively via the **event channels of the socket adapters**:

Adapter	Event Port	Description
CD	CD.E1	Count Down Pulse
LD	LD.E1	Load Preset Value Pulse
PV	PV.E1	Preset Value Update

Note: Each of these events triggers processing of the internal counter. The counter is always recalculated – regardless of whether the value actually changes.

Event Outputs

Name	Description
CNF	Confirmation after each processing

Additionally, the plug adapters provide **two event outputs**:

Adapter	Event Port	Description
Q	Q.E1	Outputs with every processing
CV	CV.E1	Outputs with every processing

Note: Since the events fire with every update (CD, LD, PV), it is recommended to filter a change edge with a `AX_D_FF` if necessary (see Technical Specifications).

Data Inputs

Data is also provided via the **socket adapters**:

Adapter	Data Port	Data Type	Description
CD	CD.D1	`BOOL`	Count pulse (rising edge)
LD	LD.D1	`BOOL`	Load command (rising edge)
PV	PV.D1	`ULINT`	Preset value (loaded during LD or PV update)

Data Outputs

Data is output via the **Plug Adapters**:

Adapter	Data Port	Data Type	Description
Q	Q.D1	`BOOL`	Counter reading = 0 (output signal)
CV	CV.D1	`ULINT`	Current counter reading

Adapter

Direction	Adapter type	Short description
Socket	`AX` (bidirectional)	Countdown control (Event + BOOL)
Socket	`AX` (bidirectional)	Load Control (Event + BOOL)
Socket	`AULI` (bidirectional)	Preset Value (Event + ULINT)
Plug	`AX` (bidirectional)	Output Q (Event + BOOL)
Plug	`AULI` (bidirectional)	Counter Value Output (Event + ULINT)

Functionality

This function block encapsulates the IEC 61131-3 function `FB_CTD_ULINT`. The internal logic is triggered by events from the three socket adapters:

1. **CD Event (Count Down):**

On a rising edge of `CD.D1` and a simultaneous event on `CD.E1`, the counter value is decremented by 1 (provided it is > 0). The new value is output at plug `CV`.

1. **LD Event (Load):**

On a rising edge of `LD.D1` and an event on `LD.E1`, the current preset value (`PV.D1`) is loaded into the counter. The counter value is then set to the preset value.

1. **PV Event (Preset Value Update):**

An event on `PV.E1` updates the internally stored preset value (without changing the counter). This is useful for dynamically changing the preset during operation.

After each processing operation, the confirmation event `CNF` is sent, along with the events on the plug adapters `Q.E1` and `CV.E1`. The data `Q.D1` and `CV.D1` are updated accordingly.

Technical Features

- **ULINT Data Type:**

The module uses unsigned 64-bit integers (ULINT), enabling counting ranges from 0 to 2^{64}-1 – suitable for very large counting tasks.

- **Adapter-Based Connection:**

All inputs and outputs are via adapters (`AX` for binary signals, `AULI` for ULINT values). This allows for clean encapsulation and modular wiring in the 4diac IDE.

- **Event Output on Every Update:**

The module fires the output events on every incoming event (CD, LD, PV) – even if the counter reading or output value does not change. This creates **permanent triggering** of the downstream network.
→ **Recommendation:** Use a `AX_D_FF` (differentiator/filter) at the outputs if you only want to react to value changes.

- **No internal state machine:**

The function block itself does not have its own state machine; the state logic (e.g., rising edge detection) is handled by the internal `FB_CTD_ULINT`.

State Overview

Internally, the function block only manages the **counter value** (CV) and the **current preset value** (PV). There is no explicit state machine. The possible actions are:

State / Action	Trigger	Result
Count Down	CD Event & CD.D1=TRUE	CV := CV - 1 (if CV>0)
Load	LD event & LD.D1=TRUE	CV := PV (current preset)
Preset Update	PV event	PV is overwritten (CV remains unchanged)

The output `Q` is set to `TRUE` when `CV = 0` is present; otherwise, it is `FALSE`..

Application Scenarios

- **Large-Range Piece Counter:**

Detection of production quantities with a value range > 32 bits (e.g., 10 billion pieces).

- **Preselection or Sequence Control:**

Used as a downward counter in a sequence where a signal is triggered upon reaching the value 0 (e.g., batch end).

- **Dynamic Preset Values:**

Changing the end-of-count value during operation via the PV event, without affecting the current counter reading.

- **Modular Systems:**

Integration into larger control architectures that rely entirely on adapter communication (e.g., via the Eclipse 4diac adapter mechanism).

Comparison with Similar Components

Component	Data Type	Interface	Special Feature
`FB_CTD_ULINT`	ULINT	Standard I/O	Basic down counter without adapter
AULI_FB_CTD	ULINT	Adapter (AX, AULI)	Adapter-encapsulated, all events result in an update
`FB_CTD` (Standard)	INT/UINT	Standard I/O	Usually 16-bit or 32-bit, fixed event logic

The **AULI_FB_CTD** offers more flexible integration into complex networks due to its adapter coupling, but has the "side effect" that output events are sent even if the values remain unchanged. For applications that should only fire upon a value change, the basic function block `FB_CTD_ULINT` or a combination with an edge detector (`AX_D_FF`) is preferable.

Conclusion

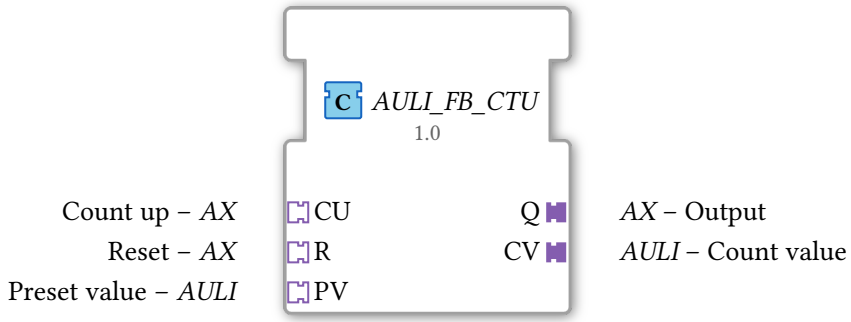
The **AULI_FB_CTD** is a powerful down counter for 64-bit values that is integrated into 4diac projects via adapter interfaces. It is particularly suitable for large counting ranges and modular control topologies. The continuous event output requires careful handling of the downstream network, but this can be mitigated by suitable filter blocks (e.g., `AX_D_FF`). Thanks to the encapsulation of the proven `FB_CTD_ULINT`, its operation is robust and standards-compliant.

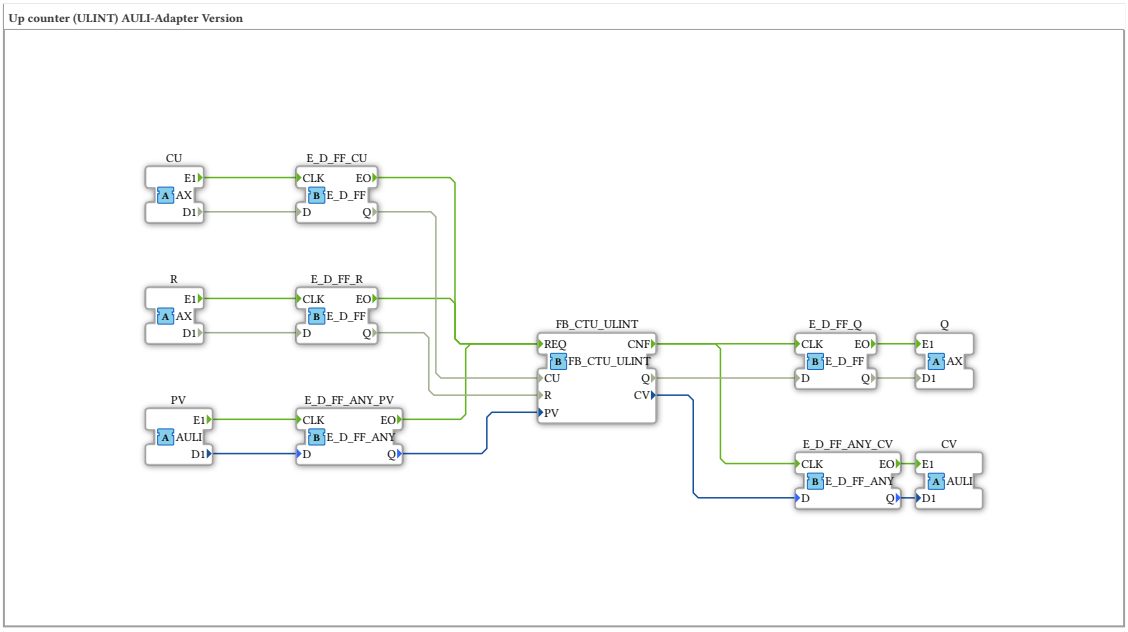
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AULI_FB_CTU





• – ○ • – ○ • – ○ •

The function block **AULI_FB_CTU** implements an up counter based on the ULINT (unsigned long integer) data type. It is designed as an IEC 61499-compliant component with an **AULI adapter interface**. Internally, the counter uses a standardized IEC 61131-3 CTU component (`FB_CTU_ULINT`) and makes its functionality available via adapter ports. Events are triggered with every update – regardless of a value change.

No dedicated event inputs. Events are received exclusively via the **AX adapters** of the sockets (CU, R, PV).

Name	Type	Comment
CNF	Event	Execution Confirmation – triggered with each counter update

No direct data inputs. The required data (CU pulse, reset signal, preset value) is provided via the **AULI or AX adapters** of the sockets.

No direct data outputs. The result values (current counter value CV and threshold status Q) are output via the **AULI or AX adapters** of the plugs.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

| Direction | Adapter Name | Type | Comment | **Socket (Input)** | CU | `adapter::types::unidirectional::AX` | Count-Up Pulse (Event + Data) | **Socket (Input)** | R | `adapter::types::unidirectional::AX` | Reset Pulse (Event + Data) | **Socket (Input)** | PV | `adapter::types::unidirectional::AULI` | Preset Value (Comparison Value for Q Output) | **Plug (Output)** | Q | `adapter::types::unidirectional::AX` | Counter Reading ≥ Preset? (Boolean Status) | | **Plug (Output)** | CV | `adapter::types::unidirectional::AULI` | Current Meter Reading (ULINT) |

Functionality

This module encapsulates an IEC 61131-3 compliant CTU meter (`FB_CTU_ULINT`). Events and data are received via the **socket adapters**:

- **CU Update**: A rising edge at CU triggers a counting step (CV = CV + 1).
- **R Update**: A reset resets the meter reading (CV = 0).
- **PV Update**: The preset value is applied; CV is compared to PV with each counting step.

The event at a socket is internally routed to the `REQ` input of the embedded CTU block. After processing, the `CNF` event of the CTU block is forwarded to all three outputs:

- **Q-Plug**: Outputs the status event (Q.E1) – sent **with every update**, regardless of whether the boolean value has changed.
- **CV-Plug**: Outputs the event (CV.E1) – also **with every update**.
- **CNF-Event**: Provided as an external event.

The data flows are analogous: The socket data (CU.D1, R.D1, PV.D1) is transferred to the corresponding data inputs of the internal counter. The output data (Q.D1, CV.D1) is output via the plugs.

- **Always Update**: The function block triggers the output events (Q.E1, CV.E1, CNF) with **every** incoming event (CU, R, PV) – even if the counter reading or the Q value does not change.
- **Filter Recommendation**: If triggering is only desired upon an actual value change, the use of an **AX_D_FF** function block as a pre-filter is recommended (see comment in the identification).
- **Data Width**: The counter uses the ULINT data type (64-bit unsigned integer), suitable for large counting ranges.
- **Adapter Interface**: All inputs and outputs are handled via standardized adapters (`AX` for Boolean/ event-based values, `AULI` for ULINT values). This facilitates modular design and reuse in adapter-based projects.
- **No Top-Level State Machine**: The function block is purely transaction-based: Every incoming event triggers immediate processing.

The function block does not have its own state machine. The internal `FB_CTU_ULINT` can pass through the following states (simplified):

- **Idle**: Waiting for an event.
- **Increment**: Upon a CU event, the internal counter is incremented.
- **Reset**: Upon a R event, the counter is reset to 0.
- **Preset Update**: The comparison value is updated upon a PV event.

After each processing operation, the outputs are updated and the CNF event is sent.

- **High-Frequency Pulse Counting**: Acquisition of counting pulses (e.g., encoders, clock signals) with 64-bit resolution.
- **Threshold Monitoring**: Counting of products or events up to a configurable threshold (PV); the Q output is activated upon reaching this threshold.

- **Reset-Controlled Batch Counting**: After an R pulse, the counter starts from zero; the current count is output via CV.
- **Adapter-Based Automation Modules**: Easy integration into systems that rely on AX/AULI adapters (e.g., standardized I/O modules).
- **Standard CTU (IEC 61131-3)**: Typically offers separate event and data ports. The `AULI_FB_CTU` encapsulates these in adapters, which simplifies bus-level wiring.
- **CTU with Change Detection**: Some counter function blocks only fire output events when a value changes. This function block triggers with every update – resulting in deterministic behavior (one event per input event), but may require downstream filters.
- **CTU as a Basic Function Block**: Other implementations use their own state machine; this function block delegates the logic to an embedded block, thus remaining lean.

The **AULI_FB_CTU** offers an IEC 61499-compliant, adapter-based increment counter function with ULINT data width. Its strengths lie in its modular interface and reliable event triggering with every update. For applications requiring only change detection, an external filter block (AX_D_FF) is recommended. Overall, the block is suitable for a wide range of counting scenarios in industrial automation, especially in adapter-oriented architectures.

Technical Features

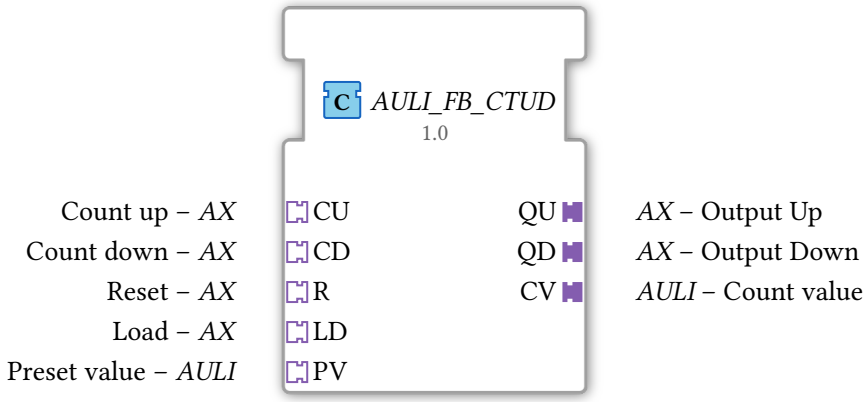
State Overview

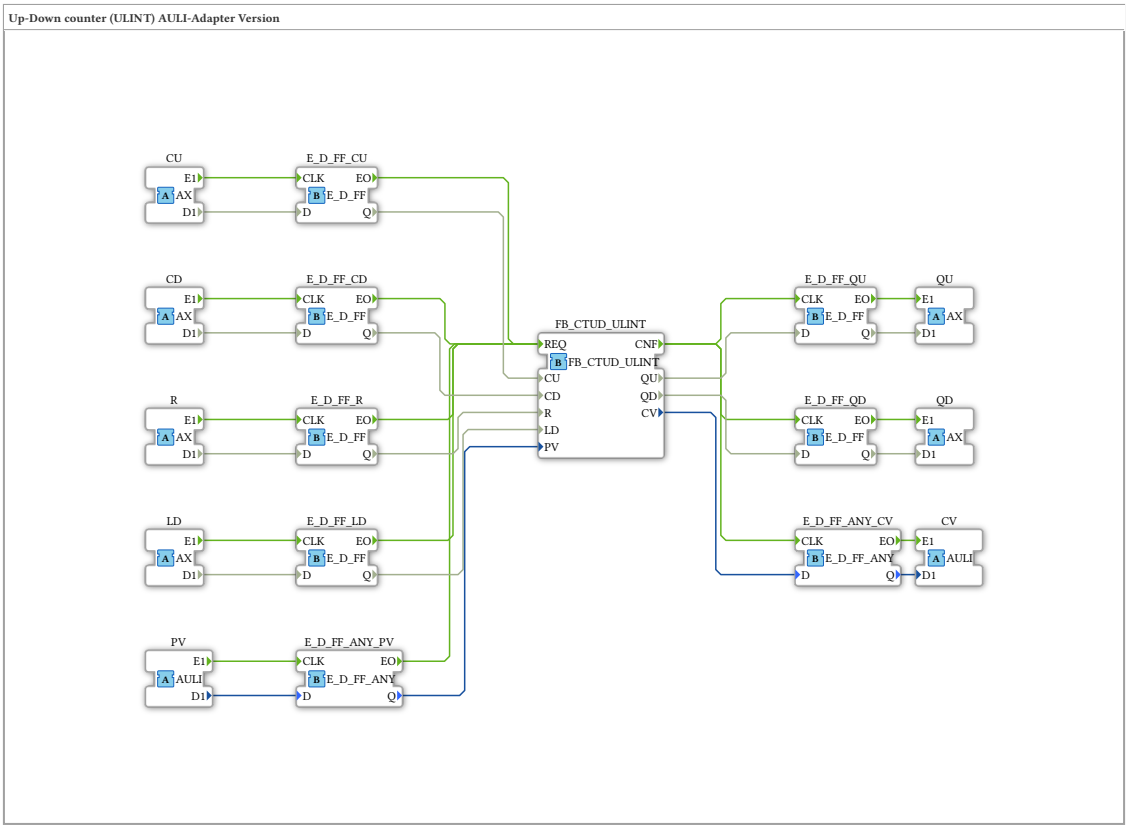
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AULI_FB_CTUD





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_FB_CTUD** implements an up/down counter for unsigned 64-bit integers (ULINT). It exclusively uses adapter interfaces according to the IEC 61499-2 standard for event and data connections. The block encapsulates the standard FB `FB_CTUD_ULINT` and extends its inputs and outputs with adapter-based connections, enabling flexible and standardized integration into adapter-oriented architectures.

Interface Structure

Event Inputs

The block does not have any traditional event inputs. Instead, the counter events are transmitted via the **adapter sockets**:

- **CU** (Count Up) – Event via `CU.E1`
- **CD** (Count Down) – Event via `CD.E1`
- **R** (Reset) – Event via `R.E1`
- **LD** (Load) – Event via `LD.E1`
- **PV** (Preset Value) – Event via `PV.E1`

Event Outputs

- **CNF** – Standard event output, triggered with every counter update.

Additionally, events are output via the **adapter plugs**:

- **QU.E1** – Counter overflow event (maximum value reached)
- **QD.E1** – Counter underflow event (value 0)
- **CV.E1** – Update the current counter reading

Data Inputs

The block does not have separate data inputs. All input data is provided via the adapter sockets:

- **CU.D1** (BOOL) – Enables increment (TRUE increments the counter at CU.E1)

- **CD.D1** (BOOL) – Enables decrement
- **R.D1** (BOOL) – Enables reset
- **LD.D1** (BOOL) – Enables load
- **PV.D1** (ULINT) – Value for the load operation

Data Outputs

There are no separate data outputs. The output data is sent via the adapter plugs:

- **QU.D1** (BOOL) – Signals reaching the maximum value
- **QD.D1** (BOOL) – Signals reaching 0
- **CV.D1** (ULINT) – Current counter reading

Adapter

Adapter	Direction	Type	Purpose
CU	Socket	AX	Count Up – Event + Boolean enable
CD	Socket	AX	Count Down – Event + Boolean enable
R	Socket	AX	Reset – Event + Boolean enable
LD	Socket	AX	Load – Event + Boolean enable
PV	Socket	AULI	Preset Value – Event + ULINT value
QU	Plug	AX	Output Up – Event + Boolean State
QD	Plug	AX	Output Down – Event + Boolean State
CV	Plug	AULI	Current Value – Event + ULINT Counter Value

Functionality

The function block reacts to events at the input adapters (`CU.E1`, `CD.E1`, `R.E1`, `LD.E1`, `PV.E1`). For each event, the corresponding data value (`DX.D1`) is evaluated:

- **CU**: When `CU.E1` and `CU.D1 = TRUE` occur, the counter is incremented.
- **CD**: When `CD.E1` and `CD.D1 = TRUE` are encountered, the counter is decremented.
- **R**: When `R.E1` and `R.D1 = TRUE` are encountered, the counter is reset to 0.
- **LD**: When `LD.E1` and `LD.D1 = TRUE` are encountered, the counter is set to the value of `PV.D1`.

After each processing operation, the event `CNF` is output. Simultaneously, the output adapters are updated:

- `QU.D1` becomes `TRUE` when the counter reaches its maximum value (`2^64 - 1`).
- `QD.D1` becomes `TRUE` when the counter value is 0.
- `CV.D1` provides the current counter value.

The entire process is synchronous – each incoming event triggers a calculation and subsequently the output of the results.

Technical Features

- **Adapter-based communication**: All inputs and outputs are handled via adapters (sockets/plugs). This enables a decoupled connection between components and simplifies reuse in different contexts.
- **Unidirectional Adapters**: The adapters used (AX, AULI) are unidirectional – sockets receive, plugs send.
- **Trigger Behavior**: The function block fires the output events (`QU.E1`, `QD.E1`, `CV.E1`) on **every** counter update (including reset or load). For change-based triggering, an AX_D_FF (differentiator) must be used.
- **Value Range**: The counter operates in the range 0 ... 2^64-1 (ULINT). Overflows are signaled by `QU`, underflows by `QD`.

State Overview

The function block does not have an explicit state machine; The counter behavior is implemented by the internal function `FB_CTUD_ULINT`. Essentially, three states are distinguished:

1. **Normal Operation** – The counter value is between 1 and `2^64-2`. Neither `QU` nor `QD` are active.
2. **Overflow** – The counter has reached its maximum value. `QU.D1 = TRUE`.

3. **Underflow** – The counter reading is 0. `QD.D1 = TRUE`.

After a reset (`R`) or load (`LD`), the counter can immediately jump to one of these states.

Application Scenarios

- **Industrial Piece Counters**: Recording production quantities with up-and-down counting (e.g., good/bad parts).
- **Pallet or Workpiece Tracking**: Counting inputs and outputs in a buffer memory.
- **Event-Driven Systems**: Combination with sensors (light barriers, proximity switches) via the adapter interfaces.
- **Adapter-Based Control Architectures**: Seamless integration into projects that utilize the socket/plug concept of IEC 61499-2.

Comparison with Similar Components

Component	Properties
`FB_CTUD_ULINT`	Same counter logic, but with separate event and data inputs/outputs (no adapters).
AULI_FB_CTU`	Up counter only, as an adapter version.
AULI_FB_CTD`	Down counter only, as an adapter version.
CTUD` (Standard IEC 61499)	Similar functionality, but usually with different data types (e.g., INT) and without an adapter.

The `AULI_FB_CTUD` combines up and down counting in one block and offers particularly flexible integration with other adapter-based components via its adapter interfaces.

Conclusion

The `AULI_FB_CTUD`This is a high-performance up/down counter for ULINT values, distinguished by its complete adapter interface. It combines proven counting logic with the flexibility of the IEC 61499-2 adapter concept. Thanks to its standardized event and data transmission via plugs and sockets, it is particularly suitable for modular, reusable control applications. Its ease of use and integrated overflow/underflow detection make it a reliable building block in many industrial counting tasks.

Numerical Index (iec61131-3)

Here you will find all exercises from the `iec61131-3` workspace in ascending order.

AX_FB_F_TRIG

ECC: AX_FB_F_TRIG

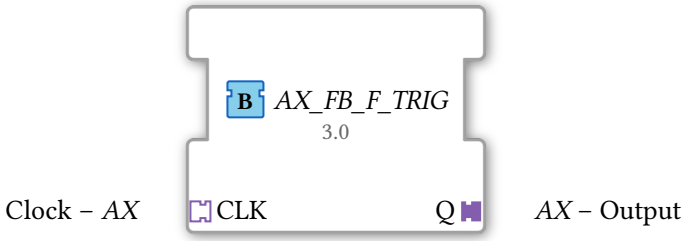


Insert placeholder for the function block image here, if available

• – ◦ • – ◦ • – ◦ •

Introduction

The **AX_FB_F_TRIG** function block (Falling Edge Trigger with AX Adapter) is used to detect a falling edge (signal change from TRUE to FALSE) within an adapter-based architecture. It implements the logic of a standard IEC 61131-3 function block (`F_TRIG`), but uses **AX Adapters** to encapsulate data and event signals, thus reducing wiring complexity in IEC 61499 applications.



Interface Structure

The function block does not use traditional separate event and data pins, but instead utilizes adapter interfaces.

Event Inputs

- *No direct event inputs (encapsulated in adapter `CLK`).*

Event Outputs

- *No direct event outputs (encapsulated in adapter `Q`).*

Data Inputs

- *No direct data inputs (encapsulated in adapter `CLK`).*

Data Outputs

- *No direct data outputs (encapsulated in adapter `Q`).*

Adapter

Name	Type	Comment	Description
CLK	`adapter::types::unidirectional::AX`	Clock	**Socket (Input):** The input adapter that provides the signal to be monitored. Typically contains an event (`E1`) and a Boolean data value (`D1`).
Q	`adapter::types::unidirectional::AX`	Output	**Plug (Output):** The output adapter that sends the result of the edge detection.

Functionality

The function block monitors the logical state of the data input in adapter `CLK` (field `D1`). For each event at input `CLK` (`E1`), the following algorithm is executed:

1. The function block checks whether a transition from **TRUE** (High) to **FALSE** (Low) has occurred. 2. An internal memory (`MEM`) is used to store the inverted state of the last cycle.
2. The logic is: `Q.D1 := NOT CLK.D1 AND NOT MEM`.
3. The memory is updated: `MEM := NOT CLK.D1`.

Behavior:

- When the input signal changes from 1 to 0, the output `Q.D1` is set to TRUE for exactly one cycle.
- In all other cases (0->0, 0->1, 1->1), the output remains FALSE.
- The result is signaled via the event `Q.E1`.

Technical Features

- **Adapter Technology:** By using the `AX` adapter (generic data type, interpreted here as Boolean), event flow and data flow are bundled. This results in clearer function block diagrams.
- **Internal Storage:** The function block has an internal variable `MEM` (initial value TRUE) to reference the previous state of the input signal.
- **Minimalist ECC:** The state machine (ECC) consists of only a single state, which ensures very fast and direct algorithm execution.

State Overview

The function block's Execution Control Chart (ECC) is very simple:

- **REQ (State):** This is the only state.
- **Transition:** It is triggered by the event `CLK.E1` (from the input adapter).
- **Action:** Executes the algorithm `REQ` and then triggers the output event `Q.E1`.

Application Scenarios

The **AX_FB_F_TRIG** is suitable for various control tasks where "switching off" or "releasing" needs to be detected:

- **Button Evaluation:** Detects when a button is released (falling edge).
- **Sensors:** Triggers an action when a workpiece leaves a sensor area (light barrier becomes clear).
- **Alarm Reset:** Detects when an error signal is no longer present.
- **Structured Programming:** Used in complex applications that rely heavily on adapters to avoid "spaghetti code" (many individual connection lines).

⚖️ Comparison with similar function blocks

Function Block	Difference
F_TRIG (Standard)	Uses separate `CLK` (BOOL) and `Q` (BOOL) pins, as well as separate `REQ`/`CNF` events. Functionally identical, but with a different interface.
AX_FB_R_TRIG	The counterpart to this function block. It detects the **rising edge (0 to 1).

E_F_TRIG	Event-based trigger in IEC 61499, often operates purely at the event level without encapsulated data adapters.

Conclusion

The **AX_FB_F_TRIG** is an essential component for developers who rely on adapter-based designs within the 4diac IDE and IEC 61499. It offers the reliable functionality of classic edge detection, packaged in a modern, bandwidth-efficient interface.

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

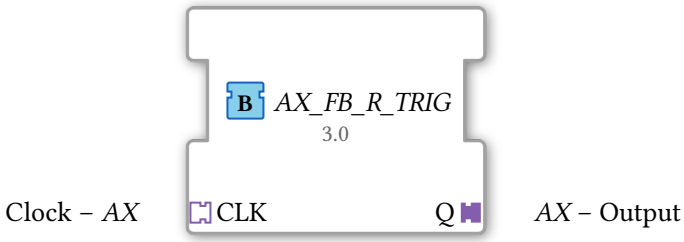
AX_FB_R_TRIG



• – ○ • – ○ • – ○ •

Introduction

The function block **AX_FB_R_TRIG** (Rising Trigger with AX Adapter) is used to detect the rising edge of a Boolean signal. Unlike the standard `R_TRIG` block according to IEC 61131-3, this block uses **AX adapters** for input and output. It encapsulates the edge detection logic in an adapter-based architecture, which simplifies integration into systems that use `adapter::types::unidirectional::AX`. The goal is to generate a `TRUE` signal at the output when the input signal transitions from `FALSE` to `TRUE`.



Interface Structure

Event Inputs

This block has no direct event inputs. Event control is handled via the adapter `CLK`.

Event Outputs

This block has no direct event outputs. Event forwarding is handled via the adapter `Q`.

Data Inputs

This block has no direct data inputs. Data is received via the adapter `CLK`.

Data Outputs

This block has no direct data outputs. Data is output via the adapter `Q`.

Adapters

Name	Type	Comment
CLK	`adapter::types::unidirectional::AX`	**Socket (Input):** The clock input being monitored. Contains the data signal (D1) and the associated event (E1).
Q	`adapter::types::unidirectional::AX`	**Plug (Output):** The pulse output. Provides `TRUE` only on a rising edge at input `CLK`.

Functionality

The component implements the classic logic of a "Rising Edge Detector":

1. The component waits for an event at the adapter input `CLK` (signal `CLK.E1`).
2. As soon as the event occurs, the algorithm `REQ` is executed.
3. The algorithm checks the current value of the data input (`CLK.D1`) and compares it to the stored state from the previous cycle (`MEM`).

- **Logic:** `Q.D1 := CLK.D1 AND NOT MEM`

1. If `CLK.D1` is true (`TRUE`) and the stored value `MEM` was false (`FALSE`), the output `Q.D1` is set to `TRUE`. In all other cases, `Q.D1` is `FALSE`.
2. The internal memory `MEM` is then updated with the current value of `CLK.D1`.
3. Finally, the event is triggered at the output adapter `Q` (`Q.E1`) to inform subsequent function blocks.

Technical Features

- **Adapter Encapsulation:** The function block is specifically designed for the type `adapter::types::unidirectional::AX`. This adapter type typically bundles an event (`E1`) and a Boolean data point (`D1`). This reduces the visible wiring in the application diagram, as event and data flow are combined in a single connection line.
- **Internal Memory:** The block uses an internal Boolean variable `MEM` to store the state of the input signal between executions.

State Overview

The block is based on a very simple Execution Control Chart (ECC) with only one state:

- **State `REQ`:** This is the only state.
- **Transition:** The block remains in this state and responds to the input adapter event `CLK.E1`.
- **Action:** On each event, the algorithm `REQ` is executed, and the output adapter event `Q.E1` is sent.

Application Scenarios

- **Signal Conditioning:** Converting a continuous signal (e.g., switch is ON) into a single pulse to trigger an action exactly once when the device is switched on.
- **Wiring Simplification:** Used in systems that consistently rely on adapter technology to minimize the number of visible connections between functional blocks.
- **Start Trigger:** Detects when a process or machine is started (transition from 0 to 1).

⚖️ Comparison with Similar Components

- **`R_TRIG` (Standard):** The functional core is identical. The difference lies in the interface: `R_TRIG` uses separate `CLK` (BOOL) and `REQ` (Event) inputs, as well as `Q` (BOOL) and `CNF` (Event) outputs. `AX_FB_R_TRIG` uses adapters.
- **`AX_FB_F_TRIG`:** The counterpart to this function block, which reacts to falling edges (high-to-low).

Conclusion

The **AX_FB_R_TRIG** is an essential utility function block for developers working with AX adapters within the 4diac environment. It offers the robust functionality of standard edge detection, packaged in a modern, adapter-based interface, resulting in cleaner and more modular control applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

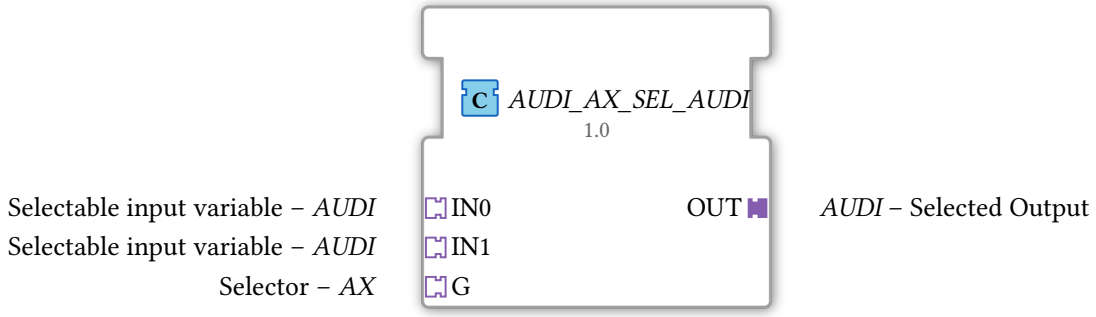
Numeric Index (iec61131-3)

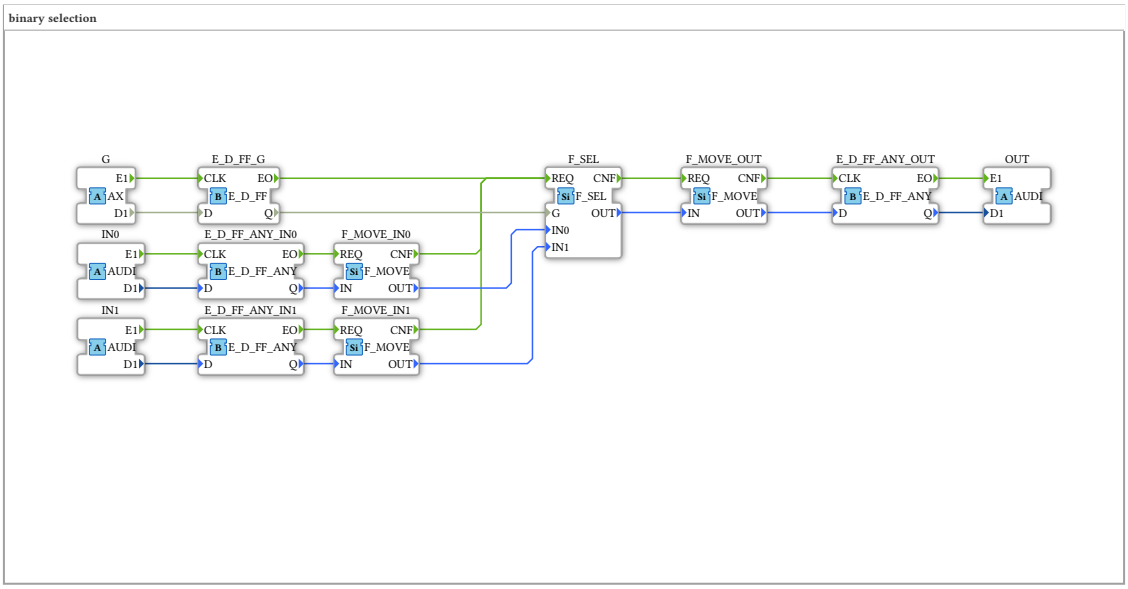
Here you will find all exercises from the `iec61131-3` workspace, listed in ascending order. ...

In this category, you will find function blocks for signal selection and extremum computation using adapter interfaces:

- **AR_AX_SEL_AR**: Binary selection between two AR inputs based on an AX selector.
- **AR_MAX**: Computes the maximum value from two AR inputs.
- **AR_MIN**: Computes the minimum value from two AR inputs.

AUDI_AX_SEL_AUDI





• – ○ • – ○ • – ○ •

Introduction

The function block **AUDI_AX_SEL_AUDI** implements a binary selection between two input values. It acts as a multiplexer, switching one of the two inputs to the output depending on a selector signal. Communication occurs exclusively via unidirectional adapters that encapsulate both event and data signals. This allows for flexible and type-specific integration into existing signal paths.

Interface Structure

Event Inputs

The function block does not have dedicated event inputs. However, events are provided via the **Selector Adapter G**:

- **G.E1**: Event signal that triggers the selection logic.

Event Outputs

The function block does not have dedicated event outputs. Events are output via the **Output Adapter OUT**:

- **OUT.E1**: Acknowledge signal triggered after successful completion of the selected input.

Data Inputs

The FB does not have dedicated data inputs. Data values are provided via the following adapters:

- **IN0.D1** (Type: AUDI) – first selectable data value
- **IN1.D1** (Type: AUDI) – second selectable data value
- **G.D1** (Type: AX) – selector field for determining the active source

Data Outputs

The FB does not have dedicated data outputs. The selected data value is output via the **Output Adapter OUT**:

- **OUT.D1** (Type: AUDI) – selected data value (corresponds to IN0.D1 or IN1.D1)

Adapters

Adapter	Direction	Type	Comment
IN0	Socket (Input)	`adapter::types::unidirectional::AUDI`	First selectable input
IN1	Socket (Input)	`adapter::types::unidirectional::AUDI`	Second selectable input
G	Socket (Input)	`adapter::types::unidirectional::AX`	Selector (Event + Data)
OUT	Plug (Output)	`adapter::types::unidirectional::AUDI`	Output field with event and data

Functionality

The function block internally contains an instance of the IEC 61131 function block `F_SEL` (from the library `iec61131::selection`). The logic proceeds as follows:

1. **Triggering**: An event at **G.E1** (via the selector adapter) starts the processing.
2. **Data Collection**: The values received from **IN0.D1**, **IN1.D1**, and **G.D1** are forwarded to the `F_SEL` function block.
3. **Selection**: `F_SEL` checks the value of **G.D1**. Typical Semantics (depending on the data type `AX`):

- If the selector field is **False** (or 0), **IN0** is selected.
- If the selector field is **True** (or not equal to 0), **IN1** is selected.

1. **Output**: The selected signal is passed on to **OUT.D1**. Simultaneously, an event is sent to **OUT.E1** to inform downstream processing.

The internal flow is event-driven and avoids continuous read accesses, enabling efficient communication in time-controlled environments.

Technical Features

- **Pure Adapter Communication**: All interfaces are defined as unidirectional adapters. Event and data paths are combined within a single adapter, simplifying configuration in the 4diac IDE.
- **Reuse of IEC 61131 libraries**: The selection logic is fully implemented by the standard function block `F_SEL`, which requires no additional state machines or complex algorithms.
- **Type safety**: The adapters are specialized for the user-defined types `AUDI` and `AX`. This prevents miscoupling with incompatible signals at design time.
- **No state of its own**: The function block itself is stateless – it simply forwards the signals to the internal `F_SEL`. The internal function block manages the selection logic autonomously.

State overview

The function block **AUDI_AX_SEL_AUDI** does **not have its own state machine**. The entire selection logic is executed by the inner `F_SEL` block, which itself is a pure function block without state memory. Therefore, its behavior is **stateless**:

- After receiving an event at **G.E1**, the selection is made immediately and the result is output to **OUT.D1**.
- There are no initialization, wait, or error states; the block is always ready for a new selector request.

Application Scenarios

This block is ideally suited for the following applications in automation technology:

- **Signal switching** between two sensors (e.g., temperature sensors, pressure transducers) depending on an operating mode or a manual switch.
- **Bypass selection** in control paths: if one channel malfunctions, the system automatically switches to a backup channel.
- **Parameter selection**: a selector field is used to choose between two fixed parameter sets (e.g., for different products).
- **Test Mode**: Switching between a real and a simulated signal for commissioning or maintenance.

The use of user-defined adapter types (`AUDI`, `AX`) allows for easy adaptation to project-specific data structures.

Comparison with Similar Function Blocks

Criterion	**AUDI_AX_SEL_AUDI**	Simple Boolean MUX (e.g., `F_SEL`)
Interface	Pure adapters (event and data encapsulated)	Separate input/output pins
Data Type	User-defined types (`AUDI`, `AX`)	Arbitrary (configurable)
Selector	Event-driven (via adapter)	Data-driven, without extra events
Reusability	Higher due to type safety and encapsulation	More flexible, but loosely coupled
Complexity	Medium (adapters require prior knowledge)	Low (direct pins)

The **AUDI_AX_SEL_AUDI** offers higher integration into adapter-based architectures and is particularly suitable for modular, type-bound systems, while a pure `F_SEL` is more universal.

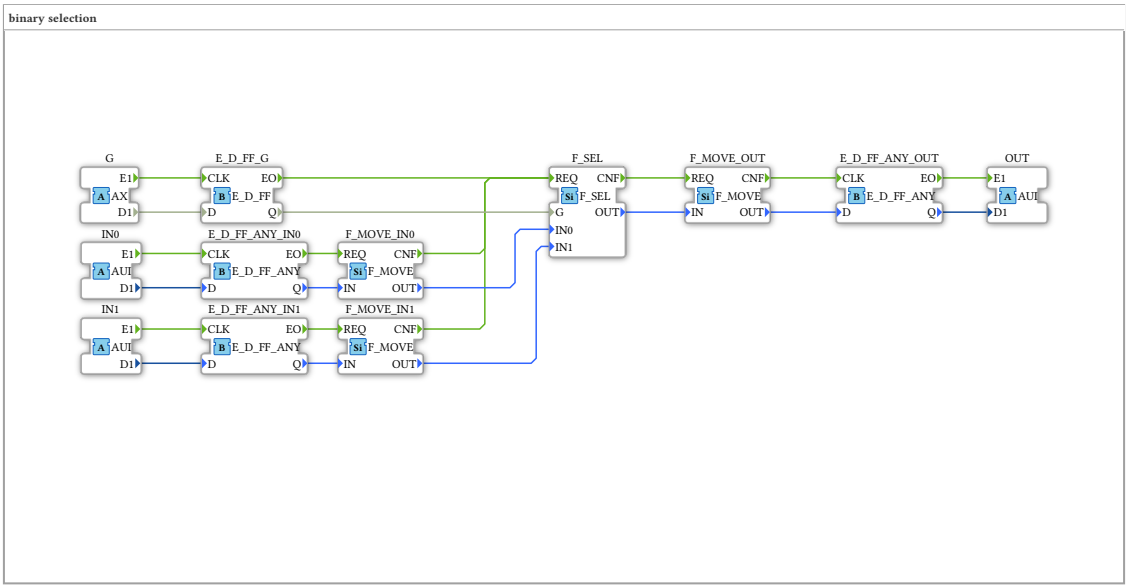
Conclusion

The **AUDI_AX_SEL_AUDI** is a specialized binary selector that leverages the advantages of adapter-based communication in 4diac. By encapsulating event and data paths in a single adapter, signal wiring in the overall system is reduced and type safety is increased. Internal delegation to the proven `F_SEL` block from the IEC 61131 library ensures robust, stateless selection logic. This block is ideally suited for use in modular control projects where clear interfaces and easy reusability are paramount.

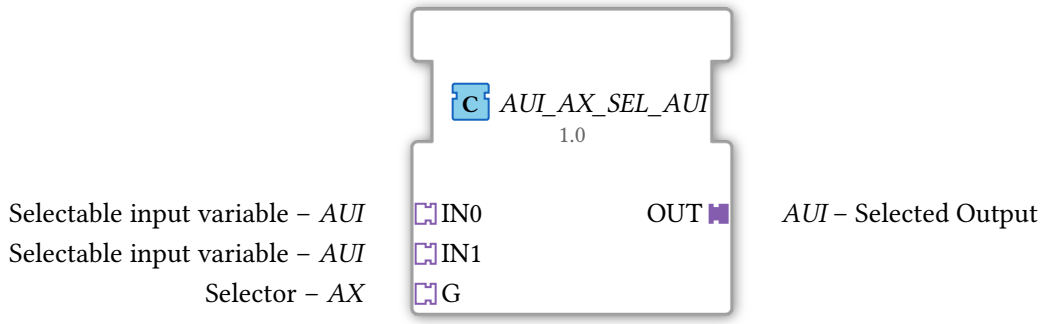
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AUI_AX_SEL_AUI



Binary selection



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AUI_AX_SEL_AUI** implements a binary selection between two input values of type **AUI**. The decision as to which input is passed to the output is made via a third adapter of type **AX**, which serves as a selector. The block encapsulates the function **F_SEL** defined in IEC 61131-3 and provides it as an adapter-based component.

Interface Structure

Event Inputs

- **G.E1** (via adapter G): Triggers the selection processing as soon as an event is present at the selector adapter.

Event Outputs

- **OUT.E1** (via adapter OUT): Is triggered after the selected value is present at the output.

Data Inputs

- **G.D1** (via adapter G): The selecting value (e.g., BOOL or numeric) that determines whether **IN0** or **IN1** is selected.
- **IN0.D1** (via adapter IN0): First input value of type **AUI**.
- **IN1.D1** (via adapter IN1): Second input value of type **AUI**.

Data Outputs

- **OUT.D1** (via adapter OUT): The selected value (type **AUI**), according to the selection logic.

Adapters

- **G** (socket, type `adapter::types::unidirectional::AX`): Provides the selection signal.
- **IN0** (socket, type `adapter::types::unidirectional::AUI`): First data channel.
- **IN1** (Socket, type `adapter::types::unidirectional::AUI`): Second data channel.
- **OUT** (Plug, type `adapter::types::unidirectional::AUI`): Result channel.

Functionality

The function block operates event-driven via the adapter interfaces. As soon as an event arrives at selector **G.E1**, the internal function block **F_SEL** is activated. **F_SEL** checks the value **G.D1**:

- If **G.D1** = `FALSE` (or 0), the value of **IN0.D1** is passed to output **OUT.D1**.
- If **G.D1** = `TRUE` (or not equal to 0), **IN1.D1** is selected.

After the selection is complete, **F_SEL.CNF** signals the event **OUT.E1** to inform the consumer of the new valid output value.

Technical Features

- The function block is implemented as a **Composite FB** – its behavior results from the internal circuitry of a standardized **F_SEL** block (IEC 61131-3).
- Data is transmitted via adapters of type **AUI** and **AX**. These adapter types are defined as unidirectional channels whose internal data structure is not specified in detail (e.g., integer, real, or user-defined structs).
- Selection is purely combinatorial, but only after a triggering event at the selector. A simple data change without an event does not result in an output update.

State Overview

Since this is a composite FB without its own state machine, there are no explicit states. The logic is purely event-driven:

1. **Wait for event** – as long as no **G.E1** is received, the output remains unchanged.
2. **Perform selection** – after **G.E1**, the corresponding input is selected and the output is set.
3. **Signal output** – **OUT.E1** is generated after successful selection.

Application Scenarios

- **Switching between two measured values** (e.g., temperature from sensor A or B) based on a control signal.
- **Operating mode switching** in automation technology, where different control variables are activated depending on the mode.
- **Redundant signal sources** – if the primary channel (IN0) fails, the selector can switch to the secondary channel (IN1).

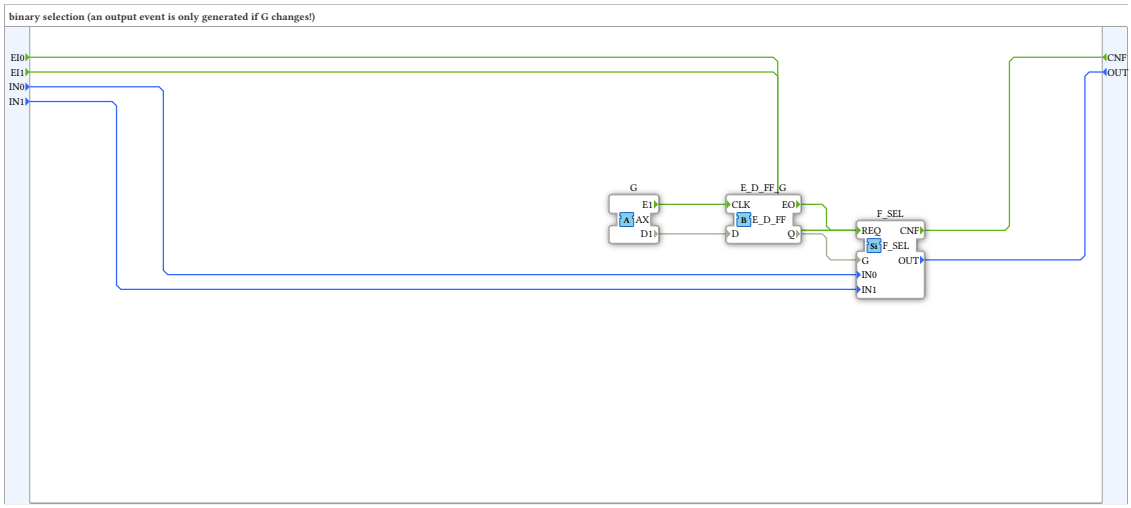
Comparison with similar components

- **MUX (Multiplexer)**: A multiplexer selects from more than two inputs; **AUI_AX_SEL_AUI** is specialized for two inputs and uses adapters instead of individual signals.
- **F_SEL (Standard IEC 61131)**: The internal function block offers identical functionality, but without adapter encapsulation. This function block extends it with an adapter-based interface, which facilitates reuse in adapter-oriented architectures.
- **SEL (Selection)**: Similar logic, but usually defined as a pure function block without event control. Here, the selection is explicitly triggered by an event.

Conclusion

AUI_AX_SEL_AUI is an adapter-based composite function block for simple binary selection between two AUI data streams. The use of `F_SEL` internally ensures robust and standards-compliant selection logic. The function block is particularly suitable for modular automation solutions where data is exchanged via unidirectional adapters.

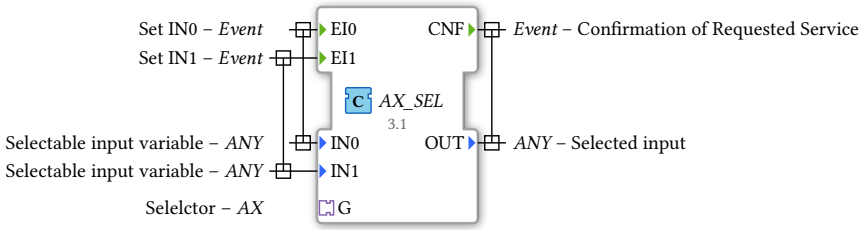
AX_SEL



• – ○ • – ○ • – ○ •

Introduction

The AX_SEL function block is a binary selector used to choose between two input values. It belongs to the category of standard selection functions according to IEC 61131-3 and enables flexible data selection based on a control signal.



Interface Structure

Event Inputs

- **EI0** - Set IN0 (data association: `IN0`)
- **EI1** - Set IN1 (data association: `IN1`)

Event Outputs

- **CNF** - Confirmation of Requested Service

Data Inputs

- **IN0** (ANY) - Selectable input variable
- **IN1** (ANY) - Selectable input variable

Data Outputs

- **OUT** (ANY) - Selected input

Adapters

- **G** (adapter::types::unidirectional::AX) - Unidirectional adapter for control

Functionality

The AX_SEL function block selects between two input values (IN0 and IN1) and outputs the selected value at output OUT. The selection is controlled via the connected adapter G: `G.E1` clocks an internal D flip-flop (`E_D_FF_G`) that takes over `G.D1` and supplies it as selector `G` to `F_SEL`. `F_SEL` is triggered by the flip-flop event (G has changed) and directly by `EI0` and `EI1`; its `CNF` is output as the block's `CNF` and the selected

value is then available at OUT. An output event is therefore generated only on an event at `G.E1`, `EI0` or `EI1`. ...

Technical Features

- Uses the ANY data type for maximum flexibility in inputs and outputs
- Implemented as a unidirectional adapter for easy control
- Supports various data types through the use of ANY

State Overview

The function block has a simple state:

- **Ready**: Waits for a control signal from adapter G
- Upon receiving an event (`EI0`, `EI1` or `G.E1`), the corresponding input is selected and the CNF event is triggered

Application Scenarios

- Selection between two different sensors
- Switching between operating modes
- Selection of data sources depending on operating states
- Switching between manual and automatic control

⚖️ Comparison with Similar Function Blocks

Compared to other selection blocks, AX_SEL offers:

- Greater flexibility through the ANY data type
- Easy integration through the unidirectional adapter
- Standardized interface according to IEC 61131-3

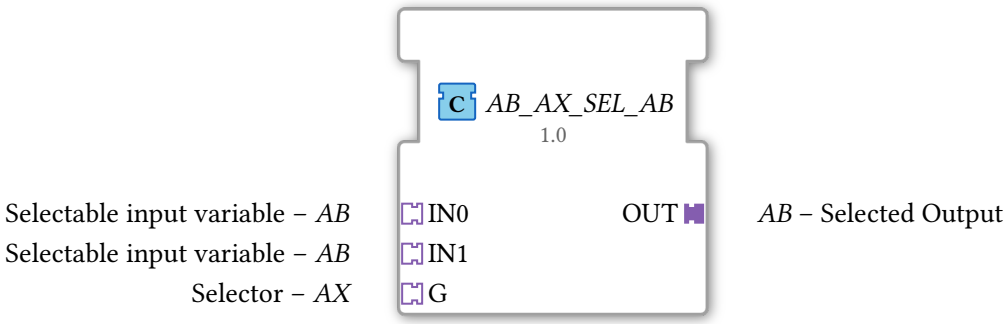
Comparison with F_SEL

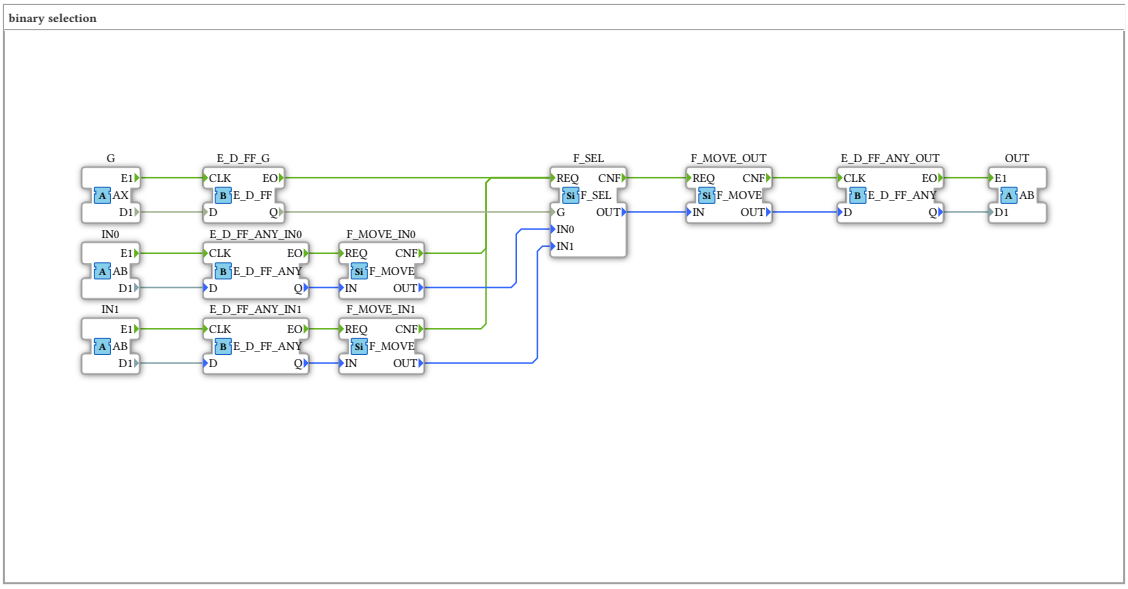
Conclusion

The AX_SEL function block provides a simple and flexible solution for binary selection tasks in control systems. By using the ANY data type and standardized adapter interfaces, it enables versatile application in various automation projects.

Adapter

AB_AX_SEL_AB





• – ○ • – ○ • – ○ •

The function block `AB_AX_SEL_AB` is a composite function block for IEC 61499 that implements binary selection at the adapter level. Based on a control signal from a selector adapter (`G`), it selects between the signals of two input adapters (`IN0` and `IN1`) and forwards the selected signal to an output adapter (`OUT`).

This block is particularly suitable for event-driven architectures where data streams need to be routed flexibly without disrupting the encapsulation of the adapter interfaces.

Since this is an adapter-based function block, the classic event and data inputs are completely encapsulated within the adapter interfaces.

No direct event inputs are available. Control is achieved via events within the adapters.
No direct event outputs are available. Signaling is achieved via events within the output adapter.
No direct data inputs are available.
No direct data outputs are available.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

Name	Direction	Type	Description
OUT	Plug (Consumer)	`adapter::types::unidirectional::AB`	The selected output. Outputs the forwarded signal.
IN0	Socket (Provider)	`adapter::types::unidirectional::AB`	First selectable input (selected when selector `G` is in state `FALSE`).
IN1	Socket (Provider)	`adapter::types::unidirectional::AB`	Second selectable input (selected when selector `G` is in state `TRUE`).
G	Socket (Provider)	`adapter::types::unidirectional::AX`	Selector adapter (control signal for selection).

Note on adapter types:

- AB represents a unidirectional adapter that carries a byte data signal (D1/D) and an event (E1).
- AX represents a unidirectional adapter that carries a Boolean data signal (D1/D) and an event (E1).

Functionality

Inside the function block is a network of standard function blocks (IEC 61131-3 and IEC 61499) that implements the logic:

1. **Event and Data Acquisition:**

- When an event (IN0, IN1, G) arrives at one of the input adapters (IN0, IN1, G), the corresponding value (D1) is temporarily stored via an edge-triggered memory element (D flip-flop E_D_FF or E_D_FF_ANY).

1. **Data Conversion / Buffering:**

- The function blocks F_MOVE_IN0 and F_MOVE_IN1 copy the byte values of the inputs to the selection element.

1. **Selection Logic (F_SEL):**

- The standard selection function block F_SEL (from the IEC 61131-3 Library) performs the actual selection:
- If the signal from the selector adapter G equals FALSE, the signal from IN0 is passed through to the output.
- If the signal from selector adapter G equals TRUE, the signal from IN1 is passed through to the output.

1. **Output:**

- The selected value is transferred via another conversion block (F_MOVE_OUT with data type BYTE) to an output flip-flop (E_D_FF_ANY_OUT).
- This flip-flop generates the corresponding output event (OUT.E1) at plug OUT and provides the selected data value (OUT.D1).
- **Event-driven data routing:** Any change to the inputs IN0 or IN1, as well as to the selector G, triggers an update and calculation of the output. No cyclic polling occurs.
- **Decoupling via memory elements:** The use of E_D_FF and E_D_FF_ANY ensures that the data is consistently transferred and that no uncontrolled signal changes (race conditions) occur at the output.
- **Decoupling via memory elements:** The use of E_D_FF and E_D_FF_ANY ensures that the data is transferred consistently and that no uncontrolled signal changes (race conditions) occur at the output.
- – **Typing:** The internal processing of the data is fixed to the data type BYTE (defined via the attribute DataType="BYTE" on the F_MOVE blocks).

State Selector G	Event at Input	Behavior at Output OUT
FALSE	Event at IN0	Value from IN0 is passed to OUT; Event OUT.E1 is triggered.
FALSE	Event sent to IN1	The value of IN1 is updated internally, but **not** passed to OUT.
TRUE	Event sent to IN0	The value of IN0 is updated internally, but **not** passed to OUT.
TRUE	Event sent to IN1	The value of IN1 is passed to OUT; Event OUT.E1 is triggered.
Switch FALSE → TRUE	Event on G	The currently stored value of IN1 is passed to OUT; OUT.E1 is triggered.
Switch TRUE → FALSE	Event on G	The currently stored value of IN0 is passed to OUT; OUT.E1 is triggered.

- **Signal Switching at the Field Level:** Switching between the signal from an active sensor (IN1) and a predefined substitute/error value (IN0) in case of a fault.

- **Manual/Automatic Switching:** Forwarding of control bytes, whereby values from the automatic control are used in automatic mode (`G = TRUE`), while manually specified values are sent to the actuators in manual mode (`G = FALSE`).
- **Recipe or Mode Selection:** Dynamic path selection for configuration or status data in complex process controls.

Comparison with Similar Function Blocks

Compared to the standard function block `F_SEL`, `AB_AX_SEL_AB` operates directly at the adapter level.

- While `F_SEL` requires manual wiring of discrete data lines and separate trigger events, `AB_AX_SEL_AB` encapsulates this logic.
- This saves engineering time, reduces the number of visible connections in the 4diac IDE Application Editor, and minimizes potential errors when linking data and event streams.

The `AB_AX_SEL_ABis` a robust and reusable auxiliary building block for signal routing based on byte adapters. Through the consistent encapsulation of the selection and synchronization logic, it significantly contributes to the clarity and modularization of control programs in the 4diac IDE.

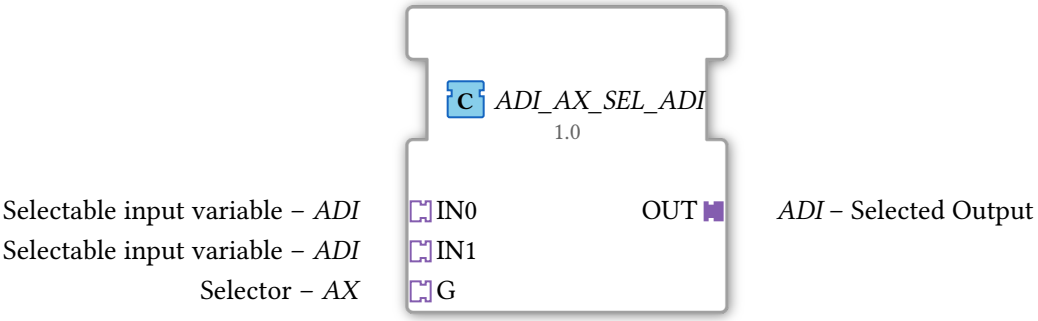
Technical Features

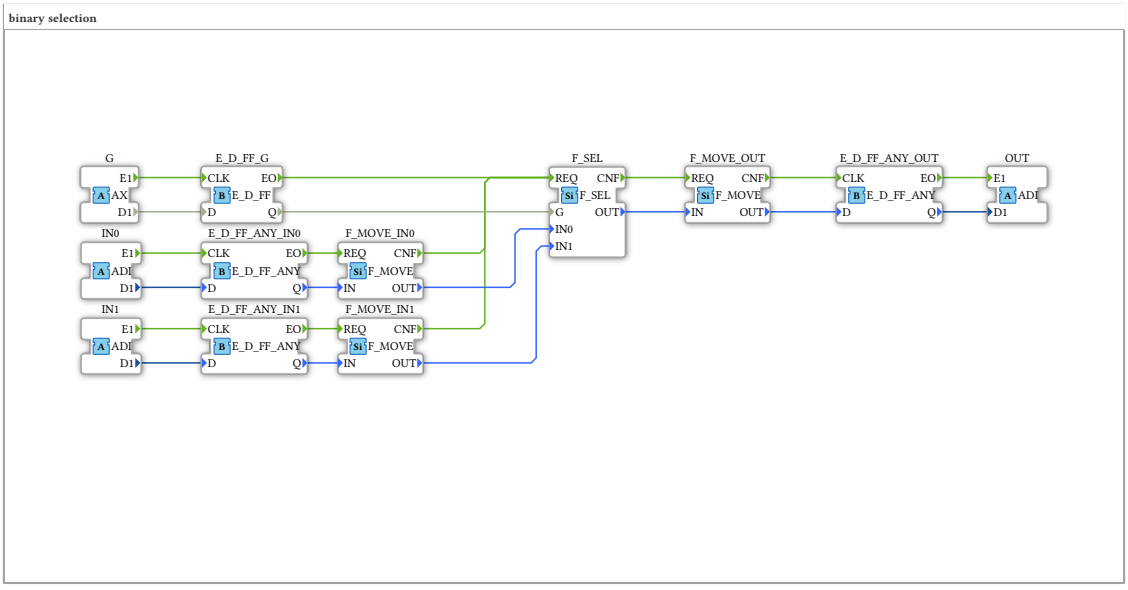
State Overview

Application Scenarios

Conclusion

ADI_AX_SEL_ADI





No image available

• – ○ • – ○ • – ○ •

Introduction

The function block `ADI_AX_SEL_ADI` is used for binary selection between two analog or integer input signals transmitted via unidirectional adapters. Based on the state of a digital selector signal, either the value of the first or the second input is passed through to the output.

The block is implemented as a composite function block (FB) and uses internally standardized selection and copy functions according to IEC 61131-3, which have been adapted for the event-driven IEC 61499 adapter architecture.

Interface Structure

The interface of this function block is implemented entirely via adapters. There are no direct elementary data or event inputs outside of the adapter structures.

Event Inputs

No direct event inputs available (events are received via the adapter interfaces).

Event Outputs

No direct event outputs available (events are sent via the adapter interfaces).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Plugs (Connectors - Outputs)

- **OUT** (Type: `adapter::types::unidirectional::ADI`):

The selected output. Sends the selected data signal (`D1`) and the associated update event (`E1`).

Sockets (Inputs)

- **IN0** (Type: `adapter::types::unidirectional::ADI`):

First selectable input. This value is passed through to the output when the selection signal `G` is set to `FALSE` (0).

- **IN1** (Type: `adapter::types::unidirectional::ADI`):

Second selectable input. This value is passed through to the output when the selection signal `G` is set to `TRUE` (1).

- **G** (Type: `adapter::types::unidirectional::AX`):

The selector (gate) for controlling the selection process. A change in state controls the switching between `IN0` and `IN1`.

Functionality

The internal structure of `ADI_AX_SEL_ADI` operates purely event-driven:

1. **Input Acquisition**:

When an event (`E1`) arrives at one of the inputs (`IN0`, `IN1`) or at the selector (`G`), the corresponding internal data value (`D1`) is acquired via event-driven memory elements (`E_D_FF_ANY` or `E_D_FF`).

1. **Data Buffering**:

The function blocks `F_MOVE_IN0` and `F_MOVE_IN1` copy the data (interpreted as data type `DINT`) to the central selection function block `F_SEL`.

1. **Selection Logic**:

The function block `F_SEL` (corresponding to the classic `SEL` standard function block from IEC 61131-3) evaluates the control signal `G`:

- If the signal from adapter `G` is equal to `FALSE`, the signal from `IN0` is selected.

- If the signal from adapter G is equal to TRUE, the signal from IN1 is selected.

1. **Output**:

The selected value is passed via F_MOVE_OUT to the output flip-flop E_D_FF_ANY_OUT. This flip-flop sets the value D1 on output adapter OUT and immediately triggers the corresponding event OUT.E1 to inform downstream components about the data change.

Technical Features

- **Adapter Encapsulation**: By using adapters (ADI and AX), the number of visible connection lines in the 4diac application diagram is drastically reduced, improving clarity and maintainability.
- **Data Type Specification**: The internal copy operations (F_MOVE_IN0, F_MOVE_IN1, and F_MOVE_OUT) are permanently configured for the data type DINT (Double Integer).
- **Real-Time Pass-Through**: The function block responds instantly to every edge of the input signals as well as the selection signal.

-

State Overview

Since this is a purely data-flow and event-driven network function block (Composite FB), it does not have an internal state machine (ECC). Its behavior can be described statically as follows:

State Selector G	Event at Input	Behavior at Output OUT
FALSE	Any Event	OUT.D1 receives the value of IN0.D1; OUT.E1 is triggered.
TRUE	Any Event	OUT.D1 receives the value from IN1.D1; OUT.E1 is triggered.

–

Application Scenarios

- **Manual/Automatic Switching**: Selection between a manually specified setpoint (e.g., via an HMI at IN1) and an automatically calculated controller setpoint (at IN0).
- **Sensor Redundancy**: Switching to a backup sensor (IN1) if diagnostic logic detects a fault in the main sensor (IN0) and sets the signal G to TRUE.
- **Recipe Control**: Dynamic selection of various predefined process parameters during operation.

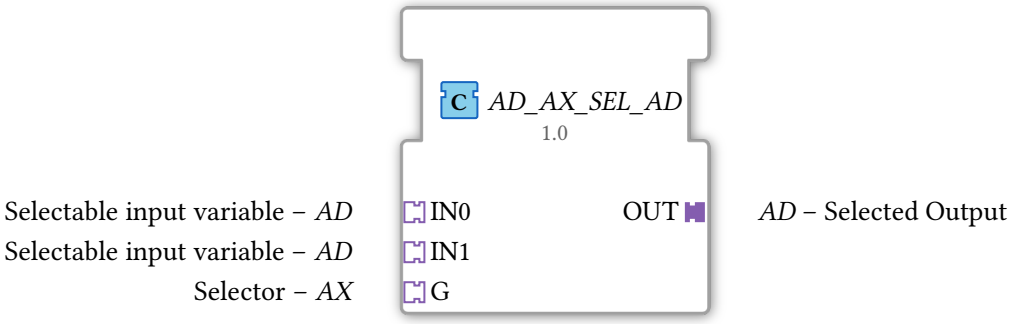
Comparison with Similar Function Blocks

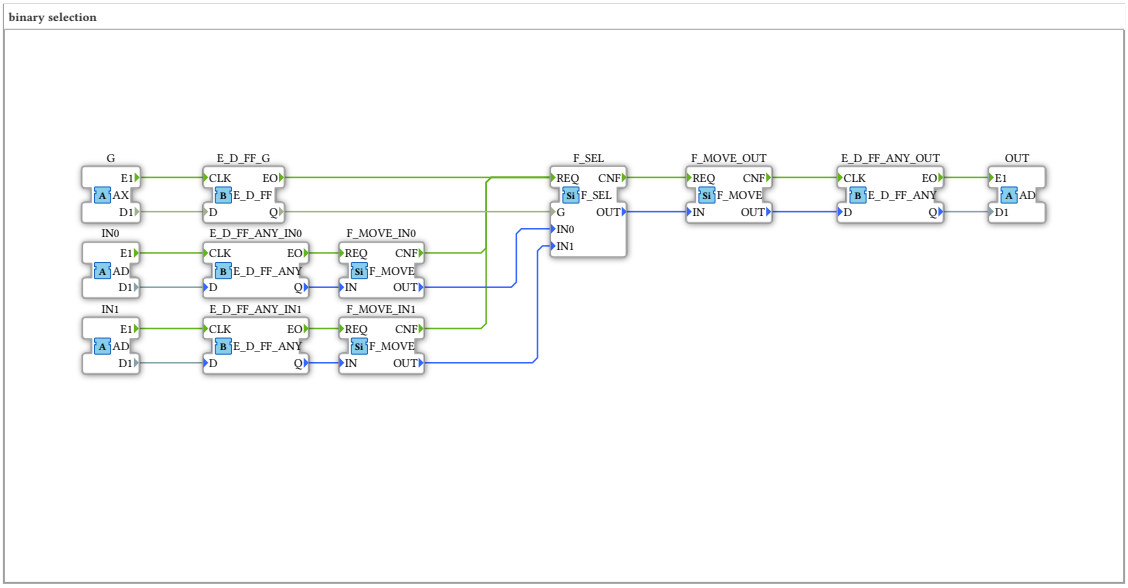
- **Standard F_SEL**: The standard selection function block works with elementary data types and requires explicit event connections for triggers and confirmations. ADI_AX_SEL_ADI fully automates this process via structured adapter channels.
- **Analog Multiplexers**: Unlike multiplexers with an indefinite number of channels, this function block specializes in fast and efficient binary selection (1 out of 2).

Conclusion

The ADI_AX_SEL_ADI is an extremely useful auxiliary function block for IEC 61499 applications in the 4diac IDE. It combines the proven selection logic of IEC 61131-3 with the modern, clean design principles of adapter-based event-oriented communication.

AD_AX_SEL_AD





• – ○ • – ○ • – ○ •

The function block `AD_AX_SEL_AD` performs binary selection between two input signals provided via adapter structures. Based on the state of a selection signal (G), either the value of input `IN0` or input `IN1` is passed to output `OUT`. Internally, the block is implemented as a composite function block and utilizes standardized IEC 61499 and IEC 61131 mechanisms to ensure event-driven, type-safe signal propagation.

The interfaces of this function block are implemented entirely via adapters, enabling modular and clear wiring within the application network.

This function block does not have direct event inputs on the housing. Event control is implicit via the adapters.

This function block does not have direct event outputs on the housing. Event forwarding is implicit via the adapters.

This component has no direct data inputs on its housing.

This component has no direct data outputs on its housing.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- **Sockets (Input Adapters):**
- `IN0` (Type: `adapter::types::unidirectional::AD`): First selectable input channel (selected when `G` is in the state `FALSE`).
- `IN1` (Type: `adapter::types::unidirectional::AD`): Second selectable input channel (selected when `G` is in state `TRUE`).
- `G` (Type: `adapter::types::unidirectional::AX`): Selector signal for controlling the selection.
- **Plugs (Output Adapters):**
- `OUT` (Type: `adapter::types::unidirectional::AD`): The selected output channel that carries the value of the selected input.

Functionality

The module encapsulates an internal network of control and conversion modules:

1. **Signal Buffering and Acquisition:** As soon as data changes on the adapters `IN0`, `IN1`, or `G`, the corresponding adapter event `E1` triggers the internal flip-flops (`E_D_FF` or `E_D_FF_ANY`).
2. **Data Forwarding:** The data values (`D1`) of inputs `IN0` and `IN1` are read via internal data converters (`F_MOVE` of type `DWORD`) and made available to the selection block `F_SEL`.
3. **Selection Logic (`F_SEL`):**

- If the signal `FALSE` is present at selector `G`, the block switches the signal from `IN0` to the output.
- – If the signal `TRUE` is present at selector `G`, the function block switches the signal from `IN1` to the output.

1. **Output Synchronization:** The selected value is passed via another function block `F_MOVE` to an output flip-flop (`E_D_FF_ANY_OUT`). This flip-flop generates the output event `E1` at output adapter `OUT` and places the selected value at `D1`.

- **Adapter Encapsulation:** The use of unidirectional adapters (`AD` and `AX`) simplifies signal routing and avoids loose data and event lines.
- **Type Independence via DWORD:** The internal data paths use the `DWORD` data type to represent the adapter data, enabling flexible transmission of different data formats.
- **Near-Real-Time Response:** Every edge or value change at the input or selector adapters immediately results in an output update.

As a composite function block, this module does not have its own Execution Control Chart (ECC). The behavior is purely data- and event-driven through the internal interaction of the standard function blocks (FBs):

- **Event on `IN0` or `IN1`:** Updates the respective internal value buffer. If the affected input is currently active, the new value is immediately output to `OUT`.
- **Event on `G`:** Switches the active channel and outputs the value of the newly selected channel to `OUT`.
- **Sensor Redundancy:** Switches between a primary sensor (`IN0`) and a redundant backup sensor (`IN1`) in case of signal loss or malfunction.
- **Setpoint Switching:** Switching between manual and automatic setpoint operation in industrial processes.
- **Signal Multiplexing:** Structured forwarding of control data in modular systems.
- **Standard `F_SEL`:** The classic `F_SEL` function block according to IEC 61131-3 operates directly on elementary data types and requires manual wiring of trigger events. `AD_AX_SEL_AD` encapsulates this logic and makes it directly available for adapter-based architectures.

- **Multiplexer (MUX):** A classic MUX allows selection from more than two channels using an integer value. `AD_AX_SEL_AD` is optimized for simple binary decisions (either/or) and thus saves processing resources.

The `AD_AX_SEL_AD` is a robust and efficient auxiliary module for event-driven signal switching. Thanks to the consistent use of adapters, it integrates perfectly into modern, service-oriented control architectures within 4diac.

Technical Features

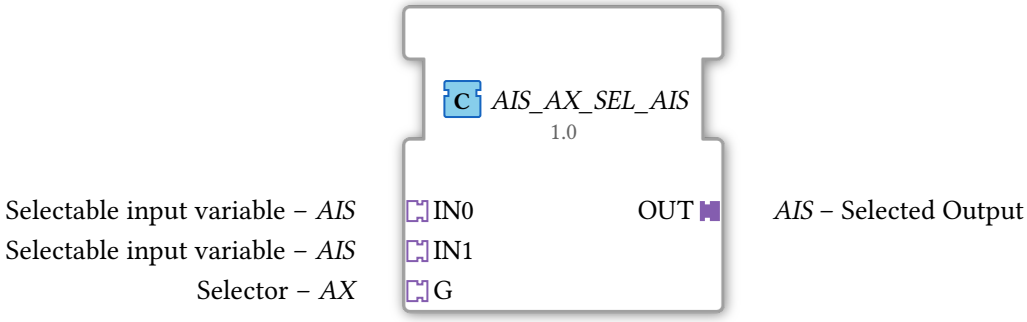
State Overview

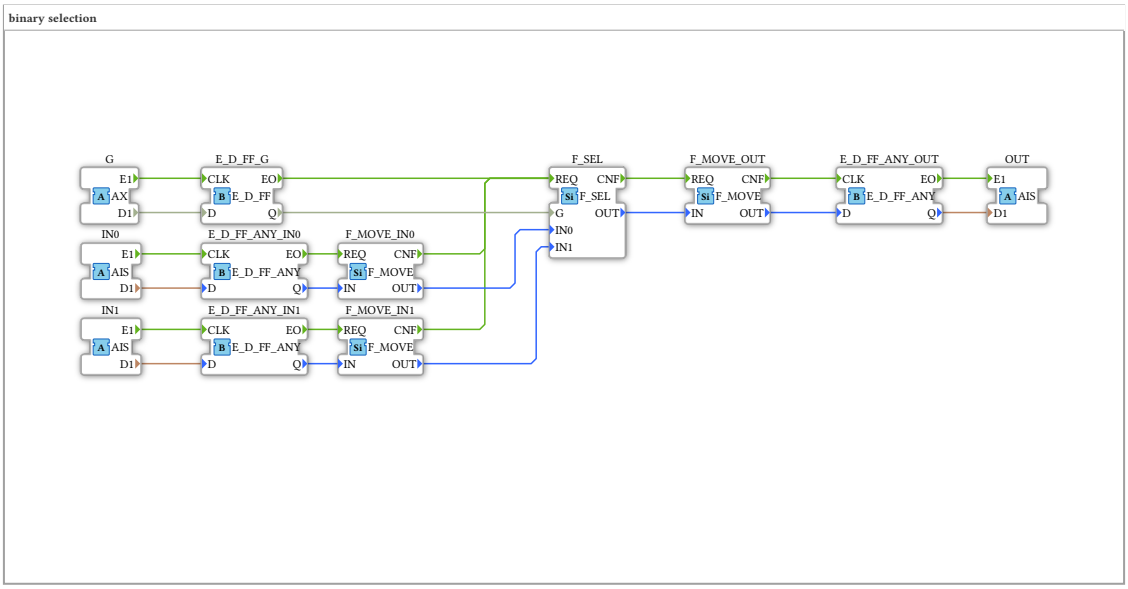
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AIS_AX_SEL_AIS





• – ○ • – ○ • – ○ •

Introduction

The function block **AIS_AX_SEL_AIS** is a binary selection block for the IEC 61499 architecture in the 4diac IDE. It dynamically selects between two unidirectional analog input signals based on the state of a selection signal (gate) and forwards the selected signal to the output.

Internally, the block uses the classic selection logic of the standardized `SEL` block (known from IEC 61131-3), but is fully optimized for event-driven execution according to IEC 61499 and encapsulates the data and event streams using standardized adapters.

Interface Structure

The function block uses an adapter-based interface design. It has no directly accessible event- or data-based inputs and outputs at the main level, but handles all communication via plugs and sockets.

Event Inputs

No direct event inputs are available. Event processing is handled encapsulated via the adapter interfaces.

Event Outputs

No direct event outputs are available. Event output is handled encapsulated via the output adapter.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Sockets (Input Interfaces)

- **G** (Type: `adapter::types::unidirectional::AX`): The selector input (gate). Determines which of the two inputs (`IN0` or `IN1`) is routed to the output.
- **IN0** (Type: `adapter::types::unidirectional::AIS`):

The first selectable input channel. This channel is activated when the selector `G` is in the state `FALSE` (0).

- **IN1** (Type: `adapter::types::unidirectional::AIS`):

The second selectable input channel. This channel is activated when the selector `G` is in the state `TRUE` (1).

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::AIS`):

The selected output channel. It provides the value of the currently active input channel and the corresponding update event.

Functionality

Inside the FB, a network of standard function blocks synchronizes the adapter events and makes the logical selection:

1. **Event Synchronization & Signal Storage:**

Data arriving at the adapters `IN0`, `IN1`, and `G` is temporarily stored in data flip-flops (`E_D_FF_ANY` and `E_D_FF`, respectively) each time a corresponding adapter event (`E1`) arrives. This ensures that the data values are stably available for further processing.

The data arriving at the adapters `IN0`, `IN1`, and `G` are temporarily stored in data flip-flops (`E_D_FF_ANY` and `E_D_FF`) whenever a corresponding adapter event (`E1`) arrives. This ensures that the data values are stably available for further processing. 2. **Data Provisioning:**

The function blocks `F_MOVE_IN0` and `F_MOVE_IN1` copy the cached data (interpreted as data type `STRING`) and forward it to the actual selection core.

1. **Selection Process (F_SEL):**

The core function block `F_SEL` (type `iec61131::selection::F_SEL`) evaluates the state of the selector `G`:

- If `G` = `FALSE`, the value of `IN0` is selected.
- If `G` = `TRUE`, the value of `IN1` is selected.

1. **Output Transfer:**

The selection result is transferred via the function block `F_MOVE_OUT` to the output flip-flop `E_D_FF_ANY_OUT`. This triggers the event `E1` at the output adapter `OUT` and places the selected data value at the output `D1`.

Technical Features

- **Adapter Encapsulation:** By using unidirectional adapters (`AIS` and `AX`), the application diagram in the 4diac IDE remains uncluttered, as data and event lines are bundled into a single connection.

- **Asynchronous Event Handling:** Any change to one of the inputs (`IN0`, `IN1`) or to the selector (`G`) automatically retriggers the selection process and updates the output accordingly.
- **Asynchronous Event Handling:** * **Data Type Specification:** The internal data copiers (`F_MOVE`) are hard-coded to the data type `STRING`, indicating that the signals to be switched are transmitted as strings in the `AIS` adapter.

State Overview

Selector State (`G`)	Last Event on…	Active Output (`OUT.D1`)	Description
`FALSE`	`IN0.E1` or `G.E1`	Value of `IN0.D1`	Input 0 is active. Changes to `IN1` have no effect on the output.
`TRUE`	`IN1.E1` or `G.E1`	Value of `IN1.D1`	Input 1 is active. Changes to `IN0` have no effect on the output.

Application Scenarios

- **Sensor Redundancy / Failsafe Systems:** Switching between a primary sensor and a backup sensor in case of signal loss or malfunction.
- **Operating Mode Selection:** Dynamically forwarding different parameter sets (e.g., automatic vs. manual setpoints) to an actuator. * **Signal Routing in Agricultural Machinery (ISOBUS Context):** Channel control for analog process values or status messages in complex control networks.

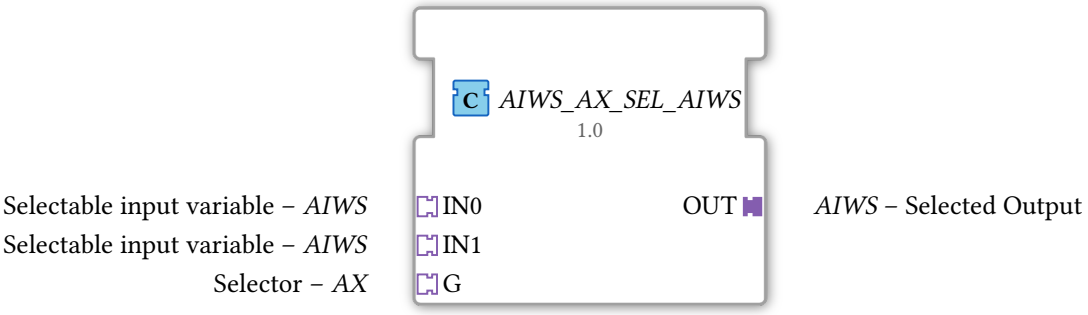
Comparison with Similar Function Blocks

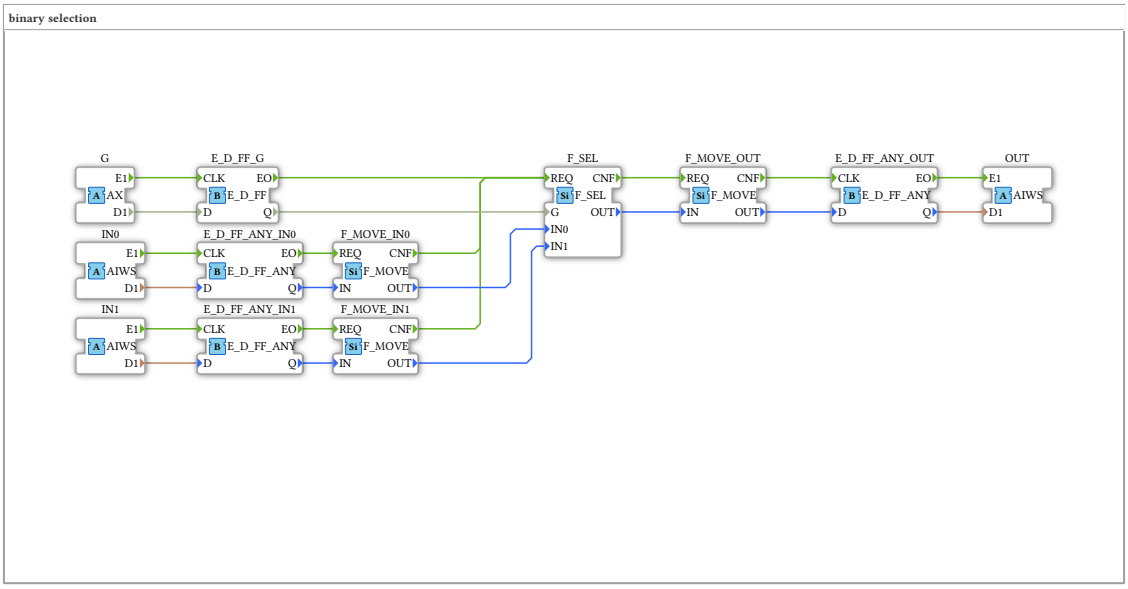
- **Classic `SEL` (IEC 61131-3):** The classic `SEL` function block does not have event control and is purely data flow-oriented. `AIS_AX_SEL_AIS` extends this principle to include event-based control for distributed systems.
- **Standard Selection Function Blocks without Adapters:** Typical IEC 61499 selection functions often use many individual event and data pins. This function block offers significantly improved maintainability and modularity in system design thanks to its adapter interfaces.

Conclusion

The **AIS_AX_SEL_AIS** is a specialized and high-performance solution for signal switching in the 4diac IDE. Its strict use of adapters and reliable internal event processing make it ideally suited for clean, modular software architectures in the fields of industrial automation and agricultural technology.

AIWS_AX_SEL_AIWS





(No image available)

• – ○ • – ○ • – ○ •

The function block `AIWS_AX_SEL_AIWS` is used for binary selection between two unidirectional adapter signals of type `AIWS`. Which of the two inputs is switched to the output is controlled by a control adapter of type `AX`.

The function block encapsulates the classic selection logic (comparable to the `SEL` standard function block from IEC 61131-3) within the event-driven environment of IEC 61499. This enables clean, modular, and transparent signal switching directly at the adapter level, eliminating the need to individually wire event and data lines in the higher-level system.

Because it is an adapter-based function block, the block itself does not have direct, classic event or data inputs and outputs on its outer casing. All communication is handled via the adapter interfaces.

No direct event inputs available (events are received via the adapter sockets).
No direct event outputs available (events are sent via the adapter plug).
No direct data inputs available.
No direct data outputs available.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Sockets (Input Adapters)

- **`G`** (Type: `adapter::types::unidirectional::AX`):

The selector adapter. The data signal it contains determines which input is activated.

- **`IN0`** (Type: `adapter::types::unidirectional::AIWS`):

The first selectable input adapter. This is passed through to the output when the selector signal is in the state `FALSE` (0).

- **`IN1`** (Type: `adapter::types::unidirectional::AIWS`):

The second selectable input adapter. This is passed through to the output when the selector signal is in the state TRUE (1).

Plugs (Output Adapters)

- **OUT** (Type: adapter::types::unidirectional::AIWS):

The output adapter. It outputs the data and events of the currently selected input (IN0 or IN1).

Functionality

The function block's internal network implements signal forwarding and event synchronization as follows:

1. **Event Acquisition and Buffering:**

As soon as the event E1 arrives at one of the sockets (G, IN0, or IN1), the corresponding data value D1 is temporarily stored via an internal event-driven flip-flop (E_D_FF for the selector and E_D_FF_ANY for the data).

1. **Data Conversion / Assignment:**

The data from inputs IN0 and IN1 are transferred to the selection block via internal assignment elements (F_MOVE) of type WSTRING.

1. **Selection Process (F_SEL):**

The internal function block F_SEL (IEC 61131-3 SEL) evaluates the state of the selector from the adapter G:

- If the signal from G is in the state FALSE, the signal from IN0 is passed to the output.
- If the signal from G is in the state TRUE, the signal from IN1 is passed to the output.
- If the signal from G is in the state TRUE, the signal from IN1 is passed to the output. 4. **Output:**

The selected signal is passed via another F_MOVE block and an output flip-flop (E_D_FF_ANY_OUT) to data point D1 of the output plug OUT. Simultaneously, the event E1 is triggered at the output to inform subsequent blocks about the data update.

- **Data type WSTRING:** The adapter data is processed internally via F_MOVE blocks with the data type WSTRING. This means that the user data within the AIWS adapter is transmitted and switched as wide strings.
- **Event Decoupling:** By using flip-flops (E_D_FF), it is ensured that any change to the inputs or the selector results in an immediate and consistent update of the output.

The logical mapping of the output, depending on the selector G, is as follows:

State of G (selector)	Active path	Data value to OUT.D1	Event to OUT.E1
FALSE	IN0	Value of IN0.D1	Triggered by events at IN0.E1 or G.E1
TRUE	IN1	Value of IN1.D1	Triggered by events at IN1.E1 or G.E1

- **Switching Analog Values with Status (formatted as WSTRING):** Redundant sensor systems where, in case of a fault, the system should switch from sensor 1 (IN0) to a backup sensor 2 (IN1).
- **Dynamic Text or Recipe Selection:** Switching configuration strings, log messages, or setpoints that are routed to an HMI or control unit via adapter structures.
- **Signal Routing:** Flexible routing in modular industrial plants where data streams need to be redirected depending on the operating mode (e.g., automatic vs. manual operation).

Comparison with Similar Function Blocks

- **Standard SEL (IEC 61131-3):** The classic SEL function block operates on elementary data types (e.g., INT, REAL, STRING) and does not have native event control or adapter support. AIWS_AX_SEL_AIWS extends this principle for IEC 61499 by providing direct switching for complex adapter types.
- **Multiplexer (MUX):** A classic multiplexer allows selection from more than two channels via an integer index. The AIWS_AX_SEL_AIWS is optimized for fast and resource-efficient binary selection (2 channels).

The AIWS_AX_SEL_AIWS is a highly specialized auxiliary module for IEC 61499 systems (such as 4diac-ide) that elegantly and deterministically switches complex data and event streams between two adapters. It significantly reduces "spaghetti code" and confusing wiring in graphical programming environments.

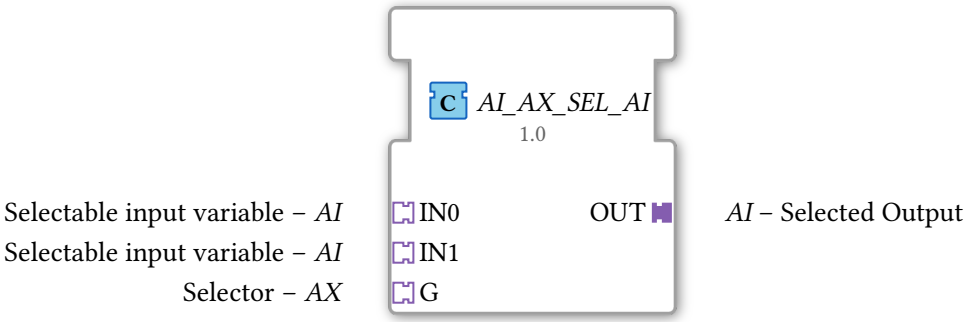
Technical Features

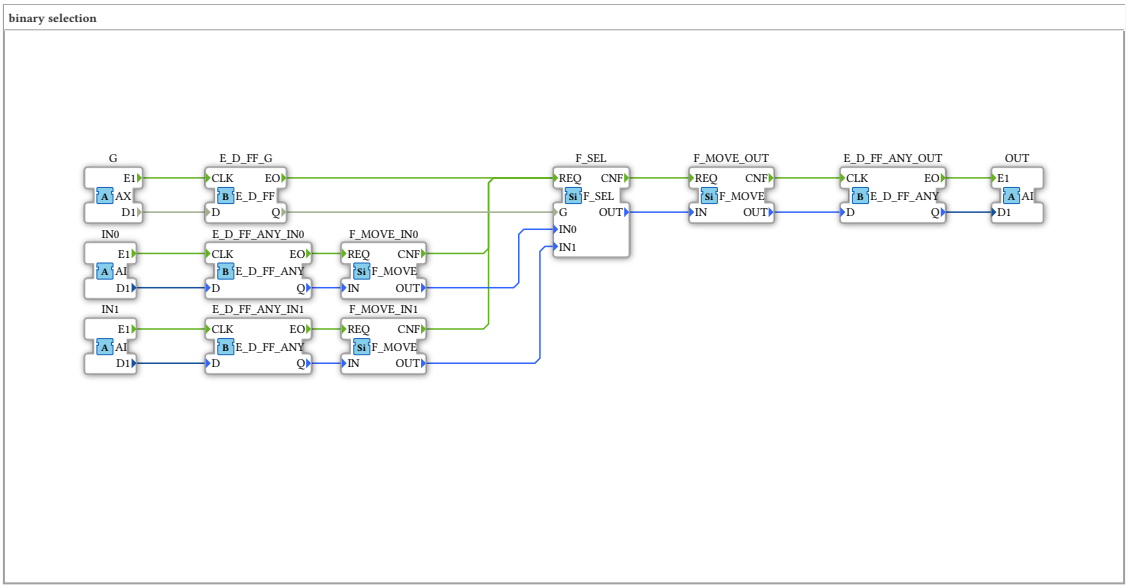
State Overview

Application Scenarios

Conclusion

AI_AX_SEL_AI





(No image available)

• – ○ • – ○ • – ○ •

The function block `AI_AX_SEL_AI` is a binary selector (multiplexer) for analog signals based on the IEC 61499 architecture. It serves to select one of two analog input signals based on a control signal (selector) and forward it to the output. The unique feature of this block lies in its consistent use of adapters, which significantly simplifies and structures the cabling within the 4diac-ide development environment.

The function block does not have traditional discrete inputs and outputs for data and events at its main level. Instead, all communication is implemented via standardized adapters.

No direct event inputs available (fully encapsulated via adapters).
No direct event outputs available (fully encapsulated via adapters).
No direct data inputs available (fully encapsulated via adapters).
No direct data outputs available (fully encapsulated via adapters).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- **G** (Type: `adapter::types::unidirectional::AX`): The selector input. This adapter provides the control signal (Boolean state) that determines which of the two analog inputs is routed to the output.
- **IN1** (Type: `adapter::types::unidirectional::AI`): Selectable analog input 1. This input is enabled when the selector signal `G` is in the state `TRUE` (1).
- **OUT** (Type: `adapter::types::unidirectional::AI`): Selected analog output. It outputs the signal of the currently active input (`IN0` or `IN1`).

Plugs (Ausgangs-Schnittstellen)

Sockets (Eingangs-Schnittstellen)

Functionality

The module's internal logic controls the data and event flow as follows:

1. **Event Control & Data Acquisition:**

- Each time an event (`E1`) arrives at one of the input adapters (`IN0`, `IN1`, or `G`), the corresponding data element (`D1`) is temporarily stored via internal edge- or event-triggered memory modules (`E_D_FF` and `E_D_FF_ANY`).
- – Internal `F_MOVE` function blocks convert and pass the values (as data type `INT`) to the central selection function block.

1. **Selection Logic:**

- The core function block `F_SEL` (a standardized IEC 61131-3 selection function block) evaluates the control signal `G`.
- If the control signal `G` is `FALSE`, the value of `IN0` is selected.
- – If the control signal `G` is on `TRUE`, the value of `IN1` is selected.

1. **Output:**

- After successful selection, the selected value is passed via another `F_MOVE` block and an output flip-flop (`E_D_FF_ANY_OUT`) to the output plug `OUT`.
- Simultaneously, the output event `OUT.E1` is triggered to inform subsequent blocks about the value change.

- **Data Type Specification:** Internally, the function block uses the data type `INT` (integer) for analog values, which is defined by the parameterization of the internal `F_MOVE` instances.
- **Adapter-Based Architecture:** By encapsulating events and data in the adapters `AI` (Analog Input) and `AX` (Auxiliary Control), the number of connection lines in the higher-level system diagram is drastically reduced.
- **Event-Driven Propagation:** Every update of an input (regardless of whether it is the data or the selector) triggers the recalculation and output of an event at the output.

–

The function block operates purely on an event- and data-flow-driven basis. There are no complex internal states (no state machine/ECC in the classical sense), but rather a direct dependency of the output on the inputs:

Selector input (`G.D1`)	Active path	Output value (`OUT.D1`)	Output event (`OUT.E1`)
`FALSE`	`IN0`	Value of `IN0.D1`	Triggered on every event
`TRUE`	`IN1`	Value of `IN1.D1`	Triggered on every event

- **Redundant Sensors:** Switching between a primary and a secondary (backup) analog sensor in the event of a detected failure.
- **Manual/Automatic Switching:** Selection between a manually setpoint (e.g., via an HMI connected to `IN0`) and an automatically calculated controller setpoint (connected to `IN1`).
- **Recipe-Controlled Signal Redirection:** Dynamic assignment of analog process parameters based on the current operating state of a system.

–

- **Standard `F_SEL` (IEC 61131-3):** The classic `F_SEL` function block requires discrete pins for events and data. The `AI_AX_SEL_AI` module encapsulates this functionality in user-friendly adapter structures, increasing reusability and clarity in complex IEC 61499 applications.
- **Classic Multiplexers (e.g., MUX):** While a multiplexer often handles multiple channels (e.g., 4 or 8), this module is optimized as a slim, highly efficient 2-way selector switch for dedicated analog signals.

The `AI_AX_SEL_AI` module provides an elegant, adapter-based solution for the binary selection of analog signals in distributed control systems. Thanks to the clean separation and encapsulation of the interfaces in standardized adapters, it is ideally suited for use in modular and service-oriented automation architectures within Eclipse 4diac.

Technical Features

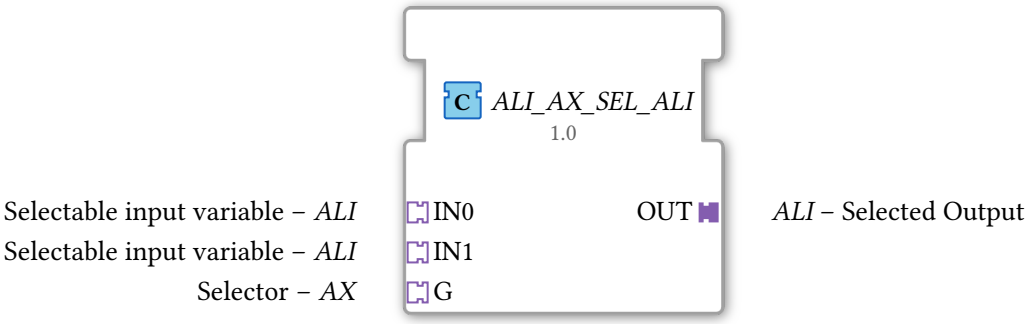
State Overview

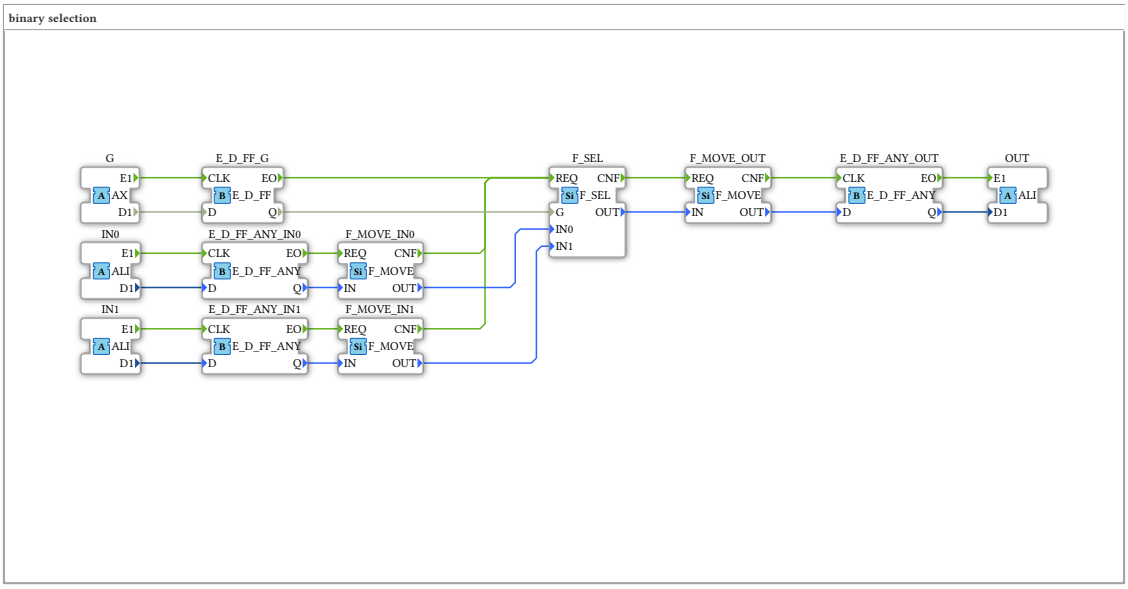
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

ALI_AX_SEL_ALI





[Image of the function block]

• – ○ • – ○ • – ○ •

Introduction

The function block `ALI_AX_SEL_ALI` is a binary selection block designed for use with unidirectional adapters. It allows you to select between two input signals based on a selector signal and then output the selected signal. This block encapsulates the standard selection logic and adapts it to an event-driven adapter interface.

Interface Structure

Because it is an adapter-based function block, it does not have direct, traditional event or data inputs at the top level. All communication is handled via adapters.

Event Inputs

No direct event inputs available (control is via the adapters).

Event Outputs

No direct event outputs available (control is via the adapters).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Interfaces)

- **G** (`adapter::types::unidirectional::AX`): The selector adapter (gate). It controls which of the two inputs is routed to the output.
- **IN0** (`adapter::types::unidirectional::ALI`): The first selectable input adapter (selected by default for `G = FALSE`).
- **IN1** (`adapter::types::unidirectional::ALI`): The second selectable input adapter (selected at `G = TRUE`).

Plugs (Output Interfaces)

- **OUT** (`adapter::types::unidirectional::ALI`): The output adapter that provides the selected signal and the corresponding event.

Functionality

The function block operates as a multiplexer for the adapter signals:

1. **Event Detection**: As soon as an event (`IN0`, `IN1`) is received by one of the input adapters (`G`) or the selector adapter (`G`), it is intercepted via internal edge-triggered flip-flops (`E_D_FF` and `E_D_FF_ANY`, respectively).
2. **Data Buffering**: The data from the inputs (`IN0.D1` and `IN1.D1`) are passed via internal `F_MOVE` function blocks (configured to the data type `LINT`) to the central selection function block `F_SEL`.
3. **Selection**: The function block `F_SEL` evaluates the state of the selector `G.D1`:

- If `G.D1` is set to `FALSE`, the value of `IN0` is selected.
- If `G.D1` is set to `TRUE`, the value of `IN1` is selected.

1. **Output**: The selected value is passed to the flip-flop `E_D_FF_ANY_OUT`. This updates the output data point `OUT.D1` and simultaneously triggers the output event `OUT.E1`.

Technical Features

- **Data Type Specificity**: The internal shift blocks (`F_MOVE`) are fixed to the data type `LINT` (Long Integer). This means that the data transmitted via the `ALI` adapter is interpreted and processed as `LINT`.
- **Event Decoupling**: The use of flip-flops (`E_D_FF`) ensures that data changes at the inputs arrive at the output consistently and synchronously with the corresponding events.

State Overview

The function block does not have its own state hierarchy (no ECC) because it is implemented as a composite network function block (FB). Its behavior is purely data- and event-flow-driven:

State Selector (`G.D1`)	Trigger Event	Action / Result
`FALSE`	Event on `IN0.E1`, `IN1.E1`, or `G.E1`	The value of `IN0.D1` is passed to `OUT.D1`; `OUT.E1` is triggered.
`TRUE`	Event on `IN0.E1`, `IN1.E1`, or `G.E1`	The value of `IN1.D1` is passed to `OUT.D1`; `OUT.E1` is triggered.

Application Scenarios

- **Setpoint Switching**: Selection between an automatic setpoint (e.g., calculated by a controller) and a manually specified setpoint in an industrial plant.
- **Sensor Redundancy**: Switching between two redundant sensors (e.g., in case of failure or for calibration) that provide their data via standardized adapter interfaces.
- **Recipe-Controlled Path Selection**: Dynamic selection of process parameters based on a binary control signal in the sequence program.

–

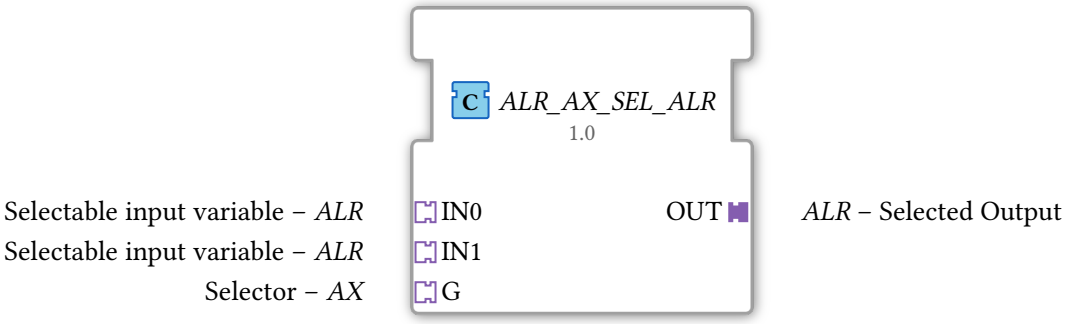
Comparison with Similar Function Blocks

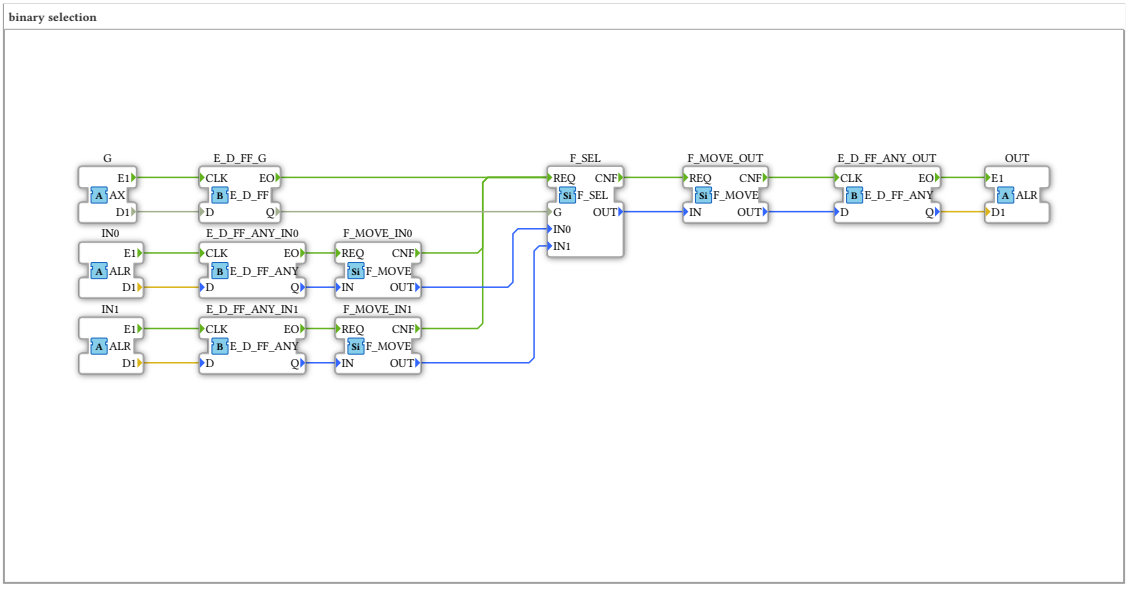
- **F_SEL (Standard IEC 61131-3)**: The standard selection effect block works directly with elementary data types and requires explicit event connections. `ALI_AX_SEL_ALI` encapsulates this component and extends it with direct support for the more structured and cleaner use of unidirectional adapters (`ALI`/`AX`).
- **Classic Multiplexers (MUX)**: While a MUX can often handle multiple channels (e.g., 4, 8), this component is strictly optimized for binary selection (2 channels) but offers streamlined adapter integration.

Conclusion

The `ALI_AX_SEL_ALI` is a specialized auxiliary component that simplifies signal and event routing in complex 4diac systems. By encapsulating the adapter logic, it significantly improves the clarity of the application network by eliminating the need for manually unpacking and repackaging adapter signals.

ALR_AX_SEL_ALR





• – ○ • – ○ • – ○ •

The function block ALR_AX_SEL_ALR is used for binary selection between two analog input signals provided by adapters of type ALR. This selection is controlled by a selector adapter of type AX. The selected signal is passed on to an output adapter of type ALR.

The function block is internally based on the IEC 61131-3 selection function F_SEL and is optimized for event-driven IEC 61499 architectures.

Since this component is designed as an adapter coupler, it does not have any classic, direct event or data interfaces at the top level. All communication is handled via sockets (input adapters) and plugs (output adapters).

No direct event inputs available.
No direct event outputs available.
No direct data inputs available.
No direct data outputs available.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- **Sockets (Input Interfaces):**
- `IN0` (Type: `adapter::types::unidirectional::ALR`): First selectable input channel (switched to output when selector `G` has the value `FALSE`).
- `IN1` (Type: `adapter::types::unidirectional::ALR`): Second selectable input channel (switched to output when selector `G` has the value `TRUE`).
- `G` (Type: `adapter::types::unidirectional::AX`): Selector input for controlling channel selection.
- **Plugs (Output Interfaces):**
- `OUT` (Type: `adapter::types::unidirectional::ALR`): Output carrying the selected signal.

Functionality

This module implements a software-controlled 1-out-of-2 selection based on adapters:

1. **Event Buffering:** Incoming events and data at sockets `IN0`, `IN1`, and `G` are buffered via internal event D flip-flops (`E_D_FF` and `E_D_FF_ANY`, respectively) to ensure consistent data transfer.
2. **Value Transfer:** Data values are prepared for the data type `LREAL` via internal assignment elements (`F_MOVE`).
3. **Selection Logic (`F_SEL`):** Selection is performed using the standard selection block:

- If the signal at `G.D1` equals `FALSE`, the value of `IN0.D1` is passed to the output.
- If the signal at `G.D1` equals `TRUE`, the value of `IN1.D1` is passed to the output.

1. **Output Triggering:** Any change to one of the inputs or the selector triggers an update of the output adapter `OUT`, signaled by the event `OUT.E1` and the new data value `OUT.D1`.

- **Precise Data Processing (LREAL):** The internal data paths are designed for the data type `LREAL`, enabling lossless transmission of highly accurate analog measurements.
- **Adapter Encapsulation:** The use of standardized adapters (`ALR`, `AX`) eliminates the need for complex individual wiring of data and event pins in the application editor.
- **Event-driven:** The output reacts dynamically and without delay to value changes and triggers at the inputs.

The behavior of the function block is determined by the data and event flow in the internal network:

State Selector `G.D1`	Channel Passed	Output Value `OUT.D1`	Output Event `OUT.E1`
`FALSE`	Channel 0 (`IN0`)	Value of `IN0.D1`	Triggers on events on `IN0.E1` or changes to `G`
`TRUE`	Channel 1 (`IN1`)	Value of `IN1.D1`	Triggered by events on `IN1.E1` or a change in `G`

- **Sensor Redundancy / Failover:** Switching between a primary sensor (`IN0`) and a backup sensor (`IN1`) in case of failure or signal interference, controlled by a diagnostic signal at `G`.
- **Setpoint Switching:** Switching between an automatic setpoint and a manual target value in process control loops.
- **Recipe and Parameter Set Selection:** Dynamic assignment of different analog process parameters based on the current plant state.

Compared to a classic IEC 61131-3 `SEL` function block, `ALR_AX_SEL_ALR` completely encapsulates the signal paths in adapter structures. This significantly reduces the visual complexity in 4diac application diagrams, as only structured adapter lines need to be drawn instead of multiple individual connections.

The ALR_AX_SEL_ALR is a robust and reusable function block for structured signal switching in IEC 61499. Through the consistent use of adapters and the support of high-resolution LREAL data, it is ideally suited for modern control concepts in industrial automation.

Technical Features

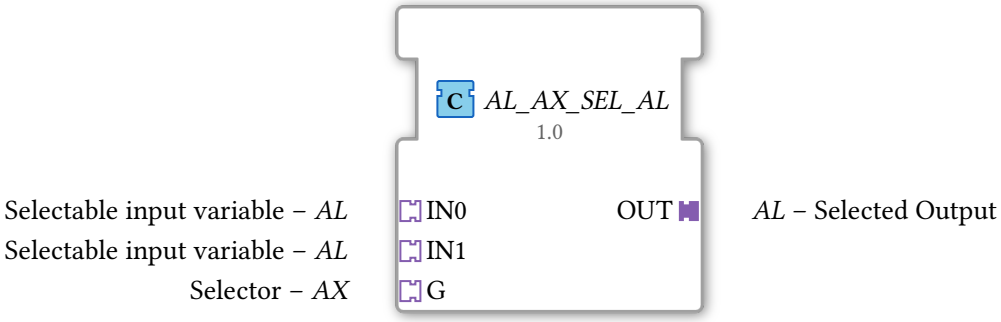
State Overview

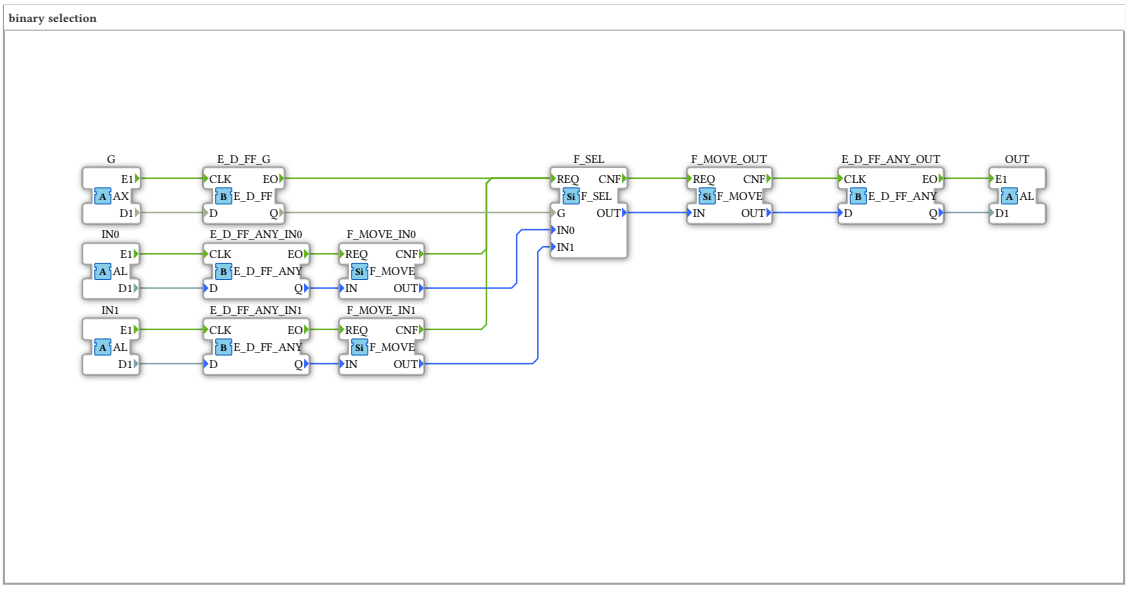
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AL_AX_SEL_AL





(No image available)

• – ○ • – ○ • – ○ •

Introduction

The function block AL_AX_SEL_AL is an event-driven selector (binary selection) for IEC 61499 systems in the 4diac IDE. It acts as a multiplexer, forwarding data from one of two input adapters to an output adapter based on the state of a selector signal. The block consistently leverages the advantages of adapters to neatly encapsulate data and event streams and minimize the complexity of the wiring in the application diagram.

Interface Structure

Since this function block is internally structured as a network and is entirely based on adapter interfaces, it does not have any directly accessible, traditional input or output variables.

Event Inputs
No direct event inputs are available. Event control is handled implicitly via the connected adapters.

Event Outputs
No direct event outputs are available. Event forwarding is handled implicitly via the output adapter.

Data Inputs
No direct data inputs are available.

Data Outputs
No direct data outputs are available.

Adapters

Plugs
- **`OUT`** (Type: `adapter::types::unidirectional::AL`):

The selected output. This adapter outputs the selected data (`D1`) and the associated update event (`E1`).

Sockets
- **`G`** (Type: `adapter::types::unidirectional::AX`):

The selector input (gate/selector). Controls which of the two inputs (`IN0` or `IN1`) is routed to the output.

- **`IN0`** (Type: `adapter::types::unidirectional::AL`):

The first selectable input channel. This is active and routed to the output when the selector `G` is in the state `FALSE` (or logic 0).

- **`IN1`** (Type: `adapter::types::unidirectional::AL`):

The second selectable input channel. This is active and is passed to the output when the selector `G` has the state `TRUE` (or logic 1).

Functionality
The component implements classic multiplexer logic (equivalent to `SEL` in IEC 61131-3) in the IEC 61499 context:

1. **Event Handling & Buffering:**

When an event occurs on one of the input adapters (`IN0`, `IN1`) or the selector adapter (`G`), internal D flip-flops (`E_D_FF` and `E_D_FF_ANY`, respectively) capture the data values and store them temporarily.

1. **Data Type Conversion:**

The buffered data values are forwarded to the central selection block via internal transfer blocks (`F_MOVE` with the data type `LWORD`).

1. **Selection Logic (`F_SEL`):**

The core of the network evaluates the state of the selector `G`:

- If `G` equals `FALSE`, the signal from `IN0` is passed to the output.
- If `G` equals `TRUE`, the signal from `IN1` is passed to the output.

1. **Output:**

After a successful selection, the result is transferred via another flip-flop to the plug `OUT` and made available there simultaneously with an event (`OUT.E1`).

Technical Features
- **Type Openness through LWORD Abstraction:** The internal use of `LWORD` (64-bit) for the copy operations (`F_MOVE`) allows for high flexibility in the data being transferred, as various bit and integer patterns can be transmitted without information loss.
- **Full Event Control:** Any change to any input or selector triggers an immediate recalculation. This ensures that consistent and up-to-date data is always available at the output.

State Overview

Selector State (G.D1)	Output Value (OUT.D1)	Trigger Event
FALSE (0)	Value of IN0.D1	Event on IN0 or G
TRUE (1)	Value of IN1.D1	Event on IN1 or G

Application Scenarios

- **Setpoint Switching:** Switching between a standard setpoint (IN0) and an alternative setpoint (IN1) based on an external control signal.
- **Manual/Automatic Operation:** Switching a control signal for a valve or motor between a manually specified value (e.g., from a visualization) and the calculated value of an automatic controller.
- – **Sensor Redundancy:** Dynamic switching to a backup sensor (IN1) if the primary sensor (IN0) is flagged as faulty via a diagnostic function (switching controlled by the fault bit on G).

Comparison with Similar Function Blocks

Compared to a standard SEL function block from the IEC 61131-3 library, AL_AX_SEL_AL offers the following advantages:

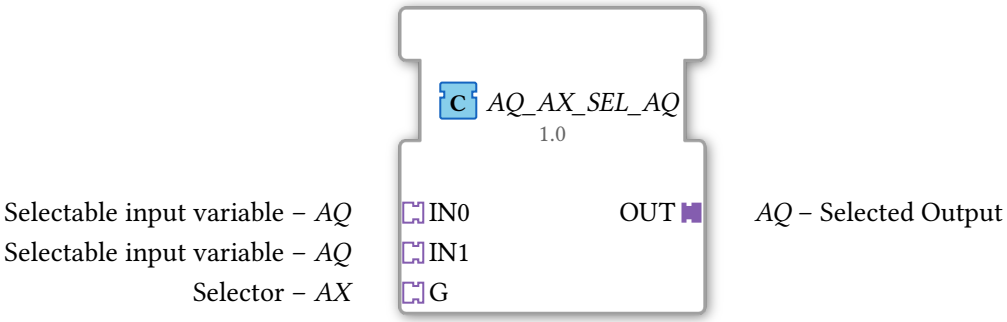
- **Reduced Wiring Effort:** Encapsulation in adapters eliminates the need to run separate event and data lines. A single connection path per channel is sufficient.

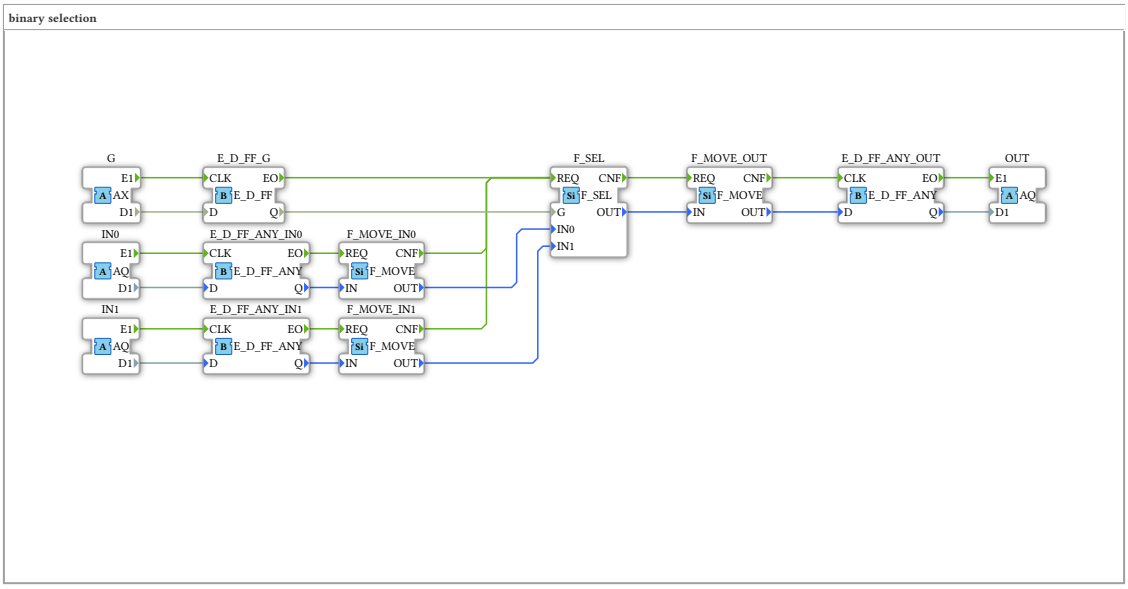
- **Reduced Wiring Effort:** Encapsulation in adapters eliminates the need for separate event and data lines. A single connection path per channel is sufficient.
- – **Event Coupling:** The function block inherently responds to the event triggers integrated into the adapters, eliminating the need for manual event-based linking in the FBD (Function Block Diagram).

Conclusion

The AL_AX_SEL_AL function block is an efficient and clean solution for switching tasks in complex 4diac applications. Through the consistent use of unidirectional adapters, it promotes modular software design and ensures clear control topologies while providing robust, event-driven signal processing.

AQ_AX_SEL_AQ





• – ○ • – ○ • – ○ •

The function block `AQ_AX_SEL_AQ` is used for binary selection between two analog or value-based input signals provided via adapters. A selector adapter controls which of the two inputs is passed through to the output. The block encapsulates the classic selection logic and prepares it for event-driven execution in IEC 61499 environments.

The function block communicates exclusively via standardized adapter interfaces. This simplifies modeling in the application network, as related data and event lines are transmitted together.

The block does not have direct event inputs. Event processing is handled implicitly via the events of the connected input adapters (`IN0`, `IN1`, `G`).

The function block does not have direct event outputs. Events are forwarded via the output adapter (`OUT`).

There are no direct data inputs on the function block interface. Data is transmitted encapsulated via the sockets.

There are no direct data outputs. Data is provided encapsulated via the plug.

- **G** (Type: `adapter::types::unidirectional::AX`):
- **Description**: Selector (selection signal).
- **Data**: `D1` (Selector state).
- **Event**: `E1` (Trigger on selector value change).
- **IN0** (Type: `adapter::types::unidirectional::AQ`):
- **Description**: First selectable input channel (selected at `G = FALSE`).
- **Data**: `D1` (Input value).
- **Event**: `E1` (Trigger on value change at IN0).
- **IN1** (Type: `adapter::types::unidirectional::AQ`):
- **Description**: Second selectable input channel (selected with `G = TRUE`).
- **Data**: `D1` (Input value).
- **Event**: `E1` (Trigger on value change at IN1).
- **OUT** (Type: `adapter::types::unidirectional::AQ`):
- **Description**: The selected output channel.
- **Data**: `D1` (Output value of the selected input).
- **Event**: `E1` (Output event for signaling a new valid value).

Plugs (Ausgangs-Schnittstellen)

Sockets (Eingangs-Schnittstellen)

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

Functionality

The function block is implemented internally as a function block network and operates according to the following sequence:

1. **Event Acquisition**: As soon as an event `E1` arrives at one of the inputs (`IN0`, `IN1`) or the selector (`G`), it is intercepted and synchronized via edge- or value-triggered auxiliary function blocks (`E_D_FF` and `E_D_FF_ANY`).
2. **Data Buffering**: The current values of the adapter inputs are buffered in the internal network using function blocks `F_MOVE` and formatted according to the specified type.
3. **Selection Logic**: The internal core function block `F_SEL` (based on the standardized selection function according to IEC 61131-3) evaluates the state of the selector:

- If `G.D1` is set to `FALSE`, the value of `IN0` is passed through.
- If `G.D1` is set to `TRUE`, the value of `IN1` is passed through.

1. **Output**: The selected value is passed to the output plug `OUT.D1`. Simultaneously, the output event `OUT.E1` is triggered to inform subsequent program components in the control network about the value change.

- **Event-Data Consistency**: The use of internal D flip-flops ensures that the data values are read and processed at the exact moment the corresponding change event occurs.
- **Typing**: The function block internally uses `F_MOVE` with the data type `BYTE` to encapsulate and assign the adapter data. This means that the `AQ` interfaces used transmit binary- or byte-compatible data structures.

State Overview

Since `AQ_AX_SEL_AQ` is a purely data- and event-driven combination block without its own internal state machine (ECC), its behavior can be described using the following logical table:

Selector (`G.D1`)	Value `IN0.D1`	Value `IN1.D1`	Output (`OUT.D1`)	Behavior of the Output Event (`OUT.E1`)
FALSE	`Wert_A`	`Wert_B`	**`Wert_A`**	Fires as soon as an event occurs at `IN0.E1`, `IN1.E1`, or `G.E1`.
TRUE	`Wert_A`	`Wert_B`	**`Wert_B`**	Fires as soon as an event occurs at `IN0.E1`, `IN1.E1`, or `G.E1`.

- **Sensor Redundancy / Switchover**: Automatic or manual switching between a primary sensor (`IN0`) and a secondary/backup sensor (`IN1`) in case of a fault via the control signal `G`.
- **Manual/Automatic Operation**: Selection of setpoints for actuators. In automatic mode (`G = TRUE`), the calculated controller value (`IN1`) is used; in manual mode (`G = FALSE`), a predefined fixed value or a manually specified parameter (`IN0`) is used.
- **Recipe-Controlled Parameterization**: Dynamic switching of process limits during operation.

2.6 · Comparison with Similar Components

- **Standard SEL (IEC 61131-3)**: The classic SEL component has no event control and reacts purely cyclically to incoming data. AQ_AX_SEL_AQ extends this functionality with the event-based paradigm of IEC 61499.
- **Classic Multiplexers (MUX)**: While a multiplexer usually selects from a large number of inputs, this component is specifically limited to efficient 1-out-of-2 selection (binary selection), which minimizes parameterization and wiring effort.

The AQ_AX_SEL_AQ is a useful infrastructure component for modular 4diac applications. By completely encapsulating the signals in adapter connections, it significantly contributes to clarity in the application window and ensures robust, event-driven signal forwarding.

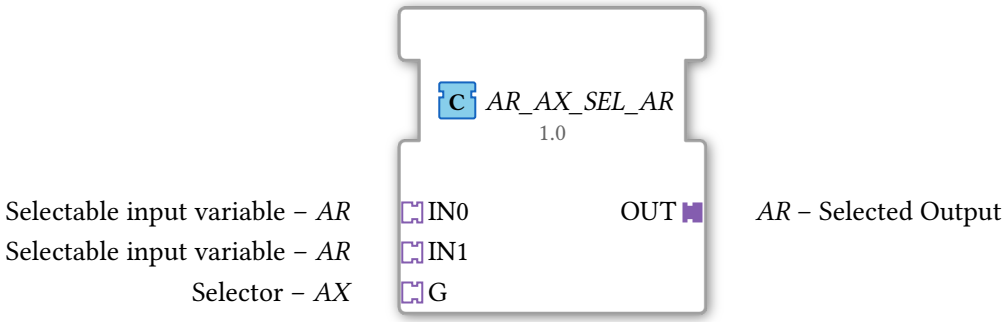
Technical Features

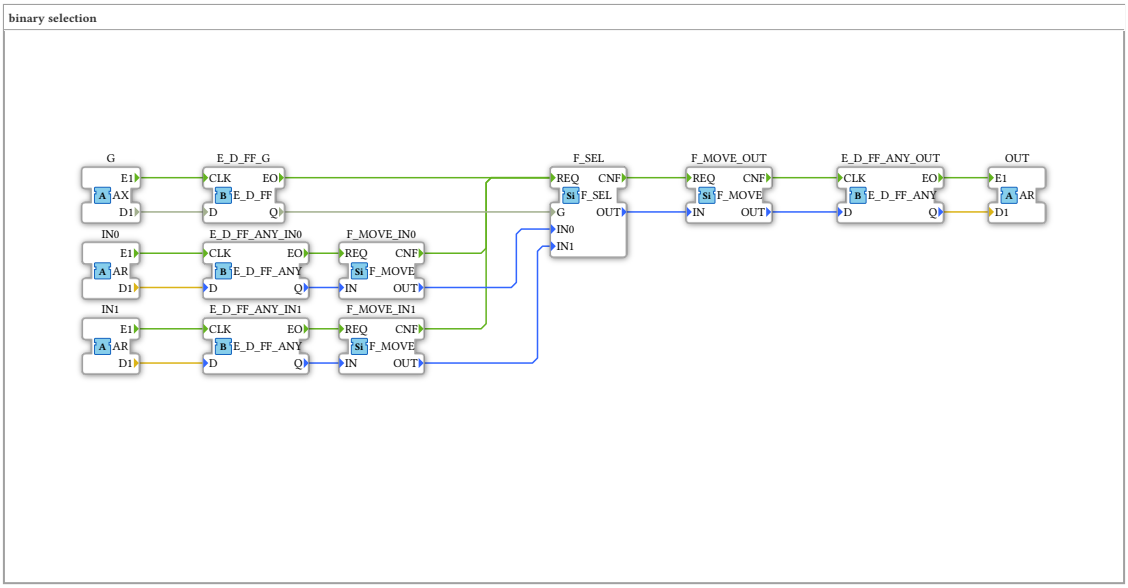
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AR_AX_SEL_AR





• – ○ • – ○ • – ○ •

The function block AR_AX_SEL_AR is used for binary selection between two analog input signals transmitted via adapters. Based on the state of a selector signal, one of the two inputs is passed through to the output.

By consistently using adapters instead of traditional discrete data and event pins, the complexity of the wiring in the higher-level IEC 61499 application diagram is significantly reduced.

Since this function block relies entirely on adapter-based communication, it has no direct, traditional event or data interfaces at the top level. All communication is handled via the declared adapters.

No direct event inputs available (events are received via the adapter interfaces).
No direct event outputs available (events are sent via the adapter interfaces).
No direct data inputs available.
No direct data outputs available.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- **G** (Type: `adapter::types::unidirectional::AX`):

The selector adapter. The signal present here controls which of the two inputs (`IN0` or `IN1`) is passed to the output.
The first selectable signal input. This value is passed to output `OUT` when selector `G` is in state `FALSE` (0).

- **IN1** (Type: `adapter::types::unidirectional::AR`):

The second selectable signal input. This value is passed to output `OUT` when selector `G` is in state `TRUE` (1).

- **OUT** (Type: `adapter::types::unidirectional::AR`):

The selected output adapter. It provides the value of the currently active input, including the corresponding update event.

Plugs (Ausgangsschnittstellen)

Sockets (Eingangsschnittstellen)

Functionality

Inside the function block `AR_AX_SEL_AR` is a network of standard IEC 61131-3 and IEC 61499 components that process the adapter signals:

1. **Event and Data Acquisition:**

As soon as an event `E1` arrives at one of the input adapters (`IN0`, `IN1`, or `G`), it is intercepted by internal edge-triggered flip-flops (`E_D_FF` or `E_D_FF_ANY`). The corresponding data values (`D1`) are temporarily stored.

1. **Data Conversion:**

The values of the analog inputs are routed through conversion blocks of type `F_MOVE` (configured to data type `REAL`) to ensure consistent data processing.

1. **Selection Logic (Multiplexing):**

The standard selection block `F_SEL` performs the actual selection:

- If the value of `G.D1` equals `FALSE`, the signal from `IN0` is passed to the output.
- If the value of `G.D1` is equal to `TRUE`, the signal is passed on to `IN1`.

1. **Output:**

The selected signal is passed via another `F_MOVE` block to the output flip-flop `E_D_FF_ANY_OUT`. This flip-flop generates the output event `E1` at plug `OUT` and makes the selected value available to `OUT.D1`.

- **Unidirectional Adapter Structure:** The function block uses unidirectional adapter types (`AR` for analog values, `AX` for binary values), ensuring clear and interference-free signal direction within the system.
- **Type Consistency:** Internally, the function block uses the data type `REAL` for analog signals (configured via the attributes of the `F_MOVE` function blocks).

- **Event-Driven:** The output is recalculated and updated immediately as soon as either the selection `G` changes or new values are signaled at the inputs `IN0` or `IN1`.

The behavior can be described using the following simple logic table:

State Selector (`G.D1`)	Value at Output (`OUT.D1`)	Triggering Event
`FALSE`	Value of `IN0.D1`	Event at `IN0.E1` or Change at `G.E1`
`TRUE`	Value of `IN1.D1`	Event at `IN1.E1` or Change at `G.E1`

- **Sensor Redundancy:** Switching between a primary and a secondary analog sensor (e.g., temperature sensor or pressure sensor) in case of primary sensor failure.
- **Manual/Automatic Switching:** Optionally applying an automatic control value (e.g., calculated by a PID controller) or a manually specified setpoint to an actuator.
- **Signal Routing:** Dynamic path selection in more complex process engineering or agricultural control applications.

–

Compared to the standard IEC 61131 function block `F_SEL`, `AR_AX_SEL_AR` offers the advantage of complete encapsulation via adapters. While the standard `F_SEL` requires events and data to be wired and synchronized separately in the network, `AR_AX_SEL_AR` reduces the wiring effort in the higher-level system to just three connections (two input adapters, one output adapter) plus the selection signal.

The `AR_AX_SEL_AR` is an extremely useful auxiliary component for IEC 61499 applications. By abstracting the data and event streams into standardized adapter channels, it ensures a streamlined application design and simplifies the implementation of switching and redundancy logic for analog signals.

Technical Features

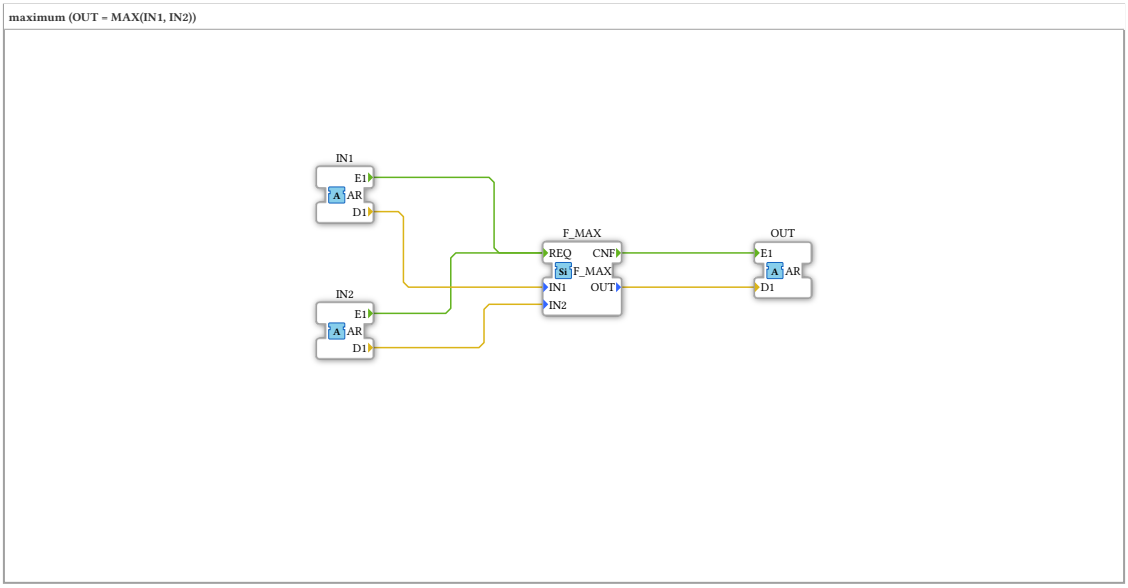
State Overview

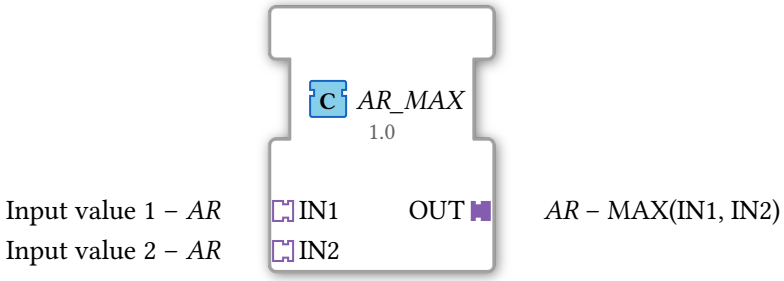
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AR_MAX





Introduction

The function block `AR_MAX` is used to determine the maximum value of two analog input signals transmitted via adapters (`AR`). It compares the data values of the two input sockets `IN0` and `IN1` and passes the larger value to the output plug `OUT`.

By consistently using adapters instead of traditional discrete data and event pins, the complexity of the wiring in the higher-level IEC 61499 application diagram is significantly reduced.

Interface Structure

Since this function block relies entirely on adapter-based communication, it has no direct, traditional event or data interfaces at the top level. All communication is handled via the declared adapters.

Event Inputs

No direct event inputs available (events are received via the adapter interfaces).

Event Outputs

No direct event outputs available (events are sent via the adapter interfaces).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Interfaces)

- **IN0** (Type: `adapter::types::unidirectional::AR`): The first analog input signal adapter.
- **IN1** (Type: `adapter::types::unidirectional::AR`): The second analog input signal adapter.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::AR`): The output adapter. It provides the maximum value $\max(\text{IN0.D1}, \text{IN1.D1})$ including the corresponding update event.

Functionality

As soon as an event `E1` arrives at either input adapter (`IN0` or `IN1`), the function block compares the current data values `IN0.D1` and `IN1.D1`:

$$\text{OUT.D1} = \max(\text{IN0.D1}, \text{IN1.D1}) \tag{2.122}$$

The computed maximum value is set on `OUT.D1` and emitted along with output event `OUT.E1`.

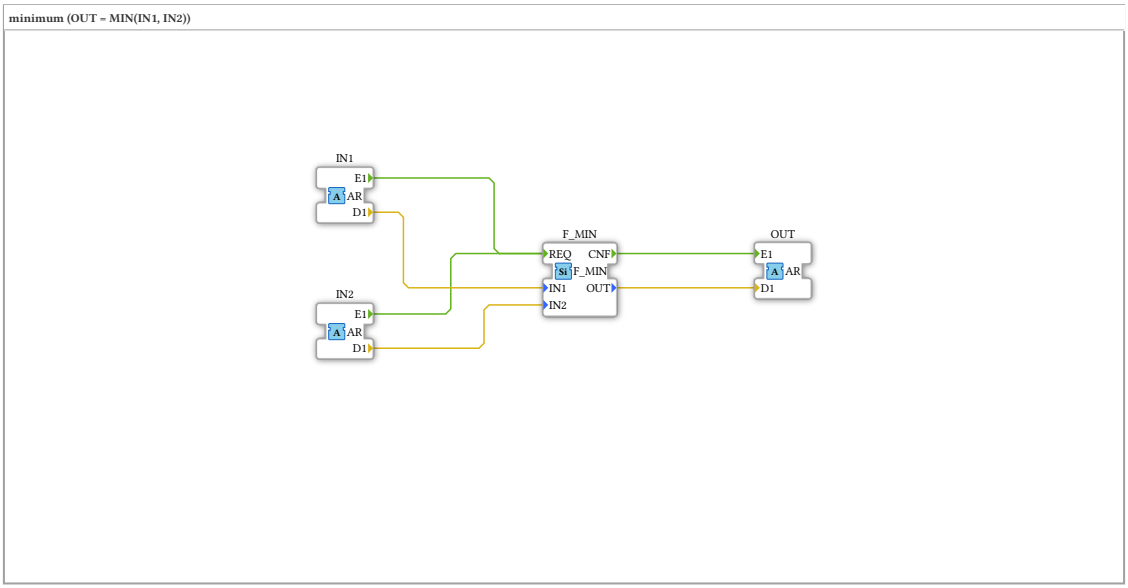
Technical Features

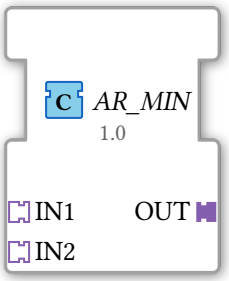
- **Unidirectional Adapter Structure**: Uses standardized `AR` adapter channels for clear signal direction.
- **Event-Driven**: Recalculation and output occur immediately upon arrival of a new measurement at `IN0` or `IN1`.
- **Type Consistency**: Operates internally using the floating-point data type `REAL`.

Conclusion

The AR_MAX is a useful selection block for calculating extreme values (maximum selection) in adapter-based IEC 61499 applications.

AR_MIN





Input value 1 – *AR*
Input value 2 – *AR*

AR – MIN(IN1, IN2)

Introduction

The function block AR_MIN is used to determine the minimum value of two analog input signals transmitted via adapters (AR). It compares the data values of the two input sockets IN0 and IN1 and passes the smaller value to the output plug OUT.

By consistently using adapters instead of traditional discrete data and event pins, the complexity of the wiring in the higher-level IEC 61499 application diagram is significantly reduced.

Interface Structure

Since this function block relies entirely on adapter-based communication, it has no direct, traditional event or data interfaces at the top level. All communication is handled via the declared adapters.

Event Inputs

No direct event inputs available (events are received via the adapter interfaces).

Event Outputs

No direct event outputs available (events are sent via the adapter interfaces).

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Interfaces)

- **IN0** (Type: `adapter::types::unidirectional::AR`): The first analog input signal adapter.
- **IN1** (Type: `adapter::types::unidirectional::AR`): The second analog input signal adapter.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::AR`): The output adapter. It provides the minimum value $\min(\text{IN0.D1}, \text{IN1.D1})$ including the corresponding update event.

Functionality

As soon as an event `E1` arrives at either input adapter (`IN0` or `IN1`), the function block compares the current data values `IN0.D1` and `IN1.D1`:

$$\text{OUT.D1} = \min(\text{IN0.D1}, \text{IN1.D1}) \tag{2.123}$$

The computed minimum value is set on `OUT.D1` and emitted along with output event `OUT.E1`.

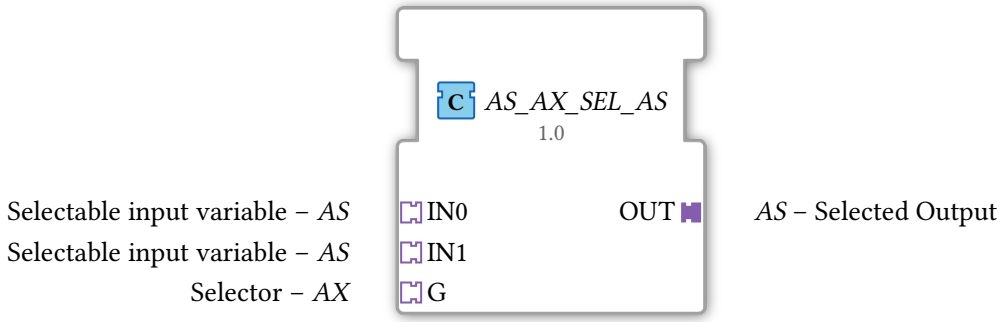
Technical Features

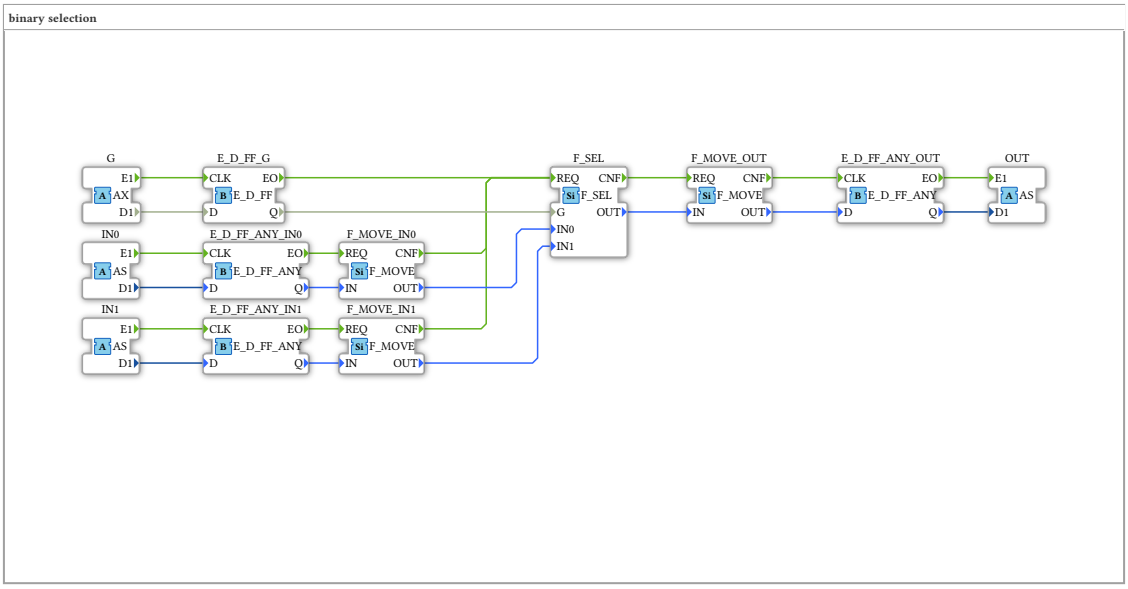
- **Unidirectional Adapter Structure**: Uses standardized `AR` adapter channels for clear signal direction.
- **Event-Driven**: Recalculation and output occur immediately upon arrival of a new measurement at `IN0` or `IN1`.
- **Type Consistency**: Operates internally using the floating-point data type `REAL`.

Conclusion

The `AR_MIN` is a useful selection block for calculating extreme values (minimum selection) in adapter-based IEC 61499 applications.

AS_AX_SEL_AS





(No image available)

• – ○ • – ○ • – ○ •

Introduction

The function block `AS_AX_SEL_AS` is used for binary selection between two input signals provided via adapters. Based on the state of a selection adapter (`G`), either the value of input `IN0` or input `IN1` is passed to output `OUT`.

This function block encapsulates the classic multiplexer function (according to `SEL` from IEC 61131-3) in an event-driven IEC 61499 architecture using unidirectional adapters. This significantly simplifies signal wiring in complex control applications.

Interface Structure

The function block does not have any classic event- or data-based inputs and outputs at the main level. All communication is handled via adapter interfaces.

Event Inputs

No direct event inputs are available. Event processing is encapsulated via the adapter sockets.

Event Outputs

No direct event outputs are available. Event forwarding is encapsulated via the output adapter's plug.

Data Inputs

No direct data inputs available.

Data Outputs

No direct data outputs available.

Adapters

Sockets (Input Interfaces)

- **G** (Type: `adapter::types::unidirectional::AX`):
- **Description:** Selector interface. Determines which input is routed to the output.
- **Event:** `E1` (Trigger for updating the selection signal).
- **Data:** `D1` (Boolean value; determines the active path).
- **IN0** (Type: `adapter::types::unidirectional::AS`):

Description: First selectable data channel (selected when `G` has the value `FALSE`).
Event: `E1` (signals new data to `IN0`).
Data: `D1` (data value of the first channel).
IN1 (Type: `adapter::types::unidirectional::AS`):

Description: Second selectable data channel (selected when G has the value TRUE).
Event: E1 (signals new data to D1). * **Event:** E1 (Signals new data to IN1).

- **Data:** D1 (Data value of the second channel).

Plugs (Output Interfaces)

- **OUT** (Type: adapter::types::unidirectional::AS):
- **Description:** The selected output channel that mirrors the data of the active input.
- **Event:** E1 (Triggered when a new selected value is output).
- **Data:** D1 (The forwarded data value from IN0 or IN1).

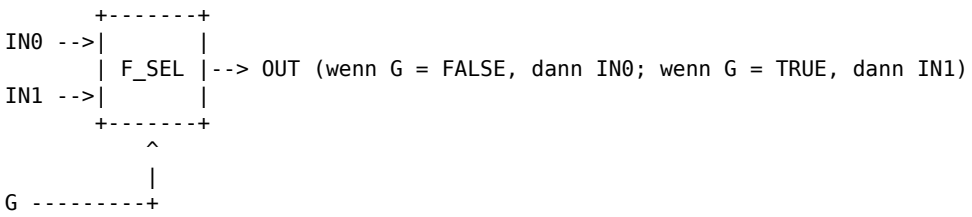
Functionality

Inside the function block is a network of standard function blocks that processes the adapter signals:

1. **Signal Storage (Latch):** The data received via the adapters (G.D1, IN0.D1, IN1.D1) is temporarily stored in event-triggered D flip-flops (E_D_FF and E_D_FF_ANY, respectively) upon the occurrence of the respective event (E1).
2. **Data Conversion & Forwarding:** The cached data of types IN0 and IN1 are transferred to the central selection block via F_MOVE blocks. Internally, these operations are configured for the data type SINT (Short Integer).
3. **Selection Logic (F_SEL):** The block F_SEL (IEC 61131-3 Selection) performs the actual selection:

- If the state of selector G is equal to FALSE, the value of IN0 is passed through.
- If the state of selector G is equal to TRUE, the value of IN1 is passed through.

1. **Output Triggering:** After successful selection, the result is passed via another F_MOVE block to the output flip-flop E_D_FF_ANY_OUT. This flip-flop generates the output event OUT.E1 and places the selected value on OUT.D1.



Technical Features

- **Data Type Specificity:** Although the adapters can be flexibly configured, the internal conversion elements (F_MOVE) are fixed to the data type **SINT**. The module is therefore specifically designed for selecting signed 8-bit integers (short integers).
- **Event-Driven:** Any change or event at the inputs IN0, IN1, or G triggers the chain and updates the output OUT.

State Overview

The following table shows the logical behavior of the function block based on the selection signal G.D1:

State G.D1	Selected Path	Output Value (OUT.D1)	Output Event (OUT.E1)
FALSE	Channel 0 (IN0)	Value of IN0.D1	Triggered on value change/event
TRUE	Channel 1 (IN1)	Value of IN1.D1	Triggered on value change/event

Application Scenarios

- **Operating Mode Switching:** Routing of setpoints (e.g., switching between a manually specified SINT value and an automatic setpoint).
- **Signal Source Selection:** Dynamic selection between two sensor values or parameter configurations distributed throughout the system via standardized unidirectional adapter structures.
- **Channel Bonding:** Reducing wiring effort in 4diac applications by encapsulating data and events in adapters.

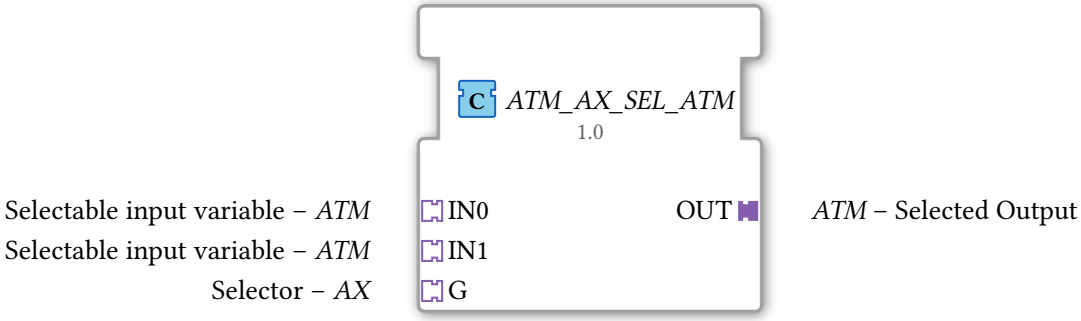
Comparison with Similar Function Blocks

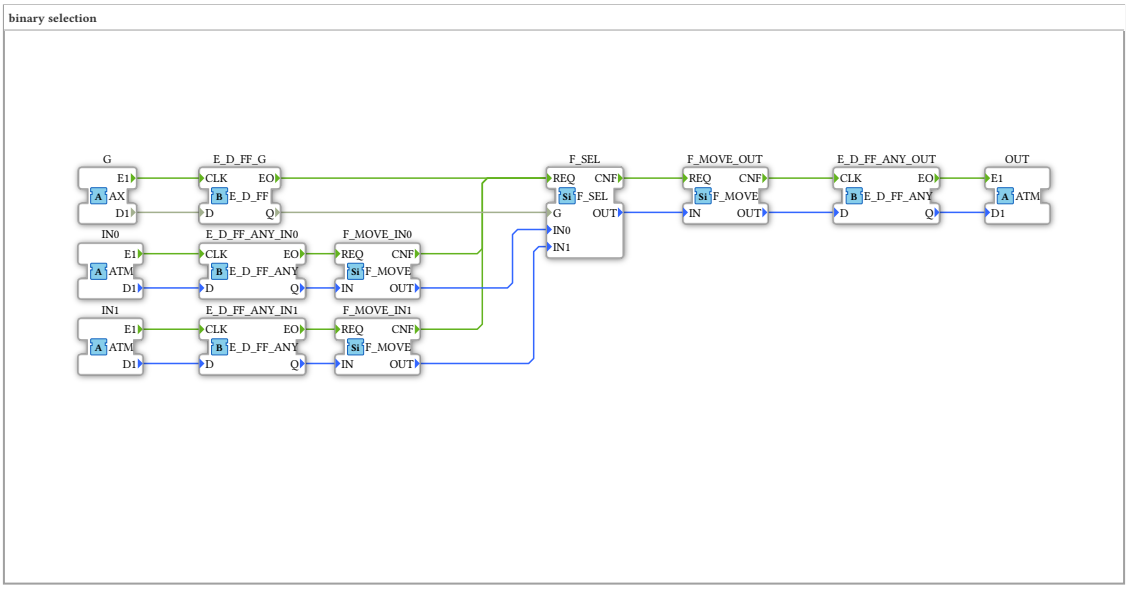
- **`F_SEL` (Standard Selection Function Block):** Works directly with standard data types and requires explicit event connections at the application level. `AS_AX_SEL_AS`, on the other hand, fully encapsulates this logic and uses adapters, resulting in a clearer application diagram.
- **Other adapter-based selectors:** These differ primarily in the data types supported by the adapters (e.g., `INT`, `REAL` instead of `SINT`).

Conclusion

The `AS_AX_SEL_AS` block provides an elegant, clean, and reusable solution for implementing binary signal switching based on unidirectional adapters in the 4diac IDE. The internal use of standardized components ensures deterministic behavior in the event-driven system.

ATM_AX_SEL_ATM





• – ○ • – ○ • – ○ •

Introduction

The function block `ATM_AX_SEL_ATM` serves as a binary selector (multiplexer) for adapter interfaces according to the IEC 61499 standard. Based on the state of a selection adapter (`G` of type `AX`), it allows dynamic selection between two input adapters (`IN0` and `IN1` of type `ATM`) and forwards the selected signal to the output adapter (`OUT` of type `ATM`). The function block operates purely event-driven and is internally designed for processing time-based signals (`TIME`).

Interface Structure

The function block does not have any directly accessible standard event or data pins. All communication is elegantly encapsulated via sockets (input adapters) and plugs (output adapters).

Event Inputs

No direct event inputs are available on the function block body. Event control is implicit via the adapter sockets.

Event Outputs

No direct event outputs are available on the function block body. Event signaling is implicit via the adapter plug.

Data Inputs

No direct data inputs are available on the FB body.

Data Outputs

No direct data outputs are available on the FB body.

Adapters

Sockets (Input Interfaces)

- **IN0** (Type: `adapter::types::unidirectional::ATM`):

The first selectable input adapter (corresponds to the selection in logical `FALSE`).

- **IN1** (Type: `adapter::types::unidirectional::ATM`):

The second selectable input adapter (corresponds to the selection in logical `TRUE`).

- **G** (Type: `adapter::types::unidirectional::AX`):

The selector adapter. The signal applied here determines which of the two inputs is routed to the output.

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::ATM`):

The output adapter that carries the data and events of the currently selected input (`IN0` or `IN1`).

Functionality

The internal structure of the `ATM_AX_SEL_ATM` consists of a network of standardized auxiliary components that ensure reliable and consistent signal transmission:

1. **Event Acquisition and Buffering**:

As soon as an event (`E1`) arrives at one of the sockets (`IN0`, `IN1`, or `G`), it is captured via edge-triggered flip-flops (`E_D_FF_ANY` or `E_D_FF`). The data values (`D1`) are stabilized and passed to internal buffers.

As soon as an event (`E1`) arrives at one of the sockets (`E_D_FF_ANY` or `E_D_FF`), it is captured via edge-triggered flip-flops. 2. **Data Conversion**:

The data from inputs `IN0` and `IN1` are buffered via two `F_MOVE` function blocks (configured to the data type `TIME`) and made available for the selection logic.

1. **Selection**:

The function block `F_SEL` (a standard selection block according to IEC 61131-3) evaluates the state of selector `G`:

- If the signal from `G` is equal to `FALSE`, the value from `IN0` is selected.
- If the signal from `G` is equal to `TRUE`, the value from `IN1` is selected.

1. **Output**:

The selected value is passed via another `F_MOVE` module to the output adapter `OUT`, whereupon the output event (`OUT.E1`) is triggered simultaneously.

Technical Features

- **Adapter Encapsulation**: By using adapters instead of individual pins, the number of connection lines in the overall control diagram is drastically reduced, which improves the readability of the entire system.
- **Data Type Focus**: The function block is internally optimized for the data type `F_MOVE` (time values) through the parameterization of the `F_MOVE` function blocks.

- **Event Integrity**: The use of event flip-flops ensures that no signal changes are lost and the output is only updated when relevant input data or the selector actually changes.

State Overview

Since this is a composite function block network (FB), the function block does not have its own classic ECC (Event Execution Control) state machine. Its behavior is purely data flow and event-driven:

State Selector `G.D1`	Triggering Event	State Output `OUT.D1`	Output Event
`FALSE`	Change to `IN0.E1` or `G.E1`	Value of `IN0.D1`	`OUT.E1`
`TRUE`	Change to `IN1.E1` or `G.E1`	Value of `IN1.D1`	`OUT.E1`

Application Scenarios

- **Process Time Switching**: In industrial or agricultural processes where switching between two different cycle, rinsing, or delay times (e.g., summer/winter operation or recipe A/recipe B) is required.
- **Setpoint Switching**: Dynamic selection of a time-based setpoint from two different sources based on a higher-level operating mode.

–

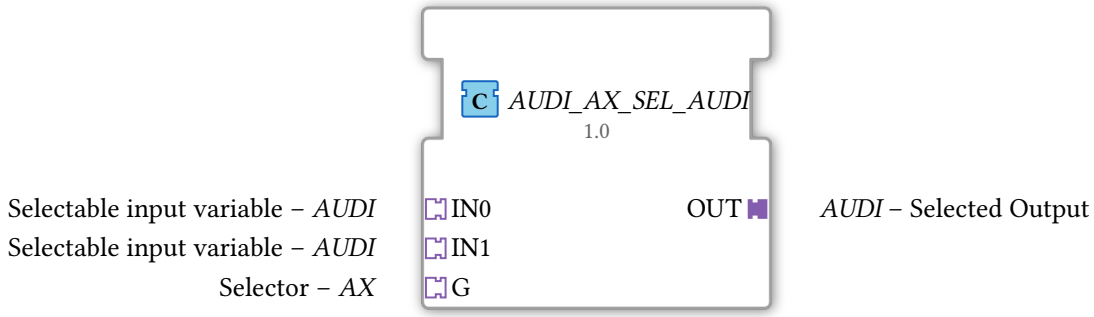
Comparison with Similar Function Blocks

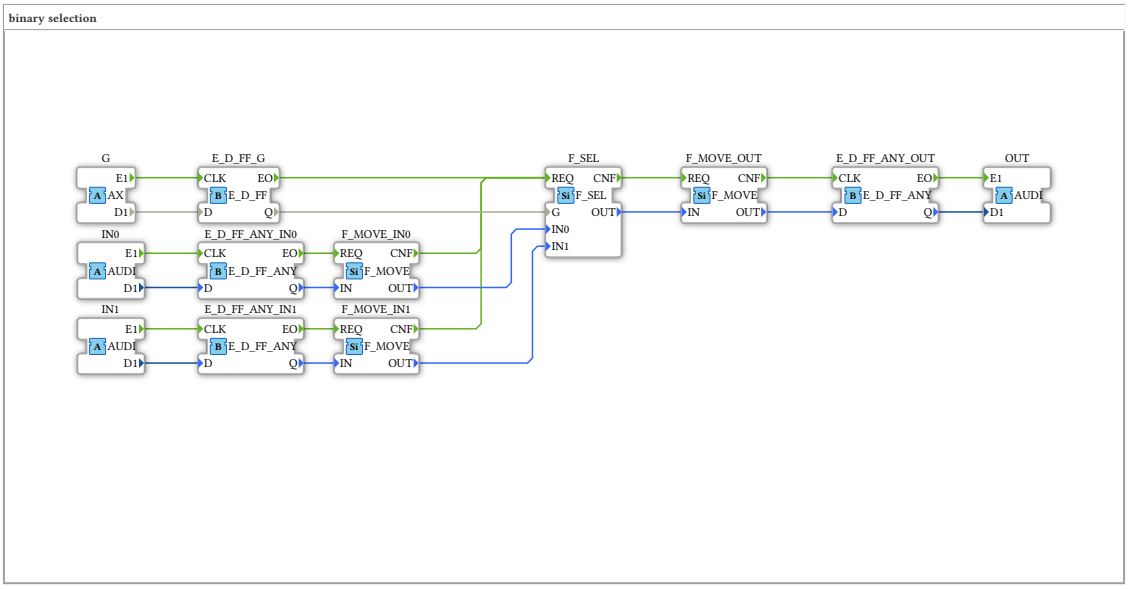
- **Standard `F_SEL`**: The classic `F_SEL` function block operates directly on elementary data types and does not have integrated event control for adapters. `ATM_AX_SEL_ATM` moves this logic to the structured adapter level.
- – **Standard Multiplexer**: Unlike general-purpose multiplexers, which often feature complex bus structures, this component is highly specialized, extremely lightweight, and perfectly matched to the combination of `ATM` and `AX` adapters.

Conclusion

The `ATM_AX_SEL_ATM` is a robust and reusable auxiliary component for modern control architectures according to IEC 61499. It elegantly solves the task of binary signal switching for time-based data streams, leveraging the advantages of adapter technology.

AUDI_AX_SEL_AUDI





(No image available)

• – ○ • – ○ • – ○ •

Introduction

The function block `AUDI_AX_SEL_AUDI` serves as a binary selector (selection switch) for unidirectional communication adapters. Based on the state of a control signal at a selector adapter, it allows the data and event stream between two alternative input adapters to be redirected to a single output adapter.

The block encapsulates the necessary synchronization and conversion logic to ensure a clean separation of events and data flows within IEC 61499 applications.

Interface Structure

Event Inputs

No direct event inputs are defined on the block interface. Event processing is fully encapsulated via the connected adapters.

Event Outputs

No direct event outputs are defined on the function block interface. Event forwarding is handled entirely via the output adapter.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

Sockets (Input Connections)

- **IN0** (Type: `adapter::types::unidirectional::AUDI`):

First selectable input adapter. Data and events arriving here are passed through to the output when the selection signal `G` has the logical state `FALSE` (or `0`).

- **IN1** (Type: `adapter::types::unidirectional::AUDI`):

Second selectable input adapter. Data and events arriving at this adapter are passed to the output when the selection signal `G` has the logical state `TRUE` (or `1`).

- **G** (Type: `adapter::types::unidirectional::AX`):

Selector adapter. The signal on this adapter determines which of the two inputs (`IN0` or `IN1`) is routed to the output.

Plugs (Output Connections)

- **OUT** (Type: `adapter::types::unidirectional::AUDI`):

Output adapter. Outputs the data and the corresponding trigger event of the currently selected input.

Functionality

The internal behavior of the function block is implemented by a network of standard function blocks:

1. **Event and Data Synchronization**:

The signals received via the sockets (`IN0`, `IN1`, `G`) are first processed through edge- or level-triggered D flip-flops (`E_D_FF` and `E_D_FF_ANY`). This ensures that data values (`D1`) and events (`E1`) are processed synchronously.

1. **Data Conversion**:

The synchronized data values from the two inputs are transferred to the selection block via two transfer blocks (`F_MOVE` with the data type `UDINT`).

1. **Selection Control**:

The function block `F_SEL` (IEC 61131-3 Selection) performs the actual logical switching:

- If the control signal of the adapter `G` is `0` (`FALSE`), the value of `IN0` is selected.
- If the control signal of the adapter `G` is `1` (`TRUE`), the value of `IN1` is selected.

1. **Output**:

The selected result is transferred via another transfer block (`F_MOVE_OUT`) to the output flip-flop `E_D_FF_ANY_OUT`. This flip-flop generates the output event `OUT.E1` and makes the selected data value available to `OUT.D1`.

Technical Features

- **UDINT Data Processing**: Although the adapters are of the generic type `AUDI`, the user data (`D1`) is explicitly processed and transmitted internally as `UDINT` (Unsigned Double Integer / 32-bit integer).
- **Event-driven behavior**: Any change to the inputs or the selector triggers a recalculation via the internal coupling network and the output event of the adapter `OUT`.
- **Encapsulation**: By using adapters instead of individual event and data ports, the wiring effort in the overall system diagram is drastically reduced.

State overview

Since the function block is structured as a function block network, it does not have a classic state machine (ECC). The logical behavior can instead be described using the following function table:

State Selector (`G.D1`)	Triggering event	Active data path	Value at output (`OUT.D1`)	Event at output (`OUT.E1`)
`FALSE`	Event on `IN0.E1` or `G.E1`	`IN0` -> `OUT`	Equivalent to `IN0.D1`	Triggered
`TRUE`	Event on `IN1.E1` or `G.E1`	`IN1` -> `OUT`	Equivalent to `IN1.D1`	Triggered

---

Application Scenarios

- **Setpoint Switching**: Switching between an automatic setpoint (e.g., from a PID controller at `IN1`) and a manual setpoint (e.g., from a visualization at `IN0`) via a selection signal at `G`.
- **Sensor Redundancy**: Fail-safe switching between a primary sensor and a backup sensor in case of signal interference.
- **Recipe Control**: Selection of different predefined parameter profiles during operation.
-

Comparison with Similar Function Blocks

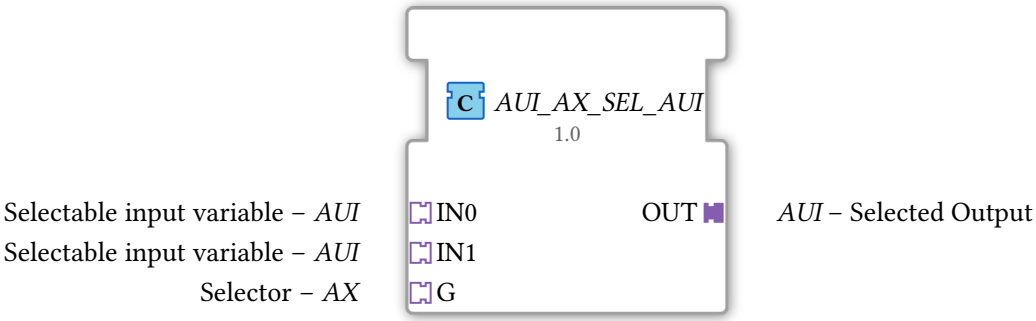
Compared to the standard selection function block `F_SEL` from the IEC 61131-3 library, `AUDI_AX_SEL_AUDI` offers the advantage of direct interaction with structured, unidirectional adapters (`AUDI` and `AX`). Using the standard `F_SEL` function block would require manually splitting all event and data lines of the adapters, connecting them individually, and then reconnecting them, which is error-prone and makes the application diagram cluttered.

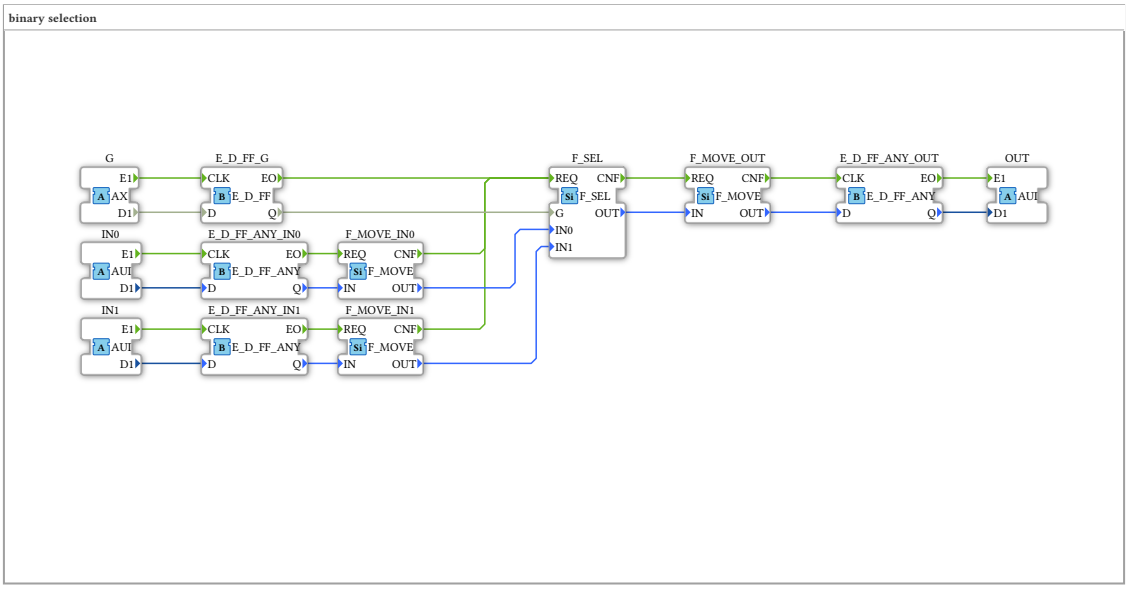
–

Conclusion

The `AUDI_AX_SEL_AUDI` function block offers an elegant, clean, and reusable solution for signal switching in 4diac-based control architectures. Through the consistent use of adapters, he makes a significant contribution to the modularity and clarity of complex control programs.

AUI_AX_SEL_AUI





No image available

• – ○ • – ○ • – ○ •

Introduction

The function block `AUI_AX_SEL_AUI` is a composite function block for 4diac-ide. It performs binary selection between two analog or numeric input signals provided via adapters. Based on the state of a selection adapter, the block switches one of the two input signals to the output.

Interface Structure

The function block does not have any traditional, direct event or data channels on its main interface. All communication is handled via adapters.

Event Inputs
No direct event inputs available.

Event Outputs
No direct event outputs available.

Data Inputs
No direct data inputs available.

Data Outputs
No direct data outputs available.

Adapters

Sockets (Input Adapters)
- **IN0** (Type: `adapter::types::unidirectional::AUI`): Unidirectional adapter for the first selectable input signal. This signal is passed to the output if the selector `G` has the value `FALSE`.
- **IN1** (Type: `adapter::types::unidirectional::AUI`): Unidirectional adapter for the second selectable input signal. This signal is passed to the output when the selector `G` has the value `TRUE`.
- **G** (Type: `adapter::types::unidirectional::AX`): Unidirectional selector adapter. Determines which of the two input signals is passed through.

Plugs (Output Adapters)
- **OUT** (Type: `adapter::types::unidirectional::AUI`): Unidirectional adapter for the selected output signal.

Functionality
Inside the module is a defined network of standard function blocks that processes the adapter signals:

1. **Buffering & Synchronization:**

Events (`E1`) and data (`D1`) arriving via the adapters `IN0`, `IN1`, and `G` are buffered and synchronized using event-data flip-flops (`E_D_FF` and `E_D_FF_ANY`).

1. **Data Conversion:**

The buffered numeric values of the inputs are forwarded to the actual selection block via internal `F_MOVE` blocks (configured to the data type `UINT`).

1. **Selection:**

The standard selection block `F_SEL` (according to IEC 61131-3) performs the actual selection:

- If the value at selector `G` equals `FALSE`, the signal from `IN0` is selected.
- If the value at selector `G` equals `TRUE`, the signal from `IN1` is selected.

1. **Output:**

The selection result is transferred to the output plug `OUT` via a further buffer containing `F_MOVE` and `E_D_FF_ANY`. Simultaneously, the output event at the plug is triggered to inform subsequent function blocks of the new value.

Technical Features
- **Pure Adapter Interface:** This function block is specifically designed for use in modern, adapter-based architectures. This results in a very clean and clear application diagram, as loose data and event lines are bundled within adapters.

** * **Event-driven:** Any change (sending an event) to the inputs `IN0`, `IN1`, or the selector `G` triggers an immediate recalculation and results in an update of the output `OUT`.

- **Typing:** The internal conversion and forwarding blocks primarily work with the data type `UINT`.

State Overview

Since this is a composite function block, it does not have its own state machine (ECC). Its behavior is purely data- and event-driven:

State Selector `G` (value of D1)	Output `OUT` (value of D1)	Triggering of `OUT.E1`
`FALSE` (0)	Corresponds to the value of `IN0`	Yes, on change/event to `IN0` or `G`
`TRUE` (1)	Corresponds to the value of `IN1`	Yes, on change/event to `IN1` or `G`

Application Scenarios

- **Setpoint Switching:** Switching between a manually setpoint (e.g., via an HMI at `IN0`) and an automatic setpoint (e.g., from a calculation logic at `IN1`).
- **Sensor Redundancy:** Switching between two analog sensors (e.g., primary sensor at `IN1` and backup sensor at `IN0`) in case of a fault via a control signal at `G`.
- **Operating Mode Selection:** Selection of different analog control parameters depending on the current machine state.
-

Comparison with Similar Components

The classic standard component `F_SEL` requires direct data lines (e.g., `ANY_ELEMENTARY`) and manual event wiring for calculation triggering (`REQ`/`CNF`).

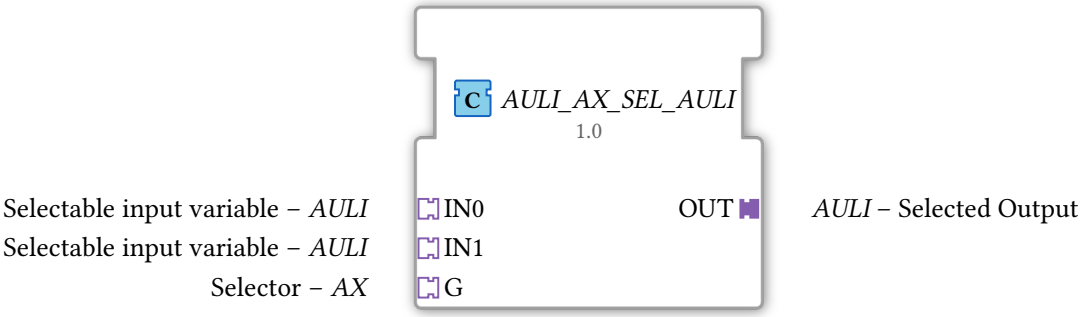
The component `AUI_AX_SEL_AUI` encapsulates this complete logic, including event synchronization and buffering. It is therefore ideally suited for modularized software architectures where signal groups are already available as adapters by default.

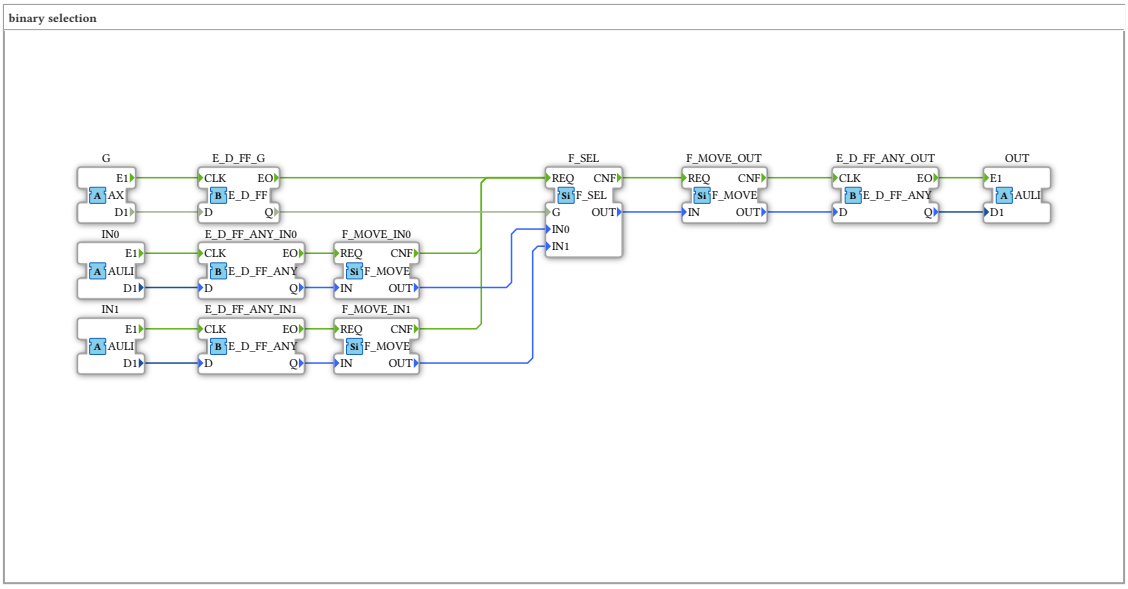
–

Conclusion

`AUI_AX_SEL_AUI` significantly simplifies signal routing in 4diac applications. It combines the proven selection logic of IEC 61131-3 with the structural advantages of event-driven adapters in IEC 61499.

AULI_AX_SEL_AULI





• — ○ • — ○ • — ○ •

Introduction

The function block `AULI_AX_SEL_AULI` serves as a binary selector (multiplexer) in the 4diac framework. Based on a control signal from a selector adapter (`G`), it allows selection between two input signals from adapter inputs (`IN0` and `IN1`) and forwarding the selected signal to an output adapter (`OUT`).

The unique feature of this function block lies in the consistent use of unidirectional adapters for encapsulating events and data, resulting in a clean and modular structure within the higher-level application network.

Interface Structure

Since this function block is entirely adapter-based, it has no direct, traditional event or data channels on its outer boundary. All communication is encapsulated within the adapters.

Event Inputs

No direct event inputs are available. Event control is implicit via the input adapters.

Event Outputs

No direct event outputs are available. Event output is implicit via the output adapter.

Data Inputs

No direct data inputs are available.

Data Outputs

No direct data outputs are available.

Adapters

Sockets (Input Adapters)

- **G** (Type: `adapter::types::unidirectional::AX`):

The selector channel. The signal present here controls which of the two inputs is routed to the output.

* **IN0 (Type: `adapter::types::unidirectional::AULI`)**

The first selectable input channel. This value is passed to the output when the selector `G` is in the state `FALSE` (0).

- **IN1** (Type: `adapter::types::unidirectional::AULI`):

The second selectable input channel. This value is passed to the output when the selector `G` is in the state `TRUE` (1).

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AULI`):

The selected output channel, which carries the signal from the selected input (`IN0` or `IN1`).

Functionality

Internally, the module is based on a network of standard function blocks from IEC 61131 and IEC 61499:

1. **Event Acquisition:** As soon as an event arrives at one of the inputs (`IN0.E1`, `IN1.E1`) or at the selector (`G.E1`), it is intercepted via edge-triggered event flip-flops (`E_D_FF` and `E_D_FF_ANY`, respectively), and the current data value (`D1`) is temporarily stored.
2. **Data Conversion & Transport:** The input data values are transferred to the central selection block via conversion blocks (`F_MOVE` with the data type `ULINT`).
3. **Selection Logic (Multiplexing):** The block `F_SEL` (Standard Selection) evaluates the state of the selector:

- If the value of `G` is equal to `FALSE`, the value of `IN0` is selected.
- If the value of `G` is equal to `TRUE`, the value of `IN1` is selected.

1. **Output:** After successful selection, the result is sent via a further conversion block (`F_MOVE_OUT`) to the output flip-flop (`E_D_FF_ANY_OUT`). This triggers the output event `OUT.E1` and assigns the selected value to `OUT.D1`.

Technical Features

- **Data Type Specificity:** Internally, the selection operates on the data type `ULINT` (Unsigned Long Integer / 64-bit). This makes the function block particularly suitable for processing high-resolution counter values, timestamps, or large ID numbers.
- **Adapter Encapsulation:** By using adapters (`AULI` and `AX`), signal and event lines are bundled. This prevents "spaghetti code" in the system design.
- **Event-Driven:** The function block operates purely event-driven. No cyclic calculations are performed, which conserves resources.

State Overview

The function block does not have its own explicit state chain (Execution Control Chart, ECC) because it is implemented as a pure network (FBNetwork). Its behavior is purely data- and event-flow-driven:

Trigger Event	State Selector (`G.D1`)	Resulting Behavior	Output Event
Event on `G.E1`, `IN0.E1`, or `IN1.E1`	`FALSE`	Value of `IN0.D1` is copied to `OUT.D1`	`OUT.E1`
Event on `G.E1`, `IN0.E1`, or `IN1.E1`	`TRUE`	Value of `IN1.D1` is copied to `OUT.D1`	`OUT.E1`

Application Scenarios

- **Sensor Redundancy:** Switching between a primary sensor (`IN0`) and a backup sensor (`IN1`) based on a status signal (`G`).
- **Setpoint Switching:** Switching between an automatic setpoint (e.g., from a recipe control system) and a manually set user-defined setpoint.
- **Operating Mode Switching:** Routing of different data streams depending on the active system status (e.g., normal operation vs. maintenance mode).

2.7 · Comparison with Similar Components

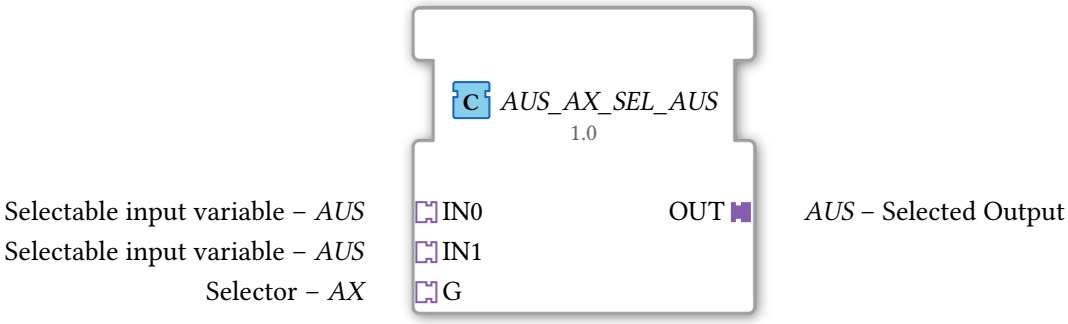
Compared to the classic standard selector block `F_SEL`, `AULI_AX_SEL_AULI` offers the following advantages:

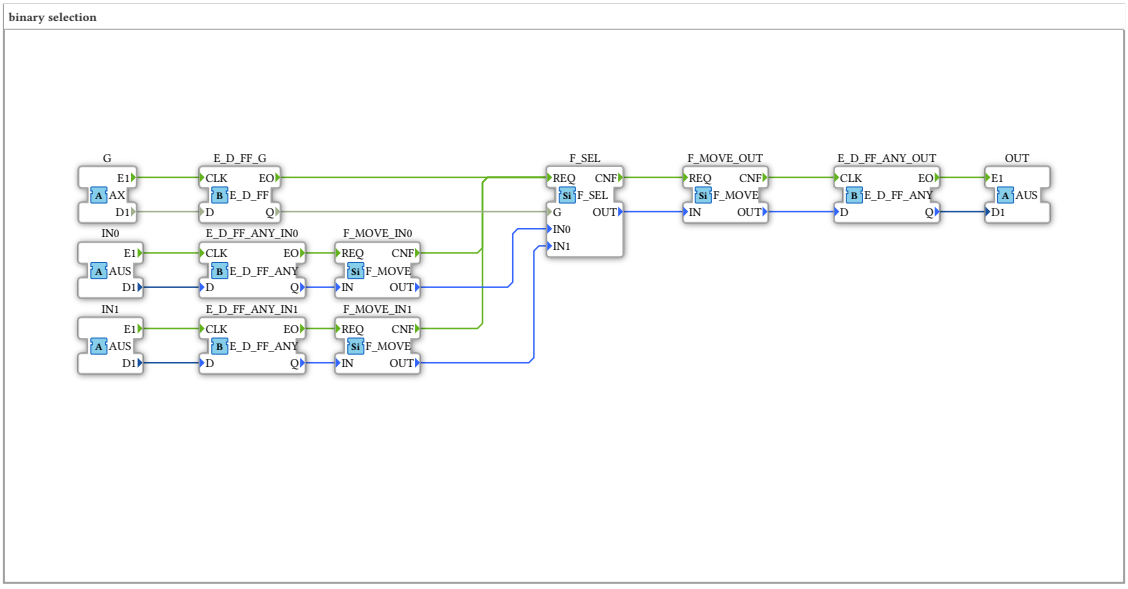
- **No manual wiring of events:** With `F_SEL`, data and event lines must be manually synchronized. Here, this happens automatically within the block through the use of adapters.
- **Increased type safety & clarity:** Connection errors in the higher-level control diagram are minimized because adapters can only be connected to matching counterparts.

Conclusion

The `AULI_AX_SEL_AULI` is a robust, reusable, and event-driven selector block for 64-bit integer values. It is ideally suited for modern, component-based software architectures in IEC 61499, where clarity and standardized interfaces are paramount.

OFF_AX_SEL_OFF





(No image available)

• – ○ • – ○ • – ○ •

The function block **OFF_AX_SEL_OFF** is a binary selector (multiplexer) for IEC 61499 systems. It is used to select between two input adapters (`IN0` and `IN1`) based on the state of a selection signal (selector `G`) and to forward the selected value to the output adapter (`OUT`).

The block uses unidirectional adapter structures, which enables clean encapsulation of data and event streams and simplifies visual routing within the development environment.

Since this function block is implemented as a composite network and is entirely adapter-based, it has no direct, traditional event or data channels at its main level. All communication is bundled within the adapters.

No direct event inputs are available (events are received via the adapter interfaces).
No direct event outputs are available (events are sent via the adapter interfaces).
No direct data inputs are available.
No direct data outputs are available.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Sockets (Input Adapters)

- **G** (Type: `adapter::types::unidirectional::AX`):
- **Description:** Selector signal.
- **Content:** Contains the selection event `E1` and the associated data signal `D1` (usually interpreted as Boolean), which determines which input is enabled.
- **IN0** (Type: `adapter::types::unidirectional::AUS`):
- **Description:** First selectable input. Passed to the output when the selector `G` is in the state `FALSE` (or `0`).
- **Content:** Event `E1` and data signal `D1` (Data Type: `USINT`).
- **Content:** Event `E1` and data signal `D1` (Data Type: `USINT`). * **IN1** (Type: `adapter::types::unidirectional::AUS`):
- **Description:** Second selectable input. It is routed to the output when selector `G` is in state `TRUE` (or `1`).
- **Content:** Event `E1` and data signal `D1` (Data Type: `USINT`).

Plugs (Output Adapters)

- **OUT** (Type: `adapter::types::unidirectional::AUS`):
- **Description:** The selected output. Receives the data and events of the currently enabled input.
- **Content:** Event `E1` and data signal `D1` (Data type: `USINT`).

Functionality

The module encapsulates an internal network that extracts and processes the adapter signals and routes them via standardized IEC 61131-3 and IEC 61499 modules:

1. **Data Acquisition:** As soon as an event (`G`, `IN0`, or `IN1`) arrives at one of the input adapters (`G`, `IN0`, or `IN1`), the corresponding data value (`D1`) is temporarily stored using flip-flops (`E_D_FF` or `E_D_FF_ANY`).
2. **Value Transfer:** The values of inputs `IN0` and `IN1` are transferred to the central selection block via copy operations (`F_MOVE` with the data type `USINT`).
3. **Selection:** The internal block `F_SEL` (standard IEC 61131-3 selector) decides based on the state of `G.D1`:

- If `G.D1` = `0` (or `FALSE`), the value of `IN0` is set to the output.
- If `G.D1` = `1` (or `TRUE`), the value of `IN1` is set to the output.

1. **Output:** The result is passed via another flip-flop (`E_D_FF_ANY_OUT`) to the output adapter `OUT`, triggering the event `OUT.E1` simultaneously with the provision of the data on `OUT.D1`.

- **Event-driven:** The output is updated immediately with every change to the inputs or the selector and assigned a new event.
- **Data Type Specific:** Internally, the processing of the data channels is designed for the type `USINT` (defined by the attributes of the internal `F_MOVE` instances).
- **Adapter Encapsulation:** Facilitates the clean design of application diagrams, as complex data and event pairs are represented as a single connection (bus).

Since this is a composite function block without its own Execution Control Chart (ECC), its behavior is determined purely by the signal flow:

State Selector (`G.D1`)	Trigger Event	Output Activity (`OUT`)
`0` / `FALSE`	Any (`G.E1`, `IN0.E1`, `IN1.E1`)	Returns the value of `IN0.D1`; triggers `OUT.E1`.
`1` / `TRUE`	Any (`G.E1`, `IN0.E1`, `IN1.E1`)	Returns the value of `IN1.D1`; Triggers `OUT.E1`.

- **Setpoint Switching:** Dynamic switching of a target variable (e.g., a speed or stage of type `USINT`) between automatic operation (`IN1`) and manual operation (`IN0`).
- **Recipe Selection:** Selection between two predefined operating modes or parameter sets in a machine via a higher-level controller.
- **Fault Protection:** Rapid switching to a substitute value (`IN0`) if a main signal path is faulty.

2.8 · Comparison with Similar Modules

Compared to the standard selection module `F_SEL` from the IEC 61131-3 library, `AUS_AX_SEL_AUS` offers the decisive advantage of being able to work directly with manufacturer-specific or standardized unidirectional adapters (`AUS` and `AX`). While event and data lines in IEC 61499 have to be laboriously wired individually for `F_SEL`, this is done fully automatically within the module.

Compared to the standard selection module `F_SEL` from the IEC 61131-3 library, `AUS_AX_SEL_AUS` offers the crucial advantage of being able to work directly with manufacturer-specific or standardized unidirectional adapters (`AUS` and `AX`). While event and data lines in `F_SEL` have to be laboriously wired individually according to IEC 61499, this is done fully automatically within the module itself. ## Conclusion

The **AUS_AX_SEL_AUS** function block represents an important bridge for modern, adapter-based IEC 61499 architectures. It combines the proven selection logic of IEC 61131-3 with the advantages of event-driven adapter coupling, thus significantly contributing to the clarity and maintainability of control software.

Technical Features

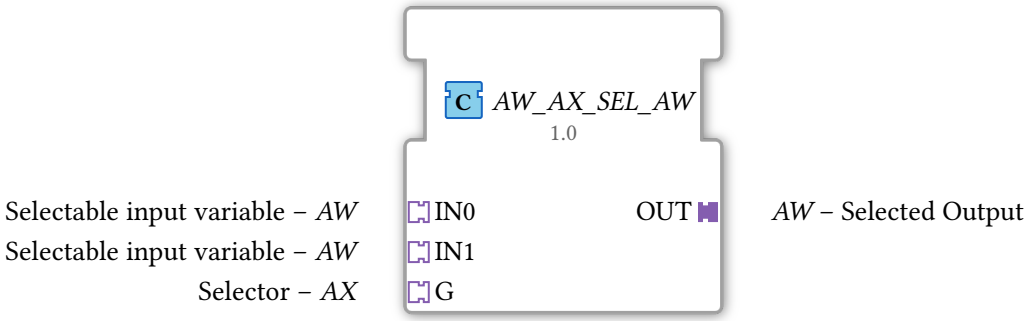
State Overview

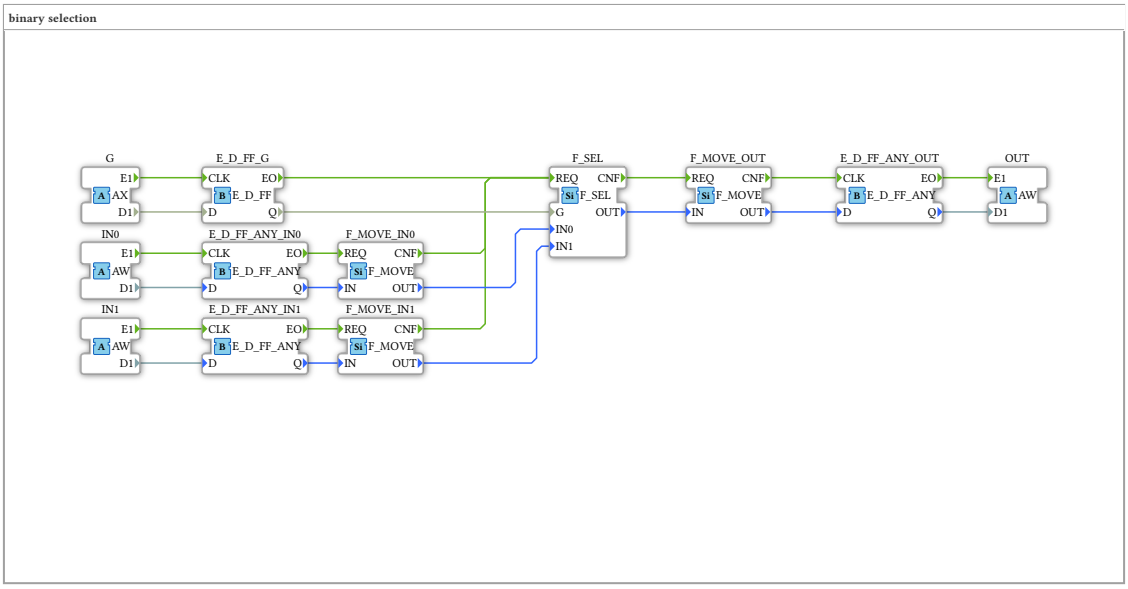
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AW_AX_SEL_AW





(No image available)

• – ○ • – ○ • – ○ •

Introduction

The function block `AW_AX_SEL_AW` is used for binary selection between two analog or word-based input signals via a binary selector. The block is specifically designed for use with adapter-based interfaces in 4diac-ide (IEC 61499). It encapsulates the classic selection logic (`SEL`) and makes it available via unidirectional adapters for `WORD` data (`AW`) and binary control signals (`AX`).

Interface Structure

Event Inputs

There are no direct event inputs at the block level. Event processing is fully encapsulated via the adapter interfaces.

Event Outputs

There are no direct event outputs at the block level. Event output is fully encapsulated via the adapter interfaces.

Data Inputs

There are no direct data inputs at the block level.

Data Outputs

There are no direct data outputs at the block level.

Sockets (Input Interfaces)

- **G** (Type: `adapter::types::unidirectional::AX`):
- Selection input (selector). Determines which of the two inputs is passed through to the output.
- **IN0** (Type: `adapter::types::unidirectional::AW`):
- First selectable input channel (selected when `G` is in state `FALSE` or `0`).
- **IN1** (Type: `adapter::types::unidirectional::AW`):
- Second selectable input channel (selected when `G` is in state `TRUE` or `1`).

Plugs (Output Interfaces)

- **OUT** (Type: `adapter::types::unidirectional::AW`):
- Selected output channel carrying the value of the selected input.

Functionality

The module is internally based on a network of standard function blocks to implement event-driven selection:

1. **Signal Buffering:** Events (`E1`) and data (`D1`) received via the adapters `IN0`, `IN1`, and `G` are buffered using D flip-flops (`E_D_FF` and `E_D_FF_ANY`, respectively). This ensures that the data is consistently available as soon as an event occurs.
2. **Data Forwarding:** The data from the WORD adapters is passed to the central selection block via `F_MOVE` blocks (configured to the data type `WORD`).
3. **Selection Logic (F_SEL):** The block `F_SEL` (standard selection function) evaluates the state of the selector `G`:

- If `G.Q` = `FALSE`, the value of `IN0` is passed to the output.
- If `G.Q` = `TRUE`, the value of `IN1` is passed to the output.

1. **Output Triggering:** Any change to the inputs or the selector triggers the process via the event connections. The selected value is passed from the output buffer `E_D_FF_ANY_OUT` to the output plug `OUT`, and the corresponding output event `OUT.E1` is triggered.

Technical Features

- **Adapter Encapsulation:** By exclusively using adapters, the cabling effort in the higher-level system is drastically reduced, as data and event lines are bundled in a single adapter channel.
- **Unidirectional Data Transmission:** The adapter types used, `AW` (Word Adapter) and `AX` (Bit/Boolean Adapter), are unidirectional, defining a clear signal flow direction.
- **Data Type:** The internal processing of analog values is fixed to the data type `WORD`.

State Overview

Since this is a composite function block (FB) without its own state machine (ECC), its behavior is directly determined by the data and event flow of the internal components:

- **Event on G.E1:** The selector value is updated. The selection logic switches accordingly and updates the output `OUT`.
- **Event on IN0.E1 / IN1.E1:** The input data is updated. If the changed input corresponds to the channel currently selected via `G`, the new value is immediately forwarded to `OUT`, triggering the event `OUT.E1`.

Application Scenarios

- **Setpoint Switching:** Optionally specify a setpoint (e.g., speed, pressure, temperature coded as `WORD`) from an automatic program (`IN1`) or manual operation (`IN0`).
- **Signal Routing:** Dynamic routing of data streams within modular control architectures.
- **Redundant Sensors:** Switching between a primary sensor and a backup sensor in case of signal errors.

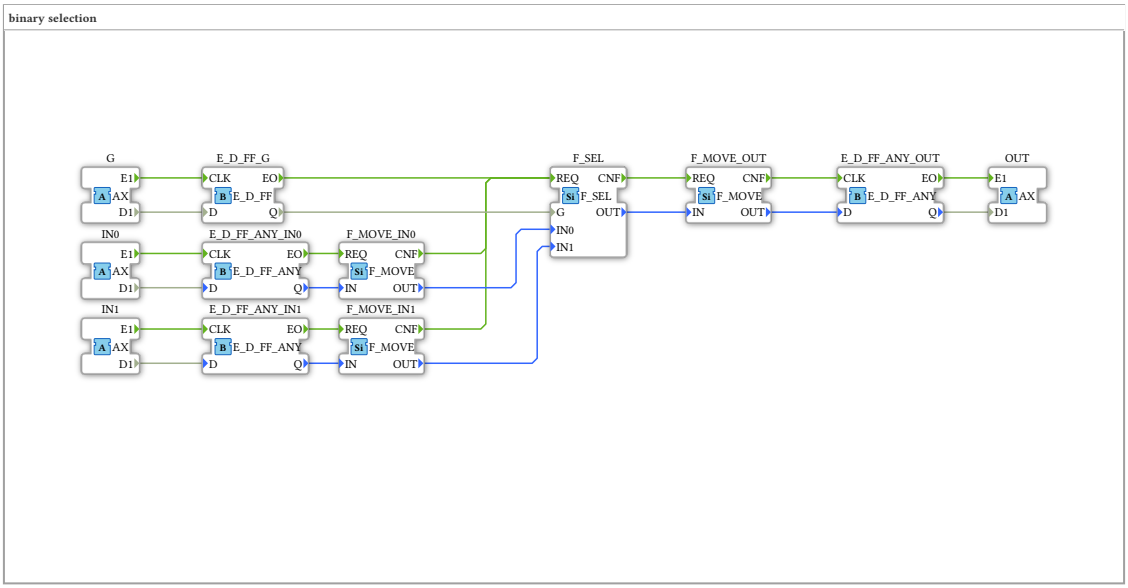
Comparison with Similar Function Blocks

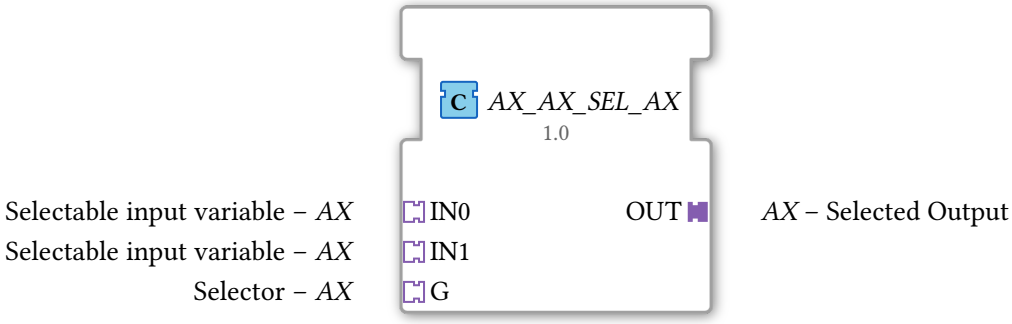
- **Standard `F_SEL` (IEC 61131-3):** The standard selection function block works with direct elementary data types (e.g., WORD, BOOL) and does not have integrated event control for IEC 61499. `AW_AX_SEL_AW` extends this functionality with event coupling and packages it in standardized adapters.
- **Multiplexer (MUX):** A multiplexer allows selection from more than two channels using an integer index (INT). `AW_AX_SEL_AW`, on the other hand, is optimized for fast, binary selection (2 channels via BOOL/AX).

Conclusion

The `AW_AX_SEL_AW` is a robust and reusable auxiliary module for structured programming in IEC 61499. It is ideally suited for applications that rely on a clean interface architecture using adapters and simplifies signal switching in industrial environments.

AX_AX_SEL_AX





• – ○ • – ○ • – ○ •

Introduction

The function block `AX_AX_SEL_AX` is used for binary selection between two Boolean signals transmitted via standardized adapters. It represents an IEC 61499-compliant encapsulation of the classic IEC 61131-3 `SEL` function. By using adapters, both the data values and the associated events are transmitted together, which significantly reduces the wiring effort in control application design.

Interface Structure

Event Inputs

This function block does not have direct event inputs. Event control is handled entirely via the adapters.

Event Outputs

This function block has no direct event outputs. Event output is handled entirely via the adapters.

Data Inputs

This function block has no direct data inputs.

Data Outputs

This function block has no direct data outputs.

Adapters

Sockets (Input Interfaces)

- **IN0** (Type: `adapter::types::unidirectional::AX`):

Selectable input adapter 0. The value applied here is passed to the output when the selection signal G is in the state FALSE.

- **IN1** (Type: adapter::types::unidirectional::AX):

Selectable input adapter 1. The value applied here is passed to the output when the selection signal G is in the state TRUE.

- **G** (Type: adapter::types::unidirectional::AX):

Selector adapter. Determines which of the two inputs (IN0 or IN1) is connected to the output.

Plugs (Output Interfaces)

- **OUT** (Type: adapter::types::unidirectional::AX):

Output adapter that outputs the selected signal (D1) and the corresponding confirmation event (E1).

Functionality

Internally, the AX_AX_SEL_AX is based on an event-driven network that processes the input signals from the sockets and outputs them via the plug:

1. **Event Detection:** Every arrival of an event (E1) at the adapters IN0, IN1, or G triggers an internal edge-triggered flip-flop (E_D_FF or E_D_FF_ANY, respectively).
2. **Data Buffering & Transport:** The data values (D1) from the adapters are forwarded to the central selection block F_SEL via conversion blocks of type F_MOVE (configured to data type BOOL).
3. **Selection Logic:**

- If the signal at adapter G equals FALSE, the internal block F_SEL switches the value of IN0 to the output.
- – … * If the signal at adapter G equals TRUE, the value of IN1 is set to the output.

1. **Output:** The selected value is passed to the flip-flop of the output adapter, which triggers the event OUT.E1 and updates the data point OUT.D1.

Technical Features

- **Typing to BOOL:** The internal data copiers (F_MOVE) are hard-coded to the data type BOOL. Therefore, this specific function block is optimized for selecting Boolean states.
- **Event Synchronization:** The internal use of E_D_FF and E_D_FF_ANY ensures that data changes at the inputs are immediately and consistently transferred to the output without any cycle delay.

State Overview

Since this is a composite function block (FB), its behavior is directly determined by the data and event flow:

State Selector G.D1	Output OUT.D1	Behavior on Event at Inputs
FALSE	Value of IN0.D1	Events at IN0.E1 or G.E1 trigger an update of OUT.E1.
TRUE	Value of IN1.D1	Events at IN1.E1 or G.E1 trigger an update of OUT.E1.

–

Application Scenarios

- **Operating Mode Switching:** Switching between an automatic and a manual command (e.g., opening/closing a valve).
- **Signal Redundancy:** Selection between a primary control signal and a backup signal in case of a failure.

Signal Redundancy: Selection between a primary control signal and a backup signal in case of a failure. *
Compact Application Development: Reduce “spaghetti code” and unwieldy line connections in 4diac-ide by utilizing adapter technology.

Comparison with Similar Components

- **Standard `F_SEL`:** The classic `F_SEL` component requires separate event connections (`REQ` / `CNF`) and data lines for each channel. `AX_AX_SEL_AX` simplifies this by encapsulating it in three input adapters and one output adapter.
- **Multiplexer (MUX):** While a multiplexer can switch between a large number of channels, the `AX_AX_SEL_AX`, as a binary selector, is specialized for exactly two channels, making it more resource-efficient and easier to configure.

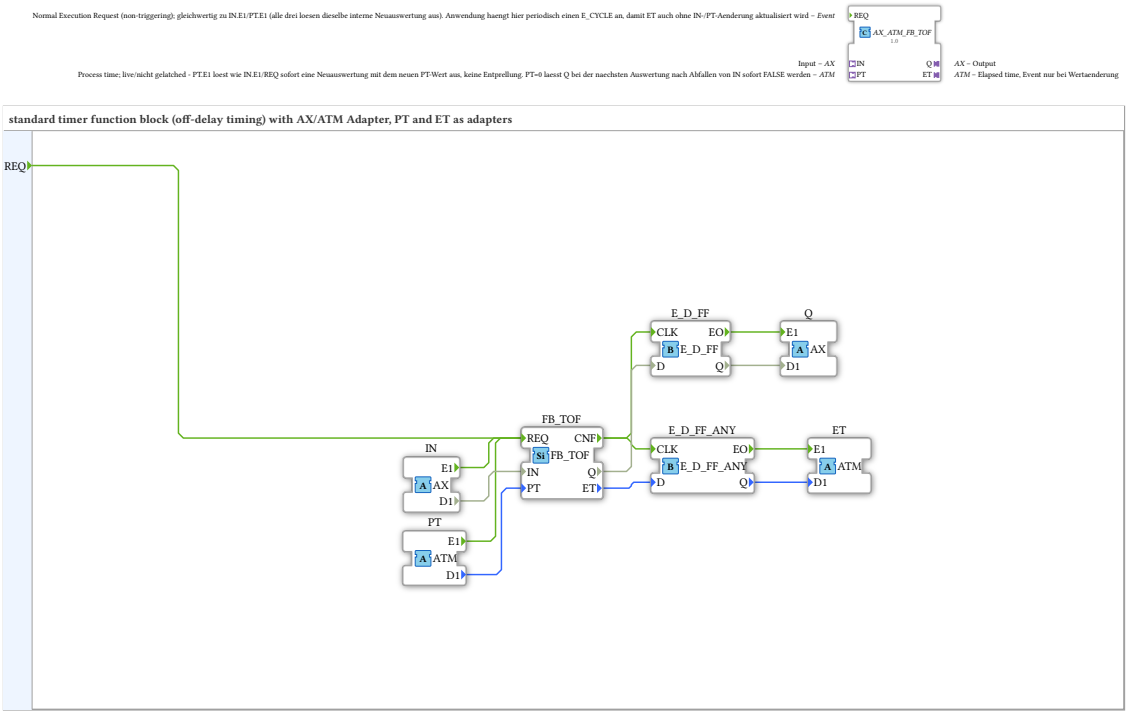
Conclusion

The `AX_AX_SEL_AX` is a practical tool for structured application development in IEC 61499. It combines the proven selection logic of IEC 61131 with the modern advantages of adapter-based signal transmission in the 4diac-ide.

Numeric Index (iec61131-3)

Here you will find all exercises from the `iec61131-3` workspace, listed in ascending order. …

AX_ATM_FB_TOF



• — ○ • — ○ • — ○ •

Introduction

`AX_ATM_FB_TOF` is the adapter-based variant of AX_FB_TOF: the same IEC 61131-3 off-delay timer (`iec61131::timers::FB_TOF`), but with `PT` (preset time) as a socket and `ET` (elapsed time) as a plug — both of type ATM — instead of plain `TIME` data variables. Like AX_ATM_FB_TON, this lets the block be wired directly into adapter-based chains without drawing separate data connections for `PT`/`ET` at every call site.

Interface Structure

Event Inputs

- **REQ** (event, non-triggering): normal execution request. The application typically attaches an `E_CYCLE` here so `ET` keeps updating while the timer is running.

Event Outputs

None of its own; events flow exclusively through the `Q` and `ET` adapter plugs.

Data Inputs

None of its own. All data is carried through the adapter sockets.

Data Outputs

None of its own. All data is carried through the adapter plugs.

Adapters

Direction	Name	Type	Description
Socket (input)	IN	adapter::types::unidirectional::AX	Start signal
Socket (input)	PT	adapter::types::unidirectional::ATM	Preset time
Plug (output)	Q	adapter::types::unidirectional::AX	Output, stays TRUE for PT after IN falls
Plug (output)	ET	adapter::types::unidirectional::ATM	Elapsed time, event only on value change

Functionality

Internally the same standard FB_TOF block used by AX_FB_TOF is wired up, with two added E_D_FF/E_D_FF_ANY buffers:

1. IN.E1, the block's own REQ event, and PT.E1 all trigger FB_TOF.REQ together.
2. FB_TOF.Q runs through an E_D_FF (change detection) onto Q.D1/Q.E1.
3. FB_TOF.ET runs through an E_D_FF_ANY (generic change detection) onto ET.D1/ET.E1 — only on an actual value change, not on every REQ cycle.

Unlike the on-delay timer, IN = TRUE sets output Q to TRUE immediately; only once IN falls back to FALSE does the PT countdown run before Q drops back to FALSE.

Technical Details

- **REQ deliberately stays a plain event, not an adapter connection**, as in AX_ATM_FB_TON — the application controls ET's update rate itself via an externally attached E_CYCLE.
- **ET events only fire on a value change**, the same principle used by ASSEMBLE_AB_FROM_AX.
- **PT is live, not latched — IN.E1, PT.E1, and REQ are equivalent triggers for FB_TOF**, exactly as in AX_ATM_FB_TON. PT.D1 sits as a plain data connection permanently wired to FB_TOF.PT, no buffering/debouncing in the adapter:
  - **PT = 0**: once IN has fallen to FALSE, Q becomes FALSE immediately at the next evaluation (no visible run-on).
  - **PT is decreased during run-on**, such that the already-elapsed ET exceeds the new preset time: Q becomes FALSE immediately.
  - **PT is increased during run-on**: Q correspondingly stays TRUE longer.
  - **Rapid consecutive PT.E1 events** each trigger their own evaluation, no coalescing.
  - **Before the very first PT.E1**, FB_TOF.PT reads TIME's default value T#0s.
- **Identical behavior in AX_ATM_FB_TON and AX_ATM_FB_TP** — all three are structurally wired identically, only the internally wrapped standard block differs.

State Overview

1. **Idle**: IN is FALSE, Q is FALSE, ET is 0.
2. **IN becomes TRUE**: Q becomes TRUE immediately, ET stays 0.
3. **IN falls to FALSE**: the timer starts, ET climbs up to PT, Q stays TRUE.
4. **Elapsed**: ET has reached PT, Q becomes FALSE.

Application Scenarios

- Run-on/coast-down times where an output needs to stay active for a defined period after an input switches off, with visible remaining time via ET.
- Anywhere AX_FB_TOF is already used but PT/ET should flow through adapter-based chains.

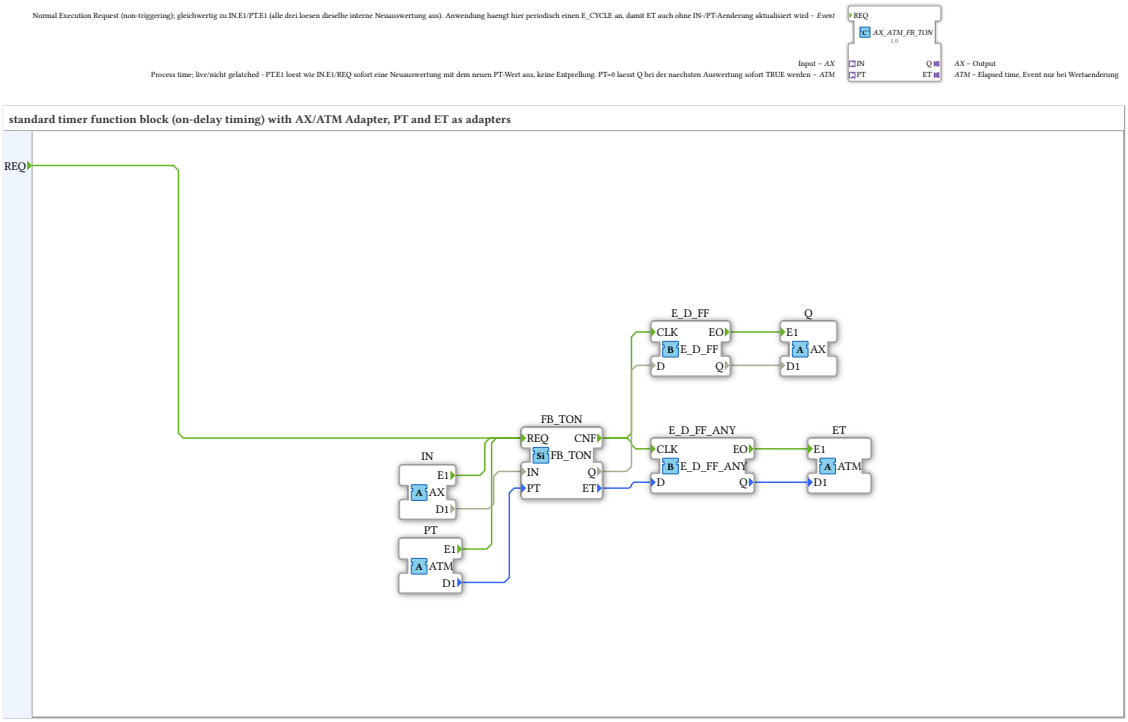
⚖️ Comparison with Similar Blocks

- AX_FB_TOF: identical timing behavior, but PT/ET as plain TIME data variables rather than adapters.
- AX_TOF: the events variant with no ET output.
- AX_ATM_FB_TON / AX_ATM_FB_TP: the same adapter extension for on-delay and pulse timing instead of off-delay.

Conclusion

AX_ATM_FB_TOF carries AX_FB_TOF's proven timing behavior into a fully adapter-based interface, with change detection on the ET output to avoid unnecessary events.

AX_ATM_FB_TON



• — ○ • — ○ • — ○ •

Introduction

AX_ATM_FB_TON is the adapter-based variant of AX_FB_TON: the same IEC 61131-3 on-delay timer (iec61131::timers::FB_TON), but with PT (preset time) as a socket and ET (elapsed time) as a plug — both of type ATM — instead of plain TIME data variables. This lets the block be wired directly into adapter-based chains (e.g. behind an AR_SUBSCRIBE_1 delivering a REAL seconds value written via OPC-UA, converted to TIME) without drawing separate data connections for PT/ET at every call site.

Interface Structure

Event Inputs

- **REQ** (event, non-triggering): normal execution request. The application typically attaches an E_CYCLE here so ET keeps updating while the timer is running — without an external E_CYCLE, ET would stay frozen between IN/PT events.

Event Outputs

None of its own; events flow exclusively through the Q and ET adapter plugs.

Data Inputs

None of its own. All data is carried through the adapter sockets.

Data Outputs

None of its own. All data is carried through the adapter plugs.

Adapters

Direction	Name	Type	Description
Socket (input)	IN	adapter::types::unidirectional::AX	Start signal
Socket (input)	PT	adapter::types::unidirectional::ATM	Preset time
Plug (output)	Q	adapter::types::unidirectional::AX	Output, TRUE once IN has been present for PT
Plug (output)	ET	adapter::types::unidirectional::ATM	Elapsed time, event only on value change

Functionality

Internally the same standard FB_TON block used by AX_FB_TON is wired up, with two added E_D_FF/E_D_FF_ANY buffers:

1. IN.E1, the block's own REQ event, and PT.E1 all trigger FB_TON.REQ together — each of the three sources (input changes, preset time changes, or an external cyclic nudge) should recompute the current time base.
2. FB_TON.Q runs through an E_D_FF (change detection) onto Q.D1/Q.E1 — the Q plug only fires its event when the boolean output actually changes.
3. FB_TON.ET runs through an E_D_FF_ANY (generic change detection) onto ET.D1/ET.E1 — likewise only on an actual value change, not on every REQ cycle.

Technical Details

- **REQ deliberately stays a plain event, not an adapter connection.** This lets the application decide the update rate for ET itself (e.g. via E_CYCLE), instead of the block imposing a fixed refresh rate.
- **ET events only fire on a value change**, the same principle used by ASSEMBLE_AB_FROM_AX and AB_AX_SEL_AB — prevents event flooding on every E_CYCLE tick when the time value (depending on resolution) hasn't actually changed.
- **PT is live, not latched — IN.E1, PT.E1, and REQ are equivalent triggers for FB_TON.** PT.D1 sits as a plain data connection permanently wired to FB_TON.PT; the adapter does not buffer or debounce PT. FB_TON itself doesn't distinguish which of the three events (IN.E1, PT.E1, REQ) triggered the re-evaluation — it always evaluates the *current* PT against the *current* ET. A preset time changed at runtime is therefore not frozen at IN's rising edge; it takes effect immediately at the next evaluation:
  - **PT = 0**: Q becomes TRUE immediately at the next evaluation, since ET (≥ 0) has already reached/exceeded the preset time.
  - **PT is decreased while timing is in progress**, such that the already-elapsed ET now exceeds the new, smaller preset time: Q becomes TRUE immediately at the next evaluation, instead of waiting for the originally larger preset time.
  - **PT is increased while timing is in progress**: Q correspondingly stays FALSE longer.
  - **Rapid consecutive PT.E1 events are not coalesced** — each individual event triggers its own FB_TON.REQ evaluation; there is no debounce logic in the adapter.
  - **Before the very first PT.E1**, FB_TON.PT reads TIME's default value T#0s, not a project-specific default — equivalent to the PT = 0 case above if IN becomes TRUE before that.
- **Identical behavior in AX_ATM_FB_TOF and AX_ATM_FB_TP.** All three blocks are structurally wired identically (the same IN.E1/PT.E1/REQ → REQ pattern, the same E_D_FF/E_D_FF_ANY change-detection pattern for Q/ET) — only the internally wrapped standard block (FB_TON/FB_TOF/FB_TP) differs. The PT behavior described above applies equally to all three; see their respective pages for the timer-specific PT = 0 edge cases.
- **The adapter::events::unidirectional::timers family (AX_TON etc.) has no ET output at all.** If the remaining/elapsed time is needed (e.g. for a countdown in the Visu), this iec61131-3 family is required, not the simpler events family.

State Overview

Like AX_FB_TON: no ECC of its own, a pure composite wiring pattern around the standard FB_TON.

1. **Idle**: IN is FALSE, Q is FALSE, ET is 0.
2. **Timing**: IN is TRUE, the timer runs, ET climbs up to PT, Q stays FALSE.
3. **Triggered**: ET has reached PT, Q becomes TRUE.

4. **Reset**: `IN` goes FALSE, `Q` becomes FALSE, `ET` becomes `0`.

Application Scenarios

- **Countdown display via OPC-UA**: feed `PT` from a subscribed REAL seconds value (e.g. `Override.Dauer`), forward `ET` (or `PT - ET` for the remaining time) to the Visu via a publish block.
- Anywhere `AX_FB_TON` is already used but `PT`/`ET` should flow through adapter-based chains instead of individual data connections.

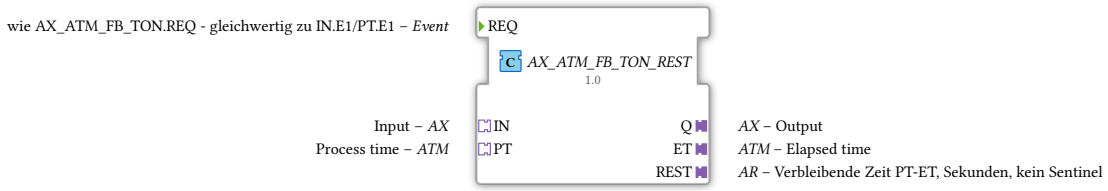
⚖️ Comparison with Similar Blocks

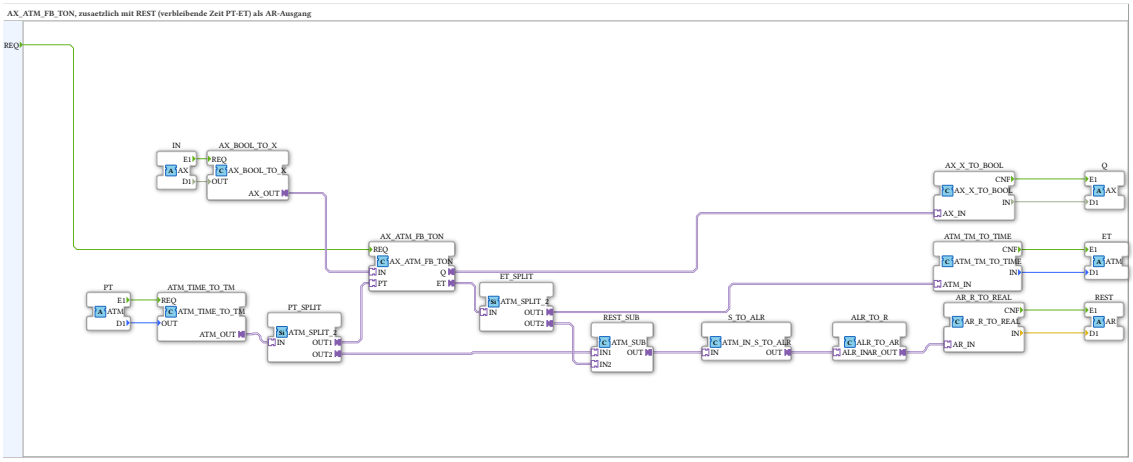
- **AX_FB_TON**: identical timing behavior, but `PT`/`ET` as plain `TIME` data variables rather than adapters.
- **AX_TON**: the `events` variant with no `ET` output at all — suitable when only the timing result (`Q`) is needed, not the elapsed time.
- **AX_ATM_FB_TOF** / **AX_ATM_FB_TP**: the same adapter extension for off-delay and pulse timing instead of on-delay.

Conclusion

`AX_ATM_FB_TON` carries `AX_FB_TON`'s proven timing behavior into a fully adapter-based interface, so both the preset time and the elapsed time can flow through adapter-based networks without extra data connections — including change detection on the `ET` output to avoid unnecessary events.

AX_ATM_FB_TON_REST





• — ○ • — ○ • — ○ •

Introduction

`AX_ATM_FB_TON_REST` wraps AX_ATM_FB_TON with a third adapter plug, `REST`, carrying the remaining time (`PT - ET`) as a REAL seconds value. Several applications need not just "has the timer elapsed" (`Q`) or "how much time has passed" (`ET`), but "how much time is left" for a countdown display — computing that from `PT`/`ET` externally means repeating the same adapter chain (`ATM_SUB` → `ATM_IN_S_TO_ALR` → `ALR_TO_AR`) at every call site. This block does it once, internally, so the caller gets `REST` as a single ready-to-publish adapter plug.

Interface Structure

Event Inputs

- **REQ** (event, non-triggering): same role as on `AX_ATM_FB_TON` — the application typically attaches an `E_CYCLE` here so `ET`/`REST` keep updating while the timer is running.

Event Outputs

None of its own; events flow exclusively through the `Q`, `ET`, and `REST` adapter plugs.

Data Inputs

None of its own. All data is carried through the adapter sockets.

Data Outputs

None of its own. All data is carried through the adapter plugs.

Adapters

Direction	Name	Type	Description
Socket (input)	`IN`	`adapter::types::unidirectional::AX`	Start signal
Socket (input)	`PT`	`adapter::types::unidirectional::ATM`	Preset time
Plug (output)	`Q`	`adapter::types::unidirectional::AX`	Output, TRUE once `IN` has been present for `PT`
Plug (output)	`ET`	`adapter::types::unidirectional::ATM`	Elapsed time, event only on value change
Plug (output)	`REST`	`adapter::types::unidirectional::AR`	Remaining time `PT - ET` in seconds, no sentinel value

Functionality

Internally, a plain `AX_ATM_FB_TON` instance does the actual timing. Around it:

1. `PT` is split (`ATM_SPLIT_2`) so its value reaches both the wrapped timer's own `PT` and an `ATM_SUB` computing `PT - ET`.
2. The wrapped timer's `ET` is likewise split so it reaches both this block's own `ET` plug and the same `ATM_SUB` as the subtrahend.
3. `ATM_SUB.OUT` (an `ATM`, i.e. `TIME`) is converted to seconds via `ATM_IN_S_TO_ALR` → `ALR_TO_AR`, producing the `REST` value.
4. `Q` is passed straight through from the wrapped timer.

Why plain Socket/Plug forwarding doesn't work here, and what the block does instead: a Socket can only be *fed by* a Plug, and a Plug can only *feed* a Socket — a bare `<Connection Source="IN" Destination="InnerTimer.IN"/>` between this block's own `IN` Socket and the wrapped timer's `IN` Socket (both Sockets) is not a legal adapter connection, and the symmetric problem exists for `Q` (both Plugs). Because `AX_ATM_FB_TON_REST` is itself a `FBType` (not a `SubAppType`), reading a Socket's current value via `IN.D1`/`IN.E1` dot-access **is** legal here — the fix is therefore to bridge through the plain-variable pair at each boundary: AX_BOOL_TO_X reads `IN.D1`/`IN.E1` and re-emits it as a Plug feeding the wrapped timer's `IN` Socket; symmetrically, AX_X_TO_BOOL reads the wrapped timer's `Q` Plug and re-emits it onto this block's own `Q` Plug. The same pattern applies to `PT`/`ET` (`ATM_TIME_TO_TM`/`ATM_TM_TO_TIME`), each pair bridging one direction in and one direction out because the wrapped timer's socket and plug both need a bound value. `REST` only needs the outbound half of that pattern — it is computed purely internally and never fed from outside — so just AR_R_TO_REAL is used: it reads the internally computed `AR` (via its own `AR_IN` Socket, fed by `ALR_TO_R`'s Plug) and re-emits the plain `REAL` onto `REST`'s own Plug pins. The same Socket-can-only-be-fed-by-a-Plug rule this works around applies to any adapter type, not just `AX`/`ATM`/`AR`.

Technical Details

- **`REST` carries no sentinel convention (no "−1 = inactive").** A caller that needs one (e.g. "−1 while the timer isn't running") gates `REST` itself with an AR_AX_SEL_AR downstream, selecting between `REST` and a constant based on its own "active" condition — this block only ever reports the raw `PT - ET` difference.
- **`REST` updates on the same resolution as `ET`**, i.e. only on an actual value change (inherited from `AX_ATM_FB_TON`'s `E_D_FF_ANY` on `ET`), not on every `REQ`/`E_CYCLE` tick.
- **All of `AX_ATM_FB_TON`'s `PT` semantics apply unchanged** — `PT` is live, not latched; `PT = 0` makes `Q` (and thus a `REST` of 0) become true/reached at the very next evaluation. See AX_ATM_FB_TON for the full breakdown of these edge cases.

State Overview

Same four states as `AX_ATM_FB_TON`, with `REST` following `PT - ET` throughout:

1. **Idle**: `IN` is FALSE, `Q` is FALSE, `ET` is 0, `REST` equals `PT` converted to seconds.
2. **Triggered**: `ET` has reached `PT`, `REST` reaches 0, `Q` becomes TRUE.
3. **Reset**: `IN` goes FALSE, `Q` becomes FALSE, `ET` becomes 0, `REST` returns to `PT`.

Application Scenarios

- **Countdown display via OPC-UA**: feed `PT` from a subscribed REAL seconds value, publish `REST` directly to the Visu — no external `ATM_SUB/ATM_IN_S_TO_ALR/ALR_TO_AR` chain needed at the call site.
- Any place that today wires `AX_ATM_FB_TON` alongside its own copy of the `ATM_SUB/ATM_IN_S_TO_ALR/ALR_TO_AR` chain to get a remaining-time value — this block replaces all four blocks with one.

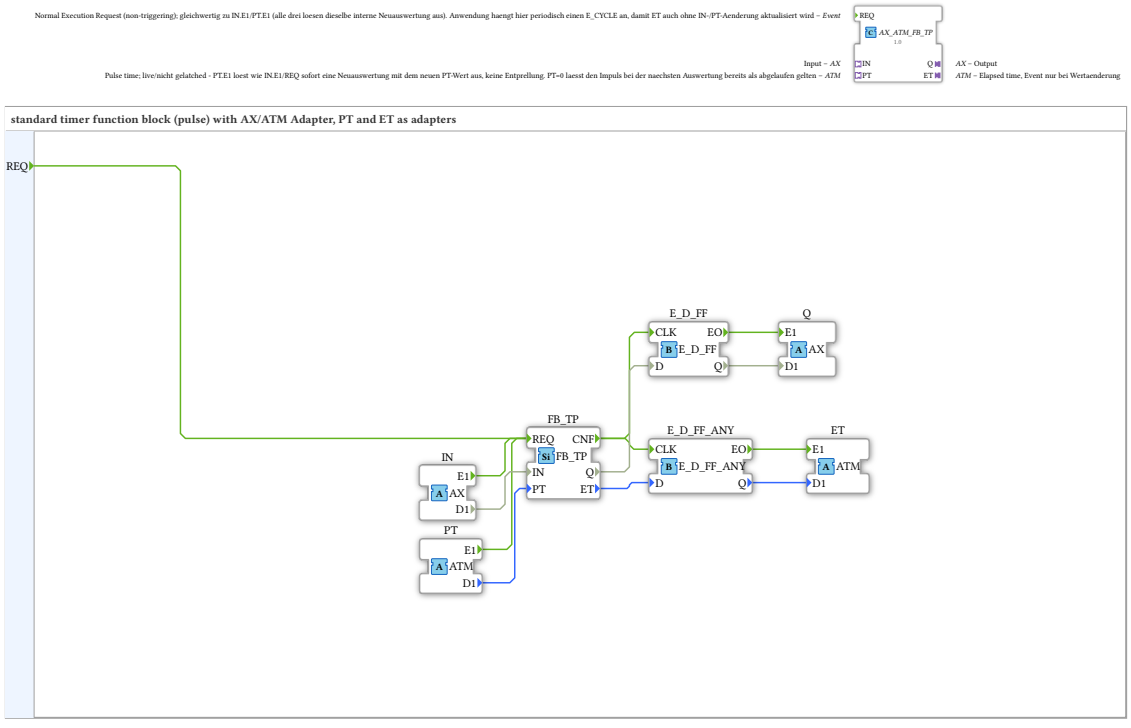
⚖️ Comparison with Similar Blocks

- AX_ATM_FB_TON: identical timing behavior and interface, minus the `REST` plug — use it when only `Q`/`ET` are needed.
- AR_AX_SEL_AR: not a replacement, but the companion block for adding a sentinel value (e.g. "−1 = inactive") on top of this block's raw `REST` output.

Conclusion

`AX_ATM_FB_TON_REST` folds the recurring "on-delay timer with a remaining-time readout" pattern — previously four separately-wired blocks per call site — into a single adapter-based block, trading a slightly more complex internal wiring (Socket/Plug bridging at every boundary) for a much simpler call site.

AX_ATM_FB_TP



• — ○ • — ○ • — ○ •

Introduction

`AX_ATM_FB_TP` is the adapter-based variant of AX_FB_TP: the same IEC 61131-3 pulse timer (`iec61131::timers::FB_TP`), but with `PT` (pulse time) as a socket and `ET` (elapsed time) as a plug — both of type ATM — instead of plain `TIME` data variables. Like AX_ATM_FB_TON, this lets the block be wired directly into adapter-based chains without drawing separate data connections for `PT`/`ET` at every call site.

Interface Structure

Event Inputs

- **REQ** (event, non-triggering): normal execution request. The application typically attaches an `E_CYCLE` here so `ET` keeps updating while the pulse is running.

Event Outputs

None of its own; events flow exclusively through the `Q` and `ET` adapter plugs.

Data Inputs

None of its own. All data is carried through the adapter sockets.

Data Outputs

None of its own. All data is carried through the adapter plugs.

Adapters

Direction	Name	Type	Description
Socket (input)	`IN`	`adapter::types::unidirectional::AX`	Start signal
Socket (input)	`PT`	`adapter::types::unidirectional::ATM`	Pulse time
Plug (output)	`Q`	`adapter::types::unidirectional::AX`	Output, TRUE for duration `PT` after a rising edge on `IN`
Plug (output)	`ET`	`adapter::types::unidirectional::ATM`	Elapsed time, event only on value change

Functionality

Internally the same standard `FB_TP` block used by `AX_FB_TP` is wired up, with two added `E_D_FF`/`E_D_FF_ANY` buffers:

1. `IN.E1`, the block's own `REQ` event, and `PT.E1` all trigger `FB_TP.REQ` together.
2. `FB_TP.Q` runs through an `E_D_FF` (change detection) onto `Q.D1`/`Q.E1`.
3. `FB_TP.ET` runs through an `E_D_FF_ANY` (generic change detection) onto `ET.D1`/`ET.E1` — only on an actual value change, not on every `REQ` cycle.

A rising edge on `IN` immediately sets `Q` to TRUE and starts the countdown; `Q` stays TRUE for the full duration `PT` regardless of whether `IN` falls back to FALSE in the meantime (a true pulse, not retriggerable).

Technical Details

- **`REQ` deliberately stays a plain event, not an adapter connection**, as in AX_ATM_FB_TON — the application controls `ET`'s update rate itself via an externally attached `E_CYCLE`.
- **`ET` events only fire on a value change**, the same principle used by ASSEMBLE_AB_FROM_AX.
- **`PT` is live, not latched — `IN.E1`, `PT.E1`, and `REQ` are equivalent triggers for `FB_TP`**, exactly as in AX_ATM_FB_TON. `PT.D1` sits as a plain data connection permanently wired to `FB_TP.PT`, no buffering/debouncing in the adapter:
  - **`PT = 0`**: the pulse is already considered elapsed at the next evaluation — from an observer's point of view, `Q` may have effectively no visible TRUE phase.
  - **`PT` is decreased while a pulse is running**, such that the already-elapsed `ET` exceeds the new preset time: `Q` becomes FALSE immediately.
  - **`PT` is increased while a pulse is running**: `Q` correspondingly stays TRUE longer.
  - **Rapid consecutive `PT.E1` events** each trigger their own evaluation, no coalescing.
  - **Before the very first `PT.E1`**, `FB_TP.PT` reads `TIME`'s default value `T#0s`.
- **Identical behavior in AX_ATM_FB_TON and AX_ATM_FB_TOF** — all three are structurally wired identically, only the internally wrapped standard block differs.

State Overview

1. **Idle**: `IN` is FALSE, `Q` is FALSE, `ET` is 0.
2. **Rising edge on `IN`**: `Q` becomes TRUE immediately, `ET` starts running.
3. **Pulse active**: `ET` climbs up to `PT`, `Q` stays TRUE — regardless of `IN`'s further behavior.

4. **Elapsed**: ET has reached PT, Q becomes FALSE.

Application Scenarios

- Fixed, non-retriggerable pulse lengths (e.g. a valve's minimum on-time) with visible time progress via ET.
- Anywhere AX_FB_TP is already used but PT/ET should flow through adapter-based chains.

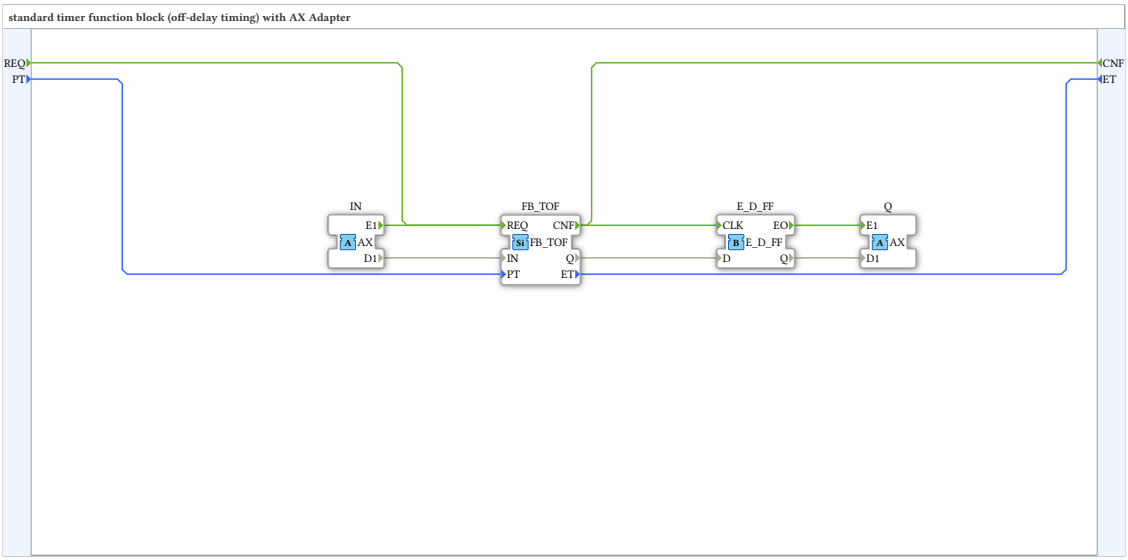
⚖️ Comparison with Similar Blocks

- **AX_FB_TP**: identical timing behavior, but PT/ET as plain TIME data variables rather than adapters.
- **AX_TP**: the events variant with no ET output.
- **AX_ATM_FB_TON** / **AX_ATM_FB_TOF**: the same adapter extension for on-delay and off-delay timing instead of a pulse.

Conclusion

AX_ATM_FB_TP carries AX_FB_TP's proven timing behavior into a fully adapter-based interface, with change detection on the ET output to avoid unnecessary events.

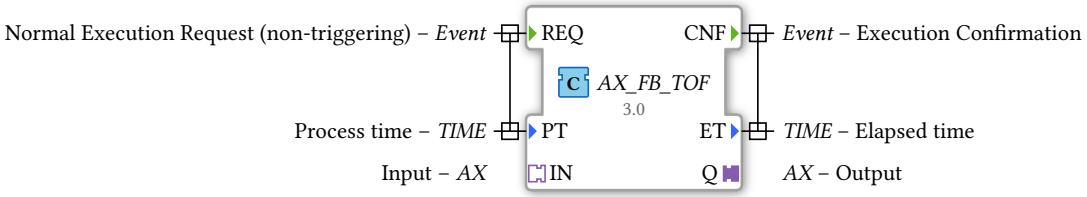
AX_FB_TOF



• – ○ • – ○ • – ○ •

Introduction

Important note: This function block only functions correctly if it is called cyclically. AX_FB_TOF is a standardized timer function block according to IEC 61131-3 that implements off-delay timing. The block maintains the output signal for a defined duration after the input signal is switched off.



Interface Structure

Event Inputs

- **REQ** (Event): Normal Execution Request (takes the value from PT); Calls the function block (must be called cyclically)

Event Outputs

- **CNF** (Event): Execution Confirmation (with `ET`)

Data Inputs

- **PT** (TIME): Process time - defines the duration of the turn-off delay

Data Outputs

- **ET** (TIME): Elapsed time

Adapters

- **IN** (Socket): Receives input events and data (Type: `adapter::types::unidirectional::AX`)
- **Q** (Plug): Sends output events and data (Type: `adapter::types::unidirectional::AX`)

Functionality

The AX_FB_TOF block functions as a turn-off delay timer:

1. When the input on the IN adapter becomes TRUE, the output on the Q adapter immediately switches to TRUE.
2. When the input at the IN adapter changes from TRUE to FALSE, the internal timer starts.
3. The output at the Q adapter remains TRUE while the timer is running and PT has not yet been reached.
4. After PT has elapsed, the output at the Q adapter changes to FALSE.
5. The data output ET displays the time elapsed since the input signal dropped.
6. If the input at the IN adapter becomes TRUE again while the timer is running, the timer is reset and Q remains TRUE.

Technical Features

- Implemented based on the IEC 61131-3 TOF standard
- Uses unidirectional adapters for integration into modern control concepts
- Provides an ET output for monitoring the remaining delay time
- Robust implementation for reliable overrun controllers

State Overview

1. **Inactive**: IN is FALSE, Q is FALSE, ET is 0.
2. **Active**: IN is TRUE, Q is TRUE, ET is 0.
3. **Delay Phase**: IN changes to FALSE, timer is running, Q remains TRUE, ET increases.
4. **Expired**: ET reaches PT, Q changes to FALSE.

Application Scenarios

- Debouncing controls (e.g., fan control after a device is switched off)
- Debouncing of signals that might drop briefly
- Ensuring minimum runtimes after a stop command

⚖️ Comparison with similar function blocks

Compared to event-based timers such as AX_TOF, this function block follows classic PLC behavior according to IEC 61131-3 and additionally offers the ET status output.

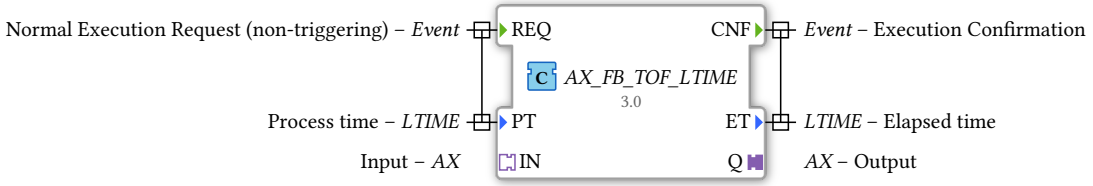
🛠️ Related Exercises

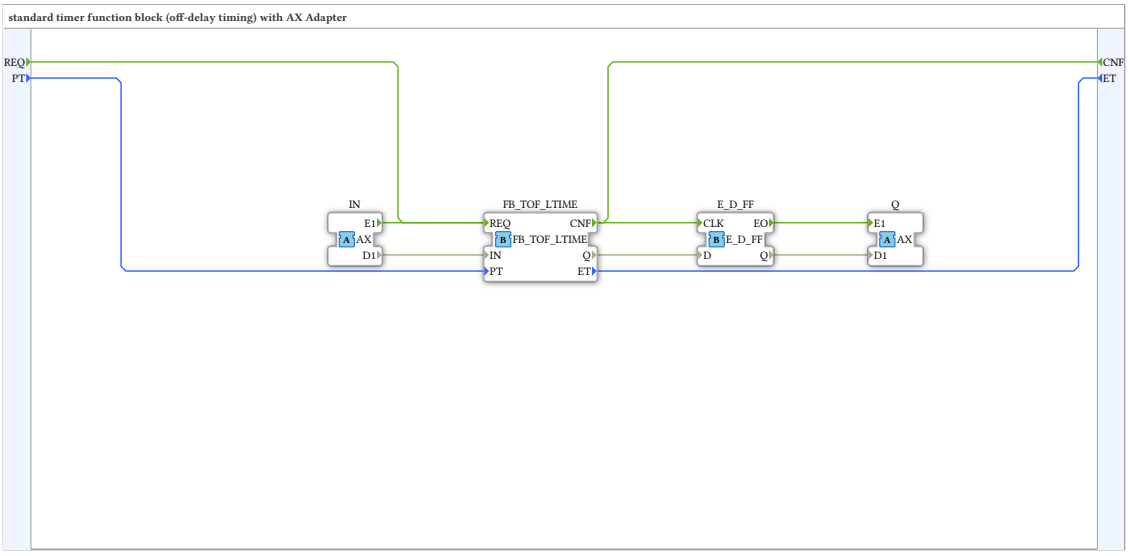
- Exercise_020e2_AX

Conclusion

AX_FB_TOF is the ideal function block for all applications that require a precise switch-off delay with status feedback on the elapsed time.

AX_FB_TOF_LTIME





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_FB_TOF_LTIME** implements a standard timer for the off-delay according to IEC 61131-3. It is specifically designed for use with an AX adapter and uses an internal timer module and a D flip-flop to precisely represent the timing behavior. The time base uses the data type `LTIME`, resulting in high temporal resolution.

Interface Structure

Event Inputs

Event	Comment
`REQ`	Normal execution command (non-triggering) – starts or updates the timer calculation. The corresponding data input is `PT`.

Event Outputs

Event	Comment
`CNF`	Execution confirmation – signals the completion of a timer update. The corresponding data output is `ET`.

Data Inputs

Variable	Type	Comment
`PT`	`LTIME`	Process Time – The desired shutdown delay duration.

Data Outputs

Variable	Type	Comment

ET	LTIME	Elapsed Time – The time elapsed since the delay started.

Adapter

Direction	Adapter Type	Comment
Socket IN	adapter::types::unidirectional::AX	Input adapter – receives the binary control signal (e.g., via event E1 and data D1).
Plug Q	adapter::types::unidirectional::AX	Output adapter – forwards the time-delayed output signal (via event E1 and data D1).

Functionality

The module behaves like a classic time-of-flight (TOF) delay circuit:

- As long as the input IN (via the adapter) is **TRUE**, the output Q is also **TRUE**, and the elapsed time ET is set to zero.
- As soon as IN changes from **TRUE** to **FALSE**, the timing starts. The output Q remains **TRUE** for the duration of the specified time PT and only then falls to **FALSE**.
- As soon as IN changes from **TRUE** to **FALSE**, the timing measurement starts. The output Q remains **TRUE** for the duration of the specified time PT and only then falls to **FALSE**.
- As long as the input Q (via the adapter) is **TRUE**, the output PT remains **TRUE**. If IN becomes **TRUE** again before the delay time expires, the timer is reset and Q remains **TRUE**.

Event-driven processing occurs as follows:

1. The event REQ (either directly or from the adapter input IN.E1) triggers the internal timer FB_TOF_LTIME.
2. The timer calculates the elapsed time and outputs the event CNF upon completion.
3. This CNF clocks the D flip-flop E_D_FF, which stores the current timer output (Q of the timer) and passes it through to the output adapter Q.D1.
4. Simultaneously, the CNF event is forwarded to the output adapter Q.E1, so that the downstream adapter is informed of the change.

The variable ET is taken directly from the timer and displays the currently elapsed delay time at all times.

Technical Features

- **High time resolution** through the use of LTIME (Long Time) according to IEC 61131-3.
- **Adapter-based I/O** via the unidirectional AX adapter enables loose coupling between different resources (e.g., between the application and the I/O driver).
- **Standard compliance** according to IEC 61131-3 – the behavior corresponds exactly to that of a TOF timer.
- **Internal structure**: The component combines a pure timer (FB_TOF_LTIME) with an edge-triggered D flip-flop to implement the turn-off delay behavior.

State overview

State	Input IN	Output Q	Elapsed time ET
Inactive / Ready	TRUE	TRUE	0 (Zero)
Delay in progress	FALSE ⟶ Return to TRUE possible	TRUE	0 ... PT
Delay expired	FALSE	FALSE	PT (constant)

A change from IN to **TRUE** resets the timer and immediately returns it to the "Inactive / Ready" state. A change to **FALSE** restarts the delay.

Application Scenarios

- **Run-on control** of fans, pumps, or conveyor belts after a signal is switched off.
- **Debouncing** of signals: a brief level change is filtered out by the delay.
- **Protection times** when switching inductive loads: the switch-off is delayed by a defined time.

- **Time-controlled logic** in production plants, where an output should remain active for a specific duration after the input event.

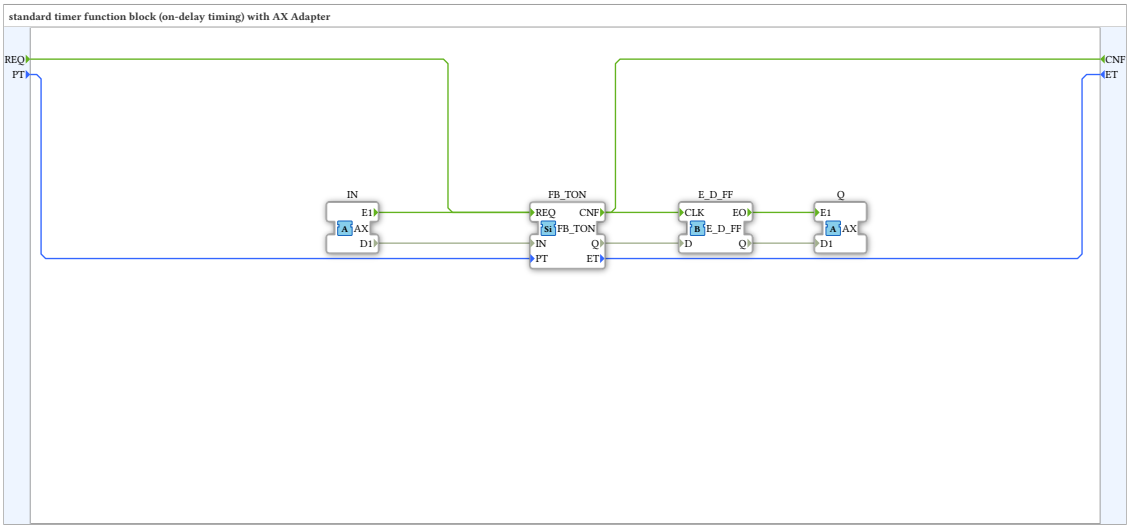
Comparison with similar function blocks

- **TON (Switch-on delay)**: The output only becomes active after a delay following the switch-on. – TOF delays the switch-off.
- **TP (Pulse)**: The output provides a pulse of a predefined length, independent of the duration of the input signal. – TOF only reacts to the falling edge.
- **RTRIG / FTRIG (Edge detection)**: Detect signal changes without a time component. – TOF combines edge detection with time control.

Conclusion

The `AX_FB_TOF_LTIME` is a powerful, standardized function block for time-critical switch-off delays in automation technology. The AX adapter allows for flexible integration into distributed systems, while the internal combination of timer and flip-flop ensures precise and reproducible timing. It is particularly suitable for applications requiring an accurate time base and a modular interface.

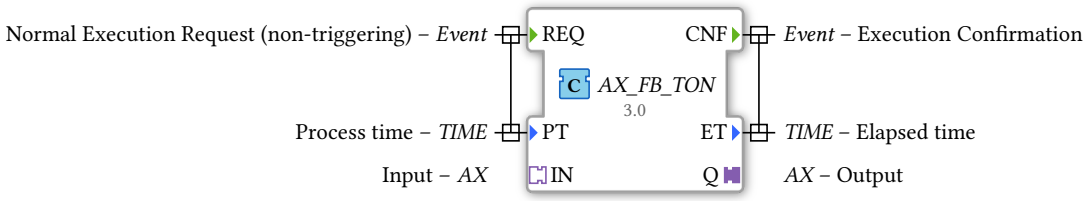
AX_FB_TON



• – ◦ • – ◦ • – ◦ •

Introduction

Important note: This function block only functions correctly if it is called cyclically. AX_FB_TON is a standardized timer function block according to IEC 61131-3 that implements on-delay timing and has adapter interfaces. After activation, the block waits a defined time before setting the output signal and provides the elapsed time.



Interface Structure

Event Inputs

- **REQ** (Event): Normal Execution Request (takes the value from PT); Calls the function block (must be called cyclically)

Event Outputs

- **CNF** (Event): Execution Confirmation; carries `ET`

Data Inputs

- **PT** (TIME): Process time - defines the delay duration

Data Outputs

- **ET** (TIME): Elapsed time

Adapters

- **IN** (Socket): Receives input events and data (Type: `adapter::types::unidirectional::AX`)
- **Q** (Plug): Sends output events and data (Type: `adapter::types::unidirectional::AX`)

Functionality

The AX_FB_TON block functions as a power-on delay timer. The internal timer starts when a signal is received via the IN adapter.

1. When the input on the IN adapter becomes TRUE, the timer starts running.
2. After the set process time (PT) has elapsed, the output on the Q adapter becomes TRUE.
3. The data output ET indicates the time elapsed since the timer started until PT is reached.
4. If the input on the IN adapter becomes FALSE, the timer is reset, and the output on the Q adapter also becomes FALSE.

Technical Features

- Implemented based on the IEC 61131-3 TON standard
- Uses unidirectional adapters for integration into adapter-based architectures
- Provides an ET output for monitoring the time progression
- Compatible with IEC 61131-3 and IEC 61499

State Overview

1. **Idle State**: IN is FALSE, timer inactive, Q is FALSE, ET is 0.
2. **Timing State**: IN is TRUE, timer running, ET is increasing, Q is FALSE.
3. **Triggered**: IN is TRUE, ET has reached PT, Q is TRUE.
4. **Reset**: IN becomes FALSE, timer stops, Q becomes FALSE, ET becomes 0.

Application Scenarios

- Classic power-on delays in PLC programming
- Monitoring of process times with visualization of remaining runtime (via ET)
- Cascading of time sequences in adapter-based systems

⚖️ Comparison with similar function blocks

Compared to AX_TON, AX_FB_TON additionally offers the **ET** output for the elapsed time and strictly follows the behavior of the IEC 61131-3 TON block, while AX_TON is optimized more for the IEC 61499 E_TON.

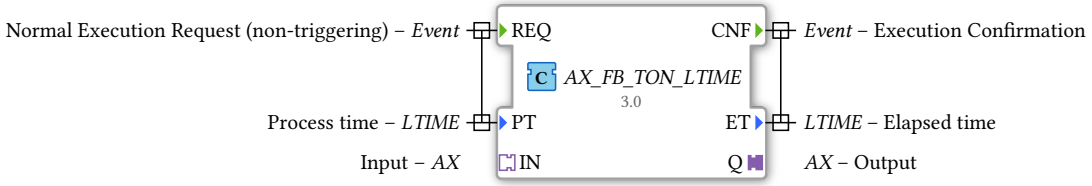
🛠️ Related Exercises

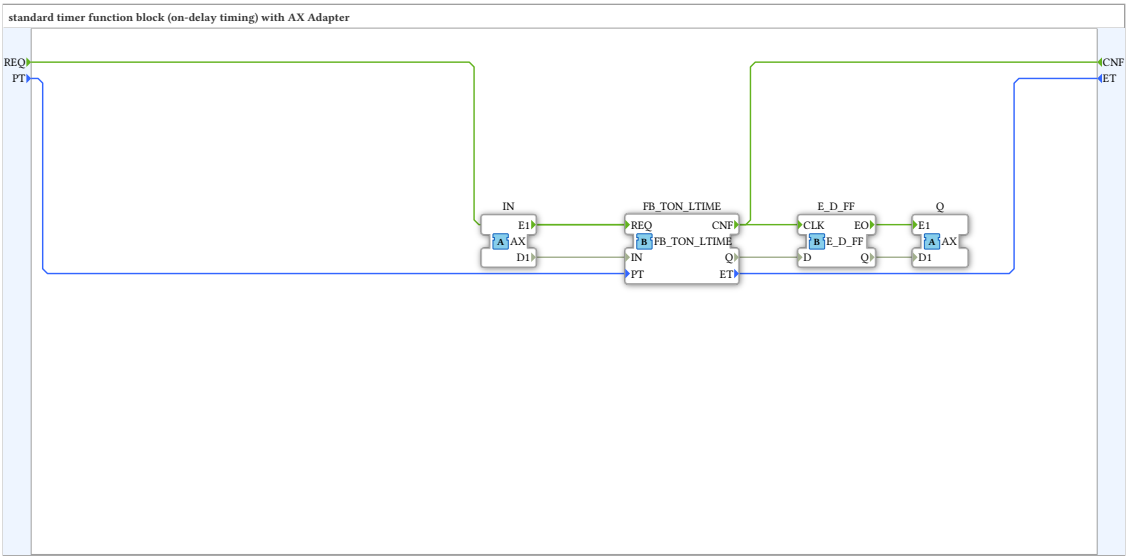
- Exercise_020c3_AX

Conclusion

AX_FB_TON combines the proven behavior of the IEC 61131-3 TON timer with the flexibility of adapter interfaces, thus enabling precise time control with feedback of the current time status.

AX_FB_TON_LTIME





• – ○ • – ○ • – ○ •

Introduction

The **AX_FB_TON_LTIME** function block implements an on-delay timer according to IEC 61131-3 with the data type `LTIME` and an integrated **AX adapter** interface. It is particularly suitable for time-critical control tasks where both events and data must be exchanged via standardized adapters.

Interface Structure

Event Inputs

Name	Type	Description
REQ	`Event`	Normal execution trigger (not edge-triggered); The timer starts when the signal at data input `IN` or at adapter `IN.E1` is active.

Event Outputs

Name	Type	Description
CNF	`Event`	Confirmation of execution. Triggered as soon as the timer expires or the condition is met.

Data Inputs

Name	Type	Description
PT	`LTIME`	Preset time for the power-on delay.

Data Outputs

Name	Type	Description
ET	`LTIME`	Elapsed time since the start of the timer.

Adapter

Type	Direction	Description
`IN`	Socket (Input)	Unidirectional AX adapter for input data. Contains an event (`E1`) and a data value (`D1`). The event starts the timer; the data value determines the timer input (`IN`).

Q	Plug (Output)	Unidirectional AX adapter for output data. Outputs the result of the time check: 1 if the elapsed time has reached the target time, otherwise 0.

Functionality

The function block is implemented as a composite and internally contains the IEC 61131-3 timer **FB_TON_LTIME** and an edge-triggered D flip-flop (**E_D_FF**).

- **Start Conditions**: The timer starts as soon as an event arrives at the adapter IN.E1 or at the event input REQ. Simultaneously, the data value IN.D1 (or the internal timer input) must be set to TRUE.
- **Expiration**: The internal timer counts up the time. The output ET outputs the currently elapsed time.
- **End**: When the elapsed time ET reaches the target time PT, the timer output (Q) is set to TRUE. This triggers the event CNF and clocks the D flip-flop.
- **Output Adapter**: The D flip-flop stores the state and outputs it via the adapter Q.D1. The event Q.E1 signals the output change.

Technical Features

- **LTIME Data Type**: Enables very high time resolution (nanosecond range) and large time ranges.
- **Adapter Interface**: The component can be integrated into existing systems via standardized AX adapters without requiring direct event or data connections.
- **Edge-Triggered Output**: The internal D flip-flop updates the output state only on a rising edge of the timer termination – this prevents unwanted fluctuations.
- **Internal Network**: Functionality is achieved through the combination of standard function blocks, making the implementation transparent and testable.

State Overview

The function block itself does not have its own state diagram, as it consists of an internal function block network. However, its behavior can be derived from the underlying IEC timer function block:

1. **Inactive** – Timer is not running, ET = 0, output Q = FALSE.
2. **Timer is running** – Input IN = TRUE, timer is incrementing ET.
3. **Expired** – ET ≥ PT; Q changes to TRUE and remains until IN falls to FALSE or a new REQ resets the timer.

Application Scenarios

- **Delayed Actuator Activation**: A valve or motor should only be activated after a defined delay time.
- **Process Synchronization**: Waiting for a signal to arrive via the adapter before triggering a time-critical action.
- **Response Time Monitoring**: Recording the elapsed time between an event and a specific condition.
- **Extension of existing AX-based systems**: Replacement of standard timer modules with the adapter variant without changing the overall structure.

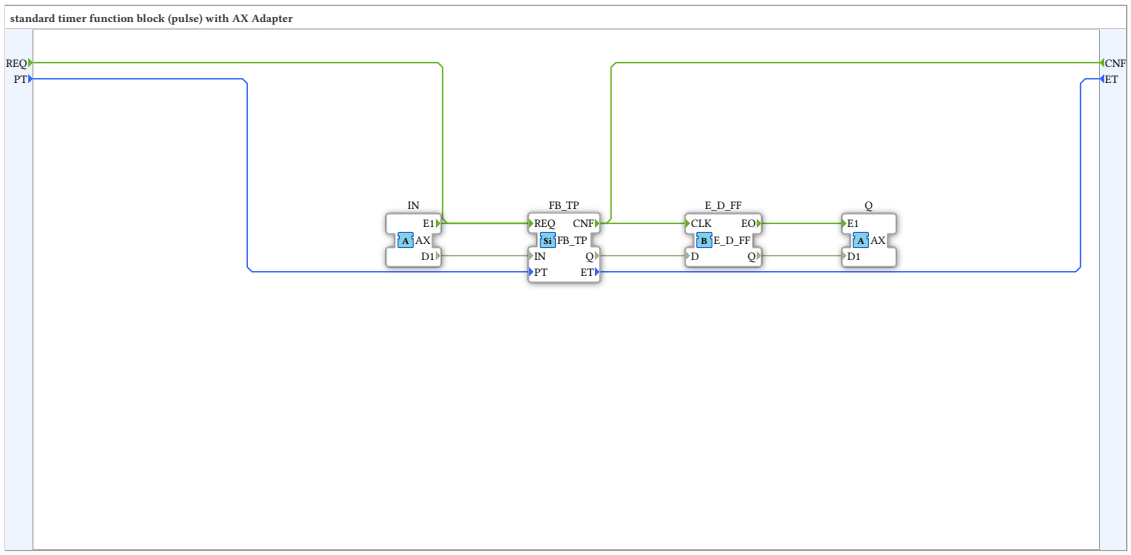
Comparison with similar modules

- **FB_TON_LTIME** (pure IEC timer): Does not have adapters and requires direct event and data connections. The AX_FB_TON_LTIME extends this with a standardized, easily connectable interface.
- **Event-driven timers** (e.g., E_TON): Often use Boolean inputs and do not have native LTIME support. The module presented here combines precise long-term measurement with flexible adapter connectivity.

Conclusion

The **AX_FB_TON_LTIME** is a powerful, adapter-based turn-on delay module for the IEC 61499 environment. It combines the advantages of the standardized IEC 61131-3 timer (LTIME, precise time measurement) with the modularity and reusability of AX adapters. The internal integration with a D flip-flop ensures reliable, edge-triggered output. It is ideally suited for time-critical controllers embedded in an adapter-based architecture model.

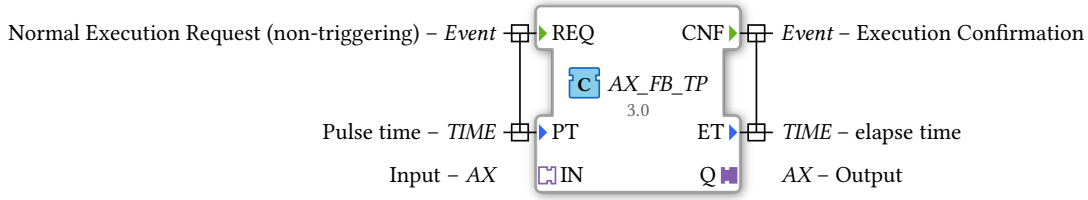
AX_FB_TP



• – ○ • – ○ • – ○ •

Introduction

Important note: This function block only functions correctly if it is called cyclically. AX_FB_TP is a standardized timer function block according to IEC 61131-3 that generates a pulse of defined length. A rising edge at the input generates an output pulse for the specified duration.



Interface Structure

Event Inputs

- **REQ** (Event): Normal Execution Request (takes the value from PT); Calls the function block (must be called cyclically)

Event Outputs

- **CNF** (Event): Execution Confirmation (with ET)

Data Inputs

- **PT** (TIME): Pulse time - defines the duration of the output pulse

Data Outputs

- **ET** (TIME): Elapsed Time

Adapters

- **IN** (Socket): Receives input events and data (Type: `adapter::types::unidirectional::AX`)
- **Q** (Plug): Sends output events and data (Type: `adapter::types::unidirectional::AX`)

Functionality

The AX_FB_TP block functions as a pulse generator:

1. On a rising edge (change from FALSE to TRUE) at the IN adapter, the output at the Q adapter immediately changes to TRUE.
2. The internal timer starts and runs for the duration PT.

3. The output on the Q-adapter remains TRUE for the entire duration PT, regardless of whether the input signal on the IN-adapter changes back to FALSE in the meantime.
4. While the pulse is running (as long as ET < PT), further edges at the input have no effect on the running timer (not retriggerable).
5. After the duration PT has elapsed, the output on the Q-adapter returns to FALSE.
6. The data output ET displays the time that has elapsed since the pulse started.

Technical Features

- Implemented based on the IEC 61131-3 TP standard
- Uses unidirectional adapters for clean encapsulation in adapter-based architectures
- Not retriggerable during the active pulse phase
- Precise timing control through the use of the TIME data type

State Overview

1. **Idle State**: IN is FALSE, Q is FALSE, ET is 0.
2. **Pulse Phase**: Rising edge detected at IN, Q is TRUE, ET rises to PT.
3. **End Phase**: ET has reached PT, Q changes to FALSE.
4. **Waiting for Reset**: If IN is still TRUE, Q remains FALSE and ET remains at PT until IN becomes FALSE again.

Application Scenarios

- Generation of time-limited control signals (e.g., start pulse for a motor)
- Time-limiting of operating processes
- Generation of defined signal pulses in process automation

⚖️ Comparison with similar function blocks

Compared to AX_PULSE, AX_FB_TP offers the classic IEC 61131-3 TP characteristic, including the ET status output.

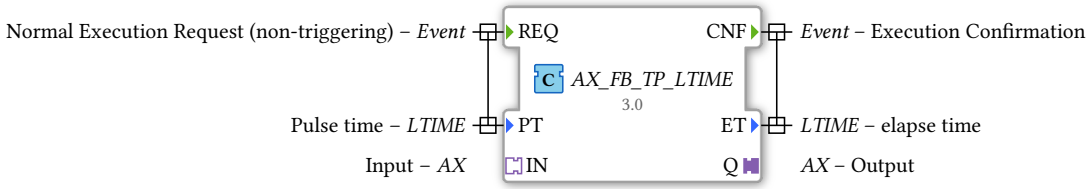
🛠️ Related Exercises

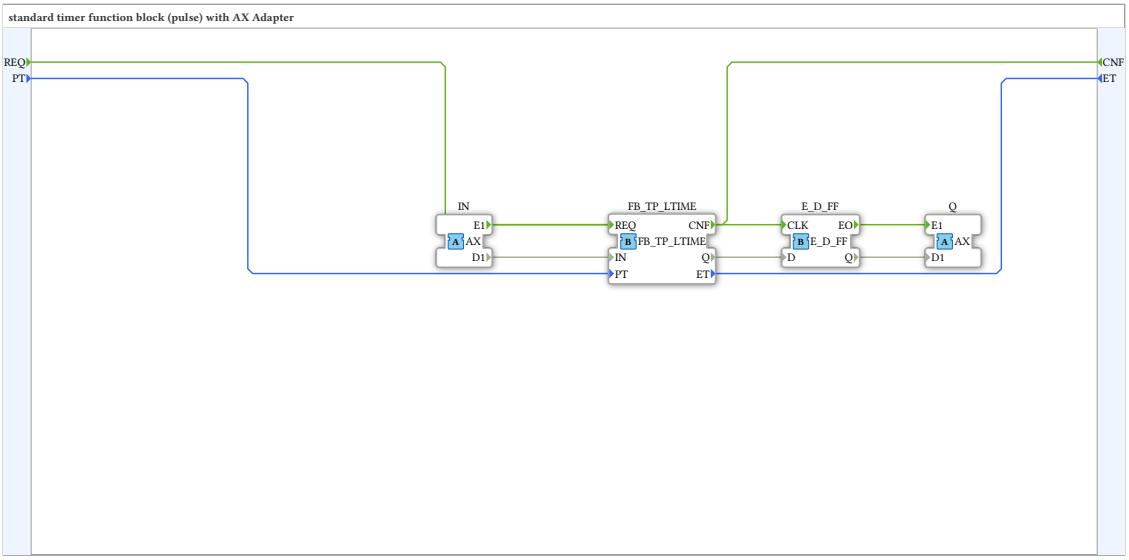
- Exercise_020f2_AX

Conclusion

AX_FB_TP is the standard function block for generating stable, precisely time-defined pulses in an adapter-based environment.

AX_FB_TP_LTIME





• – ○ • – ○ • – ○ •

Introduction

The **AX_FB_TP_LTIME** is a standardized pulse timer function block (timer pulse) according to IEC 61131-3, extended by a unidirectional **AX adapter**. It is used in the 4diac IDE to generate time-limited pulses and pass the output signals to the application via an adapter. Its behavior is similar to a **TP** timer, which sets the output to TRUE for a predefined time on a rising edge at the input.

Interface Structure

Event Inputs

Name	Type	Comment
REQ	Event	Normal execution request (non-triggering)

Event Outputs

Name	Type	Comment
CNF	Event	Execution Acknowledgement

Data Inputs

Name	Type	Comment
PT	LTIME	Pulse Time (Pulse Duration)

Data Outputs

Name	Type	Comment
ET	LTIME	Elapsed Time

Adapters

- **Plugs:**
- **Q**: Output adapter type `adapter::types::unidirectional::AX`.

This adapter passes the Boolean output signal `Q.D1` and the event `Q.E1` to the application.

- **Sockets:**
- **IN**: Input adapter type `adapter::types::unidirectional::AX`.

The function block receives the start event (via IN.E1) and, if necessary, a Boolean signal (via IN.D1) via this adapter. This signal is used internally as an enable or additional condition (not directly connected to the timer logic, but routed to an internal standard timer).

Functionality

The function block consists of two internal components:

1. **FB_TP_LTIME** – a pure pulse timer (IEC 61131-3 TP) with a time base of LTIME.
2. **E_D_FF** – an edge-triggered D flip-flop (E-D-FF according to IEC 61499) that buffers the Q output of the timer.

Procedure:

- An event via the adapter input IN.E1 or the standard event input REQ starts the timer FB_TP_LTIME (the events are internally combined – both trigger the same start).
- The timer increments the elapsed time ET and sets its internal Boolean output Q to TRUE as soon as PT is reached. After the pulse duration elapses, Q returns to FALSE and a CNF event is output.
- The CNF event clocks the D flip-flop E_D_FF, which takes over the current state of Q (i.e., TRUE during the pulse duration, FALSE afterward) and makes it available at the output Q.D1 of the adapter.
- Simultaneously, the CNF event is also directly forwarded to the output CNF of the overall module, thus informing the application about the end of the pulse.
- The output ET indicates the current elapsed time and is directly connected to the output of the internal timer.

Technical Features

- **Adapter Interface:** The function block encapsulates the timer logic in an adapter interface, allowing for easy integration into modular, adapter-based architectures.
- **D Flip-Flop for Signal Hold:** The timer's Q output is synchronized with the CNF event via a flip-flop to prevent glitches or asynchronous changes.
- **Dual Event Path:** The timer can be started via the adapter (IN.E1) or via the standard event input REQ – both are internally connected, enabling flexible control.
- **LTIME Support:** The pulse time is processed as LTIME (Long Time, 64-bit nanoseconds), enabling extremely high time resolution and large time ranges.

State Overview

Since the functional block consists of two parts, the following state behavior results for the complete module:

State	Description
IDLE	No pulse active. The adapter output Q.D1 is FALSE.
TIMING	A pulse is running – the internal timer is counting up. ET is increasing, Q.D1 is TRUE (as soon as the flip-flop was clocked at startup or after the first CNF).
DONE	The pulse is complete. CNF was output, ET is equal to PT, Q.D1 will be set to FALSE with the next CNF. The cycle starts again with a new REQ or IN.E1.

Application Scenarios

- **Pulse Output with Defined Length:** Controlling actuators (e.g., valves, lamps) that are to be switched on for a specific time.
- **Adapter-Based Subsystems:** Integrating the timer into a modular automation solution where inputs and outputs communicate via adapters (AX protocol).
- **High-Resolution Time Measurement:** Utilizing LTIME for precise timing in the micro- or nanosecond range.

Comparison with Similar Function Blocks

Function Block	Type	Feature

FB_TP_LTIME (Standard)	TP Timer	No adapter, only standard events and signals.
AX_FB_TP_LTIME (this FB)	TP Timer with AX Adapter	Enables integration into adapter-based architectures, includes a D flip-flop for signal stabilization.
FB_TON_LTIME	Turn-On Delay	Delays the activation of an output.
FB_TOF_LTIME	Turn-Off Delay	Delays the switching off of an output.

This module combines the timer function of a twisted pair (TP) with the flexibility of an adapter.

Conclusion

The **AX_FB_TP_LTIME** is a robust and precise pulse timer for the 4diac IDE. Thanks to its adapter interface and the use of `LTIME`, it is particularly well-suited for high-resolution, modular automation solutions. The internal flip-flop structure ensures clean signal transitions, and the dual event circuitry offers flexibility in control.

🌐 **Related topic subpages on ms-muc-docs.de**

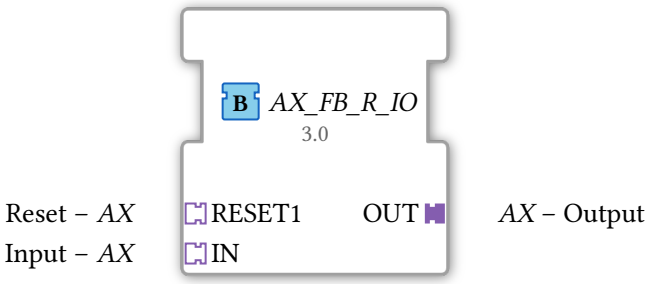
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.8.0.1 Numeric Index (adapter)

Here you will find all exercises from the `adapter` workspace, listed in ascending order. ...

AX_FB_R_IO





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_FB_R_IO** implements monostable, reset-dominant behavior. It has three adapter interfaces through which it forwards an input signal (IN) to the output (OUT) if no active reset signal (RESET1) is present. The block operates on the principle: **Reset has priority**.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

Name	Type	Direction	Comment
IN	unidirectional::AX	Socket	Input signal
RESET1	unidirectional::AX	Socket	Reset signal (prioritized)
OUT	unidirectional::AX	Plug	Output signal

The adapters of type `adapter::types::unidirectional::AX` each have an event output `E1` and a data output `D1` (of type BOOL). The adapter's data value is provided via `D1`, and processing is triggered by the event `E1`.

Functionality

The core logic of the component is represented by a simple state machine with a single state, `REQ`. As soon as an event arrives at one of the two input adapters (`IN.E1` or `RESET1.E1`), the algorithm `REQ` is executed. This algorithm checks the reset input:

- **If `RESET1.D1 = TRUE`**: The output `OUT.D1` is set to `FALSE`, regardless of the input signal.
- **Otherwise**: The output `OUT.D1` directly receives the value of the input signal `IN.D1`.

The event at the output adapter (`OUT.E1`) is triggered with each execution of the algorithm, so that subsequent function blocks are informed of the value change.

Technical Features

- **Reset Dominance**: The reset signal `RESET1` always takes precedence over the input signal `IN`. As long as `RESET1.D1` is active, the output `FALSE` remains active.
- **Monostable Behavior**: The function block automatically returns to the standby state (one state) after each processing operation. There is no stored state; each execution reacts immediately to the current inputs.
- **Adapter-Based Communication**: All inputs and outputs are handled via bidirectional adapters that transport both data and events. Event control is implicit via the adapter protocols.
- **Package Name**: `adapter::monostableElements` (defined in CompilerInfo).

State Overview

The function block has exactly one state:

State	Description
REQ	Start and end state. Waits for an event from `IN` or `RESET1`. After the algorithm has finished executing, the state always returns to `REQ` (no other states).

There are two transitions:

- From `REQ` to `REQ` upon event `IN.E1`
- From `REQ` to `REQ` upon event `RESET1.E1`

Both transitions return to the same state and trigger the execution of the algorithm.

Application Scenarios

- **Enable circuit in automation technology**: A signal (e.g., machine start) is only passed to the output if no reset command (e.g., stop) is active.
- **Safety interlock**: In a control chain, a reset signal is given priority to reliably eliminate dangerous states.
- **Priority Signal Switch**: Two sources compete for one output – here, the reset function always takes precedence.

Comparison with Similar Function Blocks

- **AX_FB_RS** (Set-Dominant): Unlike this function block, the set input has priority. `AX_FB_R_IO` is reset-dominant.
- **AX_FB_SR**: Set-reset flip-flop with equal inputs, often without priority (can lead to undefined states). `AX_FB_R_IO` avoids this through clear dominance.
- **AX_FB_MONOSTABLE**: Pure monostable flip-flops with time parameters; `AX_FB_R_IO` has no time behavior but reacts instantly to signals.

Conclusion

The function block `AX_FB_R_IO` is a simple yet reliable building block for realistically implementing prioritized reset behavior. Thanks to its adapter interface, it integrates seamlessly into event-driven automation systems. Its clear reset dominance makes it particularly suitable for safety and enabling functions where a reset signal must have absolute priority.

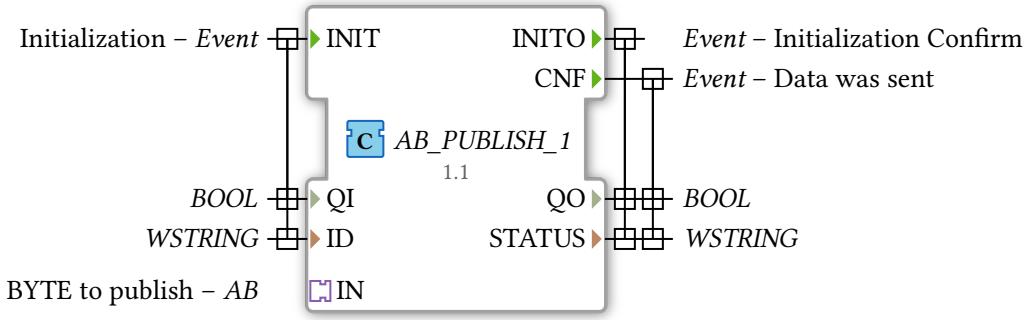
2.8.0.2 net

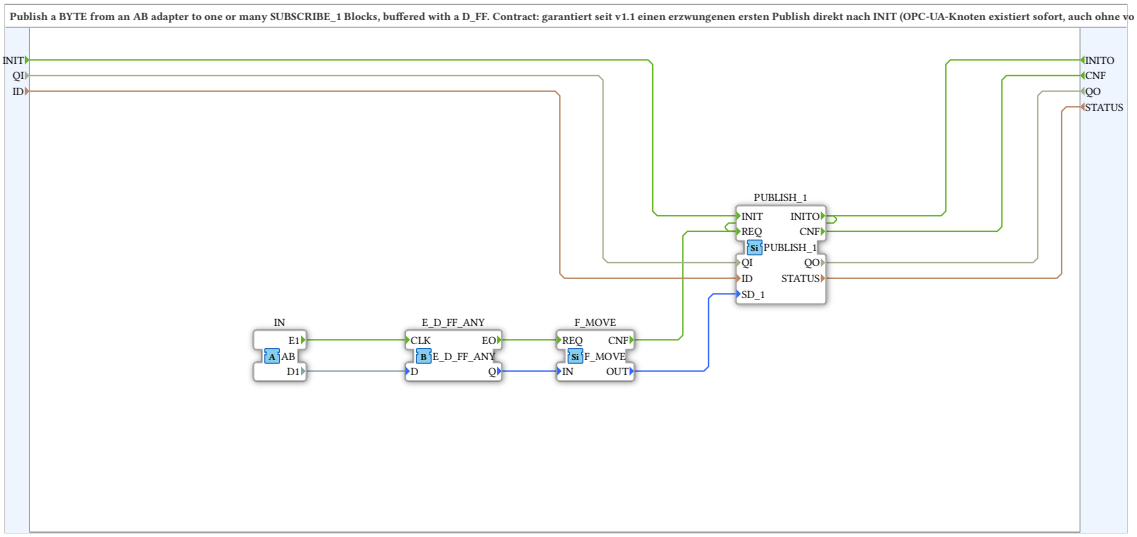
Here you will find function blocks that map network communication (publish/subscribe and OPC UA client/server) to the unidirectional adapters. These blocks encapsulate the corresponding `iec61499::net` function blocks and buffer the transmitted values using a D flip-flop.

- AB_PUBLISH_1
- AB_SUBSCRIBE_1
- AD_PUBLISH_1
- AD_SUBSCRIBE_1
- ADI_PUBLISH_1
- ADI_SUBSCRIBE_1
- AI_PUBLISH_1
- AI_SUBSCRIBE_1
- AIS_PUBLISH_1
- AIS_SUBSCRIBE_1
- AIWS_PUBLISH_1
- AIWS_SUBSCRIBE_1
- AL_PUBLISH_1
- AL_SUBSCRIBE_1
- ALI_PUBLISH_1
- ALI_SUBSCRIBE_1
- ALR_PUBLISH_1
- ALR_SUBSCRIBE_1
- AQ_PUBLISH_1
- AQ_SUBSCRIBE_1
- AR_CLIENT_1_0
- AR_CLIENT_1_0_UNGATED
- AR_PUBLISH_1
- AR_SERVER_0_1
- AR_SUBSCRIBE_1
- AS_PUBLISH_1
- AS_SUBSCRIBE_1
- ATM_CLIENT_1_0
- ATM_PUBLISH_1
- ATM_SERVER_0_1
- ATM_SUBSCRIBE_1
- AUDI_PUBLISH_1
- AUDI_SUBSCRIBE_1
- AUI_PUBLISH_1
- AUI_SUBSCRIBE_1
- AULI_PUBLISH_1
- AULI_SUBSCRIBE_1
- AUS_PUBLISH_1
- AUS_SUBSCRIBE_1
- AW_PUBLISH_1
- AW_SUBSCRIBE_1

- AX_CLIENT_1_0
- AX_PUBLISH_1
- AX_SERVER_0_1
- AX_SUBSCRIBE_1
- AE_AX_CLIENT_0_SUBSCRIBE_1
- AE_AX_SERVER_0_CLIENT_1_0
- AE_CLIENT_0
- ASR_AX_CLIENT_0_SUBSCRIBE_1
- ASR_AX_SERVER_0_CLIENT_1_0
- ASR_CLIENT_0
- ASRT_AX_CLIENT_0_SUBSCRIBE_1
- ASRT_AX_SERVER_0_CLIENT_1_0
- ASRT_CLIENT_0

AB_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AB_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AB adapter**. A **BYTE** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs
- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs
- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs
- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs
- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AB	Socket (Input)	Byte value to be published

Functionality
1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AB socket `IN` delivers an event to `IN.E1`, the corresponding BYTE value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then sent externally.

Technical Features
- **Buffering with D-Flip-Flop**: The byte value to be sent is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AB adapter interface is visible externally.

State Overview
1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AB socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios
- **Network Coupling in AB Applications**: Sending BYTE signals (e.g., switching commands or status bits) from adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AB adapters.

Comparison with Similar Function Blocks

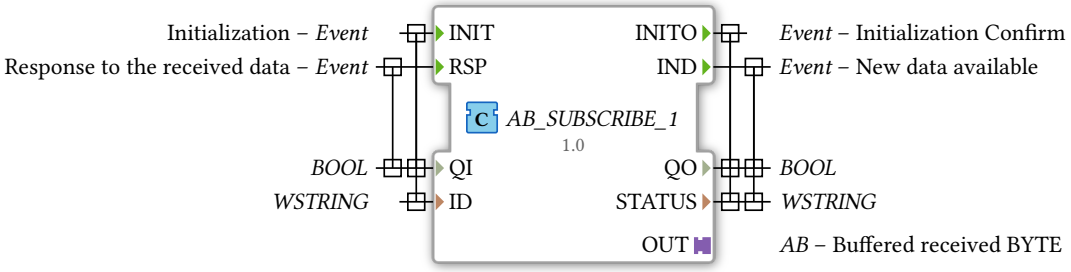
- **PUBLISH_1**: The standard function block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AB_PUBLISH_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AB adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a byte value at the adapter level but lacks network functionality. **AB_PUBLISH_1** combines buffering and publishing in a single function block.

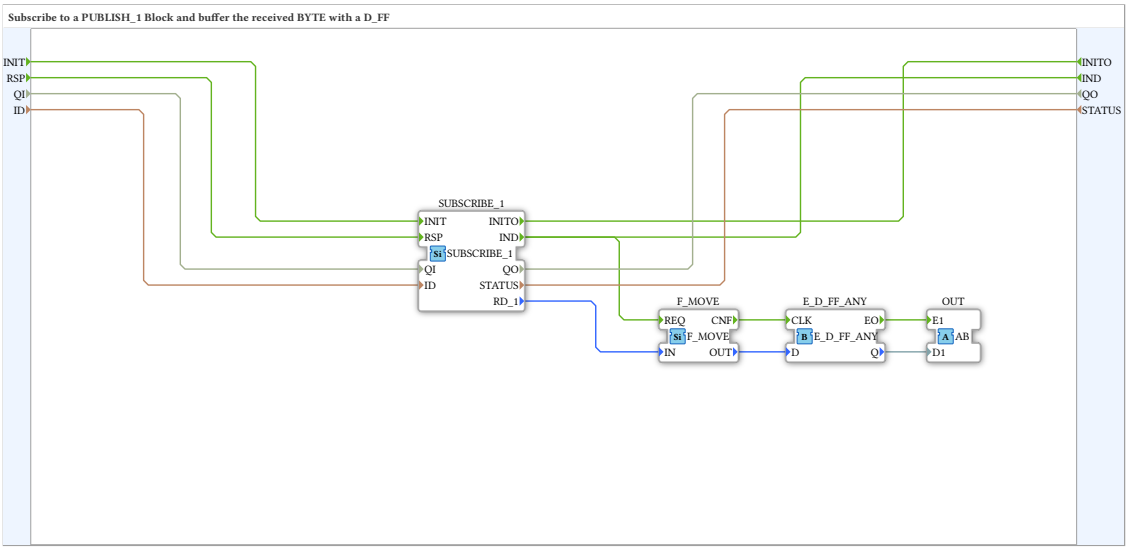
Conclusion

AB_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based BYTE processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AB interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AB_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based BYTE processing.

AB_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AB_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AB adapter**. It subscribes to a publish channel configured via `ID` and makes the received **BYTE** value buffered via a D flip-flop (E_D_FF_ANY) available at the adapter plug `OUT`. This allows for clean integration of network communication and BYTE signal processing in the AB adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AB	Plug (Output)	Buffered received BYTE value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.
5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received BYTE value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the AB adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Received**: At `IND`, the received byte value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AB Applications**: Receiving BYTE signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AB adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a cleaner fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AB adapters.

Comparison with Similar Function Blocks

- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AB_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`),

and only provides the AB adapter – making the interface significantly more compact and integrable into adapter networks.

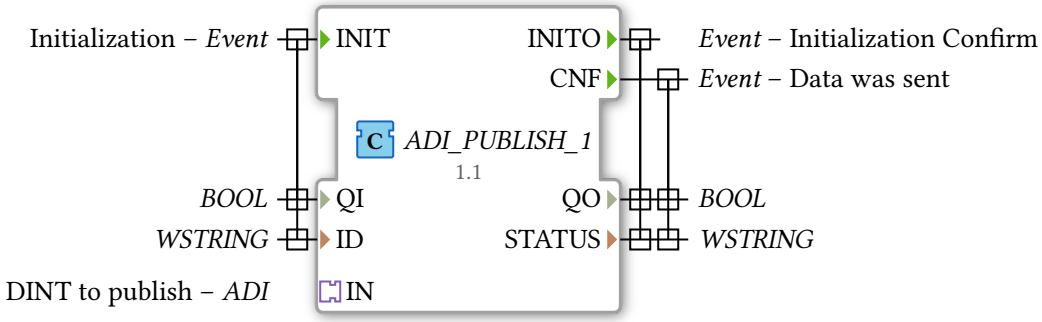
- **AX_D_FF**: Bufferes a byte value at the adapter level but lacks network functionality. **AB_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

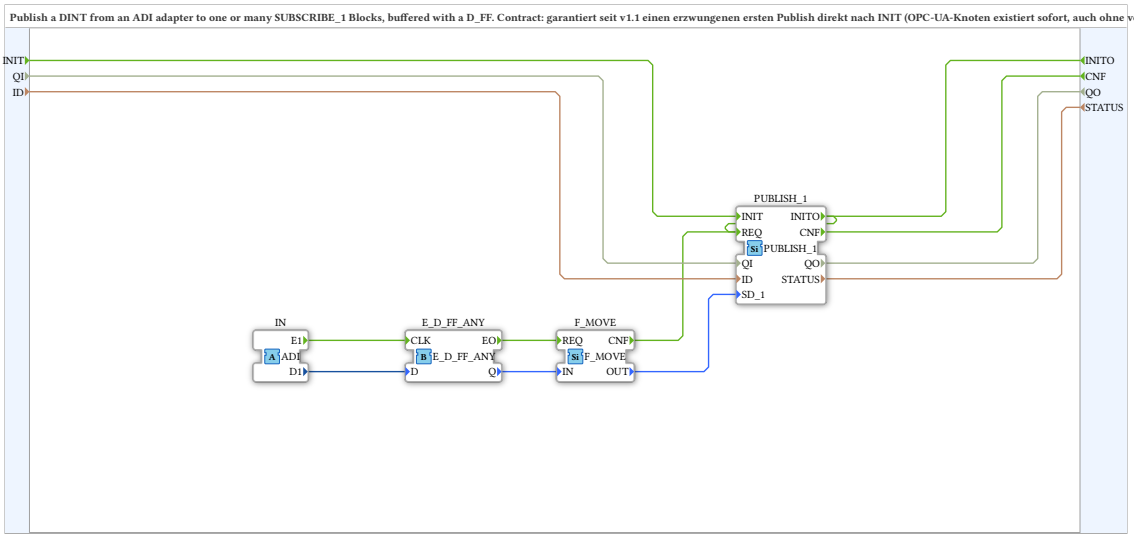
Conclusion

AB_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based BYTE processing. The internal D flip-flop buffer keeps the received value stable until new data arrives, and the reduced AB interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AB_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based BYTE processing.

ADI_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The **ADI_PUBLISH_1** function block is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **ADI adapter**. A **DINT** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::ADI	Socket (Input)	DINT value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the ADI socket `IN` delivers an event to `IN.E1`, the DINT value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then sent out.

Technical Features

- **Buffering with D-Flip-Flop**: The DINT value to be transmitted is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the ADI adapter interface remains visible externally.

Status Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the ADI socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in ADI Applications**: Sending DINT signals (e.g., switching commands or status bits) from the adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between the user's own publications and subscriptions is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on ADI adapters.

Comparison with Similar Function Blocks

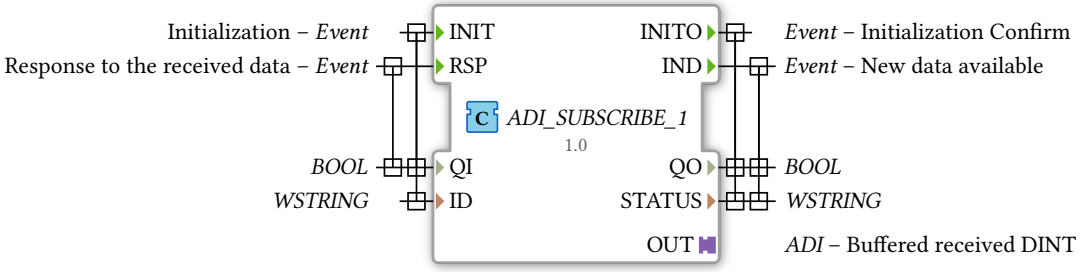
- **PUBLISH_1**: The standard function block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **ADI_PUBLISH_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the ADI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a DINT value at the adapter level but lacks network functionality. **ADI_PUBLISH_1** combines buffering and publishing in a single function block.

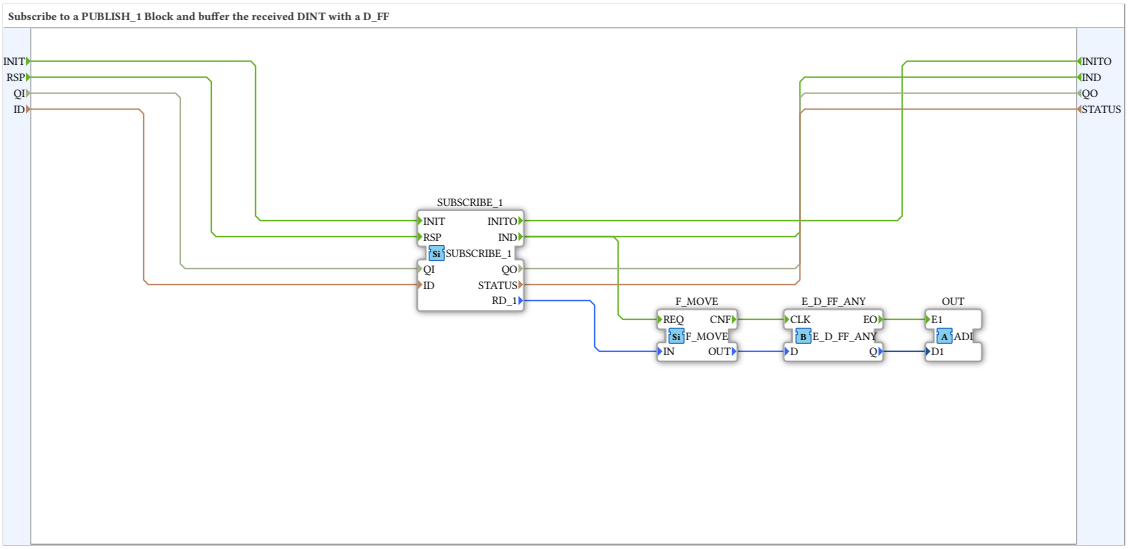
Conclusion

ADI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based DINT processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified ADI interface makes the function block ideal for fully adapter-based IEC 61499 applications.

ADI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based DINT processing.

ADI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **ADI_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **ADI adapter**. It subscribes to a publish channel configured via `ID` and provides the received **DINT** value, buffered via a D flip-flop (E_D_FF_ANY), at the adapter plug `OUT`. This allows for the clean integration of network communication and DINT signal processing in the ADI adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::ADI	Plug (Output)	Buffered received DINT value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.
5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received DINT value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the ADI adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Reception**: At `IND`, the received DINT value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in ADI Applications**: Receiving DINT signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via ADI adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a clean fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on ADI adapters.

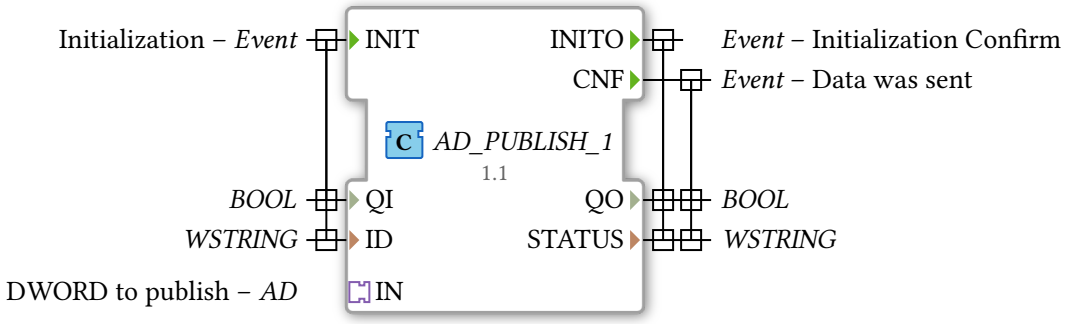
Comparison with Similar Function Blocks

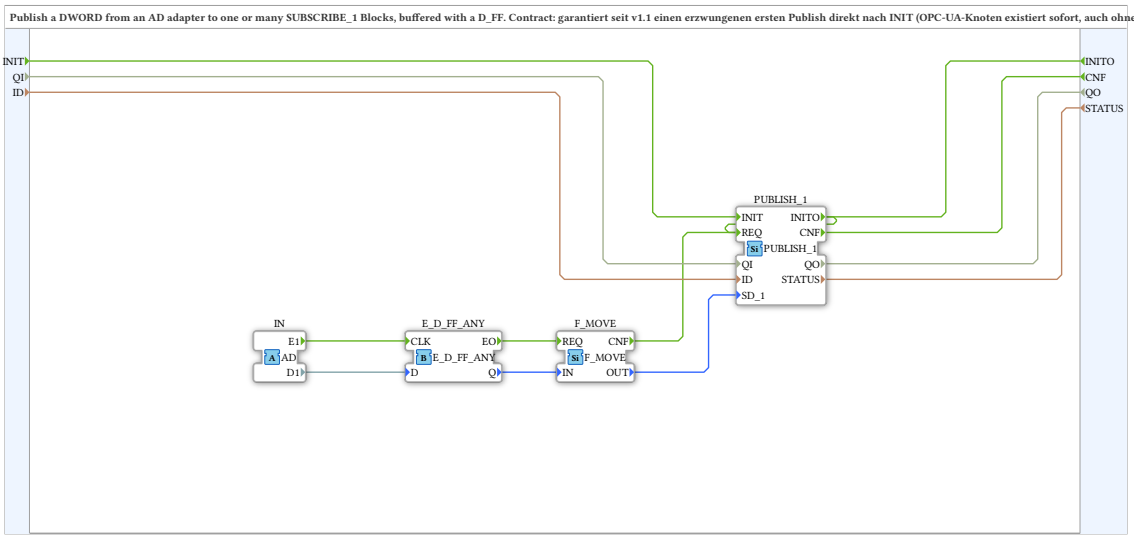
- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (`IND`, `RD_1`, ...). **ADI_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the ADI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a DINT value at the adapter level but lacks network functionality. **ADI_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

Conclusion

ADI_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based DINT processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified ADI interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AD_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AD_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AD adapter**. A **DWORD** value at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AD	Socket (Input)	DWORD value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AD socket `IN` sends an event to `IN.E1`, the DWORD value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then sent externally.

Technical Features

- **Buffering with D-Flip-Flop**: The DWORD value to be sent is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents an input value that changes during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AD adapter interface is visible externally.

Status Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AD socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in AD Applications**: Sending DWORD signals (e.g., toggle commands or status bits) from adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AD adapters.

Comparison with Similar Function Blocks

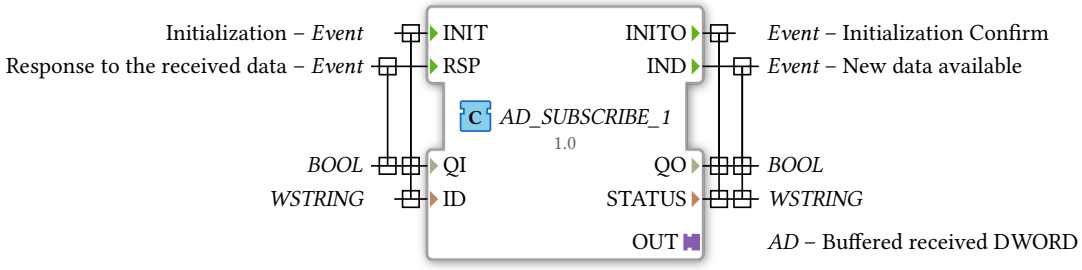
- **PUBLISH_1**: The standard function block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AD_PUBLISH_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AD adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a DWORD value at the adapter level but lacks network functionality. **AD_PUBLISH_1** combines buffering and publishing in a single function block.

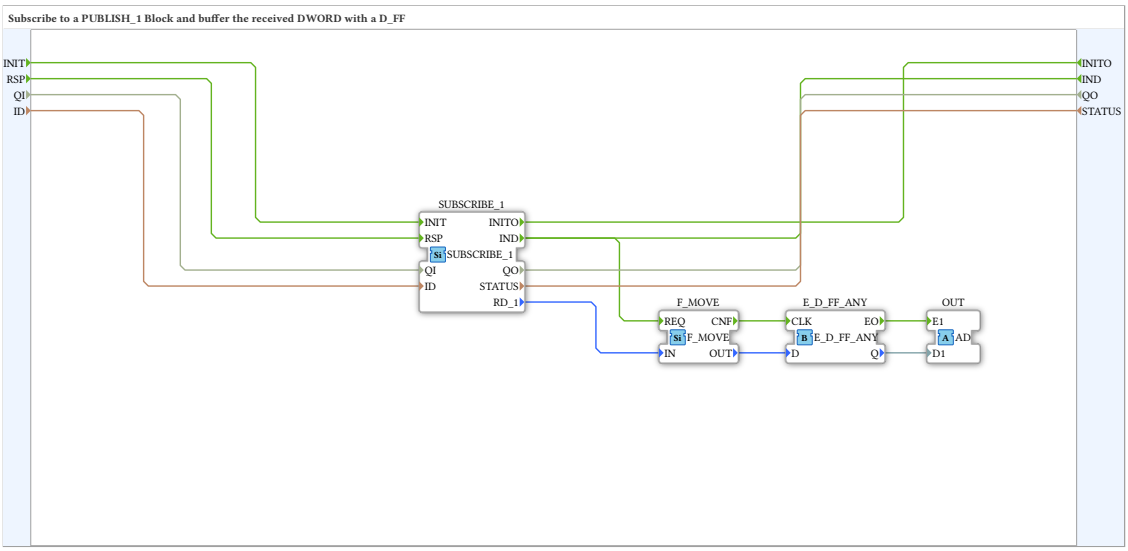
Conclusion

AD_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based DWORD processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AD interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AD_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based DWORD processing.

AD_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AD_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AD adapter**. It subscribes to a publish channel configured via `ID` and makes the received **DWORD** value buffered via a D flip-flop (E_D_FF_ANY) available at the adapter plug `OUT`. This allows for clean integration of network communication and DWORD signal processing in the AD adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description

OUT	adapter::types::unidirectional::AD	Plug (Output)	Buffered received DWORD value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.
5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received DWORD value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the AD adapter interface is visible externally.

Status Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Received**: At `IND`, the received DWORD value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AD Applications**: Receiving DWORD signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AD adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a clean fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AD adapters.

Comparison with Similar Function Blocks

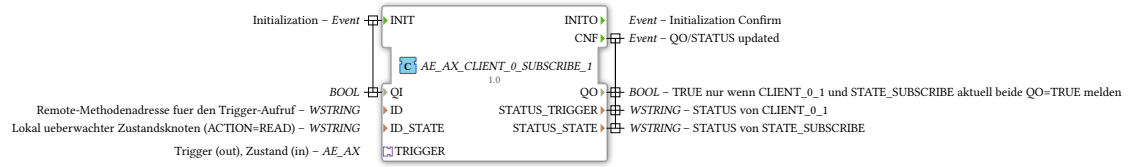
- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AD_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AD adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a DWORD value at the adapter level but lacks network functionality. **AD_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

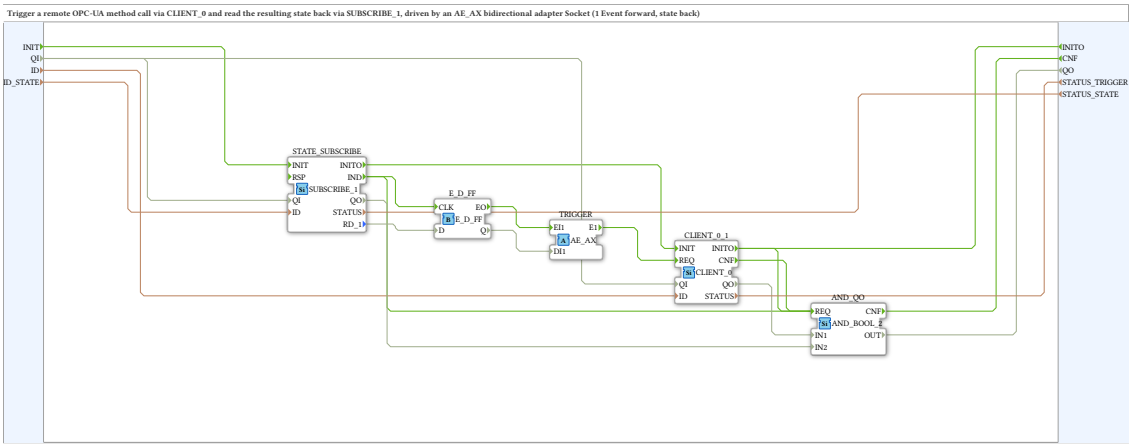
Conclusion

AD_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based DWORD processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AD interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AD_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based DWORD processing.

AE_AX_CLIENT_0_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The **AE_AX_CLIENT_0_SUBSCRIBE_1** function block is a composite function block that triggers a remote OPC UA **method call** via `CLIENT_0` and reads the corresponding state back via `SUBSCRIBE_1` – both behind a single bidirectional **AE_AX adapter socket**. It collapses the pattern of "one hand-wired `CLIENT_0` plus one `SUBSCRIBE_1`" into a single building block with one adapter port.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO`, `STATUS_TRIGGER`, and `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` updated, associated with `QO`, `STATUS_TRIGGER`, and `STATUS_STATE`

Data Inputs

- **QI** (BOOL): Qualifier input for both internal connections
- **ID** (WSTRING): Remote method address for the trigger call (`CLIENT_0_1`)
- **ID_STATE** (WSTRING): Locally monitored state node for `SUBSCRIBE_1` (ACTION=READ)

Data Outputs

- **QO** (BOOL): TRUE only if `CLIENT_0_1` and `STATE_SUBSCRIBE` currently both report `QO = TRUE`
- **STATUS_TRIGGER** (WSTRING): Status information from `CLIENT_0_1`
- **STATUS_STATE** (WSTRING): Status information from `STATE_SUBSCRIBE`

Adapter

Adapter	Type	Direction	Description
TRIGGER	adapter::types::bidirectional::AE_AX	Socket – trigger (output), state (input)	Triggering event out, state back

Functionality

1. Via the `INIT` event, `STATE_SUBSCRIBE` (`SUBSCRIBE_1`) is initialized first with `QI` and `ID_STATE`; after its `INITO`, `CLIENT_0_1` is initialized with `QI` and `ID`. After both have confirmed, `INITO` is reported externally.

2. An event arriving at the adapter socket `TRIGGER.E1` directly triggers the method call `CLIENT_0_1.REQ`.
3. As soon as `STATE_SUBSCRIBE` delivers a new state value at `RD_1` via `IND`, this event clocks the internal **E_D_FF** flip-flop, which captures and holds the value stably.
4. The flip-flop's output `EO` fires `TRIGGER.EI1` with the buffered value at `TRIGGER.DI1` – so the state is reported back over the same adapter used to send the trigger out.
5. `AND_QO` (AND_BOOL_2) ANDs the `QO` outputs of `CLIENT_0_1` and `STATE_SUBSCRIBE`; every confirmation (`CLIENT_0_1.CNF` or `STATE_SUBSCRIBE.IND`) additionally triggers `CNF` externally.

Technical Features

- **Two network connections behind one adapter**: `CLIENT_0` (trigger, write direction) and `SUBSCRIBE_1` (state, read direction) are combined into a single bidirectional AE_AX interface.
- **Buffering with D flip-flop**: The state received via `SUBSCRIBE_1` is stabilized via an internal `iec61499::events::E_D_FF` before being reported back over the adapter.
- **Sequential initialization**: `STATE_SUBSCRIBE` is initialized before `CLIENT_0_1` (composite FBTypes chain `INIT`/`INITO` serially through each instance).
- **Encapsulation**: Only the AE_AX adapter interface is visible externally; `CLIENT_0`'s `REQ` and `SUBSCRIBE_1`'s `IND`/`RD_1` remain internal.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: Both connections (trigger and state subscription) are established.
3. **Send Active**: A trigger event arriving at the adapter triggers the remote method call.
4. **State Update**: A new value from the subscription is buffered and reported back over the same adapter.

Application Scenarios

- Remotely triggering a remote action while simultaneously reading back the resulting state over a single adapter connection, e.g. for a control element that immediately shows the actual state of the remote actuator
- Reducing manual CLIENT_0/SUBSCRIBE_1 wiring to a single reusable building block

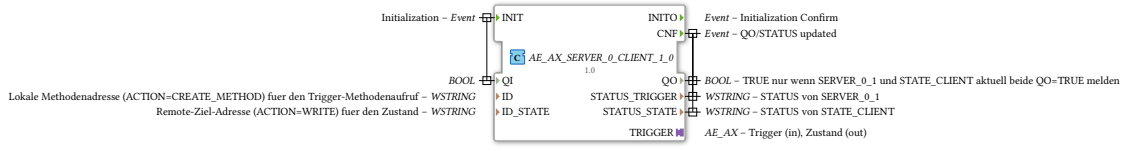
Comparison with Similar Function Blocks

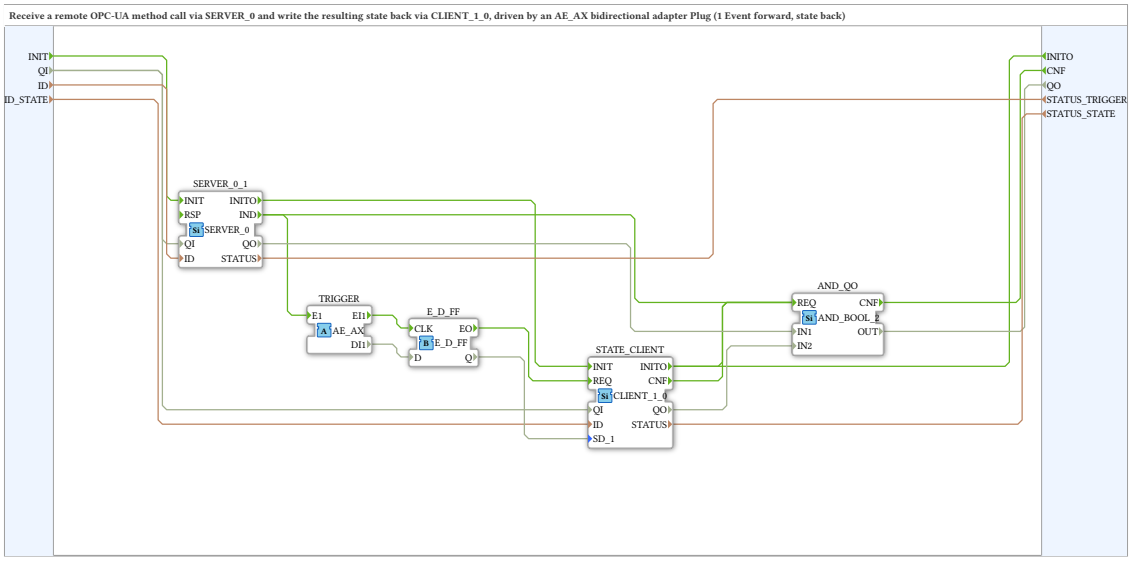
- **AE_CLIENT_0**: A pure trigger without reading the state back.
- **AI_SUBSCRIBE_1**: A pure state subscription without trigger functionality.
- **ASR_AX_CLIENT_0_SUBSCRIBE_1** / **ASRT_AX_CLIENT_0_SUBSCRIBE_1**: Extend the same pattern to 2 and 3 parallel triggers.
- **AE_AX_SERVER_0_CLIENT_1_0**: The server-side counterpart – receives instead of triggering, and writes instead of subscribing.

Conclusion

AE_AX_CLIENT_0_SUBSCRIBE_1 bundles a remote trigger and a state subscription behind a single bidirectional AE_AX adapter, turning a hand-wired CLIENT_0/SUBSCRIBE_1 combination into a single reusable building block.

AE_AX_SERVER_0_CLIENT_1_0





• – ○ • – ○ • – ○ •

Introduction

The **AE_AX_SERVER_0_CLIENT_1_0** function block is a composite function block that receives a remote OPC UA **method call** via `SERVER_0` and writes the resulting state back to a remote node via `CLIENT_1_0` – both behind a single bidirectional **AE_AX adapter plug**. It is the server-side counterpart to AE_AX_CLIENT_0_SUBSCRIBE_1.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO`, `STATUS_TRIGGER`, and `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` updated, associated with `QO`, `STATUS_TRIGGER`, and `STATUS_STATE`

Data Inputs

- **QI** (BOOL): Qualifier input for both internal connections
- **ID** (WSTRING): Local method address (ACTION=CREATE_METHOD) for the trigger method call (`SERVER_0_1`)
- **ID_STATE** (WSTRING): Remote target address (ACTION=WRITE) for the state (`STATE_CLIENT`)

Data Outputs

- **QO** (BOOL): TRUE only if `SERVER_0_1` and `STATE_CLIENT` currently both report `QO = TRUE`
- **STATUS_TRIGGER** (WSTRING): Status information from `SERVER_0_1`
- **STATUS_STATE** (WSTRING): Status information from `STATE_CLIENT`

Adapter

Adapter	Type	Direction	Description
TRIGGER	adapter::types::bidirectional::AE_AX	Plug – trigger (input), state (output)	Received event, state out

Functionality

1. Via the `INIT` event, `SERVER_0_1` is initialized first with `QI` and `ID`; after its `INITO`, `STATE_CLIENT` is initialized with `QI` and `ID_STATE`. After both have confirmed, `INITO` is reported externally.
2. A remote method call on `SERVER_0_1` generates the indication event `IND`, which is placed directly on `TRIGGER.E1` – the adapter thus reports the call inward to the downstream logic.

3. As soon as the downstream logic delivers a feedback event `TRIGGER.EI1` with data `TRIGGER.DI1` via the adapter, this clocks the internal **E_D_FF** flip-flop, which captures and holds the value stably.
4. The flip-flop's output `EO` triggers `STATE_CLIENT.REQ`; the buffered value is passed as `SD_1` and sent as an OPC UA write to the remote node configured in `ID_STATE`.
5. `AND_QO` (AND_BOOL_2) ANDs the `QO` outputs of `SERVER_0_1` and `STATE_CLIENT`; every confirmation additionally triggers `CNF` externally.

Technical Features

- **Two network connections behind one adapter**: `SERVER_0` (receiving, server role) and `CLIENT_1_0` (state, remote write) are combined into a single bidirectional AE_AX interface.
- **Buffering with D flip-flop**: The feedback value reported via the adapter is stabilized via an internal `iec61499::events::E_D_FF` before being sent via `CLIENT_1_0`.
- **Sequential initialization**: `SERVER_0_1` is initialized before `STATE_CLIENT` (composite FBTypes chain `INIT`/`INITO` serially through each instance).
- **Encapsulation**: Only the AE_AX adapter interface is visible externally; `SERVER_0`'s `IND` and `CLIENT_1_0`'s `REQ`/`SD_1` remain internal.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: Both connections (server and state client) are established.
3. **Receive Active**: A remote method call is reported to the internal logic via the adapter.
4. **State Write**: A feedback value from the internal logic is buffered and sent via remote write to the configured target node.

Application Scenarios

- Receiving a remote trigger call while simultaneously writing the resulting state back to a different remote node over a single adapter connection
- Server-side counterpart to AE_AX_CLIENT_0_SUBSCRIBE_1 in distributed control architectures

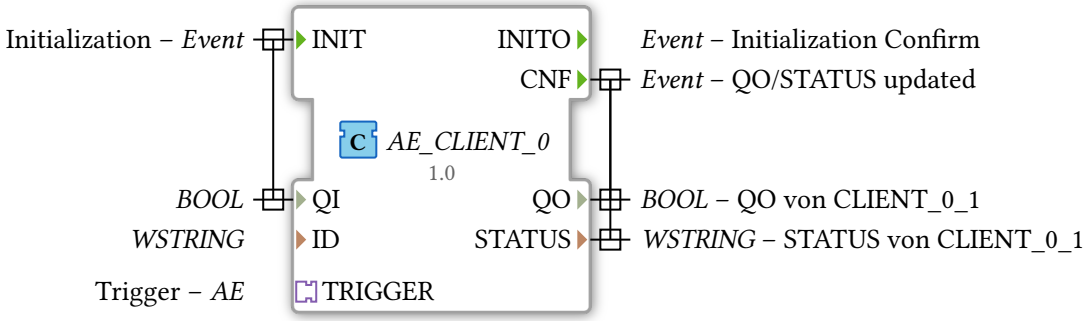
Comparison with Similar Function Blocks

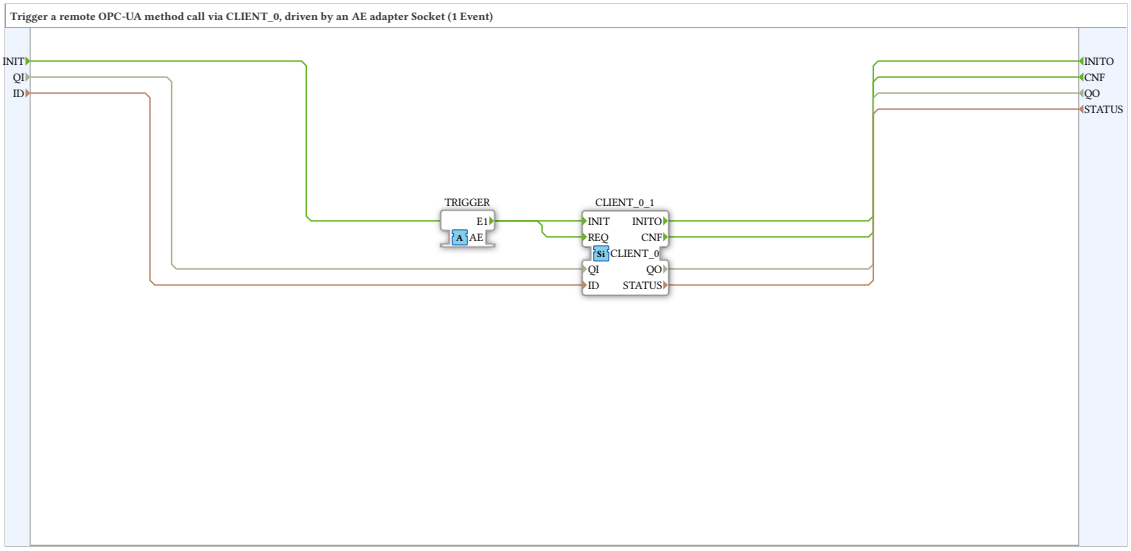
- **AE_AX_CLIENT_0_SUBSCRIBE_1**: The client/calling counterpart – actively triggers instead of receiving, subscribes instead of writing.
- **AX_CLIENT_1_0**: A pure remote write without a receiving side.
- **ASR_AX_SERVER_0_CLIENT_1_0** / **ASRT_AX_SERVER_0_CLIENT_1_0**: Extend the same pattern to 2 and 3 parallel receiving servers.

Conclusion

AE_AX_SERVER_0_CLIENT_1_0 bundles a receiving server and a state-writing client behind a single bidirectional AE_AX adapter, forming the server-side counterpart to AE_AX_CLIENT_0_SUBSCRIBE_1.

AE_CLIENT_0





• – ◦ • – ◦ • – ◦ •

Introduction

The **AE_CLIENT_0** function block is a composite function block that encapsulates the network-based `CLIENT_0` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AE adapter**. Every event arriving at the adapter socket `TRIGGER` triggers a remote OPC UA **method call** via `CLIENT_0` at the address configured in `ID`.

Unlike `CLIENT_1_0` (a remote write of a data value), `CLIENT_0` calls a remote **method without payload data** – the appropriate building block for event-driven trigger actions such as “Set”, “Reset”, or “Toggle” on a remote node.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that the method call has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, opens (TRUE) or closes (FALSE) the connection to the server
- **ID** (WSTRING): Remote method address for the call

Data Outputs

- **QO** (BOOL): Qualifier output, connection status (identical to `CLIENT_0_1.QO`)
- **STATUS** (WSTRING): Status information as a Unicode string (identical to `CLIENT_0_1.STATUS`)

Adapter

Adapter	Type	Direction	Description
TRIGGER	adapter::types::unidirectional::AE	Socket (Input)	Triggering event for the method call

Functionality

1. The internal `CLIENT_0_1` block is initialized with `QI` and `ID` via the `INIT` event; it establishes the connection to the remote server. Upon success, it returns `INITO`.
2. Every event arriving at the adapter socket `TRIGGER` (`TRIGGER.E1`) directly triggers the send event `REQ` of the internal `CLIENT_0_1` block.
3. `CLIENT_0_1` then calls the remote method configured in `ID` and confirms with `CNF`, which is sent out.

Technical Features

- **Pure trigger, no payload**: Unlike `AX_CLIENT_1_0`, `AE_CLIENT_0` transfers no data value – the adapter only supplies the triggering event.
- **Encapsulation**: The original event/data interface of `CLIENT_0` (including `REQ`) is moved internally; only the AE adapter interface is visible externally.
- The base building block from which ASR_CLIENT_0 (2 triggers) and ASRT_CLIENT_0 (3 triggers) are derived by wiring multiple `CLIENT_0` instances in parallel.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The connection to the remote server is established, the block is ready to trigger method calls.
3. **Send Active**: An event arriving at the AE socket triggers the remote method call via the internal `CLIENT_0` block.

Application Scenarios

- Remotely triggering a single remote action (e.g. “Start”, “Trigger”) without an associated data value
- Building block for composed multi-trigger blocks such as ASR_CLIENT_0 and ASRT_CLIENT_0
- Network coupling in AE-adapter-based applications

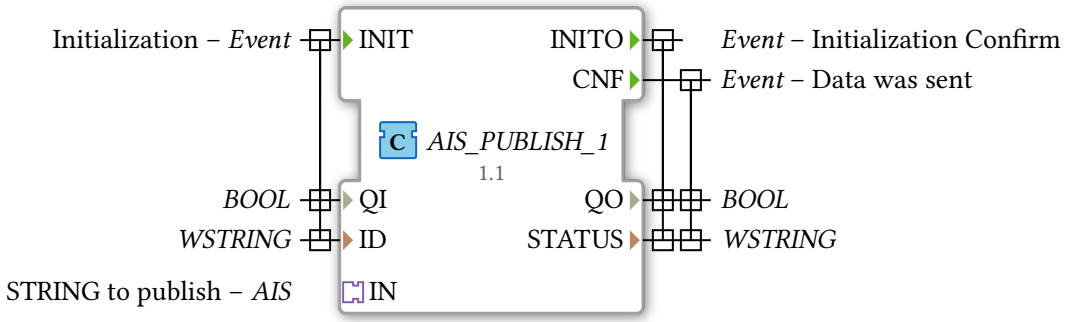
Comparison with Similar Function Blocks

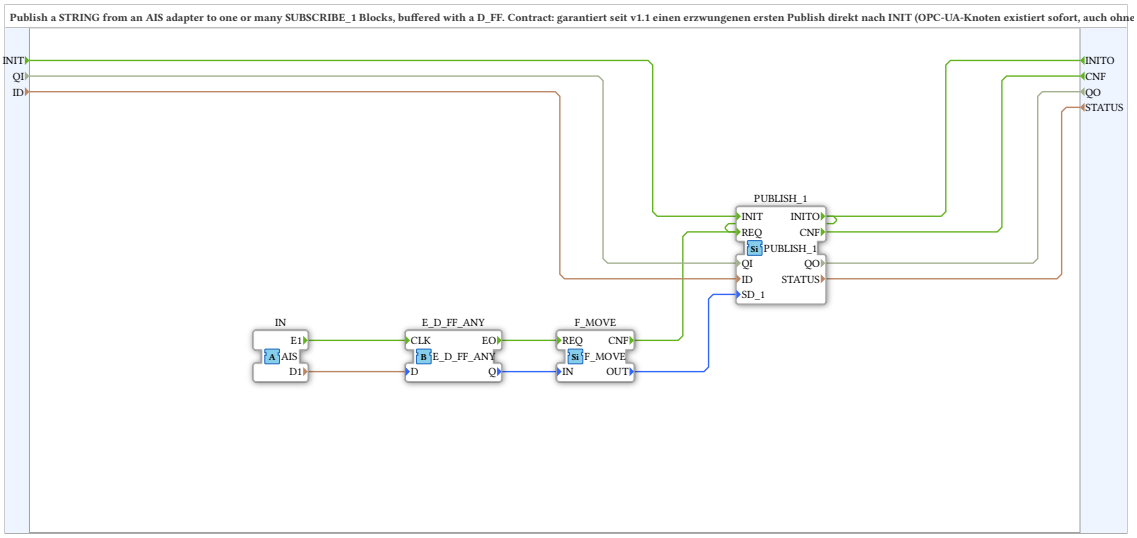
- **CLIENT_0**: The standard building block exposes its interface as events/variables (`REQ`, ...). **AE_CLIENT_0** encapsulates this building block and only provides the AE adapter.
- **AX_CLIENT_1_0**: Structurally similar, but encapsulates `CLIENT_1_0` (remote write of a data value) instead of `CLIENT_0` (pure method call without data).
- **ASR_CLIENT_0** / **ASRT_CLIENT_0**: Extend the same pattern to 2 and 3 parallel triggers (Set/Reset and Set/Reset/Toggle, respectively).

Conclusion

AE_CLIENT_0 combines the trigger semantics of the standard function block `CLIENT_0` with adapter-based event processing. The simplified AE interface makes the function block ideal for directly triggering a remote OPC UA method call in fully adapter-based IEC 61499 applications.

AIS_PUBLISH_1





• – ○ • – ○ • – ○ •

The **AIS_PUBLISH_1** function block is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AIS adapter**. A **STRING** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`
- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier
- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AIS	Socket (Input)	STRING value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AIS socket `IN` sends an event to `IN.E1`, the corresponding STRING value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then sent out.

- **Buffering with D-Flip-Flop**: The string value to be sent is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AIS adapter interface remains visible externally.

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AIS socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

- **Network Coupling in AIS Applications**: Sending STRING signals (e.g., switching commands or status bits) from the adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AIS adapters.
- **PUBLISH_1**: The standard function block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AIS_PUBLISH_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AIS adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a STRING value at the adapter level but lacks network functionality. **AIS_PUBLISH_1** combines buffering and publishing in a single function block.

PUBLISH_1 PUBLISH_1 ## Conclusion

AIS_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based string processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AIS interface makes the function block ideal for fully adapter-based IEC 61499 applications.

Technical Features

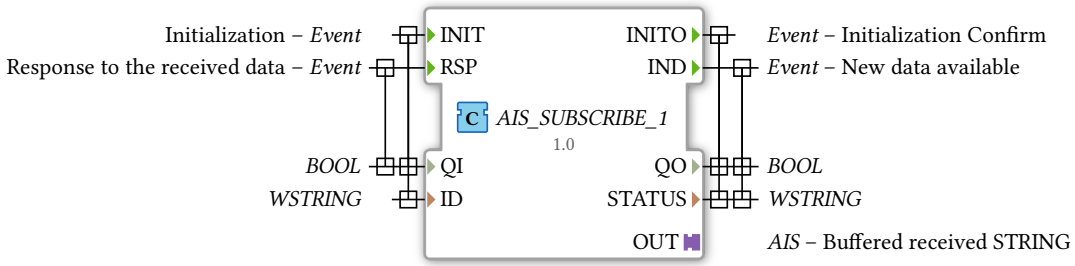
State Overview

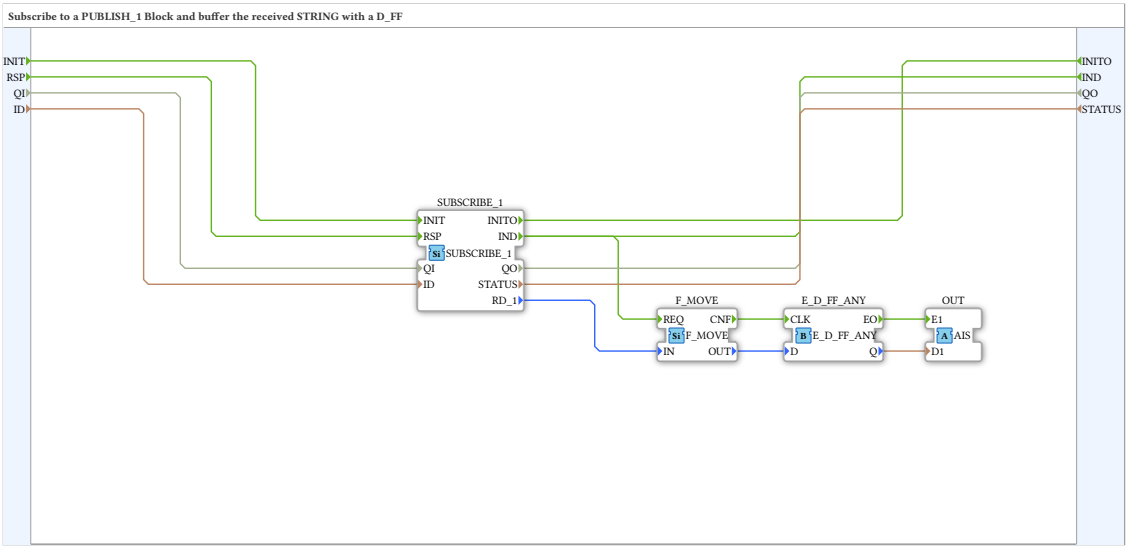
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AIS_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AIS_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AIS adapter**. It subscribes to a publish channel configured via `ID` and provides the received **STRING** value, buffered via a D flip-flop (E_D_FF_ANY), at the adapter plug `OUT`. This allows for clean integration of network communication and STRING signal processing in the AIS adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AIS	Plug (Output)	Buffered received STRING value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.

5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received STRING value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the AIS adapter interface is visible externally.

Status Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Reception**: At `IND`, the received STRING value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AIS Applications**: Receiving STRING signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AIS adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a clean fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on AIS adapters.

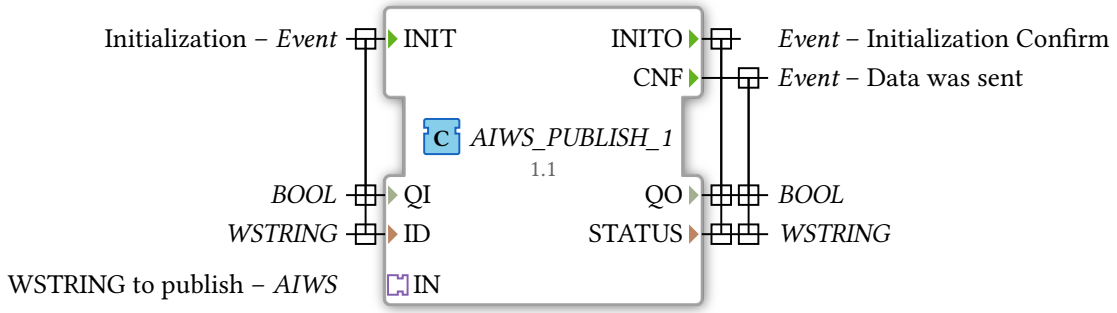
Comparison with Similar Function Blocks

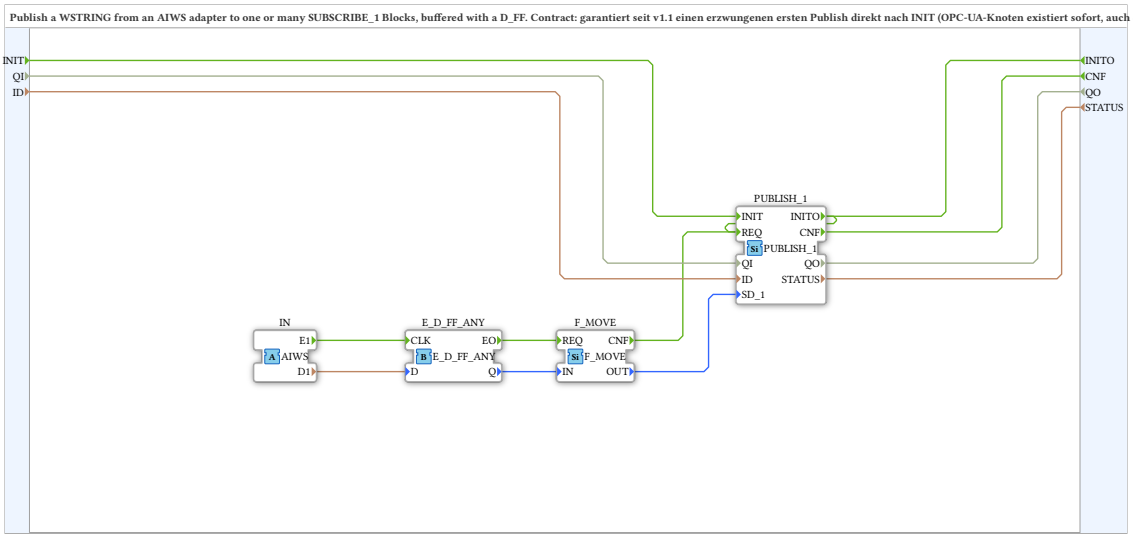
- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AIS_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AIS adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a STRING value at the adapter level but lacks network functionality. **AIS_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

Conclusion

AIS_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based string processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AIS interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AIWS_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AIWS_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AIWS adapter**. A **WSTRING** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AIWS	Socket (Input)	WSTRING value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AIWS socket `IN` delivers an event to `IN.E1`, the WSTRING value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then sent out.

Technical Features

- **Buffering with D-Flip-Flop**: The WSTRING value to be transmitted is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AIWS adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data over the configured channel.
3. **Send Active**: An event arriving at the AIWS socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in AIWS Applications**: Sending WSTRING signals (e.g., switching commands or status bits) from adapter-based logic to remote receivers via Publish/Subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AIWS adapters.

Comparison with Similar Blocks

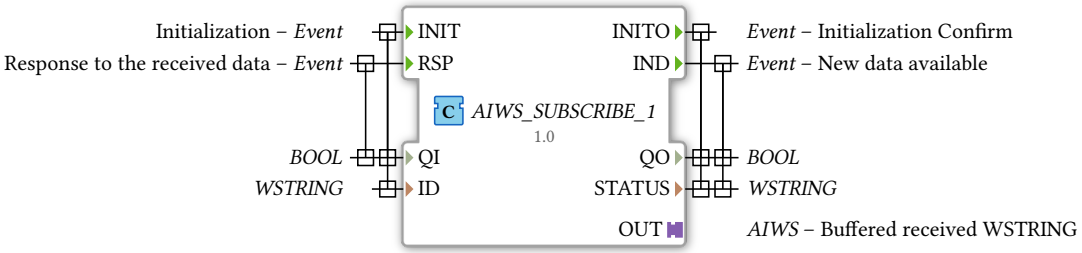
- **PUBLISH_1**: The standard block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AIWS_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AIWS adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a WSTRING value at the adapter level but lacks network functionality. **AIWS_PUBLISH_1** combines buffering and publishing in a single block.

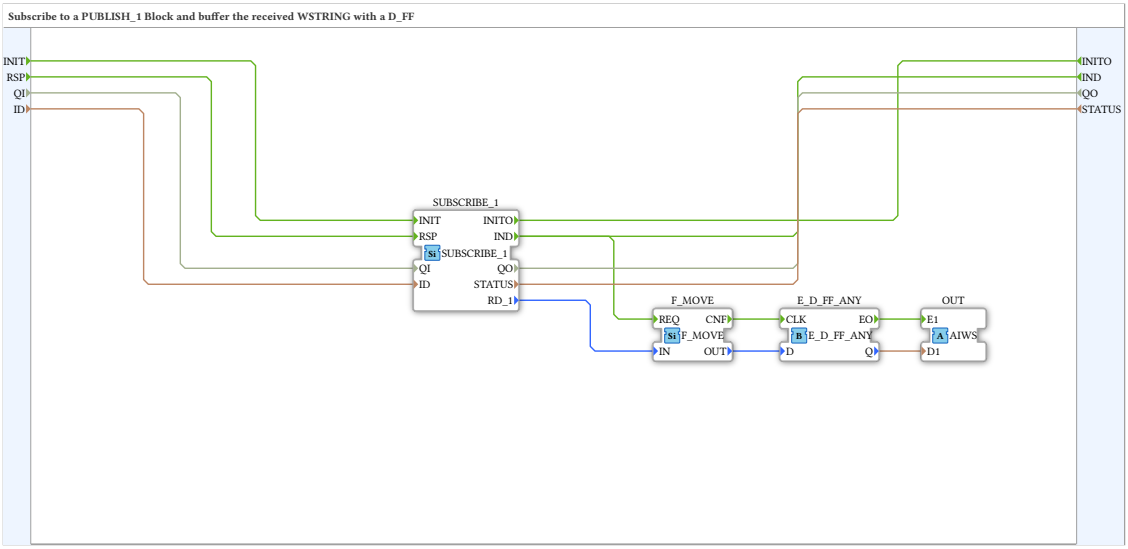
Conclusion

AIWS_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based WSTRING processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AIWS interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AIWS_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based WSTRING processing.

AIWS_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AIWS_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AIWS adapter**. It subscribes to a publish channel configured via `ID` and provides the received **WSTRING** value, buffered via a D flip-flop (E_D_FF_ANY), at the adapter plug `OUT`. This allows for clean integration of network communication and WSTRING signal processing within the AIWS adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AIWS	Plug (Output)	Buffered received WSTRING value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.

4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.
5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received WSTRING value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the AIWS adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Received**: At `IND`, the received WSTRING value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AIWS Applications**: Receiving WSTRING signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AIWS adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a cleaner fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on AIWS adapters.

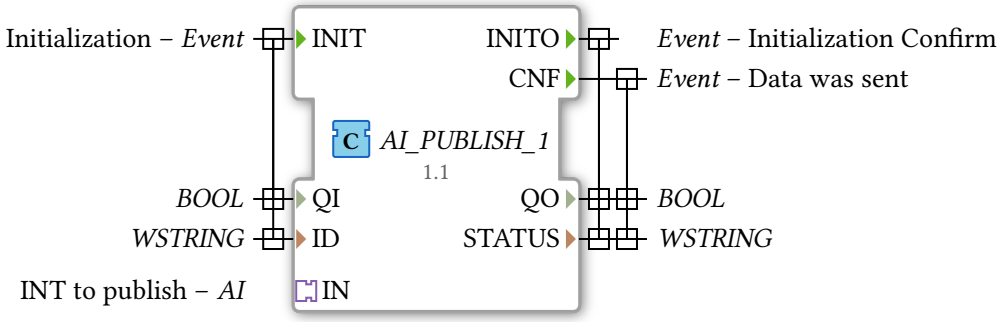
Comparison with Similar Function Blocks

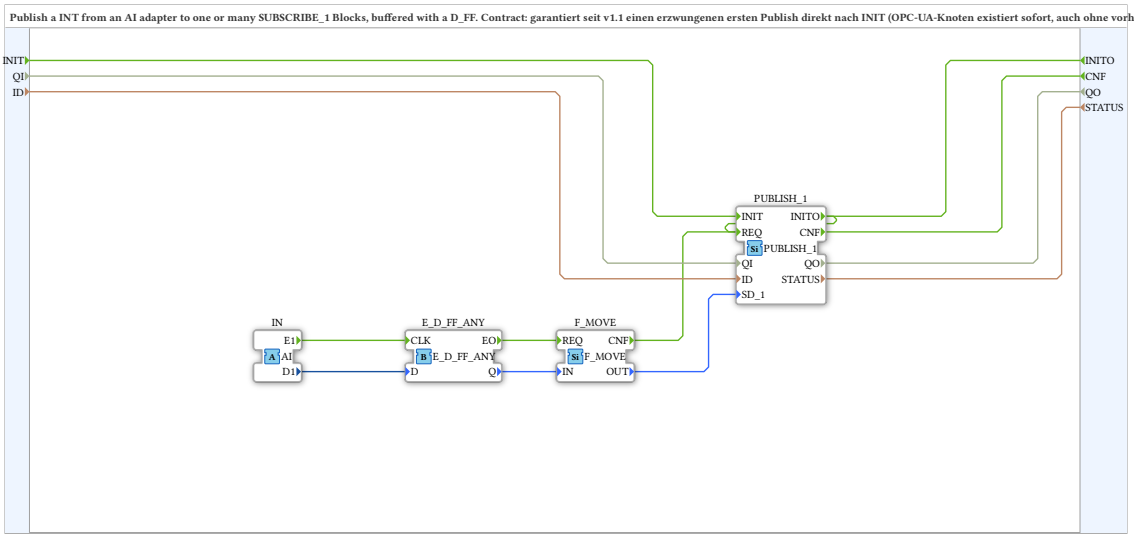
- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AIWS_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AIWS adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a WSTRING value at the adapter level but lacks network functionality. **AIWS_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

Conclusion

AIWS_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based WSTRING processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AIWS interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AI_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AI_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AI adapter**. An **INT** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AI	Socket (Input)	INT value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AI socket `IN` delivers an event to `IN.E1`, the INT value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then transmitted externally.

Technical Features

- **Buffering with D-Flip-Flop**: The INT value to be transmitted is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AI adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AI socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in AI Applications**: Sending INT signals (e.g., switching commands or status bits) from adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AI adapters.

Comparison with Similar Blocks

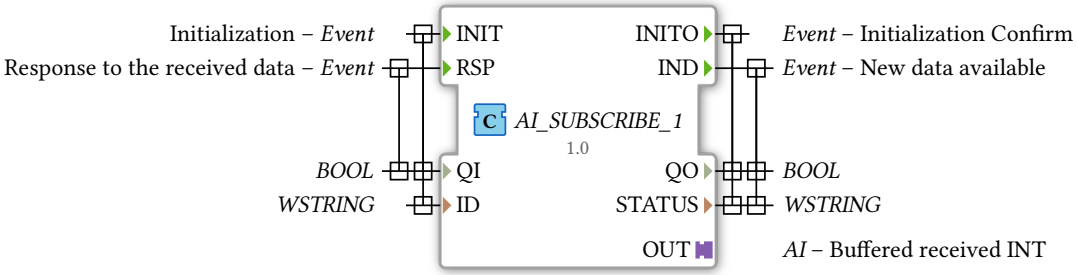
- **PUBLISH_1**: The standard block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AI_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes an INT value at the adapter level but lacks network functionality. **AI_PUBLISH_1** combines buffering and publishing in a single block.

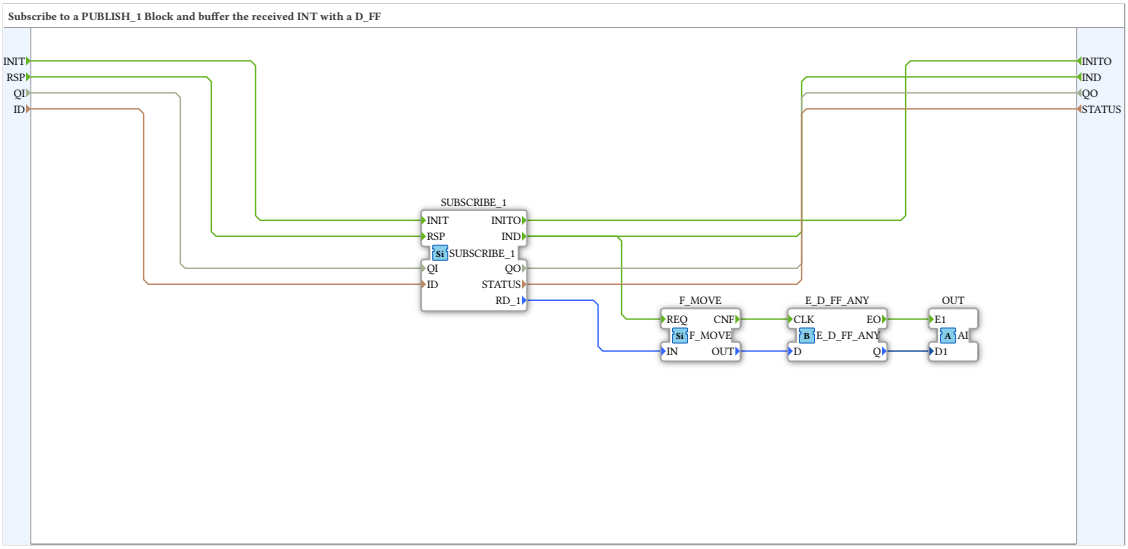
Conclusion

AI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based INT processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AI interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based INT processing.

AI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AI_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AI adapter**. It subscribes to a publish channel configured via `ID` and provides the received **INT** value, buffered via a D flip-flop (E_D_FF_ANY), at the adapter plug `OUT`. This allows for the clean integration of network communication and INT signal processing in the AI adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AI	Plug (Output)	Buffered received INT value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.

5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received INT value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the AI adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Receipt**: At `IND`, the received INT value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AI Applications**: Receiving INT signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AI adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a clean fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AI adapters.

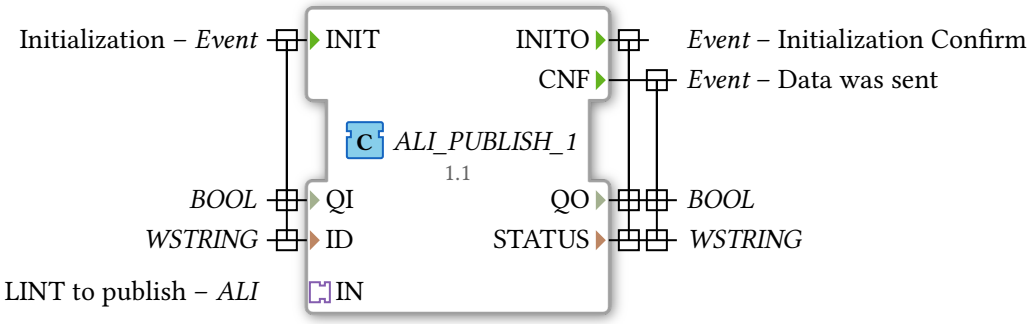
Comparison with Similar Building Blocks

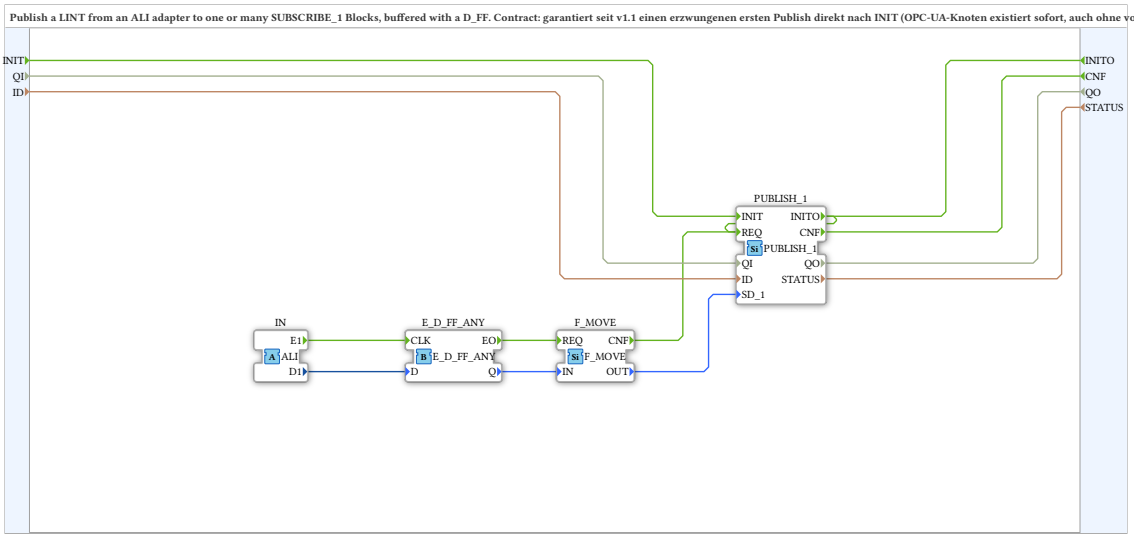
- **SUBSCRIBE_1**: The standard building block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AI_SUBSCRIBE_1** encapsulates this building block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes an INT value at the adapter level but lacks network functionality. **AI_SUBSCRIBE_1** combines subscribing and buffering in a single building block.

Conclusion

AI_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based INT processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AI interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

ALI_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The **ALI_PUBLISH_1** function block is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **ALI adapter**. A **LINT** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::ALI	Socket (Input)	LINT value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the ALI socket `IN` sends an event to `IN.E1`, the LINT value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then transmitted externally.

Technical Features

- **Buffering with D-Flip-Flop**: The LINT value to be transmitted is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the ALI adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the ALI socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in ALI Applications**: Sending LINT signals (e.g., switching commands or status bits) from the adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on ALI adapters.

Comparison with Similar Function Blocks

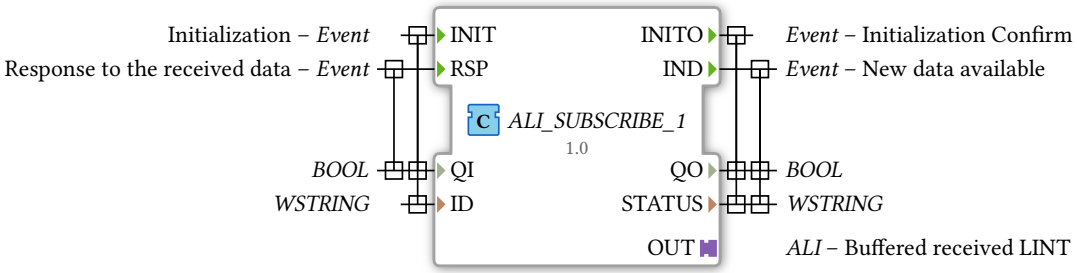
- **PUBLISH_1**: The standard function block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **ALI_PUBLISH_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the ALI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a LINT value at the adapter level but lacks network functionality. **ALI_PUBLISH_1** combines buffering and publishing in a single function block.

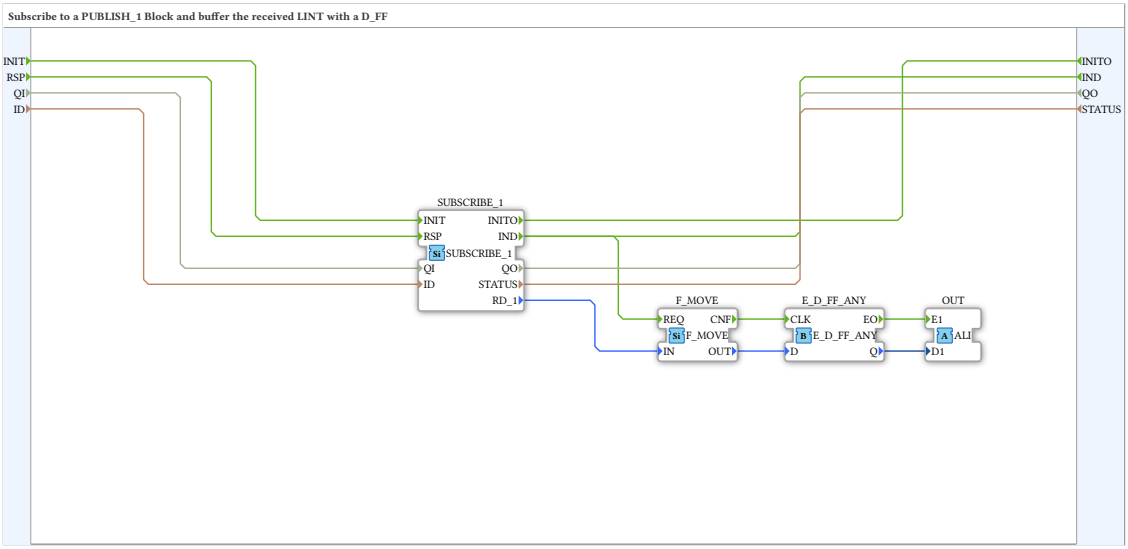
Conclusion

ALI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based LINT processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified ALI interface makes the function block ideal for fully adapter-based IEC 61499 applications.

ALI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based LINT processing.

ALI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **ALI_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **ALI adapter**. It subscribes to a publish channel configured via `ID` and provides the received **LINT** value, buffered via a D flip-flop (E_D_FF_ANY), at the adapter plug `OUT`. This allows for the clean integration of network communication and LINT signal processing in the ALI adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::ALI	Plug (Output)	Buffered received LINT value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.

5. The event EO of the flip-flop is transferred to the event output OUT.E1, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received LINT value is buffered via an internal iec61499::events::E_D_FF_ANY and remains stable at the output until the next value is received. EO only fires when Q changes.
- **RSP is a no-op for subscribers**: According to source code analysis of 4diac-forte (CCommFB::sendData()), the RSP event in SUBSCRIBE_1 (service type subscriber) does not execute any action – the branch != e_Subscriber is skipped. The event input pair INIT/RSP is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of SUBSCRIBE_1 (including IND/RD_1) is moved internally; only the ALI adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the INIT event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Reception**: At IND, the received LINT value is transferred to the internal D flip-flop and output via OUT.

Application Scenarios

- **Network Coupling in ALI Applications**: Receiving LINT signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via ALI adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with AX_PUBLISH_1 and a true event-based E_D_FF_ANY, the feedback between SUBSCRIBE_1 and PUBLISH_1 can be decoupled (a cleaner fix compared to the interim fix with AX_D_FF).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on ALI adapters.

Comparison with Similar Function Blocks

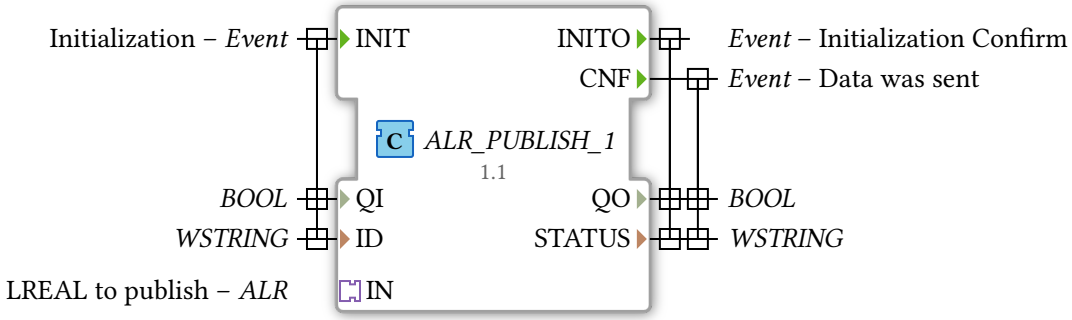
- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (IND, RD_1, ...). **ALI_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (E_D_FF_ANY), and only provides the ALI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a LINT value at the adapter level but lacks network functionality. **ALI_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

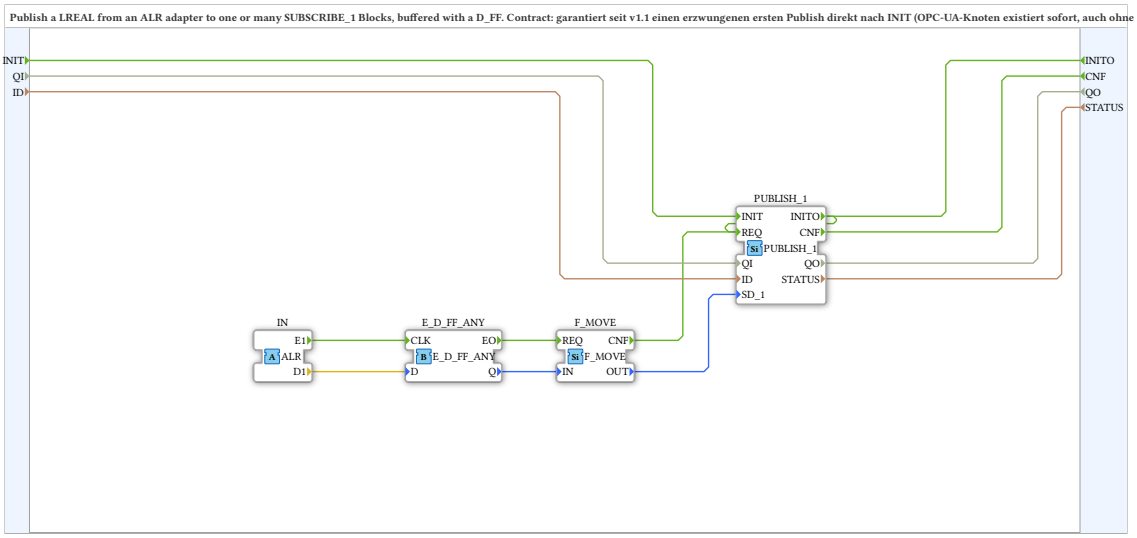
Conclusion

ALI_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block SUBSCRIBE_1 with adapter-based LINT processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified ALI interface makes the function block ideal for fully adapter-based IEC 61499 applications.

ALI_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block SUBSCRIBE_1 with adapter-based LINT processing.

ALR_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The **ALR_PUBLISH_1** function block is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **ALR adapter**. An **LREAL** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::ALR	Socket (Input)	LREAL value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the ALR socket `IN` sends an event to `IN.E1`, the LREAL value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then sent externally.

Technical Features

- **Buffering with D-Flip-Flop**: The LREAL value to be transmitted is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the ALR adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the ALR socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in ALR Applications**: Sending LREAL signals (e.g., switching commands or status bits) from the adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between the user's own publications and subscriptions is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on ALR adapters.

Comparison with Similar Blocks

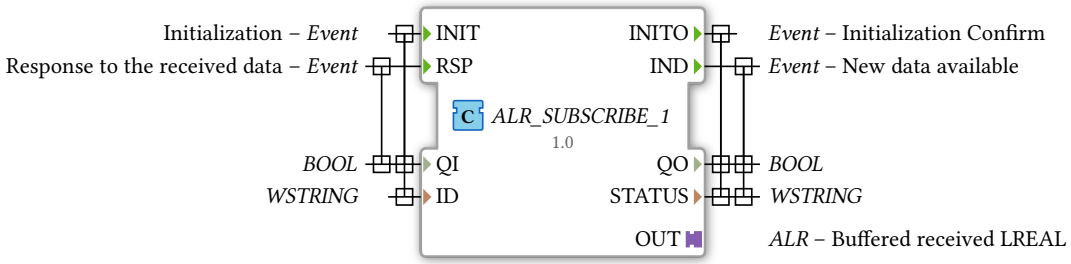
- **PUBLISH_1**: The standard block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **ALR_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the ALR adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes an LREAL value at the adapter level but lacks network functionality. **ALR_PUBLISH_1** combines buffering and publishing in a single block.

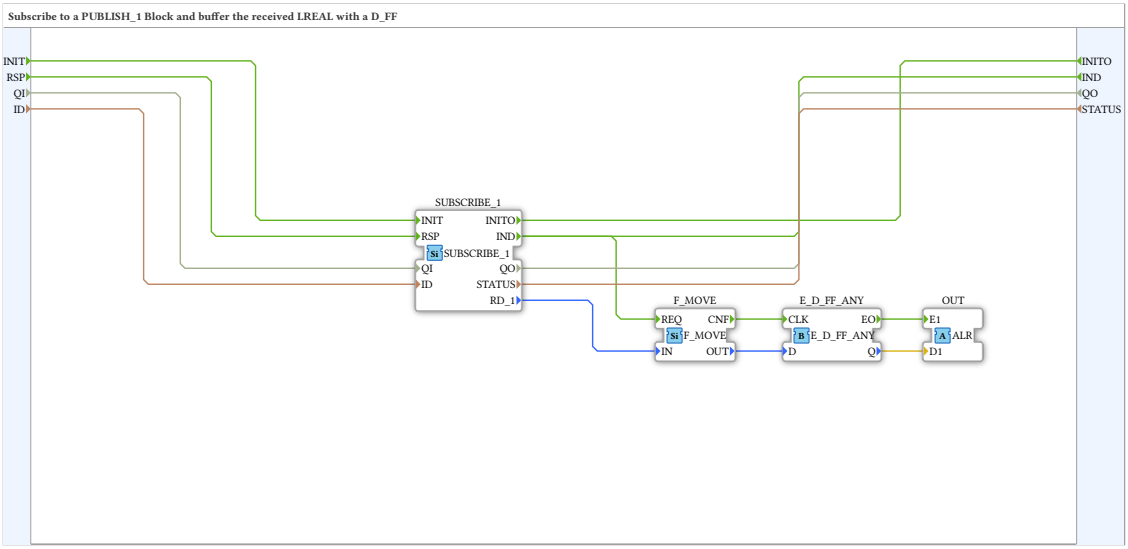
Conclusion

ALR_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based LREAL processing. Internal D flip-flop buffering stabilizes the transmit value and decouples feedback loops; the simplified ALR interface makes the function block ideal for fully adapter-based IEC 61499 applications.

ALR_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based LREAL processing.

ALR_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **ALR_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **ALR adapter**. It subscribes to a publish channel configured via `ID` and provides the received **LREAL** value, buffered via a D flip-flop (E_D_FF_ANY), at the adapter plug `OUT`. This allows for clean integration of network communication and LREAL signal processing in the ALR adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::ALR	Plug (Output)	Buffered received LREAL value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.

5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received LREAL value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the ALR adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Receipt**: At `IND`, the received LREAL value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in ALR Applications**: Receiving LREAL signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via ALR adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a clean fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on ALR adapters.

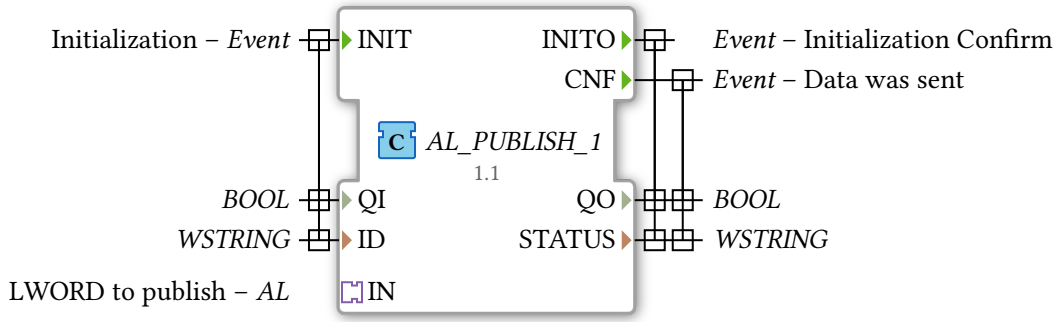
Comparison with Similar Function Blocks

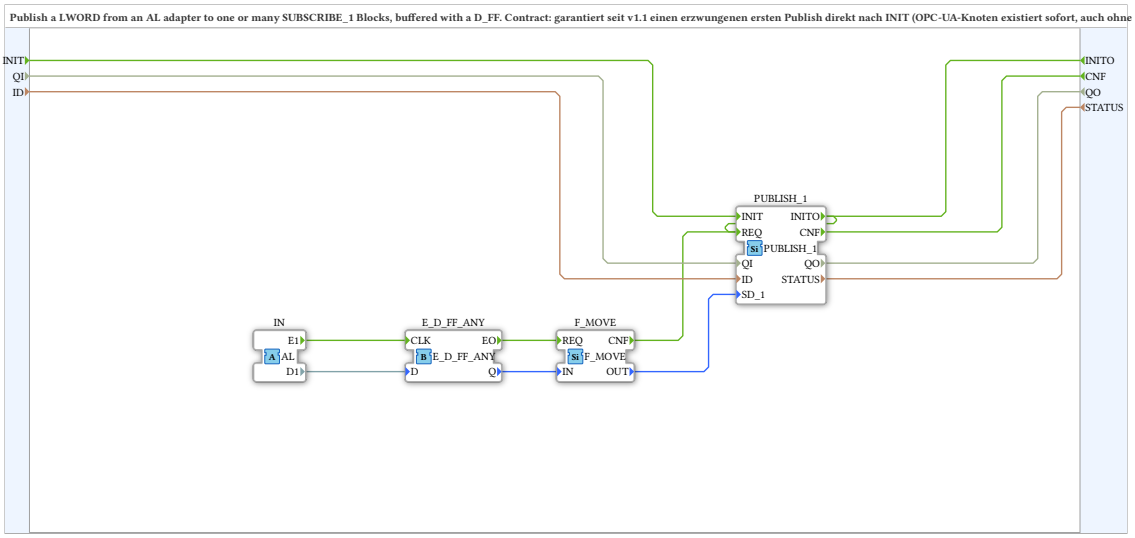
- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (`IND`, `RD_1`, ...). **ALR_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the ALR adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes an LREAL value at the adapter level but lacks network functionality. **ALR_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

Conclusion

ALR_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based LREAL processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified ALR interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AL_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AL_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AL adapter**. An **LWORD** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AL	Socket (Input)	LWORD value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AL socket `IN` sends an event to `IN.E1`, the LWORD value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then sent out.

Technical Features

- **Buffering with D-Flip-Flop**: The LWORD value to be sent is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents an input value that changes during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AL adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AL socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in AL Applications**: Sending LWORD signals (e.g., switching commands or status bits) from adapter-based logic to remote receivers via Publish/Subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AL adapters.

Comparison with Similar Blocks

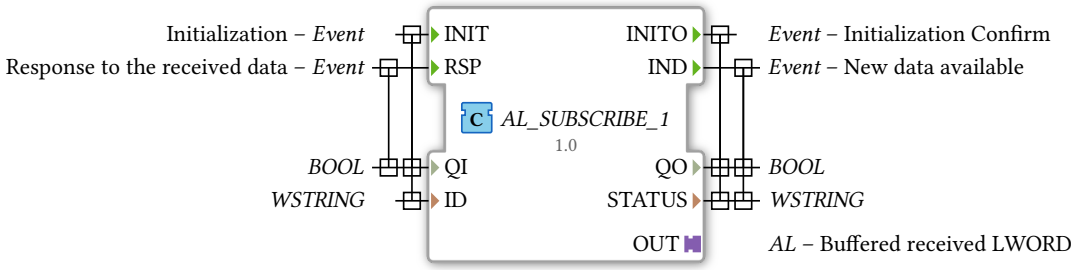
- **PUBLISH_1**: The standard block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AL_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AL adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes an LWORD value at the adapter level but lacks network functionality. **AL_PUBLISH_1** combines buffering and publishing in a single block.

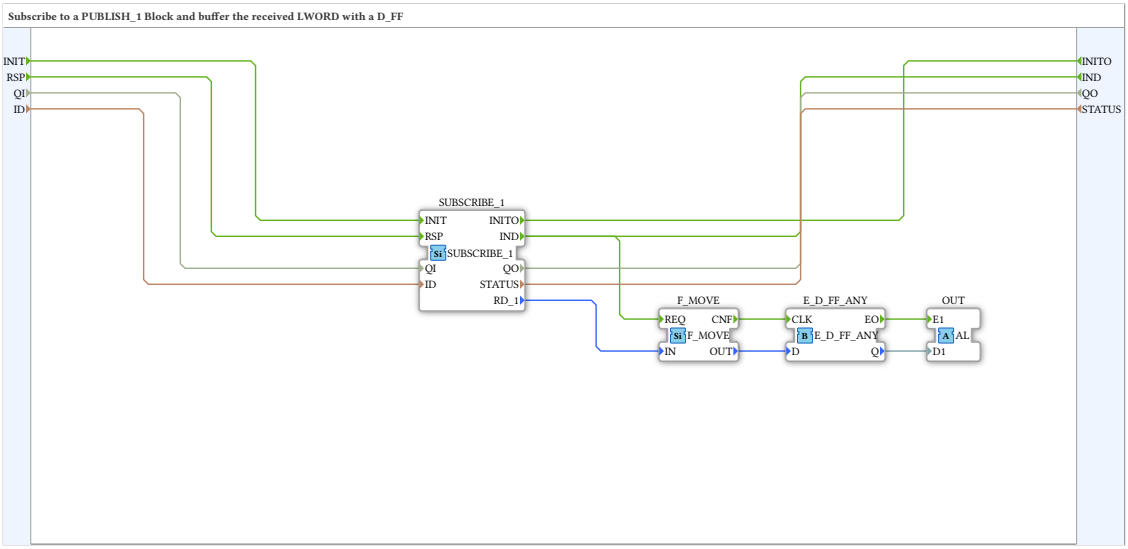
Conclusion

AL_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based LWORD processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AL interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AL_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based LWORD processing.

AL_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AL_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AL adapter**. It subscribes to a publish channel configured via `ID` and makes the received **LWORD** value buffered via a D flip-flop (E_D_FF_ANY) available at the adapter plug `OUT`. This allows for clean integration of network communication and LWORD signal processing in the AL adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AL	Plug (Output)	Buffered received LWORD value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.

4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.
5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received LWORD value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the AL adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Received**: At `IND`, the received LWORD value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AL Applications**: Receiving LWORD signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AL adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a clean fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on AL adapters.

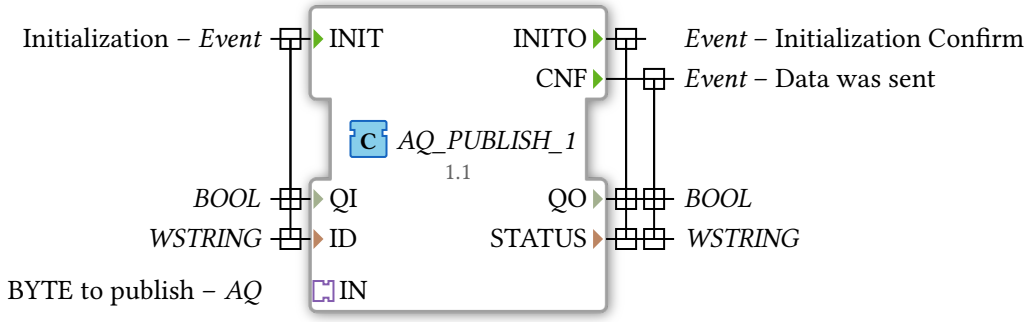
Comparison with Similar Function Blocks

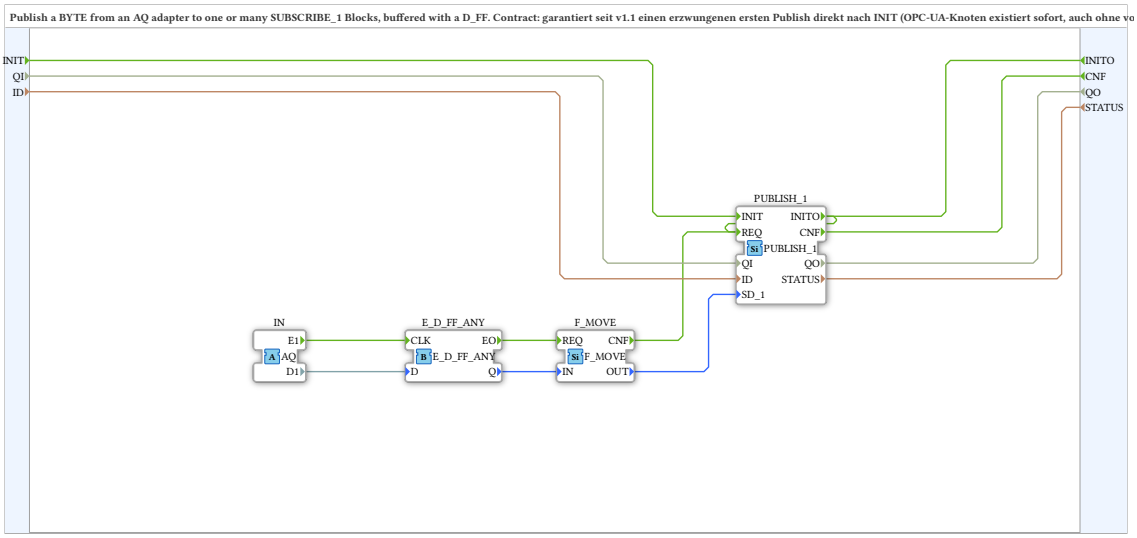
- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AL_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AL adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes an LWORD value at the adapter level but lacks network functionality. **AL_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

Conclusion

AL_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based LWORD processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AL interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AQ_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AQ_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AQ adapter**. A **BYTE** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AQ	Socket (Input)	Byte value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AQ socket `IN` sends an event to `IN.E1`, the corresponding BYTE value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then sent externally.

Technical Features

- **Buffering with D-Flip-Flop**: The byte value to be sent is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AQ adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AQ socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in AQ Applications**: Sending BYTE signals (e.g., switching commands or status bits) from the adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AQ adapters.

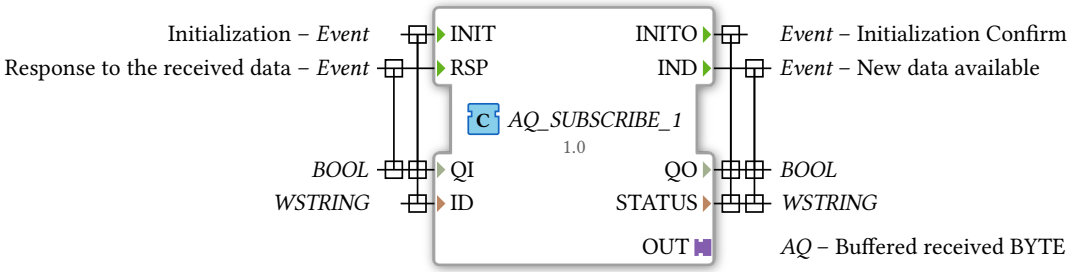
Comparison with Similar Blocks

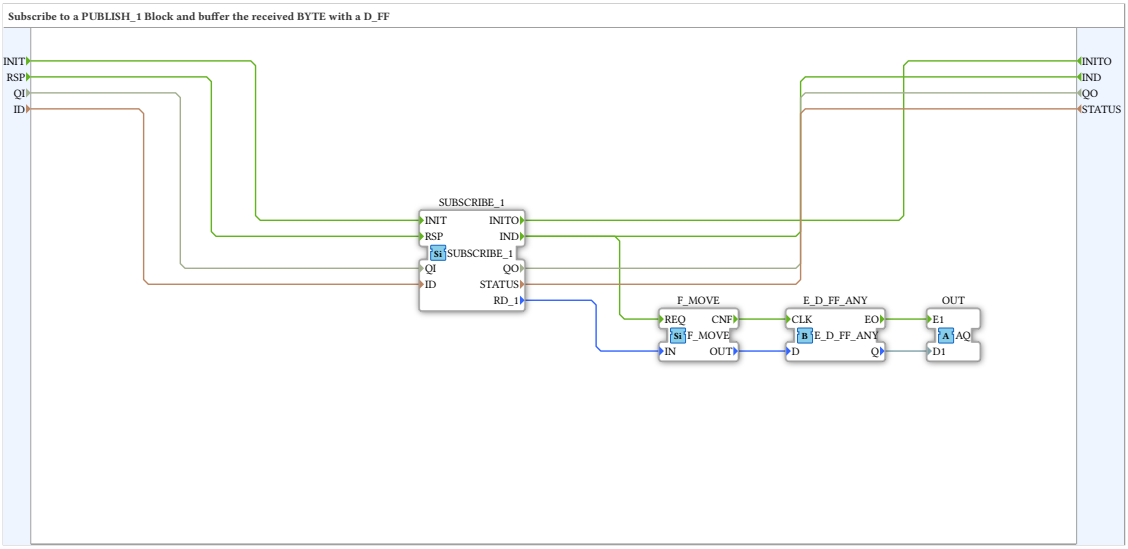
- **PUBLISH_1**: The standard block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AQ_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AQ adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a byte value at the adapter level but lacks network functionality. **AQ_PUBLISH_1** combines buffering and publishing in a single block.

Conclusion

AQ_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based BYTE processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AQ interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AQ_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AQ_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AQ adapter**. It subscribes to a publish channel configured via `ID` and makes the received **BYTE** value buffered via a D flip-flop (E_D_FF_ANY) available on the adapter plug `OUT`. This allows for clean integration of network communication and BYTE signal processing in the AQ adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AQ	Plug (Output)	Buffered received BYTE value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.

5. The event EO of the flip-flop is transferred to the event output OUT.E1, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received BYTE value is buffered via an internal iec61499::events::E_D_FF_ANY and remains stable at the output until the next value is received. EO only fires when Q changes.
- **RSP is a no-op for subscribers**: According to source code analysis of 4diac-forte (CCommFB::sendData()), the RSP event in SUBSCRIBE_1 (service type subscriber) does not execute any action – the branch != e_Subscriber is skipped. The event input pair INIT/RSP is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of SUBSCRIBE_1 (including IND/RD_1) is moved internally; only the AQ adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the INIT event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Received**: At IND, the received byte value is transferred to the internal D flip-flop and output via OUT.

Application Scenarios

- **Network Coupling in AQ Applications**: Receiving BYTE signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AQ adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with AX_PUBLISH_1 and a true event-based E_D_FF_ANY, the feedback between SUBSCRIBE_1 and PUBLISH_1 can be decoupled (a cleaner fix compared to the interim fix with AX_D_FF).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on AQ adapters.

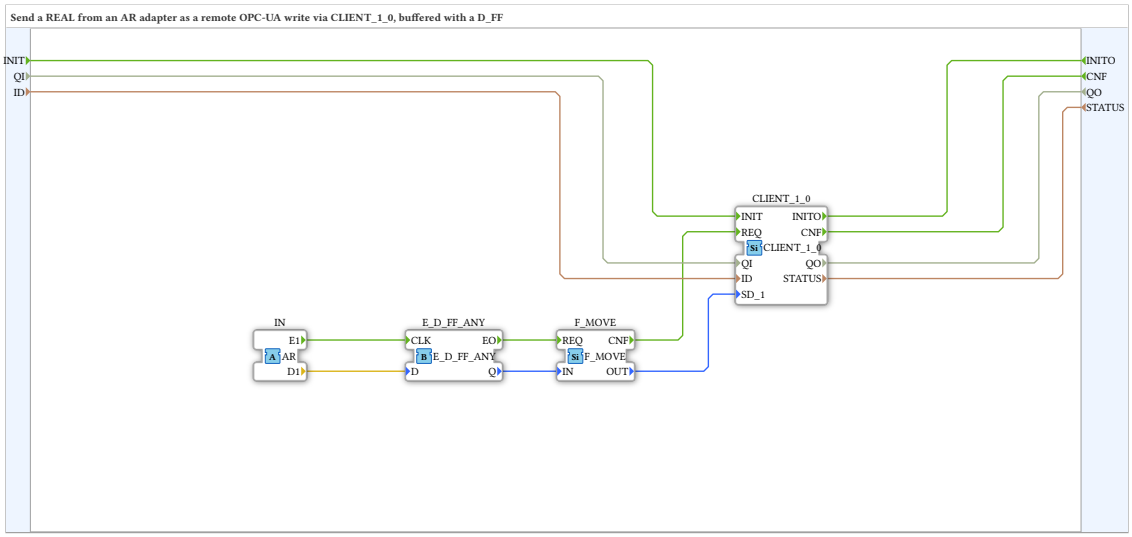
Comparison with Similar Components

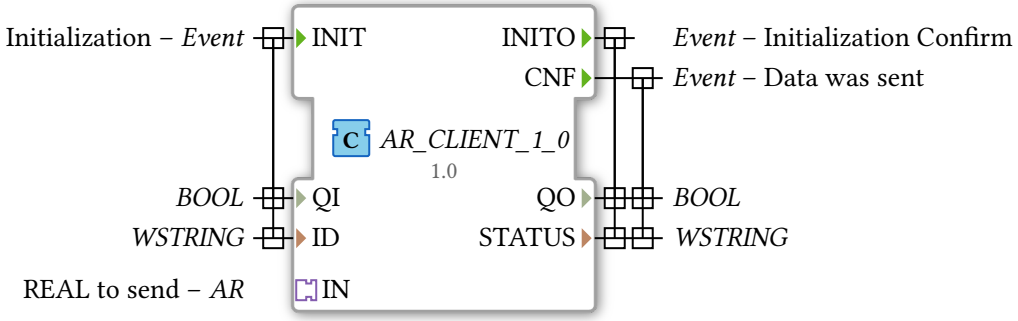
- **SUBSCRIBE_1**: The standard component exposes its interface as events/variables (IND, RD_1, …). **AQ_SUBSCRIBE_1** encapsulates this component, buffers the value with a D flip-flop (E_D_FF_ANY), and only provides the AQ adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a byte value at the adapter level but lacks network functionality. **AQ_SUBSCRIBE_1** combines subscribing and buffering in a single component.

Conclusion

AQ_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block SUBSCRIBE_1 with adapter-based BYTE processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AQ interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AR_CLIENT_1_0





• – ◦ • – ◦ • – ◦ •

Introduction

The **AR_CLIENT_1_0** function block is a composite function block that encapsulates the network-based `CLIENT_1_0` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AR adapter** (`REAL` data type). A **REAL** value present at the adapter socket `IN` is buffered via an internal `E_D_FF_ANY` flip-flop and then sent via `CLIENT_1_0` as an OPC UA **Write** or `CALL_METHOD` invocation to the remote node configured under `ID`.

Unlike **AR_PUBLISH_1** (local publish/subscribe), `CLIENT_1_0` actively writes to a **remote** server.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, connected to `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, connected to `QO` and `STATUS`
- **CNF** (Event): Confirmation that data was sent, connected to `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Input qualifier, opens (TRUE) or closes (FALSE) the connection to the server
- **ID** (WSTRING): Connection identifier (OPC UA address of the target node, e.g., `opc_ua[WRITE;opc.tcp://192.168.1.12:4840#;...]`)

Data Outputs

- **QO** (BOOL): Output qualifier, connection status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapters

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AR	Socket (Input)	REAL value to be sent

Functionality

1. The `INIT` event initializes the internal `CLIENT_1_0` block with `QI` and `ID`, establishing the connection to the remote server and acknowledging with `INITO` upon success.
2. When the AR socket `IN` receives an event on `IN.E1`, the REAL value at `IN.D1` is latched into the internal `E_D_FF_ANY` flip-flop.
3. The flip-flop holds the value stable and generates event `EO`.
4. `EO` triggers `F_MOVE` (`REAL` data type) to supply `CLIENT_1_0.SD_1` and fire the send event `REQ` on `CLIENT_1_0`.
5. After successful transmission, `CLIENT_1_0` confirms with `CNF`.

Technical Features

- **Buffering with E_D_FF_ANY**: The REAL value to be sent is buffered using an internal `E_D_FF_ANY`, preventing value instability during transmission.
- **Remote Write / Call Method**: `CLIENT_1_0` directly addresses a remote OPC UA server to transfer analog sensor readings or setpoints.
- **Encapsulation**: The raw `CLIENT_1_0` event/data interface is kept internal; only the clean AR adapter interface is exposed.

State Overview

1. **Uninitialized**: The block waits for the `INIT` event.
2. **Initialized**: Connection to the remote server is established; ready to send.
3. **Send Active**: An event on the AR socket buffers the REAL value and triggers the remote write.

Application Scenarios

- **Remote Sensor Data Transmission**: Sending analog values (e.g. pressure, temperature, position) to a remote controller or HMI via OPC UA.
- **Modular Control Architectures**: Integrating distributed REAL values into libraries built consistently on AR adapters.

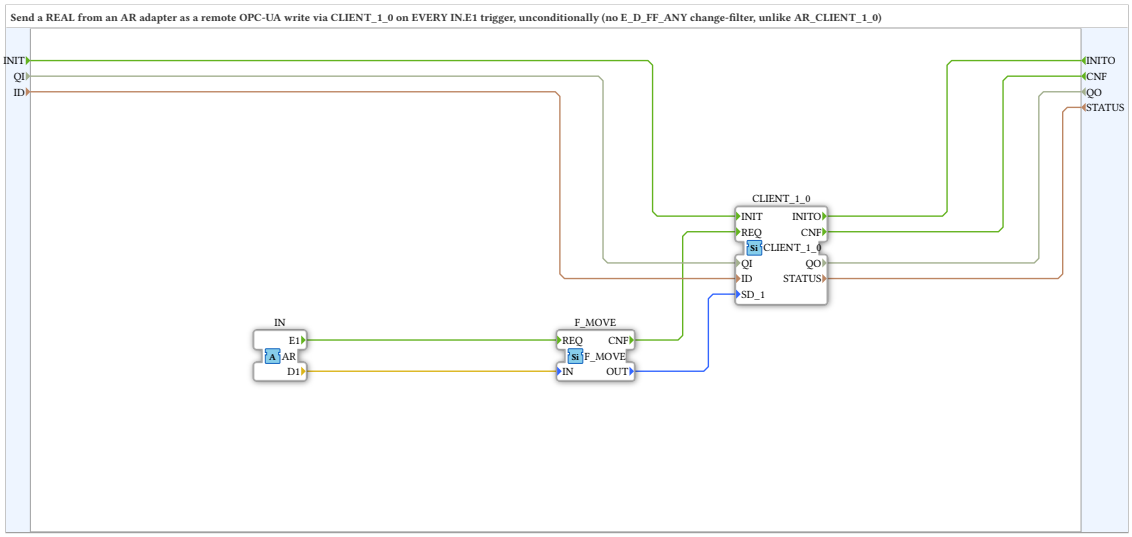
Comparison with Similar Blocks

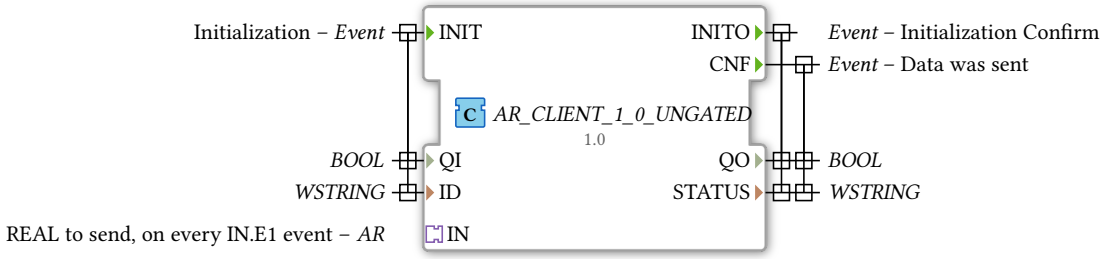
- **AR_CLIENT_1_0_UNGATED**: Omits the internal `E_D_FF_ANY` flip-flop (change filter) and triggers the remote write or method call unconditionally on every `IN.E1` event — ideal for repeated VT/HMI commits of the same value.
- **AX_CLIENT_1_0**: Identical structure, but handles BOOL values via an AX adapter.
- **ATM_CLIENT_1_0**: Identical structure, but handles TIME values via an ATM adapter.
- **AR_PUBLISH_1**: Encapsulates `PUBLISH_1` instead of `CLIENT_1_0` for local publish/subscribe.

Conclusion

AR_CLIENT_1_0 bridges connection-oriented OPC UA remote writing with adapter-based REAL data handling.

AR_CLIENT_1_0_UNGATED





• – ◦ • – ◦ • – ◦ •

Introduction

The **AR_CLIENT_1_0_UNGATED** function block is a composite function block from the `adapter::net` package. It encapsulates the network-based `CLIENT_1_0` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AR adapter** (`IN`, `REAL` data type).

Unlike **AR_CLIENT_1_0**, this block **does not** contain an internal `E_D_FF_ANY` value filter (Send-on-Change). Every incoming event on `IN.E1` triggers a send request to `CLIENT_1_0.REQ` **unconditionally and without filtering**.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, connected to `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, connected to `QO` and `STATUS`
- **CNF** (Event): Confirmation that data was sent, connected to `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Input qualifier, opens (TRUE) or closes (FALSE) the connection to the server
- **ID** (WSTRING): Connection identifier (OPC UA address of the target node, e.g., `opc_ua[CALL_METHOD;opc.tcp://192.168.1.12:4840#;...]`)

Data Outputs

- **QO** (BOOL): Output qualifier, connection status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapters

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AR	Socket (Input)	REAL value to be sent (transmitted on every `IN.E1` trigger)

Functionality

1. The `INIT` event initializes the internal `CLIENT_1_0` block with `QI` and `ID`, acknowledging with `INITO` upon success.
2. When the AR socket `IN` receives an event on `IN.E1`, the event is forwarded **directly** to `F_MOVE.REQ`.
3. `F_MOVE` (`REAL`) receives the REAL value present at `IN.D1`, passes it to `CLIENT_1_0.SD_1`, and triggers `CLIENT_1_0.REQ`.
4. The `CLIENT_1_0` block executes the OPC UA remote write or method call and confirms with `CNF`.

Technical Features

- **Unfiltered Event Triggering (Ungated)**: Because no `E_D_FF_ANY` flip-flop is included, no change filter is applied. Even if value `IN.D1` is identical to the previously transmitted value, a new `IN.E1` event forces execution.
- **Tailored for OPC UA CALL_METHOD / Remote Input**: Essential for user-interface controls (e.g. VT numeric inputs or Web HMIs) where re-committing the same numeric value must reliably invoke the remote RPC method every time.
- **Decoupled from Change-Filter Logic**: While `AR_CLIENT_1_0` is optimized for continuous send-on-change use cases, `AR_CLIENT_1_0_UNGATED` serves event-driven method execution.

State Overview

1. **Uninitialized**: The block waits for the `INIT` event.
2. **Initialized**: Connection to the server is established; ready to transmit.
3. **Send Active**: Each `IN.E1` event immediately triggers transmission via `CLIENT_1_0`.

Application Scenarios

- **Remote Method Invocations via VT/HMI**: E.g. `NumericValue_TO_CLIENT_1_0_OPC.SUB` — when an operator confirms a numeric entry on the VT, the OPC UA method must be invoked even if the entered number is unchanged.
- **Event-Driven Command Transmission**: Transferring REAL parameters where the timing of the trigger event conveys independent meaning.

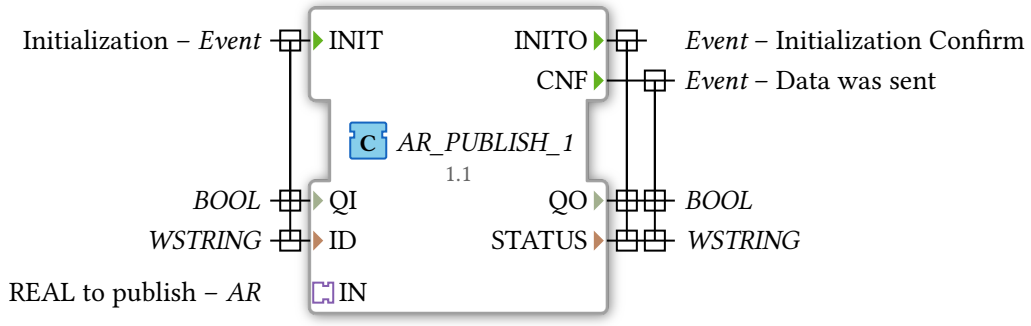
Comparison with Similar Blocks

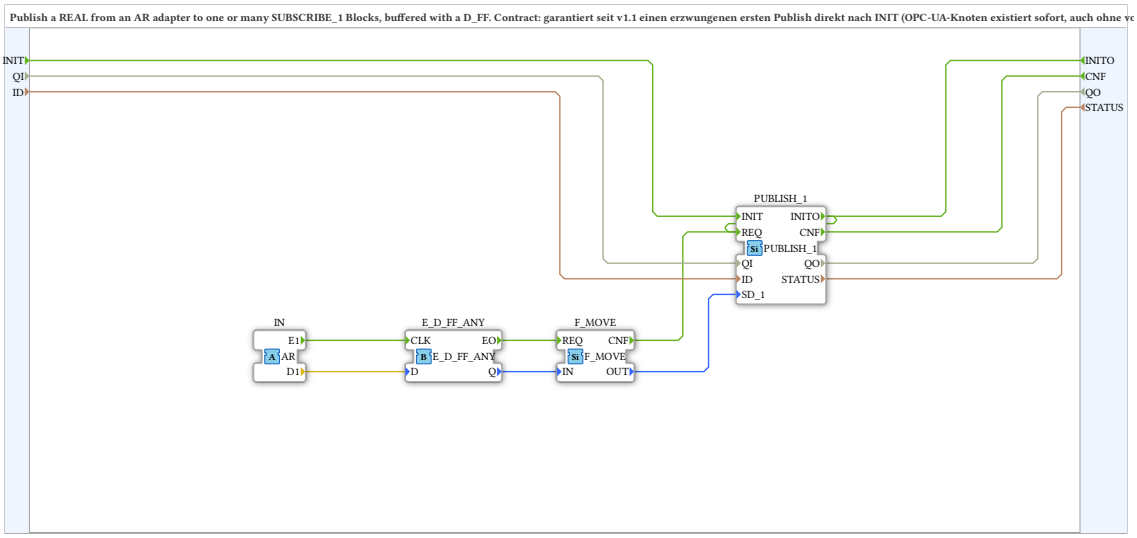
- **AR_CLIENT_1_0**: Contains an internal `E_D_FF_ANY` flip-flop that filters out unchanged values (Send-on-Change). `AR_CLIENT_1_0_UNGATED` deliberately omits this filter.
- **AX_CLIENT_1_0**: Handles BOOL values with a change filter.
- **ATM_CLIENT_1_0**: Handles TIME values with a change filter.

Conclusion

AR_CLIENT_1_0_UNGATED ensures that every trigger event on the AR adapter interface results in an OPC UA transmission — ideal for remote method calls in HMI and VT applications.

AR_PUBLISH_1





• – ○ • – ○ • – ○ •

The **AR_PUBLISH_1** function block is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AR adapter**. A **REAL** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

REAL value at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

** ## Interface Structure

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`
- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier
- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AR	Socket (Input)	REAL value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AR socket `IN` delivers an event to `IN.E1`, the REAL value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.

4. EO triggers the send event REQ of the internal PUBLISH_1 block. The buffered value Q is transmitted over the data connection as SD_1 and published on the configured channel.
5. After successful transmission, the PUBLISH_1 block confirms with CNF, which is then sent out.

- **Buffering with D-Flip-Flop**: The REAL value to be transmitted is buffered via an internal iec61499::events::E_D_FF_ANY. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based E_D_FF_ANY decouples immediate feedback loops between PUBLISH_1 and SUBSCRIBE_1 (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with AX_D_FF.
- **Encapsulation**: The original event/data interface of PUBLISH_1 (including REQ/SD_1) is moved internally; only the AR adapter interface is visible externally.

1. **Not Initialized**: The block is waiting for the INIT event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AR socket buffers the value and triggers transmission via the internal PUBLISH_1 block.

- **Network Coupling in AR Applications**: Sending REAL signals (e.g., switching commands or status bits) from the adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with AX_SUBSCRIBE_1, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AR adapters.
- **PUBLISH_1**: The standard block exposes its interface as events/variables (REQ, SD_1, ...). **AR_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (E_D_FF_ANY), and only provides the AR adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a real value at the adapter level but lacks network functionality. **AR_PUBLISH_1** combines buffering and publishing in a single block.

PUBLISH_1 PUBLISH_1 ## Conclusion

AR_PUBLISH_1 combines the publish/subscribe communication of the standard function block PUBLISH_1 with adapter-based REAL processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AR interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AR_PUBLISH_1 combines the publish/subscribe communication of the standard function block PUBLISH_1 with adapter-based REAL processing.

Technical Features

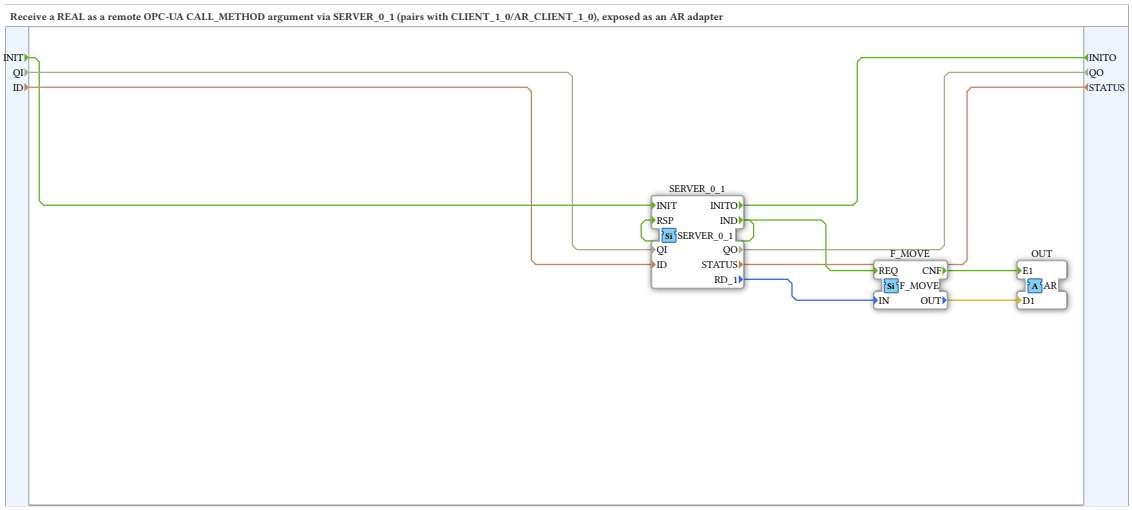
State Overview

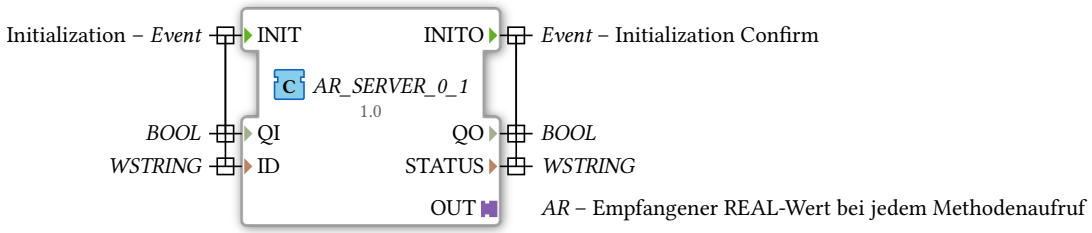
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AR_SERVER_0_1





• – ◦ • – ◦ • – ◦ •

Introduction

The **AR_SERVER_0_1** function block is a composite function block from the `adapter::net` package. It encapsulates the network server block `iec61499::net::SERVER_0_1` and exposes incoming OPC UA method calls (`CALL_METHOD`) for **REAL** arguments as a unidirectional **AR adapter** (`OUT`). It serves as the receiving counterpart to **AR_CLIENT_1_0** (or `CLIENT_1_0`).

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, connected to `QI` and `ID` (forwards to `SERVER_0_1.INIT`)

Event Outputs

- **INITO** (EInit): Initialization confirmation (`SERVER_0_1.INITO`), connected to `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Input qualifier, enables (TRUE) or disables (FALSE) the server
- **ID** (WSTRING): Server identifier (OPC UA server address and method specification)

Data Outputs

- **QO** (BOOL): Output qualifier, server operational status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapters

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AR	Plug (Output)	Received REAL value on each method invocation

Functionality

1. An `INIT` event at `AR_SERVER_0_1` is forwarded to the internal `SERVER_0_1.INIT` input. The server starts the OPC UA endpoint and acknowledges with `INITO` upon success.
2. Upon an incoming method call from a remote client (`CLIENT_1_0`/`AR_CLIENT_1_0`), `SERVER_0_1` emits indication event `IND` and presents the received REAL value at `RD_1`.
3. To prevent response-latency lockups (RSP latency bug), `SERVER_0_1.IND` is directly looped back to `SERVER_0_1.RSP` within the internal network, causing the server to acknowledge the method immediately.
4. Simultaneously, `IND` passes the received value through `F_MOVE` (`REAL` data type) to AR plug `OUT.D1` and fires output event `OUT.E1`.

Technical Features

- **Direct RSP Loopback**: To eliminate 4-second server lockups caused by delayed responses, `SERVER_0_1.IND` is internally tied back to `SERVER_0_1.RSP`.
- **Seamless Adapter Integration**: The raw `RD_1` output of `SERVER_0_1` is mapped via `F_MOVE` (REAL) directly onto the AR adapter structure (`OUT.D1` / `OUT.E1`).
- **Receiving Counterpart to `AR_CLIENT_1_0`**: In IEC 61499, a `CLIENT_1_0` block (“Connect to a SERVER_0_1 block”) pairs with a `SERVER_0_1` block.

State Overview

1. **Uninitialized**: The server is inactive and waiting for the `INIT` event.
2. **Listening**: The OPC UA server is active and waiting for method calls from remote clients.
3. **Indication**: A method call arrives; the REAL value is passed to the `OUT` adapter and the response is sent immediately.

Application Scenarios

- **Receiving Remote Setpoints**: Receiving analog setpoints (e.g. target pressure, target speed) from a remote control module via OPC UA `CALL_METHOD`.
- **Modular Server Architectures**: Providing OPC UA method endpoints in applications built consistently on adapters.

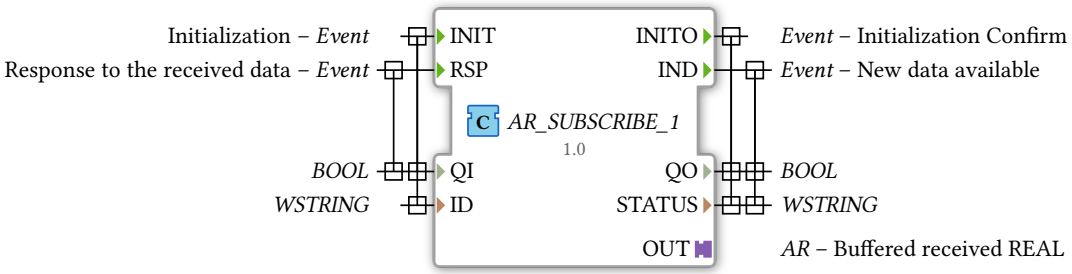
Comparison with Similar Blocks

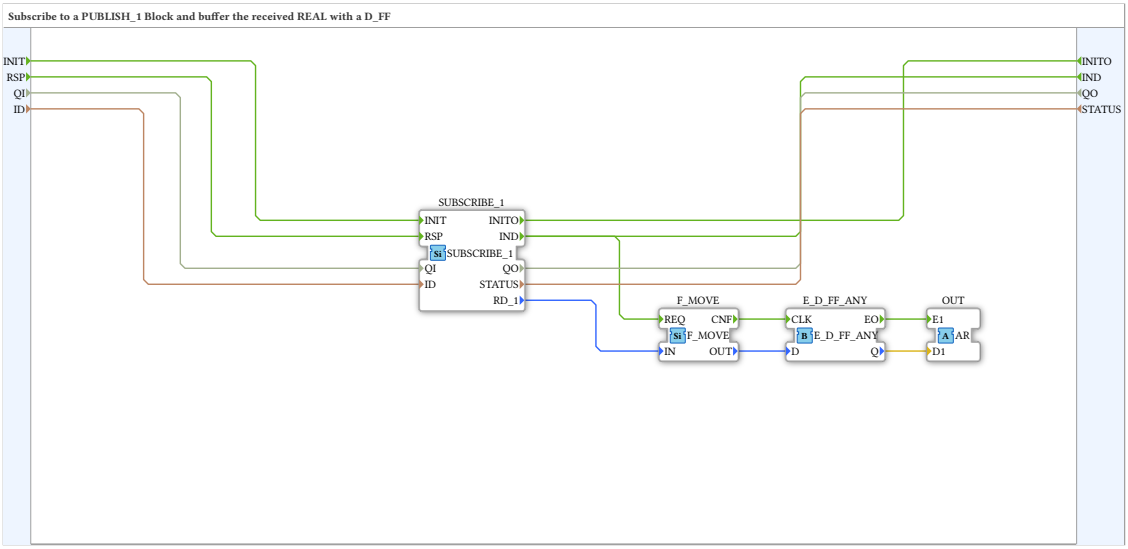
- **AX_SERVER_0_1**: Receives BOOL values via an AX adapter.
- **ATM_SERVER_0_1**: Receives TIME values via an ATM adapter.
- **AR_SUBSCRIBE_1**: Encapsulates `SUBSCRIBE_1` for local publish/subscribe rather than method invocations.

Conclusion

AR_SERVER_0_1 delivers received OPC UA method invocations for REAL values over a clean AR adapter while preventing server response locks.

AR_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AR_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AR adapter**. It subscribes to a publish channel configured via `ID` and provides the received **REAL** value, buffered via a D flip-flop (E_D_FF_ANY), at the adapter plug `OUT`. This allows for clean integration of network communication and REAL signal processing in the AR adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AR	Plug (Output)	Buffered received REAL value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.

5. The event EO of the flip-flop is transferred to the event output OUT.E1, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received REAL value is buffered via an internal iec61499::events::E_D_FF_ANY and remains stable at the output until the next value is received. EO only fires when Q changes.
- **RSP is a no-op for subscribers**: According to source code analysis of 4diac-forte (CCommFB::sendData()), the RSP event in SUBSCRIBE_1 (service type subscriber) does not execute any action – the branch != e_Subscriber is skipped. The event input pair INIT/RSP is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of SUBSCRIBE_1 (including IND/RD_1) is moved internally; only the AR adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the INIT event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Receipt**: At IND, the received REAL value is transferred to the internal D flip-flop and output via OUT.

Application Scenarios

- **Network Coupling in AR Applications**: Receiving real-world signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AR adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with AX_PUBLISH_1 and a true event-based E_D_FF_ANY, the feedback between SUBSCRIBE_1 and PUBLISH_1 can be decoupled (a clean fix compared to the interim fix with AX_D_FF).
- **Modular Control Architectures**: Integrating distributed communication into libraries that consistently rely on AR adapters.

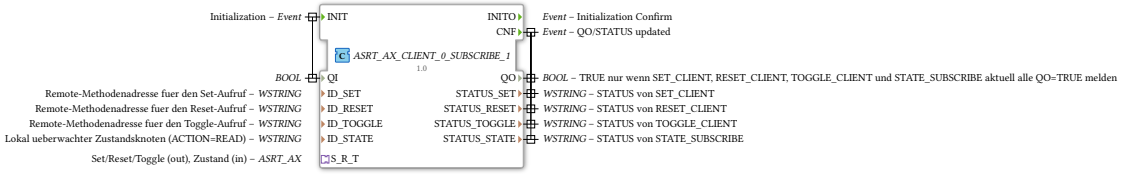
Comparison with Similar Building Blocks

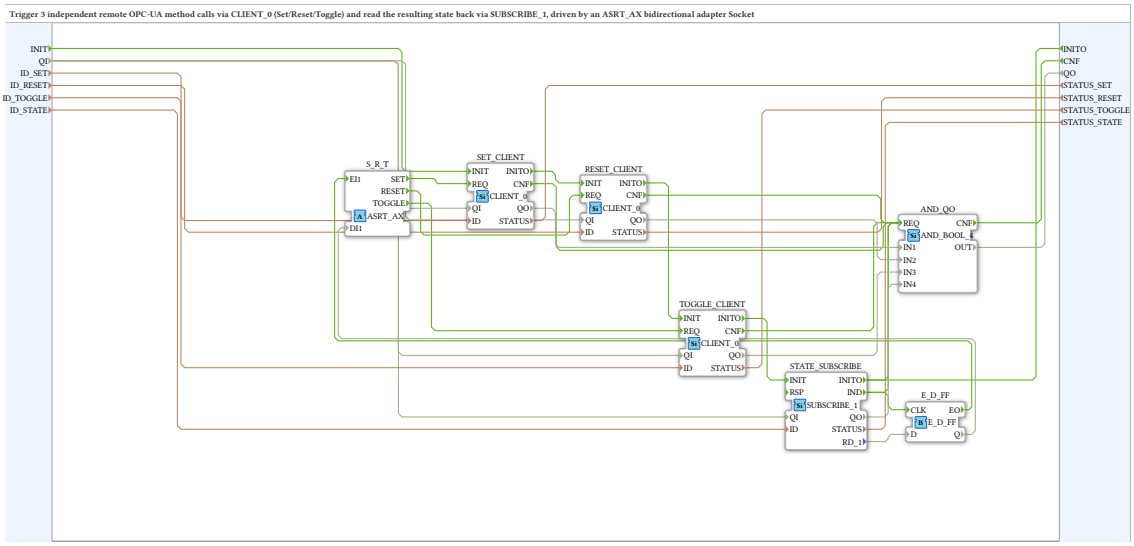
- **SUBSCRIBE_1**: The standard building block exposes its interface as events/variables (IND, RD_1, …). **AR_SUBSCRIBE_1** encapsulates this building block, buffers the value with a D flip-flop (E_D_FF_ANY), and only provides the AR adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a real value at the adapter level but lacks network functionality. **AR_SUBSCRIBE_1** combines subscribing and buffering in a single building block.

Conclusion

AR_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block SUBSCRIBE_1 with adapter-based REAL processing. The internal D flip-flop buffer keeps the received value stable until new data arrives, and the reduced AR interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

ASRT_AX_CLIENT_0_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The **ASRT_AX_CLIENT_0_SUBSCRIBE_1** function block is a composite function block that triggers three independent remote OPC UA **method calls** (Set, Reset, and Toggle), each via its own `CLIENT_0` block, and reads the corresponding state back via `SUBSCRIBE_1` – all behind a single bidirectional **ASRT_AX adapter socket**. It is the variant of ASRT_CLIENT_0 extended with a state channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO`, `STATUS_SET`, `STATUS_RESET`, `STATUS_TOGGLE`, and `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` updated, associated with the same variables

Data Inputs

- **QI** (BOOL): Qualifier input for all four internal connections
- **ID_SET** (WSTRING): Remote method address for the Set call
- **ID_RESET** (WSTRING): Remote method address for the Reset call
- **ID_TOGGLE** (WSTRING): Remote method address for the Toggle call
- **ID_STATE** (WSTRING): Locally monitored state node (ACTION=READ)

Data Outputs

- **QO** (BOOL): TRUE only if `SET_CLIENT`, `RESET_CLIENT`, `TOGGLE_CLIENT`, and `STATE_SUBSCRIBE` currently all report `QO = TRUE`
- **STATUS_SET** (WSTRING): Status information from `SET_CLIENT`
- **STATUS_RESET** (WSTRING): Status information from `RESET_CLIENT`
- **STATUS_TOGGLE** (WSTRING): Status information from `TOGGLE_CLIENT`
- **STATUS_STATE** (WSTRING): Status information from `STATE_SUBSCRIBE`

Adapter

Adapter	Type	Direction	Description
S_R_T	adapter::types::bidirectional::ASRT_AX	Socket – Set/Reset/Toggle (output), state (input)	Set/Reset/Toggle out, state back

Functionality

1. Via the `INIT` event, `SET_CLIENT`, `RESET_CLIENT`, `TOGGLE_CLIENT`, and `STATE_SUBSCRIBE` are initialized one after another. After all four have confirmed, `INITO` is reported externally.
2. `S_R_T.SET`, `S_R_T.RESET`, and `S_R_T.TOGGLE` directly trigger the corresponding `CLIENT_0` method calls.
3. As soon as `STATE_SUBSCRIBE` delivers a new state value via `IND`, this clocks the internal **E_D_FF** flip-flop, which captures and holds the value stably.
4. The flip-flop's output `EO` fires `S_R_T.EI1` with the buffered value at `S_R_T.DI1` – the state is reported back over the same adapter used to trigger Set/Reset/Toggle.
5. `AND_QO` (AND_BOOL_4) ANDs the `QO` outputs of all four internal blocks; every confirmation additionally triggers `CNF` externally.

Technical Features

- **Four network connections behind one adapter**: Three `CLIENT_0` instances (Set, Reset, Toggle) and one `SUBSCRIBE_1` (state) are combined into a single bidirectional ASRT_AX interface.
- **Buffering with D flip-flop**: The state received via `SUBSCRIBE_1` is stabilized via an internal `iec61499::events::E_D_FF` before being reported back over the adapter.
- **Sequential initialization**: `SET_CLIENT` → `RESET_CLIENT` → `TOGGLE_CLIENT` → `STATE_SUBSCRIBE` (composite FBTypes chain `INIT`/`INITO` serially through each instance).
- **Encapsulation**: Only the ASRT_AX adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: All four connections (Set, Reset, Toggle, state subscription) are established.
3. **Send Active**: A Set, Reset, or Toggle event arriving at the adapter triggers the corresponding remote method call.
4. **State Update**: A new value from the subscription is buffered and reported back over the same adapter.

Application Scenarios

- Remotely triggering Set/Reset/Toggle on a remote actuator while simultaneously reading back the resulting state over a single adapter connection
- Reducing manual CLIENT_0/SUBSCRIBE_1 wiring to a single reusable building block

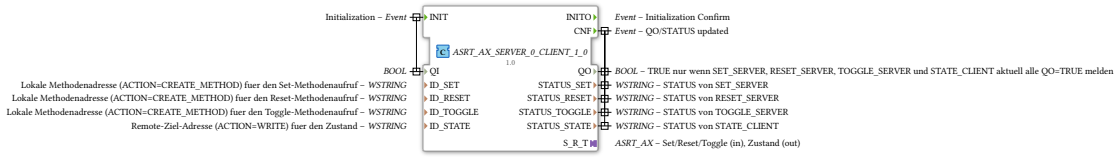
Comparison with Similar Function Blocks

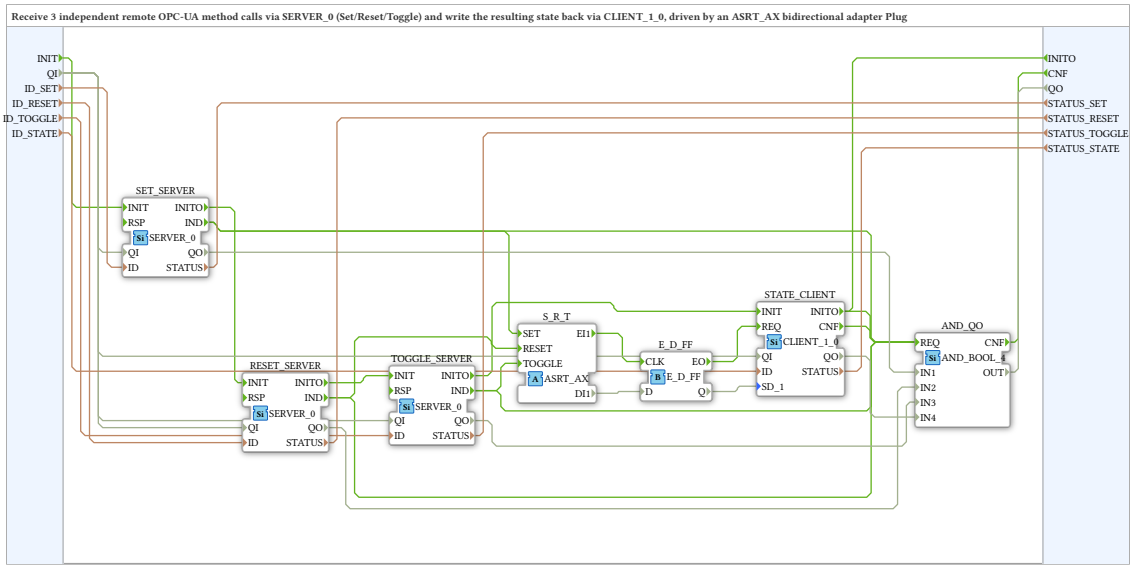
- **ASRT_CLIENT_0**: The same three triggers, but without reading the state back.
- **ASR_AX_CLIENT_0_SUBSCRIBE_1**: The same pattern with only two triggers (no Toggle).
- **ASRT_AX_SERVER_0_CLIENT_1_0**: The server-side counterpart.

Conclusion

ASRT_AX_CLIENT_0_SUBSCRIBE_1 bundles three remote triggers (Set/Reset/Toggle) and a state subscription behind a single bidirectional ASRT_AX adapter, turning a hand-wired combination into a single reusable building block.

ASRT_AX_SERVER_0_CLIENT_1_0





• – ○ • – ○ • – ○ •

Introduction

The **ASRT_AX_SERVER_0_CLIENT_1_0** function block is a composite function block that receives three independent remote OPC UA **method calls** (Set, Reset, and Toggle), each via its own `SERVER_0` block, and writes the resulting state back to a remote node via `CLIENT_1_0` – all behind a single bidirectional **ASRT_AX adapter plug**. It is the server-side counterpart to ASRT_AX_CLIENT_0_SUBSCRIBE_1.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO`, `STATUS_SET`, `STATUS_RESET`, `STATUS_TOGGLE`, and `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` updated, associated with the same variables

Data Inputs

- **QI** (BOOL): Qualifier input for all four internal connections
- **ID_SET** (WSTRING): Local method address (ACTION=CREATE_METHOD) for the Set method call
- **ID_RESET** (WSTRING): Local method address (ACTION=CREATE_METHOD) for the Reset method call
- **ID_TOGGLE** (WSTRING): Local method address (ACTION=CREATE_METHOD) for the Toggle method call
- **ID_STATE** (WSTRING): Remote target address (ACTION=WRITE) for the state

Data Outputs

- **QO** (BOOL): TRUE only if `SET_SERVER`, `RESET_SERVER`, `TOGGLE_SERVER`, and `STATE_CLIENT` currently all report `QO = TRUE`
- **STATUS_SET** (WSTRING): Status information from `SET_SERVER`
- **STATUS_RESET** (WSTRING): Status information from `RESET_SERVER`
- **STATUS_TOGGLE** (WSTRING): Status information from `TOGGLE_SERVER`
- **STATUS_STATE** (WSTRING): Status information from `STATE_CLIENT`

Adapter

Adapter	Type	Direction	Description

S_R_T	adapter::types::bidirectional::ASRT_AX	Plug – Set/Reset/Toggle (input), state (output)	Received Set/Reset/Toggle, state out

Functionality

1. Via the `INIT` event, `SET_SERVER`, `RESET_SERVER`, `TOGGLE_SERVER`, and `STATE_CLIENT` are initialized one after another. After all four have confirmed, `INITO` is reported externally.
2. A remote Set, Reset, or Toggle call on the respective `SERVER_0` block triggers `S_R_T.SET`, `S_R_T.RESET`, or `S_R_T.TOGGLE` – all three are thus reported inward via the adapter.
3. As soon as the downstream logic delivers a feedback event `S_R_T.EI1` with data `S_R_T.DI1` via the adapter, this clocks the internal **E_D_FF** flip-flop, which captures and holds the value stably.
4. The flip-flop's output `EO` triggers `STATE_CLIENT.REQ`; the buffered value is sent as an OPC UA write to the remote node configured in `ID_STATE`.
5. `AND_QO` (AND_BOOL_4) ANDs the `QO` outputs of all four internal blocks; every confirmation additionally triggers `CNF` externally.

Technical Features

- **Four network connections behind one adapter**: Three `SERVER_0` instances (Set, Reset, Toggle) and one `CLIENT_1_0` (state) are combined into a single bidirectional ASRT_AX interface.
- **Buffering with D flip-flop**: The feedback value reported via the adapter is stabilized via an internal `iec61499::events::E_D_FF` before being sent via `CLIENT_1_0`.
- **Sequential initialization**: `SET_SERVER` → `RESET_SERVER` → `TOGGLE_SERVER` → `STATE_CLIENT` (composite FBTypes chain `INIT`/`INITO` serially through each instance).
- **Encapsulation**: Only the ASRT_AX adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: All four connections (Set, Reset, Toggle server, state client) are established.
3. **Receive Active**: A remote Set, Reset, or Toggle call is reported to the internal logic via the adapter.
4. **State Write**: A feedback value from the internal logic is buffered and sent via remote write to the configured target node.

Application Scenarios

- Receiving remote Set/Reset/Toggle calls while simultaneously writing the resulting state back to a different remote node over a single adapter connection
- Server-side counterpart to ASRT_AX_CLIENT_0_SUBSCRIBE_1 in distributed control architectures

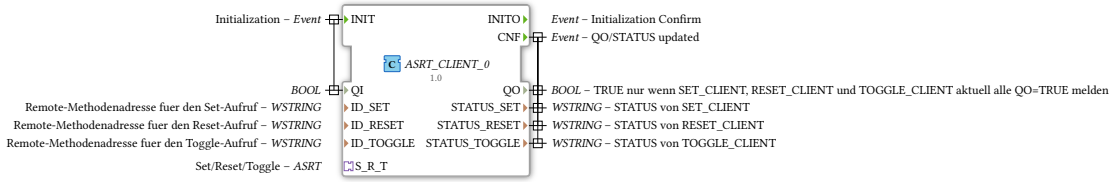
Comparison with Similar Function Blocks

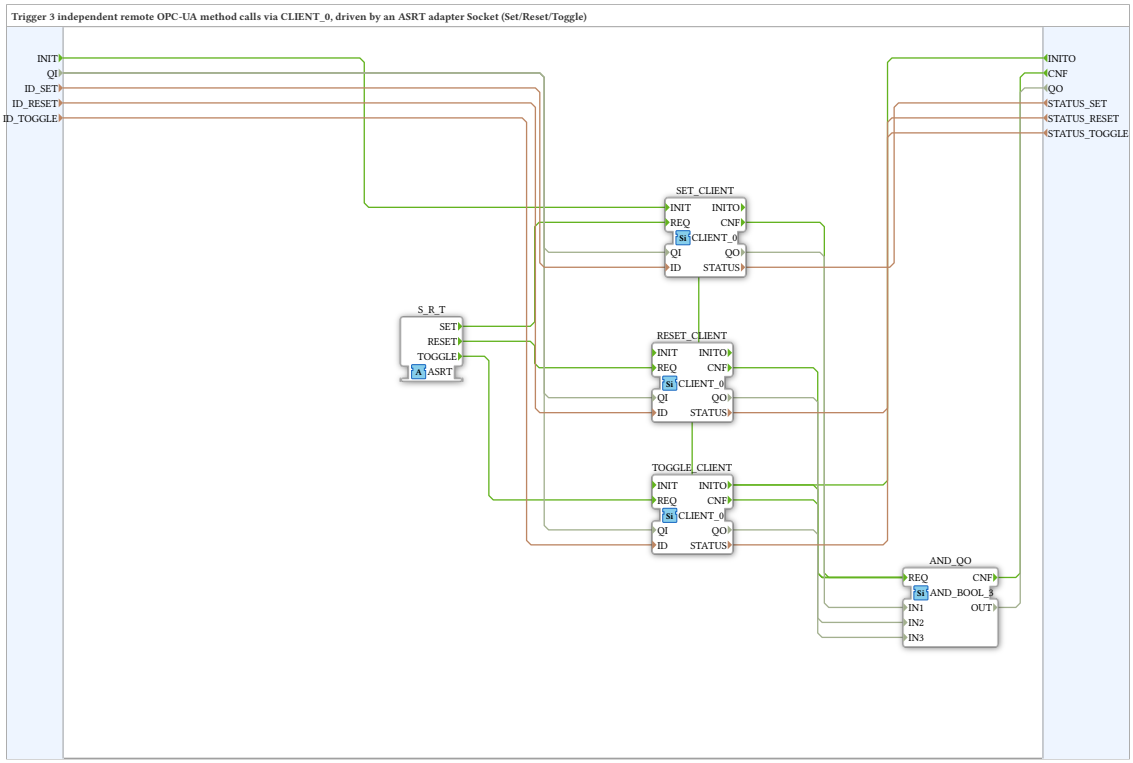
- **ASRT_AX_CLIENT_0_SUBSCRIBE_1**: The client/calling counterpart.
- **ASR_AX_SERVER_0_CLIENT_1_0**: The same pattern with only two receiving servers (no Toggle).

Conclusion

ASRT_AX_SERVER_0_CLIENT_1_0 bundles three receiving servers (Set/Reset/Toggle) and a state-writing client behind a single bidirectional ASRT_AX adapter, forming the server-side counterpart to ASRT_AX_CLIENT_0_SUBSCRIBE_1.

ASRT_CLIENT_0





• – ○ • – ○ • – ○ •

Introduction

The **ASRT_CLIENT_0** function block is a composite function block that triggers three independent remote OPC UA **method calls** (Set, Reset, and Toggle), each via its own `CLIENT_0` block, controlled through a single unidirectional **ASRT adapter socket**. It is the variant of ASR_CLIENT_0 extended with a third trigger.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO`, `STATUS_SET`, `STATUS_RESET`, and `STATUS_TOGGLE`
- **CNF** (Event): Confirmation that one of the method calls has been sent, associated with the same status variables

Data Inputs

- **QI** (BOOL): Qualifier input, opens (TRUE) or closes (FALSE) all three connections
- **ID_SET** (WSTRING): Remote method address for the Set call
- **ID_RESET** (WSTRING): Remote method address for the Reset call
- **ID_TOGGLE** (WSTRING): Remote method address for the Toggle call

Data Outputs

- **QO** (BOOL): TRUE only if `SET_CLIENT`, `RESET_CLIENT`, and `TOGGLE_CLIENT` currently all report `QO = TRUE`
- **STATUS_SET** (WSTRING): Status information from `SET_CLIENT`
- **STATUS_RESET** (WSTRING): Status information from `RESET_CLIENT`
- **STATUS_TOGGLE** (WSTRING): Status information from `TOGGLE_CLIENT`

Adapter

Adapter	Type	Direction	Description
S_R_T	adapter::types::unidirectional::ASRT	Socket (Input)	Set/Reset/Toggle

Functionality

1. Via the `INIT` event, the internal `SET_CLIENT`, `RESET_CLIENT`, and `TOGGLE_CLIENT` blocks are initialized one after another (chained serially through `INIT`/`INITO`) with `QI` and their respective `ID`. After all three have confirmed, `INITO` is reported externally.
2. An event at the adapter socket `S_R_T.SET`, `S_R_T.RESET`, or `S_R_T.TOGGLE` directly triggers the corresponding `CLIENT_0` method call.
3. All three `CLIENT_0` instances confirm independently with `CNF`, which is forwarded externally in each case.
4. The `AND_QO` block (AND_BOOL_3) ANDs the `QO` outputs of all three `CLIENT_0` instances into a single `QO`.

Technical Features

- **Three parallel CLIENT_0 instances**: Set, Reset, and Toggle are handled by separate `CLIENT_0` blocks (different `ID`s, independent connections).
- **Sequential initialization**: The three internal blocks are initialized one after another (`SET_CLIENT.INITO` → `RESET_CLIENT.INIT` → `TOGGLE_CLIENT.INIT`), since composite FBTypes must explicitly chain their `INIT`/`INITO` through each instance in series.
- **Encapsulation**: Only the ASRT adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: All three connections are established, the block is ready to trigger Set, Reset, or Toggle calls.
3. **Send Active**: An event arriving at the ASRT socket triggers the corresponding remote method call.

Application Scenarios

- Remotely triggering three independent remote actions (Set/Reset/Toggle) through a single adapter, e.g. to switch a remote actuator with an additional toggle function
- Network coupling in ASRT-adapter-based applications

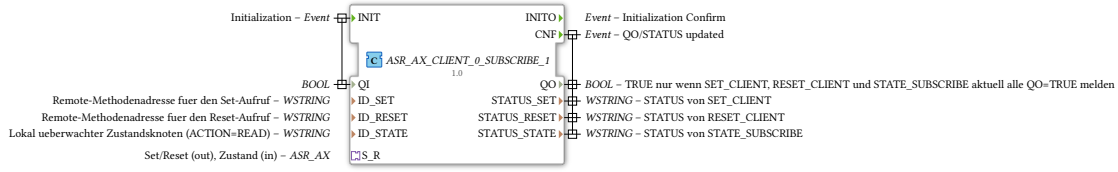
Comparison with Similar Function Blocks

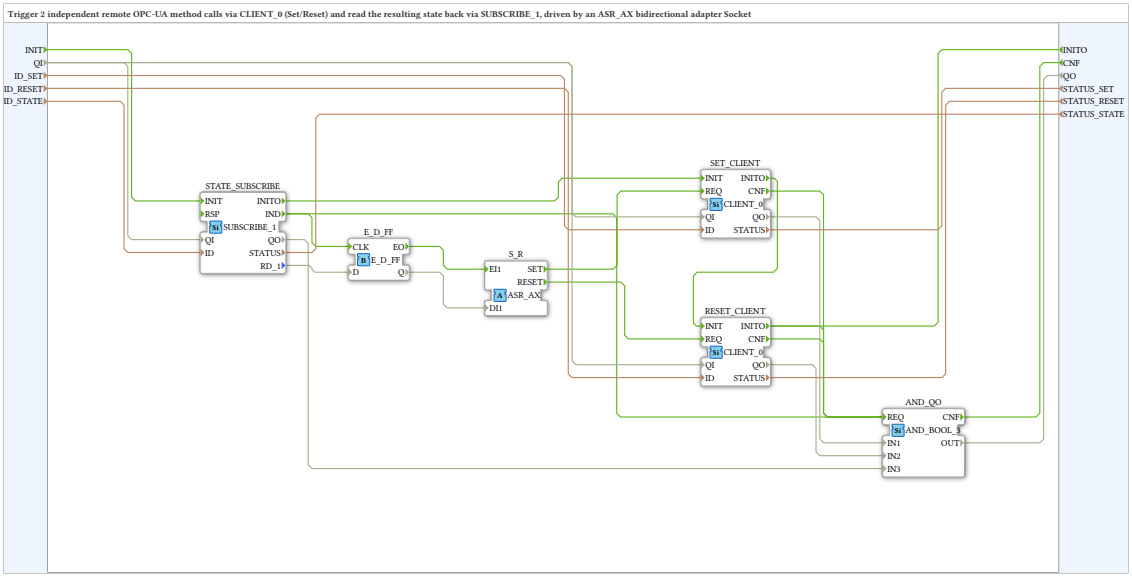
- **ASR_CLIENT_0**: The same base structure with only two triggers (no Toggle).
- **AE_CLIENT_0**: Reduces the same pattern to a single trigger.
- **ASRT_AX_CLIENT_0_SUBSCRIBE_1**: Combines the same three `CLIENT_0` triggers with an additional `SUBSCRIBE_1` state channel behind a bidirectional adapter.

Conclusion

ASRT_CLIENT_0 bundles three independent remote method calls (Set/Reset/Toggle) behind a single ASRT adapter socket, making it suitable for applications where a remote actuator is switched via three separate OPC UA methods.

ASR_AX_CLIENT_0_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The **ASR_AX_CLIENT_0_SUBSCRIBE_1** function block is a composite function block that triggers two independent remote OPC UA **method calls** (Set and Reset), each via its own `CLIENT_0` block, and reads the corresponding state back via `SUBSCRIBE_1` – all behind a single bidirectional **ASR_AX adapter socket**. It is the variant of ASR_CLIENT_0 extended with a state channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO`, `STATUS_SET`, `STATUS_RESET`, and `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` updated, associated with the same variables

Data Inputs

- **QI** (BOOL): Qualifier input for all three internal connections
- **ID_SET** (WSTRING): Remote method address for the Set call
- **ID_RESET** (WSTRING): Remote method address for the Reset call
- **ID_STATE** (WSTRING): Locally monitored state node (ACTION=READ)

Data Outputs

- **QO** (BOOL): TRUE only if `SET_CLIENT`, `RESET_CLIENT`, and `STATE_SUBSCRIBE` currently all report `QO = TRUE`
- **STATUS_SET** (WSTRING): Status information from `SET_CLIENT`
- **STATUS_RESET** (WSTRING): Status information from `RESET_CLIENT`
- **STATUS_STATE** (WSTRING): Status information from `STATE_SUBSCRIBE`

Adapter

Adapter	Type	Direction	Description
S_R	adapter::types::bidirectional::ASR_AX	Socket – Set/Reset (output), state (input)	Set/Reset out, state back

Functionality

1. Via the INIT event, STATE_SUBSCRIBE is initialized first, followed one after another by SET_CLIENT and RESET_CLIENT. After all have confirmed, INITO is reported externally.
2. S_R.SET and S_R.RESET directly trigger the method calls SET_CLIENT.REQ and RESET_CLIENT.REQ respectively.
3. As soon as STATE_SUBSCRIBE delivers a new state value via IND, this clocks the internal **E_D_FF** flip-flop, which captures and holds the value stably.
4. The flip-flop's output EO fires S_R.EI1 with the buffered value at S_R.DI1 – the state is reported back over the same adapter used to trigger Set/Reset.
5. AND_QO (AND_BOOL_3) ANDs the QO outputs of all three internal blocks; every confirmation additionally triggers CNF externally.

Technical Features

- **Three network connections behind one adapter**: Two CLIENT_0 instances (Set, Reset) and one SUBSCRIBE_1 (state) are combined into a single bidirectional ASR_AX interface.
- **Buffering with D flip-flop**: The state received via SUBSCRIBE_1 is stabilized via an internal iec61499::events::E_D_FF before being reported back over the adapter.
- **Sequential initialization**: STATE_SUBSCRIBE ⟶ SET_CLIENT ⟶ RESET_CLIENT (composite FBTypes chain INIT/INITO serially through each instance).
- **Encapsulation**: Only the ASR_AX adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the INIT event.
2. **Initialized**: All three connections (Set, Reset, state subscription) are established.
3. **Send Active**: A Set or Reset event arriving at the adapter triggers the corresponding remote method call.
4. **State Update**: A new value from the subscription is buffered and reported back over the same adapter.

Application Scenarios

- Remotely triggering Set/Reset on a remote actuator while simultaneously reading back the resulting state over a single adapter connection
- Reducing manual CLIENT_0/SUBSCRIBE_1 wiring to a single reusable building block

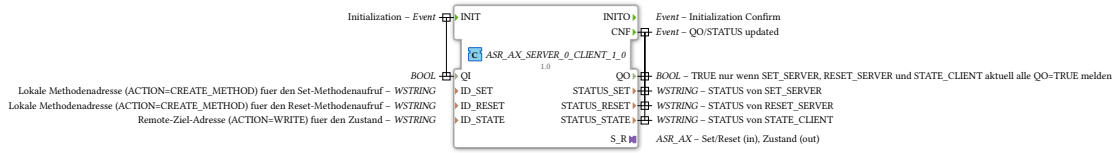
Comparison with Similar Function Blocks

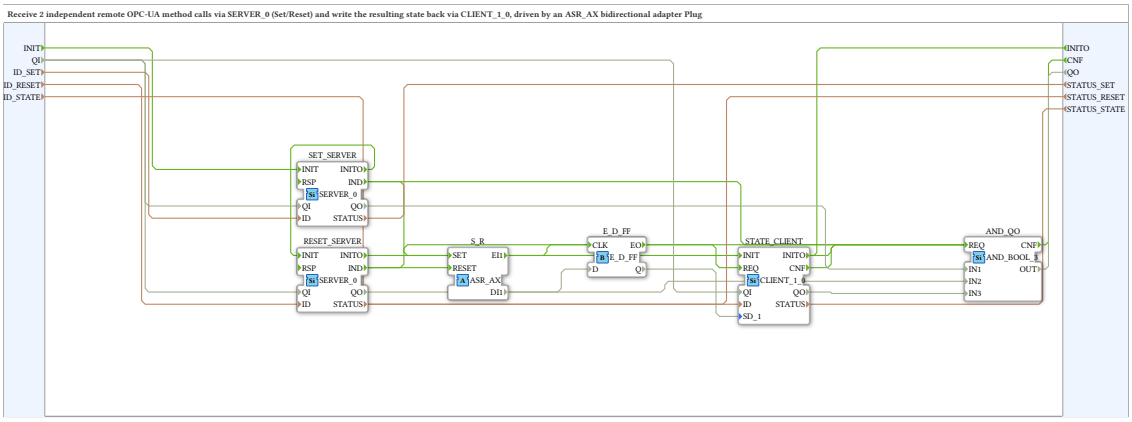
- **ASR_CLIENT_0**: The same two triggers, but without reading the state back.
- **AE_AX_CLIENT_0_SUBSCRIBE_1**: The same pattern with only one trigger instead of two.
- **ASRT_AX_CLIENT_0_SUBSCRIBE_1**: Extends the same pattern with a third trigger (Toggle).
- **ASR_AX_SERVER_0_CLIENT_1_0**: The server-side counterpart.

Conclusion

ASR_AX_CLIENT_0_SUBSCRIBE_1 bundles two remote triggers (Set/Reset) and a state subscription behind a single bidirectional ASR_AX adapter, turning a hand-wired combination into a single reusable building block.

ASR_AX_SERVER_0_CLIENT_1_0





• — ○ • — ○ • — ○ •

Introduction

The **ASR_AX_SERVER_0_CLIENT_1_0** function block is a composite function block that receives two independent remote OPC UA **method calls** (Set and Reset), each via its own `SERVER_0` block, and writes the resulting state back to a remote node via `CLIENT_1_0` – all behind a single bidirectional **ASR_AX adapter plug**. It is the server-side counterpart to ASR_AX_CLIENT_0_SUBSCRIBE_1.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO`, `STATUS_SET`, `STATUS_RESET`, and `STATUS_STATE`
- **CNF** (Event): `QO`/`STATUS` updated, associated with the same variables

Data Inputs

- **QI** (BOOL): Qualifier input for all three internal connections
- **ID_SET** (WSTRING): Local method address (ACTION=CREATE_METHOD) for the Set method call
- **ID_RESET** (WSTRING): Local method address (ACTION=CREATE_METHOD) for the Reset method call
- **ID_STATE** (WSTRING): Remote target address (ACTION=WRITE) for the state

Data Outputs

- **QO** (BOOL): TRUE only if `SET_SERVER`, `RESET_SERVER`, and `STATE_CLIENT` currently all report `QO = TRUE`
- **STATUS_SET** (WSTRING): Status information from `SET_SERVER`
- **STATUS_RESET** (WSTRING): Status information from `RESET_SERVER`
- **STATUS_STATE** (WSTRING): Status information from `STATE_CLIENT`

Adapter

Adapter	Type	Direction	Description
S_R	adapter::types::bidirectional::ASR_AX	Plug – Set/Reset (input), state (output)	Received Set/Reset, state out

Functionality

1. Via the `INIT` event, `SET_SERVER`, `RESET_SERVER`, and `STATE_CLIENT` are initialized one after another. After all three have confirmed, `INITO` is reported externally.
2. A remote Set call on `SET_SERVER` triggers `S_R.SET`, and a remote Reset call on `RESET_SERVER` triggers `S_R.RESET` – both are thus reported inward via the adapter.
3. As soon as the downstream logic delivers a feedback event `S_R.EI1` with data `S_R.DI1` via the adapter, this clocks the internal **E_D_FF** flip-flop, which captures and holds the value stably.

4. The flip-flop's output `EO` triggers `STATE_CLIENT.REQ`; the buffered value is sent as an OPC UA write to the remote node configured in `ID_STATE`.
5. `AND_QO` (AND_BOOL_3) ANDs the `QO` outputs of all three internal blocks; every confirmation additionally triggers `CNF` externally.

Technical Features

- **Three network connections behind one adapter**: Two `SERVER_0` instances (Set, Reset) and one `CLIENT_1_0` (state) are combined into a single bidirectional ASR_AX interface.
- **Buffering with D flip-flop**: The feedback value reported via the adapter is stabilized via an internal `iec61499::events::E_D_FF` before being sent via `CLIENT_1_0`.
- **Sequential initialization**: `SET_SERVER` → `RESET_SERVER` → `STATE_CLIENT` (composite FBTypes chain `INIT`/`INITO` serially through each instance).
- **Encapsulation**: Only the ASR_AX adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: All three connections (Set server, Reset server, state client) are established.
3. **Receive Active**: A remote Set or Reset call is reported to the internal logic via the adapter.
4. **State Write**: A feedback value from the internal logic is buffered and sent via remote write to the configured target node.

Application Scenarios

- Receiving remote Set/Reset calls while simultaneously writing the resulting state back to a different remote node over a single adapter connection
- Server-side counterpart to ASR_AX_CLIENT_0_SUBSCRIBE_1 in distributed control architectures

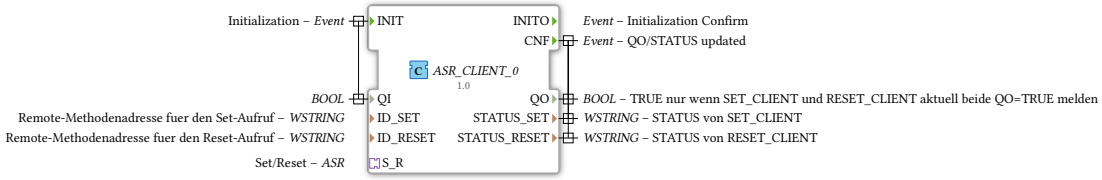
Comparison with Similar Function Blocks

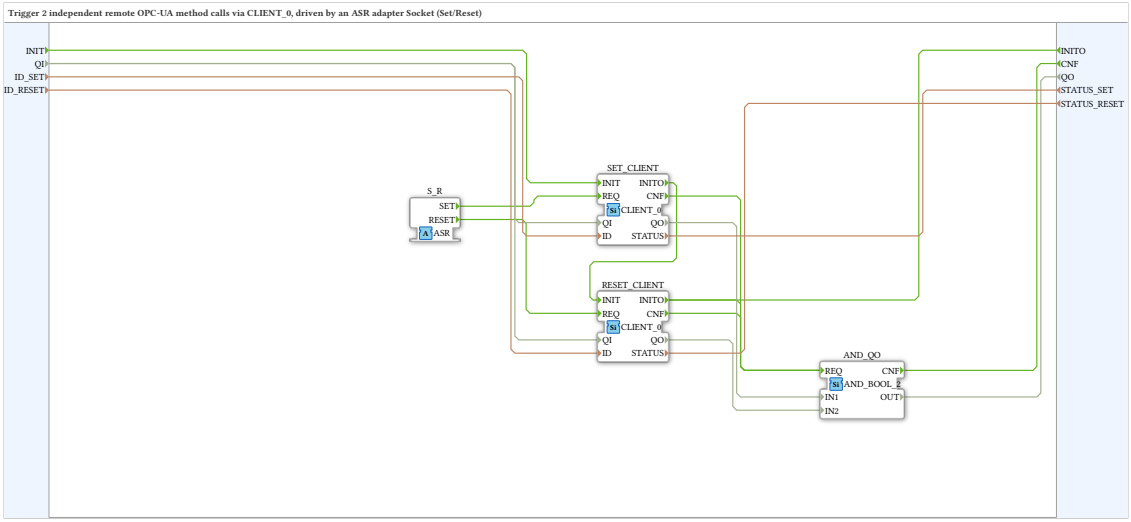
- **ASR_AX_CLIENT_0_SUBSCRIBE_1**: The client/calling counterpart.
- **AE_AX_SERVER_0_CLIENT_1_0**: The same pattern with only one receiving server instead of two.
- **ASRT_AX_SERVER_0_CLIENT_1_0**: Extends the same pattern with a third receiving server (Toggle).

Conclusion

ASR_AX_SERVER_0_CLIENT_1_0 bundles two receiving servers (Set/Reset) and a state-writing client behind a single bidirectional ASR_AX adapter, forming the server-side counterpart to ASR_AX_CLIENT_0_SUBSCRIBE_1.

ASR_CLIENT_0





• – ○ • – ○ • – ○ •

Introduction

The **ASR_CLIENT_0** function block is a composite function block that triggers two independent remote OPC UA **method calls** (Set and Reset), each via its own `CLIENT_0` block, controlled through a single unidirectional **ASR adapter socket**. It is the variant of AE_CLIENT_0 extended with a second trigger.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO`, `STATUS_SET`, and `STATUS_RESET`
- **CNF** (Event): Confirmation that one of the method calls has been sent, associated with `QO`, `STATUS_SET`, and `STATUS_RESET`

Data Inputs

- **QI** (BOOL): Qualifier input, opens (TRUE) or closes (FALSE) both connections
- **ID_SET** (WSTRING): Remote method address for the Set call
- **ID_RESET** (WSTRING): Remote method address for the Reset call

Data Outputs

- **QO** (BOOL): TRUE only if `SET_CLIENT` and `RESET_CLIENT` currently both report `QO = TRUE`
- **STATUS_SET** (WSTRING): Status information from `SET_CLIENT`
- **STATUS_RESET** (WSTRING): Status information from `RESET_CLIENT`

Adapter

Adapter	Type	Direction	Description
S_R	adapter::types::unidirectional::ASR	Socket (Input)	Set/Reset

Functionality

1. Via the `INIT` event, the internal `SET_CLIENT` and `RESET_CLIENT` blocks are initialized one after another (`INIT` → `INITO` → `INIT`) with `QI` and their respective `ID`. After both have confirmed, `INITO` is reported externally.
2. An event at the adapter socket `S_R.SET` directly triggers the internal `SET_CLIENT` method call; `S_R.RESET` triggers `RESET_CLIENT` accordingly.
3. Both `CLIENT_0` instances confirm independently with `CNF`, which is forwarded externally in each case.
4. The `AND_QO` block ANDs the `QO` outputs of both `CLIENT_0` instances into a single `QO`.

Technical Features

- **Two parallel CLIENT_0 instances**: Set and Reset are handled by separate `CLIENT_0` blocks (different `ID`s, independent connections).
- **Sequential initialization**: The two internal blocks are initialized one after another (`SET_CLIENT.INITO` → `RESET_CLIENT.INIT`), since composite FBTypes must explicitly chain their `INIT`/`INITO` through each instance in series.
- **Encapsulation**: Only the ASR adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: Both connections are established, the block is ready to trigger Set or Reset calls.
3. **Send Active**: A Set or Reset event arriving at the ASR socket triggers the corresponding remote method call.

Application Scenarios

- Remotely triggering two independent remote actions (Set/Reset) through a single adapter, e.g. to switch a remote actuator
- Network coupling in ASR-adapter-based applications

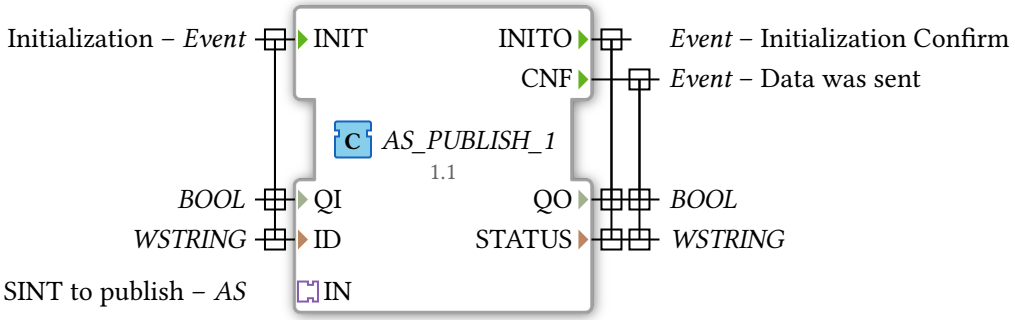
Comparison with Similar Function Blocks

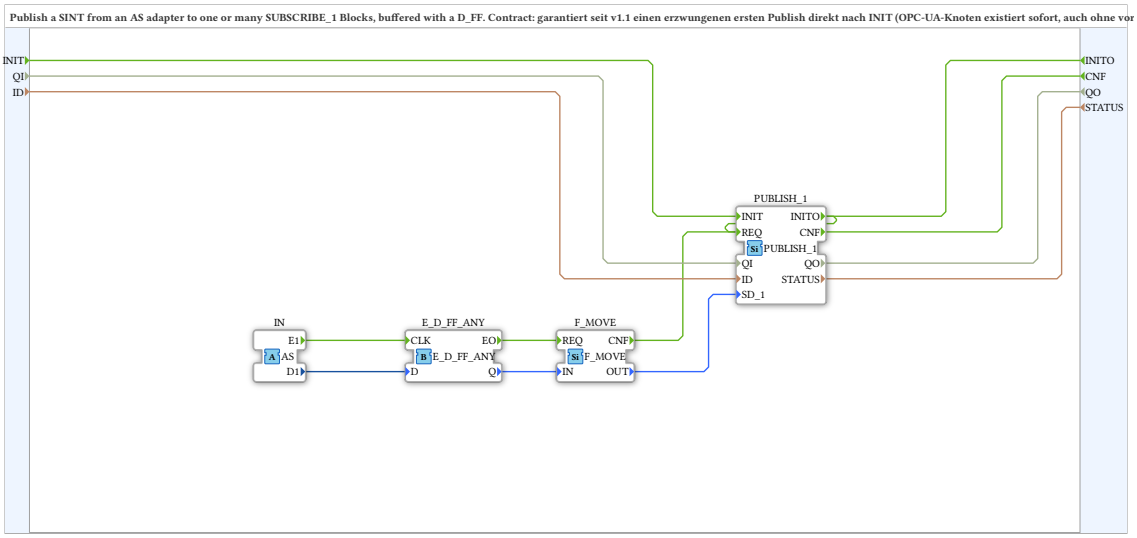
- **AE_CLIENT_0**: The same base structure with only one trigger instead of two.
- **ASRT_CLIENT_0**: Extends the same pattern with a third trigger (Toggle).
- **ASR_AX_CLIENT_0_SUBSCRIBE_1**: Combines the same two `CLIENT_0` triggers with an additional `SUBSCRIBE_1` state channel behind a bidirectional adapter.

Conclusion

ASR_CLIENT_0 bundles two independent remote method calls (Set/Reset) behind a single ASR adapter socket, making it suitable for applications where a remote actuator is switched via two separate OPC UA methods.

AS_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AS_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AS adapter**. A **SINT** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AS	Socket (Input)	SINT value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AS socket `IN` delivers an event to `IN.E1`, the SINT value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block acknowledges with `CNF`, which is then transmitted externally.

Technical Features

- **Buffering with D-Flip-Flop**: The SINT value to be transmitted is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AS adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AS socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in AS Applications**: Sending SINT signals (e.g., switching commands or status bits) from adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AS adapters.

Comparison with Similar Blocks

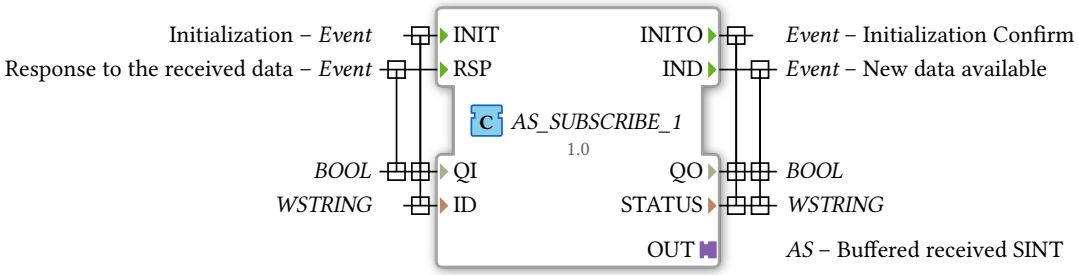
- **PUBLISH_1**: The standard block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AS_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AS adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a SINT value at the adapter level but lacks network functionality. **AS_PUBLISH_1** combines buffering and publishing in a single block.

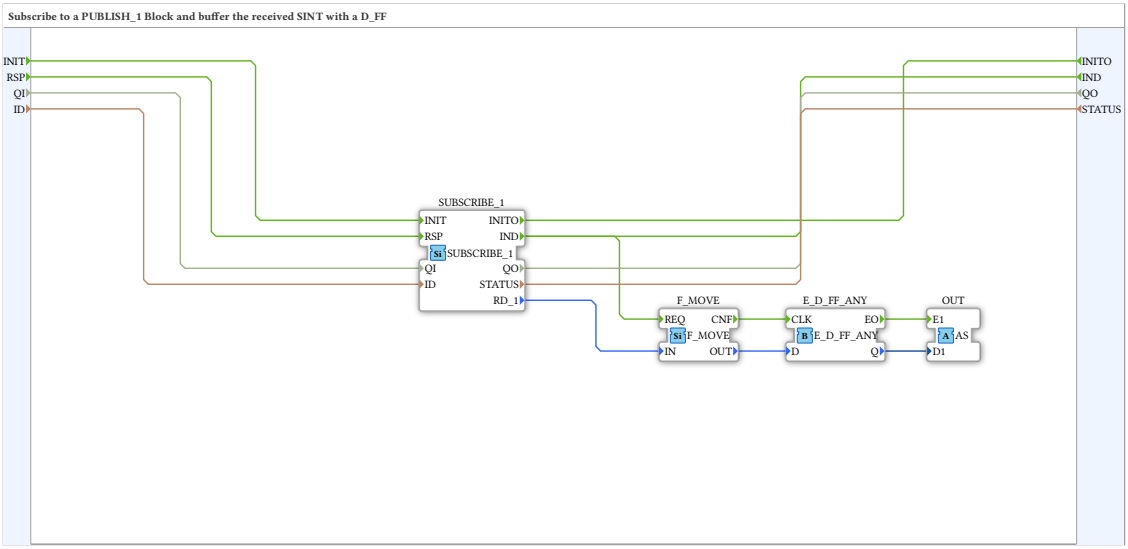
Conclusion

AS_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based SINT processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AS interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AS_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based SINT processing.

AS_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AS_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AS adapter**. It subscribes to a publish channel configured via `ID` and provides the received **SINT** value, buffered via a D flip-flop (E_D_FF_ANY), at the adapter plug `OUT`. This allows for clean integration of network communication and SINT signal processing in the AS adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AS	Plug (Output)	Buffered received SINT value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.

5. The event EO of the flip-flop is transferred to the event output OUT.E1, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received SINT value is buffered via an internal iec61499::events::E_D_FF_ANY and remains stable at the output until the next value is received. EO only fires when Q changes.
- **RSP is a no-op for subscribers**: According to source code analysis of 4diac-forte (CCommFB::sendData()), the RSP event in SUBSCRIBE_1 (service type subscriber) does not execute any action – the branch != e_Subscriber is skipped. The event input pair INIT/RSP is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of SUBSCRIBE_1 (including IND/RD_1) is moved internally; only the AS adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the INIT event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Receipt**: At IND, the received SINT value is transferred to the internal D flip-flop and output via OUT.

Application Scenarios

- **Network Coupling in AS Applications**: Receiving SINT signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AS adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with AX_PUBLISH_1 and a true event-based E_D_FF_ANY, the feedback between SUBSCRIBE_1 and PUBLISH_1 can be decoupled (a clean fix compared to the interim fix with AX_D_FF).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on AS adapters.

Comparison with Similar Function Blocks

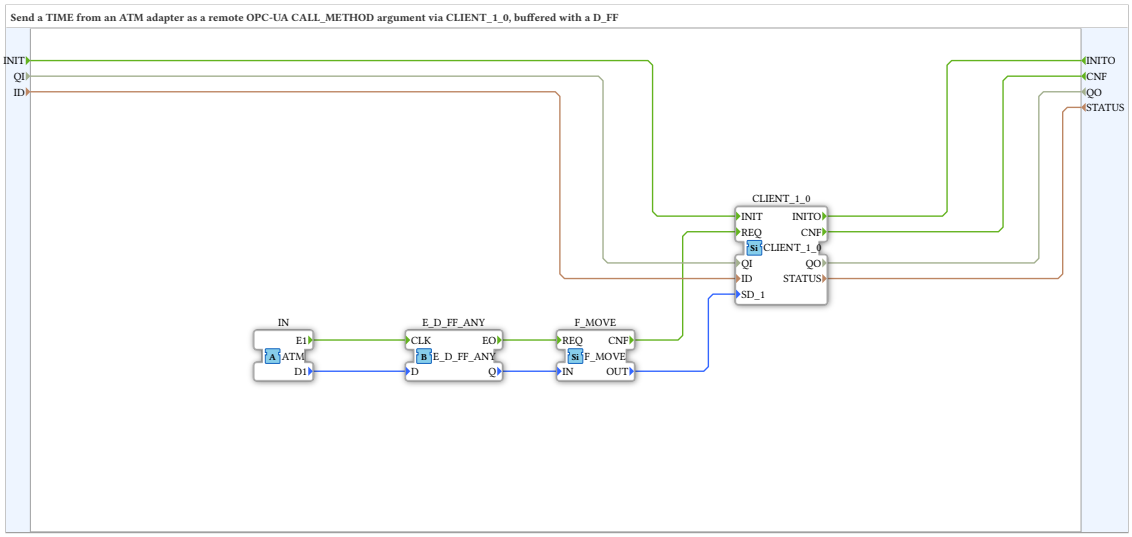
- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (IND, RD_1, …). **AS_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (E_D_FF_ANY), and only provides the AS adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a SINT value at the adapter level but lacks network functionality. **AS_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

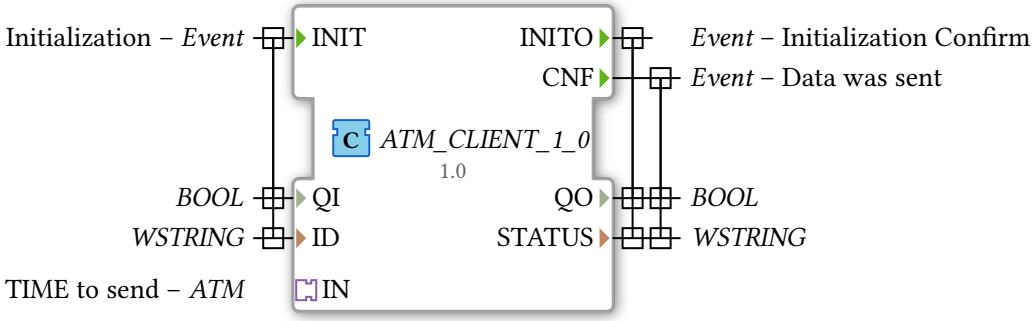
Conclusion

AS_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block SUBSCRIBE_1 with adapter-based SINT processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AS interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AS_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block SUBSCRIBE_1 with adapter-based SINT processing.

ATM_CLIENT_1_0





• – ◦ • – ◦ • – ◦ •

Introduction

The **ATM_CLIENT_1_0** function block is a composite function block that encapsulates the network-based `CLIENT_1_0` function block from the IEC 61499 standard library and maps its interface to a unidirectional **ATM adapter** (`TIME` data type). A **TIME** duration value present at adapter socket `IN` is buffered via an internal `E_D_FF_ANY` flip-flop and then sent via `CLIENT_1_0` as an OPC UA **Write** or `CALL_METHOD` invocation to the remote node configured under `ID`.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, connected to `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, connected to `QO` and `STATUS`
- **CNF** (Event): Confirmation that data was sent, connected to `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Input qualifier, opens (TRUE) or closes (FALSE) the connection to the server
- **ID** (WSTRING): Connection identifier (OPC UA address of the target node, e.g., `opc_ua[WRITE;opc.tcp://192.168.1.12:4840#;...]`)

Data Outputs

- **QO** (BOOL): Output qualifier, connection status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapters

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::ATM	Socket (Input)	TIME value to be sent

Functionality

1. The `INIT` event initializes the internal `CLIENT_1_0` block with `QI` and `ID`, establishing connection to the remote server and acknowledging with `INITO`.
2. When the ATM socket `IN` receives an event on `IN.E1`, the TIME value at `IN.D1` is latched into the internal `E_D_FF_ANY` flip-flop.
3. The flip-flop holds the value stable and generates event `EO`.
4. `EO` triggers `F_MOVE` (`TIME` data type) to supply `CLIENT_1_0.SD_1` and fire the send event `REQ` on `CLIENT_1_0`.
5. After successful transmission, `CLIENT_1_0` confirms with `CNF`.

Technical Features

- **Buffering with E_D_FF_ANY**: The TIME value to be sent is buffered using an internal `E_D_FF_ANY`, preventing value instability during transmission.
- **Remote Write / Call Method**: `CLIENT_1_0` directly addresses a remote OPC UA server to transfer time duration parameters.
- **Encapsulation**: The raw `CLIENT_1_0` event/data interface is kept internal; only the clean ATM adapter interface is exposed.

State Overview

1. **Uninitialized**: The block waits for the `INIT` event.
2. **Initialized**: Connection to the remote server is established; ready to send.
3. **Send Active**: An event on the ATM socket buffers the TIME value and triggers the remote write.

Application Scenarios

- **Remote Time Parameter Transmission**: Sending time duration values (e.g., valve pulse durations, flushing times) to a remote controller via OPC UA.
- **Modular Control Architectures**: Integrating distributed TIME values into libraries built on ATM adapters.

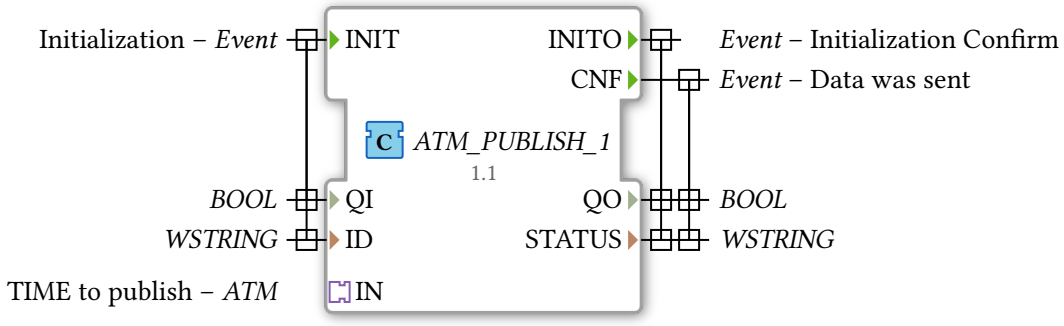
Comparison with Similar Blocks

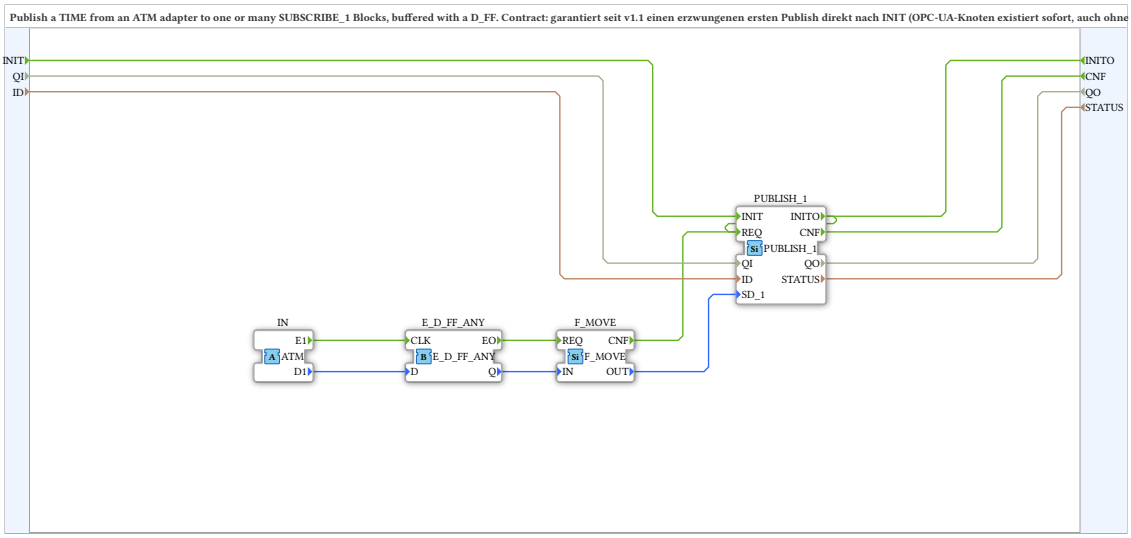
- **AX_CLIENT_1_0**: Handles BOOL values via an AX adapter.
- **AR_CLIENT_1_0**: Handles REAL values via an AR adapter.
- **ATM_PUBLISH_1**: Encapsulates `PUBLISH_1` for local publish/subscribe.

Conclusion

ATM_CLIENT_1_0 allows direct writing of TIME values to a remote OPC UA server over a clean ATM adapter interface.

ATM_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **ATM adapter**. A **TIME** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::ATM	Socket (Input)	TIME value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the ATM socket `IN` delivers an event to `IN.E1`, the TIME value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then sent externally.

Technical Features

- **Buffering with D-Flip-Flop**: The TIME value to be sent is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the ATM adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the ATM socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in ATM Applications**: Sending TIME signals (e.g., switching commands or status bits) from the adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between the user's own publications and subscriptions is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on ATM adapters.

Comparison with Similar Blocks

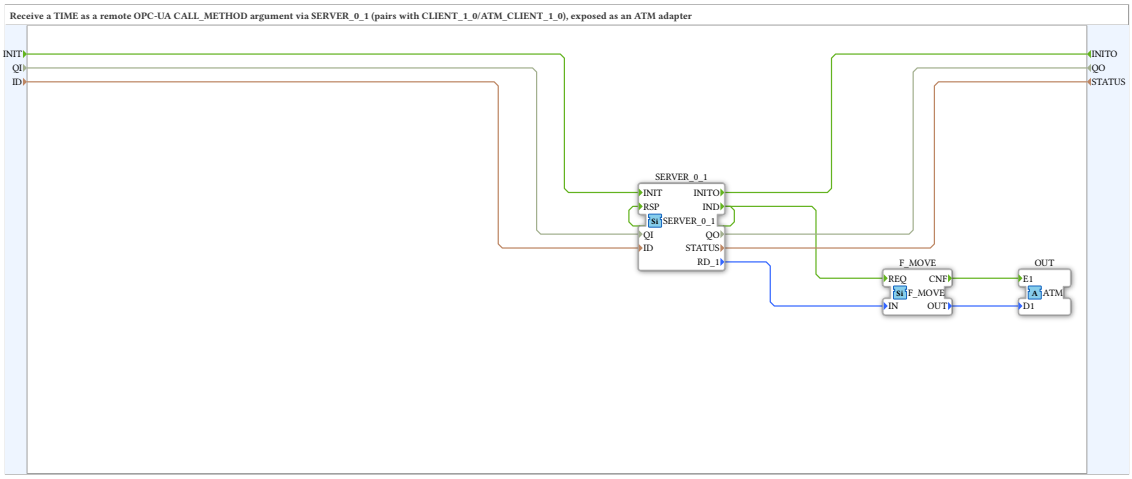
- **PUBLISH_1**: The standard block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **ATM_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the ATM adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a TIME value at the adapter level but lacks network functionality. **ATM_PUBLISH_1** combines buffering and publishing in a single block.

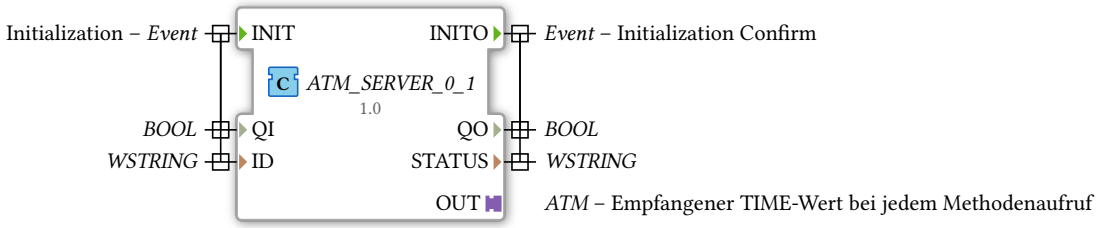
Conclusion

ATM_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based TIME processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the reduced ATM interface makes the function block ideal for fully adapter-based IEC 61499 applications.

ATM_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based TIME processing.

ATM_SERVER_0_1





• – ◦ • – ◦ • – ◦ •

Introduction

The **ATM_SERVER_0_1** function block is a composite function block from the `adapter::net` package. It encapsulates the network server block `iec61499::net::SERVER_0_1` and exposes incoming OPC UA method calls (`CALL_METHOD`) for **TIME** arguments as a unidirectional **ATM adapter** (`OUT`). It serves as the receiving counterpart to **ATM_CLIENT_1_0** (or `CLIENT_1_0`).

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, connected to `QI` and `ID` (forwards to `SERVER_0_1.INIT`)

Event Outputs

- **INITO** (EInit): Initialization confirmation (`SERVER_0_1.INITO`), connected to `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Input qualifier, enables (TRUE) or disables (FALSE) the server
- **ID** (WSTRING): Server identifier (OPC UA server address and method specification)

Data Outputs

- **QO** (BOOL): Output qualifier, server operational status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapters

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::ATM	Plug (Output)	Received TIME value on each method invocation

Functionality

1. An `INIT` event at `ATM_SERVER_0_1` is forwarded to the internal `SERVER_0_1.INIT` input. The server starts the OPC UA endpoint and acknowledges with `INITO` upon success.
2. Upon an incoming method call from a remote client (`CLIENT_1_0`/`ATM_CLIENT_1_0`), `SERVER_0_1` emits indication event `IND` and presents the received TIME value at `RD_1`.
3. To prevent response-latency lockups (RSP latency bug), `SERVER_0_1.IND` is directly looped back to `SERVER_0_1.RSP` within the internal network, causing the server to acknowledge the method immediately.
4. Simultaneously, `IND` passes the received value through `F_MOVE` (`TIME` data type) to ATM plug `OUT.D1` and fires output event `OUT.E1`.

Technical Features

- **Direct RSP Loopback**: To eliminate 4-second server lockups caused by delayed responses, `SERVER_0_1.IND` is internally tied back to `SERVER_0_1.RSP`.
- **Seamless Adapter Integration**: The raw `RD_1` output of `SERVER_0_1` is mapped via `F_MOVE` (TIME) directly onto the ATM adapter structure (`OUT.D1` / `OUT.E1`).
- **Receiving Counterpart to `ATM_CLIENT_1_0`**: In IEC 61499, a `CLIENT_1_0` block (“Connect to a SERVER_0_1 block”) pairs with a `SERVER_0_1` block.

State Overview

1. **Uninitialized**: The server is inactive and waiting for the `INIT` event.
2. **Listening**: The OPC UA server is active and waiting for method calls from remote clients.
3. **Indication**: A method call arrives; the TIME value is passed to the `OUT` adapter and the response is sent immediately.

Application Scenarios

- **Receiving Remote Time Parameters**: Receiving pulse durations or delay times (e.g. valve open duration, flush duration) from a remote module via OPC UA `CALL_METHOD`.
- **Modular Server Architectures**: Providing OPC UA method endpoints for time duration signals in adapter-based applications.

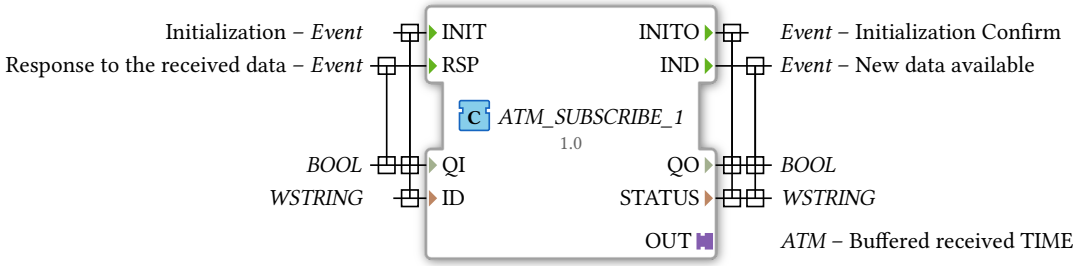
Comparison with Similar Blocks

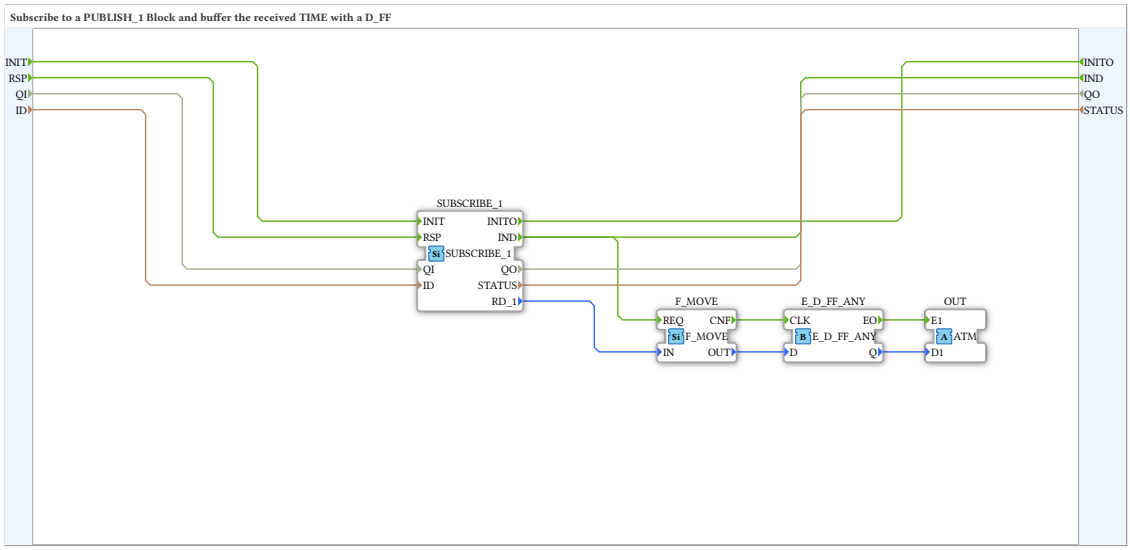
- **AR_SERVER_0_1**: Receives REAL values via an AR adapter.
- **AX_SERVER_0_1**: Receives BOOL values via an AX adapter.
- **ATM_SUBSCRIBE_1**: Encapsulates `SUBSCRIBE_1` for local publish/subscribe rather than method invocations.

Conclusion

ATM_SERVER_0_1 delivers received OPC UA method invocations for TIME values over a clean ATM adapter while preventing server response locks.

ATM_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **ATM_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **ATM adapter**. It subscribes to a publish channel configured via `ID` and provides the received **TIME** value, buffered via a D flip-flop (E_D_FF_ANY), at the adapter plug `OUT`. This allows

for the clean integration of network communication and TIME signal processing in the ATM adapter environment.

Interface Structure

Event Inputs
- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs
- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs
- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs
- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::ATM	Plug (Output)	Buffered received TIME value

Functionality
1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.
5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features
- **Buffering with D flip-flop**: The received TIME value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the ATM adapter interface is visible externally.

State Overview
1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Received**: At `IND`, the received TIME value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios
- **Network Coupling in ATM Applications**: Receiving TIME signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via ATM adapters to the adapter-based logic.

- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a clean fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integrating distributed communication into libraries that rely entirely on ATM adapters.

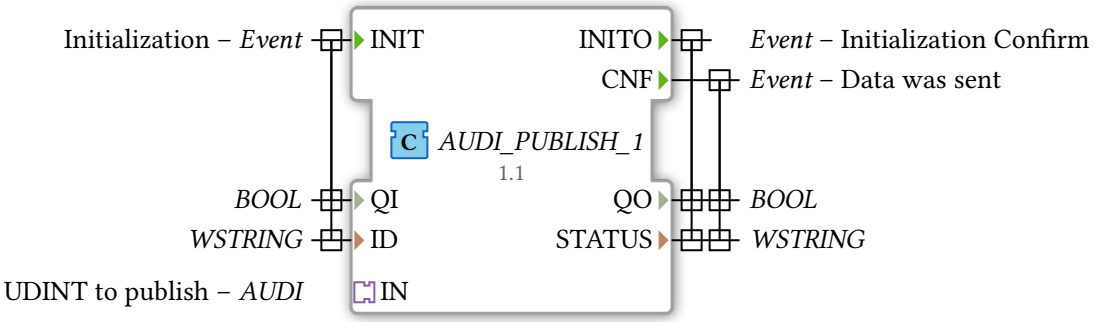
Comparison with Similar Components

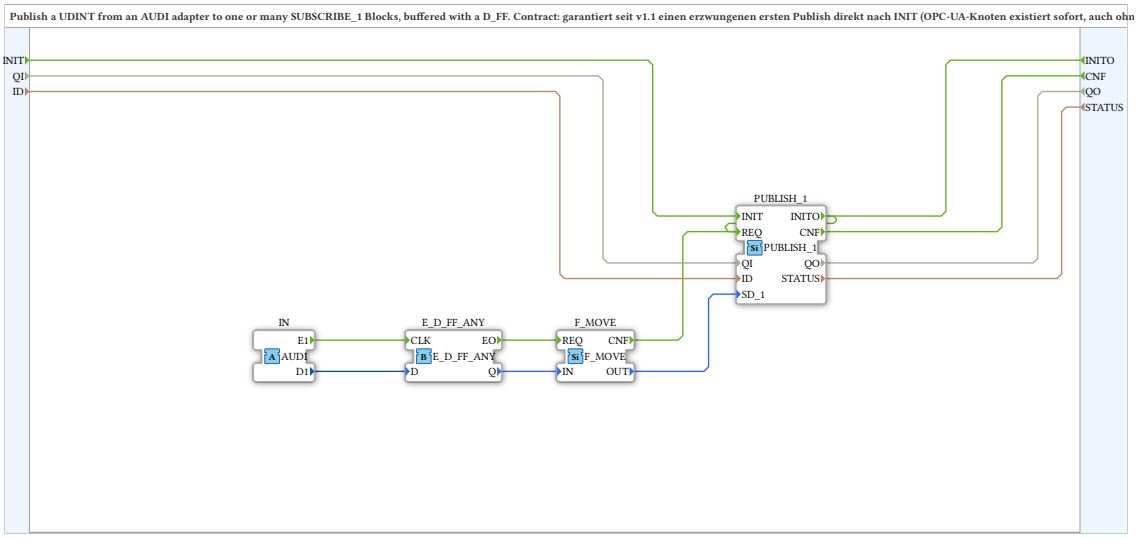
- **SUBSCRIBE_1**: The standard component exposes its interface as events/variables (`IND`, `RD_1`, ...). **ATM_SUBSCRIBE_1** encapsulates this component, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the ATM adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a TIME value at the adapter level but lacks network functionality. **ATM_SUBSCRIBE_1** combines subscribing and buffering in a single component.

Conclusion

ATM_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based TIME processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the reduced ATM interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AUDI_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AUDI_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AUDI adapter**. A **UDINT** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AUDI	Socket (Input)	UDINT value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AUDI socket `IN` sends an event to `IN.E1`, the UDINT value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then transmitted externally.

Technical Features

- **Buffering with D-Flip-Flop**: The UDINT value to be transmitted is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AUDI adapter interface remains visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AUDI socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in AUDI Applications**: Sending UDINT signals (e.g., switching commands or status bits) from adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AUDI adapters.

Comparison with Similar Blocks

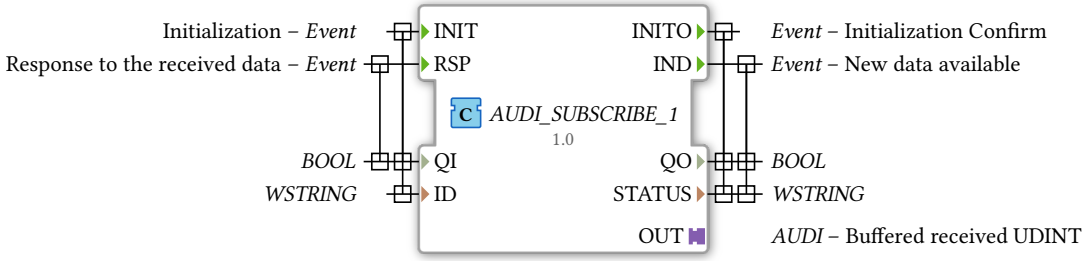
- **PUBLISH_1**: The standard block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AUDI_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AUDI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a UDINT value at the adapter level but lacks network functionality. **AUDI_PUBLISH_1** combines buffering and publishing in a single block.

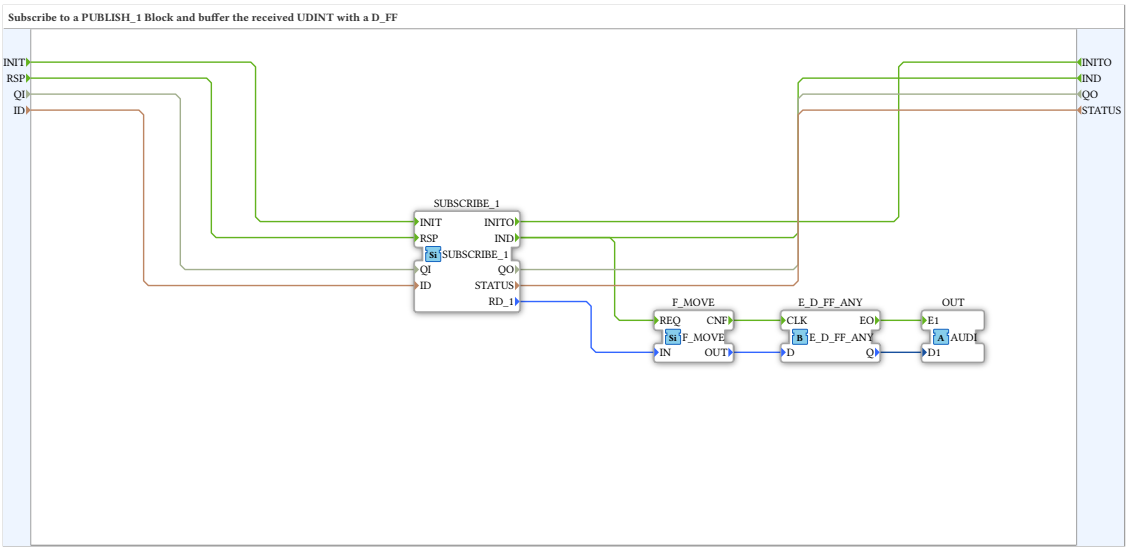
Conclusion

AUDI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based UDINT processing. Internal D flip-flop buffering stabilizes the transmit value and decouples feedback loops; the simplified AUDI interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AUDI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based UDINT processing.

AUDI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AUDI_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AUDI adapter**. It subscribes to a publish channel configured via `ID` and makes the received **UDINT** value buffered via a D flip-flop (E_D_FF_ANY) available at the adapter plug `OUT`. This allows for the clean integration of network communication and UDINT signal processing in the AUDI adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

- **RSP** (Event): Response event, associated with QI

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with QO and STATUS

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AUDI	Plug (Output)	Buffered received UDINT value

Functionality

1. The internal INIT event initializes the internal SUBSCRIBE_1 block with QI and ID. Upon success, it returns INITO.
2. The internal SUBSCRIBE_1 block monitors the configured channel. As soon as the associated PUBLISH_1 block publishes data, it generates the indication event IND and places the received value at RD_1.
3. IND clocks the internal **E_D_FF_ANY** flip-flop (CLK), which receives the value from RD_1 at its data input D.
4. The stored value appears stably at output Q and is transferred via the data connection to OUT.D1 of the AX plug.
5. The event EO of the flip-flop is transferred to the event output OUT.E1, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received UDINT value is buffered via an internal iec61499::events::E_D_FF_ANY and remains stable at the output until the next value is received. EO only fires when Q changes.
- **RSP is a no-op for subscribers**: According to source code analysis of 4diac-forte (CCommFB::sendData()), the RSP event in SUBSCRIBE_1 (service type subscriber) does not execute any action – the branch != e_Subscriber is skipped. The event input pair INIT/RSP is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of SUBSCRIBE_1 (including IND/RD_1) is moved internally; only the AUDI adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the INIT event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Reception**: At IND, the received UDINT value is transferred to the internal D flip-flop and output via OUT.

Application Scenarios

- **Network Coupling in AUDI Applications**: Receiving UDINT signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AUDI adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with AX_PUBLISH_1 and a true event-based E_D_FF_ANY, the feedback between SUBSCRIBE_1 and PUBLISH_1 can be decoupled (a clean fix compared to the interim fix with AX_D_FF).
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AUDI adapters.

Comparison with Similar Components

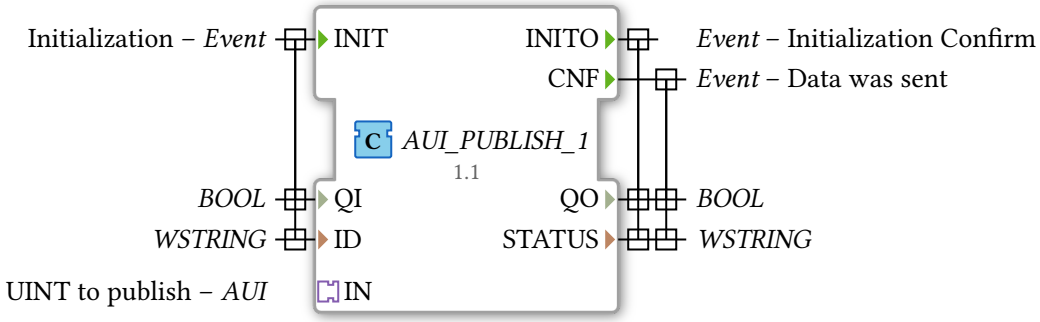
- **SUBSCRIBE_1**: The standard component exposes its interface as events/variables (`IND`, `RD_1`, ...). **AUDI_SUBSCRIBE_1** encapsulates this component, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AUDI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a UDINT value at the adapter level but lacks network functionality. **AUDI_SUBSCRIBE_1** combines subscribing and buffering in a single component.

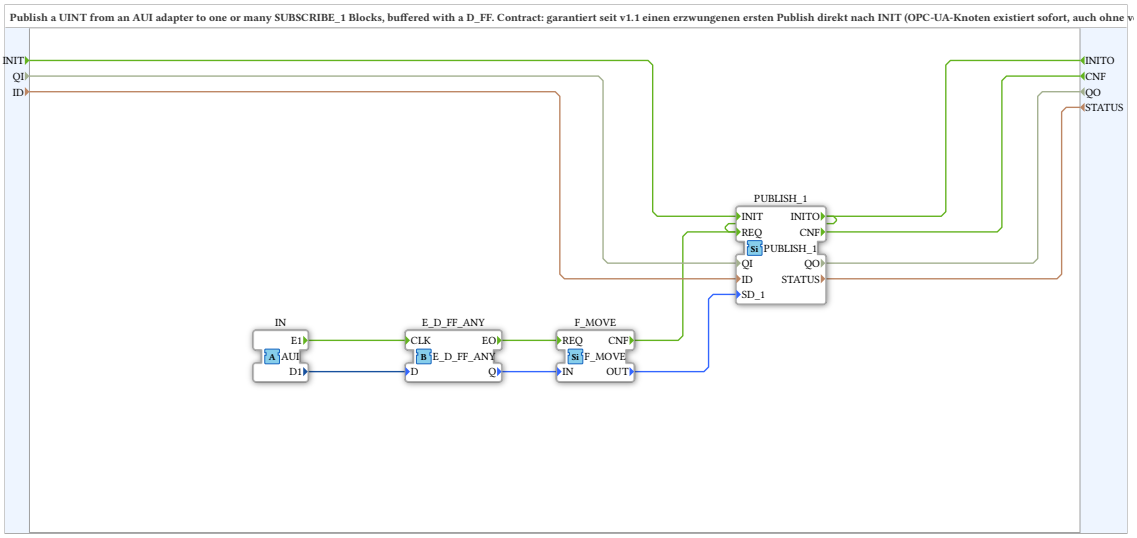
Conclusion

AUDI_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based UDINT processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AUDI interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AUDI_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based UDINT processing.

AUI_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AUI_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AUI adapter**. A **UINT** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AUI	Socket (Input)	UINT value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AUI socket `IN` sends an event to `IN.E1`, the UINT value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then transmitted externally.

Technical Features

- **Buffering with D-Flip-Flop**: The UINT value to be transmitted is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AUI adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AUI socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in AUI Applications**: Sending UINT signals (e.g., switching commands or status bits) from adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AUI adapters.

Comparison with Similar Blocks

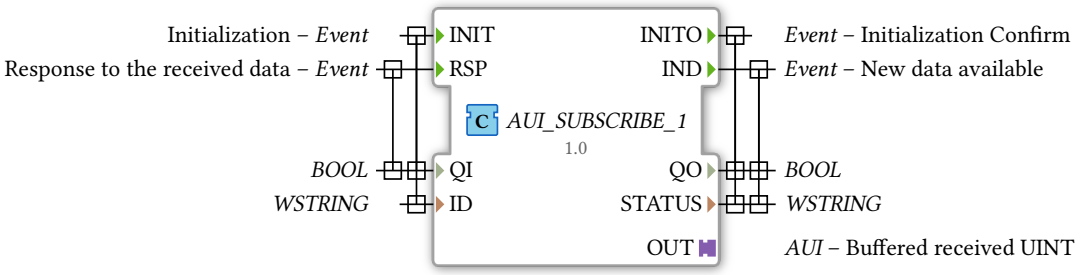
- **PUBLISH_1**: The standard block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AUI_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AUI adapter – making the interface significantly more compact and integrable into adapter networks.

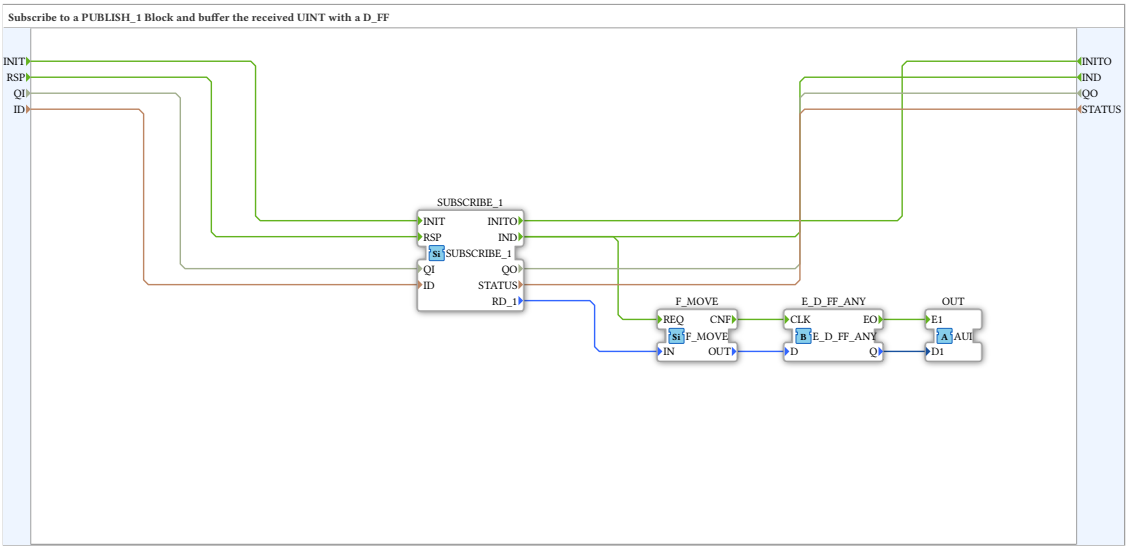
- **AX_D_FF**: Bufferes a UINT value at the adapter level but lacks network functionality. **AUI_PUBLISH_1** combines buffering and publishing in a single block.

Conclusion

AUI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based UINT processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AUI interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AUI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AUI_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AUI adapter**. It subscribes to a publish channel configured via `ID` and makes the received **UINT** value buffered via a D flip-flop (E_D_FF_ANY) available on the adapter plug `OUT`. This allows for clean integration of network communication and UINT signal processing in the AUI adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AUI	Plug (Output)	Buffered received UINT value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.
5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received UINT value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the AUI adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Receipt**: At `IND`, the received UINT value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AUI Applications**: Receiving UINT signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AUI adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a clean fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AUI adapters.

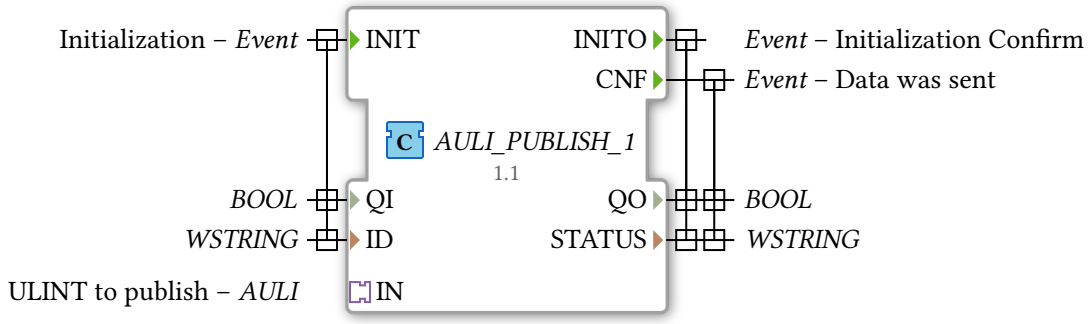
Comparison with Similar Building Blocks

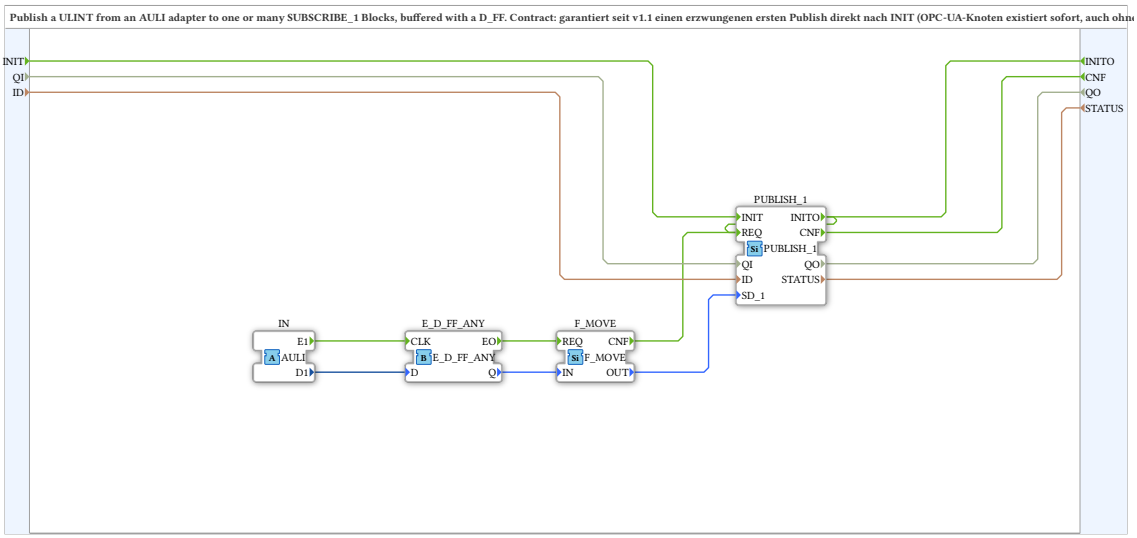
- **SUBSCRIBE_1**: The standard building block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AUI_SUBSCRIBE_1** encapsulates this building block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AUI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a UINT value at the adapter level but lacks network functionality. **AUI_SUBSCRIBE_1** combines subscribing and buffering in a single building block.

Conclusion

AUI_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based UINT processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AUI interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AULI_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AULI adapter**. A **ULINT** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AULI	Socket (Input)	ULINT value to be published

Functionality

1. The internal `PUBLISH_1` block is initialized with `QI` and `ID` via the `INIT` event. Upon success, it returns `INITO`.
2. As soon as the AULI socket `IN` delivers an event to `IN.E1`, the ULINT value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF_ANY** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `PUBLISH_1` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and published on the configured channel.
5. After successful transmission, the `PUBLISH_1` block confirms with `CNF`, which is then transmitted externally.

Technical Features

- **Buffering with D-Flip-Flop**: The ULINT value to be transmitted is buffered via an internal `iec61499::events::E_D_FF_ANY`. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based `E_D_FF_ANY` decouples immediate feedback loops between `PUBLISH_1` and `SUBSCRIBE_1` (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with `AX_D_FF`.
- **Encapsulation**: The original event/data interface of `PUBLISH_1` (including `REQ`/`SD_1`) is moved internally; only the AULI adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AULI socket buffers the value and triggers transmission via the internal `PUBLISH_1` block.

Application Scenarios

- **Network Coupling in AULI Applications**: Sending ULINT signals (e.g., switching commands or status bits) from adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with `AX_SUBSCRIBE_1`, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AULI adapters.

Comparison with Similar Blocks

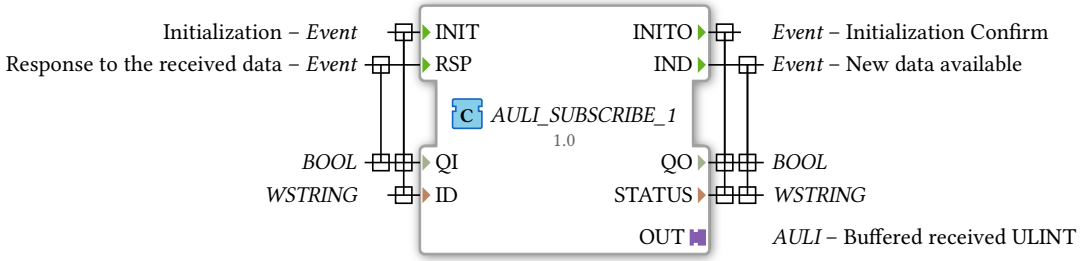
- **PUBLISH_1**: The standard block exposes its interface as events/variables (`REQ`, `SD_1`, ...). **AULI_PUBLISH_1** encapsulates this block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AULI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a ULINT value at the adapter level but lacks network functionality. **AULI_PUBLISH_1** combines buffering and publishing in a single block.

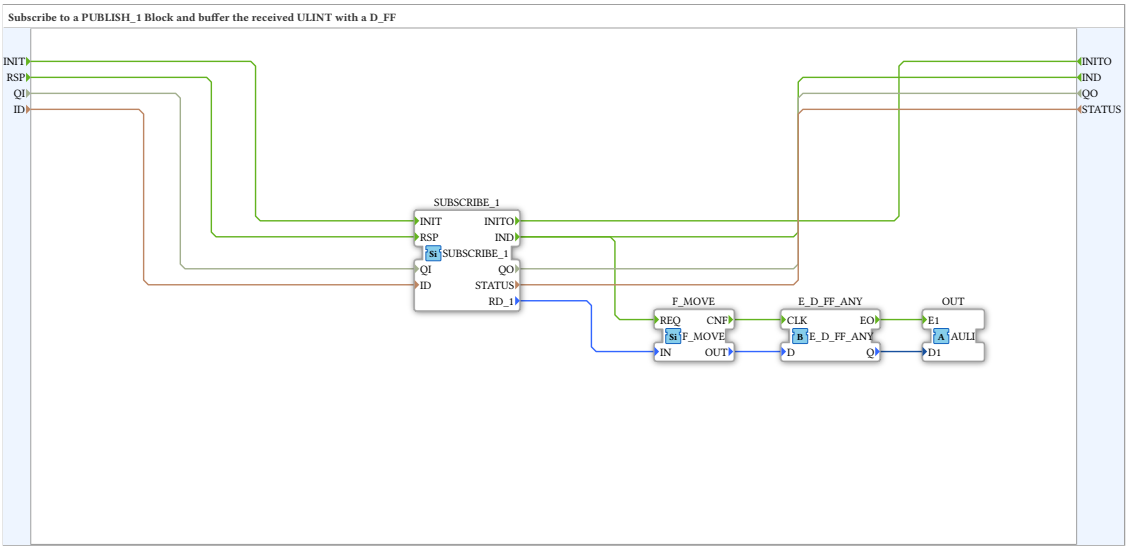
Conclusion

AULI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based ULINT processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AULI interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AULI_PUBLISH_1 combines the publish/subscribe communication of the standard function block `PUBLISH_1` with adapter-based ULINT processing.

AULI_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AULI adapter**. It subscribes to a publish channel configured via `ID` and makes the received **ULINT** value buffered via a D flip-flop (E_D_FF_ANY) available on the adapter plug `OUT`. This allows for the clean integration of network communication and ULINT signal processing in the AULI adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AULI	Plug (Output)	Buffered received ULINT value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.
5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received ULINT value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the AULI adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Receipt**: At `IND`, the received ULINT value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AULI Applications**: Receiving ULINT signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AULI adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a clean fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on AULI adapters.

Comparison with Similar Function Blocks

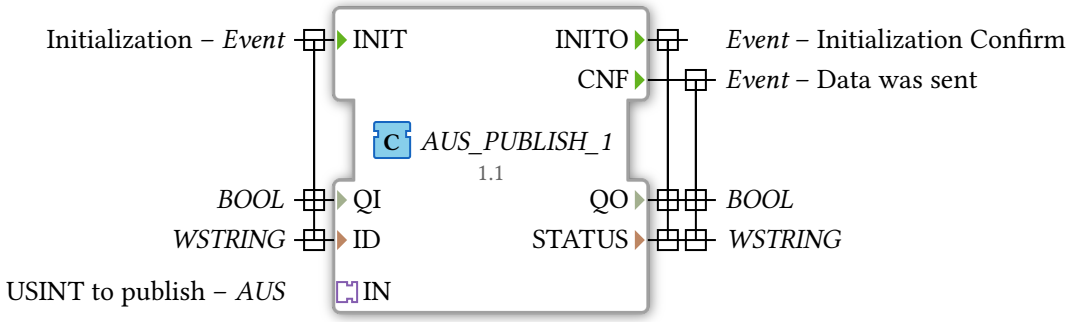
- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AULI_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AULI adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a ULINT value at the adapter level but lacks network functionality. **AULI_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

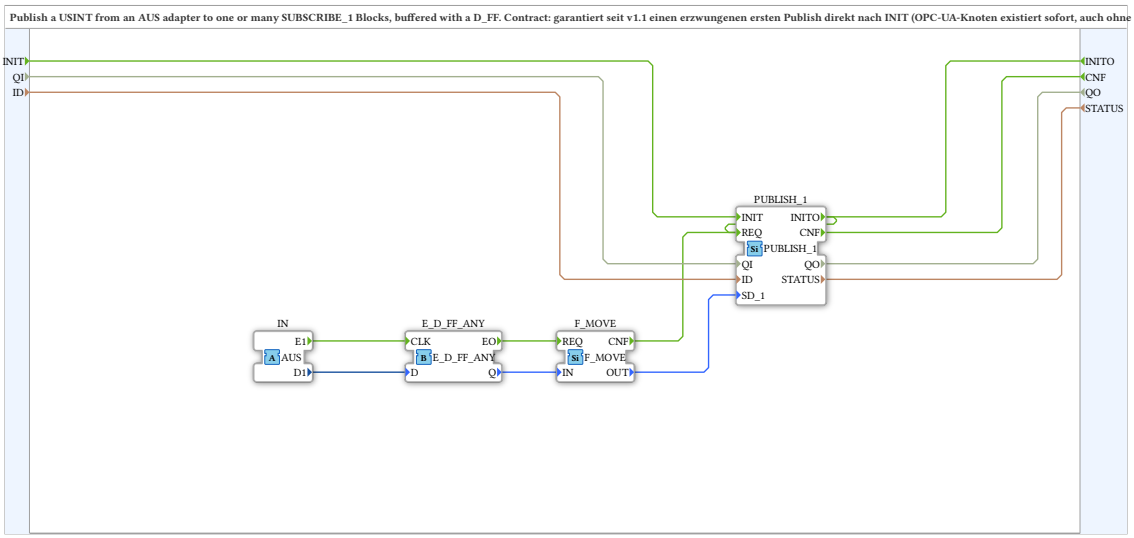
Conclusion

AULI_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based ULINT processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AULI interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AULI_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based ULINT processing.

AUS_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_PUBLISH_1** is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AUS adapter**. A **USINT** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::OFF	Socket (Input)	USINT value to be published

Functionality

1. The internal PUBLISH_1 block is initialized with QI and ID via the INIT event. Upon success, it returns INITO.
2. As soon as the OFF socket IN delivers an event to IN.E1, the USINT value simultaneously present at IN.D1 is transferred to the internal **E_D_FF_ANY** flip-flop (CLK or D).
3. The flip-flop holds the value stably at its output Q and generates the event EO when the value changes.
4. EO triggers the send event REQ of the internal PUBLISH_1 block. The buffered value Q is transmitted over the data connection as SD_1 and published on the configured channel.
5. After successful transmission, the PUBLISH_1 block acknowledges with CNF, which is then transmitted externally.

Technical Features

- **Buffering with D-Flip-Flop**: The USINT value to be transmitted is buffered via an internal iec61499::events::E_D_FF_ANY. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based E_D_FF_ANY decouples immediate feedback loops between PUBLISH_1 and SUBSCRIBE_1 (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with AX_D_FF.
- **Encapsulation**: The original event/data interface of PUBLISH_1 (including REQ/SD_1) is moved internally; only the OFF adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the INIT event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AUS socket buffers the value and triggers transmission via the internal PUBLISH_1 block.

Application Scenarios

- **Network Coupling in AUS Applications**: Sending USINT signals (e.g., switching commands or status bits) from the adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with AX_SUBSCRIBE_1, feedback between the user's own publications and subscriptions is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AUS adapters.

Comparison with Similar Function Blocks

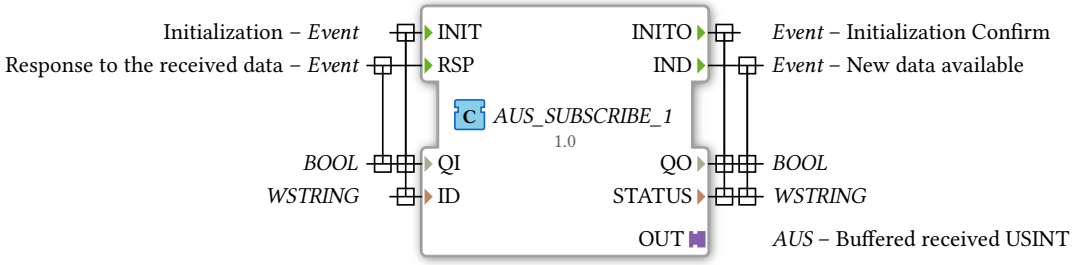
- **PUBLISH_1**: The standard function block exposes its interface as events/variables (REQ, SD_1, ...). **AUS_PUBLISH_1** encapsulates this function block, buffers the value with a D flip-flop (E_D_FF_ANY), and only provides the AUS adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a USINT value at the adapter level but lacks network functionality. **AUS_PUBLISH_1** combines buffering and publishing in a single function block.

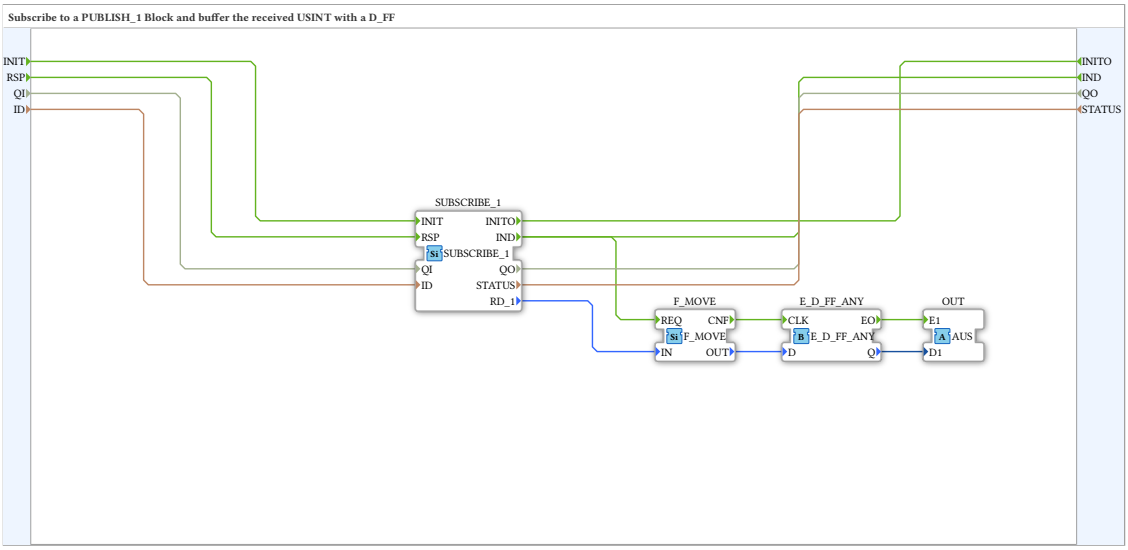
Conclusion

AUS_PUBLISH_1 combines the publish/subscribe communication of the standard function block PUBLISH_1 with adapter-based USINT processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AUS interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AUS_PUBLISH_1 combines the publish/subscribe communication of the standard function block PUBLISH_1 with adapter-based USINT processing.

AUS_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AUS_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AUS adapter**. It subscribes to a publish channel configured via `ID` and makes the received **USINT** value buffered via a D flip-flop (E_D_FF_ANY) available on the adapter plug `OUT`. This allows for clean integration of network communication and USINT signal processing in the AUS adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description

OUT	adapter::types::unidirectional::AUS	Plug (Output)	Buffered received USINT value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.
5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received USINT value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the OFF adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Reception**: At `IND`, the received USINT value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AUS Applications**: Receiving USINT signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AUS adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a cleaner fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integrating distributed communication into libraries that rely entirely on AUS adapters.

Comparison with Similar Function Blocks

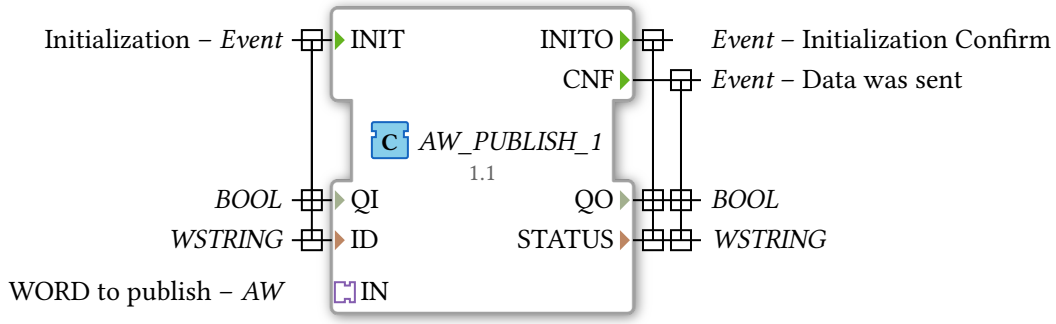
- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AUS_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the OFF adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a USINT value at the adapter level but lacks network functionality. **AUS_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

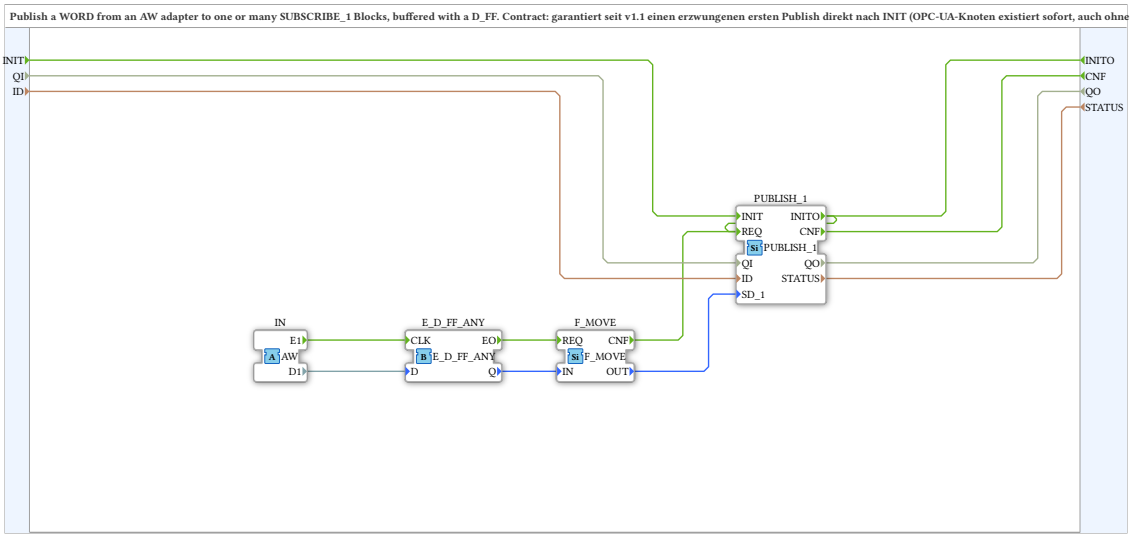
Conclusion

AUS_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based USINT processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the reduced AUS interface makes the function block ideal for end-to-end adapter-based IEC 61499 applications.

AUS_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based USINT processing.

AW_PUBLISH_1





• – ○ • – ○ • – ○ •

Introduction

The **AW_PUBLISH_1** function block is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AW adapter**. A **WORD** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF_ANY) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description

IN	adapter::types::unidirectional::AW	Socket (Input)	WORD value to be published

Functionality

1. The internal PUBLISH_1 block is initialized with QI and ID via the INIT event. Upon success, it returns INITO.
2. As soon as the AW socket IN sends an event to IN.E1, the WORD value simultaneously present at IN.D1 is transferred to the internal **E_D_FF_ANY** flip-flop (CLK or D).
3. The flip-flop holds the value stably at its output Q and generates the event EO when the value changes.
4. EO triggers the send event REQ of the internal PUBLISH_1 block. The buffered value Q is transmitted over the data connection as SD_1 and published on the configured channel.
5. After successful transmission, the PUBLISH_1 block confirms with CNF, which is then sent externally.

Technical Features

- **Buffering with D-Flip-Flop**: The WORD value to be sent is buffered via an internal iec61499::events::E_D_FF_ANY. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based E_D_FF_ANY decouples immediate feedback loops between PUBLISH_1 and SUBSCRIBE_1 (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with AX_D_FF.
- **Encapsulation**: The original event/data interface of PUBLISH_1 (including REQ/SD_1) is moved internally; only the AW adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the INIT event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AW socket buffers the value and triggers transmission via the internal PUBLISH_1 block.

Application Scenarios

- **Network Coupling in AW Applications**: Sending WORD signals (e.g., switching commands or status bits) from the adapter-based logic to remote receivers via Publish/Subscribe.
- **Event Storm Avoidance**: Together with AX_SUBSCRIBE_1, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AW adapters.

Comparison with Similar Function Blocks

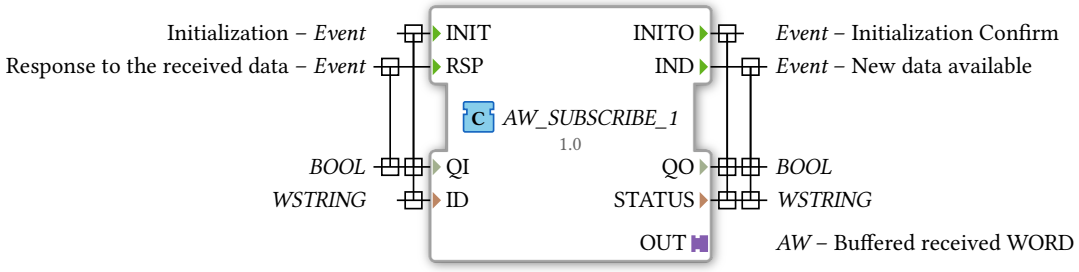
- **PUBLISH_1**: The standard function block exposes its interface as events/variables (REQ, SD_1, ...). **AW_PUBLISH_1** encapsulates this function block, buffers the value with a D flip-flop (E_D_FF_ANY), and only provides the AW adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a WORD value at the adapter level but lacks network functionality. **AW_PUBLISH_1** combines buffering and publishing in a single function block.

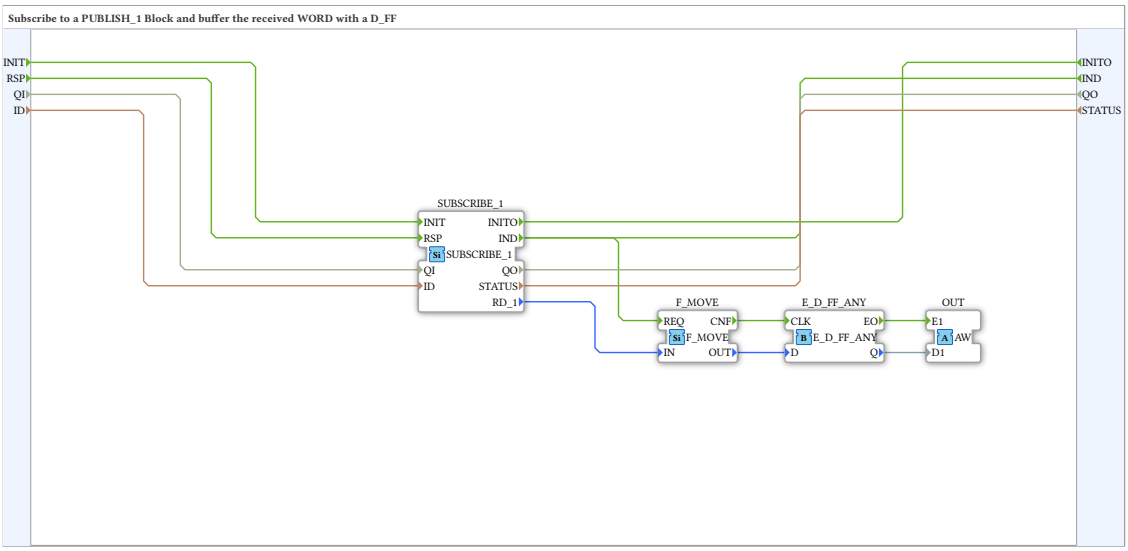
Conclusion

AW_PUBLISH_1 combines the publish/subscribe communication of the standard function block PUBLISH_1 with adapter-based WORD processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AW interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AW_PUBLISH_1 combines the publish/subscribe communication of the standard function block PUBLISH_1 with adapter-based WORD processing.

AW_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AW_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AW adapter**. It subscribes to a publish channel configured via `ID` and makes the received **WORD** value buffered via a D flip-flop (E_D_FF_ANY) available at the adapter plug `OUT`. This allows for clean integration of network communication and WORD signal processing in the AW adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description

OUT	adapter::types::unidirectional::AW	Plug (Output)	Buffered received WORD value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF_ANY** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.
5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received WORD value is buffered via an internal `iec61499::events::E_D_FF_ANY` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the AW adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Received**: At `IND`, the received WORD value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AW Applications**: Receiving WORD signals (e.g., switching commands or status bits) via Publish/Subscribe and forwarding them via AW adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF_ANY`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a cleaner fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on AW adapters.

Comparison with Similar Building Blocks

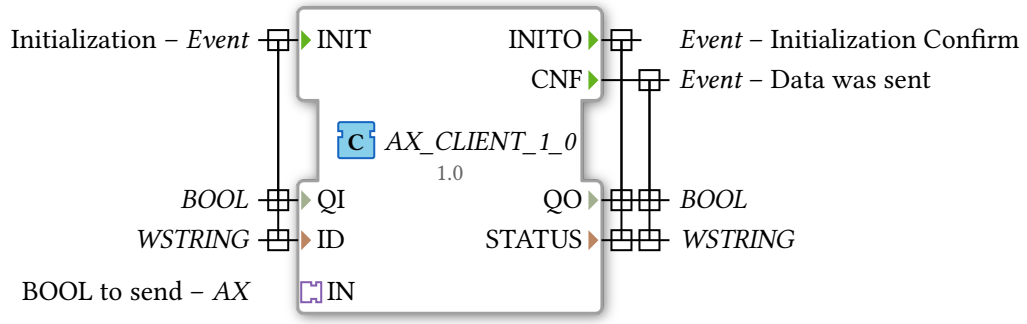
- **SUBSCRIBE_1**: The standard building block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AW_SUBSCRIBE_1** encapsulates this building block, buffers the value with a D flip-flop (`E_D_FF_ANY`), and only provides the AW adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a WORD value at the adapter level but lacks network functionality. **AW_SUBSCRIBE_1** combines subscribing and buffering in a single building block.

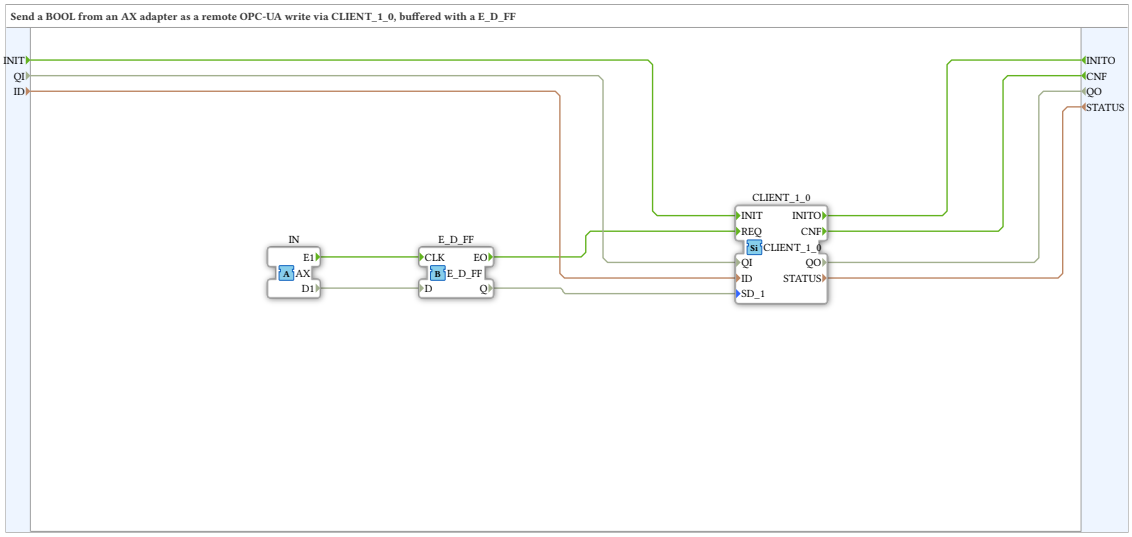
Conclusion

AW_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based WORD processing. Internal D flip-flop buffering keeps the received value stable until new data arrives, and the simplified AW interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AW_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based WORD processing.

AX_CLIENT_1_0





• – ◦ • – ◦ • – ◦ •

Introduction

The **AX_CLIENT_1_0** function block is a composite function block that encapsulates the network-based `CLIENT_1_0` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AX adapter**. A **BOOL** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF) and then sent via `CLIENT_1_0` as an OPC UA **write** to the remote node configured in `ID`.

The generic `CLIENT_1_0` network client writes to a **remote** server identified by an `ID` that corresponds to a connection on the server block. For OPC UA, an `ID` such as `opc_ua[WRITE;opc.tcp://192.168.1.12:4840#;...]` is a transport-specific example, not a requirement; unlike **AX_PUBLISH_1** (local publish/subscribe), `CLIENT_1_0` can write to a remote server.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, opens (TRUE) or closes (FALSE) the connection to the server
- **ID** (WSTRING): Connection identifier (OPC UA address of the target node, e.g. `opc_ua[WRITE;opc.tcp://192.168.1.12:4840#;...]`)

Data Outputs

- **QO** (BOOL): Qualifier output, connection status

- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AX	Socket (Input)	BOOL value to be sent

Functionality

1. The internal `CLIENT_1_0` block is initialized with `QI` and `ID` via the `INIT` event; it establishes the connection to the remote server. Upon success, it returns `INITO`.
2. As soon as the AX socket `IN` sends an event to `IN.E1`, the BOOL value simultaneously present at `IN.D1` is transferred to the internal **E_D_FF** flip-flop (`CLK` or `D`).
3. The flip-flop holds the value stably at its output `Q` and generates the event `EO` when the value changes.
4. `EO` triggers the send event `REQ` of the internal `CLIENT_1_0` block. The buffered value `Q` is transmitted over the data connection as `SD_1` and sent to the server as an OPC UA write.
5. After successful transmission, the `CLIENT_1_0` block confirms with `CNF`, which is then sent out.

Technical Features

- **Buffering with D-Flip-Flop**: The BOOL value to be sent is buffered via an internal `iec61499::events::E_D_FF`. This prevents a changing input value during processing from interfering with the transmission.
- **Remote write instead of local publish**: `CLIENT_1_0` addresses a remote OPC UA server directly (ACTION=WRITE with a full ENDPOINT in `ID`), unlike `PUBLISH_1`/`SUBSCRIBE_1`, which only work locally on the same FORTE node.
- **Encapsulation**: The original event/data interface of `CLIENT_1_0` (including `REQ`/`SD_1`) is moved internally; only the AX adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The connection to the remote server is established, the block is ready to send.
3. **Send Active**: An event arriving at the AX socket buffers the value and triggers the remote write via the internal `CLIENT_1_0` block.

Application Scenarios

- **Remote control of other modules**: A VT button press on a module with its own ISOBUS/VT connection triggers an OPC UA write to a different, remote control module (e.g. STG1 switches an output on STG2–STG5).
- **Network coupling in AX applications**: Sending Boolean signals (switching commands) from adapter-based logic directly to a remote OPC UA server.
- **Modular control architectures**: Integration of distributed remote control into libraries that consistently rely on AX adapters.

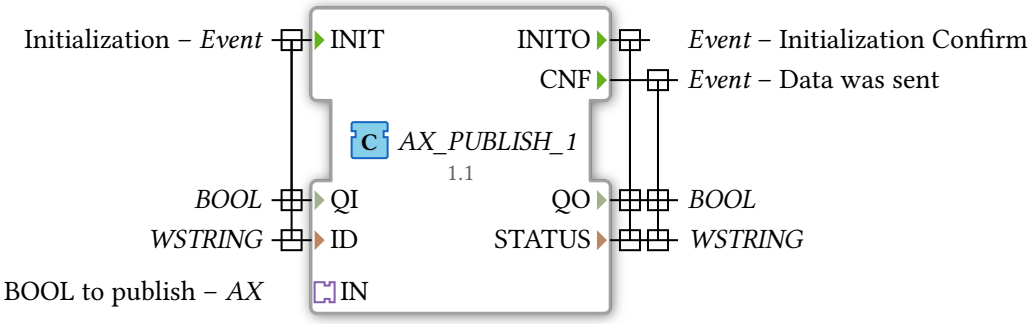
Comparison with Similar Function Blocks

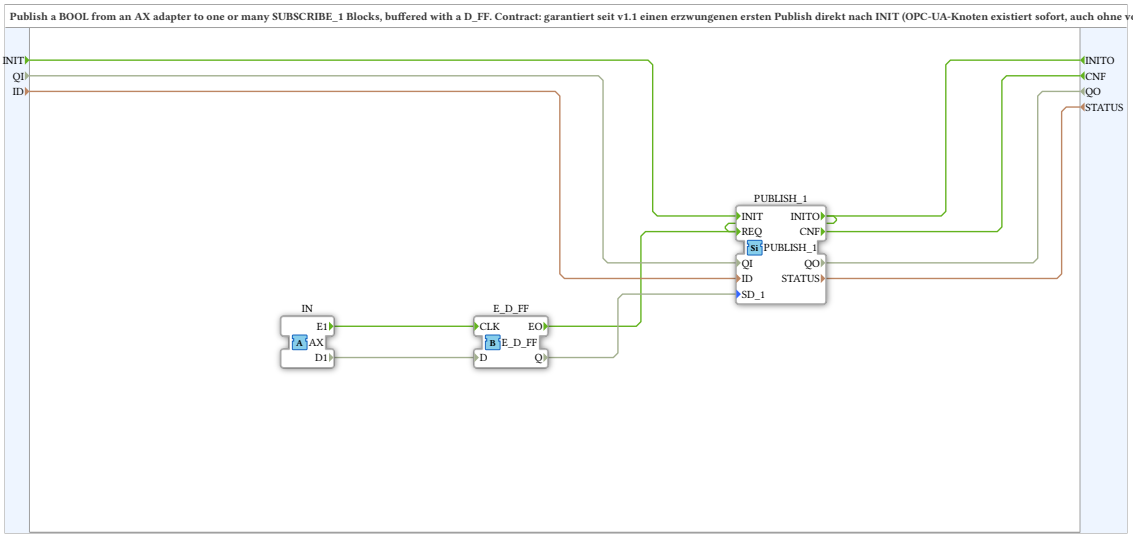
- **CLIENT_1_0**: The standard function block exposes its interface as events/variables (`REQ`, `SD_1`, …). **AX_CLIENT_1_0** encapsulates this function block, buffers the value with a D flip-flop, and only provides the AX adapter.
- **AX_PUBLISH_1**: Structurally identical (same events/variables/adapter), but encapsulates `PUBLISH_1` instead of `CLIENT_1_0` – so it only publishes locally instead of actively writing to a remote node.

Conclusion

AX_CLIENT_1_0 combines the connection-oriented remote-write communication of the standard function block `CLIENT_1_0` with adapter-based Boolean processing. Internal D flip-flop buffering stabilizes the transmitted value; the simplified AX interface makes the function block ideal for writing directly to a remote OPC UA server in fully adapter-based IEC 61499 applications.

AX_PUBLISH_1





• – ◦ • – ◦ • – ◦ •

Introduction

The **AX_PUBLISH_1** function block is a composite function block that encapsulates the network-based `PUBLISH_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AX adapter**. A **BOOL** value present at the adapter socket `IN` is buffered via an internal D flip-flop (E_D_FF) and then sent to all associated `SUBSCRIBE_1` receivers via the configured publish channel.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`
- **CNF** (Event): Confirmation that data has been sent, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the publish service
- **ID** (WSTRING): Publish channel identifier

Data Outputs

- **QO** (BOOL): Qualifier output, block execution status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description

IN	adapter::types::unidirectional::AX	Socket (Input)	BOOL value to be published

Functionality

1. The internal PUBLISH_1 block is initialized with QI and ID via the INIT event. Upon success, it returns INITO.
2. As soon as the AX socket IN sends an event to IN.E1, the BOOL value simultaneously present at IN.D1 is transferred to the internal **E_D_FF** flip-flop (CLK or D).
3. The flip-flop holds the value stably at its output Q and generates the event EO when the value changes.
4. EO triggers the send event REQ of the internal PUBLISH_1 block. The buffered value Q is transmitted over the data connection as SD_1 and published on the configured channel.
5. After successful transmission, the PUBLISH_1 block confirms with CNF, which is then sent out.

Technical Features

- **Buffering with D-Flip-Flop**: The BOOL value to be sent is buffered via an internal iec61499::events::E_D_FF. This prevents a changing input value during processing from interfering with the transmission.
- **Event-based decoupling**: The true event-based E_D_FF decouples immediate feedback loops between PUBLISH_1 and SUBSCRIBE_1 (e.g., on the same OPC UA node) – a clean fix compared to the interim fix with AX_D_FF.
- **Encapsulation**: The original event/data interface of PUBLISH_1 (including REQ/SD_1) is moved internally; only the AX adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the INIT event.
2. **Initialized**: The block is ready to publish data via the configured channel.
3. **Send Active**: An event arriving at the AX socket buffers the value and triggers transmission via the internal PUBLISH_1 block.

Application Scenarios

- **Network Coupling in AX Applications**: Sending Boolean signals (e.g., switching commands or status bits) from adapter-based logic to remote receivers via publish/subscribe.
- **Event Storm Avoidance**: Together with AX_SUBSCRIBE_1, feedback between your own publications and your own subscription is decoupled.
- **Modular Control Architectures**: Integration of distributed communication into libraries that consistently rely on AX adapters.

Comparison with Similar Function Blocks

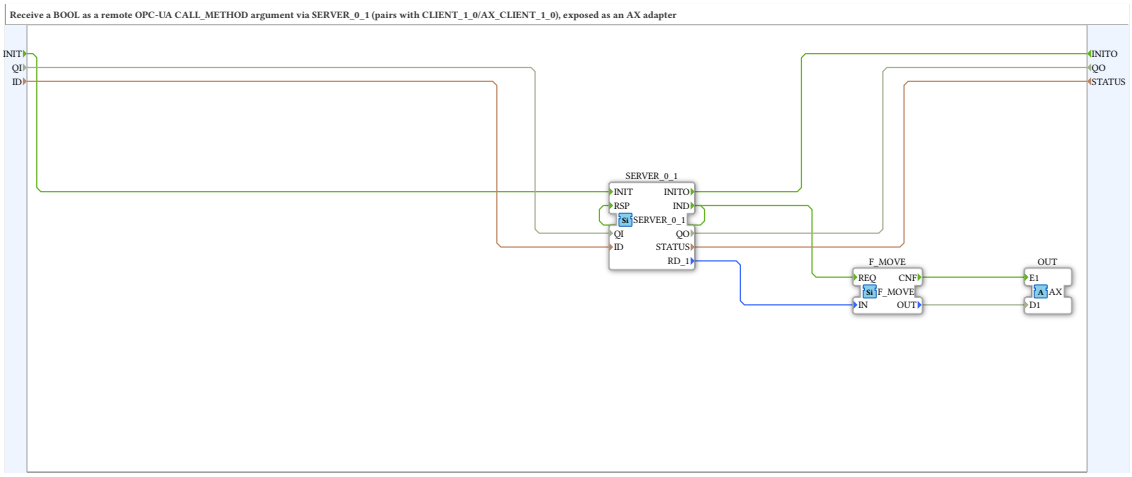
- **PUBLISH_1**: The standard function block exposes its interface as events/variables (REQ, SD_1, ...). **AX_PUBLISH_1** encapsulates this function block, buffers the value with a D flip-flop, and only provides the AX adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a Boolean value at the adapter level but lacks network functionality. **AX_PUBLISH_1** combines buffering and publishing in a single function block.

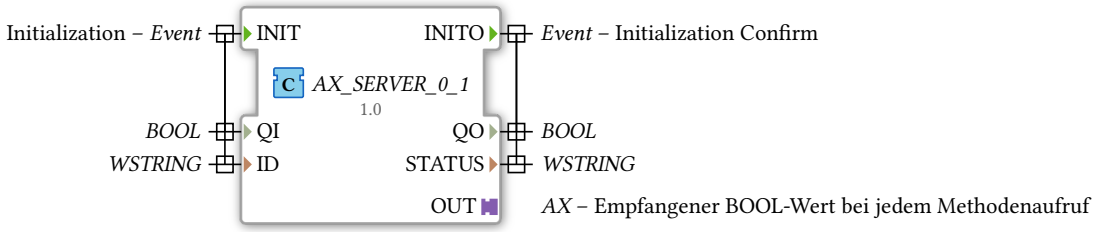
Conclusion

AX_PUBLISH_1 combines the publish/subscribe communication of the standard function block PUBLISH_1 with adapter-based Boolean processing. Internal D flip-flop buffering stabilizes the transmitted value and decouples feedback loops; the simplified AX interface makes the function block ideal for fully adapter-based IEC 61499 applications.

AX_PUBLISH_1 combines the publish/subscribe communication of the standard function block PUBLISH_1 with adapter-based Boolean processing.

AX_SERVER_0_1





• – ◦ • – ◦ • – ◦ •

Introduction

The **AX_SERVER_0_1** function block is a composite function block from the `adapter::net` package. It encapsulates the network server block `iec61499::net::SERVER_0_1` and exposes incoming OPC UA method calls (`CALL_METHOD`) for **BOOL** arguments as a unidirectional **AX adapter** (`OUT`). It serves as the receiving counterpart to **AX_CLIENT_1_0** (or `CLIENT_1_0`).

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, connected to `QI` and `ID` (forwards to `SERVER_0_1.INIT`)

Event Outputs

- **INITO** (EInit): Initialization confirmation (`SERVER_0_1.INITO`), connected to `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Input qualifier, enables (TRUE) or disables (FALSE) the server
- **ID** (WSTRING): Server identifier (OPC UA server address and method specification)

Data Outputs

- **QO** (BOOL): Output qualifier, server operational status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapters

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AX	Plug (Output)	Received BOOL value on each method invocation

Functionality

1. An `INIT` event at `AX_SERVER_0_1` is forwarded to the internal `SERVER_0_1.INIT` input. The server starts the OPC UA endpoint and acknowledges with `INITO` upon success.

2. Upon an incoming method call from a remote client (`CLIENT_1_0`/`AX_CLIENT_1_0`), `SERVER_0_1` emits indication event `IND` and presents the received BOOL value at `RD_1`.
3. To prevent response-latency lockups (RSP latency bug), `SERVER_0_1.IND` is directly looped back to `SERVER_0_1.RSP` within the internal network, causing the server to acknowledge the method immediately.
4. Simultaneously, `IND` passes the received value through `F_MOVE` (`BOOL` data type) to AX plug `OUT.D1` and fires output event `OUT.E1`.

Technical Features

- **Direct RSP Loopback**: To eliminate 4-second server lockups caused by delayed responses, `SERVER_0_1.IND` is internally tied back to `SERVER_0_1.RSP`.
- **Seamless Adapter Integration**: The raw `RD_1` output of `SERVER_0_1` is mapped via `F_MOVE` (BOOL) directly onto the AX adapter structure (`OUT.D1` / `OUT.E1`).
- **Receiving Counterpart to `AX_CLIENT_1_0`**: In IEC 61499, a `CLIENT_1_0` block (“Connect to a SERVER_0_1 block”) pairs with a `SERVER_0_1` block.

State Overview

1. **Uninitialized**: The server is inactive and waiting for the `INIT` event.
2. **Listening**: The OPC UA server is active and waiting for method calls from remote clients.
3. **Indication**: A method call arrives; the BOOL value is passed to the `OUT` adapter and the response is sent immediately.

Application Scenarios

- **Receiving Remote Control Commands**: Receiving discrete switching commands (e.g. start/stop, enable signals) from a remote module via OPC UA `CALL_METHOD`.
- **Modular Server Architectures**: Providing OPC UA method endpoints for Boolean signals in adapter-based applications.

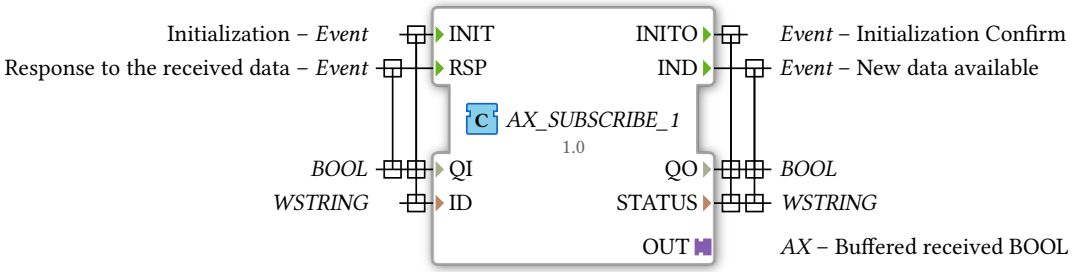
Comparison with Similar Blocks

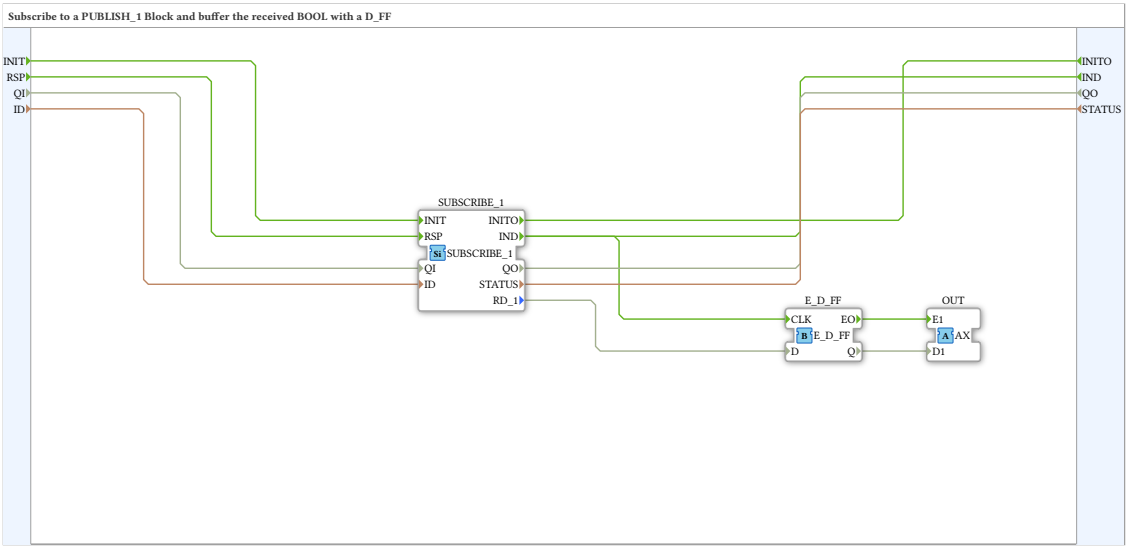
- **AR_SERVER_0_1**: Receives REAL values via an AR adapter.
- **ATM_SERVER_0_1**: Receives TIME values via an ATM adapter.
- **AX_SUBSCRIBE_1**: Encapsulates `SUBSCRIBE_1` for local publish/subscribe rather than method invocations.

Conclusion

AX_SERVER_0_1 delivers received OPC UA method invocations for BOOL values over a clean AX adapter while preventing server response locks.

AX_SUBSCRIBE_1





• – ○ • – ○ • – ○ •

Introduction

The function block **AX_SUBSCRIBE_1** is a composite function block that encapsulates the network-based `SUBSCRIBE_1` function block from the IEC 61499 standard library and maps its interface to a unidirectional **AX adapter**. It subscribes to a publish channel configured via `ID` and provides the received **BOOL** value, buffered via a D flip-flop (E_D_FF), on the adapter plug `OUT`. This allows for clean integration of network communication and Bool signal processing in the AX adapter environment.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization event, associated with `QI` and `ID`
- **RSP** (Event): Response event, associated with `QI`

Event Outputs

- **INITO** (EInit): Initialization confirmation, associated with `QO` and `STATUS`

Data Inputs

- **QI** (BOOL): Qualifier input, activates the subscription service
- **ID** (WSTRING): Identifier of the publish channel to be subscribed to

Data Outputs

- **QO** (BOOL): Qualifier output, status of the Block Execution
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AX	Plug (Output)	Buffered received BOOL value

Functionality

1. The internal `INIT` event initializes the internal `SUBSCRIBE_1` block with `QI` and `ID`. Upon success, it returns `INITO`.
2. The internal `SUBSCRIBE_1` block monitors the configured channel. As soon as the associated `PUBLISH_1` block publishes data, it generates the indication event `IND` and places the received value at `RD_1`.
3. `IND` clocks the internal **E_D_FF** flip-flop (`CLK`), which receives the value from `RD_1` at its data input `D`.
4. The stored value appears stably at output `Q` and is transferred via the data connection to `OUT.D1` of the AX plug.

5. The event `EO` of the flip-flop is transferred to the event output `OUT.E1`, so that downstream components are notified of the new information.

Technical Features

- **Buffering with D flip-flop**: The received BOOL value is buffered via an internal `iec61499::events::E_D_FF` and remains stable at the output until the next value is received. `EO` only fires when `Q` changes.
- **RSP is a no-op for subscribers**: According to source code analysis of `4diac-forte` (`CCommFB::sendData()`), the `RSP` event in `SUBSCRIBE_1` (service type subscriber) does not execute any action – the branch `!= e_Subscriber` is skipped. The event input pair `INIT`/`RSP` is only executed for acknowledgment and is non-functional for acknowledgment in the subscriber case. It is nevertheless routed externally to maintain a configurable connection.
- **Encapsulation**: The original event/data interface of `SUBSCRIBE_1` (including `IND`/`RD_1`) is moved internally; only the AX adapter interface is visible externally.

State Overview

1. **Not Initialized**: The block is waiting for the `INIT` event.
2. **Initialized**: The subscription is active; the block is waiting for data from the publisher.
3. **Data Received**: At `IND`, the received BOOL value is transferred to the internal D flip-flop and output via `OUT`.

Application Scenarios

- **Network Coupling in AX Applications**: Receiving Boolean signals (e.g., switching commands or status bits) via publish/subscribe and forwarding them via AX adapters to the adapter-based logic.
- **Event Storm Avoidance**: Together with `AX_PUBLISH_1` and a true event-based `E_D_FF`, the feedback between `SUBSCRIBE_1` and `PUBLISH_1` can be decoupled (a cleaner fix compared to the interim fix with `AX_D_FF`).
- **Modular Control Architectures**: Integration of distributed communication into libraries that rely entirely on AX adapters.

Comparison with Similar Function Blocks

- **SUBSCRIBE_1**: The standard function block exposes its interface as events/variables (`IND`, `RD_1`, ...). **AX_SUBSCRIBE_1** encapsulates this function block, buffers the value with a D flip-flop, and only provides the AX adapter – making the interface significantly more compact and integrable into adapter networks.
- **AX_D_FF**: Bufferes a Boolean value at the adapter level but lacks network functionality. **AX_SUBSCRIBE_1** combines subscribing and buffering in a single function block.

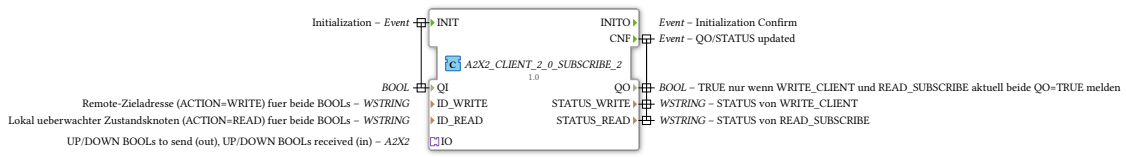
Conclusion

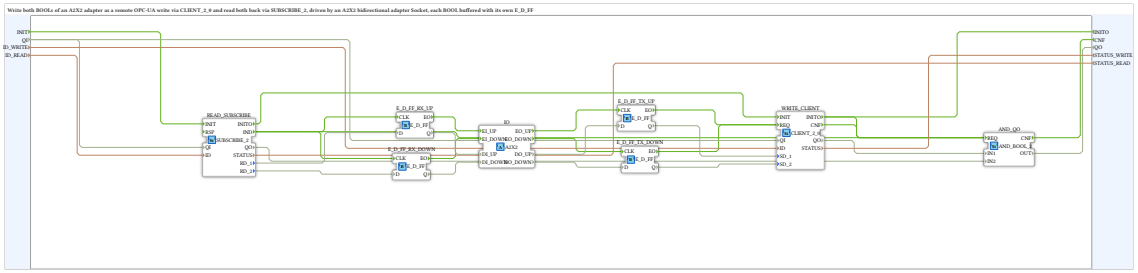
AX_SUBSCRIBE_1 combines the publish/subscribe communication of the standard function block `SUBSCRIBE_1` with adapter-based Boolean processing. The internal D-flip-flop buffer keeps the received value stable until new data arrives, and the reduced AX interface makes the device ideal for end-to-end adapter-based IEC 61499 applications.

The internal D-flip-flop buffer keeps the received value stable until new data arrives, and the reduced AX interface makes the device ideal for end-to-end adapter-based IEC 61499 applications.

BOOL

A2X2_CLIENT_2_0_SUBSCRIBE_2





• — ○ • — ○ • — ○ •

Introduction

The function block **A2X2_CLIENT_2_0_SUBSCRIBE_2** is a composite FB that bridges between a bidirectional A2X2 adapter (with two BOOL channels, UP and DOWN) and an OPC-UA server. It uses the standard `CLIENT_2_0` and `SUBSCRIBE_2` function blocks to write two BOOL values to a remote OPC-UA node and read two BOOL values from a local state node. Each input/output BOOL is individually buffered using `E_D_FF` flip-flops to ensure stable signal transfer and to decouple event sequences.

Interface Structure

Event Inputs

Event	Type	Comment
`INIT`	EInit	Initialization event. Resets the FB and starts the data flow.

Event Outputs

Event	Type	Comment
`INITO`	EInit	Initialization confirm. Emitted after both internal FB initializations complete.
`CNF`	Event	Confirmation event. Emitted when `QO`, `STATUS_WRITE`, and `STATUS_READ` are updated.

Data Inputs

Name	Type	Comment
`QI`	BOOL	Enable input. When `FALSE`, the FB is inactive and outputs are reset.
`ID_WRITE`	WSTRING	Remote target address for the write operation (both BOOLs).
`ID_READ`	WSTRING	Local state node address to monitor (both BOOLs).

Data Outputs

Name	Type	Comment
`QO`	BOOL	`TRUE` only when both the write client and the read subscriber report `QO = TRUE`.
`STATUS_WRITE`	WSTRING	Status string from the internal `WRITE_CLIENT` FB.
`STATUS_READ`	WSTRING	Status string from the internal `READ_SUBSCRIBE` FB.

Adapters

Adapter	Type	Comment
`IO`	`adapter::types::bidirectional::A2X2`	Bidirectional BOOL channels: `DO_UP`/`DO_DOWN` (send) and `DI_UP`/`DI_DOWN` (receive).

Functionality

The FB operates in two independent data paths:

1. **Write Path**: BOOL values received from the adapter (`IO.DO_UP`, `IO.DO_DOWN`) are captured in the corresponding `E_D_FF_TX_UP` and `E_D_FF_TX_DOWN` flip-flops. Each event at `IO.EO_UP` or `IO.EO_DOWN` triggers a `REQ` on the `WRITE_CLIENT` FB, which sends the buffered values to the remote OPC-UA node via `SD_1` and `SD_2`.
2. **Read Path**: The `READ_SUBSCRIBE` FB continuously monitors the configured `ID_READ` node. Upon each `IND` event, its `RD_1` and `RD_2` data outputs are fed into the `E_D_FF_RX_UP` and `E_D_FF_RX_DOWN` flip-flops. Their stored values are then presented on `IO.DI_UP` and `IO.DI_DOWN` and emitted via the corresponding event outputs (`IO.EI_UP`, `IO.EI_DOWN`).

The internal `AND_QO` block combines the `QO` signals from both the write client and the read subscriber, producing the overall `QO` output. The `CNF` event is generated whenever either the write client finishes a write operation (`WRITE_CLIENT.CNF`) or the read subscriber receives new data (`READ_SUBSCRIBE.IND`), ensuring that the status outputs are always current.

Technical Features

- Uses standard IEC 61499 `CLIENT_2_0` and `SUBSCRIBE_2` blocks for OPC-UA communication.
- Four `E_D_FF` flip-flops provide edge-triggered buffering, preventing data loss when events occur faster than the communication cycle.
- Initialization is chained: `INIT` ⟶ `READ_SUBSCRIBE.INIT` ⟶ `WRITE_CLIENT.INIT` ⟶ `INITO`.
- The output `QO` is only `TRUE` when both write and read sides are operational, giving a combined health indicator.
- Status strings are propagated separately for write and read operations, aiding diagnostics.

State Overview

The FB does not maintain a complex internal state machine. Its behaviour is driven by the internal FB instances:

- **`READ_SUBSCRIBE`** operates in the standard subscribe-mode, generating `IND` events whenever subscribed data changes.
- **`WRITE_CLIENT`** operates in client mode, generating `REQ` events to send data and `CNF` events to confirm completion.
- The `E_D_FF` flip-flops act as single-bit storage elements, toggling their `Q` outputs on each clock edge from the corresponding event inputs.
- The `AND_QO` block simply performs a logical AND of the two `QO` signals.

Overall, the FB is in an active state only when `QI = TRUE`. When `QI` is `FALSE`, all internal blocks are disabled and outputs become `FALSE`/empty strings.

Application Scenarios

This FB is suitable for applications requiring:

- Reliable bidirectional exchange of two BOOL values between an industrial controller and an OPC-UA server.
- Separation of write and read data paths with individual status reporting.
- Edge-triggered buffering to handle high-frequency updates without loss.
- Use cases where an A2X2 adapter is present, such as connecting to remote I/O, interlocking logic, or display of single-bit state information.

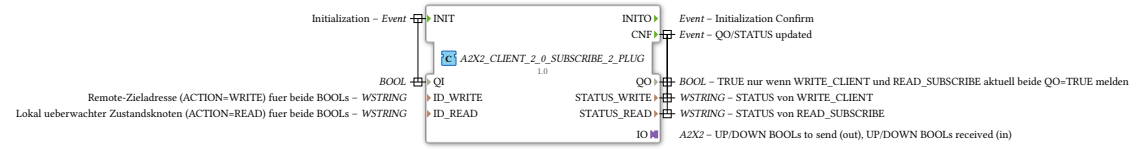
Comparison with Similar Blocks

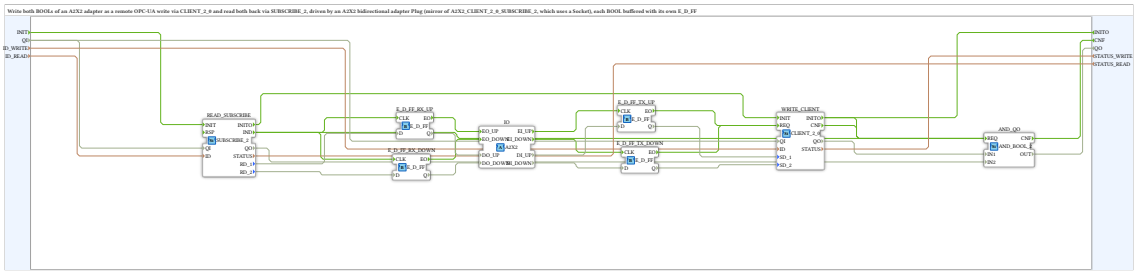
- **A2X2_CLIENT_2_0_SUBSCRIBE_1** (hypothetical) might handle only one BOOL channel; this FB processes two channels (UP and DOWN) with independent buffering.
- Using two separate `CLIENT_2_0` and `SUBSCRIBE_2` blocks for each channel would double the network connections and complicate status aggregation; this FB encapsulates both in one unit.
- Compared to a simple direct coupling without flip-flops, the `E_D_FF` buffers prevent race conditions and ensure that only stable, latched values are transmitted or displayed.

Conclusion

A2X2_CLIENT_2_0_SUBSCRIBE_2 provides a compact, reliable solution for bidirectional BOOL exchange with an OPC-UA server. It combines standard communication blocks with local buffering, yielding clear status outputs and a robust initialization sequence. Its interface is tailored to the A2X2 adapter, making it a ready-to-use component in distributed automation systems.

A2X2_CLIENT_2_0_SUBSCRIBE_2_PLUG





• – ○ • – ○ • – ○ •

Introduction

The `A2X2_CLIENT_2_0_SUBSCRIBE_2_PLUG` is a composite function block that bridges a bidirectional A2X2 adapter plug with an OPC-UA server. It writes two Boolean values (UP and DOWN) to a remote node via the `CLIENT_2_0` write service and continuously reads two Boolean values from another remote node via the `SUBSCRIBE_2` subscription service. Each Boolean direction is individually buffered using its own `E_D_FF` flip-flop, ensuring reliable edge-triggered data transfer between the adapter interface and the network services.

This block is the plug-side mirror of `A2X2_CLIENT_2_0_SUBSCRIBE_2`, which uses a socket. It is designed for applications where an OPC-UA server must be remotely controlled and monitored through a bidirectional adapter interface, such as in industrial automation and IoT gateways.

Interface Structure

Event Inputs

Event	Type	Comment	With
`INIT`	`EInit`	Initialization	`QI`

Event Outputs

Event	Type	Comment	With
`INITO`	`EInit`	Initialization Confirm	–
`CNF`	`Event`	`QO` / `STATUS` updated	`QO`, `STATUS_WRITE`, `STATUS_READ`

Data Inputs

Name	Type	Comment
`QI`	`BOOL`	Enable the block. `TRUE` starts the client and subscribe services.
`ID_WRITE`	`WSTRING`	Remote target address (ACTION=WRITE) for both Boolean values.
`ID_READ`	`WSTRING`	Locally monitored state node (ACTION=READ) for both Boolean values.

Data Outputs

Name	Type	Comment
QO	BOOL	TRUE only if both WRITE_CLIENT.QO and READ_SUBSCRIBE.QO are TRUE.
STATUS_WRITE	WSTRING	Status of the WRITE_CLIENT instance.
STATUS_READ	WSTRING	Status of the READ_SUBSCRIBE instance.

Adapters

Name	Type	Comment
IO	adapter::types::bidirectional::A2X2	UP/DOWN BOOLs to send (out), UP/DOWN BOOLs received (in).

Functionality

The block operates as a bidirectional data bridge between the A2X2 adapter plug and an OPC-UA server. Two independent paths handle the write and read directions:

Write Path (PLC → OPC-UA):

The adapter plug provides two input Booleans, DI_UP and DI_DOWN. When the plug raises the corresponding event (EI_UP or EI_DOWN), the associated E_D_FF latch captures the current value on its D input and triggers its output event (EO). This event activates WRITE_CLIENT.REQ, causing the client to send the latched value(s) (SD_1 for UP, SD_2 for DOWN) to the remote OPC-UA node specified by ID_WRITE. The client's CNF event indicates completion or error.

Read Path (OPC-UA → PLC):

The SUBSCRIBE_2 FB continuously listens to the remote node given by ID_READ. When new data arrives (event IND), the two received Booleans (RD_1 and RD_2) are latched into the corresponding E_D_FF flip-flops. The outputs of these flip-flops (Q) are placed on DO_UP and DO_DOWN, and the respective output events (EO_UP, EO_DOWN) are triggered on the adapter plug, notifying the consumer that new values are available.

Initialization and Monitoring:

- The INIT event starts the READ_SUBSCRIBE first; after its INITO, the WRITE_CLIENT is initialized. The overall INITO is emitted when both are ready.
- The AND_QO logic combines the QO outputs of both network blocks. QO is TRUE only when both the client and the subscriber report operational health. The CNF event is triggered by the AND_QO block whenever any of the two blocks updates its status. This allows external logic to monitor the combined service status.

Technical Features

- **Composite type** – Built from standard 4diac FBs (CLIENT_2_0, SUBSCRIBE_2, E_D_FF, AND_BOOL_2).
- **Edge-triggered buffering** – Each Boolean direction uses a dedicated E_D_FF to reliably capture data changes on the adapter side and on the subscription side.
- **Independent addressing** – The write and read targets are configured separately via ID_WRITE and ID_READ, allowing different OPC-UA nodes for command and feedback.
- **Combined status** – The QO output reflects the overall health of both communication channels.
- **Event-driven** – No continuous polling; all transfers are event-based, minimizing network traffic.
- **Initialization sequence** – Ensures subscribe becomes active before client writes are attempted.

State Overview

There is no explicit state machine; the block's behaviour is event-driven. However, the following operational states can be inferred:

- **Idle** – QI = FALSE or initialization not completed. No network activity.
- **Initializing** – After INIT, the subscribe service is being started, followed by the client. QO is FALSE.
- **Active** – Both WRITE_CLIENT and READ_SUBSCRIBE have QO = TRUE. The block is writing and reading data as events arrive.
- **Degraded / Error** – Either the client or subscriber reports a failed QO (FALSE). The combined QO becomes FALSE, and the corresponding STATUS output contains diagnostic information.

Application Scenarios

- **Remote I/O control** – Controlling two output bits on a remote PLC or device via OPC-UA while simultaneously monitoring two input bits.
- **HMI / SCADA integration** – Bridging an A2X2 adapter to an OPC-UA server for supervisory control and data acquisition.
- **Gateway between protocols** – Using the adapter to interface with proprietary fieldbus systems while exposing the same signals through OPC-UA.
- **Redundant monitoring** – The combined status (`QO`) can be used to trigger alarms if either the write or read channel fails.

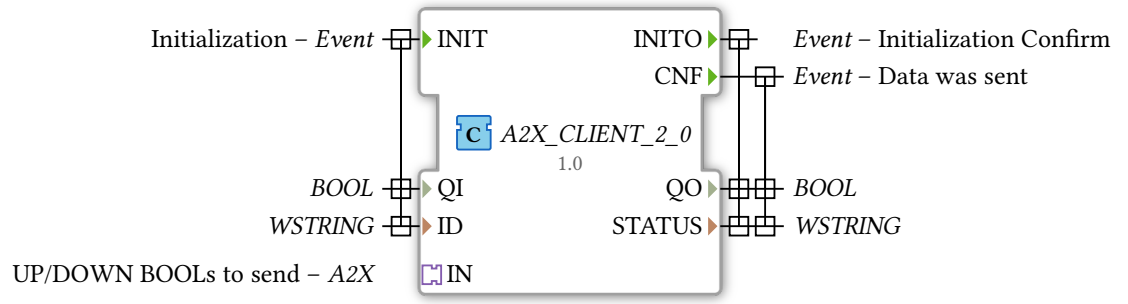
Comparison with Similar Blocks

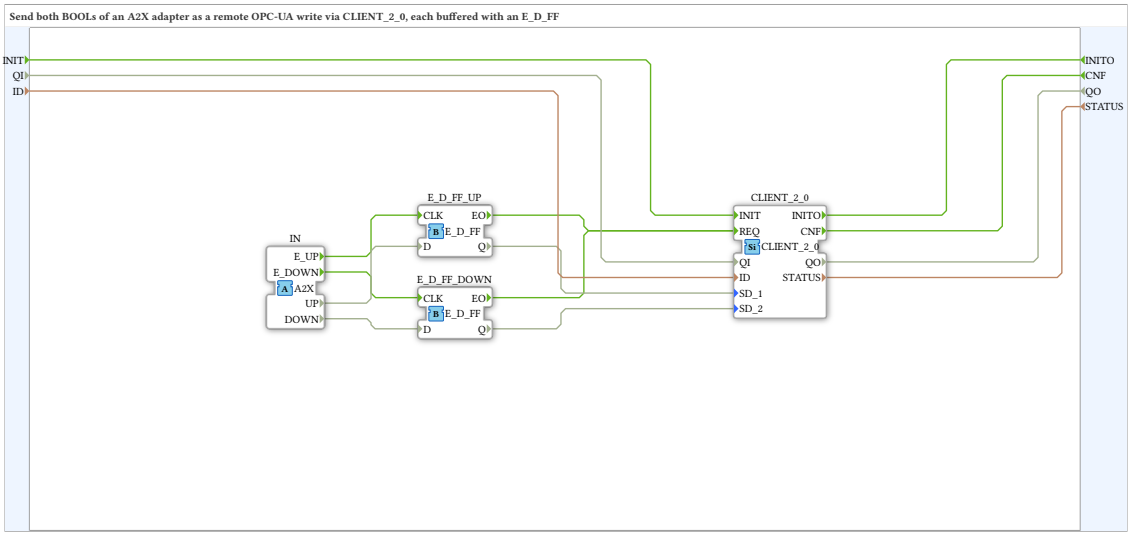
- **`A2X2_CLIENT_2_0_SUBSCRIBE_2` (Socket version)** – Operates identically but uses an A2X2 socket instead of a plug. The choice depends on the direction of data flow in the surrounding network; this block is intended for use in a system that provides the adapter (plug side).
- **Simple write/read blocks** – Standalone `CLIENT_2_0` and `SUBSCRIBE_2` blocks require manual management of events and data. This composite block encapsulates the necessary buffering and event linking, simplifying integration.
- **Alternative adapter types** – Blocks for other adapter types (e.g., single Boolean or analog) would have different interfaces but follow a similar pattern.

Conclusion

The `A2X2_CLIENT_2_0_SUBSCRIBE_2_PLUG` provides a clean, reusable solution for bridging two Boolean values from an adapter plug to an OPC-UA server and back. Its event-driven design, combined with flip-flop buffering and a unified status output, makes it well suited for real-time industrial applications where reliable and responsive data exchange is required. The block is fully self-contained and can be integrated directly into any 4diac application.

A2X_CLIENT_2_0





• – ○ • – ○ • – ○ •

Introduction

A2X_CLIENT_2_0 is a composite function block that takes two Boolean values (UP and DOWN) from an A2X adapter socket and performs a remote OPC-UA write via an embedded CLIENT_2_0 client. Each incoming value change is captured by its own edge-triggered D flip-flop (E_D_FF) before the write request is issued. This buffering ensures that only the most recent state changes are sent to the remote server, while the rest of the network remains decoupled from the exact timing of the write operation.

Interface Structure

Event Inputs

Name	Type	Comment	With Variables
INIT	EInit	Initialization	QI, ID

Event Outputs

Name	Type	Comment	With Variables
INITO	EInit	Initialization Confirm	QO, STATUS
CNF	Event	Data was sent	QO, STATUS

Data Inputs

Name	Type	Comment
QI	BOOL	Initialization qualifier, passed directly to the embedded CLIENT_2_0
ID	WSTRING	Connection identifier / endpoint address for the OPC-UA server

Data Outputs

Name	Type	Comment
QO	BOOL	Quality / state output of the embedded CLIENT_2_0
STATUS	WSTRING	Status message, holding the last operation result

Adapters

Name	Type	Direction	Comment
IN	adapter::types::unidirectional::A2X	Socket	Provides UP/DOWN Boolean values and their change events (E_UP / E_DOWN)

Functionality

The block is initialized via the INIT event. On INIT, the QI and ID input values are forwarded to the internal CLIENT_2_0, and the client establishes the connection to the configured OPC-UA server. The INITO event confirms successful completion and reports the resulting QO and STATUS.

At runtime, the adapter socket IN delivers the two Boolean values UP and DOWN. Each value is connected to the data input D of its own edge-triggered D flip-flop:

- IN.UP ⟶ E_D_FF_UP.D
- IN.DOWN ⟶ E_D_FF_DOWN.D

The corresponding adapter events (IN.E_UP and IN.E_DOWN) act as clock signals (CLK) for the respective flip-flops. When either event occurs, the current value on its data input is latched into the flip-flop output (Q). At the same time, the flip-flop's event output (EO) triggers a REQ event on the embedded CLIENT_2_0.

The latched values are connected to the client's data inputs:

- E_D_FF_UP.Q ⟶ CLIENT_2_0.SD_1
- E_D_FF_DOWN.Q ⟶ CLIENT_2_0.SD_2

All other values (the unchanged one) remain at their previous latched state. The REQ event therefore causes a single OPC-UA write operation, transmitting both SD_1 and SD_2 to the remote server. Upon completion, the client outputs the CNF event together with the current QO and STATUS values.

Technical Features

- **Edge-triggered buffering**: Two independent E_D_FF instances capture UP and DOWN changes, guaranteeing that only actual state transitions cause a write operation.
- **Shared write channel**: Both flip-flop outputs lead to a single CLIENT_2_0 REQ input, ensuring that one event produces exactly one write request.
- **Direct initialization pass-through**: QI and ID are forwarded unchanged to the CLIENT_2_0, keeping the configuration interface minimal.
- **Combined result reporting**: Both INITO and CNF events deliver the same QO and STATUS data, giving a consistent view of the connection and operation state.

State Overview

The block behaves as follows:

- **Uninitialized**: Before INIT, the internal CLIENT_2_0 is inactive. No events are processed and no writes occur.
- **Initializing**: After INIT arrives, the client is configured with QI and ID. Successful completion is signalled by INITO; the client is then ready for REQ.
- **Idle / Waiting**: Between REQ events, the flip-flops wait for new adapter events (E_UP or E_DOWN). Their outputs hold the last written values.
- **Write in progress**: When E_UP or E_DOWN fires, the corresponding flip-flop updates its output and triggers REQ. The write operation is performed by the client; its completion is signalled by CNF.

Application Scenarios

- **Remote up/down control**: Sending two binary command states (e.g., raise/lower) from a local adapter to an OPC-UA server, where each state change must be individually acknowledged.
- **Binary state mirroring**: Continuously synchronizing two Boolean process variables to a remote controller, while avoiding continuous polling or redundant writes.
- **Distributed automation**: In a system where the A2X adapter provides field-level signals, this block forwards them to a central OPC-UA data hub, independent of the client's polling cycles.

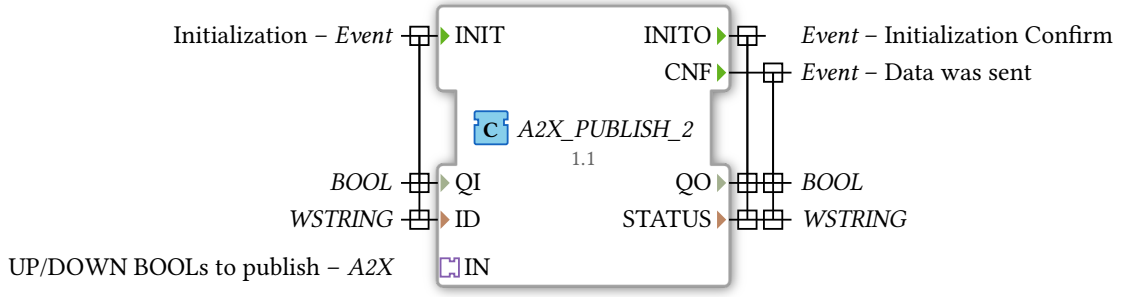
Comparison with Similar Blocks

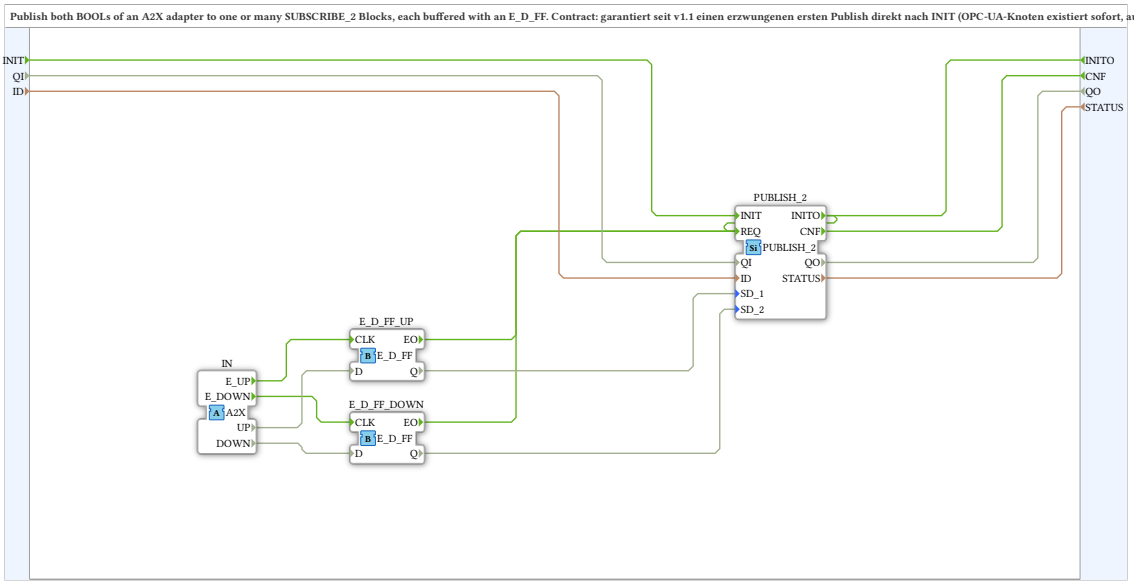
- **Without buffering (direct connection)**: A straightforward CLIENT_2_0 with direct adapter connections would only write when a client-internal trigger occurs; it would not automatically react to every value change. A2X_CLIENT_2_0 ensures a write is issued for every transition.
- **Single E_D_FF variant**: A version using only one flip-flop would concatenate both values into a single data point, losing independence of the two signals. Here, each signal has its own flip-flop, preserving individual change events.
- **Manual REQ triggering**: Some designs require an external block to generate REQ every time new data arrives. In A2X_CLIENT_2_0, this trigger is generated internally by the flip-flops, reducing the external wiring and keeping the interface compact.

Conclusion

A2X_CLIENT_2_0 provides a clean, self-contained solution for forwarding two Boolean signals from an A2X adapter to a remote OPC-UA server. By integrating two edge-triggered flip-flops with a standard OPC-UA client, it combines reliable change detection with efficient write operations. Its simple interface —only an INIT event, an adapter socket, and the standard QI/ID/QO/STATUS data—makes it easy to integrate into larger automation networks while ensuring that each state change is transmitted exactly once.

A2X_PUBLISH_2





• – ○ • – ○ • – ○ •

Introduction

The **A2X_PUBLISH_2** function block is a network publishing component that takes the two Boolean values (`UP` and `DOWN`) from an A2X adapter socket and publishes them via a standard `PUBLISH_2` block to one or more `SUBSCRIBE_2` counterparts. Each published value is buffered internally with an edge-triggered `E_D_FF` flip-flop, ensuring that only state changes trigger a network transmission, reducing unnecessary network traffic. The block is designed for scenarios where a single publisher must distribute a pair of Boolean signals (e.g., up/down commands) to multiple subscribers over Ethernet.

Interface Structure

Event Inputs

Event	Type	With Variables	Comment
`INIT`	`EInit`	`QI`, `ID`	Initialization of the publish connection.

Event Outputs

Event	Type	With Variables	Comment
`INITO`	`EInit`	`QO`, `STATUS`	Initialization confirmation.
`CNF`	`Event`	`QO`, `STATUS`	Confirmation that data was sent.

Data Inputs

Variable	Type	Comment

QI	BOOL	Quality / enable input. When TRUE, the block is active.
ID	WSTRING	Remote endpoint identifier (e.g., IP address and port) for the publish channel.

Data Outputs

Variable	Type	Comment
QO	BOOL	Quality / confirmed output. Reflects the active state after initialization.
STATUS	WSTRING	Status information of the last operation (e.g., error or success message).

Adapters

Direction	Name	Type	Comment
Socket	IN	adapter::types::unidirectional::A2X	Provides the two BOOL values UP and DOWN to be published.

Functionality

The A2X_PUBLISH_2 block combines an A2X adapter with an internal PUBLISH_2 network block and two edge-triggered D-flip-flops (E_D_FF_UP and E_D_FF_DOWN).

1. **Data capture**: The Boolean values IN.UP and IN.DOWN are continuously wired to the D inputs of two flip-flops.
2. **Edge detection**: When an event IN.E_UP occurs, the flip-flop E_D_FF_UP latches the current UP value onto its output Q. The same applies to E_D_FF_DOWN when IN.E_DOWN occurs.
3. **Trigger publishing**: On every rising edge at either flip-flop (the EO output), the internal PUBLISH_2 block receives a REQ event, capturing the latched values into its SD_1 and SD_2 data inputs.
4. **Network transmission**: The PUBLISH_2 block sends the two values to the configured endpoint (ID). When successful, it emits the CNF event and updates QO and STATUS.
5. **Initialization**: The INIT event initializes the underlying PUBLISH_2 connection. The result is reported via INITO, QO, and STATUS.

The internal buffering ensures that repeated identical values do not trigger unnecessary transmissions. Only actual changes of the UP or DOWN signals cause the publish operation.

Technical Features

- **Edge-triggered buffering**: Both input signals are latched in E_D_FF elements, ensuring that the last state is retained until a new edge appears.
- **Single publish channel**: The two buffered values are transmitted together in one PUBLISH_2 operation, minimizing network overhead.
- **Standard 61499-compliant**: Built from standard iec61499 function blocks (E_D_FF, PUBLISH_2).
- **Unidirectional adapter**: Designed for one-way communication from the publisher to subscribers.
- **Status feedback**: QO and STATUS provide clear reporting of initialization and transmission results.

State Overview

The block behaves as follows with respect to its internal flip-flops:

State	Condition	Behavior
Initial	After INIT, no event received	Flip-flops hold their default (FALSE). No publish trigger occurs.
Idle	No edge on E_UP or E_DOWN	Flip-flop outputs unchanged; no publication triggered.
Up-edge	IN.E_UP rises	UP value latched; E_D_FF_UP.EO triggers PUBLISH_2.REQ.
Down-edge	IN.E_DOWN rises	DOWN value latched; E_D_FF_DOWN.EO triggers PUBLISH_2.REQ.
Publishing	REQ emitted	PUBLISH_2 sends SD_1 and SD_2 to the network.

Confirmed	CNF emitted	Outputs QO and STATUS are updated; the block returns to idle.

The flip-flops retain their last value indefinitely until the next relevant edge, meaning the last published state is always remembered.

Application Scenarios

- **Machine control**: Publishing two limit-switch states (up/down) from one controller to several HMIs or PLCs.
- **Remote I/O**: Distributing two Boolean sensor values to all subscribers in a multicast group.
- **Building automation**: Broadcasting window open/close or door up/down commands to multiple control panels.
- **Rapid status distribution**: Any application where a single source must notify several consumers of a Boolean pair, with network efficiency achieved by transmitting only on state changes.

Comparison with Similar Blocks

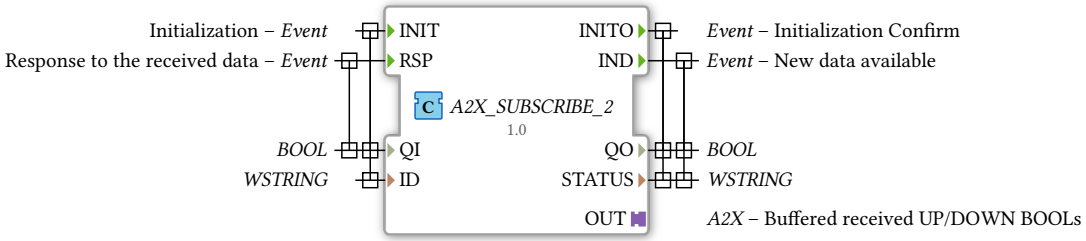
Block	Key Difference
A2X_PUBLISH_2	Buffers adapter BOOLs in E_D_FF and publishes only on change.
Plain PUBLISH_2	Publishes directly from data inputs with no internal edge detection; every REQ sends data.
A2X_SUBSCRIBE_2	Receives instead of sends, no buffering of edge events.
A2X_PUBLISH_2 with manual buffering	Requires external E_D_FF instances; this block encapsulates them for convenience.

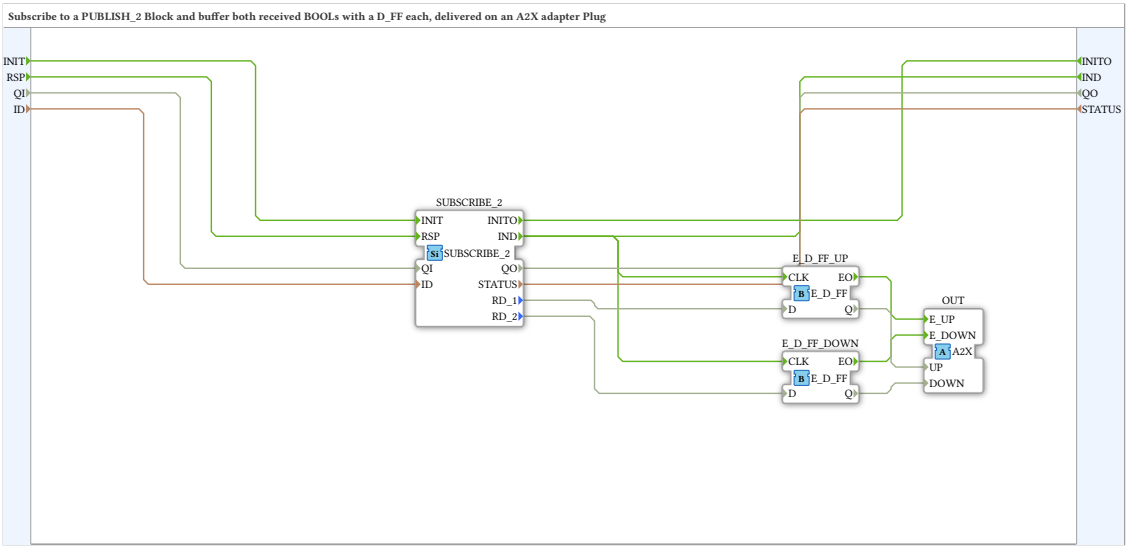
The main advantage over a direct PUBLISH_2 usage is the automatic edge-triggered buffering, which reduces network load and simplifies application logic.

Conclusion

The **A2X_PUBLISH_2** function block provides a compact, efficient solution for publishing a pair of Boolean signals from an A2X adapter to multiple subscribers. By integrating edge-triggered buffers with a standard publish block, it ensures that only meaningful state changes are transmitted, reducing network traffic and simplifying the application design. It is well suited for distributed automation systems where reliable and efficient status dissemination is required.

A2X_SUBSCRIBE_2





• – ○ • – ○ • – ○ •

Introduction

The **A2X_SUBSCRIBE_2** function block is a composite network subscriber designed to receive two Boolean values from a remote **PUBLISH_2** block via an Ethernet-based publish/subscribe mechanism. It internally buffers the two received BOOL values using dedicated D flip-flops (E_D_FF) and delivers them — together with corresponding event signals — through a unidirectional A2X adapter plug for further processing in the application logic.

This block is particularly useful in distributed IEC 61499 systems where two independent Boolean state variables (e.g., UP and DOWN commands) must be reliably received, synchronized, and made available to a local consumer in a structured and event-driven manner.

Interface Structure

Event Inputs

Event Name	Type	Associated Data	Description
`INIT`	EInit	`QI`, `ID`	Initialization event that configures the underlying subscription with the quality indicator and the communication identifier.
`RSP`	Event	`QI`	Response event used to acknowledge or re-trigger the subscription process after a previous request.

Event Outputs

Event Name	Type	Associated Data	Description
`INITO`	EInit	`QO`, `STATUS`	Initialization confirmation, indicating whether the subscription was successfully established.
`IND`	Event	`QO`, `STATUS`	Indicates that new data has been received from the publisher and is available at the adapter outputs.

Data Inputs

Data Name	Type	Description
`QI`	BOOL	Quality indicator enabling (`TRUE`) or disabling (`FALSE`) the subscription operation.

ID	WSTRING	Communication identifier (e.g., multicast address or group ID) used to bind the subscriber to the correct publisher.

Data Outputs

Data Name	Type	Description
QO	BOOL	Quality output confirming that the subscription is active and operational.
STATUS	WSTRING	Status information string providing runtime diagnostics or error messages.

Adapters

Adapter Name	Direction	Type	Description
OUT	Plug	adapter::types::unidirectional::A2X	Buffered output adapter delivering the two received Boolean values (UP, DOWN) together with their associated event indicators (E_UP, E_DOWN).

Functionality

The **A2X_SUBSCRIBE_2** block operates as follows:

1. **Initialization**: Upon receiving the INIT event, the internal SUBSCRIBE_2 network subscriber is configured using the QI enable flag and the ID communication identifier. The INITO event confirms successful initialization with updated QO and STATUS values.
2. **Data Reception**: When the underlying SUBSCRIBE_2 network service receives data from a matching publisher, it raises the IND event. This event triggers three parallel actions:
   - The newly received RD_1 value is latched into the E_D_FF_UP D flip-flop.
   - The newly received RD_2 value is latched into the E_D_FF_DOWN D flip-flop.
   - The IND event is propagated to the block's external event output to signal that fresh data is available.
3. **Buffered Output**: The latched values from both flip-flops are continuously driven onto the adapter outputs (OUT.UP and OUT.DOWN). Correspondingly, the flip-flop event outputs (EO) are connected to OUT.E_UP and OUT.E_DOWN, providing event-driven notifications for each data change.
4. **Response Handling**: The RSP event input allows the application to explicitly trigger a new subscription request or acknowledgement cycle through the internal subscriber, with the QI value supplied for re-evaluation.

Technical Features

- **Composite implementation**: Built as a network of three internal function blocks (SUBSCRIBE_2, E_D_FF_UP, E_D_FF_DOWN), demonstrating reusable modular design.
- **Ethernet-based communication**: Uses the standard IEC 61499 network SUBSCRIBE_2 service type for reliable remote data exchange.
- **Data buffering**: Each incoming Boolean value is latched into an edge-triggered D flip-flop, ensuring stable output values between publication cycles.
- **Event-driven architecture**: Both the external IND event and the internal adapter events (E_UP, E_DOWN) are generated based on the reception trigger, allowing downstream logic to react immediately.
- **Standardized status reporting**: QO and STATUS provide consistent quality and diagnostic information for monitoring and error handling.
- **Unidirectional data flow**: The A2X adapter plug strictly follows a publisher-to-subscriber direction, simplifying data flow analysis.

State Overview

The block does not maintain an explicit internal state machine in the composite network, but its behaviour can be described in terms of operational phases:

Phase	Description	Trigger

Idle	Subscription is inactive; `QO = FALSE`.	Before `INIT` or after failed initialization
Initializing	`INIT` received; internal subscriber is being configured.	`INIT` event edge
Active / Subscribed	Subscription established; `QO = TRUE`; waiting for data.	Successful `INITO` with valid status
Data Received	New publication arrived; flip-flop latches are updated; `IND` raised.	Internal `SUBSCRIBE_2.IND`
Re-Trigger	Application requested a new subscription cycle via `RSP`.	`RSP` event edge

Application Scenarios

Typical use cases for **A2X_SUBSCRIBE_2** include:

- **Distributed control systems**: Receiving two independent Boolean control commands (e.g., UP/DOWN, START/STOP) from a central controller over Ethernet and forwarding them to local actuator logic.
- **Remote status monitoring**: Acquiring two-status-bit telemetry (e.g., valve open/closed, sensor alarm/normal) from a remote field device and presenting them to a local HMI or processing unit.
- **Command buffering in safety-related applications**: Latching incoming commands to guarantee stable output levels even if network interruptions occur between publication cycles.
- **Multi-node coordination**: Acting as a reception front-end in a star or multicast network where one publisher serves multiple subscribers.

Comparison with Similar Blocks

Feature	A2X_SUBSCRIBE_2	Standard SUBSCRIBE_2	SUBSCRIBE_1
Data capacity	2 BOOL values	2 BOOL values (`RD_1`, `RD_2`)	1 value (any type)
Output format	Buffered via D flip-flops, delivered through A2X adapter	Raw output variables	Raw output variable
Event granularity	Separate events per buffered channel (`E_UP`, `E_DOWN`) plus common `IND`	Single `IND` event	Single `IND` event
Integration convenience	Direct plug connection to A2X adapter sockets	Requires manual connection of data/events	Requires manual connection
Data stability	Continues to hold last values after reception	Outputs follow last received values (no explicit latching)	Outputs follow last received values
Use case fit	Application-oriented ready-to-connect subscriber	Low-level network service	Generic single-value network reception

Compared to the raw `SUBSCRIBE_2` network FB, **A2X_SUBSCRIBE_2** adds buffering, structured event distribution, and an adapter-based interface that simplifies integration into higher-level application designs.

Conclusion

The **A2X_SUBSCRIBE_2** function block provides a clean, ready-to-use solution for receiving two Boolean values over an Ethernet-based publish/subscribe network and delivering them to the application layer through a standardized A2X adapter interface. By combining network subscription with D flip-

flop buffering and dedicated per-channel events, it significantly reduces the design effort for distributed control applications while enhancing data stability and clarity. Its modular composite implementation follows best practices of IEC 61499, making it a valuable building block in industrial automation and decentralized control scenarios.

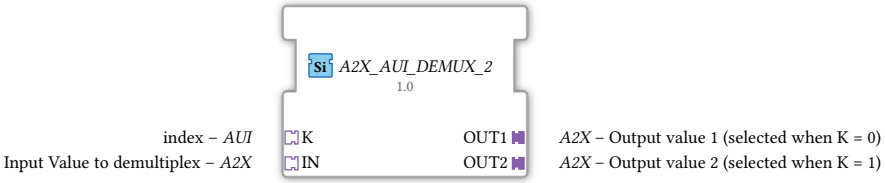
2.8.0.3 selection

unidirectional

Numerical Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace in ascending order.

A2X_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

A2X_AUI_DEMUX_2 is a generic, unidirectional demultiplexer function block for 4diac-ide. It receives one value through an A2X adapter socket and forwards it to one of two A2X output adapters. The selected output is controlled by an index value received through the AUI adapter socket K.

The block is designed as a generic FB: the actual behavior is provided by the generic implementation class `GEN_A2X_AUI_DEMUX`. The adapter outputs are only updated when the routed value actually changes.

Interface Structure

The interface is adapter-centric. There are no direct data inputs or data outputs. All values are transported via the unidirectional adapter sockets and plugs.

Event Inputs

There are no top-level event inputs. The FB is activated through data changes arriving on its adapter sockets.

Event Outputs

No event outputs.

Data Inputs

There are no direct data inputs. The input value is supplied through the `IN` adapter socket.

Data Outputs

There are no direct data outputs. The demultiplexed values are provided through the `OUT1` and `OUT2` adapter plugs.

Adapters

Adapter	Direction	Type	Description
`OUT1`	Plug / output	`A2X`	Output value 1, selected when K = 0
`OUT2`	Plug / output	`A2X`	Output value 2, selected when K = 1
`K`	Socket / input	`AUI`	Index input used to select the active output
`IN`	Socket / input	`A2X`	Input value to be demultiplexed

Functionality

The FB behaves as a 1-to-2 demultiplexer:

1. It receives an index value on the K socket.
2. If K = 0, the value from IN is routed to OUT1.
3. If K = 1, the value from IN is routed to OUT2.
4. The selected output adapter is updated only when the new value differs from the previously output value.
5. If the value is unchanged, no output update and no event take place.

This behavior reduces unnecessary communication between adapters and prevents downstream function blocks from reacting to unchanged values.

Technical Features

- Generic FB with generic class name GEN_A2X_AUI_DEMUX.
- Package: adapter::selection::unidirectional.
- Fully adapter-based interface without direct data inputs or outputs.
- Unidirectional A2X and AUI adapter types.
- Two output adapters allow selecting between two destinations.
- Event generation is suppressed when the output value has not changed.
- Suitable for use inside modular IEC 61499 applications with adapter-based communication.

State Overview

The XML type definition does not contain an explicit ECC state machine. The behavior is implemented by the generic backend. Conceptually, the FB can be viewed in the following states:

State	Description
WAITING	The FB waits for a new index value on K or a new input value on IN.
ROUTING	The current input value is routed to the output selected by K.
UPDATING	The selected output adapter is updated because the value has changed.
IDLE	The value is unchanged, so no adapter update and no event occur.

Application Scenarios

- **Selection between two values:** A single A2X value source can be forwarded to one of two different consumers based on a selector index.
- **Mode-dependent routing:** In an automation application, the same process value can be sent to different control or monitoring modules depending on the operational mode.
- **Adapter-based signal distribution:** The FB can be used as a compact demultiplexing stage in a modular IEC 61499 system where communication uses unidirectional A2X and AUI adapters.
- **Event reduction:** When values are frequently repeated, the change-only behavior avoids unnecessary event processing in the downstream logic.

Comparison with Similar Blocks

Block Type	Purpose	Difference
A2X_AUI_DEMUX_2	Demultiplexes one A2X input to two A2X outputs using an AUI index.	Adapter-based, generic, event-on-change only.
Typical multiplexer	Combines multiple inputs into one output.	Opposite data flow direction.
AX_AUI_DEMUX_2	Similar demultiplexing function with an alternative adapter type.	This block is the A2X variant and uses A2X for the value channel.

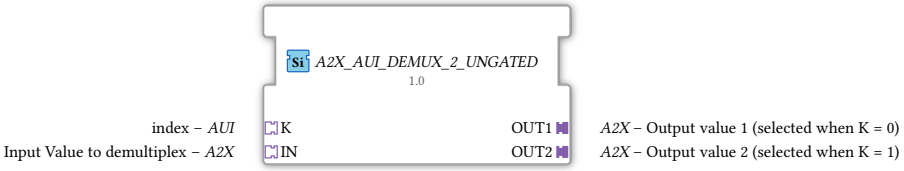
The main distinguishing feature of A2X_AUI_DEMUX_2 compared with non-adapter demultiplexers is its clean adapter-based interface and its ability to suppress events when no value change occurs.

Conclusion

A2X_AUI_DEMUX_2 is a flexible, generic demultiplexer function block for adapter-based 4diac applications. It routes one A2X input value to one of two outputs using an AUI index, while avoiding unnecessary

update events. Its compact adapter interface makes it well suited for modular, event-driven control systems.

A2X_AUI_DEMUX_2_UNGATED



• – ◦ • – ◦ • – ◦ •

Introduction

A2X_AUI_DEMUX_2_UNGATED is an adapter-based generic demultiplexer function block. It routes an incoming value received through the `IN` adapter to one of two output adapters, `OUT1` or `OUT2`, depending on the index received through the `K` adapter.

The "UNGATED" variant does not perform any change detection. Every newly computed value or result is forwarded unconditionally. This makes the block suitable for consumers that require a periodic data cadence, independent of whether the actual payload value has changed, for example in derivative or frequency calculations.

Interface Structure

The block does not use dedicated data inputs or data outputs. All value and index information is exchanged through unidirectional adapters. The block has no event outputs.

Event Inputs

None. Incoming triggers are received through the adapter sockets.

Event Outputs

No event outputs.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Socket	`K`	`adapter::types::unidirectional::AUI`	Index input used for output selection.
Socket	`IN`	`adapter::types::unidirectional::A2X`	Input value to be demultiplexed.
Plug	`OUT1`	`adapter::types::unidirectional::A2X`	Output value 1, selected when `K = 0`.
Plug	`OUT2`	`adapter::types::unidirectional::A2X`	Output value 2, selected when `K = 1`.

Functionality

The function block operates as a one-to-two demultiplexer:

1. The index adapter `K` provides the selection information.
2. The value adapter `IN` provides the value to be routed.
3. If `K = 0`, the incoming value is forwarded to `OUT1`.
4. If `K = 1`, the incoming value is forwarded to `OUT2`.

Because the block is "ungated", it does not compare the current value against the previous value. Every new calculation result or incoming value is passed through even if the payload is unchanged. This behavior is essential for consumers that rely on a steady event stream or periodic update rate.

Technical Features

- Adapter-based interface without separate data inputs or outputs.
- Unidirectional adapter types are used for both index and data transfer.
- Generic FB implementation via the attribute `eclipse4diac::core::GenericClassName` with value `GEN_A2X_AUI_DEMUX`.
- No event outputs.
- Two socket adapters: `K` and `IN`.
- Two plug adapters: `OUT1` and `OUT2`.
- No change detection or value gating logic.
- Designed for periodic, time-driven consumers.

State Overview

The XML definition does not contain an explicit ECC state machine. The behavior is handled by the generic runtime implementation behind `GEN_A2X_AUI_DEMUX`.

The relevant runtime state is limited to:

- The currently selected output path, determined by the index `K`.
- The forwarding state of the input value `IN`.

Since the block is ungated, there is no state representing “value changed” or “value unchanged”. Every incoming processing cycle leads to unconditional forwarding to the selected output.

Application Scenarios

A2X_AUI_DEMUX_2_UNGATED is useful in all cases where an incoming data stream must be distributed to one of two consumers based on an index, and where every computed result must be delivered.

Typical examples:

- Derivative or slope calculations that need to process every sample.
- Frequency or period measurement functions that require a fixed event cadence.
- Adapter-based unidirectional data distribution in a 4diac IEC 61499 application.
- Splitting a continuous value stream into two downstream processing branches.

Comparison with Similar Blocks

The closest comparable block is `A2X_AUI_DEMUX_2`. Both blocks share the same adapter interface and output selection behavior.

The key difference is:

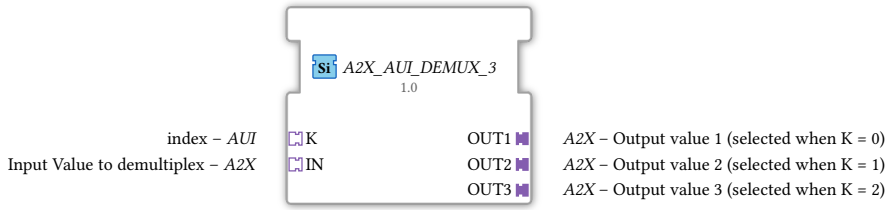
- `A2X_AUI_DEMUX_2` includes change detection and only forwards values when the result has changed.
- `A2X_AUI_DEMUX_2_UNGATED` disables this detection and forwards every newly computed result unconditionally.

This makes the ungated variant preferable for consumers that need a periodic trigger even when the data value is constant, while the gated variant is better suited for event-driven consumers that only want to react to actual value changes.

Conclusion

A2X_AUI_DEMUX_2_UNGATED is a flexible, adapter-based generic demultiplexer for unidirectional data flow. It selects between two outputs using an index adapter and forwards every incoming value without change detection. This unconditional forwarding behavior makes it especially suitable for calculation chains that require a constant update cadence, such as derivative and frequency computations.

A2X_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

A2X_AUI_DEMUX_3 is a generic A2X demultiplexer function block. It receives a value through an A2X input adapter and forwards it to one of three A2X output adapters. The active output is selected by an index value provided through an AUI adapter socket.

The block is designed to reduce unnecessary communication: the adapter output is only updated when the value actually changes, and an event is only generated for a real value change.

Interface Structure

Event Inputs

The FB does not declare dedicated event inputs on its FB interface. Event interaction is expected through the connected A2X/AUI adapters.

Event Outputs

No event outputs.

Data Inputs

None. All incoming data is transferred through the adapter sockets.

Data Outputs

None. All outgoing data is transferred through the adapter plugs.

Adapters

Sockets:

- `K` — Adapter type `adapter::types::unidirectional::AUI` — index input.
- `IN` — Adapter type `adapter::types::unidirectional::A2X` — input value to demultiplex.

Plugs:

- `OUT1` — Adapter type `adapter::types::unidirectional::A2X` — output value 1, selected when `K = 0`.
- `OUT2` — Adapter type `adapter::types::unidirectional::A2X` — output value 2, selected when `K = 1`.
- `OUT3` — Adapter type `adapter::types::unidirectional::A2X` — output value 3, selected when `K = 2`.

Functionality

A2X_AUI_DEMUX_3 acts as a 1-to-3 demultiplexer for A2X adapter values. The index received on the `K` socket selects which of the three output adapters receives the incoming value from the `IN` socket.

The block only forwards an output value if the value has actually changed compared to the last forwarded value. This reduces unnecessary output updates and event traffic.

Technical Features

- Generic FB with the generic class name `GEN_A2X_AUI_DEMUX`.
- No explicit data inputs or data outputs; all payload data is carried by A2X adapters.
- Unidirectional adapter interface.
- 1-of-3 demultiplexing.
- Change detection on the output value.
- Suitable for integration into adapter-based 4diac applications.

State Overview

The FB does not expose an internal ECC state machine in its XML definition. Conceptually, the block behaves as follows:

- **Idle state** — No new index or value has arrived. The selected output keeps its last value.
- **Value update** — A new value arrives on `IN`. If it differs from the previously forwarded value, the selected output adapter is updated and an event is generated.
- **Index change** — A new value arrives on `K`. The next value update is redirected to the newly selected output.

Application Scenarios

A2X_AUI_DEMUX_3 can be used wherever one input value must be routed to one of several possible consumers. Typical scenarios include:

- Distributing process values to different visualization or control units.
- Switching between multiple output branches based on an operating mode or index.
- Reducing communication load by suppressing unchanged values.
- Building generic adapter-based routing logic in 4diac applications.

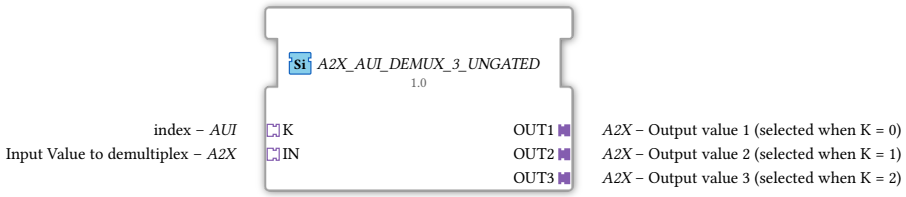
Comparison with Similar Blocks

Block / Variant	Characteristics
`A2X_AUI_DEMUX_3`	Generic A2X-based 1-to-3 demultiplexer with change detection.
`AX_AUI_DEMUX_3`	Closely related variant that does not use the A2X generic adapter backend.
Conventional demultiplexer without change detection	Forwards every input value and event unconditionally, causing higher communication load.

Conclusion

A2X_AUI_DEMUX_3 is a compact generic demultiplexer block for adapter-oriented 4diac applications. It combines selection of one of three outputs with change-based update behavior, making it suitable for efficient and selective value distribution in automation and communication systems.

A2X_AUI_DEMUX_3_UNGATED



• – ◦ • – ◦ • – ◦ •

Introduction

A2X_AUI_DEMUX_3_UNGATED is an adapter-based demultiplexer FB that distributes an incoming value stream from one input adapter to one of three output adapters. The selection is made through an index adapter. In contrast to gated demultiplexers, this FB does not perform any change detection: every newly computed result is forwarded unconditionally. It is therefore suitable for consumers that require a periodic data cadence even when the value itself has not changed, for example derivative or frequency calculations.

The FB uses the unidirectional A2X adapter type for data transport and a unidirectional AUI adapter for the index selection. It is declared as a generic FB with the generic class name `GEN_A2X_AUI_DEMUX`.

Interface Structure

Event Inputs

No event inputs are declared. The FB is driven entirely through its adapter sockets.

Event Outputs

No event outputs.

Data Inputs

No data inputs are declared.

Data Outputs

No data outputs are declared.

Adapters

Direction	Adapter	Type	Description

Socket	K	adapter::types::unidirectional::AUI	Index selector. Determines which output is used.
Socket	IN	adapter::types::unidirectional::A2X	Input value stream to be demultiplexed.
Plug	OUT1	adapter::types::unidirectional::A2X	Output value 1, selected when K = 0.
Plug	OUT2	adapter::types::unidirectional::A2X	Output value 2, selected when K = 1.
Plug	OUT3	adapter::types::unidirectional::A2X	Output value 3, selected when K = 2.

Functionality

A2X_AUI_DEMUX_3_UNGATED forwards data received on the IN adapter to exactly one of the three output adapters. The output is chosen according to the index received on the K adapter:

- K = 0 ⟶ OUT1
- K = 1 ⟶ OUT2
- K = 2 ⟶ OUT3

Each time a new value arrives on IN, the FB passes that value to the currently selected output. No comparison with a previously forwarded value is performed. This is the key difference between this block and a gated demultiplexer: the output is updated as often as new results arrive, regardless of whether the data content has changed.

Because all data and selection information is transported through adapter connections, no separate input events, output events or data inputs are required.

Technical Features

- Three unidirectional output plugs: OUT1, OUT2, OUT3.
- One unidirectional input socket IN for the data stream to be demultiplexed.
- One unidirectional index socket K for output selection.
- No change detection or gating logic.
- No internal storage of the last forwarded value.
- Generic FB implementation using the class name GEN_A2X_AUI_DEMUX.
- Intended for use in adapter-based selection and distribution scenarios.
- The adapter package is adapter::selection::unidirectional.

State Overview

The FB does not define a complex state machine. Because no change detection is required, there is no state that stores and compares previous values. The operational behavior can be summarized as:

- Awaiting index selection through K.
- Awaiting a new value on IN.
- Forwarding the received value to the output selected by K.

This simple structure makes the FB deterministic and easy to use in periodic processing chains.

Application Scenarios

A2X_AUI_DEMUX_3_UNGATED is useful wherever a data source must be routed to one of several consumers on every computation cycle. Typical scenarios include:

- Periodic derivative or frequency calculation where the consumer needs a fixed update rate even if the source value remains constant.
- Distribution of a process value to different functional modules depending on an operating mode.
- Adapter-based stream switching in automation applications.
- Demultiplexing of a single data stream into three alternative processing paths without requiring the source to detect value changes.

Because the block is un-gated, it is especially suitable for time-driven applications where the timing of the output event is more important than the value difference between two consecutive samples.

Comparison with Similar Blocks

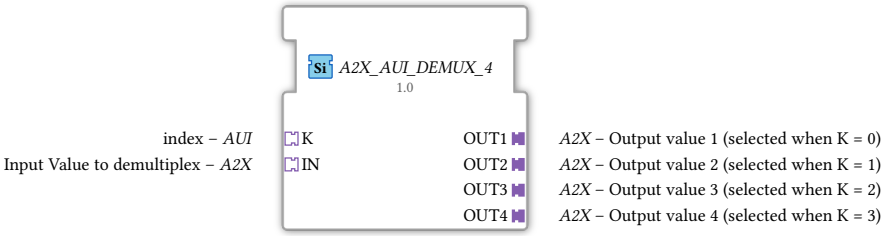
Compared with `A2X_AUI_DEMUX_3`, the `_UNGATED` variant differs in its forwarding behavior. The gated variant only forwards a value when the input value has changed. The un-gated variant forwards every newly computed value unconditionally. This makes `A2X_AUI_DEMUX_3_UNGATED` the preferred choice for consumers that need a continuous periodic stream of events.

Compared with the `AX_AUI_DEMUX_3_UNGATED` variant, this FB uses the `A2X` adapter type instead of `AX`. It is therefore suitable for systems built on the newer or different `A2X` unidirectional adapter family while preserving the same demultiplexing behavior.

Conclusion

A2X_AUI_DEMUX_3_UNGATED is a simple and flexible adapter-based demultiplexer for three output paths. Its main advantage is the unconditional forwarding of every incoming value, which guarantees a regular output cadence. The use of adapter sockets for both index selection and data input makes it well suited for modern adapter-oriented IEC 61499 applications, especially where periodic downstream processing is required.

A2X_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

A2X_AUI_DEMUX_4 is a generic IEC 61499-2 function block that implements a 1-to-4 demultiplexer for unidirectional adapter connections. It receives a value through an `IN` socket and forwards it to one of four output adapters `OUT1` … `OUT4`. The active output is selected by the index value received through the `K` socket.

The block is designed as a generic FB. Its runtime behavior is provided by the referenced generic implementation class `GEN_A2X_AUI_DEMUX`. The adapter outputs are only updated when the value actually changes, so unnecessary update events are suppressed.

Interface Structure

Event Inputs

There are no event inputs.

Event Outputs

No event outputs.

Data Inputs

There are no direct data inputs. All data is exchanged through adapter sockets and plugs.

Data Outputs

There are no direct data outputs. All data is exchanged through adapter sockets and plugs.

Adapters

Interface Role	Adapter Name	Type	Comment
Socket / Input	`K`	`adapter::types::unidirectional::AUI`	Index
Socket / Input	`IN`	`adapter::types::unidirectional::A2X`	Input Value to demultiplex

Plug / Output	OUT1	adapter::types::unidirectional::A2X	Output value 1, selected when K = 0
Plug / Output	OUT2	adapter::types::unidirectional::A2X	Output value 2, selected when K = 1
Plug / Output	OUT3	adapter::types::unidirectional::A2X	Output value 3, selected when K = 2
Plug / Output	OUT4	adapter::types::unidirectional::A2X	Output value 4, selected when K = 3

Functionality

A2X_AUI_DEMUX_4 acts as a unidirectional adapter-based demultiplexer:

1. The index K is received through the K socket.
2. The value to be routed is received through the IN socket.
3. Depending on K, one of the four output adapters OUT1 … OUT4 is selected.
4. The selected output is updated with the incoming value.

According to the FB type comment, the adapter output is only updated when the value changes. If the new value is identical to the previous value, no unnecessary output update or event is generated.

Because the FB is a generic FB, the detailed internal processing is delegated to the runtime backend GEN_A2X_AUI_DEMUX. The type definition itself mainly describes the adapter interface and the generic class binding.

Technical Features

- Generic FB implementation: GEN_A2X_AUI_DEMUX
- Package: adapter::selection::unidirectional
- 1-to-4 demultiplexing using unidirectional adapters
- Four A2X output plugs: OUT1, OUT2, OUT3, OUT4
- One AUI index socket: K
- One A2X input socket: IN
- No direct data inputs or data outputs
- No event inputs or outputs
- Output updates and events are suppressed when the value has not actually changed
- Suitable for IEC 61499-2 adapter-based applications

State Overview

The XML type definition does not contain an explicit ECC. The behavior is therefore controlled by the generic implementation. The following states describe the logical behavior of the FB:

State	Description
Idle	The FB waits for a new index or input value.
Select	The index K is read and one of the four outputs is selected.
Update	The selected output adapter is updated with the value from IN. If the value has not changed, this step is skipped.

Application Scenarios

A2X_AUI_DEMUX_4 is useful in IEC 61499 applications where a single unidirectional adapter value must be routed to one of four consumers. Typical scenarios include:

- Distributing a measured value to one of several processing blocks.
- Selecting one of four control destinations based on an index.
- Building adapter-based routing structures without using direct data connections.
- Reducing unnecessary communication traffic when the routed value changes infrequently.

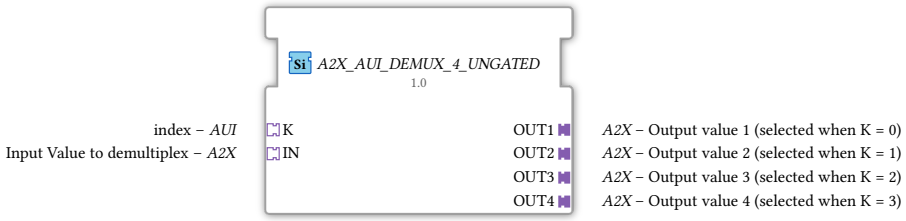
Comparison with Similar Blocks

- A standard demultiplexer typically uses direct data inputs and outputs. A2X_AUI_DEMUX_4 instead uses unidirectional adapters, which allows flexible adapter-based connections.
- It is the A2X adapter variant of the `AX_AUI_DEMUX_4` demultiplexer concept.
- Unlike a simple selector, this FB has one input and four possible output destinations.
- Unlike blocks that always forward data, this FB only updates an output when the value actually changes, reducing unnecessary events.

Conclusion

A2X_AUI_DEMUX_4 is a compact, generic, adapter-based 1-to-4 demultiplexer. It combines index-based output selection with unidirectional A2X/AUI adapter communication and change-only update behavior. This makes it suitable for efficient and flexible routing in IEC 61499-2 applications.

A2X_AUI_DEMUX_4_UNGATED



• – ◦ • – ◦ • – ◦ •

Introduction

The `A2X_AUI_DEMUX_4_UNGATED` function block is a generic one-to-four demultiplexer based on Eclipse 4diac adapters. It receives a value via the `IN` adapter and an index via the `K` adapter, and forwards the input value to one of four output adapters (`OUT1`…`OUT4`).

It is the ungated version of `A2X_AUI_DEMUX_4`: it performs no change detection and forwards every newly computed result unconditionally. This makes it suitable for downstream consumers that require a periodic data cadence independent of whether the sampled value has changed, e.g. derivative or frequency calculations.

Interface Structure

Event Inputs

None.

Event Outputs

No event outputs.

Data Inputs

None. All data is transferred through adapters.

Data Outputs

None. All data is transferred through adapters.

Adapters

Adapter	Direction	Type	Comment
`IN`	Socket	`adapter::types::unidirectional::A2X`	Input value to demultiplex.
`K`	Socket	`adapter::types::unidirectional::AUI`	Index; selects the active output.
`OUT1`	Plug	`adapter::types::unidirectional::A2X`	Output value 1, selected when `K = 0`.
`OUT2`	Plug	`adapter::types::unidirectional::A2X`	Output value 2, selected when `K = 1`.
`OUT3`	Plug	`adapter::types::unidirectional::A2X`	Output value 3, selected when `K = 2`.

OUT4	Plug	adapter::types::unidirectional::A2X	Output value 4, selected when K = 3.

Functionality

A2X_AUI_DEMUX_4_UNGATED implements a 1-to-4 demultiplexing routing logic:

1. The value to distribute is available on the IN adapter.
2. The selection index is available on the K adapter.
3. Depending on K, the value is routed to OUT1, OUT2, OUT3, or OUT4.

The term UNGATED indicates that the FB does not suppress repeated values. While a normal gated demultiplexer would compare the current value with the previous one and only forward a result when a change is detected, this variant always passes the newly calculated/received value through. Therefore, each input update triggers an output update, and the output cadence follows the input cadence.

Technical Features

- IEC 61499-2 compliant generic function block.
- Generic class name: GEN_A2X_AUI_DEMUX.
- Adapter package: adapter::selection::unidirectional.
- Four unidirectional A2X output plugs.
- Unidirectional A2X input socket for the value to demultiplex.
- Unidirectional AUI input socket for the selection index.
- No explicit data inputs or data outputs; all payload data crosses the adapter interface.
- No change-detection state or old-value storage.

State Overview

The FB type itself does not define an internal ECC state machine. Since it is ungated, it does not keep a memory of the last forwarded value and does not compare values. The only relevant selection state is the current index K. Once a new index is applied, subsequent input values are forwarded to the newly selected output. With respect to data content, the FB behaves statelessly.

Application Scenarios

- **Periodic derivative calculation:** A derivative or rate-of-change consumer needs an update in every scan/cycle, even if the measured value stayed constant. The ungated demultiplexer ensures the required cadence.
- **Frequency measurement:** Frequency calculations often rely on time stamps and event counts. Dropping unchanged values would break the calculation; unconditional forwarding prevents this.
- **Value distribution:** One sensor/calculation result can be routed to one of four consumers, selected by a controller or configuration index.
- **Streaming and telemetry:** When a periodic output stream must be maintained towards a selected sub-system, this FB can provide the required steady flow.

Comparison with Similar Blocks

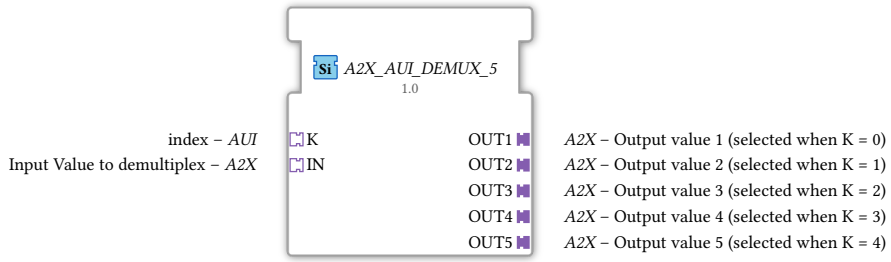
Block	Behavior
A2X_AUI_DEMUX_4	Gated demultiplexer. It performs change detection and only forwards new/changed values.
A2X_AUI_DEMUX_4_UNGATED	Ungated demultiplexer. It forwards every newly available value regardless of whether the content changed.

The main difference is therefore the presence or absence of value-change filtering. The ungated variant is preferred when downstream processing cannot miss an update cycle. The gated variant is preferred when output bandwidth should be minimized and consumers only need to react to actual value changes.

Conclusion

A2X_AUI_DEMUX_4_UNGATED is a flexible generic 1-to-4 demultiplexer built on unidirectional adapters. It combines a simple index-based output selection with an unconditional forwarding policy. By omitting change detection, it guarantees that every newly computed result reaches the selected consumer, making it particularly suitable for time-sensitive and cadence-dependent calculations such as derivatives and frequency measurements.

A2X_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`A2X_AUI_DEMUX_5` is a generic adapter-based demultiplexor. It receives a value through the unidirectional `A2X` socket `IN` and forwards it to one of five unidirectional `A2X` output plugs (`OUT1` … `OUT5`). The output selection is determined by the index value received through the `AUI` socket `K`.

The block is designed as a generic FB: the concrete type name `A2X_AUI_DEMUX_5` represents a five-output configuration of the generic class `GEN_A2X_AUI_DEMUX`. A key property is that the selected adapter output is updated only when the transferred value actually changes; no unnecessary update event is generated for unchanged values.

Interface Structure

The FB uses adapter connections instead of a direct set of data inputs and outputs. It has no direct event inputs and no direct data inputs or outputs. All input and output data is carried by the adapter sockets and plugs.

Event Inputs

The FB does not declare any direct event inputs.

Event Outputs

No event outputs.

Data Inputs

The FB does not declare any direct data inputs.

Data Outputs

The FB does not declare any direct data outputs.

Adapters

Direction	Name	Type	Description
Socket (input)	`K`	`adapter::types::unidirectional::AUI`	Selection index.
Socket (input)	`IN`	`adapter::types::unidirectional::A2X`	Input value to demultiplex.
Plug (output)	`OUT1`	`adapter::types::unidirectional::A2X`	Output selected when `K = 0`.
Plug (output)	`OUT2`	`adapter::types::unidirectional::A2X`	Output selected when `K = 1`.
Plug (output)	`OUT3`	`adapter::types::unidirectional::A2X`	Output selected when `K = 2`.
Plug (output)	`OUT4`	`adapter::types::unidirectional::A2X`	Output selected when `K = 3`.
Plug (output)	`OUT5`	`adapter::types::unidirectional::A2X`	Output selected when `K = 4`.

Functionality

The functional behavior can be summarized as:

1. The FB observes the index adapter `K`.
2. Based on the current index, it activates one of the output plugs:

- `K = 0` selects `OUT1`
- `K = 1` selects `OUT2`
- `K = 2` selects `OUT3`
- `K = 3` selects `OUT4`
- `K = 4` selects `OUT5`

3. The value received on the `IN` socket is forwarded to the selected output plug.
4. The output adapter is updated only when the value has actually changed. This avoids unnecessary adapter update events.

The valid index range is therefore `0` to `4`. Behavior for indices outside this range depends on the concrete generic runtime implementation.

Technical Features

- Generic FB class: `GEN_A2X_AUI_DEMUX`.
- Five unidirectional output adapters of type `A2X`.
- Two unidirectional input adapters: `K` (`AUI`) and `IN` (`A2X`).
- Adapter-based interface: no direct data inputs or outputs.
- Value-change-triggered output update.
- Declared as an IEC 61499-2 FB type.
- Compiler package: `adapter::selection::unidirectional`.

State Overview

The FB type does not contain an explicit ECC state machine. The behavior is therefore handled by the generic runtime implementation. Logically, the following phases can be distinguished:

Phase	Description
Wait	The FB waits for a new value on `IN` or a new index on `K`.
Select	The current value of `K` determines the active output plug.
Forward	The current `IN` value is written to the selected output if it changed.

These phases are not necessarily separate ECC states; they describe the observable behavior of the demultiplexor.

Application Scenarios

- Routing a single data source to one of five consumers based on an index.
- Dynamic selection of a target station or submodule in an adapter-based automation network.
- Reduction of event load by forwarding only actual value changes.
- Reuse in unidirectional data-flow architectures without direct data wiring.

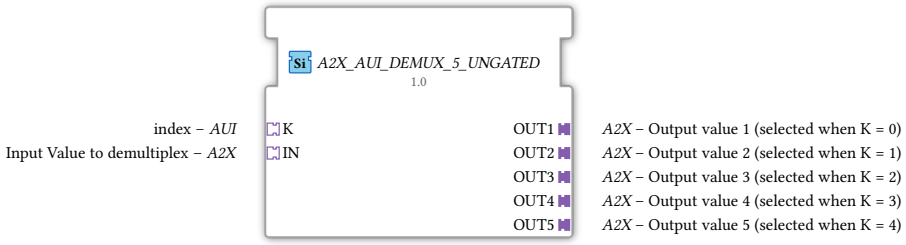
Comparison with Similar Blocks

- Compared with a conventional demultiplexer that uses direct data inputs and outputs, `A2X_AUI_DEMUX_5` uses adapter sockets and plugs. This makes it suitable for applications that already model communication endpoints as adapters.
- The `_5` variant is a concrete five-output configuration of the generic class `GEN_A2X_AUI_DEMUX`; other variants can provide a different number of outputs.
- In contrast to a multiplexer, which combines multiple input values into one output, this block distributes one input value to one of several outputs.

Conclusion

`A2X_AUI_DEMUX_5` is a compact and reusable generic demultiplexor for adapter-based unidirectional communication. Its index-controlled output selection, five outputs, and value-change-triggered update behavior make it suitable for applications where a single value must be routed to one of several destinations without generating unnecessary events.

A2X_AUI_DEMUX_5_UNGATED



• – ◦ • – ◦ • – ◦ •

Introduction

A2X_AUI_DEMUX_5_UNGATED is a unidirectional 1-to-5 demultiplexer FB. It forwards one input value received through the `A2X` adapter socket to one of five `A2X` output adapters, depending on an index value received through the `AUI` adapter socket.

In contrast to a gated demultiplexer, this block does **not** perform change detection. Every newly computed or received result is forwarded unconditionally. This makes the FB suitable for consumers that require a continuous periodic data cadence, such as derivative or frequency calculations, even when the value itself has not changed.

Interface Structure

Event Inputs

There are no explicit event inputs declared in the interface list. The FB is controlled through its adapter sockets and plugs.

Event Outputs

No event outputs.

Data Inputs

There are no explicit data inputs. The input value is transmitted through the adapter socket `IN`.

Data Outputs

There are no explicit data outputs. The demultiplexed results are transmitted through the adapter plugs `OUT1` to `OUT5`.

Adapters

Adapter	Type	Direction	Description
`K`	`adapter::types::unidirectional::AUI`	Socket	Index input used to select the active output
`IN`	`adapter::types::unidirectional::A2X`	Socket	Input value to demultiplex
`OUT1`	`adapter::types::unidirectional::A2X`	Plug	Output value 1, selected when `K = 0`
`OUT2`	`adapter::types::unidirectional::A2X`	Plug	Output value 2, selected when `K = 1`
`OUT3`	`adapter::types::unidirectional::A2X`	Plug	Output value 3, selected when `K = 2`
`OUT4`	`adapter::types::unidirectional::A2X`	Plug	Output value 4, selected when `K = 3`
`OUT5`	`adapter::types::unidirectional::A2X`	Plug	Output value 5, selected when `K = 4`

Functionality

The FB acts as a demultiplexer between one input adapter and five output adapters. The index supplied through the `K` adapter determines which output receives the current value from `IN`.

The `K` index mapping is fixed:

- `K = 0` ⟶ `OUT1`

- `K = 1` ⟶ `OUT2`
- `K = 2` ⟶ `OUT3`
- `K = 3` ⟶ `OUT4`
- `K = 4` ⟶ `OUT5`

The designation “UNGATED” indicates that no gating mechanism suppresses repeated values. If a new result arrives, it is forwarded to the selected output regardless of whether the value differs from the previous one.

Technical Features

- 1-to-5 unidirectional adapter-based demultiplexing.
- Uses `adapter::types::unidirectional::A2X` for the data path.
- Uses `adapter::types::unidirectional::AUI` for index selection.
- No value-change detection.
- Every delivered result is passed through unconditionally.
- Interface contract is based on a generic implementation class `GEN_A2X_AUI_DEMUX`.
- Suitable for periodic and time-triggered processing chains.
- Licensed under EPL-2.0.

State Overview

This block does not rely on input value history to make forwarding decisions. Because it is ungated, it does not need to remember the last output value in order to suppress duplicate updates.

The only relevant runtime information is the currently selected index `K`. The observable behavior is therefore essentially stateless with respect to the incoming data values.

Application Scenarios

- Periodic derivative or frequency calculations where a constant input value must still produce a steady output cadence.
- Distribution of one input stream to multiple alternative consumers.
- Test and simulation environments that need deterministic forwarding on every cycle.
- Systems where downstream blocks expect a continuous update signal even when the data has not changed.
- Generic adapter-based selection between five unidirectional output paths.

Comparison with Similar Blocks

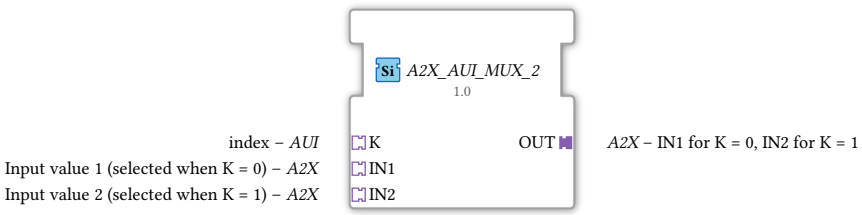
The gated variant `A2X_AUI_DEMUX_5` forwards a value only when a change is detected. `A2X_AUI_DEMUX_5_UNGATED` removes this restriction and forwards every newly produced result unconditionally.

This makes the gated variant useful for state-oriented or event-driven consumers that only react to actual changes. The ungated variant is better suited for time-driven calculations, control loops, and signal-processing blocks that require a regular data rhythm.

Conclusion

`A2X_AUI_DEMUX_5_UNGATED` is a simple and predictable adapter-based demultiplexer. It provides a clean mapping from one input adapter to five output adapters using an external index. By omitting change detection, it guarantees that every result cycle is propagated to the selected output. This behavior is ideal for applications that require a constant update flow or periodicity, independent of whether the actual data value has changed.

A2X_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

A2X_AUI_MUX_2 is a generic, adapter-based multiplexer for unidirectional IEC 61499 adapter connections. It selects one of two A2X input adapters using an AUI index adapter and forwards the selected value to an A2X output adapter. The output is only updated when the selected value actually changes, which helps to suppress unnecessary event traffic.

Interface Structure

Event Inputs

The FB does not declare any top-level event inputs. It is triggered through its connected adapter sockets.

Event Outputs

No event outputs.

Data Inputs

There are no direct data inputs. Input data is transported through the adapter sockets.

Data Outputs

There are no direct data outputs. Output data is transported through the output adapter plug.

Adapters

Kind	Name	Adapter Type	Comment
Plug	OUT	`adapter::types::unidirectional::A2X`	IN1 for K = 0, IN2 for K = 1
Socket	K	`adapter::types::unidirectional::AUI`	Index
Socket	IN1	`adapter::types::unidirectional::A2X`	Input value 1 (selected when K = 0)
Socket	IN2	`adapter::types::unidirectional::A2X`	Input value 2 (selected when K = 1)

Functionality

A2X_AUI_MUX_2 selects one of two A2X input streams based on the index supplied by the AUI adapter:

- K = 0 selects IN1.
- K = 1 selects IN2.

The selected value is written to the output adapter OUT. The generic runtime implementation `GEN_A2X_AUI_MUX` compares the newly selected value with the value currently present on OUT. If the values are the same, the output is not updated and no event is emitted. If the value differs, OUT is updated.

Technical Features

- Generic FB using the runtime class `GEN_A2X_AUI_MUX`.
- Fully adapter-based interface, no direct data I/O.
- Unidirectional data flow from the input adapters to the output adapter.
- Value-change detection avoids unnecessary output updates and events.
- Part of the package `adapter::selection::unidirectional`.
- Compliant with the IEC 61499-2 type specification.

State Overview

A2X_AUI_MUX_2 does not define an explicit ECC state machine. Instead, its behavior is implemented by the generic runtime class. Internally, the FB retains the last value written to OUT. When a new index is applied, the generic implementation checks whether the newly selected input differs from the stored value. If a change is detected, the output and internal state are updated. If no change is detected, the FB remains in its current output state.

Application Scenarios

- Selecting between two A2X data sources in an adapter-based automation system.
- Redundancy or failover switching based on an AUI index.
- Routing one of two adapter-based signal paths to a downstream consumer.
- Reducing event load by suppressing repeated identical values.

- Reusable selection logic in modular IEC 61499 applications.

Comparison with Similar Blocks

Compared with `AX_AUI_MUX_2`, this FB is the A2X variant. It uses A2X adapter types for the data paths and an AUI adapter for the index.

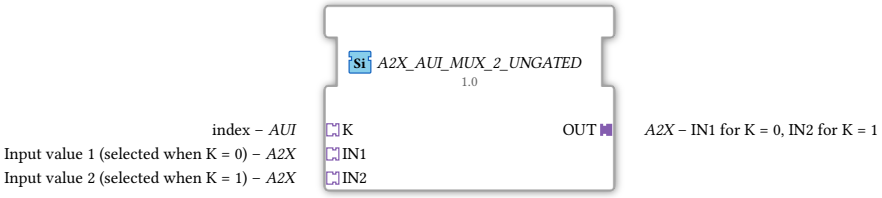
Compared with a conventional MUX function block, A2X_AUI_MUX_2 has no direct data inputs or outputs. All data exchange is encapsulated in adapters, which improves reusability and type decoupling.

Compared with a simple gate or switch, this FB provides real selection logic and integrated change detection, making it suitable for event-driven selection scenarios without redundant updates.

Conclusion

A2X_AUI_MUX_2 is a compact, generic, unidirectional multiplexer for IEC 61499 applications. It selects one of two A2X input adapters via an AUI index and forwards the selected value to an A2X output. The integrated value-change detection prevents unnecessary output events, making it well suited for event-driven automation and adapter-based system designs.

A2X_AUI_MUX_2_UNGATED



• – ○ • – ○ • – ○ •

Introduction

A2X_AUI_MUX_2_UNGATED is an adapter-based, event-driven multiplexer for selecting one of two unidirectional A2X data streams and forwarding it to a single output. It is an ungated variant of the A2X_AUI_MUX_2 function block. In contrast to a gated multiplexer, it does not perform any change detection. Every newly computed result is forwarded unconditionally, regardless of whether the value has actually changed.

This behavior is intended for consumers that require a periodic cadence or a regular data flow, such as derivative or frequency calculations, where the arrival of each new result is important even if the value remains identical to the previous one.

Interface Structure

The function block has no conventional event inputs or data inputs/outputs. Its interface is entirely built around unidirectional adapters. It has no event outputs.

Event Inputs

None.

Event Outputs

No event outputs.

Data Inputs

None.

Data Outputs

None.

Adapters

Kind	Name	Type	Description
Socket	`K`	`adapter::types::unidirectional::AUI`	Selection index. Selects `IN1` when `K = 0` and `IN2` when `K = 1`.

Socket	IN1	adapter::types::unidirectional::A2X	First value stream, selected when K = 0.
Socket	IN2	adapter::types::unidirectional::A2X	Second value stream, selected when K = 1.
Plug	OUT	adapter::types::unidirectional::A2X	Multiplexed output stream.

Functionality

The block acts as a two-input multiplexer for unidirectional adapter streams. The selection index is received through the AUI adapter socket K. When K = 0, the data arriving via IN1 is passed to OUT. When K = 1, the data arriving via IN2 is passed to OUT.

The "UNGATED" characteristic means that the block does not compare incoming values with previous values. Every incoming result is forwarded immediately to the output. This makes the block well suited for applications in which downstream blocks need each processing cycle to be triggered, even if the underlying value has not changed.

The block is a generic function block realized through the generic class name GEN_A2X_AUI_MUX. It is intended for use in unidirectional, adapter-based selection scenarios.

Technical Features

- Adapter-based interface with no direct data inputs or outputs.
- Supports one AUI index socket for selecting one of two A2X input sockets.
- Provides a single unidirectional A2X output plug.
- Forwards all incoming results unconditionally without change detection.
- Generic implementation using GEN_A2X_AUI_MUX.
- Package: adapter::selection::unidirectional.
- Suitable for periodic and cadence-sensitive data flow architectures.

State Overview

The block does not expose an explicit state machine. It behaves as a combinatorial, event-triggered adapter multiplexer. In the ungated variant, no internal state is used to store or compare previous input values. This means there is no hysteresis, no change detection latch, and no suppression of repeated values.

Application Scenarios

- **Derivative/Frequency Calculations**: Downstream blocks that need a new result at every input event, regardless of whether the value changed, benefit from the unconditional forwarding behavior.
- **Periodic Data Distribution**: When a consumer requires a fixed periodic data cadence, the ungated multiplexer ensures that every computed result is delivered.
- **Adapter-Based Selecting**: When two unidirectional data sources must be selected dynamically through an adapter-based index, this block provides a clean and reusable solution.
- **Stream Redirection**: Useful in processing chains where the active input source may switch between two producers without losing event rhythm.

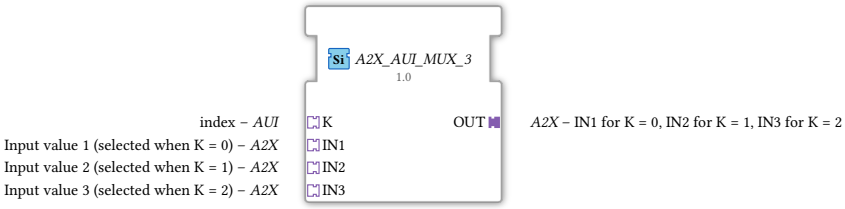
Comparison with Similar Blocks

- **A2X_AUI_MUX_2**: The gated variant performs change detection. It only forwards data when the incoming value differs from the previously forwarded value. This reduces unnecessary downstream activity but can break a periodic cadence.
- **A2X_AUI_MUX_2_UNGATED**: This variant forwards every newly calculated result without compare-and-skip logic. It preserves the event rhythm but may produce repeated identical values on the output.
- **Conventional MUX blocks**: Standard multiplexers typically use direct data inputs and a scalar selector. This block instead uses unidirectional adapters, making it more suitable for structured, event-driven adapter communication.

Conclusion

A2X_AUI_MUX_2_UNGATED is a compact, adapter-based multiplexer that selects between two unidirectional A2X streams using an AUI index. Its key feature is the absence of change detection: every result is forwarded immediately. This makes it ideal for applications that rely on a stable periodic event flow, such as derivative or frequency calculations, where the arrival of each result itself carries information.

A2X_AUI_MUX_3



• – ○ • – ○ • – ○ •

Introduction

A2X_AUI_MUX_3 is a generic adapter-based 3-to-1 multiplexer function block for IEC 61499 applications. It selects one of three A2X input adapters and forwards the selected value to an A2X output adapter. The selection index is supplied through an AUI adapter. The block is implemented as the typed generic variant `GEN_A2X_AUI_MUX` and is designed so that the output adapter is only updated when the selected value actually changes.

Interface Structure

The function block has no direct data inputs or data outputs. All value and control information is exchanged through the adapter interface.

Event Inputs

None.

Event Outputs

No event outputs.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Plug	`OUT`	`adapter::types::unidirectional::A2X`	Selected input value: IN1 for K = 0, IN2 for K = 1, IN3 for K = 2
Socket	`K`	`adapter::types::unidirectional::AUI`	Index used to select the active multiplexer input
Socket	`IN1`	`adapter::types::unidirectional::A2X`	Input value 1, selected when K = 0
Socket	`IN2`	`adapter::types::unidirectional::A2X`	Input value 2, selected when K = 1
Socket	`IN3`	`adapter::types::unidirectional::A2X`	Input value 3, selected when K = 2

Functionality

A2X_AUI_MUX_3 behaves as an adapter-based multiplexer:

1. A new selection index is received through the `K` adapter.
2. The index value determines which input adapter is active:
   - K = 0 selects `IN1`
   - K = 1 selects `IN2`
   - K = 2 selects `IN3`
3. The value of the selected input adapter is compared with the current value on the output adapter `OUT`.

This behavior reduces unnecessary event and data traffic on the adapter connection. The block is therefore especially useful in applications where the index may be set repeatedly to the same value, but only actual value changes should be propagated.

Technical Features

- Generic implementation based on `GEN_A2X_AUI_MUX`
- Three A2X input adapters: `IN1`, `IN2`, `IN3`
- One A2X output adapter: `OUT`
- One AUI index adapter: `K`
- Fully adapter-based interface; no direct data inputs or outputs
- Unidirectional adapter communication
- Change-only output update behavior
- Suitable for distributed IEC 61499 applications
- Licensed under EPL-2.0

State Overview

The block does not declare an explicit state machine in its interface definition. Its observable behavior can be described by the following conceptual states:

- **Idle**: waiting for a new index value on the `K` adapter.
- **Evaluate**: reading the index and selecting the corresponding input adapter.
- **Compare**: checking whether the selected input value differs from the current output value.
- **Publish**: updating the `OUT` adapter if an actual change occurred.

This state behavior ensures that redundant selections do not cause unnecessary output activity.

Application Scenarios

A2X_AUI_MUX_3 is suitable for the following use cases:

- Selecting one of three process values or measurement signals connected via adapters.
- Routing different data sources to a single consumer without using direct data connections.
- Building adapter-based signal selection logic in distributed control systems.
- Reducing network or communication load by suppressing output updates when the selected value is unchanged.
- Using as a reusable building block in larger adapter-based multiplexer or routing structures.

Comparison with Similar Blocks

Compared to a conventional data multiplexer, A2X_AUI_MUX_3 does not expose physical data inputs and outputs. Instead, it uses unidirectional adapters for both values and index, making integration in adapter-based system architectures easier.

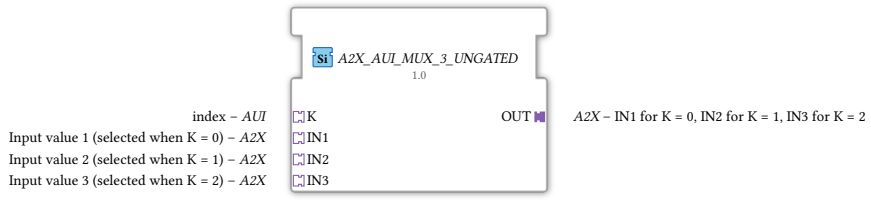
In contrast to standard multiplexers that always copy the selected input to the output, A2X_AUI_MUX_3 only updates the output when the selected value actually changes. This makes it more efficient in event-driven systems.

The block is an A2X variant of the existing `AX_AUI_MUX_3` concept. The main difference is the use of the `A2X` adapter type for the value interfaces while the index remains on the `AUI` adapter type.

Conclusion

A2X_AUI_MUX_3 is a compact, adapter-based 3-to-1 multiplexer that combines selection logic with change detection. Its unidirectional adapter interface and event-on-change behavior make it well suited for distributed IEC 61499 applications where efficient communication and minimal unnecessary events are important.

A2X_AUI_MUX_3_UNGATED



• – ○ • – ○ • – ○ •

Introduction

A2X_AUI_MUX_3_UNGATED is a multiplexer function block from the A2X family. It selects one of three A2X input streams and forwards it to a single A2X output stream. The selection is controlled through an AUI index adapter.

The key property of this block is described by the "UNGATED" suffix: it does **not** perform any change detection. Every newly calculated result is passed on unconditionally, even if the value itself has not changed. This makes the block suitable for consumers that require a periodic data cadence, for example derivative or frequency calculations.

Interface Structure

The block does not expose direct event inputs, data inputs, or data outputs. All input and output information is exchanged through adapter connections. The following subsets describe the FB interface as visible in the type definition.

Event Inputs

None.

Event Outputs

No event outputs.

Data Inputs

None at the FB boundary. Input data is carried via the adapter sockets.

Data Outputs

None at the FB boundary. Output data is carried via the adapter plug.

Adapters

Role	Name	Type	Direction	Description
Output	OUT	adapter::types::unidirectional::A2X	Plug	Selected value stream. IN1 for K=0, IN2 for K=1, IN3 for K=2.
Selection	K	adapter::types::unidirectional::AUI	Socket	Index input that determines which A2X input is routed to OUT.
Input 1	IN1	adapter::types::unidirectional::A2X	Socket	Input value 1, selected when K = 0.
Input 2	IN2	adapter::types::unidirectional::A2X	Socket	Input value 2, selected when K = 1.
Input 3	IN3	adapter::types::unidirectional::A2X	Socket	Input value 3, selected when K = 2.

Functionality

The function block behaves as a three-way multiplexer:

1. It reads the current index from the K adapter.
2. According to the index value, it selects one of the three A2X input adapters.
3. It forwards the selected input stream to the OUT adapter.

The block is "ungated", meaning there is no internal comparison between the current value and the previously forwarded value. Every new calculation result from the selected input is forwarded without filtering. This is intentional and is the main difference from a change-detecting multiplexer.

Technical Features

- Generic FB implementation based on the attribute eclipse4diac::core::GenericClassName with the value GEN_A2X_AUI_MUX.
- The block supports exactly three A2X input channels.
- The adapter types are defined in the package adapter::types::unidirectional.

- No direct data inputs or outputs; all data transfer occurs through unidirectional adapters.
- No value-change recognition, no gating logic, no event suppression.

State Overview

The block is intentionally stateless with respect to value history. It does not store previous A2X values and does not compare incoming values with earlier ones. Once an index is supplied via `K`, the corresponding input is routed to `OUT` without additional conditions.

The only implicit state is the currently active selection index. After a new index is received and processed, the new selection becomes active.

Application Scenarios

Typical use cases include:

- Periodic distribution of measured or calculated values to downstream blocks that must be triggered on every scan, regardless of whether the value changed.
- Derivative and frequency calculations that need a continuous data stream.
- Data routing in automation systems where multiple A2X sources must be cyclically switched to one consumer.
- Applications that require a steady cadence for timing-sensitive calculations.

Comparison with Similar Blocks

The closest relative is `A2X_AUI_MUX_3`. Both blocks implement the same three-input multiplexer structure. The difference is the presence of change detection:

- `A2X_AUI_MUX_3` suppresses forwarding when the value has not changed.
- `A2X_AUI_MUX_3_UNGATED` forwards every new result unconditionally.

For consumers that only need to react to actual value changes, the gated variant is more efficient. For consumers that require a periodic processing rhythm, the ungated variant is the better choice because it preserves the data cadence.

Conclusion

A2X_AUI_MUX_3_UNGATED is a useful multiplexer block for unidirectional A2X data streams. By omitting change detection, it guarantees that every incoming result is passed through, making it particularly suitable for applications with periodic timing requirements. Its interface is compact and based entirely on adapters, allowing clean integration into 4diac IEC 61499 systems.

A2X_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

A2X_AUI_MUX_4 is a generic, adapter-based 4-to-1 multiplexer for Eclipse 4diac. It connects four unidirectional A2X adapter inputs (`IN1` … `IN4`) to one A2X adapter output (`OUT`). The active input is selected by the value received on the AUI adapter `K`. The output is not simply forwarded from the selected input; it is updated only when the selected value actually changes. This avoids unnecessary downstream events.

Interface Structure

The FB has no ordinary event or data inputs/outputs. All control and data information is exchanged through adapter interfaces. There are no event outputs.

Event Inputs

None.

Event Outputs

No event outputs.

Data Inputs

None. All payload data is supplied via adapter sockets.

Data Outputs

None. All payload data is provided through the `OUT` adapter plug.

Adapters

Port	Name	Type	Direction	Description
Socket	`K`	`adapter::types::unidirectional::AUI`	Input	Carries the selection index.
Socket	`IN1`	`adapter::types::unidirectional::A2X`	Input	Value selected when K = 0.
Socket	`IN2`	`adapter::types::unidirectional::A2X`	Input	Value selected when K = 1.
Socket	`IN3`	`adapter::types::unidirectional::A2X`	Input	Value selected when K = 2.
Socket	`IN4`	`adapter::types::unidirectional::A2X`	Input	Value selected when K = 3.
Plug	`OUT`	`adapter::types::unidirectional::A2X`	Output	Selected value; updated only when the selected value changes.

Functionality

The block continuously receives the selector value `K` and the four A2X input adapter values. It determines which input is active according to the current value of `K`:

- K = 0 selects `IN1`.
- K = 1 selects `IN2`.
- K = 2 selects `IN3`.
- K = 3 selects `IN4`.

When the active index changes, or when the value on the currently active input changes, the block compares the new selected value with the value previously sent to `OUT`. If the values differ, `OUT` is updated with the new value. If the values are equal, the output remains unchanged and no event is generated. This prevents event flooding and ensures that downstream adapter consumers only react to real value changes.

Technical Features

- Generic FB implementation: the `eclipse4diac::core::GenericClassName` attribute is `GEN_A2X_AUI_MUX`, so the concrete behavior is provided by a generic runtime backend with this name.
- Adapter-based interface: uses unidirectional `A2X` and `AUI` adapter types; no elementary data ports are required.
- Change-driven output: `OUT` is only updated on actual selected-value changes.
- Four selectable inputs: enough for 4-to-1 selection scenarios.
- Compiler/package namespace: `adapter::selection::unidirectional`.

State Overview

The XML definition does not contain an explicit ECC state machine; the actual behavior is implemented by the generic backend `GEN_A2X_AUI_MUX`. Conceptually, the block maintains a single internal state: the last value forwarded to `OUT`. The relevant state transition is:

- **No change**: selected value equals the last forwarded value → no output update.
- **Change**: selected value differs from the last forwarded value → `OUT` is updated.

This implicit state allows the block to suppress repeated transmissions of identical values.

Application Scenarios

- Selecting one of four A2X data sources at runtime, using an AUI adapter as the selector.
- Connecting multiple A2X producer adapters to a single consumer without exposing elementary data types.

- Event-driven systems where downstream components should be triggered only when the selected value actually changes.
- Reducing unnecessary network or adapter communication in distributed 4diac applications.
- Using as an adapter-based alternative to a classic multiplexer in Eclipse 4diac.

Comparison with Similar Blocks

A2X_AUI_MUX_4 is an A2X variant of AX_AUI_MUX_4. Compared with a conventional data-input multiplexer, it uses adapter ports instead of elementary data inputs and outputs. This makes it suitable for structured or adapter-based data exchange.

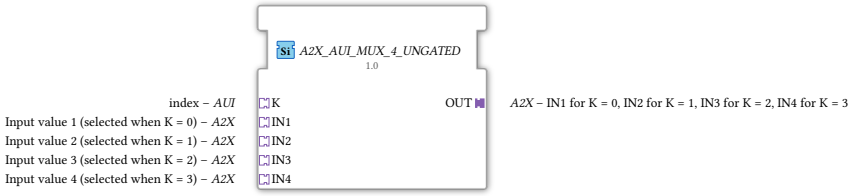
Block / Approach	Interface	Update behavior
A2X_AUI_MUX_4	4 × A2X sockets, 1 × AUI socket, 1 × A2X plug	Output update only on actual value change.
AX_AUI_MUX_4	Adapter-based, non-A2X variant	Similar selection structure; different adapter type.
Classic MUX	Elementary data inputs and data output	Usually event-triggered; may forward unchanged values and emit an event on every selection.

The change-detection behavior of A2X_AUI_MUX_4 is its main distinction from a standard multiplexer.

Conclusion

A2X_AUI_MUX_4 provides an adapter-oriented, change-driven multiplexing solution for Eclipse 4diac. It selects one of four A2X inputs using an AUI selector and forwards the selected value through an A2X output. Because the output update is suppressed when no actual value change occurs, the block helps keep event chains lean and efficient. It is especially useful in adapter-based automation architectures where only meaningful data changes should propagate to consumers.

A2X_AUI_MUX_4_UNGATED



• – ○ • – ○ • – ○ •

Introduction

A2X_AUI_MUX_4_UNGATED is a unidirectional, adapter-based 4-to-1 multiplexer for A2X adapter streams. It selects one of four A2X input adapters according to an index received through an AUI adapter and forwards the selected stream to an A2X output plug.

In contrast to a change-detecting multiplexer, the “UNGATED” variant does not compare old and new values. Every newly calculated result is forwarded unconditionally. This makes the block especially suitable for consumers that require a periodic or continuous sample flow, such as derivative or frequency calculations.

• – ○ • – ○ • – ○ •

Interface Structure

The FB has no direct data inputs or data outputs. All runtime values are transported through unidirectional adapters. The interface consists of one index socket, four input sockets, and one output plug.

Event Inputs

There are no explicit event inputs. Control and data flow are handled through the adapter connections.

Event Outputs

No event outputs.

Data Inputs

None. All input values are carried by the adapter sockets.

Data Outputs

None. The selected A2X value is provided through the `OUT` adapter plug.

Adapters

Adapter	Type	Direction	Description
`K`	`adapter::types::unidirectional::AUI`	Socket	Selection index.
`IN1`	`adapter::types::unidirectional::A2X`	Socket	Input value 1, selected when `K = 0`.
`IN2`	`adapter::types::unidirectional::A2X`	Socket	Input value 2, selected when `K = 1`.
`IN3`	`adapter::types::unidirectional::A2X`	Socket	Input value 3, selected when `K = 2`.
`IN4`	`adapter::types::unidirectional::A2X`	Socket	Input value 4, selected when `K = 3`.
`OUT`	`adapter::types::unidirectional::A2X`	Plug	Selected A2X output stream.

• – ◦ • – ◦ • – ◦ •

Functionality

A2X_AUI_MUX_4_UNGATED acts as a four-channel multiplexer for A2X adapter data. The `K` socket carries the multiplexer index. This index is interpreted as an integer value in the range `0` to `3`.

The input selection is mapped as follows:

- `K = 0` ⟶ `IN1`
- `K = 1` ⟶ `IN2`
- `K = 2` ⟶ `IN3`
- `K = 3` ⟶ `IN4`

The selected input adapter is connected to the `OUT` plug, and the data is forwarded without any value-change filtering.

The main characteristic of this FB is that it is “ungated”. A normal gated multiplexer may suppress an output update when the selected value has not changed. This block does not do that. Every updated result coming from the selected input is passed through to `OUT`, even if it is identical to the previously forwarded value.

• – ◦ • – ◦ • – ◦ •

Technical Features

- Generic FB implementation class: `GEN_A2X_AUI_MUX`
- Adapter-based interface: no conventional data inputs or outputs
- Unidirectional adapter types from the package `adapter::types::unidirectional`
- Four A2X input channels plus one AUI index channel
- One A2X output plug
- No change detection or value filtering
- Suitable for periodic, time-driven data consumers

• – ◦ • – ◦ • – ◦ •

State Overview

The FB type is declared as a generic FB, so the XML does not contain an explicit execution control chart. The actual behavior is provided by the generic implementation `GEN_A2X_AUI_MUX`. Conceptually, the block operates in the following cycle:

State	Description
Waiting	The FB waits for an update on the `K` index adapter or on the selected A2X input.

Decoding	The index K is evaluated and mapped to IN1, IN2, IN3, or IN4.
Forwarding	The selected input data is passed to the OUT plug.

No state compares current and previous values. Therefore, every processed result is forwarded unconditionally.

• – ○ • – ○ • – ○ •

Application Scenarios

- **Periodic derivative calculations**
  Derivative blocks need a constant sample rhythm. If a value remains unchanged and a change-detecting multiplexer blocks the update, the derivative calculation may miss a sample interval. This UB forwards every new result and preserves the periodic cadence.
- **Frequency or rate measurement**
  Frequency and rate calculations rely on time-based updates rather than value changes. The ungated behavior keeps the downstream calculation active.
- **Selecting among four A2X sources**
  The FB can switch between four independent A2X adapter streams while maintaining a single, continuous output stream.
- **Adapter-based signal routing**
  Because the FB uses only adapters, it can be integrated into unidirectional, adapter-oriented data flows without mixing in conventional data inputs or outputs.

• – ○ • – ○ • – ○ •

Comparison with Similar Blocks

Block	Behavior
A2X_AUI_MUX_4	A four-input A2X multiplexer with change detection. It may suppress forwarding when the selected input value has not changed.
A2X_AUI_MUX_4_UNGATED	Same multiplexing function, but without change detection. Every newly calculated result is forwarded unconditionally.
AX_AUI_MUX_4_UNGATED	A non-A2X variant of the same ungated selection concept. The A2X variant uses dedicated unidirectional A2X adapter types.

The essential difference is the “UNGATED” behavior: downstream blocks that require uniform sample timing should use this variant instead of a change-detecting multiplexer.

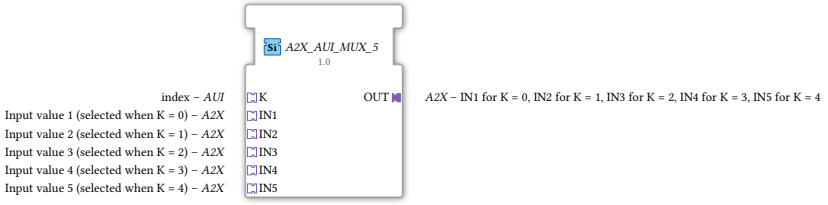
• – ○ • – ○ • – ○ •

Conclusion

A2X_AUI_MUX_4_UNGATED is a specialized adapter-based multiplexer that selects one of four A2X inputs and forwards the selected data without change detection. Its strength is deterministic, periodic forwarding of all calculated results, which makes it ideal for time-critical consumers such as derivative and frequency calculations.

By combining a clear four-input structure, and an AUI-based index interface, the block offers a simple yet flexible building block for unidirectional adapter-oriented control and measurement applications.

A2X_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

A2X_AUI_MUX_5 is an adapter-based unidirectional multiplexer for IEC 61499 applications. It selects one of five A2X input adapters, **IN1** to **IN5**, based on the index value supplied by the **K** adapter. The selected value is forwarded to the **OUT** A2X output adapter.

The function block is implemented as a generic FB and uses change detection to minimize unnecessary adapter traffic. The output adapter **OUT** is only updated when the selected value actually changes.

Interface Structure

The block communicates mainly through adapters. There are no conventional event inputs, data inputs, or data outputs. All value data is exchanged through the A2X and AUI adapter connections.

Event Inputs

There are no event inputs defined for this function block.

Event Outputs

No event outputs.

Data Inputs

There are no data inputs defined for this function block.

Data Outputs

There are no data outputs defined for this function block.

Adapters

Direction	Name	Adapter Type	Comment
Socket	K	adapter::types::unidirectional::AUI	Selection index.
Socket	IN1	adapter::types::unidirectional::A2X	Input value 1, selected when K = 0.
Socket	IN2	adapter::types::unidirectional::A2X	Input value 2, selected when K = 1.
Socket	IN3	adapter::types::unidirectional::A2X	Input value 3, selected when K = 2.
Socket	IN4	adapter::types::unidirectional::A2X	Input value 4, selected when K = 3.
Socket	IN5	adapter::types::unidirectional::A2X	Input value 5, selected when K = 4.
Plug	OUT	adapter::types::unidirectional::A2X	Selected A2X value forwarded to the output.

Functionality

A2X_AUI_MUX_5 behaves as a 5-to-1 adapter multiplexer:

1. The index value is received through the **K** AUI adapter.
2. The corresponding input adapter **IN1** to **IN5** is selected.
3. The value of the selected input is forwarded to the **OUT** adapter.

If the index changes to an input that carries the same value as the current output, the output is not updated. This avoids unnecessary event and data propagation in the connected IEC 61499 application.

Since the FB is a generic function block, the actual behavior is delegated to the generic class **GEN_A2X_AUI_MUX**. The runtime environment must provide this generic backend.

Technical Features

- 5-to-1 adapter-based multiplexing.
- Unidirectional data flow through A2X and AUI adapters.
- Change-triggered output update.
- No scalar data inputs or outputs are required.
- Generic FB design using `eclipse4diac::core::GenericClassName`.
- Generic backend: `GEN_A2X_AUI_MUX`.
- Located in the adapter package `adapter::selection::unidirectional`.
- Designed according to the IEC 61499-2 FB type identification structure.

State Overview

The FBType XML does not define an explicit ECC. Instead, the behavior is provided by the generic backend. Conceptually, the function block can be described using two states:

State	Description
Idle	Waiting for a change on the K index or on the currently selected input adapter.
Updating	The selected value has changed; OUT is updated.

After an update, the block returns to the idle state. If an incoming value or a new index selection does not produce a different output value, no state transition to updating occurs and no event is emitted.

Application Scenarios

- **Source selection:** Connect up to five A2X data sources and dynamically select the active source using an AUI index.
- **Event-efficient data forwarding:** Downstream logic is triggered only when the output value really changes.
- **Redundancy handling:** Switch between redundant input channels without propagating duplicate values.
- **Adapter-based system integration:** Integrate the block into unidirectional adapter chains where typed adapter connections are preferred over scalar data wiring.

Comparison with Similar Blocks

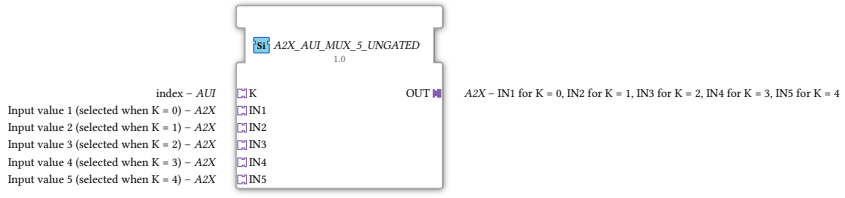
Compared to the **AX_AUI_MUX_5** block, this FB is the A2X variant and uses the A2X adapter type for its data interfaces. The main improvement is the change-detection behavior: the output update is suppressed when the selected value has not actually changed.

Compared to a conventional multiplexer FB with scalar data inputs and a numeric selector, A2X_AUI_MUX_5 uses typed adapters and provides a unidirectional connection style. This makes it more suitable for modular, adapter-oriented IEC 61499 architectures.

Conclusion

A2X_AUI_MUX_5 is a compact and event-efficient adapter multiplexer for selecting one of five A2X input channels. Its unidirectional adapter interface, generic implementation, and change-triggered output update make it suitable for modern IEC 61499 applications where adapter-based communication and reduced event traffic are important.

A2X_AUI_MUX_5_UNGATED



• – ○ • – ○ • – ○ •

Introduction

The **A2X_AUI_MUX_5_UNGATED** is an adapter-based, unidirectional 5-to-1 multiplexer. It selects one of five `A2X` input adapters through an `AUI` index adapter and forwards the selected value to an `A2X` output adapter.

The `_UNGATED` variant does **not** perform change detection: every newly computed result is forwarded unconditionally. This makes it suitable for consumers that require a periodic output cadence independent of whether the actual value has changed, for example derivative or frequency calculations.

Interface Structure

Event Inputs

None.

The FB is triggered through its adapter sockets rather than through explicit event inputs.

Event Outputs

No event outputs.

Data Inputs

None.

All data is transported through the adapter sockets.

Data Outputs

None.

All data is transported through the adapter plug.

Adapters

Name	Type	Direction	Comment
`OUT`	`adapter::types::unidirectional::A2X`	Plug	Selected input value: `IN1` for `K = 0`, `IN2` for `K = 1`, `IN3` for `K = 2`, `IN4` for `K = 3`, `IN5` for `K = 4`.
`K`	`adapter::types::unidirectional::AUI`	Socket	Index selector.
`IN1`	`adapter::types::unidirectional::A2X`	Socket	Input value 1, selected when `K = 0`.
`IN2`	`adapter::types::unidirectional::A2X`	Socket	Input value 2, selected when `K = 1`.
`IN3`	`adapter::types::unidirectional::A2X`	Socket	Input value 3, selected when `K = 2`.
`IN4`	`adapter::types::unidirectional::A2X`	Socket	Input value 4, selected when `K = 3`.
`IN5`	`adapter::types::unidirectional::A2X`	Socket	Input value 5, selected when `K = 4`.

Functionality

The FB works as a unidirectional adapter multiplexer:

1. The index adapter `K` determines which of the five input adapters is currently selected.
2. The selected input adapter is connected through to the output adapter `OUT`.
3. Every newly available result on the selected input is forwarded immediately and unconditionally.

The selection mapping is zero-based:

`K`	Selected Input
`0`	`IN1`
`1`	`IN2`
`2`	`IN3`
`3`	`IN4`
`4`	`IN5`

Because the FB is **ungated**, it does not compare old and new values and does not suppress repeated or unchanged data. Any refresh of the selected input results in an output update.

Technical Features

- Adapter-only interface with no direct data inputs or outputs.
- Unidirectional data flow from input sockets to the output plug.
- 5 input channels and 1 output channel.
- Zero-based channel selection via the `AUI` index adapter.
- No change detection, no value filtering, no hysteresis.
- Configured as a generic FB type using `GEN_A2X_AUI_MUX`.
- Belongs to the compiler package `adapter::selection::unidirectional`.

State Overview

The FB does not define an explicit internal state machine. Its behavior is essentially stateless and transparent:

Aspect	Description
Current selection	Determined by the current value delivered through the `K` adapter.
Forwarding	The selected `INx` adapter is continuously passed through to `OUT`.
Change detection	Not present. Every newly computed result is forwarded.

There is no stored “last value” and no internal comparison logic.

Application Scenarios

Typical use cases for `A2X_AUI_MUX_5_UNGATED` include:

- Periodic derivative calculations that require an input sample on every cycle, even if the value is unchanged.
- Frequency or rate-of-change computations that depend on a constant time base.
- Data acquisition systems selecting between multiple sensors while maintaining a regular output rhythm.
- Pipelines where downstream blocks expect a fixed update cadence rather than change-triggered events.
- Adapter-based unidirectional communication paths that need a simple 5-to-1 selection mechanism.

Comparison with Similar Blocks

The closest counterpart is the change-detecting `A2X_AUI_MUX_5` variant.

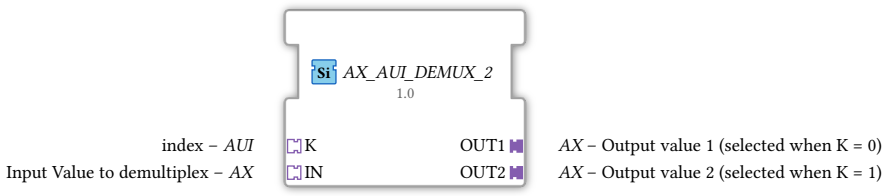
Feature	`A2X_AUI_MUX_5`	`A2X_AUI_MUX_5_UNGATED`
Change detection	Yes	No
Forwarding behavior	Only forwards when the value has changed	Forwards every newly computed result
Output cadence	Event-driven / aperiodic	Periodic cadence possible
Best suited for	Consumers that react only to value changes	Consumers that need a continuous or periodic stream of results
CPU load on unchanged data	Lower	Higher, because every result is passed through

Thus, the ungated variant is not a replacement for the gated variant, but a complementary block for time-oriented applications.

Conclusion

`A2X_AUI_MUX_5_UNGATED` is a compact, adapter-only multiplexer for selecting one of five unidirectional `A2X` inputs via an `AUI` index. Its main characteristic is the unconditional forwarding of every newly computed result, independent of value changes. This behavior makes it well suited for periodic, time-sensitive consumers, especially derivative and frequency calculations, while still providing the familiar multiplexer structure.

AX_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

AX_AUI_DEMUX_2 is the adapter-based variant of the generic demultiplexer for data type BOOL. Unlike AX_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AX_AUI_DEMUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the Boolean value to forward, routed to exactly one of the outputs (AX adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AX adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 2 output adapters (OUT1 … OUT2) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AX_AUI_DEMUX) – shared across all port counts of this block family (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) ⟶ determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a Boolean value to up to 2 downstream consumers, selected via an adapter-carried index

- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AX_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

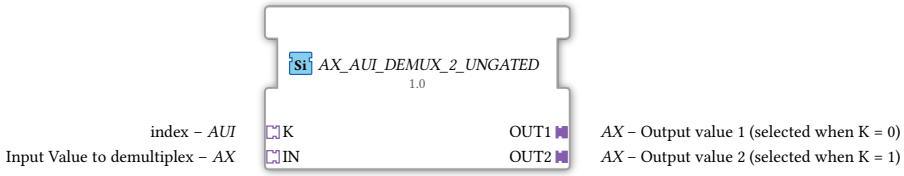
- **`AX_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AX_AUI_DEMUX_2` carries the demultiplexer logic of the `AX_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AX_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `BOOL`. Unlike `AX_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AX_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the Boolean value to forward, routed to exactly one of the outputs (AX adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AX adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_DEMUX`) – shared across all port counts of this block family (AX_AUI_DEMUX_2_UNGATED … AX_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a Boolean value to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AX_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

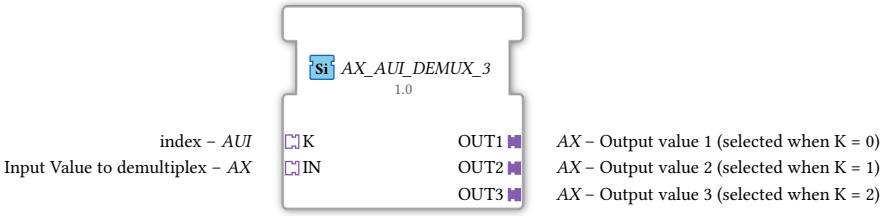
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AX_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AX_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AX_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AX_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `BOOL`. Unlike `AX_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AX_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the Boolean value to forward, routed to exactly one of the outputs (AX adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AX adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AX adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_DEMUX`) – shared across all port counts of this block family (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a Boolean value to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AX_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

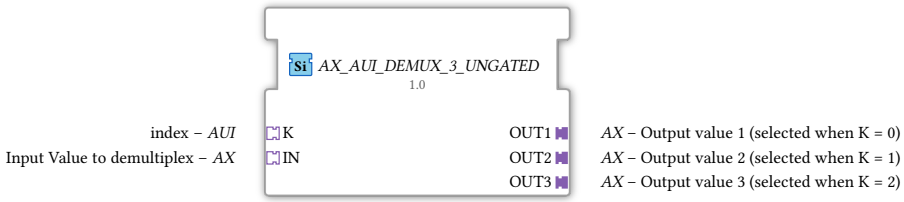
- **`AX_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AX_AUI_DEMUX_3` carries the demultiplexer logic of the `AX_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AX_AUI_DEMUX_3_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `BOOL`. Unlike `AX_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AX_AUI_DEMUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the Boolean value to forward, routed to exactly one of the outputs (AX adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AX adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AX adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_DEMUX`) – shared across all port counts of this block family (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a Boolean value to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AX_DEMUX_3 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

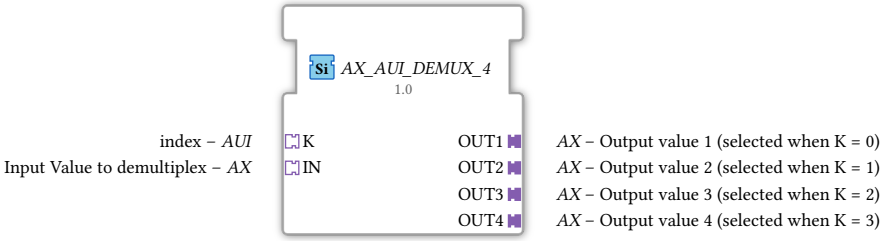
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **AX_AUI_DEMUX_3**: The gated variant – updates the output only on an actual value change.

Conclusion

AX_AUI_DEMUX_3_UNGATED carries the demultiplexer logic of the AX_DEMUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

AX_AUI_DEMUX_4 is the adapter-based variant of the generic demultiplexer for data type BOOL. Unlike AX_DEMUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AX_AUI_DEMUX_4 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the Boolean value to forward, routed to exactly one of the outputs (AX adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AX adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AX adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AX adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_DEMUX`) – shared across all port counts of this block family (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a Boolean value to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖ Comparison with Similar Blocks

Compare with `AX_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

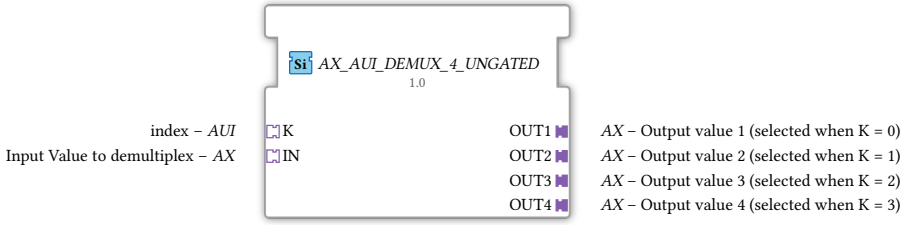
- **`AX_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AX_AUI_DEMUX_4` carries the demultiplexer logic of the `AX_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AX_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `BOOL`. Unlike `AX_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AX_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the Boolean value to forward, routed to exactly one of the outputs (AX adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AX adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AX adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AX adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_DEMUX`) – shared across all port counts of this block family (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a Boolean value to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AX_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

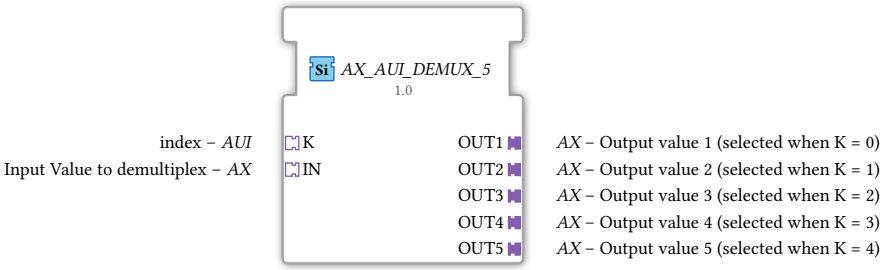
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AX_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AX_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AX_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AX_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `BOOL`. Unlike `AX_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AX_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the Boolean value to forward, routed to exactly one of the outputs (AX adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AX adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AX adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AX adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AX adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_DEMUX`) – shared across all port counts of this block family (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a Boolean value to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AX_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

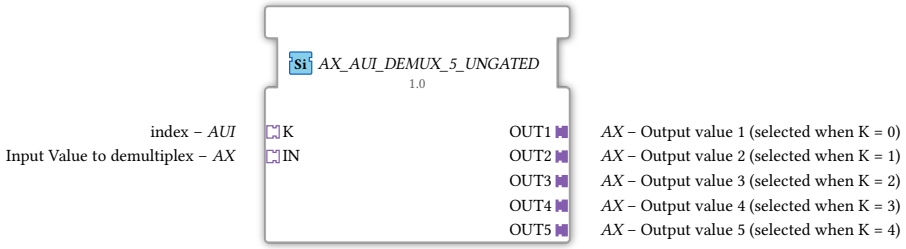
- **`AX_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AX_AUI_DEMUX_5 carries the demultiplexer logic of the AX_DEMUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of AX_AUI_DEMUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AX_AUI_DEMUX_5_UNGATED is the adapter-based variant of the generic demultiplexer for data type BOOL. Unlike AX_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AX_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the Boolean value to forward, routed to exactly one of the outputs (AX adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AX adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AX adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AX adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AX adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_DEMUX`) – shared across all port counts of this block family (AX_AUI_DEMUX_2 … AX_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a Boolean value to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AX_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

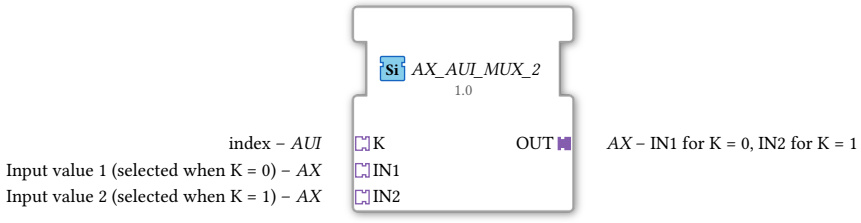
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AX_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AX_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AX_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_MUX_2



• – ○ • – ○ • – ○ •

Introduction

`AX_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `BOOL`. Unlike `AX_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its

own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AX_AUI_MUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 2, passed through to OUT when the index is K = 0 (AX adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to OUT when the index is K = 1 (AX adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN2:

1. The current value of K.D1 determines which of the 2 input adapters (IN1 … IN2) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AX_AUI_MUX) – shared across all port counts of this block family (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN2) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 Boolean value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AX_MUX_2 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

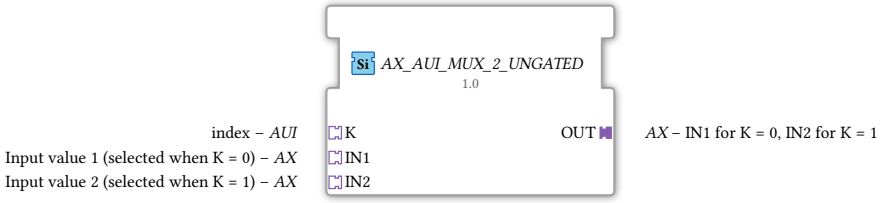
- **AX_AUI_MUX_2_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AX_AUI_MUX_2 carries the multiplexer logic of the AX_MUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AX_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AX_AUI_MUX_2_UNGATED is the adapter-based variant of the generic multiplexer for data type BOOL. Unlike AX_MUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AX_AUI_MUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.

- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AX adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AX adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` ... `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_MUX`) – shared across all port counts of this block family (AX_AUI_MUX_2_UNGATED ... AX_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 Boolean value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AX_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

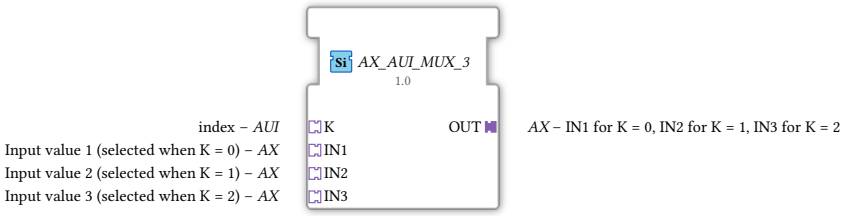
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AX_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AX_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AX_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_MUX_3



• – ○ • – ○ • – ○ •

Introduction

AX_AUI_MUX_3 is the adapter-based variant of the generic multiplexer for data type BOOL. Unlike AX_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AX_AUI_MUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (AX adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (AX adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (AX adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AX_AUI_MUX) – shared across all port counts of this block family (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 Boolean value sources through a centrally managed adapter index

- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AX_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

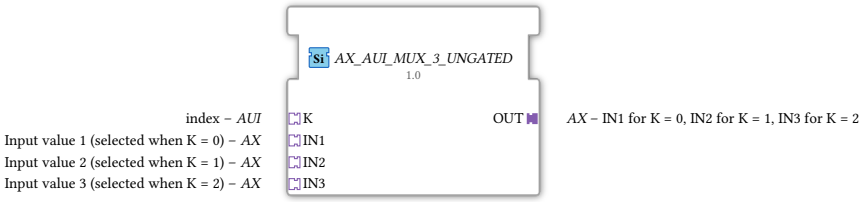
- **`AX_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AX_AUI_MUX_3` carries the multiplexer logic of the `AX_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_AUI_MUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AX_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `BOOL`. Unlike `AX_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AX_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AX adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AX adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AX adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_MUX`) – shared across all port counts of this block family (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 Boolean value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AX_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

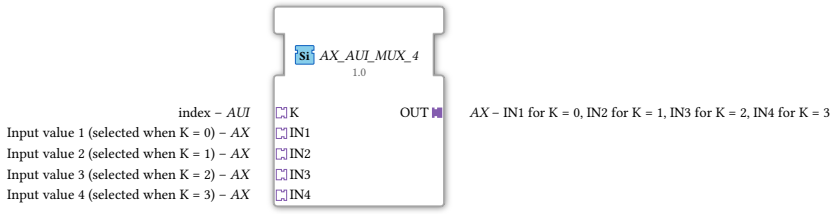
Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AX_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AX_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AX_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AX_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `BOOL`. Unlike `AX_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AX_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AX adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AX adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AX adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AX adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_MUX`) – shared across all port counts of this block family (AX_AUI_MUX_2 … AX_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 Boolean value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AX_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

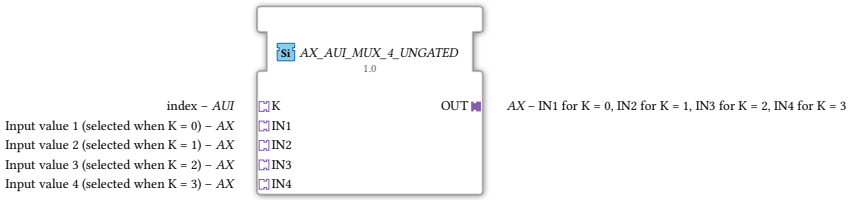
- **`AX_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AX_AUI_MUX_4` carries the multiplexer logic of the `AX_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AX_AUI_MUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• — ○ • — ○ • — ○ •

Introduction

`AX_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `BOOL`. Unlike `AX_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AX_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AX adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AX adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AX adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AX adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_MUX`) – shared across all port counts of this block family (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 Boolean value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AX_MUX_4 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

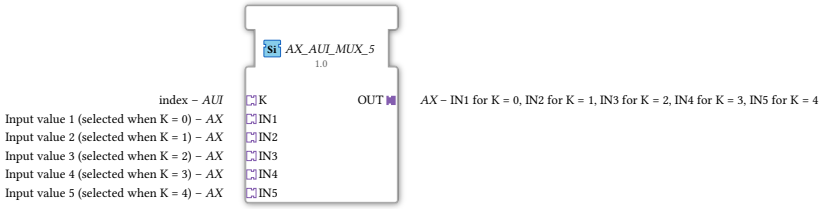
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **AX_AUI_MUX_4**: The gated variant – updates the output only on an actual value change.

Conclusion

AX_AUI_MUX_4_UNGATED carries the multiplexer logic of the AX_MUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

AX_AUI_MUX_5 is the adapter-based variant of the generic multiplexer for data type BOOL. Unlike AX_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AX_AUI_MUX_5 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (AX adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (AX adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (AX adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (AX adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (AX adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1...IN5:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AX_AUI_MUX`) – shared across all port counts of this block family (AX_AUI_MUX_2 … AX_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 Boolean value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AX_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

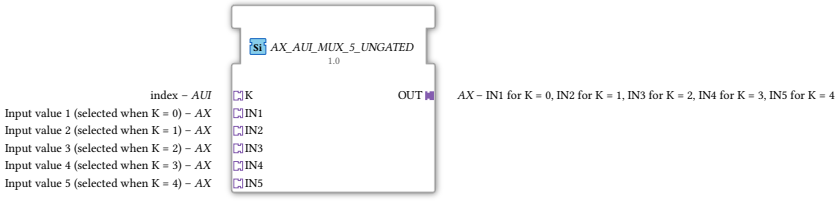
- `AX_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AX_AUI_MUX_5` carries the multiplexer logic of the `AX_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AX_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AX_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AX_AUI_MUX_5_UNGATED is the adapter-based variant of the generic multiplexer for data type BOOL. Unlike AX_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AX_AUI_MUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (AX adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (AX adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (AX adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (AX adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (AX adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AX adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN5:

1. The current value of K.D1 determines which of the 5 input adapters (IN1 … IN5) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AX_AUI_MUX) – shared across all port counts of this block family (AX_AUI_MUX_2 … AX_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN5) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 Boolean value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AX_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AX_AUI_MUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AX_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AX_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

BOOL

🎧 Podcast

- George Boole: How an autodidact laid the foundation of our digital world with 0 and 1

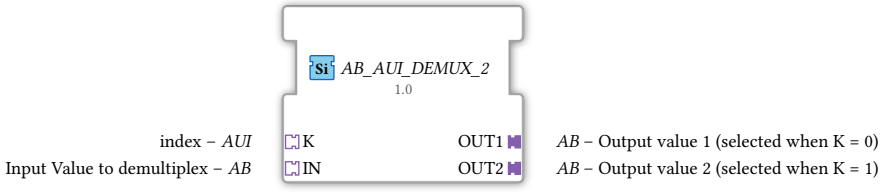
🛠️ Related exercises

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AB_AUI_DEMUX_2



• – ○ • – ○ • – ○ •

Introduction

`AB_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AB_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index

be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AB_AUI_DEMUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AB adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AB adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 2 output adapters (OUT1 … OUT2) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AB_AUI_DEMUX) – shared across all port counts of this block family (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)

- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AB_DEMUX_2 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

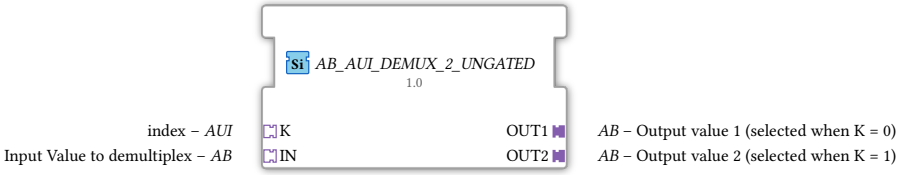
- **AB_AUI_DEMUX_2_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AB_AUI_DEMUX_2 carries the demultiplexer logic of the AB_DEMUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_AUI_DEMUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_AUI_DEMUX_2_UNGATED is the adapter-based variant of the generic demultiplexer for data type BYTE. Unlike AB_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AB_AUI_DEMUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AB adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AB adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_DEMUX`) – shared across all port counts of this block family (AB_AUI_DEMUX_2_UNGATED … AB_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AB_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

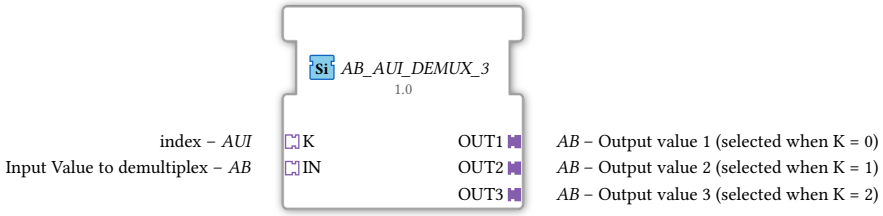
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AB_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AB_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AB_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AB_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AB_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AB_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AB adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AB adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AB adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_DEMUX`) – shared across all port counts of this block family (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AB_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

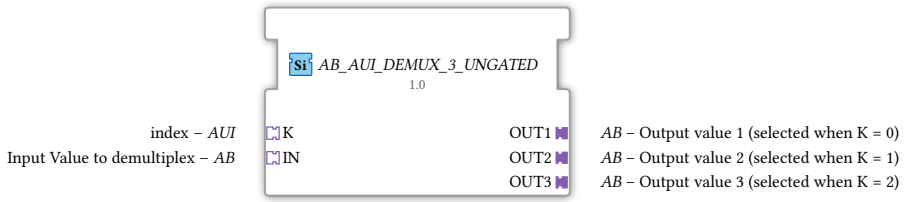
- **`AB_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AB_AUI_DEMUX_3` carries the demultiplexer logic of the `AB_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AB_AUI_DEMUX_3_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AB_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AB_AUI_DEMUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AB adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AB adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AB adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_DEMUX`) – shared across all port counts of this block family (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AB_DEMUX_3 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

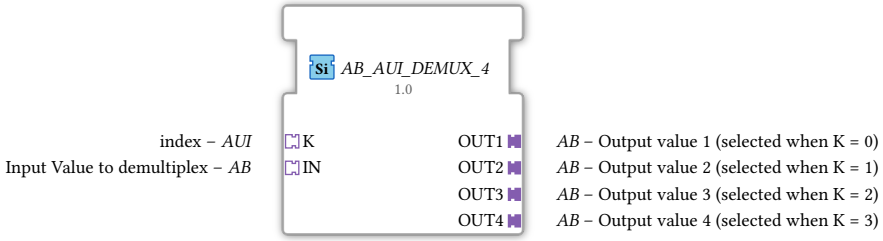
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **AB_AUI_DEMUX_3**: The gated variant – updates the output only on an actual value change.

Conclusion

AB_AUI_DEMUX_3_UNGATED carries the demultiplexer logic of the AB_DEMUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

AB_AUI_DEMUX_4 is the adapter-based variant of the generic demultiplexer for data type BYTE. Unlike AB_DEMUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AB_AUI_DEMUX_4 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AB adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AB adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AB adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AB adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_DEMUX`) – shared across all port counts of this block family (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AB_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

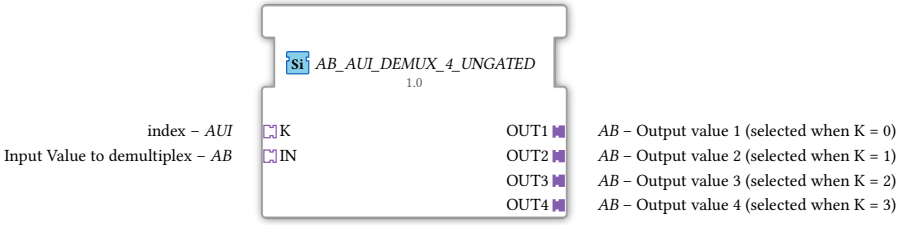
- **`AB_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AB_AUI_DEMUX_4` carries the demultiplexer logic of the `AB_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AB_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AB_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AB_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AB adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AB adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AB adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AB adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see “Change Detection” below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn’t changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_DEMUX`) – shared across all port counts of this block family (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AB_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

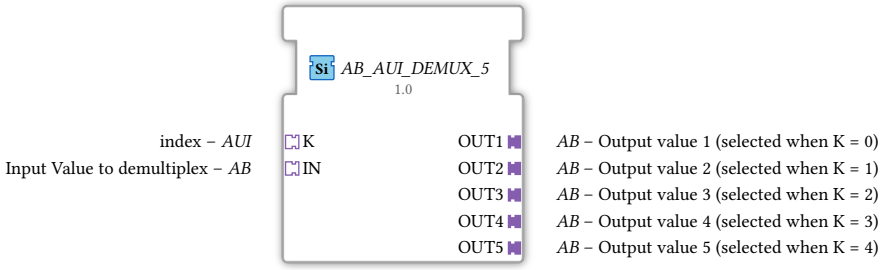
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AB_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AB_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AB_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AB_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AB_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AB_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AB adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AB adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AB adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AB adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AB adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_DEMUX`) – shared across all port counts of this block family (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AB_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

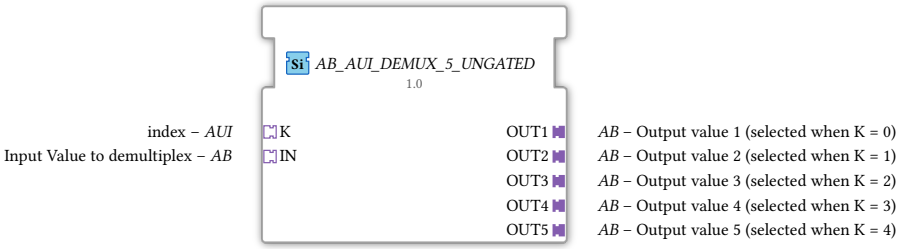
- **`AB_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AB_AUI_DEMUX_5 carries the demultiplexer logic of the AB_DEMUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_AUI_DEMUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_AUI_DEMUX_5_UNGATED is the adapter-based variant of the generic demultiplexer for data type BYTE. Unlike AB_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AB_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AB adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AB adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AB adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AB adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AB adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_DEMUX`) – shared across all port counts of this block family (AB_AUI_DEMUX_2 … AB_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AB_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

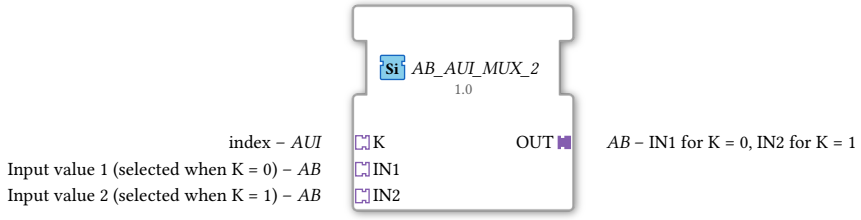
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AB_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AB_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AB_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AB_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `BYTE`. Unlike `AB_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its

own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AB_AUI_MUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 2, passed through to OUT when the index is K = 0 (AB adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to OUT when the index is K = 1 (AB adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN2:

1. The current value of K.D1 determines which of the 2 input adapters (IN1 … IN2) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AB_AUI_MUX) – shared across all port counts of this block family (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN2) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AB_MUX_2 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

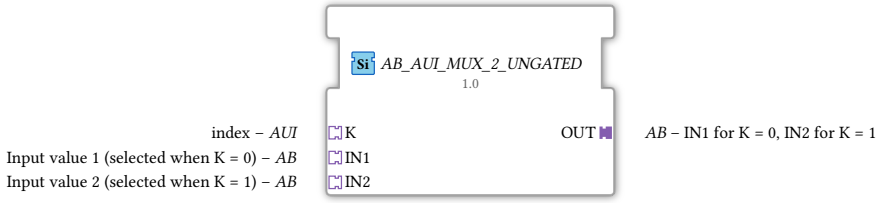
- **AB_AUI_MUX_2_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AB_AUI_MUX_2 carries the multiplexer logic of the AB_MUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AB_AUI_MUX_2_UNGATED is the adapter-based variant of the generic multiplexer for data type BYTE. Unlike AB_MUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AB_AUI_MUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.

- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AB adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AB adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_MUX`) – shared across all port counts of this block family (AB_AUI_MUX_2_UNGATED … AB_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AB_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AB_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AB_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AB_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_MUX_3



• – ○ • – ○ • – ○ •

Introduction

AB_AUI_MUX_3 is the adapter-based variant of the generic multiplexer for data type BYTE. Unlike AB_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AB_AUI_MUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (AB adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (AB adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (AB adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AB_AUI_MUX) – shared across all port counts of this block family (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 byte value (8-bit bit pattern) sources through a centrally managed adapter index

- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AB_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

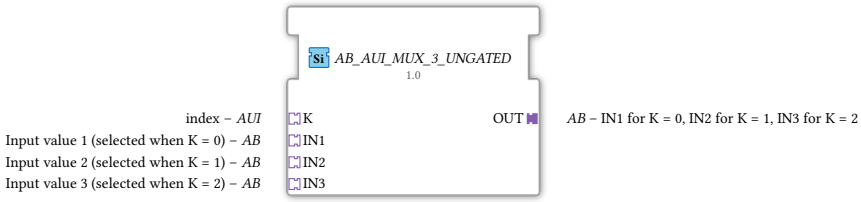
- **`AB_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AB_AUI_MUX_3` carries the multiplexer logic of the `AB_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_AUI_MUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AB_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `BYTE`. Unlike `AB_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AB_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AB adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AB adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AB adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_MUX`) – shared across all port counts of this block family (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AB_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

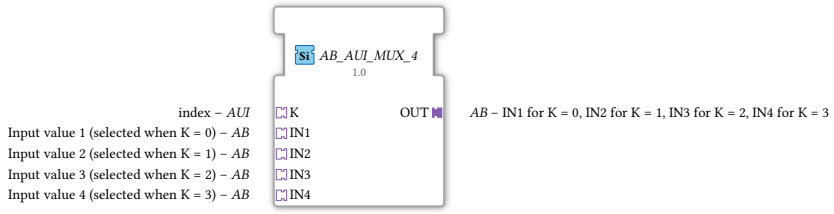
Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AB_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AB_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AB_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AB_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `BYTE`. Unlike `AB_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AB_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AB adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AB adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AB adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AB adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_MUX`) – shared across all port counts of this block family (AB_AUI_MUX_2 … AB_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AB_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

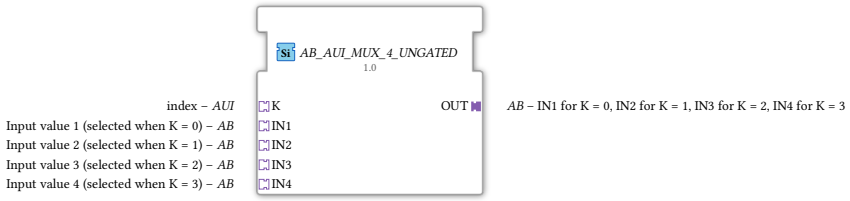
- **`AB_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AB_AUI_MUX_4` carries the multiplexer logic of the `AB_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AB_AUI_MUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AB_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `BYTE`. Unlike `AB_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AB_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AB adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AB adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AB adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AB adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_MUX`) – shared across all port counts of this block family (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AB_MUX_4 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

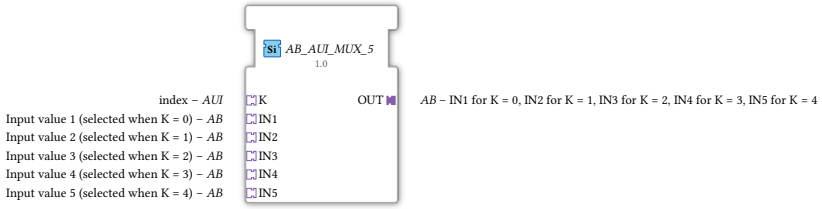
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **AB_AUI_MUX_4**: The gated variant – updates the output only on an actual value change.

Conclusion

AB_AUI_MUX_4_UNGATED carries the multiplexer logic of the AB_MUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

AB_AUI_MUX_5 is the adapter-based variant of the generic multiplexer for data type BYTE. Unlike AB_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AB_AUI_MUX_5 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (AB adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (AB adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (AB adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (AB adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (AB adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1...IN5:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AB_AUI_MUX`) – shared across all port counts of this block family (AB_AUI_MUX_2 … AB_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AB_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

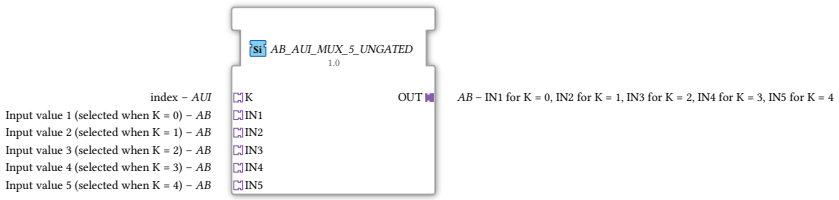
- `AB_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AB_AUI_MUX_5` carries the multiplexer logic of the `AB_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AB_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AB_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AB_AUI_MUX_5_UNGATED is the adapter-based variant of the generic multiplexer for data type BYTE. Unlike AB_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AB_AUI_MUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (AB adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (AB adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (AB adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (AB adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (AB adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AB adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN5:

1. The current value of K.D1 determines which of the 5 input adapters (IN1 … IN5) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AB_AUI_MUX) – shared across all port counts of this block family (AB_AUI_MUX_2 … AB_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN5) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AB_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AB_AUI_MUX_5`**: The gated variant – updates the output only on an actual value change.

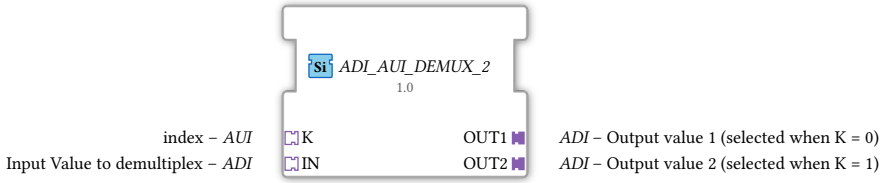
Conclusion

`AB_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AB_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ADI_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`ADI_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `DINT`. Unlike `ADI_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.

- **IN** (Socket): input adapter carrying the 32-bit signed integer to forward, routed to exactly one of the outputs (ADI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (ADI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_DEMUX`) – shared across all port counts of this block family (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit signed integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ADI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

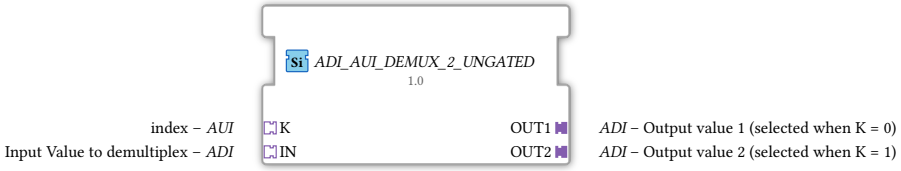
- **`ADI_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ADI_AUI_DEMUX_2` carries the demultiplexer logic of the `ADI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ADI_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `DINT`. Unlike `ADI_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit signed integer to forward, routed to exactly one of the outputs (ADI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (ADI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_DEMUX`) – shared across all port counts of this block family (ADI_AUI_DEMUX_2_UNGATED … ADI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit signed integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ADI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ADI_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ADI_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `ADI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_DEMUX_3



• – ○ • – ○ • – ○ •

Introduction

`ADI_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `DINT`. Unlike `ADI_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit signed integer to forward, routed to exactly one of the outputs (ADI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (ADI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (ADI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_DEMUX`) – shared across all port counts of this block family (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit signed integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ADI_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

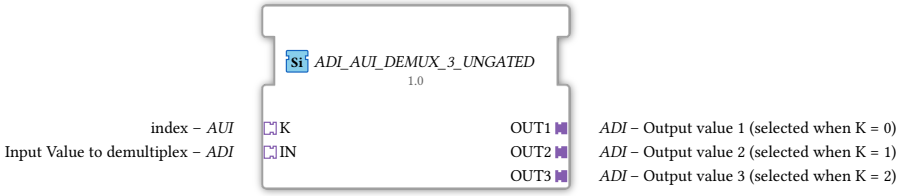
- **`ADI_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ADI_AUI_DEMUX_3` carries the demultiplexer logic of the `ADI_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_DEMUX_3_UNGATED

i UNGATED variant: This block is the ungated version of ADI_AUI_DEMUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

ADI_AUI_DEMUX_3_UNGATED is the adapter-based variant of the generic demultiplexer for data type DINT. Unlike ADI_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ADI_AUI_DEMUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 32-bit signed integer to forward, routed to exactly one of the outputs (ADI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (ADI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (ADI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 3 output adapters (OUT1 … OUT3) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_ADI_AUI_DEMUX) – shared across all port counts of this block family (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) ⟶ determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit signed integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with ADI_DEMUX_3 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

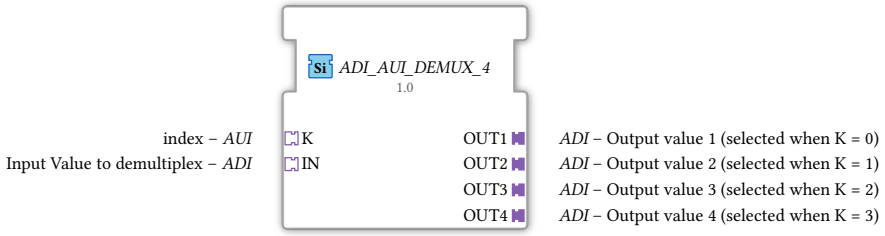
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **ADI_AUI_DEMUX_3**: The gated variant – updates the output only on an actual value change.

Conclusion

ADI_AUI_DEMUX_3_UNGATED carries the demultiplexer logic of the ADI_DEMUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_DEMUX_4



• – ○ • – ○ • – ○ •

Introduction

ADI_AUI_DEMUX_4 is the adapter-based variant of the generic demultiplexer for data type DINT. Unlike ADI_DEMUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit signed integer to forward, routed to exactly one of the outputs (ADI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (ADI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (ADI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (ADI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_DEMUX`) – shared across all port counts of this block family (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit signed integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ADI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

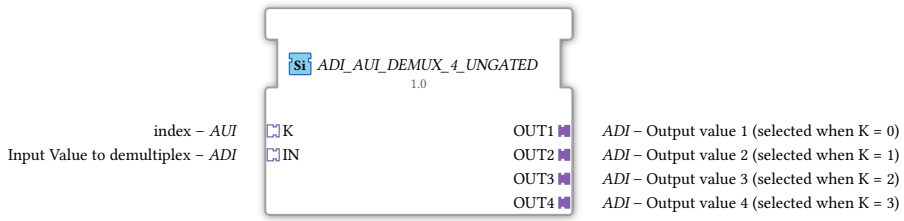
- **`ADI_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ADI_AUI_DEMUX_4` carries the demultiplexer logic of the `ADI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ADI_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `DINT`. Unlike `ADI_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.

- **IN** (Socket): input adapter carrying the 32-bit signed integer to forward, routed to exactly one of the outputs (ADI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (ADI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (ADI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (ADI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_DEMUX`) – shared across all port counts of this block family (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit signed integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ADI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ADI_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ADI_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `ADI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`ADI_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `DINT`. Unlike `ADI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit signed integer to forward, routed to exactly one of the outputs (ADI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (ADI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (ADI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (ADI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (ADI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_DEMUX`) – shared across all port counts of this block family (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit signed integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ADI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

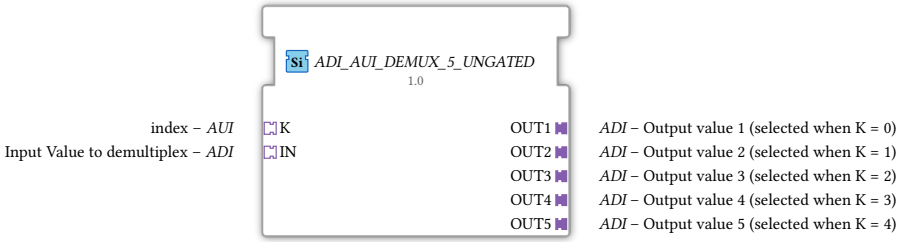
- **`ADI_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ADI_AUI_DEMUX_5` carries the demultiplexer logic of the `ADI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_AUI_DEMUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`ADI_AUI_DEMUX_5_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `DINT`. Unlike `ADI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K

data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_DEMUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit signed integer to forward, routed to exactly one of the outputs (ADI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (ADI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (ADI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (ADI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (ADI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_DEMUX`) – shared across all port counts of this block family (ADI_AUI_DEMUX_2 … ADI_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit signed integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ADI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

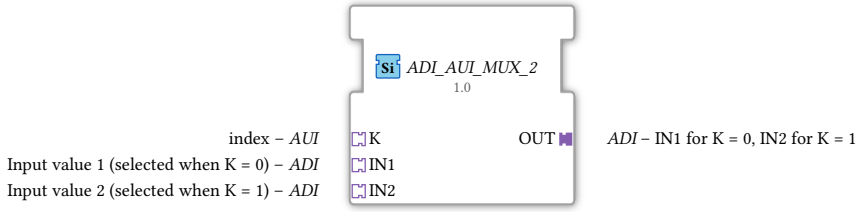
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ADI_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ADI_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `ADI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_MUX_2



• – ○ • – ○ • – ○ •

Introduction

`ADI_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `DINT`. Unlike `ADI_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (ADI adapter type).

- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (ADI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_MUX`) – shared across all port counts of this block family (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 32-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ADI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

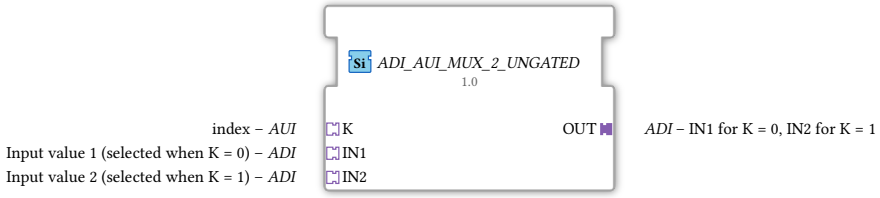
- **`ADI_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ADI_AUI_MUX_2` carries the multiplexer logic of the `ADI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of ADI_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ADI_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `DINT`. Unlike `ADI_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (ADI adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (ADI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_MUX`) – shared across all port counts of this block family (ADI_AUI_MUX_2_UNGATED … ADI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 32-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ADI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

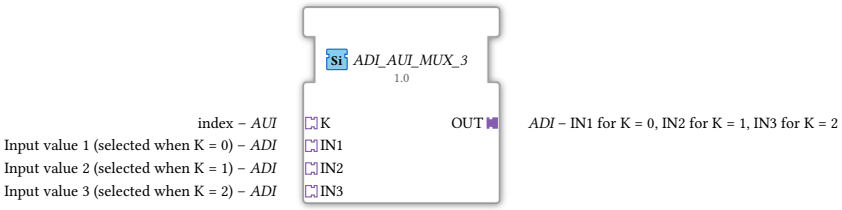
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`ADI_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ADI_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `ADI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`ADI_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `DINT`. Unlike `ADI_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (ADI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (ADI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (ADI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_MUX`) – shared across all port counts of this block family (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 32-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ADI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`ADI_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

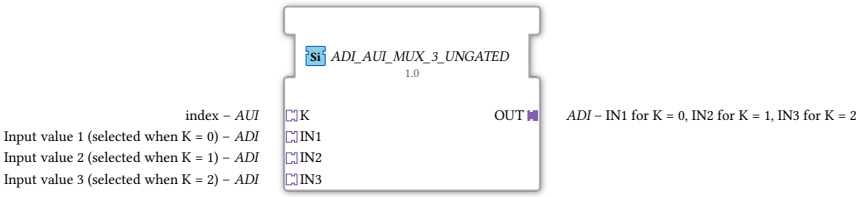
Conclusion

`ADI_AUI_MUX_3` carries the multiplexer logic of the `ADI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of ADI_AUI_MUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change

(e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ADI_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `DINT`. Unlike `ADI_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (ADI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (ADI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (ADI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_MUX`) – shared across all port counts of this block family (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 32-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ADI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`ADI_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ADI_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `ADI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`ADI_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `DINT`. Unlike `ADI_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (ADI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (ADI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (ADI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (ADI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_MUX`) – shared across all port counts of this block family (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 32-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ADI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

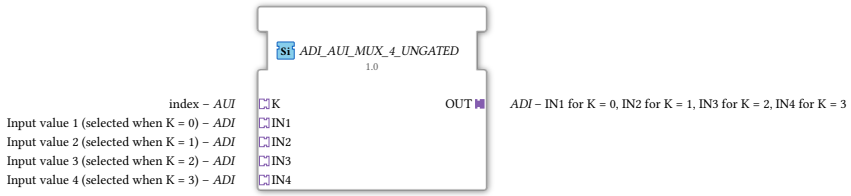
- **`ADI_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ADI_AUI_MUX_4` carries the multiplexer logic of the `ADI_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_AUI_MUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`ADI_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `DINT`. Unlike `ADI_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (ADI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (ADI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (ADI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (ADI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_MUX`) – shared across all port counts of this block family (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 32-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ADI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`ADI_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ADI_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `ADI_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`ADI_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `DINT`. Unlike `ADI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (ADI adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (ADI adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (ADI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (ADI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (ADI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_MUX`) – shared across all port counts of this block family (ADI_AUI_MUX_2 … ADI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 32-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ADI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

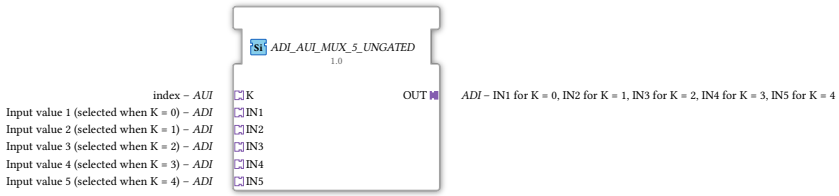
- **`ADI_AUI_MUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ADI_AUI_MUX_5` carries the multiplexer logic of the `ADI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ADI_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ADI_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`ADI_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `DINT`. Unlike `ADI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ADI_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (ADI adapter type).

- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (ADI adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (ADI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (ADI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (ADI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ADI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ADI_AUI_MUX`) – shared across all port counts of this block family (ADI_AUI_MUX_2 … ADI_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 32-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ADI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- `ADI_AUI_MUX_5`: The gated variant – updates the output only on an actual value change.

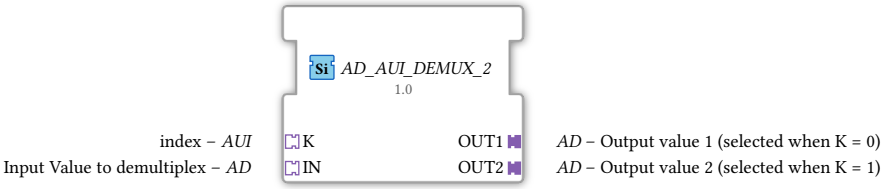
Conclusion

`ADI_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `ADI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AD_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AD_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `DWORD`. Unlike `AD_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AD_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit bit pattern to forward, routed to exactly one of the outputs (AD adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AD adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_DEMUX`) – shared across all port counts of this block family (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit bit pattern to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AD_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

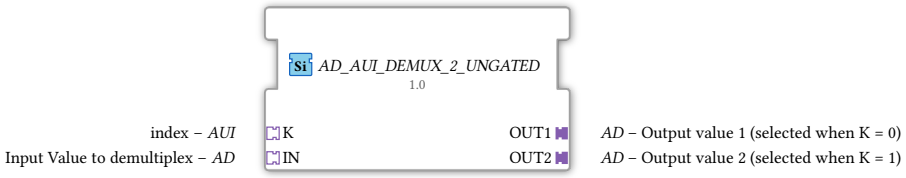
- **`AD_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AD_AUI_DEMUX_2` carries the demultiplexer logic of the `AD_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AD_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `DWORD`. Unlike `AD_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AD_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit bit pattern to forward, routed to exactly one of the outputs (AD adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AD adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_DEMUX`) – shared across all port counts of this block family (AD_AUI_DEMUX_2_UNGATED … AD_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit bit pattern to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AD_DEMUX_2 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

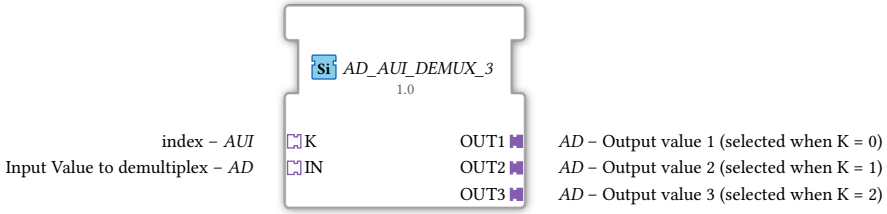
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **AD_AUI_DEMUX_2**: The gated variant – updates the output only on an actual value change.

Conclusion

AD_AUI_DEMUX_2_UNGATED carries the demultiplexer logic of the AD_DEMUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

AD_AUI_DEMUX_3 is the adapter-based variant of the generic demultiplexer for data type DWORD. Unlike AD_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AD_AUI_DEMUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 32-bit bit pattern to forward, routed to exactly one of the outputs (AD adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AD adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AD adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_DEMUX`) – shared across all port counts of this block family (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit bit pattern to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖ Comparison with Similar Blocks

Compare with `AD_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

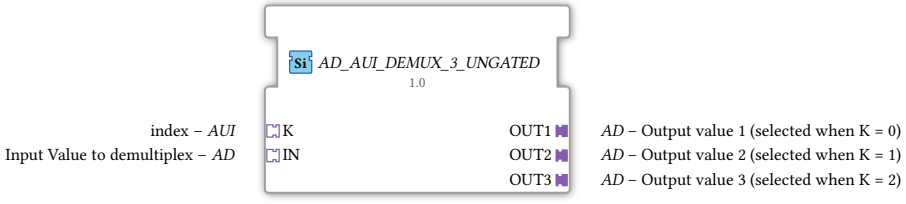
- **`AD_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AD_AUI_DEMUX_3` carries the demultiplexer logic of the `AD_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_AUI_DEMUX_3_UNGATED is the adapter-based variant of the generic demultiplexer for data type DWORD. Unlike AD_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AD_AUI_DEMUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 32-bit bit pattern to forward, routed to exactly one of the outputs (AD adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AD adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AD adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 3 output adapters (OUT1 … OUT3) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AD_AUI_DEMUX) – shared across all port counts of this block family (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit bit pattern to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖ Comparison with Similar Blocks

Compare with `AD_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

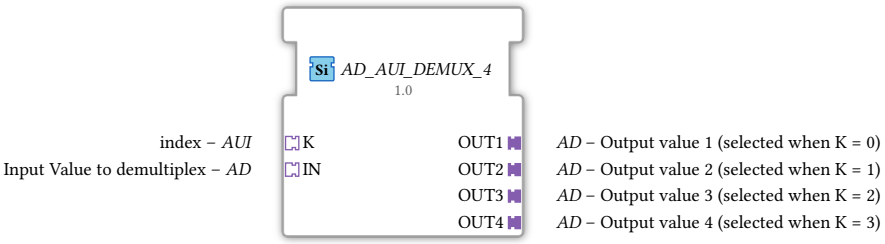
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AD_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AD_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `AD_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AD_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `DWORD`. Unlike `AD_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AD_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.

- **IN** (Socket): input adapter carrying the 32-bit bit pattern to forward, routed to exactly one of the outputs (AD adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AD adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AD adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AD adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_DEMUX`) – shared across all port counts of this block family (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit bit pattern to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AD_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

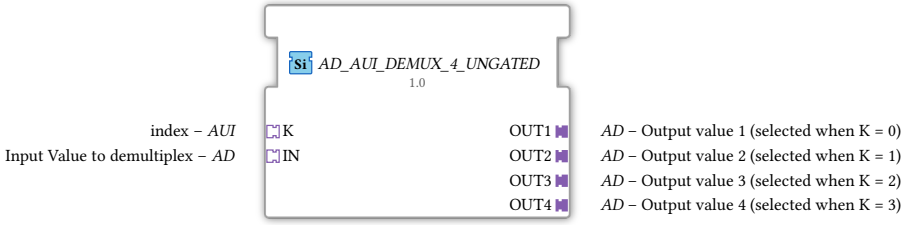
- `AD_AUI_DEMUX_4_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AD_AUI_DEMUX_4` carries the demultiplexer logic of the `AD_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AD_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `DWORD`. Unlike `AD_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AD_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit bit pattern to forward, routed to exactly one of the outputs (AD adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AD adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AD adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AD adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_DEMUX`) – shared across all port counts of this block family (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit bit pattern to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AD_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

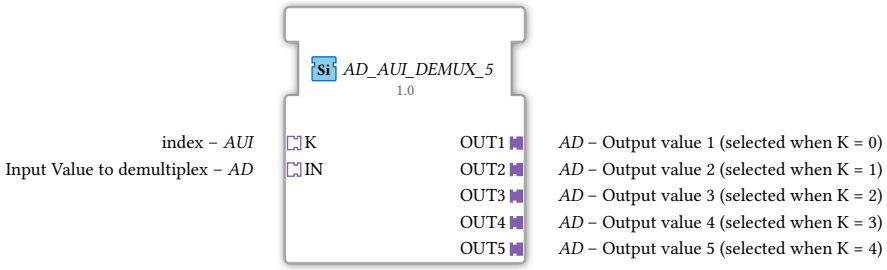
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AD_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AD_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AD_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AD_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `DWORD`. Unlike `AD_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AD_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit bit pattern to forward, routed to exactly one of the outputs (AD adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AD adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AD adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AD adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AD adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_DEMUX`) – shared across all port counts of this block family (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit bit pattern to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AD_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

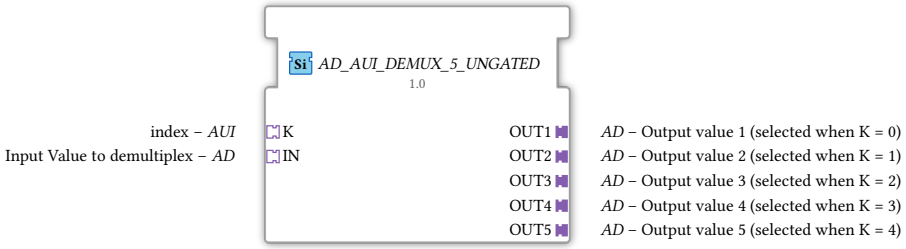
- **`AD_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AD_AUI_DEMUX_5 carries the demultiplexer logic of the AD_DEMUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_AUI_DEMUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AD_AUI_DEMUX_5_UNGATED is the adapter-based variant of the generic demultiplexer for data type DWORD. Unlike AD_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AD_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 32-bit bit pattern to forward, routed to exactly one of the outputs (AD adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AD adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AD adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AD adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AD adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_DEMUX`) – shared across all port counts of this block family (AD_AUI_DEMUX_2 … AD_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit bit pattern to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖ Comparison with Similar Blocks

Compare with `AD_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

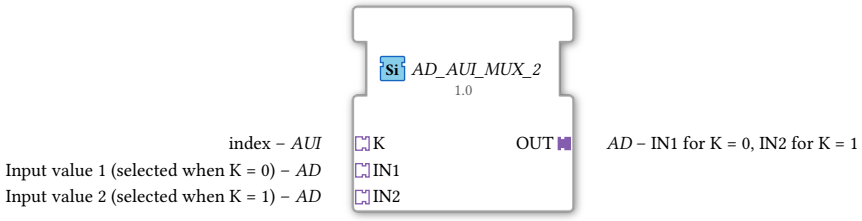
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AD_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AD_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AD_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AD_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `DWORD`. Unlike `AD_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index

be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AD_AUI_MUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 2, passed through to OUT when the index is K = 0 (AD adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to OUT when the index is K = 1 (AD adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN2:

1. The current value of K.D1 determines which of the 2 input adapters (IN1 … IN2) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AD_AUI_MUX) – shared across all port counts of this block family (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN2) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 32-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AD_MUX_2 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

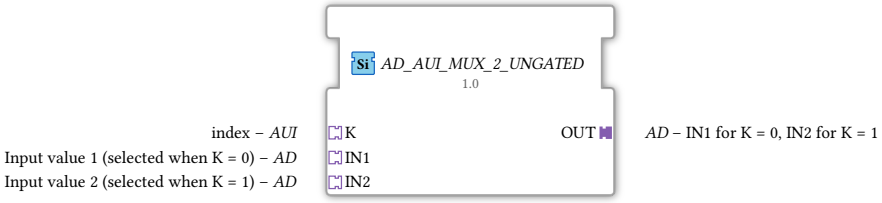
- **AD_AUI_MUX_2_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AD_AUI_MUX_2 carries the multiplexer logic of the AD_MUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AD_AUI_MUX_2_UNGATED is the adapter-based variant of the generic multiplexer for data type DWORD. Unlike AD_MUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AD_AUI_MUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.

- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AD adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AD adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` ... `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_MUX`) – shared across all port counts of this block family (AD_AUI_MUX_2_UNGATED ... AD_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 32-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AD_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

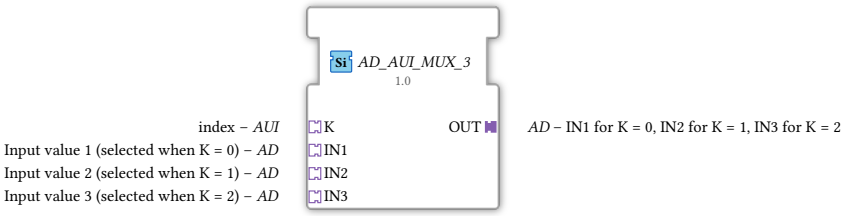
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AD_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AD_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AD_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

AD_AUI_MUX_3 is the adapter-based variant of the generic multiplexer for data type DWORD. Unlike AD_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AD_AUI_MUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (AD adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (AD adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (AD adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AD_AUI_MUX) – shared across all port counts of this block family (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) → read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 32-bit bit pattern sources through a centrally managed adapter index

- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AD_MUX_3 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

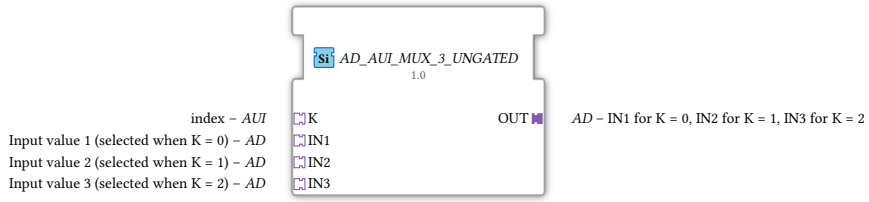
- **AD_AUI_MUX_3_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AD_AUI_MUX_3 carries the multiplexer logic of the AD_MUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_AUI_MUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AD_AUI_MUX_3_UNGATED is the adapter-based variant of the generic multiplexer for data type DWORD. Unlike AD_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AD_AUI_MUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AD adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AD adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AD adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_MUX`) – shared across all port counts of this block family (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 32-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AD_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

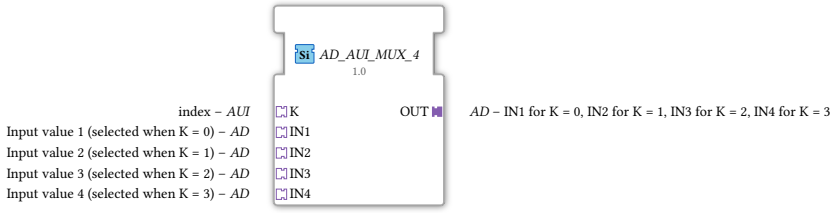
Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AD_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AD_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AD_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AD_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `DWORD`. Unlike `AD_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AD_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AD adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AD adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AD adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AD adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_MUX`) – shared across all port counts of this block family (AD_AUI_MUX_2 … AD_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 32-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AD_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

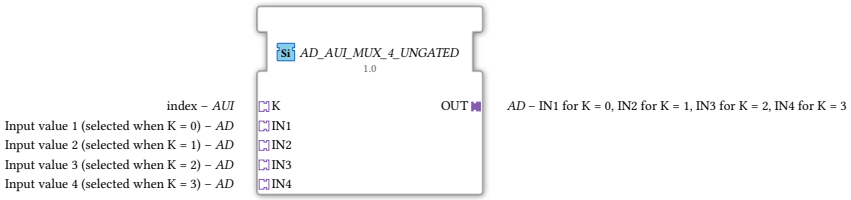
- **`AD_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AD_AUI_MUX_4` carries the multiplexer logic of the `AD_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AD_AUI_MUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AD_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `DWORD`. Unlike `AD_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AD_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AD adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AD adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AD adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AD adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_MUX`) – shared across all port counts of this block family (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 32-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AD_MUX_4 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

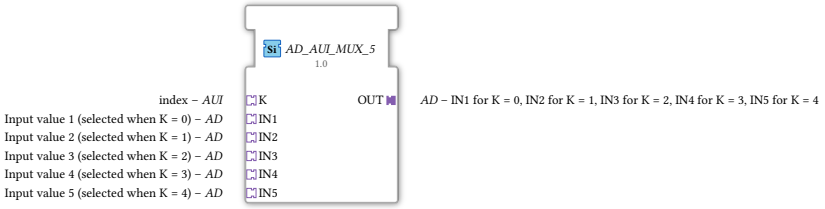
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **AD_AUI_MUX_4**: The gated variant – updates the output only on an actual value change.

Conclusion

AD_AUI_MUX_4_UNGATED carries the multiplexer logic of the AD_MUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

AD_AUI_MUX_5 is the adapter-based variant of the generic multiplexer for data type DWORD. Unlike AD_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AD_AUI_MUX_5 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (AD adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (AD adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (AD adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (AD adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (AD adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_MUX`) – shared across all port counts of this block family (AD_AUI_MUX_2 … AD_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 32-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AD_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

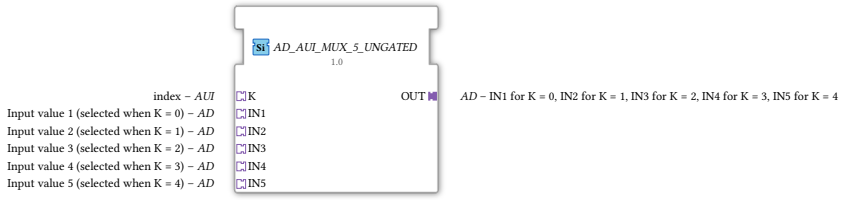
- `AD_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AD_AUI_MUX_5` carries the multiplexer logic of the `AD_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AD_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AD_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AD_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `DWORD`. Unlike `AD_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AD_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AD adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AD adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AD adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AD adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AD adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AD adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AD_AUI_MUX`) – shared across all port counts of this block family (AD_AUI_MUX_2 … AD_AUI_MUX_5_UNGATED).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 32-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AD_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- `AD_AUI_MUX_5`: The gated variant – updates the output only on an actual value change.

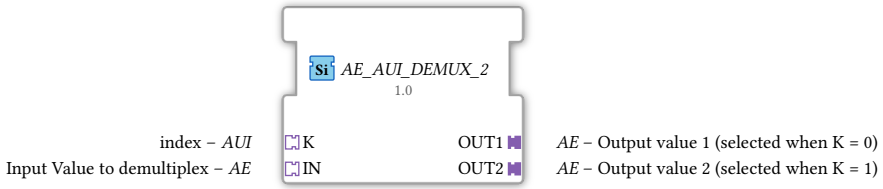
Conclusion

`AD_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AD_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AE_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AE_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `EVENT`. Unlike `AE_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AE_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the event to forward, routed to exactly one of the outputs (AE adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AE adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AE adapter type).

Functionality

`AE_AUI_DEMUX_2` receives an event on socket **IN** and forwards it to exactly one of the 2 output plugs `OUT1` … `OUT2`. Which output is written is decided by the index arriving on the adapter socket **K** (type `AUI`): as soon as the `AUI` plug connected to K fires its internal `E1` event carrying the data value `D1`, `AE_AUI_DEMUX_2` interprets that value as a 0-based index (`0` … `1`), writes the forwarded event to the corresponding `OUT` plug and triggers that plug's adapter event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AE_AUI_DEMUX`) – shared across all port counts of this block family (AE_AUI_DEMUX_2 … AE_AUI_DEMUX_5).
- The `AE` adapter carries only an event, no data value – `AE_AUI_DEMUX_2` therefore switches pure event pulses, not payload data.

State Overview

The block is stateless with respect to any sequencing logic: it waits for the `E1` event on socket **K**, evaluates the index carried with it once it arrives and updates the affected adapter plug. No state beyond the most recently written adapter value is kept between calls.

Application Scenarios

- Distributing a control event to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

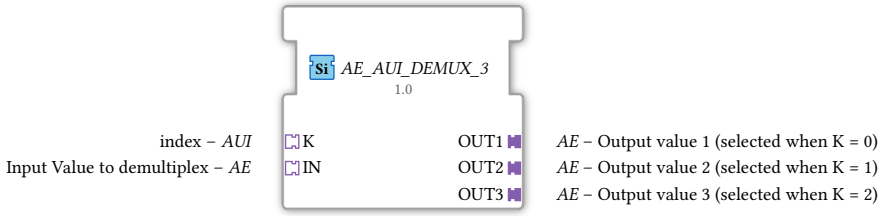
Compare with `AE_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

Conclusion

`AE_AUI_DEMUX_2` carries the demultiplexer logic of the `AE_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AE_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AE_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `EVENT`. Unlike `AE_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AE_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the event to forward, routed to exactly one of the outputs (AE adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AE adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AE adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AE adapter type).

Functionality

`AE_AUI_DEMUX_3` receives an event on socket **IN** and forwards it to exactly one of the 3 output plugs `OUT1` … `OUT3`. Which output is written is decided by the index arriving on the adapter socket **K** (type `AUI`): as soon as the `AUI` plug connected to K fires its internal `E1` event carrying the data value `D1`, `AE_AUI_DEMUX_3` interprets that value as a 0-based index (`0` … `2`), writes the forwarded event to the corresponding `OUT` plug and triggers that plug's adapter event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AE_AUI_DEMUX`) – shared across all port counts of this block family (AE_AUI_DEMUX_2 … AE_AUI_DEMUX_5).
- The `AE` adapter carries only an event, no data value – `AE_AUI_DEMUX_3` therefore switches pure event pulses, not payload data.

State Overview

The block is stateless with respect to any sequencing logic: it waits for the `E1` event on socket **K**, evaluates the index carried with it once it arrives and updates the affected adapter plug. No state beyond the most recently written adapter value is kept between calls.

Application Scenarios

- Distributing a control event to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

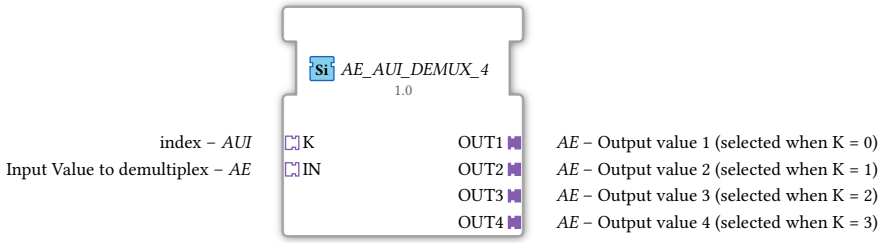
Compare with `AE_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

Conclusion

`AE_AUI_DEMUX_3` carries the demultiplexer logic of the `AE_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AE_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AE_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `EVENT`. Unlike `AE_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AE_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the event to forward, routed to exactly one of the outputs (AE adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AE adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AE adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AE adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AE adapter type).

Functionality

`AE_AUI_DEMUX_4` receives an event on socket **IN** and forwards it to exactly one of the 4 output plugs `OUT1` … `OUT4`. Which output is written is decided by the index arriving on the adapter socket **K** (type `AUI`): as soon as the `AUI` plug connected to K fires its internal `E1` event carrying the data value `D1`, `AE_AUI_DEMUX_4` interprets that value as a 0-based index (`0` … `3`), writes the forwarded event to the corresponding `OUT` plug and triggers that plug's adapter event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AE_AUI_DEMUX`) – shared across all port counts of this block family (AE_AUI_DEMUX_2 … AE_AUI_DEMUX_5).
- The `AE` adapter carries only an event, no data value – `AE_AUI_DEMUX_4` therefore switches pure event pulses, not payload data.

State Overview

The block is stateless with respect to any sequencing logic: it waits for the `E1` event on socket **K**, evaluates the index carried with it once it arrives and updates the affected adapter plug. No state beyond the most recently written adapter value is kept between calls.

Application Scenarios

- Distributing a control event to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

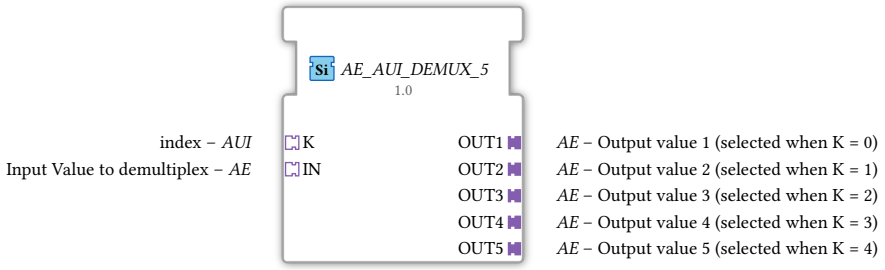
Compare with `AE_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

Conclusion

`AE_AUI_DEMUX_4` carries the demultiplexer logic of the `AE_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AE_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

AE_AUI_DEMUX_5 is the adapter-based variant of the generic demultiplexer for data type EVENT. Unlike AE_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AE_AUI_DEMUX_5 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the event to forward, routed to exactly one of the outputs (AE adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AE adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AE adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AE adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AE adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AE adapter type).

Functionality

AE_AUI_DEMUX_5 receives an event on socket **IN** and forwards it to exactly one of the 5 output plugs OUT1 … OUT5. Which output is written is decided by the index arriving on the adapter socket **K** (type AUI): as soon as the AUI plug connected to K fires its internal E1 event carrying the data value D1, AE_AUI_DEMUX_5 interprets that value as a 0-based index (0 … 4), writes the forwarded event to the corresponding OUT plug and triggers that plug's adapter event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AE_AUI_DEMUX) – shared across all port counts of this block family (AE_AUI_DEMUX_2 … AE_AUI_DEMUX_5).
- The AE adapter carries only an event, no data value – AE_AUI_DEMUX_5 therefore switches pure event pulses, not payload data.

State Overview

The block is stateless with respect to any sequencing logic: it waits for the E1 event on socket **K**, evaluates the index carried with it once it arrives and updates the affected adapter plug. No state beyond the most recently written adapter value is kept between calls.

Application Scenarios

- Distributing a control event to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)

- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

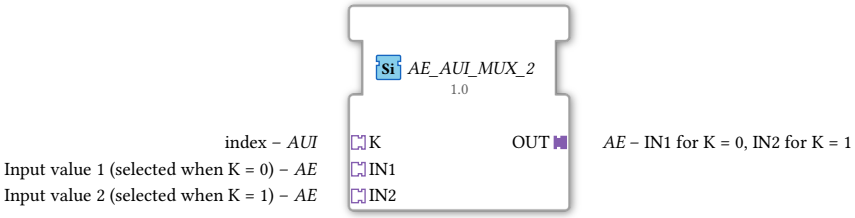
Compare with `AE_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

Conclusion

`AE_AUI_DEMUX_5` carries the demultiplexer logic of the `AE_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AE_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AE_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `EVENT`. Unlike `AE_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AE_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AE adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AE adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AE adapter type).

Functionality

`AE_AUI_MUX_2` selects one of the 2 input sockets `IN1` … `IN2` via the adapter socket **K** (type `AUI`) and forwards its value to the single output plug `OUT`. As soon as the `AUI` plug connected to K fires its internal

E1 event carrying the data value D1, AE_AUI_MUX_2 interprets that value as a 0-based index (0 … 1), copies the selected event to OUT and triggers its adapter event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AE_AUI_MUX) – shared across all port counts of this block family (AE_AUI_MUX_2 … AE_AUI_MUX_5).
- The AE adapter carries only an event, no data value – AE_AUI_MUX_2 therefore switches pure event pulses, not payload data.

State Overview

The block is stateless with respect to any sequencing logic: it waits for the E1 event on socket **K**, evaluates the index carried with it once it arrives and updates the affected adapter plug. No state beyond the most recently written adapter value is kept between calls.

Application Scenarios

- Choosing between up to 2 event sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AE_MUX_2 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

Conclusion

AE_AUI_MUX_2 carries the multiplexer logic of the AE_MUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AE_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

AE_AUI_MUX_3 is the adapter-based variant of the generic multiplexer for data type EVENT. Unlike AE_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AE_AUI_MUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AE adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AE adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AE adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AE adapter type).

Functionality

`AE_AUI_MUX_3` selects one of the 3 input sockets `IN1` … `IN3` via the adapter socket **K** (type `AUI`) and forwards its value to the single output plug `OUT`. As soon as the `AUI` plug connected to K fires its internal `E1` event carrying the data value `D1`, `AE_AUI_MUX_3` interprets that value as a 0-based index (`0` … `2`), copies the selected event to `OUT` and triggers its adapter event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AE_AUI_MUX`) – shared across all port counts of this block family (AE_AUI_MUX_2 … AE_AUI_MUX_5).
- The `AE` adapter carries only an event, no data value – `AE_AUI_MUX_3` therefore switches pure event pulses, not payload data.

State Overview

The block is stateless with respect to any sequencing logic: it waits for the `E1` event on socket **K**, evaluates the index carried with it once it arrives and updates the affected adapter plug. No state beyond the most recently written adapter value is kept between calls.

Application Scenarios

- Choosing between up to 3 event sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AE_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

Conclusion

`AE_AUI_MUX_3` carries the multiplexer logic of the `AE_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AE_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AE_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `EVENT`. Unlike `AE_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AE_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AE adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AE adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AE adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AE adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AE adapter type).

Functionality

`AE_AUI_MUX_4` selects one of the 4 input sockets `IN1` … `IN4` via the adapter socket **K** (type `AUI`) and forwards its value to the single output plug `OUT`. As soon as the `AUI` plug connected to K fires its internal `E1` event carrying the data value `D1`, `AE_AUI_MUX_4` interprets that value as a 0-based index (`0` … `3`), copies the selected event to `OUT` and triggers its adapter event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AE_AUI_MUX`) – shared across all port counts of this block family (AE_AUI_MUX_2 … AE_AUI_MUX_5).
- The `AE` adapter carries only an event, no data value – `AE_AUI_MUX_4` therefore switches pure event pulses, not payload data.

State Overview

The block is stateless with respect to any sequencing logic: it waits for the `E1` event on socket **K**, evaluates the index carried with it once it arrives and updates the affected adapter plug. No state beyond the most recently written adapter value is kept between calls.

Application Scenarios

- Choosing between up to 4 event sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

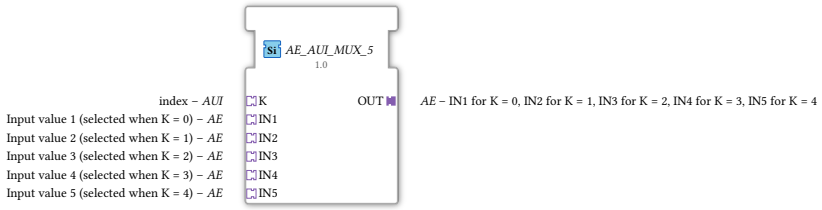
Compare with `AE_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

Conclusion

`AE_AUI_MUX_4` carries the multiplexer logic of the `AE_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AE_AUI_MUX_5



• — ○ • — ○ • — ○ •

Introduction

`AE_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `EVENT`. Unlike `AE_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AE_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AE adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AE adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AE adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AE adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AE adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AE adapter type).

Functionality

`AE_AUI_MUX_5` selects one of the 5 input sockets `IN1` … `IN5` via the adapter socket **K** (type `AUI`) and forwards its value to the single output plug `OUT`. As soon as the `AUI` plug connected to K fires its internal `E1` event carrying the data value `D1`, `AE_AUI_MUX_5` interprets that value as a 0-based index (`0` … `4`), copies the selected event to `OUT` and triggers its adapter event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AE_AUI_MUX`) – shared across all port counts of this block family (AE_AUI_MUX_2 … AE_AUI_MUX_5).
- The `AE` adapter carries only an event, no data value – `AE_AUI_MUX_5` therefore switches pure event pulses, not payload data.

State Overview

The block is stateless with respect to any sequencing logic: it waits for the `E1` event on socket **K**, evaluates the index carried with it once it arrives and updates the affected adapter plug. No state beyond the most recently written adapter value is kept between calls.

Application Scenarios

- Choosing between up to 5 event sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AE_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

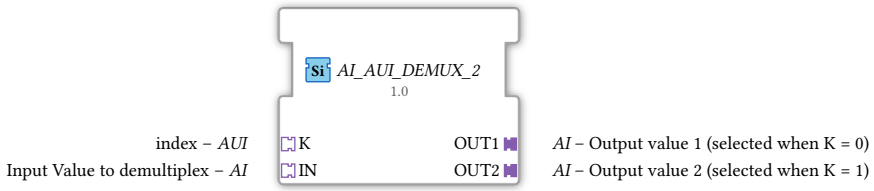
Conclusion

`AE_AUI_MUX_5` carries the multiplexer logic of the `AE_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AI_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

AI_AUI_DEMUX_2 is the adapter-based variant of the generic demultiplexer for data type INT. Unlike AI_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AI_AUI_DEMUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 16-bit signed integer to forward, routed to exactly one of the outputs (AI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 2 output adapters (OUT1 … OUT2) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AI_AUI_DEMUX) – shared across all port counts of this block family (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit signed integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

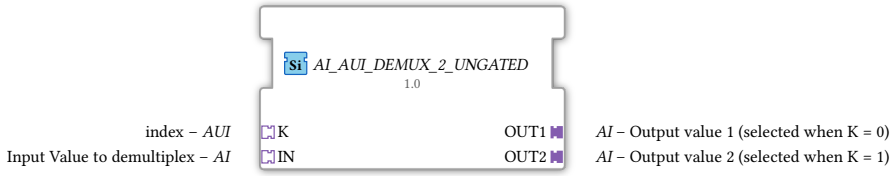
- **`AI_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AI_AUI_DEMUX_2` carries the demultiplexer logic of the `AI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AI_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `INT`. Unlike `AI_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit signed integer to forward, routed to exactly one of the outputs (AI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_DEMUX`) – shared across all port counts of this block family (AI_AUI_DEMUX_2_UNGATED … AI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit signed integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

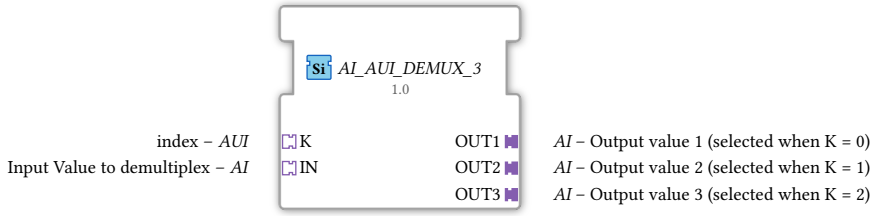
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AI_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AI_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AI_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `INT`. Unlike `AI_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit signed integer to forward, routed to exactly one of the outputs (AI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_DEMUX`) – shared across all port counts of this block family (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit signed integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AI_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

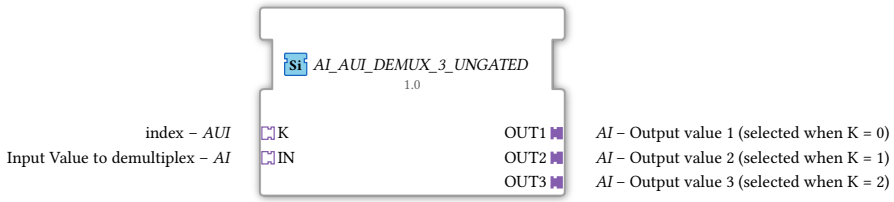
- **`AI_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AI_AUI_DEMUX_3` carries the demultiplexer logic of the `AI_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AI_AUI_DEMUX_3_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `INT`. Unlike `AI_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_DEMUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit signed integer to forward, routed to exactly one of the outputs (AI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_DEMUX`) – shared across all port counts of this block family (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit signed integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AI_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

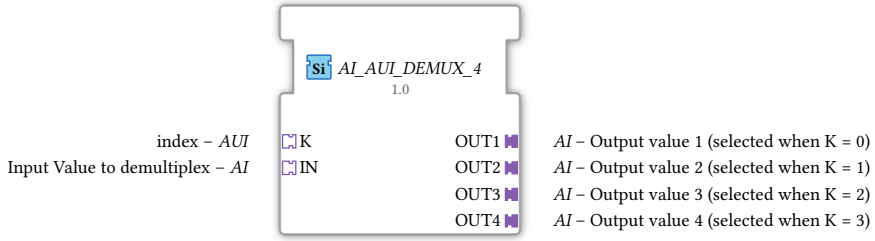
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AI_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AI_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `AI_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AI_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `INT`. Unlike `AI_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit signed integer to forward, routed to exactly one of the outputs (AI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_DEMUX`) – shared across all port counts of this block family (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit signed integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

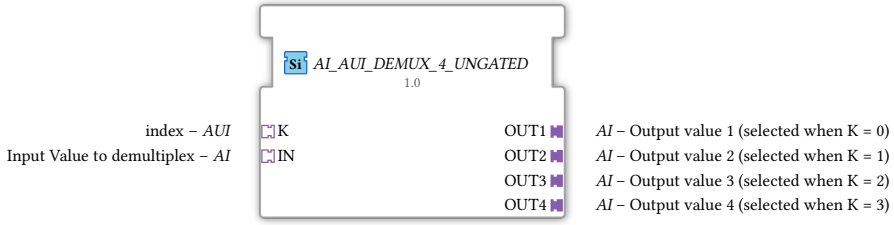
- **`AI_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AI_AUI_DEMUX_4` carries the demultiplexer logic of the `AI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AI_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `INT`. Unlike `AI_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit signed integer to forward, routed to exactly one of the outputs (AI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see “Change Detection” below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn’t changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_DEMUX`) – shared across all port counts of this block family (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit signed integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

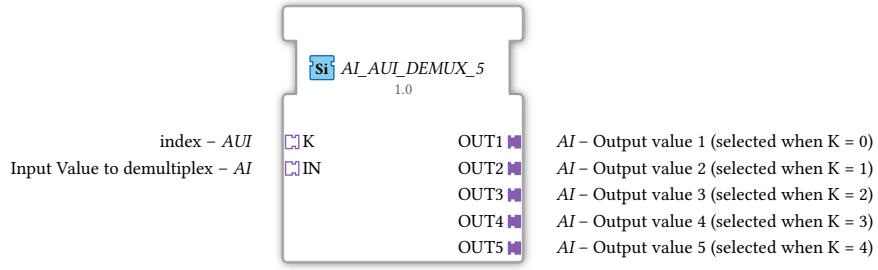
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AI_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AI_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AI_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `INT`. Unlike `AI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit signed integer to forward, routed to exactly one of the outputs (AI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_DEMUX`) – shared across all port counts of this block family (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit signed integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

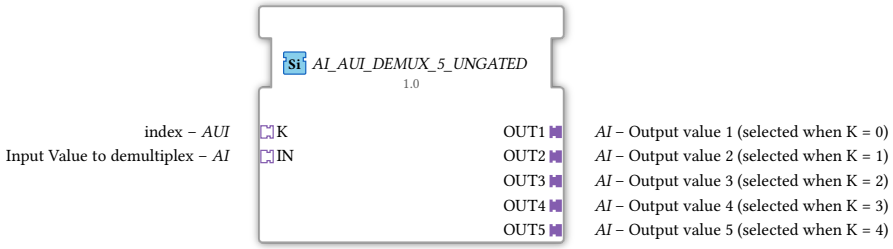
- **`AI_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AI_AUI_DEMUX_5` carries the demultiplexer logic of the `AI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_AUI_DEMUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AI_AUI_DEMUX_5_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `INT`. Unlike `AI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_DEMUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit signed integer to forward, routed to exactly one of the outputs (AI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_DEMUX`) – shared across all port counts of this block family (AI_AUI_DEMUX_2 … AI_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit signed integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

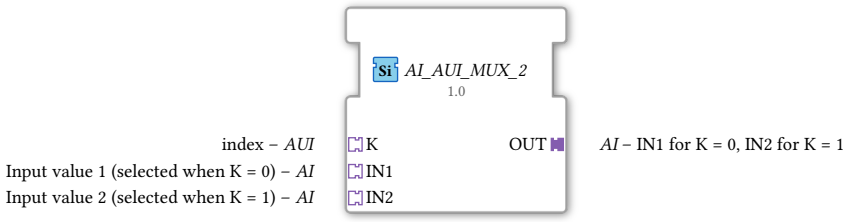
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AI_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AI_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AI_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `INT`. Unlike `AI_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its

own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AI_AUI_MUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 2, passed through to OUT when the index is K = 0 (AI adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to OUT when the index is K = 1 (AI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN2:

1. The current value of K.D1 determines which of the 2 input adapters (IN1 … IN2) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AI_AUI_MUX) – shared across all port counts of this block family (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN2) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 16-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

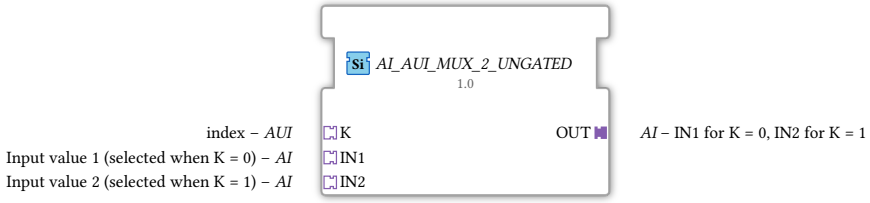
- **`AI_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AI_AUI_MUX_2` carries the multiplexer logic of the `AI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_AUI_MUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AI_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `INT`. Unlike `AI_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.

- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AI adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` ... `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_MUX`) – shared across all port counts of this block family (AI_AUI_MUX_2_UNGATED ... AI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 16-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AI_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AI_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_MUX_3



• – ○ • – ○ • – ○ •

Introduction

AI_AUI_MUX_3 is the adapter-based variant of the generic multiplexer for data type INT. Unlike AI_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AI_AUI_MUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (AI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (AI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (AI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AI_AUI_MUX) – shared across all port counts of this block family (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 16-bit signed integer sources through a centrally managed adapter index

- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

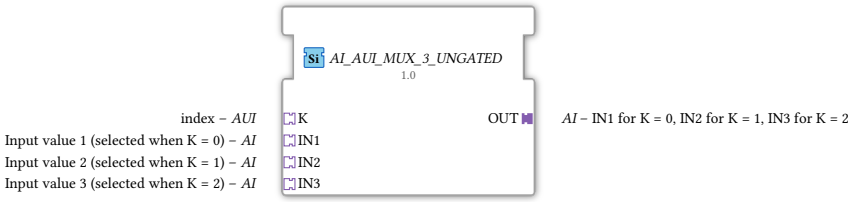
- **`AI_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AI_AUI_MUX_3` carries the multiplexer logic of the `AI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_AUI_MUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AI_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `INT`. Unlike `AI_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_MUX`) – shared across all port counts of this block family (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 16-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

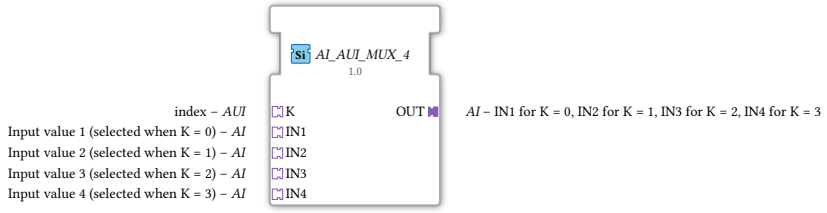
Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AI_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AI_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_MUX_4



• — ○ • — ○ • — ○ •

Introduction

`AI_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `INT`. Unlike `AI_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_MUX`) – shared across all port counts of this block family (AI_AUI_MUX_2 … AI_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 16-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

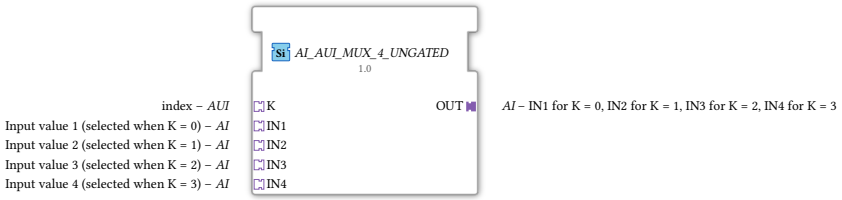
- **`AI_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AI_AUI_MUX_4` carries the multiplexer logic of the `AI_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_AUI_MUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• — ○ • — ○ • — ○ •

Introduction

`AI_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `INT`. Unlike `AI_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_MUX`) – shared across all port counts of this block family (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 16-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

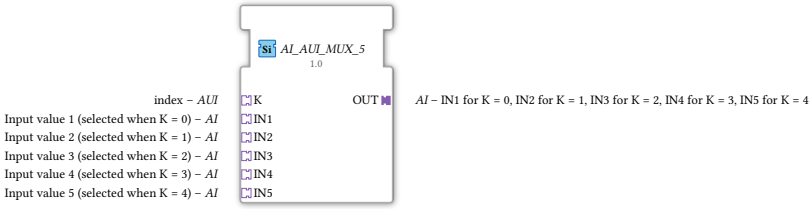
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AI_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AI_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `AI_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`AI_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `INT`. Unlike `AI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AI_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AI adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AI adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AI_AUI_MUX`) – shared across all port counts of this block family (AI_AUI_MUX_2 … AI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 16-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

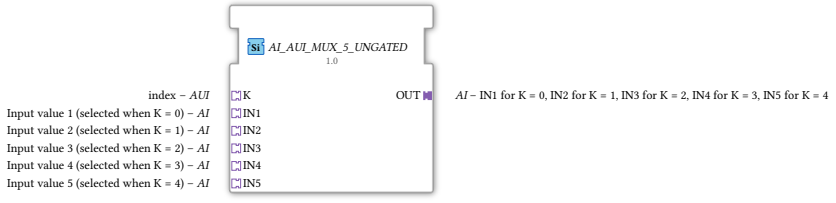
- `AI_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AI_AUI_MUX_5` carries the multiplexer logic of the `AI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AI_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AI_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AI_AUI_MUX_5_UNGATED is the adapter-based variant of the generic multiplexer for data type INT. Unlike AI_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AI_AUI_MUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (AI adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (AI adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (AI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (AI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (AI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN5:

1. The current value of K.D1 determines which of the 5 input adapters (IN1 … IN5) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AI_AUI_MUX) – shared across all port counts of this block family (AI_AUI_MUX_2 … AI_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN5) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 16-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AI_AUI_MUX_5`**: The gated variant – updates the output only on an actual value change.

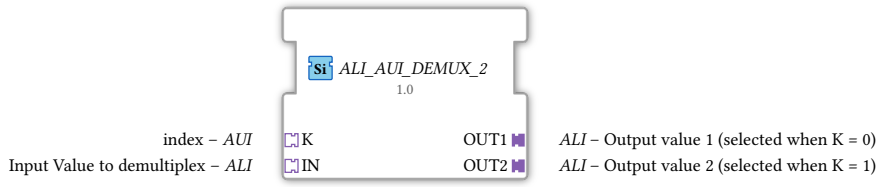
Conclusion

`AI_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALI_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`ALI_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `LINT`. Unlike `ALI_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.

- **IN** (Socket): input adapter carrying the 64-bit signed integer to forward, routed to exactly one of the outputs (ALI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (ALI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_DEMUX`) – shared across all port counts of this block family (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit signed integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

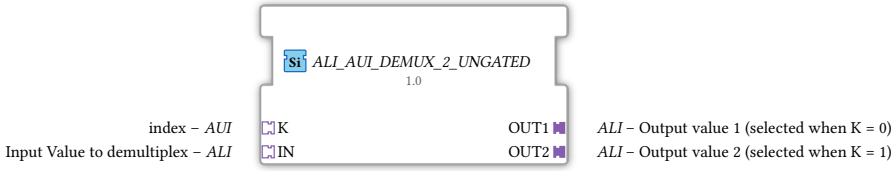
- **`ALI_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALI_AUI_DEMUX_2` carries the demultiplexer logic of the `ALI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ALI_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `LINT`. Unlike `ALI_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit signed integer to forward, routed to exactly one of the outputs (ALI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (ALI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_DEMUX`) – shared across all port counts of this block family (ALI_AUI_DEMUX_2_UNGATED … ALI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit signed integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ALI_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALI_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `ALI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_DEMUX_3



• – ○ • – ○ • – ○ •

Introduction

`ALI_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `LINT`. Unlike `ALI_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN** (Socket): input adapter carrying the 64-bit signed integer to forward, routed to exactly one of the outputs (ALI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (ALI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (ALI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_DEMUX`) – shared across all port counts of this block family (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit signed integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALI_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

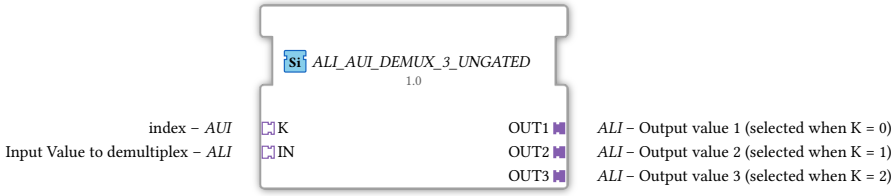
- **`ALI_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALI_AUI_DEMUX_3` carries the demultiplexer logic of the `ALI_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_DEMUX_3_UNGATED

i UNGATED variant: This block is the ungated version of ALI_AUI_DEMUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

ALI_AUI_DEMUX_3_UNGATED is the adapter-based variant of the generic demultiplexer for data type LINT. Unlike ALI_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ALI_AUI_DEMUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 64-bit signed integer to forward, routed to exactly one of the outputs (ALI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (ALI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (ALI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 3 output adapters (OUT1 … OUT3) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_ALI_AUI_DEMUX) – shared across all port counts of this block family (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) ⟶ determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit signed integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with ALI_DEMUX_3 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

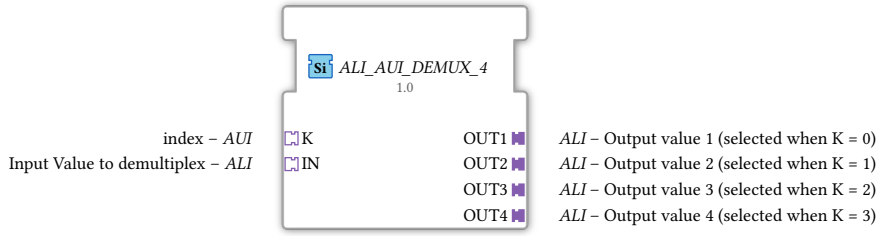
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **ALI_AUI_DEMUX_3**: The gated variant – updates the output only on an actual value change.

Conclusion

ALI_AUI_DEMUX_3_UNGATED carries the demultiplexer logic of the ALI_DEMUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_DEMUX_4



• – ○ • – ○ • – ○ •

Introduction

ALI_AUI_DEMUX_4 is the adapter-based variant of the generic demultiplexer for data type LINT. Unlike ALI_DEMUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit signed integer to forward, routed to exactly one of the outputs (ALI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (ALI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (ALI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (ALI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_DEMUX`) – shared across all port counts of this block family (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit signed integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

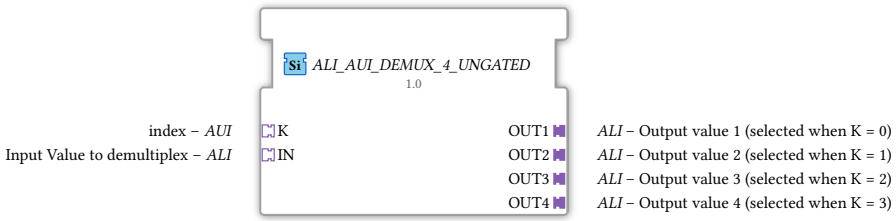
- **`ALI_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALI_AUI_DEMUX_4` carries the demultiplexer logic of the `ALI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ALI_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `LINT`. Unlike `ALI_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.

- **IN** (Socket): input adapter carrying the 64-bit signed integer to forward, routed to exactly one of the outputs (ALI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (ALI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (ALI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (ALI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_DEMUX`) – shared across all port counts of this block family (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit signed integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

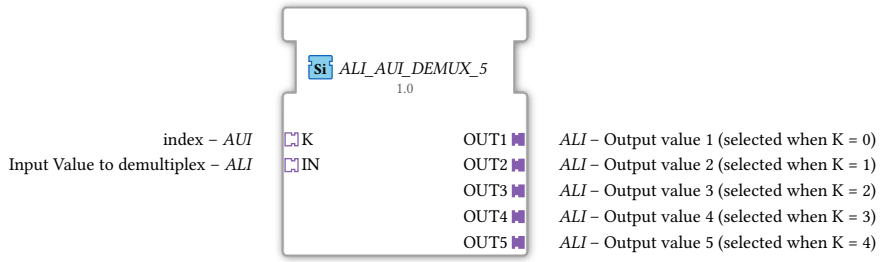
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ALI_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALI_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `ALI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`ALI_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `LINT`. Unlike `ALI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit signed integer to forward, routed to exactly one of the outputs (ALI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (ALI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (ALI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (ALI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (ALI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_DEMUX`) – shared across all port counts of this block family (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit signed integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

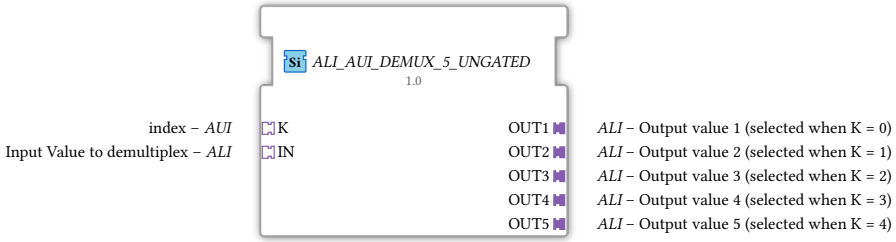
- **`ALI_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALI_AUI_DEMUX_5` carries the demultiplexer logic of the `ALI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_AUI_DEMUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ALI_AUI_DEMUX_5_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `LINT`. Unlike `ALI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K

data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ALI_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 64-bit signed integer to forward, routed to exactly one of the outputs (ALI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (ALI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (ALI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (ALI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (ALI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 5 output adapters (OUT1 … OUT5) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_ALI_AUI_DEMUX) – shared across all port counts of this block family (ALI_AUI_DEMUX_2 … ALI_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) ⟶ determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit signed integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

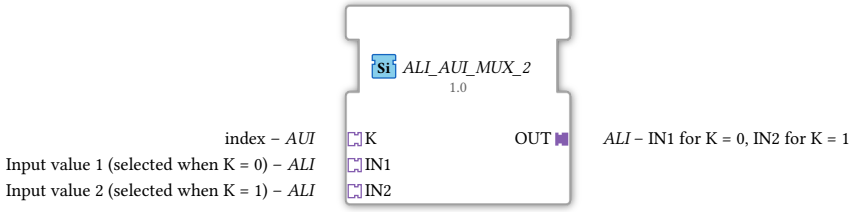
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ALI_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALI_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `ALI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`ALI_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `LINT`. Unlike `ALI_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (ALI adapter type).

- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (ALI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_MUX`) – shared across all port counts of this block family (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 64-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

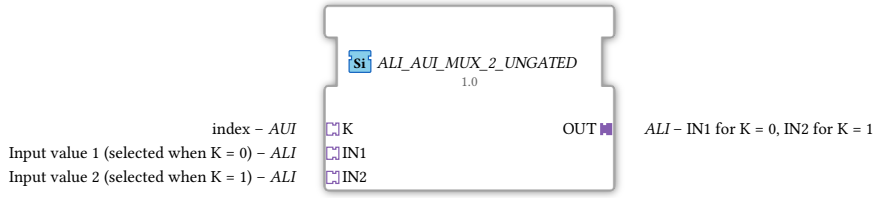
- **`ALI_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALI_AUI_MUX_2` carries the multiplexer logic of the `ALI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_AUI_MUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ALI_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `LINT`. Unlike `ALI_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (ALI adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (ALI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_MUX`) – shared across all port counts of this block family (ALI_AUI_MUX_2_UNGATED … ALI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 64-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

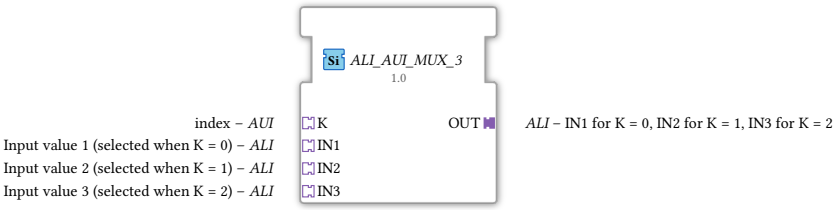
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`ALI_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALI_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `ALI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`ALI_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `LINT`. Unlike `ALI_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (ALI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (ALI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (ALI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_MUX`) – shared across all port counts of this block family (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 64-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`ALI_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

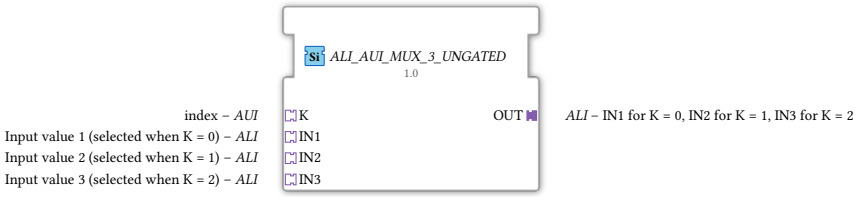
Conclusion

`ALI_AUI_MUX_3` carries the multiplexer logic of the `ALI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_AUI_MUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change

(e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ALI_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `LINT`. Unlike `ALI_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (ALI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (ALI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (ALI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_MUX`) – shared across all port counts of this block family (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 64-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`ALI_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALI_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `ALI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`ALI_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `LINT`. Unlike `ALI_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (ALI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (ALI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (ALI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (ALI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_MUX`) – shared across all port counts of this block family (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 64-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`ALI_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

ALI_AUI_MUX_4 carries the multiplexer logic of the ALI_MUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of ALI_AUI_MUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

ALI_AUI_MUX_4_UNGATED is the adapter-based variant of the generic multiplexer for data type LINT. Unlike ALI_MUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ALI_AUI_MUX_4_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 4, passed through to OUT when the index is K = 0 (ALI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to OUT when the index is K = 1 (ALI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to OUT when the index is K = 2 (ALI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to OUT when the index is K = 3 (ALI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN4:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_MUX`) – shared across all port counts of this block family (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 64-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

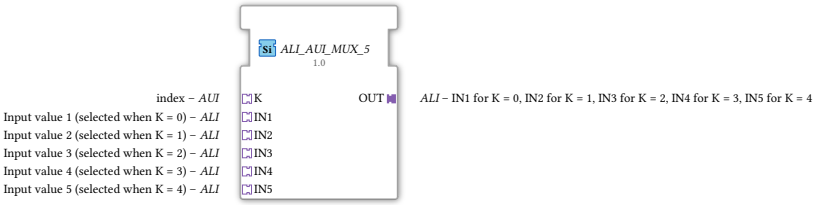
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`ALI_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALI_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `ALI_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`ALI_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `LINT`. Unlike `ALI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (ALI adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (ALI adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (ALI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (ALI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (ALI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_MUX`) – shared across all port counts of this block family (ALI_AUI_MUX_2 … ALI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 64-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

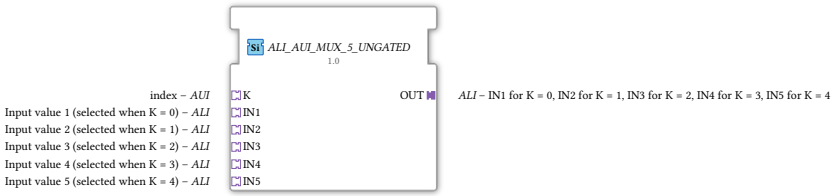
- **`ALI_AUI_MUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALI_AUI_MUX_5` carries the multiplexer logic of the `ALI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALI_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALI_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`ALI_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `LINT`. Unlike `ALI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALI_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (ALI adapter type).

- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (ALI adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (ALI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (ALI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (ALI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALI_AUI_MUX`) – shared across all port counts of this block family (ALI_AUI_MUX_2 … ALI_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 64-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`ALI_AUI_MUX_5`**: The gated variant – updates the output only on an actual value change.

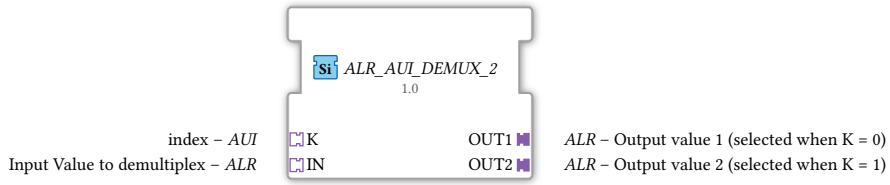
Conclusion

`ALI_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `ALI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ALR_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`ALR_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `LREAL`. Unlike `ALR_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit floating-point value to forward, routed to exactly one of the outputs (ALR adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (ALR adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_DEMUX`) – shared across all port counts of this block family (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit floating-point value to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALR_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

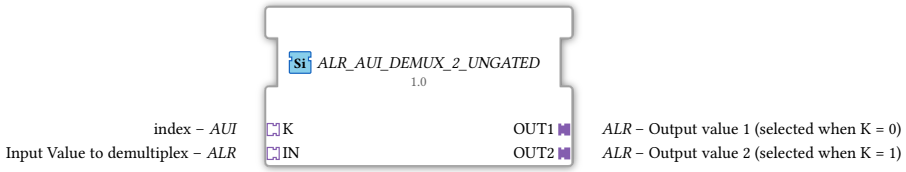
- **`ALR_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALR_AUI_DEMUX_2` carries the demultiplexer logic of the `ALR_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ALR_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `LREAL`. Unlike `ALR_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit floating-point value to forward, routed to exactly one of the outputs (ALR adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (ALR adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_DEMUX`) – shared across all port counts of this block family (ALR_AUI_DEMUX_2_UNGATED … ALR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit floating-point value to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with ALR_DEMUX_2 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

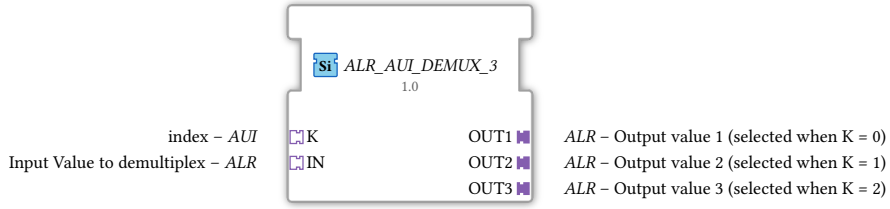
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **ALR_AUI_DEMUX_2**: The gated variant – updates the output only on an actual value change.

Conclusion

ALR_AUI_DEMUX_2_UNGATED carries the demultiplexer logic of the ALR_DEMUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

ALR_AUI_DEMUX_3 is the adapter-based variant of the generic demultiplexer for data type LREAL. Unlike ALR_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ALR_AUI_DEMUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 64-bit floating-point value to forward, routed to exactly one of the outputs (ALR adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (ALR adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (ALR adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_DEMUX`) – shared across all port counts of this block family (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit floating-point value to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALR_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

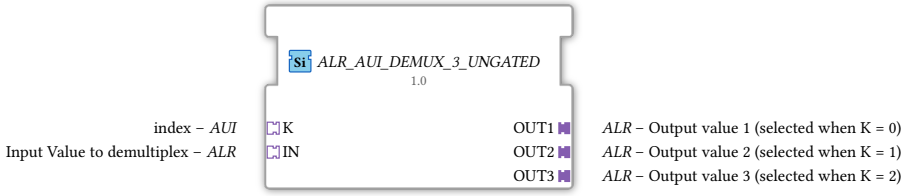
- **`ALR_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALR_AUI_DEMUX_3` carries the demultiplexer logic of the `ALR_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

ALR_AUI_DEMUX_3_UNGATED is the adapter-based variant of the generic demultiplexer for data type LREAL. Unlike ALR_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ALR_AUI_DEMUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 64-bit floating-point value to forward, routed to exactly one of the outputs (ALR adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (ALR adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (ALR adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 3 output adapters (OUT1 … OUT3) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_ALR_AUI_DEMUX) – shared across all port counts of this block family (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit floating-point value to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALR_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

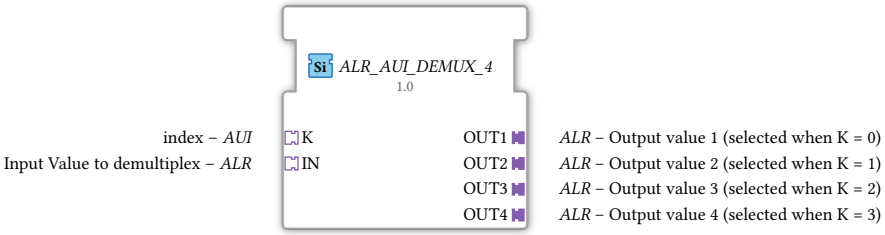
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ALR_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALR_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `ALR_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`ALR_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `LREAL`. Unlike `ALR_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.

- **IN** (Socket): input adapter carrying the 64-bit floating-point value to forward, routed to exactly one of the outputs (ALR adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (ALR adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (ALR adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (ALR adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_DEMUX`) – shared across all port counts of this block family (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit floating-point value to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALR_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ALR_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

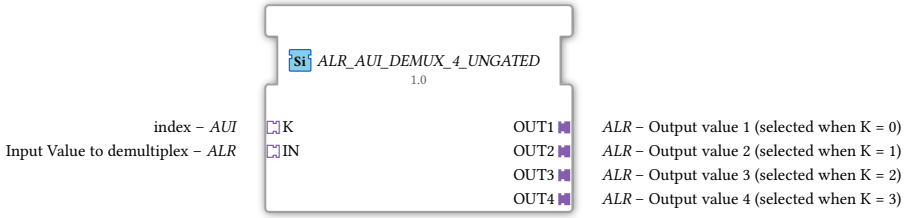
Conclusion

`ALR_AUI_DEMUX_4` carries the demultiplexer logic of the `ALR_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change

(e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ALR_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `LREAL`. Unlike `ALR_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit floating-point value to forward, routed to exactly one of the outputs (ALR adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (ALR adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (ALR adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (ALR adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_DEMUX`) – shared across all port counts of this block family (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit floating-point value to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALR_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

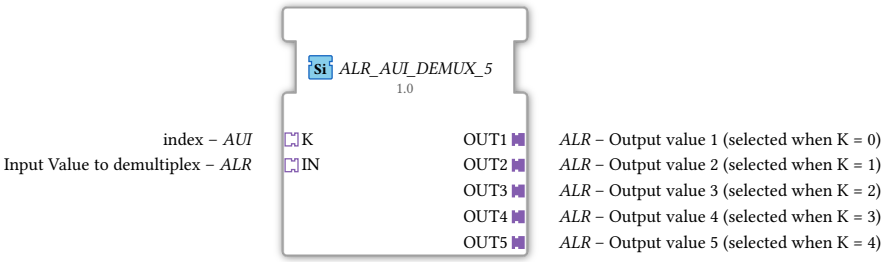
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ALR_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALR_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `ALR_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`ALR_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `LREAL`. Unlike `ALR_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit floating-point value to forward, routed to exactly one of the outputs (ALR adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (ALR adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (ALR adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (ALR adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (ALR adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_DEMUX`) – shared across all port counts of this block family (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit floating-point value to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)

- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with ALR_DEMUX_5 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

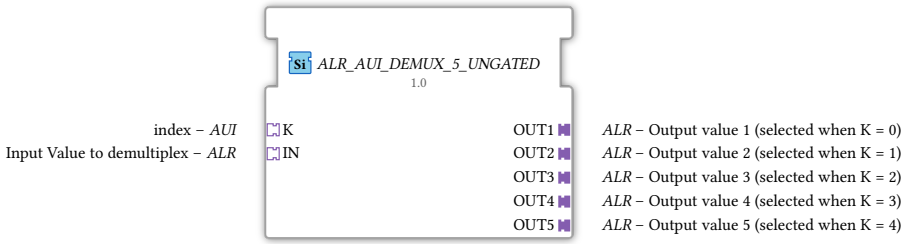
- **ALR_AUI_DEMUX_5_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

ALR_AUI_DEMUX_5 carries the demultiplexer logic of the ALR_DEMUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of ALR_AUI_DEMUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

ALR_AUI_DEMUX_5_UNGATED is the adapter-based variant of the generic demultiplexer for data type LREAL. Unlike ALR_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ALR_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN** (Socket): input adapter carrying the 64-bit floating-point value to forward, routed to exactly one of the outputs (ALR adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (ALR adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (ALR adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (ALR adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (ALR adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_DEMUX`) – shared across all port counts of this block family (ALR_AUI_DEMUX_2 … ALR_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit floating-point value to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ALR_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

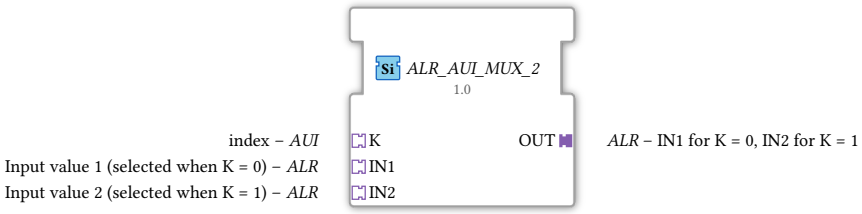
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ALR_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALR_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `ALR_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`ALR_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `LREAL`. Unlike `ALR_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (ALR adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (ALR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_MUX`) – shared across all port counts of this block family (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 64-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALR_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

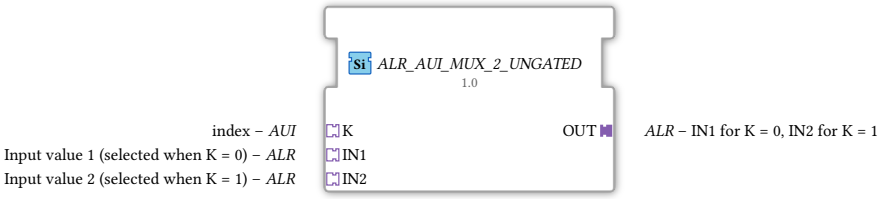
- **`ALR_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALR_AUI_MUX_2` carries the multiplexer logic of the `ALR_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ALR_AUI_MUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`ALR_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `LREAL`. Unlike `ALR_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (ALR adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (ALR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_MUX`) – shared across all port counts of this block family (ALR_AUI_MUX_2_UNGATED … ALR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 64-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALR_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

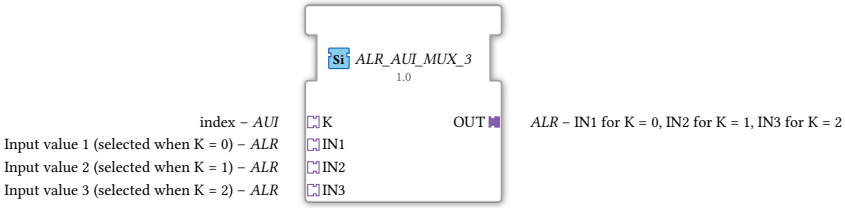
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`ALR_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALR_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `ALR_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_MUX_3



• – ○ • – ○ • – ○ •

Introduction

`ALR_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `LREAL`. Unlike `ALR_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (ALR adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (ALR adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (ALR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_MUX`) – shared across all port counts of this block family (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 64-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALR_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`ALR_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALR_AUI_MUX_3` carries the multiplexer logic of the `ALR_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_MUX_3_UNGATED

i UNGATED variant: This block is the ungated version of `ALR_AUI_MUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

ALR_AUI_MUX_3_UNGATED is the adapter-based variant of the generic multiplexer for data type LREAL. Unlike ALR_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ALR_AUI_MUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (ALR adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (ALR adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (ALR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_ALR_AUI_MUX) – shared across all port counts of this block family (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) → read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 64-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALR_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

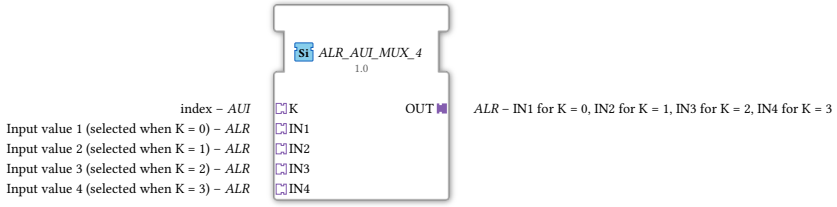
Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`ALR_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALR_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `ALR_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`ALR_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `LREAL`. Unlike `ALR_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (ALR adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (ALR adapter type).

- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (ALR adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (ALR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_MUX`) – shared across all port counts of this block family (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 64-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALR_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`ALR_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ALR_AUI_MUX_4` carries the multiplexer logic of the `ALR_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of ALR_AUI_MUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• — ○ • — ○ • — ○ •

Introduction

`ALR_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `LREAL`. Unlike `ALR_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (ALR adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (ALR adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (ALR adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (ALR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see “Change Detection” below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn’t changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_MUX`) – shared across all port counts of this block family (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 64-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALR_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

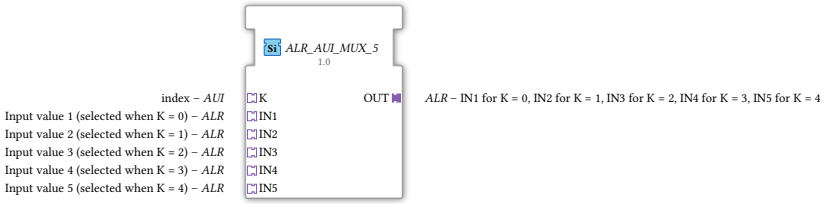
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`ALR_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ALR_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `ALR_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`ALR_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `LREAL`. Unlike `ALR_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ALR_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (ALR adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (ALR adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (ALR adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (ALR adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (ALR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_MUX`) – shared across all port counts of this block family (ALR_AUI_MUX_2 … ALR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 64-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALR_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

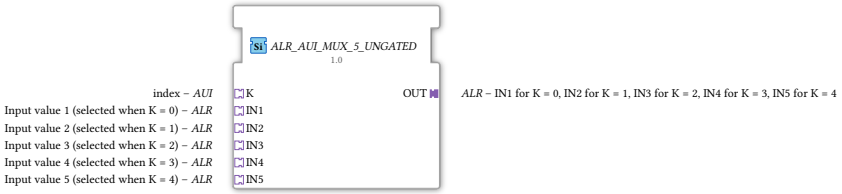
- `ALR_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

ALR_AUI_MUX_5 carries the multiplexer logic of the ALR_MUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ALR_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of ALR_AUI_MUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

ALR_AUI_MUX_5_UNGATED is the adapter-based variant of the generic multiplexer for data type LREAL. Unlike ALR_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ALR_AUI_MUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (ALR adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (ALR adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (ALR adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (ALR adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (ALR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ALR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` ... `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ALR_AUI_MUX`) – shared across all port counts of this block family (ALR_AUI_MUX_2 ... ALR_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 64-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ALR_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- `ALR_AUI_MUX_5`: The gated variant – updates the output only on an actual value change.

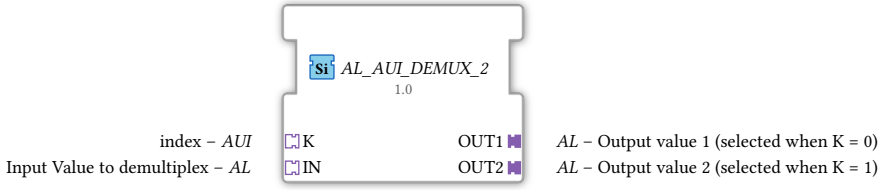
Conclusion

`ALR_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `ALR_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AL_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

AL_AUI_DEMUX_2 is the adapter-based variant of the generic demultiplexer for data type LWORD. Unlike AL_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AL_AUI_DEMUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 64-bit bit pattern to forward, routed to exactly one of the outputs (AL adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AL adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 2 output adapters (OUT1 … OUT2) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AL_AUI_DEMUX) – shared across all port counts of this block family (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit bit pattern to up to 2 downstream consumers, selected via an adapter-carried index

- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AL_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

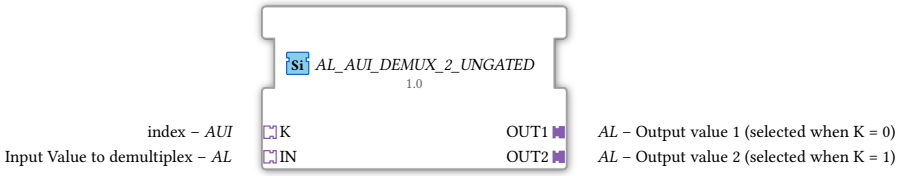
- **`AL_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AL_AUI_DEMUX_2` carries the demultiplexer logic of the `AL_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AL_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `LWORD`. Unlike `AL_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN** (Socket): input adapter carrying the 64-bit bit pattern to forward, routed to exactly one of the outputs (AL adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AL adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_DEMUX`) – shared across all port counts of this block family (AL_AUI_DEMUX_2_UNGATED … AL_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit bit pattern to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AL_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

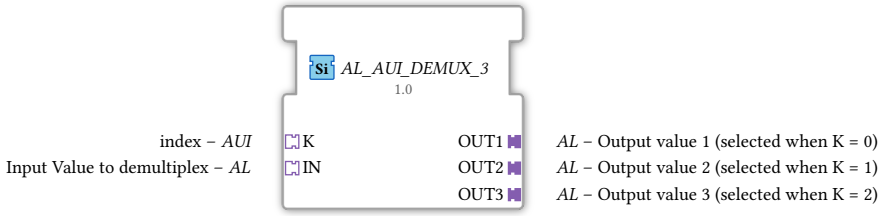
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AL_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AL_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AL_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AL_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `LWORD`. Unlike `AL_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit bit pattern to forward, routed to exactly one of the outputs (AL adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AL adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AL adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_DEMUX`) – shared across all port counts of this block family (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit bit pattern to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AL_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

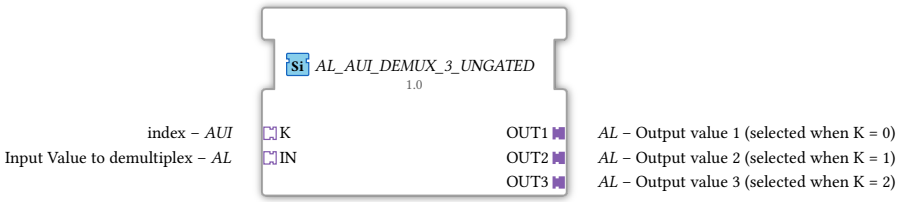
- **`AL_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AL_AUI_DEMUX_3` carries the demultiplexer logic of the `AL_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AL_AUI_DEMUX_3_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `LWORD`. Unlike `AL_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_DEMUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit bit pattern to forward, routed to exactly one of the outputs (AL adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AL adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AL adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_DEMUX`) – shared across all port counts of this block family (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit bit pattern to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AL_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

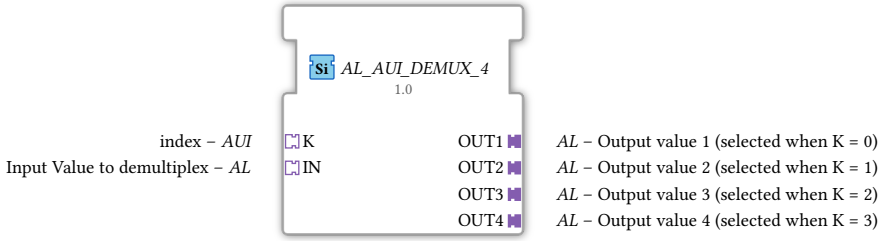
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AL_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AL_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `AL_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AL_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `LWORD`. Unlike `AL_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit bit pattern to forward, routed to exactly one of the outputs (AL adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AL adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AL adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AL adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_DEMUX`) – shared across all port counts of this block family (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit bit pattern to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AL_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

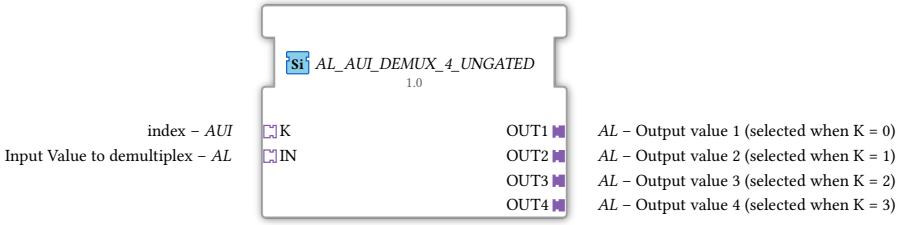
- **`AL_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AL_AUI_DEMUX_4` carries the demultiplexer logic of the `AL_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AL_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `LWORD`. Unlike `AL_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit bit pattern to forward, routed to exactly one of the outputs (AL adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AL adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AL adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AL adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_DEMUX`) – shared across all port counts of this block family (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit bit pattern to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AL_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

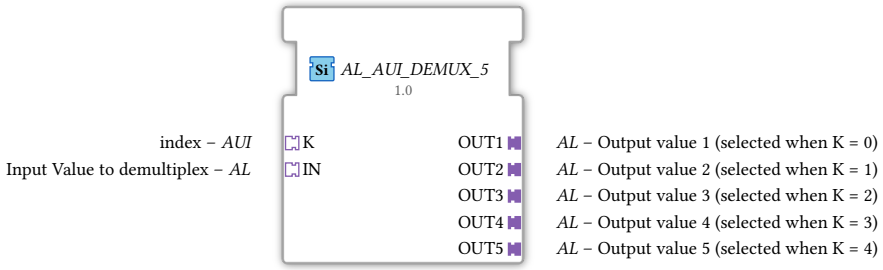
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AL_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AL_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AL_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AL_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `LWORD`. Unlike `AL_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit bit pattern to forward, routed to exactly one of the outputs (AL adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AL adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AL adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AL adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AL adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_DEMUX`) – shared across all port counts of this block family (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit bit pattern to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AL_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

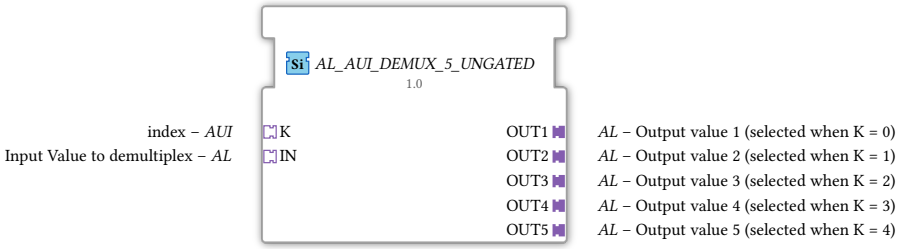
- **`AL_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AL_AUI_DEMUX_5 carries the demultiplexer logic of the AL_DEMUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_AUI_DEMUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AL_AUI_DEMUX_5_UNGATED is the adapter-based variant of the generic demultiplexer for data type LWORD. Unlike AL_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AL_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter’s own E1/D1 event.
- **IN** (Socket): input adapter carrying the 64-bit bit pattern to forward, routed to exactly one of the outputs (AL adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AL adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AL adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AL adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AL adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_DEMUX`) – shared across all port counts of this block family (AL_AUI_DEMUX_2 … AL_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit bit pattern to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖ Comparison with Similar Blocks

Compare with `AL_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

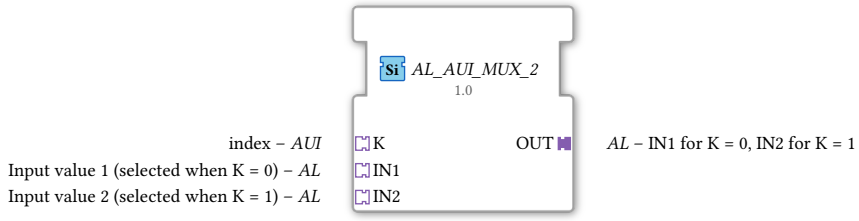
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AL_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AL_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AL_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AL_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `LWORD`. Unlike `AL_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index

be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AL_AUI_MUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 2, passed through to OUT when the index is K = 0 (AL adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to OUT when the index is K = 1 (AL adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN2:

1. The current value of K.D1 determines which of the 2 input adapters (IN1 … IN2) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AL_AUI_MUX) – shared across all port counts of this block family (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN2) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 64-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AL_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

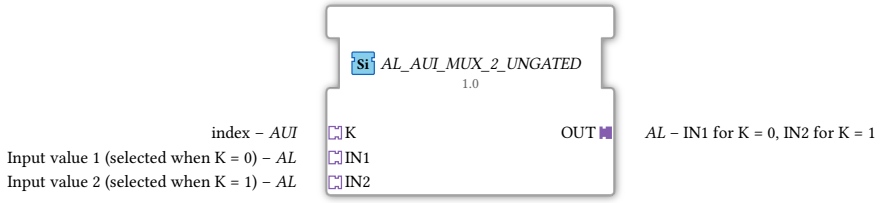
- **`AL_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AL_AUI_MUX_2` carries the multiplexer logic of the `AL_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_AUI_MUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AL_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `LWORD`. Unlike `AL_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.

- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AL adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AL adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` ... `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_MUX`) – shared across all port counts of this block family (AL_AUI_MUX_2_UNGATED ... AL_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 64-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AL_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AL_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AL_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AL_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_MUX_3



• – ○ • – ○ • – ○ •

Introduction

AL_AUI_MUX_3 is the adapter-based variant of the generic multiplexer for data type LWORD. Unlike AL_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AL_AUI_MUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (AL adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (AL adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (AL adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AL_AUI_MUX) – shared across all port counts of this block family (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 64-bit bit pattern sources through a centrally managed adapter index

- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AL_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

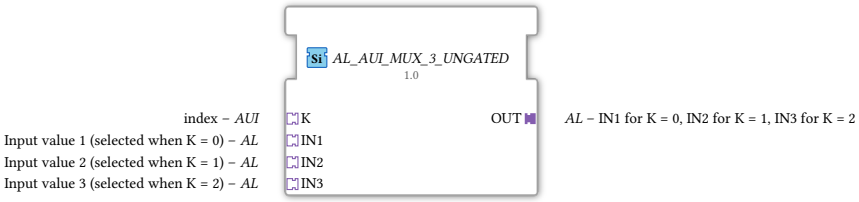
- **`AL_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AL_AUI_MUX_3` carries the multiplexer logic of the `AL_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AL_AUI_MUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AL_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `LWORD`. Unlike `AL_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AL adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AL adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AL adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_MUX`) – shared across all port counts of this block family (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 64-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AL_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

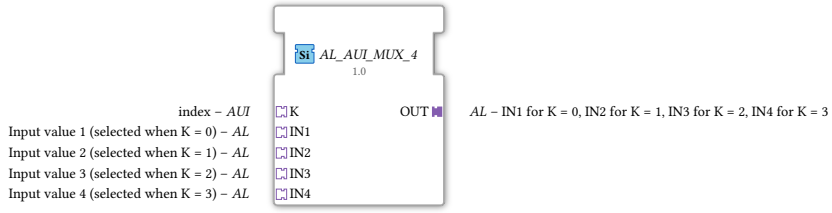
Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AL_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AL_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AL_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_MUX_4



• — ○ • — ○ • — ○ •

Introduction

`AL_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `LWORD`. Unlike `AL_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AL adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AL adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AL adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AL adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_MUX`) – shared across all port counts of this block family (AL_AUI_MUX_2 … AL_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 64-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AL_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

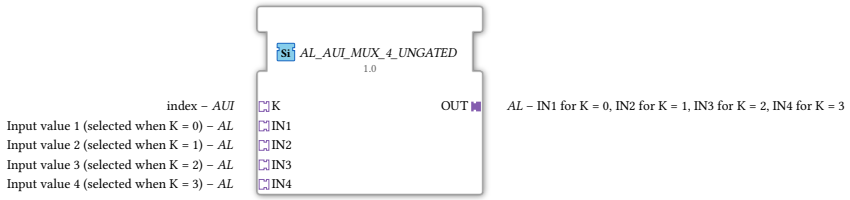
- **`AL_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AL_AUI_MUX_4` carries the multiplexer logic of the `AL_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_AUI_MUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AL_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `LWORD`. Unlike `AL_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AL adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AL adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AL adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AL adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_MUX`) – shared across all port counts of this block family (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 64-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AL_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

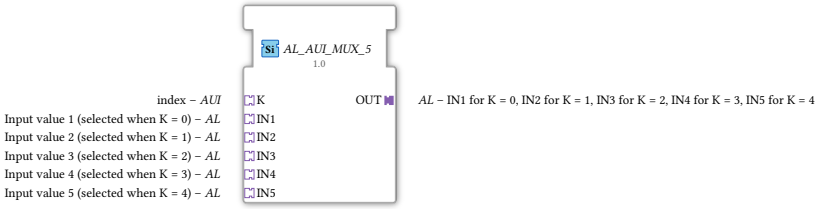
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AL_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AL_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `AL_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`AL_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `LWORD`. Unlike `AL_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AL adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AL adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AL adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AL adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AL adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_MUX`) – shared across all port counts of this block family (AL_AUI_MUX_2 … AL_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 64-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AL_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

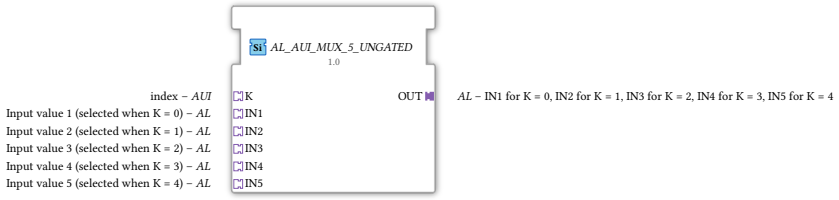
- `AL_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AL_AUI_MUX_5` carries the multiplexer logic of the `AL_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AL_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AL_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AL_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `LWORD`. Unlike `AL_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AL_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AL adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AL adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AL adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AL adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AL adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AL adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AL_AUI_MUX`) – shared across all port counts of this block family (AL_AUI_MUX_2 … AL_AUI_MUX_5_UNGATED).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 64-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AL_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- `AL_AUI_MUX_5`: The gated variant – updates the output only on an actual value change.

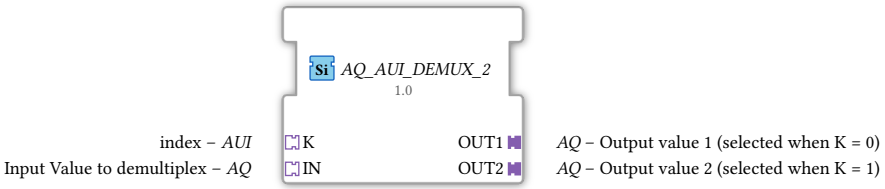
Conclusion

`AL_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AL_MUX_5` family over to a purely adapter-based index supply. This makes the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AQ_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AQ_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AQ adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AQ adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_DEMUX`) – shared across all port counts of this block family (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AQ_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

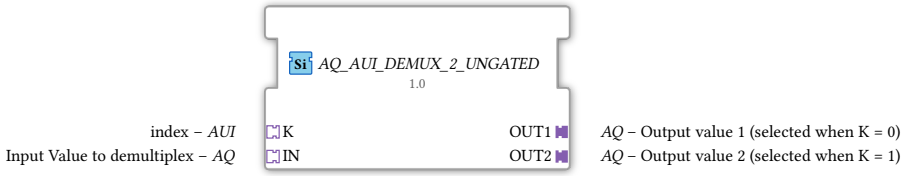
- **`AQ_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AQ_AUI_DEMUX_2` carries the demultiplexer logic of the `AQ_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AQ_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AQ adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AQ adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.

2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_DEMUX`) – shared across all port counts of this block family (AQ_AUI_DEMUX_2_UNGATED … AQ_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AQ_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

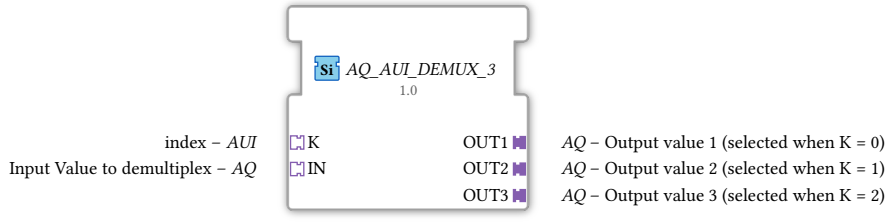
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AQ_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AQ_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AQ_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AQ_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index

be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AQ_AUI_DEMUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AQ adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AQ adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AQ adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 3 output adapters (OUT1 … OUT3) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AQ_AUI_DEMUX) – shared across all port counts of this block family (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The QUARTER value is technically transferred as a BYTE with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 3 downstream consumers, selected via an adapter-carried index

- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AQ_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

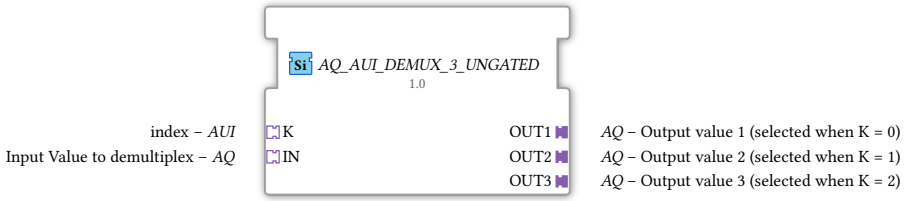
- **`AQ_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AQ_AUI_DEMUX_3` carries the demultiplexer logic of the `AQ_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_DEMUX_3_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AQ_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_DEMUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AQ adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AQ adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AQ adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_DEMUX`) – shared across all port counts of this block family (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AQ_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

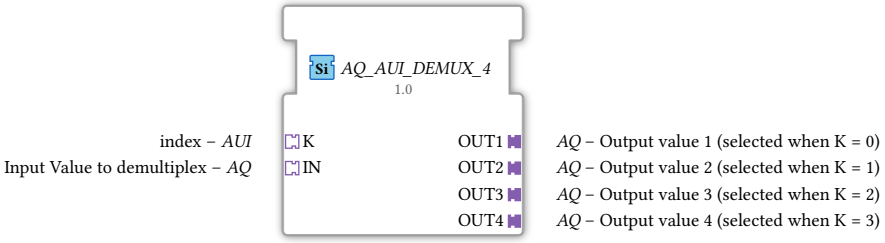
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AQ_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AQ_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `AQ_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AQ_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AQ adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AQ adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AQ adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AQ adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.

- Generic implementation (`GEN_AQ_AUI_DEMUX`) – shared across all port counts of this block family (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AQ_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

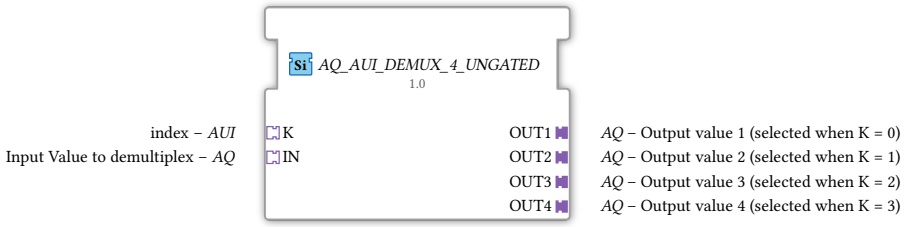
- **`AQ_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AQ_AUI_DEMUX_4` carries the demultiplexer logic of the `AQ_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AQ_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the

index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AQ_AUI_DEMUX_4_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AQ adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AQ adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AQ adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AQ adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 4 output adapters (OUT1 … OUT4) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AQ_AUI_DEMUX) – shared across all port counts of this block family (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The QUARTER value is technically transferred as a BYTE with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) ⟶ determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AQ_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

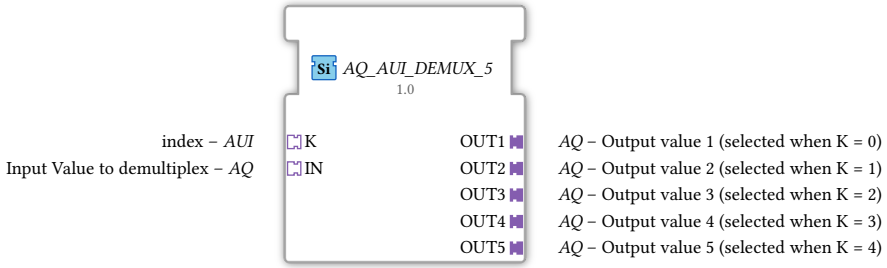
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AQ_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AQ_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AQ_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `BYTE`. Unlike `AQ_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AQ adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AQ adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AQ adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AQ adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AQ adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_DEMUX`) – shared across all port counts of this block family (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AQ_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

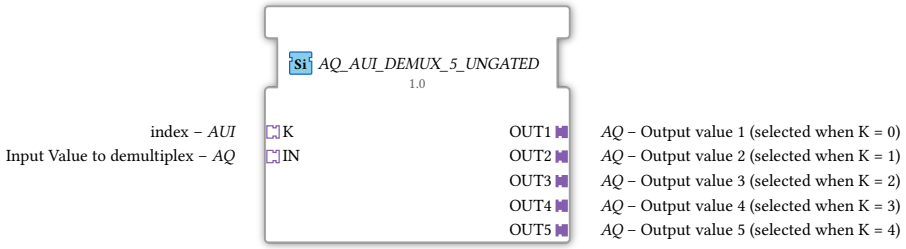
- **`AQ_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AQ_AUI_DEMUX_5` carries the demultiplexer logic of the `AQ_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_DEMUX_5_UNGATED

i UNGATED variant: This block is the ungated version of AQ_AUI_DEMUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AQ_AUI_DEMUX_5_UNGATED is the adapter-based variant of the generic demultiplexer for data type BYTE. Unlike AQ_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AQ_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the byte value (8-bit bit pattern) to forward, routed to exactly one of the outputs (AQ adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AQ adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AQ adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AQ adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AQ adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 5 output adapters (OUT1 … OUT5) is currently selected.

2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_DEMUX`) – shared across all port counts of this block family (AQ_AUI_DEMUX_2 … AQ_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a byte value (8-bit bit pattern) to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AQ_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

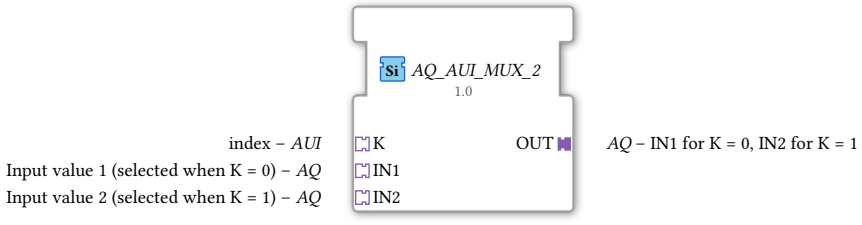
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AQ_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AQ_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AQ_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `BYTE`. Unlike `AQ_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AQ adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AQ adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_MUX`) – shared across all port counts of this block family (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AQ_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

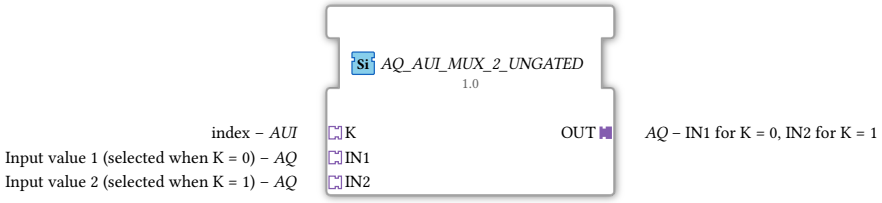
- **`AQ_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AQ_AUI_MUX_2` carries the multiplexer logic of the `AQ_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AQ_AUI_MUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `BYTE`. Unlike `AQ_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.

- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AQ adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AQ adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_MUX`) – shared across all port counts of this block family (AQ_AUI_MUX_2_UNGATED … AQ_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AQ_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AQ_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AQ_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AQ_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `BYTE`. Unlike `AQ_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AQ adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AQ adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AQ adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_MUX`) – shared across all port counts of this block family (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AQ_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

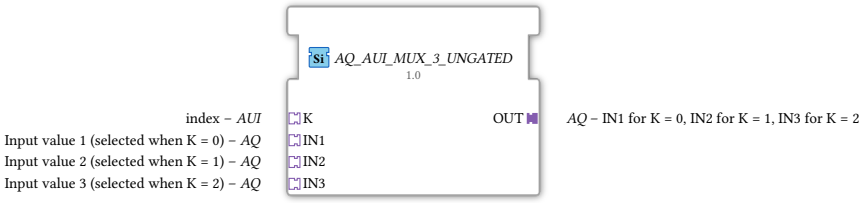
- **`AQ_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AQ_AUI_MUX_3` carries the multiplexer logic of the `AQ_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AQ_AUI_MUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AQ_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `BYTE`. Unlike `AQ_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AQ adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AQ adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AQ adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_MUX`) – shared across all port counts of this block family (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AQ_MUX_3 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **AQ_AUI_MUX_3**: The gated variant – updates the output only on an actual value change.

Conclusion

AQ_AUI_MUX_3_UNGATED carries the multiplexer logic of the AQ_MUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

AQ_AUI_MUX_4 is the adapter-based variant of the generic multiplexer for data type BYTE. Unlike AQ_MUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AQ_AUI_MUX_4 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 4, passed through to OUT when the index is K = 0 (AQ adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to OUT when the index is K = 1 (AQ adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to OUT when the index is K = 2 (AQ adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to OUT when the index is K = 3 (AQ adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1...IN4:

1. The current value of K.D1 determines which of the 4 input adapters (IN1 … IN4) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AQ_AUI_MUX) – shared across all port counts of this block family (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The QUARTER value is technically transferred as a BYTE with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN4) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AQ_MUX_4 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

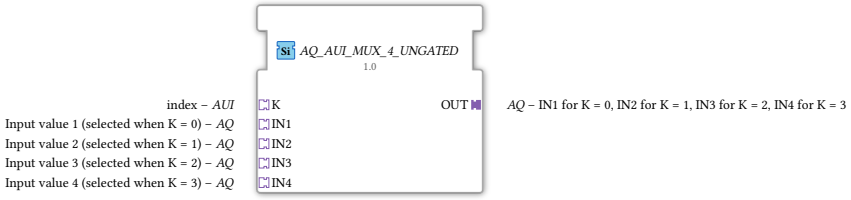
- **AQ_AUI_MUX_4_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AQ_AUI_MUX_4 carries the multiplexer logic of the AQ_MUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AQ_AUI_MUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AQ_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `BYTE`. Unlike `AQ_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AQ adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AQ adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AQ adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AQ adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_MUX`) – shared across all port counts of this block family (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AQ_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

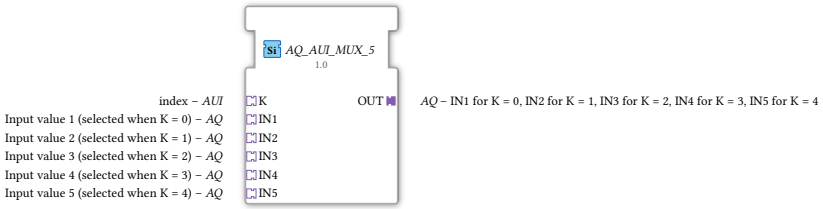
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AQ_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AQ_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `AQ_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`AQ_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `BYTE`. Unlike `AQ_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AQ_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AQ adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AQ adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AQ adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AQ adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AQ adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_MUX`) – shared across all port counts of this block family (AQ_AUI_MUX_2 … AQ_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AQ_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

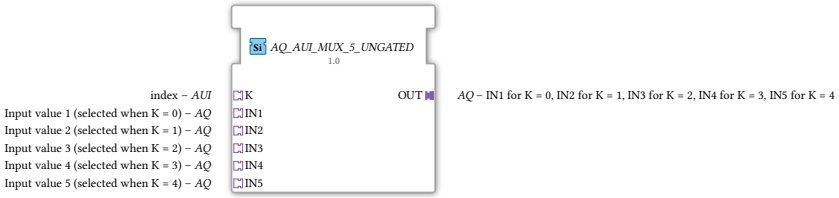
- **`AQ_AUI_MUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AQ_AUI_MUX_5 carries the multiplexer logic of the AQ_MUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AQ_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of AQ_AUI_MUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AQ_AUI_MUX_5_UNGATED is the adapter-based variant of the generic multiplexer for data type BYTE. Unlike AQ_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AQ_AUI_MUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (AQ adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (AQ adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (AQ adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (AQ adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (AQ adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AQ adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` ... `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AQ_AUI_MUX`) – shared across all port counts of this block family (AQ_AUI_MUX_2 ... AQ_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- The `QUARTER` value is technically transferred as a `BYTE` with 2 significant bits; interpretation as a quadrant/quarter value happens in the calling context.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 byte value (8-bit bit pattern) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AQ_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- `AQ_AUI_MUX_5`: The gated variant – updates the output only on an actual value change.

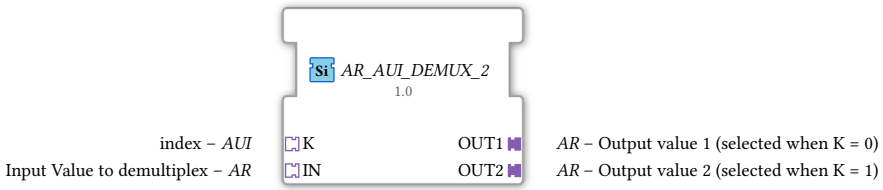
Conclusion

`AQ_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AQ_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AR_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

AR_AUI_DEMUX_2 is the adapter-based variant of the generic demultiplexer for data type REAL. Unlike AR_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AR_AUI_DEMUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 32-bit floating-point value to forward, routed to exactly one of the outputs (AR adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AR adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 2 output adapters (OUT1 … OUT2) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AR_AUI_DEMUX) – shared across all port counts of this block family (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit floating-point value to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AR_DEMUX_2 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

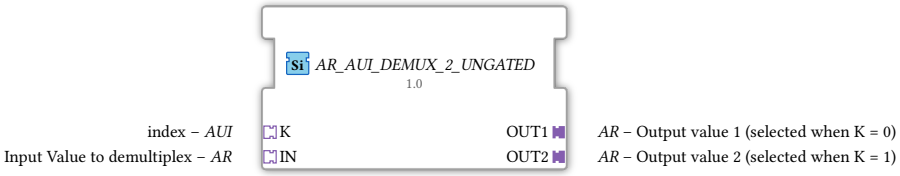
- **AR_AUI_DEMUX_2_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AR_AUI_DEMUX_2 carries the demultiplexer logic of the AR_DEMUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AR_AUI_DEMUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AR_AUI_DEMUX_2_UNGATED is the adapter-based variant of the generic demultiplexer for data type REAL. Unlike AR_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AR_AUI_DEMUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit floating-point value to forward, routed to exactly one of the outputs (AR adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AR adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_DEMUX`) – shared across all port counts of this block family (AR_AUI_DEMUX_2_UNGATED … AR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit floating-point value to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AR_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

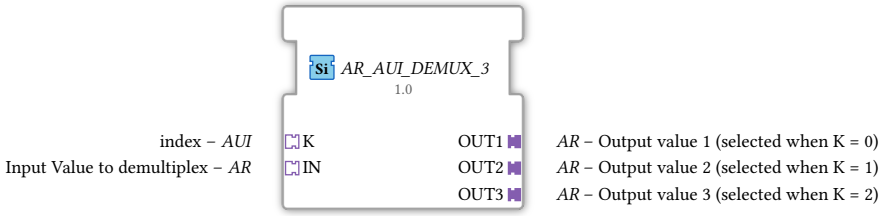
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AR_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AR_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AR_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AR_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `REAL`. Unlike `AR_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AR_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit floating-point value to forward, routed to exactly one of the outputs (AR adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AR adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AR adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_DEMUX`) – shared across all port counts of this block family (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit floating-point value to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AR_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

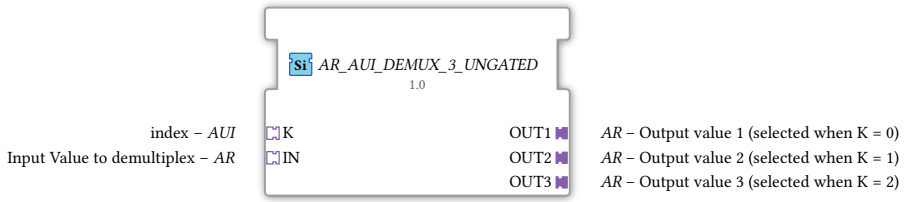
- **`AR_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AR_AUI_DEMUX_3` carries the demultiplexer logic of the `AR_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AR_AUI_DEMUX_3_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `REAL`. Unlike `AR_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AR_AUI_DEMUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit floating-point value to forward, routed to exactly one of the outputs (AR adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AR adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AR adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_DEMUX`) – shared across all port counts of this block family (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit floating-point value to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AR_DEMUX_3 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

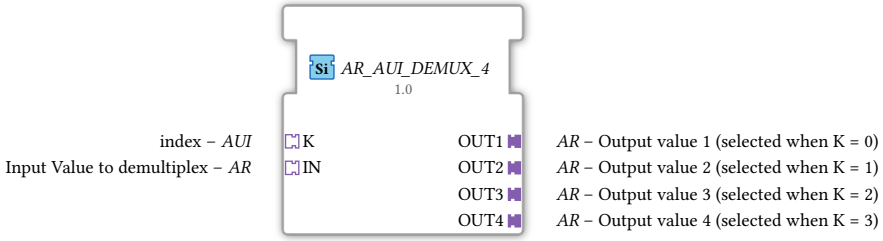
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **AR_AUI_DEMUX_3**: The gated variant – updates the output only on an actual value change.

Conclusion

AR_AUI_DEMUX_3_UNGATED carries the demultiplexer logic of the AR_DEMUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

AR_AUI_DEMUX_4 is the adapter-based variant of the generic demultiplexer for data type REAL. Unlike AR_DEMUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AR_AUI_DEMUX_4 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 32-bit floating-point value to forward, routed to exactly one of the outputs (AR adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AR adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AR adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AR adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_DEMUX`) – shared across all port counts of this block family (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit floating-point value to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AR_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

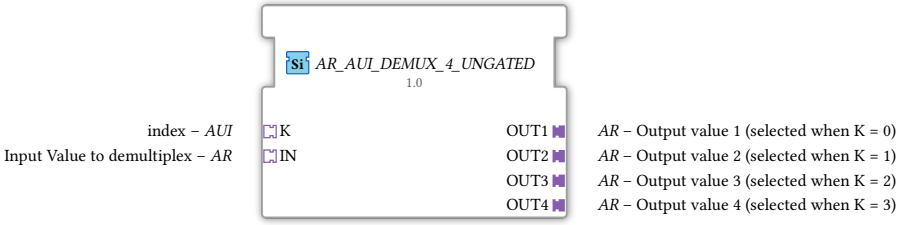
- **`AR_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AR_AUI_DEMUX_4` carries the demultiplexer logic of the `AR_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AR_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `REAL`. Unlike `AR_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AR_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit floating-point value to forward, routed to exactly one of the outputs (AR adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AR adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AR adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AR adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_DEMUX`) – shared across all port counts of this block family (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit floating-point value to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AR_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

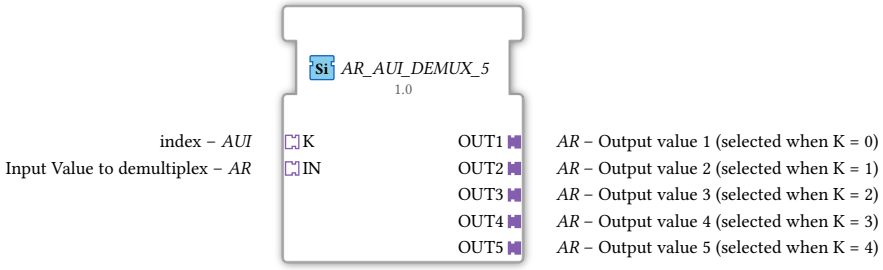
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AR_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AR_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AR_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AR_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `REAL`. Unlike `AR_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AR_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit floating-point value to forward, routed to exactly one of the outputs (AR adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AR adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AR adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AR adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AR adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_DEMUX`) – shared across all port counts of this block family (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit floating-point value to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AR_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

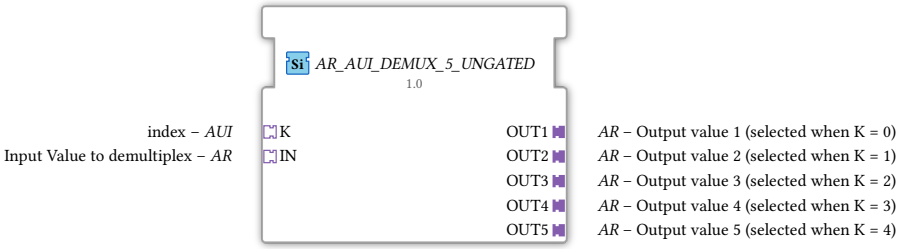
- **`AR_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AR_AUI_DEMUX_5 carries the demultiplexer logic of the AR_DEMUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of AR_AUI_DEMUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AR_AUI_DEMUX_5_UNGATED is the adapter-based variant of the generic demultiplexer for data type REAL. Unlike AR_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AR_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 32-bit floating-point value to forward, routed to exactly one of the outputs (AR adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AR adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AR adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AR adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AR adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_DEMUX`) – shared across all port counts of this block family (AR_AUI_DEMUX_2 … AR_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit floating-point value to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AR_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

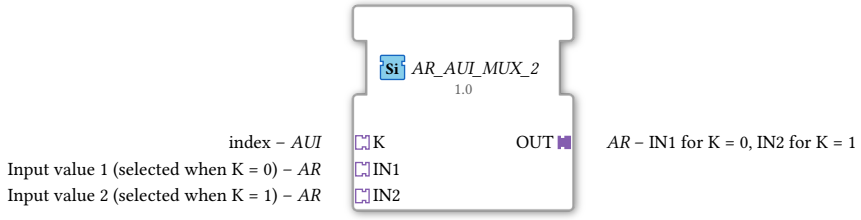
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AR_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AR_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AR_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_MUX_2



• – ○ • – ○ • – ○ •

Introduction

`AR_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `REAL`. Unlike `AR_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its

own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AR_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AR adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_MUX`) – shared across all port counts of this block family (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 32-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AR_MUX_2 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

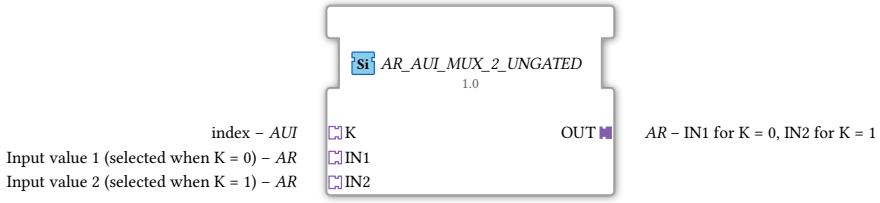
- **AR_AUI_MUX_2_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AR_AUI_MUX_2 carries the multiplexer logic of the AR_MUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AR_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AR_AUI_MUX_2_UNGATED is the adapter-based variant of the generic multiplexer for data type REAL. Unlike AR_MUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AR_AUI_MUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.

- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AR adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` ... `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_MUX`) – shared across all port counts of this block family (AR_AUI_MUX_2_UNGATED ... AR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 32-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AR_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AR_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AR_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AR_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

AR_AUI_MUX_3 is the adapter-based variant of the generic multiplexer for data type REAL. Unlike AR_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AR_AUI_MUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (AR adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (AR adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (AR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AR_AUI_MUX) – shared across all port counts of this block family (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 32-bit floating-point value sources through a centrally managed adapter index

- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AR_MUX_3 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

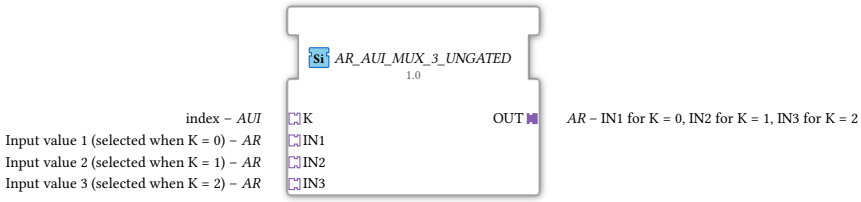
- **AR_AUI_MUX_3_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AR_AUI_MUX_3 carries the multiplexer logic of the AR_MUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AR_AUI_MUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AR_AUI_MUX_3_UNGATED is the adapter-based variant of the generic multiplexer for data type REAL. Unlike AR_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AR_AUI_MUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AR adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AR adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_MUX`) – shared across all port counts of this block family (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 32-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AR_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AR_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AR_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AR_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_MUX_4



• — ○ • — ○ • — ○ •

Introduction

`AR_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `REAL`. Unlike `AR_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AR_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AR adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AR adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AR adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_MUX`) – shared across all port counts of this block family (AR_AUI_MUX_2 … AR_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 32-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AR_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

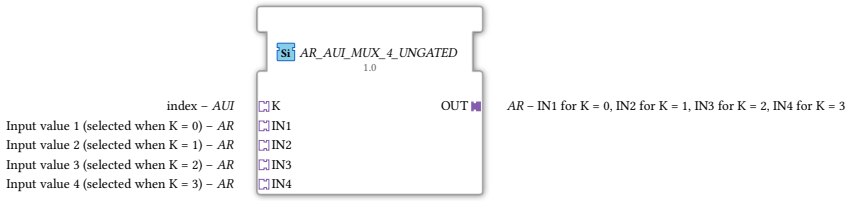
- **`AR_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AR_AUI_MUX_4` carries the multiplexer logic of the `AR_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AR_AUI_MUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AR_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `REAL`. Unlike `AR_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AR_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AR adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AR adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AR adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_MUX`) – shared across all port counts of this block family (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 32-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AR_MUX_4 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

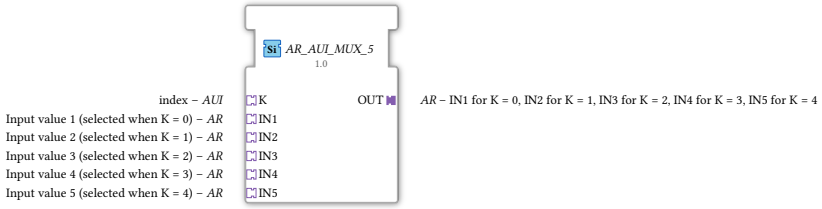
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **AR_AUI_MUX_4**: The gated variant – updates the output only on an actual value change.

Conclusion

AR_AUI_MUX_4_UNGATED carries the multiplexer logic of the AR_MUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

AR_AUI_MUX_5 is the adapter-based variant of the generic multiplexer for data type REAL. Unlike AR_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AR_AUI_MUX_5 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (AR adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (AR adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (AR adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (AR adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (AR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN5:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AR_AUI_MUX`) – shared across all port counts of this block family (AR_AUI_MUX_2 … AR_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 32-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AR_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

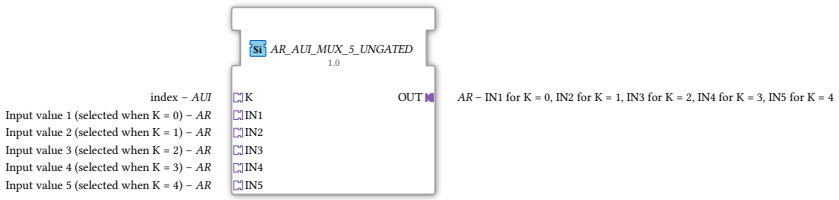
- `AR_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AR_AUI_MUX_5` carries the multiplexer logic of the `AR_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AR_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AR_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AR_AUI_MUX_5_UNGATED is the adapter-based variant of the generic multiplexer for data type REAL. Unlike AR_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AR_AUI_MUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (AR adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (AR adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (AR adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (AR adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (AR adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AR adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN5:

1. The current value of K.D1 determines which of the 5 input adapters (IN1 … IN5) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AR_AUI_MUX) – shared across all port counts of this block family (AR_AUI_MUX_2 … AR_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN5) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 32-bit floating-point value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AR_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AR_AUI_MUX_5`**: The gated variant – updates the output only on an actual value change.

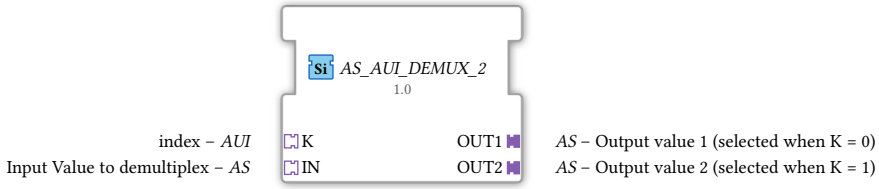
Conclusion

`AR_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AR_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AS_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AS_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `SINT`. Unlike `AS_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.

- **IN** (Socket): input adapter carrying the 8-bit signed integer to forward, routed to exactly one of the outputs (AS adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AS adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_DEMUX`) – shared across all port counts of this block family (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit signed integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AS_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

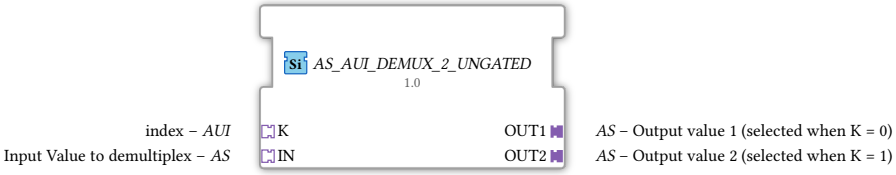
- **`AS_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AS_AUI_DEMUX_2` carries the demultiplexer logic of the `AS_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AS_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `SINT`. Unlike `AS_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 8-bit signed integer to forward, routed to exactly one of the outputs (AS adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AS adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see “Change Detection” below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn’t changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_DEMUX`) – shared across all port counts of this block family (AS_AUI_DEMUX_2_UNGATED … AS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit signed integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AS_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

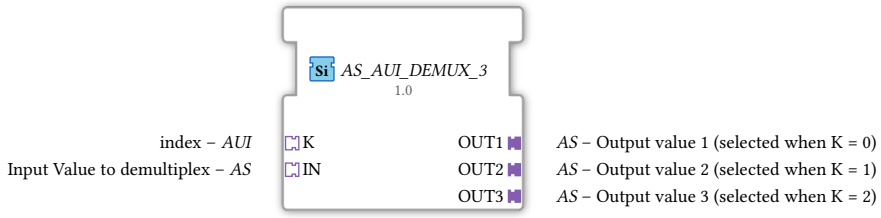
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AS_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AS_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AS_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AS_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `SINT`. Unlike `AS_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN** (Socket): input adapter carrying the 8-bit signed integer to forward, routed to exactly one of the outputs (AS adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AS adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AS adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_DEMUX`) – shared across all port counts of this block family (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit signed integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AS_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

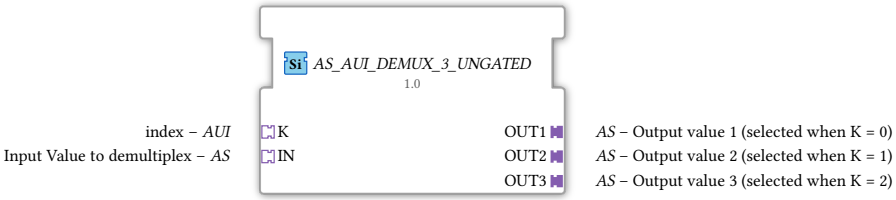
- **`AS_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AS_AUI_DEMUX_3` carries the demultiplexer logic of the `AS_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AS_AUI_DEMUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AS_AUI_DEMUX_3_UNGATED is the adapter-based variant of the generic demultiplexer for data type SINT. Unlike AS_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AS_AUI_DEMUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 8-bit signed integer to forward, routed to exactly one of the outputs (AS adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AS adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AS adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 3 output adapters (OUT1 … OUT3) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AS_AUI_DEMUX) – shared across all port counts of this block family (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) ⟶ determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit signed integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AS_DEMUX_3 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

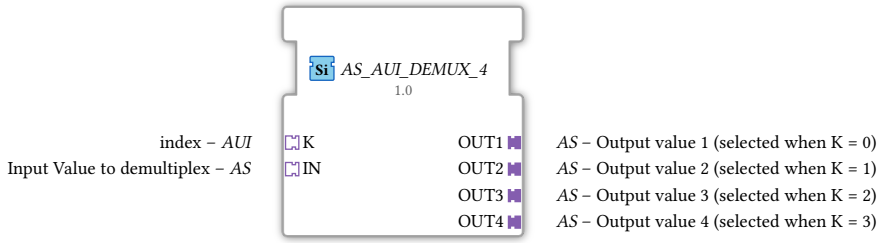
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **AS_AUI_DEMUX_3**: The gated variant – updates the output only on an actual value change.

Conclusion

AS_AUI_DEMUX_3_UNGATED carries the demultiplexer logic of the AS_DEMUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

AS_AUI_DEMUX_4 is the adapter-based variant of the generic demultiplexer for data type SINT. Unlike AS_DEMUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 8-bit signed integer to forward, routed to exactly one of the outputs (AS adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AS adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AS adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AS adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_DEMUX`) – shared across all port counts of this block family (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit signed integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AS_DEMUX_4 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

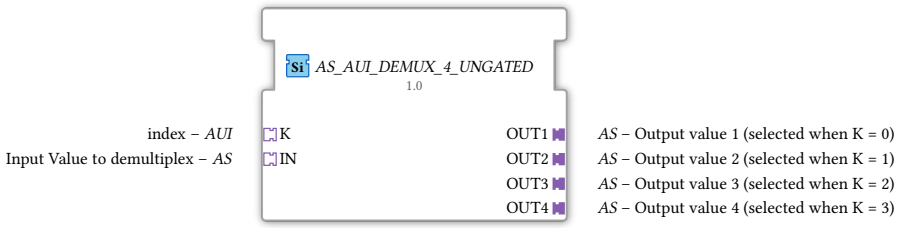
- **AS_AUI_DEMUX_4_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AS_AUI_DEMUX_4 carries the demultiplexer logic of the AS_DEMUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AS_AUI_DEMUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AS_AUI_DEMUX_4_UNGATED is the adapter-based variant of the generic demultiplexer for data type SINT. Unlike AS_DEMUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AS_AUI_DEMUX_4_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.

- **IN** (Socket): input adapter carrying the 8-bit signed integer to forward, routed to exactly one of the outputs (AS adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AS adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AS adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AS adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_DEMUX`) – shared across all port counts of this block family (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit signed integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AS_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

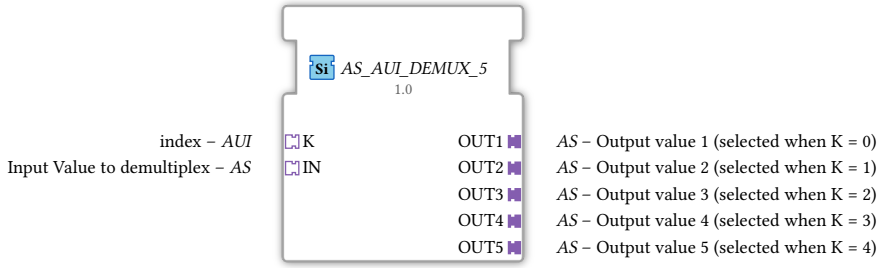
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AS_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AS_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AS_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AS_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `SINT`. Unlike `AS_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 8-bit signed integer to forward, routed to exactly one of the outputs (AS adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AS adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AS adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AS adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AS adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see “Change Detection” below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn’t changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_DEMUX`) – shared across all port counts of this block family (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit signed integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AS_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

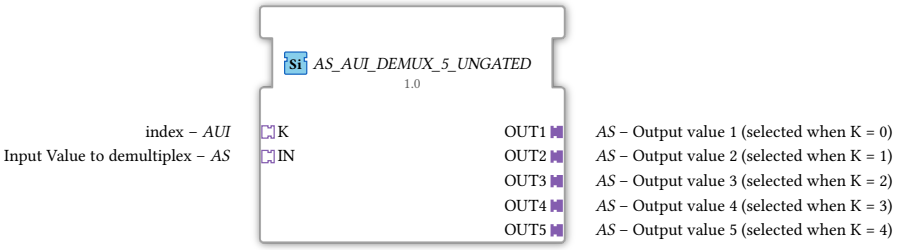
- **`AS_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AS_AUI_DEMUX_5` carries the demultiplexer logic of the `AS_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_AUI_DEMUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AS_AUI_DEMUX_5_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `SINT`. Unlike `AS_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data

input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_DEMUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 8-bit signed integer to forward, routed to exactly one of the outputs (AS adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AS adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AS adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AS adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AS adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_DEMUX`) – shared across all port counts of this block family (AS_AUI_DEMUX_2 … AS_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit signed integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AS_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

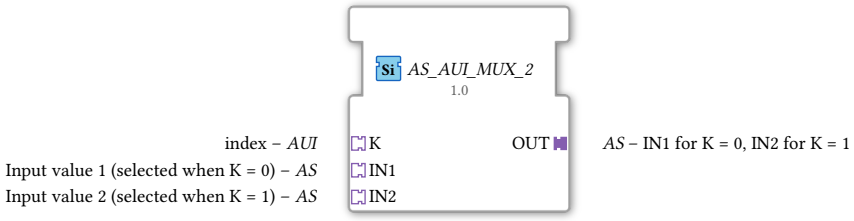
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AS_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AS_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AS_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AS_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `SINT`. Unlike `AS_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AS adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AS adapter type).

- **OUT** (Plug): output adapter, forwards whichever input the index selected (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_MUX`) – shared across all port counts of this block family (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 8-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AS_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

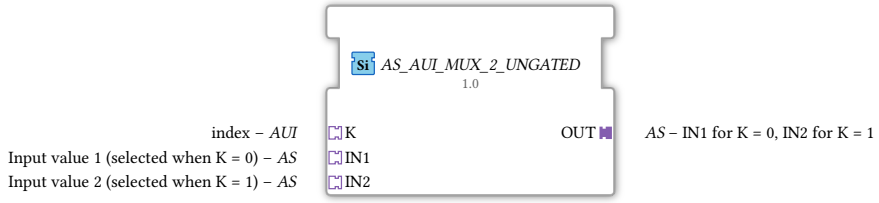
- **`AS_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AS_AUI_MUX_2` carries the multiplexer logic of the `AS_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AS_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AS_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `SINT`. Unlike `AS_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AS adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_MUX`) – shared across all port counts of this block family (AS_AUI_MUX_2_UNGATED … AS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 8-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AS_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

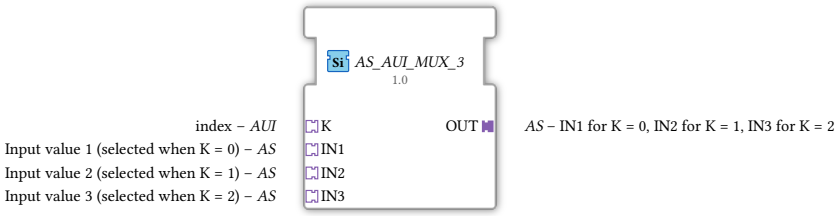
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AS_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AS_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AS_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_MUX_3



• — ○ • — ○ • — ○ •

Introduction

`AS_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `SINT`. Unlike `AS_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AS adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AS adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_MUX`) – shared across all port counts of this block family (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 8-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AS_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AS_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AS_AUI_MUX_3` carries the multiplexer logic of the `AS_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AS_AUI_MUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change

(e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AS_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `SINT`. Unlike `AS_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AS adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AS adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_MUX`) – shared across all port counts of this block family (AS_AUI_MUX_2 … AS_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 8-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AS_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

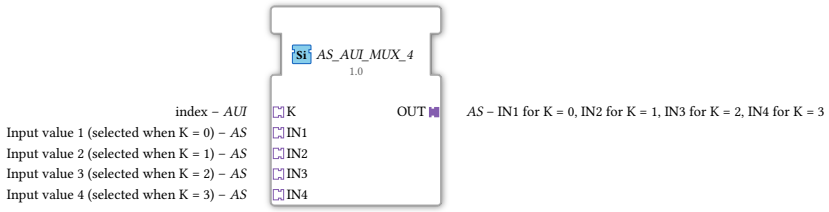
Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AS_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AS_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AS_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AS_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `SINT`. Unlike `AS_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AS adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AS adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AS adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_MUX`) – shared across all port counts of this block family (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 8-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AS_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

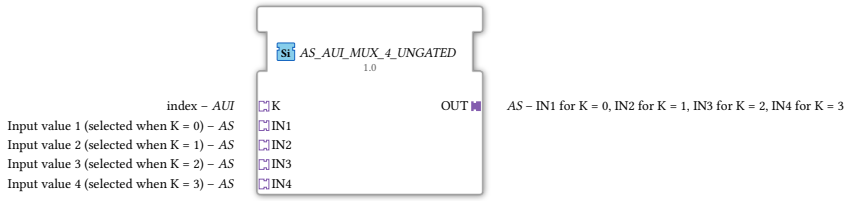
- **`AS_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AS_AUI_MUX_4` carries the multiplexer logic of the `AS_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AS_AUI_MUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AS_AUI_MUX_4_UNGATED is the adapter-based variant of the generic multiplexer for data type SINT. Unlike AS_MUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AS_AUI_MUX_4_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 4, passed through to OUT when the index is K = 0 (AS adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to OUT when the index is K = 1 (AS adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to OUT when the index is K = 2 (AS adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to OUT when the index is K = 3 (AS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN4:

1. The current value of K.D1 determines which of the 4 input adapters (IN1 … IN4) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AS_AUI_MUX) – shared across all port counts of this block family (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN4) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 8-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AS_MUX_4 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

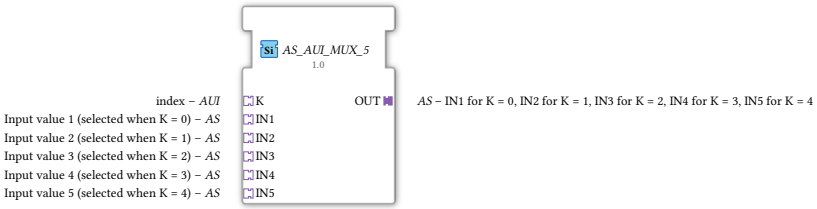
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **AS_AUI_MUX_4**: The gated variant – updates the output only on an actual value change.

Conclusion

AS_AUI_MUX_4_UNGATED carries the multiplexer logic of the AS_MUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

AS_AUI_MUX_5 is the adapter-based variant of the generic multiplexer for data type SINT. Unlike AS_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AS adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AS adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AS adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AS adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_MUX`) – shared across all port counts of this block family (AS_AUI_MUX_2 … AS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 8-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AS_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

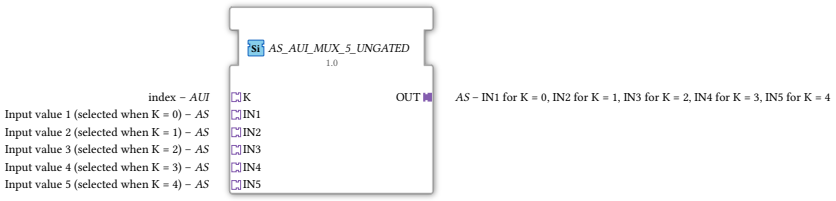
- `AS_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AS_AUI_MUX_5` carries the multiplexer logic of the `AS_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AS_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AS_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AS_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `SINT`. Unlike `AS_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AS_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AS adapter type).

- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AS adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AS adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AS adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AS_AUI_MUX`) – shared across all port counts of this block family (AS_AUI_MUX_2 … AS_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 8-bit signed integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AS_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- `AS_AUI_MUX_5`: The gated variant – updates the output only on an actual value change.

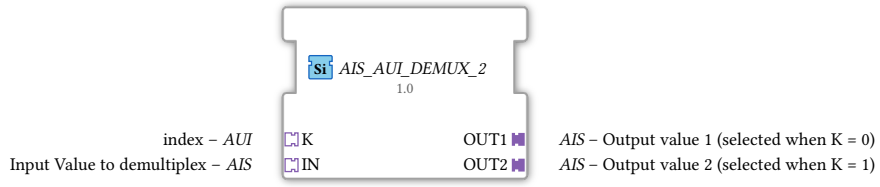
Conclusion

`AS_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AS_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AIS_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AIS_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `STRING`. Unlike `AIS_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (8-bit characters) to forward, routed to exactly one of the outputs (AIS adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AIS adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_DEMUX`) – shared across all port counts of this block family (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (8-bit characters) to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIS_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

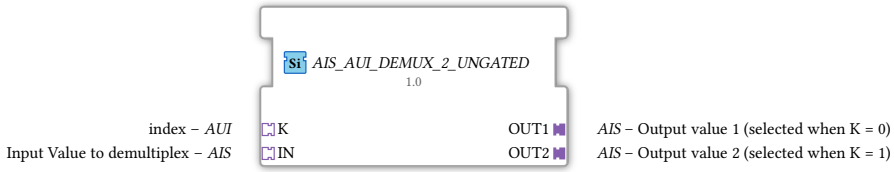
- **`AIS_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIS_AUI_DEMUX_2` carries the demultiplexer logic of the `AIS_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AIS_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `STRING`. Unlike `AIS_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (8-bit characters) to forward, routed to exactly one of the outputs (AIS adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AIS adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_DEMUX`) – shared across all port counts of this block family (AIS_AUI_DEMUX_2_UNGATED … AIS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (8-bit characters) to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIS_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

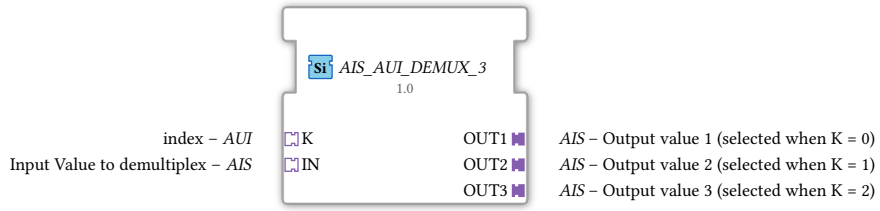
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AIS_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIS_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AIS_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AIS_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `STRING`. Unlike `AIS_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (8-bit characters) to forward, routed to exactly one of the outputs (AIS adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AIS adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AIS adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_DEMUX`) – shared across all port counts of this block family (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (8-bit characters) to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIS_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

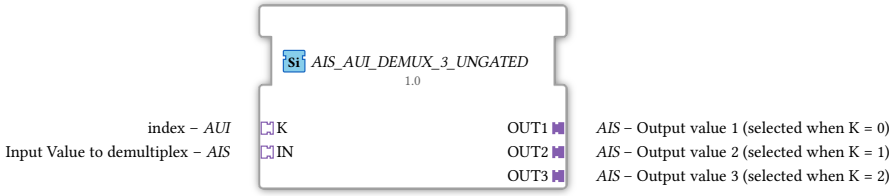
- **`AIS_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIS_AUI_DEMUX_3` carries the demultiplexer logic of the `AIS_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AIS_AUI_DEMUX_3_UNGATED is the adapter-based variant of the generic demultiplexer for data type STRING. Unlike AIS_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AIS_AUI_DEMUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the character string (8-bit characters) to forward, routed to exactly one of the outputs (AIS adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AIS adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AIS adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 3 output adapters (OUT1 … OUT3) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AIS_AUI_DEMUX) – shared across all port counts of this block family (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (8-bit characters) to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIS_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

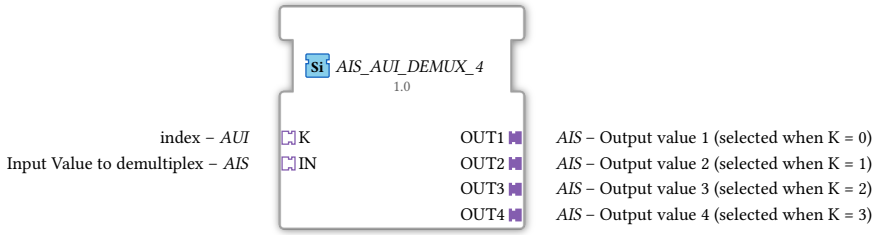
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AIS_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIS_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `AIS_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AIS_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `STRING`. Unlike `AIS_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.

- **IN** (Socket): input adapter carrying the character string (8-bit characters) to forward, routed to exactly one of the outputs (AIS adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AIS adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AIS adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AIS adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_DEMUX`) – shared across all port counts of this block family (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (8-bit characters) to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIS_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AIS_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

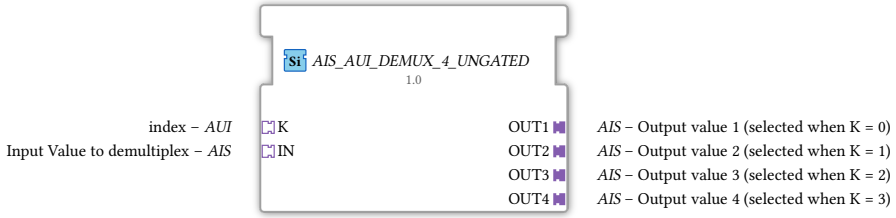
Conclusion

`AIS_AUI_DEMUX_4` carries the demultiplexer logic of the `AIS_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change

(e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AIS_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `STRING`. Unlike `AIS_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (8-bit characters) to forward, routed to exactly one of the outputs (AIS adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AIS adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AIS adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AIS adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_DEMUX`) – shared across all port counts of this block family (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (8-bit characters) to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIS_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

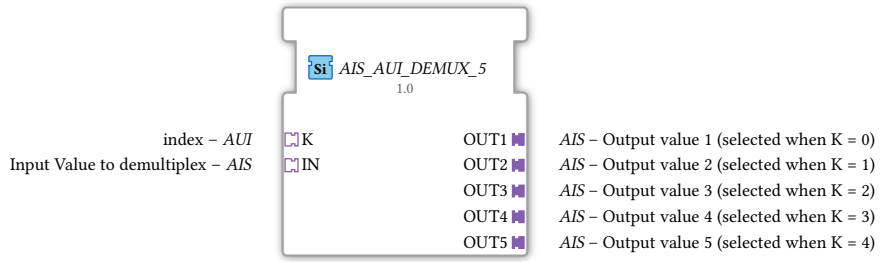
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AIS_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIS_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AIS_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AIS_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `STRING`. Unlike `AIS_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (8-bit characters) to forward, routed to exactly one of the outputs (AIS adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AIS adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AIS adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AIS adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AIS adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_DEMUX`) – shared across all port counts of this block family (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (8-bit characters) to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)

- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIS_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

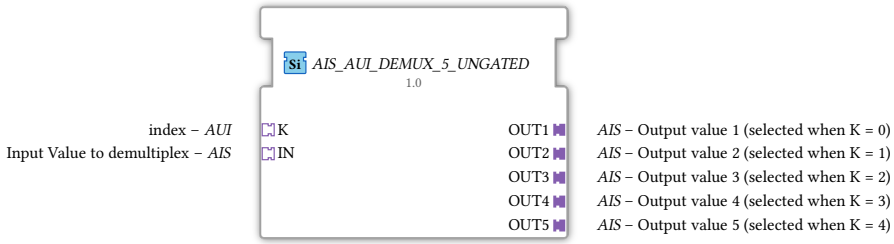
- **`AIS_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIS_AUI_DEMUX_5` carries the demultiplexer logic of the `AIS_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_AUI_DEMUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AIS_AUI_DEMUX_5_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `STRING`. Unlike `AIS_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_DEMUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (8-bit characters) to forward, routed to exactly one of the outputs (AIS adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AIS adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AIS adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AIS adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AIS adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_DEMUX`) – shared across all port counts of this block family (AIS_AUI_DEMUX_2 … AIS_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (8-bit characters) to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIS_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

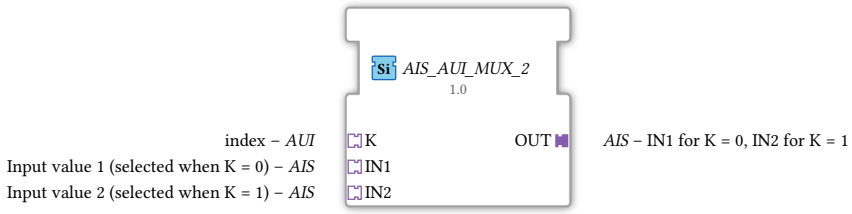
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AIS_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIS_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AIS_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AIS_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `STRING`. Unlike `AIS_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AIS adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AIS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_MUX`) – shared across all port counts of this block family (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 character string (8-bit characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIS_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

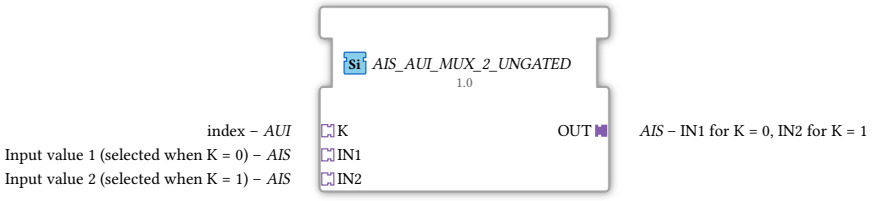
- **`AIS_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIS_AUI_MUX_2` carries the multiplexer logic of the `AIS_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_AUI_MUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AIS_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `STRING`. Unlike `AIS_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AIS adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AIS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_MUX`) – shared across all port counts of this block family (AIS_AUI_MUX_2_UNGATED … AIS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 character string (8-bit characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIS_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

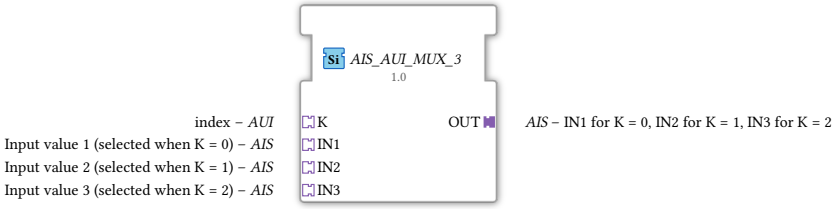
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AIS_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIS_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AIS_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_MUX_3



• – ○ • – ○ • – ○ •

Introduction

`AIS_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `STRING`. Unlike `AIS_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AIS adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AIS adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AIS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_MUX`) – shared across all port counts of this block family (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 character string (8-bit characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIS_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

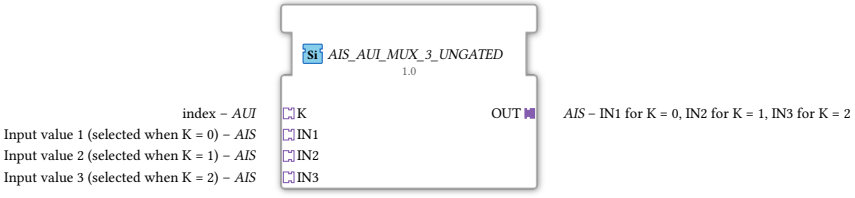
- **`AIS_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIS_AUI_MUX_3` carries the multiplexer logic of the `AIS_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_AUI_MUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AIS_AUI_MUX_3_UNGATED is the adapter-based variant of the generic multiplexer for data type STRING. Unlike AIS_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AIS_AUI_MUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (AIS adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (AIS adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (AIS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AIS_AUI_MUX) – shared across all port counts of this block family (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 character string (8-bit characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖ Comparison with Similar Blocks

Compare with `AIS_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AIS_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIS_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AIS_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AIS_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `STRING`. Unlike `AIS_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AIS adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AIS adapter type).

- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AIS adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AIS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_MUX`) – shared across all port counts of this block family (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 character string (8-bit characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIS_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

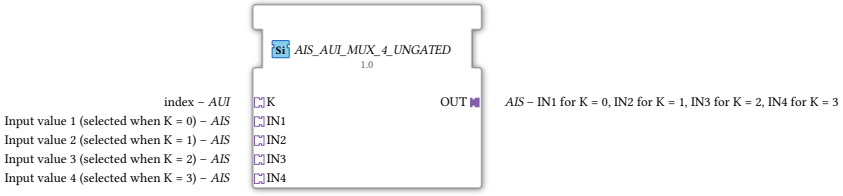
- **`AIS_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIS_AUI_MUX_4` carries the multiplexer logic of the `AIS_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIS_AUI_MUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• — ○ • — ○ • — ○ •

Introduction

`AIS_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `STRING`. Unlike `AIS_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AIS adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AIS adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AIS adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AIS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_MUX`) – shared across all port counts of this block family (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 character string (8-bit characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIS_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

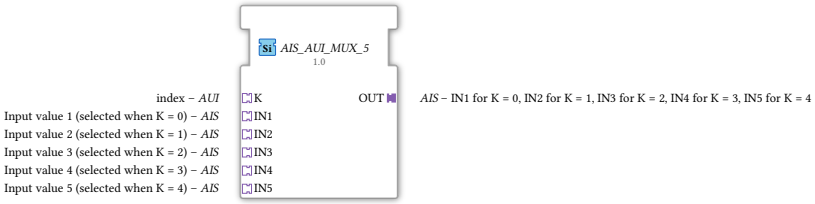
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AIS_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIS_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `AIS_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`AIS_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `STRING`. Unlike `AIS_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIS_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AIS adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AIS adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AIS adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AIS adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AIS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_MUX`) – shared across all port counts of this block family (AIS_AUI_MUX_2 … AIS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 character string (8-bit characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIS_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

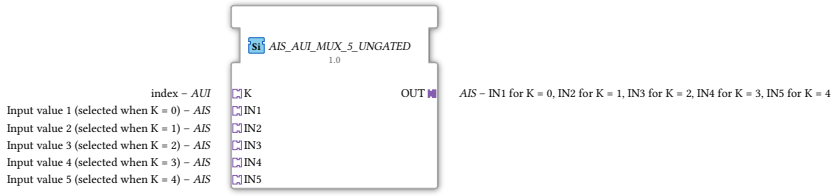
- `AIS_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AIS_AUI_MUX_5 carries the multiplexer logic of the AIS_MUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AIS_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of AIS_AUI_MUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AIS_AUI_MUX_5_UNGATED is the adapter-based variant of the generic multiplexer for data type STRING. Unlike AIS_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AIS_AUI_MUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (AIS adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (AIS adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (AIS adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (AIS adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (AIS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIS_AUI_MUX`) – shared across all port counts of this block family (AIS_AUI_MUX_2 … AIS_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 character string (8-bit characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIS_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AIS_AUI_MUX_5`**: The gated variant – updates the output only on an actual value change.

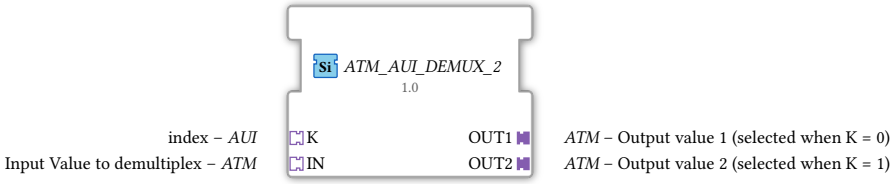
Conclusion

`AIS_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AIS_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ATM_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

ATM_AUI_DEMUX_2 is the adapter-based variant of the generic demultiplexer for data type TIME. Unlike ATM_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ATM_AUI_DEMUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the duration value to forward, routed to exactly one of the outputs (ATM adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (ATM adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 2 output adapters (OUT1 … OUT2) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_ATM_AUI_DEMUX) – shared across all port counts of this block family (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) ⟶ determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a duration value to up to 2 downstream consumers, selected via an adapter-carried index

- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with ATM_DEMUX_2 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

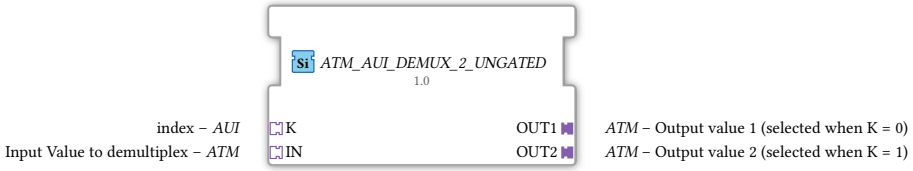
- **ATM_AUI_DEMUX_2_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

ATM_AUI_DEMUX_2 carries the demultiplexer logic of the ATM_DEMUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of ATM_AUI_DEMUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

ATM_AUI_DEMUX_2_UNGATED is the adapter-based variant of the generic demultiplexer for data type TIME. Unlike ATM_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ATM_AUI_DEMUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the duration value to forward, routed to exactly one of the outputs (ATM adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (ATM adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_DEMUX`) – shared across all port counts of this block family (ATM_AUI_DEMUX_2_UNGATED … ATM_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a duration value to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ATM_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

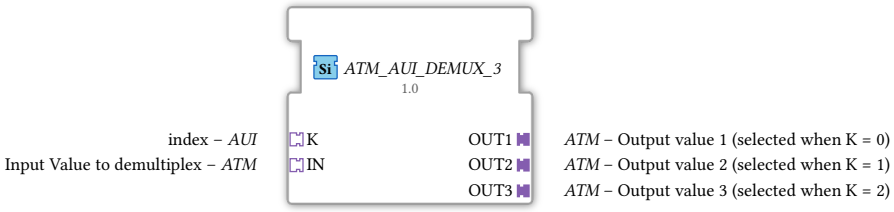
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ATM_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ATM_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `ATM_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`ATM_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `TIME`. Unlike `ATM_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ATM_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the duration value to forward, routed to exactly one of the outputs (ATM adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (ATM adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (ATM adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_DEMUX`) – shared across all port counts of this block family (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a duration value to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ATM_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

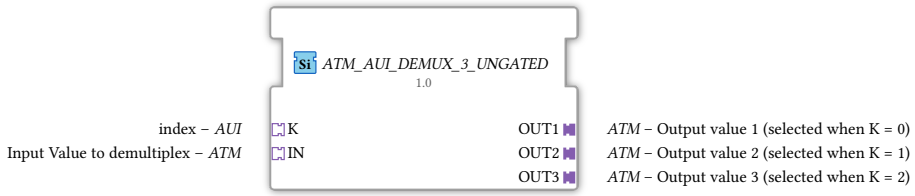
- **`ATM_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ATM_AUI_DEMUX_3` carries the demultiplexer logic of the `ATM_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ATM_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`ATM_AUI_DEMUX_3_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `TIME`. Unlike `ATM_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ATM_AUI_DEMUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the duration value to forward, routed to exactly one of the outputs (ATM adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (ATM adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (ATM adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_DEMUX`) – shared across all port counts of this block family (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a duration value to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with ATM_DEMUX_3 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

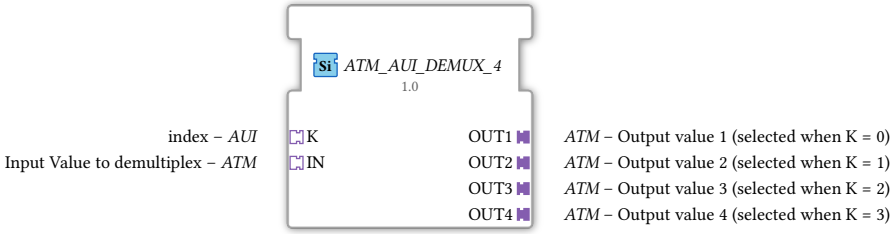
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **ATM_AUI_DEMUX_3**: The gated variant – updates the output only on an actual value change.

Conclusion

ATM_AUI_DEMUX_3_UNGATED carries the demultiplexer logic of the ATM_DEMUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

ATM_AUI_DEMUX_4 is the adapter-based variant of the generic demultiplexer for data type TIME. Unlike ATM_DEMUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ATM_AUI_DEMUX_4 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the duration value to forward, routed to exactly one of the outputs (ATM adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (ATM adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (ATM adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (ATM adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_DEMUX`) – shared across all port counts of this block family (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a duration value to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖ Comparison with Similar Blocks

Compare with `ATM_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

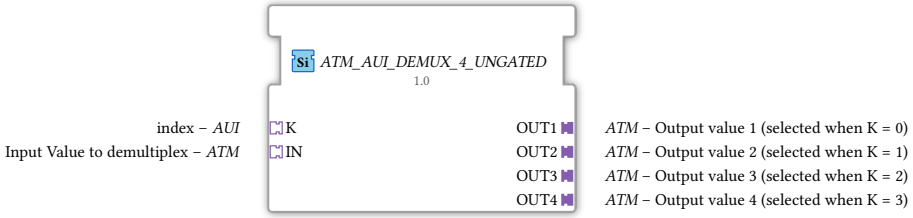
- **`ATM_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ATM_AUI_DEMUX_4` carries the demultiplexer logic of the `ATM_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ATM_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`ATM_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `TIME`. Unlike `ATM_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ATM_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the duration value to forward, routed to exactly one of the outputs (ATM adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (ATM adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (ATM adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (ATM adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see “Change Detection” below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn’t changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_DEMUX`) – shared across all port counts of this block family (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a duration value to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ATM_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

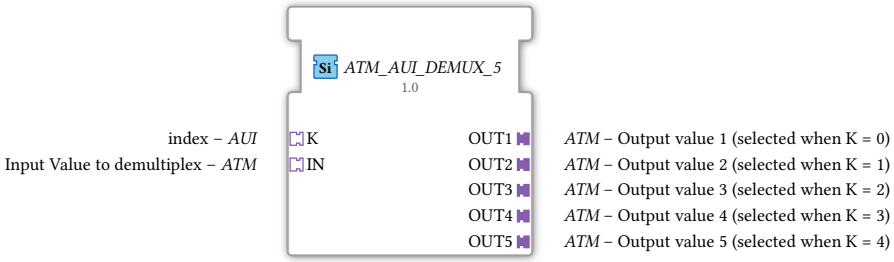
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ATM_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ATM_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `ATM_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_DEMUX_5



• – ○ • – ○ • – ○ •

Introduction

`ATM_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `TIME`. Unlike `ATM_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ATM_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the duration value to forward, routed to exactly one of the outputs (ATM adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (ATM adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (ATM adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (ATM adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (ATM adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_DEMUX`) – shared across all port counts of this block family (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a duration value to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ATM_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

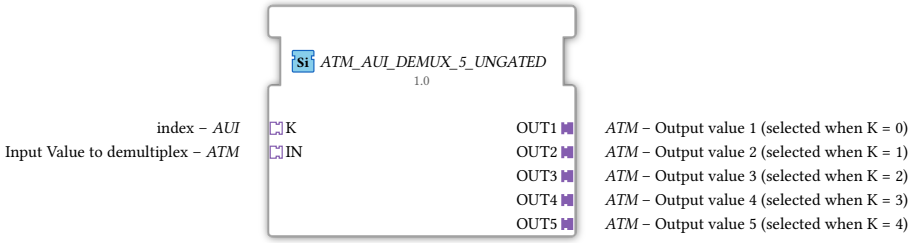
- `ATM_AUI_DEMUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

ATM_AUI_DEMUX_5 carries the demultiplexer logic of the ATM_DEMUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of ATM_AUI_DEMUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

ATM_AUI_DEMUX_5_UNGATED is the adapter-based variant of the generic demultiplexer for data type TIME. Unlike ATM_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ATM_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the duration value to forward, routed to exactly one of the outputs (ATM adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (ATM adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (ATM adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (ATM adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (ATM adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_DEMUX`) – shared across all port counts of this block family (ATM_AUI_DEMUX_2 … ATM_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a duration value to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `ATM_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

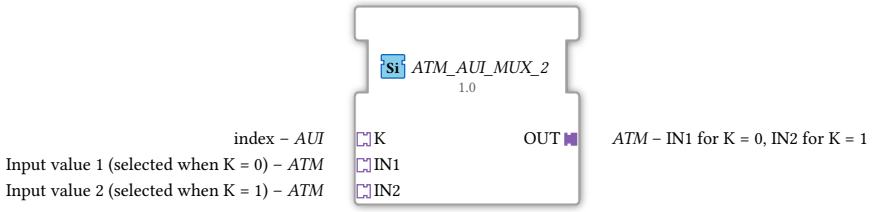
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`ATM_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ATM_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `ATM_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`ATM_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `TIME`. Unlike `ATM_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index

be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ATM_AUI_MUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 2, passed through to OUT when the index is K = 0 (ATM adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to OUT when the index is K = 1 (ATM adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN2:

1. The current value of K.D1 determines which of the 2 input adapters (IN1 … IN2) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_ATM_AUI_MUX) – shared across all port counts of this block family (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN2) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 duration value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with ATM_MUX_2 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

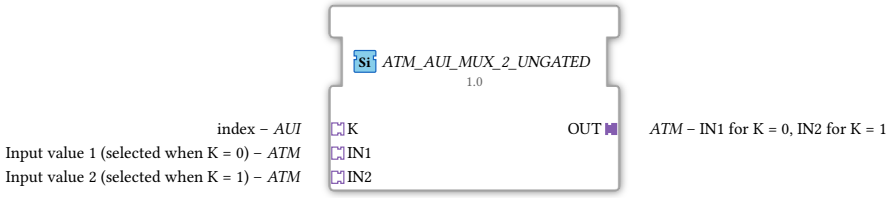
- **ATM_AUI_MUX_2_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

ATM_AUI_MUX_2 carries the multiplexer logic of the ATM_MUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of ATM_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

ATM_AUI_MUX_2_UNGATED is the adapter-based variant of the generic multiplexer for data type TIME. Unlike ATM_MUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ATM_AUI_MUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.

- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (ATM adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (ATM adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_MUX`) – shared across all port counts of this block family (ATM_AUI_MUX_2_UNGATED … ATM_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 duration value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ATM_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

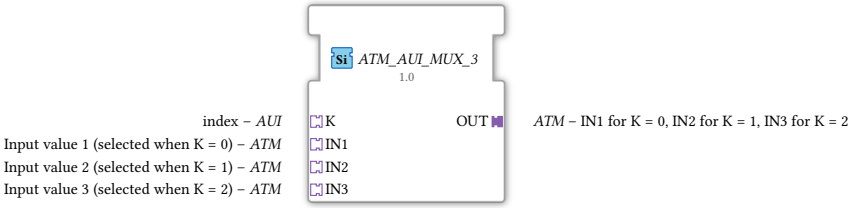
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`ATM_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ATM_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `ATM_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

ATM_AUI_MUX_3 is the adapter-based variant of the generic multiplexer for data type TIME. Unlike ATM_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ATM_AUI_MUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (ATM adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (ATM adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (ATM adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_ATM_AUI_MUX) – shared across all port counts of this block family (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 duration value sources through a centrally managed adapter index

- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ATM_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

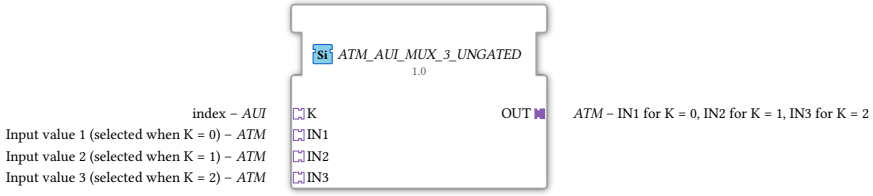
- **`ATM_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ATM_AUI_MUX_3` carries the multiplexer logic of the `ATM_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ATM_AUI_MUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`ATM_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `TIME`. Unlike `ATM_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ATM_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (ATM adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (ATM adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (ATM adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_MUX`) – shared across all port counts of this block family (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 duration value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ATM_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

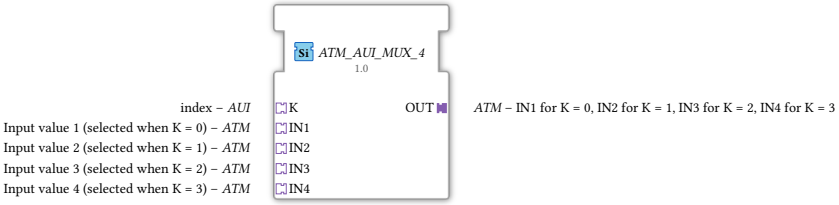
Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`ATM_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`ATM_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `ATM_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_MUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`ATM_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `TIME`. Unlike `ATM_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ATM_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (ATM adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (ATM adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (ATM adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (ATM adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_MUX`) – shared across all port counts of this block family (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 duration value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ATM_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`ATM_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ATM_AUI_MUX_4` carries the multiplexer logic of the `ATM_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ATM_AUI_MUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`ATM_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `TIME`. Unlike `ATM_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ATM_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (ATM adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (ATM adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (ATM adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (ATM adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_MUX`) – shared across all port counts of this block family (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 duration value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with ATM_MUX_4 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

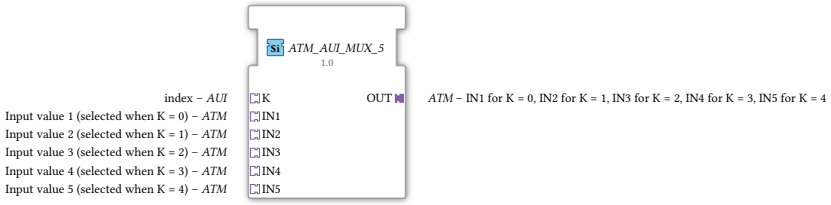
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **ATM_AUI_MUX_4**: The gated variant – updates the output only on an actual value change.

Conclusion

ATM_AUI_MUX_4_UNGATED carries the multiplexer logic of the ATM_MUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

ATM_AUI_MUX_5 is the adapter-based variant of the generic multiplexer for data type TIME. Unlike ATM_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, ATM_AUI_MUX_5 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 5, passed through to OUT when the index is K = 0 (ATM adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to OUT when the index is K = 1 (ATM adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to OUT when the index is K = 2 (ATM adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to OUT when the index is K = 3 (ATM adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to OUT when the index is K = 4 (ATM adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_MUX`) – shared across all port counts of this block family (ATM_AUI_MUX_2 … ATM_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 duration value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ATM_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

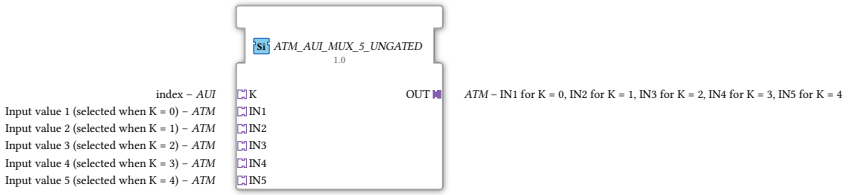
- `ATM_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`ATM_AUI_MUX_5` carries the multiplexer logic of the `ATM_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

ATM_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `ATM_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• — ○ • — ○ • — ○ •

Introduction

`ATM_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `TIME`. Unlike `ATM_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `ATM_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (ATM adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (ATM adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (ATM adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (ATM adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (ATM adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (ATM adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_ATM_AUI_MUX`) – shared across all port counts of this block family (ATM_AUI_MUX_2 … ATM_AUI_MUX_5_UNGATED).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 duration value sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `ATM_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- `ATM_AUI_MUX_5`: The gated variant – updates the output only on an actual value change.

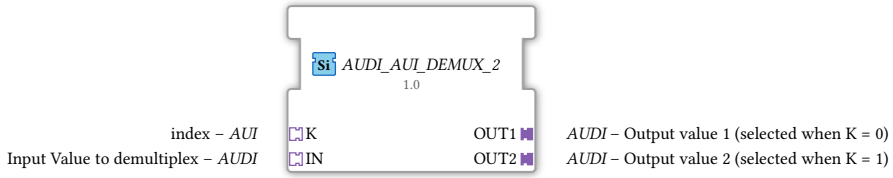
Conclusion

`ATM_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `ATM_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUDI_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AUDI_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `UDINT`. Unlike `AUDI_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit unsigned integer to forward, routed to exactly one of the outputs (AUDI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AUDI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_DEMUX`) – shared across all port counts of this block family (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit unsigned integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUDI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

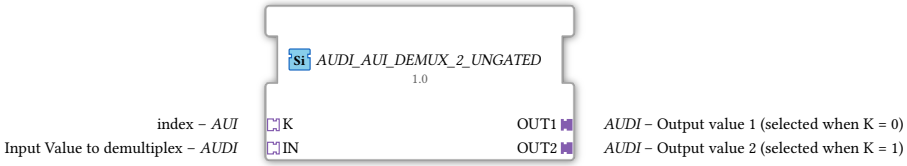
- **`AUDI_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AUDI_AUI_DEMUX_2 carries the demultiplexer logic of the AUDI_DEMUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUDI_AUI_DEMUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AUDI_AUI_DEMUX_2_UNGATED is the adapter-based variant of the generic demultiplexer for data type UDINT. Unlike AUDI_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUDI_AUI_DEMUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 32-bit unsigned integer to forward, routed to exactly one of the outputs (AUDI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AUDI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 2 output adapters (OUT1 … OUT2) is currently selected.

2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_DEMUX`) – shared across all port counts of this block family (AUDI_AUI_DEMUX_2_UNGATED … AUDI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit unsigned integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUDI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

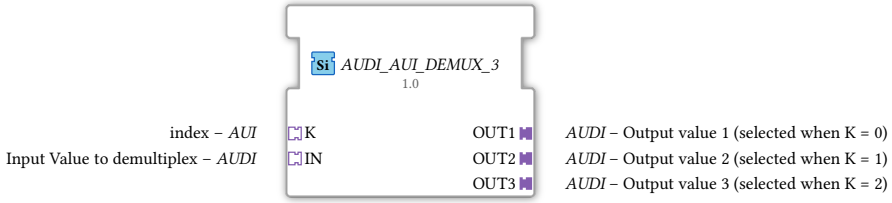
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUDI_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUDI_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AUDI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_DEMUX_3



• – ○ • – ○ • – ○ •

Introduction

`AUDI_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `UDINT`. Unlike `AUDI_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit unsigned integer to forward, routed to exactly one of the outputs (AUDI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AUDI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AUDI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_DEMUX`) – shared across all port counts of this block family (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit unsigned integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AUDI_DEMUX_3 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

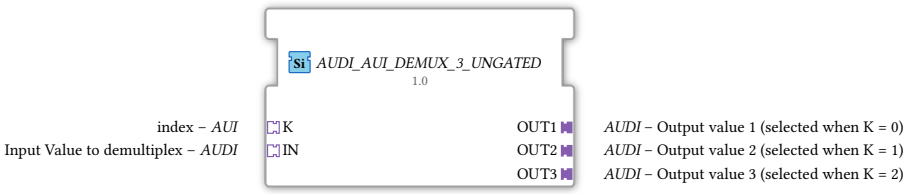
- **AUDI_AUI_DEMUX_3_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AUDI_AUI_DEMUX_3 carries the demultiplexer logic of the AUDI_DEMUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUDI_AUI_DEMUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AUDI_AUI_DEMUX_3_UNGATED is the adapter-based variant of the generic demultiplexer for data type UDINT. Unlike AUDI_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUDI_AUI_DEMUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.

- **IN** (Socket): input adapter carrying the 32-bit unsigned integer to forward, routed to exactly one of the outputs (AUDI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AUDI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AUDI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_DEMUX`) – shared across all port counts of this block family (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit unsigned integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUDI_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUDI_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUDI_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `AUDI_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AUDI_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `UDINT`. Unlike `AUDI_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit unsigned integer to forward, routed to exactly one of the outputs (AUDI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AUDI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AUDI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AUDI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.

- Generic implementation (`GEN_AUDI_AUI_DEMUX`) – shared across all port counts of this block family (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit unsigned integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUDI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

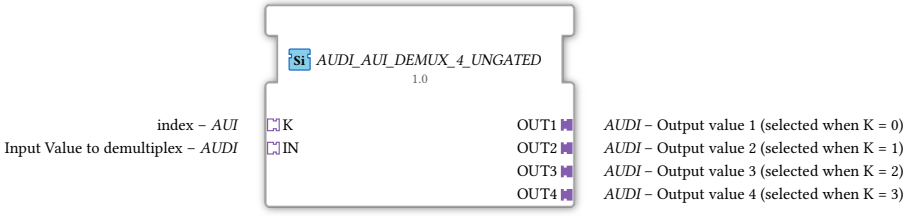
- **`AUDI_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUDI_AUI_DEMUX_4` carries the demultiplexer logic of the `AUDI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AUDI_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `UDINT`. Unlike `AUDI_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit unsigned integer to forward, routed to exactly one of the outputs (AUDI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AUDI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AUDI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AUDI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_DEMUX`) – shared across all port counts of this block family (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit unsigned integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUDI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

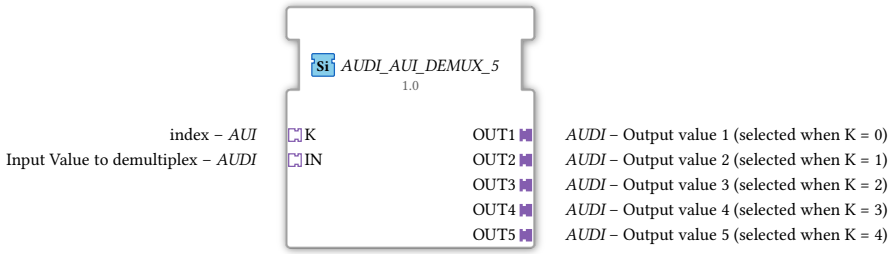
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUDI_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUDI_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AUDI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AUDI_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `UDINT`. Unlike `AUDI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit unsigned integer to forward, routed to exactly one of the outputs (AUDI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AUDI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AUDI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AUDI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AUDI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_DEMUX`) – shared across all port counts of this block family (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit unsigned integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUDI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

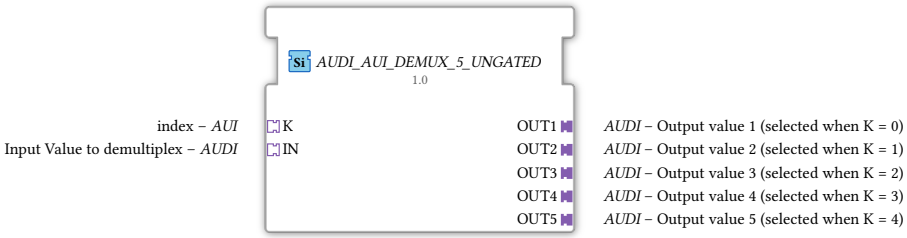
- **`AUDI_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUDI_AUI_DEMUX_5` carries the demultiplexer logic of the `AUDI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_AUI_DEMUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AUDI_AUI_DEMUX_5_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `UDINT`. Unlike `AUDI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_DEMUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 32-bit unsigned integer to forward, routed to exactly one of the outputs (AUDI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AUDI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AUDI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AUDI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AUDI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.

- Generic implementation (`GEN_AUDI_AUI_DEMUX`) – shared across all port counts of this block family (AUDI_AUI_DEMUX_2 … AUDI_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 32-bit unsigned integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUDI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

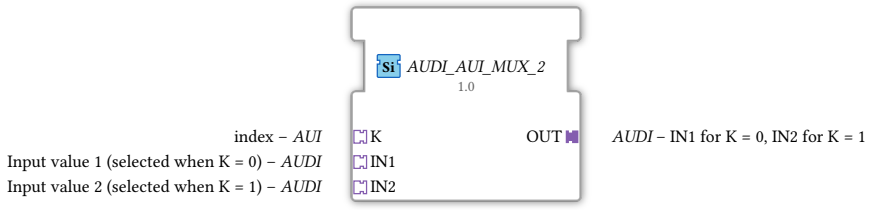
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUDI_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUDI_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AUDI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AUDI_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `UDINT`. Unlike `AUDI_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AUDI adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AUDI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_MUX`) – shared across all port counts of this block family (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 32-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUDI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

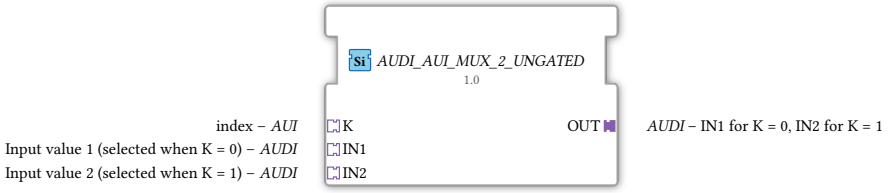
- **`AUDI_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUDI_AUI_MUX_2` carries the multiplexer logic of the `AUDI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_MUX_2_UNGATED

i UNGATED variant: This block is the ungated version of AUDI_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AUDI_AUI_MUX_2_UNGATED is the adapter-based variant of the generic multiplexer for data type UDINT. Unlike AUDI_MUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUDI_AUI_MUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 2, passed through to OUT when the index is K = 0 (AUDI adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to OUT when the index is K = 1 (AUDI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1...IN2:

1. The current value of K.D1 determines which of the 2 input adapters (IN1 ... IN2) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_MUX`) – shared across all port counts of this block family (AUDI_AUI_MUX_2_UNGATED … AUDI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 32-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUDI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

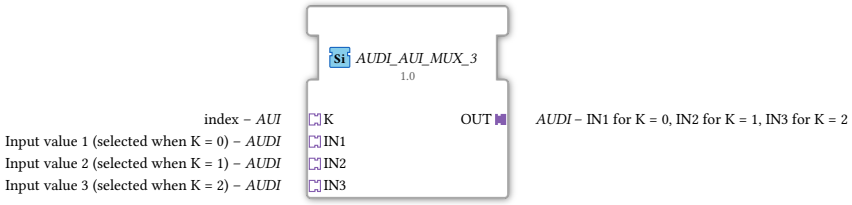
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AUDI_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUDI_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AUDI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_MUX_3



• – ○ • – ○ • – ○ •

Introduction

`AUDI_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `UDINT`. Unlike `AUDI_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AUDI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AUDI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AUDI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_MUX`) – shared across all port counts of this block family (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 32-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUDI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

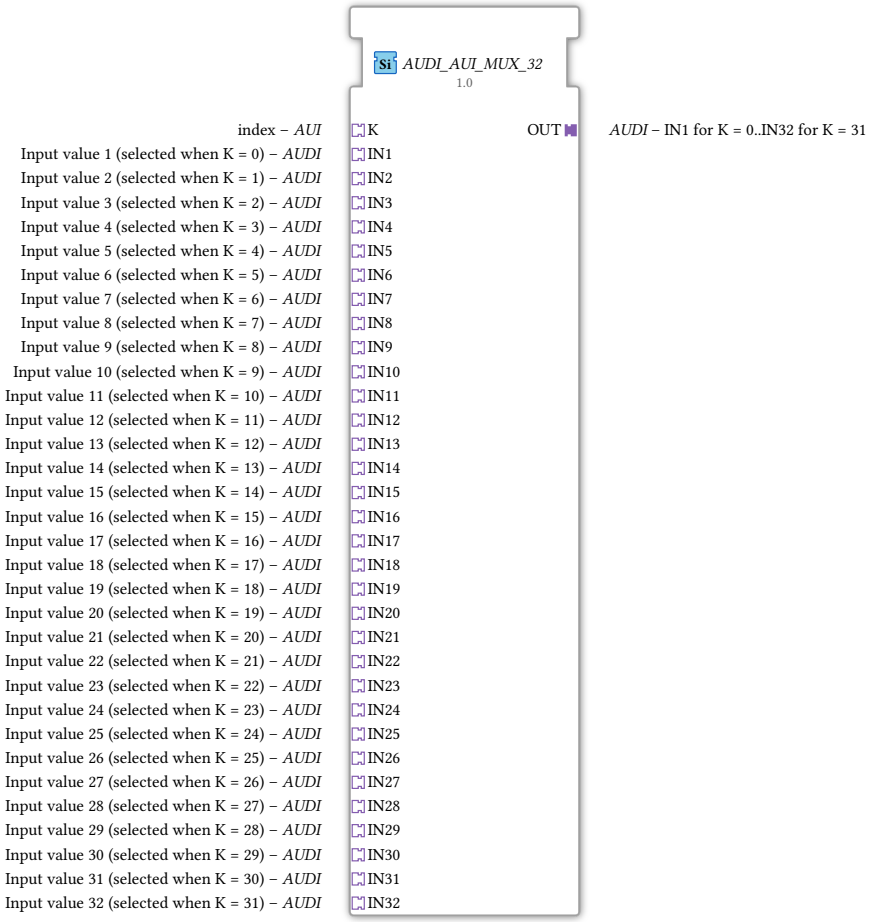
Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AUDI_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUDI_AUI_MUX_3` carries the multiplexer logic of the `AUDI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_MUX_32



• – ○ • – ○ • – ○ •

Introduction

AUDI_AUI_MUX_32 is a generic function block that implements a 32-channel adapter multiplexer. It selects one of 32 unidirectional AUDI adapter inputs (IN1 to IN32) based on an index value received via a dedicated AUI adapter input (K) and forwards the selected data to the unidirectional AUDI adapter output (OUT). The output is only updated when the value actually changes, avoiding unnecessary retransmissions.

This block is part of the adapter-based selection pattern and is designed for applications where multiple data sources must be routed to a single consumer under dynamic index control.

Interface Structure

Event Inputs

The function block has **no event inputs**. All data transfer is handled through adapter connections.

Event Outputs

The function block has **no event outputs**.

Data Inputs

There are **no direct data inputs**. All input data is provided through the socket adapters.

Data Outputs

There are **no direct data outputs**. All output data is provided through the plug adapter.

Adapters

Adapter	Direction	Type	Description
`OUT`	Plug	`adapter::types::unidirectional::AUDI`	Output adapter carrying the selected input value.
`K`	Socket	`adapter::types::unidirectional::AUI`	Index input (0 to 31) selecting which of the 32 inputs is routed to the output.
`IN1`	Socket	`adapter::types::unidirectional::AUDI`	Input value 1 (selected when K = 0).
`IN2`	Socket	`adapter::types::unidirectional::AUDI`	Input value 2 (selected when K = 1).
`IN3`	Socket	`adapter::types::unidirectional::AUDI`	Input value 3 (selected when K = 2).
`IN4`	Socket	`adapter::types::unidirectional::AUDI`	Input value 4 (selected when K = 3).
`IN5`	Socket	`adapter::types::unidirectional::AUDI`	Input value 5 (selected when K = 4).
`IN6`	Socket	`adapter::types::unidirectional::AUDI`	Input value 6 (selected when K = 5).
`IN7`	Socket	`adapter::types::unidirectional::AUDI`	Input value 7 (selected when K = 6).
`IN8`	Socket	`adapter::types::unidirectional::AUDI`	Input value 8 (selected when K = 7).
`IN9`	Socket	`adapter::types::unidirectional::AUDI`	Input value 9 (selected when K = 8).
`IN10`	Socket	`adapter::types::unidirectional::AUDI`	Input value 10 (selected when K = 9).
`IN11`	Socket	`adapter::types::unidirectional::AUDI`	Input value 11 (selected when K = 10).
`IN12`	Socket	`adapter::types::unidirectional::AUDI`	Input value 12 (selected when K = 11).
`IN13`	Socket	`adapter::types::unidirectional::AUDI`	Input value 13 (selected when K = 12).
`IN14`	Socket	`adapter::types::unidirectional::AUDI`	Input value 14 (selected when K = 13).
`IN15`	Socket	`adapter::types::unidirectional::AUDI`	Input value 15 (selected when K = 14).
`IN16`	Socket	`adapter::types::unidirectional::AUDI`	Input value 16 (selected when K = 15).
`IN17`	Socket	`adapter::types::unidirectional::AUDI`	Input value 17 (selected when K = 16).
`IN18`	Socket	`adapter::types::unidirectional::AUDI`	Input value 18 (selected when K = 17).
`IN19`	Socket	`adapter::types::unidirectional::AUDI`	Input value 19 (selected when K = 18).
`IN20`	Socket	`adapter::types::unidirectional::AUDI`	Input value 20 (selected when K = 19).
`IN21`	Socket	`adapter::types::unidirectional::AUDI`	Input value 21 (selected when K = 20).
`IN22`	Socket	`adapter::types::unidirectional::AUDI`	Input value 22 (selected when K = 21).
`IN23`	Socket	`adapter::types::unidirectional::AUDI`	Input value 23 (selected when K = 22).
`IN24`	Socket	`adapter::types::unidirectional::AUDI`	Input value 24 (selected when K = 23).
`IN25`	Socket	`adapter::types::unidirectional::AUDI`	Input value 25 (selected when K = 24).
`IN26`	Socket	`adapter::types::unidirectional::AUDI`	Input value 26 (selected when K = 25).
`IN27`	Socket	`adapter::types::unidirectional::AUDI`	Input value 27 (selected when K = 26).
`IN28`	Socket	`adapter::types::unidirectional::AUDI`	Input value 28 (selected when K = 27).
`IN29`	Socket	`adapter::types::unidirectional::AUDI`	Input value 29 (selected when K = 28).

IN30	Socket	adapter::types::unidirectional::AUDI	Input value 30 (selected when K = 29).
IN31	Socket	adapter::types::unidirectional::AUDI	Input value 31 (selected when K = 30).
IN32	Socket	adapter::types::unidirectional::AUDI	Input value 32 (selected when K = 31).

Functionality

The block continuously monitors the index value received on the K socket. When the index changes (or a new value arrives), the corresponding input adapter data (IN1 to IN32) is routed to the OUT adapter. The output value is compared with the previously forwarded value; if a difference is detected, the output is updated. If the value remains identical, no update is performed, preventing unnecessary load on downstream components.

The multiplexing logic is generic – the block is marked as a generic function block (GEN_AUDI_AUI_MUX), meaning the actual adapter types and internal behavior are resolved at deployment time.

Technical Features

- **32-to-1 multiplexing** with unidirectional adapter interfaces.
- **Change-detection** on the output: the output is only updated when the selected value actually changes.
- **No event inputs** required – the block operates autonomously on incoming adapter data.
- **Index range**: 0 to 31, corresponding to IN1 to IN32.
- **Unidirectional data flow** – all adapters are unidirectional, simplifying the interface.
- **Generic FB type** – uses the eclipse 4diac generic mechanism for late type resolution.

State Overview

The block does not expose explicit internal states. It can be conceptually described by two implicit operating phases:

1. **Idle / Monitoring** – Waiting for a new index or input value change.

No persistent state or mode beyond the current selected value and its cached copy is maintained.

Application Scenarios

- **Signal routing in agricultural machinery** (as indicated by the source context) where one of many sensors must be connected to a display or controller.
- **Dynamic sensor selection** – choosing between 32 measurement channels based on a control index.
- **Data consolidation** – merging multiple producer adapters into a single consumer path.
- **Event-driven communication** – where downstream components should only be notified when a value really changes, to reduce bus traffic.

Comparison with Similar Blocks

Feature	AUDI_AUI_MUX_32	Conventional Data MUX
Interface	Fully adapter-based (AUDI/AUI)	Direct data inputs (ANY)
Change detection	Yes, only on real change	Typically always raises event
Number of channels	32	Variable (often 2, 4, 8)
Generic type	Yes (generic FB)	Usually typed
Event inputs	None (autonomous)	Often explicit trigger event

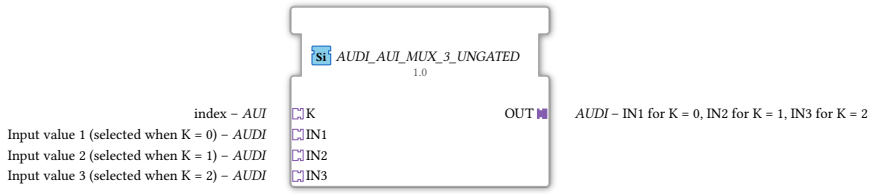
Compared to a classic IEC 61499 multiplexer FB (e.g., using data inputs and a selector input), this block is oriented towards adapter-based communication and provides inherent change filtering, making it better suited for distributed, event-driven systems.

Conclusion

AUDI_AUI_MUX_32 is a powerful and flexible adapter multiplexer for up to 32 unidirectional data sources. Its generic nature and change-detection behavior reduce unnecessary event traffic, while the adapter-based interface integrates seamlessly with modern 4diac IEC 61499 applications. It is especially useful in scenarios where a single consumer must be dynamically connected to a large number of producers.

AUDI_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUDI_AUI_MUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AUDI_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `UDINT`. Unlike `AUDI_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AUDI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AUDI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AUDI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` ... `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_MUX`) – shared across all port counts of this block family (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 32-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUDI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AUDI_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUDI_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AUDI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AUDI_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `UDINT`. Unlike `AUDI_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AUDI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AUDI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AUDI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AUDI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_MUX`) – shared across all port counts of this block family (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 32-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUDI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AUDI_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUDI_AUI_MUX_4` carries the multiplexer logic of the `AUDI_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_AUI_MUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AUDI_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `UDINT`. Unlike `AUDI_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AUDI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AUDI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AUDI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AUDI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_MUX`) – shared across all port counts of this block family (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 32-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUDI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AUDI_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUDI_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `AUDI_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`AUDI_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `UDINT`. Unlike `AUDI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AUDI adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AUDI adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AUDI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AUDI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AUDI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_MUX`) – shared across all port counts of this block family (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 32-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUDI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AUDI_AUI_MUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUDI_AUI_MUX_5` carries the multiplexer logic of the `AUDI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUDI_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AUDI_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `UDINT`. Unlike `AUDI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUDI_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AUDI adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AUDI adapter type).

- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AUDI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AUDI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AUDI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUDI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUDI_AUI_MUX`) – shared across all port counts of this block family (AUDI_AUI_MUX_2 … AUDI_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 32-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

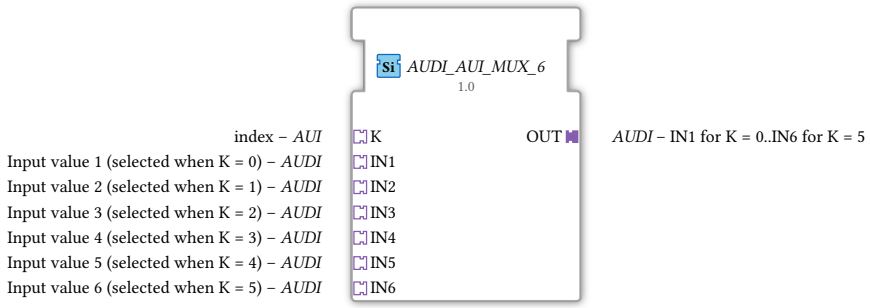
Compare with `AUDI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).
No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AUDI_AUI_MUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUDI_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AUDI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUDI_AUI_MUX_6



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUDI_AUI_MUX_6** function block is a generic 6-to-1 multiplexer specifically designed for use with unidirectional AUDI/AUI adapter types. It selects one of six input adapters (IN1…IN6) based on an index value provided via the K adapter, and forwards the selected data to a single output adapter (OUT). The block optimises communication by updating the output only when the value on the selected input actually changes, thus reducing unnecessary network traffic and processing overhead.

The block belongs to the 4diac IDE environment and follows the IEC 61499 standard. It is intended for scenarios where multiple data sources need to be routed to a common destination, with a dynamic selection mechanism.

Interface Structure

Event Inputs

None.

Event Outputs

- No event outputs.

Data Inputs

None.

Data Outputs

None.

Adapters

The block uses exclusively adapter interfaces to exchange data and control signals.

Plugs (Output):

- **OUT** (Type: `adapter::types::unidirectional::AUDI`) – The output adapter that carries the selected value. The value on this adapter is updated only when a change occurs on the currently selected input.

Sockets (Input):

- **K** (Type: `adapter::types::unidirectional::AUI`) – The index adapter. It supplies the selection index (0 to 5) that determines which of the six input adapters is routed to the output.
- **IN1** (Type: `adapter::types::unidirectional::AUDI`) – Input value 1, selected when K = 0.
- **IN2** (Type: `adapter::types::unidirectional::AUDI`) – Input value 2, selected when K = 1.
- **IN3** (Type: `adapter::types::unidirectional::AUDI`) – Input value 3, selected when K = 2.
- **IN4** (Type: `adapter::types::unidirectional::AUDI`) – Input value 4, selected when K = 3.
- **IN5** (Type: `adapter::types::unidirectional::AUDI`) – Input value 5, selected when K = 4.
- **IN6** (Type: `adapter::types::unidirectional::AUDI`) – Input value 6, selected when K = 5.

All adapters are unidirectional, meaning data flows in one direction: from the input sockets to the output plug.

Functionality

The AUDI_AUI_MUX_6 acts as a data selector. It continuously reads the value from the input adapter selected by the current index K. If the value on that input differs from the last value forwarded to the output, the block updates the output adapter. If the value remains unchanged, no output update or event is generated, thereby avoiding unnecessary communication.

The selection index K is provided through a dedicated adapter, allowing the selection to be changed dynamically during operation. The block is designed for use with adapters that carry both data and control information, but the multiplexing logic focuses solely on the data part of the adapter.

Technical Features

- **Generic Implementation** – The block is defined as a generic FB with the class name `GEN_AUDI_AUI_MUX`. This allows it to be instantiated with different underlying data types while preserving the same multiplexing behaviour.
- **Change-driven Output** – The output adapter is updated only when the value on the currently selected input changes. This reduces network load and improves efficiency in distributed automation systems.
- **Index-based Selection** – The selection index K is provided via an adapter interface, facilitating easy integration with other function blocks that can supply an AUI value.
- **No Data I/O** – The block has no dedicated data inputs or outputs; all data exchange occurs through the adapter interfaces.
- **Six Inputs, One Output** – Supports up to six different data sources, which can be selected arbitrarily.

State Overview

The block does not expose complex internal states. Its behaviour can be described by two main conditions:

1. **Value unchanged** – The value on the selected input matches the value previously forwarded to the output. In this state, no update is performed.
2. **Value changed** – The value on the selected input differs from the previously forwarded value. The output adapter is updated with the new value.

The selection index K can change at any time, and the block immediately re-evaluates the appropriate input.

Application Scenarios

- **Data Aggregation** – Combining multiple sensor readings (e.g., temperature, pressure, humidity) into a single data stream, where the current sensor is chosen via an index.
- **Mode Switching** – Switching between different parameter sets or control modes in an automation process, where the active set is determined by a selection signal.
- **Output Multiplexing** – Routing different data sources to a common display, communication module, or control output.
- **Redundancy Handling** – Selecting between redundant sensors or data paths based on a health or prioritisation index.

Comparison with Similar Blocks

Unlike a conventional multiplexer (e.g., a multi-input FB with an integer index), the AUDI_AUI_MUX_6 uses adapter interfaces for both the selection and the data. This makes it particularly suitable for distributed systems where data exchange occurs over network connections following the AUDI/AUI protocol. The key difference from a simple value multiplexer is the change-detection mechanism: most standard multiplexers unconditionally copy the selected input to the output, whereas this block suppresses redundant updates and only emits an event on actual change. This behaviour can significantly reduce network traffic in time-critical or bandwidth-limited environments.

Conclusion

The AUDI_AUI_MUX_6 is a specialised function block for multiplexing up to six unidirectional AUDI data streams to a single output, with an index-based selection via an AUI adapter. Its change-driven output update mechanism and generic design make it a valuable component in IEC 61499-based distributed automation systems where efficient data routing and minimal communication overhead are required. The block simplifies the integration of multiple data sources into a single sink, while maintaining high responsiveness.

AUDI_AUI_MUX_7



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUDI_AUI_MUX_7** is a generic multiplexer function block designed for unidirectional adapter-based selection of analog data values. It accepts up to **seven input adapters** (IN1 through IN7) and routes one of them to a single output adapter (OUT) based on an index value provided via the K adapter. The block is implemented as a generic FB with the class name `GEN_AUDI_AUI_MUX`, allowing flexible reuse in various IEC 61499 applications.

A distinctive feature of this block is its **change-driven output behavior**: the output adapter is only refreshed when the selected value actually changes. This minimizes unnecessary data traffic and event propagation in the system.

Interface Structure

Event Inputs

The block defines **no event inputs**. Activation and data flow are handled entirely through the adapter connections.

Event Outputs

The block defines **no event outputs**.

Data Inputs

The block has **no direct data inputs**. All input data is transported through adapter sockets.

Data Outputs

The block has **no direct data outputs**. All output data is transported through the adapter plug.

Adapters

Direction	Name	Type	Comment
Plug (Output)	OUT	adapter::types::unidirectional::AUDI	Delivers the selected input value.
Socket	K	adapter::types::unidirectional::AUI	Provides the index selection (0 to 6).
Socket	IN1	adapter::types::unidirectional::AUDI	Input value 1 (selected when K = 0).
Socket	IN2	adapter::types::unidirectional::AUDI	Input value 2 (selected when K = 1).
Socket	IN3	adapter::types::unidirectional::AUDI	Input value 3 (selected when K = 2).
Socket	IN4	adapter::types::unidirectional::AUDI	Input value 4 (selected when K = 3).
Socket	IN5	adapter::types::unidirectional::AUDI	Input value 5 (selected when K = 4).
Socket	IN6	adapter::types::unidirectional::AUDI	Input value 6 (selected when K = 5).
Socket	IN7	adapter::types::unidirectional::AUDI	Input value 7 (selected when K = 6).

Functionality

The AUDI_AUI_MUX_7 acts as a **7-to-1 multiplexer** operating on unidirectional adapter data. The index adapter **K** determines which of the seven input adapters (IN1…IN7) is forwarded to the output adapter **OUT**. The mapping is straightforward:

- K = 0 ⟶ OUT receives the value from IN1
- K = 1 ⟶ OUT receives the value from IN2
- …
- K = 6 ⟶ OUT receives the value from IN7

The core logic of the block performs a continuous comparison between the last transmitted output value and the newly selected input value. If the selected value differs from the previously output value, the output adapter is updated. If the value is identical to the last one, no update occurs, regardless of whether the index changed or the input data was refreshed.

This behavior ensures that downstream components only react to genuine value changes, avoiding redundant processing and reducing communication overhead.

Technical Features

- **Generic implementation** – The underlying generic FB (`GEN_AUDI_AUI_MUX`) can be instantiated for different adapter types or multiplexer widths by binding the appropriate adapter declarations.
- **Change detection** – Output update is suppressed when the selected value equals the last transmitted value.
- **Unidirectional adapters** – All adapter interfaces are of the `unidirectional` type, making the block suitable for one-way data flow scenarios without handshake or acknowledgment logic at the adapter level.
- **Seven input channels** – Specifically tailored for applications requiring up to seven selectable data sources.
- **Index range** – The K adapter must provide a value in the range 0 to 6; values outside this range are not defined and should be avoided.

State Overview

The block does not expose an explicit state machine, but its internal behavior can be modeled as follows:

State	Condition	Action
Idle	No index change and selected input value unchanged	No output update.
Update	Index changed **or** selected input value changed	New value written to OUT.

The block remains in the Idle state until one of the two trigger conditions occurs. After updating OUT, it returns to Idle until the next change is detected.

Application Scenarios

- **Sensor selection** – In agricultural or industrial machinery, the block can select between multiple analog sensors (e.g., temperature, pressure, fill level) based on a mode or channel selector.
- **Redundant measurement systems** – When several redundant sensors measure the same physical quantity, the mux can choose the active channel and only propagate the value when it actually differs from the last one.
- **Calibration and commissioning** – A service tool can switch between different test points or calibration inputs without generating unnecessary downstream events.
- **Multi-zone control** – In HVAC or process control, the block can route zone-specific values to a central controller, with event-driven updates only on real changes.

Comparison with Similar Blocks

Feature	AUDI_AUI_MUX_7	Conventional Data MUX	Event-driven MUX
Input channels	7 (adapter-based)	Typically 2/4/8 (data pins)	Variable
Interface type	Unidirectional adapters	Direct data I/O	Direct data + event I/O
Output update	Only on value change	Always on selection	On request event

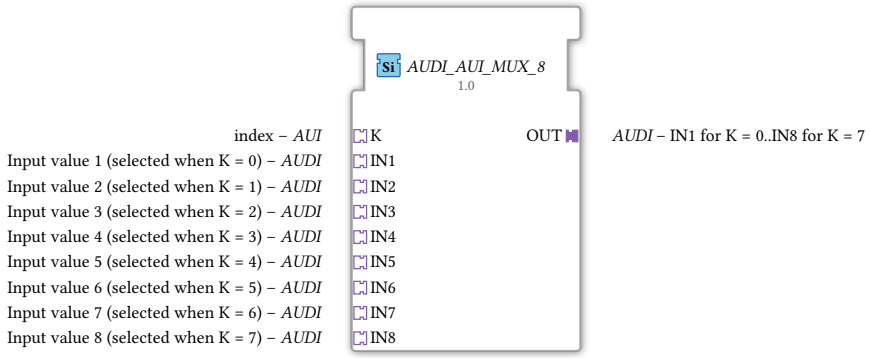
Generic capability	Yes (generic class)	No (fixed type)	Often fixed

Compared to a conventional multiplexer, AUDI_AUI_MUX_7 adds intelligent change detection and adapter-based connectivity, making it more suitable for distributed, event-driven IEC 61499 systems. Unlike a simple event-triggered mux, it suppresses unnecessary output updates, which can significantly reduce network load in cyclic communication environments.

Conclusion

The **AUDI_AUI_MUX_7** is a robust and efficient adapter-based multiplexer that combines the flexibility of a generic FB with a smart change-detection mechanism. Its unidirectional adapter interface and seven input channels make it an ideal building block for modern automation systems where data consistency and communication efficiency are critical. By updating the output only when the value actually changes, it helps to keep downstream logic simple and to minimize unnecessary processing overhead.

AUDI_AUI_MUX_8



• – ◦ • – ◦ • – ◦ •

Introduction

The AUDI_AUI_MUX_8 is a generic multiplexer function block designed to select one of eight input adapters (IN1 to IN8) and forward its value to a single output adapter (OUT). The selection is controlled by a dedicated index adapter (K). This FB is implemented as a generic type (GEN_AUDI_AUI_MUX), allowing flexible adaptation across different data types. A key feature is that the output adapter is only updated when the actual value changes.

Interface Structure

Event Inputs

None.

Event Outputs

No event outputs.

Data Inputs

None.

Data Outputs

None.

Adapters

Direction	Name	Type	Description
Plug (Output)	OUT	adapter::types::unidirectional::AUDI	Selected input value is forwarded to this output adapter.

Socket (Input)	K	adapter::types::unidirectional::AUI	Index selection adapter. Determines which input (IN1..IN8) is routed to OUT.
Socket (Input)	IN1	adapter::types::unidirectional::AUDI	Input value 1, selected when K = 0.
Socket (Input)	IN2	adapter::types::unidirectional::AUDI	Input value 2, selected when K = 1.
Socket (Input)	IN3	adapter::types::unidirectional::AUDI	Input value 3, selected when K = 2.
Socket (Input)	IN4	adapter::types::unidirectional::AUDI	Input value 4, selected when K = 3.
Socket (Input)	IN5	adapter::types::unidirectional::AUDI	Input value 5, selected when K = 4.
Socket (Input)	IN6	adapter::types::unidirectional::AUDI	Input value 6, selected when K = 5.
Socket (Input)	IN7	adapter::types::unidirectional::AUDI	Input value 7, selected when K = 6.
Socket (Input)	IN8	adapter::types::unidirectional::AUDI	Input value 8, selected when K = 7.

Functionality

The AUDI_AUI_MUX_8 function block acts as an 8-to-1 multiplexer for adapter-based data. It reads the index value from the K adapter and selects the corresponding input adapter (IN1 to IN8). The value of the selected input is then copied to the output adapter OUT. The selection follows a direct mapping:

- K = 0 ⟶ IN1
- K = 1 ⟶ IN2
- K = 2 ⟶ IN3
- K = 3 ⟶ IN4
- K = 4 ⟶ IN5
- K = 5 ⟶ IN6
- K = 6 ⟶ IN7
- K = 7 ⟶ IN8

The output is updated only when the value of the selected input differs from the current output value. If no change is detected, the OUT adapter remains unchanged and no update is performed. This behavior reduces unnecessary communication and processing overhead in the system.

Technical Features

- **Generic implementation**: The FB is based on the generic class GEN_AUDI_AUI_MUX, making it adaptable for different data types.
- **Adapter-based connectivity**: All inputs and the output use unidirectional adapter interfaces, ensuring clean and type-safe connections.
- **Eight selectable channels**: Supports up to 8 distinct input sources.
- **Single index input**: The selection is performed via a dedicated index adapter (K), simplifying control logic.

State Overview

The FB does not maintain a complex state machine. It operates in a purely combinatorial manner from the perspective of value selection. However, an internal state holds the last output value to enable change detection. When a new selection is made or an input value changes:

1. The index K is evaluated.
2. The corresponding input is read.
3. The new value is compared with the stored output value.
4. If different, the OUT adapter is updated.

Application Scenarios

- **Data source selection**: Switching between multiple sensor signals in automation systems.
- **Redundancy handling**: Selecting among redundant data sources based on availability or quality criteria.
- **Configuration switches**: Routing different parameter sets to a common processing unit.
- **Multi-zone monitoring**: Selecting zone-specific data for a central controller.

Comparison with Similar Blocks

Compared to standard multiplexer FBs (e.g., those with discrete data inputs), the AUDI_AUI_MUX_8 uses adapter interfaces instead of plain data pins. This allows it to connect to complex datatype structures and unidirectional communication channels. Unlike many other multiplexers, this FB only updates the output when the value actually changes, making it more efficient for event-driven systems. It also provides a generic implementation, which reduces the need for multiple specific FB types.

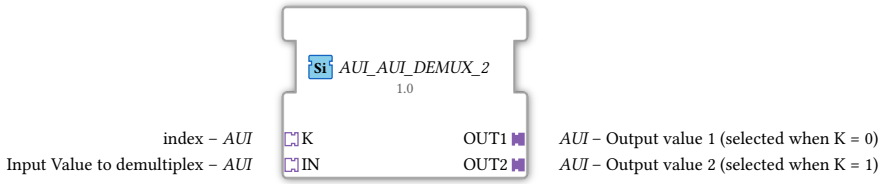
Conclusion

The AUDI_AUI_MUX_8 function block provides a flexible and efficient multiplexing solution for adapter-based data in 4diac IDE applications. Its generic nature, combined with change-detection logic, makes it well-suited for modern, event-driven industrial automation systems where bandwidth and processing efficiency are important. The clear mapping of index to input simplifies configuration and integration.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AUI_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AUI_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `UINT`. Unlike `AUI_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit unsigned integer to forward, routed to exactly one of the outputs (AUI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AUI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_DEMUX`) – shared across all port counts of this block family (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_DEMUX_2` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit unsigned integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

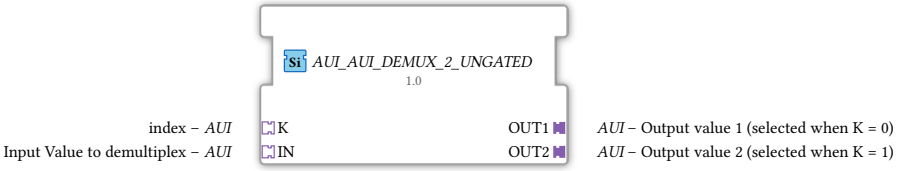
- **`AUI_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUI_AUI_DEMUX_2` carries the demultiplexer logic of the `AUI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AUI_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `UINT`. Unlike `AUI_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit unsigned integer to forward, routed to exactly one of the outputs (AUI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AUI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_DEMUX`) – shared across all port counts of this block family (AUI_AUI_DEMUX_2_UNGATED … AUI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

- Because this block's own payload type is `UINT`, `AUI_AUI_DEMUX_2_UNGATED` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit unsigned integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

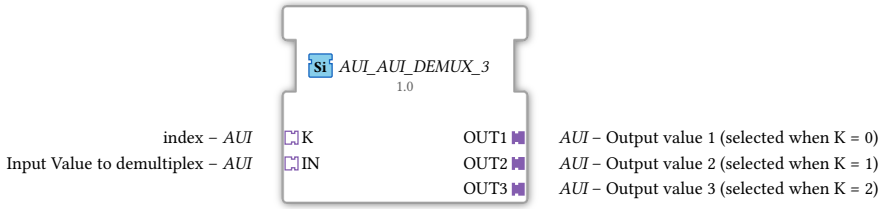
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUI_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUI_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AUI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AUI_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `UINT`. Unlike `AUI_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit unsigned integer to forward, routed to exactly one of the outputs (AUI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AUI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AUI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_DEMUX`) – shared across all port counts of this block family (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_DEMUX_3` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit unsigned integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUI_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

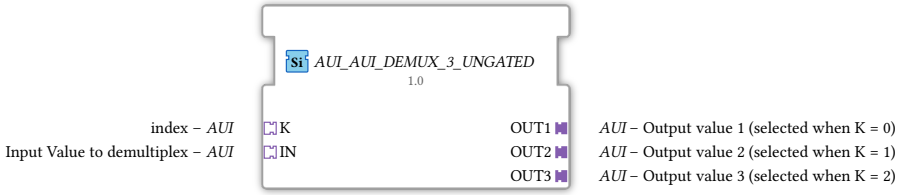
- **`AUI_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUI_AUI_DEMUX_3` carries the demultiplexer logic of the `AUI_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AUI_AUI_DEMUX_3_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `UINT`. Unlike `AUI_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_DEMUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit unsigned integer to forward, routed to exactly one of the outputs (AUI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AUI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AUI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_DEMUX`) – shared across all port counts of this block family (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_DEMUX_3_UNGATED` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit unsigned integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUI_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

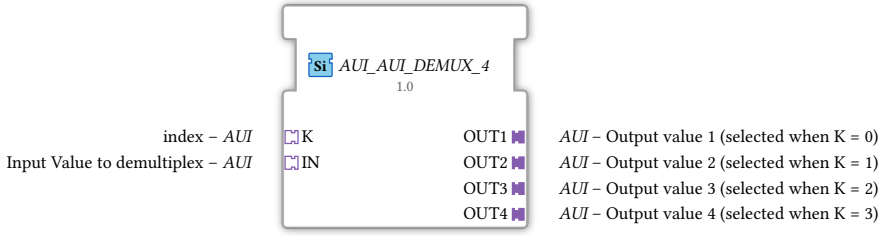
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUI_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUI_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `AUI_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AUI_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `UINT`. Unlike `AUI_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input,

but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUI_AUI_DEMUX_4 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 16-bit unsigned integer to forward, routed to exactly one of the outputs (AUI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AUI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AUI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AUI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 4 output adapters (OUT1 … OUT4) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AUI_AUI_DEMUX) – shared across all port counts of this block family (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is UINT, AUI_AUI_DEMUX_4 uses the same adapter type (AUI) for both the payload (IN/OUT) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit unsigned integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

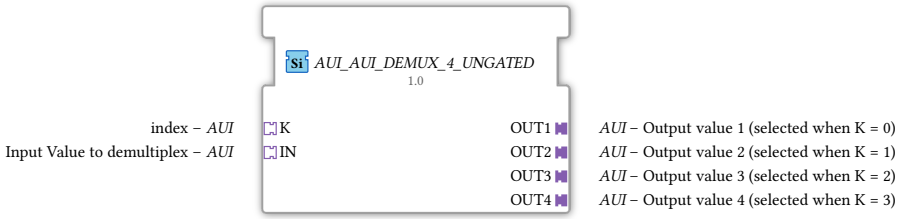
- **`AUI_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUI_AUI_DEMUX_4` carries the demultiplexer logic of the `AUI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AUI_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `UINT`. Unlike `AUI_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN** (Socket): input adapter carrying the 16-bit unsigned integer to forward, routed to exactly one of the outputs (AUI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AUI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AUI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AUI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_DEMUX`) – shared across all port counts of this block family (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_DEMUX_4_UNGATED` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit unsigned integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

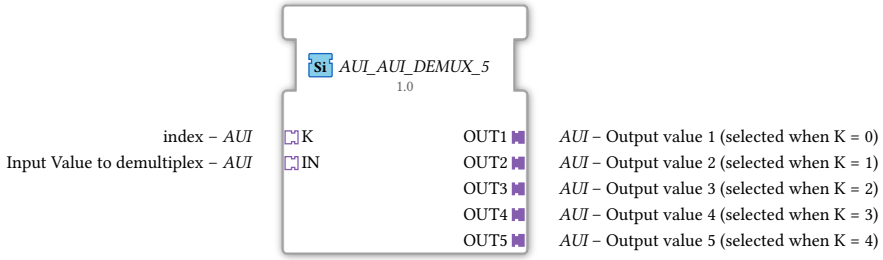
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUI_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

AUI_AUI_DEMUX_4_UNGATED carries the demultiplexer logic of the AUI_DEMUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_DEMUX_5



• – ○ • – ○ • – ○ •

Introduction

AUI_AUI_DEMUX_5 is the adapter-based variant of the generic demultiplexer for data type UINT. Unlike AUI_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUI_AUI_DEMUX_5 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 16-bit unsigned integer to forward, routed to exactly one of the outputs (AUI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AUI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AUI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AUI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AUI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 5 output adapters (OUT1 … OUT5) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_DEMUX`) – shared across all port counts of this block family (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_DEMUX_5` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit unsigned integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

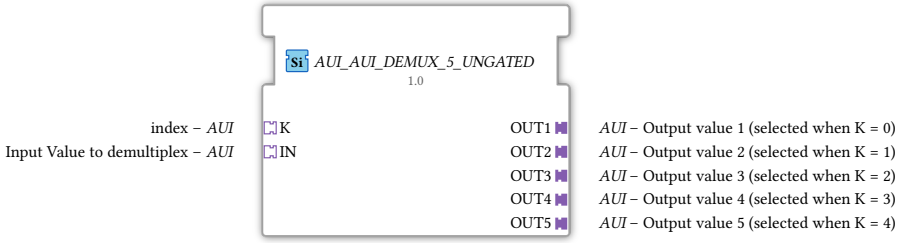
- **`AUI_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUI_AUI_DEMUX_5` carries the demultiplexer logic of the `AUI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_DEMUX_5_UNGATED

i UNGATED variant: This block is the ungated version of AUI_AUI_DEMUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AUI_AUI_DEMUX_5_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `UINT`. Unlike `AUI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_DEMUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit unsigned integer to forward, routed to exactly one of the outputs (AUI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AUI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AUI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AUI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AUI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_DEMUX`) – shared across all port counts of this block family (AUI_AUI_DEMUX_2 … AUI_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_DEMUX_5_UNGATED` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit unsigned integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

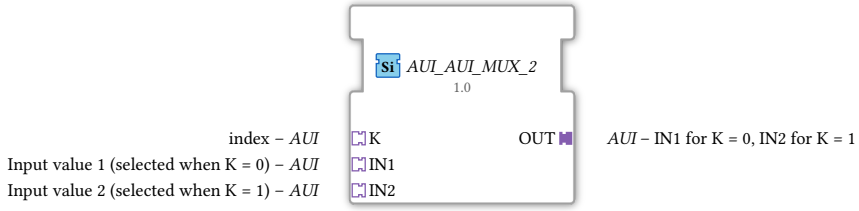
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUI_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUI_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AUI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AUI_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `UINT`. Unlike `AUI_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AUI adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AUI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_MUX`) – shared across all port counts of this block family (AUI_AUI_MUX_2 … AUI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_MUX_2` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 16-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

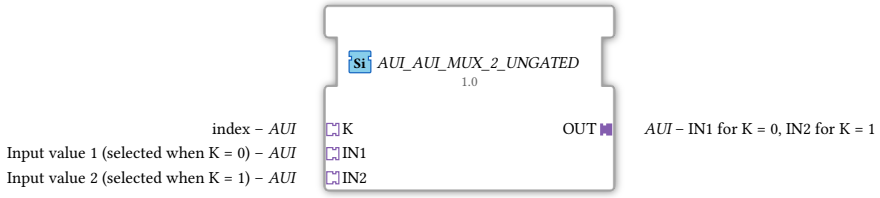
- **`AUI_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUI_AUI_MUX_2` carries the multiplexer logic of the `AUI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUI_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AUI_AUI_MUX_2_UNGATED is the adapter-based variant of the generic multiplexer for data type UINT. Unlike AUI_MUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUI_AUI_MUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 2, passed through to OUT when the index is K = 0 (AUI adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to OUT when the index is K = 1 (AUI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN2:

1. The current value of K.D1 determines which of the 2 input adapters (IN1 … IN2) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_MUX`) – shared across all port counts of this block family (AUI_AUI_MUX_2_UNGATED … AUI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_MUX_2_UNGATED` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 16-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

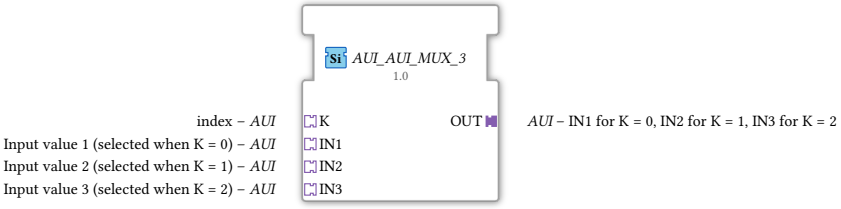
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AUI_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUI_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AUI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_MUX_3



• – ○ • – ○ • – ○ •

Introduction

`AUI_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `UINT`. Unlike `AUI_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AUI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AUI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AUI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_MUX`) – shared across all port counts of this block family (AUI_AUI_MUX_2 … AUI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_MUX_3` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) → read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 16-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AUI_MUX_3 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

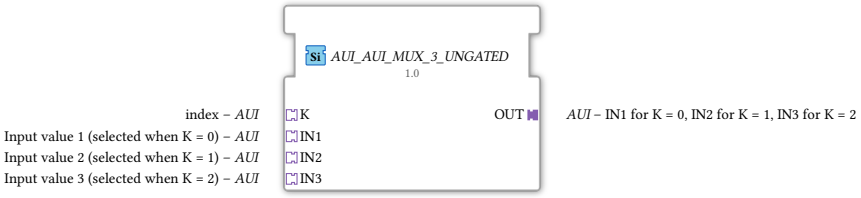
- **AUI_AUI_MUX_3_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AUI_AUI_MUX_3 carries the multiplexer logic of the AUI_MUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUI_AUI_MUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AUI_AUI_MUX_3_UNGATED is the adapter-based variant of the generic multiplexer for data type UINT. Unlike AUI_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUI_AUI_MUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.

- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AUI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AUI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AUI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_MUX`) – shared across all port counts of this block family (AUI_AUI_MUX_2 … AUI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_MUX_3_UNGATED` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 16-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖ Comparison with Similar Blocks

Compare with `AUI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AUI_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUI_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AUI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AUI_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `UINT`. Unlike `AUI_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AUI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AUI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AUI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AUI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_MUX`) – shared across all port counts of this block family (AUI_AUI_MUX_2 … AUI_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_MUX_4` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 16-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

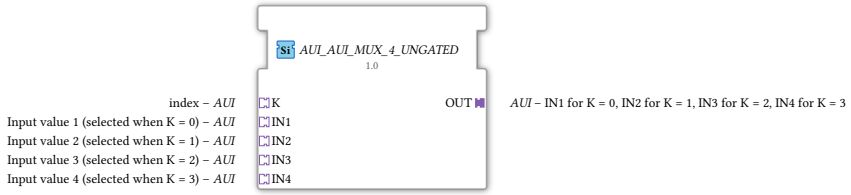
- **`AUI_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUI_AUI_MUX_4` carries the multiplexer logic of the `AUI_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_AUI_MUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AUI_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `UINT`. Unlike `AUI_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AUI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AUI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AUI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AUI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_MUX`) – shared across all port counts of this block family (AUI_AUI_MUX_2 … AUI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_MUX_4_UNGATED` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 16-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

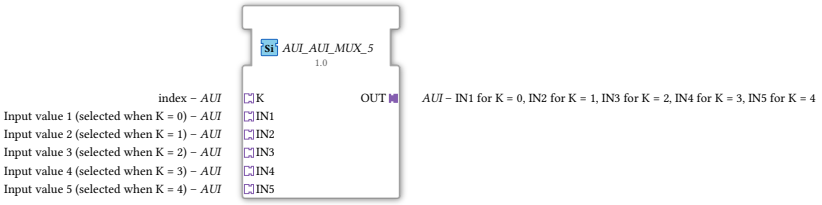
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AUI_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUI_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `AUI_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`AUI_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `UINT`. Unlike `AUI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AUI adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AUI adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AUI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AUI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AUI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` ... `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_MUX`) – shared across all port counts of this block family (AUI_AUI_MUX_2 ... AUI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_MUX_5` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 16-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

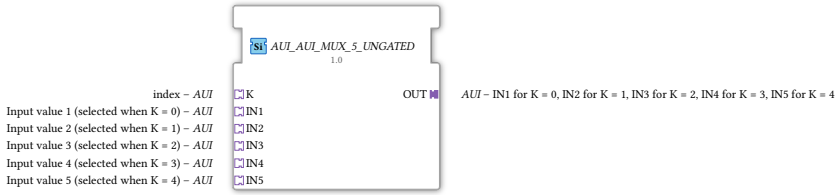
- `AUI_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUI_AUI_MUX_5` carries the multiplexer logic of the `AUI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUI_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• — ○ • — ○ • — ○ •

Introduction

`AUI_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `UINT`. Unlike `AUI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUI_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AUI adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AUI adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AUI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AUI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AUI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUI_AUI_MUX`) – shared across all port counts of this block family (AUI_AUI_MUX_2 … AUI_AUI_MUX_5_UNGATED).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.
- Because this block's own payload type is `UINT`, `AUI_AUI_MUX_5_UNGATED` uses the same adapter type (`AUI`) for both the payload (`IN`/`OUT`) and the index socket **K** – the two roles are wired independently even though they share an adapter type.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 16-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

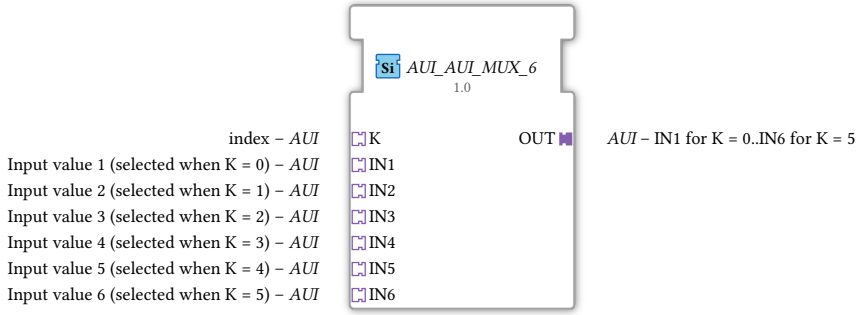
No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- `AUI_AUI_MUX_5`: The gated variant – updates the output only on an actual value change.

Conclusion

`AUI_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AUI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUI_AUI_MUX_6



• – ○ • – ○ • – ○ •

Introduction

AUI_AUI_MUX_6 is an adapter-based multiplexer function block. It selects one of six unidirectional AUI adapter inputs, `IN1` through `IN6`, based on the index value received through the `K` adapter. The selected value is forwarded to the `OUT` adapter plug. The output is updated only when the actual value changes.

Interface Structure

Event Inputs

None. The FB does not expose dedicated event inputs. Its operation is triggered through the connected adapter interfaces, in particular through changes on the `K` index adapter.

Event Outputs

No event outputs.

Data Inputs

None. Data is transferred through the AUI adapter connections instead of dedicated data inputs.

Data Outputs

None. The selected value is made available through the `OUT` adapter plug.

Adapters

Direction	Name	Adapter Type	Description
Socket	`K`	`adapter::types::unidirectional::AUI`	Index adapter. Determines which input is selected.
Socket	`IN1`	`adapter::types::unidirectional::AUI`	Input value 1, selected when `K = 0`.
Socket	`IN2`	`adapter::types::unidirectional::AUI`	Input value 2, selected when `K = 1`.
Socket	`IN3`	`adapter::types::unidirectional::AUI`	Input value 3, selected when `K = 2`.
Socket	`IN4`	`adapter::types::unidirectional::AUI`	Input value 4, selected when `K = 3`.
Socket	`IN5`	`adapter::types::unidirectional::AUI`	Input value 5, selected when `K = 4`.
Socket	`IN6`	`adapter::types::unidirectional::AUI`	Input value 6, selected when `K = 5`.
Plug	`OUT`	`adapter::types::unidirectional::AUI`	Selected AUI output. Represents `IN1` for `K = 0` up to `IN6` for `K = 5`.

Functionality

AUI_AUI_MUX_6 behaves as a multiplexer for AUI adapter connections. It reads the value provided through the `K` socket and interprets it as an integer index between `0` and `5`. Based on this index, one of the six input adapters is routed to the `OUT` adapter.

The FB is designed to avoid unnecessary communication. If a new index value is received but the selected output value has not actually changed, the output is not rewritten and no event is generated. Only when a real value change occurs is the `OUT` adapter updated.

Because `K` is itself an AUI adapter, the index can be supplied by another adapter-compatible block, making the FB suitable for use in adapter-oriented IEC 61499 architectures.

Technical Features

- Six unidirectional AUI input sockets and one unidirectional AUI output plug.
- No direct event inputs or data pins; all communication is performed via adapter interfaces.
- Index range `0` to `5` selects `IN1` to `IN6`.
- Output update is suppressed when the value has not changed.
- The FB is defined as a generic block with the generic class name `GEN_AUI_AUI_MUX`.
- The block is distributed under the Eclipse Public License 2.0.

State Overview

The FB does not contain an explicit ECC state machine in its type definition. Its behavior can therefore be described conceptually:

State	Description
Idle	Waiting for a new index value or value change on the selected input.
Evaluating	The `K` adapter value is read and interpreted as the selection index.
Updating	The selected input value is transferred to the `OUT` adapter.
Returning	The FB then returns to the idle state.

Application Scenarios

- Selecting one of six AUI data sources based on a dynamic index.
- Routing adapter-based signals in modular automation systems.
- Switching between measurement or configuration values without generating redundant events.

- Use in applications where event-on-change behavior is required to reduce communication load.

Comparison with Similar Blocks

Compared to a conventional IEC 61499 MUX block, AUI_AUI_MUX_6 does not use individual data inputs and a `REQ` event input. Instead, it uses unidirectional AUI adapter sockets, which allows a more structured and reusable connection style.

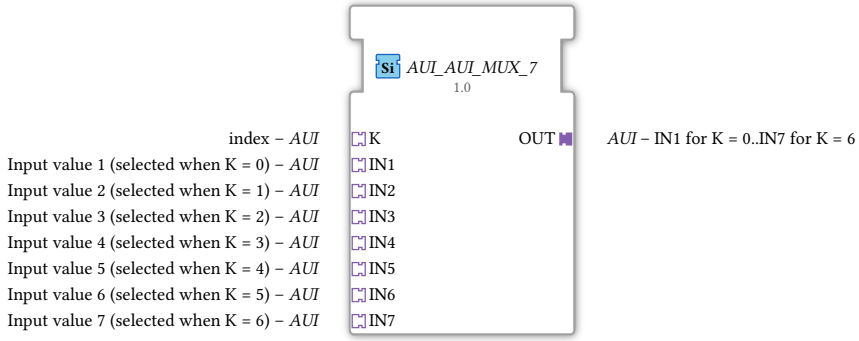
In contrast to a standard multiplexer, this FB updates its output only when the output value actually changes. This makes it particularly useful in event-driven systems where unnecessary events should be avoided.

As a generic FB, it can be instantiated through the generic class name `GEN_AUI_AUI_MUX`, providing flexible tool support while keeping the adapter-based interface consistent.

Conclusion

AUI_AUI_MUX_6 is a compact and useful building block for adapter-based selection and routing tasks. It combines multiplexing functionality with event-on-change behavior, reducing unnecessary output updates. Its six AUI inputs make it well suited for systems that need to choose among multiple unidirectional adapter connections.

AUI_AUI_MUX_7



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_AUI_MUX_7** is a generic, adapter-based multiplexer function block designed to select one of seven input channels and forward its value to a single output. The selection is performed via a dedicated index adapter (`K`). The block is part of the unidirectional adapter type system and is characterized by an event-driven output update mechanism: the output adapter is only refreshed when the selected value actually changes, thereby suppressing redundant events and data transmissions.

Interface Structure

Event Inputs

The function block does not declare any explicit event inputs. Its operation is purely data-driven through the adapter sockets; changes on the selected input adapter trigger the internal processing and subsequent output update.

Event Outputs

No event outputs.

Data Inputs

The block does not expose any direct data inputs. All input data is carried through the adapter sockets listed below.

Data Outputs

The block does not expose any direct data outputs. All output data is carried through the adapter plug `OUT`.

Adapters

Plugs (Outputs):

Name	Type	Comment
OUT	adapter::types::unidirectional::AUI	Selected input value; IN1 for K=0 ... IN7 for K=6

Sockets (Inputs):

Name	Type	Comment
K	adapter::types::unidirectional::AUI	Index; selects one of the 7 input channels (0...6)
IN1	adapter::types::unidirectional::AUI	Input value 1 (selected when K = 0)
IN2	adapter::types::unidirectional::AUI	Input value 2 (selected when K = 1)
IN3	adapter::types::unidirectional::AUI	Input value 3 (selected when K = 2)
IN4	adapter::types::unidirectional::AUI	Input value 4 (selected when K = 3)
IN5	adapter::types::unidirectional::AUI	Input value 5 (selected when K = 4)
IN6	adapter::types::unidirectional::AUI	Input value 6 (selected when K = 5)
IN7	adapter::types::unidirectional::AUI	Input value 7 (selected when K = 6)

Functionality

The block operates as a 7-to-1 multiplexer on the AUI adapter level. The index adapter `K` determines which of the seven input adapters (`IN1` ... `IN7`) is routed to the output adapter `OUT`. The mapping is straightforward:

- `K = 0` ⟶ `IN1`
- `K = 1` ⟶ `IN2`
- `K = 2` ⟶ `IN3`
- `K = 3` ⟶ `IN4`
- `K = 4` ⟶ `IN5`
- `K = 5` ⟶ `IN6`
- `K = 6` ⟶ `IN7`

The output is **not** updated on every input change. Instead, the output adapter is refreshed only when the value of the currently selected input channel deviates from the previously forwarded value. If the selected input value remains unchanged, no output update is performed and no event is generated. This behavior minimizes unnecessary data traffic on the output adapter connection.

Technical Features

- **Generic FB Implementation:** The block is defined as a generic type (`GEN_AUI_AUI_MUX`) and instantiated for the 7-input variant.
- **Unidirectional Adapter Interface:** All adapters (input and output) use the `adapter::types::unidirectional::AUI` type, ensuring a one-way data flow.
- **Event-Driven Output Update:** Output refresh occurs solely on actual value change, reducing communication overhead.
- **No Direct Data I/O:** All data exchange is encapsulated within the adapter interface, promoting modular and reusable design.

State Overview

The block does not exhibit a complex state machine. Its behavior can be described as:

1. **Idle / Monitoring State:** The block monitors the index adapter `K` and the selected input channel.
2. **Change Detection State:** When the index changes or the value of the selected input changes, the block compares the new value with the previously forwarded value.

Application Scenarios

- **Sensor Selection:** Selecting between multiple sensors or measurement sources connected via AUI adapters, forwarding only the active channel's data.
- **Data Source Routing:** Routing data from multiple production lines or machines to a single monitoring or control interface.

- **Mode Switching:** Switching between different operational parameter sets (e.g., configuration profiles) in an IEC 61499-based automation system.
- **Reducing Bus Load:** In distributed systems, using the change-detection mechanism to avoid unnecessary network traffic when values remain stable.

Comparison with Similar Blocks

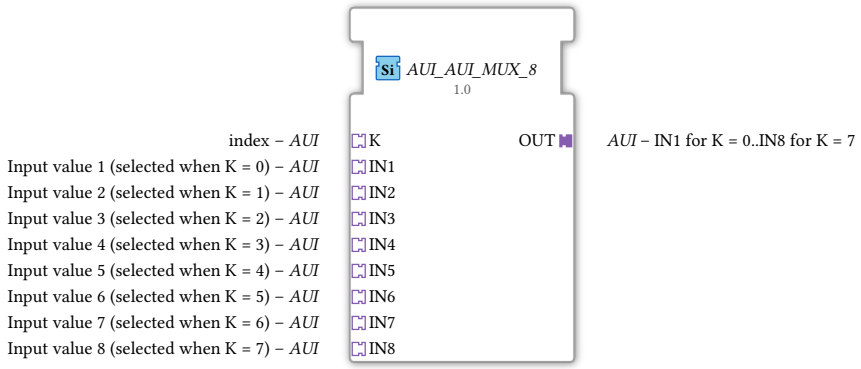
Feature	AUI_AUI_MUX_7	Typical data MUX (ANSI/IEC)
Interface	Adapter-based (AUI)	Direct data pins
Number of inputs	7	Typically 2, 4, 8 (binary select)
Selection	Via adapter `K`	Via binary select lines
Output update	Only on value change	Always on new input/select
Generic instantiation	Yes (`GEN_AUI_AUI_MUX`)	Usually fixed type

Compared to conventional data multiplexers, this block offers a cleaner adapter abstraction, supports a larger number of inputs, and incorporates an event-based change detection that can significantly reduce communication and processing overhead in event-driven 4diac applications.

Conclusion

The **AUI_AUI_MUX_7** provides a robust and efficient solution for adapter-level multiplexing in IEC 61499 environments. Its key strengths lie in the unidirectional adapter interface, the generic instantiation pattern, and the intelligent change-detection mechanism that only updates the output when actual value differences occur. This makes it particularly suitable for distributed automation scenarios where bandwidth and processing efficiency are critical.

AUI_AUI_MUX_8



• – ◦ • – ◦ • – ◦ •

Introduction

The **AUI_AUI_MUX_8** is a generic unidirectional multiplexer function block designed for selecting one of eight input adapters and forwarding its value to a single output adapter. It is part of the AUI (Agrartechnik Universal Interface) adapter family and provides a compact, event-driven solution for dynamic signal routing in automation applications.

This block acts as a selector: based on the index value provided on the **K** adapter, exactly one of the eight input channels (**IN1** to **IN8**) is connected to the output adapter **OUT**. The output is only refreshed when the selected index actually changes, reducing unnecessary network traffic and event load in distributed control systems.

The block is implemented as a generic function block (generic class name `GEN_AUI_AUI_MUX`), allowing reuse across different data types supported by the underlying AUI adapter type.

Interface Structure

Event Inputs

The block has **no event inputs**. All triggering is performed implicitly through value changes on the **K** index adapter.

Event Outputs

No event outputs.

Data Inputs

The block has **no direct data inputs**. All input values are carried via adapter sockets.

Data Outputs

The block has **no direct data outputs**. All output values are carried via the adapter plug.

Adapters

Output Plug (Plugs)

Adapter	Type	Description
OUT	`adapter::types::unidirectional::AUI`	Active output carrying the value of the selected input channel (IN1 for K = 0 … IN8 for K = 7).

Input Sockets (Sockets)

Adapter	Type	Description
K	`adapter::types::unidirectional::AUI`	Index selector. Determines which input channel is forwarded to **OUT** (valid range: 0–7).
IN1	`adapter::types::unidirectional::AUI`	Input value 1, selected when K = 0.
IN2	`adapter::types::unidirectional::AUI`	Input value 2, selected when K = 1.
IN3	`adapter::types::unidirectional::AUI`	Input value 3, selected when K = 2.
IN4	`adapter::types::unidirectional::AUI`	Input value 4, selected when K = 3.
IN5	`adapter::types::unidirectional::AUI`	Input value 5, selected when K = 4.
IN6	`adapter::types::unidirectional::AUI`	Input value 6, selected when K = 5.
IN7	`adapter::types::unidirectional::AUI`	Input value 7, selected when K = 6.
IN8	`adapter::types::unidirectional::AUI`	Input value 8, selected when K = 7.

Functionality

The multiplexer operates on an event-driven basis:

1. The index value **K** is continuously monitored through its adapter socket.
2. Whenever **K** changes to a new value within the valid range (0–7), the block connects the corresponding input adapter (**IN1** through **IN8**) to the output adapter **OUT**.
3. The connected input's current value is copied to **OUT**.

Key design characteristic: **the output is only updated when the index value actually changes**. If **K** is set repeatedly to the same index, no update occurs. This behavior minimizes unnecessary communication on the output adapter and reduces processing overhead in the connected downstream logic.

The block is declared as a generic FB (`eclipse4diac::core::GenericClassName = 'GEN_AUI_AUI_MUX'`), meaning it can be instantiated with type parameters to match the concrete AUI adapter variant required by the application.

Technical Features

- **Eight-to-one multiplexing** with unidirectional AUI adapter type.
- **Change-only update strategy** – output is refreshed exclusively when the selected index changes, avoiding redundant transmissions.
- **Generic implementation** – can be specialized via the generic class name for different adapter data types.
- **Adapter-based interface** – no direct data pins; all data traffic flows through standardised AUI adapters, enabling easy connection to other AUI-compatible blocks.
- **Standard compliance** – conforms to IEC 61499-2 for distributed automation function blocks.
- **License** – distributed under the Eclipse Public License 2.0.

State Overview

The block maintains an internal state representing the currently active index. The following state transitions occur:

Current State	Trigger Condition	Action	New State
Idle	No change on **K**	None (no output update, no event)	Idle
Idle	**K** changes to valid index n	Select **IN_(n+1)**, copy value to **OUT**	Active (index n)
Active	**K** changes to a different valid index m	Select **IN_(m+1)**, copy value to **OUT**	Active (index m)
Active	**K** set to same value again	No action (no update, no event)	Active (unchanged)
Any	**K** out of valid range (0–7)	No valid selection; output remains unchanged	Last valid state

The internal state is only the last applied index; no additional sequencing or timeout logic is required.

Application Scenarios

- **Sensor selection in agricultural machinery**: Switch between multiple sensors (e.g., temperature, pressure, fill level) connected to a single processing unit, using an index signal to select the active measurement source.
- **Mode-dependent data routing**: Redirect different data sources to a common consumer depending on the current operating mode (e.g., automatic, manual, service).
- **Redundant channel handling**: Select between redundant or alternative data paths (e.g., primary vs. backup sensor) based on a health-check index.
- **Parameter multiplexing in distributed control**: Combine multiple configuration values arriving on separate AUI adapters into one output stream for a downstream controller.
- **Test and commissioning**: Dynamically switch between different test signal generators feeding a single measurement or logging unit.

Comparison with Similar Blocks

Block	Type	Number of Inputs	Update Behaviour	Output Event
AUI_AUI_MUX_8	Adapter-based	8	Only on index change	None
Typical MUX (data pin)	Direct data	2–16	On every input change or on selector event	Usually none
AUI_AUI_MUX_2	Adapter-based	2	Only on index change	None
DEMUX variants	Adapter-based	1 input, multiple outputs	Selects one of several outputs	Optional confirmation

Compared to classic data-pin multiplexers, this block offers the advantage of a fully adapter-based interface, making it seamlessly interoperable with other AUI blocks in the 4diac environment. Its change-only update strategy distinguishes it from conventional multiplexers that propagate updates on every input variation, which can be beneficial in bandwidth-constrained or event-intensive systems.

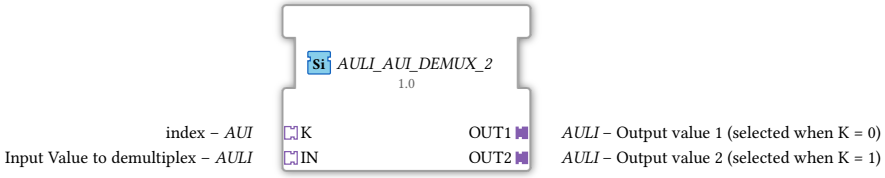
Conclusion

The **AUI_AUI_MUX_8** is a versatile, generic multiplexer block for the AUI adapter family, providing reliable eight-channel selection with a change-detection mechanism. Its adapter-only interface and change-detection logic make it well suited for modern distributed automation applications where communication efficiency and modularity are essential. By integrating seamlessly with the Eclipse 4diac suite and adhering to the IEC 61499 standard, it offers a robust solution for signal routing tasks in agricultural, industrial, and process automation environments.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AULI_AUI_DEMUX_2



• – ○ • – ○ • – ○ •

Introduction

`AULI_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `ULINT`. Unlike `AULI_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit unsigned integer to forward, routed to exactly one of the outputs (AULI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AULI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_DEMUX`) – shared across all port counts of this block family (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit unsigned integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AULI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

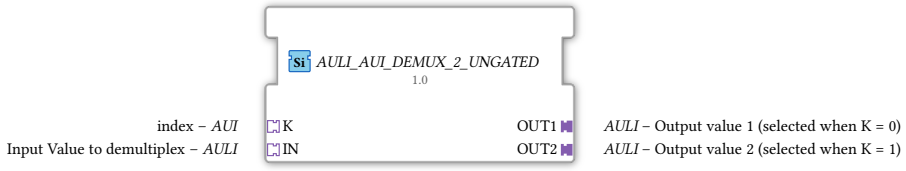
- **`AULI_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AULI_AUI_DEMUX_2` carries the demultiplexer logic of the `AULI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AULI_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `ULINT`. Unlike `AULI_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit unsigned integer to forward, routed to exactly one of the outputs (AULI adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AULI adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_DEMUX`) – shared across all port counts of this block family (AULI_AUI_DEMUX_2_UNGATED … AULI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit unsigned integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AULI_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

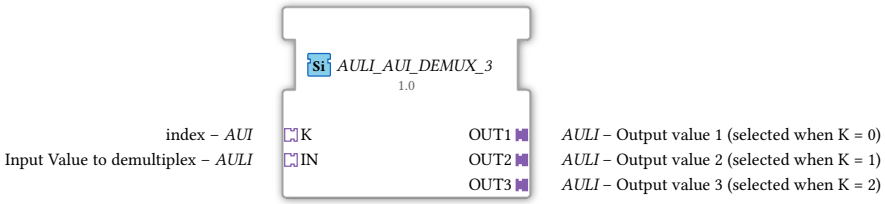
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AULI_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AULI_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AULI_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AULI_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `ULINT`. Unlike `AULI_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN** (Socket): input adapter carrying the 64-bit unsigned integer to forward, routed to exactly one of the outputs (AULI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AULI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AULI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_DEMUX`) – shared across all port counts of this block family (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit unsigned integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AULI_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

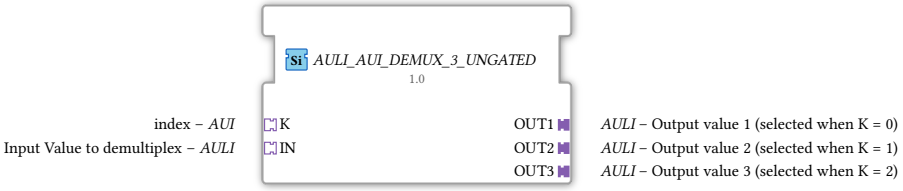
- **`AULI_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AULI_AUI_DEMUX_3` carries the demultiplexer logic of the `AULI_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_DEMUX_3_UNGATED

i UNGATED variant: This block is the ungated version of AULI_AUI_DEMUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AULI_AUI_DEMUX_3_UNGATED is the adapter-based variant of the generic demultiplexer for data type ULINT. Unlike AULI_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AULI_AUI_DEMUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 64-bit unsigned integer to forward, routed to exactly one of the outputs (AULI adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AULI adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AULI adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 3 output adapters (OUT1 … OUT3) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AULI_AUI_DEMUX) – shared across all port counts of this block family (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) ⟶ determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit unsigned integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AULI_DEMUX_3 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

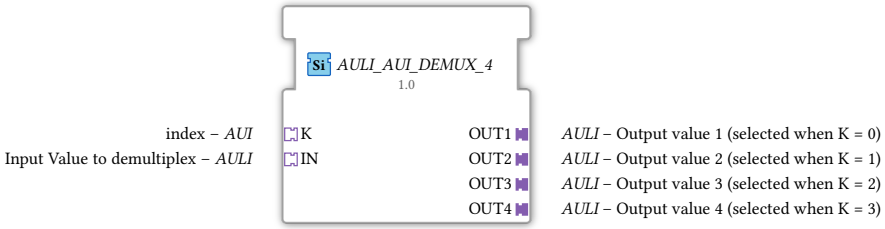
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **AULI_AUI_DEMUX_3**: The gated variant – updates the output only on an actual value change.

Conclusion

AULI_AUI_DEMUX_3_UNGATED carries the demultiplexer logic of the AULI_DEMUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

AULI_AUI_DEMUX_4 is the adapter-based variant of the generic demultiplexer for data type ULINT. Unlike AULI_DEMUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit unsigned integer to forward, routed to exactly one of the outputs (AULI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AULI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AULI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AULI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_DEMUX`) – shared across all port counts of this block family (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit unsigned integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AULI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

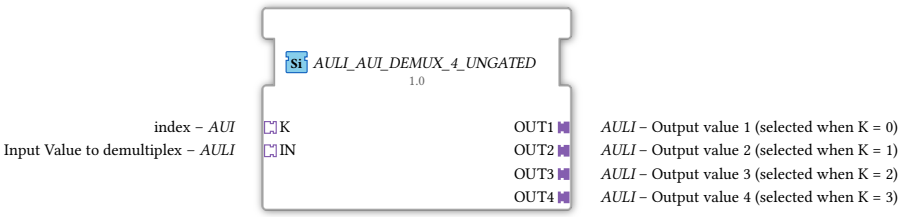
- **`AULI_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AULI_AUI_DEMUX_4` carries the demultiplexer logic of the `AULI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AULI_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `ULINT`. Unlike `AULI_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.

- **IN** (Socket): input adapter carrying the 64-bit unsigned integer to forward, routed to exactly one of the outputs (AULI adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AULI adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AULI adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AULI adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_DEMUX`) – shared across all port counts of this block family (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit unsigned integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AULI_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

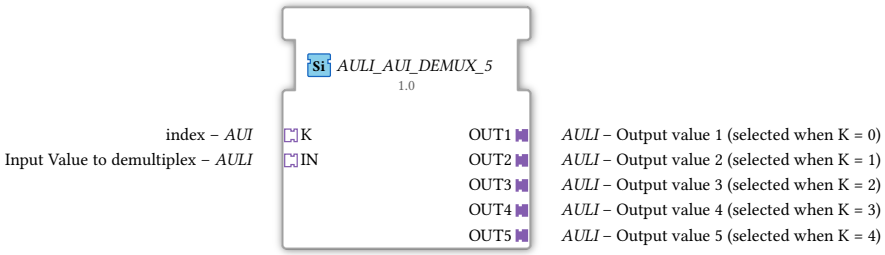
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AULI_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AULI_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AULI_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AULI_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `ULINT`. Unlike `AULI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit unsigned integer to forward, routed to exactly one of the outputs (AULI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AULI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AULI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AULI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AULI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_DEMUX`) – shared across all port counts of this block family (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit unsigned integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AULI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

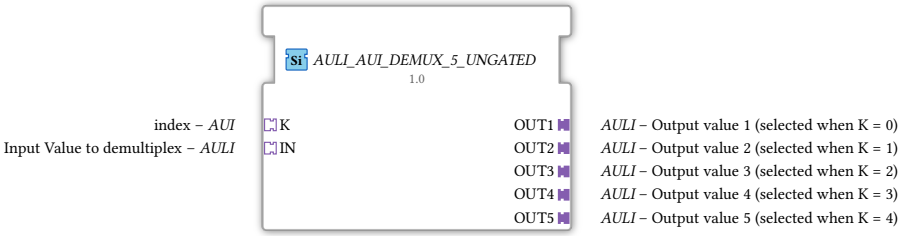
- **`AULI_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AULI_AUI_DEMUX_5` carries the demultiplexer logic of the `AULI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_AUI_DEMUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AULI_AUI_DEMUX_5_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `ULINT`. Unlike `AULI_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K

data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_DEMUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 64-bit unsigned integer to forward, routed to exactly one of the outputs (AULI adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AULI adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AULI adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AULI adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AULI adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_DEMUX`) – shared across all port counts of this block family (AULI_AUI_DEMUX_2 … AULI_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 64-bit unsigned integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AULI_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

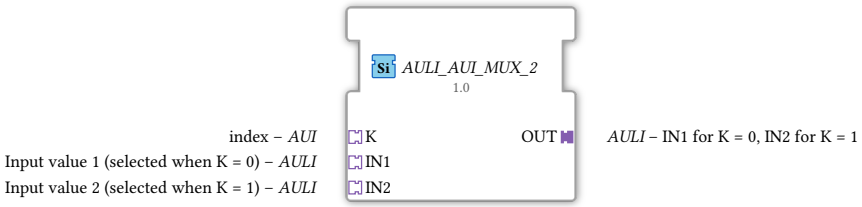
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AULI_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AULI_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AULI_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AULI_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `ULINT`. Unlike `AULI_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AULI adapter type).

- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AULI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_MUX`) – shared across all port counts of this block family (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 64-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AULI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

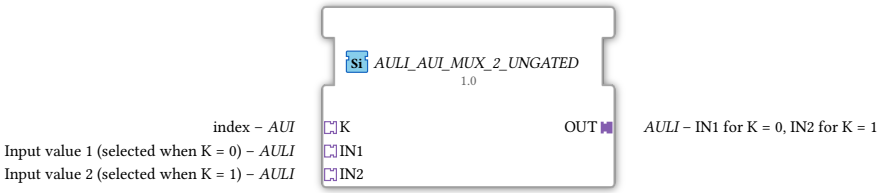
- **`AULI_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AULI_AUI_MUX_2` carries the multiplexer logic of the `AULI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_AUI_MUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AULI_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `ULINT`. Unlike `AULI_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AULI adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AULI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_MUX`) – shared across all port counts of this block family (AULI_AUI_MUX_2_UNGATED … AULI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 64-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AULI_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

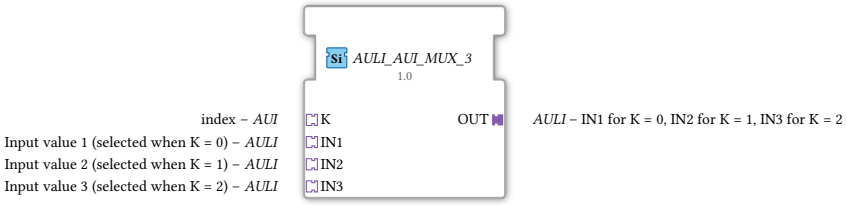
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AULI_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AULI_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AULI_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_MUX_3



• – ∘ • – ∘ • – ∘ •

Introduction

`AULI_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `ULINT`. Unlike `AULI_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AULI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AULI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AULI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_MUX`) – shared across all port counts of this block family (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 64-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AULI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AULI_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

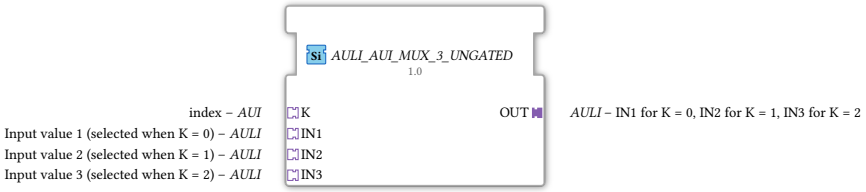
Conclusion

`AULI_AUI_MUX_3` carries the multiplexer logic of the `AULI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_AUI_MUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change

(e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• — ○ • — ○ • — ○ •

Introduction

`AULI_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `ULINT`. Unlike `AULI_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AULI adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AULI adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AULI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_MUX`) – shared across all port counts of this block family (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 64-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AULI_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AULI_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AULI_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AULI_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AULI_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `ULINT`. Unlike `AULI_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AULI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AULI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AULI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AULI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_MUX`) – shared across all port counts of this block family (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 64-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AULI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AULI_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AULI_AUI_MUX_4 carries the multiplexer logic of the AULI_MUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AULI_AUI_MUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AULI_AUI_MUX_4_UNGATED is the adapter-based variant of the generic multiplexer for data type ULINT. Unlike AULI_MUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AULI_AUI_MUX_4_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 4, passed through to OUT when the index is K = 0 (AULI adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to OUT when the index is K = 1 (AULI adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to OUT when the index is K = 2 (AULI adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to OUT when the index is K = 3 (AULI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN4:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_MUX`) – shared across all port counts of this block family (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 64-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AULI_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AULI_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AULI_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `AULI_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`AULI_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `ULINT`. Unlike `AULI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AULI adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AULI adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AULI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AULI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AULI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_MUX`) – shared across all port counts of this block family (AULI_AUI_MUX_2 … AULI_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 64-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AULI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AULI_AUI_MUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AULI_AUI_MUX_5` carries the multiplexer logic of the `AULI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AULI_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AULI_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AULI_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `ULINT`. Unlike `AULI_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AULI_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AULI adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AULI adapter type).

- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AULI adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AULI adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AULI adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AULI adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` ... `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AULI_AUI_MUX`) – shared across all port counts of this block family (AULI_AUI_MUX_2 ... AULI_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 64-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AULI_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).
No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AULI_AUI_MUX_5`**: The gated variant – updates the output only on an actual value change.

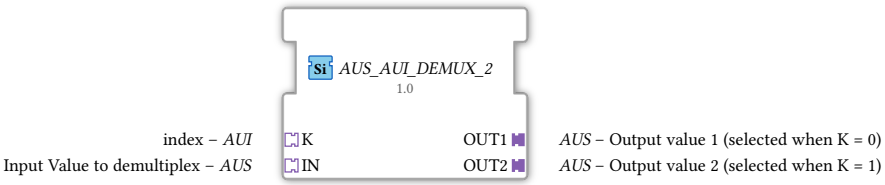
Conclusion

`AULI_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AULI_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AUS_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

AUS_AUI_DEMUX_2 is the adapter-based variant of the generic demultiplexer for data type USINT. Unlike AUS_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUS_AUI_DEMUX_2 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 8-bit unsigned integer to forward, routed to exactly one of the outputs (AUS adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AUS adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 2 output adapters (OUT1 … OUT2) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AUS_AUI_DEMUX) – shared across all port counts of this block family (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit unsigned integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUS_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

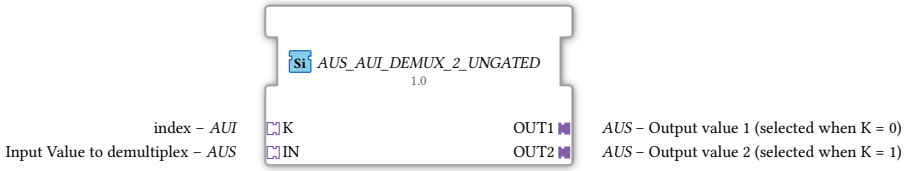
- **`AUS_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUS_AUI_DEMUX_2` carries the demultiplexer logic of the `AUS_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AUS_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `USINT`. Unlike `AUS_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 8-bit unsigned integer to forward, routed to exactly one of the outputs (AUS adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AUS adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_DEMUX`) – shared across all port counts of this block family (AUS_AUI_DEMUX_2_UNGATED … AUS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit unsigned integer to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUS_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

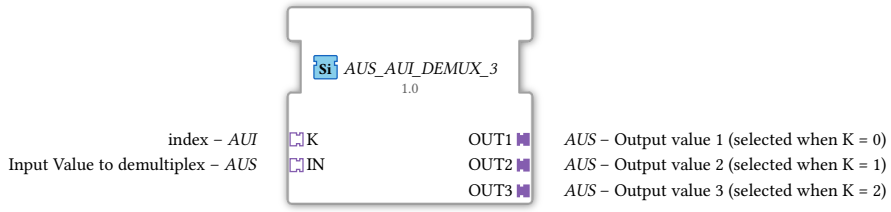
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUS_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUS_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AUS_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AUS_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `USINT`. Unlike `AUS_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 8-bit unsigned integer to forward, routed to exactly one of the outputs (AUS adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AUS adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AUS adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_DEMUX`) – shared across all port counts of this block family (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit unsigned integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUS_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

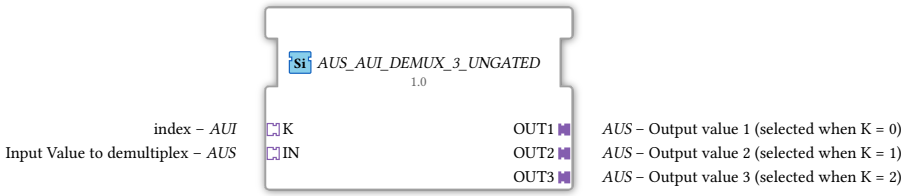
- **`AUS_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUS_AUI_DEMUX_3` carries the demultiplexer logic of the `AUS_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AUS_AUI_DEMUX_3_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `USINT`. Unlike `AUS_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_DEMUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 8-bit unsigned integer to forward, routed to exactly one of the outputs (AUS adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AUS adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AUS adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_DEMUX`) – shared across all port counts of this block family (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit unsigned integer to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUS_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

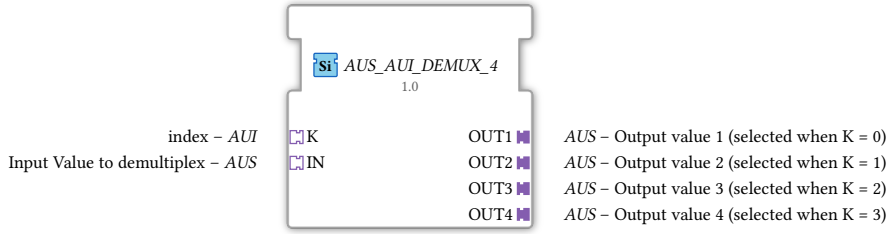
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUS_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUS_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `AUS_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AUS_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `USINT`. Unlike `AUS_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 8-bit unsigned integer to forward, routed to exactly one of the outputs (AUS adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AUS adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AUS adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AUS adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_DEMUX`) – shared across all port counts of this block family (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit unsigned integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUS_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

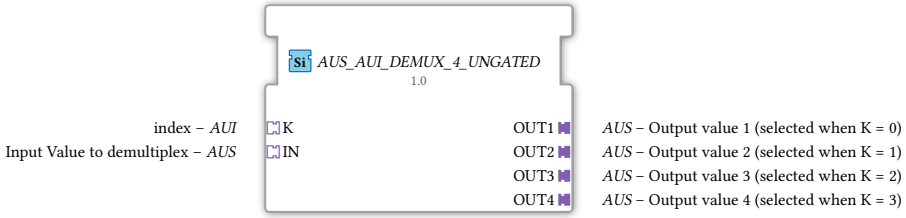
- **`AUS_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUS_AUI_DEMUX_4` carries the demultiplexer logic of the `AUS_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AUS_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `USINT`. Unlike `AUS_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 8-bit unsigned integer to forward, routed to exactly one of the outputs (AUS adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AUS adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AUS adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AUS adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_DEMUX`) – shared across all port counts of this block family (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit unsigned integer to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUS_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

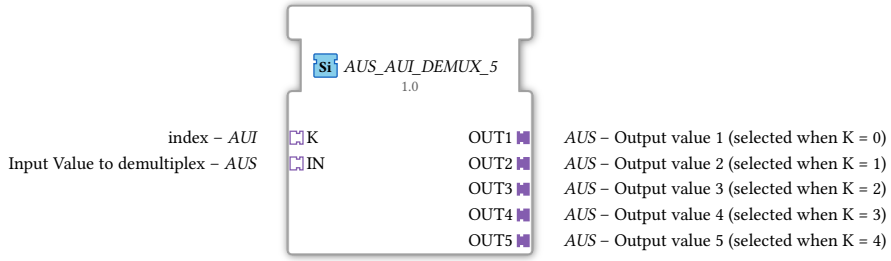
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUS_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUS_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AUS_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AUS_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `USINT`. Unlike `AUS_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 8-bit unsigned integer to forward, routed to exactly one of the outputs (AUS adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AUS adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AUS adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AUS adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AUS adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_DEMUX`) – shared across all port counts of this block family (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit unsigned integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUS_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

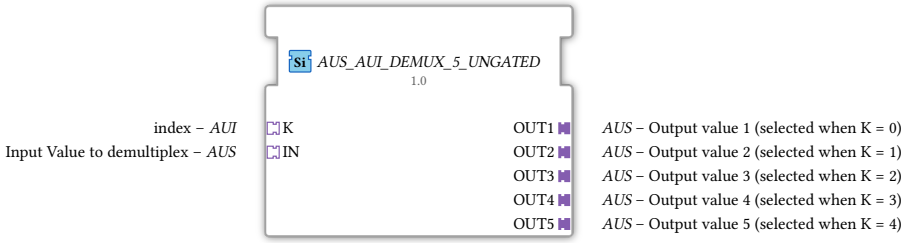
- **`AUS_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AUS_AUI_DEMUX_5 carries the demultiplexer logic of the AUS_DEMUX_5 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUS_AUI_DEMUX_5. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AUS_AUI_DEMUX_5_UNGATED is the adapter-based variant of the generic demultiplexer for data type USINT. Unlike AUS_DEMUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUS_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 8-bit unsigned integer to forward, routed to exactly one of the outputs (AUS adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AUS adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AUS adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AUS adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AUS adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_DEMUX`) – shared across all port counts of this block family (AUS_AUI_DEMUX_2 … AUS_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 8-bit unsigned integer to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AUS_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

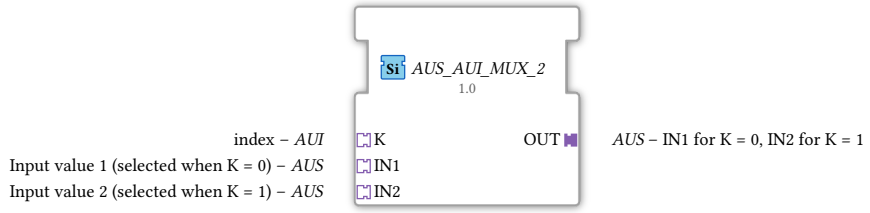
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AUS_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUS_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AUS_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_MUX_2



• – ○ • – ○ • – ○ •

Introduction

`AUS_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `USINT`. Unlike `AUS_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input,

but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AUS adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AUS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_MUX`) – shared across all port counts of this block family (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 8-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AUS_MUX_2 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

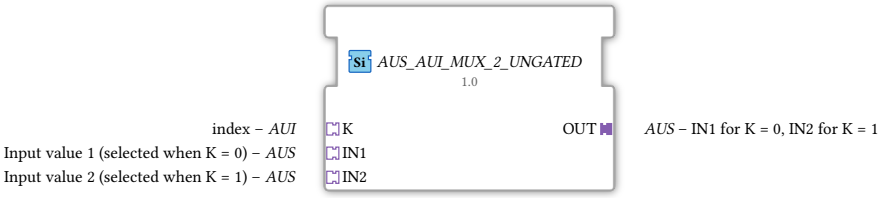
- **AUS_AUI_MUX_2_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AUS_AUI_MUX_2 carries the multiplexer logic of the AUS_MUX_2 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUS_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AUS_AUI_MUX_2_UNGATED is the adapter-based variant of the generic multiplexer for data type USINT. Unlike AUS_MUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUS_AUI_MUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.

- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AUS adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AUS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` ... `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_MUX`) – shared across all port counts of this block family (AUS_AUI_MUX_2_UNGATED ... AUS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 8-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUS_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

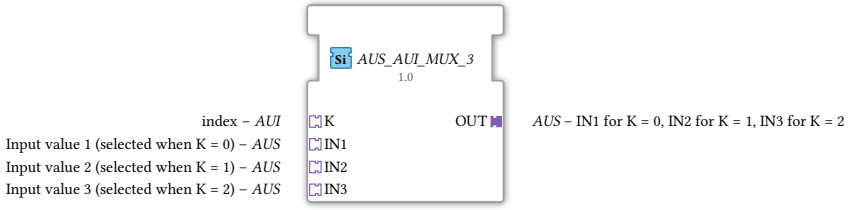
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AUS_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUS_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AUS_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

AUS_AUI_MUX_3 is the adapter-based variant of the generic multiplexer for data type USINT. Unlike AUS_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUS_AUI_MUX_3 evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (AUS adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (AUS adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (AUS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AUS_AUI_MUX) – shared across all port counts of this block family (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) → read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 8-bit unsigned integer sources through a centrally managed adapter index

- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AUS_MUX_3 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

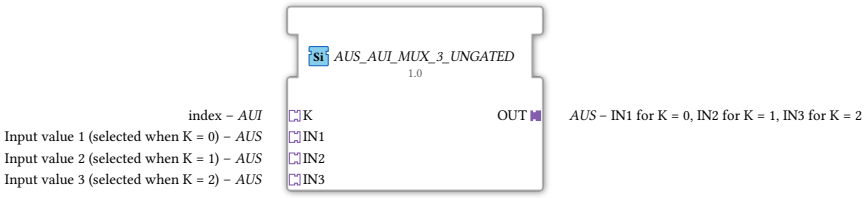
- **AUS_AUI_MUX_3_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AUS_AUI_MUX_3 carries the multiplexer logic of the AUS_MUX_3 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AUS_AUI_MUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AUS_AUI_MUX_3_UNGATED is the adapter-based variant of the generic multiplexer for data type USINT. Unlike AUS_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AUS_AUI_MUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AUS adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AUS adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AUS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_MUX`) – shared across all port counts of this block family (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 8-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUS_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AUS_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUS_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AUS_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AUS_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `USINT`. Unlike `AUS_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AUS adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AUS adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AUS adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AUS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_MUX`) – shared across all port counts of this block family (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 8-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUS_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AUS_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUS_AUI_MUX_4` carries the multiplexer logic of the `AUS_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_AUI_MUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AUS_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `USINT`. Unlike `AUS_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AUS adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AUS adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AUS adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AUS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_MUX`) – shared across all port counts of this block family (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 8-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUS_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

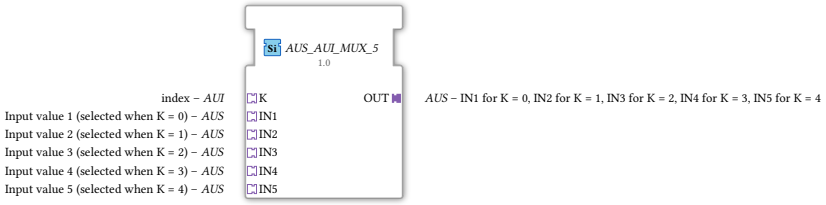
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AUS_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AUS_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `AUS_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`AUS_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `USINT`. Unlike `AUS_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AUS adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AUS adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AUS adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AUS adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AUS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_MUX`) – shared across all port counts of this block family (AUS_AUI_MUX_2 … AUS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 8-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUS_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

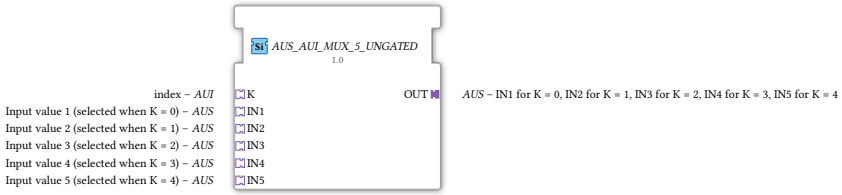
- `AUS_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AUS_AUI_MUX_5` carries the multiplexer logic of the `AUS_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AUS_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AUS_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• — ○ • — ○ • — ○ •

Introduction

`AUS_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `USINT`. Unlike `AUS_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AUS_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AUS adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AUS adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AUS adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AUS adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AUS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AUS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AUS_AUI_MUX`) – shared across all port counts of this block family (AUS_AUI_MUX_2 … AUS_AUI_MUX_5_UNGATED).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 8-bit unsigned integer sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AUS_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- `AUS_AUI_MUX_5`: The gated variant – updates the output only on an actual value change.

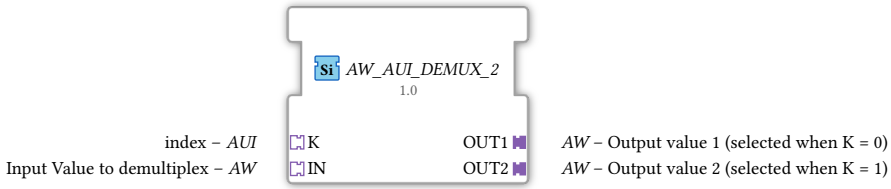
Conclusion

`AUS_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AUS_MUX_5` family over to a purely adapter-based index supply. This makes the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AW_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AW_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `WORD`. Unlike `AW_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit bit pattern to forward, routed to exactly one of the outputs (AW adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AW adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_DEMUX`) – shared across all port counts of this block family (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit bit pattern to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AW_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

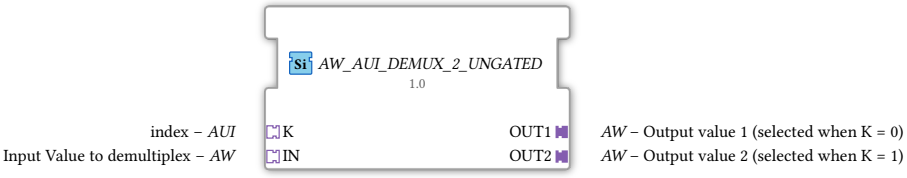
- **`AW_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AW_AUI_DEMUX_2` carries the demultiplexer logic of the `AW_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AW_AUI_DEMUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AW_AUI_DEMUX_2_UNGATED is the adapter-based variant of the generic demultiplexer for data type WORD. Unlike AW_DEMUX_2, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AW_AUI_DEMUX_2_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 16-bit bit pattern to forward, routed to exactly one of the outputs (AW adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AW adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 2 output adapters (OUT1 … OUT2) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_DEMUX`) – shared across all port counts of this block family (AW_AUI_DEMUX_2_UNGATED … AW_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit bit pattern to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AW_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

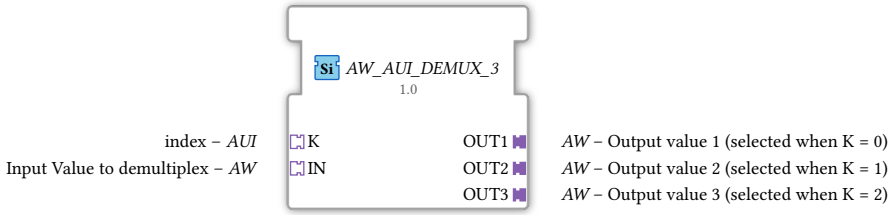
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AW_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AW_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AW_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_DEMUX_3



• – ○ • – ○ • – ○ •

Introduction

`AW_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `WORD`. Unlike `AW_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit bit pattern to forward, routed to exactly one of the outputs (AW adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AW adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AW adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_DEMUX`) – shared across all port counts of this block family (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit bit pattern to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AW_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

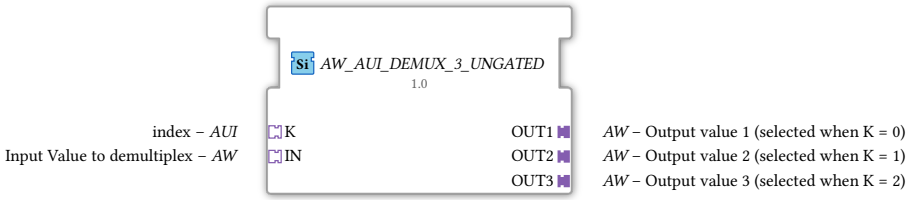
- **`AW_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AW_AUI_DEMUX_3` carries the demultiplexer logic of the `AW_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AW_AUI_DEMUX_3_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `WORD`. Unlike `AW_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_DEMUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit bit pattern to forward, routed to exactly one of the outputs (AW adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AW adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AW adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see “Change Detection” below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn’t changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_DEMUX`) – shared across all port counts of this block family (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit bit pattern to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AW_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

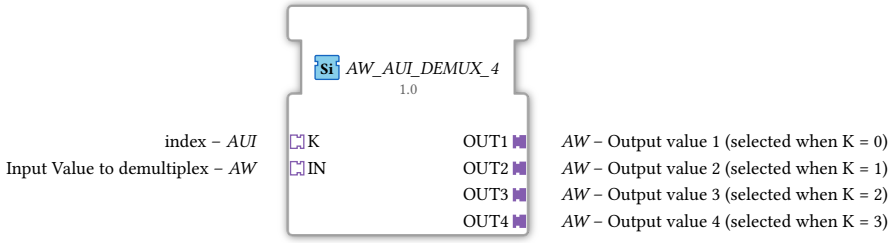
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AW_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AW_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `AW_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AW_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `WORD`. Unlike `AW_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit bit pattern to forward, routed to exactly one of the outputs (AW adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AW adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AW adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AW adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_DEMUX`) – shared across all port counts of this block family (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit bit pattern to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with AW_DEMUX_4 (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

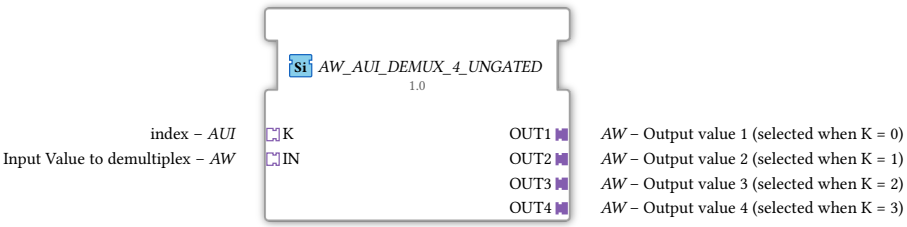
- **AW_AUI_DEMUX_4_UNGATED**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

AW_AUI_DEMUX_4 carries the demultiplexer logic of the AW_DEMUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AW_AUI_DEMUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AW_AUI_DEMUX_4_UNGATED is the adapter-based variant of the generic demultiplexer for data type WORD. Unlike AW_DEMUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AW_AUI_DEMUX_4_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.

- **IN** (Socket): input adapter carrying the 16-bit bit pattern to forward, routed to exactly one of the outputs (AW adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AW adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AW adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AW adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_DEMUX`) – shared across all port counts of this block family (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit bit pattern to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AW_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

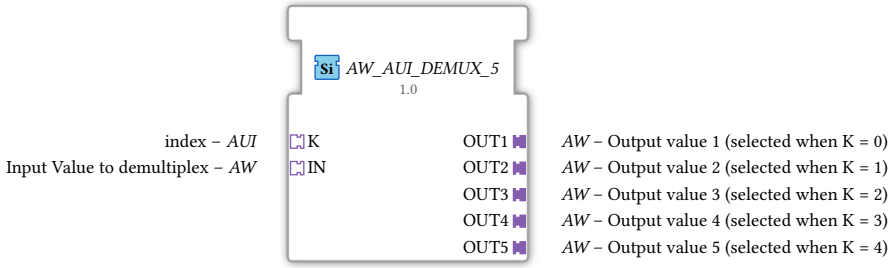
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AW_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AW_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AW_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_DEMUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

`AW_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `WORD`. Unlike `AW_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the 16-bit bit pattern to forward, routed to exactly one of the outputs (AW adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AW adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AW adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AW adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AW adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see “Change Detection” below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn’t changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_DEMUX`) – shared across all port counts of this block family (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit bit pattern to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AW_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

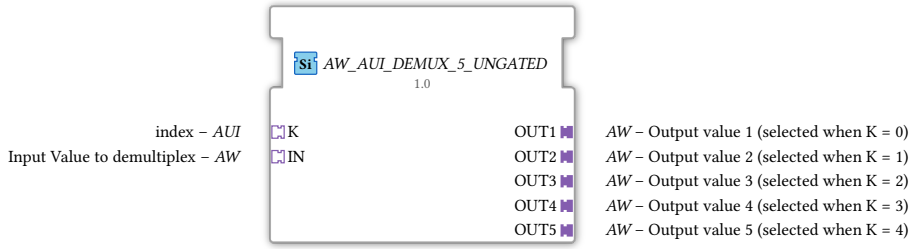
- **`AW_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AW_AUI_DEMUX_5` carries the demultiplexer logic of the `AW_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_AUI_DEMUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AW_AUI_DEMUX_5_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `WORD`. Unlike `AW_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the

index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AW_AUI_DEMUX_5_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the 16-bit bit pattern to forward, routed to exactly one of the outputs (AW adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AW adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AW adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AW adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AW adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 5 output adapters (OUT1 … OUT5) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AW_AUI_DEMUX) – shared across all port counts of this block family (AW_AUI_DEMUX_2 … AW_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a 16-bit bit pattern to up to 5 downstream consumers, selected via an adapter-carried index

- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AW_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

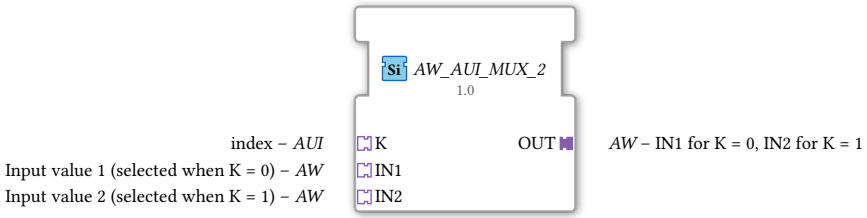
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AW_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AW_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AW_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AW_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `WORD`. Unlike `AW_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AW adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AW adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_MUX`) – shared across all port counts of this block family (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 16-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AW_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

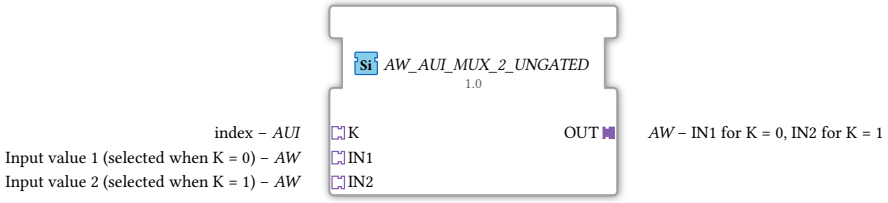
- **`AW_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AW_AUI_MUX_2` carries the multiplexer logic of the `AW_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of AW_AUI_MUX_2. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AW_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `WORD`. Unlike `AW_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AW adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AW adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_MUX`) – shared across all port counts of this block family (AW_AUI_MUX_2_UNGATED … AW_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 16-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AW_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

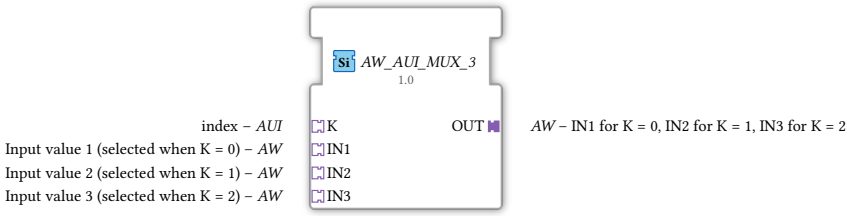
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AW_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AW_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AW_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_MUX_3



• – ○ • – ○ • – ○ •

Introduction

`AW_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `WORD`. Unlike `AW_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AW adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AW adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AW adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_MUX`) – shared across all port counts of this block family (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 16-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AW_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AW_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

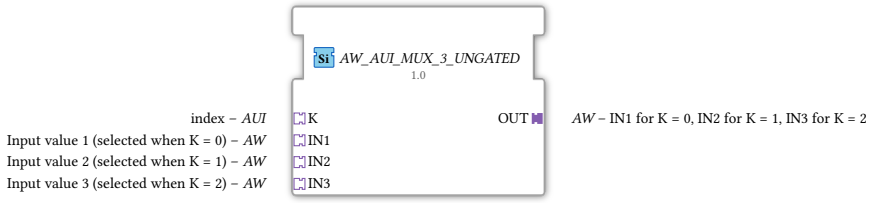
Conclusion

`AW_AUI_MUX_3` carries the multiplexer logic of the `AW_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of AW_AUI_MUX_3. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change

(e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AW_AUI_MUX_3_UNGATED` is the adapter-based variant of the generic multiplexer for data type `WORD`. Unlike `AW_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_MUX_3_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AW adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AW adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AW adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_MUX`) – shared across all port counts of this block family (AW_AUI_MUX_2 … AW_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 16-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AW_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AW_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AW_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AW_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AW_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `WORD`. Unlike `AW_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AW adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AW adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AW adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AW adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_MUX`) – shared across all port counts of this block family (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 16-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AW_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

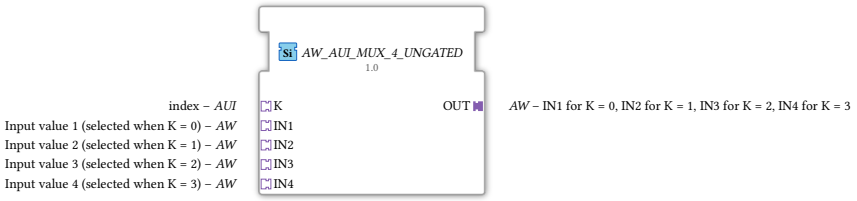
- **`AW_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AW_AUI_MUX_4` carries the multiplexer logic of the `AW_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of AW_AUI_MUX_4. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AW_AUI_MUX_4_UNGATED is the adapter-based variant of the generic multiplexer for data type WORD. Unlike AW_MUX_4, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AW_AUI_MUX_4_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 4, passed through to OUT when the index is K = 0 (AW adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to OUT when the index is K = 1 (AW adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to OUT when the index is K = 2 (AW adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to OUT when the index is K = 3 (AW adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN4:

1. The current value of K.D1 determines which of the 4 input adapters (IN1 … IN4) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AW_AUI_MUX) – shared across all port counts of this block family (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN4) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 16-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with AW_MUX_4 (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **AW_AUI_MUX_4**: The gated variant – updates the output only on an actual value change.

Conclusion

AW_AUI_MUX_4_UNGATED carries the multiplexer logic of the AW_MUX_4 family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an AUI adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

AW_AUI_MUX_5 is the adapter-based variant of the generic multiplexer for data type WORD. Unlike AW_MUX_5, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AW adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AW adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AW adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AW adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AW adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_MUX`) – shared across all port counts of this block family (AW_AUI_MUX_2 … AW_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 16-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AW_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

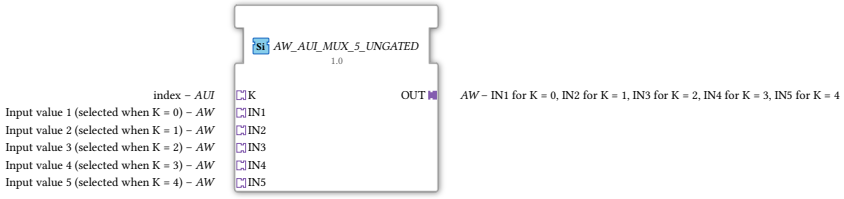
- `AW_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AW_AUI_MUX_5` carries the multiplexer logic of the `AW_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AW_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AW_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AW_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `WORD`. Unlike `AW_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AW_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AW adapter type).

- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AW adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AW adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AW adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AW adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AW adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AW_AUI_MUX`) – shared across all port counts of this block family (AW_AUI_MUX_2 … AW_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 16-bit bit pattern sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AW_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AW_AUI_MUX_5`**: The gated variant – updates the output only on an actual value change.

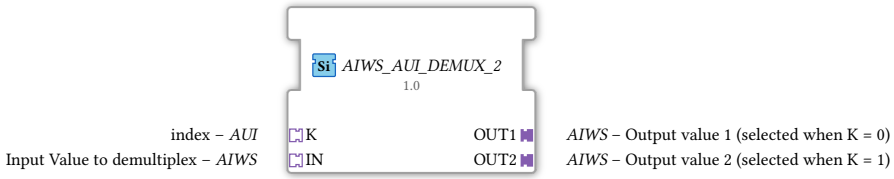
Conclusion

`AW_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AW_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

Numeric Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AIWS_AUI_DEMUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

`AIWS_AUI_DEMUX_2` is the adapter-based variant of the generic demultiplexer for data type `WSTRING`. Unlike `AIWS_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_DEMUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (16-bit wide characters) to forward, routed to exactly one of the outputs (AIWS adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AIWS adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_DEMUX`) – shared across all port counts of this block family (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (16-bit wide characters) to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIWS_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

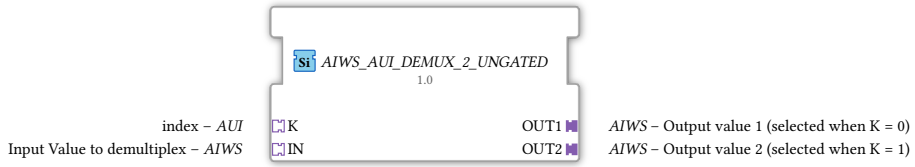
- **`AIWS_AUI_DEMUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIWS_AUI_DEMUX_2` carries the demultiplexer logic of the `AIWS_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_DEMUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_AUI_DEMUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AIWS_AUI_DEMUX_2_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `WSTRING`. Unlike `AIWS_DEMUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_DEMUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (16-bit wide characters) to forward, routed to exactly one of the outputs (AIWS adapter type).
- **OUT1** (Plug): output adapter 1 of 2 (AIWS adapter type).
- **OUT2** (Plug): output adapter 2 of 2 (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 2 output adapters (`OUT1` … `OUT2`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_DEMUX`) – shared across all port counts of this block family (AIWS_AUI_DEMUX_2_UNGATED … AIWS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (16-bit wide characters) to up to 2 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIWS_DEMUX_2` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

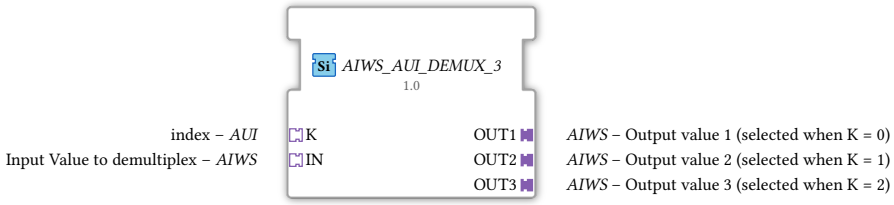
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AIWS_AUI_DEMUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIWS_AUI_DEMUX_2_UNGATED` carries the demultiplexer logic of the `AIWS_DEMUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_DEMUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AIWS_AUI_DEMUX_3` is the adapter-based variant of the generic demultiplexer for data type `WSTRING`. Unlike `AIWS_DEMUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_DEMUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (16-bit wide characters) to forward, routed to exactly one of the outputs (AIWS adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AIWS adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AIWS adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 3 output adapters (`OUT1` … `OUT3`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_DEMUX`) – shared across all port counts of this block family (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (16-bit wide characters) to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIWS_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

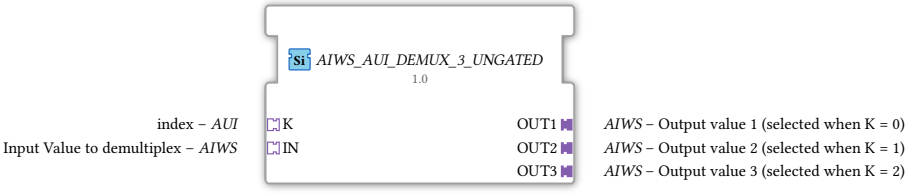
- **`AIWS_AUI_DEMUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIWS_AUI_DEMUX_3` carries the demultiplexer logic of the `AIWS_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_DEMUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_AUI_DEMUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

AIWS_AUI_DEMUX_3_UNGATED is the adapter-based variant of the generic demultiplexer for data type WSTRING. Unlike AIWS_DEMUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AIWS_AUI_DEMUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN** (Socket): input adapter carrying the character string (16-bit wide characters) to forward, routed to exactly one of the outputs (AIWS adapter type).
- **OUT1** (Plug): output adapter 1 of 3 (AIWS adapter type).
- **OUT2** (Plug): output adapter 2 of 3 (AIWS adapter type).
- **OUT3** (Plug): output adapter 3 of 3 (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from the input adapter IN:

1. The current value of K.D1 determines which of the 3 output adapters (OUT1 … OUT3) is currently selected.
2. The data value of IN is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates the current value of IN to the newly selected output immediately, even if IN hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AIWS_AUI_DEMUX) – shared across all port counts of this block family (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN) → determine the currently selected output, update it with the current IN value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (16-bit wide characters) to up to 3 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIWS_DEMUX_3` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

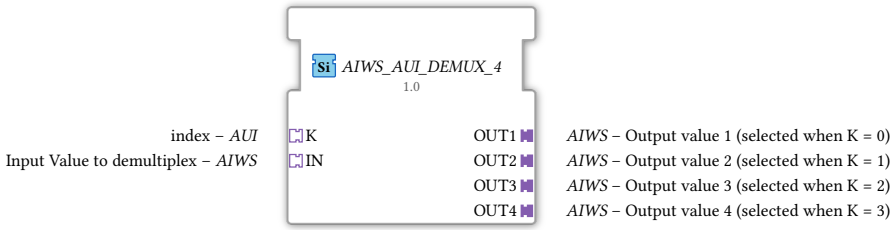
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AIWS_AUI_DEMUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIWS_AUI_DEMUX_3_UNGATED` carries the demultiplexer logic of the `AIWS_DEMUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_DEMUX_4



• – ◦ • – ◦ • – ◦ •

Introduction

`AIWS_AUI_DEMUX_4` is the adapter-based variant of the generic demultiplexer for data type `WSTRING`. Unlike `AIWS_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_DEMUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter’s own `E1`/`D1` event.

- **IN** (Socket): input adapter carrying the character string (16-bit wide characters) to forward, routed to exactly one of the outputs (AIWS adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AIWS adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AIWS adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AIWS adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_DEMUX`) – shared across all port counts of this block family (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (16-bit wide characters) to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIWS_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AIWS_AUI_DEMUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

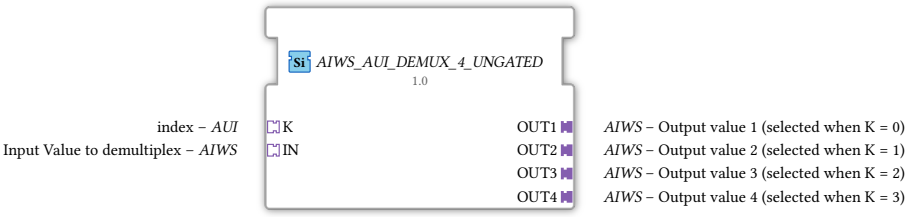
Conclusion

`AIWS_AUI_DEMUX_4` carries the demultiplexer logic of the `AIWS_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_DEMUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_AUI_DEMUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change

(e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ◦ • – ◦ • – ◦ •

Introduction

`AIWS_AUI_DEMUX_4_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `WSTRING`. Unlike `AIWS_DEMUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_DEMUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (16-bit wide characters) to forward, routed to exactly one of the outputs (AIWS adapter type).
- **OUT1** (Plug): output adapter 1 of 4 (AIWS adapter type).
- **OUT2** (Plug): output adapter 2 of 4 (AIWS adapter type).
- **OUT3** (Plug): output adapter 3 of 4 (AIWS adapter type).
- **OUT4** (Plug): output adapter 4 of 4 (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 4 output adapters (`OUT1` … `OUT4`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_DEMUX`) – shared across all port counts of this block family (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (16-bit wide characters) to up to 4 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIWS_DEMUX_4` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

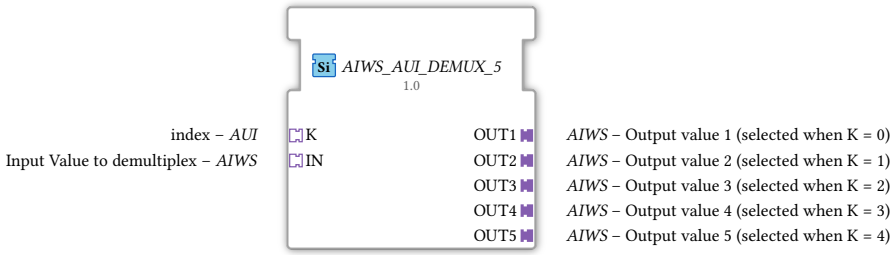
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AIWS_AUI_DEMUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIWS_AUI_DEMUX_4_UNGATED` carries the demultiplexer logic of the `AIWS_DEMUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_DEMUX_5



• – ○ • – ○ • – ○ •

Introduction

`AIWS_AUI_DEMUX_5` is the adapter-based variant of the generic demultiplexer for data type `WSTRING`. Unlike `AIWS_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_DEMUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN** (Socket): input adapter carrying the character string (16-bit wide characters) to forward, routed to exactly one of the outputs (AIWS adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AIWS adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AIWS adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AIWS adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AIWS adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_DEMUX`) – shared across all port counts of this block family (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) → determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (16-bit wide characters) to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)

- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIWS_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

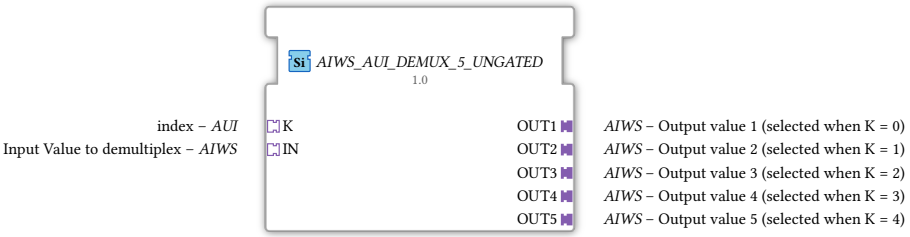
- **`AIWS_AUI_DEMUX_5_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIWS_AUI_DEMUX_5` carries the demultiplexer logic of the `AIWS_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_DEMUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_AUI_DEMUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AIWS_AUI_DEMUX_5_UNGATED` is the adapter-based variant of the generic demultiplexer for data type `WSTRING`. Unlike `AIWS_DEMUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_DEMUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1/D1` event.
- **IN** (Socket): input adapter carrying the character string (16-bit wide characters) to forward, routed to exactly one of the outputs (AIWS adapter type).
- **OUT1** (Plug): output adapter 1 of 5 (AIWS adapter type).
- **OUT2** (Plug): output adapter 2 of 5 (AIWS adapter type).
- **OUT3** (Plug): output adapter 3 of 5 (AIWS adapter type).
- **OUT4** (Plug): output adapter 4 of 5 (AIWS adapter type).
- **OUT5** (Plug): output adapter 5 of 5 (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from the input adapter `IN`:

1. The current value of `K.D1` determines which of the 5 output adapters (`OUT1` … `OUT5`) is currently selected.
2. The data value of `IN` is compared against the value currently held on the selected output. Only on an actual change is that output rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates the current value of `IN` to the newly selected output immediately, even if `IN` hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_DEMUX`) – shared across all port counts of this block family (AIWS_AUI_DEMUX_2 … AIWS_AUI_DEMUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN`) ⟶ determine the currently selected output, update it with the current `IN` value on a change and send its adapter event.

Application Scenarios

- Distributing a character string (16-bit wide characters) to up to 5 downstream consumers, selected via an adapter-carried index
- Routing in modules that already receive the selection index as an AUI adapter link from an upstream block (no separate REQ/K pair needed)
- Building cascadable distributor networks where the index itself originates from another AUI-based block

⚖️ Comparison with Similar Blocks

Compare with `AIWS_DEMUX_5` (same distribution logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

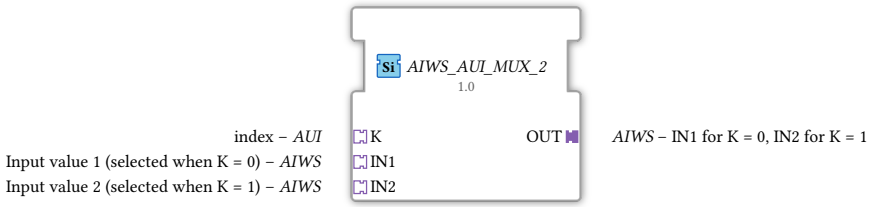
Compare with E_DEMUX, which demultiplexes purely on events without any adapter concept.

- **`AIWS_AUI_DEMUX_5`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIWS_AUI_DEMUX_5_UNGATED` carries the demultiplexer logic of the `AIWS_DEMUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_MUX_2



• – ○ • – ○ • – ○ •

Introduction

`AIWS_AUI_MUX_2` is the adapter-based variant of the generic multiplexer for data type `WSTRING`. Unlike `AIWS_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_MUX_2` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AIWS adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AIWS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_MUX`) – shared across all port counts of this block family (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 character string (16-bit wide characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIWS_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

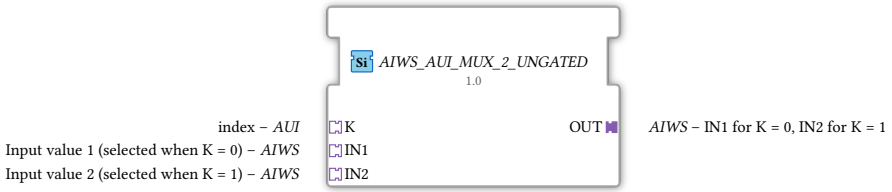
- **`AIWS_AUI_MUX_2_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIWS_AUI_MUX_2` carries the multiplexer logic of the `AIWS_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_MUX_2_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_AUI_MUX_2`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AIWS_AUI_MUX_2_UNGATED` is the adapter-based variant of the generic multiplexer for data type `WSTRING`. Unlike `AIWS_MUX_2`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` (“Adapter Unidirectional Interface”). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_MUX_2_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 2, passed through to `OUT` when the index is `K = 0` (AIWS adapter type).
- **IN2** (Socket): input adapter 2 of 2, passed through to `OUT` when the index is `K = 1` (AIWS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN2`:

1. The current value of `K.D1` determines which of the 2 input adapters (`IN1` … `IN2`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_MUX`) – shared across all port counts of this block family (AIWS_AUI_MUX_2_UNGATED … AIWS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN2`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 2 character string (16-bit wide characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIWS_MUX_2` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

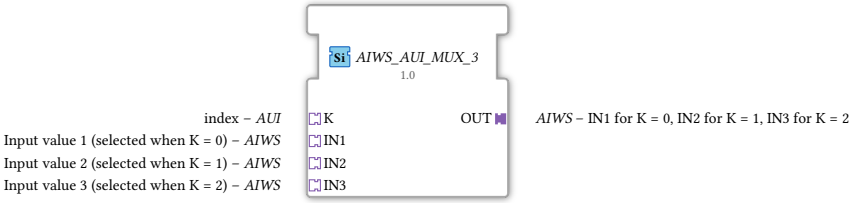
Compare with F_MUX_2, which performs the same 2:1 selection purely on data, without any adapter/event concept.

- **`AIWS_AUI_MUX_2`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIWS_AUI_MUX_2_UNGATED` carries the multiplexer logic of the `AIWS_MUX_2` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_MUX_3



• – ◦ • – ◦ • – ◦ •

Introduction

`AIWS_AUI_MUX_3` is the adapter-based variant of the generic multiplexer for data type `WSTRING`. Unlike `AIWS_MUX_3`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_MUX_3` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 3, passed through to `OUT` when the index is `K = 0` (AIWS adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to `OUT` when the index is `K = 1` (AIWS adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to `OUT` when the index is `K = 2` (AIWS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`...`IN3`:

1. The current value of `K.D1` determines which of the 3 input adapters (`IN1` … `IN3`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_MUX`) – shared across all port counts of this block family (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN3`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 character string (16-bit wide characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖ Comparison with Similar Blocks

Compare with `AIWS_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

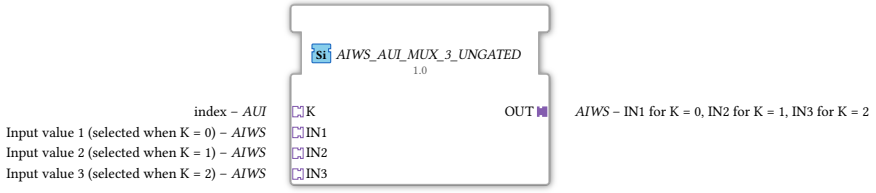
- **`AIWS_AUI_MUX_3_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIWS_AUI_MUX_3` carries the multiplexer logic of the `AIWS_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_MUX_3_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_AUI_MUX_3`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

AIWS_AUI_MUX_3_UNGATED is the adapter-based variant of the generic multiplexer for data type WSTRING. Unlike AIWS_MUX_3, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type AUI ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching AUI plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the AUI plug connected there fires its internal event E1 carrying the data value D1, AIWS_AUI_MUX_3_UNGATED evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (UINT, 0-based) arrives exclusively as the D1 value of the AUI adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type AUI): index input – selects the active input/output through the adapter's own E1/D1 event.
- **IN1** (Socket): input adapter 1 of 3, passed through to OUT when the index is K = 0 (AIWS adapter type).
- **IN2** (Socket): input adapter 2 of 3, passed through to OUT when the index is K = 1 (AIWS adapter type).
- **IN3** (Socket): input adapter 3 of 3, passed through to OUT when the index is K = 2 (AIWS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter K (type AUI) and on an event from one of the input adapters IN1…IN3:

1. The current value of K.D1 determines which of the 3 input adapters (IN1 … IN3) is currently selected.
2. The data value of that selected input is compared against the value currently held on OUT. Only on an actual change is OUT rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of K alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an AUI adapter.
- Generic implementation (GEN_AIWS_AUI_MUX) – shared across all port counts of this block family (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of K.D1 on every incoming event:

- **Any event** (selector adapter K or IN1…IN3) ⟶ read the currently selected input, update OUT on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 3 character string (16-bit wide characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIWS_MUX_3` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

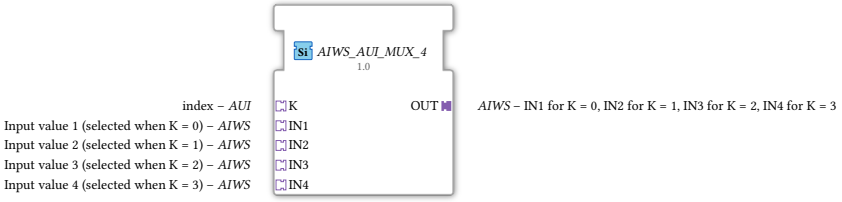
Compare with F_MUX_3, which performs the same 3:1 selection purely on data, without any adapter/event concept.

- **`AIWS_AUI_MUX_3`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIWS_AUI_MUX_3_UNGATED` carries the multiplexer logic of the `AIWS_MUX_3` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_MUX_4



• – ○ • – ○ • – ○ •

Introduction

`AIWS_AUI_MUX_4` is the adapter-based variant of the generic multiplexer for data type `WSTRING`. Unlike `AIWS_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_MUX_4` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AIWS adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AIWS adapter type).

- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AIWS adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AIWS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_MUX`) – shared across all port counts of this block family (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 character string (16-bit wide characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIWS_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

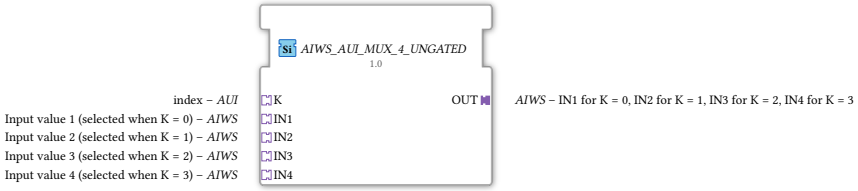
- **`AIWS_AUI_MUX_4_UNGATED`**: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIWS_AUI_MUX_4` carries the multiplexer logic of the `AIWS_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_MUX_4_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_AUI_MUX_4`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• — ○ • — ○ • — ○ •

Introduction

`AIWS_AUI_MUX_4_UNGATED` is the adapter-based variant of the generic multiplexer for data type `WSTRING`. Unlike `AIWS_MUX_4`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_MUX_4_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 4, passed through to `OUT` when the index is `K = 0` (AIWS adapter type).
- **IN2** (Socket): input adapter 2 of 4, passed through to `OUT` when the index is `K = 1` (AIWS adapter type).
- **IN3** (Socket): input adapter 3 of 4, passed through to `OUT` when the index is `K = 2` (AIWS adapter type).
- **IN4** (Socket): input adapter 4 of 4, passed through to `OUT` when the index is `K = 3` (AIWS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN4`:

1. The current value of `K.D1` determines which of the 4 input adapters (`IN1` … `IN4`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_MUX`) – shared across all port counts of this block family (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).

- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`...`IN4`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 4 character string (16-bit wide characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIWS_MUX_4` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

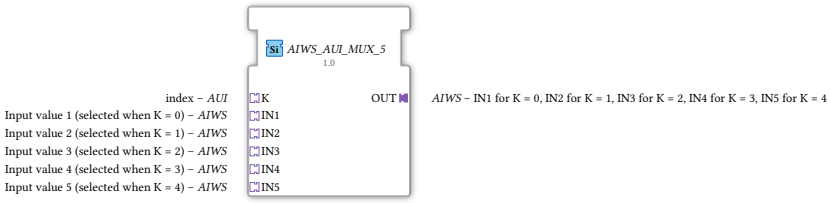
Compare with F_MUX_4, which performs the same 4:1 selection purely on data, without any adapter/event concept.

- **`AIWS_AUI_MUX_4`**: The gated variant – updates the output only on an actual value change.

Conclusion

`AIWS_AUI_MUX_4_UNGATED` carries the multiplexer logic of the `AIWS_MUX_4` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_MUX_5



• – ○ • – ○ • – ○ •

Introduction

`AIWS_AUI_MUX_5` is the adapter-based variant of the generic multiplexer for data type `WSTRING`. Unlike `AIWS_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_MUX_5` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AIWS adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AIWS adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AIWS adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AIWS adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AIWS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_MUX`) – shared across all port counts of this block family (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 character string (16-bit wide characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIWS_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).
No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

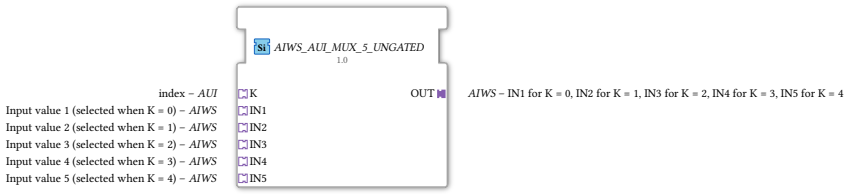
- `AIWS_AUI_MUX_5_UNGATED`: Ungated variant – updates the output on every run, even without a value change.

Conclusion

`AIWS_AUI_MUX_5` carries the multiplexer logic of the `AIWS_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

AIWS_AUI_MUX_5_UNGATED

> **i UNGATED variant:** This block is the ungated version of `AIWS_AUI_MUX_5`. It suppresses **no** unchanged repeats – every newly computed result is forwarded unconditionally, even without a value change. This matters for consumers that need a periodic cadence regardless of value change (e.g. derivative/frequency calculations that would otherwise fail to decay toward zero). Any change-detection/gating statements further down this page do **not** apply to this block.



• – ○ • – ○ • – ○ •

Introduction

`AIWS_AUI_MUX_5_UNGATED` is the adapter-based variant of the generic multiplexer for data type `WSTRING`. Unlike `AIWS_MUX_5`, it does not receive the selection index through a REQ event with an associated K data input, but through its own adapter socket **K** of type `AUI` ("Adapter Unidirectional Interface"). This lets the index be fed directly from another block with a matching `AUI` plug, without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- No explicit event inputs on the block itself. The index is received exclusively through the adapter socket **K**: as soon as the `AUI` plug connected there fires its internal event `E1` carrying the data value `D1`, `AIWS_AUI_MUX_5_UNGATED` evaluates the index internally and triggers processing.

Event Outputs

- No event outputs.

Data Inputs

- No direct data inputs. The index (`UINT`, 0-based) arrives exclusively as the `D1` value of the `AUI` adapter connected to socket **K**.

Data Outputs

- No direct data outputs.

Adapters

- **K** (Socket, type `AUI`): index input – selects the active input/output through the adapter's own `E1`/`D1` event.
- **IN1** (Socket): input adapter 1 of 5, passed through to `OUT` when the index is `K = 0` (AIWS adapter type).
- **IN2** (Socket): input adapter 2 of 5, passed through to `OUT` when the index is `K = 1` (AIWS adapter type).
- **IN3** (Socket): input adapter 3 of 5, passed through to `OUT` when the index is `K = 2` (AIWS adapter type).
- **IN4** (Socket): input adapter 4 of 5, passed through to `OUT` when the index is `K = 3` (AIWS adapter type).
- **IN5** (Socket): input adapter 5 of 5, passed through to `OUT` when the index is `K = 4` (AIWS adapter type).
- **OUT** (Plug): output adapter, forwards whichever input the index selected (AIWS adapter type).

Functionality

The function block re-evaluates the current value of **K** on every incoming event – both on an event from the selector adapter `K` (type `AUI`) and on an event from one of the input adapters `IN1`…`IN5`:

1. The current value of `K.D1` determines which of the 5 input adapters (`IN1` … `IN5`) is currently selected.
2. The data value of that selected input is compared against the value currently held on `OUT`. Only on an actual change is `OUT` rewritten and its adapter event sent (see "Change Detection" below).

As a result, a pure change of `K` alone also propagates to the output immediately, even if the data value of the newly selected input hasn't changed since its own last event.

Technical Details

- Adapter-based index input instead of the classic REQ/K input pair – reduces wiring when the index is already available as an `AUI` adapter.
- Generic implementation (`GEN_AIWS_AUI_MUX`) – shared across all port counts of this block family (AIWS_AUI_MUX_2 … AIWS_AUI_MUX_5_UNGATED).
- **Change detection**: the written output plug is only updated – and its adapter event only sent – when the new value differs from the value currently held on that plug. If the value is unchanged, no adapter event is sent, avoiding redundant updates for downstream peers.

State Overview

The function block has no explicit state machine; instead it re-evaluates the current value of `K.D1` on every incoming event:

- **Any event** (selector adapter `K` or `IN1`…`IN5`) ⟶ read the currently selected input, update `OUT` on a value change and send its adapter event.

Application Scenarios

- Choosing between up to 5 character string (16-bit wide characters) sources through a centrally managed adapter index
- Replacing a REQ/K input pair with a single AUI adapter link when the index is already provided as an adapter by another block
- Building selection networks in which several MUX blocks share the same index adapter

⚖️ Comparison with Similar Blocks

Compare with `AIWS_MUX_5` (same selection logic, but the index arrives through a classic **REQ** event plus **K** data input instead of an adapter).

No purely data-based `F_MUX` counterpart exists for this input count in the `iec61131-3` library.

- **`AIWS_AUI_MUX_5`**: The gated variant – updates the output only on an actual value change.

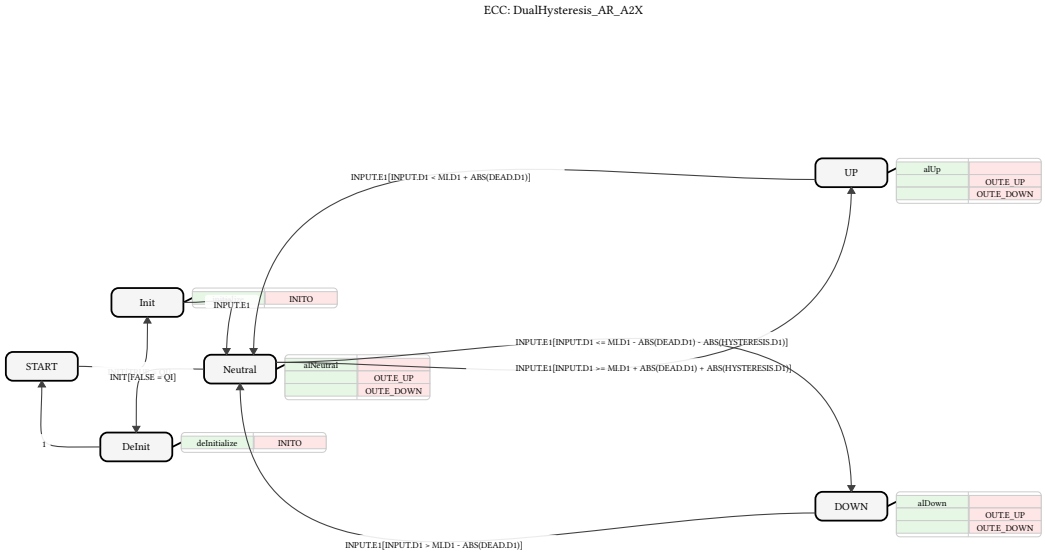
Conclusion

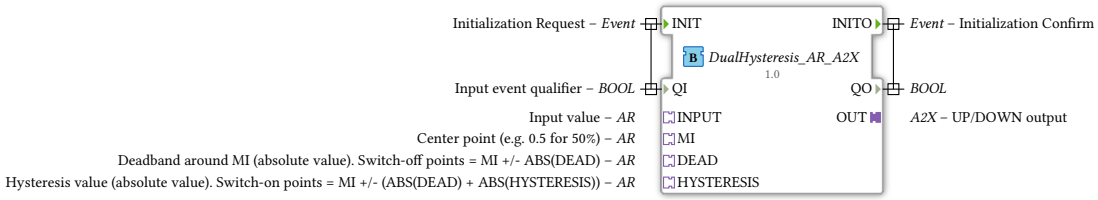
`AIWS_AUI_MUX_5_UNGATED` carries the multiplexer logic of the `AIWS_MUX_5` family over to a purely adapter-based index supply. This makes it the right choice whenever the selection index is already available as an `AUI` adapter from another block and no additional event/data wiring for the index is wanted.

2.8.0.4 Signalprocessing

Hysteresis

DualHysteresis_AR_A2X





• — ○ • — ○ • — ○ •

Introduction

The **DualHysteresis_AR_A2X** function block implements a two-way analog-to-digital conversion with configurable hysteresis and deadband. It monitors an analog input value and generates discrete UP/DOWN signals based on the relationship between the input and a configurable center point (MI). The block distinguishes between a switch-on threshold (wider due to hysteresis) and a switch-off threshold (narrower due to deadband only), providing clean and stable digital output transitions without chatter around the switching points. The outputs are provided via a single unidirectional A2X adapter interface, combining both event and data signals for UP and DOWN states.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Initialization Request.

Event Outputs

Event	Type	Comment
INIT.O	EInit	Initialization Confirm.

Data Inputs

Data	Type	Comment
QI	BOOL	Input event qualifier. When QI = TRUE, normal processing is enabled. When FALSE, the block operates in safe state.

Data Outputs

Data	Type	Comment
QO	BOOL	Output qualifier reflecting the last accepted QI value during operation.

Adapters

Direction	Name	Type	Comment
Plug	OUT	adapter::types::unidirectional::A2X	UP/DOWN output providing E_UP, E_DOWN events and UP, DOWN boolean data.
Socket	INPUT	adapter::types::unidirectional::AR	Analog input value (event E1, data D1).
Socket	MI	adapter::types::unidirectional::AR	Center point (e.g., 0.5 for 50%).
Socket	DEAD	adapter::types::unidirectional::AR	Deadband around MI (absolute value). Switch-off points = MI ± ABS(DEAD).
Socket	HYSTERESIS	adapter::types::unidirectional::AR	Hysteresis value (absolute value). Switch-on points = MI ± (ABS(DEAD) + ABS(HYSTERESIS)).

Functionality

The block converts a continuous analog input (via the INPUT adapter) into discrete two-state output signals. The conversion depends on the analog input value INPUT.D1, the center point MI.D1, the deadband DEAD.D1, and the hysteresis HYSTERESIS.D1.

The switching behavior is as follows:

- **Switch-on UP** (inclusive): When the input rises to or above MI.D1 + ABS(DEAD.D1) + ABS(HYSTERESIS.D1), the output transitions to the UP state.
- **Switch-off UP** (strict): When the input falls strictly below MI.D1 + ABS(DEAD.D1), the block leaves the UP state and returns to Neutral.
- **Switch-on DOWN** (inclusive): When the input falls to or below MI.D1 - ABS(DEAD.D1) - ABS(HYSTERESIS.D1), the output transitions to the DOWN state.
- **Switch-off DOWN** (strict): When the input rises strictly above MI.D1 - ABS(DEAD.D1), the block leaves the DOWN state and returns to Neutral.

All transitions are triggered by the INPUT.E1 event, which indicates that a new analog value is available on INPUT.D1.

The algorithms executed in each state set the UP and DOWN data outputs on the OUT adapter accordingly. The corresponding events OUT.E_UP and OUT.E_DOWN are emitted on each state change.

If QI = FALSE, the block remains in a safe state and sets both UP and DOWN outputs to FALSE, suppressing any digital output signals.

Technical Features

- **Deadband & Hysteresis Separation**: The switch-off threshold is determined only by the deadband, while the switch-on threshold adds the hysteresis value. This creates a well-defined gap between on and off points, preventing oscillation due to noise.
- **Absolute Value Handling**: The DEAD and HYSTERESIS values are taken as absolute values internally, simplifying configuration even when negative values are provided.

- **Event-Driven Processing**: All evaluations occur on the `INPUT.E1` event, ensuring deterministic behavior aligned with incoming analog updates.
- **Parameterizable Center Point**: The `MI` adapter allows setting an arbitrary center point (e.g., 50%), making the block suitable for a wide range of scaling applications.
- **Unidirectional Adapter Output**: The output is bundled in a single A2X adapter containing both events (`E_UP`, `E_DOWN`) and data (`UP`, `DOWN`), simplifying wiring in 4diac applications.
- **Initialization and De-initialization**: The block supports explicit initialization via `INIT` and safe shutdown via de-initialization, with `QO` reflecting the qualifier state.

State Overview

The internal state machine consists of six states:

State	Description
`START`	Initial power-up state.
`Init`	Performs initialization; sets `QO := QI` and clears UP/DOWN outputs; emits `INITO`.
`Neutral`	Idle state with no UP/DOWN activity; both outputs are `FALSE`.
`UP`	Active UP state; sets `OUT.UP := TRUE`, `OUT.DOWN := FALSE`; emits `E_UP`.
`DOWN`	Active DOWN state; sets `OUT.UP := FALSE`, `OUT.DOWN := TRUE`; emits `E_DOWN`.
`DeInit`	De-initialization state; clears outputs and sets `QO := FALSE`; emits `INITO`.

Transitions:

- `START` → `Init` on `INIT` with `QI = TRUE`.
- `Init` → `Neutral` on first `INPUT.E1` event.
- `Neutral` → `UP` when `INPUT.D1 >= MI.D1 + ABS(DEAD.D1) + ABS(HYSTERESIS.D1)`.
- `Neutral` → `DOWN` when `INPUT.D1 <= MI.D1 - ABS(DEAD.D1) - ABS(HYSTERESIS.D1)`.
- `UP` → `Neutral` when `INPUT.D1 < MI.D1 + ABS(DEAD.D1)`.
- `DOWN` → `Neutral` when `INPUT.D1 > MI.D1 - ABS(DEAD.D1)`.
- `Neutral` → `DeInit` on `INIT` with `QI = FALSE`.
- `DeInit` → `START` unconditionally.

Application Scenarios

- **Level Monitoring**: Monitoring tank or silo fill levels, where a sensor provides a continuous analog signal and discrete high/low alarms are required with hysteresis to avoid relay chattering.
- **Temperature Control**: Switching heating or cooling elements based on temperature thresholds with a deadband/hysteresis configuration to prevent rapid cycling.
- **Limit Switch Replacement**: Replacing mechanical limit switches with non-contact analog sensors, using the block to derive clean digital position signals.
- **Motor Control**: Generating direction commands (UP/DOWN) for actuators (e.g., valves, drives) based on a setpoint and allowable deadband.
- **Industrial Automation**: In PLC and distributed control systems, where reliable two-state signals must be derived from analog process variables.

Comparison with Similar Blocks

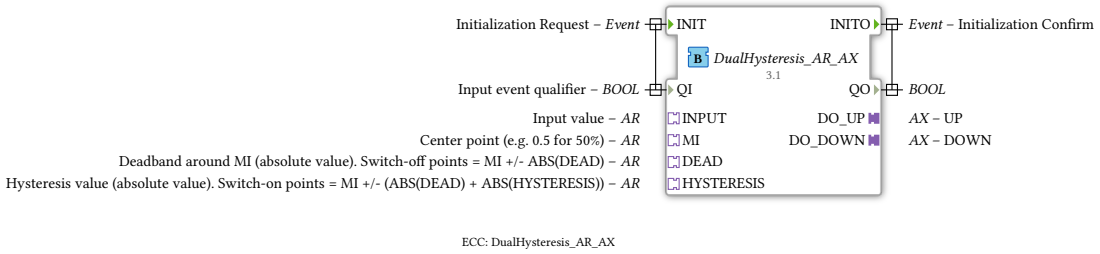
Compared to a simple comparator or single-threshold hysteresis block, **DualHysteresis_AR_A2X** offers:

- **Two-way (bidirectional) evaluation**: Both rising and falling input signals are processed symmetrically, with separate switch-on and switch-off thresholds.
- **Independent deadband and hysteresis parameters**: The user can define the deadband (non-switching zone) and hysteresis (influence on switch-on point) separately, providing greater flexibility than blocks with a single parameter.
- **Unified output adapter**: All output signals (UP/DOWN events and data) are packaged in one adapter, reducing wiring complexity in comparison to blocks with multiple separate output ports.
- **Safe-state behavior**: When `QI` is false, the block guarantees both outputs are reset to `FALSE`, which is not always the case in simpler hysteresis implementations.

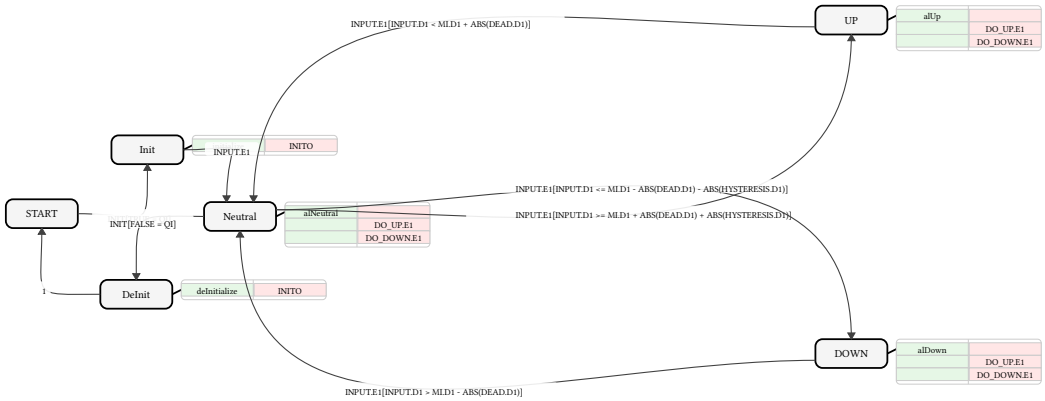
Conclusion

The **DualHysteresis_AR_A2X** function block provides a robust and flexible solution for converting analog signals into discrete UP/DOWN control outputs. By separating deadband and hysteresis, it offers precise control over switching behavior, preventing oscillation and ensuring reliable operation in industrial automation environments. Its event-driven design, adapter-based interfaces, and clear state machine make it well-suited for integration into 4diac-based control applications.

DualHysteresis_AR_AX



ECC: DualHysteresis_AR_AX



(No image available)

• – ○ • – ○ • – ○ •

Introduction

The function block `DualHysteresis_AR_AX` performs dual analog-to-digital conversion with hysteresis. It compares an analog input signal with a defined midpoint (`MI`) to output control commands via two separate digital outputs (UP and DOWN). This corresponds to the operation of a classic three-position controller with deadband and hysteresis thresholds. The block is designed for signal processing in control and regulation applications and uses adapters for structured signal coupling.

Interface Structure

Event Inputs

- `INIT` (Type: `EInit`): Initialization request. Starts or stops the function block depending on the data input `QI`.

Event Outputs

- `INITO` (Type: `EInit`): Initialization confirmation. Signals the successful transition to the initialized or deinitialized state.

Data Inputs

- `QI` (Type: `BOOL`): Input event qualifier. Determines whether the function block should operate actively (`TRUE`) or be placed in the safe deinitialization state (`FALSE`).

Data Outputs

- `QO` (Type: `BOOL`): Output event qualifier. Reflects the current status and operational readiness of the function block.

Adapters

- **Plugs:**
- `DO_UP` (Type: `adapter::types::unidirectional::AX`): Output adapter for the upward movement/ upper threshold (UP).
- `DO_DOWN` (Type: `adapter::types::unidirectional::AX`): Output adapter for downward movement/ lower threshold (DOWN).
- **Sockets:**
- `INPUT` (Type: `adapter::types::unidirectional::AR`): The current analog measurement or input value.
- `MI` (Type: `adapter::types::unidirectional::AR`): The center point or setpoint of the control (e.g., `0.5` for 50%).
- `DEAD` (Type: `adapter::types::unidirectional::AR`): The deadband around the center point (absolute value). Defines the cutoff points.
- `DEAD` (Type: `adapter::types::unidirectional::AR`): The deadband around the center point (absolute value). Defines the cutoff points. * `HYSTERESIS` (Type: `adapter::types::unidirectional::AR`): The hysteresis (absolute value). Together with the deadband, it defines the switch-on points.

Functionality

The state switching is event-driven via the input `INPUT.E1`. The mathematical evaluation ensures that negative parameter inputs for the deadband (`DEAD`) and the hysteresis (`HYSTERESIS`) are handled by using absolute values (`ABS`).

The logical thresholds behave as follows:

- **Switch to UP (switch to UP):**

$$INPUT \geq MI + |DEAD| + |HYSTERESIS| \tag{2.124}$$

(Outputs: `DO_UP.D1 = TRUE`, `DO_DOWN.D1 = FALSE`)

- **Switch UP (Return to Neutral):**

$$INPUT < MI + |DEAD| \tag{2.125}$$

(Ausgänge: `DO_UP.D1 = FALSE`, `DO_DOWN.D1 = FALSE`)

- **Einschalten DOWN (Wechsel nach DOWN):**

$$INPUT \leq MI - |DEAD| - |HYSTERESIS| \tag{2.126}$$

(Ausgänge: `DO_UP.D1 = FALSE`, `DO_DOWN.D1 = TRUE`)

- **Ausschalten DOWN (Wechsel zurück nach Neutral):**

$$INPUT > MI - |DEAD| \tag{2.127}$$

(Outputs: `DO_UP.D1 = FALSE`, `DO_DOWN.D1 = FALSE`)

Technical Features

- **Error Safety through Absolute Values:** The threshold calculations explicitly use the `ABS` function for the Parameters `DEAD` and `HYSTERESIS` prevent malfunctions caused by accidentally set negative values.

Safe State: In the deinitialized state (`QI` = `FALSE`) or in the state `Neutral`, both digital outputs (`DO_UP.D1` and `DO_DOWN.D1`) are forcibly set to `FALSE`.

Safe State: In the deinitialized state (`QI` = `FALSE`) or in the state `Neutral`, both digital outputs (`DO_UP.D1` and `DO_DOWN.D1`) are forcibly set to `FALSE`. * **Adapter Connection:** The use of `AR` (Analog Receiver) and

AX (Digital Actuator) adapters ensures standardized, clean encapsulation of the signal paths within the 4diac IDE.

State Overview

The module is based on a state machine (ECC) with the following states:

1. START: Idle state before initialization.
2. Init: Initialization phase. Sets the outputs to a safe state.
3. Neutral: The input value is within the permissible range. Both outputs are inactive.
4. UP: The upper turn-on point has been exceeded. Output DO_UP is active.
5. DOWN: The lower activation point has been reached. Output DO_DOWN is active.
6. DeInit: Transition state for deactivating the function block.

Application Scenarios

- **Three-point controller:** Temperature control (e.g., heating / neutral / cooling) or level control (e.g., open inlet valve / neutral / open outlet valve).
- **Track Guidance and Steering Systems:** Automatic steering systems in agricultural or logistics where deviations from the track center (MI) to the left (DOWN) or right (UP) need to be corrected.
- **Setpoint Band Monitoring:** Monitoring of physical quantities where an actuator should only be activated when there are significant deviations (deadband + hysteresis) to prevent constant actuator "flutter."

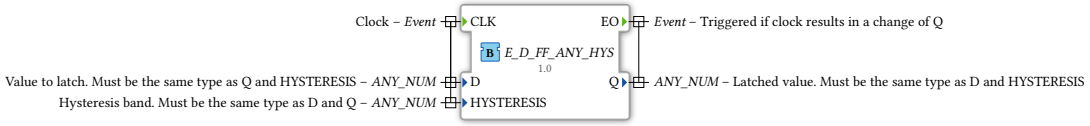
Comparison with Similar Components

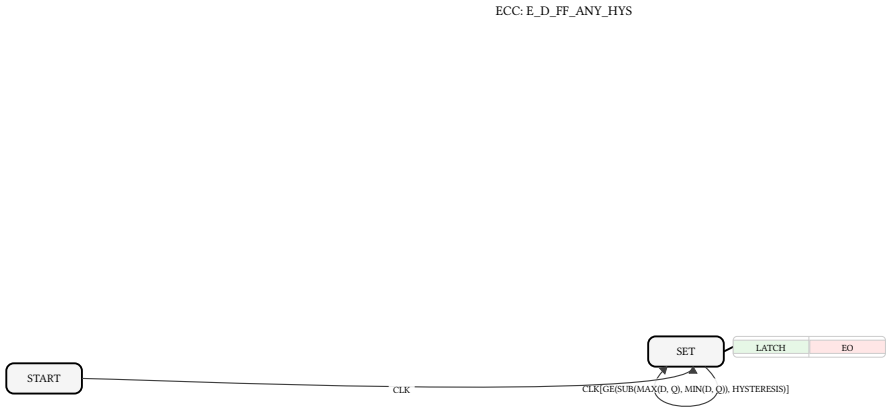
Classic hysteresis components (e.g., two-point controllers) only control a single digital output. The DualHysteresis_AR_AX combines two opposing hysteresis loops in a single component (three-point behavior). Furthermore, compared to standard function blocks with elementary data types, it is distinguished by the use of unidirectional adapters (AR and AX), which simplifies wiring in complex applications.

Conclusion

The DualHysteresis_AR_AX is a robust and flexible function block for implementing three-point control. Thanks to the mathematical validation of the limit values and the clearly defined state behavior, it is ideally suited for demanding control tasks where actuators must be controlled gently and without chatter.

E_D_FF_ANY_HYS





• – ○ • – ○ • – ○ •

Introduction

The function block `E_D_FF_ANY_HYS` is an edge-triggered D flip-flop (data latch) for numeric data types with integrated hysteresis. It serves to store (latch) an input value and make it available at the output. However, an output update and the triggering of the output event only occur with a clock signal if the change in the input value compared to the current output value reaches or exceeds the defined hysteresis threshold.

Interface Structure

Event Inputs

- **CLK**: Clock Input. Upon the occurrence of this event, the system checks whether the input value `D` has changed by at least the value of `Q` compared to the current output value `HYSTERESIS`.

Event Outputs

- **EO**: Event Output. Triggered when a clock event (`CLK`) results in an actual value change and update of the output `Q`.

Data Inputs

- **D** (`ANY_NUM`): The current data value to be read.
- **HYSTERESIS** (`ANY_NUM`): The hysteresis band. Specifies the minimum difference that must exist between `D` and `Q` for the output to be updated.

Data Outputs

- **Q** (`ANY_NUM`): The stored (latched) value.

Adapters

- *No adapters available.*

Functionality

The function block operates internally using an Execution Control Chart (ECC) and a simple assignment algorithm:

1. **Initialization**: On first startup, the block is in state `START`. The first `CLK` event leads directly to state `SET`. The algorithm `LATCH` is executed, which writes the current value of `D` to the output `Q`. Simultaneously, the output event `EO` is sent.

The process executes the algorithm `LATCH`, which writes the current value of `D` to the output `Q`.
... 2. **Hysteresis Check for Subsequent Events**: If the function block is in state `SET`, each subsequent `CLK` event will only lead to a re-execution of `SET` (and thus to an update of `Q` and the firing of `EO`) if the following condition is met:

$$| D - Q | \geq \text{HYSTERESIS} \tag{2.128}$$

In the ECC condition, this is mathematically implemented using the formula `GE(SUB(MAX(D, Q), MIN(D, Q)), HYSTERESIS)`. This corresponds to the absolute difference between the input value `D` and the currently stored value `Q`. 3. If the difference is less than the value of `HYSTERESIS`, the event is ignored, the output `Q` remains unchanged, and no `EO` event is triggered.

Technical Features

- **Generic Data Type**: By using the data type `ANY_NUM` for the inputs and outputs, the function block is highly reusable and can be interconnected with various numeric data types (e.g., `INT`, `REAL`, `LREAL`).
- **Efficient Difference Calculation**: The difference calculation using `SUB(MAX(D, Q), MIN(D, Q))` ensures that a positive absolute value is always used for comparison with the hysteresis, regardless of whether the new value is greater or less than the old value.

Status Overview

START –(CLK)–> [SET (Algorithmus: LATCH, Ausgang: EO)] | +–(CLK [Differenz >= HYSTERESIS])–> (erneuter Aufruf von SET)

- **START**: Waiting state before the first clock cycle.
- **SET**: State in which the input value is accepted and output. This state is called cyclically at each valid clock cycle (taking hysteresis into account).

Application Scenarios

- **Noise Filtering for Analog Sensor Values**: Reduction of event floods in distributed systems. If a sensor (e.g., a temperature sensor) continuously delivers slightly fluctuating values, hysteresis prevents new control events from being constantly distributed throughout the system in response to minimal changes.
- **Limit Monitoring with Deadband**: Prevention of signal bounce when transmitting process values to visualizations (HMI) or databases.

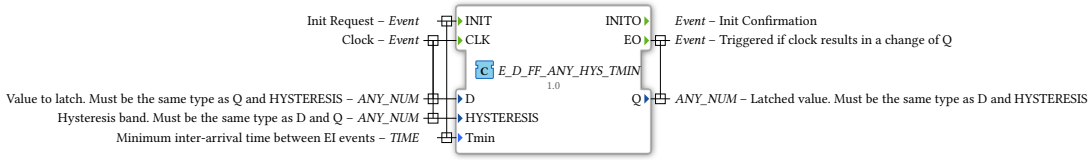
Comparison with Similar Function Blocks

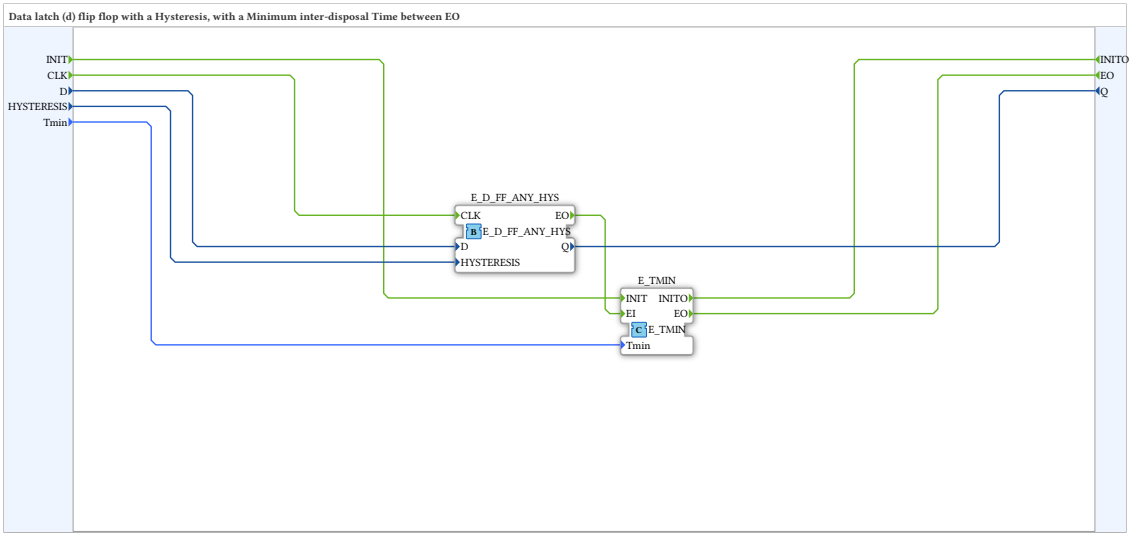
- **E_D_FF**: The standard D flip-flop typically reacts at the Boolean level or unconditionally stores values at each clock cycle. `E_D_FF_ANY_HYS` additionally filters the values using the hysteresis condition.
- **Hysteresis blocks (e.g., with Boolean output)**: Typical hysteresis blocks compare a value with fixed thresholds and output a `BOOL` signal (e.g., threshold switch). `E_D_FF_ANY_HYS`, on the other hand, outputs the numerical value itself as soon as it has changed significantly.

Conclusion

The `E_D_FF_ANY_HYS` is an extremely useful block for data- and event-driven signal processing. It combines the memory function of a D flip-flop with intelligent noise and change filtering for numerical signals, thus significantly contributing to performance optimization and reducing the event load in IEC 61499 applications.

E_D_FF_ANY_HYS_TMIN





• – ○ • – ○ • – ○ •

The function block `E_D_FF_ANY_HYS_TMIN` is a composite function block for IEC 61499-based systems. It implements an edge-triggered D flip-flop (data latch) with integrated threshold hysteresis and a minimum time interval limit (`Tmin`) for outgoing events.

This block is used to filter continuous or high-frequency changes in a numerical input signal. A new output event is only generated if the signal change exceeds the defined hysteresis and a defined minimum time interval has elapsed since the last output event. This effectively prevents so-called “event flooding” in distributed control systems.

Event	Type	Description	Associated Data
INIT	EInit	Initializes the function block and sets the minimum time limit.	`Tmin`
CLK	Event	Clock signal. When this event occurs, the value at input `D` is evaluated.	`D`, `HYSTERESIS`
Event	Type	Description	Associated Data
:—	:—	:—	:—
INITO	EInit	Confirmation of successful initialization.	None
EO	Event	Triggered when a CLK event has caused a valid change to the output value `Q` and the minimum time `Tmin` has elapsed.	`Q`
Variable	Type	Description	
:—	:—	:—	
D	ANY_NUM	The input value (Data) to be stored.	
HYSTERESIS	ANY_NUM	The hysteresis band. Determines how much `D` must differ from the current output value `Q` to trigger a change.	
Tmin	TIME	The minimum time interval that must elapse between two consecutive `EO` events.	
Variable	Type	Description	
:—	:—	:—	
Q	ANY_NUM	The stored and filtered output value (latched value).	

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters
This function block does not use any adapters.

Functionality
The module internally combines two specialized function blocks:

1. A hysteresis memory module (`E_D_FF_ANY_HYS`)
2. A time-limited event timer (`E_TMIN`)

As soon as an event arrives at input `CLK`, the internal module `E_D_FF_ANY_HYS` reads the current value from `D`.

- If the difference between the new value `D` and the currently stored value `Q` lies outside the range defined by `HYSTERESIS`, `Q` is updated with the new value and an internal event is forwarded to the limiter `E_TMIN`.
- If the change lies within the hysteresis band, the value remains unchanged and no event is forwarded.

The function block `E_TMIN` monitors the forwarding of this event to the output `EO`. It ensures that at least the time interval `Tmin` elapses between two `EO` events. Rapidly successive events are delayed or filtered accordingly to protect the subsequent logic from excessive event load.

- **Generic Data Types (`ANY_NUM`):** By using the generic data type `ANY_NUM` for `D`, `HYSTERESIS`, and `Q`, the function block is highly reusable and can be used for various numeric types (e.g., `INT`, `REAL`, `LREAL`).
- **Combined Filtering:** The function block combines amplitude filtering (hysteresis) and time filtering (Tmin) in a single component.

1. **Uninitialized:** The function block is waiting for the event `INIT`.
2. **Initialized / Ready:** The function block has configured `Tmin` and is waiting for events at input `CLK`.
3. **Value Change Blocked (Hysteresis):** A `CLK` event arrives, but the change in `D` is too small. `Q` remains unchanged; there is no `EO`.
4. **Value Change Active (Time Delay/Filtering):** A `CLK` event arrives, and the change is large enough. When `Tmin` expires, `Q` is updated and `EO` is immediately output. If `Tmin` is still active, the timer's protection function is activated.

- **Analog Sensor Value Processing:** Attenuation and noise reduction for noisy analog measurements (e.g., temperature, pressure, or fill level).
- **Network-Friendly Data Transmission:** Reduction of publish events in IoT or SCADA systems (e.g., MQTT). Data is only sent when a value has changed significantly, but at most every "X" seconds.
- **Resource Efficiency in PLC Controllers:** Relieving the CPU of cyclic calculations that would be triggered by high-frequency signal changes.

2.9 Comparison with Similar Components

- **`E_D_FF`:** Stores values directly at each clock cycle without hysteresis or time limiting. This results in extremely high event density with turbulent signals.
- **`E_D_FF_ANY_HYS`:** While it does have hysteresis, it offers no protection against rapid, continuous signal changes (e.g., during fast ramps), which would still generate too many events. `E_D_FF_ANY_HYS_TMIN` solves this problem with the integrated `Tmin` component.

The `E_D_FF_ANY_HYS_TMIN` is an extremely useful component for signal conditioning in industrial environments. It combines value-based filtering with time-based rate limiting. This makes it ideal for stable control architectures where sensor values need to be precisely acquired, but downstream systems must be protected from overload.

Technical Features

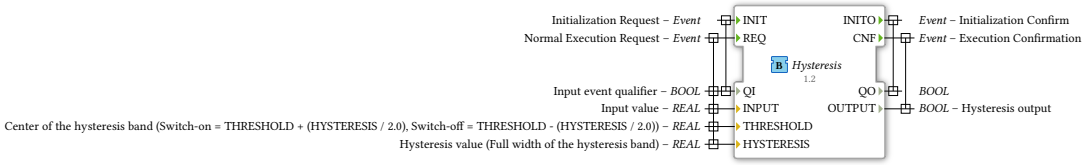
State Overview

Application Scenarios

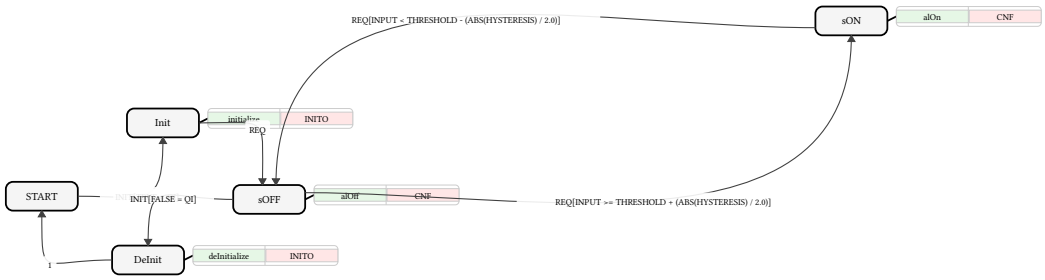
Comparison with Similar Function Blocks

Conclusion

Hysteresis



ECC: Hysteresis



• — ○ • — ○ • — ○ •

Introduction

The function block `Hysteresis` serves as an analog-to-digital converter with an integrated hysteresis curve. It converts an analog input signal (`INPUT`) into a binary output signal (`OUTPUT`). By defining a threshold (`THRESHOLD`) and a hysteresis width (`HYSTERESIS`), unwanted bounce or rapid oscillation of the output with noisy input signals is effectively prevented.

• — ○ • — ○ • — ○ •

Interface Structure

Event Inputs

- **INIT**: Initializes the function block. Updates the output qualifier `QO` based on `QI`.
- **REQ**: Triggers a calculation of the hysteresis state based on the current input values.

Event Outputs

- **INITO**: Confirms completion of initialization or deinitialization.
- **CNF**: Confirms execution of the calculation and signals that an updated output value is available.

Data Inputs

- **QI** (BOOL): Input event qualifier. Controls the validity of the input data and the activation of the function block.
- **INPUT** (REAL): The analog input signal to be monitored.
- **THRESHOLD** (REAL, default value: `0.0`): The midpoint of the hysteresis band (setpoint).
- **HYSTERESIS** (REAL, default value: `0.1`): The total width of the hysteresis band.

Data Outputs

- **QO** (BOOL): Output event qualifier (indicates whether the function block is active and ready for operation).
- **OUTPUT** (BOOL): The digital output signal of the hysteresis element.

Adapters

This function block does not use adapters.

• – ◦ • – ◦ • – ◦ •

Operation

The function block operates with a symmetrical hysteresis band around the defined midpoint (`THRESHOLD`). To ensure consistent behavior even with negative hysteresis values, the absolute value (`ABS`) of the hysteresis is used internally.

The switching thresholds are calculated as follows:

- **Switch-on point:** $THRESHOLD + \frac{|HYSTERESIS|}{2.0}$
- **Switch-off point:** $THRESHOLD - \frac{|HYSTERESIS|}{2.0}$

Switching logic

1. If the function block is in the **sOFF** state (output is `FALSE`), the output switches to `TRUE` (**sON**) as soon as the input signal reaches or exceeds the upper threshold:

$$\text{INPUT} \geq \text{THRESHOLD} + \frac{|\text{HYSTERESIS}|}{2.0} \tag{2.129}$$

1. If the function block is in the **sON** state (output is The output (`TRUE`) only switches back to `FALSE` (**sOFF**) when the input signal falls well below the lower limit:

$$\text{INPUT} < \text{THRESHOLD} - \frac{|\text{HYSTERESIS}|}{2.0} \tag{2.130}$$

• – ◦ • – ◦ • – ◦ •

Technische Besonderheiten

- **Schwingungsschutz:** Die Ausschaltbedingung verwendet eine strikte Ungleichung ($<$), während die Einschaltbedingung inklusiv ist ($\geq$). Dies verhindert Oszillationen exakt an den Grenzen des Hysteresebands.
- **Sicherheitsverriegelung über QI:** Der Ausgang `OUTPUT` kann nur dann den Zustand `TRUE` annehmen, wenn der Eingangs-Qualifier `QI` auf `TRUE` gesetzt ist. Ist `QI` im Zustand `FALSE`, bleibt der Ausgang gesperrt (`FALSE`).
- **Robustheit:** Durch die Verwendung von `ABS(HYSTERESIS)` im Algorithmus führt die Eingabe einer negativen Hysterese nicht zu einem Fehlverhalten des Bausteins.

• – ◦ • – ◦ • – ◦ •

Zustandsübersicht

Der Funktionsbaustein ist als Basic-FB mit einer Execution Control Chart (ECC) realisiert:

1. **START**: Der Standard-Ruhezustand nach dem Systemstart.
2. **Init**: Wird durch das Ereignis `INIT` bei gesetztem `QI = TRUE` aufgerufen. Initialisiert den Baustein und setzt den Ausgang `OUTPUT` auf `FALSE`.
3. **sOFF**: Der Zustand, in dem die Ausschaltbedingung erfüllt ist (`OUTPUT = FALSE`).
4. **sON**: Der Zustand, in dem die Einschaltbedingung erfüllt ist (`OUTPUT = TRUE` sofern `QI = TRUE`).

5. **DeInit**: Wird durch `INIT` bei `QI = FALSE` aufgerufen. Setzt `QO` und `OUTPUT` zurück auf `FALSE`.

• – ○ • – ○ • – ○ •

Anwendungsszenarien

- **Zweipunktregler:** Temperaturregelung (z. B. Heizung ein bei $< 19^\circ$ C, aus bei $> 21^\circ$ C with a threshold of 20° C and a hysteresis of 2° C).
- **Level Monitoring:** Activates a pump when a maximum level is reached and deactivates it when the minimum level is reached.
- **Noise Reduction:** Digitizes noisy analog sensor signals (e.g., pressure or brightness sensors) to eliminate fluctuating signal levels at the output.

• – ○ • – ○ • – ○ • –

Comparison with Similar Components

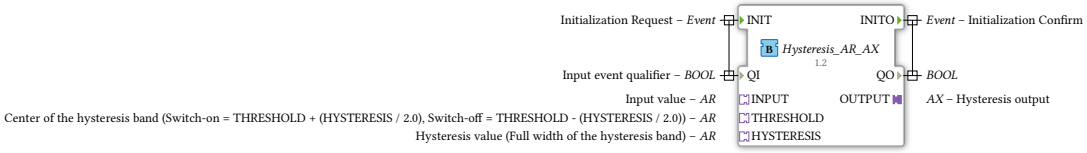
Compared to a simple comparator (e.g., `GT` or `GE`), which switches immediately upon any minimal exceedance or fall below a specified value, the `Hysteresis` component offers inherent signal stabilization through its two switching thresholds. It combines the logic of two comparators and an RS flip-flop into a single, easily accessible component.

• – ○ • – ○ • – ○ •

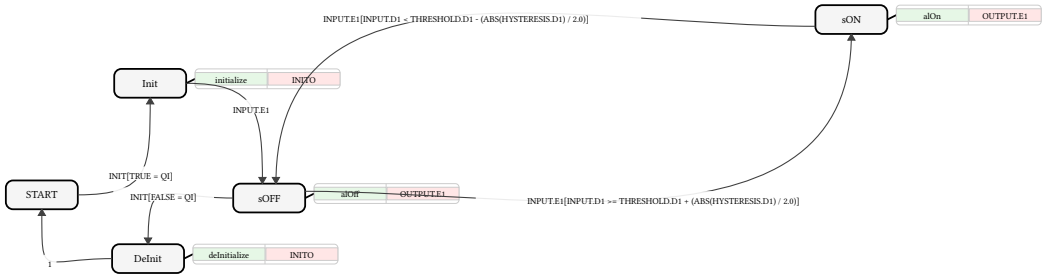
Conclusion

The `Hysteresis` component is an essential tool in signal processing and control engineering within the IEC 61499 standard. It provides a robust, easy-to-configure solution for generating stable binary control signals from continuous analog input values.

Hysteresis_AR_AX



ECC: Hysteresis_AR_AX



• – ○ • – ○ • – ○ •

Introduction

The function block `Hysteresis_AR_AX` is a basic function block (BFB) for event-driven signal processing according to IEC 61499. It performs analog-to-digital conversion of an analog input signal, taking into account an adjustable hysteresis. This prevents small signal fluctuations or noise around a threshold value from causing rapid, unwanted switching back and forth of the digital output (oscillation).

Interface Structure

Event Inputs

- **INIT**: Initialization request (`EInit`). Enables or disables the block in conjunction with the input `QI`.

Event Outputs

- **INITO**: Initialization Confirmation (`EInit`). Signals the completion of initialization or deinitialization in conjunction with `QO`.

Data Inputs

- **QI** (BOOL): Input Event Qualifier. Determines whether the function block should be active (`TRUE`) or deinitialized (`FALSE`).

Data Outputs

- **QO** (BOOL): Output Event Qualifier. Displays the current status of the function block activation.

Adapters

The function block uses standardized, unidirectional adapters for clean, modular coupling of data and events.

- **Sockets (Input Adapters):**
- **INPUT** (`adapter::types::unidirectional::AR`): The analog input value to be monitored. The value is present at `INPUT.D1`; the evaluation is triggered by the event `INPUT.E1`.
- **THRESHOLD** (`adapter::types::unidirectional::AR`): The midpoint of the hysteresis band (`THRESHOLD.D1`).
- **HYSTERESIS** (`adapter::types::unidirectional::AR`): The total width of the hysteresis band (`HYSTERESIS.D1`).
- **Plugs (Output Adapters):**
- **OUTPUT** (`adapter::types::unidirectional::AX`): The digital output signal. Outputs the switching result at data point `OUTPUT.D1` (BOOL) and triggers the output via the event `OUTPUT.E1`.

Functionality

The hysteresis is defined symmetrically around the threshold value (`THRESHOLD`). The total width of the hysteresis band is specified by the parameter `HYSTERESIS`.

Calculation Formulas

To avoid misconfigurations, the absolute value of the hysteresis width (`ABS(HYSTERESIS.D1)`) is always used internally.

1. **Switch-on Condition:**

The output switches on (`OUTPUT.D1 := TRUE`) when the input value reaches or exceeds the upper limit of the hysteresis band:

$$\text{INPUT.D1} \geq \text{THRESHOLD.D1} + \frac{|\text{HYSTERESIS.D1}|}{2.0} \tag{2.131}$$

1. **Switch-off Condition:**

The output switches off (`OUTPUT.D1 := FALSE`) when the input value falls well below the lower limit of the hysteresis band:

$$\text{INPUT.D1} < \text{THRESHOLD.D1} - \frac{|\text{HYSTERESIS.D1}|}{2.0} \tag{2.132}$$

Technical Features

- **Oscillation Prevention:** By using a A strict inequality ($<$) when switching off and a soft inequality ($\geq$) when switching on guarantees stable behavior at the exact limits of the hysteresis band.
- **Absolute Hysteresis:** By using the `ABS` function in the Execution Control Chart (ECC), the function block also functions correctly if a negative value is incorrectly passed for the hysteresis width.
- **Adapter-Based Architecture:** The use of unidirectional adapters (`AR` / `AX`) consolidates data and event lines. This significantly improves clarity in the application diagram.

State Overview

The behavior is controlled via the following states in the Execution Control Chart (ECC):

1. **START**: Idle state before initialization.
2. **Init**: Resets output `OUTPUT.D1` and sets `QO := QI`.
3. **sOFF**: State of the off output. The algorithm `alOff` ensures that `OUTPUT.D1` is set to `FALSE`. A transition to `sON` occurs as soon as the on condition is met.
4. **sON**: State of the on output. The algorithm `alOn` sets `OUTPUT.D1` to `TRUE` (if `QI` is active). A transition back to `sOFF` occurs as soon as the shutdown condition is met.
5. **DeInit**: Deinitialization state. Resets the outputs in a controlled manner.

Application Scenarios

- **Two-Point Temperature Control:** Controls a heating system. The heating system switches on when the temperature falls below the setpoint minus half the hysteresis and switches off again only when the temperature exceeds the setpoint plus half the hysteresis.
- **Level Control:** Switches pumps in tanks on and off. Prevents the pumps from switching continuously due to turbulent liquid surfaces (wave motion).
- **Limit Monitoring in Process Areas:** Converts noisy analog sensor values (e.g., pressure, flow rate) into stable digital alarm signals.

2.10 · Comparison with Similar Components

Unlike classic comparator components (such as `GT` or `LT`), which react immediately to even the slightest threshold violation, the `Hysteresis_AR_AX` buffers the signal through the defined band.

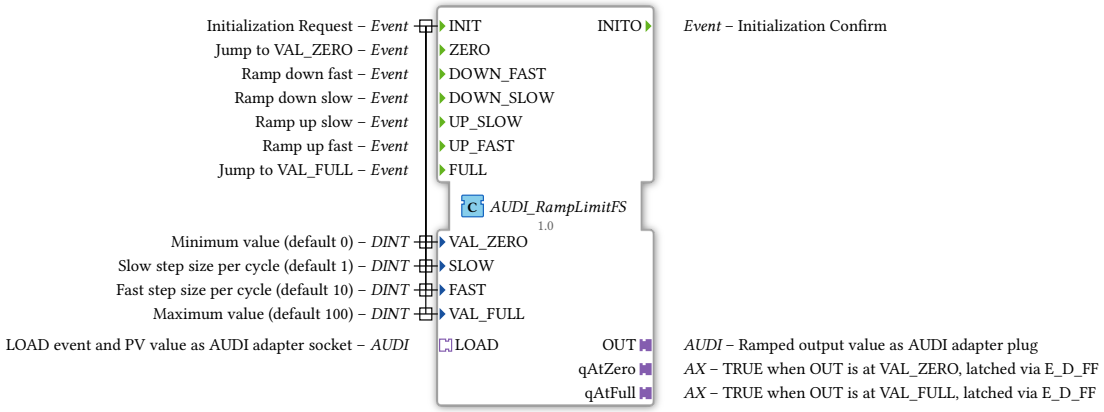
Compared to standard hysteresis circuits, which often use separate pins for data and trigger events, this component significantly simplifies signal routing in more complex 4diac applications thanks to its standardized adapter concept (`AR`/`AX`).

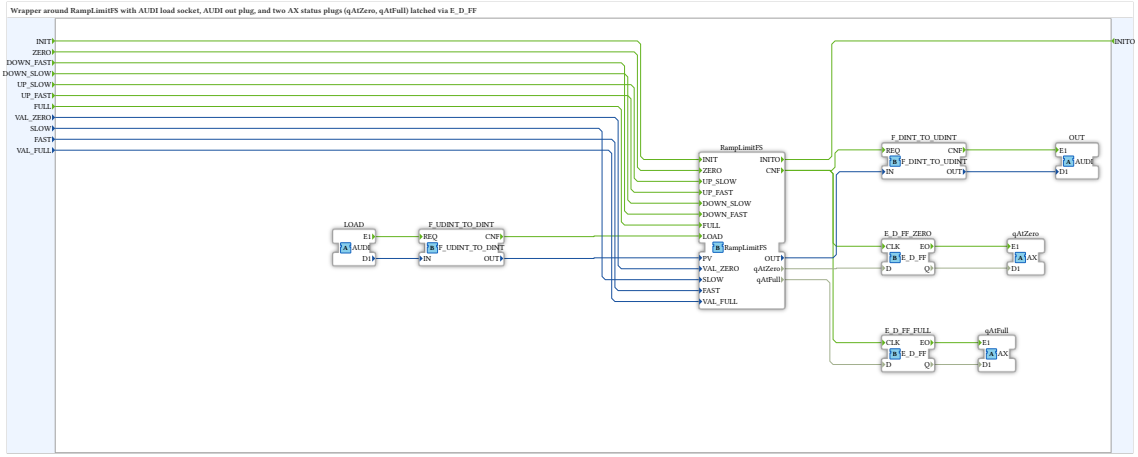
Conclusion

The `Hysteresis_AR_AX` is a highly efficient and robust component for signal preprocessing. Thanks to its integrated absolute hysteresis calculation and asymmetric switching conditions, it offers high reliability in practical operation. The consistent use of adapters allows it to integrate seamlessly into modern, service-oriented control architectures.

Ramp

AUDI_RampLimitFS





• – ○ • – ○ • – ○ •

Introduction

The **AUDI_RampLimitFS** function block is a composite wrapper designed to integrate the ramp functionality of the `RampLimitFS` FB with an AUDI adapter interface. It provides a seamless connection between unidirectional AUDI sockets/plugs and the internal ramp generator, while also exposing latched status flags (`qAtZero` and `qAtFull`) through AX adapters. The FB accepts a setpoint via a `LOAD` socket, applies ramp-up/down or jump commands, and outputs the ramped value via the `OUT` plug. The two status plugs indicate whether the current output has reached the defined minimum or maximum value, with the latched behavior ensuring the flags persist until the next corresponding event.

Interface Structure

Event Inputs

Event	Type	Comment
`INIT`	EInit	Initialization request, sets internal parameters and initializes the ramp generator.
`ZERO`	Event	Jump to `VAL_ZERO` (minimum value).
`DOWN_FAST`	Event	Ramp down at fast speed.
`DOWN_SLOW`	Event	Ramp down at slow speed.
`UP_SLOW`	Event	Ramp up at slow speed.
`UP_FAST`	Event	Ramp up at fast speed.
`FULL`	Event	Jump to `VAL_FULL` (maximum value).

Event Outputs

Event	Type	Comment
`INITO`	EInit	Initialization confirm signal.

Data Inputs

Name	Type	Initial Value	Comment
`VAL_ZERO`	DINT	0	Minimum value of the ramp range.
`SLOW`	DINT	1	Step size for slow ramp operations.
`FAST`	DINT	10	Step size for fast ramp operations.

VAL_FULL	DINT	100	Maximum value of the ramp range.

Data Outputs

There are no direct data outputs; all data is communicated through adapters.

Adapters

Name	Type	Direction	Comment
LOAD	adapter::types::unidirectional::AUDI	Socket	Receives a load request (E1) and a process value (D1, type UDINT) to be used as the ramp setpoint.
OUT	adapter::types::unidirectional::AUDI	Plug	Delivers the ramped output value (D1, type UDINT) and an output event (E1).
qAtZero	adapter::types::unidirectional::AX	Plug	Status flag: TRUE when the output equals VAL_ZERO; latched via an internal E_D_FF.
qAtFull	adapter::types::unidirectional::AX	Plug	Status flag: TRUE when the output equals VAL_FULL; latched via an internal E_D_FF.

Functionality

The AUDI_RampLimitFS block encapsulates a RampLimitFS FB and provides a convenient adapter-based interface. Internally, the following data flow is implemented:

1. **Load Request**: An event on the LOAD socket (E1) triggers a conversion of the incoming UDINT value (D1) to DINT using F_UDINT_TO_DINT. The converted value is passed as the setpoint (PV) to the RampLimitFS FB.
2. **Ramp Control**: The events ZERO, DOWN_FAST, DOWN_SLOW, UP_SLOW, UP_FAST, and FULL are directly forwarded to the corresponding event inputs of the RampLimitFS FB. These events control how the output moves towards the setpoint:
   - ZERO causes an immediate jump to VAL_ZERO.
   - FULL causes an immediate jump to VAL_FULL.
   - The ramp events (DOWN_*, UP_*) cause the output to move stepwise in the specified direction and speed.
3. **Output Generation**: When the RampLimitFS completes a step or a jump (indicated by its CNF event), the current DINT output is converted back to UDINT via F_DINT_TO_UDINT and delivered on the OUT plug along with an E1 event.
4. **Status Latching**: The RampLimitFS provides boolean flags qAtZero and qAtFull indicating whether the current output is exactly at VAL_ZERO or VAL_FULL. These flags are latched using two E_D_FF (D-type flip-flop) FBs, one per status. On each CNF event, the current flag value is sampled and stored; the latched value is then propagated to the corresponding AX plug (qAtZero.D1 or qAtFull.D1) along with an event E1.
5. **Initialization**: The INIT event transfers the parameters (VAL_ZERO, SLOW, FAST, VAL_FULL) to the internal RampLimitFS and initiates its own initialization. Upon completion, INITO is emitted.

Technical Features

- **Adapter Integration**: Combines AUDI and AX adapter types for a clean, IEC 61499-compliant interface.
- **Type Conversion**: Automatically converts between UDINT (adapter data) and DINT (internal ramp calculations) using standard conversion FBs.
- **Latched Status**: The qAtZero and qAtFull flags are latched via edge-triggered D flip-flops, ensuring that the status remains stable until the next output change.

- **Configurable Ramp Limits**: The minimum (`VAL_ZERO`) and maximum (`VAL_FULL`) values are user-definable, as are the step sizes for slow and fast ramping.
- **Composite Architecture**: Built entirely from standard FBs (`RampLimitFS`, conversion FBs, and E_D_FF), making it easy to inspect and modify.

State Overview

The behavior of the internal `RampLimitFS` can be described in terms of operational states, which are reflected in the events and the status flags:

- **Idle / Initialized**: After `INIT`, the output is typically set to `VAL_ZERO` (or the last value). No ramp operations are active.
- **Jump to Zero / Full**: When `ZERO` or `FULL` is issued, the output directly moves to the respective limit. The corresponding status flag (`qAtZero` or `qAtFull`) is set and latched.
- **Ramping**: During `UP_*` or `DOWN_*` operations, the output changes stepwise by `SLOW` or `FAST` increments/decrements per cycle. The process continues until the setpoint is reached or a different command is issued.
- **Limit Reached**: When the output reaches either limit, the associated status flag becomes active and remains latched until a new command moves the output away.

Application Scenarios

- **Motion Control**: Adjusting a motor speed or position with configurable acceleration/deceleration rates, using fast/slow ramps to avoid mechanical stress.
- **Process Automation**: Smoothly transitioning a process variable (e.g., temperature, flow) between setpoints, with automatic clamping to safety minimum/maximum values.
- **HMI Integration**: The AUDI adapter format is ideal for connecting to visualization or supervisory systems that exchange UDINT data, while the AX status plugs provide simple boolean indication for limit states.

Comparison with Similar Blocks

- **vs. Standard `RampLimitFS`**: The `AUDI_RampLimitFS` adds an adapter layer and latched status outputs. It simplifies integration by hiding the conversion steps and provides a more standardized interface for industrial applications.
- **vs. Simple Ramp FBs without status**: Many basic ramp blocks only output the ramped value; this FB additionally provides explicit, latched limit indicators, which can be used for alarm or interlock logic.
- **vs. PID or other closed-loop controllers**: Unlike controllers, this block does not perform feedback regulation; it only generates a time-based ramp signal towards a given setpoint.

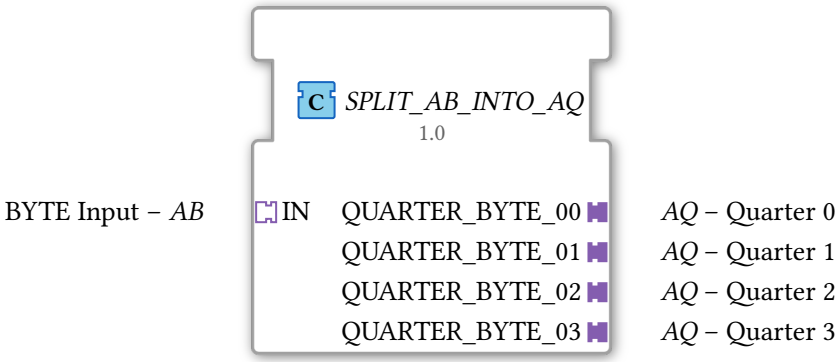
Conclusion

AUDI_RampLimitFS is a robust, adaptable function block that encapsulates a proven ramp generator with a modern adapter-based interface. Its use of AUDI and AX adapters promotes interoperability, while the internal type conversions and latched status flags reduce the effort required for integration and monitoring. The block is suitable for a wide range of applications where controlled, limit-bounded transitions of a numeric output are needed.

2.10.0.1 Numeric Index (adapter)

Here you will find all exercises from the `adapter` workspace, listed in ascending order. …

SPLIT_AB_INTO_AQ



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_AB_INTO_AQ` is used to split a data byte received via a `AB` adapter (byte) into four individual 2-bit quartets and output each of these via a `AQ` adapter (quarter). It encapsulates the decomposition of a byte into manageable partial values and makes them available for further processing via standardized unidirectional adapter interfaces.

Interface Structure

Event Inputs

Name	Description
`IN.E1`	An event at the socket triggers the decomposition and output of the quartets.

Event Outputs

No direct event outputs; output is provided via the events of the connected adapters (`QUARTER_BYTE_xx.E1`).

Data Inputs

Name	Description
`IN.D1`	The incoming data byte (type: BYTE) that is split into four quartets.

Data Outputs

No direct data outputs; the quartets are provided via the data interfaces of the output adapters (`QUARTER_BYTE_xx.D1`).

Adapters

Name	Type	Direction	Description
`IN`	`adapter::types::unidirectional::AB`	Socket (Input)	Receives the byte to be parsed, including the event.
`QUARTER_BYTE_00`	`adapter::types::unidirectional::AQ`	Plug (Output)	Outputs the quartet 0 (least significant 2 bits).
`QUARTER_BYTE_01`	`adapter::types::unidirectional::AQ`	Plug (Output)	Outputs the quartet 1.
`QUARTER_BYTE_02`	`adapter::types::unidirectional::AQ`	Plug (Output)	Outputs quartet 2.
`QUARTER_BYTE_03`	`adapter::types::unidirectional::AQ`	Plug (Output)	Outputs quartet 3 (most significant 2 bits).

Functionality

1. An event at socket `IN.E1` activates processing.
2. The byte present at the socket (`IN.D1`) is forwarded to the internal component `SPLIT_BYTE_INTO_QUARTERS`. This component splits the byte into four 2-bit parts (quartets).
3. The four quartets are fed in parallel to the data inputs of `E_D_FF_ANY` flip-flops.
4. Simultaneously, the splitter's acknowledgment event (`CNF`) is distributed to the clock inputs (`CLK`) of all four flip-flops.
5. On the rising edge of this event, the flip-flops take over the quartets and pass them on to their respective outputs (`Q`).
6. The flip-flop outputs are directly connected to the data inputs of the output adapters `QUARTER_BYTE_00` … `QUARTER_BYTE_03`. Simultaneously, the flip-flop output event (`EO`) is sent to the event input (`E1`) of the respective adapter, so that the data is output on the adapter interface.

Thus, with each activation of the socket, an entire byte is split into four quartets and transmitted synchronously to all four output adapters.

Technical Features

- **Internal Flip-Flops:** The use of `E_D_FF_ANY` ensures that the splitting occurs only once per event and that the values are buffered until a new event arrives.
- **Adapter Coupling:** Instead of direct inputs/outputs, only unidirectional adapters are used, enabling loose coupling and reusability in complex adapter networks.
- **Synchronization:** The flip-flops are clocked via the shared `CNF` event, ensuring all quartets are present simultaneously.
- **No State Machine:** The function block operates purely event-driven without its own sequential state – the logic is entirely implemented through the interconnected internal components.

State Overview

Since `SPLIT_AB_INTO_AQ` is a composite function block without its own state machine, its behavior is determined by the internal components. Essentially, there are two phases:

1. **Idle:** No event is present at the socket; the output adapters hold the last received values.
2. **Processing:** An event at the socket triggers the decomposition and updating of all outputs. Upon completion, the function block returns to its idle state.

Application Scenarios

- **Control of Four Actuators:** A byte (e.g., from a master control bus) is split into four 2-bit values to provide separate control signals for independent actuators.
- **Data Reduction:** Splitting a byte into four individual signals for transmission to parallel, simpler processing units.
- **Adapter-Based Architectures:** Used in systems that consistently operate with unidirectional adapters to increase modularity.

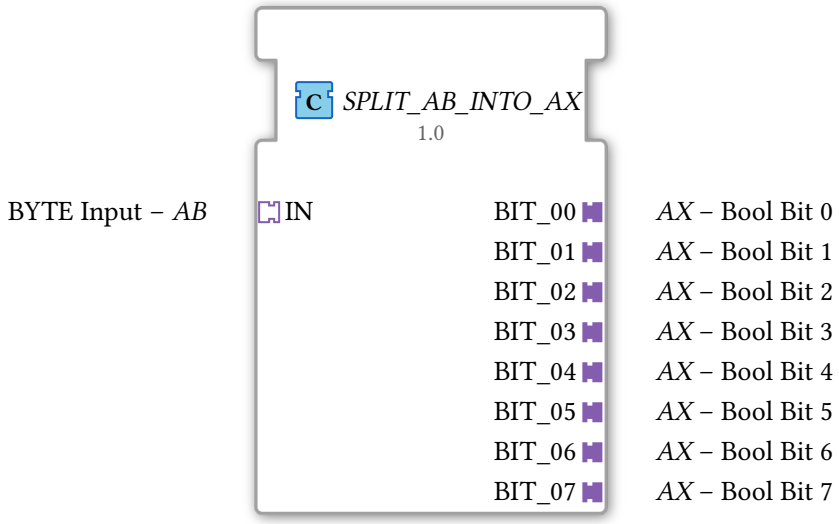
Comparison with Similar Components

- **`SPLIT_BYTE_INTO_QUARTERS`:** Performs the simple splitting of a byte into four quartets, but via direct data and event inputs/outputs. `SPLIT_AB_INTO_AQ` encapsulates this logic in an adapter interface and adds flip-flops for stable output.
- **`SPLIT_WORD_INTO_BYTES`:** Splits a 16-bit word into two bytes – analogous principle, but for different data widths and adapter types.
- **Simple Multiplexers/Demultiplexers:** While these signals switch, `SPLIT_AB_INTO_AQ` focuses on the fixed, parallel division of a composite data type.

Conclusion

SPLIT_AB_INTO_AQ` is a specialized composite function block for splitting a byte adapter input into four separate quarter adapter outputs. It simplifies the design of adapter networks by hiding the complex bit manipulation and ensuring reliable, event-driven output of the quartets. The use of flip-flops ensures stable data storage between events. This component is particularly suitable for modular automation solutions that work with unidirectional adapters.

SPLIT_AB_INTO_AX



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SPLIT_AB_INTO_AX** splits a byte received via a unidirectional AB adapter into its eight individual bits and provides these via separate AX adapters. The bits are transferred using clock-controlled D flip-flops, ensuring that the output values remain stable until a new byte is processed.

Interface Structure

Event Inputs

The function block does not have explicit top-level event inputs. Processing is triggered by an event arriving via the socket adapter **IN** (type AB). The **IN** adapter provides one event input (`E1`) and one data input (`D1`).

Event Outputs

This function block does not have explicit event outputs at the top level. Results are output via the plug adapters **BIT_00** to **BIT_07** (type AX), which also contain an event output (`E1`).

Data Inputs

- **IN.D1** – The byte to be processed is received via the socket adapter **IN** (type AB).

Data Outputs

- **BIT_00.D1 ... BIT_07.D1** – The individual Boolean values of the bits (bit 0 to bit 7) are output via the plug adapters **BIT_00** to **BIT_07** (type AX).

Adapters

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AB	Socket (Input)	BYTE value, decomposed into individual bits
BIT_00	adapter::types::unidirectional::AX	Plug (Output)	Boolean value of bit 0 (LSB)
BIT_01	adapter::types::unidirectional::AX	Plug (Output)	Boolean value of bit 1
BIT_02	adapter::types::unidirectional::AX	Plug (Output)	Boolean value of bit 2
BIT_03	adapter::types::unidirectional::AX	Plug (Output)	Boolean value of bit 3
BIT_04	adapter::types::unidirectional::AX	Plug (Output)	Boolean value of bit 4

BIT_05	adapter::types::unidirectional::AX	Plug (Output)	Boolean value of bit 5
BIT_06	adapter::types::unidirectional::AX	Plug (Output)	Boolean value of bit 6
BIT_07	adapter::types::unidirectional::AX	Plug (Output)	Boolean value of bit 7 (MSB)

Functionality

1. An incoming byte is received via the socket adapter **IN**. The event `E1` triggers the processing.
2. The internal function block `SPLIT_BYTE_INTO_BOOLS` splits the byte into eight Boolean signals (`BIT_00` ... `BIT_07`). Simultaneously, the event `CNF` is generated.
3. This event is distributed to the clock inputs (`CLK`) of eight **E_D_FF** flip-flops. Each flip-flop receives the corresponding Boolean value at its data input `D` at the clock signal.
4. The outputs `Q` of the flip-flops provide the stabilized bits. These are passed via the data connections to the plug adapters **BIT_00** ... **BIT_07**.
5. The event `EO` of each flip-flop is placed on the event output of the respective plug adapter, so that downstream components are notified of the new bit information.

Technical Features

- **State Storage**: The use of D flip-flops (E_D_FF) ensures that the output values are retained even after the processing event until a new byte is processed.
- **Parallel Output**: All eight bits are updated simultaneously as soon as the jointly clocked event `CNF` of `SPLIT_BYTE_INTO_BOOLS` occurs.
- **Unidirectional Adapters**: Both the input and output adapters are unidirectional (from the application to the resource), therefore the component is suitable for transmission directions in which only one side controls the data flow.

State Overview

The component itself does not have an explicit state machine. Its behavior results from the internal combination of:

- **SPLIT_BYTE_INTO_BOOLS**: Single-step – generates eight BOOL values and an acknowledgment event from one byte.
- **E_D_FF**: Each flip-flop stores a Boolean value on the rising clock edge and holds it until the next clock cycle.

Thus, the component can be described as a **pure combinational splitter with downstream hold elements**.

Application Scenarios

- **Digital Output Control**: A byte (e.g., from a communication adapter) is split into individual digital outputs, each connected to actuators via AX getter adapters.
- **State Monitoring**: A byte signal (e.g., from a fieldbus) is decomposed into individual status bits, which can be processed or visualized separately.
- **Bit-level data preparation**: Before passing the data to internal processing that expects individual Boolean signals.

Comparison with similar function blocks

- **SPLIT_BYTE_INTO_BOOLS**: Also splits a byte into Boolean values, but outputs them directly as data outputs and generates a single event.

SPLIT_AB_INTO_AX extends this with clock-controlled buffering and output via AX adapters, allowing for clean event synchronization with subsequent function blocks.

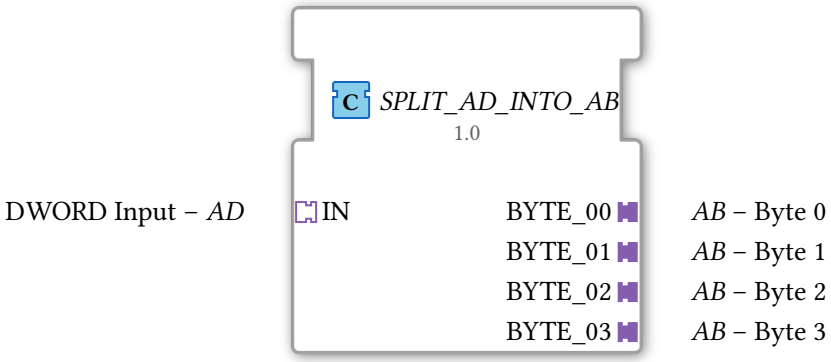
- **Simple data array splitters**: Some libraries offer function blocks that split arrays into individual elements, but without the additional storage and adapter interface.

Conclusion

SPLIT_AB_INTO_AX is a specialized function block for splitting a byte value into eight stable Boolean outputs via AX adapters. The combination of splitter and flip-flops makes it particularly suitable for scenarios where individual bits need to be accessed asynchronously or with a time delay, without the

value changing during processing. Its clear, adapter-based interface facilitates integration into modular IEC 61499 applications.

SPLIT_AD_INTO_AB



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SPLIT_AD_INTO_AB** is used to split a 32-bit data word (AD adapter) into four individual bytes. The four bytes are output via separate unidirectional AB adapters. The block is implemented as a composite function block and uses a SPLIT_DWORD_INTO_BYTES and several edge-triggered D flip-flops (E_D_FF_ANY).

Interface Structure

The function block does not have any traditional event or data interfaces. All communication takes place via adapters:

Adapters (Sockets)

Adapter	Type	Comment
IN	`adapter::types::unidirectional::AD`	DWORD Input (32-bit)

Adapter (Plugs)

Adapter	Type	Comment
BYTE_00	`adapter::types::unidirectional::AB`	Byte 0 (least significant byte)
BYTE_01	`adapter::types::unidirectional::AB`	Byte 1
BYTE_02	`adapter::types::unidirectional::AB`	Byte 2
BYTE_03	`adapter::types::unidirectional::AB`	Byte 3 (most significant byte)

Each AB adapter has at least one event output (E1) and one data output (D1) connected to the internal flip-flops.

Functionality

1. An event at the **IN** adapter (E1) triggers the internal component `SPLIT_DWORD_INTO_BYTES`.
2. `SPLIT_DWORD_INTO_BYTES` parses the DWORD input at **IN.D1** into four individual bytes (BYTE_00 to BYTE_03) and outputs them as data.
3. Simultaneously, `SPLIT_DWORD_INTO_BYTES` generates an acknowledgment event (CNF), which is forwarded to the clock inputs (CLK) of all four `E_D_FF_ANY` flip-flops.
4. Each flip-flop stores its assigned byte (D input) on the rising edge of the clock and outputs it at its output (Q).
5. The flip-flop outputs are connected to the data outputs of the AB adapters (D1). Simultaneously, the event (EO) of each flip-flop is sent to the event input (E1) of the corresponding adapter, so that the connected consumer is informed of the new data.

Technical Features

- **Synchronous Output**: All four bytes are output simultaneously with the same event (CNF of `SPLIT_DWORD_INTO_BYTES`). This ensures that the consumers of the individual bytes remain synchronized.
- **Use of E_D_FF_ANY**: These edge-triggered flip-flops ensure that the data is only transferred upon a new event and remains stable at the output until then.
- **Adapter-Based Interface**: The function block operates entirely via adapters, allowing it to integrate seamlessly into an adapter-oriented architecture (e.g., when using `adapter::types::unidirectional::AB` and `AD`).

State Overview

The function block itself does not have an explicit state machine – it is a composite of standard function blocks. Internal flow control is managed through event and data flows:

- **Wait**: No input event ⟶ output remains unchanged.
- **Split**: Upon an event at the input adapter, the DWORD is split once, and all four bytes are output.

Application Scenarios

- **Decomposition of 32-bit sensor data** (e.g., encoder values, A/D converters) into individual bytes for transmission via byte-oriented protocols.
- **Control of peripherals** that expect individual bytes as address or data words (e.g., register accesses).
- **Data collection in distributed systems** where a 32-bit value must be distributed among multiple independent processors.

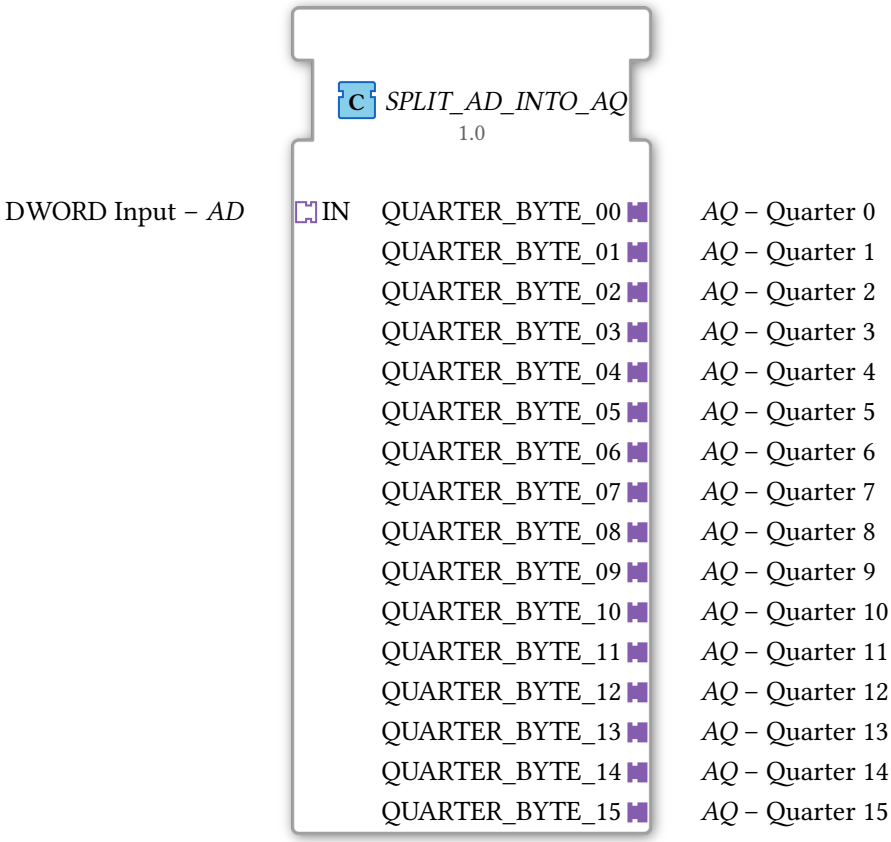
Comparison with similar components

- **SPLIT_DWORD_INTO_BYTES**: Also splits a DWORD into bytes, but outputs them directly as data outputs (BYTE_00..BYTE_03) without routing the data through adapters.
- **SPLIT_AD_INTO_AB**: Offers the same functionality, but encapsulates the output in unidirectional AB adapters. This allows for a clean separation of interfaces and facilitates reuse in different contexts.
- **AT** (Adapter-Type-Based Function Blocks): Other adapter splitters might, for example, split WORDs into nibbles, while this function block is specifically designed for DWORD ⟶ 4×BYTE.

Conclusion

The **SPLIT_AD_INTO_AB** is a useful composite function block that synchronously splits a 32-bit data value into four individual bytes using an adapter. Edge-triggered flip-flops ensure stable outputs until the next input event arrives. The adapter interfaces integrate seamlessly into modular real-time systems according to IEC 61499 and facilitate structured data transfer.

SPLIT_AD_INTO_AQ



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_AD_INTO_AQ` splits an incoming AD adapter (DWORD) into 16 individual AQ adapters (QUARTER). It serves as an interface to divide a wide data value (32 bits) into its 2-bit quarter components and forward these to separate output adapters. The block is implemented as a composite function block and internally uses a `SPLIT_DWORD_INTO_QUARTERS` block and 16 flip-flops (`E_D_FF_ANY`) for synchronous transmission.

Interface Structure

Event Inputs

The function block does not have its own independent event inputs. The event is received via the adapter interface `IN.E1` and processed internally.

Event Outputs

The function block (FB) does not have its own event outputs. Event output is provided via the AQ adapter plugs (each via `QUARTER_BYTE_xx.E1`).

Data Inputs

The FB does not have its own data inputs. Data is provided via the adapter interface `IN.D1`.

Data Outputs

The FB does not have its own data outputs. The distributed data is output via the AQ adapter plugs (each via `QUARTER_BYTE_xx.D1`).

Adapters

Name	Type	Direction	Description

IN	adapter::types::unidirectional::AD	Socket (Input)	DWORD input adapter (32 bits). Events and data are received via E1 and D1.
QUARTER_BYTE_00 ... QUARTER_BYTE_15	adapter::types::unidirectional::AQ	Plug (Output)	16 output adapters, each providing one quarter (2 bits) of the original DWORD. Each adapter has an event output E1 and a data output D1.

Functionality

The module operates entirely internally using a composite network. As soon as an event (E1) arrives via the IN adapter, it is forwarded to the internal module SPLIT_DWORD_INTO_QUARTERS. This module then parses the stored DWORD value (IN.D1) into 16 separate quarter bytes (each 2 bits).

After the division is complete, SPLIT_DWORD_INTO_QUARTERS.CNF signals a clock pulse, which is simultaneously sent to all 16 flip-flops (E_D_FF_ANY_00 to E_D_FF_ANY_15). The flip-flops then receive the respective quarter data at their data input (from SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_xx) and output it at their output Q. Simultaneously, they fire an event at their output EO, which is forwarded to the corresponding AQ adapter plug (QUARTER_BYTE_xx.E1). This transfers the output data value (Q) to the target device via the adapter data path (D1).

The flip-flops then receive the respective quarter data at their data input (SPLIT_DWORD_INTO_QUARTERS.QUARTER_BYTE_xx) and output it at their output Q. Processing is strictly synchronous: all 16 quarter values are updated at the same clock cycle.

Technical Features

- **Composite Architecture**: The function block (FB) uses other components (SPLIT_DWORD_INTO_QUARTERS and E_D_FF_ANY) to implement the decomposition and synchronization.
- **No Dedicated Inputs/Outputs**: All communication occurs exclusively via adapters. This enables clean encapsulation and reuse in adapter-based architectures.
- **Synchronization**: All 16 AQ outputs are updated simultaneously by a common clock (from SPLIT_DWORD_INTO_QUARTERS.CNF). This ensures that the split data is consistently available at the same time.
- **Scalability**: The FB is designed for 16 quarter values (corresponding to 32 bits). Adaptation to other bit widths would be possible by modifying the internal structure.

No Dedicated Inputs/Outputs:

State Overview

As a composite function block (FB), SPLIT_AD_INTO_AQ does not have its own state machine. Internal data processing is determined by:

- **Waiting for Event**: No internal event is generated during idle time.
- **Processing**: Upon arrival of IN.E1, the flip-flops are split and updated.
- **Output**: All AQ outputs are available immediately after the clock signal.

Application Scenarios

- **Data Splitting in Control Systems**: When a sensor or communication module provides a 32-bit data word (e.g., encoder position, measured value) that needs to be split into 16 separate 2-bit signals (e.g., brightness sensors, switching states).
- **Adapter-Based Communication**: Used in the 4diac IDE when adapters according to IEC 61499 are used to distribute data between different components.
- **Parallel processing of partial information**: Decomposition of a DWORD for subsequent components, each requiring only 2 bits of the original.

Comparison with similar components

Component	Description	Difference to SPLIT_AD_INTO_AQ

`SPLIT_DWORD_INTO_QUARTERS`	Decomposes a DWORD into 16 quarter values and outputs them as direct data outputs.	`SPLIT_AD_INTO_AQ` additionally encapsulates this decomposition in adapter interfaces and adds flip-flop synchronization.
`SPLIT_INT_INTO_BITS`	Splits an integer into individual bits.	Operates at the bit level and not on 2-bit quarters; Output is typically in Boolean form.
Manual partitioning with `MUX` or `DEMUX`	Could be used to implement data partitioning without adapters.	`SPLIT_AD_INTO_AQ` is specifically optimized for adapter communication and offers a bundled, synchronized solution.

Conclusion

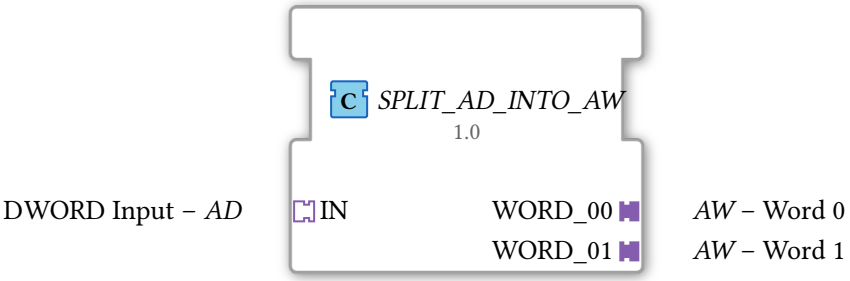
SPLIT_AD_INTO_AQ` is a useful building block for partitioning a DWORD (AD adapter) into 16 2-bit quarter-intercept adapters (AQ). Its composite architecture with internal synchronization ensures consistent data transmission and facilitates modular, adapter-based programming in the 4diac IDE. It is particularly suitable for applications that require parallel processing of partial information without the partitioning details needing to be visible in the higher-level network.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

SPLIT_AD_INTO_AW



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SPLIT_AD_INTO_AW** is used to split a 32-bit DWORD value, received via a **AD** adapter (unidirectional), into two 16-bit WORD values and output them via two separate **AW** adapters (unidirectional). The splitting is event-driven, and the output data is stabilized by flip-flops until a new trigger arrives. The function block is implemented as a composite function block and internally uses the **SPLIT_DWORD_INTO_WORDS** function block as well as two **E_D_FF_ANY** flip-flops.

Interface Structure

Event Inputs

Name	Adapter	Event	Description
IN	AD (Socket)	E1	Starts the partitioning of the incoming DWORD value.

Event Outputs

Name	Adapter	Event	Description
WORD_00	AW (Plug)	E1	Signals that the lower-order WORD (low word) is valid.

WORD_01	AW (Plug)	E1	Indicates that the higher-order WORD (High Word) is valid.

Data Inputs

Name	Adapter	Data Type	Description
IN	AD (Socket)	D1 (DWORD)	32-bit input value that is split.

Data Outputs

Name	Adapter	Data Type	Description
WORD_00	AW (Plug)	D1 (WORD)	Low-order 16-bit word of the input DWORD.
WORD_01	AW (Plug)	D1 (WORD)	High-order 16-bit word of the input DWORD.

Adapters

The adapters are designed as **unidirectional**:

- **AD** (Input): Provided via a socket, containing one event (E1) and one data (D1) channel.
- **AW** (Output): Provided via two plugs, each with one event (E1) and one data (D1) channel.

Functionality

1. An event at input **IN.E1** triggers processing.
2. The current DWORD value of **IN.D1** is read and passed to the internal function block **SPLIT_DWORD_INTO_WORDS**.
3. This block splits the 32-bit value into two 16-bit words: **WORD_00** (lower order) and **WORD_01** (higher order).
4. The two results are transferred in parallel to the flip-flops **E_D_FF_ANY_00** and **E_D_FF_ANY_01** and held there until the next event.
5. Simultaneously, the output events **WORD_00.E1** and **WORD_01.E1** are triggered, indicating that the corresponding data (**WORD_00.D1** and **WORD_01.D1**) are valid.

...

Technical Features

- **Data Synchronization:** The flip-flops ensure that the output data remains stable, even if the input value changes between two events. Only a new event at the input updates both outputs.
- **Simultaneous Output:** Both WORD outputs are updated by the same input event – there is no staggered timing.
- **Adapter Compatibility:** The function block is specifically designed for use with **AD-** and **AW-**adapters and allows seamless integration into adapter-based data flows.

State Overview

The function block does not have an explicit state machine. Its behavior is purely event-driven:

- **Idle:** Waiting for an event at **IN.E1**.
- **Processing:** Upon receipt of **IN.E1**, the data is split and the flip-flops are set.
- **Output:** The events at the outputs are triggered exactly once after processing.

Application Scenarios

- **Adapter-Based Decomposition:** Splitting a 32-bit value (e.g., from a sensor data word) into two 16-bit parts for separate processing chains.
- **Communication Protocols:** Preparing data for interfaces that transmit WORD-wise (e.g., Modbus registers or CAN messages).
- **Bit Manipulation:** Preprocessing for subsequent components that only expect WORD adapters.

Comparison with Similar Components

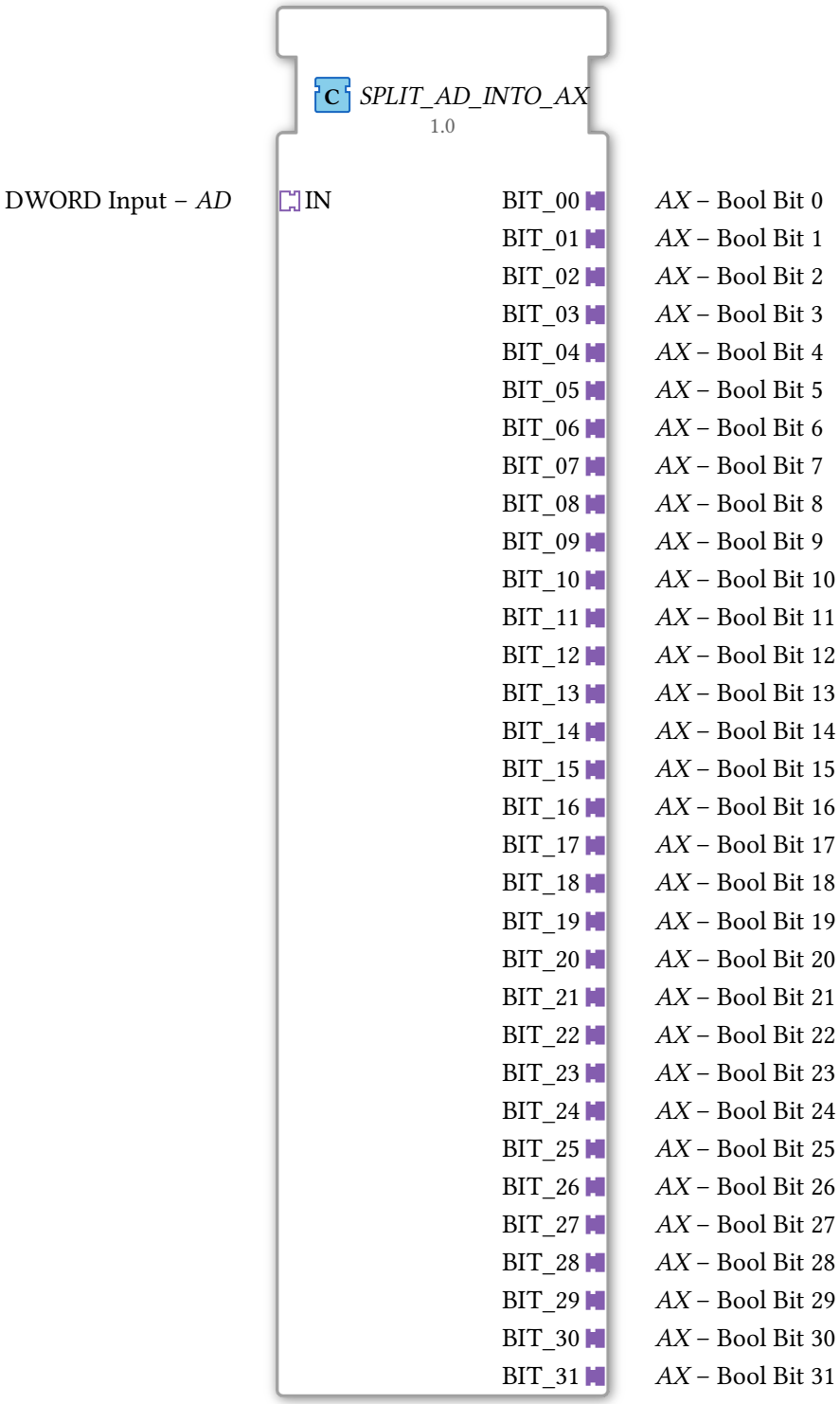
Component	Description	Difference

SPLIT_DWORD_INTO_WORDS	Pure data splitting without event output or adapters.	Provides only data outputs, no events, and no adapter connectivity.
SPLIT_AD_INTO_AW	Adapter-based variant with stabilized outputs.	Integrates flip-flops and event output, specifically for AD/AW adapters.

Conclusion

SPLIT_AD_INTO_AW is a practical, combined function block that simplifies the splitting of a DWORD adapter signal into two WORD adapter signals. Thanks to its built-in flip-flops and synchronized event output, it is particularly well-suited for time-critical and adapter-oriented applications in industrial control systems.

SPLIT_AD_INTO_AX



• – ◦ • – ◦ • – ◦ •

Introduction

The SPLIT_AD_INTO_AX function block splits a 32-bit data word received via an AD adapter (DWORD) into 32 separate AX adapters (BOOL). Each AX adapter represents a single bit of the original DWORD value. The block encapsulates the necessary event control and data storage in a modular, easy-to-use function block.

Interface Structure

Event Inputs

None. Event control is handled indirectly via the adapter socket.

Event Outputs

None. Output is provided via the adapter plugs, whose internal event behavior is defined by the types used (AX).

Data Inputs

Label	Type	Comment
IN	AD	DWORD Input (32-bit)

Data is received via the IN adapter socket. The AD adapter provides a DWORD value and an associated event (E1).

Data Outputs

Label	Type	Comment
BIT_00	AX	Bool Bit 0
BIT_01	AX	Bool Bit 1
...	...	...
BIT_31	AX	Bool Bit 31

All 32 output adapters are of type AX (unidirectional BOOL adapter). Each output returns the state of the corresponding bit of the incoming DWORD.

Adapters

- **Socket**: `IN` (type `adapter::types::unidirectional::AD`) – receives the DWORD value and its associated event.
- **Plugs**: 32 adapters `BIT_00` ... `BIT_31` (type `adapter::types::unidirectional::AX`) – provide the individual bits as Boolean signals to the connected logic.

Functionality

The internal process of the function block (FB) is divided into two steps:

1. **Division of the DWORD into Boolean Values**

An internal FB `SPLIT_DWORD_INTO_BOOLS` receives the DWORD via the data connection `IN.D1`. It divides the 32 bits into individual Boolean signals (`BIT_00` ... `BIT_31`).

1. **Synchronization and Storage**

The event `IN.E1` triggers the input `REQ` of the splitter. After complete processing, the splitter sends the acknowledgment event `CNF`. This event is forwarded to the clock inputs (CLK) of 32 flip-flops (type `E_D_FF`). Simultaneously, the flip-flops receive the Boolean values provided by `SPLIT_DWORD_INTO_BOOLS` on their data inputs `D`.

The outputs `Q` of the flip-flops are permanently connected to the data inputs `D1` of the corresponding AX adapters, so the stored bit values are immediately passed on to the output adapters.

Thus, after each event at the input adapter, the entire 32-bit value is transferred in parallel to the 32 output adapters and held there until the next event.

Technical Features

- **Parallel Processing**: All 32 bits are processed simultaneously by the flip-flops, ensuring a consistent snapshot of the DWORD.
- **Modular Design**: The function block (FB) utilizes existing standard components (`SPLIT_DWORD_INTO_BOOLS` and `E_D_FF`), facilitating maintainability and reuse.
- **Adapter-Based Communication**: All input and output is handled via adapters, allowing the FB to be seamlessly integrated into adapter-based applications.

- **No External Events**: The FB has no dedicated event inputs/outputs, as event control is handled entirely by the adapters.

State Overview

The FB itself is stateless – its functionality is implemented by the internal flip-flops. Each `E_D_FF` has an internal memory state (0 or 1) that is updated by the clock event. After startup, all flip-flops are in their initial state (0) until the first event arrives at the input.

Application Scenarios

- **Digital Signal Processing**: Decomposition of a 32-bit data word into individual Boolean control signals, e.g., for status or enable bits.
- **Interface Adaptation**: Connecting a DWORD source (e.g., bus system, register) to discrete digital inputs/outputs.
- **Testing and Simulation**: Targeted analysis of individual bits of a data word without complex bit manipulation in the application code.

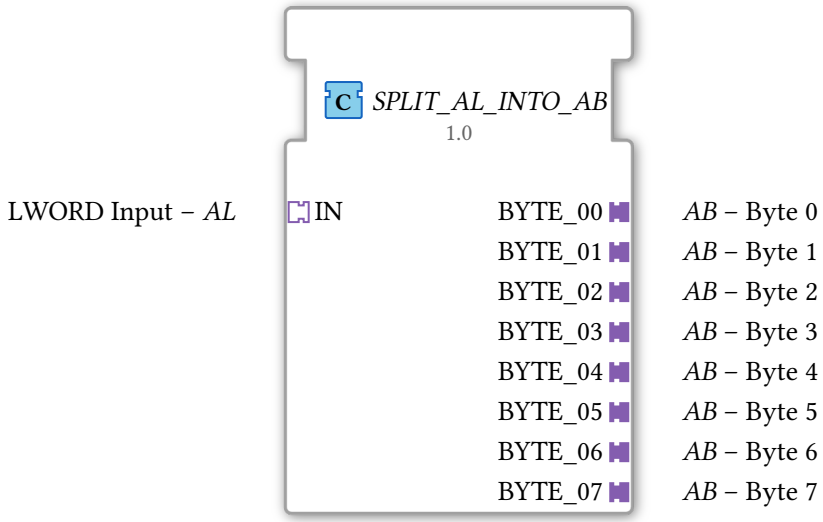
Comparison with Similar Components

- **SPLIT_WORD_INTO_BOOLS / SPLIT_BYTE_INTO_BOOLS**: These components operate at the 16-bit or 8-bit level and are optimized for smaller data word widths. SPLIT_AD_INTO_AX is specifically designed for 32-bit words and uses adapters instead of direct data ports.
- **Direct Bit Extraction with Adapters**: Alternatively, the `SPLIT_DWORD_INTO_BOOLS` could be used directly in an application, and the outputs connected via connectors. However, this function block encapsulates the entire synchronization process and provides a uniform, adapter-based interface, which simplifies reuse.

Conclusion

SPLIT_AD_INTO_AX is a practical function block for splitting a 32-bit data word into 32 individual Boolean adapter signals. Its modular design, parallel processing, and adapter-based communication make it ideally suited for use in complex automation solutions where bit information from a compact data word needs to be resolved into separate logical paths.

SPLIT_AL_INTO_AB



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SPLIT_AL_INTO_AB** is used to split a 64-bit value (LWORD) received via an **AL adapter** (Long Word) into its eight individual bytes and to provide these bytes via eight separate **AB adapters** (Bytes). The splitting occurs synchronously with an event at the input adapter, and each byte is temporarily stored in its own flip-flop (E_D_FF_ANY) before being delivered to the corresponding output adapter. The block is designed as an **adapter splitting component** and is particularly suitable for structured data decomposition in IEC 61499 systems.

Interface Structure

Event Inputs

- **IN** (Socket, Type: `adapter::types::unidirectional::AL`):

The integrated event output **E1** of the AL adapter triggers processing. An incoming event at the IN adapter starts the decomposition of the simultaneously present data value.

Event Outputs

The output adapters **BYTE_00** to **BYTE_07** (Plugs, Type: `adapter::types::unidirectional::AB`) each have an event input **E1**, which is triggered after successful data transfer. These events are generated by the internal network and signal that the corresponding byte is validly present at the adapter's data output.

Event Outputs ### Data Inputs

- **IN** (Socket, Type: `adapter::types::unidirectional::AL`):

Supplies the 64-bit data value to be parsed via the **D1** data input of the adapter (Type LWORD).

Data Outputs

- **BYTE_00** to **BYTE_07** (Plugs, Type: `adapter::types::unidirectional::AB`):

Each of these adapters provides a single byte (BYTE) via its **D1** data output. The mapping is as follows:

- BYTE_00: least significant byte (bits 0…7)
- BYTE_01: next byte (bits 8…15)
- …
- BYTE_07: most significant byte (bits 56…63)

Adapter

- **IN**: Socket (AL – Adapter Long Word) – receiving adapter
- **BYTE_00 … BYTE_07**: Plugs (AB – Adapter Byte) – sending adapters

Functionality

1. An event at the input adapter **IN** (E1) triggers the internal function block **SPLIT_LWORD_INTO_BYTES** (type: `eclipse4diac::utils::splitting::SPLIT_LWORD_INTO_BYTES`).
2. Simultaneously, the LWORD value (D1) provided by the IN adapter is forwarded to the input **IN** of this internal block.
3. The internal splitter divides the LWORD into eight separate bytes (**BYTE_00** … **BYTE_07**) and places them at its data outputs.
4. The splitter's completion event **CNF** is sent in parallel to eight **E_D_FF_ANY** flip-flops. Each flip-flop receives its assigned byte at this clock signal and outputs it at its **Q** output.
5. The flip-flops, in turn, generate an output event **EO**, which is directly connected to the event input **E1** of the corresponding **BYTE_XX** adapter.
6. This causes the individual bytes to be simultaneously output via the eight AB adapters – both as a data value (Q ⟶ D1) and as an acknowledgment event (EO ⟶ E1).

Technical Features

- **Adapter-based interface**: The component uses only adapters (AL and AB) and no conventional event/data ports. This enables modular encapsulation and reuse in adapter-based architectures.
- **Simultaneous Output**: The eight bytes are output simultaneously through the parallel clocking of all flip-flops, which is advantageous for time-critical applications.
- **Cacheloading**: Each byte is cached in its own flip-flop (E_D_FF_ANY), so the output value is retained even if the input value changes or a new decomposition cycle begins.
- **Unidirectional Data Flow**: The adapters are designed as unidirectional (IN: AL, OUT: AB) – data flows only from the input to the outputs.
- **No State Machines in the Function Block itself**: The function block delegates state management to the internal flip-flops; it is purely combinatorial with clocked takeover.

State Overview

The function block itself does not have its own visible state machine. The internal flip-flops **E_D_FF_ANY** have a dual state (Q = 0 or 1). The entire process follows a fixed pattern:

- **Idle**: No event at the input – outputs retain their last values.
- **Processing**: An event at the IN adapter is processed.
- **Output**: After the flip-flops have been clocked, all eight bytes are simultaneously available at the output adapters and are acknowledged by the respective events.

Application Scenarios

- **Decomposition of Communication Data**: Splitting a received LWORD packet (e.g., from a fieldbus or serial interface) into individual byte signals for further processing in the controller.
- **Parallel Output of Diagnostic/Status Bytes**: Splitting a composite 64-bit status word into eight separate byte outputs for display or forwarding.
- **Adapter-oriented data processing**: Integration into architectures based on adapters to ensure a strict separation of data and event interfaces.
- **Interface to sub-function blocks**: Supplies data to multiple function blocks, each requiring one byte (e.g., for seven segment displays, stepper motor control, or D/A converters).

Comparison with similar function blocks

- **SPLIT_LWORD_INTO_BYTES**: This internal function block performs the splitting without an adapter and outputs simple data outputs (BYTE) as well as an event (CNF). In contrast, **SPLIT_AL_INTO_AB** offers complete adapter encapsulation, including buffering and synchronized event output.
- **SPLIT_DWORD_INTO_BYTES** (analog, if available): Splits a 32-bit value into four bytes. SPLIT_AL_INTO_AB is designed for 64 bits and delivers eight bytes.
- **Simple MOVE Function Blocks**: Multiple MOVE blocks with bitmasks would need to be configured individually, whereas this block offers a ready-made, integrated solution.
- **Adapter-Based Alternatives**: Other splitting adapters might use symmetrical or different adapter types; this function block is specialized for AL⟶AB.

Conclusion

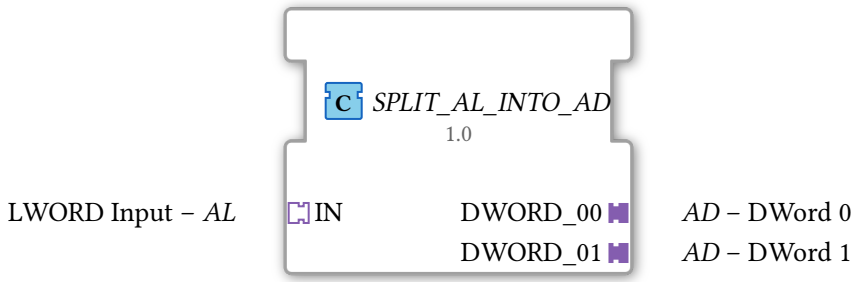
SPLIT_AL_INTO_AB is a useful and efficient function block for splitting a 64-bit data word into eight individual bytes, specifically designed for use with IEC 61499 adapters (AL/AB). The combination of an internal splitter and flip-flops ensures simultaneous, buffer-based output of all bytes. Thanks to its adapter-based interface, the block can be seamlessly integrated into modular control architectures where a clean separation of event and data channels is desired. It is particularly suitable for applications requiring parallel, structured data transfer.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

SPLIT_AL_INTO_AD



• – ◦ • – ◦ • – ◦ •

Introduction

The function block SPLIT_AL_INTO_AD splits an incoming **AL (LWORD)** adapter into two separate **AD (DWORD)** adapters. It thus implements the necessary interface conversion for applications where a large data word (LWORD) needs to be split into two smaller DWORDs and made available via unidirectional adapters.

Interface Structure

Event Inputs

- **not directly available**

Events are received via the SOCKET adapter IN (signals IN.E1). Internally, this triggers processing.

Event Outputs

- **Not directly available**

Output events occur via the PLUG adapters DWORD_00.E1 and DWORD_01.E1 as soon as a new value is available.

Data Inputs

- **IN.D1** – Input data (LWORD) from the SOCKET adapter IN.

Data Outputs

- **DWORD_00.D1** – First split DWORD (low part) on the PLUG adapter DWORD_00.
- **DWORD_01.D1** – Second split DWORD (high part) on the PLUG adapter DWORD_01.

Adapter

Type	Name	Direction	Comment
SOCKET	IN	Input	AL (LWORD) – Source Adapter
PLUG	DWORD_00	Output	AD (DWORD) – First Destination Adapter
PLUG	DWORD_01	Output	AD (DWORD) – Second Destination Adapter

Functionality

This function block operates purely event-driven. As soon as an event arrives at SOCKET IN (IN.E1), the corresponding data word (IN.D1, type LWORD) is internally forwarded to the sub-block SPLIT_LWORD_INTO_DWORDS. This sub-block splits the 64-bit word into two 32-bit DWORDs.

The two results are then passed in parallel to the inputs of the flip-flop blocks E_D_FF_ANY_00 and E_D_FF_ANY_01. Simultaneously, the splitter block triggers a common acknowledgment event (CNF) that clocks both flip-flops. This allows the new DWORD values to be adopted and output as valid output data to the respective PLUG adapters (DWORD_00.D1 and DWORD_01.D1). Simultaneously, an event is sent to the corresponding output adapter (DWORD_00.E1 and DWORD_01.E1, respectively).

The flip-flops ensure that the output data remains stable until the next valid input event and is only updated during new processing.

Technical Features

- **Full Adapter Encapsulation** – The module has no direct events or data ports, but communicates exclusively via standardized unidirectional adapters (AL/AD).
- **Internal Use of SPLIT_LWORD_INTO_DWORDS** – This separate, typed splitter module handles the data splitting; The flip-flops decouple data and event passing.
- **Zero latency for simultaneous events** – Since both flip-flops are clocked by the same CNF, both outputs are always updated synchronously.
- **No independent state behavior** – The function block is purely combinatorial with event-driven storage; there is no internal state machine.

State Overview

The function block does not have an explicit state machine. Its behavior is purely data- and event-driven:

- **Idle** – No input event; output data remains unchanged.

- **Processing** – After the arrival of `IN.E1`, the data is split and passed to the outputs.
- **Stable** – The flip-flops hold the values until the next event.

Application Scenarios

- **Industrial Automation** – When a sensor/actuator delivers an LWORD data word (e.g., a 64-bit counter or combined status/control words) via an AL adapter, but subsequent modules operate with two separate DWORD adapters.
- **Protocol Translation** – Decomposing large data packets into manageable subwords for adapter-based communication protocols.
- **Data Validation and Forwarding** – Splitting an LWORD into two DWORDs to serve different processing paths (e.g., visualization and control) in parallel.

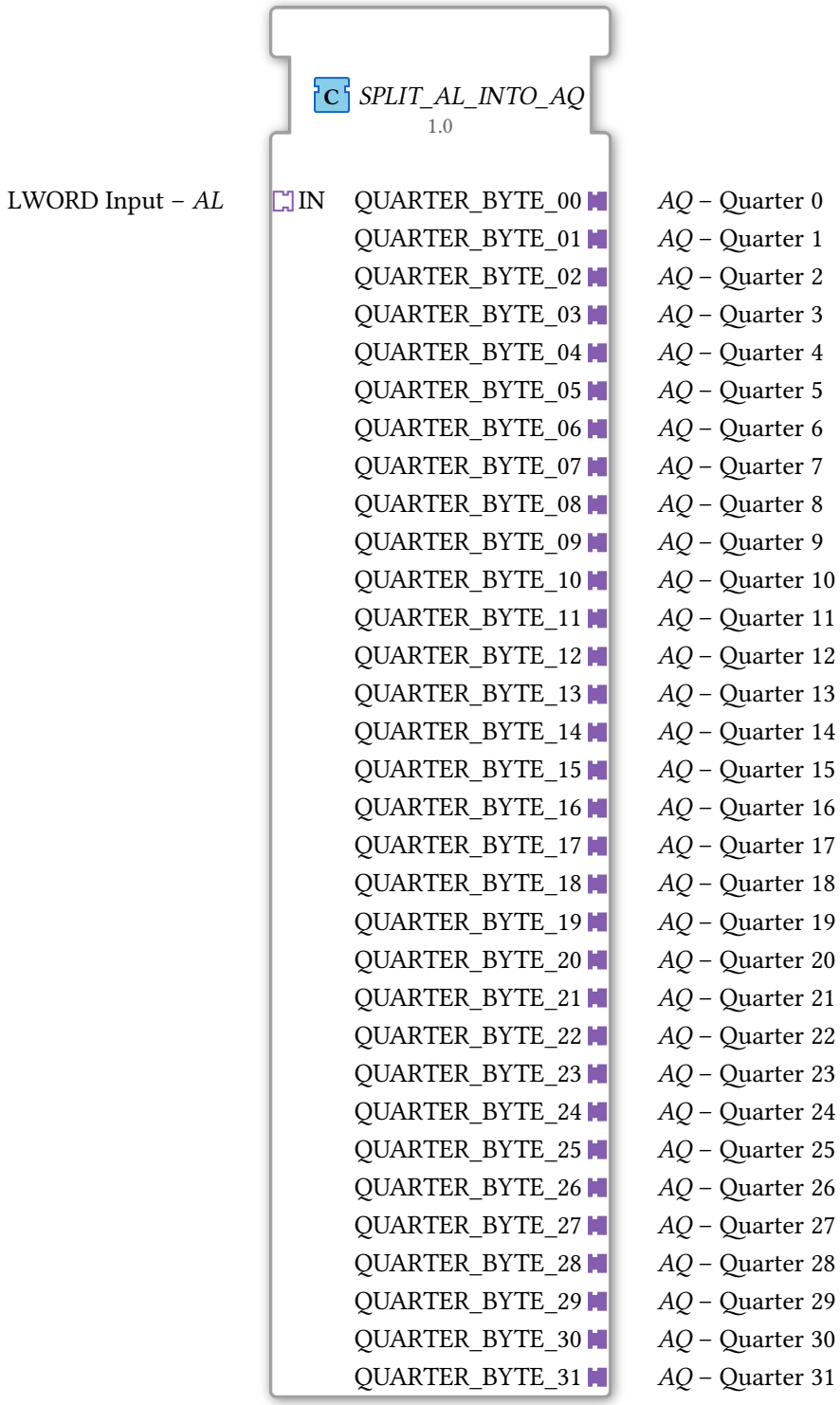
Comparison with Similar Components

- **`SPLIT_LWORD_INTO_DWORDS`** – Pure data splitter without adapter connectivity and without event synchronization. `SPLIT_AL_INTO_AD` extends this with the necessary adapter interfaces and flip-flops for correct event handling.
- **Manual Splitting** – Alternatively, the splitting could be programmed directly in the logic of the higher-level system; however, this module reduces the effort and improves reusability.

Conclusion

SPLIT_AL_INTO_AD` is a specialized yet elegant adapter module that enables seamless connectivity between an LWORD source and two DWORD sinks. Due to the clean separation of data splitting and event synchronization, it is particularly well-suited for modular, adapter-based control architectures.

SPLIT_AL_INTO_AQ



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SPLIT_AL_INTO_AQ** is a composite function block (FB) that splits an incoming LWORD value (via a `AL` adapter) into 32 separate 2-bit values and outputs each of these via its own `AQ` adapter (quarter byte). The splitting occurs synchronously with an event provided by the input adapter. The function block serves as an interface between a wide data word and several narrow, event-driven sub-segments.

Interface Structure

Event Inputs

No direct event inputs. The triggering event is provided via the incoming adapter `IN.E1`.

Event Outputs

No direct event outputs. Output events are passed on via the outgoing adapter `QUARTER_BYTE_xx.E1`.

Data Inputs

No direct data inputs. The LWORD data value is read via the incoming adapter `IN.D1`.

Data Outputs

No direct data outputs. The 2-bit data values are output via the outgoing adapter `QUARTER_BYTE_xx.D1`.

Adapters

Direction	Name	Type	Comment
Socket (Input)	`IN`	`adapter::types::unidirectional::AL`	LWORD Input (64-bit)
Plug (Output)	`QUARTER_BYTE_00` .. `QUARTER_BYTE_31`	`adapter::types::unidirectional::AQ`	32 outputs, each providing a 2-bit value (quarter)

Each adapter has one event channel and one data channel (`E1`, `D1`).

- The `AL` adapter provides an event (`E1`) and the LWORD data value (`D1`).
- The `AQ` adapters receive an event (`E1`) and the associated 2-bit value (`D1`).

Functionality

The module operates in the following steps:

1. **Event Reception**: An event at the input adapter `IN.E1` triggers the internal processing.
2. **Splitting**: The internal instance `SPLIT_LWORD_INTO_QUARTERS` splits the incoming LWORD (64 bits) into 32 consecutive 2-bit segments (quarter bytes 0 to 31). Each segment is forwarded to the data input of one of 32 `E_D_FF_ANY` flip-flops. Simultaneously, the event is distributed to the clock input (`CLK`) of all flip-flops.
3. **Output**: On the rising edge of the clock signal, the flip-flops receive the 2-bit values. The flip-flops then pass the data and an output event to the corresponding `AQ` adapters via their outputs (`Q` and `EO`). This means that the segmented values are available at all 32 outputs simultaneously.

The entire process is strictly event-driven – a new input event updates all outputs at once.

Technical Features

- **Use of D flip-flops**: The `E_D_FF_ANY` function blocks ensure that the output data is stable only after the clock event and is not affected by intermediate values.
- **Parallelization**: All 32 partial values are calculated and output in a single step. The function block is therefore deterministic and requires no loops or sequential processing.
- **Adapter-based interface**: The function block communicates exclusively via IEC 61499 adapter interfaces. This enables a clean separation of event and data flows and facilitates reuse in different contexts.
- **Use of 32-way chaining**: The vertical arrangement of the flip-flops and adapters in the network shows a systematic but very extensive structure – sufficient performance (e.g., propagation delay of the common clock signal) must be ensured during implementation.

State Overview

The component does not contain its own state machine. The internal functionality results from the combination of:

- **one** `SPLIT_LWORD_INTO_QUARTERS` (combinational partitioning)
- **32** `E_D_FF_ANY` (storage elements with set/reset states)

Each flip-flop stores the last loaded 2-bit value. A new input event overwrites all 32 values simultaneously.

Application Scenarios

- **Decomposition of a Fieldbus Data Telegram**: An LWORD contains multiple status or control bits that need to be distributed to separate actuators or sensors.
- **Parallelization of 2-Bit Signals**: When connecting BCD or quadrature encoders, multiple 2-bit pieces of information can be transmitted compactly and then processed separately.
- **Bridge between Wide and Narrow Data Buses**: If a system uses 64-bit words, but the target components only have 2-bit interfaces, this function block offers a simple way to split the data.

Comparison with Similar Function Blocks

Function Block	Output Format	Number of Outputs	Synchronization
`SPLIT_AL_INTO_AQ`	2-bit AQ adapter	32	Common event
`SPLIT_LWORD_INTO_BYTES` (hypothetical)	8-bit adapter	8	Event
`SPLIT_LWORD_INTO_WORDS` (hypothetical)	16-bit adapter	4	Event

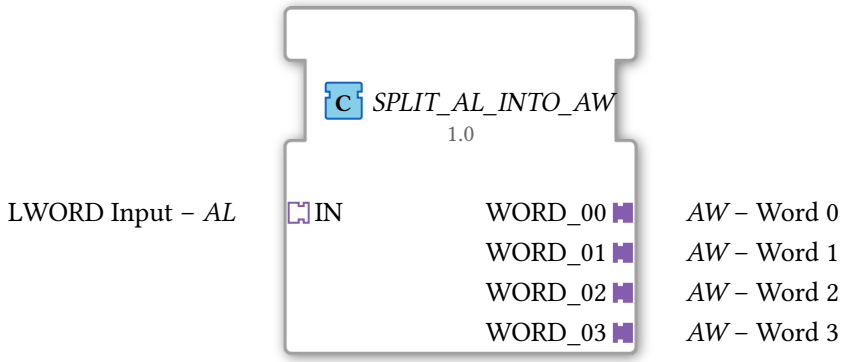
This component is specifically optimized for the fine granularity of 2-bit segments and utilizes event-driven IEC 61499 adapter technology. The main difference compared to simpler split devices lies in the number of outputs (32 instead of the typical 4 or 8) and the use of flip-flops for stable output.

This component is specifically optimized for the fine granularity of 2-bit segments and utilizes event-driven IEC 61499 adapter technology.

Conclusion

The `SPLIT_AL_INTO_AQ` function block offers efficient, parallelized partitioning of a 64-bit LWORD into 32 separate 2-bit channels. Thanks to strict event control and D flip-flop storage, deterministic and time-accurate output is guaranteed. The block is particularly suitable for applications where many narrow data streams need to be derived from a compact data word. Its adapter-based interface makes it flexible and easily integrated into existing IEC 61499 systems.

SPLIT_AL_INTO_AW



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SPLIT_AL_INTO_AW** is used to split a 64-bit integer received via an AL adapter (LWORD) into four separate 16-bit values and output them via four individual AW adapters. It implements a hard-coded split that is triggered by an incoming event.

Interface Structure

Event Inputs

- **IN.E1** (Event) – Received via the **IN** socket and starts the splitting process.

Event Outputs

- **WORD_00.E1** – Output via the **WORD_00** plug.

- **WORD_01.E1** – Output via the **WORD_01** plug.
- **WORD_02.E1** – Output via the **WORD_02** plug.
- **WORD_03.E1** – Output via the **WORD_03** plug.

Each event output is activated in parallel as soon as the splitting is complete.

Data Inputs

- **IN.D1** (LWORD) – The 64-bit input size to be split. Provided via the **IN** socket.

Data Outputs

- **WORD_00.D1** (WORD) – The least significant word (bits 0–15).
- **WORD_01.D1** (WORD) – The second word (bits 16–31).
- **WORD_02.D1** (WORD) – The third word (bits 32–47).
- **WORD_03.D1** (WORD) – The most significant word (bits 48–63).

Each output is provided by an edge-triggered memory (E_D_FF_ANY) and remains stable until the next split operation.

Adapters

- **Socket IN** – Type: `adapter::types::unidirectional::AL` (LWORD)
- **Plugs WORD_00 ... WORD_03** – Type: `adapter::types::unidirectional::AW` (WORD)

Operation

1. An incoming event at **IN.E1** activates the internal function block **SPLIT_LWORD_INTO_WORDS** via its event input **REQ**.

Socket IN – Type: `adapter::types::unidirectional::AL` (LWORD)
Plugs WORD_00 ... WORD_03 – Type: `adapter::types::unidirectional::AW` (WORD)
Functionality
An incoming event at IN.E1 activates the internal function block SPLIT_LWORD_INTO_WORDS via its event input REQ.
**** 2. Simultaneously, the data **IN.D1** is transferred to the data input **IN** of the split function block.

1. **SPLIT_LWORD_INTO_WORDS** splits the LWORD into four word parts (**WORD_00**...**WORD_03**) and generates a termination event at **CNF**.
2. This termination event is forwarded to all four edge-triggered flip-flops (**E_D_FF_ANY**). Each flip-flop receives its assigned subword at its data input **D** and places it at its output **Q**.
3. Simultaneously, the event outputs **WORD_00.E1** ... **WORD_03.E1** are activated so that the downstream adapters can receive the new data together.
4. This termination event is forwarded to all four edge-triggered flip-flops (**E_D_FF_ANY**).

Technical Features

- **Use of E_D_FF_ANY**: Each partial value is temporarily stored in an edge-triggered flip-flop. This ensures that the output value is retained even if no new event is present at the input. The values are only updated during the next splitting operation.
- **Parallel Output**: All four output events are triggered simultaneously – the data is available at all plugs at the same time.
- **Adapter-Based Interface**: The function block works exclusively with adapters, enabling clean encapsulation and reusability in modular applications.

State Overview

The function block itself does not have an explicit state machine. The internal flip-flops (E_D_FF_ANY) can assume the following states:

- **Idle**: No event is present – the output plugs hold the last stored value.
- **Busy**: The function block is briefly occupied during processing (between **IN.E1** and the appearance of the output events).
- **Output Active**: Upon completion, the new data is output to the plugs and the events are sent.

Application Scenarios

- **De-Seeralization**: Splitting a 64-bit telegram component into individual 16-bit register values for further processing.

- **Control Engineering**: Splitting a long bitmask into multiple submasks that are sent to different actuators or display units.
- **Data Preparation**: Decomposing an LWORD measurement value (e.g., from an encoder or an analog-to-digital converter card) into usable word channel values.

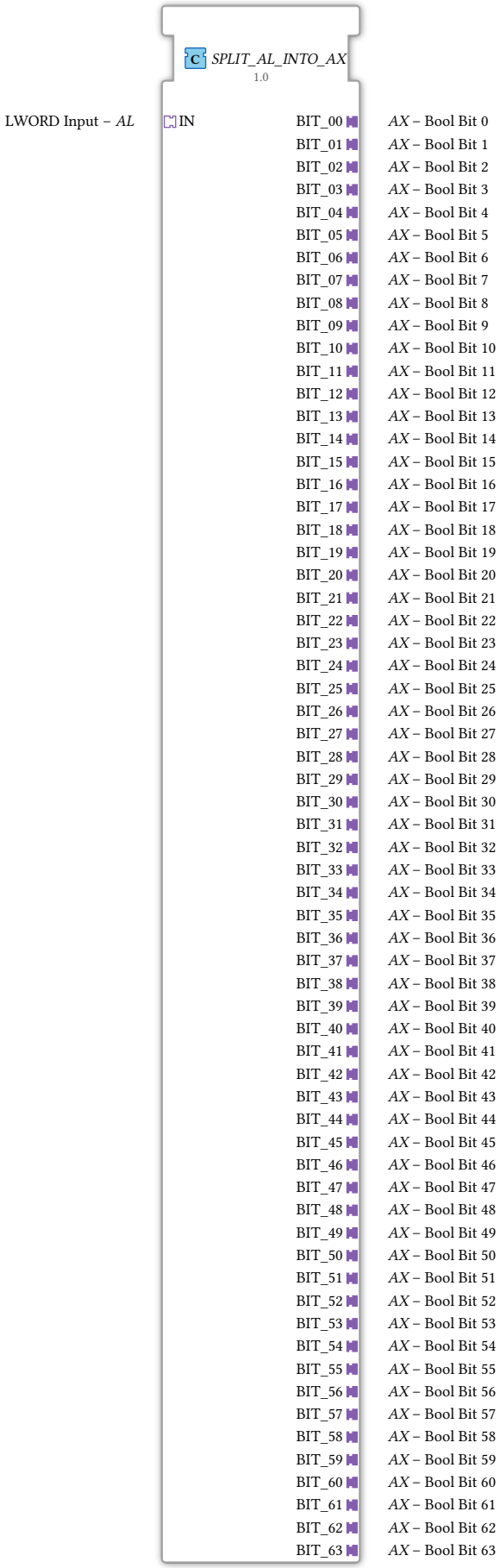
Comparison with Similar Components

- **SPLIT_LWORD_INTO_WORDS** (not adapter-based): Works with pure data and event ports. **SPLIT_AL_INTO_AW** encapsulates this function block and adds the adapter interfaces and edge-triggered storage.
- **MUX** or **DEMUX** (IEC 61499 standard): These typically split or combine data streams at the bit or bit level, while **SPLIT_AL_INTO_AW** is specifically designed for fixed 64-bit partitioning.

Conclusion

SPLIT_AL_INTO_AW offers a simple, robust solution for splitting an LWORD into four WORDS using adapters. Thanks to its integrated edge-triggered buffering, the function block is particularly well-suited for event-driven systems where data must be available in parallel and stably at different outputs. The adapter-based interface simplifies integration into existing component libraries.

SPLIT_AL_INTO_AX


SPLIT_AL_INTO_AX
1.0
LWORD Input – AL
IN
BIT_00 AX – Bool Bit 0
BIT_01 AX – Bool Bit 1
BIT_02 AX – Bool Bit 2
BIT_03 AX – Bool Bit 3
BIT_04 AX – Bool Bit 4
BIT_05 AX – Bool Bit 5
BIT_06 AX – Bool Bit 6
BIT_07 AX – Bool Bit 7
BIT_08 AX – Bool Bit 8
BIT_09 AX – Bool Bit 9
BIT_10 AX – Bool Bit 10
BIT_11 AX – Bool Bit 11
BIT_12 AX – Bool Bit 12
BIT_13 AX – Bool Bit 13
BIT_14 AX – Bool Bit 14
BIT_15 AX – Bool Bit 15
BIT_16 AX – Bool Bit 16
BIT_17 AX – Bool Bit 17
BIT_18 AX – Bool Bit 18
BIT_19 AX – Bool Bit 19
BIT_20 AX – Bool Bit 20
BIT_21 AX – Bool Bit 21
BIT_22 AX – Bool Bit 22
BIT_23 AX – Bool Bit 23
BIT_24 AX – Bool Bit 24
BIT_25 AX – Bool Bit 25
BIT_26 AX – Bool Bit 26
BIT_27 AX – Bool Bit 27
BIT_28 AX – Bool Bit 28
BIT_29 AX – Bool Bit 29
BIT_30 AX – Bool Bit 30
BIT_31 AX – Bool Bit 31
BIT_32 AX – Bool Bit 32
BIT_33 AX – Bool Bit 33
BIT_34 AX – Bool Bit 34
BIT_35 AX – Bool Bit 35
BIT_36 AX – Bool Bit 36
BIT_37 AX – Bool Bit 37
BIT_38 AX – Bool Bit 38
BIT_39 AX – Bool Bit 39
BIT_40 AX – Bool Bit 40
BIT_41 AX – Bool Bit 41
BIT_42 AX – Bool Bit 42
BIT_43 AX – Bool Bit 43
BIT_44 AX – Bool Bit 44
BIT_45 AX – Bool Bit 45
BIT_46 AX – Bool Bit 46
BIT_47 AX – Bool Bit 47
BIT_48 AX – Bool Bit 48
BIT_49 AX – Bool Bit 49
BIT_50 AX – Bool Bit 50
BIT_51 AX – Bool Bit 51
BIT_52 AX – Bool Bit 52
BIT_53 AX – Bool Bit 53
BIT_54 AX – Bool Bit 54
BIT_55 AX – Bool Bit 55
BIT_56 AX – Bool Bit 56
BIT_57 AX – Bool Bit 57
BIT_58 AX – Bool Bit 58
BIT_59 AX – Bool Bit 59
BIT_60 AX – Bool Bit 60
BIT_61 AX – Bool Bit 61
BIT_62 AX – Bool Bit 62
BIT_63 AX – Bool Bit 63

• – ◦ • – ◦ • – ◦ •

Introduction

The **SPLIT_AL_INTO_AX** function block decomposes a value encoded as an LWORD (64-bit) into 64 individual BOOL signals. The input value is provided via an **AL adapter** (analog longword), and the output is provided via 64 separate **AX adapters** (analog bits). Each output adapter provides the status of a bit along with a corresponding event. This function block is particularly suitable for processing digital signals from a compact data word, e.g., for the parallel control of 64 discrete outputs.

Interface Structure

Event Inputs

- **IN.E1** (in the IN adapter): Starts processing as soon as a new LWORD value is present at the adapter.

Event Outputs

Each AX adapter provides an event output:

- **BIT_00.E1** .. **BIT_63.E1** (in the corresponding adapters): Triggered as soon as the associated bit in the internal flip-flop is updated.

Data Inputs

- **IN.D1** (LWORD): The 64-bit data set to be parsed.

Data Outputs

- **BIT_00.D1** .. **BIT_63.D1** (BOOL): The state of the respective bit (0 or 1).

Adapters

- **IN** (Socket, Type: `adapter::types::unidirectional::AL`): Returns the LWORD value and the start event.
- **BIT_00** .. **BIT_63** (Plugs, Type: `adapter::types::unidirectional::AX`): Each outputs a BOOL signal (bit value) and an associated event.

Functionality

1. An incoming LWORD value via the AL adapter **IN** triggers processing via the **E1** event.
2. The internal component **SPLIT_LWORD_INTO_BOOLS** extracts all 64 bits from the LWORD.
3. Each bit is assigned to the **D** data input of an associated **E_D_FF** (Edge-Triggered D Flip-Flop).
4. Simultaneously, all 64 flip-flops receive the **CLK** clock pulse via the **CNF** event of the splitter. This transfers all bits to the flip-flops at once.
5. The flip-flops output the stored value at their output **Q**, which is provided via the data output **D1** of the respective AX adapter.
6. Each flip-flop generates an event at its output **EO**, which is forwarded to the event input **E1** of the corresponding AX adapter.

Technical Features

- **Synchronization**: All bits are transferred to the flip-flops simultaneously via a common event (CNF of SPLIT_LWORD_INTO_BOOLS). This ensures a consistent output state without race conditions.
- **Storage**: The use of **E_D_FF** ensures that the output state is retained until a new LWORD value is processed. The function block thus operates as a "sample and hold" for all 64 bits.
- **Large Number of Outputs**: The function block offers 64 independent BOOL outputs, which can be mapped to many device adapters in the 4diac IDE.
- **Adapter-based interface**: Inputs and outputs are implemented as unidirectional adapters, which facilitates modular encapsulation and reuse.

State overview

The function block (FB) does not have its own explicit state machine. The internal state is represented by the 64 **E_D_FF**:

- **Initial**: All flip-flops are set to FALSE (0).
- **After processing**: Each flip-flop stores the current bit value of the last received LWORD. The state changes only upon a new event on **IN.E1**.

Application scenarios

- **Parallel output of digital signals** from a serial or packetized data word, e.g., from a bus system or a communication interface.
- **Masked control**: Combination with logic blocks to selectively evaluate individual bits of a status word.
- **Test and Diagnostic Interfaces**: Processes a 64-bit error word for display on separate lights or in visualizations.
- **Interface to Discrete Actuators**: Each AX adapter can be directly connected to a digital output module.

Comparison with Similar Components

- **SPLIT_LWORD_INTO_BOOLS** (purely functional splitter without memory): Provides the bits only during processing and does not retain the value. **SPLIT_AL_INTO_AX** adds the memory function and provides a stable output until the next update.
- **Type Converters like DWORD_TO_BOOL_ARRAY**: These usually operate at the array level and not at the adapter level; the present component is specifically designed for the 4diac adapter interface.
- **Flip-flop-based solutions, each with its own F_TRIG or similar:** This function block simplifies configuration because all 64 bits are combined in a single block.

Conclusion

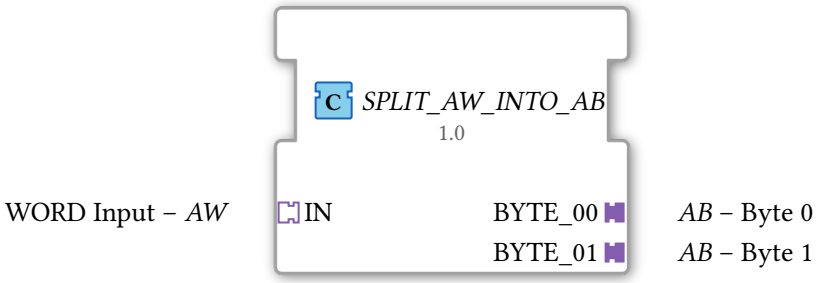
SPLIT_AL_INTO_AX is a powerful, adapter-based function block for splitting an LWORD data set into 64 discrete BOOL signals. The integrated flip-flops maintain the output state until the next update. The large number of outputs and synchronous processing make it ideal for applications requiring parallel, time-consistent signal generation. The use of unidirectional adapters ensures clear interface definition and facilitates integration into existing 4diac projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

SPLIT_AW_INTO_AB



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SPLIT_AW_INTO_AB** is used to split a word data packet (type AW, 16 bits) into two separate byte data packets (type AB, 8 bits each). The input data is received via a socket and output via two separate plugs. This function block is typically used in environments that rely on adapter-based communication, e.g., in automation technology when decomposing word-based protocols.

Interface Structure

Event Inputs

The function block does not have its own event inputs. Event control is handled indirectly via the **IN** adapter.

Event Outputs

The function block does not have its own event outputs. Events are transmitted externally via the output adapters **BYTE_00** and **BYTE_01**.

Data Inputs

The FB has no direct data inputs. Data input is indirect via the **IN** adapter.

Data Outputs

The FB has no direct data outputs. Data output is indirect via the **BYTE_00** and **BYTE_01** adapters.

Adapters

Adapter	Direction	Type	Description
IN	Socket	`adapter::types::unidirectional::AW`	Receives a 16-bit word (WORD). The adapter provides the data and an associated event.
BYTE_00	Plug	`adapter::types::unidirectional::AB`	Outputs the least significant byte (byte 0).
BYTE_01	Plug	`adapter::types::unidirectional::AB`	Outputs the most significant byte (byte 1).

Functionality

1. An incoming event on the **IN** (E1) adapter triggers the internal function block **SPLIT_WORD_INTO_BYTES**.
2. This splits the received WORD (16 bits) into two bytes: **BYTE_00** (less significant) and **BYTE_01** (most significant).
3. The completion event (CNF) of **SPLIT_WORD_INTO_BYTES** is passed to the clock inputs of two **E_D_FF_ANY** flip-flops.
4. Each flip-flop stores the corresponding byte and outputs it via its **Q** output on the next clock pulse.
5. The flip-flops each generate an event on **EO**, which is sent to the output adapters **BYTE_00** and **BYTE_01**. This makes the split bytes available externally along with an event.

Technical Features

- **Adapter-based interface:** The function block (FB) does not contain any direct event or data inputs/outputs. All communication takes place via adapters (sockets for inputs, plugs for outputs). It is therefore particularly suitable for applications that rely on the adapter-based 4diac type system.
- **Data caching:** The two **E_D_FF_ANY** flip-flops keep the split bytes stable until new data arrives. This prevents inconsistent states during the splitting process.
- **Event synchronization:** The bytes are output simultaneously because the same CNF event triggers both flip-flops.

State Overview

The function block (FB) does not have an explicit state machine. It processes pulses in an event-driven manner and outputs the split bytes on **IN** for each event. The flip-flops have the typical two states: stored value (0 or 1 per bit, depending on the byte).

Application Scenarios

- **Byte-Oriented Communication:** When a sensor or actuator sends data in WORD format, but downstream components expect individual bytes.
- **Protocol Preparation:** Splitting telegrams consisting of a 16-bit value into separate byte adapters for further processing.
- **Modbus/IO-Link Integration:** Adapting 16-bit register values to 8-bit interfaces.

Comparison with Similar Components

- **SPLIT_WORD_INTO_BYTES:** A pure data splitter without adapter encapsulation. It operates at the data level and requires its own event and data interfaces.
- **MUX/DEMUX Components:** Multiplexers or demultiplexers for multiple values, but usually with different data types and without adapter integration.
- **Adapter Wrappers like SPLIT_AW_INTO_AB:** Specifically developed for the adapter-based 4diac architecture. They encapsulate the splitting logic behind standardized adapter interfaces.

Conclusion

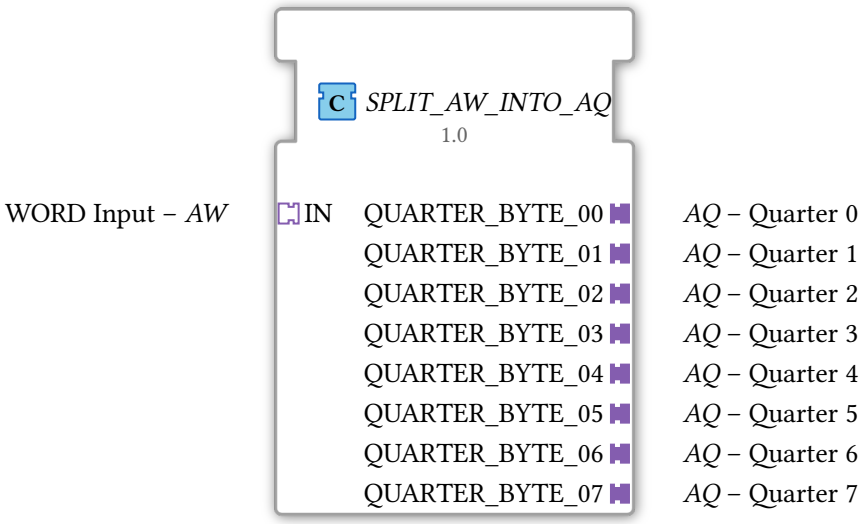
SPLIT_AW_INTO_AB is a specialized adapter wrapper that cleanly splits an incoming WORD into two separate BYTE adapters. The use of flip-flops ensures a stable data flow and avoids timing inconsistencies. This component is ideal for systems that use adapter-based communication according to IEC 61499-2 and require simple transformation between 16-bit and 8-bit data.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

SPLIT_AW_INTO_AQ



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SPLIT_AW_INTO_AQ** divides the eight quarter adapters (AQ) of a word adapter (AW). The incoming word (16 bits) is split into eight quarter units (2 bits each) and output via the corresponding AQ plugs.

Interface Structure

Event Inputs

- `IN.E1` (via Socket IN) – Triggers the splitting of a new word value.

Event Outputs

- `QUARTER_BYTE_00.E1` to `QUARTER_BYTE_07.E1` – Signals the availability of the respective quarter result.

Data Inputs

- `IN.D1` – Receives the incoming word (16 bits, type compatible with the AW adapter).

Data Outputs

- `QUARTER_BYTE_00.D1` to `QUARTER_BYTE_07.D1` – Provide the eight divided quarter values (2 bits each).

Adapters

- **Socket:** `IN` – Adapter type `adapter::types::unidirectional::AW`
- **Plugs:** `QUARTER_BYTE_00` ... `QUARTER_BYTE_07` – Adapter type `adapter::types::unidirectional::AQ`

Functionality

1. An event at input `IN.E1` activates the internal module `SPLIT_WORD_INTO_QUARTERS`.

2. `SPLIT_WORD_INTO_QUARTERS` decomposes the 16-bit word received via `IN.D1` into eight separate 2-bit quarter values.
3. The split values are temporarily stored in parallel in eight `E_D_FF_ANY` flip-flops (`D` inputs).
4. After processing is complete (CNF event of `SPLIT_WORD_INTO_QUARTERS`), all flip-flops simultaneously receive a clock pulse (`CLK`).
5. The outputs `Q` of the flip-flops are connected to the corresponding `QUARTER_BYTE_xx.D1` outputs.
6. Simultaneously, the event `QUARTER_BYTE_xx.E1` is triggered to signal the data transfer to downstream components. ...``

Technical Features

- **Synchronization via Flip-Flops:** All eight quarter values are taken from the internal splitter in sync with the clock.
- **No Persistent State Storage:** The chip only buffers the quarter values until the next processing cycle.
- **Uniform Interfaces:** The adapters follow the unidirectional AW/AQ protocol and enable easy integration in the 4diac IDE.

State Overview

The chip does not have its own state machine. The internal `E_D_FF_ANY` flip-flops can be in two states:

- **Stores Previous Value** – Until a new event occurs.
- **Updated Value** – After a clock pulse from the CNF event.

Application Scenarios

- **Data Preparation:** Splitting a 16-bit word from a communication adapter into eight 2-bit quarter-wave signals, e.g., for parallel processing in control logic.
- **Multiplexing/Demultiplexing:** Separating word data streams into separate quarter-wave channels.
- **Bus Structure Extension:** Easily splitting address or control words across multiple subsequent components.

Comparison with Similar Components

- **SPLIT_BYTE_INTO_NIBBLES** – Splits a byte into two 4-bit nibbles.
- **SPLIT_WORD_INTO_BYTES** – Splits a 16-bit word into two 8-bit bytes.
- **SPLIT_AW_INTO_AQ** is specifically designed for splitting one AW adapter (Word) into eight AQ adapters (Quarters). The interfaces are directly tailored to the adapter types and require no manual configuration.

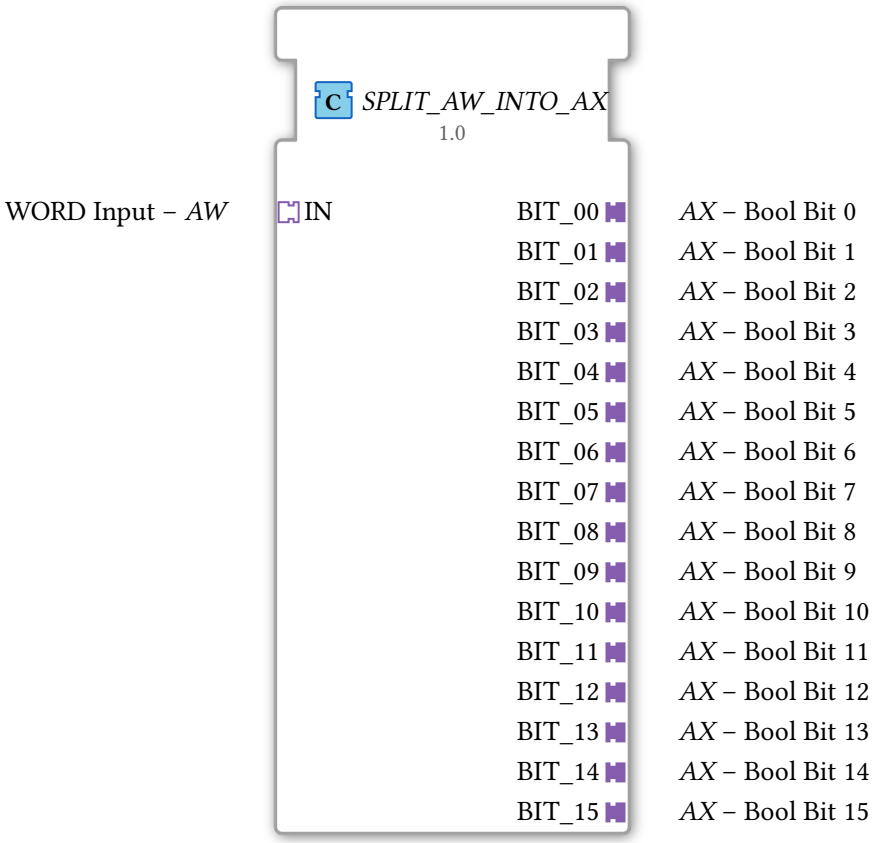
Conclusion

The **SPLIT_AW_INTO_AQ** module offers a compact and reliable way to split an incoming Word signal into eight Quarter signals. Thanks to its integrated clock synchronization and standardized adapter interfaces, it is ideally suited for modular IEC 61499 applications that require simultaneous data processing for multiple participants.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SPLIT_AW_INTO_AX



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_AW_INTO_AX` is used to split a 16-bit word (type `AW`) into 16 individual binary signals (type `AX`). Each of the 16 outputs represents one bit of the incoming word and is provided as an independent adapter with event and data lines. The splitting occurs synchronously upon the arrival of an event at the input adapter.

Interface Structure

Event Inputs

The function block does not have any independent event inputs on the interface. The initial event is received via the socket adapter `IN` (see Adapters).

Event Outputs

This module does not have any independent event outputs on the interface. Output events are provided via the plug adapters `BIT_00` ... `BIT_15` (see Adapters).

Data Inputs

This module does not have any independent data inputs on the interface. The 16-bit value to be processed is provided as a data value `D1` (type `WORD`) via the socket adapter `IN` (see Adapters).

Data Outputs

This module does not have any independent data outputs on the interface. The 16 extracted binary values are output via the plug adapters `BIT_00` ... `BIT_15` as data values `D1` (type `BOOL`).

Adapter

Direction	Name	Type	Comment

Socket (Input)	`IN`	`adapter::types::unidirectional::AW`	16-bit word as input, event input `E1`, data input `D1`
Plug (Output)	`BIT_00` … `BIT_15`	`adapter::types::unidirectional::AX`	One bit of each word, event output `E1`, data output `D1`

Functionality

1. An external event at socket `IN.E1` triggers processing.
2. The current value of `IN.D1` (type `WORD`, 16 bits) is read.
3. The internally embedded component `SPLIT_WORD_INTO_BOOLS` divides the word into 16 individual Boolean values (`BIT_00` … `BIT_15`).
4. These 16 values are passed to a `E_D_FF` (D flip-flop) at each clock cycle. The flip-flops receive the data at each clock cycle (here, the event `CNF` from `SPLIT_WORD_INTO_BOOLS`).
5. Each flip-flop outputs its stored value via the associated plug adapter `BIT_xx.D1` and an event via `BIT_xx.E1`.

… This distributes the entire word value across the 16 output adapters in a single cycle and holds it there until the next event.

Technical Features

- **Event-Synchronous Distribution:** The entire distribution occurs within a single event cycle – all 16 outputs are updated simultaneously.
- **Storage:** Each bit value is held by its own `E_D_FF`, ensuring the outputs remain stable even without constantly repeated events.
- **Adapter-Based Input/Output:** The module communicates exclusively via adapters – it can therefore be seamlessly integrated into 4diac systems that use unidirectional adapters for word or Boolean signals.
- **No Dedicated Event/Data Inputs:** The entire interface is implemented via the adapters; direct wiring on the facade is not required.

State Overview

The module does not have its own top-level state machine. The internal state is determined by the 16 D flip-flops (`E_D_FF_00` … `E_D_FF_15`):

- Each flip-flop can be in one of two states: `Q = 0` or `Q = 1`.
- The state is updated only on each clock cycle (i.e., with each new event arriving at the input).
- In the idle state between events, the last stored value is retained.

Application Scenarios

- **Control Tasks:** A higher-level controller or PLC sends a 16-bit word (e.g., as a control word) – the function block converts it into 16 individual binary control signals (e.g., for relays, valves, or indicators).
- – **Protocol Conversion:** Input signals from a bus system are stored as a word and must be distributed to discrete outputs.
- **Testing and Simulation:** Representation of a word as 16 Boolean channels for visualization or troubleshooting.

Comparison with Similar Components

- **`SPLIT_WORD_INTO_BOOLS`** – This component also divides a word into Boolean values, but without adapters and without event-driven output. It serves as an internal component of the present component.
- **`SPLIT_AW_INTO_AX`** – Extends the pure data division with adapters and event output, so that the individual bits are available as fully-fledged AX interfaces (with their own event and data). This enables direct interconnection with other 4diac components that expect AX adapters.
- **Alternative In-House Development:** Theoretically, 16 separate `SLICE` modules could be used to extract bits from a word – however, this would be more complex and would not offer synchronous event output.

Conclusion

SPLIT_AW_INTO_AX` is a compact yet powerful converter that reliably and event-synchronously divides a 16-bit word into 16 individual binary outputs. Thanks to the use of adapters and flip-flops, it is particularly well-suited for use in modular IEC 61499 applications where a clean separation of word and bit

interfaces is required. It simplifies interface adaptation and improves the readability and maintainability of the application network.

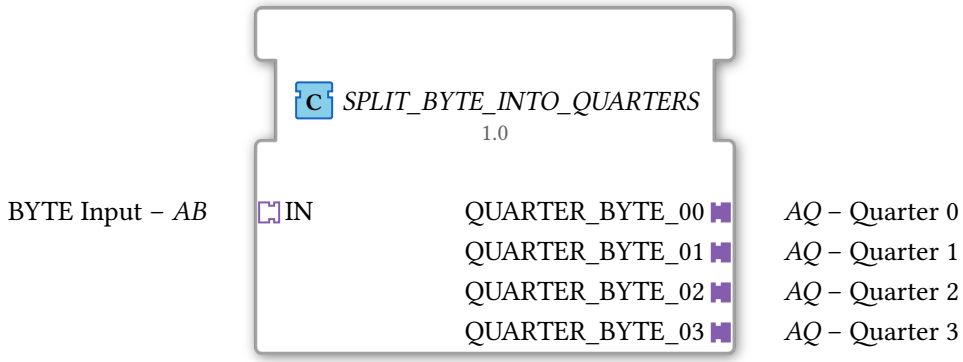
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

SPLIT_BYTE_INTO_QUARTERS

Image not available



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_BYTE_INTO_QUARTERS` is used to split an incoming byte data value into four separate quarter outputs. It implements a logical decomposition of a byte (8 bits) into four 2-bit groups, which are routed externally via unidirectional adapters. The internal implementation uses a combination of flip-flops and a recursive instance of the same function block to control the split synchronously and data-driven.

Interface Structure

Event Inputs

Not present. The function block has no independent event inputs; control is exclusively via the adapter interfaces.

Event Outputs

Not present.

Data Inputs

Not present.

Data Outputs

Not available.

Adapter

Direction	Name	Type	Description
Socket (Input)	`IN`	`adapter::types::unidirectional::AB`	Byte input that provides the data to be split.
Plug (Output)	`QUARTER_BYTE_00`	`adapter::types::unidirectional::AQ`	Output for the first quarter (least significant 2 bits).
Plug (Output)	`QUARTER_BYTE_01`	`adapter::types::unidirectional::AQ`	Output for the second quarter (bits 2-3).

Plug (Output)	QUARTER_BYTE_02	adapter::types::unidirectional::AD	Output for the third quarter (bits 4-5).
Plug (Output)	QUARTER_BYTE_03	adapter::types::unidirectional::AD	Output for the fourth quarter (most significant 2 bits).

Functionality

The function block operates as a **composite function block** with an internal network. As soon as socket `IN` receives an event (via `E1`) and the associated data, this data is forwarded to an internal instance of the same function block (`SPLIT_BYTE_INTO_QUARTERS`). Their outputs (also four quarter-wave adapters) are buffered in four parallel **E_D_FF_ANY flip-flops**. The flip-flops receive the data at the clock signal from the internal `CNF` event chain and synchronously pass it on to the outer plug-in adapters (`QUARTER_BYTE_00` ... `QUARTER_BYTE_03`). The precise mapping of the bits to the quarter-waves is implicit via the internal wiring; it is assumed that the byte is divided into four consecutive 2-bit groups (from LSB to MSB).

Technical Features

- **Recursive Type Usage**: The internal network topology includes another instance of the `SPLIT_BYTE_INTO_QUARTERS` component, enabling multi-stage or iterative decomposition. The actual depth, however, remains limited to one level due to the fixed wiring.
- **Adapter-based design**: Instead of conventional data ports, only unidirectional adapters are used, encapsulating both event and data channels. This promotes loose coupling and reusability.
- **Synchronization via flip-flops**: The four output flip-flops ensure that all quarters are updated simultaneously with a common clock signal as soon as the internal partitioning is complete.

State overview

The component has **no explicit state machine** (ECC). Its functionality is solely determined by the data and event wiring of the internal components. The system behavior is therefore purely combinatorial, triggered by a clock edge from the flip-flops.

Application scenarios

- **Parallelization of byte data**: Partitioning a received byte into four independent 2-bit channels, e.g. B. for controlling four parallel output devices or for distributed processing.
- **Serial-to-Parallel Conversion**: Conversion of a serial byte data stream into four parallel quarter signals.
- **Protocol Preparation**: Extracting partial information from a byte when each quarter has a separate meaning (e.g., status flags, configuration values).

Comparison with Similar Components

- **SPLIT_WORD_INTO_HALF_WORDS**: Analog component for splitting a 16-bit word into two 8-bit halves; usually operates with direct data ports.
- **EXTRACT_BITS_4TO1**: Standard splitting components often operate bitwise or with selectable index ranges. This module is specifically optimized for the fixed quarter-division of a byte and uses adapters instead of simple data inputs/outputs.
- **MUX/DEMUX Modules**: Multiplexers and demultiplexers offer generic division but require additional control lines. The `SPLIT_BYTE_INTO_QUARTERS` is more specialized and easier to configure.

Conclusion

The `SPLIT_BYTE_INTO_QUARTERS` function block provides a compact, adapter-based solution for decomposing a byte into four quarter-division outputs. The combination of a recursive structure and flip-flop synchronization ensures reliable, parallel data delivery. It is suitable for applications requiring a fixed division of an 8-bit value into four 2-bit units and offers good integration capabilities into modular 4diac projects thanks to its standardized adapter interface.

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.10.0.2 types

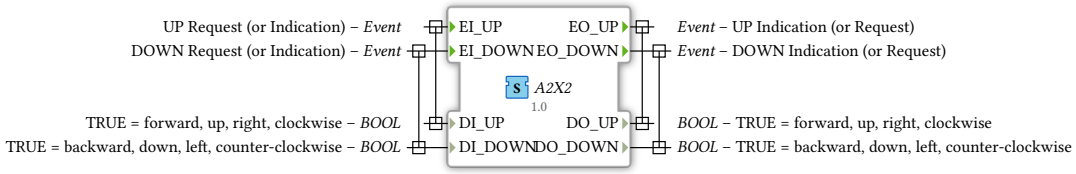
Bidirectional Adapters

Numerical Index (bidirectional)

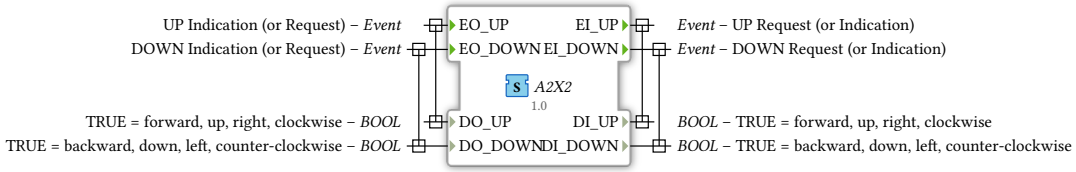
Here you will find all exercises from the `bidirectional` workspace in ascending order. ...

A2X2

Plug (adapter output)



Socket (adapter input)



bidirectional adapter interface for 2 events and 2 bools

Interface

Event Inputs

Name	Comment	With
EI_UP	UP Request (or Indication)	DI_UP
EI_DOWN	DOWN Request (or Indication)	DI_DOWN

Event Outputs

Name	Comment	With
EO_UP	UP Indication (or Request)	DO_UP
EO_DOWN	DOWN Indication (or Request)	DO_DOWN

Input Vars

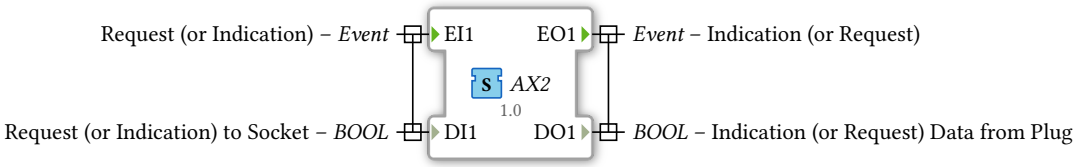
Name	Type	Comment
DI_UP	BOOL	TRUE = forward, up, right, clockwise
DI_DOWN	BOOL	TRUE = backward, down, left, counter-clockwise

Output Vars

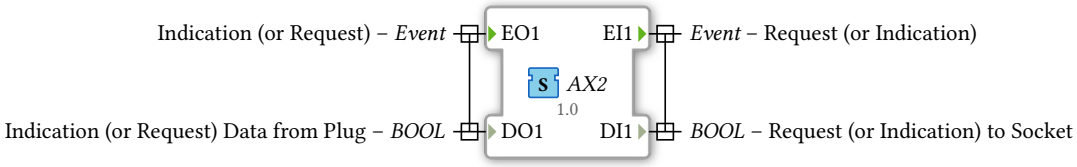
Name	Type	Comment
DO_UP	BOOL	TRUE = forward, up, right, clockwise
DO_DOWN	BOOL	TRUE = backward, down, left, counter-clockwise

AX2

Plug (adapter output)



Socket (adapter input)



bidirectional adapter interface for 1 event and 1 bool

Interface

Event inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

Name	Type	Comment
DI1	BOOL	Request (or Indication) to Socket

Output Vars

Name	Type	Comment
DO1	BOOL	Indication (or Request) Data from Plug

Bidirectional BOOL adapter

🎧 Podcast

- George Boole: How an autodidact laid the foundation of our digital world with 0 and 1

🛠️ Related exercises

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056

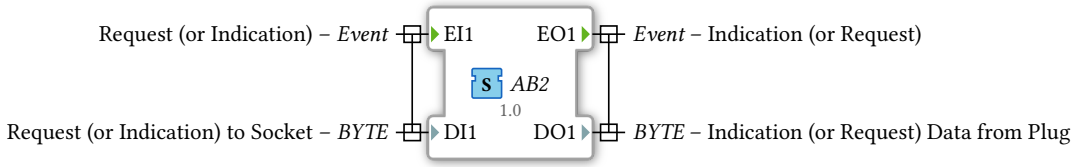
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

Numerical Index (bidirectional)

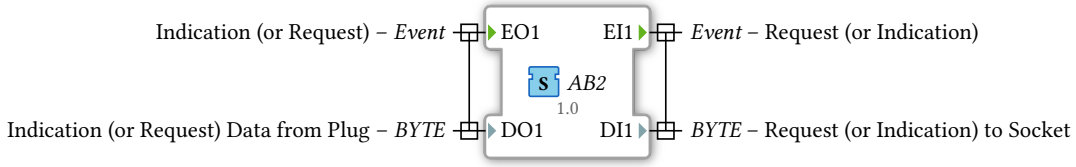
Here you will find all exercises from the `bidirectional` workspace in ascending order. ...

AB2

Plug (adapter output)



Socket (adapter input)



bidirectional adapter interface for 1 event and 1 byte

Interface

Event inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

Name	Type	Comment
DI1	BYTE	Request (or Indication) to Socket

Output Vars

Name	Type	Comment
DO1	BYTE	Indication (or Request) Data from Plug

Bidirectional BYTE adapter

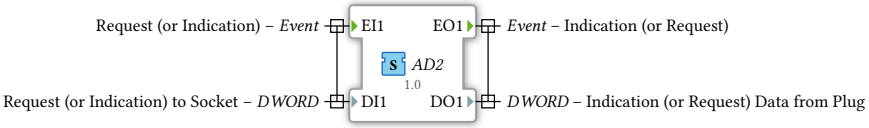
🛠️ Related exercises

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

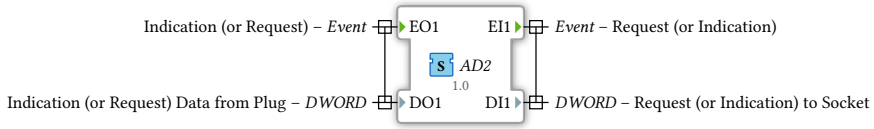
DWORD

AD2

Plug (adapter output)



Socket (adapter input)



bidirectional adapter interface for 1 event and 1 dword

Interface

Event inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

Name	Type	Comment
DI1	DWORD	Request (or Indication) to Socket

Output Vars

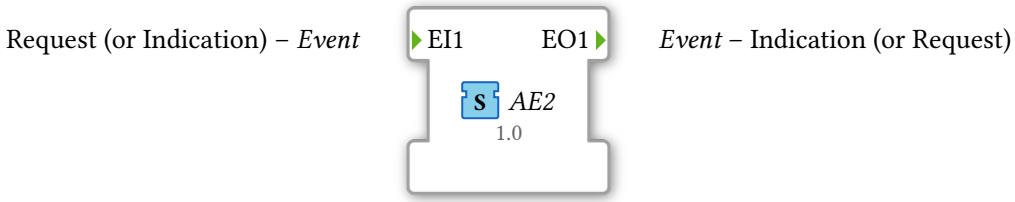
Name	Type	Comment
DO1	DWORD	Indication (or Request) Data from Plug

Numerical Index (bidirectional)

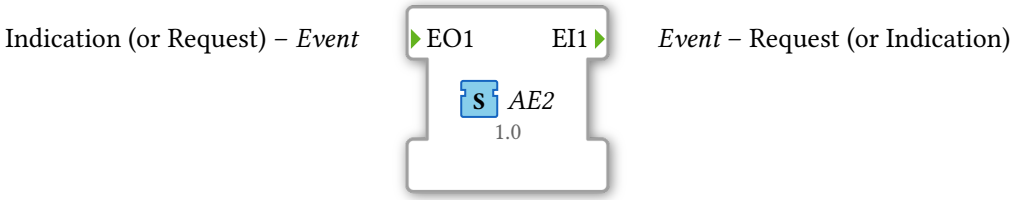
Here you will find all exercises from the `bidirectional` workspace in ascending order. ...

AE2

Plug (adapter output)



Socket (adapter input)



bidirectional adapter interface for 1 event

Interface

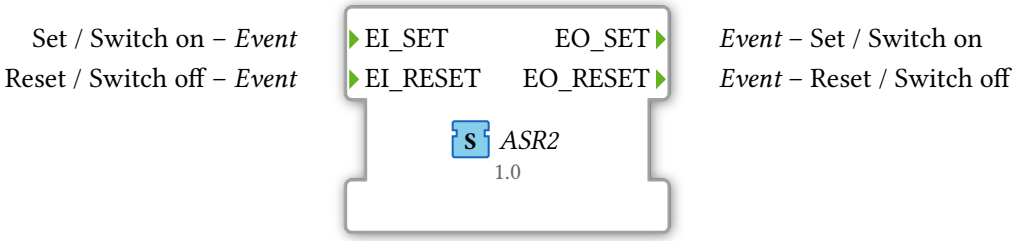
Event inputs

Name	Comment	With
EI1	Request (or Indication)	

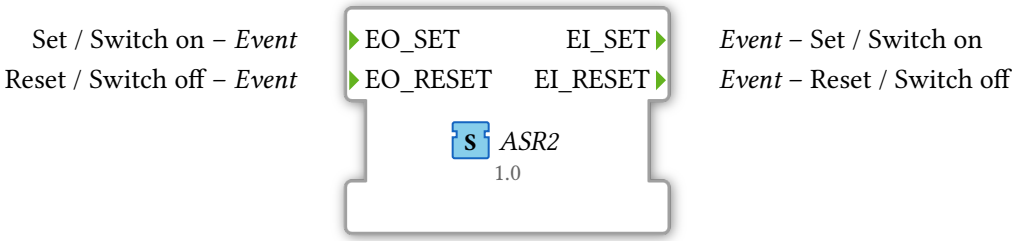
Event Outputs

Name	Comment	With
EO1	Indication (or Request)	

ASR2
Plug (adapter output)



Socket (adapter input)



bidirectional adapter interface for 2 events

Interface

Event inputs

Name	Comment	With
EI_SET	Set/Switch on	
EI_RESET	Reset/Switch off	

Event Outputs

Name	Comment	With
EO_SET	Set/Switch on	

EO_RESET	Reset/Switch off	

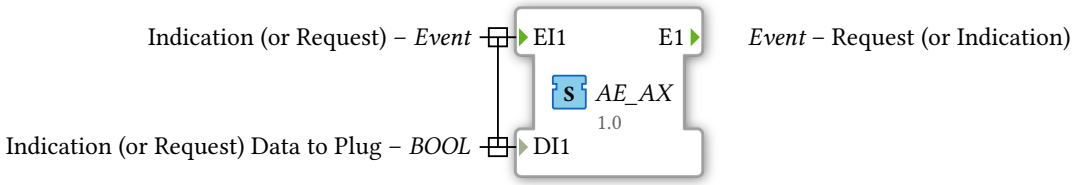
Bidirectional Event Adapter

🎧 Podcast

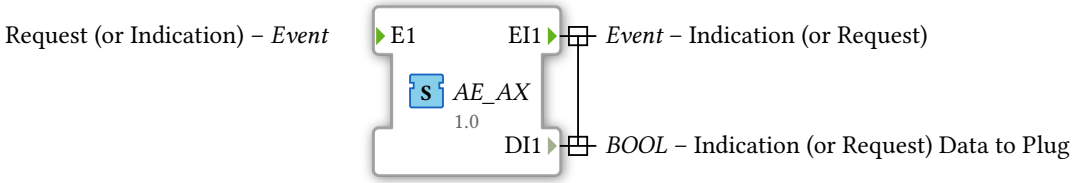
- E_TOF vs. FB_TOF: The event timer that doesn't tick cyclically – a revolution for automation systems?
- Event FBs according to IEC 61499: Building blocks of automation – how event function blocks work
- IEC 61499: Revolution of automation – event-driven FBs and distributed systems explained
- E_REND: Event Synchronization in IEC 61499
- The Unstoppable Counter: Why IEC 61499's ECTU Guarantees Safe, Event-Driven Control (and Never Overflows)
-

EVENT_BOOL

AE_AX
Plug (adapter output)



Socket (adapter input)



bidirectional adapter interface for 1 event (forward) and 1 bool (backward, AX-style)

Interface

Event Inputs

Name	Comment	With
EI1	Indication (or Request)	DI1

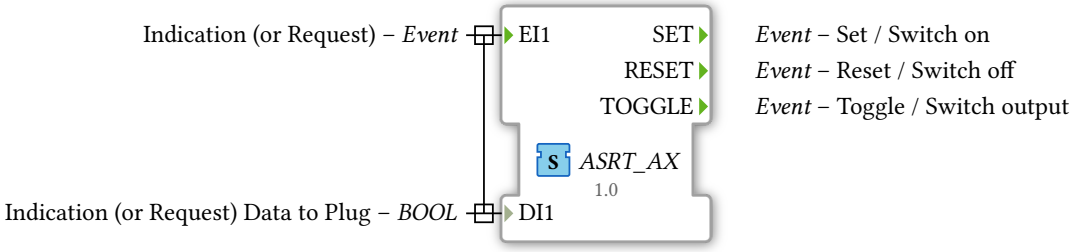
Event Outputs

Name	Comment	With
E1	Request (or Indication)	

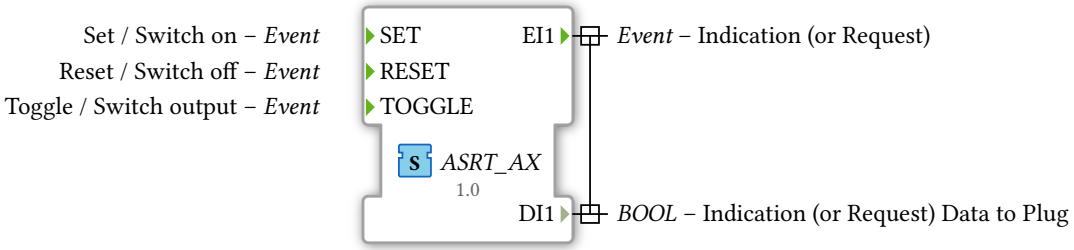
Input Vars

Name	Type	Comment
DI1	BOOL	Indication (or Request) Data to Plug

ASRT_AX
Plug (adapter output)



Socket (adapter input)



bidirectional adapter interface for 3 events (forward, Set/Reset/Toggle) and 1 bool (backward, AX-style)

Interface

Event Inputs

Name	Comment	With
EI1	Indication (or Request)	DI1

Event Outputs

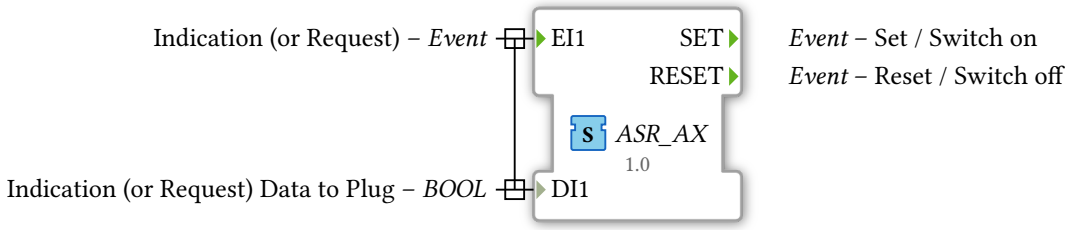
Name	Comment	With
SET	Set / Switch on	
RESET	Reset / Switch off	
TOGGLE	Toggle / Switch output	

Input Vars

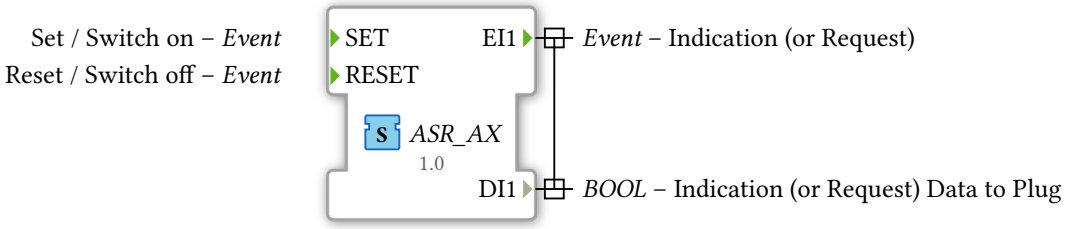
Name	Type	Comment
DI1	BOOL	Indication (or Request) Data to Plug

ASR_AX

Plug (adapter output)



Socket (adapter input)



bidirectional adapter interface for 2 events (forward, Set/Reset) and 1 bool (backward, AX-style)

Interface

Event Inputs

Name	Comment	With
EI1	Indication (or Request)	DI1

Event Outputs

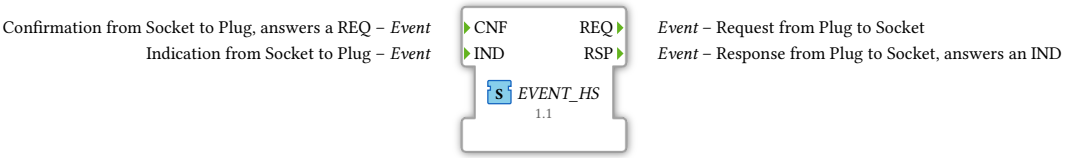
Name	Comment	With
SET	Set / Switch on	
RESET	Reset / Switch off	

Input Vars

Name	Type	Comment
DI1	BOOL	Indication (or Request) Data to Plug

Handshake

EVENT_HS
Plug (adapter output)



Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

The EVENT_HS adapter interface implements the **Handshake design pattern** from the IEC 61499 primer course (Module 6 – Design methods and patterns, category *Behavioural*). It bundles the four classical service primitives of a request/confirm/indication/response interaction into a single adapter connection, eliminating the need for four separate event connections between communication partners. This is the minimal, dataless variant of the pattern – no payload variables are declared.

The adapter distinguishes two roles: the **Plug** plays the *Requester/client* role, firing `REQ` and `RSP` events and reacting to `CNF` and `IND` events. The **Socket** plays the *Responder/server* role, reacting to `REQ` and `RSP` and firing `CNF` and `IND` events.

Interface Structure

Event Inputs

Name	Type	Comment
CNF	Event	Confirmation from Socket to Plug, answers a REQ
IND	Event	Indication from Socket to Plug

Event Outputs

Name	Type	Comment
REQ	Event	Request from Plug to Socket
RSP	Event	Response from Plug to Socket, answers an IND

Data Inputs

None – this adapter is dataless by design.

Data Outputs

None – this adapter is dataless by design.

Adapters

This is itself an adapter type. It exposes a **Socket** interface on the right side and a **Plug** interface on the left side, allowing it to be connected as a bidirectional handshake channel between two function blocks.

Functionality

The EVENT_HS adapter encapsulates two distinct service sequences:

1. **request_confirm** – A typical request cycle:
   - The Plug issues a `REQ` event, which is forwarded to the Socket.
   - The Socket processes the request and answers with a `CNF` event, which is forwarded back to the Plug.
2. **indication_response** – An unsolicited notification cycle:
   - The Socket emits an `IND` event, which is forwarded to the Plug.
   - The Plug acknowledges the indication with an `RSP` event, which is forwarded back to the Socket.

These two sequences can occur independently and asynchronously. The adapter itself does not enforce strict alternation between the two sequences; it merely provides the connection structure and the service protocol skeleton.

Technical Features

- **Dataless design** – No data variables are involved; the adapter is purely event-driven, making it lightweight and easy to integrate.
- **Role-based directionality** – The Plug and Socket roles are clearly defined: Plug = Requester, Socket = Responder. This yields a natural left-to-right layout in the FBNetwork editor.
- **Service sequence documentation** – The two service sequences (`request_confirm` and `indication_response`) are formally declared in the adapter's service model, enabling validation and traceability.
- **Compiler compatibility** – Confirmed to work with the 4diac ECC compiler. Note that the compiler only checks that `HS.<Name>` refers to a declared event on the adapter; it does *not* verify that the direction of the event makes sense at the given Socket/Plug side. Correct wiring must therefore be ensured by the developer.

State Overview

The EVENT_HS adapter does not maintain internal state. Its behaviour is defined entirely by the two independent service sequences. Each sequence moves through the following phases:

request_confirm sequence:

1. Idle – awaiting a `REQ` from the Plug.

2. Request sent – `REQ` has been forwarded to the Socket; awaiting `CNF`.
3. Confirmation received – `CNF` has been forwarded back to the Plug; sequence returns to Idle.

indication_response sequence:

1. Idle – awaiting an `IND` from the Socket.
2. Indication sent – `IND` has been forwarded to the Plug; awaiting `RSP`.
3. Response received – `RSP` has been forwarded back to the Socket; sequence returns to Idle.

The two sequences can overlap without interference, as they use disjoint event pairs.

Application Scenarios

The EVENT_HS adapter is suitable for any scenario where two function blocks need a structured, handshake-based event communication, including:

- **Client/Server coordination** – A client FB requests an operation from a server FB and waits for confirmation of completion.
- **Asynchronous notifications** – A producer FB notifies a consumer FB of an event, and the consumer acknowledges receipt.
- **Resource management** – Coordinating access to shared resources between FBs, where explicit request/confirm cycles help avoid race conditions.
- **Simplified wiring** – Replacing multiple point-to-point event connections with a single adapter connection, improving diagram readability and maintainability.

Comparison with Similar Blocks

Aspect	EVENT_HS adapter	Plain event connections	Data-carrying handshake adapters
Event bundling	Yes – 4 events in one adapter	No – separate wires per event	Yes
Payload support	None (dataless)	N/A	Yes – carries data with events
Role clarity	Explicit Plug/Socket roles	No implicit roles	Depends on implementation
Wiring complexity	Low – one connection	High – multiple wires	Low – one connection but more complex interface
Suitable for	Simple synchronization, pure event protocols	Simple one-way triggers	Data exchange with handshake

The key advantage over plain event connections is the encapsulation of the complete handshake protocol into a single, well-defined interface. Compared to data-carrying handshake adapters, EVENT_HS is minimal and focused solely on event synchronization.

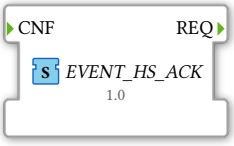
Conclusion

The EVENT_HS adapter provides a clean, dataless implementation of the IEC 61499 handshake design pattern. By defining the four core service primitives (`REQ`, `CNF`, `IND`, `RSP`) within a single adapter interface, it simplifies the wiring between communication partners and clarifies the roles of Requester and Responder. Its lightweight nature and identical behaviour to the textbook pattern make it an ideal building block for event-driven coordination in IEC 61499-based applications. Developers must, however, pay careful attention to correct Socket/Plug assignment, since the compiler does not enforce directional correctness.

EVENT_HS_ACK

Plug (adapter output)

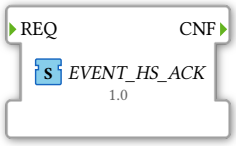
Confirmation from Socket to Plug, answers a REQ – *Event*



Event – Request from Plug to Socket

Socket (adapter input)

Request from Plug to Socket – *Event*



Event – Confirmation from Socket to Plug, answers a REQ

• – ◦ • – ◦ • – ◦ •

Introduction

The EVENT_HS_ACK adapter is a reduced member of the EVENT_HS (Handshake) design-pattern family. It implements only the request/confirm half of the full handshake vocabulary, providing the **REQ** (request) and **CNF** (confirmation) events while omitting **IND** and **RSP**. This yields a strict, one-directional-per-transaction handshake in which every request is guaranteed to receive a confirmation.

Because the adapter is dataless, it is used purely for event synchronization and control-flow coordination between the Plug and Socket sides of a connection.

Interface Structure

The adapter exposes exactly two event interfaces and no data interfaces. It defines two roles – **PLUG** (left interface) and **SOCKET** (right interface) – which determine in which direction the events travel.

Event Inputs

Event	Description
CNF	Confirmation received from the Socket, answering a previously issued REQ.

Event Outputs

Event	Description
REQ	Request issued by the Plug towards the Socket.

Data Inputs

None – the adapter carries no data.

Data Outputs

None – the adapter carries no data.

Adapters

Role	Direction	Behaviour
PLUG	Left interface	Plays the Requester/client role. It fires **REQ** and reacts to **CNF**.
SOCKET	Right interface	Plays the Responder/server role. It reacts to **REQ** and fires **CNF**.

Functionality

EVENT_HS_ACK implements a strict request/confirm handshake sequence:

1. The **Plug** issues a **REQ** event.
2. The **Socket** receives the **REQ**, performs whatever processing is required, and responds with a **CNF** event.
3. The **Plug** receives the **CNF**, which completes the transaction.

The Socket side can never push an unsolicited indication back to the Plug. Every **REQ** must be answered by exactly one **CNF**, making the pattern a reliable request/acknowledgment protocol.

Technical Features

- **Dataless** – carries no data payload; it only synchronizes events.
- **Reduced vocabulary** – only REQ and CNF; no IND or RSP.

- **Guaranteed confirmation** – every REQ is answered by a CNF, unlike unidirectional handshake variants.
- **Standard IEC 61499 compliant** – adheres to the 61499-1 identification and uses the standard service-sequence model.
- **Simple service model** – a single `request_confirm` service sequence with two transactions fully specifies the behaviour.

State Overview

The adapter defines one service sequence, `request_confirm`, consisting of two transactions:

Transaction	Input Primitive	Output Primitive	Description
1	PLUG **REQ**	SOCKET **REQ**	Plug sends a request to the Socket.
2	SOCKET **CNF**	PLUG **CNF**	Socket confirms the request back to the Plug.

The handshake is strictly alternating: the Plug initiates, the Socket confirms, and the transaction is then complete.

Application Scenarios

- **Command/acknowledgment patterns** – when a client needs to issue commands and wait for explicit confirmation that the command was executed.
- **Simple server interactions** – when the Socket side never needs to send unsolicited indications back to the Plug.
- **Control-system coordination** – useful in distributed automation where asynchronous requests must be reliably acknowledged.
- **Reduced communication overhead** – when only two event types are needed and carrying data is unnecessary.

Comparison with Similar Blocks

Block	REQ	CNF	IND	RSP	Data	Description
EVENT_HS	✓	✓	✓	✓	–	Full handshake with all four event types.
EVENT_HS_UNI	✓	–	–	–	–	One-directional handshake without any confirmation.
EVENT_HS_ACK	✓	✓	–	–	–	Request/confirm only, no unsolicited indications.
EVENT_HS_ACK_WSTRING	✓	✓	–	–	✓	Same as EVENT_HS_ACK but carries a string payload.

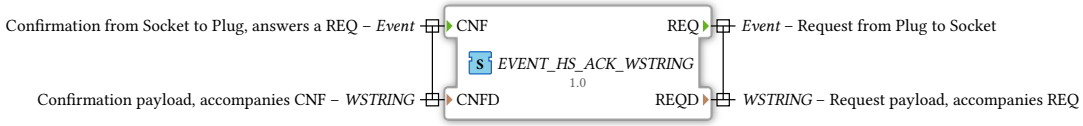
EVENT_HS_ACK differs from **EVENT_HS_UNI** in that it guarantees a confirmation for every request. It differs from the full **EVENT_HS** by removing the IND/RSP pair, which enforces a clean server/responder model where the Socket can only answer requests.

Conclusion

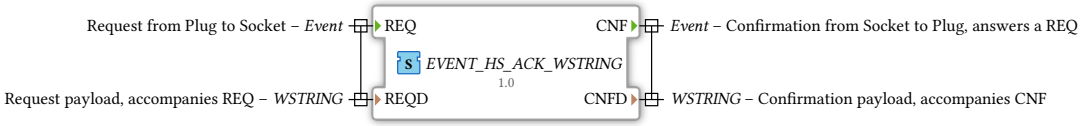
EVENT_HS_ACK is a focused, dataless handshake adapter that provides a reliable request/confirm interaction pattern. By eliminating IND and RSP, it enforces a disciplined client/server relationship where every request is answered by a confirmation. It is an excellent choice for simple command/acknowledgment flows in IEC 61499 distributed control applications, offering a reduced interface footprint while preserving the essential guarantees of a handshake.

EVENT_HS_ACK_WSTRING

Plug (adapter output)



Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

`EVENT_HS_ACK_WSTRING` is a data-carrying adapter type belonging to the Handshake (`EVENT_HS`) family defined in the IEC 61499 primer course (Module 6, Valeriy Vyatkin). It implements the request/confirm half of the full REQ/CNF/IND/RSP handshake vocabulary, augmented with `WSTRING` payloads on both events.

This adapter provides a simple **Plug → Socket request → confirmation** pattern, where the Plug (client/requester) initiates a request with an attached payload, and the Socket (server/responder) answers with a confirmation carrying its own payload. Notably, the Socket cannot initiate unsolicited traffic; no `IND`/`RSP` events exist, making this a strictly client-driven, two-message exchange.

Interface Structure

The adapter exposes two complementary roles — **Plug** (left interface, `Name>>`) and **Socket** (right interface, `>>Name`) — which must be paired by connecting a Plug of one instance to a Socket of another.

Event Inputs

Event	With	Comment
`CNF`	`CNFD`	Confirmation from Socket to Plug, answers a `REQ`

Event Outputs

Event	With	Comment
`REQ`	`REQD`	Request from Plug to Socket

Data Inputs

Data	Type	Comment
`CNFD`	`WSTRING`	Confirmation payload, accompanies `CNF`

Data Outputs

Data	Type	Comment
`REQD`	`WSTRING`	Request payload, accompanies `REQ`

Adapters

The adapter defines two interface halves that act as the connection endpoints:

- **PLUG** (`Name>>`) — the request-initiating side. It fires `REQ`/`REQD` and receives `CNF`/`CNFD`.
- **SOCKET** (`>>Name`) — the request-responding side. It receives `REQ`/`REQD` and fires `CNF`/`CNFD`.

Functionality

The adapter implements a synchronous-style request/confirm handshake:

1. The **Plug** issues a request by firing the `REQ` event, simultaneously placing a value on `REQD` (e.g., `"push,100"`).
2. The **Socket** receives the `REQ` event with `REQD` and processes the request.
3. Upon completion, the **Socket** responds by firing `CNF`, placing its own payload on `CNFD`.
4. The **Plug** receives the `CNF` event with `CNFD` and can react to the confirmation.

The entire exchange is captured in the service sequence `request_confirm`, which defines two mandatory transactions: one for the REQ/REQD transfer from Plug to Socket, and one for the CNF/CNFD transfer from Socket back to Plug.

Technical Features

- **WSTRING payloads** on both `REQ` and `CNF` events, allowing arbitrary textual data (e.g., `"name,value"` messages) to travel with each handshake step.
- **Reduced vocabulary** — only `REQ`/`CNF`; no `IND`/`RSP` events are present, preventing unsolicited Socket-initiated messages.
- **Direction-preserving roles** — the Plug always acts as requester, the Socket always as responder.
- **IEC 61499-1 compliant** — defined as an adapter type with standard event/data `With` associations.
- **Single service sequence** — the `request_confirm` sequence strictly defines the allowed interaction order.

State Overview

The adapter does not maintain an explicit internal state machine in the FB sense; the interaction is governed by the service sequence `request_confirm`. Conceptually, the exchange progresses through two states:

State	Description
Idle	No pending request. The Plug may initiate a `REQ`/`REQD`.
Waiting for Confirmation	`REQ`/`REQD` has been sent; the adapter waits for the Socket's `CNF`/`CNFD`.

After the `CNF`/`CNFD` is received, the adapter returns to Idle and is ready for the next request. The Socket side mirrors this: it waits in Idle for an incoming `REQ`, processes it, and returns to Idle after firing `CNF`.

Application Scenarios

- **Client–Server command/acknowledge** patterns, where a controller (Plug) sends a command string and expects a textual acknowledgment (e.g., `"push,100"` → `"push,100"`).
- **Configuration or parameter update** workflows requiring a paired request/confirmation with meaningful data on both sides.
- **Reduced handshake requirements** in embedded or automation systems where only a two-message exchange is needed and the server must never push unsolicited data.
- **Inter-module communication** in distributed IEC 61499 applications where a simple data-carrying request–response interaction must be explicitly typed and verified at design time.

Comparison with Similar Blocks

Adapter	Events	Payload	Notes
`EVENT_HS`	REQ, CNF, IND, RSP	None	Full four-event handshake
`EVENT_HS_ACK`	REQ, CNF	None	Reduced two-event handshake
`EVENT_HS_UNI`	REQ, CNF	None	Unidirectional variant
`EVENT_HS_UNI_WSTRING`	REQ, CNF	WSTRING on one direction	Data-carrying unidirectional variant
`EVENT_HS_ACK_WSTRING`	REQ, CNF	WSTRING on both REQ and CNF	Data-carrying reduced handshake

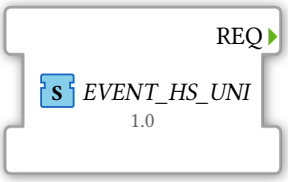
Compared to `EVENT_HS_ACK`, this adapter adds `WSTRING` payloads to both the request and the confirmation, making it suitable for message-based protocols. Compared to the full `EVENT_HS`, it drops the `IND`/`RSP` pair entirely, which simplifies the contract and eliminates the possibility of server-initiated traffic.

Conclusion

`EVENT_HS_ACK_WSTRING` is a well-defined, minimal handshake adapter for request/confirm interactions with textual payloads. Its clean separation of Plug (requester) and Socket (responder) roles, combined with its reduced event vocabulary and explicit `WSTRING` data carriage, makes it a practical building block for IEC 61499 applications that need a simple, typed, two-message client–server exchange without the overhead or ambiguity of a full four-event handshake.

EVENT_HS_UNI

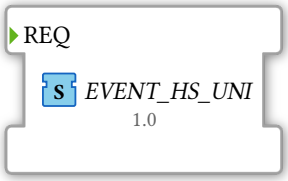
Plug (adapter output)



Event – Request/notification from Plug to Socket, no reply expected

Socket (adapter input)

Request/notification from Plug to Socket, no reply expected – *Event*



• – ◦ • – ◦ • – ◦ •

Introduction

`EVENT_HS_UNI` is a unidirectional, dataless member of the `EVENT_HS` handshake family (IEC 61499 design pattern). It declares only a single **REQ** event that travels from the Plug side to the Socket side. In contrast to the full `EVENT_HS` adapter, it omits **CNF**, **IND**, and **RSP** entirely, reducing the interaction to a pure fire-and-forget notification.

This adapter is deliberately **not a real handshake**: the Socket side has no mechanism to acknowledge or refuse the request, and the Plug side has no way to determine whether the Socket actually received or acted upon it. It should only be used in scenarios where one-way, best-effort semantics are acceptable.

Interface Structure

The adapter follows the standard Plug/Socket role split: the **Plug** side (`Name>>`) keeps the declared direction and fires **REQ** the **Socket** side (`>>Name`) mirrors it and reacts to the incoming **REQ**.

Event Inputs

None. The adapter declares no event inputs; all interaction is driven by the single output event.

Event Outputs

Name	Type	Comment
REQ	Event	Request/notification from Plug to Socket, no reply expected

Data Inputs

None. The adapter is dataless; the **REQ** event carries no associated data.

Data Outputs

None. No data is returned from the Socket side.

Adapters

The adapter provides the standard two connector roles:

- **PLUG** (left interface): initiates the interaction by emitting **REQ**.
- **SOCKET** (right interface): receives and reacts to **REQ**.

The service sequence `notify` describes a single transaction: an input primitive on the PLUG interface with event **REQ** produces an output primitive on the SOCKET interface with the same event, forwarding the notification one way.

Functionality

`EVENT_HS_UNI` implements a one-way, dataless notification channel. When the connected function block on the Plug side raises the **REQ** event, it is forwarded directly to the Socket side, where the corresponding function block reacts to it.

Because no acknowledgment path exists, the semantics are purely best-effort:

- The Plug side cannot know whether the Socket side received the event.
- The Socket side cannot confirm processing or reject the request.
- No retry, timeout, or handshake state tracking is possible at the adapter level.

The adapter is a reduction of the full `EVENT_HS` pattern, keeping only the request direction and dropping all reply mechanisms.

Technical Features

- **Unidirectional event flow**: Only **REQ** is defined; no response events exist.
- **Dataless**: The event carries no payload, making it suitable for pure control signals or notifications.
- **Best-effort semantics**: No acknowledgment, no back-pressure, no delivery guarantee.
- **Standard Plug/Socket pattern**: Compatible with the IEC 61499 connector model and standard 4diac wiring practices.
- **Typed adapter family**: Part of the `adapter::types::bidirectional::Handshake` package, alongside other reduced variants.
- **Optional service documentation**: The embedded `<Service>` block is purely informational; it does not affect event forwarding at runtime.

State Overview

The adapter has no internal state machine. It is a passive connector that forwards the **REQ** event from the Plug interface to the Socket interface. Any state handling must be implemented in the connected function blocks.

Application Scenarios

- **Fire-and-forget notifications**: Signaling a peer component that an event occurred, where no confirmation is required.
- **One-directional control pulses**: Triggering a one-shot action on the Socket side (e.g., reset, start, stop command) where the sender does not need feedback.
- **Event broadcasting / logging**: Propagating an occurrence to a downstream component without expecting a reply.
- **Reduced-resource designs**: Where the full handshake overhead of `EVENT_HS` is unnecessary and every extra event line complicates the application.

Typical use cases include simple alarm signals, watch-dog pulses, or inter-module triggers where the communication channel is inherently reliable and loss is acceptable.

Comparison with Similar Blocks

Adapter	REQ	CNF	IND	RSP	Data	Semantics
`EVENT_HS`	Yes	Yes	Yes	Yes	No	Full handshake, acknowledged request/response
`EVENT_HS_UNI`	Yes	No	No	No	No	Unidirectional fire-and-forget, no acknowledgment
`EVENT_HS_UNI_WSTRING`	Yes	No	No	No	Yes	Unidirectional with string payload, no acknowledgment
`EVENT_HS_ACK`	Yes	Yes	No	No	No	Request with acknowledgment, no independent indication
`EVENT_HS_ACK_WSTRING`	Yes	Yes	No	No	Yes	Request with acknowledgment and string payload

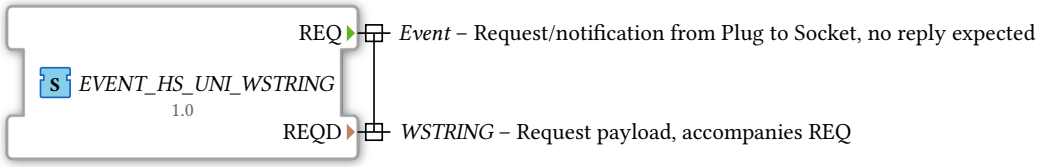
Compared to the full `EVENT_HS`, `EVENT_HS_UNI` reduces the interface to a single event and drops all handshake semantics. It is therefore not a drop-in replacement where reliable delivery or confirmation is required.

Conclusion

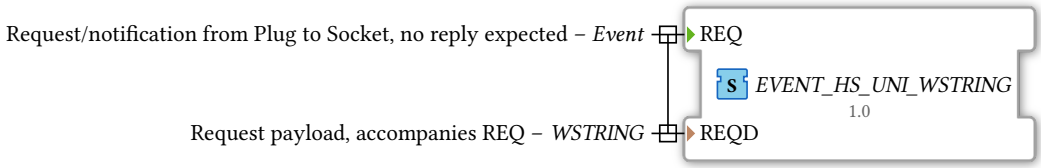
`EVENT_HS_UNI` provides a minimal, unidirectional, dataless communication channel in the `EVENT_HS` family. Its single **REQ** event enables fire-and-forget notifications between Plug and Socket with the least possible interface overhead. However, because it offers no acknowledgment or delivery guarantee, it must only be used where best-effort semantics are acceptable. For applications requiring reliable, confirmed handshakes, the full `EVENT_HS` or one of its acknowledged variants (`EVENT_HS_ACK`, `EVENT_HS_ACK_WSTRING`) is the appropriate choice.

EVENT_HS_UNI_WSTRING

Plug (adapter output)



Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

The **EVENT_HS_UNI_WSTRING** adapter is a data-carrying, reduced member of the `EVENT_HS` (Handshake) adapter family, implementing a unidirectional, fire-and-forget communication pattern. It declares a single **REQ** event paired with a **REQD** WSTRING payload that flows exclusively from the Plug to the Socket side. Unlike a true handshake, this adapter provides no acknowledgment, no indication, and no response channel whatsoever — the Socket side cannot confirm receipt or reject the request, and the Plug side has no way of verifying that the message was ever delivered or processed.

This adapter is designed for genuine one-way, best-effort notifications where the simplicity of a single event+payload pair outweighs the need for end-to-end reliability. It follows the "name,value" message style (e.g., `push,100`) commonly used in lightweight messaging systems.

Interface Structure

The adapter exposes a minimal interface with only one event and one data output on the Plug side, mirrored to the Socket side. There are no inputs on either role.

Event Inputs

None. The adapter declares no event inputs on either the Plug or Socket interface.

Event Outputs

Name	Type	With	Comment
REQ	Event	REQD	Request/notification from Plug to Socket, no reply expected

The **REQ** event is the sole trigger of the adapter. When fired on the Plug side, it is forwarded to the Socket side, carrying the current value of **REQD** as its payload.

Data Inputs

None. The adapter declares no data inputs on either role.

Data Outputs

Name	Type	Comment

REQD	WSTRING	Request payload, accompanies REQ

The **REQD** output holds the payload that is transmitted together with the **REQ** event. It is a wide string (WSTRING) that can carry arbitrary textual content, typically structured in a `name,value` format. The value is sampled at the moment the **REQ** event fires and is delivered to the Socket side.

Adapters

The adapter defines the standard Plug/Socket role split:

- **PLUG** (interface name: `Name>>`): The initiating side. It owns the declared direction and fires **REQ** carrying **REQD**.
- **SOCKET** (interface name: `>>Name`): The receiving side. It mirrors the **REQ** event and receives the **REQD** payload for reaction.

The service sequence `notify` documents the single transaction: a Plug-side `REQ` input primitive with parameter `REQD` is propagated as a Socket-side `REQ` output primitive with the same parameter.

Functionality

The adapter implements a single, atomic, unidirectional transaction:

1. The application connected to the **Plug** side sets the **REQD** data output to the desired payload value.
2. The **REQ** event is fired on the Plug side.
3. The adapter forwards the **REQ** event to the **Socket** side, passing along the current value of **REQD**.
4. The application connected to the **Socket** side receives the **REQ** event and reads the **REQD** payload.

There is no second transaction step. No `CNF` (confirmation), `IND` (indication), or `RSP` (response) events exist — neither with nor without payload. The communication is strictly one-way and fire-and-forget.

The `<Service>` block included in the adapter definition is purely documentation. Per the XSD's `minOccurs="0"`, a service sequence diagram is optional and not required for the adapter connection to forward events at runtime. It is included to make the single REQ transaction explicit and to aid designers reading the adapter documentation.

Technical Features

- **Single event/payload pair**: Exactly one event (`REQ`) and one data output (`REQD`) constitute the entire interface, minimizing connection overhead.
- **WSTRING payload**: The payload type is `WSTRING`, supporting Unicode text and arbitrary-length string data, suitable for name/value pairs, JSON snippets, or command strings.
- **Unidirectional flow**: Data and events travel exclusively from Plug to Socket. There is no reverse channel.
- **No handshake semantics**: Deliberately omits confirmation, indication, and response events. The Socket side cannot acknowledge or refuse, and the Plug side cannot detect delivery failure.
- **Zero state complexity**: With only a single transaction and no reply path, the adapter requires no internal state machine beyond event forwarding.
- **Standard Plug/Socket roles**: Consistent with the wider `EVENT_HS` family, preserving the conventional interface naming and direction conventions of IEC 61499 adapter types.

State Overview

The adapter does not maintain any internal state beyond the propagation of a single event. Its behavior can be modeled with a trivial two-state view:

- **Idle**: No transaction in progress. The adapter is ready to accept a `REQ` on the Plug side.
- **Transmitting**: The `REQ` event with the `REQD` payload is being forwarded from Plug to Socket. This state is instantaneous — the forwarding is atomic and completes within the same execution cycle.

There is no waiting, retry, or timeout state because the adapter does not expect a reply. Once the event is forwarded, the adapter returns to the Idle state. The absence of `CNF` or `RSP` events means there is no suspension of the execution resource waiting for a counterpart response.

Application Scenarios

The **EVENT_HS_UNI_WSTRING** adapter is suitable for scenarios where one-way notification with a small payload is acceptable and no acknowledgment is required:

- **Telemetry and status push**: Sensors or devices pushing measurement values (e.g., `temperature,23.5`) to a monitoring unit that does not need to confirm receipt.
- **Logging and audit trails**: Sending log entries or audit messages to a central collector where occasional loss is tolerable.
- **Command dispatch (fire-and-forget)**: Sending non-critical commands (e.g., `push,100`) where the receiving side is expected to act on them without the sender waiting for confirmation.
- **Event broadcasting**: Propagating system events (e.g., `mode,automatic`) to interested consumers in a one-to-many or one-to-one topology where reliability is not critical.
- **Integration with message-style protocols**: Matching the "name,value" textual message pattern documented in the Handshake design pattern, useful for bridging IEC 61499 applications to lightweight messaging middleware.

It should **not** be used when the sender needs to know whether the request was received or processed, when a reply is required, or when guaranteed delivery is a system requirement.

Comparison with Similar Blocks

The **EVENT_HS** family contains multiple variants tailored to different handshake depths. The following comparison highlights the key differences:

Adapter	REQ/REQD	CNF/CNFD	IND/INDD	RSP/RSPD	Payload	Use Case
EVENT_HS_UNI_WSTRING	Yes	No	No	No	WSTRING	One-way fire-and-forget with payload
EVENT_HS_UNI	Yes	No	No	No	None	One-way fire-and-forget, no payload
EVENT_HS_ACK	Yes	Yes	No	No	None	One-way with acknowledgement
EVENT_HS_ACK_WSTRING	Yes	Yes	No	No	WSTRING	One-way with acknowledgement and payload
EVENT_HS (full)	Yes	Yes	Yes	Yes	Mixed	Complete bidirectional handshake

Compared to the full `EVENT_HS`, this adapter sacrifices all reply capability for interface simplicity. Compared to `EVENT_HS_UNI`, it adds a WSTRING payload at the cost of slightly more connection wiring. Compared to the `_ACK` variants, it omits confirmation, trading reliability for minimalism.

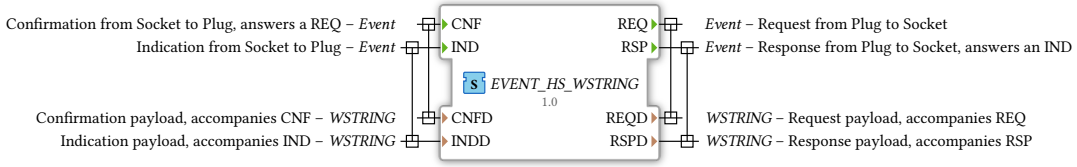
Conclusion

The **EVENT_HS_UNI_WSTRING** adapter provides a lean, purpose-built solution for unidirectional, payload-carrying notifications within the IEC 61499 adapter framework. Its single `REQ`/`REQD` event-payload pair offers a clear and maintainable interface for fire-and-forget communication, while its deliberate exclusion of handshake mechanisms keeps the design simple and avoids the complexity of error handling and timeouts.

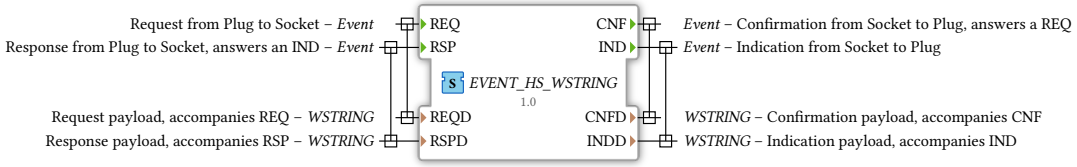
As a reduced member of the `EVENT_HS` family, it is best employed where best-effort semantics are acceptable and where the "name,value" message style aligns with the surrounding system design. For scenarios demanding acknowledgment or bidirectional interaction, the companion adapters (`EVENT_HS_ACK_WSTRING`, full `EVENT_HS`) provide richer handshake capabilities.

EVENT_HS_WSTRING

Plug (adapter output)



Socket (adapter input)



• – ○ • – ○ • – ○ •

Introduction

The **EVENT_HS_WSTRING** adapter is a data-carrying variant of the classic **Handshake design pattern** (IEC 61499 primer course, Module 6 – Design methods and patterns). It maintains the same REQ/CNF/IND/RSP event vocabulary as the dataless `EVENT_HS` adapter, but each event is accompanied by a `WSTRING` payload. This matches Vyatkin's generic *service* adapter (slide 48) used in service-oriented architectures, where messages such as `"push,100"` are exchanged.

The adapter implements a **plug/socket** role split:

- **Plug** (client/requester side) fires REQ and RSP events and reacts to CNF and IND events.
- **Socket** (server/responder side) fires CNF and IND events and reacts to REQ and RSP events.

Interface Structure

Event Inputs

Event	With Variable	Description
`CNF`	`CNFD`	Confirmation from Socket to Plug, answers a REQ. Carries a payload via `CNFD`.
`IND`	`INDD`	Unsolicited indication from Socket to Plug. Carries a payload via `INDD`.

Event Outputs

Event	With Variable	Description
`REQ`	`REQD`	Request from Plug to Socket. Carries a payload via `REQD`.
`RSP`	`RSPD`	Response from Plug to Socket, answers an IND. Carries a payload via `RSPD`.

Data Inputs

Variable	Type	Description
`CNFD`	`WSTRING`	Confirmation payload, accompanies `CNF`.
`INDD`	`WSTRING`	Indication payload, accompanies `IND`.

Data Outputs

Variable	Type	Description
`REQD`	`WSTRING`	Request payload, accompanies `REQ`.
`RSPD`	`WSTRING`	Response payload, accompanies `RSP`.

Adapters

The adapter itself acts as an interface between a Plug and a Socket. It does not contain any sub-adapters; instead, it defines the contract for bidirectional communication with WSTRING payloads.

Functionality

The adapter follows two primary interaction patterns:

1. **Request–Confirm** (`request_confirm`):
   - The Plug issues a `REQ` event with a payload (`REQD`).
   - The Socket receives the `REQ` and later responds with a `CNF` event carrying `CNFD`.
2. **Indication–Response** (`indication_response`):
   - The Socket sends an unsolicited `IND` event with payload (`INDD`).
   - The Plug acknowledges with an `RSP` event carrying `RSPD`.

Each transaction is atomic and unidirectional from the initiating side. The WSTRING payloads allow arbitrary textual data to be exchanged, making the adapter suitable for command/status or simple data transfer scenarios.

Technical Features

- **Standard compliance**: Implements the IEC 61499 service interface pattern.
- **Data type**: All payload variables are of type `WSTRING`, allowing Unicode string data.
- **Explicit event–data association**: Each event has a corresponding data variable, ensuring payload consistency.
- **Socket/Plug semantics**: Clearly defined roles for client and server sides.
- **Reusability**: Can be used as a template for typed payload adapters by replacing `WSTRING` with other types.
- **Documentation**: Detailed usage notes are embedded in the type's documentation attribute, referencing typical message formats like `"push,100"`.

State Overview

The adapter does not maintain an internal state machine; it defines a stateless protocol with two independent service sequences. However, each sequence can be considered a two-step exchange:

- **Idle** ⟶ `REQ` sent ⟶ waiting for `CNF` ⟶ `CNF` received ⟶ back to idle.
- **Idle** ⟶ `IND` received ⟶ `RSP` sent ⟶ back to idle.

These sequences can overlap or be interleaved, as the adapter does not enforce mutual exclusion.

Application Scenarios

- **Service-oriented architectures** in distributed control systems, where a client requests an operation and receives confirmation.
- **Plant/process data interfaces** where a server sends unsolicited indications (e.g., alarms) and the client acknowledges.
- **Command/response patterns**: e.g., sending `"push,100"` to a machine and receiving `"ok"` as confirmation.
- **Replacement for dataless handshake** when payload transmission is required without redesigning the event structure.

Comparison with Similar Blocks

Adapter	Payload	Use Case
`EVENT_HS`	None	Pure handshake pattern, no data transfer.
`EVENT_HS_WSTRING`	`WSTRING`	Handshake with textual payload, generic and human-readable.
Typed adapters (e.g., with `INT`/`REAL`)	specific types	When strict typing and performance are critical; require custom design.

Compared to `EVENT_HS`, this adapter adds data transfer capability while preserving the same event flow. Compared to fully typed adapters, it offers flexibility and simplicity for heterogeneous string-based data.

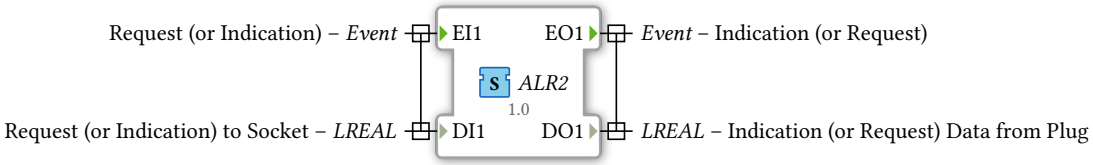
Conclusion

The `EVENT_HS_WSTRING` adapter extends the proven handshake pattern with WSTRING payloads, enabling clear, bidirectional, and data-bearing communication between plug and socket components. It is particularly useful in service-oriented industrial applications where requests, confirmations, indications, and responses need to carry textual information. Its clear separation of events and data, combined with the standard plug/socket roles, makes it a versatile building block for IEC 61499-based systems.

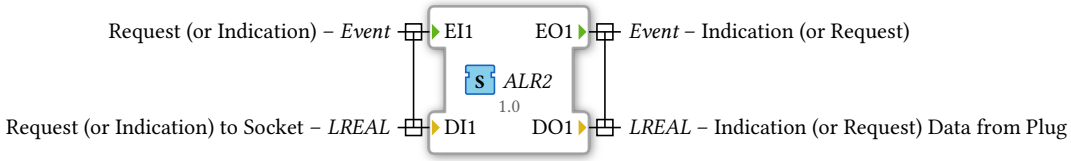
Numeric Index (bidirectional)

Here you will find all exercises from the `bidirectional` workspace, listed in ascending order. …

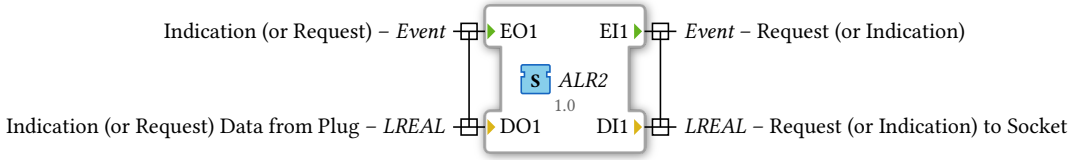
ALR2



Plug (adapter output)



Socket (adapter input)



bidirectional Adapter Interface for 1 Event and 1 LREAL

Interface

Event Inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

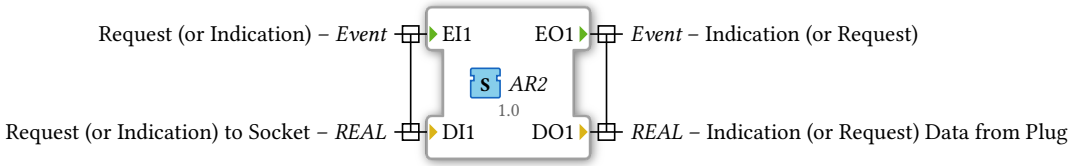
Name	Type	Comment
DI1	LREAL	Request (or Indication) to Socket

Output Vars

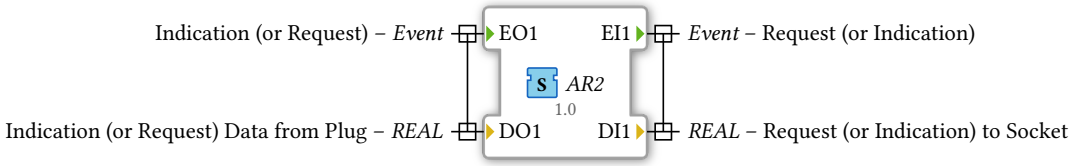
Name	Type	Comment
DO1	LREAL	Indication (or Request) Data from Plug

AR2

Plug (adapter output)



Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

The **AR2** adapter type provides a bidirectional interface between two function blocks. It transmits exactly one event along with a REAL value in both directions. The designation “AR2” stands for “Adapter Real 2-Way”. The component is designed as a generic connector (plug/socket) within the 4diac IDE and enables clean, type-safe coupling of components.

Interface Structure

Event Inputs

Name	Type	Comment	with data
EI1	Event	Request (or Indication)	DI1

Event Outputs

Name	Type	Comment	with data
EO1	Event	Indication (or Request)	DO1

Data Inputs

Name	Data Type	Comment
DI1	REAL	Request (or Indication) to Socket

Data Outputs

Name	Data Type	Comment
DO1	REAL	Indication (or Request) Data from Plug

Adapter

None defined.

Functionality

The AR2 adapter operates in an event-driven and bidirectional manner:

- When the event input **EI1** is triggered, the current REAL value at **DI1** is transmitted to the opposite end of the adapter. There, the event **EO1** appears along with the value at **DO1**.
- The same applies in the opposite direction: If the connected counterpart adapter triggers its event, this AR2 receives the event at **EI1** and makes the received value available at **DI1**.

The comments “Request (or Indication)” and “Indication (or Request)” clarify that, depending on the installation situation (plug or socket), the adapter can assume the role of a requesting or indicating channel. Therefore, AR2 is suitable for both master/slave and peer-to-peer communication.

Technical Features

- **Typed Data Transmission:** Only the **REAL** data type is supported, which provides clear semantics for physical measured values or control variables.
- **Bidirectionality:** A single adapter enables data exchange in both directions, reducing the number of required interfaces.
- **Versioning:** The module is in version 1.0, developed by HR Agrartechnik GmbH (Copyright 2026, EPL 2.0).
- **Compiler Information:** The package is located at `adapter::types::bidirectional` and uses a type hash (`eclipse4diac::core::TypeHash`) for integrity checks.
- **No State Machine:** The adapter itself does not have its own state logic; it functions purely as a connection between two event/data pairs.

State Overview

The AR2 adapter does not implement an explicit state machine. Data flows without delay or intermediate storage when events occur – the signals are passed directly from input to output.

Application Scenarios

- **Sensor Value Transmission:** A function block acquires an analog sensor value (e.g., temperature, pressure) and sends it to an evaluation block via AR2. Simultaneously, the evaluation block can return a setpoint in the opposite direction.
- **Control Coupling:** In a modular control system, AR2 connects a PID controller to an actuator. The controller sends the manipulated value (REAL) and simultaneously receives the current control deviation.
- **Parameterization:** Configuration parameters (e.g., limit values) can be exchanged between an operator panel and the control logic via AR2.

Comparison with Similar Modules

- **AR1 (Unidirectional):** An adapter that transmits data in only one direction. AR2, on the other hand, enables bidirectional exchange, which reduces cabling requirements.
- **AE2 (Adapter Event 2-Way):** Transmits two events, but no data. AR2 combines an event with a REAL value – ideal for value-based communication.
- **AR4 (Adapter Real 4-Way):** Offers four independent REAL channels. AR2 is more compact when only one value is needed.

Conclusion

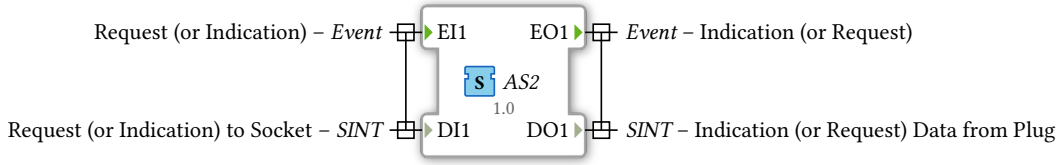
The AR2 adapter is an elegant component for the bidirectional transmission of a REAL value with its associated event. Its clear interface and ease of use make it particularly suitable for modular automation solutions where measurement data or control variables need to be exchanged between function blocks. By complying with the 61499 standard and being licensed under EPL 2.0, AR2 is open and extensible.

🌐 Related topic subpages on ms-muc-docs.de

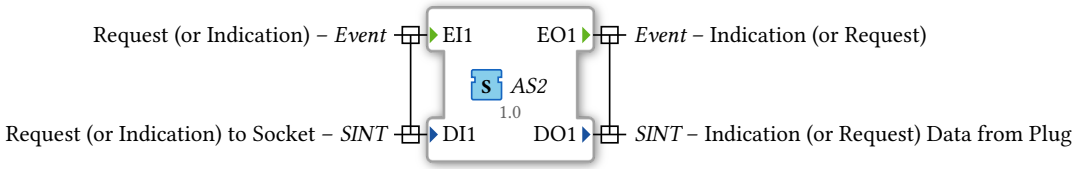
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

SINT

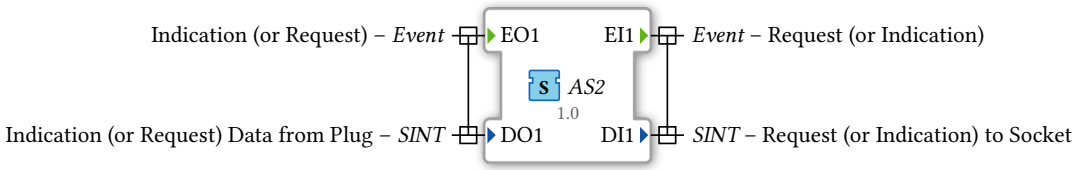
AS2



Plug (adapter output)



Socket (adapter input)



bidirectional Adapter Interface for 1 Event and 1 SINT

Interface

Event Inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

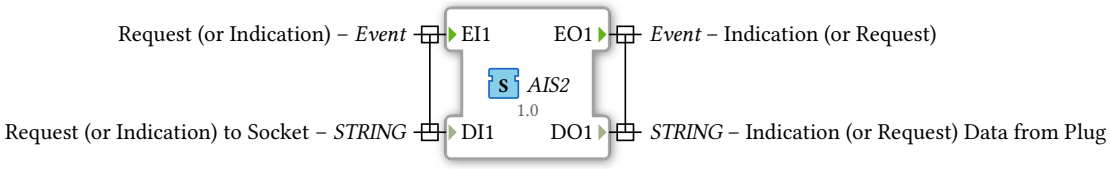
Name	Type	Comment
DI1	SINT	Request (or Indication) to Socket

Output Vars

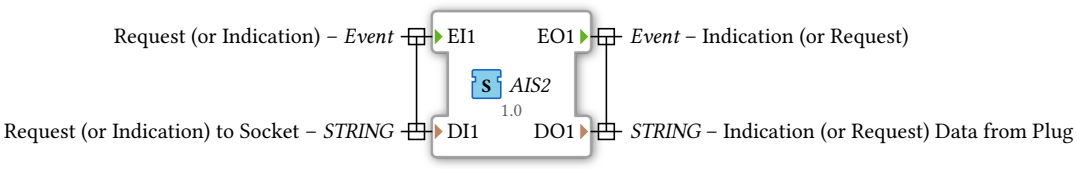
Name	Type	Comment
DO1	SINT	Indication (or Request) Data from Plug

STRING

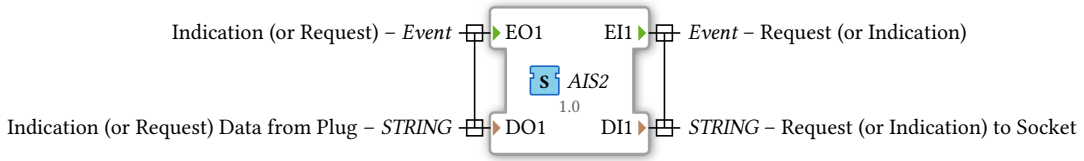
AIS2



Plug (adapter output)



Socket (adapter input)



bidirectional Adapter Interface for 1 Event and 1 STRING

Interface

Event Inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

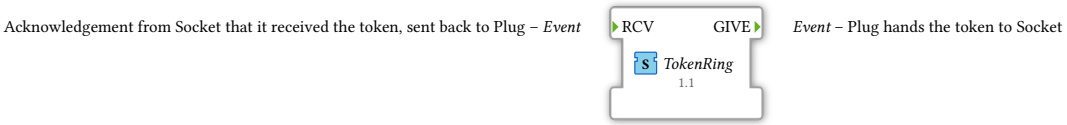
Name	Type	Comment
DI1	STRING	Request (or Indication) to Socket

Output Vars

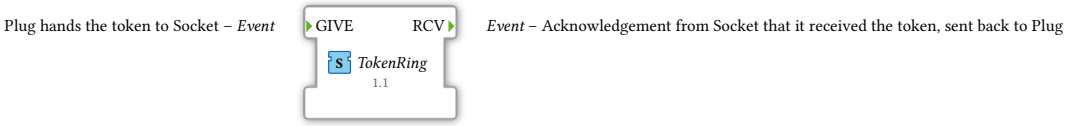
Name	Type	Comment
DO1	STRING	Indication (or Request) Data from Plug

TokenRing

TokenRing
Plug (adapter output)



Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

The **TokenRing** adapter type implements the **TokenRing / Mutual Exclusion design pattern** (IEC 61499 primer course, Module 6, V. Vyatkin, slide 15). It provides a dataless, event-based interface used to pass a mutual-exclusion "token" around a ring of controllers that share a common resource. Only the controller currently holding the token may enter its critical section; when its work is complete, it passes the token to its downstream neighbour.

The adapter defines exactly two events: **GIVE** (handing the token to the neighbour) and **RCV** (acknowledging receipt of the token). The adapter itself carries no data—the token is purely logical and conveyed by the event exchange.

Interface Structure

The TokenRing adapter exposes two event endpoints, one on each side of the adapter connection:

- **Plug** side (left, named `MTXOUT` in the reference design) plays the **giver** role: it fires `GIVE` and reacts to `RCV`.
- **Socket** side (right, named `MTXIN` in the reference design) plays the **receiver** role: it reacts to `GIVE` and fires `RCV`.

Event Inputs

Event	Type	Description
`RCV`	Event	Acknowledgement from the Socket that it received the token, sent back to the Plug.

Event Outputs

Event	Type	Description
`GIVE`	Event	The Plug hands the token to the Socket.

Data Inputs

None. The adapter is **dataless** by design—the token is represented solely by the event exchange.

Data Outputs

None. No data is carried across the adapter.

Adapters

This type is itself an adapter and does not contain nested adapters. It is instantiated as an adapter connection point inside function blocks or other adapters.

Functionality

The adapter implements a strict two-step handshake per token pass, captured in the service sequence `token_pass`:

1. **Token handover**: The Plug fires `GIVE`; the event is forwarded to the Socket, signalling that the token is being passed to the downstream neighbour.
2. **Acknowledgement**: The Socket fires `RCV`; the event is forwarded back to the Plug, confirming that the token has been received.

This handshake ensures a deterministic, lossless token transfer. A controller that participates in the ring must instantiate **two** TokenRing adapters:

- One **Plug** (`MTXOUT`) connected towards its *downstream* neighbour (the controller it passes the token to).
- One **Socket** (`MTXIN`) connected towards its *upstream* neighbour (the controller it receives the token from).

The ring is formed by wiring `MTXOUT` of each controller to `MTXIN` of the next controller in the ring, creating a closed loop.

Technical Features

- **Dataless design**: No data inputs or outputs; the token is purely logical, reducing interface complexity and eliminating data consistency issues.
- **Bidirectional event flow**: The adapter defines events travelling in both directions (Plug→Socket for `GIVE`, Socket→Plug for `RCV`), making it a true bidirectional adapter.
- **Plug/Socket role separation**: The roles are explicit and asymmetric, clearly identifying which side initiates the token pass and which side acknowledges it.
- **Standard-compliant**: Conforms to IEC 61499-1 adapter type definitions, with a formally declared service sequence (`token_pass`) that defines the valid event ordering.
- **Compact footprint**: Only two events, no data, no nested adapters—minimal resource usage in embedded controllers.

State Overview

Since the adapter is dataless, its state is expressed through the progress of the handshake:

State	Description
Idle (Token Held)	No event pending. The token is held by the local controller (or the controller is waiting for the token from upstream).
Token Being Passed	The Plug has fired `GIVE`; the token is in transit to the downstream neighbour. The adapter waits for the `RCV` acknowledgement.
Token Acknowledged	The Socket has fired `RCV`; the handshake is complete. The adapter returns to the Idle state, ready for the next token pass.

The service sequence enforces that `GIVE` must precede `RCV` in each token-pass cycle; no other event order is permitted.

Application Scenarios

- **Mutually exclusive access to a shared resource**: Multiple controllers sharing a common device (e.g., a memory bank, an I/O module, or a communication link) coordinate via the token so that only the token holder accesses the resource at any time.
- **Distributed control loops**: In agricultural machinery, industrial automation, or process control, where several PLCs or embedded controllers must serialize access to a physical actuator or sensor.
- **Ring topologies**: Closed-loop networks of controllers where deterministic, fair token rotation is required—each controller gets the token in turn, guaranteeing bounded latency for resource access.
- **Safety-critical coordination**: Where a simple, verifiable protocol is needed to prevent simultaneous access to a critical resource, the dataless handshake minimizes failure modes.

Comparison with Similar Blocks

Feature	TokenRing Adapter	Conventional Binary Semaphore / Mutex FB	Simple Event Connection
Data payload	None (pure event)	Often carries ownership/ID data	None
Acknowledgment	Explicit `RCV` handshake	Implicit release	None
Directionality	Bidirectional (Plug→Socket and Socket→Plug)	Usually unidirectional request/grant	Unidirectional
Ring topology support	Native (two instances per node)	Requires additional logic for ring wiring	Not designed for this
Failure detection	Handshake ensures delivery; no ack means no token loss	Depends on FB implementation	No delivery guarantee

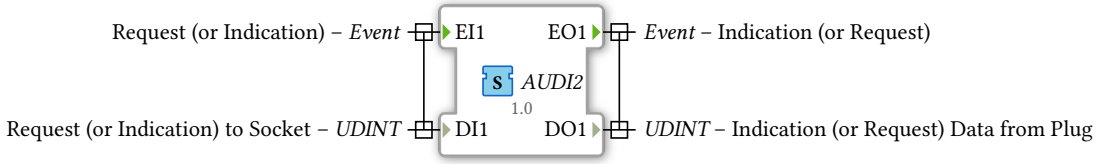
Compared to a unidirectional event connection (which merely forwards a trigger), the TokenRing adapter provides a **confirmed handshake** that guarantees the token transfer is completed only when the receiver acknowledges it. Compared to a full semaphore FB, it avoids all data overhead and protocol complexity while still providing deterministic mutual exclusion.

Conclusion

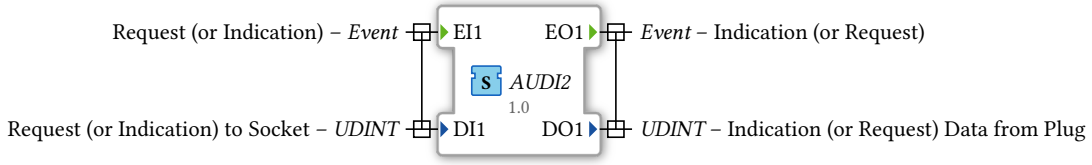
The TokenRing adapter is a minimal, robust, and standard-compliant solution for implementing mutual exclusion in ring-shaped distributed control systems. Its dataless design and strict two-event handshake make it easy to use, easy to verify, and well suited for embedded controllers where determinism and low overhead are critical. By instantiating one Plug and one Socket per controller, developers can build a closed token-passing ring that provides fair, bounded-latency access to shared resources across the entire network.

UDINT

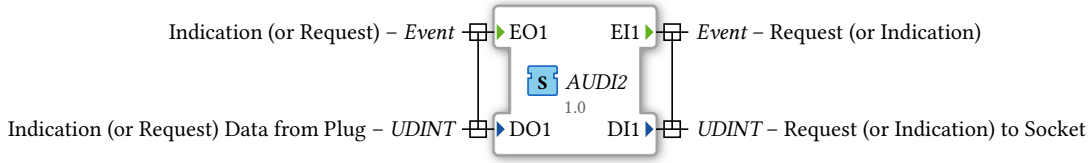
AUDI2



Plug (adapter output)



Socket (adapter input)



bidirectional Adapter Interface for 1 Event and 1 UDINT

Interface

Event Inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

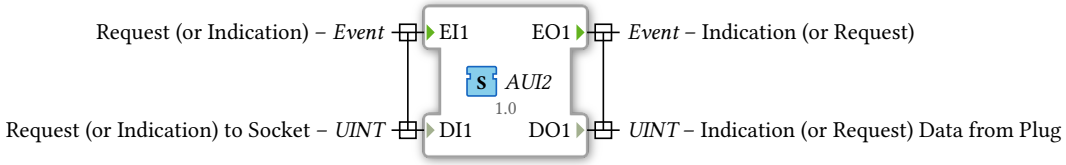
Name	Type	Comment
DI1	UDINT	Request (or Indication) to Socket

Output Vars

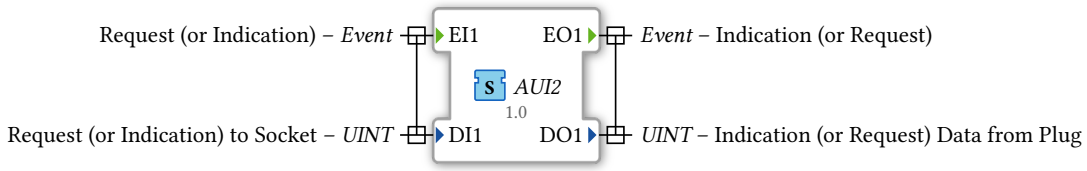
Name	Type	Comment
DO1	UDINT	Indication (or Request) Data from Plug

UINT

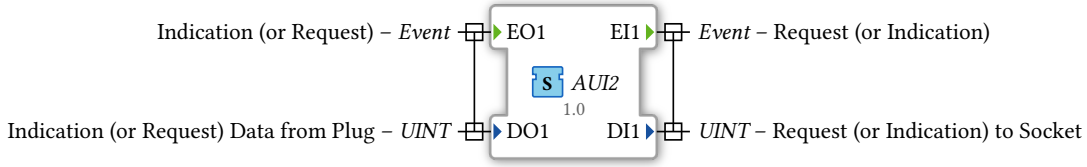
AUI2



Plug (adapter output)



Socket (adapter input)



bidirectional Adapter Interface for 1 Event and 1 UINT

Interface

Event Inputs

Name	Comment	With
EI1	Request (or Indication)	DI1

Event Outputs

Name	Comment	With
EO1	Indication (or Request)	DO1

Input Vars

Name	Type	Comment
DI1	UINT	Request (or Indication) to Socket

Output Vars

Name	Type	Comment
DO1	UINT	Indication (or Request) Data from Plug

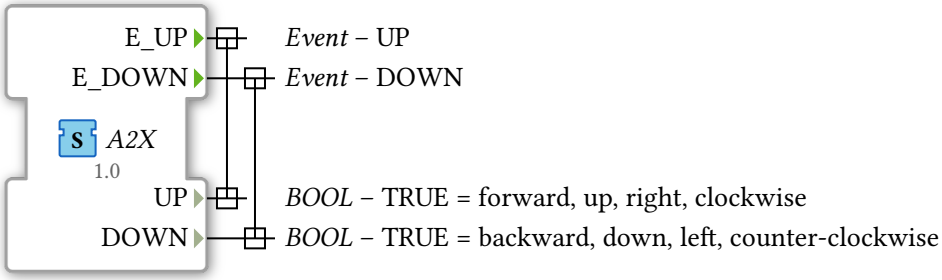
Unidirectional Adapters

Numerical Index (unidirectional)

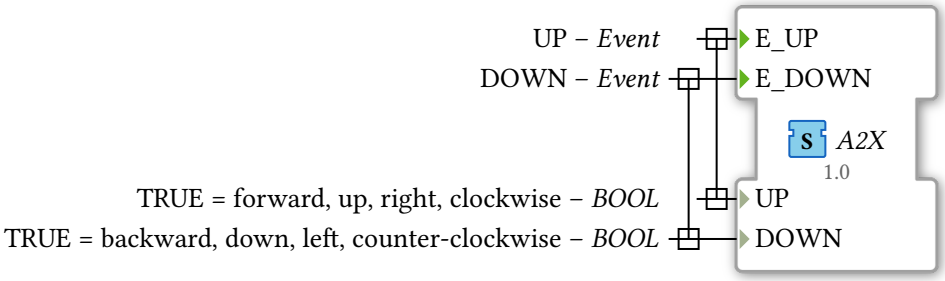
Here you will find all exercises from the `unidirectional` workspace in ascending order.

A2X (BOOL)

Plug (adapter output)



Socket (adapter input)



Unidirectional adapter interface for 2 events and 2 bools

Interface

Events

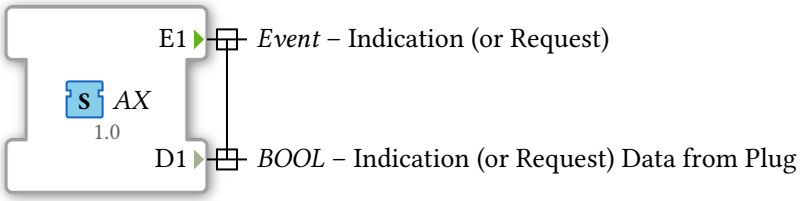
Name	Comment	With
E_UP	UP	UP
E_DOWN	DOWN	DOWN

Data

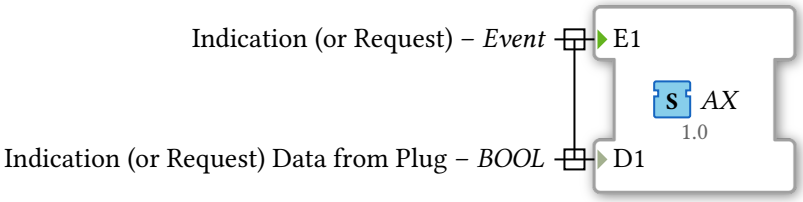
Name	Type	Comment
UP	BOOL	TRUE = forward, up, right, clockwise
DOWN	BOOL	TRUE = backward, down, left, counter-clockwise

AX (BOOL)

Plug (adapter output)



Socket (adapter input)



Podcast

- Eclipse 4diac: How Open Source Research and Practice Connect in Industrial Automation
- The Working Set Object: Understanding the Brain of the ISOBUS User Interface – From Standard to Practice in ISO Designer
- Code Renovation with AX Adapters: How Eclipse 4diac™ Conquers Complexity Through Signal Bundling
- From 1400 Errors to Clean Code: Migrating the “Grain Hoe” to Eclipse 4diac™ 3.0 and the Power of AX Adapters
- Agricultural Revolution 1883: How Max Eyth modernized England’s agriculture

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	BOOL	Indication (or Request) Data from Plug

🛠️ Related Exercises

- Uebung_003b2_AX
- Uebung_003b3_AX
- Uebung_003c_AX
- Uebung_006a3_AX
- Uebung_006a3_sub_AX
- Uebung_006a4_AX
- Uebung_010b4_AX
- Uebung_010b5_AX
- Uebung_010c2_AX
- Uebung_010c3_AX
- AuxIXA_TO_logiBUS_QXA_BG
- Uebung_010c4_AX
- Uebung_010c_AX
- AX_SPLIT_TO_2x_AX_TP
- Uebung_035a1_AX
- Uebung_035a1b_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_038_AX
- Uebung_040_AX

🌐 Related Topic Pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Unidirectional BOOL adapter

🎧 Podcast

- George Boole: How an autodidact laid the foundation of our digital world with 0 and 1

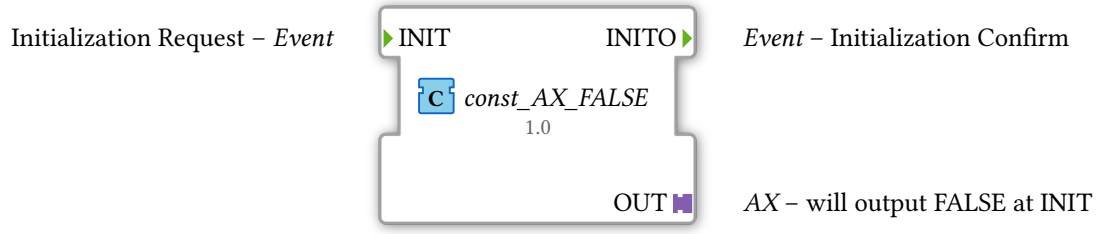
🛠️ Related exercises

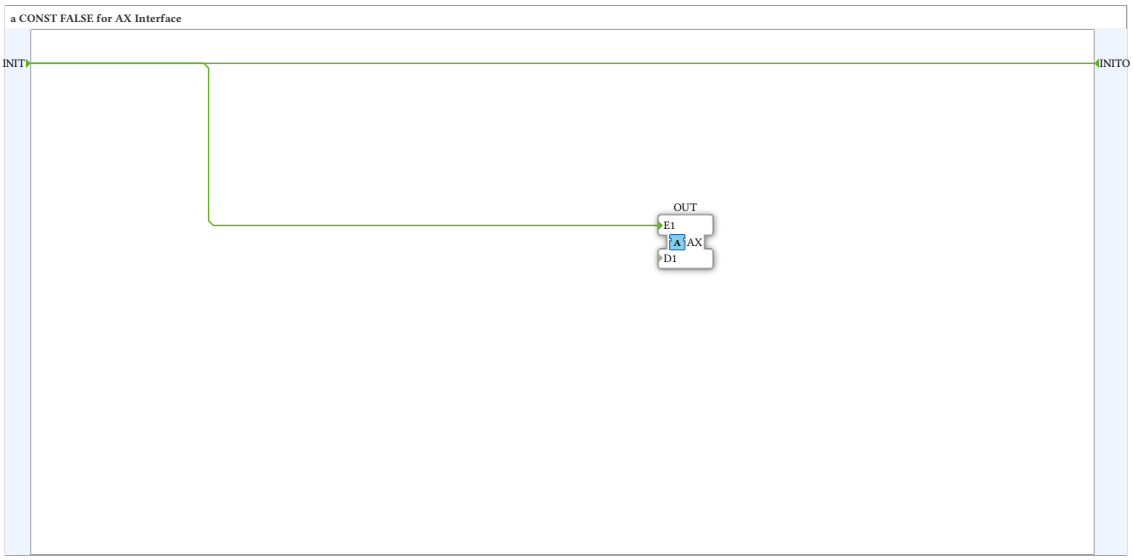
- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056

- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

constants

const_AX_FALSE





• – ○ • – ○ • – ○ •

Introduction

`const_AX_FALSE` is the counterpart to const_AX_TRUE: a simple constant block that writes the fixed value `FALSE` to an AX adapter plug on initialization. It is used to feed statically wired Boolean constants into an AX adapter network without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- **INIT**: initialization request. Triggers the constant value being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- None. The value `FALSE` is hard-wired into the block (`Parameter Name="D1" Value="FALSE"` on plug `OUT`).

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AX`): on `INIT`, outputs the fixed value `FALSE` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives, `const_AX_FALSE` triggers two internal connections: `INIT` is forwarded directly to `OUT.E1`, so the `OUT` plug fires its adapter event carrying the hard-parameterized data value `D1 = FALSE`. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The value `FALSE` is stored as a `Parameter` attribute directly on `OUT`'s `AdapterDeclaration` element, not as runtime data – the block has no way to output a different value at runtime. If a runtime-configurable start value is needed, use initval_AX instead.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances).

State Overview

The block is stateless: every `INIT` event immediately results in `FALSE` being output on `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing AX adapter connections with a fixed `FALSE` start value, e.g. as a disable constant
- Test setups where an AX input needs a fixed value without a real sensor or upstream block
- Replacing a manually wired constant connection in generic, AX-adapter-based networks

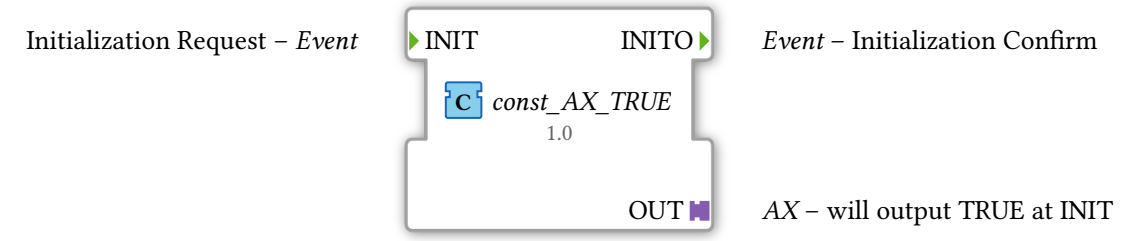
⚖️ Comparison with Similar Blocks

Compare with const_AX_TRUE, which uses the same mechanism for the value `TRUE`, and with initval_AX, which serves the same purpose but with a value configurable at runtime via `INIT_VAL` instead of a hard-wired one.

Conclusion

`const_AX_FALSE` is the simplest way to initialize an AX adapter plug with a fixed `FALSE` constant, and fits anywhere the value is known at design time and no runtime configuration is required.

const_AX_TRUE





• – ◦ • – ◦ • – ◦ •

Introduction

`const_AX_TRUE` is a simple constant block that writes the fixed value `TRUE` to an AX adapter plug on initialization. It is used to feed statically wired Boolean constants into an AX adapter network without wiring a separate event and data line for it.

Interface Structure

Event Inputs

- **INIT**: initialization request. Triggers the constant value being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- None. The value `TRUE` is hard-wired into the block (`Parameter Name="D1" Value="TRUE"` on plug `OUT`).

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AX`): on `INIT`, outputs the fixed value `TRUE` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives, `const_AX_TRUE` triggers two internal connections: `INIT` is forwarded directly to `OUT.E1`, so the `OUT` plug fires its adapter event carrying the hard-parameterized data value `D1 = TRUE`. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The value `TRUE` is stored as a `Parameter` attribute directly on `OUT`'s `AdapterDeclaration` element, not as runtime data – the block has no way to output a different value at runtime. If a runtime-configurable start value is needed, use initval_AX instead.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances).

State Overview

The block is stateless: every `INIT` event immediately results in `TRUE` being output on `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing AX adapter connections with a fixed `TRUE` start value, e.g. as an enable constant
- Test setups where an AX input needs a fixed value without a real sensor or upstream block
- Replacing a manually wired constant connection in generic, AX-adapter-based networks

⚖️ Comparison with Similar Blocks

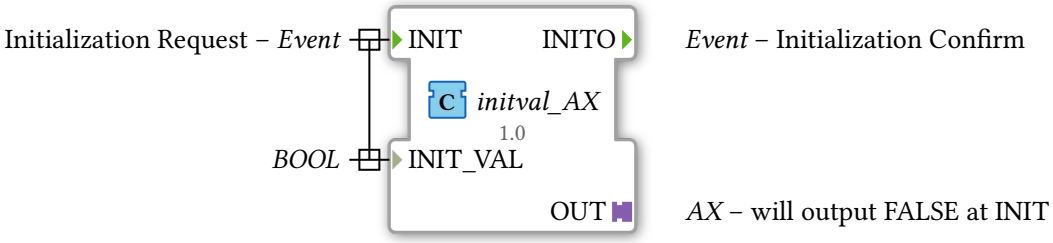
Compare with const_AX_FALSE, which uses the same mechanism for the value `FALSE`, and with initval_AX, which serves the same purpose but with a value configurable at runtime via `INIT_VAL` instead of a hard-wired one.

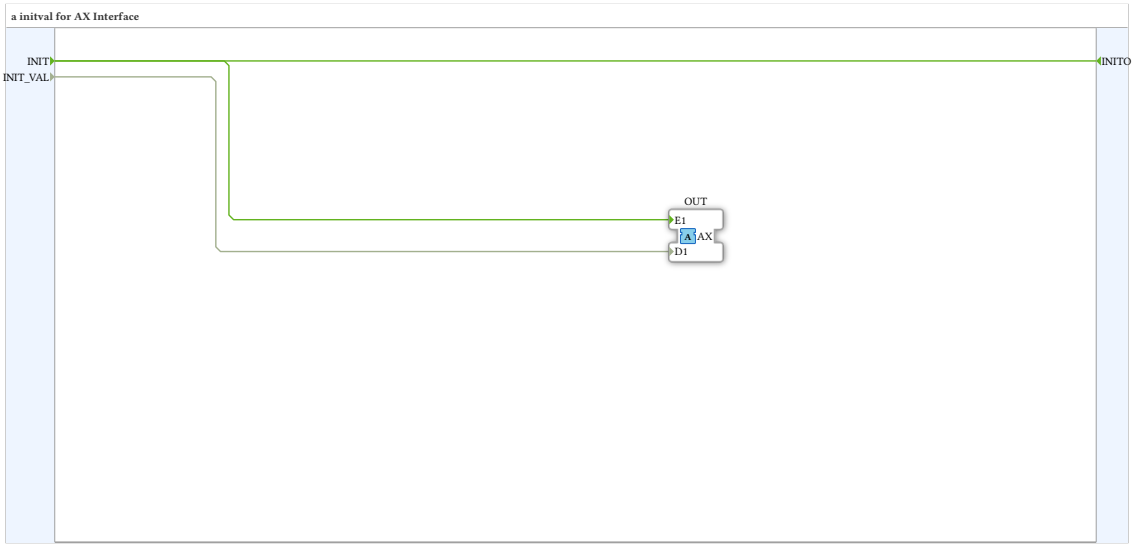
Conclusion

`const_AX_TRUE` is the simplest way to initialize an AX adapter plug with a fixed `TRUE` constant, and fits anywhere the value is known at design time and no runtime configuration is required.

initval

initval_AX





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AX` is the configurable counterpart to the const_AX_TRUE/const_AX_FALSE blocks: instead of a hard-wired value, it accepts a Boolean start value as a data input at initialization time and writes it to an AX adapter plug.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (BOOL): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AX`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AX` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- Unlike `const_AX_TRUE`/`const_AX_FALSE`, the output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set an AX value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing AX adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Dynamically setting an AX value at runtime when `const_AX_TRUE`/`const_AX_FALSE`'s hard-wired value is not sufficient
- A reusable block template for test setups where different AX start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with const_AX_TRUE and const_AX_FALSE, which use the same wiring mechanism but parameterize the output value in the block itself instead of taking it from a data input.

Conclusion

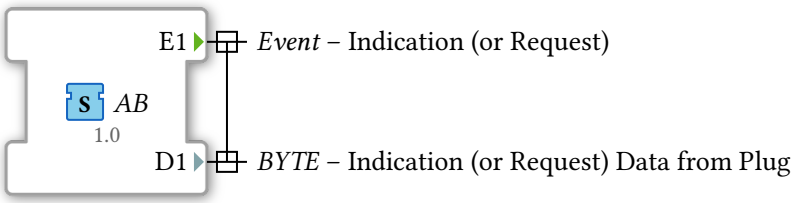
`initval_AX` provides the same simple initialization logic as `const_AX_TRUE`/`const_AX_FALSE`, but makes the output value freely configurable via `INIT_VAL`, making it the right fit wherever the start value isn't already fixed at design time.

Numerical Index (unidirectional)

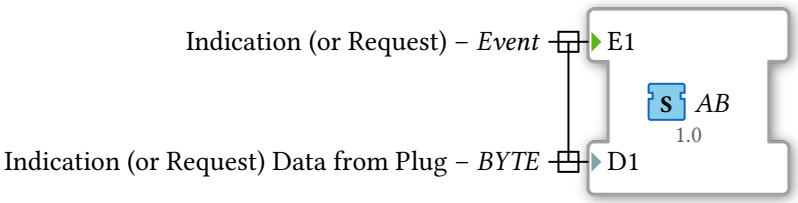
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AB (BYTE)

Plug (adapter output)



Socket (adapter input)



🎧 Podcast

- From Pyramid to Plug-and-Play: The Rise of Self-Configurable Industrial Automation
- E_CTD: Event-Driven Down Counter according to IEC 61499
- E_DELAY in IEC 61499: Precise, Cancelable Time Delay in Control Systems
- E_TON in Industrial Automation: How a Simple Timer Creates Safety and Stability
- Event Handling: The ABCs of Automation and How Systems "Think"

📺 Video

- Charles Babbage

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	BYTE	Indication (or Request) Data from Plug

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

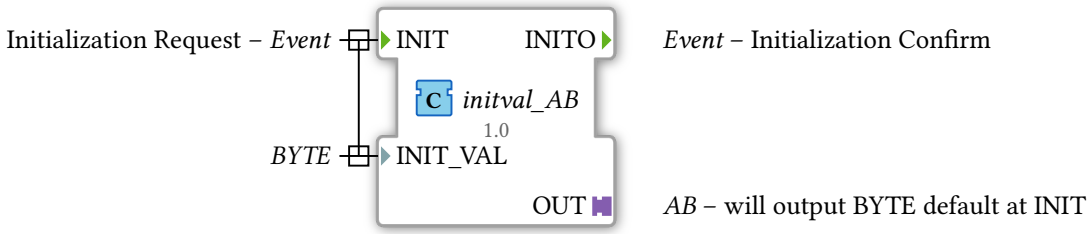
Unidirectional BYTE adapter

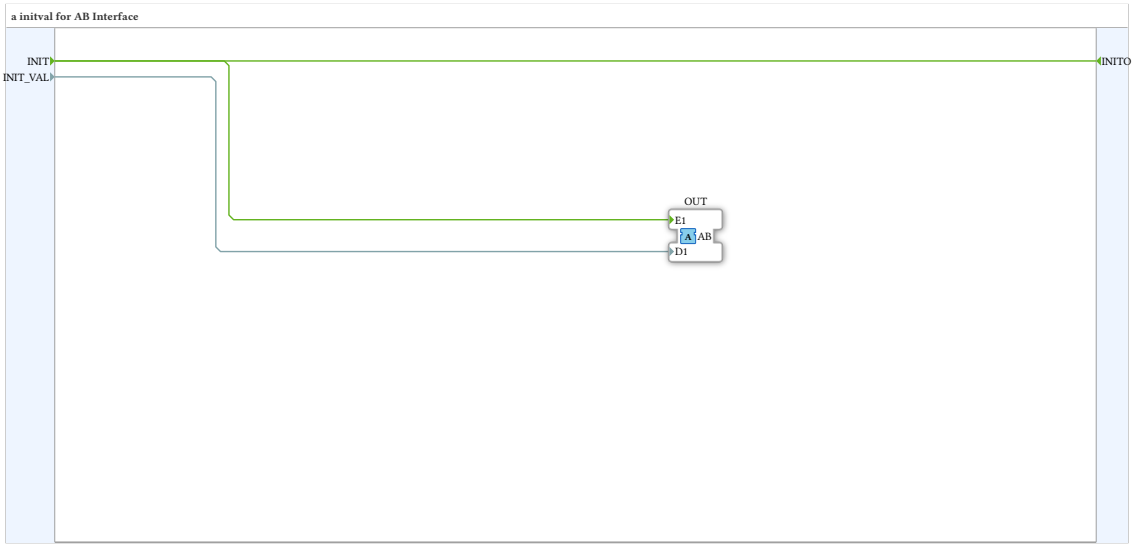
🛠️ Related exercises

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

####### Initval

initval_AB





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AB` is the configurable initialization block for the AB adapter (data type `BYTE`, 8-bit bit pattern): it accepts a start value as a data input at initialization time and writes it to a `AB` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (BYTE): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AB`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1/D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AB` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AB` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AB` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `BYTE` value is fed onto a `AB` adapter via `initval_AB` and can then be used together with the pure adapter-based `AB` comparison blocks.
- A reusable block template for test setups where different `AB` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE/const_AX_FALSE` (which only exist for `AX`), `initval_AB` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

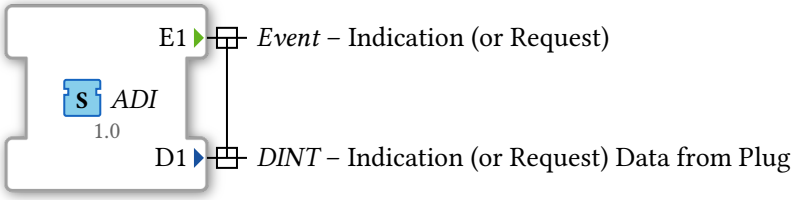
`initval_AB` provides simple, reusable initialization logic for the `AB` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `BYTE` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

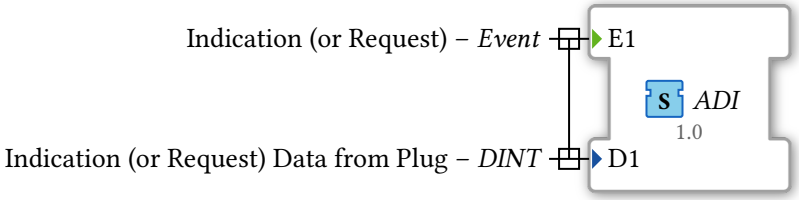
Here you will find all exercises from the `unidirectional` workspace in ascending order.

ADI (DINT)

Plug (adapter output)



Socket (adapter input)



Podcast

- DIN EN 61499-1: Function Blocks and the Paradigm Shift in Industrial Software Development
- Diesel's Radical Vision: Why the Inventor Believed All Engines of His Time to Be 'Fundamentally Wrong' – The Path to the Diesel Engine

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	DINT	Indication (or Request) Data from Plug

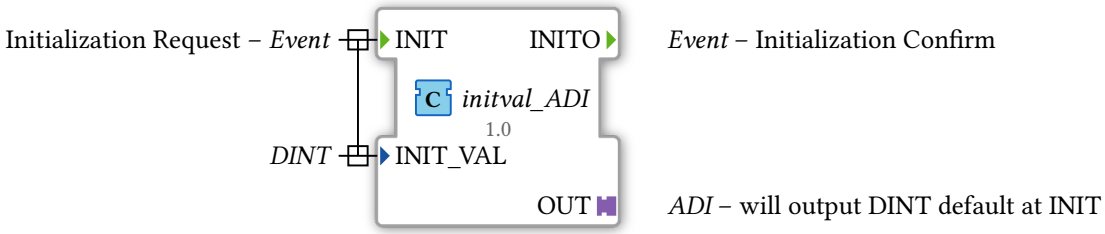
Unidirectional DINT Adapter

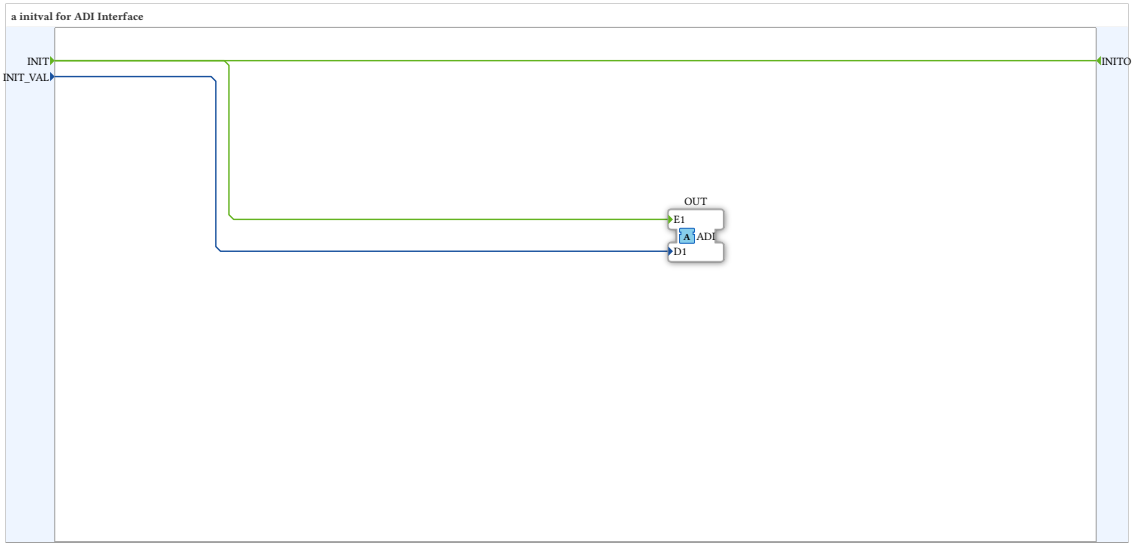
Podcast

- SINT, INT, DINT: Why the choice of data type determines efficiency and errors SINT, INT, DINT: Why the choice of data type determines efficiency and errors [SINT, INT, DINT: Why the choice of data type determines efficiency and errors ...] [SINT, INT, DINT: Why the choice of data type determines

####### Initval

initval_ADI





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_ADI` is the configurable initialization block for the ADI adapter (data type `DINT`, 32-bit signed integer): it accepts a start value as a data input at initialization time and writes it to a `ADI` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (DINT): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `ADI`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_ADI` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `ADI` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `ADI` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `DINT` value is fed onto a `ADI` adapter via `initval_ADI` and can then be used together with the pure adapter-based `ADI` comparison blocks.
- A reusable block template for test setups where different `ADI` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_ADI` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

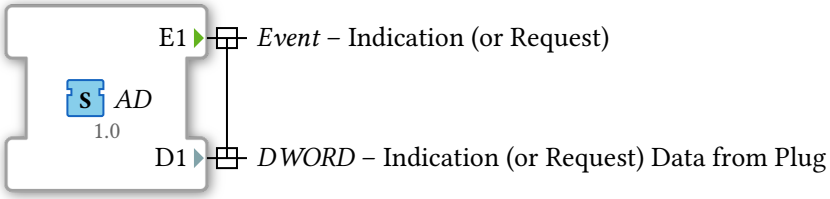
`initval_ADI` provides simple, reusable initialization logic for the `ADI` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `DINT` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

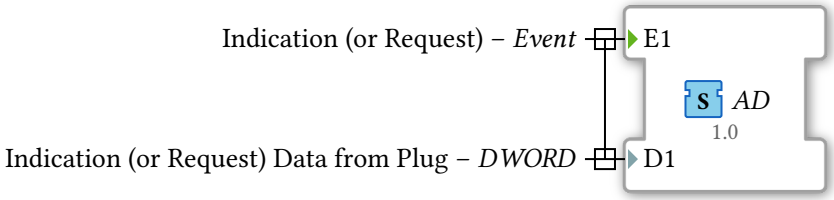
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AD (DWORD)

Plug (adapter output)



Socket (adapter input)



🎧 Podcast

- Eclipse 4diac: Your Bike for Industry 4.0 – Mastering Software Complexity and Controlling Distributed Systems
- DIN EN 61499-1: Function Blocks and the Paradigm Shift in Industrial Software Development
- Industrial Revolution Reloaded: Unpacking Plug & Produce, Data Privacy, and ETFA 2025
- Understanding Industrial Automation: PLC, SCADA, MES, and ERP Decoded – A Journey Through the Smart Factory
- PLC vs. SCADA: The Nerve Centers of Industry – Security, Efficiency, and Challenges

📺 Video

- EAGLE CAD: A Legend
- Via in Pad Plated Over

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	DWORD	Indication (or Request) Data from Plug

🌐 Related Topic Pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

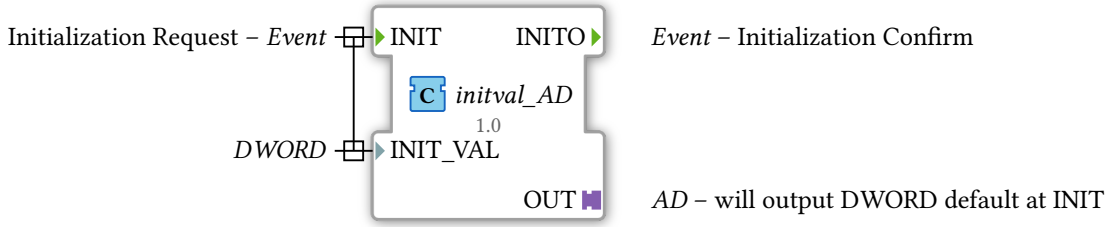
Unidirectional DWORD adapter

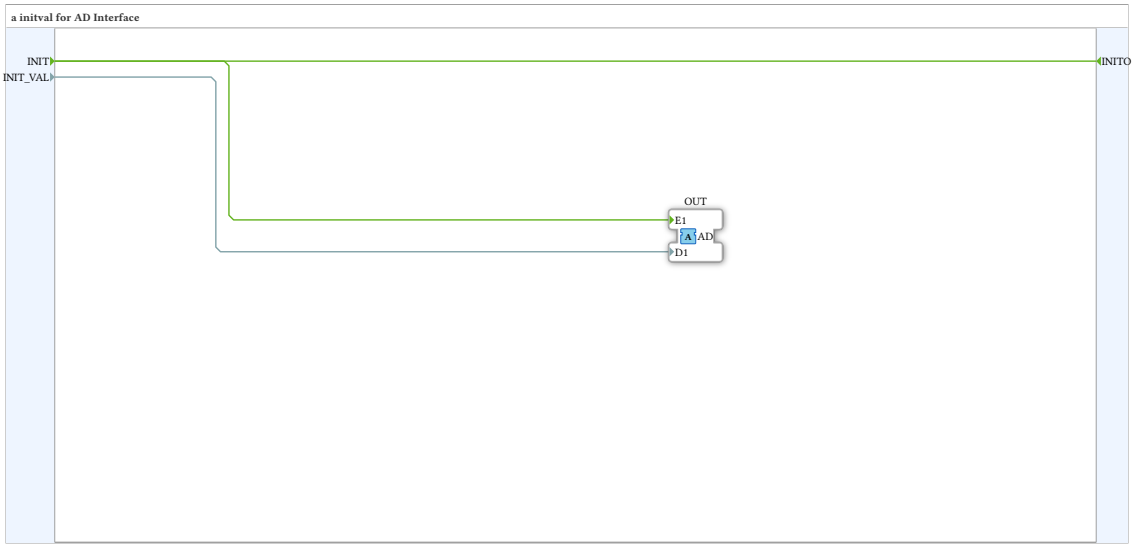
🛠️ Related exercises

- Uebung_034b
- SIN_TO_PCAN_Callback_DWord
- Uebung_152
- Uebung_153

####### Initval

initval_AD





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AD` is the configurable initialization block for the AD adapter (data type `DWORD`, 32-bit bit pattern): it accepts a start value as a data input at initialization time and writes it to a `AD` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (DWORD): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AD`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1/D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AD` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AD` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AD` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `DWORD` value is fed onto a `AD` adapter via `initval_AD` and can then be used together with the pure adapter-based `AD` comparison blocks.
- A reusable block template for test setups where different `AD` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE/const_AX_FALSE` (which only exist for `AX`), `initval_AD` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

`initval_AD` provides simple, reusable initialization logic for the `AD` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `DWORD` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

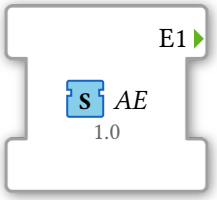
Numerical Index (unidirectional)

Here you will find all exercises from the `unidirectional` workspace in ascending order.

- AE
- ASR
- ASRT
- EVENT_Adapter_Uni

AE (EVENT)

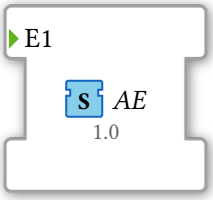
Plug (adapter output)



Event – Indication (or Request)

Socket (adapter input)

Indication (or Request) – *Event*



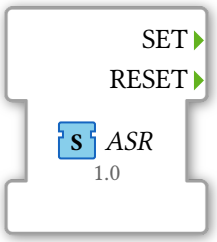
Interface

Events

Name	Comment	With
E1	Indication (or Request)	

ASR (EVENT)

Plug (adapter output)



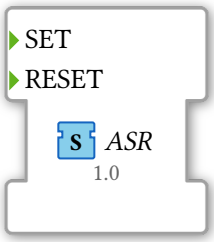
Event – Set / Switch on

Event – Reset / Switch off

Socket (adapter input)

Set / Switch on – *Event*

Reset / Switch off – *Event*



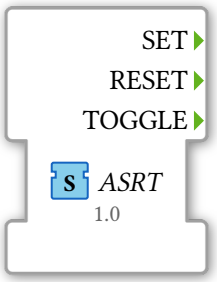
unidirectional adapter interface for 2 events

Interface

Events

Name	Comment	With
SET	Set/Switch on	
RESET	Reset/Switch off	

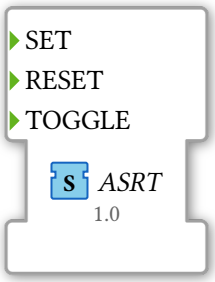
ASRT (EVENT)
Plug (adapter output)



Event – Set / Switch on
Event – Reset / Switch off
Event – Toggle / Switch output

Socket (adapter input)

Set / Switch on – *Event*
Reset / Switch off – *Event*
Toggle / Switch output – *Event*



unidirectional adapter interface for 3 events

Interface

Events

Name	Comment	With
SET	Set / Switch on	
RESET	Reset / Switch off	
TOGGLE	Toggle / Switch output	

Unidirectional Event Adapter

🎧 Podcast

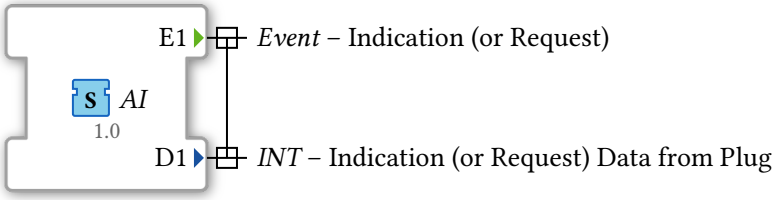
- E_TOF vs. FB_TOF: The event timer that doesn't tick cyclically – a revolution for automation systems?
- Event FBs according to IEC 61499: Lego bricks of automation – how event function blocks work
- IEC 61499: Revolution of automation – event-driven FBs and distributed systems explained
- E_REND: Event Synchronization in IEC 61499
- The Unstoppable Counter: Why IEC 61499's ECTU Guarantees Safe, Event-Driven Control (and Never Overflows)
-

Numerical Index (unidirectional)

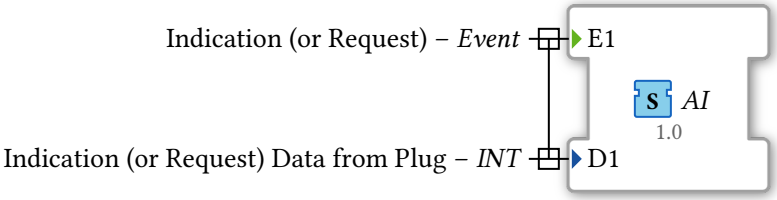
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AI (INT)

Plug (adapter output)



Socket (adapter input)



🎧 Podcast

- From Clicking Relays to Intelligent Code: How Software Revolutionized Industrial Control
- ISOBUS Containers: Dynamic Control Panels for Clear Visibility and Greater Efficiency
- Analysis of the Amendment to the Master Craftsman Examination Regulations in the Agricultural and Construction Machinery Mechatronics Trade: A Detailed Comparison of the 2024 and 2001 Regulations
- The Relay in Detail: Switching Amplifiers, Protection, and the Secrets of A1/A2, 85/86, and Hysteresis
- [The Fatal Dilemma of the Relay: Why Silver Contacts Fail with Small Signals and Gold Melts Under Load – The Burn-Off Effect] explained](https://podcasters.spotify.com/pod/show/ms-muc-lama/episodes/Das-tdliche-Dilemma-des-Relais-Warum-Silberkontakte-bei-Kleinsignalen-versagen-und-Gold-bei-Last-schmilzt–Der-Freibrenn-Effekt-erklrt-e3a9lhv)

📺 Video

- AI: The Unknown Story
- The silent failure

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	INT	Indication (or Request) Data from Plug

Unidirectional INT Adapter

🎧 Podcast

- Eclipse 4diac 3.0: ST Interpreter, FBE, and 7200 Commits – The Turbo for Distributed Automation
- From PLC to Distributed Intelligence: Understanding IEC 61499 for Industry 4.0
- Simplifying Industrial Control: Your Deep Dive into Eclipse 4diac and IEC 61499
- Events and IEC 61499: The Starting Signal for Intelligent Systems
- ERP Systems: The Digital Brain – Opportunities, Risks, and the Path to Intelligent Production

🛠️ Related Exercises

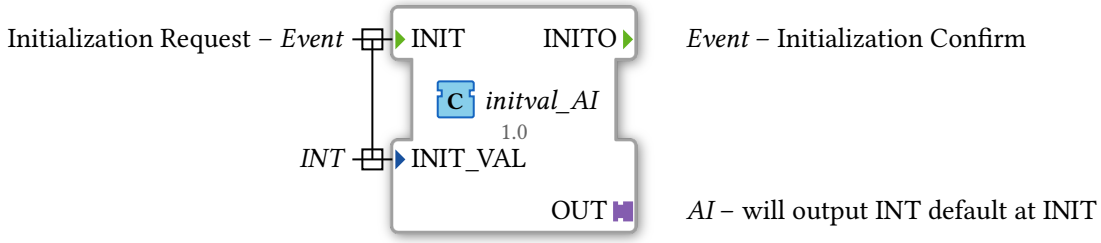
- SR_TO_QX_TRIG_EO
- Uebung_039_sub_NumbAnsicht
- Softkey_T_FF_ILOCK_TO_QX_BG
- Bool8_TO_logiBUS_QX8

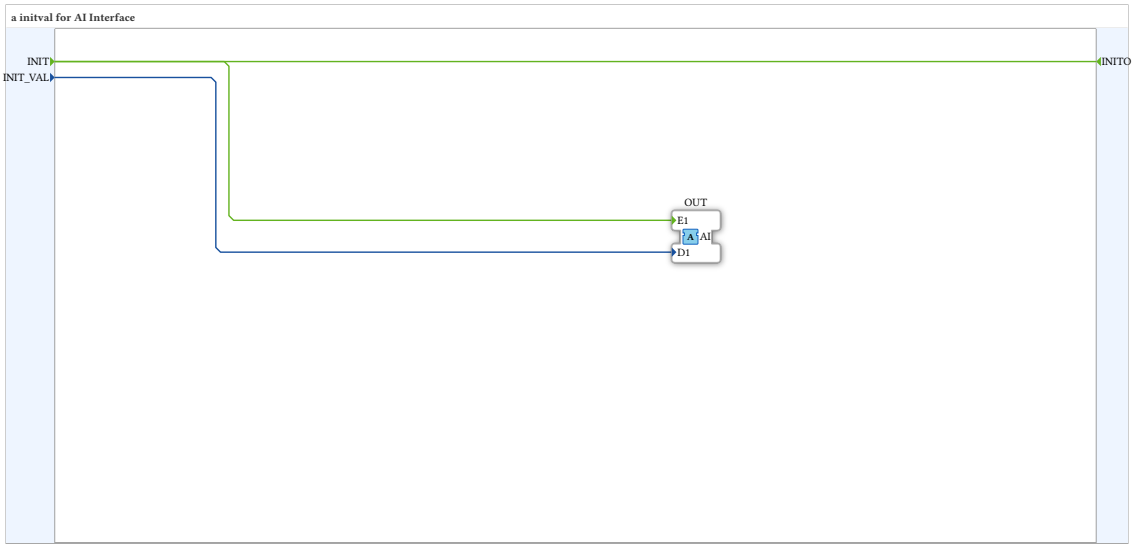
🌐 Matching topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

####### Initval

initval_AI





• – ○ • – ○ • – ○ •

Introduction

`initval_AI` is the configurable initialization block for the AI adapter (data type `INT`, 16-bit signed integer): it accepts a start value as a data input at initialization time and writes it to a `AI` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (INT): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AI`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AI` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AI` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AI` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `INT` value is fed onto a `AI` adapter via `initval_AI` and can then be used together with the pure adapter-based `AI` comparison blocks.
- A reusable block template for test setups where different `AI` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_AI` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

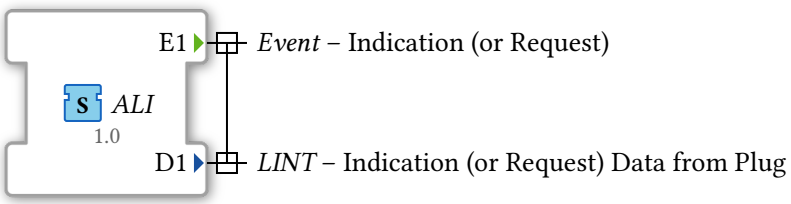
`initval_AI` provides simple, reusable initialization logic for the `AI` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `INT` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

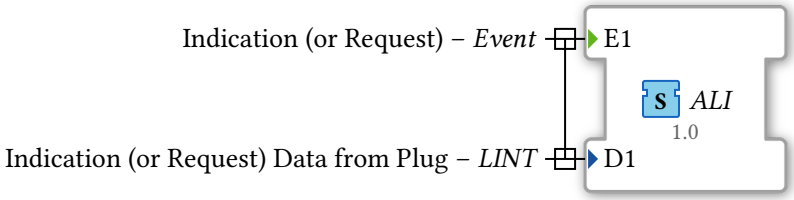
Here you will find all exercises from the `unidirectional` workspace in ascending order.

ALI (LINT)

Plug (adapter output)



Socket (adapter input)



🎧 Podcast

- The E_CTU in IEC 61499: Event-driven counting and why the minimalist solution is convincing in mechanical engineering
- IEC 61499: Factory automation in transition – From vision to industrial reality?
- R_TRIG & F_TRIG: One-time actions in the cyclic PLC world of IEC 61131-3
- ISOBUS Output Meter: Mastering dynamic displays – from pointer to visualization on the VT
- ISOBUS Scaling: When the tractor screen doesn't fit – An introduction to ISO 11783-6

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	LINT	Indication (or Request) Data from Plug

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter module on ms-muc-docs.de

]

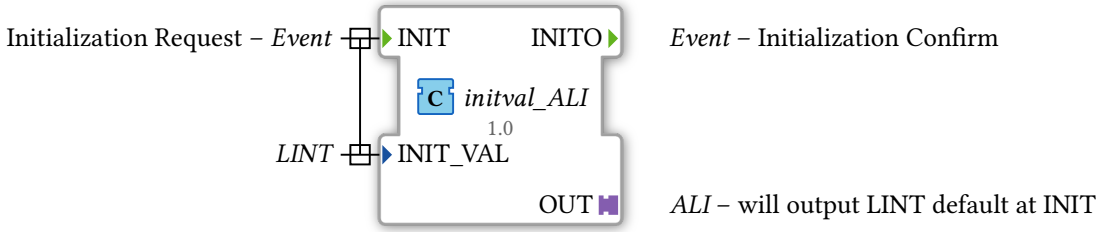
Unidirectional LINT adapter

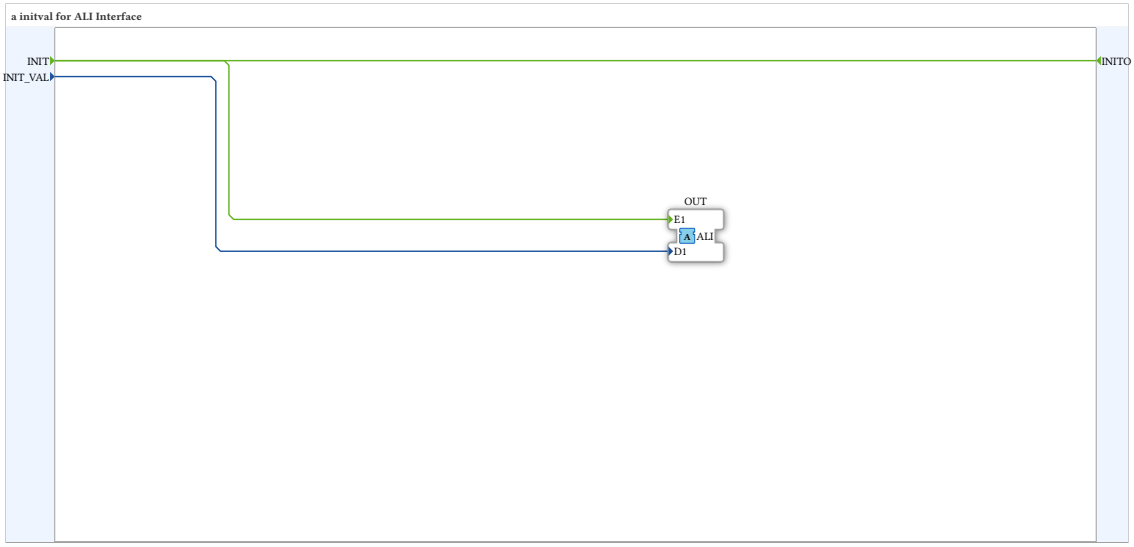
🛠️ Related exercises

- Uebung_072c

####### Initval

initval_ALI





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_ALI` is the configurable initialization block for the ALI adapter (data type `LINT`, 64-bit signed integer): it accepts a start value as a data input at initialization time and writes it to a `ALI` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (LINT): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `ALI`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_ALI` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `ALI` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `ALI` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `LINT` value is fed onto a `ALI` adapter via `initval_ALI` and can then be used together with the pure adapter-based `ALI` comparison blocks.
- A reusable block template for test setups where different `ALI` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_ALI` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

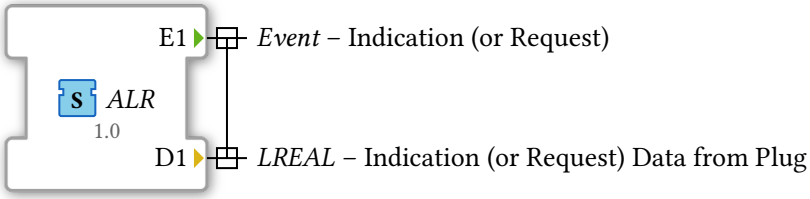
`initval_ALI` provides simple, reusable initialization logic for the `ALI` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `LINT` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

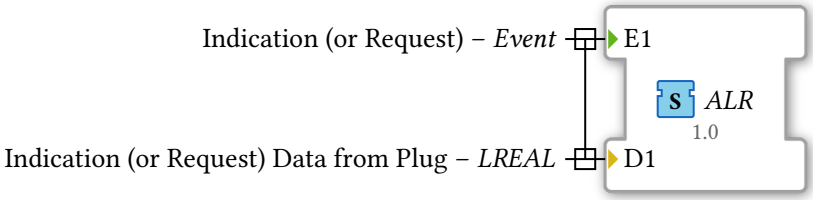
Here you will find all exercises from the `unidirectional` workspace in ascending order.

ALR (LREAL)

Plug (adapter output)



Socket (adapter input)



unidirectional Adapter Interface for 1 Event and 1 Lreal

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	LREAL	Indication (or Request) Data from Plug

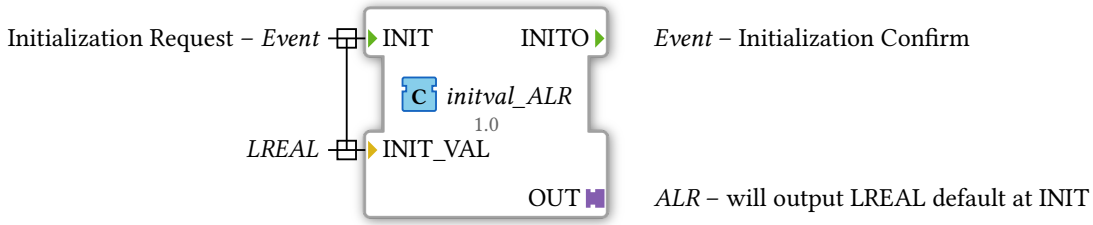
Unidirectional LREAL adapter

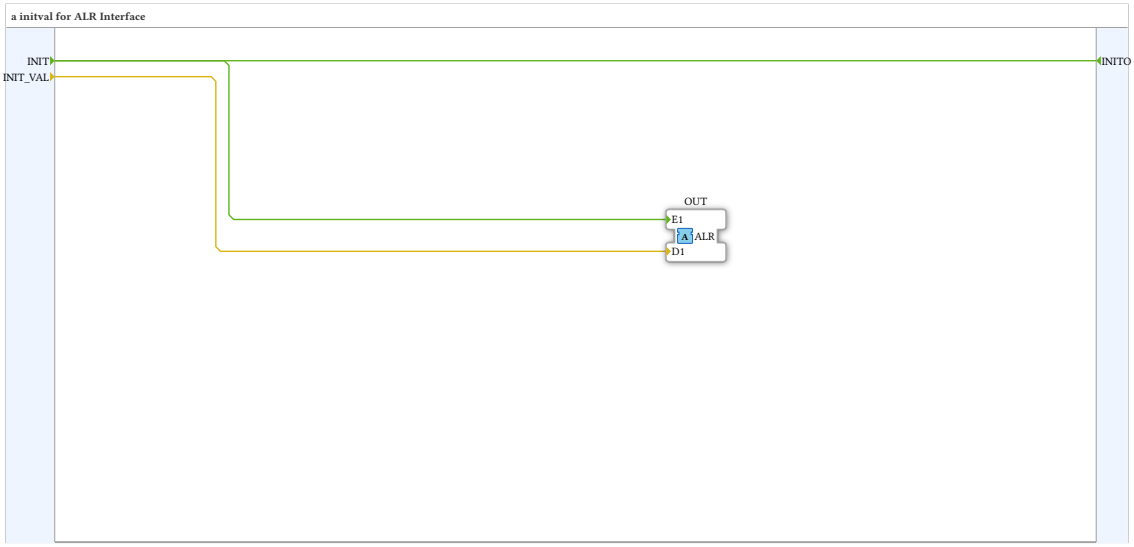
🛠️ Related exercises

- Uebung_003a0
- Uebung_003a0_AX
- Uebung_010c2
- Uebung_010c2_AX
- Uebung_035a
- Uebung_035a2
- Uebung_035a3

####### Initval

initval_ALR





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_ALR` is the configurable initialization block for the ALR adapter (data type `LREAL`, 64-bit floating-point number): it accepts a start value as a data input at initialization time and writes it to a `ALR` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (LREAL): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `ALR`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_ALR` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `ALR` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `ALR` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `LREAL` value is fed onto a `ALR` adapter via `initval_ALR` and can then be used together with the pure adapter-based `ALR` comparison blocks.
- A reusable block template for test setups where different `ALR` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_ALR` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

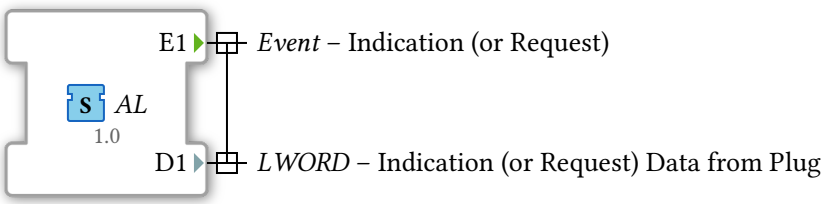
`initval_ALR` provides simple, reusable initialization logic for the `ALR` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `LREAL` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

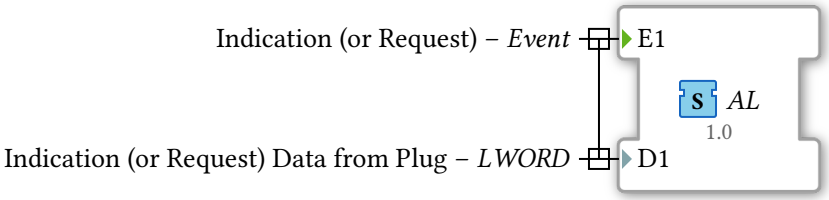
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AL (LWORD)

Plug (adapter output)



Socket (adapter input)



🎧 Podcast

- 4diac IDE: Your “Hello World” of Automation – The Blinking Tutorial Local
- 4diac Presentation: Target Groups, Structure, and Unique Selling Proposition Sharpening
- Eclipse 4diac: Open Source as a Game Changer for Industrial Controls?
- Eclipse 4diac and FORTE: Unlocking Next-Gen Industrial Automation with Open-Source Power
- Eclipse 4diac: Revolutionizing Industrial Control with Open-Source Cyber-Physical Systems

📺 Video

- The Catalog of 1863
- The Smart Switch](https://www.youtube.com/watch?v=7Q1_v_LenFs)
- The Unstoppable Chip](https://www.youtube.com/watch?v=ajRNO-jSVSo)
- Digital Decisions](https://www.youtube.com/watch?v=CivXY9CoJoM)
- Ideal Diodes](https://www.youtube.com/watch?v=cPYHaOczu6s)

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	LWORD	Indication (or Request) Data from Plug

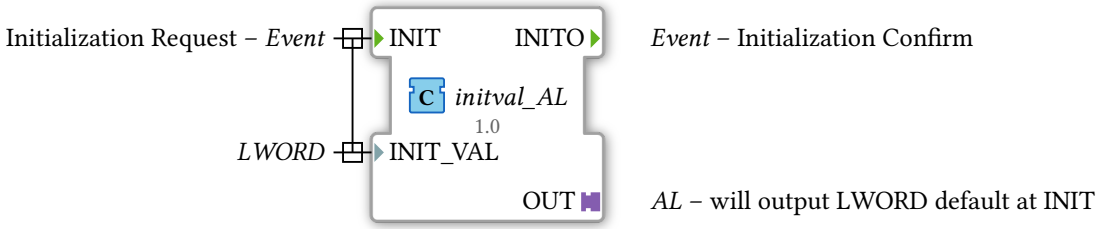
🌐 Related topic subpages on ms-muc-docs.de

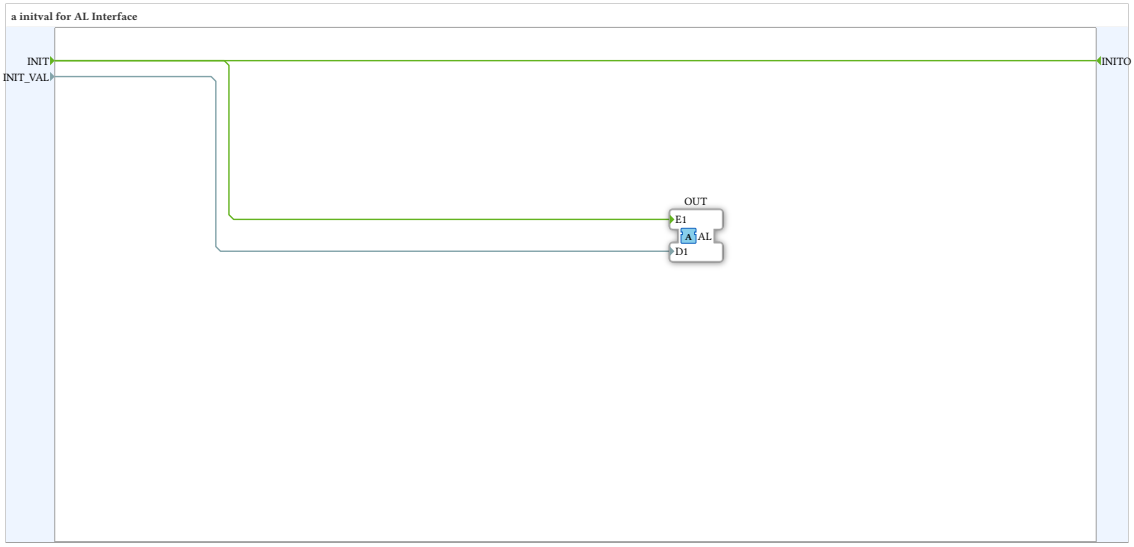
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Diode & Semiconductor Basics on ms-muc-docs.de
- 🌐 Smart Switch vs. Relay on ms-muc-docs.de

Unidirectional LWORD adapter

####### Initval

initval_AL





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AL` is the configurable initialization block for the AL adapter (data type `LWORD`, 64-bit bit pattern): it accepts a start value as a data input at initialization time and writes it to a `AL` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (LWORD): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AL`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AL` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AL` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AL` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `LWORD` value is fed onto a `AL` adapter via `initval_AL` and can then be used together with the pure adapter-based `AL` comparison blocks.
- A reusable block template for test setups where different `AL` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_AL` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

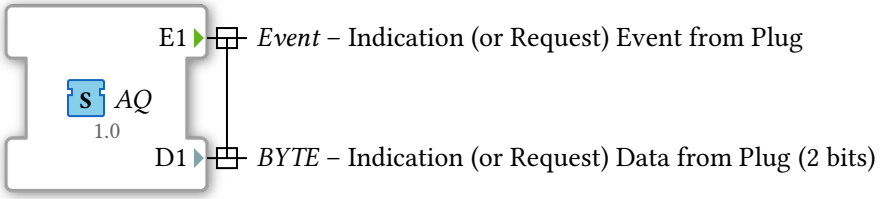
`initval_AL` provides simple, reusable initialization logic for the `AL` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `LWORD` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numeric Index (unidirectional)

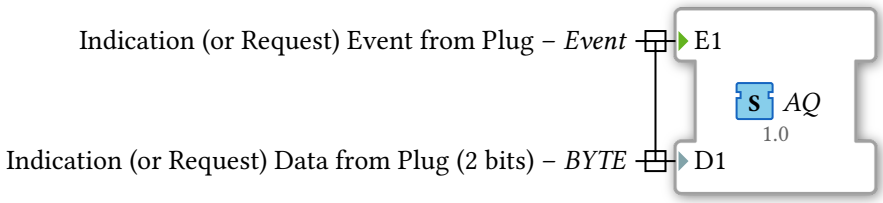
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AQ

Plug (adapter output)



Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

The `AQ` adapter provides a unidirectional interface (plug side) for transmitting an event and a byte value (used for 2 bits). It serves as a standard definition for communication between function blocks according to IEC 61499-1 and enables simple, directed data transfer.

Interface Structure

Event Inputs

None.

Event Outputs

Name	Type	Comment

E1	Event	Indication (or Request) Event from Plug (triggered together with D1)

Data Inputs

None.

Data Outputs

Name	Type	Comment
D1	BYTE	Indication (or Request) Data from Plug (2 bits used)

Adapter

None.

Functionality

The `AQ` adapter defines a unidirectional connection. A connected function block can signal via the `E1` event that the byte value at the `D1` data output is valid. The data value is a byte, of which the lower 2 bits carry the actual information. The remaining 6 bits are not defined for the application. The adapter itself contains no processing logic – it solely represents the interface specification.

Technical Features

- **Unidirectional**: Only event and data outputs are present; no inputs. Suitable for plug pages in adapter pairings.
- **Data Usage**: Of the 8 bits of the byte value, only the lower 2 bits are considered significant. The remaining bits should be set to `0` or ignored by the sending function block.
- **Event + Data**: The event `E1` is linked to the data output `D1` (`With` relation). Both are sent together.
- **Compact**: Minimal interface with one event and one byte.

State Overview

The adapter itself has no internal state. The sending function block controls the outputs based on its own state machine. The adapter simply defines the protocol: Events and data are transmitted together.

Application Scenarios

- Transmission of 2-bit status information (e.g., operating state “On/Off/Fault” or two separate flags).
- Simple command transmission with encoded bytes (e.g., 2-bit control words in automation systems).
- As part of a higher-level adapter pairing for minimal, directed communication between function blocks.

Comparison with Similar Function Blocks

Feature	AQ (this adapter)	Other adapters (e.g., with multiple events)
Events	1 output	Often multiple inputs/outputs
Data	1 byte (2 bits used)	Frequently multiple variables or complex types
Alignment	Unidirectional (Plug-side)	Bidirectional or pure socket-side
Complexity	Very low	Medium to high

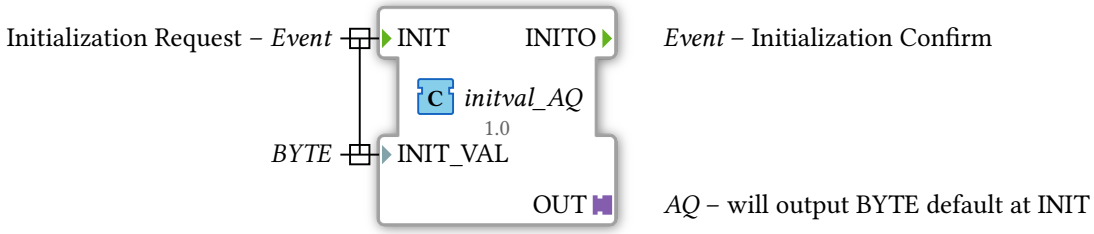
The `AQ` adapter is specifically optimized for the minimal case of “one event + one byte”.

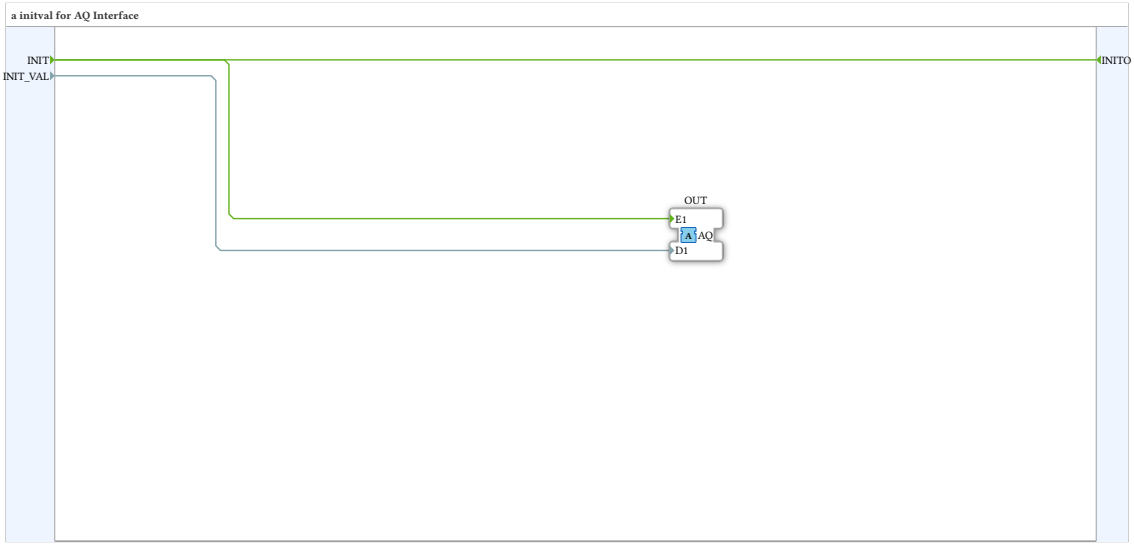
Conclusion

The `AQ` adapter offers a streamlined and standardized interface for the unidirectional transmission of an event and a 2-bit signal. Its simplicity makes it ideal for applications requiring reduced communication while maintaining IEC 61499 compatibility. The adapter is defined as a plug-side and expects a corresponding socket counterpart.

####### Initval

initval_AQ





• – ○ • – ○ • – ○ •

Introduction

`initval_AQ` is the configurable initialization block for the AQ adapter (data type `BYTE`, 2-bit value (transmitted as a byte)): it accepts a start value as a data input at initialization time and writes it to a `AQ` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (BYTE): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AQ`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1/D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AQ` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AQ` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AQ` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `BYTE` value is fed onto a `AQ` adapter via `initval_AQ` and can then be used together with the pure adapter-based `AQ` comparison blocks.
- A reusable block template for test setups where different `AQ` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_AQ` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

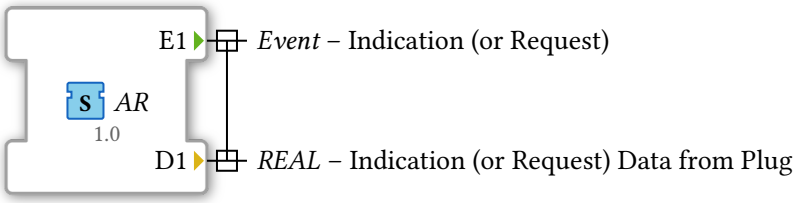
`initval_AQ` provides simple, reusable initialization logic for the `AQ` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `BYTE` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

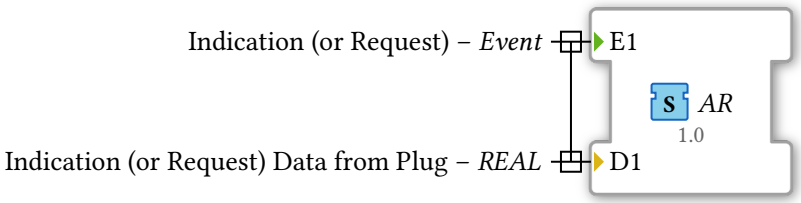
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AR (REAL)

Plug (adapter output)



Socket (adapter input)



🎧 Podcast

- From "Get Away with Electronics" to Essential: How Agricultural Tech Earned Trust and Delivered Real Relief
- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- Taming the Software Dragon: Industrial Automation and the Future of Production

- Eclipse 4diac: Your Bike for Industry 4.0 – Mastering Software Overhead and Controlling Distributed Systems
- Taming the Software Dragon: Why IEC 61499 Could Revolutionize Industry

📺 video

- Charles Babbage
- The smart switch
- The high-tech harvest
- From chaos to clarity

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	REAL	Indication (or Request) Data from Plug

🌐 Related Topic Pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Unidirectional REAL adapter

🎧 Podcast

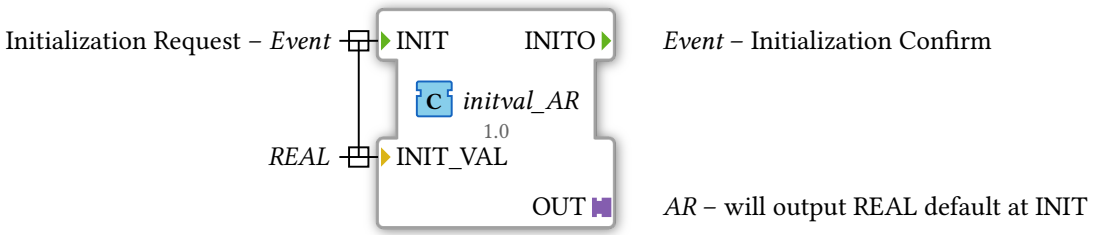
- IEC 61499: Factory automation in transition - from vision to industrial reality?
- From “Get Away with Electronics” to Essential: How Agricultural Tech Earned Trust and Delivered Real Relief

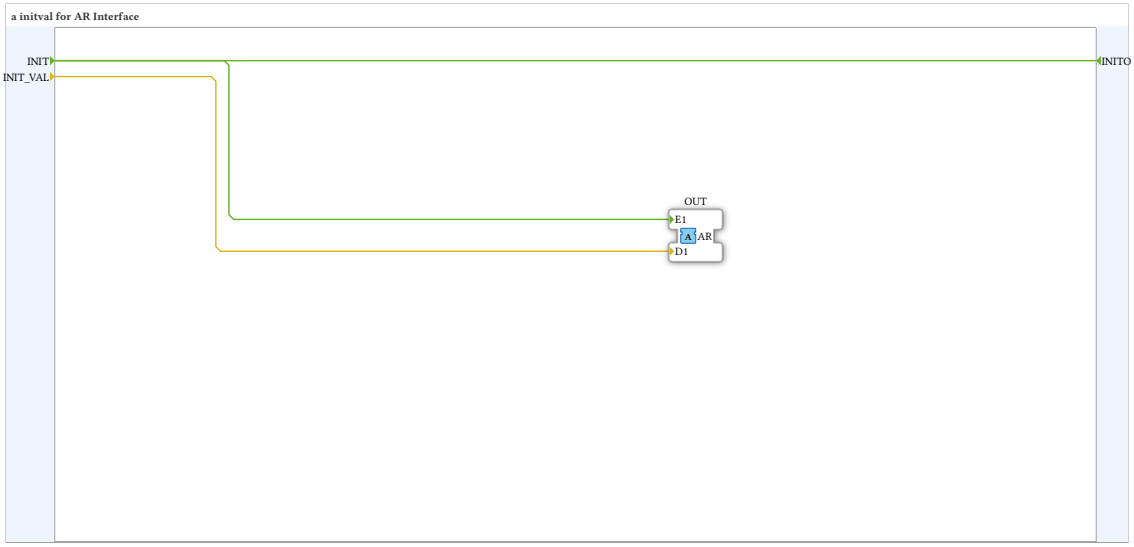
🛠️ Related exercises

- Uebung_072c
- Uebung_151_AX
- Uebung_152
- Uebung_153

####### Initval

initval_AR





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AR` is the configurable initialization block for the AR adapter (data type `REAL`, 32-bit floating-point number): it accepts a start value as a data input at initialization time and writes it to a `AR` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (REAL): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AR`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AR` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AR` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AR` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `REAL` value is fed onto a `AR` adapter via `initval_AR` and can then be used together with the pure adapter-based `AR` comparison blocks.
- A reusable block template for test setups where different `AR` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_AR` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

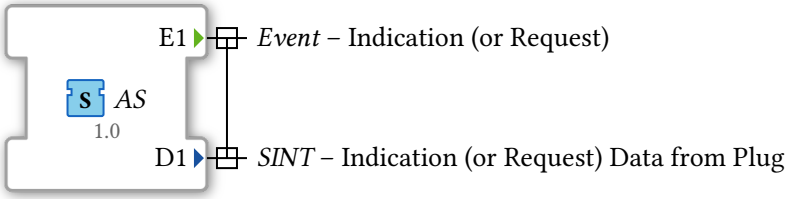
`initval_AR` provides simple, reusable initialization logic for the `AR` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `REAL` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

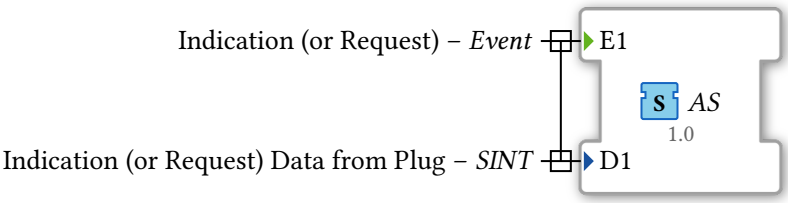
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AS (SINT)

Plug (adapter output)



Socket (adapter input)



🎧 Podcast

- 4diac IDE: Your "Hello World" of Automation – The Blinking Tutorial Local
- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- Eclipse 4diac FORTE: Understanding IEC 61499 – The LEGO® Building Kit for Your Industry 4.0 Control System
- Eclipse 4diac: Innovation Engine Research vs. User Needs – What Really Drives Development?
- Automation 4.0: Why Software Is Overtaking Hardware and What That Means for Your Skills means

📺 Video

- The ESP32-S3-DevKitC-1
- Ohm's Law
- Decoding the datasheet

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	SINT	Indication (or Request) Data from Plug

🌐 Related Topic Pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

Unidirectional SINT Adapter

🎧 Podcast

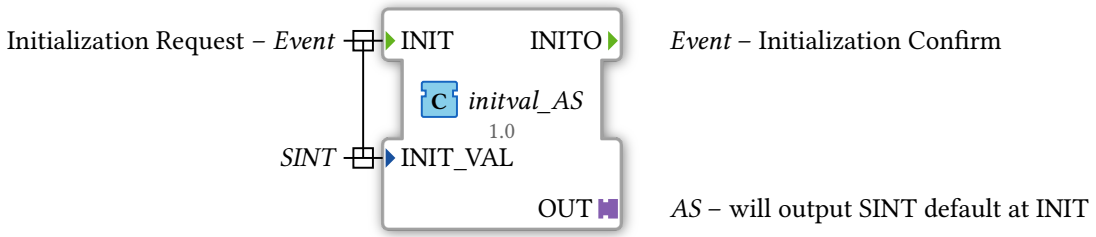
- SINT, INT, DINT: Why the choice of data type determines efficiency and errors

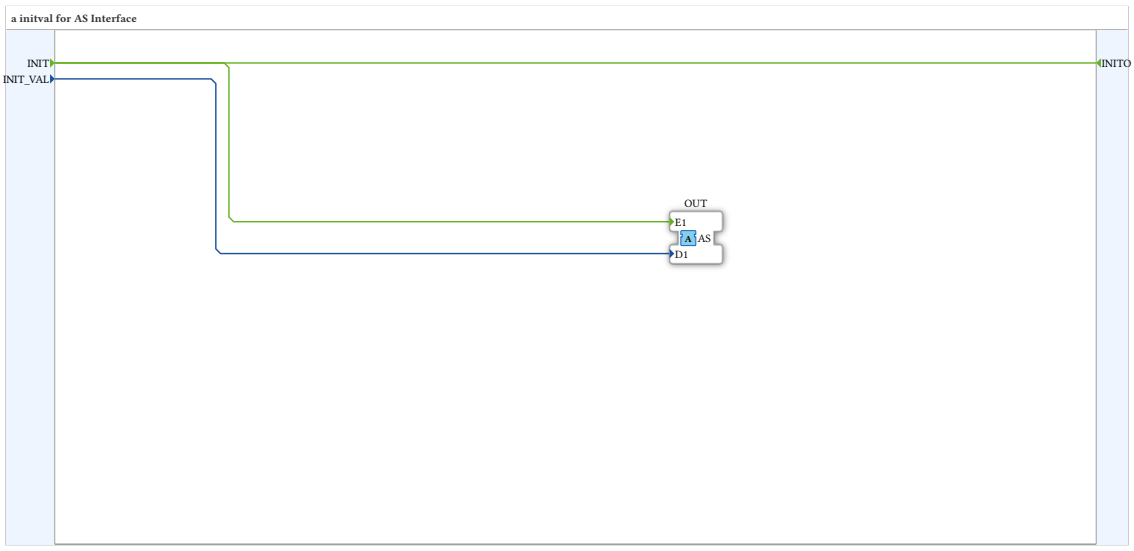
🛠️ Related Exercises

- Exercise_039_sub_NumbDisplay

####### Initval

initval_AS





• – ○ • – ○ • – ○ •

Introduction

`initval_AS` is the configurable initialization block for the AS adapter (data type `SINT`, 8-bit signed integer): it accepts a start value as a data input at initialization time and writes it to a `AS` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (SINT): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AS`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1/D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AS` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AS` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AS` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `SINT` value is fed onto a `AS` adapter via `initval_AS` and can then be used together with the pure adapter-based `AS` comparison blocks.
- A reusable block template for test setups where different `AS` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE/const_AX_FALSE` (which only exist for `AX`), `initval_AS` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

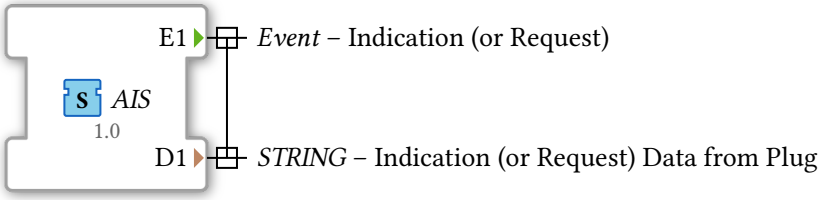
`initval_AS` provides simple, reusable initialization logic for the `AS` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `SINT` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numeric Index (unidirectional)

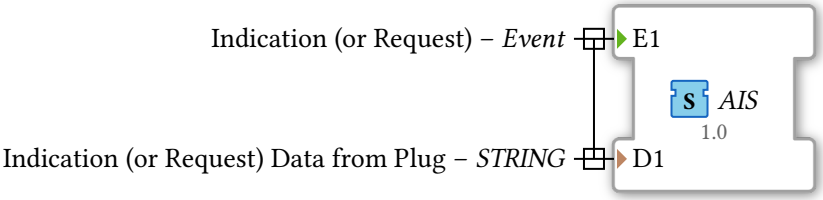
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. ...

AIS

Plug (adapter output)



Socket (adapter input)



Unidirectional Adapter for 1 Event and 1 String

• – ◦ • – ◦ • – ◦ •

Introduction

The **AIS** (Adapter Interface String) adapter provides a unidirectional interface through which exactly one event (E1) along with its associated string (D1) can be transmitted from a plug instance to a socket instance. It is used to implement simple, directed data flows where no feedback or additional control is required.

Interface Structure

Event Inputs

None.

Event Outputs

Name	Type	Comment
E1	Event	Indication (or Request) – the event is triggered when the data is sent.

Data Inputs

None.

Data Outputs

Name	Type	Comment
D1	STRING	The data (string) associated with the event.

Adapter

This component is itself an adapter. It has no other internal adapters.

Functionality

The adapter operates on the principle of a unidirectional “plug-socket” connection.

- The **plug** instance sends an event to E1 and simultaneously stores the data on D1.
- The **socket** instance receives this event and can read the value from D1.

Communication in the opposite direction is not supported.

Technical Features

- **Minimal Resource Consumption**: The adapter requires only one event and one string – ideal for resource-constrained controllers.
- **Type Safety**: The data type is fixed to STRING; the adapter can be used directly in applications that require precisely this signature.
- **Easy Reuse**: The strict separation between plug and socket allows for easy encapsulation and replacement of modules.

State Overview

The adapter does not have an internal state machine. It behaves **stateless**: Every incoming event on the plug side is immediately forwarded as an output event on the socket side. The data value D1 is transferred at the time of the event.

Application Scenarios

- **Sensor Connection**: Transfer of a single measured value (e.g., scrolling text from an RFID reader) to a processing unit.
- **Monitor Interface**: Output of a status string from a controller to a display element.
- **Debugging**: Easy extraction of diagnostic information from a function block.

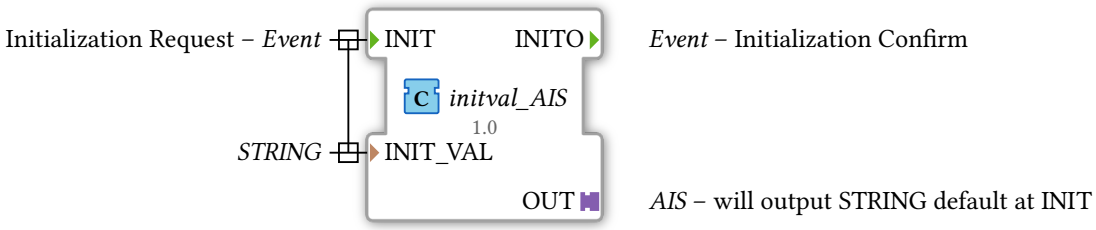
Comparison with Similar Blocks

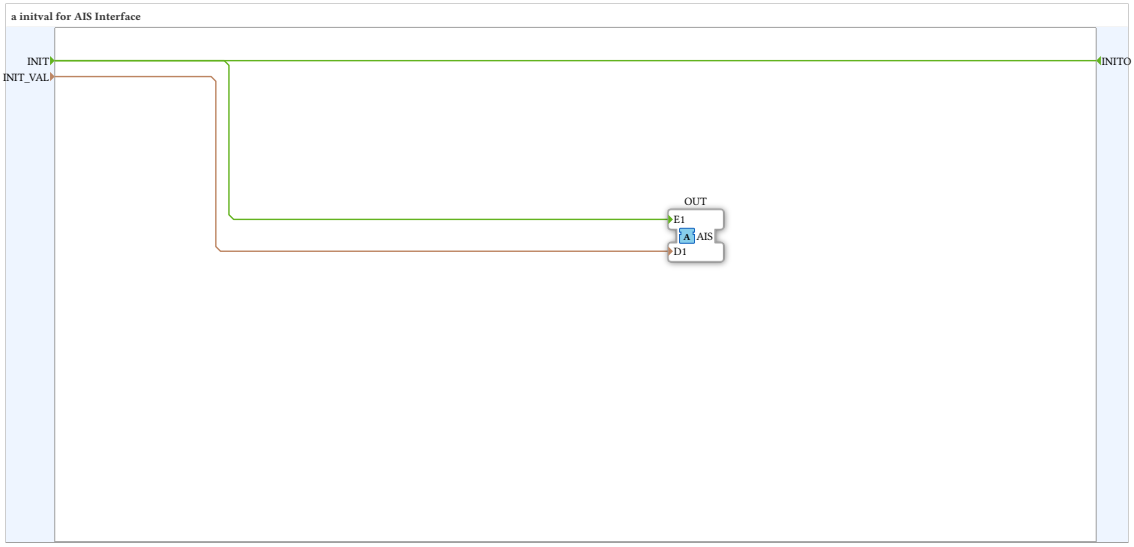
- **More complex adapters**, such as adapters with multiple events and mixed data types, offer greater flexibility but also require more configuration effort.
- **Bidirectional adapters** (e.g., with one event in each direction) allow feedback; the AIS adapter deliberately omits this to keep the interface as simple as possible.
- The AIS is specifically optimized for **exactly one event + one string** – other adapters with the same signature but a different name could be functionally identical.

Conclusion

The **AIS** adapter is a lean, stateless block for the unidirectional transmission of an event along with a string. It is ideally suited for all cases where simple, clear signal transmission without additional logic or a feedback channel is required. The minimal interface keeps the application clear and easy to understand.
####### Initval

initval_AIS





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AIS` is the configurable initialization block for the AIS adapter (data type `STRING`, character string): it accepts a start value as a data input at initialization time and writes it to a `AIS` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (STRING): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AIS`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AIS` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AIS` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AIS` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `STRING` value is fed onto a `AIS` adapter via `initval_AIS` and can then be used together with the pure adapter-based `AIS` comparison blocks.
- A reusable block template for test setups where different `AIS` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_AIS` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

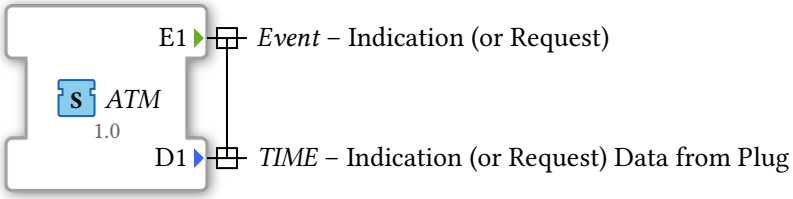
`initval_AIS` provides simple, reusable initialization logic for the `AIS` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `STRING` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numeric Index (unidirectional)

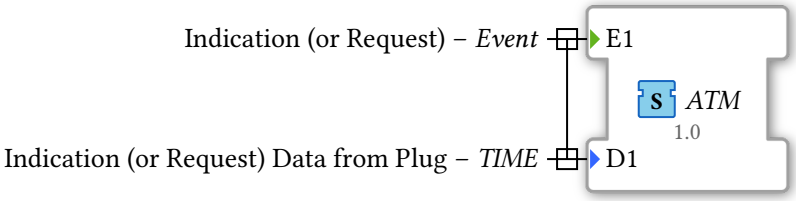
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

ATM

Plug (adapter output)



Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

The **ATM adapter** (short for "Adapter Time Message") is a unidirectional type according to IEC 61499-1 that provides an event-driven interface for transmitting a time value. It is used as an adapter to enable standardized, loosely coupled communication between function blocks (FBs). This adapter was developed by HR Agrartechnik GmbH and is available under the Eclipse Public License 2.0 (EPL-2.0).

Interface Structure

Event Inputs

The adapter has no event inputs.

Event Outputs

Name	Type	Comment
E1	Event	Indication (or request) – sent along with the data value D1

Data Inputs

The adapter has no data inputs.

Data Outputs

Name	Type	Comment
D1	TIME	Indication/Request Data from the Plug

Adapter

The module itself is of type `AdapterType`. No other adapter interfaces are included.

Functionality

The ATM adapter operates exclusively as a unidirectional interface. It is integrated into a function block as a **plug** (or as a **socket**) and, upon occurrence of the event **E1**, transmits the corresponding time value **D1** (type `TIME`) to the connected counterpart. The transmission is unidirectional: from the sending to the receiving block. The connection is automatically resolved by the 4diac IDE environment.

Technical Features

- **Unidirectional Adapter**: Only one data and event direction (plug → socket).
- **Simplest Implementation**: Exactly one event and one variable – minimal overhead.
- **Typing**: The transmitted variable is of the elementary type `TIME`, suitable for timestamps, delays, or time measurements.
- **License**: Released under EPL-2.0, therefore freely usable and extensible.
- No state machine or internal logic – pure interface.

State Overview

Since the ATM adapter has no internal state machine (stateless adapter), a state overview is not required. Its functionality is limited to the pure transmission of event data.

Application Scenarios

- **Transmission of timestamps** between different function blocks (e.g., logging, time-based control).
- **Signal for time-controlled actions**: A function block sends time information via the adapter, which is evaluated by another function block.
- **Test environments**: Simple, clearly defined interface for unit tests of function blocks that process time values.

Comparison with similar function blocks

Feature	ATM (this adapter)	Other unidirectional adapters (e.g., with multiple events/ data)
Events	1 (E1)	Usually multiple (e.g., INIT, REQ, IND, RSP)
Data	1 (TIME)	Often multiple data or different types
Complexity	Low	Medium to high
Application	Simple time transmission	Complex process scripts

Compared to more general adapters, ATM is limited to the essentials – ideal when only one event with a single time value needs to be exchanged.

Conclusion

The ATM adapter is a minimal yet practically relevant component for the unidirectional transmission of time values in IEC 61499 systems. Its clear interface and ease of use make it particularly suitable for

applications that focus on time-related communication. Thanks to its open licensing, the adapter can be integrated into or extended in your own projects without restrictions.

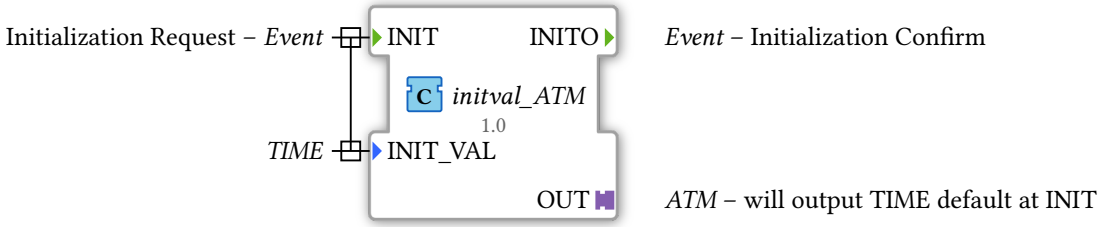
🌐 Related topic subpages on ms-muc-docs.de

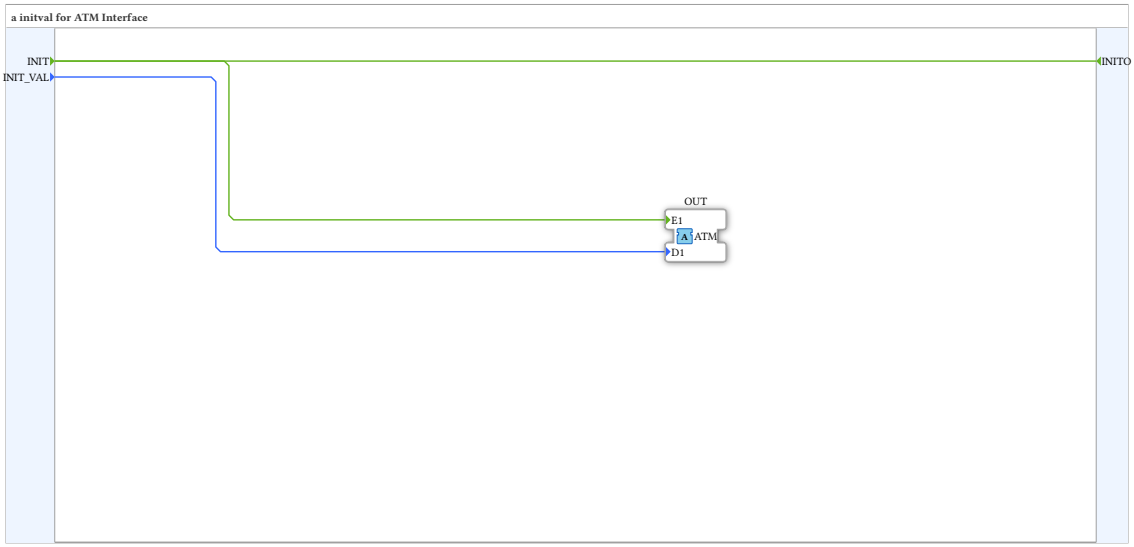
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

####### Initval

initval_ATM





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_ATM` is the configurable initialization block for the ATM adapter (data type `TIME`, duration): it accepts a start value as a data input at initialization time and writes it to a `ATM` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (TIME): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type ATM): on INIT, outputs the value passed via INIT_VAL through its E1/D1 adapter event.

Functionality

When the event INIT arrives (carrying With Var="INIT_VAL"), initval_ATM links two internal data paths: INIT_VAL is copied to OUT.D1 via a data connection, and INIT is forwarded to OUT.E1 via an event connection – so the OUT plug fires its adapter event carrying the value just supplied. INIT is also forwarded to INITO, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via INIT_VAL on every INIT call – so the block can also be used outside a pure initialization phase to dynamically set a ATM value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection INIT_VAL → OUT.D1 must be evaluated before the event connection so that OUT.E1 carries the current value – this is guaranteed by IEC 61499's FBNetwork connection execution order and needs no separate wiring.

State Overview

The block is stateless: every INIT event immediately results in INIT_VAL being written to OUT and confirmed via INITO.

Application Scenarios

- Initializing ATM adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant TIME value is fed onto a ATM adapter via initval_ATM and can then be used together with the pure adapter-based ATM comparison blocks.
- A reusable block template for test setups where different ATM start values need to be tried out

⚖ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the BOOL data type. Unlike const_AX_TRUE/const_AX_FALSE (which only exist for AX), initval_ATM does not parameterize the output value in the block itself, but takes it from the data input INIT_VAL.

Conclusion

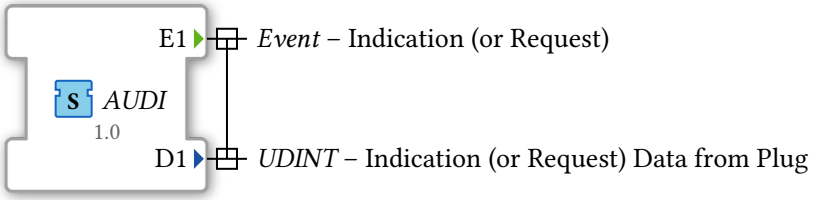
initval_ATM provides simple, reusable initialization logic for the ATM adapter and makes the output value freely configurable via INIT_VAL. It is the right fit wherever a TIME start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

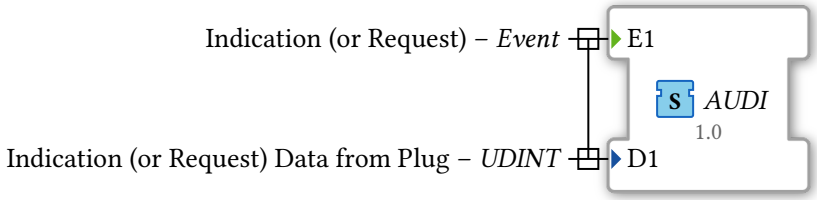
Here you will find all exercises from the unidirectional workspace in ascending order.

AUDI (UDINT)

Plug (adapter output)



Socket (adapter input)



unidirectional Adapter Interface for 1 Event and 1 Udint

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	UDINT	Indication (or Request) Data from Plug

🛠️ Related Exercises

- Uebung_009_AX
- Uebung_011_AUDI
- Uebung_083_AX

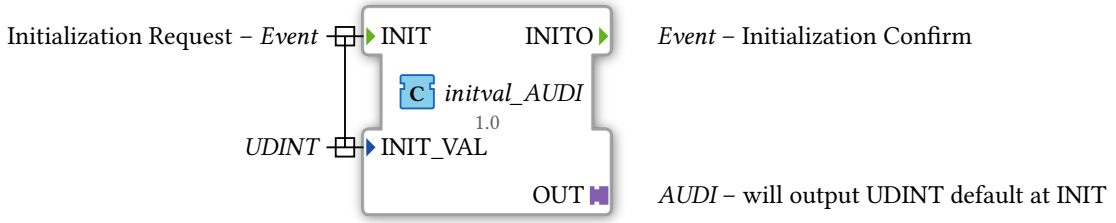
Unidirectional UDINT adapter

🛠️ Related exercises

- Uebung_009
- Uebung_009_AX
- Uebung_009a
- Uebung_011
- Uebung_011a
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_012c
- Uebung_015
- Uebung_015a
- Uebung_020c2_sub
- Uebung_028
- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_034b
- Uebung_072c
- Uebung_083
- Uebung_083_AX
- Uebung_103
- Uebung_111
- Uebung_151_AX
- Uebung_152
- Uebung_153

####### Initval

initval_AUDI





• – ○ • – ○ • – ○ •

Introduction

`initval_AUDI` is the configurable initialization block for the AUDI adapter (data type `UDINT`, 32-bit unsigned integer): it accepts a start value as a data input at initialization time and writes it to a `AUDI` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (UDINT): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AUDI`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1/D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AUDI` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AUDI` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AUDI` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `UDINT` value is fed onto a `AUDI` adapter via `initval_AUDI` and can then be used together with the pure adapter-based `AUDI` comparison blocks.
- A reusable block template for test setups where different `AUDI` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_AUDI` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

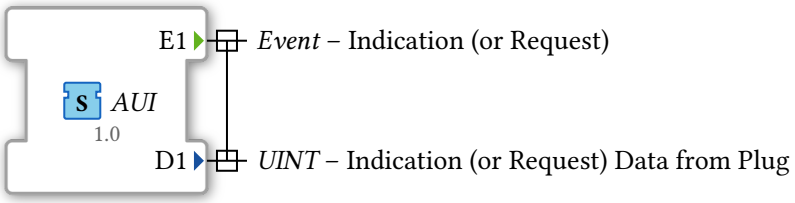
`initval_AUDI` provides simple, reusable initialization logic for the `AUDI` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `UDINT` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

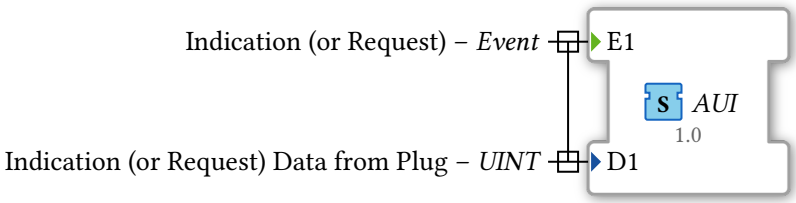
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AUI (UINT)

Plug (adapter output)



Socket (adapter input)



unidirectional Adapter Interface for 1 Event and 1 Uint

Interface

Events

Name	Comment	With

E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	UINT	Indication (or Request) Data from Plug

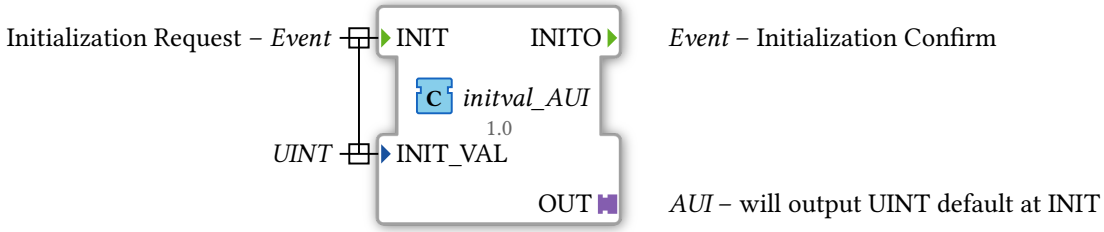
Unidirectional UINT adapter

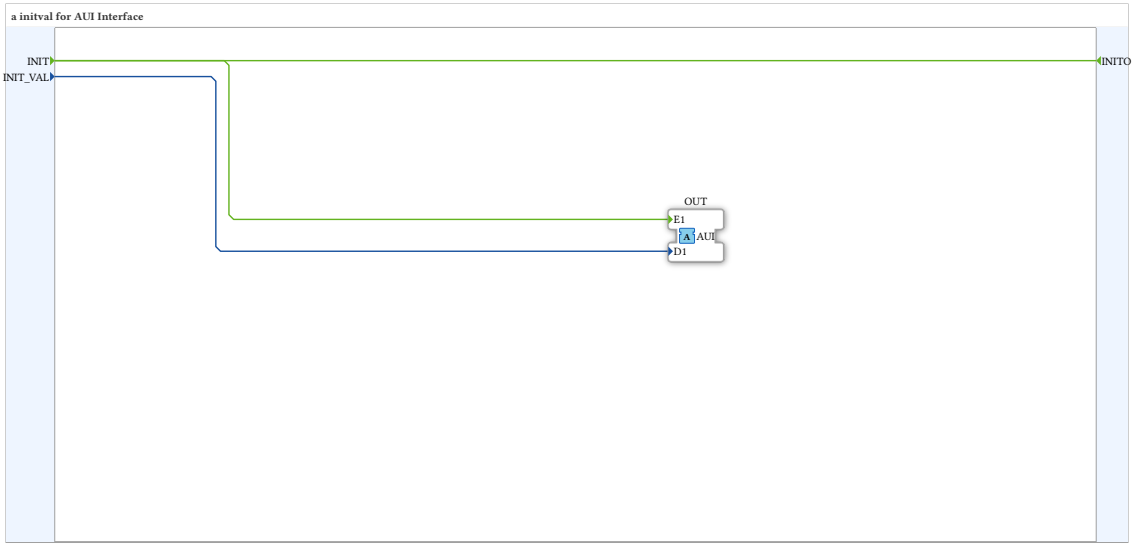
🛠️ Related exercises

- Uebung_006c
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010c2
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- NumericValue_ID_TO_NVS
- Uebung_020c2_sub
- IX_TO_logiBUS_LED_strip_QX
- Uebung_035
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- Uebung_039_sub_NumbAnsicht
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_040
- Uebung_040_AX
- Uebung_041
- Uebung_087
- Uebung_087a1
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2
- Uebung_090a2_AX
- Uebung_103

####### Initval

initval_AUI





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AUI` is the configurable initialization block for the AUI adapter (data type `UINT`, 16-bit unsigned integer): it accepts a start value as a data input at initialization time and writes it to a `AUI` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (UINT): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AUI`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AUI` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AUI` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AUI` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `UINT` value is fed onto a `AUI` adapter via `initval_AUI` and can then be used together with the pure adapter-based `AUI` comparison blocks.
- A reusable block template for test setups where different `AUI` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_AUI` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

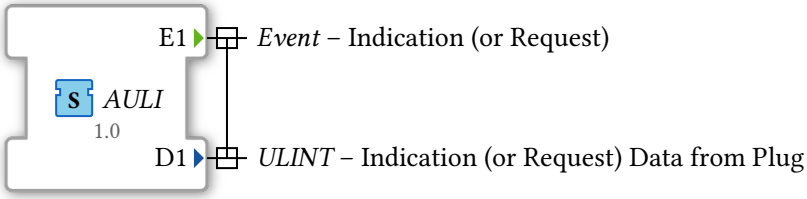
`initval_AUI` provides simple, reusable initialization logic for the `AUI` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `UINT` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

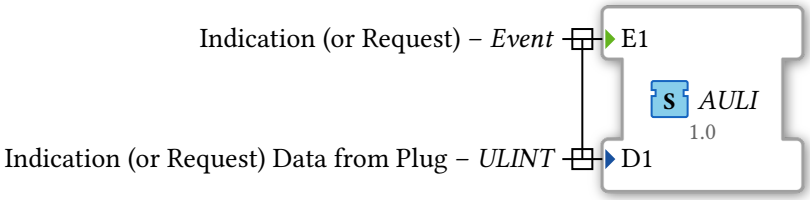
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AULI (ULINT)

Plug (adapter output)



Socket (adapter input)



🎧 Podcast

- External Gear Machines: From Workhorse to Hydraulic Intelligence – Challenges, Innovations & Kepler's Legacy
- Danfoss and Eaton Hydraulics Twin Story Comes to an End
- The Underestimated Hero: Why the Oil Reservoir Revolutionizes Your Hydraulic System
- Pressure Relief Valves: Hydraulic Life Insurance – Types, Function, and System Integration
- Powerhouses in Action: The Secret of Hydraulic Cylinders – From Excavators to High-Tech Machines

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

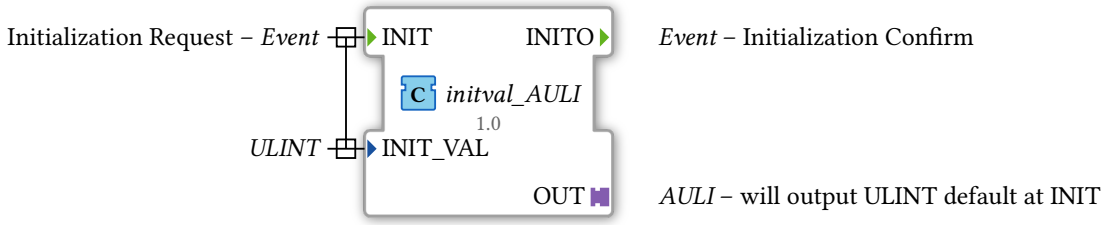
Data

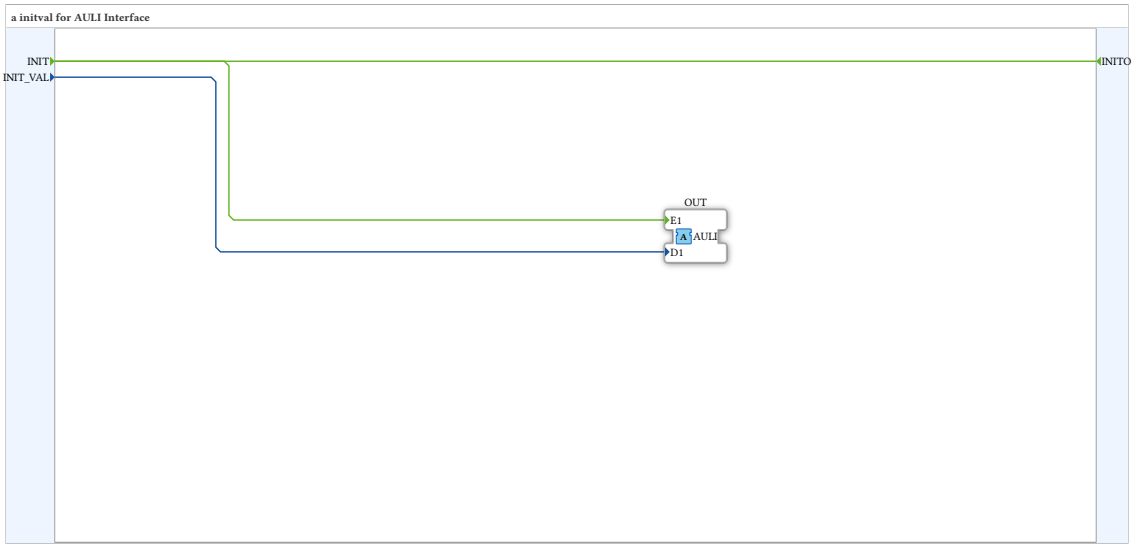
Name	Type	Comment
D1	ULINT	Indication (or Request) Data from Plug

Unidirectional ULINT adapter

####### Initval

initval_AULI





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AULI` is the configurable initialization block for the AULI adapter (data type `ULINT`, 64-bit unsigned integer): it accepts a start value as a data input at initialization time and writes it to a `AULI` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (ULINT): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AULI`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AULI` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AULI` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AULI` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `ULINT` value is fed onto a `AULI` adapter via `initval_AULI` and can then be used together with the pure adapter-based `AULI` comparison blocks.
- A reusable block template for test setups where different `AULI` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_AULI` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

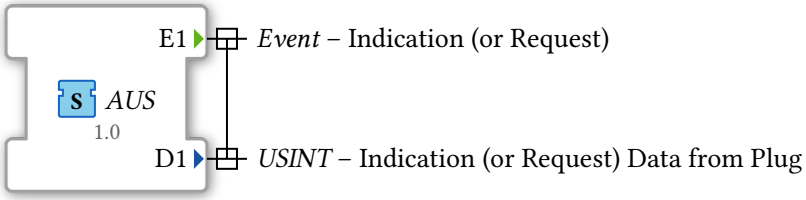
`initval_AULI` provides simple, reusable initialization logic for the `AULI` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `ULINT` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

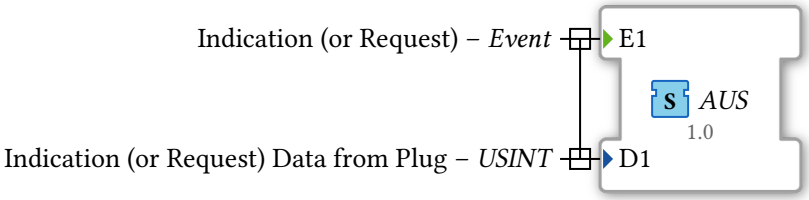
Here you will find all exercises from the `unidirectional` workspace in ascending order.

OFF (USINT)

Plug (adapter output)



Socket (adapter input)



🎧 Podcast

- The E_CTU block: Event-driven incrementing in industry according to IEC 61499
- The E_PERMIT block: The “gatekeeper” for events in IEC 61499 systems decoded
- The E_T_FF_SR block: The heart of IEC 61499 – storing, switching, and responding
- DIN EN 61499-1 demystified: Function blocks, object orientation, and distributed systems
- DIN EN 61499-1: Building blocks for Industry 4.0 – How to revolutionize Function blocks for automation ()

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

Name	Type	Comment
D1	USINT	Indication (or Request) Data from Plug

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter block on ms-muc-docs.de ()

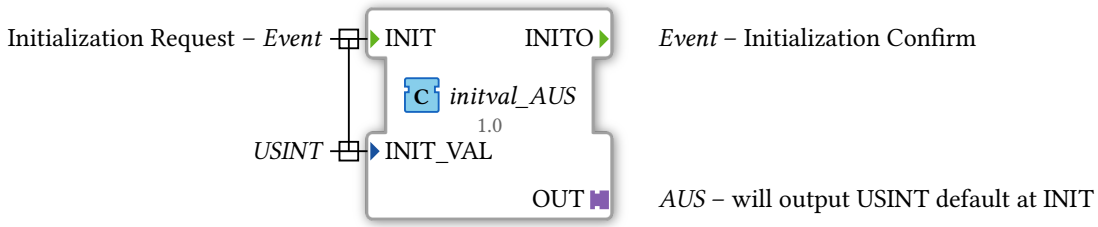
Unidirectional USINT adapter

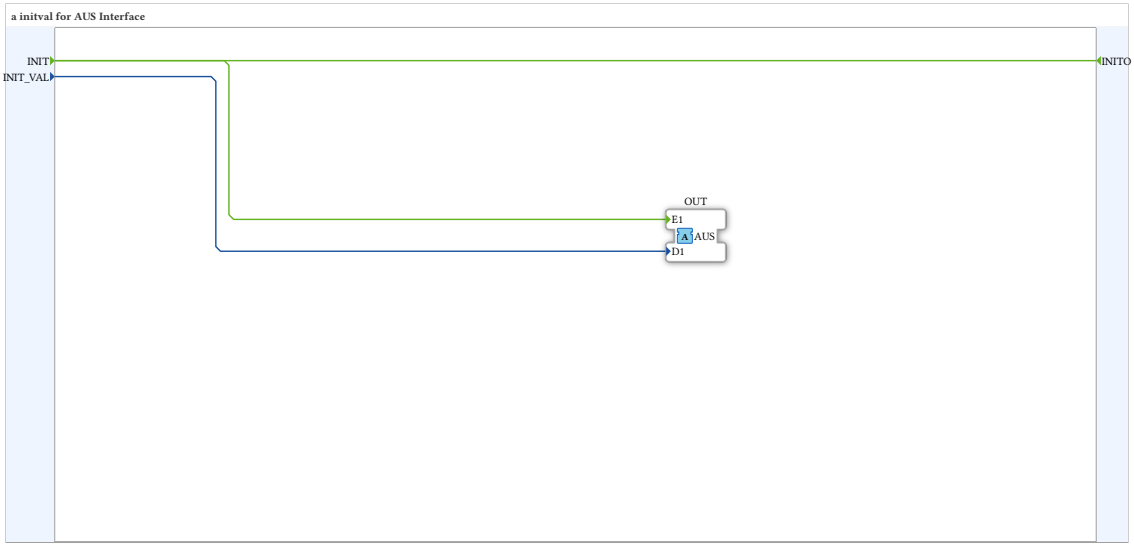
🛠️ Related exercises

- Funk_IX_TO_DataPanel_MI_QX
- Funk_IXA_TO_DataPanel_MI_QXA
- IX_TO_Aux_QX
- IXA_TO_Aux_QXA
- IX_TO_logiBUS_LED_strip_QX
- SIN_TO_PCAN_Callback_Byte

####### Initval

initval_AUS





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AUS` is the configurable initialization block for the AUS adapter (data type `USINT`, 8-bit unsigned integer): it accepts a start value as a data input at initialization time and writes it to a `AUS` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (USINT): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AUS`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AUS` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AUS` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AUS` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `USINT` value is fed onto a `AUS` adapter via `initval_AUS` and can then be used together with the pure adapter-based `AUS` comparison blocks.
- A reusable block template for test setups where different `AUS` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_AUS` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

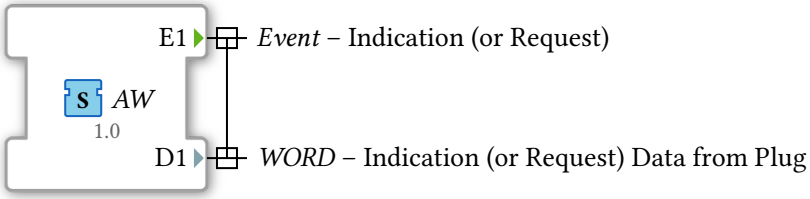
`initval_AUS` provides simple, reusable initialization logic for the `AUS` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `USINT` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numerical Index (unidirectional)

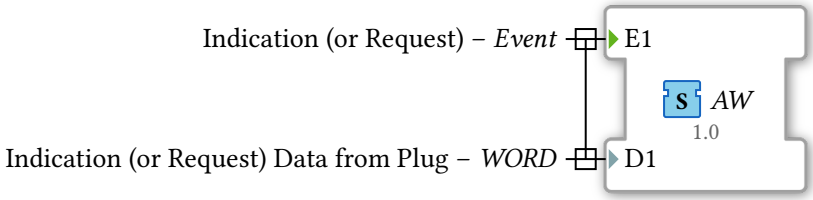
Here you will find all exercises from the `unidirectional` workspace in ascending order.

AW (WORD)

Plug (adapter output)



Socket (adapter input)



🎧 Podcast

- From "Get Away with Electronics" to Essential: How Agricultural Tech Earned Trust and Delivered Real Relief

Interface

Events

Name	Comment	With
E1	Indication (or Request)	D1

Data

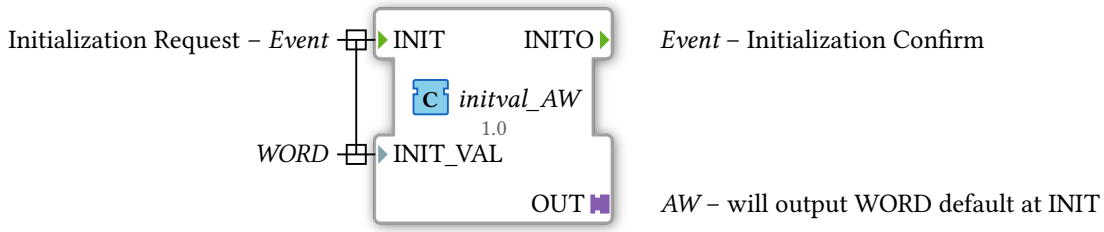
Name	Type	Comment

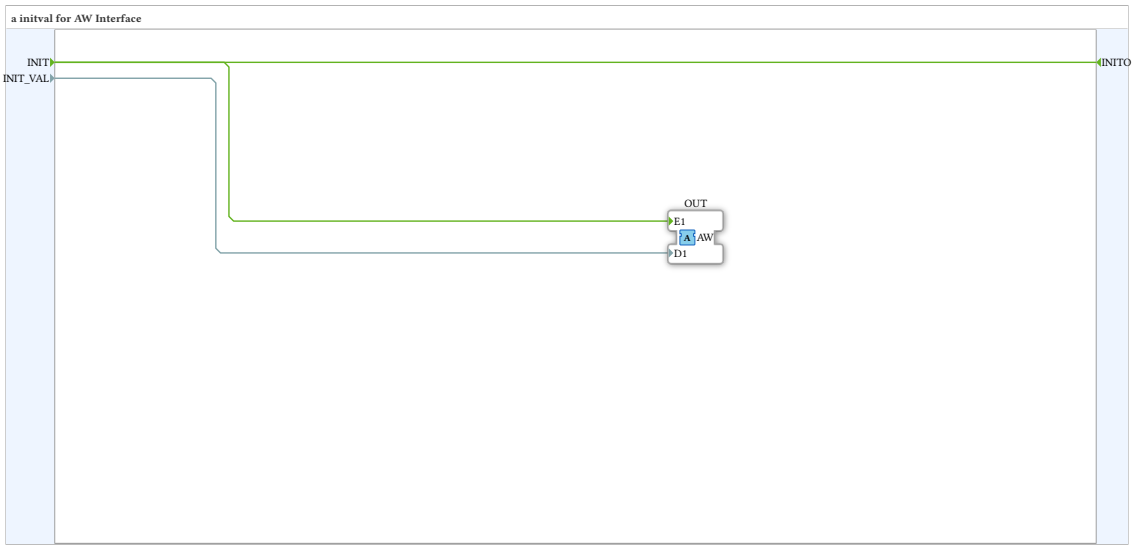
D1	WORD	Indication (or Request) Data from Plug

Unidirectional WORD adapter

####### Initval

initval_AW





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AW` is the configurable initialization block for the AW adapter (data type `WORD`, 16-bit bit pattern): it accepts a start value as a data input at initialization time and writes it to a `AW` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (WORD): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AW`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1`/`D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AW` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AW` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AW` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `WORD` value is fed onto a `AW` adapter via `initval_AW` and can then be used together with the pure adapter-based `AW` comparison blocks.
- A reusable block template for test setups where different `AW` start values need to be tried out

⚖️ Comparison with Similar Blocks

Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE`/`const_AX_FALSE` (which only exist for `AX`), `initval_AW` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

Conclusion

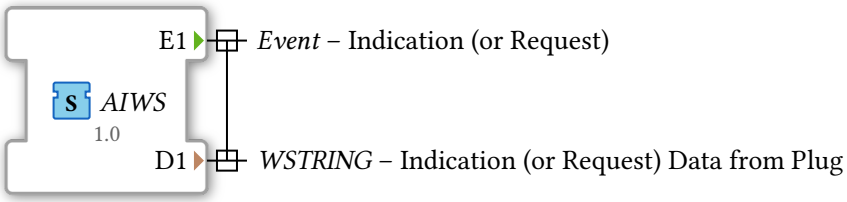
`initval_AW` provides simple, reusable initialization logic for the `AW` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `WORD` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

Numeric Index (unidirectional)

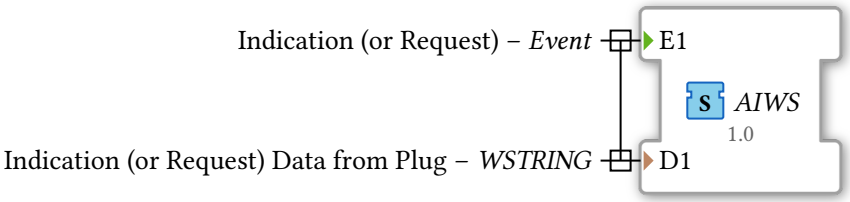
Here you will find all exercises from the `unidirectional` workspace, listed in ascending order. …

AIWS

Plug (adapter output)



Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

The **AIWS** adapter provides a unidirectional interface for transmitting exactly one event and its associated WSTRING data. It defines the communication direction from a source to a destination, with the event always accompanied by the data. The adapter serves as a basic building block for loosely coupled, event-driven data flows in IEC 61499 applications.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

Name	Type	Comment
E1	Event	Indication (or Request)

Data Inputs

No data inputs available.

Data Outputs

Name	Type	Comment
D1	WSTRING	Indication (or Request) Data from Plug

Adapter

The adapter itself is to be understood as an interface definition. It has no further internal adapters.

Functionality

The **AIWS** adapter transmits an event **E1** along with a WSTRING value **D1** in a fixed coupling. As soon as the sending component triggers the event, the current value of **D1** is passed to the receiving side. The direction is strictly from the plug side to the socket side (or vice versa, depending on the use case) – no feedback or acknowledgment takes place. The adapter thus defines a simple, unidirectional data channel.

Technical Features

- **Unidirectionality**: There is only one event output side, no inputs. This enforces a clear flow direction.
- **Compactness**: Exactly one event with a simple data type (WSTRING) minimizes interface complexity.
- **No Error Handling**: The adapter does not provide mechanisms for timeouts, retries, or error detection. Data integrity must be ensured by the connected function blocks.
- **Standard Compliant**: Complies with the IEC 61499-1 specification for adapter types.

State Overview

The adapter itself has no internal state. It serves solely as an interface definition and does not implement its own sequence control. The state logic resides with the connected function blocks.

Application Scenarios

- **Integration of Sensor Data**: Transmission of an event with a measured value (e.g., status message or identification) as a WSTRING to a higher-level controller.
- **Text-Based Commands**: Sending commands (e.g., “START”, “STOP”) as events with parameters.
- **Logging and Diagnostics**: Output of descriptive events with plain text information to a log component.

Comparison with Similar Components

Component	Events	Data	Direction	Special Feature
AIWS	1 Output	1 WSTRING	Unidirectional	Simplest adapter for WSTRING

AI (hypothetical)	1 Output	1 ANY (generic)	Unidirectional	Flexible data type, but more complex
BIWS (bidirectional)	2 Events (in/out)	2 WSTRING in both directions	Bidirectional	Requires handshake

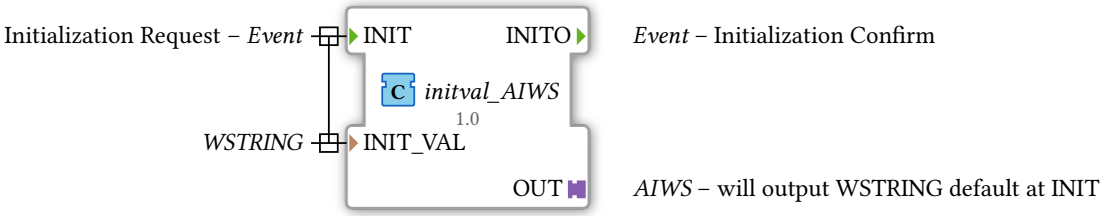
The **AIWS** stands out due to its minimal interface and is particularly suitable for simple, clearly defined point-to-point connections where no return channel is required.

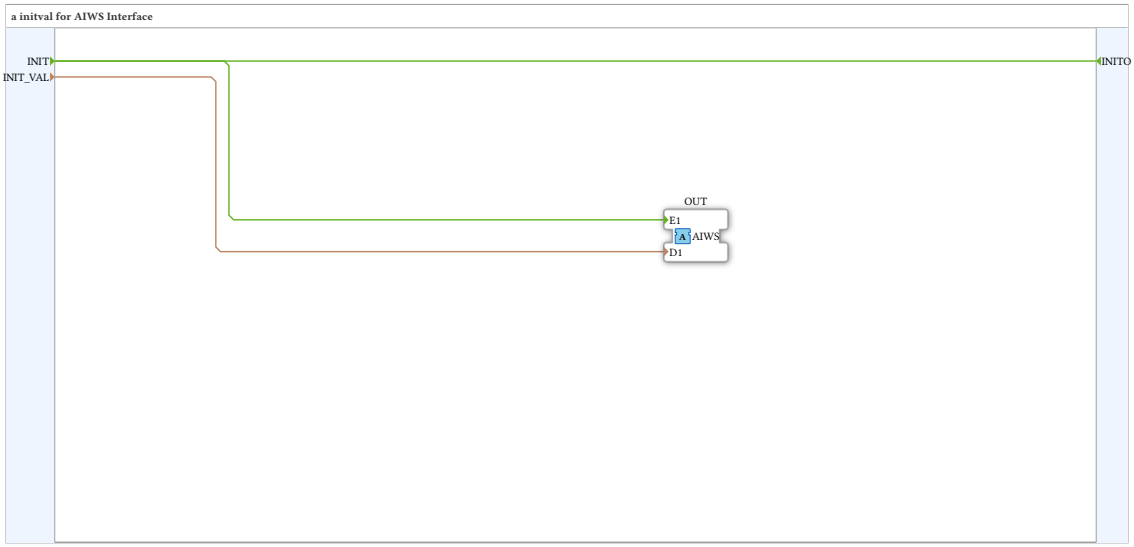
Conclusion

The **AIWS** adapter is a fundamental and efficient building block for the unidirectional transmission of an event with a WSTRING data. Its simplicity reduces coupling between components and facilitates the modular design of automation solutions according to IEC 61499. It is ideally suited for tasks requiring fast and straightforward event-driven communication.

####### Initval

initval_AIWS





• – ◦ • – ◦ • – ◦ •

Introduction

`initval_AIWS` is the configurable initialization block for the AIWS adapter (data type `WSTRING`, wide (Unicode) character string): it accepts a start value as a data input at initialization time and writes it to a `AIWS` adapter plug. This replaces the formerly common practice of feeding a constant comparison value in via a mixed adapter/elementary block.

Interface Structure

Event Inputs

- **INIT**: initialization request, carries `INIT_VAL`. Triggers `INIT_VAL` being written to plug `OUT`.

Event Outputs

- **INITO**: confirms completion of the initialization.

Data Inputs

- **INIT_VAL** (WSTRING): the start value written to adapter `OUT` when `INIT` arrives.

Data Outputs

- None directly; the value is only made available through adapter `OUT`.

Adapters

- **OUT** (Plug, type `AIWS`): on `INIT`, outputs the value passed via `INIT_VAL` through its `E1/D1` adapter event.

Functionality

When the event `INIT` arrives (carrying `With Var="INIT_VAL"`), `initval_AIWS` links two internal data paths: `INIT_VAL` is copied to `OUT.D1` via a data connection, and `INIT` is forwarded to `OUT.E1` via an event connection – so the `OUT` plug fires its adapter event carrying the value just supplied. `INIT` is also forwarded to `INITO`, letting the caller know that initialization has completed.

Technical Details

- The output value is not parameterized in the block itself but supplied fresh via `INIT_VAL` on every `INIT` call – so the block can also be used outside a pure initialization phase to dynamically set a `AIWS` value.
- Pure wiring logic with no ECC/state machine of its own (a composite FB with no internal function block instances); the data connection `INIT_VAL → OUT.D1` must be evaluated before the event connection so that `OUT.E1` carries the current value – this is guaranteed by IEC 61499's `FBNetwork` connection execution order and needs no separate wiring.

State Overview

The block is stateless: every `INIT` event immediately results in `INIT_VAL` being written to `OUT` and confirmed via `INITO`.

Application Scenarios

- Initializing `AIWS` adapter connections with a project- or parameter-dependent start value (e.g. from a configuration variable)
- Replacement for the removed "mixed" adapter/elementary comparison blocks: a constant `WSTRING` value is fed onto a `AIWS` adapter via `initval_AIWS` and can then be used together with the pure adapter-based `AIWS` comparison blocks.
- A reusable block template for test setups where different `AIWS` start values need to be tried out

⚖ Comparison with Similar Blocks

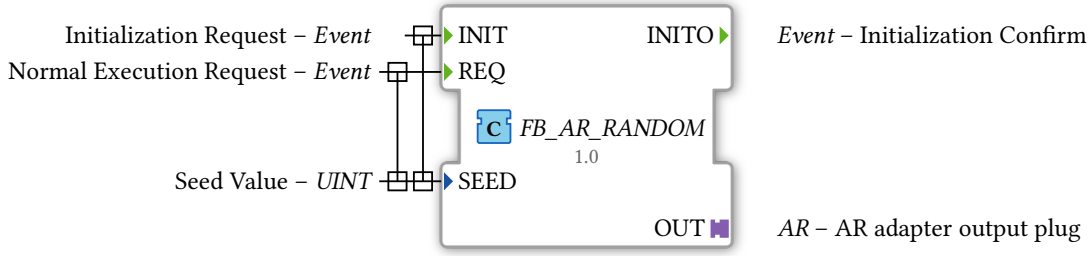
Compare with initval_AX, which uses the same wiring mechanism for the `BOOL` data type. Unlike `const_AX_TRUE/const_AX_FALSE` (which only exist for `AX`), `initval_AIWS` does not parameterize the output value in the block itself, but takes it from the data input `INIT_VAL`.

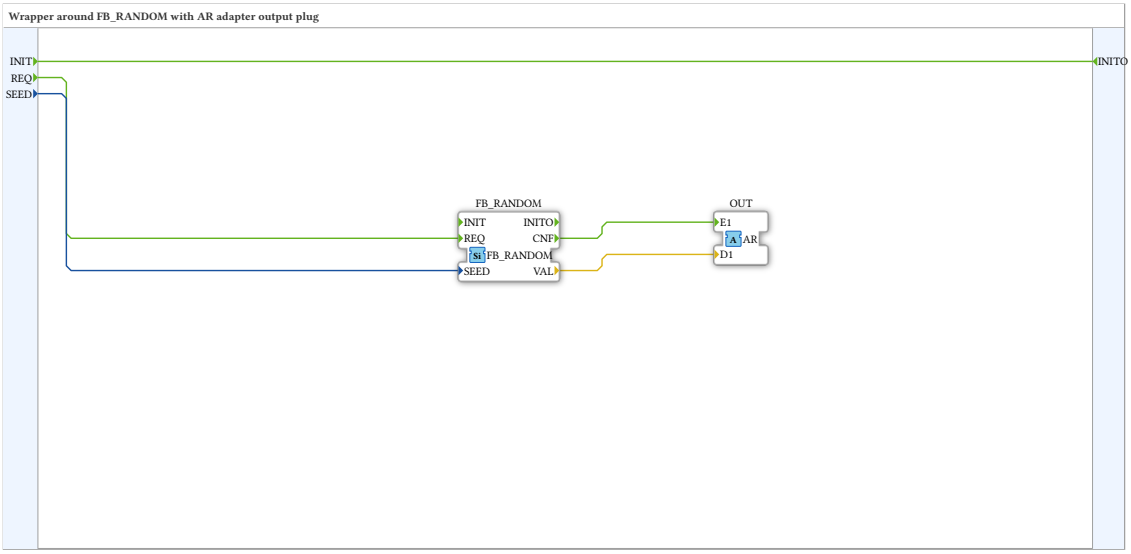
Conclusion

`initval_AIWS` provides simple, reusable initialization logic for the `AIWS` adapter and makes the output value freely configurable via `INIT_VAL`. It is the right fit wherever a `WSTRING` start value isn't already fixed at design time, or as a replacement for a removed mixed comparison block.

2.10.0.3 Utils

FB_AR_RANDOM





• – ◦ • – ◦ • – ◦ •

Introduction

FB_AR_RANDOM is a composite function block wrapper around `eclipse4diac::utils::FB_RANDOM`. It exposes a random value through a unidirectional AR adapter output, while keeping the external interface minimal. The block provides an initialization handshake via `INIT`/`INITO` and uses the `REQ` event to trigger the underlying random number generator.

Interface Structure

Event Inputs

- `INIT` (EInit): Initialization request. This event is associated with the `SEED` input and is directly confirmed by `INITO`.
- `REQ` (Event): Normal execution request. This event triggers the internal random number generation and causes the result to be sent through the adapter output. It is also associated with the `SEED` input.

Event Outputs

- `INITO` (EInit): Initialization confirm. This event acknowledges the `INIT` request.

Data Inputs

- `SEED` (UINT): Seed value used by the underlying random number generator. The default value is `0`.

Data Outputs

- None. The generated random value is not provided as a direct data output. It is transmitted through the adapter output instead.

Adapters

- `OUT` (plug, `adapter::types::unidirectional::AR`): Unidirectional AR adapter output. The generated value is sent via the adapter event `E1` with the corresponding data on `D1`.

Functionality

When an `INIT` event is received, the FB passes this request through and immediately acknowledges it with `INITO`. No random generation is performed during initialization.

When a `REQ` event is received, the `SEED` value is forwarded to the internal `FB_RANDOM` block. The internal block generates a random value and, upon completion, triggers the adapter event `E1` on the `OUT` plug. The generated value is made available on the adapter data port `D1`.

Technical Features

- Composite FB containing one instance of `eclipse4diac::utils::FB_RANDOM`.
- Unidirectional AR adapter output for decoupled, adapter-based communication.

- `UINT` seed input with a default value of `0`.
- Simple initialization handshake through `INIT` and `INITO`.
- No direct output variables; results are delivered exclusively through the adapter.
- No internal ECC; behavior is defined by the contained FBNetwork.

State Overview

The FB itself does not define an explicit state machine. Its behavior is event-driven and is determined by the internal FBNetwork. The initialization event is forwarded directly to the initialization confirm output without affecting the random generator. The `REQ` event can be used whenever a new random value is required.

Application Scenarios

- Integration of a random number generator into IEC 61499 applications through a standard AR adapter interface.
- Use in testbeds, simulations, or control applications that require pseudo-random values.
- Modular system designs where data exchange should be encapsulated in adapters instead of direct data connections.
- Supplying random setpoints, seeds, or scheduling parameters to other function blocks in a decoupled manner.

Comparison with Similar Blocks

Compared to a plain `FB_RANDOM`, this wrapper hides the `CNF` event and the `VAL` data output and instead provides them through a unidirectional AR adapter. This makes the block more suitable for adapter-based architectures and plug-and-socket connections. It also adds a distinct initialization handshake that a direct random generator does not provide. Unlike blocks with ordinary output variables, `FB_AR_RANDOM` allows the random source to be easily replaced or reused in different contexts.

Conclusion

`FB_AR_RANDOM` is a compact and reusable wrapper that makes a pseudo-random number generator available through a unidirectional AR adapter. It simplifies integration, introduces a clear initialization flow, and fits well into adapter-oriented IEC 61499 system designs.

2.10.1 iec61131-3-bool

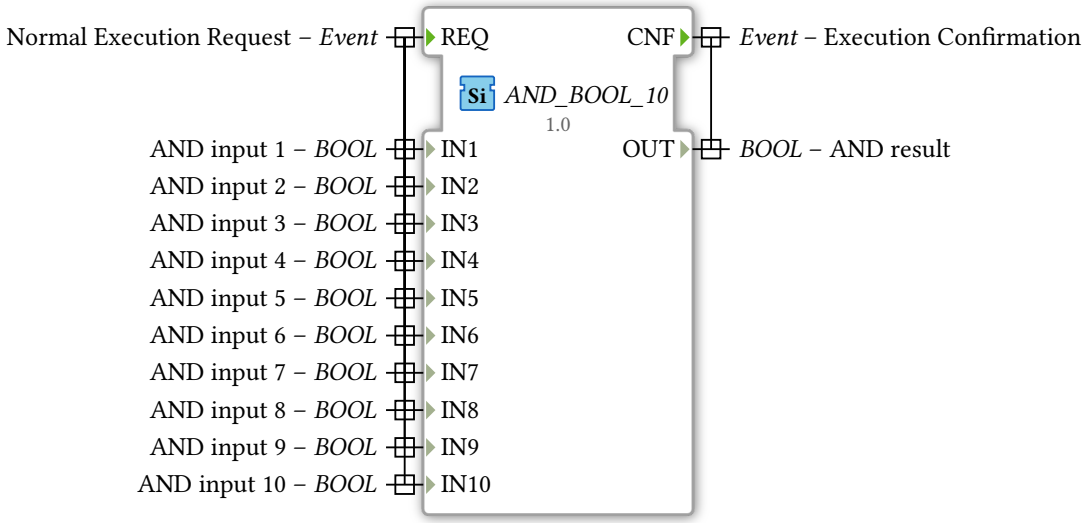
2.10.1.1 booleanOperators

AND_BOOL_10

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `AND_BOOL_10` is a generic block for calculating the logical AND operation. It performs a bitwise AND operation on up to ten separate BOOL inputs. The block is classified according to the IEC 61131-3 standard and belongs to the category of standard Boolean functions. It is typically used in control logic to link conditions where all input signals must be true (TRUE) to activate an output signal.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** This event triggers the execution of the function block. Upon its arrival, the values of all ten data inputs (`IN1` to `IN10`) are read, and the AND operation is performed.

Event Outputs

- **CNF (Execution Confirmation):** This event signals the completion of the calculation. It is output along with the calculated result at data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all ten inputs. The output is only TRUE if **all** inputs `IN1` to `IN10` have the value TRUE. In all other cases, the output is FALSE.

Adapter

This function block has no adapter interfaces.

Operation

The operation is deterministic and event-driven. Upon each occurrence of the event `REQ`, the following logical operation is performed:

OUT = IN1 AND IN2 AND IN3 AND IN4 AND IN5 AND IN6 AND IN7 AND IN8 AND IN9 AND IN10`

The result is then made available at the data output `OUT`, and the acknowledgment event `CNF` is triggered to inform subsequent blocks of the availability of the new result.

Technical Features

- **Generic Block:** The block is implemented as a generic block (attribute `GenericClassName = 'GEN_AND'`). This means that its functionality can potentially be parameterized for a different number of inputs, although this specific instance is fixed to ten inputs.

- **Package:** The block is included in the package `iec61131::bitwiseOperators`.
- **Full Evaluation:** All ten inputs are evaluated on each execution.

State Overview

The `AND_BOOL_10` block has no internal state (memoryless). Its output depends solely on the current values of the inputs at the time of triggering by `REQ`. There are no delays, filters, or hysteresis.

Application Scenarios

- **Safety Interconnects:** In machine safety circuits, where multiple safety switches (emergency stop, light barriers, door contacts) must all be closed before a hazardous process can begin.
- **Multiple Conditions:** In process control systems, to check whether multiple conditions (e.g., fill level, temperature, pressure) are simultaneously within their target ranges.
- **Enable Logic:** As part of an enable chain where multiple stations or operators must give their approval (TRUE signal).

⚖️ Comparison with Similar Function Blocks

- **`AND` (with 2 inputs):** The standard AND block according to IEC 61131-3 typically has only two inputs. `AND_BOOL_10` extends this functionality to ten inputs without the need to connect multiple two-input AND blocks. See: AND_10
- **`OR_BOOL_10`:** Performs a logical OR operation. The result is TRUE if at least one input is TRUE, while `AND_BOOL_10` requires all inputs to be TRUE.
- **`XOR` / `XNOR`:** Compute the exclusive OR and equivalent operations, respectively, which are based on the parity of the TRUE signals, not an all-or-none condition like AND.
- **Generic Blocks (`GEN_AND`):** `AND_BOOL_10` is a specific instance of a generic AND block. In other environments, a configurable `GEN_AND` block could be used, to which the desired number of inputs are parameterized.

Conclusion

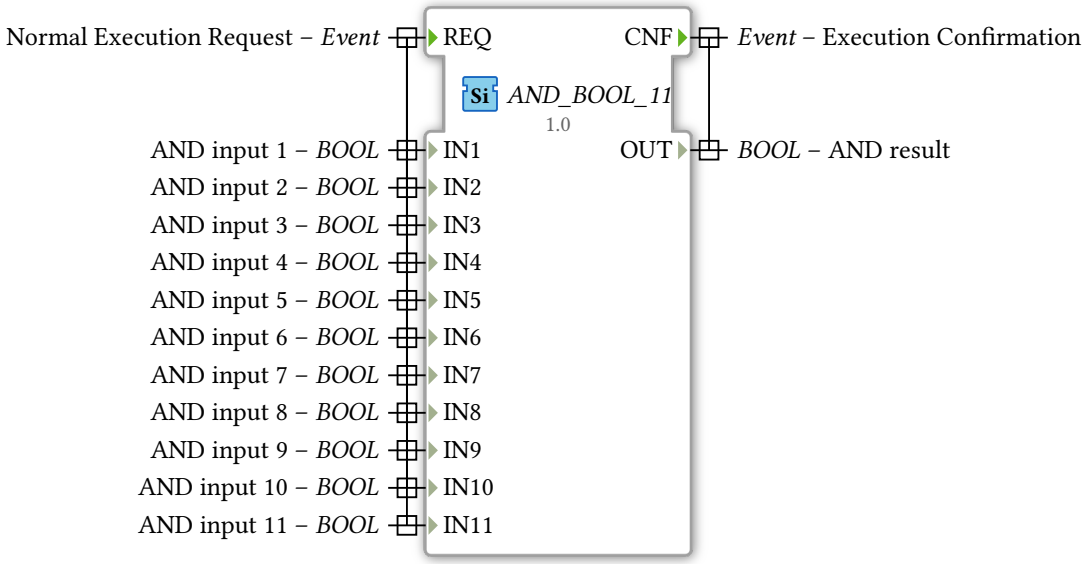
The `AND_BOOL_10` is a specialized, reliable, and easy-to-use function block for applications that require a logical AND operation across a large, fixed number of conditions. Its event-driven, stateless nature makes it predictable and easy to integrate into existing control flows. For scenarios with a variable number of conditions, flexible, parameterizable blocks should be considered.

AND_BOOL_11

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_11` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 11 separate BOOL inputs (`IN1` to `IN11`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 11 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 11 data inputs (`IN1` to `IN11`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 11 inputs. The output is TRUE only if **all** inputs `IN1` to `IN11` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 11 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN11` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 11 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_11`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_11` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_11` is a simple, event-driven block for the AND combination of 11 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

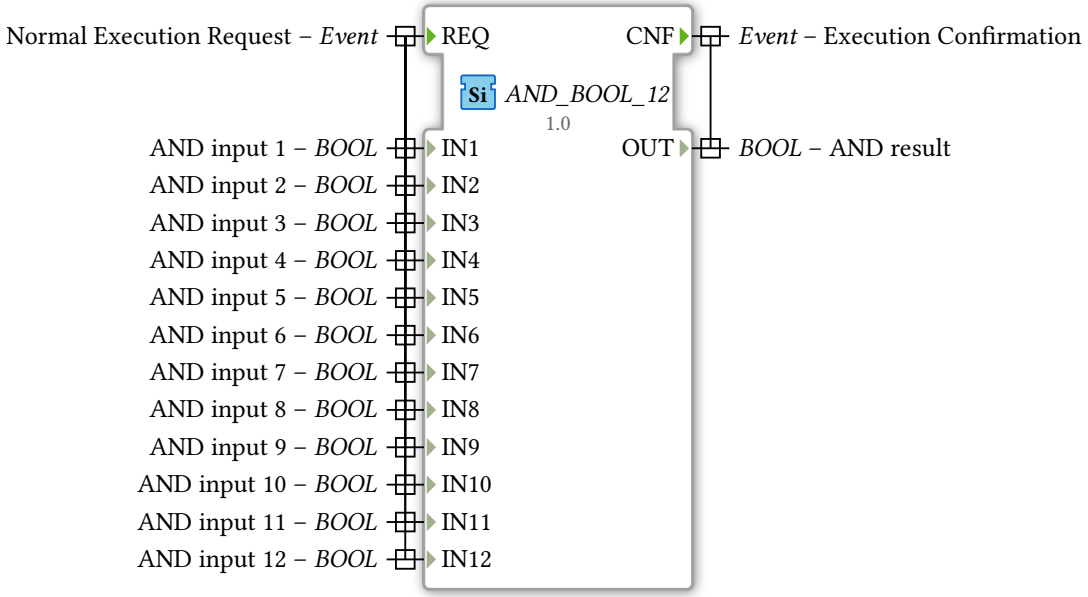
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_12

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_12` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 12 separate BOOL inputs (`IN1` to `IN12`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 12 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 12 data inputs (`IN1` to `IN12`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.
- **IN12 (AND input 12):** Boolean input 12.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 12 inputs. The output is TRUE only if **all** inputs `IN1` to `IN12` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 12 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN12` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 12 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_12`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_12` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_12` is a simple, event-driven block for the AND combination of 12 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

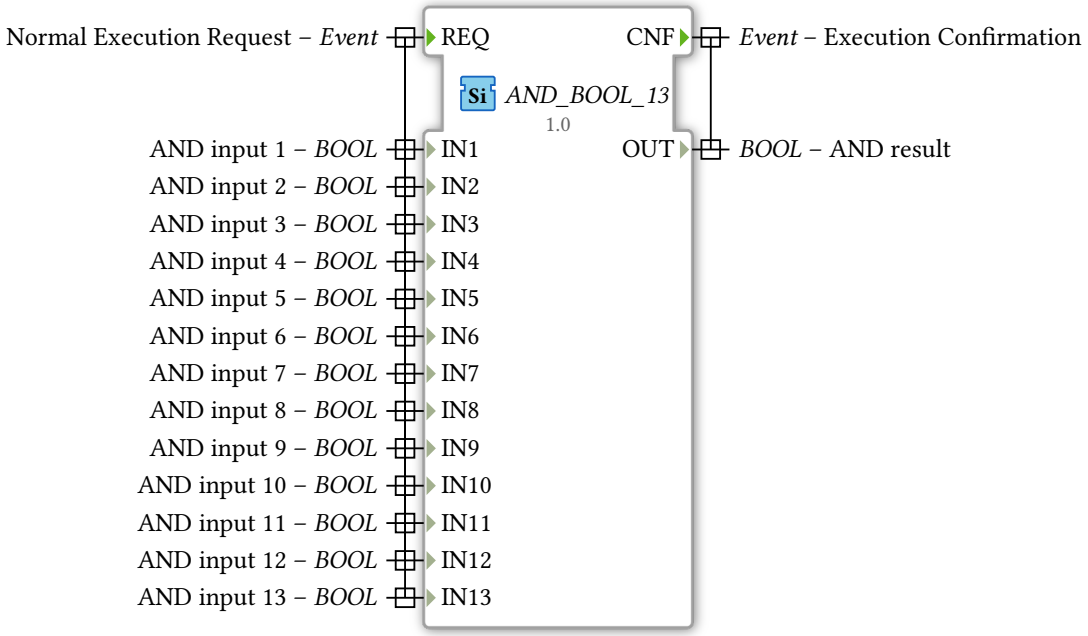
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_13

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_13` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 13 separate BOOL inputs (`IN1` to `IN13`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 13 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 13 data inputs (`IN1` to `IN13`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.
- **IN12 (AND input 12):** Boolean input 12.
- **IN13 (AND input 13):** Boolean input 13.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 13 inputs. The output is TRUE only if **all** inputs `IN1` to `IN13` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 13 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN13` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 13 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_13`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_13` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_13` is a simple, event-driven block for the AND combination of 13 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

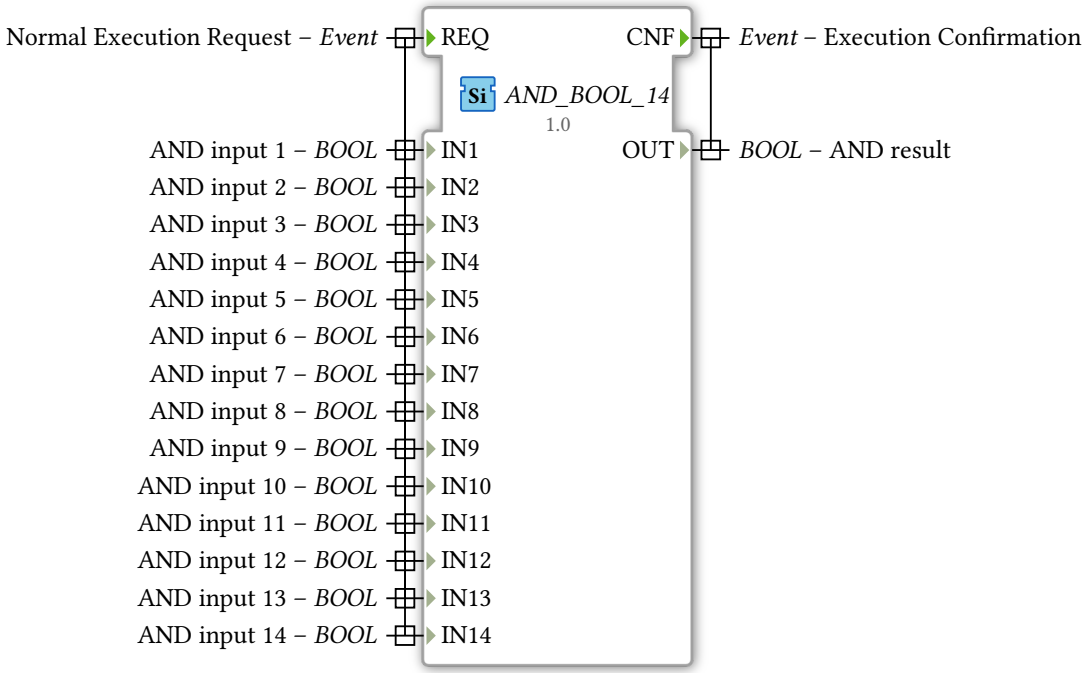
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_14

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_14` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 14 separate BOOL inputs (`IN1` to `IN14`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 14 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 14 data inputs (`IN1` to `IN14`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.
- **IN12 (AND input 12):** Boolean input 12.
- **IN13 (AND input 13):** Boolean input 13.
- **IN14 (AND input 14):** Boolean input 14.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 14 inputs. The output is TRUE only if **all** inputs `IN1` to `IN14` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 14 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN14` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 14 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_14`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_14` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_14` is a simple, event-driven block for the AND combination of 14 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

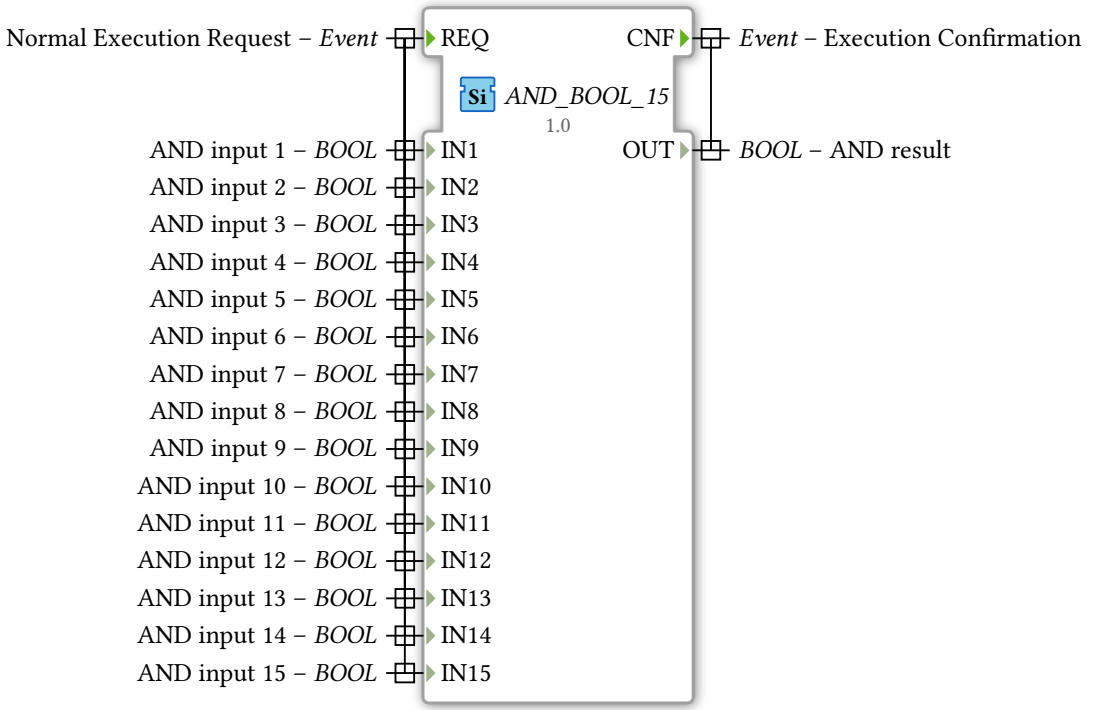
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_15

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_15` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 15 separate BOOL inputs (`IN1` to `IN15`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 15 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 15 data inputs (`IN1` to `IN15`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.

- **IN12 (AND input 12):** Boolean input 12.
- **IN13 (AND input 13):** Boolean input 13.
- **IN14 (AND input 14):** Boolean input 14.
- **IN15 (AND input 15):** Boolean input 15.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 15 inputs. The output is TRUE only if **all** inputs `IN1` to `IN15` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 15 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN15` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 15 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_15`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_15` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_15` is a simple, event-driven block for the AND combination of 15 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

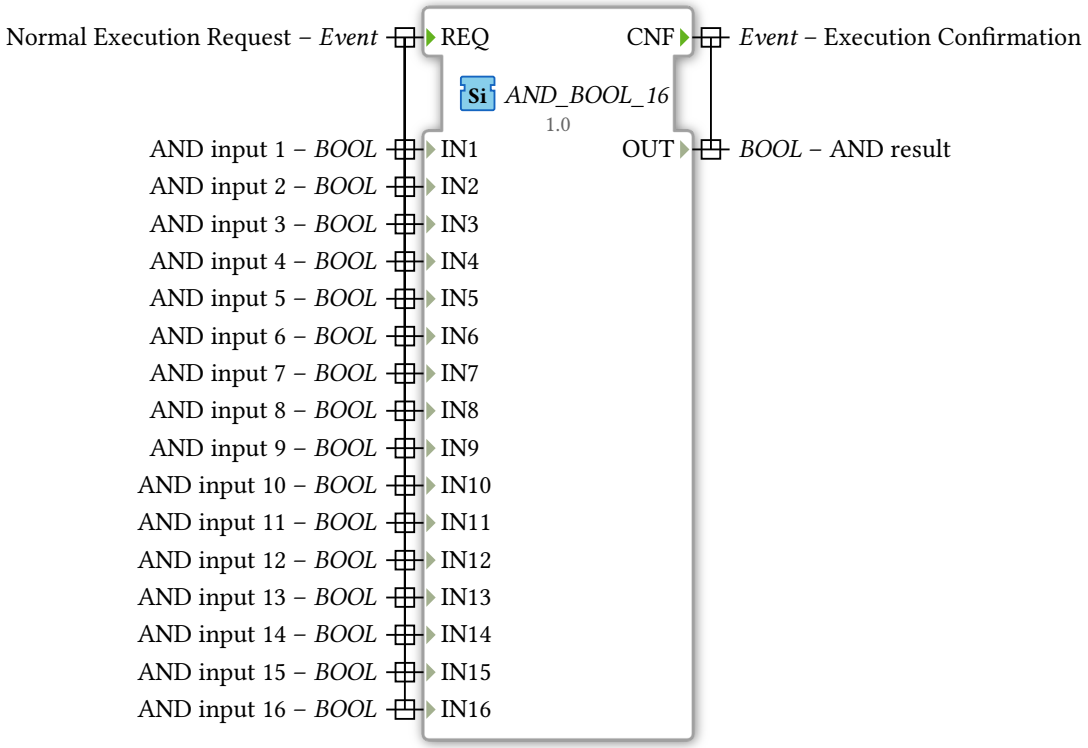
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_16

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_16` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 16 separate BOOL inputs (`IN1` to `IN16`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 16 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 16 data inputs (`IN1` to `IN16`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.
- **IN12 (AND input 12):** Boolean input 12.
- **IN13 (AND input 13):** Boolean input 13.
- **IN14 (AND input 14):** Boolean input 14.
- **IN15 (AND input 15):** Boolean input 15.
- **IN16 (AND input 16):** Boolean input 16.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 16 inputs. The output is TRUE only if **all** inputs `IN1` to `IN16` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 16 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN16` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 16 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_16`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_16` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_16` is a simple, event-driven block for the AND combination of 16 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

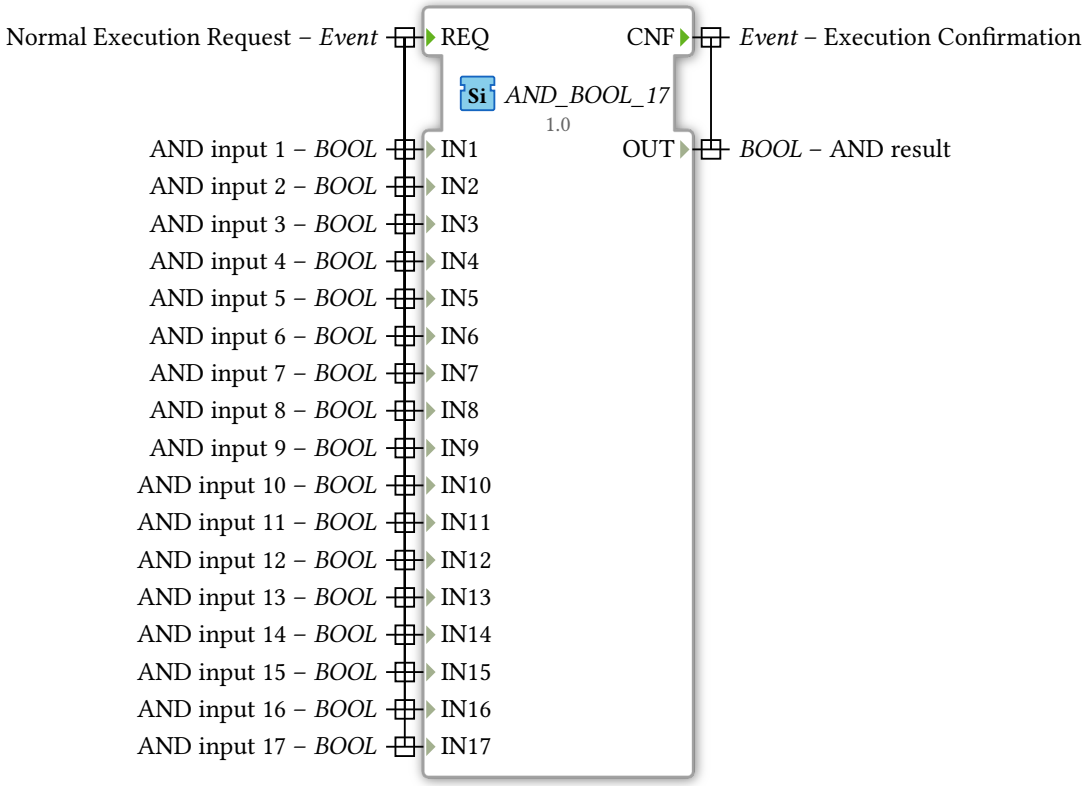
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_17

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_17` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 17 separate BOOL inputs (`IN1` to `IN17`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 17 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 17 data inputs (`IN1` to `IN17`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.
- **IN12 (AND input 12):** Boolean input 12.
- **IN13 (AND input 13):** Boolean input 13.
- **IN14 (AND input 14):** Boolean input 14.
- **IN15 (AND input 15):** Boolean input 15.
- **IN16 (AND input 16):** Boolean input 16.
- **IN17 (AND input 17):** Boolean input 17.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 17 inputs. The output is TRUE only if **all** inputs `IN1` to `IN17` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 17 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN17` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 17 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_17`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_17` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_17` is a simple, event-driven block for the AND combination of 17 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

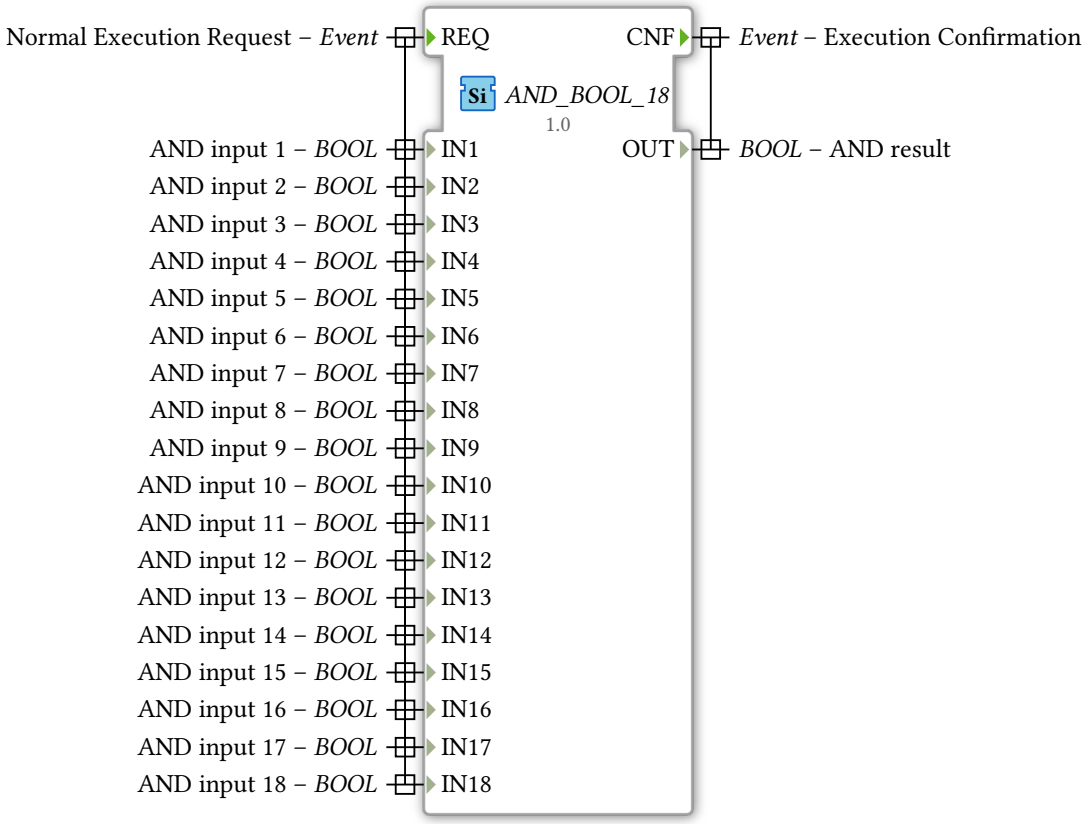
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_18

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_18` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 18 separate BOOL inputs (`IN1` to `IN18`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 18 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 18 data inputs (`IN1` to `IN18`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.
- **IN12 (AND input 12):** Boolean input 12.
- **IN13 (AND input 13):** Boolean input 13.
- **IN14 (AND input 14):** Boolean input 14.
- **IN15 (AND input 15):** Boolean input 15.
- **IN16 (AND input 16):** Boolean input 16.
- **IN17 (AND input 17):** Boolean input 17.
- **IN18 (AND input 18):** Boolean input 18.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 18 inputs. The output is TRUE only if **all** inputs `IN1` to `IN18` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 18 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN18` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 18 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_18`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_18` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_18` is a simple, event-driven block for the AND combination of 18 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

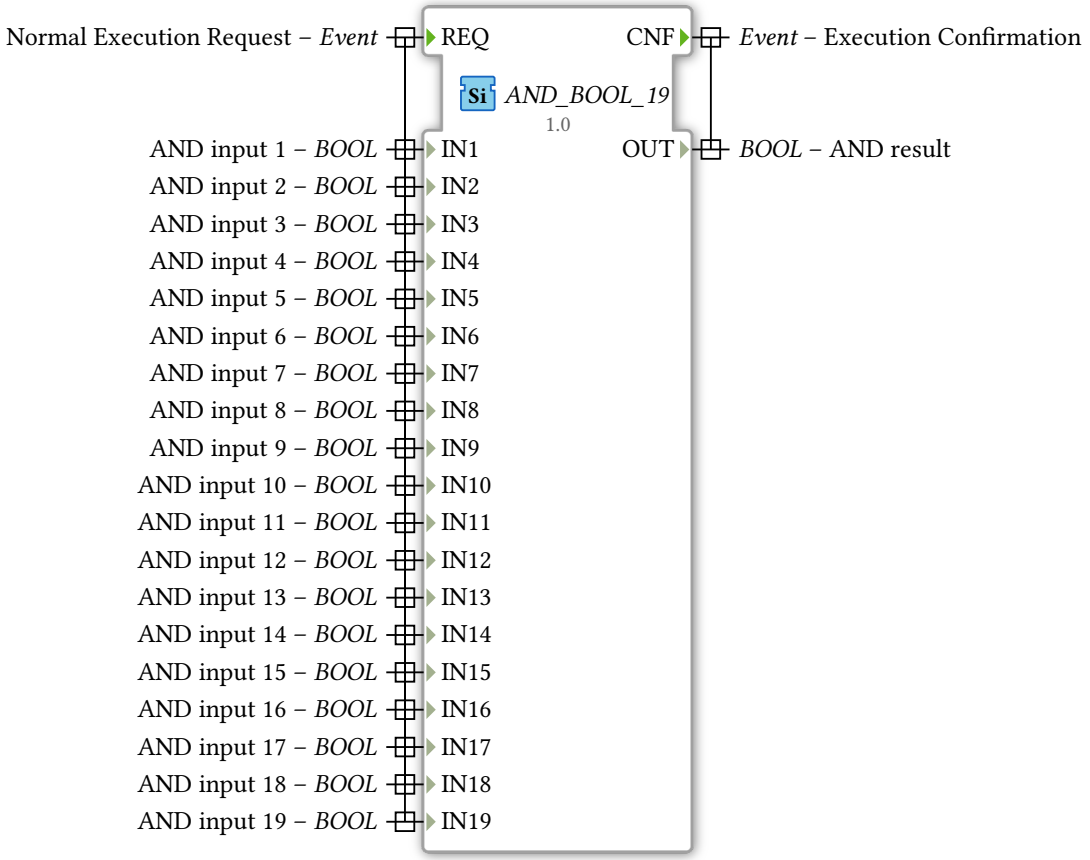
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_19

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_19` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 19 separate BOOL inputs (`IN1` to `IN19`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 19 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 19 data inputs (`IN1` to `IN19`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.
- **IN12 (AND input 12):** Boolean input 12.
- **IN13 (AND input 13):** Boolean input 13.
- **IN14 (AND input 14):** Boolean input 14.
- **IN15 (AND input 15):** Boolean input 15.
- **IN16 (AND input 16):** Boolean input 16.
- **IN17 (AND input 17):** Boolean input 17.
- **IN18 (AND input 18):** Boolean input 18.

- **IN19 (AND input 19):** Boolean input 19.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 19 inputs. The output is TRUE only if **all** inputs `IN1` to `IN19` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 19 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN19` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 19 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_19`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_19` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_19` is a simple, event-driven block for the AND combination of 19 Boolean signals with a fixed interface.

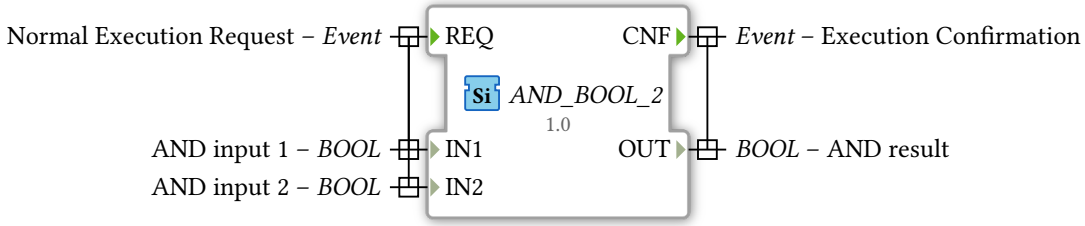
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_2

• – ◦ • – ◦ • – ◦ •

The function block `AND_BOOL_2` is a standard function block for calculating the logical AND operation (conjunction) of two Boolean input values. It complies with the IEC 61131-3 standard for basic Boolean functions and is designed for use in the 4diac IDE. The function block waits for a trigger event, executes the operation, and signals the completion of the calculation with an acknowledgment event.



The interface of the `AND_BOOL_2` function block is simple and clearly structured, consisting of an event input, an event output, and the associated data connections.

- **REQ (Normal Execution Request):** This event triggers the execution of the function block. Upon its arrival, the current values at data inputs `IN1` and `IN2` are read, and the AND operation is calculated.
- **CNF (Execution Confirmation):** This event is generated after the calculation is complete. It signals that the result has been output to data output `OUT` and is now valid.

- **IN1 (AND input 1):** First Boolean operand (Type: `BOOL`).
- **OUT (AND result):** Result of the logical AND operation of `IN1` and `IN2` (Type: `BOOL`). The output value is `TRUE` (1) if both `IN1` and `IN2` have the value `TRUE`. In all other cases, the output is `FALSE` (0).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block has no adapter interfaces.

The `AND_BOOL_2` block functions deterministically and follows classic Boolean logic:

1. **Trigger:** The occurrence of the event `REQ` starts the execution.
2. **Read Input:** The current values of the data inputs `IN1` and `IN2` are retrieved.
3. **Calculation:** The logical AND operation `OUT := IN1 AND IN2` is performed.
4. **Output:** The calculated result is written to the data output `OUT`.
5. **Confirmation:** Immediately after output, the confirmation event `CNF` is generated to inform downstream blocks about the availability of the new result.

The operation can be described by the following truth table:

IN1	IN2	OUT (IN1 AND IN2)
0	0	0
0	1	0
1	0	0
1	1	1

- **Generic Block:** The block is implemented as an instance of a generic AND block (`GEN_AND`), which can be specialized for different data types. In this specific instance, it is defined for the data type `BOOL`.

| IN1 | IN2 | OUT (IN1 AND IN2) | | 0 | 0 | 0 | | 1 | 1 | 1 |

Generic Block: The block is implemented as an instance of a generic AND block (`GEN_AND`), which can be specialized for different data types. In this specific instance, it is defined for the data type `BOOL`.

| IN1 | IN2 | OUT (IN1 AND IN2) ... * **Event-driven:** As is typical for 4diac function blocks, execution is strictly event-driven. A change in the input values alone does not trigger a calculation; this only occurs upon the arrival of the `REQ` event.

- **Low latency:** The execution time is negligible, so the `CNF` event follows the `REQ` event practically immediately.

The function block has no internal state in the sense of a memory for previous values. It is a purely combinational block whose output depends solely on the current input values at the time of triggering. Its "state" is therefore always "waiting for REQ" or "executing," with the execution phase being very short.

The `AND_BOOL_2` block is a fundamental building block in control and automation technology. Typical applications include:

- **Safety Conditions:** Linking multiple safety signals (e.g., "Safety door closed" AND "Emergency stop not pressed") to generate a machine enable signal.
- **Two-Hand Operation:** Ensuring that two buttons must be pressed simultaneously to initiate a hazardous operation.
- **Chaining of Switching Conditions:** Combining multiple sensor or switch states to control an actuator signal (e.g., "Material present" AND "Position reached" -> gripper close).
- **Signal Masking:** Selectively switching a signal (`DATA`) through or blocking it with an enable signal (`ENABLE`) using `OUT := DATA AND ENABLE`.

- **`AND_n` (generic):** The generic `GEN_AND` building block can be configured for more than two inputs and other data types (e.g., `WORD`, `DWORD`). `AND_BOOL_2` is a specific instance of this, fixed to two Boolean inputs, which can be used more easily and directly in Boolean logic circuits. See: AND_2
- **`OR_BOOL_2` / `XOR_BOOL_2`:** These blocks implement other basic Boolean operations (OR and exclusive OR, respectively). They have the same event-driven interface (`REQ`/`CNF`), but perform a different logical operation on the inputs `IN1` and `IN2`.
- **Combinatorial Logic in SFC/ST:** The AND function could also be directly encoded in Structured Text (ST) using the `&` or `AND` operator. The advantage of the `AND_BOOL_2` function block lies in its standardized, reusable, and graphically linkable form within the 4diac function block network (FBN).
- **Combinatorial Logic in SFC/ST:** The AND function could also be directly encoded in Structured Text (ST) using the `&` or `AND` operator.
- Exercise_002a4
- Exercise_002b3
- Exercise_006a3

The `AND_BOOL_2` function block is an essential, reliable, and easy-to-use building block for implementing Boolean logic in 4diac-based control applications. Its clear, event-driven interface and deterministic functionality make it the ideal building block for safety operations, control conditions, and general logical operations. It is the first choice for pure Boolean binary operations.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Functionality

Application Scenarios

State Overview

Technical Features

⚖️ Vergleich mit ähnlichen Bausteinen

🛠️ Zugehörige Übungen

Conclusion

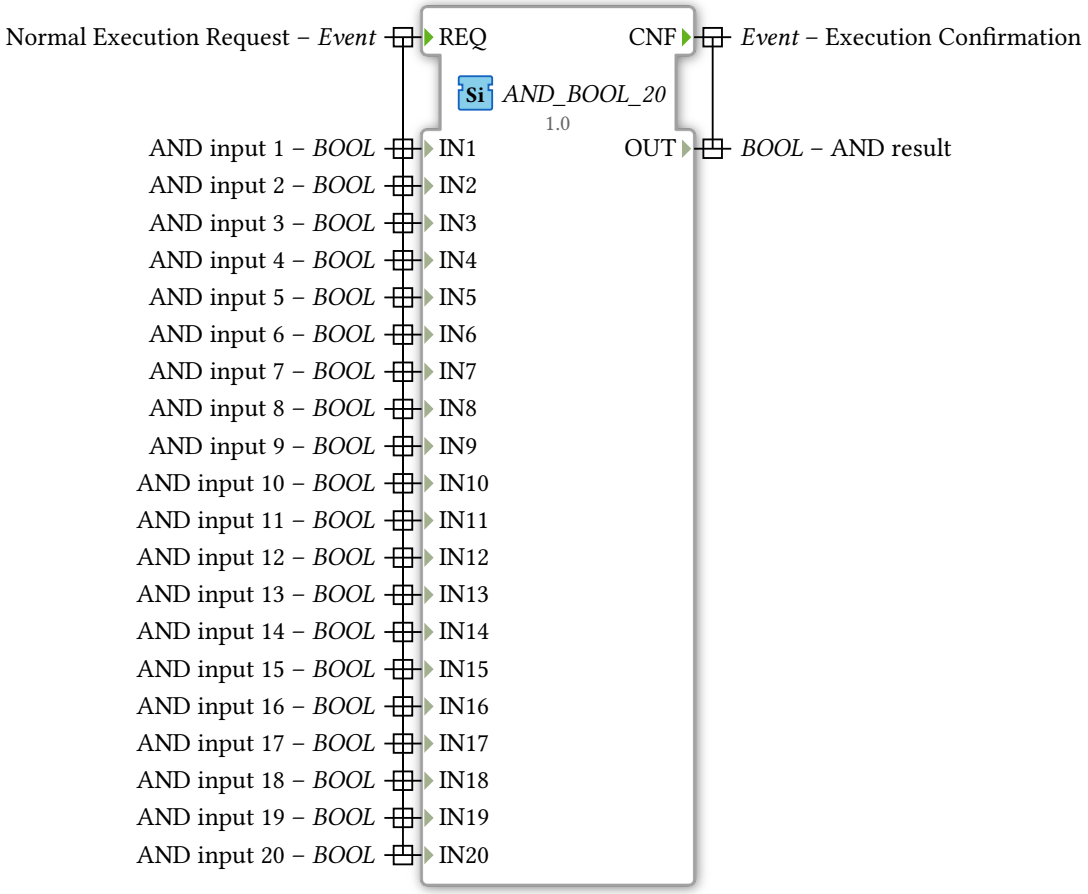
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

AND_BOOL_20

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_20` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 20 separate BOOL inputs (`IN1` to `IN20`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 20 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 20 data inputs (`IN1` to `IN20`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.
- **IN12 (AND input 12):** Boolean input 12.
- **IN13 (AND input 13):** Boolean input 13.
- **IN14 (AND input 14):** Boolean input 14.
- **IN15 (AND input 15):** Boolean input 15.
- **IN16 (AND input 16):** Boolean input 16.
- **IN17 (AND input 17):** Boolean input 17.

- **IN18 (AND input 18):** Boolean input 18.
- **IN19 (AND input 19):** Boolean input 19.
- **IN20 (AND input 20):** Boolean input 20.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 20 inputs. The output is TRUE only if **all** inputs `IN1` to `IN20` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 20 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN20` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 20 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_20`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_20` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_20` is a simple, event-driven block for the AND combination of 20 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

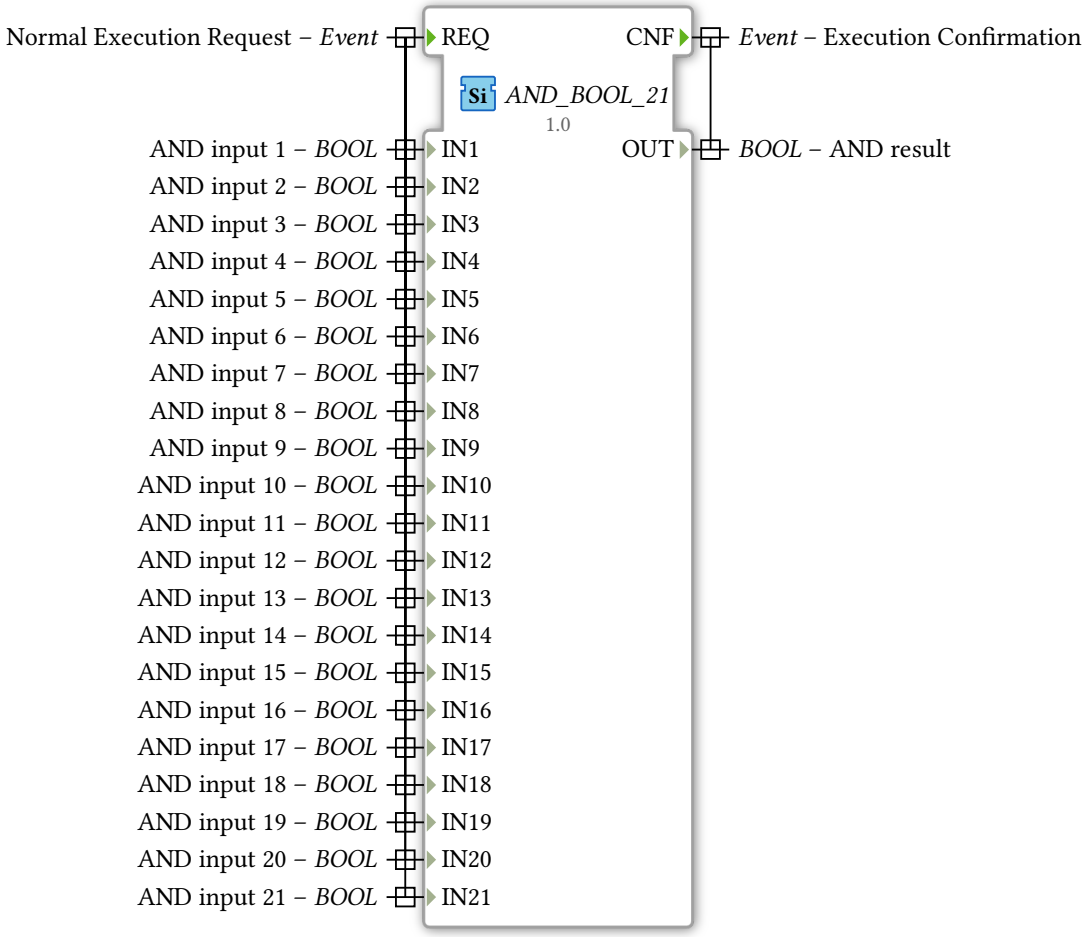
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_21

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_21` is a generic block for calculating the logical AND operation (classification `standard boolean function`, IEC 61131-3). It combines 21 separate BOOL inputs (`IN1` to `IN21`) and provides the result at the output `OUT`. It extends the standard AND_BOOL family (previously up to `AND_BOOL_10`) with the variant that has 21 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 21 data inputs (`IN1` to `IN21`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (AND input 1):** Boolean input 1.
- **IN2 (AND input 2):** Boolean input 2.
- **IN3 (AND input 3):** Boolean input 3.
- **IN4 (AND input 4):** Boolean input 4.
- **IN5 (AND input 5):** Boolean input 5.
- **IN6 (AND input 6):** Boolean input 6.
- **IN7 (AND input 7):** Boolean input 7.
- **IN8 (AND input 8):** Boolean input 8.
- **IN9 (AND input 9):** Boolean input 9.
- **IN10 (AND input 10):** Boolean input 10.
- **IN11 (AND input 11):** Boolean input 11.
- **IN12 (AND input 12):** Boolean input 12.
- **IN13 (AND input 13):** Boolean input 13.
- **IN14 (AND input 14):** Boolean input 14.
- **IN15 (AND input 15):** Boolean input 15.
- **IN16 (AND input 16):** Boolean input 16.

- **IN17 (AND input 17):** Boolean input 17.
- **IN18 (AND input 18):** Boolean input 18.
- **IN19 (AND input 19):** Boolean input 19.
- **IN20 (AND input 20):** Boolean input 20.
- **IN21 (AND input 21):** Boolean input 21.

Data Outputs

- **OUT (AND result):** The result of the AND operation of all 21 inputs. The output is TRUE only if **all** inputs `IN1` to `IN21` are TRUE, otherwise FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 21 inputs are read and the operation `OUT = IN1 AND IN2 AND ... AND IN21` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_AND_BOOL'`; this instance is fixed to 21 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Enable and safety interlocks:** All conditions (e.g. door contacts, emergency stop, operational readiness) must be met before an action may start.
- **Multiple conditions:** Check whether many prerequisites are fulfilled at the same time without having to cascade several smaller AND blocks.

⚖️ Comparison with Similar Function Blocks

- `OR_BOOL_21`: Performs the OR operation: the result is TRUE as soon as at least one input is TRUE, while `AND_BOOL_21` requires all inputs to be TRUE.
- `AND_BOOL_10`: Sibling block of the family with 10 inputs.

Conclusion

`AND_BOOL_21` is a simple, event-driven block for the AND combination of 21 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

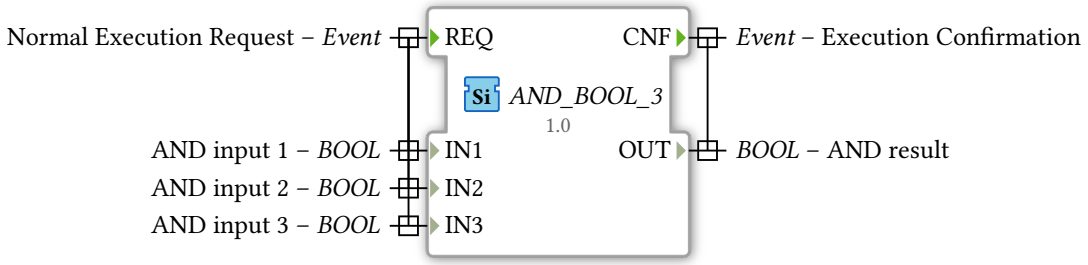
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_3

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_3` is a standardized building block for calculating the logical AND operation for three Boolean input signals. It complies with the IEC 61131-3 standard for basic Boolean functions and operates according to the 4diac real-time execution control model, where the calculation is triggered by an incoming event.



Interface Structure

The interface of the `AND_BOOL_3` block is clear and simple, consisting of one event input, one event output, three data inputs, and one data output.

Event Inputs

- **REQ (Normal Execution Request):** This incoming event triggers the execution of the function block. Upon the arrival of `REQ`, the current values at data inputs `IN1`, `IN2`, and `IN3` are read, and the logical AND operation is performed.

Event Outputs

- **CNF (Execution Confirmation):** This event is generated after the calculation is complete. It signals that the result has been output to data output `OUT` and is now valid.

Data Inputs

- **IN1 (BOOL):** First Boolean input signal for the AND operation.
- **IN2 (BOOL):** Second Boolean input signal for the AND operation.
- **IN3 (BOOL):** Third Boolean input signal for the AND operation.

Data Outputs

- **OUT (BOOL):** The result of the logical AND operation of all three inputs. `OUT` is only `TRUE` (1) if **all** inputs `IN1`, `IN2`, and `IN3` are simultaneously `TRUE`. In all other cases, `OUT` is `FALSE` (0).

Adapters

This function block has no adapter interfaces.

Functionality

The functionality is deterministic and follows a simple sequence:

1. **Trigger:** The incoming event `REQ` activates the block.
2. **Reading the Inputs:** The current values of the three Boolean data inputs (`IN1`, `IN2`, `IN3`) are read.
3. **Calculation:** The logical AND operation is performed: `OUT := IN1 AND IN2 AND IN3`.
4. **Output:** The calculated result is written to the data output `OUT`.
5. **Acknowledgement:** The output event `CNF` is generated to signal the successful completion of the operation to subsequent blocks.

Technical Features

- **Generic Function Block:** This block is marked as a generic function block (`GEN_AND`), meaning it can serve as a basis for creating similar AND blocks with a different number of inputs.
- **Real-Time Capable:** Due to its execution control model (event-driven control), this block is suitable for deterministic real-time applications.
- **Simple Logic:** Contains no internal states or memory. The output is a pure combinatorial function of the current inputs.

State Overview

The `AND_BOOL_3` block is a **combinatorial** function block without internal state (stateless). Its behavior on each call (`REQ`) depends solely on the current values at the data inputs. There are no prior calculations or states that influence the result.

Application Scenarios

Typical applications include control logics where an action may only be executed if several safety or enabling conditions are met simultaneously:

- **Two-Hand Operation:** A machine only starts when three separate buttons are pressed simultaneously (`IN1`, `IN2`, `IN3`).
- **Safety Circuit:** A process is only activated if the safety door is protected (`IN1`), the emergency stop bypass is activated (`IN2`), and material is present (`IN3`).
- **Condition Chaining:** As part of a more complex logical equation within an algorithm.

⚖️ Comparison with similar building blocks

- **`AND_BOOL_2`:** Performs the AND operation for only two inputs. `AND_BOOL_3` is a direct extension for three inputs.
- **`AND_n_BOOL` (theoretical):** A generic block for a variable number of inputs. `n` is a specific, hard-wired instance of `AND_BOOL_3`. See: AND_3
- **`OR_BOOL_3`:** Calculates the logical OR operation (output is `TRUE` if at least one input is `TRUE`). This is the complementary function.
- **Combinatorial Logic in SFC/ST:** The same function could also be implemented in Structured Text (ST) as `OUT := IN1 & IN2 & IN3;` or in an SFC action. The advantage of the function block is its reusability, clear encapsulation, and easy integration into the 4diac event system.

Conclusion

The `AND_BOOL_3` is a fundamental, reliable, and easy-to-use building block for Boolean logic in 4diac-based control applications. Its strict event-driven control ensures deterministic behavior, and its clear interface allows for intuitive integration into function block networks. For all control tasks requiring a triple AND operation, this block is the standard solution.

🌐 Related topic subpages on ms-muc-docs.de

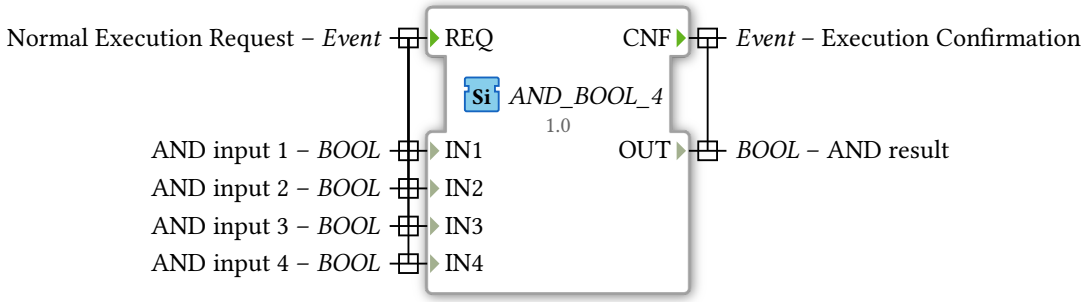
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AND_BOOL_4

• – ◦ • – ◦ • – ◦ •

The function block `AND_BOOL_4` is a standard function block for calculating the logical AND operation. It performs a bitwise AND operation on four Boolean input values. The block is implemented as a generic function block and complies with the IEC 61131-3 standard for basic Boolean functions.



- **REQ** (Normal Execution Request): This event triggers the execution of the function block. It is associated with all four data inputs (`IN1` to `IN4`).
- **CNF** (Execution Confirmation): This event signals the completion of the calculation. It is output along with the result at data output `OUT`.
- **IN1** (BOOL): First operand for the AND operation.
- **IN2** (BOOL): Second operand for the AND operation.
- **IN3** (BOOL): Third operand for the AND operation.
- **IN4** (BOOL): Fourth operand for the AND operation.
- **OUT** (BOOL): Result of the logical AND operation of all four inputs. The output is only `TRUE` if **all** inputs `IN1`, `IN2`, `IN3`, and `IN4` have the value `TRUE`.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use any adapters.

Whenever a `REQ` event occurs, the function block reads the current values at the four Boolean data inputs. It then calculates the logical AND of these four values according to the following truth table:

IN1	IN2	IN3	IN4	OUT
0	0	0	0	0
0	0	0	1	0
…	…	…	…	…
1	1	1	1	1

The result is set at the data output `OUT`, and simultaneously the `CNF` event is triggered to signal the completion of the operation.

- **Generic Implementation:** The function block is marked as a generic function block (`GEN_AND`), which means it can serve as the basis for similar function blocks with a different number of inputs.
- **Event-driven execution:** The operation is triggered exclusively by an incoming event (`REQ`), which is typical for the function block (FB) style in IEC 61499.
- **Standard compliance:** The block is classified as a standard Boolean function according to IEC 61131-3.

The function block does not have an internal state in the sense of a memory. Its behavior is purely combinatorial and depends only on the current input values at the time of triggering. The only "state" is the idle state, in which it waits for the next `REQ` event.

- **Safety chain:** Linking several safety switches (e.g., emergency stop, guardrail, two-hand control), all of which must be closed to start a machine.
- **Masking:** Selectively activates a function only if multiple enable signals are present.
- **AND (2 Inputs):** A basic AND block with only two inputs. `AND_BOOL_4` consolidates four such operations into a single block, improving diagram readability and maintainability. See: AND_4
- **OR_BOOL_4:** Performs a logical OR operation where the result is `TRUE` if at least one input is `TRUE`. This is the logical counterpart to AND.
- **GEN_AND:** The generic base class on which `AND_BOOL_4` is built. It can be configured to create AND blocks with any number of inputs.

The `AND_BOOL_4` function block is a simple, robust, and standards-compliant block for processing logic signals. Its clear, event-driven interface and fixed number of four inputs make it particularly suitable for control tasks where multiple conditions must be met simultaneously. As part of the `iec61131::bitwiseOperators` library, it is a fundamental component for designing control logic in 4diac.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

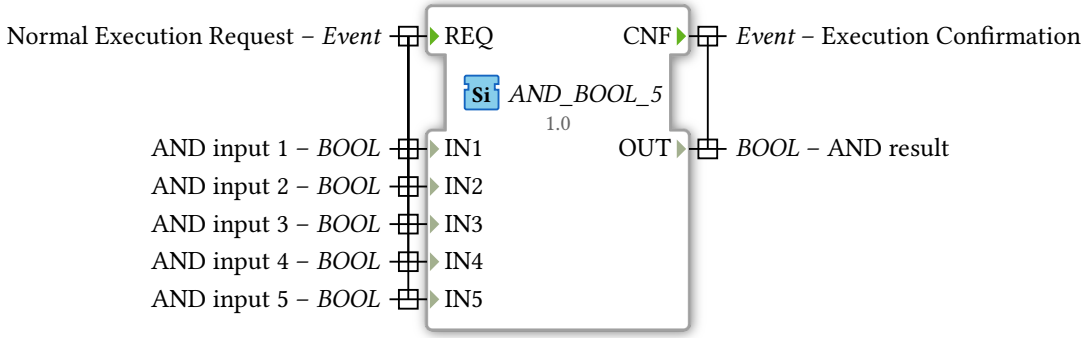
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

AND_BOOL_5

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_5` is a standard function block for calculating the logical AND operation. It performs a bitwise AND operation on five separate Boolean input values. The block follows the real-time and event-driven processing model of IEC 61499, where the calculation is triggered by an incoming event and confirmed by an outgoing event.



Interface Structure

Event Inputs

- **REQ**: Triggers the execution of the function block. Upon arrival of this event, all five inputs (`IN1` to `IN5`) are read and the AND operation is calculated.

Event Outputs

- **CNF**: Confirms completion of the operation. This event is output after the result is calculated and includes the calculated value `OUT`.

Data Inputs

- **IN1**: Boolean input 1 for the AND operation.
- **IN2**: Boolean input 2 for the AND operation.
- **IN3**: Boolean input 3 for the AND operation.
- **IN4**: Boolean input 4 for the AND operation.
- **IN5**: Boolean input 5 for the AND operation.

Data Outputs

- **OUT**: The Boolean result of the AND operation of all five inputs. The output is only `TRUE` if **all** inputs `IN1` to `IN5` have the value `TRUE`.

Adapters

This function block does not use any adapters.

Operation

The operation is deterministic and event-driven:

1. The incoming event `REQ` activates the block.
2. The current values of all five Boolean data inputs (`IN1` to `IN5`) are read.
3. The logical AND operation is calculated: `OUT = IN1 AND IN2 AND IN3 AND IN4 AND IN5`.
4. The result is provided at data output `OUT`.
5. The output event `CNF` is triggered along with the new value of `OUT` to signal the completion of the operation.

Technical Features

- **Generic Block**: The block is implemented as an instance of a generic AND block (`GEN_AND`), which can be parameterized for a variable number of inputs. This specific instance is defined for exactly five inputs.
- **Event Chain**: The execution strictly follows the REQ/CNF pattern, which allows for easy integration into event-driven control sequences.
- **No Internal State**: The block is memory-free (combinatory). The output signal depends solely on the current input values at the time of the `REQ` request.

State Overview

The function block has no persistent internal state. Its behavior is purely combinatorial and is fully described by the interface events and data. The "state" is the transition from waiting (`REQ` wait) to active (perform calculation and trigger `CNF`).

Application Scenarios

Typical applications include safety and enable chains in automation technology, where multiple conditions must be met simultaneously:

- Monitoring whether all safety guards of a machine are closed.
- Checking whether all required media (pressure, vacuum, cooling) are available before a process is started.
- Linking multiple sensor or switch states to a common enable logic.

⚖️ Comparison with similar building blocks

- **`AND_BOOL_2` / `AND_BOOL_3` etc.**: Basic AND blocks with two, three, or four inputs. `AND_BOOL_5` extends this logic to five inputs without the need to chain multiple blocks. See: AND_5
- **`GEN_AND`**: The underlying generic block. `AND_BOOL_5` is a pre-configured, unchangeable instance of it with a fixed number of inputs.
- **OR blocks**: Perform a logical OR operation. While `AND_BOOL_5` outputs `TRUE` only if all `TRUE` inputs are present, OR gates output `TRUE` even if only one `TRUE` input is present.

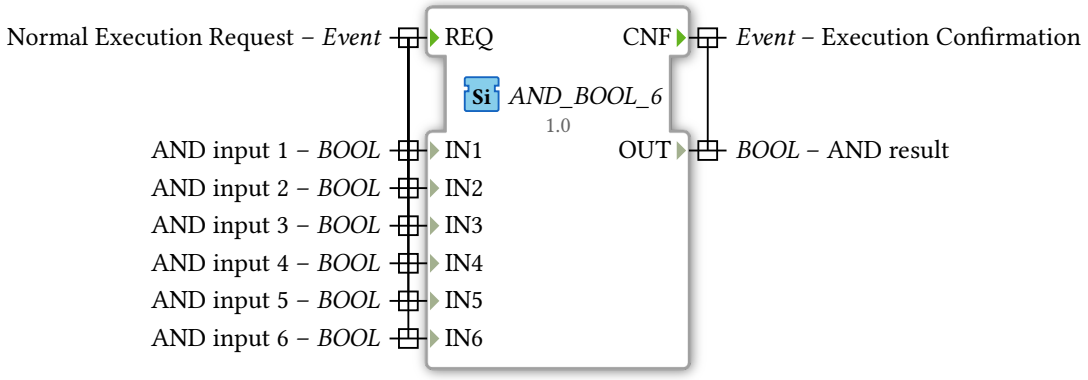
Conclusion

The `AND_BOOL_5` is a robust and easy-to-use standard function block for five-input logic gates. Its event-driven, stateless architecture makes it highly predictable and easy to integrate into more complex IEC 61499 applications. It offers an optimal and readable solution for applications that need to link exactly five conditions.

AND_BOOL_6

• − ◦ • − ◦ • − ◦ •

The function block `AND_BOOL_6` is a standard function block for calculating the logical AND operation. It performs a bitwise AND operation using six separate Boolean inputs. This block belongs to the category of standard Boolean functions and is designed for use in control applications according to IEC 61131-3.



- **REQ** (Normal Execution Request): This event triggers the calculation of the AND operation. Upon receiving `REQ`, the values of all six data inputs (IN1 to `IN6`) are read and processed.
- **CNF** (Execution Confirmation): This event signals the completion of the calculation. It is output along with the calculated result at data output `OUT`.
- **IN1** (BOOL): AND input 1.
- **IN2** (BOOL): AND input 2.
- **IN3** (BOOL): AND input 3.
- **IN4** (BOOL): AND input 4.
- **IN5** (BOOL): AND input 5.
- **IN6** (BOOL): AND input 6.
- **OUT** (BOOL): Result of the AND operation. The output is only `TRUE` if **all** six inputs (`IN1` to `IN6`) have the value `TRUE`. In all other cases, the output is `FALSE`.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

This function block has no adapter interfaces.

Functionality

The functionality is deterministic and simple: Upon each occurrence of the event `REQ`, the logical state of all six inputs is evaluated. The result of the AND operation is provided at the data output `OUT`, and simultaneously, the acknowledgment event `CNF` is triggered. The logical operation corresponds to the formula: OUT = IN1 AND IN2 AND IN3 AND IN4 AND IN5 AND IN6`

- **Generic Block:** The block is implemented as a generic block (`GEN_AND`), meaning it is a specific instance of a more general AND function block family.
- **Fixed Number of Inputs:** Unlike blocks with a variable number of inputs, this block has exactly six fixed inputs.
- **Event-driven:** The operation is executed exclusively by an incoming event (`REQ`), which is typical for the function block (FB) architecture in 4diac.

The block has no internal state or memory. Its behavior is purely combinatorial and depends solely on the current values of the inputs at the time of triggering by `REQ`.

Typical applications include safety and enable chains in control logic, where multiple conditions must be met simultaneously before an action is executed.

- **Machine Start:** A machine will only start if all safety doors are closed (`TRUE`) and all emergency stop buttons are not pressed (`TRUE`).

- **Multiple Sensor Polling:** Evaluates multiple sensors, all of which must provide a signal to confirm a position or state.
- **AND (with 2 inputs):** The standard `AND` block typically has only two inputs. `AND_BOOL_6` offers a specialized, more compact representation for applications with exactly six inputs, without the need to chain multiple blocks together. See: AND_6
- **GEN_AND (Generic AND Block):** `AND_BOOL_6` is a concrete instance of this generic type. Other instances could have a different number of inputs (e.g., `AND_BOOL_4`).
- **OR_BOOL_6:** A similar block, but it performs a logical OR operation where the result is `TRUE` if at least one input is `TRUE`.

The `AND_BOOL_6` function block is a reliable and easy-to-use block for six-input logical AND operations. Its event-driven, stateless nature makes it predictable and easily integrated into the 4diac architecture. It is the optimal choice when a fixed set of six conditions must be met simultaneously to trigger a subsequent action.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

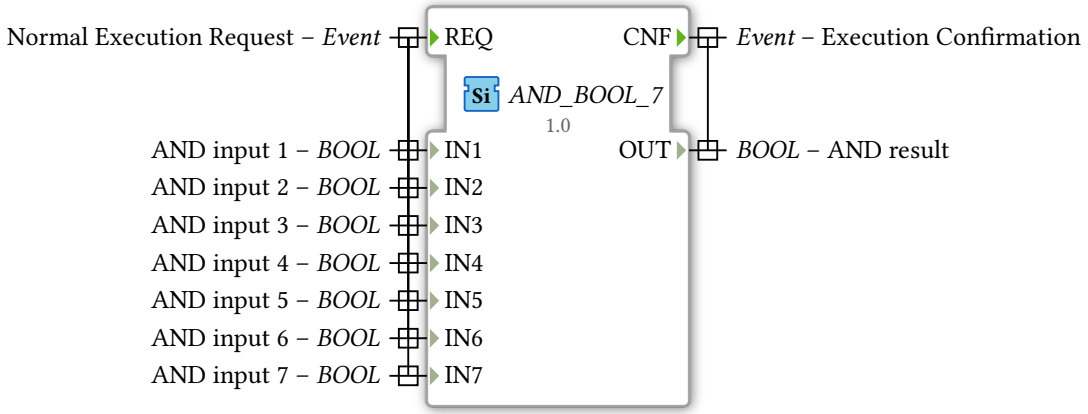
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

AND_BOOL_7

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_7` is a generic function block for calculating the logical AND operation. It performs a bitwise AND operation across seven separate BOOL inputs and outputs the result on a single BOOL output. The block follows the ECC execution model, where a calculation is triggered by an incoming event and confirmed by an outgoing event.



Interface Structure

Event Inputs

- **REQ**: Triggers the calculation of the AND operation. Upon arrival of this event, all data inputs (`IN1` to `IN7`) are read and processed.

Event Outputs

- **CNF**: Confirms completion of the calculation. This event is generated after the calculation and returns the result `OUT`.

Data Inputs

- **IN1**: AND input 1 (BOOL)
- **IN2**: AND input 2 (BOOL)
- **IN3**: AND input 3 (BOOL)
- **IN4**: AND input 4 (BOOL)
- **IN5**: AND input 5 (BOOL)
- **IN6**: AND input 6 (BOOL)
- **IN7**: AND input 7 (BOOL)

Data Outputs

- **OUT**: Result of the AND operation (BOOL). The output is only `TRUE` if **all** seven inputs (`IN1` to `IN7`) have the value `TRUE`. In all other cases, the output is `FALSE`.

Adapters

This function block has no adapter interfaces.

Operation

The operation is deterministic and follows Boolean algebra. Upon each occurrence of the event `REQ`, the following operation is performed:

OUT := IN1 AND IN2 AND IN3 AND IN4 AND IN5 AND IN6 AND IN7`

Immediately after the result is calculated, the confirmation event `CNF` is output along with the current value of `OUT`.

- **Generic Block**: This block is marked as a generic block (`GEN_AND`). This means it can serve as a template for the automatic generation of similar blocks with a different number of inputs.
- **Event-driven**: The operation is only executed when required by an event, which conserves resources and allows for clear control of the flow.
- **Fixed Number of Inputs**: This specific block has a fixed number of seven inputs. For a different number of operands, a corresponding block (e.g., `AND_BOOL_2`, `AND_BOOL_4`) must be used, or a generic block must be instantiated.

State Overview

The block does not have an internal state in the sense of a memory. Its behavior is purely combinatorial and event-driven:

1. **Idle State**: Waits for the event `REQ`.
2. **Execution State**: Upon `REQ`, all inputs are read, the AND operation is calculated, and the result `OUT` is set.
3. **Acknowledgement State**: The event `CNF` is sent immediately, after which the block returns to the idle state.

Application Scenarios

- **Safety Chain**: Linking multiple safety switches (e.g., emergency stop, safety guard, light barrier), where a machine may only start if all switches are closed (TRUE).
- **Multiple Condition**: Checking whether multiple parallel process steps have been successfully completed before a subsequent process is started.
- **Control Logic**: Implementation of complex AND conditions in sequential or logic controls.

⚖️ Comparison with similar blocks

- **`AND_n_BOOL` family**: Blocks with identical functionality but different numbers of inputs exist (e.g., `AND_BOOL_2`, `AND_BOOL_8`). `AND_BOOL_7` fills the gap for applications requiring exactly seven conditions. See: AND_7
- **`&` (Bit-AND) Operator**: While `AND_BOOL_7` only works with BOOL data types and is event-driven, the `&` operator in Structured Text (ST) can also be applied to byte, word, or DWORD data types and is evaluated cyclically or statement-wise.

- **Serial AND Interlock**: The same result could be achieved by serially concatenating multiple `AND_BOOL_2` blocks. `AND_BOOL_7` offers a more compact and concise solution.

Conclusion

The `AND_BOOL_7` is a specialized, reliable, and easy-to-use function block for the logical AND operation of seven Boolean signals. Its event-driven architecture makes it ideal for integration into control-type flowcharts within the 4diac IDE. For applications with a different number of inputs, the corresponding variant from the `AND_n_BOOL` family should be used.

🌐 Related topic subpages on ms-muc-docs.de

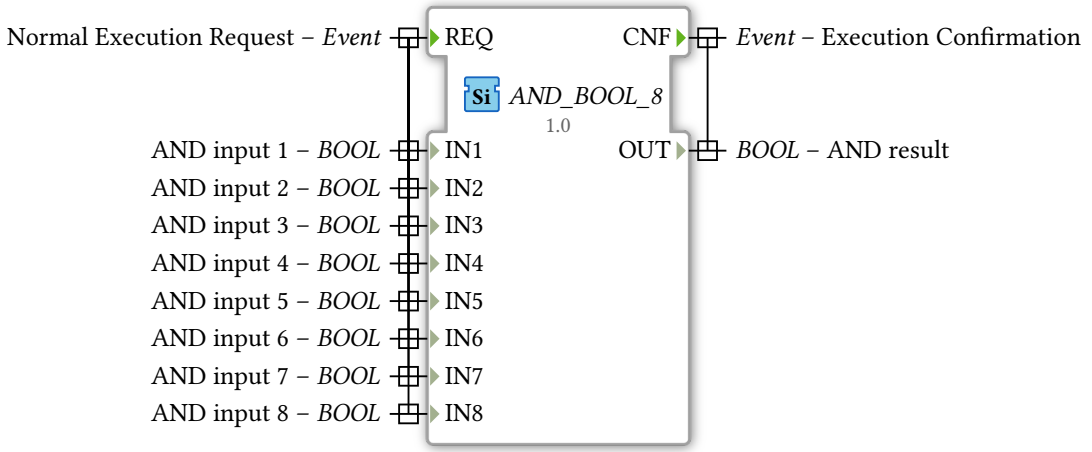
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

AND_BOOL_8

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_8` is a standard function block for calculating the logical AND operation. It performs a bitwise AND operation on eight Boolean input values and outputs the result to a single Boolean output. The function block follows the event-driven execution model of IEC 61499.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the execution of the function block. Upon receipt of this event, all eight data inputs are read, the AND operation is calculated, and the result is provided at output `OUT`. The acknowledgment event `CNF` is then generated.

Event Outputs

- **CNF (Execution Confirmation):** This event is output after the successful calculation and update of output `OUT`. It indicates that a new result is available.

Data Inputs

- **IN1 (BOOL):** AND input 1.
- **IN2 (BOOL):** AND input 2.
- **IN3 (BOOL):** AND input 3.
- **IN4 (BOOL):** AND input 4.
- **IN5 (BOOL):** AND input 5.
- **IN6 (BOOL):** AND input 6.
- **IN7 (BOOL):** AND input 7.
- **IN8 (BOOL):** AND input 8.

Data Outputs

- **OUT (BOOL):** Result of the AND operation of all eight inputs. The output is `TRUE` (1) only if **all** inputs `IN1` to `IN8` have the value `TRUE`. In all other cases, the output is `FALSE` (0).

Adapter

This function block has no adapter interfaces.

Operation

The operation is deterministic and follows classical Boolean logic. When triggered by the event `REQ`, the block evaluates the states of all eight inputs (`IN1` to `IN8`). The logical operation is performed according to the following truth table:

IN1	IN2	IN3	IN4	IN5	IN6	IN7	IN8	OUT
0	X	X	X	X	X	X	X	0
1	0	X	X	X	X	X	X	0
1	1	0	X	X	X	X	X	0
...	...	...	...	...	...	...	...	...
1	1	1	1	1	1	1	1	1

(X = Don't-Care / Any Value)

The result `OUT = IN1 AND IN2 AND IN3 AND IN4 AND IN5 AND IN6 AND IN7 AND IN8` is set immediately after the calculation. Execution is completed by triggering the confirmation event `CNF`, which accompanies the new result at the data output `OUT`.

Technical Features

- **Generic Block:** The block is implemented as an instance of a generic AND template (`GEN_AND`), ensuring a consistent architecture within the library.
- **Fixed Input Count:** Unlike variable or configurable blocks, `AND_BOOL_8` has exactly eight fixed inputs. For a different number of operands, a corresponding block (e.g., `AND_BOOL_2`, `AND_BOOL_16`) must be used.
- **Event-driven execution:** The calculation only takes place upon the arrival of a `REQ` event. Continuous sampling of the inputs without an event does not result in an output update.

State overview

The function block has no internal state (memory-free / combinatorial). Its behavior depends solely on the current values at the data inputs at the time of the `REQ` event. Previous calculations have no influence on the current result.

Application scenarios

- **Safety-critical connections:** Monitoring of multiple safety switches (e.g., emergency stop, safety gates, light barriers) where all conditions must be met to signal a safe state (`OUT=TRUE`).
- **Multiple Conditions in Process Sequences:** Checks whether all prerequisites for starting a machine cycle are met (e.g., "Workpiece inserted," "Pressure reached," "Temperature reached," "Protective cover closed").
- **Status Message Combination:** Combines multiple error or status bits into a single, summarized signal.

⚖️ Comparison with Similar Blocks

- **`AND_BOOL_2`, `AND_BOOL_16`, etc.:** These blocks offer the same logical function, but for a different number of inputs (2, 16, etc.). The choice of block depends on the required number of operands. See: AND_8
- **`OR_n_BOOL`:** Performs a logical OR operation. The result is `TRUE` if at least one input is `TRUE`.
- **`XOR_n_BOOL`:** Performs an exclusive OR (XOR) operation.
- **Continuous Function Blocks (IEC 61131-3):** In IEC 61131-3, AND operations are typically implemented as continuous contacts or operations in languages such as FBD or LD. The `AND_BOOL_8` function block, however, is event-driven and explicitly requires a trigger event.

Conclusion

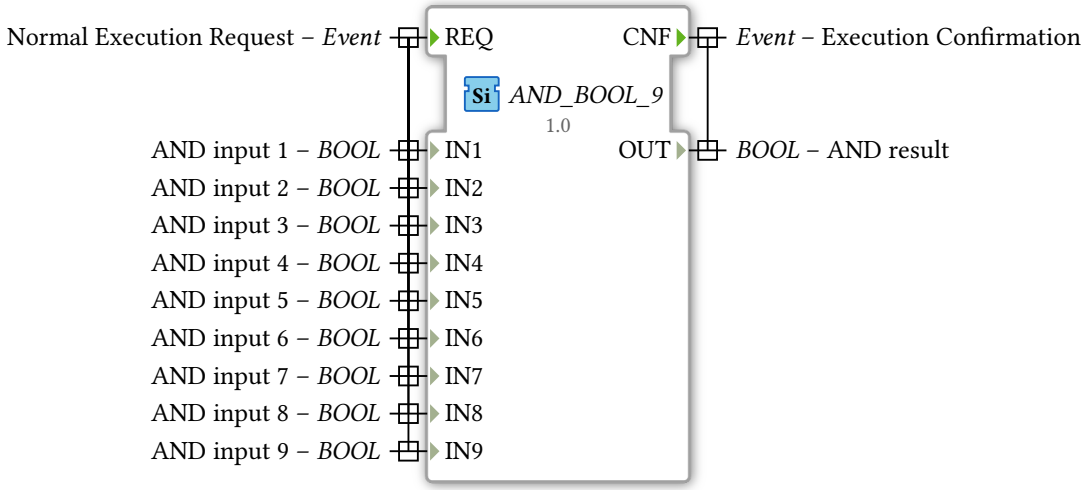
The `AND_BOOL_8` is a fundamental, reliable, and easy-to-use logic block for IEC 61499 systems. Its strength lies in its clear, event-driven semantics and the explicit handling of eight Boolean input signals. It is ideally suited for control logic where multiple conditions must be met simultaneously before an action is executed. For applications with a different number of inputs than eight, the corresponding variant of the `AND_n_BOOL` family should be used.

AND_BOOL_9

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_BOOL_9` is a standard building block for calculating the logical AND operation. It performs a bitwise AND operation across nine separate Boolean inputs. The block complies with the IEC 61131-3 standard and is designed for use in the 4diac IDE.



Interface Structure

The block has an event-driven input/output model.

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation of the AND operation. This event reads all data inputs (`IN1` to `IN9`) and calculates the result.

Event Outputs

- **CNF (Execution Confirmation):** Triggered after the calculation is complete, this indicates that the result is available at data output `OUT`.

Data Inputs

- **IN1 (BOOL):** AND Input 1.
- **IN2 (BOOL):** AND Input 2.
- **IN3 (BOOL):** AND Input 3.
- **IN4 (BOOL):** AND Input 4.
- **IN5 (BOOL):** AND Input 5.
- **IN6 (BOOL):** AND Input 6.
- **IN7 (BOOL):** AND Input 7.
- **IN8 (BOOL):** AND Input 8.
- **IN9 (BOOL):** AND Input 9.

Data Outputs

- **OUT (BOOL):** Result of the AND operation. The output is only `TRUE` if **all** nine inputs (`IN1` to `IN9`) have the value `TRUE`. In all other cases, the output is `FALSE`.

Adapters

This function block does not use any adapters.

Operation

When an event occurs at input `REQ`, the block reads the current values of all nine Boolean inputs (`IN1` to `IN9`). Then, a logical AND operation is performed on all nine values. The result of this operation is immediately output at data output `OUT`. Simultaneously, an acknowledgment event is generated at output `CNF` to signal the successful completion of the operation.

The logical function can be described as follows:

OUT = IN1 AND IN2 AND IN3 AND IN4 AND IN5 AND IN6 AND IN7 AND IN8 AND IN9`

Technical Features

- **Generic Block:** The block is implemented as an instance of a generic AND block (`GEN_AND`) that has been specialized for a fixed number of inputs (here, 9).
- **Event-driven:** The operation is only executed upon an incoming `REQ` event, enabling resource-efficient and deterministic processing.
- **Fixed Number of Inputs:** Unlike variable functions, this block operates with exactly nine inputs.

State Overview

The block does not have an internal state in the sense of a memory. Its behavior is combinatorial and depends solely on the current values of the inputs at the time of the `REQ` event. The output `OUT` retains its value until the next execution cycle.

Application Scenarios

- **Safety Chain:** Linking multiple safety switches (e.g., emergency stop, safety guard, two-hand control) that must all be closed to start a machine.
- **Multiple Conditions:** Checking whether multiple conditions are met simultaneously in a process step.
- **Logic Chaining:** As part of a larger logic circuit where the results from nine sources must be evaluated together.

⚖️ Comparison with Similar Function Blocks

- **`AND` (2 inputs):** The standard AND block with only two inputs. `AND_BOOL_9` is a special, extended version for exactly nine inputs. See: AND_9
- **`GEN_AND`:** The underlying generic function block that can be instantiated with a configurable number of inputs. `AND_BOOL_9` is a predefined instance of it.
- **`OR_*` / `XOR_*` function blocks:** Perform other basic logic operations (OR, exclusive OR) with a fixed number of inputs.

Conclusion

The `AND_BOOL_9` is a robust and easy-to-use standard function block for applications requiring a logical AND operation across exactly nine Boolean signals. Its event-driven nature and clear interface make it easily integrated into 4diac-based control applications. For a different number of inputs, the corresponding specialized blocks (`AND_BOOL_2`, `AND_BOOL_3`, etc.) or the generic `GEN_AND` should be used.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

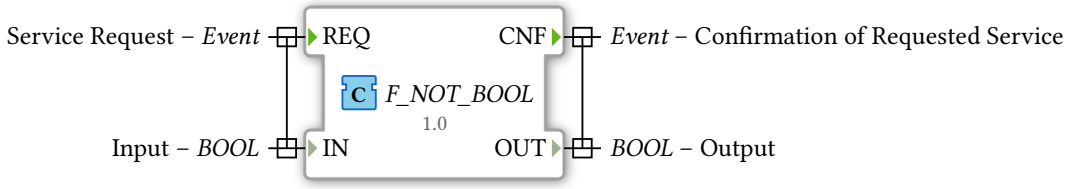
]

F_NOT_BOOL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_NOT_BOOL` implements the logical NOT function (negation) for the data type `BOOL`. It inverts the logical state of a binary input signal. This block belongs to the standard bit operators according to IEC 61131-3 and is designed for use in control applications where logical operations are required.



Interface Structure

Event Inputs

- **REQ (Service Request)**: This event triggers the execution of the logical negation. Upon its arrival, the current value at data input `IN` is processed, and the result is output to `OUT`.

Event Outputs

- **CNF (Confirmation of Requested Service)**: This event signals the completion of the operation. It is generated simultaneously with the output of the calculated value at data output `OUT`.

Data Inputs

- **IN (Input)**: Data type `BOOL`. This is the binary input value to be negated. Valid values are `TRUE` (1) or `FALSE` (0).

Data Outputs

- **OUT (Output)**: Data type `BOOL`. This is the result of the logical negation. The output value is the logical opposite of the input value `IN`.

Adapters

This function block does not use any adapter interfaces.

Operation

The operation is deterministic and simple: Upon each occurrence of a `REQ` event, the function block reads the current value at the data input `IN`. It then calculates the logical negation and writes the result to the data output `OUT`. Simultaneously, a `CNF` event is generated to signal the completion of the operation.

The logical operation can be described by the following table:

Input (IN)	Output (OUT)
FALSE (0)	TRUE (1)
TRUE (1)	FALSE (0)

Technical Features

- **Standard Compliance**: The function block is classified as a standard function according to IEC 61131-3.
- **I/O Delay**: The processing of `REQ` up to the output of `CNF` and `OUT` takes place within one cycle of the executing resource. There are no inherent delays.
- **Initialization**: The output variables are set to their default values (`FALSE` for `BOOL`) during the initialization of the function block.

State Overview

The function block does not have an internal state in the sense of a state machine. Its behavior is purely combinatorial and depends solely on the current input values when triggered by a `REQ` event. No previous values are stored.

Application Scenarios

- **Sensor Signal Inversion**: Conversion of an active low sensor signal (e.g., “Button pressed” = FALSE) into an active high signal for further logic.
- **Safety Logic**: Generation of a block signal from an enable signal or vice versa.
- **Control Logic**: As a fundamental building block in more complex logical expressions and conditions within sequential control systems.

- **Status Display**: Inversion of an internal status for controlling a lamp (e.g., “Machine Ready” = TRUE, but “Fault Lamp Lights Up” if FALSE).

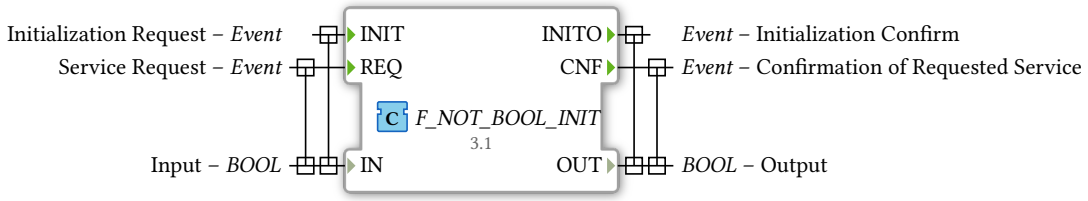
⚖️ Comparison with Similar Function Blocks

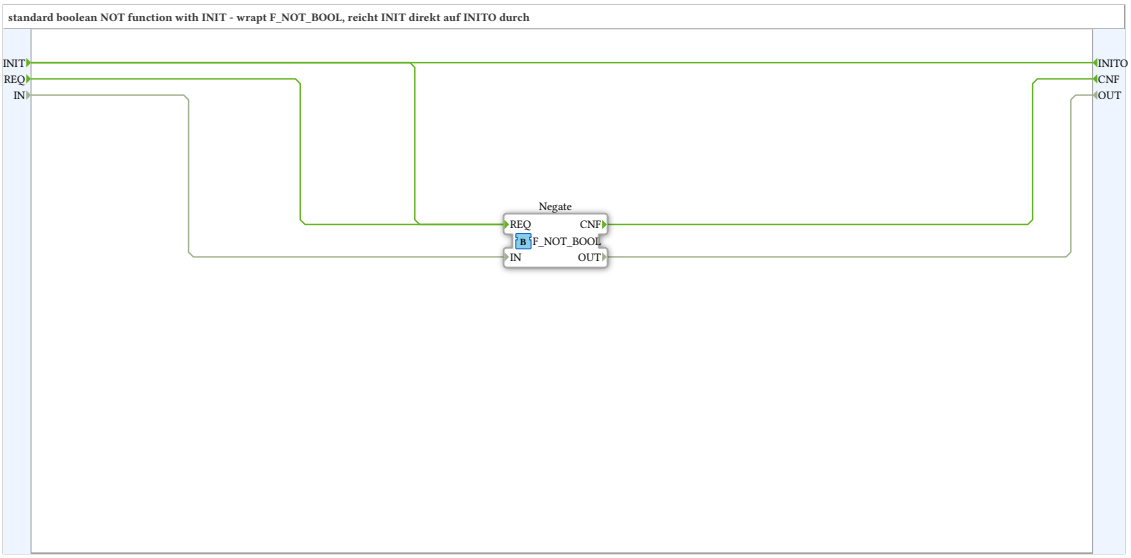
Within the `iec61131::bitwiseOperators` library, other basic logical operators exist, such as `F_AND`, `F_OR`, and `F_XOR`. Unlike these function blocks, which combine two or more inputs, `F_NOT_BOOL` has only a single data input. Its function is essential for completing logical expressions and performing direct inversions. Implementing it as a standalone function block (rather than as an integrated operator within an expression) improves clarity in graphical programming environments like FBD. See: F_NOT

Conclusion

The `F_NOT_BOOL` function block is an essential, robust, and easy-to-use basic building block for binary logic in IEC 61131-3-based control programs. Its clear interface and deterministic operation make it the first choice for inverting Boolean signals in automation technology.

F_NOT_BOOL_INIT





• – ○ • – ○ • – ○ •

The function block `F_NOT_BOOL_INIT` implements the logical negation (NOT) of a Boolean input signal and extends the basic NOT function with an initialization mechanism. It is designed as a standardized block according to IEC 61131-3 for use in programmable logic controllers (PLCs). During initialization, the function block allows a first negation to be triggered and the resulting value to be provided at the output before normal cyclic operation begins.

Event	Type	With Variables	Description
`INIT`	EInit	`IN`	Initialization request – the function block is initialized once and the input is negated.
`REQ`	Event	`IN`	Normal processing request – the logical negation is applied to the current input value.
Event	Type	With Variables	Description

| ----- | --- | ------ | ------- |
INITO	EInit	OUT	Initialization confirmation – output after successful initialization.
CNF	Event	OUT	Confirmation of normal processing – sent to REQ after each execution.
Variable	Type	Description	
-----	----	-------	
IN	BOOL	Input value to be negated.	
Variable	Type	Description	
-----	----	-------	
OUT	BOOL	Negated value of the input (NOT IN).	

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

None.

Functionality

The function block delegates the actual negation function to the internal function block F_NOT (from the library iec61131::bitwiseOperators). Processing is event-driven:

- **For INIT:** The input value IN is immediately transferred via the data connection to the internal function block F_NOT. Simultaneously, the event REQ is sent to the internal function block, which calculates the negation and passes the result to the output OUT. After this operation is completed, the event INITO is output at the main function block. This sets the output OUT with the negated initial value.
- **For REQ (during operation):** The current value of IN is passed to the internal F_NOT and negated. The result is written to OUT, and then the event CNF is triggered.

Thus, both INIT and REQ result in the same core functionality (negation), but differ in the triggered event: INIT generates INITO, and REQ generates CNF.

- The function block does not contain its own algorithmic logic, but uses an internal sub-function (F_NOT). This promotes reuse and consistency.
- The initialization mechanism allows the output to be assigned the negated input even before the first "normal" call – useful in PLC programs where start values need to be defined.
- The function block does not contain its own algorithmic logic, but uses an internal sub-function (F_NOT). This promotes reuse and consistency.
- The initialization mechanism allows the output to be set to the negated input even before the first "normal" call – useful in PLC programs where start values need to be defined.
- The simultaneous use of INIT and REQ enables a clear separation between startup behavior and cyclic operation.
- By including the input variables in the events (WITH), it is ensured that the data is always up-to-date.

The function block (FB) does not have an explicit state machine, as all logic is implemented in the internal network. The following behavior is implicit:

1. **Initial State:** After the PLC starts, outputs and internal values are undefined.
2. **After `INIT`:** The output `OUT` is set to the negated value of `IN`; the FB is ready for operation.
3. **After `REQ`:** For each `REQ` pulse, the negation is performed again, and `OUT` is updated.

A repeated triggering of `INIT` (e.g., after a reset) leads to re-initialization and sets `OUT` to the negated current input value.

- **Initializing a Control Output:** An output should assume the negated value of a sensor signal during the startup phase before the cyclic program starts.
- **Error Monitoring:** During a startup sequence, it is checked whether an input signal has the negated state of an enable signal.
- **Safety Logic:** Used in safety functions where an initial negation step is mandatory (e.g., NOT operation after system startup).

Function Block	Description	Difference to `F_NOT_BOOL_INIT`
`F_NOT` (Standard)	Pure negation without an initialization event.	The `INIT`/`INITO` pair is missing; the output is undefined after system startup.
`F_NOT_BOOL_INIT`	Like `F_NOT`, but with an explicit initialization event.	Offers defined start behavior and outputs a separate acknowledgment event upon initialization.

The standard `F_NOT` is sufficient if the start behavior is controlled by the higher-level control sequence. The initializable function block is advantageous when the function block itself should have control over its first execution.

F_NOT_BOOL_INIT`is a useful function block that extends the simple NOT operation with an initialization facility. It is particularly suitable for PLC applications where a defined start value for the negated output is required. Thanks to the clear event control and the internal use of the provenF_NOT` function block, the implementation remains robust and well-organized. The function block represents a useful extension of the IEC 61131 bit operators.

Technical Features

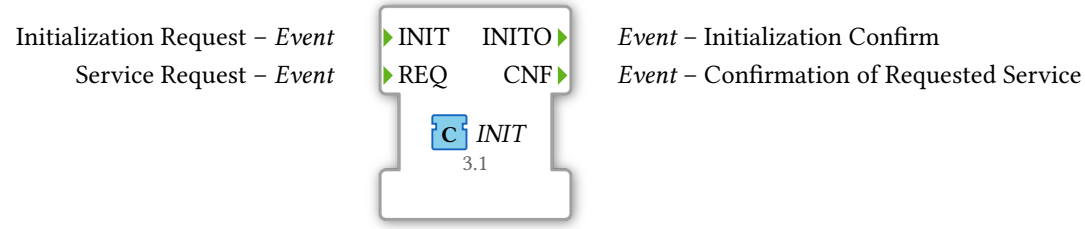
State Overview

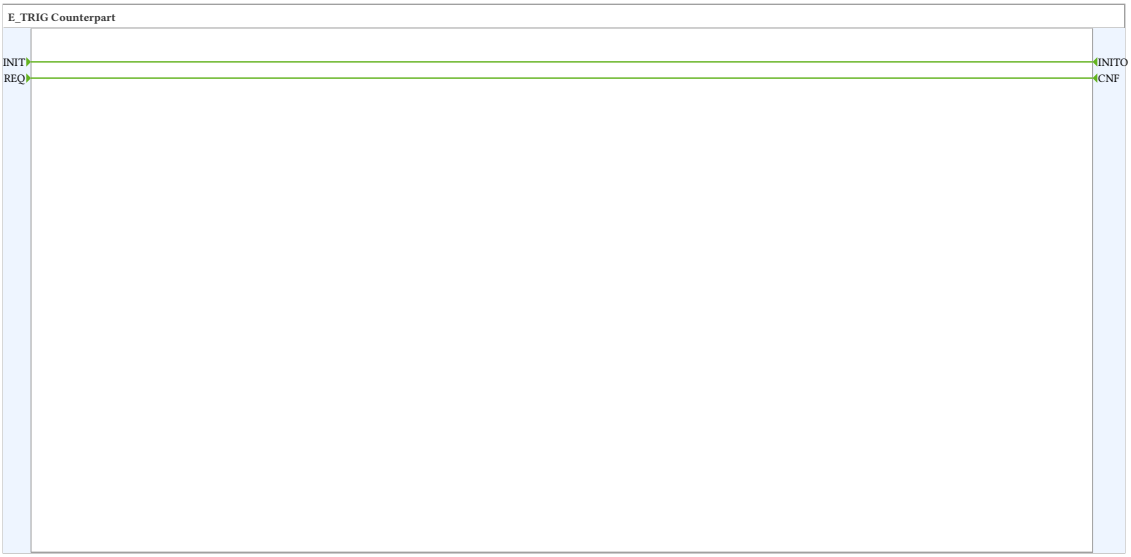
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

INIT





• – ○ • – ○ • – ○ •

Introduction

The **INIT** function block serves as a direct event pass-through. It is designed as the counterpart to a `E_TRIG` function block and is typically used in the initialization phase of a system. The function block has two event inputs and two corresponding event outputs; no data is processed.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request
REQ	Event	Service Request

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Acknowledgement
CNF	Event	Request Acknowledgement

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The function block operates in an event-driven manner and does not perform any logic or state changes of its own. Every incoming event is forwarded to the corresponding output without delay or processing:

- An event at input `INIT` immediately generates an event at output `INITO`.
- An event at input `REQ` immediately generates an event at output `CNF`.

This behavior is achieved through a simple connection of the event paths within the internal function block network. No value changes or state storage occur.

Technical Features

- **Direct Pass-Through**: The function block is extremely lightweight and requires no internal variables or time delays.
- **Event Types**: The input and output events `INIT`/`INITO` are typed as `EInit`, while `REQ`/`CNF` are declared as the default `Event`. This allows for a clear semantic separation between initialization and operational events.
- **License**: The function block is licensed under the Eclipse Public License 2.0 (EPL-2.0).

State Overview

The function block has **no internal states**. It is a pure pass-through block that reacts to incoming events at any time without changing state. Therefore, a state machine is not required.

Application Scenarios

- **Placeholder in Design Phases**: If a `E_TRIG` block is later to be replaced by more complex logic, `INIT` can temporarily serve as a simple pass-through block.
- **Initialization Cascades**: In architectures that perform multiple initialization steps, `INIT` blocks can be used to pass events to subsequent blocks without processing them.
- **Prototyping and Testing**: For bridging unused initialization logic or for debugging event flows.

Comparison with Similar Function Blocks

- **E_TRIG (Input Trigger)**: `E_TRIG` generates an output event only on a rising or falling edge of a connected Boolean input. `INIT`, on the other hand, reacts directly to an event and does not require a data value.
- **E_PERMIT (Enable)**: `E_PERMIT` allows events to pass only if a Boolean enable input is set. `INIT` has no such condition.
- **E_DELAY (Delay)**: `E_DELAY` delays events for a specific period of time. `INIT` operates without delay.

Conclusion

The `INIT` function block is a minimalist yet useful building block for direct event forwarding. Due to its simplicity, it is particularly suitable for design phases, initialization processes, and as a temporary placeholder. It fulfills the basic requirement of passing on initialization and service events unchanged and can be replaced by more functionally richer building blocks if needed.

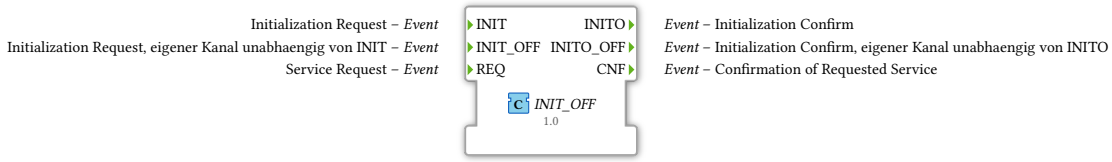
INIT_OFF



• – ○ • – ○ • – ○ •

Introduction

`INIT_OFF` is the counterpart of `E_TRIG`, extended by an independent `INIT_OFF`/`INITO_OFF` pass-through (event type `EInitOff`). The block contains no logic; every input event is forwarded directly to the corresponding output.



Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization request
INIT_OFF	EInitOff	Initialization request, own channel independent of `INIT`
REQ	Event	Service request

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization confirm
INITO_OFF	EInitOff	Initialization confirm, own channel independent of `INITO`
CNF	Event	Confirmation of requested service

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The block consists of event connections only:

- `INIT` ⟶ `INITO`
- `REQ` ⟶ `CNF`
- `INIT_OFF` ⟶ `INITO_OFF`

Technical Details

- Pure pass-through without logic and without data
- The `INIT_OFF` channel is independent of the `INIT` channel

Application Scenarios

- Forwarding initialisation and shutdown events (`EInitOff`) in a chain of blocks

⚖️ Comparison with Similar Building Blocks

- INIT only contains `INIT`/`REQ` without the `INIT_OFF` channel
- INIT_ON_OFF additionally offers an `INIT_ON`/`INITO_ON` channel

Conclusion

`INIT_OFF` forwards `INIT`, `INIT_OFF` and `REQ` unchanged as `INITO`, `INITO_OFF` and `CNF`.

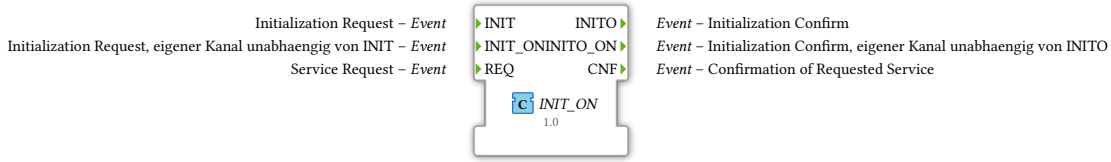
INIT_ON



• – ○ • – ○ • – ○ •

Introduction

`INIT_ON` is the counterpart of `E_TRIG`, extended by an independent `INIT_ON`/`INITO_ON` pass-through (event type `EInitOn`). The block contains no logic; every input event is forwarded directly to the corresponding output.



Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization request
INIT_ON	EInitOn	Initialization request, own channel independent of `INIT`
REQ	Event	Service request

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization confirm
INITO_ON	EInitOn	Initialization confirm, own channel independent of `INITO`
CNF	Event	Confirmation of requested service

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The block consists of event connections only:

- `INIT` ⟶ `INITO`
- `REQ` ⟶ `CNF`
- `INIT_ON` ⟶ `INITO_ON`

Technical Details

- Pure pass-through without logic and without data
- The `INIT_ON` channel is independent of the `INIT` channel

Application Scenarios

- Forwarding initialisation events of a dedicated channel (`EInitOn`) in a chain of blocks

⚖️ Comparison with Similar Building Blocks

- INIT only contains `INIT`/`REQ` without the `INIT_ON` channel
- INIT_OFF offers an `INIT_OFF`/`INITO_OFF` channel (`EInitOff`) instead, INIT_ON_OFF offers both channels

Conclusion

`INIT_ON` forwards `INIT`, `INIT_ON` and `REQ` unchanged as `INITO`, `INITO_ON` and `CNF`.

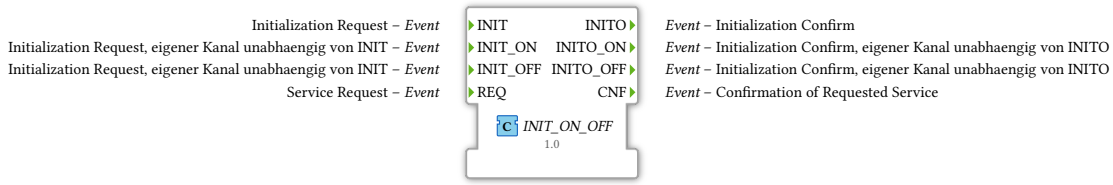
INIT_ON_OFF



• – ○ • – ○ • – ○ •

Introduction

`INIT_ON_OFF` is the counterpart of `E_TRIG`, extended by two independent pass-throughs: `INIT_ON`/`INITO_ON` (event type `EInitOn`) and `INIT_OFF`/`INITO_OFF` (event type `EInitOff`). The block contains no logic; every input event is passed directly to the associated output.



Interface Structure

Event Inputs

- **INIT** (`EInit`): Initialization request
- **INIT_ON** (`EInitOn`): Initialization request, own channel independent of `INIT`
- **INIT_OFF** (`EInitOff`): Initialization request, own channel independent of `INIT`
- **REQ**: Service request

Event Outputs

- **INITO** (`EInit`): Initialization confirmation
- **INITO_ON** (`EInitOn`): Initialization confirmation, own channel independent of `INITO`
- **INITO_OFF** (`EInitOff`): Initialization confirmation, own channel independent of `INITO`
- **CNF**: Confirmation of the requested service

Data Inputs

No data inputs

Data Outputs

No data outputs

Adapters

This function block uses no adapters.

How it works

Pure event connections:

- `INIT` ⟶ `INITO`
- `REQ` ⟶ `CNF`
- `INIT_ON` ⟶ `INITO_ON`
- `INIT_OFF` ⟶ `INITO_OFF`

Technical Details

- Composite function block (`FBNetwork`) without data and without internal state
- The three initialization channels are independent of each other, so that on- and off-related initializations can be wired separately.
- See also `INIT`.

Application Scenarios

- Separate initialization events for the on and off path in a network

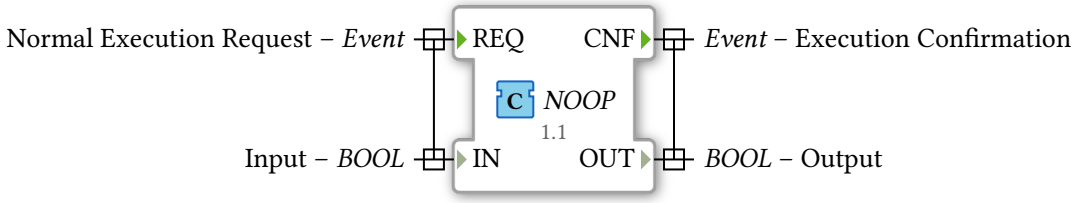
🛠️ Related Exercises

- Uebung_006g_AX

Summary

`INIT_ON_OFF` provides three independent initialization pass-throughs and one `REQ`/`CNF` pass-through.

NOOP





• — ○ • — ○ • — ○ •

Introduction

The NOOP function block (“No Operation”) serves as a simple pass-through block. It copies the BOOL value present at data input `IN` to data output `OUT` as soon as an event occurs at input `REQ`. The output `CNF` is then triggered. The function block does not perform any logical or arithmetic operation; it simply passes the signal on unchanged.

Interface Structure

Event Inputs

Event	Type	Comment
`REQ`	Event	Normal Execution Pulse

Event Outputs

Event	Type	Comment
`CNF`	Event	Execution Acknowledgement

Data Inputs

Name	Type	Comment
`IN`	BOOL	Input Signal

Data Outputs

Name	Type	Comment
`OUT`	BOOL	Output signal (identical to `IN`)

Adapter

None.

Functionality

Internally, the function block consists of an instance of `F_MOVE` (from the library `iec61131::selection::F_MOVE`), configured with the data type `BOOL`.

- Upon an event on `REQ`, the value of `IN` is passed to `F_MOVE.IN`, triggering execution.

- After successful processing, `F_MOVE.OUT` outputs the value, which is then set to `OUT`. Simultaneously, `F_MOVE.CNF` is triggered, which activates the event output `CNF`.

Thus, the behavior corresponds exactly to a passthrough: `OUT := IN`.

Technical Features

- The function block uses a pre-built MOVE block internally, so no custom state logic is required.
- The data paths are explicitly designed for `BOOL`. Extending it to other data types would be possible by modifying the `F_MOVE` attribute, but this is not included in the current design.
- No side effects such as delays or state changes beyond pure signal transmission are implemented. ...

State Overview

Since the function block does not have its own state machine, the process can be viewed as a simple two-step operation:

1. **Wait** – for an event `REQ`.
2. **Forward** – immediate transfer of `IN` to `OUT` and output of `CNF`.

There are no internal memory states or branches.

Application Scenarios

- **Debugging and Testing**: Used as a placeholder to test signal paths in a complex network without affecting functionality.
- **Signal Forwarding**: When an event signal needs to be passed through unchanged, e.g., for synchronizing multiple branches.
- **Placeholder for Future Extensions**: In an early development phase, NOOP can be placed at locations that will later require specific processing.
- **Cascading of Events**: Event coupling allows multiple receivers to be served sequentially.

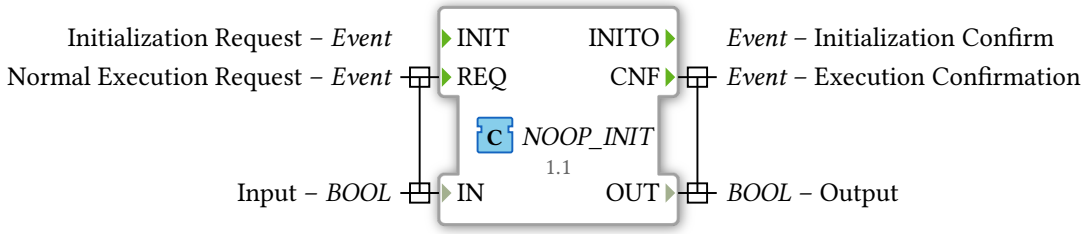
Comparison with Similar Function Blocks

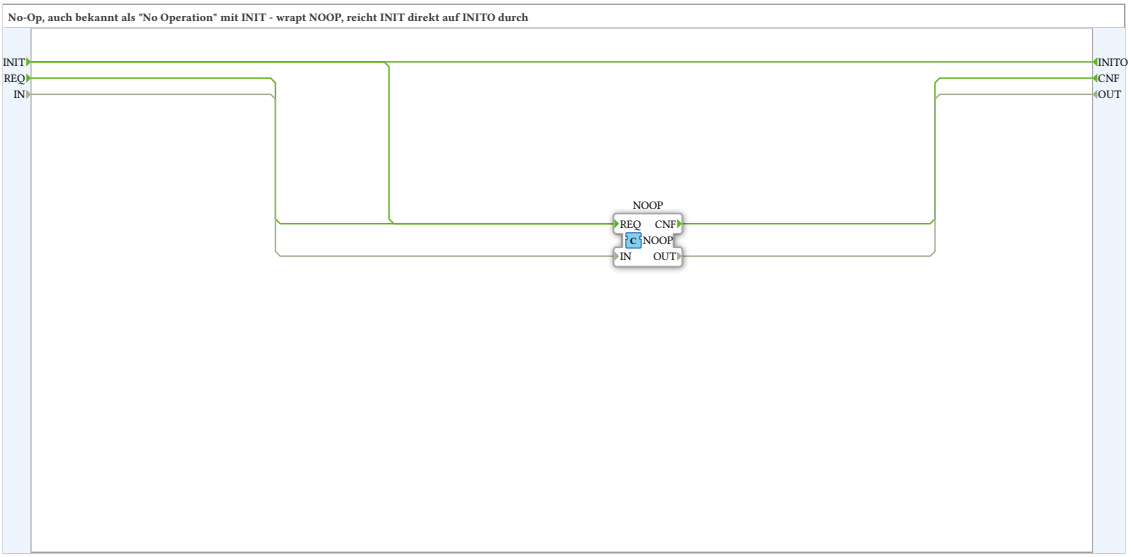
- **MOVE**: A MOVE function block (like the one used internally) also performs a value assignment. NOOP differs only in name and its explicit documentation as "no operation." Both are functionally almost identical, with the difference that MOVE is often typed for arbitrary data types, while NOOP is statically set to BOOL.
- **BISTABLE (RS / SR)**: These function blocks change their state based on inputs. NOOP does not store a state but only outputs the current input value.
- **CONSTANT**: A constant function block always returns a fixed value. NOOP, on the other hand, reacts to changing inputs and events.

Conclusion

The NOOP function block is a simple yet useful function block that requires no internal logic. It copies the incoming BOOL value to the output upon each event and confirms execution. Due to its pass-through characteristics, it is ideally suited for testing purposes, as a placeholder or for event-driven signal transmission in IEC 61499 networks.

NOOP_INIT





• – ○ • – ○ • – ○ •

Introduction

The function block **NOOP_INIT** (No Operation with INIT) serves as a simple pass-through or placeholder block. It enables the initialization and normal data transmission of a Boolean signal from an input to an output. Its special feature lies in the additional handling of the INIT event, which triggers both an acknowledgment (INITO) and data transmission.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request
REQ	Event	Normal Execution Request (connected to IN)

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Acknowledgement
CNF	Event	Execution Acknowledgement (connected to OUT)

Data Inputs

Name	Type	Comment
IN	BOOL	Input Signal

Data Outputs

Name	Type	Comment
OUT	BOOL	Output Signal (corresponds to IN after one iteration)

Adapters

No adapters available.

Functionality

The function block processes incoming events as follows:

1. **INIT Event**:

- An **INITO** event is immediately triggered.
- Simultaneously, the `F_MOVE` function block is triggered internally, which copies the value from `IN` to `OUT`. Upon completion of this copy operation, the **CNF** event is triggered.

1. **REQ Event**:

- The same `F_MOVE` function block is activated, which transfers `IN` to `OUT`. Upon completion, **CNF** is triggered.

As a result, data is transferred during both INIT and REQ, and an acknowledgment event is generated in each case. The data output `OUT` always corresponds to the last received value of the input `IN`.

Technical Features

- The function block internally uses an instance of the standard FB `iec61131::selection::F_MOVE` for data copying.
- The INIT process is twofold: It immediately acknowledges with `INITO` and simultaneously executes the data transfer. This can be useful for initialization sequences where the system should start working after the first input of a start value.
- There are no internal states or timings; the response is event-driven.

State Overview

The FB does not have an explicit state machine; its behavior is purely event-driven:

- In the idle state (no event), the inputs wait.
- Data copying is started on **INIT** or **REQ** Upon completion, `CNF` is sent. Upon INIT, `INITO` is also immediately emitted.

Application Scenarios

- **Placeholder** in the development phase: Instead of a complex function block, NOOP_INIT is inserted to test data flows.
- **Debugging**: The explicit INIT event can be used to set an initial value and simultaneously receive confirmation.
- **Signal coupling**: When a signal needs to be passed on unchanged, but the INIT interface of a subsequent function block needs to be accessed.
- **Testing initialization sequences**: Simulation of a start routine that sets a value and acknowledges it.

Comparison with similar function blocks

- **NOOP (without INIT)**: Simple pass-through function block without an initialization event. NOOP_INIT extends this with the INIT/INITO pairing.

NOOP (without INIT): Simple pass-through function block without an initialization event. - **MOVE (direct)**: A pure data copier without event handling; NOOP_INIT offers an event-driven version with acknowledgments.

- **E_CYCLE or E_PERMIT**: These blocks control events but do not transfer data. NOOP_INIT combines data transfer with event forwarding.

Conclusion

The **NOOP_INIT** function block is a simple yet flexible utility block for event-driven data transfer. Its combination of INIT and REQ events makes it particularly suitable for initialization processes and test environments. Its internal structure makes it easy to understand and integrates well into existing 4diac networks.

–

🌐 Related topic subpages on ms-muc-docs.de

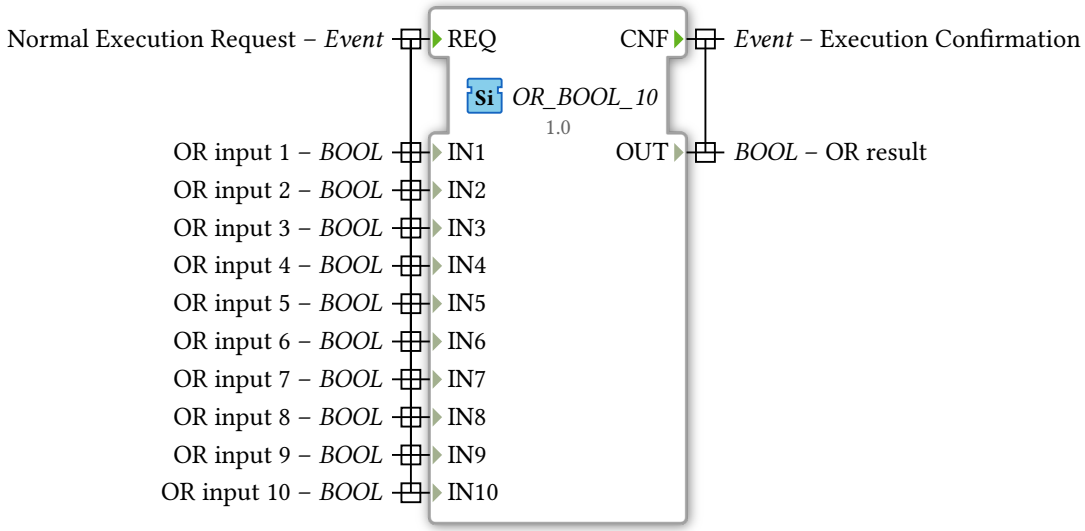
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

OR_BOOL_10

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_10` is a generic block for calculating the logical OR operation. It performs the OR operation on up to ten separate Boolean input values and outputs the result on a single output. This block is classified according to the IEC 61131-3 standard and serves as a standard function for bitwise operations in control applications.



Interface Structure

The function block follows the standard I/O and event model of the 4diac IDE.

Event Inputs

- **REQ (Normal Execution Request):** This event triggers the execution of the function block. Upon its arrival, all ten data inputs (`IN1` to `IN10`) are read and the OR operation is calculated.

Event Outputs

- **CNF (Execution Confirmation):** This event signals the completion of the calculation. It is output along with the calculated result at data output `OUT`.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.

Data Outputs

- **OUT (OR result):** Boolean result of the OR operation of all ten inputs. The output is `TRUE` (1) if at least one of the inputs is `TRUE`. It is only `FALSE` (0) if all ten inputs are `FALSE`.

Adapter

This function block has no adapter interfaces.

Operation

The operation is deterministic and event-driven:

1. The arrival of the event REQ starts the execution.
2. The current values of all ten Boolean inputs (IN1 to IN10) are read.
3. The logical OR operation OUT = IN1 OR IN2 OR IN3 OR ... OR IN10 is calculated.
4. The result is made available at data output OUT.
5. The event CNF is triggered to signal the successful completion of the operation to subsequent blocks.

Technical Features

- **Generic Block:** The block is implemented as a generic function block (GEN_OR), which allows for flexible reuse in different contexts.
- **Fixed Number of Inputs:** Unlike blocks with a variable number of inputs, OR_BOOL_10 has exactly ten fixed inputs. Unused inputs should be set to FALSE.
- **Event-driven:** The calculation only occurs when a REQ event occurs, enabling energy- and computationally efficient processing in the control system.

State Overview

The function block has no internal state (memoryless). The output signal OUT is a pure function of the current input values at the time of the REQ request. There is no delay, hysteresis, or storage of previous states.

Application Scenarios

- **Monitoring Logic:** Combining multiple error or warning signals (e.g., from different sensors) into a single alarm signal.
- **Enable Logic:** Generating an enable signal for a process step when at least one of several prerequisites is met.
- **Button Group Linking:** In an operator station where a process can be initiated by pressing at least one of several "Start" buttons.
- **Redundant Sensor Evaluation:** Evaluates multiple redundant sensors, where the signal from any one sensor is considered valid.

⚖️ Comparison with Similar Blocks

- **OR_BOOL_2 / OR_BOOL_4:** These blocks offer the same OR functionality, but for a smaller number of inputs (2 and 4, respectively). OR_BOOL_10 is intended for applications with a higher number of signals to be linked. See: OR_10
- **AND_BOOL_10:** Performs the logical AND operation. The result is only TRUE if *all* inputs are TRUE, whereas for OR_BOOL_10, it is sufficient if *at least one* input is TRUE.
- **XOR_BOOL_10:** Performs the exclusive OR operation. The result is TRUE if an odd number of inputs are TRUE, which is fundamentally different from inclusive OR logic.
- **Blocks with variable input count:** Some libraries offer OR blocks where the number of inputs is configurable. OR_BOOL_10, however, offers a fixed, explicit interface.

OR_BOOL_10:

Conclusion

The OR_BOOL_10 is a robust and easy-to-use basic building block for logical signal processing in IEC 61131-3-based control applications. Its strength lies in its clear interface with ten inputs and the reliable, event-driven calculation of the inclusive OR function. It represents an optimal solution for applications that require combining multiple Boolean sources into a single signal.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

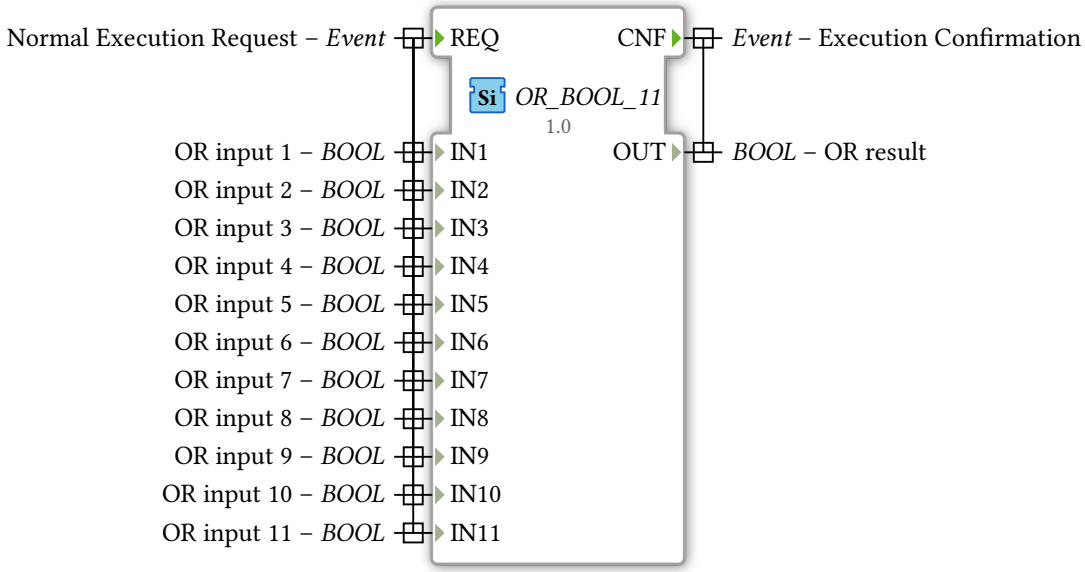
OR_BOOL_11

• – ◦ • – ◦ • – ◦ •

Introduction

The function block OR_BOOL_11 is a generic block for calculating the logical OR operation (classification standard boolean function, IEC 61131-3). It combines 11 separate BOOL inputs (IN1 to IN11) and pro-

vides the result at the output OUT. It extends the standard OR_BOOL family (previously up to OR_BOOL_16) with the variant that has 11 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 11 data inputs (IN1 to IN11).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output OUT.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.
- **IN11 (OR input 11):** Boolean input 11.

Data Outputs

- **OUT (OR result):** The result of the OR operation of all 11 inputs. The output is TRUE if at least one input is TRUE and FALSE only if all inputs IN1 to IN11 are FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event REQ, all 11 inputs are read and the operation `OUT = IN1 OR IN2 OR ... OR IN11` is calculated. The result is then made available at OUT and CNF is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_OR_BOOL'`; this instance is fixed to 11 inputs.

- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Collective fault:** Combine several fault or warning signals into one common signal.
- **Multiple triggers:** An action is to start as soon as at least one of many conditions or buttons is active.

⚖️ Comparison with Similar Function Blocks

- `AND_BOOL_11`: Performs the AND operation: the result is TRUE only if all inputs are TRUE, while `OR_BOOL_11` needs just one TRUE input.
- `OR_BOOL_16`: Sibling block of the family with 16 inputs.

Conclusion

`OR_BOOL_11` is a simple, event-driven block for the OR combination of 11 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

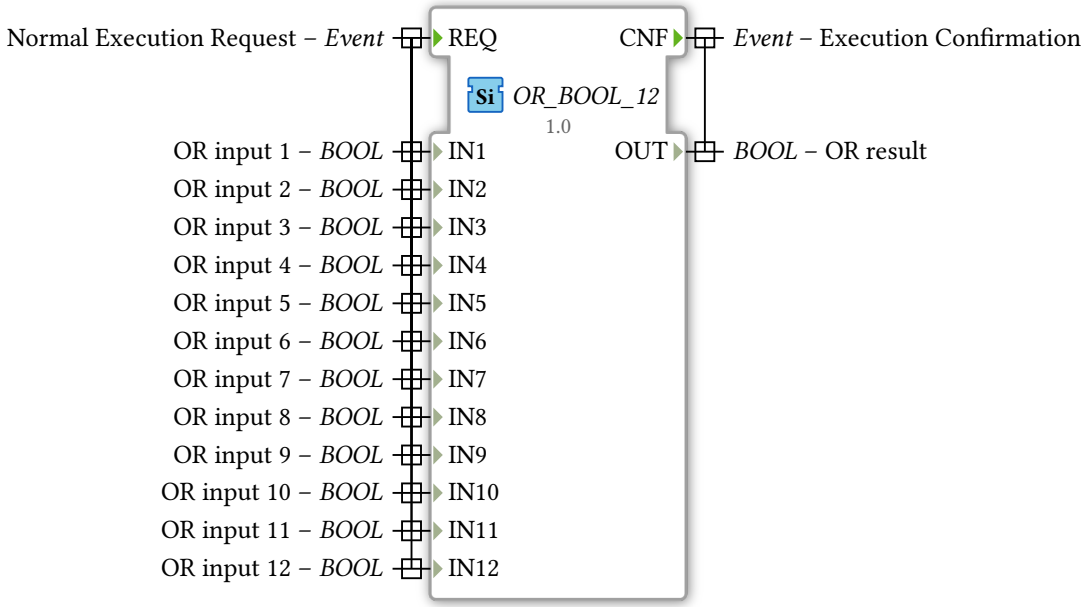
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

OR_BOOL_12

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_12` is a generic block for calculating the logical OR operation (classification `standard boolean function`, IEC 61131-3). It combines 12 separate BOOL inputs (`IN1` to `IN12`) and provides the result at the output `OUT`. It extends the standard OR_BOOL family (previously up to `OR_BOOL_16`) with the variant that has 12 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 12 data inputs (`IN1` to `IN12`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.
- **IN11 (OR input 11):** Boolean input 11.
- **IN12 (OR input 12):** Boolean input 12.

Data Outputs

- **OUT (OR result):** The result of the OR operation of all 12 inputs. The output is TRUE if at least one input is TRUE and FALSE only if all inputs `IN1` to `IN12` are FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 12 inputs are read and the operation `OUT = IN1 OR IN2 OR ... OR IN12` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_OR_BOOL'`; this instance is fixed to 12 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Collective fault:** Combine several fault or warning signals into one common signal.
- **Multiple triggers:** An action is to start as soon as at least one of many conditions or buttons is active.

⚖️ Comparison with Similar Function Blocks

- `AND_BOOL_12`: Performs the AND operation: the result is TRUE only if all inputs are TRUE, while `OR_BOOL_12` needs just one TRUE input.
- `OR_BOOL_16`: Sibling block of the family with 16 inputs.

Conclusion

`OR_BOOL_12` is a simple, event-driven block for the OR combination of 12 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

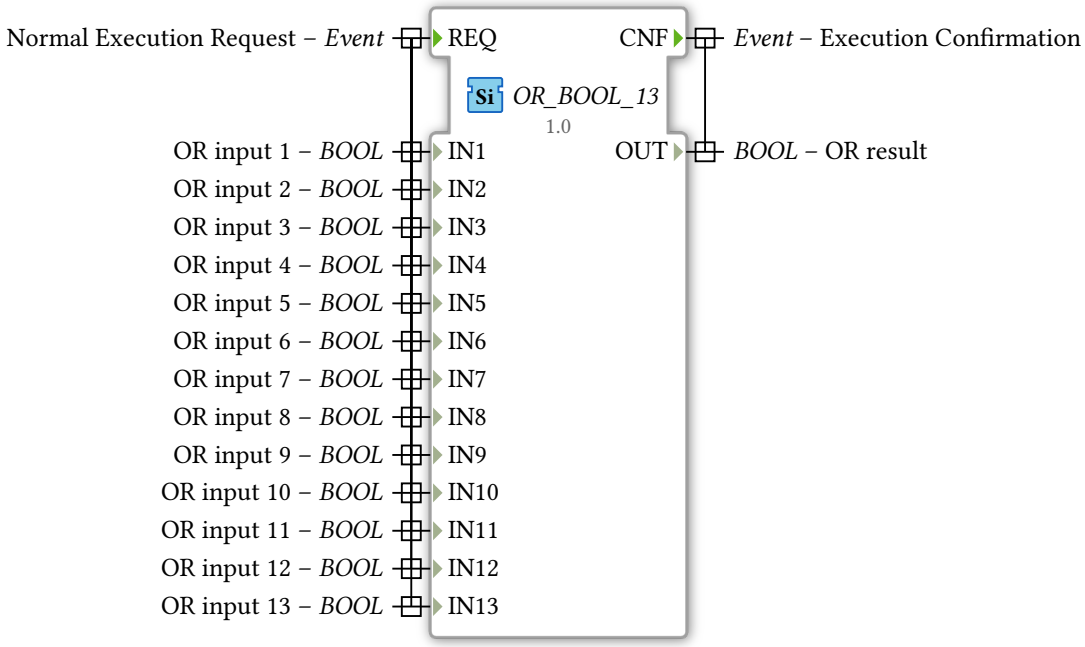
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

OR_BOOL_13

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_13` is a generic block for calculating the logical OR operation (classification `standard boolean function`, IEC 61131-3). It combines 13 separate BOOL inputs (`IN1` to `IN13`) and provides the result at the output `OUT`. It extends the standard OR_BOOL family (previously up to `OR_BOOL_16`) with the variant that has 13 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 13 data inputs (`IN1` to `IN13`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.
- **IN11 (OR input 11):** Boolean input 11.
- **IN12 (OR input 12):** Boolean input 12.
- **IN13 (OR input 13):** Boolean input 13.

Data Outputs

- **OUT (OR result):** The result of the OR operation of all 13 inputs. The output is TRUE if at least one input is TRUE and FALSE only if all inputs `IN1` to `IN13` are FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 13 inputs are read and the operation `OUT = IN1 OR IN2 OR ... OR IN13` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_OR_BOOL'`; this instance is fixed to 13 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Collective fault:** Combine several fault or warning signals into one common signal.
- **Multiple triggers:** An action is to start as soon as at least one of many conditions or buttons is active.

⚖️ Comparison with Similar Function Blocks

- `AND_BOOL_13`: Performs the AND operation: the result is TRUE only if all inputs are TRUE, while `OR_BOOL_13` needs just one TRUE input.
- `OR_BOOL_16`: Sibling block of the family with 16 inputs.

Conclusion

`OR_BOOL_13` is a simple, event-driven block for the OR combination of 13 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

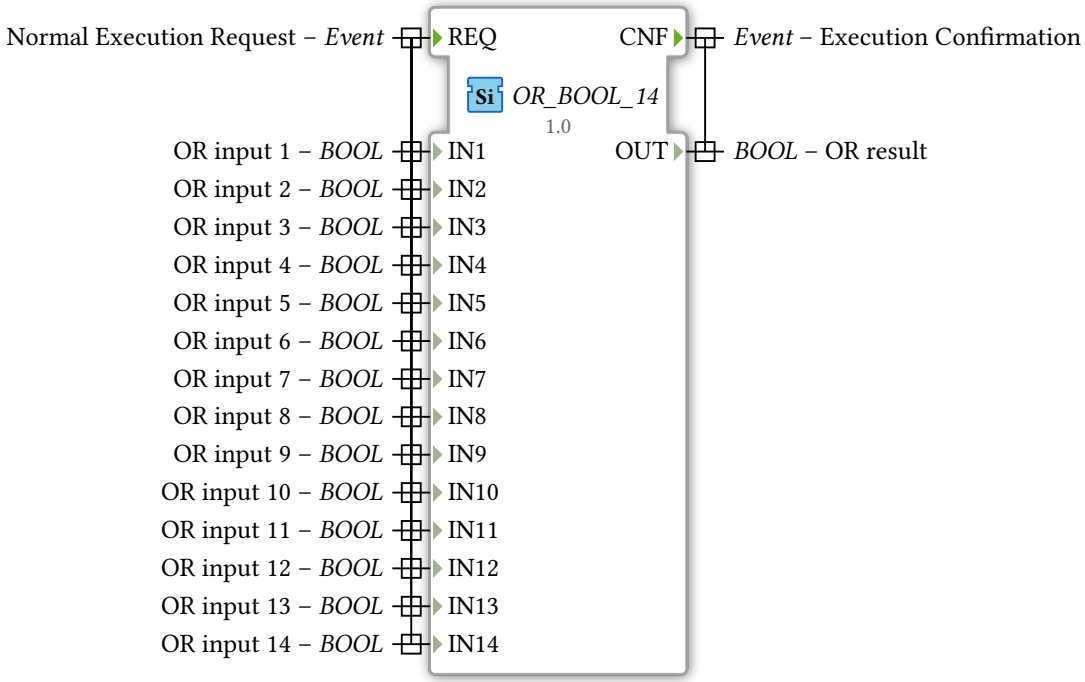
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

OR_BOOL_14

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_14` is a generic block for calculating the logical OR operation (classification `standard boolean function`, IEC 61131-3). It combines 14 separate BOOL inputs (`IN1` to `IN14`) and provides the result at the output `OUT`. It extends the standard OR_BOOL family (previously up to `OR_BOOL_16`) with the variant that has 14 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 14 data inputs (`IN1` to `IN14`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.
- **IN11 (OR input 11):** Boolean input 11.
- **IN12 (OR input 12):** Boolean input 12.
- **IN13 (OR input 13):** Boolean input 13.
- **IN14 (OR input 14):** Boolean input 14.

Data Outputs

- **OUT (OR result):** The result of the OR operation of all 14 inputs. The output is TRUE if at least one input is TRUE and FALSE only if all inputs `IN1` to `IN14` are FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 14 inputs are read and the operation `OUT = IN1 OR IN2 OR ... OR IN14` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_OR_BOOL'`; this instance is fixed to 14 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Collective fault:** Combine several fault or warning signals into one common signal.
- **Multiple triggers:** An action is to start as soon as at least one of many conditions or buttons is active.

⚖️ Comparison with Similar Function Blocks

- `AND_BOOL_14`: Performs the AND operation: the result is TRUE only if all inputs are TRUE, while `OR_BOOL_14` needs just one TRUE input.
- `OR_BOOL_16`: Sibling block of the family with 16 inputs.

Conclusion

`OR_BOOL_14` is a simple, event-driven block for the OR combination of 14 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

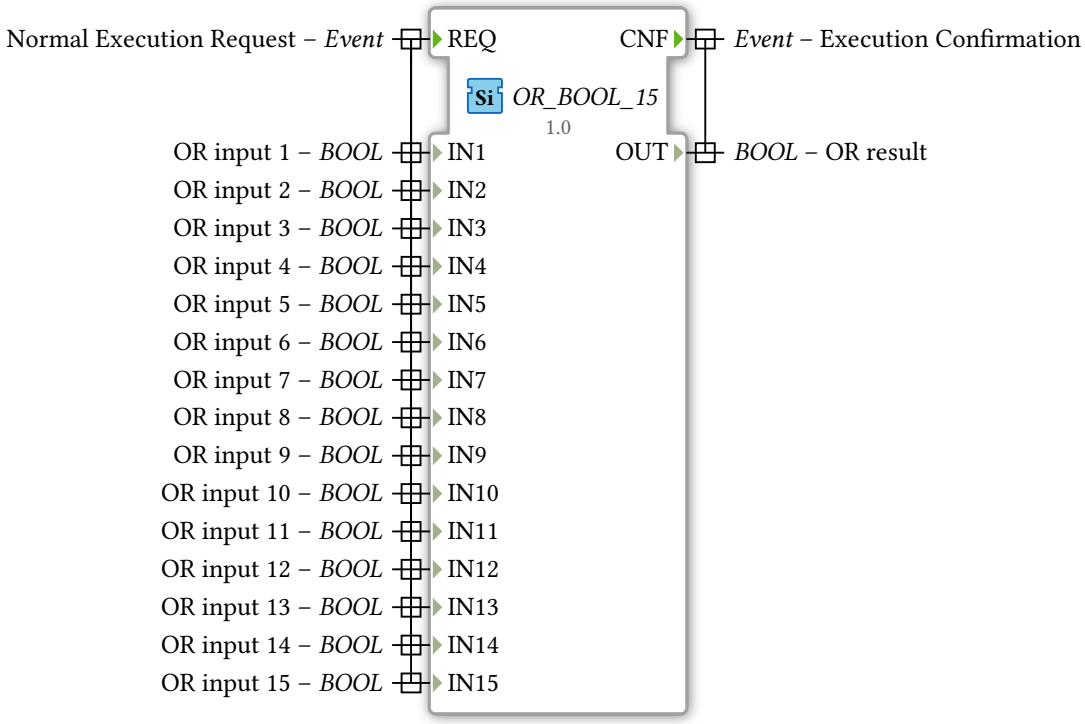
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

OR_BOOL_15

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_15` is a generic block for calculating the logical OR operation (classification `standard boolean function`, IEC 61131-3). It combines 15 separate BOOL inputs (`IN1` to `IN15`) and provides the result at the output `OUT`. It extends the standard OR_BOOL family (previously up to `OR_BOOL_16`) with the variant that has 15 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 15 data inputs (`IN1` to `IN15`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.
- **IN11 (OR input 11):** Boolean input 11.
- **IN12 (OR input 12):** Boolean input 12.
- **IN13 (OR input 13):** Boolean input 13.
- **IN14 (OR input 14):** Boolean input 14.
- **IN15 (OR input 15):** Boolean input 15.

Data Outputs

- **OUT (OR result):** The result of the OR operation of all 15 inputs. The output is TRUE if at least one input is TRUE and FALSE only if all inputs `IN1` to `IN15` are FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 15 inputs are read and the operation `OUT = IN1 OR IN2 OR ... OR IN15` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_OR_BOOL'`; this instance is fixed to 15 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Collective fault:** Combine several fault or warning signals into one common signal.
- **Multiple triggers:** An action is to start as soon as at least one of many conditions or buttons is active.

⚖️ Comparison with Similar Function Blocks

- `AND_BOOL_15`: Performs the AND operation: the result is TRUE only if all inputs are TRUE, while `OR_BOOL_15` needs just one TRUE input.
- `OR_BOOL_16`: Sibling block of the family with 16 inputs.

Conclusion

`OR_BOOL_15` is a simple, event-driven block for the OR combination of 15 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

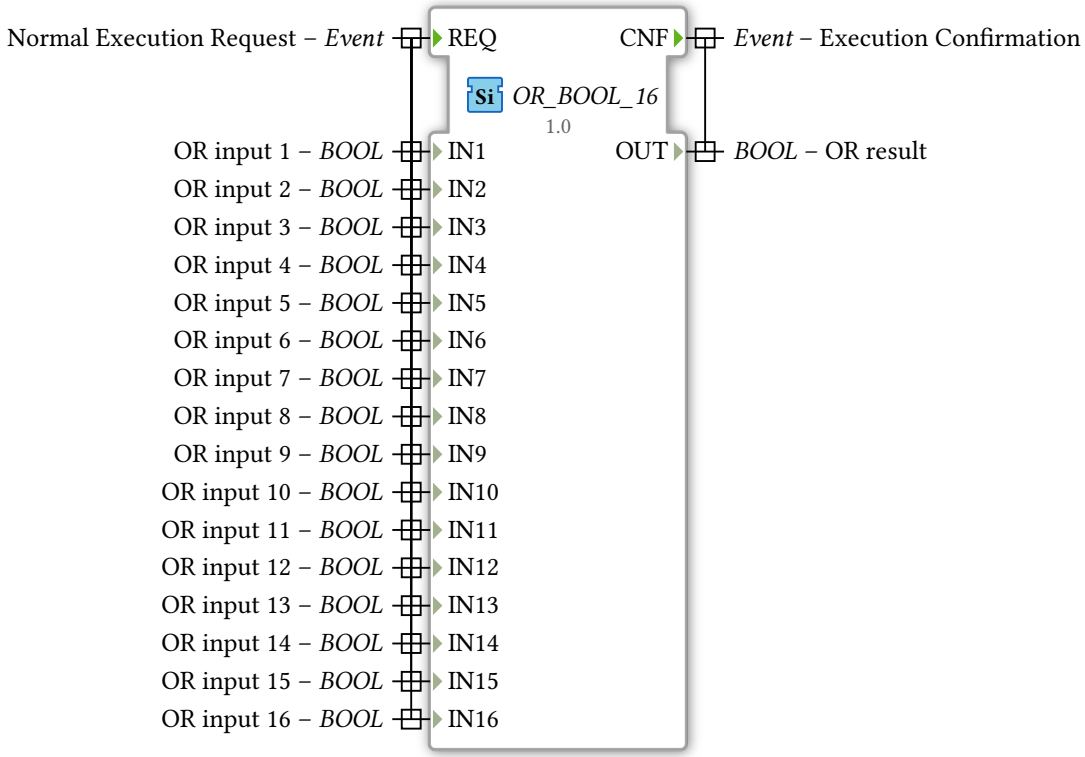
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

OR_BOOL_16

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_16` is a standard function block for calculating the logical OR operation. It performs the OR operation via 16 separate Boolean inputs and provides the result at a single output. This function block is part of the IEC 61131-3 compliant library for bitwise operations and is suitable for applications where a logical combination of multiple signals is required.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** This event triggers the calculation of the OR function. It is linked to all 16 data inputs (`IN1` to `IN16`).

Event Outputs

- **CNF (Execution Confirmation):** This event signals the completion of the calculation. It is output along with the updated data output `OUT`.

Data Inputs

The function block has 16 identical Boolean data inputs:

- **IN1** to **IN16 (BOOL):** The input signals to be combined. Each input represents an operand for the OR operation.

Data Outputs

- **OUT (BOOL):** The result of the logical OR operation on all 16 inputs. The output is `TRUE` (1) if at least one of the inputs is `TRUE`. It is only `FALSE` (0) if all 16 inputs are `FALSE`.

Adapter

This function block does not use any adapter interfaces.

Operation

The operation is deterministic and event-driven:

1. The block is activated when an event occurs at input `REQ`.
2. The block reads the current values of all 16 inputs `IN1` to `IN16`.
3. The logical OR operation is calculated on all read values: `OUT = IN1 OR IN2 OR ... OR IN16`. 4. The result is set at the data output `OUT`.
4. Immediately after the calculation, the confirmation event `CNF` is output along with the new value of `OUT`.

Technical Features

- **Generic Block:** The block is marked as a generic block (`GEN_OR`), indicating that the underlying logic can be reused for other OR blocks with different numbers of inputs.
- **Fixed Number of Inputs:** Unlike blocks with a variable number of inputs, `OR_BOOL_16` has exactly 16 inputs. This provides a clear and fixed interface.
- **Event-driven execution:** The calculation only occurs upon receipt of an incoming `REQ` event, enabling energy-efficient and demand-based processing in the real-time system.

State overview

The function block does not have an internal state in the sense of a memory. It behaves purely combinatorially with respect to the input data. Its "state" is defined solely by the presence or absence of a pending calculation job (`REQ`). After processing and sending `CNF`, it returns to a passive waiting state.

Application scenarios

- **Monitoring logic:** Combining multiple error or warning signals into a single fault signal. (e.g., "Stop the machine if sensor A OR sensor B OR ... OR sensor P reports a fault").
- **Enable Logic:** Checks whether at least one of several conditions for starting a process is met.
- **Button Group Linking:** In an operator station where an action can be triggered by pressing at least one of several (up to 16) buttons.

⚖️ Comparison with Similar Function Blocks

- **`OR_BOOL_2`, `OR_BOOL_8`:** These are identical OR function blocks with a smaller number of inputs (2 and 8, respectively). `OR_BOOL_16` extends this series for applications with a higher number of inputs. See: OR_16
- **`AND_BOOL_16`:** Performs the logical AND operation. The result is only `TRUE` if *all* inputs are `TRUE`, whereas for `OR_BOOL_16`, it is sufficient if *at least one* input is `TRUE`.
- **Blocks with variable input count:** Some libraries offer OR blocks where the number of inputs is configurable. `OR_BOOL_16`, on the other hand, offers a fixed, optimized interface for exactly 16 signals.

Conclusion

The `OR_BOOL_16` is a robust and easy-to-use standard block for the logical OR operation of a large group of Boolean signals. Its fixed structure with 16 inputs makes it predictable and easy to use, especially when the required number of signals is known and constant. Event-driven execution seamlessly integrates it into 4diac's dataflow-oriented architecture. It's the ideal choice when up to 16 digital signals need to be reduced to a common OR result.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

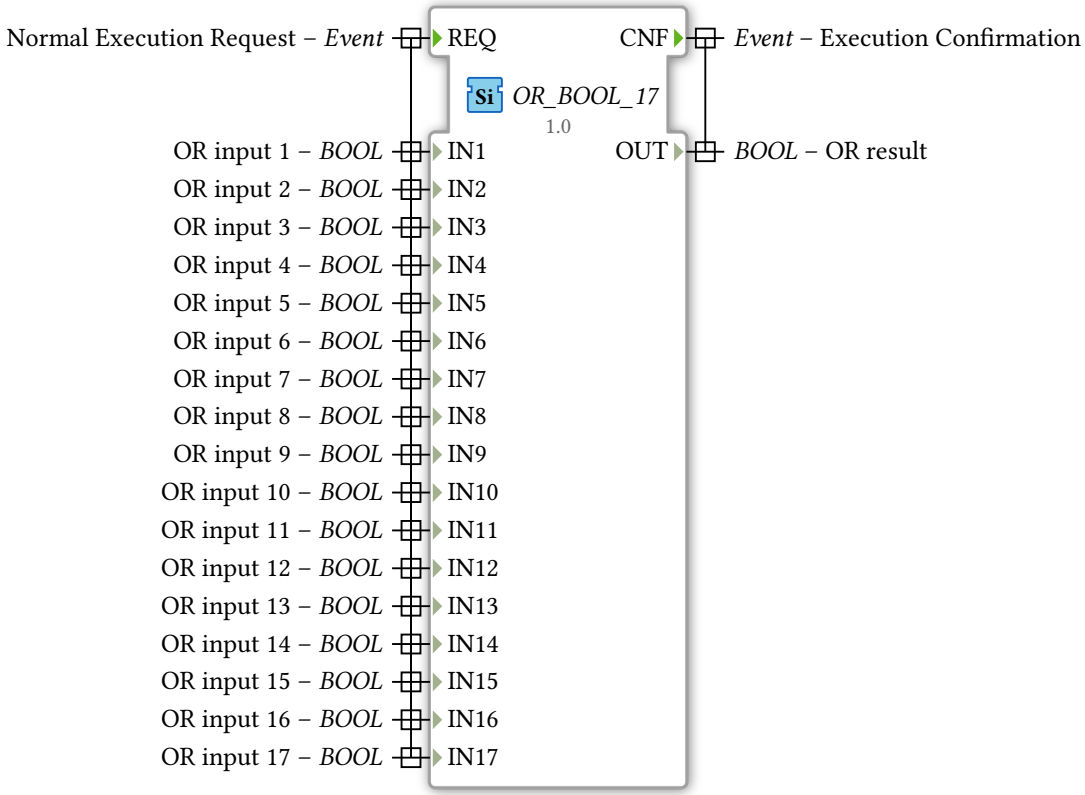
]

OR_BOOL_17

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_17` is a generic block for calculating the logical OR operation (classification `standard boolean function`, IEC 61131-3). It combines 17 separate BOOL inputs (`IN1` to `IN17`) and provides the result at the output `OUT`. It extends the standard OR_BOOL family (previously up to `OR_BOOL_16`) with the variant that has 17 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 17 data inputs (`IN1` to `IN17`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.
- **IN11 (OR input 11):** Boolean input 11.
- **IN12 (OR input 12):** Boolean input 12.
- **IN13 (OR input 13):** Boolean input 13.
- **IN14 (OR input 14):** Boolean input 14.
- **IN15 (OR input 15):** Boolean input 15.
- **IN16 (OR input 16):** Boolean input 16.
- **IN17 (OR input 17):** Boolean input 17.

Data Outputs

- **OUT (OR result):** The result of the OR operation of all 17 inputs. The output is TRUE if at least one input is TRUE and FALSE only if all inputs `IN1` to `IN17` are FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 17 inputs are read and the operation `OUT = IN1 OR IN2 OR ... OR IN17` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_OR_BOOL'`; this instance is fixed to 17 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Collective fault:** Combine several fault or warning signals into one common signal.
- **Multiple triggers:** An action is to start as soon as at least one of many conditions or buttons is active.

⚖️ Comparison with Similar Function Blocks

- `AND_BOOL_17`: Performs the AND operation: the result is TRUE only if all inputs are TRUE, while `OR_BOOL_17` needs just one TRUE input.
- `OR_BOOL_16`: Sibling block of the family with 16 inputs.

Conclusion

`OR_BOOL_17` is a simple, event-driven block for the OR combination of 17 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

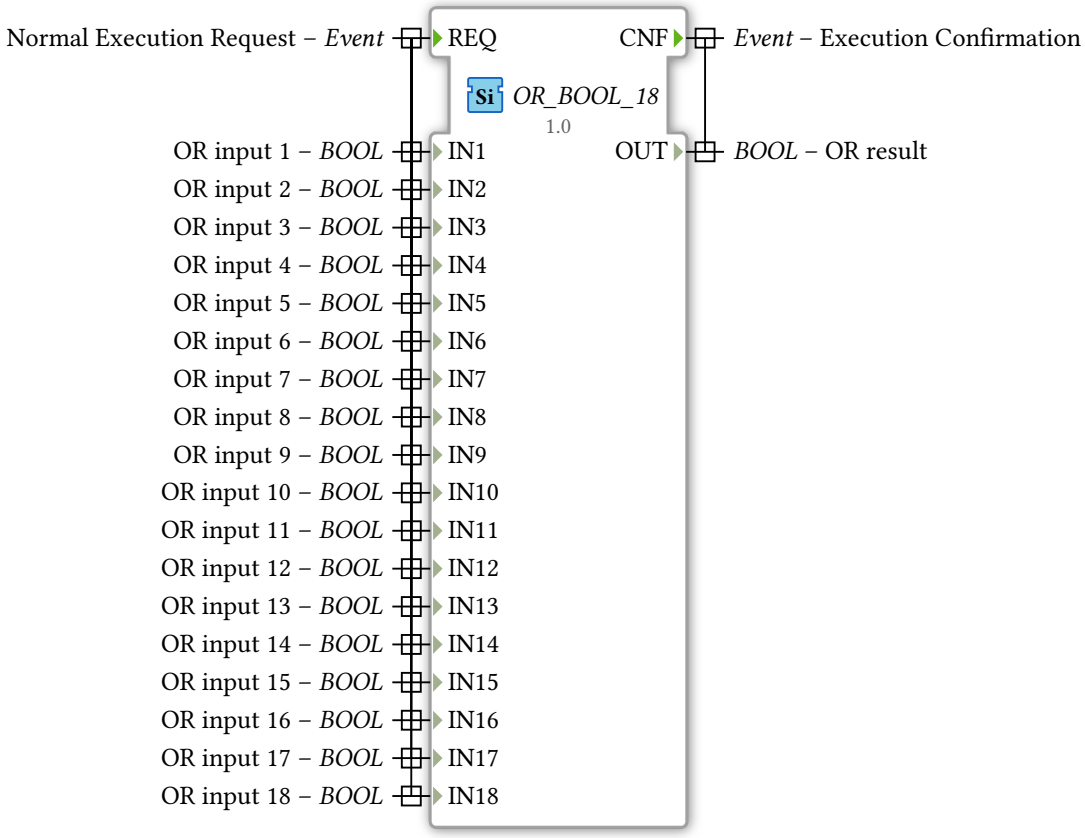
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

OR_BOOL_18

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_18` is a generic block for calculating the logical OR operation (classification `standard boolean function`, IEC 61131-3). It combines 18 separate BOOL inputs (`IN1` to `IN18`) and provides the result at the output `OUT`. It extends the standard OR_BOOL family (previously up to `OR_BOOL_16`) with the variant that has 18 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 18 data inputs (`IN1` to `IN18`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.
- **IN11 (OR input 11):** Boolean input 11.
- **IN12 (OR input 12):** Boolean input 12.
- **IN13 (OR input 13):** Boolean input 13.
- **IN14 (OR input 14):** Boolean input 14.
- **IN15 (OR input 15):** Boolean input 15.
- **IN16 (OR input 16):** Boolean input 16.
- **IN17 (OR input 17):** Boolean input 17.
- **IN18 (OR input 18):** Boolean input 18.

Data Outputs

- **OUT (OR result):** The result of the OR operation of all 18 inputs. The output is TRUE if at least one input is TRUE and FALSE only if all inputs `IN1` to `IN18` are FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 18 inputs are read and the operation `OUT = IN1 OR IN2 OR ... OR IN18` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_OR_BOOL'`; this instance is fixed to 18 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Collective fault:** Combine several fault or warning signals into one common signal.
- **Multiple triggers:** An action is to start as soon as at least one of many conditions or buttons is active.

⚖️ Comparison with Similar Function Blocks

- `AND_BOOL_18`: Performs the AND operation: the result is TRUE only if all inputs are TRUE, while `OR_BOOL_18` needs just one TRUE input.
- `OR_BOOL_16`: Sibling block of the family with 16 inputs.

Conclusion

`OR_BOOL_18` is a simple, event-driven block for the OR combination of 18 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

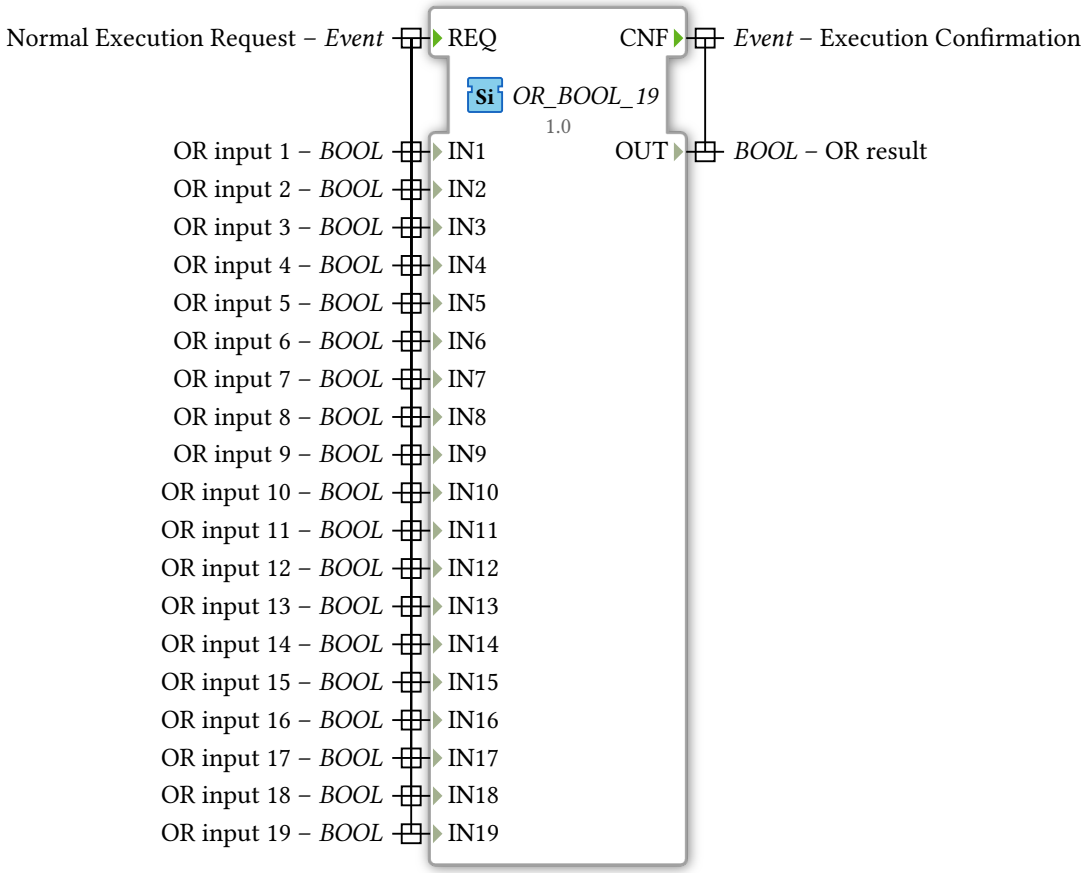
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

OR_BOOL_19

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_19` is a generic block for calculating the logical OR operation (classification `standard boolean function`, IEC 61131-3). It combines 19 separate BOOL inputs (`IN1` to `IN19`) and provides the result at the output `OUT`. It extends the standard OR_BOOL family (previously up to `OR_BOOL_16`) with the variant that has 19 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 19 data inputs (`IN1` to `IN19`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.
- **IN11 (OR input 11):** Boolean input 11.
- **IN12 (OR input 12):** Boolean input 12.
- **IN13 (OR input 13):** Boolean input 13.
- **IN14 (OR input 14):** Boolean input 14.
- **IN15 (OR input 15):** Boolean input 15.
- **IN16 (OR input 16):** Boolean input 16.
- **IN17 (OR input 17):** Boolean input 17.
- **IN18 (OR input 18):** Boolean input 18.

- **IN19 (OR input 19):** Boolean input 19.

Data Outputs

- **OUT (OR result):** The result of the OR operation of all 19 inputs. The output is TRUE if at least one input is TRUE and FALSE only if all inputs `IN1` to `IN19` are FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 19 inputs are read and the operation `OUT = IN1 OR IN2 OR ... OR IN19` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_OR_BOOL'`; this instance is fixed to 19 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Collective fault:** Combine several fault or warning signals into one common signal.
- **Multiple triggers:** An action is to start as soon as at least one of many conditions or buttons is active.

⚖️ Comparison with Similar Function Blocks

- `AND_BOOL_19`: Performs the AND operation: the result is TRUE only if all inputs are TRUE, while `OR_BOOL_19` needs just one TRUE input.
- `OR_BOOL_16`: Sibling block of the family with 16 inputs.

Conclusion

`OR_BOOL_19` is a simple, event-driven block for the OR combination of 19 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

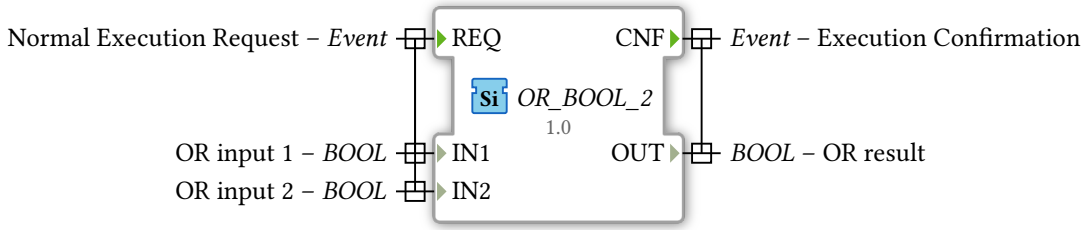
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

OR_BOOL_2

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_2` performs a logical OR operation (disjunction) on two Boolean input values. It is a generic function block classified according to the IEC 61131-3 standard for standard Boolean functions. The block waits for an execution request, calculates the result, and signals its availability.



Interface Structure

Event Inputs

- **REQ** (Normal Execution Request): This event triggers the execution of the function block. Upon its arrival, the values at data inputs `IN1` and `IN2` are read, and the OR operation is performed.

Event Outputs

- **CNF** (Execution Confirmation): This event is generated after the calculation is complete. It indicates that the result has been output to data output `OUT` and is valid.

Data Inputs

- **IN1** (BOOL): First operand for the OR operation.
- **IN2** (BOOL): Second operand for the OR operation.

Data Outputs

- **OUT** (BOOL): Result of the logical OR operation of `IN1` and `IN2`. The output value is `TRUE` if at least one of the two inputs is `TRUE`. Otherwise, it is `FALSE`.

Adapter

This function block has no adapter interfaces.

Operation

The operation is event-driven and follows a simple request-acknowledgment cycle:

1. The arrival of the event `REQ` starts the execution.
2. The current values of the data inputs `IN1` and `IN2` are retrieved.
3. The logical OR operation `OUT := IN1 OR IN2` is calculated.
4. The result is placed at the data output `OUT`.
5. The event `CNF` is generated to indicate the completion of the operation.

The truth table of the operation is:

IN1	IN2	OUT (IN1 OR IN2)
0	0	0
0	1	1
1	0	1
1	1	1

Technical Features

- **Generic Block:** The block is marked as a generic block (`GEN_OR`), which means it can serve as a basis for deriving similar blocks with other data types.
- **Simple Runtime:** The execution time is constant and very short, as it involves a simple logical operation.
- **No Internal State:** The block is stateless (combinatorial). The result depends solely on the current input values at the time of the `REQ` request.

State Overview

The function block has no persistent internal state. Its behavior is purely combinatorial and is fully described by the interface events and data.

Application Scenarios

- **Logical Connections in Control Logic:** Combination of two conditions where an action is to be executed if at least one of the conditions is met (e.g., "Start motor if button is pressed OR remote control signal is received").
- **Error Aggregation:** Summarizing multiple error signals into a single error status. If at least one error is present (`TRUE`), the combined error status is activated.
- **Lever Logic:** Creates a allow when one of several possible sources is active.
- **Basic Building Block in More Complex Logic Circuits:** As part of AND-OR logic gates (SOPs - Sum of Products) to implement switching functions.

⚖️ Comparison with Similar Building Blocks

- **AND_BOOL_2:** Performs a logical AND operation. The result is `TRUE` only if **both** inputs are `TRUE`.

- **XOR_BOOL_2:** Performs an exclusive OR operation (antivalence). The result is `TRUE` if exactly **one** input is `TRUE`.
- **NOT_BOOL:** Performs a logical NOT operation (negation) on a single input.
- **OR_n_BOOL:** A generic OR gate that can combine a variable number of inputs (n), whereas `OR_BOOL_2` has exactly two fixed inputs. See: OR_2

🛠️ Related Exercises

- Exercise_002b3
- Exercise_160
- Exercise_160b
- Exercise_160b2

Conclusion

The `OR_BOOL_2` is a fundamental, reliable, and easy-to-use function block for implementing the basic logical OR function in 4diac IDE applications. Its event-driven, stateless architecture makes it predictable and easily integrated into the data flow of IEC 61499 systems. It is an indispensable component for implementing control logic and conditional evaluation.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

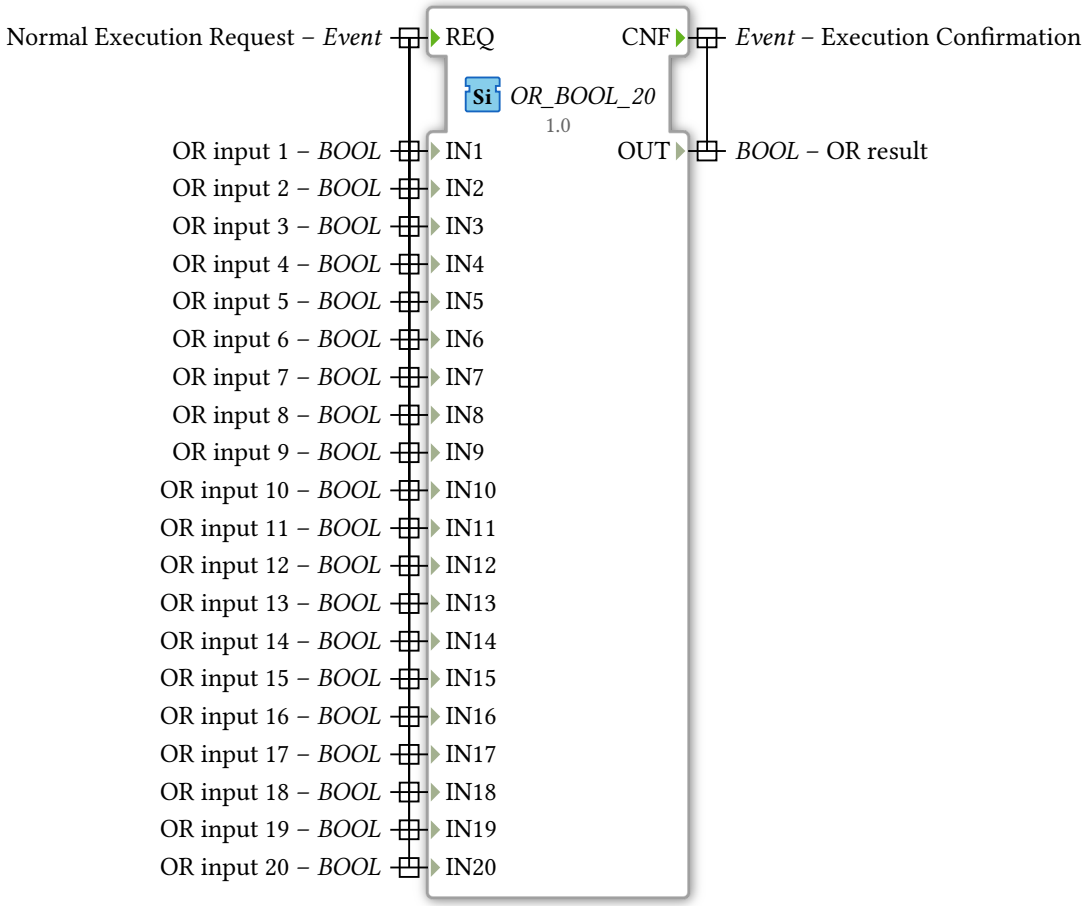
]

OR_BOOL_20

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_20` is a generic block for calculating the logical OR operation (classification `standard boolean function`, IEC 61131-3). It combines 20 separate BOOL inputs (`IN1` to `IN20`) and provides the result at the output `OUT`. It extends the standard OR_BOOL family (previously up to `OR_BOOL_16`) with the variant that has 20 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 20 data inputs (`IN1` to `IN20`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.
- **IN11 (OR input 11):** Boolean input 11.
- **IN12 (OR input 12):** Boolean input 12.
- **IN13 (OR input 13):** Boolean input 13.
- **IN14 (OR input 14):** Boolean input 14.
- **IN15 (OR input 15):** Boolean input 15.
- **IN16 (OR input 16):** Boolean input 16.
- **IN17 (OR input 17):** Boolean input 17.

- **IN18 (OR input 18):** Boolean input 18.
- **IN19 (OR input 19):** Boolean input 19.
- **IN20 (OR input 20):** Boolean input 20.

Data Outputs

- **OUT (OR result):** The result of the OR operation of all 20 inputs. The output is TRUE if at least one input is TRUE and FALSE only if all inputs `IN1` to `IN20` are FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 20 inputs are read and the operation `OUT = IN1 OR IN2 OR ... OR IN20` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_OR_BOOL'`; this instance is fixed to 20 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Collective fault:** Combine several fault or warning signals into one common signal.
- **Multiple triggers:** An action is to start as soon as at least one of many conditions or buttons is active.

⚖️ Comparison with Similar Function Blocks

- `AND_BOOL_20`: Performs the AND operation: the result is TRUE only if all inputs are TRUE, while `OR_BOOL_20` needs just one TRUE input.
- `OR_BOOL_16`: Sibling block of the family with 16 inputs.

Conclusion

`OR_BOOL_20` is a simple, event-driven block for the OR combination of 20 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

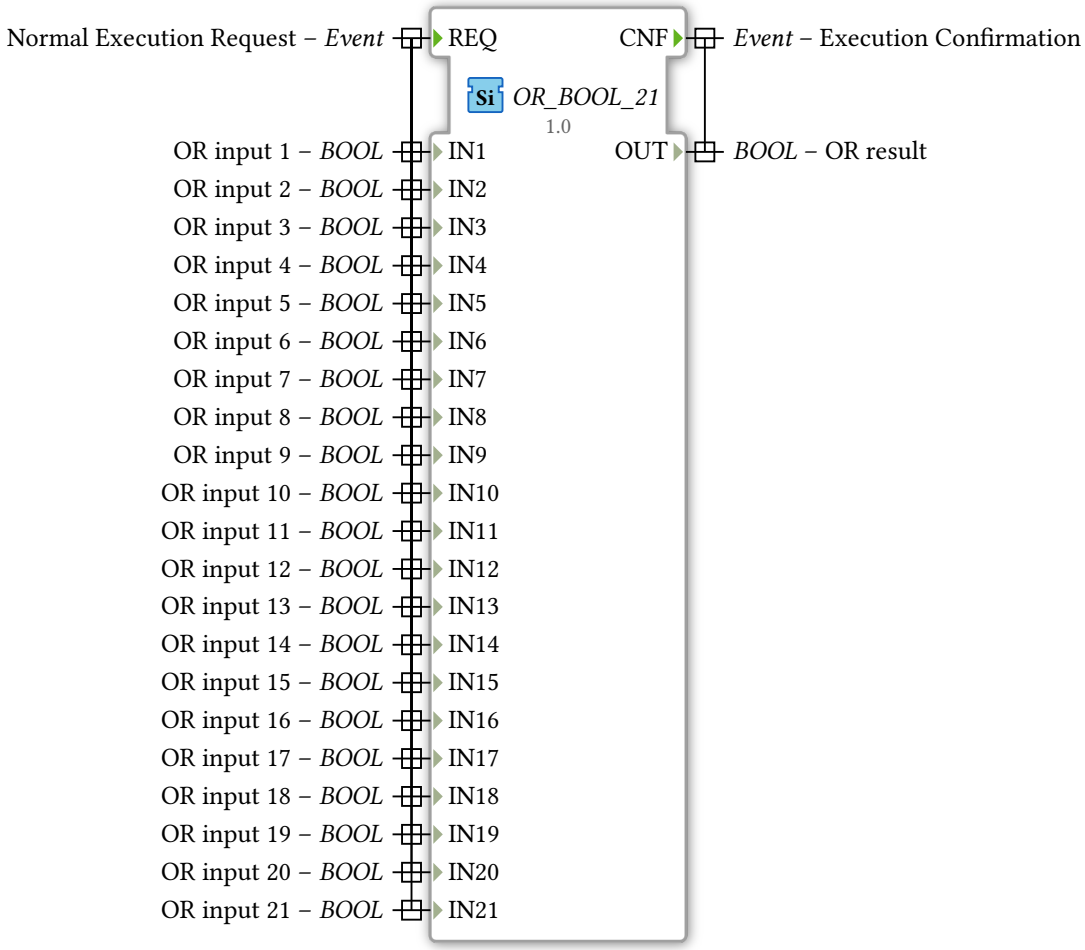
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

OR_BOOL_21

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_21` is a generic block for calculating the logical OR operation (classification `standard boolean function`, IEC 61131-3). It combines 21 separate BOOL inputs (`IN1` to `IN21`) and provides the result at the output `OUT`. It extends the standard OR_BOOL family (previously up to `OR_BOOL_16`) with the variant that has 21 inputs.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation. The event is associated with all 21 data inputs (`IN1` to `IN21`).

Event Outputs

- **CNF (Execution Confirmation):** Signals completion of the calculation. The event is associated with the data output `OUT`.

Data Inputs

- **IN1 (OR input 1):** Boolean input 1.
- **IN2 (OR input 2):** Boolean input 2.
- **IN3 (OR input 3):** Boolean input 3.
- **IN4 (OR input 4):** Boolean input 4.
- **IN5 (OR input 5):** Boolean input 5.
- **IN6 (OR input 6):** Boolean input 6.
- **IN7 (OR input 7):** Boolean input 7.
- **IN8 (OR input 8):** Boolean input 8.
- **IN9 (OR input 9):** Boolean input 9.
- **IN10 (OR input 10):** Boolean input 10.
- **IN11 (OR input 11):** Boolean input 11.
- **IN12 (OR input 12):** Boolean input 12.
- **IN13 (OR input 13):** Boolean input 13.
- **IN14 (OR input 14):** Boolean input 14.
- **IN15 (OR input 15):** Boolean input 15.

- **IN16 (OR input 16):** Boolean input 16.
- **IN17 (OR input 17):** Boolean input 17.
- **IN18 (OR input 18):** Boolean input 18.
- **IN19 (OR input 19):** Boolean input 19.
- **IN20 (OR input 20):** Boolean input 20.
- **IN21 (OR input 21):** Boolean input 21.

Data Outputs

- **OUT (OR result):** The result of the OR operation of all 21 inputs. The output is TRUE if at least one input is TRUE and FALSE only if all inputs `IN1` to `IN21` are FALSE.

Adapter

This function block has no adapter interfaces.

Operation

Upon each occurrence of the event `REQ`, all 21 inputs are read and the operation `OUT = IN1 OR IN2 OR ... OR IN21` is calculated. The result is then made available at `OUT` and `CNF` is triggered.

Technical Features

- **Generic block:** The type carries the attribute `GenericClassName = 'GEN_OR_BOOL'`; this instance is fixed to 21 inputs.
- **Package:** `iec61131::booleanOperators`.
- **Stateless:** The block is memoryless: `OUT` depends solely on the input values at the time of `REQ`.

Application Scenarios

- **Collective fault:** Combine several fault or warning signals into one common signal.
- **Multiple triggers:** An action is to start as soon as at least one of many conditions or buttons is active.

⚖️ Comparison with Similar Function Blocks

- `AND_BOOL_21`: Performs the AND operation: the result is TRUE only if all inputs are TRUE, while `OR_BOOL_21` needs just one TRUE input.
- `OR_BOOL_16`: Sibling block of the family with 16 inputs.

Conclusion

`OR_BOOL_21` is a simple, event-driven block for the OR combination of 21 Boolean signals with a fixed interface.

🌐 Related topic subpages on ms-muc-docs.de

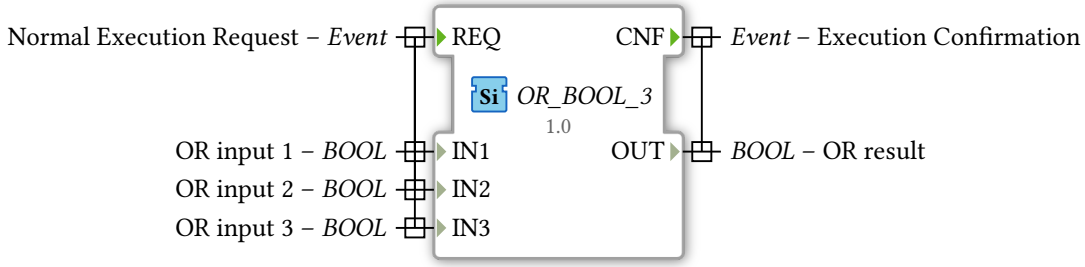
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

OR_BOOL_3

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_3` is a standardized block according to IEC 61131-3 for performing a logical OR operation. It calculates the logical OR (disjunction) of three Boolean input signals. The block is implemented as a generic function and is executed on every trigger event.



Interface Structure

The function block has one event input, one event output, three data inputs, and one data output. No adapter interfaces are available.

Event Inputs

- **REQ** (Normal Execution Request): This event triggers the calculation of the OR function. Upon receiving `REQ`, the current values at inputs `IN1`, `IN2`, and `IN3` are read, and the result is calculated.

Event Outputs

- **CNF** (Execution Confirmation): This event signals the completion of the calculation. It is output along with the calculated result at output `OUT`.

Data Inputs

- **IN1** (BOOL): First OR input signal.
- **IN2** (BOOL): Second OR input signal.
- **IN3** (BOOL): Third OR input signal.

Data Outputs

- **OUT** (BOOL): Result of the OR operation (`IN1` OR `IN2` OR `IN3`).

Adapters

No adapter interfaces are available.

Functionality

Whenever the event `REQ` occurs, the function block performs a logical OR operation on the three Boolean inputs `IN1`, `IN2`, and `IN3`. The result of this operation is output to data output `OUT`. Immediately after the calculation, the confirmation event `CNF` is triggered to signal the successful completion of the operation.

The logical function can be described by the following truth table:

IN1	IN2	IN3	OUT (IN1 OR IN2 OR IN3)		
0	0	0	0		
0	0	1	1		
0	1	0	1		
0	1	1	1		
1	0	0	1		
1	0	1	1		
1	1	1	1		
0 …	0	1	1	1	1
0	0	1	1	1	1
The output `OUT` is `TRUE` (1) if and only if at least one of the three inputs is `TRUE`.					

Technical Features

- **Generic Block:** The block is implemented as an instance of a generic OR block (`GEN_OR`), enabling consistent integration into libraries.
- **Event-Driven Execution:** Function execution is strictly linked to the occurrence of the `REQ` event, ensuring deterministic control in real-time systems.
- **Standard Compliance:** Conforms to the “standard boolean function” classification according to IEC 61131-3.

State Overview

The function block has no internal state (memory-free, combinatorial). Its output depends solely on the current values of the inputs at the time of the trigger event `REQ`.

Application Scenarios

- **Safety and Monitoring Logic:** Combination of multiple fault sensors (e.g., temperature too high, pressure too low, level critical), where each sensor can trigger an alarm.
- **Control Logic:** Activation of an actuator (e.g., a pump or motor) when at least one of several conditions is met (e.g., manual start command OR automatic setpoint reached OR emergency start active).
- **Chaining of Logic Operations:** As part of more complex logical expressions in conjunction with AND and NOT blocks.

⚖️ Comparison with Similar Blocks

- **OR_BOOL_2:** Combines only two inputs. `OR_BOOL_3` extends this functionality to include a third input. See also: OR_3
- **GEN_OR:** The underlying generic function block, which can be configured for a variable number of inputs. `OR_BOOL_3` is a specific instance of this, fixed to three inputs.
- **AND_n_BOOL:** Performs a logical AND operation. The result is `TRUE` only if **all** inputs are `TRUE`, whereas for the OR operation, **just one** `TRUE` input is sufficient.

🛠️ Related Exercises

- Exercise_002a5b

Conclusion

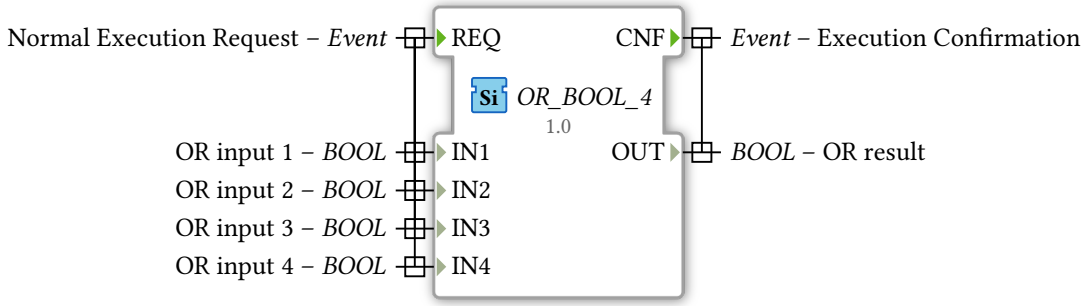
The `OR_BOOL_3` is a fundamental, reliable, and standards-compliant function block for the logical OR operation of three signals. Its event-driven, stateless operation makes it ideal for use in clearly structured control sequences in automation technology. It represents an essential component for implementing monitoring, safety, and control logic.

OR_BOOL_4

• – ○ • – ○ • – ○ •

Introduction

The function block `OR_BOOL_4` is a standard function block for calculating the logical OR operation. It performs an OR operation with up to four Boolean input values and outputs the result as a single Boolean value. The function block follows the IPO (Event-driven Processing) principle and is designed for use in control applications according to IEC 61131-3.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. Upon this event, the values at the data inputs `IN1` to `IN4` are read, and the OR operation is calculated.

Event Outputs

- **CNF**: Confirms completion of the operation. This event is triggered after the result is calculated and outputs the value to the data output `OUT`.

Data Inputs

- **IN1**: Boolean OR input 1.
- **IN2**: Boolean OR input 2.

- **IN3**: Boolean OR input 3.
- **IN4**: Boolean OR input 4.

Data Outputs

- **OUT**: Boolean result of the OR operation of all active inputs.

Adapters

This function block does not have any adapter interfaces.

Functionality

When the event `REQ` occurs, the current values at the four Boolean inputs `IN1`, `IN2`, `IN3`, and `IN4` are read. The function block calculates the logical OR of these four values according to the following rule:

The result `OUT` is `TRUE` (1) if at least one of the four inputs is `TRUE`. The result is `FALSE` (0) only if all four inputs are `FALSE`.

The result `FALSE` (0) is `FALSE`. After the calculation, the confirmation event `CNF` is triggered, and the calculated value is carried over to the output `OUT`.

Technical Features

- **Generic Block**: The block is marked as a generic block (`GEN_OR`). This allows it to be used as a basis for creating OR blocks with a different number of inputs.
- **Event-driven**: The operation is only executed upon an incoming `REQ` event, ensuring resource-efficient and deterministic processing in the control system.
- **Standard-compliant**: Conforms to the IEC 61131-3 classification for standard Boolean functions.

State Overview

The function block has no internal state (memory-free). Its behavior is combinatorial and depends solely on the current values at the inputs at the time of the `REQ` event.

Application Scenarios

- **Linking of Alarm Signals**: Combining multiple status or fault alarms where a single signal should trigger an action (e.g., “Stop machine if sensor A OR sensor B OR emergency stop is activated”).
- **Logic in Safety Controllers**: Implementing OR gates in safety circuits.
- **Simplifying Control Logic**: As a clearly defined building block for structuring programs in function block diagrams (FBDs) or sequenced function charts (SFCs).
-

⚖️ Comparison with similar function blocks

- **`OR_BOOL_2` / `OR_BOOL_8`**: These blocks perform the identical logical OR operation, but with a different number of inputs (2 and 8, respectively). The choice of function block depends on the required number of signals to be combined. See: OR_4
- **`AND_BOOL_4`**: Performs a logical AND operation. The result is only `TRUE` if *all* inputs are `TRUE`.
- **`XOR_BOOL_4`**: Performs an exclusive OR (XOR) operation. The result is `TRUE` if there is an odd number of inputs, and `TRUE` if there is an odd number of inputs.

Conclusion

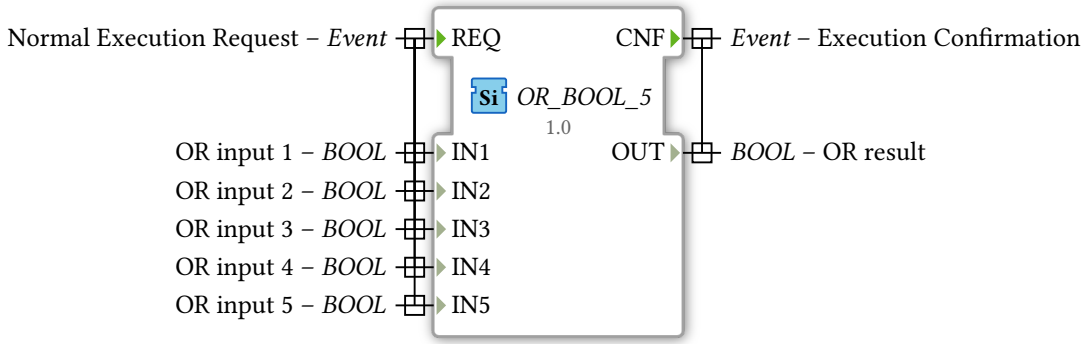
The `OR_BOOL_4` is a basic, reliable, and standards-compliant function block for implementing four-input OR logic. Its event-driven, stateless operation makes it ideal for integration into larger, event-driven control applications according to IEC 61131-3. Its generic nature also makes it a suitable basis for library extensions.

OR_BOOL_5

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_5` is a standard function block for calculating the logical OR operation. It performs an OR operation on five separate Boolean input signals. The function block belongs to the category of standard Boolean functions according to IEC 61131-3 and is implemented as a generic function block.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** This event triggers the execution of the function block. Upon receiving `REQ`, the values of all five inputs (IN1 to `IN5`) are read, and the OR operation is calculated.

Event Outputs

- **CNF (Execution Confirmation):** This event signals the completion of the calculation. It is output along with the calculated result `OUT`.

Data Inputs

- **IN1 (BOOL):** OR Input 1.
- **IN2 (BOOL):** OR Input 2.
- **IN3 (BOOL):** OR Input 3.
- **IN4 (BOOL):** OR Input 4.
- **IN5 (BOOL):** OR Input 5.

Data Outputs

- **OUT (BOOL):** The result of the OR operation on all five inputs. The output is `TRUE` if at least one of the inputs `IN1` to `IN5` has the value `TRUE`.

Adapters

This function block does not have any adapter interfaces.

Functionality

The functionality is deterministic and event-driven:

1. The arrival of the event `REQ` starts the execution.
2. The block reads the current values of the five Boolean inputs `IN1` to `IN5`.
3. The logical OR operation `OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5` is calculated.
4. The result is provided at the data output `OUT`, and simultaneously the confirmation event `CNF` is triggered to signal the successful completion of the operation to subsequent blocks.

Technical Features

- **Generic Block:** The block is marked with the attribute `eclipse4diac::core::GenericClassName` as `'GEN_OR'`. This indicates that it is part of a family of generic OR gates with varying numbers of inputs.
- **Package:** The gate is included in package `iec61131::bitwiseOperators`.
- **Event-driven execution:** Computation only occurs when requested by a `REQ` event, enabling energy-efficient and deterministic execution.

State overview

The function block has no internal state in the sense of a memory. Its behavior is purely combinatorial and depends solely on the current input values at the time of the `REQ` event. The block returns to a passive wait state after the `CNF` event is triggered.

Application Scenarios

Typical applications can be found in control logic, where a condition is considered fulfilled if one of several possible subconditions is met:

- Monitoring: Alarm triggering when at least one of several monitoring sensors exceeds a threshold.
- Enabling logic: Starting a machine when at least one of several safety devices is activated.
- Selection or prioritization logic in interconnected control systems.

⚖️ Comparison with Similar Function Blocks

In contrast to this function block with five fixed inputs, 4diac often also includes:

- **OR_BOOL_2, OR_BOOL_3, etc.:** Same functionality, but with a smaller, fixed number of inputs (e.g., 2, 3, 4). See: OR_5
- **GEN_OR (Generic OR):** A higher-level, configurable function block that can have a variable number of inputs depending on its instantiation. `OR_BOOL_5` is a specific instantiation of this principle.
- **AND_BOOL_5:** Performs a logical AND operation across all five inputs (the result is only `TRUE` if *all* inputs are `TRUE`).

Conclusion

The `OR_BOOL_5` is a simple, robust, and standards-compliant function block for implementing an OR operation with five inputs. Its event-driven architecture integrates seamlessly with 4diac's function block networks and is ideally suited for clear and concise logic connections in automation applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

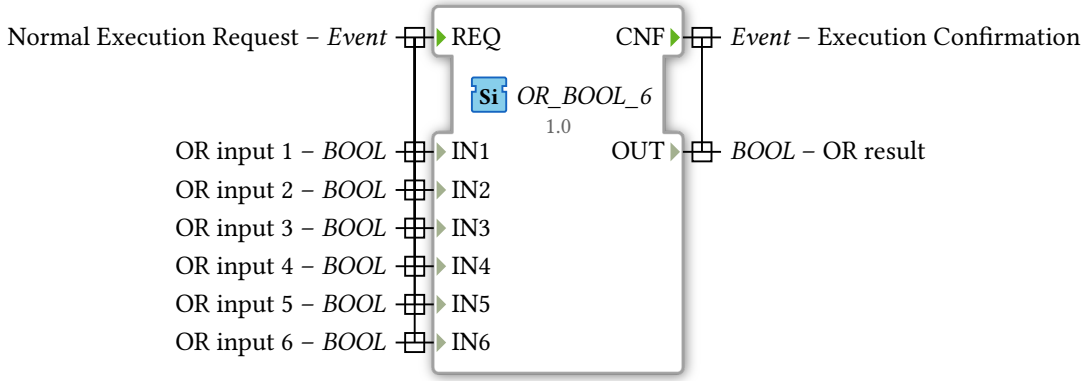
]

OR_BOOL_6

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_6` is a generic block for calculating the logical OR operation. It performs an OR operation on six separate BOOL inputs and outputs the result on a single output. The block complies with the IEC 61131-3 standard and is designed for use in control applications where multiple conditions must be combined to trigger an action.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** This event triggers the calculation of the OR function. It is linked to all six data inputs (`IN1` to `IN6`).

Event Outputs

- **CNF (Execution Confirmation):** This event signals the completion of the calculation. It is output along with the calculated result at data output `OUT`.

Data Inputs

- **IN1 (BOOL):** OR input 1.
- **IN2 (BOOL):** OR input 2.
- **IN3 (BOOL):** OR input 3.
- **IN4 (BOOL):** OR input 4.
- **IN5 (BOOL):** OR input 5.
- **IN6 (BOOL):** OR input 6.

Data Outputs

- **OUT (BOOL):** Result of the OR operation on all six inputs. The output is `TRUE` if at least one of the inputs is `TRUE`.

Adapter

This function block has no adapter interfaces.

Operation

The operation is deterministic and event-driven. Upon each occurrence of the input event `REQ`, the current values of all six BOOL inputs (`IN1` to `IN6`) are read. The logical OR operation of these values is then calculated. The result of this operation is immediately set at the data output `OUT`, and simultaneously, the acknowledgment event `CNF` is triggered to signal the completion of the operation to subsequent blocks.

The logical operation corresponds to the formula: `OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5 OR IN6`

Technical Features

- **Generic Block:** The block is marked as a generic block (`GEN_OR`). This means it can serve as the basis for creating similar blocks with a different number of inputs.
- **Event-Driven Execution:** The calculation only occurs when requested by an event, enabling efficient and demand-driven processing in the control system.
- **Fixed Number of Inputs:** This specific instance is defined for exactly six inputs.

State Overview

The function block has no internal state or memory. Its behavior is purely combinatorial and depends solely on the current values of the inputs at the time of the `REQ` event. After outputting `CNF`, the block remains in a waiting state until the next `REQ`.

Application Scenarios

Typical applications include monitoring and safety logic where an action should be triggered when one of several possible conditions is met.

- **Emergency Stop Logic:** The block can be used to monitor multiple emergency stop buttons. If any of the buttons is pressed (`TRUE`), a shutdown signal (`OUT = TRUE`) is generated.
- **Alarm Aggregation:** Multiple sensors, each capable of detecting a fault condition, are interconnected. A general alarm is triggered as soon as at least one sensor reports a fault.
- **Multiple Operator Control:** In a machine, multiple operator stations can start a process. The block combines all start commands into a single start signal.
-

⚖️ Comparison with similar function blocks

- **AND_BOOL_6:** Performs a logical AND operation. The result is only `TRUE` if **all** inputs are `TRUE`. In contrast, `OR_BOOL_6` requires only **one** true input.
- **XOR_BOOL:** Performs an exclusive OR (XOR) operation. The result is `TRUE` if there is an odd number of inputs, `TRUE`. `OR_BOOL_6` does not distinguish between one or more true conditions. See: OR_6
- **Blocks with Variable Number of Inputs:** OR blocks with two (`OR`) or other numbers of inputs exist. `OR_BOOL_6` offers a fixed interface with six inputs, optimized for specific applications.

Conclusion

The `OR_BOOL_6` function block is a simple, robust, and standards-compliant building block for logical signal combination. Its strength lies in its clear, event-driven interface and the fixed number of six inputs, making it ideal for applications that require precisely this number of conditions to be combined. As a generic building block, it also supports reuse and customization within the 4diac IDE.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

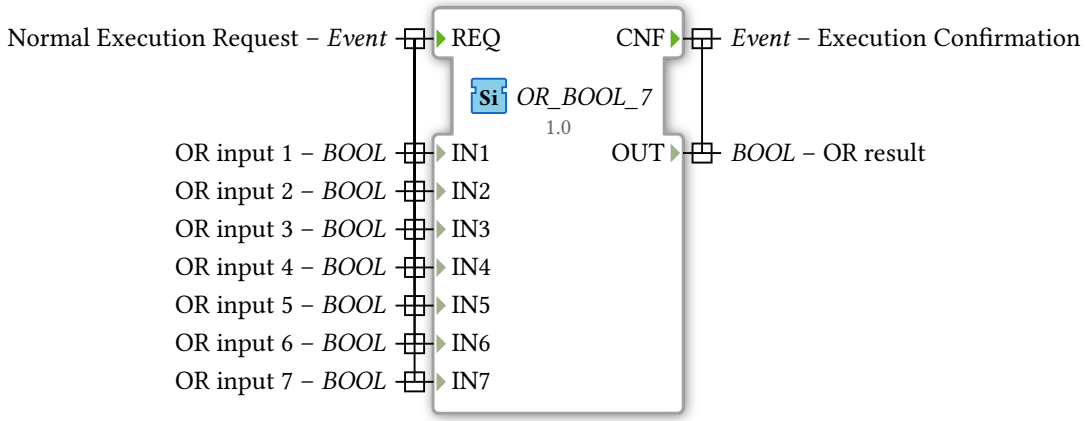
]

OR_BOOL_7

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_7` is a standard function block for calculating the logical OR operation. It performs an OR operation on seven separate Boolean inputs and provides the result at a single output. This function block is part of the IEC 61131-3 compliant library for bitwise operations.



Interface Structure

Event Inputs

- **REQ** (Normal Execution Request): This event triggers the calculation of the OR function. It is associated with all seven data inputs (`IN1` to `IN7`).

Event Outputs

- **CNF** (Execution Confirmation): This event signals the completion of the calculation. It is output along with the calculated result at data output `OUT`.

Data Inputs

- **IN1** (BOOL): OR input 1.
- **IN2** (BOOL): OR input 2.
- **IN3** (BOOL): OR input 3.
- **IN4** (BOOL): OR input 4.
- **IN5** (BOOL): OR input 5.
- **IN6** (BOOL): OR input 6.
- **IN7** (BOOL): OR input 7.

Data Outputs

- **OUT** (BOOL): Result of the OR operation of all seven inputs.

Adapters

This function block does not have any adapter interfaces.

Functionality

Upon receiving the event `REQ`, the function block reads the values of all seven Boolean inputs (`IN1` to `IN7`). A logical OR operation is then applied to these values. The result of this operation is set at the data output `OUT`, and simultaneously, the confirmation event `CNF` is triggered.

The logical function can be described as follows:
OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5 OR IN6 OR IN7`

This means that the output OUT is TRUE (1) if at least one of the seven inputs is TRUE. The output is FALSE (0) only if all seven inputs are FALSE.

Technical Features

- **Generic Block:** The function block is marked as a generic implementation (GEN_OR), indicating that the core can be reused for similar blocks.
- **Hard-wired Logic:** The OR operation is hard-coded for exactly seven inputs. For a different number of inputs, a corresponding block (e.g., OR_BOOL_2, OR_BOOL_4) must be used.
- **Event-driven Execution:** The calculation is performed only when the input event REQ occurs, enabling resource-efficient and deterministic processing in the real-time system.

State Overview

The function block has no internal state or memory. Its behavior is purely combinatorial and event-driven:

1. **Idle State:** Waits for the event REQ.
2. **Calculation State:** Upon REQ, the inputs are read and the OR operation is performed.
3. **Output State:** The result is output to OUT, and the event CNF is generated. The block then returns to the idle state.

Application Scenarios

- **Monitoring Logic:** Combines multiple error or warning signals into a common alarm output. If one of seven possible sources of interference is activated, a common warning light illuminates.
- **Leverage Logic:** Combines multiple enable signals (e.g., from different safety systems) for a process step. The process starts when at least one condition is met.
- **Sensor Linking:** Evaluates multiple limit detectors to check if at least one sensor has exceeded a threshold.

⚖️ Comparison with Similar Function Blocks

- **AND_BOOL_7:** Performs a logical AND operation. The output is only TRUE if *all* inputs are TRUE.
- **XOR_BOOL_7:** Performs an exclusive OR (XOR) operation. The output is TRUE if an odd number of inputs are TRUE.
- **OR_n_BOOL (n=2,4,...):** Function blocks of the same family that provide the OR function for a different, fixed number of inputs. The choice of block depends on the required number of signals. See: OR_7
- **Generic FB (GEN_OR):** The underlying generic implementation used to create the specific OR_n_BOOL variants.

Conclusion

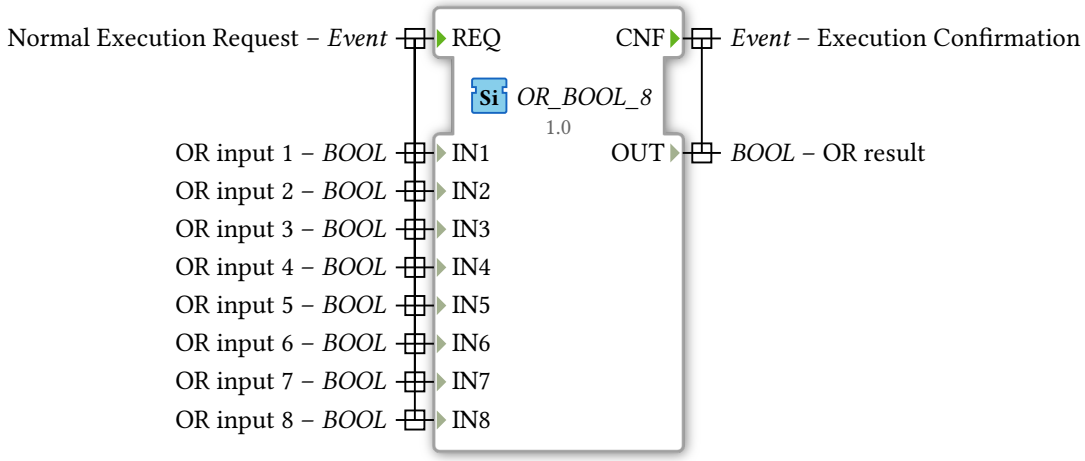
The OR_BOOL_7 is a simple, robust, and standardized function block for the logical OR operation of seven Boolean signals. Its event-driven architecture makes it ideal for integration into control engineering applications according to IEC 61131-3, where it operates reliably and deterministically. It is the right choice when exactly seven binary states need to be combined to produce a common result.

OR_BOOL_8

• – ◦ • – ◦ • – ◦ •

Introduction

The function block OR_BOOL_8 is a standardized block for calculating the logical OR operation. It performs the OR operation on eight separate Boolean input values and outputs the result on a single Boolean output. This block is part of the IEC 61131-3 compliant library for bitwise operations and is used for simple and structured logic processing in control applications.



Interface Structure

Event Inputs

- **REQ** (Normal Execution Request): This event triggers the execution of the function block. Upon its arrival, all eight data inputs (`IN1` to `IN8`) are read and the OR operation is calculated.

Event Outputs

- **CNF** (Execution Confirmation): This event signals the completion of the calculation. It is output along with the calculated result at data output `OUT`.

Data Inputs

- **IN1** (BOOL): OR input 1.
- **IN2** (BOOL): OR input 2.
- **IN3** (BOOL): OR input 3.
- **IN4** (BOOL): OR input 4.
- **IN5** (BOOL): OR input 5.
- **IN6** (BOOL): OR input 6.
- **IN7** (BOOL): OR input 7.
- **IN8** (BOOL): OR input 8.

Data Outputs

- **OUT** (BOOL): Result of the OR operation of all eight inputs. The output is `TRUE` (1) if at least one of the inputs `IN1` to `IN8` has the value `TRUE`. Otherwise, the output is `FALSE` (0).

Adapter

This function block does not use any adapter interfaces.

Operation

The operation is event-driven and follows a simple request-confirm cycle:

1. The arrival of the event `REQ` starts execution.
2. The block reads the current values of all eight Boolean inputs (`IN1` to `IN8`).
3. The logical OR operation is calculated across all eight inputs: `OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5 OR IN6 OR IN7 OR IN8`.
4. The result is output at `OUT`, and simultaneously, the confirmation event `CNF` is triggered to signal the successful completion of the operation.

Technical Features

- **Generic Block:** This block is marked as a generic block (`GEN_OR`), meaning it can serve as a basis for creating similar OR blocks with a different number of inputs.
- **Standard Compliant:** It is classified according to the IEC 61131-3 standard and implements a standardized Boolean function.

- **Deterministic Behavior:** Execution is deterministic and does not result in any internal state storage between calls.

State Overview

The function block `OR_BOOL_8` is stateless (combinatorial logic). It has no internal memory. The output `OUT` is, at any given time, solely a direct function of the current input values read in by the last `REQ` event.

Application Scenarios

Typical applications include:

- **Monitoring Logic:** Combining multiple error or warning signals (e.g., from different sensors or subsystems) into a single alarm output.
- **Enable Logic:** Checking whether at least one of several possible enable conditions (e.g., "Manually enabled" OR "Automatically enabled") is met.
- **Linking Button Inputs:** In control panels to trigger an action when one of several buttons is pressed.

⚖️ Comparison with similar function blocks

- **`OR_BOOL_2`, `OR_BOOL_4`:** These are structurally identical function blocks, but they only connect two or four inputs, respectively. `OR_BOOL_8` offers a higher number of inputs without having to cascade multiple instances of blocks with fewer inputs. See: OR_8
- **`AND_n_BOOL`:** Performs the logical AND operation. The result is only `TRUE` if *all* inputs are `TRUE`, whereas for the OR operation, a single `TRUE` input is sufficient.
- **`XOR_n_BOOL`:** Performs an exclusive OR (XOR) operation. The result is `TRUE` if there is an odd number of inputs; it is `TRUE`, which differs from inclusive OR logic.

Conclusion

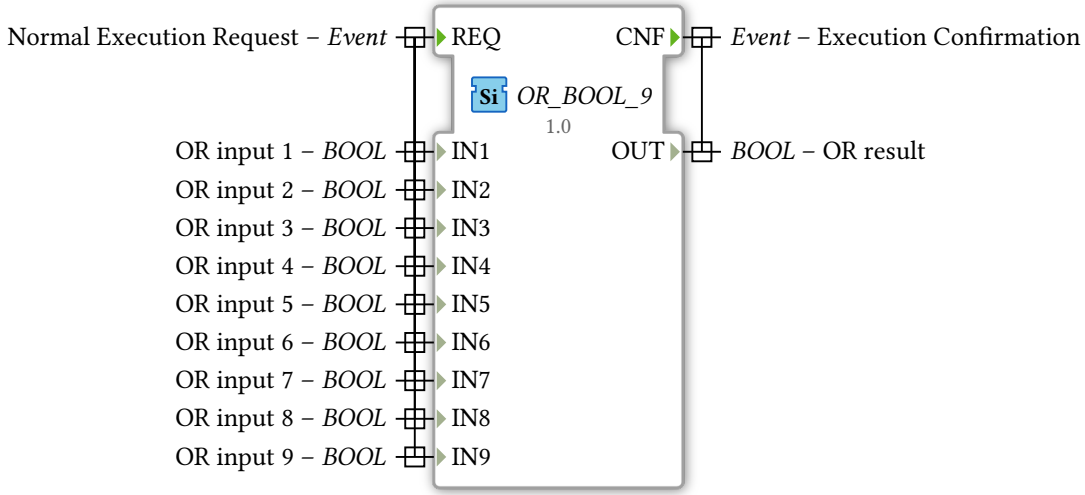
The `OR_BOOL_8` function block is a fundamental, reliable, and easy-to-use building block for Boolean logic processing. Its strength lies in its clear, standards-compliant interface and efficient processing of up to eight signals into a single statement. It is the first choice for applications requiring an inclusive OR operation across multiple signals.

OR_BOOL_9

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_BOOL_9` is a standardized building block according to IEC 61131-3 for calculating the logical OR operation. It performs the OR operation via nine separate Boolean inputs. The block follows the event-driven execution model of the 4diac IDE: The calculation is triggered by an incoming event, and the result is output along with an acknowledgment event.



Interface Structure

Event Inputs

- **REQ**: Triggers the execution of the function block. Upon arrival of this event, all nine input values (`IN1` to `IN9`) are read, and the OR operation is calculated.

Event Outputs

- **CNF**: This event is generated after the calculation is complete. It signals that the result is valid and available at the data output `OUT`.

Data Inputs

- **IN1** (BOOL): OR input 1.
- **IN2** (BOOL): OR input 2.
- **IN3** (BOOL): OR input 3.
- **IN4** (BOOL): OR input 4.
- **IN5** (BOOL): OR input 5.
- **IN6** (BOOL): OR input 6.
- **IN7** (BOOL): OR input 7.
- **IN8** (BOOL): OR input 8.
- **IN9** (BOOL): OR input 9.

Data Outputs

- **OUT** (BOOL): Result of the logical OR operation of all nine inputs. The output is `TRUE` (1) if at least one of the inputs is `TRUE`. It is only `FALSE` (0) if all nine inputs are `FALSE`.

Adapter

This function block has no adapter interfaces.

Operation

The `OR_BOOL_9` block operates according to the following algorithm:

1. **Trigger**: The incoming event `REQ` activates the block.
2. **Input Collection**: The current values of all nine Boolean inputs (`IN1` to `IN9`) are read.
3. **Calculation**: The logical OR function is applied to the input values. The operation can be described as `OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5 OR IN6 OR IN7 OR IN8 OR IN9`.
4. **Output**: The calculated result is set to the data output `OUT`.
5. **Acknowledgement**: Immediately after the output is set, the acknowledgement event `CNF` is generated to inform downstream blocks about the availability of the new result.

Technical Features

- **Generic Function Block**: The block is implemented as a generic function block (attribute `eclipse4diac::core::GenericClassName` is set to `'GEN_OR'`). This means it can serve as the basis for deriving similar OR blocks with a different number of inputs.
- **Standard Compliance**: It is explicitly classified as a standardized Boolean function according to IEC 61131-3.
- **Deterministic Behavior**: The execution time is constant and predictable, as it is a simple logical operation.

State Overview

The block has no internal state or memory. Its behavior is purely combinatorial and depends solely on the current input values at the time of the `REQ` event. Previous calculations have no influence on the current result.

Application Scenarios

- **Monitoring Logic**: Combination of multiple error or status messages (e.g., from nine sensors), where even a single message (`TRUE`) triggers a collective alarm message (`OUT = TRUE`).
- **Enable Logic**: In safety circuits, where one of several possible enable conditions is sufficient to start a process step.
- **Simplification of Logic Networks**: Replaces multiple cascaded two-input OR gates, improving clarity in a function block diagram (FBD).

•

⚖️ Comparison with similar building blocks

- **`OR_BOOL_2`, `OR_BOOL_4` etc.**: These blocks offer the same OR functionality, but for a smaller number of inputs (2, 4, ...). `OR_BOOL_9` is optimized for applications with nine signals to be combined. See: OR_9
- **`AND_BOOL_9`**: Performs the logical AND operation. The result is only `TRUE` if *all* inputs are `TRUE`, unlike OR, where *at least one* input must be `TRUE`.
- **`XOR_BOOL_9`**: Calculates the exclusive OR (antivalence). The result is `TRUE` if there is an odd number of inputs, resulting in `TRUE`.

Conclusion

The `OR_BOOL_9` is a robust and easy-to-use standard block for the logical OR operation of nine Boolean signals. Its event-driven, stateless architecture makes it a reliable and efficient component for control and monitoring logic in 4diac-based automation systems. Using this specialized block instead of a cascade of basic gates improves the readability and maintainability of the application code.

🌐 Related topic subpages on ms-muc-docs.de

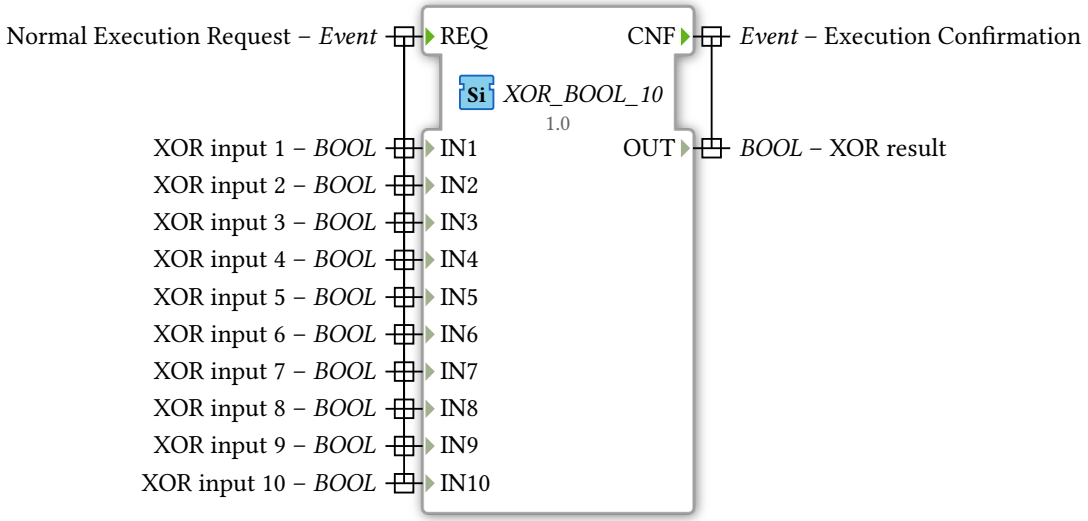
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

XOR_BOOL_10

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `XOR_BOOL_10` is a generic function block for calculating the logical exclusive OR (XOR) operation on up to ten Boolean input signals. It complies with the IEC 61131-3 standard and is designed for use in the 4diac IDE. The function block evaluates all connected inputs for each request and returns the corresponding result.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** This event triggers the calculation of the XOR function. It is linked to all ten data inputs (`IN1` to `IN10`).

Event Outputs

- **CNF (Execution Confirmation):** This event signals the completion of the calculation. It is output along with the calculated result at the data output `OUT`.

Data Inputs

- **IN1 (XOR input 1):** Boolean input 1.

- **IN2 (XOR input 2):** Boolean input 2.
- **IN3 (XOR input 3):** Boolean input 3.
- **IN4 (XOR input 4):** Boolean input 4.
- **IN5 (XOR input 5):** Boolean input 5.
- **IN6 (XOR input 6):** Boolean input 6.
- **IN7 (XOR input 7):** Boolean input 7.
- **IN8 (XOR input 8):** Boolean input 8.
- **IN9 (XOR input 9):** Boolean input 9.
- **IN10 (XOR input 10):** Boolean input 10.

Data Outputs

- **OUT (XOR result):** Boolean result of the XOR operation of all active inputs Inputs.

Adapter

This function block has no adapter interfaces.

Functionality

Upon receiving the `REQ` event, the block reads the values of all ten Boolean data inputs (`IN1`...`IN10`). It then performs the logical XOR operation on all these inputs. The XOR function outputs a `TRUE` signal (1) at the output `OUT` if and only if there is an odd number of inputs (`TRUE`). If there are zero, two, four, six, eight, or ten inputs (`TRUE`), the result is `FALSE`. After the calculation, the `CNF` event is triggered along with the current result value.

Technical Features

- **Generic Block:** The block is marked as a generic block (`GEN_XOR`). This allows for flexible reuse and potential specialization.
- **Fixed Number of Inputs:** Unlike blocks with a variable number of inputs, this block offers exactly ten fixed inputs. Unused inputs should be set to a defined logic level (e.g., `FALSE`).
- **Event-Driven Execution:** The calculation is performed exclusively event-driven by the `REQ` input. There is no cyclic execution.

State Overview

The function block does not have an internal state in the sense of a memory. Its behavior is purely combinatorial and depends on the current input values. The only "state" is waiting for a `REQ` event. Upon receiving this event, the system immediately calculates and sends the `CNF` event.

Application Scenarios

- **Parity Check:** Detects whether an odd number of signals (e.g., error messages, switch positions) are active.
- **Control Logic:** Implements special logic logic in sequence controls where a state change should only occur with an odd activation pattern.
- **Encoding/Decoding:** Simple cryptographic or coding tasks based on the XOR operation.
-

⚖️ Comparison with Similar Blocks

- **Standard XOR Blocks (e.g., 2-Input XOR):** This block extends the classic 2-input XOR function to up to ten inputs in a single block, improving clarity when dealing with many signals. See: XOR_10
- **Blocks with Variable Input Count:** Some libraries offer XOR blocks to which inputs can be dynamically added. `XOR_BOOL_10` has a fixed, explicitly declared interface, which can simplify code analysis.
- **Combinatorial Logic in SFC/ST:** The same functionality could also be implemented in Structured Text (ST) using an expression like `OUT := IN1 XOR IN2 XOR ... XOR IN10;`. The advantage of the FB lies in its clear encapsulation and event-driven interface, which integrates better into FB networks.

Conclusion

The `XOR_BOOL_10` is a specialized and useful function block for applications requiring XOR operations across more than two signals. Its robust, well-documented ten-input interface and event-driven IEC 61499 model make it a reliable building block for implementing parity checks and specialized control logic. However, for simpler operations with fewer inputs or for maximum flexibility, alternative implementations should be considered.

–

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

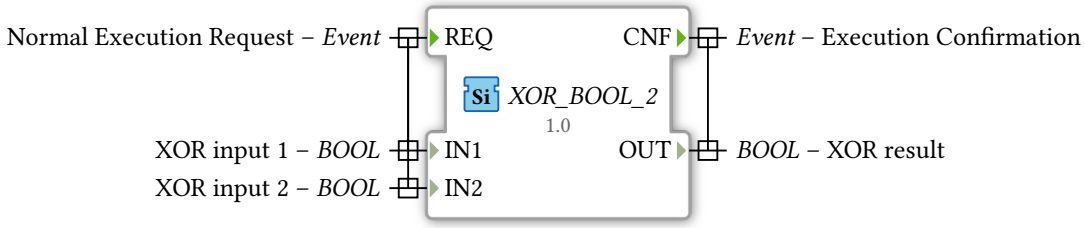
]

XOR_BOOL_2

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `XOR_BOOL_2` is a standard function block for calculating the logical exclusive OR (XOR) operation of two Boolean input signals. It belongs to the category of bitwise operators and is implemented as a generic function block used in control applications according to IEC 61131-3. The block waits for an execution request, calculates the result, and returns it along with an acknowledgment.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the calculation of the XOR operation. The data associated with this event (`IN1`, `IN2`) are read during processing.

Event Outputs

- **CNF**: Execution Confirmation. This event is generated after the calculation and signals the availability of the result. It is associated with the data output `OUT`.

Data Inputs

- **IN1**: Boolean input 1 for the XOR operation.
- **IN2**: Boolean input 2 for the XOR operation.

Data Outputs

- **OUT**: Boolean result of the XOR operation of `IN1` and `IN2`.

Adapters

This function block does not have any adapter interfaces.

Functionality

The `XOR_BOOL_2` block operates in an event-driven manner. Upon the arrival of the event `REQ`, the current values at the data inputs `IN1` and `IN2` are read. The logical XOR operation (`OUT = (IN1 AND NOT IN2) OR (NOT IN1 AND IN2)`) is then performed. The calculated result is output at the data output `OUT`, and simultaneously, the confirmation event `CNF` is activated to signal the completion of the operation to subsequent blocks.

The truth table of the operation is:

IN1	IN2	OUT
0	0	0
0	1	1
1	0	1

1	1	0

Technical Features

- **Generic Block**: The block is identified with the generic class name `'GEN_XOR'`, which facilitates its reusability and potential integration into higher-level structures.
- **Event-Driven Execution**: The functionality is strictly linked to the input events, enabling deterministic and energy-efficient processing in real-time systems.
- **Simple Interface**: The block has a minimal and intuitive interface that corresponds to the classic XOR function commonly used in digital technology.

State Overview

The function block does not have an internal state in the sense of a memory. Its behavior is combinatorial and depends solely on the current input values when the `REQ` event occurs. After outputting `CNF`, the block remains in a waiting state until the next `REQ` event arrives.

Application Scenarios

- **Comparison and Change Detection**: Detects when exactly one of two states (e.g., two limit switches) has changed.
- **Parity Check**: Simple check for an odd number of `TRUE` signals with two inputs.
- **Control Logic**: Implementation of special branches in sequential control systems where a decision must be made when the inputs are different.
- **Safety-Critical Circuits**: Generates signals that are only active when two monitored variables contradict each other (e.g., in emergency stop circuits).
-

⚖️ Comparison with similar building blocks

Compared to other basic logic function blocks like `AND_BOOL_2` or `OR_BOOL_2`, `XOR_BOOL_2` is the only one that returns a `TRUE` result when the inputs are *different*. Blocks like `NOT` only invert a single input. `XOR_BOOL_2` can be considered a special case of an inequality comparison (`NE_BOOL`) for two Boolean values. See: XOR_2

Conclusion

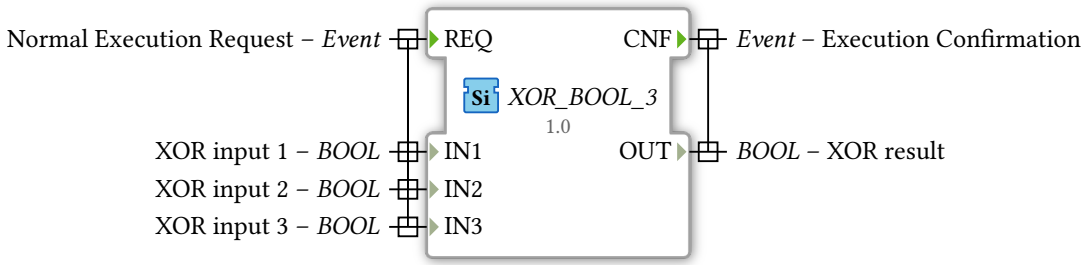
The `XOR_BOOL_2` is a fundamental, reliable, and easy-to-use building block for logical operation in IEC 61131-3-based control applications. Its event-driven, stateless architecture makes it particularly suitable for integration into complex, deterministic control networks. Its clear functionality and minimal interface make it an indispensable tool for every control system developer.

XOR_BOOL_3

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `XOR_BOOL_3` is a standardized function block for calculating the logical exclusive OR (XOR) operation for three Boolean input values. It is implemented as a generic function block and performs its operation on every incoming execution event.



Interface Structure

Event Inputs

- **REQ** (Normal Execution Request): Triggers the calculation of the XOR function. The data associated with this event (`IN1`, `IN2`, `IN3`) are read and processed.

Event Outputs

- **CNF** (Execution Confirmation): Triggered after the calculation is complete. This event returns the calculated result via the corresponding data output `OUT`.

Data Inputs

- **IN1** (BOOL): First Boolean operand for the XOR operation.
- **IN2** (BOOL): Second Boolean operand for the XOR operation.
- **IN3** (BOOL): Third Boolean operand for the XOR operation.

Data Outputs

- **OUT** (BOOL): The result of the triple XOR operation of the inputs `IN1`, `IN2`, and `IN3`.

Adapter

This function block has no adapter interfaces.

Operation

The block operates in an event-driven manner. Upon each occurrence of the event `REQ`, the current values of the three Boolean inputs `IN1`, `IN2`, and `IN3` are read. The logical XOR operation is then calculated for all three inputs. The result is output at the data output `OUT`, and simultaneously, the confirmation event `CNF` is triggered to signal the completion of the operation.

The corresponding logical function is: `OUT = (IN1 XOR IN2) XOR IN3`. For three inputs, the result is `TRUE` (1) if and only if there is an odd number of inputs (`TRUE`) (i.e., 1 or 3 inputs).

Technical Features

- **Generic Block:** The block is marked as a generic block (`'GEN_XOR'`) with the attribute `eclipse4diac::core::GenericClassName`. This allows for potential reuse of the logic for other data types that support the XOR operation.
- **Standard Compliance:** The block is classified as a "standard boolean function" according to the IEC 61131-3 standard.
- **Package Assignment:** The block is assigned to the package `iec61131::bitwiseOperators`.

State Overview

The function block does not have an internal state in the sense of a memory. Its behavior is purely combinatorial and depends solely on the current input values when the `REQ` event arrives. The only "state" is the idle state, in which the block waits for the next `REQ` event.

Application Scenarios

- **Parity Check:** Detects whether an odd number of three Boolean signals are active.
- **Control Logic:** In specialized control algorithms that require "either-or" logic with more than two conditions.
- **Encryption and Coding:** As a fundamental component in simple cryptographic or error-detecting circuits (e.g., for parity bits).
- **Signal Comparison:** Determines whether a third signal differs from the combination of two other signals.
-

⚖️ Comparison with Similar Blocks

- **XOR (2 Inputs):** The standard XOR block with only two inputs. `XOR_BOOL_3` extends this function to three operands. See: XOR_3
- **OR / AND (3 Inputs):** These blocks calculate the logical OR or AND operation of all inputs. The XOR logic is more specialized and differs, in particular, when all three inputs are `TRUE` (OR=TRUE, AND=TRUE, XOR=TRUE).

- **GEN_XOR:** This is the generic basis from which `XOR_BOOL_3` is derived as a specific instance for the data type `BOOL`.
-

Conclusion

The `XOR_BOOL_3` function block is a specialized, standards-compliant building block for reliably calculating triple exclusive OR operations. Its clear, event-driven interface and deterministic functionality make it ideal for use in control applications that require this specific Boolean logic. As an instance of a generic building block, it also benefits from the maintenance and consistency advantages of a generic design.

🌐 Related topic subpages on ms-muc-docs.de

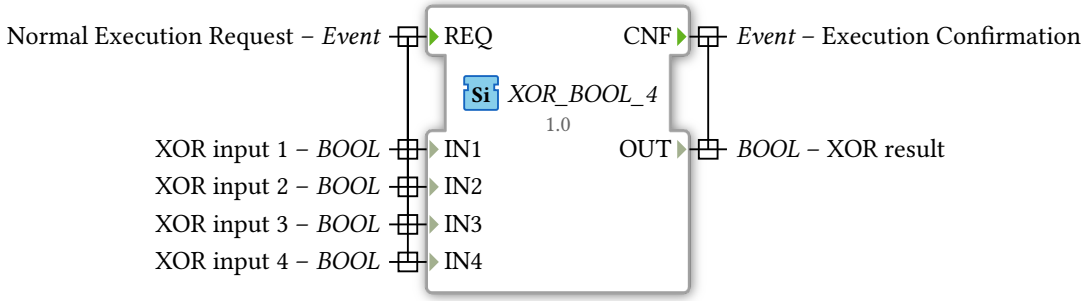
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

XOR_BOOL_4

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `XOR_BOOL_4` is a standard block for calculating the logical exclusive OR (XOR) operation for up to four Boolean input values. It complies with the IEC 61131-3 standard and is implemented as a generic block used in control applications for combinational logic processing.



Interface Structure

Event Inputs

- **REQ** (Normal Execution Request): This event triggers the calculation of the XOR operation. It is linked to all four data inputs (`IN1` to `IN4`).

Event Outputs

- **CNF** (Execution Confirmation): This event signals the completion of the calculation. It is output along with the calculated result at data output `OUT`.

Data Inputs

- **IN1** (BOOL): XOR input 1.
- **IN2** (BOOL): XOR input 2.
- **IN3** (BOOL): XOR input 3.
- **IN4** (BOOL): XOR input 4.

Data Outputs

- **OUT** (BOOL): Result of the XOR operation of all active inputs.

Adapters

This function block does not have any adapter interfaces.

Functionality

When the event `REQ` occurs, the function block reads the values of all four Boolean inputs (`IN1`, `IN2`, `IN3`, `IN4`). It then calculates the logical exclusive OR over these values. The result `OUT` is `TRUE` (1) if and only

if an odd number of the inputs have the value `TRUE`. If the number of `TRUE` inputs is even (0, 2, or 4), the result is `FALSE` (0). After the calculation, the confirmation event `CNF` is output along with the result value.

Technical Features

- **Generic Block:** The block is marked with the attribute `eclipse4diac::core::GenericClassName` as `'GEN_XOR'`. This allows for flexible instantiation and reuse within the 4diac IDE.
- **Event-Driven Execution:** The functionality is strictly event-driven. A calculation only takes place upon an incoming `REQ` event.
- **Four Inputs:** Unlike a simple 2-input XOR, this block offers direct processing of up to four signals without additional nesting.

State Overview

The function block has no internal state (memory-free, combinatorial). Its behavior depends solely on the current values at the data inputs during each call.

Application Scenarios

- **Parity Check:** Detects whether an odd number of conditions (e.g., error messages, limit exceedances) are active in a group of four signals.
- **Control Logic:** Implements safety circuits where exactly one of several possible input combinations is allowed to trigger an action.
- **Nested Logic Simplification:** Replaces multiple cascaded 2-input XOR blocks with a single `XOR_BOOL_4` block for improved clarity in the function block diagram (FBD).

⚖️ **Comparison with Similar Blocks**

- **Standard XOR (2 Inputs):** A classic XOR block with only two inputs. The `XOR_BOOL_4` consolidates the functionality of several such blocks. See: XOR_4
- **OR_BOOL_4 / AND_BOOL_4:** These blocks calculate the logical OR or AND operation across four inputs. The XOR operation differs fundamentally, as it evaluates the case where all inputs are `TRUE` as `FALSE`.
- **Parity Check Blocks:** Specialized blocks for parity calculation often exist for data words (BYTE, WORD). `XOR_BOOL_4` provides this function for individual bits/Bool values.

Conclusion

The `XOR_BOOL_4` is a useful and efficient standard block for implementing exclusive OR logic with up to four inputs in 4diac-based control applications. Its event-driven, stateless architecture and direct support for multiple inputs make it a practical choice for parity checks and specialized combinatorial control tasks.

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

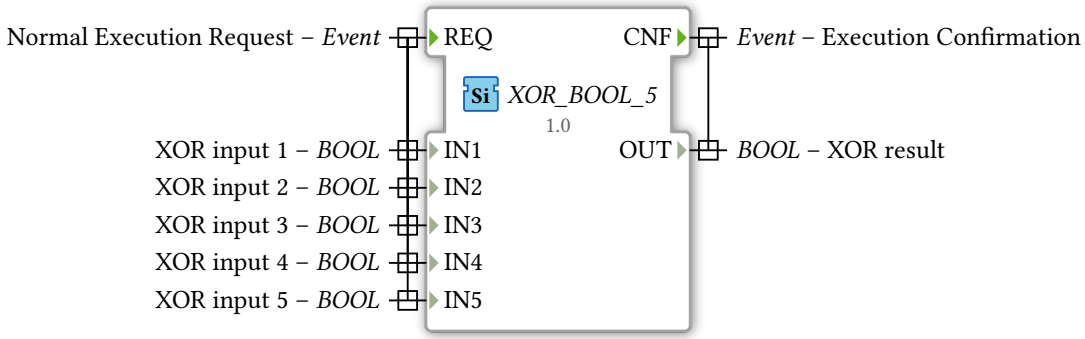
]

XOR_BOOL_5

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `XOR_BOOL_5` is a standard function block for calculating the logical exclusive OR (XOR) operation for five Boolean input values. It complies with the IEC 61131-3 standard and is implemented as a generic function block. The operation is performed on each incoming execution event, and the result is output via an acknowledgment event.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** Triggers the calculation of the XOR operation. This event is associated with all five data inputs (`IN1` to `IN5`).

Event Outputs

- **CNF (Execution Confirmation):** Signals the completion of the calculation. This event is linked to the data output `OUT` and returns the result.

Data Inputs

- **IN1 (XOR input 1):** First Boolean operand (BOOL).
- **IN2 (XOR input 2):** Second Boolean operand (BOOL).
- **IN3 (XOR input 3):** Third Boolean operand (BOOL).
- **IN4 (XOR input 4):** Fourth Boolean operand (BOOL).
- **IN5 (XOR input 5):** Fifth Boolean operand (BOOL).

Data Outputs

- **OUT (XOR result):** The Boolean result of the XOR operation of all five inputs (BOOL).

Adapter

This function block has no adapter interfaces.

Functionality

Whenever the event `REQ` occurs, the function block performs the logical XOR operation on its five Boolean inputs `IN1` to `IN5`. The result is output `OUT`, and the acknowledgment event `CNF` is triggered simultaneously.

The logical function is defined as: `OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5`.

The result is `TRUE` (1) if and only if an odd number of the inputs have the value `TRUE` (1). With an even number of `TRUE` inputs, the result is `FALSE` (0).

Technical Features

- **Generic Block:** The block is marked as a generic implementation (`'GEN_XOR'`) by the attribute `eclipse4diac::core::GenericClassName`. This allows for flexible reuse of the core logic.
- **Event-driven:** The calculation is performed exclusively event-driven by `REQ` and is non-cyclic.
- **Five Inputs:** Unlike standard XOR blocks with typically two inputs, this block offers five hard-wired inputs, eliminating the need to chain multiple blocks for a higher number of operands.

State Overview

The block has no internal state (memory-free). Its output depends solely on the current values of the inputs at the time of the `REQ` event.

Application Scenarios

- **Parity Check:** Checks whether an odd number of conditions (e.g., error signals, limit exceedances) are active in a group of five signals.
- **Control Logic:** Implements special logic logic in PLC programs where exactly one, three, or five of five possible individual conditions must be met.

- **Signal Comparison:** Detects inequality in a group of five binary signals.

⚖️ Comparison with Similar Blocks

- **Standard XOR (2 Inputs):** Library blocks often only offer the XOR operation for two operands. For five operands, several of these blocks would have to be chained together (e.g., `((IN1 XOR IN2) XOR (IN3 XOR IN4)) XOR IN5`). `XOR_BOOL_5` summarizes this logic in a single, concise block. See: XOR_5
- **OR / AND Blocks:** While OR (`OR`) returns `TRUE` with at least one `TRUE` input, and AND (`AND`) only with all `TRUE` inputs, XOR specifically responds to an odd number of `TRUE` values.
- **Generic XOR Blocks:** Generic blocks could exist that handle a variable number of inputs via arrays or repeated inputs. `XOR_BOOL_5` is a fixed, specific instance for exactly five inputs.

Conclusion

The `XOR_BOOL_5` function block is a specialized and efficient building block for calculating the exclusive OR operation across exactly five Boolean signals. Its event-driven, stateless operation and clear interface make it a reliable tool for control applications that require this specific logic function, especially in the areas of parity checking and complex conditional logic.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

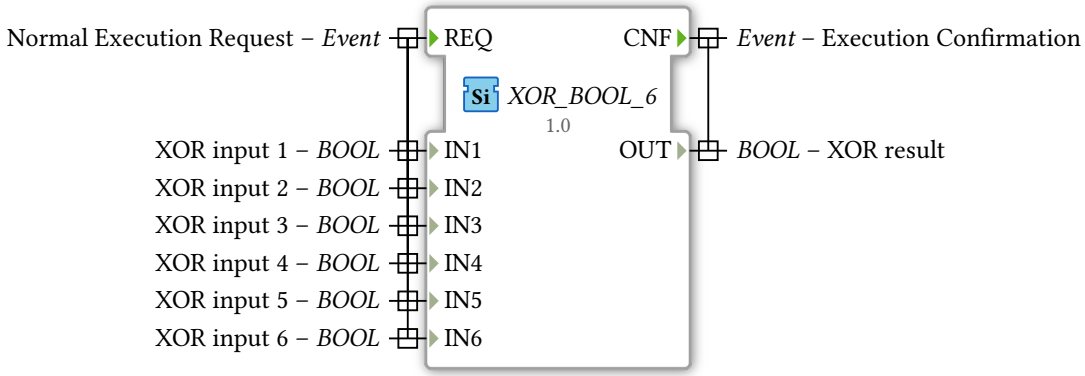
]

XOR_BOOL_6

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `XOR_BOOL_6` is a generic block for calculating the logical exclusive OR (XOR) operation for up to six Boolean input values. It complies with the IEC 61131-3 standard and is designed for use in control and automation applications. The block waits for a trigger event, calculates the result of the XOR operation across all active inputs, and outputs it along with an acknowledgment event.



Interface Structure

The function block has an event-driven input/output mechanism with six Boolean data inputs and one Boolean data output.

Event Inputs

REQ (Normal Execution Request): This event triggers the calculation. Upon arrival of `REQ`, the values at inputs `IN1` to `IN6` are read, and the XOR operation is performed.

Event Outputs

- **CNF** (Execution Confirmation): This event is triggered after the calculation is complete and signals that the result is available at output `OUT`.

Data Inputs

- **IN1** (BOOL): XOR input 1.

- **IN2** (BOOL): XOR input 2.
- **IN3** (BOOL): XOR input 3.
- **IN4** (BOOL): XOR input 4.
- **IN5** (BOOL): XOR input 5.
- **IN6** (BOOL): XOR input 6.

Data Outputs

- **OUT** (BOOL): The result of the XOR operation of all six inputs.

Adapters

This function block does not have any adapter interfaces.

Functionality

When triggered by the `REQ` event, the function block calculates the logical XOR value across all six inputs `IN1` to `IN6`. The XOR operation yields `TRUE` (1) if and only if an odd number of inputs have the value `TRUE`. If there are zero, two, four, or six inputs `TRUE`, the result is `FALSE` (0). Formally, this corresponds to the calculation: `OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5 XOR IN6`. After the calculation, the result is output at data output `OUT`, and the confirmation event `CNF` is simultaneously activated.

Technical Features

- **Generic Block:** The function block is marked as a generic block (`GEN_XOR`). This allows it to be reused in different contexts or used as a basis for specialized versions.
- **Fixed Number of Inputs:** Unlike blocks with a variable number of inputs, `XOR_BOOL_6` has exactly six fixed inputs. Unused inputs should be set to a defined logic level (e.g., `FALSE`).
- **Package Assignment:** The block is assigned to the package `iec61131::bitwiseOperators`, which underlines its classification as a bitwise/logical operation.
- **Fixed Number of Inputs:**

State Overview

The function block is stateless (combinatory). It has no internal memory. The output `OUT` is a pure function of the current input values at the time of the `REQ` event. There is no dependency on previous calculations or states.

Application Scenarios

- **Parity Check:** Detects whether an odd number of conditions are active in a group of six signals.
- **Safety-Critical Logic:** In circuits where a state may only be active if exclusive conditions are met (e.g., selection between different, mutually exclusive operating modes).
- **Status Bit Combination:** Combines multiple status or error flags into a single "odd parity" signal.

⚖️ Comparison with Similar Blocks

- **`XOR` (2 inputs):** The standard XOR block with only two inputs. `XOR_BOOL_6` extends this functionality to six inputs without the need to connect multiple 2-input XOR blocks. See: XOR_6
- **`AND_BOOL_6` / `OR_BOOL_6`:** Similar blocks for the logical AND and OR operations with six inputs. While these blocks test for "All" or "At least one," `XOR_BOOL_6` tests for "Odd number."
- **Blocks with Variable Number of Inputs:** Some implementations may offer a `XOR` block to which any number of inputs can be assigned. In contrast, `XOR_BOOL_6` offers an explicit and fixed interface.

Conclusion

The `XOR_BOOL_6` is a specialized and easy-to-use function block for exclusive OR operation on up to six Boolean signals. Its fixed interface and event-driven execution make it easily integrated into 4diac IDE-based applications. It is particularly useful in scenarios requiring parity checking or the evaluation of exclusive conditions across a small, fixed number of signals. For applications with a variable or significantly larger number of inputs, alternative solutions should be considered.

The `XOR_BOOL_6` is a specialized and easy-to-use function block for exclusive OR operation on up to six Boolean signals. —

🌐 Related topic subpages on ms-muc-docs.de

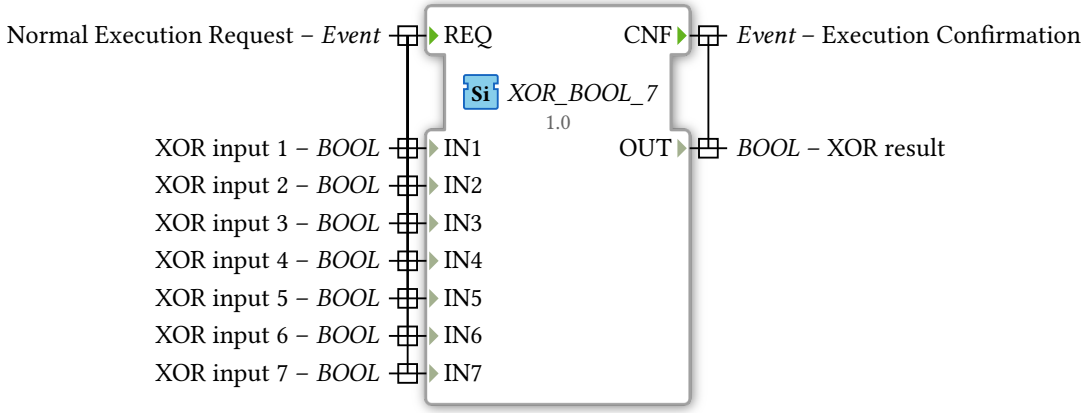
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

XOR_BOOL_7

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `XOR_BOOL_7` is a generic block for calculating the logical exclusive OR (XOR) operation. It processes seven Boolean input signals and returns the result of the XOR operation as a Boolean output signal. The block complies with the IEC 61131-3 standard and is designed for use in control applications where odd parity checking or logical antivalence across multiple signals is required.



Interface Structure

Event Inputs

- **REQ (Normal Execution Request):** This event triggers the execution of the function block. Upon its arrival, all seven data inputs (`IN1` to `IN7`) are read and the XOR calculation is performed.

Event Outputs

- **CNF (Execution Confirmation):** This event signals the completion of the calculation. It is output along with the calculated data output `OUT`.

Data Inputs

- **IN1 (XOR input 1):** Boolean input 1.
- **IN2 (XOR input 2):** Boolean input 2.
- **IN3 (XOR input 3):** Boolean input 3.
- **IN4 (XOR input 4):** Boolean input 4.
- **IN5 (XOR input 5):** Boolean input 5.
- **IN6 (XOR input 6):** Boolean input 6.
- **IN7 (XOR input 7):** Boolean input 7.

Data Outputs

- **OUT (XOR result):** Boolean result of the XOR operation of all seven inputs.

Adapters

This function block does not have any adapter interfaces.

Functionality

When the event `REQ` occurs, the block performs the logical XOR operation on all seven Boolean inputs `IN1` to `IN7`. The result `OUT` is `TRUE` (1) if and only if an odd number of the inputs have the value `TRUE`. If the number of `TRUE` inputs is even (0, 2, 4, 6), the result is `FALSE` (0).

Mathematically expressed: `OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5 XOR IN6 XOR IN7`

Immediately after the calculation, the completion event `CNF` is triggered along with the valid result `OUT`.

Technical Features

- **Generic Block:** The block is implemented as a generic function block (`GEN_XOR`), which allows for flexible reuse in different contexts.

- **Fixed Number of Inputs:** Unlike blocks with a variable number of inputs, this block has exactly seven fixed inputs.
- **Event-Driven Execution:** The operation is triggered exclusively by an incoming event (`REQ`), which enables deterministic and energy-efficient processing in the real-time system.

State Overview

The function block does not have an internal state in the sense of a memory. Its behavior is purely combinatorial and depends solely on the current values at the inputs at the time of the `REQ` event. After the output of `CNF`, the block remains in a waiting state until the next `REQ` event.

Application Scenarios

- **Parity Check:** Monitoring whether an odd number of sensors (e.g., limit switches, light barriers) are in a specific state (`TRUE`).
- **Safety-Critical Logic:** In safety circuits where an antivalence condition across multiple safety devices is required.
- **Control Logic:** Implementation of special logic operations in sequence controls or logic controllers that require an XOR function with exactly seven operands.

⚖️ Comparison with similar building blocks

- **`XOR` (2 inputs):** The standard XOR block only combines two inputs. To combine seven signals, this block would have to be nested multiple times, which reduces program readability. `XOR_BOOL_7` offers a direct and clear solution. See: XOR_7
- **`GEN_XOR` (Generic, n inputs):** The underlying generic block `GEN_XOR` can be instantiated with a variable number of inputs. `XOR_BOOL_7` is a specific, pre-configured instance with seven inputs that can be used directly without having to parameterize the number of inputs.
- **`PARITY_CHECK`:** A pure parity check block could deliver similar results, but is often conceptually designed for error detection in data streams, whereas `XOR_BOOL_7` serves as a general-purpose logical operator in control logic.

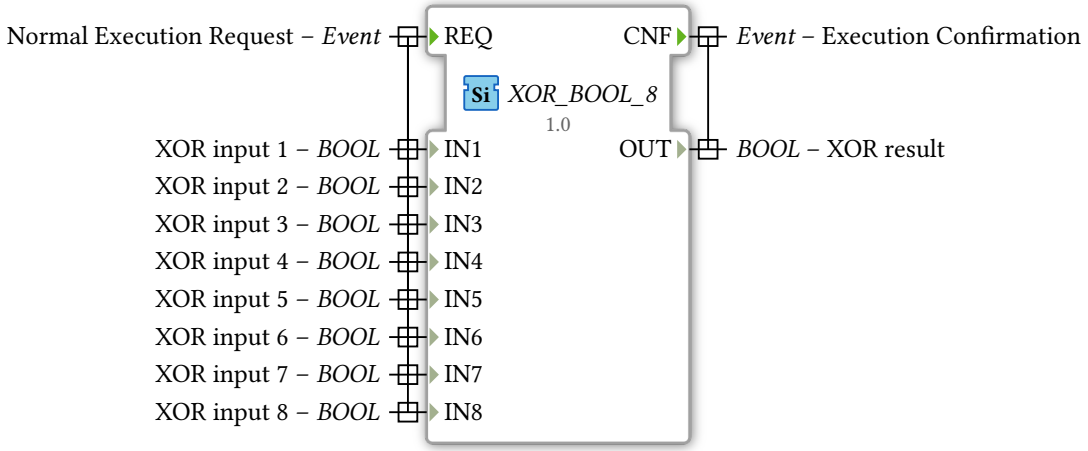
Conclusion

The `XOR_BOOL_7` function block is a specialized and efficient building block for the logical anti-valence operation of seven Boolean signals. Its event-driven, stateless operation and clear interface make it ideal for use in structured IEC 61131-3-based control programs where such a fixed operation is required. It simplifies programming compared to nesting multiple two-input XOR blocks and provides a ready-to-use, reliable component.

XOR_BOOL_8

• – ◦ • – ◦ • – ◦ •

The function block `XOR_BOOL_8` is a generic block for calculating the logical exclusive OR (XOR) operation on up to eight Boolean input values. It complies with the IEC 61131-3 standard and is designed for use in the 4diac IDE. The block executes its operation in an event-driven manner and returns the result via a single Boolean output.



- **REQ** (Normal Execution Request): This event triggers the calculation. It is associated with all eight data inputs (`IN1` to `IN8`).
- **CNF** (Execution Confirmation): This event signals the completion of the calculation. It is linked to the data output `OUT`.
- **IN1** (BOOL): XOR input 1.
- **IN2** (BOOL): XOR input 2.
- **IN3** (BOOL): XOR input 3.
- **IN4** (BOOL): XOR input 4.
- **IN5** (BOOL): XOR input 5.
- **IN6** (BOOL): XOR input 6.
- **IN7** (BOOL): XOR input 7.
- **IN8** (BOOL): XOR input 8.
- **OUT** (BOOL): The result of the XOR operation of all active inputs.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use adapters.

Functionality

Upon the occurrence of the event `REQ`, the function block reads the values of all eight Boolean inputs (`IN1` to `IN8`). It then calculates the logical exclusive OR operation on all these input values. The result of this calculation is output at the data output `OUT`, and simultaneously, the confirmation event `CNF` is set to signal the completion of the operation.

The logical function can be described as follows:

OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5 XOR IN6 XOR IN7 XOR IN8`

An XOR (exclusive OR) operation returns `TRUE` (1) as the result if and only if there is an odd number of inputs connected, and `TRUE` (1) is the result. If there is an even number of inputs, the result is `FALSE` (0).

- **Generic Block:** The block is marked as a generic block (`GEN_XOR`), indicating its reusability and possible parameterization for other data types. In this specific instance, however, it is fixed to the data type `BOOL`.
- **Generic Block:** The block is marked as a generic block (`GEN_XOR`), which indicates its reusability and possible parameterization for other data types. In this specific instance, however, it is fixed to the data type `BOOL`.
- **Event-driven execution:** The operation is only performed upon an incoming `REQ` event, enabling efficient and deterministic control.
- **Fixed number of inputs:** The block provides exactly eight inputs. Unused inputs should be set to a defined logical value (e.g., `FALSE`).

The function block does not have an internal state in the sense of a memory. Its behavior is purely combinatorial and event-driven:

1. **Idle state:** Waits for the arrival of the `REQ` event.
2. **Computation state:** Upon `REQ`, all inputs are read and the XOR operation is performed.
3. **Output State:** The result is output to `OUT`, and the `CNF` event is generated. The block then returns to its idle state.

- **Parity Check:** Detects whether an odd number of signals (e.g., error messages, limit exceedances) are active in a group.
- **Toggle Logic:** Control where an action is only executed if exactly one of several possible control signals is active (e.g., selection between manual and automatic mode).

- **Basic Encryption and Coding Functions:** As a basic building block for simple cryptographic operations or error detection codes.
- **Comparison of Two or More States:** Determines whether at least one signal in a group differs from its expected state.
- **`OR_BOOL_8` / `AND_BOOL_8`:** These blocks calculate the logical OR and AND operations, respectively. Unlike XOR, OR (`TRUE`) returns true if *at least* one input is true, and AND (`TRUE`) returns true only if *all* inputs are true. XOR is more specific and responds to the *parity* (even/odd) of the true inputs.
- **`XOR` (with 2 inputs):** A standard XOR block with only two inputs. `XOR_BOOL_8` extends this functionality to up to eight inputs without the need to chain multiple blocks together. See: XOR_8
- **Combinatorial Logic in SFC/ST:** The function could also be programmed directly in Structured Text (ST) (e.g., `OUT := IN1 XOR IN2 XOR ...;`). The advantage of this function block lies in its standardized, reusable, and graphically representable interface within FBD networks.

The `XOR_BOOL_8` is a specialized, reliable, and easy-to-use function block for logical XOR operations with up to eight inputs. Its event-driven architecture makes it ideal for integration into control-typical flowcharts in the 4diac IDE. It is particularly suitable for applications where the parity of signal groups is important and offers a clearly defined alternative to basic OR and AND operations.

[Following the previous sentence about the function block, the `XOR_BOOL_8` is not relevant to the context and should be omitted.] —

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

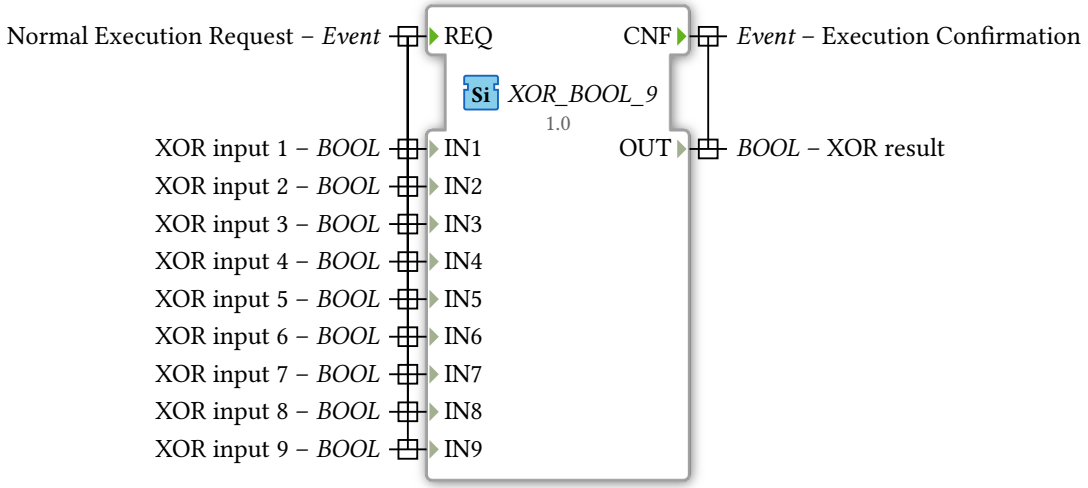
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

XOR_BOOL_9

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `XOR_BOOL_9` is a generic block for calculating the logical exclusive OR (XOR) operation for up to nine Boolean input signals. It implements an n-ary XOR function where the output is `TRUE` if and only if there is an odd number of active inputs, such as `TRUE`. The block complies with the IEC 61131-3 standard and is designed for use in control applications that require checking for odd parity or specific selection or monitoring logic.



Interface Structure

The function block has a simple event-driven interface with a request event and an acknowledge event.

Event Inputs

- **REQ (Normal Execution Request):** This event triggers the calculation of the XOR function. It is associated with all nine data inputs (`IN1` to `IN9`).

Event Outputs

- **CNF (Execution Confirmation):** This event signals the completion of the calculation. It is output along with the calculated result at data output `OUT`.

Data Inputs

- **IN1 (XOR input 1):** Boolean input 1.
- **IN2 (XOR input 2):** Boolean input 2.
- **IN3 (XOR input 3):** Boolean input 3.
- **IN4 (XOR input 4):** Boolean input 4.
- **IN5 (XOR input 5):** Boolean input 5.
- **IN6 (XOR input 6):** Boolean input 6.
- **IN7 (XOR input 7):** Boolean input 7.
- **IN8 (XOR input 8):** Boolean input 8.
- **IN9 (XOR input 9):** Boolean input 9.

Data Outputs

- **OUT (XOR result):** Boolean result of the n-way XOR operation of all active inputs.

Adapters

This function block does not use adapters.

Functionality

When the event `REQ` arrives, the logical operation is executed. The output `OUT` is set to `TRUE` if the number of inputs with the value `TRUE` is odd. If the number of `TRUE` inputs is even (or zero), `OUT` is set to `FALSE`. Immediately after the calculation, the acknowledgment event `CNF` is output along with the current value of `OUT`.

The mathematical description is: `OUT = IN1 XOR IN2 XOR IN3 XOR IN4 XOR IN5 XOR IN6 XOR IN7 XOR IN8 XOR IN9`.

Technical Features

- **Generic Function Block:** The function block is designated as a generic function (`GEN_XOR`), which means it can serve as the basis for deriving specific XOR function blocks with a fixed number of inputs.
- **Hard-wired Logic:** The operation is performed across all nine inputs. For applications with fewer inputs, the unused inputs must be set to a defined value (typically `FALSE`).
- **Event-driven Execution:** The calculation only occurs when the `REQ` event occurs, enabling resource-efficient and deterministic processing.

State Overview

The function block does not have an internal state in the sense of a memory. Its behavior is purely combinatorial and depends solely on the current values at the data inputs at the time of the `REQ` event. The "calculation in progress" state is transient and ends immediately with the output of `CNF`.

Application Scenarios

- **Parity Check:** Monitoring whether an odd number of sensors (e.g., limit monitors, safety switches) report an alarm condition.
- **Selection or Toggle Logic:** Control where an action is executed only when the state of an odd number of conditions has changed.
- **Error Detection in Redundant Systems:** Simple plausibility check in systems with multiple redundant channels.
- **Encryption and Coding Methods:** As a fundamental component in simple cryptographic or error-correcting codes.
-

⚖️ Comparison with Similar Blocks

- **Standard XOR blocks (e.g., XOR, E_XOR):** These typically have only two inputs. `XOR_BOOL_9` extends this functionality to up to nine inputs in a single block.
- **OR blocks (OR) / AND blocks (AND):** Return a `TRUE` if at least one or all inputs are `TRUE`. The XOR logic is more specific (odd number).
- **Parity blocks:** Specialized blocks for parity calculation, often working directly with data words (BYTE, WORD). `XOR_BOOL_9` works with individual Boolean bits and is more flexible in terms of the number of inputs.
- – **Combinatorial Logic Blocks (GEN_AND, GEN_OR):** Similar generic blocks for other basic logical operations.

Conclusion

The `XOR_BOOL_9` is a specialized, generic function block that provides the logical XOR operation for up to nine signals in a single, event-driven component. Its strength lies in its clear interface and direct implementation of the n-ary XOR function, making it ideal for parity checks and specialized control logic. For applications with a variable or very large number of inputs, the logic may need to be built from multiple basic components.

2.10.1.2 Numeric Index (iec61131-3-bool)

Here you will find all exercises for the `iec61131-3-bool` workspace in ascending order.

F_MUX_32



• – ○ • – ○ • – ○ •

Introduction

The function block **F_MUX_32** is a generic multiplexer (selection function) according to IEC 61131-3. It allows the selection of one of 32 input values (IN1 ... IN32) and outputs it at output OUT. The selection is made using the integer selector K. The block is event-driven: When an event occurs at the REQ input, the current value of K is evaluated and the corresponding input value is applied.

Interface Structure

Event Inputs

Name	Type	Comment
REQ	Event	Service Request – triggers the selection

Event Outputs

Name	Type	Comment
CNF	Event	Confirmation – confirms the selection has been made

Data Inputs

Name	Type	Comment
K	ANY_INT	Selection value (0 ... 31), selects one of the 32 inputs

IN1	ANY	Input value 1
IN2	ANY	Input value 2
…	…	…
IN32	ANY	Input value 32

Data Outputs

Name	Type	Comment
OUT	ANY	Output value: IN1 if K=0; IN2 if K=1; …; IN32 if K=31

Adapter

No adapters available.

Functionality

The FB `F_MUX_32` operates in an event-driven manner:

1. An event at the **REQ** input triggers processing.
2. The selector **K** (integer value) is evaluated.
3. Depending on the value of K, the corresponding data input is selected:

- K = 0 ⟶ **IN1**
- K = 1 ⟶ **IN2**
- …
- K = 31 ⟶ **IN32**

1. The selected value is passed to the **OUT** output.
2. An event at the **CNF** output signals the completion of the operation.

Values outside the valid range (0 … 31) result in undefined behavior; the function block does not offer range checking.

Technical Features

- **Generic Data Type (ANY):** All data inputs and the output are declared with `ANY`. The function block can therefore be used with various data types (BOOL, INT, REAL, etc.). However, all inputs and the output must have the same specific type.
- **Event Association:** The **REQ** event is associated with all data inputs (IN1 … IN32 and K). When a REQ event occurs, all these inputs are read simultaneously.
- **Lack of Range Check:** The function block does not check whether K is in the range 0 … 31. An invalid value can lead to unexpected results (e.g., selecting a non-existent channel). This must be addressed in the application.
- **Standard Compliance:** The function block implements the `SELECT` functionality from IEC 61131-3 and is classified as a “standard selection function”.

State Overview

The function block does not have an explicit state machine in its XML definition. It behaves like an event-driven function: After the **REQ** event, the output is immediately updated and **CNF** is triggered. There are no internal states.

Application Scenarios

- **Switching between multiple sensor values** in a controller (e.g., temperature, pressure, or level sensors).
- **Configurable Configuration:** Selection of different operating modes or setpoints via an index.
- **Test and diagnostic functions** where different signal sources are connected to an analysis point.
- **Extension of existing multiplexer solutions** with fewer channels to 32 inputs.

Comparison with Similar Function Blocks

- **F_MUX_2, F_MUX_3, ... F_MUX_n:** These function blocks have a smaller number of inputs (2, 3, ...). `F_MUX_32` covers the maximum requirement of 32 channels. If only a smaller number is needed, smaller variants are more space-saving and easier to understand.
- **CASE Statement (ST):** A `CASE` structure in Structured Text can represent the same behavior but is not encapsulated as a reusable function block. The function block offers event-driven encapsulation for graphical and textual programming.
- **MUX Function in IEC 61131-3:** The integrated `MUX` function is typically limited to smaller numbers (e.g., 8). The function block extends this to 32 channels.

Conclusion

The **F_MUX_32** is a powerful and flexible multiplexer IC for use in automation technology. Its generic type and large number of 32 inputs make it suitable for a wide variety of selection tasks. The simple event control and the clean separation of event and data paths make it easily integrated into IEC 61131-3 environments. However, users should ensure the valid selection range is used to avoid errors.

2.10.2 isobus

2.10.2.1 🎧 Podcast

- The Working Set Object: Understanding the Brain of the ISOBUS User Interface – From Standard to Practice in the ISO Designer
- ISOBUS Button: More Than Just a Click – Standardizing Agricultural Machinery
- ISOBUS Object Pointer: The Secret of Dynamic Displays and Efficient Fault Diagnosis
- ISOBUS Output Meter: Mastering Dynamic Displays – From Pointer to Visualization on the VT
- ISOBUS Scaling: When the Tractor Screen Doesn't Fit – An Introduction to ISO 11783-6

2.10.2.2 TC

🎧 Podcast

- E_SWITCH: The Switch of Automation – Why Simplicity Revolutionizes IEC 61499
- E_SWITCH: The Unsung Hero of Industrial Automation's Modular Design
- Unpacking E_T_FF_SR: The Secret Toggle Switch of Industrial Control Systems
- ESP32-S3 Development Boards ESP32-S3-DevKitC-1
- ESP32-S3-DevKitC-1 Documentation Analysis: The Memory Monster (32MB Flash/16MB PSRAM) and the Power of Dual USB Ports

📺 Video

- The ESP32-S3-DevKitC-1

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

]

🔌 io

📺 Video

- Automation of the Future
- The Great Migration
- Digital Decisions
- Ideal Diodes
- The Dual Wi-Fi Solution

🎧 Podcast

- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Target Groups, Structure, and Unique Selling Proposition

- Taming the Software Dragon: Industrial Automation and the Future of Production
- Eclipse 4diac: Innovation Engine Research vs. User Needs – What Really Drives Development?
- Taming the Software Dragon: Why IEC 61499 Could Revolutionize Industry

---

🌐 Related Topic Subpages on ms-muc-docs.de

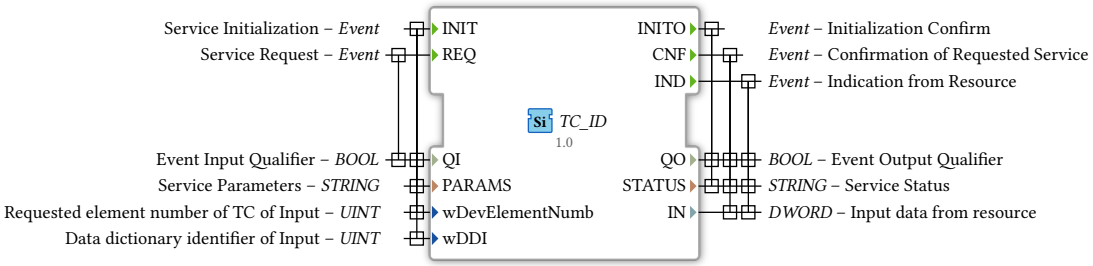
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Diode & Semiconductor Basics on ms-muc-docs.de
- 🌐 Smart Switch vs. Relay on ms-muc-docs.de

TC_ID

• – ◦ • – ◦ • – ◦ •

Introduction

The TC_ID function block is an input service interface module for double-word input data. It serves as an interface for communication with tractor control (TC) input devices and enables access to specific input data via defined data words.



Interface Structure

Event Inputs

- **INIT**: Service Initialization
- Linked to: QI, PARAMS, wDevElementNumb, wDDI
- **REQ**: Service Request
- Linked to: QI

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Service Request
- Linked to: QO, STATUS, IN
- **IND**: Resource Indication
- Linked to: QO, STATUS, IN

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **wDevElementNumb** (UINT): Requested Element Number TC Input
- **wDDI** (UINT): Data Dictionary Identifier of the input

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status
- **IN** (DWORD): Input data from the resource

Adapter

No adapter interfaces are available.

Functionality

The TC_ID block enables access to TC input data via standardized service interfaces. During initialization (INIT), the device parameters are configured. Service requests (REQ) trigger data acquisition, with the results being returned via CNF or IND events. The block supports both requested and spontaneous data deliveries from the connected resource.

Technical Features

- Processes double-word data (DWORD, 32-bit)
- Supports Data Dictionary Identifiers (DDI) for standardized data access
- Provides both requested (CNF) and spontaneous (IND) data deliveries
- Includes comprehensive status feedback via STRING parameters

State Overview

The block goes through typical service interface states:

1. **Not Initialized**: Before an INIT event
2. **Initialized**: After successful INITO feedback
3. **Ready**: For REQ requests
4. **Active**: During data querying and processing

Application Scenarios

- Agricultural control systems (ISOBUS environment)
- Tractor control and monitoring
- Data acquisition from TC input devices
- Automated agricultural systems with standardized data formats

⚖️ Comparison with Similar Blocks

Compared to simple input blocks, TC_ID offers:

- Standardized ISOBUS Interface
- DDI-based data identification
- Comprehensive status and error management
- Both requested and spontaneous data transmission

🛠️ Related Exercises

- Exercise_060

Conclusion

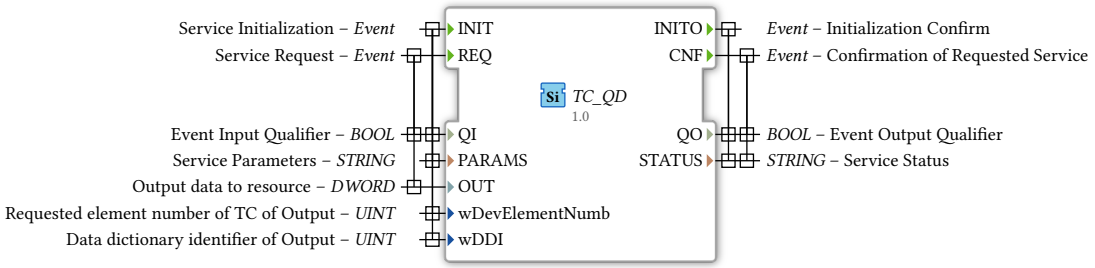
The TC_ID function block provides a robust and standardized interface for TC input data in ISOBUS environments. Its flexible architecture, with both requested and spontaneous data transmission mechanisms, makes it ideal for complex agricultural applications that require reliable access to specific input data.

TC_QD

• – ○ • – ○ • – ○ •

Introduction

The TC_QD function block is an output service interface function block for double-word output data. It serves as an interface for communication with external resources and enables the output of 32-bit data values via specified device elements.



Interface Structure

Event Inputs
- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs
- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs
- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **OUT** (DWORD): Output Data for the Resource
- **wDevElementNumb** (UINT): Requested TC Element Number for Output
- **wDDI** (UINT): Output Data Dictionary Identifier

Data Outputs
- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status

Adapter
No adapter interfaces are available.

Functionality
The TC_QD function block initializes and manages communication with external output devices. Upon activating the INIT event, the service is initialized, configuring the parameters PARAMS, wDevElement-Numb, and wDDI. The REQ event triggers the actual data output, sending the double-word value provided in the OUT input to the specified resource.

Technical Features
- Processes 32-bit data values (DWORD)
- Supports Data Dictionary Identifiers (DDI) for standardized data referencing
- Provides device element number specification for precise device control
- Delivers detailed status information via the STRING output STATUS

State Overview
The function block has two main operating states:

1. **Initialization State**: Activated by the INIT event and confirmed by INITO
2. **Service Operating State**: Activated by the REQ event and completed by CNF

Application Scenarios
- Control of ISOBUS-compatible agricultural machinery
- Output of control commands to industrial actuators
- Communication with TC (Tractor Controller) output devices
- Applications in agricultural technology and mobile machinery

⚖️ Comparison with Similar Function Blocks
Compared to simpler output blocks, TC_QD offers enhanced functionality through:

- DDI-based data identification
- Configurable device element numbers
- ISOBUS specification compatibility
- Comprehensive status feedback

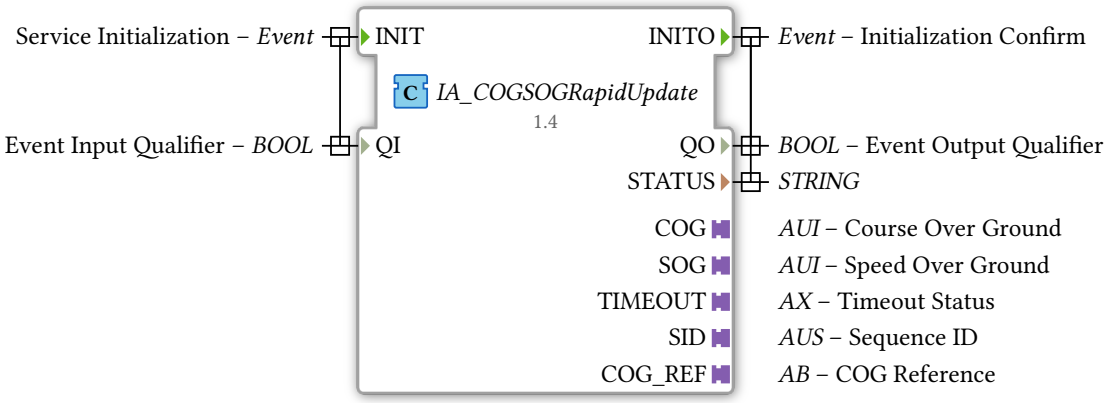
🛠️ Related exercises
- Exercise_060

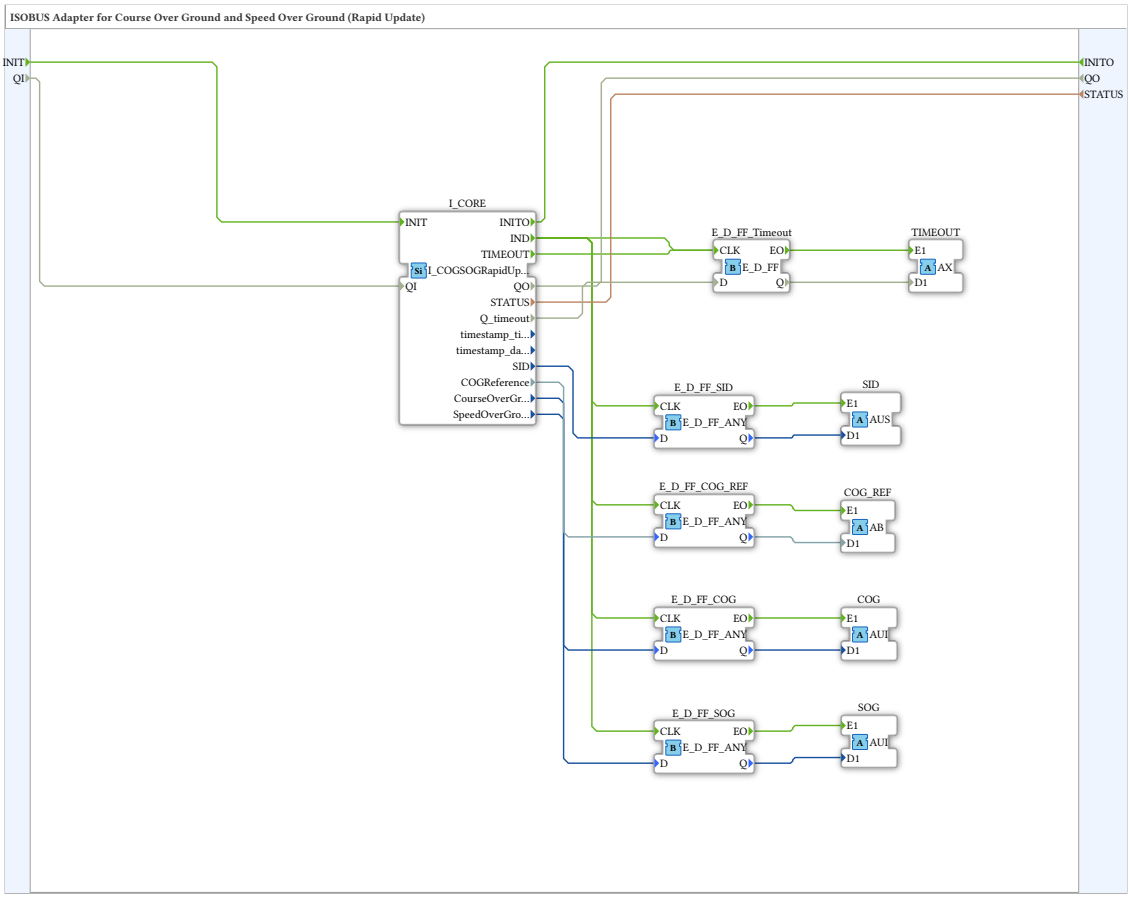
Conclusion

The TC_QD function block represents a powerful and flexible solution for outputting double-word data in ISOBUS environments. Its structured interface and extensive parameterization options make it particularly suitable for demanding control applications in mobile automation.

2.10.2.3 TECU

IA_COGSOGRapidUpdate





• – ○ • – ○ • – ○ •

Introduction

The function block **IA_COGSOGRapidUpdate** serves as an ISOBUS adapter for the NMEA 2000 PGN 129026 messages “Course Over Ground (COG)” and “Speed Over Ground (SOG)” with support for rapid updates. It decouples the ISOBUS communication and provides the received navigation data and a timeout

status via standardized adapter interfaces. The block was developed under the Eclipse Public License 2.0 and is optimized for use in agricultural control systems (TECU).

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Service initialization, triggered by input QI

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization acknowledgment, returns QO and STATUS

Data Inputs

Variable	Type	Comment
QI	BOOL	Input qualifier for initialization

Data Outputs

Variable	Type	Comment
QO	BOOL	Output qualifier (initialization status)
STATUS	STRING	Status message (e.g., error text or “OK”)

Adapter

Adapter Name	Type	Comment
COG	adapter::types::unidirectional::AUI	Course Over Ground
SOG	adapter::types::unidirectional::AUI	Speed Over Ground
TIMEOUT	adapter::types::unidirectional::AX	Timeout status (active when no update occurs)
SID	adapter::types::unidirectional::AUS	Sequence ID (sequential number of the data record)
COG_REF	adapter::types::unidirectional::AB	Course reference (e.g., True/Magnetic)

Functionality

1. **Initialization**

An event at **INIT** activates the function block. The **QI** input controls the start. After successful initialization, **INITO** is triggered, and the **QO** (TRUE on success) and **STATUS** outputs are set.

1. **ISOBUS Message Processing**

The internal core (FB `I_CORE` of type `isobus::tecu::I_COGSOGRapidUpdate`) continuously receives NMEA 2000 PGN 129026 messages. As soon as new data is available, the following values are output via the corresponding adapters:

- **COG**: Course over ground (e.g., in 0.01° increments)
- **SOG**: Speed over ground (e.g., in 0.01 km/h)
- **SID**: Sequence ID for synchronization
- **COG_REF**: Reference (True or Magnetic)

1. **Timeout Monitoring**

If no valid message is received within a configured time period, the function block activates the **TIMEOUT** adapter. This event can be used by the application to trigger error conditions or plausibility checks.

1. **Output Cycles**

The **COG**, **SOG**, **SID**, and **COG_REF** adapters are triggered together with each valid data reception (event connection `IND`). The **TIMEOUT** adapter is triggered independently upon timeout.

Technical Features

- **NMEA 2000 PGN 129026** – This module is specifically designed to interpret this PGN, enabling rapid updates of COG and SOG.
- **Adapters instead of direct inputs/outputs** – All navigation data is provided via unidirectional adapters (type `unidirectional::AUI`, `AX`, etc.), allowing for flexible further processing and encapsulation in higher-level networks.
- **License and Origin** – This module is available under the Eclipse Public License 2.0 and was developed by **HR Agrartechnik GmbH** (Version 1.0, April 19, 2026, Author: Franz Höpfinger).

Adapters instead of direct inputs/outputs – - **Integration in isobus::tecu** – The function block is part of the `isobus::tecu` library and is based on a low-level ISOBUS implementation.

State Overview

The function block does not have explicitly modeled states; however, the following operating phases can be derived:

State	Description
Inactive	INIT not yet triggered or QI = FALSE
Initializing	After INIT, until INITO is reported with QO
Active (Data Received)	Regular reception of PGN 129026, adapter updates
Timeout	No valid message within the timeout threshold – TIMEOUT adapter active
Error	Initialization failed (STATUS = error text)

Application Scenarios

- **Agricultural Tractors** – Providing course and speed data for automatic steering systems or application maps.
- **ISOBUS-based ECUs** – Integration into a TECU network for evaluating GPS/GNSS data.
- **Fast Control Loops** – Utilizing the rapid update function for dynamic control variables (e.g., for site-specific treatment).
- **Fault Monitoring** – Using the TIMEOUT adapter to detect data failures and activate redundancy mechanisms.

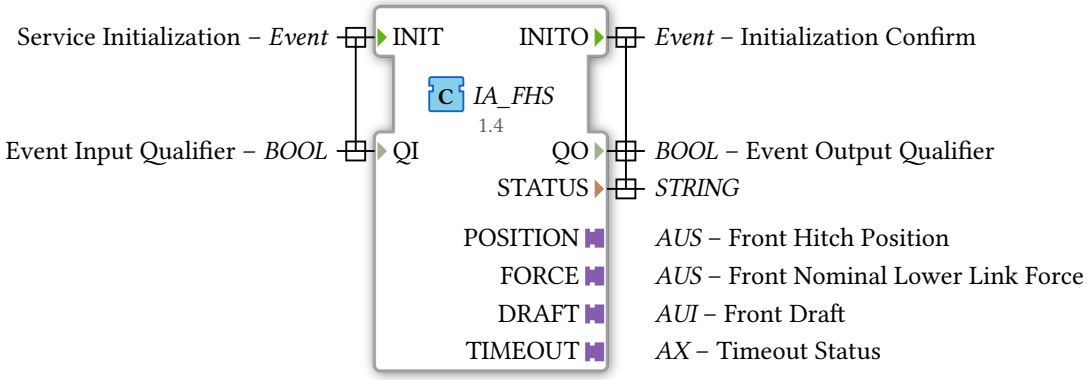
Comparison with Similar Modules

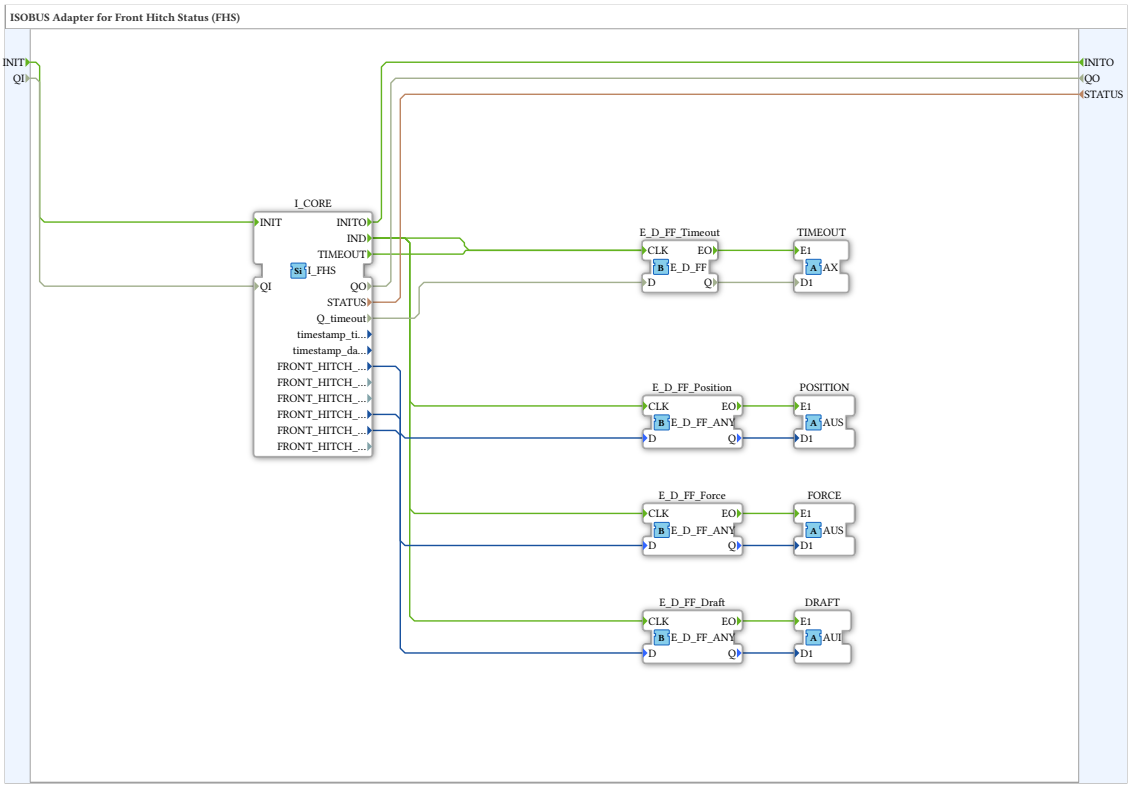
There are general ISOBUS adapters for COG/SOG (e.g., `I_COGSOG`), but these do not offer rapid update support. The **IA_COGSOGRapidUpdate** is specifically designed for PGN 129026 and additionally provides the sequence ID and timeout status. Similar modules like `I_COGSOGStd` may use a slower update rate or lack timeout detection. Using adapter interfaces (instead of a fixed data port) increases flexibility but requires corresponding adapter configuration in the parent project.

Conclusion

The **IA_COGSOGRapidUpdate** is a specialized, license-free ISOBUS adapter for high-frequency navigation data. It encapsulates the complex NMEA 2000 communication and offers standardized adapter outputs for heading, speed, sequence ID, and timeout. Its clear initialization logic and error monitoring capabilities make it ideal for real-time applications in agricultural technology. The use of adapter technology enables clean modularization and reusability in a wide variety of control architectures.

IA_FHS





• – ○ • – ○ • – ○ •

Introduction

The function block **IA_FHS** serves as an ISOBUS adapter for the Front Hitch Status (FHS) according to ISO 11783-7 (PGN 65094). It encapsulates the internal processing of the function block `I_FHS` and provides the acquired measurement and status values via standardized unidirectional adapter outputs. Initialization occurs via the event `INIT`; after successful completion, the data is provided via the adapter interfaces.

Interface Structure

Event Inputs

Event	Type	with variables	Description
`INIT`	EInit	`QI`	Service Initialization

Event Outputs

Event	Type	with Variables	Description

INITO	EInit	QO, STATUS	Initialization Acknowledgement

Data Inputs

Variable	Type	Description
QI	BOOL	Event Input Qualifier

Data Outputs

Variable	Type	Description
QO	BOOL	Output Qualifier
STATUS	STRING	Status Message (e.g., “OK”)

Adapters

Adapter	Type	Description
POSITION	adapter::types::unidirectional::AUS	Front Hitch Position (analog, unsigned)
FORCE	adapter::types::unidirectional::AUS	Front Nominal Lower Link Force (analog)
DRAFT	adapter::types::unidirectional::AUI	Front Draft (analog, integer, unsigned)
TIMEOUT	adapter::types::unidirectional::AX	ISOBUS Reception Timeout Status

Functionality

The function block internally contains an instance I_CORE of type isobus::tecu::I_FHS. Upon arrival of the initialization event INIT, the kernel is activated with the qualifier QI. After successful initialization, it triggers the event INITO, sets QO = TRUE, and outputs the status as STATUS. Simultaneously, the kernel generates the event IND, which is forwarded via internal event connections to the adapters POSITION, FORCE, and DRAFT. The corresponding data values (FRONT_HITCH_POSITION, FRONT_HITCH_NOMINAL_LOWER_LINK_FORCE, FRONT_HITCH_DRAFT) are transmitted to the adapter outputs via the appropriate data connections. Any ISOBUS reception timeout is reported via the adapter event TIMEOUT.E1 and the data value Q_timeout.

Technical Features

- **Standard Compliance**: The module meets the requirements of ISO 11783-7 for **PGN 65094** (Front Hitch Status).
- **Adapter Interfaces**: All sensor values are output via unidirectional adapters, enabling flexible and decoupled further processing in the 4diac IDE.
- **Initialization Sequence**: Initialization must be explicitly triggered via the event INIT; the function block will only function correctly after successful completion.
- **Timeout Monitoring**: The adapter TIMEOUT signals whether ISOBUS data reception occurs within the expected time. This serves to detect errors in fieldbus communication.

State Overview

The function block does not have its own visible state machine. Internal sequence control is handled by the core function block I_FHS. Broadly speaking, two phases can be distinguished:

- **Before Initialization**: The function block is inactive. The event outputs and adapters do not provide any data.
- **After Initialization (INIT successful)**:
- INITO is triggered, QO = TRUE.
- The adapters POSITION, FORCE, and DRAFT are supplied with current measured values cyclically or event-driven.
- The adapter TIMEOUT reports the current timeout status.

If an error occurs during operation (e.g., connection loss), this is signaled via the timeout adapter or the status output.

Application Scenarios

- **Tractor Control**: Acquisition of the front linkage position and the load on the lower links for controlling implements (e.g., front loader, front linkage).
- **ISOBUS Diagnostics**: Monitoring of the ISOBUS data stream for timeouts and plausibility of the force and position values.
- **Data Recording**: Provides position, force, and tractive force data for telemetry or logging applications.

Comparison with Similar Modules

- **IA_RHS** (Rear Hitch Status): Analog adapter for the rear hitch. The addressing and PGN number differ (e.g., PGN 65096). While `IA_FHS` provides front hitch data, `IA_RHS` focuses on the rear hitch.
- **Direct I_FHS Module**: Without an adapter wrapper, the user would have to integrate the core module `I_FHS` themselves and manually configure the necessary data and event connections. The adapter module simplifies handling through standardized, pre-configured interfaces.

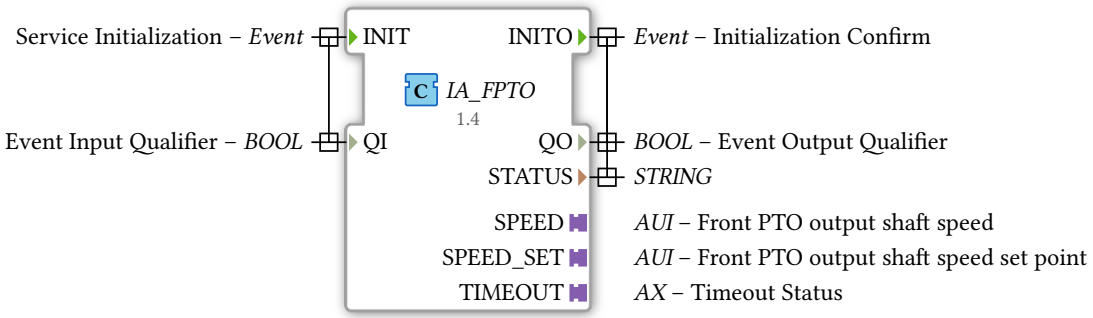
Conclusion

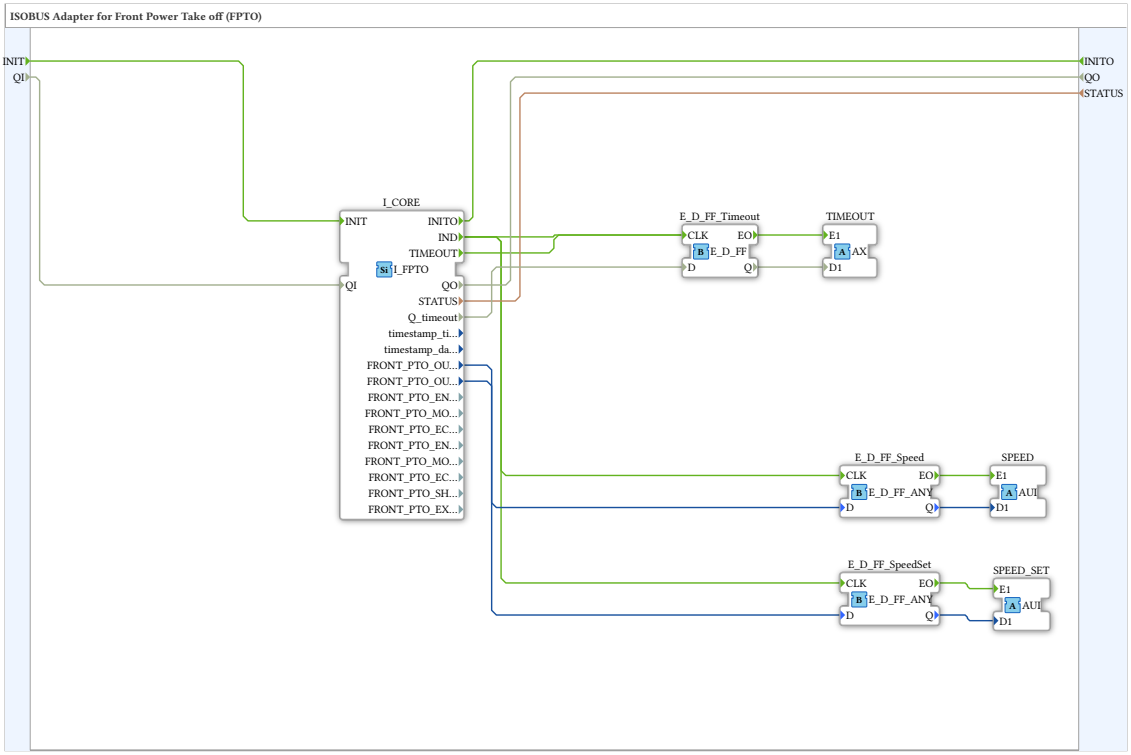
The **IA_FHS** function block offers a ready-to-use ISOBUS adapter for front hitch status. Thanks to the internal encapsulation of the `I_FHS` core and the output via unidirectional adapters, users can quickly and reliably integrate the values for position, force, and tensile force into their own control or monitoring applications. The integrated timeout monitoring increases the operational reliability of the ISOBUS communication.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

IA_FPTO





[Image of the function block, if available]

• – ○ • – ○ • – ○ •

Introduction

The **IA_FPTO** function block serves as an ISOBUS adapter for the front power take-off (FPTO) according to ISO 11783-7 (PGN 65092). It encapsulates the internal logic of the **I_FPTO** block and provides the relevant signals via standardized adapter interfaces. This enables seamless integration into ISOBUS-based control systems.

Interface Structure

Event Inputs

Event	Type	Description	With Variables
INIT	EInit	Service Initialization	QI

Event Outputs

Event	Type	Description	With Variables
INITO	EInit	Initialization Confirmation	QO, STATUS

Data Inputs

Variable	Type	Description
QI	BOOL	Qualifier for the INIT input

Data Outputs

Variable	Type	Description
QO	BOOL	Qualifier for the INITO output
STATUS	STRING	Status message after initialization

Adapter

Adapter	Type	Description
SPEED	adapter::types::unidirectional::AUI	Front PTO output speed (actual value)
SPEED_SET	adapter::types::unidirectional::AUI	Front PTO setpoint speed
TIMEOUT	adapter::types::unidirectional::AX	Communication timeout status

All adapters are unidirectional: Values are sent by the function block (FB), but not received.

Functionality

The FB is activated via the **INIT** event. If the input qualifier **QI** is set, the internal logic initializes the connection to the ISOBUS communication layer. After successful initialization, the **INITO** event is triggered, and the output variables **QO** and **STATUS** return the initialization status.

After initialization, the function block operates autonomously: It cyclically provides the current speed of the front PTO via the **SPEED** adapter and the corresponding setpoint via **SPEED_SET**. The **TIMEOUT** adapter indicates whether a communication timing error has occurred (e.g., if the control unit is not sending current data). These outputs are continuously updated as long as the initialization remains valid.

Technical Features

- The function block is based on the internal function block **I_FPTO** from the library `isobus::tecu`. The adapter interfaces decouple the application from the specific signal encoding of the ISOBUS.
- The adapters **SPEED** and **SPEED_SET** use the unidirectional type **AUI** (address unknown, integer), which is suitable for analog values such as rotational speeds.
- The adapter **TIMEOUT** uses the type **AX** (address unknown, Boolean value) to represent the timeout state in binary.
- The function block (FB) does not contain its own behavior for events outside of initialization; data updates are implicitly handled via the network connections of the block **I_FPTO**.

State Overview

The FB does not have an explicit state machine. Its behavior can be described as follows:

1. **Initialization Phase**: After the **INIT** event, the internal FB **I_FPTO** is initialized. During this phase, the adapter outputs are invalid or set to a defined initial value (0/FALSE).

The FB does not have an explicit state machine. 2. **Active Phase**: After confirmation of the initialization (**INITO**), the current values are stably available at the **SPEED**, **SPEED_SET**, and **TIMEOUT** adapters.

1. **Error State**: If a timeout occurs, the **TIMEOUT** adapter is set to TRUE. The speed values may remain at the last valid value until communication is restored.

Reinitialization (another INIT pulse) can reset and restart the FB.

Application Scenarios

- **Front PTO Control in Agricultural Tractors**: The FB receives the target and actual speeds of the front PTO via ISOBUS and forwards them to higher-level controllers or displays.
- **Integration into ISOBUS-Compliant Implements**: Implements such as front mowers or front loaders can use the speed and timeout information to adjust their functions accordingly.
- **Remote Diagnostics and Monitoring**: The timeout signal enables monitoring of the communication link to the PTO control unit, which is important for fault detection and safety functions.

Comparison with Similar Modules

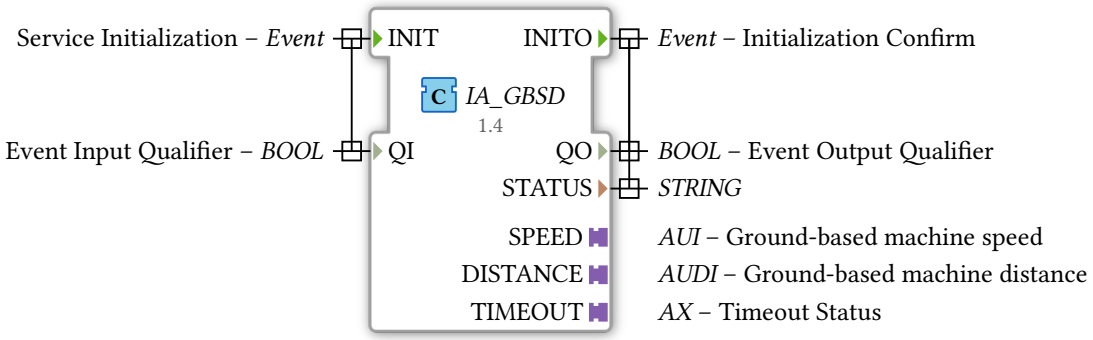
In the ISOBUS environment, there are other adapters for different PTOs (e.g., **IA_RPTO** for the rear PTO) or for general control units. **IA_FPTO** is specifically tailored to the front PTO and contains only the signals relevant to this application (actual/target speed and timeout). Compared to a generic ISOBUS adapter, it therefore offers a higher level of abstraction and easier handling for the application developer.

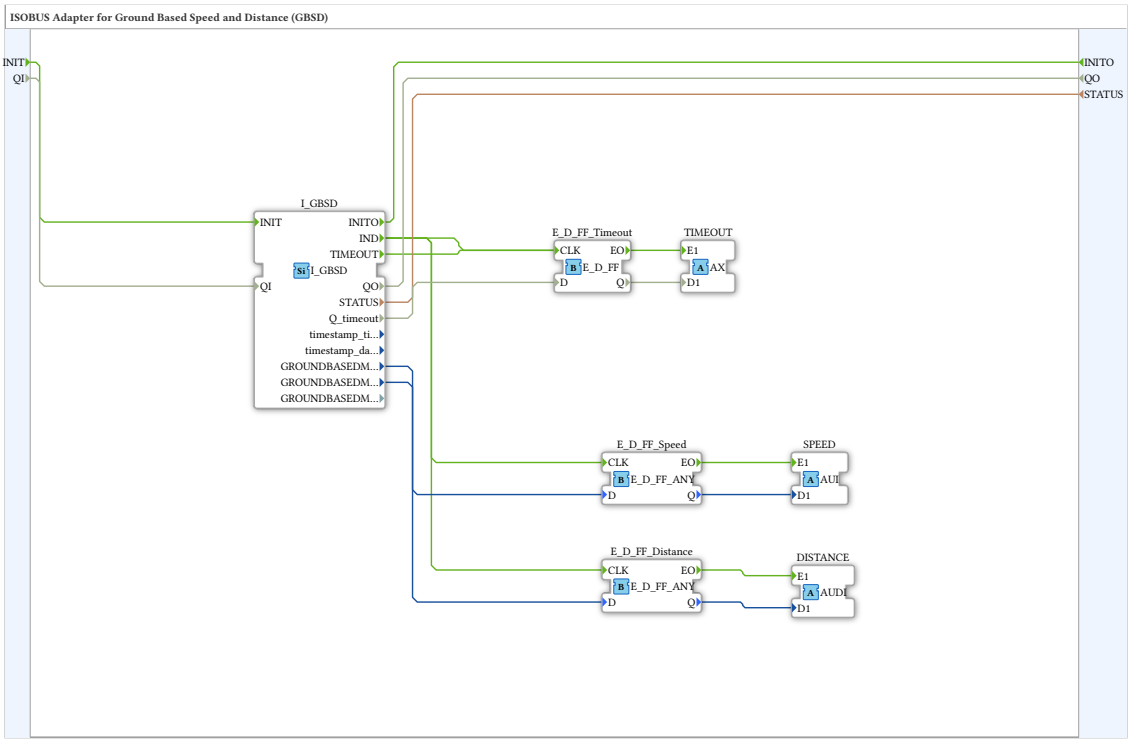
Conclusion

The **IA_FPTO** function block provides a streamlined and specific interface for ISOBUS-compliant use of the front PTO. The encapsulation of the internal communication logic and its provision via unidirectional adapters significantly simplifies integration into control applications. The module is particularly

suitable for tractors and agricultural machinery where reliable speed monitoring and timeout detection are required.

IA_GBSD





• – ○ • – ○ • – ○ •

Introduction

The function block **IA_GBSD** serves as an adapter wrapper for the ISOBUS-compatible module **I_GBSD** (“Ground Based Speed and Distance”). It encapsulates the initialization and data communication of a ground-based speed and distance sensor according to ISO 11783-7 (PGN 65097) and provides the values via unidirectional adapter interfaces.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Service Initialization

Event Outputs

Name	Type	Comment

INITO	EInit	Initialization Confirm

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier

Data Outputs

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

Adapters

Name	Type	Comment
SPEED	adapter::types::unidirectional::AUI	Ground-based machine speed
DISTANCE	adapter::types::unidirectional::AUDI	Ground-based machine distance
TIMEOUT	adapter::types::unidirectional::AX	Timeout status

Functionality

The function block is implemented as a pure adapter wrapper:

- An INIT event at the input is directly forwarded to the internal block **I_GBSD**.
- The data input **QI** activates or deactivates processing.
- After initialization, the **INITO** event is output with the qualifier **QO** and a **STATUS**.
- The calculated values (speed and distance) are output via the adapters **SPEED** and **DISTANCE**. If a timeout occurs, the **TIMEOUT** adapter is activated.
- All data connections are established within the FB network between the inputs/outputs and the internal module.

Technical Features

- **Adapter-based interface**: The output data is not provided as fixed variables, but via unidirectional adapters (AUI, AUDI, AX), allowing for flexible integration with other components.
- **ISOBUS compliance**: The FB implements PGN 65097 according to ISO 11783-7 – the standard for ground-based speed and distance in agricultural technology.
- **Timeout monitoring**: The separate timeout adapter signals whether the sensor values have been updated within an expected time period.
- **Compact design**: There is no separate state machine; all control logic resides in the internal module **I_GBSD**.

State Overview

The function block itself does not have an explicit finite state machine. The internal block goes through an initialization state (triggered by INIT) and an operating state in which speed, distance, and timeout status are continuously updated.

Application Scenarios

- **ISOBUS Control Units (TECUs)**: Integration of a radar sensor or other ground-based speed/distance sensor into an ISOBUS-compatible system.
- **Agricultural Vehicles**: Provision of driving speed and distance traveled for downstream functions (e.g., metered applications, section control).
- **Test and Simulation Environments**: Use of the adapter wrapper to replace the real sensor with a simulated data source.

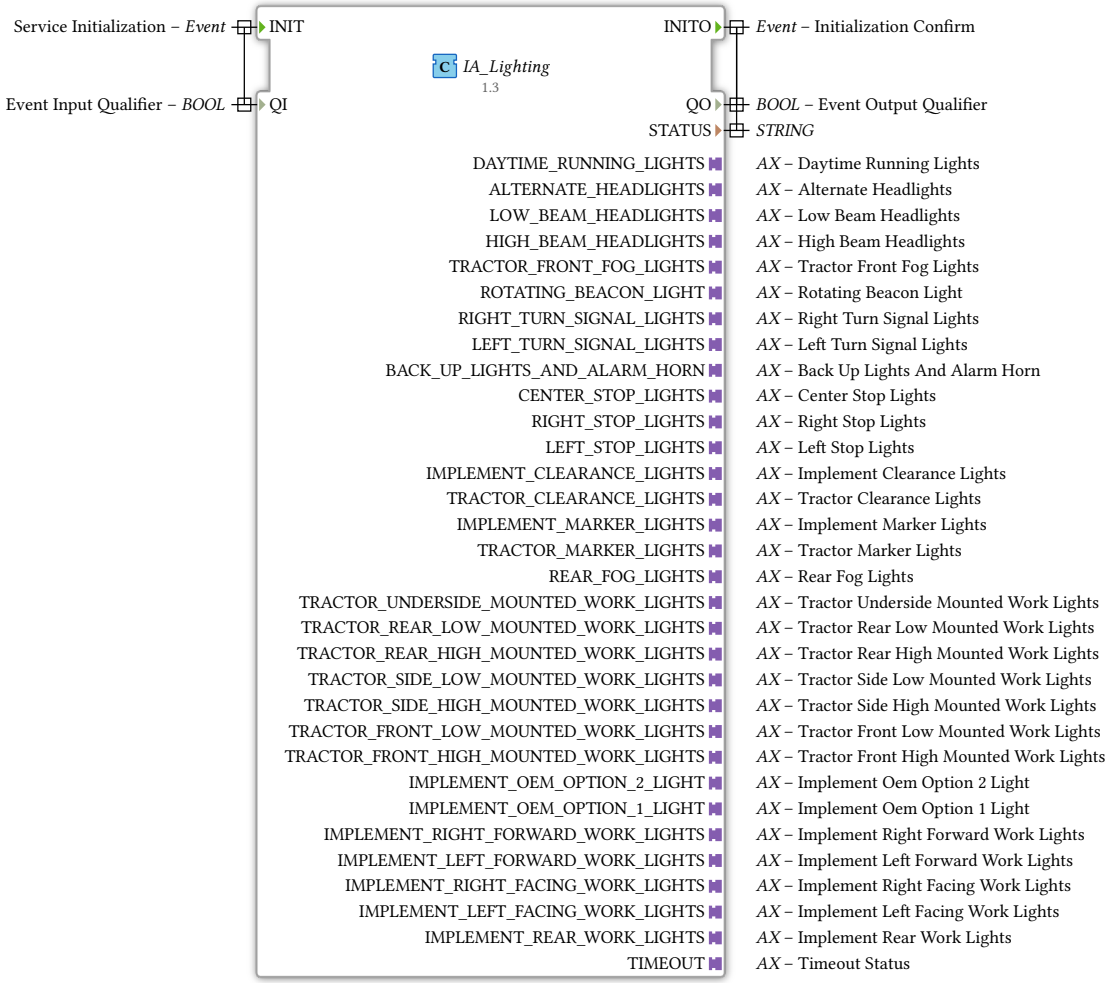
Comparison with Similar Modules

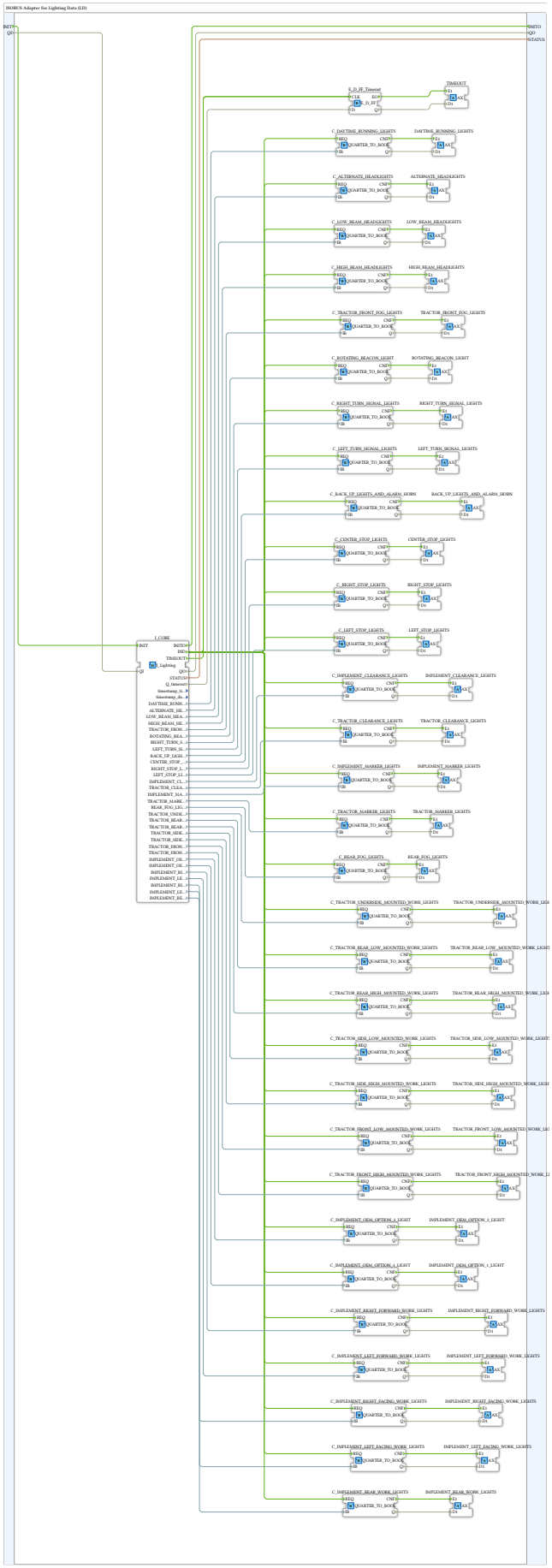
- **IA_GBSD** vs. **I_GBSD**: I_GBSD is the natively programmed ISOBUS module, while IA_GBSD makes its functionality accessible exclusively via adapters. The wrapper simplifies wiring in graphical editors and separates the actual logic from the interface connection.
- **Alternative Adapter Wrappers**: Similar wrappers exist for other ISOBUS PGNs (e.g., for navigation data or hydraulic systems), which also provide adapters for their output values.

Conclusion

IA_GBSD is an efficient adapter wrapper that encapsulates ground-based speed and distance measurement according to ISO 11783-7. The adapter interfaces enable clean, modular integration into higher-level control logic, while the actual ISOBUS implementation remains intact. This makes the component usable for both direct applications and for simulation and testing purposes.

IA_Lighting





• – ◦ • – ◦ • – ◦ •

Introduction

The **IA_Lighting** function block serves as an ISOBUS adapter for lighting data (LD) according to ISO 11783-7 (PGN 65088). It encapsulates an internal `I_Lighting` core and converts its 32-bit integer outputs for each lighting function into individual Boolean signals. A variety of adapter plugs transmit the individual lighting functions, such as daytime running lights, low beams, turn signals, work lights, etc., as separate logical signals to the application.

• – ◦ • – ◦ • – ◦ •

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initialization of the function block. Triggered by `QI`.

Event Outputs

Event	Type	Description
INITO	EInit	Confirmation of successful initialization. Output along with `QO` and `STATUS`.

Data Inputs

Name	Type	Description
QI	BOOL	Qualifier for initialization (enabling).

Data Outputs

Name	Type	Description
QO	BOOL	Output qualifier – indicates whether the block is ready for operation.
STATUS	STRING	Status message (e.g., error text or success message).

Adapters

The block has **32 unidirectional adapter plugs** (type `adapter::types::unidirectional::AX`). Each adapter represents a specific lighting function according to ISO 11783-7 and provides an event output (`E1`) and a data output (`D1`) of type `BOOL`:

Adapter Name	Description
DAYTIME_RUNNING_LIGHTS	Daytime Running Lights
ALTERNATE_HEADLIGHTS	Alternative High Beam (e.g., High Beam Assist)
LOW_BEAM_HEADLIGHTS	Low Beam
HIGH_BEAM_HEADLIGHTS	High Beam
TRACTOR_FRONT_FOG_LIGHTS	Front Fog Lights for Tractor
ROTATING_BEACON_LIGHT	Rotating Beacon
RIGHT_TURN_SIGNAL_LIGHTS	Right Turn Signal
LEFT_TURN_SIGNAL_LIGHTS	Left Turn Signal
BACK_UP_LIGHTS_AND_ALARM_HORN	Reversing Light and Alarm Horn
CENTER_STOP_LIGHTS	Center Brake Light
RIGHT_STOP_LIGHTS	Right Brake Light
LEFT_STOP_LIGHTS	Left Brake Light

IMPLEMENT_CLEARANCE_LIGHTS	Implement Marker Lights
TRACTOR_CLEARANCE_LIGHTS	Tractor Marker Lights
IMPLEMENT_MARKER_LIGHTS	Implement Marker Lights
TRACTOR_MARKER_LIGHTS	Tractor Marker Lights
REAR_FOG_LIGHTS	Rear Fog Lights
TRACTOR_UNDERSIDE_MOUNTED_WORK_LIGHTS	Tractor Work Lights (Underside Mounted)
TRACTOR_REAR_LOW_MOUNTED_WORK_LIGHTS	Tractor Work Lights (Rear, Low Mounted)
TRACTOR_REAR_HIGH_MOUNTED_WORK_LIGHTS	Tractor Work Lights (Rear, High Mounted)
TRACTOR_SIDE_LOW_MOUNTED_WORK_LIGHTS	Tractor work lights (side, low)
TRACTOR_SIDE_HIGH_MOUNTED_WORK_LIGHTS	Tractor work lights (side, high)
TRACTOR_FRONT_LOW_MOUNTED_WORK_LIGHTS	Tractor work lights (front, low)
TRACTOR_FRONT_HIGH_MOUNTED_WORK_LIGHTS	Tractor work lights (front, high)
IMPLEMENT_OEM_OPTION_2_LIGHT	Attachment OEM Option 2 Light
IMPLEMENT_OEM_OPTION_1_LIGHT	Attachment OEM Option 1 Light
IMPLEMENT_RIGHT_FORWARD_WORK_LIGHTS	Attachment work lights (front right)
IMPLEMENT_LEFT_FORWARD_WORK_LIGHTS	Attachment work light (front left)
IMPLEMENT_RIGHT_FACING_WORK_LIGHTS	Attachment work light (side right)
IMPLEMENT_LEFT_FACING_WORK_LIGHTS	Attachment work light (side left)
IMPLEMENT_REAR_WORK_LIGHTS	Attachment work light (rear)
TIMEOUT	Timeout status of the internal core (Boolean signal).

• – ◦ • – ◦ • – ◦ •

Functionality

The function block is initialized via the event input `INIT` with the data value `QI`. Upon successful initialization, the event `INITO` is output, and the data `QO = TRUE` and `STATUS` are set with a success message.

Internally, the module contains a core of type `isobus::tecu::I_Lighting`, which handles the ISOBUS communication for the lighting PGN (Parameter Group Number 65088). The core provides the 32 lighting functions as 32-bit integer values. These integer values are then divided into individual Boolean signals by instances of the module `logiBUS::utils::quarter::QUARTER_TO_BOOL`. Each `QUARTER_TO_BOOL` instance presumably extracts 4 bits from the input value and outputs the corresponding four Boolean outputs – however, the exact structure is application-specific.

The resulting Boolean signals are then provided via the adapter plugs simultaneously with an event (`E1`) on the associated data outputs (`D1`). Thus, when the ISOBUS data is updated, the module provides a synchronous event stream for each individual lighting function.

• – ◦ • – ◦ • – ◦ •

Technical Features

- **ISOBUS Compliance**: The module implements the standardized PGN 65088 (Lighting Data) according to ISO 11783-7 and can be directly connected to an ISOBUS bus.
- **Bit Division**: The internal 32-bit values from the ISOBUS telegram are divided into individual Boolean signals using `QUARTER_TO_BOOL` modules. The term “quarter” indicates a division into groups of 4 bits each.

- **Unidirectional Adapters**: Each adapter plug is unidirectional (output only) and provides both an event (`E1`) and a Boolean value (`D1`). This allows for easy further processing in IEC 61499 applications, e.g., for controlling actuators.
- **Status Output**: In addition to the actual light status, there is a special adapter, `TIMEOUT`, which signals the timeout status of the ISOBUS core.

• – ◦ • – ◦ • – ◦ •

State Overview

The module itself does not have an explicit state machine, as it is essentially a data converter. Its behavior is controlled by the internal kernel `I_Lighting`:

- **Initialization**: After a `INIT` event, the kernel switches to the operational state (provided `QI = TRUE` is present). The process is completed with `INITO`.Acknowledged.
- **Data Provisioning**: As long as the core is active, it updates the output data with each incoming ISOBUS telegram and generates an event on the corresponding adapter for each lighting function.
- **Timeout**: The timeout adapter is set if no ISOBUS messages are received.

• – ◦ • – ◦ • – ◦ •

Application Scenarios

- **Agricultural Control Systems**: Integration of all vehicle lighting (tractor and implement) into an IEC 61499-based control system, e.g., for automatic lighting control according to ISO 11783.
- **ISOBUS Gateway Modules**: This module is suitable as an intermediary layer to convert ISOBUS lighting data into a simpler binary signal format and thus transmit it to programmable logic controllers (PLCs) or visualization systems.
- **Retrofit**: Older tractors without a CAN bus can be equipped with modern ISOBUS lighting control using this adapter.

• – ◦ • – ◦ • – ◦ •

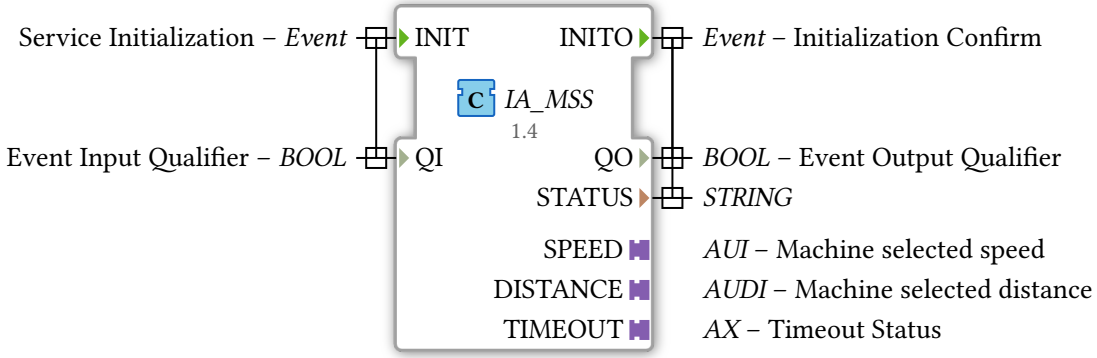
Comparison with similar modules

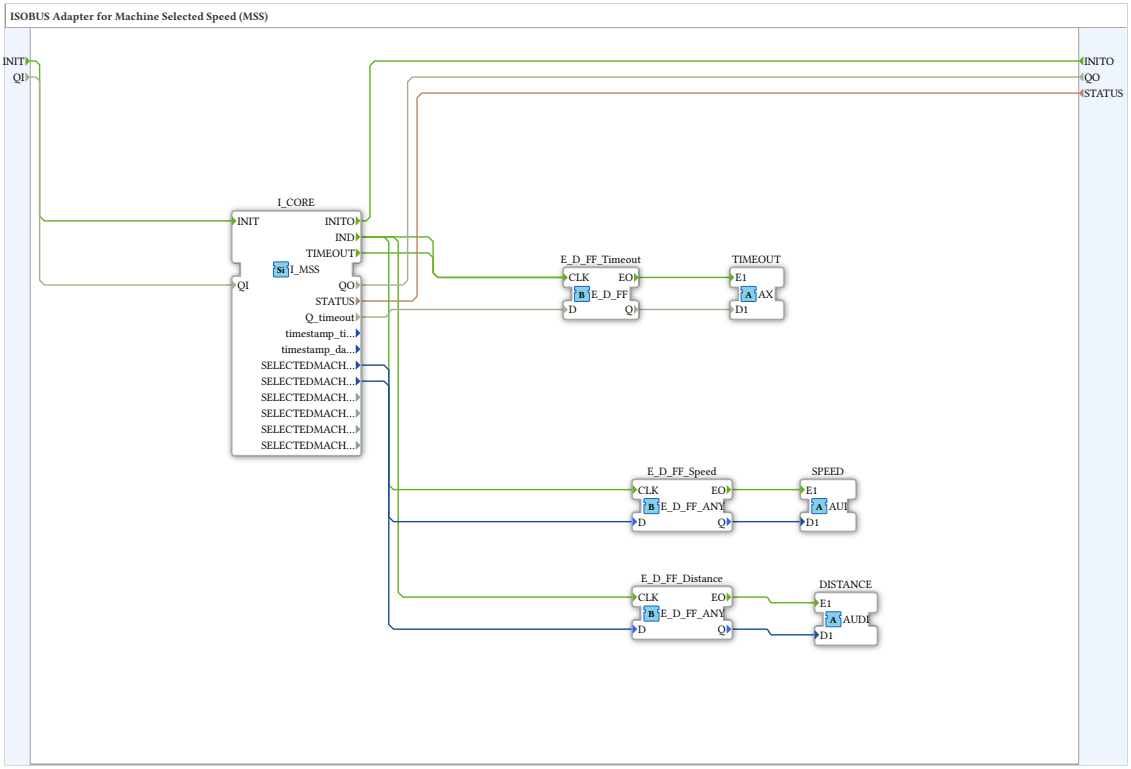
Other ISOBUS adapter wrappers exist for other PGNs (e.g., for work hydraulics, seat control, or PTO control). These modules follow the same principle: An internal, specialized core is connected to the application code via an adapter. The main difference lies in the number and type of output signals – `IA_Lighting` offers a particularly high number of lighting functions with 32 adapters. Other adapters (e.g., `IA_ImplementSteer`) have fewer outputs because they report only a few states.

Conclusion

The function block `IA_Lighting` enables convenient and standardized integration of ISOBUS lighting data into IEC 61499 applications. By splitting the telegram content into individual Boolean signals via adapters, simple further processing in the application logic is achieved. This module is particularly suitable for agricultural control systems that require a complete representation of all common lighting functions according to ISO 11783-7.

IA_MSS





• — ○ • — ○ • — ○ •

Introduction

The **IA_MSS** function block represents an ISOBUS adapter for machine-selected speed (MSS). It encapsulates the communication with the underlying ISOBUS protocol (PGN 61474) and provides the determined values for speed, distance traveled, and timeout status via standardized adapter interfaces. The block is typically embedded in an ISOBUS-compatible control unit (TECU) and enables easy integration into IEC 61499-based automation systems.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Service Initialization; is triggered by the data input `QI`.

Event Outputs

Event	Type	Description
INITO	EInit	Initialization confirmation; outputs the data `QO` and `STATUS`.

Data Inputs

Variable	Type	Description
QI	BOOL	Qualifier for the initialization input.

Data Outputs

Variable	Type	Description
QO	BOOL	Qualifier for the initialization output.

STATUS	STRING	Initialization status message.

Adapter

Adapter	Type	Description
SPEED	adapter::types::unidirectional::AUI	Output for the machine-selected speed (value of type `AUI`).
DISTANCE	adapter::types::unidirectional::AUDI	Output for the machine-selected distance (value of type `AUDI`).
TIMEOUT	adapter::types::unidirectional::AX	Output for the timeout status (value of type `AX`).

Functionality

The **IA_MSS** block is implemented as a pure adapter wrapper. Internally, it contains an instance of the function block `isobus::tecu::I_MSS` (*I_CORE*), which implements the actual ISOBUS logic. The external interfaces of the adapter are mapped to the internal component via event and data connections:

1. **Initialization** – An event on `INIT` (with `QI` set) activates `I_CORE.INIT`. After successful initialization, `INITO` is triggered, which forwards the qualifiers `QO` and `STATUS` to the adapter's outputs.
2. **Data Provision** – The internal module provides the selected speed (`SELECTEDMACHINESPEED`), the distance (`SELECTEDMACHINEDISTANCE`), and the timeout status (`Q_timeout`). These values are output via the adapters `SPEED`, `DISTANCE`, and `TIMEOUT` as soon as the corresponding events (`IND` and `TIMEOUT`, respectively) occur.

The adapter thus provides a clean separation between ISOBUS communication and the application level.

Technical Features

- **Standard Compliance** – The module is based on the ISOBUS standard ISO 11783-7 (PGN 61474) for machine-selected speed.
- **Unidirectional Adapters** – The outputs `SPEED`, `DISTANCE`, and `TIMEOUT` are defined as unidirectional adapters (types `AUI`, `AUDI`, `AX`). They only transmit data from the module to the calling application; feedback is not provided.
- **Adapter Wrapper** – The internal logic is fully encapsulated in component `I_CORE`. `IA_MSS` is used exclusively for interface adaptation and event forwarding.
- **Error Handling** – The initialization status is signaled via `STATUS` (STRING) and `QO` (BOOL). Any timeout is reported via the `TIMEOUT` adapter.

State Overview

Since the function block does not have its own execution state (ECC), the state results from the interaction of input and output events:

State	Description
Not Initialized	After system startup or after a failed initialization. `INIT` can be triggered.
Initialized	After successful execution of `INIT` ⟶ `INITO` with `QO` set. The adapters deliver valid values.
Error/Timeout	If the internal initialization fails or a timeout is detected, this is signaled via `STATUS` and the `TIMEOUT` adapter.

Application Scenarios

- **ISOBUS Control Units (TECU)** – Integration into agricultural machinery to read the speed provided by the tractor or implement.
- **Traction Control** – Use of the selected speed for site-specific applications (e.g., fertilization, crop protection).
- **Distance and Path Measurement** – Evaluation of the distance traveled for controlling working widths or for documentation purposes.

- **Communication Monitoring** – Detection of timeout situations in the ISOBUS network via the `TIMEOUT` adapter.

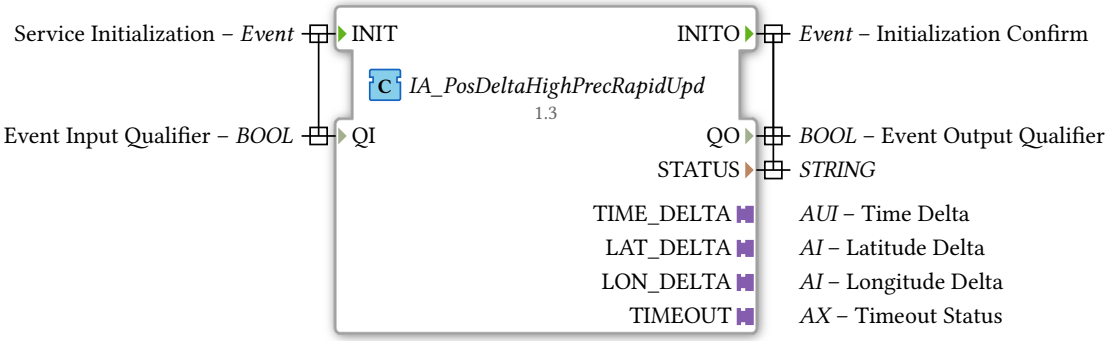
Comparison with Similar Modules

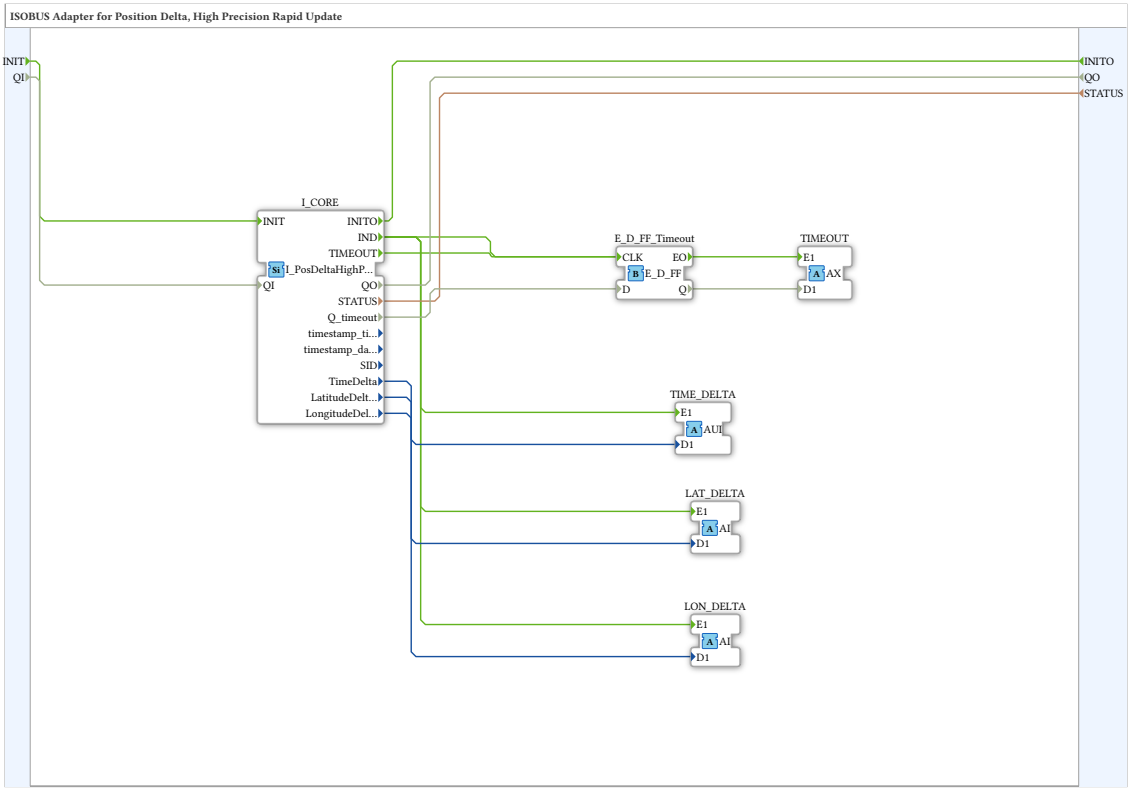
- **I_MSS (Direct)** – The internal module `I_CORE` of type `isobus::tecu::I_MSS` offers the same functionality, but without adapter interfaces. `IA_MSS` facilitates integration into systems based on standardized adapters (AUI, AUDI, AX).
- **Other ISOBUS Adapters** – Numerous adapters exist for other PGNs (e.g., for GPS, working width, engine speed). `IA_MSS` is specifically designed for machine-selected speed.

Conclusion

The **IA_MSS** block is a practical adapter wrapper for the ISOBUS "Machine Selected Speed" data set. It abstracts the communication with the ISOBUS protocol and provides the relevant parameters (speed, distance, timeout) via standardized unidirectional adapters. This enables simple and reusable integration into IEC 61499 applications for agricultural machinery. The block is standards-compliant, robust thanks to timeout monitoring, and particularly suitable for use in TECU-based controllers.

IA_PosDeltaHighPrecRapidUpd





• – ○ • – ○ • – ○ •

Introduction

The function block `IA_PosDeltaHighPrecRapidUpd` is an ISOBUS adapter module for high-precision, rapidly updated position deltas according to the NMEA 2000 standard (PGN 129027). It encapsulates the internal core module `I_PosDeltaHighPrecRapidUpd` and makes its output data available via adapter interfaces. This allows position changes (time, latitude, and longitude deltas) as well as a timeout status to be seamlessly integrated into automation systems.

Interface Structure

Event Inputs

Event	Type	Description
`INIT`	EInit	Initialize the function block; used with `QI`.

Event Outputs

Event	Type	Description
`INITO`	EInit	Initialization confirmation; returns `QO` and `STATUS`.

Data Inputs

Name	Type	Description
`QI`	`BOOL`	Qualifier for INIT event.

Data Outputs

Name	Type	Description
`QO`	`BOOL`	Output qualifier (Initialization successful?).
`STATUS`	`STRING`	Initialization status message.

Adapters

Adapter name	Type (Adapter)	Description
`TIME_DELTA`	`adapter::types::unidirectional::AUI`	Returns the time delta (unsigned integer).
LAT_DELTA`	`adapter::types::unidirectional::AI`	Returns the latitude delta (signed integer).
LON_DELTA`	`adapter::types::unidirectional::AI`	Returns the longitude delta (signed integer).
TIMEOUT`	`adapter::types::unidirectional::AX`	Indicates whether a timeout occurred (Bool).

Functionality

This function block acts as a wrapper around the internal function block `I_PosDeltaHighPrecRapidUpd`. The kernel is initialized when a `INIT` event occurs with a set `QI`. After successful initialization, the kernel reports back with `INITO` and delivers `QO = TRUE` along with a corresponding status. As soon as new position data arrives, the kernel generates the event `IND` (internal, not directly visible), which forwards the data to the three delta adapters. Simultaneously, if a timeout occurs, the event `TIMEOUT` is triggered, and the corresponding adapter is updated.

... The internal data connections ensure that the values from `TimeDelta`, `LatitudeDelta`, `LongitudeDelta`, and `Q_timeout` are transferred to the corresponding adapter data port `D1`.

Technical Features

- **Standard Compliance**: The module implements NMEA 2000 PGN 129027 – "Position Delta, High Precision Rapid Update".
- **Adapter-Based**: The output data is provided via flexible, unidirectional adapters, allowing direct connection to other modules in a 4diac system.
- **High Precision & Update Rate**: Optimized for applications requiring accurate and rapid position changes (e.g., in agriculture or vehicle navigation).
- **Initialization with Status**: The function block provides an explicit initialization confirmation (QO, STATUS) and can therefore be integrated into safety-critical processes.

State Overview

The function block does not have an explicit finite state machine (ECC) in its XML definition. Its behavior is entirely controlled by the internal core function block `I_PosDeltaHighPrecRapidUpd`:

1. **Wait for Initialization**: After system startup, `QI = FALSE` is present; `INIT` is not triggered.
2. **Start Initialization**: Upon the `INIT` event with `QI = TRUE`, the core is activated.
3. **Initialization Confirmed**: The core reports `INITO` with `QO = TRUE` and a positive status; data provisioning then begins.
4. **Data Update**: With each new position delta, the adapters are automatically updated with the current values.
5. **Timeout**: If no more valid data is received, the core sets `Q_timeout = TRUE` and triggers the `TIMEOUT` event.

Application Scenarios

- **Precision Agriculture**: Monitoring of position changes of a tractor or agricultural machine in real time (e.g., for steering systems or application maps).
- **Vehicle Navigation**: Processing of high-precision GNSS deltas in autonomous vehicles.
- **ISOBUS Integration**: Connection to existing ISOBUS control units based on PGN 129027.
- **Safety Monitoring**: Detection of data failures via the timeout adapter.

Comparison with Similar Modules

Similar ISOBUS adapter modules (e.g., `IA_PosSpeed` or `IA_TimeDate`) also encapsulate specific PGNs and provide their data via adapters. The `IA_PosDeltaHighPrecRapidUpd` differs in that:

- Specialization in **position deltas** (not absolute positions)
- Support for **time, latitude, and longitude deltas** as well as a separate **timeout** signal
- Focus on high **precision** and fast **update rates**

Other modules often only provide a single measurement or lack timeout monitoring.

Conclusion

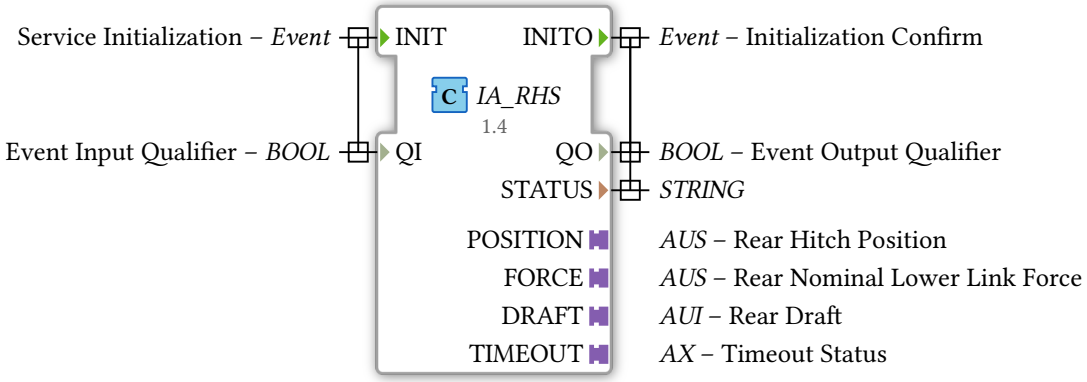
The `IA_PosDeltaHighPrecRapidUpd` is a specialized ISOBUS adapter for high-precision position changes according to NMEA 2000 PGN 129027. It offers a standards-compliant, adapter-based interface that integrates seamlessly into 4diac systems. Thanks to its integrated timeout detection and explicit initialization confirmation, it is particularly suitable for safety-critical automation solutions in agriculture and automotive engineering.

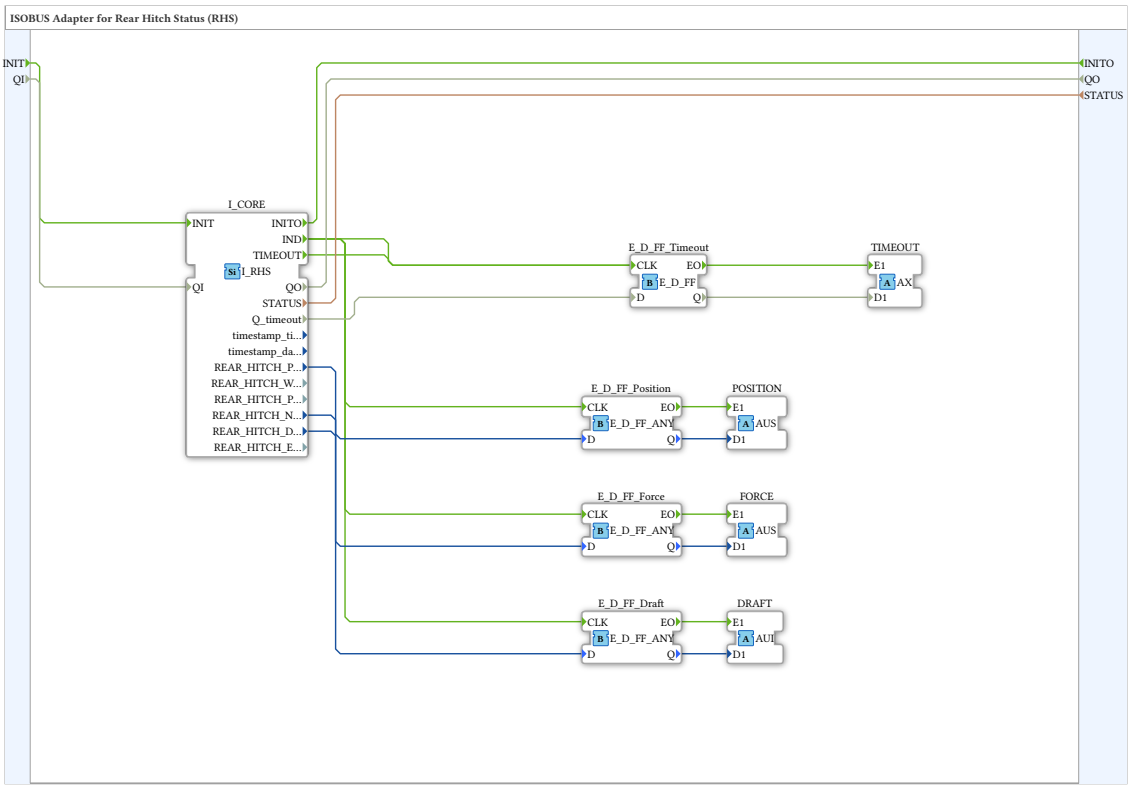
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

IA_RHS





• – ○ • – ○ • – ○ •

Introduction

The function block **IA_RHS** (ISOBUS Adapter for Rear Hitch Status) serves as an interface between the ISOBUS system (according to ISO 11783-7, PGN 65093) and an application network. It encapsulates the communication with the internal module I_RHS and provides the measured rear hitch data (position, force, tractive force) as well as the timeout status via standardized, unidirectional adapters. This module is primarily used in tractor and agricultural machinery control systems.

Interface Structure

Event Inputs

Name	Type	Comment	With Data
INIT	EInit	Service Initialization	QI

Event Outputs

Name	Type	Comment	With Data
INITO	EInit	Initialization Confirm	QO, STATUS

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier

Data Outputs

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

Adapters

Name	Type	Comment
POSITION	OFF	Rear Hitch Position
FORCE	OFF	Rear Nominal Lower Link Force
DRAFT	AUI	Rear Draft
TIMEOUT	AX	Timeout Status

Functionality

The function block is activated by the **INIT** event. The qualifier signal **QI** is then forwarded to the internal function block **I_RHS**. After successful initialization (or an error), the function block reports back via the **INITO** event and outputs the **QO** and **STATUS** signals. Simultaneously, the data supplied by I_RHS (position, lower link force, draft force) is applied to the corresponding adapters **POSITION**, **FORCE**, and **DRAFT**. The timeout status of I_RHS is output via the **TIMEOUT** adapter. The adapters each use a unidirectional data format (e.g., OFF for analog, unscaled values). Output to the adapters occurs via a separate event (through internal connections of I_CORE.IND). This allows a receiving module to react to the arrival of new data.

Technical Features

- The module is based on the ISOBUS standard ISO 11783-7 and uses PGN 65093.
- The internal calculations and ISOBUS communication are performed entirely by the **I_RHS** module. IA_RHS serves solely as an adapter wrapper for providing data to the 4diac network.
- All four adapters are unidirectional (output), meaning data is only sent but not received by the connected module.
- Data output to the adapters occurs simultaneously with the **IND** event (internal), which originates from I_RHS.
- Timeout information is provided separately from the main data path.

State Overview

This function block does not have explicitly modeled states. Its behavior is determined by the internal function block I_RHS:

- **Initialization (INIT)**: The qualifier QI controls activation.
- **Operation**: After successful initialization, I_RHS cyclically or event-driven delivers new data to the adapters.
- **Error/Timeout**: A timeout is signaled via the TIMEOUT adapter, and the status is output as text via the **STATUS** output.

Application Scenarios

- **Tractor Control with ISOBUS Connection**: Reading the rear linkage data (hitch status) from an ISOBUS-compatible terminal or control unit and forwarding it to a higher-level controller.
- **Agricultural Implements**: Connecting implements that require the rear linkage position or force (e.g., plow depth control).
- **Diagnostics and Monitoring**: Recording timeout status and system status for maintenance and fault analysis.

Comparison with Similar Modules

Unlike a general ISOBUS receiver (e.g., a raw PGN parser), **IA_RHS** abstracts the ISOBUS-specific message processing and provides the data in standardized adapter formats. Compared to a direct I_RHS module, IA_RHS offers the advantage that data output is via separate adapters, enabling loose coupling in the network and easier reuse. Modules such as **IA_RHC** (Rear Hitch Control) or **IA_PTO** follow a similar adapter concept but are designed for other ISOBUS functions.

Conclusion

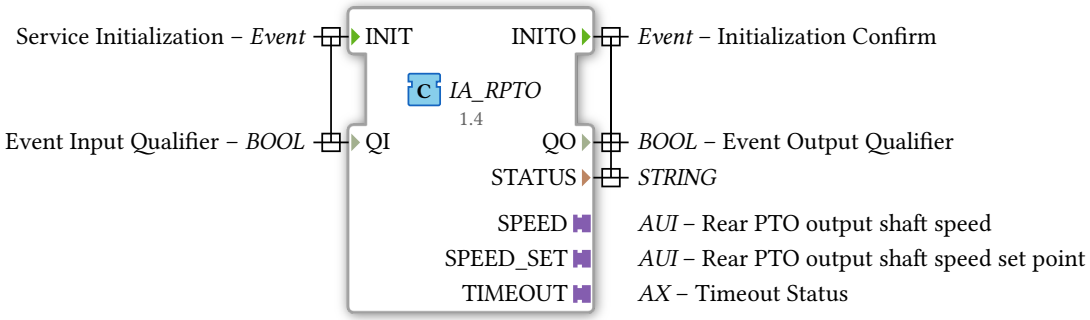
The IA_RHS module is a practical, standards-compliant encapsulation of the rear hitch status according to ISO 11783-7. It simplifies the integration of ISOBUS data into a 4diac-based control system by hiding the complex bus interface behind easy-to-use adapters. The division into multiple data adapters and the provision of a timeout status make it flexibly applicable to various agricultural technology applications.

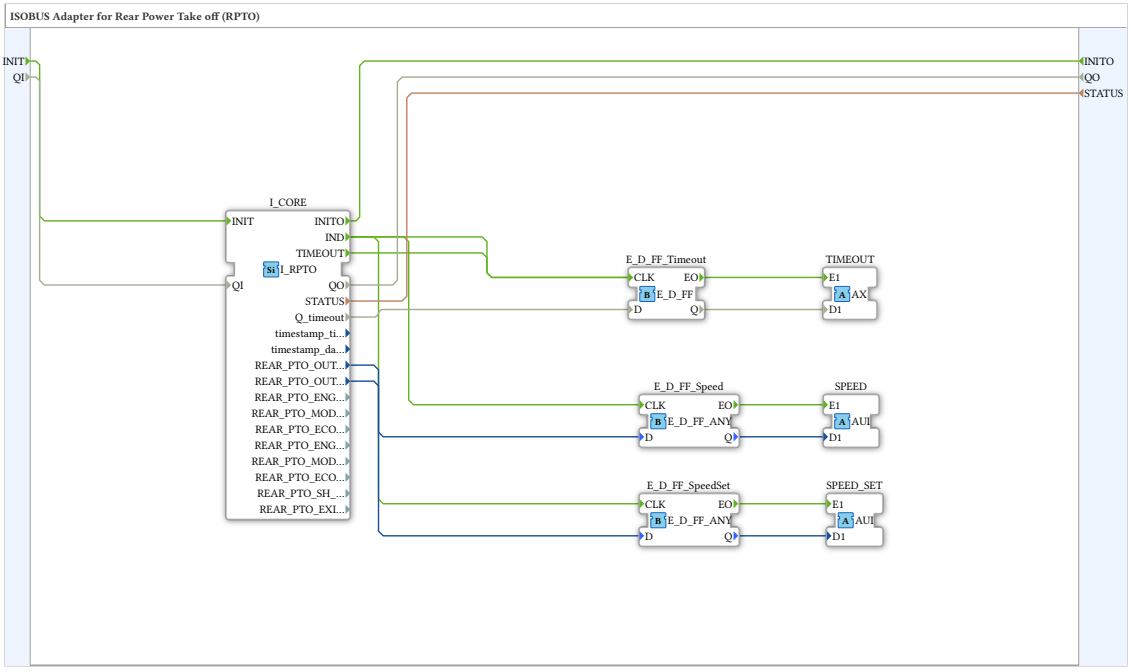
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

IA_RPTO





Introduction

The IA_RPTO function block is an ISOBUS adapter for the rear power take-off (RPTO) according to ISO 11783-7 (PGN 65091). It provides an encapsulated interface to an internal I_RPTO function block and outputs the most important parameters (speed, setpoint, timeout status) via adapter ports.

Interface Structure

Event Inputs

Name	Type	Comment	With Data
INIT	EInit	Service Initialization	QI

Event Outputs

Name	Type	Comment	With Data
INITO	EInit	Initialization Confirm	QO, STATUS

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier

Data Outputs

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

Adapter

Name	Type	Comment
SPEED	adapter::types::unidirectional::AUI	Rear PTO output shaft speed

SPEED_SET	adapter::types::unidirectional::AUI	Rear PTO output shaft speed set point
TIMEOUT	adapter::types::unidirectional::AX	Timeout Status

Functionality

This function block internally instantiates a `I_RPTO` block, which handles the actual ISOBUS communication for the rear PTO shaft. Initialization is triggered via an INIT event; upon successful execution, INITO acknowledges with QO = TRUE and a corresponding STATUS. The values `REAR_PTO_OUTP_SHAFT_SPEED`, `REAR_PTO_OUTP_SH_SPEED_SET`, and `Q_timeout` of the inner block are forwarded directly to the corresponding adapter ports SPEED, SPEED_SET, and TIMEOUT.

Technical Specifications

– The block operates according to ISO 11783-7 (PGN 65091) and is intended for use in agricultural control units (tractor ECUs).
  – All adapters are unidirectional (AUI, AX), meaning they only transmit data in one direction.
  – The inner block is from the package `isobus::tecu`.
  – License: Eclipse Public License 2.0.

State Overview

The function block has two main states: **not initialized** and **initialized**. Upon receiving INIT, the inner block performs the initialization. Upon success, QO = TRUE is set and the STATUS displays a positive value. If an error or timeout occurs, this is signaled via the TIMEOUT adapter and the STATUS. A detailed state diagram is not included as part of the outer description, as all logic is implemented within the inner block.

Application Scenarios

– Integration of an ISOBUS-compatible rear PTO into a tractor control unit.
  – Monitoring and control of the PTO speed via the ISOBUS network.
  – Time monitoring of data communication to detect connection interruptions.

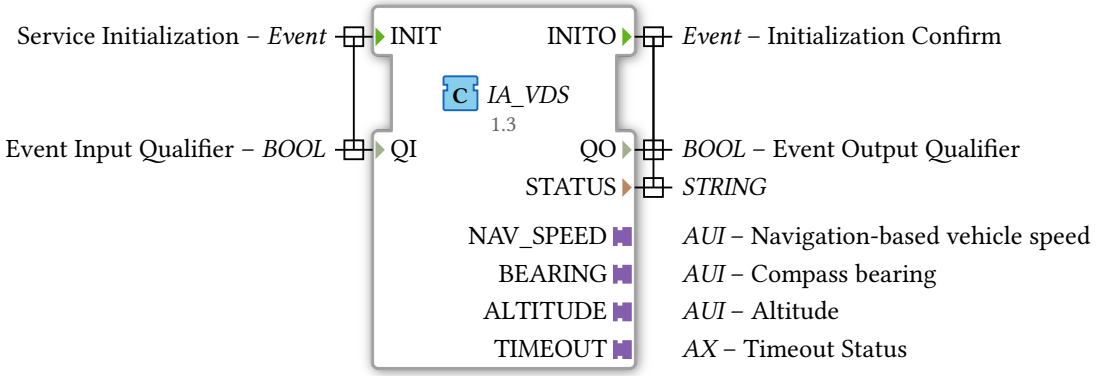
Comparison with Similar Blocks

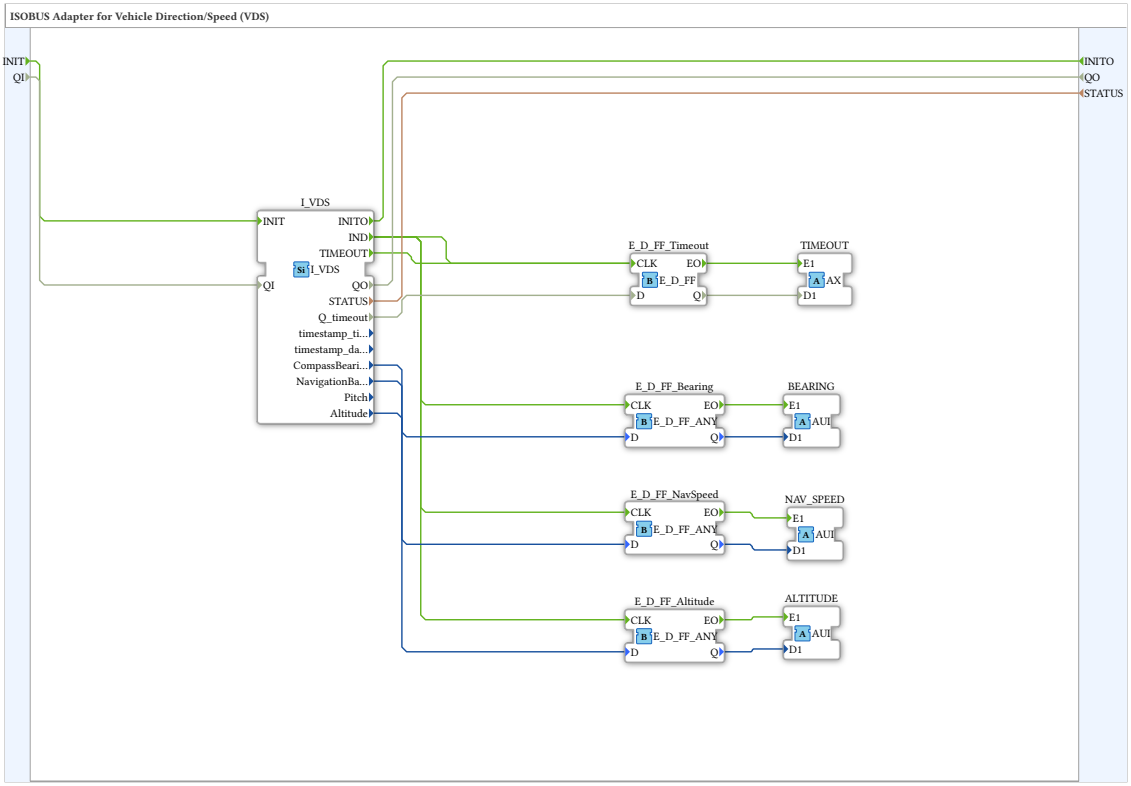
Related adapter blocks exist for other ISOBUS functions, e.g., IA_FPTO (front PTO), IA_HITCH (lift mechanism), or IA_TSC (tractor control). The main difference lies in the PGN (65091) used and the specific data points (speed, setpoint, timeout). IA_RPTO thus offers an interface specifically tailored to the rear PTO.

Conclusion

IA_RPTO encapsulates the ISOBUS communication for the rear PTO and provides the relevant information via clearly defined adapter ports. The module simplifies integration into IEC 61499 applications and ensures standards-compliant connection according to ISO 11783-7.

IA_VDS





• — ○ • — ○ • — ○ •

Introduction

The **IA_VDS** function block is an ISOBUS adapter for vehicle direction and speed according to ISO 11783-7 (PGN 65256). It encapsulates access to the corresponding CAN bus data and provides it as modern 4diac adapter interfaces. The block is initialized via an INIT event and subsequently provides continuous navigation speed, compass direction, altitude, and a timeout status.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initializes the block. Expects the qualifier QI.

Event Outputs

Event	Type	Description
INITO	EInit	Confirms successful initialization. Returns QO and STATUS.

Data Inputs

Name	Type	Description
QI	BOOL	Event Input Qualifier (TRUE = Initialization active).

Data Outputs

Name	Type	Description
QO	BOOL	Event Output Qualifier (TRUE = Initialization OK).
STATUS	STRING	Message text (e.g., error description or “Ready”).

Adapter

Adapter	Type	Description
NAV_SPEED	AUI	Navigation-based vehicle speed (unidirectional, receiver).
BEARING	AUI	Compass direction (unidirectional, receiver).
ALTITUDE	AUI	Altitude above sea level (unidirectional, receiver).
TIMEOUT	AX	Timeout status (unidirectional, receiver).

All adapters are implemented as plugs (outgoing) – the IA_VDS sends the data to the connected sinks.

Functionality

The module internally contains another function block, **I_VDS** (from the library `isobus::tecu`), which handles the actual ISOBUS communication. Upon arrival of the INIT event, the initialization of I_VDS is triggered. After successful initialization, I_VDS reports this via its INITO event, which is forwarded to the INITO output of IA_VDS. Simultaneously, I_VDS begins cyclically updating the values for NavigationBasedVehicleSpeed, CompassBearing, and Altitude. These values are output via the corresponding adapters (NAV_SPEED, BEARING, ALTITUDE) as soon as the IND event from I_VDS arrives. If a timeout occurs (e.g., a CAN bus error), this is reported via the TIMEOUT adapter.

Technical Features

- The module is based on the **Eclipse Public License 2.0 (EPL-2.0)** and is designed as an ISOBUS adapter for PGN 65256.
- Data is distributed asynchronously to the adapter outputs via dedicated events (IND), enabling decoupled processing in the 4diac environment.
- The internal block I_VDS is implemented in the `isobus::tecu` library and optimized for direct bus access.
- The timeout status is provided by a dedicated adapter (type AX) to clearly separate error handling from the normal data stream.

State Overview

The IA_VDS does not have an explicit state machine; its behavior is determined by the internal I_VDS:

1. **Initial** – No INIT signal has been received yet.
2. **Initialization in progress** – After INIT, as long as I_VDS is not yet ready.
3. **Active** – After successful initialization (INITO = TRUE). Values are updated regularly.
4. **Error** – In case of timeout or invalid bus data (STATUS contains an error message, QO = FALSE).

Application Scenarios

- **Agricultural Machinery Control** – Integration of driving speed, direction, and height into control logic (e.g., for site-specific applications).
- **ISOBUS Network Connection** – As an adapter in 4diac applications that use ISOBUS data from a tractor or harvester.
- **Telemetry Systems** – Forwarding of VDS data to higher-level systems (cloud, display).

Comparison with Similar Modules

- **I_TC** (ISOBUS Tractor Control) – Provides general tractor data (speed, PTO speed, linkage) via similar adapters, but not specifically the VDS-PGN.
- **I_GNSS** (ISOBUS GNSS Adapter) – Provides GPS-based position data, but without a focus on direction/speed via CAN.
- **IA_VDS** is specialized for pure vehicle direction/speed information (PGN 65256) and has a particularly streamlined interface.

Conclusion

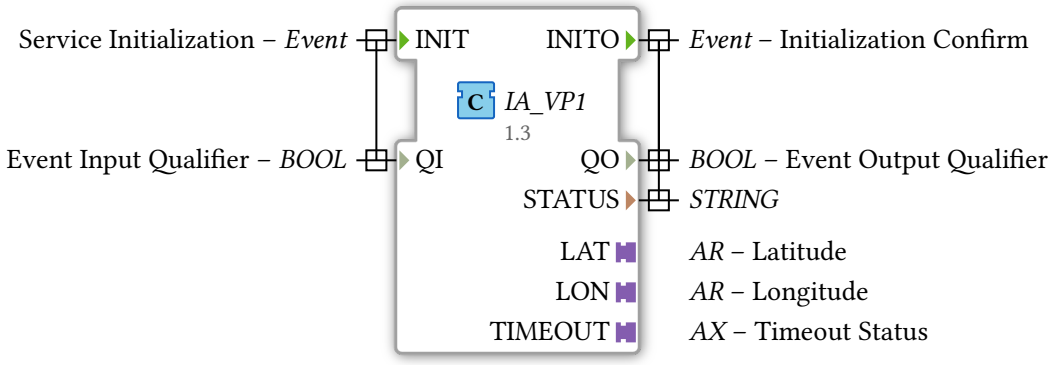
The **IA_VDS** is a practical module for easily integrating ISO-11783-7-specific vehicle data (direction/speed) into a 4diac application. The adapter-based output keeps the architecture modular and extensible. The division into data and timeout adapters facilitates robust error handling. Ideal for all applications requiring precise, bus-based vehicle dynamics.

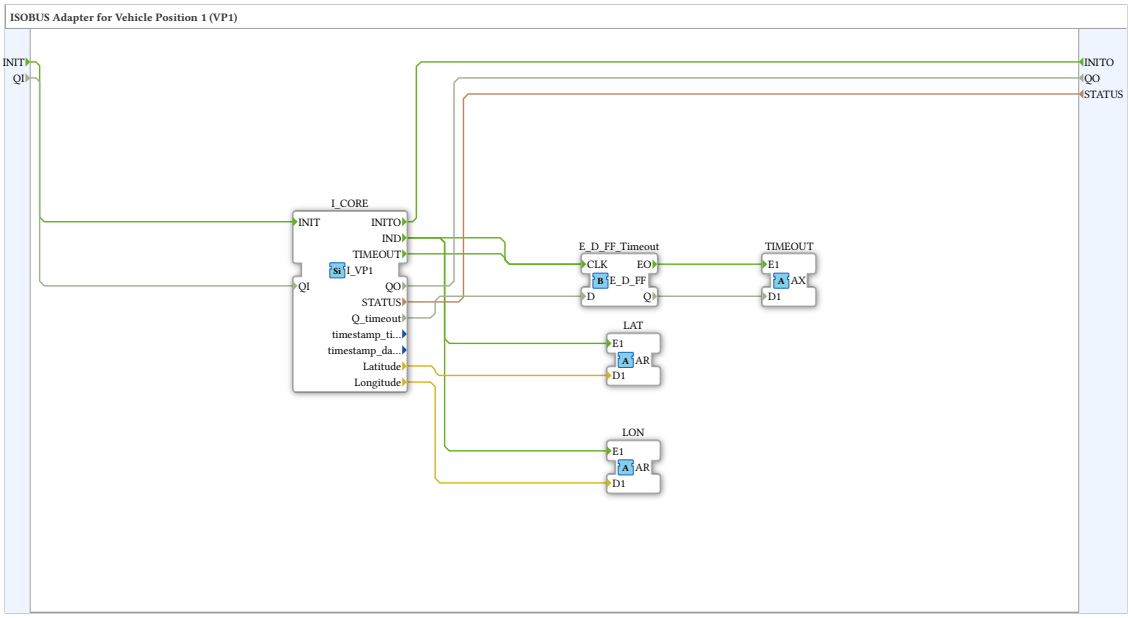
Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

IA_VP1





• – ○ • – ○ • – ○ •

Introduction

The function block **IA_VP1** is an ISOBUS adapter for Vehicle Position 1 (VP1) according to ISO 11783-7 (PGN 65267). It encapsulates the communication for the vehicle position and provides the received data via unidirectional adapter interfaces for further processing in the 4diac IDE.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Service Initialization

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization Confirm

Data Inputs

Variable	Type	Comment
QI	BOOL	Event Input Qualifier

Data Outputs

Variable	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Status Information

Adapters

Adapter	Type	Comment
LAT	adapter::types::unidirectional::AR	Latitude
LON	adapter::types::unidirectional::AR	Longitude
TIMEOUT	adapter::types::unidirectional::AX	Timeout Status

Functionality

The function block internally contains an instance of the function block `I_VP1` from the library `isobus::tecu`. Upon arrival of the INIT event, initialization is triggered – the qualifier `QI` determines whether initialization is active. After successful initialization, the INITO event is triggered, accompanied by the output data `QO` and `STATUS`.

The actual position data (latitude and longitude) and the timeout status are output via the unidirectional adapters. Each adapter is triggered by an event (e.g., `IND` from the core module) and provides the corresponding data value (`D1`) at the adapter interface.

Technical Features

- **Unidirectional Adapters:** The LAT and LON interfaces use the type `AR`, which provides only one output data value. The TIMEOUT adapter uses the type `AX` for a Boolean timeout status.
- **ISOBUS Compliance:** The module implements the Vehicle Position 1 (VP1) message format according to ISO 11783-7.
- **Latitude/Longitude Provisioning:** The values are passed directly from the internal core block and are available as floating-point numbers in the adapter.
- **Timeout Monitoring:** The TIME OUT adapter signals whether the position data has arrived within the expected time.

State Overview

The block does not have an explicitly displayed state machine. Its behavior is event-driven:

- **Inactive:** No initialization occurs before the INIT event.
- **Initialization:** As soon as INIT arrives with `QI=TRUE`, the initialization sequence is started.
- **Active:** After successful INITO, the position and timeout data are available at the adapters. The data is updated with each new internal event (triggered by the core).
- **Error Case:** The status string and `QO` provide information about errors or timeouts.

Application Scenarios

- **Agricultural Vehicles:** Reading the current GPS position (latitude/longitude) via the ISOBUS on-board computer.
- **Precision Agriculture:** Using the position data to control section control, variable application rates, or guidance systems.
- **Documentation and Telemetry:** Recording vehicle movements and position data for fleet management.

Comparison with Similar Modules

- **IA_VP0, IA_GP, IA_PD:** Other ISOBUS position and data adapters (e.g., for speed, machine position). In contrast, IA_VP1 focuses solely on the vehicle position (latitude, longitude) and the timeout status. The data is provided here as unidirectional adapters and not as simple data outputs.
- **I_VP1 (Core Block):** The internal core block offers the same functionality, but without the adapter encapsulation. IA_VP1 makes the interfaces more accessible for the modular 4diac environment.

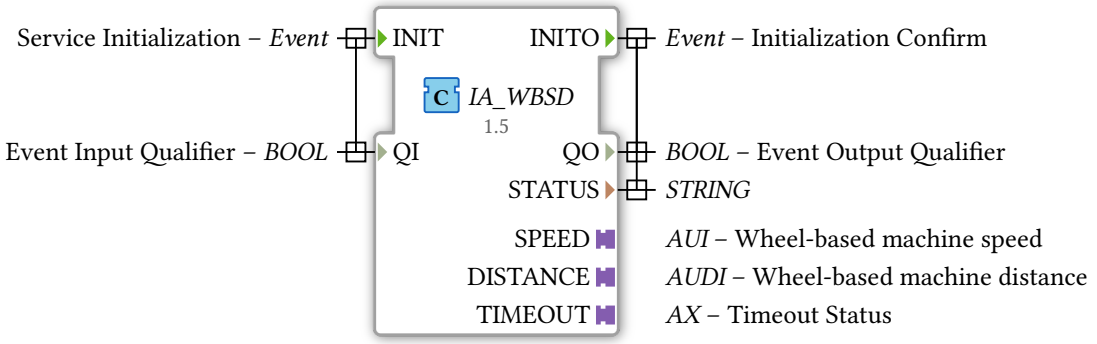
Conclusion

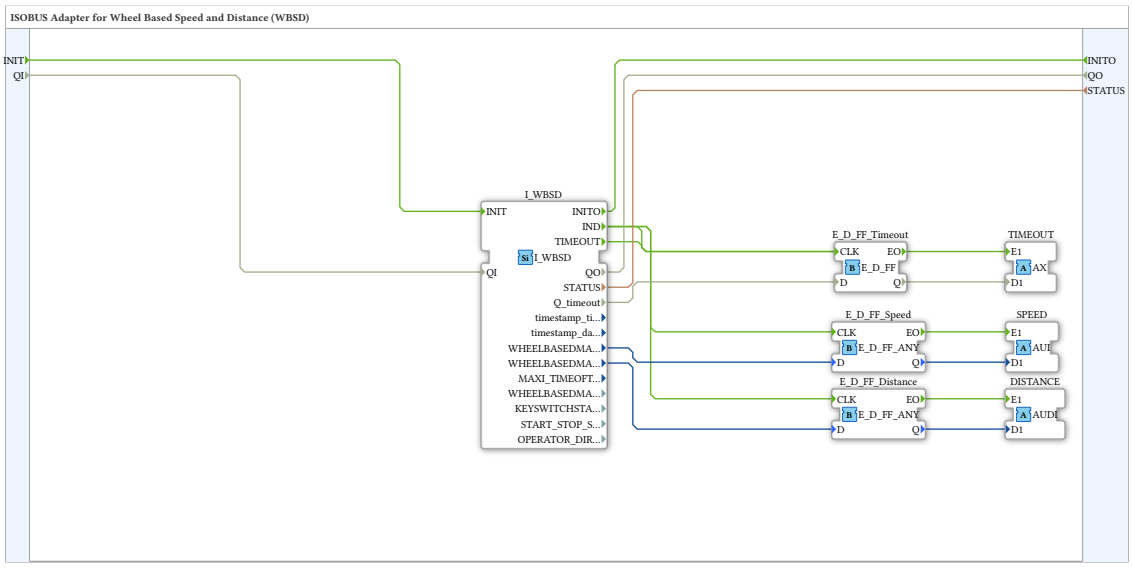
The **IA_VP1** function block is a practical and standards-compliant block for integrating vehicle position data from ISOBUS into the 4diac world. The use of unidirectional adapters enables clean separation and easy integration into larger automation networks. Thanks to its standards compliance and simple initialization, it is ideally suited for agricultural control systems and IoT applications.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

IA_WBSD





• – ○ • – ○ • – ○ •

Introduction

The function block **IA_WBSD** (ISOBUS Adapter for Wheel Based Speed and Distance) serves as an interface wrapper for ISOBUS communication of wheel-based speed and distance values according to ISO 11783-7. It encapsulates the internal module `I_WBSD` and provides its functionality via standardized adapter connections.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Service Initialization

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization Confirm

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier (Initialization Confirmation)

Data Outputs

Name	Type	Comment
QO	BOOL	Event Output Qualifier (Initialization Result)
STATUS	STRING	Status Message (e.g., Error or Success)

Adapters

Adapter	Type	Comment
SPEED	adapter::types::unidirectional::AUI	Wheel-based machine speed (unidirectional, analog value)
DISTANCE	adapter::types::unidirectional::AUDI	Wheel-based machine distance (unidirectional)
TIMEOUT	adapter::types::unidirectional::AX	Timeout status (unidirectional)

Functionality

The **IA_WBSD** function block is a pure adapter wrapper. It connects the external event and data interfaces as well as the three adapters directly to the internal function block `I_WBSD` (of type `isobus::tecu::I_WBSD`).

- The initialization of the ISOBUS service is triggered via the **INIT** input and the qualifier signal **QI**.
- After successful initialization, **INITO** is triggered and **QO** is set to `TRUE`. **STATUS** contains textual feedback.
- The **SPEED** and **DISTANCE** adapters provide the current measured values for wheel-based speed and distance, respectively.
- The **TIMEOUT** adapter signals whether a timeout has occurred in the ISOBUS communication.

The internal logic (calculation, timeout monitoring, ISOBUS frame handling) is handled entirely by the internal function block `I_WBSD`; **IA_WBSD** makes this available externally as an IEC 61499-compliant adapter interface.

Technical Features

- **Standard:** ISO 11783-7 (ISOBUS) – PGN 65096
- **Development History:**
- Version 1.0: Initial implementation as an adapter wrapper for `I_WBSD`
- Version 1.1: Bug fixes – correct connection between internal pins and adapter plugs through additional conversion blocks
- **Compiler Package:** `isobus::tecu`
- **License:** Eclipse Public License 2.0

- **Author:** - (not named)

State Overview

The block does not have its own explicit state machine. Its behavior is controlled by the internal function block `I_WBSD`:

- **Idle State:** No initialization pending.
- **Initialization Run:** After **INIT** with **QI** = **TRUE**, the internal process is started. If **QI** = **FALSE**, no action is taken.
- **Initialization Complete: INITO** is triggered, **QO** indicates success (`TRUE`) or failure (`FALSE`). Afterward, measured values and timeout status can be continuously received via the adapters.

Application Scenarios

- **Agricultural Machine Control:** Integration of wheel-based speed and distance sensors into an ISOBUS control system (e.g., for tractors, harvesters).
- **Positioning without GPS:** Use of the wheel sensors for relative position determination in field work applications.
- **Monitoring of Driving Cycles:** Acquisition of speed and distance data for diagnostics or process control.

Comparison with Similar Function Blocks

- **IA_WBSD vs. Direct I_WBSD:** The internal function block `I_WBSD` provides the pure ISOBUS implementation, while **IA_WBSD** encapsulates this in an IEC 61499 adapter interface. This facilitates reuse in various tools (e.g., 4diac IDE) and loose coupling with other function blocks.
- **IA_WBSD vs. Other ISOBUS Adapters:** Specific to the PGN 65096 (Wheel Based Speed and Distance). Adapters for other PGNs (e.g., for GPS position or motor speed) have a similar structure but differ in data types and internal interpretation.

IA_WBSD vs. Other ISOBUS Adapters:

Conclusion

The **IA_WBSD** is a practical adapter wrapper for the ISOBUS module `I_WBSD`. It reduces the complexity of integrating wheel-based speed and distance data into IEC 61499 projects and provides the functionality via standardized unidirectional adapters. Thanks to its EPL 2.0 license, it can be freely used and adapted in your own projects. Version 1.1 ensures that all internal connections are correctly routed to the adapter plugs.

🌐 Related topic subpages on ms-muc-docs.de

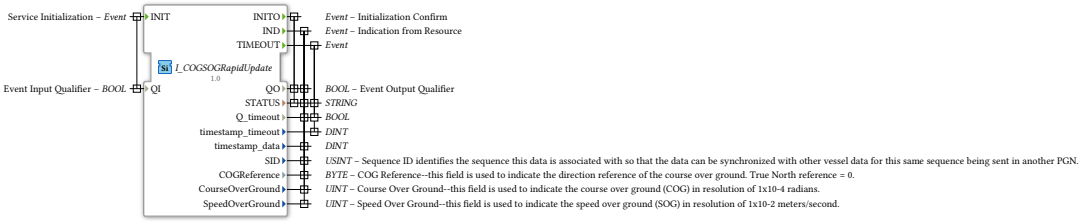
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

I_COGSOGRapidUpdate

• — ○ • — ○ • — ○ •

Introduction

The function block `I_COGSOGRapidUpdate` implements the processing of the NMEA 2000 Parameter Group Number (PGN) 129026 “COG & SOG, Rapid Update”. This block is used to receive and provide navigation data, specifically the current course over ground (COG) and speed over ground (SOG) at a high update rate. It is designed for use in maritime or mobile machinery control systems based on the ISOBUS standard.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block. Triggered together with the qualifier `QI`.

Event Outputs

- **INITO**: Confirms successful initialization. Triggers the outputs `QO` and `STATUS`.
- **IND**: Indicates successful reception and processing of new COG/SOG data. Triggers the corresponding data outputs.
- **TIMEOUT**: Triggered if a timeout occurs during data reception.

Data Inputs

- **QI** (BOOL): Qualifier for the INIT event input. Controls the initialization (`TRUE` = start).

Data Outputs

- **QO** (BOOL): Qualifier for the INITO and IND event outputs. Indicates the general operating status.
- **STATUS** (STRING): Status message providing additional information (e.g., error descriptions).
- **Q_timeout** (BOOL): Indicates whether the last received event was a timeout (`TRUE`) or valid data (`FALSE`).
- **timestamp_timeout** (DINT): Timestamp associated with the TIMEOUT event.
- **timestamp_data** (DINT): Timestamp of the last received valid COG/SOG data.
- **SID** (USINT): Sequence identifier. Enables synchronization of this data with other PGNs sent by the vessel in the same cycle.
- **COGReference** (BYTE): Reference direction for the course over ground. The value `0` represents the "True North" reference. * **CourseOverGround** (UINT): Course over ground (COG). The unit is 1x10 radians.
- **SpeedOverGround** (UINT): Speed over ground (SOG). The unit is 1x10 meters per second.

Adapter

This function block has no adapter interfaces.

Functionality

The block acts as a passive receiver for the NMEA 2000 PGN 129026. After initialization via `INIT` with `QI=TRUE`, it waits for incoming data frames. Upon receiving a valid frame, the contained data (COG, SOG, reference, SID) is decoded and made available via the `IND` output along with the associated data values. Simultaneously, `Q_timeout` is set to `FALSE`. If no new data frame occurs within a configured time window, a `TIMEOUT` event is generated, and `Q_timeout` is set to `TRUE`. The `STATUS` output can be used for diagnostic purposes.
STATUS`

Technical Features

- **NMEA 2000 Compliance**: Implements the specification for PGN 129026 exactly.
- **Resolution**: The physical values for heading and speed are encoded in the fixed resolutions defined in the NMEA standard (COG: 0.0001 rad/LSB, SOG: 0.01 m/s/LSB). Conversion to more common units (degrees, knots) must be performed in subsequent blocks, if necessary.
- **Sequencing**: The `SID` supports the correlation of data sent simultaneously in different PGNs.

Status Overview

1. **Inactive**: Before initialization.
2. **Initialized/Pending**: After successful `INIT`. The module listens on the CAN bus or the corresponding interface on PGN 129026.
3. **Data Reception**: Upon arrival of a frame, the data is processed and a `IND` event is generated.
4. **Timeout**: Activates if expected data does not arrive within the specified time. Triggers a `TIMEOUT` event.

Application Scenarios

- **Maritime Navigation**: Displays current heading and speed on a multifunction display (MFD).
- **Autonomous Control**: Provides basic navigation data for autopilots or routing algorithms of construction machinery.

- **Data Logging**: Logs vehicle movement data with high temporal resolution.
- – **Sensor Fusion**: Combining COG/SOG data with other position and motion sensors (e.g., GNSS, gyroscope) to improve overall accuracy.

⚖️ Comparison with Similar Components

- **Compared to Generic CAN Receiver Blocks**: `I_COGSOGRapidUpdate` is specialized for PGN 129026. It handles the complete decoding of the raw data according to the NMEA specification and provides the physical values directly. A generic receiver would only deliver the raw bytes.
- **Compared to PGN 129025 (COG/SOG)**: PGN 129026 is the "Rapid Update" variant, optimized for a higher update rate and lower latency, while PGN 129025 can contain additional fields such as timestamps. The choice of function block depends on the application's requirements for data timeliness and scope.

🛠️ Related Exercises

- Exercise_079

Conclusion

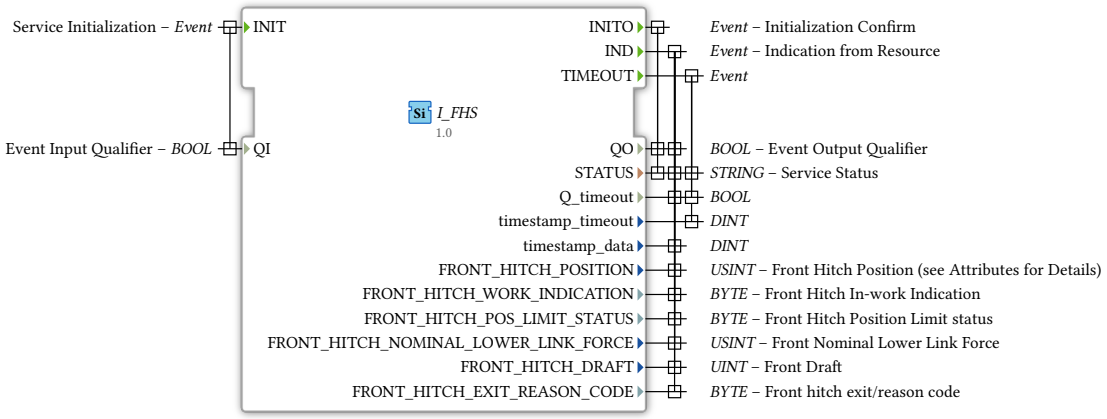
The `I_COGSOGRapidUpdate` function block is an essential component for any ISOBUS- or NMEA 2000-based system that relies on precise and rapidly updated course and speed information over ground. Its standards-compliant implementation ensures reliable data exchange and easy integration into higher-level control and display systems.

I_FHS

• – ◦ • – ◦ • – ◦ •

Introduction

The **I_FHS** is a standards-compliant function block for front-mounted implement status information according to ISO 11783-7 (PGN 65094). Developed under the EPL-2.0 license, it enables the monitoring and control of front-mounted implements in modern agricultural systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request

Event Outputs

- `INITO`: Initialization Acknowledgement
- `IND`: Status Indication
- `TIMEOUT`: Timeout Event

Data Inputs

- `QI` (BOOL): Initialization Qualifier

Data Outputs

- `QO` (BOOL): Status Qualifier
- `STATUS` (STRING): Operational Status
- `timestamp_data` (DINT): Timestamp of the data (with `IND`)
- `timestamp_timeout` (DINT): Timestamp of the timeout (with `TIMEOUT`)
- `Q_timeout` (BOOL): Timeout Indicator

Parameters (SPN Details)

Parameter	SPN	Type	Length	Scale	Value Range	Description
FRONT_HITCH_POSITION	1872	USINT	8 bits	0.4%/bit	0-100%	Current Position of Front Attachment
FRONT_HITCH_WORK_INDICATION	1876	BYTE	2 bits	4 States	0-3	Working Status Indicator
FRONT_HITCH_POS_LIMIT_STATUS	5158	BYTE	3 bits	8 states	0-7	Position limit status
FRONT_HITCH_NOMINAL_LOWER_LINK_FORCE	1873	USINT	8 bits	0.8%/bit	−100% to +100%	Normal lower link force
FRONT_HITCH_DRAFT	1878	UINT	16 bits	10 N/bit	−320 kN to +320 kN	Pulling force at front implement
FRONT_HITCH_EXIT_REASON_CODE	5016	BYTE	6 bits	64 states	0-63	Basic code for system shutdown

Functionality

1. **Initialization**:

- Activation via `INIT` event
- Confirmation via `INITO` with status information

1. **Data Update**:

- Automatic status messages via `IND` event
- Contains all relevant front-mounted implement parameters

1. **Error Handling**:

- Timeout detection via `TIMEOUT` event
- Detailed status codes in the `STATUS` field

Technical Features

✓ **ISO 11783-7 compliant** (PGN 65094)
✓ **Real-time** status monitoring ✓ **Comprehensive parameters** for front-mounted implements ✓ **Integrated fault diagnostics**

Application Scenarios

- **Front Loader Control**: Precise Position Control
- **Force Measurement**: Real-time Monitoring of Traction Forces
- **Safety Systems**: Detection of Limit Exceedances
- **Diagnostics**: Systematic Fault Analysis

⚖️ **Comparison with Similar Modules**

Feature	I_FHS	Standard Hitch Control	Advanced Hitch Manager
ISO Standard	✓	✖	✖
Front Mounting	✓	✖	✓
Force Measurement	✓	✓	✓

Diagnostic Codes	✓	✖	✓

🛠️ Related Exercises

- Exercise_079

Conclusion

The I_FHS module offers comprehensive front implement control:

- **Standard-compliant**: Full ISO 11783-7 compatibility
- **Precise**: High-resolution measurement parameters
- **Reliable**: Integrated fault detection

Essential for:

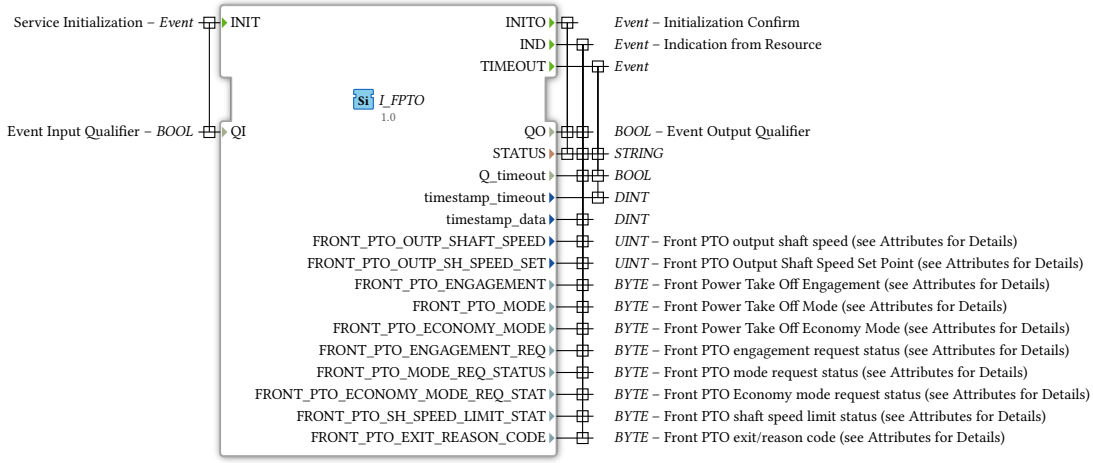
- Modern front loader systems
- Precision agriculture
- Safety-critical applications

I_FPTO

• – ◦ • – ◦ • – ◦ •

Introduction

The **I_FPTO** is a standards-compliant function block for controlling and monitoring the front power take-off (FPTO) output shaft, developed under the EPL 2.0 license. Version 1.0 implements the ISO 11783-7 specification (PGN 65092) for measuring and controlling FPTO parameters.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with qualifier `QI`)

Event Outputs

- `INITO`: Initialization Confirmation (with status)
- `IND`: Indication Event with all FPTO parameters
- `TIMEOUT`: Timeout Event

Data Inputs

- `QI` (BOOL): Qualifier for Initialization

Data Outputs

- `QO` (BOOL): Qualifier for Output Events
- `STATUS` (STRING): Operating status message
- `Q_timeout` (BOOL): Timeout indicator
- `timestamp_timeout` (DINT): Timeout timestamp

- `timestamp_data` (DINT): Data timestamp

FPTO Parameters

Parameter	Type	Description	SPN	Bit length	Scaling
FRONT_PTO_OUTP_SHAFT_SPEED	UINT	Current FPTO shaft speed	1882	16	0.125 rpm/bit
FRONT_PTO_OUTP_SH_SPEED_SET	UINT	Target speed of FPTO shaft	1884	16	0.125 rpm/bit
FRONT_PTO_ENGAGEMENT	BYTE	FPTO coupling status	1888	2	4 states/2 bits
FRONT_PTO_MODE	BYTE	FPTO operating mode	1889	2	4 states/2 bits
FRONT_PTO_ECONOMY_MODE	BYTE	FPTO economy mode	1891	2	4 states/2 bits
FRONT_PTO_ENGAGEMENT_REQ	BYTE	Coupling request status	5152	2	4 states/2 bits
FRONT_PTO_MODE_REQ_STATUS	BYTE	Mode Request State	5153	2	4 states/2 bits
FRONT_PTO_ECONOMY_MODE_REQ_STAT	BYTE	Economy Mode Request State	5154	2	4 states/2 bits
FRONT_PTO_SH_SPEED_LIMIT_STAT	BYTE	Speed Limit State	5155	3	8 states/3 bits
FRONT_PTO_EXIT_REASON_CODE	BYTE	FPTO Failure Base Code	5817	6	64 states/6 bits

Functionality

1. **Initialization**:

- `INIT` with `QI`=TRUE starts initialization
- `INITO` confirms operational readiness with `QO` and `STATUS`

1. **Data Provision**:

- `IND` provides all FPTO parameters with timestamps
- Automatic updates upon state changes

1. **Error Handling**:

- `TIMEOUT` in case of communication problems
- Detailed status messages in the `STATUS` field

Technical Features

✓ **ISO 11783-7 compliant** (PGN 65092)
✓ **16-bit Speed measurement** with 0.125 rpm resolution ✓ **State control** for clutch and operating modes
✓ **Diagnostic functions** with basic codes

Application scenarios

- **Agricultural machinery**: Control of front PTOs on tractors
- **Speed control**: Precise speed control
- **Condition monitoring**: Real-time diagnostics of the FPTO system
- **Energy efficiency**: Economy mode control

⚖️ **Comparison with similar components**

Feature	I_FPTO	Standard_PTO	Advanced_PTO
ISO standard	✓ (ISO 11783-7)	✓	✖
Front PTO	✓	✖	✓
Economy Mode	✓	✖	✓

Diagnostic Codes	✓	✖	✓

🛠️ Related Exercises

- Exercise_079

Conclusion

The I_FPTO module offers the standard implementation for front PTO systems:

- **Precise**: High-resolution speed measurement
- **Comprehensive**: Complete condition monitoring
- **Robust**: Integrated fault diagnostics

Ideal for use in:

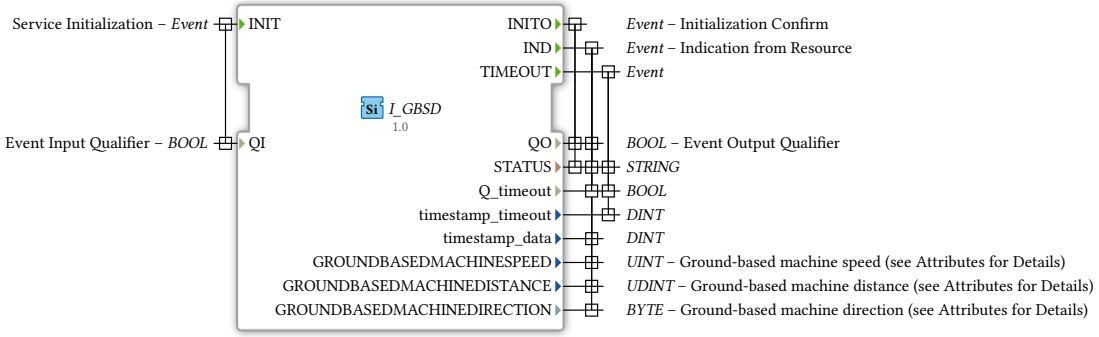
- Front PTO systems in agricultural machinery
- Applications with high demands on speed accuracy
- Systems with advanced diagnostic requirements

I_GBSD

• – ◦ • – ◦ • – ◦ •

Introduction

The **I_GBSD** (Ground-based Speed and Distance) is a standards-compliant function block for measuring vehicle speed, distance, and direction of travel, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-7 specification (PGN 65097) for precise motion data in agricultural and construction machinery.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with qualifier `QI`)

Event Outputs

- `INITO`: Initialization Acknowledgement (with status)
- `IND`: Data Indication with Speed, Distance, and Direction
- `TIMEOUT`: Timeout Event

Data Inputs

- `QI` (BOOL): Qualifier for Initialization

Data Outputs

- `QO` (BOOL): Qualifier for Output Events
- `STATUS` (STRING): Operational status message
- `Q_timeout` (BOOL): Timeout indicator
- `timestamp_timeout` (DINT): Timeout timestamp
- `timestamp_data` (DINT): Movement data timestamp

Movement data parameters

Parameter	Type	Description	SPN	Bit length	Scale	Accuracy
GROUNDBASEDMACHINESPEED	UINT	Machine speed	1859	16	0.001 m/s per bit	±0.1% at >1 m/s
GROUNDBASEDMACHINEDISTANCE	UDINT	Distance traveled	1860	32	0.001 m per bit	±0.5% cumulative
GROUNDBASEDMACHINEDIRECTION	BYTE	Direction of travel	1861	2	4 states/2 bits	-

Direction of travel states

Value	State	Description
0	Stationary	No movement detected
1	Forward	Moving forward
2	Reverse	Moving backward
3	Undefined	Direction cannot be determined

Functionality

1. **Initialization**:

- `INIT` with `QI`=TRUE starts sensor calibration
- `INITO` confirms operational readiness with system status

1. **Data Update**:

- `IND` delivers all motion data with a 100ms update rate
- Distance measurement as a free-running counter (32-bit)

1. **Special Operating Modes**:

- Automatic accuracy adjustment at low speeds (<0.5 m/s)
- Integrierte Plausibilitätsprüfung der Sensordaten

Technische Besonderheiten

✓ **ISO 11783-7 konform** (PGN 65097) ✓ **Hochpräzise Messung** mit 1mm Auflösung ✓ **32-bit Distanzzähler** (bis zu 4,294,967km Reichweite) ✓ **Robuste Richtungserkennung** mit 4 Zuständen

Anwendungsszenarien

- **Präzisionslandwirtschaft**: Geschwindigkeitskontrolle für Saat- und Düngemittel
- **Flächenberechnung**: Automatische Arbeitsflächenmessung
- **Fahrassistenz**: Richtungserkennung bei Rückwärtsfahrt
- **Telematik**: Betriebsdatenerfassung für Maschinenmanagement

Genauigkeitsmerkmale

Geschwindigkeitsbereich	Typische Genauigkeit	Update-Rate
> 2 m/s (7.2 km/h)	±0.1%	100 ms
0.5 - 2 m/s	±1%	200 ms
< 0.5 m/s	±5%	500 ms

⚖️ **Comparison with similar systems**

Feature	I_GBSD	Wheel-based	GPS-based

Ground contact	✓ Direct	✓ Indirect	✖
Low speed	✓ Good	✖ Inaccurate	✖ Inaccurate
Direction detection	✓ Precise	✓	✖ Ambiguous
Signal loss	Robust	Susceptible	Prone to interference

🛠️ Related exercises

- Exercise_072
- Exercise_072b
- Exercise_072c
- Exercise_073
- Exercise_079

Conclusion

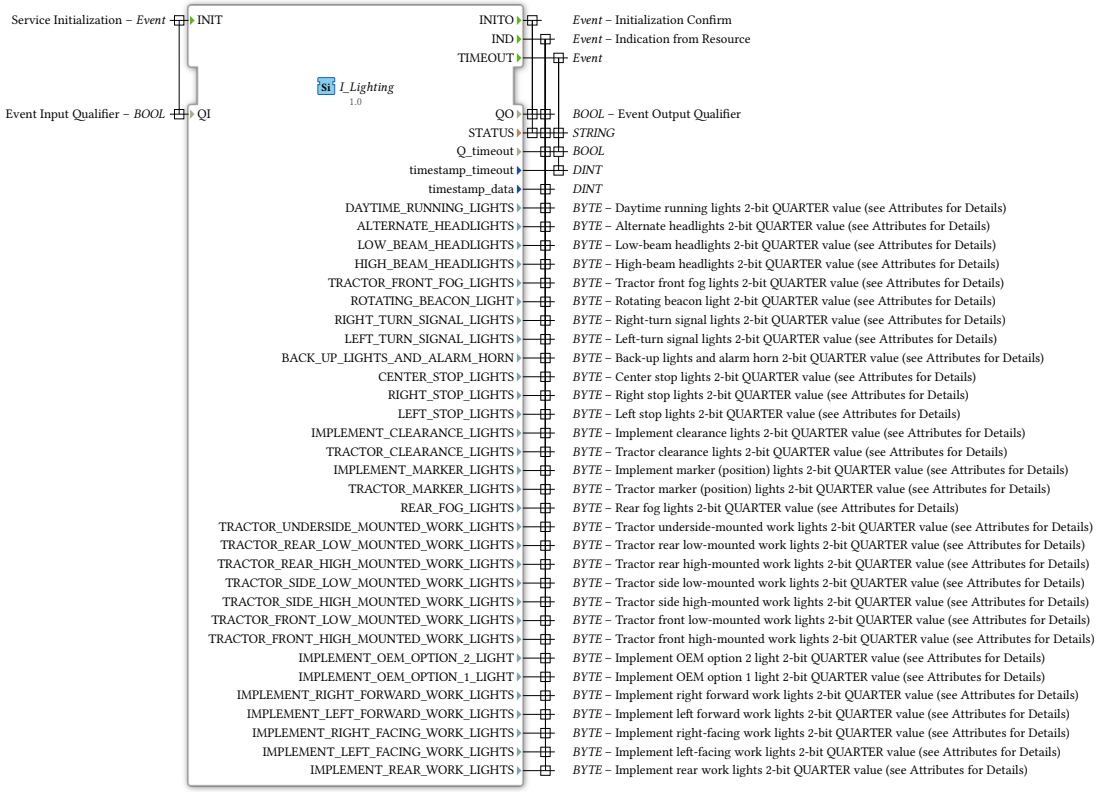
The I_GBSD module provides reliable motion data for mobile machinery:

- **Precision**: Submillimeter-accurate distance measurement
- **Reliability**: Functionality even under poor GPS conditions
- **Flexibility**: Universal application in agriculture and construction machinery

Ideal for:

- Automatic steering systems
- Working width calculations
- Machines with high accuracy requirements
- Applications with frequent changes of direction

I_Lighting



• — ○ • — ○ • — ○ •

Introduction

The **I_Lighting** function block processes the lighting data of an agricultural vehicle according to ISO 11783-7 (ISOBUS). It receives and decodes the parameter group number (PGN) 65088, which transmits the status of all lighting functions of a tractor and connected implements. The block serves as an interface between the ISOBUS network and the application logic for monitoring and controlling the lighting.

Interface Structure

Event Inputs

Event	Type	Description	Data Carried
INIT	EInit	Initializes the block and activates processing.	QI

Event Outputs

Event	Type	Description	Data Carried
INITO	EInit	Confirms successful initialization.	QO, STATUS
IND	Event	Signals new lighting data from the bus.	QO, timestamp_data, STATUS, Q_timeout, and all 32 luminaire status outputs
TIMEOUT	Event	Triggered when the expected data is not received (timeout).	timestamp_timeout, STATUS, Q_timeout

Data Inputs

Name	Data Type	Description
QI	BOOL	Activation signal for initialization; The function block is started when TRUE.

Data Outputs

Name	Data Type	Description
QO	BOOL	Acknowledgement signal after successful initialization or data processing.
STATUS	STRING	Status message (e.g., error messages or operating instructions).
Q_timeout	BOOL	Signals whether a timeout has occurred (TRUE = Timeout active).
timestamp_timeout	DINT	Timestamp of the last timeout event.
timestamp_data	DINT	Timestamp of the last received lighting data.
32 Lighting Outputs	BYTE	One 2-bit value (QUARTER) for each lighting function. Initial value: 0xFF. - DAYTIME_RUNNING_LIGHTS - ALTERNATE_HEADLIGHTS - LOW_BEAM_HEADLIGHTS - HIGH_BEAM_HEADLIGHTS – TRACTOR_FRONT_FOG_LIGHTS – ROTATING_BEACON_LIGHT – RIGHT_TURN_SIGNAL_LIGHTS – LEFT_TURN_SIGNAL_LIGHTS – BACK_UP_LIGHTS_AND_ALARM_HORN – CENTER_STOP_LIGHTS – RIGHT_STOP_LIGHTS – LEFT_STOP_LIGHTS – IMPLEMENT_CLEARANCE_LIGHTS – TRACTOR_CLEARANCE_LIGHTS – IMPLEMENT_MARKER_LIGHTS – TRACTOR_MARKER_LIGHTS – REAR_FOG_LIGHTS – TRACTOR_UNDERSIDE_MOUNTED_WORK_LIGHTS

		– TRACTOR_REAR_LOW_MOUNTED_WORK_LIGHTS – TRACTOR_REAR_HIGH_MOUNTED_WORK_LIGHTS – TRACTOR_SIDE_LOW_MOUNTED_WORK_LIGHTS – TRACTOR_SIDE_HIGH_MOUNTED_WORK_LIGHTS – TRACTOR_FRONT_LOW_MOUNTED_WORK_LIGHTS – TRACTOR_FRONT_HIGH_MOUNTED_WORK_LIGHTS – IMPLEMENT_OEM_OPTION_2_LIGHT – IMPLEMENT_OEM_OPTION_1_LIGHT – IMPLEMENT_RIGHT_FORWARD_WORK_LIGHTS – IMPLEMENT_LEFT_FORWARD_WORK_LIGHTS – IMPLEMENT_RIGHT_FACING_WORK_LIGHTS – IMPLEMENT_LEFT_FACING_WORK_LIGHTS – IMPLEMENT_REAR_WORK_LIGHTS

Adapter

None.

Functionality

The **I_Lighting** block is activated by the event `INIT`. This prepares the internal processing. After successful initialization, it acknowledges with `INITO`. The function block then waits for incoming ISOBUS messages (PGN 65088). When a valid message arrives, the event `IND` is triggered, and all lighting outputs are updated with the decoded 2-bit values (QUARTER). Simultaneously, a timestamp is stored in `timestamp_data`. If no new data arrives within a configured time period, the event `TIMEOUT` is triggered, and `Q_timeout` is set to TRUE. The function block thus ensures reliable monitoring of the lighting status.

Technical Features

- **ISO 11783-7 Compliance:** The function block implements the parameter group PGN 65088 “Lighting Data LD” according to the ISOBUS standard.
- **2-Bit QUARTER Encoding:** Each light output encodes four states in two bits – typical interpretation: 0 = off, 1 = on, 2 = error, 3 = unavailable. The initial value `16#FF` (decimal 255) corresponds to the “unavailable” state.
- **SPN Attributes:** Each output is provided with detailed metadata (SPN, position in the data telegram, scaling, link to the specification), which facilitates traceability and configuration.
- **Timeout Detection:** The module can detect and report the absence of bus messages, e.g., for error handling or redundant behavior.
- **Timestamps:** Both data arrival and timeouts are timestamped (DINT), enabling analysis over time.

State Overview

The function block does not have an explicit, published state machine. However, its logical behavior can be described as follows:

- **Initialization:** After `INIT`, the function block enters the active state, acknowledged with `INITO`.
- **Ready to Receive:** It waits for incoming messages. Upon receipt ⟶ `IND`, it updates all outputs and resets `Q_timeout`.
- **Timeout Monitoring:** If an internal timer expires without new data arriving, the function block briefly enters the timeout state and signals this via `TIMEOUT`. It then returns to the ready-to-receive state.
- **Error handling:** If an error occurs during decoding, `STATUS`Set accordingly.

Application Scenarios

- **Tractor Lighting Control:** Integration into a vehicle control unit (ECU) for monitoring and displaying the lighting status on the terminal.
- **ISOBUS-Compliant Implements:** Use in implements that report their own lighting via the ISOBUS network (e.g., front loaders, rear-mounted implements).
- **Diagnostics and Troubleshooting:** Reading the lighting status and timeouts for fault analysis or logging in the service workshop.
- **Light Control in Automated Systems:** Combination with other modules for automatically adjusting the lighting to ambient conditions (e.g., day/night detection).

Comparison with Similar Modules

Similar ISOBUS modules exist for other parameter groups, e.g., **I_Engine** (engine data), **I_Steering** (steering), or **I_WorkState** (working state). In comparison to these:

- **Specialization: I_Lighting** is focused on lighting and offers a high number of 32 specific lighting outputs.
- **Data Width:** The outputs are BYTE (2 bits used), while other function blocks often use WORD or DWORD.
- **Timeout Handling:** Not all function blocks implement their own timeout detection – here it is explicitly provided.
- **Initial Value:** The outputs start with 0xFF ("not available"), which enables robust initialization without validity conflicts.

Conclusion

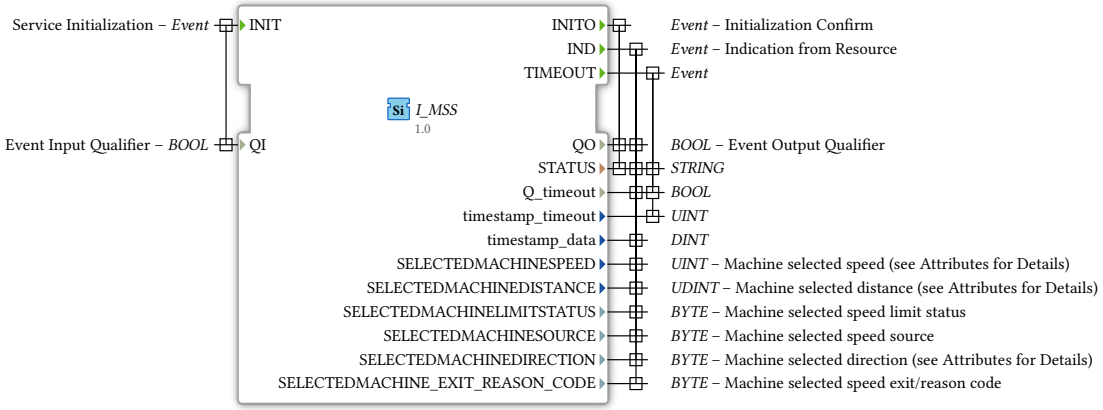
The **I_Lighting** function block is a specialized, standards-compliant function block for processing ISOBUS lighting data (PGN 65088). It decodes the status of 32 different lighting functions from the CAN bus and offers reliable timeout monitoring. Thanks to detailed SPN metadata and a simple interface, it is ideally suited for integration into agricultural control systems that require precise and standards-compliant light control.

I_MSS

• – ◦ • – ◦ • – ◦ •

Introduction

The **I_MSS** (Machine Selected Speed) is a standards-compliant function block for controlling and monitoring the machine-selected speed, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-7 specification (PGN 65097) for precise speed control in agricultural and construction machinery.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with qualifier `QI`)

Event Outputs

- `INITO`: Initialization Confirmation (with status)
- `IND`: Indication Event with all speed parameters
- `TIMEOUT`: Timeout Event

Data Inputs

- `QI` (BOOL): Qualifier for Initialization

Data Outputs

- `QO` (BOOL): Qualifier for Output Events
- `STATUS` (STRING): Operating status message
- `Q_timeout` (BOOL): Timeout indicator
- `timestamp_timeout` (UINT): Timeout timestamp

- `timestamp_data` (DINT): Speed data timestamp

Speed parameters

Parameter	Type	Description	SPN	Bit length	Scaling
SELECTEDMACHINESPEED	UINT	Machine speed	4305	16	0.001 m/s/bit (LSB), 0.256 m/s/bit (MSB)
SELECTEDMACHINEDISTANCE	UDINT	Distance Traveled	4306	32	0.001 m/bit
SELECTEDMACHINELIMITSTATUS	BYTE	Speed Limit Status	4307	3	8 states/3 bits
SELECTEDMACHINESOURCE	BYTE	Speed Source	4308	3	8 states/3 bits
SELECTEDMACHINEDIRECTION	BYTE	Direction of Travel	4309	2	4 states/2 bits
SELECTEDMACHINE_EXIT_REASON_CODE	BYTE	Speed Failure Root Code	5818	6	64 States/6 bits

Direction of Travel States

Value	State	Description
0	Stationary	No movement detected
1	Forward	Moving forward
2	Reverse	Moving backward
3	Undefined	Direction cannot be determined

Functionality

1. **Initialization**:

- `INIT` with `QI`=TRUE starts system initialization
- `INITO` confirms operational readiness with `QO` and `STATUS`

1. **Data Provision**:

- `IND` provides all speed parameters with timestamps
- Automatic updates upon state changes

1. **Error Handling**:

- `TIMEOUT` in case of communication problems
- Detailed error codes in `SELECTEDMACHINE_EXIT_REASON_CODE`

Technical Features

✓ **ISO 11783-7 compliant** (PGN 65097)
✓ **Dual Speed Scaling** for High Precision ✓ **Multi-Source Support** (8 possible speed sources) ✓ **Intelligent Limit Detection** with 3-stage status

Application Scenarios

- **Tractors**: Automatic speed adjustment for field work
- **Harvesting Machines**: Precise row length calculation
- **Construction Machinery**: Speed limitation in hazardous areas
- **Fleet Management**: Operational data collection and analysis

Speed Sources

Code	Source	Typical Application
0	Undefined	System startup
1	Wheel sensor	Standard operation

2	Soil sensor	Precision agriculture
3	GPS	Monitoring
4-7	Reserved	Manufacturer-specific

⚖️ Comparison with similar systems

Feature	I_MSS	Standard	GPS-based
Accuracy	±0.2%	±1-2%	±5-10%
Response Time	<100ms	200ms	1-2s
Source Flexibility	8	1-2	1
Low Speed	✓ Optimal	✖ Inaccurate	✖ Unusable

🛠️ Related Exercises

- Exercise_079

Conclusion

The I_MSS module offers precise speed control for mobile machinery:

- **Flexible**: Supports multiple speed sources
- **Precise**: Dual scaling for high accuracy
- **Robust**: Integrated fault diagnostics

Ideal for use in:

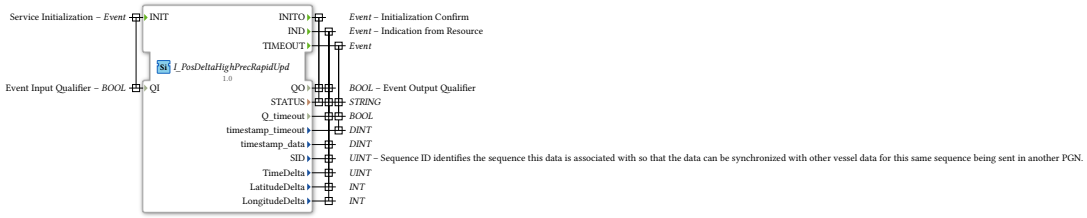
- Automatic steering systems
- Precision agriculture
- Machines with high speed requirements
- Applications with varying operating conditions

I_PosDeltaHighPrecRapidUpd

• – ○ • – ○ • – ○ •

Introduction

The function block `I_PosDeltaHighPrecRapidUpd` implements the processing of the NMEA 2000 Parameter Group Number (PGN) 129027 “Position Delta, High Precision Rapid Update”. This block is designed for applications requiring very high precision and very fast update rates for position data. It can provide position changes (delta) with a resolution of up to 1 millimeter and a delta time interval with an accuracy of 5 milliseconds.



Interface Structure

Event Inputs

- **INIT**: Service initialization. Triggered together with the qualifier `QI` to initialize or deactivate the function block.

Event Outputs

- **INITO**: Initialization confirmation. Triggered in response to the `INIT` event, it returns the current status (`STATUS`) and the qualifier `QO`.

- **IND**: Indication from the resource manager. Triggered when new position data is available. Transmits all relevant delta position data and status information.
- **TIMEOUT**: Triggered when a timeout occurs, for example, if no new data is received within an expected timeframe.

Data Inputs

- **QI** (BOOL): Event Input Qualifier. Controls the initialization. The function block is activated on `TRUE` and deactivated on `FALSE`.

Data Outputs

- **QO** (BOOL): Event Output Qualifier. Indicates the current operating state of the function block (e.g., active/inactive).
- **STATUS** (STRING): Contains status or error messages of the function block.
- **Q_timeout** (BOOL): Indicates whether the last received data is outdated due to a timeout.
- **timestamp_timeout** (DINT): Timestamp associated with the `TIMEOUT` event.
- **timestamp_data** (DINT): Timestamp of the last received valid position data.
- **SID** (UINT): Sequence ID. Identifies the sequence to which this data is assigned to enable synchronization with other vehicle data of the same sequence in another PGN.
- **TimeDelta** (UINT): The time difference in milliseconds over which the position change was measured.
- **LatitudeDelta** (INT): The change in latitude (delta) at high-precision resolution (1/1e-7 min = 1.85 mm).
- **LongitudeDelta** (INT): The change in longitude (delta) at high-precision resolution (1/1e-7 min = 1.85 mm).

Adapter

This function block does not use any adapter interfaces.

Operation

The function block is started or stopped via the `INIT` event with the corresponding `QI` value. After successful initialization, it confirms this with `INITO`. In the active state, the function block listens for incoming PGN 129027 messages from the NMEA 2000 network.

Upon receiving new, valid data, it triggers the `IND` event and makes all calculated delta values (time, latitude, longitude) available at its outputs, along with the sequence ID (`SID`) and a data timestamp. The `STATUS` output can provide additional information.

If no new message is received for a defined period, the function block can generate a `TIMEOUT` event. This indicates that the currently held data may be outdated (`Q_timeout` = TRUE).

Technical Features

- **High Precision**: Processes positional deltas with millimeter resolution (1/1e-7 arcminutes).
- **Fast Update**: Supports very short update intervals, represented by `TimeDelta` (minimum 5 ms accuracy).
- **Sequencing**: `SID` (Sequence ID) enables the correct assignment and synchronization of position deltas with other time-critical data streams in the system.
- **Timeout Detection**: Integrated monitoring of data freshness to detect communication failures or delayed messages.

State Overview

The function block essentially has two main states:

1. **Inactive / Initialized**: The block is initialized but not active (`QI` = FALSE). No data is processed.
2. **Active / Listening**: The module is activated (`QI` = TRUE) and is waiting for incoming PGN 129027 messages. Upon receipt, these are processed and output via `IND`. A missing signal can lead to a timeout state, which is indicated by the `TIMEOUT` event.

Application Scenarios

- **Precision Agriculture**: For high-precision guidance systems and automatic steering systems that require real-time position corrections.
- **Dynamic Positioning**: In maritime applications for the precise attitude control of ships or offshore platforms.
- **Autonomous Vehicles**: For high-frequency odometry and path correction in real time.

- **Surveying and Mapping**: For the acquisition of terrain data with very high spatial and temporal resolution.

⚖️ Comparison with Similar Function Blocks

Compared to standard position function blocks (e.g., those that process PGN 129029 "Position, Rapid Update"), `I_PosDeltaHighPrecRapidUpd` offers:

- **Delta-based data**: Instead of absolute positions, changes (deltas) are transmitted, which can be more bandwidth-efficient at high update rates.
- **Higher precision**: Specified for applications requiring higher accuracy than typical rapid update PGNs.
- **Explicit time difference**: The `TimeDelta` field is an integral part of the message, enabling more precise velocity and acceleration calculations.

Conclusion

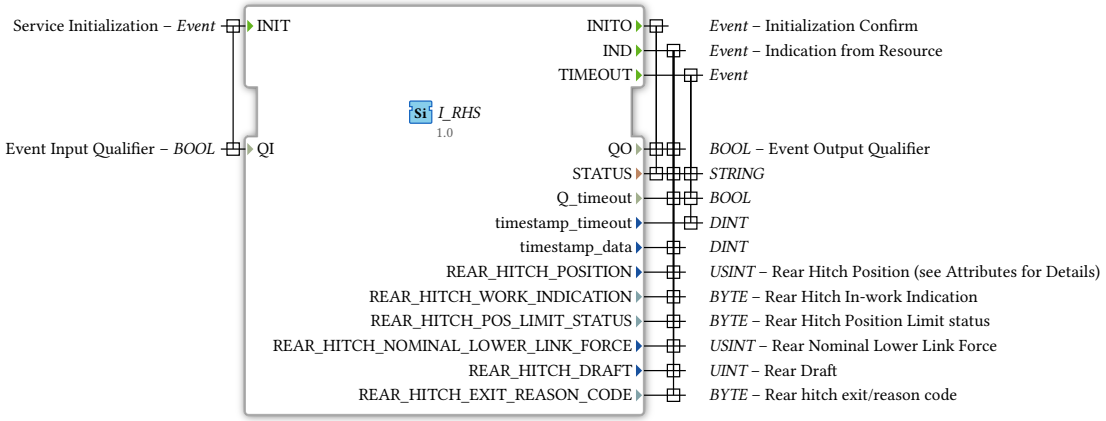
The `I_PosDeltaHighPrecRapidUpd` function block is a specialized tool for demanding real-time position determination. Its ability to deliver and synchronize highly precise position changes at very high frequencies makes it ideal for precision-critical control tasks in agricultural, maritime, and autonomous systems. The integrated timeout detection further enhances the application's robustness.

I_RHS

• – ◦ • – ◦ • – ◦ •

Introduction

The **I_RHS** (Rear Hitch Status) is a standards-compliant function block for monitoring rear hitch parameters, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-7 specification (PGN 65093) for the precise acquisition of the position, force, and status of the rear hitch in agricultural machinery.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with qualifier `QI`)

Event Outputs

- `INITO`: Initialization Confirmation (with status)
- `IND`: Indication Event with all coupling parameters
- `TIMEOUT`: Timeout Event

Data Inputs

- `QI` (BOOL): Qualifier for Initialization

Data Outputs

- `QO` (BOOL): Qualifier for Output Events
- `STATUS` (STRING): Operating status message
- `Q_timeout` (BOOL): Timeout indicator

- `timestamp_timeout` (DINT): Timeout timestamp
- `timestamp_data` (DINT): Coupling data timestamp

Coupling parameters

Parameter	Type	Description	SPN	Bit length	Scaling
`REAR_HITCH_POSITION`	USINT	Coupling position	1873	8	0.4 %/bit
`REAR_HITCH_WORK_INDICATION`	BYTE	Operating State	1877	2	4 states/2 bits
`REAR_HITCH_POS_LIMIT_STATUS`	BYTE	Position Limitation	5151	3	8 states/3 bits
`REAR_HITCH_NOMINAL_LOWER_LINK_FORCE`	USINT	Lower Link Force	1881	8	0.8%/bit (-100% Offset)
`REAR_HITCH_DRAFT`	UINT	Tensile Force	1879	16	10 N/bit (-320kN Offset)
`REAR_HITCH_EXIT_REASON_CODE`	BYTE	Fault Reason Code	5819	6	64 states/6 bits

Functionality

1. **Initialization**:

- `INIT` with `QI`=TRUE starts system calibration
- `INITO` confirms operational readiness with `QO` and `STATUS`

1. **Data Provision**:

- `IND` provides all coupling parameters with timestamps
- Automatic updates upon state changes

1. **Error Handling**:

- `TIMEOUT` in case of communication problems
- Detailed error codes in `REAR_HITCH_EXIT_REASON_CODE`

Technical Features

✓ **ISO 11783-7 compliant** (PGN 65093)
✓ **Bidirectional force measurement** (+/- 100% range)
✓ **Precise position detection** with 0.4% resolution
✓ **64-level fault diagnosis** with detailed codes

Coupling characteristics

Feature	Description
Position range	0-100% (0 = fully down)
Force measurement	±100% of rated load
Tensile force range	−320kN to +350kN
Update rate	100ms in normal operation

Return codes (REAR_HITCH_EXIT_REASON_CODE)

Code range	Meaning
0-15	System error
16-31	Position Error
32-47	Force Measurement Error
48-63	Reserved

Application Scenarios

- **Plow Control**: Automatic Depth Control
- **Load Management**: Traction Force Adjustment
- **Diagnostics**: Early Detection of Hydraulic Problems
- **Safety**: Coupling Position Monitoring

⚖️ Comparison with Similar Components

Feature	I_RHS	Standard	Premium
Accuracy	±0.4%	±2%	±0.2%
Force Measurement	Bidirectional	Traction Only	Triaxial
Diagnostic Codes	64	8	128
ISO Compliance	Full	Partial	Full

🛠️ Related Exercises

- Exercise_079

Conclusion

The I_RHS module offers comprehensive monitoring for rear hitch systems:

- **Precise**: High-resolution position and force measurement
- **Reliable**: Robust fault detection
- **Flexible**: Suitable for various implements

Ideal for use with:

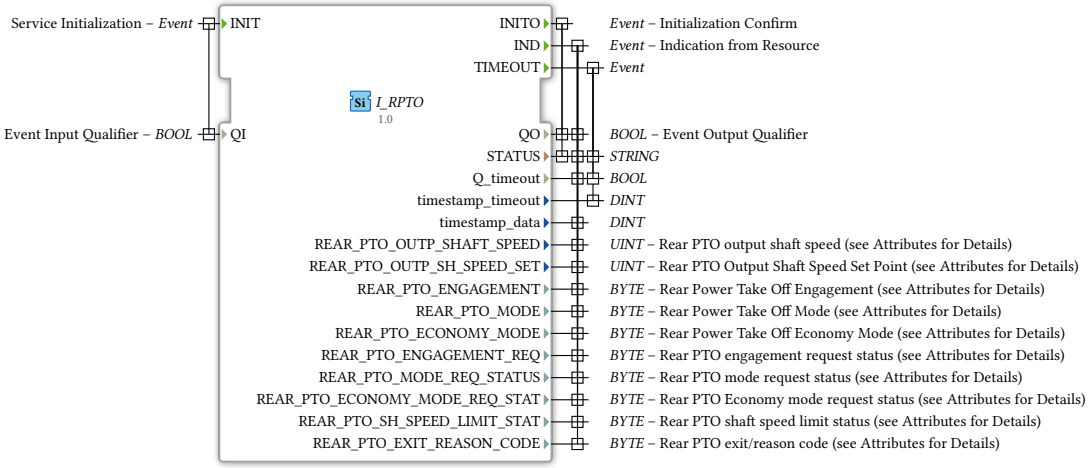
- Modern tractors
- Precision agriculture
- Heavy implements
- Automatic steering systems

I_RPTO

• – ○ • – ○ • – ○ •

Introduction

The **I_RPTO** (Rear Power Take-Off) is a standards-compliant function block for controlling and monitoring the rear power take-off (PTO), developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-7 specification (PGN 65091) for the precise control of PTO parameters in agricultural machinery.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with qualifier `QI`)

Event Outputs

- `INITO`: Initialization Confirmation (with status)
- `IND`: Indication Event with all PTO Parameters
- `TIMEOUT`: Timeout Event

Data Inputs

- `QI` (BOOL): Qualifier for Initialization

Data Outputs

- `QO` (BOOL): Qualifier for Output Events
- `STATUS` (STRING): Operating status message
- `Q_timeout` (BOOL): Timeout indicator
- `timestamp_timeout` (DINT): Timeout timestamp
- `timestamp_data` (DINT): PTO data timestamp

PTO parameters

Parameter	Type	Description	SPN	Bit length	Scaling
REAR_PTO_OUTP_SHAFT_SPEED	UINT	Current speed	1883	16	0.125 rpm/bit
REAR_PTO_OUTP_SH_SPEED_SET	UINT	Target Speed	1885	16	0.125 rpm/bit
REAR_PTO_ENGAGEMENT	BYTE	Clutch State	2408	2	4 states/2 bits
REAR_PTO_MODE	BYTE	Operating Mode	1890	2	4 states/2 bits
REAR_PTO_ECONOMY_MODE	BYTE	Economy Mode	1892	2	4 states/2 bits
REAR_PTO_ENGAGEMENT_REQ	BYTE	Clutch Request	5156	2	4 states/2 bits
REAR_PTO_MODE_REQ_STATUS	BYTE	Mode Request	5157	2	4 states/2 bits
REAR_PTO_ECONOMY_MODE_REQ_STA	BYTE	Economy Mode Request	5158	2	4 states/2 bits
REAR_PTO_SH_SPEED_LIMIT_STAT	BYTE	Speed Limit	5159	3	8 states/3 bits
REAR_PTO_EXIT_REASON_CODE	BYTE	Fault Reason Code	5820	6	64 states/6 bits

Operating Modes

Code	Mode	Description
0	Standby	Standby Mode
1	Speed Mode	Constant Speed
2	Load Mode	Load-Dependent Control
3	Reserved	Manufacturer-specific

Functionality

1. **Initialization**:

- `INIT` with `QI`=TRUE starts system calibration
- `INITO` confirms operational readiness with `QO` and `STATUS`

1. **Speed Control**:

- `IND` provides actual and target speeds
- Automatic adjustment to load changes

1. **Error Handling**:

- `TIMEOUT` in case of communication problems
- Detailed error codes in `REAR_PTO_EXIT_REASON_CODE`

Technical Features

✓ **ISO 11783-7 compliant** (PGN 65091)
✓ **Precise speed control** with 0.125 RPM Resolution ✓ **4 Operating Modes** for Different Applications
✓ **64-Level Fault Diagnostics** with Detailed Codes

Application Scenarios

- **Agricultural Machinery**: Drive for Combine Harvesters and Balers
- **Speed Control**: Precise Adaptation to Implements
- **Energy Efficiency**: Economy Mode for Part-Load Operation
- **Diagnostics**: Early Detection of Mechanical Problems

⚖️ Comparison with Similar Systems

Feature	I_RPTO	Standard	Premium
Speed Accuracy	±0.2%	±1%	±0.1%
Operating Modes	4	2	6
Diagnostic Scope	64 Codes	8 Codes	128 Codes
Response Time	<100ms	200ms	50ms

🛠️ Related Exercises

- Exercise_074
- Exercise_079

Conclusion

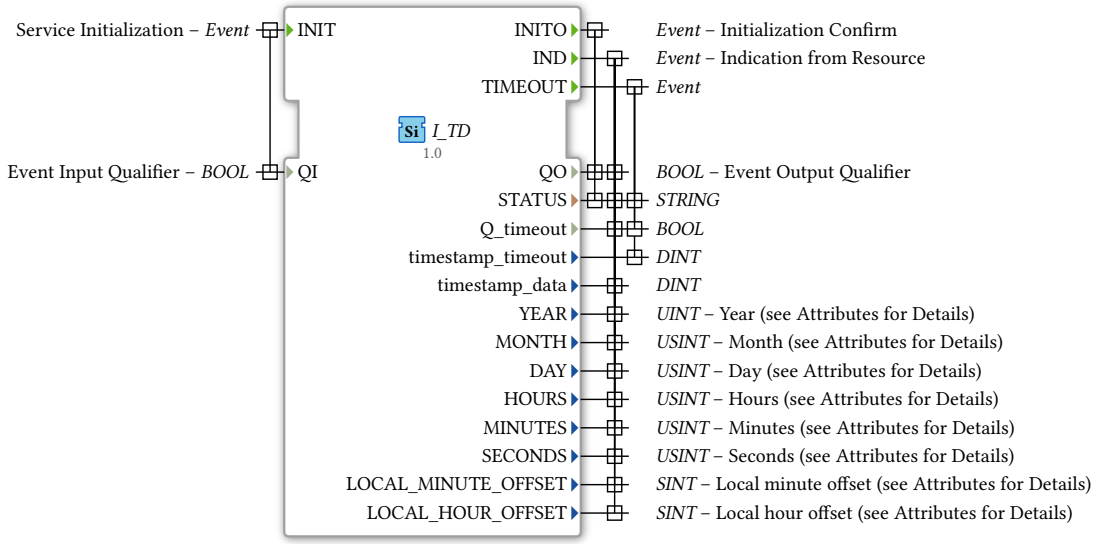
The I_RPTO module offers reliable PTO control for modern agricultural machinery:

- **Precise**: High-resolution speed control
- **Flexible**: Multiple operating modes for various applications
- **Robust**: Integrated fault detection and diagnostics

Ideal for use in:

- Tractors with demanding implements
- Precision agriculture
- Energy-efficient drive systems
- Automated work processes

I_TD



• – ◦ • – ◦ • – ◦ •

Introduction

The **I_TD** (Time/Date) function block is a special component for ISOBUS that enables the processing of time and date information according to the ISO 11783-7 standard (PGN 65254). It receives and processes the current time and date transmitted over the ISOBUS network. The output data corresponds to the defined SPNs (Suspect Parameter Numbers) of the protocol and is provided including the associated scaling and offsets.

Interface Structure

Event Inputs

Event	With Variable	Description
`INIT`	`QI`	Initialization of the function block.

Event Outputs

Event	With variables	Description
`INITO`	`QO`, `STATUS`	Confirmation of successful initialization.
`IND`	`QO`, `timestamp_data`, `STATUS`, `Q_timeout`, `SECONDS`, `MINUTES`, `HOURS`, `MONTH`, `DAY`, `YEAR`, `LOCAL_MINUTE_OFFSET`, `LOCAL_HOUR_OFFSET`	Display of a received time/date telegram.
`TIMEOUT`	`timestamp_timeout`, `STATUS`, `Q_timeout`	Timeout event if no valid telegram arrives within an expected timeframe.

Data Inputs

Variable	Type	Description
`QI`	BOOL	Qualifier for initialization (TRUE = enable).

Data Outputs

Variable	Type	Description (with ISOBUS attributes)
`QO`	BOOL	Output qualifier (TRUE = Operational).

STATUS	STRING	Status message (e.g., error or success).
Q_timeout	BOOL	Indicates whether a timeout occurred.
timestamp_timeout	DINT	Timestamp of the timeout (if it occurred).
timestamp_data	DINT	Timestamp of the received data telegram.
SECONDS	USINT	Seconds (scale 0.25s/bit, offset 0).
MINUTES	USINT	Minutes (scale 1min/bit, offset 0).
HOURS	USINT	Hours (Scale 1h/bit, Offset 0).
MONTH	USINT	Month (Scale 1 month/bit, Offset 0).
DAY	USINT	Day (Scale 0.25d/bit, Offset 0).
YEAR	UINT	Year (Scale 1y/bit, Offset 1985, Initial value 0xFFFF = "not available").
LOCAL_MINUTE_OFFSET	SINT	Local Minute Offset (Scale 1 min/bit, Offset −125).
LOCAL_HOUR_OFFSET	SINT	Local hour offset (scaling 1h/bit, offset −125).

Adapter

None.

Functionality

The function block is initialized via the event input INIT with QI set. After successful initialization, the event INITO is triggered and the output QO is set to TRUE. The function block then waits for incoming ISOBUS telegrams (PGN 65254). When such a telegram arrives, the event IND is generated and all corresponding data outputs are updated with the decoded values. The scaling and offsets defined in the attributes are applied automatically.

If no valid telegram is received within an unspecified timeframe, the event TIMEOUT is triggered and the output Q_timeout is set to TRUE. The timeout timestamp is stored in timestamp_timeout.

Technical Features

- **ISOBUS Compliance:** The FB implements exactly PGN 65254 of the ISO 11783-7 standard, part "Time/Date TD".
- **Initial Values:** Many data outputs are initialized with predefined "not available" values (e.g., YEAR = 0xFFFF), which allows for easy error detection.
- **Scaling & Offset:** All values are scaled and offset according to the ISOBUS specification. The raw values of the SPNs are converted into physical units.
- **Local Time Offset:** The outputs LOCAL_MINUTE_OFFSET and LOCAL_HOUR_OFFSET allow the specification of the deviation from UTC time in the range of −125 min to +130 min.
- **No State Machine:** The function block (FB) is event-driven; there is no explicit state machine in the XML. The internal logic is based on the IEC 61499 execution rules.

State Overview

An explicit state machine is not defined in the XML. The FB operates as a reactive block:

1. **Idle State** – waits for INIT.
2. **Initialization Phase** – After INIT, if TRUE QI, INITO is triggered.
3. **Operational Ready** – Receiving telegrams ⟶ IND; if no telegrams are received ⟶ TIMEOUT.

Application Scenarios

- **Agricultural Machinery:** Receiving current time and date data from the ISOBUS on-board network, e.g., for loggers, controllers, or displays.
- **Time Stamping:** Using the timestamp_data outputs to synchronize events in various control units.
- **Time Zone Adjustment:** Evaluating local offsets to display local time in user interfaces.

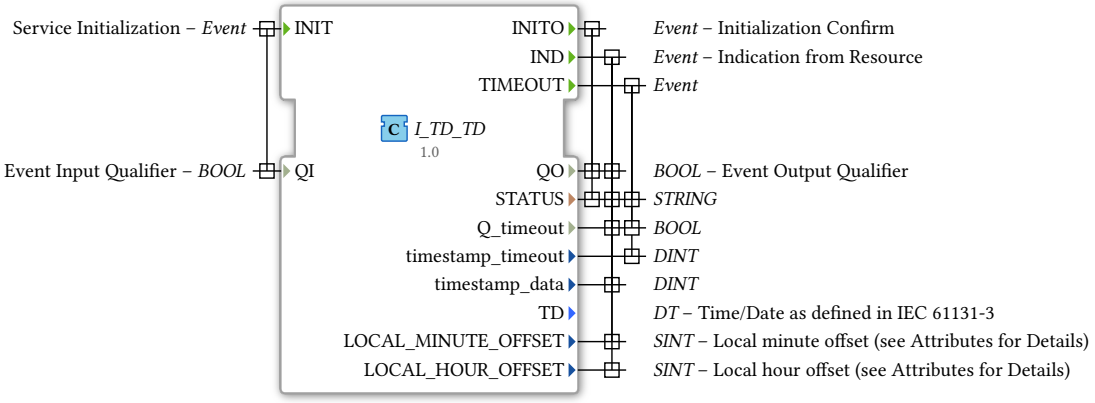
Comparison with Similar Function Blocks

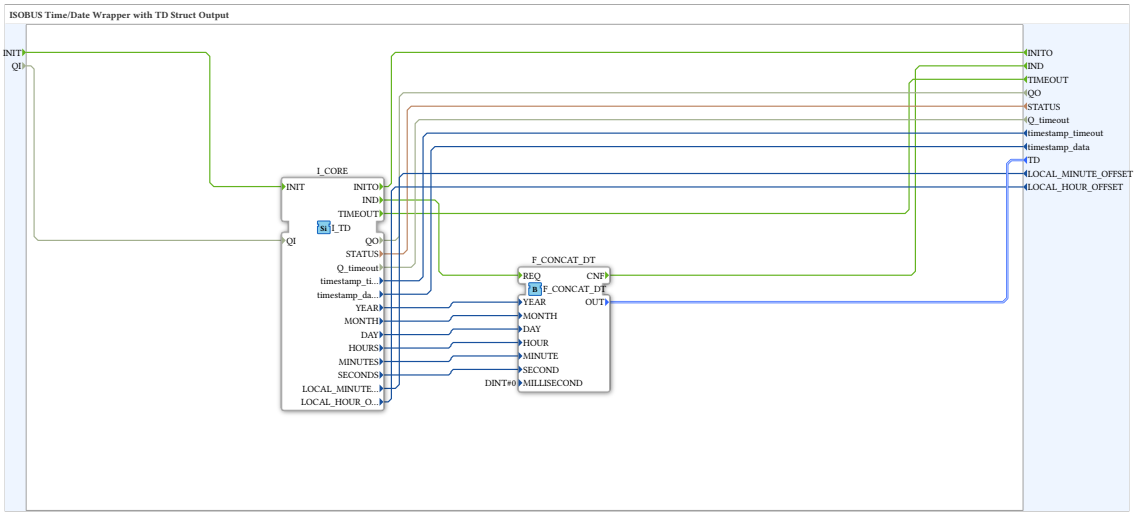
Compared to general IEC 61499 time blocks (e.g., `E_CYCLE` or `E_TIME`), `I_TD` is specifically designed for ISOBUS communication. It does not decode an internal system time but receives external telegrams and delivers the data in ISOBUS-compliant formats. A universal function block, `I_DATE` (if available), would be suitable for proprietary systems but does not offer the ISOBUS SPN structure.

Conclusion

The `I_TD` function block is an essential component for ISOBUS applications that rely on accurate time and date information from the network. Its complete mapping of SPNs and built-in timeout handling make it robust and standards-compliant. Thanks to automatic scaling and offsets, the physical values can be processed directly without manual conversion.

I_TD_TD





• – ○ • – ○ • – ○ •

Introduction

The function block **I_TD_TD** is a wrapper around the base block `I_TD` and generates a combined IEC 61131-3 date/time struct (type `DT`) from the individual time and date components. It serves as an interface to an ISOBUS-compliant time/date service (PGN 65254) and provides the processed time information as well as local time offsets as separate outputs.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Service Initialization, controlled by qualifier `QI`

Event Outputs

Name	Type	Carrying Data
INITO	EInit	QO, STATUS
IND	Event	QO, timestamp_data, STATUS, Q_timeout, LOCAL_MINUTE_OFFSET, LOCAL_HOUR_OFFSET
TIMEOUT	Event	timestamp_timeout, STATUS, Q_timeout

Data Inputs

Name	Type	Comment
QI	BOOL	Input Qualifier (Controls the Initialization Process)

Data Outputs

Name	Type	Comment
QO	BOOL	Output Qualifier (Processing Status)
STATUS	STRING	Status Message
Q_timeout	BOOL	Timeout Flag
timestamp_timeout	DINT	Timeout Timestamp
timestamp_data	DINT	Timestamp of valid data
TD	DT	Combined time/date (IEC 61131-3 DT)
LOCAL_MINUTE_OFFSET	SINT	Local minute offset (SPN 1601, Unit: 1 min/bit, Offset −125)
LOCAL_HOUR_OFFSET	SINT	Local hour offset (SPN 1602, Unit: 1 h/bit, Offset −125)

Adapter

No adapters available.

Functionality

The function block encapsulates the following steps:

1. **Initialization**

An event at input INIT starts the internal function block I_TD (I_CORE). This block executes its own initialization sequence and confirms with INITO. The qualifiers QO and STATUS are passed to the corresponding outputs.

1. **Cyclic Time/Date Capture**

Once the internal function block is ready, it cyclically generates a IND event. This event triggers the downstream function block F_CONCAT_DT. F_CONCAT_DT combines the individual components (year, month, day, hour, minute, second) provided by I_TD into a DT struct. The millisecond component is fixed at 0. After concatenation is complete, a CNF event is generated, which activates the external IND output, along with all other data outputs (timestamps, offsets, etc.).

1. **Timeout**

If the internal service reports a timeout, the TIMEOUT event is activated at the output. The associated data (timestamp_timeout, STATUS, Q_timeout) are taken directly from the inner block.

1. **Forwarding the Raw Data**

The outputs timestamp_data, timestamp_timeout, LOCAL_MINUTE_OFFSET, and LOCAL_HOUR_OFFSET are taken unchanged from I_TD.

Technical Features

- **Wrapper Principle**

This block simplifies the handling of the ISOBUS time service by combining the individual time components into a platform-compatible `DT` struct. The user does not need to perform any custom chaining of year, month, etc.

- **ISO 11783-7 compliant**

The underlying service corresponds to PGN number 65254 and provides the SPNs (1601, 1602) required by ISO 11783-7 for local time offsets. The attribute information (scaling, offset) is stored in the output variables.

- **Timestamp handling**

Separate timestamps (`DINT`) are provided for both valid data and timeout events, allowing the sequence and recency of events to be tracked.

- **Event-driven process**

Processing is purely event-driven. Each cycle is initiated by the `IND` of the internal component and ends with the `IND` of the wrapper.

State Overview

The function block does not have an explicit internal state machine that would be visible to the user. However, the process can be divided into the following phases:

1. **Idle State** – After initialization, the function block waits for the first `IND` of `I_TD`.
2. **Active Processing** – `IND` of `I_TD` triggers `F_CONCAT_DT`; subsequently, the outer `IND` is output.
3. **Timeout State** – If `I_TD` times out, `TIMEOUT` is output.
4. **Initialization Phase** – The internal module is initialized during `INIT` and `INITO`.

Application Scenarios

- **ISOBUS Control Units**

Used in agricultural control units that regularly query the current UTC time and the local time offset from the ISOBUS bus and require this data as a uniform data type (`DT`) for display or logging.

- **Time Synchronization**

Used in systems that rely on an exact time base, e.g., recording devices (loggers), GPS-based applications, or process controls.

- **Wrapper for IEC 61131-3 Compatibility**

If subsequent function blocks do not expect access to individual time components, but only to a `DT` struct, `I_TD_TD` can serve as a link.

Comparison with Similar Function Blocks

- **I_TD (Basic Function Block)**

Returns the time components as separate outputs (YEAR, MONTH, DAY, etc.). The user would have to concatenate them manually to a `DT`. `I_TD_TD` does this automatically and also adds the raw data timestamps.

- **Other Time/Date Wrappers**

Some implementations use direct system time functions. `I_TD_TD` is specifically optimized for use in an ISOBUS context and provides the required SPN attributes as well as timeout handling.

Conclusion

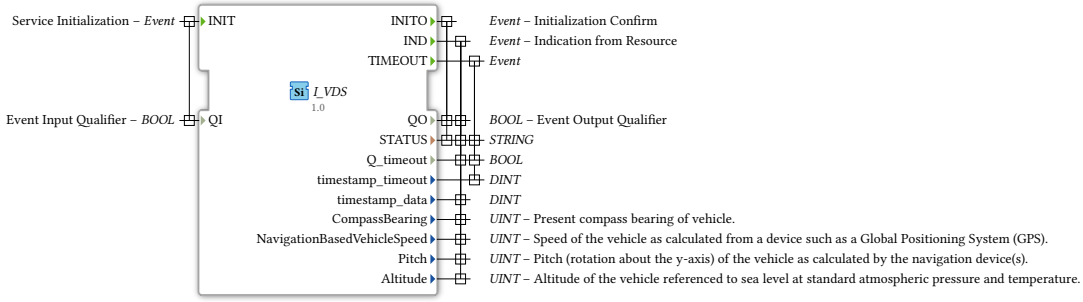
I_TD_TDextends the standardized ISOBUS time blockI_TDby adding a direct output of the IEC 61131-3 date-time typeDT`. This simplifies integration into control applications that do not want to perform a separate date conversion. The transparent transmission of raw data and timeout information, as well as the consistent event handling, make this block a practical and standards-compliant component for ISOBUS-based systems.

I_VDS

• – ◦ • – ◦ • – ◦ •

Introduction

The **I_VDS** function block (Vehicle Direction/Speed) is an ISO 11783-7 compliant system for the precise detection of vehicle direction, speed, and inclination, developed under the EPL-2.0 license. The block integrates navigation data for modern agricultural machinery and commercial vehicles.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with qualifier `QI`)

Event Outputs

- `INITO`: Initialization Confirmation (with status)
- `IND`: Data Indication with Navigation Parameters (with `QO`, `timestamp_data`, `CompassBearing`, `NavigationBasedVehicleSpeed`, `Pitch`, `STATUS`, `Q_timeout`, `Altitude`). Fires on every cycle without an active timeout and thereby also publishes the reset of `Q_timeout` to FALSE
- `TIMEOUT`: Timeout Event (with `timestamp_timeout`, `STATUS`, `Q_timeout`). Fires only when a timeout actually occurs; the reset of `Q_timeout` is reported via `IND`, not here

Data Inputs

- `QI` (BOOL): Qualifier for Initialization

Data Outputs

| Parameter | Type | SPN | Bit | Scale | Range | Description | | |–––––|––|––|––|–––––|–––––|–––––| | `QO` | BOOL | - | 1 | - | TRUE/FALSE | Event Qualifier | | `STATUS` | STRING | - | - | - | - | System Status Message | | `CompassBearing` | UINT | 165 | 16 | 1/128°/bit | 0-360° | Compass Direction | | `NavigationBasedVehicleSpeed` | UINT | 517 | 16 | 1/256 km/h/bit | 0-250 km/h | GPS-based speed | | `Pitch` | UINT | 583 | 16 | - | 0xFFFF | Pitch angle (reserved) | | `Altitude` | UINT | 580 | 16 | - | 0xFFFF | Altitude above sea level (reserved) | | `Q_timeout` | BOOL | - | - | - | TRUE/FALSE | Timeout flag (set with `TIMEOUT`, reset via `IND`) | | `timestamp_data` | DINT | - | - | - | - | Timestamp of the data (with `IND`) | | `timestamp_timeout` | DINT | - | - | - | - | Timestamp of the timeout (with `TIMEOUT`) |

Functionality

1. **Initialization**:

- `INIT` with `QI`=TRUE starts the GPS/navigation system
- `INITO` confirms operational readiness with `QO` and `STATUS`

1. **Data Update**:

- `IND` provides continuously updated navigation data
- Standard update rate: 100ms for real-time applications

1. **Error Handling**:

- `TIMEOUT` in case of signal loss or communication errors
- Status messages in the `STATUS` field

Technical Features

✓ **ISO 11783-7 compliant** (PGN) 65256)
✓ **High-precision direction detection** with 0.008° resolution
✓ **GPS-based speed** with 0.004 km/h accuracy

✓ **Extended sensor data** (pitch angle/altitude reserved)

Application Scenarios

- **Precision agriculture**: Automatic guidance
- **Fleet management**: Real-time tracking and speed monitoring
- **Slope detection**: Safety systems for slopes
- **Telematics**: Operational data acquisition and analysis

Data formats

Parameter	Example value	Meaning
`CompassBearing`	16384	128° (16384 × 1/128)
`NavigationBasedVehicleSpeed`	12800	50 km/h (12800 × 1/256)
`Pitch`	0xFFFF	Not used
`Altitude`	0xFFFF	Not used

⚖️ **Comparison with similar systems**

Feature	I_VDS	Standard GPS	Premium Navigation
Accuracy	±0.1° Direction	±5°	±0.05°
Speed	±0.1%	±2%	±0.05%
Update Rate	10 Hz	1 Hz	20 Hz
ISO Compliance	Full	Partial	Full

🛠️ **Related Exercises**

- Exercise_073
- Exercise_079

Conclusion

The I_VDS module offers precise navigation for mobile machinery:

- **Precision**: Subgrade-accurate direction detection
- **Reliability**: Robust GPS-based speed
- **Future-proof**: Reserved parameters for extensions

Ideal for use in:

- Automatic steering systems
- Precision agriculture
- Fleet management systems
- Safety-critical applications

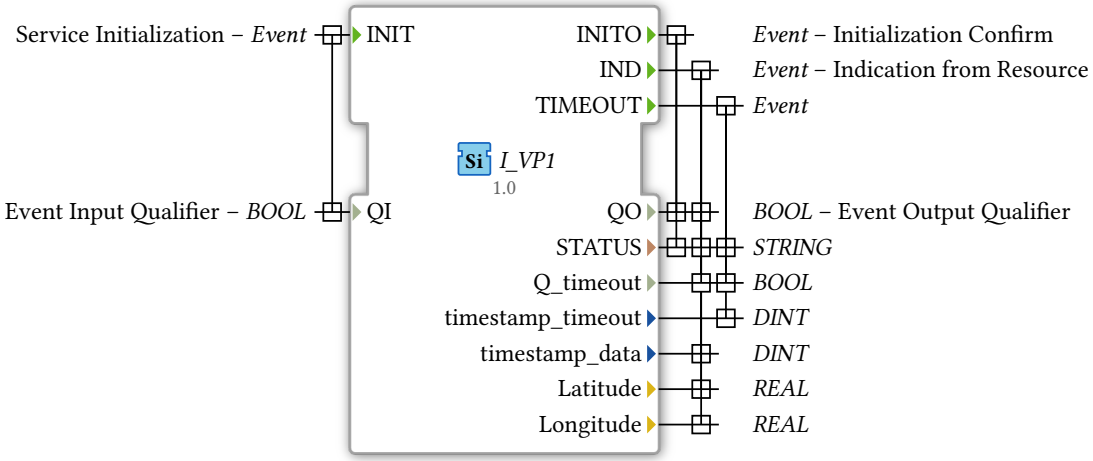
Designed for high-precision vehicle guidance and monitoring

I_VP1

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `I_VP1` is used to process and provide vehicle position data according to the ISO 11783 standard (ISOBUS). It specifically implements the “Vehicle Position 1” function, which is defined by the parameter Group Number (PGN) 65267. The block receives position data from a resource (e.g., a GNSS receiver) and makes it available via standardized interfaces for downstream control and display functions in an agricultural or mobile machinery network.



Interface Structure

Event Inputs

- **`INIT`**: Initialization event. Triggered on `QI=TRUE` to start the function block. Deactivated on `QI=FALSE`.

Event Outputs

- **`INITO`**: Confirms completion of initialization. Sent in response to the `INIT` event.
- **`IND`**: Indication event. Signals successful reception of new, valid position data from the connected resource.
- **`TIMEOUT`**: Timeout event. Triggered if no new position data is received within a configured time frame.

Data Inputs

- **`QI` (BOOL)**: Qualifies the `INIT` event. Controls the activation state of the function block (`TRUE` = activate, `FALSE` = deactivate).

Data Outputs

- **`QO` (BOOL)**: Qualifies the event outputs `INITO` and `IND`. Reflects the current operating state of the function block (`TRUE` = active and ready for operation).
- **`STATUS` (STRING)**: Provides status or error messages of the function block in a readable format.
- **`Q_timeout` (BOOL)**: Indicates whether a timeout has occurred (`TRUE` = timeout active, `FALSE` = no timeout).
- **`timestamp_timeout` (DINT)**: Timestamp (e.g., in milliseconds) associated with the `TIMEOUT` event.
- **`timestamp_data` (DINT)**: Timestamp of the last received position data output via `IND`.
- **`Latitude` (REAL)**: The geographical latitude of the vehicle's position in degrees. The value is scaled according to PGN 65267 (SPN 584) and includes an offset of −210°.
- **`Longitude` (REAL)**: The geographical longitude of the vehicle's position in degrees. The value is scaled according to PGN 65267 (SPN 585) and includes an offset of −210°.

Adapters

This function block does not use any adapter interfaces.

Operation

1. **Initialization**: Sending a `INIT` event with `QI=TRUE` activates the function block. It reports a successful start with the `INITO` output event, accompanied by `QO=TRUE` and a `STATUS` string.
2. **Data Reception and Processing**: In the active state, the function block monitors a connected resource (e.g., an ISOBUS data stream) for the arrival of position data according to PGN
3. Upon receiving valid data, it is decoded, scaling and offset are applied, and the resulting `Latitude` and `Longitude` values are calculated.
4. **Data Output**: Upon successful processing, the function block triggers the `IND` event. Simultaneously, the calculated position data (`Latitude`, `Longitude`), a corresponding timestamp (`timestamp_data`), the active state (`QO=TRUE`), and a status are provided via the output variables. `Q_timeout` is set to `FALSE`.

5. **Timeout Monitoring**: The function block continuously monitors the data stream. If no new position data is received for a configured period, it triggers the `TIMEOUT` event. This sets `Q_timeout=TRUE`, a timestamp (`timestamp_timeout`), and a corresponding `STATUS`.
6. **Deactivation**: A `INIT` event with `QI=FALSE` resets the module to an inactive state, which is acknowledged by `INITO` with `QO=FALSE`.

Technical Features

- **ISOBUS Compliance**: The module is specifically designed for use in ISO 11783 (ISOBUS) networks and processes data according to the official specification for PGN 65267.
- **Data Encoding**: The raw data for latitude (`Latitude`, SPN 584) and longitude (`Longitude`, SPN 585) are received as 4-byte values. The function block applies the scaling defined in the attributes and an offset of −210 degrees to calculate the final `REAL` values.
- **Attribute-based metadata**: The output variables for the position data are annotated with detailed attributes (SPN, name, length, offset, reference link). This simplifies configuration, documentation, and maintenance within an IEC 61499 development environment.

State overview

The function block can transition to the following main states:

- **Inactive**: Initial state. `QO=FALSE`. No data processing.
- **Active (waiting for data)**: After successful initialization. `QO=TRUE`. The function block monitors the input data stream.
- **Data processing**: Upon receiving a new PGN 65267 data packet. Decoding, scaling, and calculation of position values.
- **Data Output**: Triggering the `IND` event and setting the output variables.
- **Timeout**: Triggering the `TIMEOUT` event if no data is received.

Application Scenarios

- **Precision Farming / Sub-Area Management**: Providing precise vehicle positions for application maps, guidance systems, or documentation purposes.
- **Machine and Fleet Management**: Tracking agricultural or municipal vehicles within a company's premises.
- **Assistance Systems**: Basis for collision avoidance systems or automatic steering systems that require accurate and standardized position information.

⚖️ Comparison with Similar Blocks

- **Generic Input Blocks (e.g., `E_SR`, `E_RTRIG`)**: These offer basic read or trigger functions but are not designed to process a specific ISOBUS PGN. `I_VP1`, on the other hand, contains the complete logic for decoding, scaling, and error handling for PGN 65267.
- **General ISOBUS Input Blocks**: More general blocks that read various PGNs may exist. `I_VP1` is specialized and optimized for the efficient and reliable processing of vehicle position data, which simplifies configuration and reduces the potential for errors.

Conclusion

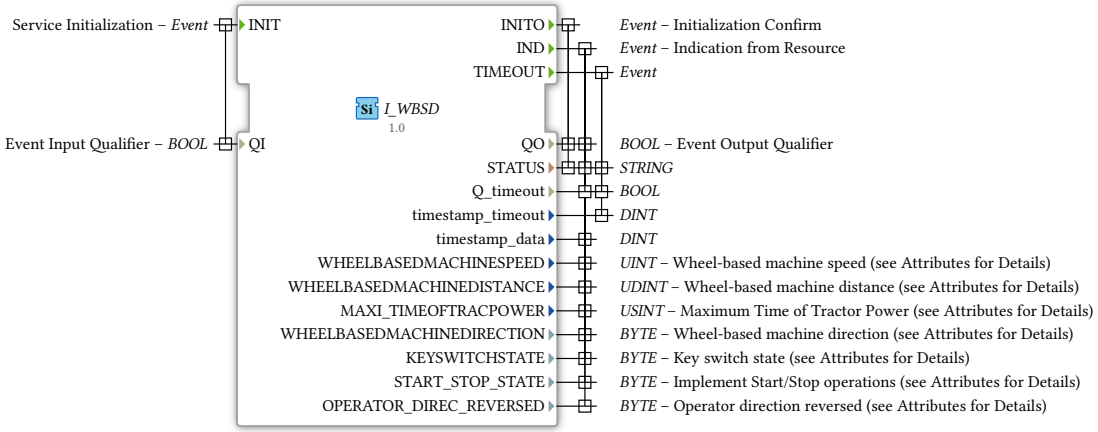
The `I_VP1`The -Function Block is a specialized, standards-compliant tool for integrating vehicle position data into IEC 61499-based control systems for mobile machinery. Through the direct implementation of the ISOBUS specification (PGN 65267) and integrated features such as timeout monitoring, it offers a reliable and easy-to-use interface for applications in precision farming and automation. Its attribute-based documentation supports developers in correct implementation and maintenance.

I_WBSD

• − ◦ • − ◦ • − ◦ •

Introduction

The **I_WBSD** function block (Wheel-based Speed and Distance) is an ISO 11783-7 compliant system for acquiring wheel-based speed and distance data, developed under the EPL-2.0 license. The block enables the precise monitoring of vehicle movements and operating conditions for agricultural and construction machinery.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with qualifier `QI`)

Event Outputs

- `INITO`: Initialization Confirmation (with status)
- `IND`: Data Indication with Motion and State Parameters (incl. `timestamp_data`, `Q_timeout`); fires on every cycle without an active timeout and thereby also publishes the reset of `Q_timeout` to `FALSE`
- `TIMEOUT`: Timeout Event (with `timestamp_timeout`, `STATUS`, `Q_timeout`); fires only when a timeout actually occurs (entry trigger). The reset of `Q_timeout` to `FALSE` is not reported here but via `IND`

Data Inputs

- `QI` (BOOL): Qualifier for Initialization

Data Outputs

Parameter	Type	SPN	Bit	Scale	Range	Description	
`QO`	BOOL	-	1	-	TRUE/FALSE	Event Qualifier	
`STATUS`	STRING	-	-	-	-	System Status Message	
`Q_timeout`	BOOL	-	-	-	TRUE/FALSE	Timeout flag (`TRUE` while a timeout is active)	
`timestamp_timeout`	DINT	-	-	-	-	Timestamp of the timeout (with `TIMEOUT`)	
`timestamp_data`	DINT	-	-	-	-	Timestamp of the data (with `IND`)	
`WHEELBASEDMACHINESPEED`	UINT	1862	16	0.001 m/s/bit	0-65.535 m/s	Wheel-Based Speed	
`WHEELBASEDMACHINEDISTANCE`	UDINT	1863	32	0.001 m/bit	0-4,294,967 km	Distance traveled	
`MAXI_TIMEOFTRACPOWER`	USINT	1866	8	1 min/bit	0-255 min	Maximum operating time	
`WHEELBASEDMACHINEDIRECTION`	BYTE	1864	2	4 states	0-3	Direction of travel	
`KEYSWITCHSTATE`	BYTE	1865	2	4 states	0-3	Ignition switch state	
`START_STOP_STATE`	BYTE	5203	2	4 states	0-3	Start/Stop status	
`OPERATOR_DIREC_REVERSED`	BYTE	5244	2	4 States	0-3	Reverse

Driving Direction States

Code	State	Description
0	Stationary	No movement
1	Forward	Forward travel
2	Reverse	Reverse travel
3	Undefined	Direction not determinable

Functionality

1. **Initialization**:

- `INIT` with `QI`=TRUE starts wheel sensor calibration
- `INITO` confirms operational readiness with system status

1. **Data Provision**:

- `IND` provides continuously updated motion data
- Automatic adjustment of the update rate (100 ms at >0.5 m/s)

1. **Error Handling**:

- `TIMEOUT` in case of signal loss from the wheel speed sensors
- Status messages in the `STATUS` field

Technical Features

✓ **ISO 11783-7 compliant** (PGN 65096) ✓ **Precise speed measurement** with 1 mm/s Resolution ✓ **32-bit distance counter** for extended operating times ✓ **Integrated status monitoring** (ignition, start/stop)

Application Scenarios

- **Tractors**: Speed control for field work
- **Harvesting machines**: Working distance calculation
- **Fleet management**: Operating hour recording
- **Safety systems**: Direction detection when reversing

Status Codes

Parameter	Code	Meaning
`KEYSWITCHSTATE`	0	Off
	1	On
	2	Start
	3	Undefined
`START_STOP_STATE`	0	Stop
	1	Start
	2	Pause
	3	Reserved

⚖️ Comparison with similar systems

Feature	I_WBSD	Standard	GPS-based
Accuracy	±0.5%	±2%	±5%
Low speed	Good	Excellent	Poor
Signal stability	High	Medium	Low
ISO compliance	Full	Partial	Full

🛠️ Related exercises

- Uebung_070
- Uebung_071
- Uebung_071a
- Uebung_071b
- Uebung_072
- Uebung_072b
- Uebung_079

Conclusion

The I_WBSD module offers reliable movement data for mobile machines:

- **Robust**: Independent of GPS signal
- **Precise**: Millimeter-accurate distance measurement

- **Comprehensive**: Integrated Condition monitoring

Ideal for use in:

- Precision agriculture
- Automatic steering systems
- Machines with frequent changes of direction
- Applications with high accuracy requirements

Designed for use in demanding environments

2.10.2.4 UT

🎧 Podcast

- 4diac IDE: Your "Hello World" of Automation – The Blinking Tutorial Local
- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- Taming the Software Dragon: Industrial Automation and the Future of Production
- Eclipse 4diac 3.0: ST Interpreter, FBE, and 7200 Commits – The Turbo for Distributed Automation

📺 Video

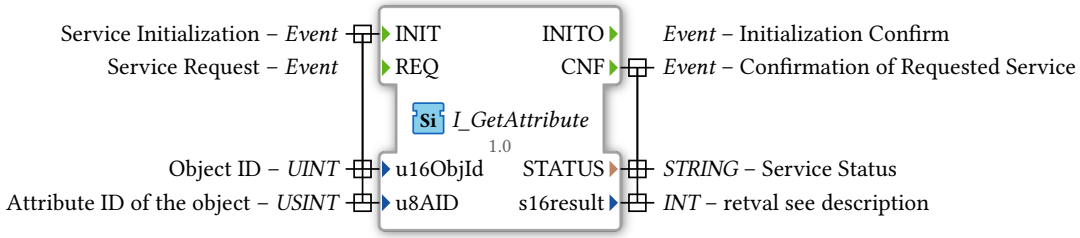
- Automation of the Future
- The Dual Wi-Fi Solution

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

I

I_GetAttribute



• – ◦ • – ◦ • – ◦ •

Introduction

`I_GetAttribute` implements the **Get Attribute Value** command (ISO 11783-6, F.58). It is available from VT version 4 and is used by a working set to query the VT for the current state of objects within the VT.

Interface Structure

Event Inputs

- **INIT** (`EInit`): Service initialization
  - Associated with: `u16ObjId`, `u8AID`
- **REQ**: Service request

Event Outputs

- **INITO** (`EInit`): Initialization confirmation
- **CNF**: Confirmation of the requested service
  - Associated with: `STATUS`, `s16result`

Data Inputs

- **u16ObjId** (`UINT`, initial value `ID_NULL`): Object ID
- **u8AID** (`USINT`): Attribute ID of the object

Data Outputs

- **STATUS** (STRING): Service status
- **s16result** (INT): Return value (see below)

Adapters

No adapter interfaces.

How it works

The Get Attribute Value command (F.58) is a fixed 8-byte message (object ID in bytes 2 and 3, attribute ID in byte 4; no Transport Protocol).

Before the command is ever sent, INIT validates both the object type (iso_has_readable_attribute_id: does this object type have any attribute ID at all, read-only or writable?) and the specific attribute ID against that type's attribute table (iso_is_readable_attribute). Unlike Change Attribute, both read-only and writable attribute IDs are accepted here. The underlying C implementation (cmd_get_attribute_value) performs no type check of its own, only ID_NULL handling; the VT additionally validates the object ID and the attribute ID on its side and returns an error response if either is invalid.

ID_NULL (65535) is not a command target but can be used in INIT to deactivate the function block.

Return values (s16result)

Value	Meaning
VT_E_NO_ERR (0)	OK
VT_E_DEACTIVATED (-40)	FB deactivated via ID_NULL on INIT
VT_E_INVALID_OBJECT_ID (-132)	The object ID's type has no readable AID at all (iso_has_readable_attribute_id)
VT_E_INVALID_ATTRIBUTE_ID (-133)	Attribute ID is not a valid/readable AID for this object type (iso_is_readable_attribute)
VT_E_NOT_READY (-131)	Buffered: INIT not yet completed / VT not yet ready
VT_E_OVERFLOW (-6)	Buffer overflow
VT_E_NOACT (-8)	Command not possible in current state
VT_E_NO_INSTANCE (-21)	No VT client available
VT_E_HANDLE_INVALID (-128)	Error cause: invalid handle
VT_E_ISO_INSTANCE_INVALID (-129)	Identifier for instance (connection) invalid
VT_E_NOT_ALIVE (-130)	Identifier for instance valid, but VT dead

Technical Details

- **Instance uniqueness**: This block requires uniqueness with respect to the pair (u16ObjId, u8AID). Only one instance of I_GetAttribute may exist in the whole program for the same object/attribute pair. u8AID is read in at INIT; a second instance for the same object ID and attribute ID is deactivated at INIT.
- The block delivers only status and return value via CNF; the read attribute value is delivered by service interface blocks such as Attribute_IDA.

Application Scenarios

- Querying the current state of an object in the VT (from VT version 4)

Summary

I_GetAttribute sends the Get Attribute Value command (F.58) for an object/attribute pair and reports status and return value.

Q

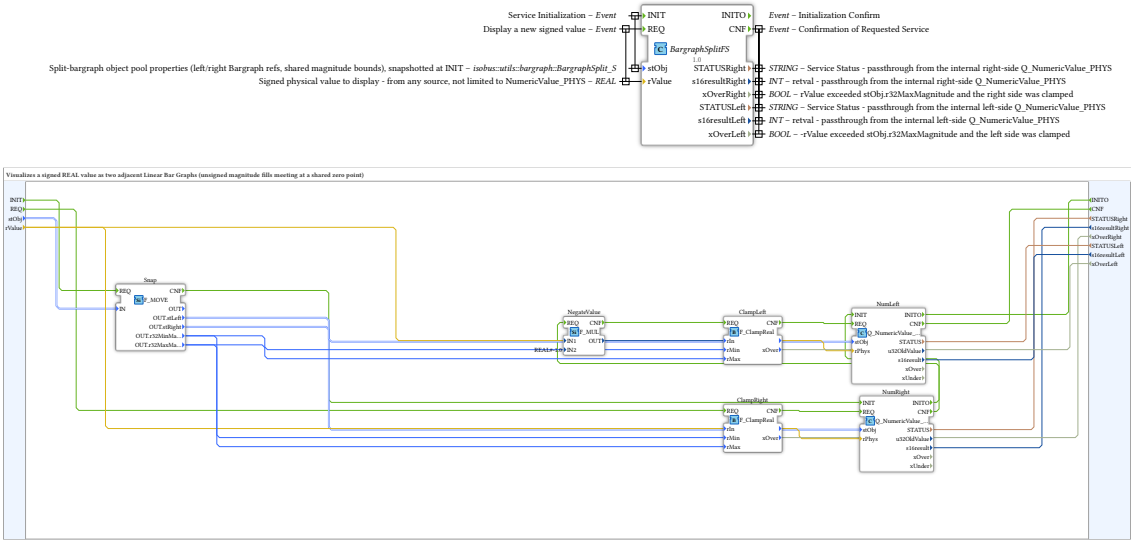
📌 Architecture & Guidelines

- Instance Uniqueness of Q_* Function Blocks

🎧 Podcast

- Das Geheimnis des 2-Bit-Quarter: Effizienz im CAN-Bus für Nutzfahrzeuge
- QUARTER
- 0 und 1: Warum das Binärsystem trotz Quantensprung die Grundlage unserer digitalen Welt bleibt
- Open Circuits: Die verborgene Ästhetik und faszinierende Komplexität im Querschnitt unserer Elektronik

BargraphSplitFS



• – ○ • – ○ • – ○ •

Introduction

The **BargraphSplitFS** function block visualizes a signed REAL value on a pair of adjacent Linear Bar Graphs that meet at a shared zero point. The left bar graph renders the magnitude of negative values, while the right bar graph renders the magnitude of positive values. This arrangement provides an intuitive "split" display where the user sees a single continuous bar that extends from the center to either side, depending on the sign of the input value.

The block wraps two `Q_NumericValue_PHYS` instances (ISO 11783-6, clause F.22) that command the two bar graph objects defined in Annex B.11.3. The object configuration is provided once at initialization and remains fixed for the lifetime of the block instance.

Interface Structure

Event Inputs

Event	Description	With
`INIT`	Service initialization. Snapshots the `stObj` configuration.	`stObj`
`REQ`	Displays a new signed value on the split bar graph.	`rValue`

Event Outputs

Event	Description	With
`INITO`	Initialization confirm. Emitted after both internal bar graph services are initialized.	–

CNF	Confirmation of the requested display service. Emitted after both sides have processed the value.	STATUSRight, s16resultRight, xOverRight, STATUSLeft, s16resultLeft, xOverLeft

Data Inputs

Data	Type	Description
stObj	isobus::utils::bargraph::BargraphSplit_S	Split-bargraph object pool properties (left/right bar graph references, shared magnitude bounds). Snapshotted once at INIT.
rValue	REAL	Signed physical value to display. May originate from any source, not limited to a NumericValue_PHYS.

Data Outputs

Data	Type	Description
STATUSRight	STRING	Service status – passthrough from the internal right-side Q_NumericValue_PHYS.
s16resultRight	INT	Return value – passthrough from the internal right-side Q_NumericValue_PHYS.
xOverRight	BOOL	TRUE when the positive input value exceeded stObj.r32MaxMagnitude and the right side was clamped.
STATUSLeft	STRING	Service status – passthrough from the internal left-side Q_NumericValue_PHYS.
s16resultLeft	INT	Return value – passthrough from the internal left-side Q_NumericValue_PHYS.
xOverLeft	BOOL	TRUE when the negated negative input value exceeded stObj.r32MaxMagnitude and the left side was clamped.

Adapters

None.

Functionality

The block operates in two phases: initialization and value display.

Initialization (INIT):

1. The stObj structure is snapshotted into an internal copy using an F_MOVE block (Snap).
2. From the snapshot, the right-side configuration (stRight) is forwarded to the internal right Q_NumericValue_PHYS, and the left-side configuration (stLeft) is forwarded to the internal left Q_NumericValue_PHYS.
3. Both internal instances are initialized sequentially. The INITO event confirms completion.

Value display (REQ):

The signed rValue is processed along two parallel paths:

- **Right side (positive branch):** rValue is fed directly into a clamping stage (ClampRight) that bounds the value to [stObj.r32MinMagnitude, stObj.r32MaxMagnitude]. The clamped magnitude is then written to the right bar graph via Q_NumericValue_PHYS. If rValue > r32MaxMagnitude, xOverRight is set to TRUE.
- **Left side (negative branch):** rValue is first negated by an F_MUL with -1.0, producing the magnitude of the negative input. This value is clamped with the same bounds (ClampLeft) and written to the left bar graph. If -rValue > r32MaxMagnitude, xOverLeft is set to TRUE.

Both branches are processed using a strict sequential event chain: `REQ → ClampRight → NumRight → NegateValue → ClampLeft → NumLeft → CNF`. This guarantees deterministic ordering and ensures both bar graphs are updated before the confirmation event is emitted.

The clamping stage maps `r32MinMagnitude` (typically 0) to the empty state, while `r32MaxMagnitude` corresponds to the full-scale fill of the bar graph. Values below `r32MinMagnitude` are clamped upward to the minimum, and values above `r32MaxMagnitude` are clamped to full scale.

There are deliberately no `xUnder` outputs. A magnitude falling below `r32MinMagnitude` on the currently inactive side is the normal resting state during ordinary operation and does not constitute a diagnostic condition.

Technical Features

- **Object snapshotting:** The `BargraphSplit_S` structure is captured once at `INIT` via `F_MOVE`, making the block immune to later changes of the source object references.
- **Reusable internal services:** The block leverages standard `Q_NumericValue_PHYS` (ISO 11783-6 F.22) and `F_ClampReal` building blocks, ensuring consistent behavior with other parts of the ISOBUS object pool.
- **Shared bounds:** Both bar graph sides use the same `r32MinMagnitude` and `r32MaxMagnitude` from the snapshot, guaranteeing a symmetric, coherent presentation around the zero point.
- **Deterministic event sequencing:** The internal event chain guarantees that both sides are fully processed in a fixed order before `CNF` is emitted.
- **Over-range indication:** Dedicated boolean outputs report magnitude overruns independently for each side, allowing the application to react (e.g., log a diagnostic or highlight the value).
- **No under-range diagnostic:** By design, the block omits under-range flags to avoid spurious diagnostics during normal zero-crossing operation.

State Overview

The block is stateless from the application perspective. Its internal lifecycle is:

State/Phase	Trigger	Actions
Idle	–	Waiting for `INIT` or `REQ`.
Initializing	`INIT`	Snapshot `stObj`, initialize right instance, then left instance, emit `INITO`.
Processing	`REQ`	Clamp right branch, update right bar graph, negate/clamp left branch, update left bar graph, emit `CNF`.
Ready	`INITO` / `CNF`	Return to idle, awaiting next service request.

Application Scenarios

- **Tank level / fill indication with direction:** Displaying a differential value (e.g., pressure difference, flow balance) where positive values fill the right bar and negative values fill the left bar.
- **Dual-directional machine parameter display:** Showing joystick deflection, steering angle, or actuator position in both directions from a neutral center point.
- **Operator diagnostics:** In ISOBUS-based agricultural machinery, presenting signed physical quantities (speed deviation, draft force imbalance) in a compact split-bar format.
- **Generic human-machine interface:** Any HMI requiring a signed value to be rendered as two adjacent magnitude bars with a common zero, using standard ISO 11783-6 bar graph objects.

Comparison with Similar Blocks

Feature	BargraphSplitFS	Simple single bar graph FB
Display of signed values	Yes – split into two directional bars	Usually requires absolute value pre-processing
Zero point	Shared central zero between two bars	Typically at one end of the bar
Over-range reporting	Per-side via `xOverRight` / `xOverLeft`	Often single over-range flag

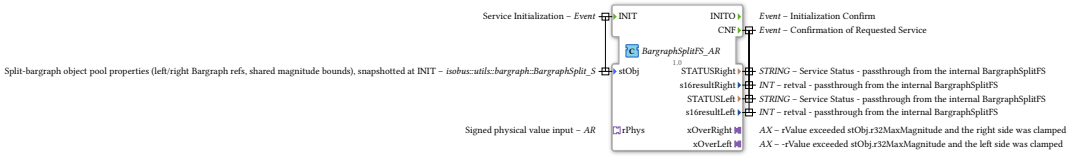
Under-range reporting	None (intentional)	Often present, may cause false alarms on inactive side
Configuration snapshotting	Yes, at `INIT`	Varies

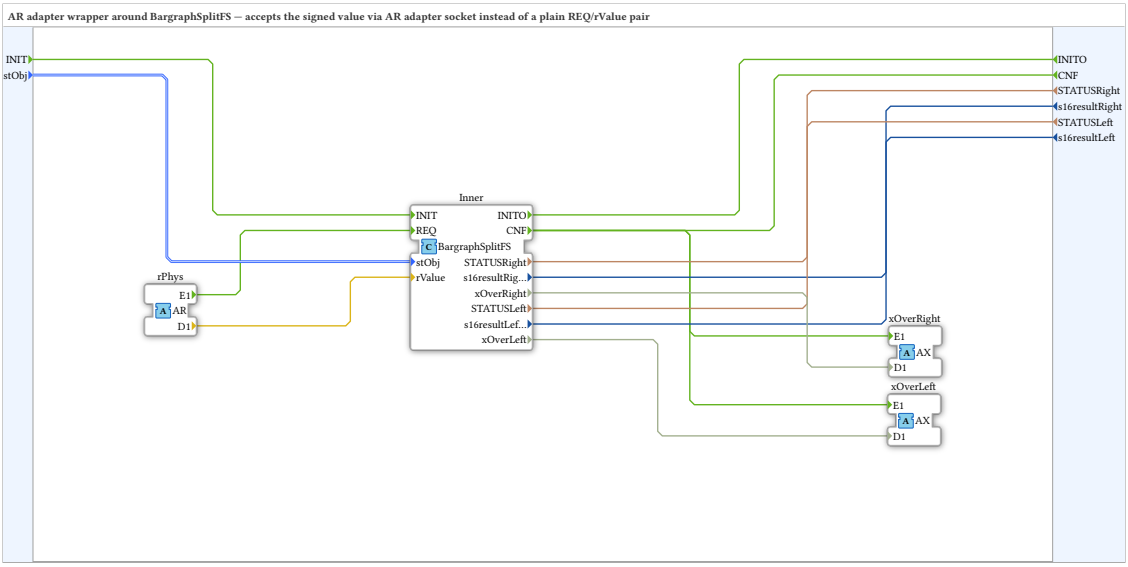
Unlike a single `Q_NumericValue_PHYS` block, which can only display an unsigned magnitude, BargraphSplitFS preserves the sign information natively by assigning the value to one of two bar graph objects. This removes the need for external sign-detection and absolute-value logic in the application.

Conclusion

The BargraphSplitFS function block provides a robust and reusable solution for displaying signed REAL values on a pair of adjacent linear bar graphs. By combining ISO 11783-6 compliant `Q_NumericValue_PHYS` services with systematic clamping and deterministic event sequencing, it delivers a clean, symmetric visual representation where the zero point is always shared between the two bars. Its per-side over-range flags, configuration snapshotting, and deliberate omission of under-range diagnostics make it well suited for agricultural and industrial HMIs that require clear directional magnitude indication.

BargraphSplitFS_AR





• — ○ • — ○ • — ○ •

Introduction

`BargraphSplitFS_AR` is an adapter-based wrapper around the function block `BargraphSplitFS`. Instead of requiring an explicit `REQ` event and a plain `rValue` data input, this block accepts the value through a unidirectional `AR` (adapter-request) socket. This design simplifies integration in systems where the value is already provided via an adapter interface, such as those defined for ISO 11783-6 (ISOBUS) applications. The wrapper internally maps the adapter's event and data to the inner block's `REQ` and `rValue` pins, and passes through the results (status codes and numerical values) to the external interface. Over-range conditions on the left and right sides are re-exposed as `AX` adapter plugs, following the same pattern as `PositionMarkerFSA`.

Interface Structure

Event Inputs

Event	Type	Comment	With Var

INIT	EInit	Service Initialization	stObj

Event Outputs

Event	Type	Comment	With Var
INITO	EInit	Initialization Confirm	–
CNF	Event	Confirmation of Requested Service	STATUSRight, s16resultRight, STATUSLeft, s16resultLeft

Data Inputs

Name	Type	Comment
stObj	isobus::utils::bargraph::BargraphSplit_S	Split-bargraph object pool properties (left/right Bargraph refs, shared magnitude bounds), snapshotted at INIT

Data Outputs

Name	Type	Comment
STATUSRight	STRING	Service Status – passthrough from the internal BargraphSplitFS
s16resultRight	INT	Return value – passthrough from the internal BargraphSplitFS
STATUSLeft	STRING	Service Status – passthrough from the internal BargraphSplitFS
s16resultLeft	INT	Return value – passthrough from the internal BargraphSplitFS

Adapters

Type	Name	Direction	Comment
AX	xOverRight	Plug	rValue exceeded stObj.r32MaxMagnitude, right side clamped
AX	xOverLeft	Plug	–rValue exceeded stObj.r32MaxMagnitude, left side clamped
AR	rPhys	Socket	Signed physical value input (adapter request)

Functionality

The block is a pure wrapper that delegates all processing to an internal instance of BargraphSplitFS. Upon receiving an INIT event, the block snapshots the provided stObj structure and passes it to the inner block. When a value arrives through the rPhys adapter socket (its event E1 and data D1), the internal block's REQ and rValue inputs are activated and supplied accordingly. The inner block computes the two-sided bar graph and issues a CNF event together with the status and result values for both the right and left sides. These outputs are directly forwarded to the parent block's corresponding pins. Any over-range conditions (xOverRight, xOverLeft) are also propagated via the adapter plugs, using the same event/data connection mechanism.

The internal network connections confirm this behaviour:

- INIT ⟶ Inner.INIT
- rPhys.E1 ⟶ Inner.REQ
- rPhys.D1 ⟶ Inner.rValue
- Inner.INITO ⟶ INITO
- Inner.CNF ⟶ CNF, xOverRight.E1, xOverLeft.E1
- Data: stObj ⟶ Inner.stObj; all four status/result outputs are directly mapped from the inner block; Inner.xOverRight ⟶ xOverRight.D1 and Inner.xOverLeft ⟶ xOverLeft.D1.

Technical Features

- **Adapter-Based Input** – The signed physical value is supplied via a unidirectional `AR` socket, eliminating the need for a separate `REQ` event and `rValue` data pin.
- **Passthrough Outputs** – All status and return values of the internal block are directly forwarded without additional processing.
- **Over-Range Indication** – Over-range conditions are exposed as `AX` plugs in the same style as `PositionMarkerFSA`.
- **Single-Shot Initialization** – The `INIT` event transfers the configuration structure `stObj` to the internal block.
- **Complete Encapsulation** – The wrapper hides the internal event/data protocol, presenting a clean adapter interface to the surrounding application.

State Overview

The block itself does not maintain a state machine; it acts as a transparent bridge. Its operational behaviour is fully defined by the internal `BargraphSplitFS` block, which manages the actual split-bargraph calculation. From the perspective of an external caller, the block is ready to accept a value via `rPhys` after initialization (`INITO` has been issued). The `CNF` event signals that the processing of the current request has finished and that the output data are valid.

Application Scenarios

This block is suitable for ISOBUS (ISO 11783-6) applications where a split bar graph must be displayed for a signed physical value, and where the value is already delivered through an adapter interface. Typical use cases include:

- Tractor and implement displays that receive sensor data via a CAN-based adapter.
- Scenarios where the value source is reused across multiple logic parts and both left and right over-range indicators need to be forwarded to other visualisation components.
- Systems that prefer a uniform adapter-based interface for all value inputs, reducing wiring complexity and improving modularity.

Comparison with Similar Blocks

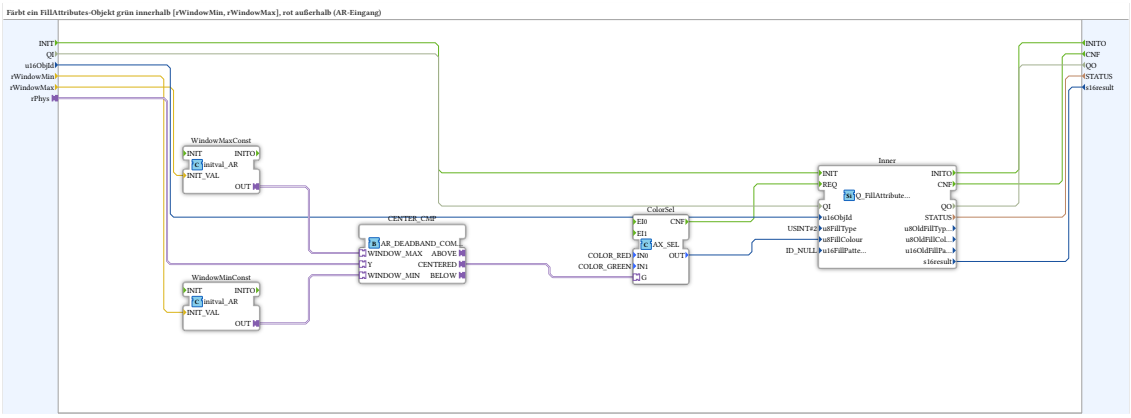
The primary difference to the base block `BargraphSplitFS` is the input mechanism. `BargraphSplitFS` requires an explicit `REQ` event and a separate `rValue` data input, while `BargraphSplitFS_AR` encapsulates these into a single `AR` adapter socket. This makes the block more convenient in adapter-centric architectures, but adds the overhead of a wrapper – the internal block still performs the same calculations.

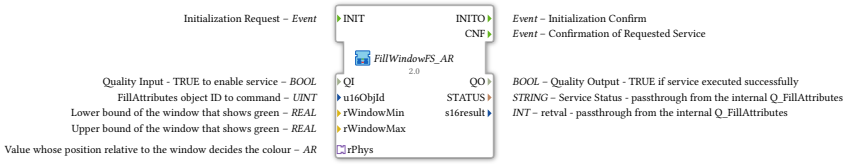
Compared to `PositionMarkerFSA`, which wraps a position marker block in a similar fashion, this block follows the same design pattern: it uses `AR` for the value input and `AX` for the over-range indications. The naming and structure are consistent, promoting a uniform look and feel across a suite of adapter-wrapped ISOBUS widgets.

Conclusion

`BargraphSplitFS_AR` provides a clean, adapter-based interface to the `BargraphSplitFS` functionality. It simplifies integration in systems that use adapter sockets for value input, while preserving all the features and outputs of the underlying block. The block is well-suited for ISOBUS applications that require a split bar graph with over-range indication and prefer a modular, adapter-oriented design.

FillWindowFS_AR





• — ○ • — ○ • — ○ •

Introduction

The **FillWindowFS_AR** subapplication is a reusable IEC 61499 composite function block designed to colorize a FillAttributes object based on the position of a physical value relative to a configurable window. It accepts a real-valued input through an AR adapter, evaluates whether this value falls within the interval `[rWindowMin, rWindowMax]`, and then commands an internal `Q_FillAttributes` function block (window comparison by an internal `AR_DEADBAND_COMPARE`) to apply either green (inside the window) or red (outside the window) to the specified visualization object. This subapp encapsulates the complete evaluation and coloring logic into a single, network-embeddable component.

Interface Structure

The subapplication exposes the initialization events `INIT`/`INITO`, the quality signals `QI`/`QO`, one confirmation event, three further data inputs, two further data outputs and one adapter socket.

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization request - passed through to the internal `Q_FillAttributes` (`Inner.INIT`).

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization confirmation (from `Inner.INITO`).
CNF	Event	Confirmation of the requested service, passed through from the internal `Q_FillAttributes` block upon completion.

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	TRUE	Quality Input - TRUE to enable the service (to `Inner.QI`).
u16ObjId	UINT	ID_NULL	FillAttributes object ID to command. Identifies the visualization object whose color shall be set.

rWindowMin	REAL	—	Lower bound of the window that shows green. Values equal to or greater than this bound are considered inside the window.
rWindowMax	REAL	—	Upper bound of the window that shows green. Values equal to or less than this bound are considered inside the window.

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if the service was executed successfully (from `Inner.QO`).
STATUS	STRING	Service status — passthrough from the internal `Q_FillAttributes` block, providing textual feedback about the last operation.
s16result	INT	Return value — passthrough from the internal `Q_FillAttributes` block, providing a numeric result code.

Adapters

Name	Type	Direction	Comment
rPhys	adapter::types::unidirectional::AR	Socket	Physical value whose position relative to the window determines the applied color. The incoming AR adapter carries a real number that is evaluated against the configured bounds.

Functionality

The subapplication implements a window evaluation combined with a two-color selection scheme. The functional flow proceeds as follows:

1. **Window Constants** — Two `initval_AR` blocks (`WindowMinConst` and `WindowMaxConst`) convert the externally provided `rWindowMin` and `rWindowMax` values into AR adapter values.
2. **Window Comparison** — The incoming AR adapter `rPhys` is connected as measured value `Y` to an `AR_DEADBAND_COMPARE` block (named `CENTER_CMP`), together with `WINDOW_MIN` (from `WindowMinConst`) and `WINDOW_MAX` (from `WindowMaxConst`). Its AX plug `CENTERED` is `TRUE` while `WINDOW_MIN ≤ Y ≤ WINDOW_MAX` holds (the plugs `ABOVE`/`BELOW` are not used here), i.e. the closed interval `[rWindowMin, rWindowMax]`.
3. **Color Selection** — `CENTERED` drives an `AX_SEL` block (named `ColorSel`) with `IN0` fixed to `COLOR_RED` and `IN1` fixed to `COLOR_GREEN`. When `CENTERED` is `TRUE`, green is selected; otherwise, red is selected. The chosen color is passed as the `OUT` value.
4. **Fill Command** — The color output from `ColorSel` is routed to the `u8FillColour` input of the internal `Q_FillAttributes` block (named `Inner`). Simultaneously, the external `u16ObjId` is connected to the block's object ID input. The block is parameterized with `u8FillType = USINT#2` (solid fill) and `u16FillPatternId = ID_NULL`.
5. **Event Handshaking** — The `ColorSel.CNF` event triggers the `Inner.REQ` event, initiating the fill operation. Upon completion, the internal block's `CNF` event is propagated to the subapplication's external `CNF` output.
6. **Result Propagation** — Both `STATUS` and `s16result` from the internal `Q_FillAttributes` block are forwarded directly to the subapplication's corresponding output ports, providing full transparency about the operation's outcome.
7. **Initialization and Quality Signals** — These are passed straight through: `INIT` ⟶ `Inner.INIT`, `Inner.INITO` ⟶ `INITO`, `QI` ⟶ `Inner.QI` and `Inner.QO` ⟶ `QO`.

Technical Features

- **Composite SubApplication Type** — Implemented as a `SubAppType` rather than a custom FBType, allowing easy reuse and modification through standard 4diac-IDE tooling.
- **Adapter-Based Input** — The physical value is acquired via a unidirectional AR adapter, decoupling the subapp from specific sensor or data source implementations and promoting type-safe connections.
- **Constant Initialization Pattern** — Utilizes `initval_AR` blocks to convert runtime REAL variables into adapter-based constant values, leveraging the adapter framework's type system.
- **Window Comparison** — Closed interval evaluation by `AR_DEADBAND_COMPARE` (`CENTERED` output), guaranteeing that boundary values are treated as inside the window. Since version 2.0 this replaces the former own split/GE/LE/AND wiring.
- **Color Mapping** — Pre-configured `AX_SEL` parameters with `COLOR_RED` and `COLOR_GREEN` constants, drawing from the `isobus::UT::Q::const::colours` namespace.
- **Event Propagation Chain** — A clean event cascade from selection confirmation to fill request, then back to the caller via the `CNF` output. Apart from `INIT` there are no event inputs; the coloring is triggered by new data on the adapter `rPhys` (or on the window bounds).
- **Status Passthrough** — Both status string and result integer are exposed at the subapp boundary, enabling callers to monitor and log operations without inspecting internal state.
- **Quality Pattern** — `QI` (default `TRUE`) enables the service, `QO` reports the success of the internal `Q_FillAttributes`.
- **Namespace Imports** — Explicit imports of `ID_NULL` and the color constants ensure compile-time correctness in the target runtime environment.

State Overview

`FillWindowFS_AR` does not maintain an explicit internal state machine of its own (`AR_DEADBAND_COMPARE` only reports changes of its outputs, and `INIT`/`INITO` are merely passed through). The observable behavior per service cycle can be summarized as:

Condition	Evaluation Result	Selected Color	Fill Command
`rPhys < rWindowMin`	Outside (below)	RED	Solid red fill
`rWindowMin ≤ rPhys ≤ rWindowMax`	Inside window	GREEN	Solid green fill
`rPhys > rWindowMax`	Outside (above)	RED	Solid red fill

The behavior is fully deterministic for a given set of inputs.

Application Scenarios

- **Visualization of Process Variables** — In a process control HMI, the subapp can color-code a numeric indicator (e.g., temperature, pressure, level) green when it lies within an acceptable operating range and red when it deviates, providing immediate visual feedback to operators.
- **Alarm and Limit Monitoring** — When integrated with alarm handling logic, the subapp can be used to repaint warning icons or trend markers based on whether a measured value violates configured thresholds.
- **Quality Control and Sorting** — In manufacturing or inspection systems, the block can visually flag measured dimensions or test results that fall outside tolerance windows.
- **Interactive Panels and Dashboards** — The subapp can drive color changes on dashboard widgets where a physical quantity's range (e.g., speed, fuel level, temperature) determines the displayed color.
- **Educational Demonstrations** — As a clearly structured composite, it serves as a learning example for adapter-based data flow, window comparisons, and event-driven service invocation in IEC 61499 systems.

Comparison with Similar Blocks

Feature	FillWindowFS_AR	Custom FBType with ST Helper	GreenRedBackground1_AX
Implementation	SubAppType composite	Custom compiled FBType	Similar SubApp approach
Input Paradigm	AR adapter for physical value	Direct data inputs	AR adapter

Window Evaluation	Internal `AR_DEADBAND_COMPARE (CENTERED)`	Inline ST condition	Configurable via parameters
Color Constants	Fixed RED/GREEN	Depends on implementation	Fixed RED/GREEN
Event Output	Single CNF	Depends on design	Single CNF
Reconfigurability	High (network editable)	Medium (code changes)	High
Runtime Dependencies	Only standard adapter FBs	Custom compiled code	Standard adapter FBs

Compared to a hand-coded FBType with ST logic, this subapp leverages standard building blocks, making it more transparent, portable, and inspectable at the network level. Relative to similar background-coloring subapps, it adds the window-based comparison and the physical-value adapter input in an integrated form.

Conclusion

The **FillWindowFS_AR** subapplication provides a compact, well-structured, and reusable solution for range-based color visualization of physical values on FillAttributes objects. By combining the standard window comparison block `AR_DEADBAND_COMPARE`, a selection block, and a `Q_FillAttributes` service block, it implements a complete green-inside / red-outside logic with minimal external wiring. The clear interface, event passthrough (`INIT`/`INITO`, `CNF`), quality signals (`QI`/`QO`) and status propagation make it straightforward to integrate into larger IEC 61499 applications, while the adapter-based input ensures compatibility with a wide range of sensor and data sources. It is a practical building block for HMI development, alarm visualization, and quality-related color-coding tasks in industrial automation environments.

Instance Uniqueness of Q_* Function Blocks

Each `Q_*` function block maintains its own independent value/pending buffer (`var_qPending`, `var_*OldValue`, see `sendXxx()`/`resetPendingState()`). If two instances of the SAME class are connected to the same target (Object ID, Mask ID, etc.), they operate independently without mutual awareness, leading to competing VT bus commands. Therefore, each block requires an instance uniqueness criterion defining what may exist at most once.

Since commit `792e35673` (“Detect duplicate object-ID claims across instances of the same FB class”), an enforcement mechanism exists: `CQ_VTCommandBase::mTargetObjectId` (+ optional `mTargetSubId` for composite keys, see `Q_Attribute` below) + `IsObjectIdAlreadyClaimed(objId, subId = 0xFFFF)` (`CQ_VTCommandBase.h`/`.cpp`). A second instance encountering an already claimed (Object ID[, Sub-ID]) pair upon `INIT` is deactivated with `STATUS = "This objID is already in use"`.

Criterion: One Instance PER OBJECT ID (Enforced)

The target is a variable VT Object ID (`u16ObjId`, set during `INIT` via `var_u16ObjId` and validated against class-specific ID ranges). Two instances with the same Object ID would compete independently over the same object state.

Function Block	Target ID Definition
`Q_Attribute`	**Pair** (Object ID, Attribute ID) - `mTargetObjectId=u16ObjId`, `mTargetSubId=u8IdAttribute`. Two instances targeting the same object but different attributes are valid; only duplicate pairs collide.
`Q_BackgroundColour`	Object with background color attribute. May share an Object ID with ONE `Q_BackgroundColourAux` instance (different FB class), but NEVER two `Q_BackgroundColour` instances on the same Object ID.
`Q_BackgroundColourAux`	AuxFunction object (AUX VT, **enforced**). Unique per Object ID within its own FB class (`getFBTypeId()`). May share an Object ID with ONE `Q_BackgroundColour` instance, but NEVER two `Q_BackgroundColourAux` instances on the same Object ID.

Q_ChangeObjectLabel	Labelable object
Q_ChangePolygonPoint	Polygon object
Q_ChangePolygonScale	Polygon object
Q_ChildLocation	Child object (Child Location)
Q_ChildPosition	Child object (Child Position)
Q_EndPoint	Line object
Q_FillAttributes	Object with fill attributes
Q_FontAttributes	Object with font attributes
Q_GraphicsContext	Graphics Context object
Q_LineAttributes	Object with line attributes
Q_ListItem	**Pair** (List Object ID, List Index) - mTargetObjectId=u16ObjId, mTargetSubId=u8ListIndex. Read at INIT via With Var="u8ListIndex" — claimed during INIT in iso_init().
Q_NumericValue	Numeric Value object. (ObjectPointer exception allows co-existence with Q_NumericValueAux).
Q_ObjEnableDisable	Enableable/disableable object
Q_ObjHideShow	Container object (3000–3999)
Q_ObjSelectInput	Input-selectable object
Q_Size	Resizable object
Q_StringValue	String Value object

Criterion: One Instance PER OBJECT ID (Not Yet Enforced)

Function Block	Target ID Definition	Notes
Q_NumericValueAux	AUX VT Object	Additional gate (VtAuxAssignment)
Q_ExecuteMacro	Macro object	
Q_ExecuteExtendedMacro	Macro object	
Q_LockUnlockMask	Mask Object (u16MaskId, read at INIT)	Target is a mask (u16MaskId)
Q_Priority	Alarm Mask object	

Criterion: Exactly ONE Instance Program-Wide (Singleton)

Function Block	Real Target (Parameter at REQ)	Status
Q_ActiveMask	u16NewMaskId (Fixed target ID = 0)	**Enforced**
Q_SelectActiveWorkingSet	Working Set Name	Open gap
Q_SelectColourMap	Colour Map Object ID	Open gap
Q_SetAudioVolume	Volume setting	Open gap
Q_SoftKeyMask	Data Mask / SoftKey Mask IDs	Open gap
Q_CtrlAudioSignal	Signal parameters	Open gap

No Criterion Needed (Stateless)

Function Block	Reason
`Q_ESC`	No Object ID or old value buffering. Direct transmission upon `REQ`.

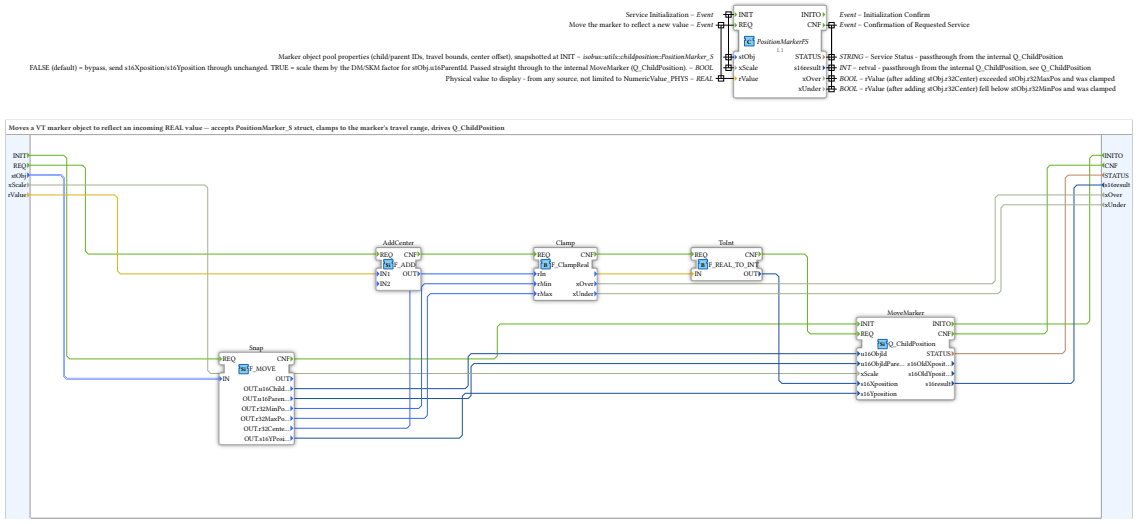
Adapter Wrappers (`_AUI`, `_AUDI`, `_AB`, `_AX`, `_AR`)

Adapter variants (e.g., `Q_NumericValue_AUDI`, `Q_ObjHideShow_AB`) are wrapper function blocks around their respective base classes (e.g., `Q_NumericValue`, `Q_ObjHideShow`). Because the adapter wrapper builds on the underlying base class, **instance uniqueness applies across the wrapper boundary**: An Object ID may be claimed program-wide by at most **one** instance (either base class or adapter wrapper). Simultaneous usage of base and wrapper instances on the same Object ID is invalid.

Separate I_* Module

`I_GetAttribute` (`isobus_UT/src/isobus/UT/I/`) requires pair uniqueness over (`u16ObjId`, `u8IdAttribute`).

PositionMarkerFS



• – ○ • – ○ • – ○ •

Introduction

PositionMarkerFS is a function block designed to move a Virtual Terminal (VT) marker object so that it reflects an arbitrary incoming REAL value. It wraps a single **Q_ChildPosition** instance (ISO 11783-6, Annex F.16) and provides a clean, high-level interface for positioning a marker (e.g., a triangle or other graphical element) within its parent container on an ISOBUS VT display.

The block accepts a structured configuration object (`PositionMarker_S`) containing the marker's object pool identifiers, travel bounds, and center offset. It snapshots this structure once at initialization, then on each request:

1. Offsets the incoming value by the configured center offset.
2. Clamps the result to the marker's minimum/maximum travel range.
3. Converts the clamped value to an integer and forwards it as the marker's X position.
4. Passes through the Y position unchanged.

Clamping events are reported via dedicated status outputs, and all results from the underlying `Q_ChildPosition` service are propagated to the caller.

Interface Structure

Event Inputs

Event	Type	With Variables	Description

`INIT`	EInit	`stObj`, `xScale`	Service initialization. Snapshots the marker configuration and initializes the internal `Q_ChildPosition` instance.
`REQ`	Event	`rValue`	Requests the marker to move to reflect the newly supplied value.

Event Outputs

Event	Type	With Variables	Description
`INITO`	EInit	—	Initialization confirm. Emitted after `INIT` has been successfully processed.
`CNF`	Event	`STATUS`, `s16result`, `xOver`, `xUnder`	Confirmation of a requested move, carrying the service status, return value, and clamp indicators.

Data Inputs

Name	Type	Initial Value	Description
`stObj`	`sitobus::utils::childposition::PositionMarker_S`	—	Marker object pool properties: child ID, parent ID, travel bounds (`r32MinPos`, `r32MaxPos`), center offset (`r32Center`), and fixed Y position (`s16YPosition`). Snapshot once at `INIT`.
`xScale`	`BOOL`	`FALSE`	When `FALSE` (default), X and Y positions are passed through unchanged. When `TRUE`, they are scaled by the DM/SKM factor for the parent object. Passed straight through to the internal `Q_ChildPosition`.
`rValue`	`REAL`	—	The physical value to display. Can originate from any source — a VT input number, a sensor reading, or any other logic.

Data Outputs

Name	Type	Description
`STATUS`	`STRING`	Service status, passed through directly from the internal `Q_ChildPosition` instance.
`s16result`	`INT`	Return value from the internal `Q_ChildPosition`, indicating the outcome of the child-position command.
`xOver`	`BOOL`	Set when the requested value (after adding `r32Center`) exceeded `r32MaxPos` and was clamped.
`xUnder`	`BOOL`	Set when the requested value (after adding `r32Center`) fell below `r32MinPos` and was clamped.

Adapters

This function block does not expose any adapters.

Functionality

The block operates in two distinct phases:

Initialization — When `INIT` is triggered, the input structure `stObj` is copied into an internal snapshot using an `F_MOVE` function block. This snapshot is then used to configure the embedded `Q_ChildPosition`

service with the correct child ID, parent ID, and scale flag. The INITO event confirms successful initialization.

Value Processing — On each REQ event, the following pipeline is executed:

1. **Offset**: The incoming rValue is added to the snapshot's r32Center value using F_ADD. This accounts for any centering offset of the marker within its travel range.
2. **Clamp**: The offset result is passed through F_ClampReal, which restricts the value to the interval [r32MinPos, r32MaxPos]. If clamping occurs, the xOver or xUnder flags are set accordingly.
3. **Conversion**: The clamped REAL value is converted to an INT using F_REAL_TO_INT (truncation).
4. **Positioning**: The converted value is written to s16Xposition of the internal Q_ChildPosition service, along with the snapshot's s16YPosition (passed through unchanged). The service is then requested to move the marker.

The CNF event propagates the STATUS, s16result, xOver and xUnder outputs from the internal service and the clamping stage, giving the caller full visibility into the operation's outcome.

Technical Features

- **ISO 11783-6 compliant** — the wrapped Q_ChildPosition service follows the ISOBUS VT child position semantics (Annex F.16).
- **Snapshot-based configuration** — the PositionMarker_S structure is captured once at INIT, preventing later modifications from corrupting the working state.
- **Integer conversion** — the REAL input is converted to an INT only once, at the point where it is needed by the VT service, minimizing unnecessary conversions.
- **Clamp reporting** — over-range and under-range conditions are explicitly reported, allowing the caller to react to out-of-bounds input values.
- **Scale pass-through** — the xScale flag is forwarded verbatim to the internal Q_ChildPosition, giving direct control over DM/SKM scaling behavior.
- **Fully event-driven** — both initialization and operation follow the strict REQ/CNF and INIT/INITO handshake patterns common to 4diac service interface function blocks.

State Overview

The block does not maintain an explicit state machine but relies on the internal Q_ChildPosition service for its lifecycle. The externally observable states are:

- **Uninitialized** — before INIT has been processed, or after an unsuccessful initialization. A REQ issued in this state will produce undefined behavior and should be avoided.
- **Initialized / Idle** — after INITO is emitted, the block is ready to accept REQ events. It remains idle while no request is being processed.
- **Processing** — between receiving a REQ and emitting the corresponding CNF. During this window, the internal pipeline (add → clamp → convert → move) is active.
- **Clamped** — a sub-state during processing where the value was out of range; indicated by xOver or xUnder being TRUE on the CNF output.

Application Scenarios

- **ISOBUS VT input displays** — Display a numeric value (e.g., from a NumericValue_PHYS object) as a moving marker on a VT screen.
- **Sensor value visualization** — Map a continuous sensor reading (temperature, pressure, position) to the position of a graphical indicator on a machine display.
- **Process control dashboards** — Drive a marker inside a tank, gauge, or scale graphic, where the marker's horizontal position encodes a real-world measurement.
- **Machine automation** — Combine with any REAL-valued process variable to provide immediate visual feedback on a VT, including automatic clamping when limits are exceeded.

Comparison with Similar Blocks

Feature	PositionMarkerFS	Typical raw Q_ChildPosition usage
Input type	Accepts a physical REAL value	Requires an already-formatted INT position
Center offset handling	Built-in via r32Center	Must be handled by the caller

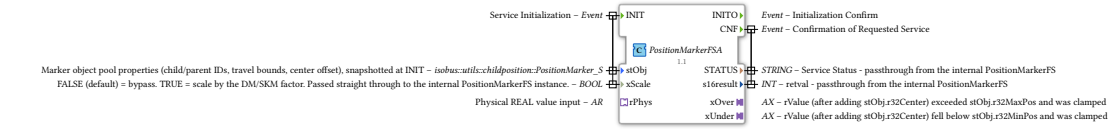
Clamp protection	Automatic, with status flags	Not provided; out-of-range values would corrupt the display
Configuration	Snapshot-based, single `stObj` struct	Requires manual set-up of multiple parameters
Y-position handling	Passed through automatically	Must be managed separately
Scale handling	Optional pass-through `xScale`	Caller must know when to scale

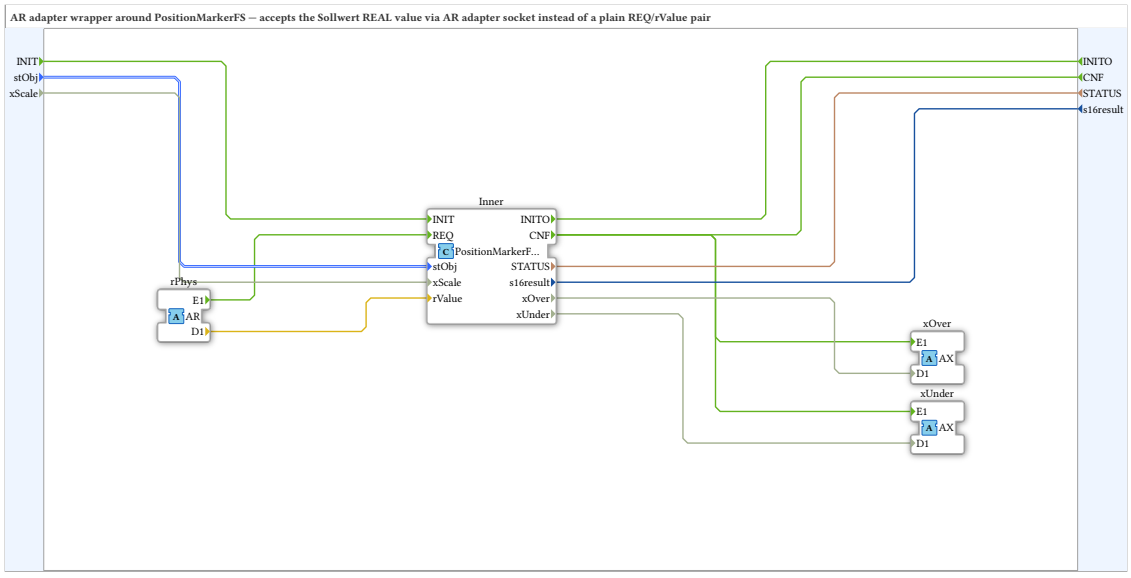
Compared to using `Q_ChildPosition` directly, `PositionMarkerFS` abstracts away the type conversion, clamping, and center offset, resulting in fewer errors at the call site and a clearer data flow. It is more specialized than a generic clamp/scale block but significantly more convenient for VT marker positioning tasks.

Conclusion

PositionMarkerFS bridges the gap between arbitrary REAL-valued process data and precise, clamp-safe VT marker positioning in the ISOBUS environment. By encapsulating the offset, clamping, conversion, and child-position service calls into a single, well-defined interface, it simplifies application code and reduces the risk of marker positioning errors. The block is particularly suited for dashboards, gauges, and any scenario where a physical value must be rendered visually on a Virtual Terminal with predictable behavior at range limits.

PositionMarkerFSA





• — ○ • — ○ • — ○ •

Introduction

PositionMarkerFSA is an adapter wrapper around the `PositionMarkerFS` function block. It accepts a physical REAL value (Sollwert) via an AR (incoming) adapter socket and forwards it as the setpoint to an internally instantiated `PositionMarkerFS`. This design follows the same pattern as `Q_NumericValue_PHYSA` around `Q_NumericValue_PHYS`, re-exposing the over- and under-limit indications through AX (outgoing) adapter plugs instead of distinct event/data ports.

The wrapper handles all communication between the adapter-based interface and the internal FB, relieving the developer from manually connecting the `REQ`/`rValue` pair for each invocation.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Service initialization; snapshots stObj and xScale into the internal PositionMarkerFS instance.

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization confirm – emitted after the internal FB has been successfully initialized.
CNF	Event	Confirmation of a requested service – raised whenever the internal FB confirms a position-rating request. Carries STATUS and s16result as data.

Data Inputs

Data	Type	Comment
stObj	forte::isobus::utils::childposition::PositionMarkerObj	Marker object pool properties: child/parent IDs, travel bounds, and center offset. This structure is captured at INIT and passed unchanged to the internal FB.
xScale	BOOL	Scaling control: FALSE (default) bypasses the DM/SKM scaling factor; TRUE enables it. This flag is forwarded to the internal PositionMarkerFS.

Data Outputs

Data	Type	Comment
STATUS	STRING	Service status – a direct passthrough from the internal PositionMarkerFS.
s16result	INT	Return value – a direct passthrough from the internal PositionMarkerFS.

Adapters

Adapter	Direction	Type	Comment
rPhys	Socket (Input)	adapter::types::unidirectional::AR	Physical REAL value input – the Sollwert that drives the internal FB.
xOver	Plug (Output)	adapter::types::unidirectional::AX	Signals when the actual position (after adding stObj.r32Center) exceeds stObj.r32MaxPos and is clamped.
xUnder	Plug (Output)	adapter::types::unidirectional::AX	Signals when the actual position falls below stObj.r32MinPos and is clamped.

Functionality

PositionMarkerFSA encapsulates the complete data and event flow of a position marker calculation. On INIT, it passes the stObj and xScale values to the internal PositionMarkerFS instance, which initializes its internal state. Later, each incoming event from the rPhys adapter (via its internal E1 output) triggers the REQ event of the internal FB, simultaneously transferring the physical value (rPhys.D1) as the new setpoint (rValue). The internal FB performs the marker position analysis and responds with a CNF event.

The wrapper forwards the INIT0 and CNF events to its own outputs, and also re-emits the CNF event to both adapter plugs xOver and xUnder so that limit violations can be propagated asynchronously. The data outputs STATUS and s16result are copied directly from the internal FB.

Technical Features

- **Adapter-based I/O**: The input (physical value) and outputs (limit flags) are provided using unidirectional AR/AX adapter types, making the block easy to connect in CAN-based environments where such adapter patterns are common.
- **Single internal FB instance**: The block instantiates exactly one PositionMarkerFS, keeping the implementation lean and preserving all its original semantics.
- **Passthrough of status and result**: STATUS and s16result are forwarded without modification, allowing the host system to inspect detailed results.
- **Scaling control**: The xScale flag is passed straight through to the internal FB, giving developers the option to enable or disable scaling without altering the adapter interface.
- **Automatic event wiring**: The adapter's event output (E1) is directly connected to the internal REQ event; the internal CNF event is distributed to both the main CNF output and the two adapter plugs.

State Overview

PositionMarkerFSA does not introduce its own state machine. Instead, it delegates all state handling to the internal PositionMarkerFS. Therefore, the state behaviour matches that of the underlying block:

- **Initialization phase**: After INIT, the internal FB is configured with the marker object pool and scaling flag. It then emits INIT0.
- **Request/confirmation phase**: Each request via rPhys (or the corresponding adapter event) leads to a calculation cycle. On completion, the internal FB emits CNF and updates STATUS and s16result. Additionally, if the calculated position exceeds or falls below the given limits, the corresponding adapter plug (xOver/xUnder) is activated.

Application Scenarios

PositionMarkerFSA is best suited for systems that use adapter-based communication, particularly in agricultural or mobile machinery applications following the ISO 11783 (ISOBUS) standard. Common use cases include:

- Integrating a position marker into a CAN network where the setpoint is transmitted as a physical REAL value and the limit flags are expected as separate adapter-based outputs.
- Replacing older designs that used plain REQ/rValue input pairs with a more structured adapter interface, without changing the core calculation logic.
- Using the xScale flag to conditionally apply scaling factors, making the block adaptable to different machine configurations.

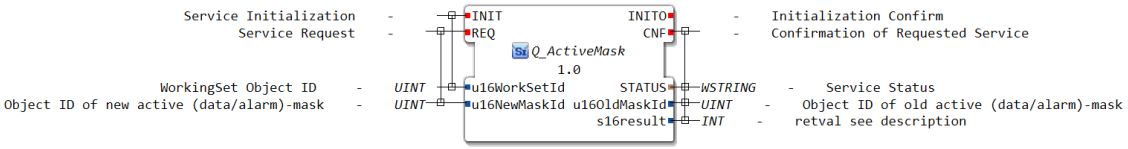
Comparison with Similar Blocks

- **PositionMarkerFS** – This is the underlying FB, which provides a plain REQ/rValue interface and separate xOver/xUnder data outputs. PositionMarkerFSA wraps it to present an adapter-based interface, simplifying integration in adapter-friendly networks.
- **Q_NumericValue_PHYSA** – A similar wrapper pattern that wraps Q_NumericValue_PHYS; PositionMarkerFSA follows the same design philosophy but is specifically tailored for the position marker logic.
- **PositionMarker_S** – This is a data structure used as input, not an FB itself. In comparison, PositionMarkerFSA is a complete function block that consumes PositionMarker_S and produces status and limit indications.

Conclusion

PositionMarkerFSA provides a clean, adapter-oriented interface for driving a position marker calculation. By wrapping the established PositionMarkerFS FB, it preserves all functional behaviour while improving integration flexibility in adapter-based systems. The clear separation of the input (physical value), outputs (status, result, and limit flags), and the optional scaling control makes it a valuable building block for ISOBUS-compliant implementations.

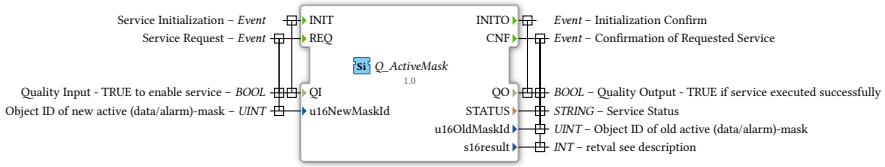
Q_ActiveMask



• — ○ • — ○ • — ○ •

Introduction

The **Q_ActiveMask** is a standards-compliant function block for controlling active masks in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.34) specification for agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request
- `REQ`: Screen Change Request (with parameters)

Event Outputs

- `INITO`: Initialization Confirmation
- `CNF`: Screen Change Confirmation (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16NewMaskId` (UINT): Object ID of the new screen

!!! Note: “WorkingSet Object ID fixed to 0”

The **WorkingSet Object ID** (`u16WorkSetId`) is **fixed to 0** by the authors of logiBUS®. The user must ensure that the WorkingSet object in the object pool always has Object ID 0. This is the default setting in most tools:

- ISO-Designer from Bucher Automation AG
- Isobus Studio from to-the-future / Tobias Tenberg

Since the working set is a single object and only exists once, this limitation is not a problem.

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operating status message
- `u16OldMaskId` (UINT): Object ID of the previous screen
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16NewMaskId` — valid mask object types (Annex F.34): Data Mask (1000–1999), Alarm Mask (2000–2999).

ID_NULL (65535) is not a valid command target for the new active mask. Any ID outside these ranges is invalid for commanding.

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` event starts the function block
- `INITO` confirms successful setup

1. **Mask Change**:

- Trigger `REQ` with new mask parameters
- Return `CNF` result and previous mask ID

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.34)
✓ **Deterministic** behavior ✓ **Multi-client capable** architecture ✓ **Real-time capable** execution

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Success
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	Invalid State
−21	VT_E_NO_INSTANCE	No VT Instance

Application Scenarios

- **Tractor Controls**: Operating Mode Switching
- **Harvest Monitoring**: Data Acquisition Masks
- **Diagnostic Systems**: Fault Display Masks
- **Multi-Terminal Operation**: Synchronized Displays

⚖️ Comparison with Similar Function Blocks

Feature	Q_ActiveMask	VtMaskManager	VtDynamicDisplay
ISO Standard	✓	✖	✖
Error Codes	Standard	Manufacturer	Partial
Condition Management	Full	Basic	Advanced

🛠️ Related Exercises

- Exercise_019
- Exercise_019a
- Exercise_019b
- Exercise_019c

Conclusion

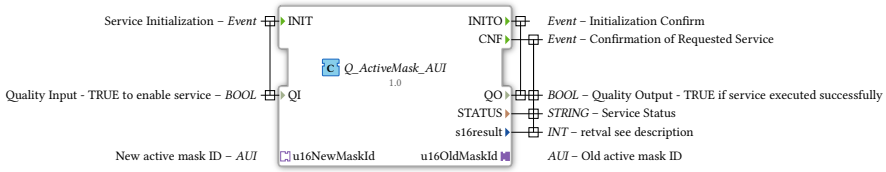
The Q_ActiveMask block provides the reference implementation for ISOBUS mask switching:

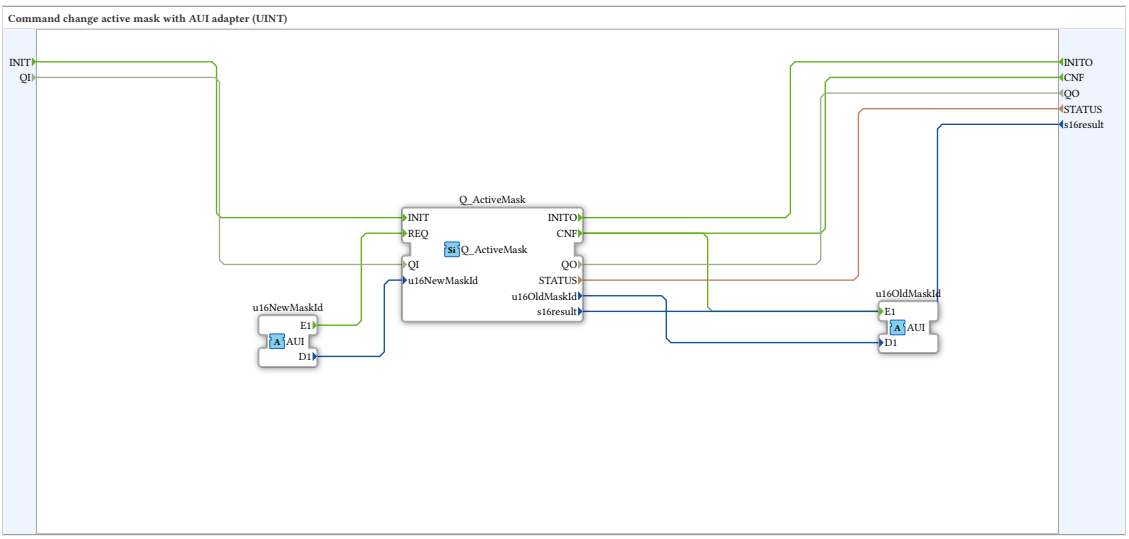
- **Standard-compliant**: Full ISO 11783-6 compatibility
- **Robust**: Proven technology in series production
- **Flexible**: Supports complex display scenarios

Essential for:

- Manufacturers of ISOBUS terminals
- Developers of agricultural control units
- System integrators in agricultural technology

Q_ActiveMask_AUI





• – ○ • – ○ • – ○ •

Introduction

The **Q_ActiveMask_AUI** function block is an AUI (Adapter User Interface) wrapper for the core block **Q_ActiveMask**. It provides a convenient, unidirectional adapter–based interface to change the active mask on an ISO 11783-6 (ISOBUS) Virtual Terminal. The block encapsulates the functionality of the underlying service, translating adapter signals into the internal request/confirm protocol of the wrapped block.

This wrapper is designed for applications that already use the unidirectional AUI adapter type for data exchange, simplifying integration into existing adapter-based networks. It handles the complete command sequence to change the active mask, returning both the status of the operation and a result code.

Interface Structure

Event Inputs

Event	Type	Comment
`INIT`	EInit	Service Initialization (with `QI`)

Event Outputs

Event	Type	With Variables	Comment
`INITO`	EInit	`QO`	Initialization Confirm
`CNF`	Event	`QO`, `STATUS`, `s16result`	Confirmation of Requested Service

Data Inputs

Data Name	Type	Comment
`QI`	BOOL	Quality input, `TRUE` enables the service (default `TRUE`)

All other input data is exchanged via the adapters.

Data Outputs

Data Name	Type	Comment
QO	BOOL	Quality output, TRUE if the service was executed successfully
STATUS	STRING	Service Status
s16result	INT	Retval – see description in the core block

Adapters

Direction	Name	Type	Comment
Socket	u16NewMaskId	adapter::types::unidirectional::AUI	New active mask ID
Plug	u16OldMaskId	adapter::types::unidirectional::AUI	Old active mask ID

The adapter data (D1) carries a 16-bit unsigned integer (UINT) representing the mask ID. The AUI adapter provides a single event (E1) on both sides to trigger the transfer.

Functionality

The block acts as a bridge between an AUI adapter network and the internal Q_ActiveMask function block. When the **u16NewMaskId** socket receives a new value via its E1 event, the wrapper internally issues a REQ to the Q_ActiveMask block, supplying the new mask ID. After processing, the Q_ActiveMask block produces a CNF event, which is propagated to the output as CNF, together with the STATUS and s16result values. Additionally, the old mask ID is sent back through the **u16OldMaskId** plug using its E1 event, and the corresponding data value is transferred.

Initialization is handled by the INIT event, which is forwarded to the internal block, and INITO is generated once the initialization completes. QI is passed through to Q_ActiveMask.QI, and Q_ActiveMask.QO is passed through to QO.

This wrapper does not implement the service logic itself; it relies entirely on the Q_ActiveMask function block for the actual ISO 11783-6 command (Part 6, F.34). The wrapper only adapts the data flow to and from the AUI interfaces.

Technical Features

- **AUI Adapter Compatible**: Uses the standard unidirectional AUI adapter type, allowing direct connection to other adapter-based components.
- **Seamless Integration**: The internal Q_ActiveMask block is fully encapsulated; only the adapter interface is exposed.
- **Error and Status Reporting**: Provides textual status (STATUS) and a numeric result (s16result) on every confirmation.
- **Event-Driven**: All operations are triggered by incoming events (init, new mask ID), making it suitable for cyclic or event-driven control systems.
- **Platform Independent**: The block is written in standard IEC 61499 and can be used in any 4diac-compatible runtime.

State Overview

The block does not maintain an explicit state machine of its own; it inherits the behaviour of the wrapped Q_ActiveMask block. The internal block typically operates in the following states:

1. **Idle**: Waiting for a new mask ID command.
2. **Busy**: Processing a request (between REQ and CNF).
3. **Completed**: A CNF has been issued, the block returns to idle.

The wrapper only forwards events and data accordingly. The INIT event initialises the internal block, and INITO confirms successful initialisation.

Application Scenarios

- **ISOBUS Virtual Terminal Control**: Use this block in agricultural machinery to change the currently displayed working set or main mask on a VT.
- **Adapter-Based Architectures**: In systems where all external communication is performed via AUI adapters, this wrapper eliminates the need to expose raw event/data ports.

- **Multi-Mask Management**: Combine multiple instances to manage several mask IDs or to switch between different screens based on operational state.
- **Prototyping and Simulation**: The wrapper simplifies testing by providing a clean adapter interface that can be easily driven by simulation models or other adapters.

Comparison with Similar Blocks

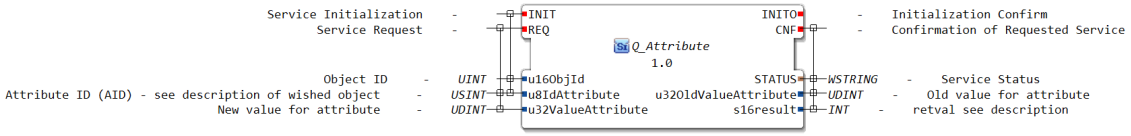
- **Q_ActiveMask (core block)**: The core block provides separate event and data inputs/outputs. The AUI wrapper abstracts those into a single adapter pair, reducing wiring complexity, but adds an extra layer of indirection.
- **Q_ActiveMask_AUI vs. other adapter wrappers**: Many ISOBUS commands have similar AUI wrappers (e.g., `Q_SetBackgroundColor_AUI`). They all follow the same pattern – a socket for the "new" data, a plug for the "old" data, and common event outputs.

The main advantage of this wrapper is that it fits directly into adapter-based communication frameworks, promoting modular design and reusability. The disadvantage is a slight increase in block count and the need to manage two adapter connections.

Conclusion

The **Q_ActiveMask_AUI** function block provides a pragmatic and clean solution for changing the active mask in an ISOBUS VT using unidirectional AUI adapters. It encapsulates the underlying `Q_ActiveMask` logic, exposes a minimal and well-defined interface, and integrates smoothly into event-driven industrial control systems. Its use of standard AUI types ensures high compatibility with existing adapter-oriented designs, making it a valuable component for ISOBUS applications.

Q_Attribute



• — ○ • — ○ • — ○ •

Introduction

The **Q_Attribute** is a standards-compliant function block for modifying VT object properties, developed under the EPL 2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.38) specification for agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object ID and attribute ID)
- `REQ`: Attribute Change Request (with new value)

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Change Acknowledgement (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Object Identifier
- `u8IdAttribute` (USINT): Attribute Identifier (AID)
- `u32ValueAttribute` (UDINT): New Attribute Value

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operational status message
- `u32OldValueAttribute` (UDINT): Previous attribute value
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` (the object whose attribute is changed) and `u8IdAttribute` (the Attribute ID, AID) are subject to **different** validity rules.

`u16ObjId` — valid object types (Annex F.38): Data Mask (1000–1999), Alarm Mask (2000–2999), Soft Key Mask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Auxiliary Function Type 2 (31000–31999).

`u8IdAttribute` (AID): Valid attribute IDs are object-type specific and defined in Annex B, in the attribute tables of each object type. An AID with no attribute assigned is rejected by the VT ("Invalid Attribute ID", F.39).

ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`.

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with object and attribute ID
- `INITO` confirms operational readiness

1. **Attribute Change**:

- `REQ` triggers with new value
- `CNF` returns result and old value

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 Compliant** (F.38) ✓ **Deterministic** Execution ✓ **32-bit Value Support** ✓ **Real-time** Processing

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Success
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	Invalid State
−21	VT_E_NO_INSTANCE	No VT Instance

Application Scenarios

- **Display Settings**: Visibility/Position of VT Objects
- **Machine Configuration**: Adjusting Operating Parameters
- **Diagnostic Systems**: Controlling Debug Information
- **Multi-Terminal Operation**: Consistent Object Attributes

⚖️ Comparison with Similar Blocks

Feature	Q_Attribute	VtPropertySet	VtDynamicConfig
ISO Standard	✓	✖	✖

Value Range	32-bit	16-bit	32-bit
State Management	Full	Partial	Extended

Conclusion

The Q_Attribute block provides the reference implementation for ISOBUS attribute control:

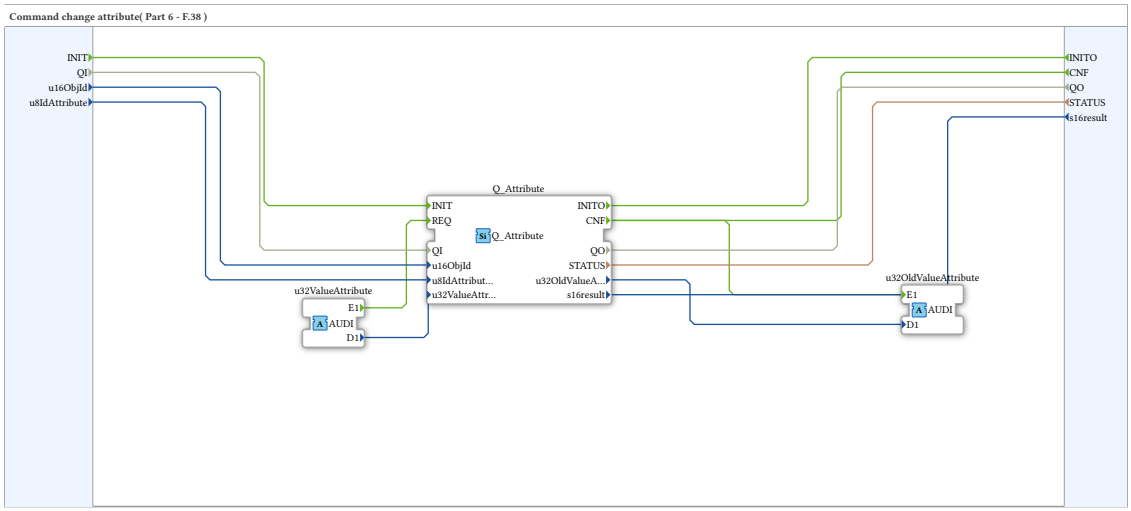
- **Standard-compliant**: Full ISO 11783-6 compatibility
- **High-performance**: 32-bit value support
- **Reliable**: Proven technology in field devices

Essential for:

- VT object controllers
- Dynamic display configurations
- Machine parameter adapters

Q_Attribute_AUDI





• – ○ • – ○ • – ○ •

Introduction

The function block **Q_Attribute_AUDI** implements the “Change Attribute” command of the ISOBUS standard ISO 11783-6 (Part 6 – F.38). It allows you to change an attribute of an object that is identified by a unique attribute ID (AID). The block supports passing the new attribute value via an adapter interface and returns the old value via an adapter as well. String attributes cannot be changed with this command.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initializes the service and starts communication with the ISOBUS VT

Event Outputs

Event	Type	Description

INITO	EInit	Confirms successful completion of initialization
CNF	Event	Confirms execution of the requested attribute change command

Data Inputs

Variable	Type	Description
QI	BOOL	Quality Input - TRUE to enable service
u16ObjId	UINT	Object ID of the attribute to be changed (initial value: ID_NULL)
u8IdAttribute	USINT	Attribute ID (AID) – defines which attribute of the object is to be changed

Data Outputs

Variable	Type	Description
QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Status message of the executed service
s16result	INT	Return value of the command (error codes, see Functionality)

Adapters

Adapter	Direction	Type	Description
u32ValueAttribute	Socket	unidirectional::AUDI	New value for the attribute (32-bit, fed into the FB via the adapter)
u32OldValueAttribute	Plug	unidirectional::AUDI	Old attribute value (32-bit, output by the function block via the adapter)

Quality signals QI/QO: The data input `QI` (BOOL, initial value `TRUE`) is associated with the event `INIT` and is passed through to `QI` of the internal block `Q_Attribute`. The data output `QO` (BOOL) is taken over from `Q_Attribute.QO` and is associated with the events `INITO` and `CNF`.

Valid Object IDs

`u16ObjId` (the object whose attribute is changed) and `u8IdAttribute` (the Attribute ID, AID) are subject to **different** validity rules.

`u16ObjId` — valid object types (Annex F.38): Data Mask (1000–1999), Alarm Mask (2000–2999), Soft Key Mask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Auxiliary Function Type 2 (31000–31999).

`u8IdAttribute` (AID): Valid attribute IDs are object-type specific and defined in Annex B, in the attribute tables of each object type. An AID with no attribute assigned is rejected by the VT ("Invalid Attribute ID", F.39).

ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`.

Functionality

1. **Initialization**: After a rising edge at **INIT**, the service is initialized. The required parameters (object ID and attribute ID) are read from the data inputs.
2. **Command Execution**: As soon as the event output **E1** is triggered at socket **u32ValueAttribute** (e.g., by a connected function block providing a new value), the attribute change command is started. The new attribute value is adopted by the adapter.
3. **Feedback**: After the command is completed, the output **CNF** is activated. The STATU string and the return value **s16result** contain the result. Simultaneously, the previously stored old attribute value is signaled at plug **u32OldValueAttribute** via the event output **E1** and made available via **D1**.

The function block delegates the actual ISOBUS communication to the internal function block **Q_Attribute**, which performs the necessary protocol steps.

Technical Features

- **Error Codes** (output via s16result):
- `VT_E_NO_ERR (0)` – no error
- `VT_E_OVERFLOW (-6)` – buffer overflow
- `VT_E_NOACT (-8)` – command not possible in the current state
- `VT_E_NO_INSTANCE (-21)` – no VT client instance available
- `VT_E_ISO_INSTANCE_INVALID (-129)` – invalid connection instance
- `VT_E_NOT_ALIVE (-130)` – instance valid, but VT not active
- **Adapter Interfaces**: The use of unidirectional adapters (AUDI) enables loose coupling between the encoder and the function block. The old and new values can thus be managed in separate function blocks (FBs).
- **Limitations**: The command does not support string attributes.

State Overview

The FB does not have an explicitly displayed state machine. The sequence control is event-driven:

- After **INIT**, it transitions to a ready state.
- Another event is only triggered by the connected adapter socket (E1 of u32ValueAttribute). After processing, **CNF** is sent.
- The function block is **stateless** in the sense of a reusable sequence; each request is processed individually.

Application Scenarios

- **ISOBUS VT Applications**: Changing parameters of a virtual terminal (e.g., background color, label, visibility) at runtime.
- **Control of Agricultural Equipment**: Dynamically adjusting attributes such as machine settings or diagnostic data.
- **Implementation of ISO 11783-6 Workouts**: Replicating the “Change Attribute” command structure from the standard.

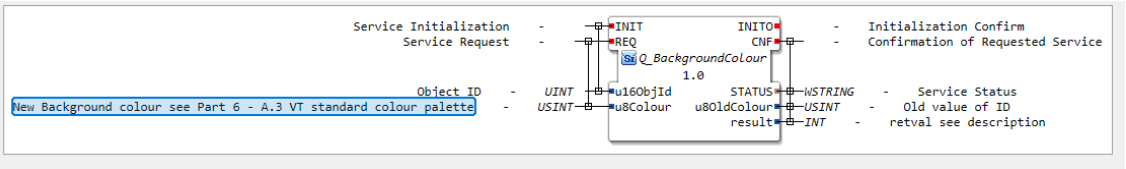
Comparison with Similar Function Blocks

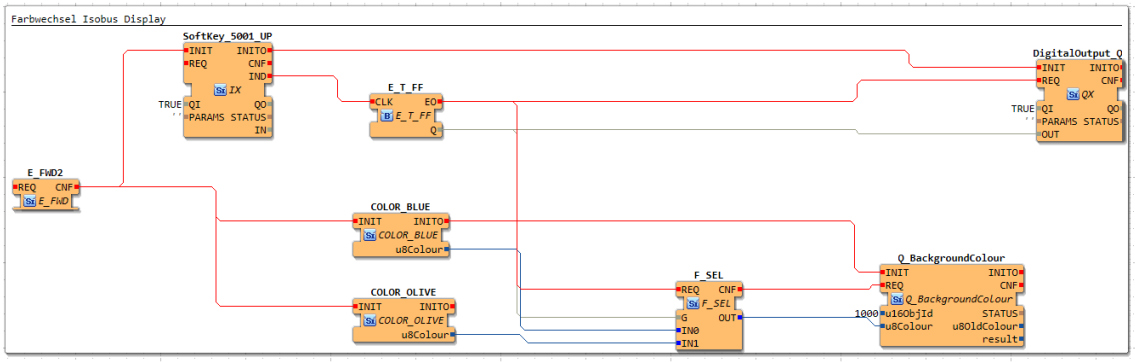
- **Q_Attribute** (without AUDI): Offers the same core functionality, but without adapter interfaces. The new value must be passed directly as a data input, and no feedback of the old value is provided. **Q_Attribute_AUDI** extends this function block with flexible, adapter-based value passing and the return of the old value, increasing reusability and encapsulation.

Conclusion

Q_Attribute_AUDI is a specialized function block for ISOBUS applications that enables the secure and standards-compliant modification of attributes. The adapter-based interface for attribute values ensures high modularity, and the output of the old value allows for easy change tracking. The function block is ideal for complex VT implementations where values originate from different sources or are managed in separate logic units.

Q_BackgroundColour

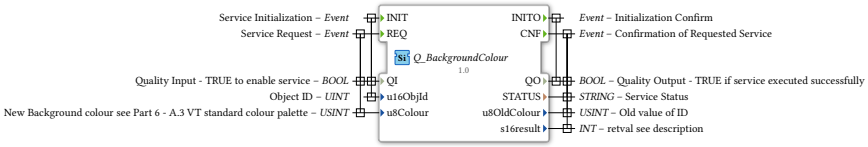




• — ○ • — ○ • — ○ •

Introduction

The **Q_BackgroundColour** is a standards-compliant function block for controlling background colors in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.20) specification for agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object ID)
- `REQ`: Color Change Request (with color code)

Event Outputs

- `INITO`: Initialization Confirmation
- `CNF`: Change Confirmation (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Object Identifier
- `u8Colour` (USINT): New Color Value (ISO 11783-6 palette)

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operating status message
- `u8OldColour` (USINT): Previous color value
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` — valid object types (Annex F.20, objects with background colour attribute): WorkingSet (0), Data Mask (1000–1999), Alarm Mask (2000–2999), Soft Key Mask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Output String Field (11000–11999), Output Number Field (12000–12999), Window Mask (34000–34999), Graphics Context (36000–36999), Auxiliary Function (29000–29999), Auxiliary Function Type2 (31000–31999), Auxiliary Input Type2 (32000–32999).

ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`. Any ID outside these ranges is invalid for commanding.

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with object ID
- `INITO` confirms operational readiness

1. **Color Change**:

- `REQ` triggers with new color code (0-255)
- `CNF` returns result and previous color value

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.20 + A.3 color palette) ✓ **8-bit color support** (256 values) ✓ **Real-time processing** ✓ **State preservation** (u8OldColour)

Standard color palette (excerpt)

Code	Color	RGB equivalent
0	Black	#000000
1	Red	#FF0000
2	Green	#00FF00
3	Yellow	#FFFF00
4	Blue	#0000FF

Return codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Success
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	Invalid State
−128	VT_E_HANDLE_INVALID	Invalid Object ID

Application Scenarios

- **Machine Status**: Color-coded Alarm Indication
- **Night Mode**: Dark Color Palettes
- **Markings**: Highlighting Objects
- **Branding**: Manufacturer-Specific Colors

⚖️ Comparison with Similar Function Blocks

Feature	Q_BackgroundColour	VtColourManager	VtDynamicStyle
ISO Standard	✓	✖	✖
Color Palette	Standard	Extended	User Defined
Object Binding	Explicit	Implicit	Stylesheet

🛠️ Related Exercises

- Exercise_016
- Exercise_016a

Conclusion

The Q_BackgroundColour block provides the reference implementation for ISOBUS color control:

- **Standard-compliant**: Full ISO 11783-6 compatibility
- **Easy integration**: Clear color palette definition
- **Robust**: Proven field reliability

Essential for:

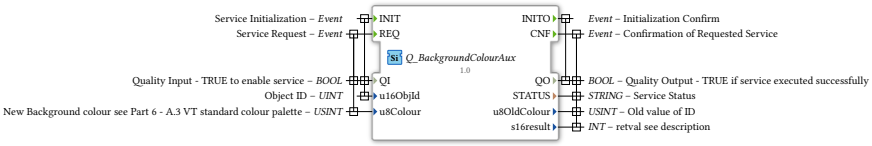
- Color status indicators
- User interface design
- Dynamic visualizations

Q_BackgroundColourAux

• – ◦ • – ◦ • – ◦ •

Introduction

The **Q_BackgroundColourAux** is a standards-compliant function block for controlling secondary background colors in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.20) specification for auxiliary colors in agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object ID)
- `REQ`: Color Change Request (with color code)

Event Outputs

- `INITO`: Initialization Confirmation
- `CNF`: Change Confirmation (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Object Identifier
- `u8Colour` (USINT): New Auxiliary Color Value (ISO 11783-6 palette)

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operational status message
- `u8OldColour` (USINT): Previous auxiliary color value
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` — valid object types (Annex F.20, objects with background colour attribute): WorkingSet (0), Data Mask (1000–1999), Alarm Mask (2000–2999), Soft Key Mask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Output String Field (11000–11999), Output Number Field (12000–12999), Window Mask (34000–34999), Graphics Context (36000–36999), Auxiliary Function (29000–29999), Auxiliary Function Type2 (31000–31999), Auxiliary Input Type2 (32000–32999).

ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`. Any ID outside these ranges is invalid for commanding.

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with object ID for auxiliary element
- `INITO` confirms operational readiness

1. **Color Change**:

- `REQ` triggers with new auxiliary color code (0-255)
- `CNF` returns result and previous color value

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.20 (for auxiliary colors)
✓ **8-bit color support** (256 values)
✓ **Compatible with Q_BackgroundColour**
✓ **State preservation** (u8OldColour)

Standard auxiliary colors (excerpt)

Code	Scope	Typical color
16	Status bar	Blue
32	Secondary areas	Gray
48	Guidelines	Light blue

Return codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Success
−128	VT_E_HANDLE_INVALID	Invalid object ID
−129	VT_E_ISO_INSTANCE_INVALID	Invalid instance

Application Scenarios

- **Status Indicators**: Secondary colors for bars
- **Groupings**: Color separation of areas
- **Editor Modes**: Guidelines in configurators
- **Diagnostics**: Additional information

⚖️ **Comparison with similar building blocks**

Feature	Q_BackgroundColourAux	Q_BackgroundColour	VtAuxColour
ISO Standard	✓	✓	✖
Color range	Auxiliary elements	Primary elements	All
Usage	Secondary	Primary	Universal

Conclusion

The Q_BackgroundColourAux module extends ISOBUS color control for auxiliary elements:

- **Specialized**: Optimized for secondary display elements
- **Consistent**: Same API as Q_BackgroundColour
- **Proven in practice**: Used in modern tractor displays

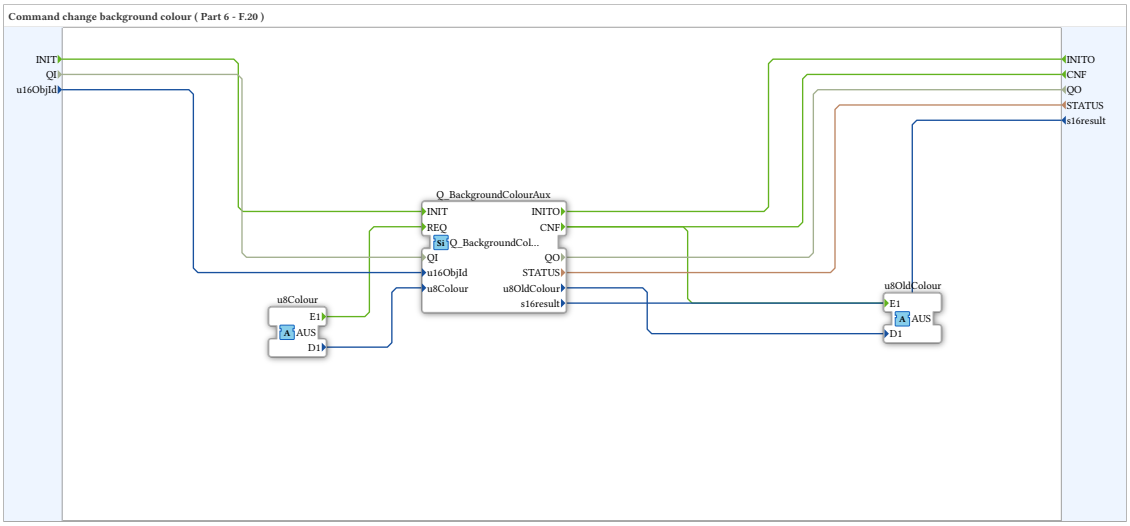
Ideal for:

- Complex display layouts

- Multi-layered visualizations
- Systems with advanced color management

Q_BackgroundColourAux_AUS





• — ○ • — ○ • — ○ •

Introduction

This function block implements the command to change the background color of an object according to ISO 11783-6, Part F.20. It enables interaction with the Virtual Terminal (VT) and handles the transmission of the new color as well as the feedback of the result.

Interface Structure

Event Inputs

Event	Type	With Parameter	Description
INIT	EInit	QI, u16ObjId	Service Initialization

Event Outputs

Event	Type	With Parameter	Description
INITO	EInit	QO	Initialization confirmed
CNF	Event	QO, STATUS, s16result	Confirmation of requested service

Data Inputs

Name	Type	Initial Value	Description
QI	BOOL	TRUE	Quality Input - TRUE to enable service
u16ObjId	UINT	ID_NULL	Object ID of the affected object

Data Outputs

Name	Type	Description

QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Service status (e.g., error message)
s16result	INT	Return value (see below)

Adapter

Direction	Name	Type	Description
Socket	u8Colour	OFF	New background color (according to ISO 11783-6 A.3)
Plug	u8OldColour	OFF	Old background color (returned by the VT)

Quality signals QI/QO: The data input `QI` (BOOL, initial value `TRUE`) is associated with the event `INIT` and is passed through to `QI` of the internal block `Q_BackgroundColourAux`. The data output `QO` (BOOL) is taken over from `Q_BackgroundColourAux.QO` and is associated with the events `INITO` and `CNF`.

Valid Object IDs

`u16ObjId` is valid for the same object types as the wrapped base block `Q_BackgroundColourAux` / `Q_BackgroundColour` (Annex F.20, objects with background colour attribute): WorkingSet (0), Data Mask (1000–1999), Alarm Mask (2000–2999), Soft Key Mask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Output String Field (11000–11999), Output Number Field (12000–12999), Window Mask (34000–34999), Graphics Context (36000–36999), Auxiliary Function (29000–29999), Auxiliary Function Type2 (31000–31999), Auxiliary Input Type2 (32000–32999).

ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`.

Functionality

The module first expects initialization with the object ID (`INIT` with `u16ObjId`).

As soon as a new color value is present at the socket adapter `u8Colour` (triggered by the adapter's event `E1`), an internal event `REQ` is sent to the encapsulated function block `Q_BackgroundColourAux`.

This executes the VT command and returns the status (`STATUS`) and the result (`s16result`) via the output `CNF`.

The previous color is output via the plug adapter `u8OldColour`.

All the logic is encapsulated in a subordinate function block of the same family, so this block serves only as an outer shell for the adapter coupling.

This function block executes the VT command and returns the status (`STATUS`) and the result (`s16result`) via the output `CNF`.

Technical Features

- The initial value for `u16ObjId` is `ID_NULL`, meaning the function block must be initialized with a valid object ID before use.
- Color values are transmitted as 8-bit values via the AUS adapter. The valid colors are defined in the VT standard color palette (ISO 11783-6 Annex A.3).
- The initial value for `u16ObjId` is `ID_NULL`, meaning the function block must be initialized with a valid object ID before use.
- The color values are transmitted as 8-bit values via the AUS adapter. The valid colors are defined in the VT standard color palette (ISO 11783-6 Annex A.3). - Possible return values (`s16result`):
- `VT_E_NO_ERR` (0): Success
- `VT_E_OVERFLOW` (-6): Buffer overflow
- `VT_E_NOACT` (-8): Command not possible in the current state
- `VT_E_NO_INSTANCE` (-21): No VT client available
- `VT_E_ISO_INSTANCE_INVALID` (-129): Invalid connection identifier
- `VT_E_NOT_ALIVE` (-130): VT instance valid, but VT not reachable

State Overview

The FB does not have explicit state machines in the XML. Event control is handled solely via event connections:

1. **INIT** → Initialization of the service.

2. **External event via socket adapter** ⟶ Start of the color change request.
3. **CNF** ⟶ Completion of the operation with status and return value.

Application Scenarios

- Dynamic adjustment of the background color of an ISOBUS VT object (e.g., button, text field) depending on operating states or user input.
- Implementation of color changes in agricultural control and visualization systems.

Comparison with Similar Function Blocks

Analogous function blocks exist for other visual properties (e.g., `Q_TextColourAux_AUS`, `Q_VisibilityAux_AUS`).

All of them share the schema-based adapter pattern, which implements the signaling of value changes via OFF adapters.

This function block specializes in background color and uses the same error and return values.

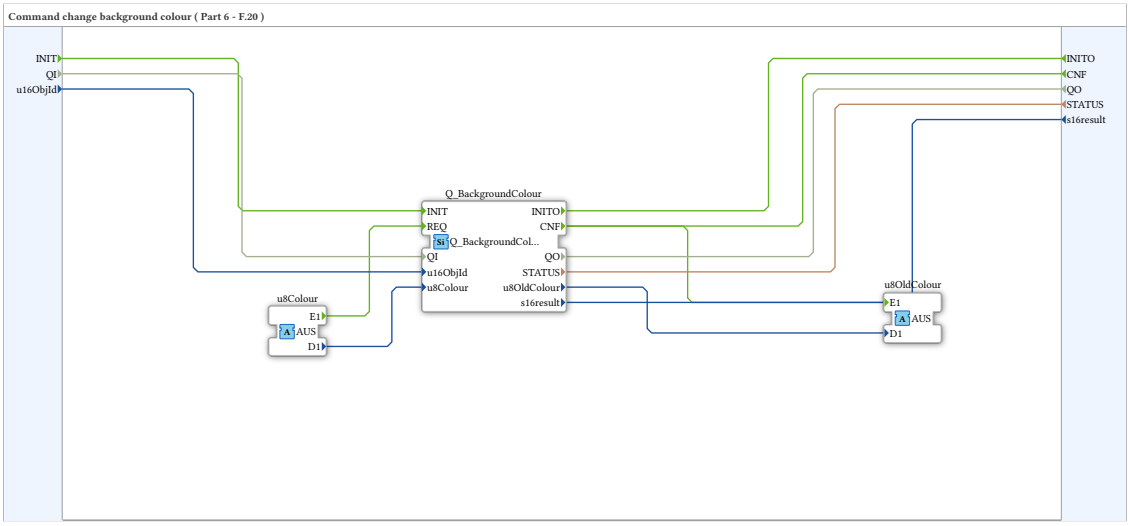
Conclusion

The `Q_BackgroundColourAux_AUS` function block compactly encapsulates an ISOBUS-VT service for background color change and offers easy integration into higher-level controllers via its adapter interfaces.

It is reliable, standards-compliant, and allows for robust error handling through status feedback.

Q_BackgroundColour_AUS





• – ○ • – ○ • – ○ •

Introduction

The function block **Q_BackgroundColour_AUS** is used to change the background color of an object in the ISOBUS Virtual Terminal (VT). It encapsulates the corresponding command interface according to ISO 11783-6 (Part 6 – F.20). The block is supplied with the new color via a socket and returns the previous color via a plug. The actual color change is performed by the internal block `Q_BackgroundColour`.

Interface Structure

Event Inputs

Name	Type	Comment	With Variables
`INIT`	EInit	Service Initialization	`QI`, `u16ObjId`

Event Outputs

Name	Type	Comment	With Variables
INITO	EInit	Initialization Acknowledgement	QO
CNF	Event	Acknowledgement of Requested Service	QO, STATUS, s16result

Data Inputs

Name	Data Type	Initial Value	Comment
QI	BOOL	TRUE	Quality Input - TRUE to enable service
u16ObjId	UINT	ID_NULL	Object ID of the object to be modified

Data Outputs

Name	Data Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Service status (e.g., error message or “OK”)
s16result	INT	Return value (see description – error codes)

Adapter

- **Socket** u8Colour

Type: adapter::types::unidirectional::AUS
Returns the new background color (according to ISO 11783-6, A.3 VT standard color palette).
The adapter provides both the event E1 (to trigger the color change) and the data value D1 (color value).

- **Plug** u8OldColour

Returns the previous background color after the change command has been executed.
The event E1 signals that the value is valid; the data value D1 contains the old color.
Quality signals QI/QO: The data input QI (BOOL, initial value TRUE) is associated with the event INIT and is passed through to QI of the internal block Q_BackgroundColour. The data output QO (BOOL) is taken over from Q_BackgroundColour.QO and is associated with the events INITO and CNF.

Valid Object IDs

u16ObjId is valid for the same object types as the wrapped base block Q_BackgroundColour (Annex F.20, objects with background colour attribute): WorkingSet (0), Data Mask (1000–1999), Alarm Mask (2000–2999), Soft Key Mask (4000–4999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Output String Field (11000–11999), Output Number Field (12000–12999), Window Mask (34000–34999), Graphics Context (36000–36999), Auxiliary Function (29000–29999), Auxiliary Function Type2 (31000–31999), Auxiliary Input Type2 (32000–32999).
ID_NULL (65535) is not a command target but deactivates the FB when used with INIT.

Functionality

1. **Initialization**

An event INIT starts the function block. The passed object ID (u16ObjId) is forwarded to the internal function block Q_BackgroundColour and initialized there. After successful initialization, INITO is output.

1. **Triggering a Color Change**

As soon as an event arrives at socket u8Colour (via the adapter path E1), the internal function block is triggered with REQ. The color value (u8Colour.D1) is passed to Q_BackgroundColour.u8Colour.

1. **Feedback**

After the command is completed, the internal module sends a CNF event.

- The outputs STATUS and s16result are set accordingly and are valid at the event output CNF.

- Simultaneously, the previous color value (data `D1`) and an acknowledgment event (`E1`) are output via the plug `u8OldColour`.
-

Technical Features

- The module implements the “Change Background Colour” command according to ISO 11783-6, section F.20.
- The color values correspond to the VT standard color palette from Annex A.3 of the standard.
- Possible return values (`s16result`) are:
- `VT_E_NO_ERR (0)` – Success
- `VT_E_OVERFLOW (-6)` – Buffer overflow
- `VT_E_NOACT (-8)` – Command not possible in the current state
- `VT_E_NO_INSTANCE (-21)` – No VT client available
- `VT_E_ISO_INSTANCE_INVALID (-129)` – Invalid connection identifier
- `VT_E_HANDLE_INVALID (-128)` – Invalid handle
- `VT_E_NOT_ALIVE (-130)` – VT not reachable
- The function block uses a unidirectional adapter (`AUS`) that combines the event and data channels – this reduces the number of interfaces.

State Overview

The function block (FB) does not have an explicit state machine at the top level, but the interaction with `INIT`, `REQ` (via adapter), and `CNF` results in the following implicit states:

1. **IDLE** – Waiting for `INIT` or for an event on socket `u8Colour`.
2. **INIT** – After `INIT`, the internal block is initialized.
3. **READY** – After `INITO`, the FB is ready to receive color changes.
4. **BUSY** – During the execution of the color change command (after `REQ`). 5. **DONE** – After `CNF`, the next command can be processed.

Application Scenarios

- **ISOBUS Virtual Terminal**

Changing the background color of a graphical object (e.g., button, group, softkey) on the VT screen in an agricultural control system.

- **HMI Customization**

Responding to user input or system states to dynamically adjust the visual display (e.g., alarm colors, highlighting active elements).

- **Recovering the Previous Color**

The `u8OldColour` plugin allows you to save the previous color value or use it for later restoration.

-

Comparison with Similar Function Blocks

- **`Q_BackgroundColour`** (without `_AUS`):

Offers the same core functionality, but usually with individual event/data ports instead of encapsulated adapters. The `_AUS` function block simplifies the connection to other function blocks that also use AUS adapters.

- **`Q_Colour`** or **`Q_ForegroundColour`**:

Modify different color properties (foreground, full color) and use similar return values and status mechanisms.

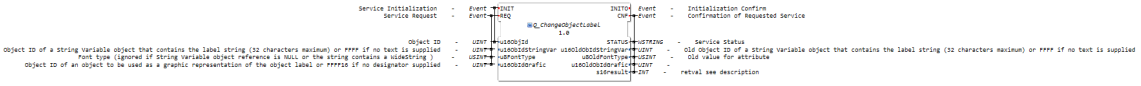
- **`Command_ChangeColour`** (more general):

Could modify multiple color parameters simultaneously, while `Q_BackgroundColour_AUS` specializes in background colors.

Conclusion

The `Q_BackgroundColour_AUS` function block offers a standards-compliant, compact interface for changing background colors in an ISOBUS VT. The use of the unidirectional adapter simplifies integration with other components and improves clarity. The documented error codes and the clear separation of initialization and service execution make it a reliable building block for dynamic HMI adjustments in agricultural control systems.

Q_ChangeObjectLabel



• — ○ • — ○ • — ○ •

Introduction

The **Q_ChangeObjectLabel** is a standards-compliant function block for changing object labels in Virtual Terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.50) specification for agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object ID)
- `REQ`: Label Change Request (with text/graphic parameters)

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Change Acknowledgement (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Target Object Identifier
- `u16ObIdStringVar` (UINT): Text Variable Object ID (max. 32 characters)
- `u8FontType` (USINT): Font (for text labeling)
- `u16ObIdGrafic` (UINT): Graphic object ID (for symbol labeling)

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operating status message
- `u16OldObIdStringVar` (UINT): Previous text variable ID
- `u8OldFontType` (USINT): Previous font
- `u16OldObIdGrafic` (UINT): Previous graphic object ID
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

The F.50 command addresses, in bytes 2,3, the **Object ID of the object to label** — per Annex F.51 (error bit 0) and B.21 this object must be **listed in the Object Label Reference List** (i.e. any pool object whose ID is an entry of the reference list).

Implementation note (VTClientHelper): `iso_is_object_label_refer_list_id` accepts only the **ObjectLabelReferList** range (the Object Label Reference List object's own ID) — a known deviation from the Annex F.50 semantics:

- **ObjectLabelReferList**: 40000 – 40999

ID_NULL (65535) is not a valid command target but deactivates the FB when sent via `INIT`.

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with target object ID
- `INITO` confirms operational readiness

1. **Label Change**:

- `REQ` with new label parameters
- Supports text (with font) or graphic symbols
- `CNF` returns the result and the old values

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.50) ✓ **Flexible Labeling**: Text or graphics ✓ **Font Control** (8-bit) ✓ **32-character limit** for text labels

Standard Parameters

Parameter	NULL value	Description
u16ObIdStringVar	0xFFFF	No text
u16ObIdGrafic	0xFFFF	No graphic

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Success
−6	VT_E_OVERFLOW	Buffer overflow (text too long)
−128	VT_E_HANDLE_INVALID	Invalid object ID

Application Scenarios

- **Machine Control**: Dynamic button labels
- **Language Switching**: Multilingual interfaces
- **Status Indicators**: Icon change on alarms
- **Personalization**: User-defined labels

⚖️ Comparison with similar building blocks

Feature	Q_ChangeObjectLabel	VtDynamicLabel	VtTextManager
ISO Standard	✓	✖	✖
Labeling	Text + Graphics	Text Only	Text Only
Font	Controllable	Fixed	Partial

Conclusion

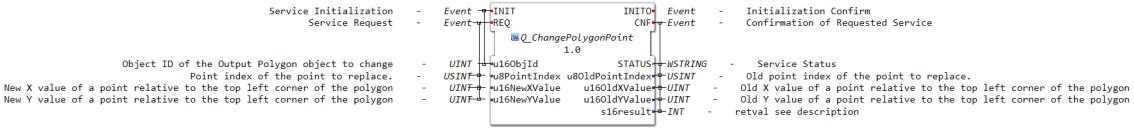
The Q_ChangeObjectLabel block provides the reference implementation for ISOBUS labeling:

- **Versatile**: Combination of text and graphics
- **Standard-compliant**: Full ISO 11783-6 compatibility
- **Proven in practice**: Used in modern agricultural machinery

Essential for:

- Adaptive user interfaces
- Multilingual systems
- Icon-driven operating concepts

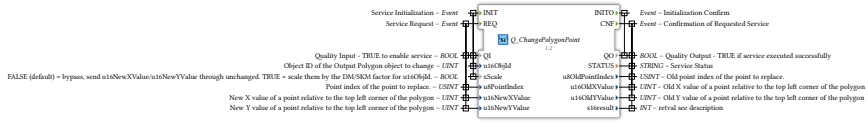
Q_ChangePolygonPoint



• — ○ • — ○ • — ○ •

Introduction

The **Q_ChangePolygonPoint** is a standards-compliant function block for modifying polygon points in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.52) specification for agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with polygon object ID and xScale)
- `REQ`: Point Change Request (with index and coordinates)

Event Outputs

- `INITO`: Initialization Confirmation
- `CNF`: Change Confirmation (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Polygon Object ID
- `xScale` (BOOL): If TRUE, `u16NewXValue`/`u16NewYValue` are scaled by the DM/SKM factor for `u16ObjId` before sending; if FALSE (default) they are passed through unchanged.
- `u8PointIndex` (USINT): Point Index (0-based)
- `u16NewXValue` (UINT): New X-coordinate (relative)
- `u16NewYValue` (UINT): New Y-coordinate (relative)

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operational status message
- `u8OldPointIndex` (USINT): Previous point index
- `u16OldXValue` (UINT): Previous X-coordinate
- `u16OldYValue` (UINT): Previous Y-coordinate
- `s16result` (INT): ISO-compliant result code

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with polygon object ID
- `INITO` confirms operational readiness

1. **Point Change**:

- `REQ` with index and new coordinates
- Coordinates relative to the upper left corner

- `CNF` returns result and old values

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.52) ✓ **Precise coordinate control** (16-bit) ✓ **Index-based access** (0-255 points) ✓ **One point per command** (fixed 8-byte message, no Transport Protocol) ✓ **Real-time processing**

Coordinate Range

Parameter	Range	Description
X-value	0 - 65535	Horizontal position (px)
Y-value	0 - 65535	Vertical Position (px)

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Success
−6	VT_E_OVERFLOW	Invalid Point Index
−128	VT_E_HANDLE_INVALID	Invalid Polygon ID

Application Scenarios

- **Machine Visualization**: Contour Adjustment
- **Map Display**: Dynamic Polygon Deformation
- **Diagnostic Displays**: Geometric Highlighting
- **Animated Elements**: Moving Polygon Shapes

⚖️ Comparison with Similar Building Blocks

Feature	Q_ChangePolygonPoint	VtGeometryEditor	VtDynamicShape
ISO Standard	✓	✖	✖
Accuracy	16-bit Coordinates	8-bit	16-bit
Point Count	Up to 255	Unlimited	Limited

Conclusion

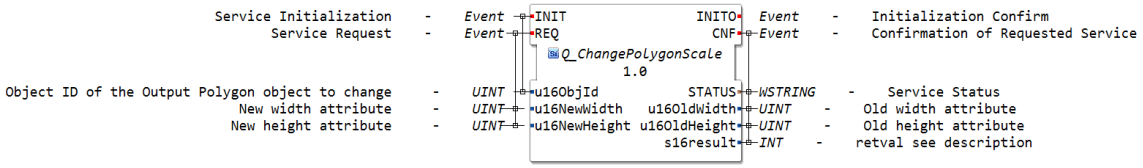
The Q_ChangePolygonPoint block provides the reference implementation for dynamic polygon changes:

- **Precise**: Millimeter-accurate position control
- **Powerful**: Real-time processing
- **Standard-compliant**: Full ISO 11783-6 compatibility

Essential for:

- Adaptive geometry representations
- Technical drawings
- Interactive mapping applications

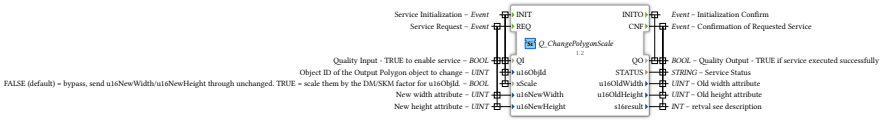
Q_ChangePolygonScale



• — ○ • — ○ • — ○ •

Introduction

The **Q_ChangePolygonScale** is a standards-compliant function block for scaling polygon objects in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.54) specification for agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization request (with polygon object ID and `xScale`)
- `REQ`: Scaling request (with new dimensions)

Event Outputs

- `INITO`: Initialization confirmation
- `CNF`: Scaling confirmation (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Polygon object ID
- `xScale` (BOOL, initial value `FALSE`): `FALSE` = bypass, `u16NewWidth`/`u16NewHeight` are sent unchanged; `TRUE` = both are scaled by the DM/SKM factor for `u16ObjId`
- `u16NewWidth` (UINT): New width in pixels
- `u16NewHeight` (UINT): New height in pixels

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operational status message
- `u16OldWidth` (UINT): Previous width
- `u16OldHeight` (UINT): Previous height
- `s16result` (INT): ISO-compliant result code

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with polygon object ID
- `INITO` confirms operational readiness

1. **Scaling**:

- `REQ` triggers with new dimensions
- Proportional scaling of all polygon points
- `CNF` returns result and previous values

1. **Error Handling**:

- ISO-standardized Error Codes
- Detailed Status Messages

Technical Features

✓ **ISO 11783-6 compliant** (F.54) ✓ **Proportional scaling** (width/height) ✓ **16-bit resolution** (0-65535 pixels) ✓ **Real-time processing**

Scaling Behavior

Parameter	Effect
Width	Horizontal scaling
Height	Vertical scaling

Per Annex F.54 the VT rescales all polygon points using 32-bit integer math: `new_x = [(old_x × new_width) + (old_width/2)] / old_width` and analogously for `new_y`.

Return Codes (s16result)

Code	Constant	Meaning				
0	VT_E_NO_ERR	Success		−6	VT_E_OVERFLOW	Invalid Dimensions
−128	VT_E_HANDLE_INVALID	Invalid Polygon ID				
−127	VT_E_ISO_INSTANCE_INVALID	Invalid Instance				

Application Scenarios

- **Machine Display**: Dynamic Resizing
- **Zoom Functions**: Detail Magnification
- **Responsive Designs**: Adapting to Display Sizes
- **Animated Transitions**: Smooth Size Changes

⚖️ Comparison with Similar Building Blocks

Feature	Q_ChangePolygonScale	VtGeometryScaler	VtDynamicResize
ISO Standard	✓	✖	✖
Scaling	Proportional	Free	Width Only
Resolution	16-bit	8-bit	16-bit

Conclusion

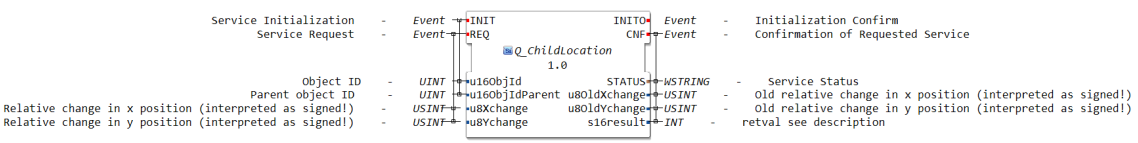
The Q_ChangePolygonScale block provides the reference implementation for polygon scaling:

- **Precise**: Pixel-accurate size control
- **Proportional**: Maintains the original shape
- **Standards-compliant**: Full ISO 11783-6 compatibility

Ideal for:

- Adaptive visualizations
- Dynamic map views
- Resizable controls

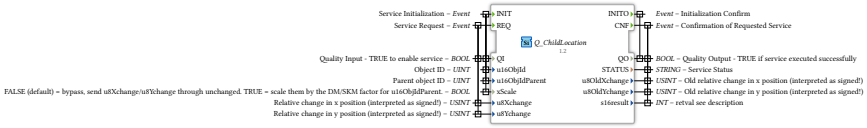
Q_ChildLocation



• — ○ • — ○ • — ○ •

Introduction

The **Q_ChildLocation** is a standards-compliant function block for changing the position of child objects in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.14) specification for agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object and parent ID as well as `xScale`)
- `REQ`: Position Change Request (with relative coordinates)

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Change Acknowledgement (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Child Object ID
- `u16ObjIdParent` (UINT): Parent Object ID
- `xScale` (BOOL, default `FALSE`): `FALSE` = bypass, `u8Xchange`/`u8Ychange` are sent through unchanged; `TRUE` = scale them by the DM/SKM factor for `u16ObjIdParent`
- `u8Xchange` (USINT): Relative X-Position Change (-128 to +127)
- `u8Ychange` (USINT): Relative Y-position change (-128 to +127)

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operational status message
- `u8OldXchange` (USINT): Previous X-change
- `u8OldYchange` (USINT): Previous Y-change
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` (the child object to move) and `u16ObjIdParent` (the parent object) are subject to **different** validity rules. ISO 11783-6 Annex F only defines which object types are valid as the child — which object types may act as parent is defined separately in Annex B, under each object type's "Allowed commands" list.

`u16ObjId` — valid child object types (Annex F.14): Container (3000–3999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Output String Field (11000–11999), Output Number Field (12000–12999), Output Line (13000–13999), Output Rectangle (14000–14999), Output Ellipse (15000–15999), Output Polygon (16000–16999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Picture Graphic (20000–20499), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999), Auxiliary Function/Type2 (29000–29999 / 31000–31999), Auxiliary Input/Type2 (30000–30999 / 32000–32999), Auxiliary Control Designator (33000–33999).

`u16ObjIdParent` — valid parent object types (Annex B, "Allowed commands"): WorkingSet (B.1), Data Mask (B.2), Alarm Mask (B.3), **Container (B.4)**, Key (B.6), Window Mask, Button, Auxiliary Function (Type 1/2), Auxiliary Input (Type 1/2).

Not valid as parent: Soft Key Mask (B.5 does not list the command as allowed) and Animation Object (its children are indexed frames, not X/Y-positioned).

ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`.

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with child and parent object IDs
- `INITO` confirms operational readiness

1. **Position Change**:

- `REQ` with relative coordinate changes
- Values are interpreted as signed 8-bit
- `CNF` returns the result and previous values

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.14) ✓ **Relative position change** (relative to the parent) ✓ **Signed 8-bit values** (-128 to +127) ✓ **Real-time processing**

Position Change Range

Parameter	Range	Description
X Change	−128 - +127	Horizontal Displacement
Y-Change	−128 - +127	Vertical Shift

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Success
−6	VT_E_OVERFLOW	Invalid Position Values
−128	VT_E_HANDLE_INVALID	Invalid Object ID

Application Scenarios

- **Dynamic Layouts**: Object Positioning
- **Animated Elements**: Smooth Movements
- **Responsive Designs**: Adaptation to Different Screen Sizes
- **Grouped Objects**: Shared Movements

⚖️ Comparison with Similar Building Blocks

Feature	Q_ChildLocation	VtAbsolutePosition	VtGroupMove
ISO Standard	✓	✖	✖
Movement	Relative	Absolute	Groups
Value Range	±127 pixels	0-65535 pixels	±127 pixels

Conclusion

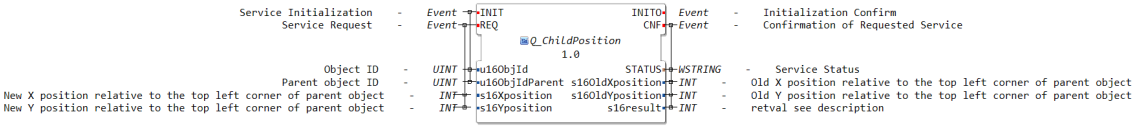
The Q_ChildLocation block provides the reference implementation for relative position changes:

- **Flexible**: Simple relative positioning
- **Precise**: Pixel-accurate control
- **Standard-compliant**: Full ISO 11783-6 compatibility

Ideal for use with:

- Dynamic surfaces
- Animated transitions
- Parent-child object relationships

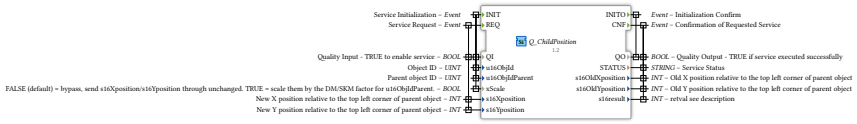
Q_ChildPosition



• — ○ • — ○ • — ○ •

Introduction

The **Q_ChildPosition** is a standards-compliant function block for the absolute position change of child objects in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.16) specification for agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization request (with object ID, parent ID and xScale)
- `REQ`: Position change request (with absolute coordinates)

Event Outputs

- `INITO`: Initialization confirmation
- `CNF`: Change confirmation (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Child object ID
- `u16ObjIdParent` (UINT): Parent object ID
- `xScale` (BOOL): If TRUE, `s16Xposition`/`s16Yposition` are scaled by the DM/SKM factor for `u16ObjIdParent` before sending; if FALSE (default) they are passed through unchanged.
- `s16Xposition` (INT): Absolute X-position relative to the parent
- `s16Yposition` (INT): Absolute Y-position relative to the parent

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operational status message
- `s16OldXposition` (INT): Previous X-position
- `s16OldYposition` (INT): Previous Y-position
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` (the child object to move) and `u16ObjIdParent` (the parent object) are subject to **different** validity rules. ISO 11783-6 Annex F only defines which object types are valid as the child — which object types may act as parent is defined separately in Annex B, under each object type's "Allowed commands" list.

`u16ObjId` — valid child object types (Annex F.16): Container (3000–3999), Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Output String Field (11000–11999), Output Number Field (12000–12999), Output Line (13000–13999), Output Rectangle (14000–14999), Output Ellipse (15000–15999), Output Polygon (16000–16999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Picture Graphic (20000–20499), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object

(48000–48999), Auxiliary Function/Type2 (29000–29999 / 31000–31999), Auxiliary Input/Type2 (30000–30999 / 32000–32999), Auxiliary Control Designator (33000–33999).

`u16ObjIdParent` — valid parent object types (Annex B, "Allowed commands"): WorkingSet (B.1), Data Mask (B.2), Alarm Mask (B.3), **Container (B.4)**, Key (B.6), Window Mask, Button, Auxiliary Function (Type 1/2), Auxiliary Input (Type 1/2).

Not valid as parent: Soft Key Mask (B.5 does not list the command as allowed) and Animation Object (its children are indexed frames, not X/Y-positioned).

ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`.

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with child and parent object IDs
- `INITO` confirms operational readiness

1. **Position Change**:

- `REQ` with absolute coordinates relative to the parent
- Value range: −32768 to +32767 pixels
- `CNF` returns the result and the previous position

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.16) ✓ **Absolute position change** (relative to the parent) ✓ **16-bit value range** (±32768 pixels) ✓ **Real-time processing**

Position Range

Parameter	Range	Description
X-Position	−32768 - +32767	Horizontal Position
Y-Position	−32768 - +32767	Vertical Position

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Success
−6	VT_E_OVERFLOW	Position outside the valid range
−128	VT_E_HANDLE_INVALID	Invalid object ID

Application Scenarios

- **Precise Object Placement**: Exact positioning of UI elements
- **Dynamic Layouts**: Reorientation of controls
- **Complex Arrangements**: Multi-layered interfaces
- **Machine Visualization**: Technical drawings

⚖️ Comparison with Similar Function Blocks

Feature	Q_ChildPosition	Q_ChildLocation	VtAbsoluteMove
ISO Standard	✓	✓	✖
Position Type	Absolute	Relative	Absolute

Value Range	±32767 pixels	±127 pixels	±32767 pixels

Conclusion

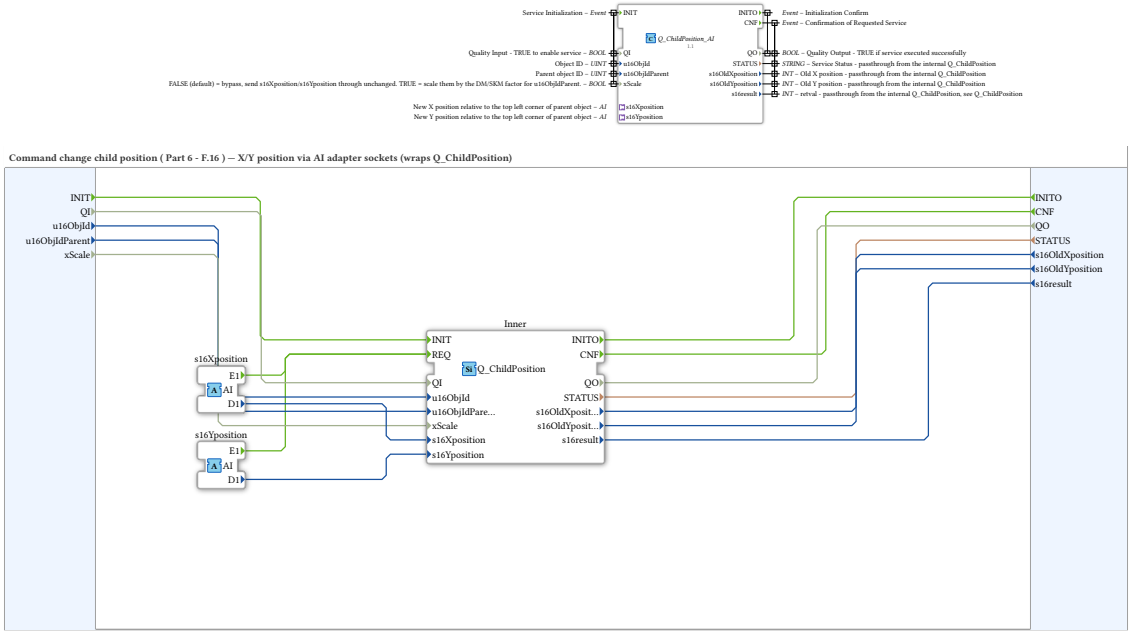
The Q_ChildPosition block provides the reference implementation for absolute position changes:

- **Precise**: Pixel-accurate positioning
- **Flexible**: Large value range
- **Standard-compliant**: Full ISO 11783-6 compatibility

Ideal for use in:

- Complex user interfaces
- Technical drawings
- Precise layout requirements

Q_ChildPosition_AI



• – ○ • – ○ • – ○ •

Introduction

The **Q_ChildPosition_AI** function block implements a command to change the position of a child object (Part 6, F.16) within an ISO 11783-6 compliant automation system. It acts as a wrapper around the `Q_ChildPosition` function block, replacing the conventional REQ/data input interface with two unidirectional AI adapter sockets. This allows the X and Y position values to be delivered through adapter connections, decoupling the position source from the command logic. A scaling option (xScale) enables the position values to be adjusted by the display-master/slave (DM/SKM) factor of the parent object.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Service Initialization – triggers the initialization of the inner block.

The `INIT` event is associated with the data inputs `u16ObjId`, `u16ObjIdParent`, and `xScale`.

Event Outputs

Event	Type	Comment

INITO	EInit	Initialization Confirm – acknowledged after `INIT` completes.
CNF	Event	Confirmation of the requested position-change service.

The `CNF` event is associated with the data outputs `STATUS`, `s16OldXposition`, `s16OldYposition`, and `s16result`.

Data Inputs

Name	Type	Comment
QI	BOOL	Quality Input - TRUE to enable service
u16ObjId	UINT	Object ID of the child object whose position is to be changed. Defaults to `ID_NULL`.
u16ObjIdParent	UINT	Object ID of the parent object (reference coordinate system). Defaults to `ID_NULL`.
xScale	BOOL	If FALSE (default), the X/Y positions are passed through unchanged. If TRUE, they are scaled by the DM/SKM factor of the parent object.

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Service status – passthrough from the internal `Q_ChildPosition`.
s16OldXposition	INT	Previous X position of the child object – passthrough from the inner block.
s16OldYposition	INT	Previous Y position of the child object – passthrough from the inner block.
s16result	INT	Return value – passthrough from the internal `Q_ChildPosition`.

Adapters

Adapter Name	Type	Comment
s16Xposition	adapter::types::unidirectional::AI	New X position relative to the top-left corner of the parent object.
s16Yposition	adapter::types::unidirectional::AI	New Y position relative to the top-left corner of the parent object.

Both adapters are unidirectional inputs (sockets). Each provides one event (`E1`) and one data value (`D1`). The adapter's event triggers the position update.

Quality signals QI/QO: The data input `QI` (BOOL, initial value `TRUE`) is associated with the event `INIT` and is passed through to `QI` of the internal block `Inner`. The data output `QO` (BOOL) is taken over from `Inner.QO` and is associated with the events `INITO` and `CNF`.

Functionality

The block encapsulates an internal instance of `Q_ChildPosition`, called `Inner`. Its public interface is adapted as follows:

- On the `INIT` event, the inner block is initialized with the current values of `u16ObjId`, `u16ObjIdParent`, and `xScale`. The resulting `INITO` of the inner block is forwarded to the external `INITO`.
- A position update is triggered either by the event of the `s16Xposition` adapter or by the event of the `s16Yposition` adapter. In both cases, the internal `Q_ChildPosition.REQ` is fired. The actual position values are always read from both adapters' `D1` data pins at that moment – so an update triggered by the Y adapter will also pick up the latest X value, and vice versa. This design ensures that the command always uses the most current values from both sources.

- The data values `D1` of the adapters are connected directly to the corresponding inputs `s16Xposition` and `s16Yposition` of the inner block.
- All outputs (`STATUS`, `s16OldXposition`, `s16OldYposition`, `s16result`) of the inner block are passed through unchanged to the outer interface.

The block therefore acts as a pure adapter layer that preserves the full semantics and state machine of the underlying `Q_ChildPosition`.

Technical Features

- **Adapter-based input decoupling** – X and Y positions are received via separate unidirectional AI adapter sockets rather than conventional event/data pairs. This allows a clean separation of data source and command logic in the surrounding network.
- **Independent event triggering** – Either adapter's event can initiate the position update, providing flexibility in the data flow topology.
- **Optional scaling** – The `xScale` flag enables automatic scaling of the position values using the DM/SKM factor of the specified parent object, without requiring external preprocessing.
- **Passthrough semantics** – All internal status and result data are forwarded transparently to the outer interface, so diagnostics and old-position reporting remain fully available.
- **Standard compliance** – Designed in accordance with ISO 11783-6, Part 6, command set F.16.

State Overview

The block does not maintain its own state machine; it inherits the behavior of the encapsulated `Q_ChildPosition`. In general terms:

- After initialization (`INIT`/`INITO`), the block is ready to accept position-change requests.
- Whenever an adapter event occurs, a single request transaction is started on the inner block, which performs the required validation, scaling (if enabled), and position update.
- Upon completion, the inner block issues `CNF`, carrying the new status, the old coordinates, and the result code.

Because both adapter events are connected to the same inner REQ input, simultaneous or rapid successive events are serialized by the inner block's execution model; the system behaves deterministically with respect to the inner state machine.

Application Scenarios

- **Decoupled HMI input handling** – When the X and Y coordinate values originate from separate UI widgets or external bus nodes, the AI adapters allow those sources to be connected directly to the position command without a central coordinator.
- **Distributed automation architectures** – In networked ISO 11783 installations, position sources can be located on different nodes, while the command logic resides in a central controller.
- **Scaled coordinate transformation** – For applications where the parent object uses a different coordinate scaling (DM/SKM factor), enabling `xScale` adapts the incoming raw values to the parent's coordinate system automatically.
- **Reusable wrapper pattern** – This block demonstrates how to adapt an existing command FB to an adapter-based interface, improving modularity and testability in larger FB networks.

Comparison with Similar Blocks

Compared to the direct `Q_ChildPosition` block, the following differences apply:

Aspect	Q_ChildPosition	Q_ChildPosition_AI
Position inputs	Plain `s16Xposition` / `s16Yposition` data inputs, triggered by a single REQ event.	Position values arrive via two AI adapter sockets; either adapter event triggers the update.
Interface style	Conventional input/output variables.	Adapter-based input sockets, clean separation of event and data for each coordinate.
Scaling control	Provided as input (xScale).	Provided as input (xScale); behavior identical.

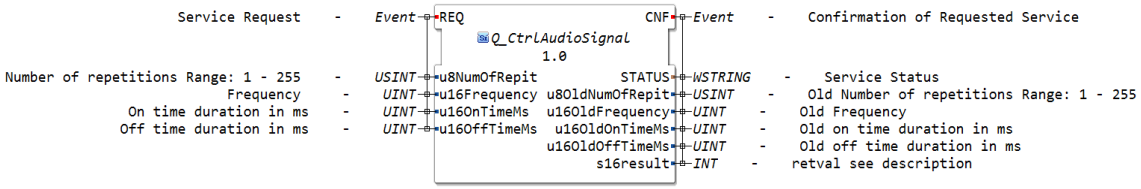
Outputs	STATUS, old positions, result.	All outputs passthrough unchanged.
Use case	Direct command invocation in a fixed network.	Flexible connection of external position sources via adapters.

The AI wrapper adds architectural flexibility but does not change the service semantics of the underlying command.

Conclusion

`Q_ChildPosition_AI` is an adapter wrapper around the ISO 11783-6 child-position command, providing a clean, event-driven interface for X and Y coordinates via unidirectional AI sockets. It preserves all functionality of the original `Q_ChildPosition` – including status reporting, old-position feedback, and optional DM/SKM scaling – while enabling a more modular and flexible network design. This makes it well-suited for distributed control systems where position data originates from multiple independent sources.

Q_CtrlAudioSignal

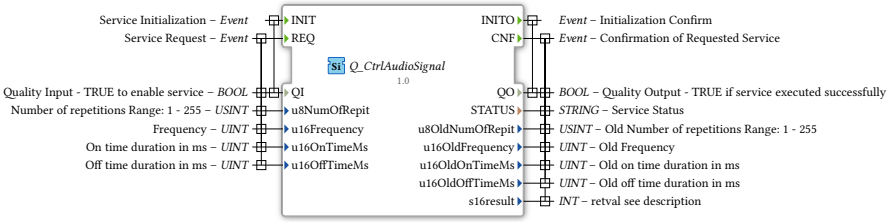


• — ○ • — ○ • — ○ •

Introduction

The **Q_CtrlAudioSignal** is a standards-compliant function block for controlling audio devices in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.10) specification for agricultural tax systems.

The F.10 command addresses the VT's **audio device** directly — it does **not reference any object**, so this block has no Object ID input. **Activations = 0** terminates any audio in process from the originating Working Set (frequency and duration values are ignored); 1–255 sets the number of audio activations.



Interface Structure

Event Inputs

- `INIT`: Initialization Request
- `REQ`: Audio Control Request (with signal parameters)

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Control Acknowledgement (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u8NumOfRepit` (USINT): Repetitions (1-255)
- `u16Frequency` (UINT): Frequency in Hz
- `u16OnTimeMs` (UINT): On-time in ms
- `u16OffTimeMs` (UINT): Off-time in ms

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operating status message
- `u8OldNumOfRepit` (USINT): Previous repetitions
- `u16OldFrequency` (UINT): Previous frequency
- `u16OldOnTimeMs` (UINT): Previous on-time
- `u16OldOffTimeMs` (UINT): Previous off-time
- `s16result` (INT): ISO-compliant result code

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` without parameters
- `INITO` confirmed Operational Readiness

1. **Audio Control**:

- Trigger `REQ` with signal parameters
- Generates tone with a configurable pattern
- `CNF` returns result and previous values

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.10) ✓ **Flexible Signal Control**: Frequency, Duration, Repetitions ✓ **16-bit Time Resolution** (0-65535 ms) ✓ **255 Repetitions** maximum

Standard Parameter Ranges

Parameter	Range	Typical Value
Frequency	0-65535 Hz	2000 Hz
Duty Cycle	0-65535 ms	200 ms
Shutdown time	0-65535 ms	100 ms
Repetitions	1-255	3

Return codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Success
−6	VT_E_OVERFLOW	Invalid parameter values
−8	VT_E_NOACT	VT in incorrect state
−128	VT_E_HANDLE_INVALID	Invalid audio configuration

Application Scenarios

- **Warning Signals**: Audible alarm messages
- **Confirmation Tones**: Operator feedback
- **Voice Announcements**: Pre-recorded messages
- **Machine Status**: Audible status indicators

⚖️ Comparison with similar modules

Feature	Q_CtrlAudioSignal	VtSoundManager	VtAudioAlert
ISO Standard	✓	✖	✖
Parameterization	Full	Partial	Simple
Frequency Control	Yes	No	No

🛠️ Related Exercises

- Exercise_017
- Exercise_018
- Exercise_018a

Conclusion

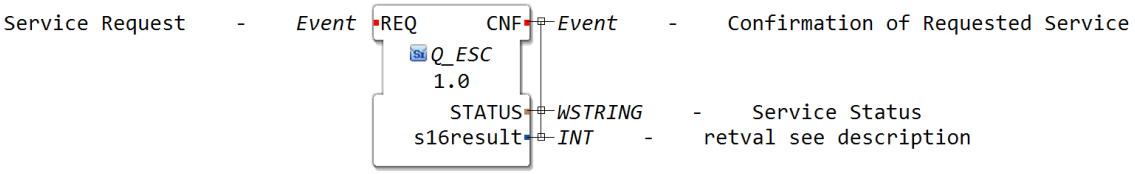
The Q_CtrlAudioSignal block provides the reference implementation for VT audio control:

- **Flexible**: Fully configurable audio signals
- **Standards compliant**: Full ISO 11783-6 compatibility
- **Robust**: Proven technology in field devices

Essential for:

- Safety-critical warning signals
- User-friendly acoustic feedback
- Machines with complex audio status messages

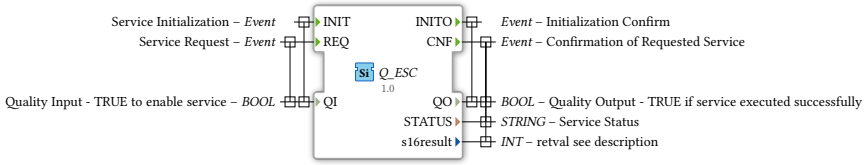
Q_ESC



• — ◦ • — ◦ • — ◦ •

Introduction

The **Q_ESC** is a standards-compliant function block for termination control in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.8) specification for agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request
- `REQ`: Abort Request

Event Outputs

- `INITO`: Initialization Confirmation
- `CNF`: Abort Confirmation (with status)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operational Status Message
- `s16result` (INT): ISO-compliant Result Code

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` without parameters
- `INITO` confirmed Operational Readiness

1. **Cancel Function**:

- `REQ` terminates the current user input
- No additional parameters required
- `CNF` returns the result status

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.8)
✓ **Universal cancel command**
✓ **No configuration required** ✓ **Immediate effect upon execution

Typical Use Cases

Scenario	Effect
Dialog cancel	Ends current input
Exit menu	Return to main screen
Alarm acknowledgement	Stops audible signals

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successfully aborted
−8	VT_E_NOACT	No active input to abort
−21	VT_E_NO_INSTANCE	No VT instance available
−130	VT_E_NOT_ALIVE	VT not reachable

Application Scenarios

- **Emergency Abort**: Quickly terminate processes
- **Input Error**: Correction option for users
- **Menu Navigation**: Back functionality
- **System Reset**: Abort all active operations

⚖️ **Comparison with similar modules**

Feature	Q_ESC	VtAbortCmd	VtCancelOp
ISO Standard	✓	✖	✖
Effect	Global	Local	Object-Specific

Parameters	None	Configurable	Configurable

Conclusion

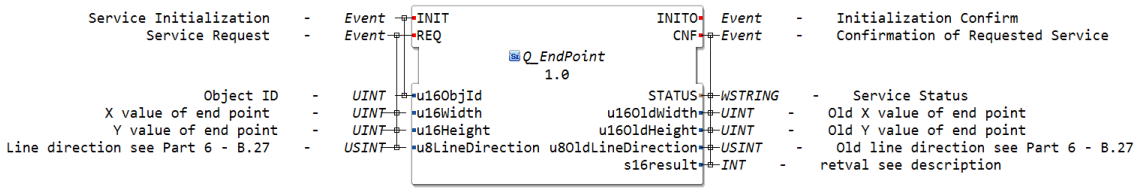
The Q_ESC block provides the standardized abort function for ISOBUS systems:

- **Universal**: Suitable for all abort scenarios
- **Simple**: No configuration required
- **Reliable**: Immediate system response

Essential for:

- Safety-critical applications
- User-friendly interfaces
- Robust error handling

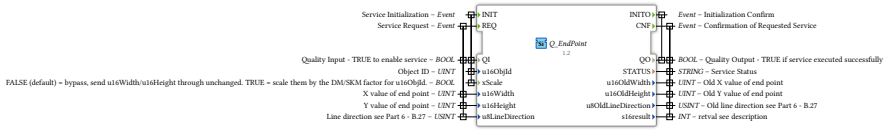
Q_EndPoint



• — ○ • — ○ • — ○ •

Introduction

The **Q_EndPoint** is a standards-compliant function block for controlling line endpoints in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.26) specification for agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object ID and `xScale`)
- `REQ`: Endpoint Change Request (with position/direction)

Event Outputs

- `INITO`: Initialization Confirmation
- `CNF`: Change Confirmation (with result data)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Line Object ID
- `xScale` (BOOL, initial value `FALSE`): `FALSE` = bypass, `u16Width`/`u16Height` are sent unchanged; `TRUE` = both are scaled by the DM/SKM factor for `u16ObjId`
- `u16Width` (UINT): New X End Position (0-65535 pixels)
- `u16Height` (UINT): New Y-End Position (0-65535 pixels)
- `u8LineDirection` (USINT): Line Direction (ISO 11783-6 B.27)

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operating Status Message
- `u16OldWidth` (UINT): Previous X-End Position
- `u16OldHeight` (UINT): Previous Y-End Position
- `u8OldLineDirection` (USINT): Previous Line Direction

- `s16result` (INT): ISO-compliant Result Code

Valid Object IDs

`u16ObjId` — valid object type (Annex F.26, Output Line object): Output Line (13000–13999).
ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`. Any ID outside these ranges is invalid for commanding.

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with Line Object ID
- `INITO` confirms operational readiness

1. **Endpoint Change**:

- `REQ` with new position and direction parameters
- Changes the endpoint and orientation of the line
- `CNF` returns the result and the old values

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.26 + B.27)
✓ **16-bit position resolution** (0-65535 pixels) ✓ **Full line control** (position + direction)
✓ **Real-time processing**

Line directions (B.27)

| Value | Direction | Description | |—|—————|———————| | 0 | Horizontal | From left to right | | 1 | Vertical | From top to bottom | | 2 | Diagonal right | Top left → Bottom right | | 3 | Diagonal left | Top right → Bottom left |

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Success
−6	VT_E_OVERFLOW	Invalid position values
−128	VT_E_HANDLE_INVALID	Invalid line object ID
−129	VT_E_ISO_INSTANCE_INVALID	Invalid instance

Application Scenarios

- **Machine Visualization**: Dynamic Connecting Lines
- **Diagrams**: Customizable Data Displays
- **Technical Drawings**: Configurable Guidelines
- **UI Design**: Adaptive Dividing Lines

⚖️ **Comparison with Similar Building Blocks**

Feature	Q_EndPoint	VtLineEditor	VtDynamicLine
ISO Standard	✓	✖	✖
Control	Endpoint	Full	Position Only
Directions	4 Options	8 Options	None

Conclusion

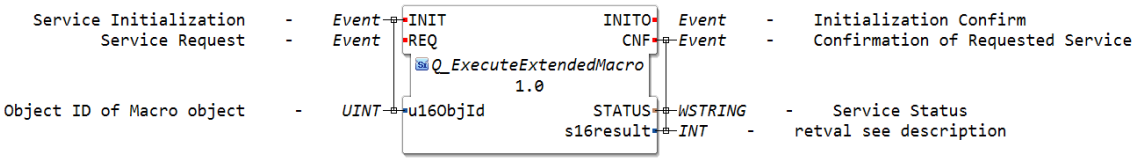
The Q_EndPoint block provides the reference implementation for line endpoint control:

- **Precise**: Pixel-accurate positioning
- **Flexible**: Combined position and direction control
- **Standard-compliant**: Full ISO 11783-6 compatibility

Ideal for use in:

- Dynamic technical drawings
- Adaptive diagram displays
- Configurable user interfaces

Q_ExecuteExtendedMacro



• — ○ • — ○ • — ○ •

Introduction

The **Q_ExecuteExtendedMacro** is a standards-compliant function block for executing extended macros in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.62) specification for VT from version 5 onwards.



Interface Structure

Event Inputs

- `INIT`: Initialization request (with macro ID)
- `REQ`: Macro execution request

Event Outputs

- `INITO`: Initialization acknowledgment
- `CNF`: Execution acknowledgment (with status)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Macro object ID (16-bit)

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operational status message
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` addresses the Macro object (range **1–999**, 16-bit field in bytes 2,3 of command F.62). Unlike Q_ExecuteMacro (F.48, 8-bit field), all Macro IDs 1–999 can be addressed here.

ID_NULL (65535) is not a valid command target but deactivates the FB when sent via `INIT`.

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with macro object ID
- `INITO` confirms operational readiness

1. **Macro Execution**:

- `REQ` without additional parameters
- Executes saved macro sequence
- `CNF` returns result status

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.62) ✓ **Exclusive to VT Version 5+** ✓ **16-bit Macro IDs** (Extended Address Space) ✓ **Atomised Execution** (Transaction-Safe)

Macro Properties

Feature	Description
Location	Permanently stored in the VT
Size	Up to 255 commands
Content	VT command sequences
ID Range	0x0001 - 0xFEFF (reserved)

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful execution
−6	VT_E_OVERFLOW	Macro too complex
−8	VT_E_NOACT	VT not ready
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance

Application Scenarios

- **Automated Processes**: Complex Command Chains
- **System Initialization**: Startup Procedures
- **Diagnostic Routines**: Automated Tests
- **User Actions**: Macro Recordings

⚖️ **Comparison with Similar Components**

Feature	Q_ExecuteExtendedMacro	VtMacroPlayer	VtCommandSequence
ISO Standard	✓	✖	✖
VT Version	5+	All	All
ID Type	16-bit	8-bit	16-bit

Conclusion

The Q_ExecuteExtendedMacro block provides the standard implementation for extended macros:

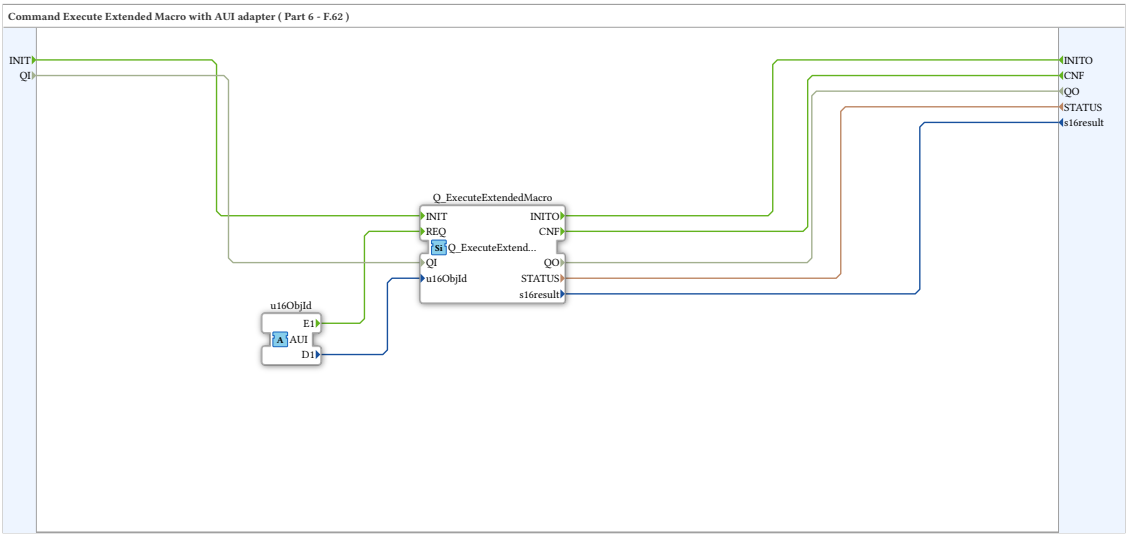
- **Powerful**: Complex automations possible
- **Future-proof**: Optimized for VT Generation 5
- **Reliable**: Atomic execution guaranteed

Essential for:

- Highly automated agricultural systems
- Complex operating processes
- Systems with extended functionality

Q_ExecuteExtendedMacro_AUI





• – ○ • – ○ • – ○ •

Introduction

The **Q_ExecuteExtendedMacro_AUI** function block is an AUI adapter wrapper for **Q_ExecuteExtendedMacro** (ISO 11783-6, Part 6 – F.62). It enables executing extended macros (16-bit macro IDs 1–999) on an ISOBUS Virtual Terminal using an AUI adapter interface.

Interface Structure

Event Inputs

Name	Type	Comment
`INIT`	`EInit`	Service Initialization

Event Outputs

Name	Type	Comment
`INITO`	`EInit`	Initialization Confirm
`CNF`	`Event`	Confirmation of Requested Service

Data Inputs

Apart from `QI`, there are no direct data inputs — extended macro Object ID is received via the AUI adapter socket.

Name	Type	Initial value	Comment
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE to enable service

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Service Status
s16result	INT	Command return code

Adapters

Type	Name	Direction	Comment
adapter::types::unidirectional::AUI	u16ObjId	Socket (Input)	Object ID of extended macro (1–999)

Quality signals QI/QO: The data input QI (BOOL, initial value TRUE) is associated with the event INIT and is passed through to QI of the internal block Q_ExecuteExtendedMacro. The data output QO (BOOL) is taken over from Q_ExecuteExtendedMacro.QO and is associated with the events INITO and CNF.

Functionality

Connects internal Q_ExecuteExtendedMacro (*isobus::UT::Q::Q_ExecuteExtendedMacro*) to the AUI adapter interface. An event on u16ObjId.E1 triggers REQ on the inner FB, sending the 16-bit macro ID from u16ObjId.D1 to the VT.

Technical Features

- **16-bit Extended Macro Range:** Supports macro IDs 1 to 999 (ISO 11783-6 VT version 5+).
- **Adapter Wrapper:** Unidirectional AUI connection.

State Overview

Delegates state handling to inner Q_ExecuteExtendedMacro.

Application Scenarios

- Triggering extended macros in VT version 5+ systems via AUI adapter lines.

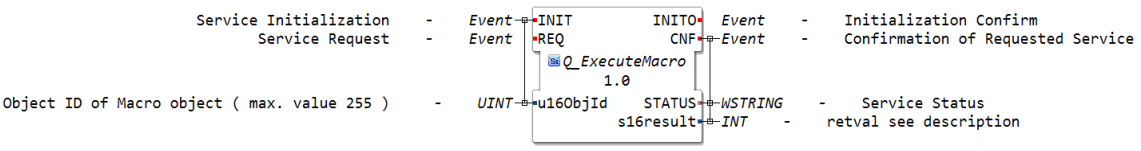
Comparison with Similar Blocks

Use **Q_ExecuteMacro_AUI** for standard 8-bit macro IDs (1–255) and **Q_ExecuteExtendedMacro_AUI** for 16-bit macro IDs (1–999).

Conclusion

Q_ExecuteExtendedMacro_AUI provides an adapter wrapper for extended macro execution in ISOBUS VT applications.

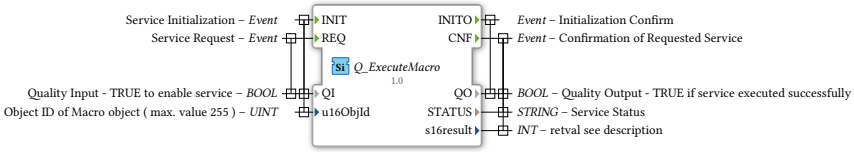
Q_ExecuteMacro



• — ○ • — ○ • — ○ •

Introduction

The **Q_ExecuteMacro** is a standards-compliant function block for executing macros in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.48) specification for VT from version 4 onwards.



Interface Structure

Event Inputs

- `INIT`: Initialization request (with macro ID)
- `REQ`: Macro execution request

Event Outputs

- `INITO`: Initialization acknowledgment
- `CNF`: Execution acknowledgment (with status)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality Input - TRUE to enable service (associated with `INIT` and `REQ`)
- `u16ObjId` (UINT): Macro object ID (max. value 255)

Data Outputs

- `QO` (BOOL): Quality Output - TRUE if service executed successfully (associated with `INITO` and `CNF`)
- `STATUS` (STRING): Operational status message
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` addresses the Macro object. Annex A defines the range 1–999, but the F.48 command carries the Macro Object ID in only a **single byte** (Byte 2). This FB can therefore address only **IDs 1–255** (values > 255 are truncated). Use Q_ExecuteExtendedMacro (F.62) for Macro IDs 256–999.

ID_NULL (65535) is not a valid command target but deactivates the FB when sent via `INIT`.

Functionality

`QI` (default `TRUE`) is taken over with the `INIT` and `REQ` events and enables the service; `QO` is output with `INITO` and `CNF` and indicates whether the service executed successfully.

1. **Initialization**:

- `INIT` with macro object ID (8-bit compatible)
- `INITO` confirms operational readiness

1. **Macro Execution**:

- `REQ` without additional parameters
- Executes stored macro sequence
- `CNF` returns result status

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ ISO 11783-6 compliant (F.48) **✓ Compatible with VT version 4+ ✓ 8-bit Macro IDs** (0-255) **✓ Atomised Execution** (Transaction-Safe)

Macro Properties

Feature	Description
Location	Permanently stored in the VT
Size	Up to 255 commands

Content	VT command sequences
ID Range	0x01 - 0xFF (8-bit)

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful execution
−6	VT_E_OVERFLOW	Macro too complex
−8	VT_E_NOACT	VT not ready
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance

Application Scenarios

- **Standard Flows**: Frequently Used Command Chains
- **System Startup**: Basic Initializations
- **Operating Sequences**: Automated Menu Navigation
- **Diagnostics**: Predefined Test Routines

⚖️ Comparison with Similar Components

Feature	Q_ExecuteMacro	Q_ExecuteExtendedMacro	VtMacroRunner
ISO Standard	✓	✓	✖
VT Version	4+	5+	All
ID Type	8-bit	16-bit	8-bit

Conclusion

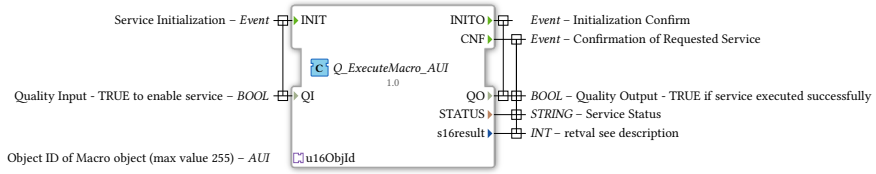
The Q_ExecuteMacro block offers the standard implementation for basic macros:

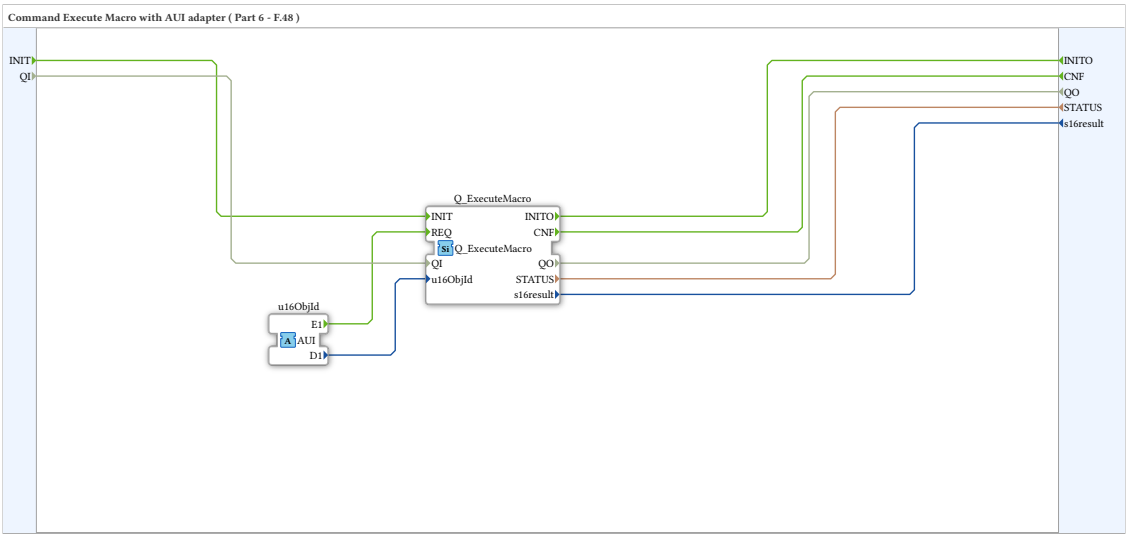
- **Simple**: Compact 8-bit addressing
- **Compatible**: Works with VT Generation 4+
- **Reliable**: Proven technology in production devices

Ideal for:

- Standard automation
- Compatibility-critical systems
- Simple macro applications

Q_ExecuteMacro_AUI





• – ○ • – ○ • – ○ •

Introduction

The **Q_ExecuteMacro_AUI** function block is an AUI adapter wrapper for **Q_ExecuteMacro** (ISO 11783-6, Part 6 – F.48). It enables executing macros on an ISOBUS Virtual Terminal using a unidirectional AUI adapter interface (UINT). The block encapsulates the internal `Q_ExecuteMacro` core and provides a clean, adapter-based interface.

Interface Structure

Event Inputs

Name	Type	Comment
`INIT`	`EInit`	Service Initialization

Event Outputs

Name	Type	Comment
`INITO`	`EInit`	Initialization Confirm
`CNF`	`Event`	Confirmation of Requested Service

Data Inputs

Apart from `QI`, there are no direct data inputs — macro Object ID is received via the AUI adapter socket.

Name	Type	Initial value	Comment
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE to enable service

Data Outputs

Name	Type	Comment
`QO`	`BOOL`	Quality Output - TRUE if service executed successfully
`STATUS`	`STRING`	Service Status
`s16result`	`INT`	Command return code

Adapters

Type	Name	Direction	Comment
adapter::types::unidirectional::AUI	u16ObjId	Socket (Input)	Object ID of macro to execute (1–255)

Quality signals QI/QO: The data input `QI` (BOOL, initial value `TRUE`) is associated with the event `INIT` and is passed through to `QI` of the internal block `Q_ExecuteMacro`. The data output `QO` (BOOL) is taken over from `Q_ExecuteMacro.QO` and is associated with the events `INITO` and `CNF`.

Functionality

Internally connects `Q_ExecuteMacro` (*isobus::UT::Q::Q_ExecuteMacro*) to the AUI adapter interface. Upon receiving an event at socket `u16ObjId.E1`, the internal FB's `REQ` event is triggered with the macro ID from `u16ObjId.D1`. Following execution, the internal FB emits `CNF` and updates `STATUS` and `s16result`.

Technical Features

- **Adapter Wrapper:** Input data received via AUI socket.
- **8-bit Macro Range:** Handles macro IDs in the range 1–255.
- **ISOBUS Standard Compliance:** Complies with ISO 11783-6 Annex F.48.

State Overview

Delegates all state handling to the inner FB `Q_ExecuteMacro`.

1. **Initialization** – `INIT` initializes the inner FB and confirms with `INITO`.
2. **Command Execution** – Event on `u16ObjId.E1` triggers macro execution.
3. **Confirmation** – `CNF` confirms completion with updated `STATUS` and `s16result`.

Application Scenarios

- Triggering VT macros via AUI adapter in ISO 11783-6 applications.
- Sequence-driven macro execution.

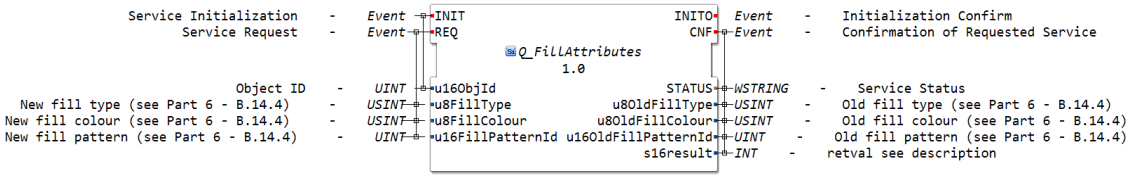
Comparison with Similar Blocks

Compared to direct `Q_ExecuteMacro`, **Q_ExecuteMacro_AUI** provides an adapter interface. For macro IDs > 255, use **Q_ExecuteExtendedMacro_AUI**.

Conclusion

Q_ExecuteMacro_AUI is a clean adapter wrapper for executing ISOBUS VT macros via AUI adapter connections.

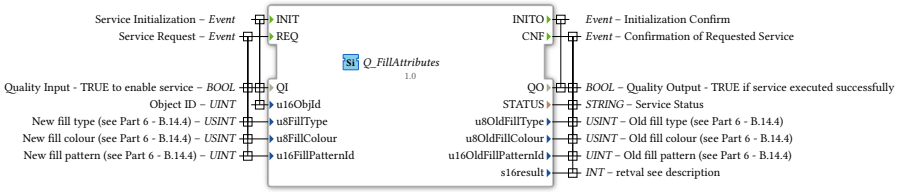
Q_FillAttributes



• — ○ • — ○ • — ○ •

Introduction

The **Q_FillAttributes** is a standards-compliant function module for modifying fill attributes in Virtual Terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.32) specification for VT systems.



Interface Structure

Event Inputs

- `INIT`: Initialization request (with object ID)
- `REQ`: Attribute change request (with new fill attributes)

Event Outputs

- `INITO`: Initialization confirmation
- `CNF`: Change confirmation (with status and old attributes)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Object ID (16-bit)
- `u8FillType` (USINT): New fill type
- `u8FillColour` (USINT): New fill color
- `u16FillPatternId` (UINT): New fill pattern

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `u8OldFillType` (USINT): Old fill type
- `u8OldFillColour` (USINT): Old fill color
- `u16OldFillPatternId` (UINT): Old fill pattern
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` addresses the target object of the **F.32** command (Change Fill Attributes). Valid is a **Fill Attributes object**:

FillAttributesObject: 25000–25999
ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with object ID
- `INITO` confirms operational readiness

1. **Attribute Change**:

- `REQ` with new fill attributes
- Changes Object Fill Properties
- `CNF` provides result status and previous values

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.32)
✓ **Comprehensive attribute control** (type, color, pattern)
✓ **Feedback of previous values** for reversal options ✓ **Transaction-safe** (Atomic execution)

Attribute Properties

Feature	Description
Fill types	According to ISO 11783-6 B.14.4

Color palette	8-bit color index
Patterns	16-bit pattern references
Object range	All fillable VT objects

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful change
−6	VT_E_OVERFLOW	Buffer overflow
−8	VT_E_NOACT	VT not ready
−21	VT_E_NO_INSTANCE	No VT client available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Design Adjustments**: Dynamic Interface Design
- **Status Visualization**: Color Highlighting
- **Pattern Change**: Context-Sensitive Displays
- **Theme Change**: Dark/Light Mode Switching

⚖️ Comparison with Similar Building Blocks

Feature	Q_FillAttributes	VtSetColor	VtPatternManager
ISO Standard	✓	✖	✖
Color Control	✓	✓	✖
Pattern Control	✓	✖	✓
Type Control	✓	✖	✖

Conclusion

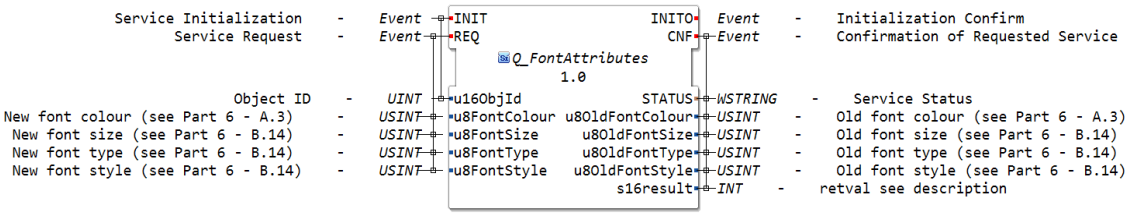
The Q_FillAttributes module provides the standard implementation for fill attribute changes:

- **Versatile**: Complete control over fill properties
- **Traceable**: Feedback of previous values
- **Robust**: ISO-compliant error handling

Essential for:

- Dynamic user interfaces
- State-dependent visualizations
- Design-flexible VT applications

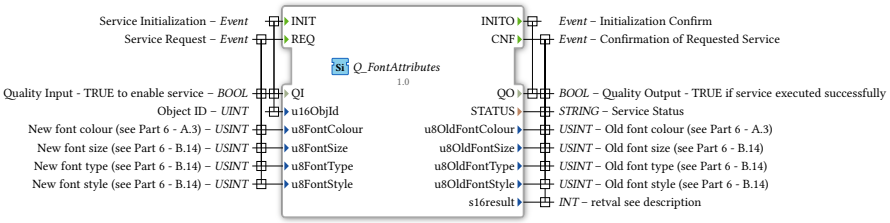
Q_FontAttributes



• — ○ • — ○ • — ○ •

Introduction

The **Q_FontAttributes** is a standards-compliant function module for modifying font attributes in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.28) specification for VT systems.



Interface Structure

Event Inputs

- `INIT`: Initialization request (with object ID)
- `REQ`: Attribute change request (with new font attributes)

Event Outputs

- `INITO`: Initialization confirmation
- `CNF`: Change confirmation (with status and old attributes)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Object ID (16-bit)
- `u8FontColour` (USINT): New font color
- `u8FontSize` (USINT): New font size
- `u8FontType` (USINT): New font
- `u8FontStyle` (USINT): New font style

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `u8OldFontColour` (USINT): Old font color
- `u8OldFontSize` (USINT): Old font size
- `u8OldFontType` (USINT): Old font
- `u8OldFontStyle` (USINT): Old font style
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` addresses the target object of the **F.28** command (Change Font Attributes). Valid is a **Font Attributes object**:

FontAttributesObject: 23000–23999
ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with object ID
- `INITO` confirms operational readiness

1. **Attribute Change**:

- `REQ` with new font attributes
- Changes the object's font properties
- `CNF` returns the result status and previous values

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.28) ✓ **Comprehensive font control** (color, size, type, style) ✓ **Feedback of previous values** for reversal options ✓ **Transaction-safe** (Atomic execution)

Attribute Properties

Feature	Description
Color Palette	8-bit Color Index (ISO 11783-6 A.3)
Sizes	Standardized Font Sizes
Fonts	Predefined Types
Styles	Bold/Italic/Underline Bit Field

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful Change
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	VT Not Ready
−21	VT_E_NO_INSTANCE	No VT Client Available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT Instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Text Display**: Dynamic font adjustments
- **Warnings**: Color highlighting
- **User Interfaces**: Style changes
- **Accessibility**: Font enlargements

⚖️ **Comparison with similar building blocks**

Feature	Q_FontAttributes	VtTextStyle	VtFontManager
ISO Standard	✓	✖	✖
Color Control	✓	✓	✖
Size Control	✓	✖	✓
Style Control	✓	✓	✓

Conclusion

The Q_FontAttributes module provides the standard implementation for font attribute changes:

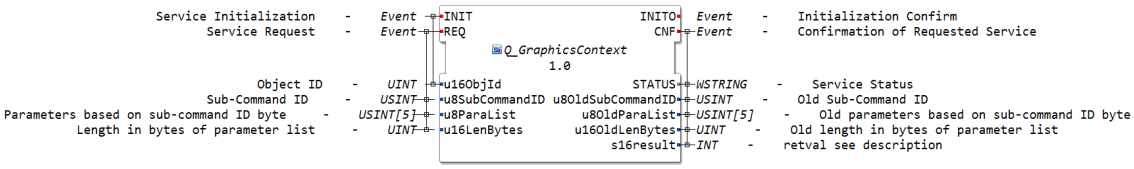
- **Versatile**: Complete control over font properties
- **Traceable**: Feedback of previous values
- **Robust**: ISO-compliant error handling

Essential for:

- Dynamic text rendering
- Accessible applications

- Design-flexible VT interfaces

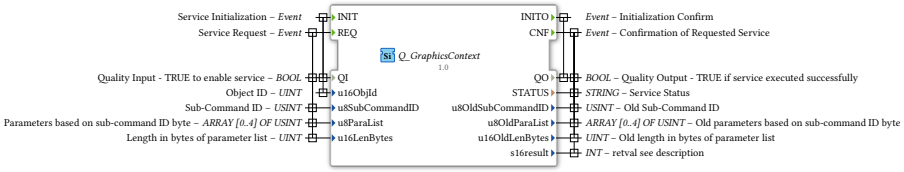
Q_GraphicsContext



• — ○ • — ○ • — ○ •

Introduction

The **Q_GraphicsContext** is a standards-compliant function block for controlling graphics context objects in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.56) specification for VT systems from version 4 onwards.



Interface Structure

Event Inputs

- `INIT`: Initialization request (with object ID)
- `REQ`: Configuration request (with sub-command and parameters)

Event Outputs

- `INITO`: Initialization acknowledgment
- `CNF`: Configuration acknowledgment (with status and previous values)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Graphics context object ID (16-bit)
- `u8SubCommandID` (USINT): Subcommand ID (8-bit)
- `u8ParaList` (USINT[5]): Parameter list (5-byte array)
- `u16LenBytes` (UINT): Parameter length in bytes

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operating status message
- `u8OldSubCommandID` (USINT): Previous subcommand ID
- `u8OldParaList` (USINT[5]): Previous parameter list
- `u16OldLenBytes` (UINT): Previous parameter length
- `s16result` (INT): ISO-compliant result code

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with graphics context object ID
- `INITO` confirms operational readiness

1. **Configuration**:

- `REQ` with sub-command ID and parameter list
- Configures graphics context properties

- `CNF` returns result status and previous configuration

1. **Error Handling**:

- ISO-standardized error codes
- Buffer overflow check

Technical Features

✓ **ISO 11783-6 compliant** (F.56) ✓ **Exclusive to VT Version 4+** ✓ **Flexible parameterization** (5-byte parameter array) ✓ **Traceability** (Previous configuration)

Sub-Command Reference

The sub-command IDs correspond to Table F.1 of ISO 11783-6 (not to be confused with earlier/derived numbering):

ID	Command	Parameter Description
0	Set Graphics Cursor	Bytes 5-6 X, 7-8 Y (signed)
1	Move Graphics Cursor	Bytes 5-6 X offset, 7-8 Y offset (signed)
2	Set Foreground Colour	Byte 5 colour index
3	Set Background Colour	Byte 5 colour index
4	Set Line Attributes Object ID	Bytes 5-6 object ID (65535 = no line)
5	Set Fill Attributes Object ID	Bytes 5-6 object ID (65535 = no fill)
6	Set Font Attributes Object ID	Bytes 5-6 object ID (65535 = no font)

⚠️ **Variable data length / Transport Protocol:** The F.56 command has a **variable data length**. Messages **larger than 8 bytes** are transmitted via the **Transport Protocol (TP)** smaller commands are padded to 8 bytes with FF16. With this FB's 5-byte parameter list (`u8ParaList`, ArraySize 5) the message becomes 9 bytes (⟶ TP). Sub-commands requiring more than 5 parameter bytes (e.g. Draw Polygon with several points, Pan & Zoom Viewport) cannot be fully expressed through this FB.

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful Execution
−6	VT_E_OVERFLOW	Parameter Buffer Too Small
−8	VT_E_NOACT	VT not ready
−21	VT_E_NO_INSTANCE	No VT client available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Graphics Rendering**: Line and Fill Styles
- **User Interfaces**: Transparency Effects
- **Charts**: Clipping Ranges for Excerpts
- **Animations**: Transformation Matrices

Safety Notes

⚠️ **Consider Buffer Size**: Parameter lists must not exceed `ISO_VTC_CMD_STR_MAX_LENGTH` ⚠️ **Sub-Command Validation**:
Unknown command IDs can lead to unexpected behavior

Conclusion

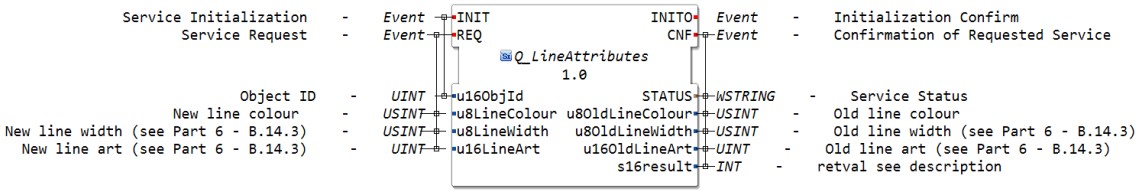
The Q_GraphicsContext block offers powerful graphics control:

- **Precise Control**: Over all graphics attributes
- **Traceable**: Previous State Backup
- **High Performance**: Optimized command load

Essential for:

- High-quality graphics rendering
- Dynamic visualizations
- Complex user interfaces

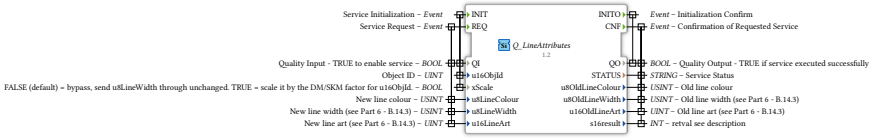
Q_LineAttributes



• — ○ • — ○ • — ○ •

Introduction

The **Q_LineAttributes** is a standards-compliant function module for modifying line attributes in Virtual Terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.30) specification for VT systems.



Interface Structure

Event Inputs

- `INIT`: Initialization request (with object ID and `xScale`)
- `REQ`: Attribute change request (with new line attributes)

Event Outputs

- `INITO`: Initialization confirmation
- `CNF`: Change confirmation (with status and old attributes)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Object ID (16-bit)
- `xScale` (BOOL, default `FALSE`): `FALSE` = bypass, `u8LineWidth` is sent through unchanged; `TRUE` = scale it by the DM/SKM factor for `u16ObjId`
- `u8LineColour` (USINT): New line color
- `u8LineWidth` (USINT): New line width
- `u16LineArt` (UINT): New line style

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `u8OldLineColour` (USINT): Old line color
- `u8OldLineWidth` (USINT): Old line width
- `u16OldLineArt` (UINT): Old line style
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` addresses the target object of the **F.30** command (Change Line Attributes). Valid is a **Line Attributes object**:

LineAttributesObject: 24000–24999
ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with object ID
- `INITO` confirms operational readiness

1. **Attribute Change**:

- `REQ` with new line attributes
- Changes Object Line Properties
- `CNF` provides result status and previous values

1. **Error Handling**:

- ISO-standardized error codes
- Detailed status messages

Technical Features

✓ **ISO 11783-6 compliant** (F.30)
✓ **Comprehensive line control** (color, width, style)
✓ **Feedback of previous values** for reversal options ✓ **Transaction-safe** (Atomic execution)

Attribute Reference

Attribute	Value range	Description
Color	0-255 (8-bit)	Color index according to ISO 11783-6 A.3
Width	1-255 pixels	Line thickness in pixels
Style	16-bit Bitmask	Line type according to B.14.3

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful change
−6	VT_E_OVERFLOW	Buffer overflow
−8	VT_E_NOACT	VT not ready
−21	VT_E_NO_INSTANCE	No VT client available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Graphic Design**: Dynamic Line Adjustments
- **Diagrams**: Highlighting Important Elements
- **User Guidance**: Visual Markers
- **Machine Visualization**: Status Indicators

⚖️ **Comparison with Similar Building Blocks**

Feature	Q_LineAttributes	VtLineStyle	VtGraphicAttributes
ISO Standard	✓	✖	✖

Color Control	✓	✓	✓
Width Control	✓	✓	✖
Style Control	✓	✓	✓

Conclusion

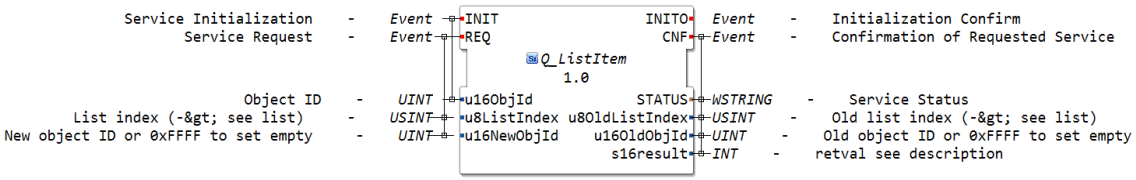
The Q_LineAttributes block offers precise control over line attributes:

- **Flexible**: Complete design freedom
- **Reliable**: ISO-compliant implementation
- **Traceable**: Previous state storage

Essential for:

- Technical drawings
- Process visualizations
- Interactive graphics applications

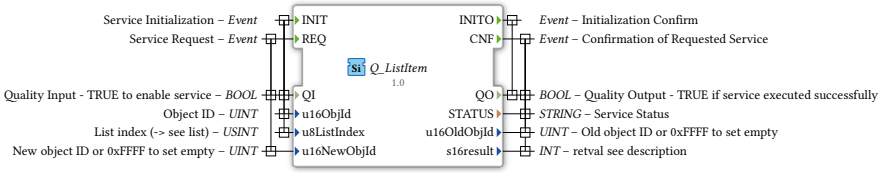
Q_ListItem



• — ○ • — ○ • — ○ •

Introduction

The **Q_ListItem** is a standards-compliant function module for managing list entries in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.42) specification for list input objects in VT systems.



Interface Structure

Event Inputs

- `INIT`: Initialization request (with list object ID `u16ObjId` and list index `u8ListIndex`)
- `REQ`: Change request (with new object ID `u16NewObjId`)

Event Outputs

- `INITO`: Initialization confirmation
- `CNF`: Change confirmation (with status and previous object ID)

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): List object ID (16-bit)
- `u8ListIndex` (USINT): List index (0-based, provided at `INIT`)
- `u16NewObjId` (UINT): New object ID or 0xFFFF (empty entry)

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `u16OldObjId` (UINT): Previous object ID
- `s16result` (INT): ISO-compliant result code

Instance Uniqueness

This block requires instance uniqueness regarding the pair (**u16ObjId**, **u8ListIndex**). Only one instance of `Q_ListItem` may exist in the entire program for the same list item entry. `u8ListIndex` is read at `INIT` — a second instance with the same list object ID and list index is deactivated during `INIT` (STATUS = "This objID is already in use"). See also Instance Uniqueness.

Valid Object IDs

`u16ObjId` (the list object) and `u16NewObjId` (the new list item) are subject to **different** rules.

`u16ObjId` — valid list object types (Annex F.42): Input List Field (10000–10999), Output List Object (37000–37999, VT v4+), Animation Object (44000–44999, VT v5+), External Object Definition (41000–41999, VT v5+).

`u16NewObjId` — new list item: any object ID (the item points to that object) or 0xFFFF (ID_NULL) to set an empty item. The validity of the new object ID is verified by the VT and reported back via F.43 (bit 2 = Invalid New List Item Object ID).

ID_NULL (65535) is not a valid command target for `u16ObjId` but deactivates the FB when used with `INIT`.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:
   - `INIT` with list object ID and list index
   - `INITO` confirms operational readiness
2. **List Change**:
   - `REQ` with new object ID
   - Updates the list item
   - `CNF` provides result status and previous object ID
3. **Special Case**:
   - `u16NewObjId = 0xFFFF` creates an empty entry

Technical Features

✓ **ISO 11783-6 compliant** (F.42) ✓ **Dynamic list management** (real-time changes) ✓ **Pairwise uniqueness at INIT** (`u16ObjId` + `u8ListIndex`) ✓ **Flexible empty items** (0xFFFF special value)

Index Reference

Index	Description
0	First list item entry
…	
255	Maximum index (uint8_t)

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful change
−6	VT_E_OVERFLOW	Buffer overflow
−8	VT_E_NOACT	VT not ready
−21	VT_E_NO_INSTANCE	No VT client available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

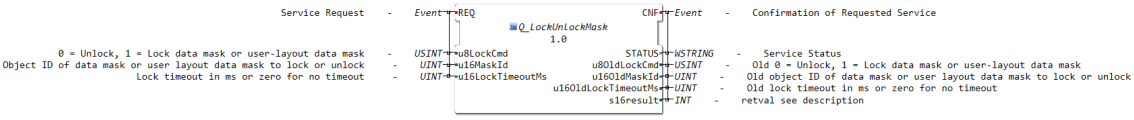
- **Dynamic Menus**: Context-dependent items
- **Device Selection**: Plug-and-play device lists
- **Parameter Settings**: Selection options
- **Diagnostic Lists**: Error code management

Conclusion

The Q_ListItem block provides efficient list management:

- **Reactive**: Instant GUI updates
- **Robust**: Integrated error handling
- **Compatible**: Full ISO compliance

Q_LockUnlockMask



• — ○ • — ○ • — ○ •

Introduction

The **Q_LockUnlockMask** is a standards-compliant function block for controlling the locking state of masks in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.46) specification for VT systems from version 4 onwards.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with mask object ID `u16MaskId`)
- `REQ`: Lock/Unlock Request (with lock command and timeout)

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Operation Acknowledgement

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16MaskId` (UINT): Mask Object ID (provided at `INIT`)
- `u8LockCmd` (USINT): Lock Command (0=Unlock, 1=Lock)
- `u16LockTimeoutMs` (UINT): Timeout in ms (0 = no timeout)

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `u8OldLockCmd` (USINT): Previous lock state
- `u16OldLockTimeoutMs` (UINT): Previous timeout
- `s16result` (INT): ISO-compliant result code

Instance Uniqueness

This block requires instance uniqueness regarding **u16MaskId**. Only one instance of `Q_LockUnlockMask` may exist in the entire program for the same mask ID. `u16MaskId` is read at `INIT` — a second instance with the same mask ID is deactivated during `INIT` (STATUS = “This objID is already in use”). See also Instance Uniqueness.

Valid Object IDs

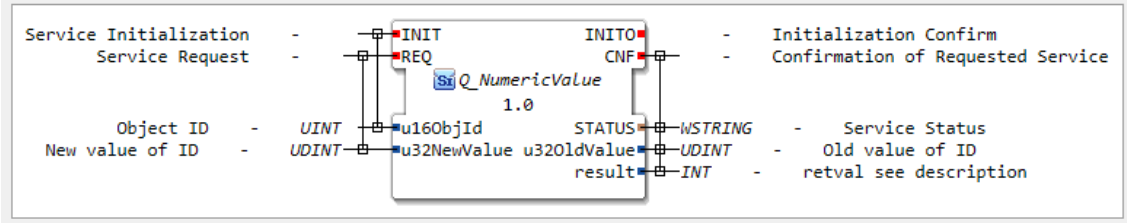
`u16MaskId` addresses the mask to lock/unlock (F.46). Valid are:
Data Mask (1000–1999) and **Window Mask / User-Layout Data Mask (34000–34999)**.
ID_NULL (65535) is not a valid command target — the command is answered by the VT with an error code (the mask must match the currently visible mask).

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:
   - `INIT` with `u16MaskId`
   - `INITO` confirms operational readiness
2. **Mask Locking**:
   - `REQ` with lock command and timeout
   - Controls the screen refresh of the mask
   - `CNF` provides operating status and previous values
3. **Timeout Behavior**:
   - Automatic unlock upon expiration

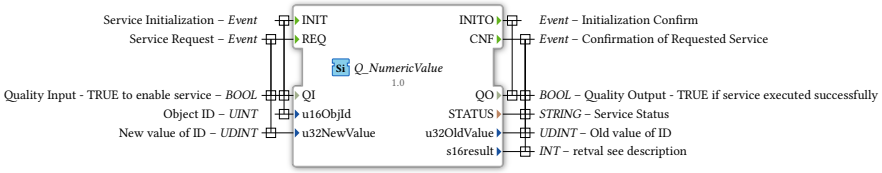
Q_NumericValue



• — ○ • — ○ • — ○ •

Introduction

The **Q_NumericValue** is a standards-compliant function block for changing numeric values in Virtual Terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.22) specification for numeric VT objects.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object ID)
- `REQ`: Value Change Request

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Change Acknowledgement

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Object ID (16-bit)
- `u32NewValue` (UDINT): New Numeric Value (32-bit unsigned)

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational Status Message
- `u32OldValue` (UDINT): Previous numeric value
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` — valid object types (Annex F.22, objects with numeric value attribute): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`. Any ID outside these ranges is invalid for commanding.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with target object ID
- `INITO` confirms operational readiness

1. **Value Update**:

- `REQ` with new 32-bit value
- Updates the numeric VT object
- `CNF` returns operational status and previous value

1. **Value Range**:

- 0 to 4,294,967,295 (32-bit unsigned)

Technical Features

✓ **ISO 11783-6 compliant** (F.22)
✓ **32-bit value range** (UDINT)
✓ **Instant update**
✓ **Traceability** (Previous value)
✓ **Internal buffering**: The function block buffers the value internally. A message is only sent to the bus if `u32NewValue` differs from `u32OldValue`. This significantly reduces the bus load and tolerates frequent REQ events.

Value range

Parameter	Type	Value range
u32NewValue	UDINT	0 to 4,294,967,295

Return codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful change
−6	VT_E_OVERFLOW	Buffer overflow
−8	VT_E_NOACT	VT not ready
−21	VT_E_NO_INSTANCE	No VT client available
−128	VT_E_HANDLE_INVALID	Invalid object ID
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance

–130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Process Visualization**: Real-time Measurement Data
- **Control Elements**: Setpoint Specifications
- **Diagnostic Systems**: Error Code Display
- **Production Data**: Counters and Statistics

⚖️ Comparison with Similar Function Blocks

Feature	Q_NumericValue	VtNumberUpdate	VtDataManager
ISO Standard	✓	✖	✖
Value Range	32-bit	16-bit	32-bit
Feedback	✓	✖	✓
Object Type	Numeric	All	All

🛠️ Related exercises

- Uebung_009
- Uebung_009a
- Uebung_011a
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_015
- Uebung_015a
- Uebung_020c2_sub
- Uebung_035
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- Uebung_039_sub_NumbAnsicht
- Uebung_040
- Uebung_040_2
- Uebung_040_AX
- Uebung_041
- Uebung_070
- Uebung_071
- Uebung_071a
- Uebung_071b
- Uebung_072
- Exercise_072b
- Exercise_072c
- Exercise_073
- Exercise_074
- Exercise_083

Conclusion

The Q_NumericValue block offers precise numerical control:

- **High-resolution**: 32-bit precision
- **Reliable**: Integrated error checking

- **Flexible**: For all numerical objects

Essential for:

- Precise process visualization
- Real-time data monitoring
- Industrial control systems

Example applications

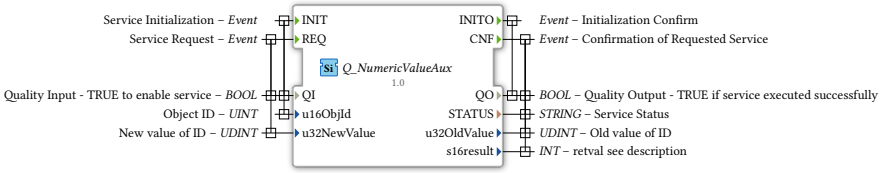
Q_NumericValue_examples

Q_NumericValueAux

• – ○ • – ○ • – ○ •

Introduction

The **Q_NumericValueAux** is a standards-compliant function block for numeric value changes in Virtual Terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.22) specification for numeric VT objects.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object ID)
- `REQ`: Value Change Request

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Change Acknowledgement

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Object ID (16-bit)
- `u32NewValue` (UDINT): New Numeric Value (32-bit unsigned)

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational Status Message
- `u32OldValue` (UDINT): Previous numeric value
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` — valid object types (Annex F.22, objects with numeric value attribute): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`. Any ID outside these ranges is invalid for commanding.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with target object ID
- `INITO` confirms operational readiness

1. **Value Update**:

- `REQ` with new 32-bit value
- Updates the numeric VT object
- `CNF` returns operational status and previous value

1. **Value Range**:

- 0 to 4,294,967,295 (32-bit unsigned)

Technical Features

✓ **ISO 11783-6 compliant** (F.22)
  ✓ **32-bit value range** (UDINT)
  ✓ **Instant update**
  ✓ **Traceability** (Previous value)

Value range

Parameter	Type	Value range
u32NewValue	UDINT	0 to 4,294,967,295

Return codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful change
−6	VT_E_OVERFLOW	Buffer overflow
−8	VT_E_NOACT	VT not ready
−21	VT_E_NO_INSTANCE	No VT client available
−128	VT_E_HANDLE_INVALID	Invalid object ID
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Measurement Displays**: Real-time data visualization
- **Meter Readings**: Odometer, production counter
- **Process Control**: Setpoint specifications
- **Diagnostic Systems**: Fault code display

⚖️ **Comparison with similar modules**

Feature	Q_NumericValueAux	VtNumericUpdate	VtValueManager
ISO Standard	✓	✖	✖
Value Range	32-bit	16-bit	32-bit
Feedback	✓	✖	✓
Object Type	Numeric Objects	All	All

Conclusion

The Q_NumericValueAux block offers precise numerical control:

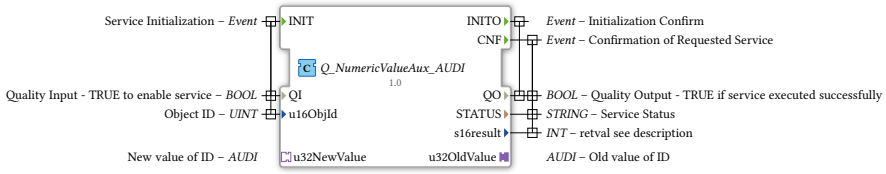
- **High-resolution**: 32-bit value range
- **Reliable**: Integrated error detection

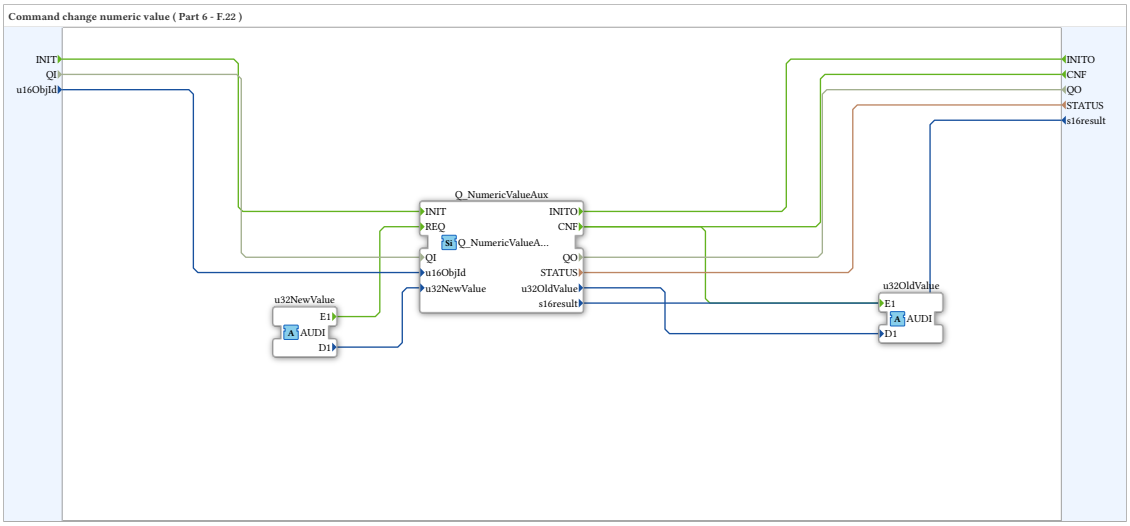
- **Flexible**: For all numeric objects

Essential for:

- Precise measurement display
- High-resolution control
- Professional process visualization

Q_NumericValueAux_AUDI





• — ○ • — ○ • — ○ •

Introduction

This function block implements the **Change Numeric Value** command according to ISO 11783-6 (Part 6, F.22). It is used to change the numeric value of an object and is only suitable for objects that have a numeric value attribute.

Interface Structure

Event Inputs

Event	Type	Comment	Carried Variable
`INIT`	EInit	Service Initialization	`u16ObjId`

Event Outputs

Event	Type	Comment	Carried Variable
`INITO`	EInit	Initialization Acknowledgement	–
`CNF`	Event	Acknowledgement of Requested Service	`STATUS`, `s16result`

Data Inputs

Variable	Type	Comment	Initial Value
`QI`	BOOL	Quality Input - TRUE to enable service	`TRUE`

u16ObjId	UINT	Object ID	ID_NULL

Data Outputs

Variable	Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Service Status
s16result	INT	Return Value (see description)

Adapters

Adapter	Direction	Type	Comment
u32OldValue	Plug	adapter::types::unidirectional::AUDI	Old Value of ID
u32NewValue	Socket	adapter::types::unidirectional::AUDI	New ID Value

Quality signals QI/QO: The data input QI (BOOL, initial value TRUE) is associated with the event INIT and is passed through to QI of the internal block Q_NumericValueAux. The data output QO (BOOL) is taken over from Q_NumericValueAux.QO and is associated with the events INITO and CNF.

Valid Object IDs

u16ObjId is valid for the same object types as the wrapped base block Q_NumericValueAux / Q_NumericValue (Annex F.22, objects with numeric value attribute): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) is not a command target but deactivates the FB when used with INIT.

Functionality

The function block (FB) expects a valid object ID (u16ObjId) via the INIT input. After successful initialization, a new value can be passed via the socket adapter (u32NewValue). The socket adapter event (E1) triggers internal processing, in which the old value is returned via the plug adapter (u32OldValue). The result is signaled via the CNF output with status and return code (s16result).

Possible return codes are:

- VT_E_NO_ERR (0) – Success
- VT_E_OVERFLOW (-6) – Buffer overflow
- VT_E_NOACT (-8) – Command not possible in the current state
- VT_E_NO_INSTANCE (-21) – No VT client available
- VT_E_ISO_INSTANCE_INVALID (-129) – Connection instance invalid
- VT_E_HANDLE_INVALID (-128) – Invalid handle
- VT_E_NOT_ALIVE (-130) – VT unreachable

Technical Features

- Use of unidirectional adapters for transmitting old and new values, enabling flexible, loose coupling in the application.
- Initialization with an object ID (u16ObjId) is mandatory.
- Returns ISO-11783-specific error codes, allowing for detailed error diagnosis.
- The function block internally contains a sub-FB (Q_NumericValueAux) that encapsulates the actual logic.

State Overview

The FB does not have any explicit state machines in its XML description. Its operation is event-driven:

1. **INIT** – Sets the object ID and initializes the internal logic.
2. **Waiting for Value** – After INIT, the function waits for an event from the socket adapter (u32NewValue.E1).

3. **Processing** – Upon arrival, the value is processed, the previous value is returned via the plug adapter, and the result is reported via `CNF`.

Application Scenarios

- Changing numerical attributes of objects in a Virtual Terminal (VT) according to the ISOBUS standard.
- Controlling agricultural machinery – e.g., setting target values for speeds, pressures, or positions within an ISOBUS network.

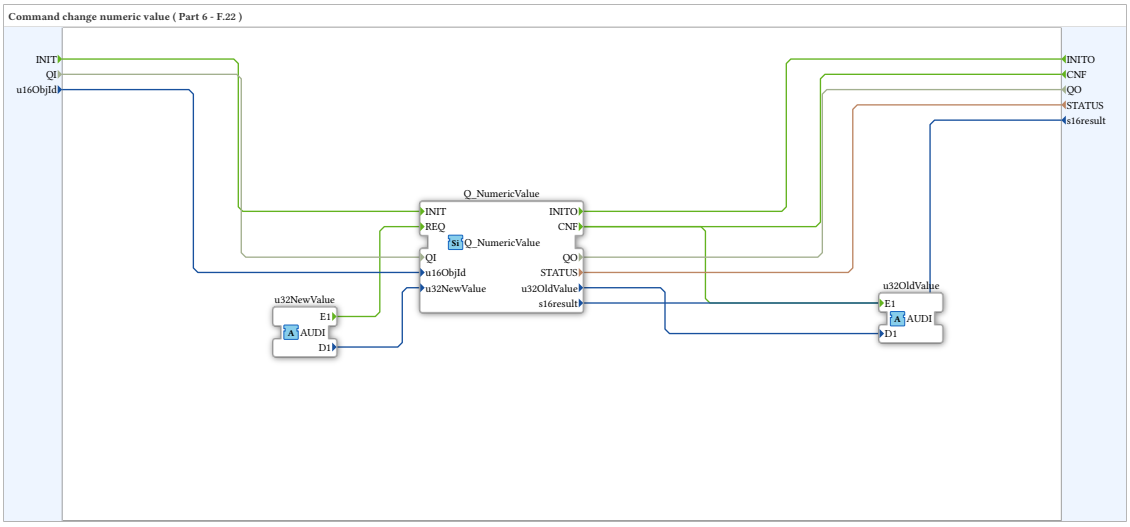
Comparison with Similar Modules

The basic module `Q_NumericValueAux` (without the suffix `_AUDI`) offers the same core functionality, but without the two unidirectional adapters. The `_AUDI` variant allows for separate and reusable connections of legacy and new values, which facilitates the development of modular and testable applications.

Conclusion

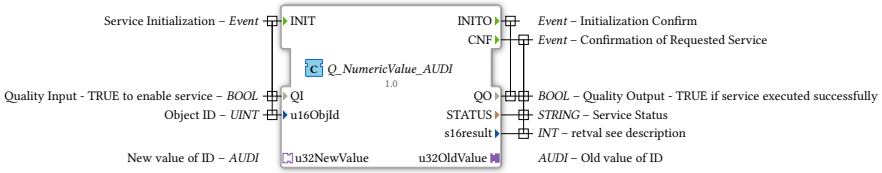
The `Q_NumericValueAux_AUDI` is a specialized function block for ISO 11783 communication. It encapsulates the "Change Numeric Value" command and provides a clean, decoupled interface for value transfer via adapters. With its detailed error feedback, it is particularly suitable for safety-critical applications in agricultural engineering.

Q_NumericValue_AUDI (ISOBUS Numeric Value Adapter)



Introduction

The `Q_NumericValue_AUDI` is a special ISOBUS function block that allows numeric values (`UDINT`) to be sent from a controller to a Universal Terminal (UT) for display or modification. It uses the `AUDI` adapter for transferring new and old values, ensuring a modern and type-safe connection.



Interface Structure

Event Inputs

- **INIT**: `EInit` - Initializes the service with `u16ObjId`. Associated with `QI`.

Event Outputs

- **INITO**: `EInit` - Confirms initialization. Associated with `QO`.
- **CNF**: `Event` - Confirms service execution. Associated with `QO`.
- Linked to variables `STATUS` and `s16result`.

Input Variables

- **QI**: `BOOL` - Quality Input - TRUE to enable service. Initial value TRUE.
- **u16ObjId**: `UINT` - The object ID of the numeric input field on the ISOBUS terminal (default: `ID_NULL`).

Output Variables

- **QO**: `BOOL` - Quality Output - TRUE if service executed successfully.
- **STATUS**: `STRING` - Service status message (e.g., "OK", "Error").
- **s16result**: `INT` - Service return value (error codes according to ISO 11783-6).

Adapter

- **u32OldValue** (Plug): `AUDI` - The old/current value of the object.
- **u32NewValue** (Socket): `AUDI` - The new value to be sent to the ISOBUS terminal.

Quality signals QI/QO: The data input `QI` (BOOL, initial value `TRUE`) is associated with the event `INIT` and is passed through to `QI` of the internal block `Q_NumericValue`. The data output `QO` (BOOL) is taken over from `Q_NumericValue.QO` and is associated with the events `INITO` and `CNF`.

Valid Object IDs

`u16ObjId` is valid for the same object types as the wrapped base block `Q_NumericValue` (Annex F.22, objects with numeric value attribute): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`.

Functionality

The function block is initialized via the `INIT` event with the object ID `u16ObjId`. As soon as an event arrives at the `u32NewValue` socket (`u32NewValue.E1`), the contained UDINT value (`u32NewValue.D1`) is sent to the ISOBUS terminal to update the displayed numeric value. After successful execution, a `CNF` event is triggered, and the object's previous value is provided via the `u32OldValue` plug.

Technical Features

✓ **ISO 11783-6 Standard**: Compliant implementation for ISOBUS Universal Terminals.
  ✓ **Adapter-based**: Uses the `AUDI` adapter for numerical values.
  ✓ **Asynchronous operation**: Event-driven communication.

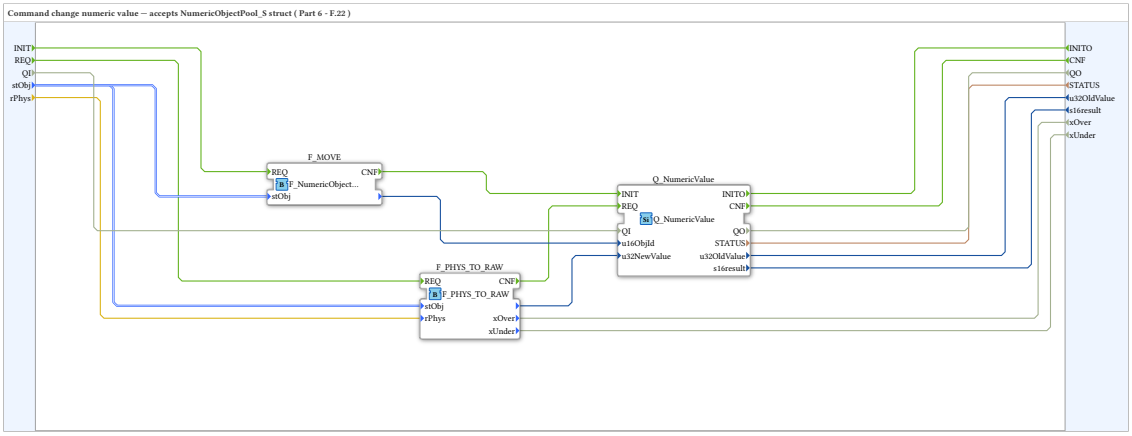
Application scenarios

- **Display of counter readings**: Visualization of counter readings (e.g., operating hours, quantities) on the terminal.
- **Setpoint specification**: Transfer of target values (e.g., temperature, pressure) from the ISOBUS terminal to the controller (in combination with a NumericValue_IDA input block).
- **Feedback systems**: Display of process values in real time.

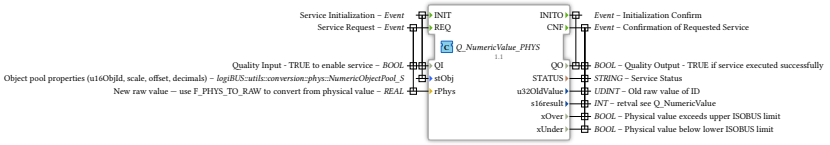
🛠️ **Related exercises**

- Exercise_009_AX
- Exercise_083_AX

Q_NumericValue_PHYS



Introduction



The function block `Q_NumericValue_PHYS` is a composite function block according to the standard **ISO 11783-6 (ISOBUS)**. It is used to change the numeric value of an ISOBUS object by specifying a physical value. The conversion from physical to raw data values is performed automatically based on the scaling, offset, and decimal places defined in the object structure `NumericObjectPool_S`. The function block encapsulates three sub-functions:

- **`F_MOVE`** – Intermediate storage of object parameters during initialization
- **`F_PHYS_TO_RAW`** – Conversion of physical values to raw integers (UDINT)
- **`Q_NumericValue`** – Actual ISOBUS write operation to the numeric object

Interface Structure

Event Inputs

Name	Type	Comment
`INIT`	EInit	Service Initialization – Loads the object parameters (`stObj`) into the function block
`REQ`	Event	Service Request – converts `rPhys` and writes the value to the ISOBUS object

Event Outputs

Name	Type	Comment
`INITO`	EInit	Initialization Confirmation
`CNF`	Event	Confirmation of Value Change – Contains Result Data

Data Inputs

Name	Type	Comment
`QI`	BOOL	Quality input, `TRUE` enables the service (default `TRUE`)
`stObj`	`logiBUS::utils::conversion::phys::NumericObjectPool_S`	Structure with object ID, scale, offset, and decimal places (initial value: ID_NULL, 1.0, 0, 0)

rPhys	REAL	Physical value to be set (e.g., temperature, pressure)

Data Outputs

Name	Type	Comment
QO	BOOL	Quality output, TRUE if the service was executed successfully
STATUS	STRING	Service status message (e.g., error or success message)
u32OldValue	UDINT	Original raw value of the ISOBUS object before the change
s16result	INT	Return value of the write operation (see Q_NumericValue)
xOver	BOOL	True if the physical value exceeds the upper limit of the ISOBUS value range
xUnder	BOOL	True if the physical value falls below the lower limit of the ISOBUS value range

Adapters

No adapters available.

Valid Object IDs

stObj.u16ObjId is valid for the same object types as the internally used Q_NumericValue (Annex F.22, objects with numeric value attribute): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) is not a command target but deactivates the FB when used with INIT.

Functionality

Quality handling (QI/QO): The input QI (BOOL, default TRUE) is associated with the events INIT and REQ; TRUE enables the service. The output QO (BOOL) is associated with INITO and CNF and indicates with TRUE that the service was executed successfully. QI is passed through to Q_NumericValue.QI, and Q_NumericValue.QO is passed through to QO.

Processing occurs in two separate steps:

1. **Initialization (event INIT):**

- The passed parameter stObj is temporarily stored via F_MOVE.
- Upon completion, Q_NumericValue.INIT is triggered, providing the object ID (u16ObjId) from F_MOVE.OUT.
- Initialization is acknowledged with INITO.

1. **Value Change (Event REQ):**

- The physical value rPhys, along with the stored structure stObj, is passed to F_PHYS_TO_RAW.
- F_PHYS_TO_RAW calculates the raw value (u32NewValue) as well as the limit flags xOver and xUnder.
- Subsequently, Q_NumericValue.REQ is triggered, which writes the calculated raw value to the ISOBUS object.
- The result data (STATUS, u32OldValue, s16result) are taken by Q_NumericValue and made available at the output.
- The event CNF signals completion.

Technical Features

- **Standard Compliance:** This module implements the ISO 11783-6 specification (Part 6, Annex F.22) – developed for agricultural ISOBUS applications.
- **Limit Violation Calculation:** The flags xOver / xUnder are determined during the conversion stage (F_PHYS_TO_RAW) and output in parallel with the actual write operation. This allows the user to identify early on whether the requested physical value is outside the permissible ISOBUS value range.

- **Typing:** The object ID is provided as `UINT` (16-bit), and the raw value as `UDINT` – this corresponds to the common ISOBUS convention for numeric attributes.
- **Reuse of Internal Components:** The division into `F_PHYS_TO_RAW` and `Q_NumericValue` allows for modular testability and reusability of the conversion logic.

State Overview

The function block does not have an explicit entry state machine, but implements its functionality as a data flow network. The process is strictly sequential:

- **Initialization Phase:**

INIT→F_MOVE.REQ→F_MOVE.CNF→Q_NumericValue.INIT→Q_NumericValue.INITO→INITO`

- **Operational Phase (Write):**

REQ→F_PHYS_TO_RAW.REQ→F_PHYS_TO_RAW.CNF→Q_NumericValue.REQ→Q_NumericValue.CNF→CNF`
During the execution of a run, the function block is not prepared for new events. The respective confirmation signal (`INITO` or `CNF`) must be awaited.

Application Scenarios

- **Vehicle terminal with ISOBUS connection:** Setting target values (e.g., working height, application rate) via user input in physical units (m, kg/h, °C).
- **Remote control of agricultural equipment:** Sending physical values from a controller to an ISOBUS device (e.g., seed drill) without manual raw value conversion.
- **Automated calibration:** Adjusting parameters during operation, where the scaling and offset are derived from a configuration structure (`NumericObjectPool_S`).

Comparison with similar modules

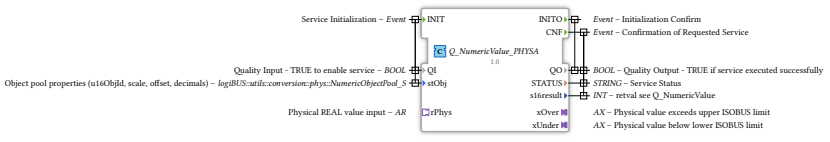
Module	Function	Difference to `Q_NumericValue_PHYS`
`Q_NumericValue` (from `isobus::UT::Q`)	Write a raw (already converted) value	Expects `u32NewValue` directly – without physical conversion and without limit checking
`F_PHYS_TO_RAW`	Pure conversion physical → raw	Returns only `xOver`, `xUnder`, and the raw value – no ISOBUS communication
`Q_NumericValue_PHYS`	Combined conversion + write access	Provides a complete interface for physical values in one step

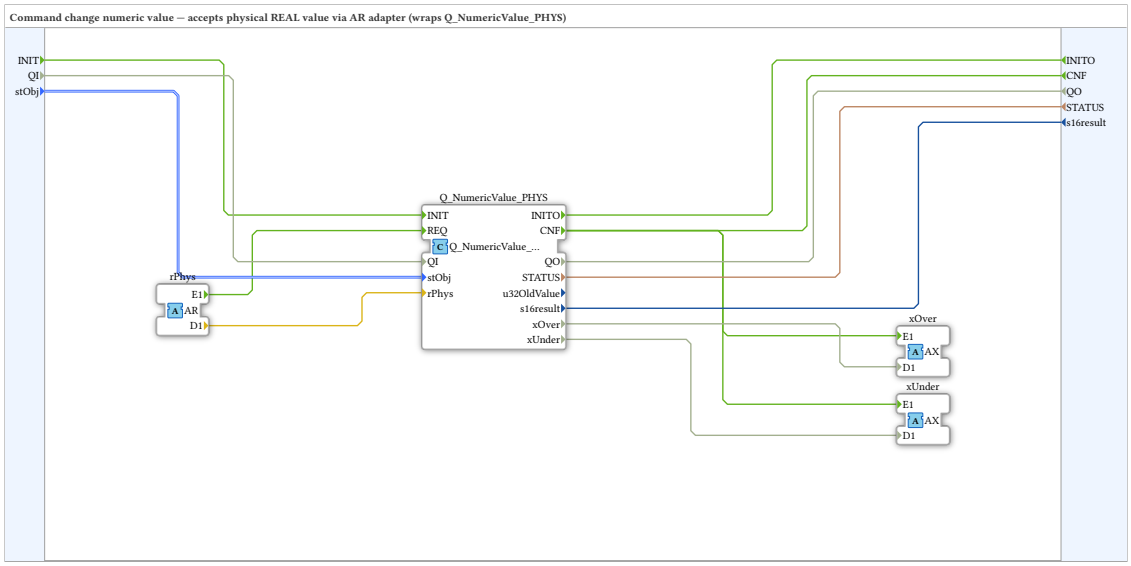
This function block simplifies application, as the user does not need to program a separate conversion step. It is particularly suitable for controllers that process values in common units.

Conclusion

Q_NumericValue_PHYS` is a practical, standards-compliant function block for ISOBUS communication. It combines the conversion of physical values with write access to numerical objects, thus reducing development effort in agricultural automation. The clear separation of tasks (storage, conversion, bus access) ensures the function block remains maintainable and testable. Outputting limit violation flags enables robust error handling in the application code.

Q_NumericValue_PHYSA





• — ○ • — ○ • — ○ •

Introduction

The function block `Q_NumericValue_PHYSA` serves as a wrapper for the function block `Q_NumericValue_PHYS`. It allows setting a numeric value, which is received as a physical value (`REAL`) via an AR adapter (socket `rPhys`). The function block is designed according to the ISO 11783-6 standard and is suitable for ISOBUS applications that require a physical value input.

Interface Structure

Event Inputs

Name	Type	Description
`INIT`	`EInit`	Service initialization; executed with data input `stObj`

Event Outputs

Name	Type	Description
`INITO`	`EInit`	Confirmation of successful initialization
`CNF`	`Event`	Confirmation of value change performed; Output together with `STATUS` and `s16result`

Data Inputs

Name	Type	Description
`QI`	`BOOL`	Quality Input - TRUE to enable service
`stObj`	`isobus::utils::conversion::phys::NumericObjectPoolObject`	Object pool properties: `u16ObjId` (Object ID), `r32Scale` (Scaling), `i32Offset` (Offset), `u8Decimals` (Decimal places). Default value: `(u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)`

Data Outputs

Name	Type	Description
`QO`	`BOOL`	Quality Output - TRUE if service executed successfully

STATUS	STRING	Service status message
s16result	INT	Return value (see documentation for Q_NumericValue)

Adapter

Type	Name	Direction	Description
adapter::types::unidirectional::AR	rPhys	Socket (Input)	Receives the physical value REAL for processing
adapter::types::unidirectional::AX	xOver	Plug (Output)	Signals that the physical value exceeds the upper ISOBUS limit
adapter::types::unidirectional::AX	xUnder	Plug (Output)	Signals that the physical value falls below the lower ISOBUS limit

Quality signals QI/QO: The data input QI (BOOL, initial value TRUE) is associated with the event INIT and is passed through to QI of the internal block Q_NumericValue_PHYS. The data output QO (BOOL) is taken over from Q_NumericValue_PHYS.QO and is associated with the events INITO and CNF.

Valid Object IDs

stObj.u16ObjId is valid for the same object types as the wrapped base block Q_NumericValue_PHYS / Q_NumericValue (Annex F.22, objects with numeric value attribute): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) is not a command target but deactivates the FB when used with INIT.

Functionality

1. After the **INIT** event, the function block is initialized with the object pool properties defined in stObj.
2. As soon as a new physical value arrives via the **rPhys** adapter (socket event E1), it is internally forwarded to the bound **Q_NumericValue_PHYS** block.
3. The Q_NumericValue_PHYS processes the value (taking scaling, offset, and decimal places into account) and triggers the **CNF** event.
4. The **CNF** event simultaneously outputs the status (STATUS), the return value (s16result), and the threshold indicators xOver and xUnder.
5. The outputs xOver and xUnder are provided as AX adapters to inform higher-level logic whether the entered value is outside the permissible ISOBUS range.

Technical Features

- The function block (FB) is a **pure adapter wrapper component**. The actual logic resides in the internally used Q_NumericValue_PHYS.
- The parameters for scaling and offset are configured via the **structure NumericObjectPool_S** – this allows for flexible adaptation to different physical units.
- Communication is **event-driven** via the adapters AR (value input) and AX (limit signal output). This enables modular integration into existing ISOBUS communication sequences.

State Overview

The function block itself does not have an explicit state machine. The initialized state is established by the first INIT event. Subsequent value changes via rPhys.E1 lead directly to processing and output. Error states are communicated via the output STATUS.

Application Scenarios

- **ISOBUS APP Control:** Setting a numeric value (e.g., setpoint for machine parameters) from a physical sensor value connected via an adapter.
- **Real to ISOBUS Format Conversion:** The function block handles the conversion of physical values to the internal integer format using scaling and offset.

- **Limit Monitoring:** The outputs `xOver` and `xUnder` allow the higher-level controller to react to exceedances or falls below limits.

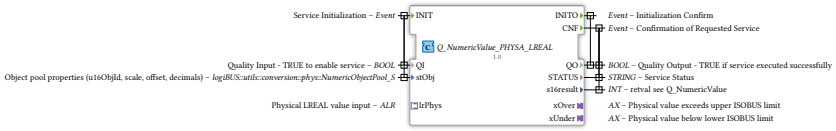
Comparison with Similar Function Blocks

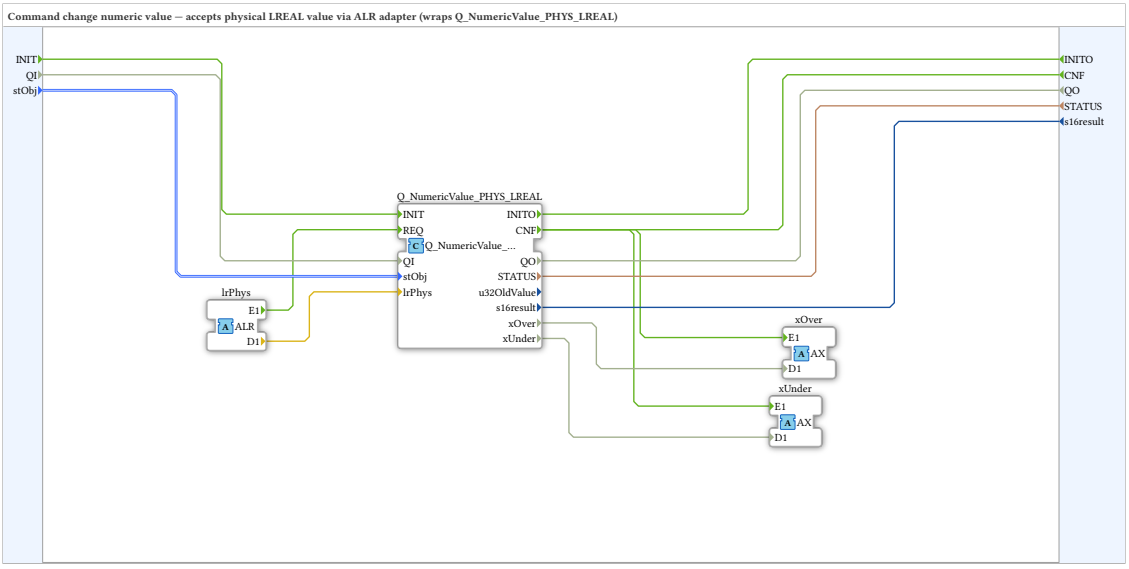
Function Block	Description	Difference
`Q_NumericValue_PHYS`	Direct function block for physical values	`Q_NumericValue_PHYSA` wraps this function block and adds explicit adapter outputs (`xOver`, `xUnder`) for limit signals
`Q_NumericValue`	Basic Function Block for Numeric Values (No Physical Conversion)	`Q_NumericValue_PHYSA` is specifically designed for real-world physical values and includes scaling/offset functionality.

Conclusion

The `Q_NumericValue_PHYSA` simplifies the integration of physical values into ISOBUS systems by encapsulating adapter communication and directly signaling limit values. Reusing the proven `Q_NumericValue_PHYS` ensures robust and standards-compliant logic.

Q_NumericValue_PHYSA_LREAL





• — ○ • — ○ • — ○ •

Introduction

The function block **Q_NumericValue_PHYSA_LREAL** serves as a command for changing a numeric value in the ISOBUS context (ISO 11783-6). It receives a physical LREAL value via the **ALR** adapter and handles the actual processing via the embedded function block **Q_NumericValue_PHYS_LREAL**. The function block uses adapter interfaces for physical value input as well as for signaling when the permissible range limits are exceeded or fallen below.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization of the function block (with stObj).

Event Outputs

Name	Type	Comment
INITO	EInit	Confirmation of successful initialization.
CNF	Event	Confirmation of the requested value change (contains STATUS and s16result).

Data Inputs

Name	Type	Comment
QI	BOOL	Quality Input - TRUE to enable service
stObj	logiBUS::utils::conversion::phys::NumericObjectPool_S	Properties of the object pool (ObjID, scale, offset, decimal places). Initial: (u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0).

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Service status message.
s16result	INT	Return value – see Q_NumericValue (result of the value change request).

Adapters

Sockets (Input Adapters)

Name	Type	Comment
lrPhys	ALR (unidirectional)	Input for the physical LREAL value (triggered via E1).

Plugs (Output Adapters)

Name	Type	Comment
xOver	AX (unidirectional)	Signals that the physical value exceeds the upper ISOBUS limit.
xUnder	AX (unidirectional)	Signals that the physical value falls below the lower ISOBUS limit.

Quality signals QI/QO: The data input `QI` (BOOL, initial value `TRUE`) is associated with the event `INIT` and is passed through to `QI` of the internal block `Q_NumericValue_PHYS_LREAL`. The data output `QO` (BOOL) is taken over from `Q_NumericValue_PHYS_LREAL.QO` and is associated with the events `INITO` and `CNF`.

Valid Object IDs

`stObj.u16ObjId` is valid for the same object types as the wrapped base block `Q_NumericValue_PHYS_LREAL` / `Q_NumericValue` (Annex F.22, objects with numeric value attribute): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`.

Functionality

1. **Initialization** (`INIT`): The function block is initialized with the object pool properties (`stObj`). The event is forwarded directly to the inner block `Q_NumericValue_PHYS_LREAL`. Upon successful initialization, `INITO` is triggered.
2. **Value Change**: As soon as the event `E1` arrives at the adapter `lrPhys`, the physical LREAL value is transferred to the inner block via the data connection `lrPhys.D1`, triggering the event `REQ`.
3. **Result**: The inner block performs the actual processing. The status (`STATUS`) and the return value (`s16result`) are output via `CNF`. Simultaneously, the plugs `xUnder` and `xOver` can signal whether the incoming value violates the defined ISOBUS limits.
4. The output events of the inner block (`INITO`, `CNF`, `xUnder.E1`, `xOver.E1`) are forwarded directly to the corresponding outputs of `Q_NumericValue_PHYSA_LREAL`.

Technical Features

- **Wrapper Design**: The block encapsulates the function block `Q_NumericValue_PHYS_LREAL` and provides a more convenient physical interface via adapters.
- **Adapter-based communication**: Value input and limit signaling are handled via unidirectional adapters (`ALR`, `AX`), which facilitates encapsulation and reuse in different contexts.
- **Configurable object pool parameters**: The object ID, scaling, offset, and decimal places are defined via `stObj`, allowing the block to be flexibly adapted to various sensor or actuator values.
- **Standard compliance**: The function block (FB) is developed according to ISO 11783-6 (ISOBUS) and optimized for agricultural applications.

State overview

An explicit state machine is not represented in the XML, as the FB is implemented as a pure network consisting of an internal building block. However, the typical behavior follows a simple sequence:

- **Idle** – waits for INIT.
- **Initialized** – after INIT and before the first value changes.
- **Processing** – from the E1 trigger (REQ) until the CNF event is received.
- **Completion** – output of STATUS/s16result via CNF, followed by readiness resumption.

Application Scenarios

- **ISOBUS Command**: Changing a setpoint (e.g., speed, position, pressure) in an agricultural control unit, where the value is available as a physical quantity (LREAL).
- **Limit Monitoring**: Used in applications that require additional alarms or responses when permissible range limits are exceeded (e.g., for sensor data) (via `xOver` / `xUnder`).
- **Scaled Values**: Using scaling and offset parameters to convert raw ISOBUS values into user-friendly physical units.

Comparison with Similar Function Blocks

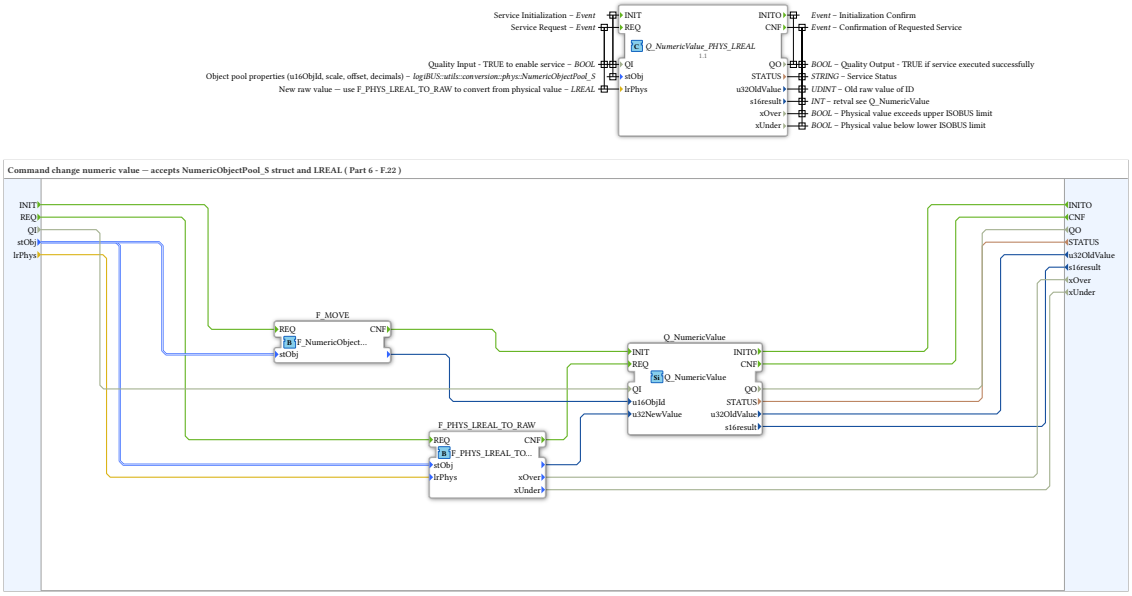
In direct comparison to the inner function block **Q_NumericValue_PHYS_LREAL**, the **Q_NumericValue_PHYSA_LREAL** offers a higher level of abstraction:

- Instead of direct input/output signals, communication is implemented via adapters – this allows for loose coupling in modular control architectures.
- The outer function block does not add its own logic, but simplifies integration through the adapter-based interface (e.g., `ALR` input).
- Unlike pure function blocks with fixed data ports, `Q_NumericValue_PHYSA_LREAL` can be flexibly adapted to different environments without having to change the signal paths in the network.

Conclusion

The **Q_NumericValue_PHYSA_LREAL** is a practical wrapper function block that simplifies physical value changes in the ISOBUS context via adapters. It combines the proven logic of `Q_NumericValue_PHYS_LREAL`With a convenient, adapter-based interface for LREAL inputs and limit signaling. Thanks to its configurable object pool parameters and compliance with ISO 11783-6, it is particularly suitable for use in modular agricultural control systems.

Q_NumericValue_PHYS_LREAL



• — ○ • — ○ • — ○ •

Introduction

The function block **Q_NumericValue_PHYS_LREAL** is used to set a numeric value as a physical quantity via ISOBUS (ISO 11783-6). It receives a physical value of type `LREAL`, automatically converts it into the required raw value, and sends the corresponding command to the connected device. This complies with the specification in Part 6, Annex F.22. The function block encapsulates the necessary steps of the physical conversion and the actual command execution, allowing the user to work directly with physical units.

Interface Structure

Event Inputs

Event	Description
`INIT`	Initializes the function block with the object pool properties (`stObj`).
`REQ`	Starts processing: the physical value (`lrPhys`) is sent to the target object.

Event Outputs

Event	Description
`INITO`	Acknowledges successful initialization.
`CNF`	Acknowledges command execution; the output data is valid.

Data Inputs

Name	Type	Description
`QI`	BOOL	Quality input, `TRUE` enables the service (default `TRUE`)
`stObj`	`logiBUS::utils::conversion::phys::NumericObjectPool_S`	Object pool properties (object ID, scale, offset, decimal places). Default value: `(u16ObjId := ID_NULL, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)`.

lrPhys	LREAL	The physical value (e.g., pressure, temperature) to be sent. Note: Before transmission, the value should be converted using F_PHYS_LREAL_TO_RAW if necessary; the function block performs this conversion automatically internally.

Data Outputs

Name	Type	Description
QO	BOOL	Quality output, TRUE if the service was executed successfully
STATUS	STRING	Status message of the service performed.
u32OldValue	UDINT	Original raw value of the object before the change.
s16result	INT	Return value (see Q_NumericValue).
xOver	BOOL	TRUE if the physical value exceeds the upper ISOBUS limit.
xUnder`	BOOL	TRUE if the physical value falls below the lower ISOBUS limit.

Adapter

None.

Valid Object IDs

stObj.u16ObjId is valid for the same object types as the internally used Q_NumericValue (Annex F.22, objects with numeric value attribute): Input Boolean Field (7000–7999), Input Number Field (9000–9999), Input List Field (10000–10999), Output Number Field (12000–12999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Number Variable (21000–21999), Object Pointer (27000–27999), Output List Object (37000–37999), External Object Pointer (43000–43999), Animation Object (44000–44999), Scaled Graphic Object (48000–48999).

ID_NULL (65535) is not a command target but deactivates the FB when used with INIT.

Functionality

Quality handling (QI/QO): The input QI (BOOL, default TRUE) is associated with the events INIT and REQ; TRUE enables the service. The output QO (BOOL) is associated with INITO and CNF and indicates with TRUE that the service was executed successfully. QI is passed through to Q_NumericValue.QI, and Q_NumericValue.QO is passed through to QO.

The function block works internally with three subordinate function blocks:

1. **F_MOVE** – On a INIT event, the passed structure stObj (object ID, scale, offset, decimal places) is copied to the internal Q_NumericValue.
2. **F_PHYS_LREAL_TO_RAW** – On a REQ event, the physical value lrPhys is converted into an ISOBUS raw value using the parameters stored in stObj.
3. **Q_NumericValue** – The generated raw value, along with the object ID, is then passed to the actual command block, which executes the command on the bus.

The outputs STATUS, u32OldValue, and s16result originate directly from Q_NumericValue. The overflow/underflow messages (xOver, xUnder) are provided by the conversion function.

Technical Features

- **Automatic Conversion**: The user does not need to manually convert physical values into raw values; this is done transparently within the function block.
- **Default Values**: If no specific object pool properties are passed, the function block uses sensible defaults (scaling 1.0, offset 0, no decimal places). The object ID is then ID_NULL.
- **Limit Check**: The outputs xOver and xUnder signal whether the passed physical value is outside the range permitted for ISOBUS. This allows applications to react early.
- **Cache Storage**: The initialization (INIT) copies the object properties only once and stores them for later REQ calls.

State Overview

The function block does not have an explicit state machine in the sense of an ECC, but operates event-driven according to the following logic:

State / Sequence	Description
Initialization	After a `INIT` event, the object properties are stored internally. Then, `INITO` is sent.
Send Command	After a `REQ` event, the physical value is converted, the command is issued, and upon completion, `CNF` is sent with the result data.
Error Handling	If an overflow/underflow occurs during the conversion, `xOver` or `xUnder`, respectively, are set before the command is issued. An erroneous command is signaled by `s16result` and the status message.

Application Scenarios

- **Control of ISOBUS Devices**: Setting setpoints for actuators (e.g., valves, drives) in physical units such as pressure (bar, kPa), temperature (°C), or fill level.

Error Handling - **Agricultural Machinery**: Change operating parameters (e.g., application rate, speed) directly from a control application.

- **Test and Simulation Environments**: Easily send physical values without having to worry about raw value conversion.

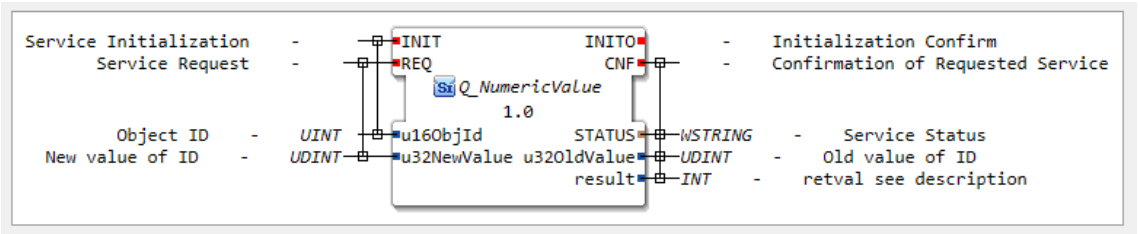
Comparison with Similar Blocks

- **`Q_NumericValue` (without PHYS)**: Expects the raw value (UDINT) directly. The user must perform the conversion manually. `Q_NumericValue_PHYS_LREAL` encapsulates this step and improves the readability and maintainability of applications that work with physical quantities.
- **`Q_NumericValue_PHYS_REAL`** (analogous for REAL type): Functions identically, but with single-precision accuracy. For applications requiring higher accuracy (LREAL), this block is the right choice.

Conclusion

Q_NumericValue_PHYS_LREAL` is a practical and standards-compliant function block that simplifies integration with the ISOBUS system. The integration of physical conversion eliminates manual conversion logic, reducing the potential for errors and increasing reusability. This module is particularly suitable for control applications where physical values must be transmitted with high accuracy.

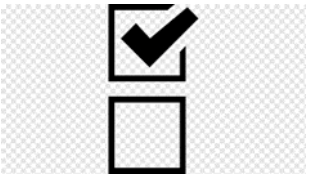
Q_NumericValue



Normative Reference: This function block refers to section ISO 11783-6 F.22 Change Numeric Value command CAUTION: This command can be called as often as desired by the application programmer; however, excessively frequent calls can severely overload the ISOBUS, therefore this function block should be used with caution.

The purpose of this function block is to update one of the following objects displayed on a VT:

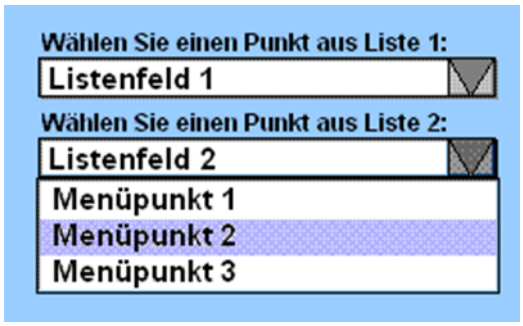
- Boolean input field: 1 byte for TRUE/FALSE



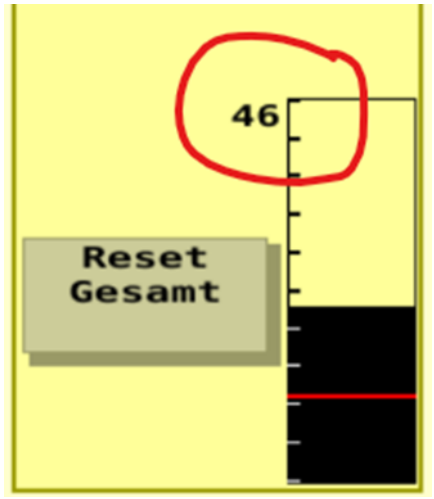
- Number input field: 4 bytes for integer input



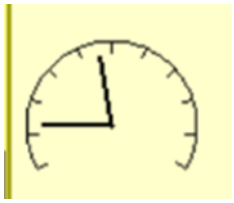
- List input field: 1 byte for list index



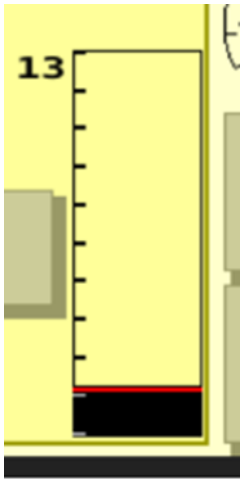
- List output field: 1 byte for list index
- (same as list input field, but not present in VT3)
- Number output field: 4 bytes for integer output



- Meter: 2 bytes for integer value



- Linear bar graph: 2 bytes for integer value



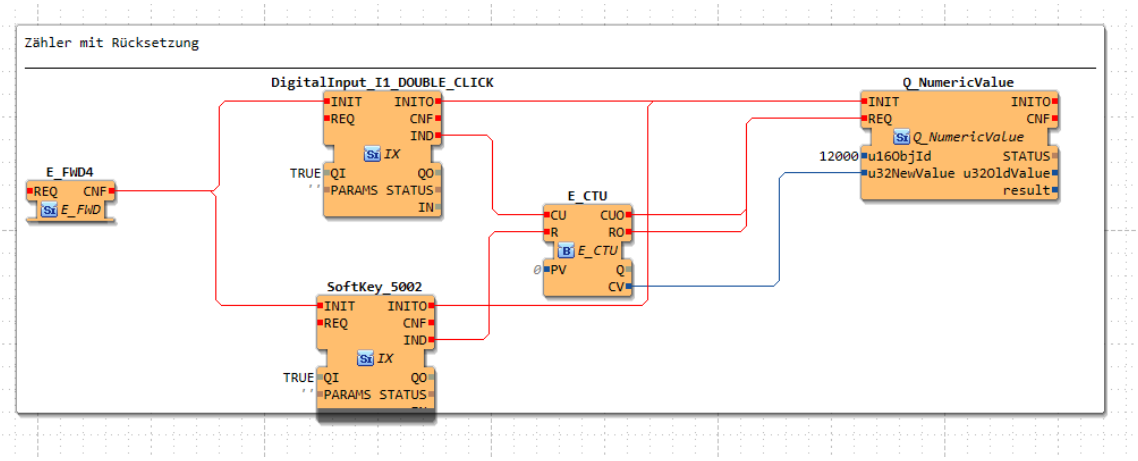
- Arched bar graph: 2 bytes for integer value



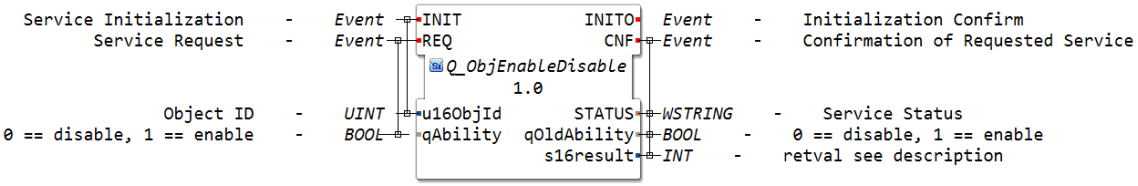
- Number variable: 4 bytes for integer value
- The Number variable cannot be displayed directly on the screen, but it can be linked to all previously mentioned objects; advantage: multiple links are possible.
- Object pointer: 2 bytes for Object ID
- Useful for showing and hiding other objects instead of the pointer.



Cross-references:
EXAMPLE
Counter with reset



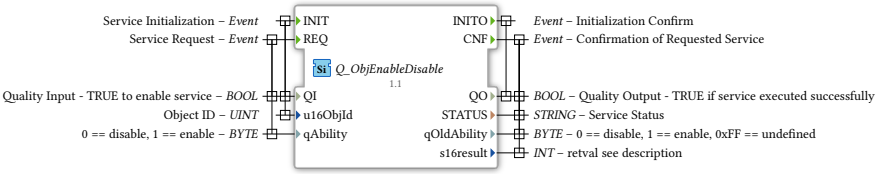
Q_ObjEnableDisable



• — ○ • — ○ • — ○ •

Introduction

The **Q_ObjEnableDisable** is a standards-compliant function block for enabling/disabling input objects in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.4) specification for VT input elements.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object ID)
- `REQ`: Activation/Deactivation Request

Event Outputs

- `INITO`: Initialization Confirmation
- `CNF`: Operation Confirmation

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Object ID (16-bit)
- `qAbility` (BOOL): Activation Status (TRUE=activate, FALSE=deactivate)

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `qOldAbility` (BOOL): Previous activation status
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` — valid object types (Annex F.4): Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Animation Object (44000–44999).

ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`. Any ID outside these ranges is invalid for commanding.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with target object ID
- `INITO` confirms operational readiness

1. **Status Change**:

- `REQ` with desired activation state
- Changes the object's interaction capability
- `CNF` returns operational status and previous state

1. **Special Cases**:

- VT_E_HANDLE_INVALID is returned for invalid object IDs

Technical Features

✓ **ISO ISO 11783-6 compliant** (F.4) ✓ **Simple Binary Control** (Enabled/Disabled) ✓ **State Feedback** (Previous State) ✓ **Broad Object Support** (All Input Elements)

Status Reference

qAbility	Object State
FALSE (0)	Disabled (gray)
TRUE (1)	Enabled (normal)

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful Execution
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	VT not ready
−21	VT_E_NO_INSTANCE	No VT client available
−128	VT_E_HANDLE_INVALID	Invalid object ID
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Form Control**: Conditional input approval
- **Security Features**: Block critical input
- **Workflow Management**: Step-by-step approval
- **Diagnostic Modes**: Input restrictions

⚖️ Comparison with similar building blocks

Feature	Q_ObjEnableDisable	VtInputControl	VtObjectState
ISO Standard	✓	✖	✖
Binary Control	✓	✓	✖
Feedback	✓	✖	✓
Object Type	Input Objects	All	All

Conclusion

The Q_ObjEnableDisable block provides essential input control:

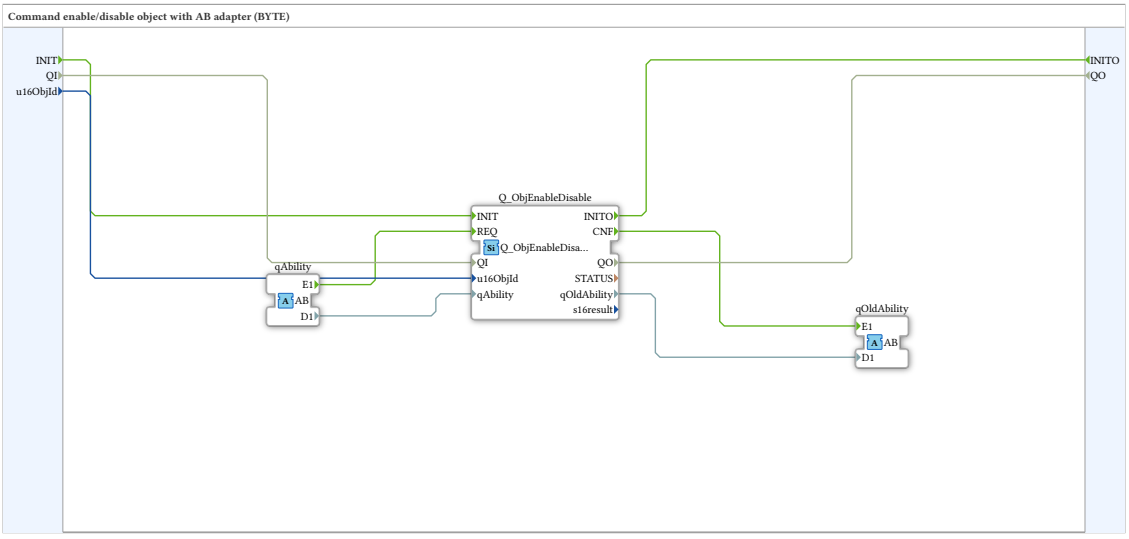
- **Simple**: Clear activation logic
- **Reliable**: Integrated error detection
- **Flexible**: For all input element types

Essential for:

- Interactive forms
- Secure user interfaces
- State-driven applications

Q_ObjEnableDisable_AB





• – ○ • – ○ • – ○ •

Introduction

The function block `Q_ObjEnableDisable_AB` serves as a wrapper that enables the control of a binary object (switching on/off) via a unidirectional **AB adapter (BYTE)**. It encapsulates the logic of the block `isobus::UT::Q::Q_ObjEnableDisable` and provides its functionality via the standardized adapter interfaces `qAbility` (socket) and `qOldAbility` (plug). The function block is suitable for connecting to adapter-based systems, e.g., for the remote control of actuators.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Service initialization: starts the internal function block and passes the object ID.

Event Outputs

Event	Type	Description
INITO	EInit	Confirmation of successful initialization.

Data Inputs

Name	Type	Description
QI	BOOL	Quality Input - TRUE to enable service
u16ObjId	UINT	Unique identifier of the object to be controlled.

Data Outputs

Apart from QO, there are no dedicated data outputs – the remaining output data is provided via the adapter plug.

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully

Adapter

Adapter Type	Direction	Description
Socket qAbility	Input	Unidirectional AB adapter: receives the command (BYTE). 0 = disable, 1 = enable, 0xFF = undefined.
Plug qOldAbility	Output	Unidirectional AB adapter: returns the previous state of the object (same encoding).

Quality signals QI/QO: The data input QI (BOOL, initial value TRUE) is associated with the event INIT and is passed through to QI of the internal block Q_ObjEnableDisable. The data output QO (BOOL) is taken over from Q_ObjEnableDisable.QO and is associated with the event INITO.

Valid Object IDs

The validity rules of the internal base function block Q_ObjEnableDisable apply:

u16ObjId — valid object types (Annex F.4): Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Animation Object (44000–44999).

ID_NULL (65535) is not a valid command target but deactivates the FB when used with INIT.

Functionality

The FB acts as a bridge between an AB adapter (BYTE) and the internal function block Q_ObjEnableDisable. The processes are as follows:

1. **Initialization**

An INIT event with a valid object ID (u16ObjId) is forwarded to the internal FB. The internal FB is initialized and reports this via INITO.

1. **Control**

A BYTE value is received via socket qAbility. The event qAbility.E1 triggers the internal FB (event REQ). The data value from qAbility.D1 is passed to the internal function block (FB).

The internal FB processes the command and sets the object's state accordingly. The previous state is passed via the internal connection to the plug `qOldAbility`. The event `CNF` of the internal FB is output as `qOldAbility.E1`.

1. **Feedback**

The plug `qOldAbility` provides the object's previous state (BYTE) and the associated event, allowing a downstream function block to react to state changes.

Technical Features

- **Adapter-Based Encapsulation**

The FB uses the standardized unidirectional AB adapter, which allows for easy integration into existing adapter networks.

- **Typical BYTE Encoding**

0→ `disable`,1→ `enable`,0xFF` ⟶ undefined (error or no change). This convention is used for both input and output.

- **Initialization with Object ID Only**

The INIT event must be triggered with a valid `u16ObjId`. The function block itself does not store any persistent configuration.

- **Event-Driven Processing**

Control is exclusively via the events of the adapter socket. No state change occurs without an event on `qAbility.E1`.

State Overview

The function block does not have its own explicit state machine – the state logic resides entirely within the internal function block `Q_ObjEnableDisable`. Nevertheless, the following operating states can be derived:

State	Description
Idle	After successful initialization (INITO sent), the function block (FB) waits for an event at socket `qAbility`.
Processing	An incoming event at `qAbility.E1` is processed; the internal FB executes the switchover.
Completed	Processing complete – the new (or old) state is available at the plug, and event `qOldAbility.E1` is output.

Error states (e.g., invalid object ID, faulty INIT sequence) are handled by the internal FB and can be signaled via return values or special BYTE codes (`0xFF`).

Application Scenarios

- **Remote Control of a Binary Output**

A higher-level control block sends the command "On" or "Off" via an AB adapter. The FB executes the command and reports back the previous state.

- **State Query Without Change**

By sending the code `0xFF`, the current state can be queried without making any changes.

- **Integration into Adapter-Based Service Interfaces**

The FB serves as a converter between an AB adapter (e.g., from an HMI or remote control) and a functional block that requires a native ISOBUS interface.

Comparison with Similar Blocks

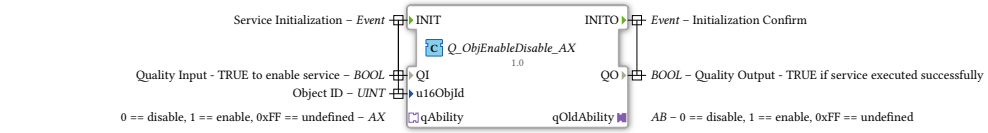
Block	Difference
`Q_ObjEnableDisable` (direct)	Requires native event and data interfaces without adapters. `Q_ObjEnableDisable_AB` adds adapter compatibility.

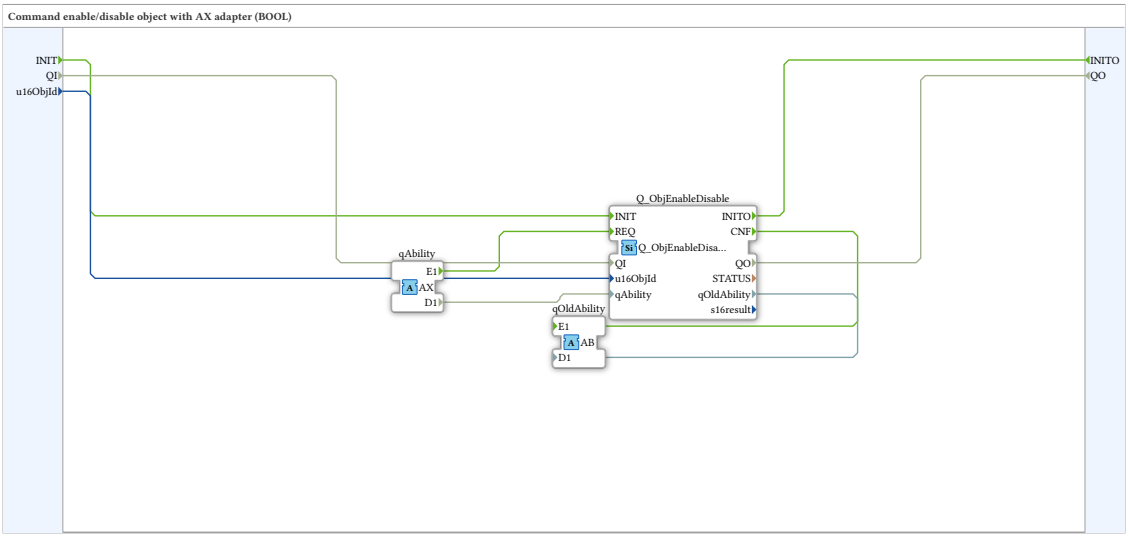
`Q_ObjEnableDisable_BB` (hypothetical)	Would use a bidirectional BB adapter. This FB uses the simpler unidirectional AB adapter.
Generic “CommandOnOff”	Is typically not targeted to a specific object (ID) and does not provide status feedback via adapters.

Conclusion

The FB `Q_ObjEnableDisable_AB` is a useful wrapper that makes the functionality of an ISOBUS object switch accessible via the standardized AB adapter. It facilitates integration into modular adapter networks and reduces wiring effort. The simple interface with only one event input for control and feedback via the plug makes it particularly suitable for reusable components in automation technology.

Q_ObjEnableDisable_AX





• – ○ • – ○ • – ○ •

Introduction

The function block **Q_ObjEnableDisable_AX** is a composite function block that serves as an adapter wrapper for the internal function block `Q_ObjEnableDisable`. It enables the control of an enable/disable command (BOOL) via a unidirectional AX adapter (socket) and returns the previous state via an AB adapter (plug). This function block is typically used in ISOBUS-based control systems.

Interface Structure

Event Inputs

Event	Type	Comment
`INIT`	EInit	Initialization of the function block; the object ID (u16ObjId) is passed.

Event Outputs

Event	Type	Comment
`INITO`	EInit	Confirmation of successful initialization.

Data Inputs

Name	Type	Comment
QI	BOOL	Quality Input - TRUE to enable service
u16ObjId	UINT	Object ID set during the INIT event.

Data Outputs

Apart from QO, there are no direct data outputs; the previous state is output via the adapter plug qOldAbility.

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully

Adapter

Name	Type	Direction	Comment
qAbility	adapter::types::unidirectional::AX	Socket	Receives the Enable/Disable command (0 = disable, 1 = enable, 0xFF = undefined).
qOldAbility	adapter::types::unidirectional::AB	Plug	Returns the object's previous state (same encoding).

Quality signals QI/QO: The data input QI (BOOL, initial value TRUE) is associated with the event INIT and is passed through to QI of the internal block Q_ObjEnableDisable. The data output QO (BOOL) is taken over from Q_ObjEnableDisable.QO and is associated with the event INITO.

Valid Object IDs

The validity rules of the internal base function block Q_ObjEnableDisable apply:

u16ObjId — valid object types (Annex F.4): Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Animation Object (44000–44999).

ID_NULL (65535) is not a valid command target but deactivates the FB when used with INIT.

Functionality

1. **Initialization**

An INIT event at input INIT sets the desired object ID (u16ObjId) and initializes the internal block Q_ObjEnableDisable. After successful initialization, the event INITO is output.

1. **Signal Processing**

- A new command is received via the socket adapter qAbility:
- The adapter's event E1 triggers the REQ input of the internal block.
- The data D1 (0/1/0xFF) is forwarded to the input qAbility of the internal block.
- The internal block processes the command, activates or deactivates the object with the given ID, and outputs the previous state.
- The acknowledgment CNF of the internal block is transferred to the event E1 of the plug adapter qOldAbility.
- The previous state is provided via the data output D1 of the plug adapter.

1. **Output of Previous State**

The adapter plug qOldAbility provides the state before the last change as a BOOL value (0/1/0xFF). The event connection ensures that the output event occurs at the correct time.

Technical Features

- This function block encapsulates the logic of the more complex Q_ObjEnableDisable and offers a simplified interface via two standardized adapters (AX/AB).
- Adapter socket qAbility expects unidirectional communication (value transfer only), while adapter plug qOldAbility returns the result.

- The object ID is set only during INIT; changing the ID during operation is not supported.

State Overview

The FB `Q_ObjEnableDisable_AX` itself does not have its own state machine. The internal state logic is implemented by the included function block `Q_ObjEnableDisable`. Its typical states are:

- **IDLE** – Waiting for a command (REQ).
- **Processing** – Processing the command (e.g., sending a CAN message).
- **Done** – Command completed, output of the previous state, and return to IDLE.

Application Scenarios

- **ISOBUS Control**: Enabling/disabling an agricultural device (e.g., valve, lighting) based on a Boolean signal from an AX adapter.
- **Universal Adapter**: Integrating enable/disable functions into systems that use standardized unidirectional adapters.
- **State Feedback**: Capturing the previous state for diagnostic or monitoring purposes.

Comparison with Similar Function Blocks

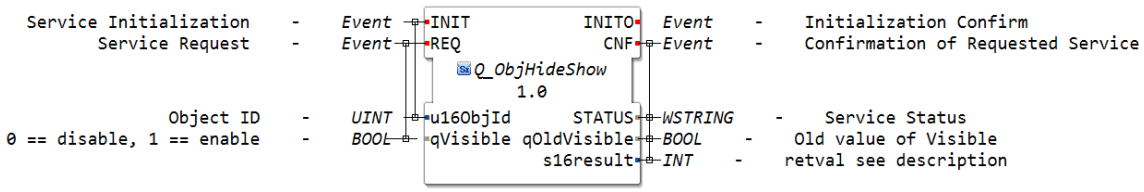
Function Block	Interface	Special Feature
Q_ObjEnableDisable	Direct event/data inputs	Requires separate Boolean inputs and events.
Q_ObjEnableDisable_AX	AX/AB Adapter	Easy integration into adapter-based architectures; reduces wiring effort.

The AX wrapper does not add new functionality but optimizes the connection to existing adapter communication.

Conclusion

The **Q_ObjEnableDisable_AX** function block is a practical adapter wrapper that simplifies the control of an ISOBUS object using enable/disable commands via a unidirectional AX adapter. It encapsulates the internal logic, enables clean interface separation, and returns the previous state via an AB adapter. This function block is particularly suitable for modular control architectures based on standardized adapter types.

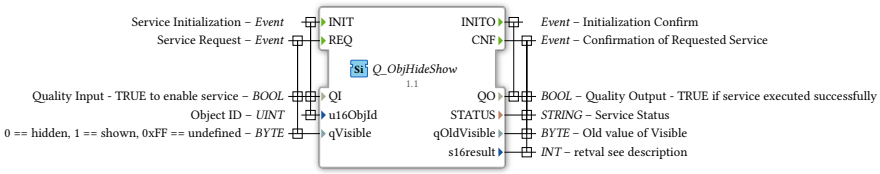
Q_ObjHideShow



• — ○ • — ○ • — ○ •

Introduction

The **Q_ObjHideShow** is a standards-compliant function block for controlling the visibility of container objects in Virtual Terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.2) specification for VT containers.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with Container Object ID)
- `REQ`: Visibility Change Request

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Change Acknowledgement

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Container Object ID (16-bit)
- `qVisible` (BOOL): Visibility Status (TRUE=show, FALSE=hide)

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `qOldVisible` (BOOL): Previous visibility status
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` — valid object types (Annex F.2): Container (3000–3999).

ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`. Any ID outside this range is invalid for commanding.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with container object ID
- `INITO` confirms operational readiness

1. **Visibility Change**:

- `REQ` with desired visibility status
- Changes the display of the container and all contained objects
- `CNF` returns operational status and previous state

1. **Special Cases**:

- VT_E_HANDLE_INVALID is returned for invalid object IDs

Technical Features

✓ **ISO 11783-6 compliant** (F.2) ✓ **Cascading effect** (Affects contained objects) ✓ **Instant GUI update** ✓ **State retention** (Remembers previous state)

State reference

qVisible	Container state
FALSE (0)	Invisible
TRUE (1)	Visible

Return codes (s16result)

Code	Constant	Meaning				
0	VT_E_NO_ERR	Successful execution				

−6	VT_E_OVERFLOW	Buffer overflow				
−8	VT_E_NOACT	VT not ready		−21	VT_E_NO_INSTANCE	No VT client available
−128	VT_E_HANDLE_INVALID	Invalid container ID				
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance				
−130	VT_E_NOT_ALIVE	VT not active				

Application Scenarios

- **Dynamic Interfaces**: Context-sensitive displays
- **Screen Manager**: View switching
- **Alarm Management**: Fault message containers
- **User Guidance**: Step-by-step instructions

⚖️ Comparison with similar components

Feature	Q_ObjHideShow	VtContainerControl	VtVisibilityManager
ISO Standard	✓	✖	✖
Container Focus	✓	✓	✖
Cascading	✓	✖	✓
Feedback	✓	✖	✓

🛠️ Related Exercises

- Exercise_014

Conclusion

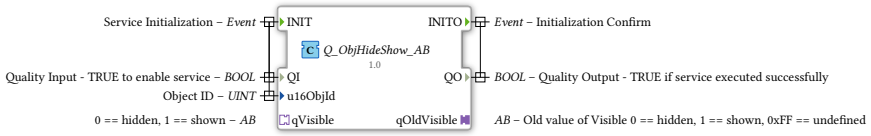
The Q_ObjHideShow block offers centralized visibility control:

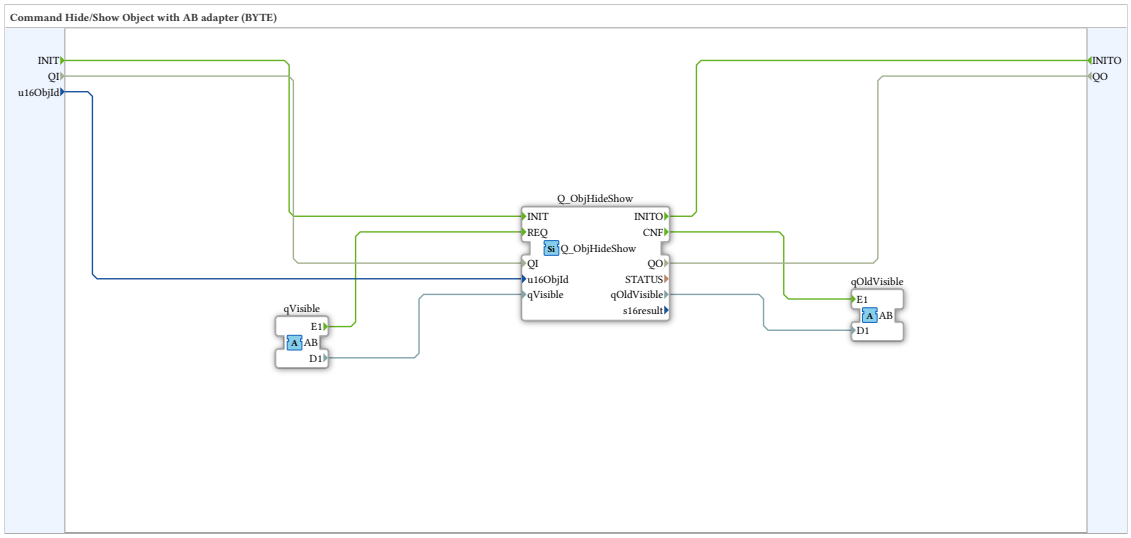
- **Efficient**: Instant GUI update
- **Comprehensive**: Affects all container elements
- **Reliable**: ISO-compliant implementation

Essential for:

- Adaptive user interfaces
- Complex form hierarchies
- State-driven visualizations

Q_ObjHideShow_AB





• – ○ • – ○ • – ○ •

Introduction

The function block **Q_ObjHideShow_AB** serves as a wrapper for the function block `Q_ObjHideShow` and enables hiding or showing an object via an **AB adapter** (unidirectional, BYTE). It encapsulates the direct control of the internal function block and provides an adapter-based interface that promotes loose coupling in the application.

Interface Structure

Event Inputs

Event	with variables	Comment
`INIT`	`QI`, `u16ObjId`	Service Initialization; Object ID is passed

Event Outputs

Event	Comment
`INITO`	Initialization confirmed

Data Inputs

Name	Type	Comment
`QI`	BOOL	Quality Input - TRUE to enable service
`u16ObjId`	UINT	Object ID

Data Outputs

Apart from `QO`, there are no explicit data outputs at the FB level. The remaining output data is provided via the adapter `qOldVisible`.

Name	Type	Comment
`QO`	`BOOL`	Quality Output - TRUE if service executed successfully

Adapters

Direction	Name	Type	Comment

Socket (Input)	qVisible	adapter::types::unidirectional::AB	Target Visibility: 0 = hidden, 1 = displayed
Plug (Output)	qOldVisible	adapter::types::unidirectional::AB	Previous Visibility Value: 0 = hidden, 1 = displayed, 0xFF = undefined

Quality signals QI/QO: The data input QI (BOOL, initial value TRUE) is associated with the event INIT and is passed through to QI of the internal block Q_ObjHideShow. The data output QO (BOOL) is taken over from Q_ObjHideShow.QO and is associated with the event INITO.

Valid Object IDs

The validity rules of the internal base function block Q_ObjHideShow apply:
u16ObjId — valid object types (Annex F.2): Container (3000–3999).
ID_NULL (65535) is not a valid command target but deactivates the FB when used with INIT.

Functionality

1. **Initialization**

An event at input INIT initializes the internal function block Q_ObjHideShow. The passed u16ObjId identifies the object to be controlled. After successful startup, the event INITO is output.

1. **Visibility Control**

As soon as an event (E1) arrives at the socket adapter qVisible, the corresponding data value (D1) is read and forwarded to the internal function block (FB). The internal FB then sets the object's visibility accordingly (0 = hidden, 1 = displayed).

1. **Feedback**

After the action is executed, the internal FB sends the event CNF to the plug adapter qOldVisible. The previous visibility state is provided as a data value (D1) on qOldVisible. This allows the calling component to query the previous state.

Technical Features

- **Wrapper Concept**: The function block (FB) encapsulates the direct use of Q_ObjHideShow and replaces the loose event/data connection with standardized AB adapters, enabling a modular and reusable interface.
- **AB Adapter (Unidirectional, BYTE)**: Communication takes place via a unidirectional adapter that transmits only one event and one associated byte at a time. This simplifies integration into adapter-based architectures.
- **State Maintenance**: The internal FB stores the last visibility state; with each new command, the previous state is reported back to the output adapter.
- **Undefined Value (0xFF)**: The output qOldVisible can return the value 0xFF if the previous state is unknown (e.g., after a restart without prior initialization).

State Overview

The FB itself does not have its own state machine. Its behavior is entirely determined by the embedded function block Q_ObjHideShow. The following generally applies:

- **After INIT**: The function block is ready for operation and waits for events from the socket adapter qVisible.
- **On every incoming socket event**: Visibility is set, and the result is reported back via the plug adapter.

Application Scenarios

- **Visualization Control**: Showing and hiding graphical objects (e.g., in HMI systems) based on binary or numerical control values.
- **State Feedback**: Monitoring of visibility changes; the previous visibility state can be used for logic decisions or logging.
- **Adapter-Based Automation**: Integration into a system that relies on adapter interfaces to achieve interchangeability and decoupling.

Comparison with Similar Function Blocks

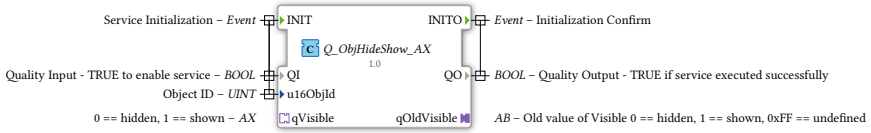
Function Block	Properties
`Q_ObjHideShow` (direct)	Event/data interface without adapter; direct connection required
`Q_ObjHideShow_AB` (this function block)	Uses AB adapters for loose coupling; easier integration into adapter-based networks
Other AB-based wrappers	Similar concept for other commands (e.g., `Q_ObjMove_AB`), but specifically designed for visibility

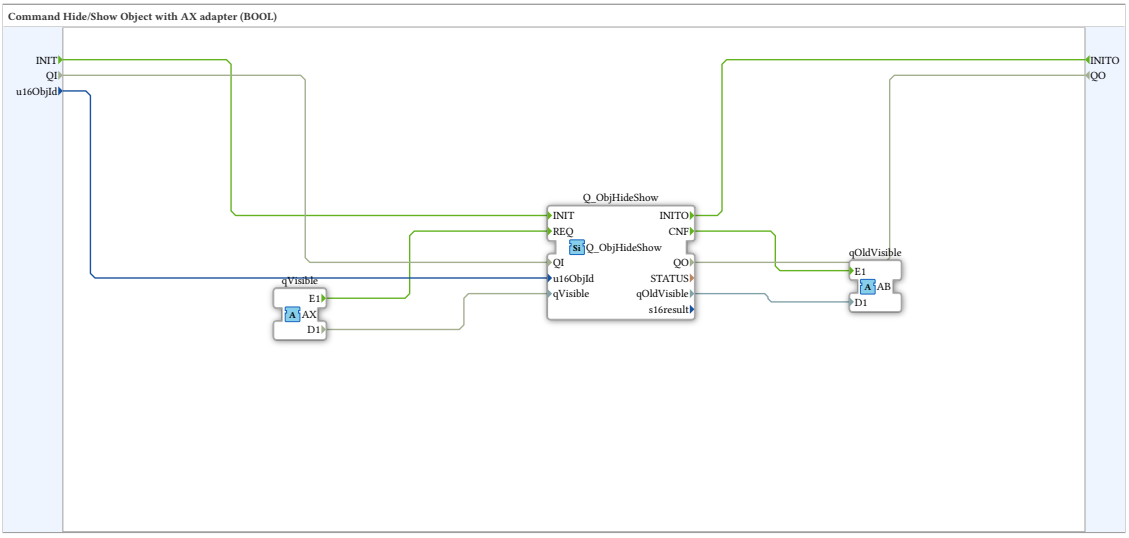
The main advantage of the wrapper lies in its adaptation to the AB adapter interface without changing the actual functionality.

Conclusion

The function block `Q_ObjHideShow_AB` represents an adapter-based, reusable component for controlling the visibility of objects. By encapsulating the internal function block and using unidirectional AB adapters, a clean separation of control logic and execution is achieved. It is particularly suitable for modular automation solutions that rely on loose coupling and standardized interfaces.

Q_ObjHideShow_AX





• – ○ • – ○ • – ○ •

Introduction

The function block **Q_ObjHideShow_AX** serves as an adapter wrapper for the existing function block `Q_ObjHideShow`. It allows the consistent hiding and showing of an object (Hide/Show) via a unidirectional AX adapter. The visibility value is passed as a Boolean state (`0` = hidden, `1` = shown).

Interface Structure

Event Inputs

Name	Type	Comment	With Variable
`INIT`	EInit	Service Initialization	`QI`, `u16ObjId`

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirmation

Data Inputs

Name	Type	Comment
QI	BOOL	Quality Input - TRUE to enable service
u16ObjId	UINT	Object Identification

Data Outputs

Apart from QO, this function block does not have its own data outputs. The old visibility value is output via the plug adapter qOldVisible.

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully

Adapter

Direction	Name	Type	Comment
Socket	qVisible	unidirectional::AX	Input of the new visibility value (0 = hidden, 1 = shown)
Plug	qOldVisible	unidirectional::AB	Output of the previous visibility value (0/1/0xFF)

Quality signals QI/QO: The data input QI (BOOL, initial value TRUE) is associated with the event INIT and is passed through to QI of the internal block Q_ObjHideShow. The data output QO (BOOL) is taken over from Q_ObjHideShow.QO and is associated with the event INITO.

Valid Object IDs

The validity rules of the internal base function block Q_ObjHideShow apply:
u16ObjId — valid object types (Annex F.2): Container (3000–3999).
ID_NULL (65535) is not a valid command target but deactivates the FB when used with INIT.

Functionality

1. The function block is initialized via the event INIT, and the object ID (u16ObjId) is passed to the internal function block Q_ObjHideShow.
2. As soon as an event arrives at socket qVisible (associated with the internal function block REQ), the new visibility value (qVisible.D1) is forwarded to the internal function block.
3. The internal function block performs the switchover and outputs the old value via its event CNF. This is reported externally via the plug qOldVisible (event E1 and data D1).
4. The output INITO signals the successful completion of the initialization.

Technical Features

- **Adapter Encapsulation:** The function block uses the standardized adapter types AX (for input) and AB (for output), allowing seamless integration into adapter-based ISOBUS networks.
- **Reuse:** The actual logic is implemented by the proven FB Q_ObjHideShow – the wrapper simply adds the adapter interface.
- **Boolean Control:** The visibility value is transmitted as a simple Boolean value (via the AX adapter), which simplifies its use in typical on/off scenarios.

State Overview

The function block does not have an explicitly visible state machine. It undergoes an initialization step (INIT → INITO) and then reacts cyclically to incoming events at socket qVisible (REQ → CNF). However, the internal function block Q_ObjHideShow can maintain an implicit state (e.g., the currently displayed visibility).

Application Scenarios

- Showing/hiding controls on an ISOBUS terminal.
- Controlling the display of symbols or text fields depending on user input or system states.
- A unified interface for all objects that are to be connected via adapters.

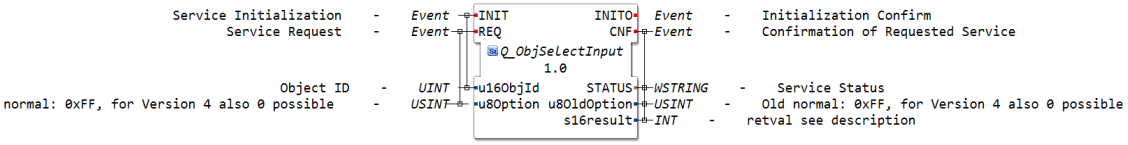
Comparison with similar function blocks

The direct function block `Q_ObjHideShow` requires separate event and data connections for control. The wrapper function block `Q_ObjHideShow_AX` encapsulates these connections in two standardized adapters (AX/AB), thus simplifying integration into existing adapter-based architectures. Furthermore, the old visibility value is automatically provided via the plug adapter.

Conclusion

The **Q_ObjHideShow_AX** is a practical wrapper that transfers the proven functionality of `Q_ObjHideShow` into an adapter-based interface. It enables clean, modular integration of Hide/Show commands and facilitates reuse in complex ISOBUS control systems.

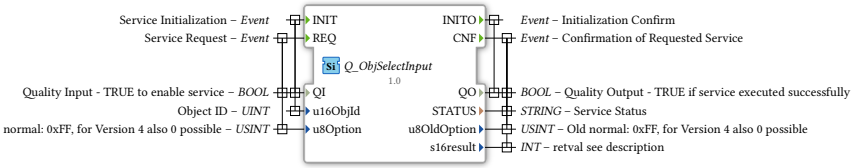
Q_ObjSelectInput



• — ○ • — ○ • — ○ •

Introduction

The **Q_ObjSelectInput** is a standards-compliant function block for the programmatic selection of input fields in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.6) specification for VT input elements.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object ID)
- `REQ`: Selection Request (with option parameter)

Event Outputs

- `INITO`: Initialization Confirmation
- `CNF`: Selection Confirmation

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Input Object ID (16-bit)
- `u8Option` (USINT): Selection Option (0x00 or 0xFF)

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `u8OldOption` (USINT): Previous selection option
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` — valid object types (Annex F.6): Key (5000–5999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999).

ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`. Any ID outside these ranges is invalid for commanding.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with target object ID
- `INITO` confirms operational readiness

1. **Field selection**:

- `REQ` with selection option
- Sets focus on the input field
- `CNF` returns operational status and previous option

1. **Options**:

- `0xFF`: Standard selection (VT version 3+)
- `0x00`: Extended selection (VT version 4+ only)

Technical Special Features

✓ **ISO 11783-6 compliant** (F.6) ✓ **VT version dependent** (Option 0x00 only available from VT4 onwards) ✓ **Instant focus change** ✓ **Feedback of previous state**

Option Reference

u8Option	Description	VT Version
0xFF	Standard selection	3+
0x00	Extended selection	4+

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful execution
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	VT Not Ready
−21	VT_E_NO_INSTANCE	No VT Client Available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT Instance
−130	VT_E_NOT_ALIVE	VT Not Active

Application Scenarios

- **Form Navigation**: Automatic Field Selection
- **Input Assistants**: Guided Operation
- **Error Management**: Direct Jump to Error Fields
- **Touch Optimization**: Programmatic Focus Control

⚖️ Comparison with Similar Function Blocks

Feature	Q_ObjSelectInput	VtFocusControl	VtInputSelector

ISO Standard	✓	✖	✖
VT Version 4+	✓	✖	✓
Option Control	✓	✖	✖
Feedback	✓	✖	✓

Conclusion

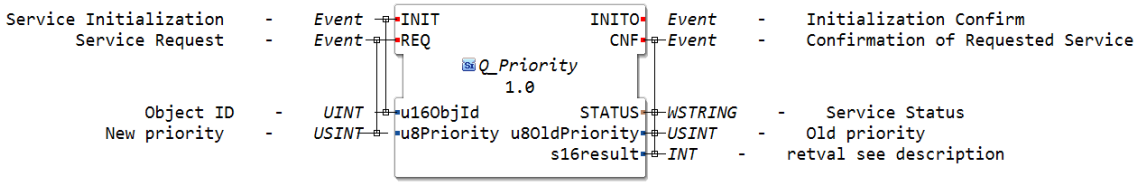
The Q_ObjSelectInput block offers precise input field control:

- **Efficient**: Direct focus change
- **Flexible**: Version-specific options
- **Reliable**: ISO-compliant implementation

Essential for:

- Professional form applications
- Guided user processes
- Touchscreen-optimized interfaces

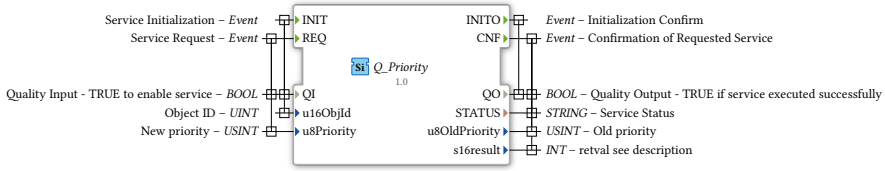
Q_Priority



• — ○ • — ○ • — ○ •

Introduction

The **Q_Priority** is a standards-compliant function block for priority control of alarm masks in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.40) specification for VT alarm management.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with Alarm Mask ID)
- `REQ`: Priority Change Request

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Change Acknowledgement

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Alarm Mask Object ID (16-bit)
- `u8Priority` (USINT): New Priority (0-255)

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message

- `u8OldPriority` (USINT): Previous priority
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` addresses the **Alarm Mask** whose priority is changed (F.40):
AlarmMask: 2000–2999
ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with alarm mask ID
- `INITO` confirms operational readiness

1. **Priority change**:

- `REQ` with new priority value
- Updates the alarm display order
- `CNF` provides result status and previous value

1. **Priority logic**:

- Lower numbers = Higher priority
- Affects alarm display order

Technical features

✓ **ISO 11783-6 compliant** (F.40)
✓ **Real-time Prioritization** (Immediate Effect) ✓ **Traceability** (Previous Priority) ✓ **Wide Value Range** (0-255)

Priority Reference

Priority	Typical Use
0-31	Critical System Alarms
32-63	Important Process Alarms
64-127	Standard Warnings
128-255	Informational Messages

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful Change
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	VT Not Ready
−21	VT_E_NO_INSTANCE	No VT Client Available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT Instance
−130	VT_E_NOT_ALIVE	VT Not Active

Application Scenarios

- **Alarm Management**: Dynamic Priority Adjustment
- **Process Control**: Importance Classification
- **Diagnostic Systems**: Escalation Management

- **Maintenance Modes**: Temporary Priority Changes

⚖️ Comparison with Similar Components

Feature	Q_Priority	VtAlarmPriority	VtEventManager
ISO Standard	✓	✖	✖
Real-time effect	✓	✓	✖
Value range	0-255	0-127	0-31
Feedback	✓	✖	✓

Conclusion

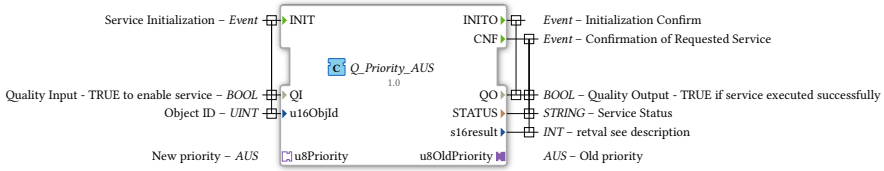
The Q_Priority block offers essential alarm control:

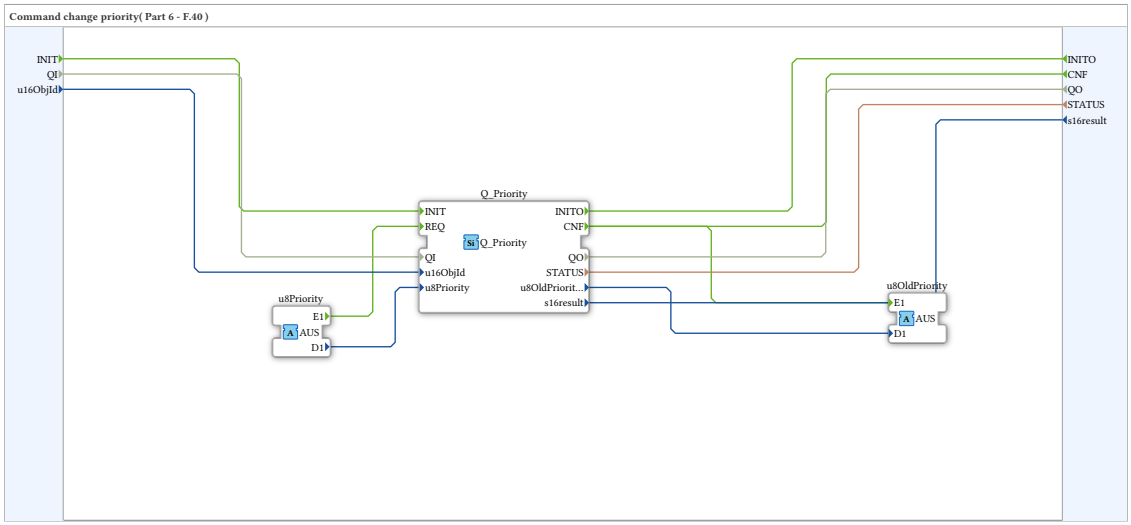
- **Dynamic**: Immediate priority adjustment
- **Transparent**: Feedback of the previous state
- **Robust**: Full ISO compliance

Essential for:

- Professional alarm management systems
- Process-critical applications
- Flexible diagnostic solutions

Q_Priority_AUS





• — ○ • — ○ • — ○ •

Introduction

The Q_Priority_AUS function block is used to change the priority of an alarm mask according to ISO 11783-6 (Part 6, F.40). An initialization process identifies the alarm mask, after which a new priority can be set. Confirmation of the service and the old priority value are output.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Service initialization; executed with the data input `u16ObjId`.

Event Outputs

Event	Type	Description
INITO	EInit	Confirmation of successful initialization.
CNF	Event	Confirmation of the requested service; output with `STATUS` and `s16result`.

Data Inputs

Name	Type	Description	Initial Value
QI	BOOL	Quality Input - TRUE to enable service	TRUE
u16ObjId	UINT	Object ID of the alarm mask	ID_NULL

Data Outputs

Name	Type	Description
QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Service status (e.g., error or success message)
s16result	INT	Return value as described (see Functionality)

Adapter

Name	Type	Direction	Description
u8Priority	OFF (unidirectional)	Socket	New priority (sent via the adapter)
u8OldPriority	OFF (unidirectional)	Plug	Old priority (received via the adapter)

Quality signals QI/QO: The data input `QI` (BOOL, initial value `TRUE`) is associated with the event `INIT` and is passed through to `QI` of the internal block `Q_Priority`. The data output `QO` (BOOL) is taken over from `Q_Priority.QO` and is associated with the events `INITO` and `CNF`.

Valid Object IDs

The function block wraps **Q_Priority** (F.40). `u16ObjId` addresses the **Alarm Mask** whose priority is changed:

AlarmMask: 2000–2999

ID_NULL (65535) is not a valid command target but deactivates the FB when used with `INIT`.

Functionality

After triggering `INIT` with the valid object ID of the alarm mask, the function block initializes the internal communication. As soon as an event arrives via the `u8Priority` adapter, the service for changing the priority is executed. The module sends the new priority to the virtual terminal and, upon completion, outputs confirmation via `CNF`, including the status and result value. Simultaneously, the previous priority (if available) is returned via the adapter `u8OldPriority`.

The module sends the new priority to the virtual terminal and, upon completion, outputs confirmation via `CNF`. **Return Values (s16result):**

- **VT_E_NO_ERR (0)** – Success
- **VT_E_OVERFLOW (-6)** – Buffer Overflow
- **VT_E_NOACT (-8)** – Command Not Possible in Current State

- **VT_E_NO_INSTANCE (-21)** – No VT Client Available
- **VT_E_ISO_INSTANCE_INVALID (-129)** – Connection Instance Invalid
- **VT_E_NOT_ALIVE (-130)** – Instance Valid, but VT Inactive

Technical Features

- The function block is based on the ISO 11783-6 protocol and uses unidirectional adapters for data transmission.
- Initialization expects a valid object ID; an invalid value (ID_NULL) results in no service.
- The error codes are standardized and enable easy diagnostics at the VT level.
- The internal logic delegates the actual priority change to the embedded function block `Q_Priority`.

State Overview

The function block does not have an explicitly displayed state machine. Sequence:

1. **INIT** ⟶ Initialization waits for a valid object ID.
2. After successful initialization ⟶ **INITO** is active.
3. The request (REQ) is initiated via the adapter `u8Priority`.
4. After processing ⟶ **CNF** with result.

Application Scenarios

- Changing the priority of an alarm mask in a virtual terminal of agricultural machinery (e.g., an ISOBUS-compliant control unit).
- Integration into a controller that needs to dynamically set different alarm levels (e.g., from “low” to “high” in the event of a critical incident).

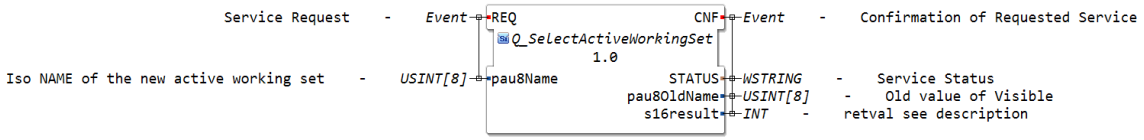
Comparison with Similar Function Blocks

- **Q_Priority** (without OFF): Offers the same core functionality but requires separate data lines for priority and age value. This function block encapsulates input and output via adapters, thus facilitating modular use in hierarchical networks.
- **Q_AlarmMask_Select**: Selects an alarm mask but does not change the priority. Q_Priority_OFF adds this functionality.

Conclusion

The Q_Priority_OFF function block implements a standardized, robust interface for changing the priority of alarm masks according to ISO 11783-6. Due to its adapter-based data transmission and the clear separation of initialization and service execution, it is particularly suitable for modular agricultural control systems.

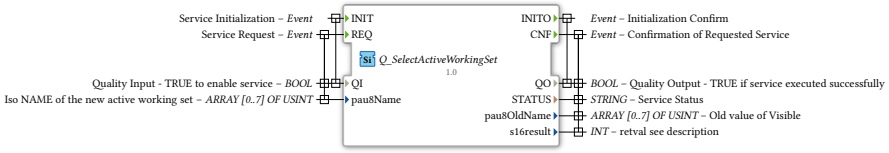
Q_SelectActiveWorkingSet



• — ○ • — ○ • — ○ •

Introduction

The **Q_SelectActiveWorkingSet** is a standards-compliant function block for controlling active workspaces in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.64) specification for VT systems from version 6 onwards.



Interface Structure

Event Inputs

- `INIT`: Initialization Request
- `REQ`: Workspace Change Request

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Change Acknowledgement

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `pau8Name` (USINT[8]): ISO NAME of the new active workspace

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `pau8OldName` (USINT[8]): ISO NAME of the previous workspace
- `s16result` (INT): ISO-compliant result code

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` without parameters
- `INITO` confirms operational readiness

1. **Workspace change**:

- `REQ` with 8-byte ISO NAME of the target workspace
- Performs context change
- `CNF` returns operational status and previous workspace

1. **ISO NAME format**:

- 8-byte array according to ISO 11783-5 specification

Technical features

✓ **ISO 11783-6 Compliant** (F.64) ✓ **Exclusive to VT Version 6+** ✓ **Full Context Switch** (All associated resources) ✓ **Traceability** (Previous workspace)

ISO NAME Structure

Byte	Content
0-1	Vendor ID
2-3	Device ID
4-5	Device Instance
6-7	Workspace ID

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful Switch
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	VT not ready

−21	VT_E_NO_INSTANCE	No VT client available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Multi-Client Systems**: Device switching
- **User Profiles**: Individual work environments
- **Process Phases**: Operating mode switching
- **Diagnostic Management**: Service/Normal mode

⚖️ **Comparison with similar building blocks**

Feature	Q_SelectActiveWorkingSet	VtContextSwitch	VtWorkspaceManager
ISO Standard	✓	✖	✖
VT Version	6+	All	All
ISO NAME	✓	✖	✖
Feedback	✓	✖	✓

Conclusion

The Q_SelectActiveWorkingSet building block offers centralized workspace control:

- **Secure**: ISO-compliant identification
- **Efficient**: Complete context switching
- **Transparent**: Previous state traceable

Essential for:

- Complex multi-client systems
- Flexible user environments
- Process-oriented VT applications

Q_SelectColourMap



• — ○ • — ○ • — ○ •

Introduction

The **Q_SelectColourMap** is a standards-compliant function block for managing color palettes in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.60) specification for VT systems from version 4 onwards.



Interface Structure

Event Inputs

- `INIT`: Initialization Request
- `REQ`: Color Palette Change Request

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Change Acknowledgement

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjIdColourMap` (UINT): Color Palette Object ID or 0xFFFF for the default palette

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `u16OldObjIdColourMap` (UINT): Previous color palette ID
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjIdColourMap` addresses (bytes 2,3) per Annex F.60:

- **ColourMap**: 39000 – 39999 (VT version 4 and later)
- **ColourPalette**: 45000 – 45999 (VT version 6 and later, Annex F.60)
- **FFFF16**: restores the default colour table (A.3)

Implementation note (VTClientHelper): `iso_is_colour_map_id` accepts only the ColourMap range. In addition, ID_NULL (65535) is swallowed by `cmd_select_colour_map_or_palette`, which answers with VT_E_NO_ERR (0) without sending any command to the VT — the documented FFFF value for restoring the default palette therefore currently does not work (known deviation).

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` without parameters
- `INITO` confirms operational readiness

1. **Pallet Change**:

- `REQ` with target color palette ID
- Global color management change
- `CNF` provides operational status and previous palette

1. **Special Values**:

- `0xFFFF`: Sets the default ISO color table

Technical Features

✓ **ISO 11783-6 compliant** (F.60) ✓ **Global effect**: Affects all workspaces ✓ **Long execution time**: For complex palettes ✓ **Traceability**: Previous palette can be saved

Color Palette Reference

Value	Meaning
0x0001-FFFE	User-defined palettes
0xFFFF	ISO Standard Palette (A.3)

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful switch
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	VT Not Ready
−21	VT_E_NO_INSTANCE	No VT Client Available

−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT Instance
−130	VT_E_NOT_ALIVE	VT Not Active

Application Scenarios

- **Design Change**: Dynamic Color Schemes
- **Night Mode**: Eye-Friendly Palettes
- **Brand Specifics**: Corporate Design
- **Accessibility**: High-Contrast Modes

⚖️ Comparison with Similar Components

Feature	Q_SelectColourMap	VtColorManager	VtPaletteControl
ISO Standard	✓	✖	✖
Global Effect	✓	✖	✓
Standard Palette	✓	✖	✖
Feedback	✓	✖	✓

Conclusion

The Q_SelectColourMap module offers professional color management:

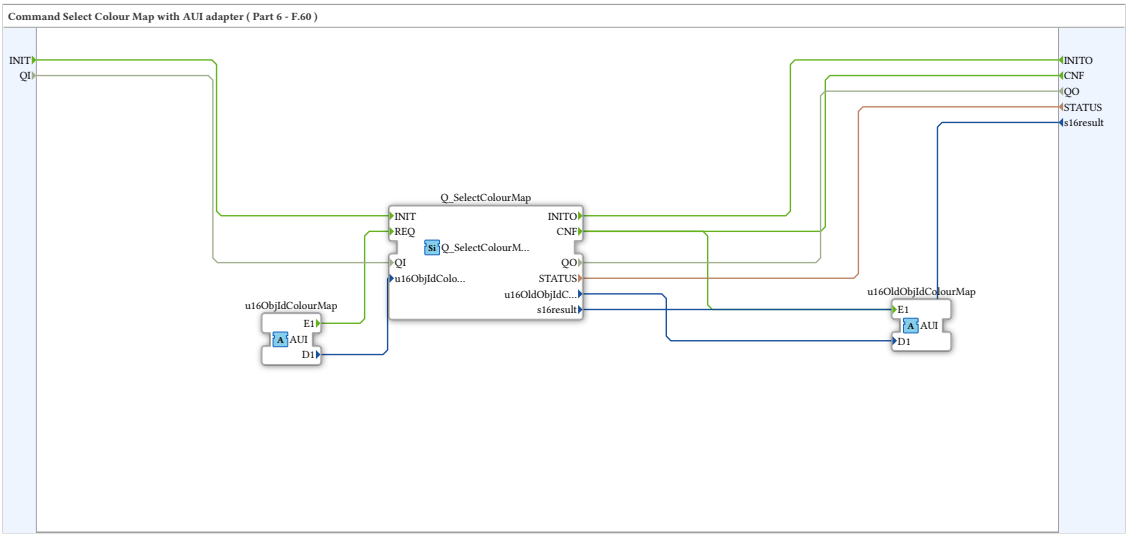
- **Comprehensive**: Global color control
- **Flexible**: User and standard palettes
- **Reliable**: ISO-compliant implementation

Essential for:

- Design-oriented applications
- Adaptive visualization systems
- Brand-specific interfaces

Q_SelectColourMap_AUI





• – ○ • – ○ • – ○ •

Introduction

The **Q_SelectColourMap_AUI** function block is an AUI adapter wrapper for **Q_SelectColourMap** (ISO 11783-6, Part 6 – F.60). It enables selecting the active Colour Map / palette on an ISOBUS Virtual Terminal using AUI adapter connections.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Service Initialization

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirm
CNF	Event	Confirmation of Requested Service

Data Inputs

Apart from QI, there are no direct data inputs — Colour Map IDs are handled via AUI adapters.

Name	Type	Initial value	Comment
QI	BOOL	TRUE	Quality Input - TRUE to enable service

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Service Status
s16result	INT	Command return code

Adapters

Type	Name	Direction	Comment
adapter::types::unidirectional::AUI16ObjIdColourMap	u16ObjIdColourMap	Socket (Input)	New Colour Map Object ID (39000–39999 for ColourMap, 45000–45999 for ColourPalette, ID_NULL / 0xFFFF for default table)
adapter::types::unidirectional::AUI16OldObjIdColourMap	u16OldObjIdColourMap	Plug (Output)	Previous Colour Map Object ID

Quality signals QI/QO: The data input QI (BOOL, initial value TRUE) is associated with the event INIT and is passed through to QI of the internal block Q_SelectColourMap. The data output QO (BOOL) is taken over from Q_SelectColourMap.QO and is associated with the events INITO and CNF.

Functionality

Encapsulates inner FB Q_SelectColourMap (*isobus::UT::Q::Q_SelectColourMap*). An event on u16ObjIdColourMap.E1 triggers REQ on the inner FB with the ID from u16ObjIdColourMap.D1. Upon completion, the previous ID is output via u16OldObjIdColourMap.D1 and CNF.

Technical Features

- **AUI Adapter Integration:** Adapter-driven selection of Colour Maps via u16ObjIdColourMap socket.

- **ID Ranges per ISO 11783-6 F.60:**
  - `ColourMap`: 39000–39999 (VT version 4 and later)
  - `ColourPalette`: 45000–45999 (VT version 6 and later)
  - `ID_NULL` (`0xFFFF` / 65535): Restores the default colour table.
- **Implementation Limitations (VTClientHelper):**
  - `VTClientHelper::iso_is_colour_map_id` currently only accepts the range 39000–39999 (`ColourPalette` IDs 45000–45999 are rejected).
  - `cmd_select_colour_map_or_palette` consumes `ID_NULL` (`0xFFFF`), returns `VT_E_NO_ERR` (0), and sends no VT command. Restoring the default colour table is therefore currently not supported.

State Overview

Delegates state processing to inner `Q_SelectColourMap`.

Application Scenarios

- Dynamic colour scheme switching (e.g. day/night mode) on ISOBUS VT terminals via AUI adapters.

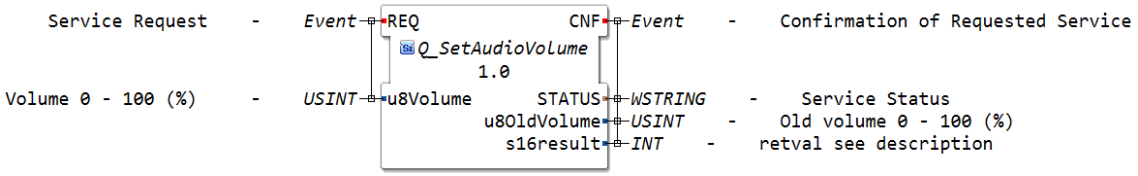
Comparison with Similar Blocks

Wraps `Q_SelectColourMap` with AUI adapter socket and plug for seamless integration into adapter networks.

Conclusion

Q_SelectColourMap_AUI is the adapter wrapper for Colour Map selection in ISOBUS VT applications.

Q_SetAudioVolume

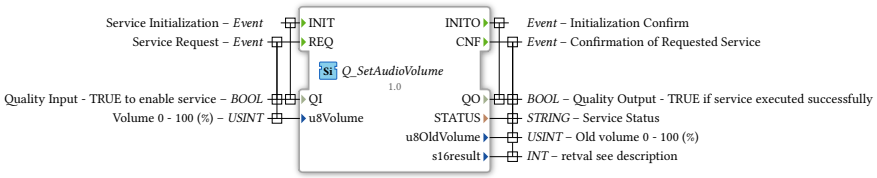


• – ◦ • – ◦ • – ◦ •

Introduction

The **Q_SetAudioVolume** is a standards-compliant function block for volume control in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.12) specification for VT audio control.

The F.12 command addresses the VT's **audio device** directly — it does **not reference any object**, so this block has no Object ID input. The message has a fixed length of 8 bytes; byte 2 carries the volume percentage (0% – 100%).



Interface Structure

Event Inputs

- `INIT`: Initialization Request
- `REQ`: Volume Change Request

Event Outputs

- `INITO`: Initialization Acknowledgement
- `CNF`: Change Acknowledgement

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service

- `u8Volume` (USINT): Volume Value (0-100%)

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operating Status Message
- `u8OldVolume` (USINT): Previous Volume Setting
- `s16result` (INT): ISO-compliant result code

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` without parameters
- `INITO` confirms operational readiness

1. **Volume Control**:

- `REQ` with desired volume level
- Changes the global VT volume
- `CNF` returns operating status and previous value

1. **Value Range**:

- 0%: Muted
- 100%: Maximum volume

Technical Features

✓ **ISO 11783-6 compliant** (F.12) ✓ **Global audio control** ✓ **Percentage control** (0-100%) ✓ **Traceability** (Previous Setting)

Volume Reference

Value	Meaning
0	Mute
50	Medium Volume
100	Maximum Volume

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful Change
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	VT Not Ready
−21	VT_E_NO_INSTANCE	No VT Client Available
−128	VT_E_HANDLE_INVALID	Invalid Audio Configuration
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Alarm Management**: Adjustable warning signals
- **Voice Output**: Volume adjustment
- **Ease of Use**: User-defined audio settings
- **Night Mode**: Automatic volume reduction

⚖️ Comparison with similar components

Feature	Q_SetAudioVolume	VtAudioControl	VtSoundManager
ISO Standard	✓	✖	✖
Value Range	0-100%	0-255	0-10 steps
Global Effect	✓	✓	✖
Feedback	✓	✖	✓

Conclusion

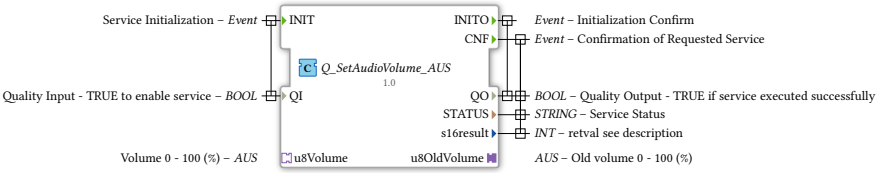
The Q_SetAudioVolume module offers precise audio control:

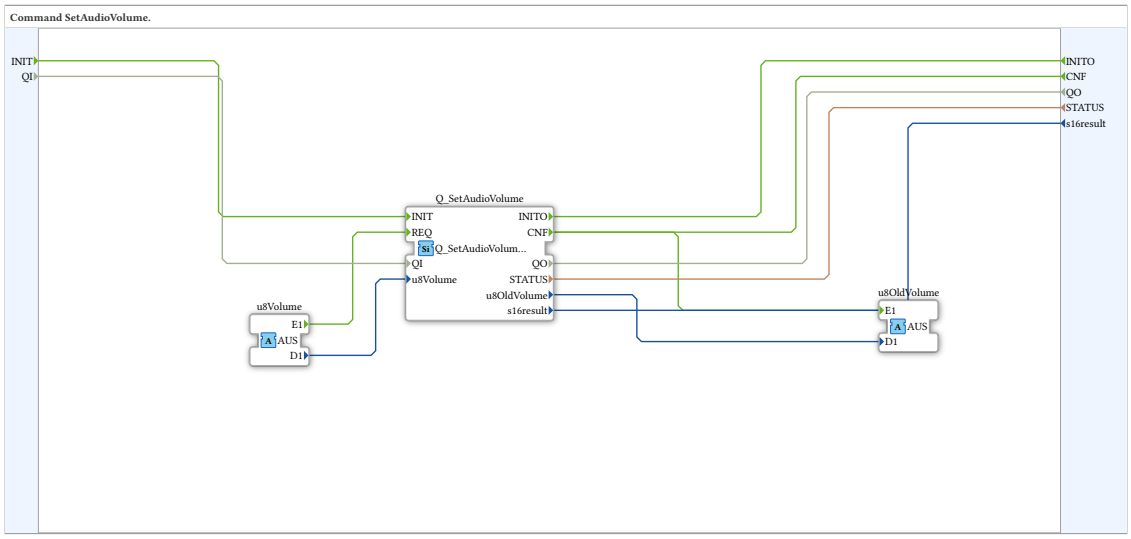
- **Intuitive**: Percentage-based control
- **Reliable**: ISO-compliant implementation
- **Flexible**: For all audio outputs

Essential for:

- Professional alarm systems
- Voice-controlled applications
- User-friendly interfaces

Q_SetAudioVolume_AUS





• – ○ • – ○ • – ○ •

Introduction

The Q_SetAudioVolume_AUS function block is a composite function block (FB) that executes the “SetAudioVolume” command according to ISO 11783-6 on a Virtual Terminal (VT). It encapsulates the base function block Q_SetAudioVolume and provides an adapter-based interface for setting the volume and reading back the previous value. The FB initializes itself via the INIT event input and acknowledges this with INITO. A new volume value can then be passed via the u8Volume adapter, whereupon the command is triggered and the result is output via CNF with STATUS and s16result.

Interface Structure

Event Inputs

- **INIT** (EInit): Service initialization. Triggers the initialization of the internal function block. Associated with `QI`.

Event Outputs

- **INITO** (EInit): Confirmation of successful initialization. Associated with `QO`.
- **CNF** (Event): Confirmation of the requested service. Outputted after setting the volume. Contains the output data STATUS and s16result. Associated with `QO`.

Data Inputs

Apart from `QI`, the FB has no direct data inputs. Volume input is via the u8Volume adapter.

Name	Type	Initial value	Comment
`QI`	`BOOL`	`TRUE`	Quality Input - TRUE to enable service

Data Outputs

- **QO** (BOOL): Quality Output - TRUE if service executed successfully.
- **STATUS** (STRING): Service status message.
- **s16result** (INT): Command return value. Possible values:
- 0 (VT_E_NO_ERR): Success.
- −6 (VT_E_OVERFLOW): Buffer overflow.
- −8 (VT_E_NOACT): Command not possible in the current state.
- −21 (VT_E_NO_INSTANCE): No VT client available.
- −129 (VT_E_ISO_INSTANCE_INVALID): Connection instance invalid.
- −128 (VT_E_HANDLE_INVALID): Handle invalid.
- −130 (VT_E_NOT_ALIVE): VT not active.

Adapters

- **u8Volume** (Socket): Unidirectional adapter of type OFF. Receives the desired volume as a value between 0 and 100%.
- **u8OldVolume** (Plug): Unidirectional adapter of type OFF. Outputs the volume level (0–100%) valid before the command.

Quality signals QI/QO: The data input `QI` (BOOL, initial value `TRUE`) is associated with the event `INIT` and is passed through to `QI` of the internal block `Q_SetAudioVolume`. The data output `QO` (BOOL) is taken over from `Q_SetAudioVolume.QO` and is associated with the events `INITO` and `CNF`.

Functionality

After startup, the FB is initialized via the INIT input. The internal block Q_SetAudioVolume is prepared, and INITO signals its readiness. As soon as a new volume value is received via the u8Volume adapter (event E1 of the adapter), the command to change the volume is sent to the VT. Upon completion of the process, CNF is triggered, and the output variables STATUS and s16result contain the result. Simultaneously, the previous volume value is output via the u8OldVolume adapter, allowing external logic to read the old value.

The possible error codes correspond to the ISO11783-6 definitions and cover typical VT error situations.

Technical Features

- The function block is implemented as a composite block that uses the basic function block Q_SetAudioVolume from the isobus library.
- The interface uses only adapters for data transmission, resulting in loose coupling and flexible wiring.
- The adapters are of type “unidirectional::OFF” and enable event-driven data transmission.
- The function block (FB) is designed for use in ISOBUS systems according to ISO 11783-6.

State Overview

Since this is a composite block with an internal FB, there is no explicit top-level state machine. The FB goes through the following phases:

1. Initialization (INIT received ⟶ INITO sent)
2. Ready (waits for volume change via u8Volume adapter)
3. Processing (internal command is executed)
4. Result output (CNF with STATUS/s16result)

A detailed state machine is implemented in the internal block Q_SetAudioVolume.

Application Scenarios

- Agricultural machine control: Adjusting the volume of voice outputs or warning tones from the virtual terminal.
- Integration into ISOBUS-compliant applications that change audio settings via the operator terminal.
- Combination with other VT command blocks such as Q_SelectAudioOutput, Q_Beep, etc.

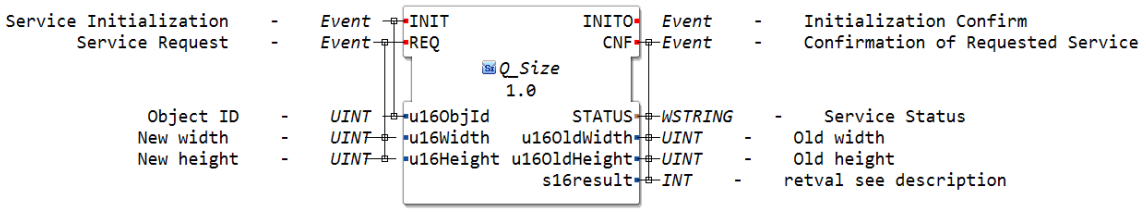
Comparison with similar blocks

The Q_SetAudioVolume_AUS block differs from a direct Q_SetAudioVolume block in its use of adapters for data transmission. While Q_SetAudioVolume may have direct data inputs (e.g., u8Volume as VarDeclaration), the adapter variant allows for decoupled, event-driven communication. Similar adapter blocks exist for other VT commands, resulting in a consistent design pattern.

Conclusion

Q_SetAudioVolume_AUS is a specialized function block for volume control of an ISOBUS VT. The adapter-based interface ensures a clear separation between the control and data layers. The block provides complete feedback on the success of the command and is suitable for reliable use in agricultural control systems.

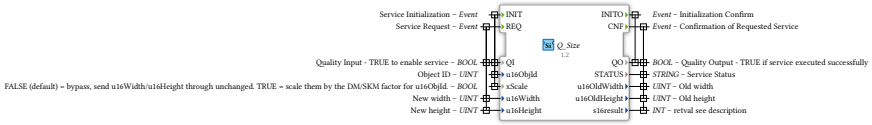
Q_Size



• — ○ • — ○ • — ○ •

Introduction

The **Q_Size** is a standards-compliant function block for resizing objects in Virtual Terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.18) specification for VT objects.



Interface Structure

Event Inputs

- `INIT`: Initialization request (with object ID and xScale)
- `REQ`: Resize request

Event Outputs

- `INITO`: Initialization acknowledgment
- `CNF`: Resize acknowledgment

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Object ID (16-bit)
- `xScale` (BOOL): If TRUE, `u16Width`/`u16Height` are scaled by the DM/SKM factor for `u16ObjId` before sending; if FALSE (default) they are passed through unchanged.

- `u16Width` (UINT): New width in pixels
- `u16Height` (UINT): New height in pixels

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `u16OldWidth` (UINT): Previous width
- `u16OldHeight` (UINT): Previous height
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` — valid object types (Annex F.18, objects with sizeable attribute): Container (3000–3999), Button (6000–6999), Input Boolean Field (7000–7999), Input String Field (8000–8999), Input Number Field (9000–9999), Input List Field (10000–10999), Output String Field (11000–11999), Output Number Field (12000–12999), Output Line (13000–13999), Output Rectangle (14000–14999), Output Ellipse (15000–15999), Output Polygon (16000–16999), Meter (17000–17999), Linear Bar Graph (18000–18999), Arched Bar Graph (19000–19999), Output List Object (37000–37999), Animation Object (44000–44999).

ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`. Any ID outside these ranges is invalid for commanding.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with target object ID
- `INITO` confirms operational readiness

1. **Resize**:

- `REQ` with new dimensions
- Dynamically adjusts the object geometry
- `CNF` provides operational status and previous dimensions

1. **Unit of measurement**:

- All values in screen pixels

Technical Features

✓ **ISO 11783-6 compliant** (F.18) ✓ **Dynamic resizing** (real-time update) ✓ **Comprehensive object support** ✓ **Traceability** (previous dimensions)

Size reference

Parameter	Value range	Description
Width	1-65535 pixels	Horizontal extent
Height	1-65535 pixels	Vertical extent

Return codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful change
−6	VT_E_OVERFLOW	Buffer Overflow
−8	VT_E_NOACT	VT Not Ready
−21	VT_E_NO_INSTANCE	No VT Client Available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT Instance

−130	VT_E_NOT_ALIVE	VT Not Active

Application Scenarios

- **Responsive Designs**: Adapting to Display Sizes
- **Zoom Functions**: Dynamic Scaling
- **Layout Manager**: Automatic Size Optimization
- **Animation Control**: Size Change Effects

⚖️ Comparison with Similar Building Blocks

Feature	Q_Size	VtResizeObject	VtGeometryManager
ISO Standard	✓	✖	✖
Real-time Update	✓	✓	✖
2D Control	✓	✓	✓
Feedback	✓	✖	✓

Conclusion

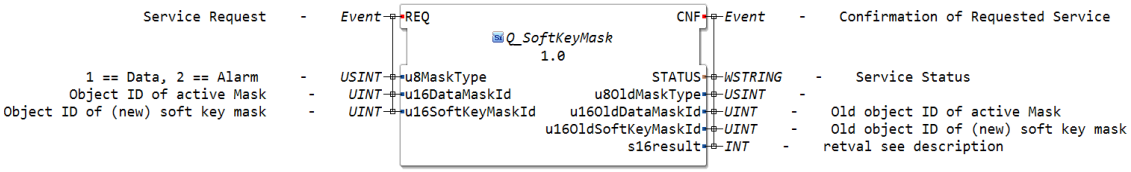
The Q_Size block offers precise size control:

- **Flexible**: Pixel-accurate adjustment
- **High-Performance**: Instant display updates
- **Reliable**: ISO-compliant implementation

Essential for:

- Adaptive User Interfaces
- Dynamic Visualizations
- Professional VT Applications

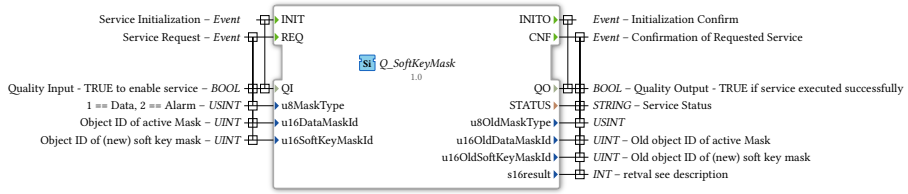
Q_SoftKeyMask



• — ○ • — ○ • — ○ •

Introduction

The **Q_SoftKeyMask** is a standards-compliant function block for managing softkey masks in virtual terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.36) specification for VT systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request
- `REQ`: Screen Change Request

Event Outputs

- `INITO`: Initialization Confirmation
- `CNF`: Change Confirmation

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u8MaskType` (USINT): Screen Type (1=Data, 2=Alarm)
- `u16DataMaskId` (UINT): Object ID of the Active Screen
- `u16SoftKeyMaskId` (UINT): Object ID of the New Softkey Screen

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational status message
- `u8OldMaskType` (USINT): Previous screen type
- `u16OldDataMaskId` (UINT): Previous data mask ID
- `u16OldSoftKeyMaskId` (UINT): Previous softkey screen ID
- `s16result` (INT): ISO-compliant result code

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` without parameters
- `INITO` confirms operational readiness

1. **Screen change**:

- `REQ` with screen type, data mask ID, and new softkey screen
- Linkes softkeys to the active screen
- `CNF` provides operating status and previous configuration

1. **Mask Types**:

- `1`: Data Mask Link
- `2`: Alarm Mask Link

Valid Object IDs

The function block addresses **two distinct Object IDs**, each with its own validity rules:

Valid values for u16DataMaskId (data or alarm mask)

The Change Soft Key Mask command (Annex F.36) associates a Soft Key Mask with a Data Mask or an Alarm Mask. `u16DataMaskId` is valid if it falls within one of the following ISO 11783-6 ID ranges:

Object type	ID range
DataMask	1000 – 1999
AlarmMask	2000 – 2999

Valid values for u16SoftKeyMaskId (soft key mask)

`u16SoftKeyMaskId` is valid if it falls within the ISO 11783-6 ID range of a Soft Key Mask:

Object type	ID range
SoftKeyMask	4000 – 4999

Note: ID_NULL (65535) is not a command target but can be used in INIT to deactivate the FB. Any data/alarm mask ID outside the ranges above, or any soft key mask ID that is not a SoftKeyMask, is invalid for commanding — independent of each other.

Technical Features

✓ **ISO 11783-6 compliant** (F.36) ✓ **Dual Mask Types** (Data/Alarms) ✓ **Dynamic Reassignment** ✓ **Traceability** (Previous Configuration)

Mask Type Reference

Value	Type	Description
1	Data	Normal Data Mask
2	Alarm	Alarm Mask Link

Return Codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Successful switch
−6	VT_E_OVERFLOW	Buffer overflow
−8	VT_E_NOACT	VT not ready
−21	VT_E_NO_INSTANCE	No VT client available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Context-Sensitive Softkeys**: Function Switching
- **Alarm Management**: Special Alarm Softkeys
- **Mask Hierarchies**: Dynamic Key Assignments
- **User Guidance**: Adaptive User Interfaces

⚖️ **Comparison with Similar Components**

Feature	Q_SoftKeyMask	VtKeyMapping	VtMaskLinker
ISO Standard	✓	✖	✖
Mask Types	2 (Data/Alarm)	1 (Universal)	1 (Data)
Feedback	✓	✖	✓
Real-Time Switching	✓	✓	✖

Conclusion

The Q_SoftKeyMask module offers flexible softkey control:

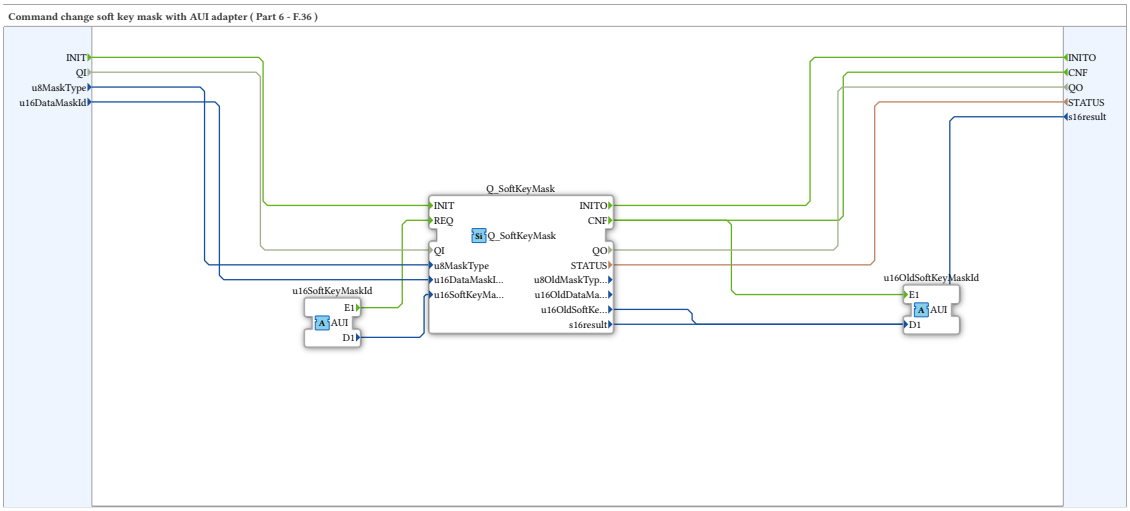
- **Context-sensitive**: Different mask types
- **Dynamic**: Instant reassignment
- **Reliable**: ISO-compliant implementation

Essential for:

- Professional user interface concepts
- Complex mask hierarchies
- Adaptive VT applications

Q_SoftKeyMask_AUI





• – ○ • – ○ • – ○ •

Introduction

The **Q_SoftKeyMask_AUI** function block is an AUI adapter wrapper for **Q_SoftKeyMask** (ISO 11783-6, Part 6 – F.36). It enables switching the active soft key mask on an ISOBUS Virtual Terminal using AUI adapter connections.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Service Initialization

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirm
CNF	Event	Confirmation of Requested Service

Data Inputs

Name	Type	Comment
QI	BOOL	Quality Input - TRUE to enable service
u8MaskType	USINT	Mask Type (1: DataMask, 2: AlarmMask)
u16DataMaskId	UINT	Object ID of target Data/AlarmMask

Data Outputs

Name	Type	Comment

QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Service Status
s16result	INT	Command return code

Adapters

Type	Name	Direction	Comment
adapter::types::unidirectional::AUI16SoftKeyMaskId	u16SoftKeyMaskId	Socket (Input)	New soft key mask Object ID
adapter::types::unidirectional::AUI16OldSoftKeyMaskId	u16OldSoftKeyMaskId	Plug (Output)	Previous soft key mask Object ID

Quality signals QI/QO: The data input `QI` (BOOL, initial value `TRUE`) is associated with the event `INIT` and is passed through to `QI` of the internal block `Q_SoftKeyMask`. The data output `QO` (BOOL) is taken over from `Q_SoftKeyMask.QO` and is associated with the events `INITO` and `CNF`.

Functionality

Encapsulates inner FB `Q_SoftKeyMask` (*isobus::UT::Q::Q_SoftKeyMask*). An event on `u16SoftKeyMaskId.E1` triggers `REQ` on the inner FB, which switches the soft key mask. Upon completion, the previous soft key mask ID is output via `u16OldSoftKeyMaskId.D1` and `u16OldSoftKeyMaskId.E1`.

Technical Features

- **Adapter-driven Softkey Control:** Flexible switching of soft key layouts via AUI adapters.
- **ISOBUS Standard Compliance:** Complies with ISO 11783-6 F.36.

State Overview

Delegates state processing to inner `Q_SoftKeyMask`.

Application Scenarios

- Dynamic soft key layout switching in ISOBUS VT terminals via AUI adapters.

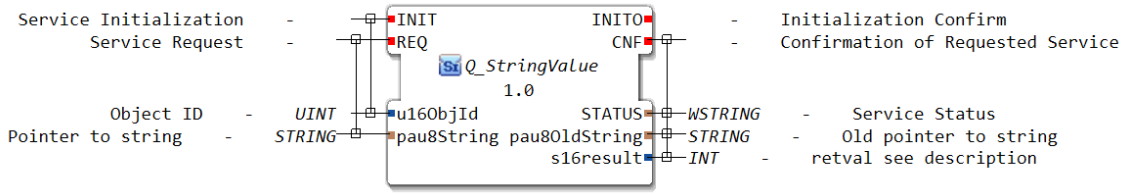
Comparison with Similar Blocks

Wraps `Q_SoftKeyMask` with an AUI adapter interface for integration into adapter networks.

Conclusion

Q_SoftKeyMask_AUI is the adapter wrapper for soft key mask switching in ISOBUS VT applications.

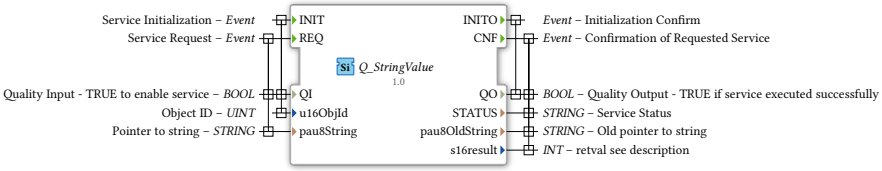
Q_StringValue



• – ○ • – ○ • – ○ •

Introduction

The **Q_StringValue** is a standards-compliant function block for dynamic text updates in Virtual Terminals, developed under the EPL-2.0 license. Version 1.0 implements the ISO 11783-6 (Part 6 - F.24) specification for VT text objects.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with object ID)
- `REQ`: Text Change Request

Event Outputs

- `INITO`: Initialization Confirmation
- `CNF`: Change Confirmation

Data Inputs

- `QI` (BOOL, default `TRUE`): Quality input – `TRUE` enables the service
- `u16ObjId` (UINT): Text Object ID (16-bit)
- `pau8String` (STRING): New Text Value (null-terminated)

Data Outputs

- `QO` (BOOL): Quality output – `TRUE` if the service was executed successfully
- `STATUS` (STRING): Operational Status Message
- `pau8OldString` (STRING): Previous text value
- `s16result` (INT): ISO-compliant result code

Valid Object IDs

`u16ObjId` — valid object types (Annex F.24, objects with string value attribute): Input String Field (8000–8999), Output String Field (11000–11999), String Variable (22000–22999), Input Attributes (26000–26999).

Data length / Transport Protocol: The F.24 command has a **variable data length** (command frame: Object ID, string byte count, string value). Strings that fit into a single CAN message are sent directly; longer strings are transmitted using the ISO 11783 **transport protocol**. Per Annex F.24 the transferred string may be shorter than the target object's length attribute (the VT pads it with space characters), but the number of transferred bytes shall not exceed the target object's length attribute.

ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`. Any ID outside these ranges is invalid for commanding.

Functionality

Quality handling (QI/QO): The input `QI` (BOOL, default `TRUE`) is associated with the events `INIT` and `REQ`; `TRUE` enables the service. The output `QO` (BOOL) is associated with `INITO` and `CNF` and indicates with `TRUE` that the service was executed successfully.

1. **Initialization**:

- `INIT` with target object ID
- `INITO` confirms operational readiness

1. **Text Update**:

- `REQ` with new text string
- Immediately updates the VT text object
- `CNF` returns operational status and previous text

1. **String Requirements**:

- Maximum `ISO_VTC_CMD_STR_MAX_LENGTH` including null termination
- Must be null-terminated

Technical Features

✓ **ISO 11783-6 compliant** (F.24)
  ✓ **Real-time text update** (FIFO-based)
  ✓ **Null termination required**
  ✓ **Traceability** (Previous text value)

String format

Parameters	Requirements
Maximum length	ISO_VTC_CMD_STR_MAX_LENGTH
Termination	'\0' at the end is mandatory
Character set	ISO-8859-1 (Latin-1) recommended

Return codes (s16result)

Code	Constant	Meaning
0	VT_E_NO_ERR	Update successful
−6	VT_E_OVERFLOW	String too long or buffer full
−8	VT_E_NOACT	VT not ready
−21	VT_E_NO_INSTANCE	No VT client available
−129	VT_E_ISO_INSTANCE_INVALID	Invalid VT instance
−130	VT_E_NOT_ALIVE	VT not active

Application Scenarios

- **Dynamic Displays**: Measured values, status messages
- **Language Switching**: Multilingual texts
- **Process Visualization**: Real-time data
- **User Communication**: Dialogs and prompts

⚖️ **Comparison with similar building blocks**

Feature	Q_StringValue	VtTextUpdate	VtStringManager
ISO Standard	✓	✖	✖
FIFO Update	✓	✖	✓
Length Check	✓	✓	✖
Feedback	✓	✖	✓

🛠️ **Related Exercises**

- Exercise_012c

Conclusion

The Q_StringValue block offers efficient text management:

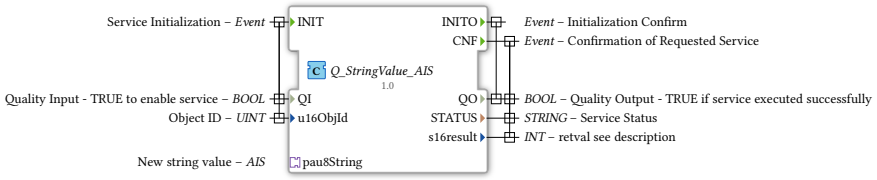
- **High-performance**: Direct FIFO update
- **Secure**: Integrated length check
- **Flexible**: For all text object types

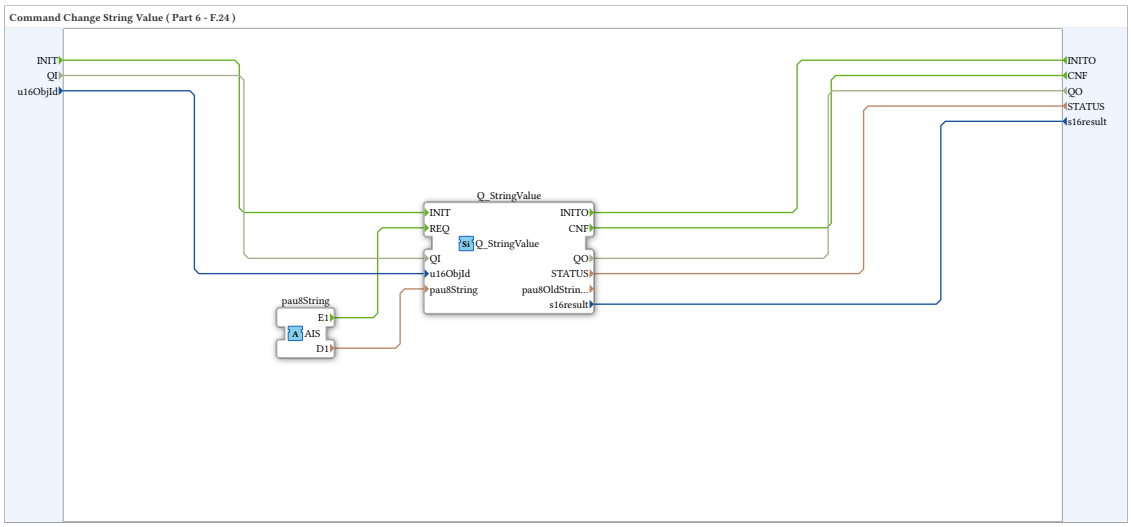
Essential for:

- Dynamic process visualization
- Multilingual applications

- Real-time data display

Q_StringValue_AIS





• — ○ • — ○ • — ○ •

Introduction

The function block **Q_StringValue_AIS** implements the “Command Change String Value” service according to ISO 11783-6 (Part 6, Section F.24). It is used to send a new string value to a connected device via an ISOBUS network. The block encapsulates all the logic for initialization, sending the command, and returning the result. The new string value is input via a unidirectional adapter (AIS), which provides the actual data.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Service initialization, sets the object ID

Event Outputs

Event	Type	Comment
INITO	EInit	Confirmation of successful initialization
CNF	Event	Confirmation of the requested service (result)

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	TRUE	Quality Input - TRUE to enable service
u16ObjId	UINT	ID_NULL	Object ID of the value to be changed

Data Outputs

Name	Type	Comment
QO	BOOL	Quality Output - TRUE if service executed successfully
STATUS	STRING	Service status (e.g., “IDLE”, “BUSY”, “ERROR”)
s16result	INT	Return value (see service description)

Adapter

Name	Type	Comment
pau8String	adapter::types::unidirectional::AIS	Returns the new string value

Quality signals QI/QO: The data input `QI` (BOOL, initial value `TRUE`) is associated with the event `INIT` and is passed through to `QI` of the internal block `Q_StringValue`. The data output `QO` (BOOL) is taken over from `Q_StringValue.QO` and is associated with the events `INITO` and `CNF`.

Valid Object IDs

`u16ObjId` is valid for the same object types as the wrapped base block `Q_StringValue` (Annex F.24, objects with string value attribute): Input String Field (8000–8999), Output String Field (11000–11999), String Variable (22000–22999), Input Attributes (26000–26999). The F.24 variable data length / ISO transport protocol behaviour of `Q_StringValue` also applies.

ID_NULL (65535) is not a command target but deactivates the FB when used with `INIT`.

Functionality

The function block internally contains an instance of the function block `Q_StringValue` (from the library `isobus::UT::Q`), which performs the actual ISOBUS communication. The wiring is as follows:

1. **Initialization**: An INIT event at the input activates the internal `Q_StringValue` function block. The provided `u16ObjId` is forwarded to this device.
2. **Triggering the Service**: As soon as the adapter `pau8String` receives an event (E1), the REQ event of the internal block is triggered. Simultaneously, the string value is passed to `Q_StringValue.pau8String` via the adapter’s data output D1.
3. **Feedback**: After processing is complete, the internal block outputs the events `INITO` and `CNF`, whose output data (`STATUS`, `s16result`) are directly passed to the corresponding outputs of the overall block.

All the logic is implemented in the included `Q_StringValue` block; the `Q_StringValue_AIS` serves as a specialized package with an adapter input.

Technical Features

- The block complies with the **ISO 11783-6** (ISOBUS) specification, Part 6, “Command Change String Value” service.
- The implementation uses a **unidirectional adapter** (`adapter::types::unidirectional::AIS`) that only transmits data from the host to the block. This simplifies integration into control systems where the new string value is provided asynchronously.
- The initial value of `u16ObjId` is `ID_NULL` – this must be set to a valid object ID before first use.
- The outputs `STATUS` and `s16result` provide the status and return value of the service as soon as the `CNF` event is triggered. The exact values are defined in the underlying service description.

State Overview

The function block itself does not have an explicit internal state machine, as the state logic is entirely contained within `Q_StringValue`. Essentially, the following phases can be distinguished:

- **Idle**: After starting or after a successful/failed execution – the function block waits for an INIT or an event on the adapter.
- **Initialization**: Active through INIT until the INITO event is sent.
- **Service Active**: After being triggered by the adapter, until the CNF event arrives.
- **Errors**: In case of invalid parameters or communication errors, a corresponding status/return value is provided.

Application Scenarios

- **ISOBUS Command:** Changing a string parameter on an agricultural device (e.g., machine name, task, variable name) via the ISOBUS network.
- **Control Systems:** Connection to a PLC or HMI that provides the new string value via an adapter (e.g., from a text box, a database field, or a communication channel).
- **Test Environments:** Simulating the service for development and testing purposes.

Comparison with Similar Function Blocks

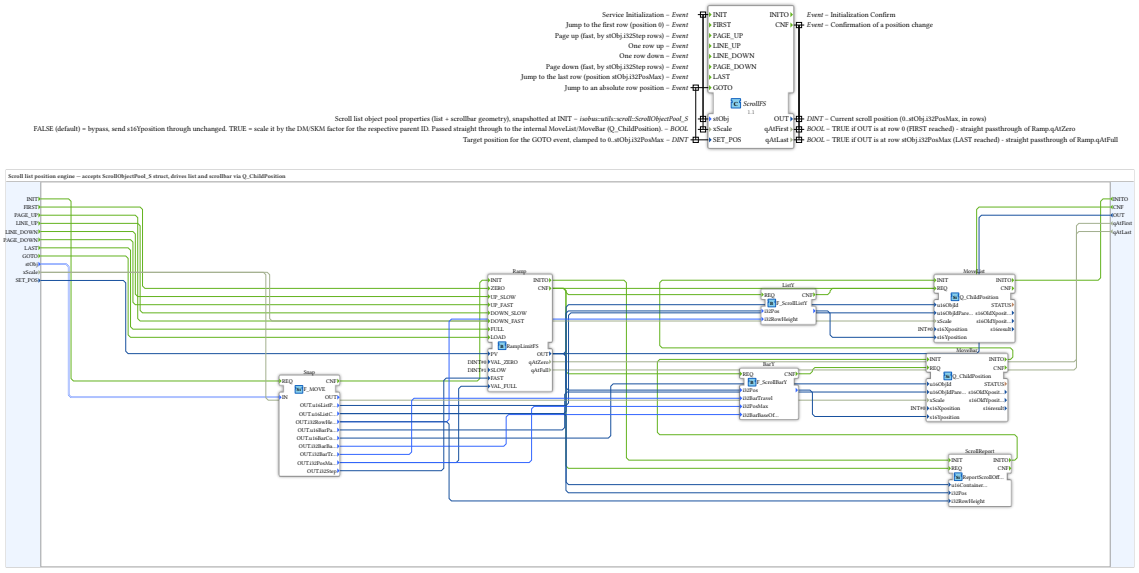
The `Q_StringValue_AIS` function block is a specialized variant of the `Q_StringValue` function block, which already includes all ISOBUS communication. The difference lies in the input interface: While `Q_StringValue` receives the string value via a standard data input `pau8String` (of type `STRING` array), `Q_StringValue_AIS` uses an adapter (AIS) that can supply data asynchronously. The adapter enables loose coupling and facilitates reuse in different contexts, for example, when the string value is provided by another, independent function block.

Other similar function blocks from the ISO 11783-6 family handle different data types (e.g., `Q_NumberValue_AIS` for numeric values).

Conclusion

The `Q_StringValue_AIS` is a practical, standards-compliant function block for modifying string values in an ISOBUS context. Using an adapter makes integration flexible and modular. It is particularly suitable for applications where the new value is provided asynchronously or from external sources. The clear separation of initialization and service execution, as well as the standardized outputs, facilitate debugging and reuse.

ScrollFS



• — ○ • — ○ • — ○ •

Introduction

`ScrollFS` is the pure position engine of a scrollable VT list: it holds the current row position (0… `i32PosMax`), reacts to navigation events (line/page up/down, first/last, absolute position) and writes the matching Y offset for both the list content and the scrollbar thumb via two `Q_ChildPosition` calls (ISO 11783-6 F.16, “Change Child Location”) to the VT. It reads **no** physical buttons or input fields — that’s what ScrollFS_PHYS_Softkey and ScrollFS_PHYS_Button are for, which wrap `ScrollFS` internally. See `Workspace_Scroll/SCROLL_KONZEPT.md` in the `4diac_training1` project for the full derivation.

Interface Structure

Event Inputs

- `INIT`: Snapshots the configuration (`stObj`) and takes `xScale`.
- `FIRST`: Jumps to the first row (position 0).

- PAGE_UP: Page up (fast, by stObj.i32Step rows).
- LINE_UP: One row up.
- LINE_DOWN: One row down.
- PAGE_DOWN: Page down (fast, by stObj.i32Step rows).
- LAST: Jumps to the last row (position stObj.i32PosMax).
- GOTO: Jumps to an absolute row position (SET_POS), clamped to 0…stObj.i32PosMax.

Event Outputs

- CNF: Confirms a position change, delivers OUT, qAtFirst and qAtLast.

Data Inputs

- stObj (isobus::utils::scroll::ScrollObjectPool_S): list/bar geometry (container IDs, row height, bar offsets, i32PosMax, i32Step) — snapshotted once at INIT.
- xScale (BOOL, default FALSE): FALSE = bypass, s16Yposition is sent through unchanged; TRUE = scale it by the DM/SKM factor for the respective parent ID. Passed straight through to the internal MoveList/MoveBar (Q_ChildPosition).
- SET_POS (DINT): Target position for GOTO.

Data Outputs

- OUT (DINT): Current scroll position (0…stObj.i32PosMax, in rows).
- qAtFirst (BOOL): TRUE if OUT is at row 0 (start of the list reached).
- qAtLast (BOOL): TRUE if OUT is at stObj.i32PosMax (end of the list reached).

Adapters

No adapters available.

Functionality

ScrollFS wires five blocks together into a complete position engine:

1. **Snap** (F_MOVE, DataType = ScrollObjectPool_S): snapshots stObj once at INIT, after which all nine geometry fields are permanently available as Snap.OUT.<field>.
2. **Ramp** (RampLimitFS, package eclipse4diac::signalprocessing): the actual position state machine, reused directly (no separate state machine needed). The eight ScrollFS events map 1:1 to RampLimitFS events:

ScrollFS event	RampLimitFS event	Effect on OUT
FIRST	ZERO	OUT := 0
PAGE_UP	DOWN_FAST	OUT -= stObj.i32Step (clamped at 0)
LINE_UP	DOWN_SLOW	OUT -= 1 (clamped at 0)
LINE_DOWN	UP_SLOW	OUT += 1 (clamped at i32PosMax)
PAGE_DOWN	UP_FAST	OUT += stObj.i32Step (clamped at i32PosMax)
LAST	FULL	OUT := stObj.i32PosMax
GOTO	LOAD (with PV := SET_POS)	OUT := SET_POS, clamped to 0…i32PosMax

VAL_ZERO := 0, VAL_FULL := Snap.OUT.i32PosMax, SLOW := 1, FAST := Snap.OUT.i32Step. 3. **ListY** (F_ScrollListY) and **BarY** (F_ScrollBarY): on every Ramp.CNF, compute the Y offset for the list and the scrollbar from Ramp.OUT. 4. **MoveList** and **MoveBar** (each a Q_ChildPosition): write the computed Y values to the VT — MoveList moves Container_Scrolling_Content within Container_Scrolling_Parent, MoveBar moves Container_Scrollbar_Content within Container_Scrollbar_Parent. Both have a fixed s16Xposition := 0 (vertical scrolling only). 5. **ScrollReport** (ReportScrollOffset): on every Ramp.CNF (and initially on Ramp.INIT0), reports the current scroll position (pixel offset = Ramp.OUT * Snap.OUT.i32RowHeight) for container Snap.OUT.u16ListContentId to the ECU visibility gate (VtMaskVisibility) so visible elements can be precisely filtered for VT refreshes.

Ramp.qAtZero/Ramp.qAtFull are passed through unchanged as qAtFirst/qAtLast.

Technical Details

- **`F_MOVE` snapshot for struct access**: `Snap.IN`/`.OUT` are generically `ANY`-typed; an `ANY` connection (here: the concretely-typed `stObj` into the generic `Snap.IN`) in 4diac always needs a `MOVE` block in between, never directly. Every individual struct field addressed via `Snap.OUT.<field>` needs its own `Parameter Name="OUT.<field>"` entry with `Attribute Name="Visible" Value="true"` on the `Snap` instance - without it, the network won't compile.
- **Init chain is serial, not parallel**: `Snap.CNF` triggers `Ramp.INIT`; only after `Ramp.INITO` do `MoveList.INIT` and `MoveBar.INIT` fire - guaranteeing `VAL_ZERO`/`VAL_FULL`/`SLOW`/`FAST` are set in `Ramp` before any position value is written to the VT.
- **`ListY`/`BarY` are concretely typed** (`DINT`/`INT`, see F_ScrollListY, F_ScrollBarY) - so no extra `MOVE` anchor is needed between them and `Ramp.OUT`, even though `Ramp.OUT` itself comes from a generic `RampLimitFS` instance.

State Overview

The only state lives in `Ramp` (`RampLimitFS`): `OUT` persists between events, each navigation event changes it incrementally or jumps directly. `qAtFirst`/`qAtLast` are recomputed on every change, never maintained separately.

Application Scenarios

- Never used directly — always wrapped by ScrollFS_PHYS_Softkey (softkeys) or ScrollFS_PHYS_Button (on-screen buttons), which derive the eight navigation events from physical button presses.

⚖️ Comparison with Similar Blocks

- **Versus `RampLimitFS`**: `ScrollFS` is at its core a `RampLimitFS` with VT wiring - the position math itself (clamping, limit flags) lives entirely in `RampLimitFS`; `ScrollFS` only adds struct configuration, row/bar conversion, and writing to the VT.

🛠️ Related Exercises

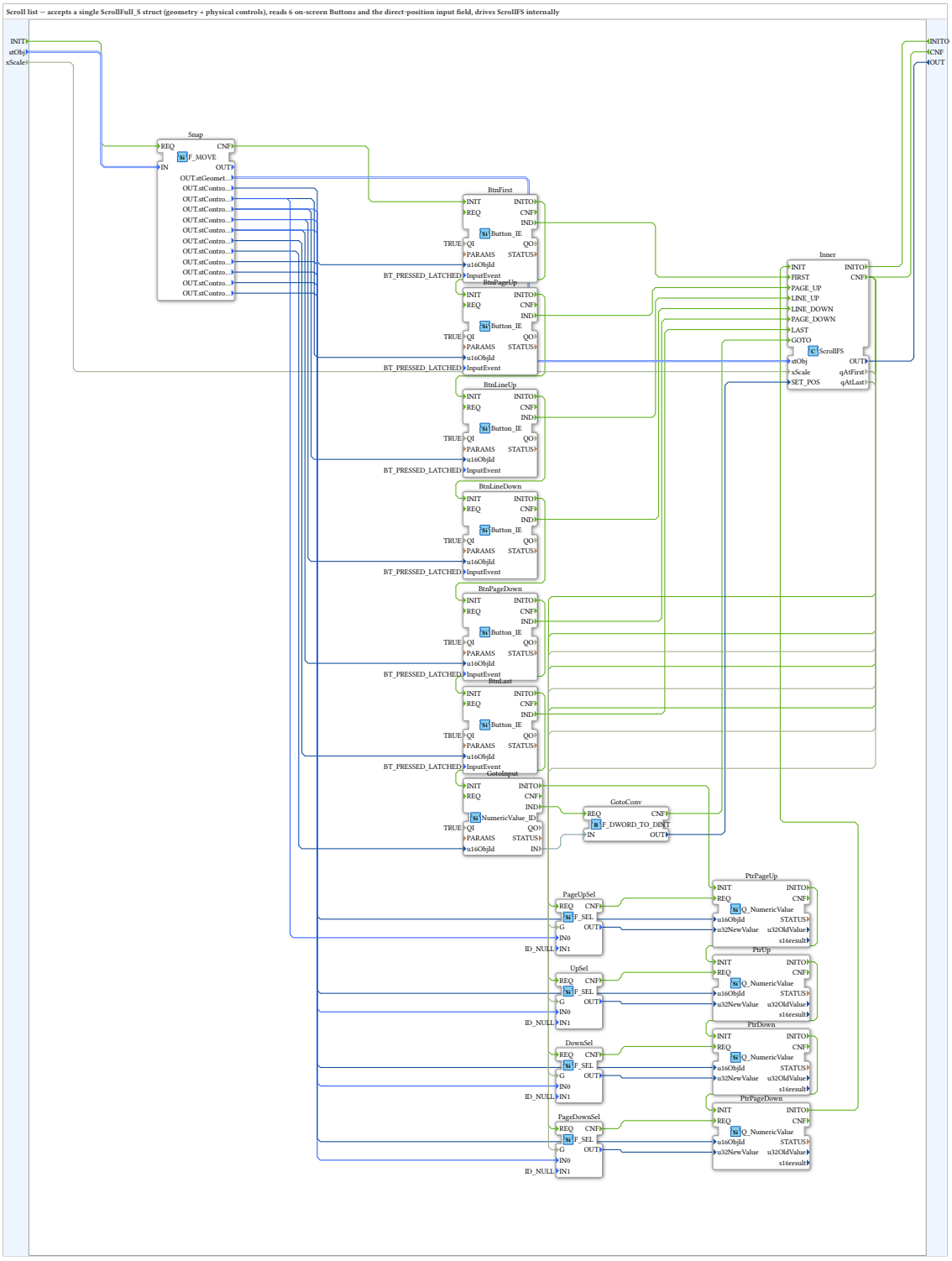
- No standalone exercise example — see `Workspace_Scroll/SCROLL_KONZEPT.md` (project `4diac_training1`) for the full derivation against a real pool.

Conclusion

`ScrollFS` bundles the position state, limit detection, and VT write access of a scrollable list into a single, reusable composite block — independent of whether it's later operated via softkeys or on-screen buttons.

ScrollFS_PHYS_Button





• — ○ • — ○ • — ○ •

Introduction

`ScrollFS_PHYS_Button` is structurally identical to ScrollFS_PHYS_Softkey — the same finished, drop-in scroll block, the same `ScrollFull_S` constant, the same limit hiding — but reads the 6 controls as on-screen `Button` objects (`Button_IE`, listening for `BT_PRESSED_LATCHED`) instead of physical `SoftKey` objects. Intended for masks that need scrolling without a softkey column available (e.g. touch operation). The 6 IDs in `ScrollControls_S` then point at `Button` instead of `SoftKey` objects; otherwise `ScrollFull_S`/`ScrollControls_S` are unchanged. See `Workspace_Scroll/SCROLL_KONZEPT.md` (project `4diac_training1`) for the full derivation.

Interface Structure

Event Inputs

- `INIT`: Snapshots the complete configuration (`stObj`) (with `stObj`, `xScale`).

Event Outputs

- `INITO`: Initialization confirmation.
- `CNF`: Confirms a position change, delivers `OUT`.

Data Inputs

- `stObj` (`isobus::utils::scroll::ScrollFull_S`): complete configuration (geometry + control IDs), snapshotted at `INIT`.
- `xScale` (BOOL): FALSE (default) = bypass, TRUE = scale by the DM/SKM factor. Passed straight through to the internal `ScrollFS` instance.

Data Outputs

- `OUT` (DINT): Current scroll position (0…`stObj.stGeometry.i32PosMax`, in rows).

Adapters

No adapters available.

Functionality

Identical to ScrollFS_PHYS_Softkey, with one difference: the 6 controls are `Button_IE` instances (`BtnFirst`, `BtnPageUp`, `BtnLineUp`, `BtnLineDown`, `BtnPageDown`, `BtnLast`) listening for `BT_PRESSED_LATCHED` instead of `SK_PRESSED`. Everything else - direct entry via `NumericValue_ID+F_DWORD_TO_DINT`, the internal ScrollFS position engine, limit hiding via `F_SEL+Q_NumericValue` on the 4 ObjectPointer IDs - is identical to `ScrollFS_PHYS_Softkey`.

Technical Details

See ScrollFS_PHYS_Softkey, “Technical Details” section - everything described there (F_SEL direction, ObjectPointer vs. Key ID, serial init chain) applies unchanged. The only difference is the control type (`Button_IE` instead of `Softkey_IE`).

State Overview

Same as ScrollFS_PHYS_Softkey - no state of its own beyond what `Inner` (`ScrollFS`/`RampLimitFS`) holds.

Application Scenarios

- Scrollable lists on masks without a softkey column, especially on touch-capable terminals where on-screen buttons are preferred over physical softkeys.

⚖️ Comparison with Similar Blocks

- **Versus `ScrollFS_PHYS_Softkey`**: identical except for the control type (`Button_IE`/`BT_PRESSED_LATCHED` instead of `Softkey_IE`/`SK_PRESSED`). See ScrollFS_PHYS_Softkey for the full description.
- **Versus `ScrollFS`**: `ScrollFS_PHYS_Button` is the practical wrapper - `ScrollFS` itself knows nothing about physical controls, only abstract navigation events.

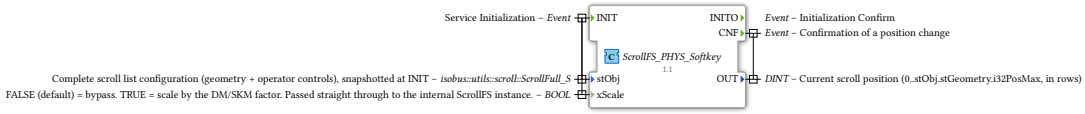
🛠️ Related Exercises

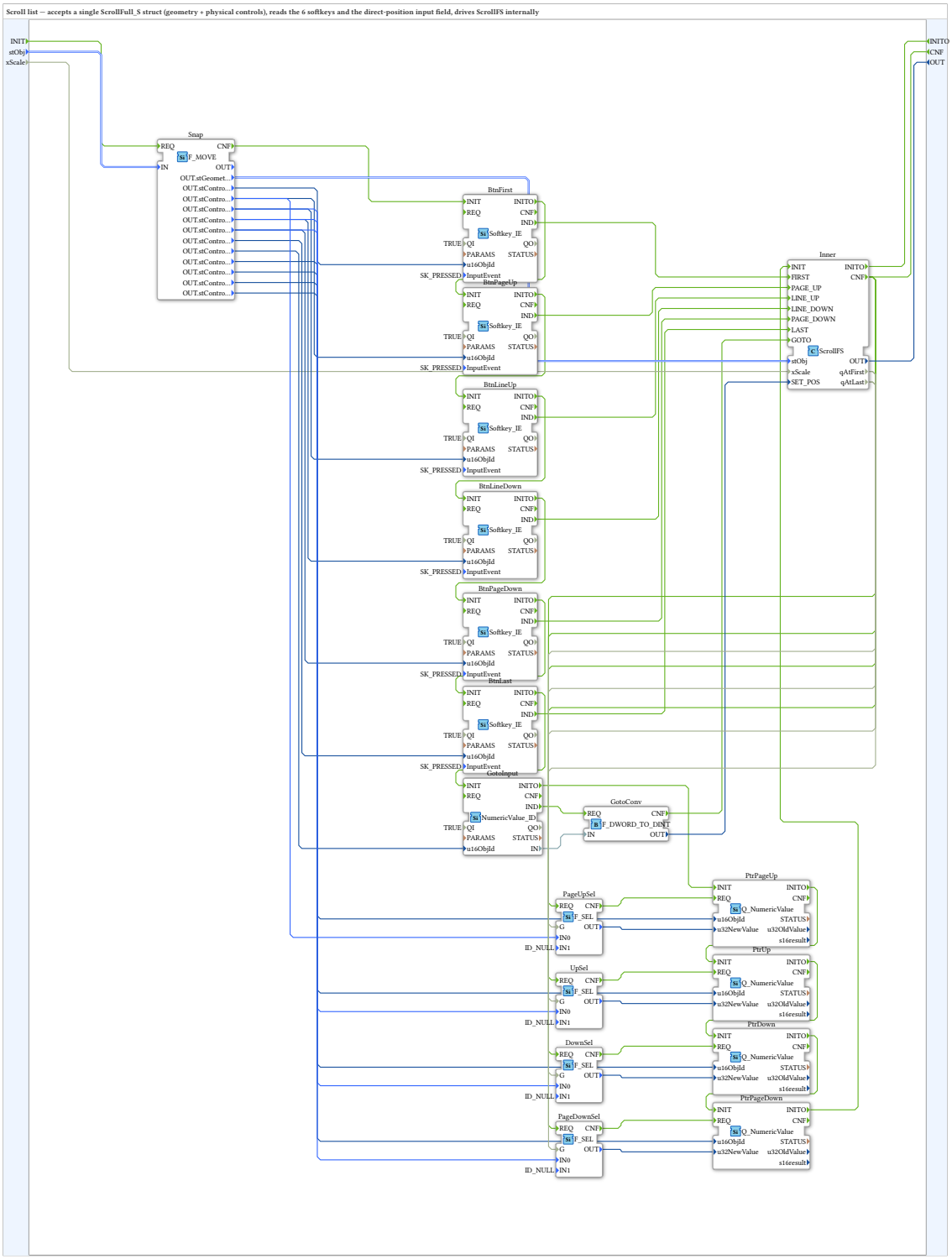
- No standalone exercise example — see `Workspace_Scroll/SCROLL_KONZEPT.md` (project `4diac_training1`).

Conclusion

`ScrollFS_PHYS_Button` offers the same finished scroll functionality as `ScrollFS_PHYS_Softkey`, just for on-screen buttons instead of softkeys - which variant fits depends solely on which object type the controls are built as in the pool.

ScrollFS_PHYS_Softkey





• — ○ • — ○ • — ○ •

Introduction

ScrollFS_PHYS_Softkey is the finished, drop-in block for a scrollable VT list operated via softkeys: feed it a ScrollFull_S constant (as generated by GcfScript.py from the ISO-Designer pool) and it wires up the 6 physical softkeys, the direct-entry field, and the internal position engine ScrollFS itself — including automatic hiding of the scroll buttons at each list limit. Two lists on one mask = two instances, each with its own ScrollFull_S constant. See Workspace_Scroll/SCROLL_KONZEPT.md (project 4diac_training1) for the full derivation.

Interface Structure

Event Inputs

- INIT: Snapshots the complete configuration (stObj), with xScale.

Event Outputs

- CNF: Confirms a position change, delivers OUT.

Data Inputs

- stObj (isobus::utils::scroll::ScrollFull_S): complete configuration (geometry + control IDs), snapshotted at INIT.
- xScale (BOOL, initial value FALSE): FALSE = bypass, TRUE = scale by the DM/SKM factor. Passed straight through to the internal ScrollFS instance.

Data Outputs

- OUT (DINT): Current scroll position (0…stObj.stGeometry.i32PosMax, in rows).

Adapters

No adapters available.

Functionality

1. **Snap** (F_MOVE, DataType = ScrollFull_S): snapshots stObj at INIT, after which geometry and all control IDs are available as Snap.OUT.stGeometry/ Snap.OUT.stControls.<field>.
2. **6× Softkey_IE** (BtnFirst, BtnPageUp, BtnLineUp, BtnLineDown, BtnPageDown, BtnLast), each listening for SK_PRESSED, u16ObjId taken from the corresponding Snap.OUT.stControls.u16Btn*Id — their IND event directly fires the matching ScrollFS navigation event on Inner.
3. **GotoInput** (NumericValue_ID) + **GotoConv** (F_DWORD_TO_DINT): reads the direct-entry field (u16GotoInputId) and feeds the value as SET_POS into Inner.GOTO.
4. **Inner** (ScrollFS): the actual position engine, see ScrollFS. Inner.OUT is passed through directly as OUT.
5. **Limit hiding** - one chain of F_SEL + Q_NumericValue per direction (up side: BtnPageUp+BtnLineUp, down side: BtnPageDown+BtnLineDown), triggered on every position change (Inner.CNF):
   - F_SEL.G := Inner.qAtFirst (up side) or Inner.qAtLast (down side)
   - F_SEL.IN0 := the real SoftKey ID (Snap.OUT.stControls.u16Btn*Id) - value when **not** at the limit
   - F_SEL.IN1 := ID_NULL - value once the limit is reached (F_SEL.OUT := G ? IN1 : IN0)
   - Q_NumericValue.u16ObjId := the matching ObjectPointer ID (Snap.OUT.stControls.u16Btn*PtrId), .u32NewValue := F_SEL.OUT
   - Result: the SoftKeyMask's ObjectPointer points at ID_NULL at the limit (button disappears), otherwise at the real SoftKey (button visible).

Technical Details

- **F_SEL direction is not symmetric**: F_SEL.OUT := G ? IN1 : IN0 - if IN0/IN1 are swapped, the button shows exactly when it should disappear (and vice versa). This ordering (IN0 = visible value, IN1 = ID_NULL) was swapped in an earlier version and had to be corrected.
- **Q_NumericValue on the ObjectPointer instead of "Hide/Show Object"**: Object Pointers are not hidden/shown via the ISO 11783-6 "Hide/Show Object" command (Annex F.2, valid for Container only), but via "Change Numeric Value" (Annex F.22) applied directly to the ObjectPointer's own ID - that changes what the pointer targets (ID_NULL = points at nothing = invisible). Reference pattern: Uebung_015 (project 4diac_training1).
- **Don't confuse the ObjectPointer ID with the Key ID**: Snap.OUT.stControls.u16Btn*Id is the **real** SoftKey object ID (target for Softkey_IE.u16ObjId **and** for F_SEL.IN0), Snap.OUT.stControls.u16Btn*PtrId is the ID of the **ObjectPointer** that points at that SoftKey inside

the SoftKeyMask tree (target for `Q_NumericValue.u16ObjId`) - both are resolved automatically from the pool by `GcfScript.py` (the SoftKeyMask child is the ObjectPointer, whose target is the real SoftKey via a `CProxy` indirection).
- **Serial init chain**: `Snap.CNF → BtnFirst.INIT → … → BtnLast.INIT → GotoInput.INIT → PtrPageUp.INIT → PtrUp.INIT → PtrDown.INIT → PtrPageDown.INIT → Inner.INIT` - each block only initializes once the previous one is done (`INITO` chain), matching the pattern used in ScrollFS.

State Overview

No state of its own beyond what `Inner` (`ScrollFS`/`RampLimitFS`) holds. Limit hiding is purely event-driven — every position change recomputes the visibility state of all four buttons, not just on the actual limit transition.

Application Scenarios

- Any scrollable VT list operated via physical softkeys (e.g. long selection lists, diagnostic/status overviews) - see `Workspace_Scroll` in project `4diac_training1` for a real, tested example.

⚖️ Comparison with Similar Blocks

- **Versus `ScrollFS`**: `ScrollFS_PHYS_Softkey` is the practical wrapper - `ScrollFS` itself knows nothing about physical controls, only abstract navigation events.
- **Versus `ScrollFS_PHYS_Button`**: identical structure and limit hiding, but reads the 6 controls as on-screen `Button` objects (`BT_PRESSED_LATCHED`) instead of `SoftKey` objects - for masks without a softkey column, or touch operation. See ScrollFS_PHYS_Button.

🛠️ Related Exercises

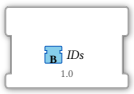
- No standalone exercise example — see `Workspace_Scroll/SCROLL_KONZEPT.md` (project `4diac_training1`).

Conclusion

`ScrollFS_PHYS_Softkey` is the complete, production-ready scroll block for softkey-operated lists - including automatic hiding of the navigation buttons at the limits, without the control application having to check limits or redirect Object Pointers itself.

const

IDs



• – ◦ • – ◦ • – ◦ •

Introduction

The global constant package `IDs` provides a collection of predefined, immutable values (constants) that can be used within function blocks of the IEC 61499 runtime environment. These constants primarily serve to improve the readability and maintainability of the source code by replacing magic numbers with meaningful identifiers. The package is defined in the namespace `isobus::UT::Q::const`.

Interface Structure

Because it is a package of global constants, it does not have event- or data-flow-based interfaces in the conventional sense of a function block. The constants are static and can be referenced directly by their names.

Event Inputs

Not present.

Event Outputs

Not present.

Data Inputs

Not present.

Data Outputs

Not present.

Adapters

Not present.

Functionality

The constants are replaced by their predefined values when the application is compiled. They behave like literals but have a semantic name. The constants included in the package are:

- `ID_NULL`: Represents an invalid or uninitialized identifier (ID) of type `UINT` with the hexadecimal value `16#FFFF` (decimal 65535).
- `ID_OFF`: Represents a special identifier value that signals "Off" or "Inactive," of type `UINT` with the value `16#0` (decimal 0).
- `GLOBAL`: An identifier of type `USINT` with the value `16#FF` (decimal 255), typically used for global or broadcast addressing.
- `OFF`: A simple status value of type `USINT` for the "Off" state with the value `16#0` (decimal 0).

Technical Features

- **Typing:** The constants are strongly typed (`UINT`, `USINT`). This ensures type safety when using them in assignments and comparisons.
- **Initial Value:** Each constant is assigned a fixed, unchanging initial value upon declaration.
- **Namespace:** The constants are encapsulated in the package `isobus::UT::Q::const`, preventing name conflicts with other libraries.

State Overview

Since these are static, constant values, there is no state machine.

Application Scenarios

- **Initialization:** Assigning a defined initial value (e.g., `ID_NULL`) to variables that will later receive a valid ID.
- **Comparison:** Checking whether a received or read ID corresponds to a specific value (e.g., `IF receivedID = IDs.ID_OFF THEN ...`).
- **Parameterization:** Using constants as meaningful parameters in function blocks instead of direct numerical values.
- **Command and Status Values:** Using `OFF` and `GLOBAL` to control processes or address devices in a network.

⚖️ Comparison with Similar Blocks

Unlike function blocks (FBs) with algorithms and event handling, this package simply provides data. It is comparable to:

- **ENUM Data Types:** While ENUMs define a list of named integer values of a type, this package defines individual, loose constants of various types.
- **#define-Preprocessor Directives (in C):** Both replace a name with a value. However, the constants in `IDs` are type-safe and manageable in the 4diac IDE.

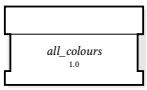
Conclusion

The global constant package `IDs` is a simple yet useful tool for structured programming in 4diac. It improves code quality, reduces typos in numeric values, and makes the programmer's intent explicit. Its use is especially recommended in larger projects or when working with protocols like ISOBUS, where special identifier values are common.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

all_colours



• – ◦ • – ◦ • – ◦ •

Introduction

The GlobalConstants block **all_colours** provides an extensive color palette with 256 defined colors (0 to 255). Each color is defined as a constant of type `USINT` and has a fixed numerical value. The comments contain the corresponding hexadecimal code, the RGB value, and English and German color names. Colors 0 to 231 correspond to the standardized ISOBUS color space, while colors 232 to 255 are reserved for manufacturer-specific purposes and are assigned a neutral gray.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

The colors are not provided via dedicated output variables, but are declared as global constants (`COLOR_0` … `COLOR_255`). These can be directly referenced by other function blocks.

Adapters

None.

Functionality

The function block declares the constants at system startup. The values are pre-initialized and remain unchanged throughout runtime. All constants are immediately valid and can be used in other function blocks without additional initialization. The numbering follows the ISOBUS standard, with the first 16 colors (0–15) covering the basic palette and the subsequent colors (16–231) containing more finely graduated shades.

Technical Features

- **Type:** `USINT` (0–255)
- **Standard Compliance:** Colors 0–231 are defined according to ISOBOBus; the remaining 24 entries (232–255) are manufacturer-specific and set to the same gray value (`#808080`).
- **Naming:** The designations `COLOR_0` … `COLOR_255` enable simple numerical referencing in algorithms or visualizations.
- **No Runtime Dependency:** The function block has no internal states or algorithms – it serves solely to provide constants.

State Overview

This function block has no state machines, as it is a purely declarative collection of constants.

Application Scenarios

- **ISOBUS-compliant UI development** in agricultural technology (terminals, machine controls)
- **Color coding of status information** (e.g., green = active, red = alarm, yellow = warning)
- **Standardization of colors** across multiple components or projects – all participants use the same color definition.

Comparison with similar function blocks

Unlike a typical `FUNCTION_BLOCK`, which processes input/output data and is controlled by events, `all_colours` is a **GlobalConstants** function block. It has no dynamic interfaces and no executable logic. Similar function blocks include, for example, `E_CTD` (counter) or `F_TRIG` (edge detection), which, however, exhibit active behavior. Other constant collections (e.g., physical units or thresholds) follow the same declarative pattern.

Conclusion

The `all_colours` function block provides a standardized, easily reusable color palette for ISOBUS applications. Clear numbering and comprehensive comments ensure consistent color usage across different modules and developers. Reserved colors (232–255) are available for customization.

colours



• – ○ • – ○ • – ○ •

Introduction

The global constant package `colours` defines a set of named constants for color values. These constants represent a set of 16 predefined colors, encoded by numeric values of type `USINT` (Unsigned Short Integer). Using such constants improves the readability and maintainability of IEC 61499 applications because meaningful identifiers like `COLOR_RED` or `COLOR_BLUE` can be used instead of magic numbers.

Interface Structure

A global constant package is not a function block in the classical sense; therefore, no event or data interfaces exist. Instead, the package provides a collection of constant declarations that can be imported and used in other function blocks or programs within the same project or namespace.

Event Inputs

Not applicable. Global constants do not have event inputs.

Event Outputs

Not applicable. Global constants do not have event outputs.

Data Inputs

Not applicable. Global constants do not have data inputs.

Data Outputs

Not applicable. Global constants do not have data outputs.

Adapters

Not applicable. Global constants do not have adapter interfaces.

How it Works

The `colours` package declares 16 constants of data type `USINT`. Each constant is assigned a fixed numeric value that represents a specific color. The values range from `USINT#0` to `USINT#15`. Once the package is imported into an application or function block, these constant names can be used wherever a `USINT` value is expected. The compiler replaces the constant name with the assigned numeric value during compilation.

Technical Details

- **Data Type:** All constants are of type `USINT` (Unsigned Short Integer), an unsigned 8-bit data type.
- **Value Range:** The defined values range from 0 to 15 and thus fit well within the value range of the `USINT` type.
- **Naming Convention:** The constant names follow the convention `COLOR_<FARBNAME>` (e.g., `COLOR_GREEN`), making them easily identifiable.
- **Compilation:** Values declared as `CONSTANT` are set at compile time and cannot be changed at runtime.

State Overview

Not applicable. Because they are static, unchanging values, global constants do not have a state machine.

Application Scenarios

This constant package is particularly useful in applications that require a limited color palette for user interfaces, visual status indicators (e.g., on HMIs or operator terminals), or for indicating process states. Typical use cases include:

- **Visualization:** Assigning colors to different machine states (e.g., Green = Operation, Red = Fault, Yellow = Warning).
- **ISOBUS-compliant applications (UT/Universal Terminal):** Standardizing color values in agricultural applications that use the ISOBUS protocol to ensure consistent display across different terminal devices.
- **Code improvement:** Replacing opaque numerical values in algorithms or configurations with self-explanatory identifiers.

⚖️ Comparison with similar building blocks

Unlike function blocks that execute logic or process data, a global constant package simply provides named values. Alternatives include:

1. **Direct use of numerical values:** Leads to code that is less readable and harder to maintain ("magic numbers").
2. **Defining constants within each function block:** Leads to redundancy and makes it difficult to centrally modify values.
3. **Using Enum Types:** Offers even greater type safety, but is not directly available as a global construct in the IEC 61499 standard library. Using `USINT` constants is a practical and widely used alternative.

Conclusion

The global constant package `colours` is a simple yet effective tool for structuring and improving IEC 61499 applications. By providing a standardized color palette as named constants, it promotes code quality, facilitates maintenance, and supports the development of consistent and understandable control applications, especially in the context of ISOBUS and visual interfaces.

####### AID

AID_AM

• – ◦ • – ◦ • – ◦ •

Introduction

The **AID_AM** global constant block defines attribute identifiers used within the ISO 11783 (ISOBUS) application layer for **Alarm Mask objects**. These constants provide standardized numeric IDs that allow applications to read or set specific attributes of an alarm mask, such as background colour, associated soft key mask, alarm priority, and acoustic signal configuration. The block is part of a package named `isobus::UT::Q::const::AID` and is intended to be used in 4diac-IDE projects that interact with ISOBUS terminals or implement virtual terminals.

Interface Structure

Since `AID_AM` is a **global constants** block, it does not contain event or data input/output ports. Instead, it exposes a set of compile-time constant values that can be referenced globally within the application.

Data Constants

Constant Name	Type	Initial Value	Description
`BACKGROUND_COLOUR`	`USINT`	1	Attribute ID for the background colour index of an Alarm Mask.
`SOFT_KEY_MASK`	`USINT`	2	Attribute ID for the object ID of a Soft Key Mask associated with the Alarm Mask.
`ALARM_PRIORITY`	`USINT`	3	Attribute ID for the alarm priority (0 = High, 1 = Medium, 2 = Low).

ACOUSTIC	USINT	4	Attribute ID for the acoustic signal setting (0 = highest priority, 1 = medium, 2 = lowest, 3 = silent).

Note: The constant values themselves are the attribute IDs that are used when communicating with an ISOBUS object pool or handling alarm mask properties.

Functionality

The primary purpose of AID_AM is to provide a human-readable and self-documenting set of constants that map to the numeric attribute IDs defined in the ISOBUS standard for alarm masks. Instead of hard-coding numeric values directly in function blocks or algorithms, developers can reference these constants to improve code clarity and maintainability. For example, to set the background colour of an alarm mask, one would use AID_AM.BACKGROUND_COLOUR as the attribute ID in a corresponding service call.

Technical Features

- **Standard Compliance**: The constants are aligned with ISO 11783-6 attribute definitions for Alarm Mask objects.
- **Type Safety**: All constants are declared as USINT (unsigned 8-bit integer), matching the expected attribute ID range (0–255).
- **Global Scope**: Being GLOBAL CONSTANT, they are accessible from any function block or resource in the project without explicit import.
- **Documentation**: Each constant includes a descriptive comment explaining its meaning and possible values.

State Overview

As a constants block, AID_AM has no internal state, states, or state transitions. It represents a static set of values that are available throughout the execution of the application.

Application Scenarios

- **Virtual Terminal Integration**: When implementing an ISOBUS virtual terminal or client, these constants are used to build or modify alarm mask attribute messages (e.g., Set Attribute, Get Attribute).
- **Alarm Handling**: In an agricultural machine control system, the constants help manage alarm priorities, acoustic feedback, and visual appearance of alarms on the terminal.
- **Object Pool Creation**: Developers creating an ISOBUS object pool for a terminal can use these IDs to correctly encode attribute references.

Comparison with Similar Blocks

AID_AM is one of several attribute-identifier constant blocks defined in the same package (e.g., AID_WK, AID_AR, etc.). While other blocks define attribute IDs for working sets, soft key masks, or other object types, AID_AM specifically targets alarm masks. Its simplicity lies in the absence of any logic or interface; it purely provides constants. Unlike a complex function block, it cannot fail, execute, or be reused in a state machine; it exists only to be referenced.

Conclusion

The AID_AM global constants block is a small but essential utility for ISOBUS applications dealing with alarm masks. By centralizing the attribute IDs, it reduces the risk of typing errors, improves code readability, and ensures consistency across the project. It is a perfect example of using compile-time constants to bring clarity to a standards-based communication protocol.

AID_AO

• – ◦ • – ◦ • – ◦ •

Introduction

AID_AO is a global constant definition block used in the 4diac IDE for the ISO 11783 (ISOBUS) Virtual Terminal protocol. It defines the attribute identifiers (AID) for animation objects (AO) within an ISOBUS Universal Terminal (UT) object pool. The block provides symbolic constant names that map to numerical attribute IDs, making configuration code more readable and maintainable when working with animation objects.

Interface Structure

AID_AO is a GlobalConstants type, not a function block or adapter. It contains no event or data interfaces. Instead, it exposes a set of global constant values of type USINT, available throughout the package `isobus::UT::Q::const::AID`.

Event Inputs

None.

Event Outputs

None.

Data Inputs

The following constants are defined as global, read-only values:

Name	Type	Value	Description
WIDTH	USINT	1	AID_AO_WIDTH – Width in pixels.
HEIGHT	USINT	2	AID_AO_HEIGHT – Height in pixels.
REFRESHINT	USINT	3	AID_AO_REFRESHINT – Refresh interval.
VALUE	USINT	4	AID_AO_VALUE – Current value.
ENABLED	USINT	5	AID_AO_ENABLED – 0 = Stopped, 1 = Animating.
FICHILDINDEX	USINT	6	AID_AO_FICHILDINDEX – First child index.
LACHILDINDEX	USINT	7	AID_AO_LACHILDINDEX – Last child index.
DEFCHILDINDEX	USINT	8	AID_AO_DEFCHILDINDEX – Default child index.
OBJECTS	USINT	9	AID_AO_OBJECTS – Options: Bit 0 = Animation Sequence (0 = Single Shot, 1 = Loop); Bits 1–2 = Disabled Behaviour (0 = Pause, 1 = Reset to First, 2 = Default Object, 3 = Blank).

Data Outputs

None.

Adapters

None.

Functionality

The AID_AO global constants serve as named references for the attribute IDs that configure and control ISOBUS animation objects. Each constant corresponds to a specific attribute used in the UT object pool:

- **WIDTH** and **HEIGHT** define the pixel dimensions of the animation object.
- **REFRESHINT** specifies the refresh interval for animation updates.
- **VALUE** holds the current animation value.
- **ENABLED** controls whether the animation is stopped or running.
- **FICHILDINDEX**, **LACHILDINDEX**, and **DEFCHILDINDEX** identify the first, last, and default child object indices respectively.
- **OBJECTS** packs two configuration options into one byte: the animation sequence mode (single shot or loop) and the behaviour when the animation is disabled (pause, reset to first, show default object, or blank).

These constants improve code readability and reduce the risk of errors that would occur if raw numeric IDs were used directly.

Technical Features

- Compliant with IEC 61499-1 standard for GlobalConstants.
- All constants are of type USINT (8-bit unsigned integer).
- Values are fixed at compile time and cannot be modified during runtime.
- The block is defined within the package namespace `isobus::UT::Q::const::AID`.

- The OBJECTS constant demonstrates bitwise encoding of multiple configuration flags into a single byte.

State Overview

Not applicable. AID_AO is a passive constant definition and does not implement stateful or sequential behaviour.

Application Scenarios

AID_AO is applied in ISOBUS Virtual Terminal applications where animation objects are configured and dynamically controlled. Typical use cases include:

- Setting the dimensions of an animation object via WIDTH and HEIGHT.
- Configuring the refresh rate with REFRESHINT.
- Starting or stopping animations using ENABLED.
- Reading or setting the current animation value via VALUE.
- Managing child object ordering with FICHILDINDEX, LACHILDINDEX, and DEFCHILDINDEX.
- Selecting animation mode (loop vs. single shot) and disabled behaviour through OBJECTS.

Comparison with Similar Blocks

AID_AO belongs to a family of attribute ID constant definitions used in ISOBUS VT contexts. Unlike function blocks, which encapsulate behaviour and state transitions, AID_AO provides only static constant values. It is comparable to other AID constant groups (e.g., for object types or control functions) but focuses specifically on animation objects. Its advantage lies in providing a single, consistent source of truth for animation attribute identifiers, eliminating magic numbers in application code.

Conclusion

AID_AO is a compact and well-structured global constant set that centralises all animation object attribute identifiers in an ISOBUS environment. By offering symbolic names for numeric IDs and encoding complex options into a single byte, it enhances code clarity, maintainability, and correctness. It is a fundamental building block for developers implementing ISOBUS Virtual Terminal animations within the 4diac IDE.

AID_AUXC2

• – ◦ • – ◦ • – ◦ •

Introduction

The AID_AUXC2 global constants provide the attribute identifiers for the Auxiliary Control Designator Type 2 (AuxC2) object within the ISOBUS (ISO 11783) protocol. These constants define the numeric IDs used to reference the pointer type and object ID attributes of an auxiliary control designator of type 2. The constants belong to the package `isobus::UT::Q::const::AID` and are used to standardize attribute access in ISOBUS device implementations.

Interface Structure

AID_AUXC2 is a global constant container and does not expose a runtime interface. Consequently, it has no event or data inputs/outputs or adapters.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None. The following global constants are provided instead:

Constant	Data Type	Initial Value	Description
`PTR_TYPE`	USINT	`USINT#1`	Attribute ID 1 (`AID_AUXC2_PTR_TYPE`) – Pointer Type. Defines what the pointer of the AuxC2 object references: 0 = Points to Aux Object, 1 = Points to Aux Function/Input, 2 = Points to Working Set owner of pointer, 3 = Points to Working Set owner of assigned function/input.

OBJ_ID	USINT	USINT#2	Attribute ID 2 (AID_AUXC2_OBJ_ID) – Object ID. Contains the object ID of a referenced Auxiliary Function or Auxiliary Input object, or NULL.

Data Outputs

None.

Adapters

None.

Functionality

The AID_AUXC2 constants are used as attribute identifiers when accessing the properties of an Auxiliary Control Designator Type 2 object in an ISOBUS system. In the ISOBUS standard, every object property is addressed via an attribute ID. These constants map the human-readable property names (PTR_TYPE, OBJ_ID) to their numeric attribute identifiers (1 and 2).

By using these pre-defined constants, application code can avoid magic numbers and ensure compliance with the ISOBUS attribute ID scheme. The PTR_TYPE constant distinguishes between different pointer destinations, while OBJ_ID specifies the referenced auxiliary object.

Technical Features

- **Global constant container**: The constants are declared as GLOBALCONSTANTS and can be referenced throughout a 4diac-IDE project without instantiation.
- **Unsigned short integer type**: Both constants are of type USINT (8-bit unsigned integer) which matches the ISOBUS attribute ID field size.
- **Pre-initialized values**: The initial values correspond to the official ISOBUS attribute IDs.
- **Package scoping**: The constants are located in the package isobus::UT::Q::const::AID, providing a clear namespace for auxiliary control designator constants.
- **SPDX-License-Identifier**: The implementation is provided under the Eclipse Public License 2.0.

State Overview

Since AID_AUXC2 is a compile-time constant set, it does not maintain any runtime state. No state transitions or lifecycle management is required.

Application Scenarios

The AID_AUXC2 constants are intended to be used in ISOBUS-compliant agricultural machinery control systems that implement the Auxiliary Control Designator Type 2 functionality. Typical applications include:

- ISOBUS Universal Terminal (UT) implementations that need to read or write the auxiliary control designator properties.
- Implement control ECU (Tractor-Implement Management) software that processes AuxC2 objects.
- Diagnostic and configuration tools that operate on the auxiliary control interface.

Comparison with Similar Blocks

The AID_AUXC2 constants are specifically tailored for Auxiliary Control Designator Type 2 objects. Other constant sets exist in the isobus::UT::Q::const::AID package for different object types (e.g., auxiliary control designator type 1). While those sets share the same pattern of numeric attribute IDs, their actual values and semantics differ based on the respective object type. The AID_AUXC2 set focuses solely on the two attributes required to define a type 2 pointer and its referenced object.

Conclusion

The AID_AUXC2 global constants provide a standardized, type-safe way to reference the attribute IDs of Auxiliary Control Designator Type 2 objects in ISOBUS applications. By encapsulating the numeric IDs as named constants, the component enhances code readability, reduces error-proneness, and ensures interoperability with the ISOBUS protocol specification.

AID_AUXF2

• – ◦ • – ◦ • – ◦ •

Introduction

AID_AUXF2 is a `GLOBALCONSTANTS` type in 4diac-ide that defines the object attribute IDs of an ISO 11783 ISOBUS Auxiliary Function Type 2 (AUX-F2) object. It is not an executable function block, but a compile-time constant container that makes object attribute IDs available as named symbols. The constants are part of the namespace `isobus::UT::Q::const::AID`.

Interface Structure

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

AID_AUXF2 does not have a dynamic IEC 61499 interface with event or data ports. Instead, it provides the following global constants:

Constant	Type	Value	Description
`BACKGROUND_COLOUR`	`USINT`	`USINT#1`	Background colour index attribute ID.
`FUNC`	`USINT`	`USINT#2`	Function attribute ID. Value is a bitmask: bits 0-4 = auxiliary function type (0-14), bit 5 = critical control, bit 6 = assignment restriction, bit 7 = single-assignment.

Functionality

The purpose of AID_AUXF2 is to provide named constants for the attribute IDs used by an Auxiliary Function Type 2 object. The two constants allow application code to refer to AUX-F2 attributes without using hard-coded integer values.

- `BACKGROUND_COLOUR` identifies the background colour index attribute.
- `FUNC` identifies the auxiliary function attribute and describes the function as a bitmask.

Because these values are global constants, they can be used in expressions, comparisons, or case selections without requiring runtime initialization or additional storage.

Technical Features

- Defined as `GLOBALCONSTANTS`, not as an executable function block.
- Both constants are of type `USINT`.
- `BACKGROUND_COLOUR` has the fixed value `1`.
- `FUNC` has the fixed value `2`.
- The constants are available in the namespace `isobus::UT::Q::const::AID`.
- The container has no events, data inputs, data outputs, adapters, or algorithms.
- The `FUNC` attribute uses bits 0-4 for the auxiliary function type, bit 5 for critical control, bit 6 for assignment restriction, and bit 7 for single-assignment.

State Overview

Not applicable. A `GLOBALCONSTANTS` type has no ECC state machine, no execution states, and no runtime behavior. The constants are static and are available to all IEC 61499 resources that reference this definition.

Application Scenarios

- **ISOBUS Virtual Terminal applications**: Use `AID_AUXF2.BACKGROUND_COLOUR` and `AID_AUXF2.FUNC` as attribute IDs when working with Auxiliary Function Type 2 objects.

- **Object pool generation**: Reference the constants when creating or modifying AUX-F2 object attributes in an ISOBUS object pool.
- **Readability improvement**: Replace numeric attribute IDs with symbolic names to reduce errors and improve code maintainability.

Example usage:

```
IF attrID = AID_AUXF2.FUNC THEN
    (* Process auxiliary function bitmask *)
END_IF
```

Comparison with Similar Blocks

AID_AUXF2 is not a function block and therefore cannot be compared with function blocks in terms of events, states, or data flow. It is best compared with other global constant containers that define ISOBUS object attribute IDs. Similar containers exist for other auxiliary function object types, such as Auxiliary Function Type 1 or Type 3. The main difference is the object type being addressed and the set of attribute IDs defined by each container. AID_AUXF2 is specifically dedicated to the Auxiliary Function Type 2 object and defines the two attribute IDs relevant for this object.

Conclusion

AID_AUXF2 is a compact global constant definition that provides named attribute IDs for ISOBUS Auxiliary Function Type 2 objects. By defining `BACKGROUND_COLOUR` and `FUNC` as named constants, it improves code clarity, reduces the use of magic numbers, and supports maintainable VT application development in 4diac-ide.

AID_AUXI2

• – ◦ • – ◦ • – ◦ •

Introduction

AID_AUXI2 is a global constant definition within the `isobus::UT::Q::const::AID` package of the 4diac-ide environment. It encapsulates a set of attribute identifiers (AIDs) specifically related to the ISO 11783 (ISOBUS) Auxiliary Input Type 2 object. These constants are used to reference individual attributes of an Auxiliary Input Type 2 object when interacting with ISOBUS virtual terminal data structures. The definition provides stable, human-readable symbolic names for the underlying attribute ID values, improving code clarity and maintainability in ISOBUS application implementations.

Interface Structure

Since AID_AUXI2 is a global constant definition rather than a function block, adapter, or subapplication, it does not expose event/data interfaces or adapters. Instead, it declares two global constants that can be referenced by other FB types across the project.

Constant	Type	Initial Value	Description
`BACKGROUND_COLOUR`	`USINT`	`1`	Attribute ID 1: Background colour index of the auxiliary input type 2 object.
`FUNC`	`USINT`	`2`	Attribute ID 2: Bitmask encoding the auxiliary function type (bits 0–4), critical control flag (bit 5), reserved bit (bit 6, set to 0), and single-assignment flag (bit 7).

Event Inputs

- None (not applicable for a global constant definition).

Event Outputs

- None (not applicable for a global constant definition).

Data Inputs

- None (not applicable for a global constant definition).

Data Outputs

- None (not applicable for a global constant definition).

Adapters

- None (not applicable for a global constant definition).

Functionality

The AID_AUXI2 global constants provide the numeric attribute ID values used to address attributes of an Auxiliary Input Type 2 (AUXI2) object within an ISOBUS virtual terminal. The constants enable code to refer to these attributes symbolically rather than using magic numbers:

- `BACKGROUND_COLOUR` is the attribute ID for the background colour index of the object.
- `FUNC` is the attribute ID for the function descriptor, a bitmask with the following layout:
  - Bits 0–4: Auxiliary function type (values 0–14).
  - Bit 5: Critical Control flag.
  - Bit 6: Reserved, must be set to 0.
  - Bit 7: Single-assignment flag.

These constants are intended to be used in conjunction with ISOBUS object pool manipulation APIs, where an attribute ID is required to specify which property of an object is being read or written.

Technical Features

- **Fixed Identification**: Each constant maps directly to a standardized ISOBUS attribute ID (1 and 2 respectively).
- **Type Safety**: Constants are declared as `USINT` (unsigned short integer), matching the ISOBUS attribute ID data type.
- **Single Assignment**: Defined as `VAR_GLOBAL CONSTANT`, the values are immutable at runtime.
- **Bit-Level Semantics**: The `FUNC` constant encodes multiple flags in a single byte, requiring bitwise operations to decode or set individual bits.
- **Package Organization**: Placed in the `isobus::UT::Q::const::AID` package namespace for structured reuse across FBs.

State Overview

A global constant definition is stateless. There is no internal state, no initialization sequence, and no runtime transitions. The values are available immediately once the containing project is loaded and remain constant throughout the execution lifetime.

Application Scenarios

- **ISOBUS Virtual Terminal Implementation**: When constructing an Auxiliary Input Type 2 object in an object pool, developers use `AID_AUXI2.BACKGROUND_COLOUR` to set or read the background colour attribute.
- **Auxiliary Function Configuration**: The `FUNC` constant is used when configuring the functional behaviour of an auxiliary input (e.g., setting the function type, marking an input as critical, or enforcing single assignment).
- **Attribute Iteration**: The constants can be used as enumeration values when building generic attribute-handling routines for AUXI2 objects.
- **Protocol Debugging**: In diagnostic tools, the symbolic names make trace logs and breakpoints more readable than raw attribute ID numbers.

Comparison with Similar Blocks

AID_AUXI2 is part of a family of attribute ID constant sets for different ISOBUS object types (e.g., `AID_AUXI1`, `AID_AUXI2`, `AID_*`). Compared to similar constant sets:

- It is specific to the Auxiliary Input Type 2 object, thus its attribute list covers only background colour and function descriptor.
- It does not contain attributes for labels, visibility, or other object properties that are common in other object type constant sets (e.g., Button, Output, or Input Boolean objects).
- Its `FUNC` constant provides a richer bitmask definition than simpler attribute constants, reflecting the unique capabilities of auxiliary input objects.
- Unlike FB-based implementations, it offers no processing logic, only the constant data, making it lighter and purely declarative.

Conclusion

AID_AUXI2 is a concise and well-structured global constant definition that encapsulates the essential attribute IDs for ISOBUS Auxiliary Input Type 2 objects. By providing symbolic names for the background colour and function descriptor attributes, it enhances the readability and type safety of ISOBUS application code. Its stateless, immutable nature makes it a reliable and reusable building block within the 4diac-ide environment for agricultural machinery communication systems.

AID_BO

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_BO` global constants type defines symbolic names for the attribute identifiers used to describe a button object in the ISOBUS (ISO 11783) standard. These identifiers are used when reading or writing button properties via the ISO-11783 protocol, providing a clear and maintainable way to refer to numeric attribute codes in application code.

Interface Structure

Since `AID_BO` is a global constants type, it does not contain any input or output variables in the traditional function-block sense. Instead, it exposes a set of read-only constant values that can be used throughout the application. These constants are listed below under **Data Outputs** for clarity, although they are not dynamic outputs.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

The following global constants are available within the `isobus::UT::Q::const::AID` package.

Constant Name	Type	Value (USINT)	Description
`WIDTH`	USINT	1	`AID_BO_WIDTH` – Width of the button in pixels.
`HEIGHT`	USINT	2	`AID_BO_HEIGHT` – Height of the button in pixels.
`BACKGROUND_COLOUR`	USINT	3	`AID_BO_BACKGROUND_COLOUR` – Background colour index.
`BORDER_COLOUR`	USINT	4	`AID_BO_BORDER_COLOUR` – Border colour index.
`KEY_CODE`	USINT	5	`AID_BO_KEY_CODE` – Key code associated with the button.
`OPTIONS`	USINT	6	`AID_BO_OPTIONS` – Bitmask containing several options:- Bit 0: latchable- Bit 1: state (0 = released, 1 = latched)- Bit 2: suppress border- Bit 3: transparent background- Bit 4: disabled- Bit 5: no border

Adapters

None.

Functionality

The `AID_BO` constants serve as a centralised registry for button object attribute identifiers. In an ISOBUS application, object attributes are addressed by numeric IDs. Using symbolic constants improves code readability and reduces the risk of errors caused by hard-coded numeric values. The package name

`isobus::UT::Q::const::AID` indicates that these constants belong to the ISOBUS utility toolkit (`UT`) and are part of the `Q` – likely "query" or "quick" – subcomponent.

The `OPTIONS` constant is particularly important as it encapsulates a bit-coded behaviour of the button. Applications can evaluate individual bits to determine the button's properties (e.g., whether it is latchable, its current latched state, whether the border is suppressed, etc.).

Technical Features

- **Data type**: All constants are of type `USINT` (unsigned short integer, 8-bit).
- **Initial values**: Each constant is explicitly initialised to its corresponding attribute ID.
- **Package**: `isobus::UT::Q::const::AID` – the global constants are grouped under this package, allowing namespaces to be used in IEC 61499 applications.
- **Compatibility**: The identifiers follow the official ISO 11783-6 (ISOBUS) attribute definitions, ensuring interoperability with other ISOBUS-compliant devices.
- **Source availability**: The original source code is embedded in the XML, making the definitions self-contained and accessible for reference.

State Overview

The `AID_BO` type has no dynamic state information. It is a static collection of constants that do not change during runtime. The values are fixed at compile time and remain unchanged throughout the execution of the application.

Application Scenarios

Typical use cases for `AID_BO` include:

- **Implementation of an ISOBUS-VT (Virtual Terminal) client**: When constructing or updating a button object on the VT, the application needs to send attribute IDs for width, height, background colour, etc. Using `AID_BO` constants makes the code self-documenting and less prone to mistakes.
- **Parsing feedback messages**: When the VT sends back the current state of a button (e.g., latched or released), the application can interpret the `OPTIONS` value by referencing the bits defined in this constant.
- **Maintenance and debugging**: Developers can trace protocol messages more easily when symbolic names are used in log output.

Comparison with Similar Blocks

While `AID_BO` is a global constants type, other similar constant types may exist for other ISOBUS objects (e.g., `AID_AL` for alarm objects, `AID_NM` for numerical display, etc.). These constants typically follow the same pattern – each object type has a dedicated set of attribute identifiers defined as USINT constants. The advantage of such symbolic constants over raw numbers is consistency and clarity across a project. Compared to function blocks, `AID_BO` does not contain any logic or user interface; it is purely a data dictionary.

Conclusion

The `AID_BO` global constants provide a well-structured and standardised way to reference button object attributes in an ISOBUS application. By using these symbolic names, developers can write more reliable and maintainable code, reduce the chance of encoding errors, and improve overall readability of IEC 61499 applications that interface with ISOBUS virtual terminals. The explicit packaging under `isobus::UT::Q::const::AID` further enhances modularity and integration into larger projects.

AID_CO

• – ◦ • – ◦ • – ◦ •

Introduction

AID_CO is a global constants definition block within the ISOBUS UT (Universal Terminal) interface library, specifically in the `isobus::UT::Q::const::AID` package. It defines constant attribute identifiers (IDs) used to reference attributes of container objects (CO) in ISOBUS object pool implementations. These attribute IDs conform to ISO 11783-6 (ISOBUS) standards for Universal Terminal object containers.

The constants map semantic meanings (e.g., width, height, hidden state) to numeric attribute IDs used in communication between a Virtual Terminal (VT) and an implement ECU.

Interface Structure

This component is not a function block or adapter; it defines global compile-time constants. As such, it has no event inputs/outputs, no data I/O ports, and no adapters in the runtime sense. The constants act as named identifiers for use in application logic.

Event Inputs

Not applicable – no event inputs are defined in this global constants container.

Event Outputs

Not applicable – no event outputs are defined in this global constants container.

Data Inputs

Not applicable – this global constants container does not accept external data inputs.

Data Outputs

Not applicable – this global constants container does not provide dynamic data outputs. Instead, it exposes the following compile-time constants:

Constant	Type	Value	Attribute ID	Description
`WIDTH`	USINT	1	AID_CO_WIDTH	Maximum width of the container's area in pixels. Objects or portions outside this area are clipped.
`HEIGHT`	USINT	2	AID_CO_HEIGHT	Maximum height of the container's area in pixels. Objects or portions outside this area are clipped.
`HIDDEN`	USINT	3	AID_CO_HIDDEN	0 = FALSE, 1 = TRUE. Indicates whether this container and its child objects are hidden.

Adapters

Not applicable – no adapters are defined.

Functionality

The AID_CO global constants provide numeric identifiers for the standard attributes of ISOBUS container objects. These constants are used when constructing attribute lists in object pool building or when setting/reading object attributes at runtime via the ISOBUS protocol. Using named constants instead of raw numeric values improves code readability and maintains compliance with the ISO 11783-6 specification.

- `WIDTH` (1) and `HEIGHT` (2) define the clipping boundary of the container's drawing area.
- `HIDDEN` (3) controls the visibility of the container and its child objects.

Technical Features

- Defined in the `isobus::UT::Q::const::AID` package.
- All constants are of type `USINT` (Unsigned Short Integer) with explicit initialization.
- Values are aligned with the ISO 11783-6 attribute ID enumeration for container objects.
- Constant values are compile-time evaluated and cannot be changed at runtime.
- License: Eclipse Public License 2.0 (EPL-2.0).
- Version: 1.0, published 2026-06-20.

State Overview

This component has no internal state machine or runtime states. It is a purely declarative constants container. All values are constant and available at compile time.

Application Scenarios

- **ISOBUS Object Pool Generation:** Use `AID_CO.WIDTH`, `AID_CO.HEIGHT`, and `AID_CO.HIDDEN` when creating container object attributes in an object pool for a Virtual Terminal.
- **VT Runtime Communication:** Refer to these constants when setting or requesting container object attributes over the ISOBUS network.
- **Clipping and Visibility Control:** Apply the `HIDDEN` attribute (value 3) to show or hide containers dynamically, and `WIDTH`/`HEIGHT` to define the container's clipping rectangle.

Comparison with Similar Blocks

In the ISOBUS `AID_*` constant families, `AID_CO` is responsible for container object attributes specifically. Similar constant groups include:

- `AID_OB` – object base attributes.
- `AID_WO` – working set attributes.
- `AID_OK` – object key attributes.

While those constants define attributes for other object types (e.g., buttons, keys, working sets), `AID_CO` focuses exclusively on container objects as defined in ISO 11783-6. Compared to generic numeric literals, using `AID_CO` constants increases readability and reduces the risk of using incorrect attribute IDs.

Conclusion

The `AID_CO` global constants block provides a standardized, human-readable set of attribute identifiers for ISOBUS container objects. It is a lightweight, compile-time-only component that aids in implementing ISO 11783-6 compliant object pools and VT communication logic. Its use improves maintainability and ensures consistency across applications using the `isobus::UT::Q::const::AID` library.

AID_CP

• – ◦ • – ◦ • – ◦ •

Introduction

The **AID_CP** global constants definition provides a set of named constant values used as attribute identifiers for colour palette objects in the ISOBUS (ISO 11783) protocol context. It is part of the `isobus::UT::Q::const::AID` package and contains a single constant, `OPTIONS`, which is used to reference the options attribute of a colour palette object.

Interface Structure

Since `AID_CP` is a global constants declaration (not a function block, adapter, or subapplication), it does not define event or data interfaces in the traditional sense. However, for consistency with the documentation template, the following sections explicitly state that no I/O elements are present.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

The sole constant defined within `AID_CP` is:

- **OPTIONS** (USINT) – Value `USINT#1`. This constant represents the attribute ID for the “Options” field of a colour palette object. According to the ISOBUS specification, the options attribute is always set to 0, but the constant is defined to standardise access to it in code.

Functionality

The purpose of `AID_CP` is to provide a central, reusable definition for the colour palette object attribute identifier. By using this constant, developers can avoid hard-coding numeric values in their logic and can ensure consistency across the system. The constant `OPTIONS` is intended to be used wherever the options attribute of a colour palette object needs to be referenced, e.g., in get/set operations or in communication frames.

Technical Features

- **Constant Data Type:** `USINT` (unsigned short integer, 8–bit).
- **Initial Value:** `1` (although the comment notes that the actual option value is always `0`; the constant serves as an attribute ID, not as a value).
- **Namespace:** The constant is defined within the `isobus::UT::Q::const::AID` package, aligning with ISOBUS utility layer conventions.
- **Packaging:** The XML declaration includes a standard identification and version information, making it suitable for exchange in 4diac-ide projects.

State Overview

As a global constants object, `AID_CP` does not have internal states or transitions. It is a compile-time constant set; its value is fixed at design time and does not change during runtime.

Application Scenarios

Typical use cases include:

- Implementing ISOBUS colour palette management in agricultural machinery control systems.
- Providing a shared constant for attribute identification across multiple function blocks or applications that handle colour palette objects.
- Serving as a reference in test and simulation environments where colour palette attributes are accessed.

Comparison with Similar Blocks

Unlike function blocks (which encapsulate behaviour and have event/data interfaces) or adapters (which define communication patterns), `AID_CP` is a passive data container. It cannot be instantiated or executed; it simply provides named constants. Similar global constants objects in the ISOBUS domain exist for other object types (e.g., AID_GA for geometric attributes), each following the same pattern of defining attribute IDs as named constants.

Conclusion

The `AID_CP` global constants object defines a single, well-documented constant (`OPTIONS`) for colour palette object attribute identification. It contributes to code clarity and maintainability by centralising an otherwise numeric value in a descriptive, reusable form. This approach aligns with best practices for ISOBUS development in the 4diac environment.

AID_DM

• – ◦ • – ◦ • – ◦ •

Introduction

AID_DM is a global constant definition used within 4diac IDE to define attribute identifiers for data mask objects in the ISOBUS protocol. It provides symbolic constants for background colour and soft key mask attributes, promoting code readability and maintainability in ISOBUS-based applications.

Interface Structure

Event Inputs

None applicable – this is a global constant definition, not a function block.

Event Outputs

None applicable.

Data Inputs

None applicable.

Data Outputs

None applicable.

Adapters

None applicable.

Functionality

This global constant set defines two `USINT` constants:

- `BACKGROUND_COLOUR` (value 1): corresponds to the attribute ID `AID_DM_BACKGROUND_COLOUR`, used to specify the background colour index of a data mask.
- `SOFT_KEY_MASK` (value 2): corresponds to the attribute ID `AID_DM_SOFT_KEY_MASK`, used to specify the object ID of a Soft Key Mask associated with the data mask.

These constants are intended to be used in ISOBUS-based applications to reference these attribute IDs symbolically, improving clarity and reducing hard-coded numeric values.

Technical Features

- Defined in package `isobus::UT::Q::const::AID`.
- Uses data type `USINT` (unsigned short integer).
- Constant values are preinitialized: 1 and 2.
- The constants are global and read-only within the 4diac environment.

State Overview

Not applicable – this definition does not contain states or state machines; it is a static data definition.

Application Scenarios

- Developing ISOBUS-compliant applications that require referencing Data Mask attributes.
- Used in conjunction with ISOBUS object pool management to set or query background colour and soft key mask attributes.
- Provides a centralized and standardised way to access these attribute identifiers in 4diac projects.

Comparison with Similar Blocks

Since this is a global constant definition, it is similar to other constant collections in 4diac that define attribute or parameter IDs. For example, other `AID_*` global constants might define attribute IDs for different ISOBUS objects. This particular set focuses solely on Data Mask attributes, making it lightweight and specific.

Conclusion

AID_DM is a simple yet essential global constant definition for ISOBUS data mask attribute management. By providing symbolic names for the background colour and soft key mask attribute IDs, it enhances code clarity and reduces the risk of hard-coded numbers. It is a foundational building block for ISOBUS applications within 4diac.

AID_EIA

• – ◦ • – ◦ • – ◦ •

Introduction

AID_EIA is a global constants block defined for the ISOBUS (ISO 11783) protocol, specifically addressing the Extended Input Attributes object. It provides a single constant that defines the validation type for the extended input attribute data set. This constant is used to indicate how the validity of characters in the extended input attributes is determined, enabling compliant implementations to interpret the attribute correctly.

Interface Structure

Since AID_EIA is a global constants block, it does not possess any event, data, or adapter interfaces. It encapsulates constant values that can be referenced by other function blocks or applications within a 4diac project.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The block defines the constant `VALTYPE` which is of type `USINT` and is initialized to the value `USINT#1`. According to the comment, this constant represents the validation type for extended input attributes:

- `0` means valid characters are listed.
- `1` means invalid characters are listed.

By setting the default value to `1`, the block indicates that the extended input attribute data uses a list of invalid characters for validation. This value is intended to be used in the context of the ISOBUS Extended Input Attributes object, where a validation type of `1` corresponds to the specification that the character set is defined by an exclusion list.

The constant is declared as a global constant within the package `isobus::UT::Q::const::AID`, making it accessible to any FB or application that needs to reference the validation type for extended input attributes.

Technical Features

- **Type**: `USINT` (Unsigned Short Integer, 8-bit).
- **Initial Value**: `USINT#1`.
- **Visibility**: Global constant, accessible from any FB.
- **Purpose**: Specifies the validate type for ISOBUS extended input attributes.
- **Standard Compliance**: Aligned with the ISOBUS (ISO 11783) standard, particularly the Extended Input Attributes object definition.

State Overview

As a global constants block, AID_EIA has no states or state transition logic. The value of `VALTYPE` is fixed at initialization and remains constant throughout the runtime of the system. No dynamic behavior is associated with this block.

Application Scenarios

AID_EIA is typically used in ISOBUS-based agricultural machinery or implement control systems where the `Extended Input Attributes` object (AID) is implemented. The constant can be referenced in any function block that needs to interpret or generate extended input attribute data, ensuring consistent handling of character validation across the system. For example, a display module or a control unit may use this constant to determine whether to treat the attribute data as a list of valid or invalid characters when processing user input or configuration data.

Comparison with Similar Blocks

In the ISOBUS standard, similar global constants exist for other object types (e.g., Input Attributes, Output Attributes). While those constants may define different validation mechanisms, AID_EIA specifically targets the Extended Input Attributes object. Unlike function blocks that process data, AID_EIA provides a static configuration parameter, making it comparable to other global constant definitions in 4diac such as `AID_VAL` or `AID_FMT`, but focused solely on the validation type.

Conclusion

AID_EIA is a simple yet essential global constants block that encapsulates the validation type for ISOBUS Extended Input Attributes. By providing a standardized constant value, it eliminates the risk of hard-coded values scattered across different function blocks and ensures compliance with the ISOBUS specification. Its integration into 4diac projects is straightforward, and it serves as a reference for any FB that handles extended input attribute data.

AID_EX

• − ◦ • − ◦ • − ◦ •

Introduction

The AID_EX global constant block defines a set of external object attribute identifiers used within the ISOBUS (ISO 11783) communication framework. These constants serve as standardized references for accessing and managing external object attributes in agricultural equipment control systems. The block provides a compact and reusable collection of attribute IDs, ensuring consistency across distributed application components.

Interface Structure

As a global constant block, AID_EX exposes no event inputs, event outputs, data inputs, or adapters. It solely provides constant data values that can be referenced by other function blocks or applications. The constants are defined as USINT (unsigned short integer) type with predefined initial values.

Data Outputs

Constant Name	Type	Initial Value	Description
OPTIONS	USINT	USINT#1	Bitmask: Bit 0 = Enabled (allows referenced NAME WS to reference listed objects).
NAME_0	USINT	USINT#2	External object attribute ID for NAME_0.
NAME_1	USINT	USINT#3	External object attribute ID for NAME_1.

No Event Inputs, Event Outputs, Data Inputs, or Adapters are present.

Functionality

The AID_EX block encapsulates three constant attribute IDs essential for ISOBUS external object handling. The OPTIONS constant is a bitmask that controls accessibility of referenced object lists, while NAME_0 and NAME_1 provide unique identifiers for two distinct attribute names. These constants are designed to be used in conjunction with the UT::Q package for unified tractor-implement communication, enabling consistent attribute addressing across different software modules.

Technical Features

- **Data Type:** All constants are of type USINT (8-bit unsigned integer).
- **Predefined Values:** Constants are initialized with fixed values to ensure uniform behavior.
- **Package Association:** Belongs to the package isobus::UT::Q::const::AID, facilitating modular integration.
- **Static Nature:** The block is completely static and has no runtime execution behavior; it only provides compile-time constants.

State Overview

Since AID_EX is a constant definition block, it maintains no runtime state. All values are predetermined and immutable, making the block inherently stateless and safe for concurrent use in distributed systems.

Application Scenarios

- **ISOBUS Implement Control:** Used to reference external object attributes in implements that follow the ISOBUS standard.
- **Configuration Management:** Provides a centralized location for attribute IDs, simplifying updates and maintenance.
- **Inter-module Communication:** Ensures consistent attribute identification across multiple function blocks within an application.

Comparison with Similar Blocks

Unlike typical function blocks (FBs) that process events and data, AID_EX serves as a static data repository. It is similar to a constant library or parameter set but without any dynamic behavior. Compared to other global constant blocks, AID_EX is specialized for ISOBUS attribute IDs, offering a focused set of constants with clear documentation. Its simplicity contrasts with more complex FB types that contain multiple interfaces and execution logic.

Conclusion

The AID_EX global constant block provides a straightforward and standardized way to define external object attribute IDs for ISOBUS applications. Its static, typed constants ensure consistency and reduce the risk of misidentification. By centralizing these IDs, the block promotes maintainability and interoperability within agricultural control systems.

AID_EXOP

• − ◦ • − ◦ • − ◦ •

Introduction

AID_EXOP is a global constant definition used in the IEC 61499 (4diac-ide) environment. It defines attribute identifiers for the **External Object Pointer** object type within the ISOBUS (ISO 11783) protocol stack. These constants are used to reference specific attributes of external object pointers in a standardized, human-readable way, replacing raw numeric IDs in application code.

This definition is part of a package named isobus::UT::Q::const::AID and is intended to be used as a shared set of constants across multiple function blocks or applications. The constant values are defined as USINT (unsigned short integer) literals.

Interface Structure

Since AID_EXOP is a global constant definition, it does **not** expose any event, data, or adapter interfaces. It is not a function block or subapplication instance; it merely provides constant values that can be referenced by other FB types.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None. The variables defined here are compile-time constants, not runtime data inputs.

Data Outputs

None.

Adapters

None.

Global Constants

The following table lists all constants defined by AID_EXOP:

Constant Name	Type	Value	Description
DEF_OBJ_ID	USINT	USINT#1	Default Object ID to display if an external ID is not valid or is NULL. This corresponds to the attribute AID_EXOP_DEF_OBJ_ID.
EXT_REF_NAME_ID	USINT	USINT#2	Attribute ID for the external reference name. Corresponds to AID_EXOP_EXT_REF_NAME_ID.
EXT_OBJ	USINT	USINT#3	Attribute ID for the external object itself. Corresponds to AID_EXOP_EXT_OBJ.

These constants are declared as VAR_GLOBAL CONSTANT and are therefore immutable and accessible from any FB or function within the same project that includes this definition.

Functionality

The primary purpose of AID_EXOP is to provide a consistent set of attribute ID constants for the ISOBUS **External Object Pointer** object. In ISOBUS implementations, object pools and virtual terminals exchange object attribute data using numeric identifiers. Using named constants improves code readability and maintainability, and reduces the risk of typographical errors.

When used in conjunction with other ISOBUS-related function blocks (e.g., those handling object pool management or VT communication), these constants are referenced to read or modify external object pointer attributes, such as the default object ID, reference name, or the external object handle.

Technical Features

- **Type:** `USINT` (unsigned 8-bit integer), suitable for attribute IDs in the ISOBUS standard (range 0–255).
- **Initialization:** Each constant has an explicit initializer using the `USINT#` literal syntax.
- **Persistence:** Declared as `VAR_GLOBAL CONSTANT`, meaning the values are fixed at compile time and cannot be changed at runtime.
- **Package:** Located in `isobus::UT::Q::const::AID`, indicating its association with ISOBUS utility and constant definitions.
- **Compatibility:** The constants follow the naming and numbering conventions of the ISOBUS AID (Attribute ID) specification, allowing direct mapping to protocol-level identifiers.

State Overview

This definition has no runtime state. It is a purely static set of constants. There are no state transitions, initialization procedures, or execution steps associated with it. The values are available as soon as the containing application or library is loaded.

Application Scenarios

`AID_EXOP` is typically used in ISOBUS applications where external object pointers are managed:

- **Virtual Terminal (VT) implementations:** Referencing the default object ID or external object attributes when building or updating object pools.
- **Task controllers:** Handling external objects like implements or sensors, where pointer attributes need to be queried or set.
- **Diagnostic and status display:** Determining which object to display when an external ID is invalid or NULL.
- **Protocol adaptation:** Mapping internal data structures to ISOBUS attribute IDs for communication with other ISOBUS nodes.

Example usage in ST (Structured Text):

```
// Use the default object ID constant
myObject := DEF_OBJ_ID;
```

Comparison with Similar Blocks

Similar global constant definitions in ISOBUS contexts are typically created for other object types, such as `AID_OBJ`, `AID_OP`, or `AID_DP`. These follow the same pattern:

- `AID_EXOP` – External Object Pointer attribute IDs
- `AID_OP` – Object Pointer attribute IDs
- `AID_DP` – Device Process Data attribute IDs

The main difference between these definitions is the set of attributes they contain and their numeric values. `AID_EXOP` specifically addresses the three attributes listed above, whereas other definitions may have more entries. All such constant sets serve the same purpose: providing named references to numeric protocol identifiers, improving code clarity and reducing maintenance effort.

Conclusion

`AID_EXOP` is a compact but essential set of global constants for ISOBUS external object pointer attribute IDs. While it has no executable interface, it plays a key role in making PLC and industrial automation code more readable and robust. By using these named constants instead of raw numbers, developers can avoid errors and ensure compliance with the ISOBUS protocol specification. Its integration into 4diac-based projects is straightforward, and it forms part of a broader library of protocol-related constants.

AID_EXRN

• – ◦ • – ◦ • – ◦ •

Introduction

AID_EXRN is an IEC 61499 global constants container defined for the 4diac-IDE. It provides symbolic constant values for the object attribute identifiers used by the external reference name object. This XML

type is not a regular executable function block; it is a global constants class intended to be shared across an application or project. It is part of the package `isobus::UT::Q::const::AID`.

Interface Structure

Event Inputs
None. AID_EXRN does not define any event inputs.

Event Outputs
None. AID_EXRN does not define any event outputs.

Data Inputs
None. AID_EXRN does not define any data inputs.

Data Outputs
None. AID_EXRN does not define any data outputs.

Adapters
None. AID_EXRN does not define any adapters.

Functionality
AID_EXRN defines a set of constant attribute identifiers for the external reference name object. These constants allow algorithms to use meaningful symbolic names instead of raw numeric values.

The following constants are declared:

Constant	Value	Meaning
`OPTIONS`	`USINT#1`	Attribute ID 1: `AID_EXRN_OPTIONS`; bitmask with bit 0 = Enabled, allowing external reference by NAME WS.
`NAME_0`	`USINT#2`	Attribute ID 2: `AID_EXRN_NAME_0`.
`NAME_1`	`USINT#3`	Attribute ID 3: `AID_EXRN_NAME_1`.

These values are fixed at global constant declaration time and can be referenced from other IEC 61499 elements such as algorithms and expressions.

Technical Features
- The XML root element is `GlobalConstants`, not a regular function block type.
- All declared constants use the `USINT` data type.
- The constants are declared with initial values and are globally available.
- The package name is `isobus::UT::Q::const::AID`.
- The compiler package is also `isobus::UT::Q::const::AID`.
- The constants are intended for ISOBUS-related use cases, specifically for external reference name attribute handling.

State Overview
AID_EXRN has no execution state machine. Because it is a global constants class, it does not contain `START`, `STOP`, or any internal algorithmic states. It is passive and simply provides constant data to the surrounding application.

Application Scenarios
AID_EXRN is useful in applications that need to interact with ISOBUS external reference name objects. Typical use cases include:

- Setting or reading attributes of an external reference name object.
- Checking or configuring the `OPTIONS` bitmask to enable external references by NAME WS.
- Writing maintainable algorithms that use symbolic constants instead of hard-coded attribute IDs.

Comparison with Similar Blocks
A regular function block typically has event inputs, event outputs, data inputs, data outputs, and internal state. AID_EXRN has none of these. It is comparable to other IEC 61499 global constants groups, but it

is specialized for external reference name attribute IDs. This makes it similar to a shared enumeration or constant header in conventional programming languages.

Conclusion

AID_EXRN is a small, focused global constants definition that centralizes the attribute IDs used by the external reference name object. It improves readability, maintainability, and consistency in ISOBUS-related 4diac-IDE applications.

AID_FA_FILL

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_FA_FILL` global constant set defines attribute identifiers for the **Fill Attributes Object** within the ISO 11783 (ISOBUS) protocol. These constants are used to reference specific fill properties inside a fill attributes object, enabling consistent and readable access to fill type, colour, and pattern settings across ISOBUS applications. The constants are defined as `USINT` (unsigned short integer) values and are intended to be used with the ISOBUS virtual terminal implementation.

Interface Structure

Since `AID_FA_FILL` is a global constants block, it does not contain event or data inputs/outputs in the conventional function block sense. Instead, it exposes three constant values that can be referenced globally within your IEC 61499 application.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

The following table summarises the constants provided by this global constants block:

Constant Name	Type	Value	Description
`FILL_TYPE`	`USINT`	`1`	Attribute ID 1: Fill type. Values: 0 = no fill, 1 = fill with line colour, 2 = fill with specified colour, 3 = fill with pattern.
`COLOUR`	`USINT`	`2`	Attribute ID 2: Colour index used for filling.
`PATTERN`	`USINT`	`3`	Attribute ID 3: Pattern index used for filling.

These constants are defined as `CONSTANT` in the `VAR_GLOBAL_CONSTANT` section, meaning they are immutable and globally accessible.

Functionality

The constants in `AID_FA_FILL` are used to address fields within an ISOBUS Fill Attributes object. In ISOBUS, objects like fill attributes are stored as records with indexed elements. By using these predefined attribute IDs, application logic can reliably refer to the correct data element without hard-coding numeric values, improving readability and maintainability.

For example, to set the fill type of a fill attributes object, an application would use `AID_FA_FILL.FILL_TYPE` as the attribute identifier, and the corresponding value would be one of the allowed fill type codes.

Technical Features

- **Global Constants**: The definitions are located in a global constants block, making them available to all function blocks and resources in the 4diac IDE project.
- **Data Type**: All constants are of type `USINT` (unsigned 8-bit integer), which is sufficient for the small enumeration values and colour indices used.
- **Constant Declaration**: The `CONSTANT` keyword ensures that these values cannot be modified at runtime, providing safety and preventing accidental changes.
- **Standard Compliance**: The constants follow the ISOBUS standard for attribute IDs, specifically for the Fill Attributes Object.

State Overview

This global constants block has no internal states or state machines; it is purely a declarative element.

Application Scenarios

The `AID_FA_FILL` constants are typically used in applications that interact with the ISOBUS Virtual Terminal (VT) or implement ISOBUS-compliant functions. Example use cases include:

- Setting up a fill attributes object in a VT display.
- Modifying the fill type of a graphic element (e.g., a bar graph or gauge) based on user input or process conditions.
- Writing colour or pattern selections to a VT object.

Comparison with Similar Blocks

There are other global constant blocks in the ISOBUS library, such as `AID_FA_LINE` or `AID_FA_TEXT`, which define attribute IDs for line and text attributes respectively. These follow the same pattern: a set of `USINT` constants representing attribute indices. Compared to those, `AID_FA_FILL` focuses exclusively on fill-related attributes, providing a clear separation of concerns.

Conclusion

The `AID_FA_FILL` global constant block provides a clean, standards-based way to reference fill attribute IDs in ISOBUS applications. By using these constants, developers can avoid magic numbers, improve code clarity, and ensure alignment with the ISOBUS specification. It is a fundamental building block for any function block that needs to manage fill attributes on a virtual terminal.

AID_FA_FONT

• − ◦ • − ◦ • − ◦ •

Introduction

This global constant type defines the attribute identifiers (AID) used for the **Font Attributes Object** within the ISOBUS Universal Terminal (UT) protocol. The constants map to specific attribute indices that are used to access and modify font properties on a virtual terminal. These IDs are essential for any IEC 61499 application that communicates with ISOBUS compliant displays.

Interface Structure

As a global constant type, `AID_FA_FONT` does not provide event or data inputs/outputs like function blocks. Instead, it exposes a set of named constants that can be referenced throughout the project. The constants are defined as unsigned short integers (USINT) and are listed below.

Event Inputs

Not applicable – this type does not process events.

Event Outputs

Not applicable – this type does not generate events.

Data Inputs

The type defines the following constants, which can be used directly in IEC 61499 logic:

Constant Name	Type	Value	Description
`COLOUR`	USINT	1	Attribute ID for the font colour index.

SIZE	USINT	2	Attribute ID for the font size.
FONT_TYPE	USINT	3	Attribute ID for the font type (e.g., sans-serif, serif).
STYLE	USINT	4	Attribute ID for the font style – a bitmask (see details below).

Data Outputs

Not applicable – constants are not outputs.

Adapters

Not applicable – this type does not use adapters.

Functionality

The AID_FA_FONT constants are used to address specific attributes of the Font Attributes Object in the ISOBUS object dictionary. When an application needs to read or write a font property, it passes the corresponding constant as the attribute identifier in the service request. This ensures correct and consistent access across different implementations.

The **STYLE** attribute is a bitmask with the following meanings per bit (bit 0 is least significant):

Bit	Meaning
0	Bold
1	Crossed Out
2	Underlined
3	Italic
4	Inverted
5	Flashing inverted
6	Flashing hidden
7	Proportional rendering

Technical Features

- All constants are of type USINT (unsigned 8-bit integer).
- Values are predefined and must not be changed at runtime.
- The constants are grouped under the package isobus::UT::Q::const::AID.
- They follow the ISOBUS 61499-1 standard for object attributes.
- The type is intended to be used with the ISOBUS UT communication stack.

State Overview

Not applicable – the type does not contain state machines or dynamic behavior.

Application Scenarios

- **Setting font properties on a Virtual Terminal** – When a client application wants to change the font of a label or text object, it uses the appropriate constant (e.g., AID_FA_FONT::COLOUR) to specify the attribute to modify.
- **Reading font attributes** – To retrieve the current font settings, the same attribute IDs are used in read requests.
- **Building ISOBUS diagnostic tools** – Developers can use the well-defined constants to implement testing or monitoring applications for UT communication.

Comparison with Similar Blocks

Unlike function blocks that process data or events, AID_FA_FONT is a read-only constant type. Similar global constant types exist for other object attribute IDs (e.g., AID_FA_* for different object types) and are part of the same isobus::UT::Q::const package. The distinguishing feature of this type is its specific set of attribute identifiers for font objects, ensuring standardised naming and values.

Conclusion

AID_FA_FONT provides a standardised and maintainable way to reference font attribute IDs in ISOBUS UT applications. By using these constants, developers avoid hard-coded magic numbers, reduce errors, and improve code clarity. The type is an essential building block for any IEC 61499 application that interacts with ISOBUS virtual terminals.

AID_GC

• − ◦ • − ◦ • − ◦ •

Introduction

The AID_GC definition provides a standardized set of global constants used to identify attribute IDs for Graphic Context (GC) objects in ISOBUS-based agricultural systems (ISO 11783-6). These constants simplify referencing common graphical parameters—such as viewport dimensions, colours, and formatting—within control applications and human-machine interfaces. The values are defined as unsigned short integers (USINT) and are intended to be used as attribute identifiers when configuring graphic context objects.

Interface Structure

Event Inputs

None – this is not a function block with event interfaces. It constitutes a set of global constant declarations.

Event Outputs

None – no event outputs are defined.

Data Inputs

None – the definition does not provide any data input ports. All values are statically defined constants.

Data Outputs

None – no data output ports are present.

Adapters

None – no adapters are associated with this definition.

Functionality

The primary purpose of AID_GC is to offer a readable and consistent way to refer to attribute IDs of ISOBUS graphic context objects. Each named constant corresponds to a numeric attribute ID, enabling developers to avoid hard-coded numbers in their code. The constants cover:

- Viewport properties (width, height, position, zoom)
- Canvas dimensions
- Cursor coordinates
- Colour settings (foreground, background, transparency)
- Referencing font, line, and fill attribute objects
- Format and option flags

The following table lists all defined constants:

Constant Name	Type	Initial Value	Description
VP_WIDTH	USINT	USINT#1	Viewport width attribute ID
VP_HEIGHT	USINT	USINT#2	Viewport height attribute ID
VP_X	USINT	USINT#3	Viewport X coordinate attribute ID
VP_Y	USINT	USINT#4	Viewport Y coordinate attribute ID
CANVAS_WIDTH	USINT	USINT#5	Canvas width attribute ID
CANVAS_HEIGHT	USINT	USINT#6	Canvas height attribute ID
VP_ZOOM	USINT	USINT#7	Viewport zoom attribute ID

GR_CURSOR_X	USINT	USINT#8	Graphic cursor X coordinate attribute ID
GR_CURSOR_Y	USINT	USINT#9	Graphic cursor Y coordinate attribute ID
FG_COLOUR	USINT	USINT#10	Foreground colour index attribute ID
BACKGROUND_COLOUR	USINT	USINT#11	Background colour index attribute ID
FONT_ATT	USINT	USINT#12	Object ID reference for a Font Attributes object
LINE_ATT	USINT	USINT#13	Object ID reference for a Line Attributes object
FILL_ATT	USINT	USINT#14	Object ID reference for a Fill Attributes object
FORMAT	USINT	USINT#15	Canvas type: 0 = Monochrome, 1 = 4-bit colour, 2 = 8-bit colour
OPTIONS	USINT	USINT#16	Bitmask: Bit 0 = Transparency (0 = opaque, 1 = transparent), Bit 1 = Colour mode
TRANS_COLOUR	USINT	USINT#17	Transparency colour attribute ID

Technical Features

- **Data Type**: All constants are of type `USINT` (unsigned short integer).
- **Initialization**: Each constant is initialized with a distinct numeric value, directly mapping to the attribute ID it represents.
- **Package Association**: The constants are defined in the package `isobus::UT::Q::const::AID`, providing a clear namespace for ISOBUS constant definitions.
- **Standards Compliance**: The definitions follow the IEC 61499-1 standard for global constants, as indicated in the identification section.
- **Documentation**: Each constant is accompanied by a comment that includes its numeric ID and, where applicable, a description of its purpose (e.g., `BACKGROUND_COLOUR` indicates it stores a colour index; `FORMAT` explains the canvas colour depth).

State Overview

This definition does not introduce any stateful behavior. The constants are immutable and purely declarative. No state transitions, timers, or conditional logic are present.

Application Scenarios

- **Graphic Context Configuration**: In ISOBUS terminals, these constants are used to set or query attributes of graphic context objects, such as defining the size of the drawing area or selecting colour modes.
- **HMI Development**: Developers can reference these constants to avoid magic numbers when programming user interface elements, improving code readability and maintainability.
- **Cross-Platform Consistency**: By using standardized attribute IDs, applications can ensure consistent behavior across different ISOBUS implementations.
- **Protocol Integration**: The constants align with the ISOBUS protocol's attribute numbering, simplifying integration with low-level communication layers.

Comparison with Similar Blocks

While no direct function blocks are compared here, `AID_GC` can be considered analogous to other constant sets used in ISOBUS, such as attribute IDs for object pools (`AID_OP`) or working sets (`AID_WS`). In comparison:

- **Readability**: Named constants provide a clearer alternative to raw numeric IDs.
- **Maintainability**: Centralizing attribute IDs in one location reduces the risk of errors when the protocol evolves.
- **Portability**: Following the official ISOBUS numbering ensures that code is interoperable across compliant devices.

Conclusion

AID_GC is a vital set of global constants for ISOBUS graphic context handling. It offers a structured, documented, and standardized way to reference critical graphical attributes, reducing development effort and enhancing reliability in agricultural automation systems. Its integration into the package isobus::UT::Q::const::AID underscores its role as a foundational building block for ISOBUS-based applications.

AID_GD

• – ◦ • – ◦ • – ◦ •

Introduction

The AID_GD global constants library defines attribute identifiers for graphic data (PNG) objects. It is used within the ISO-bus (ISOBUS) ecosystem to identify and manage graphic data attributes, specifically for PNG image encoding. The constants are defined as global constant variables, providing a single, standardized reference for graphic data attributes in accordance with ISO 11783-14 (ISOBUS) conventions.

This component is part of the isobus::UT::Q::const::AID package and is intended to be used by function blocks and other modules that need to reference graphic data attribute identifiers. It contains a single constant, FORMAT, which specifies the graphic type. The initial value USINT#1 indicates the PNG format, restricted to 32-bit RGBA maximum.

Interface Structure

Since AID_GD is a global constants library, it does not provide event, data, or adapter interfaces. It defines only constant values accessible globally.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None. The constant value FORMAT is a global constant, not a data input.

Data Outputs

None.

Adapters

None.

Functionality

The primary purpose of AID_GD is to define a global constant, FORMAT, that stores the graphic data format identifier. This constant is used to indicate the encoding type of graphic data (e.g., PNG) within ISOBUS applications. By centralizing this value, consistency and maintainability are ensured across all modules that reference graphic data attributes.

The constant is defined as:

- **FORMAT** (USINT, initial value USINT#1) – Specifies the graphic type. The value 1 corresponds to the PNG format, as per the comment: *"1: AID_GD_FORMAT - Graphic type: 0 = PNG, restricted to 32bit RGBA maximum."* Note: The comment appears to have a typo (0 vs 1), but the actual initial value is 1, and the intended meaning is that PNG is the graphic type.

Technical Features

- **Global Constant** – The constant is declared in a global constants block, making it accessible throughout the entire application without requiring instantiation.
- **Data Type** – USINT (unsigned short integer, 8-bit) ensures a small and efficient representation.
- **Standard Compliance** – The package name isobus::UT::Q::const::AID indicates alignment with ISOBUS standardized attribute IDs.
- **Initialization** – The constant is initialized to USINT#1, which is the predefined value for the PNG format.

State Overview

As a constants library, AID_GD does not maintain any internal state. It consists solely of compile-time constants that are immutable during runtime. No state transitions or lifecycle management are involved.

Application Scenarios

Typical use cases for AID_GD include:

- **ISOBUS Object Pool Generation** – When constructing object pools for virtual terminals, graphic data (e.g., icons or images) must be associated with format identifiers. AID_GD provides the correct attribute ID to be referenced.
- **Graphic Data Handling** – Function blocks responsible for processing or displaying graphic data can use FORMAT to determine whether the incoming data is PNG or another supported format.
- **Inter-Module Coordination** – Since the constant is global, multiple blocks can access the same format definition without risk of inconsistency.

Comparison with Similar Blocks

Compared to other global constant libraries in the ISOBUS environment (e.g., attribute IDs for object types, color definitions), AID_GD is specifically focused on graphic data format attributes. While similar libraries might contain multiple constants for various attribute IDs, AID_GD currently exposes only one constant, FORMAT. This makes it lightweight and specialized. It does not compete with function blocks that perform processing logic; rather, it supplies a static configuration value for those blocks.

Conclusion

AID_GD is a minimalistic but essential global constants definition that standardizes the graphic data format attribute identifier for PNG objects in ISOBUS applications. By using this constant, developers ensure that all references to graphic data formats remain consistent and easily maintainable. Its simple design—a single immutable value—makes it a reliable and efficient building block for larger ISOBUS-based systems.

AID_IB

• – ◦ • – ◦ • – ◦ •

Introduction

The AID_IB type is a global constant definition that provides the standardized attribute identifiers for **Input Boolean objects** within the ISOBUS (ISO 11783) Universal Terminal (UT) protocol. These constants serve as semantic keys for accessing and manipulating the various attributes of an input boolean UI element. By centralizing these identifiers, the type ensures consistent and maintainable code when constructing or modifying ISOBUS object pools.

This definition is part of the isobus::UT::Q::const::AID package and adheres to the IEC 61499-1 standard for data representation.

Interface Structure

Since AID_IB is a global constant type rather than a traditional function block, it does not expose event or data ports for execution. Instead, it provides a set of immutable constant values that are referenced globally by application logic and service interface function blocks when handling input boolean objects.

Event Inputs

None. This type defines constants only and has no event-driven behaviour.

Event Outputs

None. No event generation occurs from this type.

Data Inputs

None in the conventional FB sense. The constants defined within this type serve as reference data values that are used as inputs to other function blocks or service sequences.

Data Outputs

None in the conventional FB sense. The constant values are accessed directly by name from any part of the application where the type is referenced.

Adapters

No adapters are provided. The constants are intended to be used as direct scalar references in ISOBUS-related processing blocks.

Functionality

The `AID_IB` constant set enumerates the six standard attribute identifiers defined for an Input Boolean object in the ISOBUS UT specification. Each constant maps an attribute name to its numeric identifier used in object pool definition and command payloads:

Constant Name	Value	Description
`BACKGROUND_COLOUR`	1	Index of the background colour used by the input boolean object.
`WIDTH`	2	Width of the object in pixels.
`FG_COLOUR`	3	Object ID of a Font Attributes object used to determine the font colour for the displayed text or symbol.
`VARIABLE_REF`	4	Object ID of a Number Variable object to which the input value is bound. If NULL, the value is stored directly in the object.
`VALUE`	5	Current value of the input field: `0` represents FALSE, any value greater than zero represents TRUE.
`ENABLED`	6	Operational state of the object: `0` = disabled, `1` = enabled.

Use of these constants avoids hard-coded magic numbers in application code, improves readability, and ensures that attribute references remain aligned with the ISOBUS specification even if identifiers change in future revisions.

Technical Features

- **Data type:** All constants are declared as `USINT` (unsigned short integer, 8-bit range 0–255), sufficient for all defined attribute identifiers.
- **Immutability:** All constants are declared as `CONSTANT`, preventing accidental modification at runtime.
- **Standard compliance:** The identifiers correspond to the attribute IDs defined by the ISO 11783-6 / ISOBUS UT standard for the Input Boolean object type.
- **Namespace packaging:** The constants reside in the `isobus::UT::Q::const::AID` package, allowing clean import and preventing naming collisions.
- **Inline documentation:** Each constant carries a descriptive comment and its corresponding numeric attribute ID for quick cross-referencing.
- **Direct initialisation:** Initial values are assigned using type-qualified literals (e.g., `USINT#1`), ensuring type safety.

State Overview

As a set of compile-time constants, `AID_IB` has no runtime state machine. The values are fixed at compile time and remain invariant for the lifetime of the application. The only relevant "state" is the numeric value of each attribute identifier, which must match the ISOBUS UT specification for correct behaviour.

Application Scenarios

Typical use cases for the `AID_IB` constants include:

- **Object pool generation:** When building an ISOBUS Virtual Terminal object pool, these constants are used as attribute selectors in service requests (e.g., setting the background colour of an input boolean button).
- **Runtime attribute modification:** Applications that need to dynamically change the appearance or behaviour of an input boolean object (for example, toggling its enabled state) reference `AID_IB.ENABLED` or `AID_IB.VALUE` to form the correct command payloads.
- **HMI configuration:** When configuring input fields for operator interaction, such as toggle switches or buttons that represent boolean settings, these identifiers route configuration data to the correct attribute.
- **Value binding:** The `VARIABLE_REF` constant is used to link an input boolean object to a Number Variable object, enabling bidirectional data exchange between the terminal and the implement controller.

Comparison with Similar Blocks

The ISOBUS UT specification defines attribute ID sets for several object types. `AID_IB` covers the Input Boolean object specifically. Comparable constant sets exist for:

- **Input Numeric (AID_IN):** Uses additional attributes such as decimal places, minimum/maximum value, and format for numeric input fields.
- **Output Boolean (AID_OB):** Shares attributes like `BACKGROUND_COLOUR`, `FG_COLOUR`, and `WIDTH` but adds attributes for flash state, line selection indication, and active state colour.
- **Object itself (AID_OBJ):** Represents generic object attributes common to all UT objects, such as position, size, and visibility.

Unlike those broader sets, `AID_IB` focuses exclusively on the boolean input semantics, and omits properties that are only relevant to output or numeric objects — keeping the constant list concise and purpose-specific.

Conclusion

The `AID_IB` global constant type provides a clean, standardised, and type-safe way to reference Input Boolean object attribute identifiers in ISOBUS Universal Terminal applications. Its use eliminates magic numbers, improves code readability and maintainability, and guarantees alignment with the relevant protocol specifications. By incorporating these constants into object pool generation and runtime control logic, developers can build robust and portable ISOBUS HMI components with minimal risk of misaddressing.

AID_IL

• − ◦ • − ◦ • − ◦ •

Introduction

AID_IL is a global constant group used in 4diac-IDE-based ISOBUS applications. It defines the attribute identifiers for an **Input List** object of an ISOBUS Universal Terminal (UT). The constant group is not a function block or adapter; it provides a named, reusable set of numeric attribute IDs that are used when constructing or manipulating an Input List object in an ISOBUS object pool.

By using the constants defined here, application code avoids “magic numbers” and becomes more readable and maintainable.

Interface Structure

AID_IL is a global constant list and therefore does not expose an IEC 61499 function block interface. It has no event inputs, event outputs, data inputs, data outputs, or adapters. Its interface consists solely of compile-time constant values.

Event Inputs

None. AID_IL is not an executable function block and defines no event inputs.

Event Outputs

None. AID_IL has no event outputs and cannot trigger further processing.

Data Inputs

None. The constants are global values and are not provided through a data input interface.

Data Outputs

None as a block interface. However, the defined constants can be read globally as `AID_IL.<CONSTANT_NAME>`, for example `AID_IL.WIDTH`.

Adapters

None. AID_IL defines no adapter sockets or plugs.

Functionality

AID_IL centralizes the attribute IDs of an ISOBUS UT **Input List** object. The following constants are defined:

Constant	Type	Value	Description

WIDTH	USINT	1	AID_IL_WIDTH – Width of the input list in pixels.
HEIGHT	USINT	2	AID_IL_HEIGHT – Height of the input list in pixels.
VARIABLE_REF	USINT	3	AID_IL_VARIABLE_REF – Object ID of a Number Variable object that is linked to the selected list item.
VALUE	USINT	4	AID_IL_VALUE – Selected list index, from 0 to 254, or 255 if no item is selected.
OPTIONS	USINT	5	AID_IL_OPTIONS – Bitmask for input list options. Bit 0 controls whether the object is enabled; bit 1 controls real-time editing.

All constants are declared as USINT, an unsigned short integer value in the range 0 to 255. They are defined globally in the constant package:
isobus::UT::Q::const::AID

Technical Features

- Provides five named constants for Input List object attributes.
- All constants use the USINT data type.
- The values correspond to the attribute IDs used by the ISOBUS Universal Terminal protocol.
- The constant group can be reused in multiple function blocks without duplicate definitions.
- The included OPTIONS bitmask supports two distinct configuration flags in a single attribute.
- Because the constants are compile-time constants, they do not consume runtime resources and cannot be changed accidentally during execution.

State Overview

AID_IL is not stateful. It defines fixed values that are resolved at compile time. There is no initialization procedure, no internal state machine, and no runtime behavior. The values remain constant for the entire lifetime of the application.

Application Scenarios

AID_IL is typically used when an ISOBUS UT object pool or a client function block needs to set or query attributes of an Input List object. For example:

- Setting the width and height of an Input List object.
- Linking the Input List to a Number Variable object that stores the selected item.
- Reading or setting the currently selected list index.
- Changing the enabled state or activating real-time editing through the options bitmask.

Using AID_IL.WIDTH instead of the literal value 1 makes the application code easier to understand and to migrate if the attribute numbering changes.

Comparison with Similar Blocks

AID_IL is not a function block, so it cannot be directly compared to blocks that process events or data. Instead, it serves a similar role to an enumeration or a set of preprocessor constants in other programming languages.

Compared to a typical constant function block that outputs a single value, AID_IL provides multiple related constants in one global namespace. Compared to an adapter, it does not define a communication interface. Its main advantage is that all Input List attribute IDs are collected in one place and can be used consistently across many applications.

Conclusion

AID_IL is a small but useful global constant group for ISOBUS Universal Terminal applications. It defines the attribute IDs for an Input List object and helps keep application code readable, consistent, and independent from hard-coded numeric IDs. Although it has no executable interface, it is a valuable library element in 4diac-IDE-based ISOBUS projects.

AID_IN

• – ◦ • – ◦ • – ◦ •

Introduction

AID_IN is a global constants container used in 4diac-ide for ISO 11783 / ISOBUS applications. It defines the numeric attribute IDs for an Input Number object on a Virtual Terminal. The provided XML declares named constants of type `USINT` with values from `1` to `15`, allowing the attribute IDs to be referenced by meaningful names instead of magic numbers.

Although the XML structure resembles an IEC 61499 type definition, AID_IN is not a function block, adapter, or subapplication. It is a compile-time constants definition intended for use in ISOBUS-related logic.

Interface Structure

The AID_IN element does not define an IEC 61499 function block interface. It contains no event inputs, event outputs, data inputs, data outputs, or adapters. Its complete interface is represented by the global constants described in the Functionality section.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

AID_IN provides the attribute identifiers for an ISOBUS Input Number object. Each constant represents a specific attribute ID that can be used when reading from or writing to an ISOBUS Virtual Terminal.

The following table lists all defined constants:

Constant	Value	Description
`WIDTH`	`USINT#1`	Width in pixels.
`HEIGHT`	`USINT#2`	Height in pixels.
`BACKGROUND_COLOUR`	`USINT#3`	Background colour index.
`FONT_ATT`	`USINT#4`	Object ID of a Font Attributes object.
`OPTIONS`	`USINT#5`	Bitmask: Bit 0 = Transparent, Bit 1 = Display leading zeros, Bit 2 = Display zero as blank, Bit 3 = Truncate decimals.
`VARIABLE_REF`	`USINT#6`	Object ID of a Variable object.
`MIN_VALUE`	`USINT#7`	Minimum value.
`MAX_VALUE`	`USINT#8`	Maximum value.
`OFFSET`	`USINT#9`	Offset.
`SCALE`	`USINT#10`	Scale.
`NUMB_DECIMALS`	`USINT#11`	Number of decimals.
`FORMAT`	`USINT#12`	Decimal display format. `0` means fixed format decimal, `1` means exponential format.
`JUSTIFICATION`	`USINT#13`	Justification. Bits 0-1 define horizontal alignment, bits 2-3 define vertical alignment.

VALUE	USINT#14	Current value.
OPTIONS_2	USINT#15	Bitmask: Bit 0 = Enabled, Bit 1 = Real-time editing.

Technical Features

- Declared as an XML `<GlobalConstants>` element.
- Defined in the package namespace `isobus::UT::Q::const::AID`.
- All constants use the `USINT` data type.
- Each constant has an initial value equal to its corresponding ISOBUS attribute ID.
- The constants are intended for use in ISOBUS Virtual Terminal object handling.
- The `OPTIONS`, `JUSTIFICATION`, and `OPTIONS_2` constants encode their settings as bitmasks.
- No runtime behavior, algorithm, or state logic is included.

State Overview

AID_IN has no state machine and no runtime state. It is a purely declarative element. All values are fixed at compile time and do not change during program execution.

Application Scenarios

AID_IN is useful in ISOBUS applications that need to configure or update Input Number objects on a Virtual Terminal. Typical scenarios include:

- Creating attribute lists for ISOBUS object updates.
- Reading or writing Input Number object properties.
- Improving code readability by using symbolic names such as `WIDTH` or `VALUE` instead of numeric attribute IDs.
- Supporting maintainable, reusable ISOBUS communication logic in 4diac-ide.

Comparison with Similar Blocks

AID_IN is not a function block and cannot be directly compared with execution blocks such as controllers or converters. It belongs to a family of constant containers used for different ISOBUS object types. Similar constants may exist for other object attributes, such as output numbers, bar graphs, or text objects. The main difference between such containers is the set of attribute IDs and the meanings associated with them.

Conclusion

AID_IN provides a clear and maintainable way to handle ISOBUS Input Number object attribute IDs in 4diac-ide. By replacing numeric literals with named constants, it helps avoid errors, improves readability, and supports consistent communication with ISOBUS Virtual Terminals.

AID_IP

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_IP` global constants type defines a set of constant values used within the ISOBUS (ISO 11783) utility layer for handling input attributes. Specifically, it provides the validation type constant that controls how input attribute validation is performed. This type is part of the `isobus::UT::Q::const::AID` package and is intended to be included globally in IEC 61499 applications that deal with ISO-bus device communication.

Interface Structure

Since `AID_IP` is a **global constants** definition, it does not possess a traditional event/data input or output interface. Instead, it exposes a global constant that can be accessed by any function block or application component. The following sections list the available constant as part of the interface.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

The following global constant is defined:

Constant Name	Data Type	Initial Value	Description
`VALIDATION_TYPE`	`USINT`	`USINT#1`	Specifies the validation type: `0` = valid characters are listed, `1` = invalid characters are listed.

Adapters

None.

Functionality

The sole purpose of `AID_IP` is to provide a standardized, globally accessible constant for the validation mechanism of input attribute strings. By setting `VALIDATION_TYPE` to either `0` or `1`, the system can adapt its parsing and validation logic for input attributes. This constant is typically used in conjunction with other ISOBUS utility function blocks to ensure consistent behavior across an application.

Technical Features

- **Single Global Constant**: Only one constant is defined, reducing complexity and potential runtime overhead.
- **USINT Data Type**: The constant is stored as an unsigned short integer, suitable for small numeric flags.
- **Pre-initialized**: The constant is pre-set to `1` (invalid-character listing) at compile time, ensuring immediate usability without external configuration.
- **Namespace**: Belongs to the `isobus::UT::Q::const::AID` package, providing a clear hierarchical structure for related constants.

State Overview

As a global constants type, `AID_IP` does not contain any state machines, execution states, or dynamic behavior. Its value is fixed and immutable during runtime.

Application Scenarios

- **ISOBUS Input Validation**: When implementing the ISO 11783 utility layer, `VALIDATION_TYPE` can be used to switch between two validation modes for input attribute strings.
- **Configuration-Free Systems**: Because the constant is pre-defined, developers can rely on a consistent validation strategy without manual setup.
- **Multi-Device Coordination**: Global constants help ensure that all parts of a distributed IEC 61499 application agree on the same validation rule.

Comparison with Similar Blocks

Unlike function blocks (FBs) that have explicit event and data interfaces, `AID_IP` is purely a data definition. It is similar to global variable lists in other PLC programming environments but with the additional benefit of type-safe, constant-qualified definitions. FBs that require validation logic can read this constant directly, avoiding duplicate or conflicting definitions.

Conclusion

The `AID_IP` global constants type fulfills a focused but essential role in ISOBUS-based automation systems. By providing a single, well-defined constant for validation type selection, it promotes code consistency and maintainability. Its simple structure and predetermined default value make it an unobtrusive yet valuable component for applications that handle ISO-bus input attributes.

AID_IS

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_IS` global constant set defines attribute identifiers used for **Input String objects** in the context of ISOBUS (ISO 11783) Universal Terminal (UT) applications. These constants allow standardized access

to the properties of an input string object, such as width, height, background colour, font attributes, and various behavioural options. They are defined as compile-time constants of type `USINT` and are intended to be referenced in function blocks or logic that interact with ISOBUS UT object pools.

Interface Structure

This element is a global constants set and does not conform to the traditional function block interface (no events, no data I/O, no adapters). However, for documentation purposes, the constants are listed under the data section.

Event Inputs

None.

Event Outputs

None.

Data Inputs

The following constants are provided and can be used as data sources in any function block logic:

Name	Type	Initial Value	Description
`WIDTH`	USINT	1	1: AID_IS_WIDTH – Width in pixels.
`HEIGHT`	USINT	2	2: AID_IS_HEIGHT – Height in pixels.
`BACKGROUND_COLOUR`	USINT	3	3: AID_IS_BACKGROUND_COLOUR – Background colour index.
`FONT_ATT`	USINT	4	4: AID_IS_FONT_ATT – Object ID of a Font Attributes object.
`INP_ATT`	USINT	5	5: AID_IS_INP_ATT – Object ID of an Input Attributes / Extended Input Attributes object.
`OPTIONS`	USINT	6	6: AID_IS_OPTIONS – Options bitmask.
`VARIABLE_REF`	USINT	7	7: AID_IS_VARIABLE_REF – Object ID of a String Variable object.
`JUSTIFICATION`	USINT	8	8: AID_IS_JUSTIFICATION – Justification: Bits 0-1 (Horizontal: 0=Left, 1=Middle, 2=Right), Bits 2-3 (Vertical: 0=Top, 1=Middle, 2=Bottom).
`ENABLED`	USINT	9	9: AID_IS_ENABLED – 0 = Disabled, 1 = Enabled.

Data Outputs

None.

Adapters

None.

Functionality

The `AID_IS` constants act as predefined attribute identifiers for an input string object within an ISOBUS UT object pool. By using these constants, application logic can unambiguously refer to specific properties (e.g., width, height, background colour, font, options, reference to a string variable, justification, and enabled state) when reading or modifying the object's attributes via the standard ISOBUS service interfaces. The numeric values correspond to the attribute IDs defined in the ISOBUS standard for input string objects.

Technical Features

- **Type:** All constants are of type `USINT` (unsigned 8-bit integer).
- **Values:** The identifiers are sequential from 1 to 9, matching the official ISOBUS specification for input string object attributes.
- **Compile-time constants:** Declared as `VAR_GLOBAL CONSTANT`, they are resolved during compilation, offering performance benefits and guaranteeing immutability.

- **Namespace:** The constants belong to the package `isobus::UT::Q::const::AID`, ensuring a clear hierarchical organisation.

State Overview

This global constant set does not maintain any state. It provides static values that are available globally throughout the application. There are no runtime modifications, and the constants remain fixed for the entire execution.

Application Scenarios

The `AID_IS` constants are used in IEC 61499 applications that interact with ISOBUS Universal Terminal objects, particularly when a task requires:

- Configuring an input string object's dimensions or appearance.
- Changing the font or background colour.
- Setting or retrieving the associated string variable.
- Adjusting text justification or enabling/disabling the input field.
- Reading or writing options bitmask values.

For example, a function block controlling a text entry field might use `AID_IS.WIDTH` and `AID_IS.HEIGHT` to specify the size in its object pool creation request.

Comparison with Similar Blocks

While this element is not a function block, it is comparable to other **attribute ID constant sets** defined in the same package (e.g., `AID_IS` for input strings, or analogous sets for Output String, Number, etc.). The key difference is that each set defines attribute IDs for a specific object type, allowing developers to use mnemonic names instead of raw numeric values, reducing errors and improving code readability. Unlike function blocks, it provides no algorithmic functionality; it purely supplies constant data.

Conclusion

The `AID_IS` global constant set is a vital building block for ISOBUS UT applications, offering a clear and standardized way to reference attributes of input string objects. Its use enhances maintainability and compliance with the ISOBUS standard, making it an essential component for developers working with the 4diac-ide environment in agricultural or mobile machinery contexts.

AID_KG

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_KG` global constant set defines standardized attribute identifiers for ISO-bus key group objects. It is part of the `isobus::UT::Q::const` package and provides fixed numeric values that unambiguously reference specific object attributes, such as availability options and naming information. These constants simplify integration with ISO-bus communication stacks and ensure consistent code across applications.

Interface Structure

Event Inputs

None – this is a global constant definition, not a function block with event inputs.

Event Outputs

None – no event outputs are provided.

Data Inputs

None – the block does not accept any variable inputs; all values are instantiated as compile-time constants.

Data Outputs

None – the constants are accessible globally by their symbolic names (e.g., `OPTIONS`, `NAME`) and are not exchanged via formal output connections.

Adapters

None – no adapter interfaces are defined.

Functionality

The purpose of AID_KG is to centralize and standardize the numeric identifiers used for key group object attributes in ISO-bus systems. It declares two constants:

Constant Name	Type	Value	Description
OPTIONS	USINT	1	Bitmask that defines attribute availability: Bit 0 = Available, Bit 1 = Transparent.
NAME	USINT	2	Object ID of an output string or object pointer containing the attribute name.

These constants are used in conjunction with ISO-bus protocol operations to reference specific attributes without hard-coding numeric literals, improving readability and maintainability of code.

Technical Features

- **Compile-time constants**: Values are defined as GLOBALCONSTANTS in the IEC 61499 environment, ensuring that they are immutable and available throughout the application.
- **Type safety**: All constants are declared as USINT (unsigned short integer) to ensure compatibility with ISO-bus attribute fields.
- **Package organization**: The constants reside in the isobus::UT::Q::const::AID package, allowing for structured inclusion and avoiding name conflicts.
- **Simple integration**: The constants can be directly referenced in function blocks, algorithms, or other parts of the application without additional configuration.

State Overview

Since AID_KG defines static constants, it does not possess any runtime state. The values are fixed at compile time and do not change during execution. Therefore, there is no state machine, no initialization, and no internal variables to manage.

Application Scenarios

- **ISO-bus object modeling**: When implementing object dictionaries or key groups, OPTIONS and NAME are used to set or query attribute flags and naming data.
- **Protocol-related function blocks**: Blocks that interact with ISO-bus objects can use these constants to identify attributes during read/write operations.
- **Cross-component consistency**: By referencing the same constants in multiple FBs, the risk of attribute ID mismatches is eliminated, simplifying maintenance and diagnosis.

Comparison with Similar Blocks

- **vs. local constants**: Unlike local constants within an FB, AID_KG is globally accessible, avoiding duplication across multiple FBs and ensuring a single source of truth.
- **vs. enumeration types**: Enumerations provide symbolic names but are not necessarily bound to numeric values; AID_KG explicitly maps symbolic names to fixed integer IDs, which is required for wire-level protocol compatibility.
- **vs. configuration parameters**: Unlike configurable parameters, these constants are pre-defined and cannot be altered at runtime, which is appropriate for protocol-fixed identifiers.

Conclusion

AID_KG is a lightweight, standardized global constant set that defines essential attribute identifiers for ISO-bus key group objects. Its use promotes code clarity, reduces errors, and supports portable, maintainable applications within the 4diac-IDE environment. Though it contains no dynamic logic, it plays a fundamental role in ensuring consistent object attribute handling across a distributed system.

AID_LA

• − ◦ • − ◦ • − ◦ •

Introduction

AID_LA is a set of global constants that define the attribute identifiers for the ISO 11783 (ISOBUS) **Line Attributes Object**. These constants are used to address specific properties of a line object within a Virtual Terminal (VT) – primarily the line colour, line width, and line art. The constants are defined as USINT

(Unsigned Short Integer) values and are intended to be used as object attribute IDs when reading or writing line properties via the ISOBUS protocol.

The values correspond to the standard attribute IDs as specified in the ISOBUS VT standard:

- `1` – Line Colour
- `2` – Line Width
- `3` – Line Art (bit pattern for line style)

These constants simplify the development of VT applications by providing a human-readable symbolic name for each attribute ID, avoiding magic numbers in the application logic.

Interface Structure

Since AID_LA is a global constants container (not a function block or adapter), it does not have event inputs, event outputs, data inputs, data outputs, or adapters. Instead, it provides three global constants that can be referenced from any part of an IEC 61499 application.

Event Inputs

None – AID_LA defines only constants, no event inputs.

Event Outputs

None – AID_LA defines only constants, no event outputs.

Data Inputs

None – AID_LA defines only constants, no data inputs.

Data Outputs

None – AID_LA defines only constants, no data outputs.

Adapters

None – AID_LA is not an adapter type.

Functionality

The constant container `AID_LA` provides symbolic names for the three line attribute identifiers. The constants are defined as:

Constant Name	Type	Value	Description
`LINE_COLOUR`	`USINT`	`1`	Attribute ID for line colour (AID_LA_LINE_COLOUR)
`LINE_WIDTH`	`USINT`	`2`	Attribute ID for line width (AID_LA_LINE_WIDTH)
`LINE_ART`	`USINT`	`3`	Attribute ID for line art – a bit pattern where each bit represents a paintbrush spot (AID_LA_LINE_ART)

These constants are intended to be used in conjunction with VT line objects. For example, when setting the colour of a line, an application would use `AID_LA.LINE_COLOUR` as the attribute ID in the appropriate ISO 11783 command.

Technical Features

- **Data Type**: All constants are of type `USINT` (unsigned 8-bit integer), sufficient for the attribute IDs.
- **Namespace**: The constants are declared inside a package `isobus::UT::Q::const::AID`, ensuring a clear naming structure for reuse in larger projects.
- **Initialisation**: Each constant is explicitly initialised with a literal value (`USINT#1`, `USINT#2`, `USINT#3`), making the assignment unambiguous.
- **IEC 61499 Compatibility**: The constants are defined as `VAR_GLOBAL CONSTANT`, which is the IEC 61499 mechanism for global constants, accessible from any FB network.

State Overview

AID_LA is a static, immutable container. It has no internal states, no execution logic, and no lifecycle. The constants are available at compile time and remain unchanged during runtime.

Application Scenarios

Typical usage scenarios include:

- **VT Screen Design**: When building a VT page that contains a line object, the developer needs to specify the line colour, width, or style. Instead of using numeric codes directly, the developer can reference these constants for better readability and maintainability.
- **Runtime Attribute Configuration**: In a control application that dynamically changes a line's appearance (e.g., highlighting a boundary), the constant is used as the attribute ID in the `SetLineAttributes` or similar. This ensures the correct attribute is addressed.
- **Diagnostics and Logging**: When logging VT object properties, the symbolic name can be used for clarity.

Comparison with Similar Blocks

In the ISOBUS context, there are other attribute ID constant sets, for example:

- **AID_OO** – Object identifiers for object attributes.
- **AID_PO** – Point attributes.
- **AID_AO** – Auxiliary Object attributes.

AID_LA is specifically focused on line attributes. Compared to equivalent numeric literals, the constant set offers the advantage of:

- **Readability**: `AID_LA.LINE_COLOUR` is self-documenting.
- **Maintainability**: If the underlying ID ever changes (unlikely), the definition can be updated in a single place.
- **Type Safety**: Using `USINT` ensures the value fits the expected range for attribute IDs.

Conclusion

The `AID_LA` global constants provide a clean and standardised way to referencing line attribute IDs in ISOBUS VT applications. They are simple, well-defined, and align with the ISO 11783 standard. By using these constants, developers reduce the risk of errors and improve the clarity of their code, making the implementation of VT line objects more straightforward and robust.

AID_NV

• – ◦ • – ◦ • – ◦ •

Introduction

This global constant definition provides a symbolic name for the attribute ID of a number variable object in the ISO 11783 (ISOBUS) protocol. It defines the constant `VALUE`, which is assigned the numeric value `1`, corresponding to the `AID_NV_VALUE` – the "current value" attribute of a number variable. The constant is intended for use in IEC 61499 applications that interact with ISOBUS object dictionaries, improving code clarity and maintainability.

Interface Structure

As a global constants type, this element does not contain any event inputs, event outputs, data inputs, data outputs, or adapters. The following sections are therefore not applicable and are listed for completeness.

Event Inputs

None (not applicable for global constants).

Event Outputs

None (not applicable for global constants).

Data Inputs

None (not applicable for global constants).

Data Outputs

None (not applicable for global constants).

Adapters

None (not applicable for global constants).

Functionality

The primary purpose of this constant is to assign a human-readable identifier to the numeric attribute ID used in ISOBUS communication. Specifically, it defines the attribute ID for the "current value" of a number variable object. When developing IEC 61499 function blocks that read or write number variable values, this constant can be used to specify the target attribute without hard-coding the raw numeric value.

Technical Features

- **Type**: Global Constants (IEC 61499-1)
- **Constant Definition**:
  `VALUE : USINT := 1` (comment: `1: AID_NV_VALUE - Current value`)
- **Package/Namespace**: `isobus::UT::Q::const::AID` (set via `CompilerInfo`)
- **Standard Compliance**: IEC 61499-1
- **Version**: 1.0

State Overview

Since this is a static constant definition, it does not possess any runtime state. The value is fixed at compile time and remains constant throughout the execution of the application.

Application Scenarios

This constant is particularly useful in industrial automation applications that implement the ISOBUS protocol, such as agricultural machinery control systems. For example, when creating a function block to read the current numeric value from a tractor implement's object dictionary, the constant `AID_NV.VALUE` can be passed as the attribute ID parameter, ensuring that the correct attribute is addressed.

Comparison with Similar Blocks

In the ISOBUS context, global constants are defined for various object types (e.g., analog, digital, string) and their corresponding attributes. Compared to function blocks that handle data processing or communication, this constant is a static helper that simplifies code and reduces the risk of using incorrect attribute IDs. Similar constants would exist for other object attributes, each serving as a named reference to a specific numeric identifier.

Conclusion

The `AID_NV` global constant provides a well-defined, documented identifier for the "current value" attribute of number variable objects in ISOBUS. Its use promotes code readability, reduces maintenance overhead, and aligns with IEC 61499 best practices for separating constant definitions from application logic.

AID_OABG

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_OABG` global constant set provides symbolic names for the attribute identifiers (AIDs) of an **Output Arched Bar Graph (OABG)** object as specified by the ISOBUS Virtual Terminal (VT) standard. Instead of using magic numbers, applications can reference these constants to set or modify OABG attributes, improving code readability and maintainability. This constant set is defined in the package `isobus::UT::Q::const::AID` and is intended for use in IEC 61499 applications running on 4diac-ide.

Interface Structure

This global constant set does not define a standard function block interface with event or data inputs/outputs. It is a collection of compile-time constants that can be used anywhere a USINT attribute ID is required. The following sections are included for completeness, but they are not applicable to a constant set.

Event Inputs

None – no event inputs are defined.

Event Outputs

None – no event outputs are defined.

Data Inputs

None – no data inputs are defined.

Data Outputs

None – no data outputs are defined.

Adapters

None – no adapters are defined.

Functionality

Each constant in `AID_OABG` corresponds to a specific attribute of an OABG object, as defined by the ISOBUS standard. The following table lists all constants, their attribute identifiers, and their meaning:

Constant Name	Value	Attribute Description
`WIDTH`	1	Width in pixels
`HEIGHT`	2	Height in pixels
`COLOUR`	3	Colour index
`TARGET_LINE_COLOUR`	4	Target line colour
`OPTIONS`	5	Bitmask: Bit 0=Draw border, Bit 1=Draw target line, Bit 2=Undefined (set to 0), Bit 3=Bar graph type (0=filled, 1=single line), Bit 4=Deflection (0=anticlockwise, 1=clockwise)
`START_ANGLE`	6	Start angle in degrees
`END_ANGLE`	7	End angle in degrees
`BAR_GRAPH_WIDTH`	8	Bar graph width
`MIN_VALUE`	9	Minimum scaled value
`MAX_VALUE`	10	Maximum scaled value
`VARIABLE_REF`	11	Object ID of a Variable object
`TARGET_VAL_VAR_REF`	12	Object ID of a Variable object for target value
`TARGET_VALUE`	13	Target value
`VALUE`	14	Current value

These constants are all of type `USINT` and are initialized with their numeric attribute ID values. They are intended to be used in conjunction with VT commands such as those that set attribute values (e.g., `SetAttribute`).

Technical Features

- **Type:** All constants are `USINT` (unsigned 8-bit integer), matching the attribute ID type in the ISOBUS standard.
- **Package:** `isobus::UT::Q::const::AID`
- **Compliance:** The constants follow the attribute numbering defined in ISO 11783-6 for the OABG object.
- **Read-only:** The constants are `VAR_GLOBAL CONSTANT`, meaning they cannot be modified at runtime.
- **Use in 4diac-ide:** The constants can be directly referenced in ST or FBD code, e.g., `AID_OABG.WIDTH`.

State Overview

Not applicable. This constant set does not contain any stateful elements. It merely provides static definitions.

Application Scenarios

- **Creating an OABG object:** When creating a new output arched bar graph on an ISOBUS VT, the application must supply attribute IDs for properties such as width, height, colours, and angles. Using `AID_OABG.*` ensures correct values.
- **Modifying OABG attributes:** When sending a `SetAttribute` command to a VT, the attribute ID parameter can be set using these constants, avoiding hard-coded numbers.
- **Reading OABG status:** When querying the current value or target value, the attribute ID can be referenced symbolically.
- **Cross-platform consistency:** By using a central constant set, the same attribute IDs are guaranteed across different parts of an application, reducing the risk of inconsistency.

Comparison with Similar Blocks

Similar global constant sets exist for other VT objects, such as `AID_OARG` (Output Arched Bar Graph) or `AID_OMSG` (Output Message), each defining attribute IDs for their respective object types. Compared to using raw numeric literals, these constant sets improve code readability and maintainability. They also align with the ISOBUS standard, ensuring compatibility with various VT implementations. Unlike function blocks, they do not process data or events, but they serve as essential building blocks in the development of IEC 61499 applications for agricultural machinery.

Conclusion

The `AID_OABG` global constant set is a simple yet valuable utility for developers working with ISOBUS Virtual Terminal objects. By providing symbolic names for all attribute IDs of an output arched bar graph, it eliminates magic numbers and promotes clear, maintainable application code. It is an integral part of the `isobus::UT::Q::const` package and should be used whenever OABG attributes are accessed in 4diac-ide projects.

AID_OE

• – ◦ • – ◦ • – ◦ •

Introduction

The **AID_OE** global constant group defines the attribute identifiers (AIDs) for **Output Ellipse** objects within the ISOBUS (ISO 11783) object pool. It is part of the `isobus::UT::Q::const::AID` package and provides a standardized set of constant values used to reference specific attributes of an ellipse object (e.g., line style, size, type, angles, fill properties). These constants are essential for correctly reading and writing object attributes when configuring or updating an ellipse on a Virtual Terminal (VT) in agricultural machinery.

Interface Structure

This global constant group does not possess an event or data interface typical of function blocks. Instead, it exposes a set of read-only constant values that can be used globally within the IEC 61499 application.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None (the constants are global, not interface-specific).

Adapters

None.

Functionality

The `AID_OE` constants represent the numeric attribute identifiers for an output ellipse object as defined by the ISOBUS standard. Each constant corresponds to a specific attribute that can be queried or modified via

the object pool services. The values are fixed and immutable, ensuring consistent communication between the VT and the implement controller.

The constants and their meanings are:

Constant Name	Value	Description
LINE_ATT	1	Object ID of a Line Attributes object.
WIDTH	2	Width in pixels.
HEIGHT	3	Height in pixels.
ELLIPSE_TYPE	4	Ellipse type: 0=Closed Ellipse, 1=Open Ellipse, 2=Closed Segment, 3=Closed Section.
START_ANGLE	5	Start angle of the ellipse (in degrees).
END_ANGLE	6	End angle of the ellipse (in degrees).
FILL_ATT	7	Object ID of a Fill Attributes object.

Technical Features

- **Data Type**: All constants are of type `USINT` (unsigned short integer, 8-bit).
- **Fixed Values**: The values are pre-assigned and cannot be changed at runtime.
- **Global Scope**: The constants are accessible from any FB or resource within the 4diac project, promoting code readability and maintainability.
- **Standard Compliance**: They follow the ISOBUS 11783 object pool attribute definitions for output ellipses.
- **Package Location**: Defined within the `isobus::UT::Q::const::AID` namespace, grouping related attribute constants.

State Overview

This global constant group has no state information. It is a static definition file – its values remain unchanged throughout the application lifetime. There are no operational states, transitions, or internal logic.

Application Scenarios

- **Object Pool Creation**: When building an ISOBUS object pool that includes ellipse objects, these constants are used to set or retrieve specific ellipse properties such as width, height, or type.
- **Dynamic Attribute Updates**: In a running VT application, if the implement controller needs to modify an ellipse's dimensions or appearance, it uses these AIDs as command identifiers in the `Change Object Attribute` service.
- **Consistency Across Projects**: By referencing these constants instead of hard-coded numbers, developers avoid errors and improve code clarity.
- **Interoperability**: Ensures that all ellipse-related attributes use standardized identifiers, facilitating communication between equipment from different manufacturers.

Comparison with Similar Blocks

Unlike function blocks that process data or events, `AID_OE` is a passive constant container. It does not execute logic, consume CPU time, or require instantiation. In comparison:

- **Function Blocks** (e.g., `ELLIPSE_CTRL`) actively manage and update ellipse objects, often using these AIDs internally.
- **Other Global Constant Groups** (e.g., `AID_OL` for output line objects) serve the same purpose but for different object types. `AID_OE` specifically targets ellipses, ensuring accurate attribute mapping.
- **Constants vs. Variables**: Constants are immutable, whereas variables can change. Using constants helps prevent accidental modification of critical attribute identifiers.

Conclusion

The `AID_OE` global constant group provides a reliable, standardised set of attribute identifiers for ISOBUS output ellipse objects. It simplifies object pool management, enhances code maintainability, and ensures compliance with international agricultural communication standards. By leveraging these constants,

developers can focus on application logic without worrying about numeric identifier collisions or inconsistencies.

AID_OL

• – ◦ • – ◦ • – ◦ •

Introduction

`AID_OL` is a 4diac-ide global constants definition, not a function block, adapter, or subapplication. It defines a set of named `USINT` constants used as attribute IDs for ISOBUS output line objects. The constant group is part of the package `isobus::UT::Q::const::AID` and is intended to make object pool definitions more readable and maintainable.

Interface Structure

`AID_OL` does not provide an IEC 61499 event/data interface. It is a global constant collection that can be referenced by name inside other IEC 61499 elements.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None. The constants are not runtime input signals; they are globally defined compile-time values.

Data Outputs

None.

Adapters

None.

Defined Constants

Constant	Type	Initial Value	Description
`LINE_ATT`	`USINT`	`1`	Object ID of a Line Attributes object.
`WIDTH`	`USINT`	`2`	Width in pixels.
`HEIGHT`	`USINT`	`3`	Height in pixels.
`LINE_DIR`	`USINT`	`4`	Line direction: `0` = top-left to bottom-right, `1` = bottom-left to top-right.

Functionality

The purpose of `AID_OL` is to centralize the numeric attribute IDs for ISOBUS output line objects. Instead of using hard-coded numbers in function block logic, applications can reference these symbolic constants. This improves readability and reduces the risk of errors when multiple parts of an application need the same attribute IDs.

The constant group contains four attributes that are relevant for configuring line objects in an ISOBUS Virtual Terminal object pool:

- `LINE_ATT` identifies the line attributes object.
- `WIDTH` and `HEIGHT` describe the line dimensions in pixels.
- `LINE_DIR` defines the drawing direction of the line.

Technical Features

- Defined as a `GlobalConstants` element in 4diac-ide.
- Uses the IEC 61499 `USINT` data type.
- Contains fixed initial values assigned at declaration.
- Packaged under `isobus::UT::Q::const::AID`.

- Can be reused across multiple FBs, subapplications, and systems.
- Provides symbolic names for numeric ISOBUS attribute IDs.

State Overview

Not applicable. `AID_OL` is a constant definition and does not contain state machines, execution states, or dynamic runtime behavior.

Application Scenarios

- Building ISOBUS object pools with output line objects.
- Setting line attributes such as width, height, and direction in a Virtual Terminal screen.
- Replacing magic numbers in IEC 61499 applications with meaningful symbolic constants.
- Sharing a common definition of output line attribute IDs across multiple function blocks or projects.

Comparison with Similar Blocks

`AID_OL` can be compared with other ISOBUS attribute ID constant groups, typically named `AID_*`. While function blocks and adapters define runtime behavior, `AID_OL` only provides static constant values. Its main advantage over inline numeric literals is centralization and consistency. Similar constant groups exist for other ISOBUS object types and provide the same style of symbolic access.

Conclusion

`AID_OL` is a compact global constants group for ISOBUS output line object attribute IDs. It defines four `USINT` constants in a named package, making them easy to reuse in 4diac-ide applications. Although it has no function block interface or runtime state, it contributes to cleaner, more maintainable ISOBUS object pool handling.

AID_OLA

• – ◦ • – ◦ • – ◦ •

Introduction

`AID_OLA` is a set of global constants used in ISOBUS (ISO 11783) applications to identify attribute IDs for **Output List Objects** (OLA). It defines numeric identifiers for common attributes of a list object, such as its width, height, variable reference, and selected value. These constants are intended to be used together with the ISOBUS Virtual Terminal (VT) protocol to avoid hard-coding attribute IDs in application code, making the code more readable and maintainable.

Interface Structure

As a **global constants** group, `AID_OLA` does not expose any event or data interface. It is not a function block, adapter, or subapplication. Instead, it provides a set of constant values that can be referenced throughout the application.

Event Inputs

Not applicable.

Event Outputs

Not applicable.

Data Inputs

Not applicable.

Data Outputs

Not applicable.

Adapters

Not applicable.

Functionality

The constants defined in `AID_OLA` correspond to the attribute identifiers used in the ISOBUS **Output List Object** (Object type 1, “Output List”). The following attributes are defined:

Constant	Value	Description
WIDTH	1	AID_OLA_WIDTH – Width of the output list in pixels.
HEIGHT	2	AID_OLA_HEIGHT – Height of the output list in pixels.
VARIABLE_REF	3	AID_OLA_VARIABLE_REF – Object ID of a Number Variable linked to the list.
VALUE	4	AID_OLA_VALUE – Selected list index (0 to 254), or 255 if no item is chosen.

These constants should be used when constructing or interpreting VT command messages that manipulate an output list object.

Technical Features

- **Data type:** All constants are of type USINT (unsigned short integer, 8-bit).
- **Range:** The values are fixed within the range 1–4, corresponding to the official ISOBUS attribute identifiers.
- **Package:** The constants are declared in the package isobus::UT::Q::const::AID.
- **Initialization:** Each constant is explicitly initialized with its numeric value using the USINT# prefix.
- **Compatibility:** Designed for use with 4diac IDE and ISOBUS-capable control systems.

State Overview

Since AID_OLA is a constant definition, no stateful behavior exists. The values are immutable at runtime and do not change during execution.

Application Scenarios

AID_OLA is typically used in applications that implement an ISOBUS Virtual Terminal client or a control function that interacts with a VT. For example:

- **Setting the size of an output list:** When creating or updating an output list object, the WIDTH and HEIGHT attributes must be transmitted with the correct IDs. Using AID_OLA.WIDTH and AID_OLA.HEIGHT avoids magic numbers.
- **Linking a variable:** The VARIABLE_REF constant is used when associating a number variable with the list object, enabling bidirectional data exchange.
- **Reading the user selection:** The VALUE constant identifies the attribute that holds the currently selected index. This allows the application to query the selected element from the VT.

Comparison with Similar Blocks

AID_OLA is one of many global constant sets defined for ISOBUS object attributes. Similar groups exist for other object types, such as AID_OUTPUT_STRING, AID_NUMBER_VARIABLE, AID_KEY, etc. The main advantage of using such constant groups is that they provide a single source of truth for attribute IDs, reducing errors and improving code clarity. Unlike function blocks, they do not have execution logic, but they are often used together with function blocks that implement VT protocol handling.

Conclusion

The AID_OLA global constants provide a clean, standardised way to reference Output List Object attribute IDs in ISOBUS applications. By using these named constants, developers can ensure correct and maintainable code when working with Virtual Terminal objects. Although it is not a functional block itself, it is an essential building block for ISOBUS communication logic.

AID_OLBG

• – ◦ • – ◦ • – ◦ •

Introduction

The AID_OLBG Global Constants definition provides a structured set of named constants that identify attribute IDs for an **Output Linear Bar Graph** object in ISOBUS-based communication systems (ISO 11783). These constants serve as a lookup table for object attribute identifiers used when configuring or updating linear bar graph elements on virtual terminals. The constants correspond to the attribute IDs defined in the ISOBUS Object Pool specification, enabling consistent and readable access to these identifiers in IEC 61499 applications.

Interface Structure

Since AID_OLBG is a global constant definition rather than a function block, it does not contain event inputs, event outputs, data inputs/outputs, or adapters. Instead, it exposes a set of compile-time constants (all of type USINT) that can be referenced throughout an application. The following list enumerates these constants and their meanings.

Data Constants

Constant Name	Type	Value	Description
WIDTH	USINT	1	Attribute ID for width in pixels (AID_OLBG_WIDTH).
HEIGHT	USINT	2	Attribute ID for height in pixels (AID_OLBG_HEIGHT).
COLOUR	USINT	3	Attribute ID for the colour index (AID_OLBG_COLOUR).
TARGET_LINE_COLOUR	USINT	4	Attribute ID for the target line colour (AID_OLBG_TARGET_LINE_COLOUR).
OPTIONS	USINT	5	Attribute ID for the options bitmask (AID_OLBG_OPTIONS).
NUMB_TICKS	USINT	6	Attribute ID for the number of ticks (AID_OLBG_NUMB_TICKS).
MIN_VALUE	USINT	7	Attribute ID for the minimum value (AID_OLBG_MIN_VALUE).
MAX_VALUE	USINT	8	Attribute ID for the maximum value (AID_OLBG_MAX_VALUE).
VARIABLE_REF	USINT	9	Attribute ID for the reference to a Variable object (AID_OLBG_VARIABLE_REF).
TARGET_VAL_VAR_REF	USINT	10	Attribute ID for the target value variable reference (AID_OLBG_TARGET_VAL_VAR_REF).
TARGET_VALUE	USINT	11	Attribute ID for the target value (AID_OLBG_TARGET_VALUE).
VALUE	USINT	12	Attribute ID for the current value (AID_OLBG_VALUE).

Functionality

The primary purpose of AID_OLBG is to provide a named, type-safe collection of ISOBUS attribute IDs for an output linear bar graph object. In ISOBUS object pool implementations, each visual object (such as a bar graph) has a set of configurable attributes, each identified by a numeric ID. The constants defined here map human-readable names to these numeric IDs, allowing application developers to:

- Refer to attribute IDs by meaningful names instead of raw numeric literals.
- Ensure consistency across multiple functions and blocks that interact with the same bar graph.
- Simplify maintenance by centralizing the attribute ID definitions in one location.

The underlying ISO 11783 specification defines these attributes as AID_OLBG_* IDs. This constant set is packaged under the isobus::UT::Q::const::AID namespace and is intended for use in ISOBUS virtual terminal implementations.

Technical Features

- **Data Type Consistency**: All constants are declared as USINT (unsigned short integer), aligning with the ISOBUS attribute ID data type.
- **Namespace Organization**: The constants reside in the package isobus::UT::Q::const::AID, preventing name collisions and improving modularity.
- **Global Scope**: Declared as VAR_GLOBAL CONSTANT, the identifiers are accessible from any resource or function block within the IEC 61499 application.
- **Compile-Time Values**: As constants, their values are resolved at compile time, enabling compiler optimizations and reducing runtime overhead.
- **Documentation via Comments**: Each constant is annotated with its corresponding ISOBUS attribute ID name and description.

State Overview

The `AID_OLBG` definition does not maintain runtime states or internal state machines. It is a static, constant collection that remains unchanged during execution. Its "state" is fixed at compile time and does not influence runtime behavior beyond providing constant values.

Application Scenarios

Typical use cases for `AID_OLBG` include:

- **ISOBUS Virtual Terminal Displays**: Configuring a linear bar graph object on a virtual terminal by setting its attributes (size, colour, range, ticks, etc.).
- **Object Pool Construction**: Building or modifying an ISOBUS object pool where bar graph attribute IDs must be explicitly referenced.
- **Command/Response Handling**: Interpreting or constructing command messages that update bar graph properties using attribute IDs.
- **Cross-Component Communication**: Passing attribute IDs between function blocks that manage virtual terminal objects, ensuring consistent referencing.

For example, when a function block needs to update the `MIN_VALUE` attribute of a bar graph, it can use the constant `AID_OLBG.MIN_VALUE` instead of the literal `7`.

Comparison with Similar Blocks

Within the same `isobus::UT::Q::const::AID` package, there may exist similar global constant definitions for other object types (e.g., output text, output meter, or input objects). In contrast to a function block that performs logic or data transformation, these constant definitions are pure data collections. Their main advantages include:

- **Clarity**: Named constants reduce the risk of using incorrect attribute IDs.
- **Maintainability**: Changes to attribute ID values (should the ISOBUS standard evolve) require a single edit.
- **Self-Documentation**: The declaration includes descriptive comments and initial values, making the mapping explicit.

Compared to using raw numeric literals within application logic, `AID_OLBG` ensures that all references are consistent and auditable.

Conclusion

The `AID_OLBG` global constants definition provides a clean, standardized mechanism for referencing output linear bar graph attribute IDs in ISOBUS applications. By centralizing these IDs as named constants, it promotes code readability, maintainability, and correctness. It is an essential component for developers working with ISOBUS virtual terminals, serving as a reliable reference layer between the IEC 61499 application logic and the ISO 11783 object model.

AID_OLRL

• − ◦ • − ◦ • − ◦ •

Introduction

The `AID_OLRL` global constant set defines a collection of attribute identifiers used in the context of object label reference lists within the ISOBUS UT (Universal Terminal) protocol. These constants serve as predefined numeric values that facilitate the identification and handling of labelled objects in agricultural machinery communication. This constant set is part of the `isobus::UT::Q::const::AID` namespace and provides a single, clearly named constant for use in application logic.

Interface Structure

As a global constant set, `AID_OLRL` does not expose events, data inputs/outputs, or adapters. Instead, it provides a static value that can be referenced by other function blocks or applications.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The AID_OLRL constant set defines a single constant:

Constant Name	Data Type	Initial Value	Description
NUMB_LABELLED_OBJ	USINT	USINT#1	Attribute ID for the number of labelled objects that follow in an object label reference list.

This constant is intended to be used when constructing or parsing object label reference lists, ensuring consistent identification of the "number of labelled objects" field across different implementations.

Technical Features

- **Data Type**: USINT (unsigned short integer), occupying 1 byte.
- **Initial Value**: 1, representing the standard attribute ID for the number of labelled objects.
- **Namespace**: isobus::UT::Q::const::AID (accessed as AID_OLRL.NUMB_LABELLED_OBJ).
- **Compliance**: Aligns with ISO 11783 (ISOBUS) standards for UT functional objects.

State Overview

Not applicable – this is a constant set and does not maintain any runtime state.

Application Scenarios

- Used when decoding or encoding object pool transfer messages that include object label reference lists.
- Helps to write robust code that avoids hard-coded attribute IDs.
- Serves as a reference for developers implementing ISOBUS UT clients or servers.

Comparison with Similar Blocks

Similar global constant sets exist for other attribute groups (e.g., AID_OL for object labels). Unlike function blocks, this constant set provides no processing capability; it simply offers named numeric constants. When compared to direct literals, using the constant improves code readability and maintainability.

Conclusion

The AID_OLRL global constant set is a simple, yet essential, component for applications dealing with ISOBUS object label reference lists. By defining the NUMB_LABELLED_OBJ constant, it standardizes the representation of this attribute ID, reducing the risk of errors and enhancing code clarity.

AID_OM

• – ◦ • – ◦ • – ◦ •

Introduction

AID_OM is a global constant definition set used in the context of ISOBUS virtual terminal (VT) applications. It defines attribute identifiers (IDs) for the **Output Meter** object, which is a graphical element used to display a value on a scale, typically with a needle and tick marks. The constants provide a standardized, human-readable way to reference the attributes of an output meter when constructing or modifying VT object data.

This global constant set belongs to the package isobus::UT::Q::const::AID and contains twelve USINT (unsigned short integer) constants, each representing a specific attribute ID (1 to 12) of the output meter object according to the ISOBUS standard ISO 11783.

Interface Structure

Since AID_OM is a global constants definition rather than a function block, adapter, or subapplication, it does not expose any runtime interface. The following sections reflect the standard FB interface structure

and are marked “Not applicable” where no corresponding members exist. The actual constant definitions are listed in the **Global Constants** subsection.

Event Inputs

Not applicable – `AID_OM` defines no event inputs.

Event Outputs

Not applicable – `AID_OM` defines no event outputs.

Data Inputs

Not applicable – `AID_OM` defines no data inputs, as it is a set of compile-time constants.

Data Outputs

Not applicable – `AID_OM` defines no data outputs.

Adapters

Not applicable – `AID_OM` contains no adapter instances.

Global Constants

The following table lists all constants defined in `AID_OM`. Each constant holds the numeric attribute ID used for the corresponding output meter property in a VT object.

Constant Name	Type	Value	Attribute ID	Description
`WIDTH`	USINT	1	1	Width in pixels (AID_OM_WIDTH).
`NEEDLE_COLOUR`	USINT	2	2	Needle colour index (AID_OM_NEEDLE_COLOUR).
`BORDER_COLOUR`	USINT	3	3	Border colour index (AID_OM_BORDER_COLOUR).
`ARC_THICK_COLOUR`	USINT	4	4	Arc thickness / colour index (AID_OM_ARC_THICK_COLOUR).
`OPTIONS`	USINT	5	5	Bitmask options: Bit 0 = Draw Arc, Bit 1 = Draw Border, Bit 2 = Draw Ticks, Bit 3 = Deflection direction (0 = anticlockwise, 1 = clockwise).
`NUMB_TICKS`	USINT	6	6	Number of tick marks (AID_OM_NUMB_TICKS).
`START_ANGLE`	USINT	7	7	Start angle of the scale (AID_OM_START_ANGLE).
`END_ANGLE`	USINT	8	8	End angle of the scale (AID_OM_END_ANGLE).
`MIN_VALUE`	USINT	9	9	Minimum displayed value (AID_OM_MIN_VALUE).
`MAX_VALUE`	USINT	10	10	Maximum displayed value (AID_OM_MAX_VALUE).
`VARIABLE_REF`	USINT	11	11	Object ID of a Variable object (AID_OM_VARIABLE_REF).
`VALUE`	USINT	12	12	Current value shown by the meter (AID_OM_VALUE).

Functionality

The `AID_OM` constants serve as static symbolic references for the attribute identifiers of an ISOBUS output meter object. When a control application needs to create, modify, or read an output meter object on a VT,

it uses these constants to specify which attribute is being addressed. For example, setting the `OPTIONS` attribute to a desired bitmask requires referring to it as `AID_OM.OPTIONS` rather than using the raw numeric ID 5.

Because the constants are defined once in a central location, they improve code readability, reduce the risk of typographical errors, and simplify maintenance if attribute IDs ever change in the future.

Technical Features

- **Type**: All constants are of type `USINT` (8-bit unsigned integer).
- **Initial Values**: Each constant is explicitly initialized to its attribute ID value (1 to 12) using typed literals (e.g., `USINT#1`).
- **Package**: `isobus::UT::Q::const::AID` – indicates the constant set belongs to the ISOBUS utility toolkit, within the object-oriented VT class hierarchy.
- **Standard Compliance**: The constants follow the attribute ID numbering defined by ISO 11783 (ISOBUS) for the Output Meter object.
- **No Runtime Side Effects**: As a `GlobalConstants` definition, it only influences compilation and does not occupy runtime memory or trigger logic execution.

State Overview

Not applicable – `AID_OM` contains no state information, no state machines, and no execution logic. It is a purely declarative constant set.

Application Scenarios

- **VT Object Construction**: When building an output meter object via a control function, use the constants to populate attribute fields in a structured way.
- **Dynamic Update**: When updating the meter's current value (`VALUE`), the constant can be used to set the corresponding attribute in a VT command without hard-coding numeric IDs.
- **Configuration**: Use `OPTIONS` to configure the visual appearance (arc, border, ticks, deflection direction) of the meter.
- **Scaling**: Use `MIN_VALUE`, `MAX_VALUE`, `START_ANGLE`, and `END_ANGLE` to define the scale's range and angular span.

Comparison with Similar Blocks

`AID_OM` is one of several global constant definitions under the `isobus::UT::Q::const::AID` package. Similar sets exist for other VT object types (e.g., output line, output rectangle, etc.), each following the same pattern: a collection of `USINT` constants with values matching the standard attribute IDs. The advantage of this approach over raw numeric literals is consistent naming, centralized documentation, and compile-time type safety. In contrast to function blocks or subapplications, `AID_OM` provides no algorithmic behavior – it serves purely as a reference library for application developers.

Conclusion

The `AID_OM` global constants set provides a clean, standardized, and type-safe means of referencing output meter attribute IDs in ISOBUS VT applications. By using these symbolic constants instead of raw numbers, developers can write more readable, maintainable, and error-resistant code when working with output meter objects.

AID_ON

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_ON` set of global constants defines the attribute identifiers (IDs) used for the **Output Number Object** within the ISO 11783 (ISOBUS) User Interface (UI) standard. These constants map human-readable names to numeric attribute codes, which are employed when reading or writing properties of an output number object on a Virtual Terminal or similar device.

This constant set is part of the `isobus::UT::Q::const::AID` package and provides a consistent way to reference object attributes without hard-coding numeric values in application logic.

Interface Structure

`AID_ON` is not a function block or adapter; it is a collection of constant definitions. Therefore, it does not have event or data inputs/outputs in the traditional sense. The following sections reflect the standard documentation structure for function blocks, but here they are marked as **not applicable**.

Event Inputs

None – `AID_ON` defines only global constants.

Event Outputs

None – `AID_ON` defines only global constants.

Data Inputs

None – `AID_ON` defines only global constants.

Data Outputs

None – `AID_ON` defines only global constants.

Adapters

None – `AID_ON` is a constant set, not a block with adapters.

Functionality

The `AID_ON` constants assign symbolic names to the attribute identifiers (1 to 12) used by the ISO 11783 Output Number Object. Each constant is of type `USINT` and has a fixed initial value corresponding to its attribute ID. These identifiers are used, for example, when constructing or parsing CAN messages that manipulate the properties of a number display object.

The following table summarises the constants:

Constant Name	Value	Description
`WIDTH`	1	Width in pixels.
`HEIGHT`	2	Height in pixels.
`BACKGROUND_COLOUR`	3	Background colour index.
`FONT_ATT`	4	Object ID of a Font Attributes object.
`OPTIONS`	5	Options bitmask.
`VARIABLE_REF`	6	Object ID of a Number Variable object.
`OFFSET`	7	Offset to be applied to the value for display.
`SCALE`	8	Scale to be applied to the value for display.
`NUMB_DECIMALS`	9	Number of decimals to display after the decimal point.
`FORMAT`	10	Format: 0 = fixed decimal, 1 = exponential.
`JUSTIFICATION`	11	Justification: bits 0-1 horizontal, bits 2-3 vertical.
`VALUE`	12	Current value before scaling.

Technical Features

- **Type:** All constants are declared as `USINT` (unsigned short integer).
- **Initial Values:** Each constant is initialised with its numeric attribute ID as a `USINT#` literal.
- **Package:** The constants are defined in the package `isobus::UT::Q::const::AID`.
- **Compatibility:** Complies with the IEC 61499 standard and is intended for use in 4diac-IDE projects interacting with ISOBUS virtual terminals.

State Overview

`AID_ON` does not represent a state machine; it is a set of constant values. There are no states or transitions.

Application Scenarios

- **ISOBUS Virtual Terminal Development:** The constants are used when building or interpreting commands that set or query attributes of an output number object on a Virtual Terminal.
- **HMI Design in Agricultural Machinery:** Developers can reference `AID_ON_WIDTH`, `AID_ON_HEIGHT`, etc., to configure display elements programmatically.
- **Protocol Implementation:** When constructing CAN messages according to the ISOBUS standard, these constants provide a clear and maintainable mapping.

Comparison with Similar Blocks

`AID_ON` is a global constants collection, not a function block. Similar constant sets exist for other object types (e.g., `AID_OFF` for input number objects) and are typically named with the prefix `AID_` followed by the object type. The key difference is that `AID_ON` specifically covers attributes of the **output number** object, whereas other sets cover different object categories. Using such named constants improves code readability and reduces errors compared to using raw numerical IDs.

Conclusion

The `AID_ON` constant set provides a clean, standardised way to refer to attribute identifiers of ISO 11783 Output Number Objects. By using these symbolic names, developers can write more maintainable and portable code for ISOBUS-based HMI applications in the 4diac environment.

AID_OP_POLY

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_OP_POLY` global constants set defines the attribute identifiers (IDs) for an **Output Polygon** object in the ISO 11783 (ISOBUS) protocol. These constants are used in the object pool definition to reference the geometric and rendering properties of a polygon, enabling standardized communication between tractor-implements and the virtual terminal. The set provides symbolic names for the numeric attribute IDs, improving code readability and maintainability.

Interface Structure

`AID_OP_POLY` is a **global constant set** within the 4diac-IDE environment. It does not expose any operational interface, such as event inputs/outputs, data inputs/outputs, or adapters. Instead, it provides compile-time constant values that can be imported and used anywhere in the automation project.

Event Inputs

None – the constant set is passive and does not react to events.

Event Outputs

None – no events are generated.

Data Inputs

None – values are not dynamically changeable.

Data Outputs

None – constants are only read at build time.

Adapters

None – no adapters are required or provided.

Functionality

The constants in `AID_OP_POLY` represent the attribute IDs defined by the ISOBUS standard for an **Output Polygon** object. Each ID is a fixed numeric value (USINT) that corresponds to a specific attribute in the object's definition. By using these symbolic names, developers can avoid hard-coding magic numbers and ensure compliance with the standard. The set includes attributes for:

- **Width** of the polygon outline (in pixels)
- **Height** of the polygon (in pixels)
- **Line Attributes** – reference to a Line Attributes object (ID)

- **Fill Attributes** – reference to a Fill Attributes object (ID)
- **Polygon Type** – convex, non-convex, complex, or open

These constants are typically used in conjunction with the `AID_OP_*` series to build complete ISOBUS object pools.

Technical Features

- **DataType**: All constants are of type `USINT` (unsigned short integer, 8-bit) with initial values from 1 to 5.
- **Namespace**: The constants reside in the package `isobus::UT::Q::const::AID`.
- **Scope**: Declared as `GLOBALCONSTANT` entities, they are globally accessible within the 4diac project after the defining library is imported.
- **Compatibility**: Compliant with IEC 61499-1 and the ISOBUS object pool specification.

Constant	Value	Description
`WIDTH`	`USINT#1`	Attribute ID for polygon width in pixels.
`HEIGHT`	`USINT#2`	Attribute ID for polygon height in pixels.
`LINE_ATT`	`USINT#3`	Object ID of a Line Attributes object for the polygon's outline.
`FILL_ATT`	`USINT#4`	Object ID of a Fill Attributes object for the polygon's interior.
`POLYGON_TYPE`	`USINT#5`	Polygon type: 0=Convex, 1=Non-Convex, 2=Complex, 3=Open.

State Overview

The `AID_OP_POLY` constant set is **stateless**. It contains only constant values that are evaluated at compile time; there is no runtime state machine or dynamic behavior. The constants are immutable and do not change during execution.

Application Scenarios

- **ISOBUS Object Pool Definition**: Used when defining an Output Polygon within a Virtual Terminal object pool, to set attributes like outline width, fill style, and polygon type.
- **Custom Implementations**: Developers implementing ISOBUS-compliant control systems can include these constants to avoid errors in attribute ID assignments.
- **Tooling and Simulation**: May be used by simulation and test tools to validate object pool structures.

Comparison with Similar Blocks

In the 4diac-IDE environment, several global constant sets exist for different object types (e.g., `AID_OP_LINE`, `AID_OP_RECT`). Like `AID_OP_POLY`, they provide symbolic names for attribute IDs of their respective geometric objects. The structure is analogous, but the actual attribute IDs and meanings differ. For example, `AID_OP_RECT` includes attributes for corner radius and line style, while `AID_OP_POLY` focuses on polygon-specific properties like polygon type and width/height.

Conclusion

The `AID_OP_POLY` global constant set is a convenient and standard-compliant way to reference the attribute IDs of an ISOBUS Output Polygon object. By using the symbolic names instead of raw numbers, IEC 61499 applications become more readable, less error-prone, and easier to maintain. The constant set is stateless, integrates seamlessly with 4diac-IDE, and directly supports the creation of robust ISOBUS object pools.

AID_OP_PTR

• – ◦ • – ◦ • – ◦ •

Introduction

The **AID_OP_PTR** global constant container defines the object pointer attribute identifiers used in the ISOBUS (ISO 11783) protocol stack. It centralizes the enumeration of object pointer attributes, ensuring consistent and unambiguous references across the system. This resource is part of the `isobus::UT::Q::const::AID` package and provides a single constant, `VALUE`, representing the current value attribute.

Interface Structure

As a global constant container, **AID_OP_PTR** does not expose any event or data interfaces. It only provides a set of compile-time constants that can be referenced globally by other function blocks and modules.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The primary purpose of **AID_OP_PTR** is to supply a stable, well-defined identifier for the object pointer value attribute. The constant `VALUE` is defined as:

- **Name**: `VALUE`
- **Data Type**: `USINT` (Unsigned Short Integer)
- **Initial Value**: `USINT#1`
- **Semantic**: `1: AID_OP_PTR_VALUE` – Current value.

This constant can be used in the implementation of ISOBUS object pools, especially when handling object pointer references. By referencing `AID_OP_PTR.VALUE` (or the equivalent qualified name), developers avoid hard-coding numeric literals and ensure clarity and maintainability.

Technical Features

- **Type**: `USINT` – Provides a compact 8-bit unsigned representation, sufficient for attribute identifiers.
- **Initialization**: The constant is defined with an explicit initial value (`USINT#1`), guaranteeing a predictable default.
- **Namespace**: Lives under `isobus::UT::Q::const::AID`, which organizes all object attribute identifiers in a hierarchical manner.
- **Compatibility**: Works seamlessly with 4diac-IDE projects that require global constant definitions for protocol-specific parameters.

State Overview

Since **AID_OP_PTR** is a constant container, it does not have any runtime state or transitions. The value of `VALUE` remains fixed throughout the execution of the application.

Application Scenarios

- **ISOBUS Object Pool Implementation**: When defining object pointers (e.g., for the Virtual Terminal), the attribute ID `1` is required to denote the current value. Using `AID_OP_PTR.VALUE` ensures correct interpretation across different implementations.
- **Protocol Extensions**: If additional object pointer attributes are introduced in future revisions, they can be appended to this constant container, maintaining a single source of truth.
- **Code Clarity**: Enhances readability by giving a meaningful name to the numeric ID `1`, making the code self-documenting.

Comparison with Similar Blocks

Unlike other ISOBUS constant blocks (such as `AID_PTR`, `AID_OBJECT`, or `AID_VIRTUAL_TERMINAL`), **AID_OP_PTR** specifically focuses on object pointer attributes. It follows the same pattern of defining `USINT` constants for each attribute ID, but its purpose is limited to the object pointer context. This separation keeps the constant sets logically organized and prevents accidental reuse of identifiers.

Conclusion

AID_OP_PTR is a small but essential constant container in the ISOBUS stack. By providing the standard object pointer attribute ID, it supports the correct implementation of object pools and contributes to the maintainability and portability of industrial automation applications using the 4diac-IDE. Its simplicity and clear definition make it a reliable building block for any ISOBUS-compliant system.

AID_OR

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_OR` global constants block defines a set of attribute identifiers used for output rectangle objects in the ISOBUS (ISO 11783) agricultural electronics protocol. These constants provide a standardized way to reference the attributes of an output rectangle, such as line style, width, height, line suppression, and fill attributes. The values are defined as unsigned short integers (USINT) and are intended to be used in conjunction with object pool definitions and communication interfaces.

Interface Structure

The `AID_OR` block does not contain any event inputs, event outputs, or adapters. It solely provides a collection of data constants that can be referenced globally throughout a 4diac application.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

The following table lists the constant data values defined by this block. They are accessible as read-only variables.

Name	Type	Value	Comment
`LINE_ATT`	USINT	1	Object ID of a Line Attributes object.
`WIDTH`	USINT	2	Width in pixels.
`HEIGHT`	USINT	3	Height in pixels.
`LINE_SUPPR`	USINT	4	Line suppression attribute.
`FILL_ATT`	USINT	5	Object ID of a Fill Attributes object.

Adapters

None.

Functionality

The global constants are designed to be used as arguments when creating or modifying output rectangle objects in an ISOBUS object pool. By referencing these named constants instead of hard-coded numeric values, the application code becomes more readable, self-documenting, and less error-prone. For example, when setting the width attribute of a rectangle, the constant `AID_OR.WIDTH` can be used to indicate that the following value refers to the width in pixels.

Technical Features

- **Type:** All constants are of type `USINT` (unsigned short integer), capable of representing values from 0 to 255.
- **Package:** The constants belong to the package `isobus::UT::Q::const::AID`, which organizes them within the namespace hierarchy.

- **Initialization:** Each constant is initialized with a specific attribute ID, ensuring consistent behavior across all referencing components.
- **Read-Only:** These are global constants and cannot be modified at runtime, providing compile-time safety.

State Overview

Since `AID_OR` contains only constants, there is no runtime state or dynamic behavior. The values are fixed at compile time and remain constant throughout the execution of the application.

Application Scenarios

- **Object Pool Creation:** When constructing an ISOBUS object pool, the developer can use these constants to define properties of output rectangle objects, such as line type, dimensions, or fill style.
- **Communication Parameterization:** The constants can be passed to commands that update the attributes of a rectangle object on a virtual terminal or control unit.
- **Code Maintainability:** Using named constants improves code clarity and makes it easier to update attribute IDs if the ISOBUS specification changes, since only the constant definitions need to be modified.

Comparison with Similar Blocks

In the same package, there may be similar global constant blocks for other object types (e.g., `AID_OT` for output text, `AID_OB` for output bar). `AID_OR` specifically focuses on output rectangle attributes, providing a distinct set of IDs tailored to that object type. Unlike function blocks that encapsulate logic, `AID_OR` is a pure data definition unit, making it lightweight and universally applicable.

Conclusion

The `AID_OR` global constants block is an essential utility for any ISOBUS-based application that deals with output rectangle objects. It centralizes attribute identifiers, promotes code reuse, and ensures consistency across different modules. By adopting these constants, developers can build more robust and maintainable agricultural automation systems.

AID_OS

• − ◦ • − ◦ • − ◦ •

Introduction

The **AID_OS** global constant set defines the attribute identifiers (IDs) for the **Output String (OS) object** within the ISO 11783 (ISOBUS) protocol, specifically for the Working Set (WS) and Virtual Terminal (VT) layers. These IDs are used to reference specific attributes of an output string object when interacting with the object pool via the ISOBUS service primitives. The set is part of a larger collection of constant definitions that simplify and standardize the handling of object attributes in 4diac-based ISOBUS applications.

The constants are of type `USINT` (unsigned short integer) and range from 1 to 7, each corresponding to a unique attribute of the output string object, such as width, height, background color, font attributes, options, variable reference, and justification.

Interface Structure

This global constant set does not represent a function block, adapter, or subapplication. Instead, it provides a set of named constants that can be used directly in IEC 61499 applications to access the respective attribute IDs. The “interface” of the set is therefore the list of constants it exposes.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None directly. However, the constants serve as data values that can be passed to function blocks or functions that require an attribute ID.

Data Outputs

The following global constants are defined and can be referenced anywhere in the application:

Constant Name	Value	Description
WIDTH	USINT#1	1: AID_OS_WIDTH – Width in pixels.
HEIGHT	USINT#2	2: AID_OS_HEIGHT – Height in pixels.
BACKGROUND_COLOUR	USINT#3	3: AID_OS_BACKGROUND_COLOUR – Background colour index.
FONT_ATT	USINT#4	4: AID_OS_FONT_ATT – Object ID of a Font Attributes object.
OPTIONS	USINT#5	5: AID_OS_OPTIONS – Bitmask: Bit 0 = Transparent, Bit 1 = Auto-Wrap, Bit 2 = Wrap on Hyphen.
VARIABLE_REF	USINT#6	6: AID_OS_VARIABLE_REF – Object ID of a Variable object.
JUSTIFICATION	USINT#7	7: AID_OS_JUSTIFICATION – Justification: Bits 0-1 (Horizontal: 0=Left, 1=Middle, 2=Right), Bits 2-3 (Vertical: 0=Top, 1=Middle, 2=Bottom).

Adapters

None.

Functionality

The AID_OS constant set is used to map symbolic names to numeric attribute IDs for an ISOBUS Output String object. When an application needs to read or write an attribute of such an object (e.g., via the Set_Attribute or Get_Attribute services), it can use these constants instead of hard-coded numeric values. This improves code readability, maintainability, and reduces the risk of errors.

The constants are declared as global constants within the namespace isobus::UT::Q::const::AID, ensuring they are globally accessible and immutable. They are compiled directly into the application and occupy no runtime memory beyond the constant literals.

Technical Features

- **Type**: USINT (unsigned 8-bit integer)
- **Range**: 1 to 7 (inclusive)
- **Namespace**: isobus::UT::Q::const::AID
- **Immutability**: The constants are declared as CONSTANT and cannot be modified at runtime.
- **Direct Usage**: May be used in any IEC 61499 FB or ST code without further instantiation.
- **Documentation**: Each constant is accompanied by a comment describing its corresponding ISOBUS AID_OS_* definition.

State Overview

Not applicable. This is a constant set without internal states or dynamic behavior.

Application Scenarios

Typical use cases for AID_OS include:

- **ISO 11783 Virtual Terminal Implementations**: When developing a VT client or server that manages output string objects, these constants are used to construct attribute access commands.
- **Object Pool Manipulation**: When reading or modifying object attributes in an ISOBUS object pool, the constants provide a clear mapping.
- **Diagnostic and Monitoring Tools**: Applications that inspect or alter output string properties (e.g., text alignment, transparency, size) can use these IDs to identify the target attribute.
- **Integration with 4diac Function Blocks**: In 4diac, function blocks that handle ISOBUS communication can accept these constants as inputs to specify which attribute to target.

Comparison with Similar Blocks

In the same ISOBUS context, other constant sets exist for different object types, such as:

- AID_OBJ – general object attributes
- AID_AB – for Alphanumeric Input objects

- AID_WS – for Working Set objects
- AID_OS is specifically tailored to the Output String object.

Unlike a function block, AID_OS does not execute logic or contain state; it is purely a data definition. It serves as a companion to function blocks that implement the ISOBUS communication services, providing a consistent and readable way to reference attribute IDs.

Conclusion

AID_OS is a compact and essential set of global constants that encapsulate the attribute identifiers of the ISOBUS Output String object. By using these symbolic constants, developers can write clearer, more maintainable 4diac applications that interact with ISOBUS VTs. The set is well-documented, follows the standard naming convention, and aligns with the Eclipse 4diac ecosystem's philosophy of modular and reusable components.

AID_PG

• – ◦ • – ◦ • – ◦ •

Introduction

The AID_PG global constants set defines the attribute identifiers (IDs) for **Picture Graphic (PG)** objects within ISO 11783 (ISOBUS) object pools. These constants are used by function blocks and applications to reference specific properties of picture graphic objects, such as their dimensions, transparency settings, and colour format. By using symbolic names instead of raw numeric values, the code becomes more readable, self-documenting, and resilient against changes in the underlying protocol specifications.

Interface Structure

AID_PG is not a function block or adapter but a collection of global constants. It does not possess dynamic interfaces such as events, data inputs/outputs, or adapters. Instead, it provides a set of named constant values that can be referenced throughout a 4diac project.

Event Inputs

None. The constants set does not process events.

Event Outputs

None. The constants set does not generate events.

Data Inputs

None. The constants set does not accept any data inputs.

Data Outputs

The following global constants are defined, serving as static attribute IDs for picture graphic objects:

Constant	Type	Initial Value	Description
WIDTH	USINT	1	Attribute ID for the width of the picture graphic in pixels (AID_PG_WIDTH).
OPTIONS	USINT	2	Attribute ID for an options bitmask – bit 0: transparent, bit 1: flashing, bit 2: run-length encoded (RLE) (AID_PG_OPTIONS).
TRANSPARENCY_COLOUR	USINT	3	Attribute ID for the transparency colour index (AID_PG_TRANSPARENCY_COLOUR).
ACTUAL_WIDTH	USINT	4	Attribute ID for the actual width in pixels (AID_PG_ACTUAL_WIDTH).
ACTUAL_HEIGHT	USINT	5	Attribute ID for the actual height in pixels (AID_PG_ACTUAL_HEIGHT).
FORMAT	USINT	6	Attribute ID for the picture graphic format: 0 = monochrome, 1 = 4-bit colour, 2 = 8-bit colour (AID_PG_FORMAT).

Adapters

None. The constants set does not contain any adapter instances.

Functionality

The `AID_PG` constants serve as standardised numeric identifiers for the attributes of a picture graphic object in an ISOBUS object pool. When a function block needs to read or write a picture graphic attribute (e.g., during object pool construction, dynamic updates, or diagnostics), it can use these constants as the attribute ID parameter. This approach ensures interoperability with ISOBUS terminals and simplifies the maintenance of object pool definitions.

Technical Features

- All constants are of type `USINT` (unsigned short integer), requiring only a single byte of storage.
- The values are fixed at compile time and cannot be modified during runtime.
- The constant names follow an intuitive naming convention that corresponds to the official ISOBUS attribute names.
- The constants are defined in a namespace (`isobus::UT::Q::const::AID`) to avoid naming conflicts with other global constants.

State Overview

Not applicable – a global constants set does not have a state machine. It provides static data that remains unchanged throughout the execution of the application.

Application Scenarios

The `AID_PG` constants are predominantly used in ISOBUS compatible agricultural machinery and control systems. Typical scenarios include:

- **Object pool generation** – when creating or uploading graphic objects to a Virtual Terminal, the attribute IDs for picture graphics are required to set properties such as width, height, transparency, and format.
- **Dynamic graphic updates** – in applications that modify picture graphics at runtime (e.g., displaying changing weather maps or yield data), these constants allow efficient access to the respective attributes.
- **Diagnostics and debugging** – during development, referencing attribute IDs by symbolic constants enhances code readability and reduces the risk of using incorrect numeric values.

Comparison with Similar Blocks

In ISOBUS, each object type (e.g., Object, Button, Output, Input, Meter, BarGraph, etc.) has its own set of attribute ID constants. The `AID_PG` set is specifically dedicated to Picture Graphic objects. Comparable constants sets include:

- `AID_OBJ` for general object attributes,
- `AID_BAR` for bar graph attributes,
- `AID_BTN` for button attributes,
- `AID_INP` for input attributes.

Unlike function blocks, these constants do not contain logic or processing capabilities; they merely provide a structured, named collection of numeric IDs that facilitate the implementation of ISOBUS communication.

Conclusion

The `AID_PG` global constants set provides a clean and standardised way to reference Picture Graphic object attribute IDs in 4diac IDE projects targeting ISOBUS applications. By using these symbolic constants, developers can avoid hard-coded numeric values, improve code maintainability, and ensure full compliance with the ISO 11783 standard. Although it is not an executable functional element, it plays a crucial supporting role in the development of robust and interoperable agricultural automation systems.

AID_SG

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_SG` global constant set defines attribute identifiers for scaled graphic objects in the ISOBUS Universal Terminal (UT) protocol. These constants are used to reference specific properties of scaled

graphical elements, such as width, height, scaling behavior, options, and value, within a 4diac-based control application. They are part of the `isobus::UT::Q::const::AID` package and are intended to improve code readability and maintainability by providing symbolic names for numeric attribute IDs.

Interface Structure

As a set of global constants, `AID_SG` does not possess an event or adapter interface. The following sections describe the available data constants.

Event Inputs

None.

Event Outputs

None.

Data Inputs

The following constants are defined globally and can be used directly within function blocks or programs:

Name	Type	Value	Description
`WIDTH`	`USINT`	`USINT#1`	`AID_SG_WIDTH` – Width in pixels.
`HEIGHT`	`USINT`	`USINT#2`	`AID_SG_HEIGHT` – Height in pixels.
`SCALE_TYPE`	`USINT`	`USINT#3`	`AID_SG_SCALE_TYPE` – Scaling and justification encoding (see bit field description below).
`OPTIONS`	`USINT`	`USINT#4`	`AID_SG_OPTIONS` – Bitmask; Bit 0 = Flashing.
`VALUE`	`USINT`	`USINT#5`	`AID_SG_VALUE` – Current value of the scaled graphic object.

Data Outputs

None.

Adapters

None.

Functionality

The constants in `AID_SG` provide standardized identifiers for the attributes of scaled graphic objects as defined by the ISOBUS standard. The `SCALE_TYPE` constant encapsulates both scaling behaviour (bits 0–2) and horizontal/vertical justification (bits 3–6). The `OPTIONS` constant is used to enable additional visual effects, such as flashing. These constants are intended to be used in conjunction with ISOBUS UT services that read or write object properties – they map directly to the attribute ID numbers defined in the protocol.

Technical Features

- All constants are of type `USINT` (unsigned short integer), occupying one byte.
- Values are fixed at compile time and cannot be changed during runtime.
- The `SCALE_TYPE` field is a packed bit field:
  - Bits 0–2: Scaling value – `0` = no scale, `1` = scale to width, `2` = scale to height, `3` = scale to width and height, `4` = scale to fit.
  - Bits 3–4: Horizontal justification – `0` = left, `1` = middle, `2` = right.
  - Bits 5–6: Vertical justification – `0` = top, `1` = middle, `2` = bottom.
- The `OPTIONS` constant currently defines only bit 0 (flashing); remaining bits are reserved for future use.

State Overview

Not applicable – global constants do not have state.

Application Scenarios

These constants are typically used in ISOBUS UT applications where scaled graphic objects (e.g., gauges, bars, or image containers) need to be configured or dynamically updated. For example:

- Setting the scaling parameters of a graphic element using `SCALE_TYPE`.
- Reading or writing the `VALUE` attribute to update the displayed value.

- Enabling or disabling flashing via the `OPTIONS` bitmask.

By referencing these constants, developers avoid hard-coding numeric attribute IDs, reducing the risk of errors and improving code clarity.

Comparison with Similar Blocks

Other global constant sets exist for different ISOBUS object types, such as `AID_OBJ` for general objects or `AID_TXT` for text objects. `AID_SG` is specifically tailored to scaled graphic objects and provides only the attributes relevant to that class. Unlike an FB, it has no executable logic; it serves purely as a lookup table for attribute identifiers.

Conclusion

The `AID_SG` global constant set is a lightweight but essential component for ISOBUS UT development in 4diac. It encapsulates the numeric attribute IDs for scaled graphic objects in an easy-to-use symbolic form, supporting both readability and portability. Its fixed definitions align with the ISOBUS standard, making it a reliable building block for agricultural control systems.

AID_SK

• – ◦ • – ◦ • – ◦ •

Introduction

AID_SK is a global constant container that defines attribute identifiers (IDs) for soft key objects in an ISOBUS Virtual Terminal (VT) environment. It provides standardized numeric constants used to reference specific properties of soft keys during communication between the control function and the VT.

The constants are defined as unsigned 8-bit integers (USINT) and are part of the `isobus::UT::Q::const::AID` package. They serve as a central source of truth for soft key attribute IDs, ensuring consistency across applications and reducing the risk of hard-coded values.

Interface Structure

AID_SK is a global constant definition and does not expose any event interfaces, data inputs/outputs, or adapters. It contains only predefined constant values.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The purpose of AID_SK is to provide two constant values that correspond to the attribute IDs used in ISOBUS VT soft key objects:

- `BACKGROUND_COLOUR` (value = 1): This ID refers to the background colour index of a soft key.
- `KEY_CODE` (value = 2): This ID is used by the VT to report the code in a **Soft Key Activation** message.

By defining these values as global constants, all parts of an application can reference them by name rather than by numeric literals, improving code readability and maintainability.

Technical Features

- **Constants**: Both constants are of type `USINT` (unsigned 8-bit integer).
- **Initial Values**: Predefined fixed values: `BACKGROUND_COLOUR = 1`, `KEY_CODE = 2`.

- **Package**: The constants are encapsulated in the `isobus::UT::Q::const::AID` package, following the ISOBUS naming conventions.
- **Read-Only**: These are global constants, meaning they cannot be modified at runtime.

State Overview

Not applicable – AID_SK is a static constant container and has no runtime state or behavior.

Application Scenarios

AID_SK is typically used in ISOBUS VT applications, particularly when implementing soft key interactions. For example:

- When configuring a soft key's background colour, the application can use `AID_SK.BACKGROUND_COLOUR` to refer to the corresponding attribute ID.
- When handling a **Soft Key Activation** event, the VT sends a key code that matches `AID_SK.KEY_CODE`. Applications can compare or utilize this value to identify which soft key was activated.

This constant container helps standardize attribute IDs across different parts of an application and aligns with the ISOBUS standard's definitions.

Comparison with Similar Blocks

In the same package `isobus::UT::Q::const::AID`, other constant containers exist for different object types (e.g., `AID_OK` for object keys, `AID_AT` for attributes, etc.). AID_SK specifically focuses on soft key attributes. Compared to other constants, AID_SK defines only two IDs that are directly related to soft keys, making it a minimal and focused constant set.

Conclusion

AID_SK provides two essential constant values for soft key attribute IDs in ISOBUS VT applications. By centralizing these definitions, it promotes code clarity, reduces errors, and aligns with standard ISOBUS conventions. The constant container is simple, yet crucial for any application involving soft key configuration or activation handling.

AID_SKM

• – ◦ • – ◦ • – ◦ •

Introduction

AID_SKM is a global constants block that defines attribute identifiers for the soft key mask object within the ISOBUS virtual terminal. It serves as a central repository for constant values, ensuring consistent referencing and maintainability across the automation system.

Interface Structure

AID_SKM does not possess conventional function block interfaces such as event or data inputs/outputs, nor does it contain adapters. Instead, it declares global constants that are accessible system-wide.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

The block provides the following global constant, which is treated as a read-only output value:

Name	Type	Initial Value	Comment
BACKGROUND_COLOUR	USINT	USINT#1	1: AID_SKM_BACKGROUND_COLOUR – Background colour index.

Adapters

None.

Functionality

AID_SKM encapsulates constant definitions used for encoding and decoding soft key mask object attributes. The sole constant `BACKGROUND_COLOUR` represents the attribute identifier for the background colour index of the soft key mask, aligning with the ISOBUS standard.

Technical Features

- **Global Scope:** The constant is declared as `VAR_GLOBAL CONSTANT`, making it available in all POUs and configurations within the project.
- **Standard Compliance:** The attribute identifier follows the ISOBUS 11783-14 numbering scheme, facilitating interoperability with other virtual terminal implementations.
- **Type Safety:** The constant is explicitly typed as `USINT` (unsigned short integer), preventing inadvertent type mismatches.

State Overview

As a global constants block, AID_SKM is stateless. Its value is fixed at initialization and remains unchanged during runtime.

Application Scenarios

- **VT Screen Development:** Use `AID_SKM.BACKGROUND_COLOUR` when constructing or modifying soft key masks in order to set the background colour attribute.
- **Protocol Encoding:** When packing or unpacking ISOBUS commands that involve soft key mask attributes, this constant provides the correct attribute ID for the background colour.
- **Configuration Consistency:** Replacing hard-coded numeric values with named constants enhances code readability and reduces errors during maintenance.

Comparison with Similar Blocks

Unlike function blocks that process events or data, AID_SKM is a global constant container. Similar global constant blocks may exist for other object types (e.g., `AID_KI` for key masks), but they share the same principle: defining immutable attribute identifiers in a centralized location. Compared to using literals directly, a dedicated constant block improves traceability and simplifies future updates.

Conclusion

AID_SKM serves as a well-defined source for the soft key mask background colour attribute identifier. By using this global constant, developers ensure compliance with the ISOBUS standard while maintaining clean and maintainable code within 4diac-IDE environments.

AID_TYPE

• – ◦ • – ◦ • – ◦ •

Introduction

`AID_TYPE` is a global constant definition used in isobus-based applications. It defines a single constant named `OBJ_TYPE` of type `USINT` with an initial value of `0`. This constant represents the Object Type attribute identifier for an AID (Application Identification) message, as specified in the ISO 11783 (ISOBUS) standard.

Interface Structure

Since `AID_TYPE` is a GlobalConstants block, it does not contain typical FB interfaces such as event inputs, data inputs, or adapters. Instead, it exposes a single global constant:

Data Inputs

None.

Data Outputs

None.

Adapters

None.

However, the constant OBJ_TYPE is globally accessible to any FB or application that needs to reference the Object Type identifier.

Functionality

AID_TYPE provides a standardized, constant value for the Object Type attribute ID in ISOBUS communication. The constant OBJ_TYPE is assigned the value 0, which corresponds to the AID_TYPE_OBJ_TYPE attribute. This constant is used to identify the type of object in AID messages, helping to ensure consistency across different implementations.

Technical Features

- **Type:** USINT (Unsigned Short Integer, 8-bit)
- **Initial Value:** 0 (using IEC 61131-3 notation USINT#0)
- **Scope:** Global constant, accessible throughout the entire 4diac application.
- **Package Context:** Defined within the package isobus::UT::Q::const::AID.

State Overview

Global constants have no runtime state. The value of OBJ_TYPE is fixed at 0 and cannot be modified during execution.

Application Scenarios

- **ISOBUS Message Handling:** When constructing or parsing AID messages, the constant can be used to set or compare the Object Type field.
- **System Consistency:** Provides a single source of truth for the Object Type identifier, reducing the risk of hardcoding errors across multiple FBs or modules.

Comparison with Similar Blocks

Unlike function blocks or adapters, GlobalConstants blocks do not process events or data. Their purpose is to define reusable constants. Similar global constant blocks might define other attribute IDs (e.g., AID_ATTRIBUTE, AID_VALUE) but AID_TYPE specifically focuses on the Object Type attribute.

Conclusion

AID_TYPE is a simple, yet essential global constant definition that standardizes the Object Type attribute ID for ISOBUS AID messages. By using this constant, developers can avoid magic numbers and improve maintainability of their 4diac applications.

AID_WM

• – ◦ • – ◦ • – ◦ •

Introduction

The AID_WM global constant group defines attribute identifiers for window mask objects within the ISOBUS (ISO 11783) Universal Terminal protocol. These constants are used as keys to reference specific attributes when reading or writing window mask object properties through the ISOBUS communication interface. They belong to the isobus::UT::Q::const::AID package and are defined in the namespace of the 4diac-ide development environment.

Interface Structure

Since AID_WM is defined as a global constants group rather than a function block, it does not expose event inputs, event outputs, data inputs, data outputs, or adapter interfaces. Instead, it provides a set of named constant values that can be referenced throughout the application. The following constant values are available within this group:

Data Constants

Constant Name	Data Type	Initial Value	Description
BACKGROUND_COLOUR	USINT	USINT#1	Attribute ID 1: Window mask background colour index.
OPTIONS	USINT	USINT#2	Attribute ID 2: Bitmask for window mask options.

NAME	USINT	USINT#3	Attribute ID 3: Object ID of an Output String or Object Pointer containing the window mask name.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The AID_WM constants serve as identifiers matching the attribute IDs defined by the ISOBUS standard for window mask objects. When interacting with a window mask object on an ISOBUS Universal Terminal, these IDs are used in command and response messages to denote which specific attribute of the object is being addressed.

- **BACKGROUND_COLOUR** refers to the colour index used to render the background of the window mask. The colour index is typically interpreted according to the terminal's colour palette.
- **OPTIONS** is a bitmask that encodes configuration flags for the window mask: Bit 0 indicates availability (when set to 0, the window is disabled or blanked), and Bit 1 indicates transparency (when set, the window does not use a background colour).
- **NAME** provides a reference to another object (an Output String or Object Pointer) that contains the display name of the window mask.

By using these predefined constants, applications avoid hardcoded numeric values and improve code readability and maintainability.

Technical Features

- **Data Type**: All constants are of type USINT (Unsigned Short Integer, 8 bits).
- **Initial Values**: Each constant is initialized with its corresponding attribute ID value.
- **Namespace**: The constants reside in the isobus::UT::Q::const::AID package, which organizes attribute ID constants systematically for ISOBUS object types.
- **Licensing**: The original source is provided under the Eclipse Public License 2.0 (EPL-2.0).
- **Compiler Compatibility**: The package is intended for use with the 4diac-ide compiler environment targeting IEC 61499 applications.

State Overview

The AID_WM constant group does not maintain any runtime state. It is a compile-time constant definition, and all values are statically initialized. No state transitions or lifecycle management apply.

Application Scenarios

The AID_WM constants are used in agricultural machinery and implement control systems where a Universal Terminal (UT) communicates with an implement ECU (Electronic Control Unit) via the ISOBUS network. Typical application scenarios include:

- **Window Mask Creation**: When defining a window mask object on the UT, the application uses AID_WM constants to specify which attribute to set during initialization.
- **Attribute Retrieval**: When requesting the current value of a window mask attribute (e.g., background colour), the constant IDs are inserted into the command message.
- **Dynamic Configuration**: The OPTIONS constant allows runtime toggling of the window's availability or transparency, for instance, to disable a menu entry when a function is not operational.

- **UI Text Management**: The `NAME` constant is used to associate a text object with a window mask, enabling dynamic label changes.

Comparison with Similar Blocks

Within the same ISOBUS attribute ID constant family, similar groups exist for other object types, such as:

- **`AID_BTC`** – Button and ButtonMask attribute IDs.
- **`AID_OS`** – Output String attribute IDs.
- **`AID_ALM`** – Alarm object attribute IDs.
- **`AID_EP`** – External Input/Output Point attribute IDs.

While all these groups follow the same pattern of defining USINT constants for attribute IDs, `AID_WM` specifically targets window mask objects. Unlike a function block, which provides executable logic and event-driven behavior, `AID_WM` is purely a data declaration that supplies numeric identifiers. This makes it similar to enumerations or `#define` constants in traditional programming, but with the advantage of being a strongly typed global constant within the 4diac-ide environment.

Conclusion

The `AID_WM` global constants provide a standardized, readable way to reference window mask object attributes in ISOBUS applications. By encapsulating the attribute IDs as named constants, the group enhances code maintainability and reduces the risk of errors from magic numbers. As part of the broader `isobus::UT::Q::const::AID` package, it integrates seamlessly with other attribute ID constants, supporting a coherent development approach for ISO 11783-compliant systems in 4diac-ide.

AID_WS

• – ◦ • – ◦ • – ◦ •

Introduction

AID_WS is a set of global constants used in the ISOBUS (ISO 11783) working set object attribute identifiers. These constants define the attribute IDs (AIDs) for working set objects, which are essential for identifying and manipulating properties of a working set within an ISOBUS VT (Virtual Terminal) system. The constants provide a standardized way to reference background colour, selectable state, and the active mask of a working set, ensuring consistent and reliable communication between the VT and the implement control unit (ECU).

This global constants type is part of the `isobus::UT::Q::const::AID` package and is intended to be used in applications that implement ISOBUS-based user interface components. By using these predefined constants, developers can avoid magic numbers and improve code readability and maintainability.

Interface Structure

AID_WS is a global constants definition and does not have any event inputs, event outputs, data inputs, data outputs, or adapters. However, it exposes a set of constant values that can be referenced globally within the IEC 61499 application. These constants are listed below as equivalent to data outputs for clarity.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

The following constant values are defined by AID_WS and are accessible globally:

Constant Name	Type	Value	Description
`BACKGROUND_COLOUR`	`USINT`	1	Attribute ID for the background colour index of a working set.
`SELECTABLE`	`USINT`	2	Attribute ID indicating whether the working set can be selected by the operator (0 = FALSE, 1 = TRUE).

ACTIVE_MASK	USINT	3	Attribute ID for the object ID of the data or alarm mask to be displayed when the working set is active.

Adapters

None.

Functionality

The AID_WS constants are used to access and manipulate attributes of working set objects in an ISOBUS environment. Each constant corresponds to a specific attribute ID defined in the ISOBUS standard (Part 1 – Part 8, especially Part 6 for Virtual Terminal). By using these constants in conjunction with service functions or protocol requests, an application can query or set the background colour, selectable flag, or active mask of a working set.

The values are defined as `USINT` (unsigned short integer) and follow the AID numbering scheme from the ISOBUS standard. For example, the background colour attribute is assigned ID 1, the selectable attribute ID 2, and the active mask attribute ID 3. This mapping is consistent across all ISOBUS-compliant implementations.

Technical Features

- **Global Availability**: The constants are defined as `VAR_GLOBAL CONSTANT`, meaning they are accessible from any function block or resource within the same application without explicit declaration.
- **Type Safety**: All constants are of type `USINT`, which matches the attribute ID data type used in ISOBUS protocol messages.
- **Documentation**: Each constant is accompanied by a comment explaining its meaning, derived from the ISOBUS standard.
- **Compatibility**: The constants align with the ISOBUS attribute IDs, ensuring interoperability with other ISOBUS-compliant devices.
- **No Dynamic Behavior**: As a global constants type, AID_WS does not contain any logic or state; it is purely a definition.

State Overview

Since AID_WS is a constant definition, it has no runtime state. The values are fixed at compile time and cannot be modified during execution. This ensures that all references to these attributes are consistent throughout the application.

Application Scenarios

AID_WS is typically used in ISOBUS Virtual Terminal (VT) implementations, particularly in function blocks that handle working set objects. Common use cases include:

- Creating or configuring a working set on a VT by assigning the background colour index.
- Determining whether a working set is selectable and acting accordingly.
- Setting the active mask (data or alarm) that should be displayed when the working set becomes active.

For example, a function block might use `AID_WS::BACKGROUND_COLOUR` in a request to set the background of a working set, or it might parse a response and use `AID_WS::SELECTABLE` to check the selectable status.

Comparison with Similar Blocks

In ISOBUS systems, several other attribute ID sets exist for different object types, such as AID_OM (Object Masks), AID_AUX (Auxiliary Functions), etc. AID_WS specifically targets the working set object, while other constants sets would be used for other object types. Unlike function blocks that execute algorithms, AID_WS is a data definition, so it cannot be compared to functional blocks in terms of behavior. However, it serves a similar purpose to other global constant types: providing named, standardized values to avoid hardcoded numbers.

Conclusion

AID_WS provides a clear and standardized set of constant definitions for working set attribute IDs in ISOBUS applications. By using these constants, developers can write more readable and maintainable code that is compliant with the ISOBUS standard. The absence of dynamic behavior ensures reliability and consistency. This global constants type is an essential building block for any IEC 61499 application that interacts with ISOBUS Virtual Terminal working sets.

AID_WSSC

• – ◦ • – ◦ • – ◦ •

Introduction

The `AID_WSSC` global constants definition provides a set of predefined attribute identifiers for **Working Set Special Controls** objects in the ISO 11783 (ISOBUS) environment. These constants are used to reference specific attributes when interacting with such objects in a tractor-implement control system. The definition is part of the `isobus::UT::Q::const::AID` package, enabling consistent and readable addressing of these attributes across different parts of an industrial automation application.

Interface Structure

This element is a **global constants** block, not a conventional function block. As such, it does not have any event inputs, event outputs, or adapters. Instead, it exposes a set of constant values that can be directly referenced in the application logic.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

Constant Name	Data Type	Initial Value	Comment
`NUMOFBYTES`	`USINT`	`USINT#1`	Number of bytes to follow in this object (attribute ID 1).
`COLOUR_MAP`	`USINT`	`USINT#2`	Object ID of a Colour Map object, or NULL (attribute ID 2).
`COLOUR_PALETTE`	`USINT`	`USINT#3`	Object ID of a Colour Palette object, or NULL (attribute ID 3).

These constants are read-only and can be used anywhere in the program to refer to the corresponding attribute IDs.

Adapters

None.

Functionality

The `AID_WSSC` constants define the attribute identifiers used for the **Working Set Special Controls** object as specified by the ISO 11783 standard. In practice, these constants are used when constructing or interpreting ISOBUS messages that deal with working set special controls, such as setting or retrieving colour map or palette information. By using these named constants, the code becomes self-documenting and avoids magic numbers.

The values are fixed at:

- `NUMOFBYTES` = 1 (the attribute that indicates the number of bytes following in the object)
- `COLOUR_MAP` = 2 (the attribute that stores the object identifier of a colour map, or NULL)
- `COLOUR_PALETTE` = 3 (the attribute that stores the object identifier of a colour palette, or NULL)

Technical Features

- **Data Type:** All constants are of type `USINT` (unsigned short integer, 8-bit).
- **Initial Values:** Explicitly set to `USINT#1`, `USINT#2`, and `USINT#3` respectively, ensuring deterministic behaviour.
- **Package:** Declared in `isobus::UT::Q::const::AID`, which organizes the constant group under a dedicated namespace.

- **Compatibility:** Conforms to the IEC 61499 standard for global constants, making it usable in 4diac-IDE and other compliant environments.

State Overview

This block contains no state variables and does not maintain any runtime state. The constants are always available and immutable during the execution of the application. Therefore, no state diagram is applicable, and the behaviour is consistent throughout the entire runtime.

Application Scenarios

- **ISOBUS Object Pool Management:** When a working set special controls object is created or modified, the corresponding attribute IDs can be set using these constants, e.g., `AID_WSSC.NUMOFBYTES` to specify the length of the object.
- **Colour Configuration:** The `COLOUR_MAP` and `COLOUR_PALETTE` constants are used to assign colour resources to working set special controls, enabling a consistent visual representation on the virtual terminal.
- **Protocol Message Construction:** In communication tasks, these constants help to construct or parse messages that carry attribute information, improving code readability and reducing the risk of logical errors.

Comparison with Similar Blocks

Similar global constant definitions exist for other ISOBUS object types, such as:

- `AID_WSC` (Working Set Controls)
- `AID_VC` (Virtual Terminal Controls)
- `AID_IS` (Input/Output Objects)

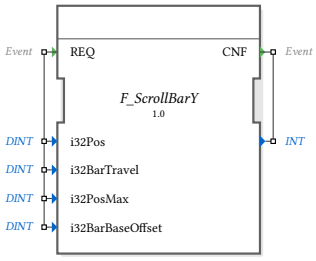
While those also provide numeric attribute IDs as constants, `AID_WSSC` specifically targets the *Working Set Special Controls* object. Its constants are tailored to the attributes defined for that object type, making it distinct in purpose and usage. In contrast to ordinary function blocks, this global constants block does not execute any logic and does not participate in data or event flow; it simply offers named integer values.

Conclusion

The `AID_WSSC` global constants block is a simple but essential component for IEC 61499 and ISOBUS applications. By encapsulating attribute identifiers in named constants, it enhances code maintainability and reduces the chance of errors when dealing with working set special controls. Its integration into the 4diac-IDE environment follows standard practice and allows for direct reuse across projects. As a static definition, it serves as a reliable reference for developers implementing ISOBUS-compliant functionality.

helpers

F_ScrollBarY



• – ◦ • – ◦ • – ◦ •

Introduction

The `F_ScrollBarY` function computes, from the current scroll position (in rows), the Y offset for the scrollbar content container (`Container_Scrollbar_Content.Top`), so the bar "thumb" proportionally reflects the list position. It's the scrollbar counterpart to F_ScrollListY and is used internally by the `ScrollFS` scroll block (package `isobus::UT::Q`).

Interface Structure

Event Inputs

- `REQ`: Calculation request, reads `i32Pos`, `i32BarTravel`, `i32PosMax` and `i32BarBaseOffset`.

Event Outputs

- `CNF`: Confirms the calculation and delivers the return value (unnamed `OutputVar`, addressed via `<InstanceName>.` with a trailing dot).

Data Inputs

- `i32Pos` (DINT): Current scroll position in rows (0…`i32PosMax`).
- `i32BarTravel` (DINT): Available travel distance of the thumb (viewport height − thumb height).
- `i32PosMax` (DINT): Maximum scroll position (last valid row).
- `i32BarBaseOffset` (DINT): `Top` value of `Container_Scrollbar_Content` at position 0 — must match the thumb's fixed internal `Top` position in magnitude, with the opposite sign, so the thumb lands at net Y = 0 when `i32Pos = 0`.

Data Outputs

- *(unnamed)* (INT): `Container_Scrollbar_Content.Top` value (`s16Yposition` for `Q_ChildPosition`).

Adapters

No adapters available.

Functionality

The calculation is a single formula:
`OUT := DINT_TO_INT(i32BarBaseOffset + (i32Pos * i32BarTravel) / i32PosMax)`
At `i32Pos = 0`, `OUT = i32BarBaseOffset` (thumb at the start of the list); at `i32Pos = i32PosMax`, `OUT = i32BarBaseOffset + i32BarTravel` (thumb at the end of the list). Everything in between is linearly interpolated.

Technical Details

- **`DINT_TO_INT` cast**: same as `F_ScrollListY` — the calculation runs in `DINT`, the result is only converted to `INT` at the end (`Q_ChildPosition.s16Yposition`).
- **Integer division**: `(i32Pos * i32BarTravel) / i32PosMax` rounds down (integer division) — with a small `i32BarTravel` relative to `i32PosMax`, the thumb can stay put for several position steps before visibly moving. Unnoticeable in practice, since `i32BarTravel` is typically much larger than `i32PosMax`.
- **Concretely typed, no `ANY`**: like `F_ScrollListY` — fixed `DINT`/`INT` typing, no `MOVE` anchor needed when wiring it up.

State Overview

Stateless — every `REQ` recomputes the result immediately from the current input values.

Application Scenarios

- Used exclusively inside `ScrollFS` to recompute `Container_Scrollbar_Content.Top` on every position change — running in parallel with `F_ScrollListY` for the list itself, sharing the same `i32Pos` value as its source.

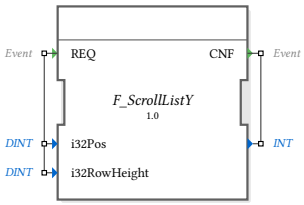
⚖️ Comparison with Similar Blocks

- **Versus `F_ScrollListY`**: same basic idea (row position to pixel offset), but scaled to the (smaller) scrollbar span instead of a direct multiplication by the row height, plus a base offset. See F_ScrollListY.

Conclusion

The scrollbar counterpart to `F_ScrollListY` — the same row position drives both the list and the scrollbar through two separate, concretely typed formulas, each with its own scaling.

F_ScrollListY



• — ◦ • — ◦ • — ◦ •

Introduction

The `F_ScrollListY` function computes, from the current scroll position (in rows) and the row height (in pixels), the Y offset that `Container_Scrolling_Content.Top` needs so the correct row appears inside the viewport. It's a pure calculation helper for the `ScrollFS` scroll block (package `isobus::UT::Q`) and is used internally there.

Interface Structure

Event Inputs

- `REQ`: Calculation request, reads `i32Pos` and `i32RowHeight`.

Event Outputs

- `CNF`: Confirms the calculation and delivers the return value (unnamed `OutputVar`, addressed via `<InstanceName>.` with a trailing dot — the IEC 61131-3 convention for a function's own return value).

Data Inputs

- `i32Pos` (DINT): Current scroll position in rows (0…`i32PosMax`).
- `i32RowHeight` (DINT): Row height in pixels.

Data Outputs

- *(unnamed)* (INT): `Container_Scrolling_Content.Top` value (`s16Yposition` for `Q_ChildPosition`).

Adapters

No adapters available.

Functionality

The calculation is a single formula:
`OUT := DINT_TO_INT(-(i32Pos * i32RowHeight))`
The larger `i32Pos` gets, the more negative `Top` becomes — the content container shifts up, later rows scroll into the viewport. At `i32Pos = 0`, `Top = 0` (start of the list, no offset).

Technical Details

- **`DINT_TO_INT` cast**: The internal calculation runs in `DINT` (32-bit); the result is only converted to `INT` (16-bit) at the end, because `Q_ChildPosition.s16Yposition` expects an `INT` (ISO 11783-6 F.16, "Change Child Location").
- **Concretely typed, no `ANY`**: Unlike generic blocks such as `F_MOVE` or `F_SEL`, all inputs/outputs are fixed-typed (`DINT`/`INT`) — so wiring it inside `ScrollFS` needs no `MOVE` anchor, even though the value originates from other generic/struct fields.

State Overview

Stateless — every `REQ` recomputes the result immediately from the current input values.

Application Scenarios

- Used exclusively inside `ScrollFS` to recompute `Container_Scrolling_Content.Top` on every position change.

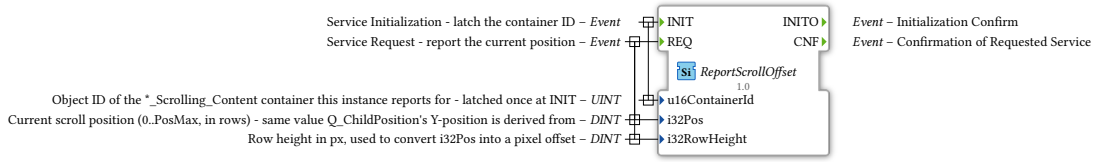
⚖️ Comparison with Similar Blocks

- **Versus `F_ScrollBarY`**: both compute a Y offset from the scroll position, but `F_ScrollBarY` additionally scales it to the (smaller) scrollbar span and applies an offset — see F_ScrollBarY.

Conclusion

A small, robust calculation function — responsible for exactly one conversion (row position to list pixel offset), concretely typed and side-effect free.

ReportScrollOffset



• – ○ • – ○ • – ○ •

Introduction

`ReportScrollOffset` is a helper Service Interface block from package `isobus::UT::Q::helpers`. It sits inside the ScrollFS FBNetwork structure (parallel to `MoveList` and `ListY`) and reports the current scroll offset (row index `i32Pos` × row height `i32RowHeight` in pixels) for container `u16ContainerId` to the ECU visibility gate (`VtMaskVisibility`).

Interface Structure

Event Inputs

- `INIT`: Service interface initialization (`With u16ContainerId`).
- `REQ`: Reports a position change (`With i32Pos`, `With i32RowHeight`).

Event Outputs

- `INITO`: Initialization confirmation.
- `CNF`: Position report confirmation.

Data Inputs

- `u16ContainerId` (UINT): Object ID of the `*_Scrolling_Content` container.
- `i32Pos` (DINT): Current row position (0…i32PosMax).
- `i32RowHeight` (DINT): Row height in pixels for pixel offset conversion.

Data Outputs

No data outputs.

Adapters

No adapters available.

Functionality

1. At `INIT`, latches the container ID `u16ContainerId`.
2. On each `REQ` event, calculates `PixelOffset := i32Pos * i32RowHeight` and forwards it to the ECU visibility gate (`VtMaskVisibility_OnScroll`).
3. The ECU gate uses this information to restrict VT updates (e.g. `Q_*` commands) to rows currently visible on screen.

See Also

- ScrollFS
- ScrollFS_PHYS_Softkey
- ScrollFS_PHYS_Button

🔌 io

📺 Video

- Automation of the Future
- The Great Migration
- Digital Decisions
- Ideal Diodes
- The Dual Wi-Fi Solution

🎧 Podcast

- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Target Groups, Structure, and Unique Selling Proposition
- Taming the Software Dragon: Industrial Automation and the Future of Production
- Eclipse 4diac: Innovation Engine Research vs. User Needs – What Really Drives Development?
- Taming the Software Dragon: Why IEC 61499 Could Revolutionize Industry

---

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Diode & Semiconductor Basics on ms-muc-docs.de
- 🌐 Smart Switch vs. Relay on ms-muc-docs.de

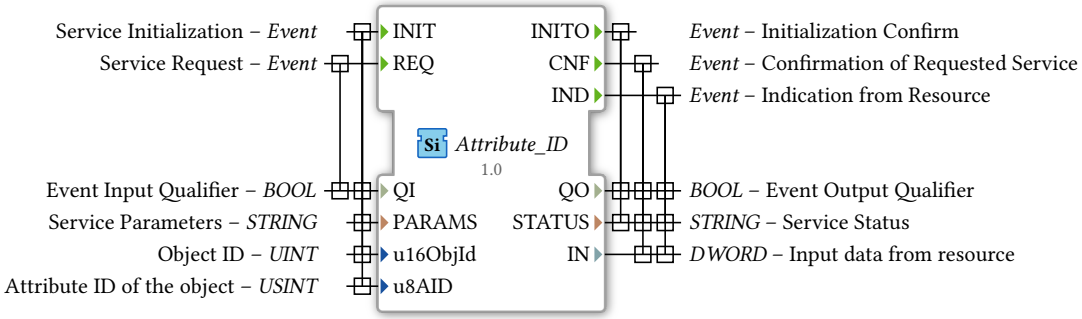
Attribute

Attribute_ID

• – ◦ • – ◦ • – ◦ •

Introduction

`Attribute_ID` is an input service interface function block for double word input data (`DWORD`). It reads an attribute of a VT object (object ID `u16ObjId`, attribute ID `u8AID`) and provides its value as `DWORD` at output `IN`.



Interface Structure

Event Inputs

- **INIT** (`EInit`): service initialisation
  - Associated with: `QI`, `PARAMS`, `u16ObjId`, `u8AID`
- **REQ**: service request
  - Associated with: `QI`

Event Outputs

- **INITO** (`EInit`): initialisation confirmation
  - Associated with: `QO`, `STATUS`
- **CNF**: confirmation of the requested service
  - Associated with: `QO`, `STATUS`, `IN`
- **IND**: indication from the resource

- Associated with: `QO`, `STATUS`, `IN`

Data Inputs

- **QI** (`BOOL`): event input qualifier
- **PARAMS** (`STRING`): service parameters
- **u16ObjId** (`UINT`): object ID, initial value `ID_NULL`
- **u8AID** (`USINT`): attribute ID of the object

Data Outputs

- **QO** (`BOOL`): event output qualifier
- **STATUS** (`STRING`): service status
- **IN** (`DWORD`): input data from the resource

Adapters

No adapter interfaces available.

Functionality

The function block is a service interface FB without an internal implementation in the type (interface declaration). `INIT` initialises the service with `QI`, `PARAMS`, `u16ObjId` and `u8AID` and is confirmed with `INITO`. `REQ` issues a request that is confirmed with `CNF`; `IND` signals an indication from the resource. In all three cases `QO`, `STATUS` and (for `CNF`/`IND`) the value `IN` are output.

Technical Details

- Uses the data type `DWORD` (32 bit) for the attribute value
- The attribute ID to read is given via the data input `u8AID`
- The object ID is preset with `ID_NULL` (`isobus::UT::Q::const::IDs::ID_NULL`)
- Originally created by fortiss GmbH (Monika Wenger, 2017) as an input service interface

Application Scenarios

- Reading attribute values (e.g. attribute ID of an object in the VT) in ISOBUS applications

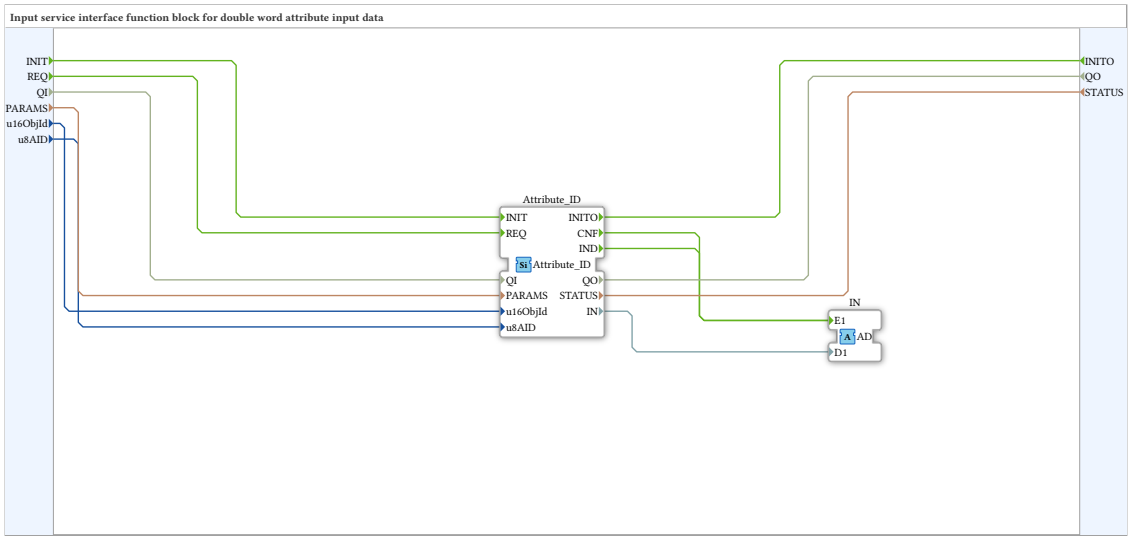
⚖ Comparison with Similar Building Blocks

- NumericValue_ID has the same service interface but reads the numeric value of an object and has no `u8AID` input.

Conclusion

`Attribute_ID` delivers the attribute value of a VT object as `DWORD` through a service interface.

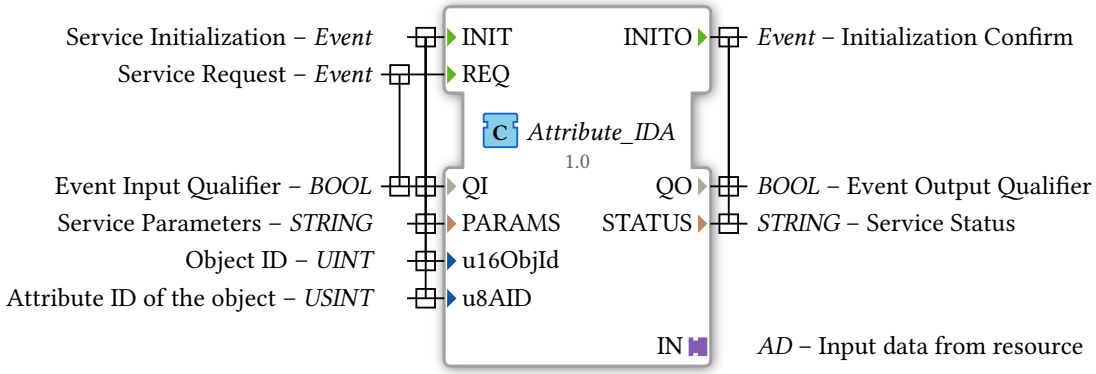
Attribute_IDA



• – ○ • – ○ • – ○ •

Introduction

`Attribute_IDA` is an input service interface function block for double word attribute data (`DWORD`). It encapsulates `Attribute_ID` and provides the read attribute value of a VT object (object ID `u16ObjId`, attribute ID `u8AID`) not via a data output but via the adapter plug `IN` (`AD`). Event and value are bundled in one adapter connection.



Function Blocks (FBs) Used

Sub-blocks: Attribute_IDA

- **Type**: Composite function block
- **Internal FBs used**:
  - **Attribute_ID**: `isobus::UT::io::Attribute::Attribute_ID` - reads the attribute of the VT object
- **Import**: `isobus::UT::Q::const::IDs::ID_NULL`

Interface Structure

Event Inputs

- **INIT** (`EInit`): Service initialization
  - Associated with: `QI`, `PARAMS`, `u16ObjId`, `u8AID`
- **REQ**: Service request
  - Associated with: `QI`

Event Outputs

- **INITO** (`EInit`): Initialization confirmation
  - Associated with: `QO`, `STATUS`

Data Inputs

- **QI** (`BOOL`): Event input qualifier
- **PARAMS** (`STRING`): Service parameters
- **u16ObjId** (`UINT`, initial value `ID_NULL`): Object ID
- **u8AID** (`USINT`): Attribute ID of the object

Data Outputs

- **QO** (`BOOL`): Event output qualifier
- **STATUS** (`STRING`): Service status

Adapters

- **IN** (plug): `adapter::types::unidirectional::AD` - input data from the resource (event `E1`, data `D1` with the `DWORD` attribute value)

Program Flow and Connections

- `INIT` ⟶ `Attribute_ID.INIT`; `REQ` ⟶ `Attribute_ID.REQ`; `Attribute_ID.INITO` ⟶ `INITO`
- `Attribute_ID.IND` ⟶ `IN.E1` and `Attribute_ID.CNF` ⟶ `IN.E1`
- `u16ObjId`, `u8AID`, `PARAMS`, `QI` ⟶ like-named inputs of `Attribute_ID`
- `Attribute_ID.IN` ⟶ `IN.D1`
- `Attribute_ID.STATUS` ⟶ `STATUS`; `Attribute_ID.QO` ⟶ `QO`

Technical Details

- Both `IND` and `CNF` of the inner block trigger `IN.E1`; the value is present at `IN.D1` in each case
- The adapter bundles event and data flow in one connection line
- Version 1.0 (2026-09-15)

Application Scenarios

- Reading attribute values of a VT object and further processing via `AD` adapters, e.g. via `AD_TO_AX` converters

⚖️ Comparison with Similar Blocks

- Attribute_ID provides `IN` as a data output and `CNF/IND` as event outputs, whereas `Attribute_IDA` bundles both in the adapter `IN`.

Summary

`Attribute_IDA` provides the attribute value of a VT object as a `DWORD` via an `AD` adapter.

Auxiliary

IN

🎧 Podcast

- 4diac IDE: Your “Hello World” of Automation – The Blinking Tutorial Local
- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Target Groups, Structure, and Unique Selling Proposition Sharpening
- Taming the Software Dragon: Industrial Automation and the Future of Production

📺 Video

- Decoding the Datasheet
- Soldering Like a Pro Profi
- Via in Pad Plated Over

🌐 Related topic subpages on ms-muc-docs.de

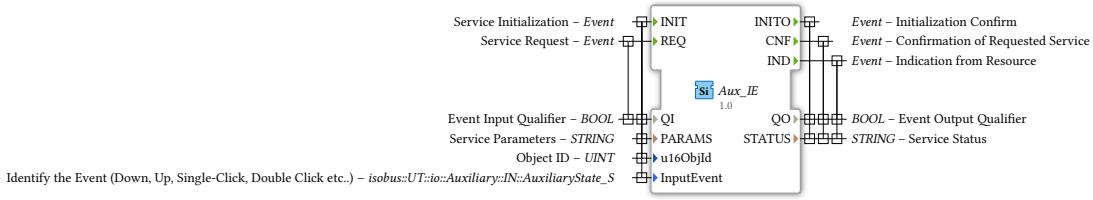
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Aux_IE

• – ◦ • – ◦ • – ◦ •

Introduction

The Aux_IE function block is an input service interface function block for event input data. It is used to process auxiliary inputs and enables the recognition of various input events such as pressing, releasing, single-clicking, double-clicking, etc.



Interface Structure

Event Inputs

- **INIT**: Service Initialization
- Linked to: QI, PARAMS, u16ObjId, InputEvent
- **REQ**: Service Request
- Linked to: QI

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Service Request
- Linked to: QO, STATUS
- **IND**: Display of Resource
- Linked to: QO, STATUS

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service parameters
- **u16ObjId** (UINT): Object ID, initial value: ID_NULL
- **InputEvent** (AuxiliaryState_S): Identifies the event (Down, Up, Single-Click, Double-Click, etc.), initial value: Invalid

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (STRING): Service status

Adapters

No adapter interfaces are available.

Functionality

The Aux_IE block processes auxiliary input events and provides a standardized interface for communication with input devices. During initialization (INIT), the service parameters and object ID are configured. Service requests can be made via REQ events, while IND events indicate incoming signals from the resource.

Technical Features

- Uses specific ISOBUS data types for agricultural machinery
- Supports various input event types via the AuxiliaryState structure
- Initialization with standardized NULL values (ID_NULL)
- String-based status feedback

State Overview

The block has an initialization state (INIT/INITO) and operational states for service requests (REQ/CNF) as well as resource indicators (IND).

Application Scenarios

- Control of agricultural machinery via auxiliary inputs
- Processing of user interactions in ISOBUS systems
- Recognition of complex input patterns (single click, double click)
- Integration into control systems for agricultural machinery

⚖️ Comparison with Similar Blocks

Compared to simple digital input blocks, Aux_IE offers enhanced functionality for recognizing more complex input events and is specifically optimized for ISOBUS environments.

🛠️ Related exercises

- Uebung_010b3
- Uebung_010b3_AX
- Uebung_010bA2
- Uebung_010bA2_AX
- Uebung_010bA3
- Uebung_010bA3_AX
- Uebung_010bA4
- Uebung_010bA4_AX

Conclusion

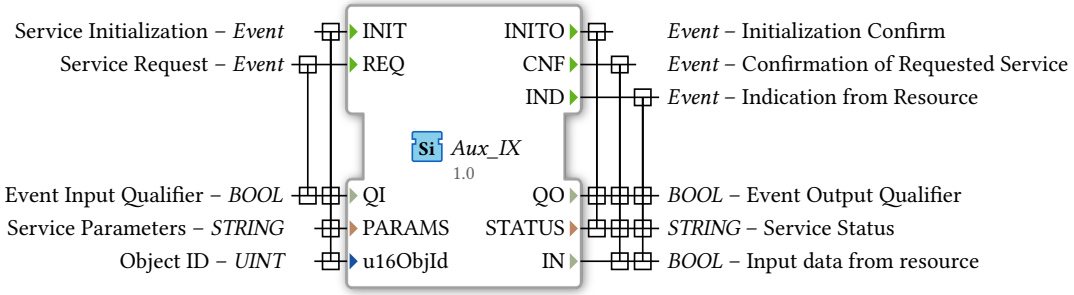
The The Aux_IE function block provides a powerful interface for processing auxiliary input events in ISOBUS-based control systems and enables the reliable detection of various user interactions.

Aux_IX

• – ◦ • – ◦ • – ◦ •

Introduction

The Aux_IX function block is a service interface function block for Boolean input data. It serves as an interface for input services and enables communication with resources for querying digital input signals.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the function block with the parameters QI, PARAMS, and u16ObjId
- **REQ**: Service Request - Triggers a service request with the qualifier QI

Event Outputs

- **INITO**: Initialization Acknowledgement - Confirms the initialization with QO and STATUS
- **CNF**: Acknowledgement of Requested Service Request - Returns QO, STATUS, and IN data
- **IND**: Resource Indication - Signals data availability with QO, STATUS, and IN data

Data Inputs

- **QI**: BOOL - Event Input Qualifier
- **PARAMS**: STRING - Service Parameter
- **u16ObjId**: UINT - Object ID with initial value ID_NULL

Data Outputs

- **QO**: BOOL - Event Output Qualifier
- **STATUS**: STRING - Service Status
- **IN**: BOOL - Input data from the resource

Adapter

No adapter interfaces are available.

Functionality

The Aux_IX function block operates as an input service interface for Boolean data. During initialization (INIT), the service parameters and the object ID are configured. Service requests (REQ) trigger the querying of input data from the associated resource. The data is returned via the CNF and IND events, with the STATUS output providing information about the success or failure of the operation.

Technical Features

- Uses specific TypeHash identification for 4diac
- Initializes the object ID with a default value (ID_NULL)
- Supports configurable service configuration via STRING parameters
- Provides both request-based (CNF) and event-based (IND) data output

State Overview

The function block goes through the following states:

1. **Not Initialized**: Before INIT processing
2. **Initialized**: After successful INIT processing with INITO confirmation
3. **Ready**: Can process REQ requests
4. **Active**: During data retrieval from the resource

Application Scenarios

- Querying digital input signals in control systems
- Integrating hardware inputs into IEC 61499 applications
- Monitoring switch states and sensors
- Implementing standardized input interfaces in distributed systems

⚖️ Comparison with Similar Function Blocks

In Compared to simple digital input blocks, Aux_IX offers enhanced service functionality with configurable settings, status feedback, and both request-driven and event-driven data provisioning.

🛠️ Related Exercises

- Exercise_010b1

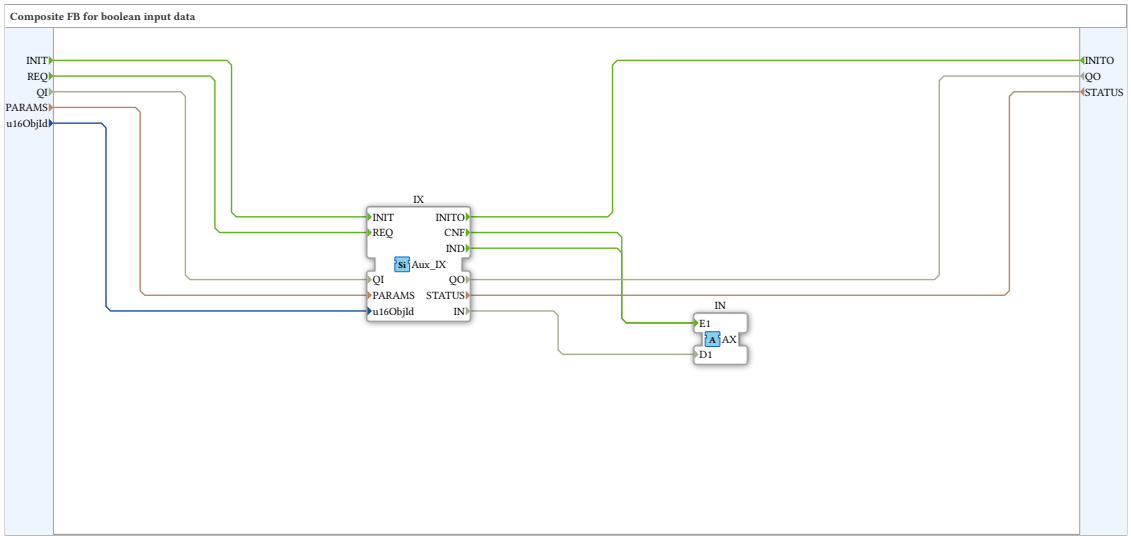
Conclusion

The Aux_IX function block provides a robust and flexible solution for integrating Boolean input data into IEC 61499-based control systems. Its service-oriented architecture and comprehensive status feedback make it particularly suitable for demanding industrial applications with high requirements for reliability and diagnostic capabilities.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

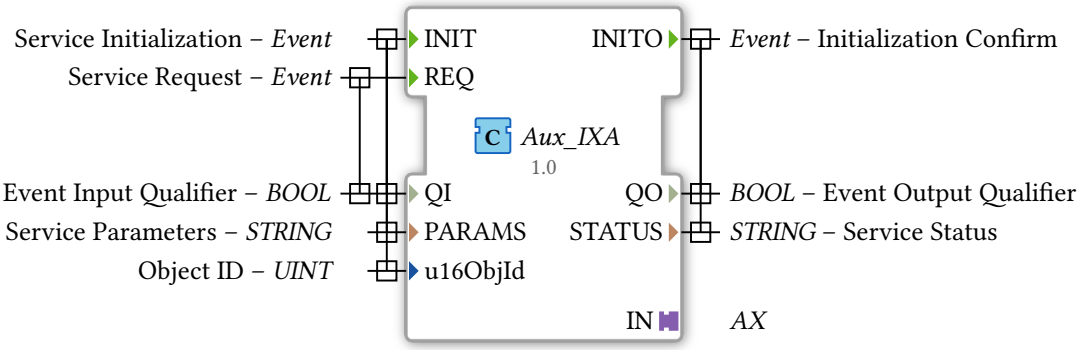
Aux_IXA



• – ○ • – ○ • – ○ •

Introduction

The Aux_IXA is a composite function block for processing Boolean input data. It serves as a wrapper for the basic function block Aux_IX and provides an extended interface for integration into ISOBUS-compatible systems.



Interface Structure

Event Inputs

- **INIT**: Service initialization with the associated data QI, PARAMS, and u16ObjId
- **REQ**: Service request with the associated data QI

Event Outputs

- **INITO**: Initialization confirmation with the associated data QO and STATUS

Data Inputs

- **QI**: Event Input Qualifier (BOOL) - Qualifier for event inputs
- **PARAMS**: Service Parameters (STRING) - Configuration parameters for the service
- **u16ObjId**: Object ID (UINT) - Object identifier with initial value ID_NULL

Data Outputs

- **QO**: Event Output Qualifier (BOOL) - Qualifier for event outputs
- **STATUS**: Service Status (STRING) - Service status information

Adapter

- **IN**: Unidirectional AX-type adapter for data communication

Functionality

The Aux_IXA acts as a composite block that encapsulates the basic function block Aux_IX. Upon an INIT event, all parameters are forwarded to the internal IX block. REQ events trigger service requests, while INITO confirms successful initialization. The IN adapter enables bidirectional communication with external components.

IN

Technical Features

- Integration into ISOBUS environments through a special package structure
- Use of TypeHash for type identification
- Unidirectional adapter for standardized interfaces
- Initialization with a NULL object ID as the default value

State Overview

The block goes through the following states:

1. **Not Initialized**: Before the INIT event
2. **Initialization**: During INIT processing
3. **Ready**: After successful initialization (INITO)
4. **Service Mode**: During REQ event processing

Application Scenarios

- ISOBUS-compatible agricultural machinery control
- Boolean signal processing in agricultural technology systems
- Wrapper implementation for existing I/O functionality

- Parameter configuration for auxiliary inputs

⚖️ Comparison with Similar Blocks

Compared to simple Boolean input blocks, Aux_IXA offers:

- Extended parameterization options
- ISOBUS specification
- Adapter-based interface
- Composite structure for improved reusability

🛠️ Related exercises

- Exercise_010b1_AX

Conclusion

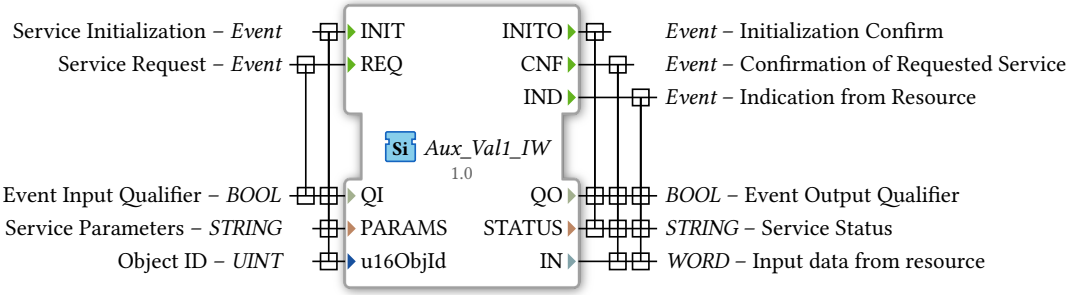
The Aux_IXA represents a robust composite function block for Boolean input data processing in ISOBUS environments. Its clear interface definition and adapter integration enable flexible integration into complex agricultural control systems.

Aux_Val1_IW

• – ◦ • – ◦ • – ◦ •

Introduction

The Aux_Val1_IW function block is an input service interface function block for word input data. It serves as an interface between the control logic and external resources for processing 16-bit word input data in an ISOBUS context.



Interface Structure

Event Inputs

- **INIT**: Service Initialization
- Linked to: QI, PARAMS, u16ObjId
- **REQ**: Service Request
- Linked to: QI

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Service Request
- Linked to: QO, STATUS, IN
- **IND**: Resource Indication
- Linked to: QO, STATUS, IN

Data Inputs

- **QI**: Event Input Qualifier (BOOL)
- **PARAMS**: Service Parameters (STRING)
- **u16ObjId**: Object ID (UINT) Initial value: ID_NULL

Data Outputs

- **QO**: Event output qualifier (BOOL)

- **STATUS**: Service status (STRING)
- **IN**: Input data from the resource (WORD)

Adapter

No adapter interfaces are available.

Functionality

This function block enables communication with external resources for capturing Word input data. The INIT initialization configures the service parameters and the object ID. The REQ request triggers data collection, while IND events signal asynchronous data updates from the resource.

Technical Features

- Uses WORD data type for 16-bit data acquisition
- ISOBUS-compatible interface
- Supports asynchronous data indications
- Initialization with configurable parameters
- Error status feedback via STATUS output

State Overview

The function block goes through the following main states:

1. **Not Initialized**: Waiting for INIT event
2. **Initialized**: Ready for REQ requests
3. **Active**: Processing data requests and indications
4. **Error**: Status message in case of problems

Application Scenarios

- ISOBUS-based agricultural machinery control
- Acquisition of 16-bit sensor data
- Communication with external I/O modules
- Automation systems in agricultural technology

⚖️ Comparison with Similar Blocks

Compared to simple input blocks, Aux_Val1_IW offers extended ISOBUS functionality with configurable initialization, status feedback, and asynchronous Indication mechanisms.

Conclusion

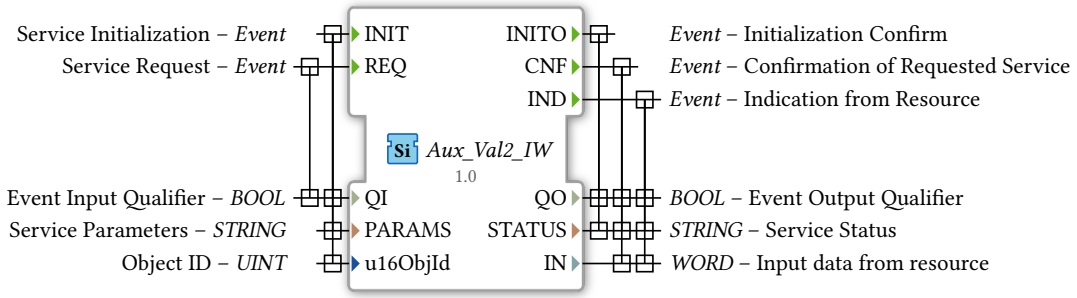
The Aux_Val1_IW function block provides a robust and flexible solution for capturing word input data in ISOBUS environments, with comprehensive error handling and configurable service parameters.

Aux_Val2_IW

• – ◦ • – ◦ • – ◦ •

Introduction

The Aux_Val2_IW is an input service interface function block for word input data. This block serves as an interface for communication with resources and enables the processing of 16-bit word input data in distributed automation systems.



Interface Structure

Event Inputs
- **INIT**: Service Initialization - Initializes the function block with parameters
- **REQ**: Service Request - Triggers a service request

Event Outputs
- **INITO**: Initialization Acknowledgement - Confirms successful initialization
- **CNF**: Confirmation of Requested Service - Acknowledges a service request
- **IND**: Indication from Resource - Signals incoming data from the resource

Data Inputs
- **QI** (BOOL): Event Input Qualifier - Enables/disables processing
- **PARAMS** (STRING): Service Parameters - Configuration parameters for the service
- **u16ObjId** (UINT): Object ID - Identifier for the data object (initial value: ID_NULL)

Data Outputs
- **QO** (BOOL): Event output qualifier - Output processing status
- **STATUS** (STRING): Service status - Service status feedback
- **IN** (WORD): Input data from the resource - The received 16-bit word data

Adapters
No adapter interfaces are available.

Functionality
This function block acts as a service interface for word input data. During INIT initialization, the service parameters and object ID are configured. Service requests can be made via REQ events, which are acknowledged with CNF events. When data is received from the resource, an IND event is issued containing the received word data.

Technical Features
- Processes 16-bit Word data types (WORD)
- Supports configurable object IDs for data identification
- Provides comprehensive status feedback on the SERVICE status
- Enables parameterizable service configuration

State Overview
The function block goes through the following states:

1. **Not Initialized**: Before INIT initialization
2. **Initialized**: After successful INIT processing with INITO confirmation
3. **Ready**: For REQ requests and IND indications
4. **Service Active**: During the processing of service requests

Application Scenarios
- Reading 16-bit sensor data in distributed systems
- Communication with ISOBUS-compatible devices
- Input of Word data in automation applications
- Interface to external resources for data acquisition

⚖️ Comparison with Similar Blocks
Compared to simpler input blocks, it offers Aux_Val2_IW offers extended service functionality with configurable settings, status feedback, and object identification. It is specifically designed for processing 16-bit word data.

Conclusion
The Aux_Val2_IW function block provides a powerful service interface for processing word input data in industrial automation systems. Its extensive configurability and status feedback make it particularly suitable for demanding applications with high requirements for reliability and diagnostic capabilities.

AuxiliaryState



• – ◦ • – ◦ • – ◦ •

Introduction

AuxiliaryState is a collection of global constants for auxiliary inputs in ISOBUS systems. These constants define various states of auxiliary inputs, typically used for control functions in agricultural and mobile machinery.

Interface Structure

AuxiliaryState consists of global constants and is not a function block in the traditional sense. Therefore, this component does not have any event or data interfaces.

Event Inputs

- Not available (Global Constants)

Event Outputs

- Not available (Global Constants)

Data Inputs

- Not available (Global Constants)

Data Outputs

- Not available (Global Constants)

Adapters

- Not available

Functionality

AuxiliaryState defines a set of predefined constants for the data type `AuxiliaryState_S`. Each constant represents a specific state of an auxiliary input with a corresponding numeric value and optional additional parameters.

Technical Features

- All constants are of type `isobus::UT::io::Auxiliary::IN::AuxiliaryState_S`
- The value ranges from 0 to 255
- Some constants also contain the flag `bCyclic := FALSE`
- The structure follows ISOBUS standards for agricultural control systems

State Overview

The defined constants include the following states:

Constant	Value	Description
`AuxDisabled`	0	Off - reverse, down, left, or not pressed
`AuxDisabled_START`	0	Off - reverse, down, left, or not pressed (with bCyclic=FALSE)
`AuxEnabled`	1	On - forward, up, right, or pressed
AuxEnabled_START`	1	On - forward, up, right, or pressed (with bCyclic=FALSE)
AuxHeld`	2	Held (forward, up, or right)
AuxHeld_START`	2	Held (forward, up, or right) (with bCyclic=FALSE)
AuxEnabledBackwards`	4	Currently on - backward, down, or left

AuxHeldBackwards`	8	Held (backward, down, or left)
AuxHeldBackwards_START`	8	Held (backward, down, or left) (with bCyclic=FALSE)
`AuxDefault`	9	Default value
`Invalid`	255	Invalid value

Application Scenarios

- Control of hydraulic functions in tractors
- Operation of implements via ISOBUS interfaces
- Status monitoring of auxiliary functions in mobile machinery
- Implementation of standardized input states in agricultural control systems

⚖️ Comparison with similar blocks

Unlike function blocks with active logic, AuxiliaryState consists of passive constant definitions that can be used as reference values in other function blocks. They offer a standardized way to define auxiliary states in ISOBUS environments.

Conclusion

AuxiliaryState provides a comprehensive collection of standardized constants for handling auxiliary inputs in ISOBUS systems. The clear definition of the different states enables a consistent implementation of control logic across different systems and facilitates the maintainability and readability of the code.

AuxiliaryState_S

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `AuxiliaryState_S` is a structured data structure (struct) that represents the state of an auxiliary input. It is used in the context of the ISOBUS interface (ISO 11783) for communication in agricultural and forestry machinery. This data type encapsulates both the actual value of the input and control information for its cyclic transmission.

Interface Structure

AuxiliaryState_S` is a structured data type (struct) and therefore does not have event or adapter interfaces in the conventional sense of a function block. Its structure consists of the following two data fields:

Data Inputs

(Not applicable – This is a data type, not a function block)

Data Outputs

(Not applicable – This is a data type, not a function block)

Structured Data Fields

Name	Data Type	Initial Value	Comment
`value`	`USINT` (0…255)	255	Represents the value of the auxiliary input.
`bCyclic`	`BOOL`	`TRUE`	Controls the cyclic transfer. When set to `FALSE`, the message is not repeated every 200 ms.

Functionality

This data type serves as a container for information about an auxiliary input. The field `value` stores the actual, read value. The initial value of 255 can represent an invalid or undefined state. The field `bCyclic` is a control flag that influences the transmission behavior in an ISOBUS network. According to the comment, when `bCyclic = TRUE` is set, the corresponding message is sent cyclically every 200 milliseconds by default. If `bCyclic` is set to `FALSE`, this regular repetition is omitted, which can reduce network traffic if the value does not change frequently.

Technical Features

- **ISOBUS Context:** This data type is specifically designed for ISOBUS communication (package `isobus::UT::io::Auxiliary::IN`).
- **Cycle Control:** The integrated cyclic control (`bCyclic`) is a characteristic feature that takes bus communication efficiency into account.
- **Initial Values:** Both fields have defined initial values (`value=255`, `bCyclic=TRUE`), ensuring deterministic initialization.

State Overview

(Not applicable – This is a passive data type without state machines.)

Application Scenarios

This struct is typically used as a data input or output in function blocks (FBs) that process or control ISOBUS auxiliary inputs. Examples include:

- Function blocks (FBs) for reading the state of switches, potentiometers, or sensors configured as auxiliary inputs.
- FBs that process auxiliary input values and forward them to other system components.
- FBs that determine whether a value needs to be updated and sent based on the `bCyclic` flag.

⚖️ Comparison with similar function blocks

As a pure data type, a direct comparison with function blocks is not meaningful. Within the ISOBUS data types, it could be compared to simpler data types such as a pure `USINT` for the value. The advantage of `AuxiliaryState_S` lies in the extension with the meta-information for transmission control (`bCyclic`), which enables higher integration and more efficient communication in the ISOBUS network.

Conclusion

AuxiliaryState_S` is a specialized, structured data type for ISOBUS communication. It offers an elegant way to encapsulate not only the value of an auxiliary input, but also an important protocol attribute for its cyclic transmission. Its use promotes the clarity and maintainability of applications that implement ISOBUS I/O functionalities.

OUT

🎧 Podcast

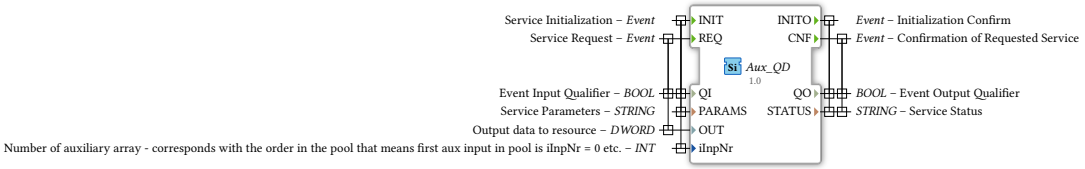
- ISOBUS Output Meter: Mastering Dynamic Displays – From Pointer to Visualization on the VT
- ISOBUS Bar Chart: The Output Linear Bar Graph Object of ISO 11783-6 Decoded
- ISOBUS Terminals: Understanding Numbers – NumberVariable, InputNumber & OutputNumber Explained

Aux_QD

• – ◦ • – ◦ • – ◦ •

Introduction

The Aux_QD function block is an output service interface module for processing double-word output data (DWORD). It serves as an interface for auxiliary output functions and supports various auxiliary types for flexible data output.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **OUT** (DWORD): Output Data for the Resource
- **iInpNr** (INT): Auxiliary Array Number (corresponds to the order in the pool, initial value: 0)

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status Information

Adapters

No adapter interfaces are available.

Functionality

The Aux_QD block processes double-word data (32-bit) and outputs it via auxiliary interfaces. During initialization (INIT), the service parameters are configured. The output data is sent to the corresponding resource via the REQ event. The block supports multiple auxiliary types and allows the parallel output of both values in a single operation.

Technical Features

- Supports Auxiliary Types: 0, 1, 2, 3, 4, 6, 11, 12, 13
- Processes 32-bit DWORD data
- Initial value for iInpNr is 0 (first auxiliary array in the pool)
- Parallel output of both values is possible in one operation

State Overview

The block has two main states:

1. **Initialization State**: Processing the INIT request with parameter configuration
2. **Operating State**: Processing REQ requests with data output

Application Scenarios

- ISOBUS-compatible control systems
- Agricultural machine controls
- Industrial automation systems
- Embedded systems with auxiliary interfaces

⚖️ Comparison with Similar Blocks

Compared to simple output blocks, Aux_QD offers extended functionality for auxiliary interfaces and supports specific ISOBUS-compatible data types. Its parallel output capability distinguishes it from sequential output blocks.

Conclusion

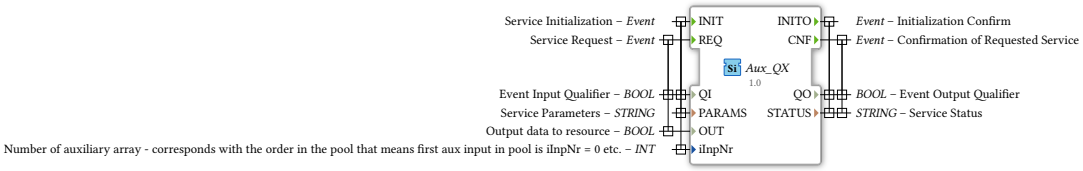
The Aux_QD function block represents a powerful solution for outputting double-word data via auxiliary interfaces. Its ISOBUS compatibility and support for various auxiliary types make it particularly suitable for demanding automation applications in industrial and mobile systems.

Aux_QX

• – ◦ • – ◦ • – ◦ •

Introduction

The Aux_QX function block is an output service interface function block for Boolean output data. It serves as an interface to auxiliary output devices and enables the control of Boolean output signals via a defined service mechanism.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **OUT** (BOOL): Output Data for the Resource
- **iInpNr** (INT): Auxiliary Array Number - corresponds to the order in the pool (the first aux input in the pool has iInpNr = 0, etc.)

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status

Functionality

The The Aux_QX function block implements a standardized service interface for Boolean output data. Upon receiving an INIT event, the service is initialized, using the PARAMS and iInpNr parameters for configuration. The REQ event triggers the output of the Boolean value to the specified auxiliary resource. Each operation is acknowledged by corresponding confirmation events (INITO, CNF).

Technical Features

- Supports addressing auxiliary outputs via the iInpNr parameter
- Flexible parameter configuration via STRING parameters
- Complete status feedback via the STATUS output
- Qualified event handling via QI/QO signals

State Overview

The function block follows a typical service interface pattern with initialization and operating states. After successful initialization (INIT → INITO), service requests (REQ → CNF) can be processed.

Application Scenarios

- Control of auxiliary output devices in ISOBUS environments
- Connection of Boolean actuators in agricultural machinery
- Integration into control systems for auxiliary functions
- Universal Boolean output interface in distributed automation systems

⚖️ Comparison with similar blocks

Compared to simple Boolean output blocks, Aux_QX offers extended service functionality with initialization sequences, parameter configuration, and detailed status management. Specific addressing via iInpNr enables integration into complex device pools. ...

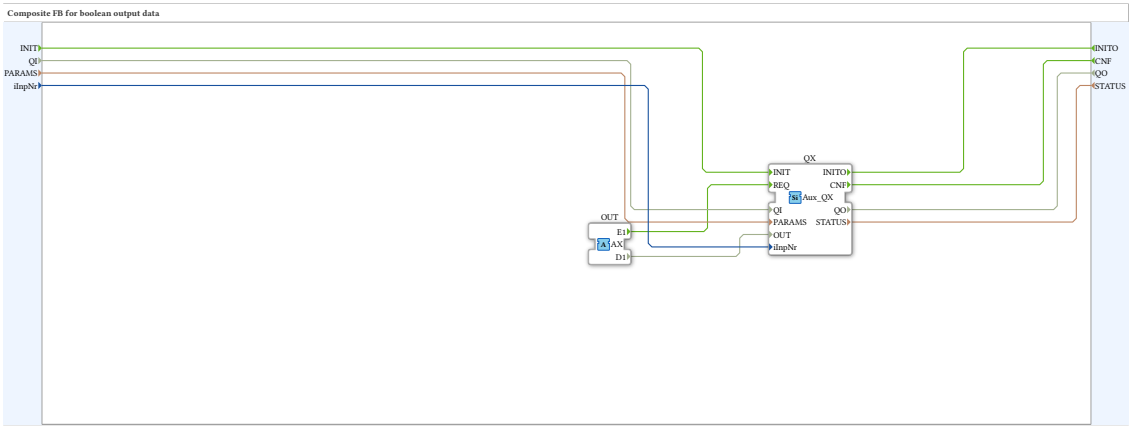
🛠️ Related Exercises

- Exercise_003c_sub

Conclusion

The Aux_QX function block represents a robust and standardized solution for controlling Boolean auxiliary outputs. Its comprehensive service interface and flexible configurability make it particularly suitable for use in professional automation environments with ISOBUS compatibility.

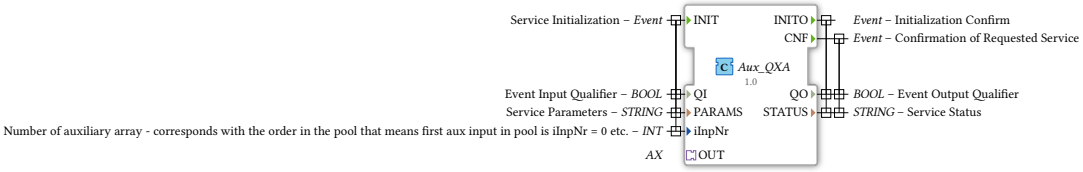
Aux_QXA



• – ○ • – ○ • – ○ •

Introduction

The Aux_QXA is a composite function block for outputting Boolean data. It serves as an interface for auxiliary outputs in ISOBUS-compatible systems and enables the control of Boolean output signals via standardized interfaces.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- Linked to: QI, PARAMS, iInpNr

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Service
- Linked to: QO, STATUS

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameter
- **iInpNr** (INT): Auxiliary Output Array Number (corresponds to the order in the pool, where the first auxiliary input has iInpNr = 0)

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status

Adapter

- **OUT**: Unidirectional adapter of type AX
- Position: x="–400", y="1000"

Functionality

The Aux_QXA function block acts as a wrapper for the internal Aux_QX block. Upon initialization (INIT event), the configuration parameters are passed and the internal block is initialized. Output requests can be sent via the OUT adapter, which are then processed by the internal block. Acknowledgement is provided via the CNF and INITO output events.

Technical Features

- Uses a unidirectional adapter for output communication
- Supports configurable service configuration via STRING parameters
- Allows addressing of multiple auxiliary outputs via the iInpNr parameter
- Implements standardized initialization and acknowledgment protocols

State Overview

The function block goes through the following states:

1. **Not Initialized**: Before the INIT event
2. **Initialization**: During INIT processing
3. **Ready**: After successful initialization (INITO)
4. **Active**: During output processing (CNF)

Application Scenarios

- Control of Boolean actuators in agricultural machinery
- ISOBUS-compatible auxiliary outputs in tractors and implements
- Automated control systems in agricultural technology
- Networked control of peripheral devices

⚖️ Comparison with Similar Blocks

Compared to simple Boolean output blocks, this offers Aux_QXA:

- Extended parameterization via STRING parameters
- ISOBUS compatibility
- Standardized initialization procedures
- Adapter-based interface for extended communication

🛠️ Related Exercises

- Exercise_003c_sub_AX

Conclusion

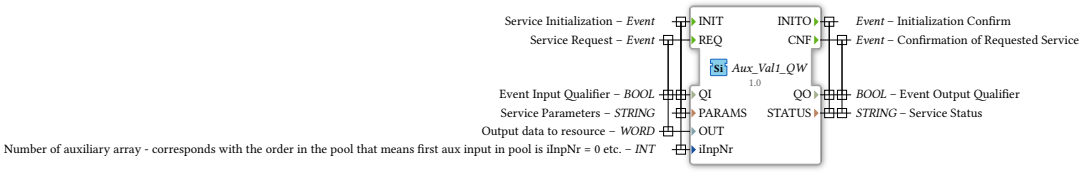
The Aux_QXA function block provides a robust and standards-compliant solution for controlling Boolean auxiliary outputs in ISOBUS systems. Its modular architecture and extended parameterization options make it particularly suitable for complex control tasks in modern agricultural technology.

Aux_Val1_QW

• – ○ • – ○ • – ○ •

Introduction

The Aux_Val1_QW is an output service interface function block for word output data. This block serves as an interface to auxiliary output resources and enables the output of 16-bit data words to external peripheral devices.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event

- **REQ**: Service Request Event

Event Outputs
- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Execution

Data Inputs
- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **OUT** (WORD): Output Data for the Resource
- **iInpNr** (INT): Auxiliary Array Number - corresponds to the order in the pool (the first aux input in the pool has iInpNr = 0, etc.)

Data Outputs
- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status

Adapter

No adapter interfaces available.

Functionality

This function block initializes and controls auxiliary output devices via the ISOBUS UT protocol stack. During initialization (INIT event), the service parameters are configured and the auxiliary device is set up accordingly. Upon service requests (REQ event), the WORD data is sent to the configured output device. The block supports various auxiliary device types and provides status feedback on the success or failure of operations.

Technical Features
- Supports Auxiliary Types: 0, 1, 2, 3, 4, 6, 11, 12, 13
- Uses WORD data type (16-bit) for output data
- Implements ISOBUS UT protocol for auxiliary I/O
- Provides detailed status information via STRING output
- Allows configuration of multiple auxiliary devices via iInpNr parameters

State Overview

The function block has two main operating states:

1. **Initialization State**: Processes the INIT request and configures the auxiliary device
2. **Service State**: Processes REQ requests and outputs data to the configured device

Application Scenarios
- Control of auxiliary output devices in mobile machinery
- ISOBUS-compliant peripheral control
- Agricultural and construction machinery applications
- Connectivity of Word-based output devices via the UT protocol

⚖️ Comparison with Similar Function Blocks

Compared to simple digital output blocks, Aux_Val1_QW offers advanced functionality for ISOBUS environments, including parameterization, status feedback, and support for various auxiliary device types. The WORD data width enables the control of more complex peripheral devices compared to simple BOOL output blocks.

Conclusion

The Aux_Val1_QW function block represents a robust and standards-compliant solution for controlling auxiliary WORD output devices in ISOBUS environments. Its flexible configurability and comprehensive status feedback make it particularly suitable for demanding industrial and mobile applications.

Button

🎧 Podcast

- ISOBUS Button: More than just a click – The standardization of agricultural machinery

ButtonActivationCode



• – ○ • – ○ • – ○ •

Introduction

ButtonActivationCode is a collection of global constants for processing button activations in ISOBUS-compatible systems. These constants define various states and events that can occur when buttons are pressed and are primarily used in agricultural applications.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

No adapters available

Functionality

The ButtonActivationCode constants define various button activation states as structured data types of type `ButtonActivationCode_S`. Each constant represents a specific key state and contains a numeric code as well as optional additional parameters.

Technical Features

- **Transmission Repetition Rate**: On key press/release and every 200 ms while the key is held
- **Lockable Keys**: Do not repeat while the key is held
- **ISOBUS Compatibility**: Specifically designed for ISOBUS Virtual Terminal (VT) systems
- **Structured Constants**: Uses a structured data type with code values and additional flags

State Overview

Constant	Code	Description
BT_RELEASED_UNLATCHED	0	Key has been unlocked or released (state change)
BT_PRESSED_LATCHED	1	Button was “pressed” or locked (state change)
BT_STILL_HELD	2	Button is still being held (lockable buttons do not repeat)
BT_STILL_HELD_START	2	Button is still being held (with bCyclic := FALSE)
BT_ABORTED	3	Button press aborted (applies to VT version 4 and higher)
Invalid	255	Invalid state

Application Scenarios

- Controlling agricultural machinery via ISOBUS Virtual Terminal
- Implementing button logic in operator panels
- Processing button inputs in vehicle control systems
- Developing user interfaces for agricultural applications

⚖️ Comparison with similar function blocks

Compared to simple digital inputs, ButtonActivationCode offers a finely granular state representation for buttons, specifically tailored to the requirements of ISOBUS systems. The constants support advanced button functions such as locking mechanisms and cancel functions.

Conclusion

ButtonActivationCode provides a standardized method for handling button activations in ISOBUS environments. The defined constants enable consistent and reliable processing of button inputs across different systems, which is particularly important in safety-critical agricultural applications.

ButtonActivationCode_S

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `ButtonActivationCode_S` is a structured data structure (struct) used to configure the activation of a button in a control system. It defines an activation code and a control flag for the repeat logic. This data type is part of the `isobus::UT::io::Button` package.

Interface Structure

Since this is a data type and not a function block, there are no event or adapter interfaces. The structure consists exclusively of data fields.

Data Inputs

(Not applicable - This is a data type, not a function block)

Data Outputs

(Not applicable - This is a data type, not a function block)

Structured Data Fields

The following fields form the internal structure of the data type `ButtonActivationCode_S`:

Name	Data Type	Initial Value	Comment
`code`	`USINT`	255	Defines the specific activation code for the button.
`bCyclic`	`BOOL`	TRUE	Controls the cyclic behavior. If set to `FALSE`, activation is not repeated every 200 ms.

Functionality

The structure `ButtonActivationCode_S` serves as a container for configuration data.

- The field `code` uniquely identifies the button within the system. The default value of 255 can be interpreted as an inactive or undefined state.
- The field `bCyclic` determines the timing behavior when the button is continuously pressed. With `TRUE` (default), the corresponding event is repeatedly triggered at a fixed interval (e.g., every 200 ms). With `FALSE`, the event is triggered only once when the button is pressed (and possibly released), even if the button is held down.
•

Technical Features

- **Initial Values:** Both fields of the structure have predefined initial values, ensuring reliable initialization.
- **Comment:** The field `bCyclic` contains a helpful comment that explains the specific effect (`do not repeat every 200ms`) directly in the declaration.

- **Type Hash:** The data type contains an attribute (`eclipse4diac::core::TypeHash`) used for type checking and versioning within the 4diac environment.

State Overview

(Not applicable - This is a static data type without state machines.)

Application Scenarios

This struct is typically used as an input or configuration parameter for function blocks that process button inputs, e.g.:

- In a human-machine interface (HMI) for configuring softkeys.
- – In machine controls for parameterizing physical control buttons.
- Wherever a distinction needs to be made between a single and repeated action upon button press.

⚖️ Comparison with similar function blocks

As a pure data type, `ButtonActivationCode_S` is comparable to other structured data types (STRUCTs) in 4diac, such as `DATE_AND_TIME`. Its specific purpose is to bundle the parameters relevant for button control (`code` for identification, `bCyclic` for timing behavior) into a single, reusable unit.

Conclusion

The data type `ButtonActivationCode_S` is a simple yet essential structure for the standardized configuration of button behavior in 4diac-based control applications. By encapsulating the code and the repetition flag, it promotes code reusability and readability. The integrated initial values and clear comments contribute to robustness and ease of use.

🌐 Related topic subpages on ms-muc-docs.de

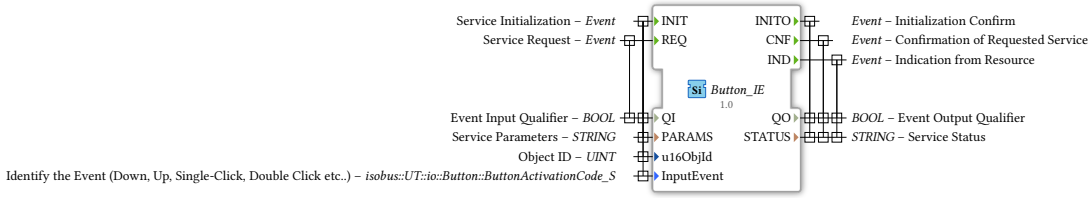
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Button_IE

• – ◦ • – ◦ • – ◦ •

Introduction

The Button_IE function block is an input service interface function block for event input data. It serves as an interface for button events in control systems and enables the processing of various button activities such as pressing, releasing, or multiple clicks.



Interface Structure

Event Inputs

- **INIT**: Service Initialization
- **REQ**: Service Request

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Request
- **IND**: Resource Indication

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **u16ObjId** (UINT): Object ID (Initial Value: ID_NULL)

- **InputEvent** (ButtonActivationCode_S): Identifies the event (Down, Up, Single-Click, Double-Click, etc.) with the initial value “Invalid”

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status

Adapter

No adapter interfaces available.

Functionality

The Button_IE function block processes button events via the ISOBUS-UT protocol. During initialization (INIT), the service parameters and object ID are configured. The block can detect and process various button activities such as single click, double click, press, and release. Incoming events from the hardware resource are reported via the IND output.

Technical Features

- Uses ISOBUS-UT-specific data types for button activation codes
- Supports initialization with specific object IDs
- Provides comprehensive status feedback via the STATUS output
- Implements qualified event processing via QI/QO signals

State Overview

The function block has an initialization state (INIT/INITO) and operational states for service requests (REQ/CNF), as well as asynchronous event indications (IND). The exact state machine depends on the implementation.

Application Scenarios

- Agricultural control systems with ISOBUS compatibility
- Operator panels with push-button inputs
- Machine controls with event-based inputs
- Systems that need to distinguish between different push-button activities

⚖️ Comparison with similar building blocks

Compared to simple digital input blocks, Button_IE offers advanced functionality for push-button-specific events such as multiple clicks and distinguishes between different activation states. Its ISOBUS integration makes it particularly suitable for agricultural applications.

🛠️ Related exercises

- Uebung_010b7
- Uebung_010b7_AX
- Uebung_010b8
- Uebung_010b8_AX
- Uebung_010b9
- Uebung_010b9_AX
- Uebung_010bA
- Uebung_010bA_AX

Conclusion

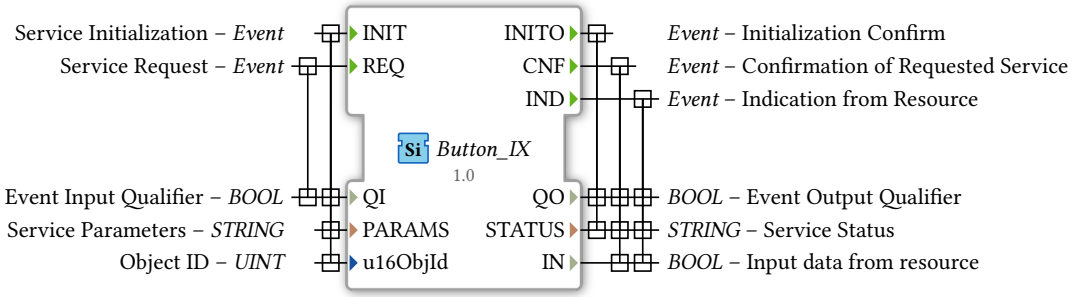
Button_IE is a specialized function block for The processing of button events in ISOBUS environments. Its ability to distinguish between different button activities and provide comprehensive status information makes it particularly suitable for demanding control applications in agricultural engineering and related fields.

Button_IX

• – ◦ • – ◦ • – ◦ •

Introduction

Button_IX is a service interface function block for Boolean input data, specifically designed for processing key presses in 4diac systems. This block serves as an interface between the application logic and physical input devices such as pushbuttons or switches.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Request
- **IND**: Indication from Resource Interface

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **u16ObjId** (UINT): Object ID with Initial Value ID_NULL

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status Information
- **IN** (BOOL): Input Data from the Resource

Adapter

None Adapter interfaces are available.

Functionality

The Button_IX function block manages the entire lifecycle of a keystroke. During initialization (INIT event), the service parameters are configured and the system is prepared for operation. The REQ event triggers a current query of the input state, while the IND event signals asynchronous state changes of the keystroke.

Technical Features

- Supports ISOBUS-compliant communication
- Uses specific TypeHash identification for 4diac
- Integrates object ID management for device identification
- Provides comprehensive status feedback via the STATUS output

State Overview

The function block goes through the following main states:

1. **Not Initialized**: Waiting for an INIT event
2. **Initialized**: Ready for REQ and IND events
3. **Active**: Processing input requests and indications
4. **Error**: In case of invalid parameters or communication problems

Application Scenarios

- Operator panels in industrial control systems
- Machine control with push-button inputs
- ISOBUS-compatible agricultural machinery control systems
- Safety-critical input systems with status monitoring

⚖️ Comparison with Similar Blocks

Compared to simple digital input blocks, Button_IX offers extended functionalities such as:

- Configurable service configuration
- Detailed Status Feedback
- ISOBUS Integration
- Asynchronous Event Handling

🛠️ Related Exercises

- Exercise_010a2
- Exercise_010a3

Conclusion

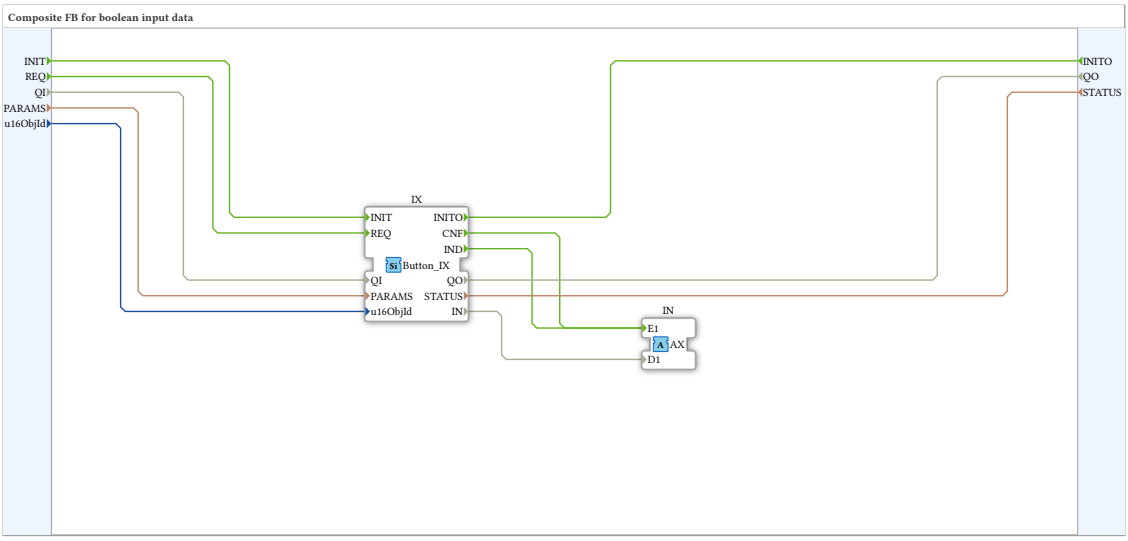
The Button_IX function block provides a robust and flexible solution for integrating keystrokes into 4diac-based control systems. Its comprehensive interface and status management make it particularly suitable for demanding industrial applications that require reliable and monitorable input processing.

–

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

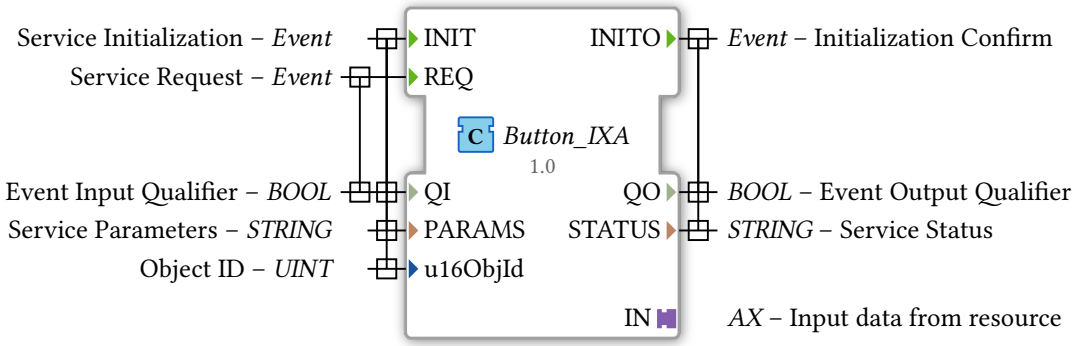
Button_IXA



• – ○ • – ○ • – ○ •

Introduction

Button_IXA is a composite function block for processing Boolean input data. The block serves as an interface for button inputs and provides standardized processing of switching states in automation systems.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - triggered by QI, PARAMS, and u16ObjId
- **REQ**: Service Request - triggered by QI

Event Outputs

- **INITO**: Initialization Acknowledgement - triggered by QO and STATUS

Data Inputs

- **QI**: Event Input Qualifier (BOOL)
- **PARAMS**: Service Parameters (STRING)
- **u16ObjId**: Object ID (UINT) - Initial Value: ID_NULL

Data Outputs

- **QO**: Event Output Qualifier (BOOL)
- **STATUS**: Service Status (STRING)

Adapters

- **IN**: Unidirectional Adapter from Type adapter::types::unidirectional::AX

Functionality

The Button_IXA function block acts as a wrapper around the internal Button_IX block. During initialization (INIT), the configuration parameters are passed and the internal block is configured. Service requests (REQ) trigger the query of the button state. The IN adapter serves as the interface for the actual button input.

Technical Features

- Uses ISOBUS-compliant types and adapters
- Supports object identification via u16ObjId
- Flexible parameterization via STRING parameters
- Unidirectional adapter interface for input data

State Overview

The function block has the following main states:

- Not initialized: Waiting for an INIT event
- Initialized: Ready for REQ requests
- Active: Processing button inputs via the adapter

Application Scenarios

- Button inputs in agricultural machinery
- ISOBUS-compatible control systems
- Boolean input processing in distributed automation systems
- Operator panels and control panels

⚖️ Comparison with Similar Function Blocks

Compared to simple Boolean input blocks, Button_IXA offers:

- ISOBUS compatibility
- Extended parameterization options
- Standardized status feedback
- Adapter-based architecture for improved reusability

🛠️ Related Exercises

- Exercise_010a2_AX
- Exercise_010a3_AX

Conclusion

Button_IXA is a specialized function block for the reliable processing of button inputs in ISOBUS environments. Its modular architecture and standardized interfaces make it ideally suited for use in complex agricultural automation systems.

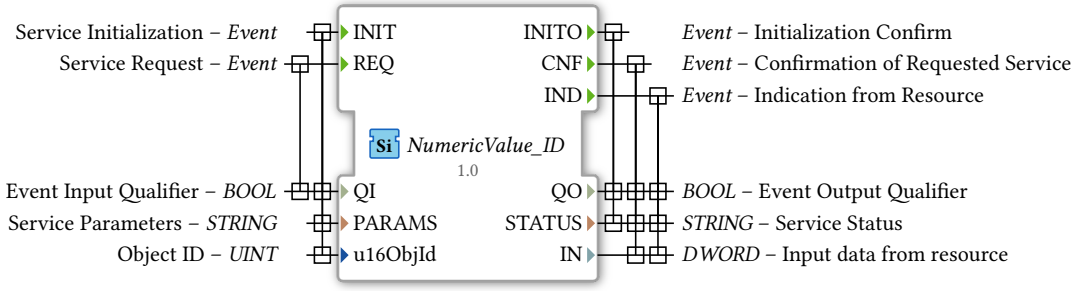
NumericValue

NumericValue_ID

• – ◦ • – ◦ • – ◦ •

Introduction

The NumericValue_ID function block is an input service interface function block for double-word input data (DWORD). It serves as an interface for communication with resources and enables the processing of numeric input values in ISOBUS-compatible systems.



Interface Structure

Event Inputs

- **INIT**: Service Initialization
- Linked to: QI, PARAMS, u16ObjId
- **REQ**: Service Request
- Linked to: QI

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Service Request
- Linked to: QO, STATUS, IN
- **IND**: Resource Indication
- Linked to: QO, STATUS, IN

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **u16ObjId** (UINT): Object ID with Initial Value ID_NULL

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (STRING): Service status
- **IN** (DWORD): Input data from the resource

Adapter

No adapter interfaces are available.

Functionality

The function block initializes itself via the INIT event and can make service requests via REQ. Upon successful initialization, it returns INITO, while CNF and IND represent acknowledgments of service requests and indications from the resource, respectively. Data is transmitted via DWORD values (32-bit), which are provided via the IN output.

Technical Features

- Uses DWORD data type for numeric input values
- ISOBUS-compatible implementation
- Initialization with standardized object ID (ID_NULL)
- String-based status feedback
- Supports configurable service configuration

State Overview

The function block goes through the following states:

1. **Not Initialized**: Before INIT call
2. **Initialized**: After successful INIT processing
3. **Service Active**: During active REQ requests
4. **Data Provisioning**: During active IND indication

Application Scenarios

- Agricultural control systems (ISOBUS)
- Industrial automation with numeric sensor data
- Embedded systems with double-word data processing
- Systems with standardized object identifications

⚖️ Comparison with Similar Blocks

Compared to simple input blocks, NumericValue_ID offers extended service functionality with initialization procedures, Status feedback and configurable options. Its specific focus on ISOBUS environments distinguishes it from generic input blocks.

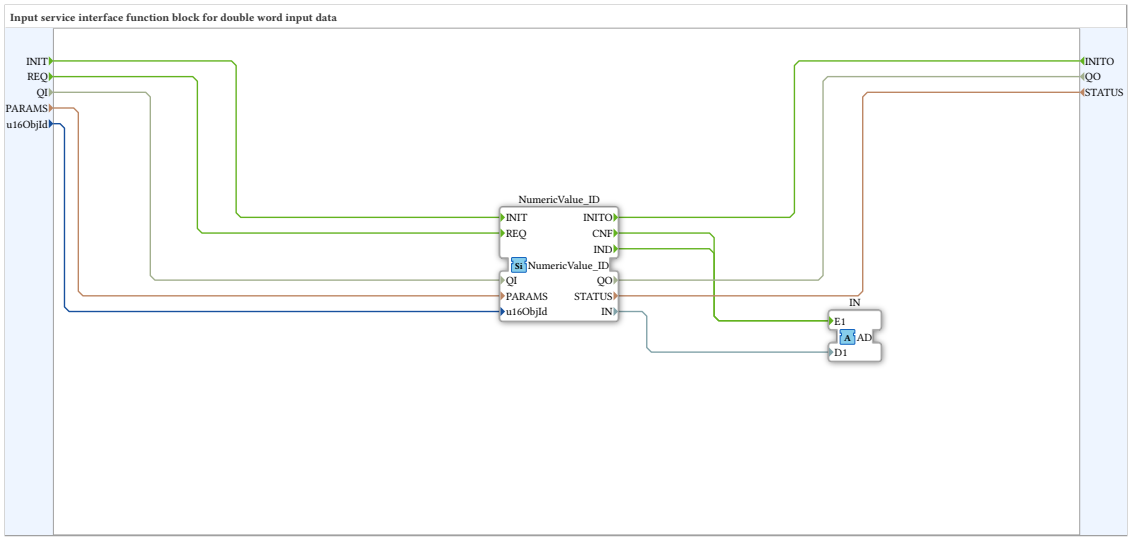
🛠️ Related exercises

- Uebung_011
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_012c
- Uebung_020c2_sub
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Exercise_034a1_Q4
- Exercise_103

Conclusion

The NumericValue_ID function block provides a robust solution for processing double-word input data in ISOBUS-based systems. Its structured initialization, clear status feedback, and flexible parameterization make it ideally suited for industrial applications with high demands on reliability and diagnostic capabilities.

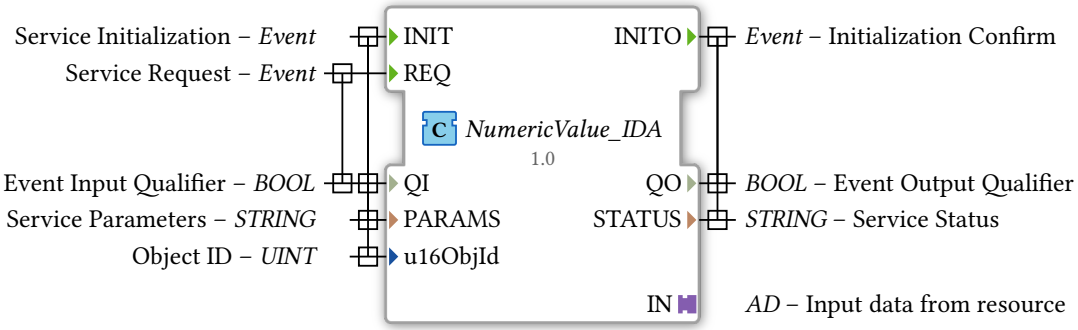
NumericValue_IDA



• – ○ • – ○ • – ○ •

Introduction

The function block **NumericValue_IDA** is a service interface function block specifically designed for processing numeric input data (double word input data) in the context of ISOBUS universal terminals (UTs). It serves to transfer changes to numeric values made on a user interface (e.g., a tractor terminal) into the control logic. Unlike standard input blocks, this block does not provide the received data via direct data outputs, but rather via an adapter connection (`IN`). It encapsulates the functionality of the block `NumericValue_ID` and redirects its signals to an adapter interface.



Interface Structure

The function block has standard initialization and request events as well as specific configuration parameters. The actual data transfer takes place via the adapter.

Event Inputs

- **INIT** (Type: `EInit`): Service initialization. Used to start and configure the function block.
- **REQ** (Type: `Event`): Service request. Can be used to update the status or request a specific interaction (depending on the internal implementation of the service layer).

Event Outputs

- **INITO** (Type: `EInit`): Initialization confirmation. Indicates whether the function block has been successfully configured.

Data Inputs

- **QI** (Type: `BOOL`): Event Input Qualifier. Enables the execution of actions associated with the events.
- **PARAMS** (Type: `STRING`): Service parameter for specific configurations of the underlying service layer.
- **u16ObjId** (Type: `UINT`): Object ID. References the specific numeric input object on the Universal Terminal (UT) whose value is to be monitored. The default value is `ID_NULL`.

Data Outputs

- **QO** (Type: `BOOL`): Event Output Qualifier. Indicates whether the last operation (e.g., initialization) was successful.
- **STATUS** (Type: `STRING`): Service status. Contains error codes or status messages for diagnostic purposes.

Adapter

- **IN** (Type: `adapter::types::unidirectional::AD`):
- This is a unidirectional adapter plug ("Input data from resource").
- It combines the event signal (E1) and the data signal (D1), which contains the current numeric value from the terminal.
- The adapter is triggered when the internal `NumericValue_ID` block sends a `IND` (Indication) or `CNF` (Confirmation) event.

Functionality

The **NumericValue_IDA** acts as a wrapper for the internal block `NumericValue_ID`.

1. **Initialization:** The connection to the specific object on the ISOBUS terminal is established via the event `INIT` and the data inputs `PARAMS` and `u16ObjId`.
2. **Data Reception:** When the value of the referenced object on the terminal changes or is updated by the system, the internal block receives this information.
3. **Forwarding:** The internal block forwards the new value (`IN`) and the associated event to the adapter `IN` of the **NumericValue_IDA**.

- The adapter's event `E1` is triggered.
- The adapter's data point `D1` provides the numerical value.

This mechanism enables clean encapsulation of communication, where the application logic only needs to connect to the adapter `IN` to react to value changes.

Technical Features

- **Composition:** The function block is implemented as a composite function block (FB) that uses instances from the package `isobus::UT::io::NumericValue`.
- **Adapter Usage:** The special feature lies in the use of the adapter `AD` for data output. This reduces the visual complexity in the function block diagram, as data and event flow are bundled in a single connection line.
- **ISOBUS Context:** This function block is explicitly designed for use in the ISOBUS stack (ISO 11783), as indicated by the use of `u16ObjId` to identify UT objects.

Status Overview

Since this is a composite function block, its status is primarily determined by its internal service blocks.

- **Not Initialized:** After startup or during `QI=FALSE` and `INIT`.
- **Initialized / Ready:** After successful `INIT` (`QO=TRUE`). The function block listens for updates from the terminal.
- – **Error:** If `QO=FALSE` and `STATUS` contain an error message (e.g., invalid object ID).

Application Scenarios

- **Setpoint Input:** An operator enters a quantity (e.g., application rate in liters) or a target speed on the terminal. This function block receives the value.
- **Parameterization:** Setting machine parameters via numeric fields on the display.
- **Modular Applications:** Used in applications that rely heavily on adapter connections to avoid the "spaghetti code" of parallel event and data lines.

⚖️ Comparison with Similar Function Blocks

- **NumericValue_ID:** This is the internal base function block. It offers the same functionality but provides the data via classic `IND`/`CNF` events and a `IN` data output. **NumericValue_IDA** is the adapter version of this function block.
- **Other `_IDA` function blocks:** Similar to `Button_IDA` (for buttons) or `String_IDA` (for text), this function block follows the design pattern of providing terminal input via adapters.
-

🛠️ Related Exercises

- Exercise_011_AUDI

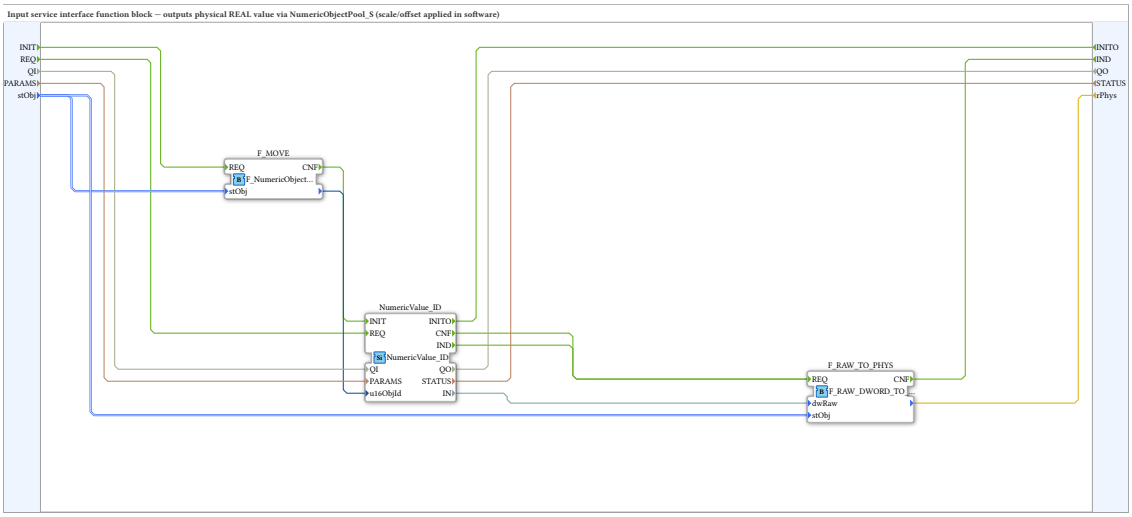
Conclusion

The **NumericValue_IDA** is a specialized interface block for ISOBUS applications in 4diac. By using an adapter for data output, it enables compact and clear routing of numerical input values in the function block diagram. It is ideally suited for developers who prefer a structured and modular application architecture and need to process numerical input from the Universal Terminal.

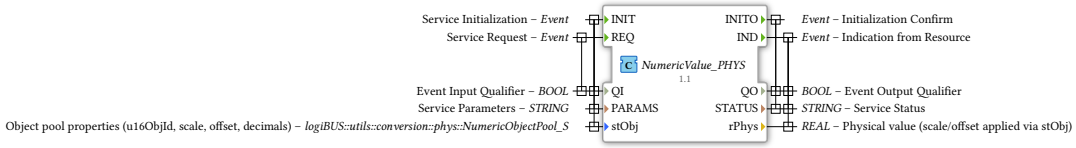
🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

NumericValue_PHYS



(Image not available)



• – ○ • – ○ • – ○ •

Introduction

The function block **NumericValue_PHYS** is an input service interface block according to ISO 11783-6. It returns a physical REAL value by reading a raw digital value (DWORD) from the ISOBUS object pool and converting it into a physical value, taking into account a predefined scaling and offset. The conversion is performed entirely in software (within the function block).

Interface Structure

Event Inputs

Event	Type	With Variables	Comment
INIT	EInit	QI, PARAMS, stObj	Initialize the block
REQ	Event	QI	Request a new physical value

Event Outputs

Event	Type	With variables	Comment

INITO	EInit	QO, STATUS	Confirm initialization
IND	Event	QO, STATUS, rPhys	Output the calculated physical value

Data Inputs

Variable	Type	Comment
QI	BOOL	Input qualifier (enables processing)
PARAMS	STRING	Service parameter (e.g., configuration string)
stObj	logiBUS::utils::conversion::phys::NumericObjectPool_S	Object pool properties: Object ID (16 bits), scaling, offset, decimal places

Data Outputs

Variable	Type	Comment
QO	BOOL	Output qualifier (processing status)
STATUS	STRING	Status message (error or success message)
rPhys	REAL	Physical value after applying scaling/offset

Adapters

No adapters available.

Functionality

The internal process is controlled via the INIT and REQ events and utilizes four sub-function blocks:

1. **INIT**

- The passed structure parameter `stObj` is copied via the sub-function block `F_MOVE` (of type `iec61131::selection::F_MOVE`).
- The copied value (`stObj.u16ObjId`) is forwarded to the sub-function block `NumericValue_ID`, which is thereby initialized (`NumericValue_ID.INIT`).

1. **REQ** (or re-output after INIT)

- The sub-function block `NumericValue_ID` is triggered by `REQ`. It outputs a raw DWORD value from the ISOBUS object pool via its output `IN`.

This DWORD value is converted into an unsigned 32-bit integer (`UDINT`) via `F_DWORD_TO_UDINT`.

The sub-module `F_RAW_TO_PHYS` receives the integer value and the original structure `stObj` (scaling, offset, decimal places) and calculates the physical REAL value from it (`rPhys`).

Finally, the output `IND` is activated, and the calculated value is output at the data output `rPhys`. ...``

The chaining ensures that the current raw value is read from the object pool and calculated for each REQ.

Technical Features

- **Standard Compliance**: The function block complies with the ISO 11783-6 standard (ISOBUS).
- **Software Scaling**: Unlike purely hardware interfaces, the conversion (scaling/offset) is performed within the function block, allowing for flexible adaptation without changing the peripherals.
- **Reusable Sub-Function Blocks**: The sub-function blocks used (`NumericValue_ID`, `F_DWORD_TO_UDINT`, `F_RAW_TO_PHYS`, `F_MOVE`) are standardized logiBUS or IEC 61131 function blocks and can be used in other contexts.
- **Object Pool Structure**: The input structure `stObj` contains all necessary parameters (object ID, scale, offset, decimal places) and can be managed centrally.

State Overview

The function block does not have an explicit state machine but reacts purely event-driven. The process is divided into two main phases:

- **Initialization Phase** (Event `INIT` ⟶ `INITO`):

Acceptance and storage of the configuration (`stObj`), initialization of `NumericValue_ID`.

- **Operation Phase** (Event `REQ` ⟶ `IND`):

Reading the raw value, conversion, and output of the physical value.
After successful initialization, `REQ` can be executed any number of times.

Application Scenarios

- **ISOBUS Vehicle Control**: Reading sensor data (e.g., speed, pressure, temperature) from the ISOBUS object pool and converting it into physical units.
- **Agricultural Automation**: Processing measured values from ISOBUS-compatible devices (tractors, implements) for further control or visualization.
- **Test and Simulation Environments**: Replacing real hardware components with software emulation using defined scaling.

Comparison with Similar Components

- **NumericValue_ID**: Returns only the raw DWORD value without scaling/offset. `NumericValue_PHYS` extends this value with the physical conversion.

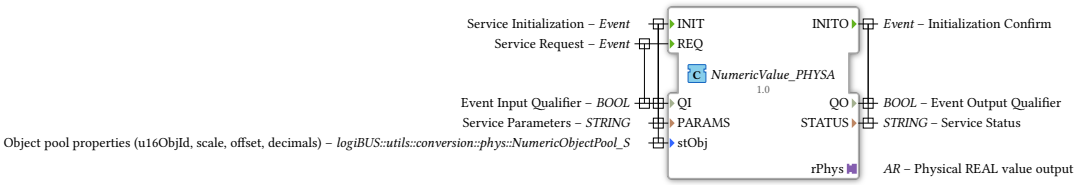
NumericValue_ID: Returns only the raw DWORD value without scaling/offset. - **Analog input blocks** (e.g., `AI_SCALED`) in PLC systems: These often scale directly in the hardware driver. `NumericValue_PHYS`, on the other hand, operates purely in software and is therefore more flexible with regard to parameterization.

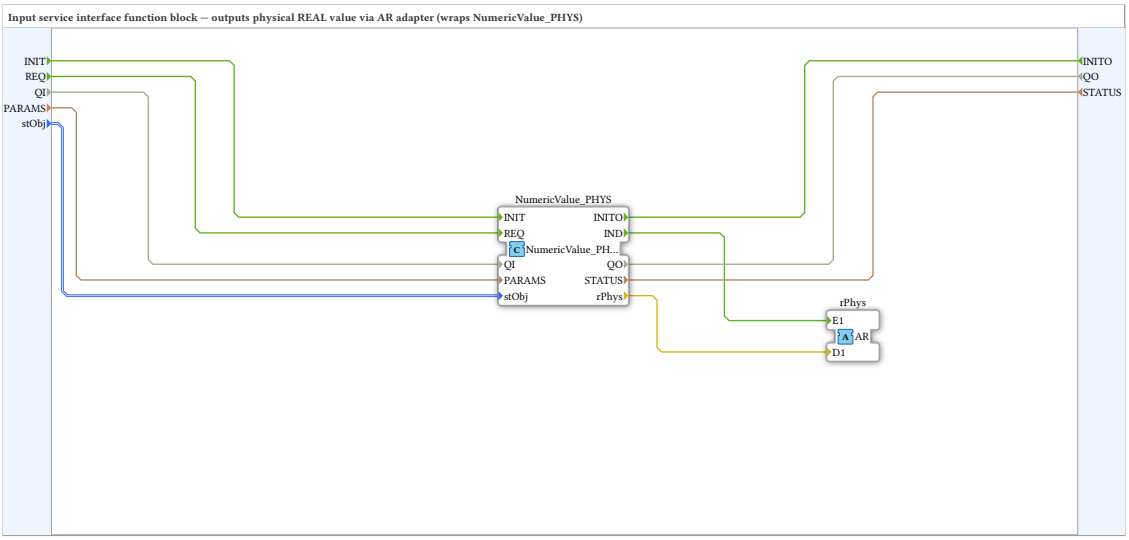
- **logiBUS blocks such as `F_RAW_TO_PHYS`**: These are used internally by `NumericValue_PHYS`; this function block encapsulates the entire chain from ID query to final output.

Conclusion

The function block `NumericValue_PHYS` offers a clean and standards-compliant way to convert ISOBUS objects into physical values with integrated scaling and offset correction. By using modular sub-components and parameterization via a structure variable, it is easy to maintain and reusable. It is particularly suitable for applications requiring software-controlled conversions without affecting the hardware configuration.

NumericValue_PHYSA





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **NumericValue_PHYSA** is an input service interface function block (SIFB) according to ISO 11783-6. Its function is to provide a physical REAL value via an AR adapter (adapter type *unidirectional::AR*). Internally, it encapsulates the block `NumericValue_PHYS` and all its logic for converting and providing the value.

Interface Structure

Event Inputs

Name	Type	Description
INIT	Initializes the block (with parameters and object pool)	
REQ	Event	Triggers the output of the current physical value

Event Outputs

Name	Type	Description
INITO	EInit	Confirmation of successful initialization

Data Inputs

Name	Type	Description
QI	BOOL	Input qualifier (enables/controls processing)
PARAMS	STRING	Parameters for the service (e.g., configuration string)
stObj	*NumericObjectPool_S*	Object pool properties: Object ID (UINT16), scale, offset, decimal places

Data Outputs

Name	Type	Description
QO	BOOL	Output qualifier (indicates successful processing)
STATUS	STRING	Status message (OK or error text)

Adapter

Name	Type	Description
rPhys	*adapter::types::unidirectional::AR*	Adapter interface for outputting the physical REAL value (event channel `E1` and data channel `D1`)

Functionality

This function block is implemented as a wrapper around the internal function block `NumericValue_PHYS`. All inputs and outputs are directly connected to the inner function block:

- Events `INIT` and `REQ` are forwarded directly to `NumericValue_PHYS`.
- The initialization confirmation event `INITO` of the inner function block is routed externally.
- The output event `IND` of `NumericValue_PHYS` is connected to the event input `rPhys.E1` of the adapter, so that the adapter receives an event with each new value calculation.
- Events `INIT` and `REQ` are forwarded directly to `NumericValue_PHYS`.
- The initialization confirmation event `INITO` of the inner function block is routed externally. - The data inputs (`QI`, `PARAMS`, `stObj`) are wired to the inner FB.

- The data outputs (`QO`, `STATUS`) come directly from the inner FB.
- The physical REAL value from `NumericValue_PHYS.rPhys` is placed on the data channel `rPhys.D1` of the adapter plug.

Thus, the FB is a pure connector that extends the functionality of `NumericValue_PHYS` with a standardized adapter interface, without implementing any additional logic itself.

Technical Features

- **Data Types Used:** The physical properties of the measured value (scaling, offset, decimal places) are defined via the structured type `logiBUS::utils::conversion::phys::NumericObjectPool_S`.
- **Adapter Coupling:** The AR adapter (unidirectional) enables loose coupling to subsequent function blocks that consume the physical value.
- **Type Hash:** The function block carries an attribute `eclipse4diac::core::TypeHash` to identify the exact version.
- **No Internal State Machine:** The function block delegates all state logic to the encapsulated function block.

State Overview

Since the FB (Function Block) does not have its own state machine, its state is entirely determined by its internal state:

1. **Initialization Phase:** After the event `INIT`, the FB is configured. It is only operational after `INITO`.
2. **Operation Phase:** With each `REQ` event, the current physical value is calculated and output via the adapter.
3. **Error State:** In case of faulty initialization or invalid parameters, `STATUS` is populated with a corresponding text message, and `QO` is set to FALSE.

Application Scenarios

- **ISOBUS Control Units:** This function block is ideally suited for converting sensor data (e.g., speed, pressure, temperature) into a standardized adapter interface, as required in agricultural machinery according to ISO 11783-6.
- **Measurement Acquisition with Scaling:** Raw values can be converted into physical units using the parameter structure with individual offsets and scaling factors.
- **Modular Systems:** The adapter interface allows multiple output blocks to be easily connected to various consumers (e.g., visualization, logic).

Comparison with Similar Function Blocks

- **NumericValue_PHYS** (without the "A" suffix): The pure function block without an adapter interface. It provides the physical value only as a data output (`rPhys`). This function block adds the adapter plug to enable coupling via a standardized channel.
- **NumericValue_RAW / NumericValue_SCALED** (hypothetical): These blocks would deliver raw or scaled values without reference to an object pool. In contrast, `NumericValue_PHYSA` uses the structure `NumericObjectPool_S` with explicit conversion parameters.

Conclusion

The function block `NumericValue_PHYSA` offers a clean, adapter-based interface for outputting physical REAL values. By encapsulating all the conversion logic in `NumericValue_PHYS`, it remains lean and maintainable. Its use is recommended wherever ISO 11783-6 compliant physical measurement values need to be provided in a modular automation system.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Softkey

🎧 Podcast

- ISO 11783-6: Understanding Softkeys and the Virtual Terminal – Your Key to Agricultural Machinery Mechatronics ...* [ISO 11783-6: Understanding Softkeys and the Virtual Terminal – Your Key to Agricultural Machinery Mechatronics]
- [ISO

SoftKeyActivationCode



• – ◦ • – ◦ • – ◦ •

Introduction

SoftKeyActivationCode is a collection of global constants for processing softkey events in ISOBUS-compatible systems. These constants define various activation states of softkeys on virtual terminals and are used to control user interactions in agricultural machinery.

Interface Structure

SoftKeyActivationCode consists of global constants and not a function block. Therefore, there are no event or data interfaces in the conventional sense.

Constant Definitions

- **SK_RELEASED**: Softkey released (state change) - Code: 0
- **SK_PRESSED**: Softkey pressed (state change) - Code: 1
- **SK_STILL_HELD**: Softkey still held - Code: 2
- **SK_STILL_HELD_START**: Softkey still held (start) - Code: 2, bCyclic: FALSE
- **SK_ABORTED**: Softkey press aborted (applies to VT version 4 and higher) - Code: 3
- **Invalid**: Invalid state - Code: 255

Functionality

The constants represent different activation states of softkeys in ISOBUS systems. They are used to indicate the status of user interactions with virtual terminals. The transmission repetition rate is 200 ms when a soft key is pressed/released and every 200 ms when a soft key is held.

Technical Features

- All constants are of type `isobus::UT::io::Softkey::SoftKeyActivationCode_S`
- The constants are defined as global constants and can be used system-wide
- The abort functionality (SK_ABORTED) is only available in Virtual Terminal version 4 and higher
- The constant SK_STILL_HELD_START additionally contains a Boolean parameter `bCyclic`

Application Scenarios

- Control of user interfaces in agricultural machinery
- Processing of touch inputs on ISOBUS-compatible displays
- Implementation of softkey interactions in Virtual Terminal applications
- Status monitoring of user inputs in real-time control systems

⚖️ Comparison with Similar Blocks

Unlike function blocks, global constants do not offer active functionality but merely serve to standardize state values. They are comparable to ENUM definitions in other programming languages.

Conclusion

SoftKeyActivationCode provides a standardized collection of constants for softkey processing in ISOBUS environments. Using these constants enables a uniform and maintainable implementation of user interactions in agricultural control systems.

SoftKeyActivationCode_S

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `SoftKeyActivationCode_S` is a structured data structure (struct) used to configure the activation of a softkey in an ISOBUS-compliant system. It defines the specific activation code and a control flag for the key press repetition frequency.

Interface Structure

This is a data type and therefore does not have event or adapter interfaces in the conventional sense of a function block. The structure consists of the following data fields:

Data Inputs

(Not applicable – this is a data type)

Data Outputs

(Not applicable – this is a data type)

Data Fields (Structure Components)

Name	Data Type	Initial Value	Comment
`code`	`USINT`	255	Defines the numeric code that identifies the specific softkey.
`bCyclic`	`BOOL`	TRUE	Controls the cyclic sending of the key code. With `FALSE`, the key is not repeatedly sent every 200ms.

Functionality

The structure `SoftKeyActivationCode_S` serves as a container for configuration data. The value in the field `code` corresponds to a specific key on a virtual or physical terminal. The field `bCyclic` allows you to control the behavior when a key is held down. If set to `TRUE` (default), the key press is repeatedly sent at a defined interval (e.g., every 200 ms) as long as the key remains pressed. With `FALSE`, only a single event is triggered upon the initial key press.

Technical Features

- **Initial Values:** Both fields have predefined initial values. The default value `255` for `code` can be interpreted as "invalid" or "unconfigured" code.
- **ISOBUS Context:** This data type is part of the `isobus::UT::io::Softkey` package, indicating its use in ISOBUS Virtual Terminal (VT) or similar I/O contexts.
- **Cycle Control:** Precise control of the repetition rate via `bCyclic` is crucial for user-friendly and reliable keyboard input, especially with long data fields or list controls.

State Overview

(Not applicable – this is a passive data type without a state machine.)

Application Scenarios

This data structure is typically used as an input or configuration parameter for function blocks that:

1. Emulate or control the virtual keyboard input of an ISOBUS terminal.
2. Process softkey events in application logic (e.g., menu navigation, value input).
3. Dynamically configure the key assignment or key behavior of a device.

⚖️ Comparison with Similar Blocks

Unlike a function block (FB), `SoftKeyActivationCode_S` is a pure **data type** (struct). It does not contain any algorithms or event-driven logic.

- **Similarity to `E_SR` or `E_RS`:** These are function blocks with memory functionality and internal state logic. `SoftKeyActivationCode_S`, on the other hand, is a pure data carrier.
- **Comparison with simple data types:** Unlike atomic types such as `INT` or `BOOL`, this struct combines two logically related pieces of information (`code` and behavior `bCyclic`) into a single, reusable unit.

Conclusion

The data type `SoftKeyActivationCode_S` is an essential auxiliary structure for controlling softkey input in ISOBUS environments. By encapsulating the key code and repeat behavior in a single struct, configuration is simplified and code reusability is increased. Its clear structure and sensible default values make it a robust building block for developing terminal I/O logic.

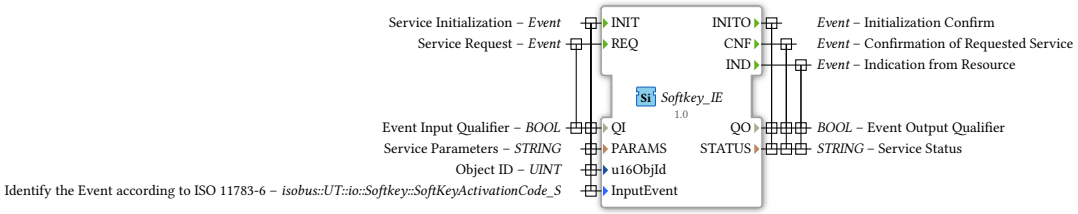
Softkey_IE

🎧 Podcast

- ISO 11783-6: Understanding Softkeys and the Virtual Terminal – Your Key to Agricultural Machinery Mechatronics

Introduction

The Softkey_IE function block is an input service interface function block for event input data, specifically designed for processing softkey events according to ISO 11783-6. It serves as an interface between the application logic and the physical softkey input devices in agricultural and mobile machinery.



Interface Structure

Event Inputs

- **INIT**: Service initialization with parameters QI, PARAMS, u16ObjId, and InputEvent
- **REQ**: Service request with parameter QI

Event Outputs

- **INITO**: Initialization acknowledgment with parameters QO and STATUS
- **CNF**: Acknowledgement of the requested service request with parameters QO and STATUS
- **IND**: Indication from the resource with parameters QO and STATUS

Data Inputs

- **QI** (BOOL): Event input qualifier
- **PARAMS** (STRING): Service parameter
- **u16ObjId** (UINT): Object ID with initial value ID_NULL
- **InputEvent** (SoftKeyActivationCode_S): Identifies the event according to ISO 11783-6 with initial value “Invalid”

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (STRING): Service status

Adapters

No adapter interfaces available.

Functionality

The Softkey_IE function block manages communication with softkey input devices according to the ISO bus standard 11783-6. During initialization (INIT), the service parameters and object ID are configured. Service requests (REQ) trigger the corresponding functionality, while indications (IND) signal incoming events from the physical softkeys.

Technical Features

- Supports ISO 11783-6 standard for agricultural vehicles
- Uses a specific SoftKeyActivationCode structure for event identification

- Integrates object ID management for device identification
- Provides comprehensive status feedback via STRING parameters

State Overview

The function block has an initialized state and an operating state. After successful INIT initialization, the block switches to the operating state, in which REQ requests can be processed and IND events can be received.

Application Scenarios

- Control of operator panels in agricultural machinery
- Implementation of softkey functionalities in mobile work equipment
- ISOBUS-compliant input processing in vehicles
- User interfaces for complex machine controls

⚖️ Comparison with Similar Blocks

Compared to generic input blocks, Softkey_IE offers specific ISO 11783-6 compliance and is optimized for the requirements of agricultural and mobile work equipment. The integration of SoftKeyActivationCode enables standardized event handling.

🛠️ Related exercises

- Uebung_010b2
- Uebung_010b2_AX
- Uebung_010b6
- Uebung_010b6_AX
- Uebung_013
- Uebung_013_AX
- Uebung_014
- Uebung_015
- Uebung_015a
- Uebung_016
- Uebung_016a
- Uebung_017
- Uebung_018
- Uebung_018a
- Uebung_019a
- Uebung_019b
- Uebung_021
- Uebung_022
- Exercise_023
- Exercise_024
- Exercise_025
- Exercise_026
- Exercise_039
- Exercise_039a
- Exercise_039a_sub_Outputs

Conclusion

The Softkey_IE function block represents a specialized solution for softkey event processing in ISO 11783-6 compliant systems. Its standardized interface and comprehensive status feedback make it ideally suited for use in complex mobile machine control systems.

Softkey_IX

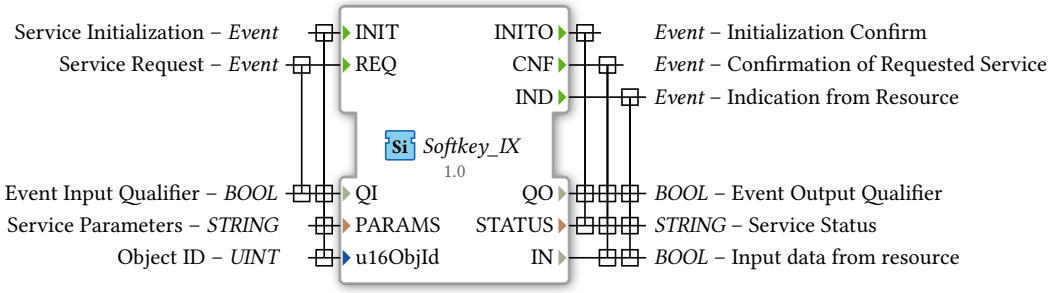
🎧 Podcast

- ISO 11783-6: Understanding Softkeys and the Virtual Terminal – Your Key to Agricultural Machinery Mechatronics

Introduction

The Softkey_IX is a service interface function block for Boolean input data, serving as an input interface for softkey functionalities. This block enables communication with resources and the processing of keyboard input in control systems.

•



Interface Structure

Event Inputs

- **INIT**: Service Initialization
- Linked to: QI, PARAMS, u16ObjId
- **REQ**: Service Request
- Linked to: QI

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Service Function
- Linked to: QO, STATUS, IN
- **IND**: Indication from Resource
- Linked to: QO, STATUS, IN

Data Inputs

- **QI**: BOOL - Event Input Qualifier
- **PARAMS**: STRING - Service Parameter
- **u16ObjId**: UINT - Object ID (Initial Value: ID_NULL)

Data Outputs

- **QO**: BOOL - Event output qualifier
- **STATUS**: STRING - Service status
- **IN**: BOOL - Input data from the resource

Adapters

No adapter interfaces are available.

Functionality

The Softkey_IX function block acts as an intermediary between the application logic and physical or virtual input devices. During initialization (INIT), the service parameters are configured. Service requests can be made via REQ events, which are acknowledged by CNF events. IND events signal asynchronous input from the resource.

Technical Features

- Uses ISOBUS-compatible object identification
- Supports configurable service configuration via STRING parameters
- Offers both synchronous (CNF) and asynchronous (IND) operating modes
- Initialization with a standardized object ID (ID_NULL)

State Overview

The function block goes through the following main states:

1. **Not Initialized**: Before INIT processing
2. **Initialized**: After successful INIT processing
3. **Ready**: Can process service requests
4. **Active**: Processes current inputs

Application Scenarios

- Operator panels in mobile machinery
- Virtual keyboards in industrial plants
- Softkey implementations in vehicle systems
- User interfaces for ISOBUS-compatible devices

⚖️ Comparison with Similar Blocks

Compared to simple digital input blocks, Softkey_IX offers extended service functionalities with configurable settings and ISOBUS compatibility. The IND functionality enables asynchronous event handling, which is not available with purely query-based function blocks.

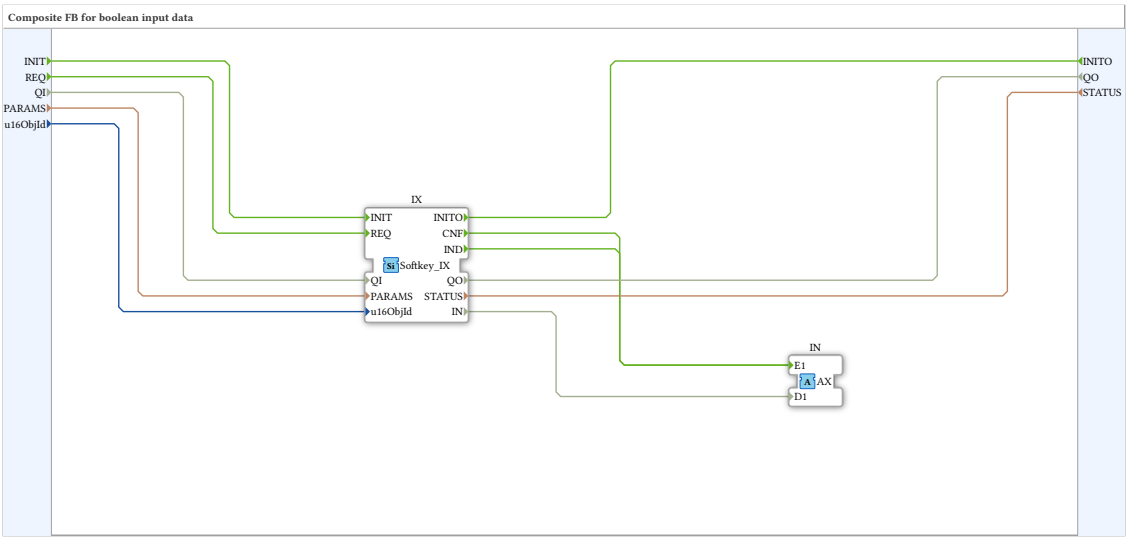
🛠️ Related exercises

- Uebung_010
- Uebung_010a
- Uebung_010a4
- Uebung_010c
- Uebung_010c2
- Exercise_039_sub_Outputs
- Exercise_039b

Conclusion

The Softkey_IX function block represents a flexible and standardized solution for softkey inputs in industrial control systems. Its ISOBUS compatibility and comprehensive service interface make it particularly suitable for demanding applications in mobile machinery and industrial plants.

Softkey_IXA

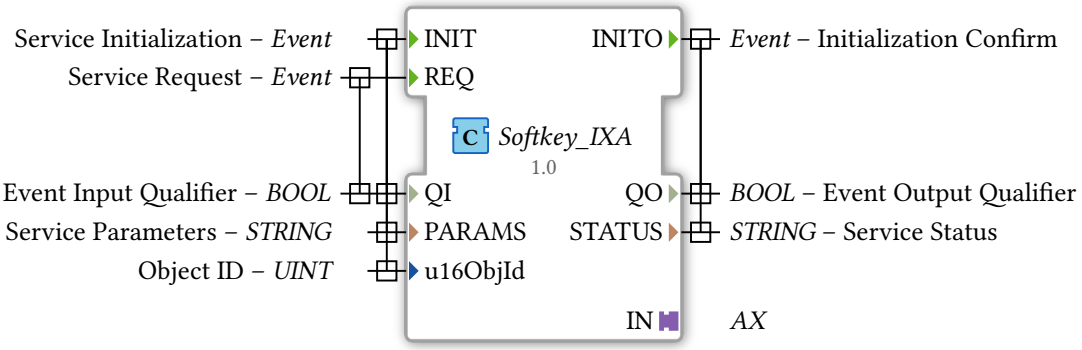


🎧 Podcast

- ISO 11783-6: Understanding Softkeys and the Virtual Terminal – Your Key to Agricultural Machinery Mechatronics

Introduction

Softkey_IXA is a composite function block for processing Boolean input data. The block serves as an interface for softkey operations and enables the initialization and querying of input states via standardized ISOBUS communication protocols.



Interface Structure

Event Inputs

- **INIT**: Service initialization event with the associated data QI, PARAMS, and u16ObjId
- **REQ**: Service request event with the qualifier QI

Event Outputs

- **INITO**: Initialization confirmation with the output data QO and STATUS

Data Inputs

- **QI**: BOOL - Event Input Qualifier
- **PARAMS**: STRING - Service Parameter
- **u16ObjId**: UINT - Object ID with initial value ID_NULL

Data Outputs

- **QO**: BOOL - Event Output Qualifier
- **STATUS**: STRING - Service Status

Adapters

- **IN**: Adapter of type adapter::types::unidirectional::AX for unidirectional communication

Functionality

The composite function block Softkey_IXA encapsulates the internal function block Softkey_IX and acts as an intermediary between external calls and the actual softkey functionality. During initialization (INIT), parameters are passed to the internal block, and during service requests (REQ), the corresponding queries are executed. The IN adapter enables communication with external components.

Technical Features

- Uses ISOBUS-specific types and constants
- Supports object identification via u16ObjId
- Flexible parameter passing via STRING parameters
- Unidirectional adapter interface for extended communication

State Overview

The function block has two main states:

1. **Initialization State**: Processing of INIT events with parameter passing
2. **Service State**: Processing of REQ requests and return of results

Application Scenarios

- Agricultural operator terminals with softkey functionality
- ISOBUS-compatible control systems
- Applications with Boolean input queries
- Systems with configurable service interfaces

⚖️ Comparison with Similar Blocks

Compared to simple Boolean input blocks, Softkey_IXA offers extended ISOBUS compatibility, configurable service interfaces, and adapter-based extensibility. The composite structure allows for a clear separation between interface logic and core functionality.

🛠️ Related exercises

- Uebung_010_AX
- Uebung_010a4_AX
- Uebung_010a_AX
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- Exercise_010c_AX

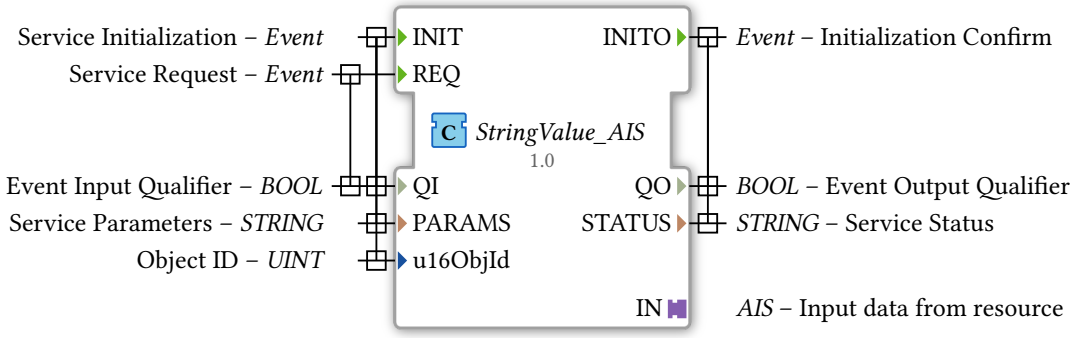
Conclusion

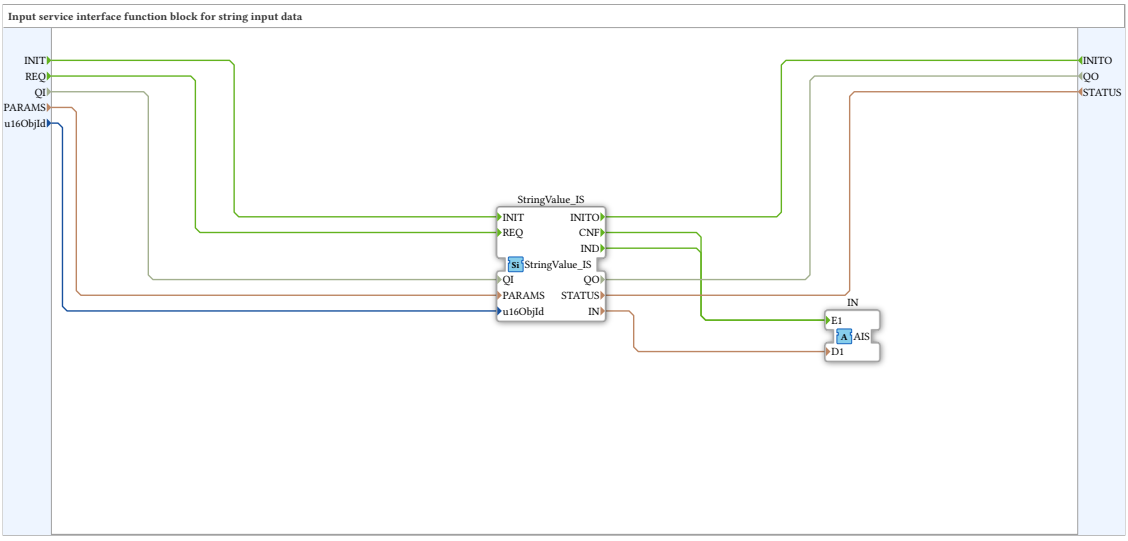
Softkey_IXA represents a robust and extensible composite function block for softkey operations in ISOBUS environments. Its clear interface separation and adapter support make it ideally suited for complex control systems in agricultural technology.

Numerical Index (io)

Here you will find all exercises from the `io` workspace in ascending order. ...

StringValue_AIS





• – ○ • – ○ • – ○ •

Introduction

The **StringValue_AIS** function block is a service interface function block for providing string input data. It serves as an interface between an external resource (e.g., ISOBUS Universal Terminal) and the control logic. The block initializes communication, processes requests, and passes status information to the application.

Interface Structure

Event Inputs

Event	Type	With Variables	Description
INIT	EInit	QI, PARAMS, u16ObjId	Initializes the service and passes qualifiers, parameters, and object IDs.
REQ	Event	QI	Requests a data query or processing.

Event Outputs

Event	Type	With Variables	Description
INITO	EInit	QO, STATUS	Confirms initialization and provides status information.

Data Inputs

Variable	Type	Initial Value	Description
QI	BOOL	–	Qualifier for events (TRUE = active).
PARAMS	STRING	–	Service parameters (e.g., configuration data).
u16ObjId	UINT	ID_NULL	Object ID referenced in the log.

Data Outputs

Variable	Type	Description
QO	BOOL	Output qualifier (service acknowledgment).
STATUS	STRING	Service status message (e.g., error message or success message).

Adapters

Adapter	Type	Direction	Description
IN	adapter::types::unidirectional::AIS	Input	Unidirectional adapter for data exchange with an external resource (e.g., a hardware interface).

Functionality

The **StringValue_AIS** block encapsulates an internal service block **StringValue_IS** and controls its event and data flows. Upon an INIT event, the input parameters (QI, PARAMS, u16ObjId) are forwarded to the internal block, which initializes the communication. The INITO event output signals success (QO, STATUS).

A REQ event triggers a further query—the internal block sends the request (e.g., reading a string value) to the external resource via the **IN** adapter. The response is received asynchronously via the adapter (IND or CNF event) and returned as a STATUS/QO output.

The connections are hard-coded in the FB network:

- **INIT** and **REQ** are routed to the internal function block **StringValue_IS**.
- The response events **IND** and **CNF** of the internal function block are routed to the adapter output **IN.E1**.
- The data lines **QI**, **PARAMS**, and **u16ObjId** feed the internal function block, while **IN.D1** (from the adapter) and **STATUS**/**QO** are passed to the outputs of the function block.

Technical Features

- The function block is designed as a **Service Interface Function Block (SIFB)** and is suitable for integration into an ISOBUS-compliant control environment.
- The interface uses a **unidirectional adapter (AIS)**, which transmits data exclusively from the resource to the controller.
- The initial value of `u16ObjId` is set to `ID_NULL` – a placeholder that must be replaced with a valid object ID during the initialization step.
- The implementation is licensed under the **Eclipse Public License 2.0**.
- The component uses an internal type hash mechanism for integrity checking.

State Overview

An explicit state machine is not represented in the XML. The component operates in an event-driven manner:

- **IDLE**: Waiting for INIT.
- **INIT completed**: Initialization complete, waiting for REQ.
- **IN processing**: After REQ, the request is sent to the adapter; the response is asynchronous.

The response via INITO signals the completion of initialization, while the STATUS output reflects the current state of the service.

Application Scenarios

- **ISOBUS UT Connection**: Reading a text field (e.g., machine name, field label) from a Universal Terminal.
- **Configuration Interface**: Accepting string parameters from a higher-level control system.
- **Data Logging**: Capturing external strings (e.g., barcode scanner, serial numbers) for further processing.

Comparison with Similar Modules

- **IntegerValue_AIS**: Similar structure, but for integer data. It differs only in the data type of the output value.
- **CommandValue_AIS**: Offers bidirectional communication (send/receive) and expects a command format.
- **StringValue_AIS** is specifically optimized for receiving strings and requires no response other than status.

Conclusion

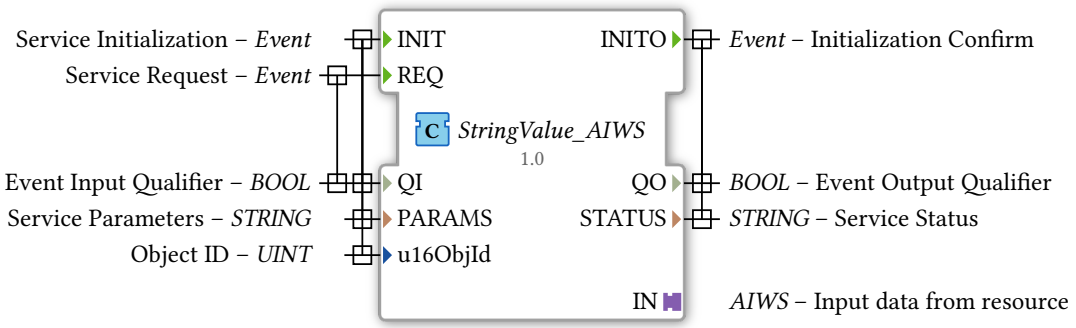
The **StringValue_AIS** is a specialized service interface module for receiving string data via a unidirectional adapter. It simplifies the integration of external text sources into automation systems and provides a clean separation between communication services and application logic. Thanks to its standardized interface (event/data connections), it can be flexibly integrated into 4diac IDE projects.

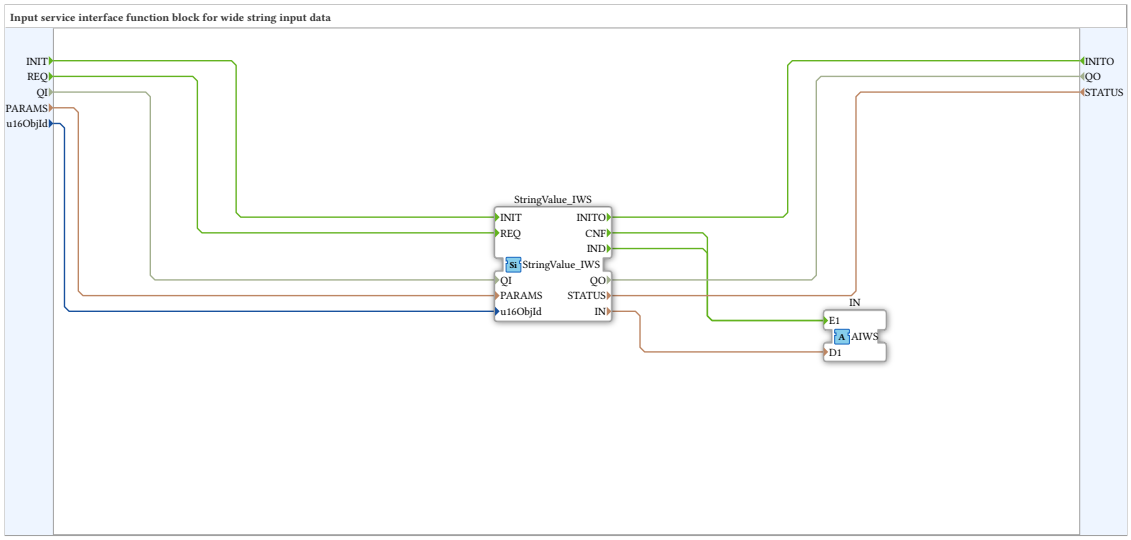
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

StringValue_AIWS





• – ○ • – ○ • – ○ •

Introduction

The function block **StringValue_AIWS** is an input-service interface block for processing wide-string data (UTF-16) in an ISOBUS context. It provides an adapter-based interface through which external resources can input strings. The block acts as a wrapper for the internal block *StringValue_IWS* and offers a standardized initialization and request interface.

Interface Structure

Event Inputs

Event	Type	With Variables	Comment
INIT	EInit	QI, PARAMS, u16ObjId	Service Initialization
REQ	Event	QI	Service Request (Query)

Event Outputs

Event	Type	With Variables	Comment
INITO	Initialization	QO, STATUS	Initialization Acknowledgement

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	–	Event Qualifier (Share)
PARAMS	STRING	–	Service Parameter (e.g., Configuration String)
u16ObjId	UINT	ID_NULL	Object ID (e.g., ISOBUS Object Identifier)

Data Outputs

Name	Type	Comment
QO	BOOL	Output Qualifier (Status of Last Operation)
STATUS	STRING	Status message (e.g., error message or success message)

Adapter

Name	Type	Comment
IN	adapter::types::unidirectional::AIWS	Input of wide-string data from the resource (unidirectional)

The adapter IN receives the actual strings and makes them available to the function block. The connection is established via the event line E1 and the data channel D1.

Functionality

The function block initializes the internal *StringValue_IWS* function block via the event INIT. The parameters QI, PARAMS, and u16ObjId are passed to this event. After successful initialization, the event INITO is triggered, and the output variables QO and STATUS are set.

The event REQ initiates a request to provide a new wide-string value. The actual string is delivered asynchronously via the adapter IN: As soon as the internal module receives a new string, it sends an event (IND or CNF) to the adapter output E1, and the data is then passed to the connected resource via the data channel D1.

The module encapsulates all the logic for initialization and communication with the underlying wide-string service and provides a uniform adapter interface to the outside.

Technical Features

- **Wide-String Support**: The module is designed for UTF-16 (Wide Strings) and is therefore particularly suitable for international character sets (Unicode).
- **Adapter-based communication**: The use of a unidirectional adapter (AIWS) allows for a clean separation between service logic and the resource interface.
- **ISOBUS compliance**: The object ID (u16ObjId) is assigned the initial value ID_NULL, indicating ISOBUS-specific assignment.
- **Event-driven processing**: Both initialization and data requests are triggered by explicit events, making the function block suitable for use in time-critical environments.

State overview

The function block does not have an explicit state machine; its behavior is controlled by events:

1. **Idle state** – The function block waits for INIT or REQ.
2. **Initialization** – After INIT, the parameters are passed to the internal function block; outputs QO and STATUS are updated, and INITO is sent.
3. **Request State** – After REQ, a data request is initiated. The response is asynchronous via the adapter IN (event E1).
4. **Error Handling** – If an error occurs during initialization, QO is set to FALSE, and a corresponding message STATUS is output.

Application Scenarios

- **Input of Unicode text** into agricultural operator terminals (ISOBUS-UT) – e.g., for vehicle names, field names, or product designations.
- **Configuration interface** for ISOBUS devices that require wide-string parameters.
- **Connection of external input devices** (keyboards, touchscreens) to an ISOBUS control unit via a standardized adapter interface.

Comparison with Similar Modules

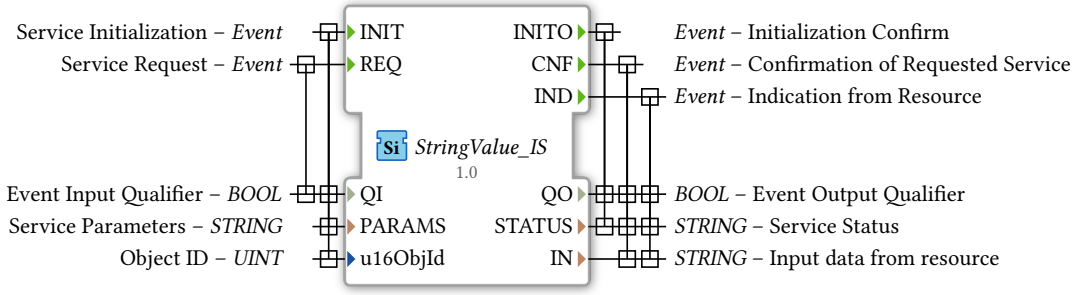
Module	Interface	Special Feature
StringValue_AIWS	Adapter (Input)	Receives wide-strings via the adapter AIWS
StringValue_IWS	Direct Events/Data	Internal logic, no adapter interface
StringValue (Standard)	Events/Data	Simple Wide-String Input (without adapter)

The StringValue_AIWS stands out due to its adapter-based interface, which enables loose coupling between resource and service. It is particularly suitable for modular systems where the data source can change dynamically.

Conclusion

The **StringValue_AIWS** is a robust input-service interface block for wide-string data in ISOBUS systems. By combining event-driven initialization, adapter communication, and a clear interface structure, it provides a reliable foundation for processing Unicode text in industrial applications. The separation of service logic and resource interface simplifies the maintainability and reusability of the block.

StringValue_IS



• – ◦ • – ◦ • – ◦ •

Introduction

The **StringValue_IS** function block is an input service interface function block (Input SIFB) for receiving string data from a resource. It serves as a standardized interface for integrating external string information into the 4diac environment. Typical applications include connecting to ISOBUS systems, where strings such as names, status messages, or identifiers are received.

Interface Structure

Event Inputs

Event	Type	Description	Data Carried
`INIT`	EInit	Service Interface Component Initialization	`QI`, `PARAMS`, `u16ObjId`
`REQ`	Event	Request for data query or update	`QI`

Event Outputs

Event	Type	Description	Data Carried
`INITO`	EInit	Confirmation of successful initialization	`QO`, `STATUS`
`CNF`	Event	Confirmation of executed request	`QO`, `STATUS`, `IN`
`IND`	Event	Asynchronous indication – new input data from the resource	`QO`, `STATUS`, `IN`

Data Inputs

Name	Type	Initial Value	Description
`QI`	BOOL	–	Event Qualifier: Controls the execution of the associated events
`PARAMS`	STRING	–	Service parameter (e.g., configuration of access to the resource)
`u16ObjId`	UINT	`ID_NULL`	Object ID that identifies the string to be read from the resource

Data Outputs

Name	Type	Description
`QO`	BOOL	Event Qualifier: Indicates whether the output provides valid data
`STATUS`	STRING	Service status message (e.g., error codes or success message)

IN	STRING	Received string from the resource

Adapter

This function block does not have any adapter interfaces.

Functionality

The **StringValue_IS** function block operates according to the standard scheme of a service interface function block for data input:

1. **Initialization (INIT)**

The function block is configured by creating a INIT event. The parameters QI (qualifier), PARAMS (service parameter), and u16ObjId (object ID) are passed. After successful initialization, the function block confirms with the event INITO and returns the status via QO and STATUS.

1. **Request (REQ)**

A REQ event triggers an active data query from the resource. The qualifier QI is evaluated. The response is returned with the event CNF. The received string is then available at the output IN.

1. **Asynchronous Indication (IND)**

The resource can send new data at any time. The function block signals this via the event IND. Simultaneously, the current data is provided at output IN, and the status is provided via QO and STATUS.

The outputs IN, QO, and STATUS are updated with each of the events CNF and IND and can be used immediately by subsequent function blocks.

Technical Features

- **Object ID:**

The input u16ObjId is predefined with the initial value ID_NULL. Before first use, a valid object ID must be set. The actual meaning of the ID depends on the underlying resource protocol (e.g., ISOBUS object number).

- **Service Parameter (PARAMS):**

This string input is used to pass configuration-specific parameters (e.g., bus address, communication path). The exact syntax depends on the implemented driver.

- **Typical Event Sequence:**

INIT→INITO→ (possibly repeatedREQ↔CNFor asynchronousIND`).

- **Error Handling:**

Errors are signaled via the STATUS output and can be used for error diagnosis.

State Overview

The function block goes through the following basic states (not explicitly modeled in the XML, but typical for SIFBs):

- **Not Initialized:** After the system starts. Only INIT is processed.
- **Initialized:** After successful INITO. The function block is ready for REQ and can receive asynchronous IND.
- **Request in Progress:** After a REQ until the arrival of CNF. No further requests are processed during this time.
- **Error State:** In case of communication errors. Can be reset by re-running INIT.

Application Scenarios

- **ISOBUS Control Units:**

Receiving strings such as device names, variable names, or status messages from an ISOBUS server.

- **User Interfaces:**

Reading text data sent from an external HMI or operator panel.

- **Configuration Data:**

Retrieving string parameters from a central configuration service.

Comparison with Similar Function Blocks

Function Block	Data Type	Special Feature
`BOOL_IS`	BOOL	Simple Boolean inputs (e.g., switches)
`INT_IS`	INT	Integer inputs (e.g., counter readings)
REAL_IS`	REAL	Floating-point inputs (e.g., measured values)
StringValue_IS`	STRING	Strings – for names, descriptions, irregularly formatted data

StringValue_ISexpands the family of input SIFBs with the flexible data typeSTRING`, which is characterized by variable length and content.

Conclusion

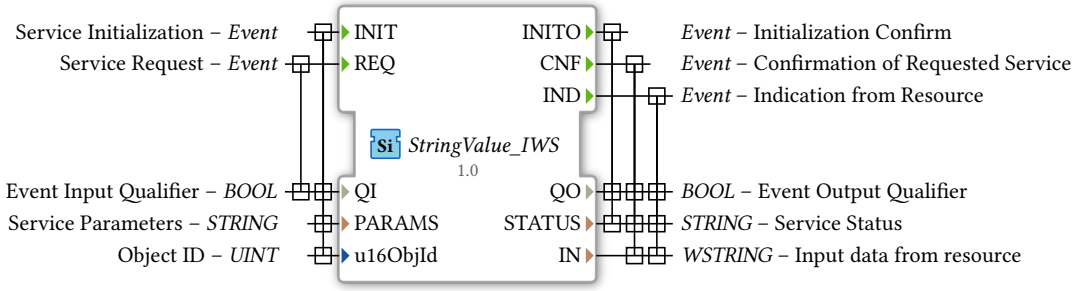
The **StringValue_IS** is an essential component for integrating string-based data into 4diac. It offers a standardized interface to the resource, enables both active queries and asynchronous indications, and can be flexibly configured using the parameters `PARAMS` and `u16ObjId`. Its use simplifies integration with protocols such as ISOBUS and increases the reusability of application logic.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

StringValue_IWS



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **StringValue_IWS** is a service interface function block (SIFB) responsible for receiving wide strings (*Wide String*, `WSTRING`) as input data from a resource-side source (e.g., hardware interface, bus system). It provides a standardized interface for managing initialization, cyclic/demand-driven data retrieval, and asynchronous indications. This function block is typically found in ISOBUS-based control environments or other fieldbus systems that process Unicode-encoded text data.

Interface Structure

Event Inputs

Event	Comment	Accompanying Data
`INIT`	Service Initialization	`QI`, `PARAMS`, `u16ObjId`
`REQ`	Service Request	`QI`

Event Outputs

Event	Comment	Accompanying Data

INITO	Initialization Confirmation	QO, STATUS
CNF	Confirmation of requested action	QO, STATUS, IN
IND	Asynchronous indication from the resource	QO, STATUS, IN

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	–	Event input qualifier
PARAMS	STRING	–	Service parameter (e.g., port/device configuration)
u16ObjId	UINT	ID_NULL	Object ID for identifying the data source

Data Outputs

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Status or Error Message
IN	WSTRING	Received Input Data (Wide String)

Adapter

None.

Functionality

The StringValue_IWS follows the typical behavior of a service interface function block:

1. **Initialization (INIT → INITO)**

The service is started by triggering the INIT event. The qualifier QI determines whether the initialization is active (TRUE) or disabling (FALSE). Necessary configuration parameters (e.g., bus address, protocol options) are passed using PARAMS. The object ID u16ObjId specifies which data source (e.g., a particular ISOBUS object) is to be addressed. After successful initialization, INITO is output; QO and STATUS indicate the initial state.

1. **Request (REQ → CNF)**

The REQ event triggers an explicit data query. The function block retrieves a new value from the connected device (e.g., via a driver). The response is provided via the output IN (as WSTRING) and the status STATUS. Simultaneously, the CNF event is sent.

1. **Asynchronous Indication (IND)**

If the resource provides new data without being prompted (e.g., when an input is changed), the IND event is triggered. IN then contains the current data, and STATUS the corresponding status. This output can be used in parallel with the CNF event.

The event outputs QO always return the current qualifier value (usually QI upon successful operation, otherwise FALSE). The status STATUS contains human-readable error or success messages.

Technical Features

- **Wide Strings (WSTRING)**

This function block uses WSTRING (UTF-16 encoded) to correctly transmit international character sets, special characters, and emojis. This is particularly important in ISOBUS agricultural technology, for example, when operator terminals display Unicode text.

- **Object ID u16ObjId**

The initial value ID_NULL is inherited from the imported constant isobus::UT::Q::const::IDs::ID_NULL. A valid object ID must be set before the INIT action to address the correct data source.

- **Service Interface Character**

The function block is implemented as a SIFB and expects a resource-side implementation (driver). The actual communication with the hardware or bus system takes place outside the function block and is abstracted via the event/data interfaces.

- **Use of `eclipse4diac::core::TypeHash`**

This attribute serves for runtime identification of the type and is used for dynamic function block instances.

State Overview

An explicit state engine is not defined in the XML; however, the following implicit process emerges from the SIFB semantics:

1. **IDLE** – Waiting for `INIT` or the first `REQ`.
2. **INIT** – Initialization phase; after successful execution, transition to **READY**.
3. **READY** – Ready for `REQ` or `IND`. Briefly enters **BUSY** upon `REQ`, then returns to **READY**.
4. **INDICATION** – Asynchronous data indexing; returns to **READY** after processing.
5. **ERROR** – In case of errors (e.g., communication interruption), `STATUS` is set and the block remains in place until `INIT` is sent again (with `QI=FALSE` as a possible reset).

Application Scenarios

- **ISOBUS Terminal (UT) – Unicode Text Input**

Receiving user input such as machine names, addresses, or diagnostic texts from an operator terminal via ISOBUS.

- **Universal Interface for Character Input**

Connecting external sensors or input devices that deliver character strings in UTF-16 format (e.g., barcode scanners, RFID readers with text output).

- **Diagnostics and Logging Interface**

Captures status messages, warnings, or error texts from a control unit for forwarding to an HMI or cloud connection.

Comparison with similar function blocks

Function block	Type	Difference
`StringValue_IWS` (present)	`WSTRING`	Receives UTF-16 encoded strings.
`StringValue_IWS` (standard version)	`STRING`	Receives ASCII/UTF-8 strings, lower character coverage.
`AnalogInput_IWS`	`REAL`	Receives analog measured values, not strings.
`EventInput_IWS`	–	Event interface only, no data.

The `StringValue_IWS`is specifically designed for processing Unicode text and is therefore suitable for international or character set-heavy applications.

Conclusion

The `StringValue_IWS` is an essential component for integrating text input into IEC 61499-based control systems, especially in ISOBUS environments. Its support for `WSTRING`, the clean separation of initialization, request, and asynchronous indication, and the flexible object ID addressing make it a robust solution for receiving long strings from distributed resources.

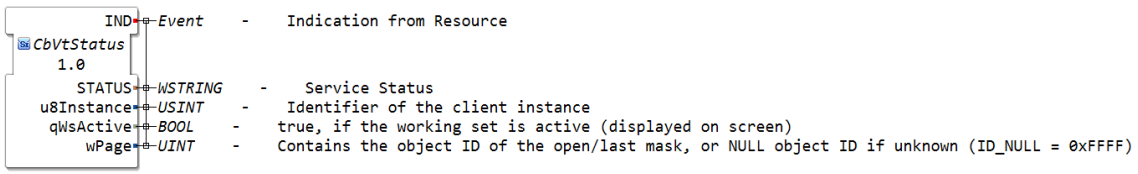
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

status

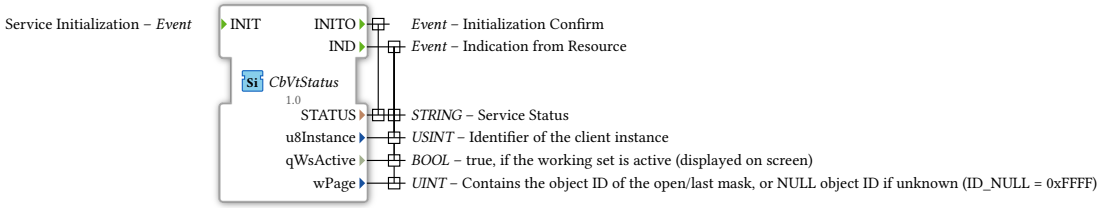
CbVtStatus



• — ○ • — ○ • — ○ •

Introduction

The **CbVtStatus** is a standardized callback module for virtual terminal page changes, developed under the EPL 2.0 license. Version 1.0 supports ISO 11783-6 (ISOBUS) in agricultural tax systems.



Interface Structure

Event Inputs

- `INIT`: Initialization Request

Event Outputs

- `INITO`: Initialization Confirmation (with STATUS)
- `IND`: Page Change Indication (with page info, `STATUS`, `wPage`, `qWsActive`, `u8Instance`, `wSKM_Scal`, `w_DMScal`, `u8SKM_Number`, `qScalingTestEnabled`)

Data Outputs

- `STATUS` (STRING): Operational Status
- `u8Instance` (USINT): Client Instance Identifier
- `qWsActive` (BOOL): Workload Status
- `wPage` (UINT): Object ID of the open/last mask, or `ID_NULL` (0xFFFF) if unknown
- `wSKM_Scal` (UINT): Softkey mask scaling factor (PoolSoftKeyMaskScalFaktor), unit 1/10000 (10000 = 100%); possibly overridden by SCALING-TEST
- `w_DMScal` (UINT): Data mask scaling factor (PoolDataMaskScalFaktor), unit 1/10000 (10000 = 100%); possibly overridden by SCALING-TEST
- `u8SKM_Number` (USINT): Number of physical softkeys (VT_PHYSICALSOFTKEYS/sknu), valid range 6..11; possibly overridden by SCALING-TEST
- `qScalingTestEnabled` (BOOL): `TRUE` if the SCALING-TEST override (`[ScalingTest] enable=1` in `settings.ini`) is active

Functionality

1. **Initialization**:

- Event `INIT` starts setup
- Event `INITO` confirms with STATUS

1. **Page Change**:

- Event `IND` is triggered on screen change
- Returns: Page ID, Active Status, Instance Number

1. **Operation**:

- Real-time monitoring of the VT status
- Multi-ECU capable architecture

Technical Features

✓ **ISO 11783-6 compliant** ✓ **Deterministic** behavior ✓ **Multi-ECU support** ✓ **Plug-in capable** architecture

Supported Operating Modes

Mode	Description
Single device	Standard operation with u8Instance=0
Multi-ECU	Parallel Instances (1-255)
Diagnostics	STATUS Messages for Service

Application Scenarios

- **Tractor Control Units**: Operator Screen Monitoring
- **Harvesting Machines**: Work Data Display
- **Diagnostic Tools**: VT Status Monitoring
- **Multi-Display Systems**: Synchronization

⚖️ **Comparison with Similar Modules**

Feature	CbVtStatus	VtPageManager	VtEventLogger
Page Switch	✓	✓	✖
Multi-ECU	✓	✖	✖
STATUS feedback	✓	✖	✓

Error Handling

- Invalid Page ID:
- STATUS = “ERR_INVALID_PAGE”
- wPage = 0xFFFF
- Initialization Error:
- STATUS = “ERR_INIT_FAILED”
- System in Safe State

🛠️ **Related Exercises**

- Exercise_012
- Exercise_012a
- Exercise_012b
- Exercise_012c

Conclusion

The CbVtStatus module is essential for ISOBUS-compliant agricultural systems:

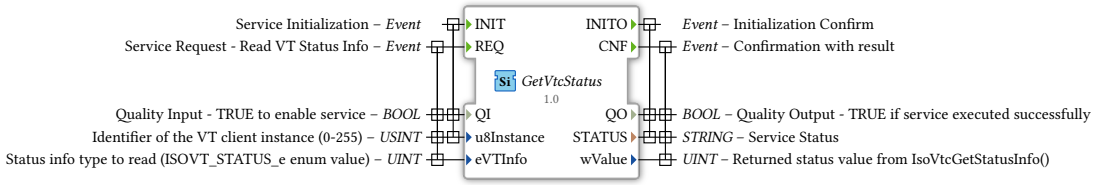
- Robust page change management
- Full ISO 11783-6 compatibility
- Proven architecture In series production

Especially valuable for manufacturers of:

- Tractor control units
- Harvesting machine control systems
- ISOBUS diagnostic devices

The standardized implementation enables reliable operation in heterogeneous system environments.

GetVtcStatus



• – ○ • – ○ • – ○ •

Introduction

The function block **GetVtcStatus** is a Service Interface Block (SIFB) that encapsulates the function `IsoVtcGetStatusInfo()` from the ISOBUS driver. It enables the reading of various status information from a VT client and the connected Virtual Terminal (VT). The block was developed according to the ISO 11783-6 standard and is suitable for integration into ISOBUS applications.

Interface Structure

Event Inputs

Event	Type	Description	Data Carried
`INIT`	EInit	Initializes the function block.	`QI`, `u8Instance`
`REQ`	Event	Requests the reading of a status value.	`QI`, `u8Instance`, `eVTInfo`

Event Outputs

Event	Type	Description	Data Carried
`INITO`	EInit	Confirmation of initialization.	`QO`, `STATUS`
`CNF`	Event	Confirmation of the read request with result.	`QO`, `STATUS`, `wValue`

Data Inputs

Name	Type	Initial Value	Description
`QI`	BOOL	–	Quality input: TRUE activates the service.
`u8Instance`	USINT	–	VT client instance identifier (0–255).
`eVTInfo`	UINT	0	Type of status information to query (values from the enumeration `ISOVT_STATUS_e`).

Data Outputs

Name	Type	Description
`QO`	BOOL	Quality output: TRUE upon successful execution.
`STATUS`	STRING	Service status – contains a textual response.
`wValue`	UINT	The value returned by `IsoVtcGetStatusInfo()` (dependent on `eVTInfo`).

Adapters

No adapters available.

Functionality

1. **Initialization**

The module is activated by the event `INIT`. The parameter `QI` must be set to TRUE for the service to start. The VT client instance to be used is specified via `u8Instance`. After successful initialization, the event `INITO` is sent with the output data `QO` and `STATUS`.

1. **Reading Status Information**

The event `REQ` initiates a specific query. The desired status value must be selected from the enumeration `ISOVT_STATUS_e` using `eVTInfo` (see the list in the technical specifications). Internally, the function block calls the function `IsoVtcGetStatusInfo()` and sends the event `CNF` upon completion. The outputs contain:

- `QO` = TRUE if the read operation was successful,
- `STATUS` = description of the result,
- `wValue` = the retrieved numerical value.

If `QI` is FALSE during the REQ call, the service is not executed and a corresponding error status is returned.

Technical Features

- The function block is implemented as a **Service Interface Block** (SIFB) and accesses the system-level ISOBUS driver library.
- The possible values for `eVTInfo` (ISOVT_STATUS_e) are:

Value	Label	Description
0	VT_SOURCE_ADDRESS	Source address of the VT
2	VT_HND	CF handle of the VT
3	CF_SOURCE_ADDRESS	Source address of the VT client
4	CF_HND	CF handle of the VT client
6	ID_VISIBLE_DATA_MASK	Data mask open on the VT
7	ID_VISIBLE_SOFTKEY_MASK	Softkey mask open on the VT
8	VT_BUSY_CODE	Busy code of the VT status message
9	AUXUNITS_TYPE1_ONBUS	TRUE if a Type 1 auxiliary device is on the bus
11	VT_ALIVE	VT unreachable for more than 3 seconds
12	VT_DOWNLOAD_FINISHED	Full announcement completed
13	VT_POOL_ACTIVE_onVT	Pool (selected) active on the VT
14	VT_STATEOFANNOUNCING	Announcing status
15	WS_VERSION_NR	Working Set version number
16	VT_NUMBOFVERSIONSTRINGS	Number of VT version strings sent
17	VT_NAVSOFTKEYS	Navigation softkeys (version 4)
18	VT_SOFTKEYXDOT	Softkey identifier – pixel X
19	VT_SOFTKEYYDOT	Softkey identifier – pixel Y
20	VT_VIRTUALSOFTKEYS	Number of virtual softkeys
21	VT_PHYSICALSOFTKEYS	Number of physical softkeys
25	VT_BOOTTIME	VT boot time
26	VT_GRAPHICTYPE	VT hardware graphics type
29	VT_VERSIONNR	Version of Working Set VT

- The function block supports both initialization and repeated calls to read operations, allowing different status values to be queried within a loop.

State Overview

The function block implements simple state-driven logic:

1. **Start State** – The function block waits for the event `INIT`.
2. **Initialization** – Upon receiving `INIT` (with a valid QI), the service is initialized. If successful, `INITO` is sent with QO=TRUE. If an error occurs, QO=FALSE and a corresponding STATUS are sent.
3. **Ready** – After successful initialization, any number of `REQ` events can be processed. Each `REQ` call triggers a call to `IsoVtcGetStatusInfo()` and the subsequent sending of `CNF`.
4. **Error Handling** – If an internal error occurs during a REQ, `CNF` is output with QO=FALSE and an error description in `STATUS`.

Application Scenarios

- **Diagnostics and Monitoring** of an ISOBUS system: Reading the current VT address, graphics type, or boot time.
- **Detection of VT Availability** by querying `VT_ALIVE` and `VT_DOWNLOAD_FINISHED`.
- **Control of the softkey configuration** by querying the number of physical and virtual softkeys (`VT_PHYSICALSOFTKEYS`, `VT_VIRTUALSOFTKEYS`).
- **Comparison of the working set version** between different components using `VT_VERSIONNR` and `WS_VERSION_NR`.

Comparison with similar modules

Similar modules such as `GetVtPool` or `GetVtObject` focus on reading pool or object data. The `GetVtcStatus` module, on the other hand, is specifically designed for accessing the general VT client status. It provides a direct interface to the system status properties of the ISOBUS specification and complements function blocks that access mask or softkey information.

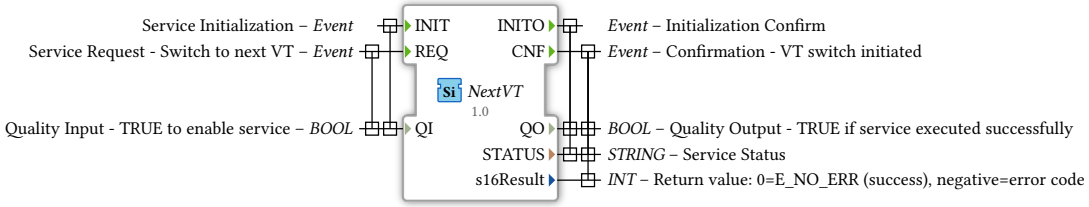
Conclusion

The function block `GetVtcStatus`It provides a compact and standards-compliant way to access the system-level status data of an ISOBUS virtual terminal. Its clear interface and diverse query options make it a valuable tool for implementing diagnostic, monitoring, and configuration functions in agricultural control systems.

Numerical Index (UT)

Here you will find all exercises from the `UT` workspace in ascending order. …

NextVT



• – ◦ • – ◦ • – ◦ •

Introduction

The **NextVT** function block is a service interface block that enables switching to the next available Virtual Terminal (VT) in an ISOBUS network (ISO 11783-6). It encapsulates the functionality of `VTC_NextVTButtonPressed()` from the ISOBUS driver examples and allows seamless switching between multiple VT devices.

Interface Structure

Event Inputs

Event	Data Type	With Variables	Description
INIT	EInit	QI	Service Initialization. Triggered to configure the block.

REQ	Event	QI	Service request. Starts the process to switch to the next available VT.

Event Outputs

Event	Data Type	With Variables	Description
INITO	EInit	QO, STATUS	Initialization confirmation. Signals whether the initialization was successful.
CNF	Event	QO, STATUS, s16Result	Switchover confirmation. Outputs as soon as the switch to the next VT has been initiated.

Data Inputs

Variable	Type	Description
QI	BOOL	Quality input. TRUE activates the service. FALSE means no action is taken.

Data Outputs

Variable	Type	Description
QO	BOOL	Quality output. TRUE if the service was successful.
STATUS	STRING	Status message regarding the result of the last operation.
s16Result	INT	Return value: 0 = E_NO_ERR (successful), negative value = error code.

Adapters

No adapters available.

Functionality

When a **REQ** event is received, the block performs the following steps:

1. **Determine list of all VTs** – All virtual terminals available on the ISOBUS network are identified using `IsoClientsReadListofExtHandles()`.
2. **Find currently connected VT** – The currently used VT is identified using `IsoVtcGetStatusInfo(VT_HND)`.
3. **Determine next VT** – The next entry after the current VT is selected from the list (cyclic sequence).
4. **Execute switchover** – The switchover to the next VT is initiated using `IsoVtcMultipleNextVT()`.

- **Error case:** If only one VT is present on the bus or the switchover fails, a negative return value (`s16Result`) and a corresponding status message are output.
- **Important Note:** The application must enter a safe state after calling this block, as the connection to the current VT is lost during the transition.

Technical Features

- **Standard Compliance:** The block is based on ISO 11783-6 (ISOBUS) and implements the VT switching function according to the standard.
- **Service Interface Block:** Execution is asynchronous; the **CNF** event signals the completion of the switching process.
- **Initialization:** Before first use, the block must be initialized via the **INIT** event to allocate internal resources.

State Overview

The block does not have explicit states in its XML definition. Its behavior is determined by event-driven processing:

- **Initialization Phase:** After **INIT** with `QI=TRUE`, initialization is performed and acknowledged by **INITO**.

- **Operational Phase:** After **REQ** with `QI=TRUE`, the switching process is initiated. Upon completion, the **CNF** event occurs.
- **Error Handling:** In case of errors, `QO=FALSE` and a corresponding error code are set in `s16Result`.

Application Scenarios

- **ISOBUS Machine Operation:** Switching between multiple Virtual Terminals in agricultural vehicles, e.g., to switch between the display and the on-board computer.
- **Multi-Monitor Systems:** Switching the active display to a different VT device on the network.
- **Diagnostics and Testing:** Simulating the pressing of the "Next VT" button during the development of ISOBUS-compliant applications.

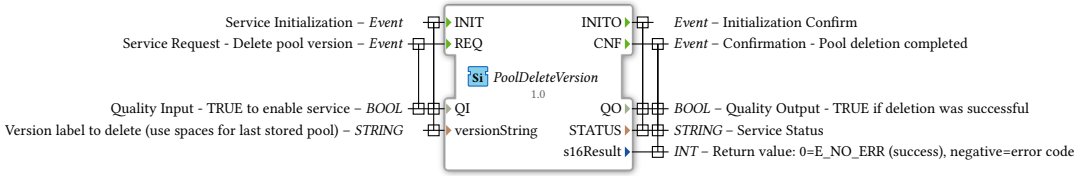
Comparison with Similar Blocks

- **VTSelection** (hypothetical): A block that selects a specific VT object instead of simply jumping to the next one. NextVT is simpler and follows standard key logic.
- **VTConnect** (hypothetical): Establishes a direct connection to a named VT. NextVT, on the other hand, works automatically and does not require a target address.

Conclusion

The **NextVT** function block is a specialized service interface block for ISOBUS applications that enables standards-compliant switching to the next available virtual terminal. Its simple interface (INIT, REQ) and clear error feedback make it ideally suited for integration into control software that requires flexible VT selection. However, the application must account for the brief connection interruption.

PoolDeleteVersion



• – ○ • – ○ • – ○ •

Introduction

The service interface block `PoolDeleteVersion` deletes a stored object pool version from the non-volatile memory of a Virtual Terminal (VT). It encapsulates the function `VTC_PoolDeleteVersion()` from the ISOBUS driver. The block calls `IsoVtcCmd_DeleteVersion()` to remove the specified pool version.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Service Initialization
REQ	Event	Service Request – Delete Pool Version

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirmation
CNF	Event	Confirmation – Deletion complete

Data Inputs

Name	Type	Comment	Initial Value
QI	BOOL	Quality Input – TRUE activates the service	

versionString	STRING	Version label to be deleted (space for last saved version)	' '

Data Outputs

Name	Type	Comment
QO	BOOL	Quality output – TRUE on successful deletion
STATUS	STRING	Service status
s16Result	INT	Return value: 0 = E_NO_ERR (success), negative = error code

Adapter

None.

Functionality

The block is initialized via the event input `INIT`, passing the version number to be deleted (`versionString`). After successful initialization, an event `REQ` triggers the deletion process. The block calls the system-level function `IsoVtcCmd_DeleteVersion()`, which removes the specified pool version from the VT's flash memory. The result is signaled via the outputs `QO`, `STATUS`, and `s16Result`, triggering a corresponding confirmation event (`CNF` or `INITO`).

Technical Specifications

- `versionString` must be null-terminated or exactly 32 bytes long (ISO V11+).
- If a string consisting of spaces (' ') is passed, the function deletes the last stored pool version.
- Maximum version entry length: 32 bytes.
- The block is designed as a Service Interface Block (SIB) for ISOBUS communication (ISO 11783-6).

State Overview

The block does not have any explicitly defined states in the XML. Control is event-driven via `INIT` and `REQ`:

1. **Initialization** (INIT) → Acknowledge (INITO)
2. **Request** (REQ) → Acknowledge (CNF) with result

Failed initialization or deletion is signaled by `QO=FALSE` and the corresponding `STATUS` message.

Application Scenarios

- Deleting old pool versions before uploading a new version
- Freeing up flash memory space on the VT
- Removing corrupted or no longer needed pool versions
- Resetting to the factory-stored version by deleting all user-defined versions

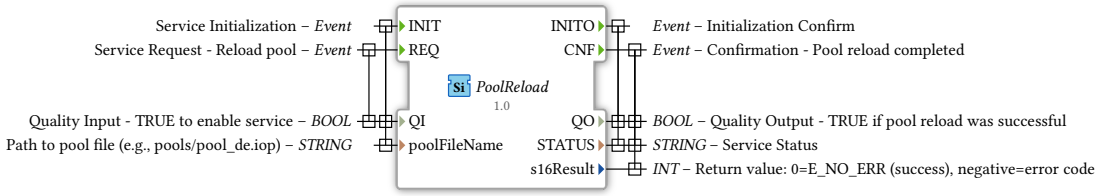
Comparison with Similar Blocks

There is no direct comparison to other function blocks. This block is specifically for managing ISOBUS object pools in VT flash memory.

Conclusion

PoolDeleteVersion` provides a standardized and reliable way to delete stored object pool versions from the flash memory of a virtual terminal. Its simple event-driven interface makes it easy to integrate into automation and control applications. Adherence to ISO specifications (version string length, spaces for the last version) makes the block particularly robust for use in agricultural machinery according to ISO 11783.

PoolReload



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `PoolReload` is a service interface block according to ISO 11783-6 (ISOBUS). It enables the reloading or updating of the object pool of a Virtual Terminal (VT) during application runtime. This block is typically used, for example, to switch between different language variants or to dynamically load modified pool files without requiring a system restart.

Interface Structure

Event Inputs

Event	Type	Comment
`INIT`	`EInit`	Service initialization (with parameters `QI` and `poolFileName`)
`REQ`	`Event`	Service request – performs pool reloading (with `QI`)

Event Outputs

Event	Type	Comment
`INITO`	`EInit`	Initialization confirmation (outputs `QO` and `STATUS`)
`CNF`	`Event`	Confirmation – Pool reload complete (outputs `QO`, `STATUS`, and `s16Result`)

Data Inputs

Name	Type	Comment
`QI`	`BOOL`	Quality Input: `TRUE` activates the service
`poolFileName`	`STRING`	Path to the pool file (e.g., `pools/pool_de.iop`)

Data Outputs

Name	Type	Comment
`QO`	`BOOL`	Quality Output: `TRUE`, if the reload was successful
`STATUS`	`STRING`	Service status (e.g., error message or success message)
`s16Result`	`INT`	Return value: `0` = `E_NO_ERR` (success), negative values correspond to error codes

Adapter

No adapters available.

Functionality

This function block encapsulates the ISOBUS function `VTC_PoolReload()`. The process is divided into the following steps:

1. **Initialization (`INIT`)**

- The pool data is loaded from the file specified in `poolFileName`.
- The pool is opened for the configured color depth.

- After successful loading, the function block returns a confirmation message via `INITO`.

1. **Service Execution (`REQ`)**

- This function block calls `IsoVtcPoolUpdate()` to update the pool on the VT.
- Optionally, ID range modes can be applied for pool manipulation.
- Upon completion of the operation, a `CNF` event is triggered, reporting success or failure (via `s16Result`).

Technical Features

- **Standard Compliance**: This function block complies with the ISOBUS standard ISO 11783-6 (Agricultural vehicles – Virtual Terminal).
- **File Path**: By default, the pool file is expected to be located at `pools/pool_de.iop`. However, the path can be configured via the input `poolFileName`.
- **File Path**: By default, the pool file is expected to be located at `pools/pool_de.iop`. However, the path can be configured via the input `poolFileName`. - **Runtime Update**: Unlike a static pool import, this function block allows a dynamic update without requiring an application restart.
- **Error Handling**: The output `s16Result` returns the detailed ISOBUS error code (0 = success, negative values = error).

State Overview

The function block can go through the following basic states:

State	Description
IDLE	Waiting for an INIT event.
INIT_PENDING	Initialization is being performed; upon completion, `INITO` is sent.
READY	Ready for `REQ` after successful initialization.
REQ_PENDING	Pool reloading is in progress; Upon completion, `CNF` is sent.
ERROR	If initialization or reloading fails, an error status is reported, and the function block remains in the error state until it is re-initialized.

Application Scenarios

- **Language Switching**: During runtime, it is possible to switch between different language variants (e.g., German, English) by replacing the pool file.
- **Dynamic Pool Updates**: New screens or symbols can be loaded without restarting the entire VT – useful for firmware updates or field modifications.
- **Last-Minute Changes**: Modified pool files can be quickly loaded during commissioning.

Comparison with Similar Function Blocks

Function Block	Description
PoolLoader`	Loads the pool only at system startup; no reloading at runtime.
PoolActivate`	Switches between already loaded pools, but requires prior loading.
PoolReload`	Combines loading and updating in one step and enables dynamic reloading during runtime.

Conclusion

The `PoolReload` function block is a specialized service interface module for ISOBUS virtual terminals that enables flexible reloading and updating of object pools at runtime. It simplifies language switching, dynamic updates, and facilitates the maintenance of VT applications. Its clear event interface and detailed error feedback make it well-suited for use in agricultural control systems.

2.10.2.5 pgn

ADDRESS



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ADDRESS` is a global constant block that defines special address constants for use in ISOBUS-based applications. It provides predefined values for source and destination addresses (SA), which are particularly important in ISOBUS communication. This block serves to make the source code more readable and maintainable by replacing magic numbers with meaningful names.

Interface Structure

This block is a `GlobalConstants` block and therefore does not have traditional event or data interfaces. Instead, it defines constants that can be used globally in other function blocks of the `isobus::pgn` package.

Event Inputs

Not present.

Event Outputs

Not present.

Data Inputs

Not present.

Data Outputs

Not present.

Adapters

Not present.

Functionality

The block `ADDRESS` declares two constants:

1. `NULL_A` (Type: `BYTE`, Value: `254`): This constant represents the NULL address (`0xFE`). Messages with this source address (SA) may only be sent by a sender that has a network manager.
2. `GLOBAL_A` (Type: `INT`, Value: `255`): This constant represents the global address (`0xFF`). This address is used exclusively in the Destination Address Field to broadcast a message to all participants in the network.

These constants are inserted into the code at compile time and cannot be changed at runtime.

Technical Features

- **Type Difference:** It is noteworthy that `NULL_A` is declared as `BYTE` and `GLOBAL_A` as `INT`. However, both values are within the representable range of both data types. The use of `INT` for `GLOBAL_A` could indicate future extensibility or compatibility with other parts of the system.
- **Package Assignment:** This block is part of the `isobus::pgn` package, indicating its use in the Parameter Group Number (PGN) communication of the ISOBUS protocol.

State Overview

Since it is a constant block, it has no state machine and no internal states.

Application Scenarios

The `ADDRESS` block is used in ISOBUS applications, for example, in agricultural machinery (tractors, implements). Typical scenarios include:

- **Message Configuration:** These constants are used to populate the source or destination address field when constructing PGN (Parameter Group Number) messages.
- **Network Management:** The `NULL_A` address is used in special network management messages.
- **Broadcast Communication:** The `GLOBAL_A` address is used to send commands or data to all connected devices on the ISOBUS network.

⚖️ Comparison with Similar Function Blocks

There is no directly equivalent function block in the IEC 61499 standard library, as global constants are a language-specific extension (derived from Structured Text) within the 4diac IDE. A similar purpose could be achieved by:

- **Direct Value Input (Magic Numbers):** Directly writing the values `254` or `255` in the code is possible, but poor practice, as it impairs readability and maintainability.

There is no directly equivalent function block in the IEC 61499 standard library, as global constants are a language-specific extension (derived from Structured Text) within the 4diac IDE. * **Constant Blocks in Other Languages:** In function block programming languages like CFC or SFC, comparable constants are often defined in separate header files or global variable lists.

The advantage of the `ADDRESS` block lies in its centralized, package-bound definition and semantic clarity.

Conclusion

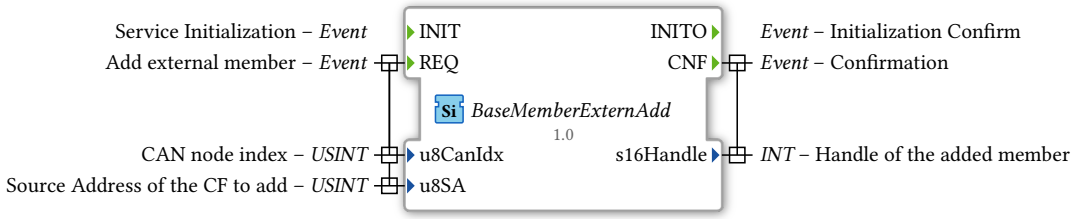
The `ADDRESS` function block is a simple yet essential building block for developing ISOBUS-compliant applications in the 4diac IDE. It improves code quality by replacing special numerical values with self-explanatory identifiers, thus preventing errors and simplifying maintenance. Its use is particularly recommended for implementing network management and broadcast functionalities.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

BaseMemberExternAdd



Adds an external member to the network list manually. The use of this function is not ISO 11783 Part 5 conform. Use this function only if you must communicate with an external CF, that does not send an address claimed message. An address claimed message from an external CF with the same source address will replace this manual added member entry. The function works independently from the configuration of the #CFG_PARAM_EXTERNAL_ADDR_CLAIM parameter. The added CF will be set to announced in the next iso_DlCyclicGlobal() run.

Interface

Event inputs

Name	Comment	With
INIT	Service Initialization	
REQ	Add external member	u8SA, u8CanIdx

Event Outputs

Name	Comment	With

INITO	Initialization Confirm	
CNF	Confirmation	s16Handle

Input Vars

Name	Type	Initial Value	Comment
u8CanIdx	USINT	NODE1	CAN node index
u8SA	USINT		Source Address of the CF to add

Output Vars

Name	Type	Comment
s16Handle	INT	Handle of the added member

🛠️ Related exercises

- Uebung_134

CAN_MSG

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `CAN_MSG` is a structured type definition (struct) for representing a classic CAN message frame (without a transport protocol, TP). It serves as a container for the essential attributes of a CAN message, including priority, payload length, and the actual data buffer. This type is intended for use in function blocks that implement CAN communication in an IEC 61499 system.

Interface Structure

Since `CAN_MSG` is a data type and not a function block, there are no event or adapter interfaces. The structure is defined exclusively by its data members.

Data Inputs

(Not applicable – Data Type)

Data Outputs

(Not applicable – Data Type)

Event Inputs

(Not applicable – Data Type)

Event Outputs

(Not applicable – Data Type)

Adapters

(Not applicable – Data Type)

Functionality

The `CAN_MSG` type is a passive data structure. It encapsulates the information relevant for a CAN message:

1. **Priority (`u8Priority`)**: Defines the arbitration priority on the CAN bus. A lower numerical value corresponds to a higher priority.
2. **Data Length (`u16DaSize`)**: Specifies the actual number of bytes used in the data buffer (0 to 8). 3. **Data Buffer (`data`)**: An 8-byte array that stores the message payload. Unused bytes are initialized with a default value.

This structure can be used as an input or output data type for function blocks that send, receive, process, or log CAN messages.

Technical Features

- **Fixed Buffer Size**: The data array is statically dimensioned to a length of 8 bytes, which corresponds to the maximum payload length of a standard CAN frame.
- **Initialization**: By default, the data array is pre-populated with the hexadecimal value `16#FF` (decimal 255) in all 8 positions. The data length `u16DaSize` defaults to `0`, and the priority `u8Priority` defaults to `7` (lowest priority).
- **Type Safety**: Using a structured type instead of individual variables improves maintainability and clarity in the function block connection view.

State Overview

(Not applicable – the data type has no behavior or state machine.)

Application Scenarios

The `CAN_MSG` type is fundamental for:

- **CAN Sender FBs**: An FB that receives a `CAN_MSG` type as an input and sends its contents as a physical CAN frame to the bus.
- **CAN Receiver FBs**: An FB that packs received CAN frames into a variable of type `CAN_MSG` and forwards them to downstream logic.
- **Message Construction**: FBs that construct a valid `CAN_MSG` structure for later transmission from individual values (e.g., from sensor FBs).
- **Message Filtering/Processing**: FBs that analyze and process incoming `CAN_MSG` structures based on priority, data content, or length.

⚖️ Comparison with similar building blocks

As a data type, `CAN_MSG` is comparable to other structured types in 4diac that represent communication protocols (e.g., `UDP_PACKET`, `MODBUS_MSG`). Its specific characteristic is its close mapping to the hardware properties of the CAN bus protocol (priority, 8-byte limit). Unlike a TP-based message type (e.g., for ISO 11783 or CANopen), it lacks fields for packetization, sequence numbers, or specific protocol headers.

Conclusion

The `CAN_MSG` data type provides an essential and well-structured foundation for implementing CAN bus communication in 4diac-based control systems. By consolidating priority, length, and data into a single type, the handling of CAN messages in function block networks is significantly simplified and standardized. Its use is recommended for all applications requiring direct access to CAN frames.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

CF_INFO_T

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `CF_INFO_T` (Control Function Information Type) is a structured data component used within ISO 11783 (ISOBUS) communication. It serves for the standardized representation and management of information about a participant (control function or user) in the ISOBUS network. This type definition summarizes all essential characteristics and states of a network participant in a single data structure.

Interface Structure

CF_INFO_T` is a structured data type and does not have any event or adapter interfaces in the conventional sense of a function block. Its structure consists exclusively of data elements (variables).

Data Inputs

Not present, as it is a pure data type.

Data Outputs

Not present, as it is a pure data type.

Data Elements (Structure)

The internal structure of type `CF_INFO_T` consists of the following elements:

Name	Type	Comment / Description	Initial Value
`s16Handle`	`INT`	Network member handle of the Control Function (CF) / User. Serves as a unique identifier within the application.	`UNVALID`
`u8SourceAddress`	`BYTE`	Source Address (SA) of the Control Function (CF) / User in the ISOBUS network.	`NULL_A`
`u8CanIdx`	`BYTE`	CAN bus node / index of the control function (CF) / user. Identifies the physical bus node.	–
`au8Name`	`isobus::pgn::CF_NAME_T`	The ISOBUS name of the control function. This is a 64-bit unique identifier according to ISO 11783-5.	–
`eIsoUserFunct`	`SINT`	Internally used CF function type (e.g., work device, terminal, task controller).	`undefined`
`bExternalCf`	`BOOL`	Indicates whether it is an external CF in the CAN network (`TRUE`) or not.	–
`bIsAnnounced`	`BOOL`	`TRUE` if the CF is active and fully announced in the network.	–
`bIsActive`	`BOOL`	NIU-specific (Network Interface Unit): `TRUE` if the CF is not in an error state or logged off.	–

Functionality

The data type `CF_INFO_T` is a passive data storage structure. It is used to store the state and identity of an ISOBUS network participant and to exchange this information between different components of a control application. The information it contains is typically read, written, and interpreted by other function blocks responsible for network management (e.g., address assignment, name management) or application logic.

Technical Features

- **Typedef for ISOBUS:** Specifically designed for use in ISO 11783 environments (agricultural machinery).
- **State Flags:** Contains several Boolean flags (`bIsAnnounced`, `bIsActive`) that represent the lifecycle and operating state of a function block.
- **Initial Values:** Critical identifiers such as `s16Handle` and `u8SourceAddress` are initialized with defined constants (`UNVALID`, `NULL_A`) to ensure an invalid output state.
- **Integration:** Uses the user-defined type `isobus::pgn::CF_NAME_T` to map the 64-bit ISOBUS name.

State Overview

Since this is a data type and not an active function block, there is no state machine. The "state" is described by the combination of the included data fields, in particular by the flags:

- **Invalid/Inactive:** `s16Handle == UNVALID`, `bIsAnnounced == FALSE`
- **Valid & Active:** `s16Handle != UNVALID`, `bIsAnnounced == TRUE`, `bIsActive == TRUE`
- **Valid & Faulty/Inactive:** `s16Handle != UNVALID`, `bIsActive == FALSE`

Application Scenarios

- **Participant Management:** In an ISOBUS manager block that manages a list of all known network participants (`CF_INFO_T[]`).

- **Diagnostics and Display:** Forwarding participant information to a diagnostic or display system to show the network status.
- **Command Forwarding:** As a parameter for communication blocks to specifically address messages to a particular CF (identified by `u8SourceAddress` and `au8Name`).
- **Filtering:** Logic blocks can use the fields `eIsoUserFunct` or `bExternalCf` to decide whether messages from a specific CF should be processed.

⚖️ Comparison with Similar Blocks

CF_INFO_Tis not a function block and therefore cannot be directly compared with blocks such asE_SRorE_CYCLE. Within ISOBUS communication, it is comparable to other structured data types that bundle network or device information (e.g., PGN_INFO_T` for Parameter Group Numbers). Its unique feature lies in the specific compilation of data required for the identification and state management of a control function.

Conclusion

The data type `CF_INFO_T` is a fundamental and essential component for implementing ISOBUS-enabled control applications in 4diac. It provides a standardized, type-safe method for grouping and transporting all relevant information about a network participant. Its use increases code reusability and ensures consistency in network communication logic.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

CF_NAME_T

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `CF_NAME_T` is a structured data type used in the 4diac IDE to model ISO CF (Communication Function) or user information data arrays. It serves as a type definition for a fixed-length byte array and is typically used in the context of ISOBUS PGN (Parameter Group Number) communication.

Interface Structure

CF_NAME_T` is a structured data type (STRUCT) and has no event or adapter interfaces. Its structure consists of a single data field.

Data Inputs

This data type itself has no inputs. It defines the structure for data.

Data Outputs

This data type itself has no outputs. It defines the structure for data.

Structured Elements

- **`data`** (BYTE[0..7]): An array of exactly 8 bytes (`BYTE`). Each element in the array is initialized to the hexadecimal value `16#FF` (decimal 255) by default. This represents a typical “invalid” or “empty” initial state in communication protocols.

Functionality

The data type `CF_NAME_T` encapsulates a byte array with a fixed length of 8 bytes. Its main function is to provide a type-safe and reusable definition for data packets exchanged in ISOBUS-based control systems. The predefined initialization of all bytes to `0xFF` ensures that instances of this type begin in a known, neutral state.

Technical Features

- **Fixed Array Size:** The array is limited to indices 0 through 7, corresponding to a fixed length of 8 bytes. This is a common size for many message blocks in fieldbus systems.

- **Predefined Initialization:** All array elements are automatically initialized with the value `16#FF` when a variable of this type is declared, unless otherwise specified.
- **ISOBUS Context:** This type is part of the `isobus::pgn` package, indicating its specific use in agricultural and mobile machine control systems (ISOBUS, ISO 11783).
-

Application Scenarios

This data type is primarily used to represent the payload of ISOBUS PGN messages within function blocks (FBs). Typical applications include:

- Declaring input or output variables in FBs that send or receive ISOBUS messages.
- Structuring data for communication between different electronic control units (ECUs) in a mobile machine.
- Using it as part of a larger, structured data type that combines multiple parameters of a complex message.

⚖️ Comparison with Similar Blocks

CF_NAME_T` is a data type, not a function block. It can be compared to other structured data types (STRUCT) in 4diac that define arrays. Its distinguishing features are its fixed 8-byte length and its ISOBUS-specific semantics. A generic `ARRAY [0..7] OF BYTE` would be functionally similar, but it lacks the type-safe reusability and contextual reference.

Conclusion

The `CF_NAME_T` data type is a specialized, reusable construct for the 4diac IDE that standardizes and simplifies the handling of 8-byte data blocks in ISOBUS applications. Its fixed size and predefined initialization increase the reliability and readability of application code based on the ISO 11783 standard.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

CommunicationPartners



• – ○ • – ○ • – ○ •

Introduction

CommunicationPartners` is a global constant data set used in the context of ISOBUS PGN (Parameter Group Number) communication. It serves to provide a predefined communication partner, in this case the global address, as a constant configuration for other function blocks. This simplifies the configuration of ISOBUS network components by defining frequently used addresses centrally and for reuse.

Interface Structure

Because it is a global constant data set, `CommunicationPartners` does not have event-driven or data-flow-based interfaces in the conventional sense of a function block. Instead, it provides a named, constant variable for the entire application.

Data Inputs

None.

Data Outputs

None.

Event Inputs

None.

Event Outputs

None.

Adapters

None.

Functionality

The data record `CommunicationPartners` declares a global constant named `GLOBAL` of data type `isobus::pgn::CF_INFO_T`. This structure is initialized with a value that sets the field `s16Handle` to the predefined value `GLOBAL_A` from the imported constant `isobus::pgn::ADDRESS::GLOBAL_A`.

During application compilation and execution, this constant is stored in memory and can be referenced by any function block within the project that requires an ISOBUS address of type `CF_INFO_T`. Using this constant ensures that the global address is used consistently and without typos throughout the code.

Technical Details

- **Data Type:** The constant is of type `isobus::pgn::CF_INFO_T`. This is likely a structured (STRUCT) or enumeration (ENUM) type specifically defined for the ISOBUS communication configuration.
- **Initialization:** Initialization occurs directly in the declaration using the structured initialization `(s16Handle := GLOBAL_A)`.
- **Import:** The data record imports an external constant (`GLOBAL_A`) from the package `isobus::pgn::ADDRESS`. This demonstrates a modular architecture where address definitions are maintained centrally.
- – **Compilation:** The `CompilerInfo` tag indicates that this element is part of the `isobus::pgn` package and is relevant for code generation.

State Overview

Since this is a static, constant definition, there is no state machine or mutable state.

Application Scenarios

- **ISOBUS Network Configuration:** Used when a function block needs to send a message to all participants in the ISOBUS network (broadcast). The constant `GLOBAL` represents this broadcast address.
- **Simplifying Parameterization:** In complex applications with many communicating blocks, using `CommunicationPartners.GLOBAL` replaces the repeated, hard-coded entry of the global address, thus increasing maintainability.
- **Reference for other constants:** Can serve as a basis for defining further, more specific communication partner constants (e.g., `IMPLEMENT`, `TOOL`).

⚖️ Comparison with similar building blocks

- **Direct value assignment:** Instead of using `CommunicationPartners.GLOBAL`, a developer could also directly assign the value `GLOBAL_A` to each data input of a function block. However, using the constants is cleaner, more type-safe, and easier to modify.
- **Other address constants:** Within the same package (`isobus::pgn::ADDRESS`), similar constants for other addresses likely exist (e.g., `NULL_A`, `TOOL_A`). `CommunicationPartners` encapsulates a specific selection or combination of these addresses in a context relevant to the application.
- **Parameter FBs vs. Global Constants:** An alternative approach would be a function block that receives its target address as a configurable parameter. Global constants, on the other hand, are simpler and more direct if the value remains unchanged for the entire runtime of the application.

Conclusion

The global constant dataset `CommunicationPartners` is a simple yet useful auxiliary construct for developing ISOBUS-enabled controllers in 4diac. It promotes the principles of "Don't Repeat Yourself" (DRY) and centralized configuration management by defining the frequently used global ISOBUS address in a single, easily visible location. Its use reduces potential sources of error and facilitates future network configuration adjustments.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

HANDLE



• – ◦ • – ◦ • – ◦ •

Introduction

The `HANDLE` constant is a global constant definition for the 4diac IDE, used in the context of ISOBUS protocol and parameter group number (PGN) processing. It defines special values to identify invalid or global addressing targets within an ISOBUS network. These constants serve as central reference values for function blocks that use handle-based communication.

Interface Structure

This is a global constant definition and therefore does not have event or data interfaces in the conventional sense of a function block. Instead, it provides named constants that can be used in other blocks.

Event Inputs

Not present. This definition does not contain any event inputs.

Event Outputs

Not present. This definition does not contain any event outputs.

Data Inputs

Not present. This definition does not contain any data inputs.

Data Outputs

Not present. This definition does not contain any data outputs.

Adapters

Not present. This definition does not contain any adapter interfaces.

Functionality

The `HANDLE` constant defines two symbolic names for special integer values:

1. `UNVALID`: Represents an invalid or unused handle with the value `-1` (decimal) or `0xFFFF` (hexadecimal). This value can be used to indicate that a handle is not initialized, an operation has failed, or a resource is unavailable.
2. `GLOBAL`: Represents a pseudo-handle with the value `32767` (decimal) or `0x7FFF` (hexadecimal). This value is used to address a global destination (broadcast) within the ISOBUS network, rather than a specific individual device.

These constants are typically used in the data connections or internal logic of other functional blocks (e.g., for ISOBUS communication) to improve code readability and maintainability, rather than using magic numbers directly in the code.

Technical Details

- **Packet Assignment:** The constants are assigned to the `isobus::pgn` package, which clearly defines their use in ISOBUS PGN communication.
- **Typing:** Both constants are of type `INT` (Integer).
- **Initialization:** The values are assigned directly upon declaration (`InitialValue`).
- **Visibility:** As global constants (`GLOBALCONSTANTS`), they are visible and usable within the defined package (`isobus::pgn`) for all function blocks contained within it.

State Overview

Since this is a static constant definition, there is no state machine or mutable state.

Application Scenarios

- **Initializing Handles:** When a system starts, communication handles can be initialized with the value `UNVALID` to indicate their "empty" state.
- **Error Handling:** Function blocks that return handles (e.g., after opening a connection) can return `UNVALID` to indicate an error.
- **Broadcast Communication:** A sender block can use the `GLOBAL` handle to address a message to all participants in the ISOBUS network instead of a specific receiver.
- **Comparisons:** The readability of conditions such as `IF currentHandle = UNVALID THEN ...` or `IF targetHandle = GLOBAL THEN ...` is significantly improved.

⚖️ Comparison with Similar Blocks

- **`CONSTANT` Function Block:** A generic `CONSTANT` block can output a fixed value but is an instance within a function block network. The `HANDLE` global constant is a central, package-wide definition that does not need to be instantiated and is referenced directly by its name.
- **Direct Value Input:** Instead of using `UNVALID` or `GLOBAL`, the numeric values (`-1`, `32767`) could be entered directly into the blocks. However, using the constants is significantly better practice, as it makes the code more self-explanatory and resistant to changes (changes only need to be made in one place).

Conclusion

The `HANDLE` global constant is a simple yet essential tool for developing ISOBUS-compliant applications in 4diac. It improves code quality by assigning meaningful names to two frequently used, special handle values. Using this promotes readable, maintainable, and fault-tolerant code in function blocks that use ISOBUS PGN addressing.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ISONETEVENT_T

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `ISONETEVENT_T` is a structured type used to describe a network event within an ISOBUS network. It encapsulates all relevant information needed to identify and process an event related to the management of network participants (control functions). This type is typically used in callback functions to report changes in the network's state.

Interface Structure

The structure `ISONETEVENT_T` consists of the following six elements:

Data Elements

- **`eNetEvent`** (Type: `USINT`, Initial value: `ISO_NETEVENT_e::Isonet_Last`)
- **Description:** Indicates which specific event occurred in the network member management (e.g., connection establishment, termination, address change).
- **`eMemberRefer`** (Type: `SINT`, Initial value: `ISOUSERHOME_e::notdef`)
- **Description:** Identifies which network member is affected by the event. Specifies whether it is the callback owner, another internal, or an external Control Function (CF).
- **`s16OwnCfHnd`** (Type: `INT`)

Description: The CF handle of the entry for which the callback was triggered. This identifies the local representation of the affected network participant.

- **`s16Handle`** (Type: `INT`)

Description: The network member handle of the involved control function or user. This is the handle of the CF that caused the event (Event Source CF Handle).

- **`u8SAMember`** (Type: `USINT`)

Description: The current source address (SA) of the network member at the time of the event.

- – **cfName** (Type: `isobus::pgn::CF_NAME_T`)

Description: The ISOBUS name of the involved control function. This name is a unique identifier according to the ISOBUS standard.

Functionality

The structure `ISONETEVENT_T` functions as a container or message object. When a relevant network event occurs (defined by `eNetEvent`), an instance of this structure is populated with the current data and passed to a registered callback function. The contained data enables the receiving software component to accurately classify the event and identify the affected network entities (`eMemberRefer`, `s16OwnCfHnd`, `s16Handle`), their current address (`u8SAMember`), and their ISOBUS name (`cfName`).

Technical Details

- **Type:** This is a `StructuredType` within the 4diac IDE.
- **Initial Values:** Two elements (`eNetEvent`, `eMemberRefer`) have defined initial values that reference enumerated types (`ISO_NETEVENT_e`, `ISOUSERHOME_e`). This ensures a defined initial state.
- **ISOBUS Integration:** The structure is specifically designed for the ISOBUS context, as evidenced by the use of the type-specific `CF_NAME_T` and the parameter designations (SA, CF).
- **Compiler Package:** The data type is assigned to the package `isobus::pgn`, indicating its association with ISOBUS Parameter Group Number (PGN) processing.
-

Application Scenarios

This data type is primarily used in scenarios where an application needs to be informed about dynamic changes in the ISOBUS network. Examples include:

- Monitoring the appearance or disappearance of electronic control units (ECUs) on the network.
- Responding to address changes of network participants.
- Implementing user-defined logic when establishing or terminating a connection to a specific CF.
- Logging network activity for diagnostic purposes.

⚖️ Comparison with Similar Building Blocks

Within the 4diac type library, various structured data types exist for communication protocols. Unlike generic communication structures (such as those for TCP/IP), `ISONETEVENT_T` is highly specialized for the event messages of the ISOBUS protocol stack, particularly the network management (NM) component. Other ISOBUS-related types might focus more on the exchange of process data (PGN data).

Conclusion

The data type `ISONETEVENT_T` is an essential tool for developing ISOBUS-enabled applications in 4diac that need to react to network events. It provides a standardized, information-rich interface for receiving and processing callback-based event notifications from the ISOBUS network stack. Its use promotes a decoupled and event-driven architecture in the control software.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

ISOUSERHOME_e



- – ◦ • – ◦ • – ◦ •

Introduction

The function block `ISOUSERHOME_e` is a **constant definition** (GlobalConstants) and not an executable function block in the traditional sense. It defines an enumeration of integer constants used to describe the network origin or destination of an event or message in an ISOBUS context (ISO 11783). These constants

are typically used in other function blocks to identify the source or receiver of data within a controller network.

Interface Structure

Because it is a constant definition, `ISOUSERHOME_e` does not have event- or dataflow-based interfaces like a standard function block. Instead, it provides named constants.

Event Inputs

Not present.

Event Outputs

Not present.

Data Inputs

Not present.

Data Outputs

Not present.

Adapters

Not present.

Functionality

This block defines four symbolic constants of type `SINT` (short integer):

- `intern` (value: 0): Identifies an internal control function (CF) or an internal user within a device or logical unit.
- `network` (value: 1): Identifies an external CF or an external user on the network.
- `thisMember` (value: 2): Specifies that an event or message is intended for this specific internal CF or user.
- `notdef` (value: −1): Indicates that the value is invalid or undefined.

These constants can be imported and used throughout the application to make code more readable and maintainable, instead of directly using magic numbers (such as 0, 1, 2, −1).

Technical Details

- **Type:** `GlobalConstants` according to IEC 61499-1.
- **Data Range:** The constants are of type `SINT` (8-bit signed integer).
- **Package:** The block is part of the `isobus::pgn` package, indicating its use in the context of ISOBUS Parameter Group Numbers (PGNs).

State Overview

Not applicable, as this is a static constant definition without state machines or behavior.

Application Scenarios

These constants are primarily used in ISOBUS-compliant control systems in agricultural machinery, for example, in tractors and their implements. Typical applications include:

- **Message Filtering:** A receiver block can use the `ISOUSERHOME_e` value to determine whether an incoming ISOBUS message is intended for its own device (`thisMember`), another device on the network (`network`), or for internal processing (`intern`).
- **Message Construction:** A sender block can set this value to identify the intended recipient of a sent message.
- **Status Feedback:** A block can return the status `notdef` if the origin or destination of a message cannot be determined.
- **Message Construction:**

⚖️ Comparison with Similar Building Blocks

- **`E_SR` or `E_RESTART`:** These are executable basic function blocks with event logic. `ISOUSERHOME_e`, on the other hand, is a pure data definition without logic.

- **User-Defined Data Types (DT):** Similar to a structured data type (STRUCT) or an enumeration (ENUM), `ISOUSERHOME_e` is used to encode semantic meaning. The difference lies in the IEC 61499 specification: `GlobalConstants` are intended for immutable values that are valid project-wide.
- **Direct Use of Numeric Values:** Using `ISOUSERHOME_e.intern` in the code is significantly more meaningful and less error-prone than directly using the number `0`.

Conclusion

ISOUSERHOME_e` is an essential auxiliary construct for developing clear and maintainable ISOBUS applications in 4diac. By providing meaningful constant names, it abstracts from the underlying numeric values, thus improving the readability and robustness of the system design. It is a data-level component that supports the communication logic in higher-level function blocks.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ISO_CAN_NODE



• – ◦ • – ◦ • – ◦ •

Introduction

The global constant file `ISO_CAN_NODE` defines a set of numeric constants for identifying CAN (Controller Area Network) nodes within a 4diac project. It serves to abstract and standardize node addresses, thereby improving the readability and maintainability of applications that operate with multiple CAN buses.

Interface Structure

This element is a collection of global constants and does not have event- or data-flow-driven interfaces like a function block.

Data Constants

The file defines the following constants of type `USINT` (Unsigned Short Integer):

Name	Type	Initial Value	Comment
`NODE1`	USINT	0	Identifier for CAN 1
`NODE2`	USINT	1	Identifier for CAN 2
`NODE3`	USINT	2	Identifier for CAN 3
`NODE4`	USINT	3	Identifier for CAN 4
`INVALID`	USINT	USINT#16#FF	Identifier for an invalid or unconfigured CAN node (decimal: 255)

Functionality

The constants are inserted into the code at compile time. Instead of using direct numerical values (such as `0` or `1`) in function blocks that control CAN communication, developers can use descriptive names (`NODE1`, `NODE2`, etc.). This makes the code more intuitive and facilitates future modifications, as the actual addresses are managed centrally in one location.

The value `INVALID` (255) represents a reserved value, typically used to indicate an uninitialized, faulty, or non-existent node.

Technical Details

- **Packet Mapping:** The constants are mapped to the package `isobus::pgn`, indicating use in the context of ISOBUS (ISOBUS, agricultural machinery) and Parameter Group Numbers (PGN). * **Data Type:** All

constants are of type `USINT`, which allows a value range from 0 to 255 and is common for CAN node addresses.
- **Hexadecimal Notation:** The value for `INVALID` is defined in hexadecimal notation (`16#FF`), which corresponds to the decimal number 255.

State Overview

Since these are static constants, no state machine exists.

Application Scenarios

- **Multi-CAN Systems:** In controllers that communicate with multiple independent CAN buses (e.g., a main bus and a drive bus) to uniquely address the buses.
- **ISOBUS Applications:** For configuring virtual terminals (VTs), task controllers (TCs), or other devices in an agricultural ISOBUS network.
- **Error Handling:** The constant `INVALID` can be used as a return value or initial value in function blocks to indicate that no valid CAN node has been selected or detected.

⚖️ Comparison with Similar Function Blocks

- **Direct Values:** The alternative approach would be the direct use of numeric literals (0, 1, 2, ...) in the code. Using `ISO_CAN_NODE` is superior to this approach in terms of code quality and maintainability.
- **Enumerations (ENUM):** In some programming languages or function block types, similar constructs could be defined as enumerations. Global constants are a simple and widely used means of implementing this pattern in IEC 61499.

Conclusion

The `ISO_CAN_NODE` constant file is a simple yet valuable tool for the structured development of CAN-based control applications in 4diac. It promotes clean, self-explanatory code and centralizes configuration data, reducing errors and simplifying adjustments. Its use is particularly recommended in projects with multiple CAN nodes or in ISOBUS environments.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ISO_NETEVENT_e



• – ◦ • – ◦ • – ◦ •

Introduction

The global constant file `ISO_NETEVENT_e` defines an enumeration of event types for network management in an ISOBUS-compatible system. It provides named, integer constants that represent various state changes and events in the network, such as adding or removing devices or address conflicts. Using these constants improves the readability and maintainability of the source code.

Interface Structure

ISO_NETEVENT_e is a **global constant file** and not a function block. Therefore, it does not have event or data interfaces in the conventional sense. Instead, it provides a collection of constants (VAR_GLOBAL CONSTANT`) that can be used system-wide.

Data Inputs

Not applicable. This resource defines constants, not inputs.

Data Outputs

Not applicable. This resource defines constants, not outputs.

Event Inputs

Not applicable.

Event Outputs

Not applicable.

Adapters

Not applicable.

Functionality

This file declares a set of constants of type `USINT` (Unsigned Short Integer, 8-bit). Each constant is assigned a unique numeric value and a descriptive name. These constants can be imported and used in other function blocks or programs to identify network events or query states, instead of using magic numbers in the code.

Technical Details

- **Type:** All constants are of type `USINT`.
- **Initialization:** Each constant is initialized with a unique value (0 to 9) upon declaration.
- **Conditional Compilation:** The constants `Isonet_HsiActive` and `Isonet_HsiInactive` are marked as conditional constants (originally from C code). Their use may depend on the definition of a preprocessor symbol (`CCI_HSI`).
- **Package Assignment:** The constants are assigned to the package `isobus::pgn`, indicating their use in the context of ISOBUS protocol handling (Parameter Group Number).

State Overview

Not applicable. Constants have no state.

Application Scenarios

These constants are typically used in function blocks that:

1. Implement ISOBUS network management (address claiming, network management). 2. Monitor and respond to network events (e.g., address conflicts).
2. Manage the status of other network participants (controller applications, ECUs).
3. Provide diagnostic or logging functions that record network events.

⚖️ Comparison with similar building blocks

Unlike function blocks (FBs) or basic function blocks (BFBs), a global constant file does not contain any executable logic or algorithms. It is comparable to a header file in C/C++ or a `ENUM` in higher-level programming languages. Its sole purpose is to define immutable values for the entire application.

Conclusion

The `ISO_NETEVENT_e` is an essential resource for the structured development of ISOBUS applications in the 4diac IDE. It standardizes the coding of network management events, promotes code clarity, and reduces errors that can arise from the use of undifferentiated numerical values. Its integration is particularly beneficial for complex, network-based control systems in the agricultural and mobile machinery sectors.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

NAMEFIELD_T

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `NAMEFIELD_T` represents a structured data component for mapping the ISO 11783 (ISOBUS) NAME field according to Part 5 of the standard and the Implemented Function Module (IFM) conventions. This structured data type serves for the unique identification and addressing of electronic control units

(ECUs) in an ISOBUS network. It combines all parameters required for device identification into a single data structure.

Interface Structure

NAMEFIELD_T` is a structured data type (STRUCT) consisting of ten individual variables. Each variable corresponds to a defined part of the 64-bit NAME field in the ISOBUS protocol.

Data Inputs

(As a structured data type, `NAMEFIELD_T` has no events, only data fields. These are listed here as components of the structure.)

Name	Data Type	Initial Value	Comment
`bSelfConf`	`BOOL`	`TRUE`	Indicates whether it is a self-configurable address ECU.
`bIndGroup`	`BYTE`	`16#FF`	Assigns the NAME to an industry group (range 0-7).
`bDevClassInst`	`BYTE`	`16#FF`	Identifier if a device class occurs multiple times on the network (0..15).
bDevClass`	`BYTE`	`16#FF`	ISO-defined device class (0..127).
bReserved`	`BOOL`	`TRUE`	Reserved bit (not used).
bFunction`	`BYTE`	`16#FF`	ISO-defined function, corresponds to the Source Address (SA) (0..255).
bFunctionInst`	`BYTE`	`16#FF`	Subgroup of a function (0..31).
bEcuInstance`	`BYTE`	`16#FF`	Indicates which control unit is assigned to a specific function (0..7).
wManufCode`	`WORD`	`16#FFFF`	ISO-defined manufacturer code (0..2047).
dwIdentNumb`	`DWORD`	`16#FFFFFFFF`	Unique serial number; used when the NAME is not unique within the same network (0 to 2,097,151).

Data Outputs

(Not applicable, as this is a data-only type.)

Adapters

(Not applicable, as this is a data-only type.)

Functionality

The `NAMEFIELD_T` data type is a passive data structure. Its primary function is the structured storage and transmission of the parameters that together form the 64-bit ISOBUS NAME. Each variable within the structure occupies a specific bit range within this overall field. The initial values (`16#FF`, `16#FFFF`, `16#FFFFFFFF`) typically represent an undefined or unassigned state.

Technical Features

- **Bit-accurate mapping:** The structure directly maps the binary layout of the ISO NAME field, enabling efficient serialization and deserialization for communication via the CAN bus protocol.
- **Initial values:** The selected initial values (maximum values for the respective data types) clearly indicate an uninitialized or placeholder state.
- **ISOBUS compliance:** The naming and division of the fields strictly adhere to the ISO 11783-5 specification and the extended IFM conventions.

State overview

Since this is a data type and not a function block, there is no state machine. The “state” is defined by the specific values of the ten included variables.

Application Scenarios

The data type `NAMEFIELD_T` is primarily used in ISOBUS-compliant control applications, for example, in agricultural and forestry technology or construction machinery.

- **Device Identification:** As part of messages (e.g., in the Address Claim process) to identify a participant in the network.
- **Data Storage:** For storing the device's own identity within ECU software.
- **Filtering:** For configuring message filters based on device class, function, or manufacturer.
- **Diagnostics:** For uniquely identifying devices in diagnostic and configuration tools.
-

⚖️ Comparison with Similar Function Blocks

Unlike function blocks (FBs) such as `E_SR` or `E_CYCLE`, `NAMEFIELD_T` is a pure **data type** (structured data type). It contains no algorithms, events, or executed code.

- **Similarity to other STRUCTs:** It is comparable to other structured data types in 4diac (e.g., for complex measured values), but is specifically designed for the ISOBUS protocol.
- **Difference to Basic Data Types:** While basic types such as `INT` or `BOOL` hold individual values, `NAMEFIELD_T` groups several logically related values into a single unit.
-

Conclusion

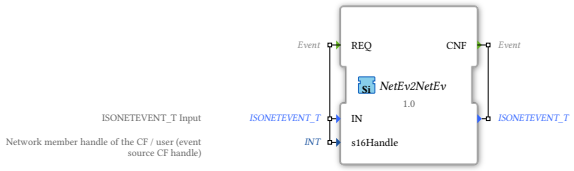
The data type `NAMEFIELD_T` is an essential tool for developing ISOBUS-compliant control applications with 4diac FORTE. It abstracts the complex bit structure of the ISO NAME field into an easily manageable, strongly typed data structure. This promotes code reusability, improves readability, and reduces errors when handling device identification data. Its use is standards-compliant and recommended for all projects implementing ISOBUS communication.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

NetEv2NetEv



• – ○ • – ○ • – ○ •

Introduction

The function block `NetEv2NetEv` is used to manipulate network events (NetEv) within an ISOBUS-based system. Its primary purpose is to take an incoming network event record and overwrite a specific value—the network member handle. The block thus functions as a simple converter or adapter for ISONETEVENT_T data structures.

Interface Structure

Event Inputs

- **REQ**: This event triggers the processing of the function block. It is linked to the data inputs `IN` and `s16Handle`.

Event Outputs

- **CNF**: This event signals the completion of processing. It is triggered after the processing of the `REQ` event and is linked to the (empty) data output that returns the manipulated `ISONETEVENT_T` structure.

Data Inputs

- **IN** (`isobus::pgn::ISONETEVENT_T`): The input for the original network event data structure to be manipulated.
- **s16Handle** (`INT`): The new network member handle to be written to the passed `ISONETEVENT_T` structure. This value identifies the controller/function (CF) or user that serves as the source of the event.

Data Outputs

- **<leer>** (`isobus::pgn::ISONETEVENT_T`): The output returns the manipulated `ISONETEVENT_T` data structure. The output name is empty, and the data type is identical to that of the input `IN`.

Adapters

This function block does not use any adapters.

Operation

The operation of the `NetEv2NetEv` block is deterministic and simple:

1. The function is executed when a `REQ` event occurs.
2. The entire contents of the input data structure `IN` are copied to the output data structure.
3. The entire contents of the input data structure `IN` are copied to the output data structure. 3. The field `.s16Handle` within this copied output structure is then overwritten with the value of the input `s16Handle`.
4. The `CNF` event is triggered to signal the output of the modified data set.

This block only modifies the specified handle field. All other fields of the `ISONETEVENT_T` structure remain unchanged.

Technical Features

- **Data Type**: This block specifically works with the data type `isobus::pgn::ISONETEVENT_T`. The exact structure of this type is defined in the underlying `isobus::pgn` library.
- **Direct Assignment**: The implementation uses a direct assignment in Structured Text (ST). No additional logic, validation, or error handling is performed.
- **Naming Convention**: The data output has no explicit name, which is permitted in 4diac. It is referenced via the `CNF` event.

State Overview

The function block has no internal state or memory. Its behavior is purely combinatorial: On every `REQ` event, the described operation is immediately performed, and a `CNF` event is generated.

Application Scenarios

- **Handle Override in Event Streams**: In networks where events are forwarded between different controllers (CFs), the source handle (`s16Handle`) may need to be adjusted to correctly identify the current sender.
- **Event Proxy/Adapter**: This block can be used as a simple proxy to assign a new source identifier to incoming network events before forwarding them to downstream processing logic.
- **Testing and Simulation**: For simulating network events from various virtual sources (CFs) within a test setup.

⚖️ Comparison with Similar Blocks

- **E_SR (SR Flip-Flop) or E_D_FF (D Flip-Flop)**: These blocks have internal state (memory). `NetEv2NetEv` is stateless and only manipulates incoming data.
- **E_SELECT / E_MUX**: These blocks select between different data sources or pass them through. `NetEv2NetEv` actively modifies a portion of the incoming data.
- **E_R_TRIG / E_F_TRIG**: These are pure event triggers without data manipulation. `NetEv2NetEv` is a data processing block.
- **Generic ASSIGN Blocks**: A simple `ASSIGN` block copies an input value 1:1 to the output. `NetEv2NetEv` is a specialized `ASSIGN` block for the type `ISONETEVENT_T`, which performs a targeted modification (handle override).

🛠️ Related Exercises

- Exercise_128
- Exercise_128b
- Exercise_134

Conclusion

The `NetEv2NetEv` function block is a specialized, lightweight tool for ISOBUS applications. It fulfills a clear and necessary niche task: overriding the source handle in network event records. Its simple, stateless implementation makes it robust and predictable. For developers working with ISOBUS PGNs and network events, it provides a useful tool for correctly identifying and customizing data flows.

🌐 Related topic subpages on ms-muc-docs.de

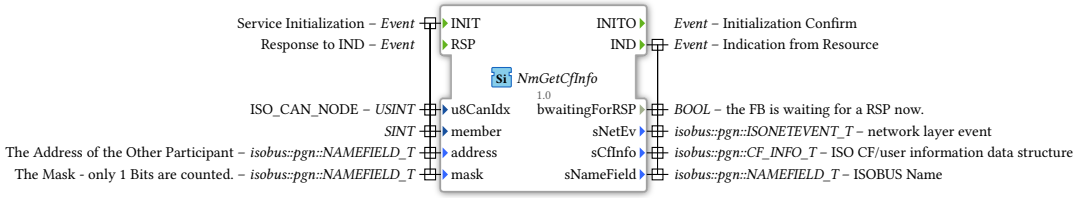
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

NmGetCfInfo

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `NmGetCfInfo` is used within an ISOBUS network to retrieve information about connected communication partners (Connected CFs). It allows both internal and external participants in the network to be identified and their details to be recorded. This block is part of ISOBUS PGN communication and is typically used for network management tasks.



Interface Structure

Event Inputs

- **INIT (Type: EInit)**: Initializes the service. When triggered, the configured data inputs are accepted and the initialization process is started.
- **RSP (Type: Event)**: Acknowledges receipt of an indication (`IND`). This event signals to the module that a previously sent indication has been processed.

Event Outputs

- **INITO (Type: EInit)**: Confirms successful completion of initialization.
- **IND (Type: Event)**: Triggered to forward a received network indication (e.g., detected communication partners) to the application.

Data Inputs

- **u8CanIdx (Type: USINT, Initial Value: ISO_CAN_NODE::INVALID)**: Identifies the CAN node used (CAN controller index).
- **member (Type: SINT, Initial Value: ISOUSERHOME_e::notdef)**: Defines the membership or role of the requesting participant in the network.
- **address (Type: isobus::pgn::NAMEFIELD_T)**: The address of the other network participant for whom information is to be retrieved.
- **mask (Type: isobus::pgn::NAMEFIELD_T)**: A bitmask that specifies which parts of the address should be considered during the search. Only set bits (1) are evaluated.

Data Outputs

- **bwaitingForRSP (Type: BOOL)**: Indicates whether the function block is currently waiting for a response (`RSP`) to a sent indication.
- **sNetEv (Type: isobus::pgn::ISONETEVENT_T)**: Contains details of the network event that occurred (e.g., error or status).

- **sCfInfo (Type: isobus::pgn::CF_INFO_T)**: The retrieved information about the found communication partner (CF = Communication Function).
- **sNameField (Type: isobus::pgn::NAMEFIELD_T)**: The ISOBUS name of the detected communication partner.

Adapter

This function block does not use any adapter interfaces.

Functionality

1. **Initialization**: The block is configured by triggering the `INIT` event. The passed parameters (`u8CanIdx`, `member`, `address`, `mask`) define the context and target of the query. After successful initialization, `INITO` is output.
2. **Query and Indication**: After initialization, the module begins searching for suitable communication partners in the network based on the `address` and `mask` events. If a participant is found, the module triggers the `IND` event and makes the found information available in the data outputs `sCfInfo` and `sNameField`. Simultaneously, `bwaitingForRSP` is set to `TRUE`.
3. **Response Processing**: The higher-level application acknowledges receipt of the indication by sending the `RSP` event. The function block then reverts from `bwaitingForRSP` to `FALSE`. This handshake mechanism ensures secure transmission.

Technical Features

- The function block implements a request-response protocol (`IND`/`RSP`) for reliable communication.
- The use of a bit-based `mask` enables flexible search queries, e.g., for participants of a specific device class.
- The data types (`ISONETEVENT_T`, `CF_INFO_T`, `NAMEFIELD_T`) are specific to ISOBUS PGN communication and contain structured information according to the ISO 11783 standard.
-

Status Overview

1. **Inactive**: Before initialization.
2. **Initialized**: After `INITO`. The function block is ready for operation and can process requests.
3. **Waiting for RSP**: After `IND` is triggered. The function block waits for confirmation (`RSP`) from the application before performing further actions.

Application Scenarios

- **Network Discovery**: When a device starts up, to identify all active participants in the ISOBUS network.
- **Diagnostics and Monitoring**: To monitor whether specific expected control units (e.g., of an implement) are connected and reachable.
- **Dynamic Configuration**: To obtain information about newly added devices and adapt the application accordingly.

⚖️ Comparison with Similar Modules

Unlike simple read or query modules, `NmGetCfInfo` is specifically designed for querying ISOBUS-specific network information. It offers more context (through `member`, `address`, `mask`) and structured result data (`CF_INFO_T`) than a generic communication module. Modules like `E_SWITCH` or `E_DEMUX` only forward events, while `NmGetCfInfo` performs active network communication and protocol handling.

Modules like `E_SWITCH` or `E_DEMUX` only forward events, while `NmGetCfInfo` performs active network communication and protocol execution.

🛠️ Related exercises

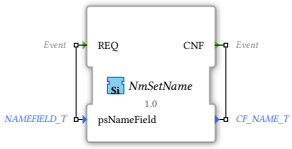
- Uebung_120
- Uebung_121
- Uebung_122
- Uebung_122b
- Uebung_123
- Uebung_124
- Uebung_125
- Uebung_126
- Uebung_126b
- Uebung_126b2

- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_130
- Uebung_131
- Uebung_132
- Uebung_133
- Uebung_134

Conclusion

The `NmGetCfInfo` function block is a An essential tool for managing ISOBUS networks, it abstracts the complex protocol communication required to detect network participants into an easy-to-use, event-driven interface. Its ability to search for participants and return detailed information makes it valuable for robust and intelligent agricultural control systems.

NmSetName



• — ○ • — ○ • — ○ •

Introduction

The function block `NmSetName` converts a structured name field definition into a compact, 8-byte ISO NAME according to the ISO 11783-5 specification (section 5.1.2). This block is an essential component for device identification and addressing in ISOBUS networks (agricultural and forestry machinery). It performs the bit-accurate mapping of the individual fields of a `NAMEFIELD_T` structure to the corresponding byte array of a `CF_NAME_T`.

Interface Structure

Event Inputs

- **REQ**: This event triggers the conversion. Upon arrival, the incoming data is processed at input `psNameField`.

Event Outputs

- **CNF**: This event signals the completion of processing. It is generated after the successful conversion of the input data.

Data Inputs

- **psNameField** (`isobus::pgn::NAMEFIELD_T`): The input structure containing all individual components of an ISOBUS name (such as manufacturer code, device class, function instance, etc.) in a format easily manageable for the programmer.

Data Outputs

- **(Unnamed)** (`isobus::pgn::CF_NAME_T`): This output provides the resulting 8-byte ISO NAME array (`data[0]` to `data[7]`), which can be used directly in ISOBUS messages.

Adapters

This function block does not use any adapters.

Functionality

The block operates as a pure data transformer. When triggered by the `REQ` event, it reads the values from the input structure `psNameField` and distributes them bitwise across the eight bytes of the output array according to the ISO standard. The logic consists of a sequential sequence of bitwise operations (AND, OR, SHL) and type conversions. Each line in the ST code sets specific bits in a particular byte of the `CF_NAME_T`

output, while the remaining bits remain unchanged (masked by the `AND` operations). After all bytes are set, the `CNF` event is output.

Technical Features

- **Bitwise Processing:** The core functionality is based on precise bitwise operations to pack the complex structure of the ISO NAME into a compact byte array.
- **Type Safety:** Explicit type conversions (e.g., `DWORD_TO_BYTE`, `WORD_TO_BYTE`) are used to ensure correct data handling.
- **Standard Compliance:** The implementation strictly adheres to the bit mapping as defined in ISO 11783-5.
- **Stateless:** The block has no internal state between calls. The output depends solely on the current input data.

State Overview

The function block is stateless. Its behavior can be described as a simple, deterministic function:

1. **Waiting:** The block waits for the `REQ` event.
2. **Processing:** At `REQ`, the input data is processed and the output array is calculated.
3. **Confirming:** Immediately after the calculation, the `CNF` event is output.

The block then returns directly to the wait state (1).

Application Scenarios

- **Initializing an ISOBUS Node:** When an electronic control unit (ECU) starts up in an ISOBUS network, its unique NAME must be constructed from configurable parameters (e.g., from an NVRAM) and entered into the communication stack.
- **Dynamic Device Configuration:** In systems where device properties (e.g., function instance) can change at runtime, this block is used to generate the new, valid NAME.
- **Test and Simulation Tools:** For generating correct ISO NAMEs for simulating various virtual devices in an ISOBUS network.

⚖️ Comparison with Similar Blocks

- **Counterpart `NmGetName`:** While `NmSetName` packs a structured description into a byte array, a hypothetical `NmGetName` block would perform the reverse operation: It extracts the individual fields from a given ISO NAME byte array and presents them in a `NAMEFIELD_T` structure. `NmSetName` is thus the "encoder," and its counterpart would be the "decoder."
- **Generic Byte Packers:** Unlike generic function blocks that serialize arbitrary data structures, `NmSetName` is specifically optimized for the ISO 11783 NAME structure, thus guaranteeing compliance with the standard without additional configuration.

🛠️ Related Exercises

- Exercise_121

Conclusion

The `NmSetName` function block is a specialized, indispensable component for ISOBUS communication. It abstracts the complex, bit-accurate composition of the ISO NAME according to the standard and provides the user with a simple, structured interface. Its deterministic and stateless operation makes it robust and easy to integrate into control sequences for device initialization and configuration.

NmSetNameField



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `NmSetNameField` converts an ISO NAME, as defined in ISO 11783-5 (section 5.1.2), into a structured representation. An ISO NAME is an 8-byte identifier used in agricultural and forestry machinery (ISOBUS) for the unique identification of electronic control units (ECUs). This function block extracts the individual fields from the byte array and populates a corresponding data structure (`NAMEFIELD_T`). This facilitates further processing and evaluation of the device information in an IEC 61499 application.

Interface Structure

Event Inputs

- **REQ**: Starts processing. Upon receiving this event, the value at `au8IsoName` is processed.

Event Outputs

- **CNF**: Triggered as soon as the conversion is complete and the structured data is available at the output.

Data Inputs

- **au8IsoName** (`isobus::pgn::CF_NAME_T`): An 8-byte array containing the raw ISO NAME according to ISO 11783-5.

Data Outputs

- **(no explicit name)** (`isobus::pgn::NAMEFIELD_T`): The output variable contains the structured breakdown of the ISO NAME. The exact structure of the `NAMEFIELD_T` structure is defined by the `isobus::pgn` library used.

Adapters

This function block does not use any adapters.

Functionality

When the `REQ` event is triggered, the function block processes the incoming byte array `au8IsoName`. The Structured Text (ST) implementation extracts the individual components of the ISO NAME bit- and byte-wise and assigns them to the corresponding fields of the `NAMEFIELD_T` output structure. The logic follows the bit mapping exactly as defined in ISO 11783-5, section 5.1.2. After the calculations are complete, the `CNF` event is set to signal the successful completion of the operation.

Technical Features

- **Bit Manipulation:** This block uses logical AND operations (`AND`), shift operations (`SHL`, `SHR`), and type conversions (`BYTE_TO_WORD`, `BYTE_TO_DWORD`) to isolate the relevant bits from the input array.
- **Library Dependency:** This block depends on the data types `CF_NAME_T` and `NAMEFIELD_T` from the `isobus::pgn` library. The exact definition of these types (e.g., the field names in `NAMEFIELD_T`) is outside this block.
- **Library Dependency:** This block depends on the data types `CF_NAME_T` and `NAMEFIELD_T` from the `isobus::pgn` library. The precise definition of these types (e.g., the field names in `NAMEFIELD_T`) is outside this block.
- – **Deterministic:** The operation is purely data-driven and has no internal state. With the same input, it always produces the same output.

State Overview

The function block has no persistent internal state. It behaves like a pure function:

1. **Wait/Idle State:** The block waits for the `REQ` event.
2. **Processing State:** At `REQ`, the input data is processed and the output structure is calculated. This state is instantaneous (no delay).
3. **Output:** Immediately after the calculation, the `CNF` event is triggered and the calculated `NAMEFIELD_T` structure is output. The block then returns to the wait state.

Application Scenarios

- **ISOBUS Device Identification:** In an ISOBUS network manager that needs to decode incoming NAME data from connected ECUs to identify device type, manufacturer, and instance.
- **Diagnostics and Logging:** For clear display and logging of device names in a human-readable, structured format instead of the raw 8-byte value.
- **Filtering and Addressing:** As a preprocessing step to make decisions about further communication based on specific fields (such as device class or manufacturer code).
- **ISOBUS Device Identification:**

⚖️ Comparison with Similar Components

- **Counterpart `NmGetIsoName`:** A hypothetical complementary component would perform the reverse operation: It would take a `NAMEFIELD_T` structure as input and generate the corresponding 8-byte `CF_NAME_T` array. `NmSetNameField` is the decoder; its counterpart would be the encoder.
- **Generic Byte Parsers:** Unlike generic byte-to-structure converters, this block is specifically tailored to the ISO NAME format and contains the exact bit logic of the standard. Using this specialized block reduces errors and increases maintainability in the ISOBUS context.
-

🛠️ Related Exercises

- Exercise_120
- Exercise_121
- Exercise_122
- Exercise_122b

Conclusion

The `NmSetNameField` function block is an essential tool for working with ISOBUS networks in 4diac. It abstracts the complex bit manipulation required to interpret the ISO NAME and provides the information in an easily processed, structured format. Its use promotes correct, standards-compliant applications and significantly improves the readability of the source code.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

PgnDef



• – ◦ • – ◦ • – ◦ •

Introduction

PgnDef` is a collection of global constants for the 4diac IDE that defines Parameter Group Numbers (PGNs) for ISOBUS and NMEA 2000 communication protocols. These constants represent numeric identifiers for specific message types used in agricultural and maritime vehicle networking. The collection serves as a central reference library for accessing standardized PGNs in IEC 61499 applications, thus improving code readability and maintainability.

Interface Structure

PgnDefis `aGlobalConstants` element and therefore does not have event- or data-flow-based interfaces in the conventional sense of a function block. Instead, it provides a collection of constants that can be declared and used in other function blocks or programs.

Event Inputs

Not present. This is a constant definition.

Event Outputs

Data Inputs

Data Outputs

Adapters

How it Works

PgnDefitself has no executable logic. It simply defines an extensive list of constants of the data typeUDINT (Unsigned Double Integer). Each constant is a PGN value initialized in hexadecimal notation (UDINT#16#XXXX) and contains a human-readable comment with the PGN name and often a reference to the corresponding standard (e.g., ISO 11783-7). Other function blocks can import these constants via their fully qualified names (e.g., isobus::pgn::PGN_HEARTBEAT`) and use them in their logic to identify or generate messages.

Technical Features

- **Extensive Coverage:** The collection contains PGNs from a variety of standards, including ISO 11783 (ISOBUS, Parts 3-7, 12, 13), SAE J1939 (71, 73, 75), and NMEA 2000.
- **Organized Structure:** The constants are commented in logical groups (PDU1/Group Functions, PDU2, J1939-71, ISO 11783-7, Diagnostics (J1939-73), Generator Sets (J1939-75), NMEA).
- **Data Type:** All PGNs are defined as `UDINT` (32-bit, unsigned), which corresponds to the addressable range of the protocols.
- **Package Assignment:** The constants are assigned to the package `isobus::pgn`, which ensures clear namespace separation.

State Overview

Not applicable, as this is a static constant declaration.

Application Scenarios

- **ISOBUS ECU Development:** For developing electronic control units (ECUs) for tractors or implements that communicate via an ISOBUS network.
- **NMEA 2000 Integration:** In maritime applications for integrating ship electronics such as GPS receivers, depth sounders, or engine controls.
- **Protocol Implementation:** As a basis for implementing protocol stacks (e.g., for transport protocols such as TP/CM or ETP/CM).
- **Message Filtering:** In gateway or bridge function blocks to filter and forward incoming messages based on their PGN.
- **Diagnostic Tools:** In applications for diagnosing and monitoring vehicle networks, to correctly name and display received messages.

⚖️ Comparison with Similar Blocks

- **Simple `VAR_GLOBAL CONSTANT` Declarations:** `PgnDef` offers the advantages of centralization, reusability, and avoidance of redundancy and typos compared to constants declared manually in the function block.
- **Enumerations (ENUM):** While ENUMs in IEC 61499 can be used for ordered lists, `PgnDef` is better suited as a collection of discrete, not necessarily consecutive, values. Constants also offer more flexibility in value assignment (hexadecimal).
- **Specific Protocol Function Blocks:** Unlike function blocks that implement a specific protocol function (e.g., a `ADDRESS_CLAIM` function block), `PgnDef` is purely a reference tool and contains no active logic.

Conclusion

The `PgnDef` constant collection is an essential tool for developing IEC 61499-based applications in the domains of mobile machinery (ISOBUS) and marine electronics (NMEA 2000). It promotes standardization, improves code quality through enhanced readability, and reduces errors through the use of centrally defined and tested values. For any developer working with these protocols in 4diac, `PgnDef` is a valuable and likely indispensable resource.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

USERFUNC_e



• – ◦ • – ◦ • – ◦ •

The global constant `USERFUNC_e` defines an enumeration of numeric values that represent specific device functions or roles within an ISOBUS network. It serves for the standardized and readable identification of various ISOBUS device types and functions in control applications. USERFUNC_eis a collection of global constants and not a function block. Therefore, there are no event or data interfaces in the conventional sense. The structure consists exclusively of named constants of typeSINT` (short integer).

Not applicable. This resource provides constants and has no inputs.

Not applicable. The constants are referenced directly in the application code.

Data Outputs

Data Inputs

Interface Structure

Introduction

Adapters

Not applicable.

Functionality

USERFUNC_eis a collection of predefined constant values. Each name (e.g.,task_controller) is associated with a fixed numeric value (e.g., 1`). These constants can be used throughout the application to identify device roles, perform comparisons, or set parameters, significantly improving code maintainability and readability compared to directly using "magic numbers."

- **Data Type:** All constants are of type `SINT` (8-bit signed integer).
- **Range:** The defined values range from `-1` (`undefined`) to 21.
- **ISOBUS Compliance:** The designations and values are based on the device functions defined in the ISOBUS standard (ISO 11783).
- **Special Values:**
- `undefined` (-1): Indicates an undefined or unknown function.
- `all_clients` (20) & `all_server` (21): Values for broadcast or group addressing.

Not applicable. Constants have no state.

- **Device Identification:** In a task controller, `IF device_function = USERFUNC_e.virtual_terminal THEN ...` can be used to check whether a connection to a terminal exists.
- **Message Filtering:** Filtering of ISOBUS messages (PGNs) based on the source or destination device function.
- **Configuration:** Parameterization of network services that should only be available for specific device roles (e.g., `file_server`).
- **Diagnostics:** Readable output of device roles in diagnostic logs or on user interfaces.

Unlike function blocks that contain active logic, `USERFUNC_e` is a passive data type/constant definition. It is comparable to a `ENUM` data type in higher-level programming languages or a `CONSTANT` section in Structured Text (ST). Its primary purpose is to provide semantically meaningful names for numeric values used in the ISOBUS context.

The global constant `USERFUNC_e` is an essential tool for developing readable and maintainable ISOBUS applications in 4diac. It abstracts the raw, standard-defined numeric values for device functions into

meaningful identifiers, thus significantly contributing to error prevention and improved code quality. Its use is recommended for all projects that work with ISOBUS device communication.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Application Scenarios

State Overview

Technical Features

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

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

Const

Global_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Introduction

`Global_NAME_Functions` is a global constant group (GlobalConstants) used within 4diac-IDE applications that interface with ISOBUS (ISO 11783) / SAE J1939 networks. It provides a set of named, industry non-specific constants that represent the **Function** attribute of the **NAME** parameter group (J1939-21 / ISO 11783-7). Each byte value uniquely identifies the functional role of an electronic control unit (ECU) within a vehicle or machine network.

The constants act as a uniform vocabulary for classifying ECUs, allowing control logic to interpret NAME messages and to generate consistent, standards-compliant identifiers. The group is compiled into the `isobus::pgn::const` package and can be referenced throughout 4diac-IDE applications.

Interface Structure

This element is a **GlobalConstants** block, not a function block, adapter, or subapplication. It exposes no runtime inputs, outputs, events, or adapters. Instead, it provides a collection of global constants that applications can read symbolically.

Event Inputs

None – a global constant group has no event inputs.

Event Outputs

None – a global constant group has no event outputs.

Data Inputs

None – all entries are constants (`VAR_GLOBAL CONSTANT`) and are not modifiable at runtime.

Data Outputs

The following BYTE constants are defined. Values follow the SAE J1939 / ISO 11783 NAME function code assignment and may be used wherever an ECU function identifier is required.

Constant	Value	Description
`F_ENGINE`	0	Engine – typically the mechanical power source / engine management system.
`F_AUX_POWER_UNIT`	1	Auxiliary Power Unit (APU) – power source for operating systems without the prime drive engine.

F_ELEC_PROPULSION_CONTROL	2	Electric Propulsion Control – control system for electrically powered drive mechanisms.
F_TRANSMISSION	3	Transmission – alters speed vs. torque output of the engine.
F_BATTERY_PACK_MONITOR	4	Battery Pack Monitor – monitors charge, temperature, and power remaining.
F_SHIFT_CONTROL_CONSOLE	5	Shift Control/Console – determines and transmits desired gear, range, and operating mode.
F_POWER_TAKEOFF_MAIN_OR_REAR	6	Power Take-Off (Main/Rear) – controls mechanical power for auxiliary items.
F_AXLE_STEERING	7	Axle Steering – adjusts attack angle as a function of steering.
F_AXLE_DRIVE	8	Axle Drive.
F_BRAKES_SYSTEMCONTROL	9	Brakes System Controller – electronically controls the service braking system.
F_BRAKES_STEER_AXLE	10	Brakes Steer Axle – actuates service brakes on a steered axle.
F_BRAKES_DRIVE_AXLE	11	Brakes Drive Axle – actuates service brakes on a drive axle.
F_RETARDER_ENGINE	12	Retarder Engine – controls engine retarder capabilities.
F_RETARDER_DRIVELINE	13	Retarder Driveline – controls driveline retarder capabilities.
F_CRUISE_CONTROL	14	Cruise Control – maintains vehicle speed at a fixed operator-selectable value.
F_FUEL_SYSTEM	15	Fuel System – controls fuel flow from tank to engine and back.
F_STEERING_CONTROL	16	Steering Controller – controls steering in steer-by-wire systems.
F_SUSPENSION_STEER_AXLE	17	Suspension Steer Axle – control system for a steered axle suspension.
F_SUSPENSION_DRIVE_AXLE	18	Suspension Drive Axle – control system for a driven axle suspension.
F_INSTRUMENT_CLUSTER	19	Instrument Cluster – gauge display mounted in the cab.
F_TRIP_RECORDER	20	Trip Recorder – accumulates data versus travel.
F_CAB_CLIMATE_CONTROL	21	Cab Climate Control – controls climate within the cab.
F_AERODYNAMIC_CONTROL	22	Aerodynamic Control – modifies drag by altering body panels.
F_VEHICLE_NAVIGATION	23	Vehicle Navigation – physical location and driving instructions.

F_VEHICLE_SECURITY	24	Vehicle Security – compares operator-provided data sequences against references.
F_NETWORK_INTERCONNECT_UNIT	25	Network Interconnect ECU – connects different network segments (bridge/gateway).
F_BODY_CONTROLLER	26	Body Controller – handles suspension control for body sections and body components.
F_POWER_TAKEOFF_SECONDARY_OR_FRONT	27	Power Take-Off (Secondary/Front) – controls mechanical power for auxiliary items.
F_OFF_VEHICLE_GATEWAY	28	Off Vehicle Gateway – connects vehicle network(s) to off-vehicle systems.
F_VTERMINAL	29	Virtual Terminal (in cab) – general purpose ISOBUS display.
F_MANAGEMENT_COMPUTER	30	Management Computer – manages vehicle systems such as the powertrain.
F_PROPULSION_BATTERY_CHARGER	31	Propulsion Battery Charger – charges propulsion batteries from an off-board source.
F_HEADWAY_CONTROLLER	32	Headway Controller – forward-looking collision avoidance/warning and speed matching.
F_SYSTEM_MONITOR	33	System Monitor.
F_HYDRAULIC_PUMP_CONTROLLER	34	Hydraulic Pump Controller – provides hydraulic power for installed equipment.
F_SUSPENSION_SYSTEM_CONTROLLER	35	Suspension System Controller – coordinates overall vehicle suspension.
F_PNEUMATIC_SYSTEM_CONTROLLER	36	Pneumatic System Controller.
F_CAB_CONTROLLER	37	Cab Controller – groups cab-located vehicle functions.
F_TIRE_PRESSURE_CONTROL	38	Tire Pressure Control – centralised tire inflation.
F_IGNITION_CONTROL_MODULE	39	Ignition Control Module – alters engine ignition.
F_SEAT_CONTROL	40	Seat Control – controls operator/passenger seats.
F_LIGHTING_OPERATOR_CONTROLS	41	Lighting Operator Controls – transmits operator lighting control messages.
F_WATER_PUMP_CONTROL	42	Water Pump Control – controls a vehicle-mounted water pump.
F_TRANSMISSION_DISPLAY	43	Transmission Display – displays transmission information (e.g., gear).
F_EXHAUST_EMISSION_CONTROL	44	Exhaust Emission Control.
F_VEHICLE_DYNAMIC_STABILITY_CONTROL	45	Vehicle Dynamic Stability Control.
F_OIL_SENSOR_UNIT	46	Oil Sensor Unit.
F_INFORMATION_SYSTEM_CONTROLLER	47	Information System Controller – manages vehicle application information.

F_RAMP_CONTROL	48	Ramp Control – loading/unloading (chairlift, ramps, lifts, tailgates).
F_CLUTCH_CONVERTER_CONTROL	49	Clutch/Converter Control – handles torque converter lock-up.
F_AUXILIARY_HEATER	50	Auxiliary Heater – heating without the prime drive engine.
F_FORWARD_LOOKING_COLLISION_WARNING_SYSTEM	51	Forward-Looking Collision Warning System – detects and warns of impending collisions.
F_CHASSIS_CONTROLLER	52	Chassis Controller – controls chassis components.
F_ALTERNATOR_CHARGING_SYSTEM	53	Alternator/Charging System – primary on-board charging controller.
F_COMMUNICATIONS_UNIT_CELLULAR	54	Communications Unit, Cellular – communicates via the cellular telephone system.
F_COMMUNICATIONS_UNIT_SATELLITE	55	Communications Unit, Satellite – communicates via a satellite system.
F_COMMUNICATIONS_UNIT_RADIO	56	Communications Unit, Radio – communicates via terrestrial point-to-point.
F_AUXILIARY_DEVICE	57	Steering Column Unit – gathers operator inputs from steering column switches/levers.
F_FAN_DRIVE_CONTROL	58	Fan Drive Control – controls the main engine cooling fan.
F_STARTER	59	Starter – initiates rotation of a stopped engine.
F_CAB_DISPLAY	60	Cab Display – elaborate in-cab display (greater than 30 ASCII characters).
F_FILESERVER	61	File Server / Printer – printing or file storage unit on the network.
F_ON_BOARD_DIAGNOSTIC_UNIT	62	On-Board Diagnostic Unit – permanently mounted diagnostic tool.
F_ENGINE_VALVE_CONTROLLER	63	Engine Valve Controller – manipulates engine intake/exhaust valve actuation.
F_ENDURANCE_BRAKING	64	Endurance Braking – all devices enabling low-wear speed reduction on long descents.
F_GAS_FLOW_MEASUREMENT	65	Gas Flow Measurement – provides gas flow rate and associated parameter measurements.
F_I_O_CONTROLLER	66	I/O Controller – reports and/or controls external input/output channels.
F_ELECTRICAL_SYSTEM_CONTROLLER	67	Electrical System Controller – includes load centers, fuseboxes, power distribution.
F_AFTERTREATMENT_SYSTEM_GAS_MEASUREMENT	68	Aftertreatment System Gas Measurement – measures gas properties (e.g., NOx, oxygen).
F_ENGINE_EMISSION_AFTERTREATMENT_SYSTEM	69	Engine Emission Aftertreatment System.
F_AUXILIARY_REGENERATION_DEVICE	70	Auxiliary Regeneration Device – used as part of an aftertreatment system.

F_TRANSFER_CASE_CONTROL	71	Transfer Case Control – selects number of drive wheels (e.g., 2WD/4WD).
F_COOLANT_VALVE_CONTROLLER	72	Coolant Valve Controller – controls coolant flow for thermal management.
F_ROLLOVER_DETECTION_CONTROL	73	Rollover Detection Control – detects vehicle rollover.
F_LUBRICATION_SYSTEM	74	Lubrication System – pumps lubricant to machine/vehicle joints.
F_SUPPLEMENTAL_FAN	75	Supplemental Fan – additional cooling fan beyond the primary fan.
F_TEMPERATURE_SENSOR	76	Temperature Sensor – devices which measure temperature.
F_FUEL_PROPERTIES_SENSOR	77	Fuel Properties Sensor – measures fuel properties.
F_FIRE_SUPPRESSION_SYSTEM	78	Fire Suppression System.
F_POWER_SYSTEMS_MANAGER	79	Power Systems Manager – manages power output of one or more power systems.
F_ELECTRIC_POWERTRAIN	80	Electric Powertrain – controls and coordinates an electric drive system.
F_HYDRAULIC_POWERTRAIN	81	Hydraulic Powertrain – controls and coordinates a hydraulic drive system.
F_FILE_SERVER	82	File Server – file storage unit on the network.
F_PRINTER	83	Printer – printing unit on the network.
F_START_AID_DEVICE	84	Start Aid Device – controls hardware (glow plug, grid heater) assisting engine start.
F_ENGINE_INJECTION_CONTROL_MODULE	85	Engine Injection Control Module – direct or port injection of fuel.
F_EV_COMMUNICATION_CONTROLLER	86	EV Communication Controller – manages connection to an external power source (EVSE).
F_DRIVER_IMPAIRMENT_DEVICE	87	Driver Impairment Device – prevents vehicle start due to driver impairment (e.g., alcohol interlock).
F_ELECTRIC_POWER_CONVERTER	88	Electric Power Converter – inverter/converter transforming AC/DC power.
F_SUPPLY_EQUIPMENT_COMMUNICATION_CONTROLLER_SECC	89	Supply Equipment Communication Controller (SECC) – part of an EV charging station.
F_VEHICLE_ADAPTER_COMMUNICATION_CONTROLLER_VACC	90	Vehicle Adapter Communication Controller (VACC) – controller inside the adapter between EVSE connector and vehicle inlet.
F_ACCESSORY_ELECTRIC_MOTOR_CONTROLLER	91	Accessory Electric Motor Controller – operates and monitors electrified accessories.
F_CURRENT_SENSOR	92	Current Sensor – measures electrical current.

F_FUEL_CELL_SYSTEM	93	Fuel Cell System – converts fuel (e.g., hydrogen) to electricity.
F_AUXILIARY_DISPLAY	94	Auxiliary Display – display inside/outside the cab, separate from Cab Display.
F_NOT_AVAILABLE	255	Not-available function; placeholder for all undefined functions.

Adapters

None – a global constant group does not provide adapter interfaces.

Functionality

The `Global_NAME_Functions` constants represent the **Function** field of the NAME parameter group used in J1939-21 / ISO 11783-7. Each ECU on an ISOBUS network announces its role through the NAME, and the Function code disambiguates which of the many possible vehicle/machine subsystems the node implements.

For 4diac-IDE applications, these constants are used primarily in two ways:

- **Decoding** incoming NAME messages to identify the functional role of remote ECUs, enabling role-based routing, diagnostics, or coordination logic.
- **Encoding** a local ECU's Function code when constructing a NAME message, ensuring the node appears with the correct classification on the network.

The values follow the industry-standard, non-specific function list and are kept as plain `BYTE` constants so they can be directly embedded in parameter groups (e.g., within the NAME 8-byte data structure).

Technical Features

- **Data Type:** All constants are of type `BYTE` (8-bit unsigned integer).
- **Range:** Values 0 to 94 are assigned to specific functions; value 255 is used as the "not available" placeholder. Values 95 to 254 are not defined within this group.
- **Declaration:** Entries are declared as `VAR_GLOBAL CONSTANT`, making them read-only and globally accessible within the project.
- **Packaging:** The constants are compiled into the `isobus::pgn::const` package, allowing include/reference from other 4diac-IDE artifacts.
- **Compliance:** The mapping matches the industry non-specific function definitions of ISOBUS/J1939, which is referenced as the "other" category in the ISO 11783 standard (distinct from industry-specific groups, e.g., agricultural or construction).
- **Standard Reference:** The definitions align with the NAME Function assignment list published at isobus.net (function/name index).

State Overview

Not applicable. `Global_NAME_Functions` is a pure constant group and does not contain state machines, internal variables, or runtime-modifiable data. Its values are fixed at compile time and remain constant during application execution.

Application Scenarios

- **ECU Role Identification:** In an ISOBUS master/slave controller, read an incoming NAME message and compare its Function byte against constants such as `F_ENGINE` or `F_CRUISE_CONTROL` to determine which type of device is present and how to interact with it.
- **NAME Construction:** When building a NAME parameter group for a local node (e.g., a general-purpose display, a hydraulics controller, or an auxiliary power unit), assign the corresponding constant to the Function field for standards-compliant network participation.
- **Diagnostic & Monitoring Tools:** Display meaningful textual descriptions of remote ECUs based on their function code, enabling service tools to show a human-readable role for each address on the bus.
- **Function-Specific Logic:** Implement distributed logic that adapts behavior based on the function of a counterpart, for example, coupling a headway controller with a brake system controller.
- **Placeholder & Unassigned Handling:** Use `F_NOT_AVAILABLE` as a safe default for nodes that do not (yet) claim a specific function, or for masking undefined codes in protocol stacks.

Comparison with Similar Blocks

Within the ISOBUS/J1939 NAME specification, the Function field is only one of several attributes that together form the unique NAME of an ECU/component. Other global constant groups typically cover:

Attribute	Purpose	Example Values
Function (this group)	Identifies the subsystem role of the node	Engine, Transmission, Brakes, Display
ECU Instance	Differentiates multiple ECUs with the same function	0, 1, 2…
Industry Group	Classifies the application domain (non-specific, agriculture, construction, etc.)	Global, Agricultural, Construction/Mining
Vehicle System	Identifies the particular vehicle system instance	Powertrain, Chassis, Cab
Arbitrary Address Capable	Indicates if the unit supports arbitrary address claiming	yes/no

While the Function group is “industry non-specific”, other constant groups assign values for specific industry-related functions (e.g., agricultural implements). The distinction is important when building networks that must interoperate across standards: the non-specific codes guarantee a common baseline, whereas industry-specific codes extend the set for a vertical domain.

Conclusion

`Global_NAME_Functions` provides a complete, standards-aligned set of BYTE constants for the Function attribute of ISOBUS/J1939 NAME messages. By centralizing these codes in a global, read-only constant group, 4diac-IDE applications can reliably decode and encode the functional role of ECUs without scattering magic numbers through the control logic. Its coverage of 95 defined values plus the `F_NOT_AVAILABLE` placeholder makes it suitable for a wide range of vehicle and machinery automation tasks, from engine and transmission control to EV charging communication and auxiliary displays.

IG0_Device_Classes

• – ◦ • – ◦ • – ◦ •

Introduction

This global constants block defines industry group 0 specific device classes for vehicle systems, according to ISO 11783 (ISOBUS) PGN constants. It provides two standard constants to identify device classes for non-specific systems and unavailable systems.

Interface Structure

This element is a global constants definition, not a function block; therefore, it has no event or data inputs/outputs or adapters.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The constants provide standard values for device class identification in vehicle systems. They are used to indicate that a device class is non-specific (value 0) or not available (value 127). These constants are intended to be referenced in other parts of the system to avoid magic numbers and ensure consistency.

Technical Features

- **Data Type:** BYTE (8-bit unsigned integer)
- **Constants:**
  - `DC_NON_SPECIFIC_SYSTEM` = 0
  - `DC_NOT_AVAILABLE` = 127
- Defined as global constants, accessible across the project.
- Part of the `isobus::pgn::const` package.

State Overview

Not applicable; constants do not have states.

Application Scenarios

- Use `DC_NON_SPECIFIC_SYSTEM` to indicate that a device class is not specifically defined or is a generic system.
- Use `DC_NOT_AVAILABLE` to indicate that the device class information is not available or unknown.
- These constants can be used in PGN (Parameter Group Number) processing for ISOBUS compliant devices.

Comparison with Similar Blocks

This is a constants definition; typical alternatives might be direct literals, but using named constants improves maintainability and readability. Similar constants may exist for other industry groups; this one focuses on vehicle systems.

Conclusion

`IG0_Device_Classes` provides a standardized set of device class constants for ISOBUS vehicle systems, ensuring consistent and self-documenting code.

IG1_Device_Classes

• – ◦ • – ◦ • – ◦ •

Introduction

The `IG1_Device_Classes` is a global constants definition used in the 4diac IDE environment, specifically designed for Industry Group 1 (IG1) of ISO 11783 (ISOBUS) applications. It provides a set of predefined, application-independent device class identifiers that represent different types of vehicle systems within the agricultural and forestry domain. These constants allow consistent referencing of device classes throughout a project, enhancing code readability and maintainability.

Interface Structure

This element is not a function block (FB) or adapter, but rather a global constants container. Consequently, it does not expose event inputs, event outputs, data inputs, data outputs, or adapters in the conventional sense. The information it provides is a set of constant values that can be used globally within the application.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None (the constants are accessible as global read-only values, not as classical block outputs).

Adapters

None.

Functionality

The global constants defined here correspond to the device class byte values used in ISOBUS Parameter Group Numbers (PGNs) to indicate the type of vehicle system. They cover the most common categories as specified by the Industry Group 1 standard:

Constant Name	Value (BYTE)	Description
`DC_NON_SPECIFIC_SYSTEM`	0	Device Class / Vehicle System: Non-specific System
`DC_TRACTOR`	1	Device Class / Vehicle System: Tractor
`DC_TRAILER`	2	Device Class / Vehicle System: Trailer
`DC_NOT_AVAILABLE`	127	Device Class / Vehicle System: Not Available

These constants are intended to be used wherever a device class needs to be compared or set, ensuring that the same numeric values are always referenced by meaningful symbolic names.

Technical Features

- **Data Type:** All constants are of type `BYTE` (8-bit unsigned integer), matching the ISOBUS device class encoding.
- **Initial Values:** Each constant is assigned a fixed initial value that cannot be changed at runtime.
- **Global Scope:** The constants are declared as `VAR_GLOBAL CONSTANT`, making them accessible from any function block or sub-application within the project.
- **Standard Compliance:** The values follow the ISOBUS 11783 standard, specifically the Industry Group 1 device class assignments.
- **Namespace Organisation:** The associated `CompilerInfo` indicates that the constants belong to the `isobus::pgn::const` package, providing a clear logical grouping.

State Overview

Not applicable – this element does not contain any internal state or runtime behaviour. It is a static definition.

Application Scenarios

- **ISOBUS Communication:** When implementing ISOBUS communication stacks or interacting with electronic control units (ECUs), the device class is often embedded in a PGN. These constants can be used to set or decode the device class field.
- **Fleet Management:** In applications that differentiate between tractors, trailers, or other implements, these constants allow clear, error-free classification.
- **Configuration Tools:** Software tools that configure vehicle systems can utilise these constants to present user-friendly names instead of raw numeric values.
- **Testing and Simulation:** During simulation of ISOBUS messages, the constants help generate realistic data without hard-coding magic numbers.

Comparison with Similar Blocks

Other global constant definitions exist for other Industry Groups (e.g., `IG0_Device_Classes`, `IG2_Device_Classes`), each containing a different set of device classes. In contrast, this set is specific to Industry Group 1, focusing on tractors, trailers, and non-specific systems. There are also byte-enumeration type definitions (e.g., enum data types in ST) that could serve a similar purpose, but global constants are simpler and require no additional data types. When compared to hard-coded literals, using these constants significantly improves code maintainability and reduces mistakes.

Conclusion

`IG1_Device_Classes` provides a compact and standardised set of global constants for representing ISOBUS Industry Group 1 device classes. Even though it is not a function block, it plays a vital role in the development of ISOBUS-based applications by promoting code readability, consistency, and compliance with international standards. Its simplicity and global availability make it an indispensable building block for any project involving tractor, trailer, or other non-specific agricultural vehicle systems.

IG1_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Introduction

The `IG1_NAME_Functions` global constant definition belongs to the ISO 11783 (ISOBUS) protocol stack and provides a standardized set of byte-valued constants for identifying **Industry Group 1 (IG1)** specific function types within a vehicle's electronic control unit (ECU) network. This constant set is used to populate the Function field of the ISO 11783 NAME parameter, enabling unambiguous identification of the role each ECU plays on the bus.

The constants are organized according to the three device classes within Industry Group 1: non-specific systems, tractors, and trailers. Each constant maps a semantic function name to a numeric code that can be transmitted and interpreted across the bus network.

Interface Structure

Note: This artifact is a global constants definition, not a function block, adapter, or subapplication. Therefore, it does not expose event or data ports. The "interface" represents the set of named constant values that can be referenced globally by IEC 61499 applications.

Event Inputs

None. The `IG1_NAME_Functions` constant group defines data values only and does not provide any event inputs.

Event Outputs

None. There are no event outputs defined in this constant group.

Data Inputs

None. The constants are static and read-only at runtime; they cannot be modified via data inputs.

Data Outputs

The following global constants are made available to IEC 61499 applications for reference:

System Group	Constant Name	Type	Value	Description
Non-specific System	`F_TACHOGRAPH`	BYTE	128	Non-specific system, function: Tachograph
Non-specific System	`F_DOOR_CONTROLLER`	BYTE	129	Non-specific system, function: Door Controller
Non-specific System	`F_ARTICULATION_TURNTABLE_CONTROL`	BYTE	130	Non-specific system, function: Articulation Turntable Control – controls the articulation turntable for joined-body buses
Non-specific System	`F_BODY_TO_VEHICLE_INTERFACE_CONTROL`	BYTE	131	Non-specific system, function: Body-to-Vehicle Interface Control – manages interaction between vehicle functions and body functions
Non-specific System	`F_SLOPE_SENSOR`	BYTE	132	Non-specific system, function: Slope Sensor – measures a slope along an axis
Non-specific System	`F_RETARDER_DISPLAY`	BYTE	134	Non-specific system, function: Retarder Display – shows retarder information

				(driveline, exhaust, or engine)
Non-specific System	F_DIFFERENTIAL_LOCK_CONTROLLER	BYTE	135	Non-specific system, function: Differential Lock Controller
Non-specific System	F_LOW_VOLTAGE_DISCONNECT	BYTE	136	Non-specific system, function: Low-Voltage Disconnect – monitors starting battery voltage and disconnects auxiliary loads
Non-specific System	F_ROADWAY_INFORMATION	BYTE	137	Non-specific system, function: Roadway Information – provides information about the roadway in which the vehicle is traveling
Non-specific System	F_AUTOMATED_DRIVING	BYTE	138	Non-specific system, function: Automated Driving – hardware and software capable of performing part or all of the dynamic driving task
Non-specific System	F_NOT_AVAILABLE	BYTE	255	Non-specific system, function: Not Available – placeholder until an explicit function is assigned
Tractor	F_TRACTOR_FORWARD_ROAD_IMAGE_PROCESSING	BYTE	128	Tractor, function: Forward Road Image Processing – determines vehicle position from lane markings (advisory/warning only)
Tractor	F_TRACTOR_FIFTH_WHEEL_SMART_SYSTEM	BYTE	129	Tractor, function: Fifth Wheel Smart System – systems related to fifth wheel coupler operation and safety monitoring
Tractor	F_TRACTOR_CATALYST_FLUID_SENSOR	BYTE	130	Tractor, function: Catalyst Fluid Sensor – measures catalyst fluid temperature, level, and quality
Tractor	F_TRACTOR_ADAPTIVE_FRONT_LIGHTING_SYSTEM	BYTE	131	Tractor, function: Adaptive Front Lighting System – adjusts front lighting for current operating conditions
Tractor	F_TRACTOR_IDLE_CONTROL_SYSTEM	BYTE	132	Tractor, function: Idle Control System – automatically starts and stops the engine to reduce excess idle time
Tractor	F_TRACTOR_USER_INTERFACE_SYSTEM	BYTE	133	Tractor, function: User Interface System – two-way interface for climate, system parameters, and other settings

Tractor	F_TRACTOR_NOT_AVAILABLE	BYTE	255	Tractor, function: Not Available – placeholder until an explicit function is assigned
Trailer	F_TRAILER_NOT_AVAILABLE	BYTE	255	Trailer, function: Not Available – placeholder until an explicit function is assigned

Adapters

None. The constant group does not use or provide adapter interfaces.

Functionality

The IG1_NAME_Functions constant set serves as a lookup table for ISO 11783 NAME function codes within Industry Group 1. When composing an ECU NAME message, an application selects the appropriate constant based on the device's role (e.g., a tachograph, a door controller, or a slope sensor). The selected byte value is then encoded into the NAME field and broadcast on the bus, allowing other ECUs to determine the capability and purpose of the sending device.

The naming convention follows the pattern F_<SYSTEM>_<FUNCTION> where the system prefix is omitted for non-specific system functions, and the value space is organized so that each device class (non-specific, tractor, trailer) has its own starting code range. The value 255 is reserved in every system group to indicate that no specific function has been assigned yet.

Technical Features

- **Data Type:** All constants are declared as BYTE (8-bit unsigned) to fit the ISO 11783 NAME function field.
- **Value Ranges:** Within each system group, valid function codes start at 128 and increase sequentially. The value 255 is universally used as the "Not Available" placeholder.
- **Grouping:** Constants are logically grouped by device class – non-specific system, tractor, and trailer – following the DC_* (Device Class) definitions of the ISO 11783 standard.
- **Global Scope:** Being declared as VAR_GLOBAL CONSTANT, the constants are available throughout an IEC 61499 application without additional instantiation.
- **Compiler Hint:** The bundled compiler information (isobus::pgn::const) indicates that the constants are intended to be used within the ISOBUS PGN (Parameter Group Number) namespace, aiding tool-based integration.

State Overview

The IG1_NAME_Functions constant group is entirely static and stateless. There is no runtime state machine associated with it. The values are fixed at compile time and do not change during execution. Consequently, there are no states, transitions, or lifecycle operations to describe.

Application Scenarios

- **ECU Identification:** A control application that needs to send its NAME to the ISOBUS network can reference F_SLOPE_SENSOR (for a slope sensor) or F_TRACTOR_FIFTH_WHEEL_SMART_SYSTEM (for a tractor's fifth wheel system) to correctly advertise its function.
- **Function Assignment:** During system commissioning, a device may be programmed with F_NOT_AVAILABLE (255) until its final function is determined, ensuring it is recognized as placeholder equipment on the bus.
- **Multi-Function Devices:** A body controller that manages both doors and articulation turntable controls can use the corresponding constants to indicate multiple capabilities (if the underlying protocol supports multiple NAME entries).
- **Diagnostics and Monitoring:** Service tools can map received NAME function codes back to human-readable names using these constants, simplifying troubleshooting.

Comparison with Similar Blocks

The IG1_NAME_Functions constant group is specific to **Industry Group 1** of the ISO 11783 standard. Similar constant sets exist for other industry groups (e.g., IG0 – agriculture-specific, IG2 – forestry), each defining their own function codes. The key difference is that IG1 covers general freight and passenger vehicle applications (tractors, trailers, and non-specific systems) and therefore includes function codes such as tachograph, door controller, and automated driving which are not present in other groups.

In contrast to a typical IEC 61499 function block, which contains event and data interfaces along with an execution algorithm, this constant group has no executable behavior. It is a pure data artifact that serves as a shared vocabulary for function naming. It may be complemented by a separate adapter or function block that performs the actual encoding/decoding of NAME messages, but the constant group itself remains passive.

Conclusion

The `IG1_NAME_Functions` global constants provide a standardized, easy-to-reference mapping of ISO 11783 Industry Group 1 function codes. By centralizing the function definitions in a single location, it reduces the risk of value collisions and improves code readability across ISOBUS-related IEC 61499 applications. The constants are applicable to a wide range of vehicle control scenarios, from simple trailer monitoring to complex automated driving systems, and are an essential building block for any compliant ISOBUS implementation targeting agricultural, construction, or commercial vehicles.

IG1_Source_Addresses

• – ◦ • – ◦ • – ◦ •

Introduction

The **IG1_Source_Addresses** GlobalConstants package defines a set of named constant values representing the ISO 11783 (ISOBUS) **Industry Group 1 specific source addresses** for the J1939-style network. These constants map human-readable device identifiers (e.g., `SA_AUTOMATED_DRIVING_CONTROLLER_1`) to their assigned byte values in the range 128–247, a range reserved for Industry Group 1 (non-standard, agriculture/forestry oriented) ECU source addresses.

The constant definitions are provided as global read-only variables of type `BYTE`, enabling embedded logic blocks and application components to reference specific source addresses by name rather than by numeric literals. This improves code readability, reduces the risk of typing errors, and centralizes the address mapping for maintenance.

The package is organized as a `GLOBALCONSTANTS` block in compliance with the IEC 61499-1 standard and is intended to be included in projects that communicate over ISOBUS networks.

Interface Structure

Since this element is a **GlobalConstants** definition and not a function block, it does not expose event inputs, event outputs, data inputs, data outputs, or adapters. Instead, it provides a set of **global constant data elements** of type `BYTE` that can be used throughout an application. The following table lists all defined constants with their assigned values and comments.

Data Constants

Constant Name	Value (BYTE)	Description
`SA_RESERVED_128`	128	Reserved for future assignment by SAE; available for self-configurable ECUs
`SA_AUTOMATED_DRIVING_CONTROLLER_2`	156	Automated Driving Controller 2 (second instance)
`SA_ELECTRIC_PROPULSION_CONTROL_UNIT_3`	157	Electric Propulsion Control Unit #3
`SA_AUTOMATED_DRIVING_CONTROLLER_1`	158	Automated Driving Controller 1
`SA_ROADWAY_INFORMATION_SYSTEM`	159	Roadway Information System
`SA_ADVANCED_EMERGENCY_BRAKING_SYSTEM`	160	Advanced Emergency Braking System
`SA_FIFTH_WHEEL_SMART_SYSTEMS`	161	Fifth Wheel Smart Systems
`SA_SLOPE_SENSOR`	162	Slope Sensor
`SA_CATALYST_FLUID_SENSOR`	163	Catalyst Fluid Sensor
`SA_ON_BOARD_DIAGNOSTIC_UNIT_2`	164	On Board Diagnostic Unit #2
`SA_REAR_STEERING_AXLE_CONTROLLER_2`	165	Rear Steering Axle Controller #2

SA_REAR_STEERING_AXLE_CONTROLLER_3	166	Rear Steering Axle Controller #3
SA_INSTRUMENT_CLUSTER_2	167	Instrument Cluster #2 (auxiliary display)
SA_TRAILER_5_BRIDGE	168	Trailer #5 Bridge
SA_TRAILER_5_LIGHTING_ELECTRICAL	169	Trailer #5 Lighting-Electrical
SA_TRAILER_5_BRAKES_ABS_EBS	170	Trailer #5 Brakes (ABS-EBS)
SA_TRAILER_5_REEFER	171	Trailer #5 Reefer
SA_TRAILER_5_CARGO	172	Trailer #5 Cargo
SA_TRAILER_5_CHASSIS_SUSPENSION	173	Trailer #5 Chassis-Suspension
SA_OTHER_TRAILER_5_DEVICES	174	Other Trailer #5 Devices
SA_OTHER_TRAILER_5_DEVICES_175	175	Other Trailer #5 Devices (subnetwork)
SA_TRAILER_4_BRIDGE	176	Trailer #4 Bridge
SA_TRAILER_4_LIGHTING_ELECTRICAL	177	Trailer #4 Lighting-Electrical
SA_TRAILER_4_BRAKES_ABS_EBS	178	Trailer #4 Brakes (ABS-EBS)
SA_TRAILER_4_REEFER	179	Trailer #4 Reefer
SA_TRAILER_4_CARGO	180	Trailer #4 Cargo
SA_TRAILER_4_CHASSIS_SUSPENSION	181	Trailer #4 Chassis-Suspension
SA_OTHER_TRAILER_4_DEVICES	182	Other Trailer #4 Devices
SA_OTHER_TRAILER_4_DEVICES_183	183	Other Trailer #4 Devices (subnetwork)
SA_TRAILER_3_BRIDGE	184	Trailer #3 Bridge
SA_TRAILER_3_LIGHTING_ELECTRICAL	185	Trailer #3 Lighting-Electrical
SA_TRAILER_3_BRAKES_ABS_EBS	186	Trailer #3 Brakes (ABS-EBS)
SA_TRAILER_3_REEFER	187	Trailer #3 Reefer
SA_TRAILER_3_CARGO	188	Trailer #3 Cargo
SA_TRAILER_3_CHASSIS_SUSPENSION	189	Trailer #3 Chassis-Suspension
SA_OTHER_TRAILER_3_DEVICES	190	Other Trailer #3 Devices
SA_OTHER_TRAILER_3_DEVICES_191	191	Other Trailer #3 Devices (subnetwork)
SA_TRAILER_2_BRIDGE	192	Trailer #2 Bridge
SA_TRAILER_2_LIGHTING_ELECTRICAL	193	Trailer #2 Lighting-Electrical
SA_TRAILER_2_BRAKES_ABS_EBS	194	Trailer #2 Brakes (ABS-EBS)
SA_TRAILER_2_REEFER	195	Trailer #2 Reefer
SA_TRAILER_2_CARGO	196	Trailer #2 Cargo
SA_TRAILER_2_CHASSIS_SUSPENSION	197	Trailer #2 Chassis-Suspension
SA_OTHER_TRAILER_2_DEVICES	198	Other Trailer #2 Devices
SA_OTHER_TRAILER_2_DEVICES_199	199	Other Trailer #2 Devices (subnetwork)
SA_TRAILER_1_BRIDGE	200	Trailer #1 Bridge
SA_TRAILER_1_LIGHTING_ELECTRICAL	201	Trailer #1 Lighting-Electrical

SA_TRAILER_1_BRAKES_ABS_EBS	202	Trailer #1 Brakes (ABS-EBS)
SA_TRAILER_1_REEFER	203	Trailer #1 Reefer
SA_TRAILER_1_CARGO	204	Trailer #1 Cargo
SA_TRAILER_1_CHASSIS_SUSPENSION	205	Trailer #1 Chassis-Suspension
SA_OTHER_TRAILER_1_DEVICES	206	Other Trailer #1 Devices
SA_OTHER_TRAILER_1_DEVICES_207	207	Other Trailer #1 Devices (subnetwork)
SA_RESERVED_208	208	Reserved for future assignment by SAE (208–227)
SA_STEERING_INPUT_UNIT	228	Steering Input Unit
SA_BODY_CONTROLLER_2	229	Body Controller #2
SA_BODY_TO_VEHICLE_INTERFACE_CONTROL	230	Body-to-Vehicle Interface Control
SA_ARTICULATION_TURNTABLE_CONTROL	231	Articulation Turntable Control
SA_FORWARD_ROAD_IMAGE_PROCESSOR	232	Forward Road Image Processor
SA_DOOR_CONTROLLER_3	233	Door Controller #3
SA_DOOR_CONTROLLER_4	234	Door Controller #4
SA_TRACTOR_TRAILER_BRIDGE_2	235	Tractor/Trailer Bridge #2
SA_DOOR_CONTROLLER_1	236	Door Controller #1 (driver side)
SA_DOOR_CONTROLLER_2	237	Door Controller #2 (co-driver side)
SA_TACHOGRAPH	238	Tachograph
SA_ELECTRIC_PROPULSION_CONTROL_UNIT_1	239	Electric Propulsion Control Unit #1
SA_ELECTRIC_PROPULSION_CONTROL_UNIT_2	240	Electric Propulsion Control Unit #2
SA_WWH_OBD_TESTER	241	WWH-OBD Tester
SA_ELECTRIC_PROPULSION_CONTROL_UNIT_4	242	Electric Propulsion Control Unit #4
SA_BATTERY_PACK_MONITOR_1	243	Battery Pack Monitor #1
SA_BATTERY_PACK_MONITOR_2_APU_4	244	Battery Pack Monitor #2 / APU #4
SA_BATTERY_PACK_MONITOR_3_APU_3	245	Battery Pack Monitor #3 / APU #3
SA_BATTERY_PACK_MONITOR_4_APU_2	246	Battery Pack Monitor #4 / APU #2
SA_AUXILIARY_POWER_UNIT_APU_1	247	Auxiliary Power Unit (APU) #1

Event Inputs

Not applicable – this is a global constant definition, not a function block.

Event Outputs

Not applicable.

Data Inputs

Not applicable.

Data Outputs

Not applicable – the constants are global read-only values that may be referenced by any application component.

Adapters

Not applicable.

Functionality

The primary functionality of `IG1_Source_Addresses` is to provide a **centralized, named mapping** of ISOBUS source addresses for devices belonging to Industry Group 1. In an ISOBUS network, every electronic control unit must use a unique source address (0–253) to identify itself in transmitted messages. Industry Group 1 covers agricultural and forestry equipment, and the SAE J1939-based address allocation assigns specific addresses to device classes within this group.

By referencing the constants defined here, an application can:

- Assign a source address to a local ECU during network initialization.
- Filter incoming messages based on the source address of the transmitting device.
- Configure diagnostic or monitoring functions that need to distinguish between different controllers.
- Ensure that address values are consistent across the entire project without hard-coding numeric values.

The constant names follow a descriptive convention (prefix `SA_` followed by the device designation), which makes the purpose of each address immediately understandable in code.

Technical Features

- **Data type:** All constants are of type `BYTE` (unsigned 8-bit integer), which matches the J1939/ISOBUS source address width.
- **Address range:** Covers the Industry Group 1 range from 128 to 247, including reserved gaps (128–155, 208–227) and the explicitly defined addresses 156–247.
- **Reserved entries:** Two constants (`SA_RESERVED_128` and `SA_RESERVED_208`) explicitly declare the start of reserved ranges, allowing applications to check for reserved addresses.
- **Self-configuration support:** The comment for `SA_RESERVED_128` notes that the range 128–155 is "available for use by self-configurable ECUs," indicating support for dynamic address assignment.
- **Trailer subsystem addressing:** For each of the five possible trailers, seven addresses are defined (bridge, lighting/electrical, brakes/ABS-EBS, reefer, cargo, chassis/suspension, and other devices), allowing a full topology of towed equipment to be addressed.
- **Electric vehicle integration:** Multiple addresses for electric propulsion control units (ECUs #1–#4) and battery pack monitors are included, reflecting modern electric-drivetrain requirements.
- **IEC 61499 compliance:** The constants are declared within a `GlobalConstants` block, conforming to the IEC 61499-1 standard for global data in distributed industrial-process measurement and control systems.

State Overview

This element is **stateless**. It does not implement a state machine or exhibit runtime behavior – it simply provides compile-time constant values. Once defined, the values are immutable and available throughout the application lifecycle. There are no states to transition between, and no events trigger changes to the constants.

Application Scenarios

- **ISOBUS controller development:** A tractor or implement ECU that needs to claim a source address can assign the address from this constant set during startup, e.g., `SA_AUTOMATED_DRIVING_CONTROLLER_1` (158) for an autonomous driving controller.
- **Message filtering and routing:** A gateway or bridge (e.g., `SA_TRACTOR_TRAILER_BRIDGE_2`) can use these constants to discriminate messages from different trailer subsystems and forward them to appropriate network segments.
- **Diagnostics:** An on-board diagnostic unit (`SA_ON_BOARD_DIAGNOSTIC_UNIT_2`, `SA_WWH_OBD_TESTER`) can reference these constants to implement diagnostic services that query specific ECUs.
- **Electric vehicle fleets:** With addresses for propulsion control units and battery monitors, an electric tractor or implement can use these constants to manage the communication with multiple high-voltage components.
- **Multi-trailer configurations:** Agricultural operations that pull one or more trailers (e.g., grain carts, sprayers) can use the trailer-specific addresses to monitor and control lighting, brakes, reefers, and cargo conditions for each trailer individually.

Comparison with Similar Blocks

- **vs. generic constant definitions:** Most IEC 61499 projects define constants individually within function blocks or applications. `IG1_Source_Addresses` centralizes a large set of related constants in one global namespace, improving consistency and reducing duplication.
- **vs. hard-coded numeric values:** Using named constants eliminates the risk of typos and makes code self-documenting. Numeric address assignments are often scattered and error-prone when repeated.
- **vs. a function block with output pins:** A function block could expose source address values via output data ports, but it would need to be instantiated and would introduce runtime dependency. Global constants are available immediately without instantiation, making them more efficient for constant data.
- **vs. other ISOBUS address constant sets:** A similar GlobalConstants block for other industry groups (e.g., Group 2 addresses 0–127 or Group 3 addresses 248–253) would complement this file. This block specifically covers Group 1, while other groups would contain their own dedicated constants.

Conclusion

The **IG1_Source_Addresses** GlobalConstants block is an essential building block for any ISOBUS-based application developed in 4diac-ide. It provides a comprehensive, well-documented set of byte constants representing Industry Group 1 source addresses, covering propulsion, braking, towing, trailer, and body control systems. By using these named constants, developers ensure that their applications are readable, maintainable, and consistent with the SAE J1939 address allocation scheme. The stateless nature and global availability of the constants make them easy to integrate into any IEC 61499 application, improving both development efficiency and network interoperability.

IG2_Device_Classes

• – ◦ • – ◦ • – ◦ •

Introduction

The `IG2_Device_Classes` type defines a set of global constants for **Industry Group 2** (vehicle systems) as specified by the ISOBUS standard (ISO 11783). These constants provide standardized BYTE values that identify the device class of agricultural equipment, such as tractors, harvesters, and sprayers. They are intended for use in 4diac applications that communicate via ISOBUS and need to assign or decode device class information in PGN (Parameter Group Number) messages.

Interface Structure

This type is a **global constant definition**, not a function block, adapter, or subapplication. Consequently, it possesses no event or data interfaces. The constants themselves are the entire content of this type.

Event Inputs

Not applicable — no event inputs are defined.

Event Outputs

Not applicable — no event outputs are defined.

Data Inputs

Not applicable — no data inputs are defined.

Data Outputs

Not applicable — no data outputs are defined.

Adapters

Not applicable — no adapter interfaces are defined.

Functionality

The primary purpose of `IG2_Device_Classes` is to supply a consistent, human-readable set of constants that map to the device class codes used in ISOBUS. Each constant is declared as a `BYTE` (8-bit unsigned integer) with a fixed value. These constants are typically referenced in FB code (e.g., in ST or Structured Text) to set or compare the device class field of a message, ensuring that all parts of an application use the same numeric representation.

The following table lists all constants defined in this type:

Constant Name	Value (BYTE)	Description
DC_NON_SPECIFIC_SYSTEM	0	Non-specific system
DC_TRACTOR	1	Tractor
DC_TILLAGE	2	Tillage
DC_SECONDARY_TILLAGE	3	Secondary tillage
DC_PLANTERS_SEEDERS	4	Planters / Seeders
DC_FERTILIZERS	5	Fertilizers
DC_SPRAYERS	6	Sprayers
DC_HARVESTERS	7	Harvesters
DC_ROOT_HARVESTERS	8	Root harvesters
DC_FORAGE	9	Forage
DC_IRRIGATION	10	Irrigation
DC_TRANSPORT_TRAILER	11	Transport / Trailer
DC_FARM_YARD_OPERATIONS	12	Farm yard operations
DC_POWERED_AUXILIARY_DEVICES	13	Powered auxiliary devices
DC_SPECIAL_CROPS	14	Special crops
DC_EARTH_WORK	15	Earth work
DC_SKIDDER	16	Skidder
DC_SENSOR_SYSTEMS	17	Sensor systems
DC_TIMBER_HARVESTERS	19	Timber harvesters
DC_FORWARDERS	20	Forwarders
DC_TIMBER_LOADERS	21	Timber loaders
DC_TIMBER_PROCESSING_MACHINES	22	Timber processing machines
DC_MULCHERS	23	Mulchers
DC_UTILITY_VEHICLES	24	Utility vehicles
DC_SLURRY_MANURE_APPLICATORS	25	Slurry / Manure applicators
DC_FEEDERS_MIXERS	26	Feeders / Mixers
DC_WEEDERS	27	Weeders – non-chemical weed control
DC_TURF_AND_LAWN_CARE_MOWERS	28	Turf and lawn care mowers
DC_PRODUCT_MATERIAL_HANDLING	29	Product / Material handling
DC_NOT_AVAILABLE	127	Not available

All values are declared as `CONSTANT` and are therefore immutable at runtime. Using these constants instead of raw numeric literals improves code readability and maintainability, especially when dealing with ISOBUS messages that carry device class information.

Technical Features

- **Data type:** `BYTE` (unsigned 8-bit integer)
- **Scope:** Global — accessible from any FB or subapplication in the project
- **Predefined values:** 31 distinct constants covering the standard device classes defined in ISOBUS Industry Group 2

- **Immutability:** Constants are declared with the `CONSTANT` keyword and cannot be changed during execution
- **Standard compliance:** Values match the assignments defined by the ISOBUS standard (ISO 11783)

State Overview

This type is not a stateful entity. It does not contain any state variables, internal algorithms, or event-driven behavior. The constants are static and exist at compile time, providing a fixed mapping between names and numeric codes.

Application Scenarios

Typical use cases for `IG2_Device_Classes` include:

- **Message composition:** When building a PGN that includes a device-class field, assign a constant such as `DC_TRACTOR` to the corresponding byte.
- **Message parsing:** When decoding an incoming PGN, compare the received device-class value with constants to determine the type of equipment.
- **Configuration:** Use the constants in parameter files or configuration dialogs to let users select a device class from a list of human-readable names.
- **Filtering:** In fleet management or diagnostics, filter messages based on the device class to process only relevant data.

By using these global constants, applications avoid magic numbers and stay aligned with the ISOBUS standard, reducing errors and easing code review.

Comparison with Similar Blocks

In the 4diac-IDE environment, `IG2_Device_Classes` is a **global constant type** rather than a function block or adapter. The differences are:

- **Function blocks** contain executable logic and have inputs/outputs, whereas this type only provides named constants.
- **Adapters** define communication interfaces between subapplications; this type has no such capability.
- Compared to a simple variable declaration, using a global constant type centralizes all device-class definitions in one place, simplifying maintenance and ensuring consistency across a project.

Conclusion

The `IG2_Device_Classes` global constants offer a standardized, type-safe method for referencing ISOBUS device classes in 4diac applications. They are invaluable for developers working with agricultural machinery communication, as they eliminate arbitrary numeric values and provide self-documenting code. The constants are easy to include in any FB or subapplication and directly support the requirements of the ISOBUS protocol, making them a fundamental building block for compliant systems.

IG2_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Introduction

The `IG2_NAME_Functions` global constant set defines Industry Group 2 (Agricultural Equipment) specific function codes used in ISO 11783 (ISOBUS) networks. These constants represent the `function` part of the NAME (Name field) of a control function, as specified in ISO 11783-7. The values are used to identify the role of a device or software component on the bus, such as “Spray Rate Control”, “Tillage Depth Control”, or “Product Flow”. This constant set covers a wide range of agricultural applications, from tractors and tillage implements to harvesters, sensor systems, and specialised devices.

The constants are grouped by device class (e.g., Tractor, Tillage, Sprayers, Harvesters) and provide a consistent, human-readable naming scheme for developers working with ISOBUS protocols. Each constant is of type `BYTE` and has a predefined default value that is unique within its device-class context.

Interface Structure

Event Inputs

None – This is a global constant set and does not contain any event inputs.

Event Outputs

None – This is a global constant set and does not contain any event outputs.

Data Inputs

None – This is a global constant set and does not contain any data inputs.

Data Outputs

The set provides a total of **85** constants, each representing a specific function code. They are listed in the following table (grouped by device class, as in the XML).

Constant Name	Value	Short Description (excerpt)
Non-specific System		
F_NON_VIRTUAL_TERMINAL_DISPLAY	128	An operator display that cannot perform as a Virtual Terminal
F_OPERATOR_CONTROLS_MACHINE_SPECIFIC	129	Operator interface controls (auxiliary or proprietary)
F_TASK_CONTROLLER_MAPPING_COMPUTER	130	Task Controller / Mapping Computer
F_POSITION_CONTROL	131	Multiple axis position control of application boom
F_MACHINE_CONTROL	132	General machine control (outputs, ancillary functions)
F_FOREIGN_OBJECT_DETECTION	133	Detection of undesired objects in product flow
F_TRACTOR_ECU	134	Tractor interface unit on the implement bus
F_SEQUENCE_CONTROL_MASTER	135	Master controller in ISO11783-14 sequence control
F_PRODUCT_DOSING	136	Adds active ingredient to liquid carrier
F_PRODUCT_TREATMENT	137	Mixes treatment to dry product
F_RESERVED	138	Reserved for future “all operations stop” function
F_DATA_LOGGER	139	Non-task related data logger
F_DECISION_SUPPORT	140	Configures operation for optimal performance
F_LIGHTING_CONTROLLER	141	Controls electrical lights and status
F_TIM_SERVER	142	Tractor Implement Management (TIM) Server
F_NOT_AVAILABLE	255	Not available / unassigned
Tractor		
F_TRACTOR_AUXILIARY_VALVE_CONTROL	129	Control of tractor-mounted auxiliary valves
F_TRACTOR_REAR_HITCH_CONTROL	130	Rear hitch control
F_TRACTOR_FRONT_HITCH_CONTROL	131	Front hitch control
F_TRACTOR_TRACTOR_MACHINE_CONTROL	132	Machine control (outputs, ancillary functions)
F_TRACTOR_CENTER_HITCH_CONTROL	134	Center hitch control

F_TRACTOR_NOT_AVAILABLE	255	Not available
Tillage		
F_TILLAGE_TILLAGE_MACHINE_CONTROL	132	Control of outputs, ancillary functions
F_TILLAGE_TILLAGE_DEPTH_CONTROL	135	Depth of tillage tool in soil
F_TILLAGE_FRAME_CONTROL	136	Folding/unfolding of frame (transport vs field)
F_TILLAGE_NOT_AVAILABLE	255	Not available
Secondary Tillage		
F_SECONDARY_TILLAGE_SECONDARY_TILLAGE_MACHINE_CONTROL	132	Machine control
F_SECONDARY_TILLAGE_SECONDARY_TILLAGE_DEPTH_CONTROL	135	Depth control for surface preparation
F_SECONDARY_TILLAGE_FRAME_CONTROL	136	Frame control
F_SECONDARY_TILLAGE_NOT_AVAILABLE	255	Not available
Planters/Seeders		
F_PLANTERS_SEEDERS_SEED_RATE_CONTROL	128	Rate of seed placed in soil
F_PLANTERS_SEEDERS_SECTION_ON_OFF_CONTROL	129	On/Off control of individual sections
F_PLANTERS_SEEDERS_POSITION_CONTROL	131	Multiple axis position control (row guidance)
F_PLANTERS_SEEDERS_PLANTERS_SEEDERS_MACHINE_CONTROL	132	Machine control
F_PLANTERS_SEEDERS_PRODUCT_FLOW	133	Product flow measurement
F_PLANTERS_SEEDERS_PRODUCT_LEVEL	134	Product level in bin
F_PLANTERS_SEEDERS_DEPTH_CONTROL	135	Depth of seed/tuber placement
F_PLANTERS_SEEDERS_FRAME_CONTROL	136	Frame control
F_PLANTERS_SEEDERS_DOWN_PRESSURE	137	Ground contact pressure on delivery unit
F_PLANTERS_SEEDERS_NOT_AVAILABLE	255	Not available
Fertilizers		
F_FERTILIZERS_FERTILIZE_RATE_CONTROL	128	Fertilizer rate control
F_FERTILIZERS_SECTION_ON_OFF_CONTROL	129	Section on/off
F_FERTILIZERS_PRODUCT_PRESSURE	130	Monitoring pressure in sprayer booms
F_FERTILIZERS_POSITION_CONTROL	131	Boom position control
F_FERTILIZERS_FERTILIZERS_MACHINE_CONTROL	132	Machine control
F_FERTILIZERS_PRODUCT_FLOW	133	Product flow monitoring
F_FERTILIZERS_PRODUCT_LEVEL	134	Product level in bin
F_FERTILIZERS_HEIGHT_DEPTH_CONTROL	135	Height/depth of boom placement
F_FERTILIZERS_FRAME_CONTROL	136	Frame control
F_FERTILIZERS_NOT_AVAILABLE	255	Not available
Sprayers		
F_SPRAYERS_SPRAY_RATE_CONTROL	128	Rate of crop protection product applied
F_SPRAYERS_SECTION_ON_OFF_CONTROL	129	Section on/off

F_SPRAYERS_PRODUCT_PRESSURE	130	Pressure in delivery booms
F_SPRAYERS_POSITION_CONTROL	131	Boom position control
F_SPRAYERS_SPRAYERS_MACHINE_CONTROL	132	Machine control
F_SPRAYERS_PRODUCT_FLOW	133	Product flow
F_SPRAYERS_PRODUCT_LEVEL	134	Product level in tank
F_SPRAYERS_BOOM_HEIGHT_CONTROL	135	Boom height above soil/crop
F_SPRAYERS_FRAME_CONTROL	136	Frame control
F_SPRAYERS_NOT_AVAILABLE	255	Not available
Harvesters		
F_HARVESTERS_TAILING_MONITOR	128	Monitor quantity of unthreshed material
F_HARVESTERS_HEADER_CONTROL	129	Control of header reel height/rotation
F_HARVESTERS_PRODUCT_LOSS_MONITOR	130	Monitor grain loss to soil
F_HARVESTERS_PRODUCT_MOISTURE	131	Monitor moisture content of grain
F_HARVESTERS_HARVESTER_MACHINE_CONTROL	132	Machine control
F_HARVESTERS_PRODUCT_FLOW	133	Yield rate (grain flow into tank)
F_HARVESTERS_PRODUCT_LEVEL	134	Grain level in tank
F_HARVESTERS_HEADER_HEIGHT_CONTROL	135	Header height above soil / below crop top
F_HARVESTERS_NOT_AVAILABLE	255	Not available
Root Harvesters		
F_ROOT_HARVESTERS_ROOT_HARVESTERS_MACHINE_CONTROL	132	Machine control
F_ROOT_HARVESTERS_PRODUCT_FLOW	133	Product flow into tank
F_ROOT_HARVESTERS_PRODUCT_LEVEL	134	Product level in tank
F_ROOT_HARVESTERS_DEPTH_CONTROL	135	Lifter share depth
F_ROOT_HARVESTERS_NOT_AVAILABLE	255	Not available
Forage		
F_FORAGE_TWINE_WRAPPER_CONTROL	128	Twine wrapping around bale
F_FORAGE_PRODUCT_PACKAGING_CONTROL	129	Packaging process control
F_FORAGE_PRODUCT_MOISTURE	131	Moisture content of forage
F_FORAGE_FORAGE_MACHINE_CONTROL	132	Machine control
F_FORAGE_PRODUCT_FLOW	133	Forage yield rate
F_FORAGE_WORKING_HEIGHT_CONTROL	135	Cutting header height
F_FORAGE_NOT_AVAILABLE	255	Not available
Irrigation		
F_IRRIGATION_NOT_AVAILABLE	255	Not available
Transport/Trailer		
F_TRANSPORT_TRAILER_TRANSPORT_MACHINE_CONTROL	132	Machine control

F_TRANSPORT_TRAILER_UNLOAD_CONTROL	136	Trailer unloading process control
F_TRANSPORT_TRAILER_NOT_AVAILABLE	255	Not available
Farm Yard Operations		
F_FARM_YARD_OPERATIONS_NOT_AVAILABLE	255	Not available
Powered Auxiliary Devices		
F_POWERED_AUXILIARY_DEVICES_POWERED_DEVICES_MACHINE_CONTROL	128	Machine control
F_POWERED_AUXILIARY_DEVICES_NOT_AVAILABLE	255	Not available
Special Crops		
F_SPECIAL_CROPS_SPECIAL_CROP_MACHINE_CONTROL	132	Machine control
F_SPECIAL_CROPS_NOT_AVAILABLE	255	Not available
Earth Work		
F_EARTH_WORK_MATERIAL_RATE_CONTROL	128	Control of material processing rate
F_EARTH_WORK_EARTHWORKS_MACHINE_CONTROL	132	Machine control
F_EARTH_WORK_MATERIAL_FLOW	133	Material flow monitoring
F_EARTH_WORK_MATERIAL_LEVEL	134	Material level monitoring
F_EARTH_WORK_DEPTH_CONTROL	135	Working depth control
F_EARTH_WORK_NOT_AVAILABLE	255	Not available
Skidder		
F_SKIDDER_SKIDDER_MACHINE_CONTROL	132	Machine control
F_SKIDDER_NOT_AVAILABLE	255	Not available
Sensor Systems		
F_SENSOR_SYSTEMS_GUIDANCE_FEELER	128	Mechanical row position sensing
F_SENSOR_SYSTEMS_CAMERA_SYSTEM	129	Camera system for control operations
F_SENSOR_SYSTEMS_CROP_SCOUTING	130	Vegetation parameter measurement
F_SENSOR_SYSTEMS_MATERIAL_PROPERTIES_SENSING	131	Material properties detection (density, particle size, ...)
F_SENSOR_SYSTEMS_INERTIAL_MEASUREMENT_UNIT_IMU	132	Inertial measurement unit
F_SENSOR_SYSTEMS_PRODUCT_FLOW	133	Product flow monitoring
F_SENSOR_SYSTEMS_PRODUCT_LEVEL	134	Product level in tank/bin
F_SENSOR_SYSTEMS_PRODUCT_MASS	135	Product mass monitoring
F_SENSOR_SYSTEMS_VIBRATION_KNOCK	136	Vibration/knock behaviour measurement
F_SENSOR_SYSTEMS_WEATHER_INSTRUMENTS	137	Weather instruments identification
F_SENSOR_SYSTEMS_SOIL_SCOUTING	138	Soil physical parameter measurement
Timber Harvesters		
F_TIMBER_HARVESTERS_TIMBER_HARVESTORS_MACHINE_CONTROL	132	Machine control
Forwarders		
F_FORWARDERS_FORWARDERS_MACHINE_CONTROL	132	Machine control

Timber Loaders		
F_TIMBER_LOADERS_TIMBER_LOADERS_MACHINE_CONTROL	132	Machine control
Timber Processing Machines		
F_TIMBER_PROCESSING_MACHINES_TIMBER_PROCESSING_MACHINE_CONTROL	132	Machine control
Mulchers		
F_MULCHERS_MULCHER_MACHINE_CONTROL	132	Machine control
Utility Vehicles		
F_UTILITY_VEHICLES_UTILITY_MACHINE_CONTROL	132	Machine control
Slurry/Manure Applicators		
F_SLURRY_MANURE_APPLICATORS_SLURRY_MANURE_RATE_CONTROL	128	Rate of product placed on soil
F_SLURRY_MANURE_APPLICATORS_SECTION_ON_OFF_CONTROL	129	Section on/off
F_SLURRY_MANURE_APPLICATORS_PRODUCT_PRESSURE	130	Pressure in delivery booms
F_SLURRY_MANURE_APPLICATORS_SLURRY_MANURE_MACHINE_CONTROL	132	Machine control
F_SLURRY_MANURE_APPLICATORS_PRODUCT_FLOW	133	Product flow
F_SLURRY_MANURE_APPLICATORS_PRODUCT_LEVEL	134	Product level in bin
F_SLURRY_MANURE_APPLICATORS_BOOM_HEIGHT_CONTROL	135	Boom height above soil
Feeders/Mixers		
F_FEEDERS_MIXERS_FEEDER_MIXER_RATE_CONTROL	128	Rate of product distributed to livestock
F_FEEDERS_MIXERS_SECTION_ON_OFF_CONTROL	129	Section on/off (e.g., Left/Center/Right)
F_FEEDERS_MIXERS_PRODUCT_PRESSURE	130	Pressure in delivery units
F_FEEDERS_MIXERS_FEEDER_MIXER_MACHINE_CONTROL	132	Machine control
F_FEEDERS_MIXERS_PRODUCT_FLOW	133	Product flow
F_FEEDERS_MIXERS_PRODUCT_LEVEL	134	Product level in bin
F_FEEDERS_MIXERS_BOOM_HEIGHT_CONTROL	135	Boom height above soil
Weeders		
F_WEEDERS_WEEDER_MACHINE_CONTROL	132	Machine control (non-chemical weed control)
Turf and Lawn Care Mowers		
F_TURF_AND_LAWN_CARE_MOWERS_TURF_AND_LAWN_CARE_MOWERS_MACHINE_CONTROL	132	Machine control for grass cutting
Product/Material Handling		
F_PRODUCT_MATERIAL_HANDLING_PRODUCT_MATERIAL_HANDLING_MACHINE_CONTROL	132	Machine control
F_PRODUCT_MATERIAL_HANDLING_PRODUCT_MATERIAL_HANDLING_PRODUCT_FLOW	133	Product flow
F_PRODUCT_MATERIAL_HANDLING_PRODUCT_MATERIAL_HANDLING_PRODUCT_LEVEL	134	Product level in bin

Adapters

None – The constant set does not contain any adapters.

Functionality

The IG2_NAME_Functions constant set provides a standardised vocabulary for identifying the function of ISOBUS control functions. Each constant corresponds to a byte value that is used in the function field of the NAME message (per ISO 11783-7). This allows devices on an agricultural bus to advertise their capabilities, enabling proper configuration and interoperability. The constants are organised by device class (e.g., Tractor, Harvesters, Sprayers) to simplify lookup and ensure that function codes are unique within a given class.

For a given device class, a device may implement one or more of these functions (e.g., a sprayer might have function codes for boom height control, product flow, and section on/off). The availability of F_*_NOT_AVAILABLE (value 255) indicates that no explicit function has been assigned yet, serving as a placeholder during development.

Technical Features

- **Type:** All constants are of type BYTE (unsigned 8-bit integer).
- **Values:** Assigned values range from 128 to 142 (non-specific system) and 128 to 137 (specific device classes), with 255 universally used as "not available".
- **Grouping:** Constants are grouped into device classes as defined in the ISO 11783 standard (e.g., DC_TRACTOR = 1, DC_TILLAGE = 2, ...). This grouping is reflected in the naming convention: F_<DEVICE_CLASS>_<FUNCTION_NAME>.
- **Naming:** Names are descriptive and follow a pattern F_<device_class>_<specific_function> to avoid collisions (e.g., F_SPRAYERS_BOOM_HEIGHT_CONTROL vs. F_FERTILIZERS_HEIGHT_DEPTH_CONTROL).
- **Purpose:** The constants are intended for use in the NAME field of an ISOBUS device, particularly within the function byte of the "NAME" attribute.

State Overview

This constant set does not represent a state machine; it is a static list of predefined values. However, the presence of a NOT_AVAILABLE (255) constant can be considered a fallback state for devices that have not yet been assigned a specific function. Once a device is configured, its actual function value is set from one of the listed constants. The constants are not subject to runtime changes – they are global immutable values.

Application Scenarios

- **ISOBUS Device Development:** Implementers can include these constants in their code to set the NAME field when initialising a control function on an ISO 11783 bus.
- **Configuration Tools:** Software that configures agricultural equipment can use the constants to allow users to select the correct function from a dropdown or list.
- **Diagnostics and Monitoring:** Tools that analyse bus messages can interpret the function byte using these constants to display meaningful device roles.
- **System Simulation:** Simulation models of agricultural networks can use these constants to model various implement types.

Comparison with Similar Blocks

There is no direct comparison to function blocks, as this is a global constant definition. However, in the context of ISOBUS, comparable constant sets may exist for other industry groups (e.g., Industry Group 1 – forestry, Industry Group 3 – municipal). The structure differs in that those groups would have their own device classes and function codes. This set is specifically tailored for agricultural equipment (Industry Group 2). Unlike a function block, which encapsulates logic and has inputs/outputs, this constant set is purely declarative and serves as a data dictionary.

Conclusion

The IG2_NAME_Functions global constant set is an essential resource for developers working with ISO 11783 (ISOBUS) agricultural networks. It provides a comprehensive, standardised list of function codes covering a wide range of agricultural machinery and sensor systems. By using these constants, developers can ensure consistency and interoperability across different implementations, facilitating seamless communication between tractors, implements, and other devices on the bus.

IG2_Source_Addresses

• – ◦ • – ◦ • – ◦ •

Introduction

The `IG2_Source_Addresses` global constant definition provides a standardized set of source addresses for devices operating in Industry Group 2 (IG2) on an ISO 11783 (ISOBUS) network. These addresses are reserved for specific control functions and are used to uniquely identify devices or services within the network. The constants are defined as `BYTE` values and are intended to be used as preferred or fixed source addresses according to the ISO 11783 standard.

Interface Structure

This global constant type does not contain any event or data inputs. It exposes a collection of constant values as read-only data outputs, which are globally accessible to function blocks and applications within the 4diac-IDE environment.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

The following table lists the constant data outputs provided by this global constant type. Each entry represents a fixed source address value.

Name	Type	Initial Value	Description
`SA_RESERVED_128`	`BYTE`	128	Reserved source addresses 128–235 for self-configurable devices.
`SA_DATA_LOGGER`	`BYTE`	236	Data Logger – a server providing LOG functionality per ISO 11783-10.
`SA_TIM_SERVER`	`BYTE`	237	TIM Server – represents a Tractor Implement Management (TIM) server.
`SA_SEQUENCE_CONTROLLER`	`BYTE`	238	Sequence Controller – master in the Sequence Control System (ISO 11783-14).
`SA_POSITION_CONTROL`	`BYTE`	239	Position Control – multiple axial position control of movable device elements.
`SA_TRACTOR_ECU`	`BYTE`	240	Tractor ECU – gateway between tractor and implement bus, representing the tractor.
`SA_TAILINGS_MONITORING`	`BYTE`	241	Tailings Monitoring – monitors quantity of unthreshed material returned to threshing machine.
`SA_HEADER_CONTROL`	`BYTE`	242	Header Control – controls header reel rate and material delivery rate.
`SA_PRODUCT_LOSS_MONITORING`	`BYTE`	243	Product Loss Monitoring – monitors produce loss in harvest processes.
`SA_PRODUCT_MOISTURE_SENSING`	`BYTE`	244	Product Moisture Sensing – monitors produce moisture content.

SA_NON_VIRTUAL_TERMINAL_DISPLAY_IMPLEMENTBUS	BYTE	245	Non-Virtual Terminal Display (Implement Bus) – cab display connected to implement bus.
SA_OPERATOR_CONTROLS_MACHINE_SPECIFIC	BYTE	246	Operator Controls – machine-specific non-auxiliary operator control inputs.
SA_TASK_CONTROL_MAPPING_COMPUTER	BYTE	247	Task Control (Mapping Computer) – responsible for sending, receiving and logging process data.

Adapters

None.

Functionality

The `IG2_Source_Addresses` global constants are used to assign fixed, well-known source addresses to ISOBUS-compliant devices. By referencing these constants, developers ensure that their function blocks or applications use the correct addresses as defined by the ISO 11783 standard. The constants are grouped together to provide a single, maintainable definition for all IG2-specific source addresses, avoiding hard-coded magic numbers in the application logic.

Technical Features

- All constants are of type `BYTE` (range 0–255).
- The values are fixed and cannot be changed at runtime; they are compile-time constants.
- The definition is encapsulated in a global constants type, making the addresses easily reusable across different projects.
- The comments provide a concise description of each source address, aiding documentation and code readability.
- The constants are defined in accordance with the ISO 11783 series, particularly parts 4, 10, and 14.

State Overview

This global constant type does not maintain any internal state; it is a static collection of constant values. There are no state transitions or lifecycle operations. The constants are available immediately upon loading the type into a project.

Application Scenarios

`IG2_Source_Addresses` is typically used in the development of ISOBUS-based agricultural or forestry machinery, where multiple electronic control units (ECUs) need to communicate over the CAN bus. Examples include:

- Implementing a virtual terminal or display interface.
- Developing a tractor-implement management (TIM) system.
- Configuring data loggers or task controllers.
- Setting up position control or header control systems.
- Assigning source addresses during network self-configuration.

Comparison with Similar Blocks

In the 4diac-IDE environment, global constants are commonly used to define fixed configuration values. Similar constants might exist for other industry groups (e.g., IG1, IG3) or for other network-specific parameters. The main advantage of this collection is its focus on IG2 addresses, providing a comprehensive and standardized set that directly corresponds to ISO 11783 recommendations. Compared to using individual magic numbers, these named constants improve code clarity and reduce the risk of address conflicts.

Conclusion

The `IG2_Source_Addresses` global constant type offers a clean and standardized way to manage source addresses for ISO 11783 Industry Group 2 devices. By using these constants, developers can ensure compliance with international standards, simplify network configuration, and enhance the maintainability of their 4diac applications.

IG3_Device_Classes

• – ◦ • – ◦ • – ◦ •

Introduction

The `IG3_Device_Classes` global constant definition provides a standardized set of byte-encoded identifiers for Industry Group 3 (IG3) vehicle systems, specifically those used in construction, agricultural, and mobile machinery applications. These constants align with the SAE J1939 / ISOBUS 11783 protocol family and are intended to be used within PGN (Parameter Group Number) communications to clearly identify the type of system or device class being addressed. By using these symbolic names instead of raw numeric values, source code becomes more readable, maintainable, and less error-prone.

Interface Structure

This element is **not** a function block, adapter, or subapplication. It is a global constant set, which means it has no event inputs, event outputs, data inputs, data outputs, or adapter connections. Instead, it exposes a collection of named constants of type `BYTE` that can be referenced throughout an automation project.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The constants represent the officially assigned device classes for Industry Group 3 as defined by the ISOBUS standard. Each constant holds a distinct integer value (`0` to `17` and `127`) that corresponds to a specific vehicle system or a "not available" state. The values are used in fields of PGN messages to indicate the type of equipment on the network, facilitating correct data interpretation and routing between ECUs.

Technical Features

- **Data Type:** All constants are of type `BYTE`, ranging from 0 to 255.
- **Naming Convention:** Each constant is prefixed with `DC_` (Device Class) and follows a descriptive uppercase name.
- **Value Assignment:** Values are fixed and conform to the ISOBUS specification:
  - `0` – Non-specific system
  - `1` – Skid Steer Loader
  - `2` – Articulated Dump Truck
  - `3` – Backhoe
  - `4` – Crawler
  - `5` – Excavator
  - `6` – Forklift
  - `7` – Four Wheel Drive Loader
  - `8` – Grader
  - `9` – Milling Machine
  - `10` – Recycler and Soil Stabilizer
  - `11` – Binding Agent Spreader
  - `12` – Paver
  - `13` – Feeder
  - `14` – Screening Plant
  - `15` – Stacker
  - `16` – Roller

  - `17` – Crusher
  - `127` – Not Available (used when the class is unknown or not specified)
- **Compiler Integration:** The constant set is declared with a compiler package name `isobus::pgn::const`, indicating that it is part of a C++-like namespace for ISOBUS PGN constants, enabling seamless inclusion in embedded firmware projects.

State Overview

Not applicable – the element defines compile-time constants; no runtime state exists.

Application Scenarios

- **ISOBUS Network Communication:** When constructing or parsing PGNs (e.g., the "Vehicle System" fields), developers can directly use `DC_EXCAVATOR` instead of the numeric value `5`.
- **Device Identification:** In diagnostic or management messages, the constant helps identify the type of attached implement or vehicle, enabling appropriate control and display.
- **Firmware Development:** Embedded software for ECUs can include this constant set to ensure consistent device class reporting across different hardware variants.

Comparison with Similar Blocks

Unlike function blocks that contain algorithms and state machines, this global constant set has no executable logic. It is comparable to header files or enumerations in conventional programming. Other constant sets may exist for different industry groups (e.g., IG1, IG2) or for other PGN parameters, but this specific set focuses solely on IG3 device classes. Using symbolic constants avoids magic numbers and improves code clarity and maintainability.

Conclusion

The `IG3_Device_Classes` constant set is a foundational element for any project that uses ISOBUS (ISO 11783) communication in the construction and agricultural machinery domain. By providing readable, standardized names for device class values, it promotes code consistency, reduces mistakes, and simplifies integration. It is an essential resource for developers working on vehicle control units, implement controllers, or diagnostic tools that interact with IG3 systems.

IG3_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Introduction

The **IG3_NAME_Functions** global constant block defines the standardized function codes (Function field of the ISO 11783 NAME) for machinery belonging to **Industry Group 3**. Industry Group 3 encompasses construction and agricultural equipment, including loaders, excavators, graders, pavers, and other specialized machinery. These constants provide symbolic names for numeric byte values (0x80 to 0xFF) that identify the specific function of an electronic control unit (ECU) or device on an ISOBUS (ISO 11783) network. They facilitate the development of interoperable applications by replacing hard-coded numeric values with human-readable, maintainable constant names.

Interface Structure

This block is a **Global Constants** definition and does not contain any event inputs, event outputs, data inputs, data outputs, or adapter interfaces. It purely defines named constants that can be referenced across the 4diac development environment.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The block defines a set of byte constants grouped by **system categories** (Subsystem ID), following the ISOBUS NAME specification. Each constant has a value in the range **128 to 255** (0x80–0xFF) and corresponds to a specific function code for a given equipment type. The constants are organised as follows:

- **Non-specific system** – Functions applicable to any machine, such as:
  - `F_SUPPLEMENTAL_ENGINE_CONTROL_SENSING` = 128
  - `F_LASER_RECEIVER` = 129
  - `F_LOADER_CONTROL` = 135
  - `F_HYDRAULIC_VALVE_CONTROLLER` = 140
  - `F_JOYSTICK_CONTROL` = 141
  - `F_ALARM_DEVICE` = 146
  - and many others.
- **Skid Steer Loader** – Includes `F_SKID_STEER_LOADER_MAIN_CONTROLLER` = 128.
- **Crawler** – Includes `F_CRAWLER_BLADE_CONTROLLER` = 128.
- **Excavator** – Includes `F_EXCAVATOR_SLOPE_SENSOR` = 128.
- **Grader** – Includes `F_GRADER_HFWD_CONTROLLER` = 128.
- **Other equipment types** – For machines like articulated dump trucks, backhoes, forklifts, milling machines, pavers, etc., a generic `NOT_AVAILABLE` constant (value 255) is defined to be used until a specific function is assigned.

These constants allow developers to build ISOBUS applications that correctly identify the function of an ECU or device when constructing the NAME field, ensuring proper network communication and interpretation.

Technical Features

- **Data Type**: All constants are of type `BYTE` (unsigned 8-bit integer).
- **Value Range**: Values are restricted between 128 (0x80) and 255 (0xFF), as per the ISOBUS standard for function codes.
- **Grouping**: Constants are logically grouped by machine type, using meaningful prefixes (e.g., `F_LASER_`, `F_SKID_STEER_LOADER_`).
- **Standard Compliance**: The values and naming follow the ISO 11783-3 specification for NAME function codes in Industry Group 3.
- **Reusability**: Because these are global constants, they can be used across multiple function blocks and systems within a project without duplication or risk of inconsistent values.

State Overview

Not applicable – This block defines constants and does not have any state behavior.

Application Scenarios

- **ISOBUS Electronic Control Unit (ECU) Development**: Use these constants to set the Function portion of the NAME message when configuring an ECU address assignment.
- **Diagnostic and Monitoring Applications**: Decode received NAME messages to identify the type of equipment connected.
- **Machine-Specific Control Logic**: When implementing features for a specific machine (e.g., a skid steer loader), reference the corresponding constants to ensure the correct function identity.
- **Interoperability Testing**: Verify that generated NAME values comply with the industry standard by comparing against these predefined constants.

Comparison with Similar Blocks

In the ISOBUS system, different industry groups define their own sets of function codes. This block is specifically for **Industry Group 3** (construction and agricultural machinery). For example:

- **IG1** (agricultural equipment) has its own function codes for tractors and implements.
- **IG2** is used for forestry machines.
- **IG3** (this block) covers construction equipment.

The main difference lies in the system categories and the specific functions defined for each machine type. This block provides a comprehensive list for the most common construction machines, while other groups may have fewer or different constants. Additionally, all groups include a `NOT_AVAILABLE` value (255) as a placeholder.

Conclusion

The **IG3_NAME_Functions** global constant block is an essential resource for any ISOBUS-based application targeting construction machinery. By providing clear, descriptive names for function codes, it improves code readability, reduces errors, and ensures compliance with the ISO 11783 standard. Its integration into 4diac-IDE projects simplifies the development of robust, interoperable control systems for the construction industry.

IG3_Source_Addresses

• – ◦ • – ◦ • – ◦ •

Introduction

IG3_Source_Addresses is a global constant definition block used within IEC 61499-based applications to provide predefined ISO 11783 (ISOBUS) source addresses for Industry Group 3. This block defines a set of fixed BYTE values that identify specific device types and reserved address ranges on the ISOBUS network. By centralizing these constants, the block ensures consistent addressing across different function blocks and systems that require ISO 11783 compliant communication.

Interface Structure

This entity is a global constants definition rather than a typical function block. Therefore, it does not expose any event or data inputs/outputs, nor does it include adapters. Instead, it provides a collection of named constants that can be referenced throughout a project.

Event Inputs

None. This block does not have any event inputs.

Event Outputs

None. This block does not have any event outputs.

Data Inputs

None. This block does not accept any runtime data inputs.

Data Outputs

None in the traditional sense. However, the defined constants act as read-only values that can be read from other blocks. The following table lists all constants defined in this block:

Constant Name	Type	Value	Description
SA_RESERVED_128	BYTE	128	Source Address: thru 207 are reserved for future assignment by SAE - Used for dynamic address assignment (self-configurable)
SA_RESERVED_208	BYTE	208	Source Address: thru 223 are reserved for future assignment - Used for individual preassigned addresses
SA_ROTATION_SENSOR	BYTE	224	Source Address: Rotation Sensor - A device that measures the rotational angle around an axis.
SA_LIFT_ARM_CONTROLLER	BYTE	225	Source Address: Lift Arm Controller - Controls the lift arms and tilt functions on a construction loader, skid steer loader, or similar machine.

SA_SLOPE_SENSOR	BYTE	226	Source Address: Slope Sensor - A device that measures the slope along an axis.
SA_MAIN_CONTROLLER_SKID_STEER_LOADER	BYTE	227	Source Address: Main Controller - Skid Steer Loader - Primary system controller for skid steer loader
SA_LOADER_CONTROL	BYTE	228	Source Address: Loader Control - Controls the hydraulic system of the loader attachment of a loader/backhoe, wheel loader, skid steer, or similar vehicle
SA_LASER_TRACER	BYTE	229	Source Address: Laser Tracer - A device that receives a laser strike and reports the vertical and horizontal position.
SA_LAND_LEVELING_SYSTEM_DISPLAY	BYTE	230	Source Address: Land Leveling System Display - This device displays position information at a remote location.
SA_SINGLE_LAND_LEVELING_SYSTEM_SUPERVISOR	BYTE	231	Source Address: Single Land Leveling System Supervisor - This device is the Land Leveling System Supervisor for a single control loop.
SA_LAND_LEVELING_ELECTRIC_MAST	BYTE	232	Source Address: Land Leveling Electric Mast - A device that moves a Sensor to maintain a specific position.
SA_SINGLE_LAND_LEVELING_SYSTEM_OPERATOR_INTERFACE	BYTE	233	Source Address: Single Land Leveling System Operator Interface - A component that allows the user to control the Land Leveling System and display information about the operation of the system.
SA_LASER_RECEIVER	BYTE	234	Source Address: Laser Receiver - A device that receives a laser strike, and reports the specific position.
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_1	BYTE	235	Source Address: Supplemental Sensor Processing Unit #1
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_2	BYTE	236	Source Address: Supplemental Sensor Processing Unit #2
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_3	BYTE	237	Source Address: Supplemental Sensor Processing Unit #3
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_4	BYTE	238	Source Address: Supplemental Sensor Processing Unit #4
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_5	BYTE	239	Source Address: Supplemental Sensor Processing Unit #5
SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_6	BYTE	240	Source Address: Supplemental Sensor Processing Unit #6
SA_ENGINE_MONITOR_1	BYTE	241	Source Address: Engine Monitor #1
SA_ENGINE_MONITOR_2	BYTE	242	Source Address: Engine Monitor #2
SA_ENGINE_MONITOR_3	BYTE	243	Source Address: Engine Monitor #3
SA_ENGINE_MONITOR_4	BYTE	244	Source Address: Engine Monitor #4

SA_ENGINE_MONITOR_5	BYTE	245	Source Address: Engine Monitor #5
SA_ENGINE_MONITOR_6	BYTE	246	Source Address: Engine Monitor #6
SA_ENGINE_MONITOR_7	BYTE	247	Source Address: Engine Monitor #7

Adapters

None. This block does not contain any adapters.

Functionality

The IG3_Source_Addresses block provides a centralized and standardized set of source addresses for devices operating in Industry Group 3 of the ISO 11783 (ISOBUS) protocol. Source addresses are essential for uniquely identifying devices on the CAN-based network. By defining them as named constants, the block removes ambiguity in application logic and simplifies maintenance. Devices such as sensors, controllers, and monitors can reference these constants directly, ensuring that the correct address is used without manual hard-coding.

Technical Features

- All constants are of type BYTE, ranging in value from 128 to 247.
- Values 128 and 208 represent reserved address ranges as defined by the SAE and are used for dynamic address assignment and individual preassigned addresses, respectively.
- Values 224 through 234 are assigned to specific device functions such as rotation sensors, lift arm controllers, slope sensors, loaders, laser tracers, land leveling equipment, and laser receivers.
- Values 235 through 240 are reserved for supplemental sensor processing units.
- Values 241 through 247 are assigned to engine monitors (up to 7 monitors).
- The block is defined as a global constant container, making the constants visible to all function blocks within the project without duplication.

State Overview

This block does not have an internal state machine or execute any runtime behavior. It is a static definition entity. The constants it defines are immutable and do not change during the execution of an application.

Application Scenarios

- **ISOBUS Implement Control:** When building a control system for agricultural or construction machinery that uses ISOBUS, these constants can be used to assign source addresses to the various electronic control units (ECUs).
- **Sensor Integration:** Connecting rotation sensors, slope sensors, or laser receivers to an ISOBUS network, ensuring each device uses the correct predefined address.
- **Engine Monitoring:** Using the engine monitor address constants to identify up to seven engine monitoring devices on a machine.
- **Reserved Address Handling:** Applications that need to dynamically assign addresses can use the reserved address constants (SA_RESERVED_128 and SA_RESERVED_208) as starting points for self-configuration or preassigned address management.

Comparison with Similar Blocks

In contrast to IEC 61499 function blocks that process data via event and data connections, this block is purely declarative. It serves a similar purpose to other global constant containers, such as `IG1_Source_Addresses` or `IG2_Source_Addresses`, which define source addresses for Industry Groups 1 and 2. The main difference lies in the specific device types and address ranges assigned to each industry group. Compared to an ordinary function block, this definition offers no executable logic but provides a clear, reusable reference that enhances maintainability and reduces errors in address allocation.

Conclusion

The IG3_Source_Addresses global constant block provides a well-organized collection of ISO 11783 source addresses for Industry Group 3 devices. It simplifies the development of ISOBUS-compliant applications by offering named, typed constants that are easy to reference and manage. This approach promotes code clarity, consistency, and correct network addressing, making it a valuable building block for machinery control and telemetry systems.

IG4_Device_Classes

• – ◦ • – ◦ • – ◦ •

Introduction

`IG4_Device_Classes` is a global constant container for Industry Group 4 specific device classes, also referred to as vehicle systems. It defines a set of named `BYTE` constants that represent standardized vehicle system categories. These constants are intended for use in ISOBUS/J1939-related applications and PGN processing.

The constant names use the prefix `DC_`, meaning Device Class, and provide readable symbols for numeric codes used to identify vehicle systems.

• – ◦ • – ◦ • – ◦ •

Interface Structure

`IG4_Device_Classes` is not an executable function block, adapter, or subapplication. It does not have an IEC 61499 event/data interface. The following interface categories are therefore not applicable.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

• – ◦ • – ◦ • – ◦ •

Functionality

The main function of `IG4_Device_Classes` is to provide a single source of truth for device class codes related to Industry Group 4 vehicle systems. Instead of using magic numbers in function blocks or ST/CFC applications, a developer can reference the named constants directly.

These constants can be used when constructing or decoding ISOBUS/J1939 device names, vehicle system identifiers, or any protocol data field that uses a `BYTE` value to represent a device class.

• – ◦ • – ◦ • – ◦ •

Technical Features

- All constants are defined as `BYTE` values.
- Values are fixed and read-only.
- Constant names use the `DC_` prefix for consistent identification.
- The constants belong to the compiler package `isobus::pgn::const`.
- The set covers the standard device class values for Industry Group 4 vehicle systems.

Constant	Value	Description
`DC_NON_SPECIFIC_SYSTEM`	0	Non-specific System
`DC_SYSTEM_TOOLS`	10	System tools
`DC_SAFETY_SYSTEMS`	20	Safety systems
`DC_GATEWAY`	25	Gateway
`DC_POWER_MGMT_LIGHTING`	30	Power management and lighting systems
`DC_STEERING_SYSTEMS`	40	Steering systems
`DC_PROPULSION_SYSTEMS`	50	Propulsion systems
`DC_NAVIGATION_SYSTEMS`	60	Navigation systems
`DC_COMMUNICATIONS_SYSTEMS`	70	Communications systems
`DC_INSTRUMENTATION_GENERAL`	80	Instrumentation/general systems
`DC_ENVIRONMENTAL_HVAC`	90	Environmental (HVAC) systems
`DC_DECK_CARGO_FISHING`	100	Deck, cargo, and fishing equipment systems
`DC_NOT_AVAILABLE`	127	Not Available

• – ◦ • – ◦ • – ◦ •

State Overview

Not applicable. `IG4_Device_Classes` is a global constant container and does not define an execution state machine. No states, transitions, or event-driven behavior are associated with this element.

• – ◦ • – ◦ • – ◦ •

Application Scenarios

Typical application areas include:

- ISOBUS/J1939 device name construction.
- Identification of vehicle systems in electronic control units.
- Filtering or routing of messages based on device class.
- Configuration of agricultural or vehicle systems.
- Replacement of numeric constants with readable symbolic names in ST and CFC code.

• – ◦ • – ◦ • – ◦ •

Comparison with Similar Blocks

`IG4_Device_Classes` is not a function block and therefore does not perform calculations or process data flows. In this respect, it is similar to a global variable list or a constant table.

Compared with executable function blocks, it offers the following benefits:

- Fixed, non-modifiable values.
- Global availability across a project.
- Clear symbolic naming.
- Direct support for protocol-specific device class values.

It can be compared with other industry-group-specific constant sets, where the same naming convention is used but the assigned values and device class definitions may differ.

• – ◦ • – ◦ • – ◦ •

Conclusion

`IG4_Device_Classes` provides a clear and maintainable set of `BYTE` constants for Industry Group 4 vehicle device classes. It removes ambiguity from numeric values and supports ISOBUS/J1939 application development by offering meaningful symbolic names. The element is simple, static, and suitable for use wherever device class information must be referenced consistently.

IG4_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Introduction

This global constant list defines the Industry Group 4 (IG4) specific function codes used in ISO 11783 (ISOBUS) networks. It provides a standardized mapping between numeric function identifiers and their meanings within various system categories, such as propulsion, navigation, communications, instrumentation, and more. The constants enable consistent identification of ECU functions across the network.

Interface Structure

Since this is a global constant list rather than a function block, it does not possess a traditional interface with events or adapters. The constants are globally accessible and can be referenced in any function block, subapplication, or script within the project.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

The constants defined in this list serve as global data outputs. They are all of type `BYTE` and carry predefined values that represent specific functions. The constants are grouped by system category, as summarized in the table below:

Constant Name	Value	System	Function Description
F_ALARM_SYSTEM_CONTROL_FOR_MARINE_ENGINES	128	Non-specific System	Alarm System Control for Marine Engines
F_PROTECTION_SYSTEM_FOR_MARINE_ENGINES	129	Non-specific System	Protection System for Marine Engines
F_DISPLAY_FOR_PROTECTION_SYSTEM_FOR_MARINE_ENGINES	130	Non-specific System	Display for Protection System for Marine Engines
F_NOT_AVAILABLE	255	Non-specific System	Not Available
F_SYSTEM_TOOLS_NOT_AVAILABLE	255	System tools	Not Available
F_SAFETY_SYSTEMS_NOT_AVAILABLE	255	Safety systems	Not Available
F_GATEWAY	130	Gateway	Gateway
F_POWER_MGMT_LIGHTING_SWITCH	130	Power management and lighting systems	Switch
F_POWER_MGMT_LIGHTING_LOAD	140	Power management and lighting systems	Load
F_STEERING_SYSTEMS_FOLLOW_UP_CONTROLLER	130	Steering systems	Follow-up Controller
F_STEERING_SYSTEMS_MODE_CONTROLLER	140	Steering systems	Mode Controller
F_STEERING_SYSTEMS_AUTOMATIC_STEERING_CONTROLLER	150	Steering systems	Automatic Steering Controller
F_STEERING_SYSTEMS_HEADING_SENSORS	160	Steering systems	Heading Sensors
F_PROPULSION_SYSTEMS_ENGINEROOM_MONITORING	130	Propulsion systems	Engineroom monitoring
F_PROPULSION_SYSTEMS_ENGINE_INTERFACE	140	Propulsion systems	Engine Interface
F_PROPULSION_SYSTEMS_ENGINE_CONTROLLER	150	Propulsion systems	Engine Controller
F_PROPULSION_SYSTEMS_ENGINE_GATEWAY	160	Propulsion systems	Engine Gateway
F_PROPULSION_SYSTEMS_CONTROL_HEAD	170	Propulsion systems	Control Head
F_PROPULSION_SYSTEMS_ACTUATOR	180	Propulsion systems	Actuator
F_PROPULSION_SYSTEMS_GAUGE_INTERFACE	190	Propulsion systems	Gauge Interface
F_PROPULSION_SYSTEMS_GAUGE_LARGE	200	Propulsion systems	Gauge Large
F_PROPULSION_SYSTEMS_GAUGE_SMALL	210	Propulsion systems	Gauge Small
F_PROPULSION_SYSTEMS_PROPULSION_SENSOR_GATEWAY	220	Propulsion systems	Propulsion Sensors & Gateway
F_NAVIGATION_SYSTEMS_SOUNDER_DEPTH	130	Navigation systems	Sounder, depth
F_NAVIGATION_SYSTEMS	140	Navigation systems	(no specific function description)
AVIGATION_SYSTEMS_GLOBAL_NAVIGATION_SATELLITE_SYSTEM_GNSS	145	Navigation systems	Global Navigation Satellite System (GNSS)
F_NAVIGATION_SYSTEMS_LORAN_C	150	Navigation systems	Loran C

F_NAVIGATION_SYSTEMS_SPEED_SENSORS	150	Navigation systems	Speed Sensors
F_NAVIGATION_SYSTEMS_TURN_RATE_INDICATOR	160	Navigation systems	Turn Rate Indicator
F_NAVIGATION_SYSTEMS_INTEGRATED_NAVIGATION	170	Navigation systems	Integrated Navigation
F_NAVIGATION_SYSTEMS_RADAR_AND_OR_RADAR_PLOTTING	200	Navigation systems	Radar and/or Radar Plotting
TION_SYSTEMS_ELECTRONIC_CHART_DISPLAY_INFORMATION_SYSTEM_ECDIS	205	Navigation systems	Electronic Chart Display & Information System (ECDIS)
F_NAVIGATION_SYSTEMS_ELECTRONIC_CHART_SYSTEM_ECS	210	Navigation systems	Electronic Chart System (ECS)
F_NAVIGATION_SYSTEMS_DIRECTION_FINDER	220	Navigation systems	Direction Finder
UNICATIONS_SYSTEMS_EMERGENCY_POSITION_INDICATING_BEACON_EPIRB	130	Communications systems	Emergency Position Indicating Beacon (EPIRB)
COMMUNICATIONS_SYSTEMS_AUTOMATIC_IDENTIFICATION_SYSTEM	140	Communications systems	Automatic Identification System
F_COMMUNICATIONS_SYSTEMS_DIGITAL_SELECTIVE_CALLING_DSC	150	Communications systems	Digital Selective Calling (DSC)
F_COMMUNICATIONS_SYSTEMS_DATA_RECEIVER	160	Communications systems	Data Receiver
F_COMMUNICATIONS_SYSTEMS_SATELLITE	170	Communications systems	Satellite
F_COMMUNICATIONS_SYSTEMS_RADIO_TELEPHONE_MF_HF	180	Communications systems	Radio-Telephone (MF/HF)
F_COMMUNICATIONS_SYSTEMS_RADIO_TELEPHONE_VHF	190	Communications systems	Radio-Telephone (VHF)
F_INSTRUMENTATION_GENERAL_TIME_DATE_SYSTEMS	139	Instrumentation/general systems	Time/Date systems
F_INSTRUMENTATION_GENERAL_VOYAGE_DATA_RECORDER	140	Instrumentation/general systems	Voyage Data Recorder
F_INSTRUMENTATION_GENERAL_INTEGRATED_INSTRUMENTATION	155	Instrumentation/general systems	Integrated Instrumentation
F_INSTRUMENTATION_GENERAL_GENERAL_PURPOSE_DISPLAYS	165	Instrumentation/general systems	General Purpose Displays
F_INSTRUMENTATION_GENERAL_GENERAL_SENSOR_BOX	170	Instrumentation/general systems	General Sensor Box
F_INSTRUMENTATION_GENERAL_WEATHER_INSTRUMENTS	180	Instrumentation/general systems	Weather Instruments
F_INSTRUMENTATION_GENERAL_TRANSDUCER_GENERAL	190	Instrumentation/general systems	Transducer/general
F_INSTRUMENTATION_GENERAL_NMEA_0183_CONVERTER	200	Instrumentation/general systems	NMEA 0183 Converter
F_ENVIRONMENTAL_HVAC_NOT_AVAILABLE	255	Environmental (HVAC) systems	Not Available
F_DECK_CARGO_FISHING_NOT_AVAILABLE	255	Deck, cargo, and fishing equipment systems	Not Available

Adapters

None.

Functionality

This global constant list provides a centralized set of function identifiers for Industry Group 4 (IG4) as defined in the ISO 11783 / ISOBUS standard. Each constant assigns a unique numeric value (BYTE) to a specific function within a system category. These constants are used in parameter group numbers (PGNs) to identify the source or purpose of a message, enabling correct routing and interpretation of data across the network.

Technical Features

- All constants are of type `BYTE` and defined as global constants.
- Values range from 128 to 255, with `255` commonly used for "Not Available".
- The constants are grouped logically by system (e.g., Propulsion systems, Navigation systems, etc.).
- The list is intended to be used as a reference for mapping function codes in ISOBUS networks.

State Overview

Not applicable. This is a constant definition, not a stateful block.

Application Scenarios

- Referencing function codes when constructing or parsing ISOBUS messages.
- Assigning a function ID to an ECU in a network configuration.
- Ensuring consistent naming and values across multiple function blocks and applications.

Comparison with Similar Blocks

This is not a typical function block but a global constant list. Similar lists exist for other industry groups (e.g., IG1, IG2, IG3, IG5) that define their own specific functions. The structure and usage are analogous, but the actual function codes and system categories differ per industry group.

Conclusion

The IG4_NAME_Functions global constant list is an essential resource for developers working with ISOBUS networks in the marine and heavy-duty vehicle domain. It standardizes function identification, improving interoperability and maintainability of industrial control applications.

IG4_Source_Addresses

• – ◦ • – ◦ • – ◦ •

Introduction

IG4_Source_Addresses is a global constant definition block within the **isobus::pgn::const** package, providing standardized ISOBUS (ISO 11783) source address constants for **Industry Group 4** (Marine applications). It defines 22 named constants covering reserved address ranges and assigned device addresses for marine engine systems, displays, transmissions, and propulsion sensor gateways. These constants are used as symbolic references instead of raw numeric values, improving code readability and maintainability in ISOBUS-based applications targeting marine powertrain and monitoring equipment.

Interface Structure

This element is a **GlobalConstants** container, not a standard function block. It does not expose any event, data, or adapter ports. Instead, it publishes named constants that can be referenced globally within the application project.

Event Inputs

None. Constants blocks do not have event inputs.

Event Outputs

None. Constants blocks do not have event outputs.

Data Inputs

None. The values are statically defined at compile time.

Data Outputs

None. The values are statically defined at compile time.

Adapters

None. This block does not contain adapters.

The following table lists all defined constants (all of type `BYTE`):

Constant Name	Value	Description
`SA_RESERVED_128`	128	Reserved (128–207) for dynamic address assignment
`SA_RESERVED_208`	208	Reserved (208–227) for individual preassigned addresses
`SA_PROPULSION_SENSOR_HUB_GATEWAY_1`	228	Propulsion Sensor Hub & Gateway #1
`SA_PROPULSION_SENSOR_HUB_GATEWAY_2`	229	Propulsion Sensor Hub & Gateway #2
`SA_PROPULSION_SENSOR_HUB_GATEWAY_3`	230	Propulsion Sensor Hub & Gateway #3
`SA_PROPULSION_SENSOR_HUB_GATEWAY_4`	231	Propulsion Sensor Hub & Gateway #4
`SA_TRANSMISSION_3`	232	Transmission ECU #3
`SA_TRANSMISSION_4`	233	Transmission ECU #4
`SA_TRANSMISSION_5`	234	Transmission ECU #5
`SA_TRANSMISSION_6`	235	Transmission ECU #6
`SA_DISPLAY_1_FOR_PROTECTION_SYSTEM_FOR_MARINE_ENGINES`	236	Display #1 for Marine Engine Protection System
`SA_PROTECTION_SYSTEM_FOR_MARINE_ENGINES`	237	Protection System ECU for Marine Engines
`SA_ALARM_SYSTEM_CONTROL_1_FOR_MARINE_ENGINES`	238	Alarm System Control #1 for Marine Engines
`SA_ENGINE_3`	239	Engine ECU #3
`SA_ENGINE_4`	240	Engine ECU #4
`SA_ENGINE_5`	241	Engine ECU #5
`SA_MARINE_DISPLAY_1`	242	Marine Display #1
`SA_MARINE_DISPLAY_2`	243	Marine Display #2
`SA_MARINE_DISPLAY_3`	244	Marine Display #3
`SA_MARINE_DISPLAY_4`	245	Marine Display #4
`SA_MARINE_DISPLAY_5`	246	Marine Display #5
`SA_MARINE_DISPLAY_6`	247	Marine Display #6

Functionality

The constant set implements the ISOBUS source address scheme for Industry Group 4. It covers:

- **Reserved ranges** (128–207 and 208–227) that are kept available for SAE future assignments, distinguishing between dynamically self-configurable addresses and individually preassigned addresses.
- **Propulsion sensor hub gateways** (228–231) for up to four sensor hub/gateway devices.
- **Transmission ECUs** (232–235) for the third through sixth transmission in a multi-transmission marine propulsion system. The first two transmissions are typically defined in other Industry Group constant sets.
- **Marine engine protection and alarm systems** (236–238), including a dedicated display for the protection system.
- **Engine ECUs** (239–241) for engines #3 through #5. Engines #1 and #2 use addresses from lower Industry Group ranges (as defined by the ISOBUS standard).
- **Marine displays** (242–247) for up to six display devices.

Technical Features

- All constants are of type `BYTE` with initial values within the 0–255 range, matching the single-byte address field of ISOBUS (ISO 11783) and SAE J1939.
- The constants are declared as `VAR_GLOBAL CONSTANT`, making them immutable and globally accessible across all function blocks in the project.
- Naming follows a descriptive `SA_` prefix convention, directly mapping the symbolic name to the numeric source address, eliminating the need to memorize raw address values.
- The block is part of the `isobus::pgn::const` package, providing a modular namespace for ISOBUS-related constants.

State Overview

This is a passive data container. It has no runtime state, no internal algorithms, and no transition logic. The values are fixed at compile time and remain constant during execution. Therefore, no state machine or operational states apply.

Application Scenarios

- **Marine engine monitoring systems** – determining the correct source address for engine ECUs when configuring a J1939/ISOBUS network with multiple engines.
- **Multi-transmission marine propulsion control** – assigning addresses for the third through sixth transmission ECUs.
- **Alarm and protection systems** – configuring alarm system control units, protection system displays, and the protection system ECU itself.
- **Display integration** – addressing up to six marine displays on the same bus.
- **Development and test setups** – using the reserved constants to validate dynamic address assignment logic or preconfigured address allocation.

Comparison with Similar Blocks

Aspect	IG4_Source_Addresses	IG0/IG1/IG2/IG3 Source Address Sets
Industry Group	4 (Marine)	0–3 (Other equipment classes)
Address range	128–247 (partially, with gaps)	Varies per group
Content focus	Marine engines, transmissions, displays, protection systems	Agriculture, construction, forestry, etc.
Reserved addresses	128–227 (two separate ranges)	Group-specific

The structure of this constant set follows the same pattern as other Industry Group source address blocks in the `isobus::pgn::const` package. The main difference is the specific device types and the address ranges assigned by the ISOBUS standard for marine applications.

Conclusion

IG4_Source_Addresses provides a complete, standardized set of source address constants for ISOBUS marine applications. It eliminates hard-coded numeric addresses, reduces configuration errors, and ensures compliance with the ISOBUS address allocation scheme. By using these constants in PGN construction and network filtering logic, developers can reliably build interoperable marine powertrain and monitoring systems. The block is particularly useful when a marine bus contains multiple engines, transmissions, sensor hubs, or displays that must be uniquely addressed without conflicts.

IG5_Device_Classes

• – ◦ • – ◦ • – ◦ •

Introduction

The **IG5_Device_Classes** global constants type defines device class identifiers for the Industry Group 5 (IG5) within the ISOBUS protocol. These constants are used to specify the type of vehicle system or equipment in a J1939 network, particularly for industrial-process control stationary applications (e.g., generator sets). The type is part of the `isobus.pgn.const` package and provides named values for the raw BYTE codes that appear in PGNs.

Interface Structure

This type does not represent a typical function block with event or data inputs/outputs. Instead, it declares global constants that can be referenced throughout a 4diac application. The following subsections list the standard interface elements.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The **IG5_Device_Classes** type defines two constants:

Constant Name	Value (BYTE)	Description
`DC_INDUSTRIAL_GEN_SET`	`0`	Device class for industrial-process control stationary (generator sets).
`DC_NOT_AVAILABLE`	`127`	Indicates that the device class is not available.

These constants are intended to be used in contexts where a device class needs to be identified in compliance with ISOBUS Industry Group 5 specifications. They help ensure consistent naming and avoid magic numbers in application logic.

Technical Features

- **Type**: Defined as `BYTE` (8-bit unsigned integer), matching the J1939 data size.
- **Package**: `isobus.pgn.const` – grouped with other PGN-related constants.
- **Scope**: Global constants, accessible from any function block within the project.
- **Naming Convention**: Prefix `DC_` (Device Class) followed by a descriptive suffix.
- **Values**: Standardized according to ISOBUS definitions; `0` for generator sets, `127` for "not available".

State Overview

Not applicable – this type does not define any state machines or behavioral states. It solely provides constant values.

Application Scenarios

- **Generator Set Monitoring**: When building an application that reads or writes the device class from a J1939 message, use `DC_INDUSTRIAL_GEN_SET` to identify the device as a stationary generator set.
- **Fallback Handling**: Use `DC_NOT_AVAILABLE` to represent an unknown or uninitialized device class, facilitating error handling or default scenarios.
- **Protocol Compliance**: Referencing these constants ensures that your application adheres to the ISOBUS Industry Group 5 naming and value conventions, improving interoperability.

Comparison with Similar Blocks

Since **IG5_Device_Classes** is a global constant type, it can be compared to other constant types that define device classes for different industry groups (e.g., IG1, IG2). Those would follow a similar pattern but with different value sets and package names. Unlike function blocks or adapters, this type has no runtime behavior; it only provides compile-time constants.

Conclusion

The **IG5_Device_Classes** global constants type offers a clean, portable way to reference Industry Group 5 device class codes in 4diac applications. By using these named constants, developers can avoid hard-coding numeric values, increase code readability, and align with ISOBUS standards. Its simplicity and clear documentation make it a valuable asset for any project dealing with vehicle systems or industrial-process control equipment.

IG5_NAME_Functions

• – ◦ • – ◦ • – ◦ •

Introduction

This document describes the **IG5_NAME_Functions** global constant set, which defines Industry Group 5 (IG5) specific function codes for ISOBUS (ISO 11783) implementation. These constants are used to identify the function of devices and components within the "Industrial-Process Control - Stationary (Gen-Sets)" system category, as well as a general "Not Available" placeholder. The constant set provides a standardized naming convention for referencing these function codes in control logic and data mapping.

Interface Structure

Since **IG5_NAME_Functions** is a global constant definition, it does not contain event inputs, event outputs, data inputs, data outputs, or adapters. It exposes a set of named `BYTE` constants that can be referenced in IEC 61499 applications or 4diac-IDE projects.

Event Inputs

Not applicable – this global constant set does not process events.

Event Outputs

Not applicable – this global constant set does not generate events.

Data Inputs

Not applicable – this global constant set does not have data inputs.

Data Outputs

Not applicable – this global constant set does not have data outputs.

Adapters

Not applicable – this global constant set does not include adapters.

Functionality

The purpose of this global constant set is to provide symbolic names for function codes defined by the ISOBUS Industry Group 5 specification. These codes are used to describe the role or operation of a device connected to an ISOBUS network. By using the constants, applications can avoid hard-coded numeric values and improve readability and maintainability.

The constants defined in this set are:

Constant Name	Value	Description
F_INDUSTRIAL_GEN_SET_SUPPLEMENTAL_ENGINE_CONTROL_SENSING	128	Reserved for supplemental engine control sensing in gen-sets.
F_INDUSTRIAL_GEN_SET_GENERATOR_SET_CONTROLLER	129	Generator set controller for collecting data and controlling the generator.
F_INDUSTRIAL_GEN_SET_GENERATOR_VOLTAGE_REGULATOR	130	Generator voltage regulator for maintaining output voltage.
F_INDUSTRIAL_GEN_SET_CHOKE_ACTUATOR	131	Choke actuator for controlling air flow on a gas engine.
F_WELL_STIMULATION_PUMP	132	Well stimulation pump used in oil and gas drilling applications.

F_INDUSTRIAL_GEN_SET_NOT_AVAILABLE	255	Placeholder for gen-set function when no explicit function is assigned.
F_NOT_AVAILABLE	255	General placeholder for any device whose function is not yet defined.

These constants are used to populate the Function field in ISOBUS parameter groups (PGNs) that transmit device information or operational status.

Technical Features

- **Data Type**: All constants are of type BYTE (8-bit unsigned integer).
- **Compatibility**: The values correspond to the ISO 11783-6 standard for function codes in the industrial/process control domain.
- **Naming Convention**: Each constant follows the pattern F_<SYSTEM>_<FUNCTION> to clearly indicate its intended application area.
- **Reusability**: The constants can be directly referenced in IEC 61499 function block implementations, ST (Structured Text) algorithms, or any other 4diac-IDE programming interface.
- **Documentation**: Each constant carries an inline comment that describes its system and function context.

State Overview

This global constant set does not have a state machine; it defines static values that remain unchanged during runtime. No state transitions, initialization sequences, or internal states are applicable.

Application Scenarios

The IG5_NAME_Functions global constants are typically used in automation projects that involve:

- **Generator Set Control**: Applications monitoring and controlling stationary diesel/gas generator sets used for emergency power or continuous operation.
- **ISOBUS Communication**: Systems that implement ISOBUS PGNs to exchange device function information, such as the "Name" object or function-specific parameters.
- **Oil & Gas Drilling**: Integration of well stimulation pumps that communicate their operational parameters over an ISOBUS network.
- **Device Identification**: Assigning function codes to allow a central controller (e.g., a generator set controller) to identify the type and role of connected peripherals.

In 4diac-IDE, these constants can be imported into a project and used alongside function blocks that require a BYTE input for function selection or reporting.

Comparison with Similar Blocks

Since this is a global constant definition rather than a function block, there is no direct comparison to FB types. However, similar constant sets exist for other Industry Groups (e.g., IG1–IG4, IG6) in the ISOBUS standard. The key difference is the set of specific functions defined for gen-set and oil/gas applications in IG5. If an application requires a different industry group, a corresponding global constant set must be used. This set provides the necessary values for typical IG5 equipment but does not include function codes for other domains.

Conclusion

The IG5_NAME_Functions global constants offer a clean and standardized way to reference ISOBUS function codes for industrial gen-sets and related equipment. By using these constants, developers can ensure compatibility with the ISO 11783 standard, reduce the risk of data misinterpretation, and make their control logic more readable and maintainable. The set covers the most common functions in this domain and provides a universal "Not Available" fallback, making it a practical resource for ISOBUS-based automation projects.

IG5_Source_Addresses

• – ◦ • – ◦ • – ◦ •

Introduction

The **IG5_Source_Addresses** type is a global constants definition set used in the 4diac IDE environment. It provides a collection of predefined ISO 11783 (ISOBUS) Industry Group 5 source addresses (SA). These

constants are employed to uniquely identify electronic control units (ECUs) on a CAN bus network in agricultural and forestry machinery applications, following the SAE J1939 / ISOBUS protocol.

The constants are declared as `BYTE` values and cover the source address range from 128 to 247, which is assigned to various engine, generator, and supplemental sensor processing units within Industry Group 5.

Interface Structure

Event Inputs

Not applicable – this is a global constants type and does not contain any event inputs.

Event Outputs

Not applicable – this is a global constants type and does not contain any event outputs.

Data Inputs

Not applicable – this is a global constants type and does not contain any data inputs.

Data Outputs

Not applicable – the type exports constant values that are globally accessible, but they are not dynamic data outputs of a function block.

Global Constants

The following table lists all constant declarations provided by this type:

Constant Name	Type	Value (decimal)	Description
`SA_RESERVED_128`	BYTE	128	Reserved range (128–207) for future SAE assignment – used for dynamic address assignment
`SA_RESERVED_208`	BYTE	208	Reserved range (208–229) for future assignment – used for individual preassigned addresses
`SA_GENERATOR_VOLTAGE_REGULATOR`	BYTE	230	Generator Voltage Regulator – controls generator output voltage
`SA_ENGINE_3`	BYTE	231	Engine #3 – ECU for the third engine within a system
`SA_ENGINE_4`	BYTE	232	Engine #4 – ECU for the fourth engine within a system
`SA_ENGINE_5`	BYTE	233	Engine #5 – ECU for the fifth engine within a system
`SA_GENERATOR_SET_CONTROLLER`	BYTE	234	Generator Set Controller – used for data collection and control of a generator system
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_1`	BYTE	235	Supplemental Sensor Processing Unit #1
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_2`	BYTE	236	Supplemental Sensor Processing Unit #2
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_3`	BYTE	237	Supplemental Sensor Processing Unit #3
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_4`	BYTE	238	Supplemental Sensor Processing Unit #4
`SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_5`	BYTE	239	Supplemental Sensor Processing Unit #5

SA_SUPPLEMENTAL_SENSOR_PROCESSING_UNIT_6	BYTE	240	Supplemental Sensor Processing Unit #6
SA_ENGINE_MONITOR_1	BYTE	241	Engine Monitor #1
SA_ENGINE_MONITOR_2	BYTE	242	Engine Monitor #2
SA_ENGINE_MONITOR_3	BYTE	243	Engine Monitor #3
SA_ENGINE_MONITOR_4	BYTE	244	Engine Monitor #4
SA_ENGINE_MONITOR_5	BYTE	245	Engine Monitor #5
SA_ENGINE_MONITOR_6	BYTE	246	Engine Monitor #6
SA_ENGINE_MONITOR_7	BYTE	247	Engine Monitor #7

Functionality

This global constants type centralizes all Industry Group 5 source addresses in one location. By referencing these named constants, applications and function blocks can avoid hard-coded numeric CAN identifiers, making the system configuration more readable, maintainable, and less error-prone. Each constant corresponds to a specific SA value defined by the ISOBUS/J1939 standard for a particular ECU function. The constants are designed to be used in network layer logic, address claiming procedures, and PGN (Parameter Group Number) filtering within ISOBUS applications.

Technical Features

- **Data Type:** All constants are of type `BYTE` (8-bit unsigned integer), fitting the SA range of 0–255 as defined in the J1939 standard.
- **Address Range:** The defined values cover the Industry Group 5 range (128–247), including reserved addresses, generator-related units, and engine monitoring ECUs.
- **Global Accessibility:** As a `GLOBALCONSTANTS` type, the constants are globally visible throughout the 4diac project once instantiated, enabling consistent use across multiple function blocks and applications.
- **Read-Only:** The constants are immutable; their values cannot be changed at runtime.
- **Standard Compliance:** Values follow the SAE J1939 / ISO 11783 source address assignments for Industry Group 5, ensuring protocol interoperability.

State Overview

Not applicable – this is a global constants type without any internal state machine or runtime behavior. It provides static values only.

Application Scenarios

- **ISOBUS Network Configuration:** Use these constants when configuring the source address (SA) for an ECU in a tractor or implement network.
- **Address Claiming Logic:** Reference the constants in function blocks that perform dynamic address claiming to ensure the correct SA is used for a specific device type.
- **CAN Message Filtering:** Use the constants to filter incoming messages based on the source address of the transmitting ECU.
- **Multi-Engine Systems:** Assign the correct engine SA (e.g., `SA_ENGINE_3`, `SA_ENGINE_4`) when managing multiple engine ECUs on the same bus.
- **Generator Monitoring:** Refer to `SA_GENERATOR_VOLTAGE_REGULATOR` and `SA_GENERATOR_SET_CONTROLLER` to identify generator-related messages.
- **Engine Diagnostics:** Use `SA_ENGINE_MONITOR_1` through `SA_ENGINE_MONITOR_7` to distinguish between up to seven engine monitoring units.

Comparison with Similar Blocks

Unlike function blocks (FBs) or adapters, `IG5_Source_Addresses` is a global constants type, meaning it has no event processing, no execution states, and no data flow. Its purpose is purely declarative. Compared to locally declared variables inside a function block, this global constants type offers:

- **Reusability:** Defined once and used across many FBs and applications.
- **Maintainability:** Updating a constant value in one location propagates to all references.

- **Safety:** Prevents accidental modification during runtime.
- **Standardization:** Enforces consistent naming and values across the entire project.

Other global constants types may exist for other industry groups (e.g., IG1–IG4), but this one is specifically scoped to Industry Group 5, avoiding naming clashes and keeping the address space organized.

Conclusion

The `IG5_Source_Addresses` global constants type provides a clean, standardized way to manage ISO 11783 Industry Group 5 source addresses in a 4diac project. Its fixed, well-documented constant set simplifies development of ISOBUS-compliant applications, improves code readability, and ensures that source addresses are always used consistently across distributed function blocks. It is an essential building block for any agricultural or forestry machinery control system operating on an ISOBUS/CAN network.

Industry_Groups

• – ◦ • – ◦ • – ◦ •

Introduction

This is a global constants group in the 4diac IDE, part of the ISOBUS library. It defines 8 industry group identifiers used in ISOBUS parameter group numbers (PGNs). These constants are of type BYTE and range from 0 to 7, representing different industrial sectors such as agricultural, construction, marine, etc.

Interface Structure

As a GlobalConstants type, it has no event inputs, event outputs, data inputs, data outputs, or adapters. Instead, it provides a set of global constants that can be referenced by other function blocks. The defined constants are listed below:

Constant Name	Type	Initial Value	Comment
IG_GLOBAL	BYTE	0	Industry Group: Global, applies to all
IG_ON_HIGHWAY_EQUIPMENT	BYTE	1	Industry Group: On-Highway Equipment
IG_AGRICULTURAL_AND_FORESTRY_EQUIPMENT	BYTE	2	Industry Group: Agricultural and Forestry Equipment
IG_CONSTRUCTION_EQUIPMENT	BYTE	3	Industry Group: Construction Equipment
IG_MARINE	BYTE	4	Industry Group: Marine
IG_INDUSTRIAL_PROCESS_CONTROL_STATIONARY	BYTE	5	Industry Group: Industrial-Process Control-Stationary (Gen-Sets)
IG_RESERVED_FOR_FUTURE_ASSIGNMENT_BY_SAE	BYTE	6	Industry Group: Reserved for future assignment by SAE
IG_RESERVED_FOR_FUTURE_ASSIGNMENT_BY_SAE_7	BYTE	7	Industry Group: Reserved for future assignment by SAE

Event Inputs

No event inputs are defined.

Event Outputs

No event outputs are defined.

Data Inputs

No data inputs are defined.

Data Outputs

No data outputs are defined.

Adapters

No adapters are defined.

Functionality

The constants defined in this group are used to identify the industry group in ISOBUS PGNs. Each PGN includes a 3-bit industry group field that determines the applicable scope for the message. By using these named constants, application developers can avoid magic numbers and improve code readability and maintainability.

Technical Features

- All constants are of type BYTE and are declared as global constants.
- Values range from 0 to 7, with 6 and 7 reserved for future assignment by SAE.
- The constants are compiled into the package `isobus::pgn::const`, making them accessible across the entire ISOBUS library.

State Overview

Not applicable; this is a constant definition type and does not contain state machines or behavior.

Application Scenarios

These constants are primarily used in the development of ISOBUS-compliant agricultural, construction, and forestry machinery control systems. They are typically referenced when constructing or parsing PGNs to set or interpret the industry group field. For example, an ECU in an agricultural tractor would use `IG_AGRICULTURAL_AND_FORESTRY_EQUIPMENT` (value 2) when sending PGNs.

Comparison with Similar Blocks

Similar global constants groups exist for other ISOBUS enumerations, such as protocol versions, function codes, or address claims. This particular group focuses specifically on industry group identification, making it distinct from other constant sets. Unlike function blocks, it does not provide any processing logic but merely a repository of standard constants.

Conclusion

The `Industry_Groups` global constants group is an essential building block for ISOBUS application development, providing well-defined and standardized identifiers for industry sectors. Its use enhances code clarity and ensures compliance with the ISOBUS standard.

Manufacturer_IDs

• – ○ • – ○ • – ○ •

Introduction

This document describes the **Manufacturer_IDs** global constant set from the ISOBUS protocol library. It defines a comprehensive catalog of numeric identifiers for manufacturers as specified by the ISO 11783 (ISOBUS) standard. These constants are used to unambiguously identify the manufacturer of an electronic control unit (ECU) in parameter group numbers (PGNs) and other ISOBUS messages.

Interface Structure

Since this is a global constant set, it does not have a traditional function block interface with event or data inputs/outputs. Instead, it exposes a large number of named constants, each of type `UINT` (unsigned 16-bit integer), that can be referenced by name in any IEC 61499 application.

Event Inputs

None – this is a data-only constant set.

Event Outputs

None – no event triggers are generated.

Data Inputs

None – constants are read-only and cannot be altered at runtime.

Data Outputs

Each constant acts as an implicit data output. All constants are of type `UINT` and are accessible by their symbol name (e.g., `M_CATERPILLAR`). The set includes over 1,600 named manufacturers, each with a unique value. A few examples are listed below:

Constant Name	Value	Description
`M_FOR_EXPERIMENTAL_OR_DEVELOPMENTAL_USE_ONLY`	0	Reserved for experimental use.
`M_BENDIX_COMMERCIAL_VEHICLE_SYSTEMS`	1	Bendix Commercial Vehicle Systems.
`M_ALLISON_TRANSMISSION`	2	Allison Transmission, Inc.
`M_CATERPILLAR`	8	Caterpillar Inc.
`M_DEERE_AND_COMPANY_PRECISION_FARMING`	12	Deere & Company, Precision Farming.
`M_AGCO`	102	AGCO (formerly AGCO GmbH & Co.).
`M_CNH_INDUSTRIAL_N_V`	94	CNH Industrial N.V.
`M_RAVEN_INDUSTRIES`	151	Raven Industries, Inc.
`M_TRIMBLE`	1856	Trimble Navigation.

The full list is defined in the XML source and includes entries from international manufacturers across agriculture, construction, marine, and automotive sectors.

Adapters

None – this set does not provide adapter interfaces.

Functionality

The primary purpose of these constants is to provide a standardized, human-readable way to refer to manufacturer identification codes in ISOBUS applications. When constructing or parsing ISOBUS messages, developers can reference these constants instead of memorizing numeric codes. The constants are defined with the prefix `M_` for easy identification in code.

Technical Features

- **Data Type:** All constants are of type `UINT` (16-bit unsigned integer), matching the ISO 11783 manufacturer ID field width.
- **Total Count:** Over 1,600 distinct manufacturer entries, including some reserved or unused codes (e.g., `M_UNUSED_64`, `M_UNUSED_76`, `M_UNUSED_228`).
- **Versioning:** The set is versioned; comments next to each constant indicate the date of last update or addition.
- **Global Scope:** As a global constant set, these values are available across all function blocks in an application without explicit wiring.
- **Read-Only:** The constants are declared as `CONSTANT` and cannot be modified during runtime.

State Overview

This constant set does not represent a state machine or dynamic behavior. It is static and immutable; all values are fixed at compile time. Therefore, no state transitions or runtime states exist.

Application Scenarios

- **ISOBUS ECU Implementation:** When developing an ECU for agricultural or construction equipment, the manufacturer ID must be set according to the ISO 11783 specification. Using these constants ensures that the correct numeric value is used.
- **Message Filtering:** In a diagnostic tool or bus monitor, manufacturer IDs are used to filter or identify messages from specific ECUs. These constants can be used in comparison logic.
- **Configuration and Setup:** When configuring a virtual terminal or task controller, the manufacturer ID of the implement or tractor can be selected from this list.
- **Emulation and Testing:** For simulation or testing, the constant `M_FOR_EXPERIMENTAL_OR_DEVELOPMENTAL_USE_ONLY` (0) can be used to indicate a non-production device.

Comparison with Similar Blocks

This constant set is analogous to other global constant sets in the ISOBUS library, such as `PGN_Constants` or `SPN_Constants`. While those sets define parameter group numbers and suspect parameter numbers, `Manufacturer_IDs` focuses exclusively on manufacturer identification. The key difference is that manufacturer IDs are unique to each equipment manufacturer, while PGNs and SPNs are standardized by the ISOBUS protocol itself. This set is often referenced alongside PGN and SPN definitions to construct complete ISOBUS messages.

Conclusion

The `Manufacturer_IDs` global constant set provides a reliable and comprehensive reference for ISOBUS manufacturer identification. By encapsulating the numeric codes as named constants, it enhances code readability, reduces errors, and simplifies compliance with the ISO 11783 standard. Its use is recommended in any IEC 61499 application that interacts with ISOBUS networks.

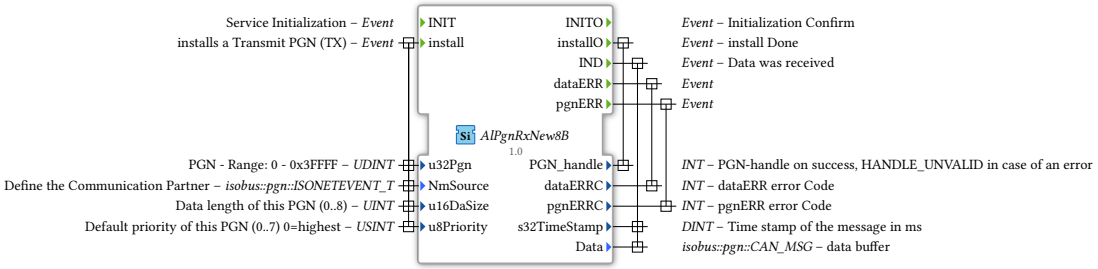
rx

AlPgnRxNew8B

• – ○ • – ○ • – ○ •

Introduction

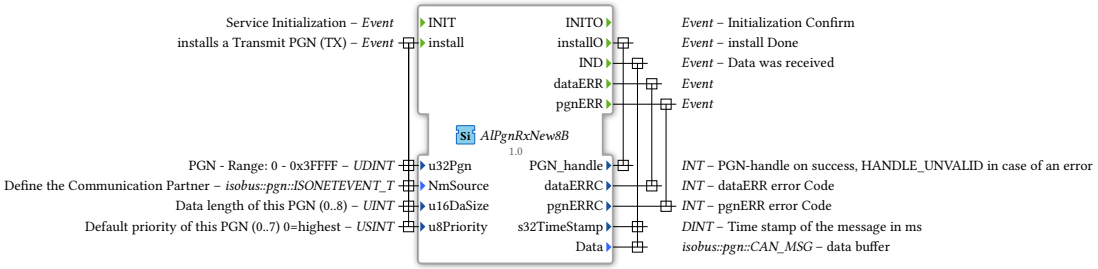
The function block `AlPgnRxNew8B` is used to receive data via a CAN network according to the ISOBUS standard (ISO 11783). Its main purpose is the installation and management of Parameter Group Numbers (PGNs) for receiving messages and the provision of received data to the application. It is part of a specialized library for ISOBUS communication.



Interface Structure

Event Inputs

- **`INIT`**: Starts the initialization of the function block. Acknowledgement is provided via `INITO`.



* **`install`**: Triggers the installation of a new PGN to be received (Transmit PGN, TX). It expects the associated parameters `u32Pgn`, `NmSource`, `u16DaSize`, and `u8Priority`. Confirmation or the result is reported via `installO`.

Event Outputs

- **`INITO`**: Confirms the completion of initialization (`INIT`).
- **`installO`**: Reports the completion of an installation request. Returns the `PGN_handle` (positive on success, `HANDLE_UNVALID` in case of error).
- **`IND`**: Triggered when new data is received for an installed PGN. Returns the received `Data` and a timestamp `s32TimeStamp`.
- **`dataERR`**: Triggered in case of an error during data reception. Returns an error code `dataERRC`.

- **pgnERR**: Triggered in case of an error related to PGN management (e.g., installation). Returns an error code pgnERRC.

Data Inputs

- **u32Pgn** (UDINT): The Parameter Group Number (PGN) to be installed or monitored. Valid range: 0 to 0x3FFFF.
- **NmSource** (isobus::pgn::ISONETEVENT_T): Defines the communication partner (e.g., a specific node address or a broadcast).
- **u16DaSize** (UINT): The expected data length of the PGN in bytes (0-8).
- **u8Priority** (USINT): The default priority of this PGN (0-7), where 0 represents the highest priority. The initial value is 7 (lowest priority).
-

Data Outputs

- **PGN_handle** (INT): A handle (reference number) for the successfully installed PGN. In case of an error, it contains the value HANDLE_UNVALID.
- **dataERRC** (INT): Error code set when the dataERR event is triggered.
- **pgnERRC** (INT): Error code set when the pgnERR event is triggered.
- **s32TimeStamp** (DINT): Timestamp of the received message in milliseconds. Initial value is −1.
- – **Data** (isobus::pgn::CAN_MSG): The buffer containing the received CAN message data.

Adapter

This function block does not use any adapter interfaces.

Operation

1. **Initialization**: The INIT event makes the function block operational. Upon completion, INITO is triggered.
2. **PGN Installation**: One or more PGNs can be registered for reception via the install event. The block configures internal filters in the CAN stack based on the passed parameters (PGN, source, length, priority). The result (handle or error) is reported back via installO.
3. **Data Reception**: As soon as a message arrives for an installed PGN, the IND event is triggered. The received data (Data) and a timestamp (s32TimeStamp) are then available at the outputs.
4. **Error Handling**: If errors occur during reception (dataERR) or PGN management (pgnERR), the corresponding events are triggered with an error code.

Technical Specifications

- This block is designed to receive PGNs with a data length of up to 8 bytes (hence "8B" in the name).
- It uses specific ISOBUS data types (isobus::pgn::CAN_MSG, isobus::pgn::ISONETEVENT_T) that require integration into an ISOBUS-specific software stack.
- Error handling is divided into two separate channels: dataERR for runtime errors during reception and pgnERR for configuration errors.

State Overview

The block has implicit states:

- **Not Initialized**: Before the first INIT event.
- **Ready**: After successful initialization (INITO). Can receive install requests.
- **Active/Receiving**: After successful installation of at least one PGN. Can trigger IND events for received data.
- **Error**: When an error occurs (pgnERR or dataERR). Depending on the implementation, the block may remain in the error state or become functional again after the cause is resolved.

Application Scenarios

- **Agricultural Machinery Control**: Receiving implement data (e.g., seed drills, fertilizer spreaders) in a tractor's ISOBUS network.
- **Trailer Communication**: Receiving status information (brake pressure, lights, door contacts) from a smart trailer.

- **Implement Monitoring**: Receiving real-time data from sensors on an implement (e.g., seed flow, working width).

⚖️ Comparison with Similar Blocks

- **Generic CAN Receive Blocks**: Unlike generic `CAN_RX` blocks, which process raw CAN IDs and data, `AlPgnRxNew8B` abstracts ISOBUS-specific PGN addressing and management.
- **`E_SR` (Service Interface Function Block)**: While `E_SR` provides a generic server request pattern for services, `AlPgnRxNew8B` is specifically designed for the cyclic or event-driven reception of ISOBUS messages.
- **Blocks for Other Protocols**: Similar blocks exist for other higher-level protocols (e.g., J1939), but they differ in their specific addressing and message formats.
- **`E_SR` (Service Interface Function Block)**: While `E_SR` provides a generic server request pattern for services, `AlPgnRxNew8B` is specifically designed for the cyclic or event-driven reception of ISOBUS messages.
- **Blocks for Other Protocols**: Similar blocks exist for other higher-level protocols (e.g., J1939), but they differ in their specific addressing and message formats.
-

🛠️ Related Exercises

- Exercise_130
- Exercise_134

Conclusion

The `AlPgnRxNew8B` is an essential building block for developing ISOBUS-compatible receiver applications in the 4diac IDE. It encapsulates the complexity of PGN filtering and management and provides the application with a clean, event-driven interface for receiving CAN messages according to the ISO 11783 standard. Its clear separation of configuration (`install`), data transfer (`IND`), and error handling makes it robust and easy to integrate into higher-level control logic.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

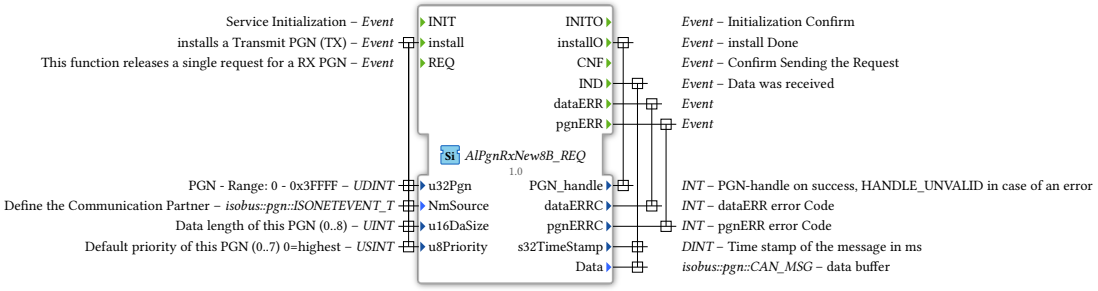
]

AlPgnRxNew8B_REQ

• – ○ • – ○ • – ○ •

Introduction

The function block `AlPgnRxNew8B_REQ` is used to request data via an ISOBUS network. It enables the installation of a receive parameter set (RX PGN) and the triggering of a one-time request for this parameter. The block is part of an ISOBUS-specific communication library and handles the interaction with the CAN network protocol according to ISO 11783.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block.
- **install**: Installs a receive parameter set (RX PGN) for subsequent data requests. Triggered with the data `u32Pgn`, `NmSource`, `u16DaSize`, and `u8Priority`.

- **REQ**: Triggers a one-time request for the previously installed RX PGN.

Event Outputs
- **INITO**: Confirms successful initialization.
- **installO**: Confirms successful installation of a PGN. Returns the assigned `PGN_handle`.
- **CNF**: Confirms successful transmission of the request to the network.
- **IND**: Triggered when the requested data message is received. Returns `Data` and `s32TimeStamp`.
- **dataERR**: Signals a data processing error. Returns the error code `dataERRC`.
- **pgnERR**: Signals an error in PGN processing (e.g., during installation). Returns the error code `pgnERRC`.

Data Inputs
- **u32Pgn** (UDINT): The parameter Group Number (PGN) in the range 0 to 0x3FFFF.
- **NmSource** (isobus::pgn::ISONETEVENT_T): Defines the communication partner in the network.
- **u16DaSize** (UINT): The expected data length of the PGN (0-8 bytes).
- **u8Priority** (USINT): The default priority of this PGN (0..7, where 0 is the highest priority). Initial value is 7.

Data Outputs
- **PGN_handle** (INT): A handle to identify the successfully installed PGN. In case of an error, an invalid handle value (`HANDLE_UNVALID`) is returned.
- **dataERRC** (INT): Error code set when the `dataERR` event is triggered.
- **pgnERRC** (INT): Error code set when the `pgnERR` event is triggered.
- **s32TimeStamp** (DINT): Timestamp of the received message in milliseconds. Initial value is −1.
- **Data** (isobus::pgn::CAN_MSG): The buffer containing the received CAN message data.

Adapter

This function block does not use any adapter interfaces.

Operation

The function block operates in two main phases: installation and request.

1. **Installation**: The `install` event configures the parameters for a PGN to be received (number, source, size, priority). The block registers this request in the underlying ISOBUS stack and returns a `PGN_handle` via `installO`. This handle is needed for later references.
2. **Request**: The `REQ` event triggers a one-time request for the most recently installed PGN on the network. After successful transmission of the request, `CNF` is triggered. When the requested message arrives from the defined `NmSource`, it is stored in the `Data` buffer and output along with a timestamp via the `IND` event.

Errors during installation (e.g., invalid PGN) result in the `pgnERR` output. Errors during data reception or processing result in the `dataERR` output.

Technical Specifications
- This block is designed to process PGNs with a data length of up to 8 bytes (`u16DaSize` 0..8).
- Priority management (`u8Priority`) follows the ISOBUS standard.
- Data is provided in a type-safe buffer (`CAN_MSG`).
- Error handling is structured via dedicated event outputs (`pgnERR`, `dataERR`).

Status Overview
1. **Not Initialized**: Before the first `INIT` event.
2. **Initialized / Ready**: After `INITO`. The block can receive `install` events.
3. **PGN Installed**: After successful `installO`. The block is ready to process `REQ` events.
4. **Request Pending**: After `REQ` and before `CNF`. The network request has been sent.
5. **Ready to Receive Data**: After `CNF`. The block is waiting for the incoming message and will trigger `IND` upon receipt.

Application Scenarios

Typical applications are in agricultural or mobile machinery control (ISO 11783 / ISOBUS):

- Querying machine parameters (e.g., speed, temperature) from an implement.
- Requesting configuration data once from a control unit on the network.

- Implementing diagnostic or service tools that need to query specific PGNs.

⚖️ Comparison with Similar Blocks

Compared to generic CAN receive blocks, `AlPgnRxNew8B_REQ` is specifically tailored to the ISOBUS protocol and its PGN concept. It abstracts the low-level CAN details (such as ID filtering) and provides a clean interface at the PGN level. Blocks like `E_CYCLE` or `E_DELAY` could be used to implement periodic requests, while `AlPgnRxNew8B_REQ` itself only handles single requests.

🛠️ Related Exercises

- Exercise_132

Conclusion

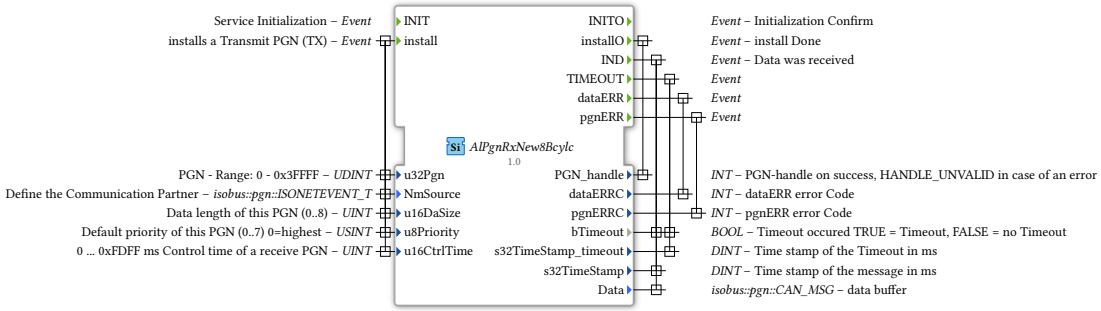
The `AlPgnRxNew8B_REQ` is a specialized function block for on-demand communication in ISOBUS networks. By separating installation (`install`) and execution (`REQ`), it enables flexible and controlled data requests. Its integrated error handling and type-safe data interface make it a robust component for demanding embedded controllers in agricultural technology.

AlPgnRxNew8Bcylc

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AlPgnRxNew8Bcylc` is used for the cyclical reception of data via an ISOBUS network according to the Parameter Group Number (PGN) protocol. It enables the configuration and monitoring of receive channels for specific PGNs, including the handling of timeouts and error conditions. The block is designed for use in control systems that require reliable and monitored communication with other ISOBUS devices.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block.
- **install**: Installs a new receive channel for a specific PGN. Triggers the configuration with the associated data inputs.

Event Outputs

- **INITO**: Confirms successful initialization.
- **installO**: Signals completion of an installation request. Returns the assigned `PGN_handle`.
- **IND**: Triggered when new data is received for the configured PGN.
- **TIMEOUT**: Signals that a timeout has occurred for the configured PGN.
- **dataERR**: Indicates an error in the received data.
- **pgnERR**: Indicates an error related to the PGN configuration or processing.

Data Inputs

- **u32Pgn** (UDINT): The parameter group number (PGN) to be received. Valid range: 0 to 0x3FFFF.
- **NmSource** (isobus::pgn::ISONETEVENT_T): Defines the communication partner (network management source).
- **u16DaSize** (UINT): The expected data size of the PGN in bytes (0..8).

- **u8Priority** (USINT): The default priority of this PGN (0..7, where 0 is the highest priority). Initial value: 7.
- **u16CtrlTime** (UINT): The control time (timeout time) for receiving this PGN in milliseconds (0 ... 0xFDFF ms). A value of 0 disables timeout monitoring.

Data Outputs

- **PGN_handle** (INT): A handle that identifies the successfully installed receive channel. In case of an error, `HANDLE_UNVALID` is output.
- **dataERRC** (INT): Error code set when the `dataERR` event is triggered.
- **pgnERRC** (INT): Error code set when the `pgnERR` event is triggered.
- **bTimeout** (BOOL): Indicates whether a timeout occurred (`TRUE` = timeout, `FALSE` = no timeout). Initial value: `FALSE`.
- **s32TimeStamp_timeout** (DINT): Timestamp of the timeout in milliseconds.
- **s32TimeStamp** (DINT): Timestamp of the last received message in milliseconds. Initial value: −1.
- **Data** (isobus::pgn::CAN_MSG): Buffer containing the received user data.

Adapters

This function block does not use adapters.

Operation

1. **Initialization**: The `INIT` event makes the function block operational. Confirmation is provided by `INITO`.
2. **Channel Configuration**: A receive channel is configured by the `install` event. The associated parameters (`u32Pgn`, `NmSource`, `u16DaSize`, `u8Priority`, `u16CtrlTime`) define the desired data stream. On successful completion, a unique `PGN_handle` event (`installO`) is returned. On failures, `pgnERR` or `dataERR` is triggered.
3. **Cyclic Reception**: Once a channel is installed, the block listens for messages from the configured PGN at the specified source. Upon receiving a valid message, the data is made available in the `Data` output, and the `IND` event is triggered. The `s32TimeStamp` event is then updated.
4. **Timeout Monitoring**: If `u16CtrlTime` > 0, the block monitors the time between received messages. If no new message is received within this time window, the `TIMEOUT` event is triggered, `bTimeout` is set to `TRUE`, and `s32TimeStamp_timeout` is updated.
5. **Error Handling**: Protocol errors or invalid configurations trigger the corresponding error events (`dataERR`, `pgnERR`) with their associated error codes.

Technical Features

- The block supports the specific data types `isobus::pgn::CAN_MSG` and `isobus::pgn::ISONETEVENT_T` from the ISOBUS PGN library.
- Timeout monitoring is optional and can be disabled by setting `u16CtrlTime` to 0.
- The `PGN_handle` serves as a reference for the installed channel and may be necessary for later management operations (e.g., uninstallation).

Status Overview

1. **Not Initialized**: After startup. Waiting for `INIT`.
2. **Ready**: After successful initialization (`INITO`). Can receive `install` requests.
3. **Ready to Receive (Channel Active)**: After successful installation of a channel (`installO`). Monitors the bus for the configured PGN and triggers `IND` upon receipt or `TIMEOUT` if the monitoring time is exceeded.
4. **Error State**: Signaled by the events `pgnERR` or `dataERR`. The exact error can be found in the respective error code. The block can usually be used again after the cause has been resolved.

Application Scenarios

- **Agricultural Machinery Control**: Receiving implement data (e.g., seed drill status) from an implement on the ISOBUS.
- **Vehicle Diagnostics**: Cyclic reception of engine data (RPM, temperature) for display and monitoring in the terminal.
- **Implement Control**: Receiving setpoint values (e.g., setpoint depth) from an operator console for a hydraulic lower link.

⚖️ Comparison with Similar Blocks

Unlike generic CAN receive blocks, `AlPgnRxNew8Bcylc` is specifically tailored to the ISOBUS PGN protocol. It offers higher abstraction through direct PGN addressing, integrated partner identification (`NmSource`), and automatic timeout management. Simple receive blocks would not provide this protocol layer and management functionality.

🛠️ Related Exercises

- Exercise_131

Conclusion

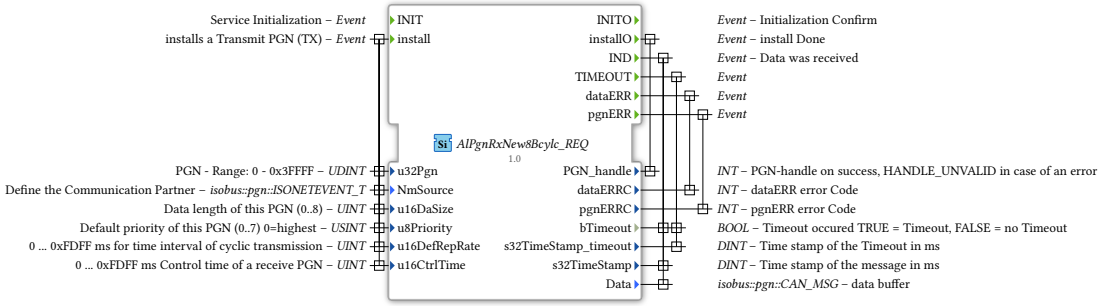
The `AlPgnRxNew8Bcylc` is an essential building block for robust ISOBUS applications that need to receive cyclic data from other network participants. Its integrated channel management, timing, and error detection features significantly simplify the development of reliable communication interfaces and relieve application developers of the burden of dealing with low-level protocol details.

AlPgnRxNew8Bcylc_REQ

• – ○ • – ○ • – ○ •

Introduction

The function block `AlPgnRxNew8Bcylc_REQ` is used for the cyclical request of data via a CAN network according to the ISOBUS standard (ISO 11783). Its main purpose is the installation and management of receive parameters for specific Parameter Group Numbers (PGNs). The block enables the configuration of cyclical receive and monitors the data flow by triggering corresponding events upon successful receive, timeouts, or errors.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block.
- **install**: Triggers the installation of a receive PGN (RX) with the associated configuration parameters. The data carried are: `u32Pgn`, `NmSource`, `u16DaSize`, `u8Priority`, `u16DefRepRate`, `u16CtrlTime`.

Event Outputs

- **INITO**: Confirms successful initialization.
- **installO**: Signals completion of the installation process. Includes `PGN_handle` as a result.
- **IND**: Triggered when new data is received. Includes `bTimeout`, `s32TimeStamp`, and `Data`. * **TIMEOUT**: Triggered on a control time timeout. Includes `bTimeout` and `s32TimeStamp_timeout`.
- **dataERR**: Indicates an error in the received data. Includes the error code `dataERRC`.
- **pgnERR**: Indicates an error in PGN processing. Includes the error code `pgnERRC`.

Data Inputs

- **u32Pgn** (UDINT): The parameter group number (PGN) to be received. Valid range: 0 to 0x3FFFF.
- **NmSource** (isobus::pgn::ISONETEVENT_T): Defines the communication partner (e.g., a specific node address).
- **u16DaSize** (UINT): The expected data size of the PGN in bytes (0..8).
- **u8Priority** (USINT): The default priority for this PGN (0..7, where 0 is the highest priority). Initial value: 7.

- **u16DefRepRate** (UINT): The expected cyclic transmission interval of the source PGN in milliseconds (0 … 0xFDFF ms).
- **u16CtrlTime** (UINT): The control time in milliseconds (0 … 0xFDFF ms), after which a `TIMEOUT` event is generated if no data is received.

Data Outputs

- **PGN_handle** (INT): A handle (identifier) for the successfully installed PGN. In case of an error, an invalid handle (HANDLE_UNVALID) is returned.
- **dataERRC** (INT): Error code returned for a `dataERR` event.
- **pgnERRC** (INT): Error code returned for a `pgnERR` event.
- **bTimeout** (BOOL): Status flag for a timeout. `TRUE` = timeout occurred, `FALSE` = no timeout. Initial value: `FALSE`.
- **s32TimeStamp_timeout** (DINT): Timestamp in milliseconds at which the timeout was detected.
- **s32TimeStamp** (DINT): Timestamp in milliseconds of the last received valid data record. Initial value: −1.
- **Data** (isobus::pgn::CAN_MSG): Buffer containing the received CAN message data.

Adapter

This function block does not use any adapter interfaces.

Operation

1. **Initialization**: The `INIT` event puts the block into a ready-to-use state, which is acknowledged by `INITO`.
2. **PGN Installation**: The `install` event configures a new receiving PGN. All necessary parameters (PGN, source, data length, etc.) are passed. Upon successful configuration, the block responds with `installO` and delivers a valid `PGN_handle`. Errors trigger either `pgnERR` or `dataERR`.
3. **Cyclic Reception**: After successful installation, the module monitors the CAN bus for messages of the configured PGN and source.
4. **Data Indication**: Upon receiving a valid message, the `IND` event is triggered. The data (`Data`), a timestamp (`s32TimeStamp`), and the timeout status (`bTimeout=FALSE`) are output.
5. **Timeout Monitoring**: If the time since the last received packet exceeds the configured `u16CtrlTime`, the `TIMEOUT` event is triggered. `bTimeout` is set to `TRUE`, and a timestamp (`s32TimeStamp_timeout`) is provided.
6. **Error Handling**: If protocol or data errors occur, the events `pgnERR` and `dataERR`, respectively, are generated with the corresponding error codes.

Technical Features

- This function block is specifically designed for use in ISOBUS environments (agricultural machinery) and utilizes type-safe data types from the `isobus::pgn` library (`CAN_MSG`, `ISONETEVENT_T`).
- Timeout monitoring (`u16CtrlTime`) is independent of the expected transmission interval (`u16DefRepRate`) and serves to enhance robustness by detecting failed communication partners.
- `PGN_handle` enables the unique identification and subsequent management (e.g., uninstallation) of a configured PGN instance within an application.

State Transition Overview

1. **Not Initialized**: After startup. Waiting for `INIT`.
2. **Ready**: After `INITO`. Can receive `install` requests.
3. **Installed**: After successful `installO`. Actively monitors the CAN bus for the configured PGN.

- On reception: Transitions to the “Data Available” state (triggers `IND`), then returns to “Installed”.
- On timeout: Triggers `TIMEOUT`, remains in the “Installed” state (continues monitoring).
- On error: Triggers `pgnERR`/`dataERR`, remains in the “Installed” state.

Application Scenarios

- **ISOBUS Implementations**: Receiving cyclic data (e.g., engine speed, pressure, position) from an electronic control unit (ECU) of an implement on the tractor.
- **Monitoring Functions**: Continuously checks whether a critical component (e.g., motor control) is still functioning and sending data (using `CtrlTime`).
- **Data Logger**: Cyclically collects process data from the CAN network for analysis or storage purposes.

⚖️ Comparison with Similar Blocks

- **E_CTU vs. AlPgnRxNew8Bcylc_REQ**: A simple counter (`E_CTU`) lacks network functionality. This block is a specialized, application-oriented communication block for a specific protocol (ISOBUS).
- **Generic CAN RX Blocks**: Unlike blocks that receive raw CAN IDs and data, this block operates at the higher, standardized PGN level of ISOBUS and handles protocol-specific decoding and parameter management.

🛠️ Related exercises

- Uebung_133

Conclusion

The `AlPgnRxNew8Bcylc_REQ`is an essential component for implementing ISOBUS-compliant receiver functionalities in 4diac. It abstracts the complexity of CAN communication and provides a clean, event-driven interface for reliable, cyclical data acquisition with integrated error and timeout detection. Its use increases the reusability and robustness of control applications in agricultural technology.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter module on ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

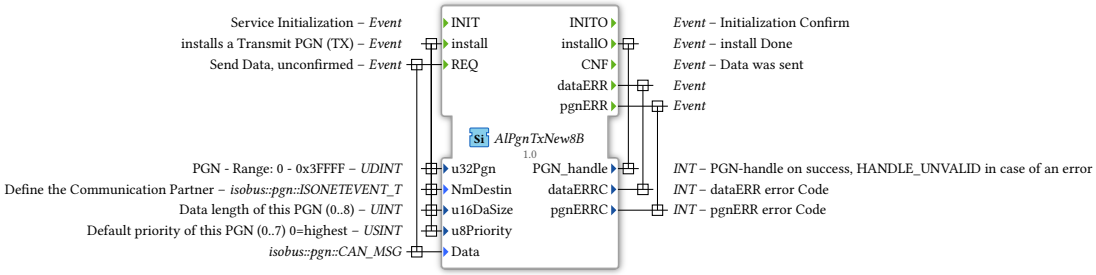
tx

AlPgnTxNew8B

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AlPgnTxNew8B` is used to send data over a CAN network according to the ISOBUS standard (ISO 11783). Its main purpose is to install and manage Parameter Group Numbers (PGNs) for transmission (TX) and subsequently send data packets as soon as a local transmit event (`REQ`) occurs. It is designed for applications that require unconfirmed data transmission.



Interface Structure

Event Inputs

- **`INIT`**: Starts the initialization of the function block.
- **`install`**: Triggers the installation of a new Transmit PGN (TX-PGN). The installation is configured using the data inputs linked via `With` (`u32Pgn`, `NmDestin`, `u16DaSize`, `u8Priority`).
- **`REQ`**: Triggers the transmission process for the previously installed PGN. The data to be transmitted is provided via the data input `Data`, which is linked to `With`.

Event Outputs

- **`INITO`**: Confirms successful completion of initialization (`INIT`).
- **`installO`**: Confirms completion of PGN installation. Returns a handle for the installed PGN or an error value via `PGN_handle`.
- **`CNF`**: Confirms that the data was sent successfully (response to `REQ`).
- **`dataERR`**: Indicates an error related to the data being sent (`Data`). Returns an error code via `dataERRC`.
- **`pgnERR`**: Indicates an error related to the PGN installation or usage. Returns an error code via `pgnERRC`.

Data Inputs

- **`u32Pgn`** (UDINT): The Parameter Group Number (PGN) to be installed. Valid range: 0 to 0x3FFFF (decimal 262143).
- **`NmDestin`** (isobus::pgn::ISONETEVENT_T): Defines the communication partner (e.g., broadcast, specific address).
- **`u16DaSize`** (UINT): The data length of the PGN in bytes. Valid range: 0 to 8.
- **`u8Priority`** (USINT): The priority of the message on the CAN bus. Range: 0 (highest) to 7 (lowest). Default value: 7.
- **`Data`** (isobus::pgn::CAN_MSG): The data to be sent, structured as a CAN message.

Data Outputs

- **`PGN_handle`** (INT): A handle (identifier) for the successfully installed PGN. In case of an error, the value `HANDLE_UNVALID` is output.
- **`dataERRC`** (INT): Numeric error code that is set when the `dataERR` event is triggered.
- **`pgnERRC`** (INT): Numeric error code set when the `pgnERR` event is triggered.

Adapter

This function block does not use any adapter interfaces.

Operation

1. **Initialization**: The `INIT` event puts the function block into an operational state. Confirmation is provided by `INITO`.
2. **PGN Installation**: The `install` event configures a specific PGN for data transmission. This sets the PGN number, destination address, data length, and priority. Upon success, `PGN_handle` is returned via `installO`. In case of an error (e.g., invalid parameters), the event `pgnERR` is triggered.
3. **Data Transmission**: As soon as a `REQ` event occurs, the block attempts to send the CAN message provided at the `Data` input using the previously installed PGN configuration. If successful, the `CNF` event is triggered. If an error occurs during transmission (e.g., invalid data), `dataERR` is triggered.

Technical Specifications

- The block is designed for transmitting data packets with a maximum length of 8 bytes (typical for classic CAN frames).
- It implements unacknowledged sending: A `CNF` error simply means that the message has been passed to the lower protocol layer for transmission, not necessarily that it has reached the receiver.
- Error handling is divided into two categories: PGN-related errors (`pgnERR`) and data-related errors (`dataERR`), which simplifies error diagnosis.
- The block uses specific ISOBUS data types (`isobus::pgn::CAN_MSG`, `isobus::pgn::ISONETEVENT_T`).

State Overview

The block implicitly goes through the following main states:

1. **Not Initialized**: After startup. Only the `INIT` input is active.
2. **Initialized / Ready**: After `INITO`. The block can now install PGNs (`install`).
3. **PGN Installed**: After successful `installO`. The block is ready to send data (`REQ`).
4. **Send Active**: During the processing of `REQ`. Transition to state 3 after `CNF` or to an error state after `dataERR`/`pgnERR`.

Application Scenarios

- **Agricultural Machinery Control (ISOBUS)**: Sending machine data (e.g., working speed, PTO speed) to a terminal or other control units in the tractor.
- **Commercial Vehicle Communication**: Transmission of vehicle data (e.g., torque, fuel consumption) within a truck bus system. * **Industrial Automation**: Sending control commands or status information via CAN networks in machines.

⚖️ Comparison with similar blocks

- **`E_SEND` (Standard 61499)**: A generic transmit block. `AlPgnTxNew8B` is specialized for ISOBUS/CAN with PGN management, prioritization, and destination addressing, while `E_SEND` is protocol-agnostic and requires an adapter connection.

- **Acknowledged Transmit Blocks**: `AlPgnTxNew8B` transmits unacknowledged (`CNF`). For acknowledged communication (request/response), other, more specific TX PGN blocks or protocol stacks would be required.

🛠️ Related Exercises

- Exercise_124
- Exercise_128

Conclusion

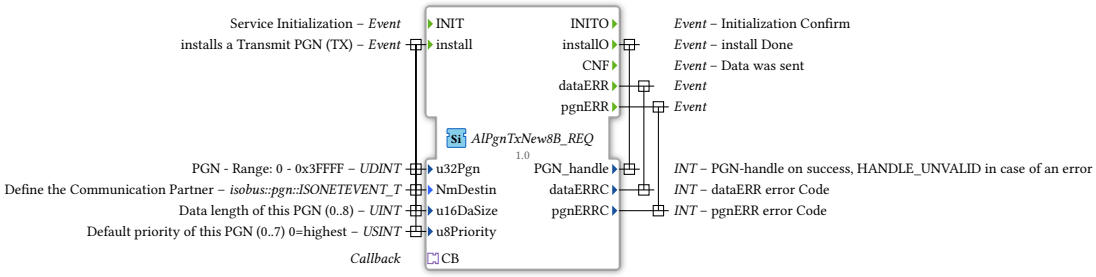
The `AlPgnTxNew8B` is a specialized function block for reliable, configurable, and unacknowledged data transmission in ISOBUS environments. Its clear separation of configuration (`install`) and operation (`REQ`), as well as its differentiated error feedback, makes it easily maintainable and simple to integrate into higher-level application logic. It is the first choice when CAN messages need to be sent according to the ISOBUS standard with a fixed PGN.

AlPgnTxNew8B_REQ

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AlPgnTxNew8B_REQ` is used to configure and control the transmission of parameter group numbers (PGNs) in an ISOBUS network (J1939). Its main purpose is to install a new transmit PGN (TX-PGN) in the network. The block responds to a REQ event (J1939 network event) and provides a callback interface for asynchronous feedback on the data transmission status. It is part of a specialized library for ISOBUS communication.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block. Must be triggered before first use.
- **install**: Triggers the installation of a new Transmit PGN (TX). The installation is configured with the data parameters bound to this event (`u32Pgn`, `NmDestin`, `u16DaSize`, `u8Priority`).

Event Outputs

- **INITO**: Confirms successful initialization of the function block.
- **installO**: Signals completion of the installation request. Returns the assigned `PGN_handle` or, in case of an error, an invalid handle.
- **CNF**: Confirms that data was successfully sent. Typically triggered via the callback adapter.
- **dataERR**: Indicates an error related to the data being sent. Returns an error code via `dataERRC`.
- **pgnERR**: Indicates an error related to PGN configuration or management. Returns an error code via `pgnERRC`.

Data Inputs

- **u32Pgn** (UDINT): The parameter group number (PGN) to be installed. Valid range: 0 to 0x3FFFF (decimal 262143).
- **NmDestin** (isobus::pgn::ISONETEVENT_T): Defines the communication partner (destination address) for the PGN. The exact type is defined by the ISOBUS library.
- **u16DaSize** (UINT): The length of the payload for this PGN in bytes. Valid range: 0 to 8.
- **u8Priority** (USINT): The default priority for this PGN. Valid range: 0 (highest) to 7 (lowest). Initial value: 7.

Data Outputs

- **PGN_handle** (INT): A handle (identifier) for the installed PGN. If the installation is successful, it contains a valid, positive value. In case of an error, an invalid handle (e.g., `HANDLE_UNVALID`) is output.
- **dataERRC** (INT): The specific error code when a `dataERR` event occurs.
- **pgnERRC** (INT): The specific error code when a `pgnERR` event occurs.

Adapter

- **CB** (Type: `isobus::pgn::tx::Callback`): A socket adapter that provides a callback interface. Asynchronous feedback (such as the `CNF` event) is reported from the underlying ISOBUS driver system to the function block via this adapter. The function block must be connected to a corresponding plug adapter to receive the feedback.

Operation

1. **Initialization**: First, the `INIT` event must be triggered to make the function block operational. This results in the output of `INITO`.
2. **PGN Installation**: The `install` event starts the configuration process. The function block passes the parameters `u32Pgn`, `NmDestin`, `u16DaSize`, and `u8Priority` to the ISOBUS protocol stack to register a new transmit PGN.
3. **Feedback**: After processing the installation request, the function block responds with the `installO` event and returns `PGN_handle`. This handle must be used for subsequent transmit operations (e.g., with a separate transmit function block).
4. **Asynchronous Operational Feedback**: During operation, when data is sent for the configured PGN, the acknowledgment (`CNF`) or error message (`dataERR`, `pgnERR`) is sent asynchronously via the respective event outputs. The triggering of the `CNF` event is typically controlled by the callback adapter (`CB`).

Technical Features

- This module is specific to ISOBUS/J1939 networks and uses type-safe data structures (`isobus::pgn::ISONETEVENT_T`).
- Error handling is divided into `dataERR` (data error) and `pgnERR` (PGN configuration error), enabling precise error diagnosis.
- Communication with the actual network stack occurs via a callback interface (`CB` adapter), which allows for loose coupling and asynchronous notification.
- `PGN_handle` serves as an abstract reference to the internally managed PGN instance.

State Overview

The component implicitly passes through the following main states:

1. **Not Initialized**: Before the first `INIT` event.
2. **Ready**: After successful initialization (`INITO` received). The function block can now receive `install` requests.
3. **Installation in progress**: After triggering `install`, the function block waits for a response from the protocol stack.
4. **Operational**: After successful `installO`, the PGN is registered in the stack and can be used for transmission. Asynchronous events (`CNF`, `dataERR`, `pgnERR`) can now occur.

Application Scenarios

- **Agricultural Machinery Control**: Integrating a new control unit into an ISOBUS network that needs to regularly send operating data (e.g., engine speed, temperature).
- **Implements**: Dynamic configuration of communication between a tractor and a specific implement that requires an individual PGN for its data.
- **Diagnostic Tools**: A diagnostic tool that temporarily installs a PGN on the network to request specific data or send commands.

⚖️ Comparison with Similar Blocks

- **`E_SEND` / `E_RCV` (Standard 61499)**: These generic communication blocks are protocol-independent. In contrast, `AlPgnTxNew8B_REQ` is specialized for ISOBUS/J1939 and handles the protocol-specific configuration (PGN, priority, destination address) that would have to be manually defined in the data parameters of the standard blocks.

- **Simple TX Blocks**: Other ISOBUS transmit blocks often expect a pre-configured `PGN_handle`. `AlPgnTxNew8B_REQ` is the upstream component that provides precisely this handle by installing a new PGN.

🛠️ Related Exercises

- Exercise_125

Conclusion

The `AlPgnTxNew8B_REQ` is an essential configuration component for ISOBUS communication in 4diac. It abstracts the complex setup of a transmitting PGN in a J1939 network behind a clear, event-driven interface. It separates installation (`install`/`installO`) from operation (`CNF`/`*ERR`).The use of a callback adapter makes it robust and easily integrated into larger control applications for mobile machine communication.

These features, along with the ability to use a callback adapter, make it robust and well-suited for integration into larger control applications for mobile machine communication.—

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

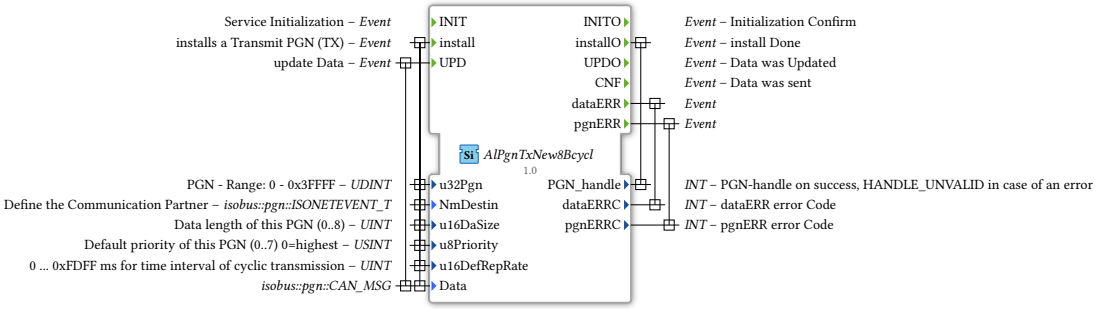
]

AlPgnTxNew8Bcycl

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AlPgnTxNew8Bcycl` is used for the cyclic transmission of data over a CAN network according to the ISOBUS standard (ISO 11783). Its main purpose is the installation, configuration, and regular transmission of Parameter Group Numbers (PGNs). It allows the definition of communication properties such as destination address, priority, and transmission interval, and ensures that data is transmitted reliably and at the configured frequency.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block.
- **install**: Installs a new PGN (TX) to be transmitted, along with its associated configuration data. Triggers the processing of the included data inputs (`u32Pgn`, `NmDestin`, `u16DaSize`, `u8Priority`, `u16DefRepRate`, `Data`).
- **UPD**: Updates the payload (`Data`) for the previously installed PGN to use it in the next cyclic transmission.

Event Outputs

- **INITO**: Confirms completion of initialization.
- **installO**: Signals that the installation process is complete. Includes `PGN_handle` as an output value.
- **UPDO**: Confirms that the payload update (`UPD`) was processed successfully.
- **CNF**: Confirms that a data record was sent successfully.
- **dataERR**: Indicates an error related to the payload (`Data`). It carries the error code `dataERRC`.
- **pgnERR**: Indicates an error related to the PGN configuration or management. It carries the error code `pgnERRC`.

Data Inputs

- **u32Pgn** (UDINT): The parameter group number (PGN) to be sent. Valid range: 0 to 0x3FFFF.
- **NmDestin** (isobus::pgn::ISONETEVENT_T): Defines the communication partner (destination address) for the message.
- **u16DaSize** (UINT): The length of the data to be sent in bytes. Valid range: 0 to 8.
- **u8Priority** (USINT): The priority of the message on the CAN bus (0 = highest, 7 = lowest). Default value: 7.
- **u16DefRepRate** (UINT): The cyclic transmission interval in milliseconds (0 … 0xFDFF ms). A value of 0 disables cyclic transmission. Default value: 0.
- **Data** (isobus::pgn::CAN_MSG): The payload to be sent as a PGN.

Data Outputs

- **PGN_handle** (INT): A handle (identifier) for the successfully installed PGN. In case of an error, an invalid handle value (`HANDLE_UNVALID`) is returned.
- **dataERRC** (INT): An error code providing detailed information about a `dataERR` event.
- **pgnERRC** (INT): An error code providing detailed information about a `pgnERR` event.

Adapter

This function block does not use any adapter interfaces.

Operation

1. **Initialization**: After the `INIT` event arrives, the block is made operational. Acknowledgement is provided via `INITO`.
2. **Installation**: The `install` event triggers the configuration of a new cyclic PGN transfer. All associated parameters (`u32Pgn`, `NmDestin`, etc.) are evaluated and stored internally. If successful, a `PGN_handle` is generated and returned with the `installO` event. Errors (e.g., invalid parameters) trigger the `pgnERR` event.
3. **Cyclic Transmission**: If the PGN is installed and `u16DefRepRate` > 0, the block automatically begins sending the payload data stored in `Data` at the defined interval. Each successful transmission is acknowledged by the `CNF` event.
4. **Data Update**: The `UPD` event allows you to change the payload data (`Data`) for the active PGN transmission. The new data will be used in the next cyclic transmission. Receipt of the new data is confirmed with `UPDO`. Incorrect data triggers `dataERR`.

Technical Features

- This block is designed for use in ISOBUS environments (agricultural machinery) and uses specific ISOBUS data types (`ISONETEVENT_T`, `CAN_MSG`).
- Cyclical transmission can be deactivated by setting `u16DefRepRate` to 0, enabling on-demand operation.
- – Error handling is structured via dedicated events (`dataERR`, `pgnERR`), enabling robust integration into higher-level controllers.
- Returning a `PGN_handle` event allows the management of multiple installed PGNs within a single system.

State Overview

1. **Not Initialized**: The block is in a sleep state after startup.
2. **Initialized / Ready**: After `INIT`/`INITO`. The block is waiting for configuration or control events.
3. **PGN Installed**: After successful `install`/`installO`. The PGN is configured, and the internal state for (cyclic) transmission is active.
4. **Send Active**: If `u16DefRepRate` > 0, the block sends data cyclically and fires `CNF` on each successful transmission.
5. **Error State**: When an error occurs (`pgnERR` or `dataERR`). Depending on the error type, the block may be in a waiting state or attempt to re-execute the operation (if implemented in the algorithm).

Application Scenarios

- **ISOBUS-compliant Machine Control**: Cyclic transmission of machine data (e.g., speed, pressure, position) from an electronic control unit (ECU) to a terminal or other network participants.
- **Diagnostic and Monitoring Systems**: Regular transmission of status and operating parameters for monitoring purposes.

- **Implementation of ISOBUS "Fast Packet" protocols**: For PGNs that carry more than 8 bytes of data and therefore need to be transmitted in multiple CAN messages (supported by the `CAN_MSG` data type).

⚖️ Comparison with similar blocks

- **Compared to simple `E_CYC` blocks**: `AlPgnTxNew8Bcycl` is specialized for ISOBUS PGNs and offers integrated handling of priority, target addressing, and error management, while a generic cyclic event generator (`E_CYC`) only provides timing.
- **Compared to generic CAN transmit blocks**: This block abstracts the low-level CAN details (identifier calculation, data frames) and operates directly at the more logical PGN level according to the ISOBUS standard.

🛠️ Related Exercises

- Exercise_127

Conclusion

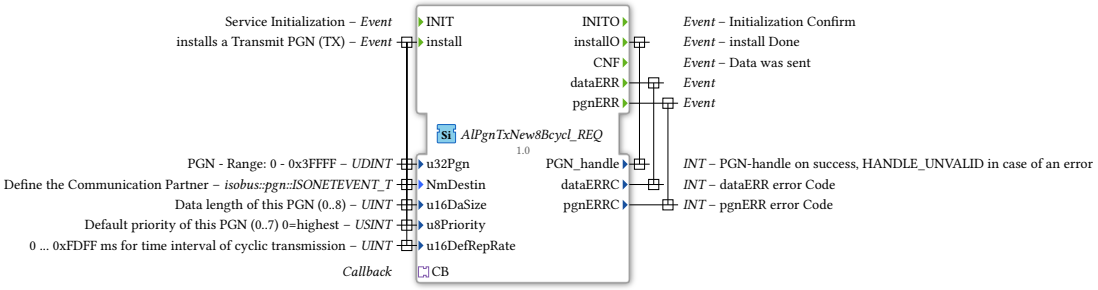
The `AlPgnTxNew8Bcyclis` a specialized and powerful function block for cyclic data communication in ISOBUS networks. Its clear interface, comprehensive configurability, and integrated error feedback make it ideally suited for the reliable implementation of transmission services in complex, distributed control systems for agricultural machinery.

AlPgnTxNew8Bcycl_REQ

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AlPgnTxNew8Bcycl_REQ` is used for the cyclic transmission of data over an ISOBUS network. Its main purpose is the installation and management of a Parameter Group Number (PGN) transmit object (TX) that sends data at a defined time interval. A key feature is the integration of a callback adapter, which enables flexible data provisioning.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block.
- **install**: Starts the installation process for a new cyclic transmit PGN. Triggers the configuration with the incoming data.

Event Outputs

- **INITO**: Confirms successful completion of initialization.
- **installO**: Confirms completion of PGN installation. Returns the generated `PGN_handle`.
- **CNF**: Triggered when data has been successfully sent.
- **dataERR**: Indicates an error related to the data being sent. Returns an error code.
- **pgnERR**: Indicates an error related to PGN configuration or management. Returns an error code.

Data Inputs

- **u32Pgn** (UDINT): The parameter Group Number (PGN) to be used. Valid range: 0 to 0x3FFFF.
- **NmDestin** (isobus::pgn::ISONETEVENT_T): Defines the communication partner (destination address) for the transmission.
- **u16DaSize** (UINT): The length of the payload to be transmitted in bytes. Maximum 8 bytes (0..8).

- **u8Priority** (USINT): The priority of the message on the bus (0..7), where 0 is the highest priority. Default value: 7 (lowest).
- **u16DefRepRate** (UINT): The cyclic transmission interval in milliseconds (0 … 0xFDFF ms). A value of 0 disables cyclic transmission. Default value: 0.

Data Outputs

- **PGN_handle** (INT): A unique handle that identifies the installed PGN instance. In case of an error, it contains the value `HANDLE_UNVALID`.
- **dataERRC** (INT): Error code that is set when the `dataERR` event is triggered.
- **pgnERRC** (INT): Error code set when the `pgnERR` event is triggered.

Adapter

- **CB** (Type: `isobus::pgn::tx::Callback`): A socket adapter that provides a callback interface. The function block uses this adapter to request the current payload from the higher-level control algorithm during each cyclic transmission.

Functionality

1. **Initialization**: The `INIT` event prepares the function block for operation. Upon completion, `INITO` is triggered.
2. **PGN Installation**: The `install` event triggers the configuration of a new cyclic transmission PGN. The values present at the data inputs (`u32Pgn`, `NmDestin`, etc.) are used to register the PGN in the ISOBUS stack.
3. **Handle Return**: Upon successful installation, the `installO` event is triggered, and the generated `PGN_handle` is made available at the data output. This handle must be saved for later operations (e.g., uninstallation, modification).
4. **Cyclic Send Operation**: If `u16DefRepRate` > 0, the function block begins sending data at the defined interval.

- Before each send operation, the block requests the current payload data via the `CB` adapter.
- After successful transmission, the `CNF` event is triggered.

1. **Error Handling**: If an error occurs (e.g., invalid configuration, communication problem), either `dataERR` or `pgnERR` is triggered and the corresponding error code is set.

Technical Features

- **Data Length**: Supports the transmission of a maximum of 8 bytes of user data per PGN, which corresponds to a typical ISOBUS data length.
- **Callback Mechanism**: The user data is not stored internally but is dynamically requested via the adapter as needed. This enables efficient and up-to-date data provision.
- **Error Handling**: Separate error events for data-related (`dataERR`) and PGN-related (`pgnERR`) problems allow for differentiated error diagnosis.
- **Initial Value**: The priority (`u8Priority`) and the transmission interval (`u16DefRepRate`) have defined initial values (7 and 0, respectively).

State Overview

The function block implicitly goes through the following main states:

1. **Not Initialized**: The block is inactive after startup.
2. **Initialized (Ready)**: After successful execution of `INIT`/`INITO`, the block waits for an installation request.
3. **PGN Installed (Active)**: After successful execution of `install`/`installO`, the PGN is configured. If `u16DefRepRate` > 0, the block sends data cyclically, triggering `CNF`. It also responds to error conditions.

Application Scenarios

- **Cyclic Status Messages**: Regular transmission of machine status data (e.g., speed, temperature, operating hours) to a display or a higher-level management system.
- **Implementation of ISOBUS "Fast Packet" Protocols**: For PGNs that contain more than 8 bytes of data and are distributed across multiple CAN telegrams, this block can control the cyclic transmission of individual packets.
- **Data Logging**: Cyclic transmission of process data to a data logger or gateway.

⚖️ Comparison with Similar Blocks

- **Vs. One-time transmit blocks (e.g., `AlPgnTx_REQ`): This block is designed for repeated, periodic transmitting, whereas simple TX blocks typically perform a single transmit per trigger event.

Vs. Blocks with Internal Data Storage: The use of a callback adapter distinguishes this block from those where the data is directly provided at an input. This makes it more flexible when the data changes frequently or originates from complex sources.

-

🛠️ Related Exercises

- Exercise_126
- Exercise_126b
- Exercise_126b2

Conclusion

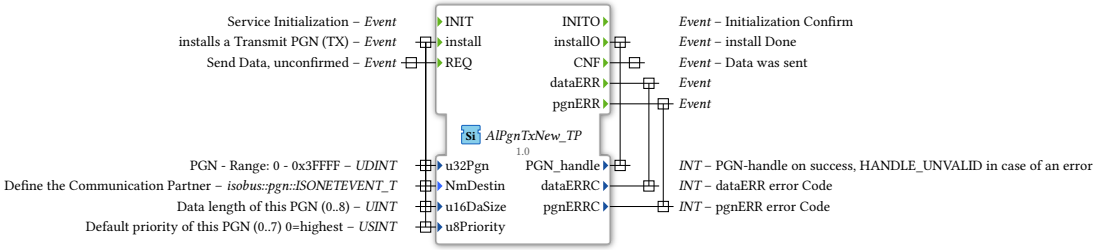
The `AlPgnTxNew8Bcycl_REQ` is a specialized function block for reliable, cyclical data communication in ISOBUS environments. Its strengths lie in the clear separation of configuration (`install`), dynamic data acquisition (callback adapter), and robust error feedback. It is ideally suited for applications that require regular status updates or process data streams.

AlPgnTxNew_TP

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AlPgnTxNew_TP** is used to transmit data over the ISOBUS/CAN bus based on Parameter Group Numbers (PGN). It enables the registration of a specific PGN and the subsequent transmission of data packets whenever a local request event (REQ) occurs. The block is part of the `isobus::pgn::tx` package and is specifically designed for handling transport protocols or generic PGN transmissions.



Interface Structure

Event Inputs

- **INIT** (Type: `EInit`): Initializes the service. Sets the block to its initial state.
- **install** (Type: `Event`): Installs or registers a transmit PGN. Parameters such as PGN number, destination, data size, and priority are adopted.
- **REQ** (Type: `Event`): Requests data transmission (unacknowledged transmission). Uses the buffer provided in input `Data`.

Event Outputs

- **INITO** (Type: `EInit`): Acknowledges initialization.
- **installO** (Type: `Event`): Acknowledges that the installation process has completed. Returns `PGN_handle`.
- **CNF** (Type: `Event`): Confirmation that the data was sent successfully.
- **dataERR** (Type: `Event`): Signals an error during data transmission or access to the data.
- **pgnERR** (Type: `Event`): Signals an error regarding the PGN configuration or registration.

Data Inputs

- **u32Pgn** (Type: `UDINT`): The parameter group number (PGN) to be sent. The range is between 0 and 0x3FFFF.

- **NmDestin** (Type: `isobus::pgn::ISONETEVENT_T`): Defines the communication partner or the destination in the network (Network Management Destination).
- **u16DaSize** (Type: `UINT`): The length of the data for this PGN in bytes (typically 0 to 8 for standard CAN, but can vary depending on the protocol).
- **u8Priority** (Type: `USINT`): The default priority of this PGN (0 to 7), where 0 is the highest priority. The default value is 7.

Data Outputs

- **PGN_handle** (Type: `INT`): Return value after installation. A valid handle on success, or `HANDLE_UNVALID` in case of failure.
- **dataERRC** (Type: `INT`): Error code output when the event `dataERR` is triggered. * **pgnERRC** (Type: `INT`): Error code output when the event `pgnERR` is triggered.

InOut Variables

- **Data** (Type: `BYTE`, ArraySize: *): A pointer to the data buffer (variable-size byte array). The payload data to be sent is stored here. Declaring it as `InOut` enables efficient memory management without unnecessary copy operations.

Adapters

- This function block does not use any explicit adapter interfaces.

Functionality

The module operates in two main phases:

1. **Configuration (Installation):**

- First, the module must be initialized via `INIT`.
- Then, the PGN is configured via the event `install`. This process defines the PGN ID (`u32Pgn`), the target (`NmDestin`), the size (`u16DaSize`), and the priority (`u8Priority`).
- After successful installation, the event `installO` is triggered, and a `PGN_handle` handle is provided. This handle represents the registered PGN in the system.

1. **Data Transfer:**

- Data must be provided in the `Data` array before it can be sent.
- The `REQ` event triggers the sending process.
- `CNF` is triggered upon successful transmission.
- If problems occur (e.g., invalid data length or bus error), `dataERR` is triggered with the corresponding error code `dataERRC`.

Technical Specifications

- **Variable Array Size:** The `Data` input is defined as the `InOut` variable with `ArraySize="*"`. This means that the function block can work with data buffers of varying sizes, making it flexible for different PGN types.
- **Error Separation:** The function block explicitly distinguishes between errors during PGN setup (`pgnERR`) and errors during runtime data transmission (`dataERR`). This significantly simplifies debugging.
- **ISOBUS Types:** The function block imports specific ISOBUS data types (`isobus::pgn::ISONETEVENT_T`), which requires tight integration into corresponding ISOBUS stacks.

State Overview

The function block has internally implicit states defined by the availability of a valid `PGN_handle`:

- **Not Initialized:** Before `INIT`.
- **Initialized, Not Installed:** After `INIT`, but before successful `install`. Send requests (`REQ`) would fail here.
- **Installed / Ready for Operation:** A valid handle has been created. Data can be sent via `REQ`.

Application Scenarios

- **ISOBUS Device Development:** Implementation of virtual terminals or task controllers that need to send specific PGNs.

- **Sensor Data Transmission:** A sensor module sends measured values to the CAN bus cyclically or based on events.
- **Control Commands:** Sending control messages to actuators in the network (e.g., valves or motors) with a defined priority.

⚖️ Comparison with Similar Function Blocks

- **AlPgnTx vs. AlPgnTxNew_TP:** While older or simpler versions may only support static PGNs, the "New_TP" suffix indicates a revised version that may offer better support for transport protocols (TP) or utilize more modern memory management (via `InOut` variables).
- **Standard CAN_WRITE:** Compared to a generic `CAN_WRITE` function block, `AlPgnTxNew_TP` abstracts the complexity of PGN management (priority, data length, handle management) and is more specifically tailored to the J1939/ISOBUS protocol.
-

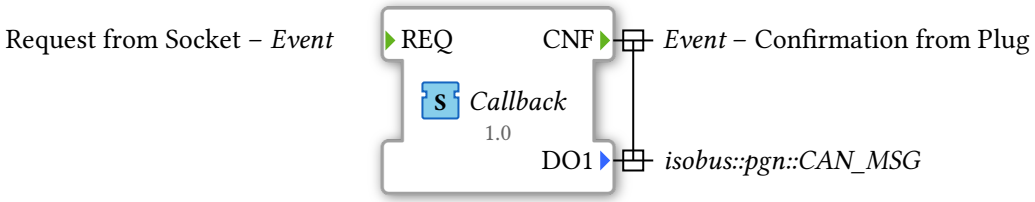
🛠️ Related Exercises

- Exercise_128b

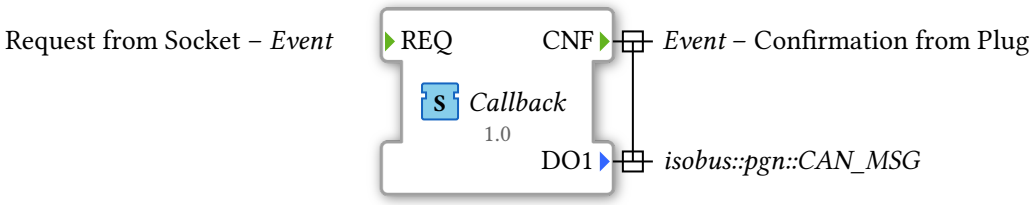
Conclusion

The **AlPgnTxNew_TP** is a specialized and robust function block for sending messages in ISOBUS networks. By separating configuration (`install`) and transmission (`REQ`) and utilizing reference data buffers (`InOut`), it is ideally suited for resource-efficient control applications that require precise control over PGN parameters.

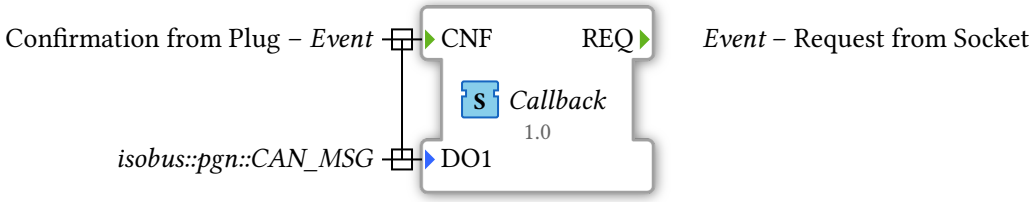
Callback



Plug (adapter output)



Socket (adapter input)



Message callback

Interface

Event inputs

Name	Comment	With
REQ	Request from Socket	

Event Outputs

Name	Comment	With
CNF	Confirmation from Plug	DO1

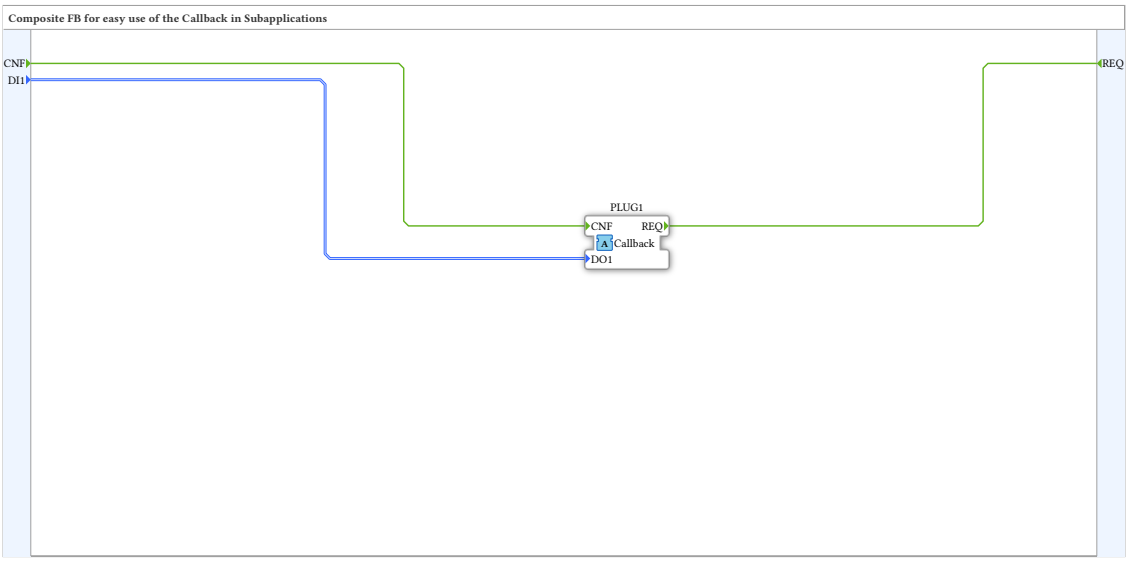
Output Vars

Name	Type	Comment
DO1	isobus::pgn::CAN_MSG	

🛠️ Related exercises

- SIN_TO_PCAN_Callback_DWord
- SIN_TO_PCAN_Callback_Byte
- Uebung_12x_sub

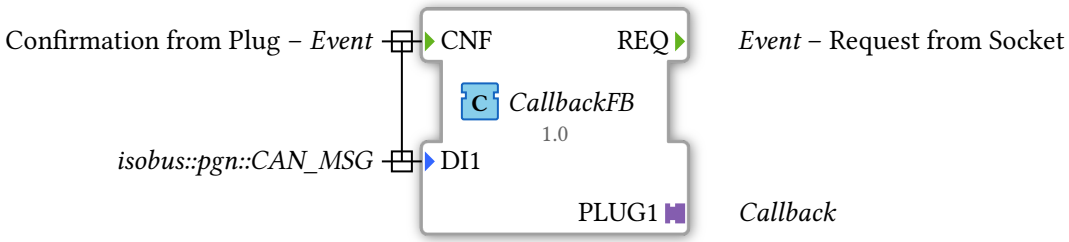
CallbackFB



• – ○ • – ○ • – ○ •

Introduction

The `CallbackFB` is a composite function block that simplifies the use of the callback mechanism in subapplications. It serves as a wrapper block to provide a standardized interface for asynchronous communication via a callback adapter. Its primary purpose is to decouple and provide structured handling of acknowledgment messages (`CNF`) and requests (`REQ`) within a function block network.



Interface Structure

Event Inputs

- **`CNF`**: Confirmation Event. This event is received by the connected plug and signals the completion of a request or the arrival of data.

Event Outputs

- **`REQ`**: Request event. This event is triggered by the adapter's internal socket and signals a pending action or data request.

Data Inputs

- **`DI1`**: Data input of type `isobus::pgn::CAN_MSG`. This input carries the CAN message data associated with the incoming `CNF` event.

Data Outputs

- *(No direct data outputs are defined on the function block. The data is forwarded via the adapter.)*

Adapters

- **`PLUG1`**: A plug adapter of type `isobus::pgn::tx::Callback`. This adapter establishes the connection to a corresponding socket adapter in another function block and implements the callback pattern.

Functionality

The `CallbackFB` acts as a transparent intermediary. Its internal network (FBNetwork) connects the public interfaces of the composite function block directly to the pins of the integrated `Callback` adapter (`PLUG1`).

1. An incoming `CNF` event is forwarded directly to the `CNF` input of the `PLUG1` adapter.
2. The data associated with `CNF` (`DI1` of type `isobus::pgn::CAN_MSG`) is simultaneously transferred to the `DO1` data input of the `PLUG1` adapter.
3. When the internal socket of the `PLUG1` adapter triggers a `REQ` event, it is directly forwarded to the public `REQ` event output of the `CallbackFB`.

The module itself does not perform any additional logic or data processing. Its function consists solely of the structured bundling and forwarding of events and data.

Technical Features

- **Typing**: The data input `DI1` uses the specific data type `isobus::pgn::CAN_MSG`, indicating an application in the context of ISOBUS and CAN-based agricultural systems.
- **Adapter Pattern**: All functionality is based on the 61499 adapter concept. The function block (FB) encapsulates a `Plug` and requires a corresponding `Socket` of type `isobus::pgn::tx::Callback` in another function block for operation.
- **Transparency**: As a composite FB without additional algorithms, its behavior is completely defined by the internal connection network and is therefore predictable.

State Overview

Since it is a purely pass-through composite function block without an internal state machine (Basic FB) or cyclic behavior (Service Interface FB), the `CallbackFB` does not have its own state machine. Its "state" is determined by the state of the connected adapter pair (`PLUG1` and its corresponding socket).

Application Scenarios

- **Structuring Subapplications**: In complex subapplications, the `CallbackFB` can be used to provide a clean and reusable interface for callback communication.
- **Decoupling Components**: It enables loose coupling between a service provider (providing the socket) and a service user (using this function block), which improves maintainability and testability.
- **ISOBUS Communication**: A specific application in control systems for agricultural machinery to asynchronously acknowledge or request the transmission of CAN messages (PGNs).
-

⚖️ Comparison with Similar Building Blocks

- **Direct Adapter Connection**: Instead of directly integrating a `Callback` adapter into a network, the `CallbackFB` offers a bundled, named interface (`CNF`/`REQ`/`DI1`) that may be more readable in some designs.
- **Simple E_CLASS vs. CallbackFB**: A simple E_CLASS building block (Event Splitter/Merger) could perform similar event routing, but would not provide the type-safe data connection for `isobus::pgn::CAN_MSG` along with the standardized adapter interface.
- **Simple E_CLASS vs. CallbackFB**: * **Generic Wrapper Function Blocks**: Unlike generic wrapper function blocks, `CallbackFB` is specialized for a specific data type and adapter, ensuring type safety and clear semantics.

🛠️ Related Exercises

- Exercise_126b2_sub
- Exercise_126b_sub
- Exercise_12x_sub

Conclusion

CallbackFB` is a useful auxiliary construct for the structured application of the adapter pattern in IEC 61499, especially in the ISOBUS environment. It reduces complexity in the main network by offloading the connection logic for callback communication into a single, reusable composite component. Its strength lies in its simplicity and clarity, as it does not implement any active logic itself. It is the ideal choice when a standardized and type-safe callback interface is required in a subapplication.

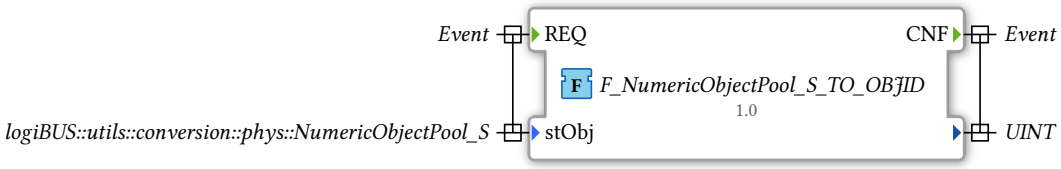
2.10.2.6 Utils

Conversion

Numeric Index (conversion)

Here you will find all exercises from the `conversion` workspace, listed in ascending order. ...

F_NumericObjectPool_S_TO_OBJID



• – ◦ • – ◦ • – ◦ •

Introduction

`F_NumericObjectPool_S_TO_OBJID` extracts the ISOBUS object ID (`u16ObjId`) from a `NumericObjectPool_S` structure (`logiBUS::utils::conversion::phys::NumericObjectPool_S`).

Interface Structure

Event Inputs

- **REQ**: execution request (associated with `stObj`)

Event Outputs

- **CNF**: confirmation (associated with the function result)

Data Inputs

- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): object pool structure of a VT number variable

Data Outputs

- **(function result)** (UINT): the object ID `stObj.u16ObjId`

Adapters

No adapters are used.

Functionality

The function consists of a single assignment: `F_NumericObjectPool_S_TO_OBJID := stObj.u16ObjId;`. On `REQ`, the object ID is read from the structure and output with `CNF`.

Technical Details

- Pure function without state, version 1.0 (2026-10-11)
- The output has no name of its own but is the return value of the function
- The function is located in the package `logiBUS::utils::conversion::phys`

Application Scenarios

- Access to the object ID of a VT number variable when a block expects only the plain `UINT` ID instead of the whole `NumericObjectPool_S` structure

Conclusion

`F_NumericObjectPool_S_TO_OBJID` returns the `u16ObjId` of a `NumericObjectPool_S` structure as a `UINT`.

F_PHYS_LREAL_TO_RAW

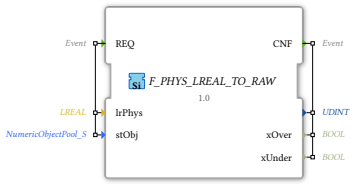


Image not available

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_PHYS_LREAL_TO_RAW converts a physical measured value in the LREAL data type into an ISOBUS-compliant raw value as a UDINT. Saturation limits are observed, and overflow and underflow states are signaled as Boolean outputs. The conversion follows the ISOBUS formula:

Display value = (Raw value + Offset) · Scaling Raw value = UDINT(UDINT(physical value / scaling) – Offset)

This function block is suitable for applications that need to integrate physical quantities into the ISOBUS protocol, particularly in agricultural and commercial vehicle technology.

Interface Structure

Event Inputs

Event	Description	Accompanying Variables
`REQ`	Starts the conversion	`lrPhys`, `stObj`

Event Outputs

Event	Description	Accompanying Variables
`CNF`	Confirms the conversion is complete	`xOver`, `xUnder`

Data Inputs

Name	Type	Initial Value	Description
`lrPhys`	`LREAL`	–	Physical Input Value (e.g., Temperature, Pressure)
`stObj`	`logiBUS.Utils::conversion::phys::NumericObjectPool_S`	`(u16ObjId := 65535, r32Scale := 1.0, i32Offset := 0, u8Decimals := 0)`	Data Structure with Scaling and Offset Parameters

Data Outputs

Name	Type	Description
(no name)	`UDINT`	Calculated ISOBUS raw value (0 … 4294967295)
`xOver`	`BOOL`	`TRUE`, if the input exceeds the upper limit (overflow)

xUnder	BOOL	TRUE, if the input falls below the lower limit (underflow)

Adapters

This function block does not use adapters.

Functionality

The function block performs the following steps:

1. **Calculation of limits:**

- Lower limit: `i32Offset * r32Scale`
- Upper limit: `(i32Offset + 4294967295) * r32Scale`

1. **Comparison and saturation:**

- If `lrPhys > obere Grenze` is present ⟶ Raw value = **UDINT#4294967295**, `xOver = TRUE`.

If `lrPhys < untere Grenze` is present ⟶ Raw value = **0**, `xUnder = TRUE`.

- Otherwise ⟶ Raw value = `DINT_TO_UDINT( LREAL_TO_DINT( lrPhys / r32Scale ) - i32Offset )`, no overflow/underflow message.

1. **Output:** The calculated raw value is provided at the unnamed `UDINT` output, and the event `CNF` is triggered.

Technical Features

- **ISOBUS Compliance:** The conversion follows the ISOBUS specification exactly (ISO 11783).
- **Saturation Behavior:** Critical ranges are represented by the overflow/underflow outputs `xOver` and `xUnder`, allowing subsequent functions (e.g., alarms) to react.
- **Scaling Unit:** The scaling factor `r32Scale` is of type `REAL` and is directly included in the calculation.
- **Integer Conversion:** The conversion from LREAL via DINT to UDINT is performed using the rounding standard in IEC 61499 (truncation of decimal places).
- **No State Machines:** The function block operates purely functionally without an internal state machine.

State Overview

The function block does not have an explicit state machine. The calculation is triggered by the event `REQ`; after one cycle, the result is provided at the data output and the event `CNF` is sent. The execution time is deterministic and depends only on the runtime environment.

Application Scenarios

- **Agricultural Control Units:** Conversion of sensor values (e.g., torque, pressure) into ISOBUS raw data for transmission to terminals or actuators.
- **Diagnostics and Monitoring:** Detection of measurement range exceedances via `xOver`/`xUnder` to trigger warnings.
- **Data Logging:** Preparation of physical measured values for archiving in ISOBUS format.

Comparison with Similar Function Blocks

Typical alternatives are custom-written calculation functions or generic conversion blocks.
The F_PHYS_LREAL_TO_RAW offers the following advantages:

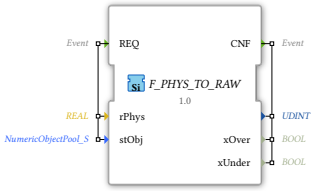
- **Predefined parameterization** via the structure `NumericObjectPool_S` (object ID, scaling, offset, decimal places).
- **Integrated saturation** and limit value monitoring.
- **Reusable** in all logiBUS projects.

Disadvantages compared to a custom solution include the reliance on the ISOBUS formula and the fixed data structure.

Conclusion

The F_PHYS_LREAL_TO_RAW function block is a compact, standards-compliant solution for converting physical measured values into the ISOBUS raw data format. Its saturation logic and monitoring outputs make it robust and reliable for use in industrial and agricultural control systems. The use of predefined structural parameters simplifies integration and reduces the potential for errors during parameterization.

F_PHYS_TO_RAW



• — ○ • — ○ • — ○ •

Introduction

The function block **F_PHYS_TO_RAW** converts a physical REAL value into an ISOBUS raw value (UDINT). Saturation limits are taken into account, and overflow and underflow alarms are issued. The conversion is based on the ISOBUS formula: `Anzeige = (Rohwert + Offset) * Skalierung`. The inverse calculation is: `Rohwert = UDINT(DINT(physikalischer Wert / Skalierung) - Offset)`.

Interface Structure

Event Inputs

Event	Description
REQ	Request to convert the physical value. Executed with the parameters `rPhys` and `stObj`.

Event Outputs

Event	Description
CNF	Confirmation that the conversion is complete. Returns the calculated raw value as well as the alarm flags `xOver` and `xUnder`.

Data Inputs

Variable	Type	Description
rPhys	REAL	The physical value to be converted.
stObj	NumericObjectPool_S	Structure with the parameters `i32Offset` (DINT) and `r32Scale` (REAL) for the ISOBUS formula.

Data Outputs

Variable	Type	Description
*	UDINT	The calculated ISOBUS raw value (function return value). Saturated to the range 0 to 4294967295.
xOver	BOOL	TRUE if the physical value exceeds the upper limit.
xUnder	BOOL	TRUE if the physical value falls below the lower limit.

Adapter

None.

Functionality

The function block first calculates the lower and upper limits:

- Lower limit: `lrLower = Offset * Skalierung`
- Upper limit: `lrUpper = (Offset + 4294967295) * Skalierung`

The physical value is then compared to these limits:

1. **Overflow (xOver = TRUE):** If `rPhys > lrUpper`, the raw value is set to the maximum UDINT value (4294967295).
2. **Underflow (xUnder = TRUE):** If `rPhys < lrLower`, the raw value is set to 0.
3. **Normal range:** Within the valid range, the raw value is calculated using the formula `UDINT(DINT(rPhys / Skalierung) - Offset)`.

This calculation is performed within the structured text code for each REQ event.

Technical Features

- The return type is UDINT, and saturation is applied to the full value range from 0 to 4294967295.
- The offset and scaling data are read from an external structure (`NumericObjectPool_S`), typically provided by the ISOBUS object pool.
- The function block does not use internal states; the calculation is purely combinatorial and is performed completely anew with each REQ.
- The conversion from REAL to DINT is performed using `REAL_TO_DINT`, which can lead to rounding errors with large values.

State Overview

The function block does not have an explicit state machine. It operates statelessly based on the current inputs. The output is recalculated with each REQ event.

Application Scenarios

- **ISOBUS Control Units:** Conversion of sensor readings (e.g., temperature, pressure) into ISOBUS-compliant raw data values for display on a terminal.
- **Agricultural Machinery Control:** Conversion of physical quantities that are represented with a specific scale and offset in the ISOBUS diagnostic protocol.
- **Test and Simulation Environments:** Generation of ISOBUS raw values from simulated physical values.

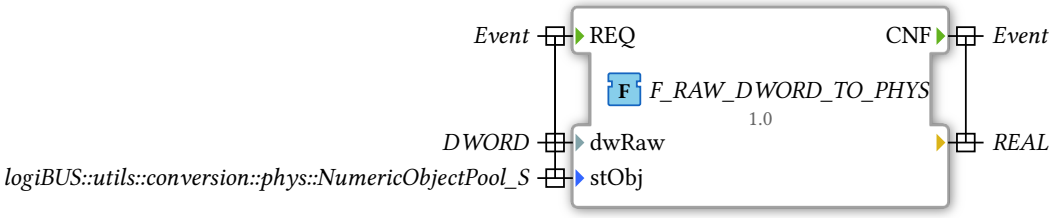
Comparison with Similar Function Blocks

Function blocks like `F_RAW_TO_PHYS` perform the inverse conversion (raw value ⟶ physical value). While this function block converts the physical quantity into the raw value, `F_PHYS_TO_RAW` additionally provides saturation and alarm signaling for overflow and underflow. Unlike general arithmetic function blocks, it is specifically designed for the ISOBUS convention with offset and scaling.

Conclusion

The `F_PHYS_TO_RAW` function block is a specialized module for ISOBUS conversion with integrated range monitoring. It offers a simple and robust interface for integrating physical measured values into the ISOBUS protocol and ensures that the raw value always remains within the valid range. The alarm outputs enable immediate fault detection in the event of limit violations.

F_RAW_DWORD_TO_PHYS



• – ◦ • – ◦ • – ◦ •

Introduction

The function `F_RAW_DWORD_TO_PHYS` converts an ISOBUS raw value (`DWORD`) into a physical `REAL` value. The ISOBUS formula is: `display = (raw + offset) * scale`.

Interface Structure

Event Inputs

- **REQ**: starts the conversion. Associated data: `dwRaw`, `stObj`.

Event Outputs

- **CNF**: signals the completed conversion. The result is available at the data output.

Data Inputs

- **dwRaw** (`DWORD`): ISOBUS raw value
- **stObj** (`logiBUS::utils::conversion::phys::NumericObjectPool_S`): object pool properties of the number variable (offset `i32Offset`, scale `r32Scale`)

Data Outputs

- **(unnamed)** (`REAL`): physical value (return value of the function)

Adapters

No adapters available.

Functionality

1. `dwRaw` is interpreted as an unsigned number (`DWORD_TO_UDINT`) and widened to `LINT` (`i64Raw`).
2. `stObj.i32Offset` is used as `LINT` without conversion (`i64Offset`).
3. The result is `LINT_TO_REAL(i64Raw + i64Offset) * stObj.r32Scale`.

Example: `raw = 50000`, `scale = 0.01`, `offset = -100000` ⟶ `50000 + (-100000) = -50000` ⟶ `-50000 * 0.01 = -500.0`.

Technical Details

- Implemented as an ST function (`FUNCTION ... : REAL`) in package `logiBUS::utils::conversion::phys`.
- The intermediate values are `LINT` to avoid overflow.
- The raw value is treated as an unsigned bit pattern and then widened to 64 bits.

Application Scenarios

- Converting ISOBUS raw values (e.g. from a VT number variable) into physical quantities

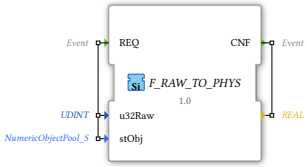
⚖️ Comparison with Similar Building Blocks

- F_RAW_TO_PHYS and F_RAW_TO_PHYS_LREAL are related conversions; the opposite direction is provided by F_PHYS_TO_RAW and F_PHYS_LREAL_TO_RAW.

Conclusion

`F_RAW_DWORD_TO_PHYS` converts an ISOBUS `DWORD` raw value into a `REAL` value using offset and scale from the object pool.

F_RAW_TO_PHYS



• — ○ • — ○ • — ○ •

Introduction

The function block **F_RAW_TO_PHYS** converts an ISOBUS raw value of type `UDINT` into a physical value of type `REAL`. The conversion is performed according to the standardized ISOBUS formula: display = (raw + offset) * scale`

It is typically used in agricultural control systems to convert sensor data or actuator values between the CAN-based ISOBUS signal layer and the application.

Interface Structure

Event Inputs

Name	Type	Description
REQ	Event	Starts the conversion of a raw value (using `u32Raw` and `stObj`).

Event Outputs

Name	Type	Description
CNF	Event	Confirms that the calculation is complete and the output value is valid.

Data Inputs

Name	Type	Description
u32Raw	UDINT	The raw integer value (e.g., from an ISOBUS message object).
stObj	`NumericObjectPool_S`	Structure containing the offset (`i32Offset : DINT`) and the scaling factor (`r32Scale : REAL`).

Data Outputs

Name	Type	Description
(no explicit name)	REAL	The calculated physical value, output as a floating-point number.

Adapters

None.

Functionality

When an event occurs at the **REQ** input, the function block performs the following calculation:

1. The raw value `u32Raw` is converted from `UDINT` to `LINT` (64-bit to prevent overflows during addition).
2. The offset `stObj.i32Offset` is also extended to `LINT`.
3. Both values are added: `(u32Raw + stObj.i32Offset)`.
4. The intermediate result is converted to `REAL` and multiplied by the scaling factor `stObj.r32Scale`.

Result: `REAL := (LINT(UDINT(u32Raw)) + LINT(stObj.i32Offset)) * stObj.r32Scale`

Example (from the source code documentation):

- Raw value = 50000
- Scaling = 0.01
- Offset = −100000
- Calculation: `(50000 + (-100000)) * 0,01 = (-50000) * 0,01 = -500,0`

The result is available at the data output after the **CNF** event.

Technical Features

- The function block uses **`LINT`** (64-bit integer) as an intermediate type to reliably prevent overflow when adding `UDINT` and `DINT` (e.g., large positive values with large negative values).
- Raw value = 50000
- Scaling = 0.01
- Offset = −100000
- Calculation = `(50000 + (-100000)) * 0,01 = (-50000) * 0,01 = -500,0`

- The function block uses **`LINT`** (64-bit integer) as an intermediate type to reliably prevent overflow when adding `UDINT` and `DINT` (e.g., large positive values with large negative values). - The output variable has no name of its own – in the 4diac IDE, it is automatically displayed as `OUT` or can be referenced via the interface.
- Offset and scaling are configured via an external structure (`NumericObjectPool_S`), which typically originates from a pool object that bundles all parameters of an ISOBUS object.

State Overview

The function block has **no** internal states or memory. Each call processes the current input data and simultaneously generates the output value. It is therefore a pure **combinatorial logic function block**.

Application Scenarios

- **ISOBUS control units** (tractors, implements) – conversion of raw CAN data into physical units (e.g., pressure, position, speed).
- **Agricultural Applications** – Conversion of sensor values from the ISOBUS protocol into displayable values.
- **Test and Simulation Environments** – Replication of the ISOBUS conversion for development and verification purposes.

Comparison with Similar Function Blocks

- **F_RAW_TO_PHYS** is specifically designed for the ISOBUS formula `(raw + offset) * scale` and uses the data types common there (`UDINT`, `DINT`).
- General converter function blocks (e.g., `LREAL_TO_REAL`) do not offer offset scaling logic and therefore must be manually supplemented with this calculation.

Other ISOBUS-specific function blocks, such as `F_PHYS_TO_RAW`, reverse the operation (converting from physical value to raw value).

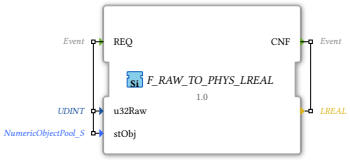
Conclusion

The **F_RAW_TO_PHYS** function block is a useful and cleanly implemented component for ISOBUS data conversion. The use of 64-bit intermediate values prevents overflow, and encapsulating the parameters in a structure facilitates reuse in projects that utilize the `NumericObjectPool` concept. It is suitable for both production use and rapid prototyping in the field of agricultural control systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

F_RAW_TO_PHYS_LREAL



• – ○ • – ○ • – ○ •

Introduction

The function block `F_RAW_TO_PHYS_LREAL` converts a raw ISOBUS integer value (UDINT) into a physical floating-point value (LREAL). The conversion follows the standardized ISOBUS formula: display = (raw + offset) * scale`

The block is optimized for use in agricultural control systems (ISOBUS) and uses the provided parameters from a pool of numeric objects.

Interface Structure

Event Inputs

Name	Type	With Variables
REQ	Event	u32Raw, stObj

The function block is triggered by an event at input `REQ`.

Event Outputs

Name	Type	With Variables
CNF	Event	(none)

After successful calculation, the result is acknowledged via event `CNF`.

Data Inputs

Name	Type	Description
u32Raw	UDINT	Raw value from the ISOBUS bus (0 … 4294967295)
stObj	NumericObjectPool_S	Structure with the conversion parameters: `i32Offset` (DINT) and `r32Scale` (REAL)

Data Outputs

Name	Type	Description
(no name)	LREAL	Calculated physical value (IEEE double precision)

The output signal has no explicit name but represents the directly calculated LREAL value.

Adapter

None.

Functionality

The function block is activated via the event `REQ`. The input data is read in, and the following calculation is performed:

1. The 32-bit raw value `u32Raw` (UDINT) is converted unsigned to a 64-bit integer (LINT).
2. The offset value `stObj.i32Offset` (DINT) is also converted to LINT and added to the raw value.
3. The sum (LINT) is converted to an LREAL.
4. The intermediate result is multiplied by the scaling factor `stObj.r32Scale` (REAL).
5. The final result is available as an LREAL at the output.
6. The event `CNF` is then triggered.

Type conversion ensures that large UDINT values (e.g., 4,000,000,000) can be processed without information loss.

Technical Features

- **Enhanced Accuracy**: The intermediate step via LINT prevents overflows during addition with large raw values and negative offsets.
- **ISOBUS Compliance**: The formula `display = (raw + offset) · scale` complies with the ISOBUS standard ISO 11783.
- **Parameterization via Structure**: All conversion coefficients are provided via the structure `NumericObjectPool_S`, making the function block universally applicable to various sensors and devices.
- **Single-Cycle Processing**: The calculation is performed without state storage within a single cycle.

State Overview

The function block does not have an internal state machine, as it is a pure transformation function. It follows this sequence:

1. Wait for `REQ` (idle state)

2. At `REQ` ⟶ calculation ⟶ output of the result ⟶ trigger `CNF` ⟶ return to idle state

Application Scenarios

- **Agricultural Engineering**: Conversion of ISOBUS raw values (e.g., speeds, pressures, flow rates) into physical units such as °/s, bar, or l/min.
- **Vehicle Control**: Processing of CAN data from tractors, harvesters, or application systems.
- **Simulation**: Replication of ISOBUS sensor data in test environments where raw values from a database with different scales are processed.

Comparison with Similar Function Blocks

Function Block	Input Data Type	Output Data Type	Special Features
`F_RAW_TO_PHYS_LREAL`	UDINT	LREAL	Double precision, LINT intermediate step
`F_RAW_TO_PHYS_REAL`	UDINT	REAL	Single precision, reduced computational load
`F_RAW_TO_PHYS_LINT`	UDINT	LINT	Output as integer, without scaling
`F_RAW_TO_PHYS_INT`	UINT	REAL	Only positive raw values, smaller range

This function block offers the highest precision and is recommended for applications requiring large ranges or fine increments.

Conclusion

F_RAW_TO_PHYS_LREAL` is a reliable and accurate converter for ISOBUS raw data processing.

By using LINT as the buffer and LREAL as the output type, rounding errors are minimized and large value ranges are covered. The module is particularly suitable for safety-critical or precision agricultural applications and can be flexibly adapted to various measured variables via its configurable structure.

2.10.3 logiBUS

2.10.3.1 📺 Video

- LogiBUS project update

2.10.3.2 🎧 Podcast

- Digital memories: The SR flip-flop in the logiBUS® system explained
- ESP32 as an industrial PLC: Revolution with Eclipse 4diac and logiBUS®
- From Cable Chaos to Custom Control: How Logibus is Revolutionizing Agricultural Tech with Accessible ISOBUS
- Logibus Revolution: Unpacking the “No Mapping” Future and Streamlined Development
- Logibus Unleashed: How Eliminating “Mapping” Simplifies Complex Systems and Boosts Usability

---

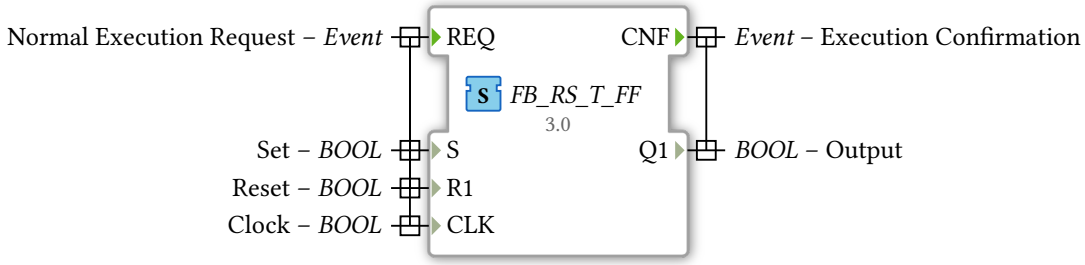
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

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

2.10.3.3 Numeric Index (logiBUS)

Here you will find all exercises from the `logiBUS` workspace, listed in ascending order.

FB_RS_T_FF



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_RS_T_FF` implements a bistable, reset-dominant latch with an additional toggle function. It combines the properties of an RS flip-flop (set and reset) with the ability to toggle the output on each rising edge of the clock signal. The reset input has priority, followed by the set input, and then the toggle function.

Interface Structure

Event Inputs

Event	Comment
`REQ`	Normal execution request – triggers the processing of the input signals.

Event Outputs

Event	Comment
`CNF`	Confirms execution after the output `Q1` has been calculated.

Data Inputs

Variable	Type	Comment
`S`	BOOL	Set – sets the output `Q1` to `TRUE`, provided `R1` is not active at the same time.
`R1`	BOOL	Reset (dominant) – sets the output `Q1` to `FALSE` and takes precedence over `S` and Toggle.
`CLK`	BOOL	Clock – clock generator for the Toggle function; on a rising edge, `Q1` is inverted if neither Reset nor Set is active.

Data Outputs

Variable	Type	Comment
`Q1`	BOOL	Latch output – displays the current stored state.

Adapters

This function block has no adapter interfaces.

Functionality

The following algorithm is executed for each event `REQ`:

- **Reset dominant**: If `R1 = TRUE` occurs, `Q1` is immediately set to `FALSE` – regardless of any other inputs.
- **Set**: If `R1 = FALSE` and `S = TRUE` occur, `Q1` is set to `TRUE`.
- **Toggle**: If neither Reset nor Set is active (`R1 = FALSE`, `S = FALSE`), the toggle mechanism is triggered: If `CLK` assumes the value `TRUE` and a rising edge is detected (i.e., `CLK` was `FALSE` in the previous call, stored in the internal variable `EDGE`), then `Q1` is inverted (`Q1 := NOT Q1`).

If CLK assumes the value TRUE and a rising edge is detected (i.e., CLK was FALSE in the previous call, stored in the internal variable EDGE), then Q1 is inverted (Q1 := NOT Q1). - The internal variable EDGE is then set to the current value of CLK to detect edges for the next call.

The algorithm is implemented in ST code as follows:

``structured text IF R1 THEN Q1 := BOOL#FALSE; ELSIF S THEN Q1 := BOOL#TRUE; ELSIF CLK AND NOT EDGE THEN Q1 := NOT Q1; END_IF; EDGE := CLK;

Technical Features

- **Reset Dominance**: The R1 input has the highest priority; when set, it overrides both Set and Toggle commands.
- **Edge Detection**: The toggle occurs only on a rising edge of CLK (transition from FALSE to TRUE), which is implemented by the internal variable EDGE.
- **Initial State**: The internal variable EDGE is initialized with TRUE, preventing an unintended toggle from being triggered on the first call, as the condition CLK AND NOT EDGE would otherwise be met at a static TRUE level.
- **Data Type**: All inputs and outputs are of type BOOL.

State Overview

The function block does not have an explicit state machine; the state is implicitly represented by the internal variables Q1 and EDGE. A state table summarizes the behavior:

Current Q1	R1	S	CLK (rising edge)	New Q1
x	TRUE	x	x	FALSE
x	FALSE	TRUE	x	TRUE
x	FALSE	FALSE	TRUE (and previously FALSE)	NOT Q1
x	FALSE	FALSE	FALSE or no edge	unchanged

Application Scenarios

- **Reset-dominant control** with additional switching option, e.g., for manual override in safety circuits.
- **Toggle function** on clock edges, e.g., as a frequency divider or switch in digital logic circuits.
- **Combined set/reset/toggle control** in automation systems, where an output can be set by sensor signals and toggled by a push button.

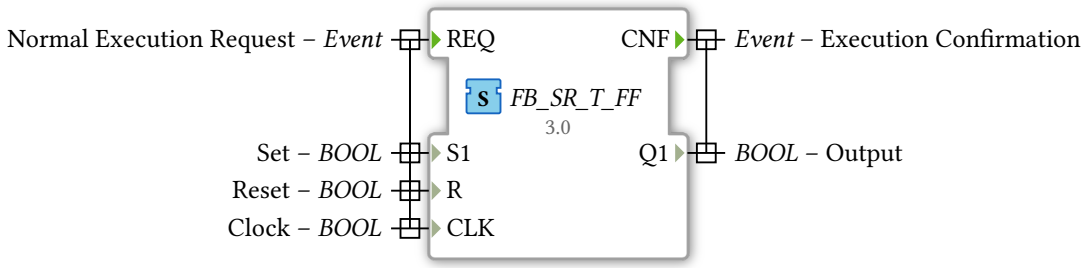
Comparison with similar components

- **RS flip-flop**: Pure RS without toggle; FB_RS_T_FF extends it with a toggle function.
- **Toggle flip-flop (T-FF)**: Pure toggle without set/reset; this component combines both functions, with the toggle only being executed when set/reset is inactive.
- **JK flip-flop**: Offers similar flexibility (set, reset, toggle) but requires two event inputs (e.g., for J and K). FB_RS_T_FF simplifies the interface to one event input and three Boolean values.

Conclusion

FB_RS_T_FF` is a versatile function block that combines a reset-dominant latch with an edge-triggered toggle function. It is suitable for applications requiring both fixed set and reset signals as well as switchable state transitions. Its clear prioritization (Reset > Set > Toggle) and built-in edge detection make it robust and easy to integrate into control logic.

FB_SR_T_FF



• – ◦ • – ◦ • – ◦ •

Introduction

The function block FB_SR_T_FF implements a bistable, set-dominant RS flip-flop with an additional toggle function. The circuit combines the behavior of an SR latch (set-dominant) with an edge-triggered T flip-flop, which toggles its output state on each clock cycle when neither set nor reset is active.

Interface Structure

Event Inputs

Name	Type	Comment
REQ	Event	Normal execution command (triggers processing)

Event Outputs

Name	Type	Comment
CNF	Event	Execution Confirmation

Data Inputs

Name	Type	Initial Value	Comment
S1	BOOL	-	Set Input (Dominant)
R	BOOL	-	Reset Input
CLK	BOOL	-	Clock Signal (Rising Edge Triggers Toggle)

Data Outputs

Name	Type	Initial Value	Comment
Q1	BOOL	-	Output Signal

Adapter

No adapters available.

Functionality

The function block is activated by an event at input REQ. On each execution, inputs S1, R, and CLK are sampled, and the internal state (Q1) is updated according to the following rules:

1. **Set Dominance:** If S1 = TRUE is active, Q1 is set to TRUE (regardless of R and CLK).
2. **Reset:** If S1 = FALSE and R = TRUE are already set, Q1 will be set to FALSE.

Reset: If S1 = FALSE and R = TRUE are already set, then Q1 will be set to FALSE. … 3. **Toggle on rising edge:** If neither S1 nor R are active (S1 = FALSE and R = FALSE, respectively), and the current value of CLK is TRUE, whereas it was FALSE in the previous cycle (i.e., rising edge), the output Q1 is inverted (Q1 := NOT Q1). Otherwise, Q1 remains unchanged.

The internal auxiliary flag EDGE stores the last state of CLK to enable edge detection. Initially, it is EDGE = TRUE, so no rising edge is detected on the first execution unless CLK is FALSE from the start.

The algorithm is implemented in Structured Text (ST), and the acknowledgment event CNF is sent after each execution.

Technical Features

- **Set-dominant behavior:** As long as S1 = TRUE is active, R is ignored. This prevents the undefined state of a classic RS flip-flop.

The algorithm is implemented in Structured Text (ST), and the acknowledgment event CNF is sent after each execution. - **Edge-Triggered Toggle:** The toggle function is triggered only on a rising edge of CLK,

not on every clock cycle. This makes the behavior deterministic and allows synchronization with the clock signal.

- **Internal Edge Storage:** The internal marker `EDGE` stores the previous value of `CLK`. Initially set to `TRUE`, it prevents the first cycle from triggering the toggle.
- **Cyclic Processing:** Processing can be called any number of times; the function depends only on the current input values and the stored edge state.

State Overview

The function block has a functional state machine with one state, `REQ`. This state executes the algorithm and then sends `CNF`. There are no other explicit states in the automaton, as the flip-flop behavior is purely data-driven within the algorithm.

State	Incoming Events	Action	Outgoing Events
REQ	REQ	Execute the algorithm, update `Q1` and `EDGE`	CNF

Application Scenarios

- **Control with Set/Reset Priority:** This function block is suitable for applications where Set takes precedence over Reset (e.g., safety circuits).
- **Toggle Function with Clock:** Combined with a clock, the output can toggle on every clock pulse, as long as no set or reset request is present. This can be used, for example, for modular counters or state machines.
- **Edge-Triggered Circuits:** In automation processes where state changes are only permitted at specific times (e.g., sampling).

Comparison with Similar Components

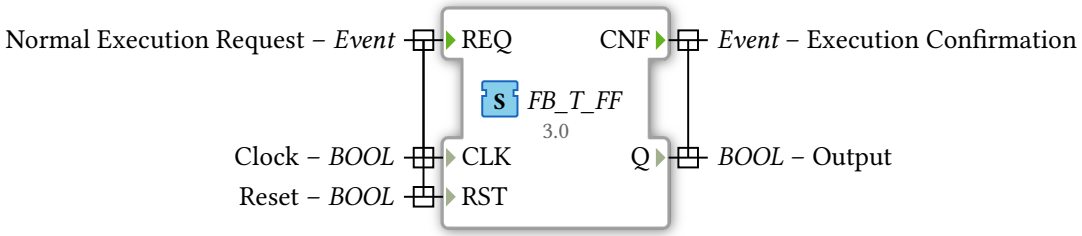
Compared to a pure SR flip-flop (e.g., `SR_FF`), this component offers a built-in toggle function that can invert the output on every clock cycle. Compared to a T flip-flop (`T_FF`), this component also has set and reset inputs with dominance. The classic JK flip-flop has two controlling inputs (J, K) without dominance, while this component represents a combination of set-dominant and edge-triggered toggle functionality.

Conclusion

The `FB_SR_T_FF` is a versatile, compact function block that combines the features of a set-dominant RS flip-flop and an edge-triggered T flip-flop. Thanks to its simple interface (one event input, three Boolean inputs), it can be easily integrated into existing control logic and is suitable for a wide variety of applications requiring a prioritized set/reset operation with an additional toggle function. Its implementation in structured text ensures clear readability and easy customization.

FB_T_FF

[[Image of the function block – shown in the 4diac IDE as a symbol with an event input REQ, output CNF, data inputs CLK (BOOL) and RST (BOOL), and data output Q (BOOL)]]



• – ◦ • – ◦ • – ◦ •

Introduction

The **FB_T_FF** (Toggle Flip-Flop) implements an edge-triggered, asynchronously resettable frequency divider or toggle switch. On each rising clock edge at the **CLK** input, the output **Q** is inverted (toggled). A high level at the **RST** input immediately resets **Q** to `FALSE`. This function block is particularly suitable for binary state transitions in control sequences.

Interface Structure

Event Inputs

Event	Associated Data	Description
REQ	CLK, RST	Normal execution job; triggers the algorithm

Event Outputs

Event	Associated Data	Description
CNF	Q	Confirms execution, returns the current value of Q

Data Inputs

Name	Type	Description
CLK	BOOL	Clock signal (edge detection)
RST	BOOL	Asynchronous reset (active high)

Data Outputs

Name	Type	Description
Q	BOOL	Output – toggles on rising CLK edge, unless reset is active

Adapters

No adapters available.

Functionality

This function block is called using the **REQ** event. On each call, the internal variable **EDGE** (Bool, initial TRUE) is updated as follows:

1. **Reset priority:** If RST = TRUE is active, Q is immediately set to FALSE.
2. **Edge detection:** If RST is not active and the condition CLK AND NOT EDGE is met, there is a **rising edge** of CLK. In this case, Q is inverted to Q := NOT Q.
3. **Edge detection:** If RST is not active and the condition CLK AND NOT EDGE is met, there is a **rising edge** of CLK. In this case, Q is inverted to Q := NOT Q. 3. **Storage:** After evaluation, EDGE := CLK is set so that the edge can be detected on the next call.

The algorithm is executed in the internal **REQ** state; after the algorithm, the **CNF** event is sent.

Technical Features

- **Immediate Edge Detection:** A rising edge is detected by comparing the current CLK value with the previous one (stored in EDGE). This is a typical software implementation of an edge-triggered flip-flop.
- **Asynchronous Reset:** The reset takes effect immediately with each call, regardless of the clock edge.
- **Initial State:** Q is undefined after the function block starts; it is only set by the first REQ call or a reset. The internal **EDGE** is initially set to TRUE, ensuring that the first rising edge (from FALSE to TRUE) is correctly detected.

State Overview

The function block is implemented as a **SimpleFB** with a single EC state:

State	Incoming Events	Outgoing Action	Outgoing Event
REQ	REQ	Execute algorithm REQ (Toggle/Reset logic)	CNF

The state is immediately reactivated after processing; there are no wait or follow-up states.

Application Scenarios

- **Frequency Divider:** Applying a square wave signal to CLK; half the frequency appears at Q.

- **Binary State Change:** Switching a light or valve on each clock pulse.
- **Reset-Proof Toggle:** By controlling RST, the output can be forced into a defined default state.
- **Pulse Counter:** Combining several FB_T_FFs in series results in a binary counter.

Comparison with Similar Function Blocks

Function Block	Property
SR Flip-Flop	Set and reset via separate inputs; no toggle behavior.
D-Flip-Flop	Takes the value of D on the clock edge; no toggle, no reset integrated.
FB_T_FF	Inverts the output on each rising edge; asynchronous reset possible.

The **FB_T_FF** is therefore ideal for simple switching tasks where no set/reset input is required.

Conclusion

The **FB_T_FF** implements a robust, edge-triggered toggle flip-flop with asynchronous reset. Its compact implementation makes it suitable for time-critical control systems in automation technology where reliable state transitions are required. Edge detection is cleanly implemented in software and requires no additional hardware.

The **FB_T_FF** is therefore ideal for simple switching tasks where a set/reset input is not needed. —

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

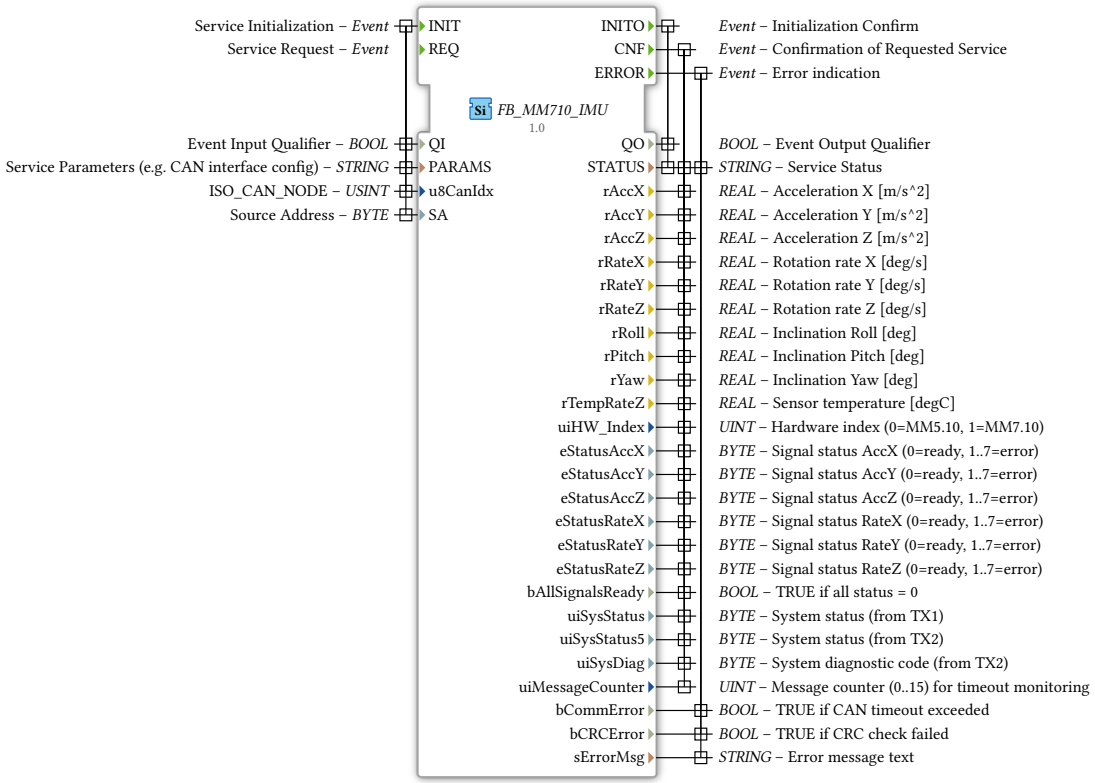
]

2.10.3.4 Bosch

Numeric Index (Bosch)

Numeric Index{index} single: Numerischer Index (bosch) Here you will find all exercises from the bosch workspace, listed in ascending order. ...

FB_MM710_IMU



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **FB_MM710_IMU** is a service-oriented module (SIFB) for connecting the Bosch MM7.10 IMU sensor via CAN/J1939. It enables the reading of acceleration, yaw rate, and tilt values, as well as the monitoring of system and error states. The FB encapsulates all CAN communication and signal processing and provides the data in a standardized format via event and data outputs.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initialization of the module. This event sets the CAN parameters (index, source address) and the activation qualifier QI.

Event Outputs

Event	Type	Description
INITO	EInit	Confirmation of successful initialization (QO = TRUE) or error message.
IND	Event	Indication from the resource: fires on every cycle without an active error and also publishes the reset of `bCommError`/`bCRCError` to FALSE. Provides QO, STATUS, all measurement and status values (`rAcc*`, `rRate*`, `rRoll`, `rPitch`, `rYaw`, `rTempRateZ`, `uiHW_Index`, `eStatus*`, `bAllSignalsReady`, `uiSysStatus`, `uiSysStatus5`, `uiSysDiag`, `uiMessageCounter`).
ERROR	Event	Fires only when an error actually occurs (entry trigger); provides QO, STATUS, `bCommError`, `bCRCError`, and `sErrorMsg`. The reset of `bCommError`/`bCRCError` to FALSE is not reported here but via IND.

Data Inputs

Variable	Type	Description
QI	BOOL	Activation Qualifier: Initialization (INIT) is only performed if QI = TRUE.
PARAMS	STRING	Service parameter, e.g., CAN interface configuration (optional).
u8CanIdx	USINT	CAN node index (default initial value: `INVALID`).
SA	BYTE	Source address for J1939 communication (initial value: `16#DA`).

Data Outputs

| Variable | Type | Description | | |-----|---|-------| | QO | BOOL | Initialization confirmation (TRUE = successful). | | STATUS | STRING | Status message (e.g., “Initialized,” “Error”). | | rAccX, rAccY, rAccZ | REAL | Acceleration values in the X, Y, and Z directions [m/s²]. | | rRateX, rRateY, rRateZ | REAL | Rotation rates around the respective axis [deg/s]. | | rRoll, rPitch, rYaw | REAL | Tilt angle (roll, pitch, yaw) [deg]. | | rTempRateZ | REAL | Sensor temperature [°C]. | | uiHW_Index | UINT | Hardware Index (0 = MM5.10, 1 = MM7.10). | | eStatusAccX … eStatusAccZ | BYTE | Acceleration signal quality (individually `eStatusAccX`, `eStatusAccY`, `eStatusAccZ`; 0 = ready, 1..7 = error). | | eStatusRateX … eStatusRateZ | BYTE | Rotation rate signal quality (individually `eStatusRateX`, `eStatusRateY`, `eStatusRateZ`; 0 = ready, 1..7 = error). | | bAllSignalsReady | BOOL | TRUE if all signal statuses are 0. | | uiSysStatus | BYTE | System status from TX message 1. | | uiSysStatus5 | BYTE | System status from TX message 2. | | uiSysDiag | BYTE | System diagnostic code (from TX2). | | uiMessageCounter | UINT | Message counter (0..15) for timeout monitoring. | | bCommError | BOOL | TRUE on CAN timeout. | bCRCError | BOOL | TRUE on failed CRC check. | sErrorMsg | STRING | Error text (e.g., “CAN timeout”). |

Adapter

No adapters defined.

Functionality

Upon receiving **INIT** with QI = TRUE, the FB_MM710_IMU initializes the CAN communication and the internal receive buffer. After successful initialization, **INITO** is set with QO = TRUE. The block has no request input: once valid data is received from the sensor, **IND** is output on every cycle without an active error and all data outputs are updated. If a communication or CRC error occurs, or if the message counter exceeds a timeout, **ERROR** is set.

The signal statuses (eStatus*) enable individual fault analysis for each axis. The hardware index distinguishes between older MM5.10 and current MM7.10 sensors.

Technical Features

- **CAN/J1939 Protocol** – Uses a fixed source address (default: `16#DA`).
- **Timeout Monitoring** – The `uiMessageCounter` (0-15) is incremented with each valid message; if it fails to increment, a communication error is reported after 16 missing messages.
- **Signal Status Bits** – Provide more granular information than simple "ready/error" flags.
- **CRC Check** – Faulty CAN frames are detected and reported via `bCRCError` and **ERROR**.
- **Hardware Differentiation** – `uiHW_Index` enables adaptive behavior for different sensor versions.

State Overview

The module goes through the following states (not explicitly as ECC, but inferable from its behavior):

1. **Inactive** – After startup, waiting for INIT.
2. **Initializing** – After an INIT event; establishing CAN communication.
3. **Ready** – After successful INITO; waiting for sensor messages.
4. **Data Received** – After a successful CAN message; IND is sent.
5. **Error** – In case of timeout or CRC error; ERROR is sent (fallback to Ready after error handling).

Application Scenarios

- **Mobile Machinery** – Tilt and acceleration monitoring of excavators, cranes, or forklifts.
- **Vehicle Dynamics** – Acquisition of roll, pitch, and yaw angles for stability control.
- **Industrial Robots** – Monitoring of vibrations and unexpected movements.
- **IoT Sensor Nodes** – Integration into higher-level controllers via CAN bus.

Comparison with Similar Function Blocks

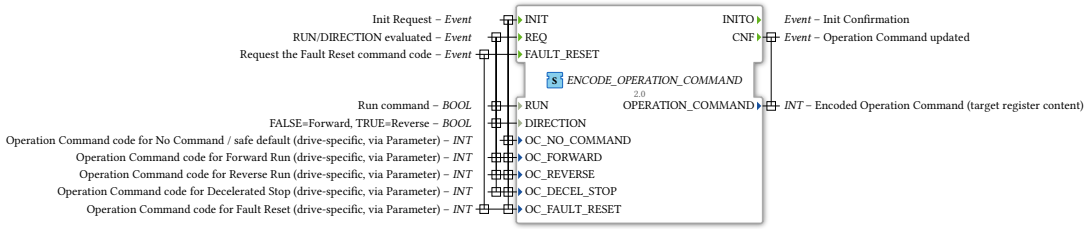
Compared to simple IMU drivers (e.g., via SPI/I²C), this function block offers direct integration into J1939 networks. The signal status bits enable diagnostics that are often lacking in standard function blocks. The integrated hardware index (MM5.10 / MM7.10) allows for easy migration. Other CAN IMU modules may use proprietary message formats, while this module is based on the open J1939 standard.

Conclusion

The FB_MM710_IMU is a powerful module for the reliable acquisition of IMU data in CAN-based automation systems. Its comprehensive status and error information supports complete diagnostics, and its simple parameterization via INIT makes it highly versatile. It is an ideal choice, especially in safety-critical applications using J1939.

2.10.3.5 drives

ENCODE_OPERATION_COMMAND



• – ○ • – ○ • – ○ •

Introduction

ENCODE_OPERATION_COMMAND builds the Operation Command word for a VFD/drive from RUN/DIRECTION, or from an explicit Fault Reset event. The block is generic for any drive whose control word follows the common Run/Direction/Stop-plus-Fault-Reset pattern (Modbus, CANopen PZD1, Profibus PPO, etc.) - all concrete command codes are supplied by the caller via Parameter, so the block itself carries no brand-specific knowledge, only the encoding shape.

The name Operation Command (rather than Control Word) is deliberate: Control Word is CANopen-specific terminology, whereas Operation Command is the vendor-neutral term used by e.g. the DELIXI H300 manual.

Interface Structure

Event Inputs

- **INIT**: Initialization - sets OPERATION_COMMAND to OC_NO_COMMAND once.
- **REQ**: Re-evaluate RUN/DIRECTION and encode OPERATION_COMMAND accordingly.
- **FAULT_RESET**: Request the Fault Reset command code, independent of the current RUN/DIRECTION state.

Event Outputs

- **INITO**: Initialization confirmation.
- **CNF**: OPERATION_COMMAND updated.

Data Inputs

- **RUN** (BOOL): Run command.
- **DIRECTION** (BOOL): FALSE=Forward, TRUE=Reverse.
- **OC_NO_COMMAND** (INT): Command code for "no command"/safe default (drive-specific, via Parameter).
- **OC_FORWARD** (INT): Command code for Forward Run (drive-specific, via Parameter).
- **OC_REVERSE** (INT): Command code for Reverse Run (drive-specific, via Parameter).
- **OC_DECEL_STOP** (INT): Command code for Decelerated Stop (drive-specific, via Parameter).
- **OC_FAULT_RESET** (INT): Command code for Fault Reset (drive-specific, via Parameter).

Data Outputs

- **OPERATION_COMMAND** (INT): Encoded Operation Command word (target register content).

Functionality

The block has no real state machine (no <ECC> with transition conditions) - each event independently triggers exactly one algorithm:

1. **INIT**: Sets OPERATION_COMMAND := OC_NO_COMMAND before the first REQ/FAULT_RESET arrives.
2. **REQ**: If RUN = TRUE, encodes OC_FORWARD or OC_REVERSE depending on DIRECTION; if RUN = FALSE, encodes OC_DECEL_STOP.
3. **FAULT_RESET**: Overwrites OPERATION_COMMAND once with OC_FAULT_RESET, independent of the current RUN/DIRECTION state - the next REQ encodes normally again afterwards.

Technical Features

- **All command codes are parameters, no hardcoded knowledge.** The block itself knows no drive-specific numeric values; OC_* are supplied by the caller via Parameter.
- **Renamed in version 2.0**: Formerly ENCODE_CONTROLWORD/CW_*/CONTROL_WORD, since "Control Word" is CANopen-specific terminology. Semantics unchanged, only names generalized.
- **Generalized from H300_ENCODE_CONTROLWORD** (version 1.0): Originally written for one specific drive type, then generalized to the generic Run/Direction/Stop-plus-Fault-Reset pattern.

Application Scenarios

- Driving a VFD/drive over a fieldbus control word (Modbus holding register, CANopen PZD1, Profibus PPO) where Run/Direction/Stop and an explicit Fault Reset are encoded in a single word.
- Generic block for multiple drive types: only the OC_* parameters need adjusting per drive, the encoding logic stays the same.

Conclusion

`ENCODE_OPERATION_COMMAND` encapsulates the common Run/Direction/Stop-plus-Fault-Reset encoding for VFD control words in a single, vendor-neutral block - all concrete command codes come from outside, the block itself carries no drive-specific knowledge.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.10.3.6 esp32

📺 Video

- The ESP32-S3-DevKitC-1

🎧 Podcast

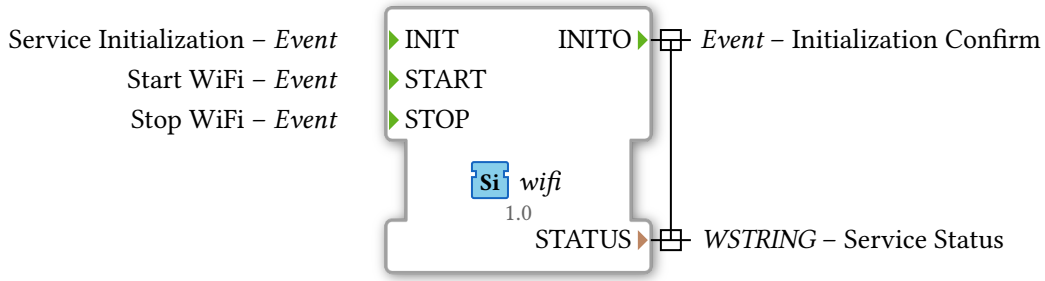
- ESP32 as an Industrial PLC: Revolution with Eclipse 4diac and logiBUS®
- ESP32 as PLC: Democratizing Industrial Automation with Eclipse 4diac
- ESP32 becomes an industrial PLC for agricultural machinery
- ESP32-S3 Development Boards ESP32-S3-DevKitC-1
- ESP32-S3 in Detail: Dual-Core, 32MB Power and CAN Bus for Agricultural and Construction Machinery Mechatronics

---

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

wifi



• – ◦ • – ◦ • – ◦ •

Introduction

The “wifi” function block is a Service Interface Function Block (SIFB) for controlling a WLAN connection. It provides the INIT, START, and STOP events for initializing, starting, and stopping the WLAN. The connection status is returned as a string via the STATUS output. The block is designed for use in automation systems that require a wireless network connection.

Interface Structure

Event Inputs

- **INIT** (EInit): Initializes the block and prepares the WLAN interface.
- **START** (EInit): Starts the WLAN connection.
- **STOP** (EInit): Stops the WLAN connection.

Event Outputs

- **INITO** (EInit): Confirms successful initialization. Triggered in conjunction with the STATUS data output.

Data Inputs

No data inputs available.

Data Outputs

- **STATUS** (WSTRING): Displays the current status of the WLAN service (e.g., “OK”, “Error”, “Connected”, “Disconnected”).

Adapters

No adapters available.

Functionality

The function block operates on the principle of a Service Interface Function Block. The internal logic is provided by the runtime environment. Typically, INIT is called first to configure the function block. The WLAN connection can then be activated with START. The connection is terminated with STOP. The STATUS output returns the current state after each operation.

Technical Features

- The function block is implemented as a SIFB (Service Interface Function Block) according to IEC 61499-2.
- The interface is open to both the RESOURCE and APPLICATION sides, enabling flexible integration.
- The exact implementation depends on the underlying hardware and operating system.

State Overview

The module goes through the following basic states:

- **Uninitialized**: After startup, before INIT is executed.
- **Initialized**: After successful INIT, ready for START.
- **Active (Connected)**: After successful START.
- **Inactive (Disconnected)**: After STOP or in case of an error.

Application Scenarios

- Controlling a WLAN interface in agricultural machinery or other vehicles.
- Integrating devices into a wireless network for data transmission in automation technology.
- Remote maintenance and diagnostics via WLAN.

Comparison with Similar Modules

Similar modules include Ethernet or serial communication modules. Unlike these, the Wi-Fi module focuses specifically on wireless WLAN connections and offers a simplified, event-driven interface. Other modules often require more complex configurations or support multiple protocols.

Conclusion

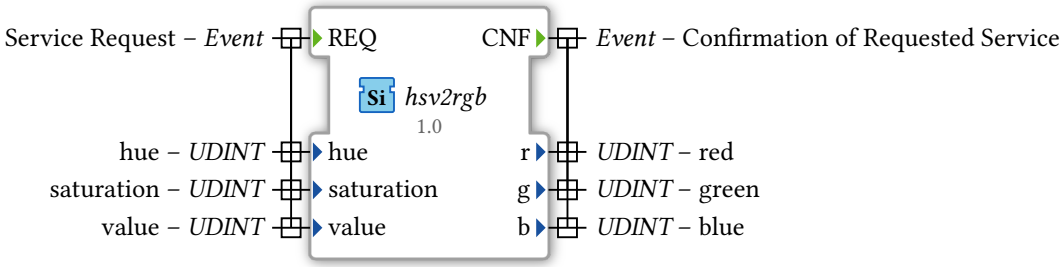
The “wifi” function block offers a simple and standardized way to integrate a WLAN connection into IEC 61499 systems. With its three events INIT, START, and STOP, as well as its STATUS output, it is intuitive to use and particularly suitable for applications requiring fast and reliable wireless communication.

rgb

hsv2rgb

• − ◦ • − ◦ • − ◦ •

The function block `hsv2rgb` is a utility block for color space conversion. It converts a color defined in the HSV color model (Hue, Saturation, Value) into the corresponding values of the RGB color model (Red, Green, Blue). This is particularly useful for applications that need to control colors based on their perceptual properties (hue, saturation, brightness), while output devices (such as LEDs or displays) expect RGB values.



- **REQ (Service Request):** Triggers the conversion calculation. This event updates the current values at data inputs `hue`, `saturation`, and `value`.
- **CNF (Confirmation of Requested Service):** Triggered once the conversion is complete. This event outputs the calculated RGB values at data outputs `r`, `g`, and `b`.
- **hue (UDINT):** The hue in the HSV model. The value range is not predefined in the function block, but typically a range of 0-360° or 0-65535 is used.
- **value (UDINT):** The brightness value in the HSV model. Typical range: 0-100% or 0-255.
- **r (UDINT):** The calculated red component in the RGB model.
- **g (UDINT):** The calculated green component in the RGB model.
- **b (UDINT):** The calculated blue component in the RGB model.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use adapters.

The block operates in an event-driven manner. The internal algorithm starts when the `REQ` event occurs. It receives the HSV values at the data inputs and calculates the corresponding RGB values according to the standardized HSV-to-RGB conversion algorithm. After the calculation is complete, the results are output at the data outputs `r`, `g`, and `b`, and the `CNF` event signals that the new values are valid and ready for retrieval.

- **Data Type:** All input and output data are of type `UDINT` (unsigned double integer, 32-bit). This enables high resolution and accuracy in color representation.
- **Value Range:** The module itself does not define any scaling or limiting of the value ranges for HSV or RGB. The interpretation of the numerical values (e.g., 0-360 for Hue, 0-255 for RGB) is the responsibility of the connecting application. The conversion algorithm must be designed accordingly for the expected input range.
- **Deterministic Behavior:** The execution is deterministic and always produces the same RGB output values for identical input values.

The function block has a simple, stateless sequence:

1. **Wait State:** The block waits for the `REQ` event.
2. **Calculation State:** At `REQ`, the input data is read and the conversion is performed. This state is transient.
3. **Output State:** The results are written to the outputs and the `CNF` event is generated. The module immediately returns to standby mode.

- **Color Control of RGB LEDs:** User-friendly color selection via HSV parameters (e.g., rotary encoder or touch interface) with subsequent conversion for LED control.
- **Visualization Systems:** Conversion of alarm or status colors defined in HSV for display on RGB monitors or panels.

- **Industrial Lighting:** Dynamic lighting control in production or logistics environments, where color transitions are intuitively defined via brightness and saturation.
-

Unlike general-purpose computing blocks (such as `MUL`, `ADD`) or scalers, `hsv2rgb` is an application-specific block that encapsulates a complex, non-linear algorithm. It saves the user the time-consuming and error-prone implementation of the conversion to Structured Text or other languages. A direct counterpart, `rgb2hsv`, for the reverse conversion would be a similarly specialized block.

- Exercise_031

The `hsv2rgb` function block is a useful and specialized tool for all 4diac applications involving color manipulation. By encapsulating the conversion algorithm in a standardized IEC 61499 block, it significantly simplifies programming, promotes reusability, and increases software reliability. Its use is recommended where intuitive, human-perceived color control meets device-oriented RGB output.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

🛠️ Zugehörige Übungen

Conclusion

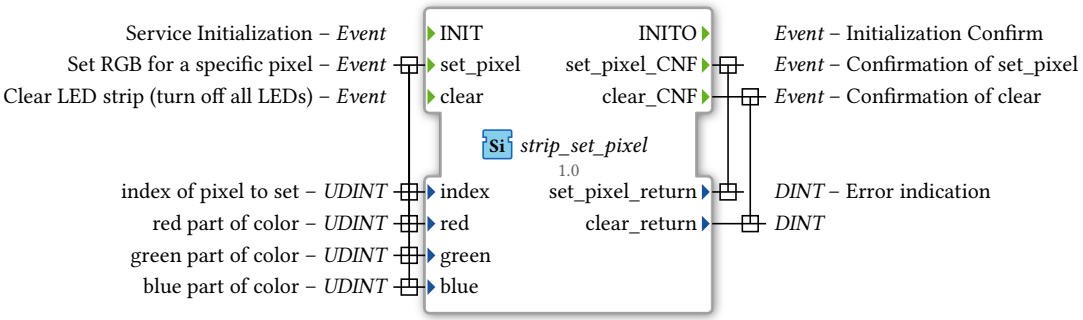
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

strip_set_pixel

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `strip_set_pixel` is used to control an RGB LED strip. Its main function is to set the color of a single pixel (a single LED) on the strip or to turn off the entire strip. It is designed for use in control systems that require precise, pixel-by-pixel control of RGB LEDs.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block. Must be triggered before first use.
- **set_pixel**: Triggers the command to set the color of a specific pixel. Linked to the data `index`, `red`, `green`, and `blue`.
- **clear**: Triggers the command to turn off (clear) all LEDs on the strip.

Event Outputs

- **INITO**: Confirms successful initialization of the function block.
- **set_pixel_CNF**: Confirms execution of the `set_pixel` command. Linked to the data output `set_pixel_return`.
- **clear_CNF**: Confirms execution of the `clear` command. Linked to the data output `clear_return`.

Data Inputs

- **index** (UDINT): The index (position) of the pixel on the LED strip to be set. The count typically starts at 0.
- **red** (UDINT): The red component of the color (value range depends on the hardware, e.g., 0-255).
- **green** (UDINT): The green component of the color (value range depends on the hardware, e.g., 0-255).
- **blue** (UDINT): The blue component of the color (value range depends on the hardware, e.g., 0-255).

Data Outputs

- **set_pixel_return** (DINT): Returns an error status after executing the `set_pixel` command. A value other than 0 indicates an error (e.g., invalid pixel index).
- **clear_return** (DINT): Returns an error status after executing the `clear` command. A value other than 0 indicates an error.

Adapter

This function block has no adapter interfaces.

Functionality

The function block operates in an event-driven manner. Upon receiving a `INIT` event, the underlying hardware (LED strip) is initialized, and the `INITO` event is generated.

The `set_pixel` event triggers the main command. The values present at the data inputs `index`, `red`, `green`, and `blue` at this time are read and sent to the LED strip to set the corresponding LED to the desired color. The `set_pixel_CNF` event is then output along with a status code (`set_pixel_return`).

The `clear` event sets all pixels of the strip to black (off), which is equivalent to clearing the current pattern. The `clear_CNF` event is then output with a status code (`clear_return`).

Technical Features

- The data inputs for the color values are of type `UDINT` (unsigned double integer), which allows for a large value range. The actual effective resolution (e.g., 8 bits = 0-255) depends on the connected hardware.
- The data outputs for feedback are of type signed `DINT` to represent positive and negative error codes.
- The function block follows the typical IEC 61499 Service Interface Pattern with separate acknowledgment events (`CNF`) for each command.

Status Overview

1. **Not Initialized**: After startup. Only the `INIT` event is valid.
2. **Initialized / Ready**: After successful execution of `INITO`, the commands `set_pixel` and `clear` can be received and processed. The function block returns to this state after each command.

Application Scenarios

- **Status Indicators**: Individual LEDs can be used as status or warning lights (e.g., machine in operation, fault at position X).
- **Simple Lighting Effects**: By sequentially setting individual pixels, chasing or flashing effects can be implemented.
- **Visualization of Process Data**: The color or brightness of a pixel can represent a process value (e.g., temperature, fill level).
-

⚖️ Comparison with Similar Components

Compared to components that transfer entire color patterns or images to an LED matrix (e.g., `strip_show` or `matrix_display`), `strip_set_pixel` operates at a very granular, pixel-by-pixel level. It offers maximum control over each individual LED but is less efficient for transferring complex images. Components like `strip_fill` typically set an entire area or all LEDs to a single color at once, while `strip_set_pixel` addresses each LED individually.

Compared to components that transfer entire color patterns or images to an LED matrix (e.g., `strip_show` or `matrix_display`), `strip_set_pixel` operates at a very granular, pixel-by-pixel level. It offers maximum control over each individual LED but is less efficient for transferring complex images. Components like `strip_fill` typically set an entire area or all LEDs to a single color at once, while `strip_set_pixel` addresses LEDs individually.

🛠️ Related Exercises

- Exercise_031

Conclusion

The `strip_set_pixel` function block is a fundamental and essential building block for controlling addressable RGB LED strips within 4diac. Its simple, event-based interface allows for precise and direct control of each individual pixel. Its strength lies in its simplicity for point-by-point control tasks, while for more complex animations, combining it with higher-level, sequencing blocks is recommended.

–

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.10.3.7 🔌 io

📺 Video

- Automation of the Future
- The Great Migration
- Digital Decisions
- Ideal Diodes
- The Dual Wi-Fi Solution

🎧 Podcast

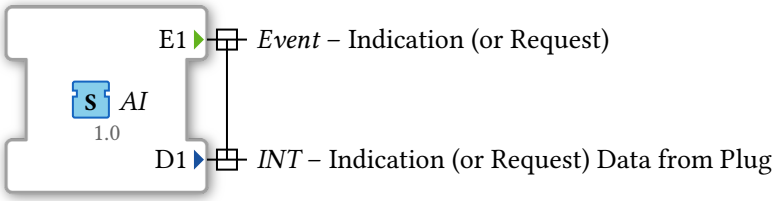
- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Target Groups, Structure, and Unique Selling Proposition
- Taming the Software Dragon: Industrial Automation and the Future of Production
- Eclipse 4diac: Innovation Engine Research vs. User Needs – What Really Drives Development?
- Taming the Software Dragon: Why IEC 61499 Could Revolutionize Industry

---

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Diode & Semiconductor Basics on ms-muc-docs.de
- 🌐 Smart Switch vs. Relay on ms-muc-docs.de

AI



📺 Video

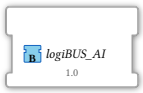
- AI: The Unknown Story
- The Silent Failure

🎧 Podcast

- From Clicking Relays to Intelligent Code: How Software Revolutionized Industrial Control
- ISOBUS Containers: Dynamic Control Panels for Clear Visibility and Greater Efficiency
- Analysis of the Amendment to the Master Craftsman Examination Regulations in the Agricultural and Construction Machinery Mechatronics Trade: A Detailed Comparison of the 2024 and 2001 Regulations
- The Relay in Detail: Switching Amplifiers, Protection, and the Secrets of A1/A2, 85/86, and the Hysteresis
- The fatal dilemma of the relay: Why silver contacts fail with small signals and gold melts under load – The burn-off effect explained

---

logiBUS_AI



• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_AI GlobalConstants define a collection of constants for configuring analog inputs in the logiBUS system. These constants are used for simple and consistent referencing of the analog input channels.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

No adapters available

Functionality

The logiBUS_AI GlobalConstants provide predefined constants of type `logiBUS_AI_S`, each representing a specific analog input channel. Each constant contains a pin number that identifies the physical input channel.

Technical Specifications

- **Constant Type**: All constants are of the structured data type `logiBUS_AI_S`
- **Pin Assignment**: The constants `AnalogInput_I1` to `AnalogInput_I9` correspond to analog inputs 1 to 9
- **Invalid Reference**: The constant `Invalid` with pin value 255 serves as a marker for invalid or non-existent inputs
- **Initialization**: All constants are initialized with their corresponding pin values upon declaration

State Overview

Since these are constants, they do not undergo state changes. Their values remain unchanged throughout runtime.

Application Scenarios

- Configuration of analog inputs in logiBUS systems
- Parameterization of function blocks that require analog inputs
- Type-safe referencing of analog channels in programming
- Error handling through the use of the Invalid constant

⚖️ Comparison with similar function blocks

Unlike function blocks with dynamic behavior, GlobalConstants offer:

- No runtime calculations
- No event handling
- Constants instead of variable values
- Direct initialization during declaration

Conclusion

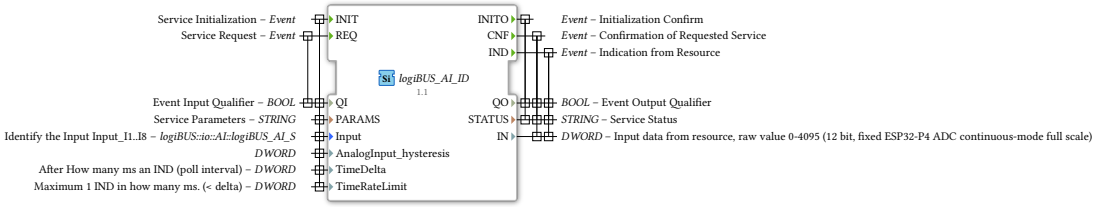
The logiBUS_AI GlobalConstants provide an efficient and type-safe method for referencing analog inputs in the logiBUS system. The predefined constants simplify programming and reduce the potential for errors during channel configuration.

logiBUS_AI_ID

• – ○ • – ○ • – ○ •

Introduction

The logiBUS_AI_ID is a Service Interface Function Block for processing double-word input data. This block serves as an interface for analog inputs and provides functions for initializing, querying, and indicating input data.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation
- **IND**: Resource Indication

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Enables/Disables the Service
- **PARAMS** (STRING): Service Parameter for Configuration
- **Input** (logiBUS_AI_S): Identifies the input (I1..I8) with the initial value “Invalid”
- **AnalogInput_hysteresis** (DWORD): Hysteresis Value for Analog Inputs. IND when the raw value changes by more than this value (0 = off, 1 = every change)
- **TimeDelta** (DWORD): Cyclic IND after at most this many ms (poll interval, 0 or 16#FFFFFFFF = off). Initial value 250
- **TimeRateLimit** (DWORD): Maximum of one IND per this many ms (less than TimeDelta). Initial value 100

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Status of the Service Output
- **STATUS** (STRING): Service status information

- **IN** (DWORD): Input data from the resource — raw value 0-4095 (12 bit), fixed on the ESP32-P4 in continuous/DMA ADC mode, no other bit width selectable

Adapter

No adapter interfaces available.

Functionality

The Function Block enables communication with analog input devices via the logiBUS system. During INIT initialization, the service parameters are configured and the input is identified. Data can be queried via REQ, while IND processes asynchronous data announcements from the resource. The hysteresis function assists in filtering signal noise.

Technical Features

- **Raw full-scale value of `IN`**: 0-4095 (12 bit) — the ESP32-P4 ADC driver (continuous/DMA mode) has no selectable bit width on this chip, `SOC_ADC_DIGI_MIN_BITWIDTH` = `SOC_ADC_DIGI_MAX_BITWIDTH` = 12. Raw-to-volt conversion: `Vout = Dout × Vmax / 4096` (`Vmax` depends on the ADC attenuation `ADC_ATTEN_DB_12`).
- Uses special data types from the logiBUS::io::AI package
- Supports hysteresis functionality for analog signals
- Offers both synchronous (CNF) and asynchronous (IND) operating modes
- Initializes inputs with a defined “Invalid” state

State Overview

The function block has the following operating states:

- Not initialized (before INIT)
- Initialized and ready (after INITO)
- Data query active (with REQ/CNF)
- Indication mode (with IND)

Application Scenarios

- Industrial automation systems with analog sensors
- Process control systems with hysteresis requirements
- Embedded systems with logiBUS communication
- Systems with multiple analog inputs (I1-I8)

⚖️ Comparison with Similar Function Blocks

Compared to simple analog input function blocks, logiBUS_AI_ID offers extended functions such as Hysteresis control, detailed status feedback, and a structured initialization procedure. Integration into the logiBUS system enables standardized communication.

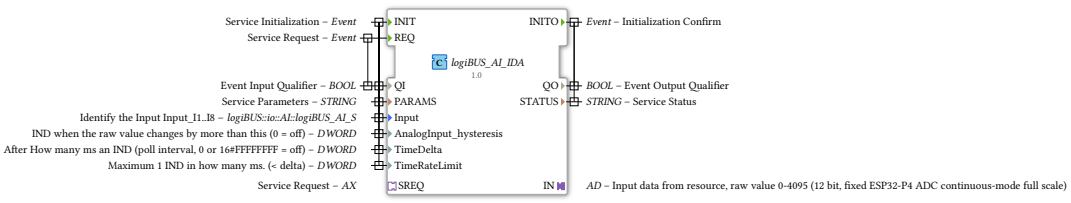
🛠️ Related Exercises

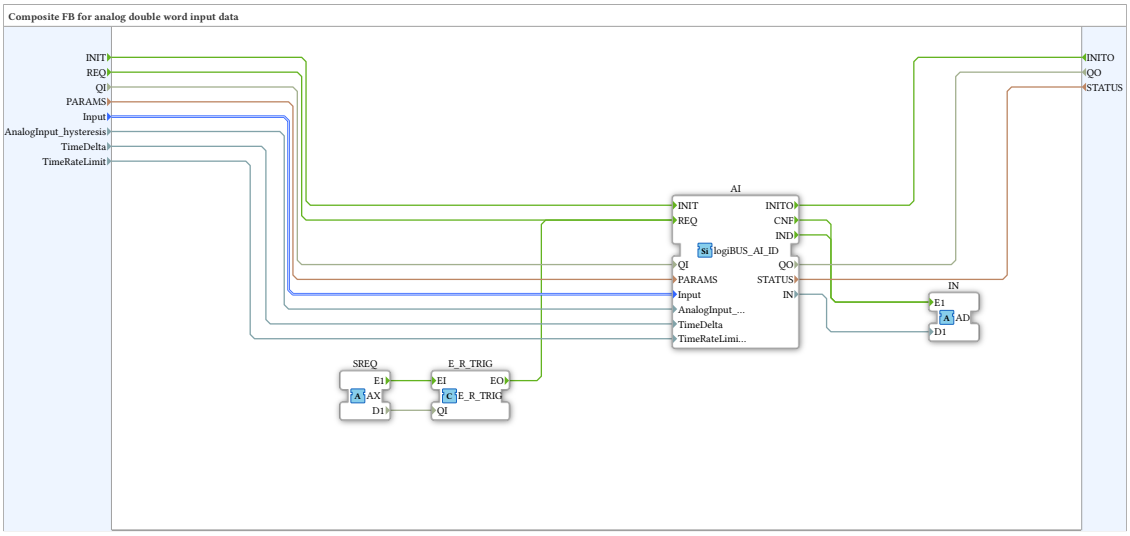
- Exercise_028
- Exercise_034

Conclusion

The logiBUS_AI_ID Function Block provides a robust and flexible solution for connecting analog input devices in industrial control systems. Its extensive parameterization options and integrated hysteresis functionality make it particularly suitable for demanding automation applications.

logiBUS_AI_IDA





• – ○ • – ○ • – ○ •

Introduction

The function block `logiBUS_AI_IDA` is a composite function block (FB) for processing analog double-word input data. It serves as an interface between a logiBUS resource and the application by providing uniform analog input values via an adapter and returning status information (QO, STATUS) to the calling instance. The function block supports both initialization-driven and event-driven processing.

Interface Structure

Event Inputs

Event	Type	With	Description
INIT	EInit	QI, PARAMS, Input, AnalogInput_hysteresis, TimeDelta, TimeRateLimit	Service Initialization: Configuration of the analog input and start of data provisioning.
REQ	Event	QI	Service Request: Triggers immediate processing or a status update.

Event Outputs

Event	Type	With	Description
INITO	EInit	QO, STATUS	Confirmation of initialization with quality and status information.

Data Inputs

Name	Type	Initial Value	Description
QI	BOOL	–	Qualifier for events (e.g., enabling processing).
PARAMS	STRING	–	Service parameters (e.g., configuration strings).
Input	logiBUS::io::AI::logiBUS_AI_S	Invalid	Selection of the analog input (e.g., Input_I1...I8).
AnalogInput_hysteresis	DWORD	–	Hysteresis for change detection. A value of 0 requires TimeDelta to be non-zero.

TimeDelta	DWORD	250	Cycle time in ms for cyclic processing (16#FFFFFFFF = only on change).
TimeRateLimit	DWORD	100	Minimum interval in ms between two events (IND) (< TimeDelta).

Data Outputs

Name	Type	Description
QO	BOOL	Output qualifier (e.g., valid state after INIT).
STATUS	STRING	Status message (e.g., initialization error or OK).

Adapter

Direction	Name	Type	Description
Plug	IN	adapter::types::unidirectional::AD	Receives the analog input data from the resource — raw value 0-4095 (12 bit), fixed on the ESP32-P4 (no other bit width selectable in continuous/DMA ADC mode).
Socket	SREQ	adapter::types::unidirectional::AX	Allows an external service request.

Functionality

The block encapsulates the internal FB `logiBUS_AI_ID`, which contains the actual analog input processing logic. The internal network connects:

- **INIT → AI.INIT** starts the resource configuration.
- **AI.INITO → INITO** returns the initialization confirmation.
- **REQ → AI.REQ** triggers immediate processing.
- **SREQ.E1** (external service request event) is routed via **E_R_TRIG** (edge detection) to **AI.REQ** — this lets an external adapter trigger processing as well.
- **AI.IND** and **AI.CNF** — both connect to **IN.E1** (the plug output) and signal externally that new data is available on the `IN` adapter.
- The data inputs (QI, PARAMS, Input, hysteresis, TimeDelta, TimeRateLimit) are passed straight through to the internal block.
- The QO and STATUS outputs come from the internal block.

Cyclic processing follows `TimeDelta`. If `TimeDelta = 16#FFFFFFFF` is set, an event is only generated when the analog value changes (taking hysteresis into account).

Technical Details

- **Raw full-scale value of `IN`**: 0-4095 (12 bit) — the ESP32-P4 ADC driver (continuous/DMA mode) has no selectable bit width on this chip, `SOC_ADC_DIGI_MIN_BITWIDTH = SOC_ADC_DIGI_MAX_BITWIDTH = 12`. Raw-to-volt conversion: `Vout = Dout × Vmax / 4096` (`Vmax` depends on the ADC attenuation `ADC_ATTEN_DB_12`).
- **Hysteresis (`AnalogInput_hysteresis`)**: If the value is 0, the cycle time (`TimeDelta`) must be non-zero, otherwise no events can be triggered.
- **Timing**: `TimeDelta` and `TimeRateLimit` allow fine-tuning the behavior — e.g., cyclic polling (TimeDelta > 0) or pure change notification (TimeDelta = 0xFFFFFFFF).
- **External Service Request**: Another component (e.g., a higher-level control block) can request an update via socket `SREQ`.
- **Composite Architecture**: The block is implemented as a composite, which allows reuse of the proven `logiBUS_AI_ID` and simultaneously extends the interface.

State Overview

The block itself does not have an explicit state machine, as its behavior is determined by the internal FB `logiBUS_AI_ID`. The following basic processes can be identified:

1. **Initialization Phase**

- Event `INIT` → Internal function block is configured → Confirmation `INITO` with QO/STATUS.

1. **Data Provisioning (Cyclical / Change-Based)**

- After INIT, the internal function block regularly sends events via `IND` (or `CNF`) to the plug `IN`, or upon value change.

1. **Manual Request**

- The current processing is triggered by `REQ` or `SREQ`; results are also output via `IN`.

Application Scenarios

- **Analog Sensor Acquisition**: Logging in analog sensors (e.g., temperature, pressure, level) with configurable hysteresis and sampling rate.
- **Monitoring with Minimal Bus Load**: By setting `TimeDelta = 0xFFFFFFFF`, events are only sent when relevant changes occur.
- **Safety-Critical Applications**: Combination of cyclic polling (e.g., 250 ms) with a fast change alarm (`TimeRateLimit` limits the event frequency).
- **Control Bus Connection**: This function block is suitable as a generic interface for logiBUS-compatible analog input modules with a uniform adapter protocol.

Comparison with Similar Function Blocks

- **logiBUS_AI_ID** (internal FB): Provides the core logic but does not have an additional socket for external service requests. `logiBUS_AI_IDA` extends this functionality with `SREQ` and indirect triggering via `E_R_TRIG`.
- **logiBUS_AI_S** (data structure): Serves as a pure data type for identifying the analog channel; `logiBUS_AI_IDA` uses it as an input parameter.
- **Other composite function blocks for digital inputs**: Unlike digital function blocks, the focus is on analog conversion, hysteresis, and time-controlled change detection.

Conclusion

The `logiBUS_AI_IDA` offers a flexible and extended interface for processing analog inputs in logiBUS-based automation systems. Its configuration options (hysteresis, time parameters, external service requests) make it suitable for a wide range of applications – from simple cyclic acquisition to efficient change-based communication. The composite structure allows for a clear separation of core logic and event handling customizations.

logiBUS_AI_S

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `logiBUS_AI_S` is a structured data type (struct) used to configure an analog input (AI) channel within the logiBUS library. It defines the basic parameter structure required to address a physical pin of a logiBUS hardware module for acquiring analog signals.

Interface Structure

This is a structured data type (struct), therefore it does not have event or adapter interfaces in the conventional sense of an executable function block. Its structure consists of a single data element.

Data Inputs

The structure itself contains the following data element, which serves as an input parameter for a configuring function block (e.g., `logiBUS_AI`):

- **`Pin`** (`USINT`): This value specifies the physical hardware pin or channel of the logiBUS module from which the analog signal is to be read. The default value is `255`, which is typically interpreted as an invalid or unconfigured state.

Functionality

The data type `logiBUS_AI_S` is a pure data structure and has no active functionality. It serves as a container for configuration data. Its main purpose is the parametric description of an analog input. A higher-level I/O manager or device driver function block (such as `logiBUS_AI`) reads the values stored in an instance of this structure and uses them to correctly configure and control the corresponding hardware interface.

Technical Features

- **Initial Value:** The member `Pin` is predefined with the initial value `255`. This value is outside the typical range of valid pin numbers (often 0 to n) and can therefore be used to detect an uninitialized or faulty configuration.
- **Type Safety:** Using a structured data type instead of individual, unconnected parameters increases the type safety and maintainability of the code in the 4diac IDE.
- **Library Affiliation:** This type is part of the `logiBUS::io::AI` package, clearly indicating its membership in the logiBUS IO library for analog inputs.

Application Scenarios

This struct is used exclusively in scenarios where logiBUS hardware with analog input modules (e.g., for voltage or current measurement) is integrated into a 4diac FORTE application. Typical applications include:

- Process monitoring (temperature, pressure, level via sensors)
- Control voltage acquisition
- Configuration of multi-channel measurement systems where a separate instance of `logiBUS_AI_S` is created for each channel.

⚖️ Comparison with Similar Building Blocks

Within the logiBUS library, similar structured data types exist for other I/O functions, such as `logiBUS_DI_S` for digital inputs or `logiBUS_AO_S` for analog outputs. These differ primarily in their context-specific names and are designed for the respective type of signal processing. However, the basic structure (a `Pin` field) is often identical or very similar.

Conclusion

The `logiBUS_AI_S` data type is an essential configuration element for using analog inputs with logiBUS in 4diac. As a simple, well-defined structure, it enables clear and centralized management of hardware parameters and forms the basis for reliable connection of analog sensors to an IEC 61499 control system.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

DI

📺 Video

- Decoding the datasheet
- The Great Migration
- The Art of Soldering
- The World of Standardization
- Digital Decisions

🎧 Podcast

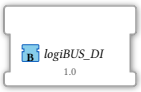
- 4diac IDE: Your “Hello World” of Automation – The Blinking Tutorial
- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- 4diac IDE: How the IEC 61499 Standard is Shaping Industrial Automation Revolutionized
- 4diac presentation: Target groups, structure, and unique selling proposition Sharpening
- Taming the software dragon: Industrial automation and the future of production

---

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

logiBUS_DI



• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_DI GlobalConstants definition provides a collection of predefined constants for digital inputs of the logiBUS system. These constants define the pin assignments for various digital input channels and are used to configure logiBUS_DI function blocks.

Interface Structure

Event Inputs

No event inputs available - this is a constant definition

Event Outputs

No event outputs available - this is a constant definition

Data Inputs

No data inputs available - this is a constant definition

Data Outputs

No data outputs available - this is a constant definition

Adapters

No adapters available - this is a constant definition

Functionality

The GlobalConstants definition logiBUS_DI provides structured constants of type `logiBUS_DI_S`, each containing a specific pin value:

- **Input_I1** to **Input_I9**: Defines digital inputs 1 to 9 with corresponding pin numbers
- **Invalid**: Defines an invalid pin with the Value 255

These constants are used as initialization values for configuring digital inputs in the logiBUS system.

Technical Features

- **Structure Type**: All constants use the structured data type `logiBUS_DI_S`
- **Pin Range**: Valid pins range from 1 to 9; invalid configurations are indicated by pin 255
- **Package Membership**: The constants are organized in the package `logiBUS::io::DI`
- **Initialization**: All constants are initialized with their pin values upon declaration

State Overview

Since this is a pure constant definition, there are no state transitions or state machines.

Application Scenarios

- Configuration of digital input modules in logiBUS systems
- Parameterization of function blocks that process digital inputs
- Type-safe pin assignments in control programs
- Error handling through the use of the invalid constant

⚖️ Comparison with similar function blocks

Unlike function blocks with active logic, this is a pure constant definition that:

- Does not perform any active processing
- Does not process any events
- Provides only configuration data
- Is evaluated at compile time

Conclusion

The logiBUS_DI GlobalConstants offer a structured and type-safe method for configuring digital inputs in the logiBUS system. The predefined constants reduce the susceptibility to errors in pin assignment and improve the maintainability of control programs.

logiBUS_DI_Events



• – ◦ • – ◦ • – ◦ •

Introduction

The GlobalConstants definition `logiBUS_DI_Events` provides a comprehensive collection of event constants for digital inputs (DI) in the logiBUS system. These constants define various button press events and their unique identifiers, which are used in the control logic for processing user input via pushbuttons.

Functionality

The GlobalConstants definition `logiBUS_DI_Events` contains 14 different event constants of type `logiBUS_DI_Events_S`, each identified by a unique numeric value (`EType`). These constants are used to represent various states and events of pushbuttons in the control logic.

Technical Features

- **Structure Type**: All constants use the data type `logiBUS_DI_Events_S`
- **Value Range**: The EType values range from 0-12 for valid events, with 255 for invalid events
- **Package Membership**: `logiBUS::io::DI`
- **Constant Definition**: All values are declared as global constants (CONSTANT)

Documentation for the individual events: https://components.espressif.com/components/espressif/button

State Overview

The event constants cover the following states:

EType	Constant	Description
0	BUTTON_PRESS_DOWN	Button pressed
1	BUTTON_PRESS_UP	Button released
2	BUTTON_PRESS_REPEAT	Button pressed again
3	BUTTON_PRESS_REPEAT_DONE	Button repeat ended
4	BUTTON_SINGLE_CLICK	Single click
5	BUTTON_DOUBLE_CLICK	Double click
6	BUTTON_MULTIPLE_CLICK	Multi-click
7	BUTTON_LONG_PRESS_START	Long press begins
8	BUTTON_LONG_PRESS_HOLD	Long press is held
9	BUTTON_LONG_PRESS_UP	Long press ends
10	BUTTON_PRESS_END	Press ends
11	BUTTON_EVENT_MAX	Maximum event ID

12	BUTTON_NONE_PRESS	No key press
255	Invalid	Invalid event

Application Scenarios

- **User Interface Control**: Processing keystrokes on operator panels
- **Machine Control**: Capturing manual inputs on industrial controllers
- **Safety Systems**: Monitoring emergency stop buttons and safety switches
- **Menu Navigation**: Controlling display menus via keystrokes
- **State Monitoring**: Detecting various keystroke patterns

⚖️ Comparison with Similar Function Blocks

Unlike simple digital input blocks that only detect binary states (ON/OFF), `logiBUS_DI_Events` offers advanced event detection with:

- Time-resolved events (Press/Release)
- Complex interaction patterns (Clicks, Double-Clicks)
- Long-term detection (Long Press)
- Repeat functionality

Conclusion

The `logiBUS_DI_Events` The GlobalConstants definition provides a comprehensive and structured basis for processing push-button events in industrial control systems. Through standardized event codes, it enables a consistent and maintainable implementation of user interactions via digital inputs.

logiBUS_DI_Events_S

• – ◦ • – ◦ • – ◦ •

Introduction

The structured data type `logiBUS_DI_Events_S` is used to configure events for digital inputs (DI) within the logiBUS framework. It defines the event type that can be assigned to a specific digital input channel. This data type is an essential component for parameterizing logiBUS I/O function blocks.

Interface Structure

This block is a pure structured type and does not have any event or adapter interfaces in the conventional sense of a function block. Its structure is used as an input or output data parameter in other function blocks.

Data Inputs

Not applicable – This is a data type.

Data Outputs

Not applicable – This is a data type.

Adapters

Not applicable – This is a data type.

Functionality

The data type `logiBUS_DI_Events_S` encapsulates a single variable (`EType`) that represents an event type code. The default value is set to `255`, which typically corresponds to an inactive or unconfigured state. Specific values (e.g., 0, 1, 2, ...) are used to identify specific events (such as edge transitions) that can then be processed by higher-level control logic.

Technical Features

- **Data Type:** This is a structured data type (`STRUCT`) that is used as a complex data parameter in other function blocks.
- **Initial Value:** The contained variable `EType` is pre-initialized with the value `255`. This value likely serves as a placeholder or "invalid" indicator.

- **Base Type:** The variable `EType` is of type `USINT` (Unsigned Short Integer, 0-255), which provides a limited and efficient range of values for event codes.

State Overview

Since this is a passive data type, there is no state machine. The "state" is the stored value of the variable `EType`.

Application Scenarios

This structured data type is primarily used in conjunction with logiBUS I/O function blocks, for example:

- As an element in an array for configuring all digital inputs of a module.
- As a parameter to specify which physical event (e.g., rising edge) should trigger a logical event in the control program.

⚖️ Comparison with Similar Function Blocks

Unlike function blocks (FBs) such as `E_R_TRIG` or `F_TRIG`, which independently generate edge events, `logiBUS_DI_Events_S` is a pure configuration data type. It provides the information about *which* hardware event should be translated into such a logical edge event by a separate driver or hardware abstraction block.

Conclusion

The `logiBUS_DI_Events_S` is a fundamental configuration block within the logiBUS architecture. It enables flexible and parameterizable mapping of hardware events at digital inputs to software-based processing routines. Its simple structure, reduced to a `USINT`, ensures efficiency and ease of use in higher-level configuration structures.

logiBUS_DI_S

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `logiBUS_DI_S` is a structured data type (struct) used to configure a digital input (DI) within the logiBUS framework. It defines the basic parameters necessary to control a physical or logical input point on a logiBUS-compatible I/O system. This data type is typically used as input for specific I/O function blocks.

Interface Structure

As a structured data type, `logiBUS_DI_S` does not have event or adapter interfaces in the conventional sense. Its structure consists of a single data element.

Data Inputs

The structure itself represents the configurable data inputs. It is passed as a whole to a higher-level function block.

Name	Data Type	Initial Value	Comment
`Pin`	`USINT` (Unsigned Short Integer)	255	Defines the number of the input pin or channel to be addressed.

Data Outputs

This data type has no data outputs of its own.

Functionality

The data type `logiBUS_DI_S` is purely a data storage structure. Its function is limited to storing the value of the member `Pin`. The initial value `255` often serves as a default or invalid value (e.g., "not configured"). A specific I/O function block that expects this struct as an input reads the `Pin` value and uses it to perform hardware addressing for the digital input.

Technical Details

- **Data type `USINT`:** The use of a `USINT` (0…255) for the pin suggests that up to 256 different input channels per device or module can be addressed.

- **Initial value 255:** This value is outside the typical usable range (0...n, where n<255) and can be interpreted by the application logic or driver as an indicator of a missing configuration.
- **StructuredType:** As `StructuredType`, this block is not an executable function block, but a pure data type used in the declaration of variables and interfaces of other function blocks.

State Overview

Since it is a passive data type, `logiBUS_DI_S` does not have a state machine.

Application Scenarios

- **I/O Block Configuration:** This struct primarily serves as an input parameter for function blocks such as `logiBUS_DI` or similar, which perform the actual reading operation from a digital input.
- **Parameterization via Networks:** The structure can easily be instantiated and configured (`Pin` number set) in higher-level controllers (e.g., a PLC) and passed on to subordinate I/O nodes or driver blocks.
- **Structured Programming:** Enables clean encapsulation of configuration data for digital inputs, increasing code reusability and readability.

⚖️ Comparison with Similar Building Blocks

- **Simple `USINT` Variable:** Instead of a struct, a `USINT` variable could be used directly. However, the struct offers semantic clarity (the name `logiBUS_DI_S` describes its purpose) and is easier to extend if further configuration parameters (such as filter time or inversion) need to be added in the future.
- **Device-Specific Structs:** Within the logiBUS ecosystem, similar structs could exist for other functions (e.g., `logiBUS_DO_S` for digital outputs, `logiBUS_AI_S` for analog inputs). This would enable a uniform configuration philosophy across different I/O types.

🛠️ Related exercises

- Uebung_003a_sub
- IX_TO_Aux_QX
- IXA_TO_Aux_QXA
- IX_TO_logiBUS_LED_strip_QX

Conclusion

The `logiBUS_DI_S` is a basic and specific configuration block for the logiBUS I/O architecture. As a structured data type, it offers a standardized and clear method for defining the target address (pin) for a digital input. Its strength lies in its simplicity and its readiness for future extensions, making it a useful component for the structured configuration of fieldbus systems in 4diac FORTE.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

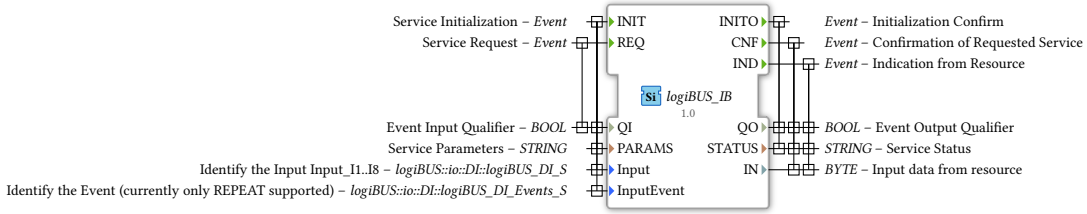
]

logiBUS_IB

• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_IB is a Service Interface Function Block for processing byte input data. This block serves as an input interface for logiBUS systems and enables communication with digital inputs. It supports special event handling such as key press repetitions and provides a standardized interface for initializing and operating input devices.



Interface Structure

Event Inputs
- **INIT**: Service Initialization Event
- Linked to: QI, PARAMS, Input, InputEvent
- **REQ**: Service Request Event
- Linked to: QI

Event Outputs
- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Service Operation
- Linked to: QO, STATUS, IN
- **IND**: Indication from Resource Interface
- Linked to: QO, STATUS, IN

Data Inputs
- **QI** (BOOL): Event Input Qualifier - Controls the execution of service operations
- **PARAMS** (STRING): Service Parameters for Configuration
- **Input** (logiBUS_DI_S): Identifies the input (I1..I8), Initial value: Invalid
- **InputEvent** (logiBUS_DI_Events_S): Identifies the event (currently only REPEAT supported), Initial value: Invalid

Data Outputs
- **QO** (BOOL): Event Output Qualifier - Status of the service operation
- **STATUS** (STRING): Service status information
- **IN** (BYTE): Input data from the resource

Adapter
No adapter interfaces available.

Functionality
The logiBUS_IB Function Block operates as an input service interface for logiBUS systems. During initialization (INIT event), the service parameters are configured and the inputs are defined. The block can respond to both explicit requests (REQ event) and asynchronous events from the hardware. It offers special support for repeated button press events (BUTTON_PRESS_REPEAT).

Technical Features
- Supports special event types such as BUTTON_PRESS_REPEAT and BUTTON_PRESS_REPEAT_DONE
- Byte-based data input (8-bit)
- Flexible input configuration via PARAMS string
- Separate event outputs for acknowledgments (CNF) and hardware indications (IND)
- Initial value “Invalid” for unconfigured inputs

State Overview
The function block has the following operating states:

- **Not Initialized**: Before INIT event
- **Initialized**: After successful INIT operation
- **Ready**: For REQ requests and IND events
- **Error State**: In case of faulty operations (reportable via STATUS)

Application Scenarios
- Digital input modules with 8 inputs
- Keyboard interface with repeat function
- General digital input interfaces in logiBUS systems
- Hardware-related input operations with event handling

⚖️ Comparison with Similar Function Blocks

Compared to simple digital input blocks, logiBUS_IB offers enhanced service functionality with:

- Configurable configuration
- Event-based communication
- Status feedback
- Special support for key press repetitions

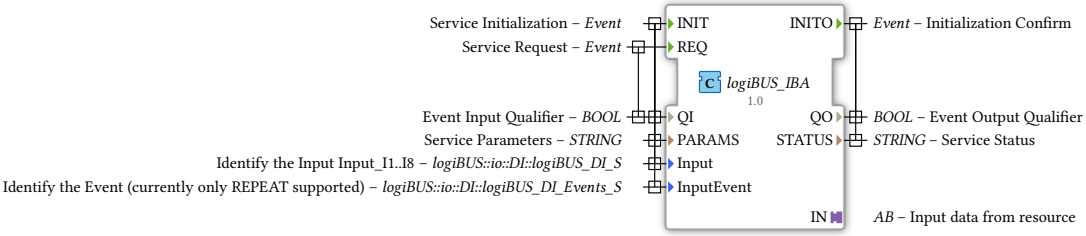
🛠️ Related Exercises

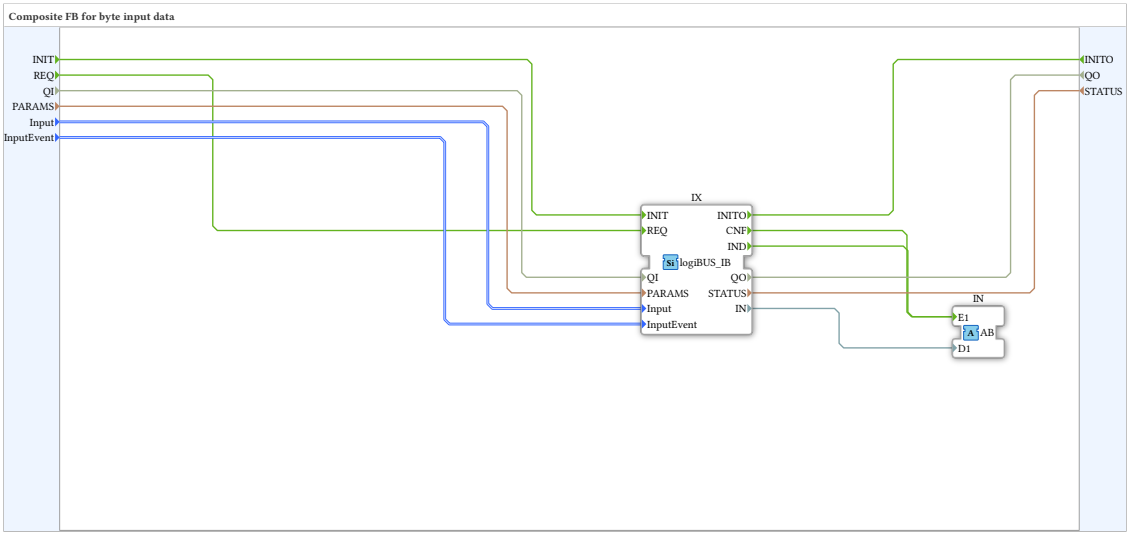
- Exercise_006c
- Exercise_011a

Conclusion

The logiBUS_IB Function Block represents a powerful and flexible solution for byte input data in logiBUS systems. Its service-oriented architecture and support for special event types make it particularly suitable for applications requiring advanced input functionalities such as key press repetitions.

logiBUS_IBA





• — ○ • — ○ • — ○ •

Introduction

The **logiBUS_IBA** function block is a composite block for processing byte input data via the logiBUS protocol. It encapsulates the initialization, configuration, and event control of a logiBUS input block and provides the data via an adapter. The block is designed for use in agricultural control systems.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Service initialization; takes over the configuration parameters (QI, PARAMS, Input, InputEvent)

REQ	Event	Service request to execute a function (dependent on QI)

Event Outputs

Event	Type	Description
INITO	EInit	Initialization confirmation; signals successful or failed initialization (QO, STATUS)

Data Inputs

Variable	Type	Description
QI	BOOL	Qualifier for event inputs; controls the activation of event processing
PARAMS	STRING	Service parameter (e.g., configuration data for the logiBUS driver)
Input	logiBUS::io::DI::logiBUS_DI_S	Selection of the physical input (e.g., Input_I1..I8); Default value: `logiBUS_DI::Invalid`
InputEvent	logiBUS::io::DI::logiBUS_DI_Events_S	Selection of the event type (currently only `REPEAT` is supported); Default value: `logiBUS_DI_Events::Invalid`

Data Outputs

Variable	Type	Description
QO	BOOL	Qualifier for event outputs; indicates the success of the operation
STATUS	STRING	Status message (e.g., error or success message)

Adapters

Adapter	Type	Description
IN	adapter::types::unidirectional::AB	Unidirectional adapter for providing input data to the higher-level resource

Functionality

The **logiBUS_IBA** function block works internally with an instance of the **logiBUS_IB** function block (from the `logiBUS::io::DI` library). The event and data paths are linked as follows:

- During the **INIT** event, the inputs `QI`, `PARAMS`, `Input`, and `InputEvent` are forwarded to the corresponding initialization of the internal function block. After initialization is complete, the output event **INITO** is set, and the outputs `QO` and `STATUS` are updated.
- The **REQ** event triggers the internal function block for processing a service request. The processing depends on the current state and the input parameters.
- Data read from the logiBUS input is transferred to the output adapter **IN** via the adapter connection (`IN.E1` and `IN.D1`). The adapter then provides the data to a higher-level function block or resource.
- Error and status signals are reported back via `QO` and `STATUS`.

Technical Features

- **Composite FB:** This function block encapsulates the complex initialization and data provision of a logiBUS input in a single interface.
- **Adapter-Based Data Output:** Input data is not output via separate data outputs, but rather via a **unidirectional adapter** (type `AB`). This allows for flexible coupling with other function blocks.
- **Configurable Inputs:** The physical input (I1..I8) and the event behavior (e.g., repetition) can be selected using the parameters `Input` and `InputEvent`.

- **License and Origin:** This function block is provided under the **Eclipse Public License 2.0** and is optimized for use in agricultural technology.

State Overview

This function block does not have an explicitly displayed state machine, as it is a composite function block. The internal behavior is determined by the included function block **logiBUS_IB**, which typically implements the following states:

- **IDLE** – Waiting for INIT or REQ
- **INIT** – Initialization with parameters
- **RUN** – Ready for operation, data processing active
- **ERROR** – Error state (status message via STATUS)

Application Scenarios

- **Agricultural Control Systems:** Acquisition of digital input signals (e.g., sensors, switches) via logiBUS and forwarding to a controller.
- **Input Modules in Fieldbus Systems:** The function block serves as a standardized interface for logiBUS inputs in IEC 61499-2 applications.
- **Prototypes and Test Setups:** Thanks to its configurability, the function block can be quickly adapted to various input configurations.

Comparison with Similar Function Blocks

- **logiBUS_IB:** The internal function block is a pure Basic Function Block; **logiBUS_IBA** adds an adapter output and simplified event control.
- **SYNC/D_SYNC:** These function blocks from the Sync library offer similar input configuration but without specific logiBUS protocol support.
- **Generic Input Adapter:** Compared to generic adapters, **logiBUS_IBA** offers a predefined logiBUS configuration, reducing development effort.

Conclusion

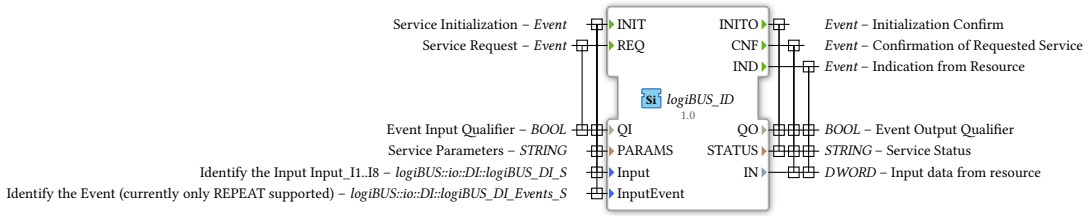
The **logiBUS_IBA** is a practical composite function block for integrating logiBUS inputs into IEC 61499-based automation systems. By combining initialization, parameter configuration, and adapter-based data output, it creates a clean and reusable interface. It is particularly suitable for agricultural applications that rely on the logiBUS protocol.

logiBUS_ID

• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_ID function block is an input service interface module for double-word input data (DWORD). It serves as an interface for communication with logiBUS input devices and enables the processing of digital input signals in 4diac control systems.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the function block with parameters and configuration
- **REQ**: Service Request - Triggers a query of the current input data

Event Outputs

- **INITO**: Initialization Acknowledgement - Confirms successful initialization
- **CNF**: Confirmation of Requested Service - Acknowledges a REQ request

- **IND**: Indication from Resource - Signals asynchronous events from the input device

Data Inputs

- **QI**: Event Input Qualifier (BOOL) - Enables/Disables the service
- **PARAMS**: Service Parameters (STRING) - Configuration parameters for the service
- **Input**: Input Identifier (logiBUS_DI_S) - Identifies the input (I1..I8)
- **InputEvent**: Event Identifier (logiBUS_DI_Events_S) - Identifies the event (currently only REPEAT supported)

Data Outputs

- **QO**: Event Output Qualifier (BOOL) - Service output status
- **STATUS**: Service Status (STRING) - Service status information
- **IN**: Input Data from the Resource (DWORD) - Current input data from the device

Adapter

No adapter interfaces are available.

Functionality

The logiBUS_ID function block acts as a bridge between the 4diac system and logiBUS input devices. Device parameters are configured during initialization (INIT). The block can process both synchronous queries (REQ/CNF) and asynchronous events (IND). Input data is provided as a 32-bit double word (DWORD).

Technical Features

- Supports double-word input data (32-bit)
- Uses specific logiBUS data types for input and event configuration
- Offers both synchronous and asynchronous communication modes
- Initial values for input parameters are defined as “Invalid”

State Overview

The function block has the following operating states:

- Not initialized (before INIT)
- Initialized and ready (after successful INITO)
- Query mode (during REQ/CNF cycle)
- Event mode (during IND signaling)

Application Scenarios

- Connecting logiBUS input devices in automation systems
- Processing digital input signals in control applications
- Integration of button inputs (BUTTON_PRESS_UP, BUTTON_LONG_PRESS_HOLD, BUTTON_LONG_PRESS_UP)
- Monitoring of state changes at digital inputs

⚖️ Comparison with Similar Blocks

Compared to standard digital input blocks, logiBUS_ID:

- Specific logiBUS integration
- Advanced event handling
- Configurable parameters via STRING parameters
- Asynchronous event indication

🛠️ Related exercises

- Exercise_011a2

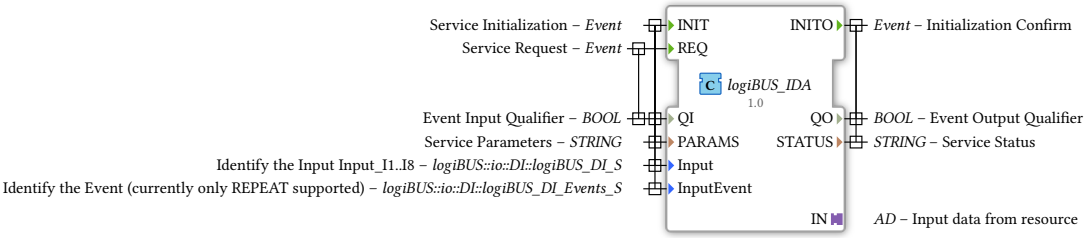
Conclusion

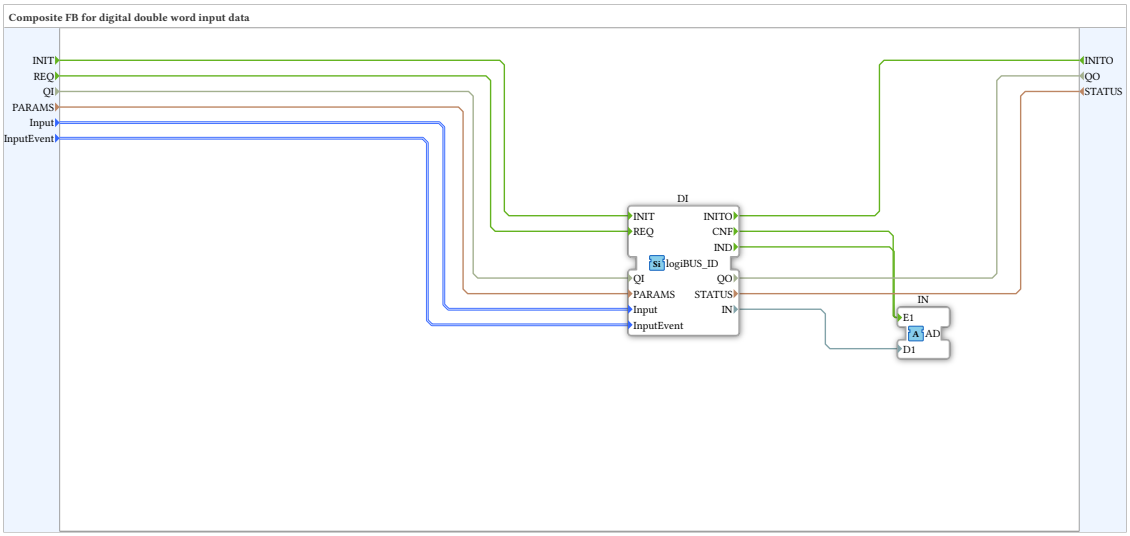
The logiBUS_ID function block represents a specialized interface for logiBUS input devices in 4diac systems. Its flexible configuration and support for various operating modes make it ideal for applications requiring reliable processing of digital input signals with advanced event handling.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

logiBUS_IDA





• – ○ • – ○ • – ○ •

Introduction

The **logiBUS_IDA** is a composite function block (CFB) for digital double-word input processing. It encapsulates an internal **logiBUS_ID** block and provides a uniform interface for initialization, parameterization, and event-driven data querying. The block is designed for use in logiBUS-based automation systems and supports the selection of an input channel as well as the configuration of the repeat event.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Service initialization; expects valid parameters (QI, PARAMS, Input, InputEvent)
REQ	Event	Service request; triggers a new data query at the internal function block

Event Outputs

Event	Type	Description
INITO	EInit	Initialization confirmation; returns the status (QO, STATUS)

Data Inputs

Variable	Type	Description
QI	BOOL	Qualifier for the event input; controls the execution

PARAMS	STRING	Service parameter (e.g., configuration string)
Input	logiBUS_DI_S	Selection of the digital input channel (I1...I8); Default value: *Invalid*
InputEvent	logiBUS_DI_Events_S	Selection of the event type (currently only *REPEAT* supported); Default value: *Invalid*

Data Outputs

Variable	Type	Description
QO	BOOL	Qualifier of the output event
STATUS	STRING	Service status (e.g., error messages or operating state)

Adapters

Adapter	Type	Description
IN	adapter::types::unidirectional::AD	Unidirectional input adapter for receiving digital double-word data from the resource

Functionality

The **logiBUS_IDA** is implemented as a composite block that encapsulates a single **logiBUS_ID** function block.

Initialization occurs via the **INIT** event. The parameters *QI*, *PARAMS*, *Input*, and *InputEvent* are passed to the inner block. The inner block then configures the selected digital input channel and prepares for data acquisition. After successful initialization, **INITO** is output with the values *QO* and *STATUS*.

A cyclic or one-time read operation is triggered via the **REQ** event. The internal block reads the current value of the configured input (8-bit/double word) via the **IN** adapter and makes the result available internally for further processing. The actual data values are not directly used as output variables of the **logiBUS_IDA** They are available exclusively via the **IN** adapter at the output of the internal function block (see FBNetwork connections: DI.IN ⟶ IN.D1).

Technical Features

- **Composite Architecture**: Encapsulates complex logic in a reusable function block.
- **Event-driven Processing**: Separation of initialization (INIT) and operational request (REQ).
- **Adapter-Based Input/Output**: Digital double-word information is transmitted via a unidirectional adapter, which keeps the coupling to the resource flexible.
- **Configurable Parameters**: The input channel (I1...I8) and the event behavior (e.g., REPEAT) can be set at runtime.
- **Standard-Compliant IEC 61499-2**: The function block is modeled according to the standard and uses type safety through predefined structures (logiBUS_DI_S and logiBUS_DI_Events_S).

State Overview

The function block goes through the following main states:

1. **Initialization (INIT)**

- Entry: Event **INIT** is activated.
- Action: Internal function block is parameterized and started.
- Output: **INITO** with QO/STATUS.

1. **Operation (REQ)**

- Entry: Event **REQ** is activated.
- Action: Data query at the internal function block; value is provided via adapter **IN**.
- Cyclic repetition is possible if *InputEvent* is set to *REPEAT*.

Error states (e.g., invalid parameters or connection problems) are signaled via **STATUS**.

Application Scenarios

- **Digital Multiple Input in Agricultural Technology**: Acquisition of up to 8 digital sensors (e.g., limit switches, proximity switches) and summarization as a double word.
- **Fieldbus Connection**: Use in logiBUS networks for decentralized I/O acquisition.
- **Configurable Monitoring**: Configuration of the sampling behavior (event type) according to the controller's requirements.
- **Extended Diagnostics**: The STATUS output allows feedback of errors or operating states to the higher-level controller.

Comparison with Similar Function Blocks

Compared to a simple **logiBUS_DI**, the **logiBUS_IDA** offers an extended interface with explicit initialization (INIT) and event-driven requests (REQ). While a pure **logiBUS_ID** function block directly delivers the raw data via adapters, the **IDA** integrates the configuration of the channel and event behavior in a compact block. Similar composite function blocks from other fieldbus systems (e.g., CANopen, PROFIBUS) often implement comparable functionalities close to the processor, while the **logiBUS_IDA** is specifically tailored to the logiBUS protocol environment.

Conclusion

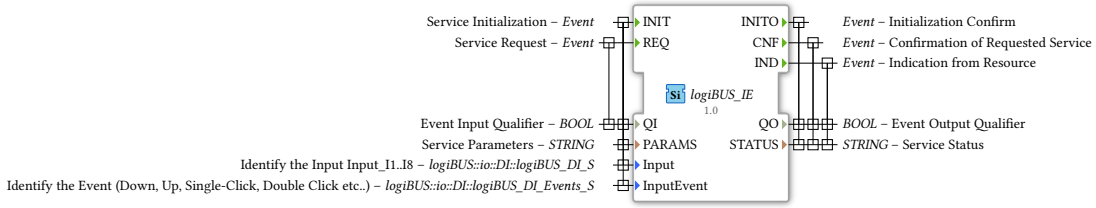
The **logiBUS_IDA** is a flexible composite function block for digital double-word input processing in IEC 61499 environments. It simplifies commissioning through its clear interface with initialization and request events and enables channel- and event-configured data acquisition. Thanks to adapter-based data transfer, it can be seamlessly integrated into existing logiBUS resources. The block is particularly suitable for applications requiring reliable initialization and repeated, parameterizable queries of digital inputs.

logiBUS_IE

• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_IE function block is an input service interface module for event input data. It serves as an interface between the logiBUS system and the 4diac runtime environment for processing digital input signals with extended event detection functions.



Interface Structure

Event Inputs

- **INIT**: Service Initialization
- Linked to: QI, PARAMS, Input, InputEvent
- **REQ**: Service Request
- Linked to: QI

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Services
- Linked to: QO, STATUS
- **IND**: Resource Indication
- Linked to: QO, STATUS

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameter
- **Input** (logiBUS_DI_S): Identifies the Input (I1..I8)

- Initial Value: Invalid
- **InputEvent** (logiBUS_DI_Events_S): Identifies the event (Down, Up, Single-Click, Double-Click, etc.)
- Initial value: Invalid

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (STRING): Service status

Adapter

No adapter interfaces available.

Functionality

This function block enables the processing of digital input signals with advanced event detection. Specific parameters can be configured via the INIT initialization. The block recognizes various input events such as key presses, releases, single clicks, and double clicks and signals these via the corresponding output events.

Technical Features

- Supports multiple event types (Down, Up, Single-Click, Double-Click)
- Flexible input configuration via PARAMS string
- Status feedback via STATUS output
- Initialization with invalid values for a safe start

State Overview

The function block has an initialized state and an operating state. After successful INIT initialization, it switches to the operating state, in which event processing and service requests can be handled.

Application Scenarios

- Operator Panel Controllers
- Keyboard Input Processing
- Mechanical Switch Monitoring
- User Interaction Systems
- Industrial Control Pushbuttons

⚖️ Comparison with Similar Function Blocks

Compared to simple digital input blocks, logiBUS_IE offers advanced event detection capabilities and can recognize complex input patterns such as double-clicks. The configurable design allows for flexible adaptation to various hardware interfaces.

🛠️ Related exercises

- Uebung_004a
- Uebung_004a2
- Uebung_004a2_2
- Uebung_004a2_3
- Uebung_004a2_AX
- Uebung_004a3
- Uebung_004a3_AX
- Uebung_004a4
- Uebung_004a4_AX
- Uebung_004a5
- Uebung_004a5_AX
- Uebung_004a6
- Uebung_004a6_AX
- Uebung_004a7
- Uebung_004a7_AX
- Uebung_004a8
- Uebung_004a8_AX
- Uebung_004a9
- Uebung_004a9_AX

- Uebung_004a_AX
- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_004c1
- Uebung_004c1_AX
- Uebung_004c2
- Uebung_004c2_AX
- Uebung_004c3
- Uebung_004c3_AX
- Uebung_004c4
- Uebung_004c4_AX
- Uebung_004c5
- Uebung_004c5_AX
- Uebung_006
- Uebung_006_AX
- Uebung_006a
- Uebung_006a2
- Uebung_006a2_AX
- Uebung_006a3
- Uebung_006a3_AX
- Uebung_006a4
- Uebung_006a4_AX
- Uebung_006a_AX
- Uebung_006b
- Uebung_006b_AX
- Uebung_006d
- Uebung_006d_AX
- Uebung_007a1
- Uebung_007a1_AX
- Uebung_007a2
- Uebung_007a2_AX
- Uebung_007a3
- Uebung_007a3_AX
- Uebung_009a
- Uebung_019
- Uebung_019a
- Uebung_019b
- Uebung_019c
- Uebung_020f3
- Uebung_020f3_AX
- Uebung_020h
- Uebung_020h_AX
- Uebung_020i
- Uebung_020i_AX
- Uebung_031
- Uebung_034b
- Uebung_035
- Uebung_035a
- Uebung_035a1_AX
- Uebung_035a1b_AX
- Uebung_035a2
- Uebung_035a2_AX
- Uebung_035a3
- Uebung_035a3_AX
- Uebung_035b

- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- Uebung_039
- Uebung_039a
- Uebung_040
- Uebung_040_2
- Uebung_040_AX
- Uebung_041
- Uebung_042
- Uebung_043
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_081
- Uebung_082
- Uebung_083
- Uebung_083_AX
- Uebung_084
- Uebung_085
- Uebung_087
- Uebung_087a1
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Uebung_094a
- Uebung_094a_AX
- Uebung_095
- Uebung_095_AX
- Uebung_110
- Uebung_111
- Uebung_124
- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_132
- Uebung_150_AX
- Uebung_151_AX
- Uebung_152
- Uebung_153
- Uebung_160b
- Uebung_160b_AX
- Uebung_171_AX
- Uebung_179
- Uebung_180

Conclusion

The logiBUS_IE function block provides a powerful interface for digital input devices Advanced event detection is available. Its flexible configurability and support for various event types make it ideal for applications requiring complex user interactions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

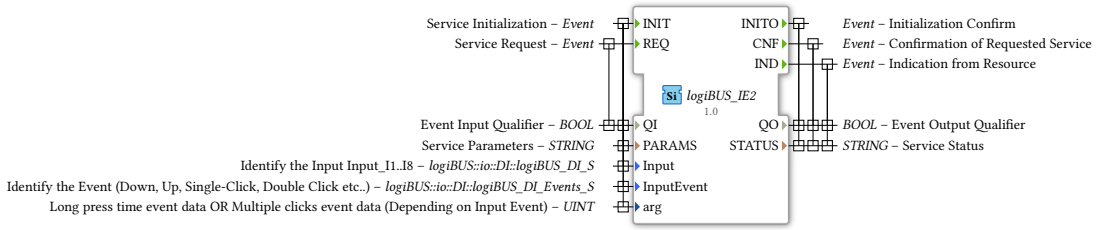
]

logiBUS_IE2

• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_IE2 is an input service interface function block for event input data. It serves as an interface for processing input events in logiBUS systems and enables the processing of various input event types such as key presses, clicks, and time-based events.



Interface Structure

Event Inputs

- **INIT** (Service Initialization) - Initializes the service with the following data:
- QI
- PARAMS
- Input
- InputEvent
- arg
- **REQ** (Service Request) - Requests a service with:
- QI

Event Outputs

- **INITO** (Initialization Confirm) - Confirms the initialization with:
- QO
- STATUS
- **CNF** (Confirmation of Requested Service) - Confirms the requested service with:
- QO
- STATUS
- **IND** (Indication from Resource) - Message from the resource with:
- QO
- STATUS

Data Inputs

- **QI** (BOOL) - Event Input Qualifier
- **PARAMS** (STRING) - Service Parameters
- **Input** (logiBUS_DI_S) - Identifies input Input_I1..I8 (Initial value: Invalid)
- **InputEvent** (logiBUS_DI_Events_S) - Identifies the event (Down, Up, Single-Click, Double-Click, etc.) (Initial value: Invalid)
- **arg** (UINT) - Long-press time event data OR multi-click event data (depending on the input event) (Initial value: 65535)

Data Outputs

- **QO** (BOOL) - Event Output Qualifier
- **STATUS** (STRING) - Service Status

Adapter

No adapter interfaces available.

Functionality

The logiBUS_IE2 function block processes input events from logiBUS inputs and converts them into standardized event messages. It supports various event types such as key presses (down/up), single clicks,

double clicks, and time-based events like long presses. The block can be configured via the INIT input and returns status information and acknowledgments via its outputs.

Technical Features

- Supports multiple inputs (I1 to I8)
- Processes various event types via the InputEvent variable
- Can process time-based events (long press) and count-based events (multiple clicks)
- Initialization with specific parameters via the PARAMS input
- Provides detailed status information via the STATUS output

State Overview

The function block has an initialization state (INIT/INITO) and operational states for service requests (REQ/CNF) as well as asynchronous event messages (IND). The exact state engine depends on the implementation.

Application Scenarios

- Operator panel controls with keypad input
- Machine control with event-based input
- Systems with complex keypad input sequences
- Applications with long-press and multi-click detection

⚖️ Comparison with similar modules

Compared to simple digital input modules, logiBUS_IE2 offers enhanced functionality for complex event detection and supports various input event types with configurable parameters.

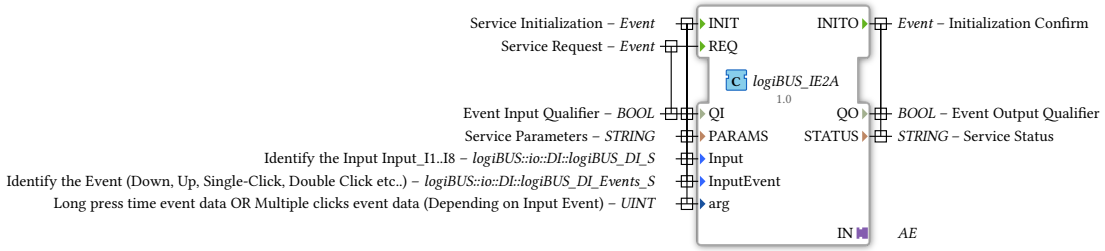
🛠️ Related Exercises

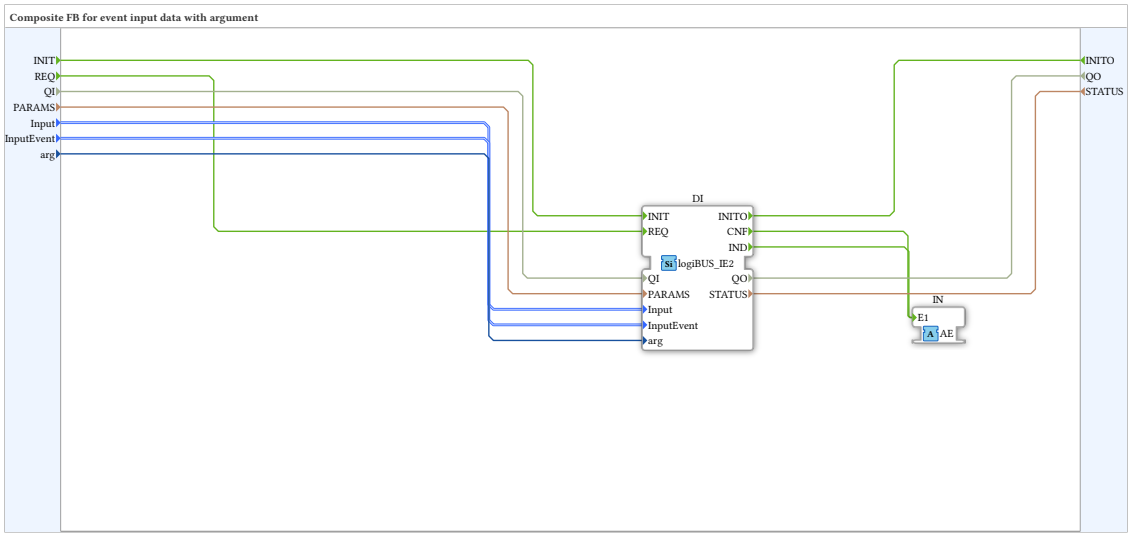
- Exercise_004c6
- Exercise_004c6_AX
- Exercise_004c7
- Exercise_004c7_AX

Conclusion

The logiBUS_IE2 is a powerful function block for processing event input data in logiBUS systems. Its flexibility in event detection and extensive configuration options make it suitable for demanding control applications.

logiBUS_IE2A





• – ○ • – ○ • – ○ •

Introduction

The function block `logiBUS_IE2A` is a composite block (FB) for processing events at digital inputs of the logiBUS system. It extends the functionality of the underlying block `logiBUS_IE2` by adding a dedicated adapter output for event messages and allows the transfer of parameters such as long press times or multi-click counters. The block serves as a standardized interface between the logiBUS hardware and a higher-level control logic.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Service Initialization
REQ	Event	Service Request

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Acknowledgement

Data Inputs

Name	Type	Comment	Initial Value
QI	BOOL	Event Input Qualifier	–
PARAMS	STRING	Service Parameter	–
Input	`logiBUS::io::DI::logiBUS_DI_S`	Digital input identification (e.g., Input_I1..I8)	`logiBUS_DI::Invalid`
InputEvent	`logiBUS::io::DI::logiBUS_DI_Events`	Event identification (Down, Up, Single-Click, Double-Click, etc.)	`logiBUS_DI_Events::Invalid`
arg	UINT	Event data: Long press time or number of multiple clicks (depending on the event)	65535

Data Outputs

Name	Type	Comment
QO	BOOL	Event output qualifier
STATUS	STRING	Service status

Adapters

Name	Type	Description
IN	`adapter::types::unidirectional::AE`	Unidirectional adapter output for events (E1)

Functionality

The `logiBUS_IE2A` is a purely structural block that encapsulates the logic of the internal block `logiBUS_IE2`.

- Upon the arrival of an **INIT** event, the data inputs `QI`, `PARAMS`, `Input`, `InputEvent`, and `arg` are forwarded to the internal function block, where initialization is performed. After successful processing, an acknowledgment containing the output data `QO` and `STATUS` is returned via the **INITO** output.
- A **REQ** event triggers a new request (e.g., for another measurement) without resetting the parameters.
- The actual event messages of the digital input are routed by the internal function block (FB) to the adapter output **IN.E1** via its outputs `IND` and `CNF`. This allows the higher-level application to react to incoming events without needing to know the internal details of the function block.

The data flows are hard-coded in the internal network and conform to the protocol of `logiBUS_IE2`.

Technical Features

- **Composite Structure**: The function block does not contain its own state machine, but delegates all logic to the internal FB `logiBUS_IE2`.
- **Predefined Initial Values**: The inputs `Input` and `InputEvent` are pre-assigned to `Invalid`, and `arg` to `65535`. This prevents misconfigurations as long as the parameters are not explicitly set.
- **Adapter Output**: Event output is provided via a unidirectional adapter (`AE`), which enables loose coupling to the calling application.
- **Type Imports**: The function block uses specific type definitions from the packages `logiBUS::io::DI` and `logiBUS::io::DI::logiBUS_DI_Events`, ensuring consistent use within the logiBUS ecosystem.

State Overview

The `logiBUS_IE2A` does not have its own state machine (no ECC). The internal function block `logiBUS_IE2` implements the necessary states for event detection and output. From the perspective of the composite function block, only the cyclic processes INIT → INITO and REQ → (event output via adapter) are relevant.

Application Scenarios

- **Button and Switch Evaluation**: Detection of single and multiple clicks as well as long presses on the digital inputs of a logiBUS module.
- **Configurable Event Detection**: The `arg` input allows for dynamic definition of thresholds for long press times or the number of multiple clicks.
- **Integration into Higher-Level Controllers**: The adapter output enables seamless integration with automation systems that communicate via adapter interfaces.

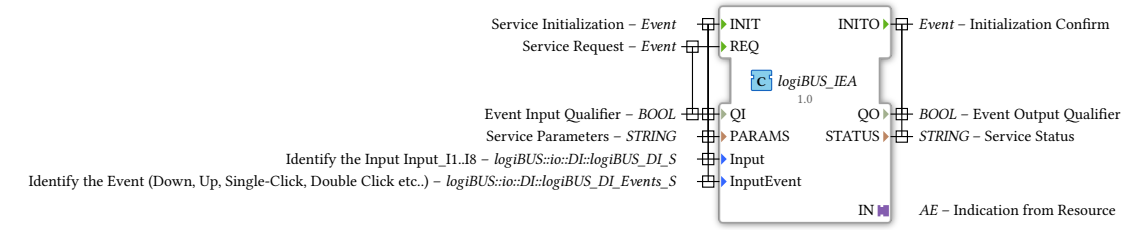
Comparison with Similar Function Blocks

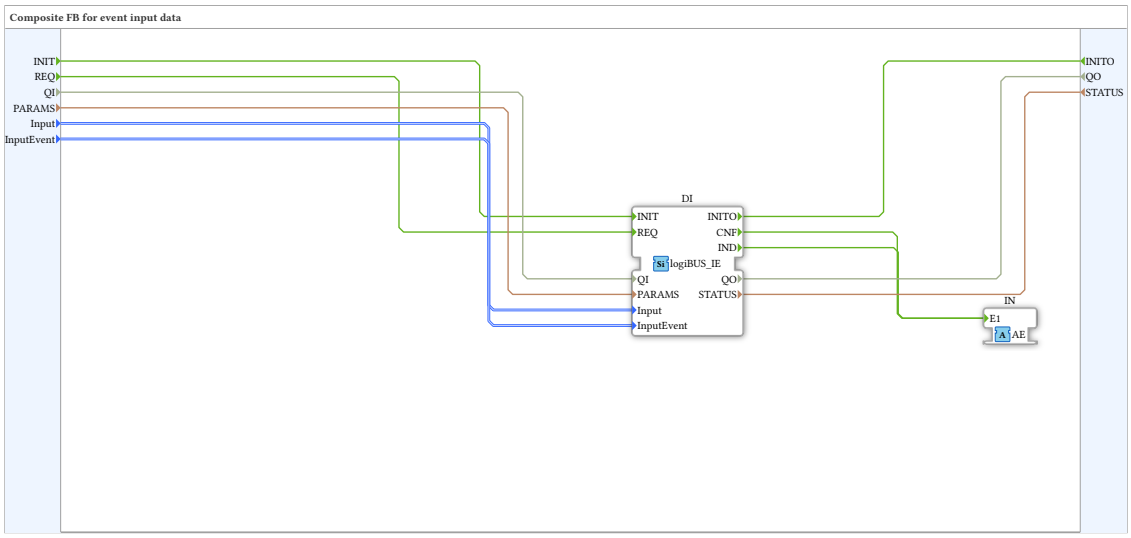
- **`logiBUS_IE2`**: This function block is used within `logiBUS_IE2A`. It offers the same functionality, but without the dedicated adapter output and without the aggregation of parameters in a composite structure.
- **`logiBUS_IE2S`** (if present): Another function block in the same family, which may differ in additional status outputs or other event selection.
- The **`logiBUS_IE2A`** is characterized by its simple configuration and standardized adapter interface, which facilitates reuse in different projects.

Conclusion

The `logiBUS_IE2A` is a pre-configured composite module for event-based processing of logiBUS digital inputs. It combines the proven logic of the `logiBUS_IE2` with an adapter output and configurable event data. Thanks to its clear interface and predefined initial values, it is particularly suitable for modular and standardized automation solutions where rapid integration and ease of use are paramount.

logiBUS_IEA





• — ○ • — ○ • — ○ •

Introduction

The composite function block `logiBUS_IEA` is used for event-driven processing of digital input data in a logiBUS fieldbus system. It encapsulates an internal `logiBUS_IE` block and makes its functionality available via a standardized interface. The function block enables the initialization and querying of events at a digital input and forwards the resulting indications to the resource via an adapter.

Interface Structure

Event Inputs

Name	Type	Comment	Data Carried
`INIT`	EInit	Service Initialization	`QI`, `PARAMS`, `Input`, `InputEvent`
`REQ`	Event	Service Request	`QI`

Event Outputs

Name	Type	Comment	Data Carried
`INITO`	EInit	Initialization Confirm	`QO`, `STATUS`

Data Inputs

Name	Type	Comment	Initial Value

QI	BOOL	Event Input Qualifier	–
PARAMS	STRING	Service Parameters	–
Input	logiBUS::io::DI::logiBUS_DI_S	Identifies the input (e.g., Input_I1..I8)	logiBUS_DI::Invalid
InputEvent	logiBUS::io::DI::logiBUS_DI_Events_S	Identifies the event (e.g., Down, Up, Single-Click, Double-Click)	logiBUS_DI_Events::Invalid

Data Outputs

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

Adapters

Name	Type	Comment
IN	adapter::types::unidirectional::AE	Indication from Resource

The adapter IN receives the events generated by the internal function block (DI.IND and DI.CNF) and forwards them as an "Indication" to the higher-level resource.

Functionality

The function block logiBUS_IEA operates as a pure composite function block, meaning it has no control over its own flow. Instead, it forwards all events and data to the embedded function block logiBUS::io::DI::logiBUS_IE.

- **INIT Event**:

The internal function block is initialized via the event input INIT. The parameters QI, PARAMS, Input, and InputEvent are passed. After successful initialization, the internal function block acknowledges with the event INITO, which is returned to the higher-level function block. The output data QO and STATUS are updated in the process.

- **REQ Event**:

With REQ (if QI is set), a request is sent to the internal function block. This block processes the configuration and triggers the corresponding actions when events are detected.

- **Indication via Adapter**:

The events IND (Indication) and CNF (Confirm) generated by the internal function block are routed to the adapter output IN.E1. This allows the surrounding resource to react to events such as button presses or double-clicks.

Technical Features

- **Composite Function Block**: The function block does not contain its own ECC (Execution Control Chart) but delegates all logic to the internal logiBUS_IE. This increases reusability and testability.
- **Generic Parameterization**: System-specific parameters can be passed as strings via the PARAMS input.
- **Adapter-Based Event Output**: The use of a unidirectional adapter (type AE) enables flexible coupling to the resource without hardwiring to event outputs.
- **Typed Input Selection**: The inputs Input and InputEvent utilize specific enumeration types (logiBUS_DI_S and logiBUS_DI_Events_S) that allow for secure and unambiguous configuration.

State Overview

Since logiBUS_IEA is a pure composite component, it does not have its own state machine. The state is determined by the internal component. Typical states of the internal logiBUS_IE include:

- **IDLE**: Waiting for INIT or configuration

- **WAIT**: Waiting for a detected event at the input
- **ACTIVE**: Event detected, indication sent
- **ERROR**: Initialization or communication error

The state transitions are indicated by the events `IND` and `CNF` and communicated externally via the adapter.

Application Scenarios

- **Event Detection at Digital Inputs**: Pushbuttons, switches, or sensors with multi-click functionality (single/double click) can be configured via `InputEvent`.
- **Machine Mode Control**: A digital input triggers a mode change when activated (e.g., a down event).
- **Configurable Initialization**: Bus parameters or filter times can be passed to the logiBUS controller via `PARAMS`.
- **Integration into Larger Composite Blocks**: The adapter `IN` allows connection to a higher-level control block (e.g., a resource manager).

Comparison with Similar Blocks

FB Type	Property
`logiBUS_DI`	Pure digital input without event detection; simple level retrieval
`logiBUS_IEA`	**This FB:** Event-based input with initialization and adapter for indication
`logiBUS_DI_Events`	Event Detection on a Single Input, but Without Initialization Logic

Compared to a simple digital input, the `logiBUS_IEA` offers a complete initialization interface and standardized event signaling via an adapter. Compared to a pure event block, it also includes the INIT functionality required for commissioning a logiBUS device.

Conclusion

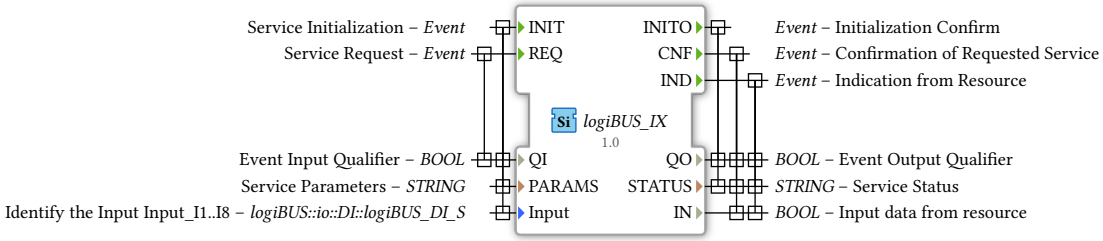
The `logiBUS_IEA` composite block encapsulates the complex initialization and event processing of a logiBUS input module in a compact, reusable unit. Thanks to the clear separation of initialization and request processing, support for typed inputs, and adapter-based event output, it is particularly well-suited for use in modular automation solutions where reliable event-driven input acquisition is required.

logiBUS_IX

• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_IX function block is an input service interface for Boolean input data, specifically designed for communication with logiBUS input modules. It serves as an interface between the control logic and physical input signals, enabling the querying of digital input values.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Request
- **IND**: Display from Resource Interface

Data Inputs

- **QI**: Event Input Qualifier (BOOL)
- **PARAMS**: Service Parameters (STRING)
- **Input**: Identifies Input I1..I8 (logiBUS_DI_S) - Initial Value: Invalid

Data Outputs

- **QO**: Event Output Qualifier (BOOL)
- **STATUS**: Service Status (STRING)
- **IN**: Input Data from Resource (BOOL)

Adapter

No adapter interfaces are available.

Functionality

The function block initializes itself via the INIT event and can then request input data via the REQ event. Upon successful initialization, it returns the INITO event. The actual input values are provided via the IN output, while the STATUS output provides information about the current operating state.

Technical Features

- Supports up to 8 digital inputs (I1..I8)
- Uses special logiBUS data types for input identification
- Provides comprehensive status feedback via the STATUS output
- Initializes with an invalid input value (Invalid)

State Overview

The function block typically goes through the following states:

1. **Not Initialized**: Before the INIT request
2. **Initialized**: After successful INIT processing
3. **Ready**: Can process REQ requests
4. **Active**: Currently processing a service request

Application Scenarios

- Querying digital input signals in automation systems
- Integrating logiBUS input modules into 4diac-based controllers
- Monitoring switch states and sensor signals
- Industrial I/O control with status monitoring

⚖️ Comparison with Similar Blocks

Compared to simple digital input blocks, this offers logiBUS_IX:

- Advanced status information
- Parameterizable service parameters
- Specific logiBUS hardware integration
- More comprehensive initialization and confirmation mechanisms

🛠️ Related exercises

- Uebung_001
- Uebung_001c
- Uebung_002
- Uebung_002a
- Uebung_002a2
- Uebung_002a3
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b2
- Uebung_002b3
- Uebung_003
- Uebung_003a0

- Uebung_003a_sub
- IX_TO_Aux_QX
- Uebung_003d
- Uebung_005
- Uebung_006e1
- Uebung_006e2
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c
- Uebung_020c2
- Uebung_020c3
- Uebung_020d
- Uebung_020e
- Uebung_020e2
- Uebung_020f
- Uebung_020f2
- Uebung_020g
- Uebung_020i
- Uebung_028
- Uebung_029
- Uebung_030
- Uebung_032
- IX_TO_logiBUS_LED_strip_QX
- Uebung_049
- Uebung_051
- Uebung_052
- Uebung_053
- Uebung_054
- Uebung_055
- Uebung_056
- Uebung_085
- Uebung_086
- Uebung_087
- Uebung_087a1
- Uebung_087a2
- Uebung_088
- Uebung_089
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2
- Uebung_090a2_AX
- Exercise_094
- Exercise_094a
- Exercise_095
- Exercise_160
- Exercise_160b2
- Exercise_177
- Exercise_178

Conclusion

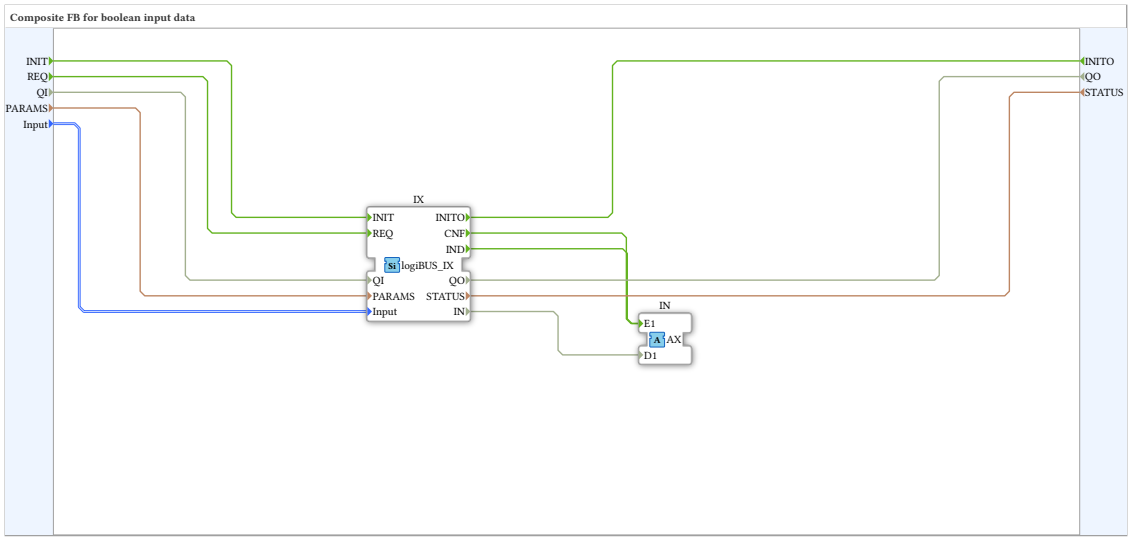
The logiBUS_IX function block represents a robust and flexible solution for integrating logiBUS input modules into 4diac-based control systems. Its comprehensive status feedback and configurable interface make it particularly suitable for industrial applications where reliable I/O communication is required.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

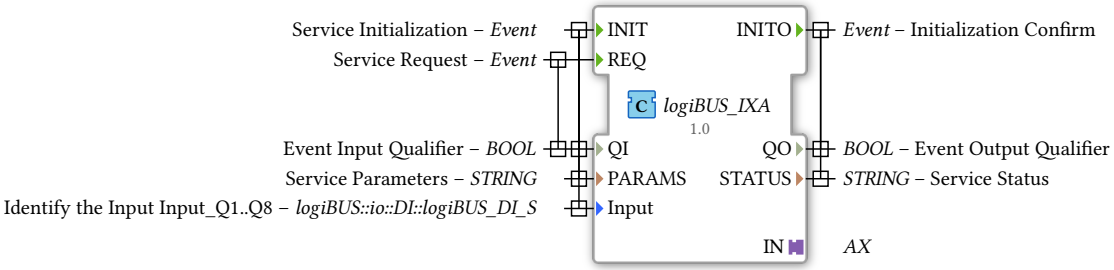
logiBUS_IXA



• – ○ • – ○ • – ○ •

Introduction

The logiBUS_IXA is a composite function block for processing Boolean input data. It serves as an interface for digital inputs and enables the initialization and querying of input signals via standardized service interfaces.



Interface Structure

Event Inputs

- **INIT**: Service initialization event with the associated data QI, PARAMS, and Input
- **REQ**: Service request event with the associated data value QI

Event Outputs

- **INITO**: Initialization confirmation with the associated data values QO and STATUS

Data Inputs

- **QI**: Boolean event input qualifier
- **PARAMS**: Service parameters as a string
- **Input**: Identifies inputs Q1 to Q8 with type logiBUS_DI_S and initial value “Invalid”

Data Outputs

- **QO**: Boolean event output qualifier
- **STATUS**: Service status as a string

Adapters

- **IN**: Unidirectional AX-type adapter for input data processing

Functionality

The logiBUS_IXA composite function block encapsulates the logiBUS_IX base function block and extends its functionality with additional adapter interfaces. Upon INIT events, parameters are forwarded to the internal IX block, which performs the initialization. IND events from the IX block are passed to the IN input adapter, while the data connections ensure the corresponding signal processing.

Technical Features

- Uses the specific data type logiBUS_DI_S for input identification
- Implements standardized service interfaces according to DIN 61499-2
- Supports parameter passing via the STRING data type
- Provides status feedback via the STATUS output

State Overview

The function block has two main states:

1. **Not initialized**: Before the INIT event
2. **Initialized and ready for operation**: After successful INIT confirmation

Application Scenarios

- Connecting digital inputs in automation systems
- Integration into logiBUS-based control architectures
- Use in PLC systems with Boolean signal processing requirements

⚖️ Comparison with Similar Blocks

Compared to simple digital input blocks, logiBUS_IXA offers extended service functionalities with parameterization options and status feedback. The composite structure enables better reusability and enhanced diagnostic capabilities.

🛠️ Related exercises

- Uebung_001_AX
- Uebung_001_AX_b
- Uebung_001c_AX
- Uebung_002_AX
- Uebung_002a2_AX
- Uebung_002a3_AX
- Uebung_002a5_AX
- Uebung_002a5b_AX
- Uebung_002a6_AX
- Uebung_002a7_AX
- Uebung_002a_AX
- Uebung_002b3_AX
- Uebung_003_AX
- Uebung_003a0_AX
- IXA_TO_Aux_QXA
- Uebung_003d_AX
- Uebung_005_AX
- Uebung_006e1_AX
- Uebung_006e2_AX
- Uebung_020a_AX
- Uebung_020b_AX
- Uebung_020c3_AX
- Uebung_020c_AX
- Uebung_020d_AX
- Uebung_020e2_AX
- Uebung_020e_AX
- Uebung_020f2_AX
- Uebung_020f_AX
- Uebung_020g_AX
- Uebung_020i_AX

- Uebung_020j2_AX
- Uebung_020j_AX
- Uebung_090a1_AX
- Uebung_090a2_AX
- Uebung_094a_AX
- Uebung_095_AX
- Uebung_103
- Uebung_103c
- Uebung_103c2
- Uebung_160_AX
- Uebung_160b2_AX
- Exercise_177_AX
- Exercise_178_AX

Conclusion

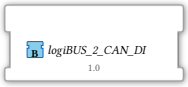
The logiBUS_IXA function block represents a robust and flexible solution for processing digital input signals in 4diac-based automation systems. Thanks to its standardized interfaces and extended service functions, it is particularly suitable for complex applications with high demands on diagnostics and parameterization.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

DI_CAN

logiBUS_2_CAN_DI



• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_2_CAN_DI GlobalConstants define constants for configuring digital inputs (DIs) in the logiBUS system via CAN bus. These constants are used for pin assignment for digital input channels and provide a standardized interface for hardware configuration.

Interface Structure

Constant Definitions

The GlobalConstants define three main constants for the digital input configuration:

- **Input_CI1**: Configuration for digital input channel 1
- **Input_CI2**: Configuration for digital input channel 2
- **Invalid**: Indicates an invalid or unconfigured input

Data Types

All constants use the structured data type `logiBUS_2_CAN_DI_S`, which contains the following component:

- **Pin**: Integer value for identifying the physical input pin

Functionality

The constants are not modified at runtime and are used exclusively for configuring the digital inputs. They define the hardware mapping between logical channels and physical pins in the CAN-based logiBUS system.

Technical Features

- **Pin Assignment**: Channel 1 is assigned to Pin 1, Channel 2 is assigned to Pin 2
- **Invalid Identifier**: The value 255 indicates an invalid or unavailable input

- **CAN Integration**: Specifically optimized for CAN-based logiBUS systems
- **Type Safety**: Use of structured data types for consistent configuration

Application Scenarios

- Configuration of digital inputs in logiBUS controllers
- Hardware abstraction in CAN-based automation systems
- Pin assignment in distributed I/O systems
- Error handling for unconfigured inputs

⚖️ Comparison with Similar Function Blocks

Compared to simple numeric constants, these structured constants offer:

- Higher type safety through structured data types
- Improved maintainability through centralized configuration
- Extensibility for additional parameters
- Compatibility with logiBUS system architecture

Conclusion

The logiBUS_2_CAN_DI GlobalConstants offer a robust and type-safe solution for configuring digital inputs in CAN-based logiBUS systems. Their structured approach enables clear hardware mapping and simplifies the maintenance and expansion of automation solutions.

logiBUS_2_CAN_DI_S

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `logiBUS_2_CAN_DI_S` is a structured data type (struct) used to configure a digital input (DI) for a logiBUS-CAN system. It defines the basic parameters required to address and use a physical input pin on a logiBUS-CAN module. This data type typically serves as input for higher-level function blocks that handle the actual communication via the CAN protocol.

Interface Structure

This is a structured data type (struct) and therefore has no event or adapter interfaces. Its structure consists of a single data element.

Data Inputs

Since it is a data type, these elements are used as input data within a higher-level function block.

Name	Data Type	Initial Value	Comment
`Pin`	`USINT` (Unsigned Short Integer)	255	Defines the number of the physical input pin on the logiBUS CAN module.

Data Outputs

(Not applicable for a structured data type)

Event Inputs

(Not applicable for a structured data type)

Event Outputs

(Not applicable for a structured data type)

Adapters

(Not applicable for a structured data type)

Functionality

The data type `logiBUS_2_CAN_DI_S` functions as a configuration container. The value `Pin` specifies which particular digital input of a logiBUS CAN I/O module is to be addressed. The initial value `255` represents a typical invalid or unconfigured state, requiring the user to explicitly assign a valid pin number. A higher-

level function block (e.g., a driver function block) reads this structure and uses the `Pin` information to generate or interpret the correct CAN telegrams that query the state (on/off) of the respective input.

Technical Features

- **Data type `USINT`:** The use of `USINT` (0 to 255) for the pin suggests that the logiBUS hardware can theoretically address up to 256 digital inputs per configuration structure. The value 255 is reserved as the default initial value.
- **Initial value 255:** This value serves as an indicator for an inactive or undefined configuration. It should be replaced with a valid pin number in the application.
- **Package name:** The data type is organized in the package `logiBUS::io::DI_CAN`, which clearly indicates its association with logiBUS CAN input communication.
-

State Overview

(Not applicable for a structured data type. The "state" is simply the stored value of the `Pin` variable.)

Application Scenarios

This structure type is used in scenarios where digital signals (e.g., switches, limit switches, sensor contacts) are to be read into a controller (e.g., an IEC 61499-based application in 4diac) via a CAN-based logiBUS network. Typical applications include:

- Machine controllers with decentralized, CAN-networked sensors.
- Mobile applications (vehicles, robots) with a robust bus system.
- Systems where digital input signals are distributed over long distances.
-

⚖️ Comparison with Similar Function Blocks

Unlike direct I/O function blocks (such as `E_SR` or `E_D_FF`), which process logical states, `logiBUS_2_CAN_DI_S` is a pure **configuration data type**. It is comparable to other structured hardware configuration types in 4diac (e.g., for serial interfaces or other fieldbuses) that encapsulate parameters for hardware connectivity. Its counterpart for digital outputs would be a corresponding struct type (e.g., `logiBUS_2_CAN_DO_S`).

Conclusion

The `logiBUS_2_CAN_DI_S` data type is an essential and streamlined configuration element for connecting digital inputs to a CAN-based logiBUS system within the 4diac IDE. By encapsulating the pin number in a structured variable, it enables clear, type-safe, and maintainable configuration of the hardware interface in higher-level control algorithms. Using a unique initial value helps to detect misconfigurations early.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

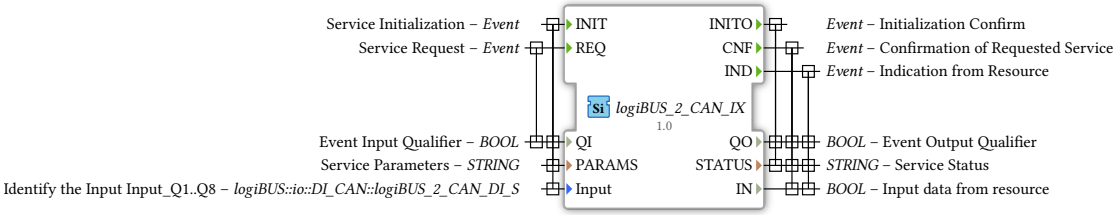
]

logiBUS_2_CAN_IX

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `logiBUS_2_CAN_IX` is a service interface function block for Boolean input data, specifically designed for integrating logiBUS systems with CAN bus communication. It serves as an input interface for digital signals and enables communication between the control system and connected CAN-enabled devices.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- Linked to: QI, PARAMS, Input
- **REQ**: Service Request Event
- Linked to: QI

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Service Request
- Linked to: QO, STATUS, IN
- **IND**: Indication from Resource
- Linked to: QO, STATUS, IN

Data Inputs

- **QI**: BOOL - Event Input Qualifier
- **PARAMS**: STRING - Service Parameter
- **Input**: logiBUS_2_CAN_DI_S - Identifies the Input Input_Q1..Q8
- Initial value: logiBUS_2_CAN_DI::Invalid

Data Outputs

- **QO**: BOOL - Event output qualifier
- **STATUS**: STRING - Service status
- **IN**: BOOL - Input data from the resource

Adapter

No adapter interfaces available.

Functionality

This function block operates as an input service interface for digital CAN signals. During initialization (INIT event), the service parameters are configured and the input channel is identified. Service requests can be made via the REQ event, while IND events signal incoming data from the resource. The block manages the communication status and outputs the received Boolean input data via the IN output.

Technical Features

- Special data types for logiBUS-CAN integration
- Initialization with invalid state as standard
- Separate acknowledgment and indication events for different operating states
- String-based status feedback for detailed fault diagnosis

State Overview

The function block goes through the following main states:

1. **Not Initialized**: Before INIT event
2. **Initialized**: After successful INIT acknowledgment
3. **Service Ready**: Can process REQ and IND events
4. **Fault State**: In case of communication problems (signaled via STATUS)

Application Scenarios

- Integration of logiBUS I/O modules into 4diac controllers
- CAN-based digital input signal processing
- Industrial automation systems with distributed I/Os
- Robot controllers with CAN peripherals

⚖️ Comparison with Similar Blocks

Compared to standard digital input blocks, this offers `logiBUS_2_CAN_IX` features dedicated CAN integration and enhanced status feedback. While simple DI blocks only offer direct signal processing, this block supports more complex protocol handling and error handling for industrial CAN networks.

Conclusion

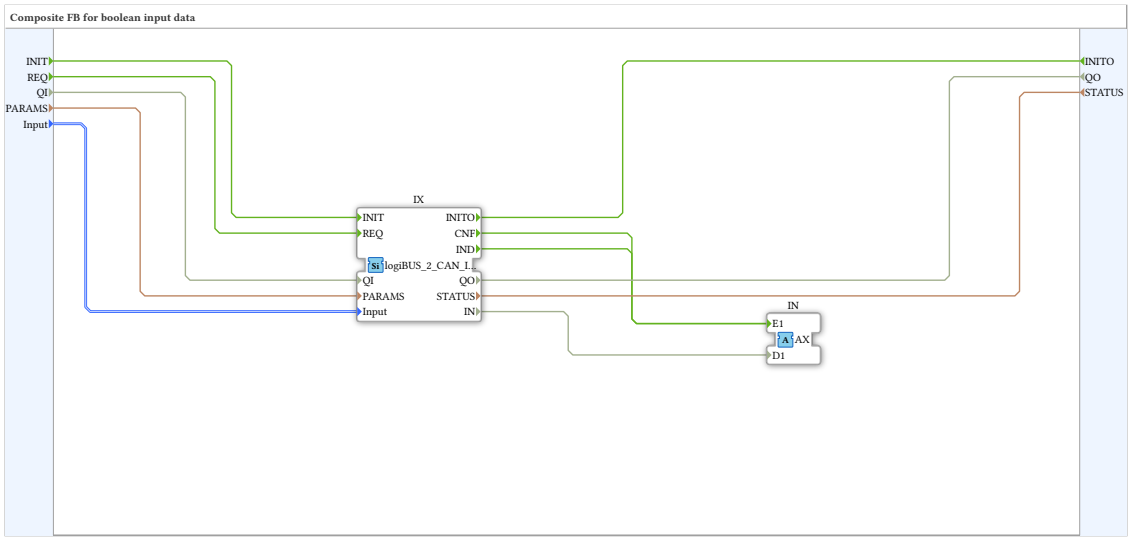
The `logiBUS_2_CAN_IX` function block provides a robust and reliable interface for integrating CAN-based digital input signals into 4diac control systems. Its enhanced error handling and status feedback make it particularly suitable for demanding industrial applications where reliable communication with distributed I/O systems is required.

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

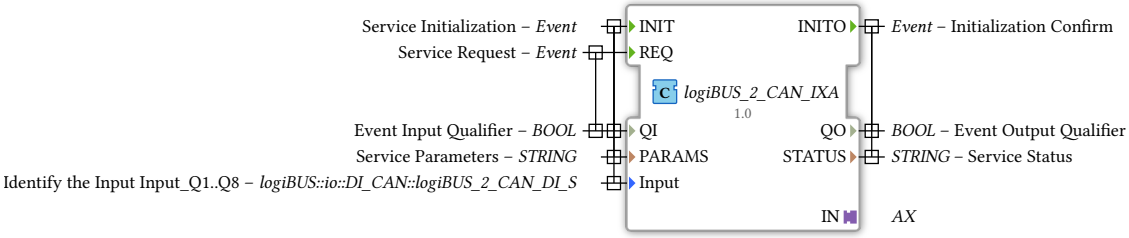
logiBUS_2_CAN_IXA



• – ○ • – ○ • – ○ •

Introduction

The logiBUS_2_CAN_IXA is a composite function block for processing Boolean input data via CAN bus interfaces. The block serves as an interface between logiBUS systems and CAN-based input devices and enables the initialization and querying of digital inputs.



Interface Structure

Event Inputs
- **INIT**: Service Initialization (with QI and PARAMS)
- **REQ**: Service Request (with QI)

Event Outputs
- **INITO**: Initialization Acknowledgement (with QO and STATUS)

Data Inputs
- **QI**: Event Input Qualifier (BOOL)
- **PARAMS**: Service Parameters (STRING)
- **Input**: Input Identification Input_Q1..Q8 (logiBUS_2_CAN_DI_S)

Data Outputs
- **QO**: Event Output Qualifier (BOOL)
- **STATUS**: Service Status (STRING)

Adapters
- **IN**: Unidirectional AX-type adapter for the Data Communication

Functionality
The composite function block internally uses the logiBUS_2_CAN_IX module to process CAN-based input data. During INIT initialization, the service parameters are passed and the module is configured. Input data can be queried via the REQ event. Processing is performed via the IN adapter, which enables communication with CAN input devices.

Technical Features
- Uses specific logiBUS data types for CAN communication
- Supports up to 8 digital inputs (Input_Q1 to Input_Q8)
- Initial value for input is “Invalid”
- Integrated error handling via STATUS output

State Overview
The function block goes through the following states:

1. **Not Initialized**: Before INIT call
2. **Initialization**: During INIT processing
3. **Ready**: After successful initialization (INITO confirmed)
4. **Query**: During REQ processing

Application Scenarios
- Connecting CAN-based sensors and switches to logiBUS systems
- Industrial automation with digital input signals
- Control systems in agricultural technology
- Distributed I/O systems via CAN bus

⚖️ Comparison with Similar Function Blocks
Compared to simple digital input function blocks, this offers logiBUS_2_CAN_IXA:

- CAN bus communication capability
- Extended parameterization options
- Integrated status feedback
- Specific logiBUS integration

Conclusion
The logiBUS_2_CAN_IXA is a specialized composite function block for the reliable integration of CAN-based digital inputs into logiBUS systems. Its structured initialization and query mechanisms make it particularly suitable for industrial applications requiring robust and configurable I/O interfaces.

DO

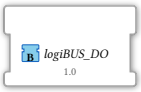
🎧 Podcast

- R_TRIG & F_TRIG: How PLCs reliably detect signal edges – without double processing
- Adolf Weber's sharp analysis: Why technology alone isn't enough – Lessons from three industrial phases
- Bulldog legend: How the simple LANZ tractor revolutionized agriculture and became a standard term in the dictionary
- Bulldog, boom, and business: Heinrich Lanz – Rise from trading house to agricultural machinery empire (The underestimated role of family)
- [The tracked monster awakens: Lanz Bulldog Caterpillar – The fascinating revival of the 10-liter hot-bulb workhorse after 25 years] standstill](https://podcasters.spotify.com/pod/show/ms-muc-lama/episodes/Das-Kettenmonster-erwacht-Lanz-Bulldog-Raupe–Die-faszinierende-Wiederbelebung-des-10-Liter-Glhkopf-Arbeitstiers-nach-25-Jahren-Stillstand-e39arpd)

📺 Video

- The downfall of Lanz

logiBUS_DO



• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_DO GlobalConstants define a collection of constants for configuring digital outputs (DOs) in the logiBUS system. These constants are used for pin assignment to digital output channels and provide a standardized interface for hardware configuration.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

No adapters available

Functionality

The logiBUS_DO GlobalConstants define structured constants of type `logiBUS_DO_S`, each containing a specific pin value for digital outputs. Constants `Output_Q1` to `Output_Q10` represent the available output channels with the corresponding pin numbers 1 to 10. The constant `Invalid` with the pin value 255 serves as an indicator for invalid or unconfigured outputs.

Technical Features

- **Constant Type**: All constants are of the structured data type `logiBUS_DO_S`
- **Pin Mapping**: Clear mapping of output names to physical pin numbers
- **Invalid Marking**: Special constant with pin value 255 for error handling
- **Package Structure**: Organized in the package `logiBUS::io::DQ`

State Overview

Since these are constants, there are no state transitions. The values are static and unchanging during runtime.

Application Scenarios

- Hardware configuration of logiBUS systems
- Pin assignment in function blocks for digital outputs
- Error handling for invalid output configurations
- Standardization of output designations in control applications

⚖️ Comparison with similar function blocks

Compared to other constant definitions, logiBUS_DO offers a specific structure for digital outputs with clear pin assignment and a defined invalid constant for robust error handling.

Conclusion

The logiBUS_DO GlobalConstants provide a standardized and maintainable solution for configuring digital outputs in logiBUS systems. The structured definition of the output channels and the provided invalid constant enable reliable and fault-tolerant hardware configuration.

logiBUS_DO_S

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `logiBUS_DO_S` is a structured data type (struct) used to configure a digital output (DO) within the logiBUS system. It defines the necessary parameters to uniquely address and control a physical output pin of a logiBUS I/O module. This data type is typically used as input for function blocks that provide the actual output functionality.

Interface Structure

As a structured data type, `logiBUS_DO_S` does not have event or adapter interfaces in the conventional sense of a function block. Instead, it defines a data structure that is passed as a whole to other blocks.

Data Inputs

The structure itself contains the following data field, which is populated when the type is instantiated:

Name	Data Type	Initial Value	Description
`Pin`	`USINT` (Unsigned Short Integer)	255	Defines the number of the physical output pin to be controlled. The value 255 represents an invalid/undefined state and serves as the default initialization.

Event Inputs

Not present. This type is a data structure only.

Event Outputs

Not present. This type is a data structure only.

Data Outputs

Not present. This type is a data structure only.

Adapters

Not present. This type is a data structure only.

Functionality

The `logiBUS_DO_S` struct acts as a configuration container. Its main task is to transport the information about the desired pin number (`Pin`) in a bundled and typed manner. A higher-level function block (e.g., a `logiBUS_DO` block) receives an instance of this struct, reads the `Pin` value, and uses this information to switch the corresponding physical output of the logiBUS hardware module.

The initial value of 255 for `Pin` indicates that the pin is not yet configured. Valid pin numbers are within the range supported by the respective logiBUS I/O module (e.g., 0-7 for an 8-channel module).

Technical Features

- **Typing:** Using a structured data type instead of a simple `USINT` increases type safety and application readability. It is clear that the value represents a logiBUS output pin.
- **Initial Value:** The initial value of 255 serves as a "Safe State" indicator. Before the structure is correctly configured, this value indicates an inactive or faulty state.
- **Package Assignment:** The struct is part of the `logiBUS::io::DQ` package, indicating its membership in the digital input/output (Digital I/O) of the logiBUS framework.

State Overview

Since it is a passive data type, `logiBUS_DO_S` does not have an internal state machine. The "state" is determined solely by the stored value of the `Pin` variable.

Application Scenarios

1. **Output Configuration:** This struct is primarily used to parameterize function blocks that control lamps, valves, relays, or other binary actuators via logiBUS modules.
2. **Structured Parameter Passing:** In more complex applications, multiple `logiBUS_DO_S` instances can be organized in arrays or other structures to centrally manage the configuration of all outputs in a station.
3. **Initialization:** The default initial value allows variables of this type to be declared without having to populate them immediately. The actual configuration (assigning a valid pin number) can be performed later in the program flow.

⚖️ Comparison with Similar Building Blocks

- **Simple `USINT`:** Directly using a `USINT` value for the pin number is possible, but offers less semantic clarity and type safety. The `logiBUS_DO_S` makes the purpose of the data element explicit.
- **`logiBUS_DI_S`:** This is the structural counterpart for digital inputs (DIs). It has an identical structure (a `USINT` field `Pin`), but conceptually belongs to the input side. Separating DO and DI structs improves code readability and maintainability.
-

🛠️ Related exercises

- Uebung_003a_sub
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010c2
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- SR_TO_QX_TRIG_EO
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG

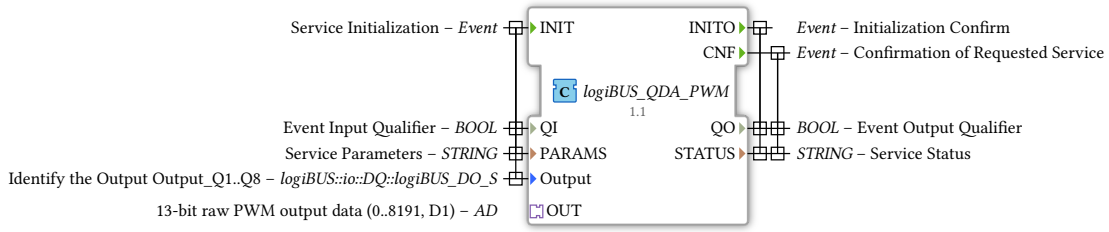
Conclusion

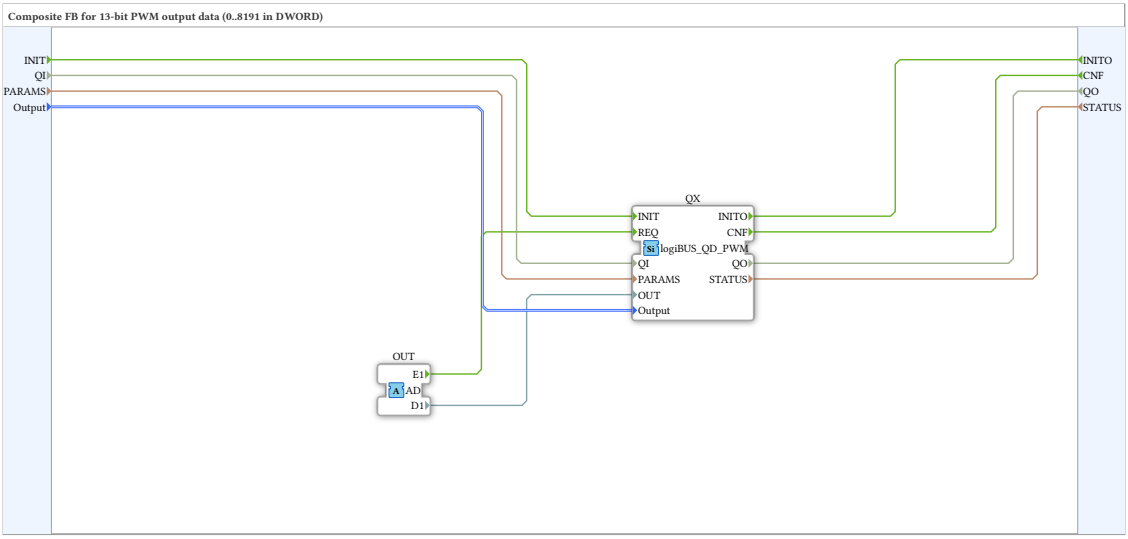
The The `logiBUS_DO_S` struct is a simple yet essential helper element in the logiBUS ecosystem of the 4diac IDE. It standardizes and formalizes the addressing of digital output pins, promotes type-safe programming, and, thanks to its defined initial value, serves as the basis for robust initialization logic. Its use is recommended when configuring logiBUS library function blocks for digital outputs.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

logiBUS_QDA_PWM





• – ○ • – ○ • – ○ •

Introduction

The **logiBUS_QDA_PWM** function block is a composite block for controlling a PWM (Double Word) output via a logiBUS system. It encapsulates the initialization and triggering of an internal PWM output block and provides an adapter interface for receiving commands (events and data) from external sources. The block was developed for use in agricultural technology and is licensed under EPL 2.0.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Service Initialization
no further event input		*All triggers are handled via the adapter*

Event Outputs

Event	Type	Comment
INITO	EInit	Initialization Acknowledgement
CNF	Event	Acknowledgement of a requested service

Data Inputs

Variable	Type	Comment
QI	BOOL	Input Qualifier (Enable Initialization)
PARAMS	STRING	Service Parameter (e.g., Addressing, Configuration)

Output	logiBUS::io::DQ::logiBUS_DO_S	Output Identification (e.g., Output_Q1..Q8); Initial Value: *Invalid*

Data Outputs

Variable	Type	Comment
QO	BOOL	Output Qualifier (Initialization Status)
STATUS	STRING	Service Status (Failure/Success Message)

Adapters

Type	Name	Direction	Comment
adapter::types::unidirectional::AD	OUT	Socket	Receives trigger event (E1) and output data (D1) from the resource

Functionality

The function block is implemented as a composite and contains an internal instance of the function block `logiBUS::io::DQ::logiBUS_QD_PWM` (referred to here as *QX*). The logic can be described as follows:

1. **Initialization:**

An INIT event at the input triggers the initialization of the internal function block. The data inputs *QI*, *PARAMS*, and *Output* are then forwarded to the internal function block. After successful initialization, the event *INITO* is output, along with the output data *QO* and *STATUS*.

1. **Triggering via Adapter:**

The adapter *OUT* receives an external event *E1* and a data value *D1*. The event is forwarded to the internal function block as a *REQ* (Request), and the data *D1* as an *OUT* value. The internal function block processes this request and acknowledges it with the event *CNF*, which is then output to the external system via the *CNF* output. The associated output data *QO* and *STATUS* are updated.

This function block allows for a clean separation of initialization and cyclic output: Configuration is performed once via INIT, while the actual PWM output is triggered by the adapter.

Technical Features

- **Composite Function Block:** The function block encapsulates all the logic of a PWM output function block and provides a standardized adapter interface for data exchange with the resource.
- **13-Bit PWM Output (`0–8191`):** The adapter data element *D1* (`DWORD`) expects a 13-bit raw PWM value (`0` = 0% duty cycle, `8191` = 100% duty cycle, $2^{13} = 8192$ states).
- **Initialization Parameters:** Flexible configuration data, necessary for addressing or parameterizing the logiBUS module, can be passed via *PARAMS* (STRING).
- **Error Handling:** The *STATUS* output provides a textual description of the service status (e.g., error messages for invalid configurations).

State Overview

Since this is a composite function block without its own state machine, the state is defined by the internal function block *logiBUS_QD_PWM*. Typical states are:

- **IDLE:** Waiting for initialization or a trigger.
- **INITIALIZING:** During initialization (INIT received, INITO not yet sent).
- **OPERATIONAL:** Ready for cyclic triggers (via adapter).
- **ERROR:** Error state, indicated by *QO* = *FALSE* or *STATUS* with an error message.

The internal function block switches between these states depending on the events and data.

Application Scenarios

- **Agricultural Machinery:** Control of PWM-driven actuators (e.g., hydraulic valves, engine speed) via a logiBUS network.
- **Automation Systems:** Output of pulse-width modulated signals with 13-bit resolution (`0–8191`), controlled by a higher-level controller.

- **Remote Maintenance and Configuration:** The adapter allows new PWM values to be sent from a higher-level resource (e.g., HMI or PLC) without repeating the initialization process.

Comparison with Similar Function Blocks

Function Block	Type	Special Feature
logiBUS_QD_PWM	Composite/Basic	Directly controllable via INIT, REQ, CNF; without an adapter interface.
logiBUS_QDA_PWM	Composite	Like logiBUS_QD_PWM, but with an adapter for external triggering and data supply.
logiBUS_DO (simple)	Basic	Simple digital output, no PWM function.

The function block described here offers greater flexibility, as the actual output via the adapter can occur asynchronously from initialization. A disadvantage is the dependence on the correct provision of the adapter signals.

Conclusion

The **logiBUS_QDA_PWM** function block is a practical encapsulation of a PWM output for logiBUS. By combining INIT-based configuration and adapter-based triggering, it is particularly suitable for applications requiring one-time parameterization followed by cyclic or event-driven output. The use of standardized types and the clear separation of interfaces facilitate integration into IEC 61499-based control systems.

Related topic subpages on ms-muc-docs.de

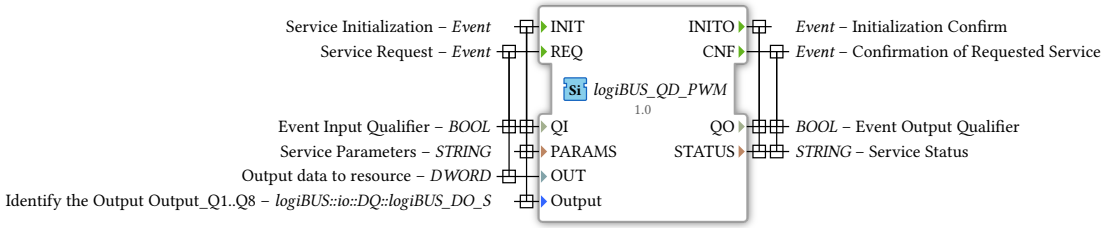
- The PWM signal & infographic on ms-muc-docs.de

logiBUS_QD_PWM

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `logiBUS_QD_PWM` is an output service interface function block for double-word output data. It serves as an interface for controlling PWM (pulse-width modulation) outputs via the logiBUS system and enables the control of outputs Q1 to Q8.



Interface Structure

Event Inputs

- **INIT**: Service Initialization
- Linked to: QI, PARAMS, Output
- **REQ**: Service Request
- Linked to: QI, OUT

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **CNF**: Acknowledgement of Requested Service Operation
- Linked to: QO, STATUS

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **OUT** (DWORD): Output Data for the Resource (13-bit raw PWM value `0` to `8191`, corresponding to `0 %` to `100 %` duty cycle, $2^{13} = 8192$ states)
- **Output** (logiBUS_DO_S): Identifies the output Output_Q1..Q8
- Initial Value: `logiBUS_DO::Invalid`

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status

Adapters

No adapter interfaces available.

Functionality

This function block enables PWM control of outputs via the logiBUS system. During initialization (INIT), the service parameters are configured and the specific output is identified. PWM data (`OUT`, 13-bit value `0` to `8191` inside a DWORD variable) can be sent to the configured output via a REQ request. The block acknowledges both initialization and service requests via the corresponding output events.

Technical Features

- **13-Bit PWM Resolution**: Uses 13-bit normalization (`0` to `8191`) stored in a `DWORD` data type (`0` = 0% duty cycle, `8191` = 100% duty cycle, scaling factor for ISO-Designer: `0.0122085215480405` or `100 / 8191`).
- Supports up to 8 outputs (Q1-Q8) via output configuration
- String-based parameter configuration for flexible service settings
- Status feedback via STRING variable for detailed error information

State Overview

The function block has two main states:

1. **Not Initialized**: Block waits for an INIT event
2. **Initialized and Ready**: Block can process REQ requests and output PWM data

Application Scenarios

- Control of PWM-controlled actuators (motors, heating elements)
- Control of LED lighting with brightness control
- Control of valves with proportional control
- Industrial automation applications with logiBUS hardware

⚖️ Comparison with Similar Blocks

Compared to simple digital output blocks, `logiBUS_QD_PWM` offers precise PWM control with 13-bit resolution (`0–8191` inside a DWORD). Compared to analog output blocks, it enables direct PWM control without additional conversion.

🛠️ Related exercises

- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_034b
- Uebung_152
- Uebung_153

Conclusion

The `logiBUS_QD_PWM` function block provides a powerful interface for PWM outputs in the logiBUS system. Thanks to its flexible configuration and support for 13-bit PWM data (`0–8191`), it is ideally suited for precise control applications in industrial automation systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 The PWM signal & infographic on ms-muc-docs.de

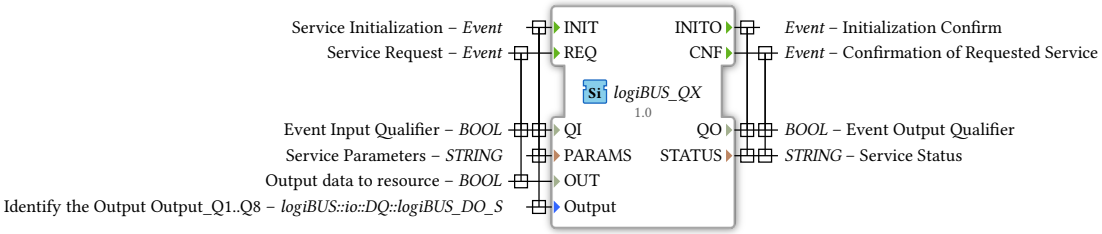
]

logiBUS_QX

• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_QX function block is an output service interface block for Boolean output data. It serves as an interface for controlling digital outputs via the logiBUS system and enables communication between the control application and the physical output channels.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **OUT** (BOOL): Output Data for Resource
- **Output** (logiBUS_DO_S): Identifies the Output (Output_Q1..Q8), Initial Value: Invalid

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status Information

Adapters

No adapter interfaces available.

Functionality

The logiBUS_QX block manages communication with the digital output channels of the logiBUS system. During initialization (INIT event), the service parameters are configured and the specific output channel is identified. Boolean values (OUT) can be sent to the configured output channel via the REQ event. The block acknowledges each operation via the corresponding output events INITO and CNF.

Technical Features

- Uses specific logiBUS data types for output identification
- Supports up to 8 digital output channels (Q1-Q8)
- Provides detailed status feedback via the STATUS output
- Initializes outputs with an invalid state (Invalid)

State Overview

The block goes through the following main states:

1. **Not Initialized**: Before the first INIT operation
2. **Initialized**: After successful INIT operation, ready for REQ operations
3. **Active**: During the processing of REQ operations

Application Scenarios

- Control of digital actuators (relays, valves, lamps)
- Connection to logiBUS hardware outputs
- Automation systems with distributed I/O components
- Safety-relevant switching operations with status feedback

⚖️ Comparison with Similar Blocks

Compared to standard digital output blocks, logiBUS_QX offers:

- Specific Integration in logiBUS systems
- Advanced status feedback
- Parameterizable output configuration
- Higher level of abstraction for hardware access

🛠️ Related exercises

- Uebung_001
- Uebung_001c
- Uebung_002
- Uebung_002a
- Uebung_002a2
- Uebung_002a3
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b2
- Uebung_002b3
- Uebung_003
- Uebung_003a0
- Uebung_003a_sub
- Uebung_003d
- Uebung_004a
- Uebung_004a2
- Uebung_004a2_2
- Uebung_004a2_3
- Uebung_004a3
- Uebung_004a4
- Uebung_004a5
- Uebung_004a6
- Uebung_004a7
- Uebung_004a8
- Uebung_004a9
- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_004c1
- Uebung_004c2
- Uebung_004c3
- Uebung_004c4
- Uebung_004c5
- Uebung_004c6
- Uebung_004c7
- Uebung_005
- Uebung_006
- Uebung_006a
- Uebung_006a2
- Uebung_006a3

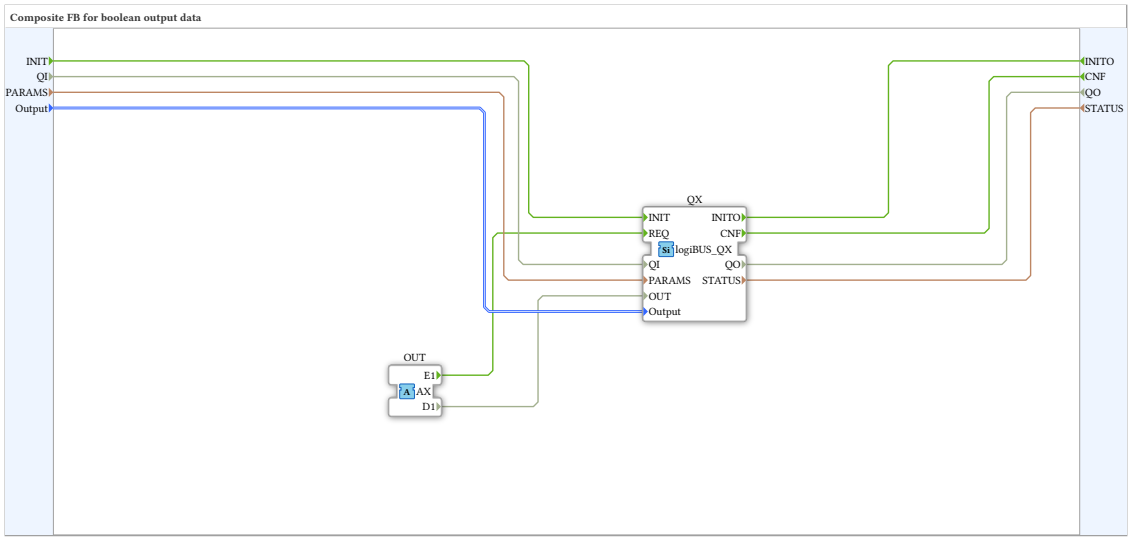
- Uebung_006a4
- Uebung_006b
- Uebung_006c
- Uebung_006d
- Uebung_006e1
- Uebung_006e2
- Uebung_007
- Uebung_007a1
- Uebung_007a2
- Uebung_007a3
- Uebung_008
- Uebung_009
- Uebung_010
- Uebung_010a
- Uebung_010a2
- Uebung_010a3
- Uebung_010b1
- Uebung_010b2
- Uebung_010b3
- Uebung_010b6
- Uebung_010b7
- Uebung_010b8
- Uebung_010b9
- Uebung_010bA
- Uebung_010bA2
- Uebung_010bA3
- Uebung_010bA4
- Uebung_010c
- Uebung_010c2
- Uebung_013
- Uebung_019b
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c
- Uebung_020c2
- Uebung_020c3
- Uebung_020d
- Uebung_020e
- Uebung_020e2
- Uebung_020f
- Uebung_020f2
- Uebung_020f3
- Uebung_020g
- Uebung_020h
- Uebung_020i
- Uebung_021
- Uebung_022
- Uebung_023
- Uebung_024
- Uebung_025
- SR_TO_QX_TRIG_EO
- Uebung_028
- Uebung_035
- Uebung_035a
- Uebung_035a2
- Uebung_035a3
- Uebung_035b
- Uebung_035c

- Uebung_036
- Uebung_037
- Uebung_038
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_039b
- Uebung_040
- Uebung_040_2
- Uebung_041
- Uebung_049
- Uebung_051
- Uebung_052
- Uebung_053
- Uebung_054
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_071
- Uebung_071a
- Uebung_071b
- Uebung_072b
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_081
- Uebung_082
- Uebung_083
- Uebung_084
- Uebung_085
- Uebung_087
- Uebung_087a1
- Uebung_087a2
- Uebung_088
- Uebung_089
- Uebung_090a1
- Uebung_090a2
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Uebung_094a
- Uebung_095
- Uebung_110
- Uebung_111
- Uebung_160
- Uebung_160b
- Uebung_160b2
- Uebung_177
- Uebung_178
- Uebung_179
- Uebung_180

Conclusion

The logiBUS_QX function block represents a robust and reliable interface for controlling digital outputs in logiBUS-based automation systems. Its clear event structure and comprehensive status feedback make it suitable It is particularly suitable for applications that require high reliability and traceability of output operations.

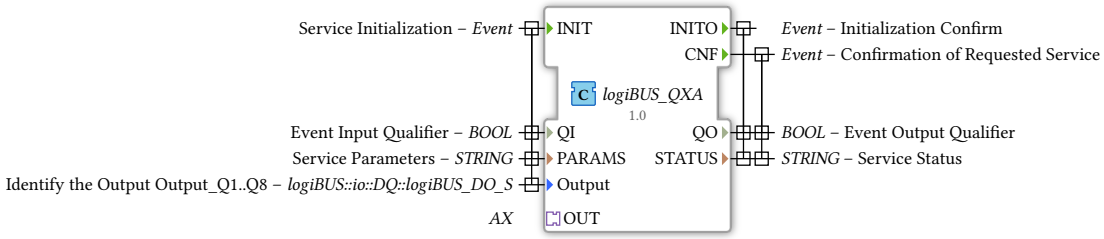
logiBUS_QXA



• – ○ • – ○ • – ○ •

Introduction

The logiBUS_QXA is a composite function block for outputting Boolean data. It serves as an interface for digital output functions and enables the control of up to 8 digital outputs via a standardized protocol.



Interface Structure

Event Inputs

- **INIT**: Service initialization event with the associated data QI, PARAMS, and Output

Event Outputs

- **INITO**: Initialization acknowledgment with the data QO and STATUS
- **CNF**: Acknowledgement of the requested service with the data QO and STATUS

Data Inputs

- **QI**: Boolean event input qualifier
- **PARAMS**: Service parameters as a string
- **Output**: Identification of outputs Q1 to Q8 of type logiBUS_DO_S with initial value “Invalid”

Data Outputs

- **QO**: Boolean event output qualifier
- **STATUS**: Service status as a string

Adapters

- **OUT**: Unidirectional adapter of type AX For output communication

Functionality

The logiBUS_QXA acts as a wrapper for the logiBUS_QX function block and provides a unified interface for digital output functions. During initialization (INIT event), the configuration parameters are passed

and the outputs are configured accordingly. The block enables the control of up to 8 digital outputs via the output data structure.

Technical Features

- Uses the logiBUS_QX core function block for the actual output logic
- Supports up to 8 digital outputs (Q1 to Q8)
- Initialization with specific parameters via PARAMS input
- Feedback on the operating status via STATUS output

State Overview

The function block goes through the following states:

1. **Not Initialized**: Before the INIT event
2. **Initialization**: During processing of the INIT event
3. **Ready for Operation**: After successful initialization (INITO confirmation)
4. **Active Operation**: Processing of output requests via the OUT adapter

Application Scenarios

- Control of digital actuators in automation systems
- Connection of output modules in distributed control systems
- Integration into logiBUS-based control architectures
- Industrial automation with Boolean output signals

⚖️ Comparison with Similar Blocks

Compared to simple logiBUS_QXA offers the following features for digital output blocks:

- Extended parameterization options
- Status feedback for fault diagnosis
- Standardized interface via adapter
- Support for multiple outputs in a single structure

🛠️ Related Exercises

- Exercise_001_AX
- Exercise_001_AX_b
- Exercise_001c_AX
- Exercise_002_AX
- Exercise_002a2_AX
- Exercise_002a3_AX
- Uebung_002a5_AX
- Uebung_002a5b_AX
- Uebung_002a6_AX
- Uebung_002a7_AX
- Uebung_002a_AX
- Uebung_002b3_AX
- Uebung_003_AX
- Uebung_003a0_AX
- Uebung_003d_AX
- Uebung_004a2_AX
- Uebung_004a3_AX
- Uebung_004a4_AX
- Uebung_004a5_AX
- Uebung_004a6_AX
- Uebung_004a7_AX
- Uebung_004a8_AX
- Uebung_004a9_AX
- Uebung_004a_AX
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_004c1_AX

- Uebung_004c2_AX
- Uebung_004c3_AX
- Uebung_004c4_AX
- Uebung_004c5_AX
- Uebung_004c6_AX
- Uebung_004c7_AX
- Uebung_005_AX
- Uebung_006_AX
- Uebung_006a2_AX
- Uebung_006a3_AX
- Uebung_006a4_AX
- Uebung_006a_AX
- Uebung_006b_AX
- Uebung_006d_AX
- Uebung_006e1_AX
- Uebung_006e2_AX
- Uebung_007_AX
- Uebung_007a1_AX
- Uebung_007a2_AX
- Uebung_007a3_AX
- Uebung_008_AX
- Uebung_009_AX
- Uebung_010_AX
- Uebung_010a2_AX
- Uebung_010a3_AX
- Uebung_010a_AX
- Uebung_010b1_AX
- Uebung_010b2_AX
- Uebung_010b3_AX
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010b6_AX
- Uebung_010b7_AX
- Uebung_010b8_AX
- Uebung_010b9_AX
- Uebung_010bA2_AX
- Uebung_010bA3_AX
- Uebung_010bA4_AX
- Uebung_010bA_AX
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- Uebung_010c_AX
- Uebung_013_AX
- Uebung_020a_AX
- Uebung_020b_AX
- Uebung_020c3_AX
- Uebung_020c_AX
- Uebung_020d_AX
- Uebung_020e2_AX
- Uebung_020e_AX
- Uebung_020f2_AX
- Uebung_020f3_AX
- Uebung_020f_AX
- Uebung_020g_AX
- Uebung_020h_AX
- Uebung_020i_AX
- Uebung_020j2_AX
- Uebung_020j_AX
- Uebung_035a1_AX

- Uebung_035a1b_AX
- Uebung_035a2_AX
- Uebung_035a3_AX
- Uebung_038_AX
- Uebung_040_AX
- Uebung_083_AX
- Uebung_090a1_AX
- Uebung_090a2_AX
- Uebung_094a_AX
- Uebung_095_AX
- Uebung_103
- Uebung_103c
- Uebung_103c2
- Uebung_150_AX
- Uebung_151_AX
- Uebung_160_AX
- Uebung_160b2_AX
- Uebung_160b_AX
- Uebung_171_AX
- Exercise_177_AX
- Exercise_178_AX

Conclusion

The logiBUS_QXA is a robust and flexible function block for digital output functions in industrial automation systems. Its standardized interface and comprehensive parameterization options make it ideally suited for use in complex control architectures with high demands on reliability and diagnostic capabilities.

DO-LED

LED_COLOURS



• – ◦ • – ◦ • – ◦ •

Introduction

LED_COLOURS is a global constant definition that provides standardized color values for LED displays. These constants are used in logiBUS systems for the uniform control of colored LED outputs.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

No adapters available

Functionality

LED_COLOURS defines a collection of global constants of type UINT (16-bit unsigned integer) that represent different LED colors. Each color is assigned a fixed numerical value:

- LED_RED: 1
- LED_YELLOW: 2
- LED_ORANGE: 3
- LED_GREEN: 4
- LED_BLUE: 5
- LED_WHITE: 6
- LED_MAGENTA: 7
- LED_CYAN: 8

Technical Specifications

- All constants are of type UINT (16-bit unsigned integer)
- Value range: 1-8 for the defined colors
- Constants are set at compile time and are immutable
- Part of the logiBUS::io::DO_LED package

State Overview

Since these are static constants, no state model exists

Application Scenarios

- Color coding in machine controllers
- Status indicators with colored LEDs
- Standardization of color control in distributed systems
- Visualization of operating states in industrial plants

⚖️ Comparison with similar Building Blocks

Unlike function blocks with dynamic behavior, these are purely static constant definitions. Similar constructs would be:

- ENUM data types for color definitions
- #define preprocessor instructions in C/C++
- Constant blocks in other IEC 61131 languages

Conclusion

LED_COLOURS offers a standardized and maintainable solution for controlling the color of LED displays in logiBUS systems. Centralized definition avoids inconsistencies and increases code reusability.

LED_FREQ



• – ○ • – ○ • – ○ •

Introduction

LED_FREQ is a collection of global constants for controlling LED blinking patterns with priority management. The constants define various blinking frequencies and operating modes for LEDs with assigned priority values that can be used in control applications.

Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

No adapters available

Functionality

The global constants provide numerical values for different LED operating modes. Each mode has a specific priority value, with higher numbers representing a higher priority. The constants can be used in function blocks to control LED behavior and resolve priority conflicts.

Technical Features

- **Data Type**: All constants are of type UINT (unsigned integer)
- **Priority Range**: Values from 0 to 6, where 6 represents the highest priority
- **Frequency Ranges**: 5 Hz, 2 Hz, 1 Hz, 0.5 Hz blinking and breathing effect
- **Package Membership**: logiBUS::io::DO_LED

State Overview

No state machine present - these are static constants

Application Scenarios

- Control of status LEDs in industrial control systems
- Prioritized display control in machine operator panels
- Visualization of operating states with varying urgency
- Multi-channel LED control with priority management

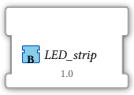
⚖️ Comparison with Similar Function Blocks

Unlike function blocks with dynamic behavior control, LED_FREQ offers static, predefined operating modes with fixed priorities. This allows for simpler implementation with known, standardized display requirements.

Conclusion

LED_FREQ offers an efficient method for defining standardized LED operating modes with integrated priority management. The constant collection is particularly suitable for applications requiring clear, predictable LED behavior patterns with defined priorities.

LED_strip



• – ○ • – ○ • – ○ •

Introduction

LED_strip is a GlobalConstants function block that provides global constants for controlling LED strips. The block defines an output variable for controlling LED strips and is part of the logiBUS system.

Interface Structure

Event Inputs

No event inputs available.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

No adapters available.

Functionality

The LED_strip block defines a global constant `Output_strip` of type USINT (Unsigned Short Integer) with an initial value of 0. This constant serves as a reference for controlling the output of LED strips and can be used by other function blocks in the system.

Technical Details

- The constant `Output_strip` is of type USINT (8-bit unsigned integer)
- Initial value is 0
- Part of the logiBUS::io::DO_LED package
- Supports the Eclipse Public License 2.0

State Overview

Since this is a GlobalConstants block, there are no state transitions or management. The defined constant retains its fixed value throughout runtime.

Application Scenarios

- Definition of output channels for LED strips
- Central management of hardware output references
- Use in combination with digital output blocks
- Integration into lighting control systems

⚖️ Comparison with similar blocks

Compared to other GlobalConstants blocks, LED_strip offers a specific constant for LED strip control. While general constant blocks define universal values, this block is designed for the specific requirements of LED control.

Conclusion

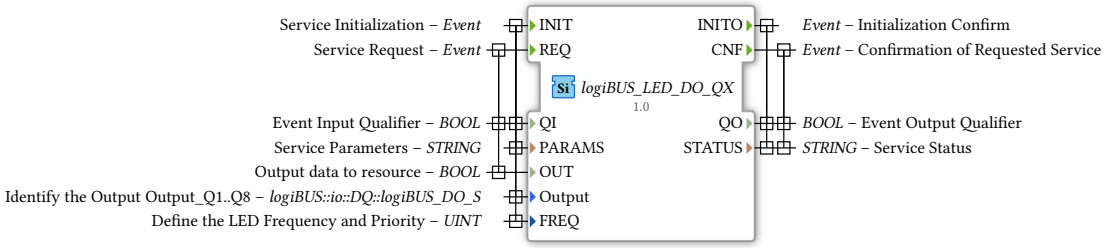
The LED_strip GlobalConstants block provides a simple and effective way to centrally manage output references for LED strips. By using it as a global constant, it enables consistent and maintainable integration into larger control systems.

logiBUS_LED_DO_QX

• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_LED_DO_QX function block is an output service interface function block for Boolean output data. It is used to control LED outputs via the logiBUS system and offers special functions for frequency control of the LEDs.



Interface Structure

Event Inputs
- **INIT**: Service initialization with the associated variables QI, PARAMS, Output, and FREQ
- **REQ**: Service request with the associated variables QI and OUT

Event Outputs
- **INITO**: Initialization acknowledgment with the associated variables QO and STATUS
- **CNF**: Acknowledgement of requested service execution with the associated variables QO and STATUS

Data Inputs
- **QI**: Event input qualifier (BOOL)
- **PARAMS**: Service parameters (STRING)
- **OUT**: Output data to the resource (BOOL)
- **Output**: Identifies the output Output_Q1..Q8 (logiBUS::io::DQ::logiBUS_DO_S) - Initial value: Invalid
- **FREQ**: Defines the LED frequency and priority (UINT) - Initial value: LED_FREQ::LED_OFF

Data Outputs
- **QO**: Event output qualifier (BOOL)
- **STATUS**: Service status (STRING)

Adapter
No adapter interfaces available.

Functionality
This function block allows the control of LED outputs with configurable frequency settings. The INIT event input initializes the service, allowing configuration of the specific output (Output_Q1 to Q8) and the LED frequency. The REQ event input triggers the actual output operation, sending the Boolean value to the configured output.

Technical Features
- Supports frequency control for LED operation (blinking)
- Initialization with invalid value for the output
- Predefined frequency constants (LED_FREQ::LED_OFF as default)
- Specific output identification via the output parameter

State Overview
The function block cycles through typical service interface states:

1. Uninitialized state
2. Initialization phase after an INIT event
3. Ready for operation after successful initialization
4. Active operating state during REQ processing

Application Scenarios
- Control of status LEDs in automation systems
- Visualization of process states with blinking signals
- Integration into logiBUS-based control systems
- Priority-controlled LED displays

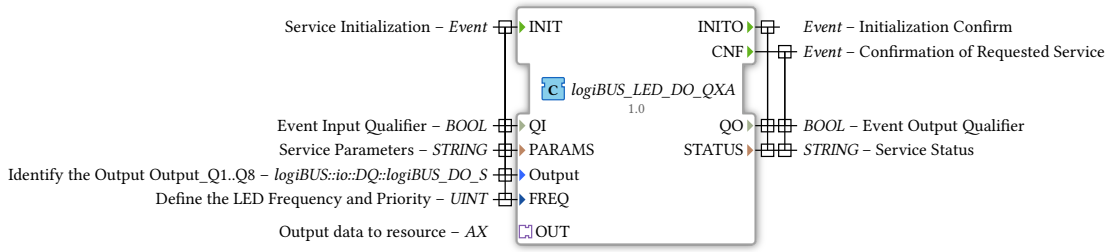
⚖️ Comparison with similar function blocks
Compared to simple digital output blocks, logiBUS_LED_DO_QX offers advanced frequency control functions and specific LED optimizations. While standard DO blocks only offer simple on/off control, this block enables more complex blinking patterns and priority control.

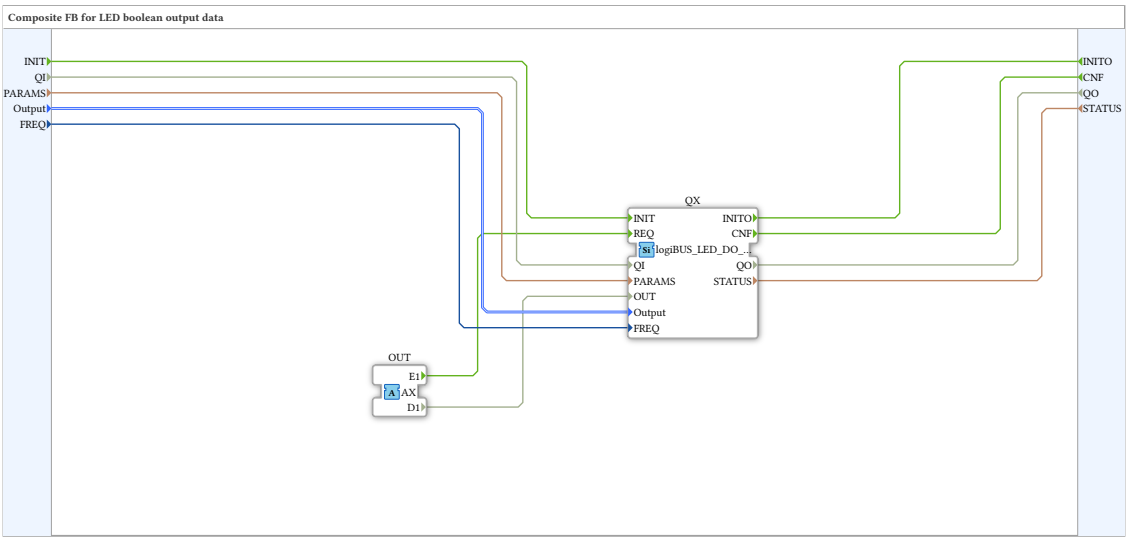
🛠️ Related Exercises
- Exercise_029

Conclusion

The logiBUS_LED_DO_QX function block represents a specialized solution for LED output control in logiBUS systems. Thanks to its integrated frequency control and flexible output configuration, it is particularly suitable for applications requiring advanced visualization functions.

logiBUS_LED_DO_QXA





• – ○ • – ○ • – ○ •

Introduction

The **logiBUS_LED_DO_QXA** is a composite function block (FB) that simplifies the control of a single LED via the logiBUS system. It receives a Boolean value (e.g., from a control logic) and configures the desired LED output (Q1–Q8) with an adjustable frequency and priority. The FB encapsulates the communication with the underlying hardware driver and provides a standardized interface.

Interface Structure

Event Inputs

- **INIT** (EInit): Initializes the entire function logic.

Wired data: `QI`, `PARAMS`, `Output`, `FREQ`

Event Outputs

- **INITO** (EInit): Confirmation of successful initialization.

Wired data: `QO`, `STATUS`

- **CNF** (Event): Acknowledgement of a completed service order (e.g., after a value change).

Wired data: `QO`, `STATUS`

Data Inputs

- *QI* (BOOL): Event qualifier – enables processing of the input event.
- *PARAMS* (STRING): Service parameter, typically address or configuration data for the logiBUS node.

- *Output* (logiBUS::io::DQ::logiBUS_DO_S): Identification of the physical output (e.g., Q1...Q8). Initial value: `Invalid`.
- *FREQ* (UINT): Defines the desired LED frequency and priority. Initial value: `LED_FREQ::LED_OFF`.

Data Outputs

- *QO* (BOOL): Output qualifier – indicates the state of the last processing.
- *STATUS* (STRING): Message about the current service status (e.g., error code or "OK").

Adapter

- **OUT** (adapter::types::unidirectional::AX): Unidirectional adapter for passing the Boolean value to the resource.
- *E1* (Event): Signals a new data request.
- *D1* (BOOL): The value to be transmitted (LED on/off).

Functionality

1. **Initialization**

The FB is started by an INIT event. The supplied data (`QI`, `PARAMS`, `Output`, `FREQ`) are forwarded to the inner FB `logiBUS_LED_DO_QX`, which performs the actual logiBUS communication. If successful, `INITO` is sent with the output data `QO` and `STATUS`.

1. **Operation**

After initialization, the function block waits for the event `OUT.E1`. This event is triggered by the connected resource as soon as a new Boolean value (`OUT.D1`) is present. The event is passed on to the internal function block as `REQ`. This block then updates the LED output according to the configured `Output` identification and the specified `FREQ`.

1. **Confirmation**

After successful processing, the internal function block sends a `CNF` event, which is output externally as `CNF`. The outputs `QO` and `STATUS` indicate the current state.

Technical Features

- **Composite Function Block** – This function block simplifies the use of the complex driver function block `logiBUS_LED_DO_QX` by using a reduced interface.
- **Adapter Design** – The adapter `OUT` enables loose coupling between the control logic and the output channel. The Boolean value is passed in response to an event.
- **LED Frequency** – The parameter `FREQ` uses a predefined enumeration (`LED_FREQ`) to define blinking patterns or priorities (e.g., `LED_OFF`, `LED_ON`, `BLINK_1HZ`, ...).
- **Error Handling** – The output `STATUS` returns detailed messages (e.g., from `logiBUS::io::DQ::logiBUS_DO::Invalid`).

State Overview

The function block (FB) does not have explicit state machines in the XML network. The state logic is implemented through the interaction of INIT and the event/data connections:

- **Idle State** – After successful initialization, the FB waits for events from the adapter.
- **Processing** – A `OUT.E1` event triggers an update of the LED output.
- **Error State** – In case of failed initialization or communication errors, `STATUS` is set accordingly.

Application Scenarios

- **Agricultural and Agricultural Technology** – Control of indicator lights on machines, display of operating states via logiBUS.
- **Automation** – Simple on/off signals to distributed I/O modules, e.g., control cabinet LEDs.
- **Wireless Connection** – Use of the logiBUS protocol to bridge larger distances between the controller and the actuator.

Comparison with Similar Function Blocks

Compared to the direct driver function block `logiBUS_LED_DO_QX`, `logiBUS_LED_DO_QXA` offers a leaner interface and integrates initialization in a single step. Other composite function blocks in the `logiBUS_DO`

family focus on digital-binary outputs, while this function block is specifically designed for LED applications (with frequency support).

Conclusion

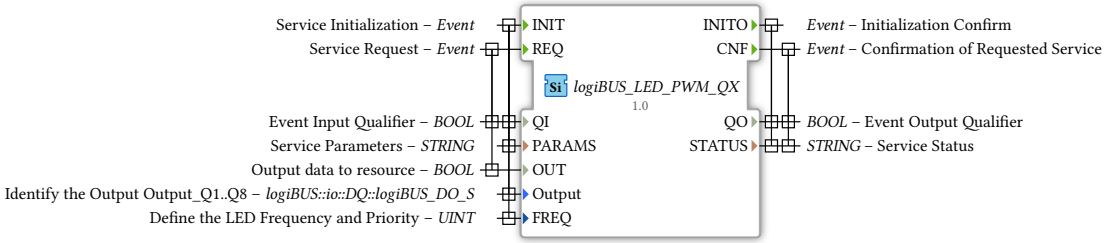
The `logiBUS_LED_DO_QXA` is a convenient composite function block for LED output via logiBUS. It reduces implementation complexity to just a few parameters and an event adapter. Its clear structure makes it particularly suitable for recurring switching tasks in distributed control systems.

logiBUS_LED_PWM_QX

• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_LED_PWM_QX is an output service interface function block for Boolean output data with PWM functionality for LED control. This block enables the control of LED outputs with configurable frequencies and provides a standardized interface for communication with logiBUS I/O resources.



Interface Structure

Event Inputs

- **INIT**: Service initialization with parameters QI, PARAMS, Output, and FREQ
- **REQ**: Service request with parameters QI and OUT

Event Outputs

- **INITO**: Initialization acknowledgment with parameters QO and STATUS
- **CNF**: Acknowledgement of the requested service operation with parameters QO and STATUS

Data Inputs

- **QI**: BOOL - Event input qualifier
- **PARAMS**: STRING - Service parameter
- **OUT**: BOOL - Output data for the resource
- **Output**: logiBUS_DO_S - Identifies the output (Output_Q1..Q8), initial value: Invalid
- **FREQ**: UINT - Defines the LED frequency and priority Initial value: LED_FREQ::LED_OFF

Data Outputs

- **QO**: BOOL - Event output qualifier
- **STATUS**: STRING - Service status

Adapters

No adapter interfaces available.

Functionality

This function block manages the PWM control of LED outputs via the logiBUS system. During initialization (INIT event), the output configuration, frequency settings, and service parameters are passed. Output data (OUT) can be sent via the REQ event, which is then applied to the specified output at the configured PWM frequency.

Technical Features

- Supports various LED frequencies via the FREQ parameter
- Integrated error handling via STATUS output
- Typed output identification via logiBUS_DO_S data type

- Initialization with an invalid state for unconfigured outputs
- Frequency priority control for LED control

State Overview

The function block has the following operating states:

- **Not Initialized**: Before the INIT event, outputs are inactive
- **Initialized**: After successful INIT processing, ready for REQ operations
- **Active**: During PWM signal generation
- **Error**: In case of invalid parameters or configuration errors

Application Scenarios

- Control of status LEDs in industrial control systems
- PWM-controlled lighting systems
- Visual signaling with variable brightness control
- Priority-controlled LED displays in logiBUS-based applications

⚖️ Comparison with similar devices Modules

Compared to simple digital output modules, logiBUS_LED_PWM_QX offers advanced PWM functionality with frequency control and priority-based control. Integration into the logiBUS ecosystem enables standardized communication with I/O resources.

🛠️ Related Exercises

- Exercise_030

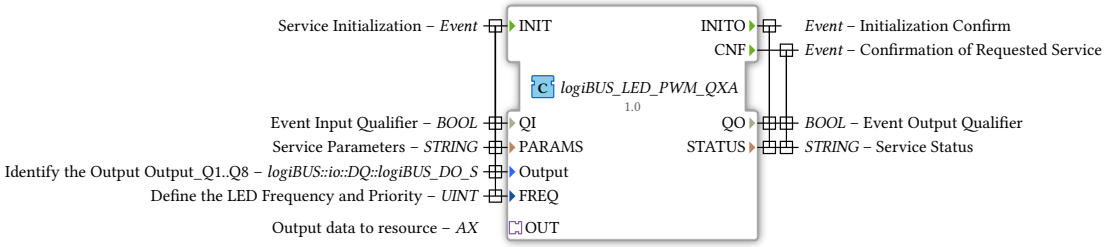
Conclusion

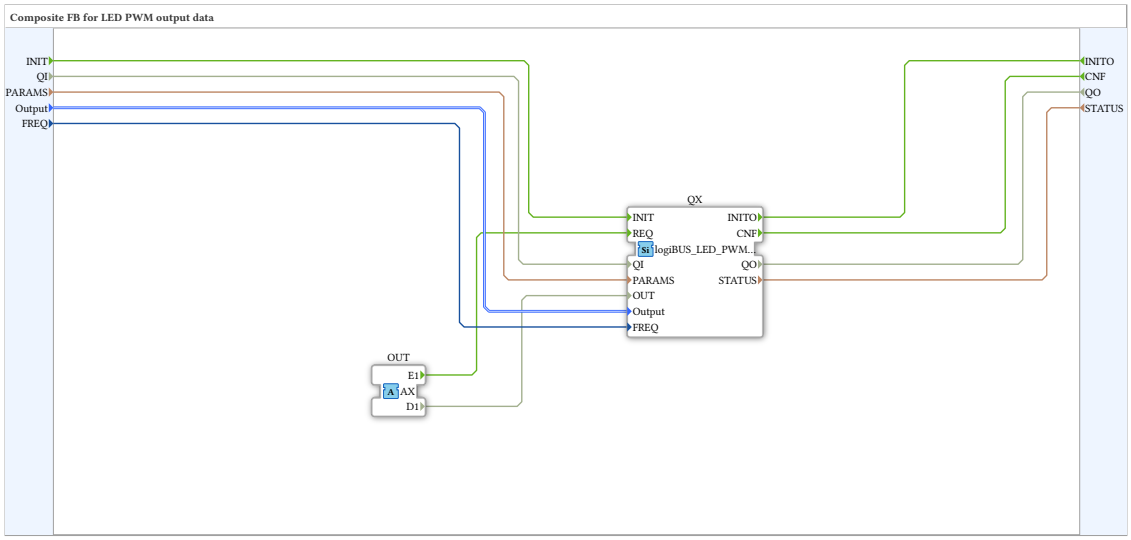
The logiBUS_LED_PWM_QX is a specialized function block for demanding LED control applications in industrial automation systems. With its PWM functionality, configurable frequencies, and integration into the logiBUS system, it provides a powerful solution for visual output scenarios with high demands on flexibility and reliability.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 The PWM signal & infographic on ms-muc-docs.de

logiBUS_LED_PWM_QXA





• – ○ • – ○ • – ○ •

Introduction

The function block `logiBUS_LED_PWM_QXA` is a composite block for controlling a PWM-controlled LED via the logiBUS. It combines the configuration and output of an LED PWM signal and provides a uniform interface for initialization, parameterization, and operation.

Interface Structure

Event Inputs

Event	Type	With Variables	Description
`INIT`	EInit	`QI`, `PARAMS`, `Output`, `FREQ`	Service Initialization; starts the function block with the specified parameters

Event Outputs

Event	Type	With Variables	Description
`INITO`	Initialization	`QO`, `STATUS`	Confirmation of successful initialization
`CNF`	Event	`QO`, `STATUS`	Acknowledgement of a requested service (e.g., after a data request via the adapter)

Data Inputs

Variable	Type	Initial Value	Description
`QI`	BOOL	–	Qualifier for the event input (activation)
`PARAMS`	STRING	–	Service parameter for bus configuration
`Output`	`logiBUS::io::DQ::logiBUS_DO_S`	`Invalid`	Selection of physical output Q1..Q8
`FREQ`	UINT	`LED_OFF`	Frequency and priority of the LED PWM (e.g., from enumeration `LED_FREQ`)

Data Outputs

Variable	Type	Description
QO	BOOL	Qualifier for the event output (activation acknowledgement)
STATUS	STRING	Service status (error/success message)

Adapter

Adapter	Type	Description
OUT	adapter::types::unidirectional::AX	Output adapter for data transmission to the logiBUS resource (via event E1 and data D1)

Functionality

The function block is implemented as a composition and internally contains the sub-FB logiBUS_LED_PWM_QX, which implements the actual PWM control.

1. **Initialization (INIT):**

The FB is started by the event INIT. The parameters QI, PARAMS, Output, and FREQ are forwarded to the sub-function block. After successful initialization, the sub-function block returns INITO and the output data QO and STATUS.

1. **Operation:**

The adapter OUT receives an event from the resource at E1. This event is then passed through to the sub-function block as REQ. The sub-FB processes the request and sends the output data (the current PWM value and the status signal) to the resource via the adapter data channel OUT.D1. The sub-FB's acknowledgment CNF is output as CNF by the composite FB.

1. **Error Handling:**

If errors occur during initialization or operation, the status is reported via the variable STATUS. Output validation is performed via QO.

Technical Features

- **Composite Block**: The FB encapsulates the complexity of the PWM output and provides a simple interface for the user.
- **Typed Output Selection**: The input Output of type logiBUS_DO_S allows explicit assignment to a logiBUS digital output Q1..Q8. The initial value Invalid forces a valid selection before first use.
- **Frequency Setting**: The PWM frequency and priority can be defined via FREQ (type UINT). The initial value LED_OFF switches the LED off by default.
- **Adapter-Based Communication**: The unidirectional adapter AX transmits events and data between the FB and the higher-level logiBUS resource. The FB itself waits for external requests (E1) and returns data.

Adapter-Based Communication:** The unidirectional adapter AX transmits events and data between the FB and the higher-level logiBUS resource. The FB itself waits for external requests (E1) and returns data.
**

State Overview

Since this is a composite function block, it does not have explicit states of its own. However, the internal sub-FB logiBUS_LED_PWM_QX can pass through states such as "Initialization," "Ready," or "Error." These are reflected indirectly via the event outputs and the status variable:

- **After INIT**: The FB is in the initialized state (provided QO = TRUE and STATUS = „OK“ are present).
- **After CNF**: A requested action (e.g., a call via an adapter) has been confirmed. The FB remains operational.
- **Error State**: If STATUS contains an error message, the FB must be reinitialized. Repeating the INIT event resets all parameters.

Application Scenarios

- **PWM-Controlled Lighting**: Use in agricultural machinery or automation systems to adjust the brightness of LED work lights.
- **Light Signaling Systems**: Control of multiple LED outputs with different frequencies (e.g., flashing light, continuous light).
- **Dimming via logiBUS**: Integration into a logiBUS network where the PWM values are calculated centrally or decentrally.

Comparison with Similar Components

- **logiBUS_DO_S** (direct digital output): This component only switches binary outputs (on/off) without PWM. In contrast, `logiBUS_LED_PWM_QXA` offers a PWM function for dimmed LEDs.
- **logiBUS_LED_PWM_QX**: This is the internal sub-component that executes the pure PWM logic. The `QXA` composite function block extends it with an adapter connection and standard interface definition, simplifying integration into higher-end controllers.
- **Generic PWM Function Blocks**: Compared to platform-independent PWM blocks, this function block is specifically tailored to logiBUS hardware and offers the parameters and types commonly found there.

Conclusion

The `logiBUS_LED_PWM_QXA` function block encapsulates the PWM control of an LED via logiBUS in a compact composite block. Its clearly defined interface with initialization, adapter communication, and status feedback makes it easy to integrate into automation projects. The combination of output selection, frequency setting, and error handling makes it a robust component for dimmable lighting applications in industrial environments.

🌐 Related topic subpages on ms-muc-docs.de

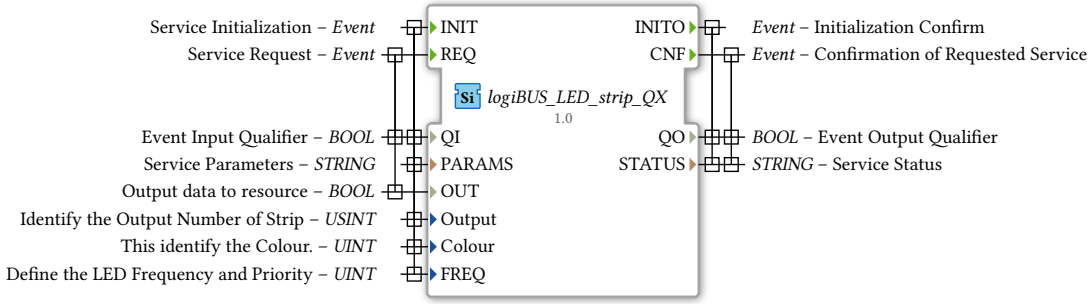
- 🌐 The PWM signal & infographic on ms-muc-docs.de

logiBUS_LED_strip_QX

• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_LED_strip_QX function block is an output service interface function block for Boolean output data, specifically designed for controlling LED strips. It offers extensive configuration options for various LED parameters such as color, frequency, and output number.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters

- **OUT** (BOOL): Output Data to the Resource
- **Output** (USINT): Identifies the strip's output number (initial value: LED_strip::Output_strip)
- **Colour** (UINT): Identifies the color (initial value: LED_COLOURS::LED_GREEN)
- **FREQ** (UINT): Defines the LED frequency and priority (Initial value: LED_FREQ::LED_OFF)

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (STRING): Service status

Adapters

No adapter interfaces available.

Functionality

This function block enables the control of LED strips via a standardized interface. During initialization (INIT), configuration parameters such as output number, color, and frequency are set. Output data (OUT) can be sent to the LED strip via the REQ event. The block acknowledges each operation via the corresponding output events INITO and CNF.

Technical Features

- Support for various LED colors via the Colour parameter
- Configurable frequency settings for blinking functions
- Multiple outputs supported via the Output parameter
- Predefined initial values for quick commissioning
- Status feedback via the STRING parameter

State Overview

The function block has two main states:

1. **Initialization State**: Activated by the INIT event
2. **Operating State**: Processes REQ requests after successful initialization

Application Scenarios

- Industrial lighting control
- Status indicators in automation systems
- Warning and signal light control
- Visualization of process states
- Building automation with LED lighting

⚖️ Comparison with Similar Function Blocks

Compared to simple digital output blocks, logiBUS_LED_strip_QX offers extended functionality for LED-specific applications, particularly through integrated color and frequency control and the ability to address multiple LED strips independently.

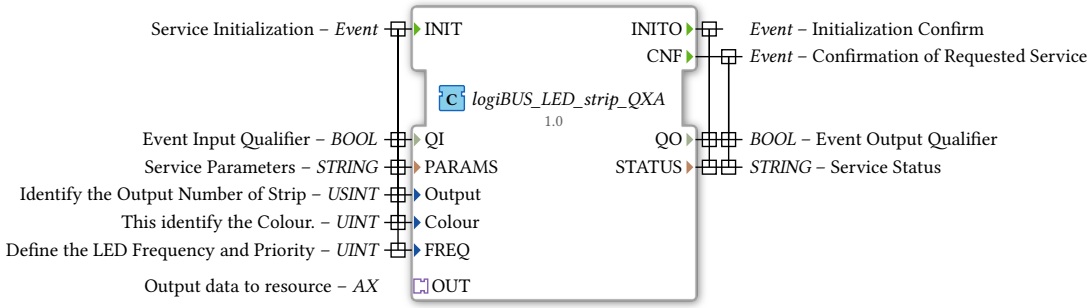
🛠️ Related Exercises

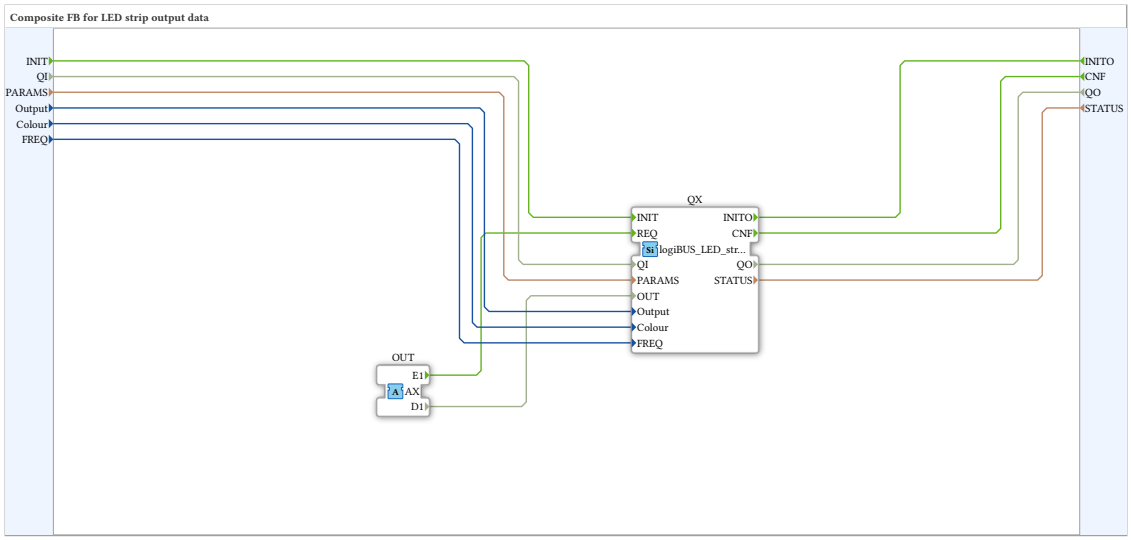
- Exercise_032
- Exercise_033_sub

Conclusion

The logiBUS_LED_strip_QX is a powerful function block for professional LED strip control in industrial automation solutions. Its flexible parameterization and reliable status feedback make it ideal for demanding lighting applications.

logiBUS_LED_strip_QXA





• – ○ • – ○ • – ○ •

Introduction

The function block **logiBUS_LED_strip_QXA** is a composite function block for controlling an LED strip via the logiBUS protocol. It encapsulates the communication with the hardware and enables color- and frequency-dependent control of individual outputs. This block is particularly suitable for use in agricultural technology, where flexible LED signaling is required.

Interface Structure

Event Inputs

Event	Description	With
INIT	Service Initialization	QI, PARAMS, Output, Color, FREQ

Event Outputs

Event	Description	With
INITO	Initialization Confirmation	QO, STATUS
CNF	Confirmation of Executed Request	QO, STATUS

Data Inputs

Name	Type	Description	Initial Value
QI	BOOL	Processing Enable (Event Qualifier)	-
PARAMS	STRING	Service Parameter (e.g., Bus Configuration)	-

Output	USINT	Output Identification (Strip Number)	`LED_strip::Output_strip`
Colour	UINT	LED Color Code	`LED_COLOURS::LED_GREEN`
FREQ	UINT	Frequency/Priority of the LED Display	`LED_FREQ::LED_OFF`

Data Outputs

Name	Type	Description
QO	BOOL	Output Qualifier (Processing Status)
STATUS	STRING	Status Message (e.g., Error Code)

Adapters

Adapter	Type	Description
OUT	`adapter::types::unidirectional::AX`	Unidirectional adapter interface for data transfer to the resource (output data to logiBUS)

Functionality

The function block operates as a composite FB, internally using the sub-block **logiBUS_LED_strip_QX**. After an **INIT** event, the parameters (QI, PARAMS, Output, Colour, FREQ) are passed to the internal FB. Initialization is acknowledged via **INITO**.

As soon as an event (E1) arrives via the **OUT* adapter, a **REQ** request is triggered in the internal FB. The current data (Colour, FREQ) is sent to the logiBUS LED strip. After successful execution, the internal FB acknowledges with **CNF**, which appears at the external output **CNF**.

Data flows:

- **Out.D1** → internal FB.QX.OUT (output data of the adapter)
- **Qi**, **PARAMS**, **Output**, **Colour**, **FREQ** → forwarded to QX accordingly.

The outputs **QO** and **STATUS** reflect the internal state of the sub-block.

Technical Features

- Composite FB: facilitates the reuse and encapsulation of the hardware control.
- Use of an adapter (OUT) for unidirectional data transfer to the logiBUS resource.
- Initial parameter values are predefined as constants (e.g., `LED_strip::Output_strip`, `LED_COLOURS::LED_GREEN`) but can be overwritten at runtime.
- The internal FB `logiBUS_LED_strip_QX` is responsible for the actual bus communication; this block only provides a simplified interface.
- Copyright and developer: HR Agrartechnik GmbH (Version 1.0, 2026-02-23).

State Overview

The function block has no explicitly modeled states; its behavior is determined by the sequence control of the composite network:

1. **Initialization State**: After **INIT**, the internal function block is configured and the bus connection is established.
2. **Ready State**: After successful initialization, the function block can be addressed via the adapter (OUT.E1).
3. **Execution State**: The request is sent to the LED strip until acknowledgment (**CNF**) is received.
4. **Error State**: If QO = FALSE or STATUS contains an error, communication is disrupted.

Application Scenarios

- **Agricultural Machinery**: Color-coded status indicators (e.g., green for ready, red for alarm) on LED strips.
- **Field Edge Lighting**: Control of multiple LED strip outputs with different colors and flashing frequencies.
- **Custom Signaling**: Integration into higher-level control systems for visualizing operating states.

- **Development and Testing**: Use as a simple building block for simulating and commissioning logiBUS components.

Comparison with Similar Building Blocks

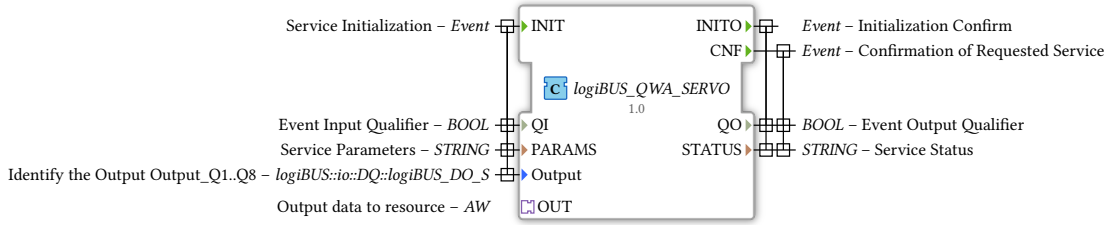
- **logiBUS_LED_strip_QX** (direct FB): Offers a more detailed interface with multiple events (REQ) and data. The composite FB **QXA** described here simplifies its use through an adapter interface and a clear separation of initialization and execution.
- **logiBUS_DO_Bit**: Controls individual digital outputs; no color/frequency control.
- **Composite FBs in General**: The QXA is specifically designed for LED strip applications and reduces wiring effort in the higher-level network.

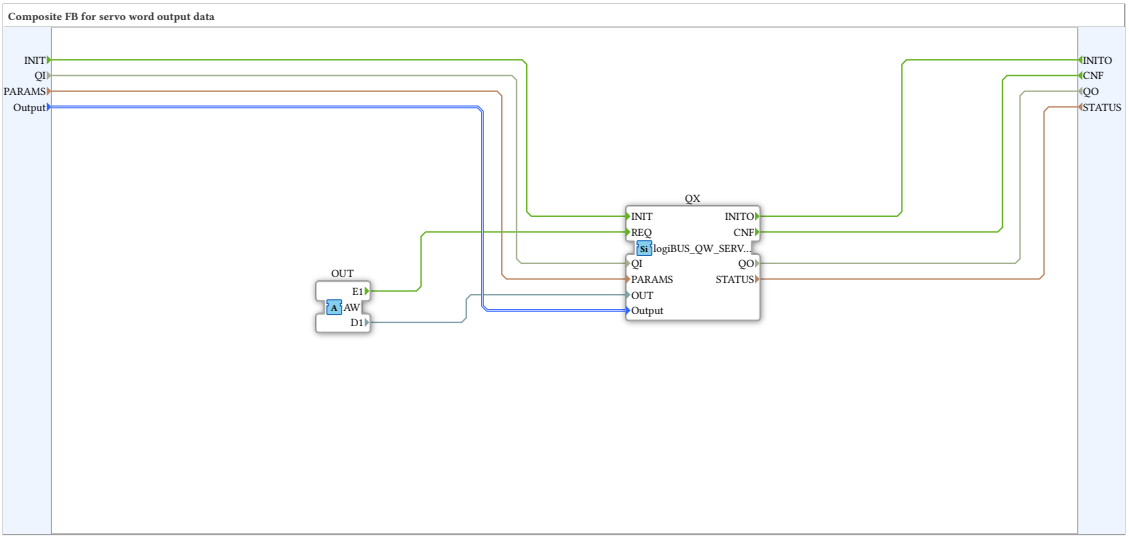
Conclusion

The **logiBUS_LED_strip_QXA** offers a convenient and standardized way to control LED strips via logiBUS in terms of color and flashing frequency. Thanks to its adapter-based interface, it can be easily integrated into larger systems without requiring knowledge of the bus communication details. The composite approach ensures clarity and reusability.

DO servo

logiBUS_QWA_SERVO





• – ○ • – ○ • – ○ •

Introduction

The **logiBUS_QWA_SERVO** is a composite function block (CFB) for outputting servo control words via a logiBUS. It encapsulates the initialization, parameterization, and output based on an internal function block (`logiBUS_QW_SERVO`) and provides a standardized interface. Developed under the Eclipse Public License 2.0 (Copyright 2026 HR Agrartechnik GmbH). *Comment from the XML: "Composite FB for servo word output data".*

Interface Structure

Event Inputs

Event	Type	With Variables	Description
INIT	EInit	QI, PARAMS, Output	Service Initialization

Event Outputs

Event	Type	With Variables	Description
INITO	EInit	QO, STATUS	Initialization Confirmation
CNF	Event	QO, STATUS	Confirmation of a Requested Action

Data Inputs

Name	Type	Initial Value	Description
QI	BOOL	–	Event Input Qualifier
PARAMS	STRING	–	Service Parameter (e.g., Bus Configuration)
Output	logiBUS::io::DQ::logiBUS_DO_S	Invalid	Identifies the output (Output_Q1 ... Output_Q8)

Data Outputs

Name	Type	Description
QO	BOOL	Event Output Qualifier
STATUS	STRING	Status Message (Error, OK, etc.)

Adapters

Adapter	Type	Direction	Description
OUT	adapter::types::unidirectional::AW	Socket	Output data for the resource (word output)

Functionality

The function block operates as a composite:

1. **Initialization** (INIT):

- Qualifier QI, parameter string PARAMS, and the desired output (Output) are passed.
- The internal function block logiBUS_QW_SERVO is initialized; after successful initialization, INITO is sent.

1. **Execution** (via adapter OUT):

- The event OUT.E1 is received by the adapter and forwarded to the internal function block (FB) as REQ.
- Simultaneously, the data is transferred to the internal FB (QX.OUT) via OUT.D1.

1. **Feedback**:

- The internal FB signals completion via CNF, which is forwarded to the external output CNF.
- Status and qualifier are output via QO and STATUS.

The actual control of the servo hardware is handled via the adapter OUT, which is connected to the resource.

Technical Features

- **Composite Design**: Encapsulates the complex initialization and execution control of the servo word output.
- **Adapter Interface**: The unidirectional adapter `AW` enables flexible connection to various physical output channels.
- **Output Selection**: The input `Output` allows selection between eight different outputs (`Q1`…`Q8`); the initial value `Invalid` prevents unintended actions before configuration.
- **Error Handling**: Status information is provided via `STATUS`; event sequences guarantee correct flow control.
- **License**: Eclipse Public License 2.0 (EPL-2.0) – free use and further development under the license terms.

State Overview

The function block (FB) does not have an explicitly modeled state machine; however, the following implicit state logic results from the event sequences:

State	Trigger	Response
Idle	–	Waiting for `INIT`
Initialize	Received `INIT`	Internal FB is started
Ready	`INITO` sent	Ready for requests via the adapter
Busy	`OUT.E1` received	Action is being executed (`REQ` is running)
Done	`CNF` sent	Action completed, returning to **Ready**
Error	Error during initialization or action	`STATUS` contains error description; reinitialization may be necessary

Application Scenarios

- **Agricultural Machinery**: Control of servo drives (e.g., for flaps, valves, actuators) via a logiBUS fieldbus.

Done | `CNF` | Action completed, returning to **Ready** | **Error** | Error during initialization or action | `STATUS` contains error description; reinitialization may be necessary |

- **Agricultural Machinery**: Control of servo drives (e.g., for flaps, valves, actuators) via a logiBUS fieldbus.

** - **Modular Output Block**: Used in distributed control systems where multiple servo outputs need to be parameterized depending on the machine configuration.

- **Parameterizable Configuration**: Bus address, protocol settings, and timing behavior can be defined via `PARAMS`.

Comparison with Similar Blocks

- **logiBUS_QW_SERVO** (internal FB): Implements direct word output and bus communication, while `logiBUS_QWA_SERVO` offers a higher level of abstraction with consolidated inputs/outputs.
- **logiBUS_DO_S Family**: Other blocks in this family address individual discrete outputs (e.g., logiBUS_DO) – the FB described here is specifically designed for servo word data and multiple simultaneous outputs.
- **Generic Word Output Function Blocks**: Compared to standardized `W`-type function blocks, this block offers an interface adapted to servo applications (e.g., `logiBUS_DO_S`-type).

Conclusion

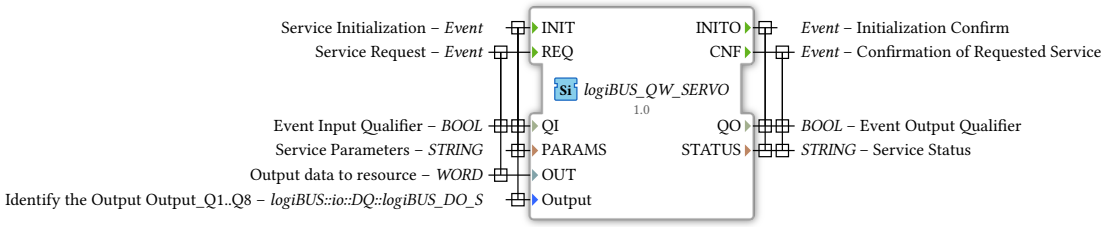
The `logiBUS_QWA_SERVO` is a well-structured composite function block that simplifies the control of servo word outputs on a logiBUS. By separating initialization, execution, and error feedback, and using an adapter for the hardware interface, it is ideally suited for modular automation solutions in agricultural technology and similar industries. Licensing under EPL-2.0 allows for unrestricted use and customization.

logiBUS_QW_SERVO

• – ◦ • – ◦ • – ◦ •

Introduction

The logiBUS_QW_SERVO is an output service interface function block for word output data, specifically designed for controlling logiBUS servo outputs. This block enables the initialization and control of digital outputs via a standardized interface.



Interface Structure

Event Inputs

- **INIT**: Service initialization event with associated data QI, PARAMS, and Output
- **REQ**: Service request event with associated data QI and OUT

Event Outputs

- **INITO**: Initialization acknowledgment with associated data QO and STATUS
- **CNF**: Acknowledgement of the requested service operation with associated data QO and STATUS

Data Inputs

- **QI** (BOOL): Event input qualifier
- **PARAMS** (STRING): Service parameters for configuration
- **OUT** (WORD): Output data for the resource
- **Output** (logiBUS::io::DQ::logiBUS_DO_S): Identifies the output Output_Q1..Q8, initially with a value “Invalid”

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (STRING): Service status information

Adapters

No adapter interfaces available.

Functionality

This function block enables the initialization and control of logiBUS servo outputs. During INIT initialization, the service parameters are configured and the specific output is identified. Output data in WORD format can be sent to the resource via the REQ event. The block confirms both operations with corresponding output events (INITO and CNF) containing status information.

Technical Features

- Uses WORD data type for output data (16-bit)
- Supports specific logiBUS output identification via the output parameter
- Initialization with an “Invalid” value for the output parameter
- String-based parameter and status transmission

State Overview

The function block has two main operating states:

1. **Not Initialized**: Before INIT initialization
2. **Initialized and Ready**: After successful INIT initialization, ready for REQ requests

Application Scenarios

- Control of digital outputs in logiBUS systems
- Servo controllers with word data output
- Industrial automation applications with precise output control
- Embedded systems with a logiBUS interface

⚖️ Comparison with Similar Function Blocks

Compared to simple digital output blocks, logiBUS_QW_SERVO offers extended functionality through:

- Word-based data output (16-bit instead of individual bits)
- Specific servo control functions
- Detailed status feedback
- Configurable initialization

Conclusion

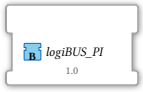
The logiBUS_QW_SERVO is a specialized function block for demanding output applications in logiBUS systems. Its word-based data processing and comprehensive status feedback make it particularly suitable for precise control tasks in industrial automation solutions.

PI

🎧 Podcast

- Decoding IEC 61499: The Industrial Revolution Reshaping Factories and Careers
- ISOBUS: How logos get on your tractor terminal - The Picture Graphic object explains
- From “Mass Errors” to Masterpiece: Streamlining Industrial Software by Eliminating Mapping
- Logibus Revolution: Unpacking the “No Mapping” Future and Streamlined Development
- Logibus Unleashed: How Eliminating “Mapping” Simplifies Complex Systems and Boosts Usability

logiBUS_PI



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `logiBUS_PI` is a collection of global constants for configuring pulse inputs in logiBUS systems. It defines preconfigured structures that represent the physical input pins of a logiBUS I/O module. These constants facilitate the simple and error-free assignment of hardware pins in higher-level function blocks or applications.

Interface Structure

logiBUS_PI` is a GlobalConstants block, not a traditional function block with event or data interfaces. Therefore, it has no event inputs, outputs, data inputs, outputs, or adapters.

The interface consists solely of the defined global constants. Each constant is of data type `logiBUS::io::PI::logiBUS_PI_S` and is instantiated with a specific initial value.

Defined Constants

The following constants are provided:

- `PulseInput_I1` to `PulseInput_I9`: Represent the physical pulse inputs 1 to 9. The initial value sets the structure field `Pin` to the corresponding value (1-9).
- `Invalid`: Represents an invalid or unused input. The initial value sets the structure field `Pin` to the value 255.

Functionality

The block `logiBUS_PI` itself does not perform any active functionality. It merely provides a library of predefined values. These values are of type `logiBUS_PI_S`, a structure that presumably contains at least the member `Pin`.

The constants are inserted into the application code at compile time. If another function block (e.g., a driver or measurement module) expects a parameter of type `logiBUS_PI_S`, one of these constants can be passed to uniquely select a specific hardware input.

Technical Features

- **Typing:** All constants are strictly of the structured data type `logiBUS_PI_S`. This enforces type-safe usage and prevents the unintentional assignment of simple integer values.
- **Initialization:** The initial values are specified using the ST syntax `(Pin := X)`, indicating structured initialization.
- **Invalid State:** The constant `Invalid` with pin value 255 provides a standardized way to indicate an unused or faulty channel.

State Overview

Since it is a constant block, `logiBUS_PI` has no internal state or state machine. The constant values are immutable (read-only) and remain constant throughout the application's runtime.

Application Scenarios

- **Hardware Configuration:** Primarily used to configure logiBUS I/O modules that provide pulse inputs (e.g., for encoders, frequency, or speed measurement).
- **Function Block Parameterization:** Simplified and readable parameterization of function blocks that require a specific pulse input. Instead of the magic number `3`, the self-explanatory constant `PulseInput_I3` is used.
- **Maintenance and Readability:** Increases the maintainability and readability of the application code, as the hardware assignment is directly apparent in the constant name.

⚖️ Comparison with Similar Blocks

Unlike function blocks such as `E_SR` (flip-flop) or `E_CYCLE` (cyclic event generator), `logiBUS_PI` does not implement any logic or algorithms. It is comparable to other constant blocks or ENUM definitions in IEC 61499 that encapsulate hardware-related parameters (e.g., `GPIO_PIN_XX` constants for Raspberry Pi). Its purpose is purely declarative and configurative.

Conclusion

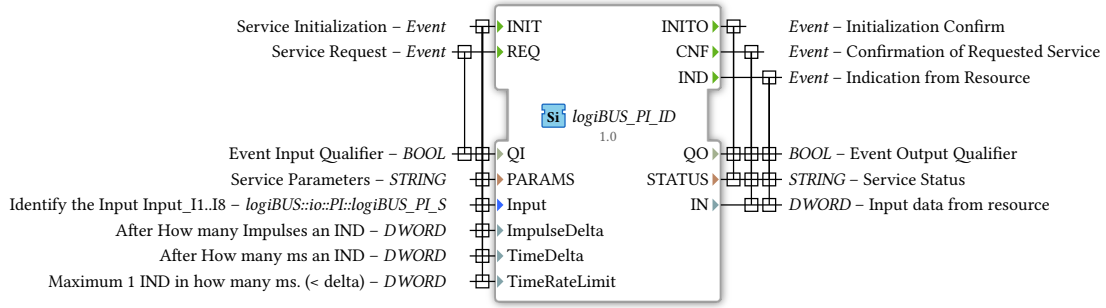
The `logiBUS_PI` GlobalConstants block is an essential tool for low-level programming of logiBUS systems. By providing type-safe, predefined constants for pulse inputs, it enables clean, fault-resistant, and easily maintainable configuration of the input/output hardware. Its use promotes best practices in industrial automation programming.

logiBUS_PI_ID

• – ○ • – ○ • – ○ •

Introduction

The function block `logiBUS_PI_ID` is an input service interface module for 32-bit DWORD input data. It serves as an interface to a physical input module (presumably part of the logiBUS system) and enables the initialization, cyclic polling, and event-driven (interrupt) output of digital input values. The module can be configured to report status changes either after a specific number of pulses or after a defined time interval.



Interface Structure

Event Inputs

- **`INIT`**: Initializes the service. Accompanied by the data `QI`, `PARAMS`, `Input`, `ImpulseDelta`, `TimeDelta`, and `TimeRateLimit`.

- **REQ**: Triggers a cyclic poll of the input value. Accompanied by the data QI.

Event Outputs

- **INITO**: Confirms initialization. Returns the data QO and STATUS.
- **CNF**: Acknowledges a requested query (REQ). Returns the data QO, STATUS, and the current input value IN.
- **IND**: Indicates an event-driven status change (interrupt). Returns the data QO, STATUS, and the new input value IN.

Data Inputs

- **QI (BOOL)**: Qualifies the associated event input. The service is activated/executed at TRUE and deactivated at FALSE.
- **PARAMS (STRING)**: Contains service-specific parameters for initialization (e.g., hardware address, channel configuration).
- **Input (logiBUS_PI_S)**: Identifies the specific physical input (e.g., I1..I8). The initial value is logiBUS_PI::Invalid.
- **ImpulseDelta (DWORD)**: Defines after how many consecutive state changes (pulses) a IND event should be generated.
- **TimeDelta (DWORD)**: Defines the time interval in milliseconds after which a IND event should be generated if the value has changed.
- **TimeRateLimit (DWORD)**: At most one IND event per the given time in milliseconds (must be smaller than TimeDelta). The initial value is 100.

Data Outputs

- **QO (BOOL)**: Displays the service execution status. TRUE indicates success, FALSE signals an error.
- **STATUS (STRING)**: Provides a detailed status or error message from the service.
- **IN (DWORD)**: Contains the current 32-bit value read from the physical input.

Adapters

This function block has no adapter interfaces.

Functionality

The function block operates in two basic modes, controlled by the events REQ and IND:

1. **Polling Mode**: A REQ event queries the current input value, and the result is returned with a CNF event.
2. **Interrupt Mode**: After successful initialization (INIT/INITO), the function block continuously monitors the input. Upon a state change, the parameters ImpulseDelta and TimeDelta are evaluated. If one of the criteria is met (e.g., the defined number of pulses is reached or the time interval is exceeded *and* the value has changed), a IND event with the new value is automatically triggered.

Initialization (INIT) is a prerequisite for both operating modes. During initialization, the hardware resource is configured via PARAMS and the specific input via Input.

Technical Features

- **Data Type**: Processes 32-bit input data (DWORD).
- **Structured Input**: The input is not identified by a simple index, but by a structured data type (logiBUS_PI_S), which enables type-safe and unambiguous addressing.
- **Flexible Event Triggering**: The conditions for automatic event generation (IND) can be configured on both an impulse and time basis.
- **Service Interface**: Follows the typical pattern of a 4diac service interface function block (FB) with variables QI/QO and STATUS for consistent error handling.

Status Overview

1. **Inactive**: After startup or upon QI=FALSE.
2. **Initialization**: Upon receiving INIT with QI=TRUE. Configures the hardware interface. Ends with INITO (QO indicates success/failure).
3. **Ready (Polling)**: After successful initialization. The system responds to REQ events with CNF and the current value.
4. **Ready (Monitoring)**: After successful initialization. Continuously monitors the input and triggers IND events according to the configured ImpulseDelta and TimeDelta parameters.

Application Scenarios

- **Reading Counter Signals**: Acquiring pulses from a rotary encoder or rotary switch, using `ImpulseDelta` for preprocessing (e.g., reporting every 10th revolution).
- **Monitoring Status Groups**: Reading a 32-bit status word from a connected device, where changes only need to be reported at specific intervals (`TimeDelta`) to reduce CPU load.
- **Cyclic Polling of Switch Banks**: Polling multiple digital inputs grouped into a DWORD via regular `REQ` events.
-

⚖️ Comparison with similar function blocks

- **Compared to `E_DEMUX` or `E_SELECT`**: These function blocks forward events or select data. `logiBUS_PI_ID` is specific to hardware communication and includes driver logic and initialization.
- **Compared to generic I/O function blocks (e.g., `WAGO_750_5xx_DI`)**: Similar function, but manufacturer-specific (here, logiBUS). Configuration is done via the structured parameters `Input` and `PARAMS` instead of fixed channel numbers.
- – **Compared to simpler input blocks**: Offers advanced features such as filtering event generation (`IND`) via `ImpulseDelta`/`TimeDelta`, which are typically not available in simple "Read" blocks.

🛠️ Related exercises

- Uebung_150
- Uebung_150_AX
- Uebung_151
- Uebung_151_AX
- Uebung_152
- Uebung_153

Conclusion

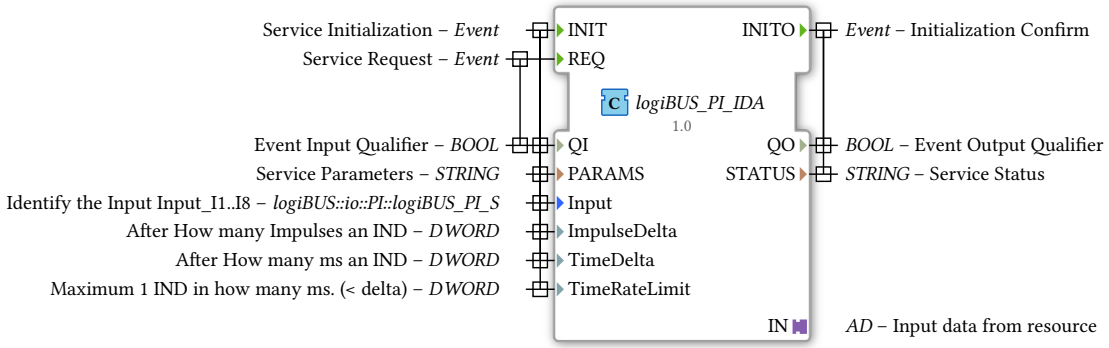
The `logiBUS_PI_ID` function block is a powerful and flexible interface for 32-bit digital inputs within the logiBUS ecosystem. By combining polling (`REQ`/`CNF`) and event-driven querying (`IND`)With configurable trigger criteria, it is suitable for a wide range of applications, from simple status queries to complex pulse evaluation. The strict separation of initialization and operating logic, along with comprehensive status reporting, makes it a robust component for industrial control applications.

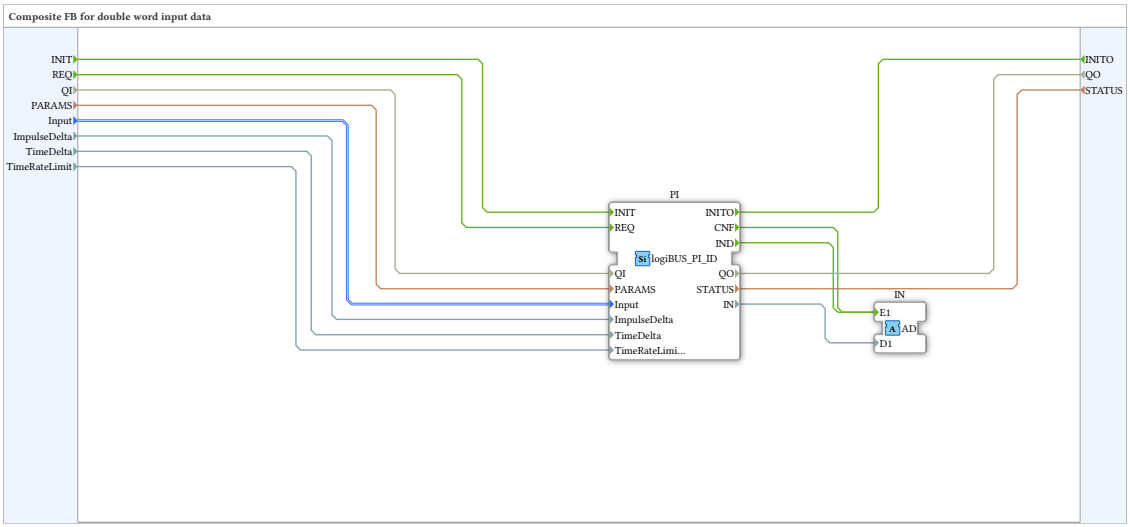
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

logiBUS_PI_IDA





• – ○ • – ○ • – ○ •

Introduction

The **logiBUS_PI_IDA** is a composite function block (FB) for processing double-word input data (DWORD). It serves as an interface between a logiBUS fieldbus and the control logic. The FB enables parameterizable monitoring of pulse and time changes at a digital input. By configuring pulse delta, time delta, and a rate limit, the signaling behavior can be precisely tailored to the application. The block initializes the internal sub-block `logiBUS_PI_ID` and provides its results via a unidirectional adapter.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Service Initialization – Parameterization and Start
REQ	Event	Service Request – Triggers processing

Event Outputs

Name	Type	Comment
INIT	EInit	Initialization Confirm – Confirms successful initialization

Data Inputs

Name	Type	Initial Value	Comment
QI	BOOL	–	Event Input Qualifier (Enable/Disable)
PARAMS	STRING	–	Service Parameters (e.g., bus address, channel configuration)
Input	logiBUS::io::PI::logiBUS_PI_S	`Invalid`	Identifies the physical input (Input_I1..I8)
PulseDelta	DWORD	–	Number of pulses after which an indication (IND) is triggered
TimeDelta	DWORD	`250`	Time in ms after which a periodic indication (IND) is triggered

TimeRateLimit	DWORD	100	Minimum interval in ms between two indications (rate limiting)

Data Outputs

Name	Type	Comment
QO	BOOL	Event Output Qualifier – Status of the last processing
STATUS	STRING	Service Status – Error or Diagnostic Message

Adapter

Name	Type	Comment
IN	adapter::types::unidirectional::AD	Output data of the function block – delivers the processed events and values to the application

The adapter `IN` is connected to the events `IND` (indication) and `CNF` (confirmation), as well as the data value `D1` of the internal function block `logiBUS_PI_ID`.

Functionality

The function block (FB) operates as a composite, delegating all initialization and processing logic to the internal FB `logiBUS_PI_ID`.

1. **Initialization (INIT):**

Upon arrival of `INIT`, the parameters `QI`, `PARAMS`, `Input`, `ImpulseDelta`, `TimeDelta`, and `TimeRateLimit` are forwarded to the internal FB. This FB configures the hardware input and starts monitoring. After successful initialization, `INITO` is output.

1. **Processing (REQ):**

An event `REQ` triggers a query of the input value. The internal function block checks whether the configured thresholds (pulse counter, time limit) have been reached or exceeded. If so, the events `IND` and/or `CNF` are sent to the downstream logic via the adapter `IN`.

1. **Reporting Behavior:**

- **Pulse-Dependent:** If `ImpulseDelta > 0` occurs, an internal pulse counter is incremented after each change in the input signal. When the counter reaches the value of `ImpulseDelta`, an indication (`IND`) is triggered and the counter is reset.
- **Time-dependent:** Additionally or alternatively, a periodic indication is generated after `TimeDelta` milliseconds.
- **Lock logic:** If `ImpulseDelta = 0` is set, `TimeDelta` must not be 0 (error prevention). If `TimeDelta = 0xFFFFFFFF` is set, cyclic processing is deactivated – only pure value changes are reported.
- **Rate limit:** `TimeRateLimit` prevents excessively frequent indications; it ensures that at least the specified time in milliseconds elapses between two consecutive `IND` events.

Technical Features

- The function block (FB) internally uses the module `logiBUS_PI_ID`, which encapsulates the actual hardware connection and counting logic.
- If `ImpulseDelta = 0` is present, `TimeDelta > 0` must be set; otherwise, the behavior is undefined (see license information).
- A `TimeDelta` from `0xFFFFFFFF` disables the time-based indication – only pulse-driven or pure value changes are transmitted.
- The adapter `IN` provides the output data as a unidirectional data stream, which is further processed in the higher-level application.

If `ImpulseDelta = 0` is present, `TimeDelta > 0` must be set; otherwise, the behavior is undefined (see license information).

- A `TimeDelta` from `0xFFFFFFFF` disables the time-based indication – only pulse-driven or pure value changes are transmitted.
- The adapter `IN` provides the output data as a unidirectional data stream, which is further processed in the higher-level application. - The initial values (`Input`: `Invalid`, `TimeDelta`: `250`, `TimeRateLimit`: `100`) are sensibly preset to enable fast operation without additional configuration.

State Overview

Since the FB is a composite and the state logic is completely encapsulated in the internal FB `logiBUS_PI_ID`, a general state description is given here:

- **IDLE** – After a reset or failed initialization. No processing.
- **INIT** – Initialization is in progress (waiting for hardware feedback).
- **RUN** – Normal operation: Monitoring of pulses and time, triggering of indications.
- **ERROR** – Error state (e.g., invalid parameters, hardware error). Signaled via `STATUS`.

Transitions are triggered by `INIT`, `REQ`, and internal errors. The internal function block (FB) likely uses an Execution Control Chart (ECC) to implement these states.

Application Scenarios

- **Monitoring of Digital Inputs:** Connection of pushbuttons, switches, or limit switches, where both rapid pulse changes and regular status messages are required.
- **Pulse Counting:** E.g., detection of flow sensors or rotary encoders; the FB reports a measured value after each defined number of pulses.
- **Time-Controlled Queries:** In applications where the input value must be queried regularly (every ms), e.g., B. for long-term monitoring or safety testing.
- **Rate Limiting:** Prevents overloading of the bus communication during high-frequency signal changes.

Comparison with Similar Function Blocks

Function Block	Difference
`logiBUS_PI`	Simpler input function block without pulse and time-delta filters; only raw value changes.
`logiBUS_PI_ID`	Predecessor without composite structure and without adapter interface; direct event outputs.
`logiBUS_PI_IDA` (this function block)	Additionally offers an **adapter** for flexible further processing and combines pulse, time, and rate logic in a composite.

The `logiBUS_PI_IDA`This represents an extended and more modular variant, particularly suitable for complex automation projects with standardized interfaces.

Conclusion

The **logiBUS_PI_IDA** is a powerful composite function block for the configurable acquisition of digital input signals via the logiBUS. By combining pulse and time delta measurements with rate limiting, it can be flexibly adapted to a wide variety of requirements. The encapsulation of the logic within the internal block and the adapter interface facilitate reuse and integration into higher-level control programs. The function block is ideal for applications requiring precise and configurable signal evaluation with minimal resource consumption.

logiBUS_PI_S

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `logiBUS_PI_S` is a structured data type used to configure a digital input (PI - likely "Peripheral Input") within the logiBUS framework. It defines the necessary parameters to address a specific physical pin of an I/O module or controller. This data type is typically used as an input parameter for corresponding I/O function blocks.

Interface Structure

As a data type (STRUCT), `logiBUS_PI_S` does not have event-driven interfaces, but defines only data fields.

Data Inputs

Not applicable – this is a data type, not an executable function block.

Internal Structure (Variables)

The structure consists of a single member:

- **`Pin`** (USINT, initial value: 255): This variable specifies the hardware pin number to be used as a digital input. The data type `USINT` (Unsigned Short Integer) allows values from 0 to 255. The default initial value of 255 is often used to represent an inactive or undefined state.

Adapter

Not present.

Functionality

The data type `logiBUS_PI_S` functions purely as a configuration container. It encapsulates the information about which physical pin should be addressed. The actual logic for querying the pin state lies within the function blocks that expect a variable of this type as an input (e.g., `logiBUS_PI` or similar). These blocks read the value from the `Pin` field and use it to control the corresponding hardware interface.

Technical Features

- **Initial Value:** The initial value of 255 for `Pin` is common practice to indicate an unconfigured or faulty state, as pin numbers in practice are usually significantly lower.
- **Package Assignment:** This data type is part of the `logiBUS::io::PI` package, which assigns it to the peripheral input category within the logiBUS library.
- **Type Hash:** The attribute `eclipse4diac::core::TypeHash` with an empty value indicates that an automatically generated hash is used for type compatibility within the system.

State Overview

Since this is a passive data type, there is no state machine or active states.

Application Scenarios

This structured data type is used to centralize and ensure type safety in the configuration of digital input channels in control applications with logiBUS-compatible hardware. Typical applications include:

- Configuration of push-button, switch, or sensor signals (e.g., limit switches, light barriers).
- Parameterization of function blocks that read digital inputs in a distributed I/O system (such as logiBUS).
- Simplification of code reuse and readability, as the pin configuration is bundled in a named structure.

⚖️ Comparison with Similar Components

- **Simple Data Types (e.g., USINT directly):** Using a structured `logiBUS_PI_S` variable is preferable to directly using a `USINT` variable for the pin number because it is semantically clearer and can be more easily extended when the configuration is expanded (e.g., to include filter time or inversion) without having to change all call points.
- **logiBUS_PO_S (Digital Output):** A complementary data type `logiBUS_PO_S` for configuring digital outputs (PO - Peripheral Output) with an identical or very similar structure likely exists.
- **logiBUS_PO_S (Digital Output):**

Conclusion

The `logiBUS_PI_S` data type is a fundamental, specific configuration block for connecting digital input signals in logiBUS-based IEC 61499 applications. Its simple, single-parameter structure enables clear and centralized hardware configuration and forms the basis for the corresponding functionality in the executing I/O function blocks.

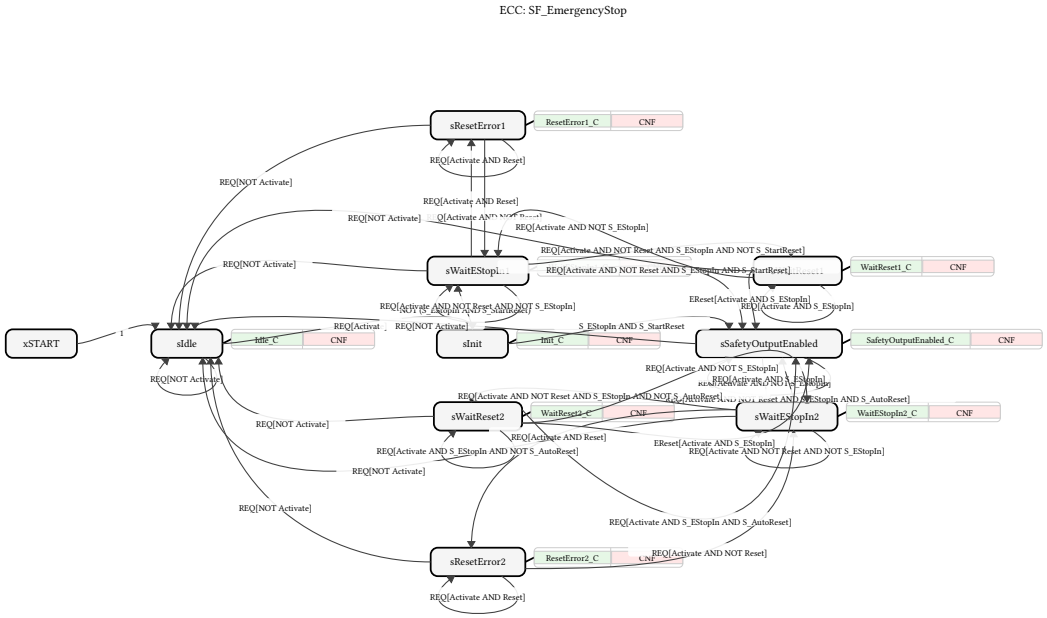
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.10.3.8 Safety

PLCopen

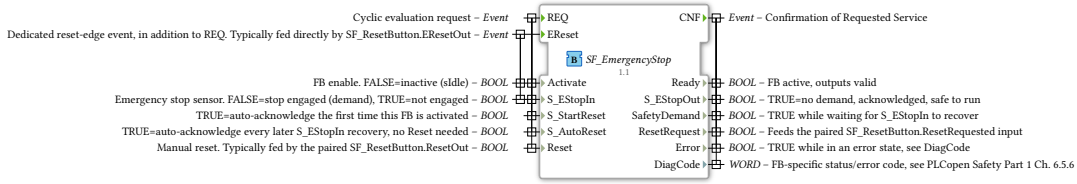
SF_EmergencyStop



• — ○ • — ○ • — ○ •

Introduction

`SF_EmergencyStop` is the PLCopen Safety function `SF_EmergencyStop` (Part 1 v2.10, Ch. 6.5) for **monitoring an emergency stop sensor** (`S_EStopIn`). The output `S_EStopOut` only becomes `TRUE` again after `S_EStopIn` is `TRUE` again **and** an acknowledgment has occurred (`S_StartReset`/`S_AutoReset`/`Reset`) - per EN ISO 13849-1:2015 5.2.2 and IEC 60204-1:2016 9.2.3.4.



Interface Structure

Event Inputs

- **REQ**: Cyclic evaluation request. Associated data: `Activate`, `S_EStopIn`, `S_StartReset`, `S_AutoReset`, `Reset`
- **EReset**: Dedicated reset-edge event in addition to `REQ`, typically fed directly by `SF_ResetButton.EResetOut`. Associated data: `Activate`, `S_EStopIn`

Event Outputs

- **CNF**: Confirmation of the requested service. Associated data: `Ready`, `S_EStopOut`, `SafetyDemand`, `ResetRequest`, `Error`, `DiagCode`

Data Inputs

- **Activate** (`BOOL`, default `FALSE`): Block enable; `FALSE` = inactive (`sIdle`)
- **S_EStopIn** (`BOOL`, default `FALSE`): Emergency stop sensor; `FALSE` = stop engaged (demand), `TRUE` = not engaged
- **S_StartReset** (`BOOL`, default `FALSE`): `TRUE` = auto-acknowledge the first time this block is activated

- **S_AutoReset** (BOOL, default FALSE): TRUE = auto-acknowledge every later recovery of S_EStopIn, no Reset needed
- **Reset** (BOOL, default FALSE): Manual reset, typically fed by the paired SF_ResetButton.ResetOut

Data Outputs

- **Ready** (BOOL): Block active, outputs valid
- **S_EStopOut** (BOOL): TRUE = no demand, acknowledged, safe to run
- **SafetyDemand** (BOOL): TRUE while waiting for S_EStopIn to recover
- **ResetRequest** (BOOL): Feeds the paired SF_ResetButton.ResetRequested input
- **Error** (BOOL): TRUE in an error state, see DiagCode
- **DiagCode** (WORD, default 16#0000): Block-specific status/error code (PLCopen Safety Part 1, Ch. 6.5.6)

Adapters

This function block uses no adapters.

How it works

The block is a basic FB with nine states. The outputs are set in each state by its own algorithm; afterwards CNF is fired.

State	DiagCode	S_EStopOut	SafetyDemand	ResetRequest	Error	Meaning
sIdle	0000	FALSE	FALSE	FALSE	FALSE	Block not active
sInit	8001	FALSE	FALSE	FALSE	FALSE	Activated, S_StartReset is checked
sWaitEStopIn1	8802	FALSE	TRUE	FALSE	FALSE	First activation, waiting for S_EStopIn
sResetError1	C001	FALSE	FALSE	FALSE	TRUE	Reset stuck TRUE while waiting for S_EStopIn
sWaitReset1	8402	FALSE	FALSE	TRUE	FALSE	S_EStopIn is TRUE, waiting for a rising edge on Reset
sSafetyOutputEnabled	8000	TRUE	FALSE	FALSE	FALSE	Safety output enabled
sWaitEStopIn2	8804	FALSE	TRUE	NOT Reset	FALSE	Demand detected, waiting for S_EStopIn
sResetError2	C011	FALSE	FALSE	FALSE	TRUE	Reset stuck TRUE while waiting for S_EStopIn
sWaitReset2	8404	FALSE	FALSE	NOT Reset	FALSE	Waiting for S_AutoReset or a rising edge on Reset

Sequence:

1. `REQ` with `Activate = TRUE` in `sIdle` leads to `sInit`. If `S_EStopIn AND S_StartReset`, it goes directly to `sSafetyOutputEnabled`, otherwise to `sWaitEStopIn1`. `S_StartReset` only counts on this very first activation.
2. First activation: in `sWaitEStopIn1`, `Reset = TRUE` leads to the error `sResetError1`; once `S_EStopIn` is `TRUE` again, `sSafetyOutputEnabled` or `sWaitReset1` follows depending on `S_StartReset`. In `sWaitReset1`, the event `EReset` (with `S_EStopIn`) leads to `sSafetyOutputEnabled`; if `S_EStopIn` drops, it goes back to `sWaitEStopIn1`.
3. Operation: in `sSafetyOutputEnabled`, `S_EStopIn = FALSE` leads to `sWaitEStopIn2`.
4. Subsequent cycles: in `sWaitEStopIn2`, `Reset = TRUE` leads to the error `sResetError2`; once `S_EStopIn` is `TRUE` again, `sSafetyOutputEnabled` or `sWaitReset2` follows depending on `S_AutoReset`. In `sWaitReset2`, `S_AutoReset` or the event `EReset` leads to `sSafetyOutputEnabled`; if `S_EStopIn` drops, it goes back to `sWaitEStopIn2`.
5. From any state, `REQ` with `Activate = FALSE` leads back to `sIdle`.

Technical Details

- The error states `sResetError1`/`sResetError2` prevent a permanently present `Reset` from replacing the acknowledgment (`Reset` must be an edge).
- `ResetRequest` is combined with `NOT Reset` in the subsequent cycles (`sWaitEStopIn2`, `sWaitReset2`).
- The diagnostic codes come from `logiBUS::safety::PLCopen::const::DiagCodes` and `EmergencyStopDiagCodes`.

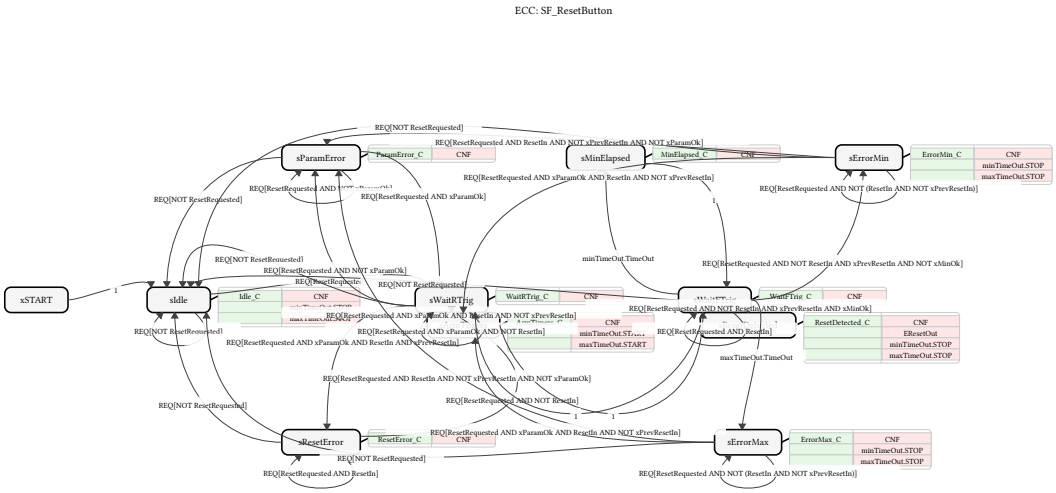
Application Scenarios

- Monitoring an emergency stop button with manual acknowledgment via `SF_ResetButton`
- Automatic restart on demand via `S_StartReset`/`S_AutoReset`

Summary

`SF_EmergencyStop` implements the PLCopen Safety state machine for emergency stop monitoring and enables the safety output only after the sensor has recovered and has been acknowledged.

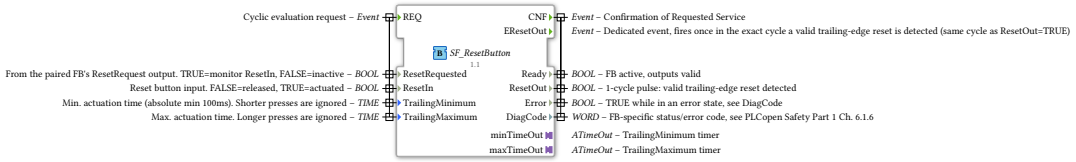
SF_ResetButton



• – ○ • – ○ • – ○ •

Introduction

`SF_ResetButton` is the PLCopen safety block `SF_ResetButton` (PLCopen TC5 Safety Part 1 v2.10, chapter 6.1) with trailing-edge reset. It monitors a reset button: a valid reset exists when the button is pressed (rising edge) and released again (falling edge) after a time between `TrailingMinimum` and `TrailingMaximum`. `ResetOut` then outputs a single pulse.



Interface Structure

Event Inputs

- **REQ**: cyclic evaluation request
  - **Associated data**: `ResetRequested`, `ResetIn`, `TrailingMinimum`, `TrailingMaximum`

Event Outputs

- **CNF**: confirmation of the requested service
  - **Associated data**: `Ready`, `ResetOut`, `Error`, `DiagCode`
- **EResetOut**: dedicated event that fires once in exactly the cycle in which a valid trailing-edge reset is detected (same cycle as `ResetOut = TRUE`)

Data Inputs

- **ResetRequested** (BOOL): from the paired FB (`ResetRequest` output). TRUE = monitor `ResetIn`, FALSE = inactive (default TRUE)
- **ResetIn** (BOOL): reset button input. FALSE = released, TRUE = actuated (default FALSE)
- **TrailingMinimum** (TIME): minimum actuation time (absolute minimum 100 ms). Shorter actuations are not accepted as reset (default `T#350ms`)
- **TrailingMaximum** (TIME): maximum actuation time. Longer actuations are not accepted as reset (default `T#2s`)

Data Outputs

- **Ready** (BOOL): FB active, outputs valid
- **ResetOut** (BOOL): one-cycle pulse when a valid trailing-edge reset has been detected
- **Error** (BOOL): TRUE while an error state exists (see `DiagCode`)
- **DiagCode** (WORD): FB-specific status/error code (PLCopen Safety Part 1, chap. 6.1.6)

Adapters

- **minTimeOut** (plug, `iec61499::events::ATimeOut`): timer for `TrailingMinimum`
- **maxTimeOut** (plug, `iec61499::events::ATimeOut`): timer for `TrailingMaximum`

Functionality

The block is a basic FB with an ECC. Internal variables: `xPrevResetIn` (value of `ResetIn` in the previous cycle for edge detection), `xMinOk` (TRUE as soon as `TrailingMinimum` has elapsed since the rising edge) and `xParamOk` (`TrailingMinimum <= TrailingMaximum` and `TrailingMinimum >= T#100ms`).

- **Activation**: with `ResetRequested = FALSE` the block stays in state `sIdle`. With `ResetRequested = TRUE` it moves to `sWaitRTrig`.
- **`sWaitRTrig`**: if the parameters are invalid, `sParamError` follows. If `ResetIn` was already pressed before activation (`ResetIn AND xPrevResetIn`), `sResetError` follows. A fresh rising edge (`ResetIn AND NOT xPrevResetIn`) leads to `sArmTimers`.
- **`sArmTimers`**: sets `minTimeOut.DT := TrailingMinimum` and `maxTimeOut.DT := TrailingMaximum`, clears `xMinOk`, starts both timers (`minTimeOut.START`, `maxTimeOut.START`) and moves unconditionally to `sWaitFTrig`.
- **`sWaitFTrig`**: if `minTimeOut` expires, `xMinOk` is set via `sMinElapsed`. If `maxTimeOut` expires, `sErrorMax` follows. If the button is released (`NOT ResetIn AND xPrevResetIn`), the state `sResetDetected` follows when `xMinOk` is set, otherwise `sErrorMin`.
- **`sResetDetected`**: `ResetOut := TRUE`, `EResetOut` is output, both timers are stopped; then unconditionally back to `sWaitRTrig`.
- **Error states** (`sParamError`, `sResetError`, `sErrorMin`, `sErrorMax`): `Error := TRUE` with the respective `DiagCode`. From `sErrorMin`/`sErrorMax` a new rising edge leads back to `sArmTimers`, from `sResetError` releasing the button leads to `sWaitRTrig`. In every state, `ResetRequested = FALSE` leads back to `sIdle`.

Technical Details

- `EResetOut` is a non-literal IEC 61499 extension of the PLCopen interface: a dedicated event lets consumers wire the reset pulse directly instead of deriving it from `CNF` and `ResetOut` through an external gate (version 1.1)
- Structure following the pattern of `sequence_T_05` (ECC + `ATimeOut` plugs)
- `ResetOut` and `EResetOut` are active in exactly one cycle only
- The diagnostic codes come from the constants `DiagCodes` and `ResetButtonDiagCodes` (`logiBUS::safety::PLCopen::const`)

State Overview

State	DiagCode	Meaning
xSTART	-	initial state, moves unconditionally to `sIdle`
sIdle	0000	FB not active
sParamError	C000	parameter error: `TrailingMinimum > TrailingMaximum` or `TrailingMinimum < 100 ms`
sResetError	C001	reset error: `ResetIn` was already TRUE when `ResetRequested` became TRUE
sWaitRTrig	83E2	active, waits for the button to be actuated (rising edge)
sArmTimers	83F2	one-shot: (re)start both timers, then `sWaitFTrig`
sWaitFTrig	83F2	button pressed, waits for release (falling edge)
sMinElapsed	83F2	one-shot: set `xMinOk`, then back to `sWaitFTrig`
sErrorMin	C3F0	error: released before `TrailingMinimum` elapsed, waits for a new rising edge
sErrorMax	C3E0	error: `TrailingMaximum` elapsed before release, waits for a new rising edge
sResetDetected	8000	valid reset, 1-cycle pulse on `ResetOut` and `EResetOut`

Application Scenarios

- Acknowledging a safety function (e.g. after an emergency stop) with a deliberate, time-limited button actuation
- Interplay with a block that requests the acknowledgement via `ResetRequest` and couples to `ResetRequested`

🛠️ Related Exercises

- Uebung_006_SAFE_AX

Conclusion

`SF_ResetButton` detects a valid trailing-edge reset according to PLCopen Safety, monitors the minimum and maximum actuation duration with two timers and reports errors via `Error` and `DiagCode`.

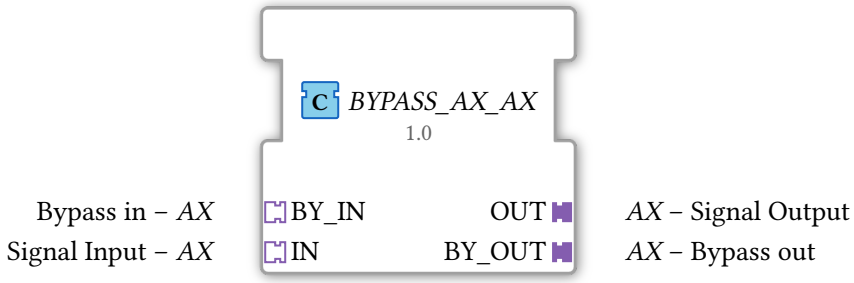
2.10.3.9 signal processing

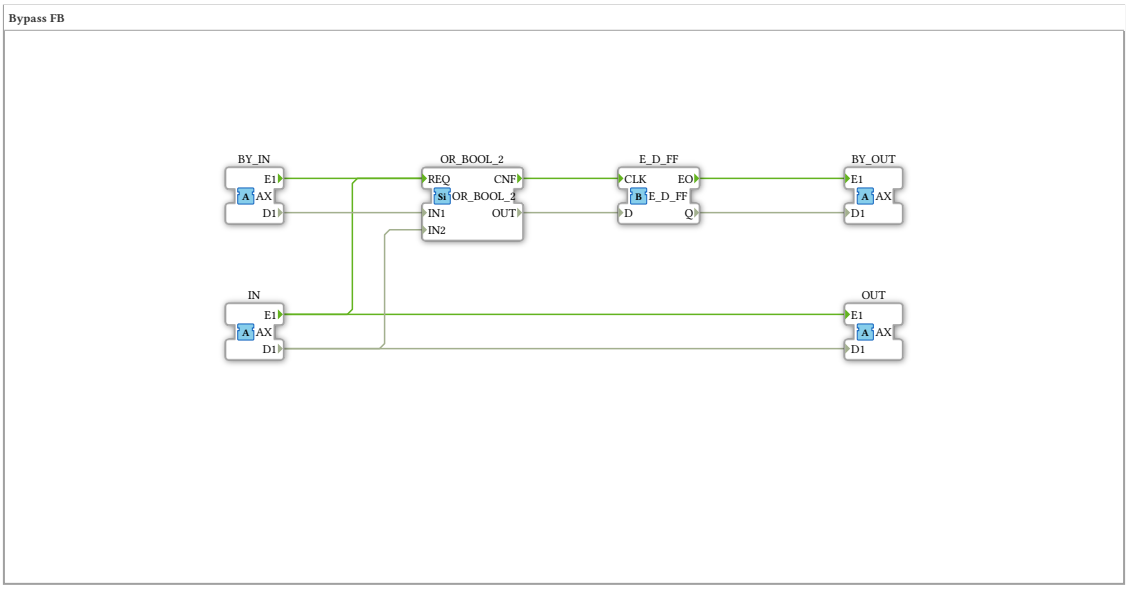
...

Numeric Index (signal processing)

Here you will find all exercises from the `signalprocessing` workspace, listed in ascending order.

BYPASS_AX_AX





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **BYPASS_AX_AX** is used for forwarding and parallel bypass processing of AX signals. It combines a direct pass from input **IN** to output **OUT** with a Boolean OR operation on the incoming data, which is passed to the bypass output **BY_OUT** via an edge-triggered D flip-flop.

Interface Structure

Event Inputs

- **IN.E1** – Signal Input Event
- **BY_IN.E1** – Bypass Input Event

Event Outputs

- **OUT.E1** – Signal Output Event
- **BY_OUT.E1** – Bypass Output Event

Data Inputs

- **IN.D1** – Signal Input Data Value (Type depends on the AX adapter used)
- **BY_IN.D1** – Bypass Input Data Value (same type)

Data Outputs

- **OUT.D1** – Signal Output Data Value (identical to IN.D1)
- **BY_OUT.D1** – Bypass Output Data Value (Result of the OR operation, stored in the flip-flop)

Adapters

The function block uses four adapters of type `adapter::types::unidirectional::AX`:

- **IN** (Socket) – Signal input
- **BY_IN** (Socket) – Bypass input
- **OUT** (Plug) – Signal output
- **BY_OUT** (Plug) – Bypass output

Each AX adapter transmits one event/data pair (E1/D1) in one direction.

Functionality

The function block implements two parallel paths:

1. **Direct Path:**

An event on **IN.E1** is passed directly to **OUT.E1**, as is the corresponding data value **IN.D1** to **OUT.D1**. This represents a 1:1 pass-through of the signal.

1. **Bypass Path:**

An event on both **IN.E1** and **BY_IN.E1** triggers processing in the function block `OR_2_BOOL`. This block performs a Boolean operation on the data values **IN.D1** and **BY_IN.D1** – a logical OR operation on the two binary inputs.

The result of the operation is applied to the data input **D** of the D flip-flop (`E_D_FF`). The flip-flop's clock is triggered by the feedback signal `OR_2_BOOL.CNF`, i.e., after the OR operation is complete.

The flip-flop's output **Q** is set to **BY_OUT.D1**, and the corresponding event **BY_OUT.E1** is triggered by the flip-flop's output **EO**.

Technical Features

- **Boolean OR Operation:** The data is not added or mixed, but rather logically ORed. Therefore, the data values are considered Boolean (TRUE/FALSE).
- **Edge-Triggered Flip-Flops:** The bypass value is updated whenever at least one of the two inputs (IN or BY_IN) provides an event. The flip-flop stores the last calculated OR value.
- **Independent Event Propagation:** The direct path is asynchronous – every event on IN is immediately passed to OUT without waiting for the bypass path.

State Overview

The FB itself does not have its own state machine; however, a binary state is implemented using the internal D flip-flop:

- **State 0:** Flip-flop output Q = FALSE ⟶ BY_OUT.D1 = FALSE
- **State 1:** Flip-flop output Q = TRUE ⟶ BY_OUT.D1 = TRUE

The state changes to the current D value on each rising edge of the clock (`OR_2_BOOL.CNF`). The initial state is undefined and may need to be initialized externally.

Application Scenarios

- **Bypass Circuit in Controllers:** A normal signal (IN ⟶ OUT) is to be passed through unchanged, while simultaneously creating an OR gate between the normal signal and an additional bypass signal for monitoring purposes.

State 0: - **Emergency Activation:** The bypass input can be used for manual override (e.g., "manual operation") while automatic operation continues at the main output.

- **Logical Signal Tracing:** The separate outputs allow both the original signal and the applied logic to be evaluated in parallel.

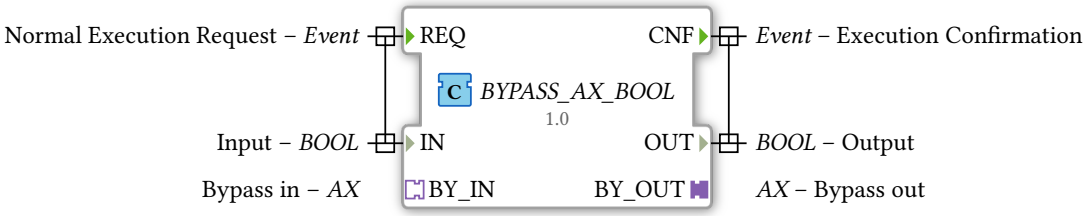
Comparison with Similar Function Blocks

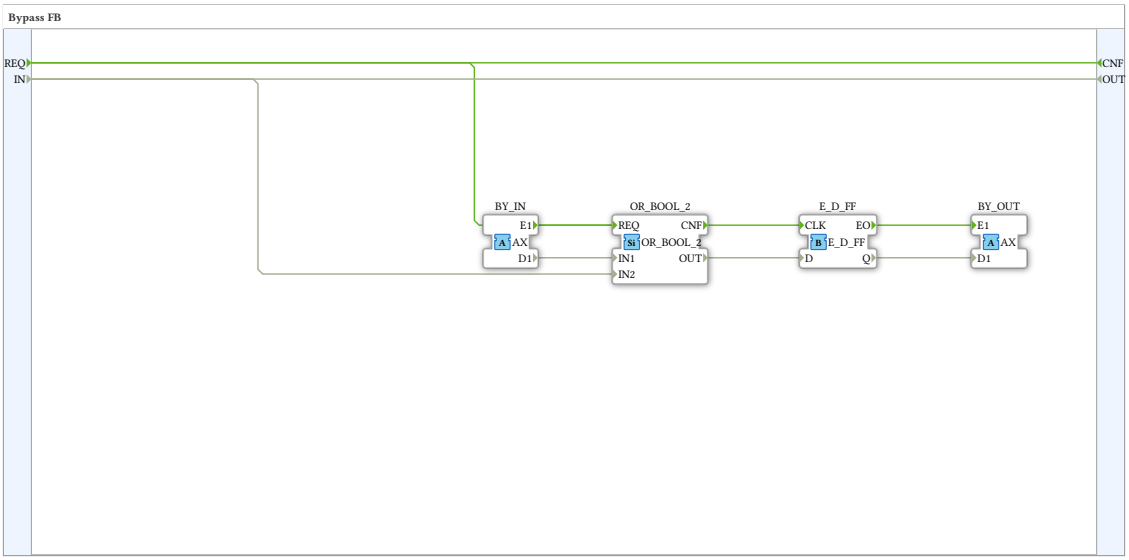
- **Simple Pass-Through Function Block (e.g., MOVE_AX):** Passes the signal through 1:1 but does not generate a bypass output. BYPASS_AX_AX additionally offers the logic gate and a second output.
- **OR Function Block without Flip-Flop:** A simple OR gate would continuously update the bypass output without buffering. The integrated flip-flop provides edge-triggered updates, which is advantageous for cyclical control or time-decoupled events.
- **Adapter-based solution with two separate function blocks:** While it would be possible to wire an OR function block and a flip-flop function block separately, BYPASS_AX_AX encapsulates this functionality in a single compact block, reducing the number of connections and simplifying handling.

Conclusion

The **BYPASS_AX_AX** is a useful function block for applications requiring both direct signal forwarding and a logical OR operation with a stored output. Combining the pass-through and bypass paths in a single block keeps the network structure clear. The use of unidirectional AX adapters makes it compatible with typical industrial communication protocols based on the IEC 61499 standard.

BYPASS_AX_BOOL





• – ○ • – ○ • – ○ •

Introduction

The function block `BYPASS_AX_BOOL` implements a bypass function for a Boolean signal. It passes the value of the input `IN` directly to the output `OUT` and simultaneously creates a bypass path connected via an adapter. This path allows an external signal (`BY_IN.D1`) to be inserted into the internal logic and outputs the result via the output adapter `BY_OUT`.

Interface Structure

Event Inputs

Name	Type	Description
REQ	Event	Normal Execution Trigger

Event Outputs

Name	Type	Description
CNF	Event	Execution Acknowledgement

Data Inputs

Name	Type	Description

IN	BOOL	Input Signal

Data Outputs

Name	Type	Description
OUT	BOOL	Pass-Through Input Signal (Identical to IN)

Adapters

Role	Name	Type	Direction	Description
Plug	BY_OUT	adapter::types::unidirectional::AX	Output	Outputs the stored bypass state
Socket	BY_IN	adapter::types::unidirectional::AX	Input	Receives the bypass input signal (D1) and event (E1)

Functionality

Upon each event `REQ`, the output `CNF` is immediately set (event pass-through). Simultaneously, the data word `IN` is copied directly to `OUT`.

Internally, an OR operation is performed between `IN` and the adapter signal `BY_IN.D1`. This operation is triggered by `REQ` or by the event `BY_IN.E1`. The result of the OR operation is stored in an edge-triggered D flip-flop; the clock edge is generated by the completion of the OR calculation (event `CNF` of the internal OR block). The stored value of the flip-flop is output via the output adapter `BY_OUT.D1`, and the corresponding event output `BY_OUT.E1` is activated.

The module thus functions as a combination of a simple passthrough for `IN → OUT` and an edge-triggered bypass path that links and stores the incoming adapter signal with the current `IN`.

Technical Features

- The direct connection to `IN → OUT` is established without delay via internal logic – the value is available immediately.
- The bypass path uses a D flip-flop (`E_D_FF`) for state storage. The stored value is only updated on a rising edge of the internal clock, which is generated by the OR operation.
- The adapter `BY_IN` must be connected to use the bypass path; without it, the module behaves like a simple passthrough.
- The event `BY_IN.E1` can trigger the OR operation independently of `REQ`, allowing the bypass to be updated asynchronously.

State Overview

The component does not have an explicit state machine, but the internal D flip-flop stores a Boolean state (`Q = 0` or `Q = 1`). This state represents the last valid bypass value, which is output via the output adapter. The state only changes on a rising edge of the clock signal (completion of an OR operation).

Application Scenarios

- **Manual Override** of a digital signal in automation technology: The normal value (`IN`) is passed to the controller, while a manual intervention (e.g., via an operator interface) can be added and stored via the bypass.
- **Signal Monitoring and Correction**: An external monitoring system can display a correction value via the bypass adapter, which is then linked to the original signal.
- **Bypass for Testing Purposes**: During commissioning, the normal path (IN→OUT) can be maintained, while a test signal can be inserted via the bypass.

Comparison with Similar Function Blocks

- **Simple Passthrough** (e.g., `MOVE`): Would only transfer `IN` to `OUT`, without the possibility of external intervention.
- **Set/Reset Flip-Flop** (e.g., `SR`): Allows setting and resetting an output, but without directly passing through the input.
- **Selection Block** (e.g., `SEL`): Selects between two inputs – in this case, the input is always passed through, and the bypass is additively configured.

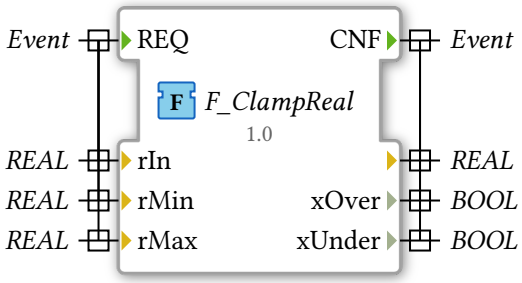
The BYPASS_AX_BOOL combines pass-through with integrated, edge-triggered logic for external signals.

Conclusion

The BYPASS_AX_BOOL function block is a specialized component for applications where a Boolean signal is passed through unchanged, but an external bypass signal is simultaneously mixed in and stored. The combination of direct passthrough and edge-triggered OR logic with memory enables flexible intervention in signal chains without interrupting normal operation. This component is particularly suitable for control systems that require both standard operation and temporary or permanent overrides.

Clamp

F_ClampReal



• – ◦ • – ◦ • – ◦ •

Introduction

The F_ClampReal function block (FB) restricts a REAL input value rIn to a defined range [rMin, rMax]. If rIn exceeds rMax, the output is set to rMax and the over-range flag is activated. If rIn is below rMin, the output is set to rMin and the under-range flag is set. Otherwise, the input passes through unchanged. This FB is useful for safe value limiting in control systems, signal conditioning, and boundary checking.

Interface Structure

The FB has one event input (REQ) and one event output (CNF). Data inputs (rIn, rMin, rMax) are read when REQ occurs, and the data outputs (rOut, xOver, xUnder) are provided with CNF.

Event Inputs

Event	Description
REQ	Trigger for performing the clamping operation.

Event Outputs

Event	Description
CNF	Confirmation that the operation has completed and outputs are valid.

Data Inputs

Name	Type	Description
rIn	REAL	Input value to be clamped.
rMin	REAL	Lower bound of the valid range.
rMax	REAL	Upper bound of the valid range.

Data Outputs

Name	Type	Description

rOut	REAL	Clamped result (equal to rIn if inside range). This corresponds to the unnamed output in the interface definition.
xOver	BOOL	TRUE if rIn > rMax, otherwise FALSE.
xUnder	BOOL	TRUE if rIn < rMin, otherwise FALSE.

Adapters

This FB does not use any adapters.

Functionality

Upon receiving the REQ event, the FB evaluates the following logic (written in Structured Text):

- If rIn > rMax, then rOut = rMax, xOver = TRUE, xUnder = FALSE.
- Else if rIn < rMin, then rOut = rMin, xOver = FALSE, xUnder = TRUE.
- Else, rOut = rIn, xOver = FALSE, xUnder = FALSE.

After the calculation, the CNF event is emitted, signalling that the output data are ready. The operation is purely combinatorial and does not rely on internal states.

Technical Features

- **Pure function**: No internal state, side effects, or memory usage.
- **Data types**: All inputs and the result are REAL (floating-point). Flags are BOOL.
- **Event-driven**: Uses the standard REQ/CNF handshake pattern for synchronous execution.
- **Range validation**: Assumes rMin ≤ rMax. If violated, behavior is undefined (but still follows the comparison logic).
- **Implementation**: Written in Structured Text, compatible with IEC 61499 and the 4diac IDE.

State Overview

The FB is stateless. It does not maintain any information between two invocations. Each execution is independent and based solely on the input values at the moment of REQ.

Application Scenarios

- **Safety interlocks**: Limit a setpoint or measured value to a safe operating range.
- **Signal conditioning**: For sensors that might produce out-of-range readings, e.g., clamping a temperature to a plausible band.
- **Control loops**: Prevent integral wind-up by limiting the controller output.
- **Validation**: Generate flags (xOver / xUnder) that can be used for alarming or diagnostics.

Comparison with Similar Blocks

Other clamping blocks may differ in:

- **Return type**: Some FBs only output the clamped value without flags; F_ClampReal additionally provides over- and under-range indicators.
- **Input type**: Analogous blocks exist for INT, LREAL, or other numeric types, but this FB is specifically for REAL.
- **Behavior on invalid range**: Some implementations swap rMin and rMax if rMax < rMin; this FB does not perform such a check.

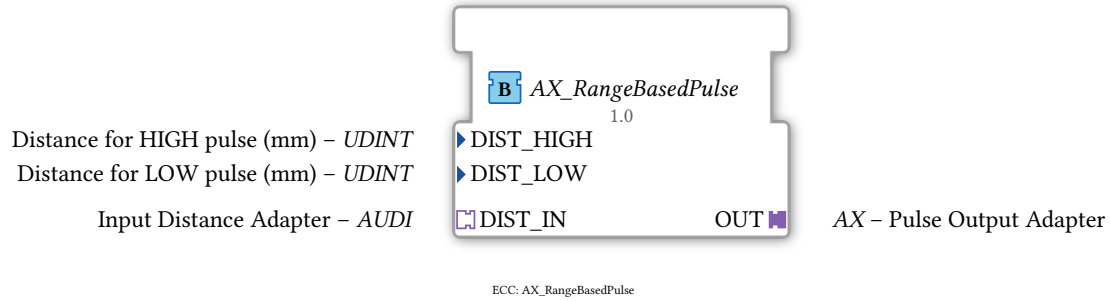
Conclusion

F_ClampReal is a simple, reliable, and reusable function block for clamping REAL values to a specified interval. Its clean interface, stateless design, and integrated flag outputs make it suitable for a wide range of control and monitoring tasks in industrial automation environments.

Numeric Index (signal processing)

Here you will find all exercises from the signalprocessing workspace, listed in ascending order.

AX_RangeBasedPulse



• – ○ • – ○ • – ○ •

Introduction

This function block generates a pulse on an AX adapter based on a distance traveled (adapter version). It is licensed under the Eclipse Public License 2.0 and designed for use in signal processing of distance data.

Interface Structure

Event Inputs

- `DIST_IN.E1`: Event from the distance adapter that signals an update of the distance value.

Event Outputs

- `OUT.E1`: Event on the output adapter that is sent when the pulse output changes state.

Data Inputs

- `DIST_HIGH` (UDINT): Distance in mm that must be traveled to generate the HIGH pulse (initial value: 5000).
- `DIST_LOW` (UDINT): Distance in mm that must be traveled to generate the LOW pulse (initial value: 5000).
- `DIST_IN.D1` (UDINT): Current distance from the input adapter (updated via the event).

Data Outputs

- `OUT.D1` (BOOL): Pulse output – toggles between TRUE (HIGH) and FALSE (LOW) depending on the accumulated distance.

Adapters

- **DIST_IN (Socket):** Adapter of type `adapter::types::unidirectional::AUDI`. Provides a unidirectional distance value (`D1`) and a corresponding event (`E1`).
- **OUT (Plug):** Adapter of type `adapter::types::unidirectional::AX`. Provides a unidirectional Boolean value (`D1`) and an event (`E1`).

Functionality

The function block operates according to a hysteresis principle with two threshold values:

1. **Initialization:** During the first iteration (`FirstRun = TRUE`), the current distance is stored as the start value `LastDist`, the output `OUT.D1` is set to FALSE, and `FirstRun` is reset.
2. **State Check:** On each incoming event via `DIST_IN.E1`, the block executes the `CHECK` algorithm:

- If the current output is LOW (FALSE) and the new distance `DIST_IN.D1` reaches or exceeds the value `LastDist + DIST_LOW`, `OUT.D1` is set to TRUE and `LastDist` is updated to the current distance value.
- If the output is HIGH (TRUE) and the new distance reaches or exceeds the value `LastDist + DIST_HIGH`, `OUT.D1` is set to FALSE and `LastDist` is updated.

1. **Output:** When the state of `OUT.D1` changes, the event `OUT.E1` is sent, and the new state is stored in `LastState` for the next check.

Technical Features

- **Adapter-based Communication:** The use of plugs and sockets (`AX`, `AUDI`) enables modular and flexible interconnection in the 4diac IDE.
- **Asymmetric Pulse Widths:** The separate parameters `DIST_HIGH` and `DIST_LOW` allow for different distances for HIGH and LOW phases.
- **Asymmetric Pulse Widths:** - **Initialization Logic:** An internal `FirstRun` flag ensures that the first measurement is correctly set as the starting point.
- **Package Context:** The function block (FB) is intended for the `logiBUS::signalprocessing::distance` package.

State Overview

The Execution Control Chart (ECC) consists of four states:

- **START:** Waiting state after initialization or processing.
- **INIT:** Executes the `INIT` algorithm (setting the initial values). This state is only entered on the first iteration (`FirstRun = TRUE`).
- **CHECK:** Executes the `CHECK` algorithm for distance checking and modifies the output if necessary.
- **EMIT:** Executes the `SAVE` algorithm (stores the current output state) and sends the event `OUT.E1`.

Transitions:

- `START → INIT`: When `DIST_IN.E1` and `FirstRun = TRUE` arrive.
- `START → CHECK`: When `DIST_IN.E1` and `FirstRun = FALSE` arrive.
- `INIT → CHECK`: Immediately after `INIT` (condition 1).
- `CHECK → EMIT`: If `OUT.D1` has changed compared to `LastState`.
- `CHECK → START`: If no change occurs (`OUT.D1 = LastState`).
- `EMIT → START`: Immediately after `EMIT` (condition 1).

Application Scenarios

- **Distance Pulse Generators in Agricultural Technology:** Generating pulses after each defined distance traveled (e.g., a HIGH pulse every 5m, then a LOW pulse after another 5m) for control or counting tasks.
- **Distance-Based Switching Functions:** Triggering actions in conveyor systems, automated guided vehicles (AGVs), or automated machinery after specific distances have been traveled.
- **Pulse Width Modulation over Distance:** Variable pulse widths can be achieved by setting different values for `DIST_HIGH` and `DIST_LOW`, e.g., 10m HIGH and 2m LOW.

Comparison with Similar Function Blocks

- **Tone-Based Pulse Generators (e.g., `E_CYCLE`):** Generate pulses based on time, while `AX_RangeBasedPulse` operates based on distance – for applications without a fixed time base.

- **Simple Edge Detectors:** Detect signal changes, but without hysteresis and without accumulating distance.
- **Other Distance Adapter Function Blocks:** Function blocks exist that aggregate distance values, but this function block combines event processing with threshold logic and asymmetric configuration.

**

Conclusion

The `AX_RangeBasedPulse` function block offers a robust and flexible solution for distance-based pulse generation. Thanks to its adapter interface, separately configurable distance thresholds, and hysteresis logic, it is ideally suited for modular use in industrial control systems, particularly in agricultural technology and automation. Its implementation as a BasicFB ensures deterministic and predictable behavior.

–

🌐 Related topic subpages on ms-muc-docs.de

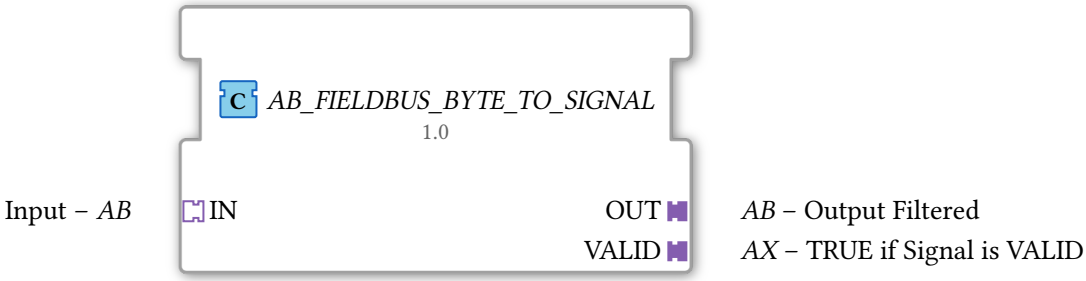
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 The PWM Signal & Infographic on ms-muc-docs.de

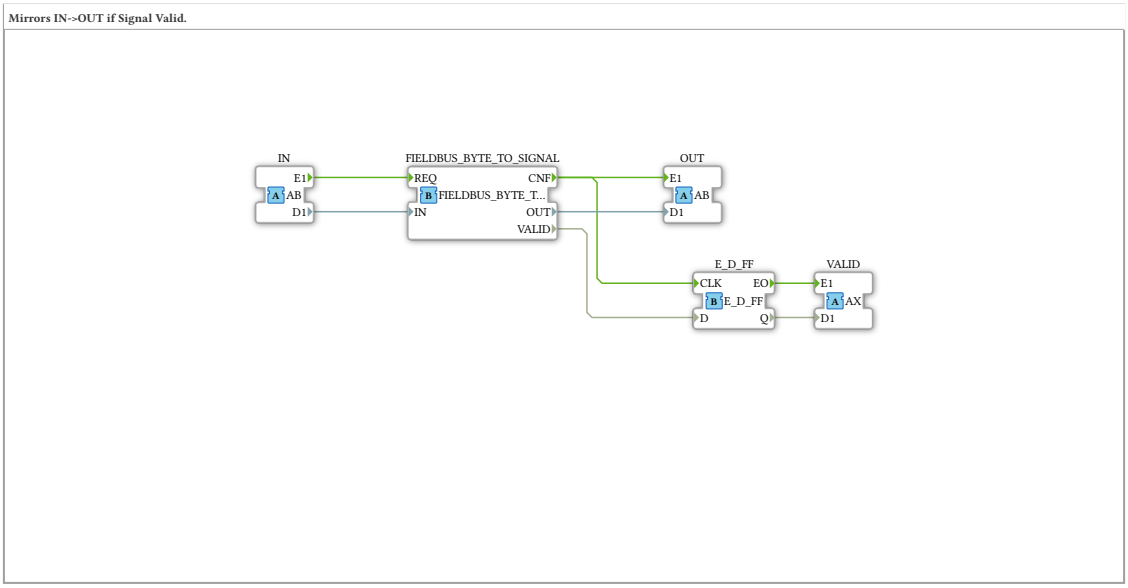
]

Numeric Index (signal processing)

Here you will find all exercises from the `signalprocessing` workspace, listed in ascending order.

AB_FIELDBUS_BYTE_TO_SIGNAL





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AB_FIELDBUS_BYTE_TO_SIGNAL** mirrors an incoming byte signal (via the adapter *IN*) to the output (*OUT*) if the signal is recognized as valid. Validity is indicated by a separate

output (*VALID*). The block encapsulates the processing of a fieldbus byte signal and ensures that only valid data is passed on to the subsequent logic. It is based on an internal `FIELDBUS_BYTE_TO_SIGNAL` block, supplemented by a D flip-flop for stable output of the validity signal.

Interface Structure

The function block has **no** direct event or data inputs/outputs at the top level. All communication takes place via three **adapter interfaces**:

Adapter	Direction	Type	Description
`IN`	Socket	`adapter::types::unidirectional::AB`	Input adapter for the byte signal and its associated event.
`OUT`	Plug	`adapter::types::unidirectional::AB`	Output adapter for the mirrored byte signal.
`VALID`	Plug	`adapter::types::unidirectional::AX`	Output adapter indicating the signal's validity status.

The adapters are of the **unidirectional** type, meaning they transmit data and events in one direction. The types `AB` and `AX` each contain one event input/output (E1) and one data input/output (D1, of type `ANY` and `BOOL`, respectively).

Event Inputs

None (events are received via the socket adapter `IN`).

Event Outputs

None (events are sent via the plug adapters `OUT` and `VALID`).

Data Inputs

None (data is received via the socket adapter `IN`).

Data Outputs

None (data is sent via the plug adapters `OUT` and `VALID`).

Adapters

IN (Socket)

- **E1**: Event input – triggers the processing of a new byte value.
- **D1**: Data input – the byte to be processed (e.g., a fieldbus datagram).

OUT (Plug)

- **E1**: Event output – activated after successful mirroring of a valid signal.
- **D1**: Data output – the mirrored byte signal (only if the value is valid).

VALID (Plug)

- **E1**: Event output – activated with every processing cycle, regardless of validity.
- **D1**: Data output (BOOL) – `TRUE` if the currently processed signal is valid; otherwise, `FALSE`.

Functionality

1. An incoming event at adapter `IN.E1` triggers the internal block `FIELDBUS_BYTE_TO_SIGNAL`, which processes the byte at `IN.D1`.
2. The internal block outputs `OUT` (the mirrored byte) and `VALID` (the validity information) at its outputs.
3. The output signal is immediately forwarded to adapter `OUT`, triggering an event (`OUT.E1`).
4. Simultaneously, the validity signal of the internal function block is applied to the D input of the D flip-flop (`E_D_FF`). The same event (`CNF` of the internal function block) clocks the flip-flop.
5. The flip-flop stores the validity value and outputs it stably to `VALID.D1`. An event is sent to `VALID.E1`.

Thus, the byte is always passed to `OUT` when it is valid. The validity state is retained until the next processing iteration.

Technical Features

- **Adapter-based communication**: The function block uses only adapter interfaces, which enables a modular and typed connection in the 4diac IDE.
- **D flip-flop for debouncing**: The validity signal is clocked by a flip-flop to ensure stability and prevent DIN rail effects.
- **License**: This component is released under the **Eclipse Public License 2.0** (Copyright HR Agrartechnik GmbH).
- **Compiler package**: `logiBUS::signalprocessing::fieldbus` – specifically for fieldbus applications.

State overview

The component does not have an explicit state machine at the top level; the states result from the interaction of the internal components:

State	Description
Idle	Waiting for an event at `IN.E1`.
Processing	Internal `FIELDBUS_BYTE_TO_SIGNAL` processes the byte; `OUT` and `VALID` are updated.
Valid stable	After the flip-flop has been clocked, `VALID.D1` remains stable until the next event.

The state is cycled through in each cycle.

Application Scenarios

- **Fieldbus Integration**: Receiving a byte value from a fieldbus (e.g., CAN, Profibus) and forwarding it to a control logic, whereby only valid telegrams are passed through.
- **Signal Conditioning**: Converting a raw byte stream into a clocked, validation-checked signal for downstream function blocks.
- **Error Detection**: The function block can be combined with an external validation checker connected to the `IN` adapter.

Comparison with Similar Function Blocks

Simpler **Mirror** function blocks forward a signal without validation. This function block adds explicit validation and outputs the validity signal separately. Unlike a purely flip-flop-based hold function block, it processes a byte and not just Boolean values. The internal `FIELDBUS_BYTE_TO_SIGNAL` handles the specific fieldbus interpretation.

Conclusion

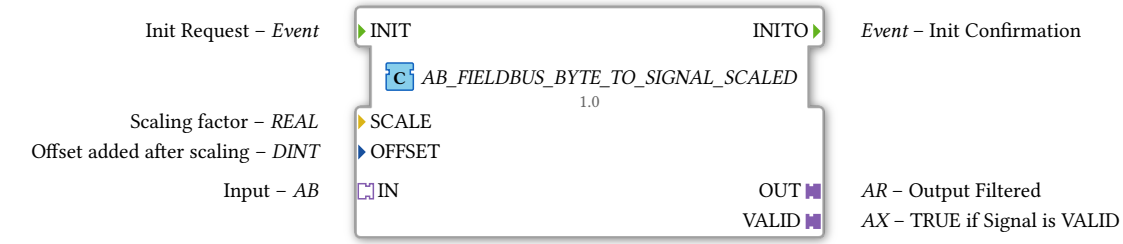
The `AB_FIELDBUS_BYTE_TO_SIGNAL` is a specialized function block for the safe transmission of fieldbus byte signals. The combination of mirroring and validation makes it ideal for real-time applications where only valid data may be processed. The adapter-based interface allows for easy integration into existing 4diac projects.

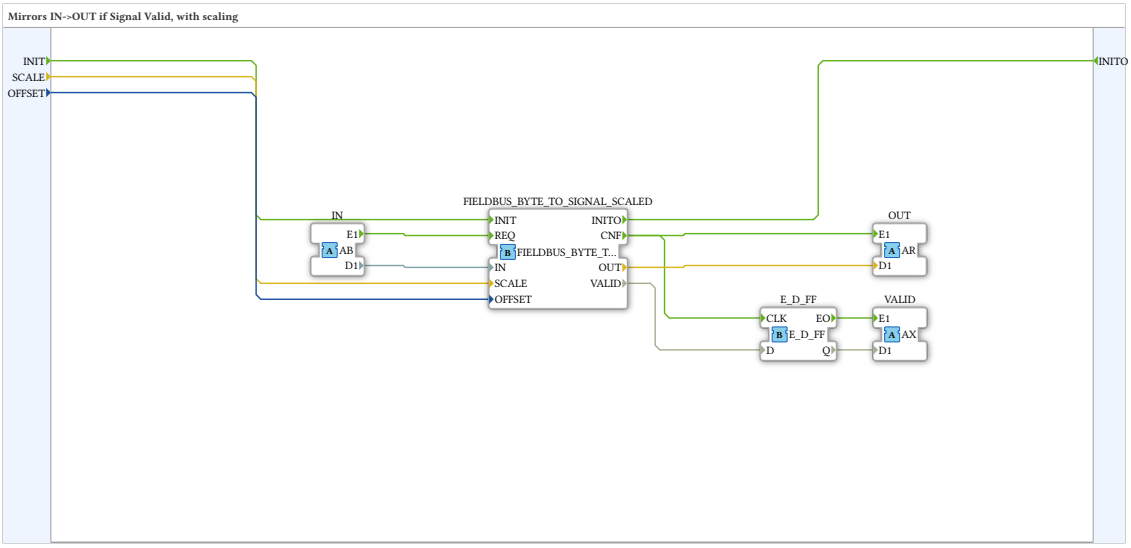
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AB_FIELDBUS_BYTE_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Introduction

The function block `AB_FIELDBUS_BYTE_TO_SIGNAL_SCALED` is used to receive an incoming byte-shaped signal via an adapter, process it with a configurable scaling factor and offset, and provide the result via an output adapter. Simultaneously, the validity of the signal is indicated via a separate adapter. This function block is specifically designed for use in fieldbus environments where raw data needs to be converted into physical or standardized values.

Interface Structure

Event Inputs

Event	Type	Description
`INIT`	EInit	Initialization request (e.g., for setting parameters)

Event Outputs

Event	Type	Description
`INITO`	EInit	Confirmation of successful initialization

Data Inputs

Name	Type	Initial Value	Description
`SCALE`	REAL	1.0	Scaling factor applied to the raw value
`OFFSET`	DINT	0	Integer offset added after scaling

Data Outputs

The module has no independent data inputs or outputs – input/output is handled exclusively via adapters.

Adapters

Adapter	Type	Direction	Description
`IN`	AB (Adapter Type A⟶B)	Socket	Receives the raw BYTE signal
`OUT`	AR (Adapter Type A⟶R)	Plug	Outputs the scaled signal

VALID	AX (Adapter Type A⟶X)	Plug	Signals whether the current output value is valid (TRUE = valid)

Functionality

1. After successful initialization (INIT ⟶ INITO), the function block becomes operational.
2. As soon as a new BYTE arrives via the IN adapter (event E1), the event is forwarded to the internal function block FIELDBUS_BYTE_TO_SIGNAL_SCALED.
3. This internal block calculates the output value: OUT = IN * SCALE + OFFSET. Simultaneously, a validity signal (VALID) is generated.
4. After the calculation is complete, the result is output via the OUT adapter (OUT.E1). Simultaneously, the validity signal is temporarily stored in a D flip-flop (module E_D_FF) and output via the VALID adapter.
5. The valid status is maintained until a new signal is processed. An invalid state (e.g., due to a missing input) can be set externally using appropriate logic.

The process is purely event-driven: Each incoming event triggers exactly one processing step.

Technical Features

- **Scaling and Offset**: The configuration of SCALE (REAL) and OFFSET (DINT) allows for flexible adaptation to different measurement ranges.
- **Adapter Interface**: The use of IEC 61499 adapters allows for loose coupling and easy reusability in different fieldbus contexts.
- **Valid Storage**: The D flip-flop (E_D_FF) ensures that the validity status remains stable until the next signal arrives.
- **Initialization**: The function block has its own INIT path to reliably set internal parameters (SCALE, OFFSET).

State Overview

The function block does not define an explicit state machine (ECC). Processing is exclusively event-driven:

- **In idle state**, it waits for an event.
- **On INIT**, the internal function block is initialized.
- **On IN.E1**, the translation is triggered, and the results are output asynchronously.

Thus, the function block behaves like a combinational, but clocked, function.

Application Scenarios

- **Fieldbus Signal Conditioning**: Receiving raw data (e.g., pressure sensor values as BYTE) and converting it into physical units (e.g., bar) using scaling and offset.
- **Normalization of Measured Values**: Adapting to a standardized value range, e.g., 0…100% or −50… +50°C.
- **Validity Check**: If the sensor only sends a signal when the measurement is valid, the Validation Adapter can be used to control downstream blocks.

Comparison with Similar Function Blocks

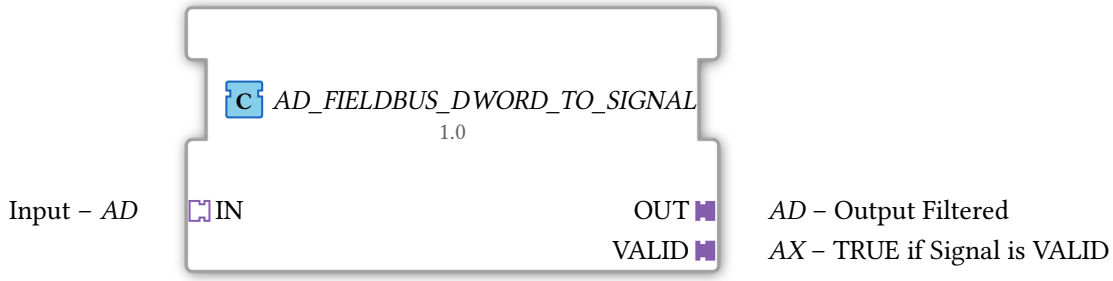
- **AB_FIELDBUS_BYTE_TO_SIGNAL** (without “SCALED”): Does not offer scaling/offset but transmits the raw value directly.
- **AB_FIELDBUS_SIGNAL_SCALED** (based on REAL): May work directly with REAL values but requires a different adapter configuration.

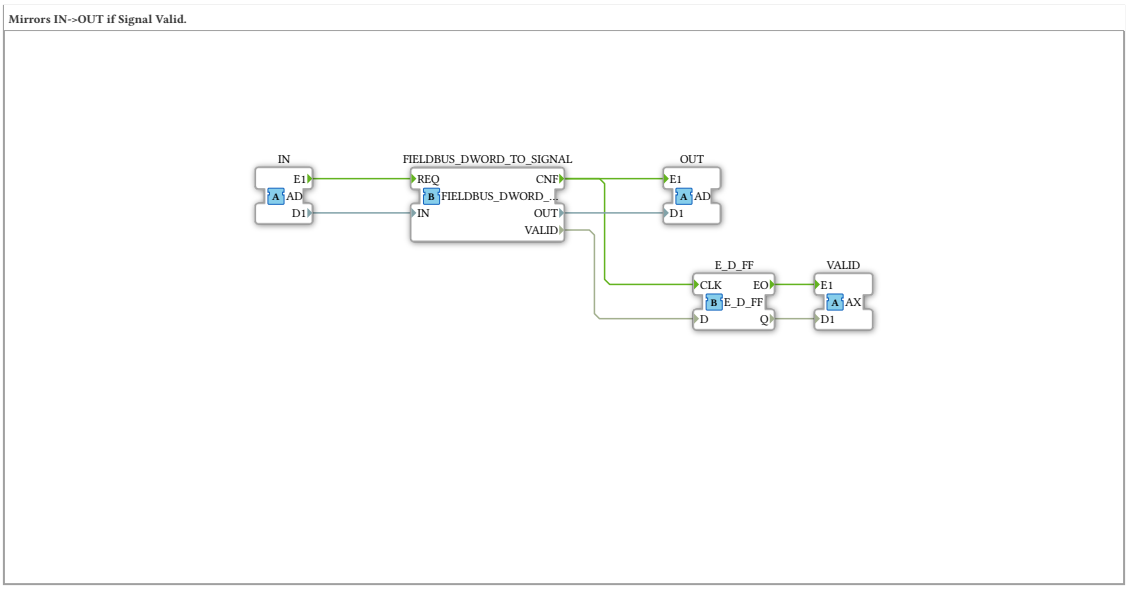
This module combines BYTE-to-REAL conversion with compact, configurable scaling and an output validity signal.

Conclusion

AB_FIELDBUS_BYTE_TO_SIGNAL_SCALED` is a specialized function block for the flexible conversion of raw BYTE data into scaled values with integrated validity signaling. Its adapter-based interface allows for easy integration into IEC 61499 systems and makes it particularly suitable for fieldbus applications where raw values need to be transformed into physical quantities.

AD_FIELDBUS_DWORD_TO_SIGNAL





• – ○ • – ○ • – ○ •

Introduction

The function block `AD_FIELDBUS_DWORD_TO_SIGNAL` is used to forward an incoming data word (DWORD) to the output – but only if the signal is classified as valid. It combines a specialized fieldbus function block with an edge-triggered D flip-flop to implement reliable, data-driven validation and forwarding.

Interface Structure

The function block has only adapter interfaces; there are no separate event or data ports at the top level. The following adapters define the inputs and outputs:

Adapter	Type	Direction	Description
`IN`	`adapter::types::unidirectional::AD`	Socket (Input)	Input data word and associated event
`OUT`	`adapter::types::unidirectional::AD`	Plug (Output)	Filtered output data word (DWORD)
`VALID`	`adapter::types::unidirectional::AX`	Plug (Output)	Validation signal (BOOL) of the current data word

Event Inputs (via adapter `IN`)

Port	Description
`E1`	Event to start processing a new data word

Event Outputs (via adapters `OUT` and `VALID`)

Adapter	Port	Description
`OUT`	`E1`	Signals that the filtered data word is present at the output
`VALID`	`E1`	Signals that the validity status (TRUE/FALSE) has been updated

Data Inputs (via adapter `IN`)

Port	Type (assumed)	Description
`D1`	`DWORD`	The data word to be processed

Data Outputs (via adapters `OUT` and `VALID`)

Adapter	Port	Type (assumed)	Description
`OUT`	`D1`	`DWORD`	The filtered – possibly identical – data word
`VALID`	`D1`	`BOOL`	`TRUE` if the attached data word is considered valid, otherwise `FALSE`

Adapters

The adapters are of type `unidirectional`, meaning they each transmit one event and one piece of data in one direction. The function block (FB) uses two different adapter types:

- **AD**: Transmits an event and a data word (DWORD)
- **AX**: Transmits an event and a Boolean signal

Functionality

The interaction of the internal function blocks can be described in the following steps:

1. An event at `IN.E1` triggers the internal FB `FIELDBUS_DWORD_TO_SIGNAL` via its `REQ` input.
2. The internal FB processes the incoming data word (`IN.D1`) and outputs two results:

- The (possibly identical) data word at `OUT`
- A Boolean signal `VALID` indicating whether the data word is valid.

1. After processing is complete, the internal function block (FB) signals with `CNF`:

- The event is forwarded to `OUT.E1` ⟶ the output adapter releases the new data word.
- Simultaneously, the event clocks the edge-triggered D flip-flop `E_D_FF` via its `CLK`.

1. The flip-flop receives the current validity status (`VALID` signal) from `FIELDBUS_DWORD_TO_SIGNAL` at its `D` input and outputs it at its `Q` output.

The flip-flop receives the current validity status (`VALID` signal) from `FIELDBUS_DWORD_TO_SIGNAL` at its `D` input and outputs it at its `Q` output. 5. The `EO` output of the flip-flop generates an event that is sent to `VALID.E1` – thus updating the validity status in sync with the data word.

In other words: The function block *mirrors* the input DWORD to the output, provided the fieldbus function block's internal validation mechanism classifies it as valid. Validity is stabilized by a flip-flop and output as a separate signal.

Technical Features

- **Composite Function Block**: The function block is implemented as a network of two subordinate function blocks (`FIELDBUS_DWORD_TO_SIGNAL` and `E_D_FF`).
- **License**: The function block is subject to the Eclipse Public License 2.0 (EPL-2.0).
- **Packet Structure**: Integrated into the namespace `logiBUS::signalprocessing::fieldbus`.
- **Edge-Triggered Validation Memory**: The use of a D flip-flop ensures that the `VALID` signal is only updated with the next clock edge (the `CNF` event) – this prevents asynchronous state transitions.
- **No Dedicated Event/Data Ports**: All communication takes place exclusively via standardized adapters.

State Overview

The function block (FB) does not have an explicit state machine, but operates purely on data flow. The internal flip-flop `E_D_FF` has two internal states:

State	Description
`Q = FALSE`	The currently transmitted `VALID` signal is `FALSE` (data word is considered invalid)
`Q = TRUE`	The currently transmitted `VALID` signal is `TRUE` (data word is considered valid)

The state changes only on a rising edge at `CLK` (corresponds to the `CNF` event of the internal fieldbus module).

Application Scenarios

- **Fieldbus Data Filtering**: Used in PLC controllers where only valid telegrams from a bus system (e.g., CANopen, PROFIBUS) should be passed on to the application logic.
- **Quality Marking**: A sensor provides a measured value and a validation bit – the function block cleanly separates both pieces of information and keeps them synchronized.
- **Secure Data Transmission**: In safety-critical environments, the function block can be used to only allow verified data words into subsequent calculations.

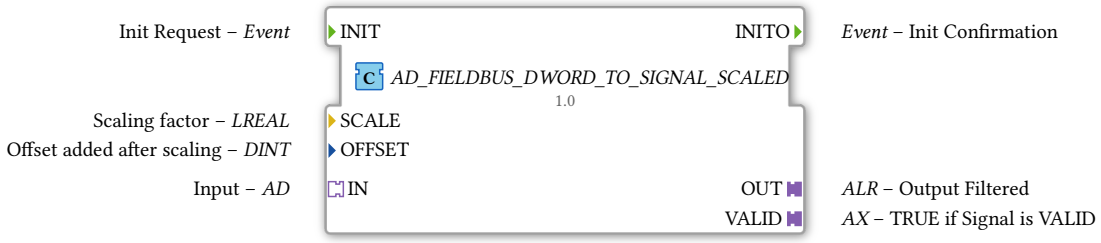
Comparison with Similar Function Blocks

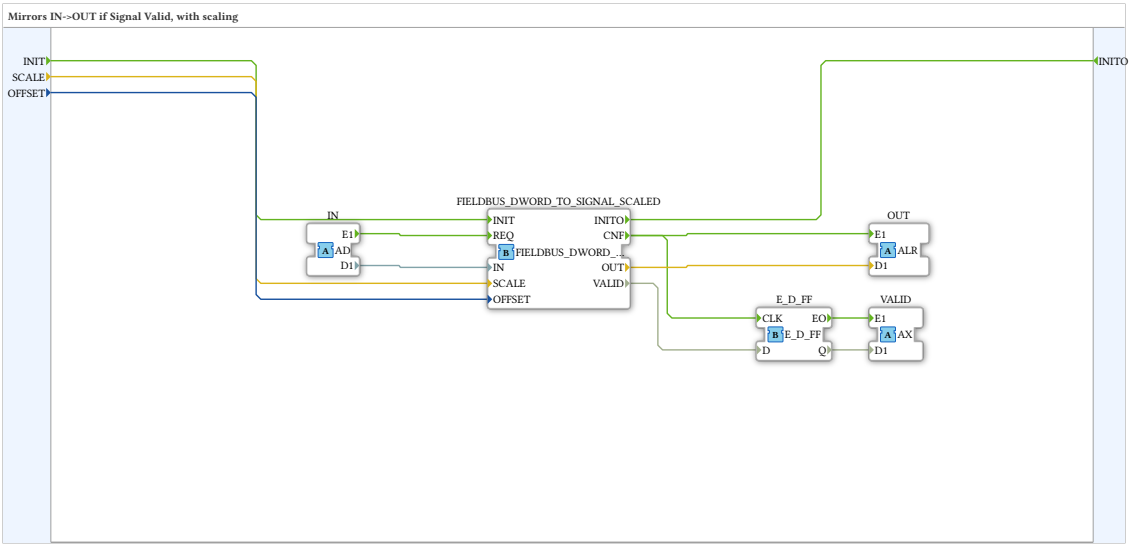
- **Simple Buffer Blocks (e.g., `MOVE`)**: These pass data without evaluation. `AD_FIELDBUS_DWORD_TO_SIGNAL` adds validation logic and separates the validity signal.
- **Typical quality function blocks (e.g., `QHandler`)**: These often process multiple quality bits; this function block focuses on a single Boolean signal, `VALID`, and works with DWORD data.
- **Adapter-based function blocks**: The pure adapter interfaces promote loose coupling and reusability in various runtime environments.

Conclusion

AD_FIELDBUS_DWORD_TO_SIGNAL` offers a compact and standardized solution for forwarding fieldbus data words to the application only when validity has been established. The combination of a specific fieldbus function block and a flip-flop ensures a timely and stable output of the validation status. Due to its adapter-based interface, the function block can be flexibly integrated into existing IEC 61499 systems.

AD_FIELDBUS_DWORD_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Introduction

The function block `AD_FIELDBUS_DWORD_TO_SIGNAL_SCALED` processes a digital fieldbus signal (DWORD) into a scaled analog output value. It mirrors the input value to the output, provided the signal is valid, and applies linear scaling with an offset. The valid signal state information is synchronized via an edge-triggered flip-flop and provided as a separate output signal.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initialization request for the entire function block (forwarded to internal core function block)

Event Outputs

Event	Type	Description
INITO	EInit	Initialization confirmation after successful initialization

Data Inputs

Name	Data Type	Initial Value	Description
SCALE	LREAL	1.0	Scaling factor by which the input value is multiplied
OFFSET	DINT	0	Integer offset that is added after scaling

Data Outputs

This module has no direct data outputs. All output data is provided via the adapter interfaces.

Adapters

Name	Type	Direction	Description
IN	AD (unidirectional)	Socket	Input adapter for the DWORD value (fieldbus signal)
OUT	ALR (unidirectional)	Plug	Output adapter for the scaled analog value
VALID	AX (unidirectional)	Plug	Output adapter for signal validity (TRUE = valid)

Functionality

The module implements the following signal chain:

1. **Initialization**: The event `INIT` is forwarded directly to the internal core module `FIELDBUS_DWORD_TO_SIGNAL_SCALED`. Upon completion, `INITO` is output.
2. **Data Processing**: The DWORD value present at socket `IN` is inherited with each data event (E1). The core module multiplies the value by `SCALE` (LREAL) and adds `OFFSET` (DINT). The result is output as LREAL to plug `OUT`.
3. **Validity Signal**: The core module outputs an internal Boolean validity signal indicating whether the current data value is valid. This signal is synchronized via an edge-triggered D flip-flop (`E_D_FF`) and output to the plug `VALID`. The flip-flop clocks with the same event (`CNF` of the core module), so the validity is updated simultaneously with the output value.

The internal circuitry ensures that the validity status always corresponds to the last processed data value.

Technical Features

- **Scaling Order**: First, multiplication by `SCALE` (LREAL), then addition by `OFFSET` (DINT). This allows the offset to be specified in units of the scaled range.
- **Edge Synchronization**: The validity signal is output via a D flip-flop using edge triggering, so there are no metastable states on the output side.
- **Initial Value Default**: `SCALE` is predefined to 1.0, `OFFSET` to 0. In the unscaled case, the output thus directly corresponds to the input.
- **Adapter Structure**: The inputs and outputs are implemented as adapters, enabling type-safe connections with other devices from the same adapter family (AD, ALR, AX).

State Overview

The device does not have its own explicit state diagram (ECC). All state logic resides in the internal core device `FIELDBUS_DWORD_TO_SIGNAL_SCALED` and the edge-triggered D flip-flop. This stores the last valid validity state. With each processing event, the flip-flop is clocked and outputs the current validity value.

Application Scenarios

- **Fieldbus Data Acquisition**: A raw DWORD value (e.g., from an analog-to-digital converter or encoder) is converted using physical factors (scaling) and offset and transmitted as an analog value to a controller.
- **Signal Validation with Storage**: When the fieldbus provides a validity flag (e.g., "Data Updated"), this can be evaluated via the validity adapter level. The flip-flop ensures that the validity signal remains stable until the next update.
- **Configurable Scaling**: By externally specifying `SCALE` and `OFFSET`, the same function block can be used for different sensor ranges without modifying the function block itself.
-

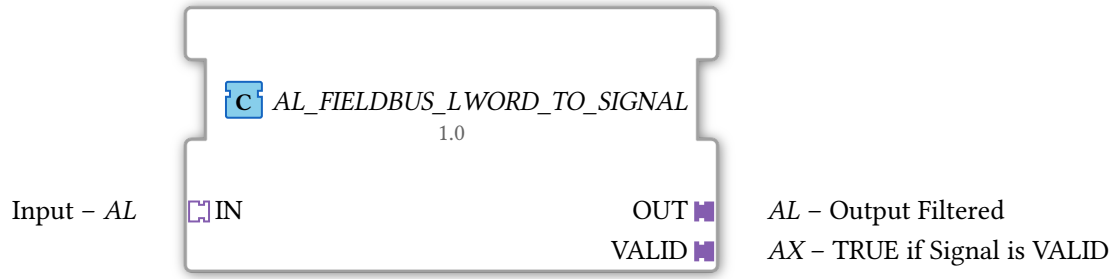
Comparison with Similar Function Blocks

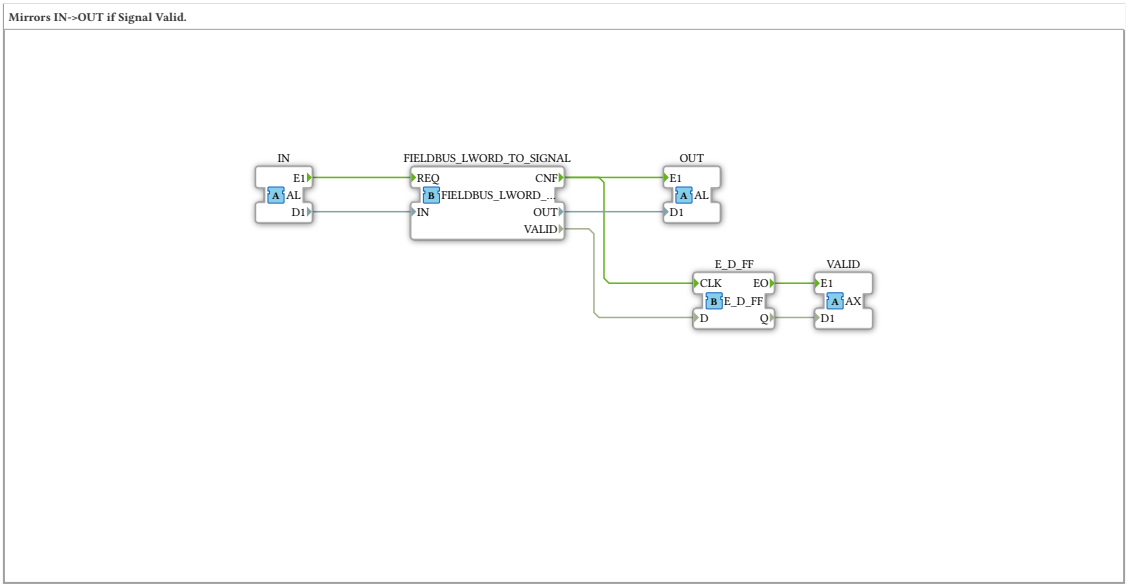
Compared to a simple `MOVE` or `SCALE` function block, this FB offers integrated validation handling and a clean separation between data and event adapters. Similar function blocks (e.g., `FIELD_DWORD_TO_ANALOG`) often lack separate signal validity output or edge-synchronous storage. The combination of core function block and D flip-flop used here is a proven solution for safety-critical applications where the validity of a data value must be reliably maintained.

Conclusion

The `AD_FIELDBUS_DWORD_TO_SIGNAL_SCALED` function block is a robust and configurable function block for converting digital fieldbus signals into scaled analog values with synchronous validity output. Thanks to its modular adapter connections and integrated edge synchronization, it is particularly suitable for industrial automation systems where data consistency and signal quality are equally important.

AL_FIELDBUS_LWORD_TO_SIGNAL





• – ○ • – ○ • – ○ •

Introduction

The function block `AL_FIELDBUS_LWORD_TO_SIGNAL` filters and passes a fieldbus signal of type `LWORD`. It only passes the input value to the output if the corresponding valid signal is set. The valid status is provided via a separate output and stabilized by an internal flip-flop. The block is implemented as a composite function block and encapsulates the signal processing logic and the valid signal state.

Interface Structure

Event Inputs

- **`IN.E1`** (via socket `IN`): Event to request signal processing. An incoming event triggers the mirroring of the input value to the output and updates the validity status.

Event Outputs

- **`OUT.E1`** (via plug `OUT`): Event output after successful processing. Signals that the current output value (`OUT.D1`) is valid.
- **`VALID.E1`** (via plug `VALID`): Event output simultaneously with `OUT.E1`. Confirms the update of the validity signal (`VALID.D1`).

Data Inputs

- **`IN.D1`** (via socket `IN`, type: `LWORD`): The fieldbus signal to be processed. The value is passed unchanged to the output, provided the internal validity signal is active.

Data Outputs

- **`OUT.D1`** (via plug `OUT`, type: `LWORD`): The filtered output signal. Contains the value of `IN.D1` if the signal is valid; otherwise, the last valid value is retained (due to the internal behavior of the function block used, `FIELDBUS_LWORD_TO_SIGNAL`).

- **VALID.D1** (via plug VALID, type: BOOL): Indicates whether the currently output signal is valid. The value is buffered by an internal flip-flop (E_D_FF) and updated with each processing cycle.

Adapter

Adapter	Direction	Type	Description
IN	Socket	adapter::types::unidirectional::AL	Receives the event and data signal from an upstream component.
OUT	Plug	adapter::types::unidirectional::AL	Sends the filtered signal and associated event to subsequent function blocks.
VALID	Plug	adapter::types::unidirectional::AX	Sends the validity status as a Boolean value and an acknowledgment event.

Functionality

This function block operates as a composite function block with the following internal logic:

1. An incoming event at IN.E1 triggers the internal function block FIELDBUS_LWORD_TO_SIGNAL via its REQ input.
2. This internal function block (FB) reflects the value of IN.D1 to its output OUT and simultaneously generates a Boolean signal VALID indicating whether the value is valid.
3. The internal FB's CNF event is distributed to three destinations:

- It triggers the output plug OUT.E1, so the filtered value (OUT.D1) is passed to the next processing stage.
- It triggers the CLK input of the flip-flop E_D_FF.

1. The VALID signal of the internal function block is applied to the D input of the flip-flop. On each rising edge of the clock (triggered by CNF), the flip-flop takes this value and outputs it to Q.
2. The output Q of the flip-flop is set to VALID.D1, and the simultaneous event VALID.E1 is triggered by the flip-flop event EO.

This ensures that the validity status is only passed to the Valid output once the signal processing is complete. The output value OUT.D1 corresponds to the input value IN.D1, provided it was recognized as valid; otherwise, the last valid value is retained (depending on the implementation of the internal function block FIELDBUS_LWORD_TO_SIGNAL).

Technical Features

- **Composite Architecture:** The functionality is implemented through an internal network consisting of a specialized function block (FIELDBUS_LWORD_TO_SIGNAL) and an edge-triggered flip-flop (E_D_FF). This enables a modular and testable implementation.
- **State Maintenance:** The validity status is stored via a flip-flop. This ensures that the valid output remains stable between processing cycles and is only updated upon new events.
- **Unidirectional Adapters:** All interfaces are implemented as unidirectional adapters (AL or AX), which allows for a clear separation of event and data flow.
- **Licensing:** This function block is available under the Eclipse Public License 2.0 (Copyright HR Agrartechnik GmbH).

State Overview

This function block does not have an explicit ECC, as it is a composite function block. The internal state logic is limited to the flip-flop E_D_FF, which has two states:

State	Q (VALID.D1)	Meaning
RESET (Initial)	FALSE	Signal is invalid (initial).
SET	TRUE	Signal is valid after a valid value has been detected.

The state change occurs with each incoming event at CLK (from CNF) depending on the D input (from the internal VALID signal). The state is maintained until the next event arrives.

Application Scenarios

- **Fieldbus Signal Processing:** Acquisition of LWORD signals (e.g., from a CANopen or Profibus network) and their transmission only if the status is valid.
- **Filtering of Invalid Values:** If a sensor temporarily delivers invalid measured values, the function block transmits the last valid value and signals the invalidity via `VALID.D1`.
- **Synchronization of Multiple Paths:** Due to the separate outputs for signal and valid, downstream function blocks can process both pieces of information simultaneously.

Comparison with Similar Function Blocks

Simpler function blocks like `MOVE` or `REPEAT` pass incoming values unfiltered, without validation. `AL_FIELDBUS_LWORD_TO_SIGNAL` extends this functionality with validation logic and a separate validation output. Other function blocks from the `logiBUS` library might offer similar tasks for other data types (e.g., `WORD` or `BOOL`), but they are not specifically designed for `LWORD`.

Conclusion

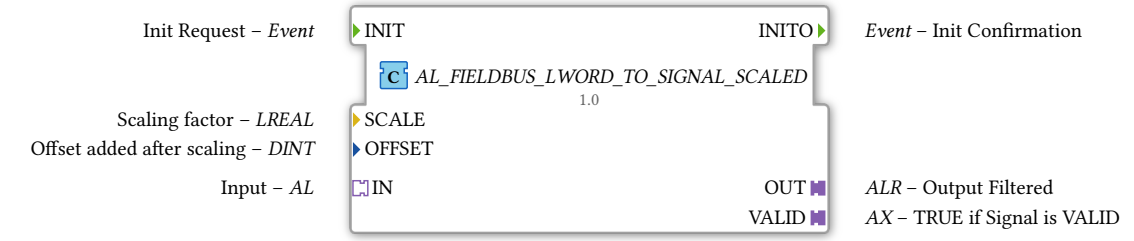
The `AL_FIELDBUS_LWORD_TO_SIGNAL` Function Blockis a specialized composite component for the reliable transmission of fieldbus signals while adhering to a validity criterion. Its combination of signal processing and state management makes it ideal for safety-critical or quality-conscious applications in automation technology. The clearly defined adapter interfaces facilitate integration into existing 4diac IDE projects.

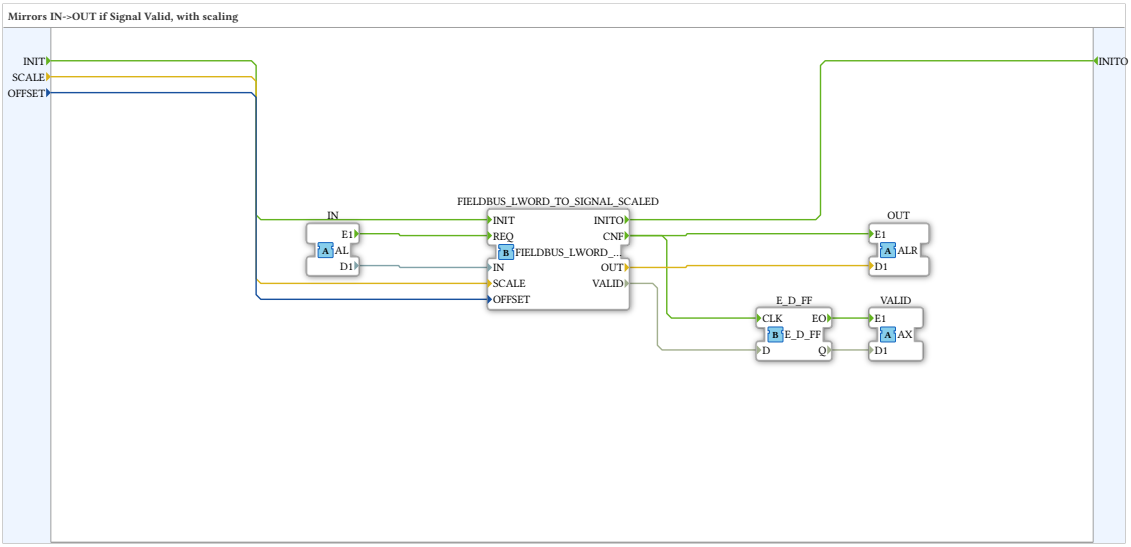
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AL_FIELDBUS_LWORD_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Introduction

The function block `AL_FIELDBUS_LWORD_TO_SIGNAL_SCALED` converts a fieldbus signal (LWORD) into a scaled value with an offset. A valid output signal is only generated if the input signal is marked as valid. Validity is maintained via an internal D flip-flop and output as a Boolean signal.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirmation

Data Inputs

Name	Type	Initial Value	Comment
SCALE	LREAL	1.0	Scaling Factor
OFFSET	DINT	0	Offset Added After Scaling

Adapters

Type	Direction	Name	Comment
AL	Socket	IN	Input Signal (LWORD)
ALR	Plug	OUT	Output signal (scaled)
AX	Plug	VALID	Signal valid (TRUE/FALSE)

Functionality

The function block contains two internal components:

- **FIELDBUS_LWORD_TO_SIGNAL_SCALED**: Performs the actual conversion of the LWORD input using the SCALE and OFFSET parameters. Upon an event at `IN.E1` (connected to `REQ`), the processed value is set to `OUT` and the validity signal to `VALID`.
- **E_D_FF**: An edge-triggered D flip-flop that takes the validity signal from the previous function block. The clock signal (`CLK`) is triggered by the acknowledgment event (`CNF`) of the first block. The output `Q` is written to the adapter `VALID.D1`.

This results in the following sequence:

1. An incoming event at `IN.E1` starts the processing.
2. The internal block calculates the scaled value and outputs the validation signal.
3. The flip-flop stabilizes this signal and passes it to the `VALID` adapter.
4. Simultaneously, the scaled value is output via the `OUT` adapter.

The INIT interface is used for initialization and is passed through directly.

Technical Features

- The INIT and INITO connections are marked as invisible in the network, but they exist for a clean initialization cascade.
- Scaling and offset are passed internally to the subordinate function block (FB) and are directly configurable.
- The D flip-flop prevents unwanted edges or momentary invalidity changes at the output.

State Overview

The FB does not have an explicit state machine (ECC). Its behavior is controlled by event processing:

- **After INIT**: The FB is ready to receive events at `IN.E1`.
- **After each REQ/CNF cycle**: A new scaled value and an updated validity signal are available at the adapters.

Application Scenarios

Typical application areas include:

- Conversion of raw fieldbus data (e.g., LWORD from PROFIBUS, CAN, etc.) into physical quantities (temperature, pressure, speed) with user-defined scaling.
- Forwarding of sensor data where only values marked as valid should be passed to the subsequent logic.
- Integration into plant control systems that require a clean initialization sequence (via INIT).

Comparison with Similar Function Blocks

Simpler LWORD conversion function blocks often lack integrated validity checking and output the signal directly. `AL_FIELDBUS_LWORD_TO_SIGNAL_SCALED` adds this functionality with:

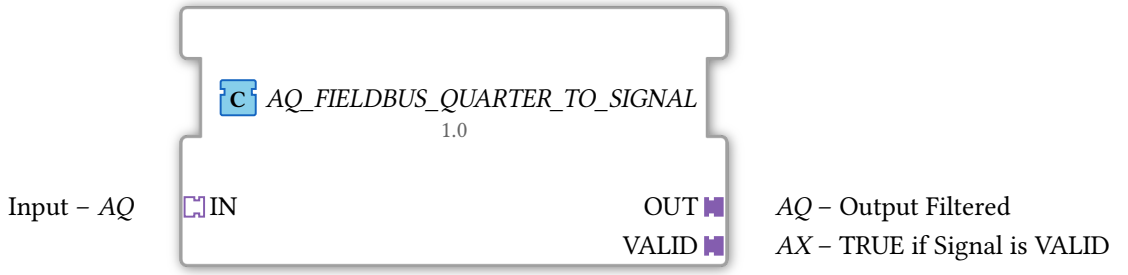
- A separate validity signal, edge-triggered via a D flip-flop.
- An explicit initialization interface.

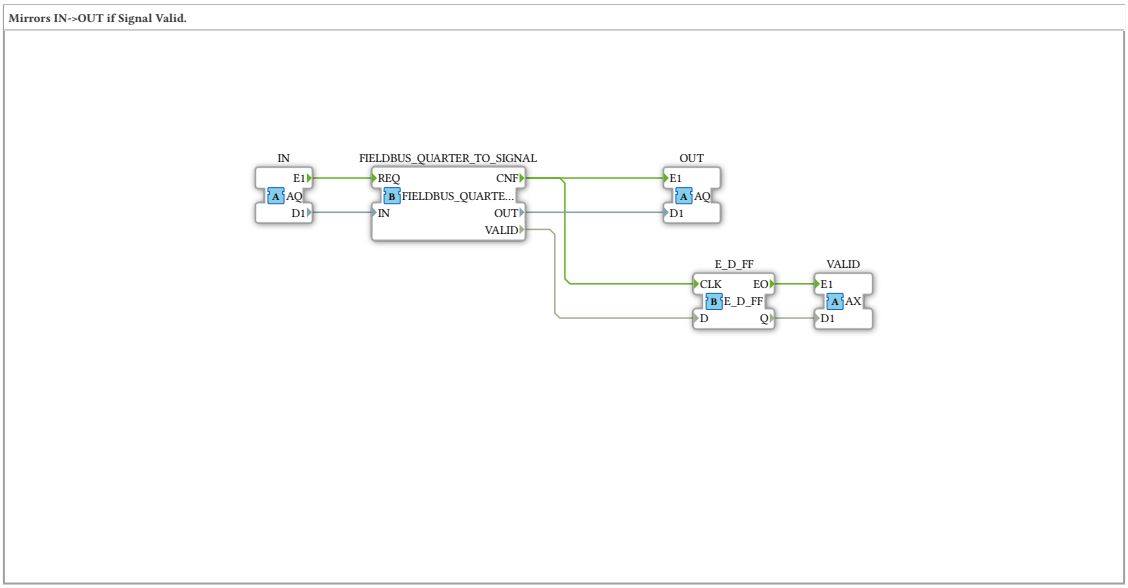
The combination of scaling, offset, and validity latch distinguishes it from simple conversion function blocks.

Conclusion

This functional block represents a robust solution for conditioning fieldbus signals. It combines scaling, offset shifting, and reliable validity indication in a compact, initializable component. The use of a D flip-flop stabilizes the signal quality, which is particularly advantageous in safety-critical or noisy environments.

AQ_FIELDBUS_QUARTER_TO_SIGNAL





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AQ_FIELDBUS_QUARTER_TO_SIGNAL** is used to mirror a signal (of type AQ) received via a fieldbus adapter to an output adapter – but only if the signal is marked as valid. It combines a fieldbus-specific processing block with an edge-triggered D flip-flop to buffer the validity status and provide it as a separate output signal.

Interface Structure

Event Inputs

The function block does not have direct event inputs at the top level. Instead, events are provided via the **socket adapter IN** (type AQ). The event output IN.E1 initiates the processing of the incoming signal.

Event Outputs

Event outputs are also exclusively handled via the **plug adapters OUT** and VALID:

- **OUT.E1**: Triggered after successful signal mirroring. (The result is available at the data output OUT.D1.)
- **VALID.E1**: Triggered as soon as the signal's validity status is updated (rising edge of the internal D flip-flop).

Data Inputs

Data is also received via the **socket adapter IN**:

- **IN.D1**: Receives the analog or digital input signal (adapter type AQ).

Data Outputs

- **OUT.D1**: Outputs the mirrored input signal – but only if it has been recognized as valid. (If the signal is invalid, a defined value or the last valid value can be output; this depends on the internal implementation of FIELDBUS_QUARTER_TO_SIGNAL.)
- **VALID.D1**: Outputs the current validity status (TRUE = valid, FALSE = invalid). This value is the output signal Q of the D flip-flop.

Adapter

Name	Type	Direction	Description
IN	adapter::types::unidirectional::AQ	**Socket** (Input)	Fieldbus signal (event and data)
OUT	adapter::types::unidirectional::AQ	**Plug** (output)	Filtered/mirrored output signal
VALID	adapter::types::unidirectional::AX	**Plug** (output)	Signal validity status

Note: Adapter types AQ and AX are unidirectional protocols that typically combine an event and a data field.

Functionality

The function block operates as follows:

1. An incoming event on IN.E1 triggers the internal function block **FIELDBUS_QUARTER_TO_SIGNAL** (called via REQ).
2. This internal function block processes the data value (IN.D1) and generates two outputs:

- **OUT** – the processed signal (e.g., mirrored, normalized, filtered).
- **VALID** – a Boolean value (TRUE if the signal is valid).

1. The processed signal is forwarded directly to the output adapter OUT (via OUT.D1) and simultaneously triggers the event OUT.E1.
2. The validity state (VALID) is fed to the D input of an edge-triggered D flip-flop (E_D_FF). The clock edge is generated by the same event (CNF) that also triggers OUT.E1. Thus, the validity state is updated in the flip-flop with each cycle.
3. The flip-flop output Q is output via the adapter VALID.D1. The corresponding event VALID.E1 is triggered by the flip-flop output EO.

The function block ensures that the output signal is always synchronized with the currently processed validity status.

Technical Features

- The function block is implemented as a **composition** of two internal function blocks: `FIELDBUS_QUARTER_TO_SIGNAL` and `E_D_FF`. This offers high reusability and a clear separation of processing logic and state storage.
- The use of a **D flip-flop** buffers the validity status between event cycles. This ensures that the status is retained even if the input signal fails temporarily.
- The adapter interfaces follow the **unidirectional** pattern typical of fieldbus protocols. An event carries both a data value and an execution edge.
- The internal processing (`FIELDBUS_QUARTER_TO_SIGNAL`) can be adapted depending on the application (e.g., scaling, formatting, or plausibility checks) without changing the external interface.

State Overview

The component does not have its own visible state machine. The internal state is primarily determined by the **D flip-flop**:

- **State 0**: Flip-flop output `Q = FALSE` ⟶ Signal is considered **invalid**.
- **State 1**: Flip-flop output `Q = TRUE` ⟶ Signal is considered **valid**.

The state change occurs on the clock edge during each processing cycle (event on `IN.E1`). The current state can be read via the adapter `VALID.D1`.

Application Scenarios

- **Fieldbus Connection in Agricultural Engineering**: Receiving an analog sensor signal (e.g., torque, pressure) from a fieldbus, which is only passed on if communication is valid.
- **Signal Quality Filtering**: Integration into a chain of processing modules where only values marked as valid are used in the control or visualization.
- **Redundancy Check**: Combination with a second signal path, where the validity status serves as the switching criterion.

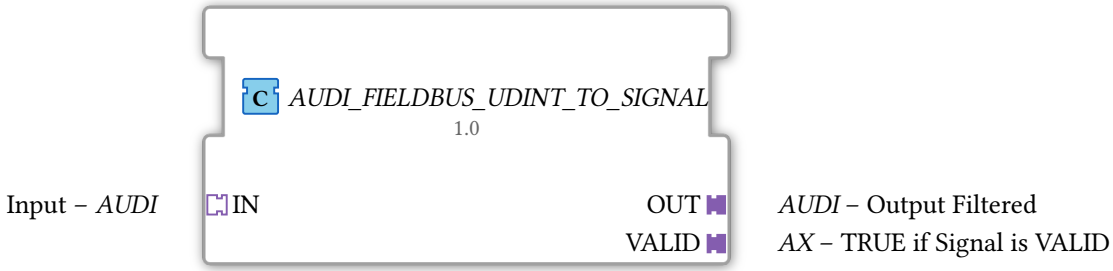
Comparison with Similar Modules

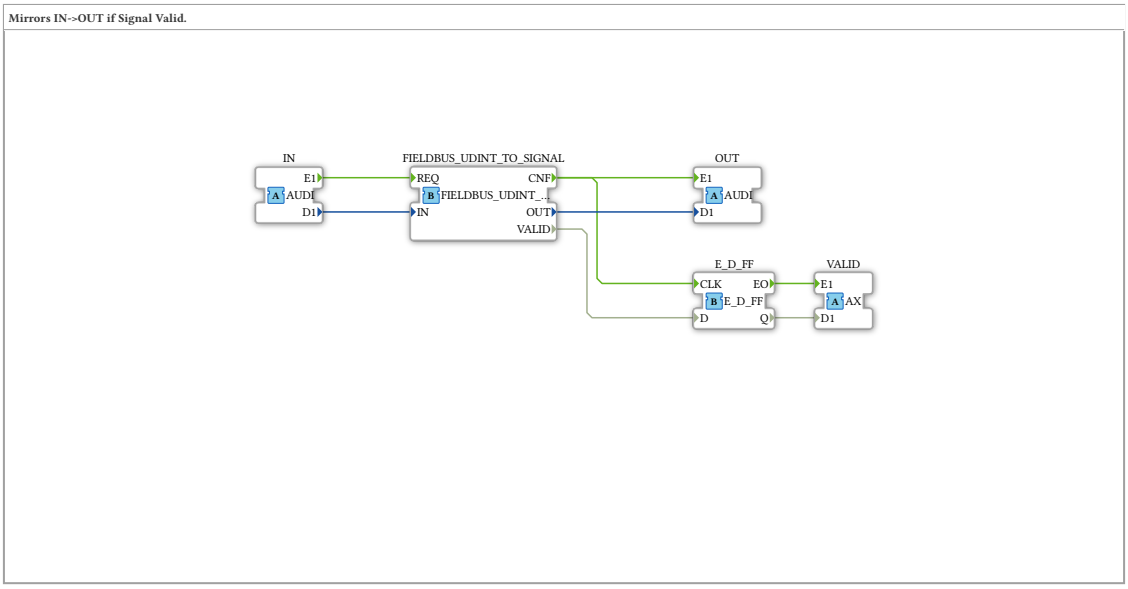
- **`AQ_FIELDBUS_SIGNAL_TO_QUARTER`** (hypothetical): Would represent the reverse process – conversion of a general signal into a fieldbus format.
- **`AQ_BUFFER`**: A simple buffer without validity checks; this module extends this functionality with validation and state storage.
- **`AQ_SELECT`**: Selects between two inputs; in this function block, the validity of a single signal is evaluated instead.

Conclusion

AQ_FIELDBUS_QUARTER_TO_SIGNAL is a compact, adapter-based function block for the reliable mirroring of fieldbus signals while taking signal validity into account. The combination of processing logic and edge-triggered state memory makes it particularly suitable for safety-critical or quality-controlled applications in industrial environments. The clear separation of the internal components facilitates maintenance and customization.

AUDI_FIELDBUS_UDINT_TO_SIGNAL





• – ○ • – ○ • – ○ •

Introduction

The function block `AUDI_FIELDBUS_UDINT_TO_SIGNAL` mirrors an incoming `UDINT` value (via the `IN` adapter) to the `OUT` adapter, provided the incoming signal is recognized as valid. Additionally, the validity signal is output via the `VALID` adapter. This function block acts as a filter, allowing only valid data packets to pass through and providing their status synchronously.

Interface Structure

This function block has no direct input/output events or data, but operates exclusively via adapters that transport both events and data.

Event Inputs

- **`IN.E1`** (via the `IN` adapter) – Starts processing of the incoming signal.

Event Outputs

- **`OUT.E1`** (via the `OUT` adapter) – Triggered after the `IN` value has been processed and passed to `OUT.D1`.
- **`VALID.E1`** (via the `VALID` adapter) – Triggered as soon as the signal's validity has been updated (see Functionality).

Data Inputs

- **`IN.D1`** (via the `IN` adapter) – The incoming `UDINT` value (e.g., from a fieldbus).

Data Outputs

- **`OUT.D1`** (via the `OUT` adapter) – The filtered `UDINT` value (identical to `IN.D1`, if valid).
- **`VALID.D1`** (via the `VALID` adapter) – Boolean value that signals `TRUE` if the incoming signal is valid.

Adapter

Adapter	Type	Direction	Description
`IN`	`adapter::types::unidirectional::AUDI`	Socket	Signal input (reads data and events)
`OUT`	`adapter::types::unidirectional::AUDI`	Plug	Filtered signal output
`VALID`	`adapter::types::unidirectional::AX`	Plug	Validation Indicator (Output for Boolean & Event)

Functionality

The function block contains two internal function blocks:

1. **FIELDBUS_UDINT_TO_SIGNAL** – Converts the incoming UDINT value into a signal and determines its validity.
2. **E_D_FF** – An edge-triggered D flip-flop that synchronizes the validity signal.

Process:

- An event on IN.E1 triggers the processing block FIELDBUS_UDINT_TO_SIGNAL (input REQ).
- This block reads IN.D1 and outputs an event to CNF upon completion.
- The event CNF is forwarded to three locations:
- To OUT.E1 ⟶ the output value (FIELDBUS_UDINT_TO_SIGNAL.OUT) is assigned to OUT.D1.
- To the clock input CLK of the flip-flop (E_D_FF).
- Simultaneously, the validity status (FIELDBUS_UDINT_TO_SIGNAL.VALID) is set to the data input D of the flip-flop.
- On the rising edge of CLK, the flip-flop inherits the value from D to Q and outputs an event on EO.
- The flip-flop output Q feeds VALID.D1, and the event EO triggers VALID.E1.

Thus, the validity signal is only updated and output after the signal processing is complete.

Technical Features

- **Validity Synchronization:** The D flip-flop ensures that the validity status is only available in the next clock cycle after data processing – this prevents inconsistent states.
- **Adapter-based interface:** All data and events are exchanged via standardized unidirectional adapters, which facilitates reuse in hierarchical projects.
- **License notice:** This function block is subject to the **Eclipse Public License 2.0** and contains a copyright notice (HR Agrartechnik GmbH).

State overview

The function block does not have its own state machine; its behavior is determined by the internal D flip-flop. This flip-flop has two states:

State	Q (Output)	Meaning
0	FALSE	Signal currently invalid
1	TRUE	Signal valid

The state changes on each rising clock edge (CLK) to the current value of D. A reset is not provided – if the signal is invalid, Q remains at the last known valid value until a new clock signal (D = FALSE) arrives.

Application Scenarios

- **Fieldbus Signal Conditioning:** A sensor delivers a UDINT value along with a validity flag via a fieldbus (e.g., CANopen, PROFIBUS). This function block filters out invalid values and provides only valid data to the controller.
- **Data Validation in Safety-Critical Environments:** If a higher-level application should only receive processed, valid measured values, this function block can be placed between the bus and the logic.
- **Synchronization of multiple parallel signals:** The separate validity signal can be used to clock downstream function blocks or trigger alarms.

Comparison with similar function blocks

Function block	Feature
FIELDBUS_UDINT_TO_SIGNAL alone	Passes the invalid signal on immediately – without validity synchronization.
AUDI_FIELDBUS_UDINT_TO_SIGNAL (this function block)	**Additional synchronization** of the validity signal via a D flip-flop, so that VALID is only updated with the next clock cycle.
Other validation function blocks	Often without dynamic synchronization; this function block is particularly suitable for cyclic

	bus systems where data and validity may arrive with a time delay.

Conclusion

AUDI_FIELDBUS_UDINT_TO_SIGNALis a specialized filter module for fieldbus signals that only forwards valid incomingUDINT` values to the output and outputs the status via a synchronized path. The use of an internal flip-flop prevents inconsistent states and makes it ideal for use in time-critical automation environments. Its adapter-based interface allows for easy integration into larger 4diac projects.

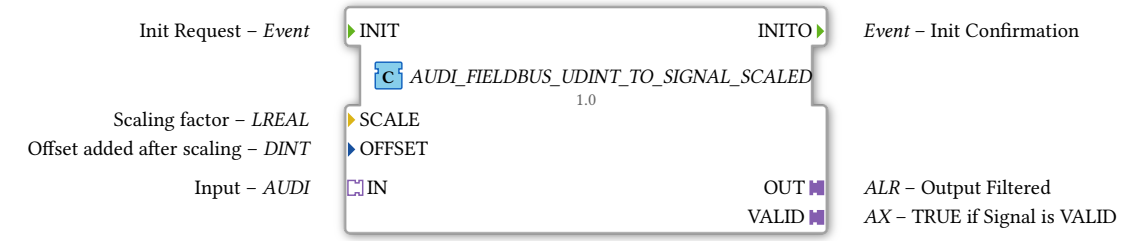
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AUDI_FIELDBUS_UDINT_TO_SIGNAL_SCALED



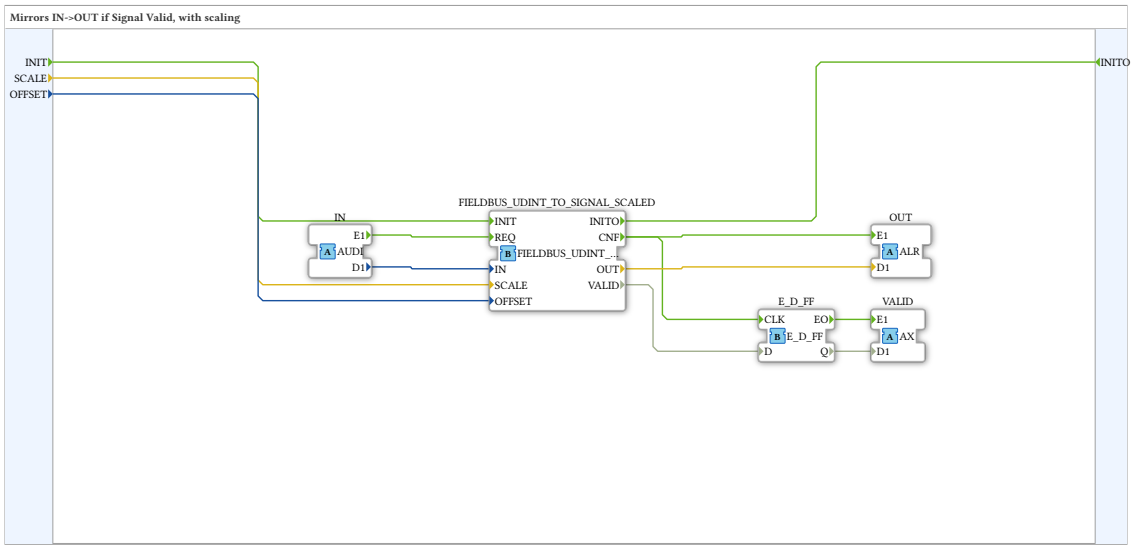


Image of the function block: not available

• – ○ • – ○ • – ○ •

Introduction

The function block **AUDI_FIELDBUS_UDINT_TO_SIGNAL_SCALED** is used to convert a received UDINT value (unsigned 32-bit integer) from a fieldbus protocol into a scaled signal and only forward it to the output if the input signal is marked as valid. It represents a combined solution for scaling and signal validity checking and is primarily used in automation technology for LogiBUS applications.

Interface Structure

Event Inputs

Name	Type	Comment

INIT	EInit	Initialization Request

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Acknowledgement

Data Inputs

Name	Type	Comment	Initial Value
SCALE	LREAL	Scaling Factor	LREAL#1.0
OFFSET	DINT	Offset Added After Scaling	DINT#0

Data Outputs

No separate data outputs – the scaled data is provided via the adapter output `OUT`.

Adapter

Name	Type	Direction	Comment
IN	unidirectional::AUDI	Input	Fieldbus UDINT input signal
OUT	unidirectional::ALR	Output	Scaled signal (if valid)
VALID	unidirectional::AX	Output	Validation flag (TRUE = valid)

Functionality

The function block operates internally as a composite block. It consists of a sub-function block `FIELDBUS_UDINT_TO_SIGNAL_SCALED` (which performs the actual scaling) and an edge-triggered D flip-flop `E_D_FF`.

- A UDINT value (via `IN.D1`) and an event (`IN.E1`) are received via the adapter `IN`.
- The incoming value is forwarded to the internal function block `FIELDBUS_UDINT_TO_SIGNAL_SCALED`, which multiplies the value by `SCALE` and then adds `OFFSET`. The result is available at the output `OUT.D1`.
- Simultaneously, the internal function block generates a validity signal (`VALID`) indicating whether the received raw value is considered valid.
- This validity signal is fed to the D flip-flop at its data input (`D`). The rising edge of the processing event (`CNF`) clocks the flip-flop (`CLK`).
- The instantaneous, synchronized validity signal is present at the flip-flop's output (`Q`) and is output via the adapter (`VALID.D1`).
- The event (`OUT.E1`) is triggered simultaneously with the update of the output data, allowing downstream function blocks to retrieve the new values.

Technical Features

- The function block (FB) is implemented as a **composite FB**, meaning its functionality is represented by an internal network of other function blocks. This allows for a clear separation of scaling and validity logic.
- Initialization occurs via the `INIT` event, which is forwarded to the internal FB. After successful initialization, `INITO` is output.
- The scaling factors `SCALE` and `OFFSET` are set during initialization and cannot be changed at runtime, as they are not updated via separate events.
- The use of a D flip-flop ensures that the validity state remains stable until the next valid data value is processed.

- This function block is designed for use in the LogiBUS environment (package `logiBUS::signalprocessing::fieldbus`).

State Overview

This function block does not have explicit states in the sense of a state machine. Its internal logic is executed purely data-flow controlled via the event chain:

1. **Initialization Phase**: After receiving `INIT`, the internal function block is initialized and `INITO` is sent.
2. **Data Processing Phase**: An incoming `IN.E1` triggers the scaling and validity check. Upon completion, `OUT.E1` and the output data are updated.

Application Scenarios

- **Scaling of Fieldbus Signals**: When a fieldbus delivers a UDINT value (e.g., a sensor counter reading) that needs to be converted into a physical unit, e.g., scaling and offset.
- **Validity Check**: Only valid measured values should be passed on to the controller. The flag `VALID` can be evaluated by a higher-level safety block, for example.
- **Coupling of LogiBUS and IEC 61499 Systems**: The function block serves as a bridge between the LogiBUS protocol and standardized function blocks.

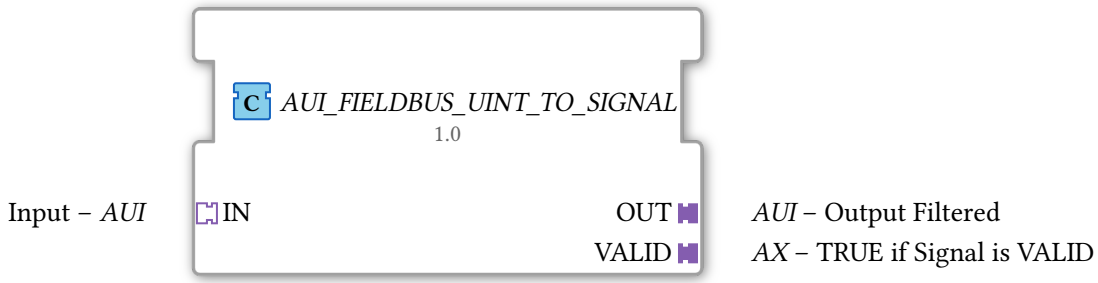
Comparison with Similar Function Blocks

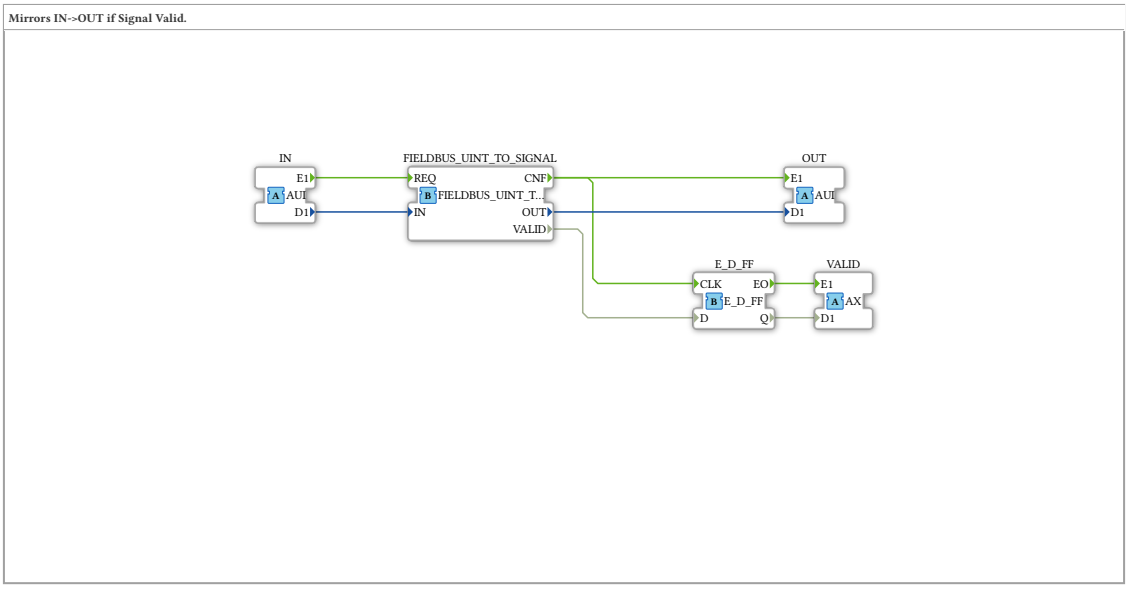
- **FIELDBUS_UDINT_TO_SIGNAL_SCALED** (bare scaling block): This block performs only the scaling, without the validity check and without the memory logic. The present composite function block extends it to include the status management of the validity flag.
- **Standard IEC 61499 Scaling Blocks** (e.g., `SCALE`): Usually implemented as simple arithmetic blocks, they often lack integrated validity logic and connectivity to fieldbus adapters.

Conclusion

The function block `AUDI_FIELDBUS_UDINT_TO_SIGNAL_SCALED` offers a compact, reusable solution for scalable signal processing with integrated validity checking in LogiBUS environments. It combines scaling, offset, and an edge-triggered validity lock, ensuring that only valid and correctly scaled values are present at the output. Thanks to its composite structure, it is easily extensible and readily integrates into existing IEC 61499 projects.

AUI_FIELDBUS_UINT_TO_SIGNAL





• — ○ • — ○ • — ○ •

Introduction

The function block **AUI_FIELDBUS_UINT_TO_SIGNAL** forwards a fieldbus signal encoded as `UINT` to a downstream AUI adapter, provided the signal is recognized as valid. It also provides a separate validity indicator (`VALID`). The block encapsulates a data converter and an edge-triggered D flip-flop, which buffers the validity signal until the next event.

Interface Structure

Event Inputs

The function block does not have separate event inputs. All event-driven processes are handled via the **adapter socket `IN`**.

Event Outputs

The function block does not have separate event outputs. Events are output via the **adapter plugs `OUT` and `VALID`**.

Data Inputs

The function block does not have separate data inputs. The input value is provided as a data signal `D1` via the **adapter socket `IN`**.

Data Outputs

The function block does not have separate data outputs. Output data is transmitted via the **adapter plugs `OUT` and `VALID`** as data signals `D1`.

Adapter

Name	Type	Direction	Description
`IN`	AUI (unidirectional)	Socket	Receives the raw fieldbus signal as `UINT`. Provides event `E1` and data `D1`.
`OUT`	AUI (unidirectional)	Plug	Outputs the filtered signal as `UINT`. Event `E1` signals incoming data.
`VALID`	AX (unidirectional)	Plug	Provides a validity signal (`TRUE`/`FALSE`) via `D1`; event `E1` indicates an update.

Functionality

1. An external event on the **socket `IN`** (via its event input `E1`) triggers the process.
2. The incoming data value (`IN.D1`) is processed by the internal function block `FIELDBUS_UINT_TO_SIGNAL`. This generates an output value (`OUT`) and a validity flag (`VALID`).
3. The processed value is immediately passed to the **plug `OUT`** (data signal `D1`) and an event is triggered at `OUT.E1`.
4. Simultaneously, the validity flag of the internal function block is transferred to the **D flip-flop `E_D_FF`** (clocked by the same event).
5. The output of the flip-flop (`Q`) is connected to the **plug `VALID`** (data signal `D1`); simultaneously, an event is sent to `VALID.E1`.
6. The state of the validity signal is retained until the next processing cycle.

Technical Features

- The function block is implemented as a **composite FB** its functionality consists of two internal FBs:
- `FIELDBUS_UINT_TO_SIGNAL` (data converter)
- `E_D_FF` (edge-triggered D flip-flop)
- The validity indicator is **event-triggered** and is buffered via a flip-flop. This ensures stability even if the input signal is absent for several cycles.
- The signal and validity outputs are **quasi-parallel** (both via the same event of the internal FB).
- The interfaces are defined exclusively as **adapters**, which facilitates modular integration into fieldbus systems.

State Overview

The FB itself does not have an explicit state machine. However, its behavior can be described by its internal logic:

State	Description
Idle	No input event; the outputs `OUT` and `VALID` retain their last values.
Processing	An event at `IN.E1` starts processing.
Output	Upon completion, `OUT.D1` and `VALID.D1` are updated, and the events are sent to `OUT.E1` and `VALID.E1`.
Hold	The validation value is held in the flip-flop until the next event arrives.

Application Scenarios

- **Fieldbus Interface**: A `UINT` value originating from a fieldbus is to be converted into a standardized AUI signal and only passed on if data integrity is maintained.
- **Validation-Checked Forwarding**: Applications where the output signal is only considered valid after successful internal validation (e.g., CRC check).
- **Single-Channel Signal Conditioning**: This function block can be used in safety-related chains to separately signal the result of a plausibility check.

Comparison with Similar Function Blocks

Function Block	Difference / Similarity
`FIELDBUS_UINT_TO_SIGNAL`	Contains only the pure data conversion without a validity buffer.
`AUI_SIGNAL_FILTER`	Filters signals but does not offer an explicit validity indicator.
`E_D_FF`	Pure flip-flop without data conversion – used here as an auxiliary function block.

This function block combines the conversion with **event-driven validity control**, making it particularly suitable for sequential fieldbus protocols.

Conclusion

The `AUI_FIELDBUS_UINT_TO_SIGNAL` function block is a compact, adapter-based module for tested signal forwarding in fieldbus systems. Through the internal coupling of data conversion and edge-triggered

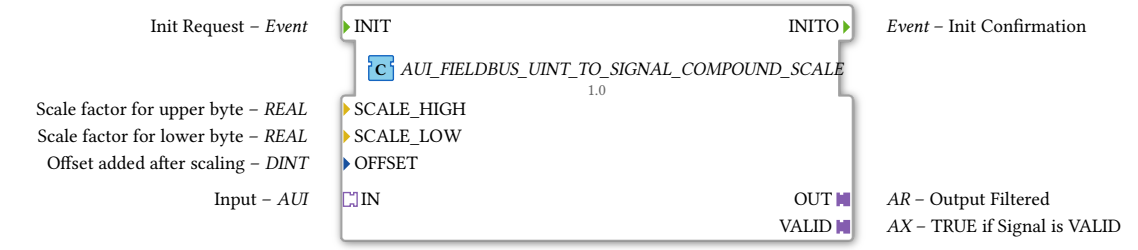
validity indication, it offers a robust and traceable interface for industrial automation. The use of adapters enables easy integration into existing 4diac networks.

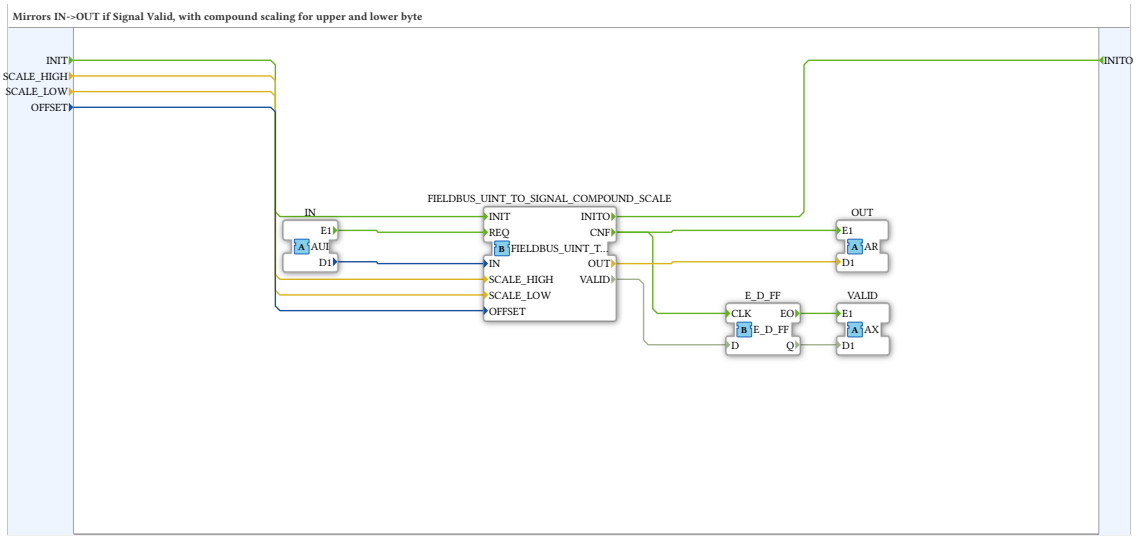
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE





• – ○ • – ○ • – ○ •

Introduction

The function block `AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` is used to convert and scale a 16-bit value received via an AUI fieldbus adapter into a scalable signal. The block mirrors the input value to the output, provided the validity signal is active, and performs compound scaling on the upper and lower bytes of the input value. An offset can be added after scaling. The output is provided via a unidirectional AR adapter, while the validity is signaled via an AX adapter. The block is specifically designed for industrial fieldbus protocols where user data consists of two separately scalable byte ranges.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initialization request; sets the function block to a defined start state.

Event Outputs

Event	Type	Description
INITO	EInit	Initialization confirmation; sent after successful initialization.

Data Inputs

Variable	Type	Initial Value	Description
SCALE_HIGH	REAL	0.256	Scaling factor for the upper byte of the incoming value.
SCALE_LOW	REAL	0.001	Scaling factor for the lower byte of the incoming value.
OFFSET	DINT	0	Integer offset added after scaling.

Data Outputs

This function block has no separate data outputs other than `VarDeclaration`. The output data is provided via the adapters `OUT` and `VALID`.

Adapters

Adapter	Direction	Type	Description
IN	Socket	adapter::types::unidirectional::AUI	Input for the fieldbus data value to be processed.
OUT	Plug	adapter::types::unidirectional::AR	Output of the scaled value (e.g., REAL or DINT, depending on the internal function block).
VALID	Plug	adapter::types::unidirectional::AX	Output signal: TRUE if the current output value is valid.

Functionality

The function block operates as a composite function block that uses an internal function block `FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` in combination with an edge-triggered D flip-flop (`E_D_FF`). The processing proceeds as follows:

1. **Input Event**: An event at the event input `IN.E1` of the AUI adapter triggers the processing of the current data value `IN.D1`.
2. **Scaling**: The incoming 16-bit value is split into an upper and a lower byte. Each byte is multiplied by its corresponding scaling factor (`SCALE_HIGH`, `SCALE_LOW`). The result of both scaling operations is summed and added to `OFFSET`. The calculated value is then passed to `OUT.D1` via the data output of the internal module.
3. **Validity Signal**: In parallel, the internal component generates a logic signal `VALID`, indicating whether the calculated value is valid. This signal is clocked by the D flip-flop with the completion event `CNF` and forwarded to the `VALID.D1` adapter. This ensures that the validity signal is only activated if the value has actually been recalculated.
4. **Output Event**: The internal block's `CNF` event is used to control both the output adapter `OUT` (event `E1`) and the `VALID` adapter (event `E1`).
5. **Initialization**: The `INIT` event is forwarded directly to the internal block; the initialization confirmation `INITO` is returned accordingly.

Technical Features

- **Compound Scaling**: Unlike simple scaling blocks, this function block allows separate weighting of the two byte halves of a 16-bit value. This is useful when a sensor value is composed of two different physical quantities (e.g., an integer and a decimal place).
- **Integrated Validation Logic**: The D flip-flop makes the `VALID` signal edge-triggered and only activates it when new calculations are actually performed. This prevents unwanted output when no update is available.
- **Initialization Pass-Through**: The INIT/INITO events are passed transparently through the composite block, ensuring that the internal scaling block is correctly initialized.

State Overview

The block itself does not have an explicit state machine at the composite level. Operational readiness is established by the `INIT` event. After successful initialization, the function block waits for incoming events at the `IN` adapter and processes them cyclically. As long as no INIT event has been sent, the function block is in an undefined state (no output, `VALID` inactive).

Application Scenarios

- **Fieldbus Interfaces**: Conversion of analog sensor values transmitted as a 16-bit word with separate scaling factors for the integer and decimal parts (e.g., temperature/pressure values with decimal places).
- **Measurement Processing**: Scaling of digital input signals from fieldbus controllers that deliver raw data in two-byte ranges.
- **Signal Validation**: Used in safety-related applications where only valid and correctly calculated values may be passed on to the subsequent controller.

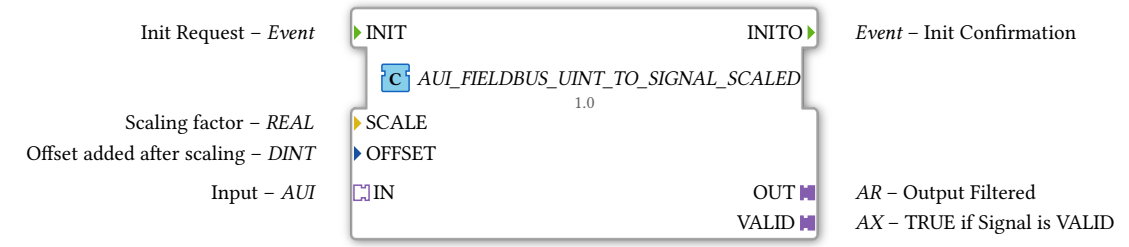
Comparison with Similar Function Blocks

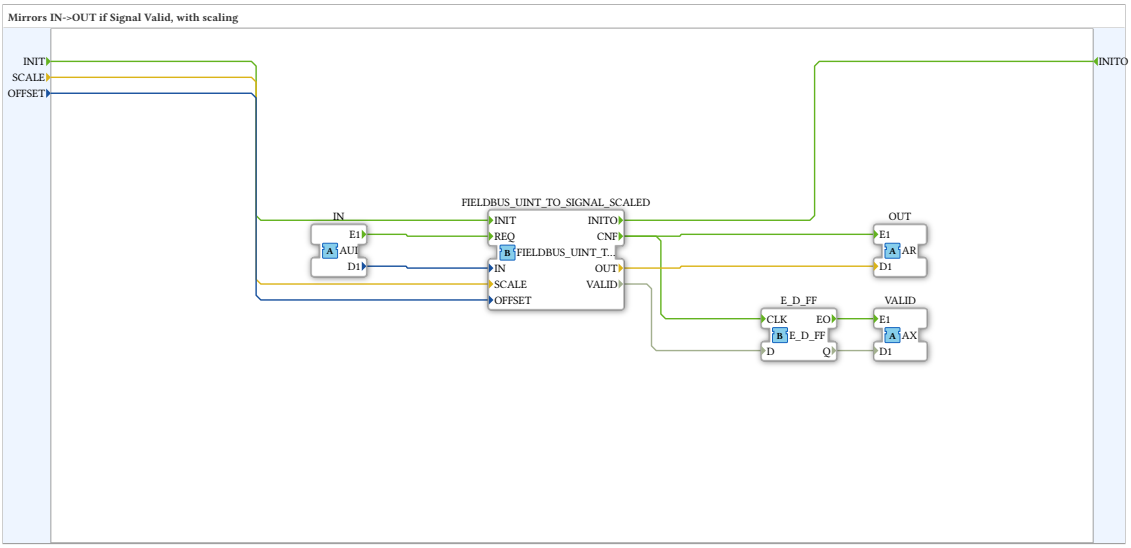
Unlike simple scaling blocks (e.g., `SCALE` or `UINT_TO_REAL`), this function block offers differentiated processing of the two byte halves. It is specifically designed for data formats where the upper and lower bytes represent different physical units or scaling ranges. Alternative function blocks would have to implement this logic manually by splitting, scaling separately, and then adding the values, which increases the implementation effort.

Conclusion

The `AUI_FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` is a specialized yet flexible function block for processing fieldbus signals. Its ability to process an incoming 16-bit word with two different scaling factors and to control the validity of the output signal makes it a valuable component in IEC-61499-based automation systems, especially when it comes to integrating devices with compounded data formats.

AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Introduction

The AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED function block is used for the scalable transmission of an incoming UINT value (fieldbus signal) to an analog REAL output. It only operates if the incoming signal is marked as valid (VALID signal). An additional initialization interface allows the internal logic to be reset. The block is implemented as a composite and combines a scalable sub-block with a D flip-flop for validity storage.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initialization Request

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Acknowledgement

Data Inputs

Name	Type	Initial Value	Comment
SCALE	REAL	REAL#1.0	Scaling Factor
OFFSET	DINT	DINT#0	Offset after Scaling

Data Outputs

(No direct data outputs – outputs are via adapters)

Adapters

Type	Name	Direction	Comment
adapter::types::unidirectional::AUI	IN	Socket	UINT input (fieldbus signal)
adapter::types::unidirectional::AR	OUT	Plug	Scaled REAL output
adapter::types::unidirectional::AX	VALID	Plug	Validation signal (TRUE = valid)

Functionality

The function block consists internally of two sub-function blocks: the scalable core function block `logiBUS::signalprocessing::fieldbus::FIELDBUS_UINT_TO_SIGNAL_SCALED` and a D flip-flop `E_D_FF`.

Initialization is performed via the INIT event input, which is forwarded directly to the internal core function block. The acknowledgment INITO is returned after successful initialization.

The actual signal processing is triggered by an incoming event (E1) from the IN adapter. This event triggers the core module, which processes the UINT value received via IN.D1 as follows:

- **Scaling**: `Wert_skaliert = (REAL)IN_D1 * SCALE`
- **Offset**: `Wert_skaliert = Wert_skaliert + OFFSET`

The result (REAL) is output as OUT.D1, and an event OUT.E1 is simultaneously generated. In parallel, the VALID signal (TRUE = valid, FALSE = invalid) calculated by the core module is applied to the D input of the D flip-flop. The same CNF event from the core module serves as the clock signal for the flip-flop. This ensures that the validity state is maintained at the Q output until the next clock cycle. The final validity value is output via the VALID adapter (D1) and its event E1.

This means the VALID signal is only updated after scaling is complete and remains stable until a new value arrives at the input.

Technical Features

- **Composite Architecture**: The function block (FB) is built entirely from sub-FBs, enabling easy adaptation and reuse.
- **Validity Storage**: A D flip-flop stores the validity signal between processing cycles. This allows the output to display a defined state even when no new data is present.
- **Scaling and Offset**: Scaling is performed using a REAL factor and a DINT offset. This allows for flexible adaptation to physical units (e.g., conversion of digital values to pressure or temperature).
- **Adapter-Based Input/Output**: All signals are exchanged via standardized 61499 adapters (AUI, AR, AX), which facilitates integration into modular fieldbus systems.

State Overview

The function block itself does not have its own state machine. The internal state is represented by the D flip-flop:

- **State Q = FALSE**: The last processed value was invalid or no value has yet been received.
- **State Q = TRUE**: The last processed value was valid and is scaled and available at the OUT output.

The state change occurs with each new event on the IN adapter (E1) when the core module provides a new VALID signal.

Application Scenarios

- **Agricultural Sensors**: Transmission of scaled measured values (e.g., fill level, pressure) from fieldbus sensors with a validity flag.
- **Data Preprocessing**: Conversion of raw values from digital converters (e.g., 0–10 V ⟶ REAL values) with automatic validity checks.
- **Quality Assurance**: Only values marked as valid are passed on to subsequent control logic.

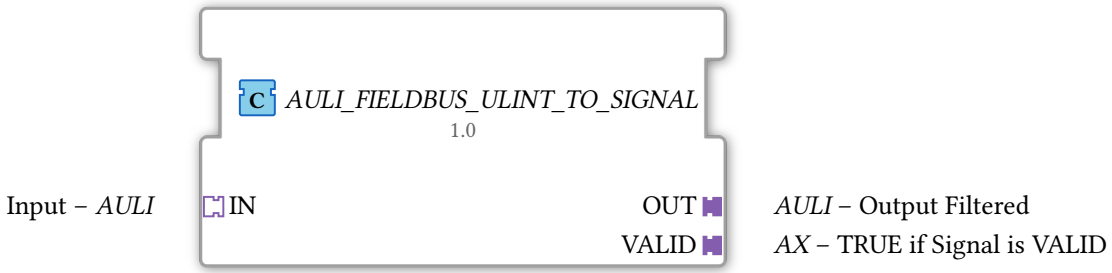
Comparison with Similar Function Blocks

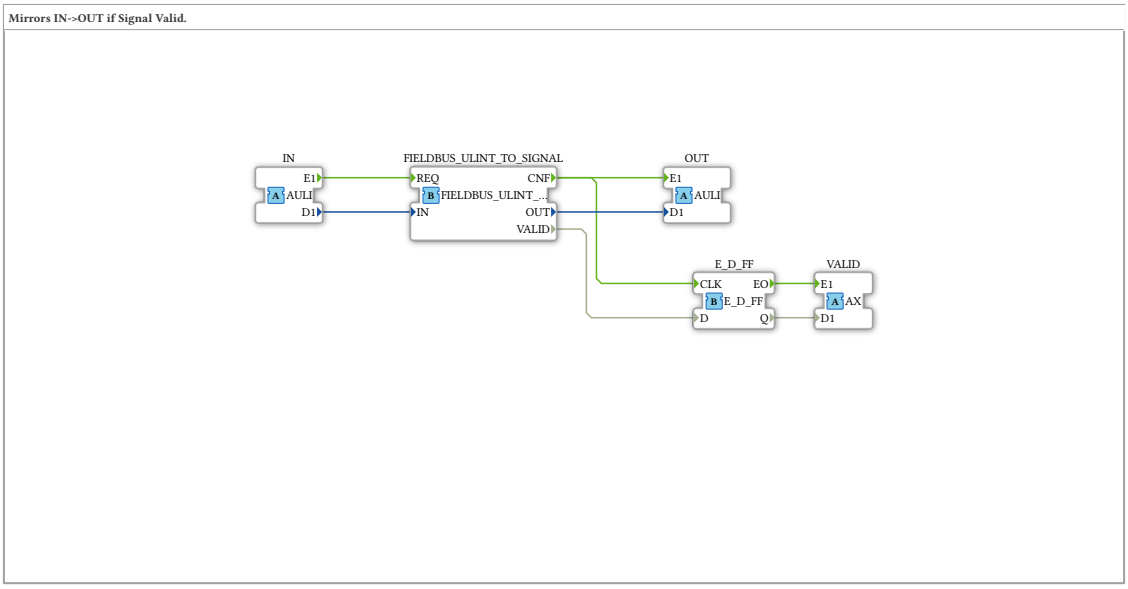
- **AUI_FIELDBUS_UINT_TO_SIGNAL** (without validity flag): Always outputs a scaled value, regardless of signal quality.
- **AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED** (Core): Already provides a scaled output with a validity flag, but without storage. This composite adds this storage capability and allows for more stable output between events.
- **Adapter-Based Filters**: Other function blocks use similar principles, but with specific validity conditions (e.g., time filters).

Conclusion

The AUI_FIELDBUS_UINT_TO_SIGNAL_SCALED offers a reliable and flexible solution for the scaled transmission of UINT fieldbus signals with explicit validity marking. The combination of scaling, offset, and stored validity signal makes it particularly suitable for safety-critical or quality-controlled applications in automation technology.

AULI_FIELDBUS_ULINT_TO_SIGNAL





• — ○ • — ○ • — ○ •

Introduction

The function block **AULI_FIELDBUS_ULINT_TO_SIGNAL** serves as a composite module for processing a ULINT value from a fieldbus. It mirrors the incoming value to the output if the signal is recognized as valid and also outputs a synchronized validity signal. The module combines an internal fieldbus converter with a D flip-flop to ensure clocked, reliable signal transmission.

Interface Structure

The module does not have traditional top-level event or data ports, but communicates exclusively via adapters. These are implemented as sockets (inputs) and plugs (outputs).

Event Inputs

Port	Name	Description
`IN.E1`	Event Input (via Socket IN)	Triggers the processing of a new ULINT value.

Event Outputs

Port	Label	Description
`OUT.E1`	Event Output (via Plug OUT)	Signals completion of the output of the mirrored value.
`VALID.E1`	Event Output (via Plug VALID)	Indicates that the validity signal has been updated.

Data Inputs

Port	Label	Data Type	Description
`IN.D1`	Data Input (via Socket IN)	ULINT	The fieldbus value to be processed.

Data Outputs

Port	Label	Data Type	Description
`OUT.D1`	Data Output (via Plug OUT)	ULINT	The mirrored input value (if valid).
`VALID.D1`	Data Output (via Plug VALID)	BOOL	Validation signal (TRUE = value is valid).

Adapter

Adapter	Direction	Type	Description
IN	Socket (Input)	adapter::types::unidirectional::AULI	Receives the ULINT value and the associated event.
OUT	Plug (Output)	adapter::types::unidirectional::AULI	Outputs the processed ULINT value and an event.
VALID	Plug (Output)	adapter::types::unidirectional::AX	Provides the validity signal (BOOL) and an event.

Functionality

The function block operates internally with two sub-functions:

1. **FIELDBUS_ULINT_TO_SIGNAL** – A fieldbus converter that converts a ULINT value into a signal and simultaneously generates a validity signal (BOOL).
2. **E_D_FF** – An edge-triggered D flip-flop that receives the validity signal synchronously with the converter's event.

Process:

1. An event at input `IN.E1` starts the processing.
2. The input value `IN.D1` is passed to the internal converter.
3. The converter calculates the output value `OUT` and the `VALID` signal.
4. The result `OUT` is immediately mirrored to the output adapter `OUT.E1` / `OUT.D1`.
5. Simultaneously, the `VALID` signal is passed to the flip-flop and clocked by the event `CNF`.
6. The flip-flop output `Q` is output to the adapter `VALID.D1`, generating an event `VALID.E1`.

`` ``VALID`` ``Q `VALID.D1` VALID.E1` ... This ensures that the output value and the validity signal are always updated consistently and synchronously.

Technical Features

- **Adapter-Based Interface**: The function block uses standardized unidirectional adapters (`AULI`, `AX`), enabling loose coupling and reusability.
- **Synchronization via Flip-Flop**: The validity signal is edge-triggered to avoid clock and glitch issues.
- **Composite Structure**: The internal logic is implemented as a network of two standard function blocks, increasing maintainability and transparency.
- **No Direct Event/Data Ports**: All communication occurs via adapters, facilitating integration into adapter-based architectures.

State Overview

The function block itself is stateless (purely combinatorial with clocked takeover). The internal flip-flop `E_D_FF` has two states:

- **Q** = **FALSE**: No valid signal (initial or after an invalid value).
- **Q** = **TRUE**: Valid signal is present.

A state diagram for the entire function block is not required, as the behavior is fully described by the event and data flows.

Application Scenarios

- **Fieldbus Signal Conditioning**: A ULINT value (e.g., counter reading, position value) is received from a fieldbus, checked for validity, and passed on to a controller or visualization system.
- **Quality Indication**: A Boolean signal indicating validity (e.g., "Data valid" / "Data faulty") is provided along with the data value.
- **Safety-Oriented Transmission**: By synchronously receiving the validity signal, errors in the data line can be detected and signaled.

Comparison with Similar Function Blocks

- **Simple Pass-Through Function Block**: A simple `MOVE` or `REPEAT` function block forwards the value without validation. `AULI_FIELDBUS_ULINT_TO_SIGNAL` adds explicit validity checking and signaling.

- **FIELDBUS_ULINT_TO_SIGNAL (Single)**: This core function block already provides the validity signal, but without edge-triggered synchronization. This composite function block enhances signal quality with a flip-flop.
- **Adapter-Based Function Blocks**: Specifically designed for use in 4diac environments that rely on adapter communication; other function blocks with direct ports are less flexible.

Conclusion

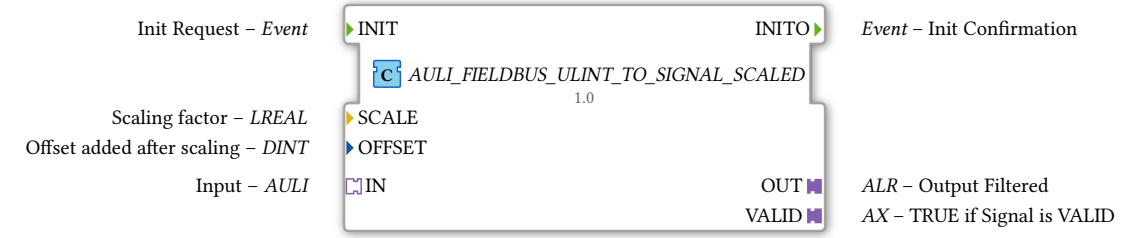
AULI_FIELDBUS_ULINT_TO_SIGNAL` is a robust, adapter-based function block for the reliable transmission of a ULINT fieldbus signal with validity indication. The combination of converter and flip-flop ensures synchronized and reliable signal output. Its clear interface structure makes it particularly suitable for modular automation solutions that rely on loosely coupled adapters.

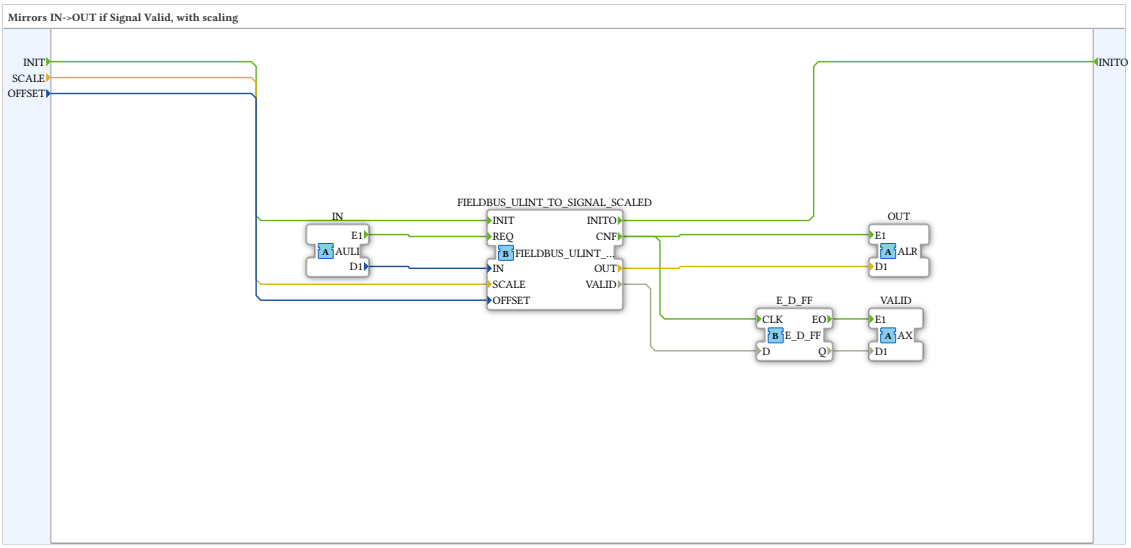
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AULI_FIELDBUS_ULINT_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Introduction

The function block **AULI_FIELDBUS_ULINT_TO_SIGNAL_SCALED** is used for the scalable mirroring of an unsigned integer value (ULINT) into a scaled signal value (LREAL), taking validity information into account. It is implemented as a composite block and combines the actual scaling logic with a synchronized valid output. The block is designed for use in fieldbus environments where raw data (e.g., sensor values) must be converted using a linear factor and offset, and the signal's validity must be reliably transmitted.

Interface Structure

Event Inputs

| Name | Type | Comment | |—–|—–|———–| | INIT | EInit | Init Request |

Event Outputs

Name	Type	Comment
INITO	EInit	Init Confirmation

Data Inputs

Name	Type	Initial Value	Comment
SCALE	LREAL	1.0	Scaling Factor
OFFSET	DINT	0	Offset added after scaling

Data Outputs

No explicit data outputs are available on the FB interface – output is provided via the adapters.

Adapter

Socket (Input)

Name	Type	Comment
IN	adapter::types::unidirectional::AULI	Input

Plug (Output)

Name	Type	Comment
OUT	adapter::types::unidirectional::ALR	Filtered Output
VALID	adapter::types::unidirectional::AX	TRUE if Signal is VALID

Functionality

The function block operates in an event-driven manner. After an INIT event (at input `INIT`), the internal scaling block is initialized and acknowledged with `INITO`. The actual processing occurs with each event that arrives via the adapter `IN.E1`:

1. **Data Transfer:** The ULINT value provided by `IN.D1` is forwarded to the internal function block `FIELDBUS_ULINT_TO_SIGNAL_SCALED`.
2. **Scaling:** The internal block applies the scaling factor `SCALE` (LREAL) and the offset `OFFSET` (DINT) to the raw value, generating a scaled LREAL value and a validity signal (`VALID`).
3. **Synchronization:** The validity signal is synchronized via an edge-triggered D flip-flop (`E_D_FF`). The clock edge is provided by the internal block's `CNF` event.
4. **Output:** The scaled value is sent to adapter `OUT` via `OUT.D1` and the associated event `OUT.E1`. The synchronized validity signal is output to adapter `VALID` via `VALID.D1` and the event `VALID.E1`.

The internal scaling block is of the same type as the outer one, meaning that the scaling logic could be recursive. However, in this design, the inner instance contains the actual computational logic. The outer shell adds the synchronization of the validity information and provides the adapter interfaces.

The internal scaling block is of the same type as the outer one, meaning that the scaling logic could be recursive.

Technical Features

- **Adapted Inputs/Outputs:** The module uses adapters exclusively for data transmission. This enables loose coupling in fieldbus and component networks.
- **Validity Synchronization:** The validation information is clocked via a D flip-flop so that it coincides with the output signal value.
- **Scaling Parameters:** `SCALE` and `OFFSET` are directly accessible as input variables and can be changed at runtime. Initial values allow immediate operation without configuration.

State Overview

The module does not have an explicit state machine. Its behavior is determined by event control:

- **Initialized:** After successful `INIT`, the internal scaling block is ready.
- **Ready:** Data processing is triggered upon an event on `IN.E1`. Upon completion, `OUT.E1` and `VALID.E1` are triggered.
- **Error Case:** An erroneous scaling operation (e.g., overflow) is handled by the internal block; the validation signal would then be `FALSE`.

Application Scenarios

- **Fieldbus Sensor Evaluation:** A sensor delivers a raw ULINT value via a bus (e.g., CANopen, PROFIBUS). The block scales the value into a physical unit (e.g., temperature, pressure) and passes it to the controller as LREAL, along with a validation flag.
- **Signal conditioning in agricultural machinery** (as indicated in the copyright notice): Conversion of speed, fill level, or velocity data into standardized signals for a higher-level control system.
- **Quality-assured data transmission:** When the validity of a signal (e.g., sensor error, communication failure) must be explicitly tracked, the synchronized valid output ensures consistent signals.

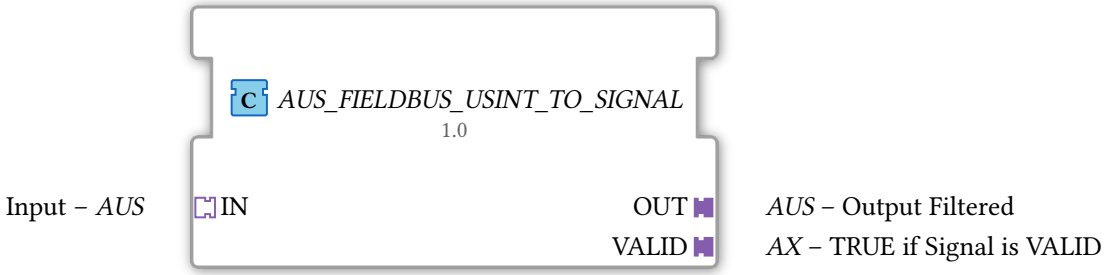
Comparison with similar function blocks

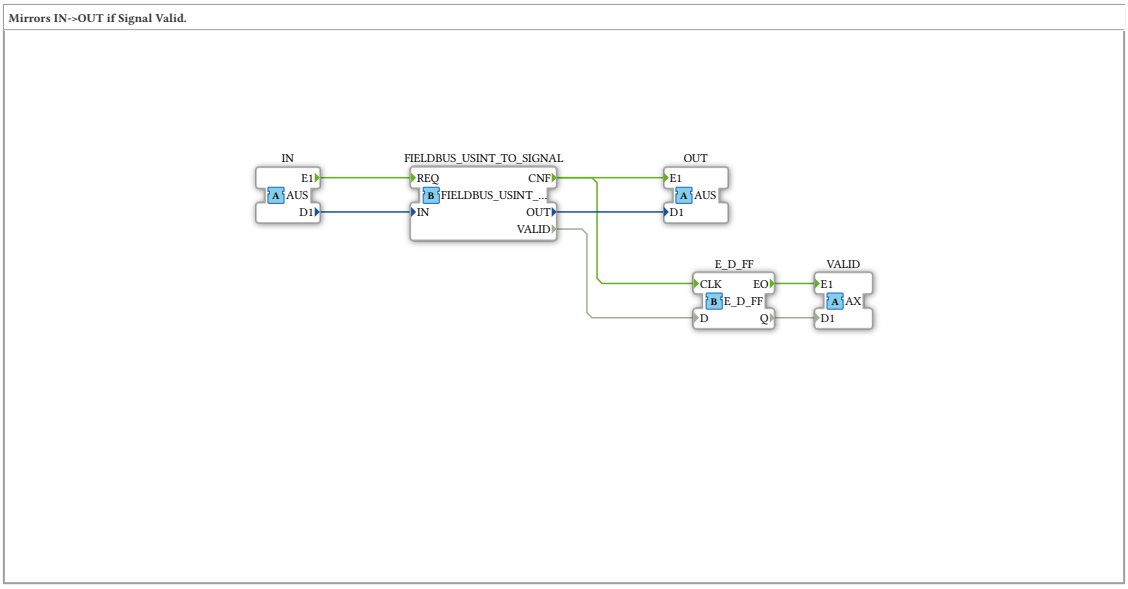
- **FIELDBUS_ULINT_TO_SIGNAL** (without scaled): A direct pass without scaling – scaling is integrated here.
- **Standard scaling function blocks** (e.g., from IEC 61131-3): These often operate with simple inputs/outputs and without adapters or validity synchronization.
- **Function blocks with integrated validation:** Many fieldbus function blocks output only a single data value. This function block stands out due to its separate, clocked validity output.

Conclusion

The **AULI_FIELDBUS_ULINT_TO_SIGNAL_SCALED** is a specialized composite function block for the scaled conversion of ULINT fieldbus data into LREAL signals with synchronized validity information. Thanks to the use of adapters and the clear separation of scaling and synchronization, it is particularly suitable for modular, safety-critical applications in automation and agricultural technology. Parameterization via `SCALE` and `OFFSET` makes it highly versatile.

AUS_FIELDBUS_USINT_TO_SIGNAL





• — ○ • — ○ • — ○ •

Introduction

The function block `AUS_FIELDBUS_USINT_TO_SIGNAL` is used to pass a USINT signal from a fieldbus adapter, taking a validity check into account. It mirrors the input value to the output if the signal is recognized as valid. An additional adapter outputs the validity status. The internal logic uses a dedicated fieldbus conversion block and an edge-triggered D flip-flop for synchronization.

Interface Structure

Event Inputs

Name	Description
`IN.E1`	Event received via adapter `IN` (socket) that triggers processing.

Event Outputs

Name	Description
`OUT.E1`	Event output via adapter `OUT` (plug) after successful mirroring of the valid signal.
`VALID.E1`	Event output via adapter `VALID` (plug) after updating the validity status.

Data Inputs

Name	Type	Description
`IN.D1`	USINT (implicit)	Input data value that is checked for validity and mirrored.

Data Outputs

Name	Type	Description
`OUT.D1`	(analogous to IN)	Output data value that mirrors the input value if the signal is valid.
`VALID.D1`	BOOL	Validation flag: `TRUE` if the input signal is recognized as valid, otherwise `FALSE`.

Adapter

Name	Type	Direction	Description
IN	adapter::types::unidirectional::AUS	Socket	Receives the raw signal (event + data) from the fieldbus.
OUT	adapter::types::unidirectional::AUS	Plug	Passes the validated signal to downstream components.
VALID	adapter::types::unidirectional::AX	Plug	Provides the validity status (event + BOOL).

Functionality

The function block (FB) reacts to an incoming event IN.E1. This triggers the internal function block FIELDBUS_USINT_TO_SIGNAL, which processes the data value IN.D1 and delivers two results:

- An output value (OUT)
- A validation signal (VALID)

The output value is passed directly to OUT.D1 and acknowledged via OUT.E1. Simultaneously, the validation signal is fed to the data input D of the D flip-flop E_D_FF. The flip-flop is clocked by the same event (via CFN), so the validity status is held stably at Q and output at VALID.D1. Only after the clock pulse is VALID.E1 triggered.

In summary: The output OUT mirrors the input IN, provided the signal is recognized as valid in the fieldbus module. The validity status is stored using edge triggering.

Technical Features

- **Adapter-based communication:** All inputs and outputs are implemented via standardized unidirectional adapters (AUS, AX), which enable a clear separation between event and data paths.
- **Edge-Triggered Validation:** A D flip-flop ensures that the validity state is only updated upon an event (rising edge) – this prevents glitches and asynchronous state transitions.
- **Reused Internal Function Block:** The function block FIELDBUS_USINT_TO_SIGNAL handles the actual fieldbus-specific conversion and validity check; the outer function block serves purely as wrapper and synchronization logic.

State Overview

The internal state is defined by the D flip-flop E_D_FF. It has two states:

State	Description
Q = FALSE (Initial)	Output VALID.D1 indicates "invalid". After each event, the current value of VALID (D) is used.
Q = TRUE	Output VALID.D1 indicates "valid". The state remains stable until the next event.

State transitions occur exclusively on each rising edge of the clock signal (event CNF of the fieldbus function block).

Application Scenarios

- **Fieldbus Signal Conditioning:** A sensor sends USINT values via a fieldbus; the function block filters out invalid values and only forwards valid data and a clear validity status.
- **Safety-Oriented Data Transmission:** In control systems that rely on valid signals (e.g., in agricultural technology), this function block (FB) can be used as a simple validation stage.
- **Interface Adaptation:** The function block connects a fieldbus-specific adapter (e.g., CANopen, PROFIBUS) to a standardized logic adapter that only triggers events when the data is valid.

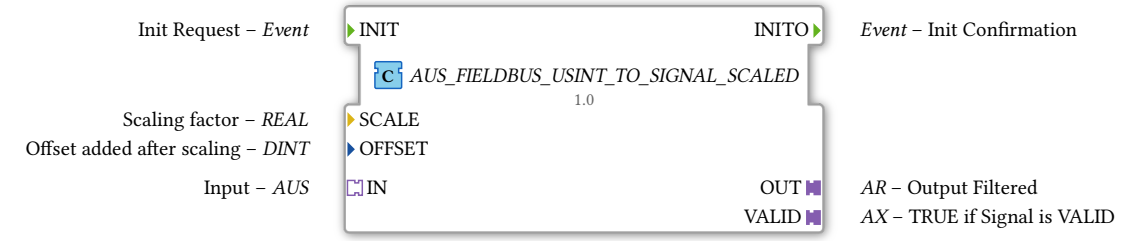
Comparison with Similar Function Blocks

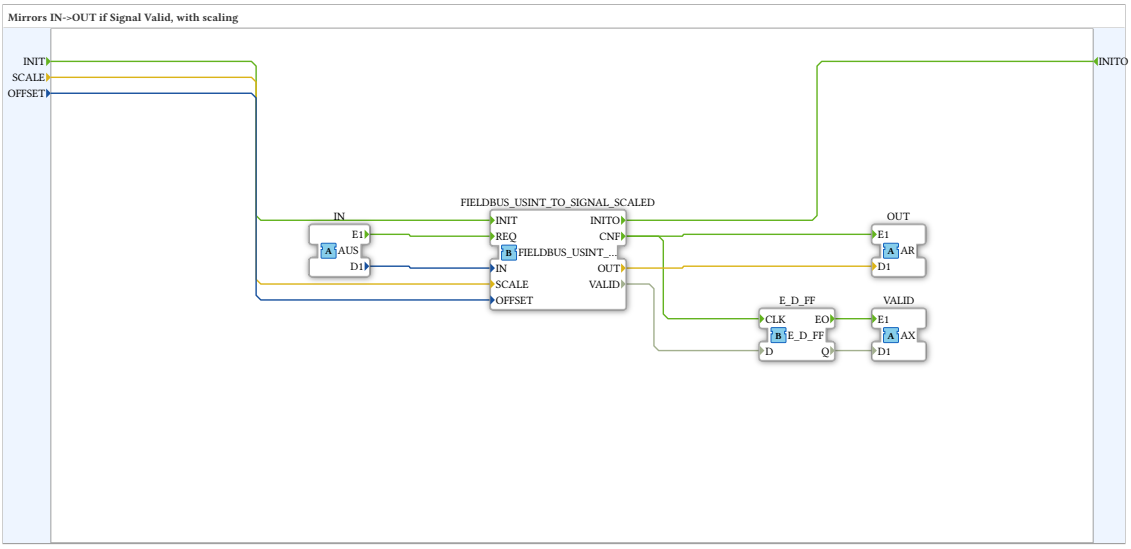
- **Simple Mirror Function Blocks** (e.g., MOVE or AUS_MIRROR): These pass the signal without validation. This FB additionally offers validation logic and a separate validity output.
- **Fieldbus Converters Without Memory:** Some function blocks output the validity signal directly and without a clock signal. Using a flip-flop here avoids metastable states and ensures deterministic output.

Conclusion

The `AUS_FIELDBUS_USINT_TO_SIGNAL` is a useful function block for the clean, validated forwarding of USINT signals from fieldbus systems. The combination of a specialized fieldbus module and an edge-triggered flip-flop ensures reliable separation of valid and invalid data. The adapter-based interface facilitates integration into modular control architectures.

AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

The function block AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED mirrors an incoming fieldbus signal value (USINT) to the output, provided the signal is recognized as valid. The input value is multiplied by a configurable scaling factor, and an offset is added. The function block is implemented as a composite block and consists of a sub-function block for scaling and an edge-triggered D flip-flop for synchronizing the validity signal. It is typically used in fieldbus signal processing to convert raw USINT values into physical quantities and pass on the data validity information.

- **INIT** (EInit): Initialization request. Passed through to the internal sub-function block.
- **INITO** (EInit): Initialization confirmation. This is handled by the internal sub-FB.
- **SCALE** (REAL): Scaling factor that is multiplied by the input value. Default value: 1.0.
- **OFFSET** (DINT): Offset value that is added to the scaled value after multiplication. Default value: 0.

No direct data outputs – output is exclusively via the adapters.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- **IN** (Socket, type `adapter::types::unidirectional::AUS`): Input adapter for the raw signal. Provides USINT data via `D1` and an event via `E1`.
- **VALID** (Plug, type `adapter::types::unidirectional::AX`): Output adapter for the validity signal. Outputs a Boolean value (TRUE = valid) via `D1` and an event via `E1`.

Functionality

1. **Initialization**: The event `INIT` is forwarded directly to the internal sub-FB `FIELDBUS_USINT_TO_SIGNAL_SCALED`. Its `INITO` is passed through to the external `INITO` output.
2. **Data Processing**: As soon as an event arrives at `IN.E1`, it triggers the execution of the internal sub-FB via its `REQ` input. The sub-FB processes the current value of `IN.D1` (USINT) using `SCALE` and `OFFSET` and calculates a scaled output value (REAL) and a validity flag (BOOL).
3. **Result Output**: After the calculation, the sub-FB generates a `CNF` event. This triggers:

- The output of the scaled value via `OUT.D1` and the event `OUT.E1`.
- The clock input (`CLK`) of the D flip-flop `E_D_FF`.

1. **Validation Signal**: The validation flag of the sub-FB is set to the data input (`D`) of the flip-flop. With each clock event, the flip-flop updates the current `D` value to its output `Q`. The event `E_D_FF.EO` then triggers the output of the validity signal via `VALID.D1` and `VALID.E1`.
2. **Interaction**: This ensures that the validity signal is updated only after scaling is complete and in a strictly timed manner with the output scaled value.

- **Composite Structure**: The function block consists of a specialized sub-function block (`FIELDBUS_USINT_TO_SIGNAL_SCALED`) for scaling and a standardized D flip-flop (`E_D_FF`) from the IEC 61499 event library.
- **Hidden Connections**: Some internal connections (e.g., for `INIT`, `SCALE`, `OFFSET`) are marked as invisible (`Visible="false"`) to improve clarity in the graphical representation.
- **Data Types**: The input value is assumed to be USINT (0…255), and the output is scaled to REAL. The offset is declared as DINT.
- **Edge-Triggered Synchronization**: The D flip-flop ensures debounced propagation of the validity signal, so it is only updated during a new processing cycle.

The function block (FB) does not have its own explicit states. The state logic is derived from the following sub-components:

- **Sub-FB `FIELDBUS_USINT_TO_SIGNAL_SCALED`**: Performs a flow-oriented calculation without state memory; it responds to each `REQ` event with a `CNF`.
- **D-Flipflop `E_D_FF`**: Has two states (Q = 0 or 1). On each positive clock cycle (CLK), Q = D is set. The state is retained until the next clock cycle.
- **Fieldbus Sensor Evaluation**: A sensor delivers a USINT value via fieldbus (e.g., 0–100% fill level or 0–255 pressure values). The function block scales this value to a physical unit (e.g., meters, bar) and outputs it along with a validity flag.
- **Signal conditioning in agricultural technology**: Agricultural machinery often requires scaled signals where the validity of the data (e.g., after a communication failure) must be explicitly indicated.
- **Initialization cascades**: The passed-through `INIT` output enables the initialization of downstream function blocks that also rely on this signal.
- **`AUS_FIELDBUS_USINT_TO_SIGNAL`** (without scaling): Provides only direct transmission of the USINT value without scaling or offset.

- **`AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED` with separate validation toggle**: An alternative design could forward the validity signal asynchronously (without a flip-flop), but this could lead to potential jitter.
- **`AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED` with integrated state machine**: Instead of a flip-flop, a custom state logic could manage the validity signal; however, this block uses proven standard components.

The `AUS_FIELDBUS_USINT_TO_SIGNAL_SCALED` function block offers a robust and maintainable solution for scaling USINT fieldbus signals and reliably forwarding the validity status. Its composite structure and the use of a D flip-flop ensure a clean temporal separation between data processing and output. It is particularly useful in systems with cycle-based data processing and explicit signal validity.

Technical Features

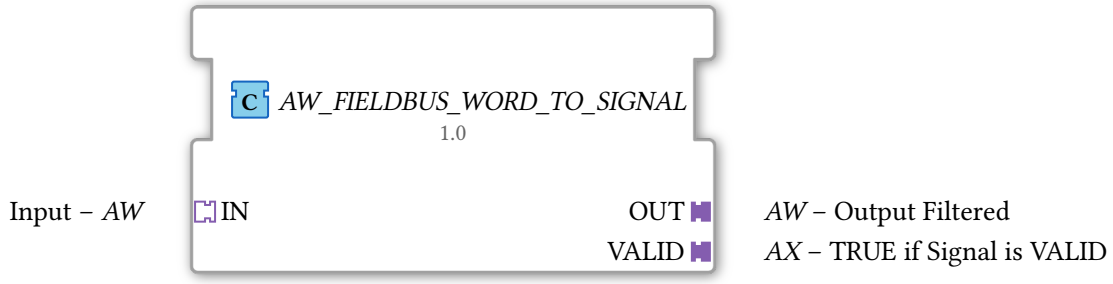
State Overview

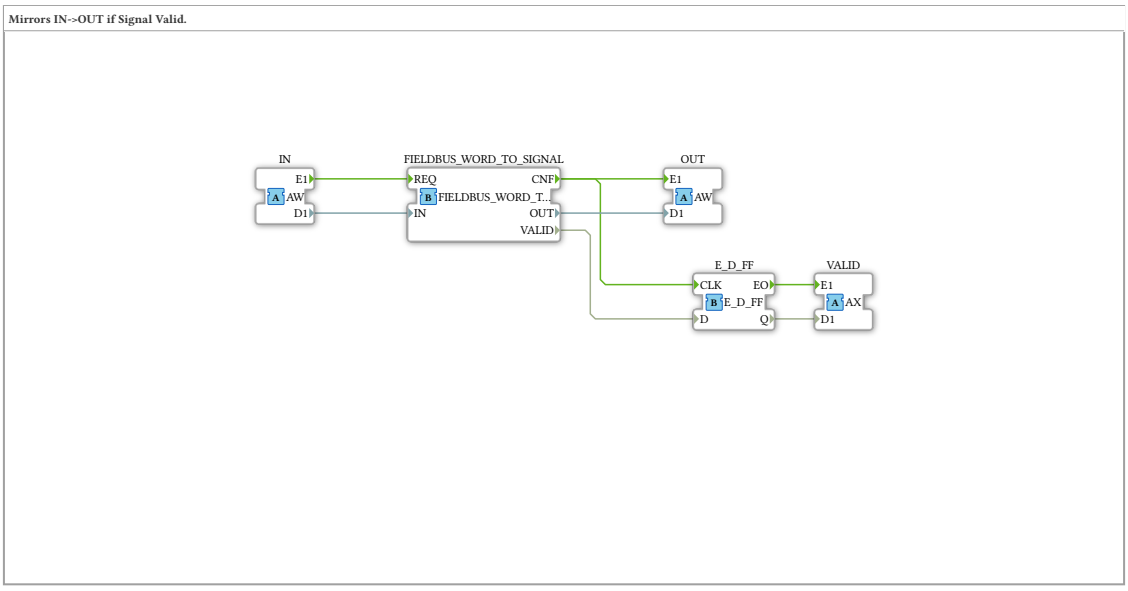
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

AW_FIELDBUS_WORD_TO_SIGNAL





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **AW_FIELDBUS_WORD_TO_SIGNAL** acts as a filter for fieldbus words. It mirrors the incoming signal (via the **IN** adapter) to the output (**OUT** adapter) if the signal is recognized as valid. Validity is indicated via a separate output adapter (**VALID**). This block is typically used in fieldbus environments to ensure that only valid data values are forwarded to subsequent components.

Interface Structure

Event Inputs

No separate event inputs are available. Event control is handled implicitly via the adapters used (see section **Adapters**).

Event Outputs

No separate event outputs available. Event output is implicit via the adapters.

Data Inputs

No separate data inputs available. Data is transferred via the connected adapters.

Data Outputs

No separate data outputs available. Data output is via the connected adapters.

Adapters

Name	Type	Description
IN	`adapter::types::unidirectional::AW` (Socket)	Input of the word to be filtered.
OUT	`adapter::types::unidirectional::AW` (Plug)	Output of the filtered word (only with a valid signal).
VALID	`adapter::types::unidirectional::AX` (Plug)	Outputs the validity status of the output signal (`TRUE` = valid).

Functionality

The function block consists internally of two sub-function blocks: `FIELDBUS_WORD_TO_SIGNAL` and `E_D_FF` (D flip-flop).

- An incoming event on `IN.E1` triggers the processing of the current word via `FIELDBUS_WORD_TO_SIGNAL.REQ`.
- The sub-block `FIELDBUS_WORD_TO_SIGNAL` checks the validity of the word and outputs a `CNF` event upon completion.
- This `CNF` event causes:
- The filtered word is passed to the output adapter via `OUT.D1` and output via `OUT.E1`.
- Simultaneously, the D flip-flop (`E_D_FF`) is clocked: The current validity status (`VALID` signal from `FIELDBUS_WORD_TO_SIGNAL`) is adopted.
- The output of the flip-flop (`Q`) is passed to the **VALID** adapter via `VALID.D1` and the event `VALID.E1` (triggered by `E_D_FF.EO`).

Thus, a valid word is only output if the internal check of the incoming signal is successful. The validity status is retained until the next processing cycle.

Technical Features

- The function block is based entirely on adapters and does not have a separate top-level event/data interface.
- The internal implementation uses a standardized fieldbus processing block (`FIELDBUS_WORD_TO_SIGNAL`) and a D flip-flop, which allows for a clear separation of the data path and status logic.
- The implementation is licensed under the **Eclipse Public License 2.0**.

State Overview

The function block itself does not have an explicit state machine. The internal state is represented by the D flip-flop:

State	Meaning
`VALID = FALSE`	The output value is invalid (old data or initial state).

VALID = TRUE`	The output value is valid and was recognized as valid during the last processing cycle.

Application Scenarios

- **Fieldbus Data Filtering**: In automation systems where only valid measured values or control commands should be passed on to subsequent logic.
- **Signal Validation**: Used in safety-critical paths where invalid data (e.g., due to transmission errors) must not be passed on without verification.
- **Adapter-Based Interfaces**: Suitable for modular systems that communicate via standardized adapters (e.g., logiBUS environment).

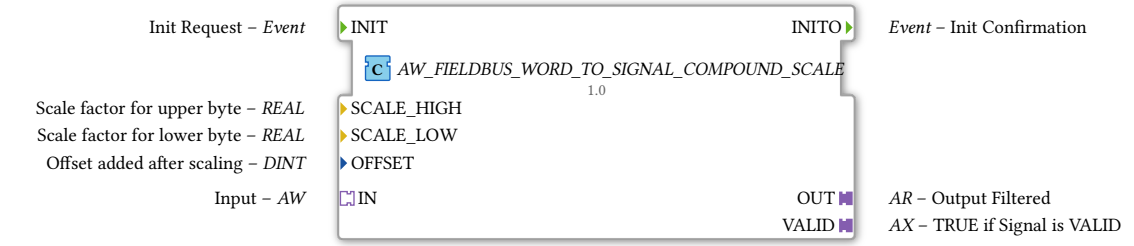
Comparison with Similar Components

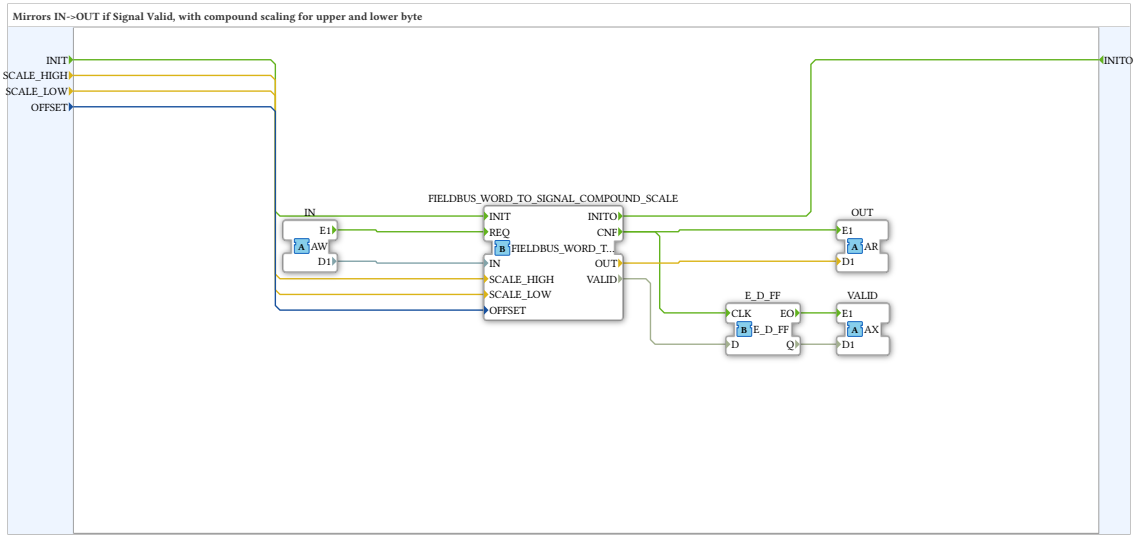
- Simple **Word-to-Signal** components without validation pass the input signal directly, regardless of its validity.
- In contrast, the **AW_FIELDBUS_WORD_TO_SIGNAL** provides explicit validity feedback and suppresses invalid values.
- Other filter blocks might include additional configuration options (e.g., thresholds, time windows), while this block is reduced to a simple "valid/invalid" decision.

Conclusion

The AW_FIELDBUS_WORD_TO_SIGNAL is a specialized filter block for fieldbus word data. It combines simple passthrough with a validity check and provides the result via separate adapters. By using standard sub-blocks and flip-flop logic, it is robust, traceable, and easily integrated into adapter-based architectures.

AW_FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE





• – ○ • – ○ • – ○ •

Introduction

The function block **AW_FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE** converts a 16-bit word received via a fieldbus into a scaled signal value. The word is multiplied byte by byte by different

scaling factors, an offset is added, and the result is output as a normalized signal. Additionally, a validity signal is provided to indicate whether the incoming value is valid.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initialization Request

Event Outputs

Event	Type	Description
INITO	EInit	Initial Confirmation

Data Inputs

Name	Type	Initial Value	Description
SCALE_HIGH	REAL	0.256	Scaling factor for the upper byte
SCALE_LOW	REAL	0.001	Scaling factor for the lower byte
OFFSET	DINT	0	Additive offset after scaling

Data Outputs

The FB does not have its own data outputs. The output data is provided via the adapters listed below.

Adapters

Adapter	Type (Direction)	Description
IN	`adapter::types::unidirectional::AW`	Incoming 16-bit word
OUT	`adapter::types::unidirectional::AR`	Output signal (scaled)
VALID	`adapter::types::unidirectional::AX`	Validation signal (TRUE = valid)

Functionality

After successful initialization (INIT event → INITO), the FB begins responding to incoming data via the **IN** adapter. As soon as the event **IN.E1** is received, the 16-bit word currently present at **IN.D1** is processed:

1. **Byte-wise scaling**

The upper byte (bits 15…8) is multiplied by `SCALE_HIGH`, the lower byte (bits 7…0) by `SCALE_LOW`.

1. **Offset addition**

The sum of both scaled byte values is incremented by the configured `OFFSET`.

1. **Output**

The result is passed to the **OUT** adapter (event **OUT.E1** with data **OUT.D1**).

1. **Validity Handling**

An internal flip-flop (`E_D_FF`) is clocked in parallel. The validity status (signal **VALID**) is taken from the internal **VALID** output of the scaling module and is retained until the next valid signal.

The internal flip-flop ensures that the validity signal is only updated upon the next input event. This guarantees a stable signal state at the output.

Technical Features

- **Compound Scaling**

Separate scaling of the upper and lower bytes allows the processing of fieldbus signals encoded in two differently scaled byte halves (e.g., temperature or pressure values with range and resolution).

- **Integrated Flip-Flop**

The validity output is implemented using an edge-triggered D flip-flop. This ensures that the validity signal is only updated upon a new input event, guaranteeing a stable output.

- **Configurable Parameters**

Scaling and offset are freely adjustable via the data inputs and can be changed at runtime.

- **Initialization**

Before first use, the function block (FB) must be initialized with the INIT event. This resets the internal states.

State Overview

The FB has an internal state represented by the flip-flop `E_D_FF`:

State	Description
UNVALID	No valid signal. The **VALID** output returns FALSE.
VALID	Valid signal detected. The **VALID** output displays TRUE.

The state changes with each new input event (i.e., with each word transmission) according to the validity information of the internal scaling module.

Application Scenarios

- **Analog Value Acquisition via Fieldbus**

A fieldbus device sends a 16-bit raw value, where the upper byte represents a coarse value (e.g., 0…255) and the lower byte a fine value (e.g., 0…1000). The physical measured value is calculated using `SCALE_HIGH` and `SCALE_LOW`.

- **Process Data Preparation**

A controller requires several scaled signals whose validity must be monitored. The function block combines scaling and validation in a single block.

- **Gateway Function**

This function block can be used as a converter between a fieldbus protocol (word-based) and an internal signal protocol (with a validity flag).

Comparison with Similar Function Blocks

- **FIELDBUS_WORD_TO_SIGNAL** (simple)

This function block scales the entire 16-bit word by a single factor. The compound function block, on the other hand, allows byte-by-byte scaling.

- **SCALE** (general)

A universal scaling function block without a fieldbus interface. This function block integrates the fieldbus adapters and validity management.

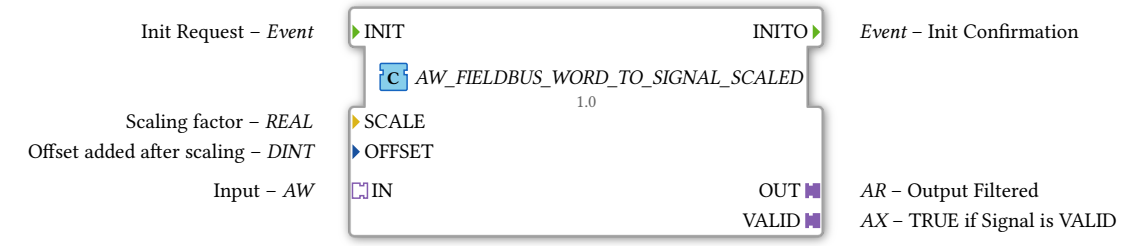
- **BADAPTER_STATUS**

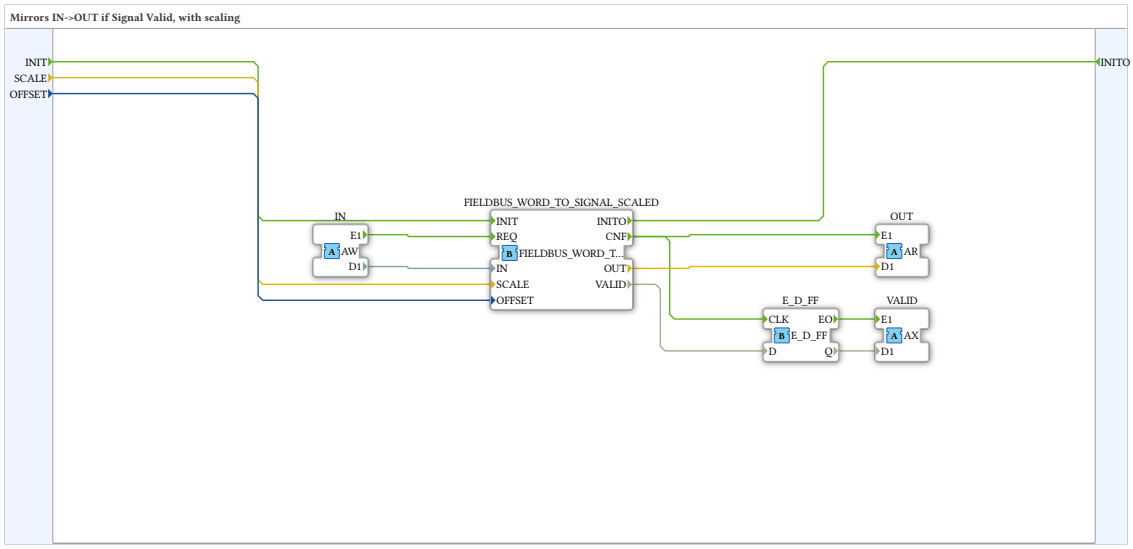
Returns only a status regarding validity, without scaling. The compound function block combines both functions.

Conclusion

The **AW_FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE** is a specialized function block for processing fieldbus word signals with different scaling for the upper and lower bytes. Its integrated validity management via a flip-flop makes it particularly suitable for safety-critical or application-specific signal processing in automation technology. Thanks to its configurable parameters and clearly structured interface, it can be easily integrated into higher-level control logic.

AW_FIELDBUS_WORD_TO_SIGNAL_SCALED





• – ○ • – ○ • – ○ •

Introduction

The function block `AW_FIELDBUS_WORD_TO_SIGNAL_SCALED` is used to convert an incoming fieldbus word (16-bit) to a physical output value, taking into account a scaling factor and offsets. Additionally, the validity of the signal is monitored and stored via a dedicated output. The block encapsulates the typical processing chain from digital transmission to the standardized analog or signal value.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Initialization request (e.g., reset of internal logic)

Event Outputs

Event	Type	Comment
INITO	EInit	Confirmation of successful initialization

Data Inputs

Name	Type	Default	Comment
SCALE	REAL	REAL#1.0	Scaling Factor
OFFSET	DINT	DINT#0	Integer Offset (after scaling)

Data Outputs

This function block does not have separate data outputs. The output values are provided via the adapters.

Adapters

Adapter	Direction	Type (unidirectional)	Comment
IN	Socket	AW	Input signal (fieldbus word)
OUT	Plug	AR	Processed and scaled output signal
VALID	Plug	AX	Boolean value: TRUE if the signal is valid

Functionality

The function block operates with an internal sub-FB `FIELDBUS_WORD_TO_SIGNAL_SCALED` and an edge-triggered D flip-flop (`E_D_FF`).

1. **Initialization:** An INIT event is forwarded to the sub-FB. Upon its completion, INITO is output.
2. **Data Processing:** An incoming event on `IN.E1` triggers the sub-FB (`REQ`). This reads the current word from `IN.D1`, scales it by the factor `SCALE`, adds `OFFSET`, and outputs the result to the internal output `OUT`. Simultaneously, a validity flag (`VALID`) is generated.
3. **Passing and Storage:** After successful processing (`CNF`), the output event is sent to `OUT.E1`, and the validity flag is transferred to the D flip-flop (clock edge). The stored value is stably available at the validation output (`VALID.D1`).
4. **Signal Validity:** If the incoming word is recognized as invalid (e.g., due to error bits or plausibility checks in the sub-FB), the sub-FB sets the `VALID` signal to FALSE. The flip-flop freezes this state until the next valid cycle occurs.

Technical Features

- **Scaling with Offset:** Processing occurs in the sequence `(Wort * SCALE) + OFFSET`. This allows for linear conversions, e.g., raw values to physical units.
- **Validity Storage:** The D flip-flop decouples the dynamic validity from the output signal. A signal marked as invalid remains invalid until a valid cycle is completed – this prevents momentary misinterpretations.
- **Adapter-based interface:** The function block uses unidirectional adapters of a type system designed for fieldbus communication (`AW`, `AR`, `AX`). The exact data types of the adapters (e.g., Word, Real, Boolean) are provided by the adapter definitions.

State overview

The function block itself does not have an explicit state machine in the XML. Its behavior results from the sequence of its internal elements:

- **Waiting for INIT:** No operation occurs before INIT is executed.
- **Operation:** After INIT, the function block listens for events at the `IN` socket. Each incoming event triggers a processing step.
- **Output:** The output event `OUT.E1` is sent immediately after the internal scaling is complete. The validity signal `VALID` is updated simultaneously.

Application Scenarios

- **Fieldbus Analog Value Processing:** A 16-bit signal transmitted from a fieldbus (e.g., 0…65535) is converted into a physical range such as 0…10V or 4…20mA using scaling.
- **Sensor Value with Validity Flag:** Sensors that deliver data via a fieldbus protocol can also send a validity status (e.g., “Data valid” bit). This is transmitted via the `VALID` output.
- **Safety-Oriented Transmission:** If the fieldbus word contains an error bit, the sub-FB `VALID` can set it to FALSE, causing downstream logic to recognize the signal as invalid.

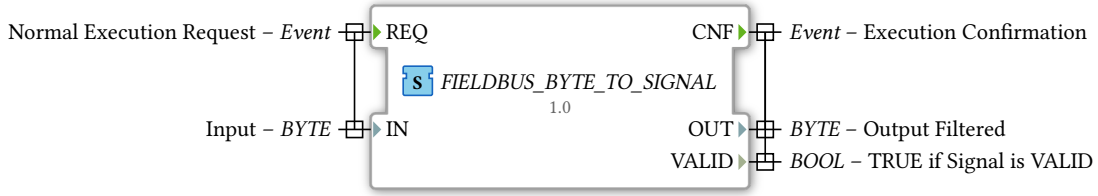
Comparison with Similar Function Blocks

Unlike simple word-to-signal converters (e.g., `FIELDBUS_WORD_TO_SIGNAL`), `AW_FIELDBUS_WORD_TO_SIGNAL_SCALED` features integrated validity storage and separate adapters for output and validity. This reduces wiring in the application and simplifies the handling of temporary errors. Function blocks without offset/scaling logic would require external computation blocks.

Conclusion

AW_FIELDBUS_WORD_TO_SIGNAL_SCALED` is a compact, functional function block for conditioning fieldbus signals. It combines scaling, offset, and validity monitoring in a reusable module. The adapter-based interface and stored validity make it particularly suitable for industrial controls where signal quality and simple parameterization are paramount.

FIELDBUS_BYTE_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **FIELDBUS_BYTE_TO_SIGNAL** is used for the simple validation of a BYTE signal from a fieldbus. It only passes the input value unchanged to the output if it lies within a defined valid range. The result of the validation is signaled via a separate Boolean output.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Normal execution trigger; processes the current value at the IN input.

Event Outputs

Event	Data Type	Comment
CNF	Event	Confirmation of execution; the outputs OUT and VALID have been updated.

Data Inputs

Name	Data Type	Initial Value	Comment
IN	BYTE	NOT_AVAILABLE_B	Input signal as a byte value.

Data Outputs

Name	Data Type	Initial Value	Comment
OUT	BYTE	16#00	Filtered output value; 0 if signal is invalid.
VALID	BOOL	FALSE	TRUE if the input signal is recognized as valid.

Adapter

None.

Functionality

The block checks the current BYTE input value **IN** against a predefined constant **VALID_SIGNAL_B** (from the import `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`) on each REQ event.

The logic in the algorithm is:

```
IF (BYTE_TO_USINT(IN) <= BYTE_TO_USINT(VALID_SIGNAL_B)) THEN
OUT := IN;
VALID := TRUE;
ELSE
OUT := BYTE#0;
```

```
VALID := FALSE;
END_IF;
```

Technical Features

- The filter is based on **two external constants** from the `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL` library:
- `NOT_AVAILABLE_B` – defines the value for "Signal not available" (initial value of IN).
- `VALID_SIGNAL_B` – defines the upper limit for valid signals.
- The conversion `BYTE_TO_USINT` ensures that the values can be compared as numbers 0…255.
- The block is implemented as a **SimpleFB** with only one state (**REQ**) – not a state machine with multiple steps.
- License: Eclipse Public License 2.0.

State Overview

The function block has exactly one execution state:

State	Action	Output Event
REQ	Executes the **REQ** algorithm (filter logic)	CNF

There are no further wait or initialization states.

Application Scenarios

- **Fieldbus Signal Validation**: A sensor delivers a BYTE value that is only physically meaningful within a specific range (e.g., 0…100). This block prevents erroneous or implausible values from entering the control logic.
- **Safety-Related Systems**: Can be used as a simple range checker for analog or digital fieldbus signals in automation technology (e.g., agriculture, machine control).
- **Signal Switch**: Separation of valid and invalid data streams – e.g., for controlling an AND filter or for error detection.

Comparison with Similar Blocks

- Unlike simple **threshold switches** (e.g., `HYSTERESIS` or `LIMIT_ALARM`), this block operates purely at the BYTE level and outputs the original value directly as long as it is within the valid range – no hysteresis or alarm function.
- Compared to a **multiplexer** (`MUX`), it does not require a second control signal but makes the decision automatically based on the input value.
- Similar blocks in IEC 61499 often exist as `RANGE_CHECK` or `VALIDATE`, but are usually designed for generic data types. This block is specifically tailored to the BYTE format commonly used in fieldbuses.
-

Conclusion

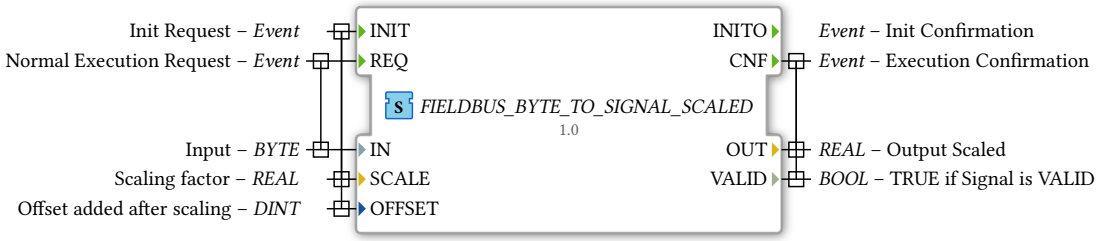
FIELDBUS_BYTE_TO_SIGNAL is a compact, specialized filter block for validating BYTE signals. It requires no boundary configuration, as these are defined via the imported constants in the library. Due to the clear separation of the data output and the validity signal, it is ideally suited for processing fieldbus data in safety-critical or fault-tolerant automation systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

FIELDBUS_BYTE_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **FIELDBUS_BYTE_TO_SIGNAL_SCALED** converts an incoming byte value into a scaled real value. It checks whether the incoming signal is valid. If so, the input value is multiplied by a scaling factor and an offset is added. Otherwise, the output is set to zero and the validity signal is reset. This block is designed for use in fieldbus systems where sensor or actuator data is transmitted as bytes and needs to be linearized or normalized.

Interface Structure

Event Inputs

Event	Type	Description	Data Carried
`INIT`	EInit	Initialization request; sets scaling and offset	`SCALE`, `OFFSET`
`REQ`	Event	Normal processing request; evaluates input `IN`	`IN`

Event Outputs

Event	Type	Description	Accompanying Data
`INITO`	EInit	Initialization Acknowledgement	–
`CNF`	Event	Confirmation of processing; provides scaled value and validity	`OUT`, `VALID`

Data Inputs

Name	Data Type	Initial Value	Description
`IN`	BYTE	`NOT_AVAILABLE_B`	Raw fieldbus value (byte)
`SCALE`	REAL	`1.0`	Scaling factor (multiplier)
`OFFSET`	DINT	`0`	Additive Offset (after scaling)

Data Outputs

Name	Data Type	Initial Value	Description
`OUT`	REAL	`0.0`	Scaled Output Value
`VALID`	BOOL	`FALSE`	`TRUE`, if the incoming byte value was deemed valid

Adapter

None.

Functionality

The function block is controlled by two events:

1. **Initialization (`INIT`)**

Upon arrival of INIT, the parameters SCALE and OFFSET are accepted. The associated algorithm (INIT) does not perform any further calculations. After successful initialization, the event INITO is sent.

1. **Processing (REQ)**

Upon arrival of REQ, the value of IN is compared to a fixed threshold value VALID_SIGNAL_B. This threshold is obtained from the imported package eclipse4diac::signalprocessing::FIELDBUS_SIGNAL.

- **Signal valid:** If IN <= VALID_SIGNAL_B is valid, the output value is calculated according to the formula

OUT := IN * SCALE + OFFSET`
and VALID is set to TRUE.

- **Signal invalid:** Otherwise, OUT is set to 0.0 and VALID is set to FALSE.

The event CNF is then sent with the current values of OUT and VALID.

The initial value of IN is NOT_AVAILABLE_B – another imported constant value that indicates an unavailable signal. Therefore, the first time REQ is called (without a previous valid input), it always produces an invalid output.

Technical Features

- The constants NOT_AVAILABLE_B and VALID_SIGNAL_B are imported from a separate library package and are not defined locally. This ensures they are consistent across the entire project.
- Scaling is linear: multiplication by SCALE and addition by OFFSET. The offset is converted from type DINT to REAL before being converted to REAL.
- The input IN is first converted from BYTE to USINT (unsigned short integer) to obtain an integer range of 0…255.
- The block has an explicit initialization mode (INIT) that allows the scaling and offset to be reset at runtime.

State Overview

The function block has two simple states, each activated by the corresponding event:

State	Triggering Event	Executable Algorithm	Outgoing Events
INIT	INIT	INIT (empty)	INITO
REQ	REQ	REQ (validation check + scaling)	CNF

There are no automatic state transitions; Each state is addressed exclusively through its corresponding event.

Application Scenarios

- **Exchange of analog sensor data via fieldbuses:**

An 8-bit sensor value (e.g., temperature 0…255) is to be converted into physical units (e.g., °C). The characteristic curve is adjusted via SCALE and OFFSET.

- **Error detection during data transmission:**

If the fieldbus defines certain byte values as invalid (e.g., 0xFF for “sensor error”), the block detects these using the threshold VALID_SIGNAL_B and reports them via VALID = FALSE.

- **Initializing Configuration Parameters:**

Before the first processing, the desired scaling factor and offset can be set using INIT.

Comparison with Similar Blocks

- **FIELDBUS_BYTE_TO_SIGNAL** (without scaling):

This block outputs the byte value directly as REAL, without multiplication or addition. It is suitable for signals that are already in the correct unit.

- **FIELDBUS_BYTE_TO_SIGNAL_SCALED** (existing block):

Additionally offers linear scaling and an offset. The user can flexibly adjust the conversion without changing the algorithm.

- **FIELDBUS_WORD_TO_SIGNAL_SCALED**:

Analog block for 16-bit values (WORD) with identical scaling logic, but a different input data type.

Conclusion

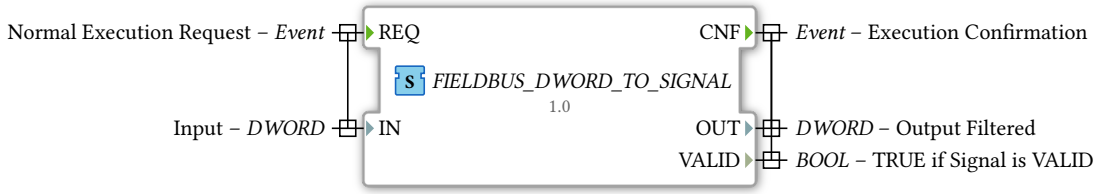
The function block `FIELDBUS_BYTE_TO_SIGNAL_SCALED` is a practical tool for processing fieldbus signals in byte format. It combines simple validity checking with freely configurable linear scaling. Thanks to the use of imported constants, it can be integrated into standardized signal processing pipelines. The clear separation of initialization and operation, along with the two event interfaces, makes it a robust and easy-to-understand building block for industrial automation solutions.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

FIELDBUS_DWORD_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FIELDBUS_DWORD_TO_SIGNAL` mirrors and validates a DWORD input signal. If the input value is valid, it is passed unchanged to the output, and the `VALID` signal is set. Otherwise, the block returns a zero value, indicating invalidity.

Interface Structure

Event Inputs

Event	Comment
REQ	Normal execution trigger; processes the current value of `IN`.

Event Outputs

Event	Comment
CNF	Execution confirmation; sent after calculation.

Data Inputs

Variable	Type	Initial Value	Comment
IN	DWORD	`NOT_AVAILABLE_DWM`	Input signal

Data Outputs

Variable	Type	Initial Value	Comment
OUT	DWORD	`16#00000000`	Filtered output value
VALID	BOOL	`FALSE`	`TRUE`, if the signal is valid

Adapter

None.

Functionality

The function block is activated by an event at REQ. The associated algorithm compares the input value IN (as an unsigned integer) with a predefined threshold VALID_SIGNAL_DW:

- If DWORD_TO_UDINT(IN) <= DWORD_TO_UDINT(VALID_SIGNAL_DW), the signal is considered **valid**:
- OUT receives the value of IN.
- VALID is set to TRUE.
- Otherwise (signal invalid):
- OUT is set to DWORD#0.
- VALID is set to FALSE.

After the calculation is complete, the output event CNF is sent.

Technical Features

- The limit value VALID_SIGNAL_DW comes from the imported package eclipse4diac::signalprocessing::FIELDBUS_SIGNAL. It defines the maximum permissible value for a valid signal.
- The initial value of the input IN is NOT_AVAILABLE_DWM (also imported), so the function block already assumes an invalid state at startup.
- Converting DWORD to UDINT enables simple numerical comparison (≤) without sign issues.
- The function block is implemented as a **SimpleFB** with a single ECA state, REQ.

State Overview

State	Action	Output Event
REQ	Execute REQ Algorithm	CNF

There is only one state; after its processing, the function block returns to the IDLE state (implicitly).

Application Scenarios

- **Fieldbus Signal Processing**: A DWORD value from a fieldbus (e.g., CANopen, Profibus) is validated before being used in the controller.
- **Data Validation**: This function block can be used as a filter to detect invalid or erroneous telegrams and replace them with a defined zero value.
- **Safety-Critical Systems**: The clear separation of valid and invalid signals allows for monitoring and further processing of signal quality.

Comparison with Similar Function Blocks

- A simple **MOVE** function block would pass the value through without validation; FIELDBUS_DWORD_TO_SIGNAL adds the validation logic.
- Function blocks like FIELDBUS_SIGNAL_TO_DWORD (if available) perform the opposite conversion – here, a DWORD signal is converted into a validated signal with a validity flag.
- This function block is specialized for DWORD values; corresponding variants for other data types (e.g., WORD, SINT) would be conceivable.

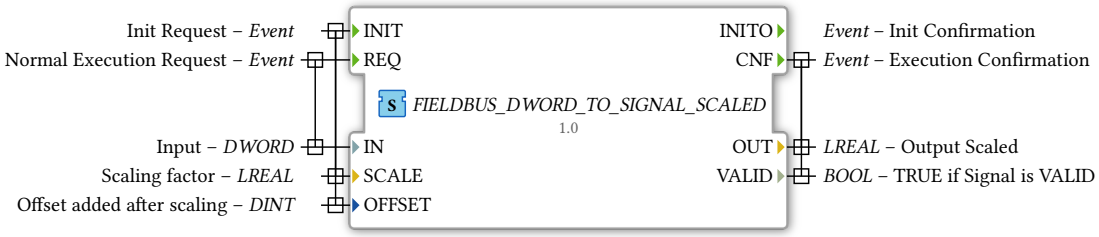
Conclusion

FIELDBUS_DWORD_TO_SIGNAL` is a compact yet essential function block for robust fieldbus connectivity. It combines signal transmission with simple validity checking and provides the validity as a separate Boolean signal. This allows downstream functions to quickly react to valid data or initiate error handling. The use of imported constants ensures a consistent definition of the validity threshold across multiple functions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

FIELDBUS_DWORD_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

The function block **FIELDBUS_DWORD_TO_SIGNAL_SCALED** converts a fieldbus DWORD signal into a scaled LREAL value. It checks the validity of the input signal against a predefined range and, if the signal is valid, calculates the output value by multiplying it by a scaling factor and adding an offset. If the signal is invalid, the output is set to 0 and the validity flag is set to FALSE.

Event	Type	Description	
INIT	EInit	Initialization request; passes scaling parameters	
REQ	Event	Normal execution request for processing the input	
Event	Type	Description	
–––––	––	–––––––	
INITO	EInit	Initialization Acknowledgement	
CNF	Event	Execution Acknowledgement; returns scaled output and validity flag	
Name	Type	Initial Value	Description
————	———	—————————	———————
IN	DWORD	NOT_AVAILABLE_DWM	Input value from the fieldbus
SCALE	LREAL	LREAL#1.0	Scaling factor (multiplier)
OFFSET	DINT	DINT#0	Offset added after scaling
Name	Type	Initial Value	Description
———	———	———————	———————
OUT	LREAL	LREAL#0.0	Scaled output value
VALID	BOOL	FALSE	TRUE if the input signal is valid

Adapters

No adapters available.

Adapter

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

The function block has two states: **INIT** and **REQ**.

- **INIT state**: Triggered by the INIT event. The associated algorithm is empty, meaning no further actions are performed. The INITO output confirms the completion of the initialization.
- **REQ state**: Triggered by the REQ event. The REQ algorithm performs the following steps:

1. Checks whether the input value IN, interpreted as a UDINT, is less than or equal to the constant `VALID_SIGNAL_DW`.
2. If this condition is met, the signal is considered valid:

- `OUT` is calculated as: `UDINT_TO_LREAL(DWORD_TO_UDINT(IN)) * SCALE + DINT_TO_LREAL(OFFSET)`
- `VALID` is set to `TRUE`.

1. Otherwise (signal invalid):

- `OUT` is set to `0.0`.
- `VALID` is set to `FALSE`.

1. The CNF event is then triggered, providing the results (OUT, VALID).

The constants `NOT_AVAILABLE_DWM` and `VALID_SIGNAL_DW` are taken from imported libraries and define which DWORD values are considered "not available" and "valid," respectively.

- The function block uses a range check by comparing `DWORD_TO_UDINT(IN)` with `VALID_SIGNAL_DW`. This assumes that valid fieldbus signals lie within a specific range of numbers.
- The conversion is performed without floating-point errors because the DWORD is first converted to UDINT and then to LREAL.
- The scaling factor `SCALE` and the offset `OFFSET` can be set during initialization or runtime, but are only used in the REQ algorithm.
- The function block is implemented as a SimpleFB and is suitable for cyclic processing.

State	Trigger	Action	Output
INIT	INIT Event	INIT Algorithm (empty)	INITO
REQ	REQ Event	REQ Algorithm (calculation and validation)	CNF

The function block does not require state transitions between INIT and REQ – both states are triggered directly by their respective events.

- **Agricultural Fieldbus Controllers**: Conversion of raw DWORD sensor values (e.g., rotational speeds, pressures) into physical units, taking sensor characteristics into account.
- **Scaling of Analog Values**: When a fieldbus device delivers a 32-bit raw value that must first be converted into a normalized value (e.g., 0-100%) using a factor and offset.
- **Signal Validation**: Only valid signals (within a defined range) are processed further; for invalid values, a defined zero value is output.
- **FIELDBUS_DWORD_TO_SIGNAL**: A similar function block without scaling capabilities, which outputs the raw value directly as LREAL.
- **FIELDBUS_DWORD_TO_SIGNAL_SCALED** additionally offers the parameters `SCALE` and `OFFSET` for flexible conversion.

- **Event-driven converters**: Other function blocks may use different validation logic (e.g., bitmasks) or support different fieldbus types.

This function block combines signal validation with a simple linear transformation, making it suitable for many applications.

The `FIELDBUS_DWORD_TO_SIGNAL_SCALED` is a compact and reusable function block for processing fieldbus signals. It enables reliable validity checking and simultaneous scaling in a single step. Due to the clear separation of initialization and processing, it is suitable for both one-time and cyclic calls in automation systems, particularly in agricultural technology.

Technical Features

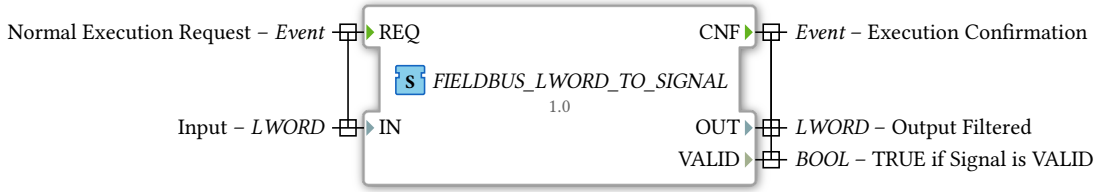
State Overview

Application Scenarios

Comparison with Similar Function Blocks

Conclusion

FIELDBUS_LWORD_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **FIELDBUS_LWORD_TO_SIGNAL** converts an LWORD value from a fieldbus system into a valid signal. It accepts an input value and passes it on unchanged only if its value lies within a defined valid range. Otherwise, a zero value is output, and the validity is signaled as `FALSE`.

Interface Structure

Event Inputs

Name	Type	Comment
REQ	Event	Normal execution request (with data input `IN`)

Event Outputs

Name	Type	Comment
CNF	Event	Execution Confirmation (with data outputs `OUT` and `VALID`)

Data Inputs

Name	Type	Initial Value	Comment
IN	LWORD	`NOT_AVAILABLE_LWM`	Input value from the fieldbus

Note: `NOT_AVAILABLE_LWM` is a constant defined in the project (e.g., `LWORD#16#FFFFFFFFFFFFFFFF`) that represents an invalid or unavailable value.

Data Outputs

Name	Type	Initial Value	Comment
OUT	LWORD	`LWORD#16#0000000000000000`	Filtered output value (0 for invalid signal)

VALID	BOOL	FALSE	Validation flag (TRUE if signal is valid)

Adapters

No adapters available.

Functionality

The function block is activated by an event at the input REQ. The current value of IN is read and processed according to the following logic:

1. The incoming LWORD value is converted into an unsigned integer (ULINT).
2. This value is compared to a constant VALID_SIGNAL_LW, also interpreted as a ULINT.

- If the input value is **less than or equal to** this threshold, the signal is considered valid:
- OUT receives the value of IN.
- VALID is set to TRUE.
- Otherwise (value greater than threshold), the signal is considered invalid:
- OUT is set to zero (LWORD#16#0000000000000000).
- VALID is set to FALSE.

1. After processing is complete, the event CNF is output to confirm completion.

The constant VALID_SIGNAL_LW defines the upper limit of the valid value range. Common values are, for example, LWORD#16#0000FFFFFFFFFFFF or similar ranges specified by the fieldbus protocol.

Technical Features

- **Type Conversion:** The comparison is performed after converting both LWORD values to ULINT using LWORD_TO_ULINT. This interprets the values as unsigned 64-bit integers, enabling a consistent comparison independent of bit pattern-specific interpretations.
- **Use of project-wide constants:** The values NOT_AVAILABLE_LWM (initial value for IN) and VALID_SIGNAL_LW (limit value) are obtained via imports from a common signal processing package. They are not defined within the function block itself and can therefore be adjusted globally.
- **No additional state machine:** The function block has only a single state (REQ), which is exited immediately after the algorithm is executed. There are no internal delays or wait loops.

State Overview

State	Description	Action
REQ	Waits for an event at input REQ.	Executes the algorithm REQ (checking and, if necessary, filtering) and then sends the event CNF.

After each execution, the function block returns to the state REQ, so it is immediately ready for a new task after each processing cycle.

Application Scenarios

- **Fieldbus Signal Validation:** Used in automation systems where analog or digital signals are received via a fieldbus (e.g., PROFIBUS, CANopen). The function block filters out values that are outside the permissible range (e.g., error or initialization values) and outputs a defined zero value instead.
- **Safety-Related Applications:** If the fieldbus protocol specifies certain bit patterns for "signal not available" or "error," these can be detected by the limit check and marked as invalid.
- **Data preprocessing before logical operations:** The output VALID can be used directly as an enable signal for subsequent function blocks, so that they are only activated with valid input signals.

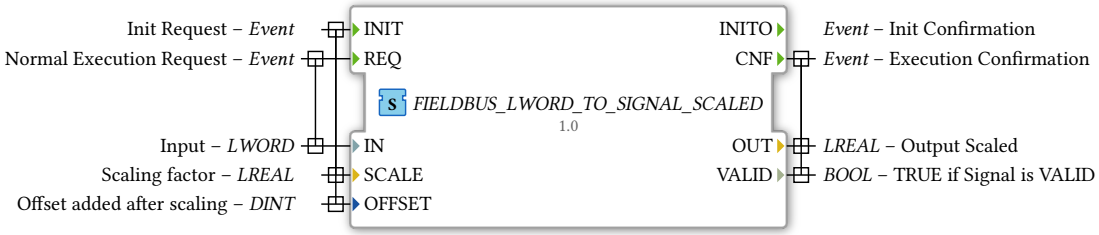
Comparison with similar function blocks

- **Simple filters (e.g., FIELDBUS_LWORD_FILTER):** These usually perform a simple limit check without a validity flag. The function block described here additionally offers the Boolean output VALID, which simplifies decision-making in the controller.
- **Signal converters (e.g., LWORD_TO_REAL):** These convert the data type but do not check the validity of the original value. The function block FIELDBUS_LWORD_TO_SIGNAL combines validation and optional conversion (here as a 1:1 pass-through on validity).
- **Monitor blocks (e.g., WATCHDOG):** These only monitor the signal's validity, not its value range. This block adds range checking.

Conclusion

The function block `FIELDBUS_LWORD_TO_SIGNAL` is a simple yet effective tool for processing fieldbus LWORD signals. The combination of range checking and outputting a validity flag ensures a clear separation of valid and invalid values. The use of project-wide constants provides flexibility, while the clear state logic guarantees reliable and fast processing. It is particularly suitable for applications where error signals or initialization values must be reliably detected and forwarded.

FIELDBUS_LWORD_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **FIELDBUS_LWORD_TO_SIGNAL_SCALED** converts a raw fieldbus signal in **LWORD** format into a scaled **LREAL** output value. It mirrors the input signal to the output, provided the signal is valid, and applies linear scaling. This function block is designed for use in industrial automation, particularly in the context of fieldbus systems.

Interface Structure

Event Inputs

Event	Type	Description	Accompanying Data
INIT	EInit	Initialization Request: Set Scale Factor (SCALE) and Offset (OFFSET)	SCALE, OFFSET
REQ	Event	Normal Execution Request: Process Input Signal (IN)	IN

Event Outputs

Event	Type	Description	Accompanying Data
INITO	EInit	Initialization Acknowledge	–
CNF	Event	Execution Acknowledge	OUT, VALID

Data Inputs

Name	Type	Initial Value	Description
IN	LWORD	NOT_AVAILABLE_LWM	Fieldbus Raw Signal (LWORD)
SCALE	LREAL	LREAL#1.0	Scaling Factor (Multiplication)
OFFSET	DINT	DINT#0	Offset added after scaling

Data Outputs

Name	Type	Initial Value	Description
OUT	LREAL	LREAL#0.0	Scaled Output Value (LREAL)
VALID	BOOL	FALSE	Indicates whether the input signal is valid (TRUE = valid)

Adapters

No adapters available.

Functionality

The module has two algorithms that are triggered by the **INIT** and **REQ** events.

- **INIT** (on event `INIT`): The algorithm is empty, but sets the output event `INITO`. This allows the scaling parameters (SCALE and OFFSET) to be initialized via the INIT call without immediate processing.
- **REQ** (on event `REQ`): Performs the actual signal processing:

1. The input signal `IN` (LWORD) is interpreted as an unsigned integer value (`LWORD_TO_ULINT`).
2. This value is compared to an internally defined valid threshold value, **VALID_SIGNAL_LW** (also an LWORD, imported constant). The comparison is performed at the ULINT level.
3. **Validity**: If `IN` ≤ `VALID_SIGNAL_LW`, the signal is considered valid:

- `OUT = (ULINT_TO_LREAL(IN)) * SCALE + DINT_TO_LREAL(OFFSET)`
- `VALID = TRUE`

1. **Invalidity**: Otherwise, the signal is considered invalid:

- `OUT = 0.0`
- `VALID = FALSE`

1. After the calculation, the output event `CNF` is sent.

The scaling is linear: `OUT = IN_als_LREAL * SCALE + OFFSET`. The default value of `SCALE` is 1.0, and that of `OFFSET` is 0, so identical transmission occurs without adjustment.

Technical Features

- **Validity Check**: The function block uses two imported constants:
- `NOT_AVAILABLE_LWM`: Initial value for the input `IN` (indicates "not available").
- `VALID_SIGNAL_LW`: Threshold up to which a signal is considered valid (inclusive).

The comparison is performed by converting both LWORD values to ULINT, so that negative values (as bit patterns) are also correctly interpreted.

- **Type Conversion**: The raw value is converted from LWORD to LREAL via ULINT (64-bit). Potential precision losses should be noted for very large ULINT values.
- **Scaling Parameters**: `SCALE` (LREAL) and `OFFSET` (DINT) are set only during the INIT event and remain unchanged for all subsequent REQ calls.
- **Event Control**: The function block has a simple state machine with two states and requires explicit initialization before first processing.

State Overview

State	Triggered by	Output Algorithm	Event Output	Description
INIT	Event INIT	INIT	INITO	Initialization: Adopt Scaling Parameters
REQ	Event REQ	REQ	CNF	Signal Processing: Scaling and Validation Check

The function block switches between these two states only through the corresponding events. There is no explicit transition to a wait state after initialization; the function block remains in the last state until a new event arrives.

Application Scenarios

- **Analog Values from the Fieldbus**: A raw LWORD value from an analog input module (e.g., 0...10V, scaled as a raw value) is converted into a physical value (e.g., pressure, temperature) using SCALE and OFFSET.
- **Validation Filter**: Signals that are outside the permissible range (e.g., wire break or overflow) are detected by the validity check and marked as invalid, while the output is set to 0.
- **Parameter Initialization**: Scaling can be configured via the INIT event when the controller starts, without the actual processing having to run yet.

Comparison with Similar Function Blocks

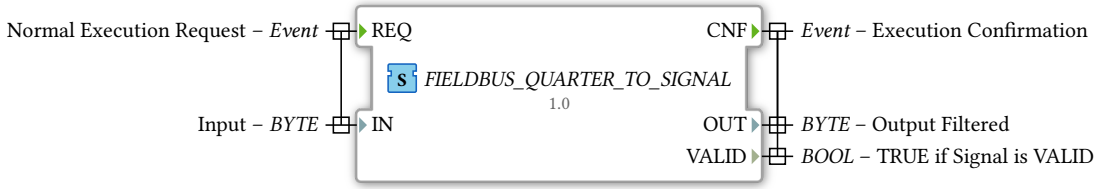
- **FIELDBUS_LREAL_TO_SIGNAL_SCALED**: An analog function block for LREAL inputs that offers similar validation and scaling.
- **FIELDBUS_SINT_TO_SIGNAL_SCALED**: For signed 8-bit values.
- **Simple Scalers without Validation**: Function blocks like `SCALE` or `MUL_ADD` only offer the mathematical operation, but no fieldbus-specific validation.

This function block combines validation (via the threshold `VALID_SIGNAL_LW`) with linear scaling and is therefore particularly suitable for fieldbus applications where the validity of the signal must be ensured before further processing.

Conclusion

FIELDBUS_LWORD_TO_SIGNAL_SCALED is a compact yet powerful function block for processing fieldbus signals in LWORD format. It offers simple scaling, integrated validation, and a clean separation of initialization and execution. Through the use of standardized imports (`VALID_SIGNAL_LW`)It can be easily integrated into existing libraries and adapted to specific fieldbus protocols. The module is available under the Eclipse Public License 2.0 and is suitable for use in industrial control systems with IEC 61499-compliant runtime environments.

FIELDBUS_QUARTER_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FIELDBUS_QUARTER_TO_SIGNAL` serves as a signal validation and pass-through block. It checks whether an incoming BYTE value `IN` is within a valid range and, if so, passes it unchanged to `OUT`. Additionally, a Boolean signal `VALID` is set, indicating the validity status of the output. This block is suitable for filtering fieldbus signals where certain values are marked as invalid or “don’t care”.

Interface Structure

Event Inputs

Name	Type	Comment	With Variables
`REQ`	Event	Normal Execution Order	`IN`

Event Outputs

Name	Type	Comment	With Variables
`CNF`	Event	Execution Confirmation	`OUT`, `VALID`

Data Inputs

Name	Type	Comment	Initial Value
`IN`	`BYTE`	Input Signal	`NOT_AVAILABLE_2bit` (Imported Constant)

Data Outputs

Name	Type	Comment	Initial Value
`OUT`	`BYTE`	Filtered Output Signal	`16#00`

VALID	BOOL	TRUE, if the signal is valid	FALSE

Adapter

No adapters available.

Functionality

When an event occurs at input REQ, the algorithm REQ is executed:

1. The incoming value IN is compared to the imported constant DONT_CARE_2bit.
2. If IN is less than DONT_CARE_2bit, the value is considered valid:

- OUT receives the value of IN.
- VALID is set to TRUE.

1. Otherwise (if IN is greater than or equal to DONT_CARE_2bit), the signal is classified as invalid:

- OUT is reset to BYTE#0.
- VALID is set to FALSE.

1. After processing, the event CNF is triggered.

The comparison uses the semantics of the imported constants:
DONT_CARE_2bitdefines the threshold above which values are considered "irrelevant" or invalid. The initial value ofINis set toNOT_AVAILABLE_2bit`, which is typically above this threshold, so the function block outputs a defined invalid state upon startup.

Technical Features

- **Constant Import**: The values DONT_CARE_2bit and NOT_AVAILABLE_2bit are imported from a higher-level signal processing package. They must be defined in the target system and represent meaningful threshold values (e.g., 16#FC for "don't care").
- **Simple State Logic**: The function block has only a single EC state, REQ. Every event immediately triggers a calculation and output.
- **Compatibility**: Designed according to IEC 61499-1, usable in the 4diac IDE and other compliant runtime environments.

State Overview

Only one state exists:

State	Description	Output Event	Output Actions
REQ	Processes the input and updates outputs	CNF	Set OUT and VALID

Application Scenarios

- **Fieldbus Monitoring**: A sensor delivers 8-bit data on a fieldbus (e.g., CANopen, PROFIBUS), where certain values are coded as "not available" or "error." The function block extracts the valid measured values and displays their validity.
- **Signal Preprocessing**: Before further processing in a controller, invalid telegram segments are filtered out, so that subsequent blocks only work with plausible data.
- **Gateway Between Bus Systems**: Translation of a bus's "don't care" conventions into a uniform validity signal for the target system.

Comparison with Similar Function Blocks

- **FIELDBUS_QUARTER_TO_SIGNAL** specializes in distinguishing between valid and invalid values using a fixed threshold.
- A general **value filter** (e.g., F_BAND) could, on the other hand, query a lower and upper limit.
- A **validation checker** (e.g., VALIDITY_CHECK) often additionally sets Quality of Service flags.

This function block impresses with its simple logic, tailored to fieldbus types.

Conclusion

FIELDBUS_QUARTER_TO_SIGNAL` is a compact, efficient function block for validating 8-bit fieldbus signals. It reliably separates valid measured values from those marked as invalid and provides the validity

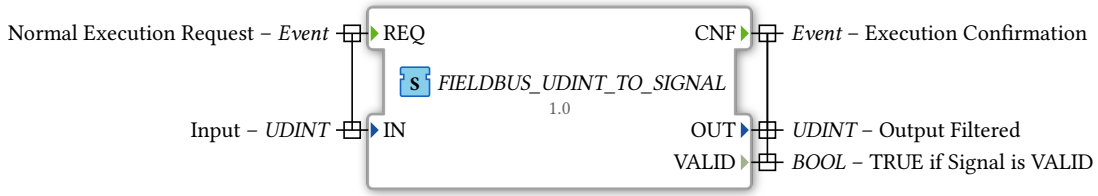
as a separate Boolean signal. Thanks to its streamlined state logic and the use of imported constants, it can be flexibly integrated into various signal processing chains.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

FIELDBUS_UDINT_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **FIELDBUS_UDINT_TO_SIGNAL** is used to check the validity of an incoming UDINT value and, if the check is successful, forward it unchanged to the output. If the value is invalid, the output is set to a defined error value (0) and the validity indicator is deactivated. This function block is typically used in fieldbus signal processing to ensure that only valid measurement or control values are passed on to subsequent logic.

Interface Structure

Event Inputs

Event	Description
REQ	Normal execution request; triggers the processing of the IN input.

Event Outputs

Event	Description
CNF	Confirms processing is complete. Triggered according to the algorithm.

Data Inputs

Name	Type	Initial Value	Description
IN	UDINT	`DWORD_TO_UDINT(NOT_AVAILABLE_DWM)`	The input value to be checked.

Data Outputs

Name	Type	Initial Value	Description
OUT	UDINT	`16#00000000`	The filtered output value. With a valid signal, it corresponds to IN; otherwise, it's 0.
VALID	BOOL	FALSE	TRUE if the signal is valid, i.e., IN is within the valid range.

Adapter

None.

Functionality

As soon as an event arrives at the **REQ** input, the algorithm is executed:

1. The value of **IN** is compared to the predefined threshold `VALID_SIGNAL_DW`.

2. If **IN** is less than or equal to this threshold, **OUT** is set to **IN** and **VALID** to `TRUE`.
3. Otherwise, **OUT** is set to `0` (UDINT) and **VALID** to `FALSE`.

The limit value `VALID_SIGNAL_DW` and the initial value `NOT_AVAILABLE_DWM` are imported as constants from the package `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`. They represent a maximum permissible signal value and an "unavailable" marker, respectively.

Technical Features

- **Data Type** – The function block works exclusively with the type `UDINT` (Unsigned Double Integer, 32-bit).
- **Initial Value** – The input IN is set to the value `NOT_AVAILABLE_DWM` by default. This is converted from a previously defined DWORD using the function `DWORD_TO_UDINT`.
- **Limit Value** – The validity is determined by a simple comparison with the imported constant `VALID_SIGNAL_DW`. All values ≤ this limit are considered valid.
- **Simple State Logic** – The function block consists of exactly one state (REQ) and executes the algorithm on every request.

State Overview

The function block is implemented as a simple FB (SimpleFB) and has only one state:

State	Action	Output Event
REQ	Algorithm REQ	CNF

There is no state machine with multiple steps or branches.

Application Scenarios

- **Fieldbus Signal Conditioning** – In automation systems where UDINT values are received via a fieldbus, the function block is used to filter invalid values or values outside the valid range (e.g., sensor data that must not exceed a maximum value).
- **Agricultural Technology** – Originally developed for use in agricultural control systems (e.g., HR Agrartechnik) to validate analog measurements such as rotational speeds or pressures.
- **Quality Assurance** – Preprocessing of measurement data before it is incorporated into control loops or logging.

Comparison with Similar Function Blocks

- **FIELDBUS_DINT_TO_SIGNAL** – Analog function block for signed 32-bit values (DINT).
- **FIELDBUS_BOOL_TO_SIGNAL** – Comparable function block for Boolean signals; threshold validation is omitted.
- **FIELDBUS_REAL_TO_SIGNAL** – For floating-point numbers, possibly with different tolerance logic.

This function block is specifically designed for the UDINT data type and uses a simple absolute limit for validation.

Conclusion

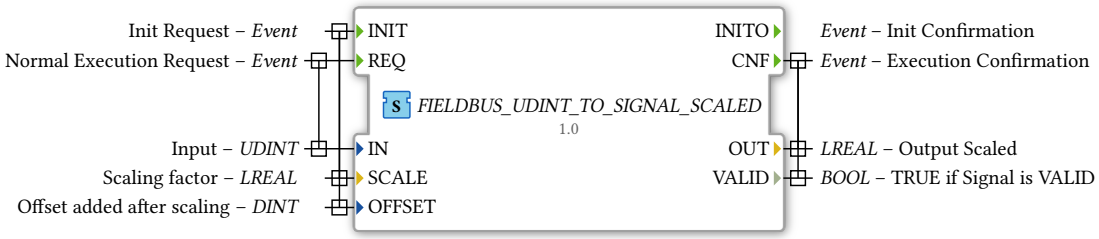
FIELDBUS_UDINT_TO_SIGNAL is a compact and efficient function block for filtering UDINT fieldbus signals. Its clear separation of valid and invalid values, along with its simple interface, makes it well-suited for robust automation solutions. The use of imported constants increases reusability and adaptability to various fieldbus protocols.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

FIELDBUS_UDINT_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FIELDBUS_UDINT_TO_SIGNAL_SCALED` processes a UDINT value received via the fieldbus. It mirrors the input `IN` to the output `OUT`, but only if the incoming signal is recognized as valid. Additionally, the value is scaled by a factor and an offset is applied. This function block is typically used in the signal processing chain of fieldbus protocols to convert raw integer values into physical quantities (e.g., as LREAL).

Interface Structure

Event Inputs

Event	Type	With Parameters	Description
`INIT`	EInit	`SCALE`, `OFFSET`	Initialization – sets scaling factor and offset.
`REQ`	Event	`IN`	Normal execution event – processes the current input value.

Event Outputs

Event	Type	With Parameters	Description
`INITO`	EInit	–	Confirmation of successful initialization.
`CNF`	Event	`OUT`, `VALID`	Processing confirmation – provides the scaled value and the validity signal.

Data Inputs

Name	Type	Initial Value	Description
`IN`	UDINT	`DWORD_TO_UDINT(NOT_AVAILABLE_DWM)`	Raw value from the fieldbus.
`SCALE`	LREAL	`LREAL#1.0`	Multiplicative scaling factor.
`OFFSET`	DINT	`DINT#0`	Additive Offset (after scaling).

Data Outputs

Name	Type	Initial Value	Description
`OUT`	LREAL	`LREAL#0.0`	Scaled output value.
`VALID`	BOOL	`FALSE`	`TRUE`, if the input value was recognized as valid.

Adapter

No adapters available.

Functionality

The function block has two states: `INIT` and `REQ`.

- **INIT State**: Upon a `INIT` event, the empty algorithm is executed (no action is taken), and subsequently, `INITO` is output. The parameters `SCALE` and `OFFSET` are passed to the function block but are only used during the next `REQ` processing.
- **REQ State**: On a `REQ` event, the following logic is executed:

1. The incoming value `IN` is compared to a predefined threshold `VALID_SIGNAL_DW` (from the import `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL::VALID_SIGNAL_DW`).
2. If `IN <= DWORD_TO_UDINT(VALID_SIGNAL_DW)` is true, the input is considered valid. Then `OUT` is calculated as:

OUT := UDINT_TO_LREAL(IN) * SCALE + DINT_TO_LREAL(OFFSET)`
and `VALID` is set to `TRUE`.

1. Otherwise, `OUT` is set to `0.0` and `VALID` to `FALSE`. This indicates that the transmitted fieldbus value is invalid (e.g., error value or communication failure).

The initial value of `IN` is preset to `DWORD_TO_UDINT(NOT_AVAILABLE_DWM)`, a constant from the same import that represents a special "not available" value.

Technical Features

- The validity check is based on two imported constants:
- `VALID_SIGNAL_DW`: Upper limit for valid values.
- `NOT_AVAILABLE_DWM`: Default value for unavailable signals (used as the initial value for `IN`).
- Scaling is performed in LREAL (double precision), allowing for precise processing of very large or small input values.
- The offset is defined as `DINT`, but is converted to `LREAL` before addition. The entire computational range is covered by the LREAL representation.

State Overview

The function block uses a simple state diagram with two states:

- **INIT**: Executed after the `INIT` event. The algorithm `INIT` is empty, and the output `INITO` is triggered.
- **REQ**: This event is executed after the `REQ` event. Here, the algorithm `REQ` is executed, which performs scaling and validation, and then outputs `CNF`.

There are no other states or loops.

Application Scenarios

- **Scaling of Sensor Values**: A fieldbus device provides a raw UDINT value (e.g., 0 … 65535). `SCALE` converts a physical quantity (e.g., pressure in bar), and the offset accounts for zero-point shifts.
- **Validation Check**: Typical fieldbus protocols define certain bit patterns as "invalid." This function block automatically filters out these values and sets `VALID = FALSE`.
- **Initialization**: At system startup, the scaling factor can be set via `INIT`, while the `REQ` cycle regularly processes the current process values.

Comparison with Similar Function Blocks

Similar function blocks are often generic `SCALE` functions that do not include a validation check. This function block adds explicit validation of the fieldbus signal. Other implementations might make the validity range configurable as a parameter or via separate thresholds. This function block uses fixed constants defined in the import library, which simplifies configuration but is less flexible.

Conclusion

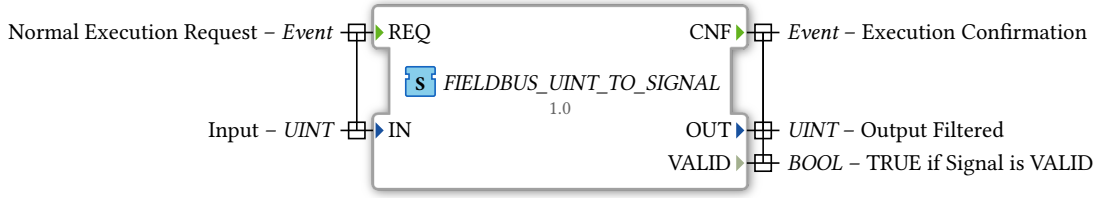
The `FIELDBUS_UDINT_TO_SIGNAL_SCALED` function block provides a robust and simple way to scale raw data from a fieldbus while simultaneously evaluating signal quality. The combination of scaling factor and offset, along with integrated validation, makes it particularly suitable for industrial automation applications where both the conversion and plausibility check of measured values are required.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

FIELDBUS_UINT_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

The FIELDBUS_UINT_TO_SIGNAL function block is used to process fieldbus signals. It mirrors an incoming UINT value to the output, provided it is classified as valid. Validity is determined by comparing it to a configured threshold (VALID_SIGNAL_W). If the input value exceeds this threshold, the output is reset and the signal is marked as invalid.

Event	Data Type	Comment	
REQ	Event	Normal Execution Trigger (connected to IN)	
Event	Data Type	Comment	
–––––	–––––	———–	
CNF	Event	Execution confirmation (connected to OUT and VALID)	
Name	Data Type	Initial Value	Comment
——	–––––	———–	———–
IN	UINT	WORD_TO_UINT(NOT_AVAILABLE_WM)	Input value (raw fieldbus signal)
Name	Data Type	Initial Value	Comment
——	–––––	———–	———–
OUT	UINT	16#0000	Filtered Output Value
VALID	BOOL	FALSE	TRUE if the signal is recognized as valid

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

None.

Functionality

On each REQ event, the REQ algorithm is executed:

1. The incoming value IN is compared to the constant `VALID_SIGNAL_W`.
2. If `IN <= WORD_TO_UINT(VALID_SIGNAL_W)` is valid, the signal is considered valid:

- `OUT` receives the value of `IN`.
- `VALID` is set to TRUE.

1. Otherwise (signal invalid):

- `OUT` is reset to 0.
- `VALID` is set to FALSE.

1. The CNF event is then triggered.

The constants `NOT_AVAILABLE_WM` (for the initial value of IN) and `VALID_SIGNAL_W` (as the threshold) are obtained from external imports and define the valid range of the fieldbus signal.

- Use of type conversions (`WORD_TO_UINT`) for comparison with the imported constants.
- The function block imports two constants from the package `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`: `NOT_AVAILABLE_WM` and `VALID_SIGNAL_W`. These must be defined or provided accordingly in the application.
- The function block is implemented as a simple FB, i.e., It has exactly one execution state.

The function block consists of a single state, **REQ**.

- **START → REQ**: The algorithm is executed as soon as the REQ event occurs. After processing, CNF is output, and the state remains in REQ (cyclic processing).
- Processing of analog sensor values from a fieldbus where a specific value range is defined as valid (e.g., 0–1000 for a pressure sensor).
- Filtering of communication errors or invalid telegrams by setting values outside the valid range to zero and marking them as invalid.
- Preprocessing of fieldbus signals before forwarding them to subsequent logic blocks.

Unlike simple pass-through blocks (e.g., `MOVE`), `FIELDBUS_UINT_TO_SIGNAL` performs a deliberate validity check and distinguishes between valid and invalid values. Other blocks, such as `LIMIT` or `SCALE`, offer range limits but no separate Boolean output for validity. This block is particularly suitable for applications requiring explicit signal status indication.

The FIELDBUS_UINT_TO_SIGNAL function block provides a simple and efficient way to check the validity of fieldbus signals and either pass them through or discard them. Its clear interface and the use of configurable constants make it versatile for use in IEC 61499 automation applications.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

State Overview

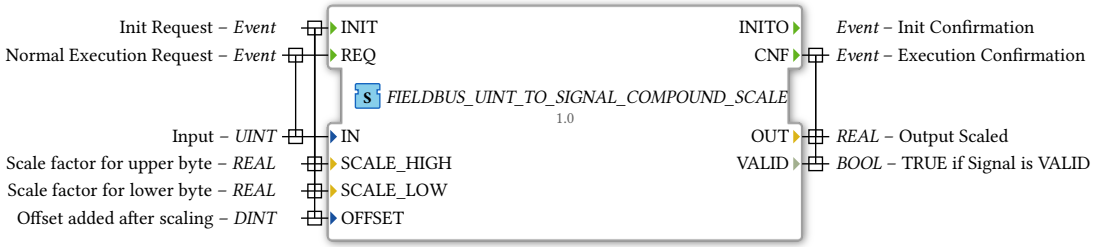
Application Scenarios

Comparison with Similar Function Blocks

Conclusion

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

FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` is used to process a 16-bit fieldbus signal (type `UINT`) into a scaled real value. The incoming word is split into an upper and a lower byte, each byte

is multiplied by its own scaling factor, and an offset is added. Additionally, a validity signal (`VALID`) is output, based on a predefined threshold. This function block is particularly suitable for protocols that encode two measured variables (e.g., temperature and pressure) in a single register.

Interface Structure

Event Inputs

Event	Type	Description
`INIT`	EInit	Initialization Request (Parameter Transfer)
`REQ`	Event	Normal Execution Request (Processing of `IN`)

Event Outputs

Event	Type	Description
`INITO`	EInit	Initialization Confirmation
`CNF`	Event	Execution Confirmation (after `REQ`)

Data Inputs

Variable	Type	Initial Value	Description
`IN`	UINT	`WORD_TO_UINT(NOT_AVAILABLE_WM)`	Raw value from fieldbus
`SCALE_HIGH`	REAL	`0.256`	Scaling factor for the upper byte
`SCALE_LOW`	REAL	`0.001`	Scaling factor for the lower byte
OFFSET`	DINT	`0`	Offset added after scaling (in REAL)

Data Outputs

Variable	Type	Initial Value	Description
`OUT`	REAL	`0.0`	Scaled Output Value
`VALID`	BOOL	`FALSE`	`TRUE`, if the input signal is recognized as valid

Adapter

None.

Functionality

1. **Initialization** (`INIT`):

The algorithm `INIT` is empty, but the parameters `SCALE_HIGH`, `SCALE_LOW`, and `OFFSET` are linked to the event via the `With` relationship and can thus be set when `INIT` is called.

1. **Normal Processing** (`REQ`):

- The input value `IN` is compared to a constant `VALID_SIGNAL_W` (from the imported name `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL::VALID_SIGNAL_W`):

IF (IN <= WORD_TO_UINT(VALID_SIGNAL_W)) THEN`

- If the condition is **true** (signal valid):
- The upper byte is extracted by right-shifting 8 bits and masking with `0x00FF`:

highByte := (IN >> 8) & 0xFF`

- The lower byte is obtained directly by masking:

lowByte := IN & 0xFF`

- Both bytes are converted to `REAL` and multiplied by the respective scaling factors. and the offset is added:

OUT := highByte * SCALE_HIGH + lowByte * SCALE_LOW + OFFSET`

- `VALID` is set to `TRUE`.
- If the condition is **false** (invalid signal):
- `OUT` is set to `0.0`.
- `VALID` is set to `FALSE`.
- The event `CNF` is then output.

Technical Features

- **Compound Scaling**: The function block allows separate scaling for the upper and lower bytes of a 16-bit word. This is useful when two different physical quantities are combined in one register.
- **Validity Check**: Comparison with `VALID_SIGNAL_W` defines a threshold. Values above this threshold are considered invalid – typical for fieldbus error codes (e.g., "Not Available").
- **Parameterization via `INIT`**: The scaling factors and offset can be set once via the `INIT` event so that they remain constant during operation.
- **Initial Values**: The predefined values (`SCALE_HIGH=0.256`, `SCALE_LOW=0.001`, `OFFSET=0`) are examples; they should be adapted to the specific application.

State Overview

The FB has two EC states:

- **INIT**: Executes the empty algorithm `INIT` and exits the state via the output `INITO`.
- **REQ**: Executes the algorithm `REQ`, which performs all signal processing, and exits the state via the output `CNF`.

The FB is implemented as a SimpleFB; there are no other states such as IDLE or WAIT.

Application Scenarios

- **Two-in-one Sensor Values**: A temperature sensor and a pressure sensor provide their values as 8-bit numbers each in a 16-bit register. The function block (FB) converts these values into physical units using separate scaling.
- **Fieldbus Error Detection**: If the fieldbus protocol specifies a particular bit code (e.g., all bits 1) for invalid measured values, validity can be detected using the threshold value `VALID_SIGNAL_W`.
- **Post-processing of Logging Data**: Raw data from a fieldbus logger is converted back into physical units using the original scaling.

Comparison with Similar Function Blocks

Function Block	Property	Difference
`SCALE` (e.g., according to IEC 61131)	Linear factor and offset on the entire value	No division into bytes, no validity check
`LINEAR` (Fieldbus Standard)	Moving average or simple linear transformation	No compound approach
`UINT_TO_REAL`	Direct type conversion	No scaling, no error detection

This function block is unique in combining byte separation with dual scaling and integrated validation, making it particularly suitable for special fieldbus protocols with compound registers.

Conclusion

The `FIELDBUS_UINT_TO_SIGNAL_COMPOUND_SCALE` is a powerful function block for processing 16-bit fieldbus signals that encode two independent measured values in a single register. Separate scaling of high and low bytes, along with integrated validation, allows for the efficient and reliable conversion of complex sensor registers into physical quantities. Parameterization via `INIT`The event and the clear separation of initialization and operation make the function block reusable and adaptable.

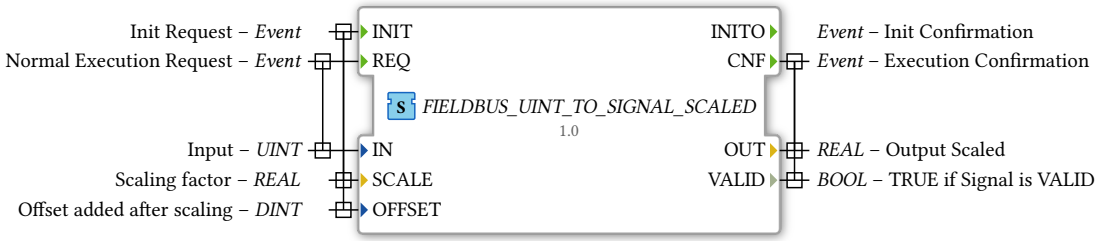
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

FIELDBUS_UINT_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **FIELDBUS_UINT_TO_SIGNAL_SCALED** processes a UINT value received via a fieldbus. It checks the validity of the input signal and, if valid, scales it to a REAL output using a multiplication constant (SCALE) and an additive offset (OFFSET). If the signal is invalid, the output is set to zero and a validity flag is set to FALSE. A typical application is the conversion of raw fieldbus data into physical quantities while simultaneously detecting exceptional conditions such as wire breaks or sensor errors.

Interface Structure

Event Inputs

- **INIT** (EInit): Initialization request. This event sets the parameters `SCALE` and `OFFSET`.
- **REQ** (Event): Normal execution request. This event processes the current value `IN`.

Event Outputs

- **INITO** (EInit): Initialization confirmation. Sent after processing the INIT event.
- **CNF** (Event): Execution confirmation. Sent after processing the REQ event and carries the output data `OUT` and `VALID`.

Data Inputs

Name	Type	Initial Value	Description
IN	UINT	WORD_TO_UINT(NOT_AVAILABLE_WM)	Raw value from the fieldbus to be scaled.
SCALE	REAL	REAL#1.0	Scaling factor (multiplication).
OFFSET	DINT	DINT#0	Additive offset to be added after scaling.

Data Outputs

Name	Type	Initial Value	Description
OUT	REAL	REAL#0.0	Scaled output value. 0.0 if the signal is invalid.
VALID	BOOL	FALSE	Validation flag: TRUE if the input signal is valid.

Adapter

None.

Functionality

The FB has two operating states, which are controlled by the corresponding events.

- **INIT state**: Upon receiving `INIT`, the parameters `SCALE` and `OFFSET` are adopted (the algorithm is empty; the transfer occurs implicitly via the `With` link). The event `INITO` is then sent.
- **REQ State**: Upon receiving `REQ`, the following algorithm is executed:

1. Check whether the input value `IN` is less than or equal to a predefined validity threshold. This threshold is given by the imported constant `VALID_SIGNAL_W` (from the namespace `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`).
2. **Valid (`IN <= VALID_SIGNAL_W`)**:

- `OUT := UINT_TO_REAL(IN) * SCALE + DINT_TO_REAL(OFFSET)`
- `VALID := TRUE`

1. **Invalid (`IN > VALID_SIGNAL_W`)**:

- `OUT := 0.0`
- `VALID := FALSE`

1. Subsequently, the event `CNF` is sent with the current values of `OUT` and `VALID`.

The validity threshold is provided as part of the imported library and must be defined according to the fieldbus standard used. The initial value of `IN` is `WORD_TO_UINT(NOT_AVAILABLE_WM)`, which, by default, results in an invalid signal being detected if no data is passed.

Technical Features

- **Use of Imported Constants**: Validation is performed using constants from the `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL` namespace. These must be correctly defined in the target system (e.g., `VALID_SIGNAL_W` as the maximum valid UINT value, `NOT_AVAILABLE_WM` as an indicator for "not available").
- **Empty INIT Algorithm**: The INIT event is used solely for parameter passing. No explicit initialization logic exists – the values of `SCALE` and `OFFSET` are taken directly from the data inputs. Therefore, these parameters can also be set before the first INIT (see Initial Values).
- **Type Preservation**: The input is UINT, the output is REAL. Scaling is performed using REAL arithmetic after an explicit type conversion.
- **No State Machine Logic**: The FB is implemented as a SimpleFB, which only has the two states INIT and REQ. There are no time-based or repetitive processes; each execution is strictly event-driven.

State Overview

The FB has exactly two EC states:

State	Input	Algorithm	Output Event
INIT	INIT	– (empty)	INITO
REQ	REQ	REQ	CNF

There is no state loop or dwell time. After initialization, the function block (FB) starts in the INIT state and can then be executed any number of times via REQ. A subsequent INIT resets the parameters.

Application Scenarios

- **Scaling of Analog Values**: A fieldbus delivers a raw 16-bit measurement value (e.g., 0–4095). Using `SCALE = 0.001` and `OFFSET = -5.0`This results in, for example, a pressure in bar.
- **Error Value Detection**: Many fieldbus protocols define specific bit patterns for "sensor errors" or "wire breaks." The function block (FB) detects such values by classifying them as being above the validity threshold (e.g., 0xFFFE or 0x8000) and setting the output to zero.
- **Initialization Sequence**: After system startup, an INIT call can load the scaling parameters from a configuration file. Subsequently, REQ calls are sent in rapid succession for cyclical measurement processing.

Comparison with Similar Function Blocks

- **FIELDBUS_UINT_TO_SIGNAL (without scaling)**: A simpler FB that directly outputs the UINT value as a REAL and only performs the validity check. Scaling must be done externally.
- **FIELDBUS_UINT_TO_SIGNAL_FILTERED**: In addition to scaling, a low-pass or median filter is applied – suitable for noisy signals.
- **FIELDBUS_INT_TO_SIGNAL_SCALED**: Analog function block for signed inputs (INT), with corresponding adjustment of the validation logic.

This function block is characterized by its simplicity, clear separation of scaling and validation detection, and the use of imported constants, which allows for flexible adaptation to various fieldbus standards.

Conclusion

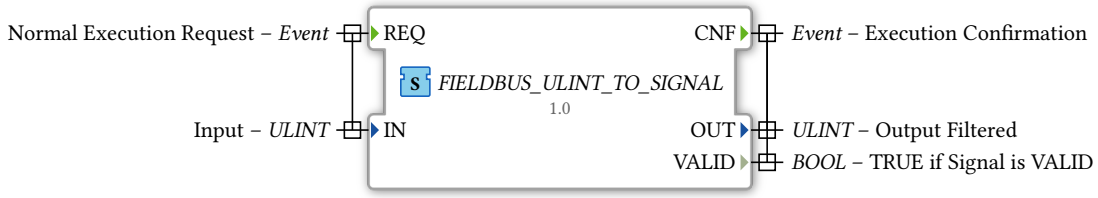
FIELDBUS_UINT_TO_SIGNAL_SCALED is a compact, event-driven function block for converting a UINT fieldbus signal into a scaled REAL value. The integrated validation check enables easy error detection. This block is particularly suitable for applications where raw bus data needs to be converted into physical quantities with minimal effort, while simultaneously monitoring signal quality. The clear interface and the use of imported constants facilitate reuse in different project contexts.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

FIELDBUS_ULINT_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **FIELDBUS_ULINT_TO_SIGNAL** serves as a filter block for fieldbus signals of type `ULINT`. It checks whether an incoming value lies within the valid signal range and, depending on the result, either outputs it unchanged or sets the output to a defined invalid value. The VALID output signal indicates the signal validity status. This block is particularly suitable for use in safety-critical or quality-monitoring applications where invalid or faulty fieldbus values must be reliably detected and handled.

Interface Structure

Event Inputs

Event	Description
REQ	Normal execution request – triggers the processing of the input value IN. The event output CNF is sent after the action is completed.

Event Outputs

Event	Description
CNF	Execution confirmation – sent after the calculation of OUT and VALID.

Data Inputs

Name	Type	Initial Value	Description
IN	ULINT	`LWORD_TO_ULINT(NOT_AVAILABLE_LWM)`	Input signal (raw value from the fieldbus)

Note: `NOT_AVAILABLE_LWM` is a constant imported from the library `FIELDBUS_SIGNAL`, representing an invalid or unavailable signal value.

Data Outputs

Name	Type	Initial Value	Description
OUT	ULINT	`16#0000000000000000`	Filtered output value – corresponds to IN if valid, otherwise 0.
VALID	BOOL	`FALSE`	Signal validity – `TRUE` if valid input, otherwise `FALSE`.

Adapter

This component has no adapter interfaces.

Functionality

After an event at the **REQ** input, the following algorithm is executed:

1. Check: **IN** ≤ `LWORD_TO_ULINT(VALID_SIGNAL_LW)`

VALID_SIGNAL_LWis an imported constant from theFIELDBUS_SIGNAL` library and defines the upper limit of the valid value range.

1. **Case – Valid Input** (Condition met):

- `OUT` := `IN`
- `VALID` := `TRUE`

1. **Case – Invalid Input** (Condition not met):

- `OUT` := `0`
- `VALID` := `FALSE`

The **CNF** event output is then triggered.

Technical Features

- The function block operates deterministically and requires only a single processing stage.
- The threshold values `NOT_AVAILABLE_LWM` and `VALID_SIGNAL_LW` are defined as external constants and stored centrally in the library `FIELDBUS_SIGNAL`. They can be adjusted without modifying the function block.
- The output value for an invalid signal is set to 0, which represents a defined "invalidity value" in many fieldbus systems.
- The initial value of `IN` is declared as "not available," so the function block provides a defined state even without a prior event.

State Overview

The function block is implemented as a **SimpleFB** and has only one state:

State	Description
REQ	Input state: Waiting for event REQ. Upon entry, the algorithm `REQ` is executed, and then the output `CNF` is sent. Afterward, the function block remains in the same state.

There are no further states or branches – the functionality is purely process-oriented.

Application Scenarios

- **Fieldbus Signal Conditioning:** A sensor delivers a ULINT value that is only physically meaningful within a specific range (e.g., rotational speed, pressure). The function block filters out invalid measurements.
- **Safety Chain:** In a safety-related controller, faulty or deviating fieldbus values are detected and replaced by a defined substitute value (0). Simultaneously, a validity signal is provided for the downstream logic.
- **Quality Monitoring:** The VALID output can be used to monitor data integrity, e.g., to trigger alarms or switch to alternative signal sources.

Comparison with Similar Function Blocks

The 4diac framework includes other filter blocks for different data types, such as `FIELDBUS_DINT_TO_SIGNAL` or `FIELDBUS_REAL_TO_SIGNAL`. All of these share the principle of validity checking using a predefined upper limit. This particular block is specifically designed for the `ULINT` data type, enabling efficient and type-safe use in applications with 64-bit unsigned values.

Conclusion

The **FIELDBUS_ULINT_TO_SIGNAL** is a simple yet reliable function block for signal validation in fieldbus environments. Its clear distinction between valid and invalid values, along with the provision of a BOOL output for validity, makes it ideally suited for use in modular automation systems where data

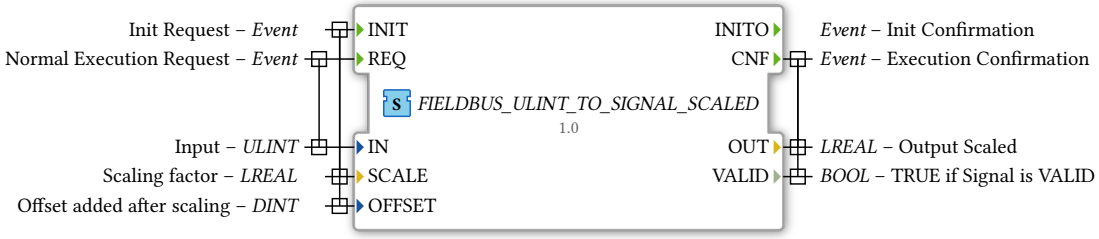
quality and error detection are essential. The use of external constants allows for flexible adaptation to different signal ranges without modifying the component itself.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

FIELDBUS_ULINT_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **FIELDBUS_ULINT_TO_SIGNAL_SCALED** is used to convert a raw ULINT value (e.g., from a fieldbus) into a scaled LREAL value. The validity of the input signal is checked against a constant threshold (`VALID_SIGNAL_LW`). If the input is valid, linear scaling is performed (multiplication by a factor and addition of an offset); otherwise, the output is set to 0,0 and the validity is negated.

Interface Structure

Event Inputs

Event	Type	Description	With Variables
INIT	EInit	Initialization Request	SCALE, OFFSET
REQ	Event	Normal Execution Request	IN

Event Outputs

Event	Type	Description	With Variables
INITO	EInit	Initialization Acknowledgement	–
CNF	Event	Execution Acknowledgement	OUT, VALID

Data Inputs

Variable	Type	Initial Value	Description
IN	ULINT	`LWORD_TO_ULINT(NOT_AVAILABLE_LWM)`	Raw value from the fieldbus
SCALE	LREAL	`1.0`	Scaling factor
OFFSET	DINT	`0`	Offset added after scaling

Data Outputs

Variable	Type	Initial value	Description
OUT	LREAL	`0.0`	Scaled output value (valid only if VALID = TRUE)
VALID	BOOL	`FALSE`	Validation signal: TRUE if IN ≤ `VALID_SIGNAL_LW`, otherwise FALSE

Adapters

No adapters available.

Functionality

The function block has two operating states, which are activated by the **INIT** and **REQ** events:

1. **INIT State**

When the INIT event arrives, the algorithm `INIT` is executed (currently empty). The **INITO** event is then sent.

The **SCALE** and **OFFSET** parameters can be set via the inputs connected to INIT.

1. **REQ State**

When a REQ event occurs, the algorithm `REQ` is executed:

- Check: `IN ≤ LWORD_TO_ULINT(VALID_SIGNAL_LW)`
- **Valid** (condition met):

OUT := ULINT_TO_LREAL(IN) * SCALE + DINT_TO_LREAL(OFFSET)`
VALID := TRUE`

- **Invalid** (condition not met):

OUT := 0.0`
VALID := FALSE`

- Subsequently, the **CNF** event is output with the new values of **OUT** and **VALID**.

The constants `NOT_AVAILABLE_LWM` and `VALID_SIGNAL_LW` are taken from the imported package `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL` and define the threshold for validation.

Technical Features

- The function block is implemented as a **SimpleFB** with two states, making it deterministic and eliminating the need for complex state machines.
- Validation is performed by comparing the result with the imported `VALID_SIGNAL_LW`. This allows for application-specific configuration of the valid value range.
- In the event of an invalid signal, the output **OUT** is reset to 0,0 – a defined error handling mechanism.
- Scaling and offset are passed as LREAL and DINT, respectively. The offset is internally converted to LREAL, enabling negative values.
- The initial value of **IN** (`NOT_AVAILABLE_LWM`) is chosen so that it is recognized as invalid on the first pass if no valid fieldbus value is present.

State Overview

State	Description	Trigger	Output Event
INIT	Initialization (no action)	INIT	INITO
REQ	Input signal processing (scaling or error handling)	REQ	CNF

Application Scenarios

- **Fieldbus Sensors**: A 64-bit ULINT value from a pressure sensor is converted to bar using a factor (e.g., 0.001), and a zero-point offset is added. Validation prevents the processing of communication errors.
- **Error Handling**: If the value exceeds `VALID_SIGNAL_LW` (e.g., due to sensor failure), **VALID** is set to FALSE, preventing the higher-level controller from accepting the value.
- **Initialization**: During system startup, the scaling and offset can be loaded from a parameter set via INIT.

Comparison with Similar Function Blocks

- **FIELDBUS_SINT_TO_SIGNAL_SCALED** – same logic, but for signed 8-bit values (SINT). This block processes 64-bit unsigned values.
- **FIELDBUS_ULINT_TO_SIGNAL (without scaling)** – simple type conversion without scaling and without integrated validation.

- **GENERIC_SCALER** – a general scaling block where the validity check must be implemented externally.

Conclusion

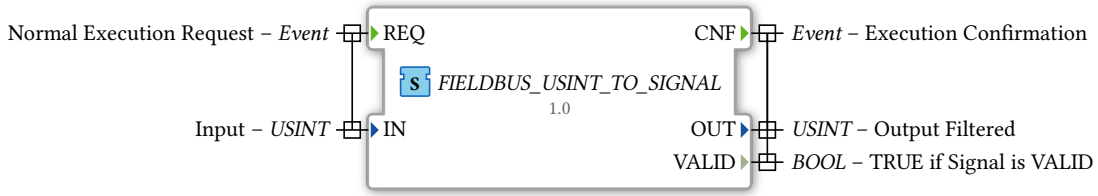
The **FIELDBUS_ULINT_TO_SIGNAL_SCALED** function block offers a compact, configurable solution for the reliable conversion of raw fieldbus data into physical values. By combining validity checking, scaling, and offset, it is ideally suited for industrial automation tasks where data integrity and easy calibration are required.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

FIELDBUS_USINT_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Introduction

The FIELDBUS_USINT_TO_SIGNAL function block processes a USINT signal from a fieldbus. It checks whether the incoming value represents a valid signal range and, if so, mirrors it to the output. Otherwise, the output is set to a defined zero value, and the validity signal is reset.

Interface Structure

Event Inputs

Event	Description	With
REQ	Normal Execution Order	IN

Event Outputs

Event	Description	With
CNF	Execution Confirmation	OUT, VALID

Data Inputs

Name	Type	Comment	Initial Value
IN	USINT	Input Value from Fieldbus	BYTE_TO_USINT(NOT_AVAILABLE_B)

Data Outputs

Name	Type	Comment	Initial Value
OUT	USINT	Filtered Output Value	16#00
VALID	BOOL	TRUE if signal is valid	FALSE

Adapter

None.

Functionality

When a REQ event arrives, the algorithm is executed. The incoming value IN is compared to the constant threshold `VALID_SIGNAL_B` (interpreted as USINT). If `IN <= BYTE_TO_USINT(VALID_SIGNAL_B)` is true, the input value is directly transferred to the output OUT, and the validity signal VALID is set to TRUE. Otherwise, OUT is set to 0 and VALID to FALSE. The result is acknowledged with the CNF event.

The two constants `NOT_AVAILABLE_B` and `VALID_SIGNAL_B` are part of the package `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL` and define a reference value for "not available" and the upper limit for valid signals, respectively.

Technical Features

- The initial value of the IN input is set to `BYTE_TO_USINT(NOT_AVAILABLE_B)`, so that on the first call without a valid value, an invalid state is immediately signaled.
- The comparison uses the function `BYTE_TO_USINT` to convert the constant's byte value to the USINT data type.
- The function block is licensed under the Eclipse Public License 2.0.

State Overview

The function block has a single state, `REQ`. As soon as a REQ event arrives, the algorithm is executed and the CNF event is immediately sent. There are no internal state changes or delays.

Application Scenarios

- **Fieldbus Signal Conditioning**: A sensor provides values in the range 0–127 as valid signals. The VALID_SIGNAL_B threshold is set to 127. Values above this threshold (e.g., 255 as an error code) are recognized as invalid.
- **Initialization**: At system startup, the input is preset to the "not available" value, so that until the first valid signal is received, the output remains at 0 and VALID is set to FALSE.
- **Error Detection**: Any permissible value range can be defined by adjusting the threshold.

Comparison with Similar Function Blocks

Similar function blocks such as `FIELDBUS_BYTE_TO_SIGNAL` or `FIELDBUS_SINT_TO_SIGNAL` operate on the same principle, but with different data types. This function block specializes in USINT. A simple "mirror" function block without validation would pass the value through unfiltered.

Conclusion

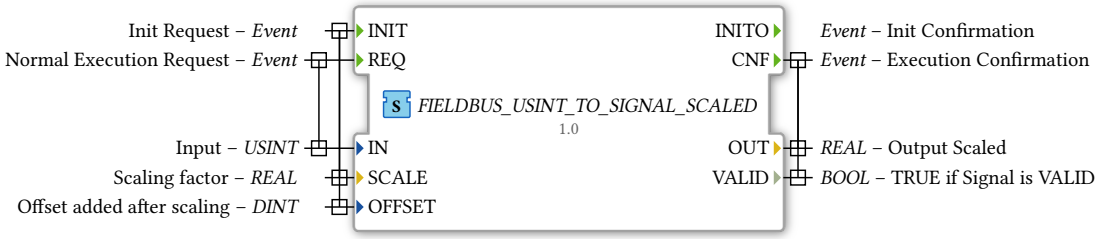
The `FIELDBUS_USINT_TO_SIGNAL` function block offers a simple and reliable way to check the validity of USINT fieldbus signals and only pass valid values on to subsequent processing. Its use of configurable limits (via package constants) makes it highly versatile.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

FIELDBUS_USINT_TO_SIGNAL_SCALED



• – ○ • – ○ • – ○ •

Introduction

The function block **FIELDBUS_USINT_TO_SIGNAL_SCALED** converts an incoming USINT signal into a scaled REAL value, taking a validity check into account. It is typically used in fieldbus signal processing

to apply a scaling factor and an offset to raw fieldbus values (e.g., sensor data) and simultaneously evaluate the validity of the input signal. The block is structured according to the IEC 61499-1 standard.

Interface Structure

Event Inputs

Event	Type	Description
`INIT`	EInit	Initialization Request; linked to the parameters `SCALE` and `OFFSET`.
`REQ`	Event	Normal execution request; linked to the data input `IN`.

Event Outputs

Event	Type	Description
`INITO`	EInit	Initialization confirmation.
`CNF`	Event	Execution confirmation; linked to the outputs `OUT` and `VALID`.

Data Inputs

Name	Data Type	Initial Value	Description
`IN`	USINT	`BYTE_TO_USINT(NOT_AVAILABLE_B)`	Input signal as an unsigned 8-bit value. The initial value corresponds to the “not available” signal.
`SCALE`	REAL	`REAL#1.0`	Scaling factor (multiplication).
`OFFSET`	DINT	`DINT#0`	Offset added after scaling.

Data Outputs

Name	Data Type	Initial Value	Description
`OUT`	REAL	`REAL#0.0`	Scaled output value.
`VALID`	BOOL	`FALSE`	Validation flag: `TRUE`, if the input signal is valid.

Adapter

None available.

Functionality

The function block operates with two states: `INIT` and `REQ`.

- On a `INIT` event, the parameters `SCALE` and `OFFSET` are taken (the algorithm is empty, but the values are read in by the `With` binding). Then, `INITO` is output.
- On a `REQ` event, the algorithm `REQ` is executed:

If the input value `IN` is less than or equal to the predefined threshold `BYTE_TO_USINT(VALID_SIGNAL_B)`, the signal is considered valid. In this case, the value is scaled as follows:
OUT := USINT_TO_REAL(IN) * SCALE + DINT_TO_REAL(OFFSET) and `VALID` is set to `TRUE`.
If `IN` exceeds the threshold (i.e., the signal is invalid), `OUT` is set to `0.0` and `VALID` to `FALSE`.
This behavior mirrors the input 1:1 to the output, provided the signal is valid – with additional scaling and offset.

Technical Details

- The constants `NOT_AVAILABLE_B` and `VALID_SIGNAL_B` come from the external package `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`. They define the boundaries between valid and invalid USINT values. `BYTE_TO_USINT(NOT_AVAILABLE_B)` is the initial state of the input and indicates an “unavailable” signal.

- The `INIT` algorithm contains no ST code – initialization is limited to providing the parameters via event binding. The function block expects a `INIT` call before the first `REQ` to set the scaling and offset.
- Scaling is always performed as REAL arithmetic; therefore, the offset is converted from DINT to REAL.

State Overview

The function block has two elementary states as simple FBs:

State	Action	Output Event
`INIT`	Execute algorithm `INIT` (empty)	`INITO`
`REQ`	Execute algorithm `REQ`	`CNF`

There are no further wait states – each incoming event immediately triggers the processing of the corresponding algorithm and the output of the associated result.

Application Scenarios

- **Fieldbus Sensor Evaluation**: A sensor delivers a raw USINT value (e.g., 0…200). The physical quantity (e.g., pressure in bar) is calculated from this using `SCALE` and `OFFSET`. The limit value `VALID_SIGNAL_B` separates valid measured values from error or initialization values.
- **Process Variable Update**: In PLC or control applications, the function block is triggered cyclically with `REQ` to process the latest fieldbus data and incorporate it into the control context.

Comparison with Similar Function Blocks

- **`FIELDBUS_USINT_TO_SIGNAL`**: A simpler function block without scaling or offset; it outputs the USINT value directly as REAL or as another data type.
- **`FIELDBUS_WORD_TO_SIGNAL_SCALED`**: Analog function block for WORD inputs – offers higher resolution but requires a different data type.

`FIELDBUS_USINT_TO_SIGNAL_SCALED` is characterized by the combination of validity checking (based on the constant `VALID_SIGNAL_B`) and simple linear scaling (multiplication + addition).

Conclusion

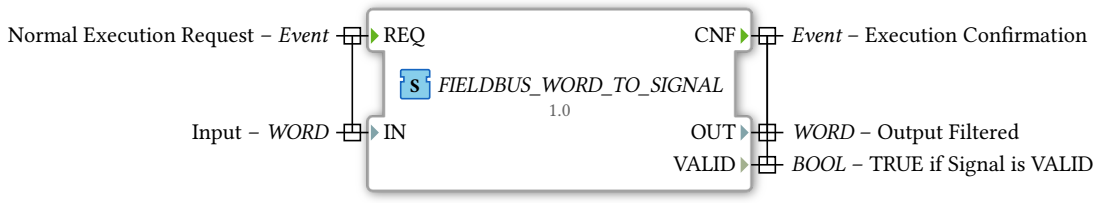
The function block `FIELDBUS_USINT_TO_SIGNAL_SCALED` is a compact yet powerful component for processing 8-bit fieldbus signals. It combines validity detection, scaling, and offset in a single logical unit and is particularly suitable for applications where raw values need to be converted into physical quantities and their plausibility checked simultaneously. Its simple state machine and clearly defined interfaces make it robust and easy to use.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

FIELDBUS_WORD_TO_SIGNAL



• – ◦ • – ◦ • – ◦ •

Introduction

The FIELDBUS_WORD_TO_SIGNAL function block is used for simple signal filtering in fieldbus applications. It reflects an incoming WORD value at the output, provided the signal is classified as valid. A Boolean output indicates the validity status. Implementation is based on a comparison with an externally defined limit constant.

Interface Structure

Event Inputs

Name	Type	Comment
REQ	Event	Normal execution command; triggered by the IN input.

Event Outputs

Name	Type	Comment
CNF	Event	Confirmation of execution; output after processing is complete.

Data Inputs

Name	Type	Initial Value	Comment
IN	WORD	NOT_AVAILABLE_WM	Input value that is checked for validity.

Data Outputs

Name	Type	Initial Value	Comment
OUT	WORD	16#0000	Filtered output value (only if the signal is valid).
VALID	BOOL	FALSE	TRUE if the input signal is recognized as valid.

Adapter

No adapters available.

Functionality

The module operates using a simple algorithm:

1. Upon arrival of the **REQ** event, the incoming value **IN** is read.
2. The value is converted to an unsigned integer using `WORD_TO_UINT`.
3. This is compared to the external constant `VALID_SIGNAL_W`.

– If the converted value is **less than or equal to** `VALID_SIGNAL_W`, **IN** is directly passed to **OUT** and **VALID** is set to `TRUE`.
Otherwise, **OUT** is reset to `16#0000` and **VALID** to `FALSE`.

1. The **CNF** event is then triggered.

The initial value of input `IN` is `NOT_AVAILABLE_WM`, so the function block signals an invalid signal in its default state.

Technical Features

- **External Constants:** The comparison limit `VALID_SIGNAL_W` and the default unavailable value `NOT_AVAILABLE_WM` are imported from the package `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`. Their values are defined outside the function block.
- **Data Type Conversion:** The comparison is performed after converting `WORD` to `UINT` (`WORD_TO_UINT`) to enable an unsigned numerical comparison.
- **Initial Values:** The output `OUT` starts with `16#0000`, and `VALID` with `FALSE`. The input `IN` starts with the invalid value `NOT_AVAILABLE_WM`.
- **Simple State Machine:** There is only one execution state (REQ) – the function block is event-driven and does not remain in any other states.

Initial State Machine:

State Overview

The function block has exactly one state:

State	Description
REQ	Executes the algorithm, sets the outputs accordingly, and sends the confirmation event **CNF**.

There are no loops or branches in the state machine; each call to REQ is executed only once.

Application Scenarios

- **Fieldbus Signal Filtering:** Checks whether a WORD value originating from a fieldbus is within a valid range (e.g., sensor values, control commands).
- **Plausibility Check:** Suppresses values marked as "unavailable" or "erroneous" (represented by `NOT_AVAILABLE_WM`).
- **Data Consistency:** Used as a simple gate in industrial controllers, passing only valid data to subsequent logic.

Comparison with Similar Function Blocks

Similar function blocks such as **FIELDBUS_SIGNAL_FILTER** or **VALUE_CHECK** also perform comparisons using threshold values. The advantage of `FIELDBUS_WORD_TO_SIGNAL` lies in the direct integration of the validity identifier (`VALID`) and the use of predefined constants from the fieldbus library, which enables standardized handling of "unavailable" states. Unlike generic multiplex blocks, the logic here is optimized for specific use in the fieldbus context.

Conclusion

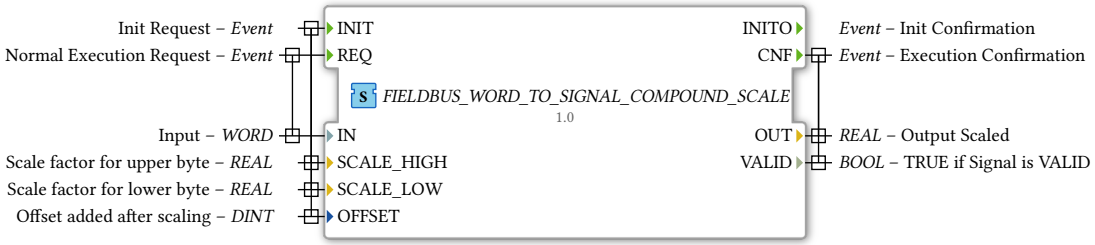
The function block `FIELDBUS_WORD_TO_SIGNAL` offers a compact and reliable way to check fieldbus signals for validity and forward only values recognized as valid. Using imported constants ensures a consistent definition of validity limits and unavailable values across the entire project. This component is particularly suitable for safety-critical applications where invalid data must be detected and suppressed.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE



• – ◦ • – ◦ • – ◦ •

Introduction

This function block maps a 16-bit word input to a scaled real value. The incoming word is first checked for validity. If the signal is valid, the upper and lower bytes are multiplied by their respective scaling factors and added with an offset. The result is output as the scale of the original fieldbus signal.

Interface Structure

Event Inputs

- **INIT** (EInit) – Initialization request; with the parameters `SCALE_HIGH`, `SCALE_LOW`, and `OFFSET`.
- **REQ** (Event) – Normal execution request; with the parameter `IN`.

Event Outputs

- **INITO** (EInit) – Confirmation of initialization.
- **CNF** (Event) – Confirmation of execution; with the parameters `OUT` and `VALID`.

Data Inputs

Name	Type	Initial Value	Description
IN	WORD	NOT_AVAILABLE_WM	The 16-bit fieldbus signal to be processed.
SCALE_HIGH	REAL	0.256	Scaling factor for the upper byte (high byte).
SCALE_LOW	REAL	0.001	Scaling factor for the lower byte (low byte).
OFFSET	DINT	0	Integer offset added after scaling.

Data Outputs

Name	Type	Initial Value	Description
OUT	REAL	0.0	Calculated scalar output value.
VALID	BOOL	FALSE	Indicates whether the input signal is valid (TRUE) or not (FALSE).

Adapters

No adapters available.

Functionality

The function block operates in two steps, controlled by events:

1. **Initialization (INIT)**

- The internal algorithm is empty; only the event output INIT0 is set.
- The scaling parameters (SCALE_HIGH, SCALE_LOW, OFFSET) are passed during the initialization event but are not used until the **REQ** cycle.

1. **Processing (REQ)**

- The incoming value IN is compared to the external constant VALID_SIGNAL_W.
- **Valid Signal** (IN ≤ VALID_SIGNAL_W):

1. The upper byte is extracted by right-shifting 8 bits and masking with 0x00FF.
2. The lower byte is obtained by masking with 0x00FF.
3. Both bytes are converted to REAL, multiplied by the corresponding scaling factors, and added to the offset:

OUT = (highByte * SCALE_HIGH) + (lowByte * SCALE_LOW) + OFFSET`

1. VALID is set to TRUE.

- **Invalid signal** (otherwise):
- OUT is set to 0.0.
- VALID is set to FALSE.

The following ST code illustrates the internal logic:
IF (WORD_TO_UINT(IN) <= WORD_TO_UINT(VALID_SIGNAL_W)) THEN temp := SHR(IN, SINT#8); highByte := temp AND WORD#16#00FF; lowByte := IN AND WORD#16#00FF; OUT := UINT_TO_REAL(WORD_TO_UINT(highByte)) * SCALE_HIGH

- UINT_TO_REAL(WORD_TO_UINT(lowByte)) * SCALE_LOW
- DINT_TO_REAL(OFFSET); VALID := BOOL#TRUE; ELSE OUT := REAL#0.0; VALID := BOOL#FALSE; END_IF;

Technical Features

- **Compound Scaling:** The two bytes of a word are scaled separately. This allows for the representation of measured values whose information is distributed across two bytes (e.g., different resolutions or units).
- **Fieldbus Constants:** The two external constants VALID_SIGNAL_W and NOT_AVAILABLE_WM define the valid value range and the "not available" state, respectively. They are taken from the package eclipse4diac::signalprocessing::FIELDBUS_SIGNAL.

- **Initial Values:** The default values `SCALE_HIGH = 0.256`, `SCALE_LOW = 0.001`, and `OFFSET = 0` are field-tested values for typical fieldbus signals (e.g., temperature measurement).
- **No State Dependency:** The component has no internal state machines beyond INIT/REQ – the response is purely combinatorial during the REQ event.

State Overview

The component has two simple states that directly correspond to the events:

State	Triggered by	Action performed	Output event
`INIT`	Event `INIT`	Algorithm `INIT` (empty)	`INITO`
`REQ`	Event `REQ`	Algorithm `REQ` (Scaling)	`CNF`

No further states or branches exist.

Application Scenarios

- **Temperature Measurement via Two Channels** – The high byte represents the integer part, the low byte the decimal part of a temperature (e.g., 0.256°C/LSB for high, 0.001°C/LSB for low).
- **Pressure or Level Sensors** – Provide a 16-bit raw value, which is decomposed into two differently scaled parts to determine the physical value.
- **Fieldbus Data Preprocessing** – In a logiBUS signal processing chain, this component can be used as the first stage for converting word values into real physical quantities.

Comparison with Similar Function Blocks

- **FIELDBUS_WORD_TO_SIGNAL** (without compound) – Applys only a single scaling factor to the entire word. Suitable when the high and low bytes have the same weight.
- **FIELDBUS_BYTE_TO_SIGNAL** – Processes only a single byte. Requires two instances to handle both bytes of a word.
- **FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE** is characterized by the separate scaling of the two bytes, which enables a more flexible and accurate representation of many sensor protocols.

Conclusion

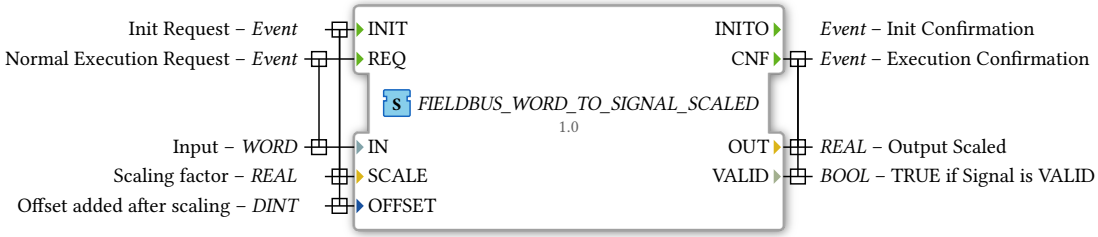
The **FIELDBUS_WORD_TO_SIGNAL_COMPOUND_SCALE** function block is a specialized tool for processing 16-bit fieldbus signals where the two bytes have different physical meanings or resolutions. The clear separation of validation, byte extraction, and compound scaling makes it a reliable and configurable element in the signal processing chain. Externally defined fieldbus constants allow for easy adaptation to various protocols.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

FIELDBUS_WORD_TO_SIGNAL_SCALED



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **FIELDBUS_WORD_TO_SIGNAL_SCALED** is used to convert a digital fieldbus raw value (16-bit WORD) into a scaled physical signal value (REAL). A valid output is only provided if the

incoming value is recognized as a valid signal. The block checks the validity against a predefined constant (VALID_SIGNAL_W) and can reliably suppress unsafe or "unavailable" values.

Interface Structure

Event Inputs

Event	Description
INIT	Initialization request; sets the scaling factors (SCALE, OFFSET).
REQ	Normal processing request; evaluates the current input IN.

Event Outputs

Event	Description
INITO	Initialization confirmation.
CNF	Processing confirmation; signals that outputs OUT and VALID have been updated.

Data Inputs

Name	Data Type	Default Value	Description
IN	WORD	NOT_AVAILABLE_WM	Raw value from the fieldbus (e.g., 0…65535). The constant NOT_AVAILABLE_WM indicates an invalid starting value.
SCALE	REAL	1.0	Scaling factor (multiplied by the read value).
OFFSET	DINT	0	Integer offset added after scaling (converted to REAL).

Data Outputs

Name	Data Type	Default Value	Description
OUT	REAL	0.0	Scaled signal value. Valid only if VALID = TRUE.
VALID	BOOL	FALSE	Validation flag – TRUE, if the incoming raw value is recognized as a valid signal.

Adapter

No adapters available.

Functionality

The module operates in a state-controlled manner:

1. **INIT State**:

After the INIT event occurs, the initialization algorithm (empty) is executed and the INITO event is output. The scaling factors are then applied to the inputs SCALE and OFFSET (the values are already present at the inputs).

1. **REQ State** (Main Logic):

For each REQ event, the following algorithm is executed:

- The incoming IN (WORD) is first converted to UINT.
- If this value is **less than or equal to** the threshold defined in the global constant VALID_SIGNAL_W (also as WORD/UINT), the signal is considered **valid**.
- If valid:

OUT = UINT_TO_REAL(IN) * SCALE + DINT_TO_REAL(OFFSET)`
  VALID = TRUE`

- If invalid:

OUT = 0.0`
VALID = FALSE`

- Subsequently, the `CNF` event is triggered, which accompanies the updated outputs `OUT` and `VALID`.

The comparison value `VALID_SIGNAL_W` and the default input value `NOT_AVAILABLE_WM` are imported as constants in the namespace `eclipse4diac::signalprocessing::FIELDBUS_SIGNAL`.

Technical Features

- **Use of External Constants**: The validity threshold (`VALID_SIGNAL_W`) and the "Not Available" marker (`NOT_AVAILABLE_WM`) are retrieved from a global library. This enables consistent signal definition across multiple FB instances.
- **Scaling with REAL and DINT**: The offset is defined as `DINT`, but is converted to a floating-point number by `DINT_TO_REAL`. This can lead to minor rounding errors with large values, but is usually negligible in practice.
- **Validation Logic**: Unlike simple "word-to-real" converters, the value is only output if it lies within the defined valid range. This prevents misinterpretations of errors or invalid telegrams.
- **Initial Behavior**: The inputs `IN` have a default value of `NOT_AVAILABLE_WM`, so the function block immediately reports "invalid" after a reset or in case of no communication until a valid value is received.

State Overview

State	Content	Action	Output Event
`INIT`	Initialization (empty algorithm)	–	`INITO`
`REQ`	Signal Processing	Calculation of `OUT` and `VALID`	`CNF`

The function block (FB) has no further states; after each event, it returns to the corresponding start state. There is no explicit error handling or timeouts.

Application Scenarios

- **Agricultural Technology / Agricultural Control Systems**: A fieldbus device (e.g., ISOBUS) provides raw values for sensors (speed, temperature, pressure). The FB scales these to physical units and detects invalid measurement ranges.
- **Industrial Automation**: With Profibus/IO-Link connectivity, word codes can be converted into process-relevant quantities – e.g., 0…1000 ⟶ 0.0…10.0 V.
- **Diagnostic and Safety Functions**: Validation checks allow for easy detection of signal losses or range exceedances and the initiation of an error response.

Comparison with Similar Function Blocks

Function Block	Differences
FIELDBUS_WORD_TO_SIGNAL (without scaling)	Maps the raw value 1:1, without multiplication/addition.
WORD_TO_REAL (standard IEC function block)	Pure type conversion without validation or scaling.
SCALING_FB	Usually with separate validation and scaling branches; this function block combines both in one step.

The strength of `FIELDBUS_WORD_TO_SIGNAL_SCALED` lies in the **combination of validation, scaling, and offset** in a single, manageable function block.

Conclusion

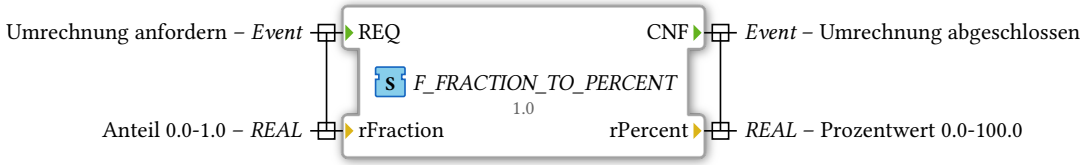
This function block offers a compact and robust solution for processing fieldbus raw data. It integrates the typical task chain "Check ⟶ Scale ⟶ Offset" and, thanks to its fixed validity threshold, ensures a clear separation between normal measured values and error signals. The use of external constants allows for flexible adaptation without complicating the internal logic. Ideal for safety-critical or diagnostic applications in automation technology.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

F_FRACTION_TO_PERCENT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **F_FRACTION_TO_PERCENT** is the counterpart to `F_PERCENT_TO_FRACTION`: it converts a fraction (REAL 0.0–1.0), such as one delivered by `MyLib::sys::F_PWM_RAW_TO_PERCENT`, into a percent value (REAL 0.0–100.0), e.g. for sending to a web client via OPC-UA publish.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Request conversion; processes the current value of `rFraction`.

Event Outputs

Event	Data Type	Comment
CNF	Event	Conversion complete; `rPercent` has been updated.

Data Inputs

Name	Data Type	Comment
rFraction	REAL	Fraction 0.0-1.0.

Data Outputs

Name	Data Type	Comment
rPercent	REAL	Percent value 0.0-100.0.

Adapter

None.

Functionality

On each `REQ` event, the `rFraction` input is multiplied by 100 without clamping:
`rPercent := rFraction * 100.0;`
The `CNF` event is then emitted.

Technical Details

- Implemented as a **SimpleFB** with a single execution state (`REQ`) — no multi-step state machine.
- Unlike its counterpart `F_PERCENT_TO_FRACTION`, it does **not clamp** the input value: since `rFraction` is already bounded to 0.0-1.0 by `RampLimitFS/F_PWM_RAW_TO_PERCENT` earlier in the PWM chain, an additional range check is not needed here.

- Deliberately kept separate from `MyLib::sys::F_PWM_RAW_TO_PERCENT`, which converts the fieldbus raw value (not percent!) into a fraction — together, the two blocks form the complete fieldbus-raw-value → fraction → percent chain.
- License: Eclipse Public License 2.0.

State Overview

State	Action	Output Event
REQ	Executes the **REQ** algorithm (multiplication by 100)	CNF

There are no further waiting or initialization states.

Application Scenarios

- Converting an internally processed fraction (0.0-1.0) back into the percent value expected by a web client via OPC-UA publish — e.g. in the PWM training example `RampLimitFS_TO_logiBUS_QDA_PWM_OPC`.
- In general: any interface that computes internally with a normalized 0.0-1.0 fraction while communicating externally in percent.

Comparison with Similar Blocks

- Its counterpart is `F_PERCENT_TO_FRACTION` (percent → fraction, for the forward direction, including clamping).
- The simplest conceivable scale converter with no range check at all, since the input is already guaranteed to arrive normalized.

Conclusion

F_FRACTION_TO_PERCENT is a pure multiplication by 100 that brings the internally used 0.0-1.0 fraction to the externally communicated 0.0-100.0 percent value — without any error handling of its own, since the input is already pre-normalized.

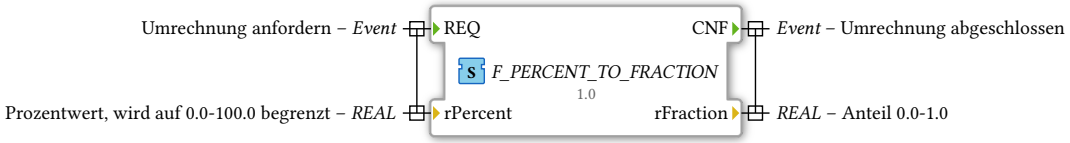
🛠️ Related Exercises

- RampLimitFS_TO_logiBUS_QDA_PWM_OPC
- InputOutputTesterButton_PWM_OPC_UA

🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

F_PERCENT_TO_FRACTION



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **F_PERCENT_TO_FRACTION** converts a percent value (REAL 0.0–100.0), such as one received via OPC-UA subscribe from a web client, into a fraction (REAL 0.0–1.0) as expected as input by `MyLib::sys::F_PWM_PERCENT_TO_RAW`. Values outside the valid range are clamped instead of producing an error.

Interface Structure

Event Inputs

Event	Data Type	Comment

REQ	Event	Request conversion; processes the current value of `rPercent`.

Event Outputs

Event	Data Type	Comment
CNF	Event	Conversion complete; `rFraction` has been updated.

Data Inputs

Name	Data Type	Comment
rPercent	REAL	Percent value, clamped to 0.0-100.0.

Data Outputs

Name	Data Type	Comment
rFraction	REAL	Fraction 0.0-1.0.

Adapter

None.

Functionality

On each `REQ` event, the `rPercent` input is checked and clamped to the valid range:

```
IF rPercent <= 0.0 THEN
    rFraction := 0.0;
ELSIF rPercent >= 100.0 THEN
    rFraction := 1.0;
ELSE
    rFraction := rPercent / 100.0;
END_IF;
```

- Values `<= 0.0` yield `rFraction = 0.0`, values `>= 100.0` yield `rFraction = 1.0`.
- All other values are divided linearly by 100.
- The `CNF` event is then emitted.

Technical Details

- Implemented as a **SimpleFB** with a single execution state (`REQ`) — no multi-step state machine.
- Clamps instead of error-handling: implausible percent values (e.g. from faulty web input) do not trigger an error state but are silently limited to 0-100 %.
- Deliberately kept separate from `MyLib::sys::F_PWM_PERCENT_TO_RAW`, which converts a fraction (not percent!) into the fieldbus raw value — together, the two blocks form the complete percent → fraction → fieldbus-raw-value chain.
- License: Eclipse Public License 2.0.

State Overview

State	Action	Output Event
REQ	Executes the **REQ** algorithm (clamping + division by 100)	CNF

There are no further waiting or initialization states.

Application Scenarios

- Converting a web setpoint received via OPC-UA subscribe (percent REAL) into the fraction expected by internal fieldbus conversion blocks — e.g. in the PWM training example `RampLimitFS_TO_logiBUS_QDA_PWM_OPC`.
- In general: any interface that communicates externally in percent while computing internally with a normalized 0.0-1.0 fraction.

Comparison with Similar Blocks

- Its counterpart is `F_FRACTION_TO_PERCENT` (fraction ⟶ percent, for the reverse direction).
- Unlike the `FIELDBUS_*_TO_SIGNAL` blocks (e.g. `FIELDBUS_BYTE_TO_SIGNAL`), this block does not validate against a constant but performs a pure, clamping scale conversion.

Conclusion

F_PERCENT_TO_FRACTION is a minimal, clamping scale converter that bridges the gap between the externally communicated percent value and the internally used 0.0-1.0 fraction — without any error handling of its own, since out-of-range values are simply clamped.

🛠️ Related Exercises

- RampLimitFS_TO_logiBUS_QDA_PWM_OPC
- InputOutputTesterButton_PWM_OPC_UA

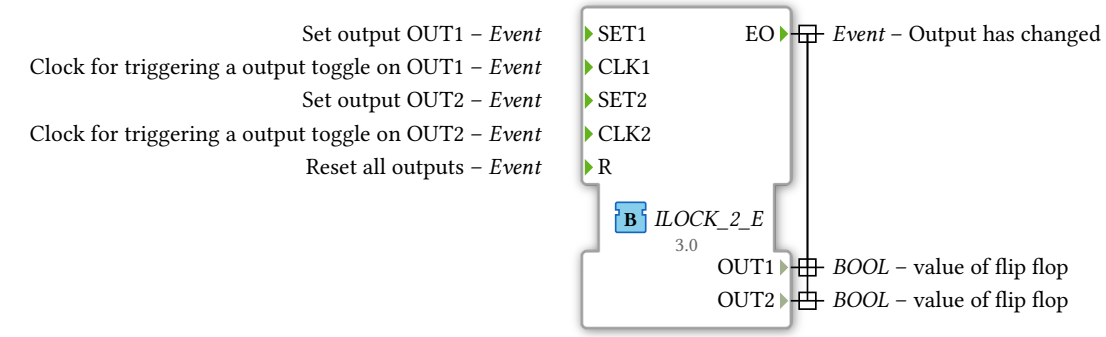
🌐 Related topic pages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numeric Index (signal processing)

Here you will find all exercises from the `signalprocessing` workspace, listed in ascending order.

ILOCK_2_E



ECC: ILOCK_2_E



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ILOCK_2_E** is an event-driven bistable toggle block with dual interlock. It is used to control two mutually exclusive outputs (OUT1, OUT2). In addition to explicit set and reset functions, the block offers an edge-triggered toggle function that allows a single toggle per activation. The block is particularly suitable for control logics where both outputs must never be TRUE simultaneously and a repeated toggle command only takes effect after being released again (e.g., by setting the other channel or resetting).

Interface Structure

Event Inputs

Event	Comment
SET1	Sets OUT1 to TRUE and OUT2 to FALSE.
CLK1`	Toggles OUT1 (upon initial activation after a set/reset) and sets OUT2 to FALSE.
SET2`	Sets OUT2 to TRUE and OUT1 to FALSE.
CLK2`	Toggles OUT2 and sets OUT1 to FALSE.
R`	Resets both outputs to FALSE.

Event Outputs

Event	Comment
EO`	Triggered after every output change (SET, TOGGLE, or RESET). Stores the current values of OUT1 and OUT2.

Data Inputs

None (the function block is purely event-driven).

Data Outputs

Output	Type	Comment
OUT1	BOOL	First output (bistable value).
OUT2	BOOL	Second output (bistable value).

Adapters

None.

Functionality

The function block has six states: the initial idle state **STOP** and five action states (**SET1**, **SET2**, **TOGGLE1**, **TOGGLE2**, **RESET**). Upon the occurrence of an event, the block immediately switches to the corresponding state, executes the associated algorithm, and then outputs the event EO. Afterward, the block immediately returns to the **STOP** state (condition 1 on all transitions).

The internal logic provides **dual interlocking**: When one output is set, the other is forcibly set to FALSE. Similarly, when one output is toggled, the other output is reset.

The **toggle function** is edge-triggered: The internal variables EDGE1 and EDGE2 store whether a toggle has already been executed on the corresponding channel. Only after a reset or a set of the other channel (EDGE = FALSE) is the toggle actually executed and the flag set to TRUE. Repeated CLK events without an intermediate set/reset of the same or the other channel are ignored.

Technical Features

- **Edge Detection for Toggle** – The internal flags EDGE1/EDGE2 prevent multiple toggling events with the same enable signal.

- **Always Exclusive Outputs** – Each algorithm ensures that both outputs are never TRUE simultaneously.
- **No Data Inputs** – All control is exclusively event-based, enabling easy integration into event-driven automation systems.
- **Robust Reset Behavior** – A reset clears both the outputs and the internal edge flags, making the toggle function available again.

Status Overview

Status	Description	Trigger	Algorithm Effect (Simplified)	Output
STOP	Initial, waits for event	-	–	–
SET1	Set Output 1	SET1	OUT1=TRUE, OUT2=FALSE, EDGE1=FALSE, EDGE2=FALSE	E0
SET2	Set Output 2	SET2	OUT1=FALSE, OUT2=TRUE, EDGE1=FALSE, EDGE2=FALSE	E0
TOGGLE1	Toggle Output 1	CLK1	If EDGE1=FALSE: OUT1 = NOT OUT1; OUT2=FALSE; EDGE1=TRUE; EDGE2=FALSE	E0
TOGGLE2	Toggle Output 2	CLK2	OUT1=FALSE; if EDGE2=FALSE: OUT2 = NOT OUT2; EDGE1=FALSE; EDGE2=TRUE	E0
RESET	Reset all outputs	R	OUT1=FALSE, OUT2=FALSE, EDGE1=FALSE, EDGE2=FALSE	E0

All action states immediately revert to STOP after the algorithm is executed.

Application Scenarios

- **Direction Control** (e.g., motor left/right) – both directions are never active simultaneously.
- **Pump Control with Priority** – two pumps, but only one may run at a time.
- **Mode Switching with Single Toggle** – e.g., "Start/Stop" behavior, where a repeated toggle command only takes effect after a reset or other action.
- **Interlocked Switching States in Safety Controllers** – guaranteed exclusivity of the outputs.

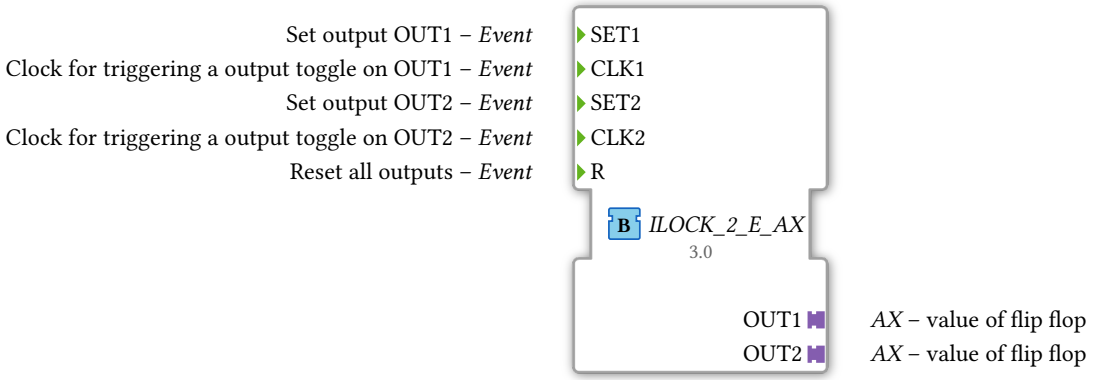
Comparison with Similar Function Blocks

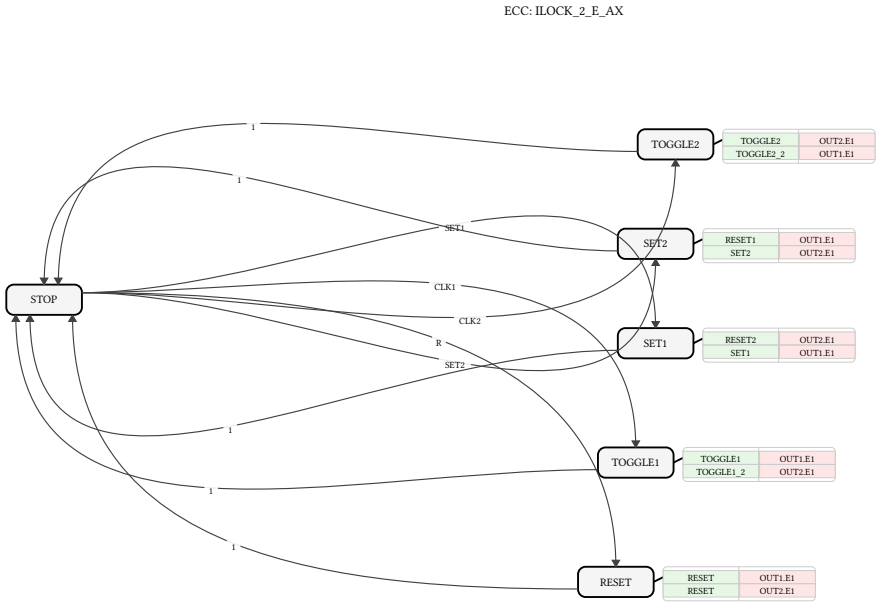
Function Block	Properties
E_SR	Simple set/reset flip-flop, no toggle, no dual locking.
SR-Flipflop (two outputs)	Often only combinatorial or without edge detection.
ILOCK_2_E	Offers both: **Set** (exclusive) and **edge-triggered toggle** with automatic locking. Ideal for advanced requirements.

Conclusion

The ILOCK_2_E function block is a powerful function block for applications with two mutually exclusive states. The combination of explicit setting, edge-triggered toggle, and robust reset behavior makes it particularly suitable for control logics that require a clear separation of outputs and a defined response to repeated commands. Integration into event-based automation systems (e.g., IEC 61499) is achieved through the purely event-driven interface and the unified output E0.

ILOCK_2_E_AX





• – ○ • – ○ • – ○ •

Introduction

The function block **ILOCK_2_E_AX** provides two bistable outputs with mutual interlock and a toggle function. It is event-driven and outputs its states via two adapters (type `AX`). This block is suitable for applications requiring two mutually exclusive signals, such as controlling switches, valves, or motors.

Interface Structure

Event Inputs

Name	Description
`SET1`	Sets output `OUT1` to TRUE and `OUT2` to FALSE
`CLK1`	Toggles the state of `OUT1` if it is not already toggled
`SET2`	Sets output `OUT2` to TRUE and `OUT1` to FALSE
`CLK2`	Toggles the state of `OUT2` if it is not already toggled

R	Sets both outputs to FALSE (Reset)

Event Outputs

The function block has no dedicated event outputs. Output signals are transmitted exclusively via the **adapters** (see below).

Data Inputs

None.

Data Outputs

No direct data outputs. The current states of the flip-flops can be retrieved via the **adapters** (see below).

Adapters

Name	Type	Description
OUT1	AX	Provides the state of the first flip-flop (data output D1) and acknowledges changes via event output E1
OUT2	AX	Provides the state of the second flip-flop (data output D1) and acknowledges changes via event output E1

Functionality

This function block operates in an event-driven manner. An incoming event triggers a state change, which is immediately output to the corresponding adapters.

- **SET1**: Sets `OUT1.D1 = TRUE` and `OUT2.D1 = FALSE`. The internal edge buffers (`EDGE1`, `EDGE2`) are reset.
- **SET2**: Sets `OUT1.D1 = FALSE` and `OUT2.D1 = TRUE`. The edge buffers are reset.
- **CLK1**: Toggles `OUT1.D1` (if not already toggled by a previous `CLK1` event) and sets `OUT2.D1 = FALSE`. The edge buffer `EDGE1` prevents repeated toggling on the same event.
- **CLK2**: Toggles `OUT2.D1` (edge buffer `EDGE2`) and sets `OUT1.D1 = FALSE`.
- **R**: Sets both outputs to FALSE and clears the internal edge buffers.

The interlock ensures that only one of the two outputs can be TRUE at any given time. When one output is toggled, the other is always set to FALSE.

Technical Features

- The module uses **adapters** (type `AX`) as outputs. This allows for flexible connection to other modules that support the same adapter.
- An **internal edge detection** (variables `EDGE1`, `EDGE2`) prevents multiple toggling on a single event. The edge buffer is only reset after a reset or an explicit setting, allowing for further toggling.
- All algorithms are executed **atomically** in the ECC states; the module outputs an event via the adapters (`OUT1.E1` or `OUT2.E1`) after each action.
- The initial states of all outputs are **FALSE**.

State Overview

The following states underlie the ECC (Executive Control Cycle):

- **STOP** – Wait state (initial)
- **SET1** – Sets channel 1
- **SET2** – Sets channel 2
- **TOGGLE1** – Toggles channel 1, disables channel 2
- **TOGGLE2** – Toggles channel 2, disables channel 1
- **RESET** – Resets both channels

The transitions between states are triggered by the received events (`SET1`, `SET2`, `CLK1`, `CLK2`, `R`) and automatically return to state `STOP` after the actions are executed (transition condition). 1).

Application Scenarios

- **Alternating operation** of two consumers, e.g., feed/return valves, where both must never be active simultaneously.

- **Clock-controlled switching** between two states (e.g., direction change of a conveyor belt) using the toggle inputs.
- **Central reset function** for a group of bistable actuators via the `R` input.

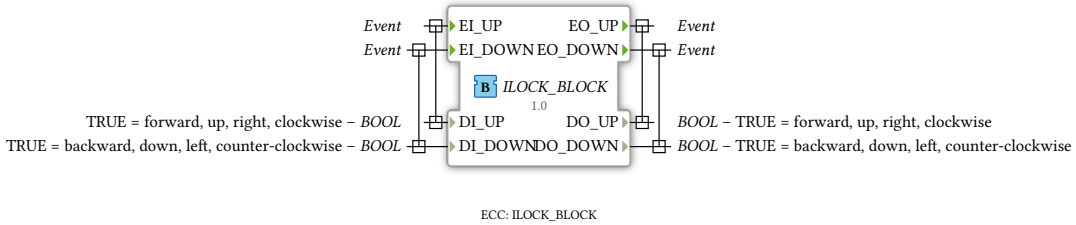
Comparison with Similar Function Blocks

Compared to a simple set-reset flip-flop (e.g., `E_SR`), **ILOCK_2_E_AX** offers two outputs with integrated interlocking and a toggle function. While `E_SR` only switches a single output, this function block can control two coupled signals. The use of adapters instead of direct data ports allows for standardized and reusable coupling with other components.

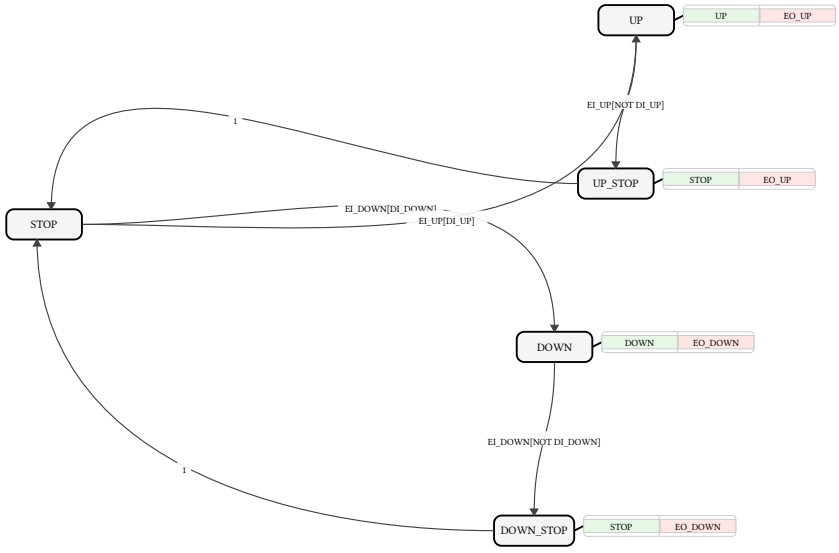
Conclusion

The **ILOCK_2_E_AX** function block is a compact, event-driven component for implementing interlocked, bistable outputs with an optional toggle function. Thanks to its adapter-based interface and the clear separation of setting, toggling, and resetting, it is particularly suitable for safety-critical or logically linked control tasks in automation technology.

ILOCK_BLOCK



ECC: ILOCK_BLOCK



• – ○ • – ○ • – ○ •

Introduction

The function block **ILOCK_BLOCK** implements an interlock between two opposing signals. It prioritizes the first arriving active signal and ignores all subsequent conflicting signals until the initial signal is released. This ensures that two opposing actions (e.g., Up/Down, Right/Left) are never activated simultaneously.

Interface Structure

Event Inputs

Event	With Variable	Description
EI_UP	DI_UP	Event to Set UP Direction
EI_DOWN	DI_DOWN	Event to set the DOWN direction

Event Outputs

Event	With Variable	Description
EO_UP	DO_UP	Triggered when the UP direction is activated or deactivated
EO_DOWN	DO_DOWN	Triggered when the DOWN direction is activated or deactivated

Data Inputs

| Variable | Type | Comment | | |---|---|---| | DI_UP | BOOL | TRUE = forward, upward, right, clockwise | | DI_DOWN | BOOL | TRUE = backward, downward, left, counterclockwise |

Data Outputs

Variable	Type	Comment
DO_UP	BOOL	TRUE = forward, upward, right, clockwise
DO_DOWN	BOOL	TRUE = backward, downward, left, counterclockwise

Adapter

None.

Functionality

The module has two activation states (UP, DOWN) and two intermediate states (UP_STOP, DOWN_STOP) for deactivation. Control is achieved exclusively via the event inputs in conjunction with the data inputs.

- **In the idle state (STOP)**, both outputs are set to FALSE.
- **UP activation:** When the event EI_UP with DI_UP = TRUE arrives, the state changes to **UP**. In this state, DO_UP = TRUE and DO_DOWN = FALSE are set, and EO_UP is output.
- **DOWN Activation:** When the event EI_DOWN with DI_DOWN = TRUE occurs, the state changes to **DOWN**. DO_UP = FALSE and DO_DOWN = TRUE are set, and EO_DOWN is output.
- **Deactivation of an Active State:**
- In the UP state, another EI_UP with DI_UP = FALSE is expected (release). The state then immediately changes back to STOP via **UP_STOP** EO_UP is triggered once more (signaling the stop).
- Similarly, in the DOWN state, a new EI_DOWN followed by DI_DOWN = FALSE is required to return to STOP via **DOWN_STOP** this outputs EO_DOWN.
- **Ignoring Conflicting Signals:** As long as the function block is active (UP or DOWN), events in the opposite direction are completely ignored (no state change). This preserves the priority of the first signal.

Technical Features

- State transitions are event-driven and instantaneous (no delays).
- Unlike a simple set/reset function block, the second input direction is not accepted during the interlock; the interlock can only be released by the original event itself.
- All outputs are set back to FALSE after a stop.

State Overview

State	Description
STOP	Idle state. Both outputs FALSE. Waiting for activation.

UP	UP direction active. DO_UP = TRUE, DO_DOWN = FALSE. Waiting for release by `EI_UP` with `DI_UP = FALSE`.
DOWN	DOWN direction active. DO_UP = FALSE, DO_DOWN = TRUE. Waiting for release by `EI_DOWN` with `DI_DOWN = FALSE`.
UP_STOP	Intermediate state after release of UP. Immediately executes the STOP algorithm, sends `EO_UP`, and switches back to STOP.
DOWN_STOP	Intermediate state after releasing DOWN. Immediately executes the STOP algorithm, sends `EO_DOWN`, and switches back to STOP.

DOWN_STOP **Transition Matrix (Simplified):**

- `STOP → UP` : `EI_UP` & `DI_UP = TRUE`
- `STOP → DOWN` : `EI_DOWN` & `DI_DOWN = TRUE`
- `UP → UP_STOP` : `EI_UP` & `DI_UP = FALSE`
- `DOWN → DOWN_STOP` : `EI_DOWN` & `DI_DOWN = FALSE`
- `UP_STOP → STOP` : always (immediately)
- `DOWN_STOP → STOP`: always (immediately)

Application Scenarios

- **Motor control (e.g., lifting platform, crane):** Prevents simultaneous travel in opposite directions.
- **Valve control:** Protects against the simultaneous opening and closing of a process valve.
- **Directional interlock in conveyor systems:** Ensures that a belt only activates one direction of rotation at a time.
- **Safety-critical controls:** Enforces a clear, prioritized signal sequence.

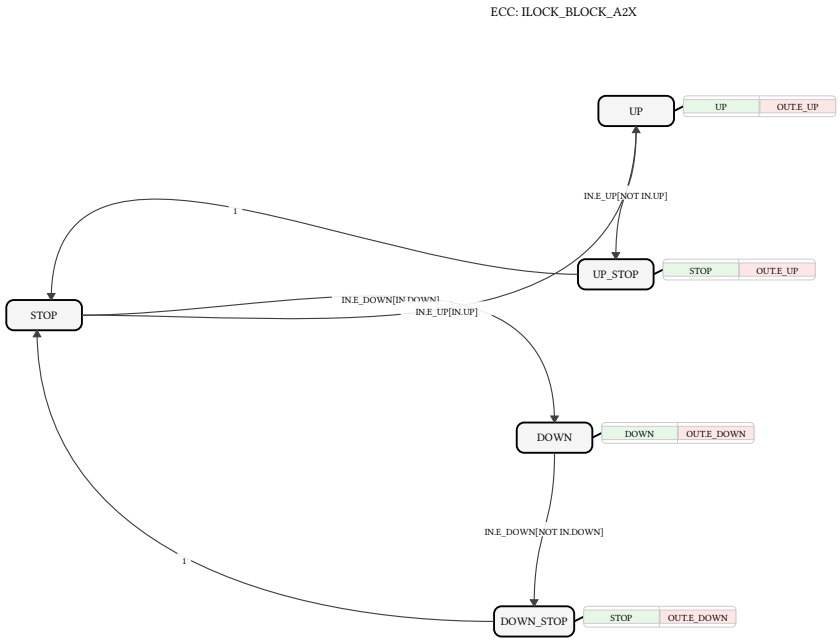
Comparison with similar function blocks

- **Set/Reset (SR/R-SR):** Allows the simultaneous setting of both directions, which can lead to undefined states. The ILOCK_BLOCK prevents this through strict interlocking.
- **State timer (e.g., with multiple states):** Offers more flexibility but requires manual implementation of the prioritization logic. The ILOCK_BLOCK directly encapsulates this logic.
- **Simple Interlock via AND Gate:** Pure signal processing ignores the temporal sequence. The ILOCK_BLOCK reacts event-driven to the first valid activation.

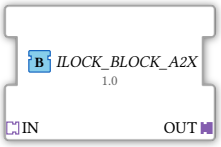
Conclusion

The **ILOCK_BLOCK** is a specialized function block for interlock applications with prioritization of the first active input. Due to its clear state machine and event-driven processing, it is particularly suitable for time-critical and safety-related control systems where conflicting signals must be strictly excluded. It offers a robust alternative to classic set/reset logic and significantly reduces implementation effort.

ILOCK_BLOCK_A2X



Input for forward/up and backward/down direction – *A2X*



A2X – Output for forward/up and backward/down direction

• – ○ • – ○ • – ○ •

Introduction

The **ILOCK_BLOCK_A2X** is a basic function block (FB) that implements an interlock mechanism for directional commands (UP/DOWN) over a unidirectional A2X adapter. It prioritizes the *first* active input event and latches that direction until the corresponding input signal is withdrawn, ensuring that only one output direction can be active at a time. This makes it suitable for safety-related control of actuators such as hoists, sliding gates, or lifting platforms.

Interface Structure

The FB communicates exclusively through a single adapter pair of type `adapter::types::unidirectional::A2X`. The input socket `IN` receives events and data from the environment, while the output plug `OUT` forwards the latched state and corresponding event triggers.

Event Inputs

The events are carried by the input adapter `IN` and carry the data elements `UP` and `DOWN` as qualifiers (per IEC 61499 WITH semantics).

Event	Associated Data	Description
`IN.E_UP`	`IN.UP`	Request/trigger for the UP direction.
`IN.E_DOWN`	`IN.DOWN`	Request/trigger for the DOWN direction.

Event Outputs

The events are emitted on the output plug `OUT` together with the corresponding state data.

Event	Associated Data	Description

OUT.E_UP	OUT.UP, OUT.DOWN	Acknowledges the UP state or the UP release.
OUT.E_DOWN	OUT.UP, OUT.DOWN	Acknowledges the DOWN state or the DOWN release.

Data Inputs

The input data are accessed through the input socket IN and are only valid when the respective event is present.

Data	Type	Description
IN.UP	BOOL	Level signal requesting the UP direction.
IN.DOWN	BOOL	Level signal requesting the DOWN direction.

Data Outputs

The output data are provided on the output plug OUT and reflect the currently latched direction.

Data	Type	Description
OUT.UP	BOOL	TRUE while the UP direction is latched.
OUT.DOWN	BOOL	TRUE while the DOWN direction is latched.

Adapters

Adapter	Direction	Type	Description
IN	Socket	adapter::types::unidirectional::A2X	Receives UP/DOWN commands and events.
OUT	Plug	adapter::types::unidirectional::A2X	Sends the latched direction and events.

Functionality

The FB is implemented as a basic FB with an ECC (Execution Control Chart). It operates in a purely event-driven manner:

1. **From STOP state**
   - If IN.E_UP occurs and IN.UP is TRUE, the FB transitions to the UP state, sets OUT.UP := TRUE / OUT.DOWN := FALSE and emits OUT.E_UP.
   - If IN.E_DOWN occurs and IN.DOWN is TRUE, the FB transitions to the DOWN state, sets OUT.UP := FALSE / OUT.DOWN := TRUE and emits OUT.E_DOWN.
2. **In UP state**
   - The FB ignores any DOWN commands. If IN.E_UP occurs again while IN.UP is FALSE (the requesting signal has been released), it transitions to UP_STOP, resets both outputs to FALSE and emits OUT.E_UP to signal the release.
3. **In DOWN state**
   - Symmetrically, if IN.E_DOWN occurs while IN.DOWN is FALSE, the FB transitions to DOWN_STOP, resets both outputs to FALSE and emits OUT.E_DOWN.
4. **Stop handshake**
   - UP_STOP and DOWN_STOP are transient states; the FB immediately returns to STOP (unconditional transition), ready for the next command.

This behaviour ensures that the first active input (UP or DOWN) is prioritised and kept latched until it is explicitly withdrawn. The outputs OUT.UP / OUT.DOWN are never TRUE simultaneously, satisfying an interlock requirement.

Technical Features

- **IEC 61499-2 compliant** basic FB with ECC formalism.

- **Adapter-based interface** – uses the unidirectional `A2X` adapter type for both input and output, allowing direct connection to a partner FB via a single plug/socket pair.
- **Structured Text (ST)** algorithms for state actions.
- **Event-qualified data** – input data are sampled together with their associated event, avoiding glitches on the input line.
- **Deterministic state machine** – five ECC states (`STOP`, `UP`, `DOWN`, `UP_STOP`, `DOWN_STOP`) ensure reproducible behaviour.
- **Latched output** – the output direction is held until an explicit release event is received.
- **EPL-2.0 licensed** – can be freely used and modified in 4diac-based projects.

State Overview

State	Description	Actions / Outputs
`STOP`	Idle state, no direction latched.	none
`UP`	UP direction is latched and active.	`OUT.UP := TRUE`, `OUT.DOWN := FALSE`, emit `OUT.E_UP`
`DOWN`	DOWN direction is latched and active.	`OUT.UP := FALSE`, `OUT.DOWN := TRUE`, emit `OUT.E_DOWN`
`UP_STOP`	UP direction is being released; both outputs reset.	`OUT.UP := FALSE`, `OUT.DOWN := FALSE`, emit `OUT.E_UP`
`DOWN_STOP`	DOWN direction is being released; both outputs reset.	`OUT.UP := FALSE`, `OUT.DOWN := FALSE`, emit `OUT.E_DOWN`

Transitions:

- `STOP` ⟶ `UP` : `IN.E_UP[IN.UP]`
- `STOP` ⟶ `DOWN` : `IN.E_DOWN[IN.DOWN]`
- `UP` ⟶ `UP_STOP` : `IN.E_UP[NOT IN.UP]`
- `DOWN` ⟶ `DOWN_STOP` : `IN.E_DOWN[NOT IN.DOWN]`
- `UP_STOP` ⟶ `STOP` : unconditional
- `DOWN_STOP` ⟶ `STOP` : unconditional

Application Scenarios

- **Actuator control** – controlling a single motor with forward/reverse (UP/DOWN) commands, e.g., roller shutters, conveyors, adjustable tables.
- **Interlocking in safety circuits** – preventing simultaneous activation of two contradictory commands (e.g., raise/lower) in manual control panels.
- **First-command priority logic** – when both directions are possible but only the first activated direction should be executed until it is released.
- **Adapter-chained signal chains** – because the FB uses unidirectional A2X adapters, it can be easily chained with other adapter-based logic blocks in a modular 4diac application.

Comparison with Similar Blocks

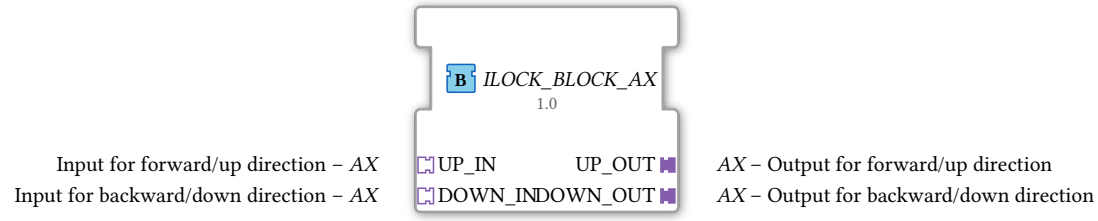
Feature	ILOCK_BLOCK_A2X	Classic SR Latch (RS-FF)	Plain A2X pass-through
Interface	A2X adapter (event + level)	Separate BOOL inputs	A2X adapter
Priority handling	First active input is latched	Set overrides Reset (or vice versa)	None (direct forwarding)
Direction exclusivity	Guaranteed (never both outputs `TRUE`)	Not guaranteed (can be both)	Not guaranteed
Release acknowledgement	Explicit event on stop (`E_UP` / `E_DOWN`)	None	None

Use in event-driven environment	Fully integrated (events with data)	Level-based, no events	Events forwarded, no latched state

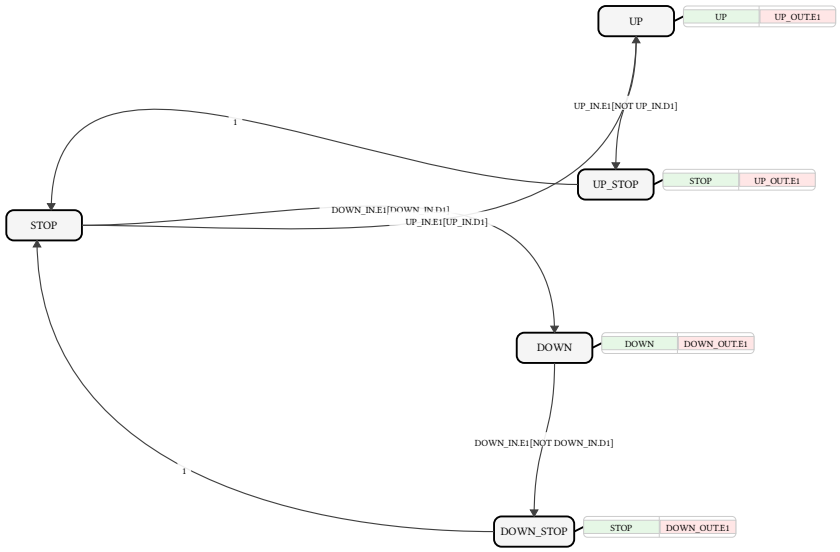
Conclusion

The **ILOCK_BLOCK_A2X** is a robust, event-driven interlock component that elegantly enforces first-input priority and exclusive output direction. Its adapter-based design makes it highly reusable and easy to connect within IEC 61499 systems, while the explicit stop events provide clean handshaking with downstream logic. It is well suited for any application where conflicting directional commands must be safely serialised.

ILOCK_BLOCK_AX



ECC: ILOCK_BLOCK_AX



• — ○ • — ○ • — ○ •

Introduction

The function block **ILOCK_BLOCK_AX** implements an interlock for two opposing directions (up/down or forward/backward). It prioritizes the first active input and ensures that only one direction can be active at a time. The block is implemented as a basic function block (FB) according to IEC 61499-2.

Interface Structure

The function block uses adapter interfaces of type `adapter::types::unidirectional::AX`. Each adapter consists of an event input `E1` and a data input `D1` (BOOL).

Event Inputs

No discrete event inputs. Events are received via the adapter interfaces:

- `UP_IN.E1` – Upward/forward direction event
- `DOWN_IN.E1` – Downward/backward direction event

Event Outputs

No discrete event outputs. Outputs are provided via the adapter plugs:

- `UP_OUT.E1` – Event when the up direction is enabled/disabled
- `DOWN_OUT.E1` – Event when the down direction is enabled/disabled

Data Inputs

- `UP_IN.D1` (BOOL) – Validation signal for the up direction
- `DOWN_IN.D1` (BOOL) – Validation signal for the down direction

Data Outputs

- `UP_OUT.D1` (BOOL) – Validation signal for the up direction at the output
- `DOWN_OUT.D1` (BOOL) – Validation signal for the down direction at the output

Adapters

- **Sockets (Inputs):**
- `UP_IN` – Adapter for upward control
- `DOWN_IN` – Adapter for downward control
- **Plugs (Outputs):**
- `UP_OUT` – Adapter for upward output
- `DOWN_OUT` – Adapter for downward output

Functionality

The module operates using a finite state logic controller (ECC). It starts in the **STOP** state. As soon as one of the two events occurs and the corresponding `D1` signal is `TRUE`, it switches to the corresponding directional state. The state is maintained as long as `D1` of the active input remains `TRUE`. If the `D1` signal of the active input is set to `FALSE` (upon a subsequent event), the machine first switches to a **stop intermediate state** (UP_STOP or DOWN_STOP) and then automatically returns to **STOP**. The other input is ignored as long as one direction is active – the function block prioritizes the input that was active first.

The algorithms set the output signals:

- **UP**: `UP_OUT.D1 := TRUE`, `DOWN_OUT.D1 := FALSE`
- **DOWN**: `UP_OUT.D1 := FALSE`, `DOWN_OUT.D1 := TRUE`
- **STOP**: Both outputs set to `FALSE`

Technical Features

- **Prioritizing the first active input:** Only the first valid command received is executed; The other is blocked until the active one is deactivated.
- **Adapter-based interface:** The use of adapters of type `unidirectional::AX` allows flexible coupling with other components without direct event/data separation.
- **Interlocking at the automaton level:** The state transitions simulate hardware-level safety – `UP_OUT.D1` and `DOWN_OUT.D1 = TRUE` can never occur simultaneously.
- **Intermediate states:** The states `UP_STOP` and `DOWN_STOP` ensure that deactivation is properly acknowledged with an event on the corresponding output adapter.

State overview

State	Description	Output signals
STOP	Idle state, no direction active	`UP_OUT.D1 = FALSE`, `DOWN_OUT.D1 = FALSE`
UP	Up direction active	`UP_OUT.D1 = TRUE`, `DOWN_OUT.D1 = FALSE`
DOWN	Down direction active	`UP_OUT.D1 = FALSE`, `DOWN_OUT.D1 = TRUE`
UP_STOP	Disable UP (exit immediately)	`UP_OUT.D1 = FALSE`, `DOWN_OUT.D1 = FALSE`
DOWN_STOP	Disable DOWN (exit immediately)	`UP_OUT.D1 = FALSE`, `DOWN_OUT.D1 = FALSE`

Transitions:

- `STOP` → `UP` on event from `UP_IN` to `UP_IN.D1 = TRUE`
- `STOP` → `DOWN` on event from `DOWN_IN` to `DOWN_IN.D1 = TRUE`
- `UP` → `UP_STOP` on event from `UP_IN` to `UP_IN.D1 = FALSE`
- `DOWN` → `DOWN_STOP` on event from `DOWN_IN` to `DOWN_IN.D1 = FALSE`
- `UP_STOP` → `STOP` automatically (Transition 1)
- `DOWN_STOP` → `STOP` automatic (Transition 1)

Application Scenarios

- **Traction control (e.g., lifting tables, conveyor belts):** Prevents simultaneous up/down movement.
- **Valve control:** Opening/closing a slide valve with mutual interlocking.
- **Safety-related logic:** Ensures that both drive directions are never activated simultaneously.
- **Control of switches or flaps:** Only one position may be active.

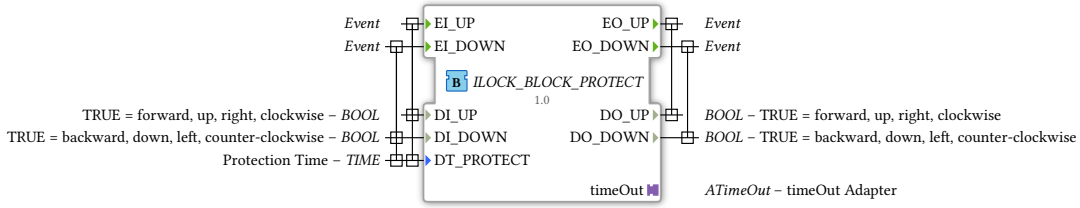
Comparison with similar function blocks

Compared to a simple RS flip-flop, `ILOCK_BLOCK_AX` offers explicit interlocking and the consideration of validation signals (`D1`). A conventional SR function block would allow simultaneous activation of both outputs without external logic. This function block guarantees mutual exclusivity at the state level.

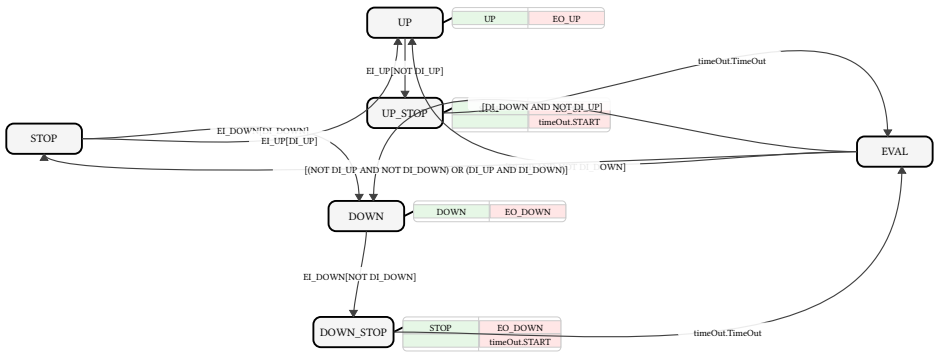
Conclusion

ILOCK_BLOCK_AX` is a compact, safety-oriented functional block for interlocking two opposing directions. Its adapter-based interface and clearly defined mechanism make it easy to integrate and reliable in applications requiring exclusive directional control.

ILOCK_BLOCK_PROTECT



ECC: ILOCK_BLOCK_PROTECT



• – ○ • – ○ • – ○ •

Introduction

The function block `ILOCK_BLOCK_PROTECT` implements an interlock-protected direction control with an adjustable dead time. As soon as an active input (e.g., `EI_UP` with `DI_UP = TRUE`) is detected, it is prioritized, and all conflicting signals are ignored until the active input is reset. After resetting, a configurable protection time (`DT_PROTECT`) elapses before a new direction can be activated. This reliably prevents unwanted direction changes or short circuits.

Interface Structure

Event Inputs

Event	with variables	Description
`EI_UP`	`DI_UP`, `DT_PROTECT`	Event to activate forward direction
`EI_DOWN`	`DI_DOWN`, `DT_PROTECT`	Event to activate reverse direction

Event Outputs

Event	with variables	Description
`EO_UP`	`DO_UP`	Acknowledge active forward direction
`EO_DOWN`	`DO_DOWN`	Acknowledge active reverse direction

Data Inputs

Variable	Type	Initial Value	Description
`DI_UP`	BOOL	–	`TRUE` = forward, upward, right, clockwise
`DI_DOWN`	BOOL	–	`TRUE` = reverse, downward, left, counterclockwise
`DT_PROTECT`	TIME	`T#50ms`	Dead time after resetting a direction

Data Outputs

Variable	Type	Description
`DO_UP`	BOOL	`TRUE` = Forward direction active
`DO_DOWN`	BOOL	`TRUE` = Reverse direction active

Adapters

Adapter	Type	Description
`timeOut`	`iec61499::events::ATimeOut`	Adapter for the time-based control of dead time

Functionality

The function block operates according to the **first priority** principle:

1. **Initial state (`STOP`)**

Both outputs are `FALSE`. When an event with a valid condition is received (e.g., `EI_UP` for `DI_UP = TRUE`), the state changes in the corresponding direction (`UP` or `DOWN`).

1. **Directional States (`UP` / `DOWN`)**

The corresponding output (`DO_UP` or `DO_DOWN`) is set to `TRUE`, the other to `FALSE`.
As long as the active input remains active, new events are ignored (especially opposing ones).
A new event with the same input is only processed if the input previously fell to `FALSE` (falling edge) – see `UP_STOP`/`DOWN_STOP`.
... 3. **Reset to Protection Phase (`UP_STOP` / `DOWN_STOP`)**

If the active input is reset (e.g., DI_UP from TRUE to FALSE), the output is immediately set to FALSE and the timer timeOut is started. The dead time DT_PROTECT begins to run.

1. **Evaluation State (EVAL)**

After the dead time has elapsed, the function block leaves the protection phase and enters the EVAL state. The following decisions are made based on the current inputs:

- DI_UP = TRUE and DI_DOWN = FALSE ⟶ Transition to UP
- DI_DOWN = TRUE and DI_UP = FALSE ⟶ Transition to DOWN
- Both FALSE or both TRUE ⟶ Return to STOP

Important: As long as a new event arrives in the state UP/DOWN while the corresponding input is still TRUE, this event is ignored (no State transition). The stop phase is only initiated upon a falling edge.

Technical Features

- **Internal timer** via adapter timeOut (type ATimeOut) – the protection time is started by each state transition that deactivates a direction.
- **No simultaneous outputs** – DO_UP and DO_DOWN are never simultaneously TRUE. In the EVAL state, both outputs are FALSE.
- **Configurable dead time** via input DT_PROTECT (factory default 50 ms).
- **Compact implementation** as a Basic Function Block with a finite state machine (6 states).
- The interlock not only prevents conflicting commands but also enforces a minimum pause between two direction changes.

State Overview

State	Description
STOP	Idle state: both outputs FALSE, waiting for the first valid event
UP	Forward direction active: DO_UP = TRUE, DO_DOWN = FALSE
DOWN	Reverse direction active: DO_DOWN = TRUE, DO_UP = FALSE
UP_STOP	Protection phase after resetting the forward direction: DO_UP to FALSE, timer running
DOWN_STOP	Protection phase after resetting the reverse direction: DO_DOWN to FALSE, timer running
EVAL	Evaluation state after timer expiration: Decision on next direction or return to STOP

Application Scenarios

- **Directional control of motors** (e.g., conveyor belts, hoists, revolving gates) – prevents simultaneous control in both directions and enforces a dead time for the mechanical change of direction.
- **Interlocking of valves or flaps** – e.g., open/close control with protection against rapid switching to prevent mechanical stress.
- **Safety-related controls** – as part of a simple interlock logic when a safety-certified component is not required.
- **Control of feeders** in agricultural technology (see Copyright HR Agrartechnik GmbH) or in conveyor technology.

Comparison with similar components

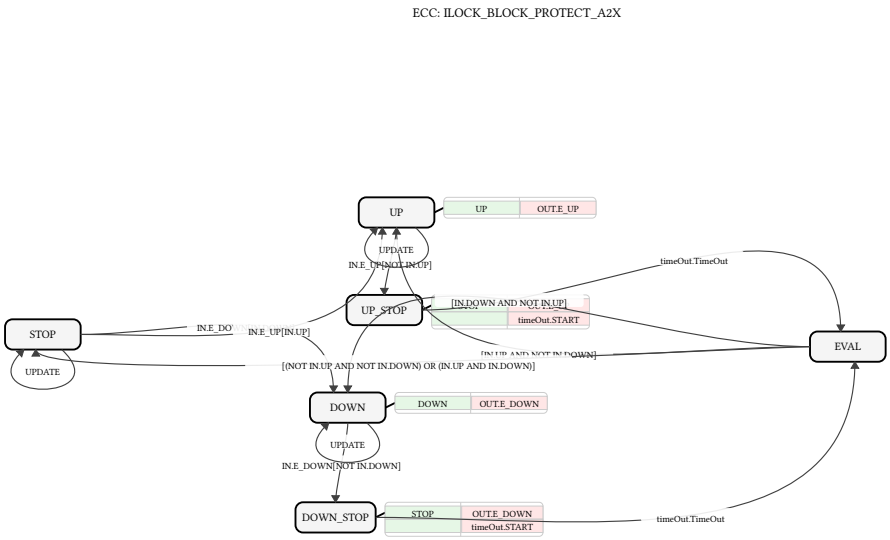
Component	Properties
SR Flip-Flop	Simple set/reset logic, no dead time, no protection against simultaneous signals
ILOCK_BLOCK_PROTECT	Prioritizes the first active input, dead time after each direction change, both outputs never active simultaneously TRUE
Interlock Block without Timer	Blocking logic only, immediate switching possible, no protection time
RS Lock with Time Delay	Similar, but often less configurable and event-driven

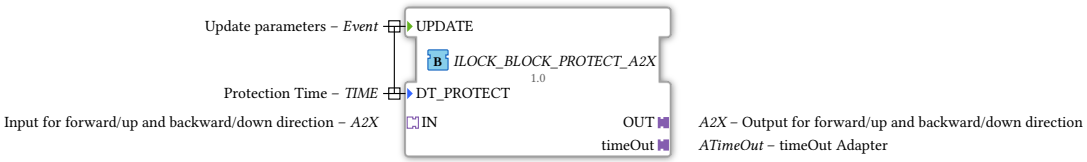
The ILOCK_BLOCK_PROTECTIt offers an integrated, configurable dead time and is specifically optimized for event-driven systems according to IEC 61499.

Conclusion

The ILOCK_BLOCK_PROTECT is ideally suited for applications requiring reliable directional interlocking with adjustable protection time. Its clear state machine and use of the standard ATimeOut adapter make it easy to integrate into larger control projects. Prioritizing the first active input ensures deterministic behavior, and the enforced dead time protects both the mechanical components and the control logic from unwanted states.

ILOCK_BLOCK_PROTECT_A2X





• – ○ • – ○ • – ○ •

Introduction

The ILOCK_BLOCK_PROTECT_A2X function block implements an interlock mechanism for UP/DOWN control signals with a configurable protection dead-time. It prioritizes the first active input and suppresses conflicting commands during the protection interval, preventing rapid switching and potential mechanical or logical damage.

This block is an adapter-based version using the A2X type for unidirectional UP/DOWN signals, and a timer adapter for dead-time handling.

Interface Structure

Event Inputs

- **UPDATE** (Event) – Triggers an update of the protection time parameter (DT_PROTECT). While in any state, an UPDATE event causes the block to refresh its internal parameter without altering the current output state.

Event Outputs

None. The block does not produce direct event outputs. All state changes are reflected through the adapter outputs.

Data Inputs

- **DT_PROTECT** (TIME) – Protection dead-time. Default: T#50ms. This value defines the minimum time during which the output state is held after a change, ignoring conflicting input changes.

Data Outputs

None directly. Output data is provided via the OUT adapter.

Adapters

- **OUT** (Plug, type `adapter::types::unidirectional::A2X`) – Output adapter carrying the UP and DOWN boolean signals to the controlled device.
- **timeOut** (Plug, type `iec61499::events::ATimeOut`) – Timer adapter used to measure the protection dead-time. It receives the DT value and emits a TimeOut event when the interval expires.
- **IN** (Socket, type `adapter::types::unidirectional::A2X`) – Input adapter receiving the UP and DOWN commands from the source.

Functionality

The block acts as a priority interlock with dead-time protection:

1. **Idle (STOP)** – No active output. When either UP or DOWN becomes active (and the other is inactive), the block transitions to the corresponding active state (UP or DOWN) and sets the appropriate output.
2. **Active (UP or DOWN)** – The corresponding output is set. If the active input is released (goes FALSE) or the opposite input becomes active, the block enters a stop phase (UP_STOP or DOWN_STOP) while keeping the current output signal active for the protection time.
3. **Stop Phase (UP_STOP or DOWN_STOP)** – The output is cleared, but the timer is started. After the dead-time expires, the block evaluates the input conditions and moves to:
   - UP if only UP is active,
   - DOWN if only DOWN is active,
   - STOP if both are inactive or both active.
4. **Evaluation (EVAL)** – This state occurs after the dead-time and decides the next state based on the inputs.

The priority rule is: if both UP and DOWN are simultaneously active, the block goes to STOP (no output). If only one is active, that direction is selected. If none, it stays STOP.

The dead-time ensures that after a change, any subsequent changes within the protection time are ignored, preventing rapid toggling.

Technical Features

- Adapter-based design using A2X for propagation of UP/DOWN signals.
- Configurable dead-time via DT_PROTECT input.
- Handles input changes gracefully with a timer-based state machine.
- Supports runtime parameter updates via the UPDATE event.
- Meets IEC 61499 standard for function blocks.

State Overview

The block has six states:

- **STOP**: No outputs active. Waits for valid UP/DOWN command.
- **UP**: UP output active, DOWN inactive.
- **DOWN**: DOWN output active, UP inactive.
- **UP_STOP**: Entered when UP input goes inactive or DOWN becomes active while in UP. Output is cleared and timer starts.
- **DOWN_STOP**: Entered when DOWN input goes inactive or UP becomes active while in DOWN. Output is cleared and timer starts.
- **EVAL**: After timer expiry, evaluates inputs to determine next state.

Transitions are based on input events (IN.E_UP, IN.E_DOWN) combined with the boolean values (IN.UP, IN.DOWN) and the timer TimeOut event.

Application Scenarios

This block is suitable for:

- Controlling bidirectional drives (e.g., motors, actuators) where conflicting commands must be avoided.
- Systems requiring a minimum dwell time between direction changes to protect mechanical components.
- Applications using adapter-based communication (A2X) in IEC 61499 distributed control systems.

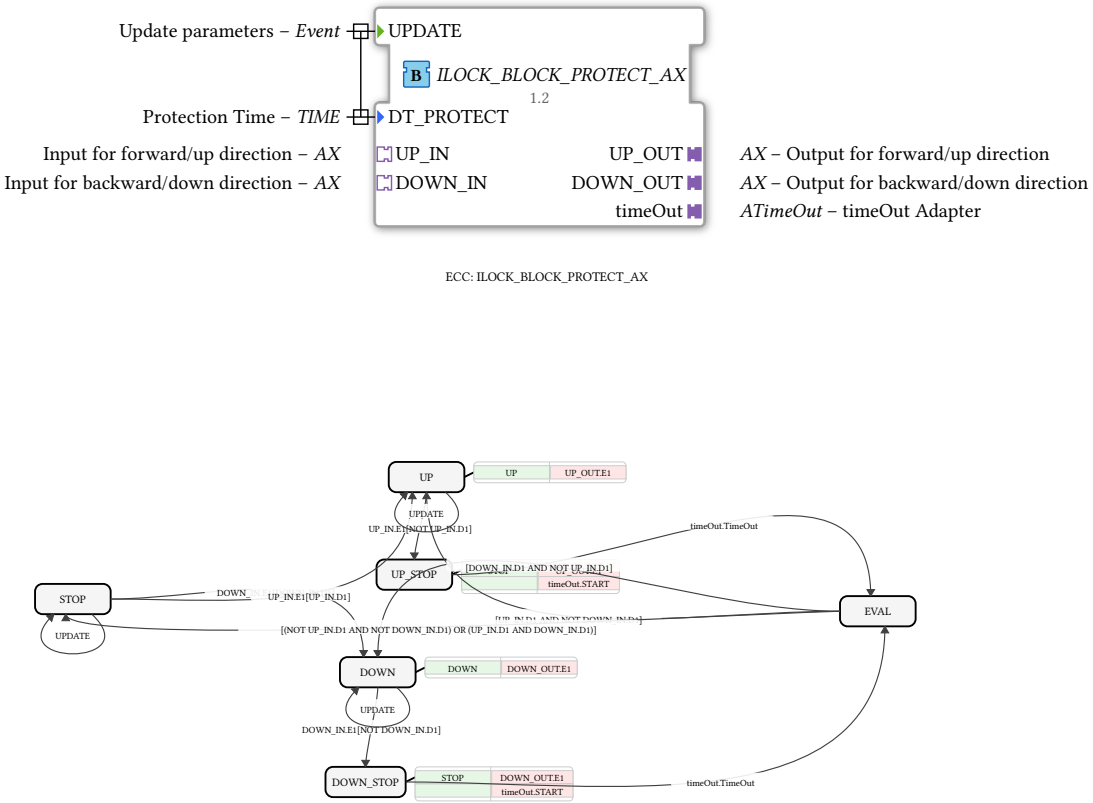
Comparison with Similar Blocks

Compared to a simple interlock block without dead-time, this block adds a protection period to prevent rapid switching. The use of the A2X adapter allows easier integration with other A2X-compatible components, and the timer adapter makes the dead-time configurable. It is more robust than a direct logic block because it incorporates time-based state management.

Conclusion

The ILOCK_BLOCK_PROTECT_A2X provides a reliable interlock solution with adjustable dead-time, leveraging adapter-based communication in IEC 61499. Its state machine ensures safe operation by prioritizing the first active input and suppressing conflicting commands during the protection interval.

ILOCK_BLOCK_PROTECT_AX



• – ○ • – ○ • – ○ •

Introduction

The function block `ILOCK_BLOCK_PROTECT_AX` implements a prioritized interlock for two opposing directions (UP/DOWN) while taking into account a configurable protection dead time. The block is implemented as an adapter version, meaning that the inputs and outputs are provided via IEC 61499 adapters. The goal is the safe control of actuators where only one direction may be active at a time, for

example, during lifting or swiveling movements. The first activated input takes precedence; a change in direction only occurs after the protection time has elapsed and a new evaluation has been performed.

Interface Structure

Event Inputs

Name	Type	Data Carried	Description
`UPDATE`	Event	`DT_PROTECT`	Event for dynamically updating the protection time `DT_PROTECT`.

Event Outputs

The FB does not have any standalone event outputs. Events are provided via the adapters `UP_OUT`, `DOWN_OUT`, and `timeOut`, which are implemented as *plugs*.

Adapter Output	Event	Description
`UP_OUT`	`E1`	Triggered when the UP state is active or the UP_STOP state is reached.
`DOWN_OUT`	`E1`	Triggered when the DOWN state is active or the DOWN_STOP state is reached.
`timeOut`	`TimeOut`	Triggered when the set protection time has expired (generated by the adapter itself).

Data Inputs

Name	Type	Initial Value	Description
`DT_PROTECT`	TIME	`T#50ms`	Protection time that elapses after an input is deactivated before the inputs are re-evaluated.

Data Outputs

The FB does not have any standalone data outputs. Output data is provided via the adapters `UP_OUT`, `DOWN_OUT`, and `timeOut`.

Adapter Output	Data	Description
`UP_OUT`	`D1` (BOOL)	Signals the active UP state (`TRUE`) or inactive UP state (`FALSE`).
`DOWN_OUT`	`D1` (BOOL)	Signals the active DOWN state (`TRUE`) or inactive DOWN state (`FALSE`).
`timeOut`	`DT` (TIME)	Set by the algorithm to the value of `DT_PROTECT` and determines the timer duration.

Adapter

Type	Name	Direction	Description
`adapter::types::unidirectional::AX`	`UP_IN`	Socket	Input for the UP direction. The rising edge event `E1` activates processing; `D1` (BOOL) specifies the desired state.
`adapter::types::unidirectional::AX`	`DOWN_IN`	Socket	Input for the DOWN direction. Analogous to `UP_IN`.
`adapter::types::unidirectional::AX`	`UP_OUT`	Plug	Output for the UP direction. Provides the event and data for the connected logic.
`adapter::types::unidirectional::AX`	`DOWN_OUT`	Plug	Output for the DOWN direction. Provides the event and data.

iec61499::events::ATimeOut	timeOut	Plug	Adapter for a timer. The function block sets the duration (DT) and starts the timer via the event START. The event TimeOut signals the completion of the timer.

Functionality

The function block operates as a state-controlled interlock with two priority inputs. The basic functionality is as follows:

1. **Initial state** (STOP): No direction is active. An incoming event at UP_IN.E1 or DOWN_IN.E1 is processed only if the corresponding data signal D1 has the value TRUE. The first input that meets this condition changes to the corresponding state (UP or DOWN). If both are TRUE simultaneously, neither is accepted – the function block remains in STOP.

The first input that meets this condition changes to the corresponding state (UP or DOWN). 2. **Active State** (UP or DOWN): The output adapter (UP_OUT or DOWN_OUT) signals the active direction via D1 = TRUE and triggers the event E1. The protection time is stored in the timer, but it is not yet active.

1. **Deactivation and Protection Time**: When the active input resets its data signal to FALSE (or the event without TRUE), the FB switches to the corresponding stop state (UP_STOP or DOWN_STOP). The output is set to FALSE, and the timer is started via the event timeOut.START. After the set protection time has elapsed (timeOut.TimeOut), the function block switches to the evaluation state (EVAL).

The function block is set to FALSE, and the timer is started via the event timeOut.START. 4. **Evaluation** (EVAL): In the evaluation state, the current data signals of both inputs are evaluated:

- Only UP_IN.D1 = TRUE ⟶ Switch to UP
- Only DOWN_IN.D1 = TRUE ⟶ Switch to DOWN
- Neither or both TRUE ⟶ Switch back to STOP

1. **Dynamic Update**: The event UPDATE allows the protection time DT_PROTECT to be changed during operation. The updated time takes effect the next time the timer is used (the timer is only started in the stop states). The event can be received in any state without changing the current control state.

Technical Features

- **Adapter Interface**: All data exchange takes place via adapters. This ensures a clean separation between control logic and peripherals, increases reusability, and improves encapsulation.
- **Dynamic Protection Time**: The dead time can be changed at runtime via the UPDATE event without restarting the function block. This is particularly useful in applications with variable switching times (e.g., temperature-dependent delays).
- **Dead Time Principle**: After an input is deactivated, the function block does not immediately switch to the other state, but enforces a minimal delay (DT_PROTECT). This prevents bounce or unwanted rapid switching.
- **Prioritization after first activation**: The logic only reacts to the first TRUE signal of an input event. Simultaneous activation of both inputs does not result in a change of direction.
- **Self-holding without external event**: As long as an active input maintains its D1 at TRUE, the corresponding output state is retained, even without further events.

State Overview

State	Description	Activities
STOP	Initial and idle state. No direction active.	–
UP	UP direction active.	Algorithm UP: setsUP_OUT.D1 = TRUE, DOWN_OUT.D1 = FALSE, transfers DT_PROTECT to timeOut.DT and sends UP_OUT.E1.
DOWN	DOWN direction active.	Algorithm DOWN: sets DOWN_OUT.D1 = TRUE, UP_OUT.D1 = FALSE, transfers DT_PROTECT to timeOut.DT and sends DOWN_OUT.E1.

UP_STOP	Wait time after deactivating UP.	Algorithm STOP: sets both outputs to FALSE, transfers DT_PROTECT to timeOut.DT, and starts the timer with timeOut.START.
DOWN_STOP	Wait time after deactivating DOWN.	Algorithm STOP (analogous to UP_STOP).
EVAL	Evaluation after the protection period expires.	No algorithm; the transitions determine the next state based on the current input data.

Transitions (Simplified):

- STOP → UP if UP_IN.E1 and UP_IN.D1 = TRUE
- STOP → DOWN if DOWN_IN.E1 and DOWN_IN.D1 = TRUE
- UP → UP_STOP if UP_IN.E1 and UP_IN.D1 = FALSE
- DOWN → DOWN_STOP if DOWN_IN.E1 and DOWN_IN.D1 = FALSE
- UP_STOP → EVAL if timeOut.TimeOut
- DOWN_STOP → EVAL if timeOut.TimeOut
- EVAL → UP if UP_IN.D1 = TRUE and DOWN_IN.D1 = FALSE
- EVAL → DOWN if DOWN_IN.D1 = TRUE and UP_IN.D1 = FALSE
- EVAL → STOP if both FALSE or both TRUE
- In any state, the event UPDATE can return to the same state (only used to update DT_PROTECT).

Application Scenarios

- **Lifting Platform Control**: Interlocking of lifting and lowering; after releasing the lifting command, a dead time of, for example, 100 ms is observed before lowering is possible.
- **Valve Control**: Switching between two media (e.g., heating/cooling) with a protection time to prevent pressure surges.
- **Robot Axes**: Collision prevention through simultaneous control of opposing movements.
- **Door Control**: Opening/closing with a delay when changing direction to protect end stops.

Comparison with Similar Function Blocks

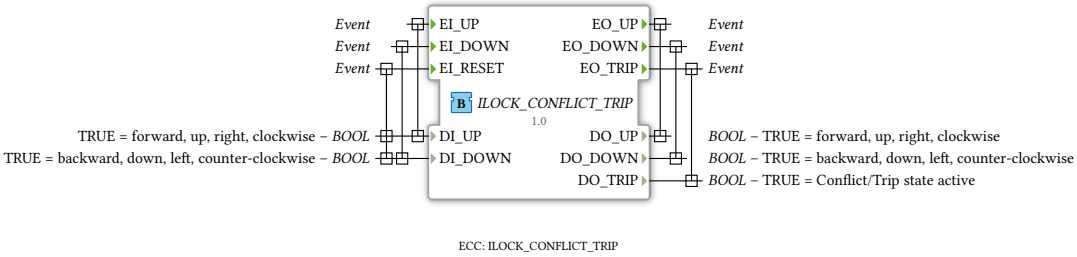
Function Block / Property	ILOCK_BLOCK_PROTECT_AX	Simple Interlock (without dead time)	Interlock with fixed dead time
Dead Time	Dynamically configurable via UPDATE	None	Fixed (usually wired in the block)
Interface	Adapter (AX), loose coupling	Mostly direct inputs/outputs	Direct inputs/outputs
Priority	First active input	Often fixed priority	Varies
Evaluation after dead time	Yes, with a clear state machine	No	Partially
Flexibility	High (runtime change)	Low	Medium

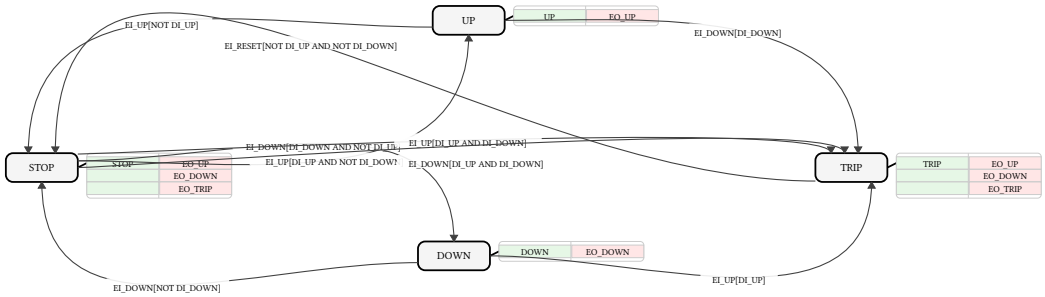
The ILOCK_BLOCK_PROTECT_AX offers a particularly flexible solution thanks to its adapter interface and dynamically adjustable dead time. It is ideal for modular, reusable control projects according to IEC 61499.

Conclusion

The ILOCK_BLOCK_PROTECT_AX function block is a robust, state-controlled interlocking block for two opposing directions. The combination of prioritization of the first active input, a configurable protection dead time, and a full adapter interface makes it an excellent choice for safe and flexible control tasks in automation technology. Dynamic adjustment of the dead time via the UPDATE event increases adaptability to changing process conditions. Thanks to its clear state machine and compliance with IEC 61499 standards, the module can be used in both simple and complex distributed systems.

ILOCK_CONFLICT_TRIP





• – ○ • – ○ • – ○ •

Introduction

The **ILOCK_CONFLICT_TRIP** function block is used for **prioritized interlocking** with **conflict detection**. It evaluates two opposing binary signals (e.g., “Forward” and “Reverse”) and only actively passes on one of the two commands as long as they are not present simultaneously. If both inputs are active at the same time, a **trip state** (error/lock) is triggered, which can only be cleared by an explicit reset (with inactive inputs). This function block is specifically designed for safety-critical applications where conflicting control commands must be reliably detected.

Interface Structure

Event Inputs

Event	Description
EI_UP	Event for processing an “up” request (with data `DI_UP`)
EI_DOWN	Event for processing a “down” request (with data `DI_DOWN`)
EI_RESET	Event for resetting the trip state (reads both data inputs)

Event Outputs

Event	Description
EO_UP	Acknowledges the output of the “up” command (when the UP state is active)

EO_DOWN	Acknowledges the output of the “down” command (when the DOWN state is active)
EO_TRIP	Indicates that a trip state exists (when TRIP is active)

Data Inputs

Name	Type	Description
DI_UP	BOOL	TRUE = forward, up, right, clockwise
DI_DOWN	BOOL	TRUE = backward, down, left, counterclockwise

Data Outputs

Name	Type	Description
DO_UP	BOOL	TRUE = “Up” output active
DO_DOWN	BOOL	TRUE = Output “Down” active
DO_TRIP	BOOL	TRUE = Conflict/Trip state active

Adapters

No adapters available.

Functionality

The function block has four operating states: **STOP**, **UP**, **DOWN**, and **TRIP**.

- **STOP (Idle State):** Both data outputs are FALSE.
- With `EI_UP` with `DI_UP = TRUE` and `DI_DOWN = FALSE`, the block switches to the **UP** state.
- With `EI_DOWN` with `DI_DOWN = TRUE` and `DI_UP = FALSE`, it switches to **DOWN**.
- If `EI_UP` or `EI_DOWN`, and both data inputs are TRUE, it switches directly to **TRIP** (conflict).
- **UP (upward active):** `DO_UP = TRUE`, `DO_DOWN = FALSE`, `DO_TRIP = FALSE`.
- If `EI_UP` is encountered again, and `DI_UP = FALSE` is encountered again, it switches back to **STOP** (deactivation).
- If `EI_DOWN` is encountered, and `DI_DOWN = TRUE` is encountered, it switches to **TRIP** (an opposing request is detected during operation).
- **DOWN (Active Downward):** `DO_DOWN = TRUE`, `DO_UP = FALSE`, `DO_TRIP = FALSE`.
- On `EI_DOWN`, when `DI_DOWN = FALSE` occurs again, it switches to **STOP**.
- On `EI_UP`, when `DI_UP = TRUE` occurs, it switches to **TRIP**.
- **TRIP (Error/Lock):** `DO_TRIP = TRUE`, both direction outputs FALSE.
- **Only way to exit the trip:** A `EI_RESET` event where `DI_UP = FALSE` and `DI_DOWN = FALSE` are present. Then it returns to **STOP**.

Prioritization mechanism: The first valid input received is served until it is withdrawn or a conflict with the other input occurs. Simultaneous TRUE values on both data inputs immediately lead to the trip state.

Technical features

- **Reset only allowed in trip:** The function block can only be reset from the TRIP state to the STOP state by `EI_RESET`. A reset during normal states (UP/DOWN/STOP) has no effect.
- **Conditions for Trip Transitions:**
- From STOP: `(EI_UP UND DI_UP UND DI_DOWN)` OR `(EI_DOWN UND DI_UP UND DI_DOWN)`
- From UP: `(EI_DOWN UND DI_DOWN)`
- From DOWN: `(EI_UP UND DI_UP)`
- **Event Acknowledgment:** The event outputs are **always** output together with the data outputs (see `With` links), so the caller can immediately access the new state.
- **Use of 4diac (IEC 61499):** The FB is implemented as a Basic Function Block with its own ECC (Execution Control Chart).

State Overview

State	DO_UP	DO_DOWN	DO_TRIP	Description
STOP	FALSE	FALSE	FALSE	Idle state, no direction active
UP	TRUE	FALSE	FALSE	Upward direction active
DOWN	FALSE	TRUE	FALSE	Downward direction active
TRIP	FALSE	FALSE	TRUE	Conflict/lock active

Application Scenarios

- **Motor control for linear units or rotary actuators:** Prevents simultaneous forward/reverse commands.
- **Hydraulic valve control:** Protection against conflicting switching commands (e.g., raising and lowering simultaneously).
- **Safety Interlock in Automation:** Detection of operator errors and triggering of a safe stop.
- **PLC Replication in Distributed Systems:** As part of a control logic that must avoid conflicting states.

Comparison with Similar Function Blocks

- **SR Latch / Flip-Flop:** A simple SR latch stores a state but does not detect conflicts with simultaneous "Set" and "Reset" signals. ILOCK_CONFLICT_TRIP enters the trip state instead of creating an undefined state.
- **F_TRIG / R_TRIG (Edge Detection):** These function blocks only detect signal edges but do not have any interlock logic.
- **Standard Interlock Function Blocks (e.g., from IEC 61131-3):** Many offer simple mutual interlocking (e.g., motor interlocking) but no dedicated trip state with a reset requirement. The ILOCK_CONFLICT_TRIP is more robust in fault situations.

Conclusion

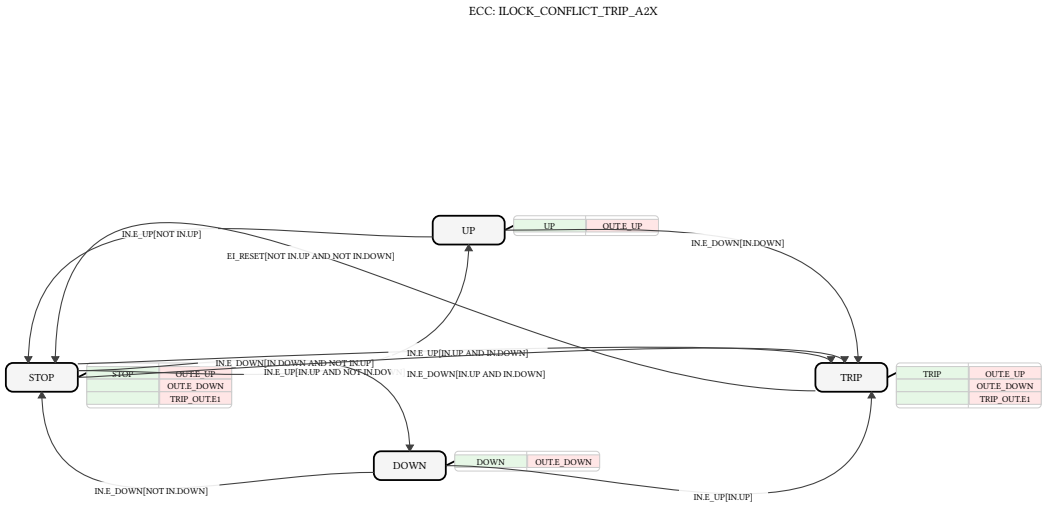
The **ILOCK_CONFLICT_TRIP** is a compact, safety-oriented function block for the robust interlocking of two opposing control signals. It offers clear prioritization of the first activated input, detects conflicts through simultaneous activity, and forces an explicit reset after a fault. Its state machine is easy to understand and is ideally suited for applications where conflicting control commands must be reliably intercepted—for example, in machine or vehicle control.

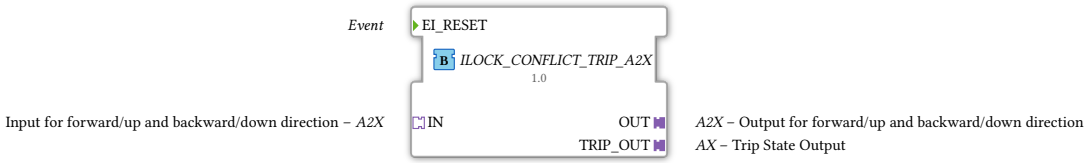
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ILOCK_CONFLICT_TRIP_A2X





• — ○ • — ○ • — ○ •

Introduction

The **ILOCK_CONFLICT_TRIP_A2X** function block implements an interlock mechanism for control applications requiring a safe handling of conflicting directional commands. It accepts two mutually exclusive direction signals (e.g., forward/up and backward/down) via a single bidirectional adapter and ensures that only one direction is ever active at a time. If both commands are present simultaneously, the block enters a **trip** state, setting a dedicated alarm output. Recovery from the trip state requires an explicit reset event, but only after both conflicting inputs have been released.

Interface Structure

The block communicates exclusively through adapter instances. There are no dedicated event or data ports other than the reset event input.

Event Inputs

Event	Description
`EI_RESET`	Clears the trip state when the conflict is resolved (i.e., both `IN.UP` and `IN.DOWN` are `FALSE`).

Event Outputs

There are no direct event output ports. All event-driven signalling is performed via the adapter outputs (`OUT.E_UP`, `OUT.E_DOWN`, `TRIP_OUT.E1`).

Data Inputs

There are no standalone data inputs. All data is read from the `IN` adapter (see below).

Data Outputs

There are no standalone data outputs. All data values are written to the OUT and TRIP_OUT adapters.

Adapters

Adapter	Type	Description
IN	adapter::types::unidirectional::A2X	Receives the directional commands. Carries the events E_UP, E_DOWN and the data signals UP, DOWN.
OUT	adapter::types::unidirectional::A2X	Mirrors the currently active and safe direction. Provides the events E_UP, E_DOWN and data UP, DOWN.
TRIP_OUT	adapter::types::unidirectional::AX	Outputs the trip status. Contains the event E1 and data D1 (trip indication, TRUE when tripped).

Functionality

The block operates as a state machine with four states: STOP, UP, DOWN, and TRIP. Its core logic is as follows:

- **STOP**: No direction is active. The block waits for a valid event on the IN adapter.
- **UP / DOWN**: A single direction is active. The corresponding output data and event are sent.
- **TRIP**: A conflict has been detected. The block deactivates both directions, sets TRIP_OUT.D1 to TRUE, and emits the trip event. It remains in this state until an EI_RESET event occurs while both IN.UP and IN.DOWN are FALSE.

The block prioritises the first active input: if only one of the two inputs is active, the block moves to the corresponding state. If both inputs become active simultaneously (or while a direction is already active and the other input arrives), the block immediately trips. This ensures that opposing commands can never be forwarded to the actuator.

Technical Features

- **State machine based on IEC 61499 ECC** – implemented as a basic function block.
- **Adapter-based I/O** – uses unidirectional adapters to separate input and output paths.
- **Conflict detection** – immediately reacts to simultaneous UP and DOWN signals.
- **Reset safety** – the trip state can only be cleared when the conflict has been fully removed (both inputs FALSE).
- **Event synchronisation** – transitions are triggered by events on the IN adapter, ensuring deterministic behaviour.

State Overview

State	Description	Transition Conditions
STOP	No direction active, all outputs FALSE.	⟶ UP on IN.E_UP if IN.UP and NOT IN.DOWN;⟶ DOWN on IN.E_DOWN if IN.DOWN and NOT IN.UP;⟶ TRIP on either event if both IN.UP and IN.DOWN are TRUE.
UP	Forward/up direction active.	⟶ STOP on IN.E_UP if NOT IN.UP;⟶ TRIP on IN.E_DOWN if IN.DOWN.
DOWN	Backward/down direction active.	⟶ STOP on IN.E_DOWN if NOT IN.DOWN;⟶ TRIP on IN.E_UP if IN.UP.
TRIP	Conflict detected, both directions deactivated, TRIP_OUT.D1 set to TRUE.	⟶ STOP on EI_RESET if NOT IN.UP and NOT IN.DOWN.

Application Scenarios

This block is well suited for safety-critical interlocking in machinery, conveyor systems, or any application where two opposing actuation commands must never be delivered simultaneously. Typical use cases include:

- Hoist or crane direction control.
- Hydraulic or pneumatic cylinder extension/retraction with safety interlocks.
- Motor control with forward/reverse operation and anti-conflict logic.
- Any system requiring a “trip” (fault) condition when conflicting commands are issued.

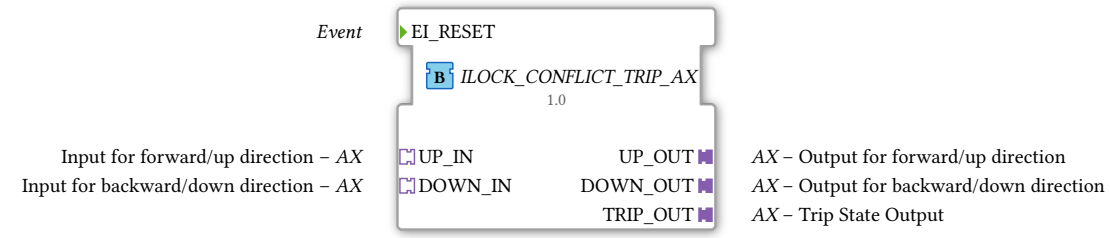
Comparison with Similar Blocks

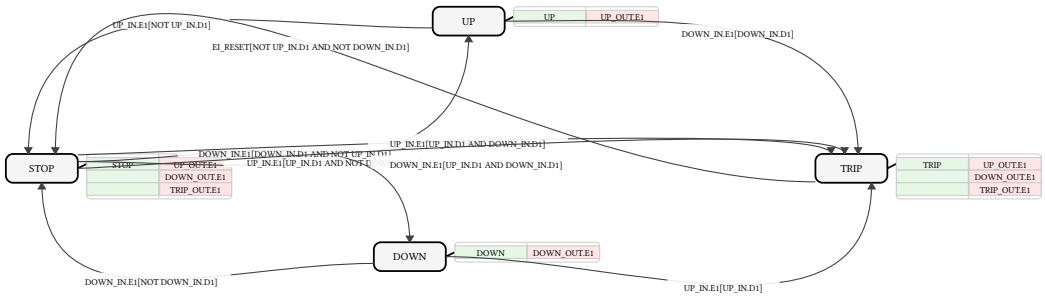
Compared to a simple interlock that merely prioritises one input, the `ILOCK_CONFLICT_TRIP_A2X` adds an explicit trip state to signal a hardware or software conflict. This makes it more suitable for systems that require a visible alarm and a manual or logical reset after a fault. Unlike a pure priority encoder, it does not silently ignore the secondary input – it reacts to the conflict and forces the system into a safe state. The adapter-based design also simplifies integration with other 4diac components using the same adapter types.

Conclusion

The `ILOCK_CONFLICT_TRIP_A2X` function block provides a robust and safe interlocking solution for directional commands. Its clear state machine, conflict detection, and trip/reset behaviour make it a valuable component for automation projects that demand high reliability and fail-safe operation. By using standard adapters, it can be easily connected to existing I/O infrastructure and integrated into larger control applications.

ILOCK_CONFLICT_TRIP_AX





Introduction

The function block `ILOCK_CONFLICT_TRIP_AX` implements interlock logic for two opposing directions (UP and DOWN). It prioritizes the first active input signal and immediately switches to a trip state when both inputs are activated simultaneously. Resetting from the trip state is only possible via the event `EI_RESET` when both inputs are inactive. All communication takes place via adapters of type `unidirectional::AX`, enabling flexible and modular integration.

Interface Structure

Event Inputs

- **EI_RESET** – Event to reset the trip state (only effective if both data inputs of the adapters are FALSE).

Event Outputs

The FB has no direct event outputs. State changes are signaled via the event outputs of the output adapters (plugs):

- **UP_OUT.E1**
- **DOWN_OUT.E1**
- **TRIP_OUT.E1**

Data Inputs

The FB has no direct data inputs. The input data is provided via the adapter sockets:

- **UP_IN.D1** (BOOL) – Enables the up direction.
- **DOWN_IN.D1** (BOOL) – Enables the down direction.

Data Outputs

The FB has no direct data outputs. The output data is provided via the adapter plugs:

- **UP_OUT.D1** (BOOL) – Signal for the up direction.
- **DOWN_OUT.D1** (BOOL) – Signal for the down direction. - **TRIP_OUT.D1** (BOOL) – Signal for trip state.

Adapters

Sockets (Inputs)

Adapter	Type	Description
`UP_IN`	`adapter::types::unidirectional::AX`	Input for up-direction (event + data)
`DOWN_IN`	`adapter::types::unidirectional::AX`	Input for down-direction (event + data)

Plugs (Outputs)

Adapter	Type	Description
`UP_OUT`	`adapter::types::unidirectional::AX`	Output for Upward Direction
`DOWN_OUT`	`adapter::types::unidirectional::AX`	Output for Downward Direction
`TRIP_OUT`	`adapter::types::unidirectional::AX`	Trip State Output

Note: The interface `unidirectional::AX` is interpreted here as an adapter with one event (E1) and one BOOL data element (D1), which carries the activation state.

Functionality

The function block operates as a finite state machine with four states:

1. **STOP** – Idle state. All outputs are FALSE.

- On event `UP_IN.E1` with the condition `UP_IN.D1 AND NOT DOWN_IN.D1` ⟶ transition to **UP**.
- On event `DOWN_IN.E1` with condition `DOWN_IN.D1 AND NOT UP_IN.D1` ⟶ transition to **DOWN**.
- On event at either input with condition `UP_IN.D1 AND DOWN_IN.D1` ⟶ transition to **TRIP**.

1. **UP** – Upward direction active. `UP_OUT.D1 = TRUE`, all others FALSE.

- On event `UP_IN.E1` with `NOT UP_IN.D1` ⟶ return to **STOP**.

- On event `DOWN_IN.E1` with `DOWN_IN.D1` → **TRIP** (conflict).

1. **DOWN** – Downward direction active. `DOWN_OUT.D1 = TRUE`, all others FALSE.

- On event `DOWN_IN.E1` with `NOT DOWN_IN.D1` → return to **STOP**.
- On event `UP_IN.E1` with `UP_IN.D1` → **TRIP** (conflict).

1. **TRIP** – Error/conflict state. `TRIP_OUT.D1 = TRUE`, all others FALSE.

- Only output: Event `EI_RESET`, if `NOT UP_IN.D1 AND NOT DOWN_IN.D1` → return to **STOP**.

Prioritization is implicit: As long as there is no conflict, the first detected and valid request is executed. A change from UP to DOWN (or vice versa) without an intermediate stop is only possible via STOP, unless a conflict occurs that immediately triggers a trip.

Technical Features

- **Adapter-based communication:** All inputs and outputs are implemented as adapters, enabling loose coupling and easy reuse in complex systems.
- **Trip on conflict:** Simultaneous activation of both directions immediately triggers a separate trip state, which can only be exited by an explicit reset.
- **Reset condition:** The reset (`EI_RESET`) is only effective if both input data are FALSE – this prevents accidental release while a conflict persists.
- **No direct events/data:** The function block has no traditional event or data inputs/outputs, but uses only the adapter interfaces.

State overview

State	UP_OUT.D1	DOWN_OUT.D1	TRIP_OUT.D1	Description
`STOP`	FALSE	FALSE	FALSE	Resting position, no direction active
`UP`	TRUE	FALSE	FALSE	Upward direction active
`DOWN`	FALSE	TRUE	FALSE	Downward direction active
`TRIP`	FALSE	FALSE	TRUE	Conflict / Trip, requires reset

Important Transitions:

- STOP → UP: when UP_IN is active and DOWN_IN is inactive.
- STOP → DOWN: when DOWN_IN is active and UP_IN is inactive.
- STOP → TRIP: when both inputs are active simultaneously (conflict).
- UP → STOP: when UP_IN is deactivated.
- UP → TRIP: when DOWN_IN is active.
- DOWN → STOP: when DOWN_IN is deactivated.
- DOWN → TRIP: when UP_IN is active.
- TRIP → STOP: only via EI_RESET when both inputs are inactive.

Application Scenarios

- **Interlock Control on Machines:** Prevents simultaneous movement in two opposite directions (e.g., raising/lowering, forward/backward).
- **Safety-Related Applications:** Detects critical conflict situations and enforces a defined error state that must be acknowledged.
- **Adapter-Based Systems:** Integration into modular automation solutions where components communicate via standardized adapters.

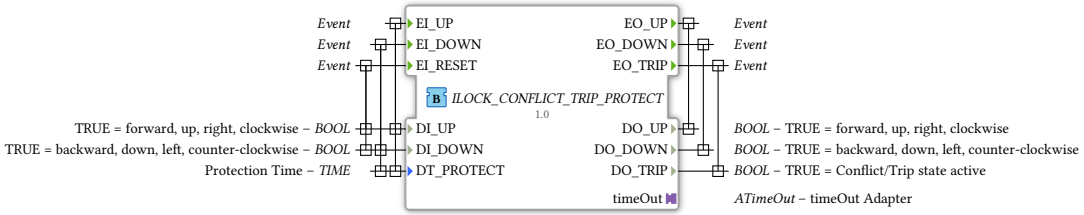
Comparison with Similar Function Blocks

- Simple interlock function blocks (e.g., `ILOCK`) often switch directly between directions without implementing a trip state.
- Function blocks with priority logic favor one direction, while this function block prioritizes the first arriving request and reports an error in case of a conflict.
- `ILOCK_CONFLICT_TRIP_AX` is distinguished by its explicit trip state and adapter interface – this enables a clean separation of control and diagnostic functions.

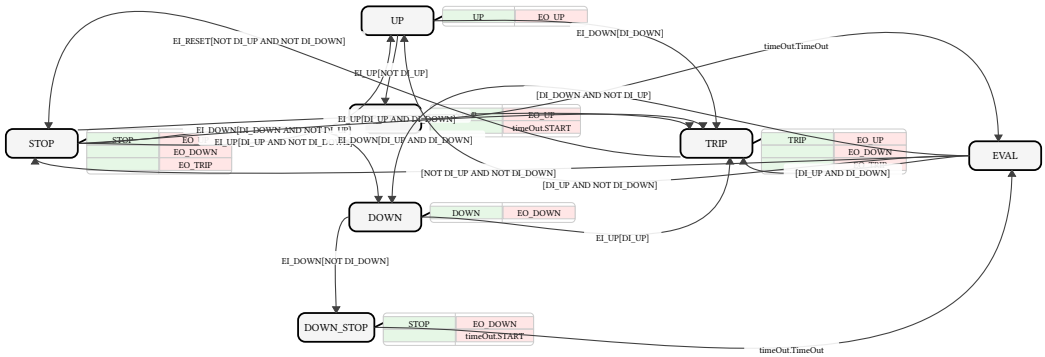
Conclusion

The ILOCK_CONFLICT_TRIP_AX is a robust interlock function block specifically designed for conflict situations. The trip state with external reset ensures safe handling of parallel requests. The adapter-based structure facilitates integration into larger automation projects and promotes reusability. This function block represents a reliable and flexible solution for safety-critical or fault-tolerant systems.

ILOCK_CONFLICT_TRIP_PROTECT



ECC: ILOCK_CONFLICT_TRIP_PROTECT



• – ○ • – ○ • – ○ •

Introduction

The function block ILOCK_CONFLICT_TRIP_PROTECT extends ILOCK_CONFLICT_TRIP with a configurable protection dead time (DT_PROTECT), analogous to how ILOCK_BLOCK_PROTECT extends the behavior of ILOCK_BLOCK. It still prioritizes the first active input, triggers a trip state when both directions are activated simultaneously, and requires an explicit EI_RESET thereafter. What's new is that after the active input is released, the dead time DT_PROTECT must elapse before the function block re-evaluates the current input signals – only then is the next direction (or, if applicable, a trip) taken.

Interface Structure

Event Inputs

Name	Carrying Data	Description
EI_UP	DI_UP, DT_PROTECT	Event for the up/forward direction.
EI_DOWN	DI_DOWN, DT_PROTECT	Down/Backward Direction Event.
EI_RESET	DI_UP, DI_DOWN	Reset Trip State; only effective if both data inputs are FALSE.

Event Outputs

Name	Carrying Data	Description
EO_UP	DO_UP	Triggered when the UP direction is enabled or disabled.
EO_DOWN	DO_DOWN	Triggered when the DOWN direction is enabled or disabled.
EO_TRIP	DO_TRIP	Sent in the TRIP state, it indicates the trip status.

Data Inputs

- DI_UP (BOOL) – TRUE = forward/upward/right/clockwise.
- DI_DOWN (BOOL) – TRUE = backward/downward/left/counterclockwise.
- DT_PROTECT (TIME, initial value T#50ms) – Protection dead time that must elapse after the active input is released before it can be re-evaluated.

Data Outputs

- DO_UP (BOOL) – Signals the active UP direction.
- DO_DOWN (BOOL) – Signals the active DOWN direction.
- DO_TRIP (BOOL) – Signals the trip/conflict state.

Adapters

Adapter	Type	Direction	Description
timeOut	iec61499::events::ATimeOut	Plug	Timer adapter for protection time. The function block sets timeOut.DT and starts it via timeOut.START; timeOut.TimeOut signals the execution.

Functionality

The function block operates as a finite state machine (ECC) with seven states:

1. **STOP** – Idle state, all outputs FALSE. At EI_UP with DI_UP AND NOT DI_DOWN → **UP** at EI_DOWN with DI_DOWN AND NOT DI_UP → **DOWN** if both data signals are TRUE at EI_UP or EI_DOWN → immediate **TRIP**.
2. **UP** – DO_UP = TRUE. At EI_UP[NOT DI_UP] → **UP_STOP** at EI_DOWN[DI_DOWN] (conflict during active UP) → immediate **TRIP**.
3. **DOWN** – DO_DOWN = TRUE. At EI_DOWN[NOT DI_DOWN] → **DOWN_STOP** at EI_UP[DI_UP] (conflict during active DOWN) → immediate **TRIP**.
4. **UP_STOP** / **DOWN_STOP** – Transition states after the active input is released. The algorithm STOP sets all outputs to FALSE, transfers DT_PROTECT to timeOut.DT, and starts the timer (timeOut.START). Upon expiration (timeOut.TimeOut) → **EVAL**.
5. **EVAL** – No separate algorithm. The current values of DI_UP/DI_DOWN are re-evaluated: only DI_UP TRUE → **UP** only DI_DOWN TRUE → **DOWN** both FALSE → **STOP** both TRUE → **TRIP**.
6. **TRIP** – DO_TRIP = TRUE, all other outputs FALSE. Exit only via EI_RESET if NOT DI_UP AND NOT DI_DOWN → **STOP**.

Trip detection itself still occurs immediately and independently of the dead time: A conflict during STOP, UP, or DOWN always directly triggers TRIP. The dead time DT_PROTECT only affects the period between the release of an active input and the acquisition of the next direction.

Trip detection itself continues to occur immediately and independently of the dead time: A conflict during STOP, UP, or DOWN always directly triggers TRIP. The dead time DT_PROTECT is only effective between the release of an active input and the acquisition of the next direction.

Technical Features

- **Combination of two patterns:** This function block combines the trip-on-conflict logic of ILOCK_CONFLICT_TRIP with the dead-time logic of ILOCK_BLOCK_PROTECT.

- **Immediate trip, delayed release:** Conflicts are detected without delay; only the return to a new valid state after release is delayed by `DT_PROTECT`.
- **Reset condition:** `EI_RESET` only takes effect if both data inputs are inactive – a reset while a conflict persists has no effect.
- **Timer restart in every intermediate stop state:** `timeOut.DT` is reused from `DT_PROTECT` upon each entry into `UP_STOP`/`DOWN_STOP`, so that any parameter change takes effect immediately in the next delay.

State Overview

State	DO_UP	DO_DOWN	DO_TRIP	Description
`STOP`	FALSE	FALSE	FALSE	Idle state, no direction active.
`UP`	TRUE	FALSE	FALSE	Up direction active.
`DOWN`	FALSE	TRUE	FALSE	Downward direction active.
`UP_STOP`	FALSE	FALSE	FALSE	Waiting for `DT_PROTECT` to expire after UP is released.
`DOWN_STOP`	FALSE	FALSE	FALSE	Waiting for `DT_PROTECT` to expire after DOWN is released.
`EVAL`	–	–	–	No algorithm; decides the next state based on current inputs.
`TRIP`	FALSE	FALSE	TRUE	Conflict/Trip, requires `EI_RESET`.

Application Scenarios

- **Drives with Overrun:** When a mechanical or hydraulic overrun time must be observed after releasing a travel command before switching to the opposite direction is safe.
- **Safety-Oriented Interlock with Acknowledgement Required:** Applications where a simultaneous command in both directions is considered an error and must be explicitly acknowledged.
- **Valve or Flap Controls:** Protection against pressure surges or mechanical overload through a minimum pause between direction changes.

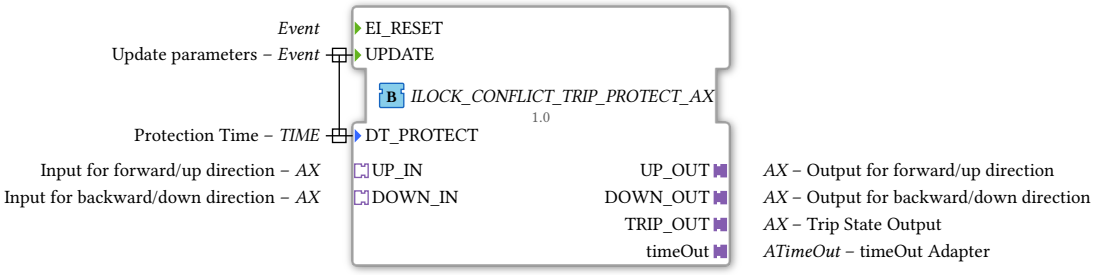
Comparison with Similar Function Blocks

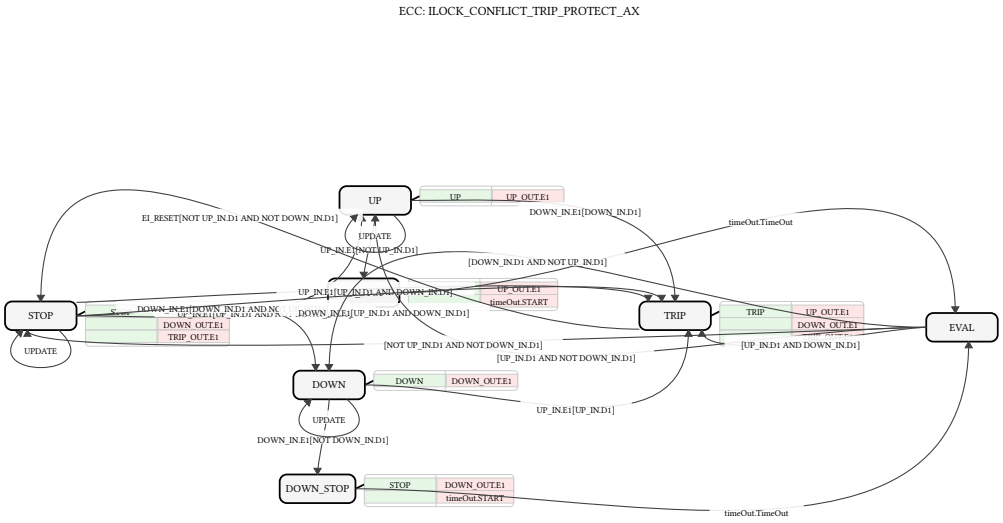
Compared to `ILOCK_CONFLICT_TRIP`, this function block adds the states `UP_STOP`, `DOWN_STOP`, and `EVAL`, as well as the `timeOut` adapter – the trip logic itself remains identical. It differs from `ILOCK_BLOCK_PROTECT` in that a simultaneous command in both directions is not ignored, but rather treated as an explicit error condition (`TRIP`) that requires `EI_RESET`.

Conclusion

`ILOCK_CONFLICT_TRIP_PROTECT` combines the clear error detection of `ILOCK_CONFLICT_TRIP` with the protection dead time of `ILOCK_BLOCK_PROTECT`. It is suitable for applications that require both strict conflict detection with mandatory acknowledgment and a minimum pause between direction changes.

ILOCK_CONFLICT_TRIP_PROTECT_AX





• — ○ • — ○ • — ○ •

Introduction

The function block `ILOCK_CONFLICT_TRIP_PROTECT_AX` is the adapter version of `ILOCK_CONFLICT_TRIP_PROTECT`: It combines the trip-on-conflict logic of `ILOCK_CONFLICT_TRIP_AX` with the protection dead time of `ILOCK_BLOCK_PROTECT_AX`. The first active input is prioritized; simultaneous activation of both directions immediately triggers a trip, which can only be reset via `EI_RESET`. After the active input is released, the block additionally waits for the configurable time `DT_PROTECT` before re-evaluating the inputs. The process data (`UP_IN`/`DOWN_IN`/`UP_OUT`/`DOWN_OUT`/`TRIP_OUT`) is routed via adapters of type `unidirectional::AX`; the protection-time timer additionally uses the adapter `iec61499::events::ATimeOut` (`timeOut`).

Interface Structure

Event Inputs

Name	Carrying Data	Description
`EI_RESET`	–	Resets the trip state; only effective if `UP_IN.D1` and `DOWN_IN.D1` are both FALSE.
`UPDATE`	`DT_PROTECT`	Updates the protection time `DT_PROTECT` at runtime without changing the current state.

Event Outputs

No direct event outputs. State changes are signaled via the events of the output adapters (plugs):

- `UP_OUT.E1`, `DOWN_OUT.E1`, `TRIP_OUT.E1`

Data Inputs

`UP_IN`/`DOWN_IN` are not direct data inputs - they are provided via the socket adapters; `DT_PROTECT`, in contrast, is a direct data input carried by the `UPDATE` event:

- `UP_IN.D1` (BOOL) – Upward direction activation (socket adapter).
- `DOWN_IN.D1` (BOOL) – Downward direction activation (socket adapter).

- DT_PROTECT (TIME, initial value T#50ms) – Protection dead time after releasing the active input (direct data input, carried by UPDATE).

Data Outputs

No direct data outputs. Provided via the plug adapters:

- UP_OUT.D1 (BOOL) – Up signal.
- DOWN_OUT.D1 (BOOL) – Down signal.
- TRIP_OUT.D1 (BOOL) – Trip signal.

Adapters

Sockets (Inputs)

Adapter	Type	Description
UP_IN	adapter::types::unidirectional::AX	Up input.
DOWN_IN	adapter::types::unidirectional::AX	Down-direction input.

Plugs (Outputs)

Adapter	Type	Description
UP_OUT	adapter::types::unidirectional::AX	Up-direction output.
DOWN_OUT	adapter::types::unidirectional::AX	Down-direction output.
TRIP_OUT	adapter::types::unidirectional::AX	Trip state output.
timeOut	iec61499::events::ATimeOut	Timer adapter for the protection time; the module sets timeOut.DT and starts it via timeOut.START.

Functionality

The circuit is structurally identical to ILOCK_CONFLICT_TRIP_PROTECT, except that all signals are routed via adapters:

1. **STOP** – Idle state. At UP_IN.E1[UP_IN.D1 AND NOT DOWN_IN.D1] ⟶ **UP** at DOWN_IN.E1[DOWN_IN.D1 AND NOT UP_IN.D1] ⟶ **DOWN** if both data signals are TRUE at either event ⟶ immediate **TRIP**.
2. **UP** – UP_OUT.D1 = TRUE. At UP_IN.E1[NOT UP_IN.D1] ⟶ **UP_STOP** at DOWN_IN.E1[DOWN_IN.D1] (conflict) ⟶ immediate **TRIP**.
3. **DOWN** – DOWN_OUT.D1 = TRUE. At DOWN_IN.E1[NOT DOWN_IN.D1] ⟶ **DOWN_STOP** at UP_IN.E1[UP_IN.D1] (conflict) ⟶ immediate **TRIP**.
4. **UP_STOP** / **DOWN_STOP** – The algorithm STOP sets all adapter outputs to FALSE, transmits DT_PROTECT to timeOut.DT, and starts the timer. At timeOut.TimeOut ⟶ **EVAL**.
5. **EVAL** – No separate algorithm. Re-evaluation: only UP_IN.D1 TRUE ⟶ **UP** Only DOWN_IN.D1 is TRUE ⟶ **DOWN** both are FALSE ⟶ **STOP** both are TRUE ⟶ **TRIP**.
6. **TRIP** – TRIP_OUT.D1 = TRUE. Exit only via EI_RESET[NOT UP_IN.D1 AND NOT DOWN_IN.D1] ⟶ **STOP**.

Additionally, in states STOP, UP, and DOWN, there is a self-loop transition to the event UPDATE, which does not leave the respective state and serves solely to acquire a new DT_PROTECT value.

Technical Features

- **Pure Adapter Interface:** All process data is routed via unidirectional::AX adapters (the protection-time timer additionally via an iec61499::events::ATimeOut adapter). There are no traditional event/data ports except for EI_RESET, UPDATE, and DT_PROTECT.
- **Dynamic Dead Time:** The UPDATE event allows DT_PROTECT to be modified at runtime without exiting the automaton. The new value takes effect upon the next entry into UP_STOP/DOWN_STOP.
- **Immediate Trip in Case of Conflict:** As with the non-adapter variant, a simultaneous command in both directions is detected without delay.

- **Reset Condition:** `EI_RESET` only takes effect if both input adapters report `D1 = FALSE`.

State Overview

State	UP_OUT.D1	DOWN_OUT.D1	TRIP_OUT.D1	Description
`STOP`	FALSE	FALSE	FALSE	Idle state, no direction active.
`UP`	TRUE	FALSE	FALSE	Up direction active.
`DOWN`	FALSE	TRUE	FALSE	Downward direction active.
`UP_STOP`	FALSE	FALSE	FALSE	Waiting for `DT_PROTECT` to expire after UP is released.
`DOWN_STOP`	FALSE	FALSE	FALSE	Waiting for `DT_PROTECT` to expire after DOWN is released.
`EVAL`	–	–	–	No algorithm; decides on the next state.
`TRIP`	FALSE	FALSE	TRUE	Conflict/Trip, requires `EI_RESET`.

Application Scenarios

- **Adapter-Based Drive Controls:** Modular systems where direction signals are already routed through the system via `unidirectional::AX` adapters.
- **Safety-Oriented Interlocking with Overrun Time:** Applications requiring both strict conflict detection with mandatory acknowledgement and a minimum pause between direction changes.
- **Runtime Parameterization:** Systems where the protection time must be adjusted via `UPDATE` depending on the operating state (e.g., temperature, load).

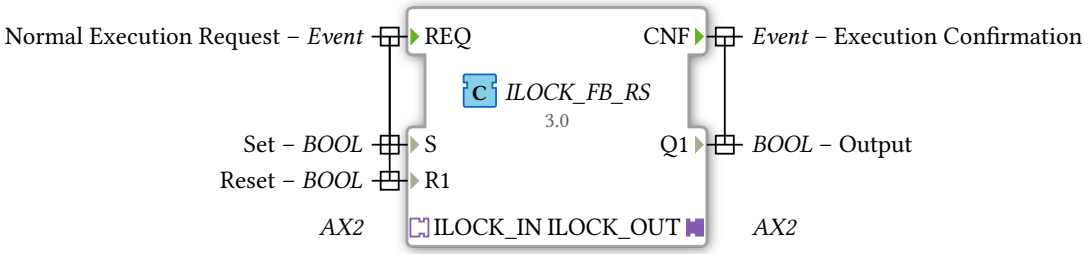
Comparison with similar building blocks

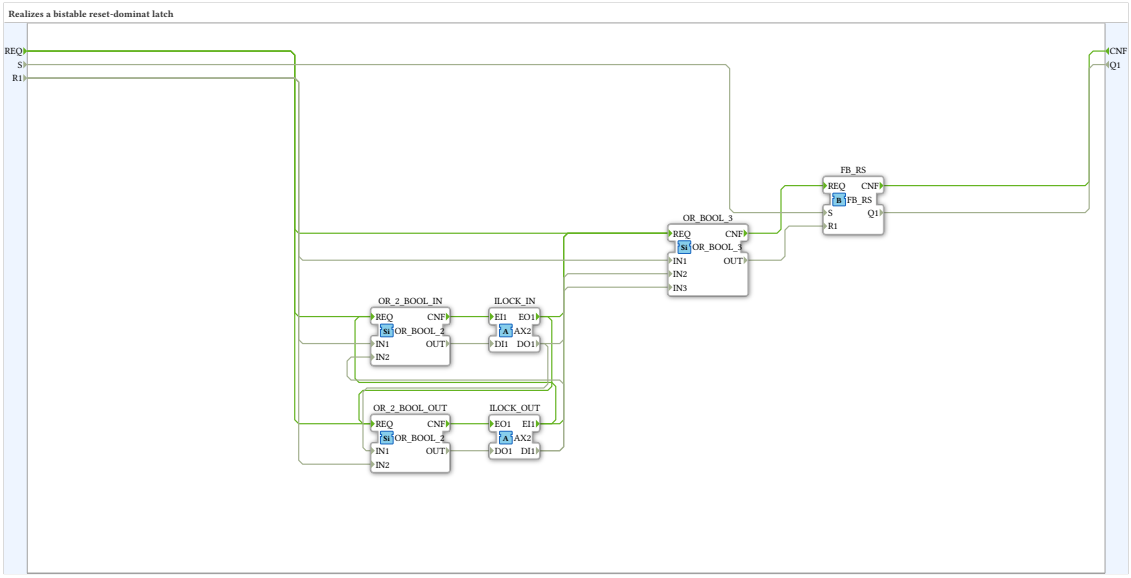
Compared to `ILOCK_CONFLICT_TRIP_AX`, this building block adds the states `UP_STOP`, `DOWN_STOP`, `EVAL`, as well as the `timeOut` adapter and the `UPDATE` event. Compared to `ILOCK_BLOCK_PROTECT_AX`, it differs in that a simultaneous command in both directions is not silently ignored, but rather treated as an explicit `TRIP` state, which requires an `EI_RESET` response. Compared to the non-adapter variant `ILOCK_CONFLICT_TRIP_PROTECT`, the interface is fully adapter-based, which facilitates integration into modular, adapter-oriented systems.

Conclusion

`ILOCK_CONFLICT_TRIP_PROTECT_AX` combines adapter-based conflict detection with mandatory acknowledgment and a configurable protection dead time in a single module. It is suitable for modular automation solutions that require strict mutual exclusivity of two directions combined with overrun time and runtime parameterization.

ILOCK_FB_RS





• – ○ • – ○ • – ○ •

Introduction

The function block `ILOCK_FB_RS` implements a bistable reset-dominant RS latch (reset-dominant flip-flop). It extends a simple RS flip-flop with a bidirectional coupling interface, enabling multiple such blocks to be chained together in an interlock chain. The output state is set by the Set input (S) and reset by the Reset input (R1) – also taking into account signals from adjacent blocks.

Interface Structure

Event Inputs

Event	Comment
`REQ`	Normal execution request (linked to data S and R1)

Event Outputs

Event	Comment
`CNF`	Execution Acknowledgement (linked to data output Q1)

Data Inputs

Variable	Type	Comment
`S`	BOOL	Set (active high)
`R1`	BOOL	Reset (active high) – Reset-dominant

Data Outputs

Variable	Type	Comment
`Q1`	BOOL	Flip-flop output

Adapter

Name	Direction	Type	Description
`ILOCK_IN`	Socket	bidirectional::AX2	Receives signals from the previous component in the chain (event EO1, data DO1)
`ILOCK_OUT`	Plug	bidirectional::AX2	Sends signals to the next component in the chain (event EI1, data DI1)

The adapters of type AX2 each provide two logic channels:

- Event channel: EI1 (input) / EO1 (output)
- Data channel: DI1 (input) / DO1 (output)

Functionality

Internally, the component consists of an RS flip-flop (FB_RS, reset-dominant), two OR gates, and a 3-OR gate. The logic links the external inputs to the adapter signals:

- **Set:** The input S goes directly to the set input of the internal flip-flop. With S active and the reset input inactive, Q1 = TRUE is used.
- **Reset:** The reset input R1 is connected to the data outputs of the two adapters (ILOCK_IN.DO1 and ILOCK_OUT.DI1) via a 3-factor OR gate. The result controls the reset input of the internal flip-flop. As soon as **one** of these three sources becomes active, the output Q1 is reset (reset dominance).

Reset dominance: - **Coupling:** The internal reset (R1) is additionally routed to the adapter outputs via OR gates:

- At ILOCK_OUT.DO1, R1 is ORed with the received signal from ILOCK_IN.DO1 (OR_2_BOOL_OUT), so that a reset is passed on.
- At ILOCK_IN.DI1, R1 is ORed with the received signal from ILOCK_OUT.DI1 (OR_2_BOOL_IN), so that a reset from the other side is also passed on.
- **Event Control:** All events (REQ, as well as the adapter events) are collected and forwarded to the internal flip-flop via OR logic. The acknowledgment (CNF) is output after the flip-flop has processed.

Technical Features

- **Reset Dominance:** In case of conflict (S and R1 active simultaneously), the reset takes precedence → Q1 = FALSE.
- **Cascading:** Multiple ILOCK_FB_RS units can be cascaded using the adapters. A reset occurring at one point propagates through the chain.
- **Bidirectional Coupling:** The adapters enable both forward and reverse communication, allowing the device to receive and forward set and reset signals from both sides.
- **Compatibility:** Implemented according to IEC 61499-1 and utilizes subapplications from the iec61131 library.

State Overview

The internal RS flip-flop has the following state table (reset-dominant):

S	R1_eff	Q1 (new)	Description
0	0	Q1_old	Hold
1	0	1	Set
0	1	0	Reset
1	1	0	Reset dominant

The output R1_eff is the result of the OR operation on R1, ILOCK_IN.DO1, and ILOCK_OUT.DI1. The output Q1 is only set if none of these three reset sources are active.

Application Scenarios

- **Interlock Chains** in safety controllers: Several components are chained together. A fault or reset at one point resets all subsequent outputs.
- **Priority Logic:** Reset-dominant latch structures where a specific event (e.g., emergency stop) takes precedence over set signals.
- **Signal Interlocking** in redundant systems where multiple conditions for setting and resetting must be combined.

Comparison with Similar Components

Component	Properties
FB_RS (IEC 61131)	Simple reset-dominant RS flip-flop without adapter, no cascading

ILOCK_FB_RS	Same core function, but with bidirectional adapters for cascading and additional reset signal linking

The ILOCK_FB_RS thus adds the ability to operate multiple such flip-flops in a chain, with a reset signal being passed from one link to the others.

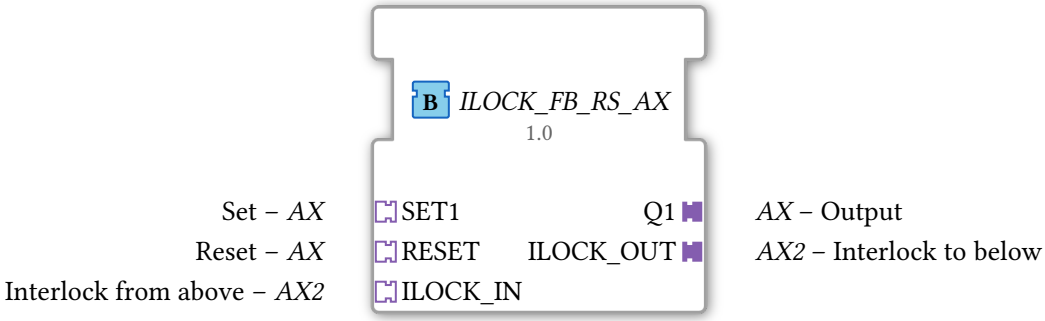
Conclusion

The ILOCK_FB_RS is a versatile component for implementing reset-dominant interlock logic. Its integrated bidirectional coupling makes it particularly suitable for modular, expandable safety and interlock circuits, eliminating the need for additional external logic.

ILOCK_FB_RS_AX



Introduction



The function block **ILOCK_FB_RS_AX** implements a bistable, reset-dominant RS latch with an AX interface for set and reset signals, as well as a bidirectional AX2 interface for chaining multiple interlock blocks in a hierarchy. The block is designed for use in safety-related and interlock-based control environments.

Interface Structure

The block does not have separate event or data ports, but rather encapsulates all signal types via **adapters**. The following list groups the events and data received via the individual adapters.

Event Inputs

- **SET1.E1** (via socket adapter `SET1`): Set event, triggers the latch function.
- **RESET.E1** (via socket adapter `RESET`): Reset event, triggers the reset logic.
- **ILOCK_IN.EO1** (via socket adapter `ILOCK_IN`): Event from a parent or neighboring interlock instance (bidirectional side).
- **ILOCK_OUT.EI1** (via plug adapter `ILOCK_OUT`): Event from a child interlock instance.

Event Outputs

- **Q1.E1** (via plug adapter `Q1`): Output event after each update of the latch state.
- **ILOCK_IN.EI1** (via socket adapter `ILOCK_IN`): Event sent to the connected partner adapter of `ILOCK_IN`.
- **ILOCK_OUT.EO1** (via plug adapter `ILOCK_OUT`): Event sent to the connected partner adapter of `ILOCK_OUT`.

Data Inputs

- **SET1.D1** (via socket adapter `SET1`): Set signal (Boolean value, TRUE sets the latch).
- **RESET.D1** (via socket adapter `RESET`): Reset signal (Boolean value, TRUE resets the latch, overrides Set).
- **ILOCK_IN.DO1** (via socket adapter `ILOCK_IN`): Interlock status from above (Boolean value, TRUE blocks the latch).
- **ILOCK_OUT.DI1** (via plug adapter `ILOCK_OUT`): Interlock status from below (Boolean value, TRUE blocks the latch).

Data Outputs

- **Q1.D1** (via plug adapter `Q1`): Latch output state (Boolean value).
- **ILOCK_IN.DI1** (via socket adapter `ILOCK_IN`): Forwards the reset signal upwards.
- **ILOCK_OUT.DO1** (via plug adapter `ILOCK_OUT`): Forwards the reset signal downwards.

Adapter

Adapter	Type	Direction	Description
`SET1`	Unidirectional AX	Socket	Set Input
`RESET`	Unidirectional AX	Socket	Reset Input
`ILOCK_IN`	Bidirectional AX2	Socket	Interlock Input (from top)
`Q1`	Unidirectional AX	Plug	Latch Output
`ILOCK_OUT`	Bidirectional AX2	Plug	Interlock Output (downward)

Functionality

The function block executes the algorithm `REQ` upon each incoming event (via `SET1.E1`, `RESET.E1`, `ILOCK_IN.EO1`, or `ILOCK_OUT.EI1`). The internal logic calculates the new output value `Q1.D1` using the reset-dominant formula:

Q1.D1 := (SET1.D1 OR Q1.D1) AND NOT (RESET.D1 OR ILOCK_IN.DO1 OR ILOCK_OUT.DI1);`

- If the reset signal (`RESET.D1`) or one of the interlock inputs (`ILOCK_IN.DO1` or `ILOCK_OUT.DI1`) is TRUE, the latch is immediately reset (Q = FALSE), regardless of the set signal.
- If no reset or interlock block is present, the latch is set by a set signal (`SET1.D1` = TRUE) and remains self-latching.
- The interlock signals of the AX2 interface are propagated:
- `ILOCK_IN.DI1` (upwards) = `RESET.D1 OR ILOCK_OUT.DI1`
- `ILOCK_OUT.DO1` (downwards) = `RESET.D1 OR ILOCK_IN.DO1`

This creates a cascade in which a reset signal is passed on in both directions (upwards and downwards), blocking all subordinate and superior latch components.

Technical Features

- **Adapter-based interfaces**: Instead of individual ports, the component uses standardized AX/AX2 adapters, enabling a modular and reusable connection with other components.

- **Cascadable Interlock Chain**: Multiple ILOCK_FB_RS_AX instances can be connected in series using the bidirectional `ILOCK_IN/ILOCK_OUT` adapters. A reset signal is automatically propagated in both directions, blocking the entire chain.
- **Reset Dominance**: The reset signal always takes precedence over the set signal. This meets typical requirements for safety interlocks that demand a defined priority for the reset signal.
- **Continuous Event Processing**: The function block reacts to every incoming event (set, reset, up/down interlock) and immediately updates all outputs. There are no state changes in the ECC (only one state, `REQ`); the logic is purely combinational with feedback via the stored latch value.

Reset Dominance:

State Overview

The component has only one state, `REQ`, in its ECC, which is always active. The internal latch state is represented by the boolean value `Q1.D1`.The state overview therefore describes the behavior of the internal memory cell:

Current Q	SET1.D1	RESET.D1	Interlock active?	Next Q	Description
FALSE	FALSE	FALSE	NO	FALSE	Latch remains reset
FALSE	TRUE	FALSE	NO	TRUE	Latch is set
FALSE	TRUE	TRUE	YES/NO	FALSE	Reset dominates
TRUE	FALSE	FALSE	NO	TRUE	Latch remains set (self-latching)
TRUE	TRUE	FALSE	NO	TRUE	Latch remains set
TRUE	FALSE	TRUE	YES/NO	FALSE	Reset dominates
TRUE	TRUE	TRUE	YES/NO	FALSE	Reset dominates
arbitrary	arbitrary	arbitrary	YES (ILOCK)	FALSE	Interlock blockage sets or holds latch

Application Scenarios

- **Machine Safety**: Interlocking drives or tools where a reset signal (e.g., emergency stop) takes precedence and also blocks adjacent modules.
- **Process Control with Hierarchy**: In a cascade of control modules, a higher-level signal can reset the subordinate blocks to their default state.
- **Redundant Interlocking**: Multiple ILOCK_FB_RS_AX instances can be arranged in a ring or line to create a distributed interlocking system without central logic.
- **Latch with External Blocking**: External conditions (e.g., sensor values or enable/disable) can block the latch via the interlock adapters without affecting the set/reset function.

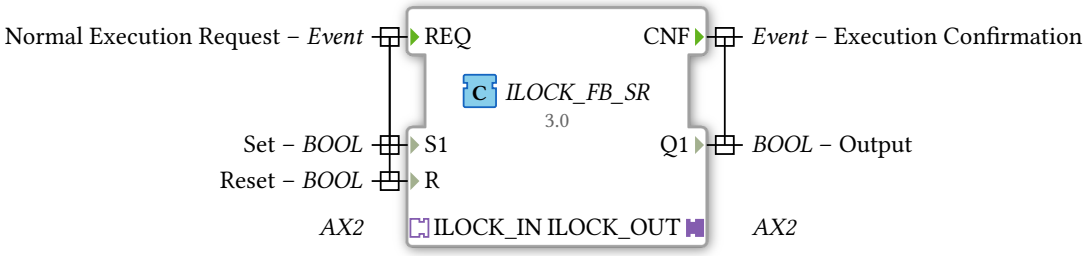
Comparison with Similar Function Blocks

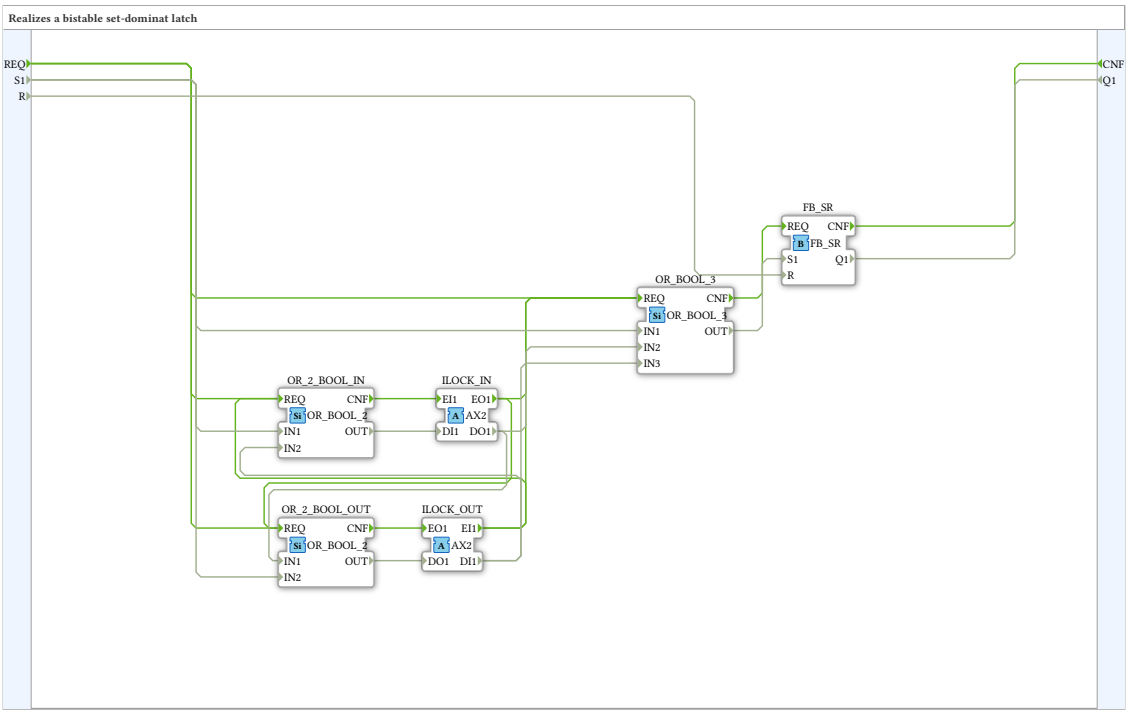
- **Standard RS Latch (e.g., SR_FF)**: A typical RS latch has equal set and reset inputs. The ILOCK_FB_RS_AX is reset-dominant and additionally offers an interlock function for cascading.
- **Reset-Dominant Latch without Interlock**: A simple RS latch with reset dominance only has set/reset ports. The function block described here extends this to include bidirectional interlocking via AX2 adapters.
- **Interlock Function Blocks with Dedicated Ports**: Some libraries offer interlock function blocks with individual Boolean inputs for “InterlockFromAbove” and “InterlockFromBelow.” The ILOCK_FB_RS_AX encapsulates these channels in adapters, thus simplifying the connection between instances.

Conclusion

The **ILOCK_FB_RS_AX** is a specialized function block for interlock-based control tasks. Its reset-dominant logic ensures a defined priority for the reset signal. The use of AX/AX2 adapters enables a clean, modular structure and easy cascading of multiple function blocks. This makes it particularly suitable for hierarchical or distributed safety and locking systems in automation technology.

ILOCK_FB_SR





• – ○ • – ○ • – ○ •

Introduction

The ILOCK_FB_SR function block implements a bistable set-dominant SR latch, specifically designed for use in interlock circuits. It extends a simple SR latch with bidirectional adapters for coupling with other function blocks, enabling the implementation of complex interconnections and priority logic.

Interface Structure

Event Inputs

Event	Description	Accompanying Data
`REQ`	Execution Request for Processing the Set and Reset Signals	`S1`, `R`

Event Outputs

Event	Description	Carrying Data
`CNF`	Execution Confirmation, outputs the current output value	`Q1`

Data Inputs

Name	Data Type	Description
`S1`	`BOOL`	Set Input

R	BOOL	Reset Input

Data Outputs

Name	Data Type	Description
Q1	BOOL	Latch Output (set-dominant)

Adapters

Adapter	Type	Direction	Description
ILOCK_IN	adapter::types::bidirectional::AX2	Socket	Receives data and events from an upstream interlock block
ILOCK_OUT	adapter::types::bidirectional::AX2	Plug	Sends data and events to a downstream interlock block

The adapters enable bidirectional chaining of interlock blocks by allowing both the reception of enable/lock signals and the transmission of states.

Functionality

The core of the block is an internal SR latch (FB_SR) that operates in a **set-dominant** manner: If both Set (S1) and Reset (R) are active simultaneously, the block sets the output Q1 to TRUE.

The logic is supplemented by three OR gates:

- **OR_3_BOOL** combines the signals from the Set input (S1) with the feedback signals from the adjacent Interlock devices (ILOCK_IN.DO1 and ILOCK_OUT.DI1). The result is fed to the Set input of the internal SR latch.
- **OR_2_BOOL_IN** combines the Set signal (S1) and the signal from the downstream Interlock device (ILOCK_OUT.DI1). The result is sent to the upstream device via the adapter ILOCK_IN.DI1.
- **OR_2_BOOL_IN** combines the Set signal (S1) and the signal from the downstream Interlock device (ILOCK_OUT.DI1). - **OR_2_BOOL_OUT** combines the set signal (S1) and the signal of the upstream interlock block (ILOCK_IN.DO1). The result is sent to the downstream block via the adapter ILOCK_OUT.DO1.

This structure allows multiple interlock blocks to be cascaded: A set signal can be blocked or forwarded by an upstream block. Simultaneously, the release of its own latch is reported to the adjacent blocks.

Technical Features

- **Set-Dominant Behavior**: In case of conflict (S1=TRUE, R=TRUE), the set signal takes precedence.
- **Adapter-Based Coupling**: The bidirectional adapters ILOCK_IN and ILOCK_OUT allow for modular interconnection without the need to manually establish direct data or event connections between the components.
- **Feedback**: The internal logic processes both external set and reset signals as well as the states of adjacent components, creating a dynamic interlocking chain.

State Overview

The internal SR latch has four states based on the (combined) inputs:

S1 (after logic)	R	Q1 (previous)	Q1 (new)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1

1	1	0	1
1	1	1	1

The effective values for S1 (Set) result from the OR operation of `S1`, `ILOCK_IN.DO1`, and `ILOCK_OUT.DI1`.

Application Scenarios

- **Safety Interlocks**: In systems where multiple conditions must be met simultaneously before a process is enabled (e.g., safety gates, emergency stop).
- **Cascaded Controls**: Superior and subordinate interlock units that interact via adapters (e.g., in production lines).
- **Priority-Based Sets**: Used in systems where a set signal must take precedence over a reset signal to enforce safety-critical states.

Comparison with Similar Devices

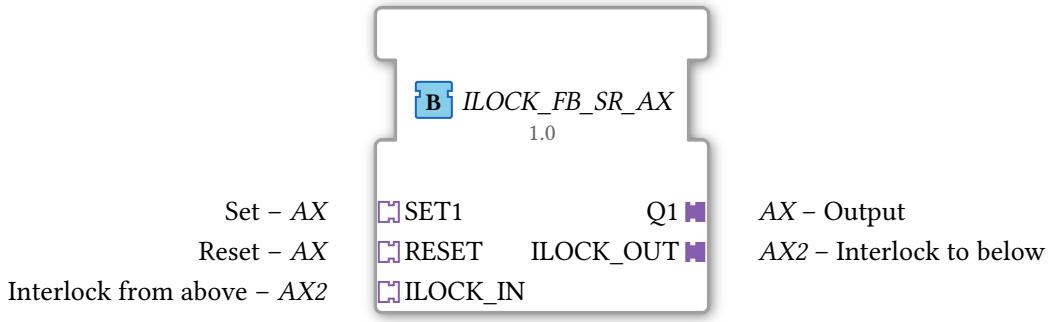
- **IEC 61131 SR (reset-dominant)**: Standard SR latch where the reset signal takes precedence in case of a conflict. The ILOCK_FB_SR reverses this priority.
- **IEC 61131 RS (set-dominant)**: Similar behavior at its core, but without the adapter-based interconnection and the logic for chaining multiple devices.
- **D Flip-Flop**: Clocked memory cell that only takes over on a rising edge – this is an asynchronous, clock-independent latch.

The ILOCK_FB_SR is specifically optimized for interlock chains, while simple SR latches operate in isolation.

Conclusion

The ILOCK_FB_SR offers a flexible and robust solution for set-dominant interlock circuits. By combining an SR latch with bidirectional adapters and OR logic, arbitrarily complex interconnections can be built without the need for extensive wiring of signals between the components. The set-dominant behavior is particularly advantageous in safety-critical applications where a set signal must take precedence.

ILOCK_FB_SR_AX





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ILOCK_FB_SR_AX** implements a set-dominant (set-priority) bistable flip-flop (RS latch) with interlock functionality. It is designed for use in safety and interlocking chains where a set signal can be propagated and fed back through multiple stages. The interface consists of two unidirectional AX adapters (SET1, RESET) and one AX2 bidirectional adapter pair (ILOCK_IN / ILOCK_OUT), as well as an output adapter Q1. The block operates in an event-driven manner and recalculates the output values with each incoming event.

Interface Structure

Event Inputs

Event	Data Source	Description
`SET1.E1`	Socket SET1	Sets output Q1 (set-dominant)
`RESET.E1`	Socket RESET	Resets output Q1 (only effective if SET1 = 0)
`ILOCK_IN.EO1`	Socket ILOCK_IN	Receives the propagation event from the parent interlock stage
`ILOCK_OUT.EI1`	Plug ILOCK_OUT	Receives an event from the child interlock stage (e.g., acknowledgment)

Event Outputs

Event	Data Source	Description
`Q1.E1`	Plug Q1	Output event after each Q1 update
`ILOCK_IN.EI1`	Socket ILOCK_IN	Sends propagation event to the parent stage
`ILOCK_OUT.EO1`	Plug ILOCK_OUT	Sends propagation event to the child stage

Data Inputs

Data	Type	Description
SET1.D1	BOOL	Set input (dominant)
RESET.D1	BOOL	Reset input (active when SET1=0)
ILOCK_IN.DO1`	BOOL	Data from the parent interlock stage (propagation signal)
ILOCK_OUT.DI1`	BOOL	Data from the child interlock stage (feedback)

Data Outputs

Data	Type	Description
Q1.D1`	BOOL	Latch output (set by Set or Interlock)
ILOCK_IN.DI1`	BOOL	Propagated Set signal to the parent stage
ILOCK_OUT.DO1	BOOL	Propagated Set signal to the subordinate stage

Adapter

Adapter	Type	Direction	Description
Q1	AX (unidirectional)	Plug	Latch output
ILOCK_OUT	AX2 (bidirectional)	Plug	Interlock interface to the subordinate stage
SET1	AX (unidirectional)	Socket	Set input
ax (unidirectional)	Socket	Reset input	
ILOCK_IN	AX2 (bidirectional)	Socket	Interlock interface to the higher-level interface

Functionality

The function block implements an algorithm that performs the following calculations on each incoming event:

1. **Q1.D1 (Latch Output):**

Q1.D1 := SET1.D1 OR ILOCK_IN.DO1 OR ILOCK_OUT.DI1 OR ((NOT RESET.D1) AND Q1.D1)`

- The output is set if **SET1.D1**, **ILOCK_IN.DO1** (from above), or **ILOCK_OUT.DI1** (from below) has the value TRUE.
- If no set signal is active and **RESET.D1** = **FALSE**, the current value of Q1 is retained (memory behavior).
- Set is dominant: An active set overrides a reset.

1. **ILOCK_IN.DI1 (Propagation Upwards):**

ILOCK_IN.DI1 := SET1.D1 OR ILOCK_OUT.DI1`

- The set signal is propagated upwards when either its own set input or the signal of the lower stage is active.

1. **ILOCK_OUT.DO1 (Propagation Downwards):**

ILOCK_OUT.DO1 := SET1.D1 OR ILOCK_IN.DO1`

- The set signal is propagated downwards when either its own set input or the signal of the upper stage is active.

After each calculation, the event outputs Q1.E1, ILOCK_IN.EI1, and ILOCK_OUT.EO1 are activated.

Technical Features

- **Interlock Cascade:** The function block can be used in a chain of multiple identical blocks. The AX2 adapter (ILOCK) propagates a set signal both upwards and downwards. This enables mutual interlocking across multiple stages.
- **Bidirectional Communication:** The AX2 adapter allows both sending and receiving events and data in both directions, enabling feedback from subordinate stages.
- **Set Dominance:** An active set input immediately sets the output and blocks a reset. This function block is therefore an **SR flip-flop with set signal priority**.
- **Pure Event Control:** The function block has no cyclical execution; every change is triggered only by a corresponding event.

State Overview

The function block has a single internal state, **REQ**, in which it always resides. The algorithm is executed upon each incoming event (SET1.E1, RESET.E1, ILOCK_IN.EO1, ILOCK_OUT.EI1), and then the system returns to the REQ state. There are no other states; the function block (FB) is essentially stateless (combinatorically with memory via feedback from Q1.D1).

Application Scenarios

- **Safety Interlock in Agricultural Engineering:** Cascading multiple ILOCK_FB_SR_AX functions to create a continuous interlock chain (e.g., for machine axes). A set signal from any point blocks all subsequent stages.
- **Production Systems with Release Chains:** Setting and resetting release signals in a hierarchical control system, where higher stages override lower ones.
- **Redundant Control Systems:** Utilizing the interlock function for mutual monitoring and interlocking of multiple control channels.

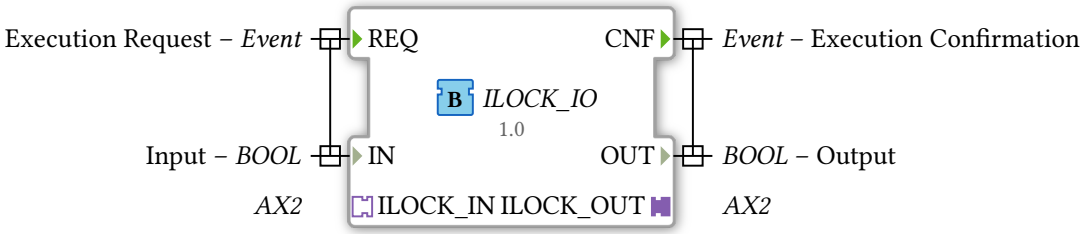
Comparison with Similar Function Blocks

- **Standard RS Flip-Flop (SR):** A simple SR function block has no interlock mechanism and no bidirectional data propagation. It can only be set or reset locally.
- **ILOCK_FB_SR_AX** extends the classic RS flip-flop with cascading capability via the AX2 interfaces. The dominance of the set signal and the bidirectional propagation make it particularly suitable for safety chains.
- **Monostable Function Blocks:** Unlike monostable functions, this function block retains its state until the next reset or set (bistable).

Conclusion

The ILOCK_FB_SR_AX is a specialized function block for building interlock chains with a set-dominant latch function. By combining unidirectional set/reset inputs and bidirectional interlock adapters, it is ideally suited for safety-related applications requiring cascading and mutual interlocking of multiple stages. Its simple, event-driven logic enables efficient and deterministic processing in automation technology.

ILOCK_IO





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **ILOCK_IO** implements a cascadable, momentary (non-resettable) interlock with a BOOL interface. It is used to ensure that, in a chain of multiple function blocks, only one block sets its output to TRUE at a time, as long as its input is active and no other block in the chain is already active. This function block is particularly suitable for controllers where competing requirements must be handled exclusively.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Execution Request (triggers the processing of the IN input)

Event Outputs

Event	Data Type	Comment
CNF	Event	Execution Confirmation (signals that the output OUT has been updated)

Data Inputs

Variable	Data Type	Comment
IN	BOOL	Input Value (Set Request)

Data Outputs

Variable	Data Type	Comment
OUT	BOOL	Output Value (Enabled/Disabled)

Adapters

Label	Type	Direction	Description
ILOCK_IN	adapter::types::bidirectional::AX2	Socket	Input adapter for receiving locking information from the chain (from the predecessor)
ILOCK_OUT	adapter::types::bidirectional::AX2	Plug	Output adapter for sending locking information to the chain (to the successor)

The adapters of type `AX2` are bidirectional and contain the data points `DI1` (Data Input 1) and `DO1` (Data Output 1). In the chain, `ILOCK_OUT.DO1` and `ILOCK_IN.DI1`, as well as `ILOCK_OUT.DI1` and `ILOCK_IN.DO1`, are connected in pairs to implement the interlock logic.

Functionality

The function block operates as a **momentary latch with interlock**. The following algorithm is executed each time the event `REQ` is called:

1. The output `OUT` is set to TRUE if the input `IN` is TRUE **and** no other function block in the chain is active. The activity of other blocks is queried via the adapters:

- `ILOCK_IN.DO1`: Signal from the predecessor (whether it is active)
- `ILOCK_OUT.DI1`: Signal from the successor (whether it is active)
- Logic: `OUT := IN AND NOT (ILOCK_IN.DO1 OR ILOCK_OUT.DI1)`

1. Subsequently, the current state is propagated to its neighbors via the adapters:

- `ILOCK_IN.DI1 := OUT` – writes the state back to the predecessor
- `ILOCK_OUT.DO1 := OUT` – writes the state to the successor

The chain is structured so that only one block can be active at a time. If `IN` goes to FALSE, `OUT` immediately goes to FALSE, and the interlock is released.

Technical Features

- **Cascading**: Any number of `ILOCK_IO` blocks can be connected in series using the bidirectional adapters. Each block checks whether its predecessor or successor is already active before it becomes active itself.
- **Instantaneous Behavior**: The output is only TRUE as long as the input is active. There is no memory function – the block is not a flip-flop.
- **Exclusive Release**: Even with multiple simultaneous set requests, only one block can be activated. The priority is determined by the order of the `REQ` events (the one processed first blocks the others).

State Overview

The block has a single **algorithmic state** `REQ`. Each time `REQ` is called, the algorithm is executed, and the system immediately transitions back to the same state. There is no state storage – the logic is purely combinatorial with feedback via the adapters.

State	Action	Output
REQ	Execute algorithm `REQ`	`CNF` is sent after calculation

Application Scenarios

- **Traffic light control with multiple request buttons**: Each button is connected to a `ILOCK_IO`; only the first button pressed receives the enable signal.
- **Exclusive machine control**: Multiple control panels compete for control of a machine – only the active panel receives the authorization.
- **Safety Interlock**: In a conveyor system, several safety doors block each other; only one door can be open at a time.

Comparison with Similar Components

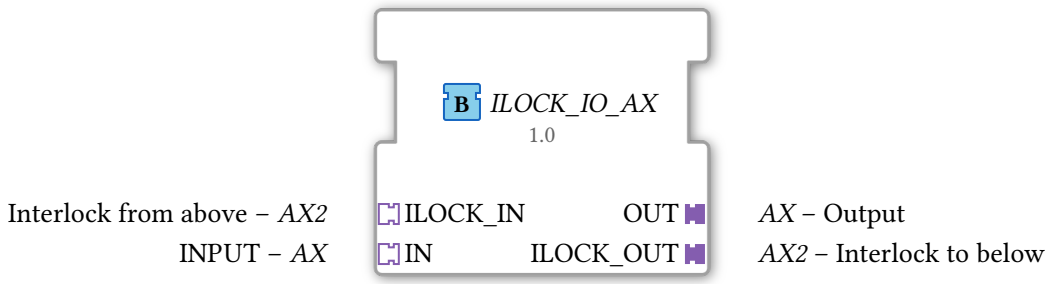
- **SR Flip-Flop (`SR`)** – stores a state, `ILOCK_IO` does not store (instantaneously).
- **Monostable Flip-Flop (`RTRIG`, `FTRIG`)** – reacts only to edges, `ILOCK_IO` operates on a level-controlled basis.

- **Simple `AND` Gate Component** – only local logic, no networking with other components. `ILOCK_IO` offers this networking via adapters.

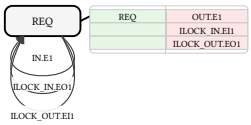
Conclusion

The `ILOCK_IO` function block is a specialized block for implementing exclusive release chains. Its simple interface and cascadability make it ideal for modular controllers where multiple competing requirements need to be managed. The bidirectional adapters enable flexible and expandable wiring.

ILOCK_IO_AX



ECC: ILOCK_IO_AX



• – ○ • – ○ • – ○ •

Introduction

The function block **ILOCK_IO_AX** implements a chainable momentary latch with interlock functionality. It ensures that only one element in a chain is active at any given time – as soon as another function block in the chain becomes active, it is reset. The function block is specifically designed for use with AX/AX2 adapter interfaces.

Interface Structure

The block does not have any traditional event or data inputs/outputs at the function block level. All communication takes place via **adapter interfaces**. The signals transmitted via the adapters are detailed below.

Event Inputs

Signal	Adapter	Description
`IN.E1`	IN (AX)	Trigger for processing the input signal.
`ILOCK_IN.EO1`	ILOCK_IN (AX2)	Event from the parent function block in the chain (e.g., when a different output becomes active there).
`ILOCK_OUT.EI1`	ILOCK_OUT (AX2)	Event from the child function block in the chain (e.g., when it becomes active).

Event Outputs

Signal	Adapter	Description
`OUT.E1`	OUT (AX)	Confirmation that processing is complete.
`ILOCK_IN.EI1`	ILOCK_IN (AX2)	Event sent to the parent FB (sent with each processing cycle).
`ILOCK_OUT.EO1`	ILOCK_OUT (AX2)	Event sent to the child FB (sent with each processing cycle).

Data Inputs

Signal	Adapter	Description
`IN.D1`	IN (AX)	Set input (BOOL) – if TRUE and no other FB in the chain is active, the output becomes active.
`ILOCK_IN.DO1`	ILOCK_IN (AX2)	State of the parent FB (TRUE means that the output is active there).
`ILOCK_OUT.DI1`	ILOCK_OUT (AX2)	State of the child function block (TRUE means that its output is active).

Data Outputs

Signal	Adapter	Description
`OUT.D1`	OUT (AX)	Output signal (BOOL) – active when `IN.D1` is set and no other function block in the chain is active.
`ILOCK_IN.DI1`	ILOCK_IN (AX2)	Passes its own output signal to the parent function block.
`ILOCK_OUT.DO1`	ILOCK_OUT (AX2)	The output signal is passed on to the subordinate function block.

Adapter

Adapter	Direction	Type	Description
`IN`	Socket (Input)	`adapter::types::unidirectional::AX`	Input signal from the user (set input and event).
`OUT`	Plug (Output)	`adapter::types::unidirectional::AX`	Output signal to the user (output value and confirmation).
`ILOCK_IN`	Socket (Input)	`adapter::types::bidirectional::AX2`	Connection to the **parent** function block in the chain.
`ILOCK_OUT`	Plug (Output)	`adapter::types::bidirectional::AX2`	Connection to the **subordinate** FB in the chain.

Functionality

The FB operates as a **momentary switch with interlock** (also known as an "exclusive latch"). In state `REQ`, the following logic is executed:

- The output `OUT.D1` is only **TRUE** if the input `IN.D1` is active **and** no other FB in the chain is currently active (`ILOCK_IN.DO1` from the predecessor and `ILOCK_OUT.DI1` from the successor must be FALSE).
- The active state of the function block (`OUT.D1`) is passed on via the adapters `ILOCK_IN.DI1` (to the predecessor) and `ILOCK_OUT.DO1` (to the successor).
- Each triggering event input (whether from `IN.E1`, `ILOCK_IN.EO1`, or `ILOCK_OUT.EI1`) leads to a recalculation of the logic and the output of all events (`OUT.E1`, `ILOCK_IN.EI1`, `ILOCK_OUT.EO1`).

Due to the bidirectional state transmission via the ILOCK adapters, all function blocks in the chain are aware of each other, so only one can be active at a time.

Technical Features

- **Chainable Interlock Structure**: Multiple ILOCK_IO_AX function blocks can be chained together by connecting `ILOCK_OUT` of one function block to `ILOCK_IN` of the next. The chain can be of any length.
- **No Own Events/Data at the Function Block Level**: All signal transmission takes place via the standardized AX/AX2 adapters. This allows for easy integration into existing adapter-based systems.
- **Only One Processing at a Time**: Since only one state (`REQ`) exists, the same logic is executed for each event. There are no internal state changes.

State Overview

The function block has only one state:
+---+ | REQ | (Standardzustand) +---+ | ^ | | (Ereignisse: IN.E1, ILOCK_IN.EO1, ILOCK_OUT.EI1) +-+ In state `REQ`, the algorithm `REQ` and the event outputs are executed. This state is never exited.

Application Scenarios

- **Control of switching devices** (e.g., valves, motors) in a line where only one may be active at a time.
- **Non-priority selection** from multiple sources (e.g., a push-button panel with interlock).
- **Chained safety controllers** where activating a subsequent stage must deactivate the previous one (or vice versa).

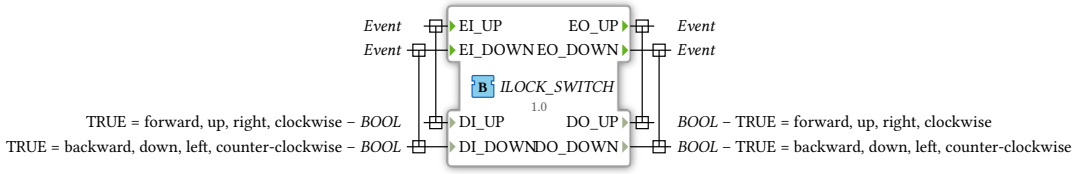
Comparison with Similar Function Blocks

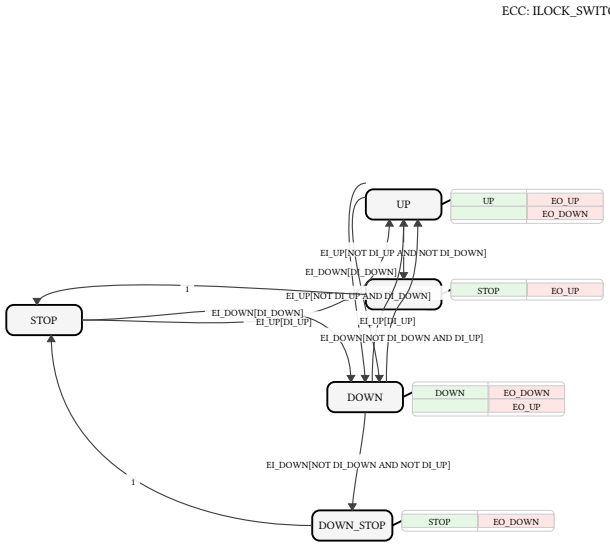
Function Block	Description	Difference to ILOCK_IO_AX
SR Flip-Flop	Sets output on SET, resets on RESET.	No interlock – multiple function blocks (FBs) can be active simultaneously.
Simple Latch (e.g., LATCH_AX)	Holds the value until triggered again.	No coupling with other FBs.
ILOCK_IO_AX	Exclusive activation in a chain.	Enforces that only one FB is active – ideal for line control systems.

Conclusion

The **ILOCK_IO_AX** is a specialized function block for exclusive activation in chained configurations. By using AX/AX2 adapters and bidirectional status feedback, robust interlock logic is implemented without separate wiring. It is particularly suitable for modular control structures where multiple actuators may only be active sequentially or individually.

ILOCK_SWITCH





• – ○ • – ○ • – ○ •

Introduction

The function block **ILOCK_SWITCH** serves as a priority-controlled switch with an interlock function. It evaluates two event signals (**EI_UP** and **EI_DOWN**) in combination with associated Boolean data values and sets the outputs **DO_UP** and **DO_DOWN** accordingly. The last active input is always prioritized – simultaneous activation of both outputs is impossible. The logic prevents oscillations and ensures a defined state even with invalid or conflicting input assignments.

Interface Structure

Event Inputs

Event	with Data	Description
EI_UP	DI_UP	Event for requesting the up/forward direction.
EI_DOWN	DI_DOWN	Event for requesting the down/backward direction.

Event Outputs

Event	with data	Description
EO_UP	DO_UP	Triggered when the up direction is activated or when exiting the UP state.
EO_DOWN	DO_DOWN	Triggered when the down direction is activated or when exiting the DOWN state.

Data Inputs

Name	Type	Comment
DI_UP	BOOL	TRUE = forward, up, right, clockwise
DI_DOWN	BOOL	TRUE = backward, down, left, counterclockwise

Data Outputs

Name	Type	Comment
DO_UP	BOOL	TRUE = forward, up, right, clockwise
DO_DOWN	BOOL	TRUE = backward, down, left, counterclockwise

Adapter

None.

Functionality

This component implements a finite automaton with six states. The basic idea: **The last valid instruction received sets the output state.** A command is only considered valid if the corresponding Boolean data value is **TRUE**.

- **STOP** (Idle State): Both outputs are **FALSE**.
- **UP**: DO_UP = TRUE, DO_DOWN = FALSE.

This is reached when **EI_UP** occurs with **DI_UP = TRUE** while the STOP state is in the STOP state.

- **DOWN**: DO_UP = FALSE, DO_DOWN = TRUE.

This is reached when **EI_DOWN** occurs with **DI_DOWN = TRUE** while the STOP state is in the STOP state.

- **UP_STOP**: An intermediate state that occurs when **EI_UP** occurs during the UP state, but **DI_UP = FALSE** (and also **DI_DOWN = FALSE**).

In UP_STOP, both outputs are set to FALSE, and the system immediately returns to STOP.

- **DOWN_STOP**: Analogous to UP_STOP for the DOWN state when **EI_DOWN** is **DI_DOWN = FALSE**.
- Direct switching is possible:

From **UP → DOWN** when **EI_DOWN** is **DI_DOWN = TRUE** or **EI_UP** is **DI_UP = FALSE** and **DI_DOWN = TRUE** (implicit request for the opposite direction).

From **DOWN → UP** analogously.

This logic ensures that only one direction is active at any given time and that the module cleanly returns to its idle state when invalid signals (FALSE values) are received.

Technical Features

- **Interlocking via state machines**: It is not possible to set both outputs to TRUE simultaneously – even with simultaneous events, switching occurs sequentially.
- **Intermediate States (UP_STOP, DOWN_STOP)**: These prevent unwanted flickering of the outputs by immediately resetting them and transitioning to STOP upon receiving invalid signals.
- **Implicit Switching**: An event whose data value requests the opposite direction (e.g., EI_UP with DI_UP=FALSE and DI_DOWN=TRUE) directly causes the direction change without requiring a separate EI_DOWN event.

State Overview

State	DO_UP	DO_DOWN	Achieved by
STOP	FALSE	FALSE	Start / after UP_STOP and DOWN_STOP
UP	TRUE	FALSE	EI_UP with DI_UP=TRUE from STOP or DOWN
DOWN	FALSE	TRUE	EI_DOWN with DI_DOWN=TRUE from STOP or UP
UP_STOP	FALSE	FALSE	EI_UP with DI_UP=FALSE and DI_DOWN=FALSE in the UP state
DOWN_STOP	FALSE	FALSE	EI_DOWN with DI_DOWN=FALSE and DI_UP=FALSE in the DOWN state

The transitions between states always occur via an incoming event and the evaluation of the current data values.

Application Scenarios

- **Motor control for linear actuators, swivel arms, or lifting devices**, where simultaneous movement in both directions must be prevented mechanically or for safety reasons.
- **Interlocking of valve actuators** (open/close) with feedback via limit switches.
- **Operator interfaces with pushbuttons for "Up" and "Down"**, where the last button press takes precedence and permanent blockage is avoided.

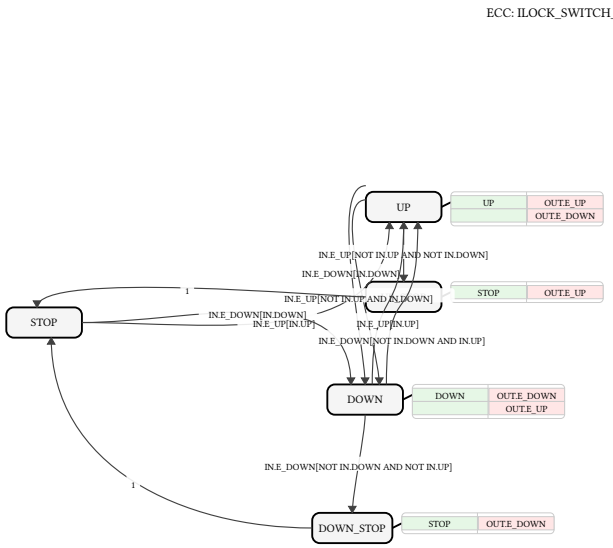
Comparison with Similar Function Blocks

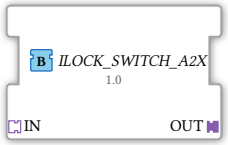
- A simple **AND/OR gate** cannot implement interlocking and would activate both outputs with simultaneous TRUE signals.
- An **RS flip-flop** stores one state, but may allow both set/reset inputs to be activated simultaneously (metastable). ILOCK_SWITCH avoids this through its rigid transition logic.
- A **mux** would only pass through one data value, but cannot implement event-driven prioritization with intermediate states. ILOCK_SWITCH offers a solution specifically optimized for drive interlocks.

Conclusion

The **ILOCK_SWITCH** is a robust, event-driven function block for interlocked direction control. The combination of a state machine, data dependency, and explicit intermediate states reliably prevents typical problems such as simultaneous activation of both outputs or oscillations. It is particularly suitable for safety-critical controls in automation technology, where a clear and resettable priority logic is required.

ILOCK_SWITCH_A2X





Input for forward/up and backward/down direction – *A2X* *A2X* – Output for forward/up and backward/down direction

• — ○ • — ○ • — ○ •

Introduction

The `ILOCK_SWITCH_A2X` is an event-driven basic function block that implements a two-directional interlock switch for an A2X adapter signal. It accepts UP/DOWN requests through an `IN` adapter and forwards

the interlocked result through an OUT adapter. Only one direction can be active at a time. If both directions are requested, the last active input wins.

Interface Structure

The FB does not expose ordinary IEC 61499 event or data inputs/outputs. All control information is exchanged through two A2X adapter instances.

Event Inputs

None at the FB interface.

The FB receives events through the IN adapter:

- IN.E_UP — up-direction event, evaluated together with IN.UP.
- IN.E_DOWN — down-direction event, evaluated together with IN.DOWN.

Event Outputs

None at the FB interface.

The FB sends events through the OUT adapter:

- OUT.E_UP — up-direction output event, accompanied by OUT.UP.
- OUT.E_DOWN — down-direction output event, accompanied by OUT.DOWN.

Data Inputs

None at the FB interface.

The IN adapter supplies the boolean data:

- IN.UP — indicates whether the up direction is requested.
- IN.DOWN — indicates whether the down direction is requested.

Data Outputs

None at the FB interface.

The OUT adapter supplies the boolean data:

- OUT.UP — TRUE while the up direction is active.
- OUT.DOWN — TRUE while the down direction is active.

Adapters

Adapter	Direction	Type	Description
IN	Socket	adapter::types::unidirectional::A2X	Input for forward/up and backward/down direction.
OUT	Plug	adapter::types::unidirectional::A2X	Output for forward/up and backward/down direction.

Functionality

The FB is a basic FB controlled by an ECC. It evaluates the A2X input events and data and produces an interlocked A2X output.

- In the STOP state, both OUT.UP and OUT.DOWN are FALSE; no direction is active.
- If an IN.E_UP event arrives with IN.UP = TRUE, the FB enters UP and activates OUT.UP.
- If an IN.E_DOWN event arrives with IN.DOWN = TRUE, the FB enters DOWN and activates OUT.DOWN.
- If a direction event arrives that requests the other direction while one direction is active, the FB switches to the other direction. This ensures last-active-input priority.
- If the active direction is released and no other direction is requested, the FB passes through a stop state and returns to STOP.
- If the active direction is released but the other direction is still requested, the FB switches directly to the remaining direction.

The algorithms used by the states are:

Algorithm	OUT.UP	OUT.DOWN
UP	TRUE	FALSE

DOWN	FALSE	TRUE
STOP	FALSE	FALSE

The output events are generated together with these data values, so a receiving block can react both to a direction change and to the release of the opposite direction.

Technical Features

- IEC 61499 basic FB with ECC-based, event-driven behavior.
- No external event/data inputs or outputs; all signals are transported by the unidirectional A2X adapters.
- Interlocked output: `OUT.UP` and `OUT.DOWN` can never be `TRUE` at the same time.
- Last-event priority when both input directions are active.
- Automatic stop handling through transient `UP_STOP` and `DOWN_STOP` states.
- Suitable for integration in `logiBUS::signalprocessing::interlock` signal-processing chains.

State Overview

State	Output Data	Output Events	Meaning
STOP	OUT.UP = FALSE, OUT.DOWN = FALSE	—	No direction active.
UP	OUT.UP = TRUE, OUT.DOWN = FALSE	OUT.E_UP, OUT.E_DOWN	Up direction active.
DOWN	OUT.UP = FALSE, OUT.DOWN = TRUE	OUT.E_DOWN, OUT.E_UP	Down direction active.
UP_STOP	OUT.UP = FALSE, OUT.DOWN = FALSE	OUT.E_UP	Transient stop from up direction.
DOWN_STOP	OUT.UP = FALSE, OUT.DOWN = FALSE	OUT.E_DOWN	Transient stop from down direction.

The ECC transitions are:

From	Condition	To
STOP	IN.E_UP[IN.UP]	UP
STOP	IN.E_DOWN[IN.DOWN]	DOWN
UP	IN.E_DOWN[IN.DOWN]	DOWN
UP	IN.E_UP[NOT IN.UP AND IN.DOWN]	DOWN
UP	IN.E_UP[NOT IN.UP AND NOT IN.DOWN]	UP_STOP
DOWN	IN.E_UP[IN.UP]	UP
DOWN	IN.E_DOWN[NOT IN.DOWN AND IN.UP]	UP
DOWN	IN.E_DOWN[NOT IN.DOWN AND NOT IN.UP]	DOWN_STOP
UP_STOP	1	STOP
DOWN_STOP	1	STOP

Application Scenarios

- Direction control of hydraulic or electric actuators that must not run in two directions simultaneously.
- Interlocking of forward/backward or up/down control signals on a logiBUS A2X connection.
- Conversion of a raw A2X input into a clean, interlocked A2X output for downstream control logic.
- Use as an adapter-based replacement for a conventional two-input interlock FB where UP/DOWN signals are fed from a single A2X input/output.

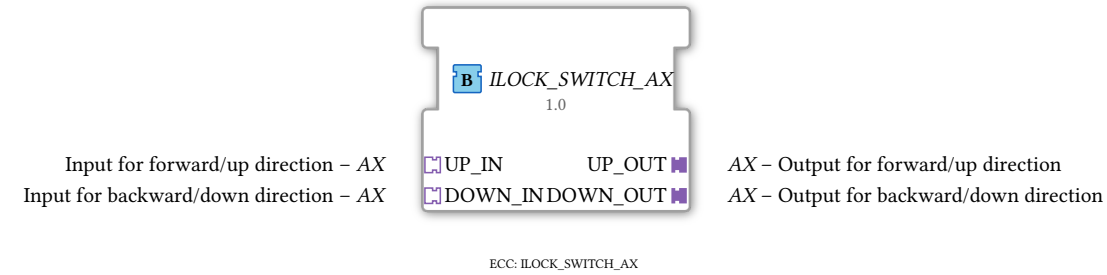
Comparison with Similar Blocks

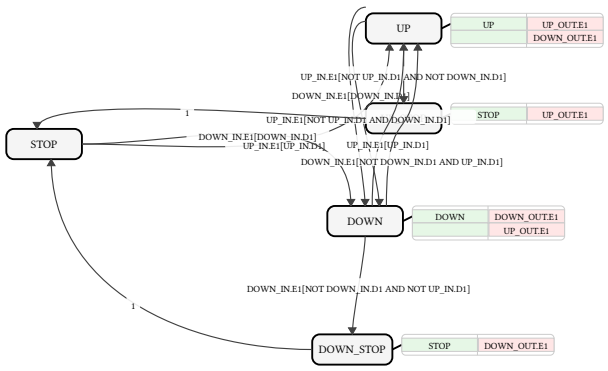
- Compared with a non-adapter `ILOCK_SWITCH`, this variant uses an A2X adapter for both input and output. This simplifies wiring when the surrounding system already uses A2X signals.
- Compared with an SR latch, this FB provides two separate direction outputs with associated events and an explicit stop state. It does not use a fixed set/reset priority; instead, the last active input wins.
- Compared with a fixed-priority direction selector, this FB can react to event/data combinations that indicate “one direction released, other still active” and switch to the remaining direction.

Conclusion

`ILOCK_SWITCH_A2X` is a compact, adapter-based interlock element for two-direction control signals. It combines a simple state machine with A2X adapter input/output, provides last-active-input prioritization, and guarantees that the output never activates both directions at the same time. It is especially useful in logiBUS signal-processing applications where UP/DOWN commands are transported over a single A2X connection.

ILOCK_SWITCH_AX





(No image available)

• – ○ • – ○ • – ○ •

Introduction

The function block **ILOCK_SWITCH_AX** implements interlocking switch control for bidirectional axes. It prioritizes the last active input and ensures that both directions are never activated simultaneously. Communication occurs exclusively via adapter interfaces of type `adapter::types::unidirectional::AX`. This version is specifically designed for use in modular control systems with standardized adapters.

Interface Structure

Event Inputs
None (all events are transmitted via the adapter sockets).

Event Outputs
None (all events are transmitted via the adapter plugs).

Data Inputs
None (all data is transmitted via the adapter sockets).

Data Outputs
None (all data is transmitted via the adapter plugs).

Adapters

Name	Type	Direction	Comment
UP_IN	adapter::types::unidirectional::AX	Socket	Input for forward/upward direction
DOWN_IN	adapter::types::unidirectional::AX	Socket	Input for reverse/downward direction
UP_OUT	adapter::types::unidirectional::AX	Plug	Output for forward/upward direction
DOWN_OUT	adapter::types::unidirectional::AX	Plug	Output for reverse/downward direction

Each adapter provides an event output `E1` and a data input/output `D1` (of type BOOL). The sockets receive requests, and the plugs pass on the switched states.

Functionality
The logic is implemented using a finite state machine (ECC). The function block is initially in state `STOP`. A change to the states `UP` or `DOWN` occurs when the corresponding socket returns an event `E1` with the data value `D1` set (i.e., `UP_IN.E1[UP_IN.D1]` or `DOWN_IN.E1[DOWN_IN.D1]`).

- In the state `UP`, `UP_OUT.D1` is set to `TRUE` and `DOWN_OUT.D1` is set to `FALSE`.
- In the state `UP`, `UP_OUT.D1` is set to `TRUE` and `DOWN_OUT.D1` to `FALSE`.
- In state `DOWN`, `DOWN_OUT.D1` is set to `TRUE` and `UP_OUT.D1` to `FALSE`.

A change from the current state to the other is only possible when the other socket becomes active (prioritization of the last active input). The conditions are differentiated as follows:

- **UP → DOWN**: either directly via `DOWN_IN.E1[DOWN_IN.D1]` or when the current input is deactivated and the other is active (`UP_IN.E1[NOT UP_IN.D1 AND DOWN_IN.D1]`).
- **DOWN → UP**: analogously via `UP_IN.E1[UP_IN.D1]` or `DOWN_IN.E1[NOT DOWN_IN.D1 AND UP_IN.D1]`.

If the active input is lost without the other becoming active, an intermediate state (`UP_STOP` or `DOWN_STOP`) is reached. In this state, the outputs are set to `FALSE`, and then the state machine immediately transitions to the `STOP` state.

Technical Features
- **Adapter-based interface**: The function block uses only adapters of type `unidirectional::AX`, which enables clean encapsulation of event and data communication and facilitates reuse in different contexts.
- **Prioritization of the last active input**: Unlike a simple state machine that only reacts to rising edges, this system considers the last valid signal sent.
- **Secure Interlock**: The ECC structure prevents both outputs from becoming `TRUE` simultaneously.
- **Explicit Stop Intermediate States**: The states `UP_STOP` and `DOWN_STOP` ensure a defined response to the disappearance of a control signal before returning to the idle state.

State Overview

State	UP_OUT.D1	DOWN_OUT.D1	Description
`STOP`	FALSE	FALSE	Idle state, no direction active
`UP`	TRUE	FALSE	Forward/Upward Direction Active
DOWN`	FALSE	TRUE	Backward/Downward Direction Active
UP_STOP`	FALSE	FALSE	Intermediate State After Exiting UP
DOWN_STOP`	FALSE	FALSE	Intermediate State After Exiting DOWN

Application Scenarios

- **Control of lifting/lowering drives** (e.g., hoists, hydraulic cylinders) – prevents simultaneous lifting and lowering.
- **Interlocking of switches** in conveyor systems – prioritizes the most recent request.
- **Safety-oriented switching** in modular automation systems using an adapter-based data model (e.g., logiBUS).

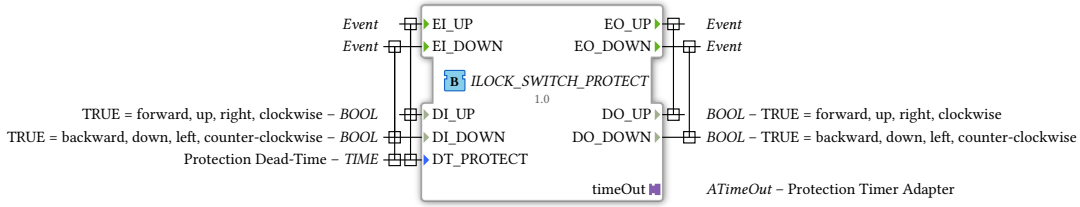
Comparison with similar function blocks

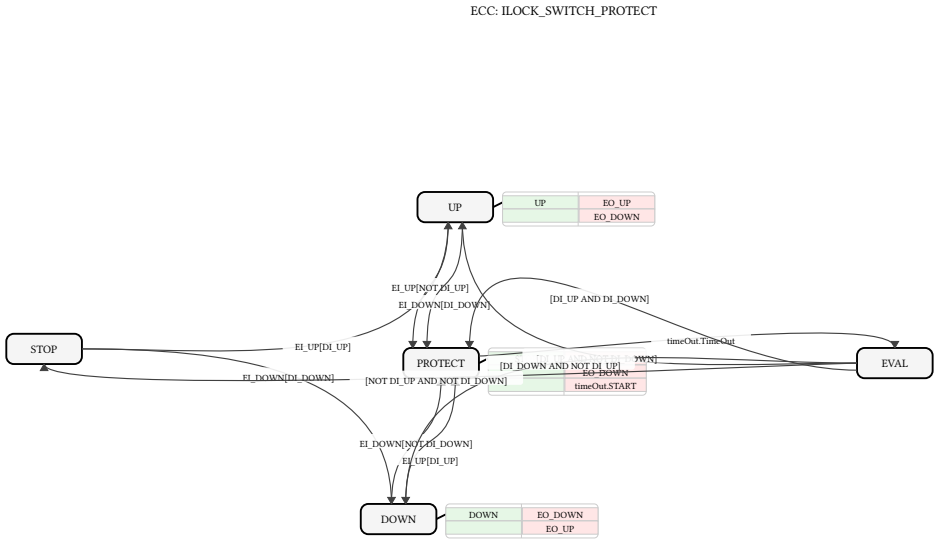
- **ILOCK_SWITCH** (without adapter): Offers similar interlock logic but uses direct event and data ports. The adapter version `ILOCK_SWITCH_AX` allows for easier coupling to standardized interfaces and is better suited for hierarchical architectures.
- **SR-Flipflop** (Set-Reset): Can also lock two states, but always prioritizes the set or reset input (depending on the implementation) and not the last active one.

Conclusion

The function block `ILOCK_SWITCH_AX` offers robust, adapter-based interlocking for bidirectional controllers. Due to the state automation, only one direction is active at a time, with priority assigned to the last received command. The clean separation of event and data channels, as well as the defined stop states, make it a reliable component in safety-critical applications.

ILOCK_SWITCH_PROTECT





• – ○ • – ○ • – ○ •

Introduction

The **ILOCK_SWITCH_PROTECT** function block implements a prioritized interlock between two switching directions – e.g., up/down, forward/backward, or right/left. It features a configurable dead-time delay that prevents immediate re-switching after a change of direction. The block evaluates the two binary inputs and outputs a switching signal only if the last active input is still active after the dead-time has expired. This prevents momentary oscillations and protects connected actuators.

Interface Structure

Event Inputs

Name	With Parameters	Description
EI_UP	`DI_UP`, `DT_PROTECT`	Event requesting the “Up” direction
EI_DOWN	`DI_DOWN`, `DT_PROTECT`	Event requesting the “Down” direction

Event Outputs

Name	With Parameters	Description
EO_UP	`DO_UP`	Confirmation that the “Up” direction has been activated
EO_DOWN	`DO_DOWN`	Confirmation that the “Down” direction has been activated

Data Inputs

Name	Data Type	Initial Value	Description
DI_UP	BOOL	–	TRUE = forward, up, right, clockwise
DI_DOWN	BOOL	–	TRUE = backward, down, left, counterclockwise
DT_PROTECT	TIME	T#50ms	Protection delay (dead time) before a change of direction

Data Outputs

Name	Data Type	Description
DO_UP	BOOL	TRUE = Output active for direction “Up”
DO_DOWN	BOOL	TRUE = Output active for direction “Down”

Adapter

Name	Type	Description
timeOut	`iec61499::events::ATimeOut`	Adapter for implementing the protection delay

Functionality

The module operates as a finite state machine with five states:

1. **STOP** – Idle state. Both outputs are FALSE.

- Upon an event with a valid input (`EI_UP[DI_UP]` or `EI_DOWN[DI_DOWN]`), the module switches directly to the corresponding state.

1. **UP** – The “Up” direction is active. `DO_UP = TRUE`, `DO_DOWN = FALSE`.

- A repeat `EI_UP` with an inactive request (`NOT DI_UP`) or a `EI_DOWN` with an active request (`DI_DOWN`) triggers the protection state.

1. **DOWN** – The “Down” direction is active. `DO_UP = FALSE`, `DO_DOWN = TRUE`.

Similarly: `EI_DOWN[NOT DI_DOWN]` or `EI_UP[DI_UP]` initiates the protection state.

1. **PROTECT** – Protection state. Both outputs are immediately reset (FALSE) and the timer of the adapter `timeOut` is started.

- Only after the configured time (`DT_PROTECT`) has elapsed does the evaluation state begin.

1. **EVAL** – Evaluation after the protection time.

- Based on the current inputs, the following is determined:
- `DI_UP AND NOT DI_DOWN` ⟶ **UP**
- `DI_DOWN AND NOT DI_UP` ⟶ **DOWN**
- `NOT DI_UP AND NOT DI_DOWN` ⟶ **STOP**
- `DI_UP AND DI_DOWN` (both active simultaneously) ⟶ **PROTECT** again (invalidation case)

The event outputs are triggered along with the respective executed algorithm. In the UP, DOWN, and PROTECT states, **both** event outputs (EO_UP and EO_DOWN) are sent sequentially – this is a specific behavior of this function block.

Technical Features

- **Adapter-Based Timer Implementation**: The protection delay is implemented using the standardized IEC61499 adapter `ATimeOut`. This makes the time control platform-independent and allows it to be used in various runtime environments.
- **Dead Time Preset**: The time `DT_PROTECT` is set in the algorithms for the UP, DOWN, and STOP states before the timer is started, if applicable. This ensures that the current time configuration always applies, even if the protection state is repeatedly entered.
- **Lockout for Simultaneous Requests**: If both inputs (`DI_UP` and `DI_DOWN`) are simultaneously requested, the timer is locked.If both conditions are TRUE, the function block remains in the protection state until the request is resolved.

State Overview

State	Description	DO_UP	DO_DOWN	Triggered Event Outputs
STOP	Waiting for a valid request; outputs inactive	FALSE	FALSE	–

UP	Up direction active; switching blocked without intermediate step	TRUE	FALSE	EO_UP, EO_DOWN
DOWN	Down direction active; switching blocked without intermediate step	FALSE	TRUE	EO_DOWN, EO_UP
PROTECT	Protection delay in progress; all outputs disabled	FALSE	FALSE	EO_UP, EO_DOWN, timeOut.START
EVAL	After the protection time has elapsed, the inputs are evaluated	–	–	– (state change)

Application Scenarios

- **Drive Control**: Prevents rapid changes of direction in motors that require a mechanical or electrical dead time (e.g., fans, conveyor belts, sliding gates).
- **Valve or Damper Control**: Ensures that a defined pause is observed before a change of direction (opening/closing) to prevent pressure surges or cavitation.
- **Safety-Oriented Interlock**: The function block can be used to debounce competing control signals and only accept the last pending command after a defined hold time.

Comparison with Similar Function Blocks

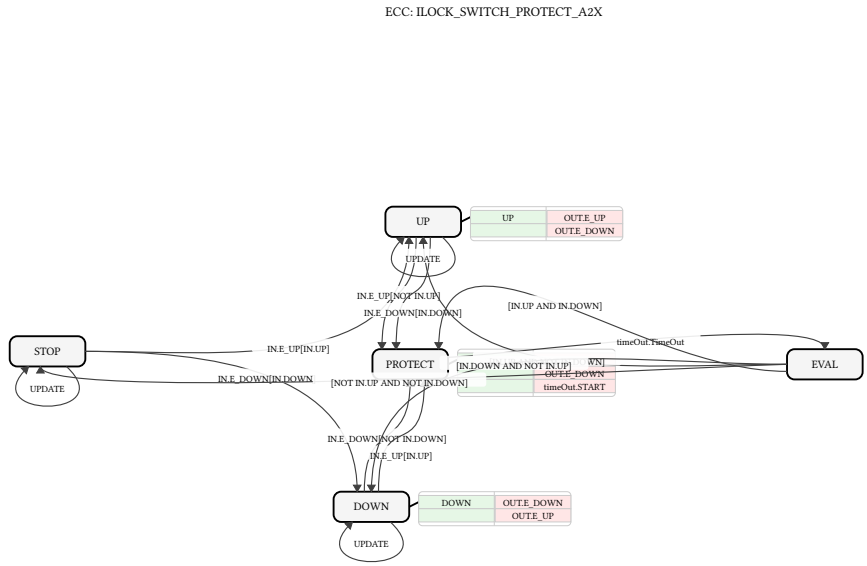
Function Block	Special Feature
SR Latch	Simple set-reset flip-flop with no delay
Interlock (simple)	Switches instantly, no dead time
ILOCK_SWITCH_PROTECT	Configurable protection delay, evaluation after the delay, handling of simultaneous requests

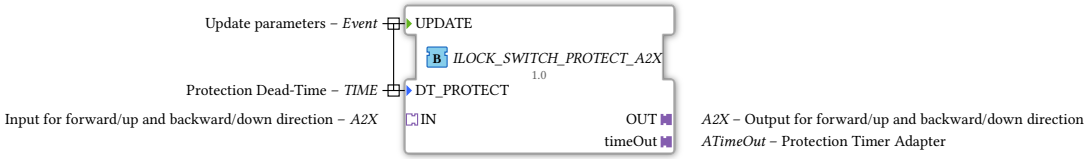
Compared to simple latches, **ILOCK_SWITCH_PROTECT** offers controlled switching, which is particularly advantageous with inductive loads or mechanically slow systems.

Conclusion

The **ILOCK_SWITCH_PROTECT** function block is a robust IEC 61499 component for applications requiring time-delayed, prioritized switching between two states. Its adjustable dead time, adapter-based timer implementation, and clear state logic make it particularly suitable for industrial control systems where reliability and protection mechanisms are paramount. The simultaneous activation of both event outputs in the action states should be considered during integration.

ILOCK_SWITCH_PROTECT_A2X





• – ○ • – ○ • – ○ •

Introduction

The **ILOCK_SWITCH_PROTECT_A2X** function block is an adapter-based implementation of a protective interlock switch. It takes forward/up and backward/down commands via a single `A2X` input adapter, prioritizes the most recent active command, and applies a configurable dead-time (`DT_PROTECT`) before actually switching the output direction. This prevents rapid or accidental direction changes that could damage mechanical systems.

The block is designed for applications where a controlled directional actuation (e.g., a motor, valve, or shutter) must be protected against abrupt reversals. It uses an internal state machine to manage the protective delay and outputs the resulting direction via an `A2X` output adapter.

Interface Structure

Event Inputs

Event	With	Comment
`UPDATE`	`DT_PROTECT`	Updates the protection dead-time parameter. The new value is taken from the associated data input.

Event Outputs

The block does not have direct event outputs. Instead, events are emitted through its output adapters:

- `OUT.E_UP` – signals the active upward/forward direction (via `OUT` adapter).
- `OUT.E_DOWN` – signals the active downward/backward direction (via `OUT` adapter).
- `timeOut.START` – starts the protective timer (via `timeOut` adapter).

Data Inputs

Name	Type	Initial Value	Comment
DT_PROTECT	TIME	T#50ms	Protection dead-time, i.e., the minimum delay between a command change and the actual output transition.

Data Outputs

The block does not have direct data outputs. Direction data is provided via the OUT adapter:

- OUT.UP – boolean indicating upward/forward direction is active.
- OUT.DOWN – boolean indicating downward/backward direction is active.

Adapters

Adapter	Type	Direction	Comment
IN	adapter::types::unidirectional::A2X	Socket (input)	Provides input commands: events E_UP/E_DOWN and data UP/DOWN.
OUT	adapter::types::unidirectional::A2X	Plug (output)	Outputs the resulting direction: events E_UP/E_DOWN and data UP/DOWN.
timeOut	iec61499::events::ATimeOut	Plug (output)	Timer adapter used for the protective dead-time. It receives a DT data value and a START event. The TimeOut event is used internally to trigger evaluation after the delay.

Functionality

The block operates as a directional interlock with a configurable protection delay. It continuously monitors the input adapter IN for two command types:

- UP (forward/upward)
- DOWN (backward/downward)

When a command is received and there is no current movement, the block sets the output direction immediately (after a short internal evaluation). However, if a direction change is requested while the opposite direction is already active, the block does **not** switch directly. Instead, it enters a PROTECT state, stops all output, and starts the protection timer with DT_PROTECT. Once the timer expires, the block evaluates the current input conditions and decides the next state:

- If only UP is active, it switches to the UP state.
- If only DOWN is active, it switches to the DOWN state.
- If neither is active, it returns to STOP.
- If both are active, it remains in PROTECT and restarts the timer (this ensures that the last active command eventually wins, but no direction is set while both are given).

The UPDATE event can be used anytime to change the dead-time value. The new value is applied immediately, even if a protection cycle is already running (the timer uses the updated DT_PROTECT value).

Technical Features

- **Adapter-based I/O**: Uses standard A2X adapters for both input and output, making it easy to interface with other blocks and reuse in different contexts.
- **Configurable dead-time**: The DT_PROTECT parameter (via UPDATE) allows the user to adjust the minimum delay before a direction change takes effect.
- **Priority handling**: The block resolves simultaneous UP and DOWN commands by staying in protective mode until only one direction remains active.
- **Transparent state machine**: The internal ECC clearly defines the states and transitions, ensuring deterministic behavior.
- **Timer integration**: Uses a standard ATimeOut adapter for timing, which simplifies integration with other timing resources.

State Overview

The internal ECC consists of five states:

State	Description
STOP	Idle state. No output direction is active. The block waits for an input command.
UP	Upward/forward direction is active. Outputs OUT.UP = TRUE and OUT.DOWN = FALSE. Emits OUT.E_UP.
DOWN	Downward/backward direction is active. Outputs OUT.UP = FALSE and OUT.DOWN = TRUE. Emits OUT.E_DOWN.
PROTECT	Protection delay active. All outputs are cleared (OUT.UP = FALSE, OUT.DOWN = FALSE). The timer is started with DT_PROTECT.
EVAL	Evaluation state after the delay. Determines the next direction based on current input values.

Transitions:

- From STOP to UP if IN.E_UP occurs and IN.UP is true.
- From STOP to DOWN if IN.E_DOWN occurs and IN.DOWN is true.
- From UP to PROTECT if IN.E_UP occurs with IN.UP false (i.e., release) or IN.E_DOWN occurs with IN.DOWN true (request opposite direction).
- From DOWN to PROTECT if IN.E_DOWN occurs with IN.DOWN false or IN.E_UP occurs with IN.UP true.
- From PROTECT to EVAL when the timer expires (timeOut.TimeOut).
- From EVAL to UP, DOWN, STOP, or PROTECT based on the combination of IN.UP and IN.DOWN.
- From any state back to itself on UPDATE (parameters are refreshed without changing the operating state).

Application Scenarios

The **ILOCK_SWITCH_PROTECT_A2X** block is suitable for any system that requires safe direction control with a mandatory dwell time between reversals. Typical examples include:

- **Electric motors** driving conveyor belts or winches where sudden reversal can cause mechanical stress.
- **Roller shutters, blinds, or gates** that must not change direction abruptly.
- **Valves or actuators** with inherent mechanical limits that require a short settling time.
- **Robotic axes** where a reversal must be delayed to avoid overcurrent or overshoot.
- **Pumps or fans** that should not be toggled rapidly.

The block is especially useful in safety-critical applications where a defined delay is mandated by regulations.

Comparison with Similar Blocks

Compared to a simple interlock switch (that switches direction immediately), this block adds a **protective dead-time**. This feature is essential when:

- The driven load has significant inertia and can be damaged by rapid direction changes.
- The control system must prevent simultaneous activation of both directions (e.g., to avoid short-circuiting a bridge).
- The application requires a minimum delay for settling or safety reasons.

The adapter version (A2X) offers a clean interface that separates control commands from the actual output, making it more modular than a version with direct I/O ports. It also simplifies integration with other adapter-based systems. The use of a standard timer adapter (ATimeOut) allows the dead-time to be synchronized with global timing resources.

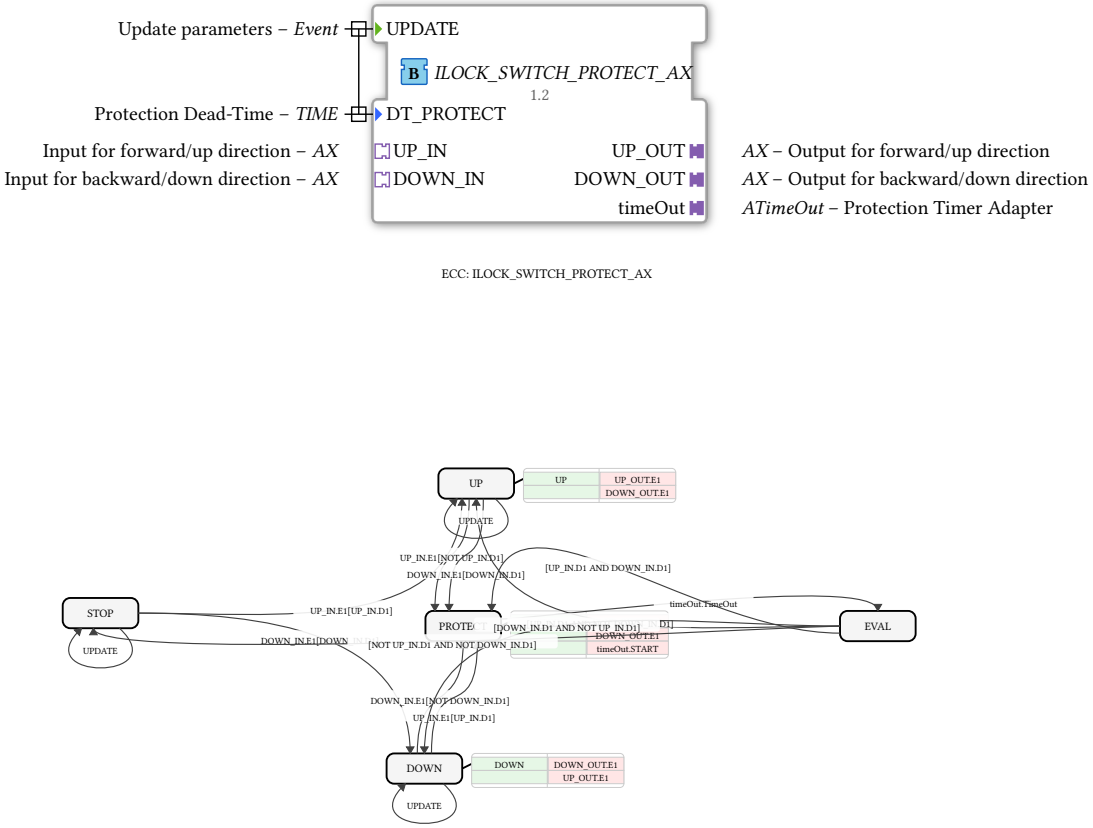
In contrast to blocks without a prioritization mechanism, this block handles the case where both UP and DOWN are given simultaneously by waiting in PROTECT until one command disappears. This ensures that the last active input eventually wins, preventing an undefined state.

Conclusion

The **ILOCK_SWITCH_PROTECT_A2X** function block provides a robust and configurable solution for interlocked direction control with a protective delay. Its adapter-based interface makes it easy to integrate into larger IEC 61499 applications, while the internal state machine guarantees deterministic behavior under all input conditions. The ability to dynamically update the dead-time via the UPDATE event adds

flexibility for changing operational requirements. This block is particularly suited for applications where mechanical or electrical protections demand a minimum time between direction reversals.

ILOCK_SWITCH_PROTECT_AX



• – ○ • – ○ • – ○ •

Introduction

The function block `ILOCK_SWITCH_PROTECT_AX` is used for interlocked switching between two input channels (UP/DOWN) with an adjustable protection dead time. It prioritizes the last active input and, through the dead time, prevents uncontrolled switching back and forth (bouncing) in the case of simultaneous or rapidly changing demands. The interface is provided via standardized unidirectional adapters (type `AX`) and a timer adapter.

Interface Structure

Event Inputs

Event	Description
`UPDATE`	Parameterization event; The current value of `DT_PROTECT` is assigned to the timer.

Event Outputs

No direct event outputs. Output is implicit via the adapters:

- `UP_OUT.E1` – Event output of the Up adapter
- `DOWN_OUT.E1` – Event output of the Down adapter
- `timeOut.TimeOut` – Event of the Timer adapter (internal use)

Data Inputs

Name	Type	Initial Value	Description
DT_PROTECT	TIME	T#50ms	Protection dead time that is activated after a state change.

Data Outputs

No direct data outputs. State output is provided via the adapters:

- UP_OUT.D1 – Boolean value for the up output (TRUE when active)
- DOWN_OUT.D1 – Boolean value for the down output (TRUE when active)

Adapters

Direction	Name	Type	Description
Socket	UP_IN	adapter::types::unidirectional::AX	Input for the "Up/Forward" request. Contains event E1 and Boolean data D1.
Socket	DOWN_IN	adapter::types::unidirectional::AX	Input for the "Down/Backward" request.
Plug	UP_OUT	adapter::types::unidirectional::AX	Output for the Up direction.
Plug	DOWN_OUT	adapter::types::unidirectional::AX	Output for the Down direction.
Plug	timeOut	iec61499::events::ATimeOut	Timer adapter for implementing dead time. Controls the event TimeOut after DT_PROTECT has elapsed.

Functionality

This module implements a **latched two-channel circuit with dead time**:

- The two inputs UP_IN and DOWN_IN can never be active simultaneously.
- As soon as an input becomes active (i.e., its event E1 occurs and the corresponding data D1 is TRUE), the module switches to the corresponding state (UP or DOWN) after a short dead time.
- If a direction change occurs, the **protection state PROTECT** is first entered, in which all outputs are set to FALSE and the timer is started with DT_PROTECT. After the dead time has elapsed, the system switches to state EVAL, which determines the final state based on the current input signals.
- The dead time prevents crosstalk or signal bounce, e.g., with mechanical switches or overlapping requirements.

Technical Features

- **Adapter-based interface**: The function block communicates exclusively via standardized adapters (AX for Boolean control, ATimeOut for timers). This allows for easy reuse in various container blocks.
- **Integrated Protection Logic**: In state PROTECT, all outputs are deactivated before a new direction is activated – this protects connected hardware (e.g., motors, valves) from short circuits or mutual blocking.
- **Parameter Update**: The event UPDATE allows dynamic adjustment of the dead time at runtime without having to reset the function block.
- **Dead Time on Every State Change**: Every transition (Stop→Up, Down→Up, etc.) always goes through the protection phase, regardless of the previous direction.

State Overview

State	Description
STOP	Idle state. No output active. Waiting for an input to be activated.

UP	Output UP_OUT.D1 = TRUE, DOWN_OUT.D1 = FALSE. Activates when UP_IN.D1 is present and DOWN_IN.D1 is inactive.
DOWN	Output DOWN_OUT.D1 = TRUE, UP_OUT.D1 = FALSE. Activates when DOWN_IN.D1 is present and UP_IN.D1 is inactive.
PROTECT	Protection phase. Both outputs are immediately set to FALSE, and the timer is started. After DT_PROTECT expires, the process switches to EVAL.
EVAL	Evaluation status. Based on the current input data, a decision is made as to which final state (STOP, UP, DOWN) or PROTECT (in case of simultaneous requests) will be applied.

The transitions defined in the ECC are:

- STOP ⟶ UP (at UP_IN.E1 and UP_IN.D1)
- STOP ⟶ DOWN (at DOWN_IN.E1 and DOWN_IN.D1)
- UP ⟶ PROTECT (at UP_IN.E1 and NOT UP_IN.D1 OR DOWN_IN.E1 and DOWN_IN.D1)
- DOWN ⟶ PROTECT (analogous)
- PROTECT ⟶ EVAL (after timeOut.TimeOut)
- EVAL ⟶ UP (at UP_IN.D1 AND NOT DOWN_IN.D1)
- EVAL ⟶ DOWN (at DOWN_IN.D1 AND NOT UP_IN.D1)
- EVAL ⟶ STOP (at NOT UP_IN.D1 AND NOT DOWN_IN.D1)
- EVAL ⟶ PROTECT (at UP_IN.D1 AND DOWN_IN.D1)
- In each state, an UPDATE event can re-evaluate the state and Update the dead time.

Application Scenarios

- **Interlocking of drives**: For example, in a lifting table or conveyor belt that may only be active in one direction at a time.
- **Short-circuit protection** in H-bridge circuits: Before the polarity of a motor is switched, both outputs are briefly deactivated.
- **Switching safety logic** in agricultural technology or automation, where mechanical limit switches or proximity sensors require time-based debouncing.
- **Interface between two control systems**: If both systems control a common axis, the function block prevents collisions by prioritizing the last active input.

Comparison with similar function blocks

- **Simple RS flip-flop**: Offers no dead time and switches instantly – can oscillate with overlapping signals.
- **SWITCH (without protection)**: Standard toggle switch also switches without delay and without debouncing.
- **SR Latch with Timer**: Similar function, but usually implemented as a composite function block. ILOCK_SWITCH_PROTECT_AX encapsulates all the logic, including the timer and evaluation, in a single Basic Function Block.
- **Interlock Function Blocks in IEC 61499**: Many libraries offer interlock function blocks, but often without an adapter interface or with fixed delays. This block is specifically designed for use with unidirectional AX adapters.

Conclusion

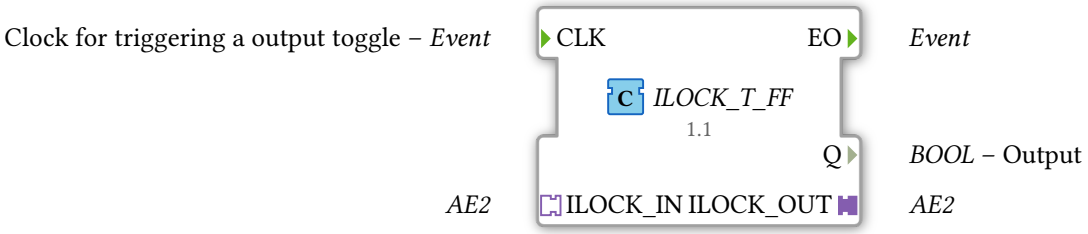
ILOCK_SWITCH_PROTECT_AX`This robust and flexible component for interlocked switching with adjustable protection dead time. The adapter-based interface enables seamless integration into modular 4diac architectures. The clear state machine ensures deterministic behavior and provides protection against faulty switching in time-critical applications. Ideal for applications in agricultural technology, automation, and drive technology where reliable direction changes are required.

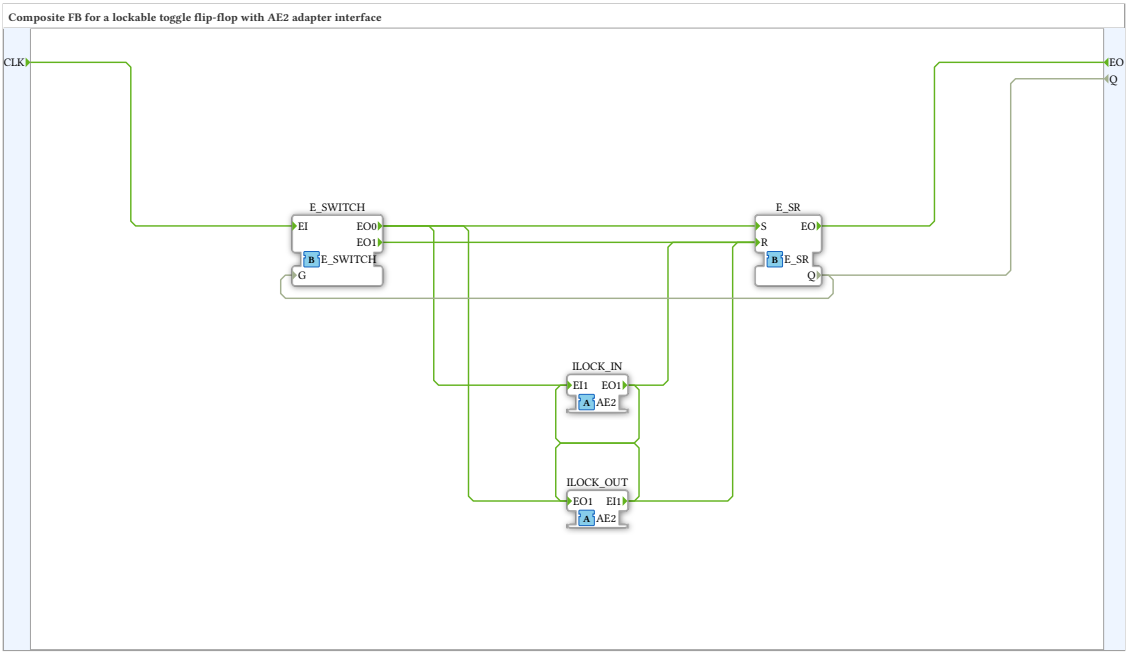
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ILOCK_T_FF





• – ○ • – ○ • – ○ •

Introduction

The **ILOCK_T_FF** is a composite function block (FB) that implements a lockable toggle flip-flop (T flip-flop). It extends a simple T flip-flop with a bidirectional interlock interface, allowing multiple blocks to be chained together in an interlock chain. This block is particularly suitable for safety-related control systems where states that have been switched must be blocked and passed on to subsequent elements.

Interface Structure

Event Inputs

Name	Type	Description
CLK	Event	Trigger signal to toggle the output.

Event Outputs

Name	Type	Description
EO	Event	Triggered as soon as the internal set/reset latch changes its state.

Data Inputs

No data inputs (control is achieved solely through events and the adapters).

Data Outputs

Name	Type	Description

Q	BOOL	Current state of the flip-flop (0 or 1).

Adapter

Name	Type	Direction	Description
ILOCK_IN	`adapter::types::bidirectional::AE2`	Socket	Receives and sends lock signals to and from upstream components in the chain.
ILOCK_OUT	`adapter::types::bidirectional::AE2`	Plug	Sends and receives lock signals to and from downstream components.

Functionality

Internally, the component consists of an **E_SWITCH** (event branch) and an **E_SR** (set-reset latch):

1. **Normal Operation (No Lock)**

- An event at the **CLK** is forwarded to the **E_SWITCH**.
- The output **Q** of the internal latch serves as the control signal **G** for the **E_SWITCH**:
- If **Q** = **FALSE**, the **E_SWITCH** switches the event to **E_SR.S** (Set) – **Q** becomes **TRUE**.
- If **Q** = **TRUE**, the **E_SWITCH** switches the event to **E_SR.R** (Reset) – **Q** becomes **FALSE**.
- This ensures a clean toggle (edge transition) with every **CLK**.

1. **Lock Mechanism (Interlock)**

- Lock events can be fed into the chain via the adapters **ILOCK_IN** and **ILOCK_OUT**.
- A lock event (e.g., from an upstream component via **ILOCK_IN.EO1** or from a downstream component via **ILOCK_OUT.EI1**) is applied to the **Reset** (**R**) of the latch.
- This immediately resets **Q** to **FALSE**. The toggle operation is interrupted because the **E_SWITCH** now remains in the **Q** = **FALSE** state and would attempt to set it on the next **CLK** event – however, the lock signal can take effect again.
- **Lock Propagation**: Each incoming lock event is simultaneously propagated to the other adapter output:
- **ILOCK_IN.EO1 → ILOCK_OUT.EO1** (to subsequent devices)
- **ILOCK_OUT.EI1 → ILOCK_IN.EI1** (to previous devices)
- This allows a lock to propagate in both directions (**Multi-Hop Chain Propagation**).

1. **Result**

- As long as no lock is active, the device operates like a normal T flip-flop.
- As soon as a lock becomes active, it forces **Q** = **FALSE** and blocks further toggle attempts until the lock is released.

Technical Features

- **Bidirectional Lock Propagation**: The device can both receive and propagate lock signals, enabling a daisy chain of multiple ILOCK_T_FF devices.
- **Feedback of its own state**: The output signal **Q** is used internally as a control signal for the **E_SWITCH** and determines the toggle direction.
- **No dedicated ECC**: All logic is implemented using the two included basic modules (E_SR, E_SWITCH).

State Overview

Since the module does not have its own state diagram, the states are derived from the internal **E_SR**:

State **Q**	Meaning in normal operation	Meaning when Lock is active
0	On the next CLK, it toggles to 1.	Lock has set Q to 0 and holds it there.
1	On the next CLK, it toggles to 0.	Lock immediately resets Q to 0.

An active lock is present as soon as one of the adapter events (ILOCK_IN.EO1 or ILOCK_OUT.EI1) is received. The lock is maintained until no further lock events are present (the exact resolution depends on the external signal control).

Application Scenarios

- **Safety interlocks in agricultural technology** (e.g., control of implements): Once activated, a lock state blocks all subsequent switching operations and is propagated to all other components in the chain.
- **Chained emergency stop systems**: Allows the propagation of a stop signal along multiple stations, while each station manages its own state.
- **Protocol adaptation for interlock signals**: The function block can be used as a universal link in an interlock chain where the output Q represents a switching operation.

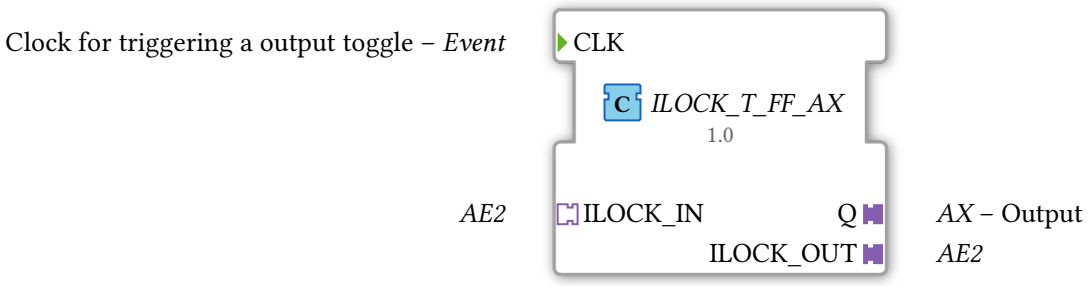
Comparison with similar function blocks

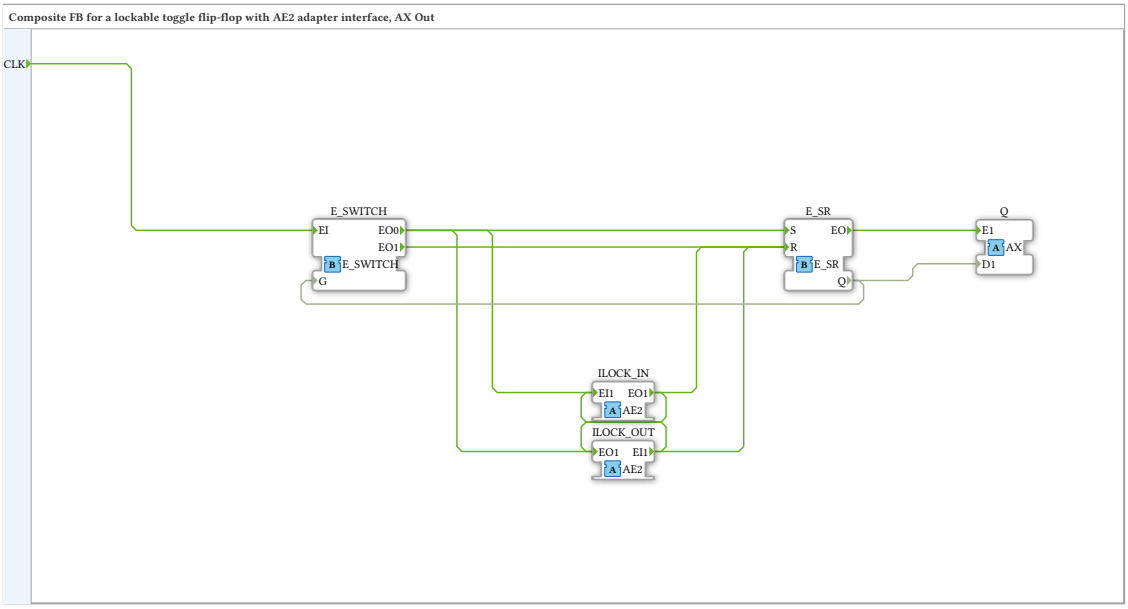
Function block	Properties	Difference from ILOCK_T_FF
T Flip-Flop (e.g., `E_TOGGLE`)	Pure toggle function without locking logic.	Lacks lock/interlock capability, no adapter interface.
RS Flip-Flop (e.g., `E_SR`)	Set/reset with two inputs, no toggle.	Can be extended with external logic to enable toggle functionality, but without integrated lock propagation.
Simple Interlock Block	Only lock propagation, no toggle.	The ILOCK_T_FF combines toggle and lock propagation in a single block.

Conclusion

The **ILOCK_T_FF** combines the functionality of a clocked T-flip-flop with efficient, bidirectional interlock propagation. Thanks to its integrated multi-hop propagation, it is ideally suited for networked safety circuits where a switching state must be blocked and simultaneously propagated to the entire circuit. Its compact design as a composite function block makes it easy to understand and reusable.

ILOCK_T_FF_AX





• – ○ • – ○ • – ○ •

Introduction

The function block `ILOCK_T_FF_AX` is a composite function block that implements a lockable toggle flip-flop. It features an AE2 adapter interface for connecting to a higher-level interlocking logic and outputs the current switching state via a unidirectional AX adapter. The function block is designed for use in safety-related or interlocked control applications where the switching of an output must be influenced by external conditions.

Interface Structure

Event Inputs

Event	Type	Description
`CLK`	Event	Clock signal that triggers a state change (toggle).

Event Outputs

This function block does not have its own event outputs. Events are passed through the adapter interfaces.

Data Inputs

No direct data inputs.

Data Outputs

No direct data outputs. The current state is provided via the AX adapter.

Adapters

Adapter	Type	Direction	Description
`Q`	`adapter::types::unidirectional::AX`	Plug/Output	Unidirectional output adapter that provides the current state (Boolean value) as data and outputs an event upon a state change.
`ILOCK_IN`	`adapter::types::bidirectional::AE2`	Socket	Bidirectional adapter (input) for receiving locking signals.

ILOCK_OUT	adapter::types::bidirectional::AE2	Plug	Bidirectional adapter (output) for forwarding locking events to the higher-level logic.

The AE2 adapters each have two event pairs (EI1/EO1, EI2/EO2) and two associated data ports. Only the first two ports (EI1/EO1) are used in this module.

Functionality

The internal structure consists of two IEC 61499 standard components: E_SR (set-reset flip-flop) and E_SWITCH (event switch). The toggle behavior is implemented as follows:

1. An incoming CLK event is fed to the event switch E_SWITCH.
2. The switch input G is connected to the output Q of the SR flip-flop.

If G = false is present, the event is forwarded to the output EO0 (set path).

- If G = true is present, the event is forwarded to output EO1 (reset path).

1. The set path (EO0) sets the SR flip-flop (E_SR.S) and simultaneously generates an event on ILOCK_OUT.EO1 and ILOCK_IN.EI1 to inform the adapters of the locking mechanism.
2. The reset path (EO1) resets the SR flip-flop (E_SR.R). This reset can also be triggered externally via ILOCK_IN.EO1 or ILOCK_OUT.EI1, enabling a **latched reset**.
3. The output of the SR flip-flop (E_SR.Q) is written to the AX adapter Q.D1, and the event Q.E1 is triggered with each state change.
4. Additionally, events between ILOCK_IN and ILOCK_OUT are passed through in both directions to allow communication with neighboring devices in the same latching array.

The device thus implements an edge-triggered toggle flip-flop that can be reset by external latching signals (via the AE2 adapters).

Technical Features

- **Composite Implementation:** The function block is built as a network of standard FBs (E_SR and E_SWITCH) and can therefore be easily adapted or integrated into other projects.
- **Bidirectional Interlocking Interface:** The AE2 adapters allow both the reception and transmission of interlocking events, enabling modular cascading of multiple function blocks.
- **Unidirectional AX Output:** The state is output as a clean Boolean signal with event provisioning; no additional data conversion is required on the receiver side.
- **Use of Only the First AE2 Ports:** The second ports of the AE2 adapters remain unused and can be added in a derived version if needed.

State Overview

The internal state of the flip-flop is binary:

State	Description
0 (false)	Output Q is inactive. The block will be set on the next CLK event.
1 (true)	Output Q is active. The block will be reset on the next CLK event.

State transitions occur exclusively on a CLK event (Toggle) or on a lock reset via ILOCK_IN.EO1 or ILOCK_OUT.EI1. Simultaneous setting and resetting are resolved by the logic of the SR flip-flop (reset takes priority if both events occur).

Application Scenarios

- **Interlocked Output Control:** In machine controls where an output may only be switched under specific safety conditions. The interlock signals (e.g., from emergency stop circuits or light curtains) are read via ILOCK_IN and suppress the toggle.
- **Cascaded Interlock Chains:** Multiple ILOCK_T_FF_AX blocks can be connected via the AE2 adapters to create a tiered interlock hierarchy.
- **Clock Synchronous Switching State Management:** This block is suitable for applications requiring clocked state changes, e.g., in step sequencers or sequential control systems.

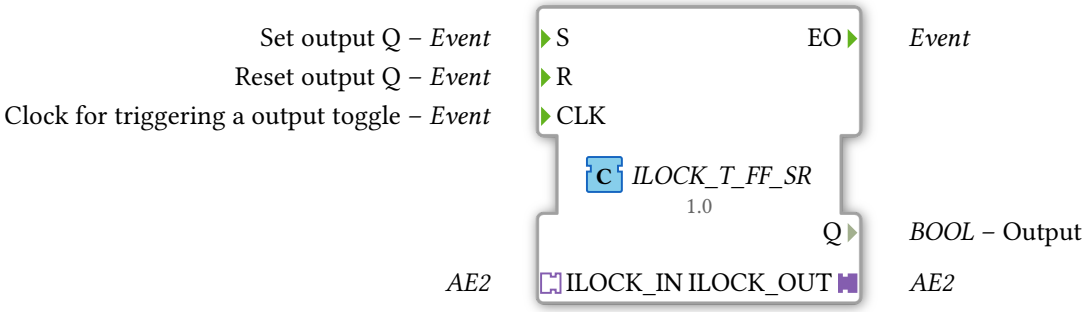
Comparison with Similar Components

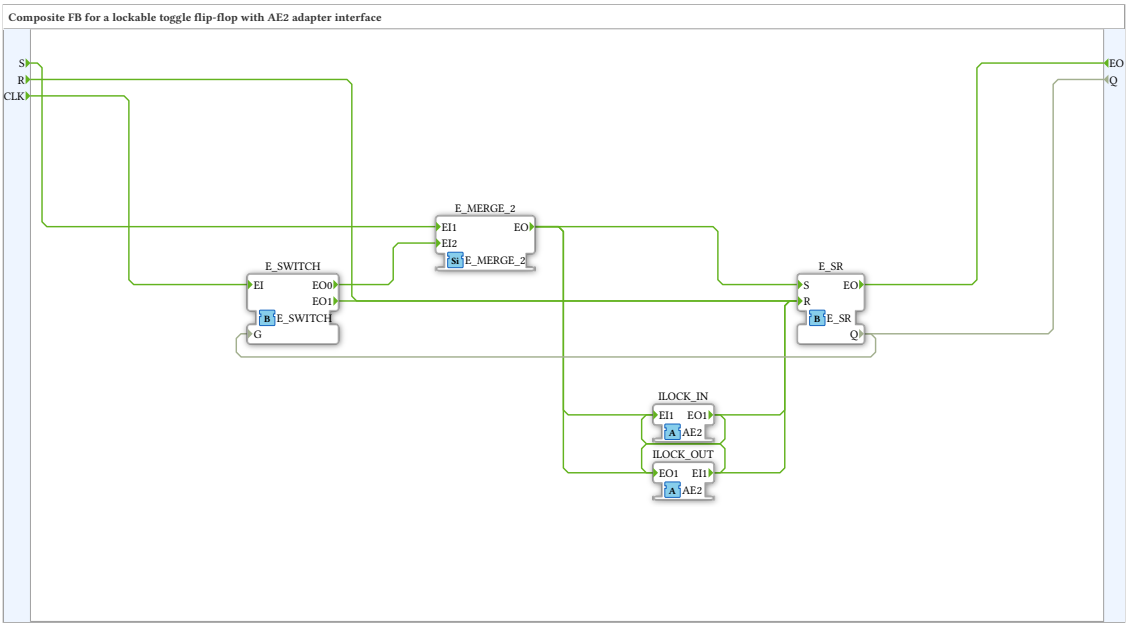
- **Standard Toggle Flip-Flop (e.g., our own IEC 61499 component `E_TOGGLE`):** A simple toggle flip-flop has no external latching interface and cannot be reset by external conditions. `ILOCK_T_FF_AX` extends this basic principle with bidirectional AE2 communication.
- **Set-Reset Flip-Flop (e.g., `E_SR`)):** An SR flip-flop has separate set and reset inputs, but no toggle mechanism. This component combines toggle and SR behavior with interlock logic.
- **Components with AX output:** The AX adapter is a common standard for unidirectional Boolean outputs. Other components often use separate data ports, whereas this one uses an encapsulated adapter interface.

Conclusion

ILOCK_T_FF_AX` offers a compact, standards-compliant solution for a interlockable toggle flip-flop in the IEC 61499 environment. Its use of AE2 adapter interfaces allows for seamless integration into modern, modular automation architectures. The composite design facilitates maintenance and customization, while the clear separation of event and data flows simplifies troubleshooting. The component is specifically optimized for safety-related or interlock-dependent switching tasks and provides a solid basis for the development of more complex control logics.

ILOCK_T_FF_SR





• – ○ • – ○ • – ○ •

Introduction

The function block `ILOCK_T_FF_SR` is a composite function block (FB) that implements a latchable toggle flip-flop with set and reset functionality. It has an adapter interface (AE2) that enables bidirectional communication with other instances and serves for interlock chaining. The flip-flop toggles its output `Q`

on every rising edge of the clock input CLK, provided no reset or set event is present. Latching is achieved via the adapters, which can be set by external function blocks.

Interface Structure

Event Inputs

Name	Type	Description
S	Event	Sets output Q to TRUE
R	Event	Sets output Q to FALSE
CLK	Event	Clock event – triggers a toggle on output Q

Event Outputs

Name	Type	Description
EO	Event	Triggered after every change of the internal state (Set, Reset, or Toggle)

Data Inputs

None.

Data Outputs

Name	Type	Description
Q	BOOL	Current state of the flip-flop (TRUE = set, FALSE = reset)

Adapter

Name	Type	Direction	Description
ILOCK_IN	AE2 (bidirectional)	Socket (input)	Allows the reception of latching signals from upstream devices
ILOCK_OUT	AE2 (bidirectional)	Plug (output)	Sends interlock signals to downstream components

The adapters ILOCK_IN and ILOCK_OUT are of type AE2 (general, bidirectional adapter). They are used to cascade multiple instances of the component, allowing a set state in one component to reset other components (interlock).

Functionality

The component is based on an internal network consisting of three primitive functional blocks: a E_SR (set/reset flip-flop), a E_SWITCH (event-driven toggle switch), and a E_MERGE_2 (event merger).

- **Set (S):** An event at S is routed via E_MERGE_2 directly to the Set input of E_SR. The output Q is set to TRUE. Simultaneously, the events are passed on to the connected function blocks via the adapters.
- **Reset (R):** An event at R goes directly to the Reset input of E_SR. The output Q is set to FALSE. Reset signals received from the adapters (via ILOCK_IN.EO1 and ILOCK_OUT.EI1) also trigger a reset.
- **Toggle (CLK):** An event at CLK is routed to E_SWITCH. Its control input, G, is connected to the current output, Q.
- If Q = FALSE is active, the clock event is forwarded via the output EO0 to E_MERGE_2.EI2 and from there to the set input of E_SR. Thus, Q is set to TRUE.
- If Q = TRUE is set to TRUE, the clock event is routed directly to the reset input of E_SR via the output EO1. Q is set to FALSE.

This results in the toggle behavior: Each clock pulse reverses the current state of Q.

- **Interlock:** The adapters ILOCK_IN and ILOCK_OUT are interconnected such that an incoming event from ILOCK_IN (e.g., from a previous block) triggers the reset input and is simultaneously passed on to the next block via ILOCK_OUT. This allows for the creation of a chain of flip-flops, where only one may be active at any given time.

After each state change, the event output EO is triggered.

Technical Features

- **Composite FB:** The block is implemented as a network of IEC 61499 elementary modules. This ensures transparent and traceable behavior.
- **Bidirectional Adapters:** Using the adapter type AE2 allows for flexible coupling as both a socket (input) and a plug (output).
- **Conflict Handling:** If a set and a reset event occur simultaneously (e.g., S and R), the reset event takes precedence because the reset input of E_SR has multiple sources and is treated with priority. The exact priority is determined by the E_SR implementation (default: reset takes precedence over set).
- **No Data Inputs:** The function block operates purely event-driven; its state is only changed by events.

No Data Inputs:

State Overview

The internal flip-flop E_SR has two states:

- **State FALSE** (Q = FALSE): The output is reset. A clock event causes it to be set.
- **State TRUE** (Q = TRUE): The output is set. A clock event causes it to be reset.

A set event (S) always results in the state TRUE, regardless of the current state.
A reset event (R) always results in the state FALSE, regardless of the current state.

Current State	Event	New State	Output Actions
FALSE	S	TRUE	EO is triggered
FALSE	R	FALSE	EO is triggered
FALSE	CLK	TRUE	EO is triggered
TRUE	S	TRUE	EO is triggered
TRUE	R	FALSE	EO is triggered
TRUE	CLK	FALSE	EO is triggered

The adapter events act like an external reset and set the state to FALSE.

Application Scenarios

- **Interlock Control in Safety Circuits:** Several machines or drives should interlock with each other. Only one may be active at any given time. By chaining multiple ILOCK_T_FF_SR devices, it can be ensured that only one flip-flop is set at any given time.
- **State Machines with Changing Modes:** This component is suitable as a toggle flip-flop that can be overridden by external set/reset signals, e.g., for manual intervention.
- **Cascaded Circuits with Feedback:** The adapter interface allows for easy integration into existing AE2-based communication structures.

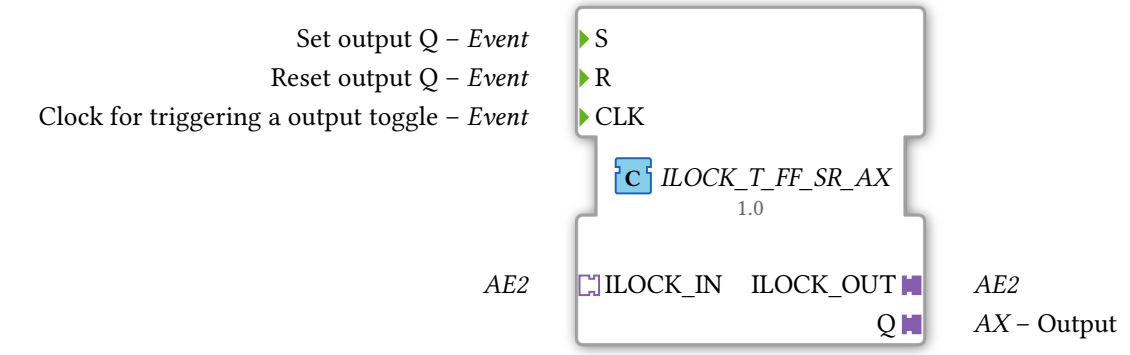
Comparison with Similar Components

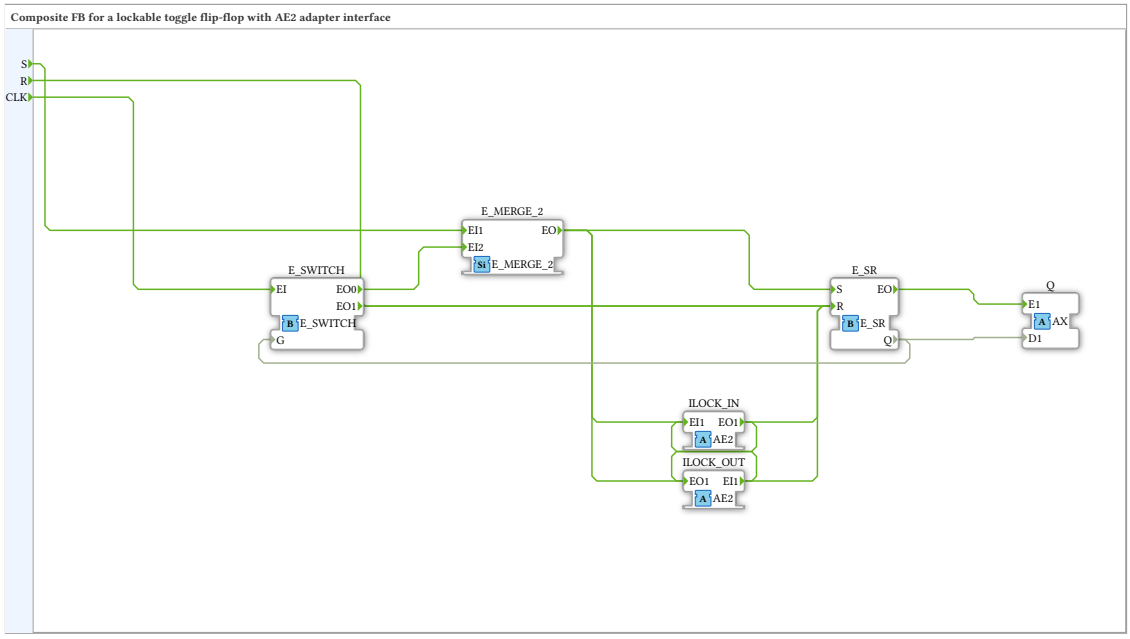
- **Standard T-FF (IEC 61499)E_T_FF):** Offers only toggle functionality without separate set/reset inputs. The ILOCK_T_FF_SR extends this to include set and reset control.
- **Standard SR flip-flop (e.g., E_SR):** Has only set and reset, no toggle. This component combines toggle with SR.
- **Interlock component:** Many manufacturer-specific blocks implement interlock via buses; this component uses standardized adapters (AE2) and is therefore more portable.

Conclusion

The ILOCK_T_FF_SR is a versatile function block that combines a toggle flip-flop with set/reset and an interlock interface. The clear separation between logic and adapter connections allows for the construction of safe, cascadable circuits. By using IEC 61499 standard components, the behavior remains traceable and expandable.

ILOCK_T_FF_SR_AX





• – ○ • – ○ • – ○ •

Introduction

The function block **ILOCK_T_FF_SR_AX** is a composite function block (FB) for a latching toggle flip-flop with set/reset functionality and an AE2 adapter interface. It enables the targeted setting, resetting, and clocking of an output signal, taking into account latching states that are read and output via bidirectional adapters.

Interface Structure

Event Inputs

Name	Type	Comment
S	Event	Sets output Q (if not latched)
R	Event	Resets output Q
CLK	Event	Clock input for toggling the output

Event Outputs

This function block does not have explicit event outputs. Output is provided via adapters.

Data Inputs

There are no direct data inputs. The interlock information is exchanged via the adapters `ILOCK_IN` and `ILOCK_OUT`.

Data Outputs

Name	Type	Comment
Q	Adapter AX (unidirectional)	Output signal of the internal SR flip-flop

Adapters

Direction	Name	Type	Comment
Socket	ILOCK_IN	AE2 (bidirectional)	Receives lock signals from external components
Plug	ILOCK_OUT	AE2 (bidirectional)	Sends lock signals to external components
Plug	Q	AX (unidirectional)	Outputs the current state of the flip-flop

Functionality

This component implements an SR flip-flop with a toggle function and a lock mechanism.

- **Set (S)**: An event at `S` causes the internal SR flip-flop `E_SR` to be set via a merge (E_MERGE_2). Output Q is set to `true` unless a locking reset occurs at `ILOCK_IN` or `ILOCK_OUT`.
- **Reset (R)**: An event at `R` sets output Q to `false`. Additionally, both input adapter `ILOCK_IN` and output adapter `ILOCK_OUT` can also trigger a reset – this serves as an interlock.
- **Toggle (CLK)**: The current state of Q is evaluated on each clock event at `CLK`. A switch (E_SWITCH) is used to:
- Generate a set pulse when Q = `false` (Q ⟶ `true`).
- Generate a reset pulse when Q = `true` (Q ⟶ `false`).

Thus, the output toggles with every CLK event, unless a latch prevents this.

The latch is implemented using the bidirectional AE2 protocol: The signals from `ILOCK_IN` and `ILOCK_OUT` are passed through each other and can both reset the internal flip-flop and be exchanged with each other. This allows for a secure lock (e.g., mutual exclusion).

Technical Features

- **Composite Architecture**: The function block consists of the embedded functions `E_SR`, `E_SWITCH`, and `E_MERGE_2`, which are logically linked.
- **Bidirectional Interlocking**: The adapters `ILOCK_IN` and `ILOCK_OUT` are of type AE2, which enables bidirectional exchange of events and data. This allows for interlocking with other function blocks.
- **Toggle Only on State Change**: The toggle mechanism evaluates the output Q, so a toggle only occurs if the flip-flop has not already been set by S or R.
- **No Direct Data Inputs**: Control is achieved exclusively via events and adapters, which facilitates integration into event-driven systems.
-

State Overview

The internal state of the flip-flop can take the values `false` (0) or `true` (1). The possible transitions are:

Current	Event	New State	Conditions
0	S	1	No locking via ILOCK
0	R	0	–
0	CLK	1	No locking
0	ILOCK_IN/ILOCK_OUT	0	–
1	S	1	–
1	R	0	No locking (R takes precedence)

1	CLK	0	No Interlock
1	ILOCK_IN/ILOCK_OUT	0	–

Application Scenarios

- **Machine Control**: Switching an output (e.g., valve or motor) at a specific time, where a higher-level safety controller can lock the output in case of danger.
- **Mutual Interlock**: Multiple `ILOCK_T_FF_SR_AX` function blocks can be interconnected via bidirectional adapters so that only one of its outputs is active at any given time (e.g., in turnout or drive control).
- **Clock-Controlled Signal Generation**: Generating a periodic square wave signal by applying a clock signal to `CLK`, where setting and resetting can be controlled externally.

Comparison with Similar Function Blocks

- **Simple SR Flip-Flop**: Has no toggle function and no interlock adapters.
- **TcFF (Toggle Flip-Flop)**: Allows toggling only on a clock event, but has no separate set/reset inputs and no interlock interface.
- **Standard Interlock Devices**: Often limited to binary signals, while `ILOCK_T_FF_SR_AX` combines toggle, SR, and bidirectional interlock functionality in a single device.

Conclusion

The `ILOCK_T_FF_SR_AX` function block offers a flexible and safe way to control an output that can be switched on a clock signal and influenced via separate set/reset inputs as well as an expandable interlock interface. The combination of SR flip-flop and toggle logic makes it particularly suitable for safety-critical applications in automation technology.

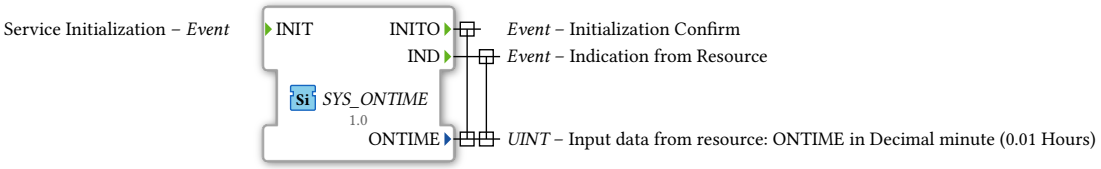
measurement

SYS_ONTIME

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SYS_ONTIME` is a Service Interface Function Block (SIFB). Its main task is to read and provide the uptime (ontime) of the resource in which it is executed. It serves as a standardized interface between the application logic and the system-level services of the runtime environment.



Interface Structure

Event Inputs

- **`INIT`**: This event initializes the function block. It must be triggered before the block can begin its service.

Event Outputs

- **`INITO`**: This event confirms the successful initialization of the function block. It is triggered along with the first valid `ONTIME` value.
- **`IND`**: This event signals that a new, updated `ONTIME` value has been provided by the system resource manager.

Data Inputs

- This function block has no data inputs.

Data Outputs

- **`ONTIME` (UINT)**: This output provides the cumulative uptime of the resource. The unit is “decimal minutes,” which is equal to 0.01 hours (e.g., a value of 150 equals 1.5 hours).

Adapters

- This function block does not use adapters.

Functionality

The `SYS_ONTIME` block acts as a client for an internal system service. After receiving the `INIT` event, it registers with the resource manager to gain access to the uptime data. Once initialization is complete, the `INITO` event is triggered along with the current uptime value.

The block then continuously monitors the system resource. As soon as the uptime changes (typically due to a system tick or similar mechanism), the resource manager triggers an indication. The `SYS_ONTIME` block receives this, updates its internal `ONTIME` value, and forwards the `IND` event, along with the new value, to the connected application logic.

The block then continuously monitors the system resource. As soon as the uptime changes (typically due to a system tick or similar mechanism), the resource manager triggers an indication.

Technical Features

- **Service Interface FB**: As a SIFB, its implementation is specific to the respective 4diac runtime (FORTE) and the underlying hardware/platform. The exact method of time measurement (e.g., based on processor startup time or the runtime environment) is runtime-defined.
- **Unit**: The operating time is provided in the unusual but practical unit for control tasks: the decimal minute (1/100 of an hour). This facilitates calculations in the decimal system, e.g., for maintenance intervals or energy consumption estimates.
- **Read-Only Access**: The block provides read-only access to an internal system value. Resetting the operating time is not possible via this interface.

Status Overview

1. **Inactive**: The block has been created but not yet initialized. No communication with the system resource is taking place.
2. **Initialized**: After receiving `INIT`, the block is registered with the resource manager and awaits data. The state changes immediately after successful initialization, followed by the output of `INITO`.
3. **Ready**: The block has received the first `ONTIME` value and is ready for operation. It now awaits indication events (`IND`) from the system to output updated values.

Application Scenarios

- **Predictive Maintenance**: Monitoring the operating time of machines or plant components to trigger maintenance intervals based on actual usage.
- **Energy Management**: Recording the operating hours of consumers (e.g., motors, heaters) to calculate energy consumption or for load profiles.
- **Production Data Acquisition (PDA)**: Logging of production or machine runtimes for Overall Equipment Effectiveness (OEE) analysis.
- **Lifetime Monitoring**: Checking whether critical components have reached or exceeded their specified lifespan (in operating hours).

⚖️ Comparison with similar modules

- **`E_CYCLE` / `E_DELAY`**: These modules themselves generate periodic events or delays. `SYS_ONTIME`, on the other hand, *reads* a passively existing, cumulative time value from the system.
- **`E_SR` (Set-Reset) with counter**: A manually configured counter for power-on processes could provide similar data, but would be implemented application-side and not persistent system-wide. `SYS_ONTIME` provides a reliable, system-maintained value that is usually persistent even after a restart.
- Other SYS_ *blocks** (such as `SYS_TIME`): While `SYS_TIME` provides the current time and date, `SYS_ONTIME` specifically provides the pure uptime of a resource.

🛠️ Related Exercises

- Exercise_140

Conclusion

The `SYS_ONTIME` function block is an essential tool for all applications that rely on recording and analyzing uptime. As a standardized service interface, it abstracts away the platform-specific implementation and

provides application logic with simple, reliable access to this important system metric. Its use is particularly recommended in the areas of maintenance, energy efficiency, and operational data acquisition.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.10.3.10 stations

Computers

Introduction

The global constants `Computers` provide a collection of predefined network addresses (IP addresses and ports) as strings (`WSTRING`). They serve to centrally manage connection information for various computers or stations within a logiBUS project. Using these constants improves the maintainability and readability of the code, as changes to network addresses only need to be made in one central location.

Interface Structure

Global constants are not function blocks in the traditional sense; therefore, no event or data interfaces exist. Instead, constant declarations are provided that are available project-wide.



Data Inputs

Not applicable. Constants are fixed values and have no inputs.

Data Outputs

Not applicable. Constants are directly incorporated into expressions or assignments.

Adapters

Not applicable.

Functionality

The constants are defined when the project is compiled. Each constant is assigned a fixed, unchanging value of type `WSTRING` (Wide String). Once the program is running, these constant names (`C_LO`, `C_ET`, etc.) can be used instead of the hard-coded strings in any part of the program that imports or recognizes these constants.

Technical Specifications

- **Type:** All constants are of type `WSTRING`, which supports Unicode strings.
- **Scope:** As global constants, they are available within the declared package (`logiBUS::stations`).
- **Initialization:** The values are assigned directly upon declaration and are read-only at runtime.
- **Usage:** They are typically intended for configuring communication modules (e.g., `PUBLISH`, `SUBSCRIBE`, `CLIENT`, `SERVER`) where the address of the communication partner is required.

State Overview

Not applicable. Constants have no state.

Application Scenarios

1. **Simplified Address Configuration:** In a distributed control system with multiple controllers (e.g., main controller, floor controller), the IP addresses of the individual devices can be stored centrally here. 2. **Portability between environments:** By simply swapping the values in this constant file, a program can be quickly adapted between different network environments (e.g., development local host, test network, production network).
2. **Typo prevention:** Using the constant name `C_ET` is less error-prone than repeatedly typing the string `"192.168.5.1:61499"`.

⚖️ Comparison with similar components

- **`VAR_GLOBAL` (Global variables):** Unlike global variables, constants (`CONSTANT`) cannot be changed after initialization. They therefore offer greater program security.
- **`VAR_CONFIG` (Configuration variables):** Configuration variables are also constants, but they can be configured per resource. The global constants defined here have a fixed, project-wide value.
- **Direct Value Specification:** While it's possible to directly use string literals in the code, this leads to poorer maintainability and a higher susceptibility to errors compared to using central constants.

Conclusion

The global constants `Computers` are a simple yet extremely useful tool for structuring and maintaining network connection data in 4diac projects. They promote clean programming practices, reduce redundancy, and make the code more adaptable to changes in the network topology. Their use is particularly recommended in larger or distributed automation systems.

–

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.10.3.11 storage

Const

storageStatusMessages

• – ◦ • – ◦ • – ◦ •

Introduction

`storageStatusMessages` is a **GlobalConstants** element from the `logiBUS::storage::const` package. It is not an executable function block, adapter, or subapplication. Instead, it provides a centralized collection of `STRING` constants used by storage-related function blocks to report status and error information.

The constant set is designed for storage modules that work with initialization (INI) and non-volatile storage (NVS) systems. It contains human-readable messages for successful operations, missing or invalid data, initialization state, and common NVS error cases.

Interface Structure

Because `storageStatusMessages` is a GlobalConstants definition, it does not provide an IEC 61499 event or data interface. It has no inputs, no outputs, and no adapters. The declared constants are globally visible to other elements in the same library or project.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

None.

Adapters

None.

Functionality

The main purpose of `storageStatusMessages` is to define a single source of truth for storage status strings. Instead of using hard-coded literals in multiple function blocks, storage-related FBs can reference these global constants. This reduces typing errors, simplifies translations or adjustments, and makes the behavior of different storage blocks more consistent.

The following table lists all constants exposed by this element.

Constant Name	Type	Value
OK	STRING	OK
NO_CHANGE	STRING	Value was not stored. Old is new
INITIALISED	STRING	initialized
NOT_INITIALISED	STRING	Not initialized
ERR	STRING	ERROR
ERR_NVS_NOT_FOUND	STRING	ESP_ERR_NVS_NOT_FOUND
ERR_NVS_DEFAULT_SET	STRING	Default Value was used
ERR_SECTION_EMPTY	STRING	SECTION empty
ERR_KEY_EMPTY	STRING	KEY empty
ERR_SECTION_WHITESPACE	STRING	SECTION contains whitespace
ERR_KEY_WHITESPACE	STRING	KEY contains whitespace
ERR_READ_ONLY	STRING	Key is read-only

The constants cover the following categories:

- General success and error states: OK, ERR.
- Initialization states: INITIALISED, NOT_INITIALISED.
- No-operation result: NO_CHANGE.
- NVS-related errors: ERR_NVS_NOT_FOUND, ERR_NVS_DEFAULT_SET.
- Validation errors for storage sections and keys: empty or whitespace.
- Access restriction error: ERR_READ_ONLY.

Technical Features

- Defined as GLOBALCONSTANTS in the package logiBUS::storage::const.
- Contains 12 STRING constants.
- All constants are read-only and cannot be modified at runtime.
- The values are globally accessible, avoiding local duplication of status strings.
- The names follow a consistent prefix style: ERR_ for error constants.
- The message set is versioned, allowing controlled evolution of the status vocabulary.
- Supports both human-readable logging and comparison in application logic.

State Overview

storageStatusMessages is not an FB and therefore has no state machine. It does not process events, maintain runtime variables, or transition between states. The element has exactly one static condition: all declared constants are defined and available as soon as the containing library is loaded.

Application Scenarios

This GlobalConstants element is useful wherever storage services need to report their result in a standardized way.

Typical scenarios include:

- Function blocks that access NVS storage and need to return ERR_NVS_NOT_FOUND when a stored key does not exist.
- Initialization routines that distinguish between INITIALISED and NOT_INITIALISED.
- Write operations that detect that the new value is identical to the old one and return NO_CHANGE.
- Validation logic that checks section and key names for empty strings or whitespace.
- Error handling for read-only keys using ERR_READ_ONLY.
- Logging or HMI displays that show status messages from storage operations.

Comparison with Similar Blocks

Since storageStatusMessages is a passive constant container rather than a function block, it cannot be compared directly to executable FBs. However, it can be compared with common alternatives:

- Compared to hard-coded literals, it provides a central, reusable message catalog and avoids inconsistent wording.
- Compared to local constants inside individual FBs, it makes the same messages available across an entire library or project.
- Compared to integer error codes, it delivers readable text directly, but it does not provide numeric encoding for binary protocols.
- Compared to a function block that generates status strings, it requires no runtime processing and has no execution overhead.

Conclusion

`storageStatusMessages` is a compact global constant set that standardizes status and error reporting for storage-related function blocks. It is especially suited for INI and NVS handling on platforms such as ESP32, where storage errors need to be communicated consistently. By using these global constants, applications become more readable, maintainable, and robust against inconsistent status strings.

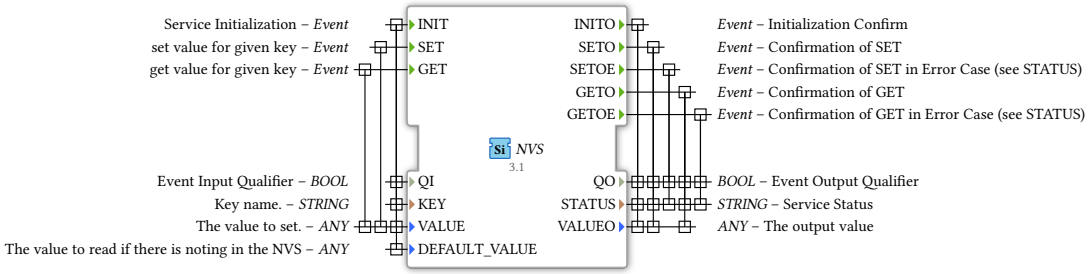
esp32_nvs

NVS

• – ◦ • – ◦ • – ◦ •

Introduction

The NVS (Non-Volatile Storage) function block enables persistent storage and loading of data on an ESP32 microcontroller. It utilizes the ESP32's non-volatile storage (NVS) to store values under a defined key and retrieve them as needed. This block is particularly suitable for applications where configuration data, calibration values, or status information must be retained across device restarts.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block. Triggers the read operation of the value specified by `KEY`. If no value is stored, `DEFAULT_VALUE` is returned.
- **SET**: Saves the value passed to `VALUE` under the previously configured key (`KEY`) in the NVS.
- **GET**: Reads the value stored under the configured key (`KEY`) in the NVS.

Event Outputs

- **INITO**: Confirms the execution of the initialization (INIT) and returns the result.
- **SETO**: Confirms the execution of the write operation (SET) and returns the status.
- **SETOE**: Confirmation of SET in the error case (see `STATUS`); carries `QO` and `STATUS`.
- **GETO**: Confirms the execution of the read operation (GET) and returns the read value.
- **GETOE**: Confirmation of GET in the error case (see `STATUS`); carries `QO` and `STATUS`.

Data Inputs

- **QI** (BOOL): Qualifies the INIT event input. The service is activated by `TRUE` and deactivated by `FALSE`.
- **KEY** (STRING): The unique key name under which the value is stored in the NVS or from which it is read. This value must be set during the INIT phase.
- **VALUE** (ANY): The data record to be stored. It can be any elementary data type (e.g., BOOL, INT, REAL, STRING).
- **DEFAULT_VALUE** (ANY): The default value returned during a read operation (INIT or GET) if no value is yet stored in the NVS under the specified key.
•

Data Outputs

- **QO** (BOOL): Reflects the current operating state of the function block (`TRUE` = ready/successful, `FALSE` = not ready/error).
- **STATUS** (STRING): Contains a status message describing the success or type of error of the last operation (e.g., "OK", "Key not found", "Storage full").
- **VALUEO** (ANY): The value actually read from the NVS or, if no value was present, the value from `DEFAULT_VALUE`.

Adapters

This function block does not use adapters.

Operation

The NVS block operates based on its state. During initialization (INIT with `QI=TRUE`), the specified `KEY` is set. The system then automatically attempts to read the corresponding value from the NVS (Non-Volatile Storage). The result (stored value or `DEFAULT_VALUE`) is output along with the status via `INITO`.

Afterward, the block can perform two main operations:

1. **Write (SET)**: The value assigned to `VALUE` is stored in the NVS under the specified key (`KEY`). Confirmation is provided via `SETO`.
2. **Read (GET)**: The value currently stored under the key is read from the NVS and made available via `VALUEO` at output `GETO`.

The data types of the inputs and outputs `VALUE`, `DEFAULT_VALUE`, and `VALUEO` must be consistent. The block is designed for repeated access to the same key.

Technical Features

- **ESP32-Specific**: This block directly uses the NVS API of the ESP32-IDF and is therefore only compatible with ESP32-based target platforms.
- **Type Flexibility**: The use of the generic data type `ANY` allows for the storage of various elementary data types. The specific type is set during initial use (INIT) and must be retained for all subsequent operations.

Persistence: Data in the NVS survives a microcontroller restart and is also retained after a power failure (depending on the NVS memory configuration).

State Overview

1. **Not Initialized**: The block is inactive after startup. The key (`KEY`) is not set.
2. **Initialized/Ready**: After a successful `INIT` event, the block is ready for operation. The key is configured, and an initial value (from NVS or default) has been loaded. The block is waiting for `SET` or `GET` events.
3. **Write Operation**: During the processing of a `SET` event.
4. **Read Operation**: During the processing of a `GET` event or the implicit read operation during `INIT`.

Application Scenarios

- **Device Configuration**: Saving network credentials (SSID, password), IP addresses, or other parameters to be retained after a reset.
- **Operational Data Acquisition**: Persistently counting operating hours, cycle counts, or units produced.
- **Calibration Data**: Storing sensor- or actuator-specific calibration values.
- **Last Known State**: Saving the system state before an unexpected shutdown to restore it upon the next startup.

⚖️ Comparison with similar building blocks

- **E_R_TRIG / F_TRIG**: These are purely event triggers and do not offer persistent data storage.
- **BLINK / DELAY**: Time-based blocks without memory functionality.
- **E_EEPROM**: A generic EEPROM block. The NVS block is specific to the ESP32 platform and utilizes its optimized, wear-leveling file system (NVS), while a generic E_EEPROM block would be hardware-dependent and often operate on simpler memory.

🛠️ Related Exercises

- Exercise_012
- Exercise_012a_sub
- Exercise_020c2_sub

Conclusion

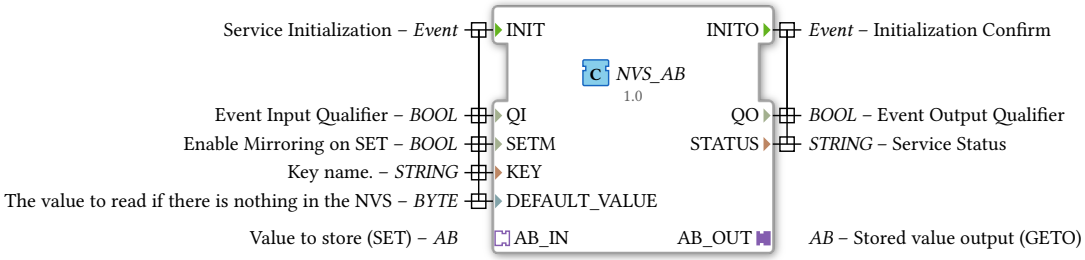
The NVS function block provides a simple and robust interface to the ESP32's non-volatile memory. Its clear separation of read and write operations, along with its type-flexible data processing, makes it ideal for a wide range of applications requiring persistent data on embedded ESP32 systems. Its integration into the 4diac IDE enables platform-specific functionality within the portable IEC 61499 model.

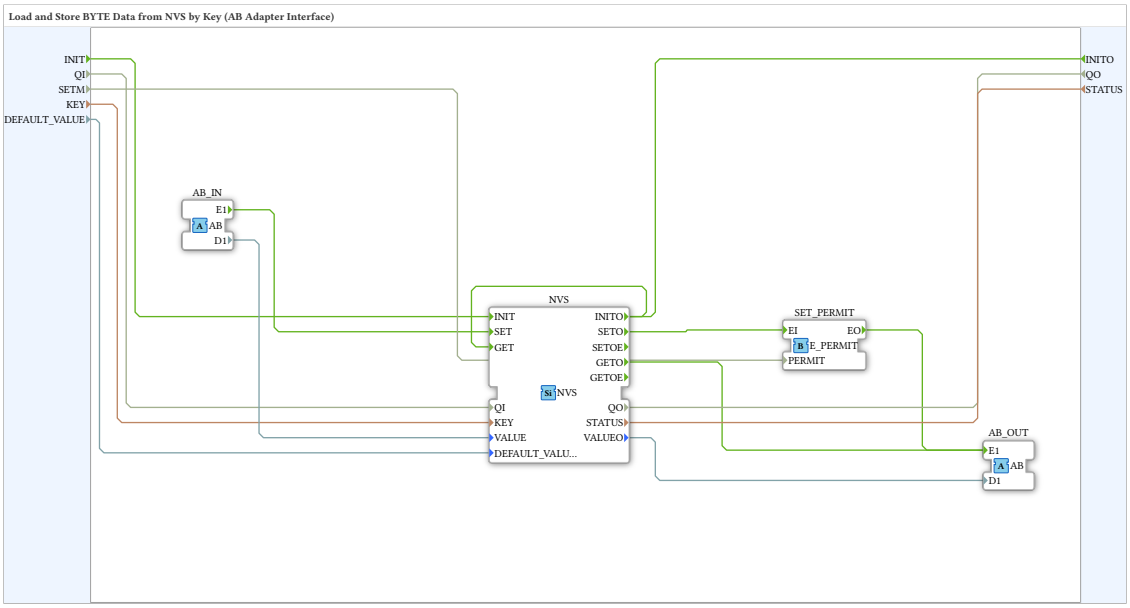
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de
- 🌐 MCU vs. MPU Comparison Guide on ms-muc-docs.de

]

NVS_AB





• – ○ • – ○ • – ○ •

Introduction

`NVS_AB` loads and stores a `BYTE` value under a key (`KEY`) in the ESP32's non-volatile storage (NVS). Unlike the bidirectional NVS_AB2, it uses two separate unidirectional `AB` adapters: a socket for writing (`AB_IN`) and a plug for reading/confirming (`AB_OUT`).

Interface Structure

Event Inputs

- **INIT**: initializes the block and reads the value stored under `KEY` (or `DEFAULT_VALUE` if none exists yet). Carries `QI`, `SETM`, `KEY`, `DEFAULT_VALUE`.

Event Outputs

- **INITO**: confirms initialization and the initial read, carries `QO` and `STATUS`.

Data Inputs

- **QI** (BOOL): qualifier for the `INIT` event.
- **SETM** (BOOL): enables mirroring (confirmation) on write (`SET`).
- **KEY** (STRING): name of the key in the NVS.
- **DEFAULT_VALUE** (BYTE): value returned when nothing is stored under `KEY` yet.

Data Outputs

- **QO** (BOOL): success qualifier, carried with `INITO` and the internal adapter confirmations.
- **STATUS** (STRING): status message of the NVS access.

Adapters

- **AB_IN** (Socket, type `AB`): new value to store – an adapter event on `AB_IN.E1` triggers a `SET` on the internal NVS block, `AB_IN.D1` carries the value to write.
- **AB_OUT** (Plug, type `AB`): outputs the most recently read value – both after the automatic initial read on `INIT` and after each confirmed write (`AB_OUT.D1`/`AB_OUT.E1`).

Functionality

`NVS_AB` wraps an internal instance of `logiBUS::storage::esp32_nvs::NVS` plus an `E_PERMIT` gate (`SET_PERMIT`). On `INIT`, the internal NVS instance is initialized (`NVS.INIT`) and a `GET` is triggered automatically, returning the value stored under `KEY` (or `DEFAULT_VALUE`); the result is output via `AB_OUT`, and `INITO` confirms completion.

To write, the block connected to `AB_IN` fires an adapter event (`AB_IN.E1`) carrying the new value (`AB_IN.D1`), which triggers `NVS.SET`. Whether the successful write is additionally reported back via `AB_OUT` is controlled by `SETM`: if `SETM` is set, `SET_PERMIT` releases the `AB_OUT.E1` event after a successful `SET` ("mirroring").

Technical Details

- Pure composite FB with no ECC of its own – the actual flash logic lives in the internal `NVS` instance; `NVS_AB` only wires events and data between it and the two `AB` adapters.
- Two separate unidirectional adapters (`AB_IN`/`AB_OUT`) instead of a single bidirectional one – this lets the read and write paths connect to different peers, which NVS_AB2 (a single shared `AB2` socket) cannot do.
- `SETM` only controls whether a write is additionally mirrored back via an `AB_OUT` adapter event – the value is stored either way.

State Overview

The block is stateless in terms of its own ECC: on `INIT` it runs through initialization plus an initial read once, then reacts to every `AB_IN` adapter event with an NVS write and, optionally, a confirmation via `AB_OUT`.

Application Scenarios

- Persisting individual `BYTE` configuration values (e.g. operating mode, channel selection) on the ESP32
- Building a read/write pair where the read and write sides need to connect to different blocks
- Restoring a last-stored `BYTE` value after a restart via `AB_OUT`

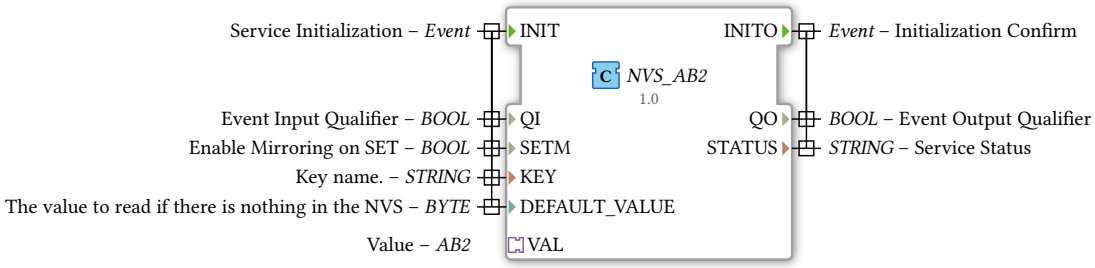
⚖️ Comparison with Similar Blocks

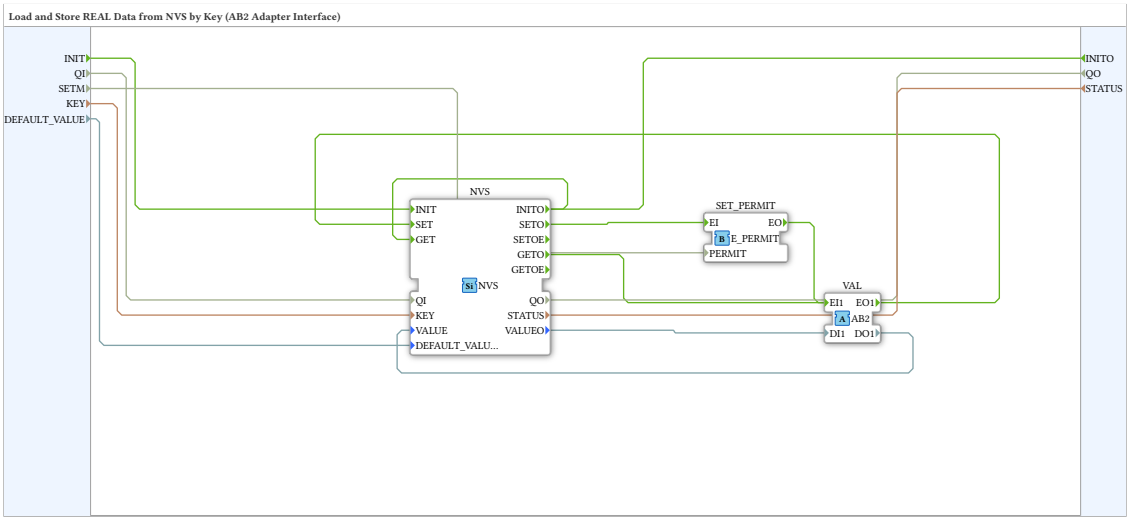
Compare with NVS_AB2, which serves the same purpose through a single bidirectional `AB2` adapter instead of two unidirectional `AB` adapters, and with INI_AB, which provides the same interface for the text-based `settings.ini` instead of the NVS flash storage.

Conclusion

NVS_AB provides simple, unidirectionally wired access to a persistent BYTE value in the ESP32's NVS, and fits anywhere the read and write paths need to be kept separate.

NVS_AB2





• – ○ • – ○ • – ○ •

Introduction

The function block **NVS_AB2** enables the loading and saving of BYTE data in non-volatile storage (NVS) using a string key. It encapsulates NVS access in a convenient, asynchronous interface and communicates with the environment via a bidirectional **AB2** adapter. This makes the block particularly suitable for scenarios where configuration values or status data need to be permanently stored on an ESP32.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initializes the function block. The input data (QI, SETM, KEY, DEFAULT_VALUE) are transferred when this event occurs.

Event Outputs

Event	Type	Description
INITO	EInit	Confirmation of successful initialization and the first read operation.

Data Inputs

Name	Type	Description
QI	BOOL	Qualifier for the initialization event (e.g., enabling processing).

SETM	BOOL	Enable mirroring on SET.
KEY	STRING	Name of the key under which the value is stored in the NVS.
DEFAULT_VALUE	BYTE	Value returned if no value exists in the NVS for the specified key.

Data Outputs

Name	Type	Description
QO`	BOOL	Qualifier for the output event – signals success (TRUE) or failure (FALSE).
STATUS	STRING	Detailed status message (e.g., error text or confirmation).

Adapter

Name	Type (AB2 adapter)	Description
VAL	adapter::types::bidirectional::AB2	Bidirectional interface for the data value. Reading and writing are performed via the adapter channels (events: EI1, EO1; data: DI1, DO1).

Functionality

Upon arrival of the **INIT** event, the internal NVS module is initialized. Immediately after successful initialization, a **read operation** (GET) is automatically initiated, which reads the value corresponding to the key KEY from the NVS. If no value is found, DEFAULT_VALUE is used. The read value is output via the adapter (VAL.DI1), and the event INITO signals completion.

After initialization, the function block can process asynchronous read and write requests via the adapter:

- **Read**: An event received at the adapter port VAL.EO1 triggers another read operation (GET) in the NVS. The read value is again made available via VAL.DI1, and the acknowledgment (VAL.EI1) is sent.
- **Write**: The value to be saved is made available via the adapter on VAL.DO1. An event at VAL.EO1 triggers the **write operation** (SET) in the NVS. Acknowledgement is sent via VAL.EI1 after successful saving.

The outputs QO and STATUS are set to the current status of the NVS module upon each of the output events (INITO, as well as the internal adapter acknowledgments).

Technical Features

- **NVS Access on the ESP32**: The module utilizes the function block defined at logiBUS::storage::esp32_nvs::NVS, which encapsulates the ESP32's flash memory management.
- **Bidirectional Adapter Interface**: The AB2 adapter allows the function block to process both read and write requests from the environment without requiring separate trigger events on the function block itself.
- **Automatic First Read Access**: After initialization, the stored value is read immediately, so that the current data value is available via the adapter after INITO.
- **Error Handling**: The status (STATUS) and the qualifier (QO) indicate the success or failure of each NVS operation.

State Overview

The function block does not have explicit state machines, as it uses the behavior of the internal NVS function block as a composite function block. The following phases can be distinguished:

1. **Initialization** – triggered by INIT
2. **Ready** – after successful initialization, the module waits for adapter events
3. **Read/Write Operation** – brief access to the NVS during the processing of an adapter event

Application Scenarios

- **Configuration storage** for device parameters (e.g., setpoints, calibration data) on ESP32-based controllers.
- **Persistent state maintenance**, e.g., last operating mode or counter readings across a restart.

- **Data exchange via AB2 protocol** with other modules that support the AB2 adapter interface – easy integration into existing automation solutions.

Comparison with Similar Function Blocks

Compared to simpler read/write function blocks (e.g., direct NVS FBs without an adapter), **NVS_AB2** offers:

- **Asynchronous, event-driven communication** via the AB2 adapter – decouples data access from the main control flow.
- **Automatic initialization and first read** reduces programming effort.
- **Uniform error signaling** via `QO` and `STATUS`.

In contrast, function blocks without an adapter require separate event inputs for reading and writing and do not offer a standardized bidirectional interface.

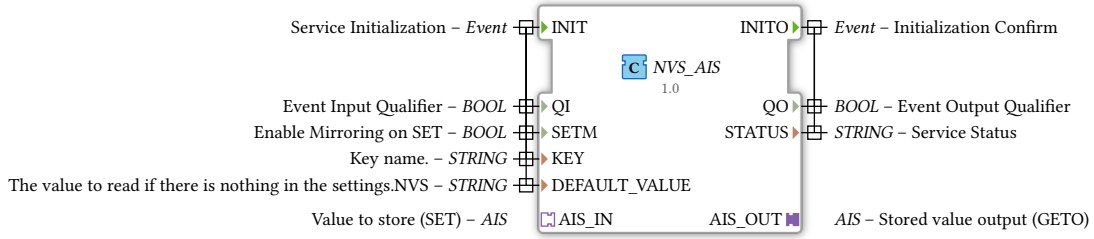
Conclusion

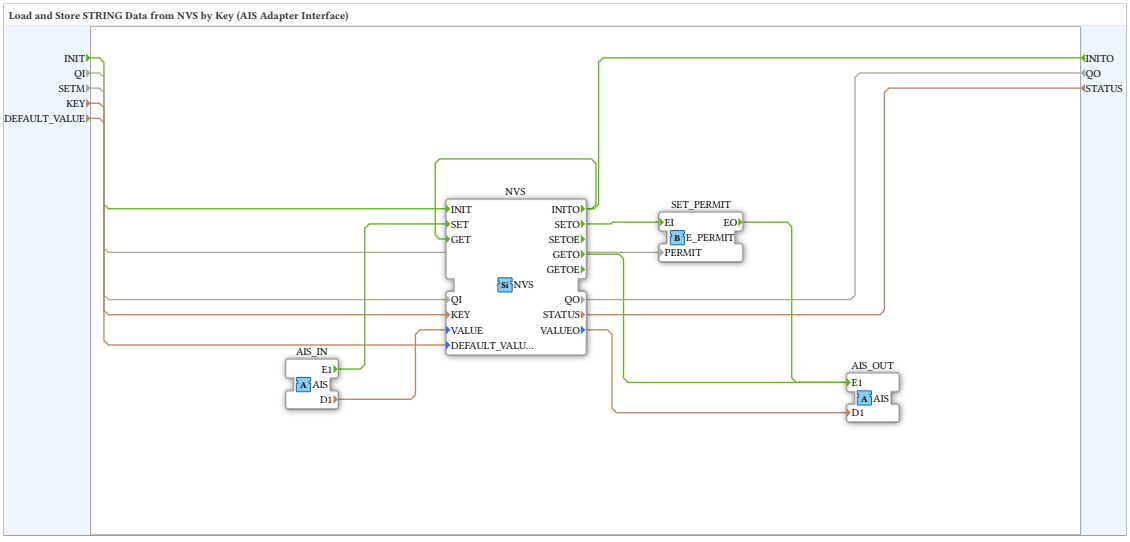
NVS_AB2 is a practical function block for reliable, asynchronous access to the non-volatile memory of an ESP32. By encapsulating the NVS logic in a composite module and providing an AB2 adapter interface, it can be flexibly integrated into automation projects requiring persistent data storage. The combination of automatic initialization, read/write support, and clear status feedback makes it an efficient solution for embedded controllers.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

NVS_AIS





• – ○ • – ○ • – ○ •

Introduction

The function block `NVS_AIS` is used to store and load strings (STRING) in the non-volatile storage (NVS) of an ESP32 microcontroller. The data is addressed via a key (KEY) and exchanged via AIS (Acyclic

Information Service) adapters. The block encapsulates the NVS initialization as well as the basic read and write operations and provides a standardized interface for integration into industrial control applications.

Interface Structure

Event Inputs

Event	Type	Description	Data Carried
INIT	EInit	**Service Initialization** – Starts the function block and communication with the NVS.	QI, SETM, KEY, DEFAULT_VALUE

Event Outputs

Event	Type	Description	Carrying Data
INITO	EInit	**Initialization Confirm** – Confirms successful initialization or signals an error.	QO, STATUS

Data Inputs

Name	Data Type	Description
QI	BOOL	**Event Input Qualifier** – Controls the behavior during the INIT phase (e.g., activation).
SETM	BOOL	Enable Mirroring on SET: if TRUE, the confirmation of a SET (SETO) is additionally forwarded to the output adapter (mirrored)
KEY	STRING	**Key name** – The key under which the value is stored/retrieved in the NVS.
DEFAULT_VALUE	STRING	**Default value** – Value that is read if no entry exists in the NVS for the specified KEY.

Data Outputs

Name	Data type	Description
QO	BOOL	**Event Output Qualifier** – Indicates the success of the last operation.
STATUS	STRING	**Service Status** – Feedback on the NVS driver status (e.g., error messages).

Adapters

Adapter	Type	Description
AIS_IN	`adapter::types::unidirectional::AIS` (Socket)	**Value to store (SET)** – Receives a STRING value to be stored via the AIS protocol.
AIS_OUT	`adapter::types::unidirectional::AIS` (Plug)	**Stored value output (GETO)** – Sends the read STRING value from the NVS via the AIS protocol.

Functionality

1. **Initialization**

An event at the `INIT` input starts the internal logic. The function block calls the embedded `NVS` function block, which prepares the non-volatile memory. The supplied data (`QI`, `SETM`, `KEY`, `DEFAULT_VALUE`) are forwarded to the internal function block.

1. **Read Operation**

After a successful INIT phase (event `INITO` of the internal NVS), the `GET` event input of the internal NVS is automatically activated. The read value (or the `DEFAULT_VALUE` entry if no entry exists) is output via the `AIS_OUT` adapter as the `E1` event and `D1` data.

1. **Write Operation**

A new value can be received via the `AIS_IN` adapter. The associated `SET` event (E1) is routed to the internal NVS module, which stores the received value (D1) under the previously set `KEY`. The acknowledgment (`SETO`) is then sent via the `AIS_OUT` adapter.

1. **Feedback**

After both reading and writing, the module outputs the status (`QO`, `STATUS`) as well as the `INITO` event. The status can be used for error diagnosis.

Technical Features

- **NVS for ESP32** – The module is specifically designed for the ESP32's non-volatile memory and uses the corresponding driver API.
- **Adapter Interface** – Communication with the outside world is exclusively via AIS adapters (unidirectional). This decouples the module from specific bus or application protocols.
- **Automatic Read Operation After INIT** – After initialization, a read operation is started immediately, so the stored value is instantly available at the output.
- **Unified Status** – All relevant status information is reported back via the outputs `QO` and `STATUS`.

State Overview

The function block `NVS_AIS` itself does not have an explicit state machine. Its behavior is entirely controlled by the embedded function block `NVS`, which cycles through the typical states of an NVS driver (e.g., initialization, ready, error). The events `INIT` and `INITO` represent the start-stop cycle. The adapter events `E1` initiate the read/write requests.

Application Scenarios

- **Configuration Memory** – Persistent storage of device settings (e.g., network parameters, operating modes) as strings.
- **Data Exchange with Other Devices** – Using the AIS adapters for easy integration into an IEC 61499 application without having to program direct bus or memory accesses.
- **Recovery After Restart** – The combination of INIT, automatic reading, and default values ensures that a defined initialized state is reached.

Comparison with Similar Devices

- **`NVS` (direct)**: The direct NVS device offers more detailed control options (dedicated events for GET/SET) but requires manual connection of the interfaces. `NVS_AIS` simplifies the application by embedding the read/write logic and using standardized adapters.
- **Additional memory modules (e.g., for SD cards)**: These usually operate with different protocols (SPI, I²C) and offer different memory schemes. `NVS_AIS` is optimized for the specific characteristics of the ESP32-NVS (key-value pairs, low latency).

Conclusion

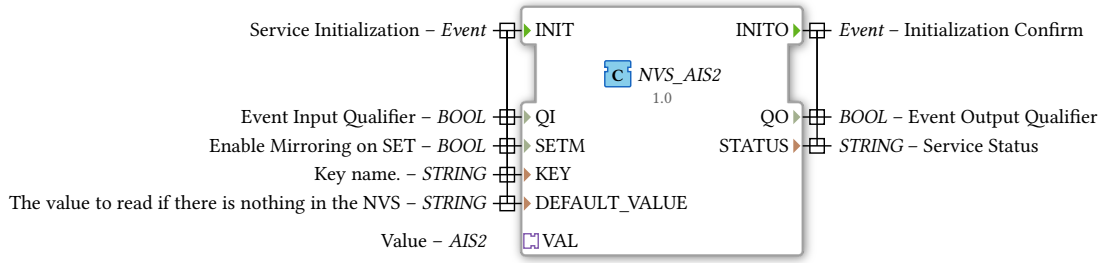
NVS_AIS`is a practical function block that encapsulates the saving and loading of strings in the ESP32's non-volatile memory and provides it via a clean adapter interface. With its automated initialization and integrated read and write operations, it is particularly well-suited for applications requiring simple, reliable persistence of configuration data. The adapter interface enables loose coupling with other components and increases reusability across various IEC 61499 projects.

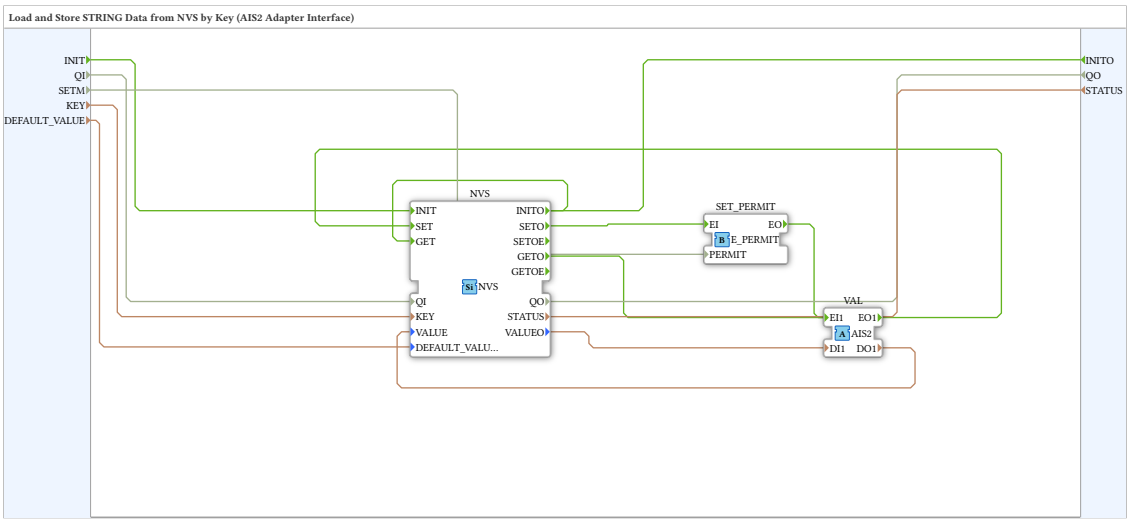
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de
- 🌐 MCU vs. MPU comparison guide on ms-muc-docs.de

]

NVS_AIS2





• – ○ • – ○ • – ○ •

Introduction

The **NVS_AIS2** function block is used to load and store STRING data in non-volatile storage (NVS) of an ESP32 via an adapter. It encapsulates initialization and access to a single NVS entry identified by a KEY.

Interface Structure

Event Inputs

Event	Type	Comment	Associated Variables
INIT	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Event Outputs

Event	Type	Comment	Associated Variables
INITO	EInit	Initialization Confirm	QO, STATUS

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier
SETM	BOOL	Enable mirroring on SET
KEY	STRING	Key name for NVS entry
DEFAULT_VALUE	STRING	Default value if key does not exist in NVS (Default: ’’)

Data Outputs

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

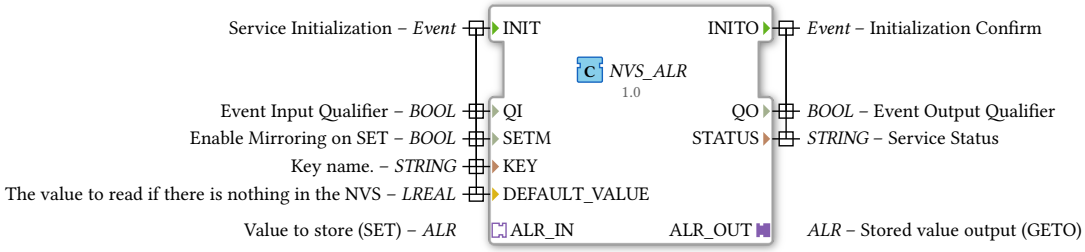
Adapters

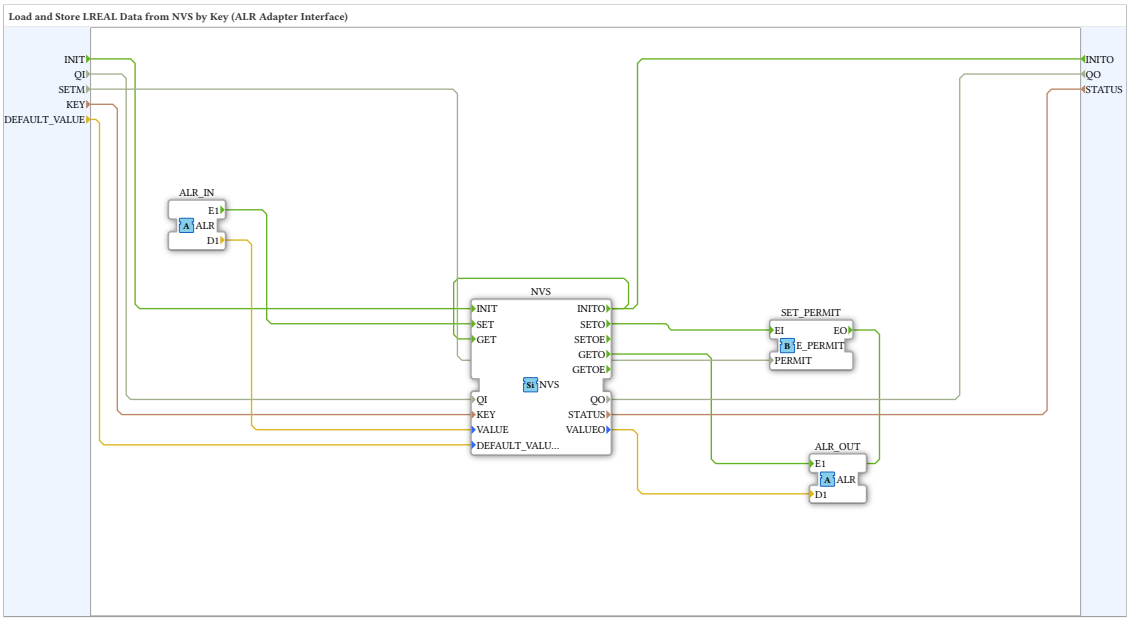
Adapter	Type	Comment
VAL	adapter::types::bidirectional::AIS2 (Socket)	Value (STRING)

Functionality

1. **Initialization**: An INIT event initializes the internal NVS block.
2. **Read value**: The read value is output via VAL.DI1.
3. **Store value**: An external block sends E01 via VAL to write a new value (D01) to NVS.

NVS_ALR





• – ○ • – ○ • – ○ •

Introduction

The function block **NVS_ALR** is used to load and store values of type LREAL in non-volatile storage (NVS) using a key. It is specifically designed for connection via the unidirectional **ALR adapter**. The block combines an initialization phase, in which the memory contents are read, with a subsequent read and write operation via the adapter.

Interface Structure

Event Inputs

Event	Description	Associated Variables
INIT	Initializes the function block and performs an initial read operation from the NVS.	QI, SETM, KEY, DEFAULT_VALUE

Event Outputs

Event	Description	Included Variables
INITO	Confirmation of initialization and feedback of the result.	QO, STATUS

Data Inputs

Variable	Type	Description
QI	BOOL	Control for initialization (TRUE = active).
SETM	BOOL	Enable mirroring on SET.
KEY	STRING	Key name for NVS access.
DEFAULT_VALUE	LREAL	Value to be read if no entry exists in the NVS under the specified key.

Data Outputs

Variable	Type	Description
QO	BOOL	Initialization confirmation (TRUE = successful).
STATUS	STRING	Status message (e.g., error message for failed access).

Adapters

Adapter	Direction	Type	Description
ALR_IN	Socket	`adapter::types::unidirectional::ALR`	Receives write commands (stores value).
ALR_OUT	Plug	`adapter::types::unidirectional::ALR`	Sends read values after a read operation.

Functionality

1. **Initialization**

After an event at the **INIT** input, the internal function block `NVS` is called with the specified key (`KEY`), the default value (`DEFAULT_VALUE`), and the activation (`QI`). The initialization process opens the NVS area and immediately reads the existing value. This value is output via the **ALR_OUT** adapter as an event with the corresponding data value (`D1`). Simultaneously, the **INITO** event output is triggered with the acknowledgments (`QO`, `STATUS`).

1. **Write via ALR_IN**

An event at the **ALR_IN.E1** input (connected to the socket) triggers a **SET** operation. The supplied data value (`ALR_IN.D1`) is written to the NVS under the previously stored key. After successful writing, the event is passed to **ALR_OUT.E1**, and the written value is output there.

1. **Read via ALR_IN?**

In this network, a **GET** command is automatically triggered after initialization. An explicit read call via the adapter is not included in the design; the function block always outputs the current value via **ALR_OUT** after each NVS operation (INIT, SET).

Technical Features

- **Adapter Interface**

Communication with the outside world occurs exclusively via the unidirectional ALR adapter. This makes the module particularly suitable for modular architectures where data flows are handled via standardized interfaces.

- **Internal NVS Module**

The FB `NVS` (from the library `logiBUS::storage::esp32_nvs`) is used in the network. This encapsulates the actual read and write logic on the ESP32-NVS.

- **Automatic Read Operation**

After initialization, a GET request is executed immediately, so the user receives the current value (default value or stored value) right away.

- **Error Handling**

The outputs `QO` and `STATUS` allow for easy verification of initialization success. Errors are reported as a string.

State Overview

The function block does not cycle through explicitly modeled states, but is event-driven. The following behavior occurs:

- **Idle State** – No INIT is performed. No actions are executed.
- **Initializing** – INIT is received; NVS operations are started; upon completion, INITO is triggered and a GET request is initiated.
- **Ready** – After successful initialization, the function block waits for events via **ALR_IN.E1** (write) or internal GET requests (only triggered automatically initially).
- **Write** – At ALR_IN.E1, the value is saved and the result is sent to ALR_OUT.

Application Scenarios

- **ESP32 Configuration Parameters**

Storage of LREAL values such as calibration factors, thresholds, or PID parameters that should be retained after a restart. The ALR adapter enables easy integration with other function blocks that use the same adapter type.

- **Encapsulated Data Access**

If a function block needs to read and write LREAL values from the NVS, but directly using the NVS function block seems too complex, NVS_ALR offers an abstracted interface.

- **Modular Automation**

Use in Industry 4.0 applications where devices communicate via standardized adapter services (e.g., via an ALR configuration layer).

Comparison with Similar Function Blocks

- **NVS (Direct)**

The `NVS` function block offers more flexibility (e.g., random access, different data types), but requires more complex wiring and explicit GET/SET calls. NVS_ALR reduces the complexity to an adapter interface.

- **BOOL_NVS, INT_NVS (Hypothetical)**

Analogous function blocks for other data types. NVS_ALR is specifically designed for LREAL and utilizes the ALR adapter, which is typically intended for LREAL data.

- **Retentive Variables**

In some systems, retentive variables can be used directly in the function block model. However, NVS_ALR is explicitly designed for use with an external flash memory (NVS) and is therefore more portable.

Conclusion

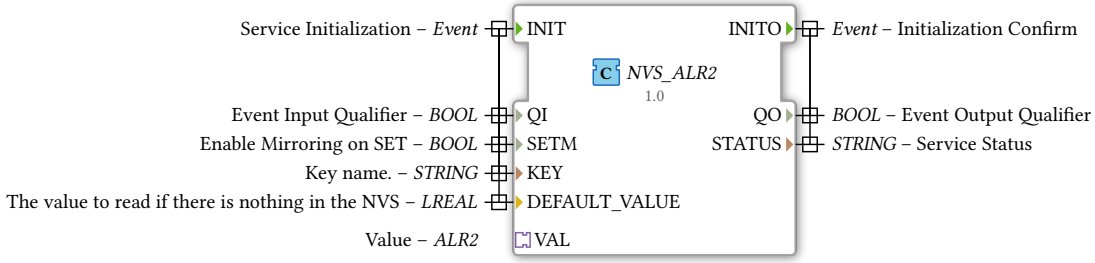
The **NVS_ALR** function block offers a compact and user-friendly way to persistently store LREAL values in an NVS and exchange them via a unidirectional adapter. By combining initialization, automatic reading,

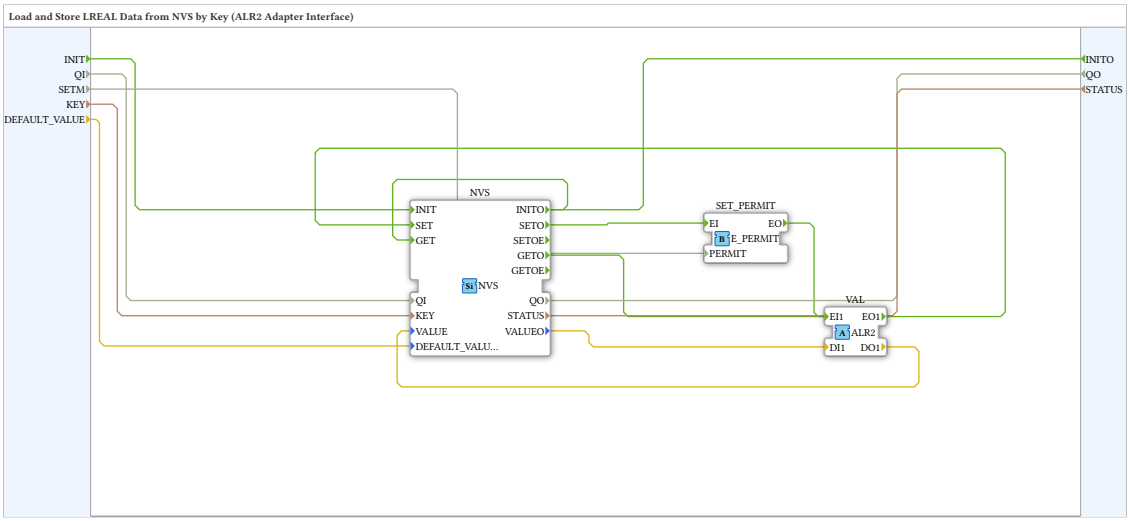
and adapter-based writing, it is particularly well-suited for modular and reusable ESP32-based control applications. Its design adheres to the IEC 61499 standard and enables a clean separation of memory logic and application.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

NVS_ALR2





• – ○ • – ○ • – ○ •

Introduction

The **NVS_ALR2** function block is used to load and store LREAL data in non-volatile storage (NVS) of an ESP32 via an adapter. It encapsulates initialization and access to a single NVS entry identified by a KEY.

Interface Structure

Event Inputs

Event	Type	Comment	Associated Variables
`INIT`	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Event Outputs

Event	Type	Comment	Associated Variables
`INITO`	EInit	Initialization Confirm	QO, STATUS

Data Inputs

Name	Type	Comment

QI	BOOL	Event Input Qualifier
SETM	BOOL	Enable Mirroring on SET: if TRUE, the confirmation of a SET (SETO) is additionally forwarded to the output adapter (mirrored)
KEY	STRING	Key name for NVS entry
DEFAULT_VALUE	LREAL	Default value if key does not exist in NVS (Default: 0.0)

Data Outputs

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

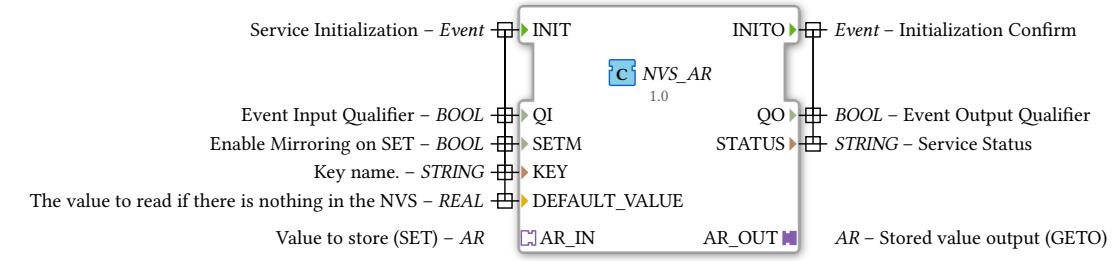
Adapters

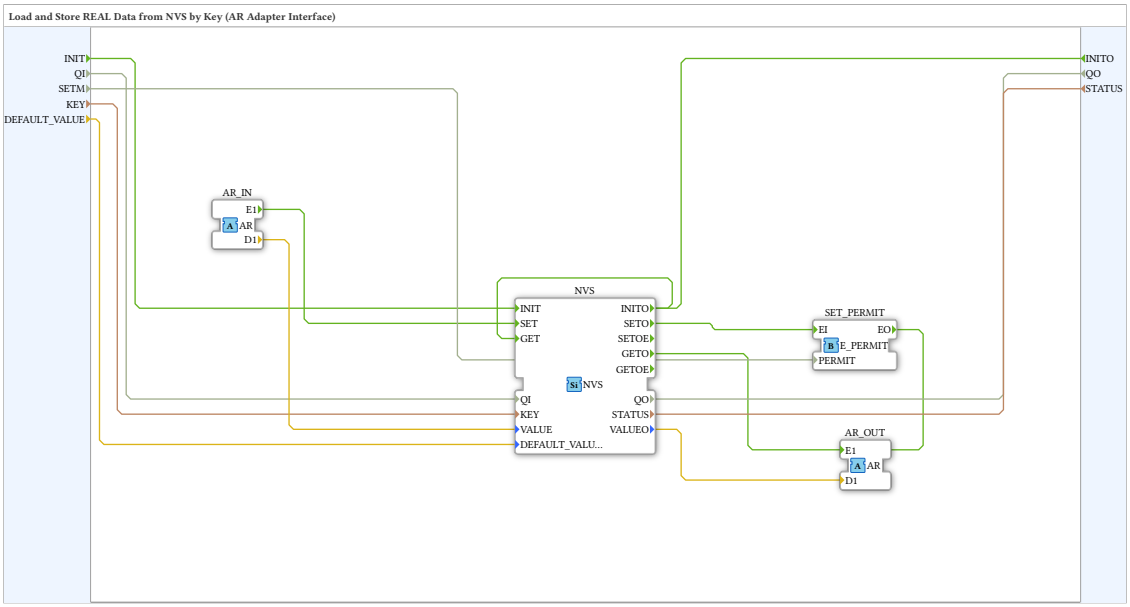
Adapter	Type	Comment
VAL	adapter::types::bidirectional::ALR2 (Socket)	Value (LREAL)

Functionality

1. **Initialization**: An INIT event initializes the internal NVS block.
2. **Read value**: The read value is output via VAL.DI1.
3. **Store value**: An external block sends E01 via VAL to write a new value (D01) to NVS.

NVS_AR





• — ○ • — ○ • — ○ •

Introduction

The function block NVS_AR is used to load and store REAL data in non-volatile storage (NVS) using a key. It provides unidirectional communication with other IEC 61499 function blocks via two AR adapter interfaces. The block encapsulates the internal NVS function block and extends its functionality with a standardized adapter connection.

Interface Structure

Event Inputs

- **INIT** (EInit) – Initializes the function block. Upon this event, the data inputs QI, SETM, KEY, and DEFAULT_VALUE are accepted.

Event Outputs

- **INITO** (EInit) – Confirms successful completion of the initialization. After this event, the outputs QO and STATUS are valid.

Data Inputs

- **QI** (BOOL) – Input qualifier; controls the execution of the initialization.
- **SETM** (BOOL) – Enable mirroring on SET.
- **KEY** (STRING) – Key name under which the value is stored/retrieved in the NVS.
- **DEFAULT_VALUE** (REAL) – Default value that is returned if no entry with the specified key exists in the NVS.

Data Outputs

- **QO** (BOOL) – Output qualifier; indicates successful initialization.
- **STATUS** (STRING) – Status message for error diagnosis or confirmation.

Adapters

- **AR_IN** (Adapter type adapter::types::unidirectional::AR) – Receives a value to be stored (SET event and data).
- **AR_OUT** (Adapter type adapter::types::unidirectional::AR) – Makes the read value available to other function blocks (GETO event and data).

Functionality

The function block operates as follows:

1. **Initialization**: An event at INIT starts the initialization of the internal NVS function block. The inputs QI, SETM, KEY, and DEFAULT_VALUE are forwarded to NVS.
2. **After Initialization**: Once NVS completes initialization (event INITO), a **read command** (GET) is automatically sent to NVS. The read value (or the default value) is then output via the adapter output AR_OUT. … 3. **Save**: An event received via the adapter input AR_IN (AR_IN.E1) triggers a **save command** (SET) in the internal NVS. The value to be saved is taken from the adapter data channel (AR_IN.D1).
3. **Feedback**: After both a read and a save operation, the result (success/error) is reported via the adapter event AR_OUT.E1 and via the data outputs QO/STATUS.

Technical Features

- The module uses the unidirectional AR adapter, which enables loose coupling between function blocks.
- It is specifically designed for REAL data; other data types cannot be processed.
- The implementation is based on the generic NVS block from the logiBUS::storage::esp32_nvs library.
- The event control ensures that the current value is automatically read from the NVS and made available after initialization.

State Overview

The NVS_AR block does not have an explicit state engine in its XML definition. Its functional behavior is determined by the event chaining:

- **Idle State**: The block waits for an event on INIT or AR_IN.E1.
- **Initialization Phase**: After INIT, the internal NVS block is started. The outputs are not yet active during this phase.

- **Operation Phase**: After `INITO`, the block is ready to read values (automatically) and have them saved via `AR_IN`.

Application Scenarios

- **Configuration Memory**: Storing and retrieving real-world configuration parameters (e.g., limits, factors) in an ESP32-NVS.
- **Calibration Data**: Storing calibration values for sensors or actuators.
- **Persistent Operational Data**: Storing count values or totals that should be retained during power outages.

Comparison with Similar Blocks

Feature	`NVS_AR`	Direct `NVS` block
Interface	AR adapter (unidirectional)	Standard event/data inputs
Intended use	Easy integration into adapter-based networks	Flexible, but more complex wiring
Data type	REAL	Multiple types (via generic parameter)

The `NVS_AR` simplifies integration by standardizing the adapter connection, while the pure `NVS` block offers broader type support.

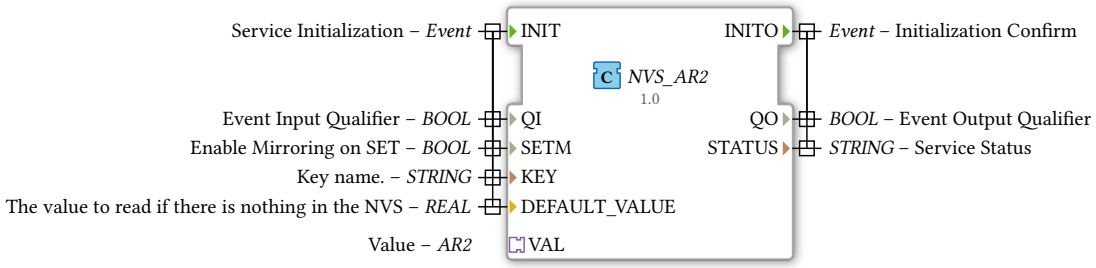
Conclusion

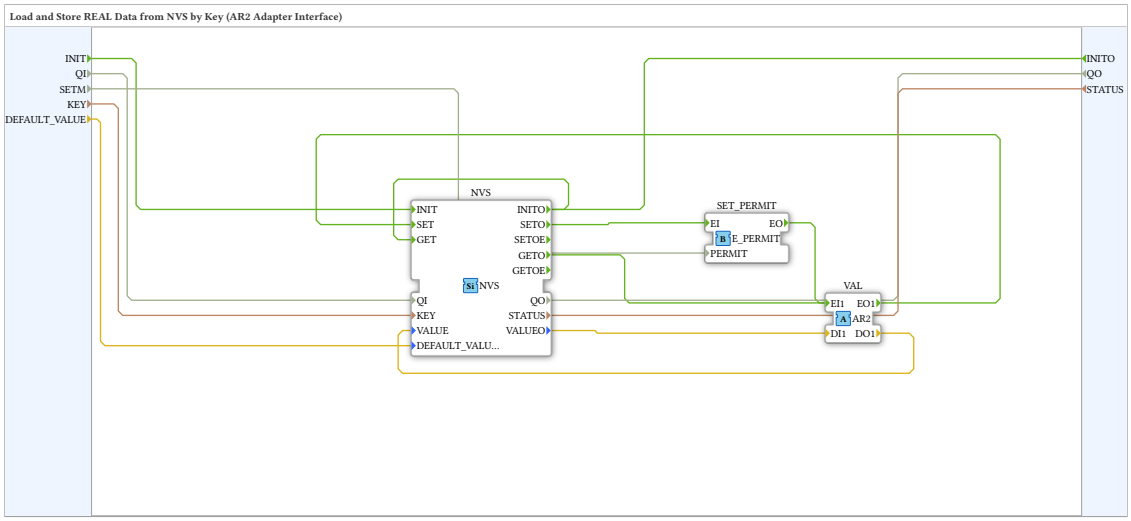
The `NVS_AR` function block offers a compact, adapter-based solution for loading and storing real data in non-volatile memory. Thanks to the AR adapter, it can be easily integrated into existing IEC 61499 networks without requiring detailed connections. Automatic initial readout after initialization simplifies its use in typical real-time control scenarios.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

NVS_AR2





• – ○ • – ○ • – ○ •

Introduction

The NVS_AR2 function block is used to store and load real values (floating-point numbers) into non-volatile storage (NVS) using a key. It encapsulates the logic of the underlying NVS block and provides a bidirectional AR2 adapter interface for asynchronous communication. This allows the block to be integrated into an event-driven environment where read and write operations are triggered asynchronously.

Interface Structure

Event Inputs

- **INIT**: Initializes the block. This event passes the parameters QI (Qualifier), SETM (mirroring on SET), KEY (Key), and DEFAULT_VALUE (Default Value). After successful initialization, the INITO output is triggered.

Event Outputs

- **INITO**: Initialization confirmation. This event signals that the function block is ready. It is also triggered after a read or write operation via the adapter (since the events are chained).

Data Inputs

- **QI** (BOOL): Input qualifier for controlling execution. Typically, this is set to TRUE to activate initialization.
- **SETM** (BOOL): Enable mirroring on SET.
- **KEY** (STRING): The key name under which the value is to be stored or read in the NVS.
- **DEFAULT_VALUE** (REAL): The value returned if no entry exists in the NVS under the specified key.

Data Outputs

- **QO** (BOOL): Output qualifier. Displays the status of the executed operation.
- **STATUS** (STRING): Status message of the operation (e.g., success, error code).

Adapter

- **VAL** (of type `adapter::types::bidirectional::AR2`): Bidirectional adapter for asynchronous reading and writing of values. The adapter has two events (EO1 – from the module to the adapter; EI1 – from the adapter to the module) and two data ports (DO1 – data from the module to the adapter; DI1 – data from the adapter to the module). External components can send read and write requests to the NVS_AR2 and receive results via this adapter.

Functionality

The NVS_AR2 acts as an intermediary between the actual NVS memory module (`logiBUS::storage::esp32_nvs::NVS`) and a higher-level controller via the bidirectional adapter. Upon

receipt of the INIT event, the parameters are forwarded to the internal NVS module. The NVS module performs initialization and acknowledges with INITO.

After initialization, a read operation (GET) is automatically triggered: The NVS block reads the value under the specified key. If no entry exists, DEFAULT_VALUE is used. The read result is sent to the connected component via the adapter output (VAL.DI1) and the event VAL.EI1.

A write operation occurs when the external component sends a write command via the adapter (VAL.EO1). The desired value is then passed to NVS_AR2 via VAL.DO1, which forwards it to the NVS block (SET). After a successful write, the NVS block acknowledges with SETO, which in turn triggers the event VAL.EI1 to send the confirmation back to the external component.

Technical Features

- **Asynchronous Communication**: The block supports asynchronous read and write operations via the AR2 adapter, allowing it to be integrated into concurrent systems without blocking calls.
- **Default Value**: Specifying a DEFAULT_VALUE ensures that a defined value is returned even when a key is used for the first time.
- **Error Handling**: STATUS and QO indicate the success or failure of the operation.
- **Reusability**: The function block encapsulates the general NVS function block and adds a standardized adapter interface, facilitating reuse in various control logics.

State Overview

The function block does not have an explicitly defined state machine in the XML, as it is controlled via the internal NVS function block and event chaining. The state is determined by the sequence of events:

1. Waiting for INIT → Initialization runs → INITO and GET are triggered.
2. After GET: Waiting for read result → Value sent to adapter.
3. Waiting for write request via adapter → SET executed → Confirmation returned.
4. After each step, the function block is ready for new requests.

Application Scenarios

- **Persistent Configuration Memory**: Saving and retrieving settings (e.g., setpoints, parameters) in an embedded system with NVS (such as ESP32 NVS).
- **Data Logging**: Asynchronously storing measured values in NVS.
- **Load Initial Default Values**: Loading a saved value at system startup or using a default value if it has never been saved.

Comparison with Similar Devices

- **NVS**: The basic NVS device without an adapter has direct event interfaces (INIT, REQ, etc.) and requires a direct connection. NVS_AR2 adds a bidirectional adapter interface, enabling a more flexible, protocol-independent connection.
- **Other Memory Devices (e.g., EEPROM, file system)**: These devices often have different data types and access methods. NVS_AR2 is specifically optimized for REAL values and NVS access.

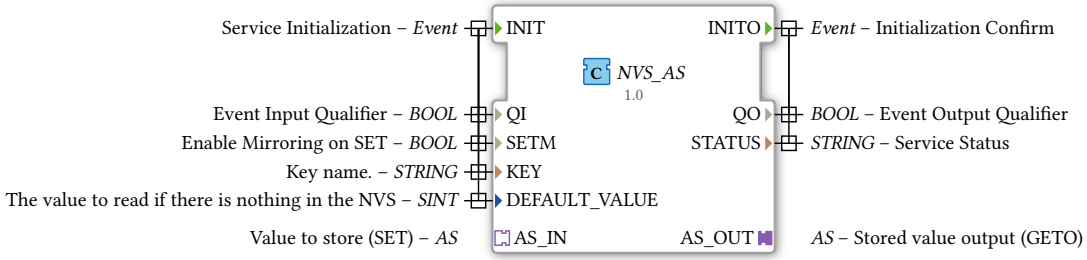
Conclusion

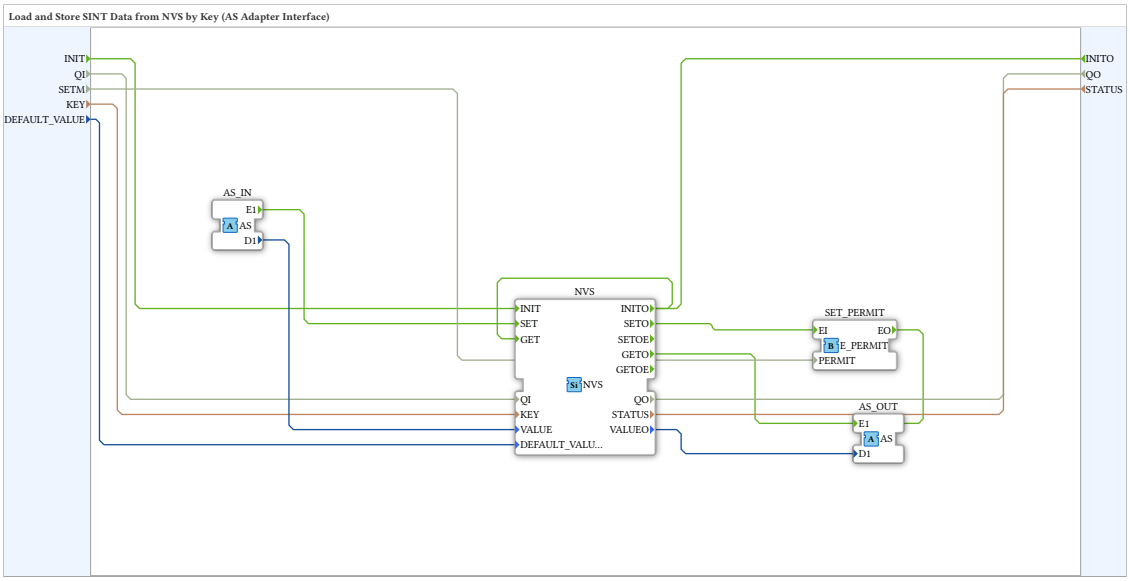
The NVS_AR2 function block offers a clean, adapter-based interface for asynchronously reading and writing real data to non-volatile memory. Thanks to the encapsulation of the NVS block and the integration of the AR2 adapter, it is ideally suited for modular, event-driven control systems that require persistent data management.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

NVS_AS





• – ○ • – ○ • – ○ •

The function block **NVS_AS** serves as an interface for reading and writing SINT (signed integer) data in non-volatile storage (NVS). Storage is performed using a user-defined key (KEY). The block extends access to the NVS with an adapter interface (unidirectional AS type), allowing values to be received and sent via standardized adapter connections. This enables modular and reusable connectivity to storage functions in IEC 61499 applications.

Event	Description	Associated Variables
INIT	Initializes the NVS and triggers the first loading of a stored value.	QI, SETM, KEY, DEFAULT_VALUE
Event	Description	Associated Variables
–––––	–––––	–––––––
INITO	Confirmation of initialization and the first loading process.	QO, STATUS
Variable	Type	Description
–––––	–––	–––––––
QI	BOOL	Enables initialization (TRUE = enable).
SETM	BOOL	Enable Mirroring on SET: if TRUE, the confirmation of a SET (SETO) is additionally forwarded to the output adapter (mirrored)
KEY	STRING	Key under which the value is stored in the NVS.

DEFAULT_VALUE	SINT	Default value, read if no value exists under the key.
Variable	Type	Description
-----	----	-------
QO	BOOL	Confirms successful initialization and operational readiness.
STATUS	STRING	Status message (e.g., “Success” or error message).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Adapter	Type	Direction	Description
AS_IN	AS	SOCKET	Receives a SINT value to be stored via a unidirectional adapter protocol (Event + Data).
AS_OUT	AS	PLUG	Sends the read SINT value to connected blocks via a unidirectional adapter protocol.

Functionality

The **NVS_AS** block encapsulates an instance of the **NVS** block (from the `logiBUS::storage::esp32_nvs` library). The function operates in two modes:

1. **Initialization and First Read Operation**

After an INIT event, the internal NVS block is initialized. Immediately afterward (automatic chaining of `INIT0` to `GET`), the value stored under the specified KEY is read. If no value exists, `DEFAULT_VALUE` is returned. The read or specified value is sent to downstream function blocks via the **AS_OUT** adapter (event `E1` and data `D1`).

1. **Writing and Reading via Adapters**

- An event (`E1`) with an associated data value (`D1`) received via the **AS_IN** adapter triggers a **SET** operation in the NVS. The value is stored under the KEY specified during INIT.
- After a successful SET, the stored value is automatically output via **AS_OUT** (through the connection between `NVS.SET0` and `AS_OUT.E1`).
- Similarly, a reread can be triggered by an INIT event or by the internal process after a successful SET. A separate external read event is not provided; the value is always updated after a change or during initialization.

The function block therefore operates as a **read and write memory access with automatic feedback of the current value**.

- **Adapter-Based Input/Output**

The use of unidirectional AS adapters allows for loose coupling: **AS_IN** receives write requests, **AS_OUT** outputs the stored value. This corresponds to a publisher/subscriber or client/server pattern at the adapter level.

- **Automatic Initialization**

After the INIT event, a GET request is executed immediately, so the function block provides the current or default value immediately after initialization.

- **Type Restriction to SINT**

The function block stores and loads only SINT values. Separate versions are required for other data types (e.g., INT, REAL, STRING).

- **Error Handling**

Errors during NVS operations (e.g., invalid key, memory error) are output as an error message via the STATUS output, and QO is set to FALSE.

The internal NVS module has its own state machine. The following processes are relevant for the user:

Phase	State
1	Waiting for INIT event.
2	INIT executed ⟶ NVS initialized ⟶ GET started.
3	GET completed ⟶ Value sent via AS_OUT ⟶ Ready for write operations via AS_IN.
4	Event via AS_IN ⟶ SET executed ⟶ Value saved ⟶ Value sent via AS_OUT ⟶ Return to Phase 3.

A new INIT event can force a re-initialization at any time.

- **Persistent Device Parameters**

Storing configuration values (e.g., brightness, delay time) in the non-volatile memory of an ESP32, automatically loading them on restart.

- **State Markers for Automation**

Remembering the last state (e.g., counter reading, production parameters) even after a power failure.

- **Adapter-Based Data Nodes**

Integration into a chain of adapters where one module sets values and another reads them.

- **NVS (Direct)**

The `NVS` module offers the same functionality, but without an adapter interface. It requires separate event and data lines. `NVS_AS` simplifies integration into adapter-oriented architectures.

- **NVS_AS_REAL, NVS_AS_STRING**

Analogous modules for other data types. The interface and behavior are identical; only the data type varies.

- **Retain Values**

In some systems, retain variables are also persistent. `NVS_AS` relies on low-level NVS (e.g., on ESP32) and is therefore platform-specific, but offers higher performance and optimized capacity.

The **NVS_AS** function block provides a practical, adapter-based encapsulation of non-volatile memory for SINT values. By combining initialization logic, automatic value feedback, and an adapter interface, it is ideally suited for modular IEC 61499 projects that require persistent storage with minimal wiring. The limitation to the SINT type and the automatic GET post-initialization should be noted, but simultaneously simplify handling in many standard applications.

- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

Technical Features

State Overview

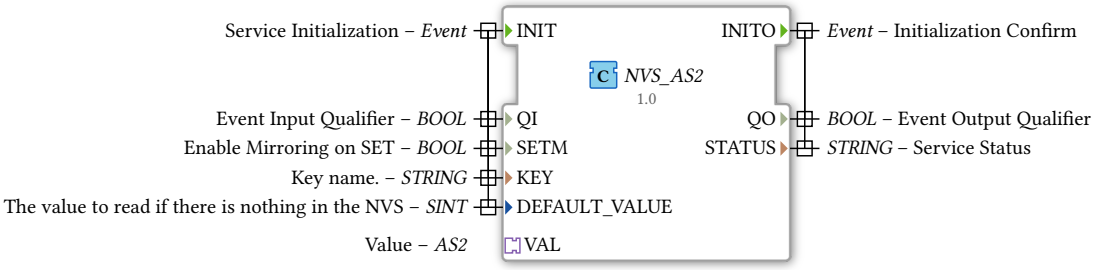
Application Scenarios

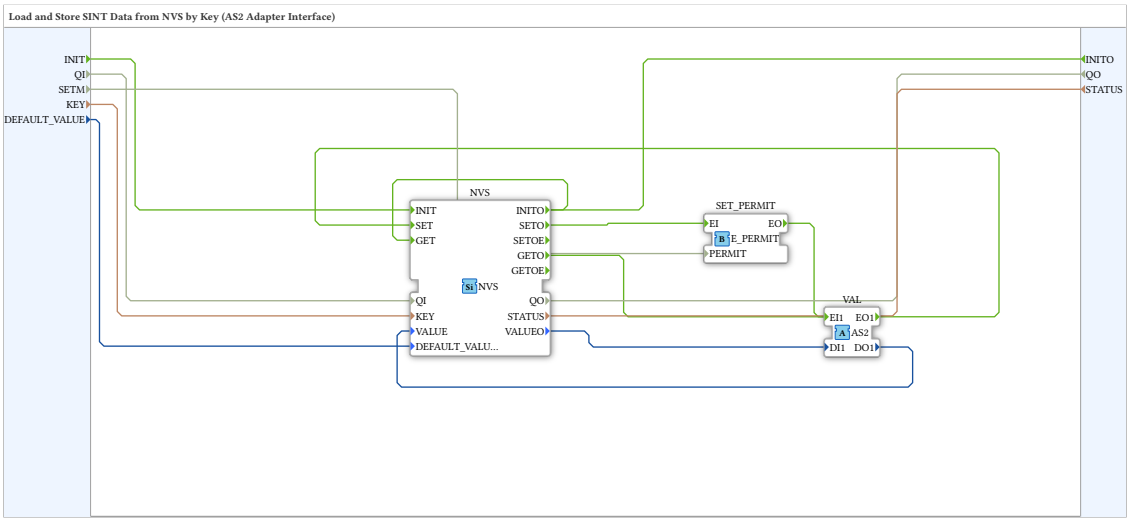
Comparison with Similar Function Blocks

Conclusion

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

NVS_AS2





• – ○ • – ○ • – ○ •

Introduction

The **NVS_AS2** function block is used to load and store SINT data in non-volatile storage (NVS) of an ESP32 via an adapter. It encapsulates initialization and access to a single NVS entry identified by a KEY.

Interface Structure

Event Inputs

Event	Type	Comment	Associated Variables
`INIT`	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Event Outputs

Event	Type	Comment	Associated Variables
`INITO`	EInit	Initialization Confirm	QO, STATUS

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier
SETM	BOOL	Enable mirroring on SET
KEY	STRING	Key name for NVS entry
DEFAULT_VALUE	SINT	Default value if key does not exist in NVS (Default: 0)

Data Outputs

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

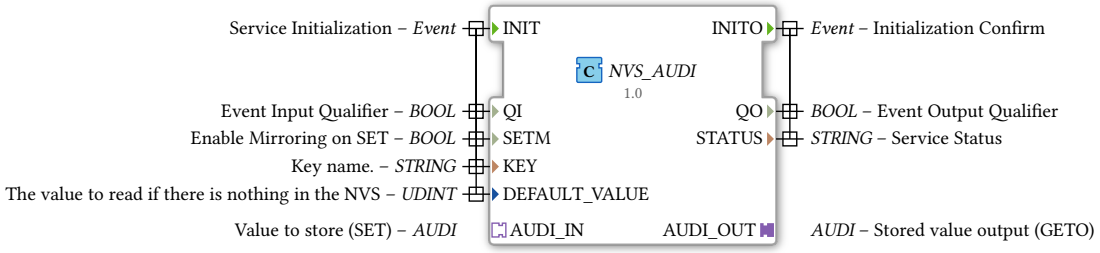
Adapters

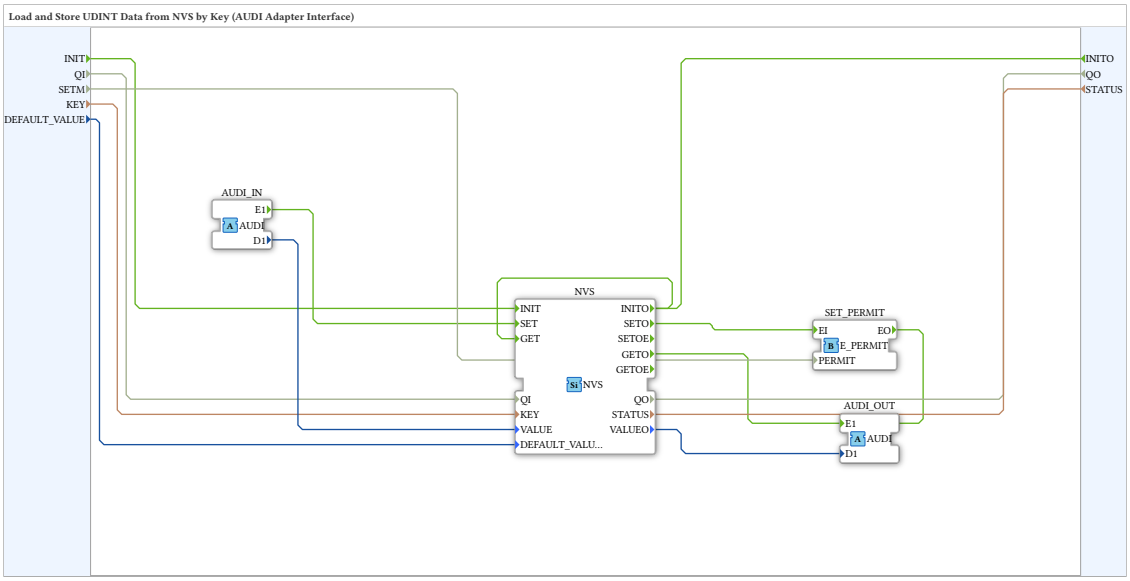
Adapter	Type	Comment
VAL	adapter::types::bidirectional::AS2 (Socket)	Value (SINT)

Functionality

1. **Initialization**: An INIT event initializes the internal NVS block.
2. **Read value**: The read value is output via VAL.DI1.
3. **Store value**: An external block sends E01 via VAL to write a new value (D01) to NVS.

NVS_AUDI





• – ○ • – ○ • – ○ •

Introduction

The **NVS_AUDI** function block enables the storage and loading of **UDINT** data in the **Non-Volatile Storage (NVS)** of an ESP32 microcontroller. The data is addressed using a unique **key (KEY)**. The block uses a **unidirectional AUDI adapter interface** to communicate with the NVS: The value to be stored is received via an input adapter (socket), and the read value is provided via an output adapter (plug).

Interface Structure

Event Inputs

Name	Type	Data Carried	Comment
INIT	EInit	QI, SETM, KEY, DEFAULT_VALUE	Initializes the service

Event Outputs

Name	Type	Carried Data	Comment
INITO	EInit	QO, STATUS	Initialization Confirmation

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier
SETM	BOOL	Enable mirroring on SET
KEY	STRING	Key Name for NVS Access
DEFAULT_VALUE	UDINT	Default value if no value exists in NVS

Data Outputs

Name	Type	Comment
QO	BOOL	Event Output Qualifier
STATUS	STRING	Service Status (Error/OK)

Adapter

Name	Type	Comment
AUDI_IN	adapter::types::unidirectional::AUDI	Value to be stored (SET)
AUDI_OUT	adapter::types::unidirectional::AUDI	Stored/loaded value (GETO)

Functionality

Upon arrival of the **INIT** event, the internal NVS module is initialized (QI, SETM, KEY, and DEFAULT_VALUE are passed). After successful initialization, **INITO** fires, and simultaneously a **GET** request is sent to the NVS to load the stored value. The result of the GET operation is provided via **AUDI_OUT** (data on D1, event on E1).

If an event (E1) with a value (D1) is received via the **AUDI_IN** adapter, a **SET** request is sent to the NVS to store the value under the current KEY. After saving, **SETO** fires, and the result is also acknowledged via **AUDI_OUT**.

The status (QO, STATUS) is updated via the outputs after both loading and saving.

Technical Features

- The function block uses the **unidirectional AUDI adapter**, which allows for simple, stream-based data transfer: A value is sent (SET), and a value is received (GETO).
- Internally, the function block `logiBUS::storage::esp32_nvs::NVS` is used, which executes the actual NVS operation on the ESP32.

- The default data type is **UDINT** (unsigned 32-bit).
- No additional states or timers are required; the function block operates purely event-driven.
- The **TypeHash** attribute is set to an empty string – it can be used for integrity checks.

State Overview

Since no explicit state machine is defined in the XML, the process is derived from the events:

1. **Initialization Phase** – INIT received ⟶ NVS-INIT ⟶ NVS-GET triggered.

Afterwards: Ready (waiting for SET or another GET by re-INIT).

1. **Storage Phase** – Event via AUDI_IN (E1) ⟶ NVS-SET ⟶ Value is written.
2. **Read Operation** – Automatically after INIT or by re-INIT.

Output via AUDI_OUT occurs after each read or write operation.

Application Scenarios

- Storing **configuration parameters** (e.g., limits, operating modes) in the NVS, which are retained even after a power failure.
- **Meter readings** or **calibration data** in ESP32-based industrial controllers.
- **Persistent states** of automation processes, uniquely identified by a key.
- Exchanging data between different FBs via the AUDI adapter without directly accessing the NVS.

Comparison with Similar Function Blocks

- **NVS (without adapter)**: Provides direct access to the NVS via individual events and data ports, but requires more wiring.
- **NVS_AUDI** encapsulates the NVS operations behind a standardized adapter interface. This makes the module easier to replace and standardizes data transfer (e.g., with other memory modules that also use AUDI adapters).

Other memory modules (e.g., Retain, SD card) differ in their hardware interface and latency; NVS is specifically optimized for the ESP32.

Conclusion

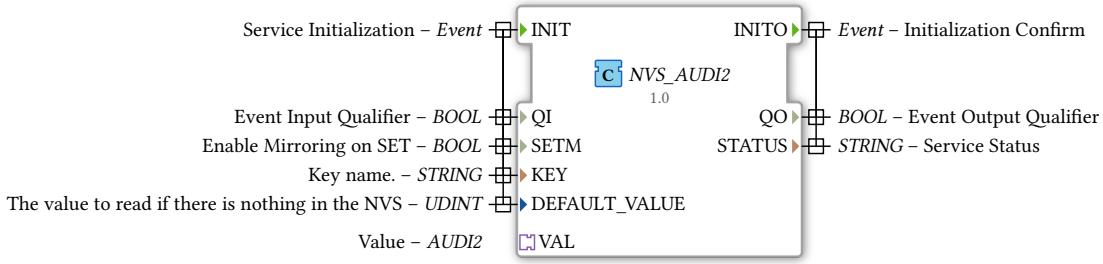
The NVS_AUDI function block provides an elegant solution for persistent storage of UDINT data on the ESP32. The integration of the AUDI adapter interface simplifies connectivity to other modules and results in clearer code structure. It is particularly suitable for applications requiring standardized data exchange between storage and processing components.

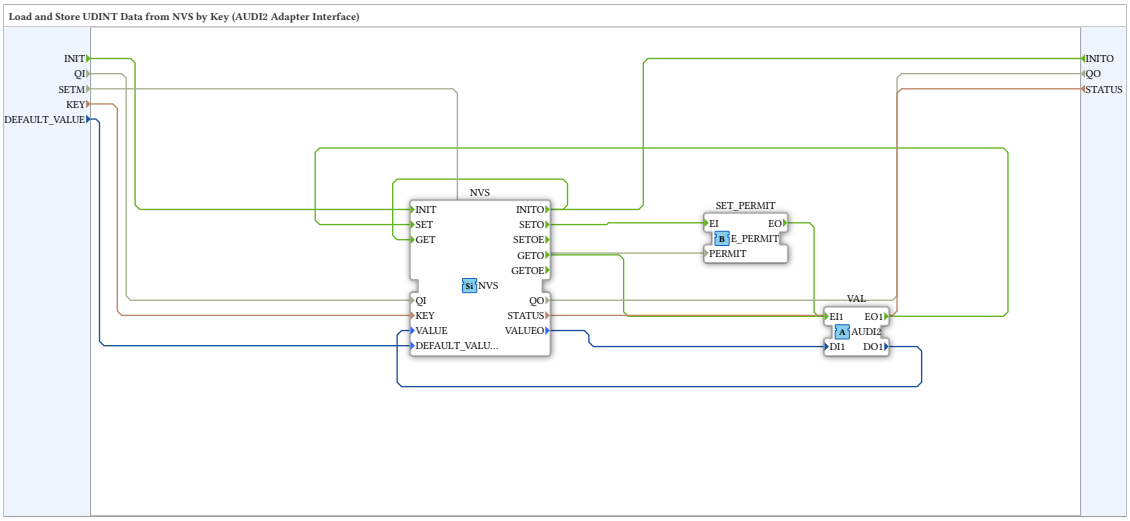
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de
- 🌐 MCU vs. MPU comparison guide on ms-muc-docs.de

]

NVS_AUDI2





• — ○ • — ○ • — ○ •

Introduction

The **NVS_AUDI2** function block is used to load and store UDINT data in non-volatile storage (NVS) of an ESP32 via an adapter. It encapsulates initialization and access to a single NVS entry identified by a KEY.

Interface Structure

Event Inputs

Event	Type	Comment	Associated Variables
INIT	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Event Outputs

Event	Type	Comment	Associated Variables
INITO	EInit	Initialization Confirm	QO, STATUS

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier
SETM	BOOL	Enable Mirroring on SET: if TRUE, the confirmation of a SET (SETO) is additionally forwarded to the output adapter (mirrored)
KEY	STRING	Key name for NVS entry
DEFAULT_VALUE	UDINT	Default value if key does not exist in NVS (Default: 0)

Data Outputs

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

Adapters

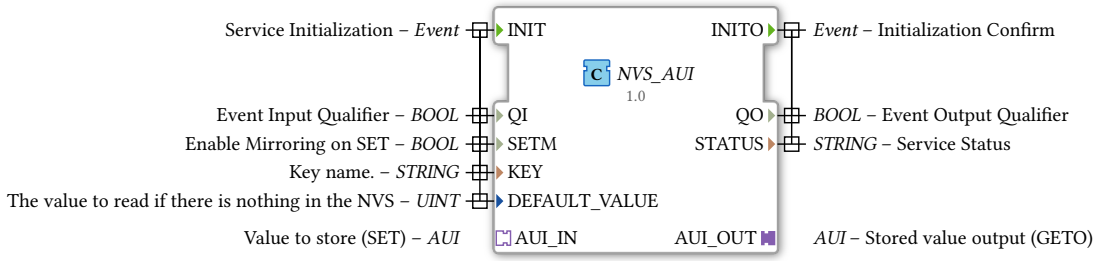
Adapter	Type	Comment

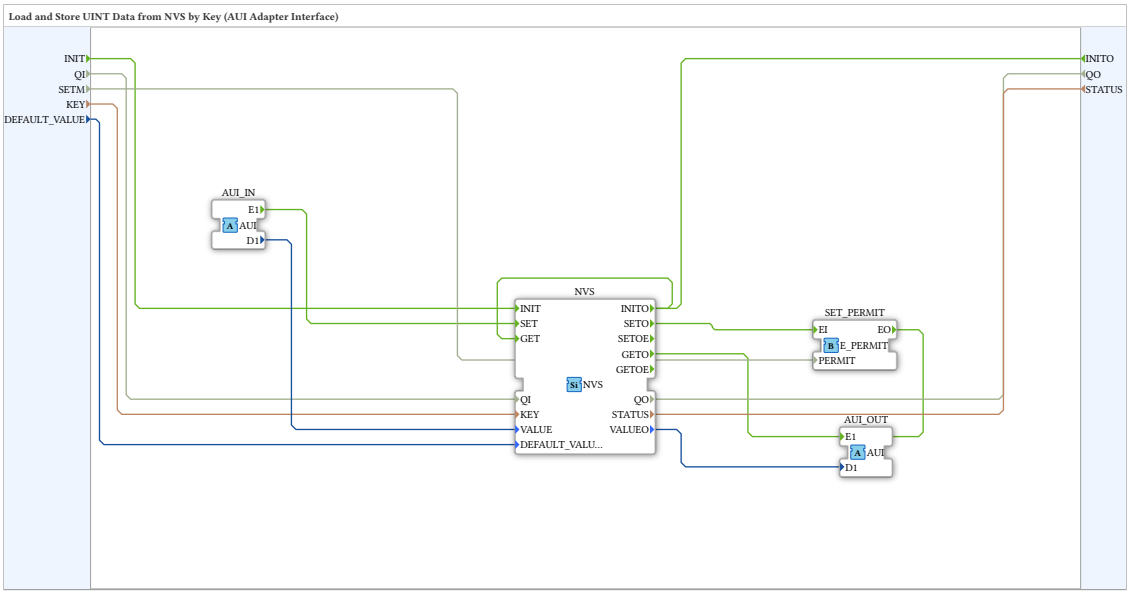
VAL	adapter::types::bidirectional::AUDI2 (Socket)	Value (UDINT)

Functionality

1. **Initialization**: An INIT event initializes the internal NVS block.
2. **Read value**: The read value is output via VAL.DI1.
3. **Store value**: An external block sends E01 via VAL to write a new value (D01) to NVS.

NVS_AUI





• — ○ • — ○ • — ○ •

Introduction

The function block **NVS_AUI** is used to load and store UINT data in non-volatile memory (NVS) using a key. It encapsulates access to the NVS and provides a standardized AUI adapter interface for outputting read values and receiving values to be stored. Typical applications include embedded systems that need to persistently store configuration or status data.

Interface Structure

Event Inputs

Event	Comment	Accompanying Data
INIT	Service Initialization	QI (BOOL), SETM (BOOL), KEY (STRING), DEFAULT_VALUE (UDINT)

Event Outputs

Event	Comment	Carrying Data
INITO	Initialization Acknowledgement	QO (BOOL), STATUS (STRING)

Data Inputs

Name	Type	Comment
QI	BOOL	Qualifier for the input event
SETM	BOOL	Enable mirroring on SET
KEY	STRING	Key name for NVS access
DEFAULT_VALUE	UDINT	Value to be read if no entry exists in the NVS

Data Outputs

Name	Type	Comment
QO	BOOL	Qualifier for the output event
STATUS	STRING	Service status message

Adapters

Adapter	Type / Direction	Comment
AUI_IN	**Socket** (Input) – Type: adapter::types::unidirectional::AUI	Returns the value to be stored (SET)
AUI_OUT	**Plug** (Output) – Type: adapter::types::unidirectional::AUI	Returns the retrieved value (GETO)

Functionality

1. **Initialization**

A INIT event triggers the connection to the non-volatile memory. The key (KEY) and the default value (DEFAULT_VALUE) are passed. After successful initialization, INITO is acknowledged with the result (QO / STATUS).

1. **Reading Data**

After initialization, the stored value for the specified key is automatically read internally and provided as an event (E1) and data (D1) via the output adapter AUI_OUT. If no entry exists, DEFAULT_VALUE is output.

1. **Storing Data**

A new value can be received via the input adapter AUI_IN. An incoming event (E1) with the associated data (D1) is forwarded to the internal NVS function block, which persistently stores the value under the initialized key.

The entire data and event flow is implemented by the embedded function block NVS (from the library logiBUS::storage::esp32_nvs).

Technical Features

- **AUI Adapter Interface**

The unidirectional adapters AUI_IN and AUI_OUT enable loose coupling between the NVS module and other components that use standardized AUI interfaces.

- **Automatic Read Operation After Initialization**

Immediately after initialization, the value is read and made available via AUI_OUT – without a separate "Get" event.

- **Error Handling**

Errors during initialization or memory accesses are signaled via STATUS.

Status Overview

The FB is in **idle state** after startup.

- **Initialization** (upon arrival of `INIT`): Establishing the NVS connection, reading the existing value.
- **Ready** (after successful `INITO`): Waiting for incoming data via `AUI_IN` to save or for external requests via `AUI_OUT`.
- **Save operation** (upon event at `AUI_IN`): Value is persisted; then returns to the ready state.
- **Error state** (in case of unsuccessful initialization): `QO` = FALSE, `STATUS` contains error text.

Application Scenarios

- **Configuration Memory** in IoT devices: Storing network settings, calibration data, or user settings.
- **Status Monitoring**: Saving the last operating state (e.g., counter readings) before a power failure.
- **Parameter Transfer** between different automation components that communicate via AUI adapters.

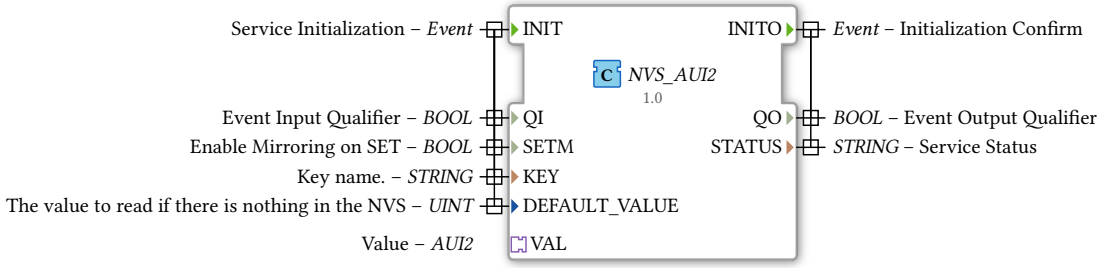
Comparison with Similar Function Blocks

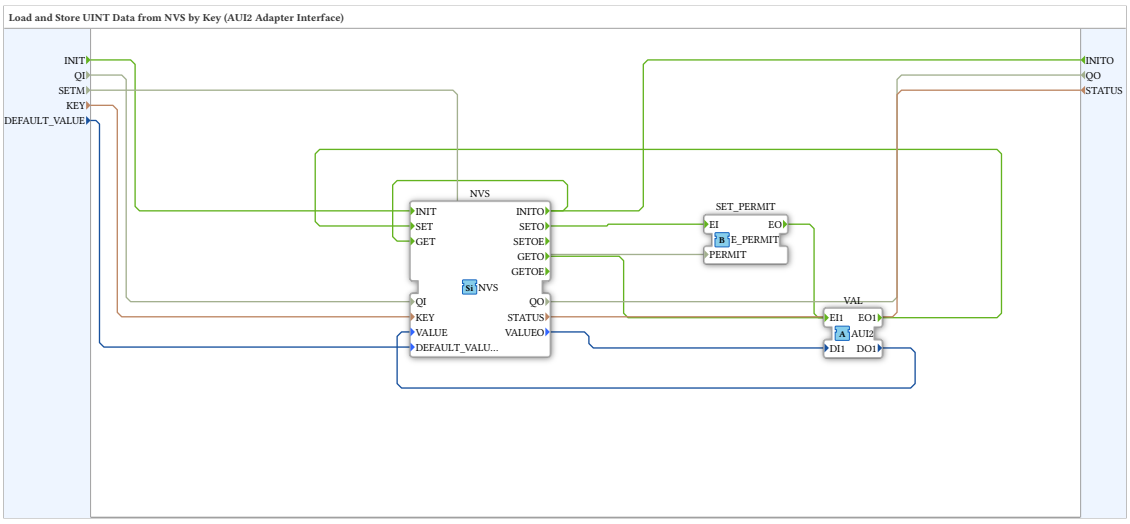
Compared to the direct **NVS** function block, `NVS_AUI` offers a higher level of abstraction through its adapter interface. While the basic function block requires separate events for reading (`GET`) and writing (`SET`), `NVS_AUI` simplifies data flow through unidirectional adapters and automatic read operation after INIT. Other memory modules (e.g., `F_MMAP`, `F_FILE`) typically use file-based or memory-mapping methods; `NVS_AUI` is specifically optimized for non-volatile memory with key-value pairs.

Conclusion

NVS_AUI` is a practical functional block for persistent data storage in automation systems. The combination of NVS memory and an AUI adapter allows for easy integration into existing IEC 61499 applications, reduces wiring effort, and ensures a clear separation between memory logic and communication.

NVS_AUI2





• – ○ • – ○ • – ○ •

Introduction

The **NVS_AUI2** function block is used to load and store UINT data in non-volatile storage (NVS) of an ESP32 via an adapter. It encapsulates initialization and access to a single NVS entry identified by a KEY.

Interface Structure

Event Inputs

Event	Type	Comment	Associated Variables
INIT	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Event Outputs

Event	Type	Comment	Associated Variables
INITO	EInit	Initialization Confirm	QO, STATUS

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier
SETM	BOOL	Enable mirroring on SET
KEY	STRING	Key name for NVS entry
DEFAULT_VALUE	UINT	Default value if key does not exist in NVS (Default: 0)

Data Outputs

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

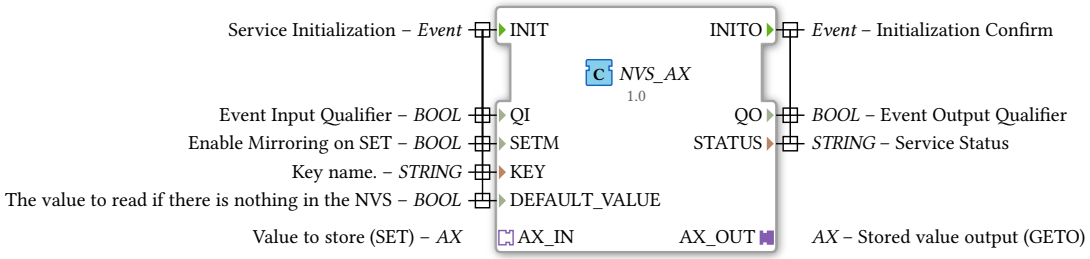
Adapters

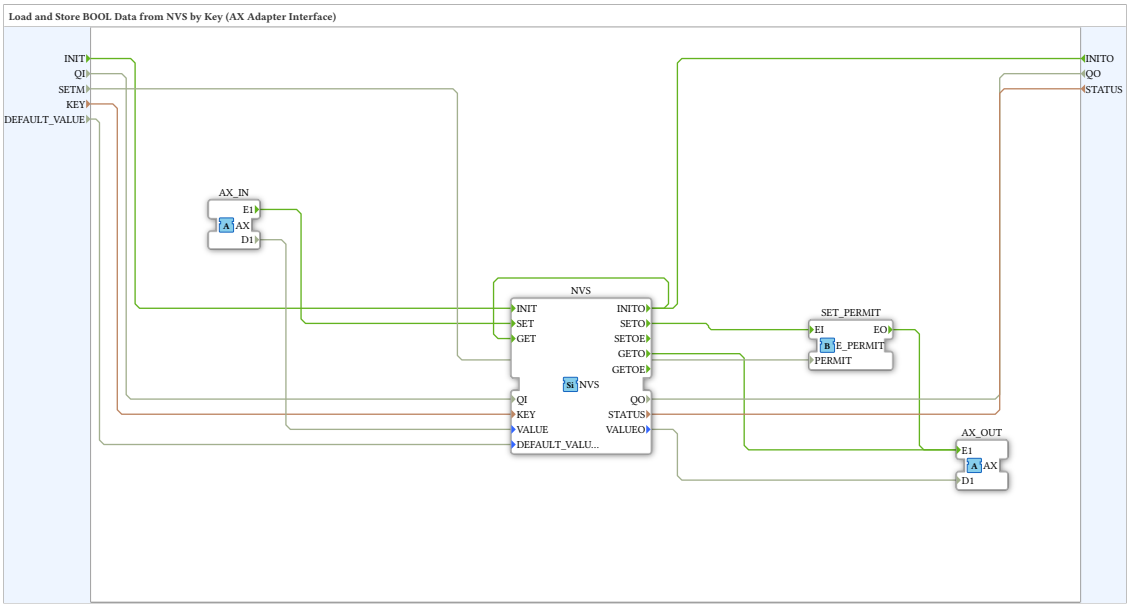
Adapter	Type	Comment
VAL	adapter::types::bidirectional::AUI2 (Socket)	Value (UINT)

Functionality

1. **Initialization**: An INIT event initializes the internal NVS block.
2. **Read value**: The read value is output via VAL.DI1.
3. **Store value**: An external block sends E01 via VAL to write a new value (D01) to NVS.

NVS_AX





• – ○ • – ○ • – ○ •

Introduction

`NVS_AX` loads and stores a `BOOL` value under a key (`KEY`) in the ESP32's non-volatile storage (NVS). It is the Boolean counterpart to NVS_AB and uses the same two-adapter architecture, but with unidirectional `AX` adapters instead of `AB`.

Interface Structure

Event Inputs

- **INIT**: initializes the block and reads the value stored under `KEY` (or `DEFAULT_VALUE` if none exists yet). Carries `QI`, `SETM`, `KEY`, `DEFAULT_VALUE`.

Event Outputs

- **INITO**: confirms initialization and the initial read, carries `QO` and `STATUS`.

Data Inputs

- **QI** (BOOL): qualifier for the `INIT` event.
- **SETM** (BOOL): enables mirroring (confirmation) on write (`SET`).
- **KEY** (STRING): name of the key in the NVS.
- **DEFAULT_VALUE** (BOOL): value returned when nothing is stored under `KEY` yet.

Data Outputs

- **QO** (BOOL): success qualifier, carried with `INITO` and the internal adapter confirmations.
- **STATUS** (STRING): status message of the NVS access.

Adapters

- **AX_IN** (Socket, type `AX`): new value to store – an adapter event on `AX_IN.E1` triggers a `SET` on the internal NVS block, `AX_IN.D1` carries the value to write.
- **AX_OUT** (Plug, type `AX`): outputs the most recently read value – both after the automatic initial read on `INIT` and after each confirmed write (`AX_OUT.D1`/`AX_OUT.E1`).

Functionality

`NVS_AX` wraps an internal instance of `logiBUS::storage::esp32_nvs::NVS` plus an `E_PERMIT` gate (`SET_PERMIT`). On `INIT`, the internal NVS instance is initialized (`NVS.INIT`) and a `GET` is triggered automatically, returning the value stored under `KEY` (or `DEFAULT_VALUE`); the result is output via `AX_OUT`, and `INITO` confirms completion.

To write, the block connected to `AX_IN` fires an adapter event (`AX_IN.E1`) carrying the new value (`AX_IN.D1`), which triggers `NVS.SET`. Whether the successful write is additionally reported back via `AX_OUT`

is controlled by SETM: if SETM is set, SET_PERMIT releases the AX_OUT.E1 event after a successful SET ("mirroring").

Technical Details

- Pure composite FB with no ECC of its own – the actual flash logic lives in the internal NVS instance; NVS_AX only wires events and data between it and the two AX adapters.
- Two separate unidirectional adapters (AX_IN/AX_OUT) – this lets the read and write paths connect to different peers.
- SETM only controls whether a write is additionally mirrored back via an AX_OUT adapter event – the value is stored either way.

State Overview

The block is stateless in terms of its own ECC: on INIT it runs through initialization plus an initial read once, then reacts to every AX_IN adapter event with an NVS write and, optionally, a confirmation via AX_OUT.

Application Scenarios

- Persisting individual BOOL configuration values (e.g. enable flags, operating-mode switches) on the ESP32
- Building a read/write pair where the read and write sides need to connect to different blocks
- Restoring a last-stored BOOL value after a restart via AX_OUT

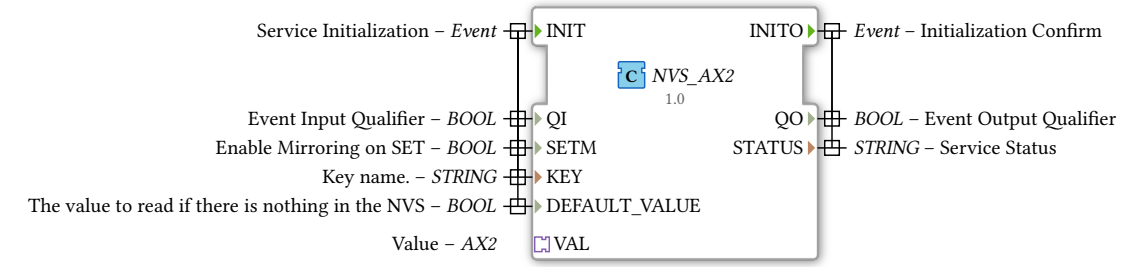
⚖️ Comparison with Similar Blocks

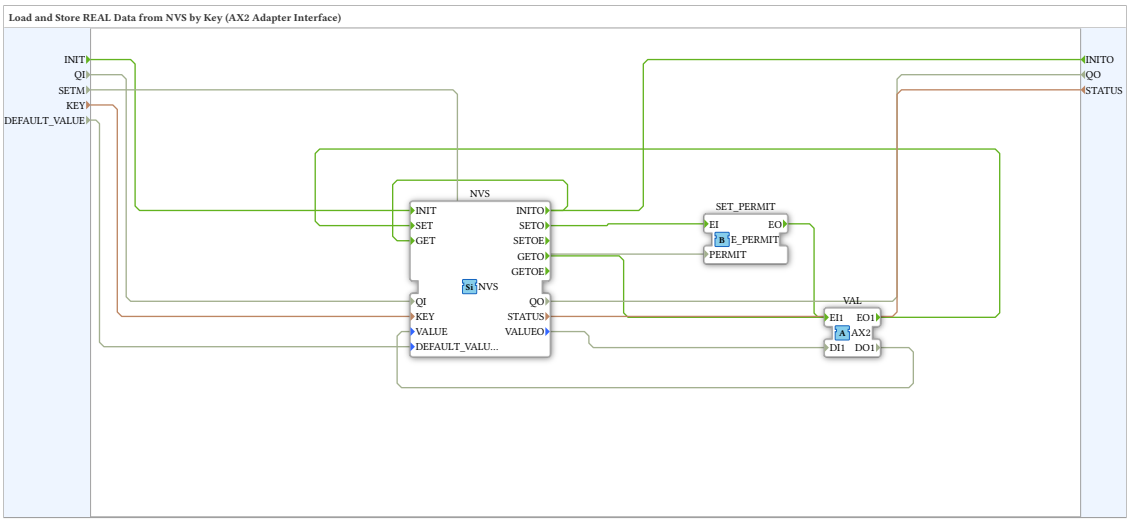
Compare with NVS_AB, which uses the same architecture for BYTE instead of BOOL values, and with INI_AX, which provides the same interface for the text-based settings.ini instead of the NVS flash storage.

Conclusion

NVS_AX provides simple, unidirectionally wired access to a persistent BOOL value in the ESP32's NVS, and fits anywhere the read and write paths need to be kept separate.

NVS_AX2





• – ◦ • – ◦ • – ◦ •

Introduction

The function block **NVS_AX2** is used to load and save REAL data to the non-volatile memory (NVS) of an ESP32 via an adapter. It encapsulates the initialization and access to a single NVS entry, which is identified by a key. The function block provides an initializing event interface and a bidirectional adapter interface through which the stored value can be accessed externally. The block is modeled according to IEC 61499 and optimized for use in distributed automation systems.

Interface Structure

Event Inputs

Event	Type	Comment	Carried Variables
INIT	EInit	Service Initialization	QI, SETM, KEY, DEFAULT_VALUE

Event Outputs

Event	Type	Comment	Carried Variables
INITO	EInit	Initialization Confirm	QO, STATUS

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier (Initialization Enabled)
SETM	BOOL	Enable Mirroring on SET: if TRUE, the confirmation of a SET (SETO) is additionally forwarded to the output adapter (mirrored)
KEY	STRING	Key name for the NVS entry
DEFAULT_VALUE	BOOL	Default value if the key does not exist in the NVS (default: FALSE)

Data Outputs

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

Adapters

Adapter	Type	Comment
VAL	adapter::types::bidirectional::AX2 (Socket)	Value (REAL)

The **VAL** adapter establishes a bidirectional connection to an external component. The stored REAL value can be read and written via this adapter. The AX2 adapter type defines an event and data protocol for data transfer.

Functionality

1. **Initialization**: An event at input INIT triggers the initialization of the internal NVS component. The values QI, SETM, KEY, and DEFAULT_VALUE are passed. After successful initialization, the event INITO is sent at the output. Simultaneously, a read operation (GET) for the specified key is automatically initiated.
2. **Read Value**: The read value is output via the adapter VAL as DI1 (data output of the adapter), and an event EI1 is triggered on the adapter. The outputs QO and STATUS reflect the result of the read operation.

3. **Save Value**: An external function block can send an event `EO1` via the adapter `VAL` to write a new value (provided as `DO1`) to the NVS. The internal NVS function block then performs a SET operation and confirms it via `SETO`. The event `EI1` is then triggered on the adapter to signal completion to the sender.
4. **Status Signaling**: The outputs `QO` and `STATUS` are updated after each read or write operation and output with the event `INITO`.

Technical Features

- **Persistence on ESP32**: The function block uses the ESP32 microcontroller's NVS (Non-Volatile Storage) to permanently store data. The data is retained even after a restart.
- **Adapter-Based Communication**: The interface to the user is via a bidirectional adapter (AX2), which enables flexible and standardized connectivity to other components. The data type is set to REAL.
- **Error Handling**: Error messages (e.g., NVS error, insufficient memory) can be queried via the output `STATUS`. The output `QO` indicates the success of the operation.
- **Standard Compliance**: The function block (FB) is implemented according to IEC 61499 and uses the block class `logiBUS::storage::esp32_nvs::NVS`.

State Overview

The FB does not have an explicit state machine, but the following logical sequence occurs:

1. **Idle**: No `INIT` event is received.
2. **Initializing**: After `INIT` – the internal NVS block is started and a GET request is executed.
3. **Ready**: After `INITO` – the module waits for read/write requests via the adapter.
4. **Write Operation**: A `EO1` event on the adapter triggers a SET operation.
5. **Read**: A subsequent read operation can only be performed by another `INIT` event or internal logic (the module reads once at startup and subsequently only during write operations).

Application Scenarios

- **Configuration Storage**: Saving user settings (e.g., setpoints, operating modes) to an ESP32 device.
- **Calibration Data**: Permanent storage of calibration values for sensors.
- **State Storage**: Saving the last operating state (e.g., counter readings, switch positions) in the event of a power failure.
- **Data Logging**: Targeted saving of individual measured values via the adapter.

Comparison with Similar Function Blocks

Unlike simple memory function blocks (e.g., `CTUO` or generic `R_TRIG` blocks), `NVS_AX2` is specifically designed for persistent storage on ESP32 hardware. While other function blocks only process volatile data, this block ensures permanent data retention. Using an adapter offers more flexibility than a fixed input/output because the data type and communication direction are clearly defined. Compared to direct NVS API calls in IEC 61499, this function block simplifies integration through its fully declarative description.

Conclusion

The function block `NVS_AX2` is an efficient and standards-compliant solution for persistently storing a REAL value in the NVS of an ESP32. Its clear interface structure with event control and bidirectional adapter facilitates integration into complex automation projects. Thanks to integrated initialization and error logging, the block is robust and suitable for industrial applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de
- 🌐 MCU vs. MPU Comparison Guide on ms-muc-docs.de

]

INI

🎧 Podcast

- The E_CTU in IEC 61499: Event-driven counting and why the minimalist approach is convincing in mechanical engineering
- From "Mass Errors" to Masterpiece: Streamlining Industrial Software by Eliminating Mapping

- The tracked monster awakens: Lanz Bulldog Caterpillar – The fascinating revival of the 10-liter hot-bulb workhorse after 25 years of inactivity
- Masters of change: How the curriculum for agricultural and construction machinery mechatronics technicians redefines lifelong learning
- Miniware TS101: The mobile soldering all-rounder – Strengths, weaknesses, and the USB-C Revolution

Introduction

The INI function block enables the loading and saving of data to a `settings.ini` file. It provides a simple interface for persistently managing values using a section name (SECTION) and a key (KEY). The block is particularly useful for configuration management in 4diac FORTE applications, for example, for setting and retrieving parameters such as IP addresses, thresholds, or operating modes.

Interface Structure

Event Inputs

- **INIT**: Service initialization. Used during startup or reconfiguration. Triggers the initialization of the block and, optionally, the first read/write of a value.
- **SET**: Sets the value for the configured key in the INI file.
- **GET**: Reads the value for the configured key from the INI file.

Event Outputs

- **INITO**: Initialization confirmation. Triggered after processing the INIT event.
- **SETO**: Write confirmation. Triggered after processing the SET event.
- **GETO**: Read confirmation. Triggered after processing the GET event.

Data Inputs

- **QI (BOOL)**: Event input qualifier. Controls whether the operation is enabled (`TRUE`) or disabled (`FALSE`) upon INIT.
- **SECTION (STRING)**: The name of the section within the `settings.ini` file.
- **KEY (STRING)**: The name of the key within the specified section.
- **VALUE (ANY_ELEMENTARY)**: The value to be written. This data type can be any elementary type (e.g., BOOL, INT, REAL, STRING).
- **DEFAULT_VALUE (ANY_ELEMENTARY)**: The default value returned if the requested key does not exist in the INI file.

Data Outputs

- **QO (BOOL)**: Event output qualifier. Returns the status of the operation (`TRUE` for success, `FALSE` for error or deactivation).
- **STATUS (STRING)**: Detailed status of the executed operation (e.g., “OK”, “ERROR”).
- **VALUEO (ANY_ELEMENTARY)**: The value read or written. For GET requests, this is either the value read from the file or the DEFAULT_VALUE.

Adapters

This function block does not use adapters.

Functionality

The INI block acts as an interface between the IEC 61499 application and a persistent `settings.ini` file on the file system. Upon a `INIT` event with `QI=TRUE`, the block is configured for the specified `SECTION` and `KEY`. Optionally, a value (`VALUE`) can be written immediately, or a default (`DEFAULT_VALUE`) can be set.

A `SET` event writes the current value of the `VALUE` input under the configured key to the file. A `GET` event reads the value from the file. If the key does not exist, `DEFAULT_VALUE` is returned instead. Each operation (`INIT`, `SET`, `GET`) is acknowledged by a corresponding output event (`INITO`, `SETO`, `GETO`), with `QO`, `STATUS`, and `VALUEO` indicating the result status.

…

Technical Features

- **Type Flexibility**: The data inputs/outputs `VALUE`, `DEFAULT_VALUE`, and `VALUEO` use the generic data type `ANY_ELEMENTARY`. This allows the storage of various elementary data types (e.g., numbers, Boolean values, strings) using the same function block.

- **Persistence**: The data is stored in text-based INI files (`settings.ini` for writable values, `settingsReadOnly.ini` for read-only factory settings).
- **Write Protection & Factory Defaults**: Keys in `settingsReadOnly.ini` override values in `settings.ini` and reject write attempts (`SET`) (`STATUS = "Key is read-only"`). For details, see Read-Only Settings (settingsReadOnly.ini) .
- **Error Handling**: The output of `STATUS` provides textual feedback on the success or failure of the operation, facilitating debugging.
- **Package**: The block is included in the package `eclipse4diac::storage`.

State Overview

The block does not have an explicit, complex state machine in the user-accessible sense. Its behavior is event-driven: After initialization (`INIT`), it waits for requests (`SET` or `GET`), processes them, and then returns to a waiting state. The internal state (configured section/key) is retained after `INIT`.

Application Scenarios

- **Configuration Management**: Loading device IP addresses, communication ports, or network settings when an application starts.
- **Parameterization**: Saving plant- or product-specific parameters (e.g., setpoints, times) that should be retained between restarts.
- **Operational Data Acquisition**: Persistently saving simple operating states or counter readings.
- **User Settings**: Managing language settings or other user preferences.
-

⚖️ Comparison with similar building blocks

- **`E_SR` (SR flip-flop) / `E_RS` (RS flip-flop)**: These blocks store a binary state (`BOOL`) in memory only during runtime. The INI block stores arbitrary data types permanently in non-volatile memory.
- **`E_DEMUX` / `E_MUX`**: These are used for event and data distribution, not persistent storage.
- **`FB_RETAIN` (from `eclipse4diac::core`)**: Stores data persistently, but typically uses device-specific, non-directly accessible retain memory. The INI block uses a standardized, readable text file.
- **Database or File I/O Blocks**: More general blocks for file access offer greater flexibility but are also more complex to use. The INI block is a specialized, simple solution for the common use case of key-value storage.

Conclusion

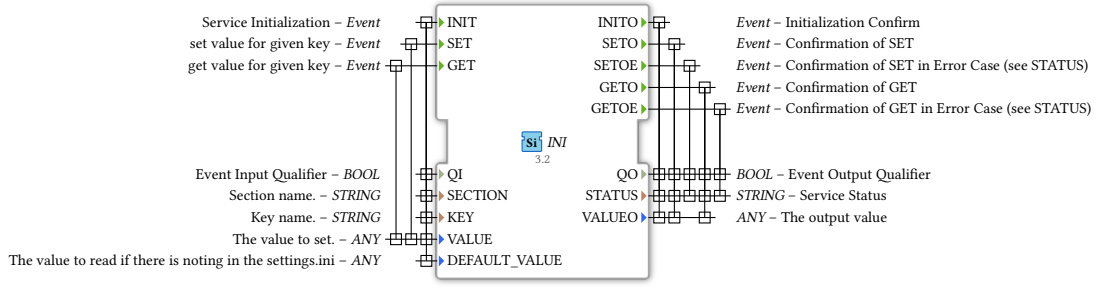
The INI function block is a practical and straightforward tool for persistently storing configuration and parameter data in 4diac FORTE applications. Its strengths lie in its ease of use via sections and keys, the flexibility provided by the `ANY_ELEMENTARY` type, and the human-readability of the stored INI file. It is an excellent choice for applications that require a simple, persistent key-value database.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter module on ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

INI



• – ○ • – ○ • – ○ •

Introduction

The **INI** function block is used for the secure reading and writing of configuration data in a `settings.ini` file. It allows access to values via a section and key name and provides both initialization and read/write operations as separate events. Its use of generic data types (`ANY`) makes it flexible.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Initializes the function block and opens the INI file
SET	Event	Writes the value of `VALUE` for the current key
GET	Event	Reads the value of the current key

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirmation
SETO	Event	Write Confirmation
SETOE	Event	Confirmation of SET in error case (see `STATUS`)
GETO	Event	Read Confirmation
GETOE	Event	Confirmation of GET in error case (see `STATUS`)

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier (controls execution)
SECTION	STRING	Section Name in the INI File
KEY	STRING	Key name within the section
VALUE	ANY	Value to be set (for SET)
DEFAULT_VALUE	ANY	Default value if key does not exist (for GET)

Data Outputs

Name	Type	Comment
QO	BOOL	Event output qualifier (indicates success/failure)
STATUS	STRING	Status information (e.g., error message)
VALUEO	ANY	Value retrieved (for GET) or confirmed set value

Adapters

No adapters defined.

Functionality

The function block operates on a file `settings.ini`, which contains a simple key-value structure in sections (`[Section]`). Its use involves three steps:

1. **INIT** – The INIT event passes the parameters `SECTION` and `KEY`. The function block opens or loads the INI file and prepares the read/write operations. Successful initialization is signaled by the INITO event.
2. **SET** – After initialization, a new value can be written to the previously defined key using SET. The passed value `VALUE` is stored in `settings.ini`, provided the key is not write-protected. If the parameter

is read-only and located in `settingsReadOnly.ini`, the write attempt is rejected, triggering the error event **`SETOE`**, and `STATUS` reports `"Key is read-only"`.

3. **GET** – The GET function reads the current value of the key. For read-only parameters, the value from `settingsReadOnly.ini` is always returned. If a normal key does not exist in `settings.ini`, the default value specified via `DEFAULT_VALUE` is returned. The read value appears at output `VALUEO`.

The execution of each event is acknowledged by the respective output (INITO, SETO/SETOE, GETO). The function block `QO` indicates whether the operation was successful, and `STATUS` provides textual feedback.

Technical Features

- **Generic Data Types** – `VALUE` and `DEFAULT_VALUE` are declared as `ANY`. The function block can therefore work with various data types (e.g., BOOL, INT, REAL, STRING), provided the runtime environment supports them.
- **Write Protection (Factory Default)** – In addition to `settings.ini`, the system also supports the write-protected default file `settingsReadOnly.ini`. Parameters from this file override modifiable values and reject write attempts using `STATUS = "Key is read-only"`. See also Read-only settings (`settingsReadOnly.ini`).
- **Event-based control** – All actions are triggered by events. The function block can therefore be fully integrated into an event-driven system (e.g., IEC 61499).
- **Error handling** – The output qualifier `QO` and the `STATUS` string enable easy error diagnosis (e.g., for read-only keys, missing file, or invalid section).

State overview

The function block goes through the following main states:

- **Idle** – Waiting for INIT.
- **Initialized** – After successful INIT; ready for SET or GET.
- **Error** – In case of failed initialization or operation; only a new INIT can exit this state.

State transitions are controlled by the INIT, SET, and GET events and their corresponding acknowledgment outputs.

Application Scenarios

- **Machine Configuration** – Saving and retrieving parameters such as speed, temperature limits, or operating modes in a central INI file.
- **Plant Initialization** – Loading settings once when the controller starts via the INIT event.
- **Online Parameterization** – Dynamically changing values during operation using SET and immediately retrieving them with GET.

Comparison with Similar Function Blocks

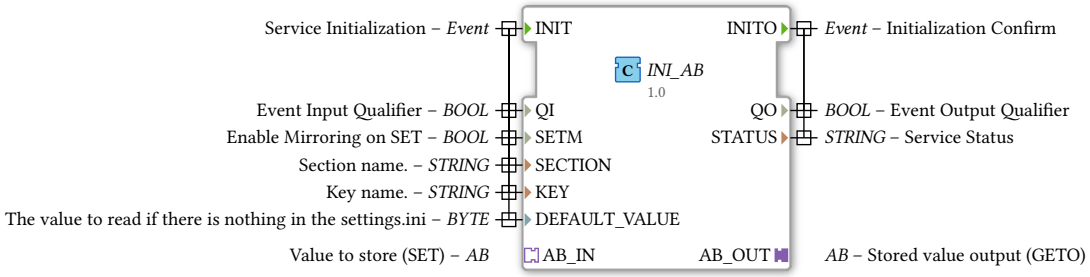
Function Block	Data Source	Type Support	Event Interface
INI	settings.ini	ANY (generic)	INIT, SET, GET
FILE_READ	Text File	STRING	REQ, CNF
PARAM_DB	Database	Type-specific	Complex

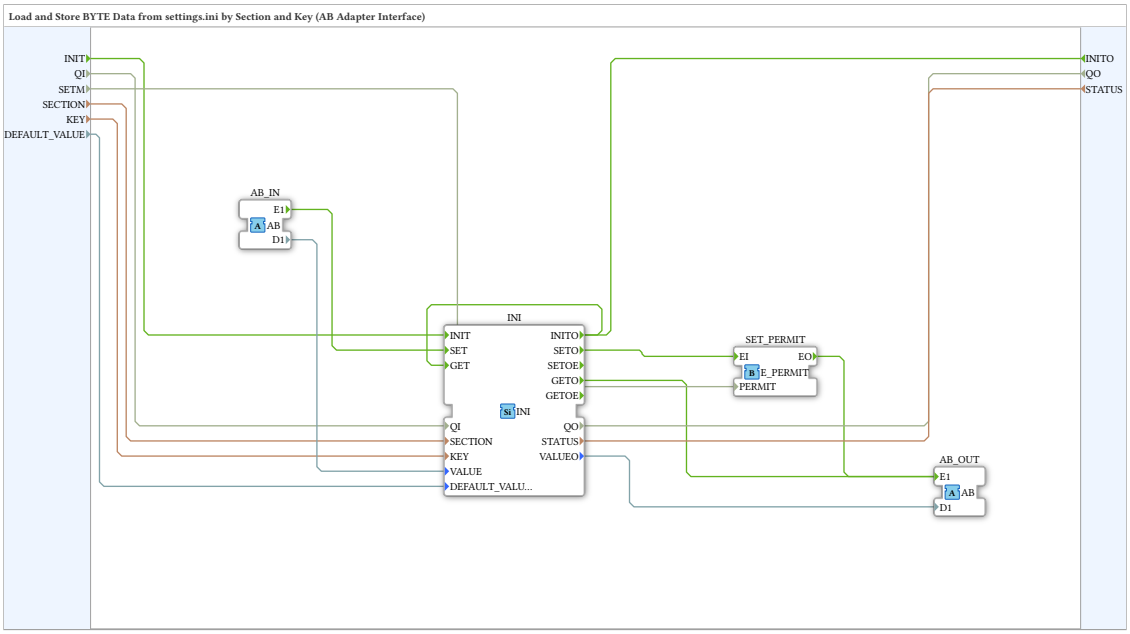
The INI function block stands out due to its simplicity and direct connection to INI files, without the need for additional drivers or databases. Other function blocks either offer more flexibility (database) or are limited to simple text input/output.

Conclusion

The INI function block provides a compact and reliable solution for the persistent storage of configuration data in IEC 61499 systems. Its generic data type definition and clear event handling make it versatile. Thanks to its integrated error handling, it is suitable for both development and production environments.

INI_AB





• – ○ • – ○ • – ○ •

Introduction

`INI_AB` loads and stores a `BYTE` value in the `settings.ini` file, addressed by section (`SECTION`) and key (`KEY`). It is the `settings.ini` counterpart to NVS_AB and uses the same two-adapter architecture with unidirectional `AB` adapters.

Interface Structure

Event Inputs

- **INIT**: initializes the block and reads the value stored under `SECTION/KEY` (or `DEFAULT_VALUE` if none exists yet). Carries `QI`, `SETM`, `SECTION`, `KEY`, `DEFAULT_VALUE`.

Event Outputs

- **INITO**: confirms initialization and the initial read, carries `QO` and `STATUS`.

Data Inputs

- **QI** (BOOL): qualifier for the `INIT` event.
- **SETM** (BOOL): enables mirroring (confirmation) on write (`SET`).
- **SECTION** (STRING): name of the section in `settings.ini`.
- **KEY** (STRING): name of the key within the section.
- **DEFAULT_VALUE** (BYTE): value returned when nothing is stored under `SECTION/KEY` yet.

Data Outputs

- **QO** (BOOL): success qualifier, carried with `INITO` and the internal adapter confirmations.
- **STATUS** (STRING): status message of the file access.

Adapters

- **AB_IN** (Socket, type AB): new value to store – an adapter event on AB_IN.E1 triggers a SET on the internal INI block, AB_IN.D1 carries the value to write.
- **AB_OUT** (Plug, type AB): outputs the most recently read value – both after the automatic initial read on INIT and after each confirmed write (AB_OUT.D1/AB_OUT.E1).

Functionality

INI_AB wraps an internal instance of eclipse4diac::storage::INI plus an E_PERMIT gate (SET_PERMIT). On INIT, the internal INI instance is initialized (INI.INIT) and a GET is triggered automatically, returning the value stored under SECTION/KEY (or DEFAULT_VALUE); the result is output via AB_OUT, and INITO confirms completion.

To write, the block connected to AB_IN fires an adapter event (AB_IN.E1) carrying the new value (AB_IN.D1), which triggers INI.SET. Whether the successful write is additionally reported back via AB_OUT is controlled by SETM: if SETM is set, SET_PERMIT releases the AB_OUT.E1 event after a successful SET ("mirroring").

Technical Details

- Pure composite FB with no ECC of its own – the actual file access logic lives in the internal INI instance; INI_AB only wires events and data between it and the two AB adapters.
- Two separate unidirectional adapters (AB_IN/AB_OUT) instead of a single bidirectional one – this lets the read and write paths connect to different peers, which INI_AB2 (a single shared AB2 socket) cannot do.
- Addressing via SECTION **and** KEY (rather than just KEY for the NVS_* blocks) matches the classic INI file format.

State Overview

The block is stateless in terms of its own ECC: on INIT it runs through initialization plus an initial read once, then reacts to every AB_IN adapter event with an INI write and, optionally, a confirmation via AB_OUT.

Application Scenarios

- Persisting individual BYTE configuration values (e.g. operating mode, channel selection) in a human-readable settings.ini
- Building a read/write pair where the read and write sides need to connect to different blocks
- Configuration files that also need to be viewable/editable outside the controller (e.g. with a text editor)

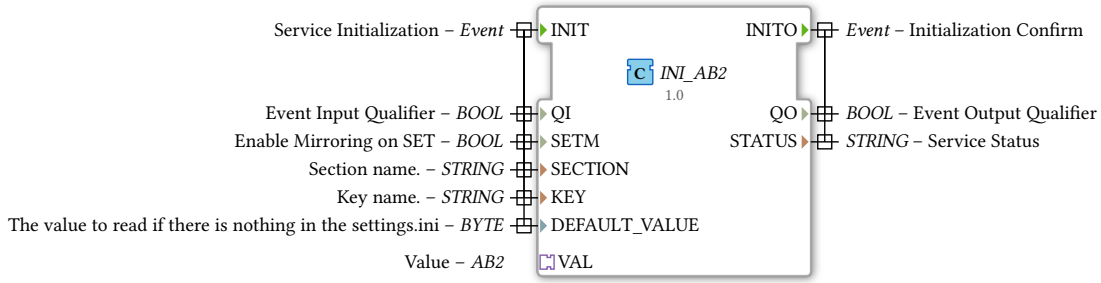
⚖️ Comparison with Similar Blocks

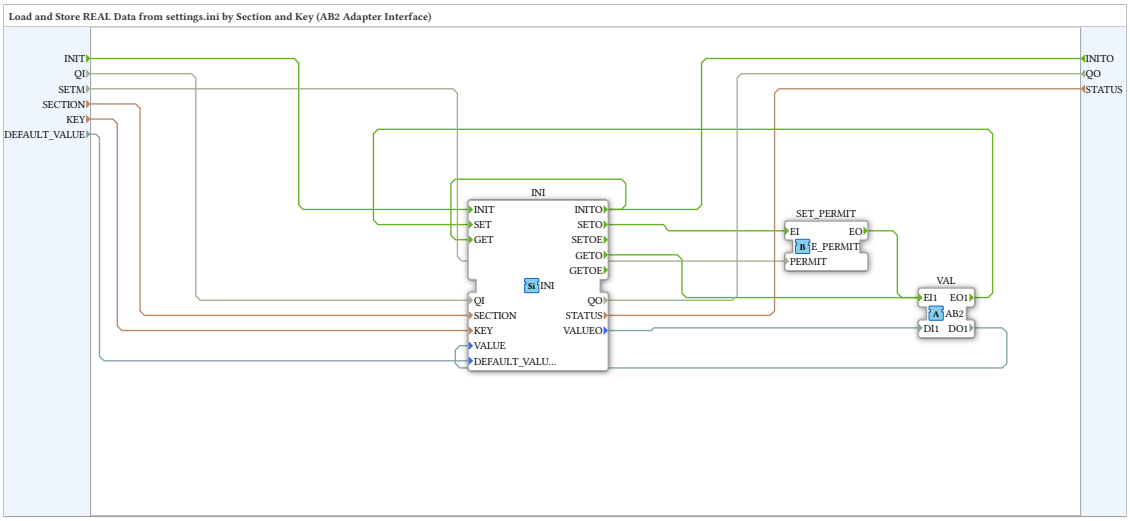
Compare with INI_AB2, which serves the same purpose through a single bidirectional AB2 adapter instead of two unidirectional AB adapters, and with NVS_AB, which provides the same interface for the ESP32 NVS flash storage instead of settings.ini.

Conclusion

INI_AB provides simple, unidirectionally wired access to a persistent BYTE value in settings.ini, and fits anywhere the read and write paths need to be kept separate.

INI_AB2





• – ○ • – ○ • – ○ •

Introduction

The function block `INI_AB2` is used to read and write BYTE values to or from a `settings.ini` file. Access is via a combination of section name and key name. Communication with external systems is via a bidirectional **AB2 adapter**, which allows values from the INI file to be passed to other function blocks or values from other function blocks to be received and stored in the file. The function block was developed according to the IEC 61499-2 standard.

Interface Structure

Event Inputs

Event	Type	Description
INIT	EInit	Initialization event; starts the read or write operation depending on the connected adapter logic. The data passed specifies the mirroring on SET (`SETM`), section, key, and a default value.

Event Outputs

Event	Type	Description
INITO	EInit	Initialization confirmation. Triggered after the read/write access is complete.

Data Inputs

Variable	Type	Description
QI	BOOL	Qualifier for the event input; controls its execution.
SETM	BOOL	Enable mirroring on SET.
SECTION	STRING	Name of the section in `settings.ini`.
KEY	STRING	Name of the key within the section.
DEFAULT_VALUE	BYTE	The value returned if the key does not exist in `settings.ini`.

Data Outputs

Variable	Type	Description
QO	BOOL	Output qualifier; indicates successful execution.

STATUS	STRING	Service status message (e.g., error messages or “OK”).

Adapters

Adapter	Type	Description
VAL	adapter::types::bidirectional::AB2	Bidirectional adapter for exchanging BYTE values. The incoming value (`DO1`) is written to the INI file; the outgoing value (`DI1`) is read from the INI file and passed on.

Functionality

The `INI_AB2` block internally contains an instance of the function block `eclipse4diac::storage::INI`. Communication with `settings.ini` occurs exclusively via this internal instance.

1. **Initialization (Read Operation):** An INIT event at the input sets the internal parameters (Section, Key, Default Value) and activates the internal INI instance (`INI.INIT`). After successful reading, the read value is sent to the connected module via the adapter output (`VAL.DI1`). Simultaneously, outputs `INITO`, `QO`, and `STATUS` are updated.
2. **Write Operation:** The connected module can provide a new value via the adapter input (`VAL.DO1`). As soon as the event `VAL.EO1` arrives, the value is saved in the INI instance (`INI.SET`). The confirmation (`INI.SETO`) is returned to the adapter via `VAL.EI1`.
3. **Write Operation:** The connected module can provide a new value via the adapter input (`VAL.DO1`). 3. **Cyclic Reading:** After each INIT operation, the internal GET command is automatically triggered, allowing the current value to be read from the file and output via the adapter.

The circuitry ensures that both read and write accesses can be handled bidirectionally via the adapter. The module thus implements persistent data storage with external control.

Technical Features

- **Bidirectional Adapter Connection:** The AB2 adapter allows both reading and writing to the INI file via a single connection.
- **Default Value:** If a specified key is not present in `settings.ini`, the value passed via `DEFAULT_VALUE` is used and output as the read value.
- **Use of `eclipse4diac::storage::INI`:** The function block encapsulates the actual INI access logic and provides robust error handling (status output).
- **IEC 61499-2 compliant:** The function block adheres to the standard for function blocks, which increases interoperability and reusability.

State Overview

The `INI_AB2` function block itself does not have explicit states (no ECC states), as the logic is fully implemented through the internal INI instance and the event/data flows. However, the interaction can be described as a simple sequential process:

1. **Idle State** – Waiting for INIT or data from the adapter.
2. **Initialize** – INIT is processed, the internal INI is started.
3. **Read** – The INI reads the value and outputs it via the adapter.
4. **Write** – Value from the adapter is saved to the INI file.
5. **Return** – Confirmation via INITO or adapter event.

Application Scenarios

- **Configuration Management:** Saving and retrieving machine parameters (e.g., setpoint temperatures, limit values) in a central `settings.ini` file.
- **Plant Parameterization:** Multiple control components access the same INI file via the AB2 adapter and can thus exchange parameters.
- **Data Exchange Between Heterogeneous Systems:** The standardized adapter enables integration into any 4diac architecture.

Comparison with Similar Function Blocks

- **INI_AB (Unidirectional):** Uses a unidirectional adapter that allows either read-only or write-only access. `INI_AB2` is more flexible due to its bidirectional interface.
- **INI_CFB (Composite FB):** A similar function block without an adapter, addressed directly via events and data. `INI_AB2` abstracts the connection and enables modular coupling.
- **eclipse4diac::storage::INI (direct):** The basic function block does not offer an adapter interface; `INI_AB2` encapsulates this and adds bidirectional communication.

Conclusion

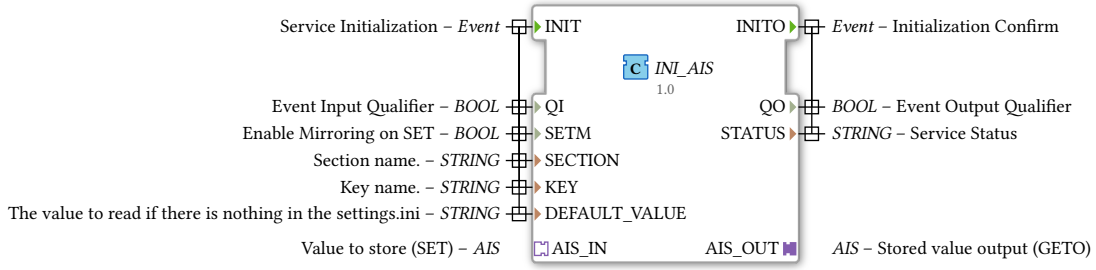
The `INI_AB2` function block is a practical extension for IEC 61499-based automation solutions. It simplifies access to INI configuration files through a standardized adapter interface and enables both read and write operations. Its ease of use and clear separation of logic and interface make it a useful tool for modular and reusable control applications.

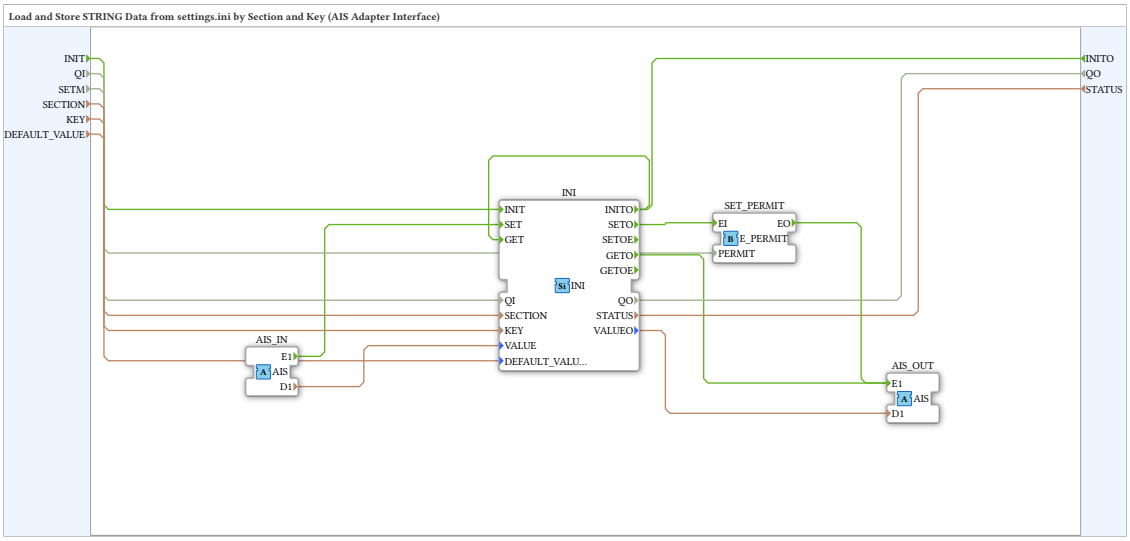
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

INI_AIS





• – ○ • – ○ • – ○ •

Introduction

The function block **INI_AIS** is used to load and save strings (STRING) from a `settings.ini` file. The desired value is accessed via a section name and a key. The interface is implemented using unidirectional AIS adapters, allowing the block to both read and write values.

Interface Structure

Event Inputs

Name	Type	Description
INIT	EInit	Service Initialization

Event Outputs

Name	Type	Description
INITO	EInit	Initialization Confirmation

Data Inputs

Name	Type	Description
QI	BOOL	Qualifier for the event input
SETM	BOOL	Enable Mirroring on SET: if TRUE, the confirmation of a SET (SETO) is additionally forwarded to the output adapter (mirrored)
SECTION	STRING	Name of the section in the configuration file
KEY	STRING	Name of the key in the specified section
DEFAULT_VALUE	STRING	Default value if the entry does not exist

Data Outputs

Name	Type	Description
QO	BOOL	Qualifier for the event output
STATUS	STRING	Status message (e.g., error text)

Adapter

Name	Type	Direction	Description
AIS_OUT	adapter::types::unidirectional::AIS	Plug	Output for the read value (GETO)
AIS_IN	adapter::types::unidirectional::AIS	Socket	Input for the value to be stored (SET)

Functionality

The function block internally contains an instance of the function block `INI` (from the library `eclipse4diac::storage`). The process is as follows:

1. **Initialization (INIT)**

- The data `QI`, `SETM`, `SECTION`, `KEY`, and `DEFAULT_VALUE` are forwarded to the internal INI block.
- The INIT event pulse starts the INI block.
- Upon completion, INI returns the event `INITO`, which is output externally as `INITO` of the INI_AIS. Simultaneously, the outputs `QO` and `STATUS` are also received.
- Immediately after initialization, the `GET` service is automatically triggered in the INI block to read the value from `settings.ini`.

1. **Reading a Value (via AIS_OUT)**

- The read value appears at the data output `VALUEO` of the INI block and is assigned to the data output `D1` of the adapter plug `AIS_OUT`.
- The event `GETO` of the INI block is transferred to the event input `E1` of `AIS_OUT`, allowing the receiving block to retrieve the value.

1. **Writing a Value (via AIS_IN)**

- An external function block can send a value (event `E1` with data `D1`) to the INI_AIS via the adapter socket `AIS_IN`.
- This event triggers the `SET` service of the internal INI function block. The incoming value is passed to the data input `VALUE` of INI.
- After successful writing, INI sends the event `SETO`, which is forwarded to the event output `E1` of `AIS_OUT`. The written value is simultaneously returned via `VALUEO`, so the writing block receives confirmation of the value.

Technical Features

- **Internal Networking:** This block utilizes the existing `INI` block and encapsulates its complex interface in a simpler, adapter-based form.
- **Unidirectional AIS Adapters:** Data exchange occurs via AIS interfaces, which enable loose coupling between the blocks. The `AIS_IN` socket is responsible for receiving write commands, and the `AIS_OUT` plug is responsible for sending read results.
- **Automatic Read Access:** After initialization, the stored value is immediately read and made available via the output – without a separate GET request.
- **Automatic Read Access: - Error Handling:** Status messages are communicated via the `STATUS` output (e.g., file not found, section/key missing).

State Overview

The function block has no explicitly programmed states. The internal INI block handles state management. From the user's perspective, the following operating phases occur:

- **Initialized (after INIT – INITO):** The function block is ready to read and write values. The value of the specified key has already been read once and is available at the `AIS_OUT` output.
- **Write Operation (via AIS_IN):** An external command triggers a write access. After completion, the new value can be retrieved via `AIS_OUT`.
- **Read Access:** Occurs implicitly after each INIT and is not explicitly controlled.

Application Scenarios

- **Persistent Configuration Data:** Saving parameters such as device addresses, limits, or operating modes in an INI file.
- **Plug-and-Play Modules:** Modules that read their configuration via the AIS adapter and overwrite it as needed – without hardwiring the data paths.
- **Initialization Values:** Reading default values at system startup, which can be customized via the `settings.ini` file.

Comparison with Similar Modules

Module	Description
INI	Direct access to INI files with separate GET and SET events.
INI_AIS	INI extends the INI function block with an adapter-based interface that simplifies the coupling of other function blocks and integrates the handling of read and write operations.

- Unlike the basic `INI` function block, INI_AIS provides a unified, event-driven interface for read and write components.
- The use of adapters enables flexible connectivity in larger automation networks without the need for individual data wiring.

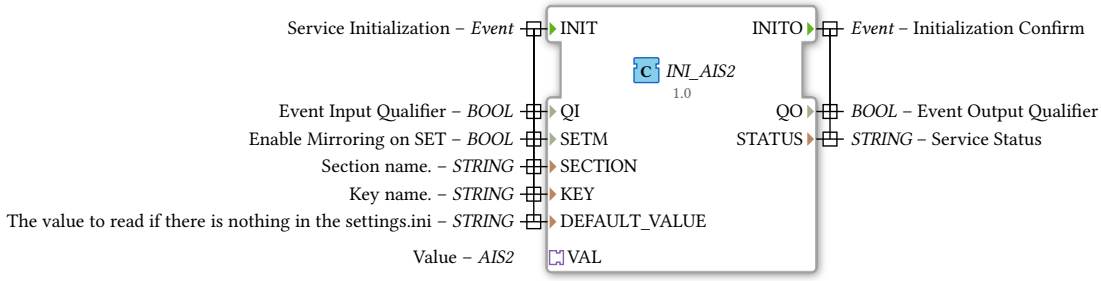
Conclusion

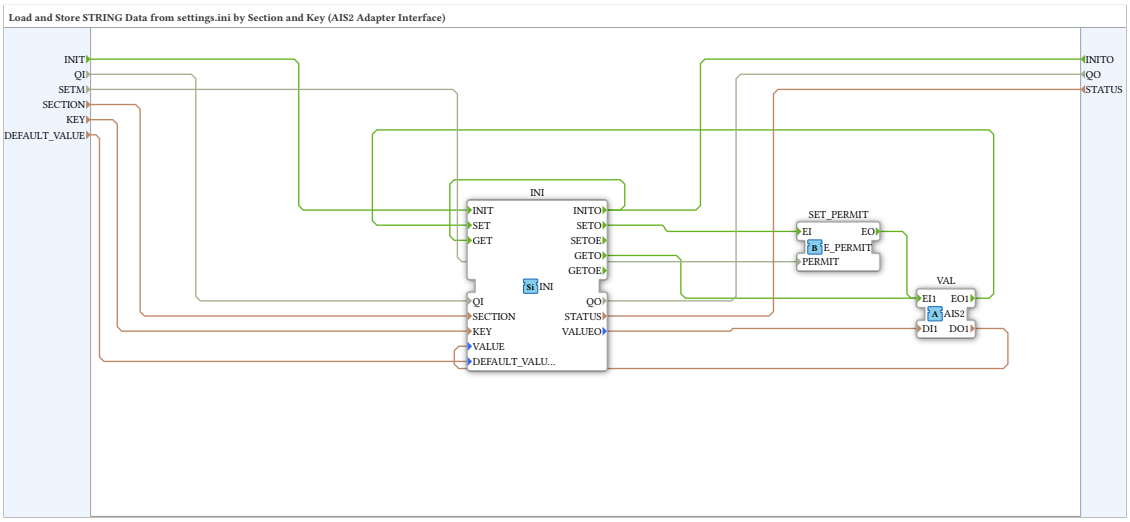
The **INI_AIS** function block is a practical extension of the basic `INI` function block.A module that significantly simplifies the use of configuration files in automation projects. By integrating read and write operations into a single adapter interface, system design becomes more modular and streamlined. This module is particularly suitable for scenarios where multiple components need to access the same data storage.

🌐 Related topic subpages on ms-muc-docs.de

``* 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

INI_AIS2





• – ○ • – ○ • – ○ •

Introduction

The INI_AIS2 function block is used to read and write STRING data from a settings.ini file via an AIS2 adapter. It combines the INI block with a bidirectional AIS2 adapter to read values from or write them back to a configuration file. Initialization is performed via the INIT event, where the section and key are specified. The read value is offered via the adapter, and a write operation is triggered via the adapter event.

Interface Structure

Event Inputs

Name	Description
INIT	Initialization event to start the read operation. Expects the parameters QI, SECTION, KEY, and DEFAULT_VALUE.

Event Outputs

Name	Description
INITO	Initialization confirmation. Sent after the read operation is complete. Status information is available via QO and STATUS.

Data Inputs

Name	Type	Description

QI	BOOL	Input qualifier for controlling processing.
SETM	BOOL	Controls confirmation/mirroring after a successful write. Regardless of SETM, VAL.EO1 always triggers INI.SET; however, the SETO confirmation reaches VAL.EI1 only when SETM=TRUE (suppressed when SETM=FALSE).
SECTION	STRING	Name of the section in the settings.ini file.
KEY	STRING	Name of the key within the section.
DEFAULT_VALUE	STRING	Default value read if the key is not present in the file. Default: ''.

Data Outputs

Name	Type	Description
QO	BOOL	Output qualifier, indicates successful processing.
STATUS	STRING	Service status message (e.g., error messages).

Adapters

Name	Type	Description
VAL	adapter::types::bidirectional::AIS2	Bidirectional adapter for exchanging values. This adapter provides the read value as an output (DI1) and receives a write value as an input (DO1).

Functionality

The INI_AIS2 function block contains an internal INI function block (eclipse4diac::storage::INI) that performs the actual file operation. The network connections implement the following processes:

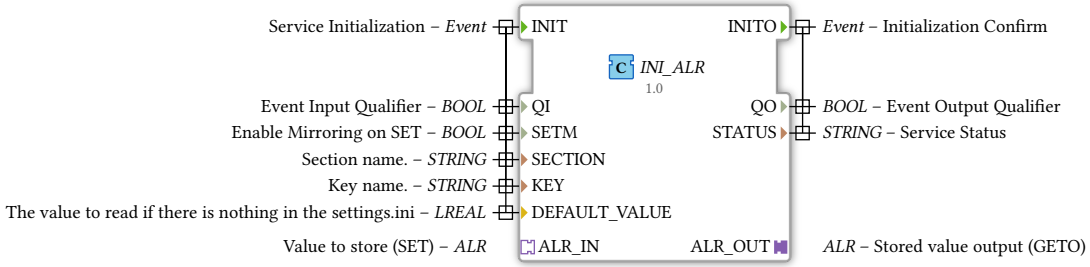
- Upon arrival of INIT, QI, SECTION, KEY, and DEFAULT_VALUE are forwarded to the INI function block.
- The INI function block performs a read operation and passes the result (via VALUEO) to the adapter output (VAL.DI1).
- Simultaneously, after a successful read, the GET event is triggered, which activates the adapter input (VAL.EI1) to transmit the value.
- A write operation is initiated as soon as the adapter sends an EO1 event. This event triggers the SET event on the INI function block, setting the value received via VAL.DO1 as the new VALUE.
- The INI block confirms the write operation with SETO. If SETM=TRUE, this confirmation is routed back to the adapter (VAL.EI1), whereas SETM=FALSE suppresses this mirror.
- The QO and STATUS outputs are directly taken from the INI block.

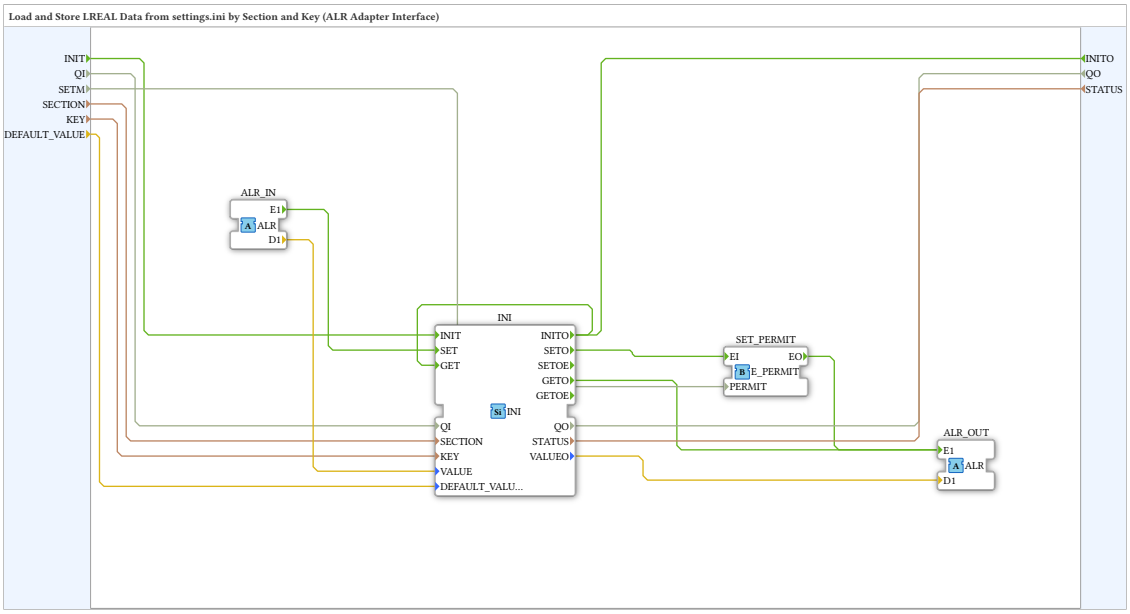
Thus, read and write access to the settings.ini file can be controlled via the AIS2 adapter.

Technical Features

- The block uses a bidirectional AIS2 adapter that can both send and receive values.
- A STRING value is used as the default value.

INI_ALR





• – ○ • – ○ • – ○ •

Introduction

The function block **INI_ALR** is used to read and write floating-point values of type LREAL to and from a `settings.ini` file. It accesses a configuration file that is parameterized via a section name and a key. The block encapsulates the use of the internal `INI` function block and additionally provides a standardized **ALR** adapter interface (unidirectional) through which values can be both set and read.

Interface Structure

Event Inputs

Event	Type	Comment	With Variables
INIT	EInit	Initializes the function block and triggers a read operation	QI, SETM, SECTION, KEY, DEFAULT_VALUE

Event Outputs

Event	Type	Comment	With variables
INITO	EInit	Confirms initialization	QO, STATUS

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier (Allow)
SETM	BOOL	Enable mirroring on SET
SECTION	STRING	Section name in the INI file
KEY	STRING	Key name within the section
DEFAULT_VALUE	LREAL	Fallback value if the key is missing in the INI file

Data Outputs

Name	Type	Comment

QO	BOOL	Event Output Qualifier
STATUS	STRING	Status message (e.g., error text)

Adapters

Adapter	Type	Comment
ALR_IN	adapter::types::unidirectional::ALR	Socket – Value to store (SET)
ALR_OUT	adapter::types::unidirectional::ALR	Plug – Read value (GET)/Confirmation

Functionality

1. **Initialization and First Read**

An event at the input `INIT` triggers the initialization of the internal `INI` block. The parameters `QI`, `SETM`, `SECTION`, `KEY`, and `DEFAULT_VALUE` are forwarded to the `INI` block. After its acknowledgment (`INITO`), the GET operation of the `INI` block is automatically triggered. The read value appears at the data output of the adapter `ALR_OUT.D1` and is simultaneously signaled via the event `ALR_OUT.E1`. The outputs `QO` and `STATUS` of the `INI` block are directly passed through to the corresponding outputs of the `INI_ALR` block.

... 2. **Write via ALR-IN**

A set event at the adapter socket `ALR_IN.E1` is forwarded to the SET input of the internal `INI` module. The value to be saved is located at `ALR_IN.D1` and is set via `INI.VALUE`. After successful writing, the `INI` module acknowledges with `SETO`, which in turn triggers `ALR_OUT.E1`.

1. **Reading via ALR-IN** (implicit)

The currently read value is always available at output `ALR_OUT` via the initial GET sequence and the forwarding of `GETO`. A new read can be triggered by another `INIT` pulse.

Technical Features

- **Reuse** of the standardized `INI` block (`eclipse4diac::storage::INI`) for file access.
- **ALR adapter interface** – allows unidirectional coupling with other blocks that use the same adapter type. This simplifies integration into modular control applications.
- **Asynchronous processing**: Read and write operations are event-driven and performed asynchronously. Acknowledgement is provided via the corresponding event outputs.
- **Compliant with IEC 61499-2 standard** – all inputs and outputs are defined with standard types.
- **Error handling** is provided via the STATUS output (e.g., for missing files or corrupted data).

State Overview

The `INI_ALR` itself does not have its own state machines. State management is fully implemented by the internal `INI` block. Typical phases are:

- **Idle** – Waiting for an INIT event.
- **Initialize** – Processing of INIT, forwarding data to `INI`.
- **Ready** – A read result is available after successful INIT acknowledgment.
- **Write Operation** – triggered by `ALR_IN.E1`.
- **Error State** – in case of faulty INI accesses, STATUS is set accordingly.

Application Scenarios

- **Machine Configuration** – Loading parameters (e.g., maximum speed, limits) from a central INI configuration file.
- **Calibration Values** – Storing and reading calibration data in automation technology.
- **Interface to ALR-based Components** – If a function block provides or expects an LREAL value via an ALR adapter, `INI_ALR` can be used as a link for persistent storage.

Comparison with Similar Function Blocks

Function Block	Description	Advantage of INI_ALR
`INI`	Direct access to INI files (without ALR)	No adapter interface, manual coupling required

INI_ALR	Encapsulates INI and provides an ALR adapter interface	Easy integration into ALR-based architectures

The INI_ALR combines the proven INI file functionality with a standardized adapter interface, thereby reducing the effort required for integration into a service-oriented control system.

Conclusion

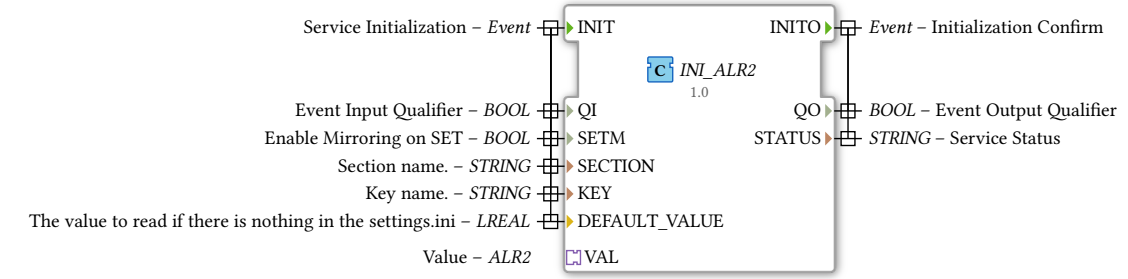
The INI_ALR is a practical function block that extends access to settings.ini for LREAL data with a flexible adapter interface. It is particularly suitable for applications that require consistent ALR-based communication while also needing simple, file-based persistence. By encapsulating the complex INIThe use of modular components and the automatic chaining of event and data flows reduces development effort and increases reusability.

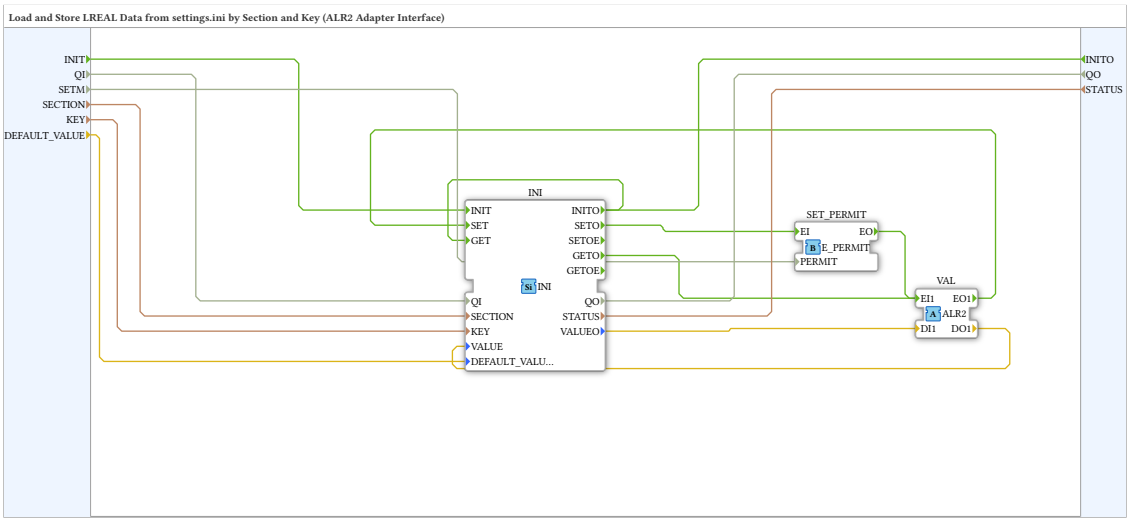
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

INI_ALR2





• – ○ • – ○ • – ○ •

Introduction

The INI_ALR2 function block is used to read and write LREAL data from a settings.ini file via an ALR2 adapter. It combines the INI block with a bidirectional ALR2 adapter to read values from or write them back to a configuration file. Initialization is performed via the INIT event, where the section and key are specified. The read value is offered via the adapter, and a write operation is triggered via the adapter event.

Interface Structure

Event Inputs

Name	Description
INIT	Initialization event to start the read operation. Expects the parameters QI, SECTION, KEY, and DEFAULT_VALUE.

Event Outputs

Name	Description
INITO	Initialization confirmation. Sent after the read operation is complete. Status information is available via QO and STATUS.

Data Inputs

Name	Type	Description
QI	BOOL	Input qualifier for controlling processing.
SETM	BOOL	Controls confirmation/mirroring after a successful write. Regardless of SETM, VAL.EO1 always triggers INI.SET; however, the SETO confirmation reaches VAL.EI1 only when SETM=TRUE (suppressed when SETM=FALSE).
SECTION	STRING	Name of the section in the settings.ini file.
KEY	STRING	Name of the key within the section.
DEFAULT_VALUE	LREAL	Default value read if the key is not present in the file. Default: 0.0.

Data Outputs

Name	Type	Description
QO	BOOL	Output qualifier, indicates successful processing.
STATUS	STRING	Service status message (e.g., error messages).

Adapters

Name	Type	Description
VAL	adapter::types::bidirectional::ALR2	Bidirectional adapter for exchanging values. This adapter provides the read value as an output (DI1) and receives a write value as an input (DO1).

Functionality

The INI_ALR2 function block contains an internal INI function block (eclipse4diac::storage::INI) that performs the actual file operation. The network connections implement the following processes:

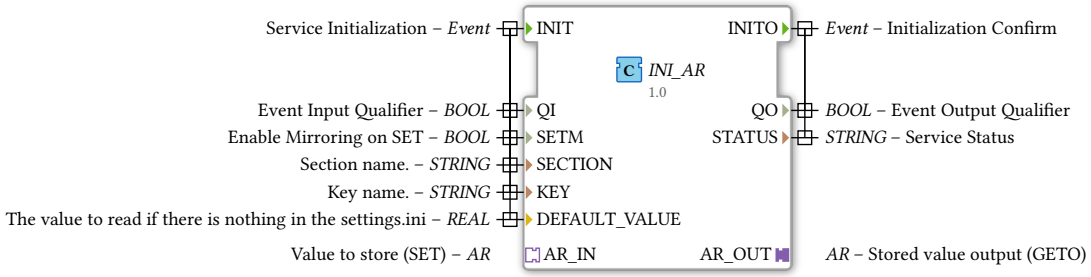
- Upon arrival of INIT, QI, SECTION, KEY, and DEFAULT_VALUE are forwarded to the INI function block.
- The INI function block performs a read operation and passes the result (via VALUEO) to the adapter output (VAL.DI1).
- Simultaneously, after a successful read, the GET event is triggered, which activates the adapter input (VAL.EI1) to transmit the value.
- A write operation is initiated as soon as the adapter sends an EO1 event. This event triggers the SET event on the INI function block, setting the value received via VAL.DO1 as the new VALUE.
- The INI block confirms the write operation with SETO. If SETM=TRUE, this confirmation is routed back to the adapter (VAL.EI1), whereas SETM=FALSE suppresses this mirror.
- The QO and STATUS outputs are directly taken from the INI block.

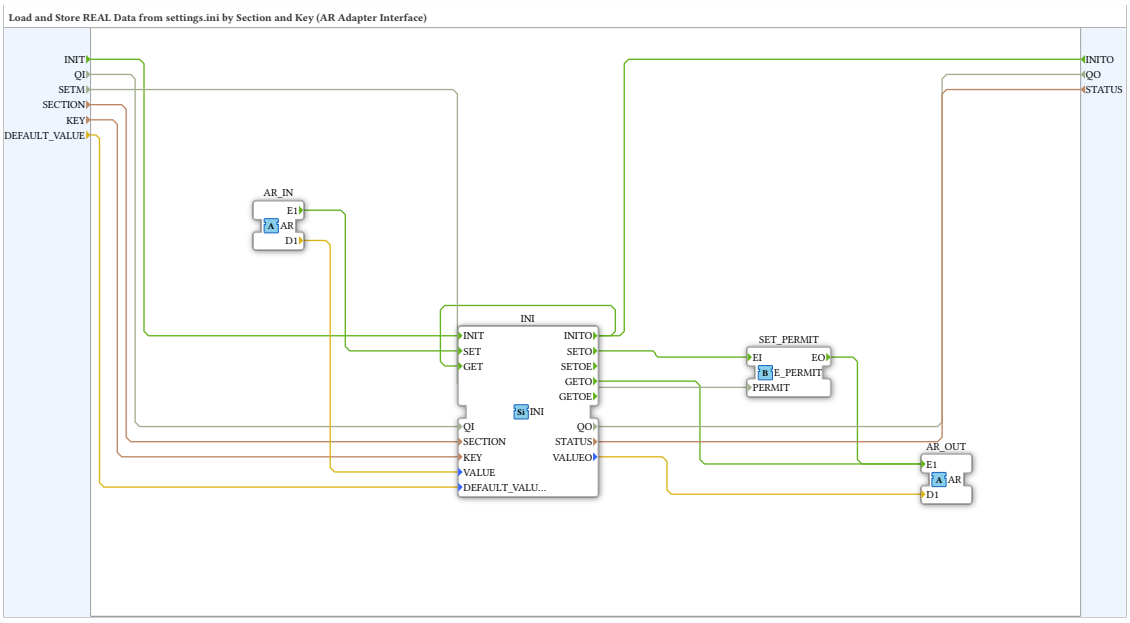
Thus, read and write access to the settings.ini file can be controlled via the ALR2 adapter.

Technical Features

- The block uses a bidirectional ALR2 adapter that can both send and receive values.
- A LREAL value is used as the default value.

INI_AR





• – ○ • – ○ • – ○ •

Introduction

The INI_AR function block is used to load and save real data from a settings.ini file. It accesses the configuration file via a section and key name and provides the read or saved values via a unidirectional adapter. This allows for flexible integration into modular storage and control architectures.

Interface Structure

Event Inputs

- `INIT` (EInit): Service initialization. Triggers the loading of the current value and processes the associated data inputs.

Event Outputs

- `INITO` (EInit): Initialization confirmation. Sent after a successful read or write operation.

Data Inputs

- `QI` (BOOL): Qualifier for the event input.
- `SETM` (BOOL): Enable mirroring on SET.
- `SECTION` (STRING): Name of the section in the INI file.
- `KEY` (STRING): Name of the key within the section.

- `DEFAULT_VALUE` (REAL): Value to be used if no entry is found in the settings.ini file.

Data Outputs

- `QO` (BOOL): Qualifier for the event output.
- `STATUS` (STRING): Status message of the service (e.g., success, error).

Adapter

- **AR_OUT** (Plug, type `adapter::types::unidirectional::AR`): Adapter output through which the read value (GETO) is sent to other function blocks.
- **AR_IN** (Socket, type `adapter::types::unidirectional::AR`): Adapter input through which a value to be stored (SET) can be received.

Functionality

After an INIT pulse, the internal function block `INI` (of type `eclipse4diac::storage::INI`) is triggered. This reads the current value from the specified section and the key from the settings.ini file. If no entry exists, `DEFAULT_VALUE` is used. After the read operation is complete, the GET event is triggered, and the read value is passed on via the adapter output `AR_OUT` (event GETO, data VALUEO).

Simultaneously, a new value (event E1, data D1) can be received via the adapter input `AR_IN`. This triggers the SET event in the INI block, which writes the value to the INI file. After writing, an acknowledgment is sent via `SETO`, which is also forwarded to the adapter output. The GETO and SET events are combined into the same output event `AR_OUT.E1`. The qualified outputs `QO` and `STATUS` are taken directly from the internal INI block.

Technical Features

- The function block uses the internal INI function block from the library `eclipse4diac::storage`.
- The adapters `AR_IN` and `AR_OUT` must be of type `adapter::types::unidirectional::AR`.
- The function block is licensed under the Eclipse Public License 2.0 (EPL-2.0).
- Author: Franz Höpfinger, Version 1.0, Date: April 24, 2026.

State Overview

The function block does not have an explicit state machine. The logic is controlled entirely via the event chaining of the internal INI function block:

- The value is read (GET) after `INIT`.
- Each `SET` pulse via `AR_IN` writes a new value to the INI file.

Application Scenarios

- Persistent storage of configuration parameters (REAL) in an INI file for automation projects.
- Exchange of runtime parameters between different function blocks via the adapter interface.
- Initializing plant parameters from a central configuration file.

Comparison with Similar Blocks

- **Standard INI Block** (`INI`): Provides direct read/write access to INI files via data inputs and outputs without an adapter.
- **INI_AR** extends this with the adapter interface, allowing values to be passed modularly to other blocks via events and data without having to encapsulate all the logic in a single block.
- Analogous building blocks exist for other data formats (JSON, XML), but these use different libraries.

Conclusion

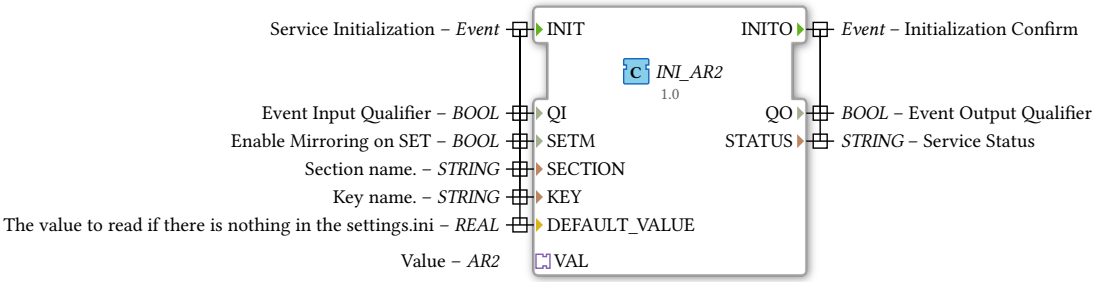
The INI_AR function block encapsulates access to an INI file for real values and simultaneously provides a standardized adapter interface for integration into distributed control networks. It simplifies saving and loading configuration data and allows for a clear separation between memory logic and data flows.

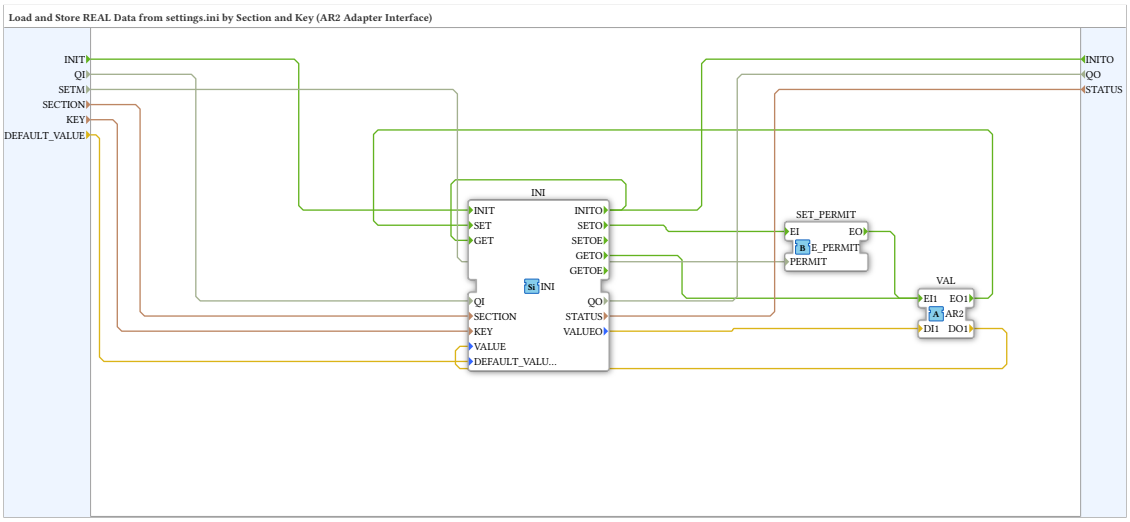
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

INI_AR2





• – ○ • – ○ • – ○ •

Introduction

The INI_AR2 function block is used to read and write real values to and from a settings.ini file using a section and key name. It encapsulates an INI function block and provides a bidirectional AR2 adapter interface. Upon initialization, the value for the specified section and key name is read and output via the adapter. Simultaneously, new values can be received via the adapter and written back to the file.

Interface Structure

Event Inputs

- **INIT** (EInit): Service initialization. Triggers the loading of the value from settings.ini and initializes the function block.

Event Outputs

- **INITO** (EInit): Initialization confirmation. Sent after successful processing of the INIT event.

Data Inputs

- **QI** (BOOL): Input qualifier. Controls execution (e.g., activation/deactivation).
- **SETM** (BOOL): Enable Mirroring on SET: if TRUE, the confirmation of a SET (SETO) is additionally forwarded to the output adapter (mirrored).
- **SECTION** (STRING): Section name in the settings.ini file (e.g., “[MySection]”).
- **KEY** (STRING): Key name in the section (e.g., “MyKey”).
- **DEFAULT_VALUE** (REAL): Default value returned if the key is not present in the file.

Data Outputs

- **QO** (BOOL): Output qualifier. Indicates the processing status.
- **STATUS** (STRING): Service status message (e.g., error message or “OK”).

Adapters

- **VAL** (AR2): Bidirectional adapter of type `adapter::types::bidirectional::AR2`. This adapter provides the read values (output) and receives new values (input) for writing back to the INI file. The adapter has the events EO1 (output) and EI1 (input) as well as data DO1 (output) and DI1 (input).

Functionality

The FB INI_AR2 is a composite function block that internally uses an INI function block (`eclipse4diac::storage::INI`). The process is as follows:

1. **Initialization**: Upon arrival of the INIT event, the parameters QI, SETM, SECTION, KEY, and DEFAULT_VALUE are passed to the internal INI block.
2. **Reading**: After INIT, the INI block automatically performs a GET operation (controlled by the event connection from INI.INITO to INI.GET). It reads the value from the settings.ini file for the specified section and key. If no entry exists, the DEFAULT_VALUE is used.
3. **Output via Adapter**: The read value is passed to the adapter output via the data connection INI.VALUEO -> VAL.DI1 and the event INI.GETO -> VAL.EI1.
4. **Write via Adapter**: When a new value arrives at VAL.DO1 via the adapter (triggered by VAL.EO1), the event VAL.EO1 -> INI.SET is sent. The INI block then writes the new value back to the file. After writing, INI.SETO -> VAL.EI1 is sent, which returns the acknowledgment to the adapter.
5. **Completion**: After the initial read operation, the INITO event is output, along with the status information QO and STATUS. After write operations via the adapter, no INITO is sent, only an acknowledgment via VAL.EI1.

Technical Features

- The function block uses the Eclipse 4diac infrastructure and imports types from `eclipse4diac::storage` and `adapter::types::bidirectional::AR2`.
- The AR2 adapter enables a clear separation between read and write accesses via a single adapter socket.
- By default, the file `settings.ini` is expected in the application's working directory. The exact path can be adjusted via the INI block configuration (not directly visible in INI_AR2).
- The function block is designed for REAL data; corresponding variants would need to be created for other data types.
- Error handling is handled via the STATUS output and QO.
- The INITO event is only triggered after the initial read operation, not after write operations via the adapter. Write operations are only acknowledged via the adapter.
-

State Overview

Since this is a composite function block, there is no explicit internal state machine. The process is determined by the event network:

- Receive INIT ⟶ Execute INI.INIT.
- After INI.INITO: Execute INI.GET automatically.
- After INI.GETO: Trigger VAL.EI1 (value output) and send INITO.
- Upon receiving VAL.EO1 (new value from an external source): Execute INI.SET, then INI.SETO ⟶ VAL.EI1 (acknowledgment to the adapter, no INITO).

Application Scenarios

- **Parameter Management**: Saving and loading settings (e.g., PID controller parameters, setpoints) from a configuration file.
- **Initial Configuration**: When an application starts, predefined values are loaded from the settings.ini file.
- **Online Parameterization**: New values can be written at runtime via the AR2 adapter using an HMI or a control system. The block immediately saves and confirms these values in the file.
- **Redundancy**: If no file exists, the DEFAULT_VALUE is used, allowing the application to start even without configuration.

Comparison with Similar Blocks

- **INI (eclipse4diac::storage::INI)**: The INI_AR2 extends the basic INI block with an adapter interface, making it easier to integrate into a higher-level structure. The standard INI block requires separate event connections for reading and writing; INI_AR2 simplifies this with its bidirectional adapter.

- Other storage blocks (e.g., for databases or OPC UA): INI_AR2 is limited to simple INI files but is very lightweight and has no additional runtime dependencies.

Conclusion

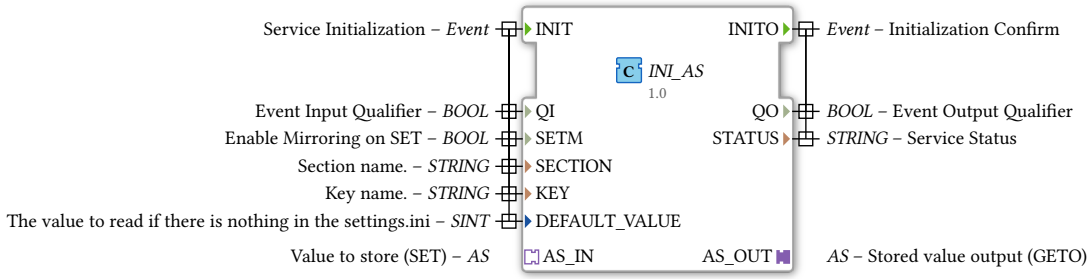
The INI_AR2 function block offers a compact and easy-to-integrate solution for persistently storing and loading REAL configuration values. Using an AR2 adapter standardizes the connection to other components. This block is particularly suitable for embedded controllers with low data storage requirements, where a single INI file is sufficient.

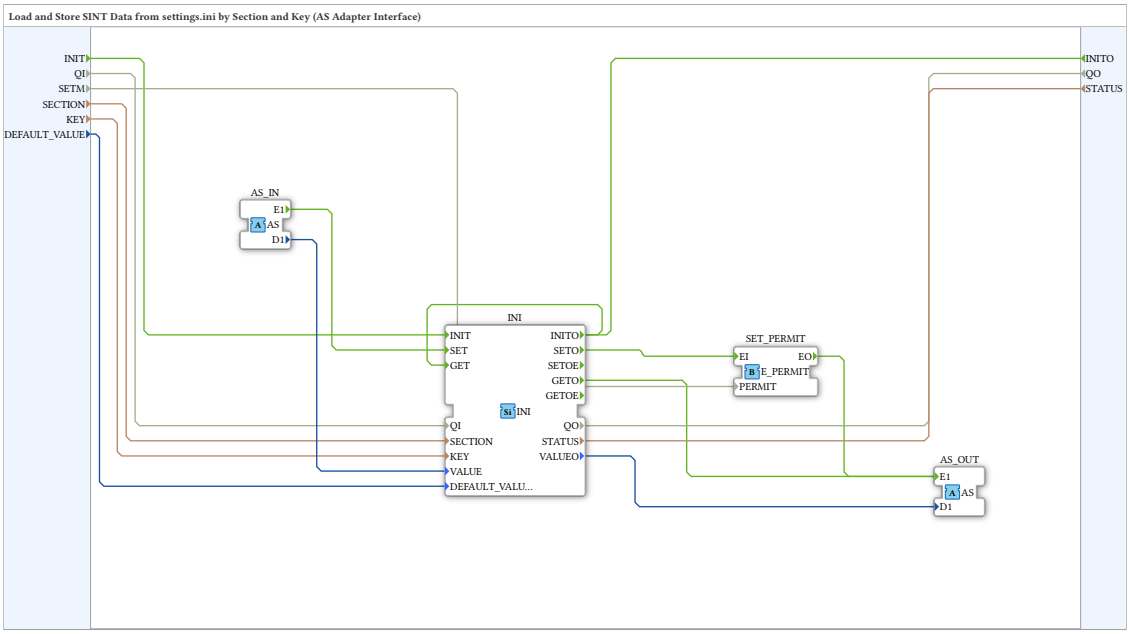
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

INI_AS





• — ○ • — ○ • — ○ •

Introduction

The function block **INI_AS** is used to load and save SINT data from a `settings.ini` file. It accesses a configuration value via a section name (`SECTION`) and a key (`KEY`). The value can be both read and written via the adapter interface `AS`. The block encapsulates the internal `INI` function block and extends it with a standardized adapter interface.

Interface Structure

Event Inputs

| Event | Type | With Vars | Description | | Event | Type | With Vars | Description | |-----|----|-----|-------| | `INIT` | EInit | QI, SETM, SECTION, KEY, DEFAULT_VALUE | Service Initialization: Reads the value from the INI file and outputs it via the adapter. |

Event Outputs

Event	Type	With Vars	Description
`INITO`	EInit	QO, STATUS	Confirmation of initialization.

Data Inputs

Name	Data Type	Description
`QI`	BOOL	Input qualifier (e.g., enable).
`SETM`	BOOL	Enable mirroring on SET
`SECTION`	STRING	Name of the section in the INI file.
`KEY`	STRING	Name of the key within the section.
`DEFAULT_VALUE`	SINT	Value returned if the key is not present in the INI file.

Data Outputs

| Name | Data Type | Description | | |-----|-----|-------| | `QO` | BOOL | Output qualifier (indicates successful operation). | | `STATUS` | STRING | Status message (e.g., error messages). |

Adapter

Name	Type	Direction	Description
`AS_IN`	`adapter::types::unidirectional::AS`	Input (Socket)	Receives a new value to be stored (SET operation).
`AS_OUT`	`adapter::types::unidirectional::AS`	Output (Plug)	Outputs the currently read or stored value (GET operation).

Functionality

1. **Initialization and First Read (INIT Event)**

The parameters `SECTION`, `KEY`, and `DEFAULT_VALUE` are passed via the `INI` event. The internal `INI` block is started, reads the value from `settings.ini`, and outputs it via the adapter output `AS_OUT`. Simultaneously, the `INITO` event is triggered, reporting the completion of the initialization.

Subsequently (possibly within the same cycle), a `GET` event is automatically executed on the `INI` block, so that the read value is immediately available at the adapter.

1. **Writing a Value (via the Adapter Input)**

A new value (event `E1` and data `D1`) can be sent to the block via the socket `AS_IN`. This event is forwarded to the `SET` input of the internal `INI` block, which saves the value to the INI file. After successful saving, the `SETO` event of the `INI` block is triggered, which in turn serves the adapter output `AS_OUT` (event `E1`) and outputs the saved value via `D1`.

... 3. **Feedback**

The internal `INI` module provides the output signals `QO` and `STATUS`, which are directly routed to the corresponding outputs of the `INI_AS` module.

Technical Features

- **Adapter Interface `AS`**: The module uses a unidirectional adapter (`adapter::types::unidirectional::AS`). This enables standardized connectivity to other modules that support the same adapter type.

- **Data Type SINT**: The read and stored value is of the “Short Integer” (SINT, 8-bit) type. This makes the module particularly suitable for small integer configuration values.
- **Reuse of the `INI` block**: All logic for accessing the INI file is taken over from the proven `eclipse4diac::storage::INI` block. The `INI_AS` block encapsulates this and extends the adapter interface.
- **Automatic GET after INIT**: After initialization, a read operation is started immediately, so the current value is available at the adapter output without a separate event.

State Overview

The block does not have an explicit state machine (ECC). Its behavior is controlled purely via event chaining within the internal FB network:

- After `INIT`, the sequence is: INIT of the `INI` block ⟶ INITO ⟶ GET (automatic) ⟶ VALUO at the adapter output.
- After a SET event from the adapter input: SET of the `INI` block ⟶ SETO ⟶ adapter output with the new value.

Application Scenarios

- **Persistent Configuration**: Saving and reading settings such as thresholds, mode flags, or device addresses in a `settings.ini` block.
- **Parameterization of Control Applications**: When a PLC or other automation system needs to load or modify values from a configuration file at runtime.
- **Adapter-Based Communication**: Integration into a higher-level adapter structure that groups several such read/write blocks.

Comparison with Similar Function Blocks

- **INI_STRING, INI_INT, INI_BOOL**: These function blocks also use the internal `INI` function block, but support different data types (STRING, INT, BOOL) and often have a different interface structure (no adapters). The `INI_AS` function block offers a uniform and reusable interface through its adapter.
- **Direct `INI` Function Block**: The `INI` function block itself has several event inputs (INIT, GET, SET) and requires manual wiring of read and write operations. The `INI_AS` simplifies handling by standardizing typical usage (reading on startup, writing via adapters).

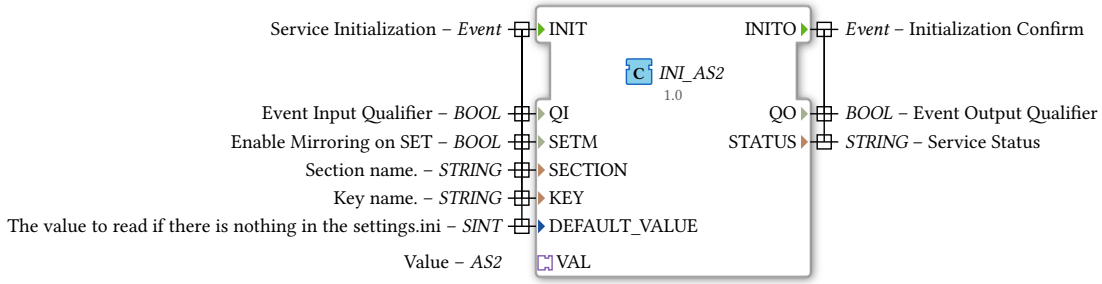
Conclusion

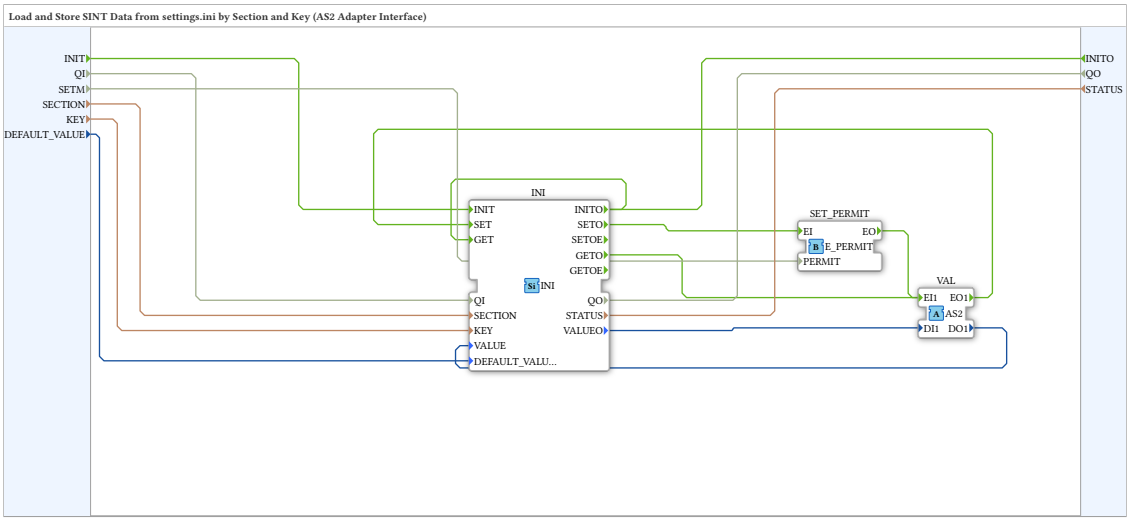
The function block `INI_AS` is a convenient building block for persistently reading and writing SINT values from an INI file. Its adapter interface allows for easy integration into existing architectures and reduces wiring effort. It is particularly suitable for configuration tasks where a single integer value needs to be cached.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

INI_AS2





• – ○ • – ○ • – ○ •

Introduction

The INI_AS2 function block is used to read and write SINT data from a settings.ini file via an AS2 adapter. It combines the INI block with a bidirectional AS2 adapter to read values from or write them back to a configuration file. Initialization is performed via the INIT event, where the section and key are specified. The read value is offered via the adapter, and a write operation is triggered via the adapter event.

Interface Structure

Event Inputs

Name	Description
INIT	Initialization event to start the read operation. Expects the parameters QI, SECTION, KEY, and DEFAULT_VALUE.

Event Outputs

Name	Description
INITO	Initialization confirmation. Sent after the read operation is complete. Status information is available via QO and STATUS.

Data Inputs

Name	Type	Description
QI	BOOL	Input qualifier for controlling processing.
SETM	BOOL	Controls confirmation/mirroring after a successful write. Regardless of SETM, VAL.EO1 always triggers INI.SET; however, the SETO confirmation reaches VAL.EI1 only when SETM=TRUE (suppressed when SETM=FALSE).
SECTION	STRING	Name of the section in the settings.ini file.
KEY	STRING	Name of the key within the section.
DEFAULT_VALUE	SINT	Default value read if the key is not present in the file. Default: 0.

Data Outputs

Name	Type	Description

QO	BOOL	Output qualifier, indicates successful processing.
STATUS	STRING	Service status message (e.g., error messages).

Adapters

Name	Type	Description
VAL	adapter::types::bidirectional::AS2	Bidirectional adapter for exchanging values. This adapter provides the read value as an output (DI1) and receives a write value as an input (DO1).

Functionality

The INI_AS2 function block contains an internal INI function block (eclipse4diac::storage::INI) that performs the actual file operation. The network connections implement the following processes:

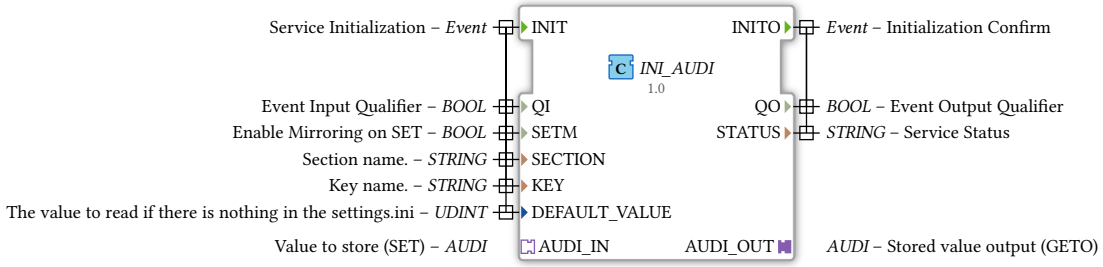
- Upon arrival of INIT, QI, SECTION, KEY, and DEFAULT_VALUE are forwarded to the INI function block.
- The INI function block performs a read operation and passes the result (via VALUEO) to the adapter output (VAL.DI1).
- Simultaneously, after a successful read, the GET event is triggered, which activates the adapter input (VAL.EI1) to transmit the value.
- A write operation is initiated as soon as the adapter sends an EO1 event. This event triggers the SET event on the INI function block, setting the value received via VAL.DO1 as the new VALUE.
- The INI block confirms the write operation with SETO. If SETM=TRUE, this confirmation is routed back to the adapter (VAL.EI1), whereas SETM=FALSE suppresses this mirror.
- The QO and STATUS outputs are directly taken from the INI block.

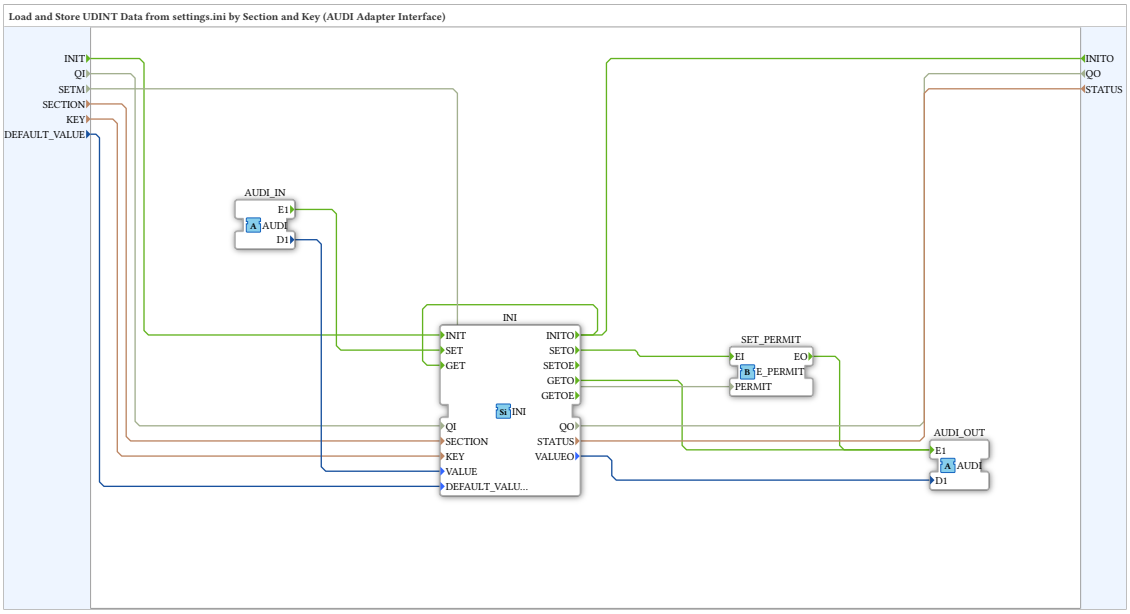
Thus, read and write access to the settings.ini file can be controlled via the AS2 adapter.

Technical Features

- The block uses a bidirectional AS2 adapter that can both send and receive values.
- A SINT value is used as the default value.

INI_AUDI





• – ○ • – ○ • – ○ •

Introduction

The **INI_AUDI** function block enables reading and saving UDINT data from a `settings.ini` file. Access to the file is implemented via an internal `INI` function block. Communication with the environment is handled via two **AUDI adapters** (unidirectional), implemented as a plug (`AUDI_OUT`) and a socket (`AUDI_IN`). This allows for modular integration with other function blocks that use the same adapter type.

Interface Structure

Event Inputs

Name	Type	Comment
INIT	EInit	Service Initialization

Event Outputs

Name	Type	Comment
INITO	EInit	Initialization Confirm

Data Inputs

Name	Type	Comment
QI	BOOL	Event Input Qualifier
SETM	BOOL	Enable mirroring on SET
SECTION	STRING	Section name in settings.ini
KEY	STRING	Key name in settings.ini
DEFAULT_VALUE	UDINT	Fallback value if the key is not found

Data Outputs

Name	Type	Comment
QO	BOOL	Output Qualifier

STATUS	STRING	Service Status

Adapter

Name	Type	Direction	Comment
AUDI_OUT	AUDI	Plug (Output)	Output of the read value
AUDI_IN	AUDI	Socket (Input)	Input of the value to be stored

Functionality

This function block works in conjunction with the internal `INI` function block. The basic sequence is:

1. **Initialization (`INIT` event)**

- The input data associated with `INIT` (`QI`, `SETM`, `SECTION`, `KEY`, `DEFAULT_VALUE`) is forwarded to the internal `INI` module.
- `INI` is started and acknowledged with `INITO`.

1. **Reading a Value**

- After INIT confirmation, the `GET` command of the `INI` block is automatically triggered.
- The read value (or `DEFAULT_VALUE`) is output to the `AUDI_OUT.D1` adapter via `INI.VALUEO`.
- Simultaneously, the event `AUDI_OUT.E1` is sent to notify the connected block.

1. **Writing a Value**

- If the adapter `AUDI_IN` receives an event `E1` with a data value `D1`, `INI_AUDI` forwards this to the `INI` block as a `SET` command.
- The internal `INI` block stores the value in the INI file under the specified section and key and acknowledges it with `SETO`.
- The acknowledgment event is output via `AUDI_OUT.E1`.

... 4. **Feedback**

- The status outputs `QO` and `STATUS` are directly inherited from the internal `INI` block and are available at the output.

Technical Features

- The block uses the **adapter `adapter::types::unidirectional::AUDI`**, which is specifically designed for the directed communication of values and events.
- The data is processed as **UDINT (Unsigned Double Integer)**, enabling efficient storage and transmission.
- The internal `INI` block is from the `eclipse4diac::storage` library and supports standard INI file formats.
- The block uses the **adapter `adapter::types::unidirectional::AUDI`**, which is specifically designed for the directed communication of values and events. - The fallback mechanism via `DEFAULT_VALUE` ensures that a defined value is returned even if entries are missing in the INI file.

State Overview

Since the function block operates in an event-driven manner, the following logical states result:

- **Idle**: Waiting for a `INIT` event.
- **Initialize**: After `INIT` – internal forwarding to the INI function block.
- **Read** (automatically after `INITO`): The `INI` function block executes a GET command.
- **Write**: Upon receiving an event via `AUDI_IN.E1`.
- **Confirm**: Output of the corresponding OUT event and update of the status outputs.

The function block is stateless in the sense of a continuous data flow; each action is triggered by an event and terminates synchronously.

Application Scenarios

- **Configuration Management** in automation systems: Reading settings (e.g., limits, parameters) from a central `settings.ini` and writing back changed values.

- **Adapter-based sensor/actuator coupling**: Connecting a function block that reads or sets values via the AUDI adapter without having to implement the file access logic itself.
- **Persistent Data Storage**: Saving counter readings or machine states to an INI file while simultaneously communicating via standardized adapters.

Comparison with Similar Function Blocks

- **Compared to a pure `INI` function block**: `INI_AUDI` encapsulates the adapter communication and offers a higher level of abstraction. The user does not have to directly handle events and data connections between multiple components.
- **Compared to Function Blocks with Other Data Types**: This function block is specifically designed for `UDINT`. Separate versions are required for other data types (e.g., `STRING`, `REAL`). However, the basic structure remains the same.
- **Advantage**: By using adapters (AUDI), the function block can easily be replaced with other interfaces, provided they implement the same adapter type.

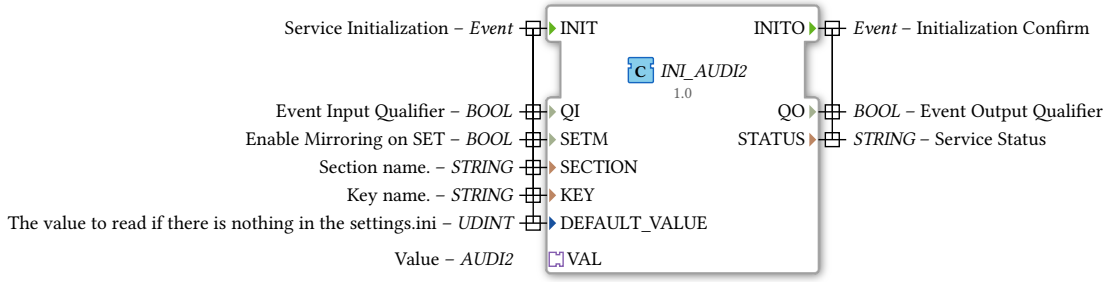
Conclusion

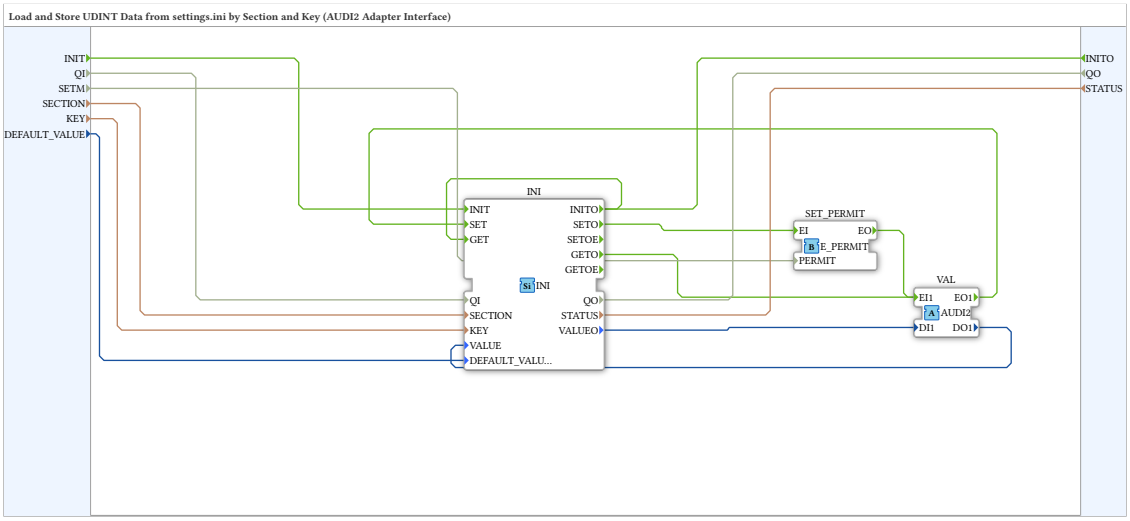
The `INI_AUDI` function block provides a compact and reusable solution for reading and writing UDINT values from an INI file, connecting to the environment via standardized AUDI adapters. Its clear separation of initialization, reading, and writing, along with the inclusion of a fallback value, makes it particularly suitable for robust configuration and data storage tasks in automation environments. Its simple structure facilitates integration into complex 4diac networks.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

INI_AUDI2





• – ○ • – ○ • – ○ •

Introduction

The INI_AUDI2 function block is used to read and write UDINT data from a settings.ini file via an AUDI2 adapter. It combines the INI block with a bidirectional AUDI2 adapter to read values from or write them back to a configuration file. Initialization is performed via the INIT event, where the section and key are specified. The read value is offered via the adapter, and a write operation is triggered via the adapter event.

Interface Structure

Event Inputs

Name	Description
INIT	Initialization event to start the read operation. Expects the parameters QI, SECTION, KEY, and DEFAULT_VALUE.

Event Outputs

Name	Description
INITO	Initialization confirmation. Sent after the read operation is complete. Status information is available via QO and STATUS.

Data Inputs

Name	Type	Description
QI	BOOL	Input qualifier for controlling processing.
SETM	BOOL	Controls confirmation/mirroring after a successful write. Regardless of SETM, VAL.EO1 always triggers INI.SET; however, the SETO confirmation reaches VAL.EI1 only when SETM=TRUE (suppressed when SETM=FALSE).
SECTION	STRING	Name of the section in the settings.ini file.
KEY	STRING	Name of the key within the section.
DEFAULT_VALUE	UDINT	Default value read if the key is not present in the file. Default: 0.

Data Outputs

Name	Type	Description
QO	BOOL	Output qualifier, indicates successful processing.
STATUS	STRING	Service status message (e.g., error messages).

Adapters

Name	Type	Description
VAL	adapter::types::bidirectional::AUDI2	Bidirectional adapter for exchanging values. This adapter provides the read value as an output (DI1) and receives a write value as an input (DO1).

Functionality

The INI_AUDI2 function block contains an internal INI function block (eclipse4diac::storage::INI) that performs the actual file operation. The network connections implement the following processes:

- Upon arrival of INIT, QI, SECTION, KEY, and DEFAULT_VALUE are forwarded to the INI function block.
- The INI function block performs a read operation and passes the result (via VALUEO) to the adapter output (VAL.DI1).

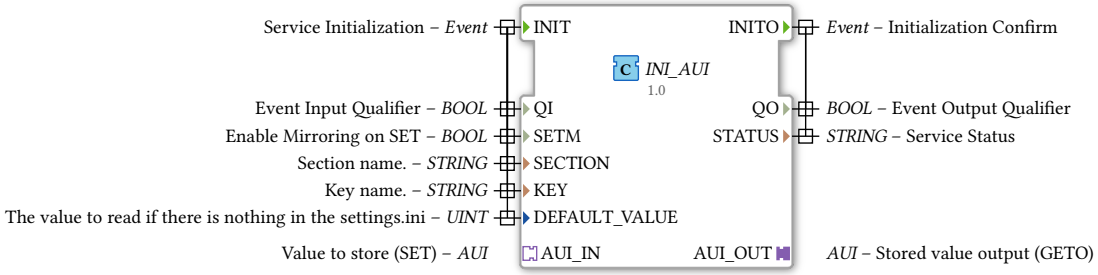
- Simultaneously, after a successful read, the GET event is triggered, which activates the adapter input (VAL.EI1) to transmit the value.
- A write operation is initiated as soon as the adapter sends an EO1 event. This event triggers the SET event on the INI function block, setting the value received via VAL.DO1 as the new VALUE.
- The INI block confirms the write operation with SETO. If SETM=TRUE, this confirmation is routed back to the adapter (VAL.EI1), whereas SETM=FALSE suppresses this mirror.
- The QO and STATUS outputs are directly taken from the INI block.

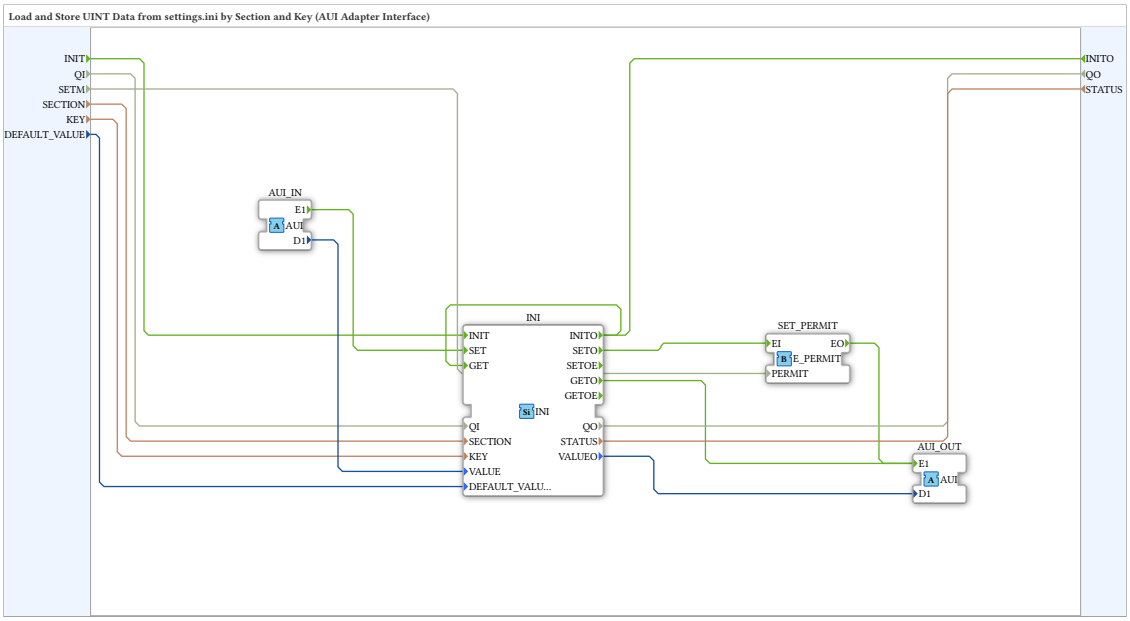
Thus, read and write access to the settings.ini file can be controlled via the AUDI2 adapter.

Technical Features

- The block uses a bidirectional AUDI2 adapter that can both send and receive values.
- A UDINT value is used as the default value.

INI_AUI





• – ○ • – ○ • – ○ •

Introduction

The function block `INI_AUI` is used to read and store **UINT data** (more precisely: `UDINT` values) from a `settings.ini` file. The parameters **Section** and **Key** determine which value is read. Additionally, a **Default Value** can be specified if no entry exists in the INI file. The function block offers both direct access via its inputs and an **Adapter Interface (AUI)** for unidirectional communication with other function blocks.

Interface Structure

Event Inputs

Event	Type	Short Description

INIT	EInit	Service Initialization: Triggers loading the value from the INI file

Event Outputs

Event	Type	Short Description
INITO	EInit	Initialization Acknowledgement (sent after completion of read/write operations)

Data Inputs

Name	Type	Short Description
QI	BOOL	Qualifier for the event input (enables processing)
SETM	BOOL	Enable Mirroring on SET: if TRUE, the confirmation of a SET (SETO) is additionally forwarded to the output adapter (mirrored)
SECTION	STRING	Name of the section in `settings.ini` (e.g., `[MySection]`)
KEY	STRING	Key name within the section
DEFAULT_VALUE	UDINT	Value returned if no entry exists

Data Outputs

Name	Type	Short Description
QO	BOOL	Qualifier for the event output (indicates successful execution)
STATUS	STRING	Status message (e.g., Error or Success)

Adapters

Direction	Adapter Type	Short Description
Plug	`adapter::types::unidirectional::AUI`	**Output (OUT):** Provides the read/written value via the adapter
Socket	`adapter::types::unidirectional::AUI`	**Input (IN):** Receives the value to be stored via the adapter

Functionality

The `INI_AUI` function block encapsulates an internal `INI` function block (`eclipse4diac::storage::INI`). The essential processes are:

1. **Reading a Value**

- An **INIT** event at the input triggers the internal `INI.INIT`.
- The data `QI`, `SETM`, `SECTION`, `KEY`, and `DEFAULT_VALUE` are forwarded to the `INI` function block.
- After successful processing, `INI.INITO` sends the confirmation event and internally triggers `INI.GET` to load the value from the file.
- The read value appears at `INI.VALUEO` and is output via the **AUI_OUT** adapter (plug) as `D1`.
- Simultaneously, `QO` and `STATUS` are taken from the internal function block.

1. **Writing a Value**

- An event at the **AUI_IN.Socket** (via the input adapter) triggers the internal `INI.SET`.
- The value provided via the adapter (D1) is passed to `INI.VALUE` and written to `settings.ini`.
- After writing, `INI.SETO` sends the confirmation event, which is output via the **AUI_OUT.Plug** as `E1`.
- Here too, the outputs `QO` and `STATUS` are updated.

> **Note:** The adapter **AUI_IN** acts as a socket (receiving), and **AUI_OUT** as a plug (providing). Both use the same unidirectional AUI type.

Technical Features

- **Adapter-based communication:** The function block enables the exchange of configuration data via a unidirectional adapter (AUI) without requiring direct data connections. This simplifies module communication in distributed systems.
- **Dual operation:** The value can be set via the classic data inputs (`SECTION`, `KEY`, `DEFAULT_VALUE`) as well as via the adapter (`AUI_IN.D1`).
- **Default Value:** If an entry is missing in the INI file, `DEFAULT_VALUE` is used – this prevents undefined states.
- **Status Information:** The user can check the success of each operation via `STATUS` and `QO`.

State Overview

The function block `INI_AUI` does **not have its own state machine**. The entire process control is implemented by the integrated `INI` function block and the defined event connections (e.g., INIT → INI.INIT, INI.INITO → INITO, INI.INITO → INI.GET). Therefore, the function block operates strictly event-driven and executes read/write operations sequentially.

Application Scenarios

- **Parameter Management** in modular automation systems where multiple components access shared configuration data via an adapter.
- **Initialization** of control functions with a default value that can be overwritten from an INI file if needed.
- **Exchange of settings** between different function blocks connected via the AUI adapter (e.g., a higher-level manager block and multiple worker blocks).

Comparison with Similar Blocks

- **`INI` (Basic Function Block):** The `INI_AUI` extends the simple `INI` function block with a standardized adapter interface. While the `INI` only has direct inputs/outputs, the `INI_AUI` allows loose coupling via AUI.
- **Other memory blocks (e.g., `Memory`, `Persist`):** These usually work with internal variables or files, but often lack an adapter interface and dedicated `settings.ini` integration.

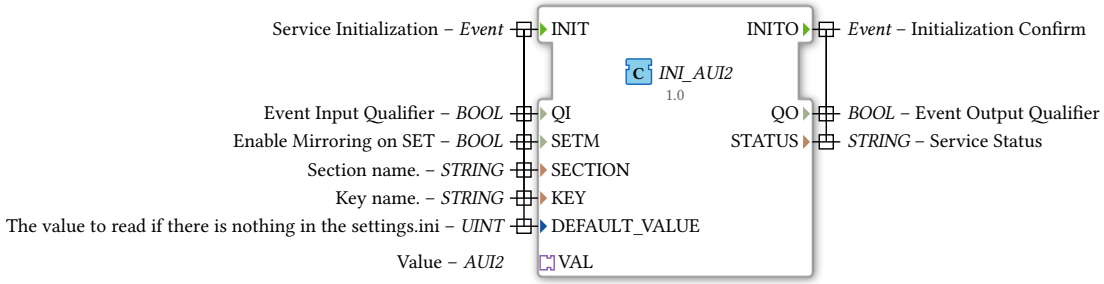
Conclusion

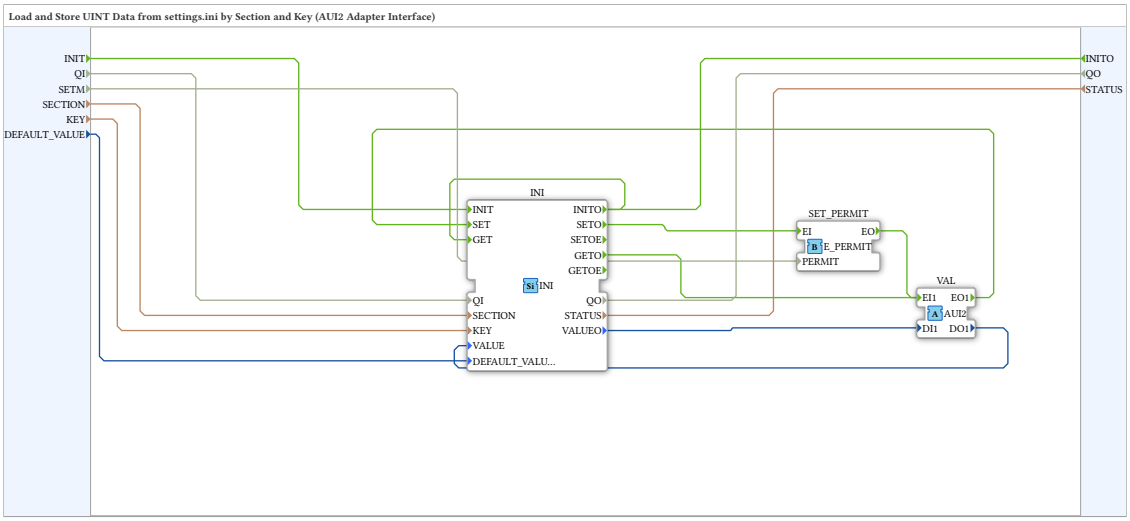
The `INI_AUI` function block is a flexible solution for reading and saving configuration values in a `settings.ini`-file. By combining direct data inputs and a unidirectional AUI adapter interface, it is ideally suited for both simple initialization tasks and modular, adapter-based system architectures. The integration of a default value and clear status feedback increase the robustness and maintainability of the control software.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

INI_AUI2





• – ○ • – ○ • – ○ •

Introduction

The INI_AUI2 function block is used to read and write UINT data from a settings.ini file via an AUI2 adapter. It combines the INI block with a bidirectional AUI2 adapter to read values from or write them back to a configuration file. Initialization is performed via the INIT event, where the section and key are specified. The read value is offered via the adapter, and a write operation is triggered via the adapter event.

Interface Structure

Event Inputs

Name	Description
INIT	Initialization event to start the read operation. Expects the parameters QI, SECTION, KEY, and DEFAULT_VALUE.

Event Outputs

Name	Description
INITO	Initialization confirmation. Sent after the read operation is complete. Status information is available via QO and STATUS.

Data Inputs

Name	Type	Description
QI	BOOL	Input qualifier for controlling processing.
SETM	BOOL	Controls confirmation/mirroring after a successful write. Regardless of SETM, VAL.EO1 always triggers INI.SET; however, the SETO confirmation reaches VAL.EI1 only when SETM=TRUE (suppressed when SETM=FALSE).
SECTION	STRING	Name of the section in the settings.ini file.
KEY	STRING	Name of the key within the section.
DEFAULT_VALUE	UINT	Default value read if the key is not present in the file. Default: 0.

Data Outputs

Name	Type	Description

QO	BOOL	Output qualifier, indicates successful processing.
STATUS	STRING	Service status message (e.g., error messages).

Adapters

Name	Type	Description
VAL	adapter::types::bidirectional::AUI2	Bidirectional adapter for exchanging values. This adapter provides the read value as an output (DI1) and receives a write value as an input (DO1).

Functionality

The INI_AUI2 function block contains an internal INI function block (eclipse4diac::storage::INI) that performs the actual file operation. The network connections implement the following processes:

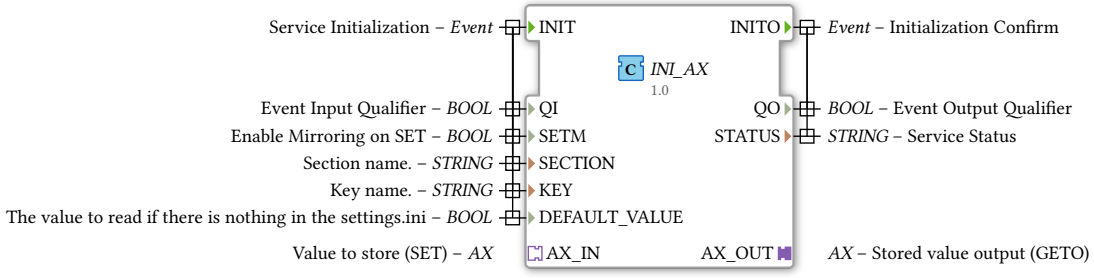
- Upon arrival of INIT, QI, SECTION, KEY, and DEFAULT_VALUE are forwarded to the INI function block.
- The INI function block performs a read operation and passes the result (via VALUEO) to the adapter output (VAL.DI1).
- Simultaneously, after a successful read, the GET event is triggered, which activates the adapter input (VAL.EI1) to transmit the value.
- A write operation is initiated as soon as the adapter sends an EO1 event. This event triggers the SET event on the INI function block, setting the value received via VAL.DO1 as the new VALUE.
- The INI block confirms the write operation with SETO. If SETM=TRUE, this confirmation is routed back to the adapter (VAL.EI1), whereas SETM=FALSE suppresses this mirror.
- The QO and STATUS outputs are directly taken from the INI block.

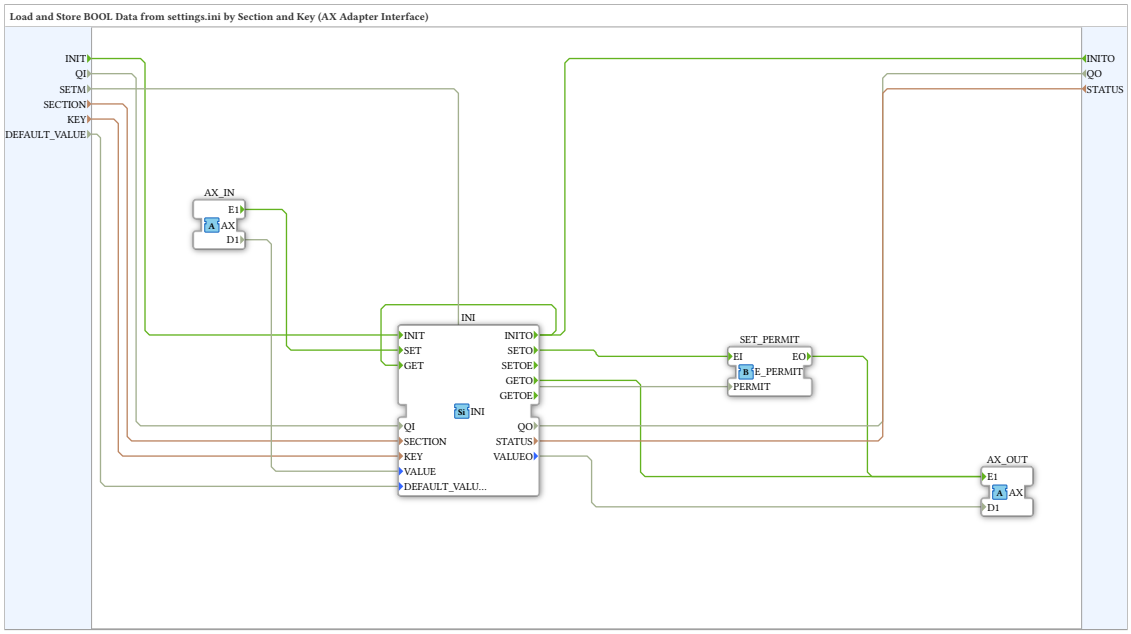
Thus, read and write access to the settings.ini file can be controlled via the AUI2 adapter.

Technical Features

- The block uses a bidirectional AUI2 adapter that can both send and receive values.
- A UINT value is used as the default value.

INI_AX





• – ○ • – ○ • – ○ •

Introduction

`INI_AX` loads and stores a `BOOL` value in the `settings.ini` file, addressed by section (`SECTION`) and key (`KEY`). It is the Boolean counterpart to INI_AB and the `settings.ini` counterpart to NVS_AX.

Interface Structure

Event Inputs

- **INIT**: initializes the block and reads the value stored under `SECTION`/`KEY` (or `DEFAULT_VALUE` if none exists yet). Carries `QI`, `SETM`, `SECTION`, `KEY`, `DEFAULT_VALUE`.

Event Outputs

- **INITO**: confirms initialization and the initial read, carries `QO` and `STATUS`.

Data Inputs

- **QI** (BOOL): qualifier for the `INIT` event.
- **SETM** (BOOL): enables mirroring (confirmation) on write (`SET`).
- **SECTION** (STRING): name of the section in `settings.ini`.
- **KEY** (STRING): name of the key within the section.
- **DEFAULT_VALUE** (BOOL): value returned when nothing is stored under `SECTION`/`KEY` yet.

Data Outputs

- **QO** (BOOL): success qualifier, carried with `INITO` and the internal adapter confirmations.
- **STATUS** (STRING): status message of the file access.

Adapters

- **AX_IN** (Socket, type `AX`): new value to store – an adapter event on `AX_IN.E1` triggers a `SET` on the internal INI block, `AX_IN.D1` carries the value to write.
- **AX_OUT** (Plug, type `AX`): outputs the most recently read value – both after the automatic initial read on `INIT` and after each confirmed write (`AX_OUT.D1`/`AX_OUT.E1`).

Functionality

`INI_AX` wraps an internal instance of `eclipse4diac::storage::INI` plus an `E_PERMIT` gate (`SET_PERMIT`). On `INIT`, the internal INI instance is initialized (`INI.INIT`) and a `GET` is triggered automatically, returning the value stored under `SECTION`/`KEY` (or `DEFAULT_VALUE`); the result is output via `AX_OUT`, and `INITO` confirms completion.

To write, the block connected to `AX_IN` fires an adapter event (`AX_IN.E1`) carrying the new value (`AX_IN.D1`), which triggers `INI.SET`. Whether the successful write is additionally reported back via `AX_OUT` is controlled by `SETM`: if `SETM` is set, `SET_PERMIT` releases the `AX_OUT.E1` event after a successful `SET` ("mirroring").

Technical Details

- Pure composite FB with no ECC of its own – the actual file access logic lives in the internal `INI` instance; `INI_AX` only wires events and data between it and the two `AX` adapters.
- Two separate unidirectional adapters (`AX_IN`/`AX_OUT`) – this lets the read and write paths connect to different peers.
- Addressing via `SECTION` **and** `KEY` (rather than just `KEY` for the `NVS_*` blocks) matches the classic INI file format.

State Overview

The block is stateless in terms of its own ECC: on `INIT` it runs through initialization plus an initial read once, then reacts to every `AX_IN` adapter event with an INI write and, optionally, a confirmation via `AX_OUT`.

Application Scenarios

- Persisting individual `BOOL` configuration values (e.g. enable flags, operating-mode switches) in a human-readable `settings.ini`
- Building a read/write pair where the read and write sides need to connect to different blocks
- Configuration files that also need to be viewable/editable outside the controller (e.g. with a text editor)

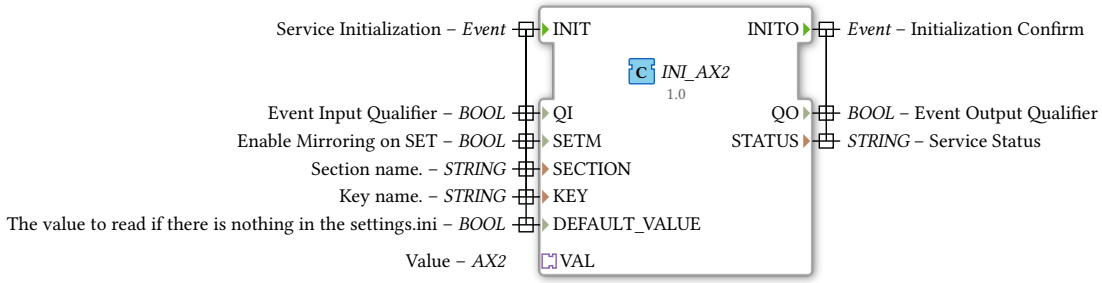
⚖️ Comparison with Similar Blocks

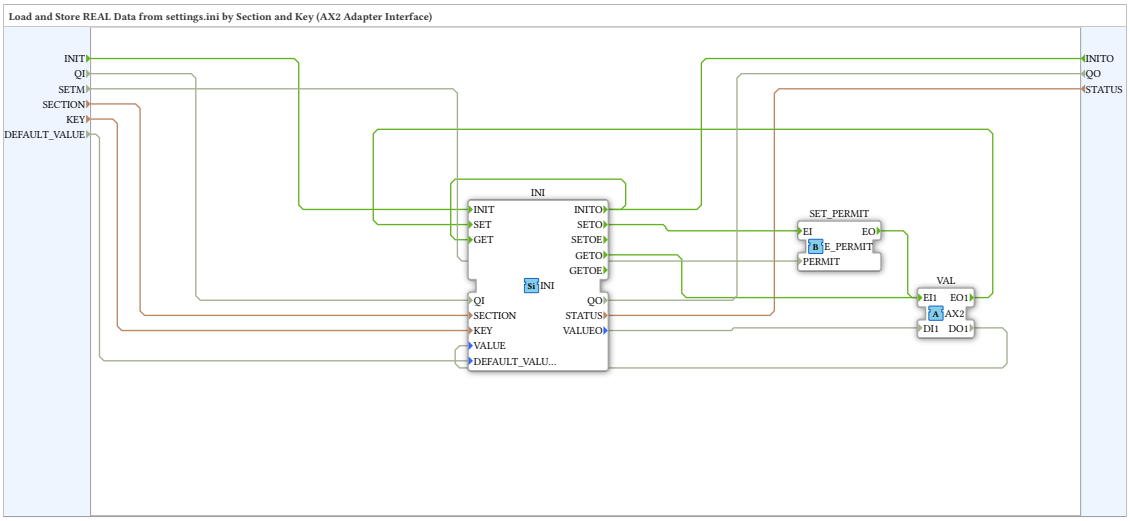
Compare with INI_AB, which uses the same architecture for `BYTE` instead of `BOOL` values, and with NVS_AX, which provides the same interface for the ESP32 NVS flash storage instead of `settings.ini`.

Conclusion

`INI_AX` provides simple, unidirectionally wired access to a persistent `BOOL` value in `settings.ini`, and fits anywhere the read and write paths need to be kept separate.

INI_AX2





• – ○ • – ○ • – ○ •

Introduction

The INI_AX2 function block is used to read and write real data from a settings.ini file via an AX2 adapter. It combines the INI block with a bidirectional AX2 adapter to read values from or write them back to a configuration file. Initialization is performed via the INIT event, where the section and key are specified. The read value is offered via the adapter, and a write operation is triggered via the adapter event.

Interface Structure

Event Inputs

Name	Description
INIT	Initialization event to start the read operation. Expects the parameters QI, SETM, SECTION, KEY, and DEFAULT_VALUE.

Event Outputs

Name	Description
INITO	Initialization confirmation. Sent after the read operation is complete. Status information is available via QO and STATUS.

Data Inputs

Name	Type	Description
QI	BOOL	Input qualifier for controlling processing.
SETM	BOOL	Enable mirroring on SET
SECTION	STRING	Name of the section in the settings.ini file.
KEY	STRING	Name of the key within the section.
DEFAULT_VALUE	BOOL	Default value read if the key is not present in the file. Default: FALSE.

Data Outputs

Name	Type	Description
QO	BOOL	Output qualifier, indicates successful processing.
STATUS	STRING	Service status message (e.g., error messages).

Adapters

Name	Type	Description
VAL	adapter::types::bidirectional::AX2	Bidirectional adapter for exchanging values. This adapter provides the read value as an output (DI1) and receives a write value as an input (DO1).

Functionality

The INI_AX2 function block contains an internal INI function block (eclipse4diac::storage::INI) that performs the actual file operation. The network connections implement the following processes:

- Upon arrival of INIT, QI, SETM, SECTION, KEY, and DEFAULT_VALUE are forwarded to the INI function block.
- The INI function block performs a read operation and passes the result (via VALUEO) to the adapter output (VAL.DI1).
- Simultaneously, after a successful read, the GET event is triggered, which activates the adapter input (VAL.EI1) to transmit the value.
- A write operation is initiated as soon as the adapter sends an EO1 event. This event triggers the SET event on the INI function block, setting the value received via VAL.DO1 as the new VALUE.
- The INI block confirms the write operation with SETO and sends this back to the adapter (VAL.EI1).
- The QO and STATUS outputs are directly taken from the INI block.

Thus, read and write access to the settings.ini file can be controlled via the AX2 adapter.

Technical Features

- The block uses a bidirectional AX2 adapter that can both send and receive values.
- A BOOL value is used as the default, although the comment indicates REAL data. The actual value type is determined by the adapter.
- The internal INI block is part of the eclipse4diac standard library for file access.
- The connections in the FBNetwork are configured so that event control is synchronized between read and write operations.

State Overview

The block itself does not have explicit states, as it is implemented as a pure network of subordinate blocks. The state logic is determined by the internal INI block and the adapter. Typical states are:

- **Initialization**: Waiting for an INIT event.
- **Read**: Processing the read operation via the INI block.
- **Write**: Processing a write command via the adapter.
- **Wait**: After completing an operation, waiting for the next event.

Application Scenarios

- Reading configuration parameters from an INI file in an automation application, where the values are passed to other components via an AX2 adapter.
- Writing modified parameters back to the file, e.g., after user input or an algorithm.
- Combining with other blocks for persistent storage of settings in an IEC 61499 system.

Comparison with Similar Blocks

- The simple **INI** block offers direct read/write access to INI files, but without adapter connectivity. INI_AX2 extends this with a standardized AX2 interface for value exchange.

Other memory blocks, such as **CSV** or **JSON** blocks, serve different file formats; INI_AX2 is specific to the INI format with a bidirectional adapter.

Conclusion

The INI_AX2 function block is a useful component for IEC 61499 applications that require easy access to configuration files via an AX2 adapter. By encapsulating the INI block and using event-driven control, it enables both read and write operations to a settings.ini file. The use of a bidirectional adapter allows for flexible integration into complex networks.

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Read-Only Settings (`settingsReadOnly.ini`)

In logiBUS® systems, configuration values and parameters are managed in INI files on the ECU's file system. In addition to the freely writable standard configuration file (`settings.ini`), the system supports a read-only settings file: **`settingsReadOnly.ini`**. This feature is used to permanently freeze factory settings, manufacturer-defined system parameters, fixed communication addresses, or critical hardware and security options. This prevents important parameters from being accidentally changed by operators or by the control system itself

---

Introduction

Concept of the Two Configuration Files

The ECU distinguishes between two levels of configuration storage:

File	Purpose	Writable?	Typical Contents
`settingsReadOnly.ini`	Factory Settings & System Defaults	❌ No (Write-protected)	ISOBUS source addresses (Node SAs), hardware pin assignments, fixed boot times
`settings.ini`	User & Runtime Settings	✅ Yes (Read & Write)	User preferences, dynamic operating parameters, selectable limit scaling

1. **Factory Settings Priority**: During boot, the control unit first reads the file `settingsReadOnly.ini`. All sections (`[Section]`) and keys (`KEY`) defined within it are loaded in read-only mode.
2. **Automatic Purge:** If a key is located in both `settingsReadOnly.ini` and the writable `settings.ini`, the duplicate key is automatically **removed** from `settings.ini` at system startup. This ensures that no conflicting values exist and the user file remains uncluttered.
3. **Additional Loading:** All parameters that are *not* located in `settingsReadOnly.ini` are loaded from `settings.ini` and behave normally as writable files.

Parameters are read transparently. If a key is stored in `settingsReadOnly.ini`, its immutable value is always returned.

If an attempt is made to change a read-only key (whether via control functions or 4diac IEC 61499 function blocks), the following occurs:

- The **write attempt is rejected**.
- The value defined in `settingsReadOnly.ini` **remains unchanged**.
- The rejection of the write operation is logged and reported back to the calling block logic.

Schreibzugriffe (`SET`)

Lesezugriffe (`GET`)

Verhalten im Betrieb (Lesen & Schreiben)

Functionality & Priorität beim Start

Behavior in 4diac FORTE (IEC 61499 INI Blocks)

For users of 4diac FORTE control programs, the INI function blocks (`INI`, `INI_AX`, `INI_AUI`, `INI_AR`, etc.) behave as follows with read-only parameters:

- **Read event (`GET`)**: Signals the confirmation event `GETO` as usual. The protected value is present at output `VALUEO`, and `STATUS` reports `"OK"`.
- **Write Event (`SET`)**:
- The **normal success event (`SETO`)** is not triggered, but rather the error event **`SETOE`** (*Set Output Error*).

- The data output **STATUS** provides the understandable message: **"Key is read-only"**.
- The output **QO** indicates the error state.

!!! note “Note for Application Developers” By evaluating the output event SETOE or the status string "Key is read-only", the 4diac application can react specifically to read-only parameters (e.g., displaying a note on the visualization).

The file settingsReadOnly.ini is located on the ECU's memory under: /data/settingsReadOnly.ini

Importing or updating the factory settings is easily done via the ECU's integrated **Web Interface** (File Server):

1. Connect to the ECU's web interface using a web browser.
2. Navigate to the file management (/data).
3. Upload your prepared settingsReadOnly.ini file.
4. After restarting the ECU, the read-only parameters will take effect.

If settingsReadOnly.ini is not present on the ECU, the system will start normally without any limitations. All keys in settings.ini remain fully writable, as before.

Praktisches Beispiel

Verhalten bei fehlender Datei

Übertragung auf das Steuergerät

Einspielen und Verwalten der Werkseinstellungen

Example file settingsReadOnly.ini (factory settings)

[CF-A] UserLanguage = DE
[User] OperatorID = 42
Result in operation:

- NODE1_SA (128), bootTimeVT (90), and DeviceName ("LOGIBUS-NODE-01") are protected and cannot be modified.
- UserLanguage ("DE") and OperatorID (42) can be freely modified and permanently saved by the controller.
- If an old value, NODE1_SA = 130, was accidentally present in settings.ini, the control unit will automatically delete it during boot, ensuring that the default value, 128, is always used.

Beispieldatei settings.ini (Benutzereinstellungen)

Summary & Best Practices

Goal	Recommended Procedure
Save Factory Settings	Create all immutable parameters (e.g., bus addresses, security limits) in settingsReadOnly.ini and upload them to the control unit.
Allow User Settings	Keep parameters that the operator is allowed to adjust (e.g., language, operating modes, counter readings) exclusively in settings.ini.
Enable Parameters	To make a parameter writable again, remove the corresponding key from settingsReadOnly.ini and restart the control unit.

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

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

2.10.3.12 utils

conversion

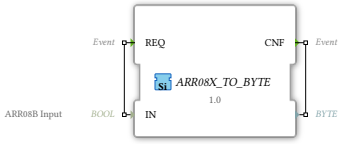
arr

🎧 Podcast

- Switching vs. Linear Controllers: Efficiency, EMC & the Compromise of Power Electronics

assembling

ARR08X_TO_BYTE



• — ○ • — ○ • — ○ •

Introduction

The function block `ARR08X_TO_BYTE` converts an array of eight Boolean values (`BOOL`) into a single byte value (`BYTE`). Each bit in the resulting byte corresponds to an element of the input array. This operation is useful for bundling multiple binary signals (e.g., switch states or error flags) to save space and preparing them for further processing or communication.

Interface Structure

Event Inputs

- `REQ` (Request): Triggers the conversion operation. Upon receiving an event at this input, the current value of the input array `IN` is read and converted into a byte.

Event Outputs

- `CNF` (Confirm): Triggered as soon as the conversion is complete. This event signals that the output value (the byte) is valid and current.

Data Inputs

- `IN` (Array[0..7] of BOOL): The input array containing the eight Boolean values to be converted. The index `0` corresponds to the least significant bit (LSB, bit 0) of the resulting byte, and the index `7` corresponds to the most significant bit (MSB, bit 7).

Data Outputs

- `OUT` (BYTE): The output value composed of the eight Boolean inputs. The data output is not explicitly named (empty name in the interface) but is accompanied by the event output `CNF`.

Adapters

This function block does not use any adapters.

Operation

Upon receiving a `REQ` event, the block performs a bitwise assignment. Each bit of the internal result byte (`ARR08X_TO_BYTE`) is directly assigned to the corresponding element of the input array `IN`. After these assignments are complete, the `CNF` event is generated, and the calculated byte value is available at the output.

The underlying ST code logic is:
ARR08X_TO_BYTE.0 := IN[0]; ARR08X_TO_BYTE.1 := IN[1]; ... ARR08X_TO_BYTE.7 := IN[7];

Technical Features

- **Hard-wired size:** This block is specifically designed for converting exactly 8 Boolean values. Different blocks are required for other array sizes.
- **Direct bit mapping:** The mapping is deterministic and performed without any arithmetic operations, enabling very fast and resource-efficient execution.
- **Index bit mapping:** It is critical to observe the mapping of array index to bit position in the byte: `IN[0]` ⟶ Bit 0 (LSB), `IN[7]` ⟶ Bit 7 (MSB).

State overview

This function block is stateless. It has no internal memory between calls. Its output at any given time is solely a direct function of the current input values when the `REQ` event occurs.

Application Scenarios

- **Communication Protocols:** Bundling multiple binary device statuses into a single data byte for transmission via serial interfaces or fieldbuses (e.g., PROFIBUS, CANopen).
- **Memory Optimization:** More compact storage of up to 8 switching states or alarm conditions in a single byte variable.
- **PLC Internal Data Handling:** Preparing data for blocks or functions that expect byte or word data types instead of individual Boolean values.

⚖️ Comparison with Similar Building Blocks

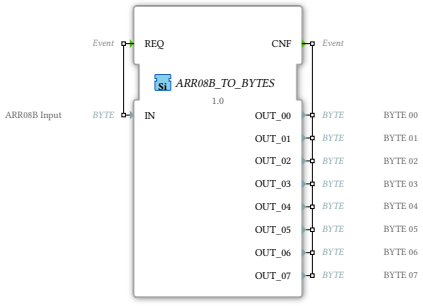
- **`BOOL_TO_BYTE` blocks:** Many libraries offer blocks that combine eight individual `BOOL` inputs (e.g., `IN0`...`IN7`) into a `BYTE`. `ARR08X_TO_BYTE`, on the other hand, uses an array as its input, which often makes wiring clearer when dealing with many signals from a common source (e.g., a previous array output).
- **`MERGE`/`PACK` blocks:** More general blocks can combine different data types. `ARR08X_TO_BYTE` is more specialized, simpler, and more efficient for its specific purpose.
- **Bitwise Shifting (`SHL`, `OR`):** The same function could be implemented using a loop and bitwise operations in ST. This pre-built block offers a standardized, ready-to-use, and less error-prone solution.

Conclusion

The `ARR08X_TO_BYTE` function block is a simple yet extremely useful tool for signal processing in IEC 61499 systems. It provides an elegant and efficient method for converting eight Boolean values from an array into a compact byte. Its clear interface and deterministic operation make it a reliable component for automation applications, especially where data needs to be prepared for communication or space-optimized storage.

forwarding

ARR08B_TO_BYTES



• – ○ • – ○ • – ○ •

The function block `ARR08B_TO_BYTES` is used to decompose an 8-element byte array (`ARR08B`) into eight individual byte outputs. It is a utility block for data conversion and forwarding within control applications, especially when structured data from an array needs to be split into discrete signals.

- **REQ**: This incoming event triggers the processing of the function block. Upon its arrival, the data at input `IN` is read and distributed to the individual outputs.

- **CNF**: This outgoing event signals the completion of processing. It is generated simultaneously with the output of the new values at data outputs `OUT_00` to `OUT_07`.
- **IN** (`ARRAY[0..7] OF BYTE`): The input for the 8-byte array to be split. The array has a fixed index range from 0 to 7.
- **OUT_00** (`BYTE`): Contains the value of the array element with index 0 (`IN[0]`).
- **OUT_02** (`BYTE`): Contains the value of the array element with index 2 (`IN[2]`).
- **OUT_03** (`BYTE`): Contains the value of the array element with index 3 (`IN[3]`).
- **OUT_04** (`BYTE`): Contains the value of the array element with index 4 (IN[4]) ... * **OUT_05** (`BYTE`): Contains the value of the array element with index 5 (`IN[5]`).
- **OUT_06** (`BYTE`): Contains the value of the array element with index 6 (`IN[6]`).
- **OUT_07** (`BYTE`): Contains the value of the array element with index 7 (`IN[7]`).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block has no adapter interfaces.

The operation is deterministic and simple: Upon each occurrence of the `REQ` event, the function block reads the entire input array `IN`. Each element of the array is then assigned to a specific byte output. The assignment is made directly according to the array index. After all assignments have been completed, the `CNF` event is triggered to report the successful completion of the operation to subsequent blocks.

The included ST (Structured Text) code shows the direct implementation:
OUT_00 := IN[0]; OUT_01 := IN[1]; ... OUT_07 := IN[7];

- **Fixed Size:** The block is specifically designed for arrays with exactly 8 elements (`ARR08B`). Separate blocks are required for arrays of other lengths.
- **Synchronous Processing:** Processing occurs in a single cycle without delay. The latency between `REQ` and `CNF` is minimal.
- **No Internal State:** The block has no internal memory. The output is a direct function of the current input at any given time.

The function block has no internal states in the sense of a state machine. Its behavior is purely combinatorial and is entirely controlled by the interface events:

1. **Idle State:** Waiting for the `REQ` event.
2. **Execution State:** At `REQ`, the data is transferred and the `CNF` event is immediately generated. The block then returns to its idle state.

- **Data Distribution:** Splitting a received data packet (e.g., from a communication interface) into its individual byte components for further processing in the application.
- **Interface Adapter:** Connecting function blocks that work with array structures to blocks that expect individual byte inputs.
- **Data Logging and Visualization:** Separating array data to selectively display or store individual bytes in HMIs or logging systems.
-
- **`SEL_n` / `MUX` blocks:** These blocks select an element from an array based on an index. `ARR08B_TO_BYTES`, on the other hand, extracts **all** elements simultaneously and in parallel.
- **`BYTES_TO_ARR08B`:** This is the inverse block, which combines eight individual byte inputs into an 8-byte array.
- **Generic Array Splitting Blocks:** More generic blocks might exist that take the array length as a parameter. `ARR08B_TO_BYTES`, due to its fixed size, is simpler, faster, and requires no configuration.

•

The `ARR08B_TO_BYTES` function block is a specialized, efficient, and reliable building block for the fundamental operation of array decomposition. Its strength lies in its simplicity and unambiguity for the specific use case of an 8-byte array. For control applications where this data structure is common, it offers an optimized and maintainable solution compared to manual indexing in more complex blocks.

Functionality

Technical Features

State Overview

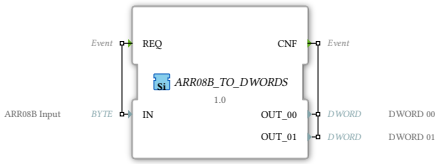
Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

ARR08B_TO_DWORDS

This function block converts an 8-byte array into two 32-bit data words (DWORDs). It is used to split a contiguous 8-byte data block into two 4-byte integer values.



• – ○ • – ○ • – ○ •

Introduction

The **ARR08B_TO_DWORDS** is a pure conversion function block that creates two separate DWORD values from an incoming 8-byte array. The conversion is performed byte by byte, with the first four bytes assigned to the first output (OUT_00) and the last four bytes to the second output (OUT_01). The function block is triggered by an event (REQ) and outputs a confirmation event (CNF) after the conversion is complete.

Interface Structure

Event Inputs

Event	Description	Data Carried
REQ	Starts the conversion of the input array.	IN (8-byte array)

Event Outputs

Event	Description	Data Carried
CNF	Confirms the completed conversion.	OUT_00, OUT_01

Data Inputs

Name	Data Type	Array Dimension	Description
IN	BYTE	[0..7]	8-byte input array that is split into two DWORDs.

Data Outputs

Name	Data Type	Description

OUT_00	DWORD	First DWORD (bytes 0–3 of the array).
OUT_01	DWORD	Second DWORD (bytes 4–7 of the array).

Adapters

No adapters available.

Functionality

This function block performs a direct byte-by-byte copy from the input array to the output variables:

1. Bytes **IN[0]** to **IN[3]** are assigned to the least significant to most significant bytes of **OUT_00**.
2. Bytes **IN[4]** to **IN[7]** are assigned to **OUT_01**.

The assignment is performed in the structured text body:
OUT_00.%B0 := IN[0]; OUT_00.%B1 := IN[1]; OUT_00.%B2 := IN[2]; OUT_00.%B3 := IN[3]; OUT_01.%B0 := IN[4]; OUT_01.%B1 := IN[5]; OUT_01.%B2 := IN[6]; OUT_01.%B3 := IN[7]; The function block operates without delay or state memory; the outputs are updated immediately upon arrival of the REQ event, and the CNF event is triggered instantly.

Technical Features

- **Endianness**: The function block interprets the array in little-endian format – the first byte (index 0) is assigned to the least significant byte of the DWORD. This assumption is typical for many industrial protocols and systems.
- **No Array Length Check**: The array is assumed to be exactly 8 bytes. A different input size would lead to undefined behavior.
- **Functional Function Block**: The function block is purely functional and has no internal states.

State Overview

The function block has no internal state machine. It operates event-driven and immediately returns the CNF signal after each activation. The outputs remain active until the next REQ activation.

Application Scenarios

- **Data Packet Decomposition**: Separation of an 8-byte telegram into two 32-bit values, e.g., for communication with industrial field devices (CANopen, Modbus, etc.).
- **Sensor Value Processing**: Splitting an 8-byte raw data block from sensors that deliver two physical measured values (4 bytes each).
- **Conversion in the logiBUS Environment**: Within the `logiBUS::utils::conversion::arr::forwarding` package, this is a standard function block for forward conversion of array data.

Comparison with Similar Function Blocks

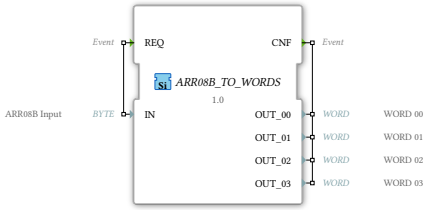
Function Block	Function	Difference
BYTE_TO_DWORD	Converts a single 4-byte array into a DWORD.	Requires separate calls for each 4-byte segment.
ARR08B_TO_DWORDS	Converts an 8-byte array into two DWORDs at once.	Integrated splitting in a single step reduces the number of calls and code.

This function block thus offers a more compact solution than combining several basic conversion blocks.

Conclusion

The **ARR08B_TO_DWORDS** is a simple yet useful function block for quickly and directly splitting an 8-byte array into two DWORDs. Thanks to its clear byte-to-byte mapping and event handling, it integrates well into dataflow-oriented control applications. Its use requires a little-endian format. Adjustments are necessary for other endianness requirements. Overall, the component is an efficient solution for preprocessing binary data.

ARR08B_TO_WORDS



• – ○ • – ○ • – ○ •

Introduction

The function block **ARR08B_TO_WORDS** is used to divide an 8-byte array into four 16-bit words (WORDs). It is typically used in data preprocessing when raw byte data from sensors or communication interfaces needs to be converted into integer values. The conversion is performed byte-wise, taking the little-endian byte order into account.

Interface Structure

Event Inputs

Event	Description
REQ	Trigger the conversion; The function block processes the current value of `IN` and outputs the results.

Event Outputs

Event	Description
CNF	Confirmation that the conversion is complete and the output values are valid.

Data Inputs

Name	Type	Array Size	Description
IN	BYTE	0...7	Input array of 8 bytes, which is split into four words.

Data Outputs

Name	Type	Description
OUT_00	WORD	Word formed from bytes IN[0] and IN[1] (least significant).
OUT_01	WORD	Word formed from bytes IN[2] and IN[3].
OUT_02	WORD	Word formed from bytes IN[4] and IN[5].
OUT_03	WORD	Word formed from bytes IN[6] and IN[7] (most significant).

Adapter

No adapters available.

Functionality

When the event `REQ` is triggered, the 8-byte array `IN` is processed sequentially. Each pair of consecutive bytes is combined into a 16-bit word, with the first byte (`IN[0]`) being the least significant byte (`%B0`) and the second byte (`IN[1]`) being the most significant byte (`%B1`) of the word (little-endian format). The resulting four words are made available at outputs `OUT_00` to `OUT_03`. The event `CNF` is then sent.

Technical Features

- **Byte Order:** Conversion always occurs in little-endian format, meaning the first byte in the array becomes the least significant part of the word.
- **No Buffering:** The function block is purely combinational and does not store any states. Each `REQ` call processes the current input value.
- **Typing:** Inputs are defined as an array of `BYTE` (0 indexing), outputs as `WORD`. The function block does not require any further type conversions.

State Overview

The function block does not have an internal state machine. Processing is strictly event-driven:

1. Wait for `REQ`.
2. For `REQ`: Copy the bytes into the output words.
3. Send `CNF` and return to wait mode.

Application Scenarios

- **Sensor Readout:** A temperature sensor provides 8 raw bytes, which must be split into four 16-bit measured values (e.g., two channels with one word each).
- **Protocol Decoding:** A serial protocol transmits user data as a byte array; the function block extracts four 16-bit register values from it.
- **Data Preprocessing in PLC Systems:** Combine multiple byte-oriented interfaces (e.g., Profibus, CAN) into uniform word signals for further processing.

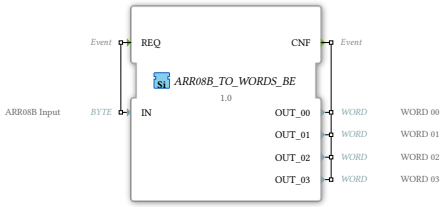
Comparison with Similar Function Blocks

- **`BYTE_TO_WORD`:** Converts only two bytes into a single word; requires four calls for 8 bytes.
- **`ARRAY_TO_WORD` (general):** Usually configurable, e.g., with start index and number of words – `ARR08B_TO_WORDS`, on the other hand, is fixed to 8 bytes and 4 words and is therefore simpler and more performant in this specific case.
- **`WORD_TO_ARR08B` (if present):** The inverse function that packs four words back into an 8-byte array.

Conclusion

The function block **ARR08B_TO_WORDS** offers a lean, specialized solution for decomposing an 8-byte array into four 16-bit words. The hardwired byte mapping ensures high execution speed and ease of use, especially in time-critical environments or for recurring conversion tasks. Little-endian interpretation is the standard for many industrial and embedded systems, so the component requires no additional parameters.

ARR08B_TO_WORDS_BE



• — ○ • — ○ • — ○ •

Introduction

The function block `ARR08B_TO_WORDS_BE` extracts four 16-bit words (WORD) from an 8-byte array (big-endian). It is primarily used to convert binary data streams stored in an ascending byte array and interpret them as an ordered word sequence. This block belongs to the package group `logiBUS::utils::conversion::arr::forwarding`.

Interface Structure

Event Inputs

Event	Description

REQ	Starts the conversion: The incoming byte array IN is converted into the four output words.

Event Outputs

Event	Description
CNF	Confirms successful conversion and indicates that the output values are valid.

Data Inputs

Name	Type	Array Size	Description
IN	BYTE	0..7 (8 bytes)	Input array – the raw data from which the words are extracted.

Data Outputs

Name	Type	Description
OUT_00	WORD	Word from byte 0 (high) and byte 1 (low)
OUT_01	WORD	Word from byte 2 (high) and byte 3 (low)
OUT_02	WORD	Word consisting of byte 4 (high) and byte 5 (low)
OUT_03	WORD	Word consisting of byte 6 (high) and byte 7 (low)

Adapter

No adapters available.

Functionality

Upon arrival of a REQ event, the eight bytes of the input array IN are encoded into four 16-bit words in strict **big-endian order**:

- OUT_00 = (IN[0] << 8) | IN[1]
- OUT_01 = (IN[2] << 8) | IN[3]
- OUT_02 = (IN[4] << 8) | IN[5]
- OUT_03 = (IN[6] << 8) | IN[7]

The output event is then The event CNF is triggered, indicating the validity of all output values. This is implemented directly in the structured text portion of the function block, without an internal state machine.

Technical Features

- **Big-Endian Convention:** The first byte of a word (IN[0], IN[2], ...) is the higher-order byte (%B1), and the second byte (IN[1], IN[3], ...) is the lower-order byte (%B0).
- **Fixed Array Size:** Exactly 8 bytes are expected; the array is declared as ARRAY[0..7] OF BYTE.
- **No side effects:** The function block is purely combinatorial – it requires no internal state and operates within a single event cycle.
- **Typical use:** Converting data from binary protocols (e.g., CAN messages, Modbus registers, or serial streams) that are stored as byte arrays.

State overview

The function block does not have its own state machine. It is triggered by the **event REQ** and outputs **CNF** after data processing. Execution occurs within a single cycle.

State	Description
Idle	Waiting for REQ; outputs retain the last value.
Processing	REQ received – data is being processed.
Completed	CNF is being sent, outputs are valid. The module immediately returns to idle state.

Application Scenarios

1. **Modbus Communication:** An 8-byte register block (e.g., holding register) is received as a byte array and must be split into four 16-bit values.
2. **CAN Messages:** An 8-byte CAN frame contains multiple 16-bit sensor values transmitted in big-endian format.
3. **Serial Data Analysis:** Summarizing consecutive bytes from a serial interface into usable WORD sizes for further processing.
4. **Gateway Functions:** Conversion of binary to general-purpose data in automation processes based on IEC 61499.

Comparison with Similar Building Blocks

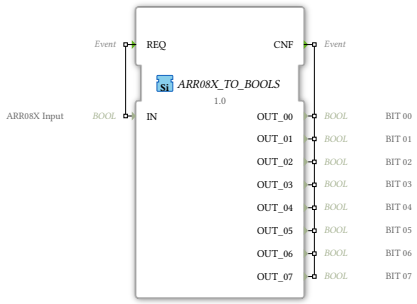
Building Block	Difference
`ARR08B_TO_WORDS_LE`	Uses little-endian order (Byte[0] = low byte).
`ARR04B_TO_WORD`	Works with a 4-byte array and creates only one WORD.
`BYTE_TO_WORD`	Converts two individual bytes into one WORD; requires separate concatenation.

ARR08B_TO_WORDS_BE` focuses on the efficient, predefined partitioning of an 8-byte array into four words in big-endian order – ideal for standardized protocols.

Conclusion

The `ARR08B_TO_WORDS_BE` function block offers a simple and reliable method for converting an 8-byte array into four big-endian words. Its compact interface (`REQ`/`CNF` + one input, four outputs) allows for easy integration into existing IEC 61499 applications. Thanks to its direct ST implementation, there are no latencies from state machines, making it particularly suitable for time-critical applications in automation technology.

ARR08X_TO_BOOLS



• – ○ • – ○ • – ○ •

Introduction

The **ARR08X_TO_BOOLS** function block converts an 8-element BOOL array into eight individual BOOL signals. It extracts each bit from the input array and makes it available via its own named data output. This block is particularly useful for breaking down compact array data from higher-level systems or communication interfaces into individual, easily processed logical signals.

Interface Structure

The block has a simple, unidirectional interface with a request/acknowledgment event pair.

Event Inputs

REQ: This event triggers the processing. Upon its arrival, the values of the input array `IN` are read and distributed to the individual outputs `OUT_00` to `OUT_07`.

Event Outputs

- **CNF**: This event is triggered as soon as the extraction of the Boolean values is complete. It confirms the execution of the function block and transmits the new values to all eight data outputs (`OUT_00` to `OUT_07`).

Data Inputs

- **IN** (ARRAY[0..7] OF BOOL): The input array containing the eight Boolean values to be extracted. Index 0 corresponds to the least significant bit (LSB) and is mapped to `OUT_00`.

Data Outputs

- **OUT_00** (BOOL): Extracted bit 0 (LSB) of input array `IN`.
- **OUT_01** (BOOL): Extracted bit 1 of input array `IN`.
- **OUT_02** (BOOL): Extracted bit 2 of input array `IN`.
- **OUT_03** (BOOL): Extracted bit 3 of input array `IN`.
- **OUT_04** (BOOL): Extracted bit 4 of input array `IN`.
- **OUT_05** (BOOL): Extracted bit 5 of the input array `IN`.
- **OUT_06** (BOOL): Extracted bit 6 of the input array `IN`.
- **OUT_07** (BOOL): Extracted bit 7 (MSB) of the input array `IN`.

Adapters

This function block does not use any adapters.

Operation

The operation is deterministic and stateless. Upon each occurrence of the `REQ` event, the function block performs a direct assignment. Each element of the input array `IN` is assigned to its corresponding, individually named output variable. The logic corresponds to the following ST assignment:
```
  iec61131-3 OUT_00 := IN[0]; OUT_01 := IN[1]; OUT_02 := IN[2]; OUT_03 := IN[3]; OUT_04 := IN[4]; OUT_05 := IN[5]; OUT_06 := IN[6]; OUT_07 := IN[7]; Immediately after this operation is completed, the CNF`` event is generated with the new values.
```

Technical Features

- **Real-time behavior:** The function block only performs assignments and causes no significant latency. Execution is completed in one cycle.
- **Memory:** No internal state is stored between calls. The output is a pure function of the current input.
- **Array limits:** The function block is specifically defined for arrays of a fixed size of 8 (indices 0 to 7). Using an array of a different size will result in a compilation error.

State overview

The function block has no internal state (stateless). Its behavior is identical for each call and depends solely on the current values at the data input `IN`.

Application Scenarios

- **Bus Telegram Unpacking:** Extracting individual status or control bits from a byte received via field-buses (e.g., CAN, PROFIBUS) or industrial networks.
- **Interface to Bit-Oriented Components:** Preparing array data for controlling individual lamps, valves, or relays addressed via discrete BOOL outputs.
- **Simplified Visualization:** Providing individually addressable signals for HMIs or alarm systems instead of having to work with array indices.
- **Decoding:** Converting a simple 8-bit binary number into its individual logical values.

⚖️ Comparison with Similar Function Blocks

- **`SEL` / `MUX` Function Blocks:** These select a single element from an array based on an index. `ARR08X_TO_BOOLS`, on the other hand, extracts **all** elements simultaneously and in parallel.
- **`BYTE_TO_BOOLS` / `WORD_TO_BOOLS`:** These building blocks decompose a byte or word data element into its individual bits. `ARR08X_TO_BOOLS` operates directly on an array of BOOLs, which is semantically clearer if the source is already a BOOL array. However, the functional effect on the bits is equivalent.
- **Manual Indexing:** Instead of using this building block, one could directly access `IN[0]`, `IN[1]`, etc., in the surrounding algorithm. However, the function block provides a clearer interface description, improves the readability of the FBD network, and encapsulates the functionality.

🛠️ Related Exercises

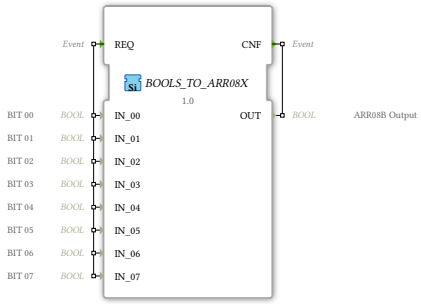
- Exercise_054

Conclusion

The ARR08X_TO_BOOLS is a specialized, yet very useful conversion module. It provides a simple and efficient method for unpacking a compact 8-bit BOOL array into eight discrete signals. Its clear, stateless operation and the explicit labeling of all outputs significantly improve the readability and maintainability of control applications that use array-based data structures.

reversing

BOOLS_TO_ARR08X



• — ○ • — ○ • — ○ •

Introduction

The function block BOOLS_TO_ARR08X converts eight individual Boolean (BOOL) input signals into a Boolean array (ARRAY[0..7] OF BOOL). It is a fundamental building block for signal aggregation and data structuring in control applications, especially when discrete digital signals need to be bundled for compact processing or transmission.

Interface Structure

Event Inputs

- REQ (Request): Triggers the conversion operation. Upon this event, the current values at the eight IN_xx inputs are read and written to the output array OUT.

Event Outputs

- CNF (Confirm): Triggered as soon as the conversion is complete and the output array OUT is ready with the new values.

Data Inputs

Eight individual Boolean inputs (BOOL), each corresponding to a bit in the resulting array:

- IN_00: Value for bit 00 / Array index 0.
- IN_01: Value for bit 01 / Array index 1.
- IN_02: Value for bit 02 / Array index 2.
- IN_03: Value for bit 03 / Array index 3.
- IN_04: Value for bit 04 / Array index 4.
- IN_05: Value for bit 05 / Array index 5.
- IN_06: Value for bit 6 / array index 6.
- IN_07: Value for bit 7 / array index 7.

Data Outputs

- OUT: The output signal of type ARRAY[0..7] OF BOOL. After the REQ event is triggered, it contains the values present at the inputs at that time, where IN_00 is mapped to OUT[0] and IN_07 to OUT[7].

Adapters

This function block has no adapter interfaces.

Functionality

The functionality is deterministic and simple. Whenever an event arrives at the `REQ` input, the contained ST code is executed. This code directly assigns the value of the corresponding individual `IN_xx` input to each element of the output array `OUT`. The mapping is hardwired:
OUT[0] := IN_00; `OUT[1] := IN_01;` ... OUT[7] := IN_07;`Immediately after this assignment, the`CNF` event is generated to signal the successful completion of the operation. There is no buffering, delay, or logical combination of the input values.

Technical Features

- **Real-time Behavior:** The operation is atomic and very fast, as it consists only of direct assignments. It introduces no significant latency into the control cycle.
- **Size Definition:** The output array is defined with the fixed size `[0..7]`, which corresponds to exactly eight elements.
- **Synchronous Processing:** All input values are acquired synchronously at the time of the `REQ` event. Changes to the inputs during processing have no effect on the current cycle.

State Overview

The function block is stateless. It has no internal memory. Its output at any given time is solely a function of the current input values at the time of the last `REQ` event.

Application Scenarios

- **Bundling of Switching States:** Combining the states of multiple limit switches, pushbuttons, or valves into a single array for transmission over a network or to a higher-level control module.
- **Preprocessing for Array-Based Blocks:** Providing input data for function blocks that perform their operations on entire arrays (e.g., logic operations, shift operations).
- **Structuring of I/O Data:** Organizing distributed digital input signals from a PLC into a defined data structure for improved program readability and maintainability.
- **Interface to Visualization Systems:** Easier mapping of many individual signals to a composite symbol or status display in an HMI.

⚖️ Comparison with Similar Blocks

- **`BOOL_TO_xxx` Blocks:** Blocks like `BOOL_TO_INT` or `BOOL_TO_BYTE` also convert multiple Boolean values into a more compact data set. `BOOLS_TO_ARR08X`, on the other hand, explicitly creates an array, which preserves indexed access to each individual bit and maintains the semantic meaning of its position.
- **Manual Wiring:** The alternative would be to directly wire each individual bit to the indices of an array variable block. `BOOLS_TO_ARR08X` encapsulates this operation, improving clarity in the FBD/LD editor and reducing wiring effort.
- **`MERGE`/`CONCAT` blocks:** General-purpose merge blocks can combine various data types. `BOOLS_TO_ARR08X` is specialized and type-safe for exactly eight BOOL inputs and a BOOL array as an output.

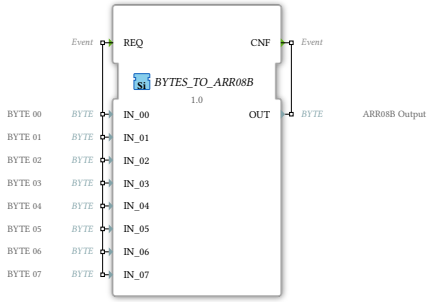
🛠️ Related Exercises

- Exercise_054

Conclusion

The `BOOLS_TO_ARR08X` is a specialized, robust, and efficient function block for a common task in industrial automation: bundling discrete signals. Its strength lies in its simplicity, deterministic execution time, and clear interface definition. It is the first choice when eight Boolean signals need to be provided in an ordered array structure for further processing.

BYTES_TO_ARR08B



• — ◦ • — ◦ • — ◦ •

Introduction

The function block `BYTES_TO_ARR08B` converts eight individual `BYTE` values into an array of type `BYTE` with eight elements. It is part of a collection of helper functions for data conversion and array operations.

Interface Structure

Event Inputs

- **REQ**: This event triggers the processing of the function block. Upon its arrival, the values at the data inputs are read and written to the output array.

Event Outputs

- **CNF**: This event is generated after successful processing and assignment of the values to the output array. It indicates the availability of the result. ...

Data Inputs

- **IN_00** (`BYTE`): Value for the array element at index 0.
- **IN_01** (`BYTE`): Value for the array element at index 1.
- **IN_02** (`BYTE`): Value for the array element at index 2.
- **IN_03** (`BYTE`): Value for the array element at index 3.
- **IN_04** (`BYTE`): Value for the array element at index 4.
- **IN_05** (`BYTE`): Value for the array element at index 5.
- **IN_06** (`BYTE`): Value for the Array element at index 6.
- **IN_07** (`BYTE`): Value for the array element at index 7.

Data Outputs

- **OUT** (`ARRAY[0..7] OF BYTE`): The resulting byte array. Indices 0 through 7 contain the values from `IN_00` to `IN_07`, sequentially.

Adapters

This function block does not use any adapters.

Operation

When triggered by the `REQ` event, the block copies the values of the eight `BYTE` inputs directly into the corresponding positions of the output array `OUT`. The assignment is sequential: `IN_00` ⟶ `OUT[0]`, `IN_01` ⟶ `OUT[1]`, and so on, up to `IN_07` ⟶ `OUT[7]`. Immediately after this assignment, the `CNF` event is triggered.

The internal implementation is in Structured Text (ST) and consists of a simple series of assignments.

Technical Features

- **Deterministic Runtime:** The operation has a constant and very short execution time because it consists only of direct assignments.
- **No Buffering:** There is no intermediate storage of the input values. Processing occurs exclusively upon the arrival of a `REQ` event.
- **Fixed Size:** The block is specifically designed for a combination of exactly eight bytes.

State Overview

The function block is stateless. Each `REQ` call is independent of previous calls and processes only the currently available input values.

Application Scenarios

- **Data Aggregation:** Combining individual byte values (e.g., from various sources or sensors) into a compact array for further processing or transmission.
- **Protocol and Interface Adaptation:** Preparing data structures for communication protocols or function blocks that expect a byte array instead of individual bytes.
- **Data Restructuring:** Converting between different data representations within a control application.

⚖️ Comparison with Similar Blocks

- **`CONCAT` Blocks:** General-purpose blocks for concatenating strings are not directly comparable, as they are designed for the `STRING` data type.
- **Manual Assignment in ST:** The functionality could be implemented directly in an ST program using assignments. However, the function block encapsulates this logic for improved reusability, readability, and maintainability within FBD/LD diagrams.
- **Custom Array Blocks:** Unlike general-purpose array blocks that have index inputs, `BYTES_TO_ARR08B` is optimized for a specific, fixed task and is therefore easier and faster to use.
-

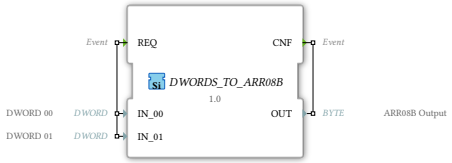
🛠️ Related Exercises

- Exercise_126b_sub
- Exercise_12x_sub

Conclusion

The `BYTES_TO_ARR08B` is a specialized and efficient utility module for a common data manipulation task. It offers a clear, event-driven interface and saves the user from repeatedly implementing array mapping manually. Its strength lies in its simplicity and unambiguity for the specific use case of combining eight bytes.

DWORDS_TO_ARR08B



• — ○ • — ○ • — ○ •

Introduction

The function block **DWORDS_TO_ARR08B** converts two 32-bit DWORD values into an eight-byte array (BYTE). The conversion is performed in little-endian format (Intel standard), meaning the least significant byte of a DWORD is written to the lowest index position of the output array. The function block is optimized for use on PC-based controllers (e.g., Beckhoff, Wago, B&R, Raspberry Pi).

Interface Structure

Event Inputs

Event	Description
`REQ`	Starts the conversion. The current values of `IN_00` and `IN_01` are read.

Event Outputs

Event	Description

CNF	Sent as soon as the output OUT has been updated.

Data Inputs

Variable	Type	Description
IN_00	DWORD	First 32-bit word (lower order).
IN_01	DWORD	Second 32-bit word (higher order).

Data Outputs

Variable	Type	Description
OUT	ARRAY[0..7] OF BYTE	Output array with 8 bytes (bytes 0–3 from IN_00, bytes 4–7 from IN_01).

Adapters

No adapters available.

Functionality

When a rising edge is detected at the event input REQ, the following logic is executed:

1. The two input DWORDs IN_00 and IN_01 are read byte by byte.
2. The system's internal byte accesses (e.g., IN_00.%B0 for the least significant byte) are used.
3. The bytes are copied into the output array OUT in **little-endian order**:

- OUT[0] ⟵ least significant byte of IN_00
- OUT[1] ⟵ second byte of IN_00
- OUT[2] ⟵ third byte of IN_00
- OUT[3] ⟵ most significant byte of IN_00
- OUT[4] ⟵ least significant byte of IN_01
- OUT[5] ⟵ second byte of IN_01
- OUT[6] ⟵ third byte of IN_01
- OUT[7] ⟵ most significant byte of IN_01

1. After the assignments are complete, the event CNF is sent.

Technical Details

- The byte order follows the **Little-Endian format** (Intel standard), which is natively used on most PC-based controllers (Windows-based systems, Raspberry Pi).
- The function block is implemented as **structured text (ST)** and does not require a state machine – processing occurs linearly with each REQ event.
- The package logiBUS::utils::conversion::arr::reversing used in the ST code indicates a reusable library for conversions.

State Overview

The function block does not have its own state machine. It performs a single, deterministic calculation with each REQ event and signals completion with CNF. There are no internal memory states that persist across multiple calls.

Application Scenarios

- **Data Protocol Preparation**: Conversion of control data (e.g., position values, timer counters) stored as two DWORDs into a byte array for serial transmission (RS-232, CAN, Ethernet).
- **Communication with Endian-Sensitive Devices**: Ensuring the correct byte order when exchanging data with sensors or actuators that expect little-endian.
- **Memory Mapping / Union Replacement**: In languages without union types, this function block serves as a portable method for decomposing DWORD values byte by byte.

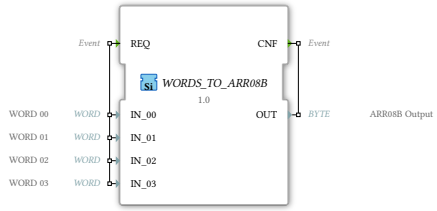
Comparison with Similar Function Blocks

- **WORD_TO_ARR04B**: Converts a WORD into a 4-byte array – here, the same approach is extended to two DWORDs.
- **DWORDS_TO_ARR08B_BIGENDIAN**: Alternative function block for big-endian systems (Motorola standard) with reversed byte order.
- **SWAP_BYTES**: Swaps the bytes of a DWORD, but does not return an array. This function block directly creates the complete 8-byte array.

Conclusion

The function block `DWORDS_TO_ARR08B` offers a simple and efficient way to combine two little-endian DWORDs into a byte array. Its consistent focus on PC-based controllers and its clear, maintainable ST implementation make it a reliable tool for data conversion in automation technology.

WORDS_TO_ARR08B



• – ○ • – ○ • – ○ •

Introduction

The function block **WORDS_TO_ARR08B** is used to concatenate four 16-bit words (type `WORD`) into a byte array of length 8 (type `ARRAY[0..7] OF BYTE`). It uses the little-endian format (Intel standard), which stores the least significant bytes first. The block is controlled via an event input and is particularly suitable for PC-based controllers (e.g., Beckhoff, Wago, B&R, Raspberry Pi).

Interface Structure

Event Inputs

Name	Type	Description
`REQ`	Event	Triggers the conversion; all input variables are read and the result is calculated.

Event Outputs

Name	Type	Description
`CNF`	Event	Sent after successful calculation, indicating that valid data is present at output `OUT`.

Data Inputs

Name	Type	Description
`IN_00`	WORD	First Word (Bytes 0 and 1)
`IN_01`	WORD	Second Word (Bytes 2 and 3)
`IN_02`	WORD	Third Word (Bytes 4 and 5)
`IN_03`	WORD	Fourth Word (Bytes 6 and 7)

Data Outputs

Name	Type	Description
`OUT`	ARRAY[0..7] OF BYTE	Result array, containing the 8 bytes in little-endian order.

Adapters

No adapters defined.

Functionality

When a rising edge is detected at the event input `REQ`, the function block performs the following mapping (described in the Structured Text Implementation):

Index `OUT`	Source	Meaning
`OUT[0]`	`IN_00.%B0`	Low-order byte of word 0
`OUT[1]`	`IN_00.%B1`	High-order byte of word 0
`OUT[2]`	`IN_01.%B0`	Low-order byte of word 1
`OUT[3]`	`IN_01.%B1`	Higher-order byte of word 1
`OUT[4]`	`IN_02.%B0`	Low-order byte of word 2
`OUT[5]`	`IN_02.%B1`	Higher-order byte of word 2
`OUT[6]	`IN_03.%B0`	Low-order byte of word 3
`OUT[7]`	`IN_03.%B1`	Higher-order byte of word 3

After the assignments are complete, the output pulse `CNF` is generated.

Technical Features

- **Little-Endian Format**: The bytes of each word are stored in ascending memory address order, with the least significant byte first. This corresponds to the Intel byte order and is common for many PC-based controllers and communication protocols (e.g., Modbus, CANopen, serial interfaces).
- **No Reversal of Overall Order**: The words themselves are inserted into the array in the order `IN_00` to `IN_03`; only within each word is the byte order reversed. If a different implementation is required to also reverse the word order (e.g., Big-Endian at the word level), this is necessary.
- **Immediate Execution**: The function block is purely combinatorial – after the event `REQ`, the result is available in the next execution cycle. No internal state is stored.

State Overview

The function block does not have an explicit state machine. Its behavior is purely event-driven:

1. **Idle**: Waiting for `REQ`.
2. **Processing**: Upon `REQ`, the input values are read and the output calculation is performed.
3. **Complete**: `CNF` is sent; the function block returns to the idle state.

The implementation is deterministic and requires only one execution step.

Application Scenarios

- **Communication with Peripherals**: A microcontroller or IPC sends 16-bit sensor data (e.g., analog-to-digital converter, encoder) via a byte-oriented serial bus. The device packs the words into a byte array that can be transmitted directly.
- **Data Logging**: Combining multiple 16-bit measurements into a buffer for storage or transmission (e.g., CSV, binary format).
- **Protocol Conversion**: Adapting data to the byte order required by a specific protocol (e.g., OPC UA, MQTT binary payload).

Comparison with Similar Function Blocks

- **WORDS_TO_ARR08B** expects exactly four words and creates an array of length 8. For other word/byte counts, related function blocks exist, such as `WORDS_TO_ARR04B`, `WORDS_TO_ARR16B`, or for big-endian.
- Unlike a general-purpose block like `MERGE`, which concatenates arbitrary data types, this function block is specifically optimized for 2-byte-per-word conversion and guarantees the correct byte order.
- Compared to simple variable assignment using `MOVE` or a pure function like `WORD_TO_BYTE`, this block avoids manual index calculations and improves readability.

Conclusion

The function block **WORDS_TO_ARR08B** is a compact, specialized module for converting four 16-bit words into a little-endian byte array. Its clear, event-driven interface and simple implementation make it ideal for use in control applications where data needs to be prepared at the byte level for communication or storage. It belongs to the `logiBUS::utils::conversion::arr::reversing` package family and adds a useful conversion capability to the 4diac toolbox.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 MCU vs. MPU Comparison Guide on ms-muc-docs.de

]

WORDS_TO_ARR08B_BE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block **WORDS_TO_ARR08B_BE** combines four 16-bit words (WORD) into an 8-byte array. The conversion is performed in **Big-Endian format** (Network Byte Order), where the most significant byte is stored first in the output array. This corresponds to the standard representation in network protocols such as Modbus TCP, Siemens S7 communication, or Ethernet headers.

Interface Structure

Event Inputs

Event	Description
`REQ`	Starts the conversion of all four input words.

Event Outputs

Event	Description
`CNF`	Sent once the output array has been fully calculated.

Data Inputs

Name	Type	Comment
`IN_00`	WORD	Word 00 (first word)
`IN_01`	WORD	Word 01 (second word)
`IN_02`	WORD	Word 02 (third word)
`IN_03`	WORD	Word 03 (fourth word)

Data Outputs

Name	Type	Comment
`OUT`	ARRAY[0..7] OF BYTE	8-byte array in big-endian format

Adapters

No adapters available.

Functionality

After receiving the event `REQ`, the values of the four inputs `IN_00` to `IN_03` are read. For each word, the **high byte** (BYTE 1) and then the **low byte** (BYTE 0) are written to the output array. The order in the array is:

- `OUT[0]` = High byte of `IN_00`
- `OUT[1]` = Low byte of `IN_00`
- `OUT[2]` = High byte of `IN_01`
- `OUT[3]` = Low byte of `IN_01`
- `OUT[4]` = High byte of `IN_02`
- `OUT[5]` = Low byte of `IN_02`
- `OUT[6]` = High byte of `IN_03`
- `OUT[7]` = Low byte of `IN_03`

The event `CNF` is then triggered.

Technical Features

- **Big-Endian Format** (Network Byte Order) – the bytes are stored in order, with the most significant byte first. This is required for many industrial communication protocols and network stacks.
- The function block operates **event-driven**: The conversion is only performed when a `REQ` event occurs, so no cyclical calculation takes place.
- The internal implementation uses **Structured Text (ST)** with direct bit element access (`%B1` and `%B0`), ensuring efficient and deterministic conversion.

State Overview

The function block has no explicit internal state (stateless). It immediately performs the calculation upon each `REQ` and signals completion with `CNF`. No blocking is expected during the calculation.

Application Scenarios

- **Network Communication**: Preparation of data for protocols that expect big-endian, e.g., Modbus TCP, Siemens S7 over Industrial Ethernet, or raw Ethernet frames.
- **Data Preprocessing**: Conversion of sensor values or control commands stored as WORD registers into a byte-oriented format.
- **Gateway Functions**: Translation between little-endian controllers and big-endian networks.

Comparison with Similar Components

Component	Special Feature
WORDS_TO_ARR08B_BE	Big-endian, 4 WORDS ⟶ 8 bytes
WORDS_TO_ARR08B_LE	Little-endian (low byte first)
BYTES_TO_WORDS_BE / LE	Reverse conversion
WORDS_TO_ARR16B_BE	Big-Endian, 8 WORDS ⟶ 16 Bytes
WORDS_TO_ARR08N	Possibly unspecified endianness

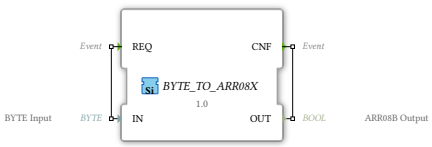
This function block is specifically optimized for big-endian applications and is kept compact.

Conclusion

WORDS_TO_ARR08B_BE` is a precise, event-driven function block for the big-endian conversion of four WORDS into an 8-byte array. It is ideally suited for all applications that require byte-wise representation in Network Byte Order and simplifies the interface between control logic and communication protocols.

splitting

BYTE_TO_ARR08X



• – ◦ • – ◦ • – ◦ •

Introduction

The function block BYTE_TO_ARR08X converts a single byte value into an array of eight Boolean values (BOOL). Each bit of the input byte is assigned to a separate element in the output array. This block is particularly useful for breaking down compact digital information stored in a byte for individual processing or visualization.

Interface Structure

Event Inputs

- REQ (Request): This event triggers the conversion function. Upon its arrival, the current value at data input IN is processed.

Event Outputs

- CNF (Confirm): This event signals the completion of the conversion. It is output along with the calculated data at output OUT.

Data Inputs

- IN (BYTE): The input for the byte value to be decomposed into Boolean bits.

Data Outputs

- OUT (ARRAY[0..7] OF BOOL): The output array with eight BOOL elements. Element OUT[0] corresponds to the least significant bit (LSB, bit 0) of the input byte, and OUT[7] corresponds to the most significant bit (MSB, bit 7).

Adapter

This function block has no adapter interfaces.

Functionality

When triggered by the REQ event, the block reads the value at the IN input. Then, each individual bit (0 to 7) of the BYTE data type is extracted and assigned to the corresponding index in the BOOL array OUT. The assignment is performed directly:

- OUT[0] receives the value of IN.0 (bit 0)
- OUT[1] receives the value of IN.1 (bit 1)
- ...
- OUT[7] receives the value of IN.7 (bit 7)

After these assignments are complete, the CNF event is output along with the populated array OUT.

- OUT[7] receives the value of IN.7 (bit 7)
- OUT[0] receives the value of IN.0
- OUT[1] receives the value of IN.1
- OUT[1] receives the value of IN.1 ...

Technical Features

- **Direct Bit Access:** The implementation uses the direct bit addressing common in IEC 61131-3 ST (Structured Text) syntax (e.g., IN.0), enabling efficient and readable operation.
- **Fixed Array Size:** The output array is defined with the fixed size [0..7], which corresponds exactly to the bit length of a byte (8 bits).
- **Real-Time Capable:** The block performs a deterministic, stateless operation and is therefore suitable for real-time applications.

State Overview

The function block is stateless. Its behavior is identical for every REQ event and depends solely on the current value at the data input IN. There are no internal variables stored between calls.

Application Scenarios

- **Control Byte Unpacking:** Processing a status or control byte received from a field device or higher-level controller to evaluate individual status bits (e.g., “Motor on,” “Valve open,” “Fault active”) separately.
- **Visualization:** Preparing data for visualization systems (HMIs), which often display individual switching states as separate Boolean lamps or indicators.
- **Interface to Bit-Oriented Logic:** Providing individual bits for subsequent function blocks that operate with Boolean logic (AND, OR, NOT, flip-flops) instead of entire bytes.

⚖️ Comparison with Similar Blocks

- **`BYTE_TO_BOOL` (or similarly named blocks):** Many libraries offer blocks that convert one byte into eight separate BOOL outputs. `BYTE_TO_ARR08X` differs in that it outputs the bits in a structured **array**, which simplifies the wiring in the 4diac IDE, as only one data output is required. Handling in ST code is also more compact.
- **Bitwise reading using `AND` & comparisons:** The same functionality could be achieved through a series of comparisons (`((IN AND 16#01) = 1`, etc.). The `BYTE_TO_ARR08X` block encapsulates this logic in a reusable, easy-to-understand block.
- **Counterpart `ARR08X_TO_BYTE`:** A complementary block that combines an array of eight BOOLs into one byte would perform the reverse operation.

Conclusion

The `BYTE_TO_ARR08X` function block is a simple, specialized, and highly practical conversion module. It elegantly solves a common problem in automation technology by abstracting bit extraction from a byte and providing the results in a manageable array. Its deterministic and stateless nature makes it a reliable and high-performance component for applications where digital bit information needs to be transferred compactly and then processed decentrally.

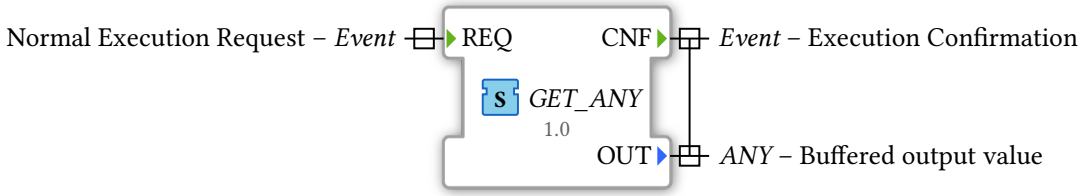
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Numeric Index (conversion)

Here you will find all exercises from the `conversion` workspace, listed in ascending order. …

GET_ANY



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_ANY** function block reads an arbitrary value (type `ANY`) from an InOut variable and provides it as a buffered output value. It serves to capture the current value of a variable source at a specific point in time and temporarily store it for subsequent processing steps.

Interface Structure

Event Inputs

Name	Type	With Variables	Description
REQ	Event	IN	Normal execution request; reads the current value of IN.

Event Outputs

Name	Type	With Variables	Description
CNF	Event	IN, OUT	Confirmation after successful execution; the values of IN and OUT are then valid.

Data Inputs

The function block has no separate data inputs. The source data is provided via the InOut variable.

Data Outputs

Name	Type	Description
OUT	ANY	Buffered output value. Contains the value read from IN at the time of the last REQ execution.

Adapter

None.

Functionality

1. An event at **REQ** triggers the internal algorithm `REQ`.
2. The algorithm executes the assignment `OUT := IN`.

- **IN** is an InOut variable that can be both read and written – here it is used only as a source.
- **OUT** receives the current value of **IN** and stores it temporarily.

1. After the assignment is complete, the **CNF** event is sent. At this point, both **IN** (unchanged) and **OUT** are valid.

Technical Features

- The function block uses an **InOut variable** (`IN`) as its source. This allows direct access to another block output or a global variable without requiring a separate connection via a data input.

IN receives the current value of **IN** and stores it temporarily. - The **type `ANY`** makes the function block generic; it can be used with any data type (e.g., INT, REAL, BOOL, STRING).

- There is only one state, therefore execution is deterministic and occurs in a single step.

State Overview

The function block is implemented as a **SimpleFB** with exactly one state:

State	Description
REQ	Executes the algorithm `REQ` and sends CNF. After the action, the function block returns to this state (default).

No other states (e.g., Wait, Error) are defined.

Application Scenarios

- **Value Sampling**: At a specific point in time (event), a current value from a sensor or calculation should be captured, e.g., for later logging or comparison.
- **Signal Decoupling**: If an output needs to be read multiple times without the value changing during processing, GET_ANY buffers the value.
- **Type-Independent Interface**: As a generic buffer in libraries that work with different data types.

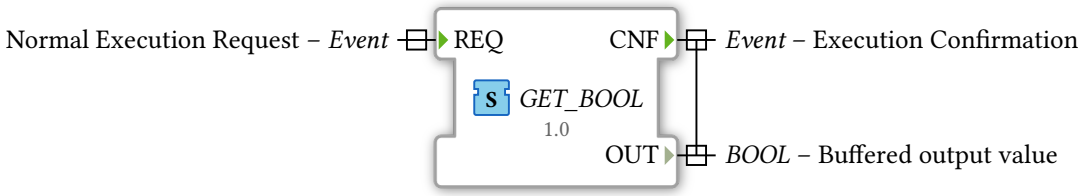
Comparison with Similar Function Blocks

- **`GET` Function Blocks** (e.g., `GET_INT`): These are type-specific and require explicit data types. In contrast, `GET_ANY` offers **type flexibility** at the expense of type safety at design time.
- **`MOVE`-FB**: A `MOVE` function block typically copies from an input to an output but does not use an InOut variable. `GET_ANY`, on the other hand, reads directly from an existing variable without requiring a separate input connection.

Conclusion

GET_ANY` is a simple yet powerful function block for **caching any value** from an InOut source. Its generic type design and direct connection to existing variables reduce wiring complexity and increase flexibility in the control logic. It is particularly suitable for applications where a value needs to be frozen at a specific event time.

GET_BOOL



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_BOOL** function block reads a BOOL value from a variable declared as `InOut` and provides it as a buffered output value. This simple encapsulation allows a Boolean signal to be passed from any context without requiring direct read/write access to the source in the calling function block.

Interface Structure

Event Inputs

Event	Description	With Variables
REQ	Normal Execution Request	IN

Event Outputs

Event	Description	With Variables
CNF	Execution Confirmation	OUT, IN

Data Inputs

There are no declared data inputs (*InputVars*). Access to the source variable is exclusively via an InOut parameter.

Data Outputs

Variable	Type	Initial Value	Description
OUT	BOOL	FALSE	Buffered Output Value

Adapters

No adapters are available.

Operation

1. Processing is started at the **event input REQ**.
2. The internal algorithm **REQ** performs the simple assignment `OUT := IN` – the current value of the InOut variable `IN` is copied to the output `OUT`.
3. After the successful copy operation, the **CNF** event is triggered. It passes both the new value `OUT` and the (unchanged) value of `IN` to the subsequent logic.

The output `OUT` remains at the last read value until a new pulse `REQ` updates the value.

Technical Features

- **In/Out Parameter** `IN`: This parameter allows bidirectional access to the connected variable. The function block reads the value but does not modify it. This allows direct access to a variable from the parent application without having to define a separate input variable.
- **Buffering**: The value is buffered locally in the output `OUT`. Changes to the source variable between two `REQ` calls are only applied with the next `REQ` call.
- **Initial Values**: Both `IN` and `OUT` are pre-assigned to `FALSE`, so the function block exhibits defined behavior even without prior initialization.

Buffering:**

State Overview

The function block has a single state:

State	Action	Output Event
REQ	`OUT := IN`	CNF

After the algorithm is executed, the initial state is immediately exited (no hold state). The function block is therefore event-driven and waits for the next request after processing.

Application Scenarios

- **Signal Acquisition**: A Boolean sensor value (e.g., from a switch or limit switch) is to be used in the application and simultaneously passed to a higher-level display function block.
- **Decoupling**: A variable from another function block is to be read without the source function block establishing a direct connection. `GET_BOOL` handles the reading.
- **Buffering and Synchronization**: In clocked systems, a `REQ` pulse can be used to sample the current value and hold it for the duration of a cycle.

Comparison with Similar Function Blocks

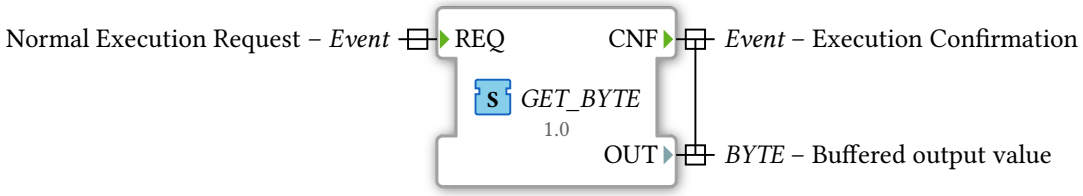
Function Block	Purpose	Data Type
GET_BOOL	Read a BOOL value via InOut	BOOL
GET_INT	Read an INT value via InOut	INT
GET_REAL	Read a REAL value via InOut	REAL
SET_BOOL	Write a BOOL value via InOut (write)	BOOL

All `GET_*` function blocks share the use of an InOut parameter and event-driven value transfer. The only difference lies in the data type used. Unlike `SET_BOOL`, `GET_BOOL` does not modify the source variable.

Conclusion

The **GET_BOOL** block is a minimalist yet useful component for reading and buffering a single Boolean value. Its strength lies in the clean decoupling of the signal source from the rest of the application code and its ease of use thanks to a single event pulse. It is particularly well-suited for modular and reusable automation solutions that require a clearly defined data flow.

GET_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_BYTE** function block is used to read a variable `BYTE`, which is provided via an **InOut** parameter. The read value is buffered at the output `OUT` and is available after execution. This function block is typically used to retrieve a current `BYTE` value from an external source (e.g., a shared data area) without modifying the source itself.

Interface Structure

Event Inputs

Name	Type	Comment	With Variables
REQ	Event	Normal execution trigger	IN

Event Outputs

Name	Type	Comment	With variables
CNF	Event	Execution confirmation	OUT, IN

Data Inputs

This function block does not have traditional data inputs, but instead uses an **InOut parameter** that can be used as both an input and an output:

Name	Type	Initial value	Comment
IN	BYTE	0	Source variable (read via InOut)

Data Outputs

Name	Type	Initial value	Comment
OUT	BYTE	0	Buffered output value

Adapters

No adapters available.

Functionality

The function block is activated by an event at input `REQ`. The current value of the variable at the **InOut** parameter `IN` is then read and copied to output `OUT`. Finally, event `CNF` is triggered, confirming successful execution. The algorithm is very simple:

OUT := IN; Since `IN` is declared as an InOut parameter, the calling application must provide a BYTE variable that can be both read and (in the case of other function blocks) written. However, `GET_BYTE` only performs a read operation.

Technical Features

- **InOut Parameter**: The function block accesses an external variable directly without connecting it via a separate data input. This enables efficient, direct data binding and saves network resources.
- **Buffered Output**: The read value is buffered in `OUT` and remains until the next execution – even if the source variable changes in the meantime.
- **No State Machine**: The function block consists of a single state, which makes the implementation lean and deterministic.

State Overview

The function block has exactly one internal state:

State	Description	Output Event
REQ	No waiting; immediate execution of the read operation	CNF

There are no wait states or branches.

Application Scenarios

- **Reading Configuration Values**: A `BYTE` value from a central data block (e.g., a global configuration) is transferred to a local buffer.
- **Synchronization Between Function Blocks**: Another function block writes a `BYTE` value to a shared InOut variable, and `GET_BYTE` reads this value as needed.
- **Safe Data Transfer**: The buffered output prevents subsequent logic from reacting to a changing source value – it always works with the value valid at the time of the request.

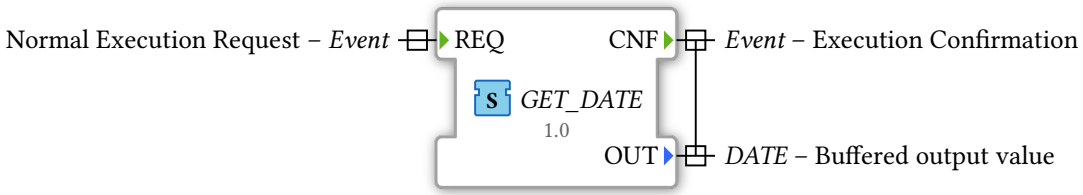
Comparison with Similar Function Blocks

Function Block	Description	Difference to GET_BYTE
GET_VALUE	Reads a value from an InOut parameter (type-independent)	GET_BYTE is specific to the data type `BYTE` and is therefore more type-safe.
MOVE	Copies a value from an input to an output	MOVE requires separate inputs and does not provide a buffer – GET_BYTE uses InOut.
READ_BYTE	Reads a byte via a hardware address	GET_BYTE operates at the variable level, not at hardware addresses.

Conclusion

GET_BYTE is a simple yet useful building block for reading a `BYTE` value from an external source via an InOut parameter and buffering it. Thanks to its streamlined design (one state, minimal interface), it is particularly suitable for resource-critical applications and for integration into existing communication or storage concepts where direct access to a shared variable is desired.

GET_DATE



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_DATE** function block reads a DATE value from an InOut variable and provides it as a buffered output. When requested by an event, the current value of the source is copied into an internal buffer and then output to the output as well as to the InOut variable itself.

Interface Structure

Event Inputs

Name	Type	Comment	With Variables
REQ	Event	Normal Execution Request	IN

Event Outputs

Name	Type	Comment	With Variables
CNF	Event	Execution Confirmation	OUT, IN

Data Inputs

Name	Type	Comment

IN	DATE	Source Variable (InOut)

Data Outputs

Name	Type	Comment
OUT	DATE	Buffered Output Value
IN	DATE	Source Variable (InOut) – also output with CNF

Adapters

No adapters available.

Functionality

The function block has a simple execution state **REQ**:

1. An incoming event at the **REQ** input triggers processing.
2. The internal algorithm copies the current value of the InOut variable **IN** to the output signal **OUT**.
3. Subsequently, the **CNF** event is sent. This passes both the value of **OUT** and the (unchanged) value of **IN** to the output.

The functionality corresponds to a simple value transfer without further processing or state storage across multiple calls.

Technical Features

- The use of an **InOut** variable allows direct access to an external variable without declaring a separate data input.
- Buffering occurs only during the execution of the algorithm; the current value of **IN** is read again with each new **REQ**.
- The function block is implemented as a **SimpleFB** and has no internal state transitions – processing is purely combinatorial in a single step.

State Overview

The function block consists of exactly one state:

- **REQ**: Upon entering this state, the algorithm that converts **OUT** to **IN** is executed, and the **CNF** event is sent.

There are no other states or conditional transitions.

Application Scenarios

- **Reading a Global Date**: When a system time or date needs to be read from a central variable and used locally.
- **Buffering for Subsequent Processing**: To hold a DATE value at a specific point in time and pass it to other function blocks.
- **Cascading InOut Accesses**: In combination with other function blocks that access the same InOut variable to ensure consistent data.

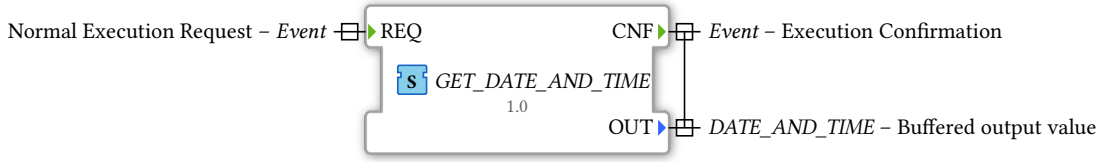
Comparison with Similar Function Blocks

- **GET_BOOL**, **GET_INT**, **GET_STRING**, etc.: These function blocks operate on the same principle but with different data types. They all use an InOut variable as a source and a buffered output.
- **SET_DATE**: The counterpart – it writes a value to an InOut variable instead of reading it.
- All these function blocks have a simple structure and are suitable for standardized access to shared data.

Conclusion

GET_DATE is a minimalist yet useful function block for reading a DATE value via an InOut variable. Its simple state logic and direct value copying make it reliable and easy to understand. It is particularly suitable for modular automation solutions where data needs to be exchanged between function blocks using a shared variable.

GET_DATE_AND_TIME



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_DATE_AND_TIME** function block is used to read a value of type `DATE_AND_TIME` from a variable declared as InOut and provide it as a buffered output. On each request, the current value of the source (InOut variable) is copied to the output.

Interface Structure

Event Inputs

Event	With	Description
REQ	IN	Normal execution request; triggers the reading of the source data.

Event Outputs

Event	With	Description
CNF	OUT, IN	Confirmation of execution; signals that the output has been updated.

Data Inputs

Name	Type	Description
IN	DATE_AND_TIME	InOut variable as the data source. It is read but not modified.

Data Outputs

Name	Type	Description
OUT	DATE_AND_TIME	Buffered output value that is taken from the InOut source on every REQ.

Adapter

None.

Functionality

The component has a simple internal state **REQ**. When this state occurs (triggered by the event `REQ`), the following algorithm is executed:

OUT := IN;

The event `CNF` is then sent. The value of the InOut variable `IN` is not modified; it is simply read and copied to the output `OUT`. Thus, the function block acts as a **buffer** for any `DATE_AND_TIME` value.

Technical Features

- The use of an **InOut variable** (`IN`) allows direct access to an external variable without having to declare it as a separate input or output. The function block accesses it in read mode.
- The output `OUT` stores the value present at the time of `REQ`. If the source changes between two `REQ` calls, the new value is only applied with the next `REQ` call (buffered provisioning).
- This is a **simple function block** (SimpleFB) without complex state management – only a single execution state.

State Overview

State	Action	Output Event
REQ	`OUT := IN;`	CNF

There is no initial state or other states; the function block waits for the `REQ` event and immediately executes the algorithm.

Application Scenarios

- **Timestamp Acquisition**: Retrieving the current time from a system clock, provided as an InOut variable, at a defined control point.
- **Buffering of Date/Time Values**: Temporarily storing a current date/time value for later processing steps.
- **Synchronization**: Transferring a common time value from a central source to various function blocks.

Comparison with Similar Function Blocks

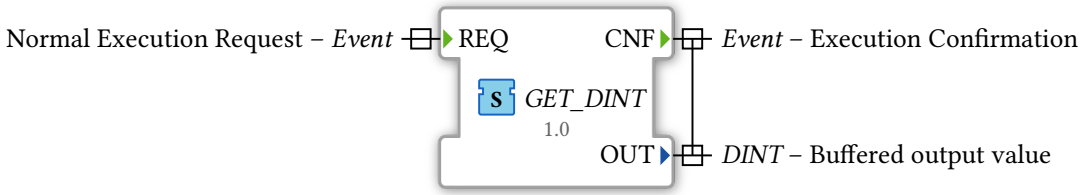
- **GET_DATE_AND_TIME**, unlike function blocks such as `E_GET_TIME` (real-time clock query) or `TIME_GET` (current system time): This function block does not use a built-in clock but reads from any InOut variable. This makes it more flexible, e.g., for accessing application-specific time values.

GET_DATE_AND_TIME Compared to a simple assignment block (e.g., `MOVE` with `DATE_AND_TIME`), it offers an event-driven, explicit request and explicit acknowledgment (`REQ`/`CNF`), which is advantageous for time-critical control applications.

Conclusion

GET_DATE_AND_TIME is a lean, clearly structured function block that allows reading and buffering a `DATE_AND_TIME` value from an InOut variable. Its ease of use and explicit event control make it a useful tool for reliably acquiring time data in automation applications.

GET_DINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **GET_DINT** reads a DINT value from an InOut variable and provides it as a buffered output value. The variable itself is not modified; the block serves purely as a read and buffer block.

Interface Structure

Event Inputs

- **REQ** – Normal execution command (activates reading the InOut variable).

Event Outputs

- **CNF** – Confirmation of execution (sent after successful value transfer).

Data Inputs

There are no traditional data inputs. The value source is provided via the InOut variable described below.

Data Outputs

- **OUT** (Type: DINT, Initial Value: 0) – Buffered output value containing the current value of the InOut variable.

Adapter

None.

Special Feature: The function block uses an **InOut variable** named **IN** (Type: DINT, Initial Value: 0). This variable serves as the source for the read operation. It is neither a pure data input nor a pure data output, but allows direct access to a memory location within the caller context. The function block accesses this variable for reading.

Functionality

When the **REQ** event input is activated, the function block performs the following steps:

1. The current value of the InOut variable **IN** is read.
2. This value is copied (buffered) to the data output **OUT**.
3. The event output **CNF** is then sent to signal successful execution.

Since this is a single action, the function block is deterministic and requires no further state transitions.

Technical Features

- **Direct Memory Access:** By using an InOut variable, the function block can access a variable outside its own instance. This allows data sharing between different function blocks without additional input connections.
- **Buffering:** The **OUT** output holds the read value until the next activation of **REQ**. This allows the value to be used multiple times without re-querying the source.
- **No Side Effects:** The function block does not modify the InOut variable **IN** it only reads its value.

State Overview

The function block has a single state, **REQ**. When this state is activated, the algorithm's action (copying the value) is executed, and the state is exited, sending **CNF**.

- – **REQ** (Start state and only state)
- Action: `OUT := IN;`
- Output event: **CNF**

There are no branches, loops, or wait times.

Application Scenarios

- **Buffering a DINT Value:** In a control application, a measured value (e.g., from a sensor) should only be read on request and temporarily stored for further processing.
- **Reading Shared Variables:** If multiple function blocks need access to a shared variable, **GET_DINT** can provide the current value without requiring the variable to be used as a data input for each function block.
- **Interface Decoupling:** Separating the read time from the actual processing through event-driven buffering.

Comparison with Similar Function Blocks

In the 4diac IDE, analogous function blocks exist for other data types (e.g., `GET_INT`, `GET_BOOL`). The basic mechanism is identical; only the data type differs.

Unlike a **MOVE** function block, which copies a value from an input to an output, **GET_DINT** reads the value from an InOut variable and buffers it. This allows reading variables that cannot be directly connected as data inputs.

Conclusion

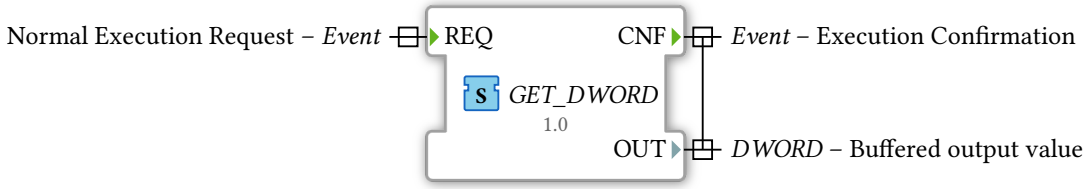
GET_DINT is a simple yet useful function block for event-driven reading and buffering of a DINT value from an InOut variable. Its strength lies in its ability to access shared variables and provide the read value without any side effects. Its clear structure and low complexity make it a reliable tool for many control tasks.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

GET_DWORD



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `GET_DWORD` is used to read the current value of a variable of type `DWORD`, provided as an InOut parameter, and to provide this value as a buffered output. The read operation is controlled by an event.

Interface Structure

Event Inputs

Event	Data Type	Comment
`REQ`	`Event`	Request to read the InOut variable and provide the output value

Event Outputs

Event	Data Type	Comment
`CNF`	`Event`	Confirmation that the read operation has completed and the output has been updated

Data Inputs

Name	Data Type	Comment
`IN`	`DWORD`	Source variable (InOut). The current value of this variable is read at `REQ`. Initial value: 0

Data Outputs

Name	Data Type	Comment
`OUT`	`DWORD`	Buffered output value. Contains the value read during the last execution. Initial value: 0

Adapter

No adapters available.

Functionality

When an event occurs at input `REQ`, the algorithm `REQ` is executed. This assigns the current value of the input variable `IN` to the output variable `OUT`. The event `CNF` is then sent.

The function block buffers the read value; that is, `OUT` retains its value until a subsequent `REQ` updates it. The InOut variable `IN` is directly bound to a parent variable, so each read operation returns the actual current value of this variable.

Technical Details

- **InOut Parameters:** `IN` is declared as an InOut variable. This means it serves as a reference to an external variable. The function block reads the value of this referenced variable but does not modify it (no feedback).
- **Initial Values:** Both `IN` and `OUT` are initialized to `0`. The initial value of `IN` only takes effect if the referenced variable is not explicitly set or is used in an environment without an external value.

- **No State Machine Complexity:** The function block is implemented as a simple SimpleFB with only one state. There are no internal state transitions other than executing the action.

State Overview

The function block has a single state, `REQ`:

State	Description
`REQ`	Waits for the event `REQ`. Upon its occurrence, the action `REQ` (OUT := IN) is executed, and then the event `CNF` is sent. After `CNF`, the function block remains in the same state.

Application Scenarios

- **Buffering a Sensor Signal:** A DWORD sensor value (e.g., counter reading) is to be sampled at specific times and temporarily stored for further processing.
- **Decomposing DWORD Data:** In combination with other function blocks such as `EXTRACT_BYTE` or `SPLIT_DWORD`, the read value can be decomposed into individual bytes or bits.
- **Updating the Process Image:** This function block is suitable for transferring the current value of an external global variable (e.g., from an industrial fieldbus) to a local buffer in a controlled manner.

Comparison with Similar Function Blocks

- **GET_DWORD with Simple Input:** A hypothetical function block with a normal data input (`VarInput`) would not pass the value by reference, but would use the current input value with each event. In contrast, `GET_DWORD` with InOut offers a fixed binding to a specific variable, which is more consistent in scenarios with multiple accesses to the same variable.
- **SET_DWORD:** The symmetric function block `SET_DWORD` would *set* the value of an InOut variable, while `GET_DWORD` only reads it. Both use the InOut concept, but with opposite actions.
-

Conclusion

GET_DWORDis a simple yet useful building block for the controlled retrieval of a DWORD value from an externally referenced variable. Its strength lies in buffering the value and separating the source (InOut) from the output (OUT). Thanks to its simple state machine and the standardizedREQ/CNF` interface pattern, it can be easily integrated into existing 4diac applications.

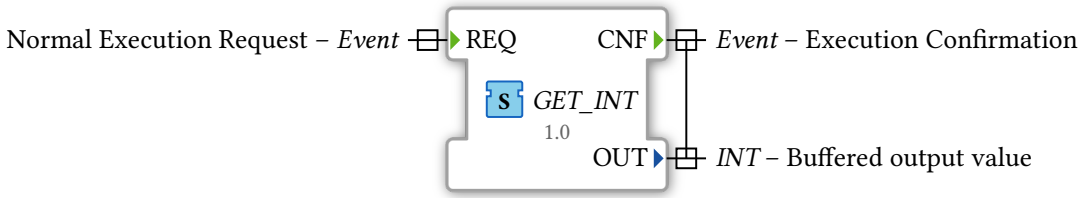
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

GET_INT



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_INT** function block is used to read the current value of an INT variable (defined as an InOut parameter) and provide it as a buffered value at the output. Buffering occurs with each execution cycle, ensuring the output value remains stable until the next request.

Interface Structure

Event Inputs

Event	Description
REQ	Normal execution request; triggers the reading of the InOut variable **IN**.

Event Outputs

Event	Description
CNF	Execution confirmation; indicates that the value of **IN** is buffered and available at output **OUT**.

Data Inputs

– *No standalone data inputs available.*
The input value is provided via the InOut parameter **IN** (see next section).

Data Outputs

Output	Type	Description
OUT	INT	Buffered output value containing the last read value from **IN**. Initial value: 0.

Adapters

– *No adapters available.*

InOut Variables (Implicit)

Variable	Type	Description
IN	INT	Source of the value to be read. Can be used as both input and output (read-only here). Initial value: 0.

Functionality

The FB operates as a simple read buffer:

1. Upon arrival of the **REQ** event, the **ST** algorithm `REQ` is executed:

``structured text OUT := IN;

1. The current value of the InOut variable **IN** is copied to the output **OUT**.
2. The **CNF** event is then issued to signal the completion of the operation.

Since **IN** is declared as an InOut parameter, the function block directly accesses the memory location of the parent application. The value is not modified, but simply read and temporarily stored.

Technical Features

- **InOut Parameter**: Using an InOut variable allows direct access to the data source without separate data inputs and outputs. This reduces the number of connections and saves resources.
- **Buffering**: The output **OUT** retains its value until the next **REQ** call. This ensures that the output value is independent of any changes to the source between two read cycles.
- **Initial Values**: Both **IN** and **OUT** start with 0, so defined values are available even without prior execution.

State Overview

The function block (FB) has a single state: **REQ**

- **REQ**: When activated, the algorithm `REQ` is executed, followed by the **CNF** event. The state persists (no transition to another state).

State	Entry Condition	Action	Output Event

REQ	Event **REQ** received	`OUT := IN;`	CNF

Application Scenarios

- **Data Buffering**: An INT value used by multiple function blocks should be stably available without each block having to directly access the source.
- **Synchronization**: In time-critical control systems, the current value of a variable can be "frozen" at a specific point in time and processed later.
- **Replacement of a MOVE function block**: If direct memory access via InOut is preferred for performance reasons, **GET_INT** replaces the classic MOVE function block with separate input and output.

Comparison with similar function blocks

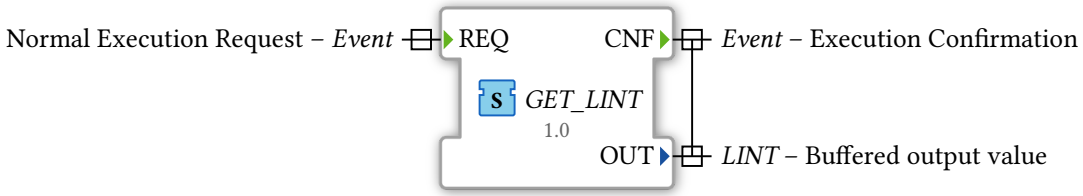
Function block	Input type	Buffering	Special feature
GET_INT	InOut (INT)	Yes (one-time read on REQ)	Accesses the original variable, no additional variable required
MOVE (e.g., IEC 61131-3)	Input (INT)	Yes (on every execution)	Copies the value from an explicit input to an output
F_TRIG / R_TRIG	Input (BOOL)	No (edge detection only)	Works with Boolean values

The main difference lies in the elimination of a separate data input through the use of an InOut parameter.

Conclusion

GET_INT is a simple yet effective function block for buffered readout of an INT variable. By using an InOut parameter, it reduces connection overhead and is particularly suitable for applications requiring a stable instantaneous value. Its clear, low-state structure makes it reliable and easy to understand.

GET_LINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **GET_LINT** reads a value of type **LINT** from an InOut variable and makes this buffered value available at its output. The block is part of the library `logiBUS::utils::conversion::provide` and implements a simple read-buffer functionality.

Interface Structure

Event Inputs

Name	Type	Comment
REQ	Event	Normal Execution Command

The event input REQ triggers the execution of the function block. The current value of the InOut variable **IN** is read and temporarily stored in the output **OUT**.

Event Outputs

Name	Type	Comment

CNF	Event	Execution Confirmation

The CNF event is sent after successful processing of the REQ event. It signals that the OUT output contains the current value of the InOut variable IN.

Data Inputs

Name	Type	Comment
IN	LINT	Source Variable (InOut) – Initial Value: 0

IN is declared as an InOut variable. It can be both read and written, but in this function block, it is only read. The passed value is copied to the OUT output with each REQ event.

Data Outputs

Name	Type	Comment
OUT	LINT	Buffered Output Value – Initial Value: 0

The output **OUT** holds the last value read from the InOut variable **IN**. It is updated with each REQ event.

Adapter

This function block has no adapter interfaces.

Functionality

The GET_LINT function block implements a simple buffer function. It has a single entry state **REQ**, which is activated by an event at the corresponding input. In this state, the algorithm `REQ` is executed:

OUT := IN;

The confirmation event **CNF** is then sent. The output **OUT** then contains the current value of the InOut variable **IN**. The function block then waits for the next REQ pulse.

Technical Features

- **InOut Variable**: The function block uses an InOut parameter (`IN`). This allows direct access to a variable that can be both read and written. In the case of GET_LINT, however, the value is only read and not modified.
- **LINT Data Type**: The function block works exclusively with the 64-bit integer type LINT (Long Integer). It is intended for environments where large integer values are processed.
- **Simplest State Machine**: There is only one state with one direct action and one output. This makes the function block deterministic and resource-efficient.
- **No Error Handling**: The function block does not contain any plausibility checks or timeouts. It assumes that the InOut variable provides valid data.

State Overview

State	Action	Output
REQ	Copy from IN to OUT	CNF

The function block starts in the **REQ** state and returns to the same state after executing the action and sending CNF (endless loop on each event).

Application Scenarios

- **Caching a LINT Value**: When a LINT value needs to be read from a global variable or another function block and held constant for a specific period (e.g., for subsequent processing).
- **Synchronization**: The function block can be used to capture the current value of an InOut variable at a specific clock cycle and pass it to other function blocks.
- **Decoupling**: The buffer decouples the reading function block from the source – changes to the source only take effect after the next REQ pulse.
- **Simple Value Transfer**: In control systems, GET_LINT can be used to retrieve an external measured value or a parameter from another function block.

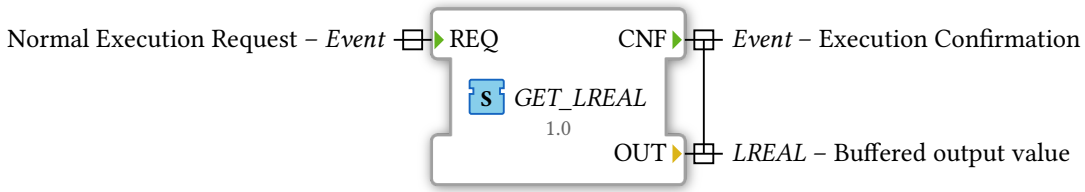
Comparison with Similar Function Blocks

- **GET_DINT / GET_BOOL**: Analogous function blocks for other data types (DINT, BOOL). The functionality is identical; only the data type differs.
- **MOVE (e.g., MOVE_INT)**: A MOVE function block also copies a value but uses input and output variables by default instead of InOut. InOut variables allow a direct reference to the source without additional data flow.
- **LINT_READER** (fictitious): A hypothetical function block that offers additional functions such as buffer timeout or read lock. GET_LINT, on the other hand, is minimalist and fast.

Conclusion

The GET_LINT function block offers a lean and reliable method for reading a LINT value from an InOut variable and buffering it. Thanks to its simple state machine and clear interface, it is ideally suited for basic synchronization tasks in IEC 61499-based systems. The use of InOut parameters keeps the implementation flexible without requiring additional data lines or protocols.

GET_LREAL



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `GET_LREAL` provides read access to a `LREAL` variable (64-bit floating-point number) which is passed as an InOut parameter. The read value is buffered internally and made available at the output `OUT`. Execution is triggered by an event.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Normal Execution Trigger; Reads the value of `IN`

Event Outputs

Event	Data Type	Comment
CNF	Event	Confirmation of execution; sent after reading

Data Inputs

This function block does not have its own data inputs. The source data is provided via the InOut parameter `IN`.

Data Outputs

Output	Data Type	Initial Value	Comment
OUT	LREAL	0.0	Buffered Output Value

Adapter

No adapters available.

Functionality

1. An event at input `REQ` triggers processing.

2. The function block executes the ST instruction `OUT := IN;`. The current value of the InOut variable `IN` is read and temporarily stored in the local output variable `OUT`.
3. After successful reading, the event `CNF` is output. The value of `OUT` is retained until the next execution.

Technical Features

- The source `IN` is declared as an **InOut variable**. It must be linked to an actual variable (e.g., a global or hardware resource) by the calling environment.
- The function block buffers the read value: as long as no new `REQ` pulse is received, `OUT` returns the last read state.
- Both parameters (`IN` and `OUT`) are initialized with the value `0.0`.

State Overview

Since this is a simple function block (SimpleFB), there is only one ECC state:

State	Action	Output Event
REQ	`OUT := IN;`	CNF

Application Scenarios

- **Buffered reading of a high-frequency process variable** (e.g., temperature or pressure) to keep the value constant during the control cycle.
- **Accessing a shared memory variable** in a multitasking environment where decoupled read speed is desired.
- **Preparing data for downstream calculations** that rely on a stable, buffered floating-point value.

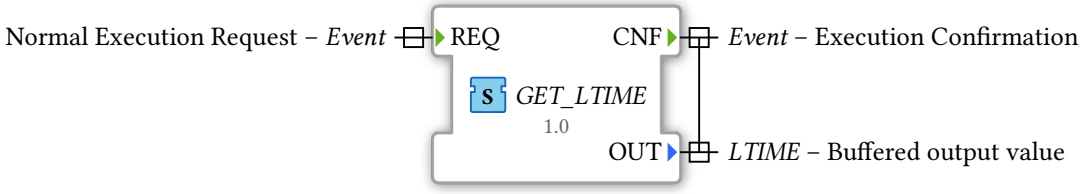
Comparison with Similar Components

Components such as `GET_REAL` or `GET_INT` operate on the same principle, but with shorter data types. `GET_LREAL` is specifically designed for applications requiring high numerical accuracy (64-bit). Unlike pure read blocks without buffering (e.g., direct variable linking), the buffered output guarantees deterministic delivery that is independent of the time of the read.

Conclusion

GET_LREAL`is a simple, reliable block for buffered reading of a`LREAL` source. It is particularly suitable for decoupling read and processing logic and ensures that the provided value remains constant throughout a complete execution cycle.

GET_LTIME



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **GET_LTIME** is used to read the current value of an LTIME variable and provide it as a buffered output value. It accesses the source via an InOut interface and makes the read value available at the output as soon as an event occurs. This enables controlled, one-time reading of a time variable.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Normal execution command. Associated with the input value IN.

Event Outputs

Event	Data Type	Comment
CNF	Event	Execution confirmation. Connected to the outputs OUT and IN.

Data Inputs

This function block does not have traditional data inputs. Access to the source value is exclusively via the InOut variable.

Data Outputs

Name	Data Type	Comment
OUT	LTIME	Buffered output value (initial value LT#0s).

Adapter

None.

Functionality

This function block implements simple, event-driven copy logic:

1. An incoming **REQ** event triggers the execution of the internal algorithm.
2. The algorithm assigns the current value of the InOut variable **IN** to the output **OUT** (`OUT := IN;`).
3. After successful assignment, the output event **CNF** is sent to signal completion to the calling instance.

Thus, the value of the external LTIME variable is only read when the REQ event arrives and is buffered in the output until the next read.

Technical Features

- **InOut Variable**: The source is accessed via an InOut interface, which represents a direct reference to an external variable. Changes to the external variable between two REQ calls are only applied during the next read operation.
- **Buffering**: The read value remains in the output OUT until another REQ event arrives. This protects against asynchronous changes to the source during processing.
- **Reading Back the Input**: The **CNF** event is connected not only to OUT but also to IN. This can be used in certain runtime environments to control the forwarding of the InOut value, but it has no additional functional significance in the standard behavior.

State Overview

The function block is implemented as a SimpleFB and has exactly one internal state:

State	Action	Output Event
REQ	Algorithm REQ (OUT := IN)	CNF

After receiving **REQ**, the state is traversed once and the output is generated. There are no further states or branches.

Application Scenarios

- **Timestamp Capture**: Freezing a current system time value (LTIME) once at a specific event, e.g., an alarm signal.
- **Synchronization of Time Measurements**: Reading a shared timer in an industrial controller to pass consistent time data to various downstream function blocks.
- **Buffering of Time Values** in data-driven architectures where an LTIME variable is used multiple times but must be read at precisely the same time.

Comparison with Similar Function Blocks

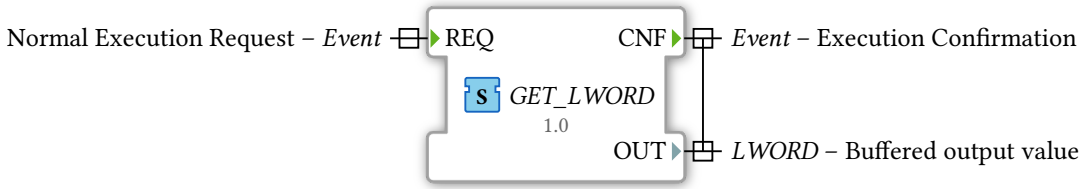
- **GET_VALUE (for other data types)**: Functions analogously, but for scalar data types such as INT, REAL, etc. GET_LTIME is specialized for the LTIME data type.

- **Latch Function Blocks**: While latch blocks store and hold a value upon an event, GET_LTIME additionally generates an acknowledgment event and reads from an InOut variable instead of a normal data input.
- **MOVE Block**: A MOVE block also copies a value from an input to an output, but without event control and without InOut support. GET_LTIME offers an event-driven, buffered version.

Conclusion

The **GET_LTIME** function block is a simple yet effective component for the controlled reading of an LTIME variable via an InOut interface. Its event-driven operation and buffering of the output value make it particularly suitable for time-critical applications where a consistent time value is required at a defined point in time. The clear interface and simple state logic facilitate integration into larger control applications.

GET_LWORD



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_LWORD** function block is used for the consistent retrieval of an LWORD value from an **InOut variable**. It buffers the current value of the source variable and makes it available at the output `OUT`. This is particularly useful when the content of a shared variable needs to be saved at a defined point in time without being affected by subsequent write operations from other function blocks.

Interface Structure

Event Inputs

Event	Description
`REQ`	Starts the read operation. The current value of `IN` is transferred to `OUT`.

Event Outputs

Event	Description
`CNF`	Confirms successful completion of the operation. Triggered after the copy operation.

Data Inputs

This function block does not have traditional data inputs but uses an **InOut parameter** that functions as a bidirectional interface.

Data Outputs

Variable	Type	Description
`OUT`	`LWORD`	Buffered output value. Contains the copy of the value of `IN` at the time of the last `REQ`. Initial value = `0`.

Adapter

None available.

Functionality

The function block implements a simple sequence:

1. **Event `REQ`** triggers the execution of algorithm `REQ`.
2. **Algorithm `REQ`**: The instruction `OUT := IN;` copies the current value of the InOut variable `IN` to the output buffer `OUT`. The InOut interface ensures direct read access to the source variable.
3. After the assignment is complete, the output event **`CNF`** is sent. The event outputs `CNF` are linked to both `OUT` and `IN`, allowing subsequent function blocks to use both the new value and the reference to the InOut variable.

Buffering ensures that the value of `OUT` remains stable during a run cycle, even if the source `IN` is modified by other function blocks in the meantime.

Technical Features

- **InOut Parameter**: The variable `IN` is declared as `InOutVar`. This allows the function block to directly access an external memory location without requiring an additional connection as a data input. The function block reads this value but does not modify it.
- **Simple State Logic**: The component has only a single state, `REQ`. Therefore, exactly one action is executed for each `REQ` event – execution is deterministic and instantaneous.
- **Variable Type**: `LWORD` (64-bit logarithmic or bifield value) is treated as a safe buffer.

State Overview

State	Description
`REQ`	Initial and only state. Upon receiving `REQ`, the algorithm is executed and `CNF` is sent. The state is retained, so the process is repeated upon receiving `REQ` again.

Application Scenarios

- **Consistent Sampling**: A `GET_LWORD` function block can be used to read the value of a cyclically updated sensor at a defined time and store it for further processing (e.g., a calculation chain).
- **Preventing Race Situations**: In systems with multiple parallel function blocks accessing the same memory location, buffering prevents a function block from receiving an inconsistent value in the interim.
- **Data Logging**: A snapshot can be created using `GET_LWORD` before a measurement is stored in a log memory.

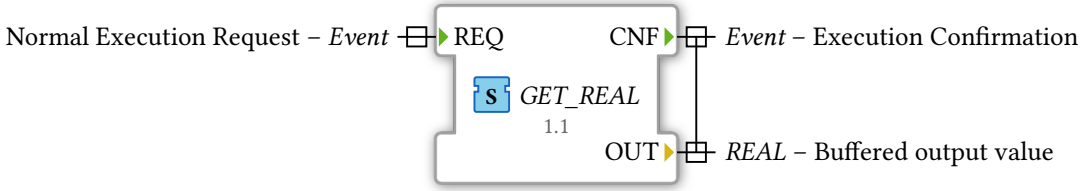
Comparison with Similar Components

- **`GET_INT`, `GET_DWORD`, `GET_BOOL`** etc.: These components follow the same principle but differ in data type. `GET_LWORD` is specifically designed for 64-bit values.
- **Simple Input Variable**: A pure data input would require the connecting component to buffer the value itself or constantly update the connection. The InOut mechanism provides a reference to the original, which is more direct and memory-efficient in some architectures.
- **`MUX` Components**: A multiplexer could also select a value but requires additional control lines. `GET_LWORD` is optimized for one-time, deterministic reading.

Conclusion

The function block `GET_LWORD` is a simple yet effective building block for the safe and time-controlled querying of an InOut variable of type `LWORD`. Its minimal state machine and direct copy logic make it ideal for real-time applications requiring consistent data retrieval. By utilizing the InOut concept, efficient coupling to existing memory locations is achieved without increasing the complexity of the connection structure.

GET_REAL



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_REAL** function block is used to read a REAL value from an **InOut variable** and make it available in a buffered output. The read value is obtained from the InOut variable `IN` when an event is received at input `REQ` and output at output `OUT`. After execution, the acknowledgment event `CNF` is sent.

Interface Structure

Event Inputs

Event	Description	With Variables
REQ	Normal execution request – triggers reading the InOut variable.	`IN`

Event Outputs

Event	Description	With Variables
CNF	Execution Acknowledgement – sent after processing `REQ`.	`OUT`, `IN`

Data Inputs

Variable	Type	Initial Value	Description
IN	REAL	0.0	InOut variable whose current value is read.

> **Note:** `IN` is declared as an *InOut variable* – it can be both read and written. However, in this function block, it is only read.

Data Outputs

Variable	Type	Initial Value	Description
OUT	REAL	0.0	Buffered output – holds the read value of `IN` until the next update.

Adapters

No adapters available.

Functionality

1. An incoming event at input `REQ` triggers the execution of the internal algorithm `REQ`.
2. In the algorithm, the current value of the input variable `IN` is copied to the output variable `OUT`:

```
OUT := IN;
```

1. The confirmation event `CNF` is then sent. The outputs `OUT` and `IN` are associated with the event, so they are valid simultaneously.
2. The output value is retained until another `REQ` event arrives and `OUT` is rewritten.

Technical Features

- **InOut Usage:** The function block uses an InOut variable (`IN`). This can be written to or read from externally; here, it serves exclusively as a source.
- **Initialization:** Both REAL variables are pre-assigned to `0.0`.
- **Minimal Overhead:** The function block (FB) has only one state (`REQ`) and a simple assignment algorithm – it operates without delay or additional logic.
- **Standard Compliance:** The function block is defined according to IEC 61499-1 and is intended for use in automation environments such as 4diac IDE.

State Overview

The FB has only one state:

- **State `REQ`:**

Upon entry, the algorithm `REQ` is executed (OUT := IN) and the event `CNF` is sent. The FB remains in this state and is always ready for the next request.

There are no further states or branches – the function block is deterministic and reacts immediately to every `REQ` event.

Application Scenarios

- **Reading a Process Value:** In a PLC, `GET_REAL` can be used to read a current sensor value, which is available as an InOut variable, and pass it to a subsequent function block (e.g., for calculations or visualization).
- **Value Comparison Between Function Blocks:** If another function block provides an InOut interface with a REAL value, `GET_REAL` can retrieve this value and make it available as a buffered output.
- **Testing and Simulation:** The function block is suitable for debugging to capture and log the current value of a variable at a specific point in time.

Comparison with Similar Function Blocks

Similar function blocks exist for other data types, e.g., `GET_BOOL`, `GET_INT`, `GET_LREAL`, etc. They all share the same principle: reading an InOut variable and providing it as a buffered output. The difference lies solely in the data type and the associated precision or value range.

Function Block	Data Type	Application Area
GET_BOOL	BOOL	Binary Signals
GET_INT	INT	Integers (16 bits)
GET_REAL	REAL	Floating-Point Numbers (32 bits)
GET_LREAL	LREAL	Precise Floating-Point Numbers (64 bits)

All these function blocks can be used analogously and offer a uniform interface for accessing InOut variables.

Conclusion

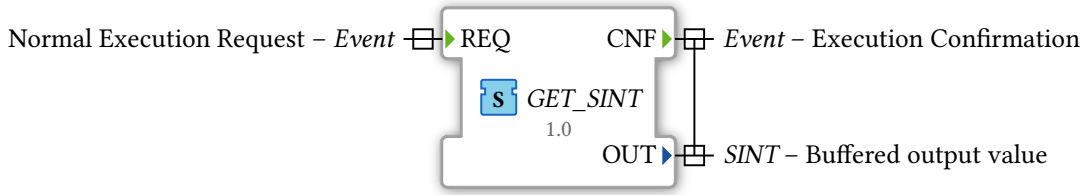
The `GET_REAL` function block is a lean and reliable block for reading a REAL InOut variable. It fulfills a fundamental task in the IEC 61499 world: forwarding a value from a bidirectional interface to a buffered output. Its simplicity and clear semantics make it suitable for both production and testing purposes. Its combination with parallel function blocks of other data types makes it a flexible tool in automation technology.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

GET_SINT



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_SINT** function block is used to read a value of the data type `SINT` (signed short integer) from a variable declared as `InOut` and provide it as a buffered output. It is typically used to access a shared variable or a physical input without modifying the original value. The function block operates according to the **IEC 61499** standard.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Normal execution request. Triggers the reading of the InOut variable.

- **With Link**: The event input REQ is linked to the data variable IN.

Event Outputs

Event	Data Type	Comment
CNF	Event	Confirmation of execution. Sent after a successful copy operation.

- **With Link**: The event output CNF is linked to the data variables OUT and IN.

Data Inputs

This function block does not have traditional data inputs. Instead, it uses a variable named InOut.

Data Outputs

Variable	Data Type	Initial Value	Comment
OUT	SINT	0	Buffered output value. Contains the copy of IN after a successful REQ execution.

Adapter

None.

Functionality

The function block performs a simple copy operation:

1. An event at the input REQ triggers the execution of the stored algorithm.
2. The algorithm assigns the current value of the InOut variable IN to the output OUT: `OUT := IN;`.
3. After the assignment is complete, the event CNF is sent to signal completion to the calling function block.

The InOut variable IN allows direct access to an external variable or memory location without requiring it to be defined as a separate input or output. The function block buffers the read value, ensuring that the output OUT remains stable even after changes to the source, until the next request is made.

Technical Features

- **Use of InOut Variables**: IN is declared as InOut, meaning it can be both read and written. In this case, it is only read, allowing for flexible coupling with the calling network.
- **Buffering**: The output OUT stores the read value until the next update by REQ. This prevents unwanted side effects from changes to the source.
- **No State Logic**: The function block consists of a single state (REQ) that reacts immediately to each incoming event. No delay, no waiting for conditions.
- **Initial Values**: Both IN and OUT are initialized with 0, ensuring a defined start state.

State Overview

The function block uses a simple ECA state machine with exactly one state:

State	Action	Outgoing Event
REQ	REQ Algorithm (OUT := IN)	CNF

- Every incoming REQ event triggers an immediate action and sends CNF. There are no branches, error states, or timing issues.

Application Scenarios

- **Reading Hardware Inputs**: When a SINT value needs to be read from an analog input chip or a programmable logic controller (PLC), `GET_SINT` can be used as an interface to capture the current value at a specific point in time.
- **Accessing Shared Variables**: In a distributed control system, `GET_SINT` can be used to transfer the value of a global or peripheral variable into a local processing chain.
- **Data Consistency**: Buffering ensures that the read value remains stable throughout the entire processing of the subsequent network, even if the source is updated asynchronously.

Comparison with Similar Building Blocks

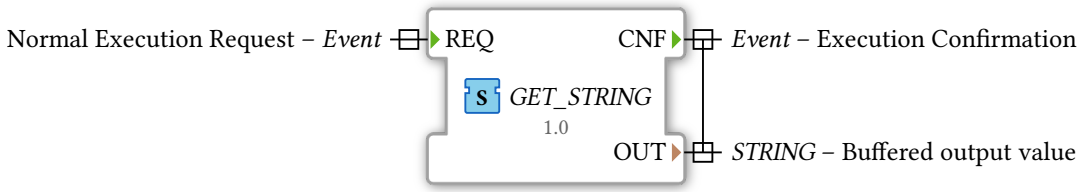
Building Block	Data Type	Special Feature
`GET_BOOL`	BOOL	Same functionality, but for Boolean values.
`GET_INT`	INT	Similar to `GET_SINT`, but with 16-bit integers.
`GET_REAL`	REAL	For floating-point numbers.
`MOVE` (generic)	Any	A generic MOVE building block can also copy values, but requires separate inputs and outputs. In contrast, `GET_SINT` uses InOut variables, which can be more practical when coupling to peripheral addresses.

All `GET_*` function blocks have in common that they read a value from an InOut source and provide it as a buffered output. In contrast, `SET_*` overwrites the value of the source.

Conclusion

GET_SINT`is a minimalist and reliable function block for reading a`SINT` value from an InOut variable. Its simplicity makes it an ideal function block for reading peripheral or shared data in IEC 61499 applications. The use of the InOut interface reduces wiring complexity and promotes a clean separation between data access and processing logic.

GET_STRING



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `GET_STRING` is used to read a string value from an InOut variable and provide it as a buffered output. It is typically used to capture the current value of an external variable at a specific time and make it available for further processing.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Normal execution request. Triggers the reading of the InOut variable.

Event Outputs

Event	Data Type	Comment
CNF	Event	Confirmation of execution. Sent after the value has been read and passed to the output.

Data Inputs

Name	Data Type	Comment
IN	STRING	InOut variable that serves as the source. Its current value is read and buffered on REQ.

Data Outputs

Name	Data Type	Comment
OUT	STRING	Buffered output value. Contains the string read from IN at the time of the last REQ.

Adapter

None.

Functionality

The function block has a state `REQ`. An incoming event at `REQ` executes the algorithm, which copies the value of the InOut variable `IN` directly to the output `OUT`. Then, the event `CNF` is sent. The InOut variable `IN` is not modified, but only read. The buffered value remains until another `REQ` reads a new value.

Technical Features

- The variable `IN` is declared as **InOut**, meaning it is passed by reference. This allows direct access to the memory of the passed string variable without first copying the value locally.
- The function block is specialized for the data type `STRING`. Corresponding variants (e.g., `GET_ANY`) are required for other data types.
- No initial values are set. The output `OUT` is undefined after the first cycle until `REQ` has been received.

State Overview

The function block has a single state:
| State Name | Description | | State Name | Description | |-------|-------| | REQ | Waits for an event at `REQ`. Executes the algorithm (copies `IN` ⟶ `OUT`) and sends `CNF`. |
There are no start or end states; the block returns to state `REQ` after each iteration.

Application Scenarios

- **Caching**: A string value from an external source (e.g., a data block) is sampled and cached for subsequent processing without affecting the source.
- **Logging**: Snapshot of a changing string at specific event times.
- **Synchronization**: Provides a consistent value for multiple downstream blocks when the source is to be read from several locations simultaneously.

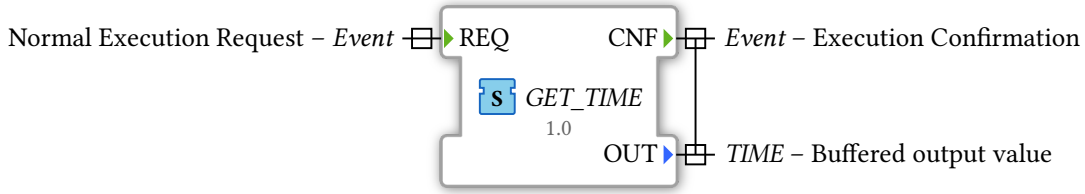
Comparison with similar blocks

- **MOVE / ASSIGN**: Simple assignment blocks for arbitrary types, often without explicit event handling. `GET_STRING`, on the other hand, is specifically designed for `STRING` and uses an InOut interface.
- **GET_ANY**: A generic block for arbitrary data types. `GET_STRING` is type-safe and avoids parameterization overhead for reading strings.
- **READ_REF**: Also a block for passing by reference, but often implemented as a function block without event handling. `GET_STRING` integrates event handling and is therefore suitable for sequential control systems.

Conclusion

GET_STRING` is a simple yet specialized function block for buffering a string value from an InOut variable. Its event-driven approach allows for precise use in time-critical processes. Using an InOut interface saves memory and processing time, as no value copy is required on the caller side. For most applications that need to sample strings, this function block offers a clean and resource-efficient solution.

GET_TIME



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **GET_TIME** allows you to read a TIME value from an InOut variable and provide it as a buffered output value. It serves as a simple interface for incorporating an external time value into the block structure.

Interface Structure

Event Inputs

Name	Type	Comment	With Variables
REQ	Event	Normal Execution Request	IN

Event Outputs

Name	Type	Comment	With Variables
CNF	Event	Execution Confirmation	OUT, IN

Data Inputs

No dedicated data inputs available.

Data Outputs

Name	Type	Comment	Initial Value
OUT	TIME	Buffered Output Value	T#0s

Adapters

No adapters available.

Functionality

GET_TIME implements a simple, event-driven process. An event at the **REQ** input triggers the execution of the algorithm. The current value of the InOut variable **IN** is copied to the data output **OUT** and temporarily stored. An event is then generated at the output **CNF** to acknowledge the successful transfer. The InOut variable **IN** serves solely as a readable source and retains its value.

Technical Features

- The variable **IN** is declared as **InOut**. It can be externally linked to another variable or output without the function block accessing it for writing. This allows for flexible integration with external timers or signals.
- The output **OUT** buffers the read value. After a **REQ** event, the current **IN** value is available at the output until the next request.
- The function block does not have its own timer or buffer for delays.

State Overview

State	Description	Action Performed

REQ	Waits for an event at the **REQ** input. Upon receipt, the algorithm is executed and a **CNF** event is sent.	`OUT := IN;`

Application Scenarios

- **Time Measurement**: Acquisition of a current system time (e.g., `TIME()` or a clock signal) into an output for further processing.
- **Signal Forwarding**: Buffered forwarding of an external TIME signal to decouple asynchronous changes in the source from the rest of the process.
- **Initialization**: Setting a start time value in a control logic.

Comparison with Similar Function Blocks

Compared to a pure **MOVE** function block for TIME data, **GET_TIME** uses an **InOut** interface, which allows a direct connection to the source even without an additional data input. It is specifically designed for TIME values and does not offer a general data copy function. Other time function blocks such as **E_TIME** or **TIME_IN** often work with their own timer or event counter, while **GET_TIME** passively retrieves an external value.

Conclusion

GET_TIME is a lean, clear function block for reading and buffering a TIME value. Its simple event handling and the use of an InOut variable make it particularly suitable for scenarios in which an external time value needs to be selectively incorporated into control logic.

GET_TIME_OF_DAY



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `GET_TIME_OF_DAY` is used to read a time of day value (type `TIME_OF_DAY`) from an InOut variable. The read value is buffered internally and made available via a data output. In this way, a consistent, uniquely read time of day is passed on – even if the source variable changes in the meantime.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Execution Request; reads the current value of IN

Event Outputs

Event	Data Type	Comment
CNF	Event	Confirmation after successful read and buffering

Data Inputs

There are no pure inputs. The source data is provided via the InOut input `IN`.

Data Outputs

Output	Data Type	Comment
OUT	TIME_OF_DAY	Buffered time of day value

Adapters

No adapters are available.

Functionality

A call to the event `REQ` triggers the internal associated action: The current value of the InOut variable `IN` is read and copied to the output variable `OUT`. The event `CNF` is then sent. The time of day is then consistently available at the output `OUT` until the next `REQ` – regardless of any interim changes to `IN`.

Technical Features

- **InOut variable as source:** The function block uses an InOut parameter that is directly linked to an external variable in the IEC 61499 model. This allows the function block to read the value "from the outside" without having to define a data input itself.
- **Buffering:** The read value is temporarily stored in the output variable `OUT`, so that subsequent queries always return the same value until `REQ` is triggered again.
- **Simple State Machine:** The function block consists of only one state, which executes the action and immediately outputs the confirmation event.

State Overview

The simple function block has only the state `REQ`. As soon as the event `REQ` is received, the action `REQ` is executed (OUT := IN) and the event `CNF` is sent. There are no further states or branches.

Application Scenarios

- **Timestamping in Controllers:** A one-time capture of a time value from a higher-level clock, for example, to document operating hours or switching times.
- **Buffering for Asynchronous Evaluation:** An external timer (e.g., a system clock) changes continuously; the function block freezes its value at the time of the call and makes it available to a slower consumer or logic.
- **Communication Across Boundaries:** In distributed systems, the current time of day can be retrieved from a component and cached locally for consistent output later.

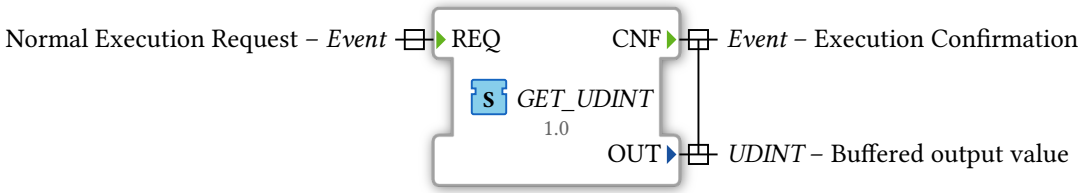
Comparison with Similar Function Blocks

- **Unlike a pure data connector** (e.g., MOVE or simple data copy), `GET_TIME_OF_DAY` uses an InOut parameter, which allows direct coupling to the source variable and thus cleanly separates the signal paths in the IEC 61499 model.
- **Unlike time function blocks**, which independently generate ticks or real-time values, this function block only accesses an externally provided value – it is purely read and buffer logic.
- **Similar function blocks** exist for other data types (e.g., `GET_BOOL`, `GET_REAL`) and follow the same design pattern.

Conclusion

GET_TIME_OF_DAY` is a compact yet useful function block that reads a time of day from an InOut variable, buffers it, and passes it on as a controlled output value. Its clear, low-state logic and buffering make it particularly suitable for tasks where a recorded time must be available in the same format at a later time.

GET_UDINT



• – ◦ • – ◦ • – ◦ •

The **GET_UDINT** function block reads the current value of a variable of type **UDINT** declared as InOut and provides it as a buffered output. Buffering ensures that the output value remains stable during a cycle, even if the source data changes later.

Name	Type	Comment
REQ	Event	Normal Execution Request

Associated Data:

- **IN** (InOut variable) is read on REQ.

Name	Type	Comment
CNF	Event	Execution Confirmation

Associated Data:

- **OUT** (Output Value) is provided after a successful read.
- **IN** (InOut Variable) is also associated, meaning its value can be read at the output after the event.

Name	Type	Default Value	Comment
IN	UDINT (InOut)	0	Source Variable to be Read
Name	Type	Default Value	Comment
––	––	–––––	–––––
OUT	UDINT	0	Buffered Output Value

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

None.

Functionality

Upon an event at the **REQ** input, the function block executes the algorithm and sends an acknowledgment event to **CNF**. The algorithm copies the current value of the InOut variable **IN** to the output variable **OUT**. This freezes the value at the time of the request and makes it stably available at the output until the next request.

- **InOut Variable:** The parameter **IN** is declared as InOut. This allows direct access to the memory address of the source variable, which can be useful for communication over distributed systems (e.g., fieldbuses). However, the function block only reads the value; it does not write back.
- **No Side Effects:** The function block does not modify the InOut variable.

The function block has only one state:

- **REQ:** The algorithm is executed. After copying the value, the output **CNF** is immediately set.

There are no other states; the function block is always ready to process a request.

- **Buffering of Process Values:** In automation systems where a measured value or manipulated variable (e.g., a counter reading or speed) needs to be frozen at a specific interval before being passed on to other function blocks.
- **Decoupling of Cycles:** When an InOut variable is read by multiple function blocks and a stable value is required within a cycle.
- **Replacement for direct access:** Instead of directly reading an InOut variable, the function block (FB) provides a clear interface (event input, output) and improves system readability.

- **MUX (Multiplexer):** A MUX selects between multiple inputs, while **GET_UDINT** buffers exactly one defined value from an InOut variable.
- **GET_DINT / GET_REAL:** Analogous function blocks for other data types that implement the same principle of buffered output of InOut values.
- **Latch function blocks:** These store values and output them upon an event; **GET_UDINT**, on the other hand, overwrites the output with the current value of the source on every request.

The **GET_UDINT** function block is a simple yet useful tool for stably and event-drivenly caching the value of an InOut variable of type UDINT. It is particularly suitable for applications where access to external or shared variables should occur in controlled steps, thus contributing to clean and predictable signal processing.

Technical Features

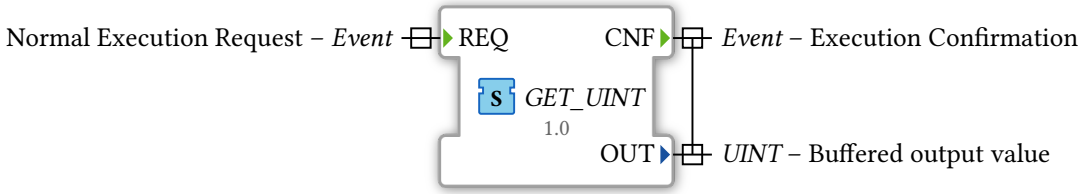
State Overview

Application Scenarios

Comparison with Similar Function Blocks

Conclusion

GET_UINT



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_UINT** function block reads the current value of a source `UINT` passed as an InOut variable and provides it as a buffered output value. It serves to capture a consistent snapshot of an external or internal UINT value at a defined point in time and make it available for further processing in the workflow.

Interface Structure

Event Inputs

Event	Description
REQ	Normal execution command; triggers reading the InOut variable and updating the output.

Event Outputs

Event	Description
CNF	Confirms the successful execution of REQ and signals that the outputs are valid.

Data Inputs

Variable	Type	Description
IN	UINT	Source variable provided as an InOut parameter. The value is read during REQ and passed unchanged to the output. Default value: `0`.

Note: `IN` is declared as an InOut variable, but is only used for reading within the function block.

Data Outputs

Variable	Type	Description
OUT	UINT	Buffered output value containing the value read from `IN` during the last REQ execution. Default value: `0`.

Adapter

None.

Functionality

The function block has a single execution state (`REQ`). When the event input `REQ` is activated, it performs the following action:

1. **Read** the current value of the InOut variable `IN`.

None.
The function block has a single execution state (`REQ`). 2. **Copy** this value to the output variable `OUT`.

1. **Trigger** the event output `CNF` to signal completion.

Processing is strictly sequential and instantaneous.

Technical Features

- **InOut variable as source:** Although `IN` is declared as an InOut variable, it is used for reading only. The function block does not write back, so the source remains unchanged.
- **Buffering:** The output `OUT` holds the value until the next output, `REQ`, is executed. This achieves decoupling between the source and the subsequent logic.
- **Simple state model:** The function block consists of only one state, so no complex sequence controls are required.

State Overview

State	Description
REQ	Waits for the arrival of event `REQ` and then performs the assignment `OUT := IN`. Subsequently, `CNF` is sent.

There is no explicit start or end state; after execution, the function block remains in state `REQ`, ready for the next task.

Application Scenarios

- **Safe Value Transfer:** A UINT value from a global variable or another function block should be frozen at a defined point in time to maintain consistency throughout a process.

Safe Value Transfer: - **Sampling of Measured Values:** In a cyclic control system, the current value of a counter or sensor is read and temporarily stored.

- **Data Transfer Between Tasks:** This function block can serve as a simple buffer layer between asynchronous communication paths.

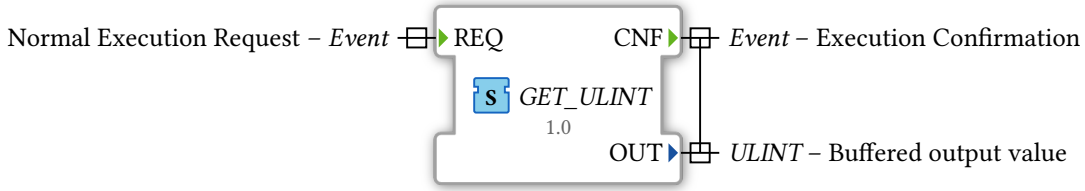
Comparison with Similar Function Blocks

Unlike a pure input function block, which uses a fixed data port, **GET_UINT** allows the use of an **InOut variable** as a source. This allows the variable to be both written to and read from external sources without requiring a separate connection to an input. Pure read function blocks (e.g., UINT input) do not offer this flexibility but have a clearer interface. Compared to a memory function block (e.g., `R_TRIG` or `SR`), **GET_UINT** does not store states, but only the last read value.

Conclusion

GET_UINT is a simple yet useful function block that buffers a UINT value from an InOut variable. Its minimalist design—one event, one action—makes it ideal for situations requiring clean, time-defined value transfer. It is particularly well-suited for applications that need a consistent data set within a flow without affecting the source.

GET_ULINT



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_ULINT** function block reads the current value of a variable of type `ULINT` (unsigned 64-bit integer) provided as an InOut parameter and makes it available as a buffered value at a data output. Upon a request (REQ), the value from the source is read and output at `OUT`. The acknowledgment (CNF) is then sent. Thus, GET_ULINT is a simple read function block for InOut variables.

Interface Structure

Event Inputs

Name	Type	Comment	With Data
`REQ`	Event	Normal Execution Request	`IN`

Event Outputs

Name	Type	Comment	With Data
`CNF`	Event	Execution Confirmation	`OUT`, `IN`

Data Inputs

Name	Type	Comment
-	-	-

Data Outputs

Name	Type	Comment	Initial Value
`OUT`	ULINT	Buffered output value	0

Adapters

No adapters available.

Internal Input/Output Variables

In addition to the inputs/outputs, the function block has an input/output variable:
| Name | Type | Comment | Initial Value | | |—|—|—|—| | `IN` | ULINT | Source variable (InOut) | 0 |

Functionality

This function block processes a single action:

1. An incoming event at input `REQ` triggers the execution of algorithm `REQ`.
2. In the algorithm, the current value of the InOut variable `IN` is copied directly to the output `OUT`:

OUT := IN;`

1. After the assignment is complete, the event `CNF` is output. The outputs `OUT` and `IN` are associated with the event `CNF`, allowing calling blocks to process the updated value.

By using an InOut variable, the block can access memory outside its own context without obtaining it via a conventional input. The value is read fresh with each request, making the block suitable as a read buffer.

Technical Features

- **InOut Variable:** The function block uses a variable named `IN` with the property `InOut`. This allows linking to a parent variable (e.g., from a resource or global data space) without the function block having to obtain the variable via a normal data input. Access is reference-based.
- **ULINT Type:** Both `IN` and `OUT` are declared as unsigned 64-bit integers (ULINT) – suitable for large count values, timestamps, or other values in the range 0 ... 2^{64}-1.
- **Instant Read:** The algorithm is extremely short (one assignment), making the execution time negligible.

State Overview

The function block is implemented as a SimpleFB with a single ECC state:

State	Action	Output Event	Meaning
`REQ`	`REQ` (OUT := IN)	`CNF`	Upon a request, the value is read and acknowledged.

There are no other states, no branches, and no internal memory variables (except for the output `OUT` itself, which buffers the last read value).

Application Scenarios

- **Safe Read Operation:** A higher-level function block wants to retrieve the current value from a shared variable (e.g., a sensor value written by another control component) without permanently referencing the value.
- **Value Buffer for Asynchronous Processing:** In a time-controlled process, `GET_ULINT` can be called to take a snapshot of a global counter variable and buffer it.
- **Parameterization via InOut Interfaces:** If the function block is embedded in a higher-level component, the InOut variable can be directly connected to a data connection of the parent function block, so that read requests point to this variable.

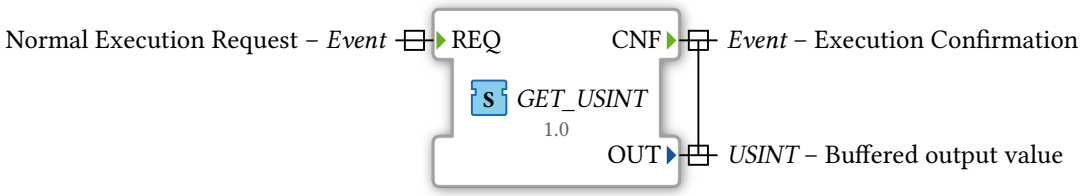
Comparison with Similar Function Blocks

- **GET_DINT / GET_LINT / GET_UINT:** Analogous function blocks for other integer types (DINT, LINT, UINT, etc.). The functionality is identical; the only difference is the data type.
- **READ_ULINT (hypothetical):** A function block that uses a separate data input for the source value instead of an InOut variable. This would have the advantage of loose coupling, but the disadvantage that the source value would have to be carried along as a data stream. The InOut approach of `GET_ULINT` allows a direct reference to a shared variable, which is preferred in some architectures.
- **SAMPLE_ULINT:** A function block that samples and buffers the current value of an input on each event – also similar, but with a normal data input instead of InOut.

Conclusion

The `GET_ULINT` function block provides a simple and efficient way to read the current value of an InOut variable of type `ULINT` and output it as a buffered output. Using an InOut variable allows direct, reference-based access to external memory locations without additional data copying during the call. Thanks to its minimalist state machine, it is ideally suited for time-critical applications where a fast, unobtrusive value read is required.

GET_USINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `GET_USINT` is used to read a `USINT` value from an InOut variable. It buffers the read value and makes it available via an output. This function block is particularly suitable when the value of a variable from a higher-level context (e.g., a group) needs to be queried once and then used as a constant intermediate value in subsequent processes.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Normal execution request. Trigger to read the current value of `IN` and pass it to `OUT`.

Event Outputs

Event	Data Type	Comment
CNF	Event	Confirmation of successful execution. Sent after reading and buffering.

Data Inputs

This function block does not have traditional data inputs. Instead, the value to be read is provided via the InOut variable `IN`.

Variable	Data Type	Comment
IN	USINT	Source variable (InOut). The current value of this variable is read and buffered upon a `REQ` event. The variable can be written to and read from external sources; within the function block, access is read-only.

Data Outputs

Variable	Data Type	Comment
OUT	USINT	Buffered output value. Contains the value read during the last execution of `IN`. Initial value: 0.

Adapter

None.

Functionality

1. The function block waits for a `REQ` event.
2. Upon arrival of `REQ`, the current value of the InOut variable `IN` is read and copied to the local output variable `OUT` (`OUT := IN;`).
3. Subsequently, the acknowledgment event `CNF` is sent.
4. The value of `OUT` is retained until another `REQ` event triggers a new read operation.

Since this is a simple function block, there are no branches or timed sequences – the action is executed in a single step.

Technical Features

- **InOut Variable `IN`** – The interface uses a variable `InOut`, which functions as both an input and an output. This is typical for 4diac function blocks that access variables of a parent group. The block does **not** modify the value of `IN`; it only reads it.
- **Buffering** – Without re-reading `REQ`, the output value remains stable, even if the source variable changes later. This decouples the value from the time it is read.
- **No State Machines** – The block consists of only one state, `REQ`, making the implementation trivial and resource-efficient.

State Overview

The block has exactly one state:

State	Action	Initial Event
REQ	Execute algorithm `REQ` (`OUT := IN;`)	CNF

A start state is not explicitly defined; after initialization, the function block is activated by the first `REQ`.

Application Scenarios

- **One-Time Query of a Configuration Variable** – A `USINT` value defined in a parent resource should be read exactly once at the beginning of the process and then used as a local constant.
- **Decoupling of Read and Write Access** – If a variable is both written to and read from multiple function blocks, `GET_USINT` prevents the read side from being constantly updated, as buffering only occurs on demand.
- **Testing Tool** – This function block can be used to display the current value of an InOut variable in a diagnostic branch.

Comparison with Similar Function Blocks

Function Block	Type	Special Feature
`GET_USINT`	Custom Function Block	Reads an InOut variable and buffers the value. The output remains stable until the next request.
`USINT_TO_*` (e.g., Converter)	Conversion Function Block	Converts a USINT to another type, but does not necessarily read from an InOut.
Direct data access	No function block	A value can be read over data connections without buffering, but without decoupling and without event control.

GET_USINT` is specialized for reading InOut variables with explicit buffering and event control – a task that cannot be accomplished with standard converters or direct connections.

Conclusion

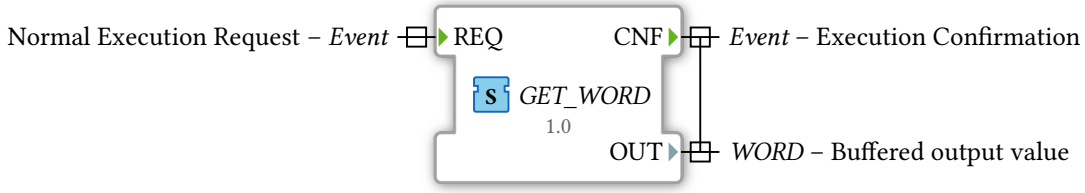
The `GET_USINT` function block offers a simple and efficient way to read a `USINT` value from an InOut variable and provide it as a buffered output. Its clear, event-driven interface and minimal state machine make it a useful tool for decoupling and initialization tasks in 4diac applications. It is ideally suited when the value of a variable should only be retrieved on demand and any changes to the source in the meantime must be ignored.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

GET_WORD



• – ◦ • – ◦ • – ◦ •

Introduction

The **GET_WORD** function block is used to read a WORD value from an InOut variable and make it available as a buffered output. It is typically used to capture a current value from a hardware peripheral or a shared memory location and make it available for further processing in the control program.

Interface Structure

Event Inputs

Event	Description
REQ	Normal execution request; clears the read operation and outputs the result at output `OUT`.

Event Outputs

Event	Description
CNF	Execution confirmation; signals that the value from `IN` was successfully transferred to `OUT`.

Data Inputs

The function block (FB) does not have explicit data inputs in the traditional sense, but uses an **InOut variable**:

Variable	Type	Description
IN	WORD	Source variable (InOut) from which the read value originates. The FB accesses it for reading.

Data Outputs

Output	Type	Description
OUT	WORD	Buffered output value – Copy of `IN` at the time of the last `REQ` execution.

Adapter

No adapters available.

Functionality

1. The function block starts with a **REQ** event at the input.
2. The **REQ** algorithm is executed:

- The current value of the input variable `IN` is copied to the output `OUT`.

1. After successful assignment, the **CNF** event is sent at the output.

Thus, the value read once remains at the output `OUT` until another **REQ** pulse triggers an update. This corresponds to a buffered read operation.

Technical Features

- **InOut Usage** – The function block reads an InOut variable, which is typically connected to a memory or hardware register.
- **Buffering** – The read value is held in the output `OUT`, even if the source `IN` changes later. Only a subsequent **REQ** call updates the buffer.
- **Initialization** – `IN` and `OUT` are pre-set to 0 by default.

State Overview

The function block implements a simple state machine with only one state:

State	Description
REQ	Always active when **REQ** is received; Performs the assignment `OUT := IN` and sends **CNF**.

There are no other states such as IDLE or WAIT, as the logic is strictly event-driven without branching.

Application Scenarios

- **Reading a Hardware Register** – For example, reading an analog-to-digital converter value or a counter value stored as a WORD.
- **Saving a Volatile Value** – If the source is only stable for a short time (e.g., after an interrupt), the function block can freeze the value at a defined moment.
- **Accessing Shared Variables** – In multi-call environments, the current value of a global WORD variable is retrieved and processed locally.

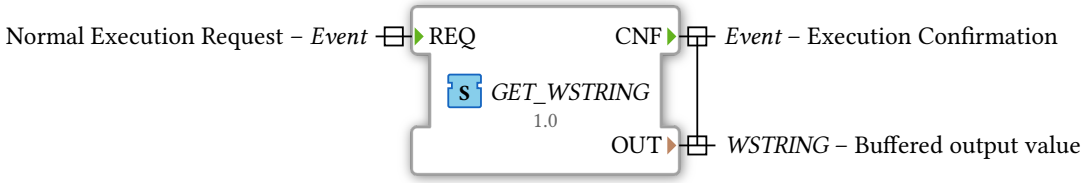
Comparison with Similar Function Blocks

Function Block	Difference
MOVE	Copies a value from a data input to an output. `GET_WORD`, on the other hand, reads from an InOut variable, which often represents a connection to an external resource.
READ	Often standardized for IEC 61499, reads from a process image. `GET_WORD` is more specific for a buffered WORD takeover from an InOut location.
LATCH	Holds a value after a set pulse. `GET_WORD` only updates on each **REQ**, thus resembling a clocked latch, but is not set/resettable.

Conclusion

GET_WORD is a simple but useful function block for selectively reading and buffering a WORD value from an InOut variable. It is particularly suitable for applications where the current state of a source needs to be captured at a specific event time and then made available in a stable manner. The function block (FB) is easy to understand, resource-efficient, and can be seamlessly integrated into event-driven processes.

GET_WSTRING



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **GET_WSTRING** is used to read a WSTRING variable via an InOut parameter and provides the buffered value as a data output. It is typically used to transfer an external WSTRING value into the internal processing of an application module without modifying the original value.

Interface Structure

Event Inputs

Event	Data Type	Description
REQ	Event	Normal execution request – triggers the reading of the InOut variable.

Event Outputs

Event	Data Type	Description
CNF	Event	Confirmation of execution – sent after successful copying.

Data Inputs

Variable	Data Type	Description
IN	WSTRING	InOut source variable whose value is read. (The variable is not modified.)

Data Outputs

Variable	Data Type	Description
OUT	WSTRING	Buffered output value – contains the content of IN read at the time of the REQ event.

Adapter

No adapters available.

Functionality

The function block has a single real-time step (EC state) called `REQ`. As soon as a pulse arrives at the **REQ** event input, the defined algorithm is executed:

1. The current value of the InOut variable **IN** is read and directly assigned to the output variable **OUT** (`OUT := IN`).
2. The **CNF** event is then triggered to signal successful transfer to the calling network.

The InOut variable remains unchanged; only a read access takes place.

Technical Features

- **Input/Output Usage**: The variable `IN` is formally declared as `InOut`. This allows the function block (FB) to access the memory address of the connected source without requiring a separate input variable. The variable can therefore be written to externally as well as read from within the FB.
- **Buffered Output**: The value of `OUT` remains stable until the FB is updated by another **REQ**. This ensures that downstream parts of the application always have access to a consistent value.
- **Type Consistency**: The FB works exclusively with the data type `WSTRING`. Separate function blocks (e.g., `GET_STRING`, `GET_DINT`) are required for other data types.

State Overview

The FB has only one active state:

State	Description
REQ	Initial and operating state. The algorithm is executed and a CNF is sent on every **REQ** event. There are no further branches or wait states.

Application Scenarios

- **Data retrieval from an external WSTRING source** – e.g., reading a global configuration string or a WSTRING value provided by another function block.
- **Decoupling of read and write accesses** – The value from a shared variable can be safely buffered before being processed further in the internal logic.
- **Implementation of a "Get" function block** in an InOut-based architecture, as commonly used in **EC-61499** networking.

Comparison with Similar Function Blocks

Function Block	Data Type	Special Feature
GET_STRING	STRING	Analog function block for ASCII strings (not Unicode).
GET_DINT	DINT	For 32-bit integers.
FORCE	Any type	Forces a value regardless of the source, often with an additional reset mechanism.

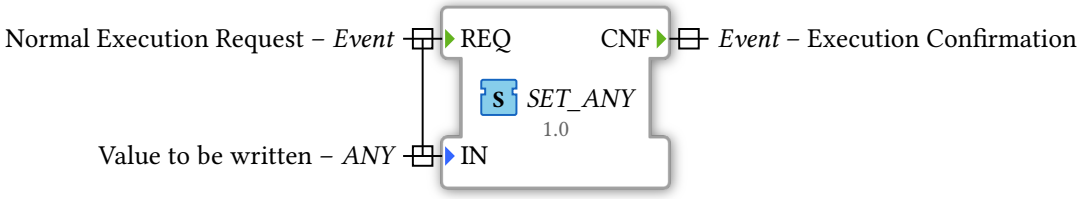
GET_WSTRING differs from these by its specific support for the wide character type (WSTRING), used for Unicode strings (UTF-16). It is a simple function block limited to reading and buffering, with no side effects.

Conclusion

The **GET_WSTRING** function block is a lean, easy-to-understand function block that performs a fundamental task in IEC 61499 environments: the safe reading of a WSTRING InOut variable and the provision

of a buffered output value. Its simple structure and the absence of state branches make it ideal for use in data-driven control tasks where values from an external source need to be retrieved once or cyclically.

SET_ANY



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_ANY** is used to write any value from input `IN` to a variable `OUT`, declared as an InOut parameter. The operation is triggered by an event at input `REQ` and acknowledged with the output event `CNF` after successful assignment. The function block is generic and can work with any data type, as long as the types of `IN` and `OUT` are compatible.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Normal execution request – triggers the assignment. Associated with the variables `IN` and `OUT`.

Event Outputs

Event	Data Type	Comment
CNF	Event	Confirmation of successful execution. Associated with the variable `OUT`.

Data Inputs

Name	Data Type	Comment
IN	ANY	Value written to the target variable.
OUT	ANY	**InOut Parameter** – used as both an input and an output (see Data Outputs).

Note: `OUT` is declared as InOut. It functions as a data input (the original variable is read if necessary) and as a data output (overwritten with the value of `IN`).

Data Outputs

Name	Data Type	Comment
OUT	ANY	**InOut Parameter** – contains the value of the input `IN` after execution. The variable can also be used outside the function block.

Adapter

None.

Functionality

The module has a single processing state, `REQ`. If an event occurs at the event input `REQ`, the algorithm `REQ` is executed:

OUT := IN; After assignment, the output event `CNF` is sent. Execution is atomic – no further state logic is applied.

Technical Features

- **Generic Data Type `ANY`:** The function block can be used with any IEC 61499 data type (e.g., BOOL, INT, REAL, STRING, structures). Type compatibility between `IN` and `OUT` must be ensured by the user.
- **InOut Mechanism:** `OUT` is declared as an InOut parameter. This means that the variable serves as both input (e.g., for read operations) and output. In the 4diac IDE, InOut is typically implemented as a reference to an external variable.
- **Simple State Machine:** The function block contains only one state, making it very lightweight and resource-efficient.

State Overview

State	Description
REQ	Waits for a `REQ` event. Upon occurrence, the algorithm is executed, then `CNF` is sent, and the state remains active.

There are no other states, no branches, and no time monitoring.

Application-Specific Scenarios

- **Variable Initialization:** A one-time initial value is set for a globally or instance-specific variable.
- **Parameter Override:** At runtime, an existing value is overwritten with a new one, e.g., to change setpoints or configuration parameters.
- **Point-to-Point Data Transfer:** When a value needs to be copied from a source to a directly connected destination without requiring separate logic.

Comparison with Similar Function Blocks

Function Block	Difference
SET	Is fixed to a specific data type (e.g., SET_BOOL, SET_INT). `SET_ANY` is generic.
MOVE	Also copies a value from an input to an output, but uses standard outputs. `SET_ANY` uses an InOut parameter that references a persistent variable.
SELECT	Selects between two input values and requires a control signal. `SET_ANY` operates without a condition.

Conclusion

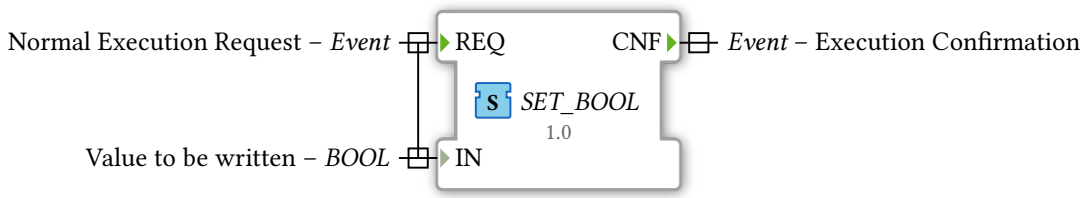
SET_ANY`is a simple yet flexible function block for selectively overwriting variables in IEC 61499. The use of the generic type`ANY` and the InOut parameter makes it universally applicable, especially in scenarios where values need to be written to existing memory locations at runtime. Thanks to its minimal state logic, it is suitable for time-critical applications or as a basic building block for more complex data flow structures.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

SET_BOOL



• – ◦ • – ◦ • – ◦ •

Introduction

The function block SET_BOOL is used to write a Boolean input value (IN) to an InOut variable (OUT). It provides simple assignment logic for BOOL data and is controlled by an event.

Interface Structure

Event Inputs

Name	Type	Comment
REQ	Event	Normal Execution Trigger

Event Outputs

Name	Type	Comment
CNF	Event	Execution Confirmation

Data Inputs

Name	Type	Initial Value	Comment
IN	BOOL	FALSE	Value to be written to the target variable

Data Outputs

No separate data outputs. The output value is provided via the InOut variable OUT.

Adapters

None.

Functionality

When the event REQ is received, the function block executes the assignment OUT := IN. The current value of input IN is copied to the InOut variable OUT. Then, the confirmation event CNF is sent. The InOut variable OUT can be externally assigned to any BOOL variable, allowing the value to be written directly to the corresponding location in the system.

Technical Features

- The function block uses an **InOut variable** (OUT) that functions as both an output and an input. This allows the value to be written directly to an external variable without requiring a separate data output.
- The initial values of IN and OUT are both FALSE, which ensures a defined state even without initialization.
- There is only one algorithm and one state; therefore, the execution logic is deterministic and unconcurrency-free.

State Overview

State	Action	Event Output	Description
REQ	OUT := IN	CNF	On the REQ event, the input value is written to the InOut variable, and execution is confirmed.

The function block has only one active state. After execution, it waits for the next REQ event.

Application Scenarios

- **Setting a Global Flag**: A Boolean flag can be set in another function block or a global variable via an InOut variable.
- **Assigning Values from a Sensor**: A Boolean sensor value (e.g., from a switch) is directly assigned to a control variable.
- **Initializing BOOL Variables**: A defined value can be written to a variable at system startup using REQ.

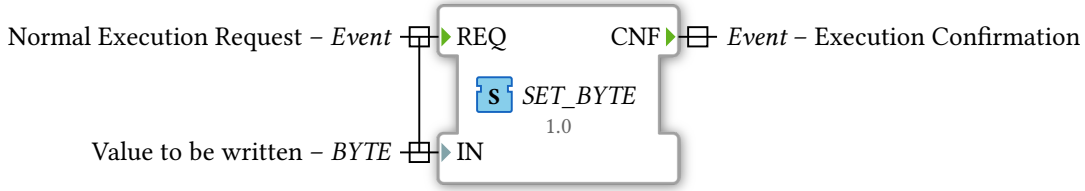
Comparison with Similar Function Blocks

- **`MOVE` (e.g., ANY to ANY)**: `MOVE` copies values of any data type but requires separate inputs and outputs. `SET_BOOL` is BOOL-based and uses an InOut variable, enabling more compact connections.
- **`SET_BOOL` vs. direct assignment in ST**: This function block abstracts the assignment as an event-driven block and can be used in graphical editors (e.g., FBD) where direct assignment is not possible.
- **`R_TRIG` / `F_TRIG`**: These function blocks detect edges and only change the output on signal transitions. In contrast, `SET_BOOL` performs an unconditional assignment on every `REQ` event.

Conclusion

SET_BOOL` is a simple yet practical function block for directly assigning a Boolean value to a variable using an InOut interface. It is ideally suited for control tasks where a Boolean state needs to be set and integrates seamlessly into the event-driven paradigm of IEC 61499.

SET_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_BYTE** is used to easily overwrite a BYTE variable. It reads the value of the data input `IN` and writes it to the InOut variable `OUT` as soon as an event arrives at the input `REQ`. The event `CNF` is then output.

Interface Structure

Event Inputs

Name	Type	Comment
`REQ`	Event	Normal execution job (start write operation)

Event Outputs

Name	Type	Comment
`CNF`	Event	Confirmation of successful assignment

Data Inputs

Name	Type	Value Range	Initial Value	Comment
`IN`	BYTE	0…255	0	Value to be written to the target variable

Data Outputs

The function block uses an **InOut** variable. Therefore, there are no separate data outputs – the output value is identical to the value of the InOut variable `OUT`.

Adapter

None present.

Functionality

When an event occurs at input `REQ`, the algorithm `REQ` is executed. This assigns the value of `IN` to the InOut variable `OUT`:

OUT := IN; Immediately after the assignment, the event `CNF` is sent. The InOut variable can be both read and written from outside the function block (e.g., via a connection in the application model). After the write operation, it has the value `IN`.

Technical Features

- **Simplest Assignment**: The algorithm consists of only one line – no delays, conditions, or filters.
- **InOut Variable**: `OUT` is declared as InOut, meaning it can be connected externally to a variable that can be both read and written. This allows direct manipulation of a global or linked BYTE variable.
- **No State Machine**: The function block has only one implicit state (`REQ`) and requires no additional state logic.

State Overview

The function block (FB) has no explicit states. Action is initiated immediately upon the event `REQ`. There are neither error nor delay states.

Application Scenarios

- **Initialization or Reset**: A specific initial value is to be written to an existing byte variable.
- **Switching/Setting Bit Patterns**: An external signal (e.g., from a communication block) is passed to an output or memory location via `IN`.
- **Testing/Diagnostics**: Temporarily overwriting a value in a control application.

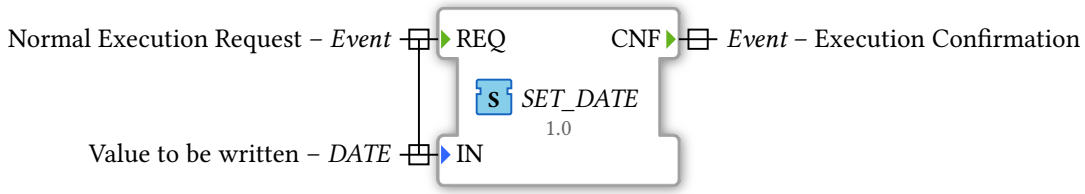
Comparison with Similar Function Blocks

- **MOVE** (general): The simple MOVE function block for arbitrary data types usually works with an input and an output, not with InOut. `SET_BYTE` is specifically optimized for BYTE and uses InOut, so the value is written directly to the target variable (no separate output is needed).
- **SET_BOOL** (if available): Analogous for BOOL – both function blocks follow the same simple principle but are limited to their respective data types.

Conclusion

SET_BYTE` is a minimalist, clearly defined function block for writing to a BYTE variable once. Its strength lies in its simplicity and the use of an InOut interface, which allows for direct value assignment without additional wiring of a data output. It is particularly suitable for applications where a predefined value needs to be written to a persistent variable at a specific time.

SET_DATE



• – ◦ • – ◦ • – ◦ •

Introduction

The **SET_DATE** function block is used to write a date value (data type `DATE`) from an input to an InOut variable. The InOut variable is written directly and is available with the new value after execution. This enables efficient transfer of date values without an additional output variable.

Interface Structure

Event Inputs

Event	Description	With Variables
REQ	Normal Execution Request	IN, OUT

Event Outputs

Event	Description	With Variables
CNF	Execution Confirmation	OUT

Data Inputs

Variable	Type	Description
IN	DATE	Value to be written to the target variable
OUT	DATE	Target variable (InOut) – initial value can be read but will be overwritten

Data Outputs

Variable	Type	Description
OUT	DATE	Target variable (InOut) – contains the value of IN after execution

Adapter

None.

Functionality

When the **REQ** event is triggered, the function block executes the algorithm `REQ`:

- The value of the input `IN` is assigned to the InOut variable `OUT` (`OUT := IN;`).
- The **CNF** event is then sent to signal successful completion.

The InOut variable `OUT` serves as both an input and an output. The current value can be read before execution, and the new value is available after execution.

Technical Features

- **InOut Variable:** The function block uses an InOut variable that is referenced directly in the calling environment. Changes take effect immediately on the connected variable without any additional data output.
- **Data Type:** Designed exclusively for the data type `DATE`.
- **Simple State Logic:** The function block consists of only one state, which directly performs the assignment.

State Overview

State	Description
REQ	Executes the value assignment `OUT := IN` and sends CNF

No other state exists; the function block is immediately ready for execution.

Application Scenarios

- Setting a date value in a PLC controller, e.g., initializing a current date.
- Transferring a calculated or parent date to a global or parent variable.
- Switching between different date values by repeated calls with different `IN` values.

Comparison with similar function blocks

Similar function blocks exist for other data types:

- **SET_TIME** (for `TIME`)
- **SET_DT** (for `DATE_AND_TIME`)

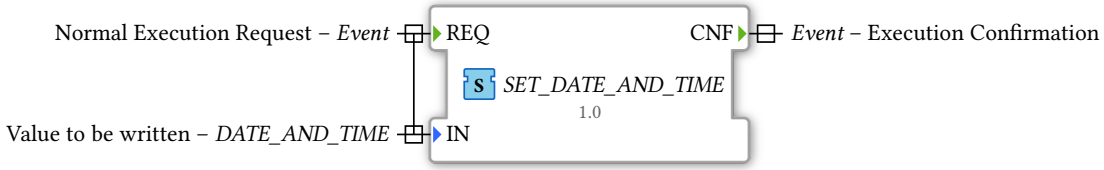
The main difference lies in the data type used. The functionality and the InOut interface are identical.

Conclusion

SET_DATE is a simple yet useful function block for directly setting a date variable. By using an InOut variable, memory and communication overhead are kept to a minimum. The function block is particularly

suitable for standardization tasks in automation technology where date values need to be transferred efficiently and reliably.

SET_DATE_AND_TIME



• – ◦ • – ◦ • – ◦ •

Introduction

The SET_DATE_AND_TIME function block is used to write an incoming DATE_AND_TIME value to a target variable. The target variable is passed as an InOut parameter, allowing it to be both read and written. The function block implements a simple assignment operation triggered by an event.

Interface Structure

Event Inputs

Event	Description
REQ	Normal execution request; starts writing the input value to the InOut variable.

Event Outputs

Event	Description
CNF	Confirmation of successful execution.

Data Inputs

Name	Type	Description
IN	DATE_AND_TIME	The value to be written to the target variable.

Data Outputs

This function block does not have explicit data outputs. The updated value is made visible externally via the InOut parameter `OUT`.

Adapters

None.

Functionality

As soon as an event occurs at input `REQ`, the ST instruction `OUT := IN` is executed. The value of input `IN` is copied directly into the InOut variable `OUT`. Upon successful completion, output `CNF` is triggered to signal completion. Using an InOut variable allows the function block to be inserted into an existing data connection without requiring a separate output variable.

Technical Features

- **InOut Variable:** The parameter `OUT` is declared as InOut. This allows the function block to directly overwrite the value of the associated variable while remaining externally readable.
- **Simple Assignment:** This is a simple copy operation without type conversion or validation checks.
- **No State Machines:** The function block is implemented as a SimpleFB with only one EZ state (REQ). Execution occurs strictly within a single cycle.

State Overview

The function block has a single state:

State	Action	Output Event
REQ	`OUT := IN`	CNF

The state machine starts in this state upon receiving `REQ`, executes the action, and immediately sends `CNF`.

Application Scenarios

- **Initializing a System Time:** A higher-level function block can provide the current time (e.g., from an interface) via `IN` and write it to a globally or persistently stored time variable (InOut) via `SET_DATE_AND_TIME`.
- **Resetting a Time Variable:** When setting a reference time, the function block can be used to assign a predefined value to a higher-level variable.
- **Parameterization:** In control/regulation applications, the function block can be used to write target time values to data blocks.

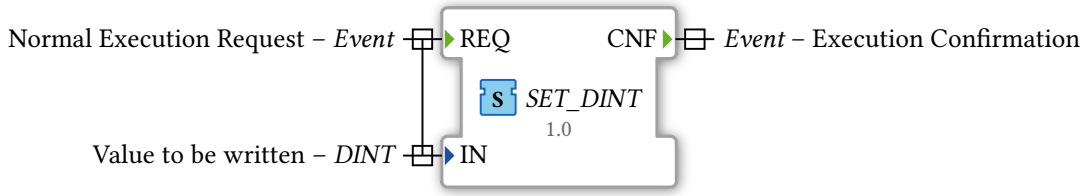
Comparison with Similar Function Blocks

Simple assignment function blocks for other data types (e.g., `SET_VALUE` for integer or boolean) follow the same pattern. The `SET_DATE_AND_TIME` function block is specifically designed for use with the `DATE_AND_TIME` type and does not offer advanced functions such as addition or subtraction. Unlike a `MOVE` block (which maps an input to an output), this block uses an InOut variable, eliminating the need to separate the source and destination.

Conclusion

The `SET_DATE_AND_TIME` function block is a simple yet useful block for overriding a date and time variable within an IEC 61499 system. Its use of an InOut interface allows for easy integration into existing data flows. Due to its straightforward structure, it is efficient and suitable for all scenarios requiring the targeted assignment of a date and time value.

SET_DINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SET_DINT` is used to write a DINT value from a data input to an InOut variable. It thus implements a simple assignment operation for the data type `DINT`. The block is triggered by an event and acknowledges successful execution with a confirmation event.

Interface Structure

Event Inputs

Name	Type	Comment	With Variables
REQ	Event	Normal Execution Trigger	IN, OUT

Event Outputs

Name	Type	Comment	With Variables
CNF	Event	Execution Confirmation	OUT

Data Inputs

Name	Type	Comment	Initial Value
IN	DINT	Value to be written	0

Data Outputs

This function block does not have traditional data outputs, but rather an InOut parameter (see below). The acknowledgment event `CNF` is linked to the output `OUT`, so that after execution, the updated value is available at the output.

Adapter

None.

Functionality

When an event occurs at the input `REQ`, the algorithm `REQ` is executed. This copies the current value of the data input `IN` directly into the InOut variable `OUT`. The acknowledgment event `CNF` is then sent.

The logic in Structured Text is:
OUT := IN; Execution is atomic: The value is applied immediately and without delay.

Technical Features

- The function block uses an **InOut variable** (`OUT`). This variable is both an input and an output and can be read and written externally. This allows the function block to overwrite the value of an existing variable in place without using an additional output.
- The function block is stateless – there is only one execution state. Every call performs the same simple assignment.
- All variables are initially predefined with `0`.

State Overview

State	Action	Output Event
REQ	OUT := IN	CNF

The function block has no other states (no wait loops, no branches). After the action, it immediately returns to standby mode for the next `REQ`.

Application Scenarios

- **Parameter Passing:** An external value (e.g., from a configuration interface) is written to a local variable in an application.
- **Signal Routing:** A digital integer value is redirected from one signal path to another, while preserving the original value.
- **Initialization:** At system startup, a defined start value can be written to a state variable via `SET_DINT`.
- **Testing and Debugging:** Manually setting a DINT value via an HMI or a script.

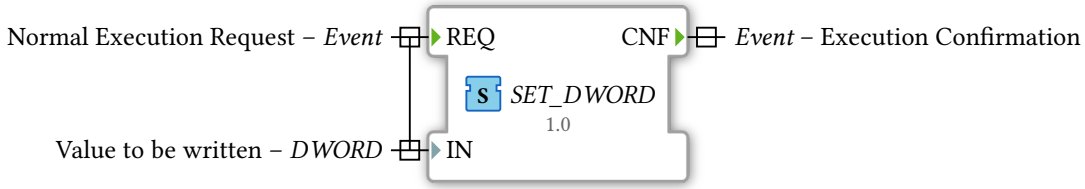
Comparison with Similar Function Blocks

- **MOVE/DINT**: Standardized function blocks such as `MOVE` from IEC 61131-3 also perform assignments. `SET_DINT` is specialized for the type `DINT` and offers a reduced interface (only one input event, no additional outputs).
- **SET_BOOL, SET_REAL, SET_SINT**: Analogous function blocks for other data types exist in the `logiBUS` library. They differ only in data type; their functionality is identical.
- **Direct Assignment in the Network**: In IEC 61499, an assignment could also be implemented without a function block via a direct data connection. The function block `SET_DINT`, however, offers explicit, event-driven encapsulation that can be monitored and triggered at runtime.

Conclusion

SET_DINT` is a simple yet useful function block for the targeted assignment of DINT values. By using an InOut variable, the interface remains streamlined, and the block can be directly integrated into existing data flows. Its clear, stateless logic makes it reliable and easy to understand – ideal for basic transfer operations in automation projects.

SET_DWORD



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SET_DWORD` is used to write a DWORD value to a target variable declared as InOut via an input. It provides a simple and efficient way to set values during operation without requiring a separate assignment in the program.

Interface Structure

Event Inputs

Name	Type	Comment
REQ	Event	Normal Execution Request – triggers the write operation

The event `REQ` simultaneously provides the data `IN` and `OUT`.

Event Outputs

Name	Type	Comment
CNF	Event	Execution Confirmation – sent after successful execution

The output `CNF` references the modified variable `OUT`.

Data Inputs

Name	Type	Comment
IN	DWORD	Value to be written to the target variable (initial value: 0)

Data Outputs

Name	Type	Comment
OUT	DWORD	Target variable (InOut) – the value of `IN` is written here (initial value: 0)

Note: `OUT` is declared as an InOut variable, meaning it can be referenced externally and its memory location is directly modified by the function block. It appears in the function block network as both an output and an input.

Adapters

No adapters available.

Functionality

The function block has a single action that is executed when the event `REQ` occurs:

```structured text
OUT := IN;
```

The event `CNF` is then sent. The function block is stateless and executes the assignment on every `REQ` event, regardless of the current value.

Technical Features

- **InOut Variable**: Using an InOut variable allows direct access to the memory location of an external variable. This reduces data copying overhead and enables the modification of variables that are also used elsewhere in the system.

- **No State Logic**: The function block does not implement a state machine with multiple states – it always reacts the same way to the input event.
- **Initial Values**: Both variables are initially set to 0, but are usually overwritten at runtime.

State Overview

The function block consists of a simple state machine with exactly one state:

State	Triggering Event	Action	Output
REQ	REQ	OUT := IN;	CNF

There are no wait states or branches.

Application Scenarios

- **Peripheral Configuration**: Setting a DWORD register in a module (e.g., time base, threshold).
- **Parameter Passing**: Passing a constant or calculated value to a globally available variable (e.g., via an InOut data type).
- **Testing and Debugging**: Temporarily overwriting a value during operation without modifying the source code.
- **Plain Text Assignment Migration**: Replacing a simple `OUT := IN` instruction in an IEC 61499 network with a reusable function block.

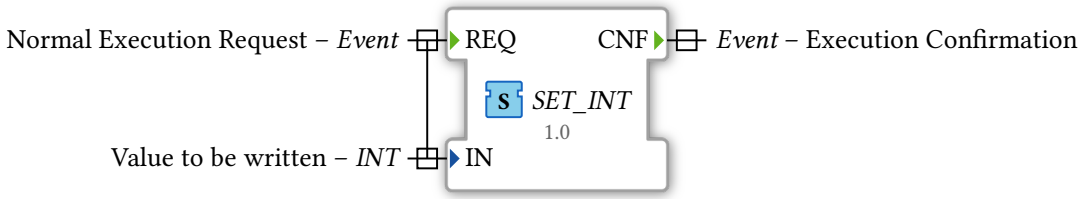
Comparison with Similar Function Blocks

- **MOVE / MV**: Available in many libraries, but usually work with simple input/output variables and require explicit buffering. `SET_DWORD`, on the other hand, directly accesses an existing variable (InOut).
- **SET_BOOL / SET_INT / SET_REAL**: Analogous function blocks for other data types; the principle is identical, only the data type varies.
- **WRITE_DWORD**: Some libraries offer a function block that writes via an address bus. `SET_DWORD` is simpler and does not require address logic.

Conclusion

SET_DWORD` is a minimal but very useful function block for directly assigning values to a DWORD InOut variable. Its simplicity makes it error-resistant, and the absence of state logic ensures predictable real-time behavior. It is particularly suitable for configuration and initialization tasks where fast, straightforward value assignment is required.

SET_INT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_INT** is used to write a value of data type `INT` from an input directly into an InOut variable. This block is a simple, event-driven function block and is typically used to set global parameters or variables of other function blocks via a defined event pulse. Its functionality is reduced to the minimal task of value assignment.

Interface Structure

Event Inputs

Event	Data Type	Description	With Data
REQ	Event	Normal Execution Request. Triggers the transfer of the value from IN to OUT.	IN, OUT

Event Outputs

Event	Data Type	Description	With Data
CNF	Event	Execution Confirmation. Sent after the value has been successfully written to OUT.	OUT

Data Inputs

Variable	Data Type	Description	Initial Value
IN	INT	The value to be copied to the InOut variable.	0

Data Outputs

The function block does not have a dedicated data output. Output is provided via the InOut variable (see below).

Adapter

None.

InOut Variable

Variable	Data Type	Description	Initial Value
OUT	INT	Target variable into which the value of IN is written. This variable is declared as InOut, meaning it is referenced from outside the function block and can be directly manipulated.	0

Functionality

The function block operates according to a very simple, direct principle:

1. An event is received at input REQ.
2. The algorithm REQ executes the assignment `OUT := IN`. The current value of the data input IN is thus copied into the InOut variable OUT.
3. After successful assignment, the event CNF is sent to signal completion to the calling environment. The CNF event is linked to the InOut variable OUT, ensuring data consistency.

Execution occurs without delay or conditions – the value is overwritten immediately upon each REQ event.

Technical Features

- **InOut Variable**

The function block uses an InOut parameter (OUT) that represents a direct reference to an external variable (e.g., a global data block or a variable of a parent function block). This eliminates the need for an explicit connection back to another function block. The function block writes its value directly to the referenced memory location.

- **No Buffering**

The function block (FB) does not buffer data. The value is immediately overwritten with each REQ-Ereignis event.

- **Initial Values**

The data inputs and InOut variables are initialized with 0. This value is used in case of a lost connection.

State Overview

The FB has a single ECC state (REQ) that executes the action and immediately triggers the output event CNF. There are no branches, timers, or error states.

State	Action	Trigger
REQ	Execution of the algorithm REQ (OUT := IN) and sending of CNF	Event REQ

Application Scenarios

- **Setting a Constant Value**

An external sensor or logic provides a setpoint (`IN`), which is transferred to a global setpoint variable (`OUT`) at a clock signal (`REQ`).

- **Initializing Variables**

When an application starts, `SET_INT` can be used to set multiple InOut variables to defined initial values.

- **Parameter Override**

A higher-level controller or user interface overrides the current value of another function block upon a specific event, without having to modify the function block itself (the InOut connection allows direct access).

Comparison with Similar Function Blocks

- **SET_BOOL / SET_REAL**

Similar to `SET_INT`, but for the data types `BOOL` and `REAL`, respectively. The functionality is identical.

- **MOVE**

A general MOVE function block copies a value from a data input to a data output. In contrast, `SET_INT` uses an InOut variable and writes directly to an external location, resulting in tighter coupling. `MOVE` requires a separate connection from the output to the destination.

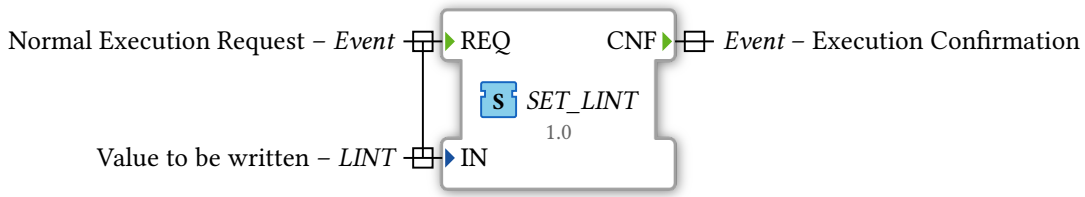
- **LATCH**

A latch function block can store a value and output it when needed. In contrast, `SET_INT` overwrites without a hold function – each call overwrites the previous value.

Conclusion

The `SET_INT` function block is a minimalist yet effective building block for directly setting an INT variable via an InOut interface. Its strength lies in its simple logic and the ability to access external variables. It is particularly suitable for flat control structures where fast and direct value assignments are required. Due to its simplicity, it is robust and reliable, but requires a well-thought-out variable concept to avoid unwanted side effects.

SET_LINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_LINT** is used to write a value of type `LINT` (64-bit integer) from a data input to a variable declared as an InOut parameter. It is typically used to explicitly set the value of a variable defined outside the block. The block complies with the IEC 61499-1 standard and provides simple, event-driven write access.

Interface Structure

Event Inputs

Name	Type	Description	With Variables
REQ	Event	Triggers Write Operation	IN, OUT

Event Outputs

Name	Type	Description	With Variables
CNF	Event	Confirmation after successful write	OUT

Data Inputs

Name	Type	Initial Value	Description
IN	LINT	0	The value to be written

Data Outputs

This function block does not have traditional data outputs, but rather an **InOut parameter** `OUT`, which functions as both an input and an output. This parameter is the target memory location for the value of `IN`.

Adapter

None.

Functionality

The behavior of the function block is reduced to a single step:

1. An event at the **REQ** input triggers the internal processing.
2. The algorithm executes the assignment `OUT := IN`. This copies the value of the data input `IN` to the InOut variable `OUT`.
3. After the assignment is complete, the **CNF** event is sent to signal successful execution.

The InOut variable `OUT` must be bound to a memory location (e.g., a global or device-specific variable) in the application. The function block overwrites the value stored there with the value of `IN`.

Technical Features

- The function block uses an **InOut parameter** instead of a separate data output. This allows direct writing to an external variable without requiring additional wiring via a data output.
- The data type `LINT` (64-bit signed integer) is platform-dependent; the implementation requires that the target system supports this type.
- There is no error handling or range checking – the assignment always occurs as soon as `REQ` is triggered.
- The function block is a **SimpleFB** with only one ECR state; therefore, its behavior is deterministic and fast.

State Overview

The function block has only one state, `REQ`. After the algorithm executes, it automatically returns to its initial state (implicitly). A state machine in the classical sense does not exist – the function block is stateless. Exactly one write operation is performed after each `REQ` event.

Application Scenarios

- **Initializing LINT Variables**: Setting a global configuration variable to a defined initial value.
- **Changing Control Parameters**: Dynamically overriding a parameter (e.g., setpoint or limit value) by a higher-level function block.
- **Test and Debug Interfaces**: Enables manual setting of variables at runtime via an HMI or script.
- **Message Header/Payload Setting**: In communication, `IN` can contain the value of a message, which is written to a buffer (`OUT`).

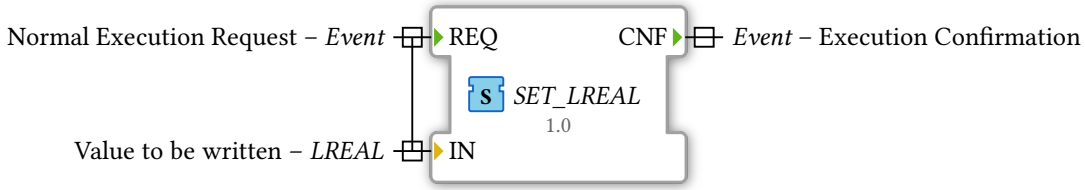
Comparison with Similar Function Blocks

- **SET_DINT / SET_INT**: Analogous function blocks for `DINT` (32-bit) and `INT` (16-bit), respectively. The only difference is the data type.
- **MOVE**: A more general function block for copying arbitrary values, often also with InOut parameters. `SET_LINT` is specialized for `LINT` and offers no further data or event inputs.
- **WRITE_LINT (hypothetical)**: A function block that explicitly writes an address via a pointer. `SET_LINT` operates without addressing and expects a directly linked variable.

Conclusion

SET_LINTis a minimalist function block that reliably performs a single task: assigning aLINT` value to an InOut variable. Due to its simplicity, it is particularly suitable for performance-critical or manageable control tasks where no complex logic is required. The absence of error handling makes it transparent and predictable, but requires the user to correctly link the InOut variable.

SET_LREAL



• – ◦ • – ◦ • – ◦ •

The function block **SET_LREAL** is used to pass a value of type LREAL to an InOut variable. The input value is written to the target variable on a rising edge of the REQ event, and the operation is acknowledged via the CNF event.

Event	With Variables	Description	
REQ	IN, OUT	Normal execution request. Triggers writing the input value to the target variable.	
Event	With Variables	Description	
–––––	–––––	–––––	
CNF	OUT	Confirmation of successful execution. The target variable has been updated.	
Name	Type	Initial Value	Description
––	–––	–––––	–––––––––––––––
IN	LREAL	0.0	Value to be written to the target variable.
OUT	LREAL	0.0	(InOut) Input side of the target variable. The current value can be read.
Name	Type	Initial Value	Description
––	–––	–––––	–––––––––––––––––
OUT	LREAL	0.0	(InOut) Output side of the target variable. After execution, the value of IN is output here.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter
No adapters available.

Functionality
When an event occurs at the **REQ** input, the algorithm is executed, copying the value of the input variable **IN** to the InOut variable **OUT**. The **CNF** event is then sent. The function block operates purely combinatorially and has no internal state maintenance beyond execution.

- The function block uses an InOut variable. This allows direct access to an external variable without requiring an additional connection between the input and output.
- The algorithm is implemented in Structured Text (ST) and is very simple (`OUT := IN;`).

The function block consists of a single state, **REQ**:

- **REQ**: Waits for the arrival of the REQ event. Then executes the algorithm and sends CNF. After sending, it returns to the REQ state (implicitly, as no other states are defined).

There are no branches or timing controls.

- **Copying an LREAL value** to a global or cross-instance variable when a direct connection is not possible or desired.
- **Setting a setpoint** in a controller, e.g., for a PID controller where the value is received via an analog input.
- **Initialization** of variables during operation, without having to restart the entire application.
- **SET_INT** or **SET_BOOL**: Functionally equivalent function blocks for other data types.
- **MOVE**: A general MOVE function block (e.g., in IEC 61131-3) copies values without event control. SET_LREAL, on the other hand, uses event-driven execution and InOut variables.
- **WRITE_LREAL**: Some libraries offer separate function blocks for writing to an address. SET_LREAL is simpler and operates at the variable level.

The **SET_LREAL** function block is a simple yet useful tool for writing an LREAL value to an InOut variable. Its event-driven operation makes it easy to integrate into IEC 61499 applications. The clear interface with only one event input and output, one data input and one InOut variable reduces complexity and facilitates reuse.

Technical Features

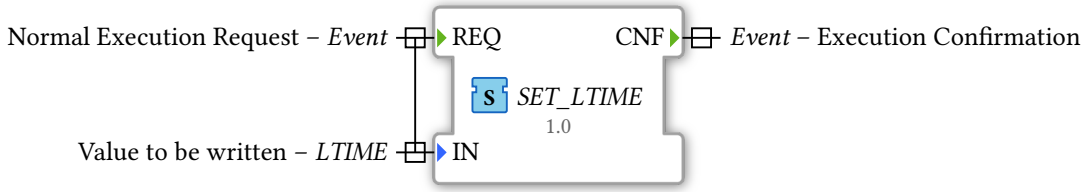
State Overview

Application Scenarios

Comparison with Similar Function Blocks

Conclusion

SET_LTIME



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_LTIME** is used to write an input value of type `LTIME` to a variable declared as InOut. This allows you to directly set the value of an external reference (e.g., an instance variable or another function block). Using an InOut variable allows for efficient data transfer without requiring a separate data output.

Interface Structure

Event Inputs

Name	Type	Comment
REQ	Event	Normal execution trigger. Associated with the data `IN` and `OUT`.

Event Outputs

Name	Type	Comment
CNF	Event	Confirmation of successful execution. Associated with data `OUT`.

Data Inputs

Name	Type	Initial Value	Comment
IN	LTIME	LT#0s	Value to be written to the target variable.

Data Outputs

Name	Type	Initial Value	Comment
OUT	LTIME	LT#0s	**InOut Variable** – Target variable that receives the passed value. The function block writes the value of `IN` to this variable. As an InOut variable, it can be both read from and written to externally.

Adapter

None.

Functionality

When an event occurs at the **REQ** input, the algorithm `REQ` is executed:

- The value of the data input `IN` is copied directly into the InOut variable `OUT`.

- The **CNF** event is then output to signal the successful assignment.

The logic is very simple and consists of a single assignment:
OUT := IN;

Technical Features

- The function block (FB) works exclusively with the **LTIME** data type (high-resolution time, according to IEC 61499).
- The target variable `OUT` is declared as **InOut**. This means that it is passed externally as a reference. Changes made by the FB to `OUT` immediately affect the referenced variable.
- Initially, both `IN` and `OUT` are set to `LT#0s` to prevent undefined states.
- The function block is a **simple FB** with only one state, which reacts immediately to a REQ event.

State Overview

The FB has exactly one state:
| State | Action | Output | | State | Action | Output | |———|————|———| | REQ | `REQ` (OUT := IN) | CNF |
There are no branches or timers. The function block is essentially combinatorial, as it reacts immediately to the input event.

Application Scenarios

- **Setting a Central Time Base**: If multiple function blocks access a common LTIME variable, `SET_LTIME` can update this variable.
- **Runtime Configuration**: In a control application, target time values from a configuration interface can be written to the corresponding InOut parameters.
- **Testing and Simulation**: This function block is suitable for selectively writing values to an observable variable in test environments.

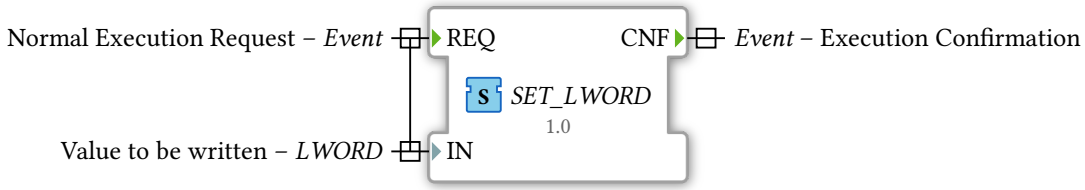
Comparison with Similar Function Blocks

- **SET_TIME**: Works analogously, but with the data type `TIME` (second resolution). `SET_LTIME` offers higher time resolution (nanoseconds) and is therefore suitable for precise applications.
- **MOVE Function Blocks**: Many IEC 61499 libraries contain generic MOVE function blocks that copy values from an input to an output. `SET_LTIME` specializes in `LTIME` and uses an InOut reference instead of a separate output, which can be advantageous in certain architectures.

Conclusion

SET_LTIME` is a minimalist yet useful function block for precisely setting an LTIME variable. Its clear interface and the use of an InOut reference make it an efficient tool in timing and configuration scenarios. The component is easy to understand and straightforward to use.

SET_LWORD



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_LWORD** is used to transfer an LWORD value (64-bit integer) from a data input to an InOut variable. It allows the direct setting of a target value that simultaneously functions as an input and output. This makes the block suitable for use in systems where a specific memory location or a connection variable needs to be updated.

Interface Structure

Event Inputs

Event	Data Type	Comment
REQ	Event	Normal Execution Trigger

Event Outputs

Event	Data Type	Comment
CNF	Event	Execution Confirmation

Data Inputs

Name	Data Type	Initial Value	Comment
IN	LWORD	0	Value to be written

Data Outputs

Name	Data Type	Initial Value	Comment
OUT	LWORD	0	Destination variable (declared as InOut)

Note: OUT is declared in the XML as `<InOutVars>`. It is a combined input/output variable that serves both as a source (read) and a destination (write).

Adapter

None.

Functionality

When an event is received at the **REQ** input, the function block executes the following algorithm:

OUT := IN; The value of the data input **IN** is directly transferred to the InOut variable **OUT**. The **CNF** event is then sent to confirm successful execution.

The function block is a simple, stateless function block – there are no delays or concurrency issues.

Technical Features

- **InOut Variable:** The variable `OUT` is declared as InOut. This means it can be used in the network as both an input (to read an existing value) and an output (to set a new value). The algorithm overwrites the contents of the connected variable with the input value.
- **Type LWORD:** The function block is designed for 64-bit values. When used with shorter data types (e.g., DWORD), an explicit type conversion is required.
- **Initial Value:** Both `IN` and `OUT` are preset to 0.

State Overview

The function block has only one EC state:

- **REQ:** Executes the algorithm and then sends the CNF event.

There are no wait states or loops.

Application Scenarios

- **Initializing Memory Locations:** Setting a 64-bit register or a global variable to a defined initial value.
- **Copying Values in InOut Connections:** If another function block provides an LWORD variable as an InOut, SET_LWORD can selectively overwrite it.
- **Test and Debugging Environments:** Manually specifying values in data paths to verify system behavior.

Comparison with Similar Function Blocks

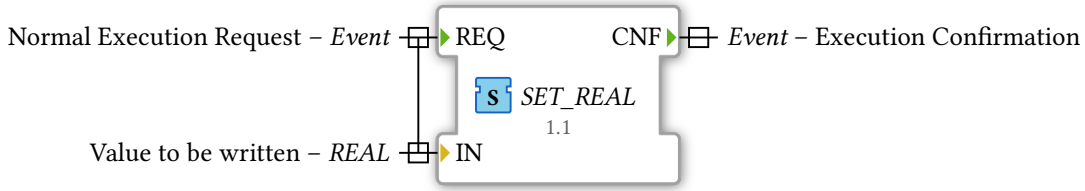
- **SET_BOOL / SET_DWORD / SET_SINT:** Analogous function blocks for other basic data types (BOOL, DWORD, SINT, etc.). SET_LWORD covers the 64-bit integer range.

- **MOVE:** A general function block that copies values between any data type, but often without InOut support. SET_LWORD specifically uses the InOut declaration and is therefore optimized for direct variable assignments.
- **WRITE_***: Some libraries contain function blocks for writing to peripheral addresses. SET_LWORD is more abstract and operates at the logical variable level.

Conclusion

The function block **SET_LWORD** is a lean but essential tool for setting a 64-bit value to an InOut variable. Its simple semantics ("write IN to OUT") and support for the standard LWORD data type make it versatile for use in IEC 61499-based automation systems – especially where direct manipulation of variables via event control is required.

SET_REAL



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_REAL** is used to write a REAL value from a data input to an InOut variable. It is a simple, event-driven function block that, upon receiving a request (REQ), passes the current input value IN to the referenced variable OUT and acknowledges the operation with a confirmation event (CNF).

Interface Structure

Event Inputs

Name	Type	Description
REQ	Event	Normal execution request (connected to IN and OUT)

Event Outputs

Name	Type	Description
CNF	Event	Execution Confirmation (connected to OUT)

Data Inputs

Name	Type	Initial Value	Description
IN	REAL	0.0	Value to be written
OUT	REAL (InOut)	0.0	Target variable (InOut – read and write)

Data Outputs

No separate data outputs are available. The InOut parameter OUT also functions as an output and is updated after assignment.

Adapters

None.

Functionality

The FB has a single execution state **REQ**. As soon as an event arrives at the **REQ** input, the defined algorithm is executed:

OUT := IN; The current value of IN is assigned to the InOut variable OUT. The **CNF** event is then triggered to signal the completion of the operation. The process is atomic and requires no further state transitions.

Technical Features

- **InOut Parameter OUT**: The variable OUT is both readable and writable. It is directly referenced by the calling environment, so the assignment immediately changes the value at the original address.
- **Initial Values**: Both IN and OUT are initialized to 0.0. This prevents undefined states before the first execution.
- **Simplicity**: The function block consists of only one state and one algorithm, guaranteeing very low runtime complexity.

State Overview

State	Description	Action	Output
REQ	Wait for execution request	Execute algorithm REQ ⟶ OUT := IN	CNF

No further state exists; the function block immediately returns to the waiting REQ state after execution.

Application Scenarios

- **Parameter setting**: Passing a REAL value to a global configuration variable, e.g., for scaling or setpoints.
- **Initialization**: Setting an InOut variable to a defined start value during commissioning.
- **Value copy within an application**: Easily overwriting an existing REAL variable with a different value, triggered by an event.

Comparison with similar function blocks

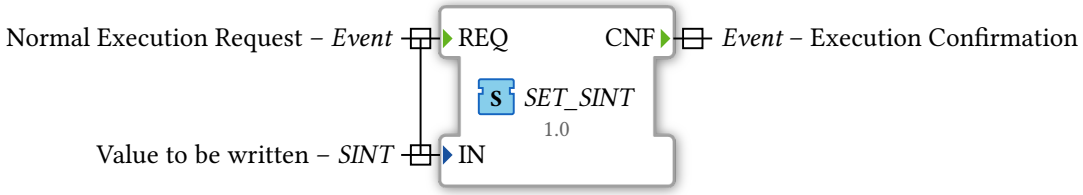
FB	Type	Difference
SET_BOOL	BOOL	Writes Boolean values, analogous functionality
SET_INT	INT	Writes integers, identical event control
MOVE (generic)	ANY	Can copy any data type, but requires type matching and often has multiple states
SET_REAL	REAL	Specifically optimized for REAL, minimal states and clear semantics

Compared to a generic MOVE block, SET_REAL offers a leaner implementation and is tailored to the frequently used REAL type.

Conclusion

The **SET_REAL** function block is a simple yet effective block for setting a REAL variable. Thanks to its minimal state machine and direct use of InOut parameters, it is ideally suited for event-driven assignments in IEC 61499 applications. It reduces complexity and increases the readability of the control code.

SET_SINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_SINT** is used to write a SINT value from its data input **IN** to an InOut variable **OUT**. It is used to set a SINT value at a specific point in the control sequence. The block implements a simple, event-driven assignment.

Interface Structure

Event Inputs

Event	Description
REQ	Normal execution request. Triggers the assignment from **IN** to **OUT**.

Event Outputs

Event	Description
CNF	Execution confirmation. Sent after successful assignment.

Data Inputs

Name	Type	Initial Value	Description
IN	SINT	0	The value to be written.

Data Outputs

The function block has no separate data outputs. The target value is returned via the InOut variable **OUT**.

Adapters

No adapters are available.

Functionality

A **REQ** event starts the execution of the algorithm. Within the algorithm, the current value of **IN** is directly assigned to the InOut variable **OUT**:

OUT := IN; After the assignment is complete, the **CNF** event is triggered. The InOut variable **OUT** serves as both input and output – its value is overwritten and then available for subsequent function blocks.

Technical Features

- **InOut Variable**: The variable **OUT** is declared as `InOut`. It can be both read and written. The function block overwrites the existing value of **OUT** with the value of **IN**.
- **Simplest Algorithm**: The function block performs only a single assignment. There are no conditions, loops, or additional logic.
- **Type Safety**: Both **IN** and **OUT** are of type `SINT`, ensuring a direct, type-compliant assignment.

State Overview

The function block has a single state (ECState) named **REQ**:

State	Action	Output Event
REQ	`OUT := IN`	CNF

There is only this one state; after execution, the function block returns to its initial state. It is not an explicit state machine with multiple states.

Application Scenarios

- **Setting a SINT value at runtime**: For example, overwriting a limit value or a counter in a controller.
- **Initializing SINT variables** in a specific step of the control sequence.
- **Passing a constant or calculated value** to another function block via the InOut variable, provided there is a connection to a global variable or another InOut port.

Comparison with Similar Function Blocks

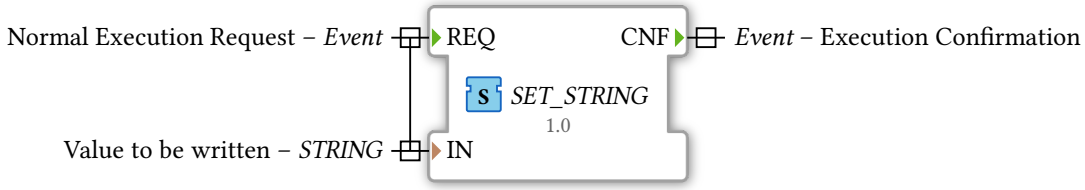
- **SET_BOOL**: Analogous function for the data type `BOOL`. Works identically, but with Boolean values.
- **SET_INT**: Corresponding function block for `INT`. The structure (event REQ/CNF, assignment) is identical.

- **MOVE (e.g., MOVE_SINT)**: A MOVE function block copies a value from an input to an output without the target variable being implemented as an InOut. **SET_SINT**, on the other hand, writes directly to an existing variable, which in some architectures offers the advantage that the variable can also be read by other function blocks without requiring a separate connection to an output.

Conclusion

SET_SINT is a minimal and reliable function block for setting a SINT variable to a desired value. Its simplicity makes it ideal for situations where a value needs to be overwritten directly without additional logic. Using an InOut variable saves on separate output variables and allows direct chaining with other function blocks that read the same variable. It is an efficient solution for simple value assignments.

SET_STRING



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SET_STRING` is used to write a string value (STRING) provided via the data input `IN` to an input-output variable `OUT`. Execution is triggered by an event at the input `REQ`. After writing, an acknowledgment event is sent at the output `CNF`.

Interface Structure

Event Inputs

Name	Type	Comment	With Variables
REQ	Event	Normal execution command	IN, OUT

Event Outputs

Name	Type	Comment	With variables
CNF	Event	Execution confirmation	OUT

Data Inputs

Name	Type	Comment
IN	STRING	The value to be written

Data Outputs

Name	Type	Comment
OUT	STRING	Target variable (In-Out) – identical to IN after execution

Note: `OUT` is declared as an In-Out variable. It serves simultaneously as an input (the current value before execution) and as an output (the new value after execution). When `REQ` is received, `OUT := IN` is set, overwriting the previous value.

Adapters

No adapters are available.

Functionality

1. An incoming event at input `REQ` starts the processing.
2. The code stored in algorithm `REQ` is executed:

- `OUT := IN;`

The value of the input variable `IN` is copied to the input-output variable `OUT`.

1. Subsequently, the event at output `CNF` is sent to signal successful execution.

Technical Features

- The function block contains only a single state (`REQ`) in which all processing takes place. It is a **simple function block** (SimpleFB) without sequential state machines.

1. The event is then sent at output `CNF` to signal successful execution. - The input-output variable `OUT` allows the function block to be directly linked to a persistently writable variable (e.g., a global data object or a linked hardware address) without requiring a separate output variable.

- The algorithm is trivial and performs no type or range checks.

State Overview

The function block has exactly one state:

State	Action	Output Event	Description
`REQ`	`OUT := IN`	`CNF`	Copies the input value to the input-output variable and acknowledges

A start state is not explicitly defined; The function block expects an external event pulse on `REQ`.

Application Scenarios

- **String Parameterization in Controllers**: Setting a string value in a global variable, e.g., a device name or IP address.
- **Initialization of Text Blocks**: Overwriting a standard string in a higher-level function block network.
- **Test and Debugging Environments**: Precisely specifying values for subsequent processing steps.

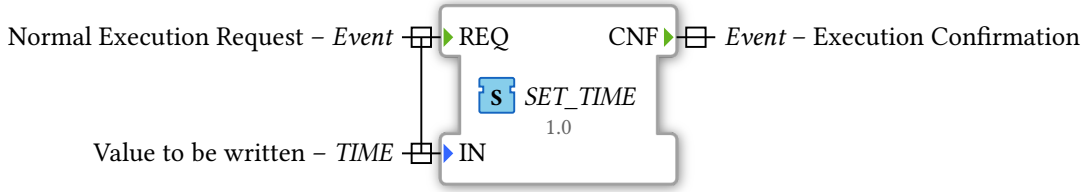
Comparison with Similar Function Blocks

- **SET_BOOL, SET_INT, SET_REAL**: Similarly structured function blocks for other data types. `SET_STRING` differs from `STRING` only in its data type.
- **MOVE**: A generic function block that can copy values of different data types. In contrast, `SET_STRING` is strictly specialized for `STRING` and works with an input/output variable, while `MOVE` uses separate inputs and outputs.
- **WRITE_STRING**: Blocks with similar functionality found in some libraries, but often equipped with additional control or error outputs.

Conclusion

The `SET_STRING` function block is a simple yet effective tool for precisely setting a string value to an input/output variable. It is particularly suitable for applications where a value needs to be written directly to an existing variable without requiring any readback or comparison logic. Due to its minimal interface and clear semantics, it can easily be integrated into more complex automation solutions.

SET_TIME



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_TIME** is used to write a time value (type TIME) provided via the data input IN to a variable OUT passed as an InOut parameter. The block is implemented as a simple, event-driven block and confirms execution via the event output CNF.

Interface Structure

Event Inputs

Event	Description
REQ	Normal execution request. Trigger the algorithm that copies the value from IN to OUT. Associated data: IN, OUT.

Event Outputs

Event	Description
CNF	Confirmation of successful execution. Sent after the algorithm completes. Associated data: OUT.

Data Inputs

Name	Type	Initial Value	Description
IN	TIME	T#0s	The time value to be written to the target variable.

Data Outputs

Name	Type	Initial Value	Description
OUT	TIME (InOut)	T#0s	Target variable passed as an InOut parameter during the call. The value of IN is copied into this variable.

> **Note:** OUT is an **InOut** parameter, meaning it serves as both an input and an output. The caller must provide a writable variable.

Adapter

None available.

Functionality

1. The function block waits for an event at input REQ.
2. When REQ occurs, the algorithm is executed: The current value of the data input IN is assigned to the InOut variable OUT (OUT := IN).
3. After successful assignment, the event CNF is triggered to signal completion to the calling function block.

Execution takes place in a single EC state (REQ), which runs the algorithm exactly once.

Technical Features

- **InOut Parameter:** OUT is an InOut data point. This allows the function block to write directly to a variable provided by the caller without requiring an additional data output. The caller must ensure that the referenced variable exists and is writable.
- **Simple Logic:** The function block consists of only one EC state and one ST algorithm. There are no branches, timers, or internal state machines.
- **Typing:** Both IN and OUT are of type TIME, meaning only time values are processed.

State Overview

The function block has exactly one state:

State	Description
REQ	Initial state upon initialization. Activated when REQ occurs. Executes the algorithm and sends CNF.

There are no other states. The function block returns to the initial state after completing the algorithm and waits for the next REQ event.

Application Scenarios

- **Setting Timer Parameters:** A higher-level control block passes a specific delay time (e.g., for a TON or TOF block) via an InOut parameter.
- **Initializing Time Variables:** In a startup sequence, predefined time values are written to global or instance-specific variables.
- **Copying Time Values:** A time variable is overwritten once with a calculated or configured value.
- **Real-Time Parameter Updates:** Dynamically changes a time reference shared by multiple blocks.

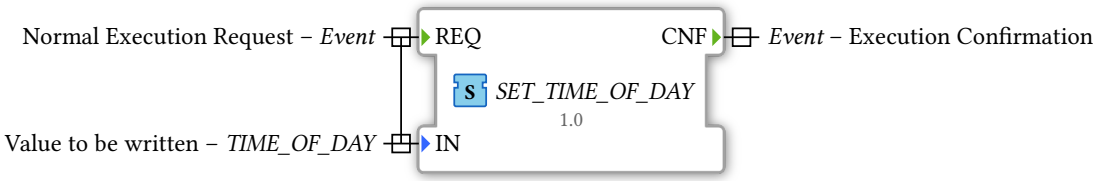
Comparison with Similar Blocks

- **Simple MOVE blocks (e.g., MOVE for generic types):** SET_TIME is specifically optimized for the TIME type and uses an InOut parameter, which saves memory and makes data transfer more efficient.
- **Direct assignment via IEC 61499 connections:** A simple connection from a OUT port of another block to the IN port of a target block could also set a value, but often requires additional event coupling. SET_TIME simplifies this through explicit REQ/CNF control.
- **Setting via global variables:** Other blocks write directly to global variables via service interfaces. SET_TIME, on the other hand, typically operates on instance-specific InOut parameters, which improves encapsulation.

Conclusion

The **SET_TIME** function block is a simple yet useful building block for writing a TIME value to a variable referenced as an InOut. Its clear event handling, minimal resource consumption, and direct coupling of input and output via the InOut mechanism make it ideal for time-critical assignment tasks in automation technology. The use of standard IEC 61499 mechanisms ensures interoperability and easy integration into existing control projects.

SET_TIME_OF_DAY



• – ◦ • – ◦ • – ◦ •

Introduction

The function block SET_TIME_OF_DAY is used to write a time of day value provided via the data input IN to a target variable OUT, which is declared as an InOut variable. The block is implemented as a simple FB (SimpleFB) and complies with the IEC 61499-1 standard.

Interface Structure

Event Inputs

Name	Type	Comment
REQ	Event	Normal execution request (bound to IN and OUT)

Event Outputs

Name	Type	Comment

CNF	Event	Execution confirmation (bound to `OUT`)

Data Inputs

Name	Type	Comment
IN	TIME_OF_DAY	The time of day value to be written

Data Outputs

No explicit data outputs are available. The target value is stored as an InOut variable.

Adapters

None.

Functionality

When an event arrives at input `REQ`, the algorithm `REQ` is executed:

- The value of `IN` is assigned to the InOut variable `OUT`: `OUT := IN`.
- After successful assignment, the event `CNF` is output to acknowledge the completion of the operation.

Technical Details

- The target variable `OUT` is declared as an **InOut variable**. This means that it serves as both input and output and must be passed externally by reference. The value of `OUT` is overwritten by the assignment.
- The function block contains only one state and one algorithm, ensuring very simple, deterministic execution.
- The data type `TIME_OF_DAY` follows the IEC 61499 data type definition for times of day (e.g., `TOD#12:30:00`).

State Overview

The function block has a single state:

- **REQ**: Activated on the event `REQ`. After the algorithm is executed, `CNF` is sent, and the state remains active (no state change).

State	Action	Output Event
REQ	Algorithm `REQ`	CNF

Application-Specific Scenarios

- **Setting a Global Time of Day**: In a controller, the current time can be retrieved from a higher-level system and written to a central InOut variable.
- **Configuring Time Parameters**: A user enters a time of day via an HMI, which is then written to a configuration variable using `SET_TIME_OF_DAY`.
- **Passing Time Values**: If a time value needs to be read from a source and passed to multiple subsequent function blocks, this function block can serve as a simple write access.

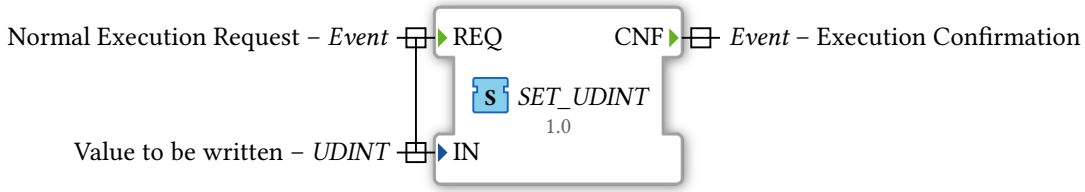
Comparison with Similar Function Blocks

- **MOVE for Elementary Types**: Similar function blocks exist for basic types such as `BOOL`, `INT`, `REAL`, etc. `SET_TIME_OF_DAY` is the specific variant for `TIME_OF_DAY`.
- **Standard MOVE Blocks**: Many IEC 61499 libraries contain generic function blocks such as `MOVE` or `ASSIGN`. The block presented here is a dedicated and type-safe implementation for `TIME_OF_DAY`.

Conclusion

SET_TIME_OF_DAY`is a minimal yet useful function block for overriding a`TIME_OF_DAY` InOut variable. Its simple structure makes it reliable and easy to understand. It can be effectively used in combination with other function blocks in timing and configuration applications.

SET_UDINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_UDINT** is used to write a value from the data input `IN` to the InOut variable `OUT`. The operation is triggered by an event at the input `REQ`, and after successful assignment, the output `CNF` is activated. The function block is specifically designed for the UDINT data type, thus enabling efficient and type-safe data transfer to a shared variable.

Interface Structure

Event Inputs

Event	Data Type	Comment
`REQ`	Event	Normal execution job; triggers writing the input value to the output variable.

Data associated with `REQ`: `IN`, `OUT`

Event Outputs

Event	Data Type	Comment
`CNF`	Event	Execution confirmation; outputs once the value has been successfully assigned.

Data associated with `CNF`: `OUT`

Data Inputs

Identifier	Data Type	Initial Value	Comment
`IN`	UDINT	0	Value to be written to the target variable.
`OUT` (InOut)	UDINT	0	Target variable that functions as both an input (readable) and an output (writable).

Data Outputs

Identifier	Data Type	Comment
`OUT` (InOut)	UDINT	Same variable as the input; after execution, it contains the value of `IN`.

Adapters

No adapters available.

Functionality

The function block works very simply:

1. An event at the input `REQ` starts the processing.
2. The ST code implemented in the algorithm `REQ` executes the assignment `OUT := IN;`.
3. After the assignment, the event at the output `CNF` is sent to signal successful completion.

The InOut variable `OUT` is both readable and writable. The assigned value remains until another `REQ` assigns a new value or the variable is modified externally.

Technical Features

- **Using an InOut Variable**: `OUT` is declared as an InOut variable. This allows the function block to directly access and modify an external variable without requiring a separate data output. The value is visible both inside and outside the function block after the write operation.
- **Simple Logic**: No intermediate storage, no delays – the process is atomic and occurs in a single step.
- **Type Binding**: The function block is defined exclusively for the type `UDINT` (unsigned 32-bit integer), guaranteeing type-safe operation.

State Overview

The function block contains a single state, `REQ` (event handling):

State	Action	Output Event
`REQ`	Execute algorithm `REQ` (OUT := IN)	CNF

After `CNF` is executed, the function block returns to its initial state and waits for the next `REQ` event.

Application Scenarios

- **Variable Initialization**: Setting a UDINT value to a predefined starting value at runtime.
- **Parameter Control**: Overriding configuration values in a running application (e.g., setpoints, counter values).
- **Data Passing Between Logic Blocks**: Using a shared InOut variable to update data without additional intermediate storage.
- **Replacement for Simple Move Functions**: Similar to a MOVE block, but with explicit UDINT focus and InOut mechanics.

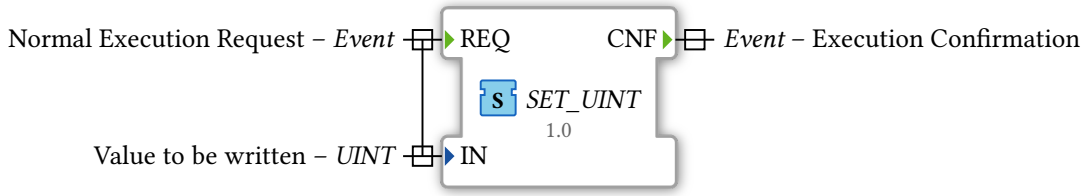
Comparison with Similar Blocks

- **MOVE (Generic)**: A general-purpose MOVE block can copy values of any type, but requires separate inputs and outputs. `SET_UDINT`, on the other hand, uses InOut, which reduces wiring and allows access to the original variable.
- **SET_BOOL, SET_INT, etc.**: Type variants of the same concept for different data types. `SET_UDINT` is specifically optimized for 32-bit unsigned integers.
- **R_TRIG / F_TRIG**: These edge detections do not set values, but rather detect changes. `SET_UDINT` directly writes values.

Conclusion

The function block `SET_UDINT` is a lean, reliable component for writing an InOut variable with a UDINT value. Its simple structure (one event, one assignment) makes it easy to understand and debug. Using an InOut interface saves on wiring and allows direct manipulation of external variables. Typical of IEC 61499, it is ideal for automation projects where fast and type-safe value assignments are required.

SET_UINT



• – ◦ • – ◦ • – ◦ •

Introduction

The SET_UINT function block is used to write a UINT value, which is present at a data input, into a variable declared as an InOut parameter. This simple but fundamental operation allows data to be copied from the FB context into an external or referenced variable.

Interface Structure

Event Inputs

Name	Type	Description
REQ	Event	Normal Execution Command

Event Outputs

Name	Type	Description
CNF	Event	Execution Confirmation

Data Inputs

Name	Type	Initial Value	Description
IN	UINT	0	Value to be written

Data Outputs

No separate outputs available. The target value is provided via the InOut variable `OUT`.

Adapters

No adapters.

Operation

Processing occurs in one step:

1. An event at the input `REQ` triggers the execution of the algorithm.
2. The algorithm copies the value of the data input `IN` to the InOut variable `OUT`.
3. Upon completion, the event `CNF` is output.

The InOut variable `OUT` is interpreted as a referenced variable – the function block writes directly to the memory location connected to the `OUT` interface during the call.

Technical Features

- **InOut Variable as Target:** The function block uses an InOut variable (`OUT`) instead of a simple output. This allows the function block to directly access and overwrite an existing variable in call environments (e.g., within resources or other function blocks).
- **Single-line algorithm:** The assignment `OUT := IN` is performed in a single ST instruction; no further calculations or intermediate storage are carried out.
- **No state machines:** The function block (FB) has only one entry state (`REQ`); state changes or loops are not implemented.

State overview

State	Action	Output
REQ	`OUT := IN`	CNF

After execution, the FB remains in an implicit end state; it is ready for each new request `REQ`.

Application Scenarios

- **Setting a Configuration Value:** A setpoint or parameter (e.g., a threshold in a control block) is input externally and written to the corresponding internal variable.
- **Copying UINT Data to a Central Memory Location:** In a controller, `SET_UINT` can be used to transfer measured values or calculation results to a global data block.
- **Exchanging Data Between Function Blocks:** Two different function blocks can access the same variable via the InOut variable – the `SET_UINT` function block is used for targeted overwriting.

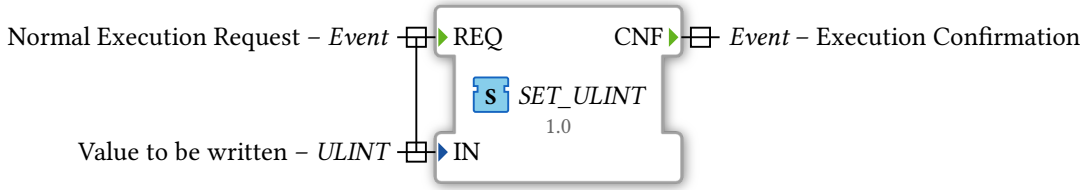
Comparison with Similar Function Blocks

- **SET_BOOL / SET_INT / SET_*:** These function blocks differ only in the data type used. Their functionality – "writing the value from IN to OUT" – is identical.
- **MOVE (e.g., `MOVE` in IEC 61131-3):** A general copy function, which is usually implemented as a standard function (without its own event interface). The `SET_UINT` function block provides explicit event control for its execution.
- **Special Case InOut:** Unlike pure output blocks, `SET_UINT` allows direct access to the target variable, which offers advantages when passing data across hierarchy boundaries.

Conclusion

The `SET_UINT` function block is a minimal but useful block for writing a UINT value to a referenced variable. Its simple structure—an event triggers an assignment—makes it particularly useful in environments where explicitly "setting" a variable via an event is required. If it does not meet any further conditions, it is suitable as a template for similar data type-specific set functions.

SET_ULINT



• – ◦ • – ◦ • – ◦ •

Introduction

The **SET_ULINT** function block is used to write an unsigned 64-bit integer value (ULINT) from a data input to an InOut variable. It is typically used to set values to linked variables or global memory locations that can also be read by other function blocks.

Interface Structure

Event Inputs

Event	Description
REQ	Normal execution command. The value of `IN` is transferred to the InOut variable `OUT`, and upon completion, the event output `CNF` is sent.

Event Outputs

Event	Description
CNF	Execution confirmation. Triggered as soon as the assignment of `IN` to `OUT` is completed.

Data Inputs

Name	Data Type	Description
IN	ULINT	The value to be written. Default: `0`.

Data Outputs

Name	Data Type	Description
OUT	ULINT	InOut variable – target of the assignment. It can be both read and written and is set to the new value after execution.

Adapter

None.

Functionality

The function block is implemented as a simple state machine (SimpleFB). A single state, `REQ`, executes the algorithm `REQ` when activated:

OUT := IN; The event output `CNF` is then sent. The passed value `IN` is therefore copied directly into the InOut variable `OUT`. Since `OUT` is declared as InOut, the function block cannot read the previous value of `OUT` – it is simply overwritten.

Technical Features

- The function block uses an **InOut variable** (`OUT`), which functions as both an input and output port in the IEC 61499 model. This allows direct connections to variables shared by multiple function blocks.
- The `logiBUS::utils::conversion::provide` package in the CompilerInfo indicates that the function block (FB) is intended for use in the logiBUS ecosystem.
- The algorithm is very simple – just a single ST assignment – and therefore extremely fast to execute.
- The initial value of `OUT` is `0`, so a defined starting value is available upon first use without prior initialization.

State Overview

The FB has exactly one EC state:

State	Action	Output
REQ	`REQ` (OUT := IN)	CNF

After startup or reset, the function block (FB) is in state `REQ`. Each `REQ` event pulse triggers the action and subsequently generates `CNF`.

Application Scenarios

- **Parameterized by Real-Time Systems**: Setting setpoints or configuration parameters in a variable monitored by multiple function blocks.
- **Global Status Variables**: Writing error codes or operating states to a central `ULINT` variable.
- **Testing and Diagnostic Purposes**: Targeted overwriting of values in a running controller for analysis purposes.

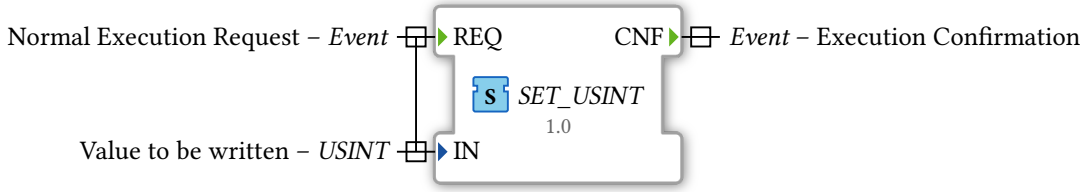
Comparison with Similar Function Blocks

- **SET_INT, SET_DINT, SET_LINT**: Similar function blocks for other integer types (INT, DINT, LINT). The functionality is identical; only the data type differs.
- **MOVE_ULINT**: Many libraries include a MOVE block that copies a value from an input to an output. `SET_ULINT` differs by using an **InOut variable**, which allows direct access to an existing variable without having to use a pure output.

Conclusion

The **SET_ULINT** function block is a simple yet effective tool for setting ULINT values in shared variables. Its InOut interface makes it particularly suitable for applications where values need to be overwritten without evaluating the previous state. Its lean implementation makes it well-suited for time-critical control applications.

SET_USINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_USINT** is used to write a value of type **USINT**, provided via the data input **IN**, directly into the variable **OUT**, which is declared as InOut. The block performs a simple assignment and confirms execution with the **CNF** event.

Interface Structure

Event Inputs

Event	Description
REQ	Normal execution request – takes the value from **IN** to **OUT** and sends **CNF**.

Event Outputs

Event	Description
CNF	Confirmation of successful execution (sent after assignment).

Data Inputs

Name	Data Type	Initial Value	Description
IN	USINT	0	Value to be written to the target variable.

Data Outputs

Name	Data Type	Initial Value	Description
OUT	USINT (InOut)	0	Target variable (declared as InOut, meaning it can be both read and written). After execution, **OUT** contains the value of **IN**.

Adapter

None.

Functionality

When the **REQ** event is received, the function block executes the **REQ** algorithm. This assigns the current value of the **IN** input to the **OUT** variable (`OUT := IN;`). The **CNF** event is then sent to signal the completion of the operation. The function block operates without delay or further conditions.

Technical Features

- The use of an **InOut** variable allows **OUT** to be used as both a target (output) and a source (input) in other function blocks.
- The function block is fixed to the **USINT** data type (8-bit unsigned integer) – direct conversion is not supported.
- Only one state exists; the function block is **event-driven** and performs exactly one assignment per **REQ**.

State Overview

State	Action	Output Event
REQ	Execute the **REQ** (`OUT := IN;`) algorithm	**CNF**

The FB has no explicit start state – it waits for an event in the **REQ** state.

Application Scenarios

- **Initializing** variables or peripherals that require a specific USINT value.
- **Parameterizing** devices or function blocks via an InOut variable.
- **Simply setting a value** in control logic when no calculation or condition is required.

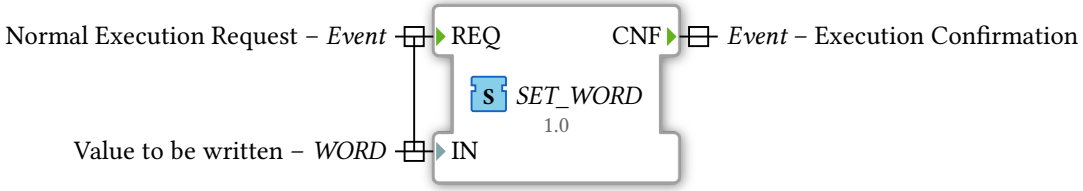
Comparison with Similar Function Blocks

- **MOVE (ANY)**: A more general function block for copying arbitrary data types, often without InOut capability.
- **SET_INT**, **SET_BYTE**, etc.: Analogous function blocks for other integer types – SET_USINT is specialized for USINT.
- Unlike **LATCH** or **SR flip-flops**, SET_USINT does not store state but performs a one-time assignment.

Conclusion

SET_USINT is a minimalist and reliable function block for directly assigning a value to an InOut variable of type USINT. Its simple structure is particularly suitable for clear, sequential control tasks where a value must be set without side effects.

SET_WORD



• – ◦ • – ◦ • – ◦ •

Introduction

The SET_WORD function block writes a WORD value from its data input IN to an InOut variable OUT. It is used to overwrite an output or a globally accessible WORD variable with a specific value.

Interface Structure

Event Inputs

- **REQ** (Event): Normal execution command. The block is triggered when an event arrives at this input.

Connected to the variables `IN` and `OUT`.

Event Outputs

- **CNF** (Event): Confirmation of execution. Sent after successful assignment.

Connected to the variable `OUT`.

Data Inputs

- **IN** (WORD): The value to be written to the target variable. Initial value: `0`.

Data Outputs

No data outputs are available. Output is exclusively via the InOut variable `OUT`.

Adapters

None.

Functionality

When an event occurs at the **REQ** input, the following algorithm is executed:
OUT := IN;`Subsequently, an acknowledgment event is sent at the **CNF** output. The function block overwrites the current value ofOUTwith the value ofIN`.

Technical Features

- `OUT` is declared as an **InOut variable**. It serves as both input (read) and output (write). The previous value of `OUT` is overwritten by the assignment.
- There are no separate data outputs – the change directly affects the associated variable.
- The function block is type-specific to `WORD`. Corresponding variants are available for other data types.

State Overview

The function block has one active state:

- **REQ**: Upon entering this state, the assignment `OUT := IN` is executed, and the **CNF** event is triggered.

There are no other states or branches.

Application Scenarios

- Setting a WORD output to a calculated or fixed value.
- Overwriting a global WORD variable in a PLC.
- One-time initialization of a WORD variable by an external function block.

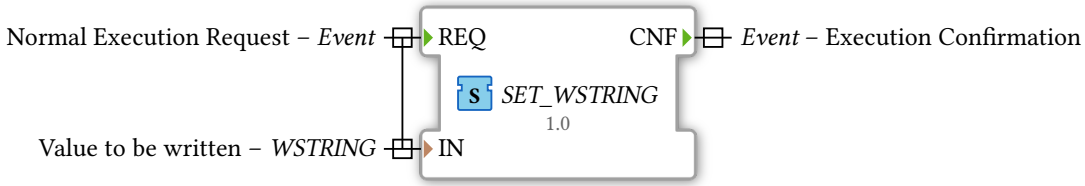
Comparison with similar function blocks

- **MOVE**: Copies a value from an input to an output (separate inputs/outputs). SET_WORD uses an InOut variable instead.
- **SET_BOOL**: Analogous functionality for the BOOL data type.
- **F_MOVE**: Function block for general data transfer, often with multiple inputs/outputs. SET_WORD is more specific and direct.

Conclusion

SET_WORD is a simple and efficient function block for writing a WORD value directly to an InOut variable. By using an InOut parameter, it reduces the number of variables required and is particularly useful in scenarios where a value needs to be overwritten without using a separate output.

SET_WSTRING



• – ◦ • – ◦ • – ◦ •

Introduction

The function block **SET_WSTRING** is used to transfer a value of data type `WSTRING` to an InOut parameter. It is typically used to write a calculated or externally specified string value to a higher-level variable (e.g., of another instance or a higher-level function block).

Interface Structure

Event Inputs

Event	Description
`REQ`	Normal execution call: The assignment of the input value to the InOut variable is triggered.

Event Outputs

Event	Description
`CNF`	Execution confirmation. Sent after successful assignment.

Data Inputs

Name	Data Type	Description
`IN`	`WSTRING`	Source value to be written to the target variable.

Data Outputs

This function block does not have traditional data outputs. The value is passed via the **InOut variable** `OUT`.

Adapters

None.

Functionality

If the event input REQ is activated, the function block executes the following assignment:
OUT := IN; The value WSTRING, which is present at input IN, is copied unchanged to the InOut variable OUT. Then, the event CNF is sent to signal the completion of the operation to the calling function block.

Technical Features

- **InOut Variable**: OUT is declared as an InOut parameter. This means that the variable can be both read and written, and its lifetime is outside the function block. The function block overwrites the current content of OUT with the value of IN.
- **Data type WSTRING**: This is a string type that supports wide characters (Unicode). The function block does not perform any length checks or conversions – the assignment is direct.
- **Simplest State Machine**: The state machine consists of a single state (REQ) that covers all functionality. There are no branches or time delays.

State Overview

The function block contains exactly one state:

State	Description
REQ	Becomes active after the event REQ occurs. Executes the assignment OUT := IN and sends CNF. The function block then returns to its initial state.

Application Scenarios

- **Passing a Formatted String**: Suppose a function block generates an error message as WSTRING and wants to write it to a parent variable. SET_WSTRING handles this assignment upon a trigger.
- **Parameterization**: In a controller, an external parameter (e.g., a product name) can be copied to a global data block using SET_WSTRING.
- **Debugging**: Temporarily setting a WSTRING variable using test or initialization logic.

Comparison with Similar Function Blocks

- **SET_BOOL**: Works analogously, but with the data type BOOL. Both use an InOut to write values to external variables.
- **MOVE (IEC Function Block)**: The standard MOVE function block copies data from an input to an output. In contrast, SET_WSTRING uses an InOut parameter, which allows connection to existing variables without an additional output port.
- **WRITE_STRING**: Some libraries offer special Write function blocks with length checks. SET_WSTRING is deliberately kept simple and does not include any checks.

Conclusion

The **SET_WSTRING** function block is a minimalist yet effective solution for writing to a WSTRING InOut variable. Its simple state machine and clear interface make it easy to understand and use wherever a trivial assignment to an event is needed. For tasks with additional functions (conversion, length constraints), specialized building blocks are preferable.

str

🎧 Podcast

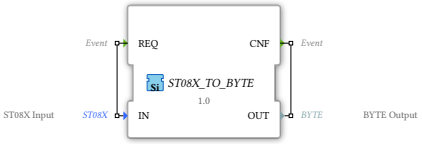
- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Sharpening Target Groups, Structure, and Unique Selling Proposition
- Taming the Software Dragon: Industrial Automation and the Future of Production
- Eclipse 4diac FORTE: Understanding IEC 61499 – The LEGO® Building Kit for Your Industry 4.0 Control System

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

assembling

ST08X_TO_BYTE



• – ○ • – ○ • – ○ •

Introduction

The function block `ST08X_TO_BYTE` converts a special data structure containing eight individual Boolean values (BOOL) into a single byte value (BYTE). This operation is useful for bundling compact digital signals for transmission or storage, or for connecting them to systems that expect data in byte format.

Interface Structure

Event Inputs

- **REQ**: This event triggers the conversion function. Upon its arrival, the current values at the data input `IN` are read and converted to the byte output `OUT`.

Event Outputs

- **CNF**: This event signals the successful completion of the conversion. It is generated simultaneously with the output of the calculated byte value at data output `OUT`.

Data Inputs

- **IN** (Type: `ST08X`): The input for the data structure to be converted. Type `ST08X` is a user-defined structure type (STRUCT) containing eight individual BOOL elements named `X_00` through `X_07`.

Data Outputs

- **OUT** (Type: `BYTE`): The output where the result of the conversion is provided. The eight bits of the byte (bits 0 to 7) directly correspond to the eight BOOL values of the input structure.

Adapter

This function block has no adapter interfaces.

Functionality

When triggered by the `REQ` event, the block performs a bitwise assignment. Each bit of the output byte `OUT` is set to the value of the corresponding Boolean element in the input structure `IN`. The assignment follows the following predefined scheme:

- `OUT`.0 (least significant bit, LSB) = `IN.X_00`
- `OUT`.1 = `IN.X_01`
- `OUT`.2 = `IN.X_02`
- `OUT`.3 = `IN.X_03`
- `OUT`.4 = `IN.X_04`
- `OUT`.5 = `IN.X_05`
- `OUT`.6 = `IN.X_06`
- `OUT`.7 (most significant bit, MSB) = `IN.X_07`

After these assignments are complete, the `CNF` event is output along with the new value of `OUT`.

Technical Features

- **Deterministic Mapping**: The assignment of individual BOOL elements to specific bit positions in the byte is hardcoded and not configurable.

- **Simple Logic**: The block contains no state machines or complex algorithms. Its functionality is limited to direct, parallel assignment.
- **Type Dependency**: The block is specific to the user-defined data type `ST08X`. Different blocks may be required for other structures or a different number of bits.

State Overview

The function block does not have an internal state in the sense of a state machine. It behaves purely combinatorially with respect to the data and sequentially with respect to event handling: An incoming `REQ` is always followed by an outgoing `CNF`.

Application Scenarios

- **Communication with Byte-Oriented Interfaces**: Preparing data for transmission via serial bus systems (e.g., Modbus, Profibus), networks, or communication protocols that expect data in byte blocks.
- **Compact Storage**: Bundling multiple digital switching states (e.g., from limit switches, machine states) into a single byte to optimize storage space.
- **Data Preparation for Visualization or Logging**: Converting structured BOOL signals into a byte format that is easier for display systems or data storage to handle.
-

⚖️ Comparison with Similar Blocks

- **`BOOL_TO_BYTE` (or similar blocks from standard libraries)**: Many IEC 61499 or IEC 61131-3 libraries offer blocks that directly convert eight separate `BOOL` inputs into a single `BYTE`. `ST08X_TO_BYTE` differs in that the input values are already bundled into a structured variable (`ST08X`), which can simplify wiring in the application diagram.
- **`PACK` / `UNPACK` blocks**: More general blocks for packing and unpacking data. `ST08X_TO_BYTE` is a specialized and type-safe implementation for precisely this one use case.
- **Manual Bit Operations**: The function could also be implemented through direct assignments in Structured Text (ST). The function block encapsulates this logic and provides a standardized, reusable interface with clear event handling.

Conclusion

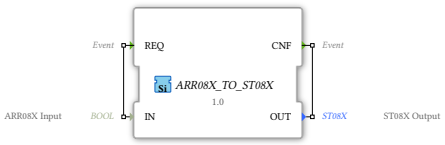
The `ST08X_TO_BYTE` function block is a specialized and efficient tool for converting a structured group of eight Boolean signals into a compact byte. Its clear, well-defined interface and simple operation make it ideal for applications where digital signals need to be bundled for efficient communication or storage. Its use promotes the reusability and readability of control programs compared to manually implemented bit operations.

converting

Numeric Index (converting)

Here you will find all exercises from the `converting` workspace, listed in ascending order. ...

ARR08X_TO_ST08X



• – ○ • – ○ • – ○ •

This function block (FB) is used to convert an array of 8 BOOL values (ARR08X) into a corresponding structure of 8 BOOL values (ST08X). It enables easy conversion between these two data formats.

- **REQ**: An event that triggers the conversion. When REQ is received, the value of input `IN` is processed, and the result is provided at output `OUT`.
- **CNF**: An acknowledgment event that is triggered once the conversion is complete and the output `OUT` has been updated.
- **IN**: An array of 8 BOOL values (BOOL[0..7]) representing the source data for the conversion.

- **OUT**: A structure of type `logiBUS::utils::conversion::types::ST08X` containing the converted 8 BOOL values.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block has no adapters.

The function block `ARR08X_TO_ST08X` performs a direct, element-by-element assignment of values from the input array `IN` to the corresponding fields of the output structure `OUT`. On each `REQ` event, the value of the `IN` array is read, and the individual elements of the array `IN[0]` to `IN[7]` are assigned to the fields `X_00` to `X_07` of the `ST08X` structure. After the assignment, the `CNF` event is triggered.

The assignments are made as follows:

- `OUT.X_00 := IN[0];`
- `OUT.X_01 := IN[1];`
- `OUT.X_02 := IN[2];`
- `OUT.X_03 := IN[3];`
- `OUT.X_04 := IN[4];`
- `OUT.X_05 := IN[5];`
- `OUT.X_06 := IN[6];`
- `OUT.X_07 := IN[7];`
- **Data Type Conversion**: This function block specializes in converting a generic BOOL array into a specific BOOL structure of type `ST08X`. This structure type must be known to the system (via `IMPORT logiBUS::utils::conversion::types::ST08X;`).
- **Data Type Conversion**: This function block specializes in converting a generic BOOL array into a specific BOOL structure of type `ST08X`. * **Fixed Size**: This function block is designed for arrays and structures with exactly 8 BOOL elements. It is not variable for other array sizes.
- **Determinism**: The conversion is direct and deterministic. Execution occurs immediately upon receiving the `REQ` event.

The `ARR08X_TO_ST08X` function block is a stateless function block. It does not store any internal states between calls. Each execution of the function block is independent of previous executions and is based solely on the current input values.

- **Interface Adaptation**: Ideal for adapting data formats between different system components, especially when one component prefers or expects an array and another a structure.
- **Standardized Data Access**: Enables access to individual Boolean values via named fields (`X_00`, `X_01`, etc.) instead of array indices, which can improve code readability.
- **Visualization**: Simplifies integration with visualization systems that may prefer a structured representation of Boolean signals over an array.
- **Legacy Systems**: Can serve as a bridge between older systems that may use Boolean arrays and newer components that work with structures.
- **Manual Assignment**: The functionality could also be implemented through direct assignments in the control program's code (e.g., in a `ST` block). The `ARR08X_TO_ST08X` function block encapsulates this logic in a reusable function block, increasing modularity and clarity.
- **Generic Converters**: Unlike a generic converter that could handle arbitrary array sizes or types, this block is specifically designed for 8-element BOOL arrays and the `ST08X` structure. This simplifies implementation but makes it less flexible for other sizes.

The `ARR08X_TO_ST08X` function block is a useful, specialized block for efficiently and clearly converting 8-element BOOL arrays into an 8-element BOOL structure. It contributes to improved code readability and modular architecture in 4diac applications, especially in scenarios requiring data format adaptation.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

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

LogiBUS BOOL conversion

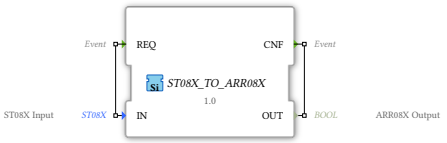
🎧 Podcast

- George Boole: How an autodidact laid the foundation of our digital world with 0 and 1

🛠️ Related exercises

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

ST08X_TO_ARR08X



• — ○ • — ○ • — ○ •

Introduction

This function block (FB) is used to convert a specific structure of type `ST08X`, which contains eight individual BOOL values, into an array of eight BOOL values. It enables seamless adaptation of the data representation in automation applications.

Interface Structure

Event Inputs

- **REQ**: This event is used to initiate the conversion operation. Upon receiving this event, the data from input `IN` is processed and written to output `OUT`.

Event Outputs

- **CNF**: This event is triggered after the data from the `IN` input has been successfully converted to the `OUT` array and the output has been updated.

Data Inputs

- **IN** (`logiBUS::utils::conversion::types::ST08X`): This is the input for the structure of type `ST08X`. This structure contains eight named BOOL fields (X_00 to X_07) to be converted.

Data Outputs

- **OUT** (`BOOL[0..7]`): This is the output for the array of BOOL values. The array has a fixed size of 8 elements containing the converted data from the input structure.

Adapters

This function block does not use any adapters.

Functionality

The function block `ST08X_TO_ARR08X` performs a direct and deterministic transformation of the input data. When the `REQ` event is received, the individual BOOL fields of the input structure `IN` are assigned to the corresponding index of the `OUT` array:

- `IN.X_00` is assigned to `OUT[0]`.
- `IN.X_01` is assigned to `OUT[1]`.
- ...
- `IN.X_07` is assigned to `OUT[7]`.

After these assignments are complete, the `CNF` event is thrown to indicate that the output `OUT` has been updated and the converted data is available.

Technical Features

- **Specific Types**: The function block is specifically designed for converting between the user-defined type `logiBUS::utils::conversion::types::ST08X` and a `BOOL[0..7]` array.
- **Fixed Mapping**: The mapping of structure fields to array indices is static and cannot be configured at runtime.
- **Atomic Operation**: The conversion is a single, self-contained operation triggered by an input increment event.

State Overview

The `ST08X_TO_ARR08X` function block is stateless. It does not store any internal state or data between executions. Each execution triggered by a `REQ` event processes the data currently assigned to `IN` completely and independently of previous operations.

Application Scenarios

- **Interface Adaptation**: Ideal for adapting data from a component that provides 8 BOOL values as a structure (`ST08X`) to another component that expects a `BOOL` array with 8 elements (e.g., for visualizations, external devices, or generic loop processing).
- **Data Format Standardization**: If a project uses different conventions for grouping 8 BOOL values, this function block can help bring the data into a consistent format.
- **Compatibility with legacy systems**: Can be used to convert data from more modern structures into array-based formats that are easier for older or simpler systems to process.

⚖️ Comparison with similar function blocks

While 4diac-IDE and the IEC 61499 standard provide generic data conversion mechanisms and array manipulation functions, the `ST08X_TO_ARR08X` is a highly specialized function block. General conversion function blocks might offer more flexible handling of different data types or array sizes, but often require configuration or additional logic. This function block provides a dedicated, simple, and efficient solution for this specific conversion case without any configuration overhead.

Conclusion

The `ST08X_TO_ARR08X` function block is a precise tool for converting an 8-BOOL structure into an 8-BOOL array. Its simplicity and specialization make it an efficient solution for interface adaptation and data formatting in 4diac-based automation applications that require such conversion.

🌐 Related topic subpages on ms-muc-docs.de

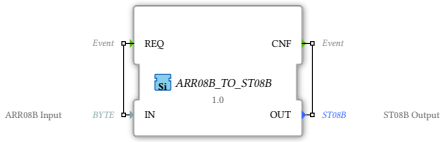
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Numeric Index (converting)

Here you will find all exercises from the `converting` workspace, listed in ascending order. ...

ARR08B_TO_ST08B



• – ○ • – ○ • – ○ •

Introduction

This function converts an 8-byte array into an 8-byte structure. It is used to convert linear byte data into a structured format.

Interface Structure

Event Inputs

- **REQ**: A rising event at this input triggers the conversion of the data from input `IN` to output `OUT`.
- Associated with: `IN`

Event Outputs

- **CNF**: An event at this output signals the completion of the conversion and that the output data at `OUT` has been updated.
- Associated with: `OUT`

Data Inputs

- **IN** (Type: `BYTE`, Array Size: `[0..7]`): The input data array containing the 8 bytes to be converted.
- Comment: `ARR08B Input`

Data Outputs

- **OUT** (Type: `logiBUS::utils::conversion::types::ST08B`): The output structure containing the converted 8 bytes.
- Comment: `ST08B Output`

Adapters

No adapters are defined for this function block.

Functionality

The function block `ARR08B_TO_ST08B` performs a direct byte-by-byte assignment. When a `REQ` event is received, the individual elements of the input array `IN` are assigned to the corresponding components of the output structure `OUT`:

- `OUT.B_00` is assigned to `IN[0]`.
- `OUT.B_01` is assigned to `IN[1]`.
- `OUT.B_07` is assigned to `IN[7]`.

After these assignments are complete, a `CNF` event is triggered to indicate that the conversion is finished and the new data is available at the `OUT` output.

Technical Details

- **Data Types:** The input array is of type `BYTE`, and the output structure `ST08B` is a user-defined type consisting of 8 individual `BYTE` components (e.g., `B_00` to `B_07`).
- **Direct Assignment:** No data manipulation or conversion takes place; the bytes are copied directly from the array position to the structure component.
- **Stateless:** The function block has no internal state and always produces the same result for the same input.

State Overview

The `ARR08B_TO_ST08B` function block is **stateless**. It performs a purely functional transformation: On each trigger event (`REQ`), the current input values (`IN`) are converted, and the result (`OUT`) is output. No values are stored between calls.

Application Scenarios

This function block is useful in scenarios where:

- Raw data received as a byte array (e.g., via communication interfaces such as CAN, SPI, UART) needs to be converted into a structured format to improve readability and further processing.
- Data is exchanged between systems that use different representations for the same 8-byte data set (array vs. structure).
- A clear and explicit type conversion from a linear to a named structure is required to improve program logic.

⚖️ Comparison with Similar Function Blocks

- **Generic Copy Functions (e.g., `MEMCPY`):** Unlike a generic memory copy function that copies memory areas byte by byte, `ARR08B_TO_ST08B` provides a type-safe and explicit assignment of array elements to structured fields. This can help avoid errors in resizing or type matching and improve code readability.
- **Manual Assignments:** The functionality could also be achieved through eight individual assignment lines in ST code. However, this function block encapsulates this logic in a reusable and easily integrated unit, promoting modularity and maintainability.
- **Type Casting:** While a type cast might enable a similar conversion in some languages, this function block provides an explicit, event-driven interface that fits well into the 4diac model and allows for clear triggering of the conversion.

Conclusion

The `ARR08B_TO_ST08B` function block is a specialized and efficient building block for converting an 8-byte array into a predefined 8-byte structure. It improves code structure, readability, and maintainability by encapsulating a frequently used conversion task in a clear and reusable unit. Its stateless nature ensures predictable and reliable operation.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

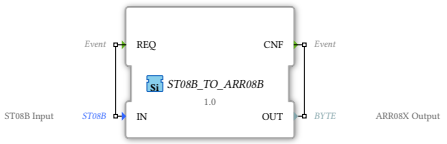
]

LogiBUS BYTE conversion

🛠️ Related exercises

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

ST08B_TO_ARR08B



• — ○ • — ○ • — ○ •

Introduction

The function block `ST08B_TO_ARR08B` converts an 8-byte structured data type (`ST08B`) into an 8-byte array. This conversion allows for the easy adaptation of structured data to interfaces or protocols that require a sequential arrangement of bytes.

Interface Structure

Event Inputs

- **REQ**: An event input that triggers the execution of the conversion. This event is linked to the data input `IN` to ensure that the conversion is performed using the current input data.

Event Outputs

- **CNF**: An event output that signals that the conversion was successful and the output `OUT` has been updated. This event is linked to the data output `OUT`.

Data Inputs

- **IN**: A data input of type `logiBUS::utils::conversion::types::ST08B`. This input expects a structure consisting of 8 bytes (e.g., `B_00` to `B_07`).

Data Outputs

- **OUT**: A data output of type `BYTE` as an array with size `0..7`. This array contains the 8 bytes that were converted from the input structure.

Adapters

This function block does not use any adapters.

How it Works

The operation of `ST08B_TO_ARR08B` is direct and straightforward. Upon receiving a `REQ` event, the individual byte elements of the input structure `IN` are directly assigned to the corresponding elements of the output array `OUT`. Each element `B_xx` of the structure `IN` is assigned to the element `OUT[xx]` of the array.

Specifically, the assignment is as follows:

- `OUT[0]` is assigned to `IN.B_00`.
- `OUT[1]` is assigned to `IN.B_01`.
- ...
- `OUT[7]` is assigned to `IN.B_07`.

After these assignments are complete, the `CNF` event is triggered to signal the completion of the operation.

Technical Features

- **Data Types:** The function block works with a specific user-defined structure type, `ST08B`, and a standard `BYTE` array.
- **Direct Mapping:** No complex logic, transformation, or calculation takes place; instead, the bytes are directly mapped one-to-one.
- **Transparency:** The implementation in ST (Structured Text) is clear and easy to understand, ensuring high transparency of the function.

State Overview

The function block `ST08B_TO_ARR08B` is a pure data conversion block without internal state in the sense of a state machine. It executes its operation deterministically on each `REQ` event and returns a result based on the current input data.

Application Scenarios

- **Data Preparation for Communication:** To prepare structured data for serial communication protocols (e.g., Modbus RTU, SPI, I2C), Ethernet frames, or other byte-oriented interfaces.
- **Connecting External Devices:** If external devices or libraries expect data as a pure byte array, this block can convert internal structured data accordingly.
- **Type Conversion and Interoperability:** Improves compatibility between different parts of an application or between different systems that use different data representations.
- **Archiving and Logging:** Stores structured data in a generic byte format for archiving or logging purposes.

⚖️ Comparison with Similar Blocks

Compared to other conversion blocks, `ST08B_TO_ARR08B` stands out due to its specialization in converting an 8-byte structure to an 8-byte array. General type conversion functions might be more complex and offer more configuration options, while this block performs a highly optimized and specific task. Blocks for array-to-structure conversion (e.g., `ARR08B_TO_ST08B`) would be the inverse function of this block.

Conclusion

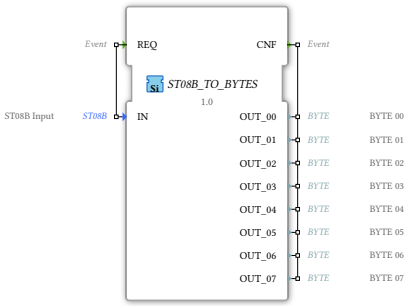
The `ST08B_TO_ARR08B` function block is a simple yet effective way to convert an 8-byte structure into an 8-byte array. It provides a reliable solution for scenarios requiring direct and transparent adaptation of structured data to byte-oriented interfaces or formats. Its clear functionality and implementation make it a valuable tool in the 4diac development environment.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

forwarding

ST08B_TO_BYTES



• – ○ • – ○ • – ○ •

Introduction

This function block extracts the individual 8 bytes of a specific 8-byte structure (`ST08B`) and provides them as separate byte outputs. It enables detailed segmentation and further processing of structured 8-byte data.

Interface Structure

Event Inputs

- **REQ**: Triggers the function and initiates the extraction of the bytes from the data input `IN`.

Event Outputs

- **CNF**: Sent after successful execution and updating of all output bytes. This event is associated with all `OUT_xx` data outputs.

Data Inputs

- **IN** (`logiBUS::utils::conversion::types::ST08B`): The input containing the 8-byte structure from which the bytes are to be extracted.

Data Outputs

- **OUT_00** (`BYTE`): The first extracted byte (byte 00).
- **OUT_01** (`BYTE`): The second extracted byte (byte 01).
- **OUT_02** (`BYTE`): The third extracted byte (byte 02).
- **OUT_03** (`BYTE`): The fourth extracted byte (byte 03).
- **OUT_04** (`BYTE`): The fifth extracted byte (Byte 04).
- **OUT_05** (`BYTE`): The sixth extracted byte (Byte 05).
- **OUT_06** (`BYTE`): The seventh extracted byte (Byte 06).
- **OUT_07** (`BYTE`): The eighth extracted byte (Byte 07).

Adapter

This function block has no adapter interfaces.

Operation

As soon as a `REQ` event is received, the function block reads the input structure `IN`. It extracts each individual byte of the structure (`IN.B_00` to `IN.B_07`) and assigns it to the corresponding output parameter (`OUT_00` to `OUT_07`). Upon completion of these assignments, the `CNF` event is triggered to signal the successful operation. The internal logic is direct and transparent:

OUT_00 := IN.B_00; OUT_01 := IN.B_01; OUT_02 := IN.B_02; OUT_03 := IN.B_03; OUT_04 := IN.B_04; OUT_05 := IN.B_05; OUT_06 := IN.B_06; OUT_07 := IN.B_07;

Technical Features

This function block specializes in decomposing the user-defined structure `ST08B`. Type assignment is performed directly without additional conversion logic, as both the source and target types (members of the structure and outputs) are of type `BYTE`. This ensures efficient and lossless data splitting. The block is deterministic and stateless, as it does not store any internal states that could influence the outcome of future executions.

State Overview

The function block `ST08B_TO_BYTES` is a pure converter without internal state. Each execution is independent of previous executions and always delivers the same result for identical input data.

Application Scenarios

- **Data Logging and Analysis**: When an 8-byte structure is received, the individual bytes can be extracted for separate logging, monitoring, or use in specific analyses.
- **Interface Adaptation**: Adapting data formats when an external interface provides an 8-byte structure, but subsequent components expect individual bytes.
- **Bit-Level Manipulation**: Preparing data for bit-level operations where each byte must be addressed individually.
- **Debugging**: Simplifies debugging data in 8-byte structures by making the contents of individual bytes visible.

⚖️ Comparison with Similar Components

Similar functionality could be achieved in other systems through explicit type conversions or pointer operations. In the 4diac context, this component represents an event-driven, clearly defined, and reusable solution for this specific structure-to-byte partitioning. The advantage lies in the direct connection to the 4diac event model and the explicit declaration of the interface, which improves readability and maintainability. Alternatively, the assignments would have to be made directly in the `ST` code of a Composite Function Block, which would limit reusability.

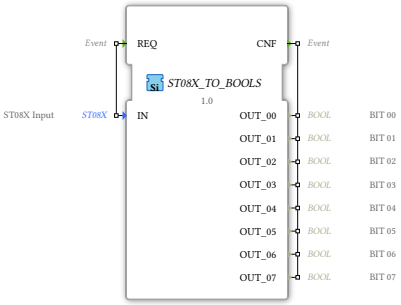
Conclusion

The `ST08B_TO_BYTES` function block is a useful tool for decomposing a specific 8-byte structure into its individual byte components. Its simplicity, clear interface, and deterministic functionality make it a reliable building block for applications that require precise control and analysis of 8-byte data. It promotes modularity and reusability in 4diac applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

ST08X_TO_BOOLS



• — ○ • — ○ • — ○ •

Introduction

This function block extracts eight individual BOOL values from a single input structure of type `ST08X`. It allows for the easy decomposition of a bundled 8-BOOL structure into its individual components.

Interface Structure

Event Inputs

- **REQ**: An event at this input requests conversion. It signals the function block to process the data from input `IN`.

Event Outputs

- **CNF**: This event is output once the conversion is complete and all output BOOL values (`OUT_00` to `OUT_07`) have been updated.

Data Inputs

- **IN** (Type: `logiBUS::utils::conversion::types::ST08X`): The input structure containing the eight BOOL values to be extracted.

Data Outputs

- **OUT_00** (Type: `BOOL`): The first extracted BOOL value (bit 00).
- **OUT_01** (Type: `BOOL`): The second extracted BOOL value (bit 01).
- **OUT_02** (Type: `BOOL`): The third extracted BOOL value (Bit 02).
- **OUT_03** (Type: `BOOL`): The fourth extracted BOOL value (Bit 03).
- **OUT_04** (Type: `BOOL`): The fifth extracted BOOL value (Bit 04).
- **OUT_05** (Type: `BOOL`): The sixth extracted BOOL value (Bit 05).
- **OUT_06** (Type: `BOOL`): The seventh extracted BOOL value (Bit 06).
- **OUT_07** (Type: `BOOL`): The eighth extracted BOOL value (Bit 7).

Adapter

No adapters available.

Functionality

As soon as an event is received at input `REQ`, the function block directly assigns the individual BOOL components of the input structure `IN` to the corresponding output variables. Specifically, `IN.X_00` is assigned to output `OUT_00`, `IN.X_01` to output `OUT_01`, and so on, up to `IN.X_07` to `OUT_07`. After successful assignment, a confirmation event `CNF` is output to indicate that the results are available.

Technical Features

- **Direct Mapping:** The function block implements a simple and direct 1:1 mapping from the fields of the `ST08X` structure to the individual BOOL outputs.

- **Type Safety:** The use of the specific type `ST08X` ensures a clear and type-safe interface.
- **Efficiency:** The conversion is highly efficient because no complex logic or calculations are required.

State Overview

This function block is stateless. It does not store any internal data between calls, and its output depends solely on the current input values. Each execution is independent of previous executions.

•

Application Scenarios

- **Decomposition of Combined Signals:** Ideal for splitting a logically grouped 8-bit structure into its individual, independent BOOL signals.
- **Interface Adaptation:** Enables connection to other function blocks or systems that expect individual BOOL inputs instead of a structure.
- **Visualization and Diagnostics:** Simplifies the separate display or monitoring of each individual bit of a `ST08X` structure.

⚖️ Comparison with Similar Blocks

This function block is specifically designed for the `ST08X` structure. While generic bit extraction functions (e.g., from a BYTE or WORD) might produce similar results, `ST08X_TO_BOOLS` provides a clearly defined, typed interface for the specific `ST08X` structure. This improves readability and avoids potential errors during manual bit indexing. It is simpler and more specific than function blocks that support complex bit field operations or variable bit counts.

Conclusion

The function block `ST08X_TO_BOOLS` is a useful tool for decomposing `ST08X` structures into individual BOOL signals. It impresses with its simplicity, efficiency, and clear, type-safe interface, which simplifies the development and maintenance of 4diac applications.

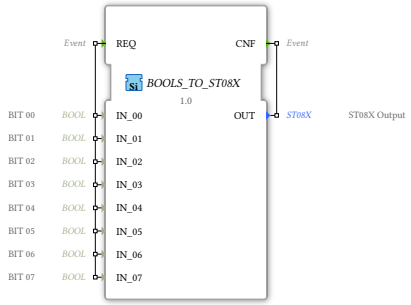
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

reversing

BOOLS_TO_ST08X



• – ○ • – ○ • – ○ •

This function block (FB) is used to combine eight individual BOOL values into a structure of type `ST08X`. It enables the clear grouping and transfer of eight binary pieces of information as a single structured variable.

- **REQ** (Event): This event triggers the conversion and assignment of the input values to the output. All data inputs are connected to this event and are read upon its arrival.
- **CNF** (Event): This confirmation event is triggered after the conversion is successfully completed and the output `OUT` has been updated. The data output `OUT` is associated with this event.
- **IN_00** (BOOL): The value for bit 0 of the output structure.
- **IN_01** (BOOL): The value for bit 1 of the output structure.

- **IN_02** (BOOL): The value for bit 2 of the output structure.
- **IN_03** (BOOL): The value for bit 3 of the output structure.
- **IN_04** (BOOL): The value for bit 4 of the output structure.
- **IN_05** (BOOL): The value for bit 5 of the output structure.
- **IN_06** (BOOL): The value for bit 6 of the output structure.
- **IN_07** (BOOL): The value for bit 7 of the output structure.
- **OUT** (logiBUS::utils::conversion::types::ST08X): The resulting structure containing the values of the eight binary inputs in their respective fields (`X_00` to `X_07`).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block has no adapters.

Functionality

The function block `BOOLS_TO_ST08X` is activated by an incoming `REQ` event. With each activation, the current values of the eight data inputs (`IN_00` to `IN_07`) are directly assigned to the corresponding fields (`X_00` to `X_07`) of the output structure `OUT`. After successful assignment, the `CNF` event is triggered at the output to signal the completion of the operation and make the updated `OUT` value available. The assignment is one-to-one, where `IN_00` corresponds to the field `X_00`, `IN_01` to the field `X_01`, and so on, until `IN_07` is assigned to the field `X_07`.

- **Direct Assignment:** The logic consists of a direct one-to-one assignment of the Boolean inputs to the fields of the `ST08X` structure.
- **Typing:** The `ST08X` type is a user-defined structure specifically designed for aggregating eight Boolean values and can be used in a type-safe manner.
- **Direct Assignment:** * **Efficiency:** Due to its simple implementation, this function block is very efficient and resource-saving.
- **Stateless:** The function block does not store any internal states and processes each request independently of previous calls.

The function block `BOOLS_TO_ST08X` is a stateless function block. This means that it does not store any information about previous executions that could influence its current behavior. Each activation by the `REQ` event results in a recalculation of the output `OUT` based on the values of the data inputs valid at that time.

- **Sensor Data Acquisition:** Combining eight individual digital sensor states (e.g., limit switches, light barriers) into a single structure for more compact data processing or visualization.
- **Interface Communication:** Prepare binary data for transmission via serial interfaces or communication protocols that expect data in structured formats.
- **Programming Logic:** Simplify the logic in more complex function blocks by combining eight separate signals into a single, more manageable variable.

This function block specializes in converting eight BOOL values to the user-defined `ST08X` type. Compare to:

- **`BOOLS_TO_BYTE` or `BOOL_TO_WORD`:** These blocks convert individual BOOL values to a primitive integer type (byte, word), where individual bits must be accessed via bitwise operations. In contrast, `BOOLS_TO_ST08X` offers a structured variable whose fields are directly accessible via names (`OUT.X_00`, `OUT.X_01`, etc.), which improves code readability.
- **Logical Operators (AND, OR, XOR):** These operators combine Boolean values, but not into a structure; instead, they combine them into a single Boolean result. `BOOLS_TO_ST08X` is used for aggregation, not logical combination.

The main advantage of `BOOLS_TO_ST08X` lies in its use of a specific structure type, which offers higher type safety and better code organization compared to generic integer types.

- **Logical Operators (AND, OR, XOR):** These operators combine Boolean values, but not into a structure; rather, they combine them into a single Boolean result.

The `BOOLS_TO_ST08X` function block is a simple yet extremely useful building block for converting eight individual BOOL signals into a standardized `ST08X` structure. It enhances the modularity and readability of 4diac applications by providing a clean and type-safe method for grouping bit information. This simplifies data handling and transmission in complex control systems.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

State Overview

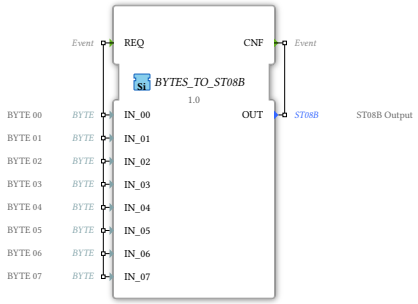
Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

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

BYTES_TO_ST08B



• – ○ • – ○ • – ○ •

Introduction

This function block combines eight individual byte values into a single 8-byte structure of type `ST08B`. It enables the orderly conversion of discrete byte values into a structured data format.

Interface Structure

Event Inputs

- **REQ** (Type: `Event`)
- Connects the activation of the function block to the data inputs: `IN_00`, `IN_01`, `IN_02`, `IN_03`, `IN_04`, `IN_05`, `IN_06`, `IN_07`.
- Triggers the conversion of the input bytes to the `ST08B` structure.

Event Outputs

- **CNF** (Type: `Event`)
- Triggered after successful conversion.
- Signals the availability of the converted value at data output `OUT`.

Data Inputs

- **IN_00** (Type: `BYTE`)
- Comment: BYTE 00

- **IN_01** (Type: BYTE)
- Comment: BYTE 01
- **IN_02** (Type: BYTE)
- Comment: BYTE 02
- **IN_03** (Type: BYTE)
- Comment: BYTE 03
- **IN_04** (Type: BYTE)
- Comment: BYTE 04
- **IN_05** (Type: BYTE)
- Comment: BYTE 05
- **IN_06** (Type: BYTE)
- Comment: BYTE 06
- **IN_07** (Type: BYTE)
- Comment: BYTE 07

Data Outputs

- **OUT** (Type: logiBUS::utils::conversion::types::ST08B)
- Comment: ST08B Output
- Outputs the 8-byte structure created from the eight input bytes.

Adapters

This function block does not have any adapter interfaces.

Functionality

Upon receiving a REQ event, the function block directly assigns the values of the eight data inputs IN_00 to IN_07 to the corresponding fields (B_00 to B_07) of the output structure OUT. After this assignment is complete, a CNF event is triggered, signaling that the OUT structure has been updated with the new values and is ready for further processing.

Technical Features

The function block uses the user-defined data type logiBUS::utils::conversion::types::ST08B as its output format. This type must be defined in the system for the function block to function correctly. The bytes are assigned positionally: IN_00 becomes OUT.B_00, IN_01 becomes OUT.B_01, and so on.

State Overview

The BYTES_TO_ST08B function block is a purely combinational function block. It has no internal states that are stored for extended periods. Each execution triggered by a REQ event processes the current input values independently of previous executions and produces a corresponding output result.

Application Scenarios

- **Data Packet Assembly:** Combining individual byte components into a predefined data packet, e.g., for communication via serial interfaces or networks.
- **Interfacing with Protocols:** Adapting data to communication protocols that expect specific 8-byte structures.
- **Data Type Conversion:** Converting raw byte data into a higher-level, structured data type for improved handling and readability within the application.
- **Hardware Interconnection:** When data from sensors or actuators is read byte by byte and needs to be aggregated into a fixed structure.

⚖️ Comparison with Similar Blocks

Compared to general type conversion or array manipulation blocks, BYTES_TO_ST08B offers a specialized solution for converting exactly eight bytes into a predefined ST08B structure. This eliminates the need for manual indexing or looping and simplifies handling for this specific use case. Other function blocks might be more flexible (e.g., able to process a variable number of bytes), but this block is optimized for its fixed size.

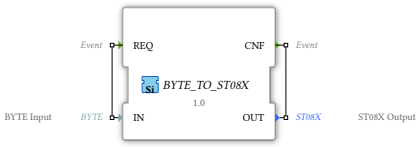
Conclusion

The function block BYTES_TO_ST08B is a useful tool for converting eight individual byte inputs into a defined 8-byte structure ST08B. It contributes to the modularization and simplification of control

programs, especially when it comes to preparing discrete byte data for structured communication tasks or internal data processing. Its direct and event-driven operation makes it reliable and easy to integrate.

splitting

BYTE_TO_ST08X



• — ◦ • — ◦ • — ◦ •

Introduction

This function block (FB) is used to break down a single byte value into a structure of eight individual BOOL values. Each bit of the input byte is assigned to a separate BOOL field in the output structure, which simplifies the handling of bitwise information.

Interface Structure

Event Inputs

- **REQ**: An event input that triggers the execution of the function block. It is associated with the data input `IN`.

Event Outputs

- **CNF**: An event output that signals that the conversion is complete. It is associated with the data output `OUT`.

Data Inputs

- **IN**: Type `BYTE`. This is the byte value whose eight bits are to be split into individual Boolean values.

Data Outputs

- **OUT**: Type `logiBUS::utils::conversion::types::ST08X`. This is a structure containing eight BOOL values (named X_00 to X_07). Each of these fields represents one bit of the input byte.

Adapters

This function block does not use adapters.

Functionality

The `BYTE_TO_ST08X` function block converts a single 8-bit byte into a predefined structure `ST08X`, which contains eight separate BOOL variables.

On a `REQ` event, the value of the `IN` byte is read. Then, each bit of the `IN` byte is assigned to the corresponding BOOL field in the `OUT` structure:

- `OUT.X_00` is assigned bit 0 of `IN`.
- `OUT.X_01` is assigned bit 1 of `IN`.
- ...
- `OUT.X_07` is assigned bit 7 of `IN`.
- After the assignment, a `CNF` event is triggered to signal the completion of the operation.

Technical Features

- **Direct Bit Extraction:** The conversion is performed by directly assigning the individual bits of the byte (e.g., `IN.0`) to the BOOL fields of the target structure (e.g., `OUT.X_00`).
- **Use of `ST08X`:** The function block uses the specific data type structure `logiBUS::utils::conversion::types::ST08X`, which is explicitly designed to hold eight Boolean values. This structure must be known to the system.
- **Event-driven:** The operation is triggered by an input event `REQ` and signals its completion with an output event `CNF`. This ensures controlled data flow in the 4diac system.

State Overview

This function block is a pure data converter without internal states. Its function is deterministic and depends solely on the input values and the triggering event.

Application Scenarios

- **Hardware Interfacing:** If a byte representing the state of eight discrete digital inputs or outputs is received from external devices (e.g., via fieldbuses such as Modbus or CANopen), this function block can be used to isolate the individual states for further processing in the control program.
- **Protocol Parsing:** Decomposition of status bytes in communication protocols, where each bit indicates a specific flag or state.
- **Logical Operations:** Preparing data for logical operations that require individual Boolean values.
- **Visualization:** Displaying individual bits of a byte value on a user interface.

⚖️ Comparison with Similar Blocks

Standard function blocks in 4diac often provide general type conversions (e.g., `BYTE_TO_INT`). However, this block specializes in decomposing a byte into a structured group of Boolean values (`ST08X`), which distinguishes it from generic converters. There is no direct equivalent in the standard library for this specific structure conversion. Other approaches could involve using bitmasks and logical operations (AND, SHIFT) with multiple individual conversion blocks, but this would result in a more complex and less user-friendly implementation. This block provides a direct and elegant solution to the task at hand.

Conclusion

The `BYTE_TO_ST08X` function block is a useful tool for efficiently and clearly decomposing a byte value into eight individual Boolean states, organized in a `ST08X` structure. It simplifies the handling of bitwise information, especially in I/O processing and protocol analysis, by providing a clear and event-driven interface.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

types

Numeric Index (types)

Here you will find all exercises from the `types` workspace, listed in ascending order. ...

LogiBUS BOOL Types

🎧 Podcast

- George Boole: How an autodidact laid the foundation of our digital world with 0 and 1

🛠️ Related exercises

- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

ST01X

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `ST01X` is a simple structured data type that encapsulates a single Boolean value (BOOL). It serves to provide a clearly defined structure for a single binary state, which can improve standardization and readability in more complex applications.

Interface Structure

The "interface structure" of a data type refers to its internal members, which form the definition of the data type. The data type `ST01X` has the following internal structure:

- **`X_00`**: A variable of type `BOOL`. This member serves as "BIT 00" within the structure, as indicated in the comment.

Event Inputs

Not applicable to a data type.

Event Outputs

Not applicable to this data type.

Data Inputs

Not applicable to this data type in the sense of function block inputs. This data type merely defines the structure for data.

Data Outputs

Not applicable to this data type in the sense of function block outputs. This data type merely defines the structure for data.

Adapters

Not applicable to this data type.

Functionality

The data type `ST01X` has no active functionality in the sense of an algorithm or execution. Its function is solely to serve as a defined container for a single Boolean value. It allows developers to clearly and type-safely reference a specific Boolean state within a structured environment, thereby increasing the modularity and readability of data structures.

Technical Features

- **Minimal Structure:** `ST01X` is one of the simplest structured data types, containing only a single `BOOL` element.
- **Encapsulation:** It encapsulates a single Boolean value, typically representing a status, flag, or state.
- **Standardization:** This data type can be used to standardize the use of single Boolean values in larger data structures or when defining communication protocols to ensure consistency.

State Overview

A data type does not have its own "state overview" in the sense of a state machine within a function block. The "state" of an object of type `ST01X` is determined solely by the value of its single member, `X_00`. This can be either `TRUE` (true) or `FALSE` (false).

Application Scenarios

- **Status Flags:** Represents a simple on/off status, e.g., to indicate whether a device is active, a process is running, or a condition is met.
- **Communication Protocols:** As a standardized block within data packets or telegrams to transmit a single bit state between systems.
- **Structured Data Storage:** When a Boolean value is needed in a higher-level structure with a specific name and comment to improve readability and maintainability, especially when many individual Boolean values are to be grouped together in a logical group.

- **Type Safety:** Enforces the use of an explicitly defined type for certain Boolean information to prevent errors due to accidental type inconsistencies.

⚖️ Comparison with Similar Building Blocks

- **Comparison with the primitive type `BOOL`:** Unlike the simple `BOOL` type, which only represents the raw value, `ST01X` offers named encapsulation (`X_00`). This improves the self-description and semantics of the value within a larger structure. While `BOOL` is an elementary data type, `ST01X` is a composite (structured) data type.
- **Comparison with larger structures (e.g., `ST08X`, `ST16X`):** `ST01X` is the most basic form of this type of structure, specializing in just one bit. Larger structures like `ST08X` or `ST16X` would contain 8 or 16 Boolean values, respectively, and are used to group multiple flags.

Conclusion

The `ST01X` data type is a fundamental yet useful building block in the 4diac IDE for structuring data. It allows the explicit definition and encapsulation of a single Boolean value, which contributes to improved code readability, maintainability, and standardization in complex automation projects. It serves as the basis for standardized data structures, especially when a bit flag has a specific meaning within a broader data landscape.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ST02X

(No image available for this data type.)

• – ◦ • – ◦ • – ◦ •

Introduction

The **ST02X** data type is a structured data type designed to encapsulate two Boolean values. It serves as a simple data structure for organizing and transporting two individual bits, or Boolean states, in a single unit. The name “ST02X” suggests a “structure with 2 [Boolean values]”.

Interface Structure

Because ST02X is a data type and not a function block, it does not have external event or data inputs/outputs in the usual sense of a function block. Its “structure” refers to the definition of its internal components, which represent the actual data.

Event Inputs

Not applicable, as ST02X is a data type and not a function block, and therefore does not process events.

Event Outputs

Not applicable, as ST02X is a data type and not a function block, and therefore does not generate events.

Data Inputs

The internal components of this data type can be considered its “data,” which can be set or read when a variable of this type is used.

- **X_00** (Type: BOOL, Comment: BIT 00)
- The first Boolean value within the structure.
- **X_01** (Type: BOOL, Comment: BIT 01)
- The second Boolean value within the structure.

Data Outputs

The internal components can also be considered “data outputs,” as their values can be read by external function blocks or program sections when a variable of type ST02X is used.

Adapters

Not applicable, as ST02X is a data type and not a function block, and therefore does not have adapter interfaces.

Functionality

The ST02X data type is a passive container for data. It does not perform any operations or logic itself. Its function is limited to logically grouping the two Boolean values `X_00` and `X_01`. When a variable of type ST02X is declared, its components can be accessed using the dot operator (e.g., `meineVariable.X_00`) to read or write its values.

Technical Features

- **Compact Structure:** The data type is very simple and consists of only two Boolean variables.
- **Typical Application:** It is particularly suitable for grouping two related status bits, flags, or switch states that need to be handled together.
- **Packaging:** The internal structure can help keep data compact and improve code readability by grouping logically related information.

State Overview

Since ST02X itself has no active logic, it does not have a "state" in the sense of a state machine. The "state" of a variable of type ST02X is determined by the combination of the values of its two Boolean components (`X_00` and `X_01`). There are four possible states for an instance of ST02X:

- `X_00 = FALSE, X_01 = FALSE`
- `X_00 = TRUE, X_01 = FALSE`
- `X_00 = FALSE, X_01 = TRUE`
- `X_00 = TRUE, X_01 = TRUE`

Application Scenarios

- **Status Display:** Grouping of two status flags (e.g., "Alarm Active" and "Warning Pending").
- **Configuration Parameters:** Bundling of two binary configuration settings.
- **Sensor Data:** Summary of the states of two simple sensors (e.g., two limit switches).
- **Interfaces:** As parameters for function blocks that need to process or return two Boolean pieces of information at once, which simplifies interface definition.

⚖️ Comparison with Similar Building Blocks

- **Single BOOL Variables:** Instead of using ST02X, two separate BOOL variables could be declared. The advantage of ST02X lies in its logical grouping and potentially clearer passing as a single parameter to function blocks.
- **ARRAY OF BOOL:** An array like `ARRAY[0..1] OF BOOL` could also hold two Boolean values. However, ST02X offers the advantage of named accesses (`.X_00`, `.X_01`), which can improve readability compared to indexed accesses (`[0]`, `[1]`).

ARRAY OF BOOL: An array like `ARRAY[0..1] OF BOOL` could also hold two Boolean values. However, ST02X offers the advantage of named accesses (`.X_00`, `.X_01`), which can improve readability compared to indexed accesses (`[0]`, `[1]`).

- – **Other Structured Types:** Depending on the application, more complex structured data types with more Boolean values (e.g., ST04X, ST08X) or mixed data types might be more suitable if more or different information needs to be grouped. ST02X is specifically optimized for the case of exactly two Boolean values.

Conclusion

ST02X is a simple yet useful structured data type that enables the logical grouping of two Boolean values. It improves code organization and readability, especially when dealing with two related binary pieces of information. Its strength lies in its simplicity and clear definition, making it a good choice for specific use cases where exactly two Boolean states need to be grouped.

ST04X

• – ◦ • – ◦ • – ◦ •

Introduction

ST04X is a structured data type that groups four Boolean values. It serves to organize and manage a collection of four individual bits in a single structure.

Interface Structure

Since ST04X is a data type and not a function block, it does not have event or data inputs/outputs in the sense of a function block or adapter. Instead, it defines an internal structure of data fields.

Event Inputs

Not applicable to a data type.

Event Outputs

Not applicable to a data type.

Data Inputs

Not applicable to a data type.

Data Outputs

Not applicable to a data type.

Adapters

Not applicable to a data type.

Structure Elements:

The ST04X structure consists of the following four Boolean fields:

- **X_00** (Type: BOOL, Comment: BIT 00)
- **X_01** (Type: BOOL, Comment: BIT 01)
- **X_02** (Type: BOOL, Comment: BIT 02)
- **X_03** (Type: BOOL, Comment: BIT 03)

Functionality

As a data type, ST04X does not have an active "function" in the sense of a function block. Its function is passive: It merely defines how four Boolean variables can be logically grouped and referenced as a single unit. This allows for better organization and readability of the code by grouping related bits in a structure.

Technical Features

- **Compactness**: Groups four individual BOOL values, which can enable more efficient data storage or transmission in some systems.
- **Structuring**: Improves code structure and readability by grouping related bits under a common name.
- **Use in Function Blocks**: This data type can be used as an input or output parameter for function blocks or as an internal variable within a function block to manage groupings of four bits.

State Overview

Not applicable to a data type. A data type represents a data structure, not a state machine.

Application Scenarios

- **Status Register**: Mapping of a small status register or group of flags where each bit indicates a specific state or property.
- **Binary Inputs/Outputs**: Grouping of four digital inputs or outputs of a hardware interface.
- **Bit Fields**: When a compact representation of four independent binary pieces of information is required, for example, when processing serial data protocols or device configurations.

⚖️ Comparison with Similar Components

ST04X is a data type and therefore cannot be directly compared to function blocks (components).

A comparison would be more feasible with other data types, for example:

- **BOOL**: A single Boolean value. ST04X groups four of them.
- **BYTE**: An 8-bit integer. ST04X provides a structured view of 4 bits, while a BYTE is typically treated as an unstructured 8-bit number. The individual bits of a BYTE would have to be accessed via bitmasks or bitwise operations, whereas with ST04X, each bit is directly accessible by its name (e.g., `myST04X.X_00`).

- **BYTE**: An 8-bit integer. * **Array of BOOL**: An array of BOOLs (`ARRAY[0..3] OF BOOL`) also provides grouping, but access is via indices (`myArray[0]`), whereas ST04X allows access via named fields (`myST04X.X_00`), which can improve readability.

Conclusion

ST04X is a useful structured data type in 4diac that simplifies the management and organization of four Boolean values. Although it does not offer any active functionality itself, it is an important tool for improving code structure and readability in more complex control programs by logically grouping related bits and enabling direct, named access to each bit.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ST08X

• – ◦ • – ◦ • – ◦ •

The **ST08X** data type is a structured data component designed for managing and grouping eight individual Boolean values. Its primary purpose is to provide a compact representation for a collection of eight independent bits, often used as status flags, control signals, or similar binary states. Because **ST08X** is a data type and not a function block, it does not have event, data inputs, or outputs in the sense of a 4diac function block. Instead, it defines an internal structure of member variables. Not applicable to a data type.

Not applicable to a data type. The data type consists of the following internal variables:

- `X_00`: Type `BOOL`, Comment: “BIT 00”
- `X_01`: Type `BOOL`, Comment: “BIT 01”
- `X_02`: Type `BOOL`, Comment: “BIT 02”
- `X_03`: Type `BOOL`, Comment: “BIT 03”
- `X_04`: Type `BOOL`, Comment: “BIT 04”
- `X_05`: Type `BOOL`, Comment: “BIT 05”
- `X_06`: Type `BOOL`, Comment: “BIT 06”
- `X_07`: Type `BOOL`, Comment: “BIT 07”

Not applicable for a data type.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Not applicable for a data type.

Functionality

The **ST08X** data type does not have its own inherent “function” in the sense of an algorithm or processing. It merely serves as a container to organize eight Boolean values under a common name. The individual bits can be read and written in other function blocks or program logic using their names (e.g., `ST08X_Variable.X_00`).

Not applicable for a data type.

The **ST08X** data type does not have an independent “function” in the sense of an algorithm or processing. It merely serves as a container to organize eight Boolean values under a common name. The individual bits can be read and written via their names (e.g., `ST08X_Variable.X_00`).

- **Compact Structure**: Combines eight `BOOL` variables into a single logical unit.

- **Explicit Naming**: Each bit is explicitly named from `X_00` to `X_07`, improving readability and addressing.
- **No Implicit Conversion**: As a structured data type, it offers type-safe encapsulation of the bits.

As a pure data type, **ST08X** has no states in the sense of a state machine or lifecycle. Its "state" is determined by the current values of its eight Boolean members.

- **Status Register**: Mapping of status bits of an external device or an internal component.
- **Configuration Parameters**: Grouping of eight binary configuration options.
- **Error Messages**: Collection of up to 8 different error message flags.
- **Binary Inputs/Outputs**: Representation of 8 digital inputs or outputs.
- **Data Bus Representation**: In cases where a group of 8 bits should be treated as a unit without interpreting it as a complete byte value, ST08X allows direct access to individual bits.

The **ST08X** data type offers an alternative to:

- **Single BOOL Variables**: Reduces the number of declarations and simplifies parameter passing when many Boolean values belong together.
- **BYTE Data Type**: While a `BYTE` also stores 8 bits, these are treated as a numeric value. **ST08X** retains the separate, named addressability of each individual bit, which improves semantics and direct access to logical flags without the need for bit masking or bit shift operations. The explicit naming, `X_00` through `X_07`, makes the code more self-explanatory.

The **ST08X** data type is a useful construct for grouping eight independent Boolean values in a structured and well-named manner. It promotes code readability and maintainability by providing clear semantic encapsulation for collections of binary flags that go beyond simple bitwise operations on a byte.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

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

ST16X

• – ◦ • – ◦ • – ◦ •

Introduction

ST16X is a user-defined data type that defines a structure of 16 individual BOOL values. It serves to encapsulate a collection of binary states or flags in a single, organized unit. Although it is a data type and not a function block, its structure and usage are described here. The data type was created to provide a clear and type-safe way to manage 16 logical states.

Interface Structure

Because ST16X is a data type and not a function block, it does not have classic interfaces in the sense of event or data inputs/outputs. Instead, its 'structure' defines the internal members, which are detailed below.

Event Inputs

Not applicable to a data type.

Event Outputs

Not applicable to a data type.

Data Inputs

Not applicable to a data type.

Data Outputs

Not applicable for a data type.

Adapters

Not applicable for a data type.

The structure of the ST16X data type is defined as follows:

Name	Type	Comment
X_00	BOOL	BIT 00
X_01	BOOL	BIT 01
X_02	BOOL	BIT 02
X_03	BOOL	BIT 03
X_04	BOOL	BIT 04
X_05	BOOL	BIT 05
X_06	BOOL	BIT 06
X_07	BOOL	BIT 07
X_08	BOOL	BIT 08
X_09	BOOL	BIT 9
B_10	BOOL	BIT 10
B_11	BOOL	BIT 11
B_12	BOOL	BIT 12
B_13	BOOL	BIT 13
B_14	BOOL	BIT 14
B_15	BOOL	BIT 15

Functionality

A data type like ST16X does not have an active ‘function’ in the sense of an algorithm that is executed. It merely defines a template for data. The ‘function’ arises from how this data type is used in function blocks or other program logic to store and manage 16 individual Boolean states. A variable of type ST16X can be created, and its individual BOOL elements can be read and written (e.g., `meineST16XVariable.X_00 := TRUE;`).

Technical Features

- **Structured Arrangement:** This data type groups 16 independent BOOL values into a single logical unit.
- **Unique Naming:** Each bit is addressed by an individual name (`X_00` to `B_15`) instead of an index, which improves code readability.
- **Comment:** Each variable is assigned a comment (“BIT xx”) that clarifies the bit position.
- **Modularity:** Promotes code modularity and reusability by bundling complex status information.

Application Scenarios

- **Status Registers:** Represents the state of up to 16 different components, sensors, or functions (e.g., “Motor ON,” “Fault Active,” “Limit Reached”).
- **Bit Flags:** Stores configuration flags or fault indicators in control applications.
- **Digital Input/Output Signals:** Groups the digital inputs or outputs of a module to transmit them as a single data structure.
- **Data Packaging:** Simplified transfer of multiple binary pieces of information between function blocks or via communication interfaces.

⚖️ Comparison with Similar Building Blocks

Since ST16X is a data type, it cannot be directly compared to a 'building block' in the sense of a function block. However, it can be compared to other methods for storing binary data:

- **Single BOOL Variables:** ST16X offers the advantage of treating 16 BOOLs as a single unit, which simplifies data structuring and transfer and reduces the number of declared variables.
- **Array of BOOL:** A `ARRAY [0..15] OF BOOL` offers similar functionality. ST16X has the advantage that individual bits can be addressed using descriptive names (e.g., `meineST16XVariable.X_00`), which improves readability, instead of using indices (`meinARRAY[0]`).
- **WORD/INT types:** These can also be used to store bits. ST16X offers a type-safe and self-documenting way to manage bits without manual bit masking operations or typical conversion errors.

Conclusion

The ST16X data type provides an efficient and clear method for structuring a group of 16 Boolean values. It improves code readability, simplifies the handling of complex binary states, and promotes type safety in 4diac applications, especially in scenarios where many individual pieces of binary information need to be managed as a coherent unit.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

ST32X

(No image available, as this is a data type and not a function block.)

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `ST32X` is a user-defined structure data type consisting of 32 individual BOOL variables. It serves to bundle a collection of Boolean states or flags into a single, coherent data structure, which can simplify the handling and passing of status information. The name `ST32X` is derived from "Structure (ST) with 32 X/B (Boolean) bits".

Interface Structure

Since `ST32X` is a data type and not a function block, it does not have event- or data-related inputs and outputs in the sense of a function block. Its "structure" merely defines its internal elements.

Event Inputs

Not applicable, as `ST32X` is a user-defined data type and not a function block.

Data Inputs

The structure consists of the following 32 BOOL elements, which can be considered "inputs" internally, as they receive the values from which the structure is formed:

- `X_00` (Type: BOOL, Comment: BIT 00)
- `X_01` (Type: BOOL, Comment: BIT 01)
- `X_02` (Type: BOOL, Comment: BIT 02)
- `X_03` (Type: BOOL, Comment: BIT 03)
- `X_04` (Type: BOOL, Comment: BIT 04)
- `X_05` (Type: BOOL, Comment: BIT 05)
- `X_06` (Type: BOOL, Comment: BIT 06)
- `X_07` (Type: BOOL, Comment: BIT 07)
- `X_08` (Type: BOOL, Comment: BIT 08)
- `X_09` (Type: BOOL, Comment: BIT 09)
- `B_10` (Type: BOOL, Comment: BIT 10)
- `B_11` (Type: BOOL, Comment: BIT 11)
- `B_12` (Type: BOOL, Comment: BIT 12)
- `B_13` (Type: BOOL, Comment: BIT 13)
- `B_14` (Type: BOOL, Comment: BIT 14)
- `B_15` (Type: BOOL, Comment: BIT 15)

- `B_16` (Type: BOOL, Comment: BIT 16)
- `B_17` (Type: BOOL, Comment: BIT 17)
- `B_18` (Type: BOOL, Comment: BIT 18)
- `B_19` (Type: BOOL, Comment: BIT 19)
- `B_20` (Type: BOOL, Comment: BIT 20)
- `B_21` (Type: BOOL, Comment: BIT 21)
- `B_22` (Type: BOOL, Comment: BIT 22)
- `B_23` (Type: BOOL, Comment: BIT 23)
- `B_24` (Type: BOOL, Comment: BIT 24)
- `B_25` (Type: BOOL, Comment: BIT 25)
- `B_26` (Type: BOOL, Comment: BIT 26)
- `B_27` (Type: BOOL, Comment: BIT 27)
- `B_28` (Type: BOOL, Comment: BIT 28)
- `B_29` (Type: BOOL, Comment: BIT 29)
- `B_30` (Type: BOOL, Comment: BIT 30)
- `B_31` (Type: BOOL, Comment: BIT 31)

Data Outputs

Not applicable. The structure `ST32X` itself is the "output" because it represents the bundled Boolean values. Individual elements can be accessed using the dot operator (`.`) (e.g., `myST32X.X_00`).

Adapters

Not applicable because `ST32X` is a user-defined data type and not a function block.

Functionality

ST32X` has no active "function" in the sense of logic or algorithms. It merely serves as a passive data container. Its function is to organize a group of 32 individual BOOL values into a single variable, which simplifies program structuring and the transfer of data between function blocks or program sections. The individual bits of the structure can be accessed, and their values can be set or read.

Technical Specifications

- **Number of Bits:** The structure comprises 32 independent BOOL variables.
- **Naming:** The elements are named `X_00` to `X_09` and then `B_10` to `B_31`. The comments indicate the bit number in each case. This could indicate an original intention to represent the bits in a word (e.g., DINT), even though they are declared here as separate BOOLs.
- **Memory Efficiency:** The exact memory allocation depends on the compiler implementation. Typically, such a structure would occupy at least 32 bytes (one byte per BOOL) or possibly more, depending on padding and alignment.

State Overview

As a passive data type, `ST32X` does not have its own state machine. Its "state" is defined solely by the values of the 32 contained BOOL variables. Each bit can be either `TRUE` or `FALSE`.

Application Scenarios

- **Status Flag Bundling:** When a system needs to manage a large amount of binary status information (e.g., error codes, component operating states, sensor flags), `ST32X` can be used to pass or store it in a single variable.
- **Interface Optimization:** Instead of passing 32 separate BOOL signals to a function block, an instance of `ST32X` can be transmitted over a single data connection.
- **Bit Mapping:** Although the elements are defined as `BOOL`, this structure could be used in conjunction with conversion function blocks (e.g., to convert to `DINT`) to access or manipulate the bits of a 32-bit word.
- – **PLC-like I/O mappings:** In environments similar to PLC programming, such a structure can be used to map digital inputs or outputs.

⚖️ Comparison with similar function blocks

- **Standard data types (e.g., `DINT`):** A `DINT` (Double Integer) is also 32 bits wide, but stores the bits as a single numeric value. `ST32X`, on the other hand, treats each bit as a separate Boolean variable, allowing direct access by name without bit masking or bitwise operations.
- **Other structured data types:** Other user-defined structures may exist that bundle a smaller or larger number of BOOLs or mix different data types. `ST32X` is specifically designed to hold 32 Boolean values.
- **Array of Boolean Values:** A `ARRAY[0..31] OF BOOL` would also store 32 Boolean values. The difference to `ST32X` lies in the access method: `ST32X` accesses named elements (e.g., `myST32X.X_00`), while an array accesses its index (e.g., `myArray[0]`). Named elements are often more readable and more robust against changes in order.

Conclusion

The `ST32X` data typeIt offers a practical method for combining 32 individual Boolean variables into a coherent whole. It promotes code organization and simplifies the passing of extensive status information or flags. Although it does not implement any active logic itself, it is a useful tool in the architecture of complex control programs based on the 4diac IDE.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ST64X

(No image available because this is a data type and not a function block.)

• – ◦ • – ◦ • – ◦ •

Introduction

The provided XML definition describes a **DataType** named `ST64X`, not a function block (FB). This structured data type serves as a container for a collection of 64 individual binary values (BOOL variables). It allows 64 independent switches, flags, or status bits to be grouped under a single name and addressed individually via named fields. Although the internal XML comment states “A Structure with 16 BOOLs,” the actual definition contains 64 BOOL variables.

Interface Structure

Since `ST64X` is a data type and not a function block, it does not have direct event or data inputs and outputs in the sense of an FB. Instead, it defines an internal structure of components that can be used as a unified variable within a function block (e.g., as an input or output parameter).

Event Inputs

Not applicable for a data type.

Event Outputs

Not applicable for a data type.

Data Inputs

Not applicable for a data type. The elements listed here are the **internal components** of the `ST64X` structure that hold data. When `ST64X` is used as a data input for a function block, these are the available fields:

Name	Type	Comment
`X_00`	`BOOL`	`BIT 00`
`X_01`	`BOOL`	`BIT 01`
`X_02`	`BOOL`	`BIT 02`
`X_03`	`BOOL`	`BIT 03`
`X_04`	`BOOL`	`BIT 04`

X_05	BOOL	BIT 05
X_06	BOOL	BIT 06
X_07	BOOL	BIT 07
X_08	BOOL	BIT 08
X_09	BOOL	BIT 09
B_10	BOOL	BIT 10
B_11	BOOL	BIT 11
B_12	BOOL	BIT 12
B_13	BOOL	BIT 13
B_14	BOOL	BIT 14
B_15	BOOL	BIT 15
B_16	BOOL	BIT 16
B_17	BOOL	BIT 17
B_18	BOOL	BIT 18
B_19	BOOL	BIT 19
B_20	BOOL	BIT 20
B_21	BOOL	BIT 21
B_22	BOOL	BIT 22
B_23	BOOL	BIT 23
B_24	BOOL	BIT 24
B_25	BOOL	BIT 25
B_26	BOOL	BIT 26
B_27	BOOL	BIT 27
B_28	BOOL	BIT 28
B_29	BOOL	BIT 29
B_30	BOOL	BIT 30
B_31	BOOL	BIT 31
B_32	BOOL	BIT 32
B_33	BOOL	BIT 33
B_34	BOOL	BIT 34
B_35	BOOL	BIT 35
B_36	BOOL	BIT 36
B_37	BOOL	BIT 37
B_38	BOOL	BIT 38
B_39	BOOL	BIT 39
B_40	BOOL	BIT 40
B_41	BOOL	BIT 41

B_42	BOOL	BIT 42
B_43	BOOL	BIT 43
B_44	BOOL	BIT 44
B_45	BOOL	BIT 45
B_46	BOOL	BIT 46
B_47	BOOL	BIT 47
B_48	BOOL	BIT 48
B_49	BOOL	BIT 49
B_50	BOOL	BIT 50
B_51	BOOL	BIT 51
B_52	BOOL	BIT 52
B_53	BOOL	BIT 53
B_54	BOOL	BIT 54
B_55	BOOL	BIT 55
B_56	BOOL	BIT 56
B_57	BOOL	BIT 57
B_58	BOOL	BIT 58
B_59	BOOL	BIT 59
B_60	BOOL	BIT 60
B_61	BOOL	BIT 61
B_62	BOOL	BIT 62
B_63	BOOL	BIT 63

Data Outputs

Not applicable for a data type. The elements listed above are the **internal components** of the `ST64X` structure that hold data. When `ST64X` is used as a data output of a function block, these are the available fields.

Adapters

Not applicable for a data type.

Functionality

ST64Xitself has no "function" in the sense of an active component that performs calculations or changes states. Its "function" is to provide a defined structure that serves as a template for data objects. Once instantiated, a variable of typeST64Xholds 64 individual BOOL values that can be read or written directly via their names (e.g.,MyVariable.X_00, MyVariable.B_63`). It is a passive data container.

Technical Features

- **Structured Data:** Defines a fixed arrangement of 64 BOOL variables.
- **Named Fields:** Each bit has a unique name (`X_00` to `X_09` and `B_10` to `B_63`), which improves code readability compared to purely numeric indices.
- **Size:** Represents 64 bits, which typically occupies 8 bytes of memory (depending on the compiler implementation and padding).
- **Package Information:** The data type is located in the package `logiBUS::utils::conversion::types`, indicating its role in utilities or conversions.

State Overview

Not applicable for a data type, as it has no internal logic or state machine. The “state” of a `ST64X`The variable is simply the current value of the 64 contained BOOL variables.

Application Scenarios

- **Bit Fields and Flags:** Ideal for representing a large number of binary status flags, switch positions, or configuration options that need to be organized in a single, coherent block.
- **Hardware Interfaces:** Can be used to map registers or I/O ports where individual bits represent specific functions or states.
- **Compact Data Transmission:** When a large number of BOOL values need to be transmitted efficiently, this type can provide a compact representation.
- **Abstract States:** Represents complex states resulting from the combination of 64 binary features.
-

⚖️ Comparison with Similar Building Blocks

Since `ST64X` is a data type, it is not directly compared to function blocks, but rather to other ways of structuring BOOL values:

- **`ARRAY OF BOOL`:** A `ARRAY[0..63] OF BOOL` would also store 64 BOOL values. The main difference lies in the access method: `ST64X` uses named arrays (e.g., `MyVar.X_00`), which is often more readable if each bit has a specific meaning. Arrays use indices (e.g., `MyArray[0]`), which is more flexible for loops and dynamic access, but less informative if the meaning of the individual indices is not obvious.
- **`ARRAY OF BOOL`:** A `ARRAY[0..63] OF BOOL` would also store 64 BOOL values. * **Individual `BOOL` variables:** You could also declare 64 separate `BOOL` variables. However, this would result in a cluttered list of variables and obscure the relationship between the bits. `ST64X` groups them logically.
- **Integer data types (e.g., `LWORD`):** A `LWORD` (64-bit integer) could also store 64 bits. However, accessing individual bits would require bitwise operations (AND, OR, SHIFT), which can be more complex and error-prone than direct array access to a structure. `ST64X` provides direct, symbolic bit access.

Conclusion

The data type `ST64X` is a useful construct in 4diac-ide for defining a structured variable containing 64 individual BOOL values. It provides a clear, named interface to each individual bit, improving code readability and maintainability, especially in applications requiring fine-grained control or status indication with many binary flags. It is a good alternative to `ARRAY OF BOOL` or bit manipulations on integer types when the individual meaning of each bit is paramount.

`ST64X`` or bit manipulations on integer types when the individual meaning of each bit is crucial.

`ST64X`` is a useful construct in 4diac-ide for defining a structured variable containing 64 individual BOOL values.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Numeric Index (types)

Here you will find all exercises from the `types` workspace, listed in ascending order. …

LogiBUS BYTE Types

🛠️ Related exercises

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b
- Uebung_12x_sub

ST01B

• – ◦ • – ◦ • – ◦ •

Introduction

The provided XML file describes the 4diac **data type** `ST01B` (Structure with 1 BYTE), not a function block (FB). This data type defines a simple data structure containing a single byte field. The following documentation attempts to apply the requested structure for a function block as closely as possible, although many sections are not applicable to a pure data type.

Interface Structure

Since `ST01B` is a data type and not a function block, it does not have event or data interfaces in the sense of a function block. Instead, it defines an internal structure for data.

Event Inputs

Not applicable, as `ST01B` is a data type and not a function block.

Event Outputs

Data Inputs

Not applicable, as `ST01B` is a data type and not a function block. It defines a structure that can be used as a type for data variables in function blocks.

Data Outputs

Adapters

Functionality

ST01B` itself has no functionality in the sense of an executable function block. This is a static definition of a data structure that comprises a single byte. This structure can be instantiated by function blocks and used to store or exchange byte values.

The structure contains the following variable:

- **B_00**: A `BYTE` field annotated as “BYTE 00”.

Technical Features

- **Simple Structure**: Defines a minimal structure for encapsulating a single byte.
- **Use in 4diac**: This data type can be used in 4diac applications to declare typed data variables or as part of more complex data types.
- **Package**: The data type is contained in the `logiBUS::utils::conversion::types` package.

State Overview

Not applicable, as `ST01B` is a data type and not an executable function block with states.

Application Scenarios

This data type can be useful in various scenarios where a single byte with a specific semantic name needs to be handled in a structured way, for example:

- As a unified type for certain register values.
- To standardize interfaces that pass single byte values.
- As a building block for more complex structures consisting of a series of bytes.

⚖️ Comparison with Similar Building Blocks

- **Direct Use of `BYTE`**: `ST01B` differs from the direct use of the primitive `BYTE` type in that it provides a specific name and encapsulation. This can improve code readability and maintainability, especially when the byte has a specific meaning beyond the generic `BYTE` type.
- **Other Structured Data Types**: Compared to more complex structured data types, `ST01B` is very simple and serves as a basic building block.

Conclusion

`ST01B`` is a simple yet useful structured data type in 4diac that allows for the encapsulation of a single byte. Although not a function block, it can improve the clarity and structure of 4diac applications by defining specifically named byte variables. The provided XML definition facilitates easy integration into 4diac projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ST02B

• – ◦ • – ◦ • – ◦ •

Introduction

ST02B` is a user-defined, structured data type within the 4diac IDE. It defines a data structure that combines two individual bytes into a logical unit. This simplifies the handling of byte pairs.

Structure Definition

Since `ST02B` is a data type and not a function block, it has no interfaces in the sense of event inputs/outputs, data inputs/outputs, or adapters. Instead, the internal components of the data type are described here.

Components of the Structure

The structure `ST02B` consists of the following two byte variables:

- **B_00** (Type: `BYTE`): The first byte of the structure (BYTE 00).
- **B_01** (Type: `BYTE`): The second byte of the structure (BYTE 01).

Functionality

As a passive data type, `ST02B` has no inherent "function." Its function is to provide a defined template for data. Instances of this data type can be used to store, organize, and exchange data as a unit between different functional blocks.

Technical Details

ST02B`is defined as`StructuredType`and consists of two consecutive`BYTE` variables. Therefore, the total size of an instance of this data type is 2 bytes.

Application Scenarios

- **Data Transfer**: Bundling 2 bytes that should be treated as a single unit.
- **Register Mapping**: Representation of a 2-byte register (similar to a WORD, but with explicit byte access).
- **Small Data Packets**: Simplified handling of small, predefined data packets.

⚖️ Comparison with Similar Building Blocks

- **Array of Bytes (ARRAY OF BYTE [2])**: An array of 2 bytes serves the same purpose. However, `ST02B` offers the advantage of named fields (B_00, B_01).
- **`WORD`**: Also represents 2 bytes. `ST02B` is more flexible if the individual bytes need to be manipulated independently.

Conclusion

ST02B` is a simple, useful structured data type for managing two bytes as a coherent unit.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

ST04B

• – ◦ • – ◦ • – ◦ •

Introduction

ST04B` is a user-defined, structured data type within the 4diac IDE. Unlike a function block (FB), which implements specific functionality, a data type defines a data structure that can be used by function blocks or other data types. This data type groups four individual bytes into a logical unit to simplify their interaction.

Structure Definition

Since `ST04B` is a data type and not a function block, it does not have interfaces in the sense of event inputs/outputs, data inputs/outputs, or adapters. The following sections adapt the original requirement of an interface structure to represent the internal components of the data type.

Components of the Structure

The structure `ST04B` consists of the following four byte variables:

- **B_00** (Type: `BYTE`): The first byte of the structure (BYTE 00).
- **B_01** (Type: `BYTE`): The second byte of the structure (BYTE 01).
- **B_02** (Type: `BYTE`): The third byte of the structure (BYTE 02).
- **B_03** (Type: `BYTE`): The fourth byte of the structure (BYTE 03).

Functionality

As a passive data type, `ST04B` has no inherent "function" in the sense of logic or algorithm. Its function is to provide a defined template for data. Instances of this data type can be declared and used in function blocks to store, organize, and exchange data as a coherent unit between different function blocks. The internal components (B_00 to B_03) are directly addressable.

Technical Details

ST04Bis defined asStructuredTypeand consists of four consecutiveBYTEvariables. The total size of an instance of this data type is therefore 4 bytes. In the context of the 4diac runtime environment, this data type is stored in memory according to its definition. The internal structure allows direct access to each individual byte via its name (e.g.,myST04B.B_00`).

The 4diac compiler assigns this data type to the package `logiBUS::utils::conversion::types`.

State Overview

As a passive data type that merely defines a data structure, `ST04B` has no internal logic or state machine in the sense of a function block. Its "state" is determined solely by the values of its four component bytes. Therefore, a state overview in the sense of behavioral states is not applicable.

Application Scenarios

- **Data Transmission**: Bundling of 4 bytes that are received or sent via a communication interface (e.g., serial, Modbus, CAN) and are to be treated as a unit.
- **Register Mapping**: Representation of a 4-byte register or group of registers in a programmable logic controller (PLC) or other control system.
- **Small Data Packets**: Simplified handling of small, predefined data packets in industrial automation.
- **Type Conversion Utility**: Internally used in conversion function blocks to process specific byte sequences and convert them to other data types.

⚖️ Comparison with Similar Building Blocks

- **Array of Bytes (ARRAY OF BYTE [4])**: An array of 4 bytes serves the same purpose for storage. However, `ST04B` offers the advantage that individual bytes can be addressed via named fields (`B_00`, `B_01`, etc.) instead of indices (e.g., `myArray[0]`, `myArray[1]`), which can improve code readability and self-description.
- **Other structured types**: If the bytes had a specific meaning (e.g., status flag, counter value), more complex structured types with more descriptive variable names or even bit fields might be more appropriate. `ST04B` is intentionally generic to allow for flexible byte grouping.
- – **`WORD`, `DWORD`**: These standard data types also represent a specific number of bytes (2 and 4, respectively). However, `ST04B` is more flexible when individual bytes need to be manipulated independently or when byte order (endianness) is important and needs to be explicitly controlled.

Conclusion

ST04B` is a basic yet useful structured data type in 4diac that simplifies the management of four bytes as a coherent unit. It improves code readability and modularity, especially in applications that require the exchange or processing of byte-sequenced data. Its simple definition makes it a flexible building block for a wide range of data management tasks in automation technology.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

ST08B

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `ST08B` is a structured data structure that sequentially groups exactly eight bytes (BYTE). It primarily serves as a container for the efficient grouping and transport of eight individual byte values within an IEC 61499-based control system. This data type is particularly useful for handling raw data, protocol frames, or for the structured transfer of parameterized data blocks between function blocks.

Interface Structure

ST08B` is a data type and, as such, does not have event-driven inputs or outputs in the sense of a function block. Its structure is defined exclusively by its data members.

Data Inputs

Not applicable. This section is intended for function blocks.

Data Outputs

Event Inputs

Not applicable. This section is for function blocks.

Event Outputs

Adapters

How it Works

As a structured data type (STRUCT), `ST08B` acts as a composite container. It groups eight independent variables of type `BYTE` under a common name. Each byte within the structure is directly addressable via its unique member name (`B_00` to `B_07`). Function blocks can declare variables of type `ST08B` to pass, store, or process all eight bytes as a single unit. Individual bytes are accessed using dot notation (e.g., `MyST08BInstance.B_04`).

Technical Features

- **Fixed Size:** The structure has a fixed and predictable size of exactly 8 bytes, which is advantageous for memory allocation and communication protocols.
- **Sequential Arrangement:** The bytes are arranged in the order `B_00` to `B_07`. This order is crucial for interpreting the data, for example, when converting it to larger data types such as `WORD`, `DWORD`, or `LWORD`.
- **TypeHash Attribute:** The data type contains the attribute `eclipse4diac::core::TypeHash` with an empty value (`''`). This indicates that type-safe validation via hash value has not (yet) been configured for this user-defined type.

State Overview

Since this is a passive data type and not an active function block, there is no state machine. The “state” of an instance of `ST08B` is simply the current value of each of its eight byte members.

Application Scenarios

1. **Data Packaging:** Grouping eight configurable parameters or status values together to be sent from one function block to another.
2. **Protocol Handling:** Representing a fixed data frame header or segment in serial or network communication.
3. **Raw Data Buffer:** Temporary storage of an 8-byte block from a sensor or communication interface before further processing or interpretation.
4. **Conversion Base:** Used as a source or target for conversion function blocks that convert `ST08B` to or generate other data types, such as `LWORD` (64-bit).

⚖️ Comparison with Similar Building Blocks

- **Simple Byte Array:** An array of 8 `BYTE` (`ARRAY[0..7] OF BYTE`) could perform a similar function. However, `ST08B` offers the advantage of named access paths (`B_01`), which, unlike index-based access (`Array[1]`), improve code readability and maintainability.
- **LWORD:** The data type `LWORD` also represents 8 bytes (64 bits), but as a single, undivided numeric value. `ST08B` allows targeted access to each individual byte within these 64 bits.
- **Other STRUCT Types:** Similar structured types such as `ST04B` (4 bytes) or `ST16B` (16 bytes) serve the same purpose, but for different amounts of data. `ST08B` fills the gap for 8-byte structured data.

Conclusion

The `ST08B` is a simple yet extremely useful structured data type for IEC 61499 programming. It provides a clear and structured method for grouping eight bytes of data, thereby promoting readability, maintainability, and type safety in applications that work with fixed data blocks or parameterized datasets. Its fixed size and sequential arrangement make it a reliable building block for data communication and conversion.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

ST16B

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `ST16B` is a structured data component defined in the 4diac framework. Unlike a function block (FB), `ST16B` has no inherent logic or executable functions. Instead, it serves as a fixed container for 16 individual byte values. Its primary purpose is to provide a clearly defined structure for handling fixed-size raw data packets, which is particularly important in communication protocols or when interacting with low-level hardware interfaces.

Interface Structure

Since `ST16B` is a data type and not a function block, it does not have event or data interfaces in the conventional sense. Its “structure” refers to the definition of its internal components, i.e., the individual bytes that comprise it.

Event Inputs

Not applicable, as `ST16B` is a data type and not a function block, and therefore does not process events.

Event Outputs

Not applicable, as `ST16B` is a data type and not a function block, and therefore does not generate events.

Data Inputs

Not applicable, as `ST16B` is a data type and not a function block, and therefore does not have direct data inputs in the context of function block connections.

Data Outputs

Not applicable, as `ST16B` is a data type and not a function block, and therefore does not have direct data outputs in the context of function block connections.

Adapter

Not applicable, as `ST16B` is a data type and not a function block, and therefore does not implement any adapter interfaces.

Component Data Structure

ST16B`consists of 16 individual variables of type`BYTE`, arranged sequentially:

Name	Type	Description
B_00	BYTE	BYTE 00

B_01	BYTE	BYTE 01
B_02	BYTE	BYTE 02
B_03	BYTE	BYTE 03
B_04	BYTE	BYTE 04
B_05	BYTE	BYTE 05
B_06	BYTE	BYTE 06
B_07	BYTE	BYTE 07
B_08	BYTE	BYTE 08
B_09	BYTE	BYTE 09
B_10	BYTE	BYTE 10
B_11	BYTE	BYTE 11
B_12	BYTE	BYTE 12
B_13	BYTE	BYTE 13
B_14	BYTE	BYTE 14
B_15	BYTE	BYTE 15

Functionality

ST16B `itself has no active functionality. It is a passive data container. Its function is to provide a defined memory template for 16 bytes. Each byte (B_00toB_15`) can be read and written individually. This structured access to individual bytes enables granular data manipulation and easy assignment to communication fields or memory areas.

Technical Features

- **Fixed Size:** The data type `ST16B` has a fixed size of exactly 16 bytes. This is advantageous for predictable memory usage and compatibility with protocols that expect fixed lengths.
- **Direct Byte Access:** Each of the 16 bytes is directly accessible via its specific name (e.g., `MyVariable.B_00`).
- **Raw Data Container:** Ideal for storing and transporting raw data whose content will be interpreted later.
- **Architecture Independence:** As a collection of bytes, the basic structure is relatively platform-independent, although the interpretation of the bytes can vary across different systems (e.g., endianness for larger values spread across multiple bytes).

State Overview

Not applicable, as `ST16B` is a data type and does not have internal states in the sense of a function block (e.g., initialization, execution, error). The "states" of `ST16B` are simply the current values stored in its 16-byte components.

Application Scenarios

- **Communication Protocols:** Ideal for packaging or unpacking messages for protocols such as Modbus, CAN bus, or proprietary serial protocols where data packets have a fixed size of 16 bytes.
- **Hardware Interfaces:** Mapping of registers or data blocks in low-level interfaces that operate in fixed byte patterns.
- **Binary Data Buffer:** Used as a buffer for binary data read from or sent to external sources.
- **Hashe or ID Caching:** For applications that need to store 128-bit hashes (e.g., MD5) or long IDs that are exactly 16 bytes long.

⚖️ Comparison with Similar Building Blocks

- **`ARRAY[0..15] OF BYTE`:** `ST16B` is similar to a 16-byte array. The main difference lies in the access method: `ST16B` uses named components (`B_00`, `B_01`, etc.), while an array allows access via indices (`MyArray[0]`, `MyArray[1]`). Named components can improve readability if each byte has a specific semantic meaning within the 16 bytes.
- **Other structured data types:** Compared to user-defined structures that mix different data types (INT, REAL, BOOL), `ST16B` specializes in pure byte collections. It offers a flat, uniform structure.

Conclusion

The data type `ST16B` is a simple yet effective structure for handling fixed-size 16-byte data blocks. It provides a clear and readable way to access individual bytes within such a block. Although it is not a function block and does not implement its own logic, it is a valuable tool in 4diac for applications that require precise control at the byte level, especially in communication and hardware interaction.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ST32B

• – ◦ • – ◦ • – ◦ •

The data type `ST32B` is a structured data type within the 4diac IDE, designed for the aggregation of 32 individual bytes. It serves as a template for data structures that require a fixed length of 32 bytes and allow direct access to individual bytes. This type is registered in the 4diac IDE under the package `logiBUS::utils::conversion::types`. Since `ST32B` is a structured data type and not a function block, it does not have external event or data interfaces in the conventional sense. Instead, it defines an internal structure of elements that can be accessed in a program. Not applicable to structured data types. `ST32B` has no event inputs.

Not applicable to structured data types. `ST32B` has no event outputs.

Not applicable to structured data types. `ST32B` has no data inputs; its elements are directly accessible via the type name.

Not applicable to structured data types. `ST32B` has no data outputs; its elements are directly accessible via the type name.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

Not applicable to structured data types. `ST32B` has no adapters.

Functionality

ST32B` is a passive data type and does not implement any active functionality or logic. It serves merely as a blueprint (schema) for organizing and storing 32-byte values under a single logical name. Instances of this data type can be declared and used in other function blocks or data types to store or transfer data in a structured manner.

The individual bytes are named as follows and can be accessed directly:

- `B_00`: Byte 00
- `B_01`: Byte 01
- ...
- `B_30`: Byte 30

- `B_31`: Byte 31
- **Fixed Size:** The data type `ST32B` is fixed to an exact size of 32 bytes.
- **Direct Byte Access:** Each of the 32 bytes can be addressed individually by its name (`B_00` to `B_31`).
- **Basic Type:** It consists exclusively of the primitive data type `BYTE`, making it ideal for low-level data operations or the representation of raw data.
- **Simplicity:** The structure is deliberately kept simple to ensure efficient use and clear data organization.

As a passive data type, `ST32B` has no internal states in the sense of a state machine. The "state" of an instance of this type is defined solely by the current values of the 32 bytes it contains.

- **Raw Data Transport:** Ideal for transmitting fixed-length raw data packets, for example, from sensors or via communication interfaces such as SPI, I2C, UART, or Ethernet, when the payload has a fixed length of 32 bytes.
- **Message Structures:** Can serve as the basis for simple message formats where a portion of the message frame occupies a defined length of 32 bytes.
- **Memory Image:** For mapping a small, fixed memory area or register set in a controller.
- **Configuration Data:** Storage of 32-byte configuration parameters.

A comparable approach would be the use of an array of type `ARRAY[0..31] OF BYTE`.
The main difference lies in the access method and semantics:

- **`ST32B` (structure):** Allows access to bytes via symbolic names (`B_00`, `B_01`, ...), which can improve code readability and maintainability, especially when individual bytes have a specific meaning.
- **`ARRAY[0..31] OF BYTE` (array):** Allows access via a numeric index (`[0]`, `[1]`, ...), which is advantageous for loops and generic operations.
- While an array can offer more flexibility in sizing, `ST32B` provides a clearly defined, named structure that makes the purpose behind each byte more explicit, even though the names are generic.

The `ST32B` data type is a fundamental structured data type in 4diac that offers a simple and efficient way to treat 32 bytes as a single unit. Its strength lies in providing a fixed, well-defined structure for raw data, making it a useful tool in applications that require precise control over byte sequences. Although it has no active functionality itself, it is an important foundation for data-driven architectures within 4diac applications.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

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

ST64B

• – ◦ • – ◦ • – ◦ •

Introduction

The `ST64B` data type is a structured data type used to encapsulate 64 individual byte values. It was developed to provide a fixed 64-byte data structure that can be used for various use cases where a precisely defined size and direct access to individual bytes are required. The original XML definition contained a comment that incorrectly stated 16 bytes; the correct value is 64 bytes.

Structure Definition

The `ST64B` data type is a composite structure consisting exclusively of individual byte variables. It has no event- or adapter-based interfaces, as it is a pure data structure.

Event Inputs

Not applicable, as `ST64B` is a data type and not a function block.

Event Outputs

Data Components

The structure `ST64B` consists of the following 64 byte variables:

Name	Type	Description
B_00	BYTE	BYTE 00
B_01	BYTE	BYTE 01
B_02	BYTE	BYTE 02
B_03	BYTE	BYTE 03
B_04	BYTE	BYTE 04
B_05	BYTE	BYTE 05
B_06	BYTE	BYTE 06
B_07	BYTE	BYTE 07
B_08	BYTE	BYTE 08
B_09	BYTE	BYTE 09
B_10	BYTE	BYTE 10
B_11	BYTE	BYTE 11
B_12	BYTE	BYTE 12
B_13	BYTE	BYTE 13
B_14	BYTE	BYTE 14
B_15	BYTE	BYTE 15
B_16	BYTE	BYTE 16
B_17	BYTE	BYTE 17
B_18	BYTE	BYTE 18
B_19	BYTE	BYTE 19
B_20	BYTE	BYTE 20
B_21	BYTE	BYTE 21
B_22	BYTE	BYTE 22
B_23	BYTE	BYTE 23
B_24	BYTE	BYTE 24
B_25	BYTE	BYTE 25
B_26	BYTE	BYTE 26
B_27	BYTE	BYTE 27
B_28	BYTE	BYTE 28
B_29	BYTE	BYTE 29
B_30	BYTE	BYTE 30

B_31	BYTE	BYTE 31
B_32	BYTE	BYTE 32
B_33	BYTE	BYTE 33
B_34	BYTE	BYTE 34
B_35	BYTE	BYTE 35
B_36	BYTE	BYTE 36
B_37	BYTE	BYTE 37
B_38	BYTE	BYTE 38
B_39	BYTE	BYTE 39
B_40	BYTE	BYTE 40
B_41	BYTE	BYTE 41
B_42	BYTE	BYTE 42
B_43	BYTE	BYTE 43
B_44	BYTE	BYTE 44
B_45	BYTE	BYTE 45
B_46	BYTE	BYTE 46
B_47	BYTE	BYTE 47
B_48	BYTE	BYTE 48
B_49	BYTE	BYTE 49
B_50	BYTE	BYTE 50
B_51	BYTE	BYTE 51
B_52	BYTE	BYTE 52
B_53	BYTE	BYTE 53
B_54	BYTE	BYTE 54
B_55	BYTE	BYTE 55
B_56	BYTE	BYTE 56
B_57	BYTE	BYTE 57
B_58	BYTE	BYTE 58
B_59	BYTE	BYTE 59
B_60	BYTE	BYTE 60
B_61	BYTE	BYTE 61
B_62	BYTE	BYTE 62
B_63	BYTE	BYTE 63

Data Outputs

Not applicable, as `ST64B` is a data type and its components do not function as “outputs” in the sense of a function block.

Adapters

Functionality

As a structured data type, `ST64B` does not have its own "function" in the sense of active behavior or logic. Its primary function is to define a memory area of 64 bytes and make it accessible in an organized form. This allows addressing individual bytes within the structure via their named variables (e.g., `ST64B.B_00`, `ST64B.B_15`). It serves as a passive data container.

Technical Features

The main feature of `ST64B` is its fixed size of exactly 64 bytes. Each byte is individually addressable via a unique variable name. This makes it ideal for situations requiring a strict data size and byte-by-byte manipulation of the content, such as when processing communication protocols or hardware registers. The sequential naming of the variables simplifies their allocation to a continuous memory area.

State Overview

As a pure data type, `ST64B` has no states in the sense of a state machine or a function block. The "state" of a `ST64B` instance is defined solely by the current values of the 64 contained byte variables. Each combination of these values represents a valid data state of the structure.

Application Scenarios

ST64B` is ideally suited for the following application scenarios:

- **Communication Buffer:** As a standard buffer for messages or packets with a fixed size of 64 bytes, e.g., in CAN bus messages (when multiple frames are needed), serial communication, or other protocols.
- **Hardware Register Map:** Mapping of a 64-byte range of hardware registers, where each byte variable corresponds to a specific register.
- **Data Serialization:** Storing raw data before serialization or after deserialization, especially when data is transmitted without explicit type information.
- **Generic Data Blocks:** Handling generic, fixed-size data blocks that can be interpreted flexibly, e.g., as part of larger protocols or data structures.

⚖️ Comparison with Similar Building Blocks

ST64B` is a basic data type and cannot be directly compared to function blocks. Within the data types, it is comparable to:

- **Other structured data types:** `ST64B` is a specific instance of `StructuredType`. There are similar structures with different numbers or types of elements (e.g., structures with 8, 16, or 32 bytes, or those containing integers, booleans, etc.). `ST64B` is characterized by its purely byte-wise division.
- **Arrays of BYTE:** An array of type `ARRAY[0..63] OF BYTE` would also store 64 bytes. The difference lies in the access method: `ST64B` uses named members (e.g., `myST64B.B_10`), while an array uses an index (e.g., `myArray[10]`). Named members can be advantageous in certain contexts for improved code readability or for directly addressing specific byte positions.

Conclusion

ST64B` is a simple yet powerful structured data type that provides a fixed data set of 64 bytes. Its strength lies in the clear definition and direct accessibility of each individual byte, making it an excellent choice for applications requiring precise control and interpretation of raw data blocks, particularly in communication and hardware interaction.

Numeric Index (utils)

...# Numeric Index (utils){index} single: Numerischer Index (utils) Here you will find all exercises from the `utils` workspace, listed in ascending order. ...

ARR_MAX

ARR_MAX



• – ○ • – ○ • – ○ •

Introduction

The function block `ARR_MAX` calculates the maximum value of a one-dimensional array of type `INT`. The implementation is in Structured Text (ST) according to IEC 61131-3 and is from the package `logiBUS::utils::dyn_arr`. The block is suitable for dynamic arrays of any size, with the lower and upper bounds determined at runtime.

Interface Structure

Event Inputs

Name	Type	With Var	Description
`REQ`	Event	`A`	Starts the calculation of the maximum. The array `A` must be valid and contain values at the time of the event.

Event Outputs

Name	Type	With Var	Description
`CNF`	Event	(empty), `A`	Confirms the end of the calculation. The event is triggered as soon as the maximum value has been determined. The array `A` remains valid during the event.

Data Inputs

Name	Type	Initial Value	Description
`A` (IN/OUT)	`ARRAY[*] OF INT`	–	One-dimensional integer array whose maximum is to be calculated. The size is determined dynamically at runtime.

Data Outputs

Name	Type	Initial Value	Description
`MAX`	`INT`	–	The determined maximum value of the array `A`. (The name `MAX` is derived from the function – in the XML definition, the output is unnamed but is treated as a function return value.)

Adapter

Omitted.

Functionality

1. **Start**: A signal at the event input `REQ` triggers the processing. The provided array `A` must be valid at this point.
2. **Initialization**: The variable `i` is set to the index of the first element (`LOWER_BOUND(A,1)`). `currentMax` receives the value of this first element.
3. **Iteration**: A loop called `FOR` iterates through all remaining elements of `A` (from index `LOWER_BOUND(A,1)+1` to `UPPER_BOUND(A,1)`). The current maximum value is updated using the standard function `MAX()`.
4. **Closure**: After the loop, `currentMax` is set as the return value of the function `ARR_MAX`. The event output `CNF` is sent, and the output signal `MAX` (interpreted here as a return value) is available.

Technical Features

- **Dynamic Array Boundaries**: The function uses `LOWER_BOUND` and `UPPER_BOUND` to determine the actual size of the array at runtime. This allows the function block to work with arrays of any length without requiring a fixed size to be specified.
- **No State Memory**: The function block is a pure function without internal state. Each call calculates the maximum value, independent of the previous call.
- **Typing**: The function block is specialized for the data type `INT`. A separate version would need to be created for other data types (e.g., `REAL`, `LINT`).
- **Error Handling**: If the array is empty (e.g., lower bound > upper bound), access to `A[i]` would be undefined. The user must ensure that the array contains at least one element.

State Overview

Since `ARR_MAX` is implemented as a **pure function** without a state machine (ECA algorithm), there are no explicit states. The process is strictly sequential within a single event cycle:

- **Waiting for REQ**
- **Calculation in progress** (no separate state signal, as the process is completed within one cycle)
- **CNF is being sent** (output of the maximum)

Therefore, an explicit state graph is not required.

Application Scenarios

- **Data Analysis**: Determining the peak value in measurement series (e.g., temperature, pressure, speed).
- **Monitoring**: Detecting threshold violations by comparing them to the maximum of an array.
- **Signal Processing**: Determining the maximum level in audio or communication signals.
- **Control Engineering**: Selecting the highest value from multiple sensor data points (e.g., for priority decisions).

Comparison with Similar Function Blocks

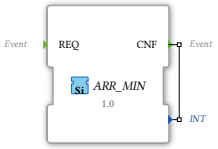
Function Block	Function	Difference from ARR_MAX
`ARR_MIN`	Minimum of an Array	Analogous, but minimum instead of maximum.
`ARR_SUM`	Sum of Array Elements	Different mathematical operation; no comparison between elements.
`ARR_MEAN`	Arithmetic Mean	Requires additional division by the number of elements; not just comparison.
`MAX` (IEC 61131-3)	Maximum of two values	Works only with scalars, not arrays. `ARR_MAX` extends this to entire arrays.

ARR_MAX` combines array traversal capability with simple maximum determination and is therefore a specialized, yet reusable function block for dynamic arrays.

Conclusion

The function block `ARR_MAX` offers a compact and efficient way to determine the maximum value of a dynamic integer array. Using `LOWER_BOUND` and `UPPER_BOUND` makes it flexible and robust against different array sizes. Due to its purely functional design without state storage, it is ideally suited for use in data flow-oriented automation applications.

ARR_MIN



• – ○ • – ○ • – ○ •

Introduction

The function block **ARR_MIN** determines the minimum of all elements in a one-dimensional array of type `INT`. The block expects a dynamic array (`ARRAY[*]`) as an input/output parameter and outputs the smallest element found via a separate data output. The calculation is triggered by the event `REQ`; the result is acknowledged upon completion with the event `CNF`.

Interface Structure

Event Inputs

Event	With Data
`REQ`	`A`

Event Outputs

Event	With Data
`CNF`	`A` (and empty data output)

Data Inputs

Name	Type	Comment
`A`	`ARRAY[*] OF INT`	Input Array (Input/Output)

Data Outputs

Name	Type	Comment
(Unnamed)	`INT`	Minimum value of the array

Adapter

No adapters available.

Functionality

1. The algorithm initializes a temporary variable `currentMin` with the value of the first array element (determined via `LOWER_BOUND(A,1)`).
2. The array is then traversed from index `LOWER_BOUND(A,1)+1` to `UPPER_BOUND(A,1)`.
3. In each iteration, the current value is compared to `currentMin`, and the minimum is updated using the function `MIN()`.
4. After the loop completes, the final minimum value is output via the unnamed data output, and the event `CNF` is sent.

The calculation is performed sequentially within a single execution step; the runtime increases linearly with the array size.

Technical Features

- **Dynamic Array**: The function block uses `ARRAY[*] OF INT`, meaning the actual array size is determined at runtime using the system's built-in functions `LOWER_BOUND` and `UPPER_BOUND`.
- **Package Assignment**: The function block belongs to the package `logiBUS::utils::dyn_arr` and is intended for use in environments that support dynamic arrays.
- **Input/Output Parameters**: The parameter `A` is declared as `VAR_IN_OUT` – the array is both read and (here without modification) passed on via the event output `CNF`.
- **No State Machine**: The function block operates purely functionally; there are no internal states or delays.

State Overview

The function block only has two implicit states:

- **Idle**: Waiting for a `REQ` event.
- **Processing**: After `REQ` arrives, the calculation is executed, immediately followed by `CNF`, and the function block returns to the idle state.

There are no multiple states, wait loops, or time-controlled processes.

Application Scenarios

- **Sensor Field Analysis**: Determining the lowest temperature in a series of measurements.
- **Quality Control**: Searching for the minimum reject rate in a production batch.
- **Optimization**: Determining the most favorable price from a list of offers.
- **Comparison Operations**: As a basic building block for further statistical evaluations (e.g., together with maximum and sum).

Comparison with Similar Blocks

- **MIN (IEC 61131-3)**: The classic MIN block compares two scalar values. `ARR_MIN` extends this concept to an entire array, thus eliminating the need for iterative chaining.
- **ARR_MAX**: A logical counterpart to the maximum; The algorithm differs only in the comparison function (`MIN` vs. `MAX`).
- **Statistical Building Blocks**: `ARR_MIN` is a special case of general aggregation functions (e.g., SUM, MEAN). Its focus is on finding the minimum, enabling simple and fast implementation.

Conclusion

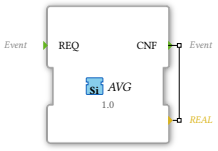
ARR_MINoffers a compact and efficient solution for determining the minimum in dynamic integer arrays. Thanks to its simple request/confirm protocol and clean package structure, it can be easily integrated into existing 4diac industrial applications. UsingLOWER_BOUND/UPPER_BOUND` makes it independent of the actual array size and allows for flexible adjustments at runtime.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

AVG



• – ◦ • – ◦ • – ◦ •

Introduction

The **AVG** (Array Average) function block calculates the arithmetic mean (average) of all elements in an integer array. It is designed for use in control engineering and enables rapid statistical analysis of measured values or process data.

Interface Structure

Event Inputs

Event	With Data
`REQ`	`A`

Event Outputs

Event	With Data
`CNF`	*(Result Output)*, `A`

Data Inputs

Name	Type	Initial Value	Comment
A	ARRAY[*] OF INT	–	Array of integers whose average is calculated (InOut variable)

Data Outputs

Name	Type	Initial Value	Comment
(no name)	REAL	–	Result of the average value

Adapters

None.

Functionality

1. The function block is activated by the event REQ. The data array A is then used.
2. Internally, a loop is executed from LOWER_BOUND(A,1) to UPPER_BOUND(A,1) to sum all elements.
3. The number of elements is determined from UPPER_BOUND(A,1) - LOWER_BOUND(A,1) + 1.
4. The sum is accumulated as DINT and then divided by the number of elements. The result is converted into a value REAL and made available at the output.
5. After the calculation is complete, the event CNF is triggered. The output value is then available.

Technical Features

- **Dynamic Array Size**: The function block accepts arrays of arbitrary length (indicated by ARRAY[*]), allowing for flexible adaptation to different data volumes.
- **Type Conversions**: The sum is represented as DINT (double word) to prevent overflows with large arrays. Division is performed after conversion to REAL, resulting in a floating-point value.
- **No State Machine**: The function block is purely functional – REQ is immediately followed by CNF. There are no internal states or delays.

State Overview

Since this is a **combinatorial function block**, there are no explicit states. The process is as follows:

- Initial: Wait for REQ
- After REQ: Calculation and immediate output of CNF

Application Scenarios

- **Averaging of Sensor Data**: Smoothing of measured values (e.g., temperature, pressure) over a defined measurement cycle.
- **Average Calculation in Batch Processes**: Determination of the average fill level, weight, or other process parameters.
- **Statistical Evaluation in Quality Control**: For example, calculating the diameter of multiple test specimens as a floating-point value for accuracy analysis.

Comparison with Similar Function Blocks

Feature	**AVG**	Typical IEC 61131-3 AVG Function Block
Parameterization	Dynamic via array (no limit)	Often static array size or lists
Event-driven	Yes (REQ/CNF)	No (pure function)
Output type	REAL	Usually REAL or LREAL
Overflow protection	Sum as DINT	Dependent on implementation

This function block combines IEC 61499 event control with the compact logic of an IEC 61131-3 average function block, making it particularly suitable for distributed automation systems.

Conclusion

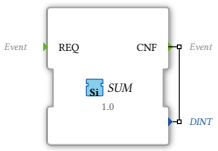
The **AVG** function block offers a simple and efficient way to calculate the average of an integer array in real time. Its dynamic array support and clear event interface make it a universal tool for averaging in industrial control applications. The seamless integration into the 4diac IDE and the transparent calculation logic ensure high reusability and ease of understanding.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

SUM



• – ◦ • – ◦ • – ◦ •

Introduction

The **SUM** function block allows you to calculate the sum of all elements in a dynamic array. It is triggered by an event at the REQ input and, after successful calculation, outputs the result via the data output and a confirmation event via CNF.

Interface Structure

Event Inputs

Name	Description
REQ	Starts the sum calculation of array A.

Event Outputs

Name	Description
CNF	Signals the completion of the calculation and the validity of the result.

Data Inputs

There are no pure data inputs. The required array is provided via the **InOut** port `A`.

Data Outputs

Name	Type	Description
(empty)	DINT	Resultant sum of all array elements (type DINT).

Adapters

None.

Functionality

The FB `SUM` accepts a one-dimensional array `A` of type `INT` with arbitrary length (declared as `ARRAY[*]`). When the event `REQ` is triggered, the array's contents are summed element by element. The sum is accumulated in a temporary variable of type `DINT` to prevent overflow. After iterating completely over all valid indices (from `LOWER_BOUND(A,1)` to `UPPER_BOUND(A,1)`), the result is passed to the output (data type `DINT`), and the event `CNF` is triggered.

Important Note: Although `A` is declared as an **InOut** variable, its value is not modified during the calculation. It functions solely as an input in this function block.

Important Note:

Technical Features

- **Dynamic Array Size:** The function block supports variable-length arrays by using the functions `LOWER_BOUND` and `UPPER_BOUND`, as specified by IEC 61131-3.
- **Type Extension:** The sum is returned as a `DINT` value (32-bit), allowing even larger sums of `INT` values (16-bit) to be represented without overflow.
- **Event-Driven Execution:** Computation is performed only when needed, enabling efficient resource utilization.

State Overview

The function block does not have an explicit state machine. Its behavior can be described as follows:

1. **Waiting:** No event is active – initial state.
2. **Calculation:** Entry after receiving `REQ`; summation of the array.
3. **Acknowledgement:** After completion of the calculation, `CNF` is sent; return to the wait state.

Application Scenarios

- Summing sensor readings over a specific period.
- Calculating totals in a production plant (e.g., quantities, fill levels).
- Preprocessing data arrays before further logical or arithmetic processing.

Comparison with Similar Function Blocks

- **SUM in IEC 61131-1:** This function block is closely based on the function of the same name from IEC 61131-3, but additionally enables event-driven execution and direct integration into the 4diac event model.
- **ADD / MULT:** While `ADD` or `MULT` process individual number pairs, `SUM` allows the processing of an entire array in one step, simplifying the application logic.

Conclusion

The function block `SUM` offers a simple and efficient way to calculate the sum of all elements in a dynamic array. Its event-driven execution and robust handling of different array sizes make it particularly suitable for industrial control tasks where large amounts of data need to be aggregated.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

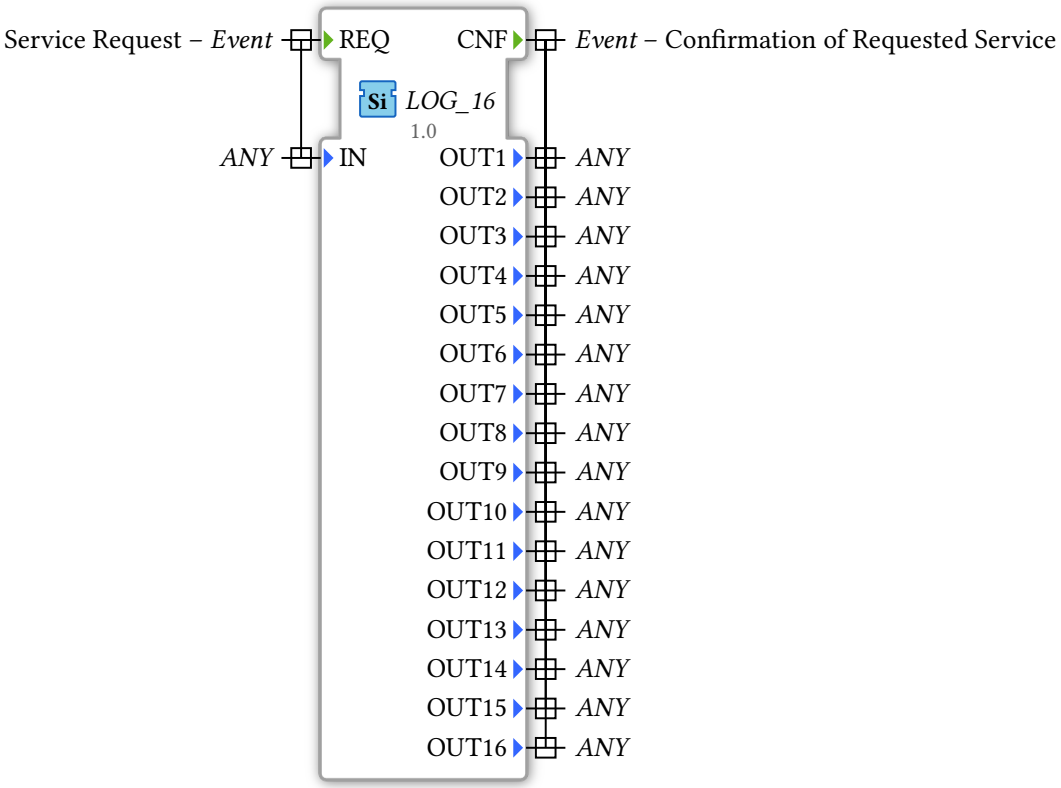
logging

LOG_16

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `LOG_16` is a ring logger designed for the cyclic recording of data of any type (`ANY`). It serves to hold incoming values in a buffer with 16 memory locations, overwriting older entries on new calls (ring buffer principle). This block is particularly suitable for logging process data or states in real-time controllers.



Interface Structure

Event Inputs

- **REQ (Service Request)**: Triggers a logging operation. When this event occurs, the current value at data input `IN` is written to the ring buffer.

Event Outputs

- **CNF (Confirmation of Requested Service)**: Triggered after successful processing of the `REQ` event. This event confirms the logging operation and simultaneously makes all 16 stored values available at the data outputs.

Data Inputs

- **IN (ANY)**: The data value to be written to the ring buffer upon a `REQ` event. The data type is arbitrary (`ANY`).

Data Outputs

- **OUT1 to OUT16 (ANY)**: The 16 outputs that represent the entire current contents of the ring buffer. `OUT1` contains the most recent entry (the last logged entry, `IN`), and `OUT16` contains the oldest. With each new log operation (`REQ`), all values in the buffer are shifted by one position.

Adapter

This function block has no adapter interfaces.

Functionality

LOG_16implements a fixed-size, first-in-first-out (FIFO) ring buffer with 16 elements. Upon each incomingREQ` event, the following algorithm is executed:

1. The current value at input `IN` is stored as the newest entry.
2. All previously stored values are shifted one position backward (toward `OUT16`).
3. The value that was previously at position 16 (`OUT16`) is discarded.
4. The confirmation event `CNF` is triggered.
5. The new 16 buffer contents are output at outputs `OUT1` (newest) to `OUT16` (oldest).

Technical Features

- **Generic Data Type**: The use of the `ANY` data type for inputs and outputs makes the function block extremely flexible. It can be instantiated and used with any data type (e.g., `BOOL`, `INT`, `REAL`, `STRING`, or even structured types).
- **Fixed Buffer Size**: The buffer size is fixed at 16 entries and is not configurable.
- **Immediate Output**: During each logging operation, the entire buffer content is updated at the outputs and confirmed with the `CNF` event.
-

State Overview

The function block does not possess a persistent internal state in the sense of a state machine, apart from the ring buffer itself. Its behavior is purely reactive: A `REQ` event is always followed by a buffer update and the output of `CNF` with the current data.

Application Scenarios

- **Process Value Logging**: Short-term recording of sensor data (e.g., temperature profile of the last 16 cycles).
- **Error History**: Storage of the last 16 error codes or alarm messages.
- **Data Preprocessing**: Provision of a sliding window of the last 16 values for subsequent calculations (e.g., in another function block).
- **Debugging**: Easy monitoring of variable behavior during development and commissioning.
-

⚖️ Comparison with similar modules

- **`E_DELAY` / Delay Modules**: These modules output an input value only after a defined delay. The `LOG_16`, on the other hand, stores a history of multiple values and outputs them immediately, but in order of recency.
- **`FIFO` Modules**: Classic FIFO (First-In-First-Out) storage devices often have variable lengths and a separate read/write interface. The `LOG_16` is a special fixed-length FIFO (16) that automatically outputs and overwrites its entire contents with each write operation.
- **Simple `LOG` Modules**: Simple loggers without a buffer typically write only a single value. The strength of `LOG_16` lies in its circular history.

🛠️ Related Exercises

- Exercise_122
- Exercise_122b

Conclusion

LOG_16` is a useful and generic function block for basic logging and buffering tasks in 4diac FORTE applications. Its strengths lie in its simplicity, generic type support, and deterministic behavior. The fixed buffer size of 16 is sufficient for many monitoring tasks; however, applications requiring a different buffer depth or selective reading will need adapted or extended blocks.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

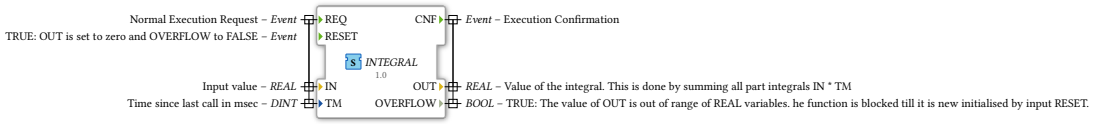
math

INTEGRAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `INTEGRAL` approximates the time integral of an input signal. The integration is performed by summing partial areas that result from the product of the input value and the time elapsed since the last call. This block is useful for applications where cumulative values such as energy consumed, quantities conveyed, or distances traveled need to be determined.



Interface Structure

Event Inputs

- **REQ**: Starts the normal execution of the integration. The data associated with this event, `IN` and `TM`, are processed.
- **RESET**: Resets the integrator. The output `OUT` is set to zero, and the overflow flag `OVERFLOW` is set to `FALSE`.

Event Outputs

- **CNF**: Signals confirmation of execution. This event is triggered after both `REQ` and `RESET` and returns the current values of `OUT` and `OVERFLOW`.

Data Inputs

- **IN** (`REAL`, initial value: `0.0`): The input value to be integrated.
- **TM** (`DINT`, initial value: `0`): The time elapsed since the last call in milliseconds (ms). This value must be positive for integration to occur.

Data Outputs

- **OUT** (`REAL`, initial value: `0.0`): The current value of the integral. It is calculated by summing all partial integrals (`IN * TM`). The unit is `[IN] * Sekunden` because the time value `TM` is internally converted from milliseconds to seconds.
- – **OVERFLOW** (`BOOL`, initial value: `FALSE`): Indicates an overflow. If this output is `TRUE`, the value of `OUT` has exceeded the representable range of the `REAL` data type. The integration function is blocked until it is reinitialized by a `RESET` event.

Adapters

This function block does not use adapters.

Functionality

The `INTEGRAL` function block has two primary operating states, which are controlled by incoming events:

1. **Integration (`REQ`)**: Upon a `REQ` event, the algorithm `REQ` is executed. First, it checks whether a previous overflow (`OVERFLOW = FALSE`) has occurred. An overflow check is then performed: If adding the next partial integral (`IN * TM`) to the current `OUT` value would exceed the limits of the `REAL` data type (approximately ±1.0E38), `OVERFLOW` is set to `TRUE`. Otherwise, and only if the time difference `TM` is greater than 0, the partial integral is calculated, converted to seconds (`/ 1000.0`), and added to the overall integral `OUT`.

If the value of the partial integral would exceed the limits of the `REAL` data type (approximately ±1.0E38), then `OVERFLOW` is set to `TRUE`. Otherwise, and only if the time difference `TM` is greater than 0, the partial integral is calculated, converted to seconds (`/ 1000.0`), and added to the total integral `OUT`.

1. **Reset (`RESET`)**: Upon a `RESET` event, the algorithm `RESET` is executed. This resets the integral value `OUT` to `0.0` and the overflow flag `OVERFLOW` to `FALSE`.

After the execution of each algorithm, an output event `CNF` containing the current values is always generated.

Technical Features

- **Overflow Handling**: The function block has integrated overflow detection. If a potential overflow is detected, the integration is stopped (`OVERFLOW = TRUE`) to prevent undefined behavior. A manual reset is required to resume operation.
- **Time Basis**: The integration is based on the user-specified time difference `TM` in milliseconds. The function block itself does not have an internal timer. This allows for flexible adaptation to different cycle times of the surrounding control program.

- **Initialization**: During the first initialization of the function block, `OUT = 0.0` and `OVERFLOW = FALSE` are present.

State Overview

The function block is a Simple FB and has explicitly defined execution states (ECStates) for the two possible events:

- **State REQ**: Activated upon arrival of the `REQ` event. Executes the integration algorithm.
- **RESET State**: Activated upon arrival of the `RESET` event. Executes the reset algorithm.

Both states result in the output of the `CNF` event after their execution.

Application Scenarios

- **Energy Measurement**: Calculation of total energy consumption from a momentary power measurement (`IN` = power in kW, `OUT` = energy in kWh).
- **Flow Measurement**: Determination of the total volume of liquid or gas delivered from a flow sensor (`IN` = flow rate in l/min, `OUT` = volume in l).
- **Velocity Integration**: Determining the distance traveled from a velocity signal (`IN` = velocity in m/s, `OUT` = distance in m).
- **Charge Calculation**: Integrating a current signal to determine the charge flow (`IN` = current in A, `OUT` = charge in Ah).

⚖️ Comparison with Similar Modules

Compared to simpler summation modules (e.g., `ADD`), `INTEGRAL` is distinguished by its explicit consideration of time as an integration variable. Unlike more complex control blocks with integrated timers (such as certain PID implementations), `INTEGRAL` is a pure, discrete-time integrator that expects the time difference as an input variable and is therefore clocked externally. This makes it more flexible in systems with variable or non-periodic cycle times.

🛠️ Related Exercises

- Exercise_072c

Conclusion

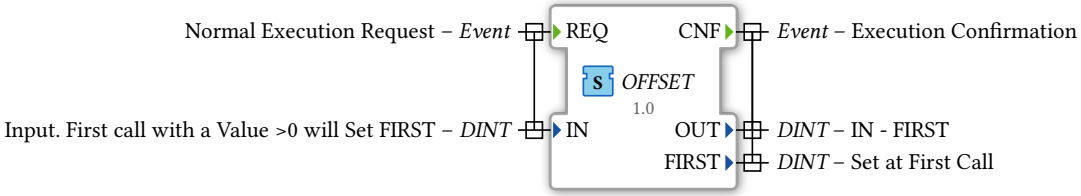
The `INTEGRAL` function block is a fundamental and robust tool for the time-integrated processing of signals in IEC 61499 controllers. Its clear interface, integrated overflow protection, and flexible handling of the time difference make it a reliable component for a wide variety of measurement and control tasks. The need to determine and provide the elapsed time externally offers maximum flexibility, but also requires correct implementation in the surrounding application program.

OFFSET

• – ◦ • – ◦ • – ◦ •

Introduction

The OFFSET function block is used to determine and apply an offset value. When an event with a non-zero input value is first processed, this value is stored as a reference point (`FIRST`). On each subsequent call, the block calculates the difference between the current input value and this stored reference value. This block is useful for measuring relative changes compared to an initial starting value.



Interface Structure

Event Inputs

- **REQ**: Starts the normal execution of the block. Linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the calculation. Linked to the data outputs `OUT` and `FIRST`.

Data Inputs

- **IN** (DINT): The input value. On the first call where this value is greater than 0, it is stored as the reference value (`FIRST`). Initial value is 0.

Data Outputs

- **OUT** (DINT): The result of the calculation `IN - FIRST`. Initial value is 0.
- **FIRST** (DINT): The reference value stored on the first relevant call. Initial value is 0.

Adapters

This function block does not use any adapters.

Functionality

The OFFSET block is a simple function block with a single state (`REQ`) and an algorithm of the same name. Upon each occurrence of a `REQ` event, the algorithm `REQ` is executed:

1. **Checking and Setting the Reference Value:** The algorithm checks whether the output `FIRST` still has its initial value of 0. If so, **and** the current input value `IN` is greater than 0, `FIRST` is set to the value of `IN`. This condition (`IN > 0`) is implicitly included in the algorithm's logic, since `FIRST` is only changed from 0 to another value during the first processing of a `IN > 0` event.
2. **Offset Calculation:** Regardless of the previous step, the output `OUT` is calculated as the difference between the current input `IN` and the stored reference value `FIRST` (`OUT := IN - FIRST`).
3. **Confirmation:** After the calculation, the `CNF` event is triggered, including the current values of `OUT` and `FIRST`.

Technical Features

- The reference value `FIRST` is set exactly once, specifically during the first `REQ` event where the data input `IN` has a value greater than 0. As long as `IN` is 0 or less during the initial calls, `FIRST` remains 0.
- Once `FIRST` has been set, it remains unchanged for the entire duration of the function block's runtime.
- The calculation `OUT = IN - FIRST` occurs with *every* call, even if `FIRST` is still 0. This means that `OUT` remains identical to `IN` until the reference value is set.

State Overview

The function block has a single, eponymous state:

- **REQ:** This state is always active. Every incoming `REQ` event triggers the execution of the associated algorithm and immediately triggers the `CNF` event. There are no internal state transitions.

Application Scenarios

- **Incremental Path or Position Measurement:** Setting a zero point (`FIRST`) after a reference run and subsequent measurement of the relative movement (`OUT`).
- **Compensation for Base Loads or Zero Deviations:** Capture an initial base value (e.g., sensor base load) and calculate the pure change relative to this base value.
- **Relative Time Measurement:** Store a start time and calculate the elapsed time since that start.

⚖️ Comparison with Similar Function Blocks

- **SUB / DINT_SUB:** A simple subtractor always calculates the difference between two current input values. OFFSET, on the other hand, stores one of the values internally as a constant reference point.
- **DELAY / TON:** Time delay functions operate on the time axis. OFFSET works with data values and has no time-based behavior.
- **SR / RS:** Memory functions (Set/Reset) hold a binary state. OFFSET stores an integer value (`DINT`) and uses it for an arithmetic operation.

Conclusion

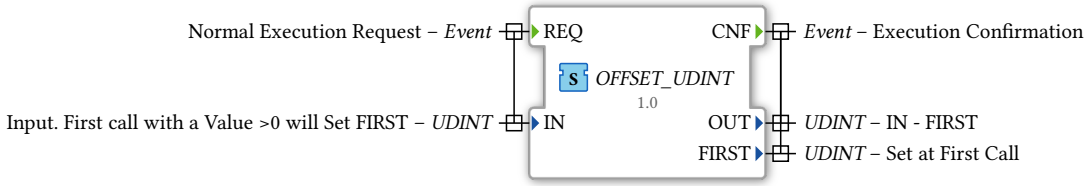
The OFFSET function block is a specialized and simple function block for determining a relative offset. Its strength lies in the one-time, automatic acquisition of a starting or reference value and the continuous calculation of the deviation from it. It is particularly suitable for applications where a system must be initially "zeroed" and only subsequent changes are relevant.

OFFSET_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OFFSET_UDINT` is used to calculate an offset value. On the first execution with an input value greater than zero, the block stores this value as a reference (`FIRST`). On each subsequent call, the difference between the current input value and this stored reference value is calculated as the output (`OUT`). This is particularly useful for determining relative changes compared to an initial starting value.



Interface Structure

Event Inputs

- **REQ**: Starts the normal execution of the algorithm. Linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the calculation. Linked to the data outputs `OUT` and `FIRST`.

Data Inputs

- **IN** (UDINT): The input value. On the first call with a value > 0, this value is stored in `FIRST`. The initial value is `0`.

Data Outputs

- **OUT** (UDINT): The result of the calculation `IN - FIRST`. The initial value is `0`.
- **FIRST** (UDINT): The reference value stored on the first call (with `IN` > 0). The initial value is `0`.

Adapters

This function block does not use any adapters.

Functionality

The `OFFSET_UDINT` is a simple function block (Simple FB) with a single state (`REQ`). Whenever the `REQ` event occurs, the associated algorithm `REQ` is executed.

The algorithm first checks whether the output `FIRST` still has the initial value `0`. If so, the current value of `IN` is stored in `FIRST`. The difference `OUT = IN - FIRST` is then calculated. After the calculation is complete, the `CNF` event is triggered, outputting the updated values of `OUT` and `FIRST`.

Important: The check `FIRST = 0` ensures that the value is only set on the *first* call where `IN` has a value greater than 0. A subsequent `IN` call from `0` will no longer change `FIRST`.

Technical Features

- **Data Type:** The block works exclusively with unsigned 32-bit integers (`UDINT`).
- **Initialization:** Both `FIRST` and `OUT` are initialized with `0`. The block is ready for operation immediately after startup.
- **One-Time Setting:** The variable `FIRST` is set exactly once (on the first `REQ` with `IN` > 0) and then remains constant for the lifetime of the function block instance, unless the function block is reinitialized.
-

State Overview

The function block has a single, stable state:

- **REQ:** Waits for the incoming `REQ` event. Upon receipt, the algorithm is executed, and immediately afterward, the `CNF` event is triggered. The function block remains in the `REQ` state.

Application Scenarios

- **Reference Point Establishment in Measurement Chains:** In a system that provides absolute position values (e.g., from an encoder), this block can store the first valid measurement as the zero point (offset). All subsequent values are output relative to this starting point.
- **Initial Value Compensation:** To eliminate a system-related default offset (e.g., a bias) during the first valid measurement.
- **Counter Start Normalization:** If a counter does not start at 0, the first value can be stored and subtracted from all subsequent values to simulate counting from 0.

⚖️ Comparison with Similar Blocks

- **SUB / DIF Blocks:** Standard subtraction blocks always calculate the difference between two variable inputs. `OFFSET_UDINT`, however, freezes one of the operands (`FIRST`) after the first operation and uses it as a fixed subtrahend thereafter.
- **Shift Registers (FIFO) or Delay Blocks:** These store a history of values. `OFFSET_UDINT`, on the other hand, only permanently stores a single, initial value.
- **Blocks with Markers (SR, RS):** These store a binary state. `OFFSET_UDINT` stores a specific numeric (`UDINT`) starting value.

🛠️ Related Exercises

- Exercise_072c

Conclusion

The `OFFSET_UDINT` function block is a specialized and efficient function block for applications where a single measured starting value serves as the basis for all subsequent relative calculations. Its strength lies in its simplicity and the reliable, one-time setting of the reference value, eliminating the need for manual logic to detect the "first cycle." It is ideal for initialization and normalization tasks in control applications.

quarter

🎧 Podcast

- The Secret of the 2-Bit Quarter: Efficiency in the CAN Bus for Commercial Vehicles
- QUARTER

BOOLS_TO_QUARTERS

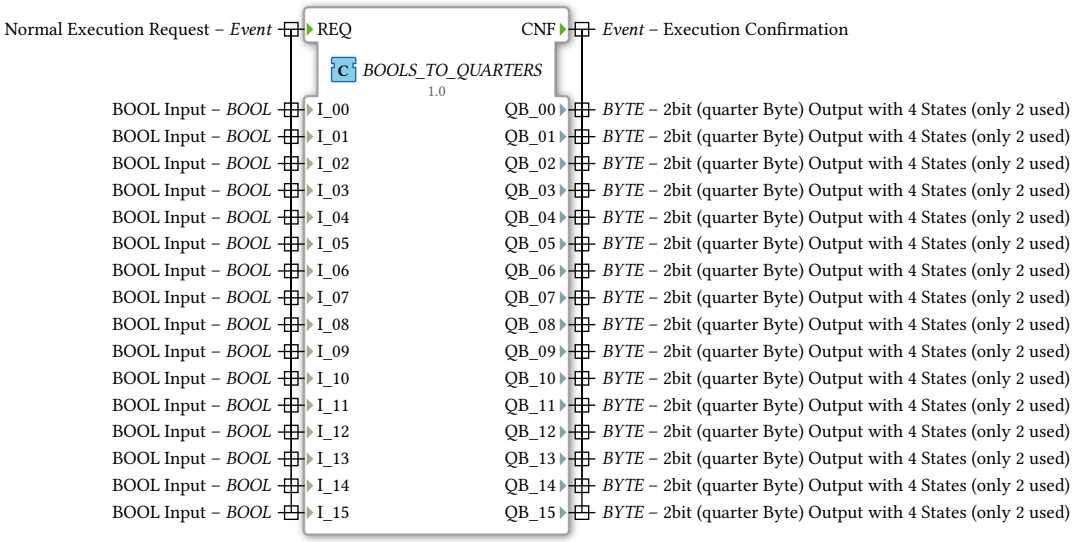

Composite FB wrapping other FBs
REQ
I_00
I_01
I_02
I_03
I_04
I_05
I_06
I_07
I_08
I_09
I_10
I_11
I_12
I_13
I_14
I_15
CNF
QB_00
QB_01
QB_02
QB_03
QB_04
QB_05
QB_06
QB_07
QB_08
QB_09
QB_10
QB_11
QB_12
QB_13
QB_14
QB_15
BOOL_TO_QUARTER_00
BOOL_TO_QUARTER_01
BOOL_TO_QUARTER_02
BOOL_TO_QUARTER_03
BOOL_TO_QUARTER_04
BOOL_TO_QUARTER_05
BOOL_TO_QUARTER_06
BOOL_TO_QUARTER_07
BOOL_TO_QUARTER_08
BOOL_TO_QUARTER_09
BOOL_TO_QUARTER_10
BOOL_TO_QUARTER_11
BOOL_TO_QUARTER_12
BOOL_TO_QUARTER_13
BOOL_TO_QUARTER_14
BOOL_TO_QUARTER_15
REQ
CNF
B
BOOL_TO_QUARTER
I

🎧 Podcast

- QUARTER

• – ◦ • – ◦ • – ◦ •

The function block `BOOLS_TO_QUARTERS` is a composite function block (FB) that converts 16 individual Boolean input signals into a special 2-bit format called “Quarter Byte.” It serves as a bundle and serial execution point for several basic conversion functions and is intended for applications where many binary states need to be converted into a compact, multi-valued control format.



- **REQ (Normal Execution Request):** Starts the processing chain. Upon receiving a REQ event, all associated data inputs (`I_00` to `I_15`) are read and the conversion is initiated.
- **CNF (Execution Confirmation):** This event is output after all 16 internal conversions are complete. It confirms the completion of the operation and provides the calculated quarter-byte values (`QB_00` to `QB_15`) to the downstream application.
- **I_00 to I_15 (BOOL Input):** 16 independent Boolean inputs (`BOOL`). Each represents a binary switching state (TRUE/FALSE). The initial value of all inputs is `FALSE`.
- **QB_00 to QB_15 (2-bit (quarter byte) output):** 16 outputs of type `BYTE`. Each output encodes the result of converting the corresponding Boolean input into a quarter byte. A quarter byte uses only the lower two bits of a byte and can theoretically represent four states. In this implementation, two states are primarily used, defined by the constants `quarter::COMMAND_DISABLE` and its counterpart. The initial value of all outputs is `quarter::COMMAND_DISABLE`.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use any adapter interfaces.

BOOLS_TO_QUARTERSis a composite function block (FB) internally composed of 16 instances of a basic function blockBOOL_TO_QUARTER`. Each instance is responsible for converting a single Boolean value.

The functionality follows a serial chain principle:

1. The incoming `REQ` event triggers the first internal instance, `BOOL_TO_QUARTER_00`.

2. This instance reads its assigned data input, `I_00`, performs the conversion, and sets its output, `QB_00`.
3. After completing its operation, `BOOL_TO_QUARTER_00` generates a `CNF` event, which is directly passed on as a `REQ` event to the next instance (`BOOL_TO_QUARTER_01`).
4. This process cascades through all 16 instances.
5. The last instance (`BOOL_TO_QUARTER_15`) passes its final `CNF` event to the `CNF` output of the enclosing `BOOLS_TO_QUARTERS` block. At this point, all 16 quarter-byte outputs (`QB_00` to `QB_15`) are available with their new values.

The data paths are organized in parallel: Each Boolean input `I_xx` is directly connected to the corresponding `I` input of the internal instance, and each `QB` output of an instance is directly connected to the corresponding `QB_xx` output of the composite FB.

- **Serial Execution:** The 16 conversions are executed sequentially, not in parallel. This results in a defined, but not simultaneous, update of the outputs. The total cycle time is the sum of the execution times of all 16 internal blocks.
- **Composite Structure:** This block primarily serves to consolidate and simplify the wiring in higher-level applications. The actual logic resides in the embedded `BOOL_TO_QUARTER` function blocks.

As a composite function block without its own explicit state machine, `BOOLS_TO_QUARTERS` does not possess an internal state in the strict sense. Its behavior is entirely determined by the cascade of its subordinate blocks and their states. The block can be in one of two macroscopic states:

1. **Idle:** Waiting for a `REQ` event. All outputs retain their last value.
2. **Processing:** A `REQ` event passes through the cascade of the 16 internal blocks. During this phase, the outputs are updated sequentially.

- **Control of compact value-added actuators:** For actuators or drivers that expect control commands not as simple on/off signals, but as 2-bit commands (e.g., on/off/error reset/emergency stop).
- **Data compression for bus communication:** Before transmission via fieldbuses, where many binary signals must be packed into a space-saving byte- or word-oriented protocol.
- **Interface to legacy systems:** As an adapter between modern IEC 61499 controllers and older systems that expect or deliver data in a special quarter-byte format.
-
- **Compared to `BOOL_TO_QUARTER`:** `BOOLS_TO_QUARTERS` is essentially an array of 16 `BOOL_TO_QUARTER` blocks with a hard-wired serial event chain. While `BOOL_TO_QUARTER` performs a single conversion, `BOOLS_TO_QUARTERS` aggregates many such conversions into a reusable building block.
- **Compared to Generic Pack Blocks (e.g., `BOOLx_TO_BYTE`):** Blocks like `BOOL8_TO_BYTE` pack multiple BOOL values into the bits of a single byte. In contrast, `BOOLS_TO_QUARTERS` generates a separate (albeit only partially used) byte for each input. There is no bit packing into a shared byte, but rather a one-to-one mapping to a special encoding format.
- Exercise_060

The `BOOLS_TO_QUARTERS`The function block offers a convenient and pre-configured solution for serially converting a large number of Boolean signals to the quarter-byte format. Its composite nature makes it easy to understand and use, as it abstracts the wiring of 16 individual blocks and their event logic. Serial processing is a crucial feature that must be considered for real-time applications. The block is ideal for specific applications requiring the quarter-byte format, but less suitable for general bit packing or unpacking operations.

Functionality

Application Scenarios

State Overview

Technical Features

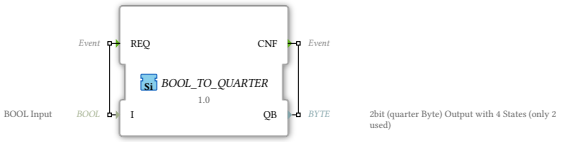
⚖️ Vergleich mit ähnlichen Bausteinen

🛠️ Zugehörige Übungen

Conclusion

BOOL_TO_QUARTER

🎧 Podcast



- QUARTER

• – ○ • – ○ • – ○ •

The function block `BOOL_TO_QUARTER` converts a binary BOOL signal into a special, predefined quarter byte. It translates the logical states `TRUE` and `FALSE` into corresponding, semantically meaningful byte constants, typically used for control commands (e.g., ENABLE/DISABLE). This block is part of the `logiBUS::utils::quarter` library.

- **REQ**: This event triggers the conversion function. Upon its arrival, the current value at the data input `I` is read and processed.
- **CNF**: This event signals the completion of the conversion. It is output along with the calculated output value `QB`.
- **I** (BOOL, Initial value: `FALSE`): The binary input value to be converted.
- **QB** (BYTE, Initial value: `quarter::COMMAND_DISABLE`): The output value as a byte. The block uses a so-called “quarter byte” (2 bits), which can represent four states, of which only two are used in this implementation. The specific value is derived from the constants of the imported library.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

This function block has no adapter interfaces.

Functionality

When triggered by the `REQ` event, the block evaluates the value at the input `I`. The processing is performed via a `CASE` instruction:

- If `I` has the value `BOOL#TRUE`, the output `QB` is set to the constant `quarter::COMMAND_ENABLE`.

- In all other cases (i.e., by default with `FALSE`), the output `QB` is set to the constant `quarter::COMMAND_DISABLE`.

Immediately after the assignment, the `CNF` event is output along with the new value of `QB`.

- **Library Dependency**: The block imports and uses constants (`quarter::COMMAND_ENABLE`, `quarter::COMMAND_DISABLE`) from the `logiBUS::utils::quarter::const::quarter` package. The actual byte values are therefore defined there and can be changed centrally.
- **Quarter Byte Concept**: The output is declared as `BYTE`, but conceptually only uses the lower two bits (a "quarter byte"). This allows for later expansion to include additional states without having to change the interface.
- **Initial Values**: Both the input `I` and the output `QB` have defined initial values (`FALSE` and `COMMAND_DISABLE`, respectively), which ensures deterministic startup behavior.

The block does not have an internal state in the sense of a memory. It behaves purely combinatorially with respect to the input data when the `REQ` event occurs. The output `QB` retains its last assigned value until the next execution cycle.

- **Actuator Control**: Conversion of a simple enable signal (`BOOL`) into a device-specific enable/disable command, which is sent as a byte via a fieldbus protocol.
- **Interface Adaptation**: Connection of logic blocks that work with BOOL values to control components that expect commands in quarter-byte format.
- **Code Standardization**: Ensuring that the same centrally defined constants for enable/disable commands are always used in a project.
-
- **`BOOL_TO_BYTE`**: A generic converter that typically maps `TRUE` to `1` and `FALSE` to `0`. `BOOL_TO_QUARTER` is more specialized and uses project-specific, semantic constants instead of numeric values.
- **Direct assignment**: The functionality could also be replicated by direct assignment in ST code (`QB := I`), but this would eliminate the advantages of centralized constant definition and a clear interface (events).
- Exercise_055
- Exercise_056

The `BOOL_TO_QUARTER` function block is a specialized but useful converter for control applications. It offers a clean interface with event-driven control, uses centrally managed constants for consistent semantics, and is potentially extensible thanks to the quarter-byte concept. Its strength lies in combining simple binary logic with device-specific control protocols.

Technical Features

State Overview

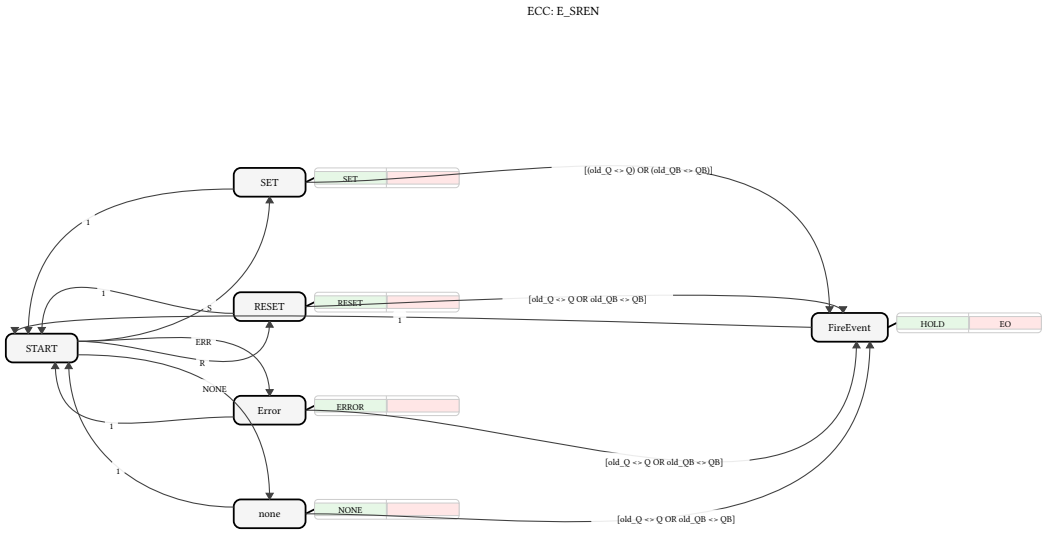
Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

🛠️ Zugehörige Übungen

Conclusion

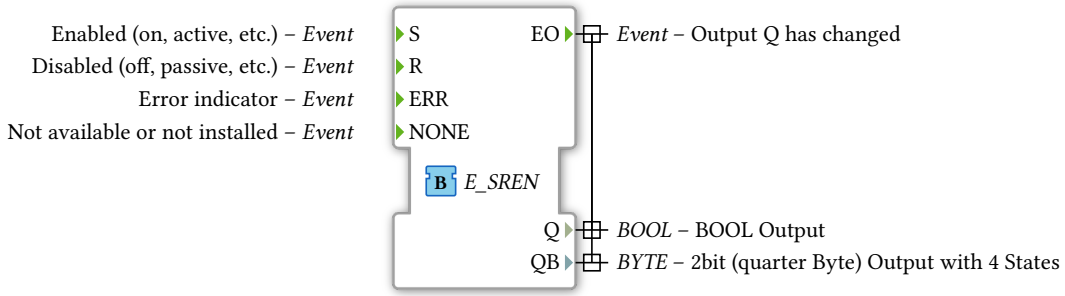
E_SREN



• – ○ • – ○ • – ○ •

Introduction

The E_SREN (Event-driven quad-state) is an event-driven function block that controls an output with four defined states. It reacts to various input events and sets its output accordingly to one of the four states: Enabled (`SET`), Disabled (`RESET`), Error (`ERROR`), or No Action (`NONE`). This function block is particularly suitable for applications where a signal must represent multiple operating states, such as in monitoring or control systems.



Interface Structure

Event Inputs

- **S (Set):** Sets the output `Q` to `TRUE` and `QB` to the state `COMMAND_ENABLE`.
- **R (Reset):** Sets the output `Q` to `FALSE` and `QB` to the state `COMMAND_DISABLE`.
- **ERR (Error):** Sets the output `Q` to `FALSE` and `QB` to the state `STATUS_ERROR`.
- **NONE:** Sets the output `QB` to the state `COMMAND_NO_ACTION`. The output `Q` remains unchanged.

Event Outputs

- **EO (Event Output):** Triggered when one of the output values (`Q` or `QB`) has changed. This event is always sent along with the current values of the data outputs `Q` and `QB`.

Data Inputs

- This function block has no data inputs.

Data Outputs

- **Q (BOOL):** Simple Boolean output. It is only `TRUE` when the `S` event occurs. It is `FALSE` for the `R` and `ERR` events and remains unchanged for `NONE`.
- **QB (BYTE):** A byte output specifically used to encode four states (2 bits). The specific values (e.g., `COMMAND_ENABLE`) are obtained from the imported library `quarter::const::quarter`.

Adapters

- This function block does not use any adapters.

Functionality

The E_SREN function block is implemented as a Basic function block and has an internal state machine (ECC). The output state is `START`. Upon receiving an input event (`S`, `R`, `ERR`, `NONE`), the automaton transitions to the corresponding state (`SET`, `RESET`, `Error`, `none`). In these states, the associated algorithm is executed, which calculates the values for `Q` and `QB`.

... A condition then checks whether the new value of `Q` or `QB` has changed compared to the last stored value (`old_Q`, `old_QB`). If so, the automaton transitions to state `FireEvent`. Here, the algorithm `HOLD` is executed, which stores the current output values in the internal variables and simultaneously triggers the output event `EO`. Afterward, the automaton always returns to state `START`. If the initial values have not changed, a direct transition occurs from states `SET`, `RESET`, `Error`, or `none` back to state `START`, without triggering `EO`.

Technical Features

- **State Detection:** The function block stores the previous initial state in the internal variables `old_Q` and `old_QB`. The output event `EO` is only generated when an actual state change occurs, thus preventing redundant event outputs.
- **State Detection:** The function block stores the previous initial state in the internal variables `old_Q` and `old_QB`. The output event `EO` is only generated when an actual state change occurs, preventing redundant event outputs.
- **Library Dependency:** The specific byte values for the `QB` output are imported from the constant library `logiBUS::utils::quarter::const::quarter`. Correct functionality requires the availability of this library.
- **Four-State Logic:** The logic of the `NONE` event is special: While `QB` is set to `COMMAND_NO_ACTION`, the Boolean output `Q` remains explicitly unchanged.

State Overview

The ECC (Execution Control Chart) consists of six states:

1. **START:** Initial and idle state.
2. **SET:** Activated by the `S` event. 3. **RESET:** Activated on event `Q=TRUE`, `QB=COMMAND_ENABLE`.
3. **Error:** Activated on event `R`. Sets `Q=FALSE`, `QB=COMMAND_DISABLE`.
4. **none:** Activated on event `ERR`. Sets `Q=FALSE`, `QB=STATUS_ERROR`.
5. **none:** Activated on event `NONE`. Sets `QB=COMMAND_NO_ACTION`. `Q` remains unchanged.
6. **FireEvent:** This event is only triggered if `Q` or `QB` has changed. It saves the new values and triggers `EO`.

Application Scenarios

- **Actuator Control:** Controlling a drive using the commands “On” (`S`), “Off” (`R`), “Fault” (`ERR`), and “Manually/Externally Controlled” (`NONE`).
- **Message Systems:** Displaying the status of a machine: “In Operation,” “Stopped,” “Error,” “Maintenance.”
- **Safety-Related Controllers:** Clear separation of normal operating (`S`/`R`), fault (`ERR`), and maintenance/override (`NONE`) states.

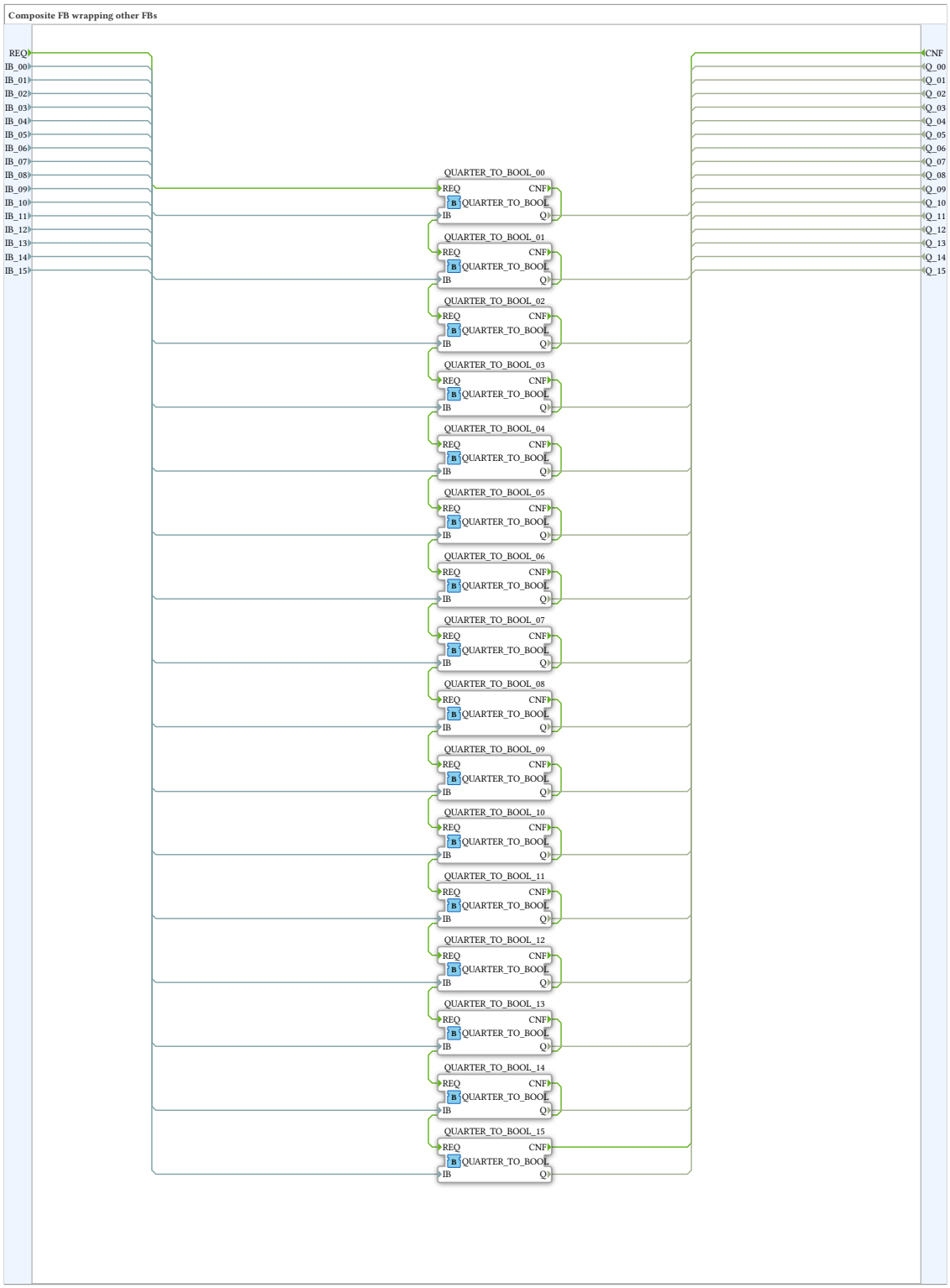
⚖️ Comparison with Similar Function Blocks

- **E_SR (Bistable Function):** The classic set-reset flip-flop only has two stable states (`TRUE`/`FALSE`). The E_SREN extends this concept to include two additional states (`ERROR`, `NO_ACTION`), which are encoded via a dedicated byte (`QB`).
- **E_D_FF (D-Flip-Flop):** Assumes a data value on a clock event. The E_SREN is event-driven (four different events) and has no separate data input. The “data” is implicitly contained in the triggering events.

Conclusion

The E_SREN is a specialized, event-driven function block for applications requiring more than two discrete states. By combining a simple Boolean signal (`Q`) with a multi-valued byte signal (`QB`) and intelligent, change-based event output (`EO`), it offers an efficient and clear solution for complex state control. Its strength lies in the clear semantics of the four input events and the reliable avoidance of redundant output events.

QUARTERS_TO_BOOLS



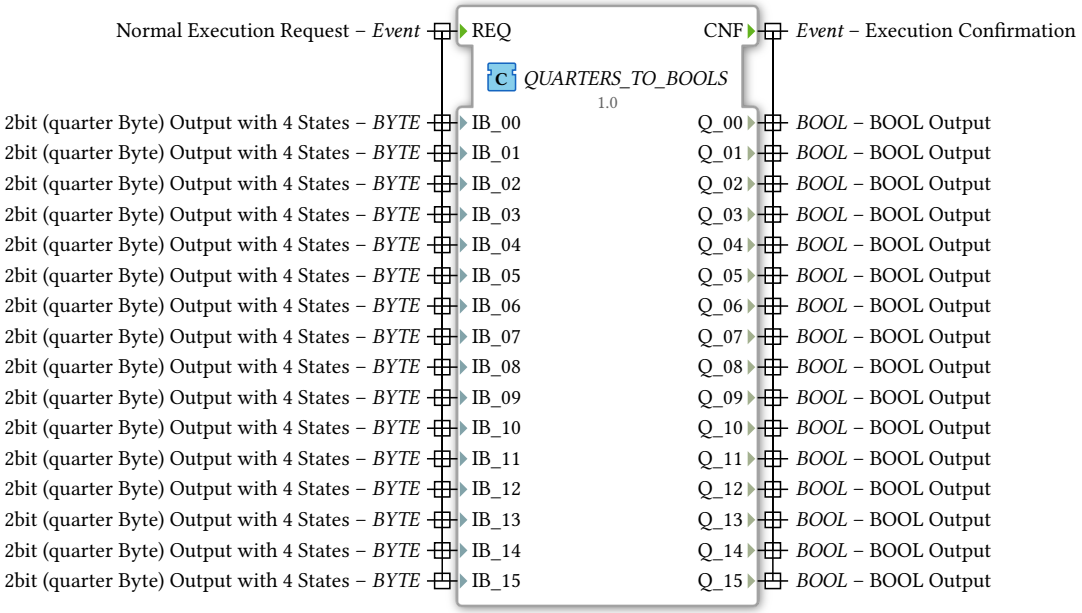
🎧 Podcast

- QUARTER

• – ○ • – ○ • – ○ •

Introduction

The function block `QUARTERS_TO_BOOLS` is a composite function block that converts 16 separate 2-bit input values (so-called “quarter bytes”) in parallel into corresponding Boolean output signals. It acts as a wrapper and simplifies handling by combining a multitude of individual conversion blocks into a single, easily manageable block. This block is particularly useful in control systems where compact data formats (such as 2-bit states in a byte) need to be converted into simple binary control signals for actuators or status indicators.



Interface Structure

Event Inputs

- **REQ**: Starts processing. Upon arrival of this event, all 16 input values (`IB_00` to `IB_15`) are read and processed.

Event Outputs

- **CNF**: Signals successful completion of processing. This event is output after all 16 internal conversions are complete and the output data (`Q_00` to `Q_15`) has been updated.

Data Inputs

- **IB_00** to **IB_15** (Type: `BYTE`): 16 inputs for 2-bit data (quarter bytes). Each input can represent one of four defined states (e.g., `quarter::COMMAND_DISABLE`). The default initial value for all inputs is `quarter::COMMAND_DISABLE`.

Data Outputs

- **Q_00** to **Q_15** (Type: `BOOL`): 16 Boolean outputs that reflect the converted state of the respective quarter-byte input. The initial value for all outputs is `FALSE`.

Adapters

This function block does not use any adapter interfaces.

Functionality

QUARTERS_TO_BOOLSis a composite function block that internally consists of 16 instances of the base function blockQUARTER_TO_BOOL. Each of these instances is responsible for converting a specific input (IB_xx`).

1. **Triggering**: The arrival of the event `REQ` triggers the processing chain.
2. **Parallel Data Distribution**: The byte values present at inputs `IB_00` to `IB_15` are forwarded in parallel to the corresponding inputs (`IB`) of the 16 internal `QUARTER_TO_BOOL` blocks.

3. **Serial Event Processing**: The REQ event is propagated serially through the chain of internal blocks. It starts at QUARTER_TO_BOOL_00 and is passed sequentially from one block to the next (CNF -> REQ) until the last block (QUARTER_TO_BOOL_15) completes its processing.
4. **Conversion**: Each internal QUARTER_TO_BOOL block interprets its byte input value according to a defined logic (presumably based on the two least significant bits) and converts it to a Boolean value (TRUE or FALSE).
5. **Result Output**: The Boolean results (Q) of the internal blocks are routed to the corresponding outputs Q_00 to Q_15 of the composite block.
6. **Completion**: Once the last internal block is complete, the composite block triggers its CNF event to signal the completion of the entire conversion cycle for all 16 channels.

Technical Features

- **Initialization**: All inputs are preset with the specific value quarter::COMMAND_DISABLE, and all outputs start with FALSE. This ensures a defined, inactive output state.
- **Processing Order**: While data inputs are distributed in parallel, event processing occurs strictly sequentially from index 00 to 15. This results in a deterministic, though not simultaneous, update of the outputs.
- **Composite Design**: The block encapsulates the complexity of 16 individual conversions and offers a clean, unified interface, which increases reusability and readability in higher-level applications.

State Overview

As a Composite Function Block, QUARTERS_TO_BOOLS does not have its own complex state machine in the conventional sense. Its behavior is defined by the network of internal blocks. In simplified terms, the overall state can be viewed as **Idle** (waiting for REQ) and **Processing** (event passing through the internal chain). The CNF output marks the transition back to the Idle state.

Application Scenarios

- **Compact PLC Connection**: Conversion of compact 32-bit data words (containing 16 2-bit states) into 16 individual binary control signals for valves, lamps, or relays.
- **Status Decoding**: Decoding of device status information transmitted in a “quarter byte” format into individual, easily processed error or operating state bits.
- **Simplifying Function Plans**: Replaces 16 separate QUARTER_TO_BOOL blocks and their wiring in a function plan with a single, clear block, simplifying project maintenance.

⚖️ Comparison with Similar Blocks

- **QUARTER_TO_BOOL**: This is the basic block that converts a single 2-bit input. QUARTERS_TO_BOOLS combines 16 instances of this block into a single unit. Using the composite block is more efficient for handling multiple channels, while the single block offers maximum flexibility in individual placement and wiring.
- **BYTE_TO_BOOL blocks**: Conventional blocks that split an entire byte into 8 individual bits. QUARTERS_TO_BOOLS is more specialized, as it assumes that each byte is already divided into four independent 2-bit units that must be interpreted separately.
-

🛠️ Related Exercises

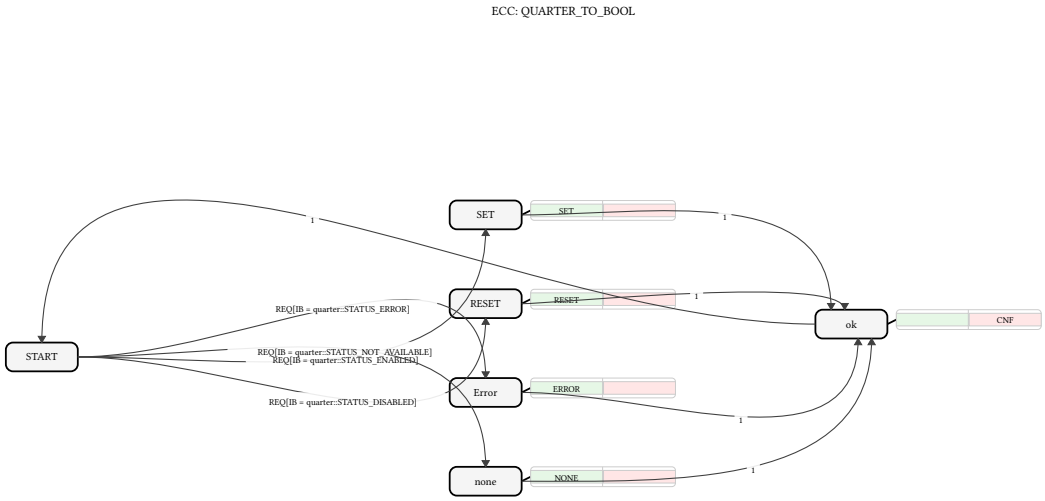
- Exercise_060

Conclusion

The QUARTERS_TO_BOOLS function block is a practical and time-saving tool for IEC 61499 programming when multiple 2-bit data channels frequently need to be converted into Boolean signals. By encapsulating 16 conversions in a single block, it significantly reduces wiring effort in higher-level applications and improves clarity. Its deterministic, serial event processing ensures reliable behavior that is well-suited for control tasks.

QUARTER_TO_BOOL

🎧 Podcast

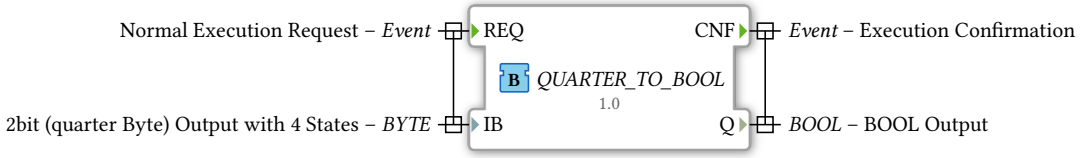


- QUARTER

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `QUARTER_TO_BOOL` converts a quad-state signal, encoded in the lower two bits of a byte value, into a simple BOOL signal. It is particularly useful for connecting to systems that provide status information with more than two states, which should be further processed using simple binary logic.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event evaluates the incoming input value `IB`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event is always output after processing a `REQ` event.

Data Inputs

- **IB** (BYTE): Contains the 4-state value to be converted in its lower two bits. The initial value is `quarter::COMMAND_DISABLE`. The expected values are specific constants from the `quarter` namespace.

Data Outputs

- **Q** (BOOL): The result of the conversion. The initial value is `FALSE`.

Adapters

This function block does not use adapters.

Functionality

When triggered by a `REQ` event, the function block reads the value at the data input `IB`. This value is then compared with predefined constants to determine the corresponding internal state. Depending on the state, a specific algorithm is executed, which either sets the BOOL output `Q` or leaves it unchanged. After the algorithm has finished executing, the `CNF` event is triggered, and the function block returns to its initial state.

The specific mapping of the input values to the output logic is as follows:

- `quarter::STATUS_ENABLED` ⟶ Algorithm `SET` ⟶ `Q := TRUE`
- `quarter::STATUS_DISABLED` ⟶ Algorithm `RESET` ⟶ `Q := FALSE`
- `quarter::STATUS_ERROR` ⟶ Algorithm `ERROR` ⟶ `Q := FALSE`
- `quarter::STATUS_NOT_AVAILABLE` ⟶ Algorithm `NONE` ⟶ `Q` remains unchanged

Technical Features

- **State Handling**: The function block is implemented as a Basic Function Block and It has an explicit state machine (ECC). The states `SET`, `RESET`, `ERROR`, and `none` are pure algorithm states, while the state `ok` is responsible for outputting the acknowledgment event.
- **Value Preservation**: In the case of state `STATUS_NOT_AVAILABLE`, the algorithm `NONE` is executed, which explicitly does not modify the current value of the output `Q`. This enables "don't care" or "hold" behavior.
- – **Initialization**: The data output `Q` is initialized to `FALSE` at startup.

State Overview

The ECC (Execution Control Chart) consists of six states:

1. **START**: Initial and wait state. Upon receiving `REQ`, a transition to `SET`, `RESET`, `Error`, or `none` occurs, depending on the value of `IB`.
2. **SET**: Executes the algorithm `SET` (sets `Q` to TRUE).
3. **RESET**: Executes the algorithm `RESET` (sets `Q` to FALSE).
4. **Error**: Executes the algorithm `ERROR` (sets `Q` to FALSE).
5. **none**: Executes the algorithm `NONE` (leaves `Q` unchanged).
6. **OK**: Triggers the `CNF` output event and then switches back to the `START` state.

Application Scenarios

- **Connecting Field Devices**: Many sensors or actuators report statuses such as "Ready," "Faulty," "Maintenance," or "Not Connected." This function block can convert such messages into a simple "On/Off" or "OK/Not OK" signal for higher-level control logic.
- **Simplifying Logic**: In controllers that only require binary decisions (e.g., "Release Process" yes/no), this function block can reduce more complex status messages to the essential binary information.
- **Error Handling**: Consistent handling of error states (`STATUS_ERROR`) by setting the output to `FALSE`.

⚖️ Comparison with Similar Building Blocks

- **Standard Converters (e.g., `BYTE_TO_BOOL`)**: A simple `BYTE_TO_BOOL` converter would typically use a threshold (e.g., anything >0 becomes TRUE). `QUARTER_TO_BOOL`, on the other hand, interprets specific, named states and provides defined behavior for each one, including the option to leave the output unchanged for a given state.
- **`E_SELECT` or `E_DEMUX` Building Blocks**: These could be used to activate different event paths based on an input value. `QUARTER_TO_BOOL` encapsulates this logic specifically for converting 4-state signals and directly returns the Boolean result.
-

🛠️ Related Exercises

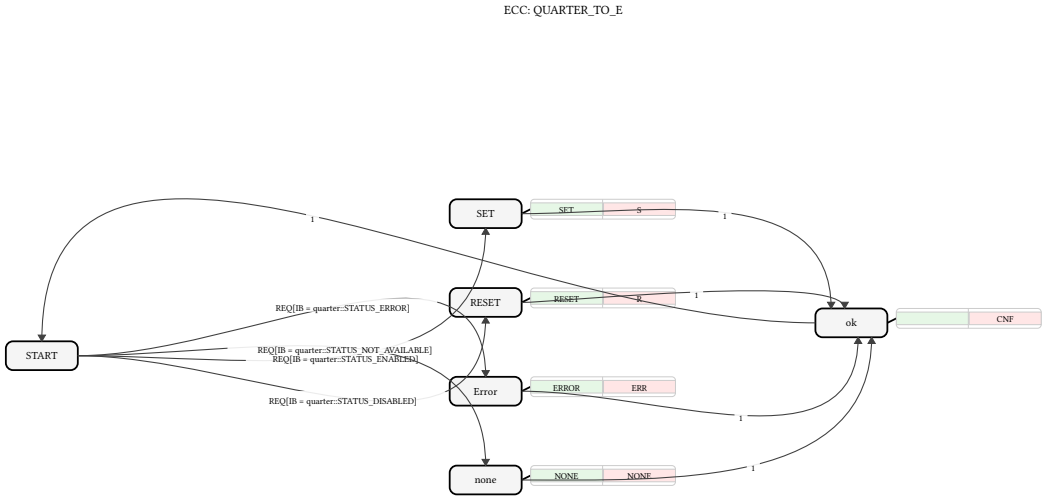
- Exercise_055
- Exercise_056

Conclusion

The `QUARTER_TO_BOOL` function block is a specialized and useful converter for applications where compact status information with four discrete states needs to be integrated into simple binary logic. Its clear definition of the behavior for each state, especially the retention of the output value in the "Not Available" case, makes it robust and highly predictable. It is ideally suited for the interface between more complex fieldbus systems and basic binary control logic.

QUARTER_TO_E

🎧 Podcast

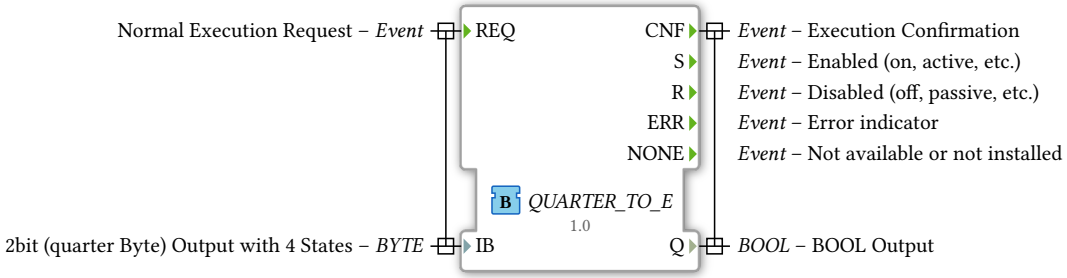


- QUARTER

• – ○ • – ○ • – ○ •

Introduction

The function block `QUARTER_TO_E` is used to translate a 2-bit state value (a so-called "quarter byte") into four different, unique events. It is an auxiliary block frequently used in conjunction with blocks that report more than the usual two states (ON/OFF), such as status messages from devices that can also indicate error or unavailability states.



Interface Structure

Event Inputs

- **REQ**: Starts processing. This event evaluates the current value at data input `IB`.

Event Outputs

- **CNF**: Signals successful completion of processing, regardless of the detected state. It is always generated after the output of one of the specific events.
- **S** (Enabled): Triggered when the "enabled" or "switched on" state is detected.
- **R** (Disabled): Triggered when the "disabled" or "switched off" state is detected.
- **ERR** (Error): Triggered when an error state is detected.
- **NONE** (Not available): Triggered when the "not available" or "not installed" state is detected.

Data Inputs

- **IB** (BYTE): Contains the 2-bit state value to be interpreted. The initial value is set to `quarter::COMMAND_DISABLE`. The actual interpretation is performed using the defined constants from the `quarter` package.

Data Outputs

- **Q** (BOOL): A Boolean output whose value is set depending on the detected state. It serves as a simple, binary representation of the main state (enabled/disabled).

Adapters

This function block does not use adapters.

Functionality

The `QUARTER_TO_E` is a Basic Function Block with an internal state machine (ECC). Upon the arrival of the `REQ` event, the value at input `IB` is compared with predefined constants. Depending on the comparison, the automaton jumps to one of four states: `SET`, `RESET`, `Error`, or `none`. In each of these states, a corresponding algorithm is executed, which sets (or leaves unchanged) the Boolean output `Q` and triggers the associated specific event (`S`, `R`, `ERR`, `NONE`). The automaton then transitions to state `ok`, from where the confirmation event `CNF` is issued, before returning to the initial state `START` and waiting for the next `REQ`.

Technical Details

- This block uses constants from the package `logiBUS::utils::quarter::const::quarter` (`STATUS_ENABLED`, `STATUS_DISABLED`, `STATUS_ERROR`, `STATUS_NOT_AVAILABLE`). These must be available in the project.
- The output `Q` is only modified in the states `SET` (TRUE), `RESET` (FALSE), and `ERROR` (FALSE). In the state `NONE`, `Q` is explicitly not modified ("Don't care").
- The output of the `CNF` event always occurs, regardless of which specific state was detected. This enables consistent process control within the application.

State Overview

1. **START**: Waiting state. At `REQ`, `IB` is evaluated, and the process transitions to a processing state.
2. **SET**: Reached at `IB = STATUS_ENABLED`. Sets `Q:=TRUE` and triggers `S`.
3. **RESET**: Reached at `IB = STATUS_DISABLED`. Sets `Q:=FALSE` and triggers `R`.
4. **Error**: Reached at `IB = STATUS_ERROR`. Sets `Q:=FALSE` and triggers `ERR`.
5. **none**: Reached at `IB = STATUS_NOT_AVAILABLE`. Triggers `NONE` (Q remains unchanged). 6. **ok**: Intermediate state that triggers the `CNF` event and returns to the `START` state.

Application Scenarios

- **Interpretation of Device Status**: A drive reports its status not only as ON/OFF, but also as "Error" or "Not Ready". This block converts the device status into separate, easily processed events.
- **Simplification of Logic**: Instead of constantly having to query the byte value in subsequent blocks, you can react directly to the specific events (`S`, `R`, `ERR`).
- **Abstraction**: Serves as an adapter between blocks that provide a quarter-byte status and logic that works with classic event/boolean signals.

⚖️ Comparison with similar blocks

- **E_DEMUX / E_SELECT**: These blocks forward an input event based on a control value to one of several outputs. `QUARTER_TO_E` is more specialized: It not only translates a specific data value (`IB`) into a selection, but also triggers different *content-related* events and additionally calculates a boolean value (`Q`).

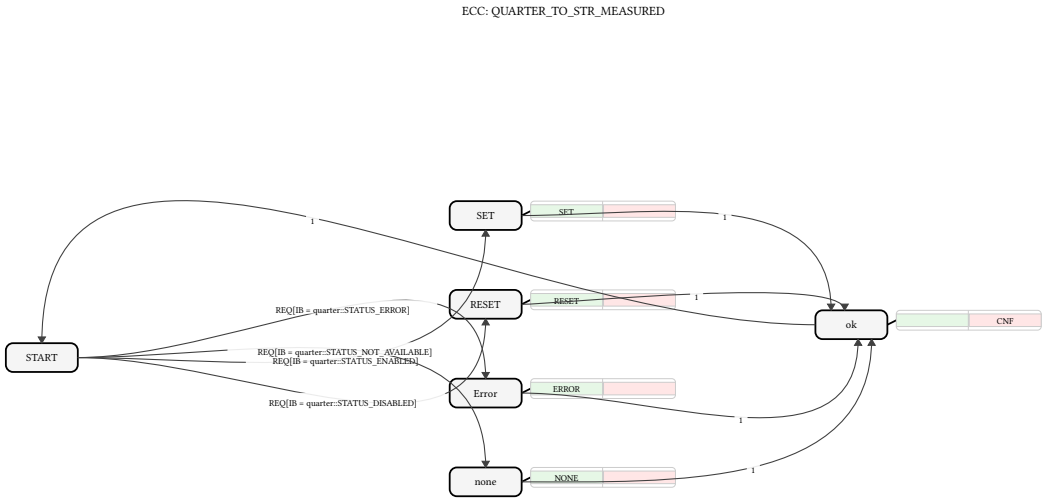
- **BYTE_TO_E**: A generic block that could convert each bit of a byte into a separate event. `QUARTER_TO_E` is semantically richer because it recognizes and outputs specific, predefined states (Enabled, Disabled, Error, None) and their meanings.

Conclusion

The `QUARTER_TO_E` is a useful and specialized component for the structured processing of four-state signals. It relieves the application logic by handling raw data interpretation and providing clear, semantic events as well as a simplified Boolean status. Its strength lies in the combination of data processing and event-based output according to IEC 61499.

QUARTER_TO_STR_MEASURED

🎧 Podcast

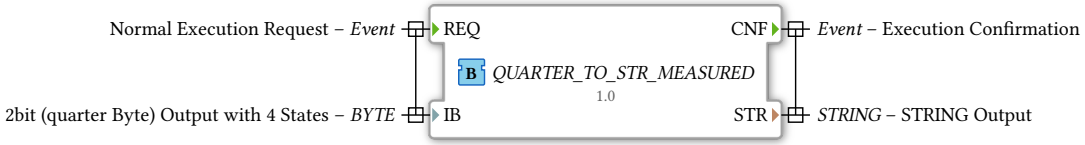


- QUARTER

• – ○ • – ○ • – ○ •

Introduction

The function block `QUARTER_TO_STR_MEASURED` converts a 4-state signal (encoded in the lower two bits of a BYTE value) into a human-readable text string (STRING). It is particularly suitable for displaying or logging status information in control systems, where discrete states such as “On,” “Off,” “Error,” or “Not Available” need to be converted into textual form.



Interface Structure

Event Inputs

- **REQ**: Starts the normal execution of the function block. Upon this event, the value at data input `IB` is evaluated, and the corresponding conversion is performed.

Event Outputs

- **CNF**: Signals the completion of the conversion and the availability of the result at data output `STR`.

Data Inputs

- **IB** (BYTE): The input for the quad-state value to be converted. Only the lower two bits (LSB) are evaluated. The block expects specific, predefined constant values (e.g., `quarter::STATUS_ENABLED`). The initial value is `quarter::STATUS_DISABLED`.

Data Outputs

- **STR** (STRING): The output where the converted text string is provided. The initial value is the text constant for the "Disabled" status (`quarter::STATUS_DISABLED_msg`).

Adapters

This function block has no adapter interfaces.

Operation

The `QUARTER_TO_STR_MEASURED` is a Basic Function Block (BFB) with an internal Execution Control Graph (ECC). Upon the arrival of the `REQ` event, the value at the input `IB` is compared with predefined constants from the library `logiBUS::utils::quarter::const::quarter`. Depending on the matching value, the controller branches to one of four states (`SET`, `RESET`, `Error`, `none`). In each of these states, a specific algorithm is executed, assigning the corresponding text constant (e.g., `quarter::STATUS_ENABLED_msg`) to the output `STR`. The block then transitions to state `ok`, from which the output event `CNF` is triggered, before the block returns to its initial state `START` and waits for the next `REQ`.

...

Technical Features

- **Typed Constants:** The block uses strongly typed constants for both the input values (`quarter::STATUS_...`) and the output strings (`quarter::STATUS_..._msg`). This increases the type safety and maintainability of the code.
- **Basic FB:** The implementation as a Basic FB with ECC enables clear and comprehensible state logic.
- **Initial Values:** Both the data input `IB` and the data output `STR` are pre-populated with meaningful initial values (disabled state).

State Overview

The ECC of the function block consists of six states:

1. **START:** The initial wait state. At `REQ`, a branch is triggered based on the value of `IB`.
2. **SET:** Activated at `IB = quarter::STATUS_ENABLED`. Executes the algorithm `SET`.
3. **RESET:** Activated at `IB = quarter::STATUS_DISABLED`. Executes the algorithm `RESET`.
4. **Error:** Activated at `IB = quarter::STATUS_ERROR`. Executes the algorithm `ERROR`.
5. **none:** Activated at `IB = quarter::STATUS_NOT_AVAILABLE`. Executes the algorithm `NONE`.
6. **ok:** Common state that is traversed after each successful conversion. Triggers the `CNF` event and returns to the `START` state.

Application Scenarios

- **HMI/SCADA Integration:** Conversion of internal control status values into readable text for display on operator panels or in visualization systems.
- **Logging and Diagnostics:** Generation of plain-text log entries for error or status messages, which are easier to analyze than numeric codes.
- **Interface to Higher-Level Systems:** Preparation of status information for transmission to MES or ERP systems that expect string data.

⚖️ Comparison with Similar Building Blocks

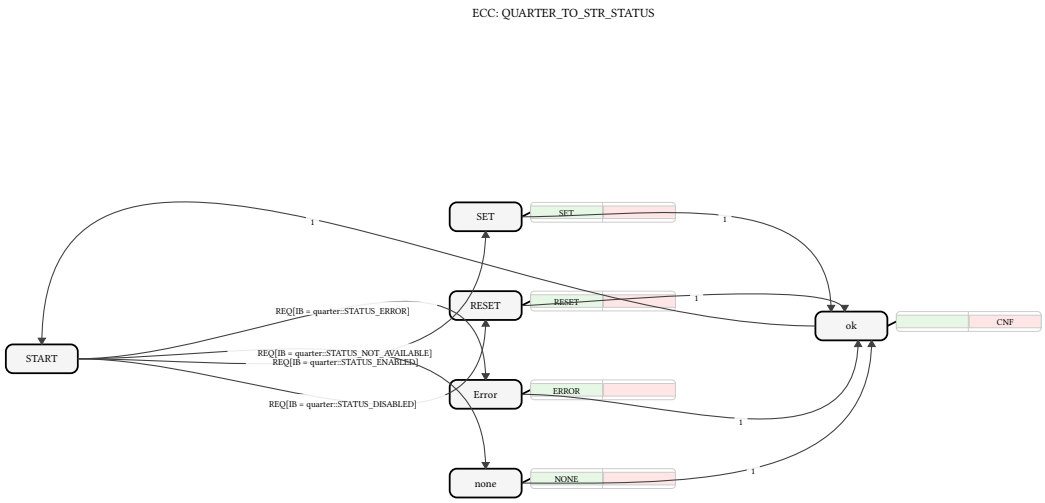
- **`BYTE_TO_STRING`:** A general-purpose converter that transforms any byte value into its decimal string representation. `QUARTER_TO_STR_MEASURED` is more specialized and converts specific, semantically meaningful values into predefined, meaningful texts, not numeric strings.
- **`E_SR` or `E_RS` (Flip-Flops):** These blocks represent binary states (SET/RESET) using Boolean signals. `QUARTER_TO_STR_MEASURED` extends this concept to four states and additionally offers textual representation.

Conclusion

The `QUARTER_TO_STR_MEASURED` is a specialized and robust function block for the semantic conversion of 4-state signals. By using library constants for inputs and outputs, it ensures high code quality and simplifies maintenance. It is the ideal choice when discrete status information from the control plane needs to be converted into textual form for display, logging, or interface purposes.

QUARTER_TO_STR_STATUS

🎧 Podcast

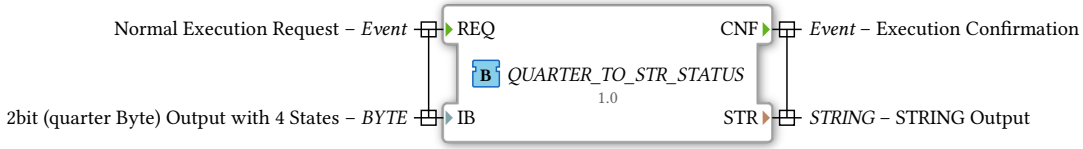


- QUARTER

• – ○ • – ○ • – ○ •

Introduction

The function block `QUARTER_TO_STR_STATUS` converts a 2-bit status value (also known as a “quarter byte”) into a human-readable text string. It is part of the `logiBUS::utils::quarter` library and is typically used to convert compact status information from controllers or devices into an understandable text format for display, logging, or diagnostic purposes.



Interface Structure

Event Inputs

- **REQ**: Starts processing. Upon arrival of this event, the value at data input `IB` is read and the corresponding conversion is performed.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event, along with the converted string, is output at data output `STR`.

Data Inputs

- **IB** (BYTE): The input for the 2-bit status value. Only the least significant two bits (bits 0 and 1) are evaluated. This block expects specific, predefined constants from the `quarter` library. The initial value is `quarter::COMMAND_DISABLE`.

Data Outputs

- **STR** (STRING): The output that provides the text string corresponding to the status value. The initial value is `quarter::COMMAND_DISABLE_msg`.

Adapters

This function block has no adapter interfaces.

Functionality

The `QUARTER_TO_STR_STATUS` is a Basic Function Block (BFB) with a defined Execution Control Chart (ECC). Upon an incoming `REQ` event, the value at `IB` is compared to predefined constants. Depending on this comparison, the controller branches to one of four possible states (`SET`, `RESET`, `Error`, `none`). In each of these states, a specific algorithm is executed that sets the initial `STR` to a corresponding text string. After the algorithm has finished executing, the block transitions to state `ok`, from which the `CNF` event is triggered. The block then returns to the initial `START` state and is ready for the next request.

In each of these states, a specific algorithm is executed that sets the initial `STR` to a corresponding text string. The specific string values are loaded from the `quarter` constant library (`logiBUS::utils::quarter::const::quarter`). The expected input values and their corresponding output strings are:

- `quarter::STATUS_ENABLED` ⟶ `quarter::COMMAND_ENABLE_msg`
- `quarter::STATUS_DISABLED` ⟶ `quarter::COMMAND_DISABLE_msg`
- `quarter::STATUS_ERROR` ⟶ `quarter::COMMAND_RESERVED_msg`
- `quarter::STATUS_NOT_AVAILABLE` ⟶ `quarter::COMMAND_NO_ACTION_msg`

Technical Features

- **Type Safety:** The block uses strongly typed constants from a dedicated library, which reduces the susceptibility to errors compared to the direct use of raw values (e.g., 0, 1, 2, 3).
- **2-Bit Processing:** Although the input is declared as `BYTE`, only one quarter (2 bits) of this byte is effectively used. The semantics of the four possible states are defined by the constant library used.

Deterministic Behavior: The state transitions depend solely on the input value at `REQ`. There is no internal memory or hysteresis effects.

State Overview

The ECC consists of six states:

1. **START:** Initial, waiting state. It waits for the `REQ` event.
2. **SET:** Activated at `IB = quarter::STATUS_ENABLED`. Executes algorithm `SET`.
3. **RESET:** Activated at `IB = quarter::STATUS_DISABLED`. Executes algorithm `RESET`.
4. **Error:** Activated at `IB = quarter::STATUS_ERROR`. Executes algorithm `ERROR`.
5. **none:** Activated at `IB = quarter::STATUS_NOT_AVAILABLE`. Executes algorithm `NONE`.
6. **ok:** Common state after every successful conversion. Triggers the `CNF` exit event and returns to the `START` state.

Application Scenarios

- **HMI/SCADA Integration:** Conversion of internal device states (e.g., “enabled,” “faulty”) into strings for display on operator panels or in visualization software.
- **Logging and Diagnostics:** Conversion of status codes into readable text for log files or diagnostic tools to facilitate fault analysis.
- **Interface to Text-Based Systems:** Preparation of status information for further processing in systems that work with string messages (e.g., MQTT topics, CSV export).

⚖️ Comparison with Similar Function Blocks

- **`E_SR` or `E_RS` (Flip-Flops):** These blocks store a binary state (SET/RESET). `QUARTER_TO_STR_STATUS`, on the other hand, only converts an existing 4-state value into a string; it has no dedicated memory.

- **`E_SELECT` or `E_MUX`:** These can also choose between different paths/values, but are more generic and not specifically designed for converting to strings with predefined quarter-state values.
- **Simple `STRING` assignment:** A direct assignment in ST code could achieve something similar, but the `QUARTER_TO_STR_STATUS` block encapsulates the logic, promotes reusability, and enforces the use of standardized constants, thus increasing consistency across the entire project.

`STRING` assignment: A direct assignment in ST code could accomplish something similar, but the `QUARTER_TO_STR_STATUS` block encapsulates the logic, promotes reusability, and enforces the use of standardized constants, increasing consistency throughout the project.

🛠️ Related Exercises

- Exercise_055
- Exercise_056

Conclusion

The `QUARTER_TO_STR_STATUS` is a specialized and useful function block for applications that work with the specific 4-state status model (Quarter Byte). By using a constant library and clearly separating logic and interface, it contributes to the robustness and maintainability of IEC 61499 applications. It is the ideal choice when compact status information needs to be reliably and consistently translated into a human-readable format.

const

quarter

🎧 Podcast



- The Secret of the 2-Bit Quarter: Efficiency in CAN Bus for Commercial Vehicles
- QUARTER

• – ◦ • – ◦ • – ◦

The `quarter` library is a collection of global constants designed for working with SAE J1939-compliant status and command values in logiBUS applications. It defines standardized 2-bit values (represented as `BYTE`) and associated plaintext descriptions (`STRING`) for states and control commands. These constants serve to standardize and improve the readability of code that works with binary status information and control commands.

quarter `is a collection of global constants` (GLOBALCONSTANTS`). Therefore, it is not a function block (FB) in the traditional sense with event or data interfaces. The interface consists solely of the defined constants, which can be referenced directly in the code.

Not present (constant collection).

Not present (constant collection). The constants themselves represent symbolic names for fixed values.

Not present (constant collection).

Event Outputs

Event Inputs

Data Outputs

Data Inputs

Interface Structure

Introduction

Adapters

Not present (constant collection).

Functionality

The library provides only named constants. When used in code, the symbolic name (e.g., `STATUS_ENABLED`) is replaced by the assigned value (e.g., `2#01`). The `_msg` variants contain descriptive text that can be used for debugging, logging, or user output.

- **Data Types:** All status and command constants are of type `BYTE`, even though they only use the lower two bits (values 0-3). The corresponding text constants are of type `STRING`.
- **Values:** The values are defined as binary literals (`2#00`, `2#01`, etc.), which clearly indicates their interpretation as 2-bit fields.
- **Package:** The constants are organized in the package `logiBUS::utils::quarter::const`.

Not applicable, as this is a static collection of values.

- **SAE J1939 Communication:** Used in function blocks that send or receive parameters according to SAE J1939, especially for status and control information transmitted as 2-bit fields.
- **Status Management:** Assigning and querying device or function states (e.g., on/off, faulty, unavailable) within a control application.
- **Logging and Diagnostics:** Use of the `_msg` strings to generate readable log entries or displays in the control panel.
- **`logiBUS::utils::bit::const`:** Contains constants for single bits (e.g., `BIT_FALSE`, `BIT_TRUE`). `quarter` extends this concept to 2-bit fields with application-specific semantics.
- **Manual Definition:** Instead of using this library, the values could be written directly in the code as numbers or binary literals. However, using `quarter` increases maintainability, readability, and consistency across the entire project.

SAE J1939-Specific Libraries: `quarter` is a basic utility library. More comprehensive J1939 libraries would likely rely on these constants to define more specific messages (PGNs) or parameters (SPNs).

The `quarter` constant library is a useful tool for developing control software in the SAE J1939 environment and other systems that use 2-bit status and control fields. It promotes standardized, self-explanatory programming and reduces the potential for errors through the use of "magic numbers." Its integration into the `logiBUS::utils` package makes it a fundamental component for structured logiBUS applications.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

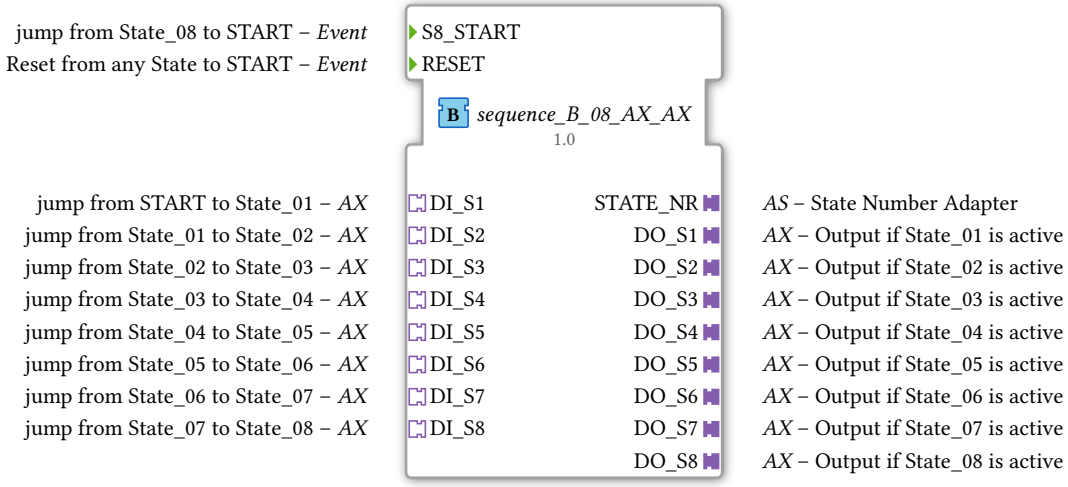
Conclusion

sequence

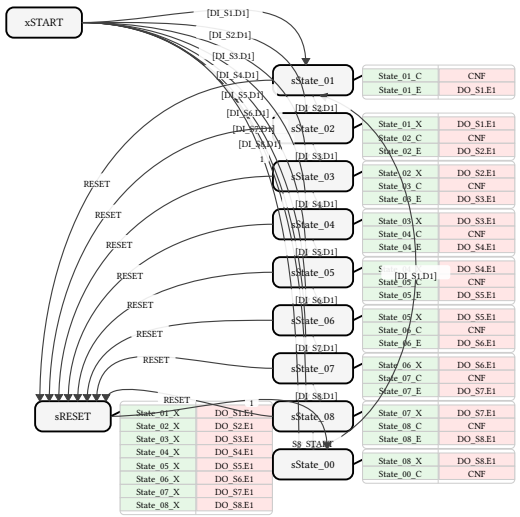
Numeric Index (sequence)

Here you will find all exercises from the `sequence` workspace, listed in ascending order. ...

sequence_B_08_AX_AX



ECC: sequence_B_08_AX_AX



• – ○ • – ○ • – ○ •

Introduction

The function block **sequence_B_08_AX_AX** implements a sequential control system with eight outputs. State transitions are level-controlled via BOOL signals provided by an AX adapter. The block is designed for fail-safe applications and allows for the restoration of the current state after a power failure. It is particularly suitable for sequence control systems in automation technology where multiple switching operations must be executed sequentially.

Interface Structure

Event Inputs

Name	Description
S8_START	Event that triggers a jump from state 8 back to the start state.

RESET	Event that triggers an immediate reset from any active state.

Event Outputs

Name	Description
CNF	Acknowledge event that outputs the current state number after each state change. (With STATE_NR)

Data Inputs

The FB has no direct data inputs. The transition conditions are read exclusively via the socket adapters.

Data Outputs

Name	Type	Description
STATE_NR	SINT	Current state number: 0 = Start, 1…8 = State_01…State_08, 9 = State_00 (End).

Adapter

Plugs (Outputs – Type `adapter::types::unidirectional::AX`)

Name	Description
DO_S1	Output active when state 1 is active.
DO_S2	Output active when state 2 is active.
DO_S3	Output active when state 3 is active.
DO_S4	Output active when state 4 is active.
DO_S5`	Output active when state 5 is active.
DO_S6`	Output active when state 6 is active.
DO_S7`	Output active when state 7 is active.
DO_S8`	Output active when state 8 is active.

Sockets (Inputs – Type `adapter::types::unidirectional::AX`)

Name	Description
DI_S1`	Signal for transition from start state to state 1.
DI_S2`	Signal for transition from state 1 to state 2.
DI_S3	Signal for transition from state 2 to state 3.
DI_S4`	Signal for transition from state 3 to state 4.
DI_S5`	Signal for transition from state 4 to state 5.
DI_S6`	Signal for transition from state 5 to state 6.
DI_S7`	Signal for transition from state 6 to state 7.
DI_S8`	Signal for transition from state 7 to state 8.

Functionality

The function block operates on the basis of an event-driven state machine (ECM). At startup, it is in state xSTART. From there, the input signals DI_S1 to DI_S8 (each corresponding to the adapter's attribute .D1) are checked for their BOOL value. If a signal matches (TRUE), the machine transitions to the corresponding state (sState_01 … sState_08). If no signal is present, the state sState_00 is reached (end of sequence).

Each state performs the following actions upon entry:

1. **Deactivate the previous output**: The output adapter of the previous state is set to FALSE (algorithm State_n_X).
2. **Update the state number**: STATE_NR is set to the constant of the current state (e.g., sequence::State_01).
3. **Activate the new output**: The value of the corresponding input (DI_Sn.D1) is transferred to the output (DO_Sn.D1) (algorithm State_n_E).
4. **Output the confirmation event**: CNF is triggered.

From state 8, the event S8_START returns to the start state (xSTART). The event RESET can force an immediate return from any active state (1…8) to state sState_00 – setting all outputs to FALSE.

After a reset (RESET), the machine briefly enters state sRESET, which deactivates all outputs, and then immediately jumps to state sState_00.

Technical Features

- **Level-Controlled Transitions**: The transitions are triggered by the applied BOOL value of the respective input (not edge-triggered). This allows for easy recovery after a power failure, as the state can be directly detected and resumed upon power-up.
- **Use of AX Adapters**: The interfaces are implemented as unidirectional AX adapters, allowing for loose coupling and easy reuse of the inputs/outputs.
- **Configurable Constants**: The state numbers are derived from the constant sequence::State_nn in the package logiBUS::utils::sequence::const::sequence – this allows them to be defined and modified centrally.
- **Type Safety**: All outputs are explicitly set to FALSE when a state is exited and receive the value of the corresponding input when entering a new state. This prevents logic errors caused by hanging values.

State Overview

State (ECC)	State Number	Output Active	Transition Condition (to the Next State)
xSTART	0	–	DI_S1.D1 ⟶ sState_01DI_S2.D1 ⟶ sState_02… DI_S8.D1 ⟶ sState_08 otherwise ⟶ sState_00
sState_01	1	DO_S1	DI_S2.D1 ⟶ sState_02RESET ⟶ sState_00
sState_02	2	DO_S2	DI_S3.D1 ⟶ sState_03RESET ⟶ sState_00
sState_03	3	DO_S3	DI_S4.D1 ⟶ sState_04RESET ⟶ sState_00
sState_04	4	DO_S4	DI_S5.D1 ⟶ sState_05RESET ⟶ sState_00
sState_05	5	DO_S5	DI_S6.D1 ⟶ sState_06RESET ⟶ sState_00
sState_06	6	DO_S6	DI_S7.D1 ⟶ sState_07RESET ⟶ sState_00
sState_07	7	DO_S7	DI_S8.D1 ⟶ sState_08RESET ⟶ sState_00
sState_08	8	DO_S8	S8_START ⟶ xSTARTRESET ⟶ sState_00
sState_00	9	–	DI_S1.D1 ⟶ sState_01 (Sequence restart)

Note: The state sRESETThis function is used solely for clearing all outputs and exits automatically.

Application Scenarios

- **Sequence Control in Agricultural Machinery**: For example, the sequential switching on and off of eight hydraulic valves or lighting units.
- **Conveyor Belt Control**: Activation of belt sections sequentially, with transitions triggered by level sensors (BOOL).
- **Laboratory Automation**: Stepwise release of reagent dosing units, controlled by level signals from flow sensors.
- **End-of-Life Processes**: Since the state can be restored after a power failure using the input signals, this function block is suitable for safety-critical processes.

Comparison with Similar Function Blocks

- **Timer-Based Sequencers**: Transitions are triggered by time sequences rather than levels. In contrast, sequence_B_08_AX_AX reacts directly to external signals and is suitable for event-driven processes.

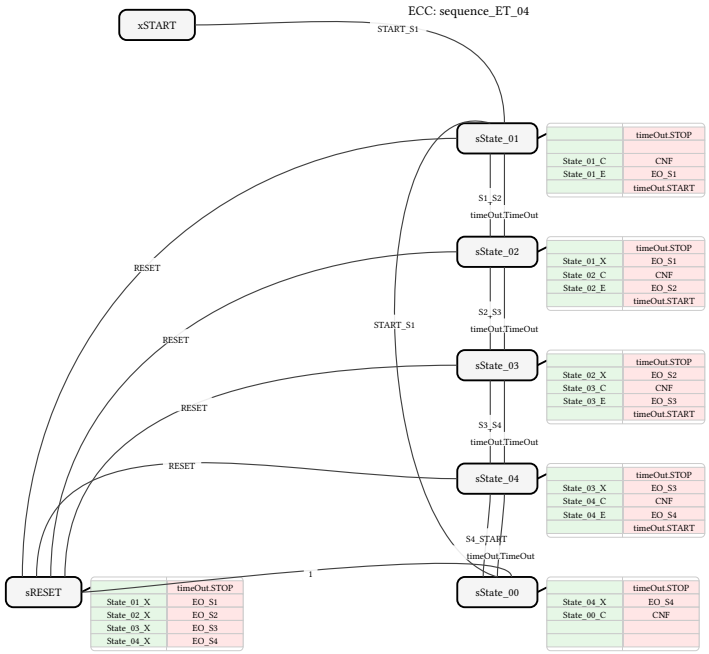
- **Edge-triggered sequencers**: These would only react to rising edges. In contrast, the function block presented here recognizes sustained levels, which improves its restart capability.
- **Adapterless blocks**: Many standard function blocks use simple BOOL inputs. The use of AX adapters achieves greater modularity and interchangeability of signal sources/sinks.

Conclusion

The `sequence_B_08_AX_AX` function block offers a robust and flexible way to implement an eight-stage sequence control with level-controlled transitions. The use of unidirectional AX adapters simplifies integration into complex automation systems, while explicit state management and reset behavior ensure high operational reliability. Thanks to its ability to recover after power outages, it is particularly suitable for applications in agricultural technology and industrial manufacturing.

combi

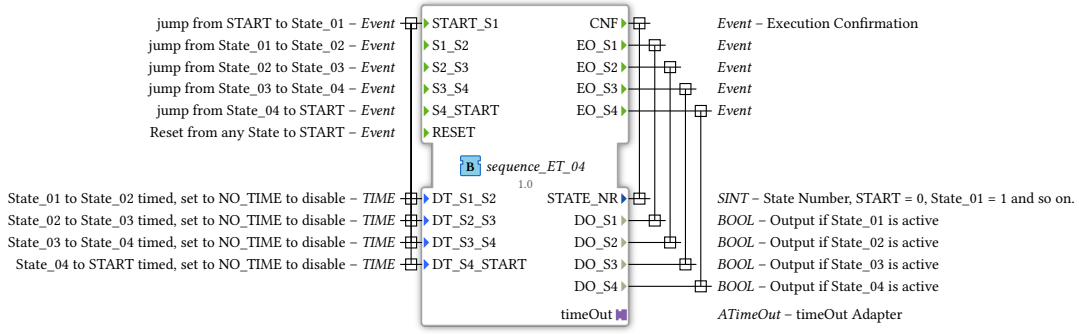
sequence_ET_04



• – ○ • – ○ • – ○ •

Introduction

The function block `sequence_ET_04` is a sequencer with four output states. It enables the control of a cyclic sequence of four steps (State_01 to State_04), where the transition between states can occur either through an external event or after an adjustable time. This function block is designed for applications where process steps must be executed sequentially with time-based or event-driven flexibility.



Interface Structure

Event Inputs

- **START_S1**: Transitions from START or State_00 to the first state, State_01. Transmits the time parameters for all state transitions (DT_S1_S2, DT_S2_S3, DT_S3_S4, DT_S4_START).
- **S1_S2**: Manual transition from State_01 to State_02.

S1_S2: Manual transition from State_01 to State_02. * **S2_S3**: Manual transition from State_02 to State_03.

- **S3_S4**: Manual transition from State_03 to State_04.
- **S4_START**: Manual transition from State_04 back to the START state (via State_00).
- **RESET**: Instantly resets the block from any state to the START state.

Event Outputs

- **CNF**: Execution confirmation. Triggered on every state change and returns the new state number (STATE_NR).
- **EO_S1**: Triggered upon entering State_01 and returns the output value DO_S1.
- **EO_S2**: Triggered upon entering State_02 and returns the output value DO_S2.
- **EO_S3**: Triggered upon entering State_03 and returns the output value DO_S3. * **EO_S4**: Triggered upon entering State_04 and returns the output value DO_S4.

Data Inputs

- **DT_S1_S2** (Type: TIME): Time for the automatic transition from State_01 to State_02. The value NO_TIME disables the time transition for this step.
- **DT_S2_S3** (Type: TIME): Time for the automatic transition from State_02 to State_03. The value NO_TIME disables the time transition for this step.
- **DT_S3_S4** (Type: TIME): Time for the automatic transition from State_03 to State_04. The value NO_TIME disables the time transition for this step.
- **DT_S4_START** (Type: TIME): Time for the automatic transition from State_04 back to the START state. The value NO_TIME disables the time transition for this step.

Data Outputs

- **STATE_NR** (Type: SINT): Current state number according to the constant sequence::State_XX (START = 0, State_01 = 1, State_02 = 2, State_03 = 3, State_04 = 4).

Data Outputs * DO_S1 (Type: BOOL): Is TRUE when State_01 is active

- **DO_S2** (Type: BOOL): Is TRUE when State_02 is active.
- **DO_S3** (Type: BOOL): Is TRUE when State_03 is active.
- **DO_S4** (Type: BOOL): Is TRUE when State_04 is active.

Adapter

- **timeOut** (Type: ATimeOut): A connector (Plug) for a timeout adapter. This adapter is used internally to implement timed state transitions.

Functionality

The module implements a finite state automaton (ECC) with the states START, State_01 to State_04, State_00, and RESET. The cycle begins in state START. An event START_S1 starts the sequence and transitions to State_01.

In each active state (State_01 to State_04), the following actions are executed:

1. The internal timer (timeOut) is stopped.
2. The output of the previous state is deactivated (Exit algorithm State_XX_X).
3. Confirmation CNF with the new state number is sent, and the time for the next possible automatic transition is passed to the timer (Confirmation algorithm State_XX_C).
4. The output of the current state is activated (Entry algorithm State_XX_E).
5. The internal timer is started with the time configured for this state (DT_...).

A state change can occur in two ways:

- **Event-driven**: Through the corresponding input event (e.g., S1_S2).
- **Time-controlled**: By a TimeOut event of the adapter, provided the time DT_... is not set to NO_TIME.

After State_04, the block switches to the State_00 state (from where the sequence can be restarted with START_S1) or directly back to the START state via RESET. A RESET event immediately deactivates all active outputs and returns the block to its initial state. ...

Technical Features

- **Hybrid Transitions**: Each state transition can be individually configured as purely event-driven, purely time-driven, or a combination of both. A time-driven transition takes precedence unless the time is NO_TIME.
- **Initial Values**: All time inputs (DT_...) are pre-set to NO_TIME by default, meaning that the sequence initially runs purely event-driven after startup.
- **Adapter Usage**: Time control is handled entirely via the coupled ATimeOut adapter, enabling a clear separation of functionality and potential reusability.
- **State Feedback**: The output STATE_NR provides a simple way to monitor or visualize the current step externally.

State Overview

1. **START**: Initial, inactive state. All outputs are FALSE.
2. **State_01**: First sequence step. DO_S1 is TRUE. Transition to State_02 via the S1_S2 event or after a certain time, DT_S1_S2.
3. **State_02**: Second sequence step. DO_S2 is TRUE. Transition to State_03 via the S2_S3 event or after a certain time, DT_S2_S3.
4. **State_03**: Third sequence step. DO_S3 becomes TRUE. Transition to State_04 via the S3_S4 event or after a certain time, DT_S3_S4.
5. **State_04**: Fourth sequence step. DO_S4 becomes TRUE. Transition to State_00 viaS4_START event or after a certain time, DT_S4_START.
6. **State_00**: Waiting state after sequence completion. All outputs are FALSE. A new sequence can be started from here with START_S1.
7. **sRESET**: Transition state for reset. Deactivates all outputs and automatically switches to State_00.

Application Scenarios

- **Batch Process Control**: Sequential activation of valves, pumps, or heaters in a chemical or process engineering process.
- **Automated Handling Devices**: Control of the individual steps of a pick-and-place robot (gripping, moving, positioning, and placing).
- **Packaging Machines**: Coordination of processes such as product feeding, packaging, labeling, and ejection.
- **Test Stands**: Automated sequence of testing and measurement steps on a component.

**Automated sequence of testing and measurement steps on a component.## ⚖️ Comparison with similar building blocks Unlike simple timer blocks (TON) or pure state machines (E_SR), sequence_ET_04 combines both in a single, specialized block. It offers a predefined, four-stage structure with dedicated outputs for each step, simplifying and improving the clarity of programming compared to manually linking multiple individual blocks. Blocks like E_CYCLE offer cyclic event triggering but lack individual, state-dependent data outputs or hybrid triggers.

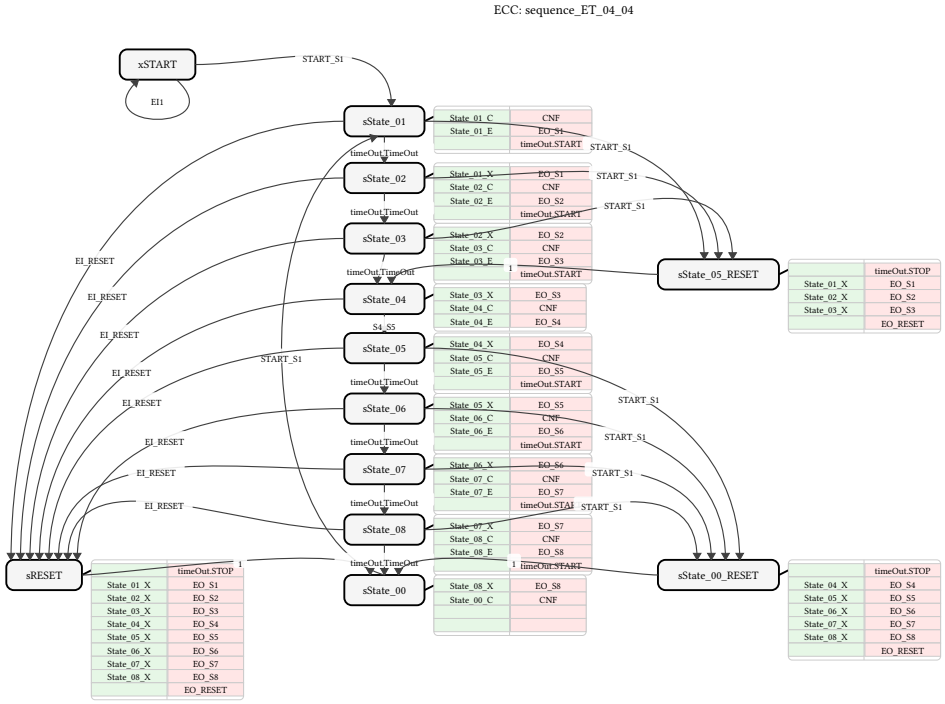
🛠️ Related Exercises

- Exercise_035
- Exercise_035b
- Exercise_035c
- Exercise_036

Conclusion

The sequence_ET_04 is a practical and flexible function block for all applications requiring a clear, cyclic sequence of steps. The combination of event-driven and time-controlled transitions, along with the clear interface featuring separate outputs for each state, makes it particularly easy to maintain and integrate into higher-level controllers. The use of a standard adapter for the timer function keeps the block lean and compatible.

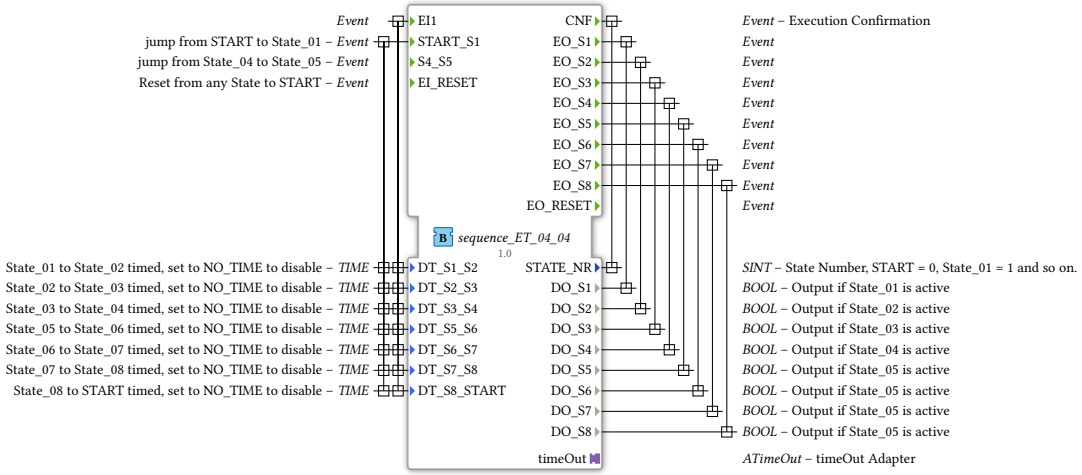
sequence_ET_04_04



• — ○ • — ○ • — ○ •

Introduction

The `sequence_ET_04_04` function block is an 8-output sequencer used to control processes in automation technology. It implements a fixed sequence of states, with transitions between states triggered either by an external event or by an adjustable time interval. This block is particularly suitable for applications requiring cyclic or step-by-step activation of outputs.



Interface Structure

Event Inputs

- **`EI1`**: General input event. In the start state (`xSTART`), this triggers a self-transition (no state change).
- **`START_S1`**: Starts the sequence or jumps from states 1-3 back to state 4, or from states 5-8 back to the final state (`sState_00`). Triggers the transition from the start state (`xSTART`) or from the final state (`sState_00`) to the first active state (`sState_01`).

- **S4_S5**: Triggers the transition from state 4 (sState_04) to state 5 (sState_05). This is the only manual transition in the sequence.
- **EI_RESET**: Resets the sequence from any active state to the reset state (sRESET) and then to the final state (sState_00).

Event Outputs

- **CNF**: Execution Confirmation. Triggered on every state change, it returns the new state number (STATE_NR).
- **EO_S1 to EO_S8**: State events. These are triggered upon entering the corresponding state (State_01 to State_08) and provide the associated Boolean data output (DO_Sx).
- **EO_RESET**: Triggered upon passing through the reset state (sRESET).

Data Inputs

- **DT_S1_S2** (TIME): Time for the automatic transition from State_01 to State_02. Initial value: NO_TIME (disabled).
- **DT_S2_S3** (TIME): Time for the automatic transition from State_02 to State_03. Initial value: NO_TIME (disabled).
- **DT_S3_S4** (TIME): Time for the automatic transition from State_03 to State_04. Initial value: NO_TIME (disabled).
- **DT_S5_S6** (TIME): Time for the automatic transition from State_05 to State_06. Initial value: NO_TIME (disabled).
- **DT_S6_S7** (TIME): Time for the automatic transition from State_06 to State_07. Initial value: NO_TIME (disabled).
- **DT_S7_S8** (TIME): Time for the automatic transition from State_07 to State_08. Initial value: NO_TIME (disabled). * **DT_S8_START** (TIME): Time for the automatic transition from State_08 back to the final state (sState_00). Initial value: NO_TIME (disabled).

Data Outputs

- **STATE_NR** (SINT): Current state number. 0 = START/State_00, 1 = State_01, ..., 8 = State_08.
- **DO_S1 to DO_S8** (BOOL): Logical outputs indicating whether the corresponding state is active. The values are set to TRUE upon entering the state and to FALSE upon exiting.

Adapter

- **timeOut** (Plug, Type: iec61499::events::ATimeOut): Used to implement timed state transitions. The function block starts (START) and stops (STOP) the timer and sets the delay time (DT).

Functionality

The function block operates as a Basic Function Block (BFB) with an extended finite state machine (ECM). The sequence typically iterates through states 1 to 8 in a fixed order. The transition from state 4 to state 5 occurs exclusively via the external event S4_S5. All other transitions (1⟶2, 2⟶3, 3⟶4, 5⟶6, 6⟶7, 7⟶8, 8⟶End) can be time-controlled, provided the corresponding time (DT_...) is not set to NO_TIME.

Each active state (1-8) sets its associated Boolean output (DO_Sx) to TRUE and, if necessary, starts the timer for the next transition. Upon exiting the state, the output is reset to FALSE. With each state change, the confirmation event CNF is triggered with the new event STATE_NR.

The event START_S1 serves not only for the initial start but also for "jumping back" within the sequence: From states 1-3, it jumps back to state 4, and from states 5-8, back to the final state (sState_00). A EI_RESET terminates the sequence from any point, deactivates all outputs, and brings the function block to the consistent final state sState_00.

Technical Features

- **Hybrid Triggering**: Combination of event-driven and time-driven state transitions.
- **Flexible Time Control**: Each timed transition can be individually configured by setting the DT_... inputs or deactivated by setting the value NO_TIME.
- **Safe Reset**: The reset operation (EI_RESET) safely stops all running timers and resets all internal outputs.
- **State Return**: The START_S1 event enables specific returns to the sequence, supporting complex control patterns.

State Overview

The ECC includes the following states:

- **`xSTART`**: Initial, inactive state.
- **`sState_01` to `sState_08`**: Active operating states of the sequence.
- **`sState_00`**: Inactive end state after completion of the sequence.
- **`sRESET`**: Reset state that can be accessed from any active state.
- **`sState_05_RESET` / `sState_00_RESET`**: Intermediate states for the special return actions via `START_S1`.

Application Scenarios

- **Batch Process Control**: Step-by-step activation of valves, pumps, or heaters in a chemical process.
- **Linked Machine Sequences**: Control of individual steps in an assembly or packaging line.
- **Test Stands**: Automated, cyclical test sequences with configurable waiting times between test steps.
- **Safety Sequences**: Ordered start-up and shutdown of a system.

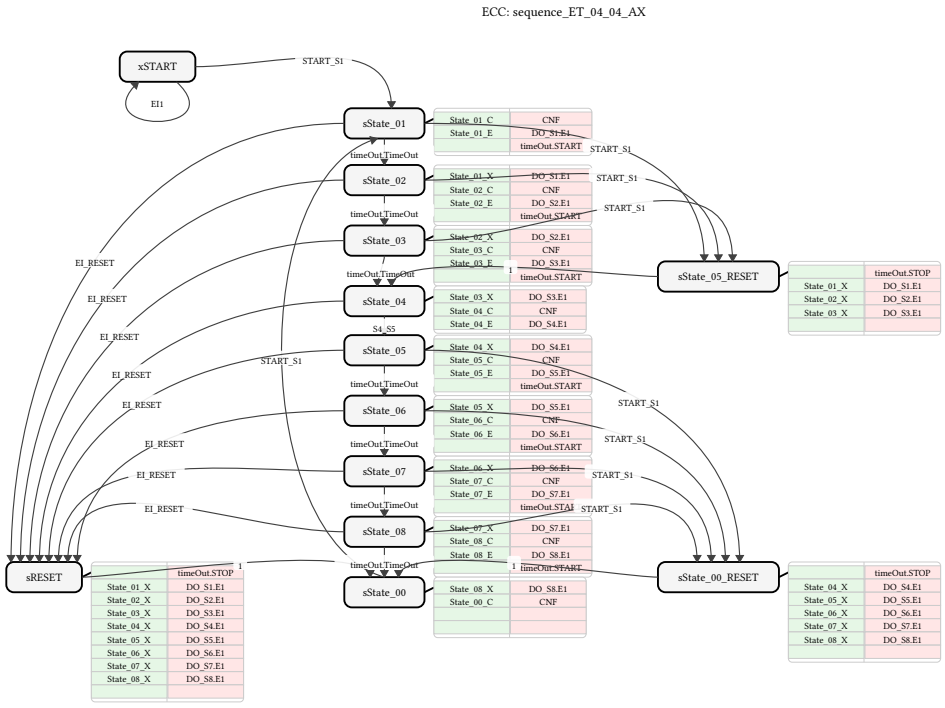
⚖️ Comparison with Similar Function Blocks

Compared to simple timers (e.g., `TON`) or counters (e.g., `CTU`), this function block offers predefined, complex state logic with multiple outputs. Compared to generic step sequence function blocks (SFCs), it is less flexible but more specific, easier to configure, and offers integrated reset and jump functions. It represents a specialized solution for 8-step processes.

Conclusion

The `sequence_ET_04_04`This is a robust and practical sequencer for standard automation tasks with up to eight steps. The combination of time and event control, along with integrated safety and reset functions, makes it a reliable component for repetitive control processes. Its strength lies in its clear structure and simple parameterization of state transitions.

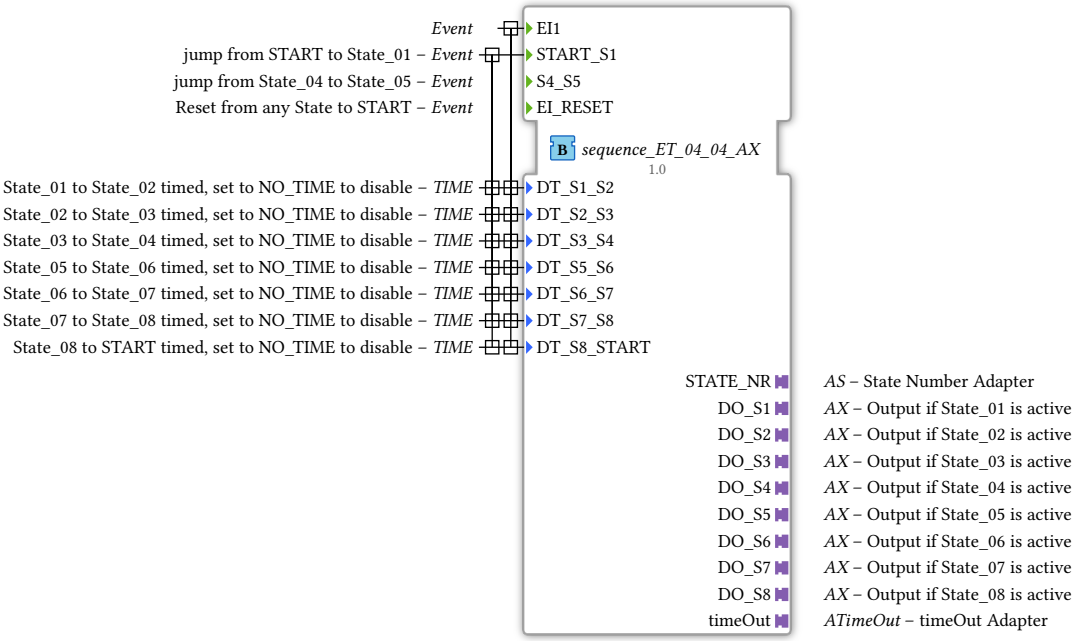
sequence_ET_04_04_AX



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_ET_04_04_AX` is the adapter-based version of `sequence_ET_04_04`. It implements an 8-stage sequence control, where the outputs are controlled via `AX` adapters (event + data) instead of simple BOOL variables. Transitions can be event-driven or time-controlled.



Interface Structure

Event Inputs

- **`EI1`**: General input event.
- **`START_S1`**: Starts the sequence or jumps back within the phases.
- **`S4_S5`**: Manual transition from state 4 to 5.
- **`EI_RESET`**: Global reset of the sequence.

Event Outputs

- **`CNF`**: Execution confirmation with current `STATE_NR`.
- *(Note: The state-specific events EO_Sx are omitted here, as they are included in the adapters)*

Data Inputs

- **`DT_S1_S2`** to **`DT_S8_START`** (TIME): Time delays for automatic transitions.

Data Outputs

- **`STATE_NR`** (SINT): Current state number (0-8).

Adapters

- **`DO_S1` to `DO_S8`** (Plug, Type: `adapter::types::unidirectional::AX`): Adapter outputs for states 1 to 8. Each entry into or exit from a state triggers the event `E1` on the corresponding adapter and sets `D1` to `TRUE` or `FALSE`, respectively.
- **`timeOut`** (Plug, Type: `iec61499::events::ATimeOut`): Interface to the external timer.

Functionality

The logic is exactly the same as `sequence_ET_04_04`. The crucial difference lies in the physical encapsulation of the outputs. While the standard version uses separate event and data lines, the `_AX` version combines these into adapter connections. This reduces complexity in the graphical editor for extensive circuits.

Technical Features

✓ **Adapter Outputs**: Uses the `AX` adapter for all 8 states.

✓ **Combined Triggers**: Events and time sequences control the chain.
✓ **State Feedback**: Continuous output of the `STATE_NR`.

State Overview

See `sequence_ET_04_04`.

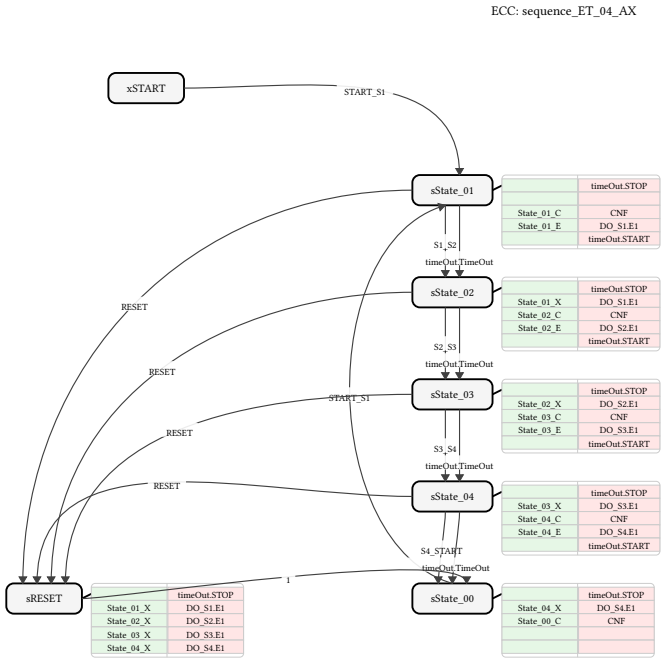
⚖️ Comparison with Similar Function Blocks

- **sequence_ET_04_04**: The version with classic BOOL outputs and EO events.

Conclusion

sequence_ET_04_04_AX` is the ideal choice for complex 8-step controls in systems that consistently rely on adapter-based communication.

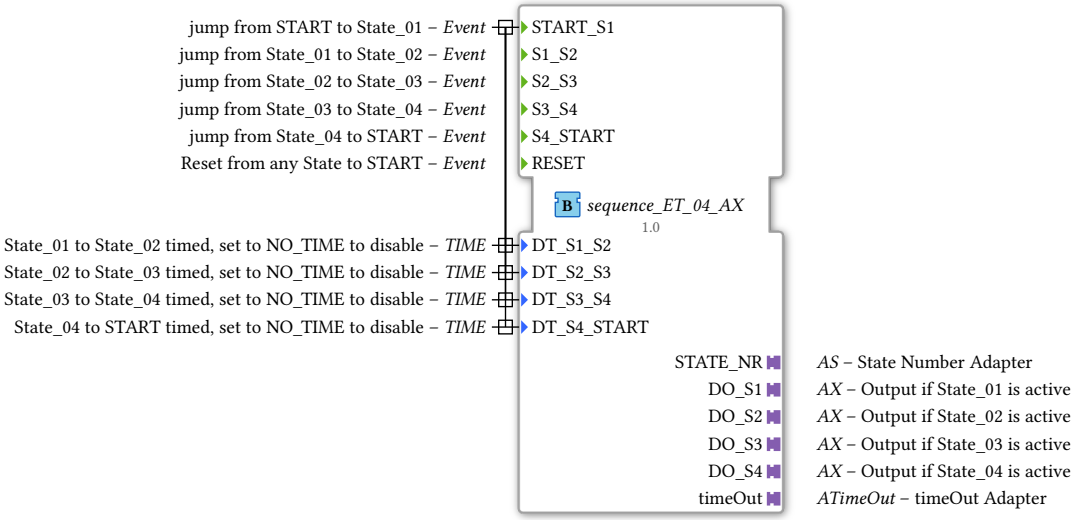
sequence_ET_04_AX



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_ET_04_AX` is a variant of `sequence_ET_04`, which additionally uses adapters (`AX`) for the outputs. It controls a sequence with 4 output states, where transitions can be event-driven or time-controlled.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2**: Manual transition State_01 -> State_02.
- **S2_S3**: Manual transition State_02 -> State_03.
- **S3_S4**: Manual transition State_03 -> State_04.
- **S4_START**: Manual transition State_04 -> START.
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirmation of execution with current state number.

Data Inputs

- **DT_S1_S2**: Time for transition State_01 -> State_02.
- **DT_S2_S3**: Time for transition State_02 -> State_03.
- **DT_S3_S4**: Time for transition State_03 -> State_04.
- **DT_S4_START**: Time for transition State_04 -> START.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

The functionality is essentially the same as `sequence_ET_04`, however, the outputs `DO_S1` to `DO_S4` are not set as simple BOOL variables, but via adapters. This allows for more flexible integration in more complex systems.

Technical Features

- Uses `adapter::types::unidirectional::AX` for the outputs.

Status Overview

See `sequence_ET_04`.

Application Scenarios

Similar to `sequence_ET_04`, but preferred when adapter connections are to be used.

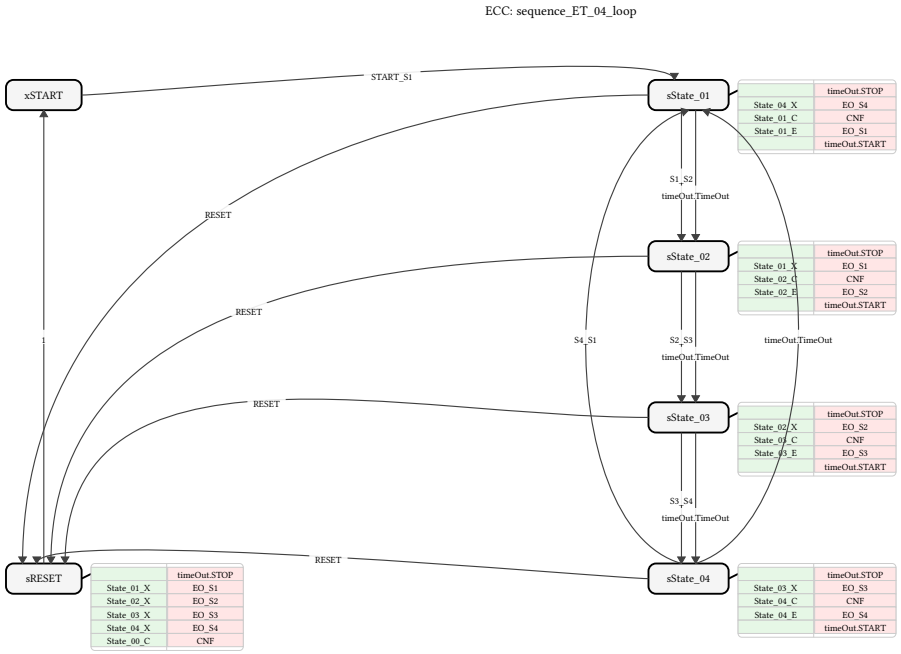
⚖️ Comparison with Similar Function Blocks

- **sequence_ET_04**: The standard version with simple BOOL outputs.

Conclusion

sequence_ET_04_AXoffers the functionality ofsequence_ET_04` with a more modern adapter interface.

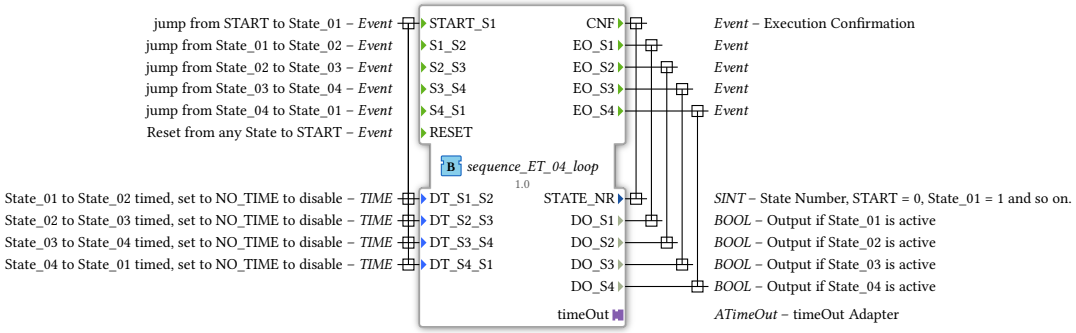
sequence_ET_04_loop



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_ET_04_loop` implements a cyclic sequence with four states. The transition between states can occur either through an external event or after a configurable time has elapsed. The block is designed to implement recurring processes in control applications where actions must be executed sequentially in a loop.



Interface Structure

Event Inputs

- `START_S1`: Starts the sequence and transitions from state `START` to state `State_01`. Transmits the four time parameters `DT_S1_S2`, `DT_S2_S3`, `DT_S3_S4`, and `DT_S4_S1`.
- `S1_S2`: Triggers the transition from `State_01` to `State_02`.
- `S2_S3`: Triggers the transition from `State_02` to `State_03`.
- `S3_S4`: Triggers the transition from `State_03` to `State_04`.
- `S4_S1`: Triggers the transition from `State_04` back to `State_01` (loop).
- `RESET`: Resets the sequence from any state back to the `START` state.

Event Outputs

- `CNF`: Acknowledge event that is triggered on every state change (including reset). Transmits the current state number `STATE_NR`.
- `EO_S1`: Triggered upon entering `State_01`. Transmits the output value `DO_S1`.
- `EO_S2`: Triggered upon entering `State_02`. Transmits the output value `DO_S2`.
- `EO_S3`: Triggered upon entering `State_03`. Transmits the output value `DO_S3`.
- `EO_S4`: Triggered upon entering `State_04`. Transmits the output value `DO_S4`.
- – ...

Data Inputs

- `DT_S1_S2` (TIME): Time for the automatic transition from `State_01` to `State_02`. A value of `NO_TIME` disables the time transition.
- `DT_S2_S3` (TIME): Time for the automatic transition from `State_02` to `State_03`. A value of `NO_TIME` disables the time transition.
- `DT_S3_S4` (TIME): Time for the automatic transition from `State_03` to `State_04`. The value `NO_TIME` disables the time transition.
- `DT_S4_S1` (TIME): Time for the automatic transition from `State_04` back to `State_01`. The value `NO_TIME` disables the time transition.

Data Outputs

- `STATE_NR` (SINT): Current state number (`0`=START, `1`=State_01, `2`=State_02, `3`=State_03, `4`=State_04).
- `DO_S1` (BOOL): Is `TRUE` when `State_01` is active. ... * `DO_S2` (BOOL): Is `TRUE` when `State_02` is active.
- `DO_S3` (BOOL): Is `TRUE` when `State_03` is active.
- `DO_S4` (BOOL): Is `TRUE` when `State_04` is active.

Adapter

- `timeOut` (Plug, Type: `iec61499::events::ATimeOut`): Used internally for implementing timed state transitions.

Functionality

The function block (FB) operates as a BASIC FB with a finite state machine (ECC). The sequence iterates through the states `State_01` -> `State_02` -> `State_03` -> `State_04` and then jumps back to `State_01`. Each state has three main actions:

1. **Exit Action (X)**: Sets the corresponding Boolean output (`DO_Sx`) to `FALSE`.
2. **Confirmation Action (C)**: Sets the state number (`STATE_NR`) and configures the `timeOut` adapter with the time scheduled for the next transition (`DT_...`). Triggers the `CNF` event.
3. **Entry Action (E)**: Sets the associated Boolean output (`DO_Sx`) to `TRUE` and triggers the corresponding output event (`EO_Sx`).

A state transition can occur in two ways:

1. **By Event**: Through the corresponding input event (e.g., `S1_S2`).
2. **By Time**: After the time set in the current state in the `timeOut` adapter has elapsed, unless this is `NO_TIME`.

The `RESET` input always leads to the special `sRESET` state, which switches off all active outputs, sets the state number to 0, and then returns to the `START` state.

Technical Features

- **Hybrid Triggering**: Each state transition can be individually configured to be either event-driven or time-driven. This allows for maximum flexibility within a sequence.
- **Initial Values**: The time parameters are initialized to `NO_TIME` by default, meaning that all time-driven transitions are initially disabled and await an external event.
- **Adapter Usage**: Time control is consistently handled via the standardized `ATimeOut` adapter, increasing reusability and clarity.
- **Status Feedback**: The current position in the sequence is always visible externally via the `STATE_NR` output.

State Overview

The ECC consists of six states:

1. **xSTART**: Initial, inactive state. Waits for `START_S1`.
2. **sState_01**: Active state
3. Sets `DO_S1`. Can switch to `sState_02` via a `S1_S2` event or timeout.
4. **sState_02**: Active State
5. Sets `DO_S2`. Can transition to `sState_03` via the `S2_S3` event or timeout.
6. **sState_03**: Active State
7. Sets `DO_S3`. Can transition to `sState_04` via the `S3_S4` event or timeout.
8. **sState_04**: Active State
9. Sets `DO_S4`. Can transition back to `sState_01` via the `S4_S1` event or timeout (loop).
10. **sRESET**: Reset state. Switches off all outputs, sets `STATE_NR` to 0, and automatically switches back to `xSTART`.

Application Scenarios

- **Cyclical Process Control**: Control of machines that perform a repeating work cycle with multiple steps (e.g., filling, heating, mixing, emptying).
- **Traffic Light Circuits**: Modeling a simple, multi-phase traffic light system where each phase can last a fixed time or be terminated prematurely.
- **Batch Processes**: Processing batch processes where individual steps are terminated either by sensors (events) or after a minimum time.

⚖️ Comparison with Similar Function Blocks

Unlike simple timer blocks or flip-flops, this function block orchestrates a complete, state-based sequence. Compared to a `E_CYCLE`The `sequence_ET_04_loop` block offers explicit state logic with clear transition conditions and the ability to trigger each step individually. It is more specialized and structured than a custom-programmed sequence of `E_SR` and `E_DELAY` blocks.

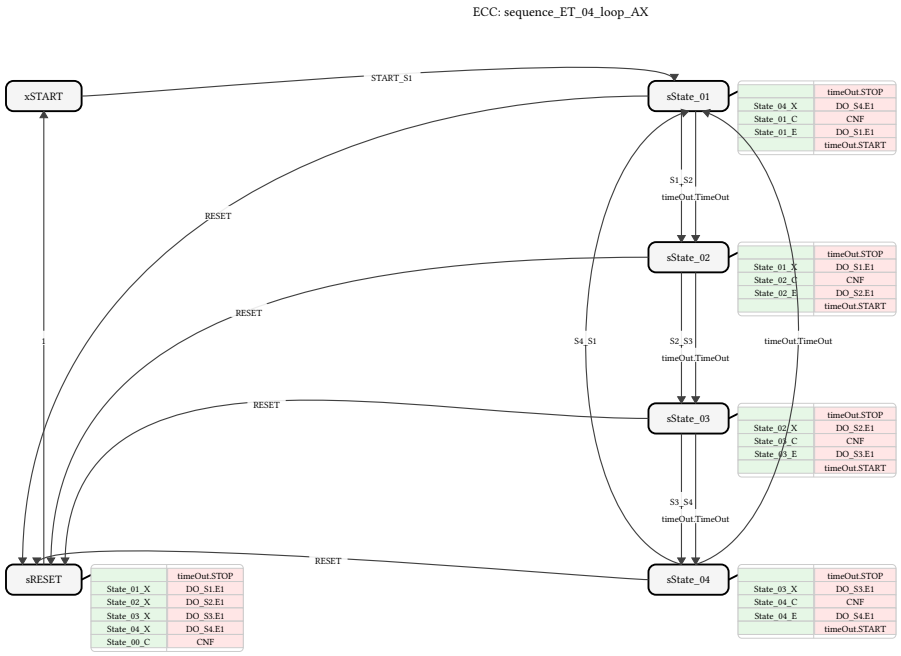
🛠️ Related Exercises

- Exercise_037

Conclusion

The `sequence_ET_04_loop` block is a robust and flexible building block for implementing cyclic 4-step sequences. Its strength lies in its hybrid triggering, which allows you to choose between event-driven and time-based triggering for each step. The clear state machine, the feedback of the current state, and the integrated reset make it a reliable component for recurring control tasks in automation technology.

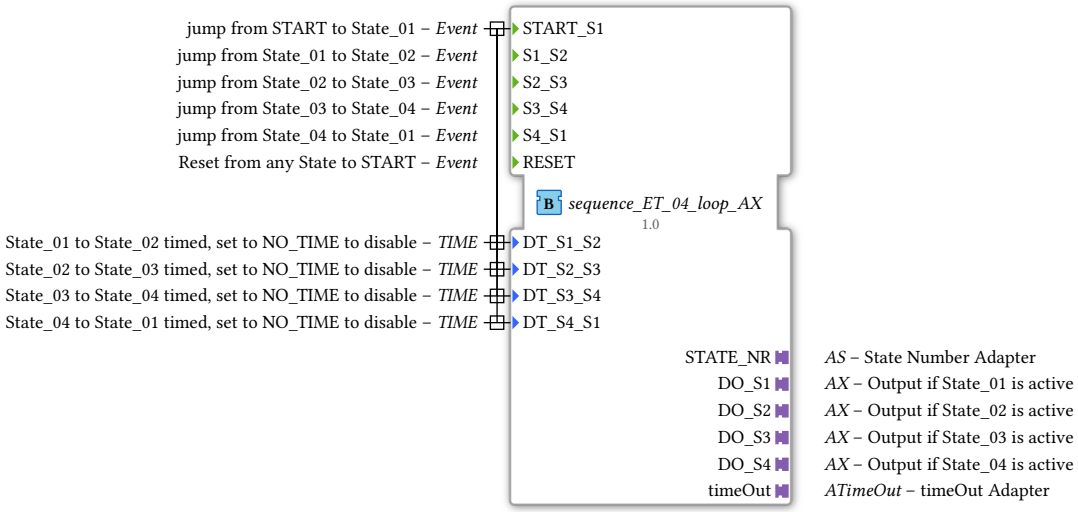
sequence_ET_04_loop_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_ET_04_loop_AXis a variant ofsequence_ET_04_loop that additionally uses adapters (AX`) for the outputs. It controls a cyclic sequence with 4 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2**: Manual transition State_01 -> State_02.
- **S2_S3**: Manual transition State_02 -> State_03.
- **S3_S4**: Manual transition State_03 -> State_04.
- **S4_S1**: Manual transition State_04 -> State_01 (Loop).
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirmation of execution.

Data Inputs

- **DT_S1_S2**: Time for transition State_01 -> State_02.
- **DT_S2_S3**: Time for transition State_02 -> State_03.
- **DT_S3_S4**: Time for transition State_03 -> State_04.
- **DT_S4_S1**: Time for transition State_04 -> State_01.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

Corresponds to `sequence_ET_04_loop`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

Status Overview

See `sequence_ET_04_loop`.

Application Scenarios

For cyclic processes that are to be controlled via adapters.

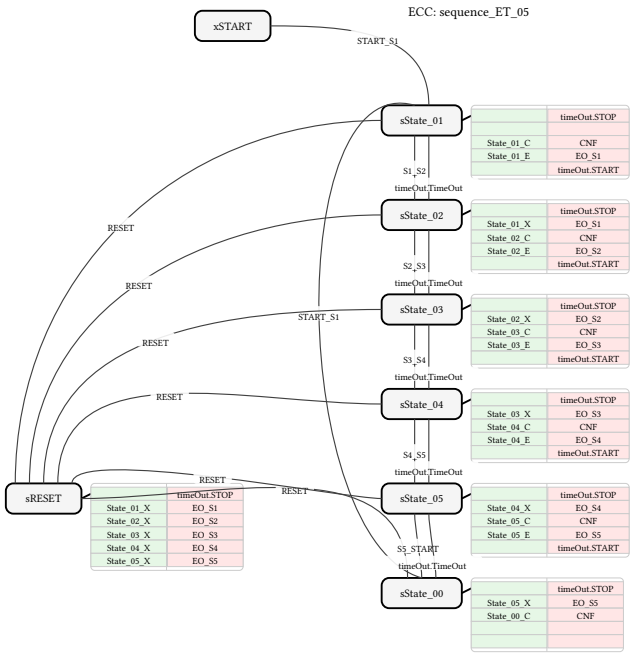
⚖️ Comparison with Similar Function Blocks

- **sequence_ET_04_loop**: Standard version without adapters.

Conclusion

Adapter version of the 4-step loop sequencer.

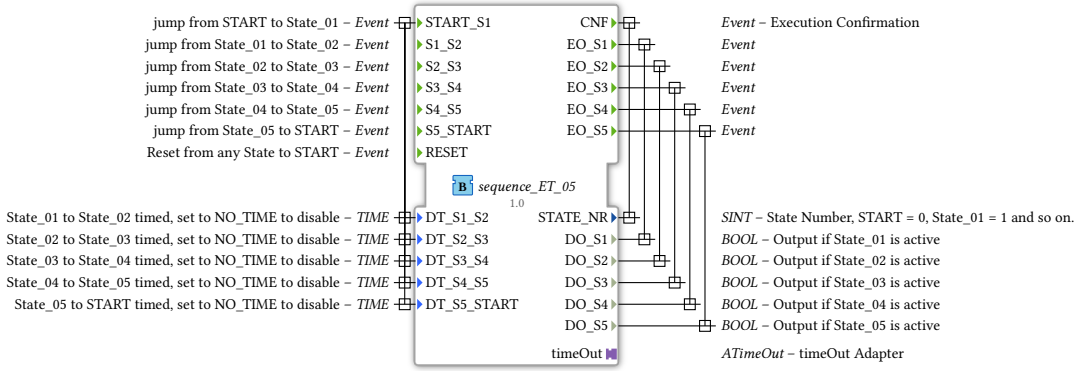
sequence_ET_05



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_ET_05` implements sequential control with five defined states (State_01 to State_05) and a start state (START). The transition between states can occur either through an external event or after an adjustable time. The block is designed for applications that require a step-by-step sequence of actions with flexible transition conditions.



Interface Structure

Event Inputs

- **`START_S1`**: Switches from the START state or State_00 to the State_01 state. Transmits the time parameters for all state transitions.
- **`S1_S2`**: Manual transition from State_01 to State_02.
- **`S2_S3`**: Manual transition from State_02 to State_03.
- **`S3_S4`**: Manual transition from State_03 to State_04.
- **`S4_S5`**: Manual transition from State_04 to State_05.
- **`S5_START`**: Manual transition from State_05 back to State_00 (representing START).
- **`RESET`**: Instantly returns to State_00 (START) from any state and disables all outputs.

Event Outputs

- **CNF**: Execution confirmation. Triggered on every state change and returns the new state number (STATE_NR).
- **EO_S1**: Triggered upon entering State_01 and returns the active output DO_S1.
- **EO_S2**: Triggered upon entering State_02 and returns the active output DO_S2.
- **EO_S3**: Triggered upon entering State_03 and returns the active output DO_S3.
- **EO_S4**: Triggered upon entering State_04 and provides the active output DO_S4.
- **EO_S5**: Triggered upon entering State_05 and provides the active output DO_S5.

Data Inputs

- **DT_S1_S2** (TIME): Dwell time in State_01 before automatic transition to State_02. Initial value: NO_TIME (disabled).
- **DT_S2_S3** (TIME): Dwell time in State_02 before automatic transition to State_03. Initial value: NO_TIME (disabled).
- **DT_S3_S4** (TIME): Time spent in State_03 before automatic transition to State_04. Initial value: NO_TIME (disabled).
- **DT_S4_S5** (TIME): Time spent in State_04 before automatic transition to State_05. Initial value: NO_TIME (disabled).
- **DT_S5_START** (TIME): Time spent in State_05 before automatic transition back to State_00 (START). Initial value: NO_TIME (disabled).

Data Outputs

- **STATE_NR** (SINT): Current state number (START = 0, State_01 = 1, ..., State_05 = 5).
- **DO_S1** (BOOL): Is TRUE when State_01 is active.
- **DO_S2** (BOOL): Is TRUE when State_02 is active.
- **DO_S3** (BOOL): Is TRUE when State_03 is active.
- – **DO_S4** (BOOL): Is TRUE when State_04 is active.
- **DO_S5** (BOOL): Is TRUE when State_05 is active.

Adapter

- **timeOut** (Plug, Type: iec61499::events::ATimeOut): Used internally for timed state transitions. The function block starts and stops the timer and sets its runtime (DT).

Functionality

The function block operates as a Basic function block with an Execution Control Chart (ECC). The initial state is xSTART. Upon a START_S1 event, the function block (FB) transitions to state sState_01. In each active state (sState_01 to sState_05), the following actions are performed:

1. The timeOut adapter is stopped.
2. The output of the previous state is deactivated (exit algorithm X).
3. The state number STATE_NR is updated, and the dwell time for the *next* transition is passed to the timer (confirmation algorithm C). The CNF event is then triggered.
4. The output of the previous state is deactivated (exit algorithm X). 4. The output of the current state is activated (entry algorithm E). The corresponding EO_Sx event is triggered.
5. The timeOut adapter with the previously set time is started.

A state change can now occur in two ways:

1. **Event-driven:** Through the corresponding manual event (e.g., S1_S2).
2. **Time-driven:** Through the adapter's timeOut.TimeOut event, provided the time (DT_Sx_Sy) is not set to NO_TIME.

The RESET event always leads to the special state sRESET, in which all outputs are deactivated, and from there immediately to the state sState_00 (representing the logical START state with STATE_NR = 0).

Technical Features

- **Flexible Transitions:** Each state transition can be configured independently as event-driven or time-driven. Time-driven control is deactivated by setting the corresponding DT_ input to the constant NO_TIME.

- **Immediate Reset:** The `RESET` input always has priority and immediately interrupts the current sequence.
- **Clearly Defined Interface:** The status number and active outputs are always available as data outputs and are confirmed by an event at each step.
- **Adapter-Based Time Control:** The use of a standardized TimeOut adapter makes the internal time management robust and reusable.

State Overview

The ECC includes the following states:

- **xSTART:** Initial idle state.
- **sState_01 ... sState_05:** The five sequential operating states.
- **sState_00:** Represents the logical START state after completion of the sequence or after a reset. No output is active in this state.
- **sRESET:** Intermediate state for deactivating all outputs during a reset operation.

Application Scenarios

- **Batch Process Control:** Step-by-step processing of recipes in mixing or filling systems, where each step is manually acknowledged or automatically advances after a defined time.
- **Linked Machine Sequences:** Control of a machine whose cycle consists of several sequentially performed actions (e.g., loading, processing, inspection, ejection).
- **Test Sequences:** Automated test benches that perform a series of tests sequentially, where each test has a fixed duration or can be manually confirmed.

⚖️ Comparison with Similar Components

Compared to simple timer chains or counters, `sequence_ET_05` offers a fully encapsulated state machine with clear input/output events and the flexible combination of time and event control. Compared to a genericThe `E_SR` or `E_CTUD` function block, when used in an ad-hoc configuration, provides a predefined, tested, and easily configurable solution for a common control task.

🛠️ Related Exercises

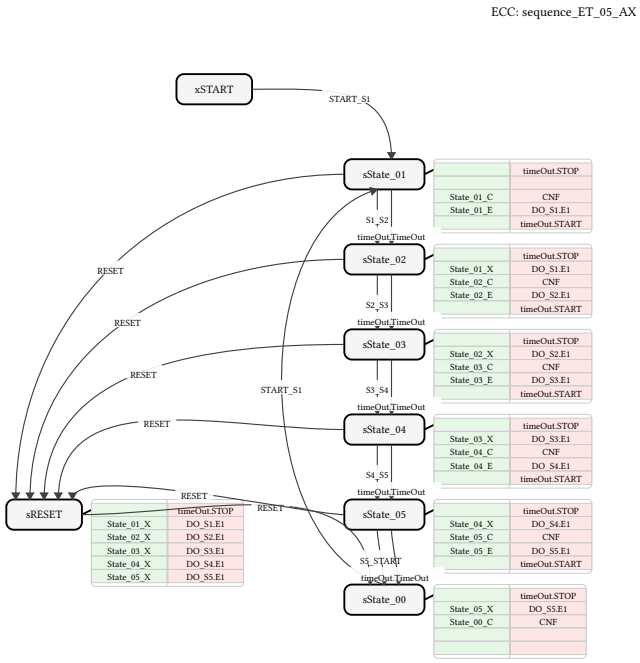
- Exercise_039
- Exercise_039a

The `sequence_ET_05` is a well-structured and flexible function block for implementing 5-step sequences. The clear separation of control flow (events) and parameters (times), as well as the option to configure each transition either manually or automatically, makes it suitable for a wide variety of control tasks in automation technology. The use of standardized adapters and comprehensive interface documentation facilitate integration into larger applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter block on ms-muc-docs.de

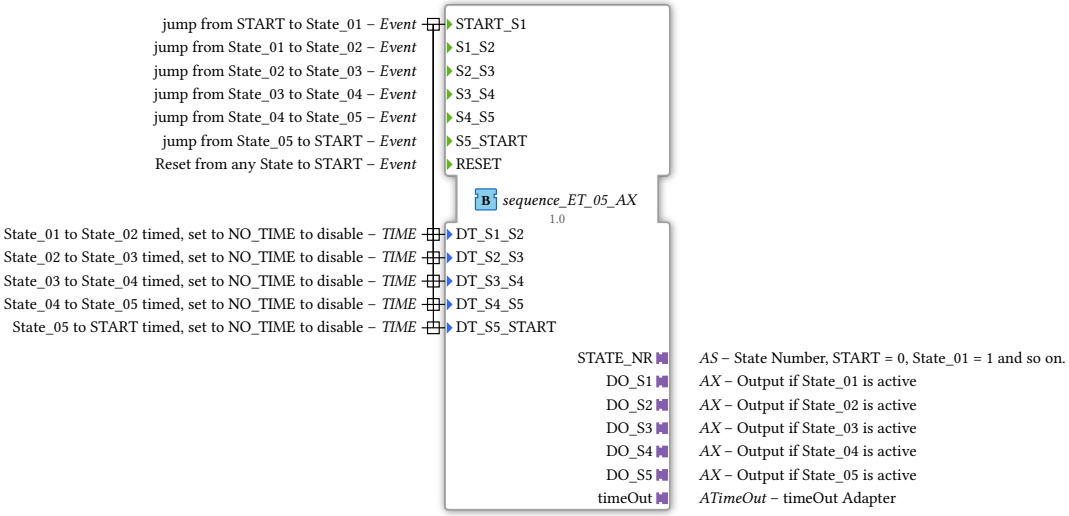
sequence_ET_05_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_ET_05_AXis a variant ofsequence_ET_05 that additionally uses adapters (AX`) for the outputs. It controls a sequence with 5 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2**: Manual transition State_01 -> State_02.
- **S2_S3**: Manual transition State_02 -> State_03.
- **S3_S4**: Manual transition State_03 -> State_04.
- **S4_S5**: Manual transition State_04 -> State_05.
- **S5_START**: Manual transition State_05 -> START.

- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirmation of execution.

Data Inputs

- **DT_S1_S2**: Time for transition State_01 -> State_02.
- **DT_S2_S3**: Time for transition State_02 -> State_03.
- **DT_S3_S4**: Time for transition State_03 -> State_04.
- **DT_S4_S5**: Time for transition State_04 -> State_05.
- **DT_S5_START**: Time for transition State_05 -> START.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Output adapter for State_05.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

Corresponds to `sequence_ET_05`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

Status Overview

See `sequence_ET_05`.

Application Scenarios

For 5-step sequences with adapter connection.

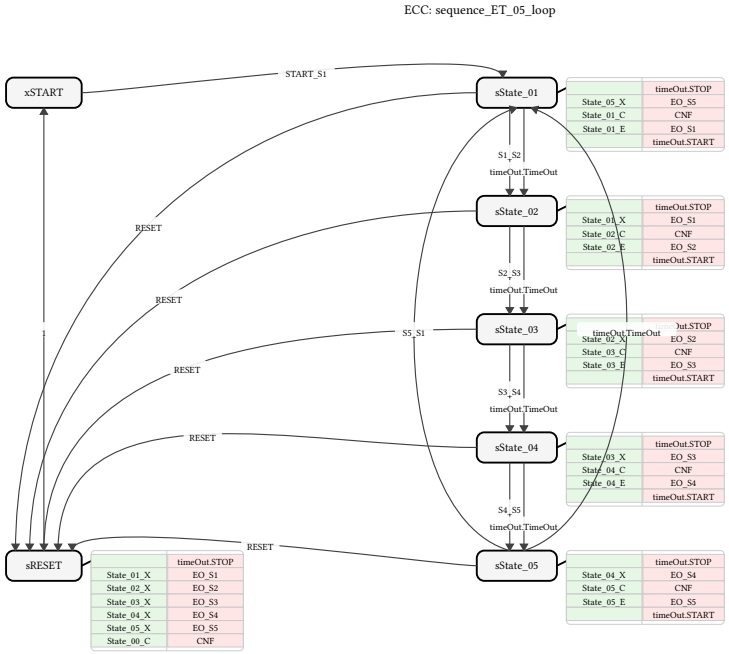
⚖️ Comparison with Similar Function Blocks

- **sequence_ET_05**: Standard version without adapter.

Conclusion

Adapter version of the 5-step sequencer.

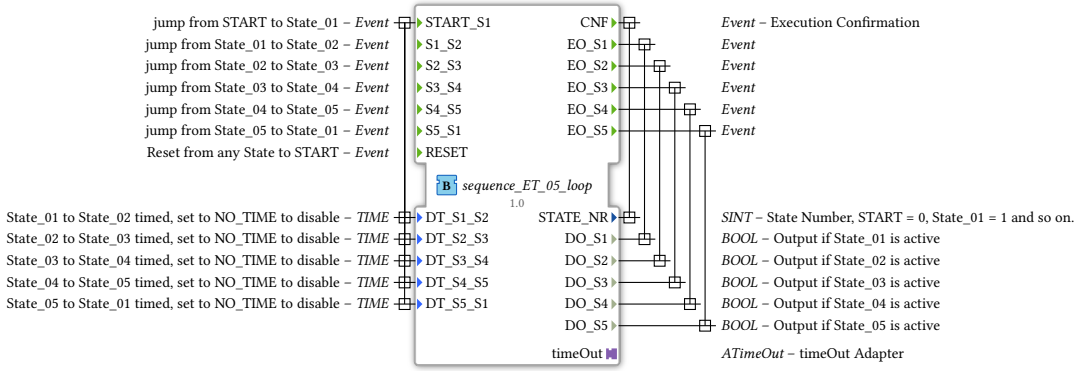
sequence_ET_05_loop



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_ET_05_loop` implements a cyclic sequence with five states. The transition between the individual states can occur either through an external event or after a configurable time has elapsed. This block is designed for applications where a process must be executed step by step, triggering various actions in a fixed sequence, such as in packaging machines, assembly processes, or washing programs.



Interface Structure

Event Inputs

- **`START_S1`**: Starts the sequence and executes the transition from the start state (`START`) to the first state (`State_01`). Transmits the time values for all time-controlled transitions.
- **`S1_S2`**: Triggers the transition from `State_01` to `State_02`.
- **`S2_S3`**: Triggers the transition from `State_02` to `State_03`.
- **`S3_S4`**: Triggers the transition from `State_03` to `State_04`.
- **`S4_S5`**: Triggers the transition from `State_04` to `State_05`.
- **`S5_S1`**: Triggers the transition from `State_05` back to `State_01` (cycle).
- **`RESET`**: Resets the sequence from any state back to the start state (`START`).

Event Outputs

- **CNF**: Acknowledge event triggered on every state change. Transmits the current state number (STATE_NR).
- **EO_S1**: Triggered upon entering State_01. Transmits the output value DO_S1.
- **EO_S2**: Triggered upon entering State_02. Transmits the output value DO_S2.
- **EO_S3**: Triggered upon entering State_03. Transmits the output value DO_S3.
- **EO_S4**: Triggered upon entering State_04. Transmits the output value DO_S4.
- **EO_S5**: Triggered upon entering State_05. Transmits the output value DO_S5.

Data Inputs

- **DT_S1_S2** (Type: TIME, Initial value: NO_TIME): Time for the automatic transition from State_01 to State_02. The timed transition is disabled at NO_TIME.
- **DT_S2_S3** (Type: TIME, Initial value: NO_TIME): Time for the automatic transition from State_02 to State_03.
- **DT_S3_S4** (Type: TIME, Initial value: NO_TIME): Time for the automatic transition from State_03 to State_04.
- **DT_S4_S5** (Type: TIME, Initial value: NO_TIME): Time for the automatic transition from State_04 to State_05.
- **DT_S5_S1** (Type: TIME, Initial value: NO_TIME): Time for the automatic transition from State_05 to State_01.

Data Outputs

- **STATE_NR** (Type: SINT): Outputs the number of the current state (START = 0, State_01 = 1, ..., State_05 = 5).
- **DO_S1** (Type: BOOL): Is TRUE when State_01 is active.
- **DO_S2** (Type: BOOL): Is TRUE when State_02 is active.
- **DO_S3** (Type: BOOL): Is TRUE when State_03 is active.
- **DO_S4** (Type: BOOL): Is TRUE when State_04 is active.
- **DO_S5** (Type: BOOL): Is TRUE when State_05 is active.

Adapter

- **timeOut** (Type: ATimeOut): A plug-in adapter for implementing timed state transitions. The function block (FB) uses the interfaces START, STOP, and DT (time value), as well as the event TimeOut.

Functionality

The FB is implemented as a Basic FB with an Execution Control Chart (ECC). The sequence begins in state xSTART. The event START_S1 activates the first state, sState_01. The following actions are executed upon each state entry:

1. The timer adapter is stopped (timeOut.STOP).
2. The output of the previous state is reset (e.g., State_05_X for DO_S5).
3. The state number STATE_NR is updated, and the timer for the next transition is configured (e.g., State_01_C).
4. The output of the new state is set (e.g., State_01_E for DO_S1).
5. The timer adapter is started with the time configured for this transition (timeOut.START).

A state change can now occur in two ways:

- **Event-driven:** Through the corresponding input event (e.g., S1_S2).
- **Time-Controlled:** Triggered by the adapter's TimeOut event, provided the time (DT_...) is not set to NO_TIME.

The sequence cycles through states 1 to 5 and then jumps from State_05 back to State_01, creating an endless cycle. The RESET event triggers the reset state sRESET, regardless of the current state, from where the sequence automatically returns to the xSTART state.

Technical Features

- **Hybrid Triggering:** Each state transition can be individually configured to be either event-driven or time-controlled. This allows for maximum flexibility within a fixed sequence.
- **Safe State Outputs:** The Boolean outputs (DO_Sx) are explicitly reset via an exit algorithm (State_XX_X) when a state is exited. This prevents the unintentional persistence of the TRUE signal.

- **Explicit Timer Control:** The timer is restarted and stopped with each state change, ensuring precise and deterministic timing.
- **Constants:** The function block uses defined constants (`sequence::State_XX`, `NO_TIME`) from imported libraries, which improves the maintainability and readability of the code.
-

State Overview

The ECC consists of seven states:

1. **`xSTART`**: Inactive initial state.
2. **`sState_01`** to **`sState_05`**: The five active sequence states.
3. **`sRESET`**: Intermediate state that resets all outputs on reset and confirms the `xSTART`-State sends.

Application Scenarios

- **Control of Cyclic Processes:** Automated machining centers (e.g., multi-tool lathes) where each step takes a specific amount of time or is completed by a sensor event.
- **Batch Processes:** Batch processing in the food or pharmaceutical industries, where mixing, heating, and filling occur sequentially.
- **Safety Sequences:** Monitored start or stop procedures where each step is manually enabled (`EVENT`) or continues automatically after a waiting period (`TIME`).
- **Test Stands:** Automated testing sequences where various measurements or functional tests are performed sequentially.

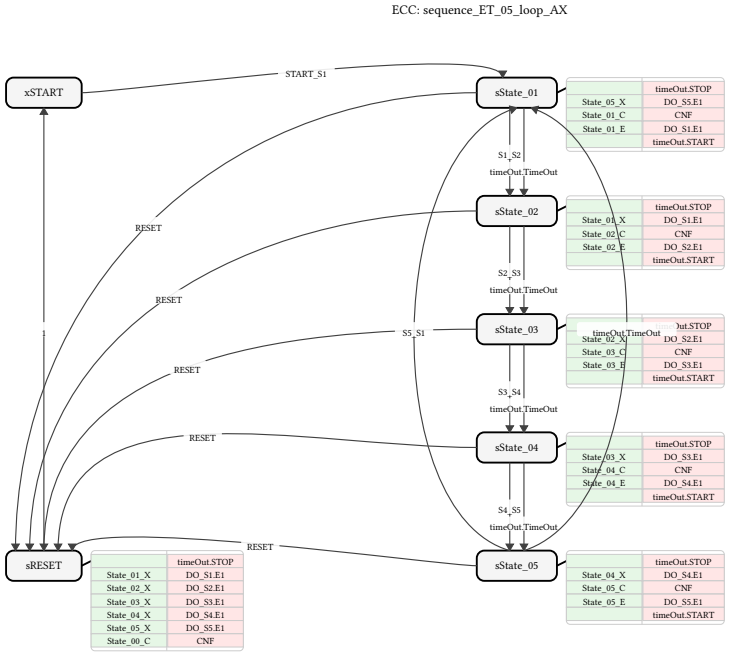
⚖️ Comparison with Similar Function Blocks

Compared to simple timer function blocks or self-holding relay cascades, `sequence_ET_05_loop` offers a predefined, cyclic structure with integrated time and event control. Compared to programming with individual blocks (e.g., `E_SR`, `E_DELAY`), it reduces wiring effort and improves clarity. Other sequencer function blocks may have a variable number of steps or different transition logic (time only OR event only). This function block is distinguished by its combined `EVENT`/`TIME` triggering per step.

Conclusion

The `sequence_ET_05_loop` is a robust and clearly structured function block for implementing cyclic 5-step sequences. The combination of event-driven and time-controlled transitions makes it extremely flexible for a wide variety of automation tasks. Explicit control of the outputs and the timer ensures deterministic and reliable behavior. It is ideally suited for efficiently and clearly mapping recurring process flows with a fixed number of steps in an IEC 61499-based controller.

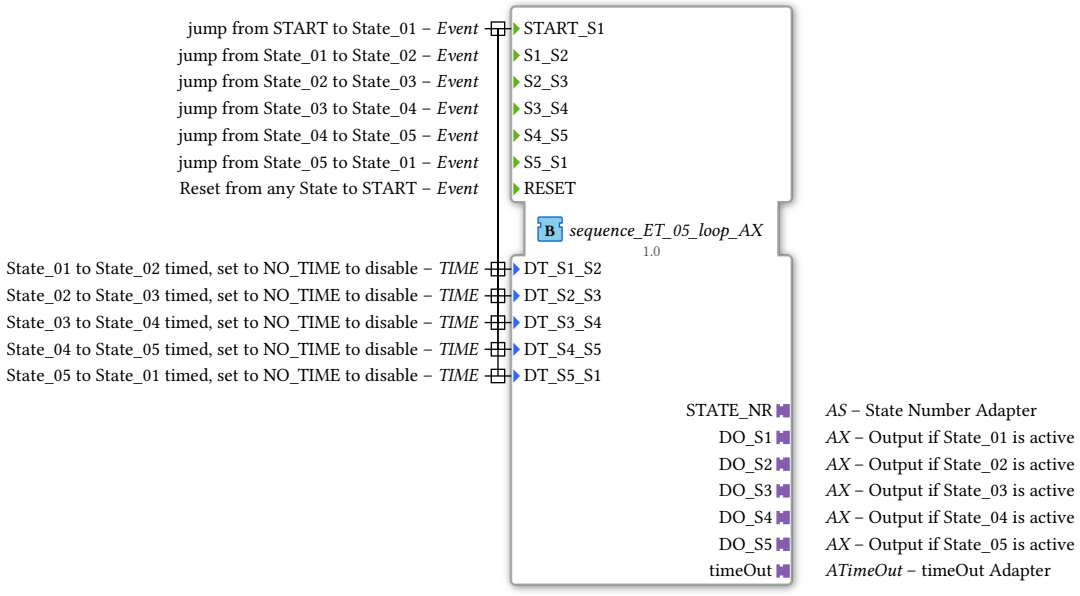
sequence_ET_05_loop_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_ET_05_loop_AXis a variant ofsequence_ET_05_loop that additionally uses adapters (AX`) for the outputs. It controls a cyclic sequence with 5 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2**: Manual transition State_01 -> State_02.
- **S2_S3**: Manual transition State_02 -> State_03.
- **S3_S4**: Manual transition State_03 -> State_04.

- **S4_S5**: Manual transition State_04 -> State_05.
- **S5_S1**: Manual transition State_05 -> State_01 (loop).
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Acknowledges execution.

Data Inputs

- **DT_S1_S2**: Time for transition State_01 -> State_02.
- **DT_S2_S3**: Time for transition State_02 -> State_03.
- **DT_S3_S4**: Time for transition State_03 -> State_04.
- **DT_S4_S5**: Time for transition State_04 -> State_05.
- **DT_S5_S1**: Transition time from State_05 to State_01.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Output adapter for State_05.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

Corresponds to `sequence_ET_05_loop`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_ET_05_loop`.

Application Scenarios

For cyclic 5-step processes that are to be controlled via adapters.

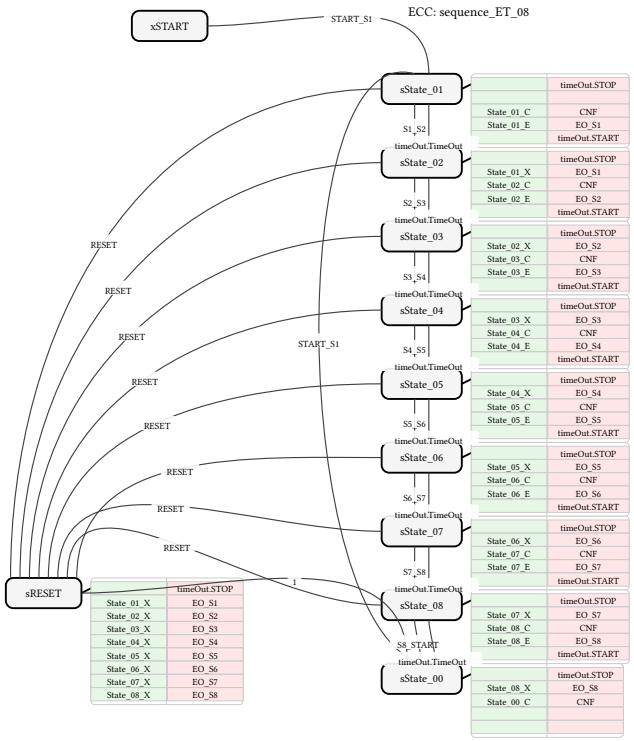
⚖️ Comparison with Similar Function Blocks

- **sequence_ET_05_loop**: Standard version without adapters.

Conclusion

Adapter version of the 5-step loop sequencer.

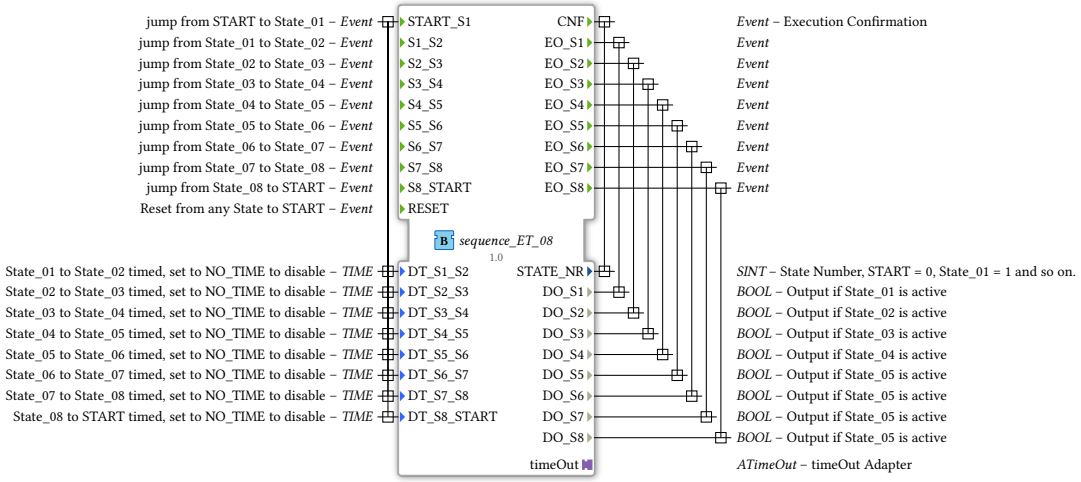
sequence_ET_08



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_ET_08` is a sequencer with eight outputs. It allows the execution of a fixed sequence of steps, where the transition from one step to the next can be triggered either by an external event or by an adjustable time interval. This block is particularly suitable for control tasks requiring a cyclic or time-controlled sequence of actions.



Interface Structure

Event Inputs

- `START_S1`: Jumps from state `START` to state `State_01`. Transfers all timestamps (`DT_S1_S2` to `DT_S8_START`).
- `S1_S2`: Jumps from `State_01` to `State_02`.
- `S2_S3`: Jumps from `State_02` to `State_03`.
- `S3_S4`: Jumps from `State_03` to `State_04`.

- S4_S5: Jumps from State_04 to State_05.
- S5_S6: Jumps from State_05 to State_06.
- S6_S7: Jumps from State_06 to State_07.
- S7_S8: Jumps from State_07 to State_08.
- S8_START: Jumps from State_08 back to START.
- RESET: Resets the function block from any state back to START.

Event Outputs

- CNF: Execution confirmation. Set with every state change and transmits the current state number (STATE_NR).
- EO_S1 to EO_S8: State-specific event outputs. These inputs are triggered upon entering the corresponding state (State_01 to State_08) and transmit the respective Boolean data output (DO_S1 to DO_S8).

Data Inputs

- DT_S1_S2 to DT_S8_START (Type TIME): Define the duration for the automatic transition from the current to the next state. If the value is set to NO_TIME, the timed transition for this step is disabled, and an event is required.
-

Data Outputs

- STATE_NR (Type SINT): Outputs the current state number (START = 0, State_01 = 1, ..., State_08 = 8).
- DO_S1 to DO_S8 (Type BOOL): Logical outputs that are TRUE as long as the function block is in the corresponding state (State_01 to State_08).

Adapter

- timeOut (Type iec61499::events::ATimeOut): A timer adapter used to implement timed state transitions.

Functionality

The function block is implemented as a BASIC FB with an Execution Control Chart (ECC). It starts in the initial state xSTART. An event START_S1 leads to the first active state sState_01. Each active state (sState_01 to sState_08) performs the following actions upon entry:

1. Stops the running timer.
2. Executes the exit algorithm of the previous state (switches off the previous output).
3. Executes the Confirmation algorithm (*_C), which sets the time for the next possible automatic transition in the adapter (STATE_NR).
4. Executes the Entry algorithm (*_E), which activates the associated Boolean output (DO_Sx) and triggers the corresponding event (EO_Sx).
5. Starts the timer with the duration set for this state (DT_*).

The transition to the next state can occur in two ways:

1. **Event-driven:** Through the corresponding jump event (e.g., S1_S2).
2. **Time-driven:** Through the adapter's TimeOut event, provided the duration (DT_*) is not NO_TIME.

The RESET event leads to a dedicated reset state (sRESET), which switches off all active outputs before transitioning to the inactive START state (sState_00).

Technical Features

- **Hybrid Transitions:** Each step offers two parallel transition conditions (event OR time), providing maximum flexibility.
- **Safe State Handling:** The timer is always stopped during a state change, and the outputs are cleanly deactivated by defined exit algorithms.
- **Configurable Times:** The time for each step can be set individually or deactivated using NO_TIME.
- **Explicit State Feedback:** The output STATE_NR allows for easy external monitoring of the current step position.

State Overview

1. **xSTART:** Initial, inactive state (at the start of the function block).
2. **sState_00:** Inactive START state (after reset or cycle end). STATE_NR = 0.

3. **sState_01 to sState_08:** Active states of the sequence. `STATE_NR = 1` to `8`. The corresponding outputs `DO_S1` to `DO_S8` are active.
4. **sRESET:** Temporary state that switches off all active outputs upon a `RESET` event.

Application Scenarios

- Control of cyclic processes in packaging or manufacturing machines.
- Step sequence for an automated test or calibration process.
- Control of an exposure or rinsing sequence in semiconductor manufacturing.
- General state machines where steps can be triggered by both sensor input (events) and fixed time intervals.

⚖️ Comparison with similar components

Compared to simpler sequencers (e.g., `E_SR` or `E_CTU` in series), `sequence_ET_08` offers a fully predefined step sequence with integrated timing and dedicated outputs for each step. Unlike a custom-programmed SFC (Sequential Function Chart), the logic is hard-coded, making it easier to use but also less flexible. Components like `E_DELAY` would need to be added externally, whereas here the timing functionality is integrated.

Compared to simpler sequencers (e.g., `E_DELAY`), `sequence_ET_08` offers a fully predefined step sequence with integrated timing and dedicated outputs for each step.

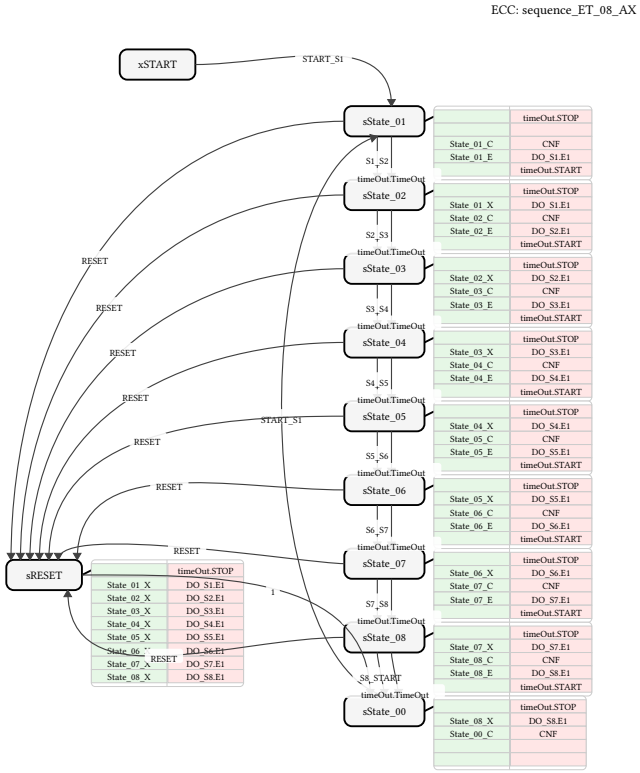
Conclusion

The `sequence_ET_08` is a robust and easy-to-configure sequencer block for IEC 61499. Its strength lies in its combined event and time control, as well as its clear, step-defined interface. It is ideally suited for standardized control sequences with up to eight steps where a high degree of predictability and simple parameterization are desired. For processes with a variable number of steps or more complex branching, more flexible solutions such as composite function blocks or custom SFCs are preferable.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter block on ms-muc-docs.de

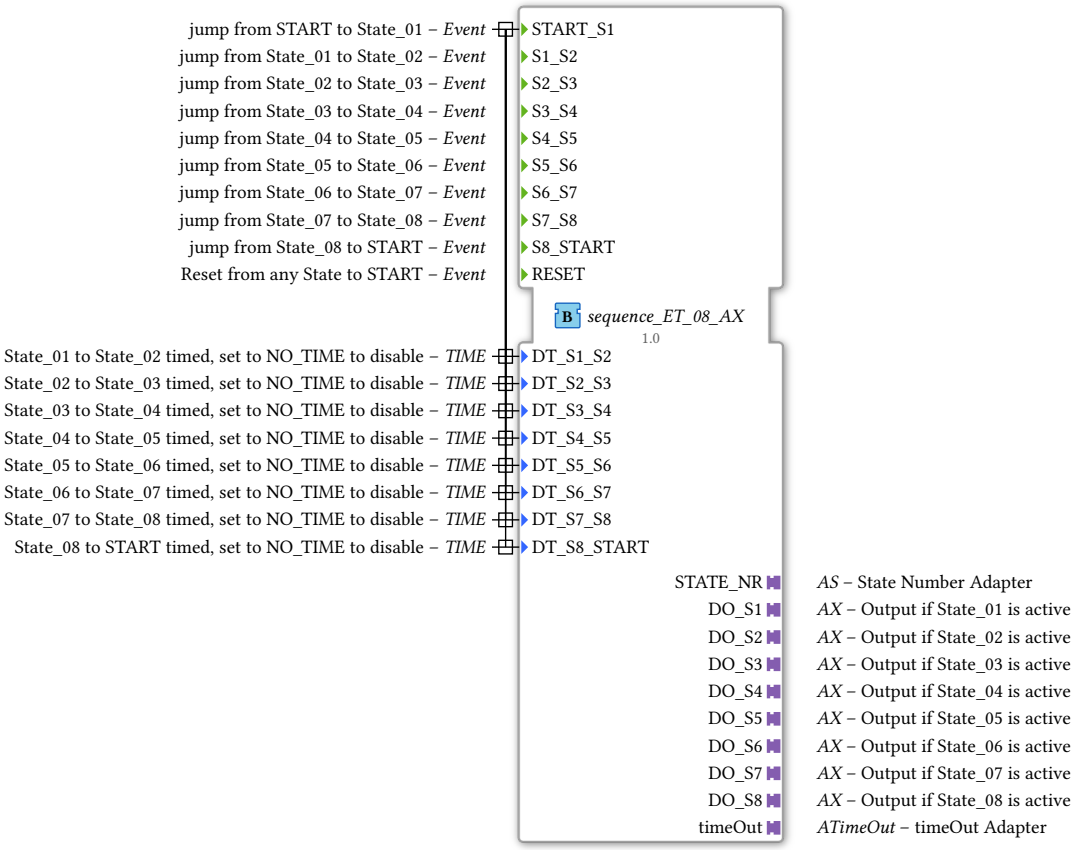
sequence_ET_08_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_ET_08_AXis a variant ofsequence_ET_08 that additionally uses adapters (AX`) for the outputs. It controls a sequence with 8 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2** ... **S7_S8**: Manual transitions between states.
- **S8_START**: Manual transition State_08 -> START.
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirmation of execution.

Data Inputs

- **DT_S1_S2** ... **DT_S8_START**: Times for automatic transitions between states.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Output adapter for State_01 to State_08.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

Corresponds to `sequence_ET_08`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

Status Overview

See `sequence_ET_08`.

Application Scenarios

For 8-step sequences with adapter connection.

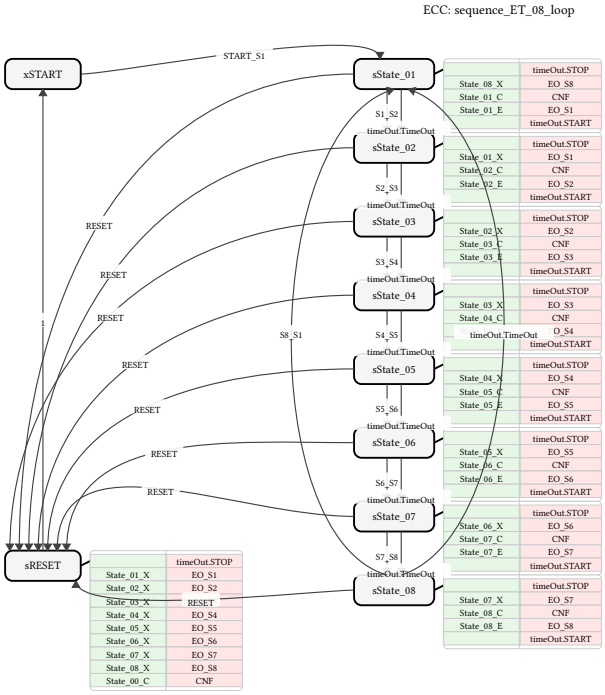
⚖️ Comparison with Similar Components

- **sequence_ET_08**: Standard version without adapter.

Conclusion

Adapter version of the 8-step sequencer.

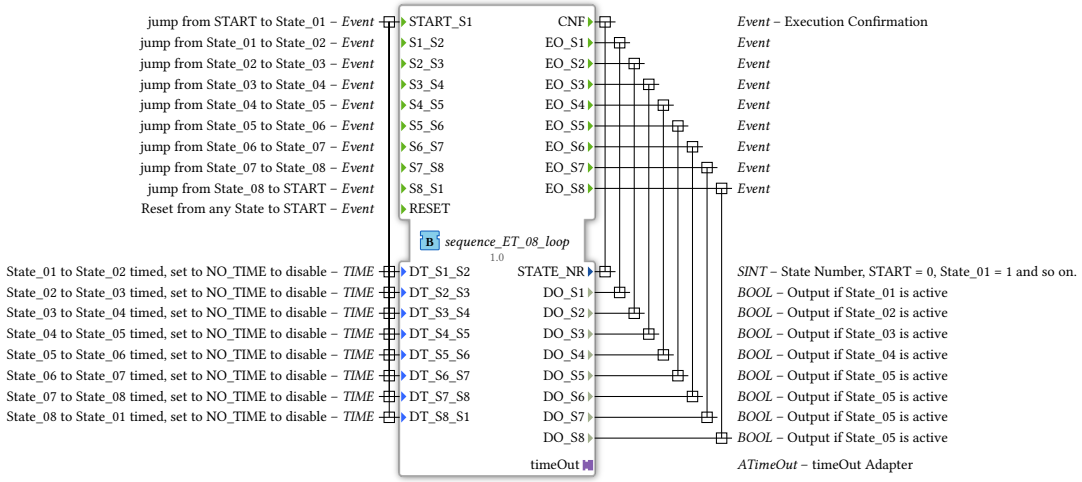
sequence_ET_08_loop



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_ET_08_loop` is a sequencer with eight output states that operates in a loop. It allows transitions between states either triggered by an external event or timed after an adjustable time interval. This block is designed for control tasks where a defined sequence of actions (represented by outputs `DO_S1` to `DO_S8`) must be executed. A key feature is the ability to individually configure each state transition as either event-driven or time-controlled.



Interface Structure

Event Inputs

- **`START_S1`**: Starts the sequence and transitions from state `START` to state `State_01`. Transfers all time data (`DT_S1_S2` to `DT_S8_S1`).
- **`S1_S2 to S8_S1`**: Performs the transition from the corresponding source state to the next target state (e.g., `S1_S2` from `State_01` to `State_02`). These transitions are only effective if timing is disabled for this step.
- **`RESET`**: Resets the sequence from any state back to the initial state `START`.

Event Outputs

- **`CNF`**: General execution confirmation event that is triggered on every state transition. Transmits the current state number `STATE_NR`.
- **`EO_S1 to EO_S8`**: State-specific output events that are triggered upon entering the corresponding state (`State_01` to `State_08`). Each event transmits the corresponding Boolean data output (`DO_Sx`).

Data Inputs

- **`DT_S1_S2 to DT_S8_S1`** (Type: `TIME`): Define the duration for which the respective state remains active before an automatic, time-controlled transition to the next state occurs. To disable the time control for a specific transition and switch to purely event-driven transitions, the value must be set to the constant `NO_TIME`. Initially, all time inputs are preset to `NO_TIME`.

Data Outputs

- **`STATE_NR`** (Type: `SINT`): Outputs the number of the currently active state (START = 0, State_01 = 1, ..., State_08 = 8).
- **`DO_S1 to DO_S8`** (Type: `BOOL`): The physical outputs of the sequence. Each output is set to `TRUE` if the function block is in the corresponding state (`DO_S1` to `State_01`, etc.), otherwise it is `FALSE`.

Adapter

- **`timeOut`** (Type: `ATimeOut`): A plug-in timeout adapter used to implement timed state transitions. The function block (FB) starts and stops the timer and passes it the respective time setting (`DT_Sx_Sy`).

Functionality

The FB is implemented as a Basic Function Block (BFB) with an Execution Control Chart (ECC). The ECC consists of the states `xSTART` (initial state), `sState_01` to `sState_08` (active sequence states), and `sRESET` (reset state).

Upon entering an active state (e.g., `sState_01`), the following actions are executed sequentially:

1. The `timeOut` timer is stopped.
2. The exit algorithm (`X`) of the previous state is executed (setting the previous output `DO_Sx` to `FALSE`).
3. The confirmation algorithm (`C`) of the new state is executed (setting `STATE_NR` and configuring the `timeOut` adapter with the time set for this state, `DT`).

4. The entry algorithm (E) for the new state is executed (setting the corresponding output DO_Sx to TRUE and triggering the event EO_Sx).
5. The timer timeOut is started with the set time.

A state change can be triggered in two ways:

1. **Event-driven:** By the corresponding event Sx_Sy, provided the timer for this transition is disabled (DT = NO_TIME).
2. **Time-Controlled:** By the adapter's TimeOut event, provided a valid time (DT != NO_TIME) is set.

The RESET event leads to the sRESET state, disables all outputs, and confirms the 0 (START) state before the function block (FB) switches back to xSTART.

Technical Features

- **Hybrid Triggering:** Each state transition can be individually configured. This offers maximum flexibility for sequences that are partly sensor-driven and partly time-controlled.
- **Initial Configuration:** By default, all transitions are event-driven (NO_TIME), which requires explicit configuration of the time values to utilize timing.
- **State Confirmation:** The CNF event with STATE_NR enables easy monitoring and visualization of the current sequence position.
- **Closed Loop:** The sequence automatically loops back from State_08 to State_01, allowing for the implementation of cyclic processes.

State Overview

The FB passes through the following states in the ECC:

- **xSTART:** Inactive initial state. Waiting for START_S1.
- **sState_01 to sState_08:** Active operating states. Each state activates its specific output (DO_Sx) and waits for a trigger to transition to the next state.
- **sRESET:** Reset state. Exited from any state upon the RESET event, this state deactivates all outputs and returns to xSTART.

The transition conditions are defined in the ECC and combine the events Sx_Sy, timeOut.TimeOut, and RESET..

Application Scenarios

- **Batch Process Control:** Sequence control for mixing, heating, or filling processes where individual steps have varying durations.
- **Linked Machine Sequences:** Control of a machine whose work cycle consists of several sequentially connected positions or functions (e.g., turning, drilling, milling).
- **Test Stands:** Automated execution of test sequences that combine tests (event-driven) with waiting times (time-driven).
- **Packaging Machines:** Control of the cycle "feed product - close packaging - label - discharge".

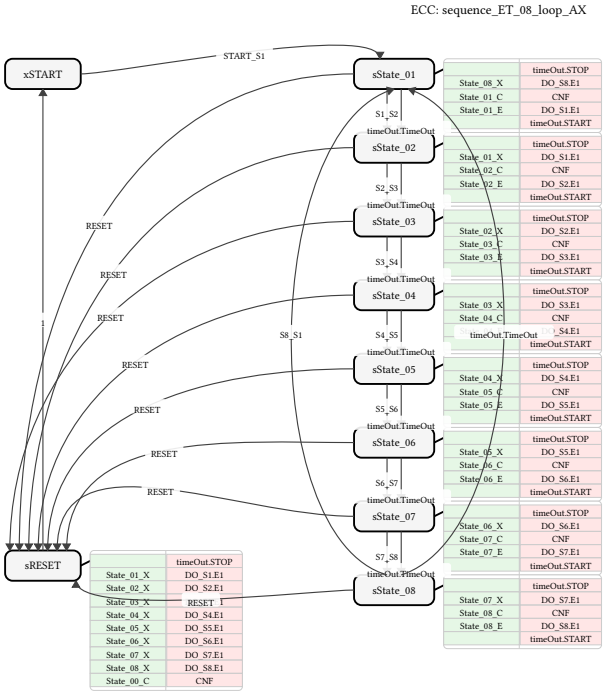
⚖️ Comparison with Similar Function Blocks

Compared to simpler timers or flip-flops, this function block offers a predefined, robust state machine for 8-step sequences. Compared to an individually programmed SFC (Sequential Function Chart) in a service or composite function block, sequence_ET_08_loop is a ready-to-use, tested, and reusable component that accelerates development and reduces the potential for errors. Function blocks with a fixed number of states are often more performant and easier to configure than fully programmable sequencers.

Conclusion

The sequence_ET_08_loop is a powerful and flexible function block for controlling cyclic 8-step sequences. Its strength lies in its hybrid triggering, which is freely selectable for each step, and its clear, event-based interface according to IEC 61499. Thanks to its integrated timing and direct Boolean outputs, it is ideally suited for directly controlling actuators in a higher-level control network.

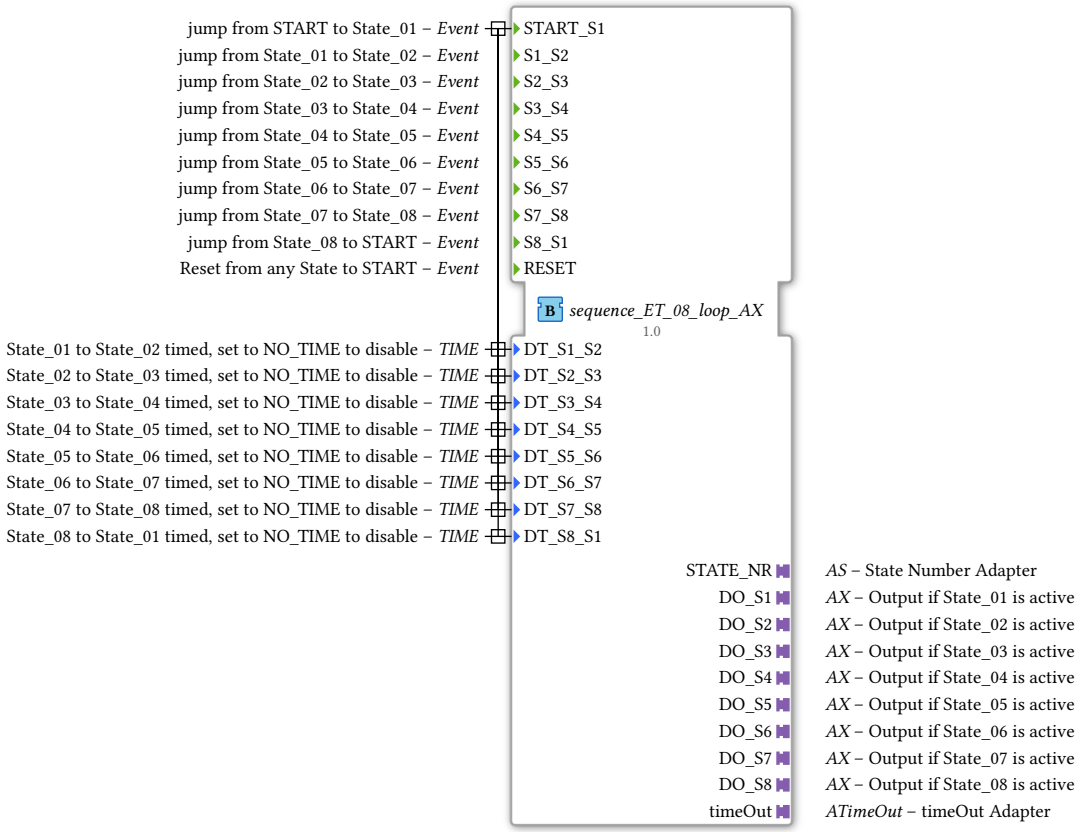
sequence_ET_08_loop_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_ET_08_loop_AXis a variant ofsequence_ET_08_loop that additionally uses adapters (AX`) for the outputs. It controls a cyclic sequence with 8 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2** ... **S7_S8**: Manual transitions between states.
- **S8_S1**: Manual transition State_08 -> State_01 (loop).
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Acknowledgement of execution.

Data Inputs

- **DT_S1_S2** ... **DT_S8_S1**: Times for automatic transitions between states.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Output adapter for State_01 to State_08.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

Corresponds to `sequence_ET_08_loop`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

Status Overview

See `sequence_ET_08_loop`.

Application Scenarios

For cyclic 8-step processes with adapter connection.

⚖️ Comparison with Similar Function Blocks

- **sequence_ET_08_loop**: Standard version without adapter.

Conclusion

Adapter version of the 8-step loop sequencer.

const

anlagenSequenz

• – ◦ • – ◦ • – ◦ •

Introduction

This document describes the global constants defined for the “AnlagenSequenz” (plant sequence) control logic. These constants provide fixed numeric codes for the operating states, fault states, and transition types used in a motor sequence control system, particularly for a plant with six motors.

Interface Structure

Since this is a set of global constants, there are no event inputs, event outputs, data inputs, data outputs, or adapters in the traditional sense. The constants are directly accessible as global constants in the IEC 61499 environment.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

The following constants are available as global constants:

Constant	Type	Value	Description
STATUS_BETRIEB_AUS	SINT	SINT#0	Anlage steht (Plant stopped)
STATUS_BETRIEB_HOCHFAHREN	SINT	SINT#1	Vorlauf-Kette aktiv (Start-up chain active)
STATUS_BETRIEB_LAEUFT	SINT	SINT#2	alle 6 Motoren laufen (All 6 motors running)
STATUS_BETRIEB_HERUNTERFAHREN	SINT	SINT#3	Nachlauf-Kette aktiv (Shutdown chain active)
STATUS_STOERUNG_KEINE	SINT	SINT#0	keine aktive Störung (No active fault)
STATUS_STOERUNG_AKTIV	SINT	SINT#4	Störung aktiv, verriegelt bis EIN (Fault active, interlocked until ENABLE)
TRANSIT_ART_KEINE	SINT	SINT#0	kein Motor aktuell im Vor-/Nachlauf (No motor currently in start-up/shut-down)
TRANSIT_ART_VORLAUF	SINT	SINT#1	TRANSIT_MOTOR startet nach Ablauf der aktuellen Zeit (motor starts after current time expires)

TRANSIT_ART_NACHLAUF	SINT	SINT#2	TRANSIT_MOTOR stoppt nach Ablauf der aktuellen Zeit (motor stops after current time expires)

Adapters

None.

Functionality

The global constants define a standardized set of status codes and transition types used within the sequence control logic. They ensure consistent representation of the plant's operational states and fault conditions across all function blocks and devices in the system. The constants separate the operational status (STATUS_BETRIEB_*) and the fault status (STATUS_STOERUNG_*), allowing clear state handling.

Technical Features

- Constants are defined as SINT (short integer) values.
- All constants are declared as `VAR_GLOBAL CONSTANT`, meaning they are read-only and globally accessible.
- The package name is `logiBUS::utils::sequence::const`.
- The values are chosen to avoid collisions between operational and fault status codes (fault status uses value 4, which is distinct from operational values 0-3).
- The transition types (TRANSIT_ART_*) allow identifying whether a motor is currently in a start-up (Vorlauf) or shutdown (Nachlauf) sequence.

State Overview

While this block does not implement a state machine itself, the constants are designed to be used in state-based control. The operational states progress from `STATUS_BETRIEB_AUS` (0) through `STATUS_BETRIEB_HOCHFAHREN` (1) to `STATUS_BETRIEB_LAEUFT` (2) and then to `STATUS_BETRIEB_HERUNTERFAHREN` (3) when shutting down. A fault is represented by `STATUS_STOERUNG_AKTIV` (4), which acts as an additional state that interlocks the sequence until a new start command (EIN) is given.

Application Scenarios

- **Plant Startup:** When the plant is commanded to start, the sequence moves from `STATUS_BETRIEB_AUS` to `STATUS_BETRIEB_HOCHFAHREN`, activating the start-up chain. The `TRANSIT_ART_VORLAUF` constant indicates that a motor is in its start-up delay.
- **Normal Operation:** After all six motors have started, the status becomes `STATUS_BETRIEB_LAEUFT`.
- **Fault Handling:** If a fault occurs, `STATUS_STOERUNG_AKTIV` is set, and the sequence is interlocked until a reset/start signal.
- **Shutdown:** During shutdown, the sequence passes through `STATUS_BETRIEB_HERUNTERFAHREN` with `TRANSIT_ART_NACHLAUF` for each motor's stop delay.

Comparison with Similar Blocks

Since this is a global constants collection, there is no direct equivalent function block. However, compared to hard-coded values or per-block constants, this approach ensures:

- Consistency across all function blocks using the sequence logic.
- Easy maintenance if status codes need to be changed.
- Clear documentation of the meaning of each numeric value.

Conclusion

The `anlagenSequenz` global constants provide a well-defined set of numeric codes for operating states, fault states, and transition types in a motor sequence control system. By centralizing these definitions, the system becomes more maintainable and less error-prone, ensuring that all components interpret the state information consistently.

motorNummer

• – ◦ • – ◦ • – ◦ •

Introduction

The `motorNummer` global constant container provides a set of predefined constants that represent motor identities in an automation system. It is part of the `logiBUS::utils::sequence::const` package and is used in sequences such as `AnlagenSequenz_06.TRANSIT_MOTOR` to unambiguously identify which motor (or no motor) is currently selected. The constants are defined as 8-bit signed integers (`SINT`) and cover the range from "no motor" to six distinct motors.

Interface Structure

Since `motorNummer` is a global constant container rather than a function block, it does not expose event or data interfaces in the traditional sense. Instead, it defines a fixed set of named constants that can be referenced globally throughout the project. The following sections describe the applicable interface elements.

Event Inputs

Not applicable. The `motorNummer` container defines no event inputs.

Event Outputs

Not applicable. The `motorNummer` container defines no event outputs.

Data Inputs

Not applicable. The `motorNummer` container defines no modifiable data inputs; all values are read-only constants.

Data Outputs

The container defines the following constant data values, each accessible by its symbolic name:

Constant Name	Type	Value	Comment
`MOTOR_KEINER`	`SINT`	0	No motor selected
`MOTOR_M1`	`SINT`	1	Motor 1
`MOTOR_M2`	`SINT`	2	Motor 2
`MOTOR_M3`	`SINT`	3	Motor 3
`MOTOR_M4`	`SINT`	4	Motor 4
`MOTOR_M5`	`SINT`	5	Motor 5
`MOTOR_M6`	`SINT`	6	Motor 6

Adapters

Not applicable. The `motorNummer` container defines no adapters.

Functionality

The primary purpose of `motorNummer` is to provide a centralized, human-readable naming scheme for motor identifiers. By using symbolic constants such as `MOTOR_M3` instead of raw numeric values, the application code becomes more readable, maintainable, and less error-prone. The constants are intended to be used in sequence logic, state machines, or any other part of the control application where a motor identity must be referenced. The special value `MOTOR_KEINER` (0) indicates that no motor is currently involved, enabling explicit handling of "no motor" states in sequences.

Technical Features

- **Data Type**: All constants are of type `SINT` (8-bit signed integer), ensuring compact storage and compatibility with typical PLC data handling.
- **Fixed Values**: The constants are immutable and defined at compile time, preventing accidental modification during runtime.
- **Global Scope**: As global constants, they are accessible from any function block, subapplication, or adapter within the same project without requiring explicit data connections.
- **Package Organization**: The constants reside in the package `logiBUS::utils::sequence::const`, promoting a structured organization of project-wide constants.

- **Documentation Support**: Each constant includes a descriptive comment (e.g., “Motor 1”, “no motor”) to clarify its intended use.

State Overview

Not applicable. `motorNummer` defines static constants and does not maintain any internal state or runtime behavior.

Application Scenarios

The following scenarios are typical use cases for the `motorNummer` constants:

- **Motor Selection in Sequences**: In a production sequence (e.g., `AnlagenSequenz_06.TRANSIT_MOTOR`), the constants determine which motor is to be transported or activated. For example, checking `IF currentMotor = MOTOR_M2 THEN ...`.
- **Initialization**: Setting the motor reference to `MOTOR_KEINER` during system startup to indicate that no motor has been selected yet.
- **Switch/Case Logic**: Using the constants in case or switch statements to dispatch distinct behaviors for each motor.
- **Parameter Passing**: Passing a motor identifier to function blocks or subapplications as a clear, self-documenting argument.

Comparison with Similar Blocks

Unlike function blocks or subapplications, `motorNummer` is a global constants container. It does not execute logic, maintain states, or process events. It is comparable to a named enum or define in traditional programming languages. A block like a motor control FB would use these constants as input parameters, while a subapplication might read them to configure its internal sequencing. The main advantage over hard-coded numeric values is the improved code readability and centralized definition, which simplifies maintenance when motor numbering changes.

Conclusion

The `motorNummer` global constant container is a simple yet essential utility for standardizing motor identification in the automation project. By providing named, typed constants for all motor slots (and the “no motor” case), it enhances code clarity, reduces the risk of numeric errors, and supports consistent behavior across sequences and applications. It is a best-practice example of structuring project-wide constants for reuse.

sequence



• – ◦ • – ◦ • – ◦ •

Introduction

The global constants file `sequence` defines a set of predefined constants that can be used in conjunction with sequence function blocks. It provides symbolic names for states and a special time value to improve the readability and maintainability of IEC 61499 applications. The constants are organized in the package `logiBUS::utils::sequence::const`.

Interface Structure

Global constants do not have event or data interfaces in the conventional sense of a function block. Instead, they are exposed as named values throughout the project. Their structure is a collection of variable declarations with fixed initial values.

Data Inputs

Not applicable. Global constants are not inputs, but rather fixed values.

Data Outputs

Not applicable. Global constants are not outputs, but rather fixed, predefined values.

Adapters

Not applicable.

Functionality

This file defines constants that are replaced by their values at compile time. They can be referenced in any function block within the project, provided the corresponding namespace (`logiBUS::utils::sequence::const`) is imported or the fully qualified name is used. Using these constants instead of direct numeric values ("magic numbers") in the code makes the logic clearer and more resistant to changes.

Technical Specifications

- **Data Types:** The constants use the basic IEC 61131-3 data types `TIME` and `SINT` (short integers).
- **Special Value `NO_TIME`:** The constant `NO_TIME` is initialized with the very large value `TIME#49d17h2m47s295ms`. This value typically serves as a placeholder or initial value to indicate an "undefined" or "unset" time, as it lies far outside the range of normal control timings.
- **State Constants:** The constants `State_00` to `State_08` represent integer states (0-8) and are of type `SINT`. They are intended for use in state machines or step sequences.

State Overview

Not applicable. This is a collection of constants and does not contain a state machine.

Application Scenarios

1. **State Machines / Sequencers:** The `State_XX` constants can be used as return values or for comparisons in step sequence function blocks (e.g., `E_SR`, `E_CTU` in conjunction with logic) to clearly identify the current step.
2. **Timer Initialization:** The constant `NO_TIME` can be used to assign an initial "invalid" value to timer function blocks before a real duration is set.
3. **Code Readability:** Replacing numbers like `SINT#5` in the code with `State_05` improves readability and indicates the semantic meaning of the value.

⚖️ Comparison with Similar Building Blocks

- **ENUM Data Type:** A user-defined enumeration type (ENUM) could fulfill a similar role to the `State_XX` constants and often offers stricter type checking. However, global constants are easier to define and use in 4diac.
- **Direct Value Input:** The main advantage over directly entering values (e.g., `3`) is the centralized definition. A change to the value only needs to be made in one place (here, in the constant file) and will affect all uses.

Conclusion

The global constant file `sequence` is a useful tool for structuring control applications. By providing descriptive names for states and a specific time value, it significantly contributes to the creation of clear, maintainable, and robust code. Their use is particularly recommended for implementing sequenced logic and state-based control systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter Block on ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

zaehlstand

• – ◦ • – ◦ • – ◦ •

Introduction

The `zaehlstand` global constants block defines a set of named constants representing the current number of running motors in an industrial plant sequence. The constants are of type `SINT` (signed 8-bit integer) and cover the range from 0 to 6 motors. The constant names are used for readability and maintainability, replacing hard-coded numeric values in application logic. The block is part of the `logiBUS::utils::sequence::const` package and is intended for use within the 4diac IDE environment.

Interface Structure

The `zaehlstand` block does not contain any executable function block logic – it is a pure constant definition. Therefore, it exposes no event inputs, event outputs, or adapters. The provided values are accessible as global constants within the application.

Event Inputs

None.

Event Outputs

None.

Data Inputs

None.

Data Outputs

The following constants are defined as global data outputs. They hold fixed values and cannot be modified at runtime.

Constant Name	Type	Initial Value	Description
`ZAEHLSTAND_0`	SINT	`SINT#0`	0 motors running (state S_AUS)
`ZAEHLSTAND_1`	SINT	`SINT#1`	1 motor running
`ZAEHLSTAND_2`	SINT	`SINT#2`	2 motors running
`ZAEHLSTAND_3`	SINT	`SINT#3`	3 motors running
`ZAEHLSTAND_4`	SINT	`SINT#4`	4 motors running
`ZAEHLSTAND_5`	SINT	`SINT#5`	5 motors running
`ZAEHLSTAND_6`	SINT	`SINT#6`	6 motors running (state S_LAEUFT, all)

Adapters

None.

Functionality

The block provides a set of symbolic constants that represent the exact number of motors currently active in a plant sequence. By referencing these constants in the application code, the developer can easily compare the current motor count (e.g., from a counter output) with a specific scenario. The constants are defined in a single location, simplifying changes and avoiding magic numbers scattered throughout the project.

Technical Features

- **Type:** All constants are of type `SINT` (signed 8-bit integer).
- **Initial Values:** Each constant is pre-initialised with the corresponding numeric value (`SINT#0` to `SINT#6`).
- **Scope:** Defined as `VAR_GLOBAL CONSTANT` – they are globally accessible and immutable.
- **Package:** Belongs to the `logiBUS::utils::sequence::const` package, which groups related sequence constants.
- **Licensing:** Distributed under Eclipse Public License 2.0.

State Overview

The constants represent discrete states of the motor count:

- `ZAEHLSTAND_0` – all motors off (S_AUS)
- `ZAEHLSTAND_6` – all motors running (S_LAEUFT)
- Intermediate values represent partially active motor sets.

These constants are typically used in sequence control logic to trigger actions or transitions when a certain motor count is reached.

Application Scenarios

- **Sequence Control:** Determine when to start or stop additional motors based on the current count.
- **Monitoring:** Compare the actual motor count (e.g., from a sensor or counter) with the expected count.
- **Interlocking:** Enable or disable processes when the number of running motors reaches a critical threshold.
- **Testing:** Use the constants as test inputs for simulation or validation of motor-counting logic.

Comparison with Similar Blocks

Unlike function blocks (FBs) or adapters, `zaehlstand` is a **constant definition** – it does not perform any computation or communication. It serves a similar purpose as an enumeration, but with explicit numeric values. Compared to using raw integer literals, the constants improve code readability and reduce the risk of typographical errors. Other possible implementations might use an enumerated type or a structured variable, but the use of simple SINT constants keeps the approach lightweight and directly compatible with IEC 61499 arithmetic operations.

Conclusion

`zaehlstand` is a compact and well-defined set of global constants for representing the number of running motors in a plant sequence. It provides clear, mnemonic names for the values 0 through 6, promoting maintainable and error-free application code. The block is an essential building block for sequence logic within the 4diac environment.

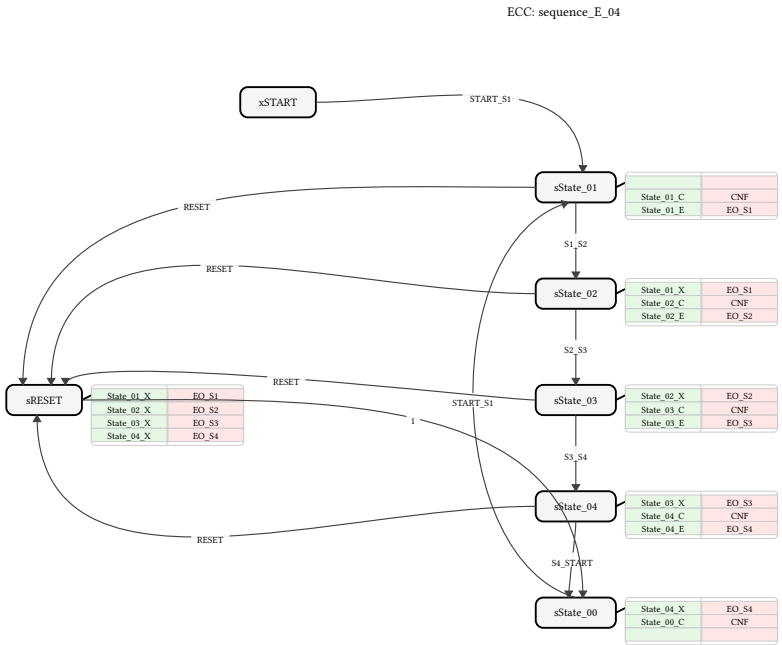
event

🎧 Podcast

- E_TOF vs. FB_TOF: The event timer that doesn't tick cyclically – a revolution for automation systems?
- Event FBs according to IEC 61499: Lego bricks of automation – how event function blocks work
- IEC 61499: Revolution of automation – event-driven FBs and distributed systems explained
- E_REND: Event Synchronization in IEC 61499
- The Unstoppable Counter: Why IEC 61499's ECTU Guarantees Safe, Event-Driven Control (and Never Overflows)

---

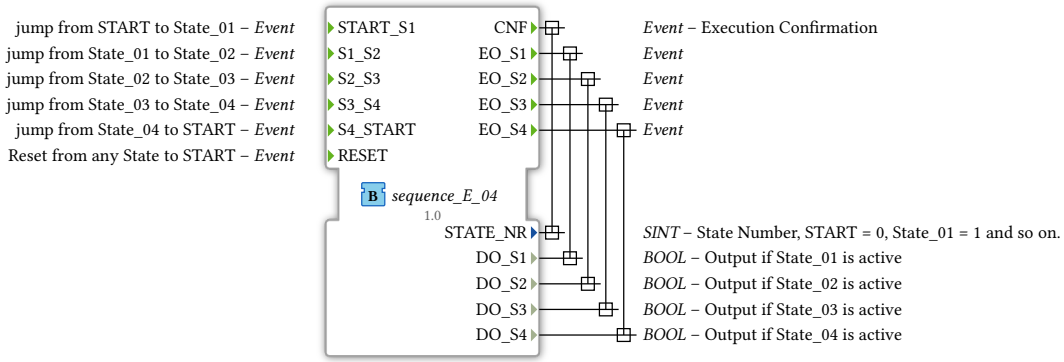
sequence_E_04



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_E_04` is a sequencer that implements a linear sequence of four states (State_01 to State_04) with a defined start state (START) and an end state (State_00). The transition between the individual states occurs exclusively through external events. The block is designed for control tasks where step-by-step, event-driven execution is required, such as in simple process or assembly sequences.



Interface Structure

Event Inputs

- **START_S1**: Switches from the START state or State_00 to the State_01 state.
- **S1_S2**: Changes from state State_01 to state State_02.
- **S2_S3**: Changes from state State_02 to state State_03.
- **S3_S4**: Changes from state State_03 to state State_04.
- **S4_START**: Changes from state State_04 to state State_00.
- **RESET**: Resets the function block from any state (State_01 to State_04) to state State_00.

Event Outputs

- **CNF**: Acknowledge event triggered on every state change. It returns the current state number via `STATE_NR`.
- **EO_S1**: Triggered upon entering state State_01 and returns the value of `DO_S1` (TRUE). * **EO_S2**: Triggered upon entering state_02 and returns the value `DO_S2` (TRUE).
- **EO_S3**: Triggered upon entering state_03 and returns the value `DO_S3` (TRUE).
- **EO_S4**: Triggered upon entering state_04 and returns the value `DO_S4` (TRUE).

Data Inputs

- None available.

Data Outputs

- **STATE_NR** (SINT): Outputs the number of the current state. The encoding is: START = 0, State_01 = 1, State_02 = 2, State_03 = 3, State_04 = 4.
- **DO_S1** (BOOL): Is TRUE when state State_01 is active.
- **DO_S2** (BOOL): Is TRUE when state State_02 is active.
- **DO_S3** (BOOL): Is TRUE when state State_03 is active.
- **DO_S4** (BOOL): Is TRUE when state State_04 is active.

Adapter

- None present.

Functionality

The `sequence_E_04` is implemented as a Basic Function Block (BFB) and has an Execution Control Chart (ECC). The ECC defines the states and the event-driven transitions between them. Specific algorithms are executed during each state transition:

1. **Exit Algorithm (X)**: Executed when exiting a state to set the corresponding data output (`DO_Sx`) to FALSE.

2. **Entry Algorithm (E)**: Executed when entering a state to set the corresponding data output (`DO_Sx`) to TRUE and trigger the corresponding event (`EO_Sx`).
3. **Confirmation Algorithm (C)**: Executed in every state (except RESET) to update the state number (`STATE_NR`) and trigger the confirmation event (`CNF`).

An event `RESET` executes all necessary exit algorithms for the active states and transitions via an intermediate state (`sRESET`) to the final state `State_00`.

Technical Features

- **Event-based Transition**: State changes are only possible through external events. There are no time- or data-driven transitions.
- **Explicit State Encoding**: The state numbers are defined as constants from the library `sequence`, which improves code reusability and readability.
- **Clean Reset**: The reset process deactivates all active outputs before the final state is reached to ensure clear and defined system behavior.
- **Initial State**: The function block starts in state `xSTART`. The first transition to the operational sequence occurs with the event `START_S1`.

State Overview

1. **xSTART**: Initial resting state.
2. **sState_01**: First active step. `DO_S1` = TRUE.
3. **sState_02**: Second active step. `DO_S2` = TRUE.
4. **sState_03**: Third active step. `DO_S3` = TRUE.
5. **sState_04**: Fourth active step. `DO_S4` = TRUE.
6. **sState_00**: Final resting state after completion of the sequence. All `DO_Sx` = FALSE.
7. **sRESET**: Intermediate state during a reset operation. Deactivates all outputs.

Application Scenarios

- **Step-by-Step Controls**: Control of machines or systems with a fixed, step-by-step workflow (e.g., pick-and-place, filling systems).
- **Process Timing**: Synchronization of subprocesses where each step is triggered manually or by a sensor signal.
- **Manual Operating Sequences**: Implementation of guided operating sequences where the operator must confirm each step.
-

⚖️ Comparison with Similar Function Blocks

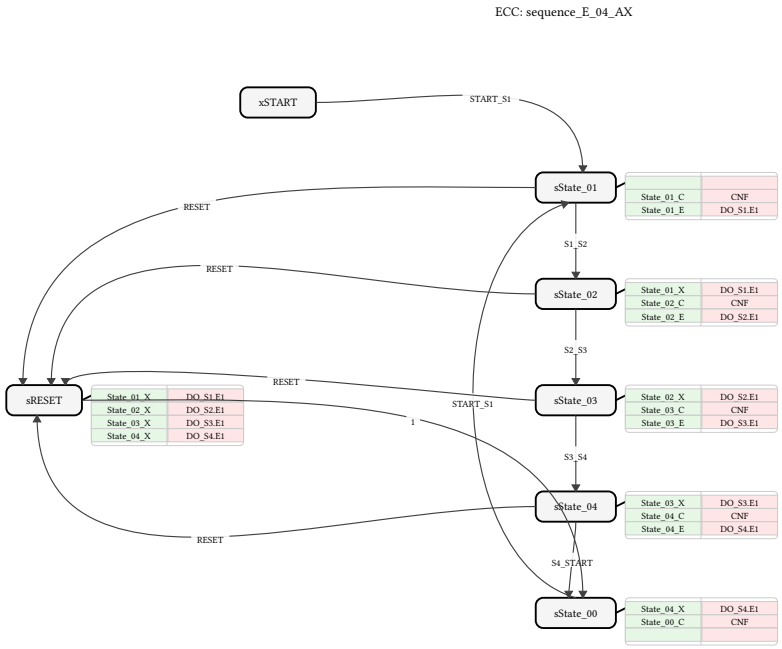
Unlike a `E_CYCLE` or `E_DELAY` function block, `sequence_E_04` offers purely event-driven transitions, not time-controlled ones. Compared to a generic `E_SR` (flip-flop) or a combination of `E_D_FF` (D flip-flop), this function block implements a predefined state machine with multiple steps and clear reset logic. It is more specialized and structured than an ad-hoc implementation using multiple individual function blocks.

...

Conclusion

The `sequence_E_04` is a robust and easy-to-use sequencer for IEC 61499. Its clear, event-driven interface and explicit state management make it ideal for applications requiring deterministic, step-by-step execution. The integrated reset functionality and state feedback via `CNF` and `STATE_NR` support secure and well-monitored integration into higher-level controllers.

sequence_E_04_AX

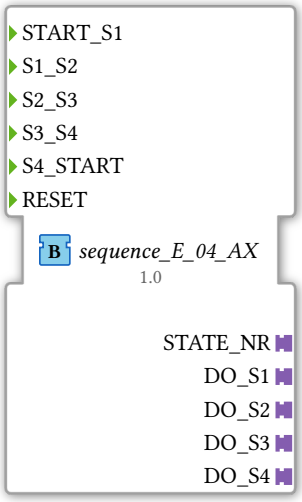


• – ○ • – ○ • – ○ •

Introduction

The `sequence_E_04_AX` is a variant of the `sequence_E_04`, which additionally uses adapters (`AX`) for the outputs. It controls a purely event-driven sequence with 4 output states.

jump from START to State_01 – *Event*
jump from State_01 to State_02 – *Event*
jump from State_02 to State_03 – *Event*
jump from State_03 to State_04 – *Event*
jump from State_04 to START – *Event*
Reset from any State to START – *Event*



AS – State Number Adapter
AX – Output if State_01 is active
AX – Output if State_02 is active
AX – Output if State_03 is active
AX – Output if State_04 is active

Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2**: Transition State_01 -> State_02.
- **S2_S3**: Transition State_02 -> State_03.
- **S3_S4**: Transition State_03 -> State_04.
- **S4_START**: Transition State_04 -> START.
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirmation of execution.

Data Inputs

- None.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.

Functionality

Corresponds to `sequence_E_04`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_E_04`.

Application Scenarios

For event-driven sequences controlled via adapters.

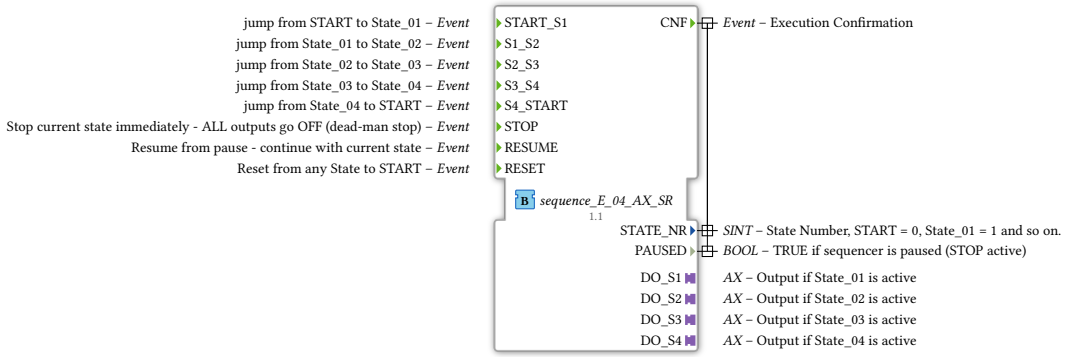
⚖️ Comparison with Similar Function Blocks

- **sequence_E_04**: Standard version without adapters.

Conclusion

Adapter version of the 4-step event sequencer.

sequence_E_04_AX_SR





• – ○ • – ○ • – ○ •

Introduction

The function block `sequence_E_04_AX_SR` implements an event-driven sequencer with four outputs via an AX adapter. It also offers a safety stop (STOP), a resume (RESUME), and a reset (RESET). The sequence cycles through the states State_00, State_01, State_02, State_03, and State_04 and can be operated cyclically.

Interface Structure

Event Inputs

Event	Description
`START_S1`	Transition from START/State_00 to State_01
`S1_S2`	Transition from State_01 to State_02
S2_S3`	Transition from State_02 to State_03
S3_S4`	Transition from State_03 to State_04
S4_START`	Transition from State_04 back to State_00
STOP`	Immediately interrupts the current state – all outputs are switched off (dead man stop)
RESUME`	Resumes the sequence from the paused state (outputs are reactivated)
RESET`	Resets the sequence from any state back to the START state (State_00)

Event Outputs

Event	Description
`CNF`	Execution confirmation. Included output data: `STATE_NR`, `PAUSED`

Data Inputs

No external data inputs.

Data Outputs

Variable	Type	Description
STATE_NR	SINT	Current state number: START = 0, State_01 = 1, ..., State_04 = 4
PAUSED	BOOL	TRUE, when the sequencer is paused (STOP active)

Adapter

Adapter	Type	Description
DO_S1	adapter::types::unidirectional::AX	Output active in State_01 (D1 = TRUE)
DO_S2	adapter::types::unidirectional::AX	Output active in State_02
DO_S3	adapter::types::unidirectional::AX	Output active in State_03
DO_S4`	adapter::types::unidirectional::AX	Output active in State_04

Functionality

The sequencer operates according to a finite state machine (ECM). The five sequential states are:

- State_00 (START)
- State_01
- State_02
- State_03
- State_04

The normal process begins with START_S1 (either from xSTART or from State_00), which then transitions to State_01. At each step, the FB activates the corresponding adapter output (e.g., DO_S1.D1 = TRUE in State_01), and the current values of STATE_NR and PAUSED are transmitted via the event output CNF. The transition to the next state occurs via the corresponding events (S1_S2, S2_S3, S3_S4), and the transition from State_04 back to State_00 with S4_START.

The transition to the next state is triggered by the corresponding events (S1_S2, S2_S3, S3_S4), and from State_04 back to State_00 with S4_START.

... Upon receiving STOP, the function block immediately deactivates the current output (exit step) and stores the current state in the internal variable savedState. The state then transitions to one of the paused states (sPAUSED_S1 to sPAUSED_S4 or sPAUSED_S0). The output PAUSED is set to TRUE. A RESUME event restores the stored state and reactivates the corresponding output.

The state then changes to one of the paused states (sPAUSED_S1 to sPAUSED_S4 or sPAUSED_S0). The RESET command puts the sequencer into a reset state, regardless of its current state. In this state, all four outputs are explicitly switched off, and then it immediately switches to State_00.

Technical Features

- **AX Adapters**: The four outputs are implemented using unidirectional AX adapters, each with a Boolean value, D1. The outputs are only set in active states and are immediately reset upon exiting or stopping.
- **Pause/Resume Mechanism**: The internal variable savedState stores the state at the time of the stop, so that the exact same state can be resumed after RESUME.
- **Safety Stop**: The STOP event causes all outputs to be switched off immediately (even without entering a paused state), which is designed as a dead-man stop.
- **Use of constants from the sequence package**: The state numbers (State_00, State_01, ...) are defined as named constants in the logiBUS::utils::sequence::const::sequence package.
- **Reset Behavior**: The RESET event switches off all four outputs before returning to State_00, thus ensuring a defined output state.

State Overview

State	Active Output	STATE_NR	PAUSED
xSTART	none	0	FALSE
sState_00	none	0	FALSE

sState_01	DO_S1	1	FALSE
sState_02	DO_S2	2	FALSE
sState_03	DO_S3	3	FALSE
sState_04	DO_S4	4	FALSE
sPAUSED_S0	none	saved	TRUE
sPAUSED_S1	none	saved	TRUE
sPAUSED_S2	none	saved	TRUE
sPAUSED_S3	none	saved	TRUE
sPAUSED_S4	none	saved	TRUE
sRESET	none	–	–

Application Scenarios

- **Control of Sequential Processes**: e.g., a machine that performs four work steps in succession, with each step controlling its own actuator.
- **Safety-Critical Applications**: When a `STOP` signal must immediately shut down all actuators (e.g., emergency stop), and the exact state is restored after release.
- **Cyclic Processes**: from step 1 to 4 and back to step 0 (e.g., in packaging machines).
- **Manual Intervention**: the process can be reset to the beginning at any time using `RESET`.

Comparison with Similar Function Blocks

Unlike simpler sequencers (e.g., without STOP/RESUME), `sequence_E_04_AX_SR` offers:

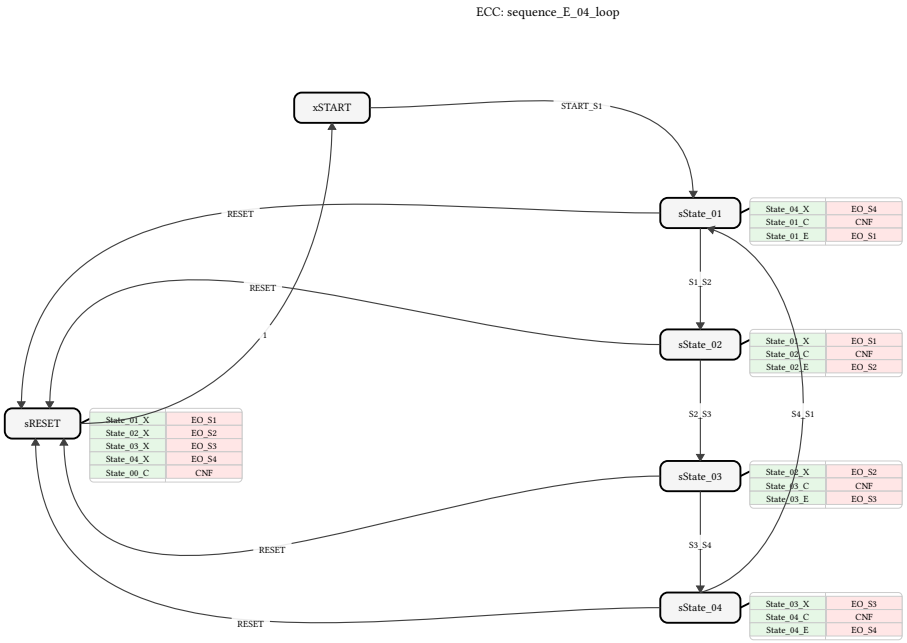
- **Safety interrupt with defined shutdown of all outputs**.
- **Pause and resume function** that saves the current state.
- **Cyclic return capability** from step 4 to step 0.
- **Output of the current state** as a number and pause status.

Other function blocks with the same number of outputs, but without STOP/RESUME, are easier to use but do not offer a safety function.

Conclusion

The `sequence_E_04_AX_SR` function blockis a versatile, safety-conscious four-step sequencer. It is particularly suitable for controllers where an interruptible sequence with defined output states and a resume option is required. The interface is clearly structured, the implementation robust, and it integrates well into automation environments through the use of AX adapters.

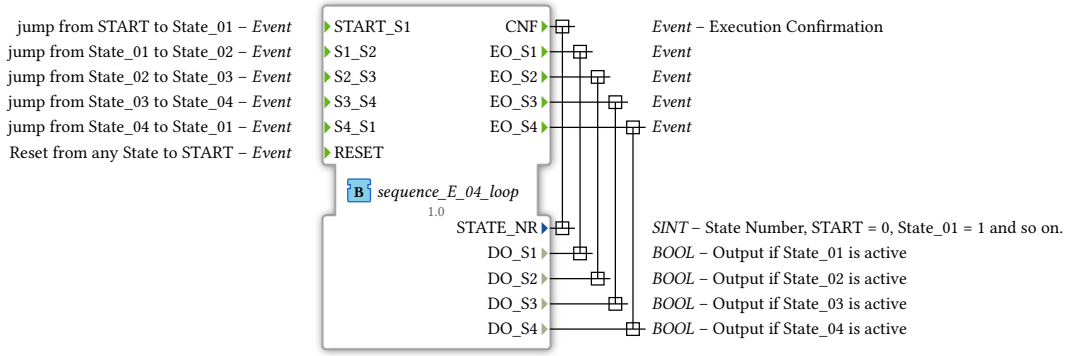
sequence_E_04_loop



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_E_04_loop` implements a cyclic sequence with four states. State transitions are triggered exclusively by external events. This block is designed for control tasks where a fixed sequence of actions (represented by outputs `DO_S1` to `DO_S4`) must be executed and a reset from any state is required.



Interface Structure

Event Inputs

- **`START_S1`**: Changes from the initial state `START` to the first active state `State_01`.
- **`S1_S2`**: Changes from `State_01` to `State_02`.
- **`S2_S3`**: Changes from `State_02` to `State_03`.
- **`S3_S4`**: Switches from `State_03` to `State_04`.
- **`S4_S1`**: Switches from `State_04` back to `State_01`, thus closing the cycle.
- **`RESET`**: Resets the function block from any active state (`State_01` to `State_04`) to the initial state `START`.

Event Outputs

- **CNF**: Triggered on every state change and confirms execution. Transports the current status code for STATE_NR.
- **EO_S1**: Triggered upon entering State_01. Transports the value TRUE for DO_S1.
- **EO_S2**: Triggered upon entering State_02. Transports the value TRUE for DO_S2.
- **EO_S3**: Triggered upon entering State_03. Transports the value TRUE for DO_S3.
- **EO_S4**: Triggered upon entering State_04. Transports the value TRUE for DO_S4.

Data Inputs

- This function block has no data inputs.

Data Outputs

- **STATE_NR** (SINT): Outputs the number of the current state. The encoding is: START = 0, State_01 = 1, State_02 = 2, State_03 = 3, State_04 = 4.
- **DO_S1** (BOOL): Is TRUE when state State_01 is active.
- **DO_S2** (BOOL): Is TRUE when state State_02 is active.
- **DO_S3** (BOOL): Is TRUE when state State_03 is active.
- **DO_S4** (BOOL): Is TRUE when state State_04 is active.

Adapters

- This function block does not use any adapters.

Functionality

The block is implemented as a Basic Function Block (BFB) with an Execution Control Chart (ECC). The internal logic is based on six states: an initial state (xSTART), four active operating states (sState_01 to sState_04), and a special reset state (sRESET).

During a state transition, three actions are executed sequentially:

1. **Exit Algorithm (X)**: The output of the previous state is set to FALSE.
2. **Confirmation Algorithm (C)**: The state number STATE_NR is updated, and the CNF event is triggered.
3. **Entry Algorithm (E)**: The output of the new state is set to TRUE, and the corresponding event (EO_Sx) is triggered.

A RESET event always leads to the sRESET state, where all outputs (DO_S1 to DO_S4) are deactivated, the state number is set to 0 (START), and a CNF is sent. The function block then automatically returns to the initial xSTART state.

Technical Features

- **Event-driven transitions**: Unlike time- or condition-driven sequencers, all state transitions occur only upon the occurrence of the specific event. There are no automatic or time-controlled advances.
- **Explicit Reset Logic**: The reset process is modeled as a separate state (sRESET) that cleanly resets all outputs before the initial state is reached again.
- **State Encoding**: Using the constants from sequence::State_xx for assignment to STATE_NR improves the maintainability and readability of the code.

State Overview

1. **xSTART**: Initial, inactive state. All outputs are FALSE, STATE_NR is 0.
2. **sState_01**: First active state. DO_S1 = TRUE, STATE_NR = 1.
3. **sState_02**: Second active state. DO_S2 = TRUE, STATE_NR = 2.
4. **sState_03**: Third active state. DO_S3 = TRUE, STATE_NR = 3.
5. **sState_04**: Fourth active state. DO_S4 = TRUE, STATE_NR = 4.
6. **sRESET**: Temporary reset state. Sets all actuator outputs (DO_Sx) to FALSE and STATE_NR to 0.

The permitted transitions are: START -> S1 -> S2 -> S3 -> S4 -> (S1) and from any state S1-S4 via RESET back to START.

Application Scenarios

- **Step-by-Step Controls**: Control of machine processes where each step must be enabled manually or by a sensor signal (e.g., manual assembly stations).
- **Clock-Controlled Processes**: In production lines where a central clock signal (Sx_Sy events) signals the progress of the assembly from station to station.

- **Test and commissioning sequences**: A structured sequence of tests that must be confirmed by the operator.

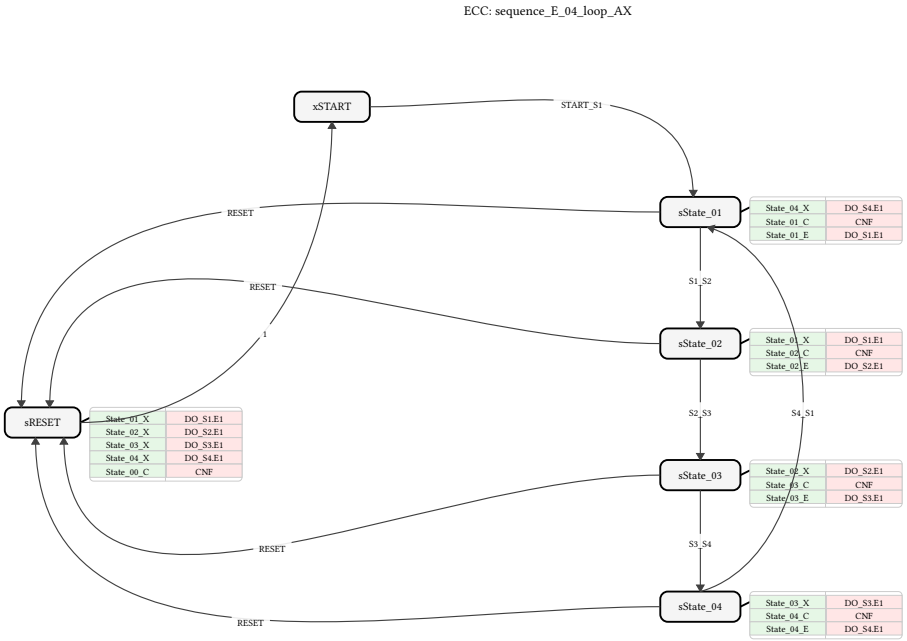
⚖️ Comparison with similar modules

Compared to a **Cyclic sequencer with timer** (e.g., `E_CYCLE`), this module lacks integrated timing; transitions are purely event-driven. Compared to a **Binary shift register** or counter, this module offers an explicit, easily understandable state machine with clear entry/exit actions and a dedicated reset path, simplifying troubleshooting.

Conclusion

The `sequence_E_04_loop` is a robust and clearly structured function block for implementing an event-driven 4-state sequence. Its strengths lie in the explicit modeling of each state transition, the clean reset logic, and the clear separation of entry, exit, and confirmation actions. It is ideally suited for applications where the sequence control is to be carried out by external signals (buttons, sensors, higher-level controllers).

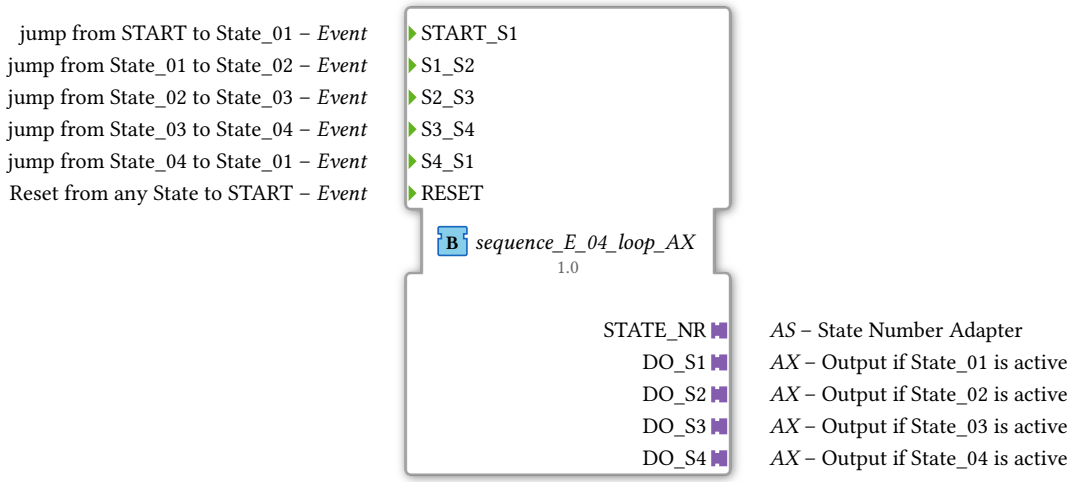
sequence_E_04_loop_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_E_04_loop_AXis a variant ofsequence_E_04_loop that additionally uses adapters (AX`) for the outputs. It controls a purely event-driven, cyclic sequence with 4 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2**: Transition State_01 -> State_02.
- **S2_S3**: Transition State_02 -> State_03.
- **S3_S4**: Transition State_03 -> State_04.
- **S4_S1**: Transition State_04 -> State_01 (Loop).
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Acknowledges execution.

Data Inputs

- None.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.

Functionality

Corresponds to `sequence_E_04_loop`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_E_04_loop`.

Application Scenarios

For cyclic event-driven sequences with adapter connectivity.

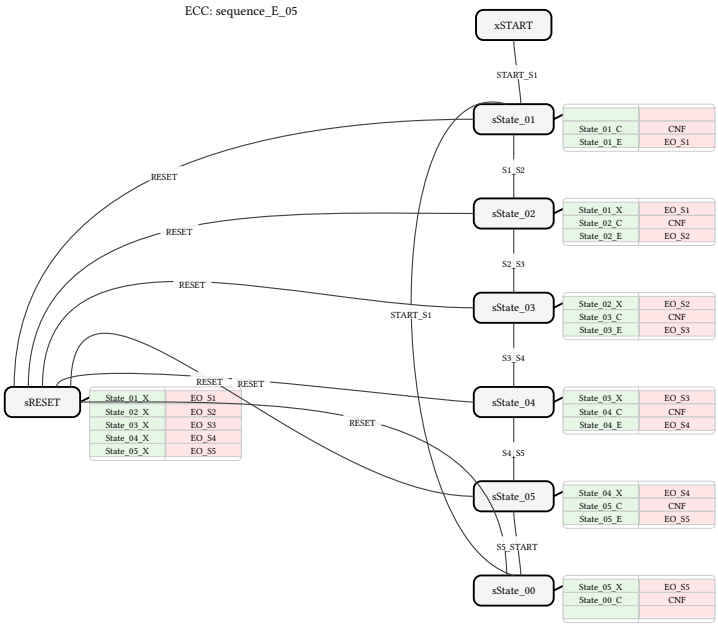
⚖️ Comparison with Similar Function Blocks

- **sequence_E_04_loop**: Standard version without adapters.

Conclusion

Adapter version of the 4-step loop event sequencer.

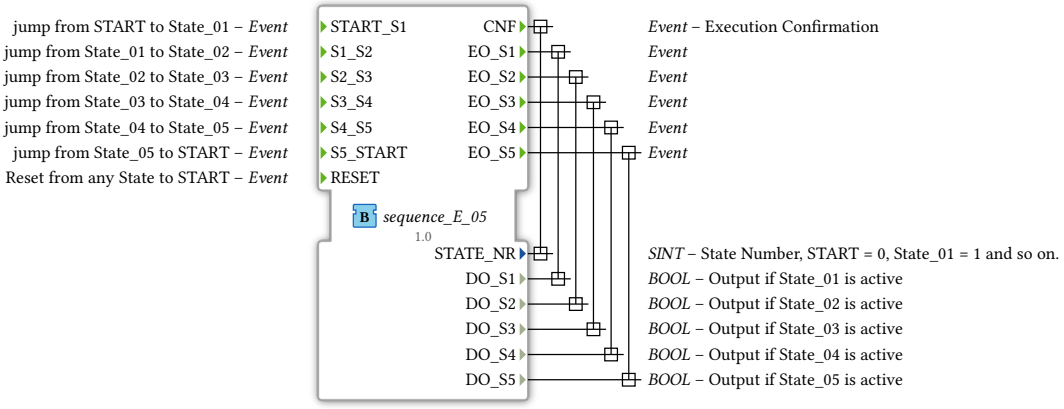
sequence_E_05



• – ○ • – ○ • – ○ •

Introduction

The function block `sequence_E_05` is a sequential state machine that cycles through a fixed sequence of five states. The transition between individual states occurs exclusively through external events. This block is designed for control tasks requiring the step-by-step execution of a process sequence, such as in handling or packaging systems. Each active state sets its own binary output and confirms execution.



Interface Structure

The function block has a pure event-based interface. State transitions are triggered by input events, and the active state is signaled via output events and data outputs.

Event Inputs

- `START_S1`: Changes from the initial START state or from state `sState_00` to the first state `State_01`.
- `S1_S2`: Changes from `State_01` to `State_02`.
- `S2_S3`: Changes from `State_02` to `State_03`.

- S3_S4: Changes from State_03 to State_04.
- S4_S5: Changes from State_04 to State_05.
- S5_START: Changes from State_05 back to the inactive state sState_00.
- RESET: Immediately resets the automaton from any state to the inactive state sState_00.

Event Outputs

- CNF: General confirmation event. Triggered on every state change and returns the current state number STATE_NR.
- EO_S1: Triggered upon entry into State_01 and returns the value DO_S1.
- EO_S2: Triggered upon entry into State_02 and returns the value DO_S2.
- EO_S3: Triggered upon entry into State_03 and returns the value DO_S3.
- EO_S4: Triggered upon entry into State_04 and returns the value DO_S4.
- EO_S5: Triggered upon entering State_05 and returns the value DO_S5.

Data Inputs

There are no data inputs.

Data Outputs

- STATE_NR (SINT): Numeric identifier of the current state. START = 0, State_01 = 1, State_02 = 2, ..., State_05 = 5.
- DO_S1 (BOOL): Is TRUE when state State_01 is active.
- STATE_NR (SINT): Numeric identifier of the current state. START = 0, State_01 = 1, State_02 = 2, ..., State_05 = 5.
- DO_S1 (BOOL): Is TRUE when state State_01 is active. * DO_S2 (BOOL): Is TRUE when state State_02 is active.
- DO_S3 (BOOL): Is TRUE when state State_03 is active.
- DO_S4 (BOOL): Is TRUE when state State_04 is active.
- DO_S5 (BOOL): Is TRUE when state State_05 is active.

Adapter

No adapter interfaces are available.

Functionality

The FB is implemented as a Basic Function Block (BFB) with an Execution Control Chart (ECC). The ECC consists of seven states: an initial state (xSTART), five active states (sState_01 to sState_05), an inactive end state (sState_00), and a special reset state (sRESET).

... Upon entering an active state (e.g., sState_01), two algorithms are executed sequentially:

1. A Confirmation Step (e.g., State_01_C) that sets STATE_NR to the corresponding constant and triggers the CNF event.
2. An Entry Step (e.g., State_01_E) that sets the associated binary output (e.g., DO_S1) to TRUE and triggers the corresponding event (e.g., EO_S1).

When exiting an active state, an *Exit Step* (e.g., State_01_X) is executed, which resets the corresponding binary output to FALSE. State transitions are strictly controlled by the incoming event inputs. An event RESET leads, via the intermediate state sRESET, to the deactivation of all active outputs (DO_S1 to DO_S5) before the automaton transitions to the inactive state sState_00.

Technical Features

- **Event-driven Transitions:** Unlike time- or condition-driven sequencers, state changes here occur exclusively through external events. This enables close coupling to other process steps or operator actions.
- **Explicit Reset Logic:** The reset process is modeled as a separate ECC state (sRESET), ensuring that all five outputs (DO_S1 to DO_S5) are properly deactivated during a reset, regardless of their current state.
- **Separate Entry/Exit Actions:** The logic for setting and resetting the outputs is divided into separate algorithms (E for Entry, X for Exit). This promotes a clear and maintainable structure.
- **Constants for State Numbers:** The values for STATE_NR are retrieved from an imported library (sequence::State_01, etc.), facilitating reusability and centralized maintenance.
-

State Overview

1. **xSTART:** Initial state after startup. Waiting for `START_S1`.
2. **sState_01:** First active step. Sets `DO_S1=TRUE` and `STATE_NR=1`. Waits for `S1_S2` or `RESET`.
3. **sState_02:** Second active step. Sets `DO_S2=TRUE` and `STATE_NR=2`. Waits for `S2_S3` or `RESET`.
4. **sState_03:** Third active step. Sets `DO_S3=TRUE` and `STATE_NR=3`. Waits for `S3_S4` or `RESET`.
5. **sState_04:** Fourth active step. Sets `DO_S4=TRUE` and `STATE_NR=4`. Waits for `S4_S5` or `RESET`.
6. **sState_05:** Fifth active step. Sets `DO_S5=TRUE` and `STATE_NR=5`. Waiting for `S5_START` or `RESET`.
7. **sState_00:** Inactive end state. All outputs are `FALSE` or `STATE_NR=0`. Waiting for `START_S1` for a new cycle.
8. **sRESET:** Temporary reset state. Disables all active outputs and then switches to `sState_00`..

Application Scenarios

- **Step Sequence in Handling Devices:** Control of a pick-and-place robot (grasping → lifting → moving → lowering → releasing), where each step is triggered by a sensor event (e.g., "part detected," "position reached").
- **Manual Operating Sequences:** Execution of a sequence specified by the operator, e.g., in a machine setup ("Release step 1" → "Release step 2").
- **Synchronization with Higher-Level Controllers:** The sequence serves as a subroutine of a main controller, which dictates the progress via the events.

⚖️ Comparison with Similar Function Blocks

Unlike a cyclically running sequencer (e.g., `CYCLE_5`), which automatically advances after a fixed time, `sequence_E_05` remains in each state until the corresponding next event occurs. This makes it more deterministic with respect to external conditions, but requires reliable event generation by the peripherals or higher-level logic. It is simpler and more transparent than a complete sequential function block (SFC) for fixed, linear sequences.

Conclusion

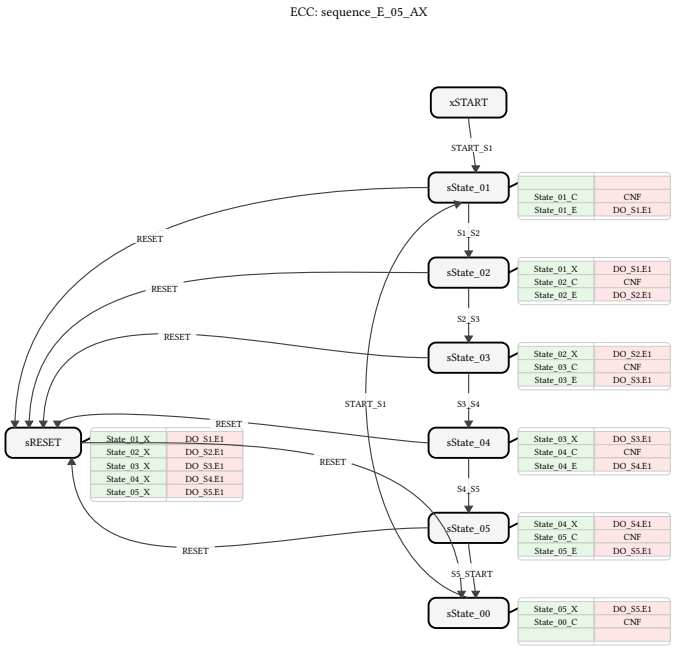
The `sequence_E_05` is a robust and easy-to-configure function block for event-driven, fixed-length sequences. Its clear separation of state logic and output actions, as well as the integrated reset mechanism, make it a reliable component for sequential control tasks in 4diac-based automation systems. The lack of flexibility in the number of steps or transition conditions is an advantage for defined, linear processes in terms of clarity and predictability.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

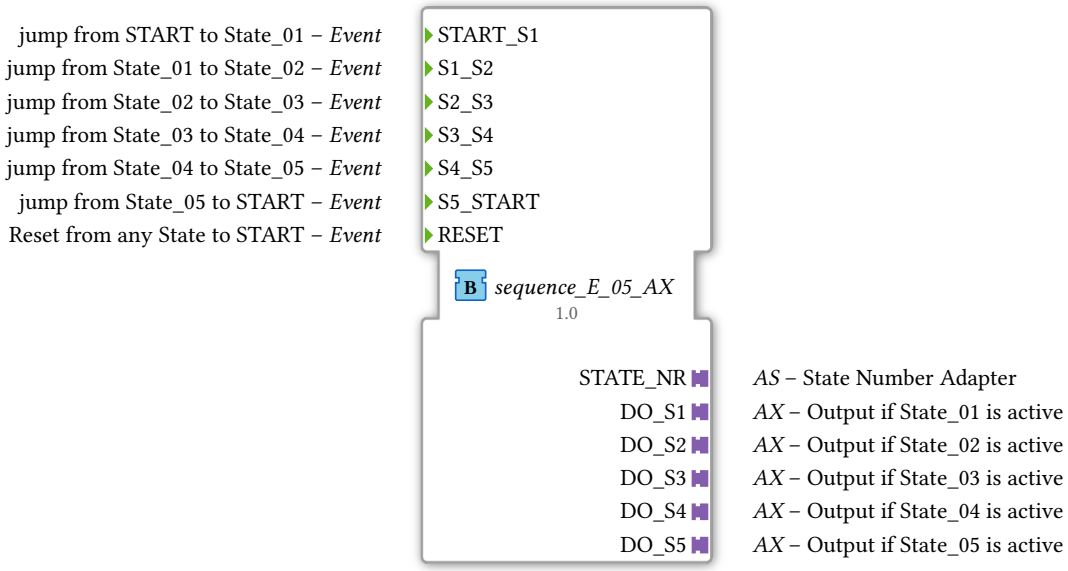
sequence_E_05_AX



• – ○ • – ○ • – ○ •

Introduction

sequence_E_05_AXis a variant ofsequence_E_05 that additionally uses adapters (AX`) for the outputs. It controls a purely event-driven sequence with 5 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2**: Transition State_01 -> State_02.
- **S2_S3**: Transition State_02 -> State_03.
- **S3_S4**: Transition State_03 -> State_04.

- **S4_S5**: Transition State_04 -> State_05.
- **S5_START**: Transition State_05 -> START.
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirmation of execution.

Data Inputs

- None.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Output adapter for State_05.

Functionality

Corresponds to `sequence_E_05`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_E_05`.

Application Scenarios

For event-driven 5-step sequences with adapter connectivity.

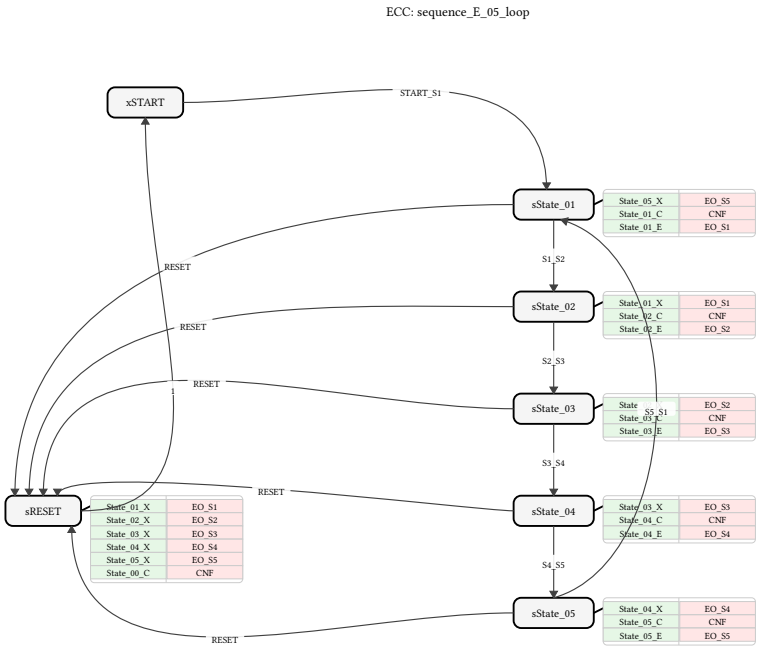
⚖️ Comparison with Similar Function Blocks

- **sequence_E_05**: Standard version without adapters.

Conclusion

Adapter version of the 5-step event sequencer.

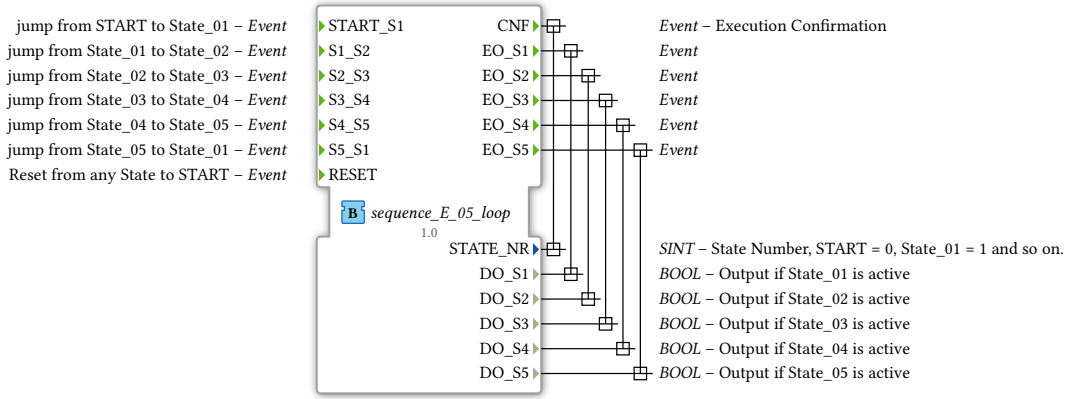
sequence_E_05_loop



• – ○ • – ○ • – ○ •

Introduction

The function block `sequence_E_05_loop` implements a cyclic sequence with five states. The transition between the individual states occurs exclusively via external events. The block is designed for applications in which a process step may only begin after the completion of a previous step and the arrival of a specific enable signal. The sequence can be reset to the initial start state from any state.



Interface Structure

Event Inputs

- `START_S1`: Switches from the start state (`START`) to the first active state (`State_01`).
- `S1_S2`: Changes from `State_01` to `State_02`.
- `S2_S3`: Changes from `State_02` to `State_03`.
- `S3_S4`: Changes from `State_03` to `State_04`.
- `S4_S5`: Changes from `State_04` to `State_05`.
- `S5_S1`: Switches from `State_05` back to `State_01` (cycle).
- `RESET`: Resets the sequence from any state to the initial start state (`START`).

Event Outputs

- CNF: General acknowledgment event that is triggered on every state change. It returns the current state number (STATE_NR).
- EO_S1: Triggered upon entering State_01 and returns the corresponding data output DO_S1. ...* EO_S1: Triggered upon entering State_01 and returns the corresponding data output DO_S1. * EO_S2: Triggered upon entry into State_02 and outputs the corresponding data output DO_S2.
- EO_S3: Triggered upon entry into State_03 and outputs the corresponding data output DO_S3.
- EO_S4: Triggered upon entry into State_04 and outputs the corresponding data output DO_S4.
- State_04: Triggered upon entry into State_04 and outputs the corresponding data output DO_S4.
- EO_S3: Triggered upon entry into State_04.
- qzmsdocs000036 ... * EO_S5: Triggered upon entering State_05 and outputs the corresponding data output DO_S5.

Data Inputs

- This function block has no data inputs.

Data Outputs

- STATE_NR (SINT): Outputs the number of the currently active state. The encoding is: START = 0, State_01 = 1, State_02 = 2, ..., State_05 = 5.
- DO_S1 (BOOL): Is TRUE when State_01 is active.
- DO_S2 (BOOL): Is TRUE when State_02 is active.
- DO_S3 (BOOL): Is TRUE when State_03 is active.
- DO_S4 (BOOL): Is TRUE when State_04 is active.
- DO_S5 (BOOL): Is TRUE when State_05 is active.

Adapters

- This function block does not use adapters.

Functionality

The FB is implemented as a Basic Function Block (BFB) with an Execution Control Chart (ECC). The internal logic is based on seven states: an initial start state (xSTART), five active states (sState_01 to sState_05), and a separate reset state (sRESET).

Upon entering an active state, three actions are executed sequentially:

1. **Exit action of the previous state:** The corresponding algorithm (State_XX_X) sets the associated data output (DO_Sx) to FALSE.
2. **Confirmation Action:** The State_XX_C algorithm sets the state number STATE_NR and triggers the CNF event.
3. **Entry Action of the New State:** The State_XX_E algorithm sets the corresponding data output (DO_Sx) to TRUE and triggers the corresponding event EO_Sx.

A RESET event leads to the sRESET state. Here, all potentially active outputs (DO_S1 to DO_S5) are deactivated via their respective exit algorithms, the state number is set to 0 (START), and a CNF event is triggered. The function block then automatically switches back to the initial xSTART state (condition = 1, i.e., always true).

Technical Features

- **Event-driven transitions:** Every state transition must be explicitly triggered by the corresponding input event. There are no time- or data-driven automatic transitions.
- **Decoupled Signals:** The Boolean state outputs (DO_Sx) and the associated event outputs (EO_Sx) are set and triggered synchronously. This allows for flexible integration with subsequent logic.
- **Explicit Reset Logic:** The reset process systematically deactivates all outputs before restoring the initial state, ensuring clean and defined behavior.
- **Constants for State Numbers:** State numbers are assigned via the imported namespace sequence:: (e.g., sequence::State_01), which improves maintainability and code readability.

State Overview

The ECC consists of the following states and possible transitions:

- **xSTART:** Initial, inactive state. Transition to `sState_01` via `START_S1`.
- **sState_01:** First active state. Transitions to `sState_02` via `S1_S2` or to `sRESET` via `RESET`.
- **sState_02:** Second active state. Transitions to `sState_03` via `S2_S3` or to `sRESET` via `RESET`.
- **sState_03:** Third active state. Transitions to `sState_04` via `S3_S4` or to `sRESET` via `RESET`.
- **sState_04:** Fourth active state. Transitions to `sState_05` via `S4_S5` or to `sRESET` via `RESET`.
- **sState_05:** Fifth active state. Transitions to `sState_01` via `S5_S1` (cycle) or to `sRESET` via `RESET`.
- – **sRESET:** Reset state. Executes reset actions and then unconditionally reverts to `xSTART`.

Application Scenarios

- **Sequence-Step Controls:** Control of machine processes where each step must be enabled manually or by a sensor signal (e.g., manual workstation with enable button).
- **Batch Processes:** Execution of recipe steps where the operator or a higher-level system must confirm the next step.
- **Safety-Critical Sequences:** Processes where uncontrolled automatic switching must be avoided.
- **Test and Commissioning Sequences:** Manually cycling through individual functions of a system.

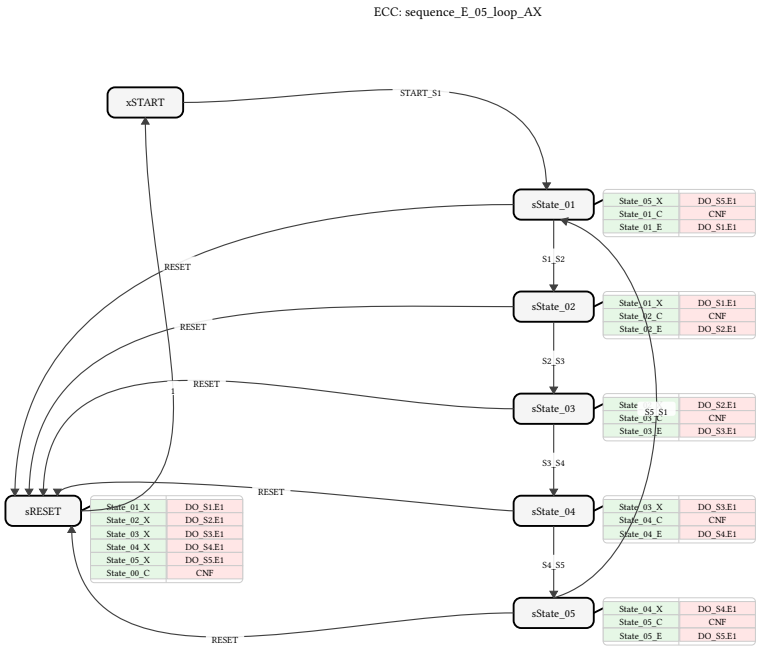
⚖️ Comparison with similar building blocks

In contrast to **cyclically running sequencers** (e.g., `E_CYCLE`) or **time-controlled sequencers** (with `DELAY`)(In blocks) this function block remains in each state until the specific change event occurs. It thus offers maximum external control. Compared to a generic **counter** (`CTU`) with downstream decoding logic, this function block provides a fully encapsulated, state-based solution with clear entry/exit actions and an integrated reset mechanism.

Conclusion

The `sequence_E_05_loop` is a robust and easy-to-configure function block for event-driven sequences with five states. Its clear interface, clean separation of entry/exit logic, and reliable reset mechanism make it ideal for applications where each process step must be explicitly enabled. The included data outputs and acknowledgment events allow for easy integration into higher-level control and visualization systems.

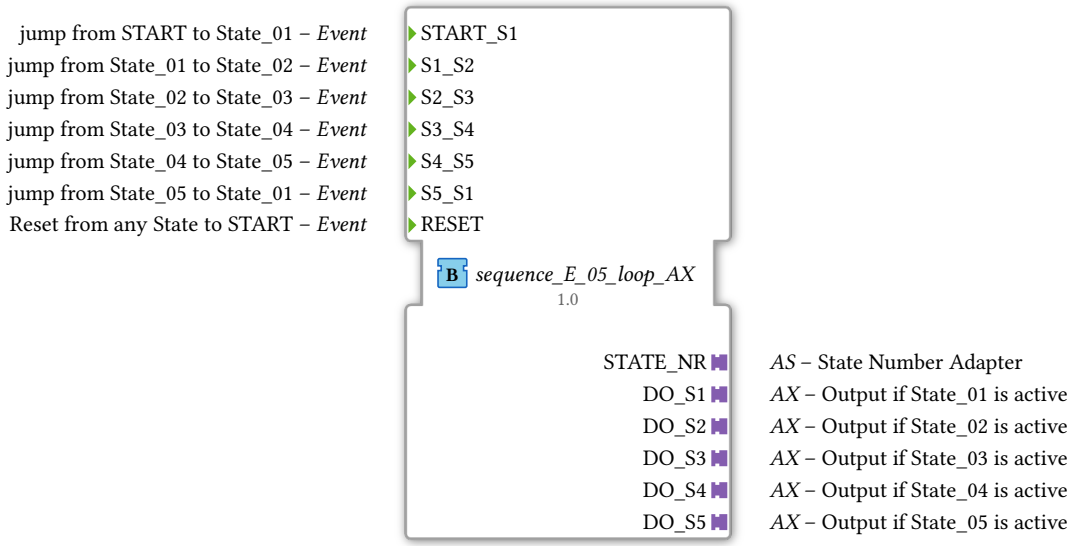
sequence_E_05_loop_AX



• – ○ • – ○ • – ○ •

Introduction

sequence_E_05_loop_AXis a variant ofsequence_E_05_loop that additionally uses adapters (AX`) for the outputs. It controls a purely event-driven, cyclic sequence with 5 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2**: Transition State_01 -> State_02.
- **S2_S3**: Transition State_02 -> State_03.
- **S3_S4**: Transition State_03 -> State_04.
- **S4_S5**: Transition State_04 -> State_05.

- **S5_S1**: Transition State_05 -> State_01 (Loop).
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Acknowledges execution.

Data Inputs

- None.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Output adapter for State_05.

Functionality

Corresponds to `sequence_E_05_loop`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_E_05_loop`.

Application Scenarios

For cyclic, event-driven, 5-stage sequences with adapter connectivity.

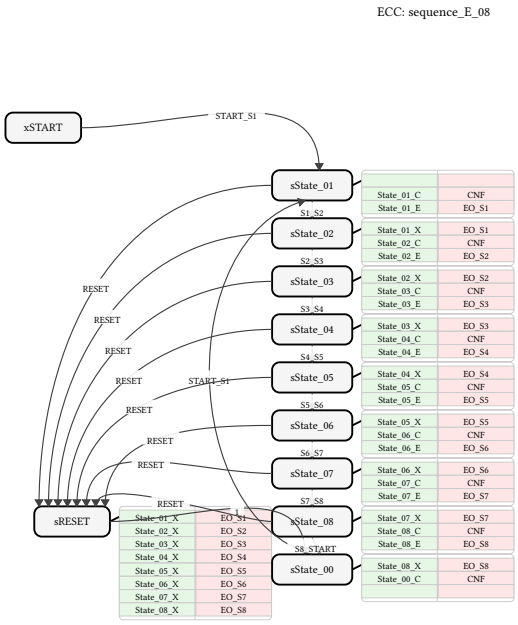
⚖️ Comparison with similar components

- **sequence_E_05_loop**: Standard version without adapter.

Conclusion

Adapter version of the 5-step loop event sequencer.

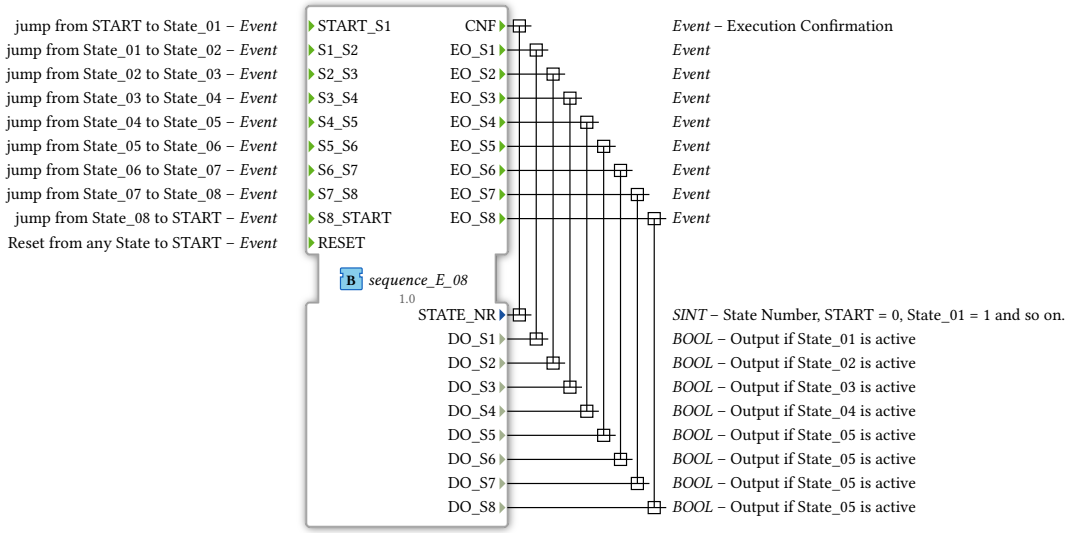
sequence_E_08



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_E_08` is an eight-output sequencer controlled by events. It implements a linear state machine with a defined start state and eight active states. The transition from one state to the next occurs exclusively upon the occurrence of a specific event. This block is suitable for control tasks where process steps must be executed sequentially and in response to events, such as in handling or assembly processes.



Interface Structure

Event Inputs

- **START_S1**: Changes from the start state (`START`) to the state `State_01`.
- **S1_S2**: Changes from `State_01` to `State_02`.
- **S2_S3**: Changes from `State_02` to `State_03`.

- **S3_S4**: Changes from State_03 to State_04.
- **S4_S5**: Changes from State_04 to State_05.
- **S5_S6**: Changes from State_05 to State_06.
- **S6_S7**: Changes from State_06 to State_07.
- **S7_S8**: Changes from State_07 to State_08.
- **S8_START**: Switches from State_08 back to the START state (represented by sState_00).
- **RESET**: Immediately resets the function block from *any state* to the START state.

Event Outputs

- **CNF**: Execution Confirmation. Triggered on every state change and returns the new state number (STATE_NR).
- **EO_S1**: Triggered upon entering State_01 and returns the value of DO_S1 (TRUE).
- **EO_S2**: Triggered upon entry into State_02 and returns the value of DO_S2 (TRUE).
- **EO_S3**: Triggered upon entry into State_03 and returns the value of DO_S3 (TRUE).
- **EO_S4**: Triggered upon entry into State_04 and returns the value of DO_S4 (TRUE).
- **EO_S5**: Triggered upon entry into State_05 and returns the value of DO_S5 (TRUE).
- **EO_S6**: Triggered upon entry into State_06 and returns the value of DO_S6 (TRUE).
- **EO_S7**: Triggered upon entry into State_07 and returns the value of DO_S7 (TRUE).
- **EO_S8**: Triggered upon entry into State_08 and returns the value of DO_S8 (TRUE).

Data Inputs

- This function block has no data inputs.

Data Outputs

- **STATE_NR** (SINT): Current state number. START = 0, State_01 = 1, State_02 = 2, ..., State_08 = 8. The values are loaded from the constant library sequence.
- **DO_S1** (BOOL): Is TRUE when state State_01 is active.
- **DO_S2** (BOOL): Is TRUE when state State_02 is active.
- **DO_S3** (BOOL): Is TRUE when state State_03 is active.
- **DO_S4** (BOOL): Is TRUE when state State_04 is active.
- **DO_S5** (BOOL): Is TRUE when state State_05 is active.
- **DO_S6** (BOOL): Is TRUE when state State_06 is active.
- **DO_S7** (BOOL): Is TRUE when state State_07 is active.
- **DO_S8** (BOOL): Is TRUE when state State_08 is active.

Adapters

- This function block does not use any adapters.

Functionality

The sequence_E_08 function block is implemented as a BASIC function block with a detailed Execution Control Chart (ECC). The logic follows a linear sequence of states (sState_01 to sState_08), a start/idle state (sState_00), and an explicit reset state (sRESET).

Three actions are performed on each state transition:

1. **Exit action of the previous state**: An algorithm (State_XX_X) sets the corresponding data output (DO_Sx) to FALSE.
2. **Confirmation Action of the New State**: A State_XX_C algorithm sets the STATE_NR and triggers the CNF event.
3. **Entry Action of the New State**: A State_XX_E algorithm sets the corresponding data output (DO_Sx) to TRUE and triggers the corresponding EO_Sx event.

A RESET event leads directly to the sRESET state. There, exit algorithms set *all* potentially active outputs (DO_S1 to DO_S8) to FALSE. An automatic transition then occurs (Condition=1, i.e., always true) to sState_00 (START).

Technical Features

- **Event-Based Transitions**: State transitions are strictly tied to the occurrence of defined events. There are no time- or data-triggered transitions.

- **Explicit Reset Path**: The reset process runs through a dedicated state (`sRESET`), which ensures that all outputs are deactivated before the start state (`sState_00`) is reached. This ensures a clean and defined restart.
- **Consistent Interface**: The naming convention for events (e.g., `S1_S2`) makes the expected transition intuitively understandable.
- **Use of Constants**: The state numbers (`STATE_NR`) are retrieved from a central constant library (`sequence`), which improves maintainability and consistency.

State Overview

- **sState_00**: Start or idle state. `STATE_NR` = 0. All outputs are `FALSE`.
- **sState_01** to **sState_08**: Active process states 1 to 8. The respective output `DO_Sx` corresponds to `TRUE`, and `STATE_NR` corresponds to the state number.
- **sRESET**: Intermediate state during a reset operation. Resets all outputs.
- **xSTART**: Initial ECC state (only executed once at application startup).

Application Scenarios

- **Sequential Control Systems**: Control of machines with clearly sequential workflows, such as presses, welding systems, or packaging machines.
- **Handling Devices**: Control of a robot arm that picks up, transports, and places parts sequentially.
- **Test Stands**: Automated sequence of test and inspection steps on a component.
- **Batch Processes**: Control of reactors or mixing vessels in which different ingredients must be added or process parameters changed sequentially.

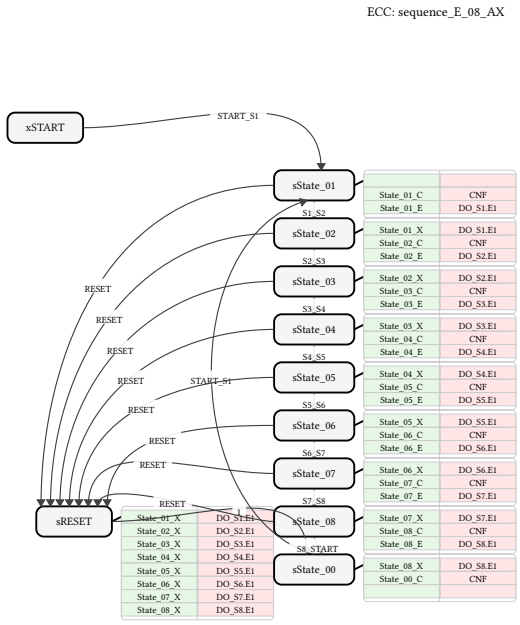
⚖️ Comparison with similar building blocks

- **VS. `E_R_TRIG` / `E_F_TRIG`**: These building blocks detect edges. `sequence_E_08`It typically uses such edge detectors as a source for its transition events, but is itself a complex state machine.
- **VS. `E_DELAY`**: A pure delay component. `sequence_E_08` can implement time-controlled processes through external timers and appropriate linking, but does not offer integrated timing by default.
- **VS. `E_SR` (Flip-Flop)**: An elementary memory component with set/reset functionality. `sequence_E_08` can be understood as a chain of such memories, where setting a state implies resetting the previous one.
- **VS. SFC (Sequential Function Chart)**: Its operation is essentially the same as a simple SFC sequence. `sequence_E_08` encapsulates this SFC-like logic in a reusable function block.

Conclusion

The `sequence_E_08` is a robust and well-structured tool for implementing event-driven sequences in IEC 61499. Its clear interface and explicit reset mechanism make it reliable and easy to integrate into higher-level controllers. It is ideally suited for applications where a fixed, linear sequence of actions is driven by external signals (e.g., limit switches, user commands). For cyclical or complex branched sequences, the function block would need to be extended or a different approach chosen.

sequence_E_08_AX

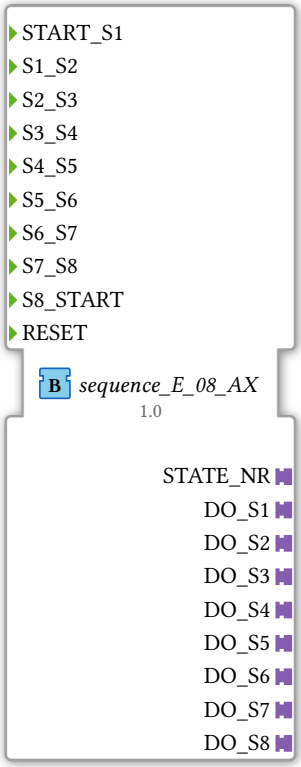


• – ○ • – ○ • – ○ •

Introduction

sequence_E_08_AXis a `variant ofsequence_E_08 that additionally uses adapters (AX`) for the outputs. It controls a purely event-driven sequence with 8 output states.

jump from START to State_01 – *Event*
jump from State_01 to State_02 – *Event*
jump from State_02 to State_03 – *Event*
jump from State_03 to State_04 – *Event*
jump from State_04 to State_05 – *Event*
jump from State_05 to State_06 – *Event*
jump from State_06 to State_07 – *Event*
jump from State_07 to State_08 – *Event*
jump from State_08 to START – *Event*
Reset from any State to START – *Event*



AS – State Number Adapter
AX – Output if State_01 is active
AX – Output if State_02 is active
AX – Output if State_03 is active
AX – Output if State_04 is active
AX – Output if State_05 is active
AX – Output if State_06 is active
AX – Output if State_07 is active
AX – Output if State_08 is active

Interface Structure

Event Inputs
- **START_S1**: Starts the sequence at State_01.
- **S1_S2** ... **S7_S8**: Manual transitions between states.
- **S8_START**: Manual transition State_08 -> START.
- **RESET**: Resets the sequence.

Event Outputs
- **CNF**: Execution acknowledgement.

Data Inputs
- None.

Data Outputs
- **STATE_NR** (SINT): Current state number.

Adapters
- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Output adapter for State_01 to State_08.

Functionality
Corresponds to `sequence_E_08`, but uses adapters for the outputs.

Technical Features
- Uses `adapter::types::unidirectional::AX`.

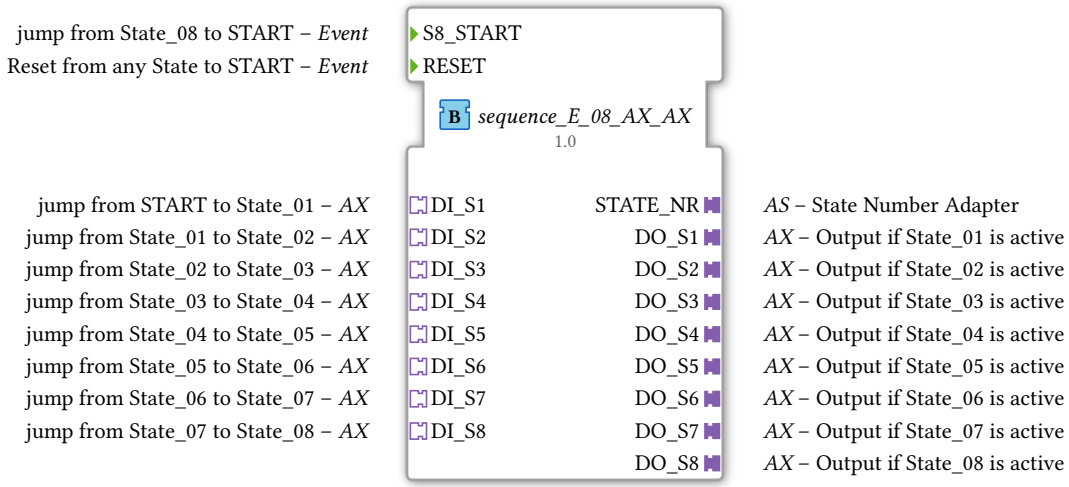
State Overview
See `sequence_E_08`.

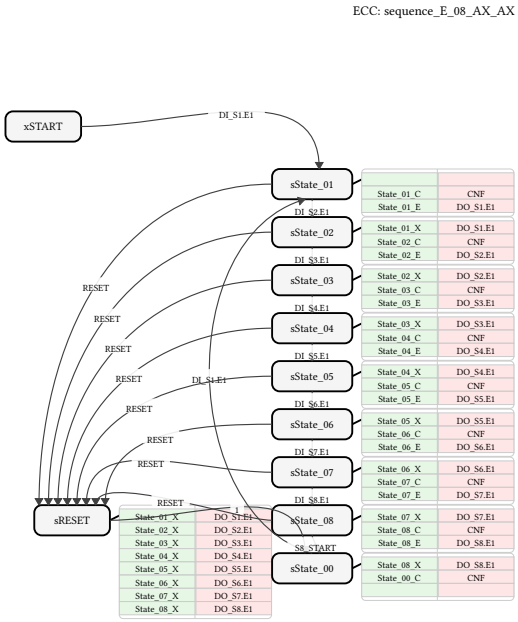
Application Scenarios
For event-driven 8-step sequences with adapter connectivity.

⚖️ Comparison with similar components
- **sequence_E_08**: Standard version without adapter.

Conclusion
Adapter version of the 8-step event sequencer.

sequence_E_08_AX_AX





• – ○ • – ○ • – ○ •

Introduction

The function block **sequence_E_08_AX_AX** implements a sequential control loop with eight output stages. It enables the step-by-step switching of states, with each state being exited by an event via an AX adapter input. An AX adapter provides a unidirectional interface with a data value (`D1`) that is transferred from the input adapter to the corresponding output adapter upon entering a state. The block is designed for use in automation systems that require a clear, event-driven sequence of steps.

Interface Structure

Event Inputs

Name	Type	Comment
`S8_START`	Event	Jumps from `State_08` back to the initial state `START`
`RESET`	Event	Resets from any state back to the initial state `START`

Event Outputs

Name	Type	Comment
`CNF`	Event	Confirmation of execution (coupled with `STATE_NR`)

Data Inputs

None (state transitions are controlled exclusively via events).

Data Outputs

Name	Type	Comment
`STATE_NR`	SINT	Current state number: `START` = 0, `State_01` = 1, ..., `State_08` = 8

Adapter

Plugs (Outputs – Unidirectional AX Adapter)

Name	Type	Comment
DO_S1	adapter::types::unidirectional::AX	Output active when State_01 is active
DO_S2	adapter::types::unidirectional::AX	Output active when State_02 is active
DO_S3`	adapter::types::unidirectional::AX	Output active when State_03 is active
DO_S4`	adapter::types::unidirectional::AX	Output active when State_04 is active
DO_S5`	adapter::types::unidirectional::AX	Output active when State_05 is active
DO_S6`	adapter::types::unidirectional::AX	Output active when State_06 is active
DO_S7	adapter::types::unidirectional::AX	Output active when State_07 is active
DO_S8	adapter::types::unidirectional::AX	Output active when State_08 is active

Sockets (Inputs – Unidirectional AX Adapter)

Name	Type	Comment
DI_S1	adapter::types::unidirectional::AX	Jumps from START to State_01
DI_S2	adapter::types::unidirectional::AX	Jumps from State_01 to State_02
DI_S3	adapter::types::unidirectional::AX	Jumps from State_02 to State_03
DI_S4	adapter::types::unidirectional::AX	Jumps from State_03 to State_04
DI_S5	adapter::types::unidirectional::AX	Jumps from State_04 to State_05
DI_S6	adapter::types::unidirectional::AX	Jumps from State_05 to State_06
DI_S7	adapter::types::unidirectional::AX	Jumps from State_06 to State_07
DI_S8	adapter::types::unidirectional::AX	Jumps from State_07 to State_08

Functionality

The function block operates on the principle of an event-driven step sequence. After starting, it is in state xSTART. As soon as an event arrives at socket DI_S1, it switches to state sState_01. Upon entering a state, the current data value of the associated input adapter (DI_Sx.D1) is transferred to the corresponding output adapter (DO_Sx.D1), and simultaneously, the event CNF with the current state number (STATE_NR) is triggered. When the component exits a state (e.g., due to an event at the next DI_Sx), the corresponding output adapter is set to FALSE (exit algorithm). The transition occurs stepwise: State_01 → State_02 → ... → State_08. After State_08, the event S8_START returns the component to the idle state sState_00. From there, a new sequence can be started with DI_S1. The event RESET interrupts the sequence at any time, sets all output adapters to FALSE, and returns to the idle state sState_00.

Technical Features

- **Use of AX Adapters**: All inputs and outputs are implemented as unidirectional AX adapters, which transmit a data value (D1) in addition to the event channel. This allows not only the event but also an associated value (e.g., a setpoint for an actuator) to be transmitted during a state change.
- **Entry/Exit Logic**: Deterministic state changes are ensured by the strict separation of entry (State_xx_E) and exit (State_xx_X) actions. The output of a state is always deactivated upon exiting to prevent looping.
- **Configurable Start Condition**: The input DI_S1 serves as the start condition both on the initial startup and after each sequence iteration. The separate event input S8_START allows for a manual or externally triggered return from the last state.
- **State Numbering**: The output STATE_NR always outputs the current state number, with 0 corresponding to the idle state.

State Overview

State (ECC)	Meaning	Actions
xSTART	Initial sleep state after activation	No output, expects DI_S1
sState_01	First step of the sequence	Sets DO_S1.D1 to DI_S1.D1; output STATE_NR=1
sState_02	Second step	Sets DO_S2.D1 to DI_S2.D1; STATE_NR=2
sState_03	Third step	Sets DO_S3.D1 to DI_S3.D1; STATE_NR=3
sState_04	Fourth Step	Sets DO_S4.D1 to DI_S4.D1; STATE_NR=4
sState_05	Fifth Step	Sets DO_S5.D1 to DI_S5.D1; STATE_NR=5
sState_06	Sixth Step	Sets DO_S6.D1 to DI_S6.D1; STATE_NR=6
sState_07`	Seventh Step	Sets DO_S7.D1 to DI_S7.D1; STATE_NR=7
sState_08	Eighth Step	Sets DO_S8.D1 to DI_S8.D1; STATE_NR=8
sState_00	Idle State After Sequence Iteration or Reset	No Output; STATE_NR=0
sRESET	Intermediate State on Reset	Sets **all** DO_Sx.D1 toFALSE; then transition to sState_00

Application Scenarios

- **Multi-stage Conveyor Control**: Each step activates a different conveyor section or a different diverter. The data value D1 can contain the speed or the direction of travel.
- **Program Flow in Machine Tools**: E.g., sequential approach to tools or stations, where each step receives a configurable parameter (e.g., pressure, temperature) via the AX adapter.
- **Lighting Control with Scenes**: Eight steps can activate different lighting scenes, where the data value of a scene (e.g., brightness values) is transferred from the input adapter to the output.
- **Test and Inspection Sequences**: Automated test sequences with eight consecutive test steps, where the measured values of the previous step serve as the target value for the next.

Comparison with Similar Function Blocks

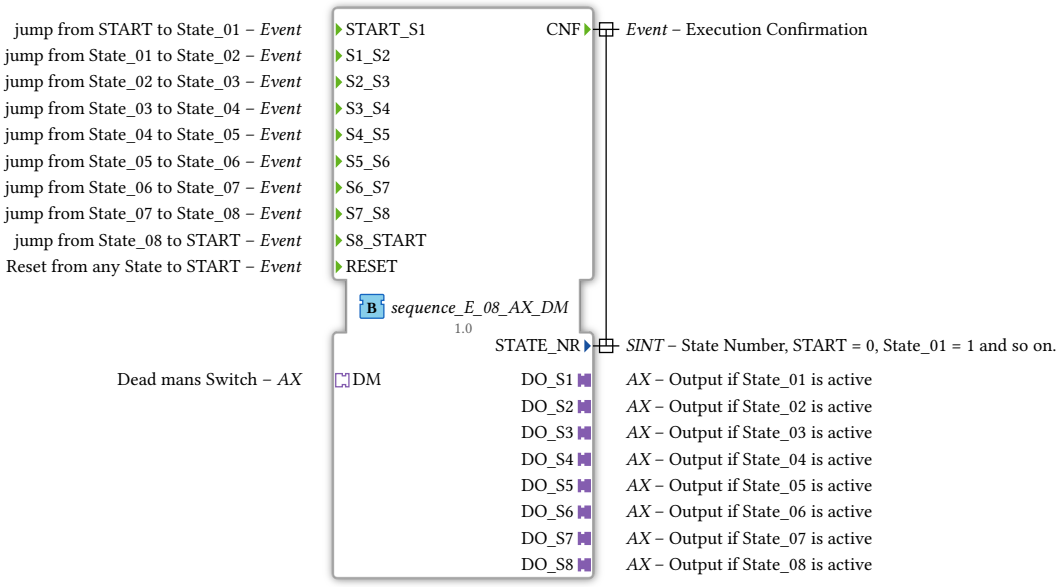
- **sequence_E_08 (without AX)**: A simple sequential function block that only controls Boolean signals. The function block described here extends this functionality with AX adapters, enabling the additional transfer of data values.
- **sequence_E_08_AX (with fewer adapters)**: Versions with fewer than eight stages offer a smaller number of steps, which may be sufficient or too restrictive depending on the application.
- **SFC Function Blocks (Step Function Chart)**: High-level language function blocks such as SFC allow parallel branching, which this simple sequencer does not support. However, it is significantly more resource-efficient and deterministic in its execution.

The function block presented here represents an optimized compromise between flexibility (through AX adapters) and clarity – ideal for standard step sequences with data transfer.

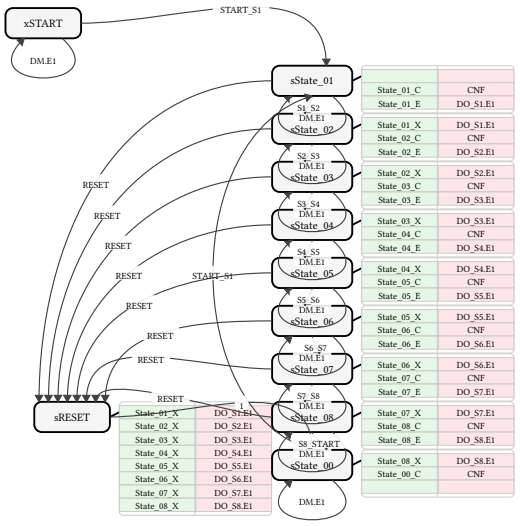
Conclusion

The sequence_E_08_AX_AX function block is a modular, event-driven sequencer with eight stages. Using AX adapters, it enables compact data transfer between steps. Its clear entry/exit logic, separate reset behavior, and state numbering make it a reliable tool for simple to medium-complexity sequence control applications in automation technology. It is particularly advantageous when each step needs to transmit not only a signal but also a configurable value to the actuator.

sequence_E_08_AX_DM



ECC: sequence_E_08_AX_DM



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_E_08_AX_DM` implements an event-driven sequence control with eight sequentially switchable outputs. It is based on a finite state machine with nine states and allows switching between states via explicit events. An integrated deadman switch (DM) allows monitoring and controlled reset of the outputs. The function block is specifically designed for use in safety-critical or monitored control sequences in agricultural technology.

Interface Structure

Event Inputs

Event	Description
START_S1	Transition from start state to state 1 (State_01)
S1_S2`	Transition from state 1 to state 2 (State_02)
S2_S3`	Transition from state 2 to state 3 (State_03)
S3_S4`	Transition from state 3 to state 4 (State_04)
S4_S5`	Transition from state 4 to state 5 (State_05)
S5_S6`	Transition from state 5 to state 6 (State_06)
S6_S7`	Transition from state 6 to state 7 (State_07)
S7_S8`	Change from state 7 to state 8 (State_08)
S8_START	Change from state 8 to the start state (State_00)
RESET	Reset from any state to the start state (State_00)

Event Outputs

Event	With Variable	Description
CNF	STATE_NR	Confirmation of state change; also returns the current state number

Data Inputs

No data inputs available.

Data Outputs

Variable	Type	Description
STATE_NR	SINT	Current state number (0 = State_00, 1 = State_01, ..., 8 = State_08)

Adapters

Direction	Name	Type	Description
Plug	DO_S1	unidirectional::AX	Output for state 1 (State_01 active)
Plug	DO_S2	unidirectional::AX	Output for State 2 (State_02 active)
Plug	DO_S3	unidirectional::AX	Output for State 3 (State_03 active)
Plug	DO_S4	unidirectional::AX	Output for State 4 (State_04 active)
Plug	DO_S5	unidirectional::AX	Output for State 5 (State_05 active)
Plug	DO_S6	unidirectional::AX	Output for State 6 (State_06 active)
Plug	DO_S7	unidirectional::AX	Output for State 7 (State_07 active)
Plug	DO_S8	unidirectional::AX	Output for state 8 (State_08 active)
Socket	DM	unidirectional::AX	Deadman switch; returns the event DM.E1 and the data value DM.D1

Functionality

The function block operates as a finite automaton with the following states: `xSTART` (initial), `sState_01` to `sState_08` (active sequence states), `sState_00` (start/wait state after sequence completion), and `sRESET` (intermediate state for reset).

Start and Process: After starting, the automaton is in state `xSTART`. The event `START_S1` transitions it to state `sState_01`. From there, the events `S1_S2`, `S2_S3`, ... up to `S7_S8` sequentially cycle through the eight states. The last state, `sState_08`, is transitioned to state `sState_00` by `S8_START`. This state serves as the rest point after the sequence and can be restarted via `START_S1`.

... - **Outputs**: Each state activates the corresponding plug adapter (`DO_Sx`). Upon entering a state, the data value of the deadman adapter (`DM.D1`) is passed to the output adapter (`DO_Sx.D1 := DM.D1`). Upon exiting a state, the output is deactivated (`DO_Sx.D1 := FALSE`).

- **Deadman Switch**: The event `DM.E1` triggers a self-transition in the current state (e.g., `sState_01 → sState_01`). This results in the exit and entry actions being re-executed, so that the current value from `DM.D1` is updated and passed to the output adapter. As long as the Deadman event is active (i.e., `DM.E1` occurs repeatedly), the output value remains at the current `DM.D1` state. If the Deadman event is no longer triggered, the state remains until a normal sequence event or `RESET` occurs.
- **Reset**: The event `RESET` transitions from every state (except `xSTART` and `sRESET`) to the intermediate state `sRESET`. There, all eight outputs are deactivated by calling the exit algorithms. The system then automatically transitions (with `1`) to the state `sState_00`. From there, the sequence can be restarted with `START_S1`.
- **Confirmation**: With each state change (except within `sRESET`), the event `CNF` is output with the current state number `STATE_NR`. The state number is set via the constants `sequence::State_00` to `sequence::State_08`.

Technical Features

- **Adapter-Based Outputs**: All eight outputs and the dead man's switch are implemented as unidirectional AX adapters. This allows for flexible connection to external hardware (e.g., analog or digital actuators) via the adapter frame.
- **Reusable Sequence Constants**: The state numbers are retrieved from a separate library (`logiBUS::utils::sequence::const::sequence`), allowing for consistent numbering across different sequence blocks.
- **Deadman Integration**: The deadman switch does not act as a block, but rather as a dynamic value generator for the outputs. Each state inherits the current value from `DM.D1` upon entry and can be updated by a repeated `DM.E1` event.
- **Explicit Reset**: The `RESET` event immediately deactivates all outputs and returns to a defined initial state – an important safety feature.

State Overview

State	Label	Output Active	Transitions
`xSTART`	Initial state	none	⟶ `sState_01` at `START_S1`; Self-transition at `DM.E1`
`sState_01`	Sequence step 1	`DO_S1`	⟶ `sState_02` at `S1_S2`; Self-transition at `DM.E1`; ⟶ `sRESET` at `RESET`
`sState_02`	Sequence step 2	`DO_S2`	⟶ `sState_03` at `S2_S3`; Self-translation at `DM.E1`; ⟶ `sRESET` at `RESET`
`sState_03`	Sequence step 3	`DO_S3`	⟶ `sState_04` at `S3_S4`; Self-translation at `DM.E1`; ⟶`sRESET` at `RESET`
`sState_04`	Sequence step 4	`DO_S4`	⟶ `sState_05` at `S4_S5`; Self-translation at `DM.E1`; ⟶ `sRESET` at `RESET`
`sState_05`	Sequence step 5	`DO_S5`	⟶ `sState_06` at `S5_S6`; Self-translation at `DM.E1`; ⟶ `sRESET` at `RESET`
`sState_06`	Sequence step 6	`DO_S6`	⟶ `sState_07` at `S6_S7`; Self-translation at `DM.E1`; ⟶ `sRESET` at `RESET`

sState_07	Sequence step 7	DO_S7	→ sState_08 at S7_S8; Self-translation at DM.E1; → sRESET at RESET
sState_08	Sequence step 8	DO_S8	→ sState_00 at S8_START; Self-translation at DM.E1; → sRESET at RESET
sState_00	Sleep state (after sequence)	None	→ sState_01 at START_S1; Self-transaction at DM.E1
sRESET	Reset state	All disabled	Automatic → sState_00

Application Scenarios

- **Agricultural Control Systems**: Step-by-step control of eight valves, actuators, or lighting units, e.g., for irrigation sequences or harvesting machines.
- **Safety-Monitored Processes**: Use in systems requiring a deadman's switch – the operator must keep the outputs active by repeatedly pressing the deadman switch.
- **Test and Inspection Stations**: Sequential activation of test steps with manual release by the operator (via events).
- **Modular Sequence Control**: Combination of multiple `sequence_E_08_AX_DM` blocks for more complex sequences with more than eight steps.

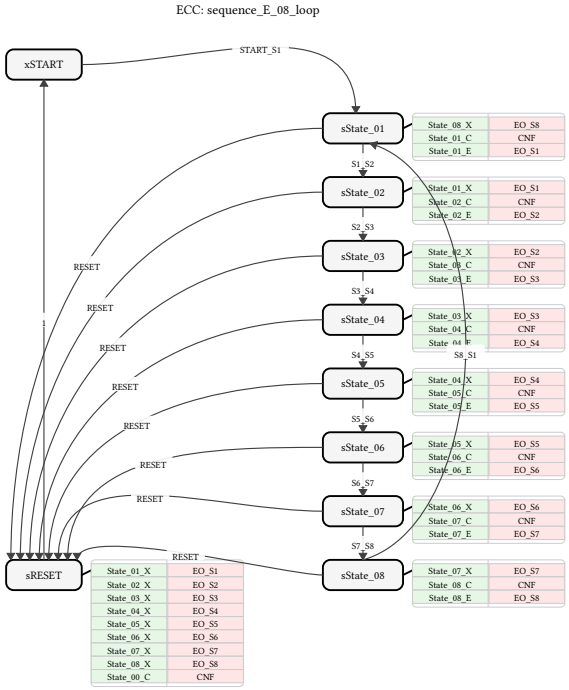
Comparison with Similar Blocks

- **`sequence_E_08_AX` (without deadman)**: Does not have a deadman switch – the outputs are permanently activated upon entry (e.g., with `TRUE`) and only deactivated upon exit. Suitable for non-critical control systems.
- **`sequence_E_04_AX_DM`**: Four-stage variant with correspondingly fewer outputs and events. Offers identical deadman functionality.
- **`sequence_T_08_AX_DM`**: Time-controlled variant – state transitions occur via timers instead of external events. Comparable deadman integration.

Conclusion

The `sequence_E_08_AX_DM` function block is a powerful and safe component for event-driven processes with up to eight steps. The integration of a deadman switch increases operational safety by requiring operator interaction. The adapter-based output interface allows for easy connection to various actuators. Thanks to its clearly structured state machine and explicit reset capability, this module is particularly suitable for safety-critical control tasks in agricultural technology and beyond.

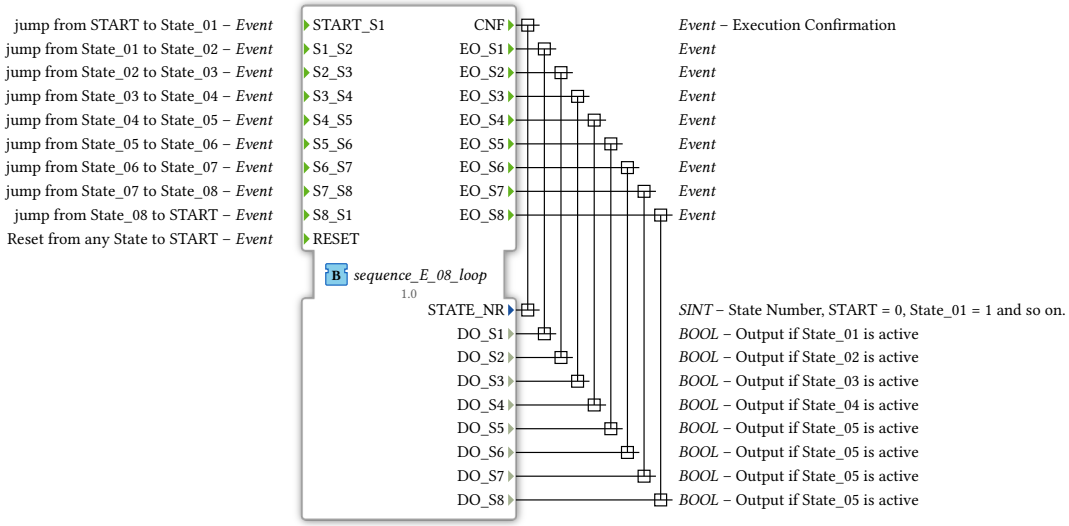
sequence_E_08_loop



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_E_08_loop` is a sequencer with eight output states that operates in a loop. It is used to control sequential processes where each step is triggered by an external event. The block is implemented as a Basic Function Block (BasicFB) according to IEC 61499 and is suitable for applications that require a clear, event-driven state machine.



Interface Structure

Event Inputs

- `START_S1`: Changes from the start state (`START`) to the state `State_01`.
- `S1_S2`: Changes from `State_01` to `State_02`.
- `S2_S3`: Changes from `State_02` to `State_03`.
- `S3_S4`: Changes from `State_03` to `State_04`.

- S4_S5: Changes from State_04 to State_05.
- S5_S6: Changes from State_05 to State_06.
- S6_S7: Changes from State_06 to State_07.
- S7_S8: Switches from State_07 to State_08.
- S8_S1: Switches from State_08 back to state State_01 (loop).
- RESET: Resets from any state back to the initial state (START).

Event Outputs

- CNF (Execution Confirmation): Triggered on every state change and returns the current state number (STATE_NR).
- EO_S1 ... EO_S8: These are triggered upon entering the respective state (State_01 to State_08) and provide the corresponding Boolean data output (DO_S1 ... DO_S8).

Data Inputs

- None present.

Data Outputs

- STATE_NR (SINT): The number of the active state. START = 0, State_01 = 1, State_02 = 2, etc.
- DO_S1 ... DO_S8 (BOOL): Logical outputs that are TRUE when the corresponding state is active.

Adapter

- No adapter interfaces are available.

Functionality

The function block operates as an event-driven state machine (ECC). The initial state is xSTART. An incoming event at one of the named event inputs (e.g., START_S1) triggers a transition to the next state (e.g., sState_01).

Three actions are performed during each state transition:

1. **Exit action (X) of the previous state**: Sets the corresponding data output (DO_Sx) to FALSE.
2. **Confirmation action (C) of the new state**: Sets the state number STATE_NR and triggers the CNF event.
3. **Entry Action (E) of the new state**: Sets the associated data output (DO_Sx) to TRUE and triggers the corresponding event EO_Sx.

The sequence traverses states 1 through 8 linearly. From State_08, the event S8_S1 returns to State_01, creating an infinite loop. The RESET event always returns to the initial state xSTART via a dedicated reset state (sRESET), resetting all active outputs in the process.

Technical Features

- **Pure Event Trigger**: State transitions occur exclusively through external events, not through conditions or timers.
- **Explicit Reset Logic**: The reset process uses a dedicated state (sRESET) in which all possible exit actions are invoked to ensure that all outputs are disabled before the initial state is reached.
- **State Confirmation**: The CNF event and the STATE_NR output provide clear feedback on the current system status.
- **Consistent Naming Scheme**: The names of events, states, and algorithms follow a consistent scheme (e.g., S1_S2, sState_01, State_01_C), which facilitates readability and maintenance.

State Overview

The ECC (Execution Control Chart) comprises the following states:

- xSTART: Initial, inactive wait state (state number 0).
- sState_01 to sState_08: The eight active sequence states (state numbers 1-8). Each activates its specific output.
- sRESET: Dedicated reset state. Upon entering this state, all potentially active outputs (DO_S1 to DO_S8) are deactivated, the state number is set to 0, and a transition to xSTART is executed.

Application Scenarios

- Control of cyclical manufacturing or assembly processes with manual or sensor-based release for the next step.

- Control of test sequences where each test step is started manually.
- Monitoring and control of batch processes where an operator initiates the next phase.
- As a central control module in machines with a clearly defined, step-by-step workflow.

⚖️ Comparison with Similar Modules

Compared to sequencers with time-controlled transitions (e.g., `sequence_T_08_loop`), this function block offers maximum flexibility because the dwell time in each state is externally defined. It has a simpler structure than building blocks with integrated error handling or complex branching, but provides a robust foundation for event-driven processes. Alternative implementations using `E_SR` or `E_CTUD` blocks would be significantly more complex and less user-friendly.

🛠️ Related Exercises

- Exercise_040
- Exercise_040_2
- Exercise_041

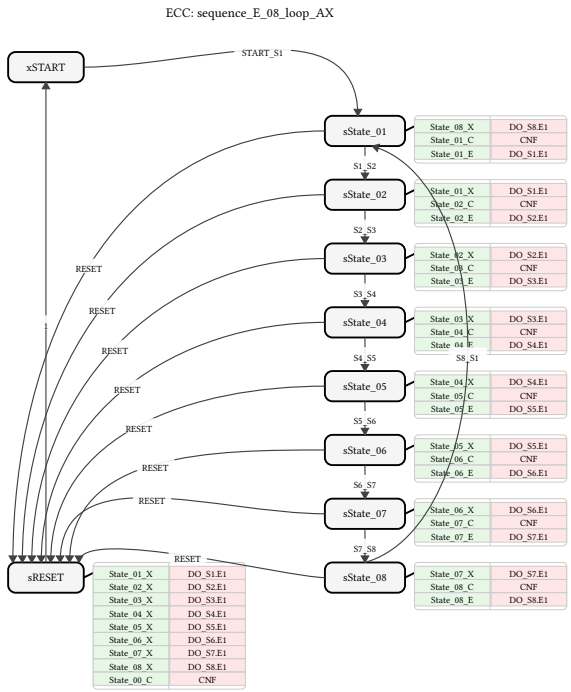
Conclusion

The `sequence_E_08_loop` is a specialized, reliable, and easy-to-configure function block for event-driven sequence flows. Its clear structure, consistent interface, and integrated reset functionality make it particularly suitable for applications where a step-by-step process needs to be driven by external signals (e.g., buttons, sensors, higher-level controllers). The output of the status number and the confirming events enable easy integration with visualization and monitoring systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter module on ms-muc-docs.de

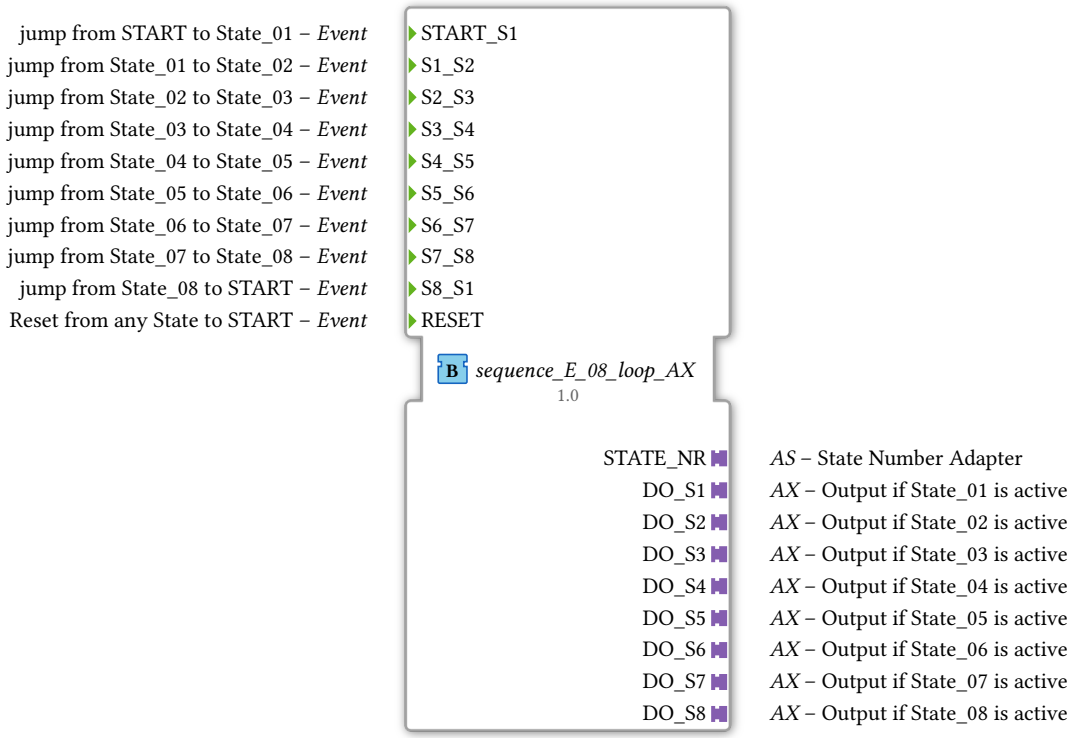
sequence_E_08_loop_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_E_08_loop_AXis a variant ofsequence_E_08_loop that additionally uses adapters (AX`) for the outputs. It controls a purely event-driven, cyclic sequence with 8 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **S1_S2** ... **S7_S8**: Manual transitions between states.
- **S8_S1**: Transition State_08 -> State_01 (Loop).
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirmation of execution.

Data Inputs

- None.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Output adapters for State_01 to State_08.

Functionality

Corresponds to `sequence_E_08_loop`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_E_08_loop`.

Application Scenarios

For cyclic, event-driven 8-step sequences with adapter connectivity.

⚖️ Comparison with similar modules

- **sequence_E_08_loop**: Standard version without adapter.

🛠️ Related Exercises

- Exercise_040_AX

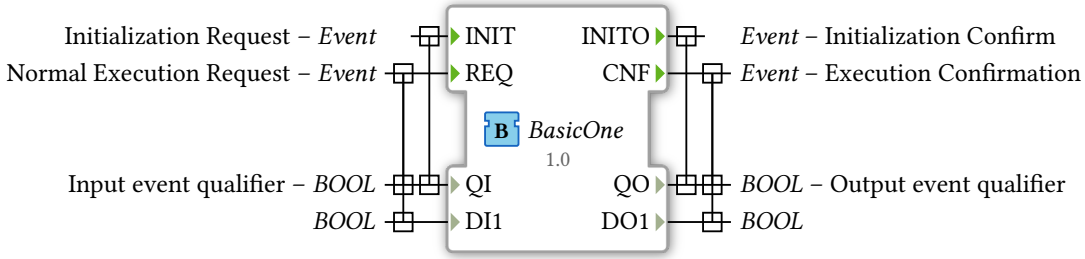
Conclusion

Adapter version of the 8-step loop event sequencer.

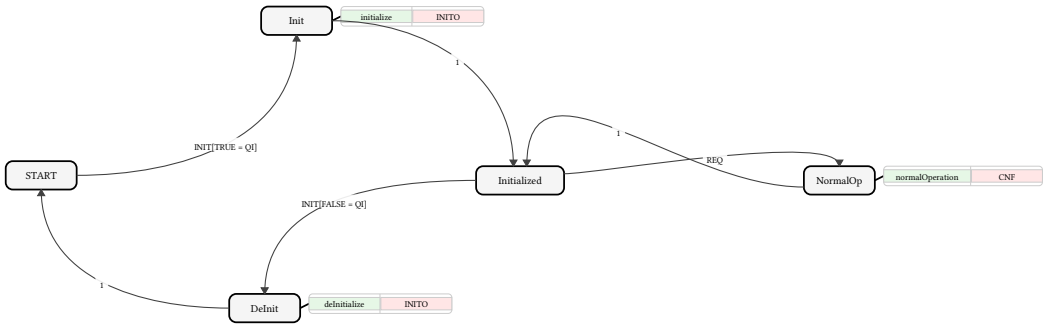
Numeric Index (sequence)

Here you will find all exercises from the `sequence` workspace, listed in ascending order. ...

BasicOne



ECC: BasicOne



• — ○ • — ○ • — ○ •

Introduction

The **BasicOne** function block is a basic, event-driven block according to IEC 61499. It serves as a simple sequence element for initialization, execution of a main operation, and proper deinitialization. The block is particularly suitable for control sequences where a resource state (e.g., device ready) needs to be monitored and reset. The block is included in the package `logiBUS::utils::sequence::one`.

Interface Structure

Event Inputs

Event	Type	Accompanying Data	Description
INIT	EInit	QI	Initialization request; the transition depends on the value of the qualifier QI.
REQ	Event	QI, DI1	Normal execution request; starts the main operation.

Event Outputs

Event	Type	Accompanying Data	Description
INITO	Initialization	QO	Confirmation of successful initialization or deinitialization.
CNF	Event	QO, DO1	Confirmation of normal execution; outputs the current state.

Data Inputs

Name	Type	Description
QI	BOOL	Input qualifier – controls whether initialization is active (TRUE) or deinitializing (FALSE). Also affects the execution of the main operation.
DI1	BOOL	First data input; The output is only transferred to DO1 during normal operation if initialization (QI = TRUE) is active.

Data Outputs

Name	Type	Description
QO	BOOL	Output qualifier – reflects the value of QI after an action (except after deinitialization: then FALSE).
DO1	BOOL	First data output; During a normal operation, the value is DI1 if QI = TRUE is present, otherwise FALSE.

Adapter

None.

Functionality

The function block goes through a clearly defined lifecycle:

1. **Start State (START)**: After startup, the function block waits for a INIT event.
2. **Initialization (Init)**: If INIT is received with QI = TRUE, the function block executes the *initialize* algorithm. This sets QO := QI (i.e., TRUE). Then, INITO is sent, and the function block transitions to the **Initialized** state.
3. **Normal Operation (NormalOp)**: In the **Initialized** state, a REQ event can initiate normal operation. The *normalOperation* algorithm sets QO := QI (still TRUE) and, if QI = TRUE is present, transfers the value from DI1 to DO1. Afterward, CNF is sent, and the function block automatically returns to **Initialized**.

The normalOperation* algorithm sets QO := QI (still TRUE) and transfers the value from DI1 to DO1. 4. **DeInitialization**: If a INIT event with QI = FALSE is received in the **Initialized** state, the function block executes the *deInitialize* algorithm. This sets QO := FALSE and DO1 := FALSE. Subsequently, INITO is sent, and the function block returns to the **START** start state.

Important: The normal operation is only executed if QI = TRUE is present. In the REQ event, the algorithm is executed, but DO1 remains unchanged (it is not set to DI1; instead, the last value or the default value FALSE remains as defined by the algorithm – in QI = FALSE, the IF condition is not met, therefore DO1 does not change).

`QI = FALSE``

Technical Features

- **Event Dependency of QI**: The transitions `INIT[TRUE = QI]` and `INIT[FALSE = QI]` demonstrate that the same event, `INIT`, triggers different state transitions depending on the value of `QI`. This enables compact control of initialization and deinitialization via a single event.
- **Automatic Fallback**: After a `REQ` event, the function block returns to the **Initialized** state without an external event (transition with condition `1`). This simplifies integration into cyclic processes.
- **No Self-Locking**: The function block does not block; after each successful execution, it is ready for the next event.

State Overview

State	Description	Possible Actions
`START`	Initial idle state after system startup or deinitialization.	Waits for `INIT` with `QI = TRUE`.
`Init`	Initialization phase; sets `QO = QI`.	Executes the *initialize* algorithm, sends `INITO`.
`Initialized`	Waiting state after successful initialization.	Can respond to `REQ` (starts NormalOp) or to `INIT` with `QI = FALSE` (starts DeInit).
`NormalOp`	Executes the main operation; transfers `DI1` to `DO1` (only if `QI = TRUE`).	Executes the *normalOperation* algorithm, sends `CNF`.
`DeInit`	Deinitialization phase; sets `QO = FALSE` and `DO1 = FALSE`.	Executes the *deInitialize* algorithm, sending `INITO`.

Application Scenarios

- **Initializing a Component**: A device must be configured upon power-up. Using `INIT` (QI=TRUE), the device state is set to "ready." `REQ` then executes the actual logic cyclically.
- **Resource Management**: The function block can be used as a simple sequencer for one-time initialization and subsequent deinitialization (e.g., for a database connection).
- **Safety-Oriented Control**: The qualifier `QI` ensures that the output `DO1` can only accept the input value if the function block is initialized. Deinitialization resets all outputs (`FALSE`).

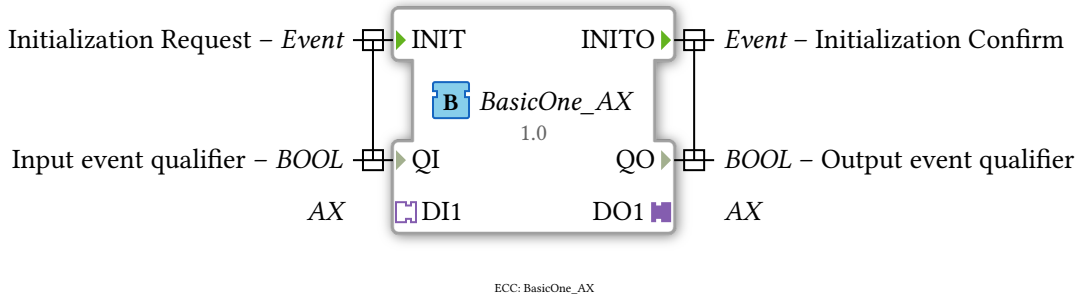
Comparison with similar building blocks

- **BasicBOOLEAN**: A simple Boolean building block without a state machine that only reacts to an event and passes a value. `BasicOne`, on the other hand, offers an explicit initialization and deinitialization sequence.
- **SR Flip-Flop**: A memory building block with set and reset capabilities. `BasicOne` is more of a state machine that implements a single start and stop operation, but does not have a hold function.
- **SimpleCycle**: A cyclic building block that repeatedly performs the same operation. `BasicOne`It distinguishes between initialization and cyclic execution, allowing for a clean separation.

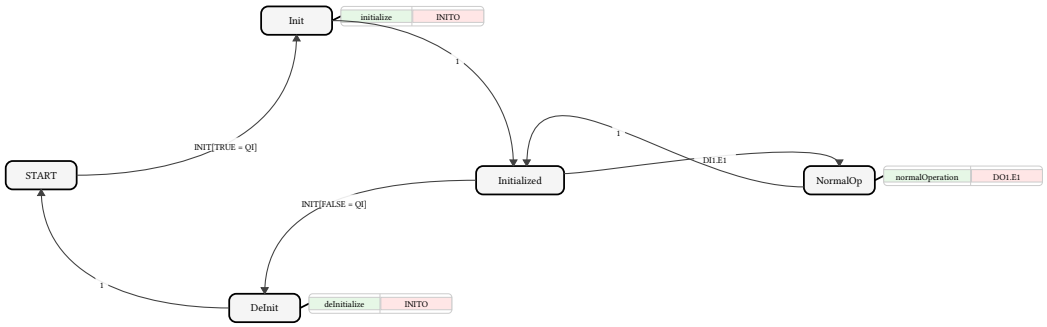
Conclusion

The function block `BasicOne` is an excellent basic element for control projects that require controlled initialization, reliable operation, and defined deinitialization. Its simple state machine and dependency on the qualifier `QI` make it flexible and easy to use without unnecessary complexity. It is suitable for both newcomers to the IEC 61499 world and experienced developers who need clear, reusable function blocks.

BasicOne_AX



ECC: BasicOne_AX



• – ○ • – ○ • – ○ •

Introduction

The function block `BasicOne_AX` is a basic IEC 61499 block of type *Basic Function Block*, serving as a template for simple control tasks. It implements an initializable switching of a Boolean signal using unidirectional adapter interfaces of type `AX`. The block is suitable for scenarios in which an input signal needs to be copied to an output and the initialization state needs to be monitored.

Interface Structure

Event Inputs

Event	Type	With Variables	Comment
`INIT`	`EInit`	`QI`	Initialization Request

Event Outputs

Event	Type	With Variables	Comment
`INITO`	`EInit`	`QO`	Initialization Confirmation

Data Inputs

Variable	Type	Comment

QI	BOOL	Input Qualifier (Turns initialization on/off)

Data Outputs

Variable	Type	Comment
QO	BOOL	Output Qualifier (Reflects the initialization state)

Adapter

Direction	Name	Type	Comment
Plug (Output)	DO1	adapter::types::unidirectional::AX	Unidirectional adapter for data transmission (AX interface)
Socket (input)	DI1	adapter::types::unidirectional::AX	Unidirectional adapter for receiving data (AX interface)

Functionality

The function block iterates through a finite state machine (ECC) with the states START, Init, Initialized, NormalOp, and DeInit.

- **START → Init**: Upon the occurrence of the event INIT, the process is executed with QI = TRUE. The algorithm initialize sets QO to the value of QI (i.e., TRUE). The output INITO signals successful initialization.
- **Init → Initialized**: After initialization, the automaton immediately transitions to the state Initialized (transition with condition 1). The module remains in this state until either an event arrives at socket DI1.E1 or a subsequent event, INIT, triggers deinitialization with QI = FALSE.
- **Initialized → NormalOp**: Upon arrival of the adapter event DI1.E1, the module transitions to state NormalOp. The algorithm normalOperation then performs the actual switching: If QI = TRUE is present, the value of the adapter input DI1.D1 is copied to the adapter output DO1.D1. Additionally, QO is set to the current value of QI (remaining TRUE until QI is set to FALSE).
- **NormalOp → Initialized**: After execution, the function block immediately returns to the state Initialized (transition 1), ready for the next DI1.E1 event.
- **Initialized → DeInit**: When the event INIT with QI = FALSE is received, deinitialization occurs. The algorithm deInitialize sets QO to FALSE and the adapter output DO1.D1 to FALSE. Then, INITO is output.
- **DeInit → START**: After deinitialization, the automaton returns to its initial state and awaits re-initialization.

In summary, the function block operates as a *triggered passthrough*: The adapter input is copied to the adapter output with each event at the socket, but only if the function block was previously initialized correctly (QI=TRUE for INIT).

Technical Features

- **Adapter-Based Interface**: Input/output data is not exchanged via direct variables, but rather via the unidirectional adapters DI1 and DO1 (type AX). This enables flexible coupling with other function blocks of the same adapter type.
- **Initialization Logic**: The function block distinguishes between an initialized (QI=TRUE) and a deinitialized (QI=FALSE) state. Incoming adapter events are only processed after successful initialization.
- **Package Name**: The function block is organized in the package logiBUS::utils::sequence::one, indicating its use in a logiBUS environment.
- **Automatic State Transitions**: The transitions of Init→Initialized, NormalOp→Initialized, and DeInit→START occur without any further condition (transition 1), meaning the execution of the algorithms is complete.

State Overview

The ECC (Execution Control Chart) of the module comprises five states:

State	Description
START	Waiting for the first INIT event with QI=TRUE.

Init	Executes initialization algorithm, sets QO=QI, and sends INITO.
Initialized	Idle state after successful initialization; waits for DI1.E1 or another INIT (with QI=FALSE).
NormalOp	Performs passthrough (copies DI1.D1 to DO1.D1 if QI=TRUE is present).
DeInit	Performs deinitialization (sets QO and DO1.D1 to FALSE), sends INITO.

Application Scenarios

- **Signal Passthrough with Fuse**: A Boolean signal (e.g., enabling a process) should only be passed from a source to a sink if the function block has been initialized beforehand. Sending INIT followed by QI=FALSE again interrupts the passthrough.
- **Initialization Sequence in Manufacturing Plants**: Before production starts, the function block is initialized (e.g., checking safety conditions). Only then may actuators be controlled via DO1.
- **Data Transfer in Modular Systems**: The unidirectional adapter AX allows loose coupling between modules – the component serves as a simple "pass-through" in a data chain.

Comparison with Similar Components

- **BasicOne (without adapter)**: A typical BasicOne component uses direct input/output variables. BasicOne_AX extends this with an adapter-based interface, improving reusability and structuring in larger systems.
- **Standard E_SR (Set/Reset)**: A set/reset flip-flop stores a state, while BasicOne_AXIt has no memory function – the output follows the input only during active initialization and upon each event.
- **Move Function Block**: A MOVE block continuously copies values. In contrast, BasicOne_AX only copies upon an event (triggered) and only after successful initialization.

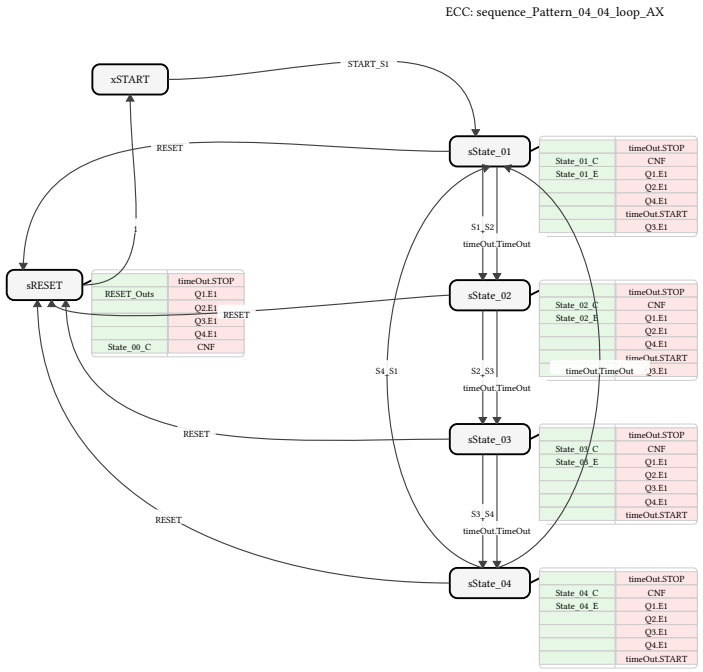
Conclusion

The BasicOne_AX is a minimalist yet robust IEC 61499 block for initialization-controlled signal switching with an adapter interface. Its clear state logic and the separation of initialization and operation make it particularly suitable for safety-critical or sequential applications where a defined start state is required before data transmission. The use of unidirectional adapters increases modularity and facilitates integration into complex automation systems.

Numerical Index (sequence)

Here you will find all exercises from the sequence workspace in ascending order. ...

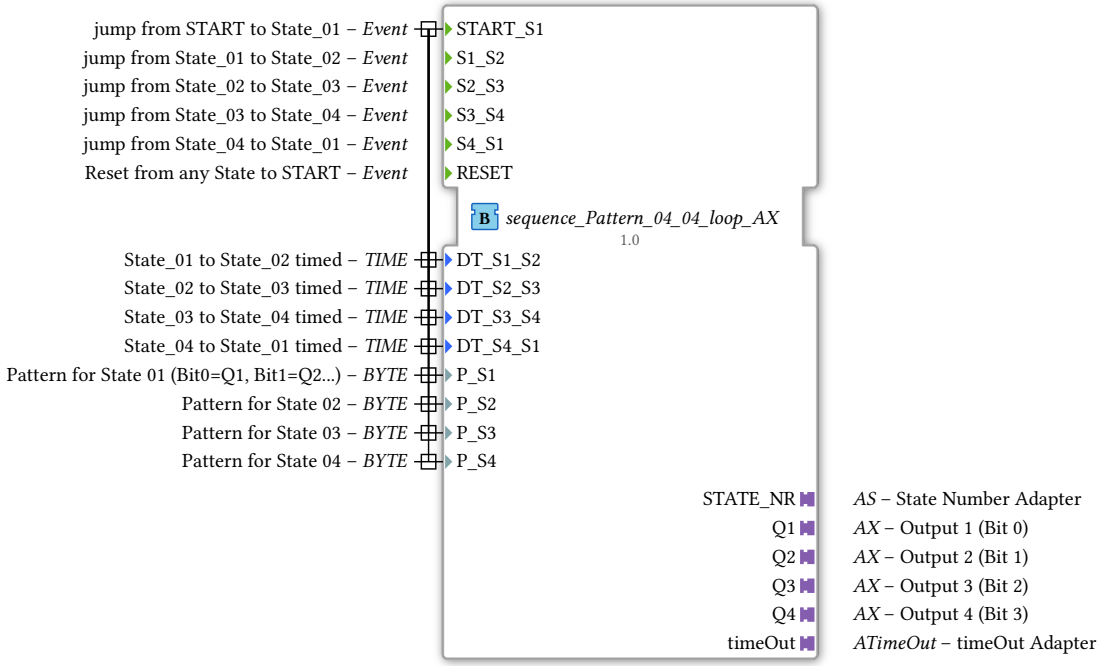
sequence_Pattern_04_04_loop_AX



• — ○ • — ○ • — ○ •

Introduction

The function block **sequence_Pattern_04_04_loop_AX** is a sequencer (step switch) that controls a sequence of four states in an endless loop. It functions similarly to an electronic cam switch. A specific output pattern can be defined for each of the four steps, controlling four outputs (Q1 to Q4). The block uses **AX adapters** for the outputs and supports both timed transitions (timeouts) and event-driven manual advancement. After the fourth step, the sequence automatically or manually returns to the first step (loop behavior).



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence and jumps directly to state 1 (`State_01`). The configuration data (times and patterns) is read in.
- **S1_S2**: Manual trigger for the transition from step 1 to step 2.
- **S2_S3**: Manual trigger for the transition from step 2 to step 3.
- **S3_S4**: Manual trigger for the transition from step 3 to step 4.
- **S4_S1**: Manual trigger for the transition from step 4 back to step 1 (loop closure).
- **RESET**: Resets the function block from any state to its initial state (`START`). All outputs are deactivated.

Event Outputs

- **CNF**: Execution Confirmation Event. Fires when the state changes or the function block is initialized.

Data Inputs

- **DT_S1_S2** (TIME): Dwell time in step 1 before automatically switching to step 2. Default value: `NO_TIME` (no automatic switch).
- **DT_S2_S3** (TIME): Dwell time in step 2.
- **DT_S3_S4** (TIME): Dwell time in step 3.
- **DT_S4_S1** (TIME): Dwell time in step 4 before automatically switching back to step 1.
- **P_S1** (BYTE): Output pattern for step 1.
- Bit 0 controls Q1
- Bit 1 controls Q2
- Bit 2 controls Q3
- Bit 3 controls Q4
- **P_S2** (BYTE): Output pattern for step 2.
- **P_S3** (BYTE): Output pattern for step 3.
- **P_S4** (BYTE): Output pattern for step 4.

Data Outputs

- **STATE_NR** (SINT): Current state number (0 = START, 1 = State_01, ..., 4 = State_04).

Adapter

- **Q1** (`adapter::types::unidirectional::AX`): Output 1 (controlled by bit 0 of the pattern).
- **Q2** (`adapter::types::unidirectional::AX`): Output 2 (controlled by bit 1 of the pattern).
- **Q3** (`adapter::types::unidirectional::AX`): Output 3 (controlled by bit 2 of the pattern).
- **Q4** (`adapter::types::unidirectional::AX`): Output 4 (controlled by bit 3 of the pattern).
- **timeOut** (`iec61499::events::ATimeOut`): Internal adapter for managing time delays (timers).

Functionality

The function block implements a state machine (ECC) that cycles through four active states.

1. **Initialization**: The function block starts in state `xSTART`. Upon the event `START_S1`, it transitions to state `sState_01`.
2. **State Processing**: In each state (`sState_01` to `sState_04`):

- The timer (`timeOut`) is stopped and restarted with the corresponding time (`DT_xx`).
- The current state number (`STATE_NR`) is updated and sent (`CNF`).
- The outputs (`Q1` to `Q4`) are set based on the corresponding input byte (`P_Sx`). The mapping is bitwise (bit 0 -> Q1, bit 1 -> Q2, etc.).
- The outputs (`Q1` to `Q4`) are set based on the corresponding input byte (`P_Sx`). * Events are fired at the adapters `Qx` (`Qx.E1`) to signal the data change.

1. **Transitions**: A transition to the next step occurs either:

- **Automatic**: When the configured time (`DT_...`) has expired (`timeOut.TimeOut`).
- **Manual**: When the explicit transition event (e.g., `S1_S2`) arrives.

1. **Loop**: After `sState_04`, the transition back to `sState_01` (loop) occurs, unless a reset is triggered.
2. **Reset**: The `RESET` event causes all outputs to be set to `FALSE`, `STATE_NR` to be reset to 0, and the function block to wait for a new start command in the `xSTART` state.

Technical Features

- **Bit Mapping**: Output control is efficiently achieved via byte patterns. This allows any combination of the four outputs per step to be defined with a single parameter.
- Example: `P_S1 = 5` (binary `0000 0101`) activates Q1 and Q3.
- **AX Adapter**: The use of AX adapters indicates that this function block is designed for modular systems where actuators are connected via standardized interfaces.
- **Time & Event Duality**: The flexibility to exit steps based on both time and event allows for mixed operating modes (e.g., Step 1 exits after a time delay, Step 2 waits for manual confirmation).

State Overview

State	Description	Exit Logic (Q1-Q4)	Next State (Auto/Manual)
xSTART	Idle State	Inactive	sState_01 (via `START_S1`)
sState_01	Step 1	P_S1	sState_02
sState_02	Step 2	P_S2	sState_03
sState_03	Step 3	P_S3	sState_04
sState_04	Step 4	P_S4	sState_01 (Loop)
sRESET	Reset Logic	All FALSE	xSTART

Application Scenarios

- **Running Light / Flashing Light**: Simple cyclical light control.
- **Cam Switch Simulation**: Control of recurring machine cycles (e.g., gripping, lifting, moving, lowering).
- **Traffic Light Control**: Simple traffic light phases can be implemented by adjusting the `DT` times (where 4 steps are usually sufficient, e.g., red, red-yellow, green, yellow).
- **Test Pattern Generator**: Generation of defined signal sequences for testing purposes.

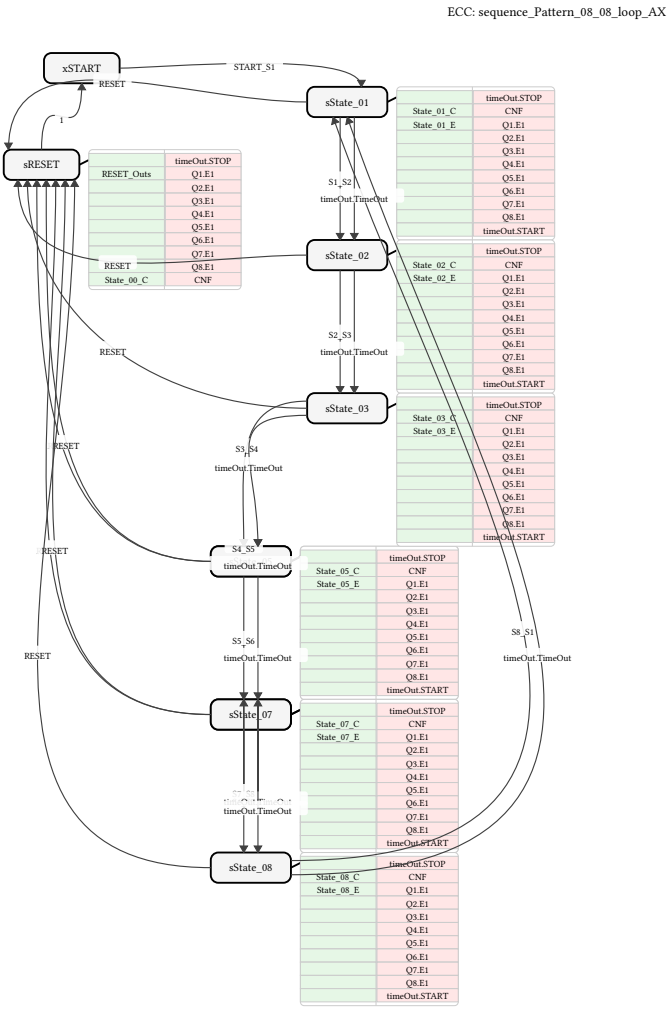
⚖️ Comparison with Similar Function Blocks

Unlike simple counters, this function block offers integrated timers per step and direct pattern assignment. Compared to more complex sequencers (e.g., SFC / Sequential Function Chart), it is more compact but limited to exactly 4 steps and 4 outputs. The “loop” variant differs from “single-shot” sequencers in that it does not stop at the end but repeats indefinitely.

Conclusion

The `sequence_Pattern_04_04_loop_AX` is a robust and flexible function block for small, cyclic control tasks in IEC 61499 applications. Its combination of time and event control, along with direct bit-to-adapter assignment, makes it ideally suited for repetitive automation sequences.

sequence_Pattern_08_08_loop_AX

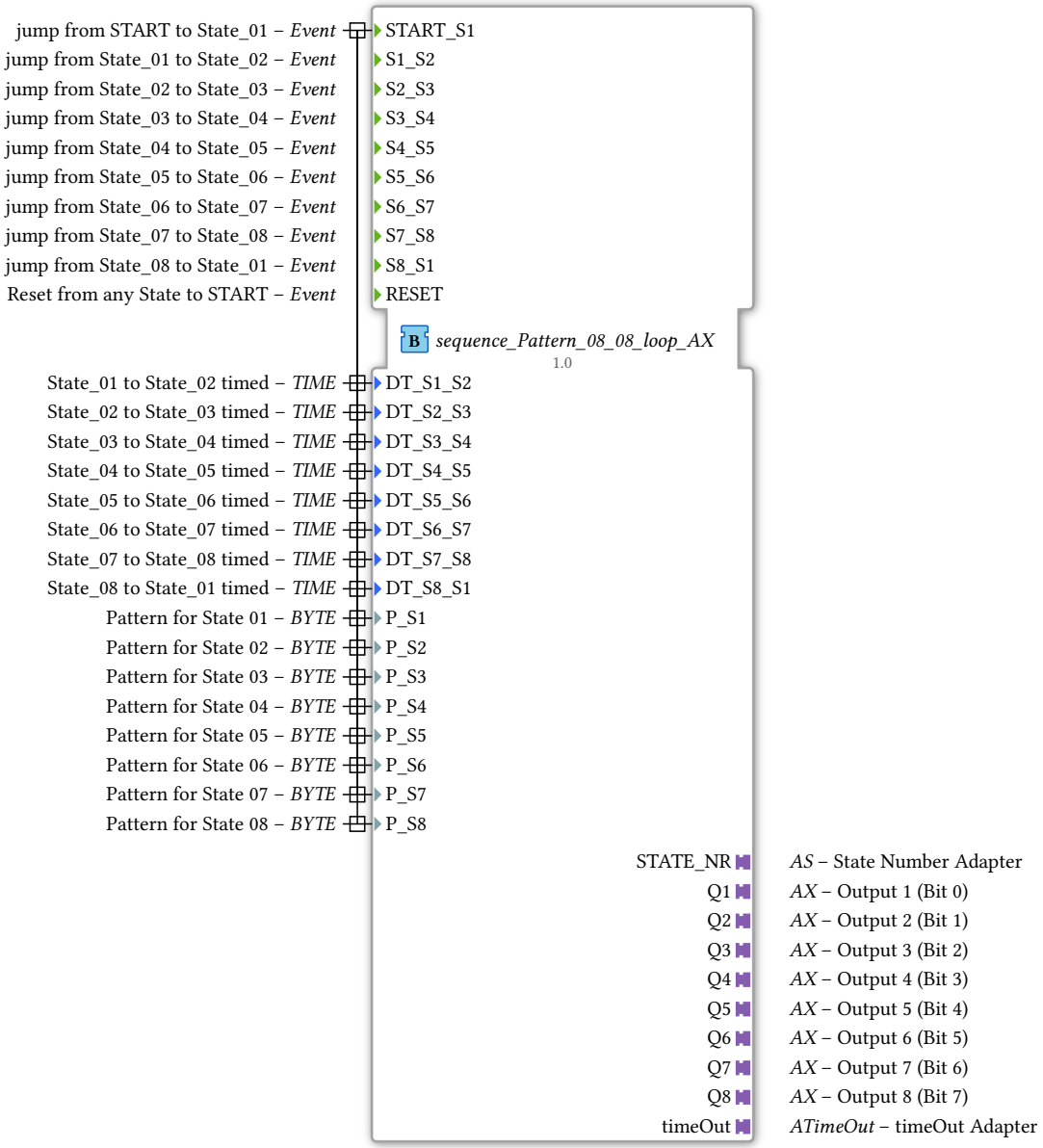


• – ○ • – ○ • – ○ •

Introduction

The function block **sequence_Pattern_08_08_loop_AX** is a sequencer (step chain) that implements a configurable cam switch with 8 states (steps) and 8 outputs. It is designed to operate cyclically (loop behavior), transitioning from step 8 back to step 1.

This block allows the definition of bit patterns for each of the 8 steps, which are output via special adapters (AX interface). Transitions between states can be either event-driven or time-controlled.



Interface Structure

Event Inputs

- **START_S1**: Initializes the sequence and jumps directly to state 1 (`State_01`). This event also inherits the parameterized time values (`DT_...`) and bit patterns (`P_...`).
- **S1_S2** to **S8_S1**: Manual trigger events to immediately switch from the current state to the next (e.g., `S1_S2` jumps from state 1 to 2).
- **RESET**: Resets the function block from any state to the initial state (`START`) and disables all outputs.
- **S1_S2** to **S8_S1**: Manual trigger events to immediately switch from the current state to the next (e.g., `S1_S2` jumps from state 1 to 2).

Event Outputs

- **CNF**: Execution Confirmation event. It fires as soon as a new state is reached and the outputs are updated.

Data Inputs

- **DT_S1_S2** to **DT_S8_S1** (Type: `TIME`): Define the dwell time in the respective state before automatically switching to the next one. The default value is `NO_TIME` (no automatic switching based on time).
- **P_S1** to **P_S8** (Type: `BYTE`): Define the output pattern for the respective state.

- Bit 0 controls adapter Q1
- Bit 1 controls adapter Q2
- ...
- Bit 7 controls adapter Q8

Data Outputs

- **STATE_NR** (Type: `SINT`): Outputs the current state number (0 = START, 1 = State_01, ..., 8 = State_08).

Adapters

- **Q1** to **Q8** (Type: `adapter::types::unidirectional::AX`): The 8 physical or logical outputs. Each adapter receives a data signal (`D1`) and an event signal (`E1`).
- **timeOut** (Type: `iec61499::events::ATimeOut`): An internal adapter for handling timed transitions between steps.

Functionality

The component operates as a finite state machine with a cyclic structure:

1. **Initialization**: The sequence is started by the event `START_S1`. The configured time values and bit patterns are read in. The state machine transitions to **State_01**.
2. **State Logic**: In each state (`State_01` to `State_08`), the corresponding byte pattern (`P_Sn`) is parsed.

- If bit 0 of `P_S1` is set, `Q1.D1` is set to TRUE.
- If bit 1 of `P_S1` is set, `Q2.D1` is set to TRUE, and so on.
- Simultaneously, the event `E1` is triggered on all adapters `Q1` through `Q8` to signal the change.

1. **Transitions**: The transition to the next state occurs when:

- The explicit event is received (e.g., `S1_S2`).
- OR the configured time (`DT_S1_S2`) has expired (via the `timeOut` adapter).
- 1. **Cycle**: After **State_08**, the process transitions back to **State_01** (`S8_S1` or timeout), creating an infinite loop.

1. **Reset**: The event `RESET` immediately interrupts the process, sets all outputs (`Q1` to `Q8`) to `FALSE`, and sets the state number to 0.

Technical Features

- **AX Adapter Usage**: Instead of classic `BOOL` outputs, this function block uses adapters of type `AX`. This allows for more flexible integration, e.g., with hardware abstraction layers or complex actuators that require an event signal for data transfer.
- **Bit Mapping**: The 8 outputs are efficiently controlled via `BYTE` variables. This drastically reduces the number of required input pins on the function block compared to individual Boolean inputs per step and output.
- **Hybrid Control**: The function block supports both time-controlled (timer) and event-controlled (external trigger) sequences, making it highly flexible.

State Overview

State ID	Name	Description	Output Logic	Transition to
0	xSTART	Waiting State / Reset	None	State_01 (at START_S1)
1	sState_01	Step 1	Q1-Q8 according to P_S1	State_02
2	sState_02	Step 2	Q1-Q8 according to P_S2	State_03
3	sState_03	Step 3	Q1-Q8 according to P_S3	State_04
4	sState_04	Step 4	Q1-Q8 according to P_S4	State_05
5	sState_05	Step 5	Q1-Q8 according to P_S5	State_06
6	sState_06	Step 6	Q1-Q8 according to P_S6	State_07

7	sState_07	Step 7	Q1-Q8 according to P_S7	State_08
8	sState_08	Step 8	Q1-Q8 according to P_S8	State_01 (Loop)
-	sRESET	Reset State	Q1-Q8 = FALSE	xSTART

Application Scenarios

- **Running Light Controls**: A simple running light effect can be implemented by setting `P_S1=1`, `P_S2=2`, `P_S3=4`, etc.
- **Cam Switches**: Control of mechanical processes where multiple actuators (cylinders, valves) must be activated in a fixed sequence.
- **Traffic Light Controls**: Cyclic sequence of red-yellow-green phases.
- **Cleaning Cycles**: Rinsing, washing, and drying at recurring intervals.
-

⚖️ Comparison with Similar Function Blocks

Unlike linear sequencers (without a loop), this function block is explicitly designed for recurring processes. It differs from simple counters in that an individual output pattern and time can be defined for each step. The use of AX adapters distinguishes it from standard IEC 61499 function blocks, which mostly use direct Boolean outputs, and makes it ideal for structured, object-oriented control designs.

Conclusion

The **sequence_Pattern_08_08_loop_AX** is a powerful and compact function block for implementing complex, cyclic sequences. Parameterization via byte patterns and time values allows for the implementation of diverse control tasks without modifying the internal logic. The integrated reset functionality and adapter interface ensure seamless integration into modern 4diac applications.

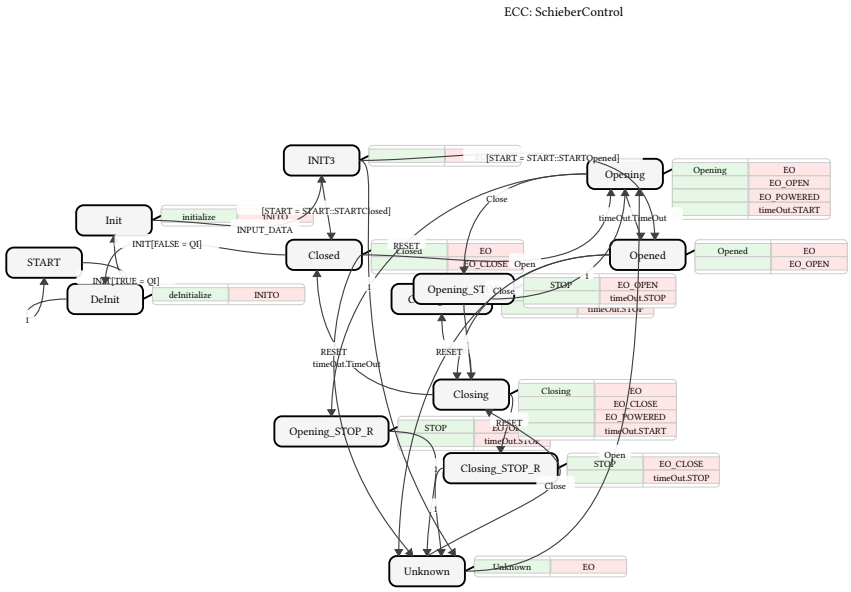
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

slider

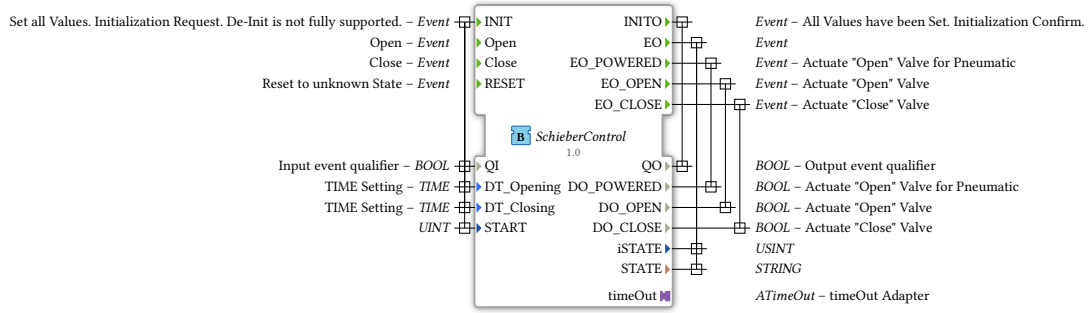
Slide Control



• – ○ • – ○ • – ○ •

Introduction

The function block `SchieberControl` is a basic function block (FB) according to IEC 61499 for controlling a slide actuator (e.g., pneumatic). It implements a complete state control (ECC) that manages the movement (open/close), holding states, and fault states of a slide. The block offers an interface for parameterization, operation via the `Open`, `Close`, and `RESET` events, as well as for outputting the state and control signals to the actuators.



Interface Structure

Event Inputs

- **`INIT`**: Initialization request. Sets all values. A complete de-initialization is not fully supported. Associated with the data `QI`, `DT_Opening`, `DT_Closing`, and `START`.
- **`Open`**: Starts the opening process of the slide.
- **`Close`**: Starts the closing process of the slide.
- **`RESET`**: Resets the internal state to “Unknown”.

Event Outputs

- **`INITO`**: Initialization confirmation. Linked to the data output `QO`.

- **EO**: General output event that provides the current system state and the configured output values. Linked to iSTATE and STATE.
- **EO_POWERED**: Control signal for actuating the "open" valve for the pneumatic system. Linked to DO_POWERED.
- **EO_OPEN**: Control signal for actuating the "open" valve. Linked to DO_OPEN.
- **EO_CLOSE**: Control signal for actuating the "close" valve. Linked to DO_CLOSE.

Data Inputs

- **QI (BOOL)**: Qualifier for the INIT event.
- **DT_Opening (TIME)**: Time duration for the opening process.
- **DT_Closing (TIME)**: Duration of the closing process.
- **START (UINT)**: Defines the desired start state after initialization (e.g., START::STARTClosed, START::STARTOpened, START::STARTUnknown).

Data Outputs

- **QO (BOOL)**: Qualifier for the INITO event.
- **DO_POWERED (BOOL)**: Binary signal to actuate the "open" valve for the pneumatics.
- **DO_OPEN (BOOL)**: Binary signal to actuate the "open" valve.
- **DO_CLOSE (BOOL)**: Binary signal to actuate the "close" valve.
- **iSTATE (USINT)**: Numeric value of the current internal state of the function block (output with EO).
- **STATE (STRING)**: String describing the current internal state of the function block (e.g., "Closed", "Opening").

Adapter

- **timeOut (ATimeOut)**: A plug adapter of type ATimeOut. Used to implement time-controlled state transitions (open/close). The function block (FB) starts (START) and stops (STOP) the timer and reacts to its TimeOut event.

Functionality

The SchieberControl FB operates as a state-controlled sequence. The internal control flow graph (ECC) defines the states Closed (Closed), Opened (Open), Opening (Opening), Closing (Closing), various STOP states, and Unknown (Unknown). Each state change triggers a corresponding algorithm that sets the data outputs (such as DO_OPEN, iSTATE, STATE) and, if necessary, starts the timer adapter. The movement between Closed and Opened is time-controlled via the states Opening and Closing. During a movement, a stop can be initiated by the opposite event (Open during Closing or Close during Opening), which transitions to the corresponding STOP state. From any state, a RESET event can move the function block to the Unknown state.

Technical Features

- **State Initialization**: The desired initial state (State_01, State_02, or State_03) can be specified via the START input after INIT.- **Adapter Usage**: The timing control is completely outsourced to the timeOut adapter, enabling a clear separation of functionality and potential reusability.
- **State Output**: The current state is output both numerically (iSTATE, USINT) and as text (STATE, STRING) with the EO event, so it can be evaluated by user interfaces.
- **Power Signal**: The signal DO_POWERED is only activated in states Opening and Opened, indicating a specific pneumatic control logic (e.g., maintaining compressed air in the open state).

State Overview

The ECC comprises the following main states:

1. **START**: Inactive initial state.
2. **Init/INIT3**: Initialization sequence.
3. **Closed**: Final state "closed". Activates DO_CLOSE.
4. **Opened**: Final state "open". Activates DO_OPEN and DO_POWERED.
5. **Opening/Closing**: Motion states. Start the timer with DT_Opening/DT_Closing and activate the corresponding actuator signals as well as DO_POWERED (only with Opening).
6. **Opening_STOP/Closing_STOP**: Stop states during movement, triggered by the opposite command. Deactivate DO_OPEN/DO_CLOSE and stop the timer.
7. **Opening_STOP_R/Closing_STOP_R**: Stop states during movement, triggered by RESET. This leads to the Unknown state.

8. **`Unknown`**: Error or unknown state. Resets all actuator signals.

Application Scenarios

- **Control of pneumatic slides** in packaging, conveying, or sorting systems.
- **Integration into higher-level sequencers** for material flow control.
- **Connection to operator panels (HMI)**, as the state is provided as a number (`iSTATE`) and text (`STATE`) for displays.
- **Logging and visualization** of the slide state via the `STATE` string output.

⚖️ Comparison with similar function blocks

Compared to simpler binary actuator function blocks (e.g., a simple cylinder function block), `SchieberControl` offers significantly higher functionality:

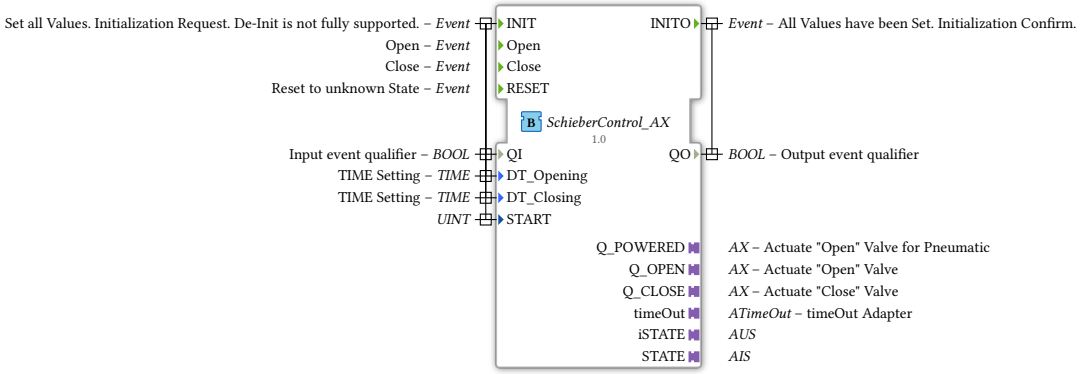
- **Complete state machine** with motion, end, and stop states.
- **Integrated timing control** for motion.
- **Extensive HMI interface** for configuring operating elements.
- **Explicit handling of an "unknown" state** (`RESET` functionality).

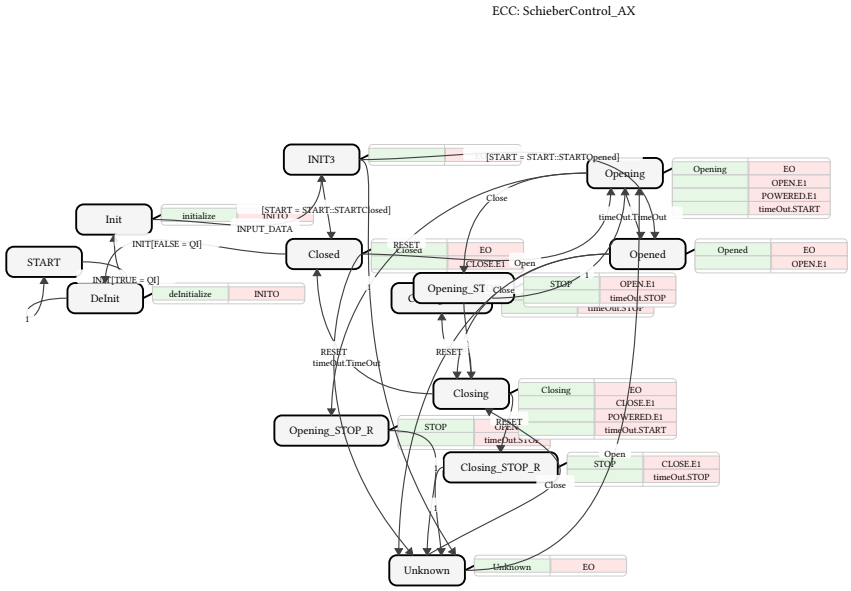
It is more specialized and complex than a generic `E_SR` (flip-flop) or a simple timer block, as it combines these functionalities and tailors them to the control of a slide valve actuator.

Conclusion

The `SchieberControl` is a robustly designed and functionally comprehensive control block for slide valve actuators. Its strengths lie in its clear state logic, the flexible configurability of the user interface, and the clean separation of control logic and timing functionality through adapters. It is well suited for use in medium to complex control applications where reliable and monitorable control of a slide valve is required.

SlideControl_AX





• – ○ • – ○ • – ○ •

Introduction

The function block **SlideControl_AX** is used to control a slide valve (valve, flap, or similar actuator) within a 61499-based control system. It implements a state machine that not only manages the logical states (Open, Closed, Opening, Closing) but also handles the timing of the movements and provides the current state via the adapters `STATE` and `iSTATE`.

This version of the block (“AX Adapter Version”) uses special adapters (`adapter::types::unidirectional::AX`) to control the physical valves.

Interface Structure

Event Inputs

Event	Type	Comment
INIT	EInit	Initialization request (with QI, DT_Opening, DT_Closing, START). Sets all values and parameters.
Open	Event	Command to open the slider.
Close	Event	Command to close the slider.
RESET	Event	Resets the function block to an unknown state (“Unknown”).

Event Outputs

Event	Type	Comment
INITO	EInit	Confirmation of initialization (with QO).

Data Inputs

Variable	Data Type	Comment
QI	BOOL	Input Event Qualifier (True = Normal operation, False = De-Init).

DT_Opening	TIME	Duration of the opening process.
DT_Closing	TIME	Duration of the closing process.
START	UINT	Start configuration (Defines the state in which the function block starts). Default: `STARTUnknown`.

Data Outputs

Variable	Data Type	Comment
QO	BOOL	Output Event Qualifier.

Adapter

Name	Type	Comment
Q_POWERED	AX	Adapter for controlling the "open" valve (pneumatic/main supply). Active during opening and in the open state.
Q_OPEN	AX	Adapter for controlling the "open" signal.
Q_CLOSE	AX	Adapter for controlling the "close" signal.
timeOut	ATimeOut	Adapter for timer functionality (monitors opening/closing times).
iSTATE	AUS	Adapter (`adapter::types::unidirectional::AUS`), outputs the current state as an `iSTATES` constant via `iSTATE.D1`.
STATE	AIS	Adapter (`adapter::types::unidirectional::AIS`), outputs the current state as a `STATES` constant via `STATE.D1`.

Functionality

This module operates as an event-controlled logic controller (ECC) that controls transitions based on the events `Open` and `Close`, as well as the timer adapter `timeOut`.

1. **Initialization:**

Upon startup (`INIT`), the initial state of the slider is checked (defined by `START`). Possible states include "Closed," "Opened," or "Unknown."

1. **Movement Sequence:**

- When the command `Open` is executed, the function block switches to the **Opening** state. This activates the adapters `Q_POWERED` and `Q_OPEN`, and starts the timer with `DT_Opening`.
- After the specified time (`timeOut.TimeOut`), the state automatically switches to **Opened**.
- – If the command `Close` is issued, the function block switches to **Closing**. The adapter `Q_CLOSE` is activated (while `Q_POWERED` and `Q_OPEN` are deactivated), and the timer is started with `DT_Closing`.
- After the specified time has elapsed, the state changes to **Closed**.

1. **Interruption:**

If the command `Close` (or vice versa) is issued during the opening process, the process is stopped (`STOP` states), the outputs are reset, and the reverse process is initiated.

1. **Visualization:**

In each state (Closed, Opening, Opened, Closing, Unknown), the adapters `STATE` (constant from `STATES`) and `iSTATE` (constant from `iSTATES`) are written with the current state. This enables dynamic adaptation of the user interface by downstream blocks.

Technical Features

- **AX Adapter Integration:** The direct use of `adapter::types::unidirectional::AX` indicates a standardized interface for hardware abstraction, making the function block reusable for various valve types, provided the adapter is compatible.

- **State Output:** The function block outputs the current state as constants (`STATES`, `iSTATES`) via the adapters `STATE` (AIS) and `iSTATE` (AUS). This reduces the logic in the HMI.
- **Stop Logic:** Explicit `STOP` states are implemented to ensure that, when changing direction, the outputs are briefly and precisely switched off (`timeOut.STOP`, Valves Off) before the new direction is initiated.

State Overview

The most important states in the ECC (Execution Control Chart) are:

- **START / Init / DeInit:** Management states for the block's lifecycle.
- **Unknown:** Error or initial state when the position is unknown.
- **Closed:** The valve is fully closed. (`Q_CLOSE`=False, `Q_OPEN`=False).
- **Opening:** The slider is currently opening. (`Q_POWERED`=True, `Q_OPEN`=True, timer running).
- **Opened:** The slider is fully open. (`Q_POWERED`=True, `Q_OPEN`=False).
- **Closing:** The slider is currently closing. (`Q_CLOSE`=True, `Q_POWERED`=False, timer running).
- **..._STOP:** Intermediate states for cleanly stopping movements.

Application Scenarios

- **Agricultural Machinery:** Control of slurry scrapers, metering flaps, or hydraulic booms.
- **Process Automation:** Simple valve controls that do not have end-position sensors but operate based on time (`DT_Opening`/`DT_Closing`).
- **HMI Integration:** Systems where the symbol or button color on the display must change depending on whether the valve is moving, open, or closed.

Comparison with Similar Modules

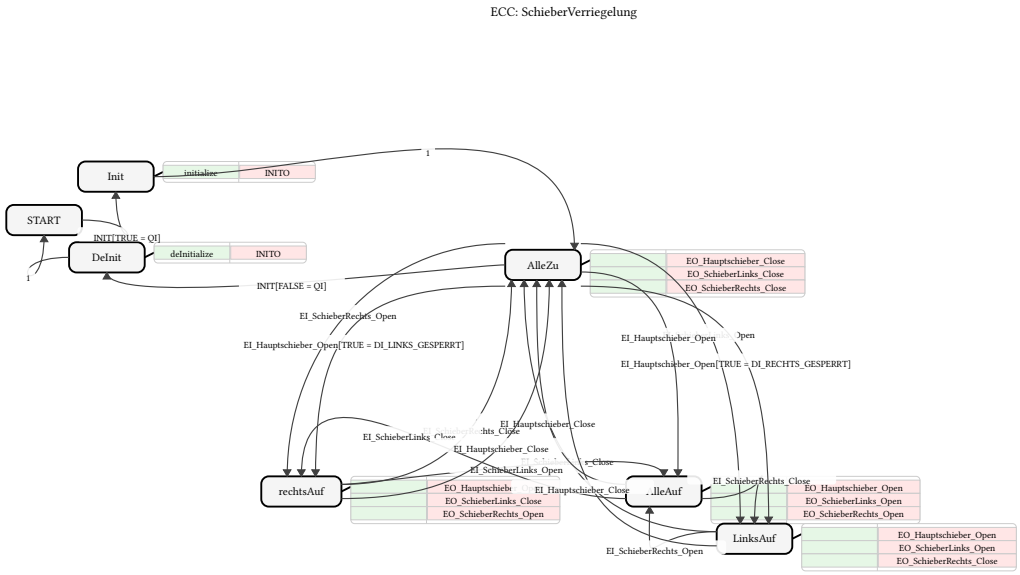
In contrast to a simple `SR`The **SchieberControl_AX** offers the following flip-flop functionality for valve control:

1. **Time Monitoring:** Integrated timers simulate the runtime.
2. **State Output:** Built-in output of the current state for UI information.
3. **Hardware Abstraction:** Uses adapters instead of direct Boolean outputs for the valves.
4. **Intermediate States:** Explicit representation of "Opens" and "Closes," not just "On" and "Off."

Conclusion

The `SchieberControl_AX` is a specialized, robust function block for time-controlled actuators. It encapsulates the complexity of state transitions, timer handling, and UI status management in a single component, thus significantly simplifying the development of control software for machines with hydraulic or pneumatic valves.

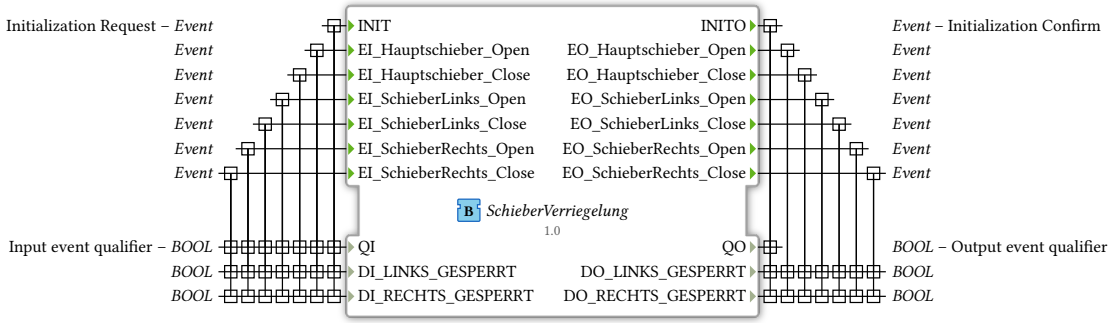
Slide Lock



• – ○ • – ○ • – ○ •

Introduction

The function block `SchieberVerriegelung` is used for the coordinated control and locking of three slides: a main slide, a left slide, and a right side slide. Its primary purpose is to ensure valid and collision-free slide combinations based on external requests (events) and locking states. This block is particularly suitable for applications where the movement of one slide must prevent or force the movement of another.



Interface Structure

Event Inputs

- **`INIT`**: Initialization request. Triggers the transition to the initialized state. Linked to the data `QI`, `DI_LINKS_GESPERRT`, and `DI_RECHTS_GESPERRT`.
- **`EI_Hauptschieber_Open`**: Requests the main slider to open.
- **`EI_Hauptschieber_Close`**: Requests the main slider to close.
- **`EI_SchieberLinks_Open`**: Requests the left side slider to open.
- **`EI_SchieberLinks_Close`**: Requests the left side slider to close.
- **`EI_SchieberRechts_Open`**: Requests the right side slider to open.
- **`EI_SchieberRechts_Close`**: Requests the right side slider to close.

Event Outputs

- **`INITO`**: Initialization confirmation. Triggered after completion of initialization (`INIT`) or deinitialization. Associated with the data `QO`, `DO_LINKS_GESPERRT`, and `DO_RECHTS_GESPERRT`.
- **`EO_Hauptschieber_Open`**: Signals the command to open the main slide.
- **`EO_Hauptschieber_Close`**: Signals the command to close the main slide.
- **`EO_SchieberLinks_Open`**: Signals the command to open the left side slide.
- **`EO_SchieberLinks_Close`**: Signals the command to close the left side slide.
- **`EO_SchieberRechts_Open`**: Signals the command to open the right side slider.
- **`EO_SchieberRechts_Close`**: Signals the command to close the right side slider.

Data Inputs

- **`QI` (BOOL)**: Qualifies the INIT event. `TRUE` starts initialization, `FALSE` starts deinitialization.
- **`DI_LINKS_GESPERRT` (BOOL)**: Signals the locked state of the left side slider (`TRUE` = locked).
- **`DI_RECHTS_GESPERRT` (BOOL)**: Signals the locked state of the right side slider (`TRUE` = locked).

Data Outputs

- **`QO` (BOOL)**: Status output reflecting the success of initialization/deinitialization.
- **`DO_LINKS_GESPERRT` (BOOL)**: Outputs the internally processed or forwarded locked state for the left slider.
- **`DO_RECHTS_GESPERRT` (BOOL)**: Outputs the internally processed or forwarded locked state for the right slider.

Adapters

This function block does not use any adapter interfaces.

Functionality

The `SchieberVerriegelung` function block is implemented as a Basic Function Block and follows a defined state machine (ECC). After initialization, the block starts in state `AlleZu` (all sliders closed). From here, transitions to other states can occur depending on the incoming event and the current lock states (`DI_LINKS_GESPERRT`, `DI_RECHTS_GESPERRT`).

The `SchieberVerriegelung` function block is implemented as a Basic Function Block and follows a defined state machine (ECC). After initialization, the block starts in state `AlleZu` (all sliders closed). The central logic lies in the interpretation of the `EI_Hauptschieber_Open` event in state `AlleZu`:

1. If **no** side slider is locked (`DI_LINKS_GESPERRT = FALSE` and `DI_RECHTS_GESPERRT = FALSE`), the function block transitions to state `AlleAuf` (all sliders open).
2. If **only the right slider** is locked (`DI_RECHTS_GESPERRT = TRUE`), the function block transitions to state `LinksAuf` (main and left sliders open, right slider remains closed).
3. If **only the left** slider is locked (`DI_LINKS_GESPERRT = TRUE`), the function block (FB) transitions to state `rechtsAuf` (main and right sliders open, left one remains closed).

States `LinksAuf` and `rechtsAuf` represent the locked operating states. From these states, the system can transition to state `AlleAuf` by selectively opening/closing the side sliders, or return to state `AlleZu` by closing the main slider.

The algorithm `normalOperation` copies the blocking states from the inputs (`DI_*_GESPERRT`) to the outputs (`DO_*_GESPERRT`) when operation is enabled (`QI=TRUE`).

Technical Features

- **State-Controlled Output**: Each operational state (`AlleZu`, `AlleAuf`, `LinksAuf`, `rechtsAuf`) immediately triggers a fixed combination of output events (`EO_*`) that define the desired slider position. ... * **Conditional Transitions**: The transition from `AlleZu` to `LinksAuf`/`rechtsAuf` is linked to the `EI_Hauptschieber_Open` event **and** the corresponding lock state (`DI_*_GESPERRT`). This represents low-level locking logic.
- **Explicit Deinitialization**: A `INIT` event with `QI=FALSE` transitions from any state to the `DeInit` state and sets the output `QO` to `FALSE`.

State Overview

1. **START**: Inactive initial state before initialization.
2. **Init**: Triggered at `INIT` with `QI=TRUE`. Executes the `initialize` algorithm and confirms with `INITO`.
3. **AllClosed** (Default operating state): All three sliders are closed. Sends `Close` commands for all sliders.

4. **AllOpen**: All three sliders are open. Sends `Open` commands for all sliders.
5. **LeftOpen**: Main slider and left side slider are open; the right side slider is closed (locked). Sends corresponding `Open`/`Close` commands.
6. **RightOpen**: The main slide and right side slide are open, the left one is closed (locked). Sends corresponding `Open`/`Close` commands.
7. **DeInit**: Deinitialization state. Sets `QO` to `FALSE` and then reverts to `START`.

Application Scenarios

Typical applications can be found in distribution and conveying systems, for example, in agricultural technology or material logistics:

- **Grain or bulk material conveyors**: The main slide conducts the flow. The side gates can be opened as needed, for example, to distribute material into different silos. The interlock prevents both sides from being open simultaneously if this is mechanically or process-related impermissible.
- **Switch control**: Analogous to a mechanical switch, where the position of one switch shoe blocks the other position.

⚖️ Comparison with similar components

Compared to simple gate control components (e.g., individual `E_SR` flip-flops per gate), the `SchieberVerriegelung`-FB:

- **Integrated Collision Avoidance**: The interlock logic is hard-coded within the state machine and does not require external wiring.

** **State-Based Coordination**: The output commands are always consistent sets (`Open`/`Close` combinations for all three slides).

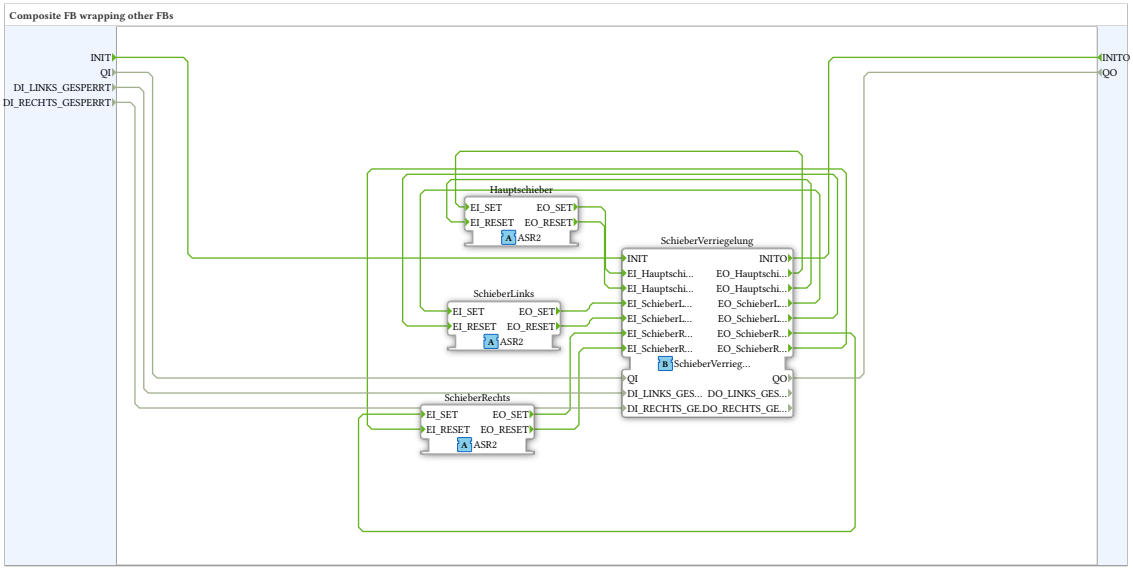
- **Explicit Lock Inputs**: The consideration of external lock signals (`DI_*_GESPERRT`) is an integral part of the control logic.

A disadvantage is the reduced flexibility. The logic is specific to exactly three slides with this particular locking relationship. For other numbers or dependencies, a new function block must be created.

Conclusion

The `SchieberVerriegelung` function block is a specialized control block for the coordinated control of three mechanically or process-coupled slides. Its strength lies in the robust, state-based implementation of the locking logic, which guarantees secure and consistent slide positions. It is ideally suited for clearly defined plant sections with clear dependencies between the actuators and relieves the higher-level control design of the wiring of this safety logic.

SlideLockComposite



• – ○ • – ○ • – ○ •

Introduction

The function block `SchieberVerriegelungComposite` is a composite function block that serves as a wrapper for other function blocks. Its main purpose is to manage and coordinate the locking logic for multiple slides (main slide, left slide, right slide). It encapsulates the internal logic and provides a unified interface for initialization and data exchange with the connected actuators and sensors.



Interface Structure

Event Inputs

- **`INIT`**: Initialization request. Triggers the initialization process of the internal `SchieberVerriegelung` function block and thus of the entire composite block. Linked to the data `QI`, `DI_LINKS_GESPERRT`, and `DI_RECHTS_GESPERRT`.

Event Outputs

- **`INITO`**: Initialization confirmation. Triggered by the internal `SchieberVerriegelung` function block, it signals successful completion of the initialization. Linked to the data output `QO`.

Data Inputs

- **`QI` (BOOL)**: Qualifier for the INIT event. Controls the initialization.
- **`DI_LINKS_GESPERRT` (BOOL)**: Status input indicating whether the left slider is locked.
- **`DI_RECHTS_GESPERRT` (BOOL)**: Status input indicating whether the right slide is locked.

Data Outputs

- **`QO` (BOOL)**: Qualifier for the INITO event. Returns the initialization status.

Adapters

The composite FB provides three bidirectional adapter interfaces for communicating with the physical or logical slide actuators. Each adapter follows the type `adapter::types::bidirectional::ASR2`, which typically supports SET and RESET commands and corresponding feedback.

1. **`Hauptschieber`**: Adapter for controlling and providing status feedback to the main slide.
2. **`SchieberLinks`**: Adapter for controlling and providing status feedback to the left slide.
3. **`SchieberRechts`**: Adapter for controlling and providing status feedback to the right slide.

Functionality

The `SchieberVerriegelungComposite` primarily acts as an intermediary. It forwards all external events and data to the internal, encapsulated function block `SchieberVerriegelung`. Likewise, the output events and data of this internal function block are passed through to the composite interfaces.

The central logic for controlling and locking the slides resides entirely within the internal function block `SchieberVerriegelung`. The composite block translates the generic adapter events (`EI_SET`/`EI_RESET`, `EO_SET`/`EO_RESET`) into the specific, named events of the internal function block (e.g., `EI_Hauptschieber_Open`, `EO_SchieberLinks_Close`) and vice versa. This enables a clean separation between application-specific control logic and generic adapter communication.

Technical Features

- **Encapsulation**: The complex interlock logic is implemented in a separate function block (`SchieberVerriegelung`), which promotes reusability and testability.

- **Adapter-based communication**: Connection to the actuators is exclusively via standardized adapter interfaces (`ASR2`). This abstracts the specific actuators and increases flexibility.
- **Pass-through design**: Except for translating the adapter event names, this composite FB has no logic of its own. It serves to structure the FBNetwork.

State overview

As a composite FB, `SchieberVerriegelungComposite` does not have an explicit state machine itself. The system state is managed entirely by the internal `SchieberVerriegelung` FB. The composite block can be in one of the following two macro states:

- **Not initialized**: Before the arrival of a valid `INIT` event.
- **Initialized and ready for operation**: After the internal function block acknowledges the initialization via `INITO`. In this state, all adapter events are passed through.

Application Scenarios

This function block is designed for control applications where multiple mechanically or logically linked slides or closures need to be coordinated, e.g.:

- Interlocking systems in conveyor systems or silos.
- Safety controls for gates or flaps that may only be opened under certain conditions.
- Process controls where the sequence of switching operations must be maintained (e.g., main slides may only open if side closures are closed).

⚖️ Comparison with similar function blocks

Unlike a simple, monolithic function block that mixes logic and interfaces, this composite block offers a clear separation of concerns. A direct comparison would be a `SchieberVerriegelung` function block, which would have integrated the adapter interfaces directly. The composite approach is more modular and allows the core logic (`SchieberVerriegelung`) to be reused unchanged in different network environments by simply adapting the enclosing composite block.

Conclusion

The `SchieberVerriegelungComposite` is a well-structured wrapper block that simplifies the integration of complex slider locking logic into a larger 4diac control system. By using standardized adapters and encapsulating the core functionality, it promotes reusability, maintainability, and clear interface definitions. It is ideal for applications requiring reliable and transparent coordination of multiple actuators.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

const

START

🎧 Podcast



- Events and IEC 61499: The Starting Signal for Intelligent Systems

Introduction

The `START` constant declaration defines a set of integer constants (of type `UINT`) for use in control programs. These constants represent specific states typically used in sequence controls, such as for sliders. They improve code readability and maintainability by replacing numeric values with meaningful names.

Interface Structure

START` is a **GlobalConstants** declaration and not a function block. Therefore, it does not have event- or data-flow-based interfaces such as inputs or outputs. Instead, it defines global, constant values that can be referenced within other function blocks or programs.

Event Inputs

Not applicable. This declaration does not contain any event inputs.

Event Outputs

Not applicable. This declaration does not contain any event outputs.

Data Inputs

Not applicable. This declaration does not contain any data inputs.

Data Outputs

Not applicable. This declaration does not contain any data outputs.

Adapters

Not applicable. This declaration does not use adapters.

Functionality

The `START` constant declaration assigns fixed, unchanging numerical values to the symbolic names `STARTClosed`, `STARTOpened`, and `STARTUnknown`. Once this declaration is included in a project, the defined names can be used throughout the application code instead of the raw numbers (5, 6, 7). During compilation, the compiler replaces any reference to, for example, `STARTClosed` with the value `UINT#5`.

Technical Details

- **Type:** All constants are of type `UINT` (unsigned integer).
- **Scope of Validity:** The constants are defined globally (`VAR_GLOBAL CONSTANT`) and available within the declared package (`logiBUS::utils::sequence::schieber::const`).
- **Immutability:** As constants (`CONSTANT`), their values cannot be changed at runtime.
- **Package Structure:** The constants are organized in a specific package hierarchy that reflects their application domain (auxiliary functions for sequence control of sliders).

State Overview

The constants themselves represent abstract states:

- `STARTClosed` (value 5): Represents the state “closed” or “in starting position”.
- `STARTOpened` (value 6): Represents the state “open” or “in end position”.
- `STARTUnknown` (value 7): Represents an undefined, unknown, or error state.

Application Scenarios

Primarily, these constants are used in state machines or sequencer function blocks to uniquely and readably encode the current position or status of an actuator (such as a linear slider). For example, a function block could set its internal state or an output value to `STARTOpened` to indicate that the slider is fully open.

⚖️ Comparison with Similar Function Blocks

Unlike function blocks such as `E_SR` (flip-flop) or `E_CYCLE` (cycle encoder), `START` has no algorithm or behavioral logic. It is a pure data declaration. Similar constructs would be other `GlobalConstants` or `Type` declarations, which also provide symbolic names for values (e.g., for error codes or operating modes).

Conclusion

The `START` constant declaration is a simple yet essential tool for structuring and professionalizing IEC 61499-based control applications. By using meaningful names instead of “magic numbers,” the code becomes significantly more understandable, less error-prone, and easier to maintain. Its integration is particularly recommended in larger projects or reusable library components.

STATES

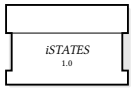
States (Closed, Opened, Transitions)

Global Constants

Name	Type	Initial Value
Closed	STRING	‘Closed’
Opening	STRING	‘Opening’
Opened	STRING	‘Opened’
Closing	STRING	‘Closing’
Unknown	STRING	‘Unknown’



iSTATES



• – ◦ • – ◦ • – ◦ •

Introduction

The **iSTATES** block is a special GlobalConstants block in the 4diac IDE. It defines a set of named constants that represent the possible states of a (slider) actuator. The constants are stored as `USINT` values and enable a uniform, human-readable reference to the states in the control logic.

Interface Structure

Event Inputs

None. The block has no event inputs.

Event Outputs

None. The block has no event outputs.

Data Inputs

None. The block has no data inputs.

Data Outputs

None. The function block has no data outputs. The defined constants are available as globally accessible symbols within the project context.

Adapter

None.

Functionality

The **iSTATES** function block provides five symbolic constants that can be used throughout the application. Each constant represents a discrete state of a slider or similar actuator. The values are defined as `USINT` (unsigned short integer) and are assigned as follows:

Constant	Value (USINT)	Meaning
Unknown	0	Unknown state

Closed	1	Closed
Opening	2	Opens
Opened	3	Opened
Closing	4	Closes

These values can be used directly in FB networks for initializing variables or for comparison with status feedback.

Technical Features

- **Type: GlobalConstants** – The function block is not instantiated but integrated into the project as a global constant definition.
- **Value Range**: All states are declared as `USINT`, which guarantees low memory usage and compatibility with many fieldbus protocols.
- **Extensibility**: The scheme can easily be extended with additional states if the application requires it.
- **Package**: The function block is organized in the package `logiBUS::utils::sequence::schieber::const`, which allows for clear structuring within the project.

State Overview

The five states cover a complete lifecycle of a slider:

1. **Unknown** (0) – Initial state or error state if the current position is unknown.
2. **Closed** (1) – Slide is closed (end position closed).
3. **Opening** (2) – Slide is in the opening phase.
4. **Opened** (3) – Slide is fully open (end position open).
5. **Closing** (4) – Slide is in the closing phase.

Application Scenarios

- **State Function Blocks for Slides, Valves, or Gates** – This function block provides the basis for state management in sequential controllers.
- **Visualization** – The symbolic names can be used in HMI or SCADA systems to display the current state.
- **Fault Detection** – The `Unknown` state allows for the unambiguous identification of invalid or initial states.

Comparison with Similar Blocks

Similar GlobalConstants blocks often exist in other packages (e.g., for motor controllers, conveyor systems). The difference lies in the specific definition of the states – while other blocks use `Idle`, `Running`, `Error`, etc., **iSTATES** is specifically tailored to the five typical slider states. Using `USINT` instead of `BOOL` or `INT` is platform-efficient and standards-compliant.

Conclusion

The **iSTATES** block is a simple yet useful tool for the consistent definition of slider states. Using global constants makes the code more readable, fault-resistant, and maintainable. It is ideally suited for all applications where an actuator cycles through the state sequence “Closed → Opens → Open → Closes → Closed”.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

str

🎧 Podcast

- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Sharpening Target Groups, Structure, and Unique Selling Proposition

- Taming the Software Dragon: Industrial Automation and the Future of Production
- Eclipse 4diac FORTE: Understanding IEC 61499 – The LEGO® Building Kit for Your Industry 4.0 Control System

---

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

SliderAuxInStruct

• – ◦ • – ◦ • – ◦ •

Introduction

The structured data type `SchieberAuxInStruct` is used to bundle information for the visual representation of an auxiliary element (AUX) in a slider control. It combines the specification of an image with a background color to define the graphical representation of an auxiliary input or element.

Interface Structure

This function block is a structured data type (struct) and therefore has no event or adapter interfaces. Its structure consists of the following two data elements:

Data Inputs

Since it is a data type, there are no direct inputs. The structure is used as a whole at the data inputs of other function blocks.

Data Outputs

Since it is a data type, there are no direct outputs. The structure is used as a whole at the data outputs of other function blocks.

Structured Elements

- **`Picture`** (`logiBUS::utils::sequence::schieber::SchieberStruct`):
- **Comment:** `u16ObjId for Picture`
- **Description:** Contains the identifier (Object ID) for the image to be displayed in the helper element. The underlying type `SchieberStruct` likely provides the necessary information for referencing and handling the image object.
- **`u8Colour`** (`logiBUS::utils::sequence::schieber::SchieberColorStruct`):
- **Comment:** `Background Color of the AUX`
- **Description:** Defines the background color of the helper element. The type `SchieberColorStruct` specifies the color used, likely in a format suitable for the target system (e.g., an HMI).

Functionality

The data type `SchieberAuxInStruct` is a passive data structure. Its sole function is to logically combine the two properties `Picture` (image) and `u8Colour` (background color) into a single, reusable data record. It is used to transport these visual attributes consistently between different functional blocks within an application.

Technical Details

- **Namespaces:** The data type and its components are defined in the package `logiBUS::utils::sequence::schieber`, indicating a specific application in agricultural machinery control technology.
- **Typing:** Using specific, user-defined types (`SchieberStruct`, `SchieberColorStruct`) for members promotes type safety and reusability throughout the project.

State Overview

As a pure data type, `SchieberAuxInStruct` has no internal state or state machine.

Application Scenarios

This data type is typically used in PLC (Programmable Logic Controller) controllers for mobile machinery, particularly in:

- **HMI Configuration:** To define the appearance of operating or display elements (e.g., sliders, buttons, status lights) on an operator panel.
- **Sequence Controls:** Within sequential control systems where the display of an element (color, image) should change depending on the process state.
- **Data Encapsulation:** Simplifies the parameterization of function blocks that control the appearance of an "AUX" element by allowing all visual parameters to be passed in a single variable.

⚖️ Comparison with Similar Building Blocks

Unlike function blocks (FBs), this building block has no algorithm or behavioral logic. It is comparable to a `STRUCT` in IEC 61131-3 or a `class`/`struct` in higher-level programming languages. Other similar elements would be simple data types (e.g., `ARRAY` of single values), but these do not offer the semantic grouping and type safety of this structured type.

Conclusion

The `SchieberAuxInStruct` is a fundamental, reusable data type for modeling the visual properties of a helper element in a specific control environment. By encapsulating image and color information, it contributes to better structuring, readability, and maintainability of the application code. Its use is particularly advantageous in projects with complex user interfaces or state-dependent visualizations.

SliderColorStruct

• – ◦ • – ◦ • – ◦ •

Introduction

The data type `SchieberColorStruct` is a structured data structure used to manage color values for different states of a slider (e.g., a valve, flap, or closure). It defines a consistent color coding for the visual representation of the states `Unbekannt`, `Geschlossen`, `Öffnend`, `Geöffnet`, and `Schließend` in a user interface or display.

Interface Structure

The data type is a structure (`STRUCT`) and does not contain any events or adapters. Its interface consists exclusively of data elements.

Data Inputs

This structure does not have event inputs in the conventional sense, as it is a data type. The following elements are the structure's member variables, which can be written to or assigned values during instantiation.

Data Outputs

This structure does not have event outputs. The member variables represent the stored data.

Data Elements (Structure Members)

Name	Data Type	Comment	Initial Value
`Unknown`	`USINT`	u8Colour for Unknown State	`COLOR_WHITE`
`Closed`	`USINT`	u8Colour for Closed State	`COLOR_WHITE`
`Opening`	`USINT`	u8Colour for Opening State	`COLOR_WHITE`
`Opened`	`USINT`	u8Colour for Opened State	`COLOR_WHITE`
`Closing`	`USINT`	u8Colour for Closing State	`COLOR_WHITE`

Adapters

This data structure does not use adapters.

Functionality

The `SchieberColorStruct` is a pure data type. It serves as a container to group five different color values (each as an 8-bit unsigned integer, `USINT`) for the five defined slider states. By using this structure, a function block or application can pass or store a single, consistent variable containing all the colors

required for display. The initial values of all members are set to the constant `COLOR_WHITE`, which comes from the imported package `isobus::UT::Q::const::colours`.

Technical Details

- **Imported Constant:** The structure depends on the externally defined constant `COLOR_WHITE`, which is provided via the compiler import `isobus::UT::Q::const::colours::COLOR_WHITE`. This ensures that a consistent, project-wide shade of white is used.
- **Type Safety:** Using a structured type instead of five individual variables improves code readability and maintainability and reduces the likelihood of errors.
- **Hash Attribute:** The structure contains the attribute `eclipse4diac::core::TypeHash` with an empty value. This is used by the 4diac system for type identification and management.

State Overview

Because it is a static data type, `SchieberColorStruct` itself does not have a state machine. Instead, it encodes the display attributes for the states of another, parent element (e.g., a `Schieber` function block).

Application Scenarios

- **Visualization in HMIs:** Passing the color configuration to visualization components to correctly display the state of a slider using color (e.g., red for "Closed," green for "Open").
- **Function Block Configuration:** A function block that controls a physical slider can have an instance of this type as a configurable input to enable user-defined colors for state feedback.
- **Data Encapsulation:** Bundling all state-related color information into a single parameter that can be exchanged between different software modules (e.g., controller and visualization).
-

⚖️ **Comparison with Similar Building Blocks**

Unlike function blocks (FBs) such as `E_SR` or `E_D_FF`, which implement logic, `SchieberColorStruct` is a pure **data type** (STRUCT). Similar structured types could be, for example, `MotorConfigStruct` (for configuration data) or `AxisPositionStruct` (for multidimensional positions). Its specific function is to define a color scheme for a predefined set of states.

Conclusion

The `SchieberColorStruct` is a simple yet useful structured data type that standardizes the management and exchange of color values for representing the state of a slider. It promotes clean interfaces, facilitates reuse, and contributes to consistency within a project through the use of initial default values and imported constants.

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

SlideStruct

• – ◦ • – ◦ • – ◦ •

Introduction

The `SchieberStruct` is a structured data type used to represent all possible states of a slide (e.g., a valve, flap, or closure) within a control system. It serves as a central container for the object identifiers (IDs) assigned to the various states of the slide. This data type is typically used in conjunction with function blocks that implement state machines or sequences for slide control.

Interface Structure

The `SchieberStruct` is a data type and therefore does not have event or adapter interfaces in the conventional sense. Its structure consists exclusively of data fields.

Data Inputs

Not applicable. The structure defines a data type, not function block interfaces.

Data Outputs

Adapter

Not applicable.

Functionality

The `SchieberStruct` is a data structure containing five fields of type `UINT` (16-bit unsigned integer). Each field represents a specific state of a slider and stores the corresponding object ID (`u16ObjId`). These IDs are used to access the graphical or logical representations of the states in a higher-level system (e.g., a visualization or logic module).

The initial values of all fields are set to the constant `ID_NULL`, indicating an undefined or unassigned state. These must be populated with valid object IDs before using the structure.

Technical Features

- **Initialization:** All member variables are initialized with the default value `ID_NULL`. This ensures that the structure starts in a defined, neutral state before specific IDs are assigned.
- **Type Safety:** Using a structured data type instead of individual, loosely coupled variables improves code maintainability and readability.
- **Compilation Note:** The data type is defined in the package `logiBUS::utils::sequence::schieber` and imports the constant `ID_NULL` from the `isobus::UT::Q::const::IDs` namespace.

State Overview

The structure explicitly maps the following five states of a slider:

1. **`Unknown`:** The slider's state is unknown or could not be determined.
2. **`Closed`:** The slide is in the closed end position.
3. **`Opening`:** The slide is in the process of opening (movement state).
4. **`Opened`:** The slide is in the open end position.
5. **`Closing`:** The slide is in the process of closing (movement state).

Application Scenarios

- **State Management:** A higher-level control block can maintain an instance of this structure to centrally manage the currently valid object IDs for all slide states.
- **Parameterization:** The structure can be passed as a configurable parameter to a function block, which then selects the corresponding ID from the structure based on its internal state and forwards it, for example, to a visualization system. * **Visualization:** An HMI or SCADA system can use the IDs stored in the structure to update the slider's graphical symbol according to its current state (e.g., color, symbol).

⚖️ Comparison with similar blocks

Unlike function blocks such as `E_SR` (memory element) or `E_CTUD` (counter), which implement active logic, `SchieberStruct` is a passive data type. Similar structured data types could be defined for other devices with state machines (e.g., `PumpeStruct`, `MotorStruct`). The unique feature lies in the specific mapping of the states characteristic of a linear slider: `Opening`, `Closing`, `Opened`, and `Closed`, including a `Unknown` state.

Conclusion

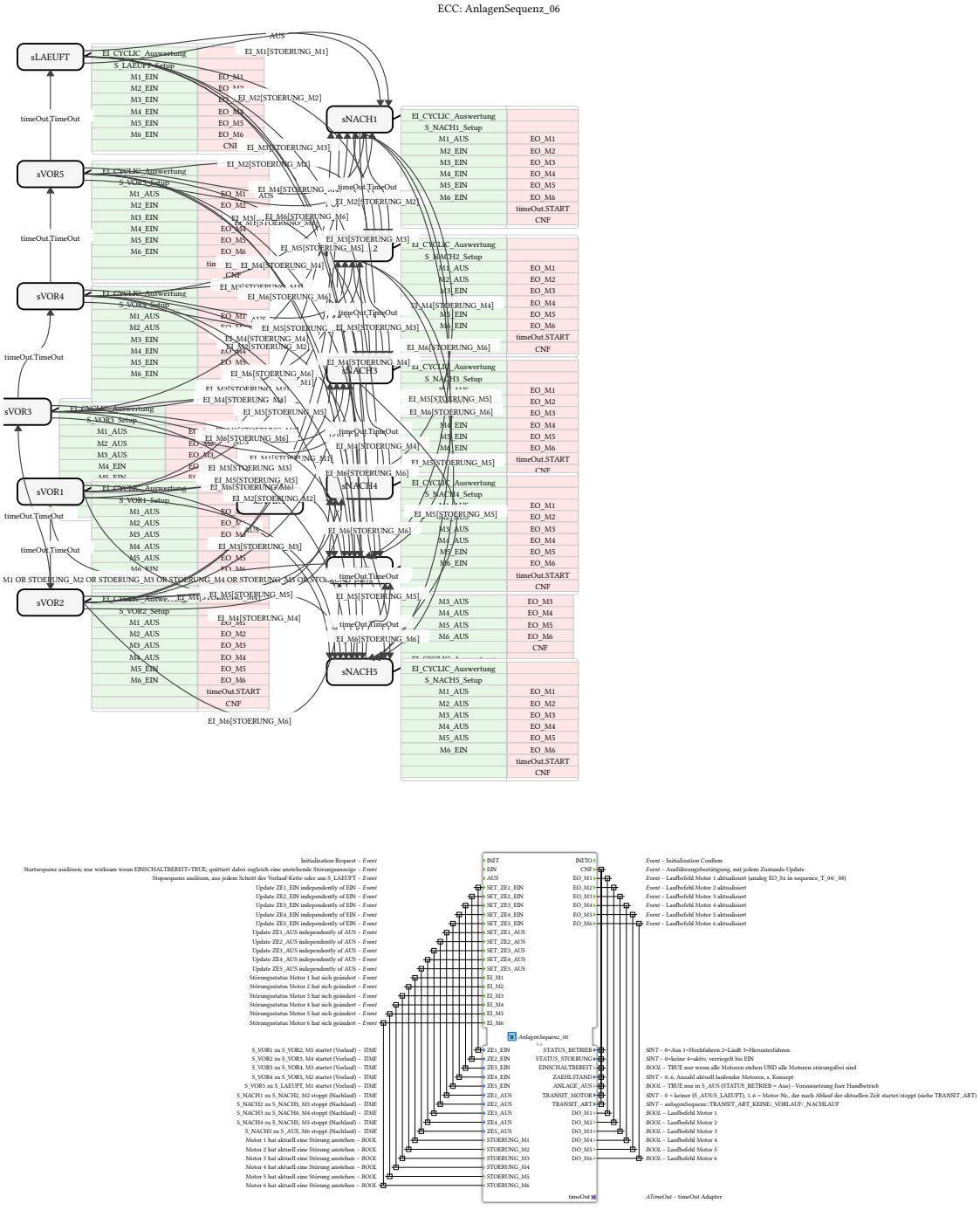
The `SchieberStruct` data type is a useful construct for the structured and type-safe handling of state information for slider actuators. By encapsulating the relevant object IDs in a single structure, the configuration and maintenance of applications that control and visualize such actuators is simplified. Its use promotes a clear and consistent programming style.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter module on ms-muc-docs.de

timed

AnlagenSequenz_06



• — ○ • — ○ • — ○ •

Introduction

The `AnlagenSequenz_06` function block is a time-controlled ring sequencer for the ordered start-up and shutdown sequence of six motors (M1..M6), as needed e.g. for a grain-intake conveyor chain. Unlike the generic sequence_T_04/ sequence_T_08 blocks, it is not a general-purpose sequencer but hard-wired to this one application pattern: start-up (motors switch on one by one, back to front) and shutdown (motors switch off one by one, front to back) are two separate, purely linear chains, joined into a closed ring by the two ring poles `sAUS` (bottom, all motors off) and `sLAEUFT` (top, all motors on). The block additionally implements a complete fault cascade: if any of the six motors reports a fault during start-up, shutdown, or

continuous operation, the FB jumps in a single step (no re-evaluation over multiple cycles) to the matching shutdown state, taking into account exactly the upstream motors that are still running at that moment.

Interface Structure

Event Inputs

- **INIT**: Initialization request (type EInit).
- **EIN**: Starts the start-up chain (transition sAUS → sVOR1). Only effective when EINSCHALTBEREIT = TRUE (condition EIN[NOT (STOERUNG_M1 OR ... OR STOERUNG_M6)]); also acknowledges a pending, latched STATUS_STOERUNG indication.
- **AUS**: Triggers the shutdown sequence from any start-up step (sVOR1..sVOR5) or from sLAEUFT. Jumps to the mirror point of the shutdown chain (sVOR_k → sNACH_(6-k)), since this is an operator request, not a motor fault.
- **SET_ZE1_EIN, SET_ZE2_EIN, SET_ZE3_EIN, SET_ZE4_EIN, SET_ZE5_EIN**: Update ZE1_EIN .. ZE5_EIN independently of EIN; each event carries (With) the identically named time variable (SET_ZEk_EIN → ZEk_EIN).
- **SET_ZE1_AUS, SET_ZE2_AUS, SET_ZE3_AUS, SET_ZE4_AUS, SET_ZE5_AUS**: Update ZE1_AUS .. ZE5_AUS independently of AUS; each event carries (With) the identically named time variable (SET_ZEk_AUS → ZEk_AUS).
- **EI_M1**: Motor 1 fault status has changed. Leads (With STOERUNG_M1) directly to sNACH1 from every state that precedes it.
- **EI_M2**: Motor 2 fault status has changed. Leads (With STOERUNG_M2) directly to sNACH2.
- **EI_M3**: Motor 3 fault status has changed. Leads (With STOERUNG_M3) directly to sNACH3.
- **EI_M4**: Motor 4 fault status has changed. Leads (With STOERUNG_M4) directly to sNACH4.
- **EI_M5**: Motor 5 fault status has changed. Leads (With STOERUNG_M5) directly to sNACH5.
- **EI_M6**: Motor 6 fault status has changed. Leads (With STOERUNG_M6) directly to sAUS (M6 is the last/outermost motor in the chain, so there is no shorter shutdown step).

Event Outputs

- **INITO**: Initialization confirmation (type EInit).
- **CNF**: Execution confirmation, triggered on every state change; carries STATUS_BETRIEB, STATUS_STOERUNG, ZAEHLSTAND, EINSCHALTBEREIT, ANLAGE_AUS, TRANSIT_MOTOR, and TRANSIT_ART.
- **EO_M1**: Motor 1 run command updated; carries DO_M1 (analogous to EO_Sx in sequence_T_04/_08).
- **EO_M2**: Motor 2 run command updated; carries DO_M2.
- **EO_M3**: Motor 3 run command updated; carries DO_M3.
- **EO_M4**: Motor 4 run command updated; carries DO_M4.
- **EO_M5**: Motor 5 run command updated; carries DO_M5.
- **EO_M6**: Motor 6 run command updated; carries DO_M6.

Data Inputs

- **ZE1_EIN .. ZE5_EIN** (TIME; individually ZE1_EIN, ZE2_EIN, ZE3_EIN, ZE4_EIN, ZE5_EIN): Dwell time per start-up step (sVOR1→sVOR2 through sVOR5→sLAEUFT) before the next motor is automatically added. Default: NO_TIME.
- **ZE1_AUS .. ZE5_AUS** (TIME; individually ZE1_AUS, ZE2_AUS, ZE3_AUS, ZE4_AUS, ZE5_AUS): Dwell time per shutdown step (sNACH1→sNACH2 through sNACH5→sAUS) before the next motor is automatically switched off. Default: NO_TIME.
- **STOERUNG_M1 .. STOERUNG_M6** (BOOL): Live, continuous fault signal per motor (not a pulse). Default: FALSE.

Data Outputs

- **STATUS_BETRIEB** (SINT): 0=Off, 1=Starting up, 2=Running, 3=Shutting down.
- **STATUS_STOERUNG** (SINT): 0=none, 4=active — stays latched until the next successful EIN acknowledgement (see Technical Details).
- **EINSCHALTBEREIT** (BOOL): TRUE only when ZAEHLSTAND = 0 (all motors stopped) AND all six STOERUNG_Mx are currently FALSE.
- **ZAEHLSTAND** (SINT): 0..6, number of motors currently running.
- **ANLAGE_AUS** (BOOL): TRUE only in sAUS (STATUS_BETRIEB = Off) — prerequisite for manual operation.
- **TRANSIT_MOTOR** (SINT): 0 = none (sAUS/sLAEUFT), 1..6 = number of the motor that starts/stops after the current time elapses (see TRANSIT_ART).

- **TRANSIT_ART** (SINT): Kind of transition: anlagenSequenz::TRANSIT_ART_KEINE, _VORLAUF or _NACHLAUF.
- **DO_M1 .. DO_M6** (BOOL): Run command per motor.

Adapters

- **timeOut** (Plug, type: iec61499::events::ATimeOut): Standardized TimeOut adapter used for the start-up and shutdown step timing, the same mechanism used by sequence_T_04.

Functionality

The FB is a Basic Function Block (BFB) with 12 real states plus a start state, arranged as a ring made of two linear chains:

1. **Start-up chain** (sVOR1 → sVOR2 → sVOR3 → sVOR4 → sVOR5 → sLAEUFT): Each step adds exactly one more motor, in the order M6, M5, M4, M3, M2, M1 (the highest-numbered motors start first). The transition to the next step happens automatically once the corresponding ZEk_EIN time elapses (timeOut.TimeOut).
2. **Shutdown chain** (sNACH1 → sNACH2 → sNACH3 → sNACH4 → sNACH5 → sAUS): Mirror image of the start-up chain — each step switches off exactly one motor, in the order M1, M2, M3, M4, M5, M6.
3. **Start**: EIN (only from sAUS, only when EINSCHALTBEREIT) acknowledges a pending STATUS_STOERUNG indication and activates sVOR1.
4. **Planned stop**: AUS from any start-up step or from sLAEUFT jumps to the mirror point of the shutdown chain (sVOR_k → sNACH_(6-k), sLAEUFT → sNACH1) — only the motors that are actually running at that moment are shut down in order; motors already stopped are left untouched.
5. **Fault cascade**: Every EI_Mx event (With STOERUNG_Mx) has a direct, single-step transition from every state preceding its target step to the matching sNACH_x (or to sAUS for x=6). This makes the FB react to any fault immediately, regardless of whether the ring is currently in start-up, shutdown, or continuous operation (sLAEUFT) — with no detour through a mirror point and no multi-cycle re-evaluation.
6. **Continuous operation**: sLAEUFT is the upper ring pole — all six motors are running, no timing is active, and the FB stays here until AUS or a fault report.
7. **End of cycle**: After sNACH5 (or directly from a fault), the ring reaches sAUS again — the idle state from which a new EIN cycle can begin.

Technical Details

- **EINSCHALTBEREIT is deliberately NOT coupled to STATUS_STOERUNG.** STATUS_STOERUNG stays latched after a fault until the next successful EIN acknowledgement (it exists for the Visu indicator). If EINSCHALTBEREIT depended on it, the EIN button could never be released again after the very first fault ever recorded — a genuine deadlock. EINSCHALTBEREIT instead checks the *live* STOERUNG_Mx signals directly ((ZAEHLSTAND = 0) AND NOT (STOERUNG_M1 OR ... OR STOERUNG_M6)).
- **The sAUS → sVOR1 transition also carries the full interlock condition inside the event bracket** (EIN[NOT (STOERUNG_M1 OR ... OR STOERUNG_M6)]), not just a check against the computed status. A plain status flag like EINSCHALTBEREIT enforces nothing by itself — only an actual ECTransition Condition prevents a start while a fault is pending.
- **Events are polar and are never combined with AND/OR.** Every condition that combines an event with a data condition uses exclusively the bracket syntax EventName[boolean_expression] (e.g. EI_M3[STOERUNG_M3]) — the only valid form for this in IEC 61499.
- **No self-loops left in the ECC.** An earlier revision used a cyclic EI_CYCLIC event to re-evaluate faults over multiple cycles; that has been removed. EI_CYCLIC_Auswertung now runs as an entry algorithm in every active state instead, re-evaluating STOERUNG_M1..M6 on every state change.
- **Direct single-step fault dispatch instead of a mirror-point detour.** Every state preceding an sNACH_x has its own direct EI_Mx transition to it (51 edges in total for the fault cascade) — no intermediate step via a mirror point as used for the planned AUS.
- **MaxBetriebRest/shutdown remaining-time tracking (demo-server scheme) is not yet implemented** — a deliberate, documented simplification of the current revision (see the comment in the FB header).

State Overview

State	Description	Active motors	Transition condition to the next state

xSTART	Initial idle state.	—	1 (immediately to sAUS)
sAUS	Ring base state, bottom.	none	EIN[NOT (STOERUNG_M1..M6)] → sVOR1
sVOR1	Start-up step 1.	M6	timeOut.TimeOut → sVOR2; AUS → sNACH5; EI_Mx → matching sNACHx/sAUS
sVOR2	Start-up step 2.	M5, M6	timeOut.TimeOut → sVOR3; AUS → sNACH4; EI_Mx → matching sNACHx/sAUS
sVOR3	Start-up step 3.	M4, M5, M6	timeOut.TimeOut → sVOR4; AUS → sNACH3; EI_Mx → matching sNACHx/sAUS
sVOR4	Start-up step 4.	M3, M4, M5, M6	timeOut.TimeOut → sVOR5; AUS → sNACH2; EI_Mx → matching sNACHx/sAUS
sVOR5	Start-up step 5.	M2, M3, M4, M5, M6	timeOut.TimeOut → sLAEUFT; AUS → sNACH1; EI_Mx → matching sNACHx/sAUS
sLAEUFT	Ring pole, top — continuous operation.	M1..M6 (all)	AUS → sNACH1; EI_Mx → matching sNACHx/sAUS
sNACH1	Shutdown step 1 (M1 stopped).	M2..M6	timeOut.TimeOut → sNACH2; EI_Mx → matching sNACHx/sAUS
sNACH2	Shutdown step 2 (M1, M2 stopped).	M3..M6	timeOut.TimeOut → sNACH3; EI_Mx → matching sNACHx/sAUS
sNACH3	Shutdown step 3 (M1..M3 stopped).	M4, M5, M6	timeOut.TimeOut → sNACH4; EI_Mx → matching sNACHx/sAUS
sNACH4	Shutdown step 4 (M1..M4 stopped).	M5, M6	timeOut.TimeOut → sNACH5; EI_Mx → matching sNACHx/sAUS
sNACH5	Shutdown step 5 (M1..M5 stopped).	M6	timeOut.TimeOut → sAUS; EI_M6 → sAUS

Global fault cascade: from every state preceding an sNACH_x, EI_Mx[STOERUNG_Mx] leads directly, with no detour, straight into that state (or to sAUS for x=6) — 51 such direct edges in total, see the network diagram.

Application Scenarios

- **Grain-intake conveyor chain**: Six conveyor elements in series (e.g. bucket elevator, cross conveyor, trough or screw conveyor) that must only be switched on and off in a fixed order so that no element ever conveys against a stopped downstream element.
- **Multi-stage conveying systems in general**: Any system with a fixed cascade of motors where a fault at one stage requires a controlled but immediate retreat of the upstream stages.
- **Safety-critical sequence chains**: Processes where a plain timed sequencer is not enough because a latched fault indication and a restart interlock (EINSCHALTBEREIT) are also required.

⚖️ Comparison with Similar Blocks

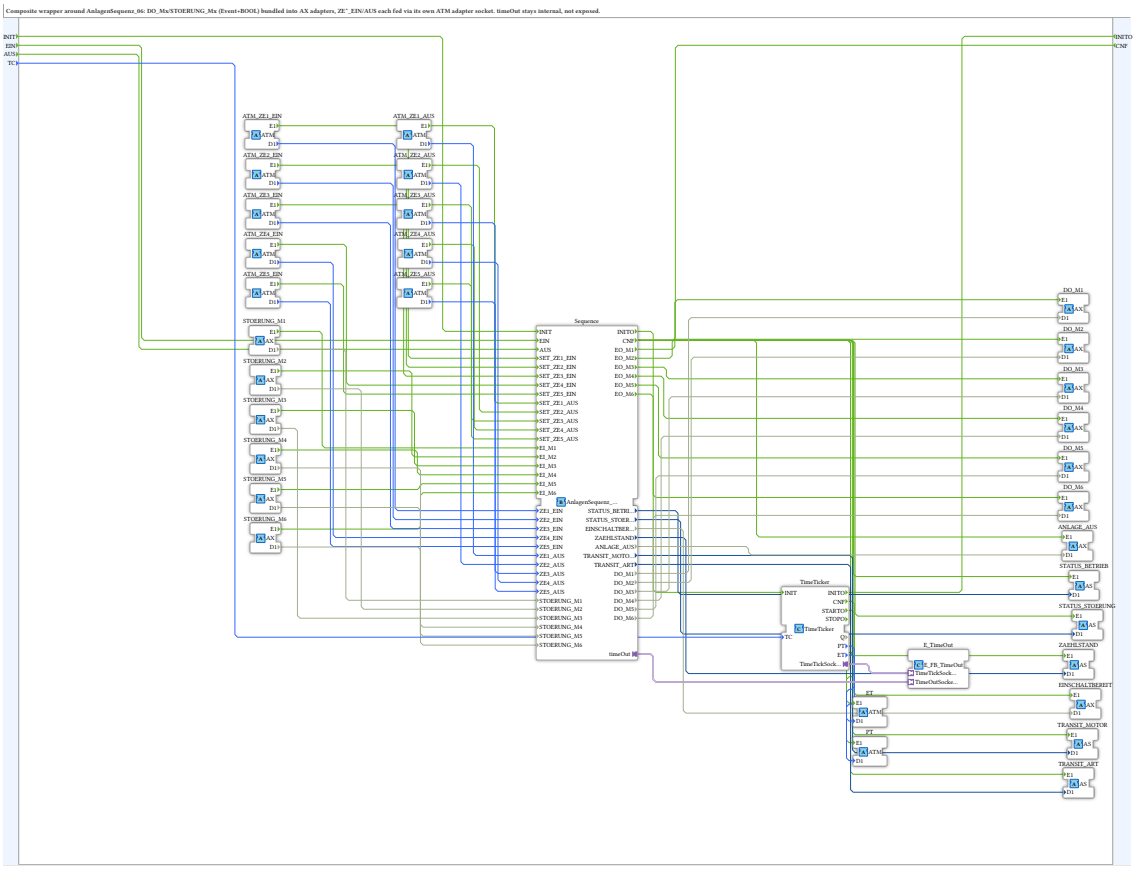
- **sequence_T_04** / **sequence_T_08**: Generic, linear timed sequencers with 4 or 8 steps and a single reset path. AnlagenSequenz_06 reuses the same ATimeOut adapter mechanism, but is not a generic block —

the ring topology (two coupled linear chains), the fixed six-motor pattern per step, and the direct single-step fault cascade are all hard-coded, not configurable.
- **Plain timer chains (TON chaining)**: Would need to reimplement state logic, the per-step motor pattern, and fault handling by hand. `AnlagenSequenz_06` encapsulates all of that, including the interlock logic.
- **Counter-/event-based sequencers**: Advance on external triggers rather than time. `AnlagenSequenz_06` is specifically designed for the case where the dwell time per step is fixed (motor start-up time), with faults as the only asynchronous interruption path.

Conclusion

`AnlagenSequenz_06` bundles a complete, safety-relevant start/stop logic for six motors operating in series into a single block: an ordered start-up and shutdown sequence via two coupled linear chains, a latched fault status with an explicit restart interlock, and a direct, delay-free fault cascade from every operating state. The clear separation of ring topology (ECC), motor switching logic (algorithms), and timing (`timeOut` adapter) keeps the block maintainable despite its complexity and significantly reduces the programming effort in the parent application network.

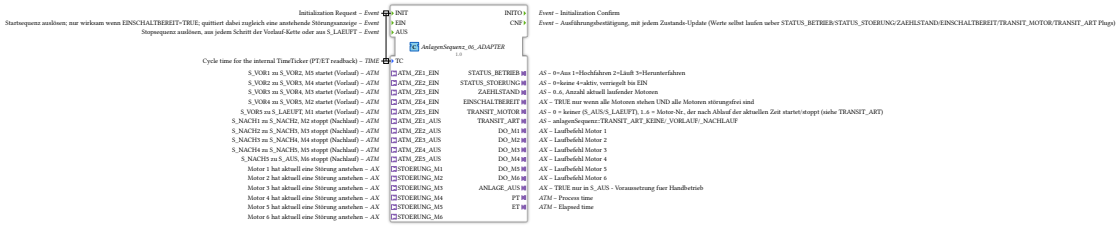
AnlagenSequenz_06_ADAPTER



• – ○ • – ○ • – ○ •

Introduction

`AnlagenSequenz_06_ADAPTER` is a composite wrapper around AnlagenSequenz_06: the same ring-sequencer logic for six motors, but with every data-carrying input/output moved onto adapter connections, so the block can be wired into an application entirely through adapters (`AX`, `AS`, `ATM`) without pulling individual Event/BOOL/TIME lines. The timing (`timeOut`) stays internal and is not exposed.



Function Blocks Used (FBs)

Sub-modules: AnlagenSequenz_06_ADAPTER

- **Type**: Composite (FBNetwork)
- **Internal FBs used**:
  - **Sequence**: `logiBUS::utils::sequence::timed::AnlagenSequenz_06` - the actual ring sequencer, see AnlagenSequenz_06 for the full state overview.
  - **TimeTicker**: `adapter::events::TimeOut::TimeTicker` - generates the cyclic time tick (`TC`, default `T#200ms`) for the internal timing.
  - **E_TimeOut**: `adapter::events::TimeOut::E_FB_TimeOut` - couples `Sequence.timeOut` to `TimeTicker`, so the lead/lag times run without a dedicated timer FB in the application.
- **Functionality**: Bundles the six `DO_Mx` run commands and six `STOERUNG_Mx` fault inputs, plus every status value (`STATUS_BETRIEB`, `STATUS_STOERUNG`, `ZAEHLSTAND`, `EINSCHALTBEREIT`, `TRANSIT_MOTOR`, `TRANSIT_ART`, `ANLAGE_AUS`), into `AX`/`AS` adapters, and feeds each of the ten dwell times (`ZE1..5_EIN`/`_AUS`) through its own `ATM` adapter socket instead of exposing them as individual `TIME` data inputs. `PT`/`ET` (process/elapsed time) are exposed as `ATM` plugs for a readback of the internal timing.
- **Dwell-time sockets (`ATM`)**: The ten sockets `ATM_ZE1_EIN`..`ATM_ZE5_EIN` and `ATM_ZE1_AUS`..`ATM_ZE5_AUS` (each of type `adapter::types::unidirectional::ATM`) deliver the dwell times of the transitions:
  - `ATM_ZE1_EIN`: S_VOR1 to S_VOR2, M5 starts (lead)
  - `ATM_ZE2_EIN`: S_VOR2 to S_VOR3, M4 starts (lead)
  - `ATM_ZE3_EIN`: S_VOR3 to S_VOR4, M3 starts (lead)
  - `ATM_ZE4_EIN`: S_VOR4 to S_VOR5, M2 starts (lead)
  - `ATM_ZE5_EIN`: S_VOR5 to S_LAEUFT, M1 starts (lead)
  - `ATM_ZE1_AUS`: S_NACH1 to S_NACH2, M2 stops (lag)
  - `ATM_ZE2_AUS`: S_NACH2 to S_NACH3, M3 stops (lag)
  - `ATM_ZE3_AUS`: S_NACH3 to S_NACH4, M4 stops (lag)
  - `ATM_ZE4_AUS`: S_NACH4 to S_NACH5, M5 stops (lag)
  - `ATM_ZE5_AUS`: S_NACH5 to S_AUS, M6 stops (lag)

Program Flow and Connections

1. `EIN`/`AUS` pass through unchanged to `Sequence.EIN`/`Sequence.AUS`.
2. Each `ATM_ZEk_EIN`/`ATM_ZEk_AUS` adapter delivers its time (`.D1`) to `Sequence.ZEk_EIN`/`Sequence.ZEk_AUS` and simultaneously (`.E1`) triggers the matching `Sequence.SET_ZEk_EIN`/`SET_ZEk_AUS` set event.
3. Each `STOERUNG_Mx` adapter delivers its fault signal (`.D1`) to `Sequence.STOERUNG_Mx` and (`.E1`) triggers `Sequence.EI_Mx`.
4. `Sequence.CNF` simultaneously triggers all status adapters (`STATUS_BETRIEB`, `STATUS_STOERUNG`, `ZAEHLSTAND`, `EINSCHALTBEREIT`, `TRANSIT_MOTOR`, `TRANSIT_ART`, `ANLAGE_AUS`); the associated data values run in parallel via their `.D1`.
5. Each `Sequence.EO_Mx` triggers `DO_Mx.E1`, with the run command carried via `DO_Mx.D1`.
6. `Sequence.INITO` starts the `TimeTicker`; its `CNF`/`STARTO` feed `ET`/`PT` for a pure time readback to the outside.
7. `Sequence.timeOut` (adapter socket) is connected internally to `E_TimeOut.TimeOutSocket`, `TimeTicker.TimeTickSocket` to `E_TimeOut.TimeTickSocket` - the entire timing stays encapsulated inside the composite.

Application Scenarios

- Applications that consistently rely on adapter connections (e.g. for VT integration or OPC-UA exposure via existing adapter conversion chains) and don't want to pull individual event/data lines to the motor sequence.

- Reusing the same ring-sequencer logic across multiple plant sections while keeping the wiring uniform through standardized adapter types (`AX`, `AS`, `ATM`).

⚖️ Comparison with Similar Function Blocks

- **AnlagenSequenz_06**: The underlying basic-FB logic with classic event/data connections. `AnlagenSequenz_06_ADAPTER` only changes the interface, not the state behavior.
- **sequence_T_08_ADAPTER**: Same adapter-wrapper pattern, applied to the generic 8-step sequencer instead of the fixed 6-motor ring topology.

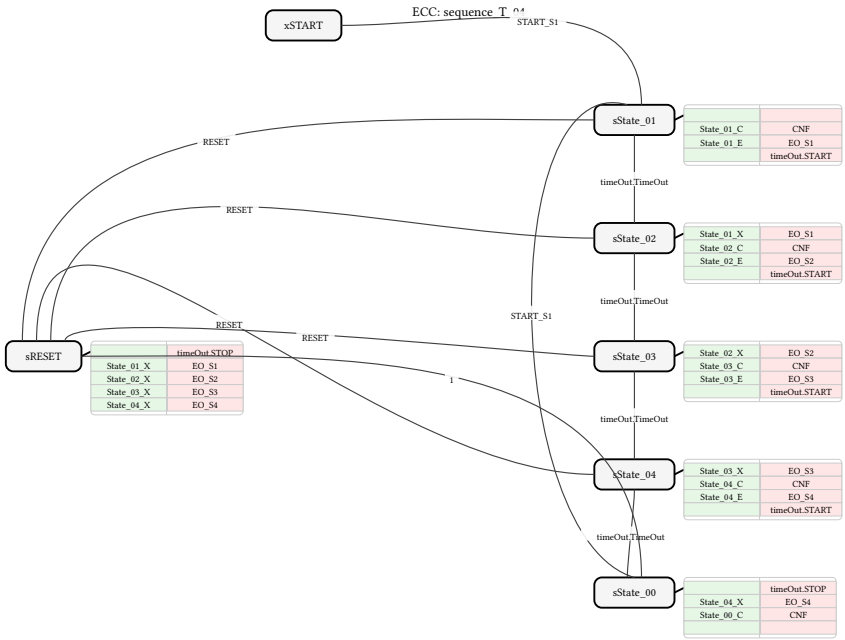
Conclusion

`AnlagenSequenz_06_ADAPTER` makes the complete ring-sequencer logic for six motors usable through pure adapter connections, without changing the underlying state logic - the timing stays fully encapsulated internally.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

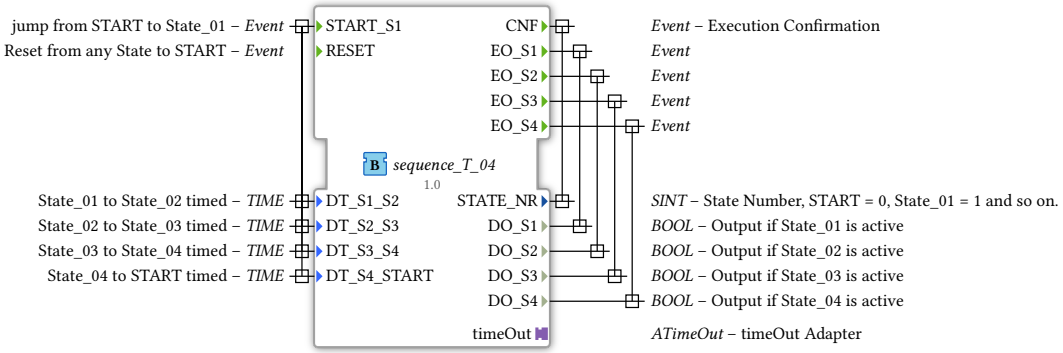
sequence_T_04



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_T_04` is a time-controlled sequencer with four defined states (State_01 to State_04) and a start state (START). It enables the cyclical or one-time execution of a fixed sequence, with the dwell time in each state controlled by configurable time values. The transition between states occurs automatically after the set time has elapsed. This function block is ideally suited for time-controlled sequences, such as those found in conveyor systems, packaging machines, or washing processes.



Interface Structure

Event Inputs

- **`START_S1`**: Starts the sequence. An event at this input causes the transition from the START state or sState_00 to the first active state, State_01. The time data `DT_S1_S2`, `DT_S2_S3`, `DT_S3_S4`, and `DT_S4_START` are read.
- **`RESET`**: Immediately resets the sequence from any state to the idle state (sState_00). All outputs (`DO_S1` to `DO_S4`) are deactivated.

Event Outputs

- **`CNF`**: Execution Confirmation event. Triggered on every state change and returns the current state number `STATE_NR`.
- **`EO_S1`**: Triggered upon entering state State_01 and returns the corresponding data output `DO_S1`.
- **`EO_S2`**: Triggered upon entering state State_02 and returns the corresponding data output `DO_S2`.
- **`EO_S3`**: Triggered upon entering state State_03 and returns the corresponding data output `DO_S3`.
- **`EO_S4`**: Triggered upon entering state_04 and outputs the corresponding data output `DO_S4`.

Data Inputs

- **`DT_S1_S2`** (TIME): Time spent in state_01 before the automatic transition to state_02. Initial value: `NO_TIME`.
- **`DT_S2_S3`** (TIME): Time spent in state_02 before the automatic transition to state_03. Initial value: `NO_TIME`.
- **`DT_S3_S4`** (TIME): Time spent in state_03 before the automatic transition to state_04. Initial value: `NO_TIME`.
- **`DT_S4_START`** (TIME): Duration in state State_04 before the automatic transition to idle state sState_00 occurs. Initial value: `NO_TIME`.

Data Outputs

- **`STATE_NR`** (SINT): Outputs the number of the currently active state. START = 0, State_01 = 1, State_02 = 2, State_03 = 3, State_04 = 4.
- **`DO_S1`** (BOOL): Logical output that is TRUE as long as the function block is in state State_01.
- **`DO_S2`** (BOOL): Logical output that is TRUE while the FB is in state State_02.
- **`DO_S3`** (BOOL): Logical output that is TRUE while the FB is in state State_03.
- **`DO_S4`** (BOOL): Logical output that is TRUE while the FB is in state State_04.

Adapter

- **`timeOut`** (Plug, Type: `iec61499::events::ATimeOut`): A standardized TimeOut adapter used for timed state transitions. The function block (FB) starts the timer (`timeOut.START`) upon entering an active state and reacts to its `TimeOut` event.

Functionality

The FB operates as a Basic Function Block (BFB) with a defined Execution Control Chart (ECC). The sequence iterates through the states in the fixed order: START -> State_01 -> State_02 -> State_03 -> State_04 -> sState_00.

1. **Start**: A `START_S1` event (from states `xSTART` or `sState_00`) activates State_01.
2. **State Activation**: Upon entering a state (State_01-04), the following actions are performed:

- The corresponding data output (`DO_Sx`) is set to TRUE (Entry Algorithm `State_x_E`).
- The associated event (`EO_Sx`) is triggered.
- The state number (`STATE_NR`) is updated (Confirmation Algorithm `State_x_C`).
- The time configured for the next transition (`DT_...`) is passed to the `timeOut` adapter, and the timer is started.
- The general confirmation event (`CNF`) is triggered.

1. **Time-Controlled Transition**: After the time set in the timer has elapsed, the adapter triggers the `TimeOut` event. This is the condition for transitioning to the next state in the sequence.
2. **State Exit**: When exiting a state, the corresponding data output (`DO_Sx`) is reset to FALSE by the exit algorithm (`State_x_X`).
3. **Cycle End and Reset**: After State_04, the function block switches to the state `sState_00` (idle state). From here, the sequence can be restarted by another `START_S1` event. A `RESET` event from any state immediately leads to state `sRESET`, disables all outputs, and then transitions to `sState_00`.

Technical Features

- **Initial Values**: The time data inputs are pre-assigned the constant `NO_TIME` by default. A value of `NO_TIME` causes the function block to wait indefinitely after entering the state until a `RESET` event occurs.
- **State sState_00**: This state is the stable idle state after the completion of a cycle. Unlike the initial `xSTART`, it is part of the ECC and can execute algorithms (set the state number to 0 here).
- **Adapter Usage**: The timing control is completely outsourced to the standardized `ATimeOut` adapter, increasing reusability and clarity.

State Overview

State Name	Description	Active Outputs	Transition Condition to Next State
xSTART	Initial Idle State.	None	`START_S1`
sState_01	First Active Step.	`DO_S1=1`, `STATE_NR=1`	`timeOut.TimeOut` (after DT_S1_S2)
sState_02	Second Active Step.	`DO_S2=1`, `STATE_NR=2`	`timeOut.TimeOut` (after DT_S2_S3)
sState_03	Third active step.	`DO_S3=1`, `STATE_NR=3`	`timeOut.TimeOut` (after DT_S3_S4)
sState_04	Fourth active step.	`DO_S4=1`, `STATE_NR=4`	`timeOut.TimeOut` (after DT_S4_START)
sState_00	Idle state after sequence completion.	`STATE_NR=0`	`START_S1` (for new cycle)
sRESET	Intermediate state for reset operation.	None	Always (`Condition=1`)

Global transition condition: From states sState_01 to sState_04, a `RESET`-Event always returns to the sRESET state.

Application Scenarios

- **Control of Cycle Machines**: Automatic sequencing of machining steps such as drilling, milling, and deburring with adjustable processing times.
- **Packaging Systems**: Control of the timing sequence: product feeding -> packaging closure -> label application -> palletizing.

``* **Cleaning Processes**: Control of a spray booth: Pre-rinse (State_01) -> Main cleaning (State_02) -> Rinse (State_03) -> Dry (State_04).

- **Test Stands**: Automated test sequence where each test step has a fixed duration.

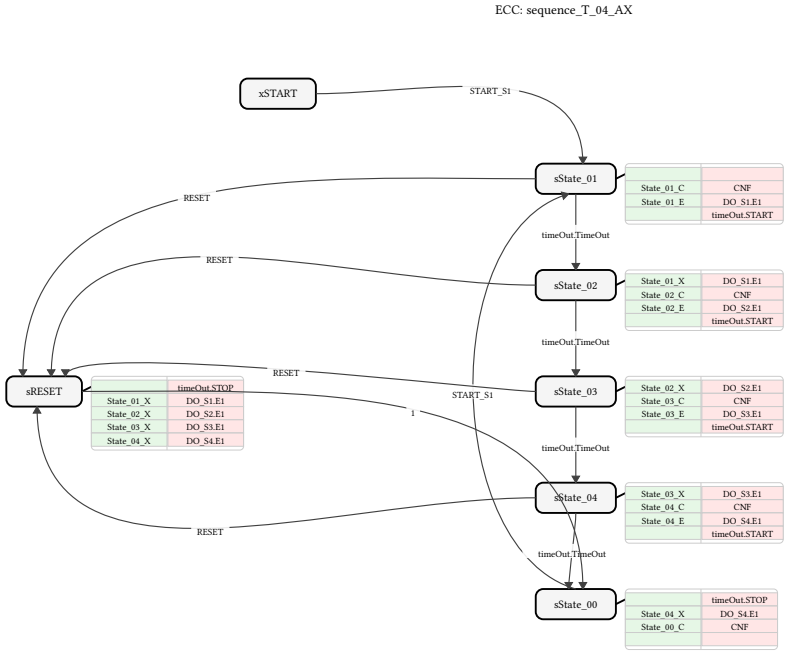
⚖️ Comparison with similar modules

- **Simple Timers (TON)**: Individual timer modules would have to be chained together for a sequence, and the logic for the state transitions and outputs would have to be programmed separately. `sequence_T_04` encapsulates this complete logic.
- **Counter-based Sequencers**: Sequencers that advance based on events (not time) offer a different type of control. `sequence_T_04` is specifically designed for time-critical processes without external triggers.
- **PLC-specific Sequence Flow Languages (S7-GRAFCET)**: This function block implements a similar principle to a Grafcet step chain, but within the portable IEC 61499 standard.

Conclusion

The `sequence_T_04` is a robust and easy-to-configure tool for time-controlled sequence functions with up to four steps. The clear separation of state logic (ECC) and time control (adapter), as well as the complete encapsulation of its behavior, makes it easy to maintain and significantly reduces programming effort in the higher-level application network. The ability to instantly reset from any state also ensures high operational reliability.

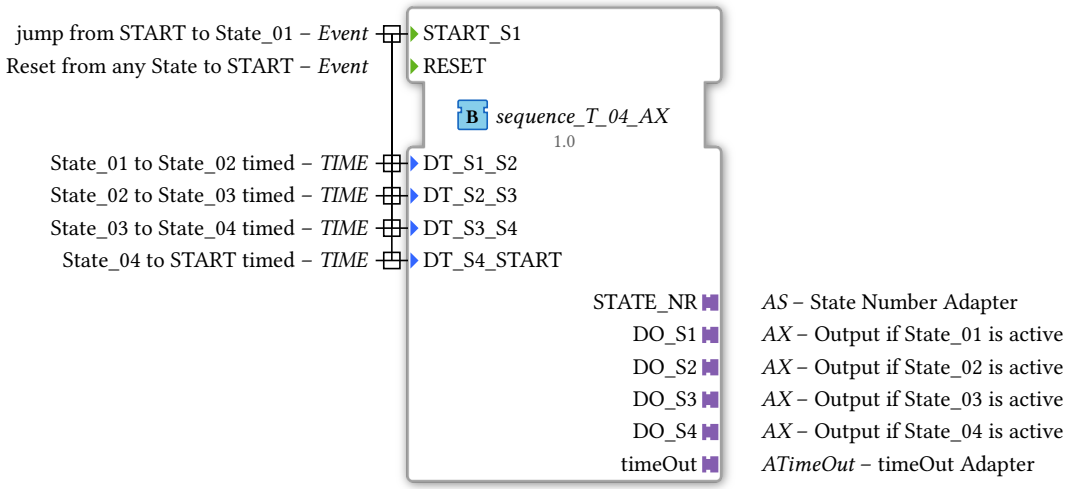
sequence_T_04_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_T_04_AXis a variant ofsequence_T_04 that additionally uses adapters (AX`) for the outputs. It controls a purely time-based sequence with 4 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirms execution.

Data Inputs

- **DT_S1_S2**: Transition time from State_01 to State_02.
- **DT_S2_S3**: Transition time from State_02 to State_03.
- **DT_S3_S4**: Transition time from State_03 to State_04.
- **DT_S4_START**: Transition time from State_04 to START.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

Corresponds to `sequence_T_04`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_T_04`.

Application Scenarios

For time-controlled 4-step sequences with adapter connection.

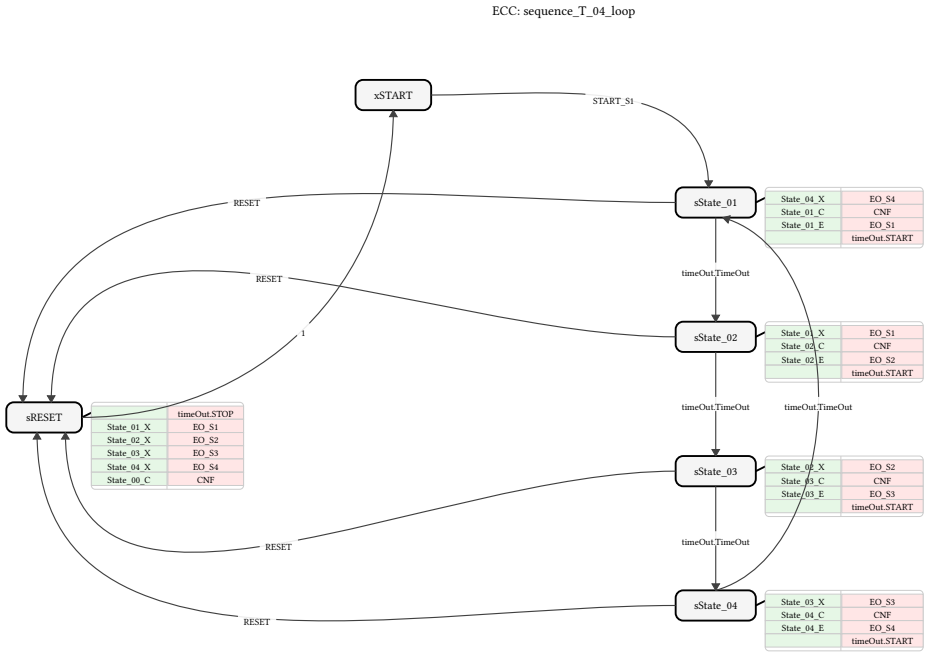
⚖️ Comparison with Similar Function Blocks

- **sequence_T_04**: Standard version without adapter.

Conclusion

Adapter version of the 4-step time sequencer.

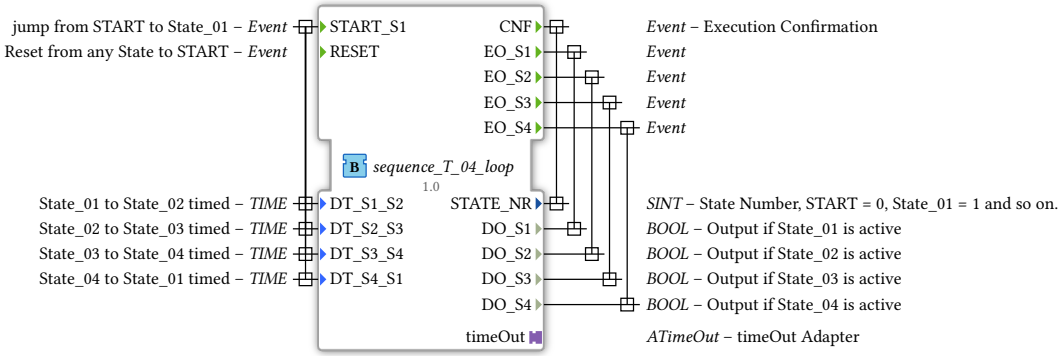
sequence_T_04_loop



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_T_04_loop` is a time-controlled sequencer with four outputs that operates in a loop. It cycles through four states (State_01 to State_04) sequentially. The transition from one state to the next occurs automatically after an adjustable time delay. The block can be reset from any state and then restarts the sequence.



Interface Structure

Event Inputs

- **`START_S1`**: Starts the sequence. The transition occurs from the initial state `START` to `State_01`. This event is linked to the four time data inputs.
- **`RESET`**: Resets the sequence from any active state back to the initial state `START`.

Event Outputs

- **`CNF`**: Confirmation. Triggered on every state change, it returns the current state number.

- **EO_S1**: Triggered upon entering State_01 and returns the corresponding data output DO_S1.
- **EO_S2**: Triggered upon entry into State_02 and returns the corresponding data output DO_S2.
- **EO_S3**: Triggered upon entry into State_03 and returns the corresponding data output DO_S3.
- **EO_S4**: Triggered upon entry into State_04 and returns the corresponding data output DO_S4.

Data Inputs

- **DT_S1_S2** (TIME): Time delay for the transition from State_01 to State_02. Initial value: NO_TIME.
- **DT_S2_S3** (TIME): Time delay for the transition from State_02 to State_03. Initial value: qzmsdocs000031 ... * **DT_S3_S4** (TIME): Time delay for the transition from State_03 to State_04. Initial value: NO_TIME.
- **DT_S4_S1** (TIME): Time delay for the transition from State_04 back to State_01 (loop). Initial value: NO_TIME.

Data Outputs

- **STATE_NR** (SINT): Current status number. 0 = START, 1 = State_01, 2 = State_02, 3 = State_03, 4 = State_04.
- **DO_S1** (BOOL): Is TRUE when State_01 is active.
- **DO_S2** (BOOL): Is TRUE when State_02 is active.
- **DO_S3** (BOOL): Is TRUE when State_03 is active.
- **DO_S4** (BOOL): Is TRUE when State_04 is active.

Adapter

- **timeOut** (Plug, Type: iec61499::events::ATimeOut): A timer adapter used for timed state transitions. The function block starts the timer when a state is entered and reacts to its TimeOut event.

Functionality

The function block (FB) is implemented as a BasicFB with an Execution Control Chart (ECC). The sequence begins in the initial state xSTART. An event START_S1 leads to the first active state sState_01.

In each active state (sState_01 to sState_04), the following actions are executed sequentially:

1. **Exit step of the previous state**: The corresponding data output (DO_Sx) is set to FALSE (except on the first entry of xSTART).
2. **Confirmation Step**: The state number STATE_NR is updated, and the delay time for the *next* transition is passed to the timeOut adapter (timeOut.DT).
3. **Entry Step of the New State**: The corresponding data output (DO_Sx) is set to TRUE, and the corresponding event (EO_Sx) is triggered.
4. **Timer Start**: The timeOut adapter is started (timeOut.START).

The transition to the next state occurs exclusively via the TimeOut event of the adapter. After State_04, the sequence jumps back to State_01 according to the loop logic.

An event RESET from any state leads to state sRESET. There, all outputs (DO_S1 to DO_S4) are deactivated, the timer is stopped, the state number is set to 0 (START), and an acknowledgment (CNF) is issued. The function block then automatically returns to state xSTART.

Technical Features

- **Time Control**: The transitions are purely time-controlled. There are no event-driven transitions between the main states. * **Initial Values**: The time data inputs are pre-assigned to NO_TIME by default. This must be adjusted for proper operation.
- **Constants**: The function block uses constants from the library logiBUS::utils::sequence::const::sequence (e.g., for state numbers) and logiBUS::utils::sequence::const::sequence::NO_TIME.

State Overview

1. **xSTART**: Initial, inactive state. Waiting for START_S1.
2. **sState_01**: First active state. DO_S1 = TRUE. Timer for transition to State_02 is running.
3. **sState_02**: Second active state. DO_S2 = TRUE. Timer for transition to State_03 is running.
4. **sState_03**: Third active state. DO_S3 = TRUE. Timer for transition to State_04 is running.
5. **sState_04**: Fourth active state. DO_S4 = TRUE. Timer for transition (back) to State_01 is running.
6. **sRESET**: Reset state. Disables all outputs, stops the timer, and confirms the reset.

Application Scenarios

- Control of cyclic processes with fixed time steps, e.g., in packaging machines, washing systems, or automated assembly lines.
- Control of actuators in a fixed, time-based sequence.
- As a central clock for higher-level control processes.

⚖️ Comparison with Similar Components

- **Simple Timers (TON)**: Individual timers do not offer integrated sequence logic. `sequence_T_04_loop` encapsulates the complete state machine with four steps.
- **Event-Driven Sequencers**: In contrast to event-driven sequencers (e.g., `sequence_E_04_loop`), transitions here occur exclusively based on time, not through external events.
- **PLC Cycle**: Time-controlled sequence monitoring is more precise and independent of the PLC cycle because it is based on the `ATimeOut` adapter.

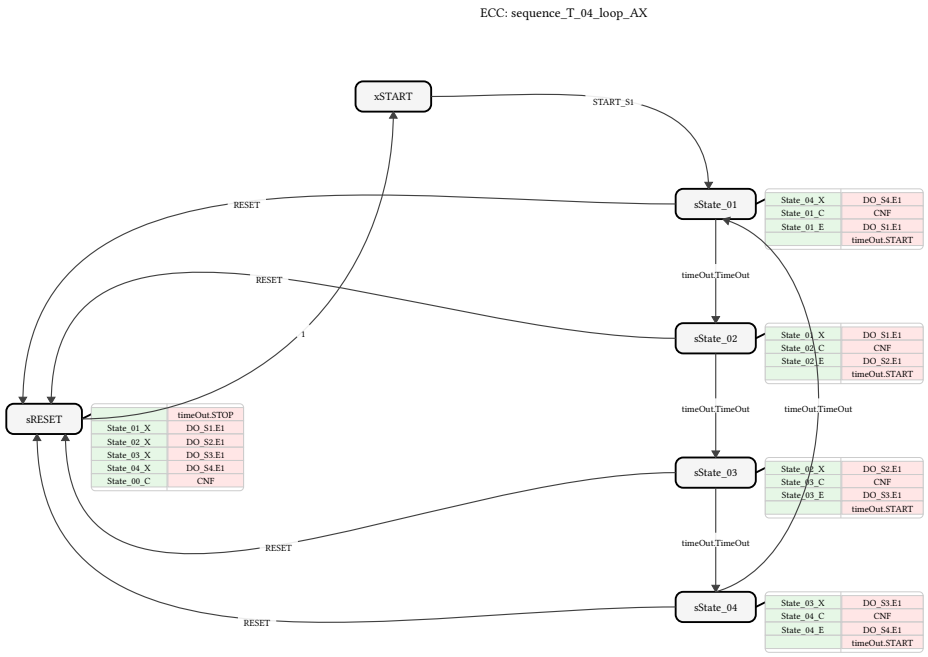
🛠️ Related Exercises

- Exercise_035a

Conclusion

The `sequence_T_04_loop`This is a robust and easy-to-configure function block for time-controlled, four-step sequences. Its clear separation of state logic and time parameters, along with its integrated reset functionality, makes it well-suited for standardized cyclic processes in automation technology. The use of a standard timing adapter ensures portability and reliability.

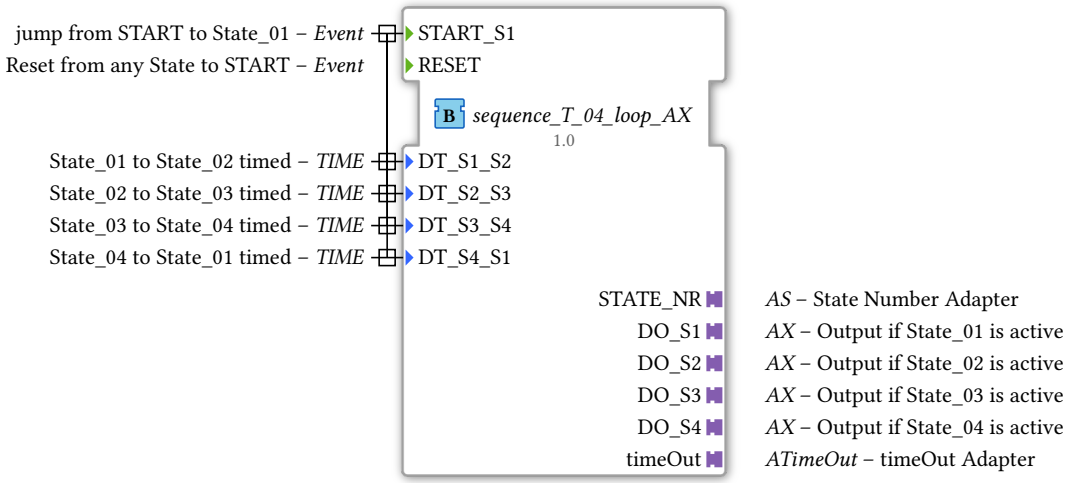
sequence_T_04_loop_AX



• – ○ • – ○ • – ○ •

Introduction

sequence_T_04_loop_AXis a variant ofsequence_T_04_loop that additionally uses adapters (AX`) for the outputs. It controls a purely time-controlled, cyclic sequence with 4 output states.



Interface Structure

Event Inputs
- **START_S1**: Starts the sequence at State_01.
- **RESET**: Resets the sequence.

Event Outputs
- **CNF**: Confirmation of execution.

Data Inputs
- **DT_S1_S2**: Transition time State_01 -> State_02.
- **DT_S2_S3**: Transition time State_02 -> State_03.
- **DT_S3_S4**: Transition time State_03 -> State_04.
- **DT_S4_S1**: Transition time State_04 -> State_01.

Data Outputs
- **STATE_NR** (SINT): Current state number.

Adapters
- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality
Corresponds to `sequence_T_04_loop`, but uses adapters for the outputs.

Technical Features
- Uses `adapter::types::unidirectional::AX`.

State Overview
See `sequence_T_04_loop`.

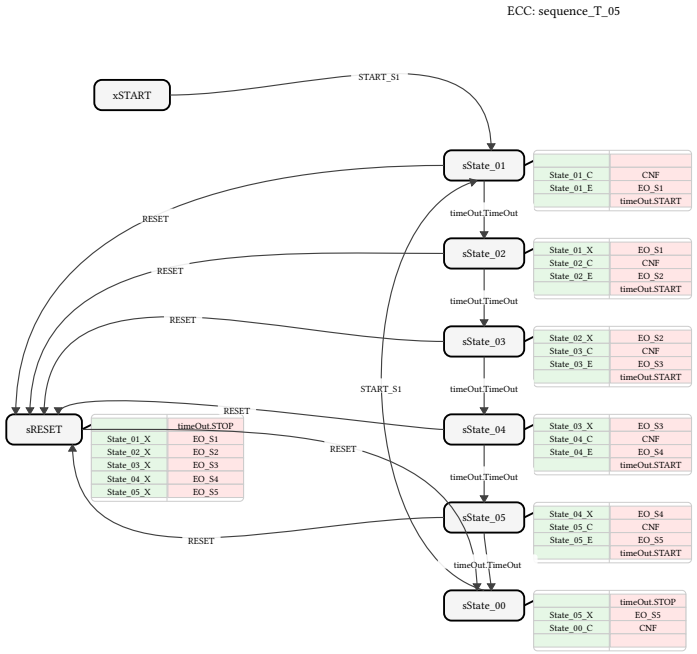
Application Scenarios
For time-controlled, cyclic 4-step sequences with adapter connection.

⚖️ Comparison with Similar Blocks
- **sequence_T_04_loop**: Standard version without adapter.

Conclusion

Adapter version of the 4-step loop time sequencer.

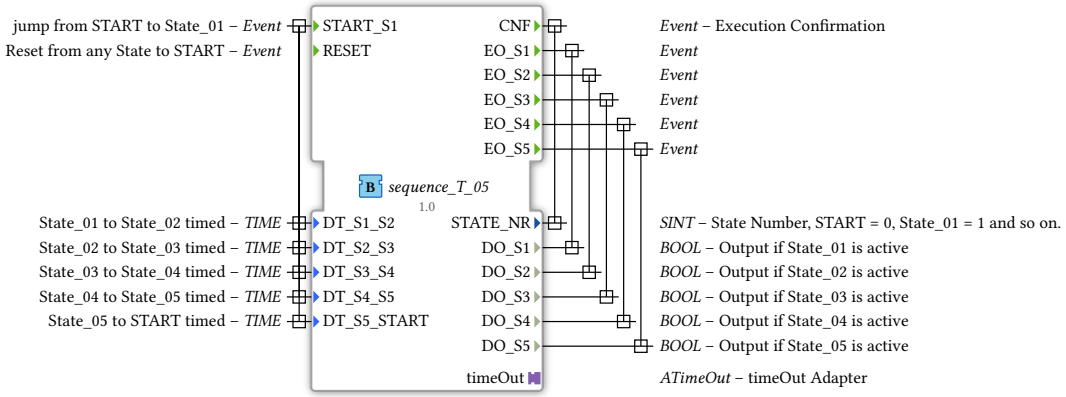
sequence_T_05



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_T_05` is a time-controlled sequencer with five output states. It cycles through a fixed sequence of states (State_01 to State_05), with the transition between each state controlled by adjustable time delays. The block is designed for applications where process steps or actions need to be activated sequentially for a defined duration.



Interface Structure

Event Inputs

- **`START_S1`**: Starts the sequence. The block transitions from the start state (`START` or `State_00`) to the first active state, `State_01`. This event is linked to the five time data inputs.
- **`RESET`**: Resets the sequence from any state back to the inactive start state (`State_00`). All outputs are disabled.

Event Outputs

- **CNF**: Confirmation. Triggered on every state transition, it returns the new state number (STATE_NR).
- **EO_S1**: Triggered upon entering state State_01 and outputs the corresponding data output DO_S1.
- **EO_S2**: Triggered upon entering state State_02 and outputs the corresponding data output DO_S2.
- **EO_S3**: Triggered upon entering state State_03 and outputs the corresponding data output DO_S3.
- **EO_S4**: Triggered upon entering state State_04, this output provides the corresponding data output DO_S4.
- **EO_S5**: Triggered upon entering state State_05, this output provides the corresponding data output DO_S5.

Data Inputs

All time data inputs are of type TIME and have the initial value NO_TIME. They define the dwell time in the respective state before the automatic transition to the next state occurs.

** ... * **DT_S1_S2**: Retention time in State_01 before the transition to State_02.

- **DT_S2_S3**: Retention time in State_02 before the transition to State_03.
- **DT_S3_S4**: Retention time in State_03 before the transition to State_04.
- **DT_S4_S5**: Retention time in State_04 before the transition to State_05.
- **DT_S5_START**: Dwell time in State_05 before transitioning back to the start state State_00.

Data Outputs

- **STATE_NR** (SINT): Outputs the current state number. 0 = START/State_00, 1 = State_01, ..., 5 = State_05.
- **DO_S1** (BOOL): Is TRUE when state State_01 is active.
- **DO_S2** (BOOL): Is TRUE when state State_02 is active.
- **DO_S3** (BOOL): Is TRUE when state State_03 is active.
- **DO_S4** (BOOL): Is TRUE when state State_04 is active.
- **DO_S5** (BOOL): Is TRUE when state State_05 is active.

Adapter

- **timeOut** (Plug, Type: iec61499::events::ATimeOut): A timeout adapter used for timed state transitions. The block starts (START) the timer when entering an active state and stops (STOP) it when leaving.

Functionality

The block operates as a Basic Function Block with a defined Execution Control Chart (ECC). The sequence is initiated by the event START_S1. The block then cycles through the states State_01 to State_05. In each active state, the corresponding data output (DO_Sx) is set to TRUE, and a timer with the duration configured for that state (DT_Sx_Sy) is started. Once the timer expires (timeOut.TimeOut), the block automatically transitions to the next state. During the state change, the previous output is deactivated, and the new one is activated. After State_05, the block returns to the inactive state State_00. An event in RESET immediately interrupts the sequence, disables all outputs, and brings the block to state State_00.

Technical Features

- **State Handling**: Each active state (State_01 to State_05) has separate algorithms for entry (_E), acknowledgment (_C), and exit (_X). This allows for a clear separation of logic.
- **Timer Integration**: The timing control is completely outsourced to the adapter ATimeOut, which increases reusability and maintainability.
- **Constants**: The block imports constants from logiBUS::utils::sequence::const::sequence (for state numbers) and ::NO_TIME for the initial duration values.
- **Initial State**: The actual inactive sleep state after a reset or sequence completion is sState_00. xSTART is the initial ECC state during the first boot.

State Overview

1. **xSTART**: Initial ECC state at system startup.
2. **sState_00**: Inactive sleep state. All outputs are FALSE. The sequence can be started from here with START_S1. 3. **sState_01 to sState_05**: Active sequence states. The respective output DO_Sx is TRUE. The transition to the next state is time-controlled.
3. **sRESET**: Intermediate state that is accessed from any active state upon a RESET event. It deactivates all outputs and then switches to sState_00.

Application Scenarios

- **Batch Process Control**: Sequentially activated steps such as filling, heating, stirring, cooling, and emptying with adjustable step times.
- **Sequence Controls in Machines**: Time-controlled sequence of cylinder movements or tool changes in an automated system.
- **Test Sequences**: Automated test sequences in which various signals are applied sequentially for a specific duration, and the results are evaluated.
- **Lighting Control**: Time-controlled choreographies for advertising or decorative lighting.

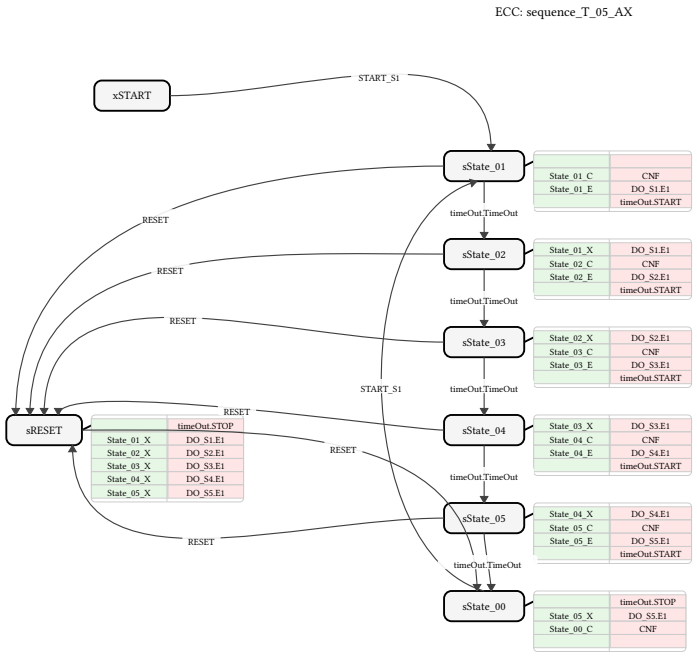
⚖️ Comparison with Similar Function Blocks

Compared to a simple `E_DELAY` or `E_SR` function block, `sequence_T_05` offersA predefined, multi-stage sequence logic in a single, configurable block. Compared to a custom-programmed sequence with multiple interconnected blocks, it significantly simplifies the application and reduces the potential for errors. Other sequencer blocks might react to events (instead of time) for transitions or allow a variable number of steps.

Conclusion

The `sequence_T_05` is a robust and easy-to-configure tool for time-controlled sequences with a fixed number of steps. Due to the clear separation of state logic and timing, as well as the comprehensive confirmation and reset mechanisms, it is ideally suited for reliable automation tasks in industrial environments. Parameterization of step times at runtime enables a high degree of flexibility.

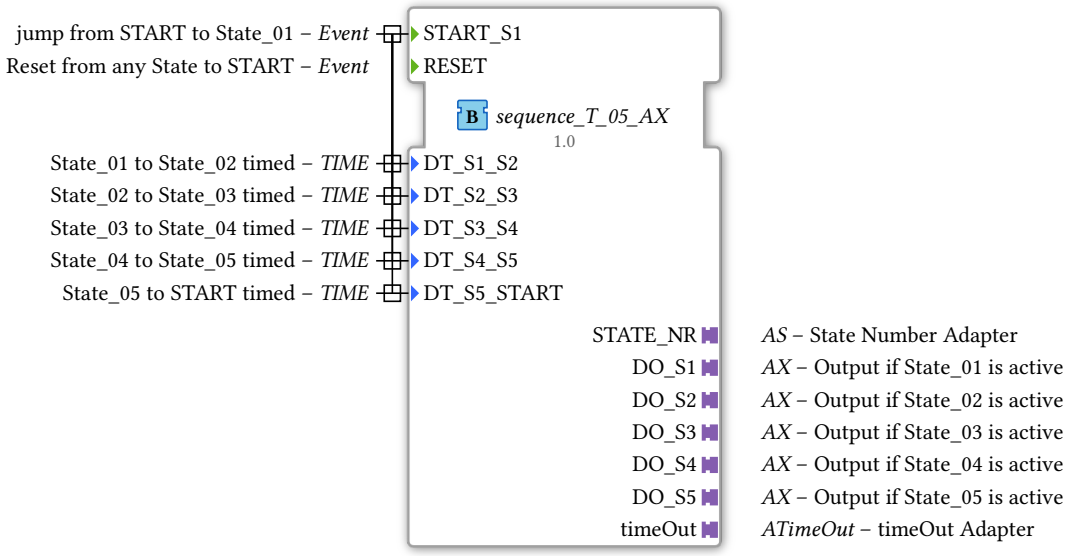
sequence_T_05_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_T_05_AXis a variant ofsequence_T_05 that additionally uses adapters (AX`) for the outputs. It controls a purely time-based sequence with 5 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirms execution.

Data Inputs

- **DT_S1_S2**: Transition time for State_01 -> State_02.
- **DT_S2_S3**: Transition time for State_02 -> State_03.
- **DT_S3_S4**: Transition time for State_03 -> State_04.
- **DT_S4_S5**: Transition time for State_04 -> State_05.
- **DT_S5_START**: Transition time for State_05 -> START.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Output adapter for State_05.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

Corresponds to `sequence_T_05`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_T_05`.

Application Scenarios

For time-controlled 5-step sequences with adapter connection.

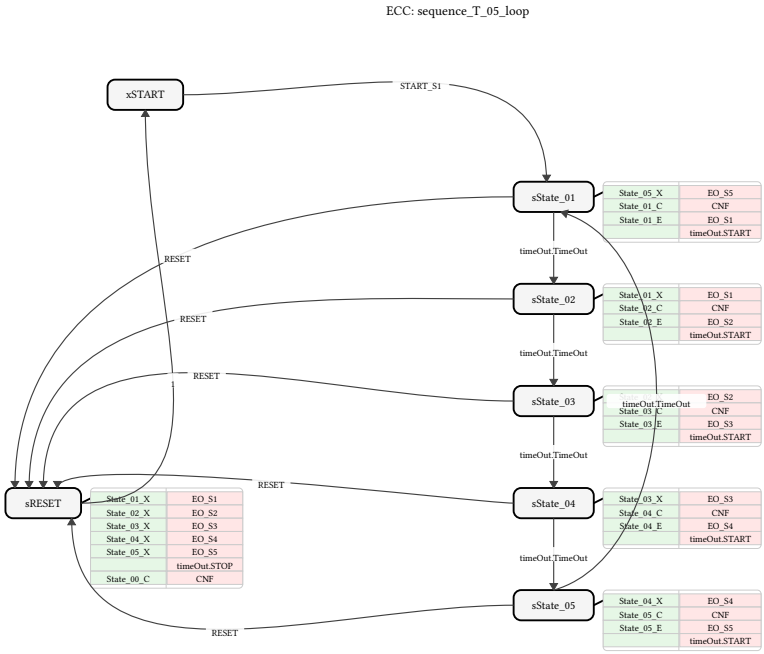
⚖️ Comparison with similar components

- **sequence_T_05**: Standard version without adapter.

Conclusion

Adapter version of the 5-step time sequencer.

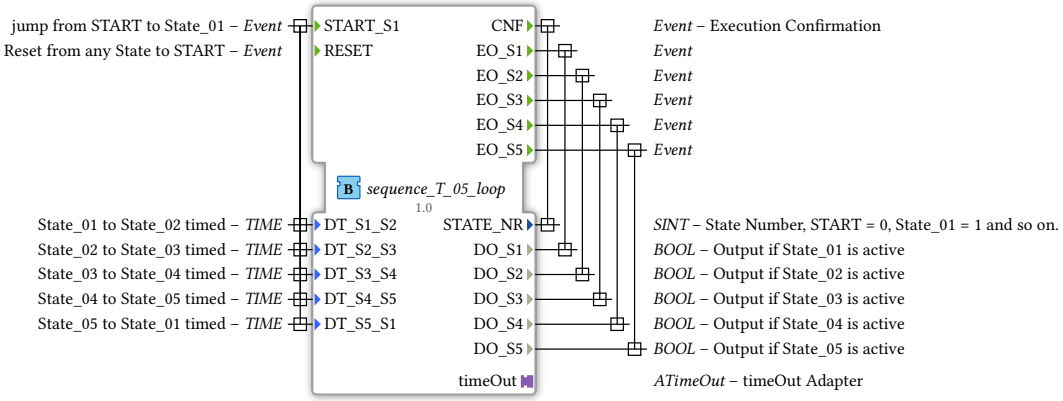
sequence_T_05_loop



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_T_05_loop` is a time-controlled sequencer that implements a cyclical sequence of five states (State_01 to State_05). The transition from one state to the next occurs after an adjustable time delay. This function block is designed for applications where actions or process steps need to be activated sequentially for a defined duration, for example, in automated handling or manufacturing processes.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence and performs a transition from the initial state (START) to the first active state (State_01). The event is linked to the five time data inputs.
- **RESET**: Aborts the current sequence and enters the reset state (sRESET), from where it automatically returns to the initial state (xSTART). Resets all outputs.

Event Outputs

- **CNF**: Confirmation event. Triggered on every state change, this output returns the current state number (STATE_NR).
- **EO_S1** to **EO_S5**: State events. Triggered upon entering the respective state (State_01 to State_05), these outputs return the corresponding Boolean data output (DO_S1 to DO_S5).

Data Inputs

- **DT_S1_S2** (Type: TIME, Initial value: NO_TIME): Duration of the transition from State_01 to State_02.
- **DT_S2_S3** (Type: TIME, Initial value: NO_TIME): Time for the transition from State_02 to State_03.
- **DT_S3_S4** (Type: TIME, Initial value: NO_TIME): Time for the transition from State_03 to State_04.
- **DT_S4_S5** (Type: TIME, Initial value: NO_TIME): Time for the transition from State_04 to State_05.
- **DT_S5_S1** (Type: TIME, Initial value: NO_TIME): Time for the transition from State_05 back to State_01, which closes the loop.

Data Outputs

- **STATE_NR** (Type: SINT): Outputs the number of the currently active state. 0 = START, 1 = State_01, ..., 5 = State_05.
- **DO_S1** to **DO_S5** (Type: BOOL): Logical outputs that are TRUE while the FB is in the corresponding state (State_01 to State_05).

Adapter

- **timeOut** (Type: iec61499::events::ATimeOut, Plug): A timer adapter used for timed state transitions. The FB starts the timer upon entering a state and transitions to the next state upon receiving the TimeOut event.

Functionality

The FB is implemented as a Basic Function Block (BFB) with an Execution Control Chart (ECC). After starting (START_S1), it cycles through states State_01 to State_05 in a fixed sequence. In each active state, the following actions are performed:

1. **Exit action of the previous state**: Sets the corresponding Boolean output (DO_Sx) to FALSE.
2. **Confirmation action**: Sets the current state STATE_NR and configures the dwell time for the *current* state in the timeOut adapter (e.g., the value from DT_S1_S2 is loaded in State_01).
3. **Entry Action of the New State**: Sets the corresponding Boolean output (DO_Sx) to TRUE.
4. **Timer Start**: Starts the timeOut adapter with the time loaded in step 2.

The transition to the next state occurs only if the timeOut adapter returns the TimeOut event. After State_05, the function block jumps back to State_01 according to DT_S5_S1, creating an infinite loop. A RESET event from any state disables all outputs, stops the timer, and returns the function block to its initial state, xSTART.

Technical Features

- **Initial Values**: The time data inputs are initialized with the constant NO_TIME by default. A value of NO_TIME or T#0s results in an immediate state transition.
- **State Confirmation**: The CNF event is triggered in every state (including reset), enabling reliable external monitoring of the function block's activity.
- **Adapter Usage**: The time control is completely outsourced to the standardized ATimeOut adapter, which promotes reusability and clear interfaces.
- **Constants**: The function block imports constants from logiBUS::utils::sequence::const::sequence, e.g., for the state numbers (State_00, State_01, ...).

State Overview

The ECC comprises seven states:

- **`xSTART`**: Inactive initial state. Waiting for `START_S1`.
- **`sState_01`** to **`sState_05`**: Active operating states. Each one activates its specific output (`DO_Sx`) and starts the timer for its own duration.
- **`sRESET`**: Reset state. Deactivates all outputs, stops the timer, and sends an acknowledgment (`CNF`) with `STATE_NR=0`. Automatically reverts to condition `1` (always true) and then back to `xSTART`.

Application Scenarios

- **Cycle Control**: Automated sequences in packaging machines where various actuators (grippers, punches, conveyors) must be activated sequentially for specific durations.
- **Process Control**: Step-by-step execution of chemical or thermal processes where each step has a defined dwell time.
- **Display or Flashing Sequences**: Control of visual or audible signals in a defined, timed sequence.

⚖️ Comparison with Similar Function Blocks

Unlike a simple TON timer (delay-on delay), this function block implements a complete state machine with multiple steps. Compared to generic sequencer function blocks, which often use step detection (e.g., via slope edges), this block is purely time-controlled. It resembles a chain of TON blocks but is encapsulated in a single, state-based function block, simplifying its arrangement and parameterization.

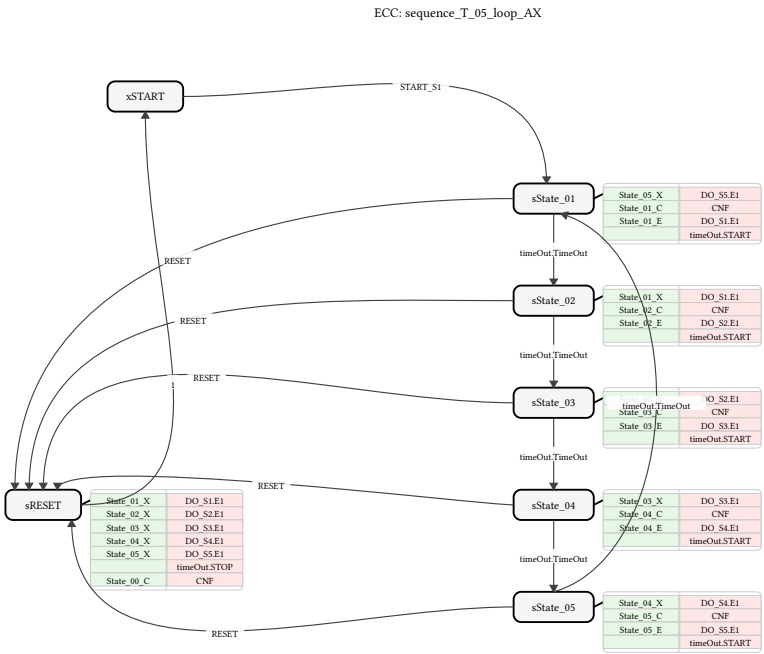
🛠️ Related exercises

- Uebung_035a2
- Uebung_035a3

Conclusion

The `sequence_T_05_loop`This is a specialized, robust, and easy-to-configure sequencer for cyclic, time-controlled, five-step processes. Its clear structure, use of standardized adapters, and comprehensive confirmation of all state changes make it a reliable component for time-critical automation tasks. The loop function and central reset are particularly well-suited for continuous operation applications.

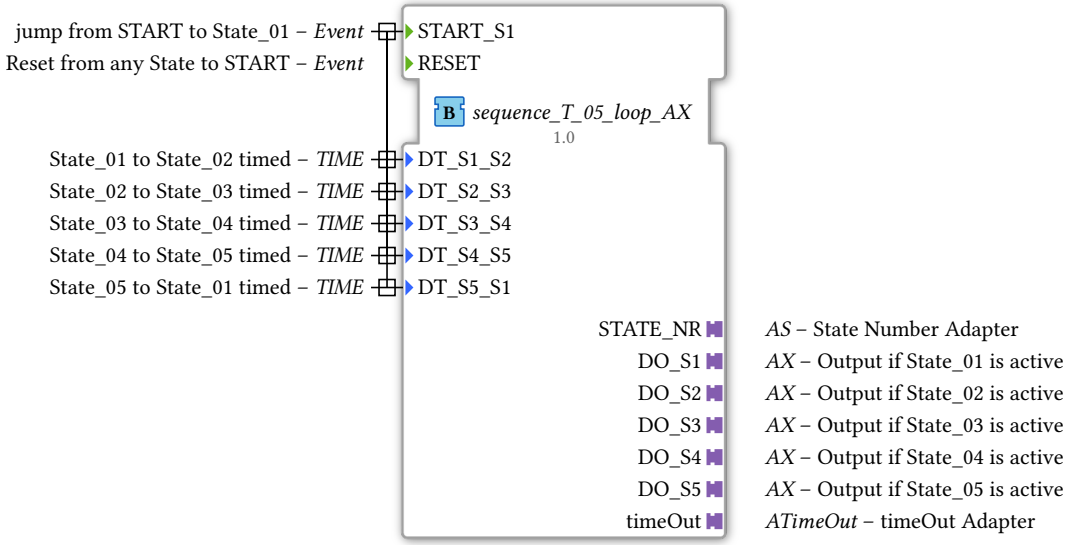
sequence_T_05_loop_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_T_05_loop_AXis a variant ofsequence_T_05_loop that additionally uses adapters (AX`) for the outputs. It controls a purely time-controlled, cyclic sequence with 5 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirmation of execution.

Data Inputs

- **DT_S1_S2**: Transition time for State_01 -> State_02.
- **DT_S2_S3**: Transition time for State_02 -> State_03.
- **DT_S3_S4**: Transition time for State_03 -> State_04.
- **DT_S4_S5**: Transition time for State_04 -> State_05.
- **DT_S5_S1**: Transition time for State_05 -> State_01.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** (adapter::types::unidirectional::AX): Output adapter for State_01.
- **DO_S2** (adapter::types::unidirectional::AX): Output adapter for State_02.
- **DO_S3** (adapter::types::unidirectional::AX): Output adapter for State_03.
- **DO_S4** (adapter::types::unidirectional::AX): Output adapter for State_04.
- **DO_S5** (adapter::types::unidirectional::AX): Output adapter for State_05.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

Corresponds to `sequence_T_05_loop`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_T_05_loop`.

Application Scenarios

For time-controlled, cyclical 5-step sequences with adapter connectivity.

⚖️ Comparison with similar building blocks

- **sequence_T_05_loop**: Standard version without adapter.

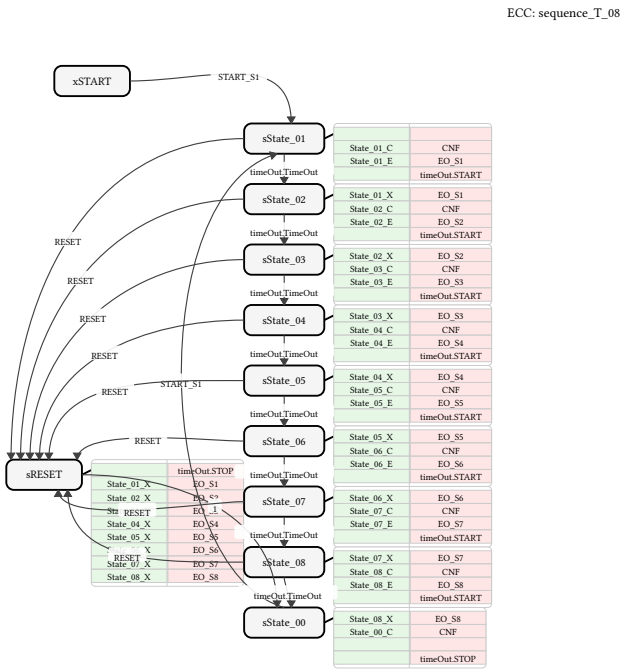
🛠️ Related exercises

- Exercise_035a2_AX
- Exercise_035a3_AX

Conclusion

Adapter version of the 5-step loop time sequencer.

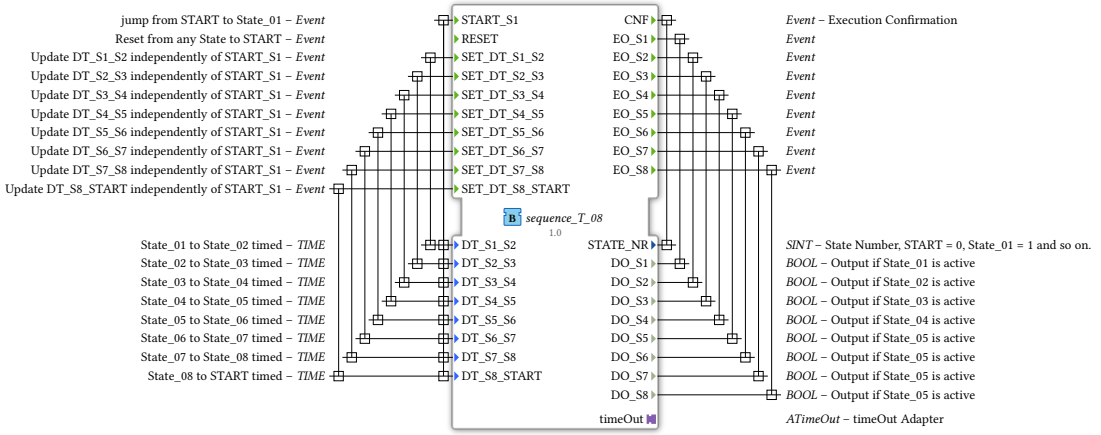
sequence_T_08



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_T_08` is a time-controlled sequencer with eight outputs. It implements a fixed sequence of states, with the transition between individual states controlled by adjustable time delays. This block is designed for applications where process steps or machine states need to be activated sequentially for a defined duration, such as in packaging machines, conveyor systems, or washing systems.



Interface Structure

Event Inputs

- `START_S1`: Starts the sequence. An event at this input causes the transition from state `START` to the first active state, `State_01`. All eight time data inputs are read.
- `RESET`: Immediately resets the sequence. An event at this input leads from any state to state `START` and disables all outputs.
- `SET_DT_S1_S2`, `SET_DT_S2_S3`, `SET_DT_S3_S4`, `SET_DT_S4_S5`, `SET_DT_S5_S6`, `SET_DT_S6_S7`, `SET_DT_S7_S8`, `SET_DT_S8_START`: Update the time of the same name (`DT_S1_S2` to `DT_S8_START`) independently of `START_S1`. If the FB is currently in the corresponding state, that state's time setting is executed again (live update of the running timer).

Event Outputs

- `CNF`: Execution Confirmation event. Triggered on every state change, it returns the new state number, `STATE_NR`.
- `EO_S1` to `EO_S8`: State events. Each of these events is triggered upon entering the corresponding state (`State_01` to `State_08`) and returns the associated Boolean data output value (`DO_S1` to `DO_S8`).

Data Inputs

- `DT_S1_S2` to `DT_S8_START` (Type `TIME`): Define the dwell time for each state. This value determines how long the function block remains in the respective state before the automatic transition to the next state occurs. The default value, `NO_TIME`, disables the timed transition, so the function block remains in the state until `RESET` occurs.
- `DT_S1_S2` to `DT_S8_START` (Type `TIME`): Define the dwell time for each state.

Data Outputs

- `STATE_NR` (Type `SINT`): Outputs the current state number. `0` corresponds to state `START`, `1` to `8` correspond to the active states `State_01` to `State_08`.
- `DO_S1` to `DO_S8` (Type `BOOL`): The physical output signals of the sequence. Each output is set to `TRUE` when the corresponding state is active; otherwise, it is `FALSE`.

Adapter Ports

- `timeOut` (Type `iec61499::events::ATimeOut`): A plug-in adapter that provides the timer. The function block (FB) uses the interface to start a timer (`timeOut.START`) and wait for its expiration (`timeOut.TimeOut`). The respective time duration is transmitted via `timeOut.DT`.

Functionality

The FB operates as a Basic Function Block (BFB) with a defined Execution Control Chart (ECC). The sequence cycles through states `State_01` to `State_08` in a fixed order. Each active state performs three essential actions:

1. **Exit action of the previous state**: Disables the corresponding output (algorithm `State_XX_X`).
2. **Confirmation action**: Sets the state number `STATE_NR` and passes the time duration configured for this state to the timer adapter (algorithm `State_XX_C`). Triggers the `CNF` event.

3. **Entry action**: Enables the corresponding output (algorithm `State_XX_E`). Triggers the corresponding `EO_Sx` event. Starts the timer.

The transition to the next state occurs only when the timer has expired (`timeOut.TimeOut`). After `State_08`, the function block jumps to the inactive `State_00` (also represented as `START`), where the timer is stopped. From here, the sequence can only be restarted by a new `START_S1` event. A `RESET` event from any state leads back to `State_00` via a dedicated reset state (`sRESET`), deactivating all active outputs.

Technical Features

- **Flexible Time Control**: Each state transition can be individually configured at runtime via the `DT_` inputs. The value `NO_TIME` allows the sequence to be paused at a specific point.
- **Immediate Reset**: The `RESET` input always has priority and immediately interrupts the running time control.
- **State Feedback**: The current position in the sequence is always traceable via the `STATE_NR` output.
- **Event-Driven Outputs**: In addition to the continuous data outputs (`DO_Sx`), the function block provides a separate event (`EO_Sx`) for each state, which simplifies the control of downstream, event-driven function blocks.

State Overview

The ECC comprises the following states:

- **xSTART / sState_00**: Inactive start and end state. `STATE_NR = 0`, all outputs are `FALSE`.
- **sState_01 to sState_08**: The eight active sequence states. `STATE_NR = 1` to `8`. The corresponding output `DO_Sx` is `TRUE`.
- **sRESET**: Internal reset state. Deactivates all outputs and returns to `sState_00`.

Application Scenarios

- **Cycle Control**: In machines where various actuators (valves, motors, heaters) need to be switched on and off sequentially for a specific period of time.
- **Batch Processes**: For the sequential control of process steps in the food or chemical industries, e.g., filling, heating, stirring, cooling.
- **Test Stands**: Automated sequence of testing and measurement steps on a component.
- **Safety Sequences**: Orderly start-up and shutdown of a system, where steps may only be triggered after a waiting period.

⚖️ Comparison with Similar Components

Unlike a simple TON (Timer On-Delay) or TOF (Timer Off-Delay), which only control a single time-delayed switching operation, `sequence_T_08` orchestrates a complete chain of time-controlled actions. Compared to a Control Unit Transducer (CTU), it offers predefined, state-based logic with dedicated outputs and is therefore easier to configure and secure. It is a specialized form of Sequential Function Chart (SFC) with a fixed number of steps and pure timing control.

Conclusion

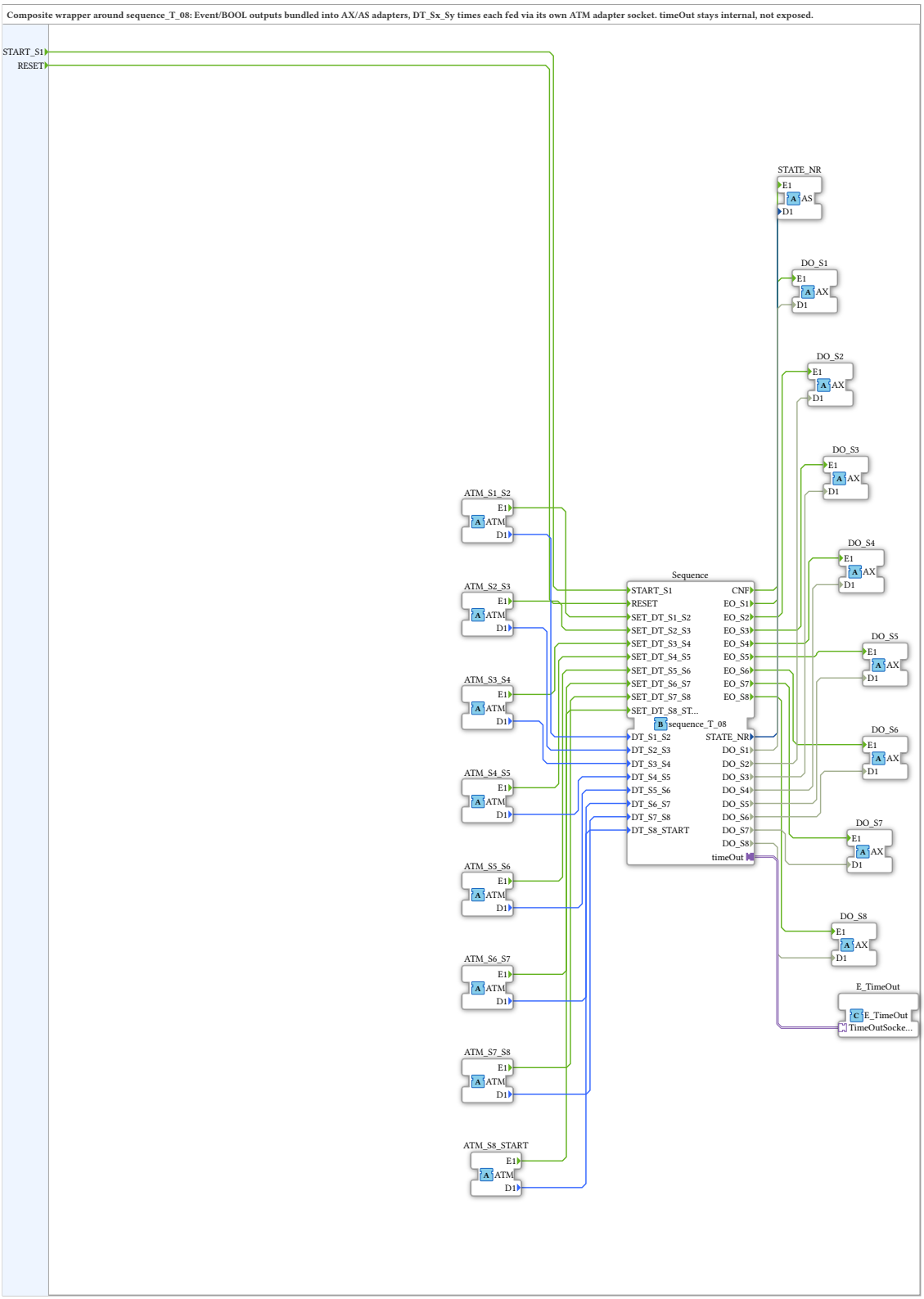
The `sequence_T_08`is a robust and easy-to-configure function block for all applications requiring a fixed, time-controlled sequence of steps. Its clear separation of state logic, time parameters, and output signals, along with its immediate reset functionality, makes it particularly suitable for well-organized and easily maintainable control programs. The integration of a standard timer adapter also makes it portable and reusable in various 4diac projects.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

sequence_T_08_ADAPTER

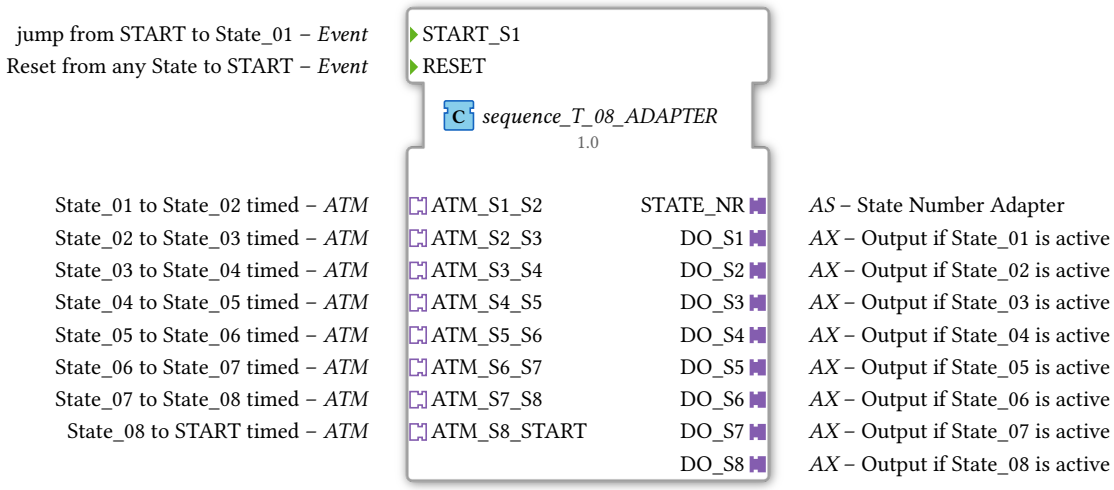


• — ○ • — ○ • — ○ •

Introduction

`sequence_T_08_ADAPTER` is a composite wrapper around sequence_T_08: the same generic time-controlled 8-step sequencer, but with every output and transition time moved onto adapter connections. Unlike sequence_T_08_AX, which only moves the eight outputs onto `AX` adapters, `sequence_T_08_ADAPTER`

additionally bundles the state number (AS) and feeds each of the eight transition times through its own ATM adapter socket.



Function Blocks Used (FBs)

Sub-modules: sequence_T_08_ADAPTER

- **Type**: Composite (FBNetwork)
- **Internal FBs used**:
  - **Sequence**: `logiBUS::utils::sequence::timed::sequence_T_08` - the generic 8-step time sequencer, see sequence_T_08.
  - **E_TimeOut**: `iec61499::events::E_TimeOut` - standard TimeOut adapter for the internal timing; unlike `AnlagenSequenz_06_ADAPTER`, a single standard FB is enough here, no separate `TimeTicker` needed.
- **Functionality**: `DO_S1`..`DO_S8` (per-state outputs) and `STATE_NR` (current state number) run through `AX`/`AS` adapters respectively; each of the eight transition times `DT_S1_S2`..`DT_S8_START` is fed through its own `ATM` adapter socket (`ATM_S1_S2`..`ATM_S8_START`) instead of an individual `TIME` data input.

Program Flow and Connections

1. `START_S1`/`RESET` pass through unchanged to `Sequence.START_S1`/`Sequence.RESET`.
2. Each `ATM_Sx_Sy` adapter delivers its time (`.D1`) to `Sequence.DT_Sx_Sy` and simultaneously (`.E1`) triggers the matching `Sequence.SET_DT_Sx_Sy` set event.
3. `Sequence.CNF` triggers `STATE_NR.E1`, with the current state number carried via `STATE_NR.D1`.
4. Each `Sequence.EO_Sx` triggers `DO_Sx.E1`, with the output value carried via `DO_Sx.D1`.
5. `Sequence.timeOut` (adapter socket) is connected internally to `E_TimeOut.TimeOutSocket` - the timing stays fully encapsulated inside the composite, unlike `sequence_T_08`/`sequence_T_08_AX`, which expose their own `timeOut` adapter.

Application Scenarios

- Applications that consistently rely on adapter connections and don't want to pull individual `TIME` lines for the transition times or a separate `timeOut` adapter connection in the application.
- Reusing the generic 8-step sequencer across multiple plant sections with uniform, adapter-based wiring.

⚖️ Comparison with Similar Function Blocks

- **sequence_T_08**: The base variant with classic event/data/adapter connections (only `timeOut` as an adapter).
- **sequence_T_08_AX**: Only moves the eight outputs onto `AX` adapters, but keeps `TIME` data inputs for the transition times and still exposes `timeOut` itself.
- **AnlagenSequenz_06_ADAPTER**: Same adapter-wrapper pattern, applied to the fixed 6-motor ring topology instead of the generic 8-step sequencer.

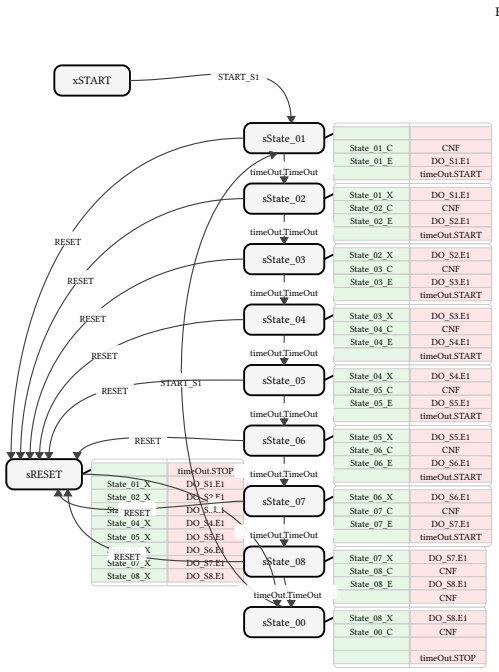
Conclusion

`sequence_T_08_ADAPTER` makes the generic 8-step time sequencer fully usable through adapter connections - outputs, state number, transition times, and timing all run exclusively through adapters, no individual event/data line leaves the block.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

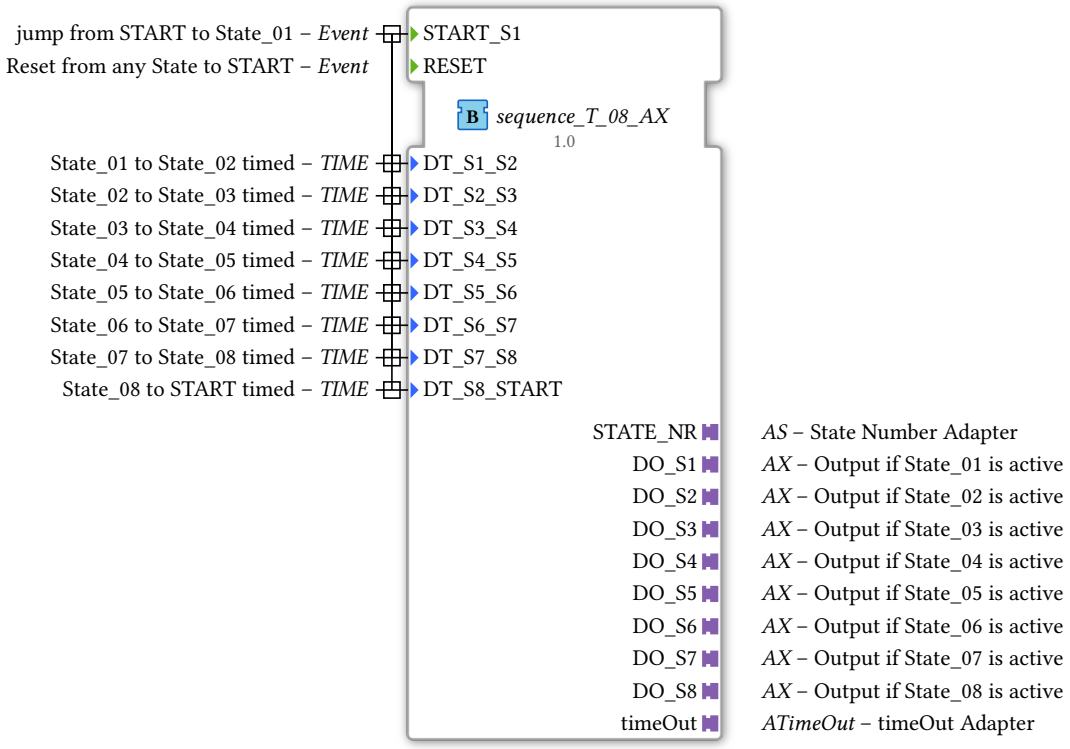
sequence_T_08_AX



• — ○ • — ○ • — ○ •

Introduction

sequence_T_08_AXis a variant ofsequence_T_08 that additionally uses adapters (AX`) for the outputs. It controls a purely time-based sequence with 8 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirms execution.

Data Inputs

- **DT_S1_S2**: Transition time State_01 -> State_02.
- **DT_S2_S3**: Transition time State_02 -> State_03.
- **DT_S3_S4**: Transition time State_03 -> State_04.
- **DT_S4_S5**: Transition time State_04 -> State_05.
- **DT_S5_S6**: Transition time State_05 -> State_06.
- **DT_S6_S7**: Transition time State_06 -> State_07.
- **DT_S7_S8**: Transition time State_07 -> State_08.
- **DT_S8_START**: Transition time State_08 -> START.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** … **DO_S8** (adapter::types::unidirectional::AX): Output adapter for State_01 to State_08.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

Corresponds to `sequence_T_08`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_T_08`.

Application Scenarios

For time-controlled 8-step sequences with adapter connection.

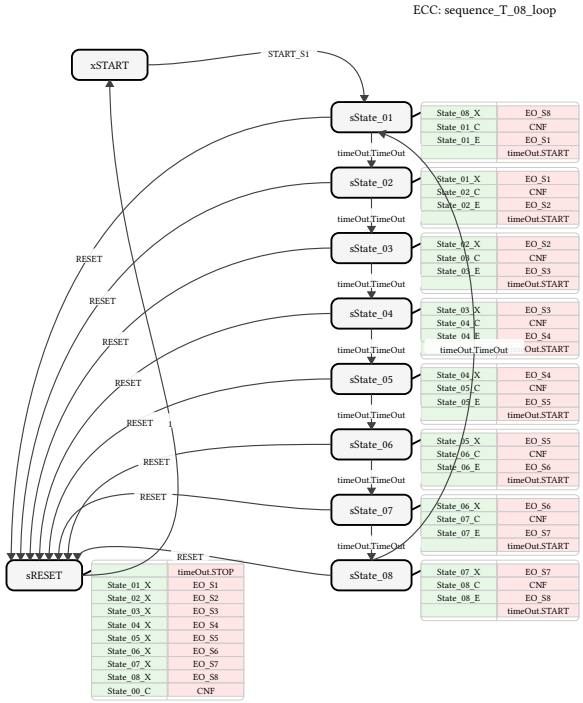
⚖️ Comparison with Similar Function Blocks

- **sequence_T_08**: Standard variant without adapters.

Conclusion

Adapter variant of the 8-step time sequencer.

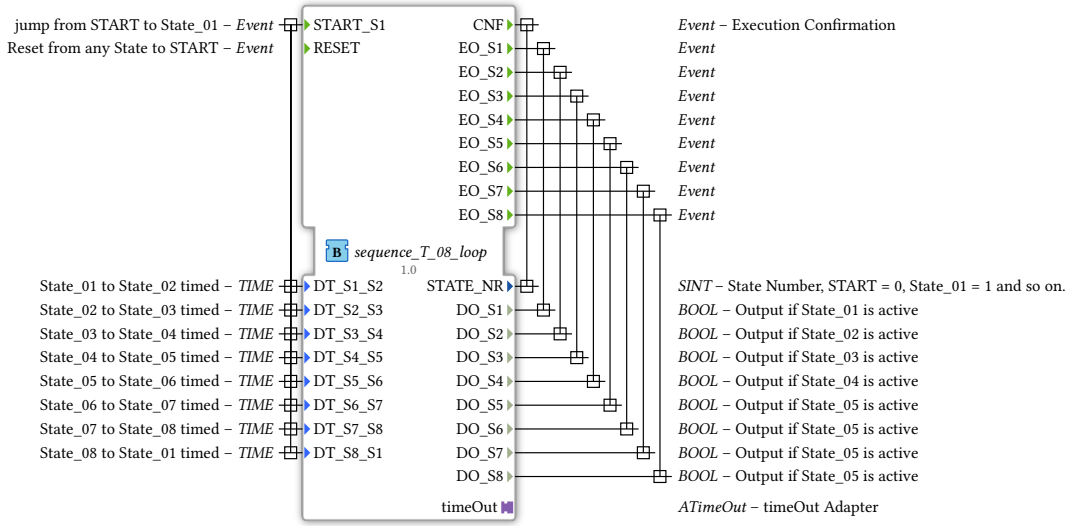
sequence_T_08_loop



• — ○ • — ○ • — ○ •

Introduction

The function block `sequence_T_08_loop` is a time-controlled sequencer with eight output states. It implements a cyclic sequence of states, with the transition between individual states controlled by adjustable time delays. This block is designed for applications where process steps or machine states need to be activated sequentially for a defined duration, for example, in conveyor systems, packaging machines, or automated assembly processes.



Interface Structure

Event Inputs

- **`START_S1`**: Starts the sequence by jumping from the initial state `START` to the first active state `State_01`. Triggers the execution of the associated algorithms.
- **`RESET`**: Immediately resets the sequence to the initial state `START`, regardless of the current state. All active outputs are deactivated.

Event Outputs

- **`CNF`** (Execution Confirmation): Triggered on every state change and confirms execution. Transmits the current state number `STATE_NR`.
- **`EO_S1` to `EO_S8`**: Event outputs triggered upon entering the respective state `State_01` to `State_08`. They transmit the corresponding Boolean data value (`DO_S1` to `DO_S8`), which is set to `TRUE`.

Data Inputs

Eight time-based data inputs of type `TIME`, defining the duration in each state. The default value is `NO_TIME` (no time, immediate transition).

* `DT_S1_S2`: Retention time in `State_01` before transition to `State_02`

- `DT_S2_S3`: Retention time in `State_02` before transition to `State_03`.
- `DT_S3_S4`: Retention time in `State_03` before transition to `State_04`.
- `DT_S4_S5`: Retention time in `State_04` before transition to `State_05`.
- `DT_S5_S6`: Retention time in `State_05` before transition to `State_06`.
- `DT_S6_S7`: Retention time in `State_06` before transition to `State_07`.
- `DT_S7_S8`: Retention time in `State_07` before transition to `State_08`.
- `DT_S8_S1`: Retention time in `State_08` before the cyclical transition back to `State_01`.

Data Outputs

- **`STATE_NR`** (SINT): Outputs the number of the currently active state. `START` = 0, `State_01` = 1, ..., `State_08` = 8.
- **`DO_S1` to `DO_S8`** (BOOL): The physical output signals of the sequence. Each output is set to `TRUE` when the corresponding state is active; otherwise, it is `FALSE`.

Adapter

- **`timeOut`** (Plug, Type: `iec61499::events::ATimeOut`): A timer adapter used to implement timed state transitions. The function block starts the timer upon entering a state and transitions to the next state upon receiving the `TimeOut` event.

Functionality

The function block operates as a Basic Function Block with an internal Execution Control Chart (ECC). The sequence begins in the initial state `xSTART`. A `START_S1` event leads to the state `sState_01`. Upon entering a state (e.g., `sState_01`), three actions are performed:

1. The exit algorithm of the previous state (e.g., `State_08_X`) disables its output.
2. The confirmation algorithm (e.g., `State_01_C`) sets the `STATE_NR` and loads the time configured for this state (`DT_S1_S2`) into the `timeOut` adapter.
3. The entry algorithm (e.g., `State_01_E`) sets the corresponding data output (`DO_S1`) to `TRUE` and triggers the corresponding event (`EO_S1`).
4. The timer adapter is started with `timeOut.START`.

After the set time has elapsed, the adapter triggers the `timeOut.TimeOut` event, which causes the transition to the next state in the ECC (e.g., from `sState_01` to `sState_02`). After the last state (`sState_08`), the system transitions back to the first state (`sState_01`), creating an infinite loop.

A `RESET` event from any state leads to state `sRESET`. Here, all outputs are deactivated, the timer is stopped, and an acknowledgment is sent with `STATE_NR=0` before the system automatically returns to state `xSTART`.

Technical Features

- **Non-Stop Cycle**: After starting, the sequence runs indefinitely in a loop until a `RESET` signal is received. There is no built-in stop command.
- **Immediate State Transitions**: By setting the time values to `NO_TIME` (default), the function block can be configured to immediately transition to the next state as soon as the entry algorithm of the current state has been executed.
- **Deterministic Execution**: The algorithms for Exit, Confirmation, and Entry are executed atomically in exactly this order upon state entry.
- **Constants for State Numbers**: The state number `STATE_NR` is set via constants from the imported state `sequence` (e.g., `sequence::State_01`), which improves maintainability and readability.

State Overview

The ECC consists of 10 states:

- **`xSTART`**: Initial, inactive state. Waiting for `START_S1`.
- **`sState_01` to `sState_08`**: The eight active sequence states. Each manages its own output and the time until the next state.
- **`sRESET`**: Reset state. Upon the `RESET` event, it jumps from any state, disables all outputs, and then returns to `xSTART`.

The transition conditions are:

- `START_S1`: From `xSTART` to `sState_01`.
- `timeOut.TimeOut`: From each state `sState_XX` to the next `sState_YY` (cyclically from `sState_08` to `sState_01`).
- `RESET`: From each active state (`sState_01` to `sState_08`) to `sRESET`.
- `1` (always true): From `sRESET` back to `xSTART`.

Application Scenarios

- **Control of Rotary Transfer Machines**: Activation of various tools or stations in a rotating machine for a precisely defined duration.
- **Batch Processes in Chemical Engineering**: Step-by-step control of valves, pumps, and heaters in a chemical process with fixed cycle times.
- **Automated Test Sequences**: Sequential execution of various measurements or tests on a component.
- **Light or Signaling Systems**: Generation of fixed flashing or running light patterns.

⚖️ Comparison with Similar Components

In contrast to aIn contrast to the `E_CYCLE` or `E_DELAY` function blocks, which generate simple periodic or delayed events, the `sequence_T_08_loop` function block offers a structured state machine with multiple independent outputs. Compared to a freely programmable `E_CTU` (counter) in combination with `SEL` blocks, this function block is preconfigured and therefore easier and faster to use for standard sequences

with up to eight steps. For more complex or state-dependent sequences, a Service Sequence Function Block (SFC) or custom Basic Function Block programming would be required.

This function block is preconfigured and therefore easier and faster to use for standard sequences with up to eight steps.## 🛠️ Related Exercises

- Exercise_038

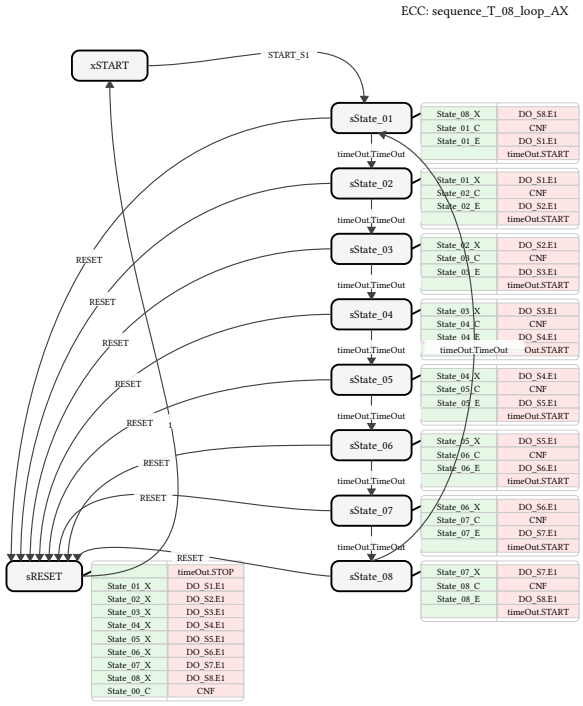
Conclusion

The `sequence_T_08_loop` is a robust and easy-to-configure function block for time-controlled sequences with a fixed number of steps. Its clear structure of states, configurable dwell times, and dedicated outputs makes it particularly suitable for standardized control tasks in automation technology where reliability and ease of parameterization are paramount. The integration of a timer adapter decouples the timing from the function block logic and promotes reusability.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter Block on ms-muc-docs.de

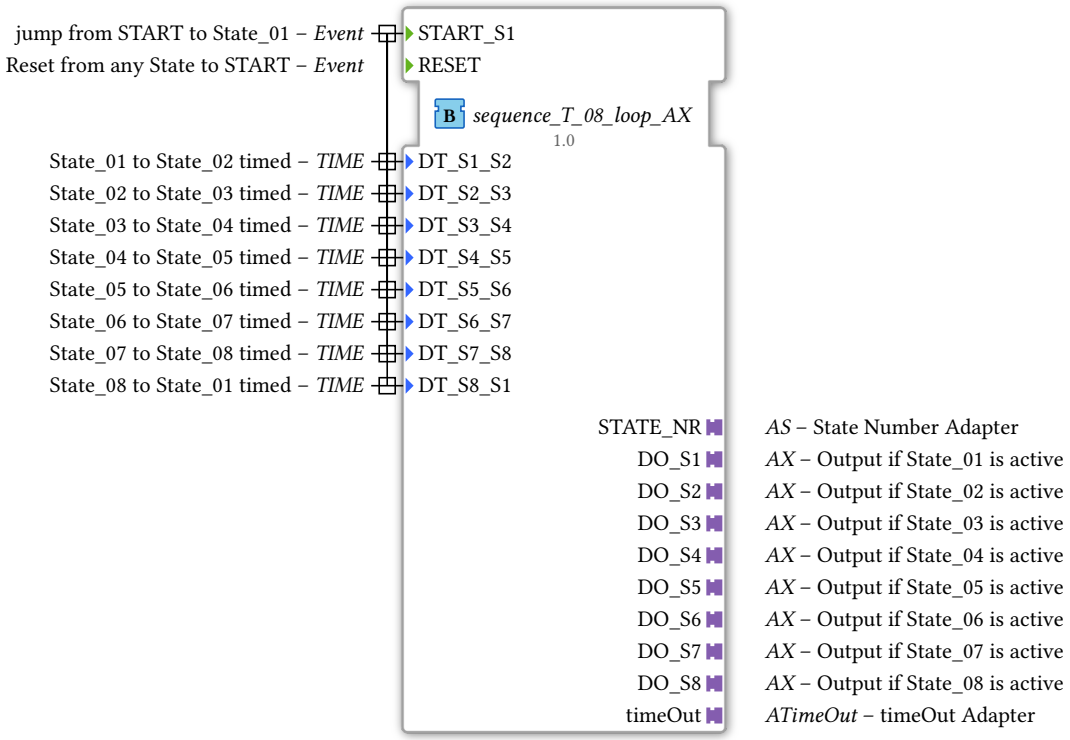
sequence_T_08_loop_AX



• – ○ • – ○ • – ○ •

Introduction

sequence_T_08_loop_AXis a variant ofsequence_T_08_loop that additionally uses adapters (AX`) for the outputs. It controls a purely time-controlled, cyclic sequence with 8 output states.



Interface Structure

Event Inputs

- **START_S1**: Starts the sequence at State_01.
- **RESET**: Resets the sequence.

Event Outputs

- **CNF**: Confirmation of execution.

Data Inputs

- **DT_S1_S2** ... **DT_S8_S1**: Times for automatic transitions between states.

Data Outputs

- **STATE_NR** (SINT): Current state number.

Adapters

- **DO_S1** ... **DO_S8** (adapter::types::unidirectional::AX): Output adapter for State_01 to State_08.
- **timeOut** (iec61499::events::ATimeOut): Timer adapter.

Functionality

Corresponds to `sequence_T_08_loop`, but uses adapters for the outputs.

Technical Features

- Uses `adapter::types::unidirectional::AX`.

State Overview

See `sequence_T_08_loop`.

Application Scenarios

For time-controlled, cyclic 8-step sequences with adapter connectivity.

⚖️ Comparison with similar modules

- **sequence_T_08_loop**: Standard version without adapter.

🛠️ Related Exercises

- Exercise_038_AX

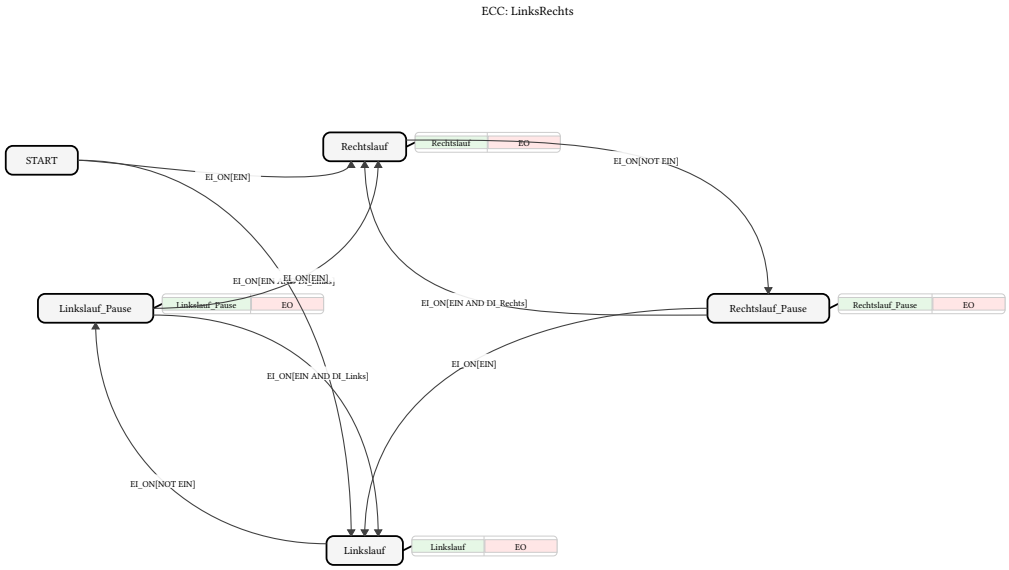
Conclusion

Adapter version of the 8-step loop time sequencer.

distribution list

...

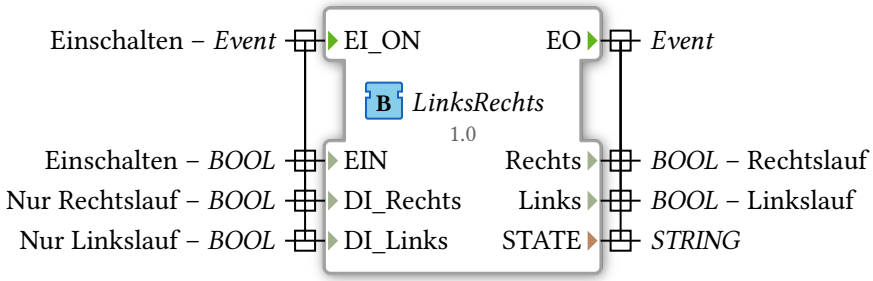
Left/Right



• – ○ • – ○ • – ○ •

Introduction

The function block `LinksRechts` is a fundamental building block for controlling bidirectional motion, such as a drive that can operate in both clockwise and counterclockwise directions. It implements a simple prioritization logic where clockwise rotation takes precedence over counterclockwise rotation unless a specific counterclockwise command is active. The block also allows pausing the motion.



Interface Structure

Event Inputs

- **`EI_ON`**: The central control event. Upon arrival, it triggers an evaluation of the current input data and a potential state transition.

Event Outputs

- **EO**: This event is triggered on every state change. It provides the updated output data Rechts, Links, and STATE.

Data Inputs

- **EIN** (BOOL): General enable/enable command. TRUE allows operation, while FALSE puts the block into a pause state.
- **DI_Rechts** (BOOL): Command for "Right-Hand Rotation Only". Enforces right-hand rotation if EIN is active.
- **DI_Links** (BOOL): Command for "Left Rotation Only". Enforces left rotation if EIN is active and no DI_Rechts command is present.

Data Outputs

- **Rechts** (BOOL): Control signal for right rotation. Is TRUE when the block is in state Rechtslauf.
- **Links** (BOOL): Control signal for left rotation. Is TRUE when the block is in state Linkslauf.
- **STATE** (STRING): Displays the current internal state of the function block as readable text (e.g., "Right Rotation", "Left Rotation_Pause").

Adapters

This function block does not use any adapters.

Functionality

The LinksRechts block is implemented as a Basic Function Block with Event Control (ECC). The arrival of the event EI_ON triggers an evaluation of the accompanying data EIN, DI_Rechts, and DI_Links. Based on the current combination of these values and the current state, a transition to a new state occurs.

In each state, a specific algorithm is executed that sets the output signals Rechts and Links and writes the state name to STATE. The output event EO is then generated to inform downstream blocks of the change.

The priority logic is defined as follows: If EIN is active (TRUE), DI_Rechts is checked first. If this is TRUE, forward rotation is activated. If DI_Rechts or FALSE, DI_Links is checked. If TRUE is present, forward scrolling is activated. If EIN or FALSE is present, the block enters a pause state, regardless of the scrolling commands.

Technical Features

- **Priority**: The specification emphasizes that "Forward scrolling only" (DI_Rechts) takes precedence over "Left scrolling only" (DI_Links). This is implemented in the ECC transition from START to Rechtslauf, which only requires EIN, while the transition to Linkslauf additionally requires DI_Links.
- **Status Output**: The output STATE is of type STRING and is fed from an imported enumeration STATES, which facilitates diagnosis and visualization.
- **Pause States**: There are two separate pause states (Rechtslauf_Pause and Linkslauf_Pause). These remember the last active direction. Upon reactivation (EIN=TRUE), the last active direction is resumed, unless a specific run command (DI_Rechts/DI_Links) is pending.

Pause States: Two separate pause states exist (Rechtslauf_Pause and Linkslauf_Pause). These remember the last active direction.

State Overview

The ECC (Execution Control Chart) of the function block comprises five states:

1. **START**: Initial state. This state is exited with the first EI_ON event.
2. **Rechtslauf**: Active state, in which the output signal Rechts is set to TRUE.
3. **Linkslauf**: Active state, in which the output signal Links is set to TRUE.
4. **Rechtslauf_Pause**: Pause state reached from Rechtslauf when EIN becomes FALSE. Both outputs are FALSE.
5. **Linkslauf_Pause**: Pause state reached from Linkslauf when EIN becomes FALSE. Both outputs are FALSE.

The transitions between states are triggered exclusively by the event EI_ON in combination with the data conditions.

Application Scenarios

Typical applications include:

- Controlling a dual-direction AC motor.
- Controlling a horizontally moving unit (e.g., a carriage, a gate).
- Any application where forward/reverse movement needs to be controlled with a general enable and individual direction commands.

⚖️ Comparison with Similar Function Blocks

Compared to a simple `SR` or `RS` flip-flop, `LinksRechts` offers a higher level of abstraction because it already encapsulates the prioritization logic and pause functionality. In contrast to a simple `E_SWITCH` block, which only switches between two outputs, `LinksRechts` additionally manages internal states (pause) and offers defined input prioritization.

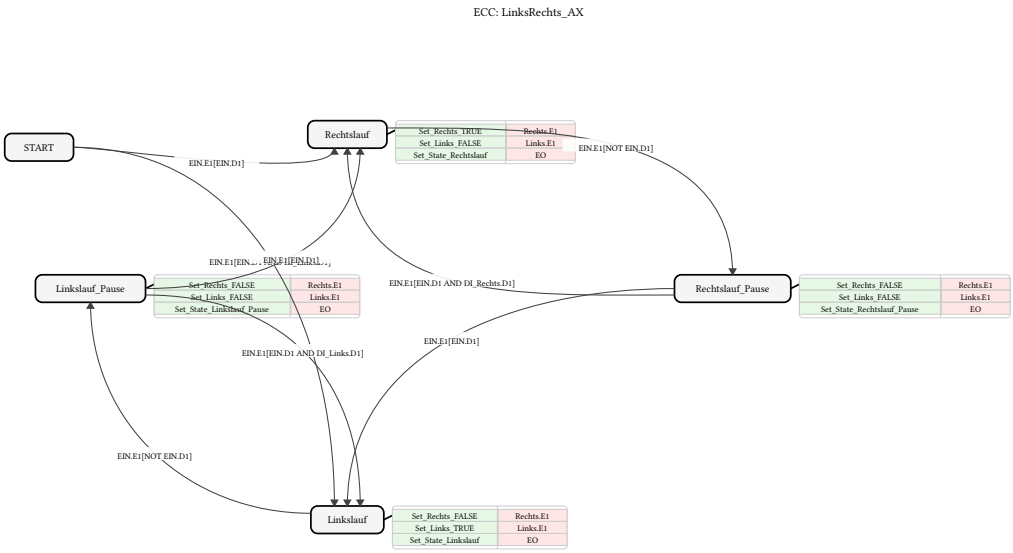
🛠️ Related Exercises

- Exercise_006a4

Conclusion

The `LinksRechts` function block is a useful and robust basic building block for controlling bidirectional movements. Its integrated prioritization logic (clockwise before counterclockwise) and state-based pause function simplify application programming and improve the clarity of control programs. Its clear interface and output status support commissioning and troubleshooting.

LeftRight_AX



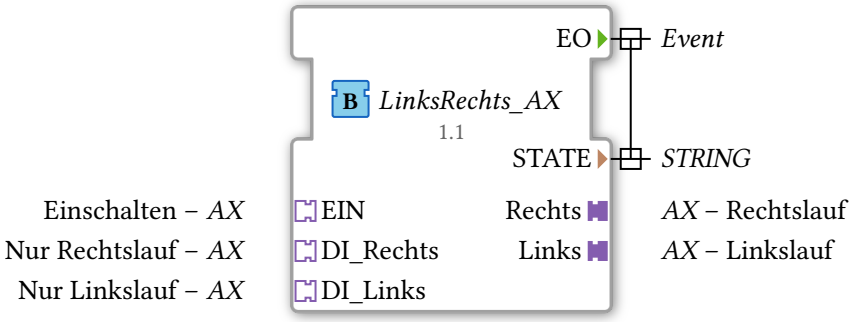
Note: An image of the function block is not available here.

• – ○ • – ○ • – ○ •

Introduction

The function block **LeftRight_AX** (from the package `logiBUS::utils::sequence::verteiler`) controls an alternating process with two directions (clockwise and counterclockwise rotation). It is designed to switch back and forth between two outputs, taking pause states into account.

Of particular note is the ability to prevent automatic switching via digital inputs (`DI_Rechts`, `DI_Links`) to force operation in only one direction. The block uses the `AX` adapter interface for this purpose.



Interface Structure

This function block primarily uses adapters for communication but also provides a status string as a direct output.

Event Inputs

There are no direct event inputs. Control is achieved via the events of the adapter sockets.

Event Outputs

Name	Type	Description
EO	Event	Event triggered when the internal state (`STATE`) changes.

Data Inputs

There are no direct data inputs. Data is read via the adapter sockets.

Data Outputs

Name	Type	Description
STATE	STRING	Current state of the function block (e.g., “Right Rotation”, “Left Rotation_Pause”).

Adapter

Sockets (Input Interfaces):

Name	Type	Comment
ON	adapter::types::unidirectional::AX	**Turn On**: The main signal to start and stop the movement.
DI_Right	adapter::types::unidirectional::AX	**Right Rotation Only**: When active, a change to left rotation is prevented, and right rotation is enforced.
DI_Left	adapter::types::unidirectional::AX	**Left Rotation Only**: When active, a change to right rotation is prevented, and left rotation is enforced.

Plugs (Output Interfaces):

Name	Type	Comment
Right	adapter::types::unidirectional::AX	**Right Rotation**: Output signal for rightward movement.
Left	adapter::types::unidirectional::AX	**Left Rotation**: Output signal for leftward movement.

Functionality

The **LeftRight_AX** function block implements a state machine that alternates between rightward and leftward movement, separated by pause phases.

1. **Start:** Starting from the `START` state, the function block decides, based on the inputs `EIN` and `DI_Links`, whether to switch to leftward or rightward rotation first.

2. **Activation (Run):** As long as the signal `EIN.D1` (data) is present together with an event `EIN.E1` as `TRUE`, the function block enters an active state (`Rechtslauf` or `Linkslauf`). The corresponding output adapter (`Rechts` or `Links`) is then set to `TRUE`.
3. **Deactivation (Pause):** When `EIN.D1` changes to `FALSE` (switch off), the function block switches to the corresponding pause state (`Rechtslauf_Pause` or `Linkslauf_Pause`). The outputs are deactivated (`FALSE`).
4. **Alternating Logic:**

- If the function block is in `Rechtslauf_Pause` and is switched on again (`EIN` = TRUE), it switches to **counterclockwise** by default.
- If the function block is in `Linkslauf_Pause` and is switched on again, it switches to **clockwise** by default.
- If the function block is in `Linkslauf_Pause` and is switched on again, it switches to **clockwise** by default. 5. **Override Logic (Forcing):**
- If input `DI_Rechts` is active in state `Rechtslauf_Pause`, the switch to counterclockwise rotation is prevented, and counterclockwise rotation is restarted.
- If input `DI_Links` is active in state `Linkslauf_Pause`, the switch to clockwise rotation is prevented, and counterclockwise rotation is restarted.

Technical Features

- **AX Adapter:** This function block uses the generic `unidirectional::AX` type. This typically combines a Boolean data signal (`D1`) with an event (`E1`).
- – **Prioritization:** According to the internal description, "clockwise rotation only takes precedence over counterclockwise rotation only," which is reflected in the start conditions. However, the sequence logic is primarily determined by the previous state (history).
- **Status Reporting:** Each state change updates the `STATE` variable and fires the `EO` event. The state names are obtained via an external enumeration (`STATES::...`).

State Overview

The ECC (Execution Control Chart) defines the following states:

State Name	Action	Description
START	-	Initial state. Waiting for the `EIN` signal.
clockwise rotation	`Set_Rechts_TRUE`, `Set_Links_FALSE`, Status update	Enables adapter `Rechts`, disables `Links`.
Right Rotation_Pause	`Set_Rechts_FALSE`, `Set_Links_FALSE`, Status update	Both outputs off. The system remembers that it was last in right rotation.
Left Rotation	`Set_Rechts_FALSE`, `Set_Links_TRUE`, Status update	Enables adapter `Links`, disables `Rechts`.
Left Rotation_Pause	`Set_Rechts_FALSE`, `Set_Links_FALSE`, Status update	Both outputs off. The system remembers that it was last on the left.

Application Scenarios

- **Pendulum Operation:** Automatic control of mechanisms that need to move back and forth (e.g., a windshield wiper mode or a cleaning head), controlled by a single button (`EIN`).
- **Irrigation Systems:** Sequential control of two sectors (Sector Right -> Pause -> Sector Left -> Pause), whereby a sector can be activated multiple times in succession if required (using `DI_Rechts`/`DI_Links`).
- **Reversing Motor:** Control of a motor that should change its direction of rotation every time it restarts, unless otherwise specified.

⚖️ Comparison with Similar Function Blocks

- **Simple Toggle (Flip-Flop):** A standard toggle switch simply turns one output on/off. `LinksRechts_AX` toggles between *two* outputs.

- **RS Gate:** An RS gate stores only one state based on set/reset. This function block incorporates sequence logic (history memory) because it remembers which state was active *before* the pause.
- **E_SELECT:** Similar to a selector, but `LinksRechts_AX` includes the timing component of the "pause" and automatic switching at the next start signal.

Conclusion

The **LeftRight_AX** function block is a specialized component for sequence control systems that require alternating operation between two outputs. Through the integration of the adapter technology (`AX`) and the ability to control the sequence through digital inputs (To influence `DI`), it offers a flexible solution for sequential control tasks with direction prioritization.

STATES



• – ◦ • – ◦ • – ◦ •

Introduction

The global constant declaration `STATES` defines a collection of predefined state labels as strings (STRING). These constants are typically used to name operating states in a control sequence, especially for processes with alternating direction (clockwise/counterclockwise) and integrated pauses. It serves as a central and consistent source for these state names within a project.

Interface Structure

Global constants are not function blocks with event or data interfaces. Instead, they declare constant values that are available project-wide.

Data Declarations

The following constants are defined:

- `Rechtslauf`: String with the value `'Rechtslauf'`
- `Linkslauf`: String with the value `'Linkslauf'`
- `Linkslauf_Pause`: String with the value `'Linkslauf_Pause'`
- `Rechtslauf_Pause`: String with the value `'Rechtslauf_Pause'`

Functionality

The constants themselves have no active functionality. They merely provide named values. Their main purpose is to avoid "magic numbers" or hard-coded strings in the program code. Using these constants makes the code more readable, maintainable, and less error-prone, as changes to the state names only need to be made in one central location.

Technical Details

- **Type:** All declared constants are of the data type `STRING`.
- **Scope:** As global constants, they are available within the entire namespace `logiBUS::utils::sequence::verteiler::LinksRechts` and in components that depend on it.
- **Initialization:** The values are set directly during declaration using the `InitialValue` attribute.

State Overview

Not applicable, as this is a static data declaration and not a stateful component.

Application Scenarios

Typical applications for these constants include:

- **State Machines:** These constants can serve as unique identifiers for the different states of a state machine that controls a process with changes of direction and pauses (e.g., a distribution table).
- **Documentation and Logging:** They can be used to display the current operating state in messages, log files, or on operator panels.

- **Comparisons and Switching Logic:** In program logic, variables or outputs can be compared with these constants to make decisions based on the current state.

⚖️ Comparison with Similar Function Blocks

- **Global Variables (VAR_GLOBAL):** Unlike global variables, the values of constants (`CONSTANT`) cannot be changed at runtime.
- **Enumerations (ENUM):** While ENUM data types serve similar purposes in some programming languages, the IEC 61499 standard library does not provide a built-in ENUM type. Using STRING constants is a common method for defining semantically named values in this environment.
- **Function Block-Internal Constants:** Constants can also be defined within function blocks. However, global constants are visible across block boundaries, thus promoting reusability and consistency throughout an entire project.

Conclusion

The global constant declaration `STATES` is a simple yet useful construct for structuring control software. It improves code quality by providing central and immutable state labels specifically tailored for sequences with forward/reverse and pause states. Its use makes programs more understandable and facilitates future modifications.

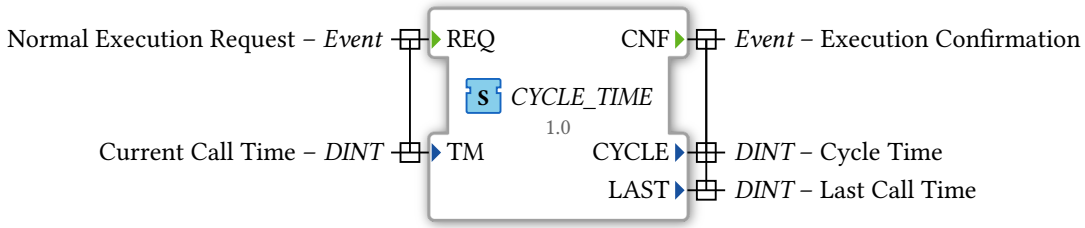
timing

CYCLE_TIME

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `CYCLE_TIME` is used to calculate the elapsed time between two consecutive calls. It is a useful tool for monitoring and diagnosing cycle times in real-time applications, for example, for performance analysis or for detecting unexpected delays in a control system.



Interface Structure

Event Inputs

- **REQ**: Starts the cycle time calculation. The algorithm is executed upon this event.

Event Outputs

- **CNF**: Triggered as soon as the calculation is complete. This event confirms execution and delivers the calculated values to the data outputs.

Data Inputs

- **TM** (DINT): Provides the current timestamp at the time of the call. This value is typically supplied by a system clock or a timer block. The initial value is `0`.

Data Outputs

- **CYCLE** (DINT): Contains the calculated cycle time, i.e., the difference between the current (`TM`) and the previous timestamp (`LAST`). The initial value is `0`.
- **LAST** (DINT): Stores the timestamp of the last call (`TM`). This value is used for the next calculation. The initial value is `0`.

Adapters

This function block does not have any adapter interfaces.

Functionality

Whenever the REQ event occurs, the corresponding algorithm REQ is executed. This algorithm performs two essential operations:

1. **Calculation of the cycle time**: The difference between the newly received time value TM and the stored value LAST (from the previous call) is calculated and output to CYCLE.
2. **Update of the timestamp**: The current input value TM is saved in the internal memory LAST for the next calculation cycle.

Immediately after the algorithm executes, the CNF event is triggered with the updated values at CYCLE and LAST.

Technical Features

- The function block is a **Simple Function Block (Basic FB)** according to IEC 61499.
- The data types used (DINT) are signed 32-bit integers suitable for representing large time values (e.g., in milliseconds or microseconds).
- The initial values of TM, CYCLE, and LAST are set to 0. The first valid CYCLE value after startup is therefore the difference TM - 0. This should be taken into account in the application's initialization logic.

State Overview

The function block has a single Execution Control State (EC State) named REQ. Upon entering this state, the algorithm REQ is always executed, followed by the CNF event.

Application Scenarios

- **Cycle Time Monitoring**: Continuous measurement of the execution time of a program or a specific program segment.
- **Performance Monitoring**: Identification of bottlenecks or unusually long cycle times in a control application.
- **Diagnostics**: Logging cycle times for analysis or to generate system runtime statistics.
- **Adaptive Control**: Providing the current cycle time for algorithms whose behavior depends on the available processing time.

⚖️ **Comparison with similar modules**

Compared to generic timing modules (such as E_DELAY or E_CYCLE), CYCLE_TIME specializes in **calculating the difference between consecutive time points**. It does not use its own timer but evaluates externally provided timestamps. Modules such as E_SR or E_R_TRIG serve fundamentally different purposes (storage, edge detection) and do not offer time difference calculation.

🛠️ **Related Exercises**

- Exercise_072c

Conclusion

The CYCLE_TIME function block is a simple yet effective tool for runtime measurement in IEC 61499-based control systems. Its clear interface and straightforward operation make it a robust component for diagnostic and monitoring tasks where the accuracy and reliability of cycle time measurement are paramount.

2.10.3.13 version

🎧 **Podcast**

- "Store Version" – Your key to managing object data pools in non-volatile VT storage (ISO 11783-6)
- Understanding VT Versions: Your path to becoming a master of agricultural and construction machinery mechatronics

fboot

Version

🎧 Podcast



- “Store Version” – Your key to managing object data pools in non-volatile VT memory (ISO 11783-6)](https://podcasters.spotify.com/pod/show/isobus-vt-objects/episodes/Store-Version–Dein-Schlssel-zur-Verwaltung-von-Objektdatenpools-im-nichtflchtigen-VT-Speicher-ISO-11783-6-e36vfh0)
- Understanding VT Versions: Your path to becoming a master of agricultural and construction machinery mechatronics](https://podcasters.spotify.com/pod/show/isobus-vt-objects/episodes/VT-Versionen-verstehen-Dein-Weg-zum-Meister-der-Land–und-Baumaschinenmechatronik-e36a8e8) The function block `version` facilitates the simple transmission of version information. It receives a string as input and outputs it unchanged. This block is useful for transporting version numbers or other identifiers within an application chain or for diagnostic purposes.
- **REQ**: This event triggers the processing of the function block. Upon its arrival, the value of the corresponding data input `VERSION` is copied to the data output `VERSIONO`.
- **CNF**: This event signals the successful completion of the operation. It is generated simultaneously with the data output `VERSIONO`.
- **VERSION** (STRING): This input expects the string to be passed on (e.g., a version number such as “1.2.3”).
- **VERSIONO** (STRING): This output outputs the received value from `VERSION` unchanged.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not have any adapter interfaces.

Functionality

The `version` block operates in an extremely simple and deterministic manner. Upon each occurrence of the `REQ` event, the current value of the input variable `VERSION` is copied to the output variable `VERSIONO`. Immediately following this assignment, the acknowledgment event `CNF` is generated along with the new output value. No formatting checks, manipulation, or storage of the value takes place.

- **Simple Pass-Through**: The block performs no transformations or logic on the data.
- **Direct Coupling**: The `REQ` and `CNF` events are directly coupled; the execution time is negligible.
- **No Internal State**: The block has no internal memory. Every execution depends solely on the current inputs.

The function block `version` is stateless. It remains in a single, passive state until a `REQ` event occurs. The response to this event (copying the data and triggering `CNF`) is instantaneous and does not result in a different state.

1. **Version Propagation**: In an application chain, a higher-level function block (e.g., a master or management function block) can pass its own version to this block to be sent to downstream diagnostic or logging functions.
2. **Data Consolidation**: In conjunction with other function blocks, it can be used to provide version information from various sources in a central location (e.g., an HMI or a network service).

- **`E_SR` or `E_RS` (Flip-Flops)**: These blocks have an internal state (set/reset) that is changed by events. The `version` block does not have such a state.
- **`E_MERGE`**: This block merges events from multiple inputs. The `version` block has only a single event input and does not process any event streams.
- **Simple data blocks (e.g., `E_R_TRIG`)**: These react to signal edges or perform timing operations. The `version` block performs only data copying.
- **Generic blocks like `COPY`**: A `COPY` block would perform a similar function (copying data from A to B) but is typically a pure data block without event control. The `version` block follows the standard IEC 61499 event/data flow paradigm.
-

The `version` function block is a minimalist and specialized tool for passing string information within an event-driven control system. Its strength lies in the simplicity and clarity of its purpose. For simple data copying without event logic, alternative implementations might be suitable; however, it is well-suited for integration into a standards-compliant IEC 61499 application with a clearly defined event flow.

Technical Features

State Overview

Application Scenarios

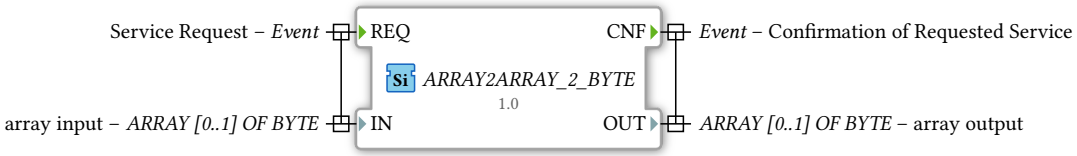
⚖️ **Vergleich mit ähnlichen Bausteinen**

Conclusion

2.11 Standard Libraries

2.11.1 convert

2.11.1.1 ARRAY2ARRAY_2_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`ARRAY2ARRAY_2_BYTE` is a service interface function block that copies a `BYTE` array of size 2 unchanged from `IN` to `OUT`. It belongs to the generic `GEN_ARRAY2ARRAY` family (cf. ARRAY2ARRAY_2_LREAL) and serves as a pure type/interface bridge between two `BYTE` arrays of the same size.

Interface Structure

Event Inputs

- **REQ**: Triggers the copy operation, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **IN** (`BYTE`, array size 2): The source array (8-bit bit pattern).

Data Outputs

- **OUT** (BYTE, array size 2): The destination array, identical values to IN.

Functionality

On REQ, the 2 elements of IN are copied unchanged to OUT, then CNF is triggered. No type conversion or value change takes place — the block exists solely to decouple two array variables of identical type and size.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY', the same C++ base as all other ARRAY2ARRAY_* variants (e.g. ARRAY2ARRAY_2_LREAL); only array size and data type differ per instantiation.
- **Fixed size 2**: For other array sizes of the same type see ARRAY2ARRAY_8_BYTE.

State Overview

Stateless: every REQ immediately results in a full copy and CNF.

Application Scenarios

- **Interface decoupling**: One block provides a BYTE array of size 2 as output, another expects a separate input array — ARRAY2ARRAY_2_BYTE acts as a bridge with no side effects.
- **Data routing** of fixed-size BYTE arrays between parts of a 4diac application.

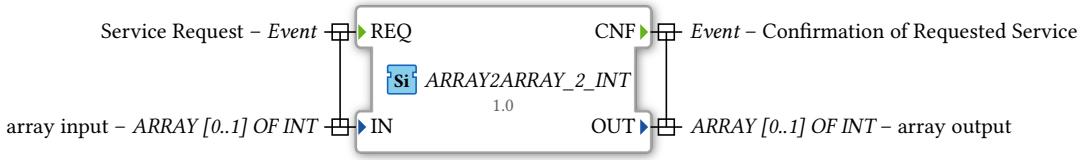
Comparison with similar function blocks

- ARRAY2ARRAY_2_LREAL: the same generic implementation for data type LREAL.
- **ARRAY2ARRAY_8_BYTE**: the same implementation with array size 8 instead of 2.
- **ARRAY2VALUES_2_BYTE**: splits the array into 2 individual scalar variables instead.

Conclusion

ARRAY2ARRAY_2_BYTE provides a simple, generically implemented one-to-one copy of a BYTE array of size 2 and serves as an uncomplicated interface bridge between like-typed array variables.

2.11.1.2 ARRAY2ARRAY_2_INT



• – ◦ • – ◦ • – ◦ •

Introduction

ARRAY2ARRAY_2_INT is a service interface function block that copies a INT array of size 2 unchanged from IN to OUT. It belongs to the generic GEN_ARRAY2ARRAY family (cf. ARRAY2ARRAY_2_LREAL) and serves as a pure type/interface bridge between two INT arrays of the same size.

Interface Structure

Event Inputs

- **REQ**: Triggers the copy operation, carries IN.

Event Outputs

- **CNF**: Confirms completion, carries OUT.

Data Inputs

- **IN** (INT, array size 2): The source array (16-bit signed integer).

Data Outputs

- **OUT** (INT, array size 2): The destination array, identical values to IN.

Functionality

On REQ, the 2 elements of IN are copied unchanged to OUT, then CNF is triggered. No type conversion or value change takes place — the block exists solely to decouple two array variables of identical type and size.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, the same C++ base as all other ARRAY2ARRAY_* variants (e.g. ARRAY2ARRAY_2_LREAL); only array size and data type differ per instantiation.
- **Fixed size 2**: For other array sizes of the same type see ARRAY2ARRAY_8_INT.

State Overview

Stateless: every REQ immediately results in a full copy and CNF.

Application Scenarios

- **Interface decoupling**: One block provides a INT array of size 2 as output, another expects a separate input array — ARRAY2ARRAY_2_INT acts as a bridge with no side effects.
- **Data routing** of fixed-size INT arrays between parts of a 4diac application.

Comparison with similar function blocks

- **ARRAY2ARRAY_2_LREAL**: the same generic implementation for data type LREAL.
- **ARRAY2ARRAY_8_INT**: the same implementation with array size 8 instead of 2.
- **ARRAY2VALUES_2_INT**: splits the array into 2 individual scalar variables instead.

Conclusion

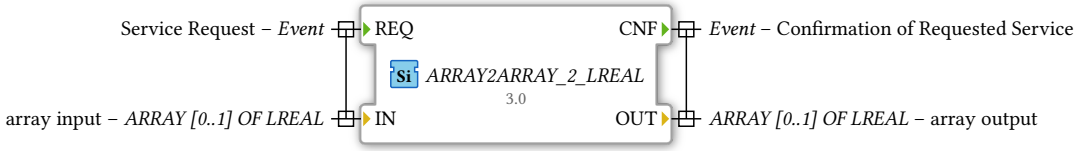
ARRAY2ARRAY_2_INT provides a simple, generically implemented one-to-one copy of a INT array of size 2 and serves as an uncomplicated interface bridge between like-typed array variables.

2.11.1.3 ARRAY2ARRAY_2_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) ARRAY2ARRAY_2_LREAL is a service interface function block designed to copy values from an input array to an output array. It serves as a simple, event-driven interface for transferring data between arrays within a 4diac application.



Interface Structure

Event Inputs

- **REQ** (Event): Triggers the copying process. This event signals the function block to start the data transfer.
- Associated with data variable: IN

Event Outputs

- **CNF** (Event): Confirms completion of the copy process. This event is triggered after successful data transfer.
- Associated with data variable: OUT

Data Inputs

- **IN** (Type: LREAL, Array Size: 2): The source array whose values are to be copied. This is a double-precision floating-point array of long real numbers.

Data Outputs

- **OUT** (Type: `LREAL`, Array Size: `2`): The target array into which the copied values are written. This is also an array of long real numbers.

Adapters

This function block does not use any adapters.

Functionality

The function block `ARRAY2ARRAY_2_LREAL` operates in an event-driven manner. As soon as the input event `REQ` is received, the function block reads the current values from the data input array `IN`. These values are then copied directly into the data output array `OUT`. After the copying process is complete, the output event `CNF` is triggered to signal the successful completion of the operation. The block thus processes a one-to-one copy of the array elements.

Technical Features

- **Service Interface:** The function block is designed as a "Service Interface Function Block Type," indicating a standard interface for specific services.
- **Generic Implementation:** Internally, this function block likely uses a generic implementation, as indicated by the attribute value `'GEN_ARRAY2ARRAY'`. This allows for the flexible creation of similar function blocks for different array sizes and data types.
- – **Fixed Array Size:** The array size for `IN` and `OUT` is fixed at `2`, limiting its use to specific scenarios with this array dimension.
- **Data Type `LREAL`:** The function block is specialized for the data type `LREAL` (Long Real Number), making it ideal for processing high-precision floating-point numbers.

State Overview

The function block typically exists in two main states:

1. **Waiting for Request:** The block waits for the `REQ` event. In this state, it is passive and performs no operations.
2. **Processing the Request:** After receiving `REQ`, the block switches to this state, executes the copy process from `IN` to `OUT`, and then emits `CNF` before returning to the "Waiting for Request" state.

Application Scenarios

- **Data Routing:** Forwarding `LREAL` data packets of size 2 between different parts of a control system.
- **Data Buffering:** Short-term storage or transmission of measured values or setpoints.
- **Interface Adaptation:** If a function block outputs a `LREAL` array of size 2 and another function block requires the same array as input, this function block can act as a bridge.
- – **Constant Data Transfer:** Transferring static or infrequently updated `LREAL` arrays from one point to another.

⚖️ Comparison with Similar Building Blocks

This building block is very specific in its function. Compared to more general data manipulation building blocks, which often allow flexible data types or array sizes via configuration parameters, `ARRAY2ARRAY_2_LREAL` is specifically designed for `LREAL` arrays of size 2. If building blocks for generic array copies existed (e.g., `ARRAY_COPY` with configurable size and type), this building block would be a specialized instance of one. It is simpler than such generic building blocks but less flexible. However, for its specific task, it offers a clear and efficient solution.

Metadata

Attribute	Value
Copyright	(c) 2014 Profactor GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2014-08-05, Matthias Plasch)

4diac Package	eclipse4diac::convert

Conclusion

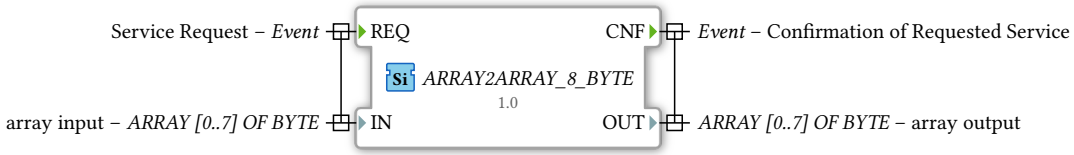
The `ARRAY2ARRAY_2_LREAL` function block provides a straightforward, event-driven method for copying `LREAL` arrays of size 2. Its specialization for this data type and array size makes it a useful tool for defined use cases where this precise data structure is required. It contributes to the modularity and readability of 4diac applications by providing an explicit function for this type of data transfer.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.1.4 ARRAY2ARRAY_8_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`ARRAY2ARRAY_8_BYTE` is a service interface function block that copies a `BYTE` array of size 8 unchanged from `IN` to `OUT`. It belongs to the generic `GEN_ARRAY2ARRAY` family (cf. ARRAY2ARRAY_2_LREAL) and serves as a pure type/interface bridge between two `BYTE` arrays of the same size.

Interface Structure

Event Inputs

- **REQ**: Triggers the copy operation, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **IN** (`BYTE`, array size 8): The source array (8-bit bit pattern).

Data Outputs

- **OUT** (`BYTE`, array size 8): The destination array, identical values to `IN`.

Functionality

On `REQ`, the 8 elements of `IN` are copied unchanged to `OUT`, then `CNF` is triggered. No type conversion or value change takes place — the block exists solely to decouple two array variables of identical type and size.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, the same C++ base as all other `ARRAY2ARRAY_*` variants (e.g. ARRAY2ARRAY_2_LREAL); only array size and data type differ per instantiation.
- **Fixed size 8**: For other array sizes of the same type see `ARRAY2ARRAY_2_BYTE`.

State Overview

Stateless: every `REQ` immediately results in a full copy and `CNF`.

Application Scenarios

- **Interface decoupling**: One block provides a BYTE array of size 8 as output, another expects a separate input array — ARRAY2ARRAY_8_BYTE acts as a bridge with no side effects.
- **Data routing** of fixed-size BYTE arrays between parts of a 4diac application.

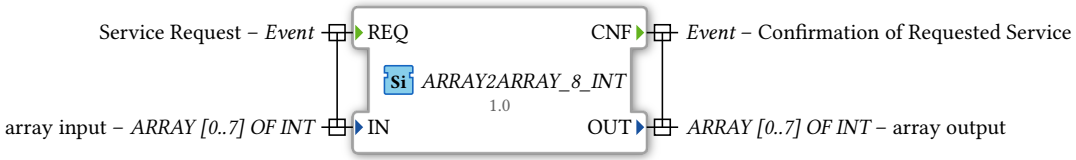
Comparison with similar function blocks

- ARRAY2ARRAY_2_LREAL: the same generic implementation for data type LREAL.
- **ARRAY2ARRAY_2_BYTE**: the same implementation with array size 2 instead of 8.
- **ARRAY2VALUES_8_BYTE**: splits the array into 8 individual scalar variables instead.

Conclusion

ARRAY2ARRAY_8_BYTE provides a simple, generically implemented one-to-one copy of a BYTE array of size 8 and serves as an uncomplicated interface bridge between like-typed array variables.

2.11.1.5 ARRAY2ARRAY_8_INT



• – ◦ • – ◦ • – ◦ •

Introduction

ARRAY2ARRAY_8_INT is a service interface function block that copies a INT array of size 8 unchanged from IN to OUT. It belongs to the generic GEN_ARRAY2ARRAY family (cf. ARRAY2ARRAY_2_LREAL) and serves as a pure type/interface bridge between two INT arrays of the same size.

Interface Structure

Event Inputs

- **REQ**: Triggers the copy operation, carries IN.

Event Outputs

- **CNF**: Confirms completion, carries OUT.

Data Inputs

- **IN** (INT, array size 8): The source array (16-bit signed integer).

Data Outputs

- **OUT** (INT, array size 8): The destination array, identical values to IN.

Functionality

On REQ, the 8 elements of IN are copied unchanged to OUT, then CNF is triggered. No type conversion or value change takes place — the block exists solely to decouple two array variables of identical type and size.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, the same C++ base as all other ARRAY2ARRAY_* variants (e.g. ARRAY2ARRAY_2_LREAL); only array size and data type differ per instantiation.
- **Fixed size 8**: For other array sizes of the same type see ARRAY2ARRAY_2_INT.

State Overview

Stateless: every REQ immediately results in a full copy and CNF.

Application Scenarios

- **Interface decoupling**: One block provides a `INT` array of size 8 as output, another expects a separate input array — `ARRAY2ARRAY_8_INT` acts as a bridge with no side effects.
- **Data routing** of fixed-size `INT` arrays between parts of a 4diac application.

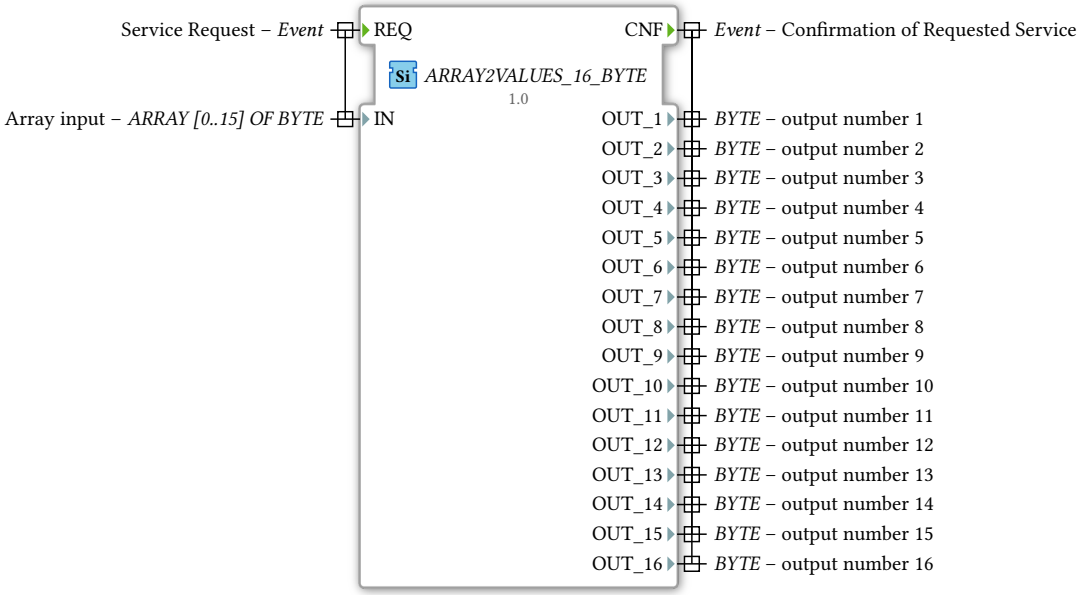
Comparison with similar function blocks

- ARRAY2ARRAY_2_LREAL: the same generic implementation for data type `LREAL`.
- **`ARRAY2ARRAY_2_INT`**: the same implementation with array size 2 instead of 8.
- **`ARRAY2VALUES_8_INT`**: splits the array into 8 individual scalar variables instead.

Conclusion

`ARRAY2ARRAY_8_INT` provides a simple, generically implemented one-to-one copy of a `INT` array of size 8 and serves as an uncomplicated interface bridge between like-typed array variables.

2.11.1.6 ARRAY2VALUES_16_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`ARRAY2VALUES_16_BYTE` splits a `BYTE` array of size 16 into 16 individual scalar variables `OUT_1`...`OUT_16`. It belongs to the generic `GEN_ARRAY2ARRAY` family (cf. ARRAY2VALUES_2_LREAL) and converts array data into discrete individual values.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8`, `OUT_9`, `OUT_10`, `OUT_11`, `OUT_12`, `OUT_13`, `OUT_14`, `OUT_15`, `OUT_16`.

Data Inputs

- **IN** (`BYTE`, array size 16): The array to split (8-bit bit pattern).

Data Outputs

- `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8`, `OUT_9`, `OUT_10`, `OUT_11`, `OUT_12`, `OUT_13`, `OUT_14`, `OUT_15`, `OUT_16` (`BYTE`): `OUT_i` corresponds to `IN[i-1]`, i.e. the i-th element of the array.

Functionality

On REQ, every element of IN is copied to the corresponding scalar output OUT_i (IN[0] ⟶ OUT_1, ..., IN[15] ⟶ OUT_16), then CNF is triggered.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, the same C++ base as ARRAY2VALUES_2_LREAL and the other ARRAY2VALUES_* variants.
- **Fixed size 16**: For other array sizes of the same type see ARRAY2VALUES_2_BYTE, ARRAY2VALUES_4_BYTE, ARRAY2VALUES_8_BYTE, ARRAY2VALUES_32_BYTE.

State Overview

Stateless: every REQ immediately results in a full split and CNF.

Application Scenarios

- **Data preparation**: An upstream block delivers a BYTE array, but a downstream block needs discrete individual variables.
- **Interface adaptation** between array-based and variable-based block interfaces.

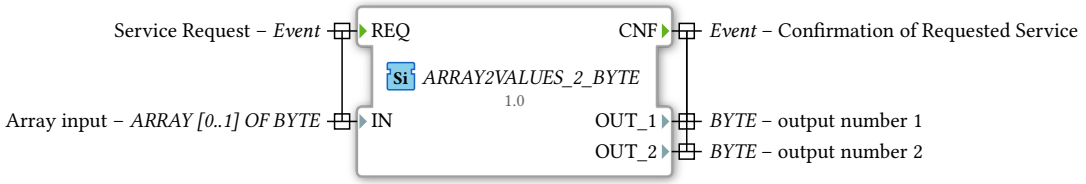
Comparison with similar function blocks

- **ARRAY2VALUES_2_LREAL**: the same generic implementation for data type LREAL.
- **VALUES2ARRAY_16_BYTE**: the reverse direction — combines 16 individual values into an array.
- **ARRAY2ARRAY_2_BYTE**: copies the array unchanged instead of splitting it.

Conclusion

ARRAY2VALUES_16_BYTE provides a simple, generically implemented split of a BYTE array of size 16 into 16 discrete individual variables and is suitable for adapting array-based to variable-based interfaces.

2.11.1.7 ARRAY2VALUES_2_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

ARRAY2VALUES_2_BYTE splits a BYTE array of size 2 into 2 individual scalar variables OUT_1...OUT_2. It belongs to the generic GEN_ARRAY2ARRAY family (cf. ARRAY2VALUES_2_LREAL) and converts array data into discrete individual values.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries IN.

Event Outputs

- **CNF**: Confirms completion, carries OUT_1, OUT_2.

Data Inputs

- **IN** (BYTE, array size 2): The array to split (8-bit bit pattern).

Data Outputs

- OUT_1, OUT_2 (BYTE): OUT_i corresponds to IN[i-1], i.e. the i-th element of the array.

Functionality

On REQ, every element of IN is copied to the corresponding scalar output OUT_i (IN[0] ⟶ OUT_1, ..., IN[1] ⟶ OUT_2), then CNF is triggered.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY', the same C++ base as ARRAY2VALUES_2_LREAL and the other ARRAY2VALUES_* variants.
- **Fixed size 2**: For other array sizes of the same type see ARRAY2VALUES_4_BYTE, ARRAY2VALUES_8_BYTE, ARRAY2VALUES_16_BYTE, ARRAY2VALUES_32_BYTE.

State Overview

Stateless: every REQ immediately results in a full split and CNF.

Application Scenarios

- **Data preparation**: An upstream block delivers a BYTE array, but a downstream block needs discrete individual variables.
- **Interface adaptation** between array-based and variable-based block interfaces.

Comparison with similar function blocks

- **ARRAY2VALUES_2_LREAL**: the same generic implementation for data type LREAL.
- **VALUES2ARRAY_2_BYTE**: the reverse direction — combines 2 individual values into an array.
- **ARRAY2ARRAY_2_BYTE**: copies the array unchanged instead of splitting it.

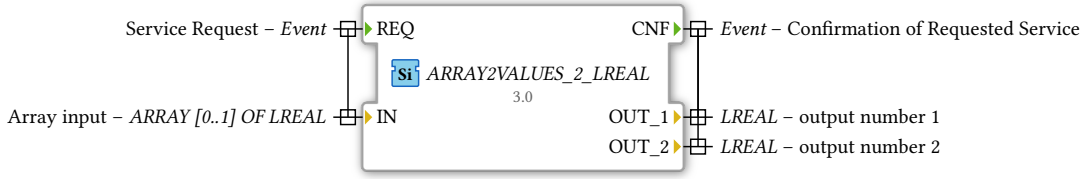
Conclusion

ARRAY2VALUES_2_BYTE provides a simple, generically implemented split of a BYTE array of size 2 into 2 discrete individual variables and is suitable for adapting array-based to variable-based interfaces.

2.11.1.8 ARRAY2VALUES_2_LREAL

• – ◦ • – ◦ • – ◦ •

The function block ARRAY2VALUES_2_LREAL is a service interface module designed to split an array of two LREAL values into two separate LREAL output variables. It facilitates the simple conversion from a structured array format to individual data points.



- **REQ**: A service request event that initiates the conversion. It is associated with the data input IN.
- **CNF**: An acknowledgment event that signals the completion of the requested service. It is associated with the data outputs OUT_1 and OUT_2.
- **IN**: An array input of type LREAL with a size of 2 elements. This array contains the two values to be split.
- **OUT_1**: The first output value of type LREAL, corresponding to the first element of the IN array.
- **OUT_1**: The first output value of type LREAL, corresponding to the first element of the IN array. * **OUT_2**: The second output value of type LREAL, corresponding to the second element of the IN array.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapter

Not present.

Functionality

When the `REQ` event is received, the `ARRAY2VALUES_2_LREAL` function block reads the two `LREAL` values from the `IN` array. The first element of the `IN` array (`IN[0]`) is assigned to the output `OUT_1`, and the second element (`IN[1]`) is assigned to the output `OUT_2`. Upon successful assignment, the `CNF` event is triggered to signal the completion of the operation.

This function block is specifically designed to process exactly one array of two `LREAL` values and split them into two separate `LREAL` variables. Its implementation is straightforward and optimized for this specific conversion purpose.

- **Generic Implementation:** Interestingly, the attribute `eclipse4diac::core::GenericClassName` references `'GEN_ARRAY2ARRAY'`. This suggests that the function block may be derived from a generic array-to-array class, even though its function is to split the array into individual values. This could be a detail of the implementation history.

This function block is stateless. Each triggering of the `REQ` event results in direct processing of the current `IN` values and output via `OUT_1` and `OUT_2` without any internal state changes beyond the current transaction.

- **Data Preparation**: When sensors or other components deliver data as an array of two `LREAL` values (e.g., X/Y coordinates, minimum/maximum values), but subsequent components require individual variables.
- **Interface Adaptation**: Adapting interfaces where a source component sends array data, but the target component expects discrete input variables.
- **Logic Simplification**: Increasing readability and simplifying logic by converting array accesses into individual variables that can be more easily processed in subsequent steps.

Similar components could be generic array splitters that can divide a variable number of array elements into a corresponding number of outputs. However, `ARRAY2VALUES_2_LREAL` is optimized for the specific case of a `LREAL` array of size 2, making it easier to configure and potentially more efficient for this particular use case than a more flexible, but potentially more complex, generic component.

⚖️ Vergleich mit ähnlichen Bausteinen

Application Scenarios

State Overview

Technical Features

Metadata

Attribute	Value
Copyright	(c) 2014 Profactor GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2014-07-09, Matthias Plasch)
4diac Package	eclipse4diac::convert

The ARRAY2VALUES_2_LREAL function block is a useful tool for the targeted conversion of a size 2 LREAL array into two separate LREAL outputs. It offers a simple and efficient solution for data preparation and interface adaptation in 4diac applications where this specific conversion is required.

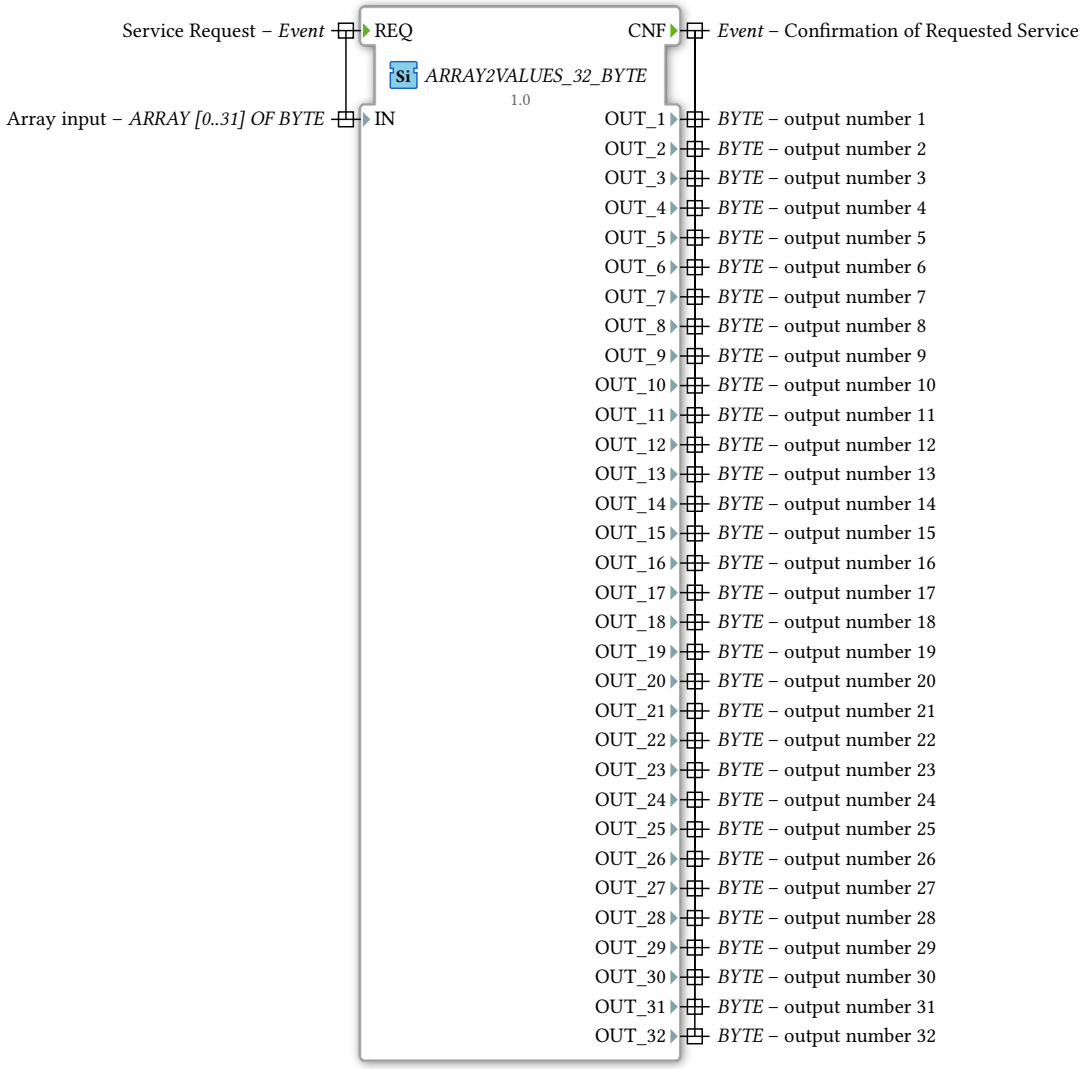
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Conclusion

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

2.11.1.9 ARRAY2VALUES_32_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

ARRAY2VALUES_32_BYTE splits a BYTE array of size 32 into 32 individual scalar variables OUT_1…OUT_32. It belongs to the generic GEN_ARRAY2ARRAY family (cf. ARRAY2VALUES_2_LREAL) and converts array data into discrete individual values.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries IN.

Event Outputs

- **CNF**: Confirms completion, carries `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8`, `OUT_9`, `OUT_10`, `OUT_11`, `OUT_12`, `OUT_13`, `OUT_14`, `OUT_15`, `OUT_16`, `OUT_17`, `OUT_18`, `OUT_19`, `OUT_20`, `OUT_21`, `OUT_22`, `OUT_23`, `OUT_24`, `OUT_25`, `OUT_26`, `OUT_27`, `OUT_28`, `OUT_29`, `OUT_30`, `OUT_31`, `OUT_32`.

Data Inputs

- **IN** (`BYTE`, array size 32): The array to split (8-bit bit pattern).

Data Outputs

- `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8`, `OUT_9`, `OUT_10`, `OUT_11`, `OUT_12`, `OUT_13`, `OUT_14`, `OUT_15`, `OUT_16`, `OUT_17`, `OUT_18`, `OUT_19`, `OUT_20`, `OUT_21`, `OUT_22`, `OUT_23`, `OUT_24`, `OUT_25`, `OUT_26`, `OUT_27`, `OUT_28`, `OUT_29`, `OUT_30`, `OUT_31`, `OUT_32` (`BYTE`): `OUT_i` corresponds to `IN[i-1]`, i.e. the i-th element of the array.

Functionality

On `REQ`, every element of `IN` is copied to the corresponding scalar output `OUT_i` (`IN[0]` ⟶ `OUT_1`, ..., `IN[31]` ⟶ `OUT_32`), then `CNF` is triggered.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, the same C++ base as ARRAY2VALUES_2_LREAL and the other `ARRAY2VALUES_*` variants.
- **Fixed size 32**: For other array sizes of the same type see `ARRAY2VALUES_2_BYTE`, `ARRAY2VALUES_4_BYTE`, `ARRAY2VALUES_8_BYTE`, `ARRAY2VALUES_16_BYTE`.

State Overview

Stateless: every `REQ` immediately results in a full split and `CNF`.

Application Scenarios

- **Data preparation**: An upstream block delivers a `BYTE` array, but a downstream block needs discrete individual variables.
- **Interface adaptation** between array-based and variable-based block interfaces.

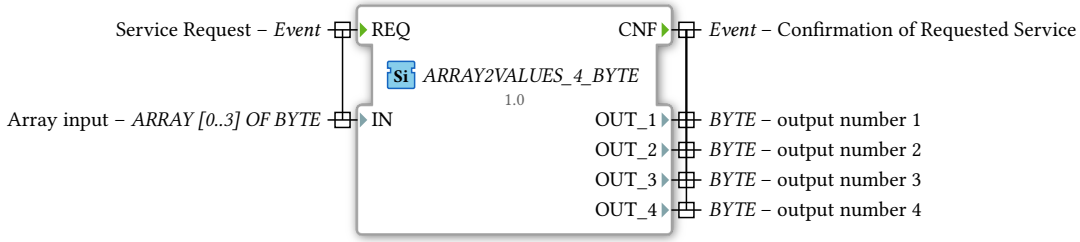
Comparison with similar function blocks

- **ARRAY2VALUES_2_LREAL**: the same generic implementation for data type `LREAL`.
- **`VALUES2ARRAY_32_BYTE`**: the reverse direction — combines 32 individual values into an array.
- **`ARRAY2ARRAY_2_BYTE`**: copies the array unchanged instead of splitting it.

Conclusion

`ARRAY2VALUES_32_BYTE` provides a simple, generically implemented split of a `BYTE` array of size 32 into 32 discrete individual variables and is suitable for adapting array-based to variable-based interfaces.

2.11.1.10 ARRAY2VALUES_4_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`ARRAY2VALUES_4_BYTE` splits a `BYTE` array of size 4 into 4 individual scalar variables `OUT_1`...`OUT_4`. It belongs to the generic `GEN_ARRAY2ARRAY` family (cf. ARRAY2VALUES_2_LREAL) and converts array data into discrete individual values.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`.

Data Inputs

- **IN** (`BYTE`, array size 4): The array to split (8-bit bit pattern).

Data Outputs

- `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4` (`BYTE`): `OUT_i` corresponds to `IN[i-1]`, i.e. the i-th element of the array.

Functionality

On `REQ`, every element of `IN` is copied to the corresponding scalar output `OUT_i` (`IN[0]` ⟶ `OUT_1`, ..., `IN[3]` ⟶ `OUT_4`), then `CNF` is triggered.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, the same C++ base as ARRAY2VALUES_2_LREAL and the other `ARRAY2VALUES_*` variants.
- **Fixed size 4**: For other array sizes of the same type see `ARRAY2VALUES_2_BYTE`, `ARRAY2VALUES_8_BYTE`, `ARRAY2VALUES_16_BYTE`, `ARRAY2VALUES_32_BYTE`.

State Overview

Stateless: every `REQ` immediately results in a full split and `CNF`.

Application Scenarios

- **Data preparation**: An upstream block delivers a `BYTE` array, but a downstream block needs discrete individual variables.
- **Interface adaptation** between array-based and variable-based block interfaces.

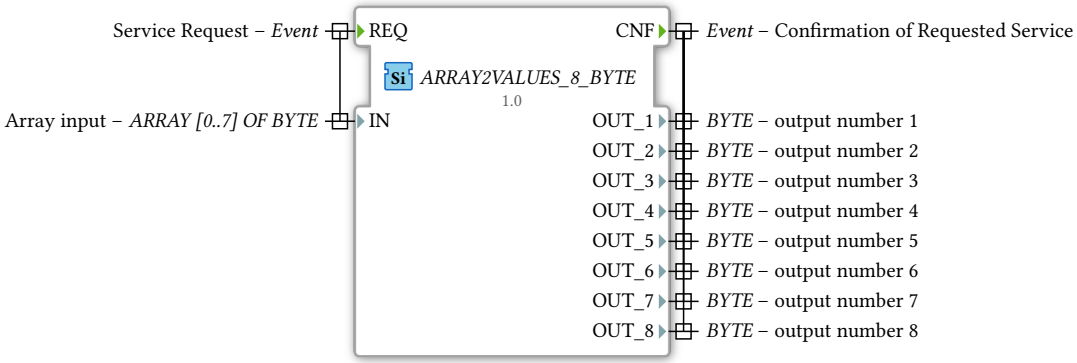
Comparison with similar function blocks

- **ARRAY2VALUES_2_LREAL**: the same generic implementation for data type `LREAL`.
- **`VALUES2ARRAY_4_BYTE`**: the reverse direction — combines 4 individual values into an array.
- **`ARRAY2ARRAY_2_BYTE`**: copies the array unchanged instead of splitting it.

Conclusion

`ARRAY2VALUES_4_BYTE` provides a simple, generically implemented split of a `BYTE` array of size 4 into 4 discrete individual variables and is suitable for adapting array-based to variable-based interfaces.

2.11.1.11 ARRAY2VALUES_8_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

ARRAY2VALUES_8_BYTE splits a BYTE array of size 8 into 8 individual scalar variables OUT_1...OUT_8. It belongs to the generic GEN_ARRAY2ARRAY family (cf. ARRAY2VALUES_2_LREAL) and converts array data into discrete individual values.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries IN.

Event Outputs

- **CNF**: Confirms completion, carries OUT_1, OUT_2, OUT_3, OUT_4, OUT_5, OUT_6, OUT_7, OUT_8.

Data Inputs

- **IN** (BYTE, array size 8): The array to split (8-bit bit pattern).

Data Outputs

- OUT_1, OUT_2, OUT_3, OUT_4, OUT_5, OUT_6, OUT_7, OUT_8 (BYTE): OUT_i corresponds to IN[i-1], i.e. the i-th element of the array.

Functionality

On REQ, every element of IN is copied to the corresponding scalar output OUT_i (IN[0] ⟶ OUT_1, ..., IN[7] ⟶ OUT_8), then CNF is triggered.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY', the same C++ base as ARRAY2VALUES_2_LREAL and the other ARRAY2VALUES_* variants.
- **Fixed size 8**: For other array sizes of the same type see ARRAY2VALUES_2_BYTE, ARRAY2VALUES_4_BYTE, ARRAY2VALUES_16_BYTE, ARRAY2VALUES_32_BYTE.

State Overview

Stateless: every REQ immediately results in a full split and CNF.

Application Scenarios

- **Data preparation**: An upstream block delivers a BYTE array, but a downstream block needs discrete individual variables.
- **Interface adaptation** between array-based and variable-based block interfaces.

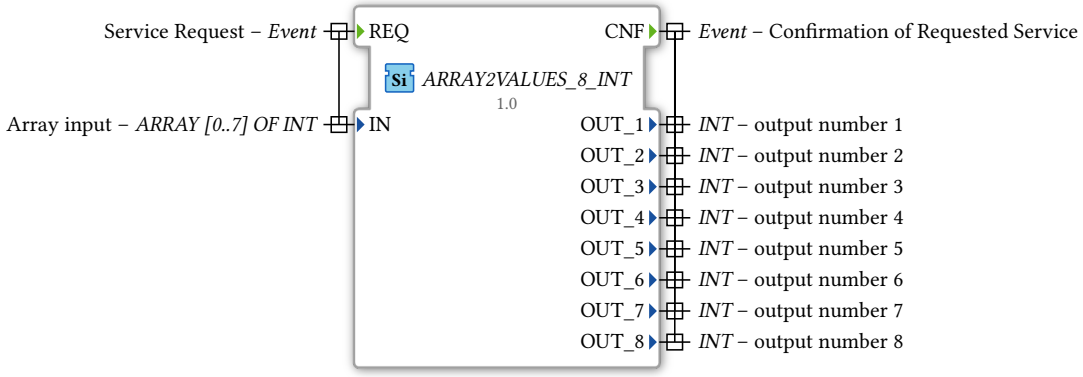
Comparison with similar function blocks

- **ARRAY2VALUES_2_LREAL**: the same generic implementation for data type LREAL.
- **VALUES2ARRAY_8_BYTE**: the reverse direction — combines 8 individual values into an array.
- **ARRAY2ARRAY_8_BYTE**: copies the array unchanged instead of splitting it.

Conclusion

ARRAY2VALUES_8_BYTE provides a simple, generically implemented split of a BYTE array of size 8 into 8 discrete individual variables and is suitable for adapting array-based to variable-based interfaces.

2.11.1.12 ARRAY2VALUES_8_INT



• – ◦ • – ◦ • – ◦ •

Introduction

`ARRAY2VALUES_8_INT` splits a `INT` array of size 8 into 8 individual scalar variables `OUT_1`...`OUT_8`. It belongs to the generic `GEN_ARRAY2ARRAY` family (cf. ARRAY2VALUES_2_LREAL) and converts array data into discrete individual values.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8`.

Data Inputs

- **IN** (`INT`, array size 8): The array to split (16-bit signed integer).

Data Outputs

- `OUT_1`, `OUT_2`, `OUT_3`, `OUT_4`, `OUT_5`, `OUT_6`, `OUT_7`, `OUT_8` (`INT`): `OUT_i` corresponds to `IN[i-1]`, i.e. the i-th element of the array.

Functionality

On `REQ`, every element of `IN` is copied to the corresponding scalar output `OUT_i` (`IN[0]` ⟶ `OUT_1`, ..., `IN[7]` ⟶ `OUT_8`), then `CNF` is triggered.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, the same C++ base as ARRAY2VALUES_2_LREAL and the other `ARRAY2VALUES_*` variants.
- **Fixed size 8**: For other array sizes of the same type see .

State Overview

Stateless: every `REQ` immediately results in a full split and `CNF`.

Application Scenarios

- **Data preparation**: An upstream block delivers a `INT` array, but a downstream block needs discrete individual variables.
- **Interface adaptation** between array-based and variable-based block interfaces.

Comparison with similar function blocks

- ARRAY2VALUES_2_LREAL: the same generic implementation for data type `LREAL`.
- **`VALUES2ARRAY_8_INT`**: the reverse direction — combines 8 individual values into an array.
- **`ARRAY2ARRAY_8_INT`**: copies the array unchanged instead of splitting it.

Conclusion

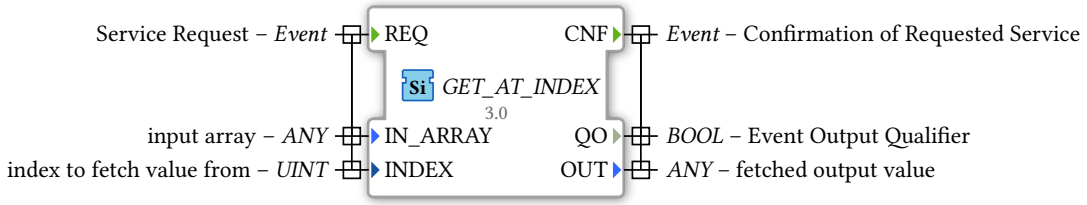
ARRAY2VALUES_8_INT provides a simple, generically implemented split of a INT array of size 8 into 8 discrete individual variables and is suitable for adapting array-based to variable-based interfaces.

2.11.1.13 GET_AT_INDEX

• – ◦ • – ◦ • – ◦ •

Introduction

This function block (FB) is used to retrieve a value from an array at a specific index position. An important prerequisite for correct operation is that the data type of the input array matches the data type of the output value. It is a fundamental building block for accessing array elements.



Interface Structure

The GET_AT_INDEX function block has a clearly defined interface structure with event and data inputs and outputs.

Event Inputs

- **REQ** (Type: Event)
- Service request to execute the function block.
- Is linked to the data inputs IN_ARRAY and INDEX, which are read when the event is triggered.

Event Outputs

- **CNF** (Type: Event)
- Acknowledgement of the requested service after successful processing.
- Is linked to the data outputs QO and OUT, which are provided after processing.

Data Inputs

- **IN_ARRAY** (Type: ANY)
- The input array from which a value is to be retrieved. The data type is generic (ANY) but must match the output.
- **IN_ARRAY** (Type: ANY)
- The input array from which a value is to be retrieved. The data type is generic (ANY) but must match the output.
- **INDEX** (Type: UINT)
- The index (position) within IN_ARRAY from which the value is to be retrieved.

Data Outputs

- **QO** (Type: BOOL)
- A qualifier for the event output, typically indicating the success or failure of the operation (e.g., TRUE for success, FALSE for failure).
- **OUT** (Type: ANY)
- The retrieved value from IN_ARRAY at the specified INDEX position. The data type is generic (ANY).

Adapters

This function block does not use any adapters.

Functionality

When a REQ event is received at the input of the function block, the GET_AT_INDEX function block reads the current values of the data inputs IN_ARRAY and INDEX. It then accesses IN_ARRAY and extracts the value at the position specified by INDEX. This retrieved value is then passed to the data output OUT. Simultaneously,

the output QO is set according to the result of the operation (e.g., TRUE for successful access). Finally, the CNF event is triggered to signal the termination of the service.

Technical Specifications

- **Data Type Match**: It is crucial that the data type of the input array (IN_ARRAY) is compatible with the expected data type of the retrieved value (OUT) to avoid type errors.
- **Index Range**: The value of INDEX must be within the valid limits of IN_ARRAY to prevent runtime errors or undefined behavior. Index range verification may need to be ensured by upstream or downstream logic.
- **ANY Type**: Using the ANY type for IN_ARRAY and OUT makes the function block very flexible, but requires careful typing in the application.

State Overview

The GET_AT_INDEX function block is essentially stateless with respect to data storage. Its states are limited to the execution cycle:

1. **Ready**: Waits for a REQ event.
2. **Executing**: Reads inputs, performs array access, and sets outputs.
3. **Completed**: Triggers the CNF event and returns to the "Ready" state.

Application Scenarios

- **Data Logging and Analysis**: Retrieving specific measured values from a time-series array.
- **Configuration Management**: Reading individual parameters from an array of configuration values.
- **List Processing**: Sequentially or randomly retrieving items from a list of tasks or objects.
- **Robot Control**: Accessing positions or motion parameters stored in arrays.
- **Visualization**: Retrieving data points for display on a user interface.

⚖️ Comparison with Similar Blocks

The GET_AT_INDEX block is a specialized block for array access. It differs from:

- **SET_AT_INDEX**: A complementary building block that sets values at a specific array position instead of retrieving them.
- **Array Iterators**: Building blocks that typically iterate through the entire array instead of retrieving a single element via a direct index.
- **General Data Manipulation Building Blocks**: These are not specifically designed for array access but perform more general operations (e.g., addition, comparison).

GET_AT_INDEX` thus fills a fundamental gap for direct, indexed read access to array data structures.

Metadata

Attribute	Value
Copyright	(c) 2014 Profactor GmbH
License	EPL-2.0
Version	Version 3.0 (April 14, 2025, Patrick Aigner), Version 1.0 (July 11, 2014, Matthias Plasch)
4diac package	eclipse4diac::convert

Conclusion

The GET_AT_INDEX function block is a basic and flexible building block that enables direct and indexed access to array elements. Its simplicity and support for generic data types (ANY) make it a versatile tool in numerous automation and control applications where data is stored in ordered lists or tables. Correct use requires only attention to data type compatibility and the valid index range.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.1.14 GET_STRUCT_VALUE

• – ○ • – ○ • – ○ •

Introduction

The function block (FB) `GET_STRUCT_VALUE` is a service interface module designed to dynamically read specific values from a given data structure. It provides a flexible way to access structure members whose names or paths are determined at runtime.



Interface Structure

Event Inputs

Name	Type	Description	Data with
`REQ`	`Event`	Requests the retrieval of a value from the structure.	`member`, `in_struct`

Event Outputs

Name	Type	Description	Data with
`CNF`	`Event`	Confirms the completion of the `REQ` event.	`QO`, `output`

Data Inputs

Name	Type	Description
`in_struct`	`ANY`	The data structure from which a value is to be read.
`member`	`STRING`	The name of the member of the structure to be retrieved. Inner members are accessed using the dot operator (`.`).

Data Outputs

Name	Type	Description
`QO`	`BOOL`	`TRUE` if no errors occurred, otherwise `FALSE`.
`output`	`ANY`	The value of the selected member if `QO = TRUE`. Otherwise, this value remains unchanged.

Adapters

No adapters available.

Functionality

The `GET_STRUCT_VALUE` function block is activated by a `REQ` event. Upon activation, the function block attempts to extract the value of a structure member specified in `member` from the data structure present at input `in_struct`. The path to the member can also specify nested members (e.g., “config.port”) using the dot operator (`.`). After reading, a `CNF` event is triggered, which provides the status (`QO`) and the extracted value (`output`). If an error occurs (e.g., member not found or invalid path), `QO` is set to `FALSE`.

Technical Features

- **Dynamic Member Access:** This function block allows access to structure members via a `STRING` input, offering high flexibility in data processing.
- **Nested Structures:** Members in nested structures can also be efficiently addressed using the dot operator (`.`) in the string `member`.
- **Type Independence:** The inputs and outputs `in_struct` and `output` are of type `ANY`, meaning that the function block can handle any data structure and member data type.

State Overview

The `GET_STRUCT_VALUE` is a stateless service function block that implements simple request acknowledgment logic (REQ/CNF). It processes each request individually and does not maintain any complex internal states.

Application Scenarios

- **Dynamic Configuration:** Reading configuration parameters from a central structure, where the required parameters are determined at runtime.
- **Generic Data Processing:** Developing generic algorithms or logic that need to access different parts of a data structure based on external inputs or rules.
- **Logging and Diagnostics:** Extracting specific status information or measured values from a complex system state structure for monitoring or diagnostic purposes.
- **Interfaces to External Systems:** Adapting to data formats of external systems where field names are passed dynamically.

⚖️ Comparison with Similar Building Blocks

There are no directly comparable function blocks in the 4diac standard library that allow dynamic access to structure members via a `STRING` path. This building block fills a niche for use cases requiring flexible, runtime-configurable structure manipulation that goes beyond the static access of conventional function blocks.

Metadata

Attribute	Value
Copyright	(c) 2018 fortiss GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2018-10-16, Jose Cabral)
4diac Package	eclipse4diac::convert

🛠️ Related Exercises

- Exercise_052

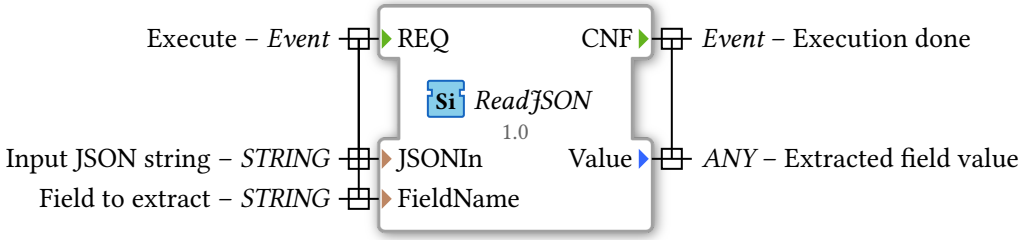
Conclusion

The `GET_STRUCT_VALUE` function block is a valuable tool for developers who need a flexible and dynamic method for reading values from data structures in 4diac. Its ability to access members via string paths, including nested structures, makes it particularly useful in scenarios where the structure of the required data is not fully known or is variable at compile time.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.1.15 ReadJSON



• – ◦ • – ◦ • – ◦ •

Introduction

The `ReadJSON` function block extracts the value of a named field from a JSON string. It serves as a simple, event-driven JSON decoder for use cases where a single field needs to be read from a JSON structure, without having to map the entire structure to dedicated IEC 61131-3 data types.

Interface Structure

Event Inputs

- **REQ**: Triggers the extraction, carries `JSONIn` and `FieldName`.

Event Outputs

- **CNF**: Confirms completion, carries `Value`.

Data Inputs

- **JSONIn** (STRING): The JSON string to search.
- **FieldName** (STRING): The name of the field to extract.

Data Outputs

- **Value** (ANY): The extracted field value, generically typed.

Functionality

On `REQ`, the block searches `JSONIn` for the field `FieldName` and outputs its value via `Value`. Since `Value` is of type `ANY`, the extracted value can be interpreted as a number, string, or boolean depending on the JSON content. `CNF` is then triggered.

Technical Features

- **ANY return type**: `Value` generically adapts to the actual JSON field type, but when wired to a concretely typed variable must be compatible with that type.
- **Single-field extraction**: The block reads exactly one named field per call; nested or multiple fields require several `ReadJSON` calls or instances.

State Overview

Stateless: every `REQ` immediately results in extraction and `CNF`.

Application Scenarios

- **Reading configuration**: A JSON configuration string received over the network or from a file needs to be evaluated without defining a full STRUCT.
- **Communicating with REST/web interfaces**: Responses from HTTP APIs that return JSON can be read field by field.

Comparison with similar function blocks

- **ToJSON**: the reverse direction — adds a field to a JSON string or creates a new one.
- **`GET_STRUCT_VALUE`**: reads a field from an IEC 61131-3 STRUCT instead of a JSON string.

Conclusion

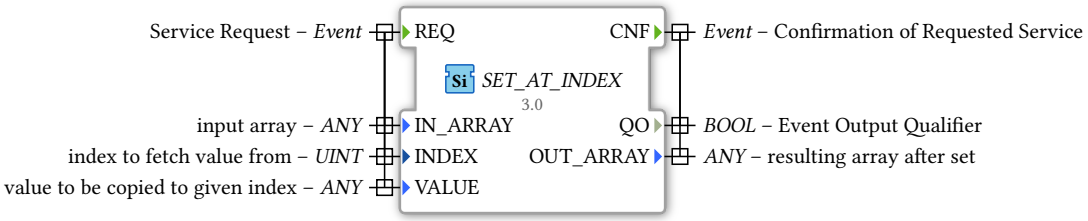
`ReadJSON` provides a simple, field-by-field JSON decoder for 4diac applications communicating with JSON-based configurations or web interfaces, without requiring a full structure type definition.

2.11.1.16 SET_AT_INDEX

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SET_AT_INDEX` is used to set the value of an element in an array at a specific index. It must be ensured that the data types of the value to be set match the data type of the array. The resulting output array will also have the same data type as the input array. This function block is useful for dynamically manipulating array data within a control program.



Interface Structure

Event Inputs

- **REQ** (Event): A request input that initiates the execution of the function block. This event processes the data inputs `IN_ARRAY`, `INDEX`, and `VALUE`.

Event Outputs

- **CNF** (Event): An acknowledgment output triggered after successful service execution. This event updates the data outputs `QO` and `OUT_ARRAY`.

Data Inputs

- **IN_ARRAY** (ANY): The input array whose value is to be modified at a specific index. The data type `ANY` allows its use with various array types.
- **INDEX** (UINT): The zero-based index of the element in the array whose value is to be set.
- **IN_ARRAY** (ANY): The input array whose value is to be modified at a specific index. * **VALUE** (ANY): The value to be set at the specified position in the array. The data type must match the data type of `IN_ARRAY`.

Data Outputs

- **QO** (BOOL): An event output qualifier that indicates the success of the operation. `TRUE` on success, `FALSE` on error (e.g., invalid index).
- **OUT_ARRAY** (ANY): The resulting array after the value has been set at the specified index.

Adapter

Not present.

Functionality

The `SET_AT_INDEX` function block waits for a `REQ` event. As soon as this event occurs, the function block checks the passed arrays `IN_ARRAY`, `INDEX`, and `VALUE`. It attempts to store `VALUE` at the address of `INDEX` within `IN_ARRAY`. If the operation is successful, a new array is created and output to `OUT_ARRAY`, setting `QO` to `TRUE`. If the specified `INDEX` is outside the valid range of `IN_ARRAY`, or if the data types of `VALUE` and the array elements do not match, the operation may fail, and `QO` could display `FALSE`.

Technical Features

- **Data Type Flexibility:** By using `ANY` for the array and value parameters, the function block is flexible with respect to the data types it can process. However, it is crucial that the data type of `VALUE` exactly matches the element data type of `IN_ARRAY` to avoid runtime errors.
- **Index Type:** The `INDEX` block is of type `UINT` (Unsigned Integer), meaning that only positive indices are allowed.
- **Zero-Based Indexing:** The index starts at 0 for the first element of the array.
- **Error Handling:** The `QO` output serves as a simple indicator of the operation's success. More advanced error handling (e.g., for index out-of-bounds errors) may need to be implemented at the application level or supplemented with additional checks.

State Overview

The `SET_AT_INDEX` block is essentially an event-driven block without complex internal state. It waits for a `REQ` event, executes the action, and outputs a `CNF` event. Its function is direct and reactive.

Application Scenarios

- **Dynamic Configuration:** Modifying configuration parameters stored in arrays at runtime.

- **Data Processing:** Updating sensor or actuator data collected in arrays.
- **Algorithmic Adjustment:** Modifying lookup tables or controller parameters in real time.
- **User Interaction:** Setting array values based on user input.

⚖️ Comparison with Similar Blocks

This block is complementary to the `GET_AT_INDEX` block, which would read values from an array at a specific index. It is a fundamental operation for array manipulation and forms the basis for more complex data structure algorithms. Unlike building blocks that copy or move entire arrays, `SET_AT_INDEX` focuses on the precise manipulation of a single element.

Metadata

Attribute	Value
Copyright	(c) 2014 Profactor GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2014-08-05, Matthias Plasch)
4diac Package	eclipse4diac::convert

Conclusion

The `SET_AT_INDEX` function block is a fundamental yet powerful tool for manipulating array data in 4diac applications. Its flexibility, thanks to the `ANY` data types and its clear event-driven interface, makes it a valuable component for many applications requiring dynamic data structures. Correct handling of data types and indices is crucial for its error-free use.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

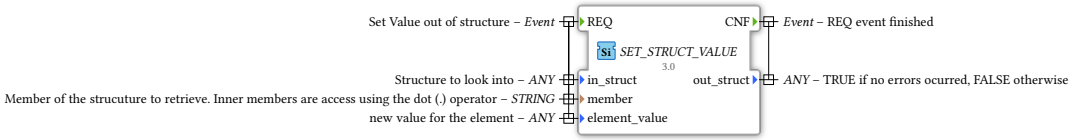
]

2.11.1.17 SET_STRUCT_VALUE

• – ○ • – ○ • – ○ •

Introduction

The function block `SET_STRUCT_VALUE` is a service interface function block used to set a specific value within a data structure. It enables the dynamic manipulation of structure elements by assigning a new value to a named member of the structure.



Interface Structure

The function block has event and data interfaces to enable its functionality.

Event Inputs

- **REQ**: This event requests the function block to set the value in the specified structure. It is associated with the data variables `member`, `in_struct`, and `element_value`.

Event Outputs

- **CNF**: This event is triggered when the `REQ` request is completed. It is associated with the data variable `out_struct`.

Data Inputs

- **in_struct** (Type: ANY): The input structure in which a value is to be changed. The type `ANY` allows the processing of any structure type.
- **member** (Type: STRING): The name of the structure member whose value is to be set. Inner members can be addressed using the dot operator (`.`) (e.g., "config.port").
- **element_value** (Type: ANY): The new value to be assigned to the specified structure member.

Data Outputs

- **out_struct** (Type: ANY): The modified structure after the element is set. The comment in the XML states "TRUE if no errors occurred, FALSE otherwise," which is a remnant of a `STATUS` output, but here it refers to the output, modified structure.

Adapters

No adapters are present.

Functionality

Upon receiving a `REQ` event, the function block accesses the data structure provided via `in_struct`. Based on the string passed as `member`, the corresponding element within the structure is identified. The function block then assigns the value provided via `element_value` to this element. The function block then outputs the modified structure via `out_struct` and signals the completion of the operation with the `CNF` event.

Technical Features

- **Generic Types (ANY)**: By using the data type `ANY` for `in_struct`, `element_value`, and `out_struct`, this function block offers high flexibility. It can work with any user-defined or predefined structure types and values without requiring specific types to be known at compile time.
- **Access to Nested Members**: The `member` variable supports access to deeper levels of a nested structure using dot notation (e.g., "outer.inner.value"). This simplifies the handling of complex data models.

State Overview

The function block primarily operates in a request-based (`REQ`) and confirmation-based (`CNF`) schema. It processes a request, modifies the data, and signals completion. There are no complex internal states beyond this simple request/confirm cycle.

Application Scenarios

- **Runtime Configuration Management**: Dynamically modify configuration parameters stored in structures without recompiling the function block.
- **Data Manipulation in SCADA/HMI Systems**: Updating specific values in complex data objects received from or sent to higher-level systems.
- **Interfaces to External Systems**: Adapting data structures used for communication with external APIs or databases.
- **Process Control**: Modifying process parameters organized in structures based on dynamic conditions or user input.

⚖️ Comparison with Similar Function Blocks

Compared to function blocks that support hard-coded structures or only read values, `SET_STRUCT_VALUE` offers unique flexibility through:

- **Write Access**: Allows setting values, not just reading.
- **Generic Access**: Works with any `ANY` type, making it more universal than function blocks bound to specific structure types.
- **Dynamic Member Selection**: The member name is passed as a string at runtime, allowing for more dynamic field selection than with fixed inputs/outputs.

It often complements function blocks that read values from structures (`GET_STRUCT_VALUE`) to provide complete read/write functionality for generic structures.

Metadata

Attribute	Value

Copyright	(c) 2023 Primetals Technologies Austria GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2023-03-06, Alois Zoitl)
4diac Package	eclipse4diac::convert

Conclusion

The `SET_STRUCT_VALUE` function block is a powerful tool for dynamically manipulating data structures within a 4diac application. Its generic nature and support for accessing nested members make it extremely versatile for applications that require flexible configuration or runtime data processing.

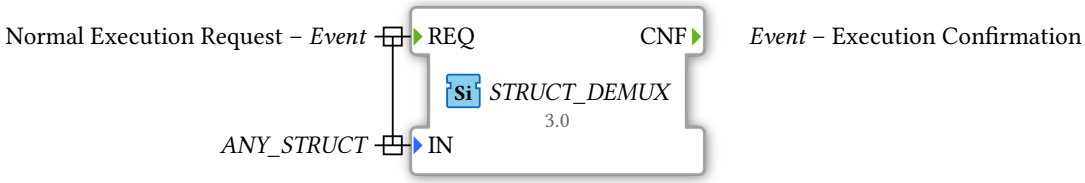
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.1.18 STRUCT_DEMUX

• – ◦ • – ◦ • – ◦ •

The function block (FB) `STRUCT_DEMUX` is a generic demultiplexer for structured data types. Its main task is to split an input data structure into its individual components (members). These individual members are then made available via separate data outputs, which are generated dynamically.



The interface of the `STRUCT_DEMUX` function block is defined generically. The actual data outputs are only determined during the instantiation of the block.

- **REQ**: Triggers the execution of the function block and causes the input structure to be read.
- **With Data Variable**: `IN`
- **CNF**: Confirms the completion of the operation after the structure's values have been passed to the outputs.
- **With Data Variables**: All dynamically generated data outputs.
- **IN** (Type: `ANY_STRUCT`): The input data structure to be split.

The data outputs of this function block are not predefined. Instead, they are created **dynamically** based on the data type associated with the `IN` input. For each member of the input structure, a corresponding data output with the same name and data type is created in the function block.

If a structure of type `MyStruct` with members `a` (type `INT`) and `b` (type `BOOL`) is connected to the `IN` input, the `STRUCT_DEMUX` block automatically generates two data outputs:

- `a` (type `INT`)
- `b` (type `BOOL`)

The image above illustrates this exact case.

- `a` (type `INT`)
- `b` (type `BOOL`)

...

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

As soon as a `REQ` event is received at the input of the `STRUCT_DEMUX` function block, the block reads the data structure present at the `IN` input. It extracts the values of each individual member of the structure and forwards them to the corresponding, dynamically generated data outputs. After all output values have been updated, the `CNF` event is triggered to signal the completion of the process.

- **Generic Block**: Thanks to the attribute `GEN_STRUCT_DEMUX`, the block is able to adapt to any structured data type (`ANY_STRUCT`).
- **Dynamic Interface**: The ability to generate its outputs based on the input data type makes it extremely flexible and reusable.
- **Service Interface Function Block Type**: The function block is designed as a standardized interface for this service.

The `STRUCT_DEMUX` is a stateless function block that operates according to a simple request-acknowledgment cycle:

1. **Ready**: Waits for a `REQ` event.
2. **Executing**: Reads the input structure, extracts the member values, and sets the corresponding outputs.
3. **Completed**: Triggers the `CNF` event and returns to the ready state.

- **Complex Data Splitting**: Decomposing complex data structures (e.g., sensor data, status information) into individual signals for further processing.
- **Improved Readability**: Instead of accessing structure members via `GET_STRUCT_VALUE`, the members can be used directly as separate data lines in the logic.
- **Interface Adaptation**: Adapting data arriving as a single structure from a function block to multiple function blocks that expect individual inputs.
- **`GET_STRUCT_VALUE`**: While `GET_STRUCT_VALUE` dynamically extracts a single member via a `STRING` name, `STRUCT_DEMUX` statically exposes all members as separate outputs. `STRUCT_DEMUX` is often easier to use when all members are needed, while `GET_STRUCT_VALUE` is more flexible when only specific members need to be addressed at runtime.
- **`STRUCT_MUX`**: The complementary component that combines individual data inputs into a single data structure.

⚖️ Vergleich mit ähnlichen Bausteinen

Application Scenarios

State Overview

Technical Features

Metadata

Attribute	Value
Copyright	(c) 2020 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::convert

- Uebung_051
- Uebung_120
- Uebung_121
- Uebung_122
- Uebung_122b
- Uebung_123
- Uebung_124
- Uebung_125
- Uebung_126
- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_130
- Uebung_131
- Uebung_132
- Uebung_133
- Uebung_134

STRUCT_DEMUX` is a fundamental and extremely useful building block for working with data structures in 4diac. Its ability to automatically decompose any structure into its constituent parts significantly simplifies application logic and promotes clear, readable wiring. It is the standard tool for accessing the contents of structures.

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

🛠️ Zugehörige Übungen

Conclusion

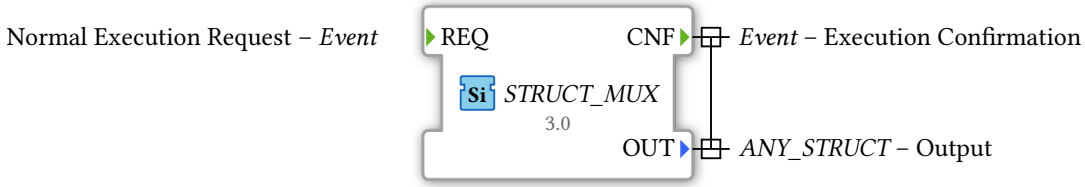
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

2.11.1.19 STRUCT_MUX

• – ◦ • – ◦ • – ◦ •

Introduction

The function block (FB) `STRUCT_MUX` is a generic multiplexer for structured data types. It functions as the counterpart to `STRUCT_DEMUX` and is responsible for combining individual data values from multiple inputs into a single data structure at the output.



Interface Structure

The interface of the `STRUCT_MUX` function block is defined generically. The actual data inputs are only determined during the instantiation of the block.

Event Inputs

- **REQ**: Triggers the execution of the function block. This causes the values at the inputs to be read and the output structure to be assembled.
- **With Data Variables**: All dynamically generated data inputs.

Event Outputs

- **CNF**: Confirms the completion of the operation after the output structure has been successfully created.
- **With Data Variable**: `OUT`

Data Inputs

The data inputs of this function block are not predefined. Instead, they are created **dynamically** based on the data type specified for the `OUT` output. For each member of the output structure, a corresponding data input with the same name and data type is created in the function block.

Example:

If the output `OUT` is set to the data type `MyStruct` with the members `a` (type `INT`) and `b` (type `BOOL`), the `STRUCT_MUX` block automatically creates two data inputs:

- `a` (type `INT`)
- `b` (type `BOOL`)

The image above illustrates this exact case.

- `a` (type `INT`)
- `b` (type `BOOL`)

...

Data Outputs

- **OUT** (Type: `ANY_STRUCT`): The output data structure, which is assembled from the values of the inputs.

Functionality

As soon as a `REQ` event is received at the input of the `STRUCT_MUX` function block, the block reads the values from all its dynamically generated data inputs. These values are then combined into a single data structure. The resulting structure is made available at the `OUT` output. Immediately afterward, the `CNF` event is triggered to signal the completion of the process.

Technical Features

- **Generic Block**: Thanks to the attribute `GEN_STRUCT_MUX`, the block can adapt to any structured data type (`ANY_STRUCT`).
- **Dynamic Interface**: The ability to generate its inputs based on the output data type makes it extremely flexible and reusable.
- **Service Interface Function Block Type**: The block is designed as a standardized interface for this service.

State Overview

The `STRUCT_MUX` is a stateless block that operates according to a simple request-acknowledgment cycle:

1. **Ready**: Waits for a `REQ` event.
2. **Executing**: Reads the input values and assembles the output structure.
3. **Completed**: Provides the structure at the `OUT` output, triggers the `CNF` event, and returns to standby mode.

Application Scenarios

- **Data Consolidation**: Bundles individual signals (e.g., configuration parameters, control values) into a single structure for further processing or communication.
- **Improved Clarity**: Combines many individual data lines into a single structured line to reduce the complexity of graphical representations.
- **Interface Adaptation**: Adapts individual data values to a function block that expects a single data structure as an input.

⚖️ Comparison with Similar Function Blocks

- **`STRUCT_DEMUX`**: The complementary function block that splits a single data structure into its individual members.
- **`SET_STRUCT_VALUE`**: While `SET_STRUCT_VALUE` dynamically modifies a single value within an existing structure, `STRUCT_MUX` creates a completely new structure from individual values.

Metadata

Attribute	Value
Copyright	(c) 2020 Johannes Kepler University Linz

License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::convert

🛠️ Related Exercises

- Exercise_051
- Exercise_052
- Exercise_121
- Exercise_126b2_sub
- Exercise_126b_sub

Conclusion

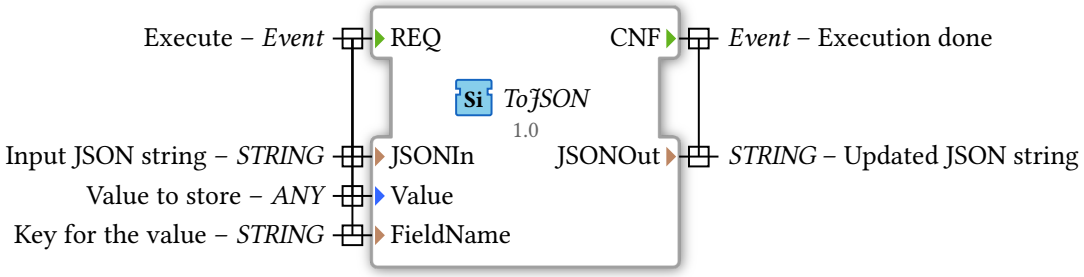
STRUCT_MUX` is a fundamental and very practical building block for working with data structures in 4diac. Its ability to generate any structure from individual values simplifies application logic and promotes clean, structured data management. It is the standard tool for bundling data and generating structured information.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.1.20 ToJSON



• – ◦ • – ◦ • – ◦ •

Introduction

The `ToJSON` function block adds a named field with a value to a JSON string. If `JSONIn` is empty, a new JSON object containing exactly that one field is created — by chaining several `ToJSON` calls (`JSONOut` of one as `JSONIn` of the next), a complete JSON object can be built up step by step.

Interface Structure

Event Inputs

- **REQ**: Triggers the addition, carries `JSONIn`, `Value`, and `FieldName`.

Event Outputs

- **CNF**: Confirms completion, carries `JSONOut`.

Data Inputs

- **JSONIn** (STRING): The existing JSON string a field is added to (an empty string creates a new object).
- **Value** (ANY): The value to store, generically typed.
- **FieldName** (STRING): The key (field name) under which `Value` is stored.

Data Outputs

- **JSONOut** (STRING): The updated JSON string with the newly added field.

Functionality

On `REQ`, the block checks `JSONIn`: if the string is empty, a new JSON object `{"FieldName": Value}` is created; otherwise the field `FieldName` with `Value` is inserted into the existing object (overwriting an existing field of the same name, if any). The result is output via `JSONOut`, then `CNF` is triggered.

Technical Features

- **ANY input type**: `Value` generically accepts numbers, strings, and boolean values, which are translated to the corresponding JSON data type.
- **Chainable**: Since `JSONOut` has the same type as `JSONIn`, multiple `ToJSON` instances can be chained to build up a JSON object with several fields step by step.
- **Empty `JSONIn` creates a new object**: No separate "create" block is needed — the first call in a chain can start directly with an empty `JSONIn`.

State Overview

Stateless: every `REQ` immediately results in the field being inserted and `CNF`.

Application Scenarios

- **Building JSON payloads**: A JSON object is assembled step by step from several process values for sending to a REST API or an MQTT topic.
- **Logging**: Structured log entries in JSON format, with individual fields added from different parts of a program.

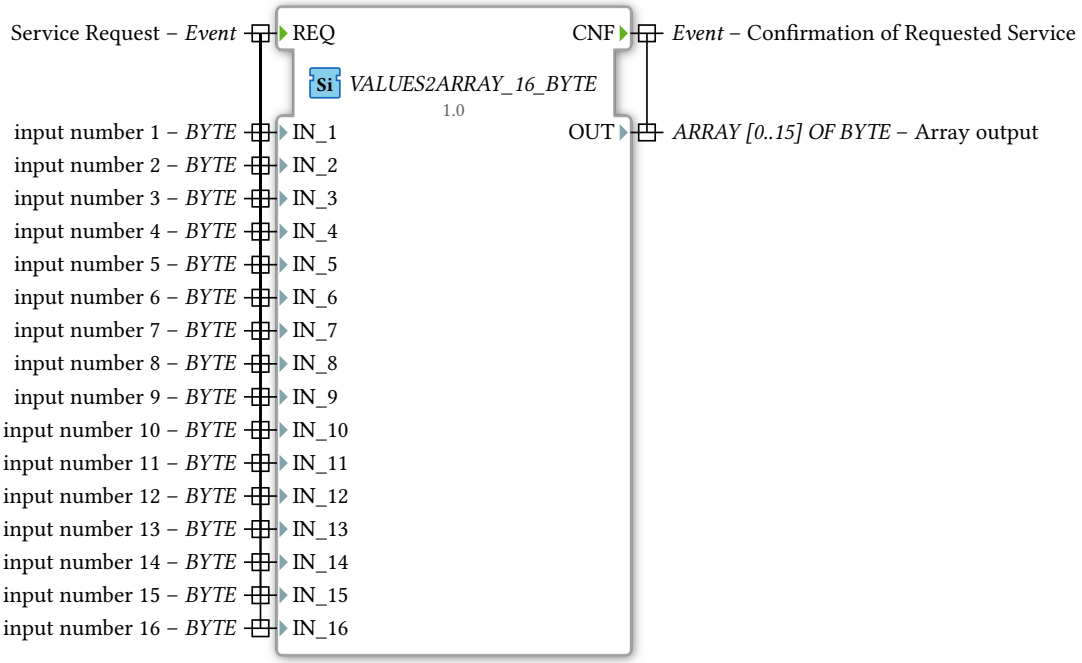
Comparison with similar function blocks

- **ReadJSON**: the reverse direction — extracts a field from a JSON string.
- **`SET_STRUCT_VALUE`**: sets a field in an IEC 61131-3 STRUCT instead of a JSON string.

Conclusion

`ToJSON` enables the step-by-step, chainable construction of JSON strings from individual field/value pairs and is suitable for building JSON payloads in 4diac applications without requiring a full structure type definition.

2.11.1.21 VALUES2ARRAY_16_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

VALUES2ARRAY_16_BYTE combines 16 individual BYTE scalar variables IN_1...IN_16 into a BYTE array of size 16. It is the reverse of ARRAY2VALUES_16_BYTE and belongs to the generic GEN_ARRAY2ARRAY family (cf. VALUES2ARRAY_2_LREAL).

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries IN_1, IN_2, IN_3, IN_4, IN_5, IN_6, IN_7, IN_8, IN_9, IN_10, IN_11, IN_12, IN_13, IN_14, IN_15, IN_16.

Event Outputs

- **CNF**: Confirms completion, carries OUT.

Data Inputs

- IN_1, IN_2, IN_3, IN_4, IN_5, IN_6, IN_7, IN_8, IN_9, IN_10, IN_11, IN_12, IN_13, IN_14, IN_15, IN_16 (BYTE): The 16 individual values (8-bit bit pattern) to be combined into the array.

Data Outputs

- **OUT** (BYTE, array size 16): OUT[i-1] corresponds to IN_i.

Functionality

On REQ, every input value IN_i is written to the corresponding element OUT[i-1] (IN_1 ⟶ OUT[0], ..., IN_16 ⟶ OUT[15]), then CNF is triggered.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY', the same C++ base as VALUES2ARRAY_2_LREAL and the other VALUES2ARRAY_* variants.
- **Fixed size 16**: For other array sizes of the same type see VALUES2ARRAY_2_BYTE, VALUES2ARRAY_4_BYTE, VALUES2ARRAY_8_BYTE, VALUES2ARRAY_32_BYTE.

State Overview

Stateless: every REQ immediately results in a full merge and CNF.

Application Scenarios

- **Building an array from individual values**: Several discrete BYTE variables need to be passed as an array to a downstream block that expects an array interface.
- **Interface adaptation** between variable-based and array-based block interfaces.

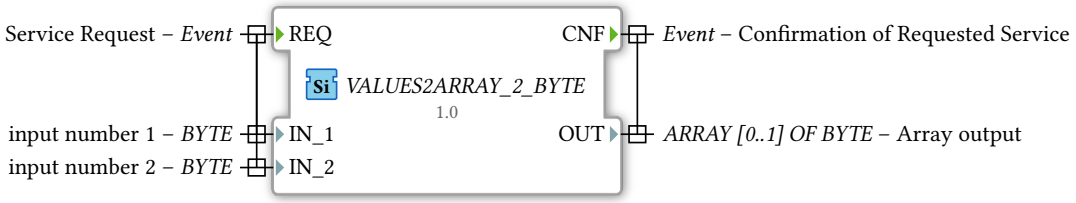
Comparison with similar function blocks

- **VALUES2ARRAY_2_LREAL**: the same generic implementation for data type LREAL.
- **ARRAY2VALUES_16_BYTE**: the reverse direction — splits an array into 16 individual values.

Conclusion

VALUES2ARRAY_16_BYTE provides a simple, generically implemented merge of 16 BYTE individual values into an array and is suitable for adapting variable-based to array-based interfaces.

2.11.1.22 VALUES2ARRAY_2_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

VALUES2ARRAY_2_BYTE combines 2 individual BYTE scalar variables IN_1...IN_2 into a BYTE array of size 2. It is the reverse of ARRAY2VALUES_2_BYTE and belongs to the generic GEN_ARRAY2ARRAY family (cf. VALUES2ARRAY_2_LREAL).

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries IN_1, IN_2.

Event Outputs

- **CNF**: Confirms completion, carries OUT.

Data Inputs

- IN_1, IN_2 (BYTE): The 2 individual values (8-bit bit pattern) to be combined into the array.

Data Outputs

- **OUT** (BYTE, array size 2): OUT[i-1] corresponds to IN_i.

Functionality

On REQ, every input value IN_i is written to the corresponding element OUT[i-1] (IN_1 ⟶ OUT[0], ..., IN_2 ⟶ OUT[1]), then CNF is triggered.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY', the same C++ base as VALUES2ARRAY_2_LREAL and the other VALUES2ARRAY_* variants.
- **Fixed size 2**: For other array sizes of the same type see VALUES2ARRAY_4_BYTE, VALUES2ARRAY_8_BYTE, VALUES2ARRAY_16_BYTE, VALUES2ARRAY_32_BYTE.

State Overview

Stateless: every REQ immediately results in a full merge and CNF.

Application Scenarios

- **Building an array from individual values**: Several discrete BYTE variables need to be passed as an array to a downstream block that expects an array interface.
- **Interface adaptation** between variable-based and array-based block interfaces.

Comparison with similar function blocks

- **VALUES2ARRAY_2_LREAL**: the same generic implementation for data type LREAL.
- **ARRAY2VALUES_2_BYTE**: the reverse direction — splits an array into 2 individual values.

Conclusion

VALUES2ARRAY_2_BYTE provides a simple, generically implemented merge of 2 BYTE individual values into an array and is suitable for adapting variable-based to array-based interfaces.

2.11.1.23 VALUES2ARRAY_2_LREAL

• – ◦ • – ◦ • – ◦ •

The function block VALUES2ARRAY_2_LREAL is used to convert two individual floating-point numbers of type LREAL into an array of the same type. It is designed as a service interface function block, providing a simple interface for this conversion task.



- **REQ** (Type: Event)
- Comment: Service Request
- Associated Data: IN_1, IN_2
- Description: This event triggers the conversion of the two input values IN_1 and IN_2 into the array OUT.
- **CNF** (Type: Event)
- Comment: Confirmation of Requested Service
- Associated Data: OUT
- Description: This event is triggered after the conversion is successful and the output array OUT is updated.
- **IN_1** (Type: LREAL)
- Comment: Input number 1
- Description: The first floating-point number to be added to the array.
- **IN_2** (Type: LREAL)
- Comment: Input number 2
- Description: The second floating-point number to be added to the array.
- **OUT** (Type: LREAL, Size: 2)
- Comment: Array output
- Description: An array of type LREAL containing the values of IN_1 and IN_2. IN_1 occupies the first index, and IN_2 the second index of the array.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not have any adapter interfaces.

Functionality

When a REQ event is received at the input of the function block, VALUES2ARRAY_2_LREAL reads the current values of the data inputs IN_1 and IN_2. These two values are then transferred to the output array OUT, with IN_1 being assigned to element OUT[0] and IN_2 to element OUT[1]. After successful completion of this operation, a CNF event is issued to signal the completion of the request.

- The function block is specialized for converting exactly two LREAL values.
- The output array OUT is fixed at size 2.
- Processing is event-driven, allowing for explicit requests to initiate the conversion process.

The VALUES2ARRAY_2_LREAL function block is stateless. Each REQ request is processed independently, without maintaining internal state across requests. The function depends solely on the input values at the time of the REQ event.

- **Data Bundling**: When two separate LREAL values need to be passed as a single data packet (array) to other function blocks that expect arrays as input.
- **Preprocessing for Mathematical Operations**: Preparing data pairs for array-based operations, such as vector addition or scaling.

This function block is specifically for converting two LREAL values into an LREAL array. General "value-to-array" blocks might be more flexible regarding the number and data type of inputs, but this block provides a simple, specialized solution for the specified use case. For a larger number of values or different data types, corresponding, specially adapted blocks would be required.

⚖️ Vergleich mit ähnlichen Bausteinen

Application Scenarios

State Overview

Technical Features

Metadata

Attribute	Value
Copyright	(c) 2014 Profactor GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner), 1.0 (2014-07-09, Matthias Plasch)
4diac package	eclipse4diac::convert

The `VALUES2ARRAY_2_LREAL` function block is a useful and efficient block for the specific task of converting two LREAL values into a fixed-dimensional LREAL array. Its simplicity and event-driven operation make it a natural component in scenarios where such data aggregation is required, and contribute to the modularity and readability of the application.

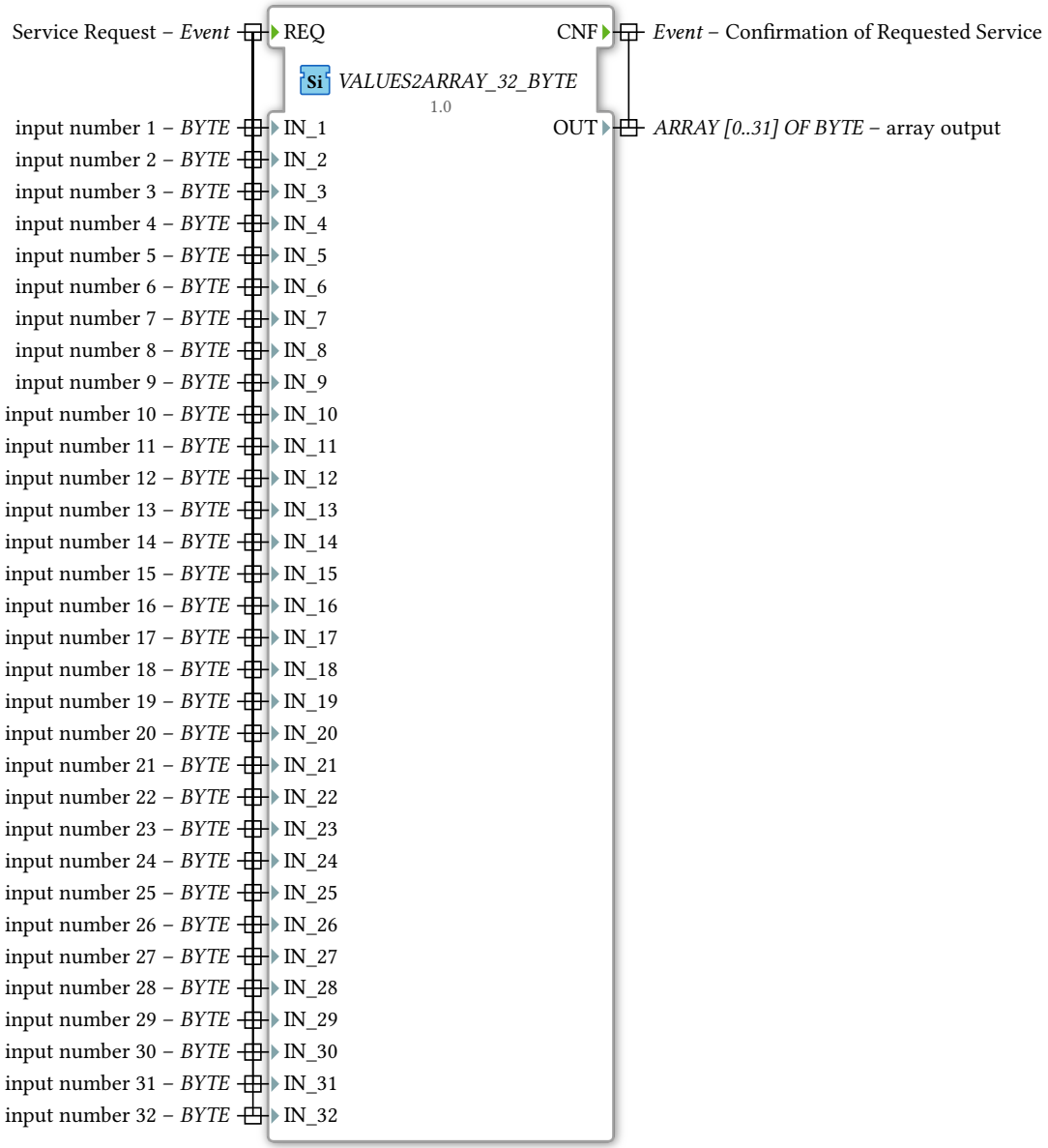
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Conclusion

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

2.11.1.24 VALUES2ARRAY_32_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`VALUES2ARRAY_32_BYTE` combines 32 individual `BYTE` scalar variables `IN_1`…`IN_32` into a `BYTE` array of size 32. It is the reverse of `ARRAY2VALUES_32_BYTE` and belongs to the generic `GEN_ARRAY2ARRAY` family (cf. VALUES2ARRAY_2_LREAL).

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries `IN_1`, `IN_2`, `IN_3`, `IN_4`, `IN_5`, `IN_6`, `IN_7`, `IN_8`, `IN_9`, `IN_10`, `IN_11`, `IN_12`, `IN_13`, `IN_14`, `IN_15`, `IN_16`, `IN_17`, `IN_18`, `IN_19`, `IN_20`, `IN_21`, `IN_22`, `IN_23`, `IN_24`, `IN_25`, `IN_26`, `IN_27`, `IN_28`, `IN_29`, `IN_30`, `IN_31`, `IN_32`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- `IN_1`, `IN_2`, `IN_3`, `IN_4`, `IN_5`, `IN_6`, `IN_7`, `IN_8`, `IN_9`, `IN_10`, `IN_11`, `IN_12`, `IN_13`, `IN_14`, `IN_15`, `IN_16`, `IN_17`, `IN_18`, `IN_19`, `IN_20`, `IN_21`, `IN_22`, `IN_23`, `IN_24`, `IN_25`, `IN_26`, `IN_27`, `IN_28`, `IN_29`, `IN_30`, `IN_31`, `IN_32` (`BYTE`): The 32 individual values (8-bit bit pattern) to be combined into the array.

Data Outputs

- **OUT** (`BYTE`, array size 32): `OUT[i-1]` corresponds to `IN_i`.

Functionality

On `REQ`, every input value `IN_i` is written to the corresponding element `OUT[i-1]` (`IN_1` ⟶ `OUT[0]`, ..., `IN_32` ⟶ `OUT[31]`), then `CNF` is triggered.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, the same C++ base as VALUES2ARRAY_2_LREAL and the other `VALUES2ARRAY_*` variants.
- **Fixed size 32**: For other array sizes of the same type see `VALUES2ARRAY_2_BYTE`, `VALUES2ARRAY_4_BYTE`, `VALUES2ARRAY_8_BYTE`, `VALUES2ARRAY_16_BYTE`.

State Overview

Stateless: every `REQ` immediately results in a full merge and `CNF`.

Application Scenarios

- **Building an array from individual values**: Several discrete `BYTE` variables need to be passed as an array to a downstream block that expects an array interface.
- **Interface adaptation** between variable-based and array-based block interfaces.

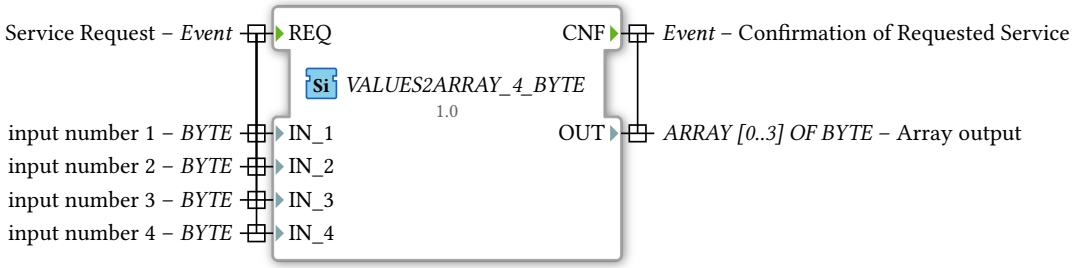
Comparison with similar function blocks

- **VALUES2ARRAY_2_LREAL**: the same generic implementation for data type `LREAL`.
- **`ARRAY2VALUES_32_BYTE`**: the reverse direction — splits an array into 32 individual values.

Conclusion

`VALUES2ARRAY_32_BYTE` provides a simple, generically implemented merge of 32 `BYTE` individual values into an array and is suitable for adapting variable-based to array-based interfaces.

2.11.1.25 VALUES2ARRAY_4_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`VALUES2ARRAY_4_BYTE` combines 4 individual `BYTE` scalar variables `IN_1`...`IN_4` into a `BYTE` array of size 4. It is the reverse of `ARRAY2VALUES_4_BYTE` and belongs to the generic `GEN_ARRAY2ARRAY` family (cf. VALUES2ARRAY_2_LREAL).

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries `IN_1`, `IN_2`, `IN_3`, `IN_4`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- `IN_1`, `IN_2`, `IN_3`, `IN_4` (`BYTE`): The 4 individual values (8-bit bit pattern) to be combined into the array.

Data Outputs

- **OUT** (`BYTE`, array size 4): `OUT[i-1]` corresponds to `IN_i`.

Functionality

On `REQ`, every input value `IN_i` is written to the corresponding element `OUT[i-1]` (`IN_1` ⟶ `OUT[0]`, ..., `IN_4` ⟶ `OUT[3]`), then `CNF` is triggered.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, the same C++ base as VALUES2ARRAY_2_LREAL and the other `VALUES2ARRAY_*` variants.
- **Fixed size 4**: For other array sizes of the same type see `VALUES2ARRAY_2_BYTE`, `VALUES2ARRAY_8_BYTE`, `VALUES2ARRAY_16_BYTE`, `VALUES2ARRAY_32_BYTE`.

State Overview

Stateless: every `REQ` immediately results in a full merge and `CNF`.

Application Scenarios

- **Building an array from individual values**: Several discrete `BYTE` variables need to be passed as an array to a downstream block that expects an array interface.
- **Interface adaptation** between variable-based and array-based block interfaces.

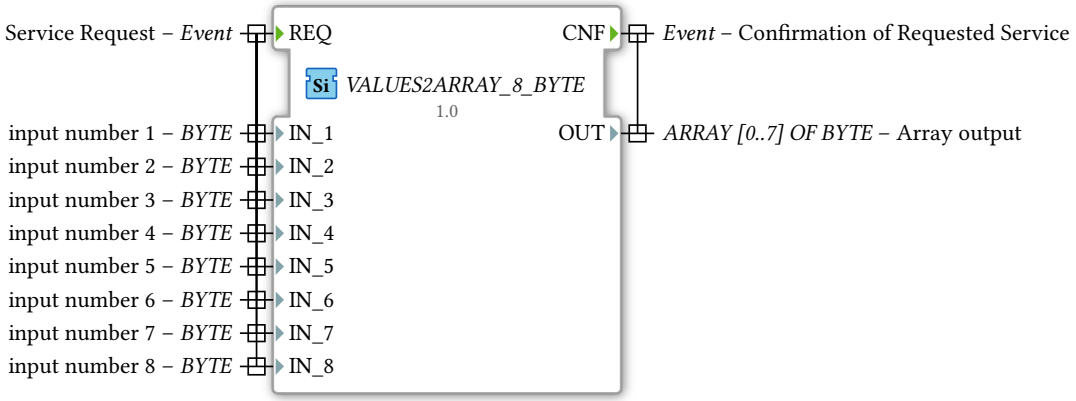
Comparison with similar function blocks

- VALUES2ARRAY_2_LREAL: the same generic implementation for data type `LREAL`.
- **`ARRAY2VALUES_4_BYTE`**: the reverse direction — splits an array into 4 individual values.

Conclusion

`VALUES2ARRAY_4_BYTE` provides a simple, generically implemented merge of 4 `BYTE` individual values into an array and is suitable for adapting variable-based to array-based interfaces.

2.11.1.26 VALUES2ARRAY_8_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

VALUES2ARRAY_8_BYTE combines 8 individual BYTE scalar variables IN_1...IN_8 into a BYTE array of size 8. It is the reverse of ARRAY2VALUES_8_BYTE and belongs to the generic GEN_ARRAY2ARRAY family (cf. VALUES2ARRAY_2_LREAL).

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries IN_1, IN_2, IN_3, IN_4, IN_5, IN_6, IN_7, IN_8.

Event Outputs

- **CNF**: Confirms completion, carries OUT.

Data Inputs

- IN_1, IN_2, IN_3, IN_4, IN_5, IN_6, IN_7, IN_8 (BYTE): The 8 individual values (8-bit bit pattern) to be combined into the array.

Data Outputs

- **OUT** (BYTE, array size 8): OUT[i-1] corresponds to IN_i.

Functionality

On REQ, every input value IN_i is written to the corresponding element OUT[i-1] (IN_1 ⟶ OUT[0], ..., IN_8 ⟶ OUT[7]), then CNF is triggered.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY', the same C++ base as VALUES2ARRAY_2_LREAL and the other VALUES2ARRAY_* variants.
- **Fixed size 8**: For other array sizes of the same type see VALUES2ARRAY_2_BYTE, VALUES2ARRAY_4_BYTE, VALUES2ARRAY_16_BYTE, VALUES2ARRAY_32_BYTE.

State Overview

Stateless: every REQ immediately results in a full merge and CNF.

Application Scenarios

- **Building an array from individual values**: Several discrete BYTE variables need to be passed as an array to a downstream block that expects an array interface.
- **Interface adaptation** between variable-based and array-based block interfaces.

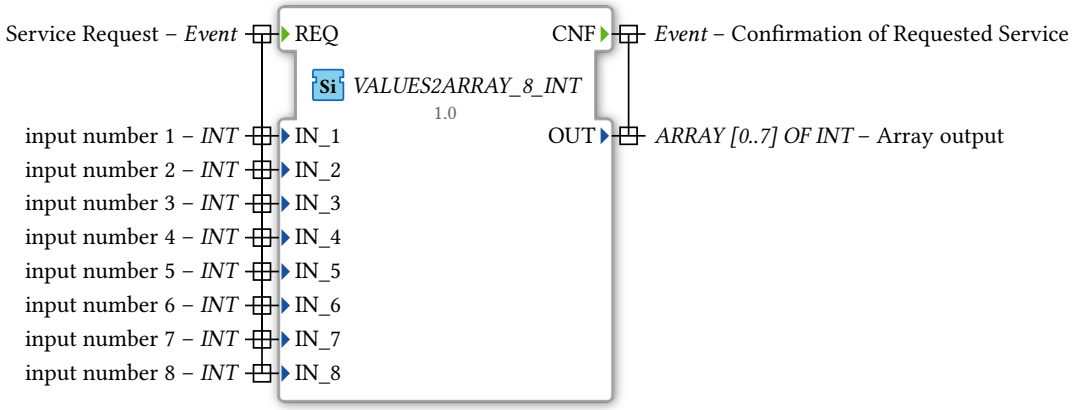
Comparison with similar function blocks

- **VALUES2ARRAY_2_LREAL**: the same generic implementation for data type LREAL.
- **ARRAY2VALUES_8_BYTE**: the reverse direction — splits an array into 8 individual values.

Conclusion

VALUES2ARRAY_8_BYTE provides a simple, generically implemented merge of 8 BYTE individual values into an array and is suitable for adapting variable-based to array-based interfaces.

2.11.1.27 VALUES2ARRAY_8_INT



• – ◦ • – ◦ • – ◦ •

Introduction

`VALUES2ARRAY_8_INT` combines 8 individual `INT` scalar variables `IN_1`...`IN_8` into a `INT` array of size 8. It is the reverse of `ARRAY2VALUES_8_INT` and belongs to the generic `GEN_ARRAY2ARRAY` family (cf. VALUES2ARRAY_2_LREAL).

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries `IN_1`, `IN_2`, `IN_3`, `IN_4`, `IN_5`, `IN_6`, `IN_7`, `IN_8`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- `IN_1`, `IN_2`, `IN_3`, `IN_4`, `IN_5`, `IN_6`, `IN_7`, `IN_8` (`INT`): The 8 individual values (16-bit signed integer) to be combined into the array.

Data Outputs

- **OUT** (`INT`, array size 8): `OUT[i-1]` corresponds to `IN_i`.

Functionality

On `REQ`, every input value `IN_i` is written to the corresponding element `OUT[i-1]` (`IN_1` ⟶ `OUT[0]`, ..., `IN_8` ⟶ `OUT[7]`), then `CNF` is triggered.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_ARRAY2ARRAY'`, the same C++ base as VALUES2ARRAY_2_LREAL and the other `VALUES2ARRAY_*` variants.
- **Fixed size 8**: For other array sizes of the same type see .

State Overview

Stateless: every `REQ` immediately results in a full merge and `CNF`.

Application Scenarios

- **Building an array from individual values**: Several discrete `INT` variables need to be passed as an array to a downstream block that expects an array interface.
- **Interface adaptation** between variable-based and array-based block interfaces.

Comparison with similar function blocks

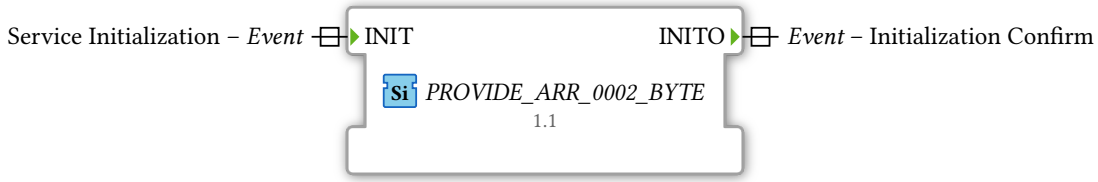
- VALUES2ARRAY_2_LREAL: the same generic implementation for data type `LREAL`.
- **`ARRAY2VALUES_8_INT`**: the reverse direction — splits an array into 8 individual values.

Conclusion

`VALUES2ARRAY_8_INT` provides a simple, generically implemented merge of 8 `INT` individual values into an array and is suitable for adapting variable-based to array-based interfaces.

2.11.1.28 Providers

PROVIDE_ARR_0002_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0002_BYTE` provides a fixed-size `BYTE` array container of size 2 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 2, index 0…1): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 2**: Other sizes of the same or different data types: `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`, `PROVIDE_ARR_0016_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

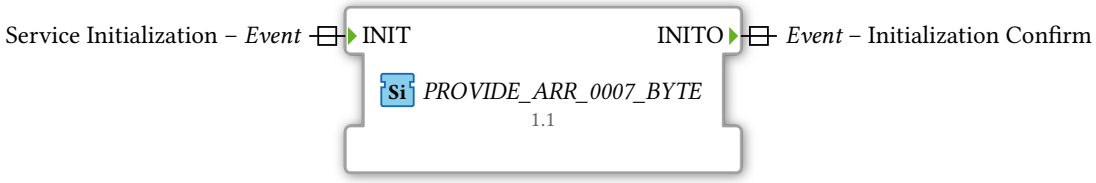
Comparison with similar function blocks

- `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`, `PROVIDE_ARR_0016_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0002_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 2 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0007_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0007_BYTE` provides a fixed-size `BYTE` array container of size 7 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 7, index 0…6): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 7**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`, `PROVIDE_ARR_0016_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.

- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

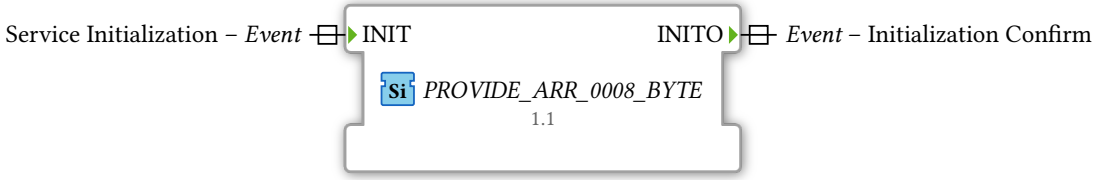
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`, `PROVIDE_ARR_0016_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0007_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 7 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0008_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0008_BYTE` provides a fixed-size `BYTE` array container of size 8 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 8, index 0…7): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 8**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0014_BYTE`, `PROVIDE_ARR_0016_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

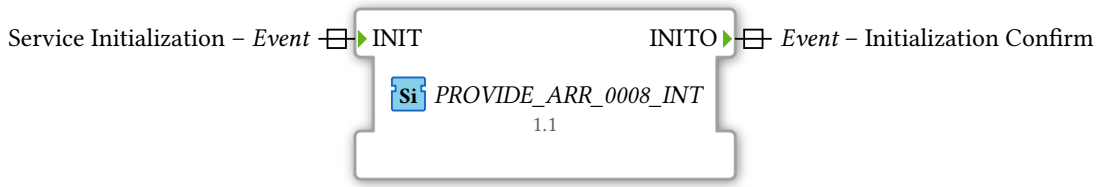
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0014_BYTE`, `PROVIDE_ARR_0016_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_8_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0008_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 8 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0008_INT



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0008_INT` provides a fixed-size `INT` array container of size 8 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`INT`, array size 8, index 0…7): The array container (16-bit signed integer per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 8**: Other sizes of the same or different data types: none.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `INT` array for test setups without manually re-wiring the array in every application.

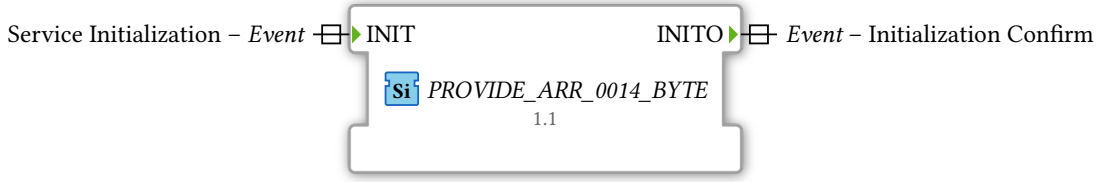
Comparison with similar function blocks

- none: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_8_INT`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0008_INT` provides a simple, design-time pre-fillable `INT` array container of fixed size 8 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0014_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0014_BYTE` provides a fixed-size `BYTE` array container of size 14 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 14, index 0…13): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 14**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0016_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

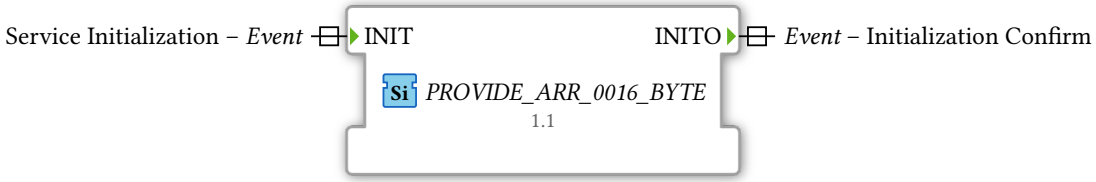
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0016_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0014_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 14 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0016_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0016_BYTE` provides a fixed-size `BYTE` array container of size 16 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 16, index 0…15): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.

- **Fixed size 16**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

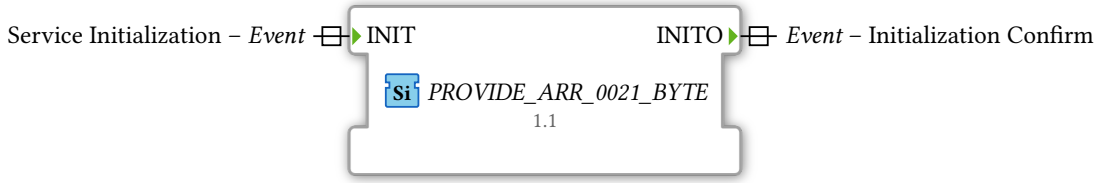
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0016_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 16 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0021_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0021_BYTE` provides a fixed-size `BYTE` array container of size 21 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 21, index 0…20): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.

- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 21**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0021_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 21 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0028_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0028_BYTE` provides a fixed-size `BYTE` array container of size 28 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 28, index 0…27): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 28**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0028_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 28 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0032_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0032_BYTE` provides a fixed-size `BYTE` array container of size 32 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 32, index 0...31): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs

no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 32**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0032_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 32 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0035_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0035_BYTE` provides a fixed-size `BYTE` array container of size 35 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 35, index 0…34): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 35**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

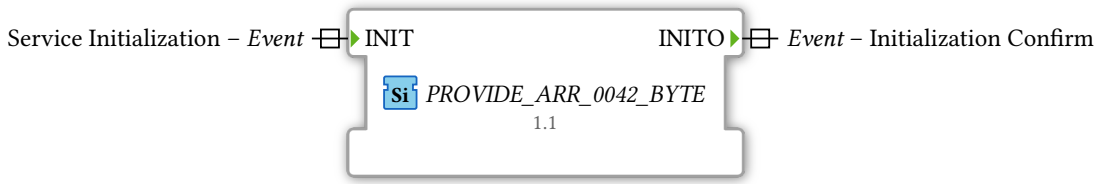
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0035_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 35 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0042_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0042_BYTE` provides a fixed-size `BYTE` array container of size 42 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (BYTE, array size 42, index 0...41): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for BYTE).

Functionality

On INIT, the block reads the InOut array D1 (which may already be pre-filled with start values at design time) and immediately confirms this via INITO, forwarding D1 unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other PROVIDE_ARR_* variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: D1 is a single InOut variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 42**: Other sizes of the same or different data types: PROVIDE_ARR_0002_BYTE, PROVIDE_ARR_0007_BYTE, PROVIDE_ARR_0008_BYTE, PROVIDE_ARR_0014_BYTE.

State Overview

Stateless: INIT reads D1 and confirms immediately via INITO, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined BYTE array for test setups without manually re-wiring the array in every application.

Comparison with similar function blocks

- PROVIDE_ARR_0002_BYTE, PROVIDE_ARR_0007_BYTE, PROVIDE_ARR_0008_BYTE, PROVIDE_ARR_0014_BYTE: the same generic implementation with a different array size or data type.
- **ARRAY2ARRAY_2_BYTE**: copies an already existing array instead of providing/pre-filling one.

Conclusion

PROVIDE_ARR_0042_BYTE provides a simple, design-time pre-fillable BYTE array container of fixed size 42 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0056_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

PROVIDE_ARR_0056_BYTE provides a fixed-size BYTE array container of size 56 as an InOut variable D1. It belongs to the generic GEN_INIT_ARR family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries D1.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 56, index 0…55): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 56**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

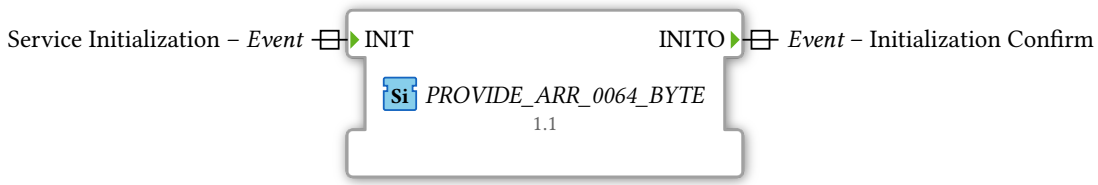
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0056_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 56 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0064_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0064_BYTE` provides a fixed-size `BYTE` array container of size 64 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 64, index 0…63): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 64**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

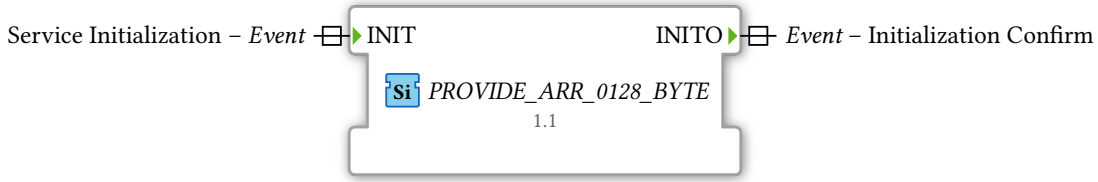
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0064_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 64 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0128_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0128_BYTE` provides a fixed-size `BYTE` array container of size 128 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ

in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 128, index 0…127): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 128**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

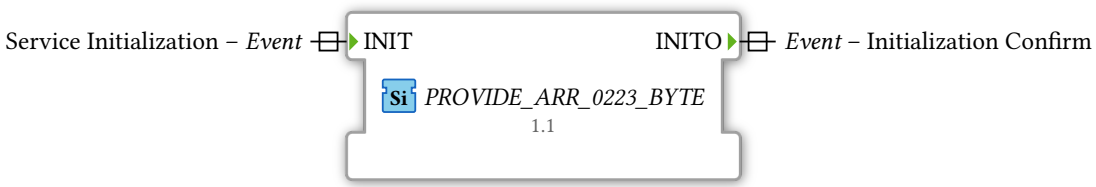
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0128_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 128 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0223_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0223_BYTE` provides a fixed-size `BYTE` array container of size 223 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 223, index 0…222): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 223**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

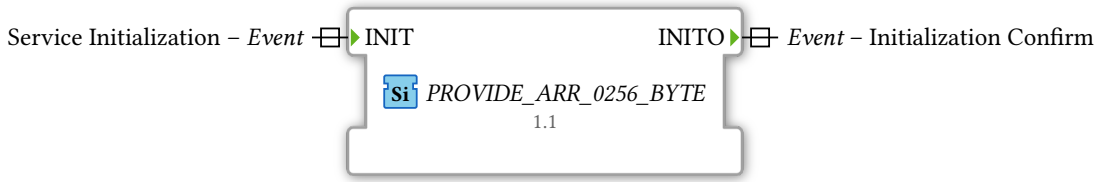
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_0223_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 223 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0256_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

PROVIDE_ARR_0256_BYTE provides a fixed-size BYTE array container of size 256 as an InOut variable D1. It belongs to the generic GEN_INIT_ARR family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries D1.

Event Outputs

- **INITO**: Confirms the initialization, carries D1.

In/Out Variables

- **D1** (BYTE, array size 256, index 0…255): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for BYTE).

Functionality

On INIT, the block reads the InOut array D1 (which may already be pre-filled with start values at design time) and immediately confirms this via INITO, forwarding D1 unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other PROVIDE_ARR_* variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: D1 is a single InOut variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 256**: Other sizes of the same or different data types: PROVIDE_ARR_0002_BYTE, PROVIDE_ARR_0007_BYTE, PROVIDE_ARR_0008_BYTE, PROVIDE_ARR_0014_BYTE.

State Overview

Stateless: INIT reads D1 and confirms immediately via INITO, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined BYTE array for test setups without manually re-wiring the array in every application.

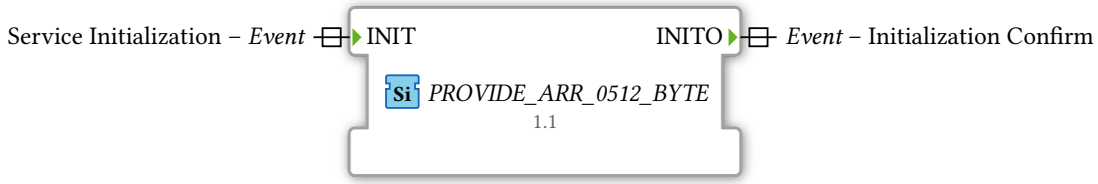
Comparison with similar function blocks

- PROVIDE_ARR_0002_BYTE, PROVIDE_ARR_0007_BYTE, PROVIDE_ARR_0008_BYTE, PROVIDE_ARR_0014_BYTE: the same generic implementation with a different array size or data type.
- **ARRAY2ARRAY_2_BYTE**: copies an already existing array instead of providing/pre-filling one.

Conclusion

PROVIDE_ARR_0256_BYTE provides a simple, design-time pre-fillable BYTE array container of fixed size 256 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_0512_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_0512_BYTE` provides a fixed-size `BYTE` array container of size 512 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 512, index 0…511): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 512**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

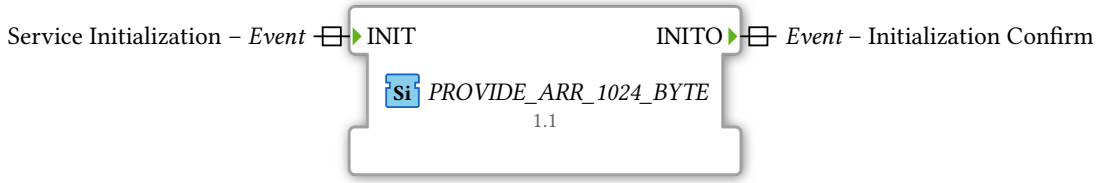
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

PROVIDE_ARR_0512_BYTE provides a simple, design-time pre-fillable BYTE array container of fixed size 512 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_1024_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

PROVIDE_ARR_1024_BYTE provides a fixed-size BYTE array container of size 1024 as an InOut variable D1. It belongs to the generic GEN_INIT_ARR family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries D1.

Event Outputs

- **INITO**: Confirms the initialization, carries D1.

In/Out Variables

- **D1** (BYTE, array size 1024, index 0…1023): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. [16#01, 16#00, ...] for BYTE).

Functionality

On INIT, the block reads the InOut array D1 (which may already be pre-filled with start values at design time) and immediately confirms this via INITO, forwarding D1 unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR', the same C++ base as all other PROVIDE_ARR_* variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: D1 is a single InOut variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 1024**: Other sizes of the same or different data types: PROVIDE_ARR_0002_BYTE, PROVIDE_ARR_0007_BYTE, PROVIDE_ARR_0008_BYTE, PROVIDE_ARR_0014_BYTE.

State Overview

Stateless: INIT reads D1 and confirms immediately via INITO, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined BYTE array for test setups without manually re-wiring the array in every application.

Comparison with similar function blocks

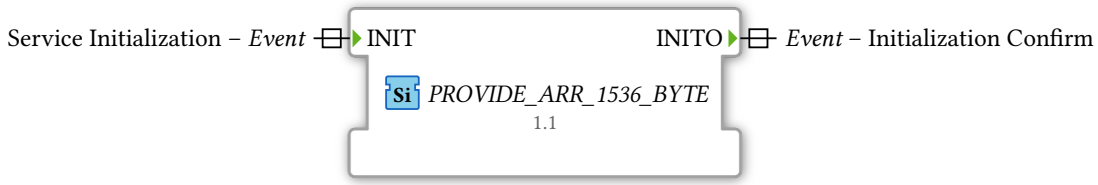
- PROVIDE_ARR_0002_BYTE, PROVIDE_ARR_0007_BYTE, PROVIDE_ARR_0008_BYTE, PROVIDE_ARR_0014_BYTE: the same generic implementation with a different array size or data type.

- **ARRAY2ARRAY_2_BYTE**: copies an already existing array instead of providing/pre-filling one.

Conclusion

PROVIDE_ARR_1024_BYTE provides a simple, design-time pre-fillable BYTE array container of fixed size 1024 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_1536_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

PROVIDE_ARR_1536_BYTE provides a fixed-size BYTE array container of size 1536 as an InOut variable D1. It belongs to the generic GEN_INIT_ARR family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries D1.

Event Outputs

- **INITO**: Confirms the initialization, carries D1.

In/Out Variables

- **D1** (BYTE, array size 1536, index 0…1535): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for BYTE).

Functionality

On INIT, the block reads the InOut array D1 (which may already be pre-filled with start values at design time) and immediately confirms this via INITO, forwarding D1 unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other PROVIDE_ARR_* variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: D1 is a single InOut variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 1536**: Other sizes of the same or different data types: PROVIDE_ARR_0002_BYTE, PROVIDE_ARR_0007_BYTE, PROVIDE_ARR_0008_BYTE, PROVIDE_ARR_0014_BYTE.

State Overview

Stateless: INIT reads D1 and confirms immediately via INITO, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.
- **Test data generation**: Providing a predefined BYTE array for test setups without manually re-wiring the array in every application.

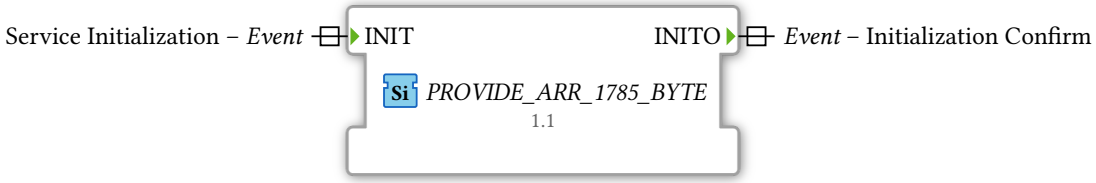
Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_1536_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 1536 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

PROVIDE_ARR_1785_BYTE



• – ◦ • – ◦ • – ◦ •

Introduction

`PROVIDE_ARR_1785_BYTE` provides a fixed-size `BYTE` array container of size 1785 as an `InOut` variable `D1`. It belongs to the generic `GEN_INIT_ARR` family: a series of identically structured blocks that only differ in array size and data type, used to make an array that can be pre-filled with start values at design time available inside a 4diac application.

Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `D1`.

Event Outputs

- **INITO**: Confirms the initialization, carries `D1`.

In/Out Variables

- **D1** (`BYTE`, array size 1785, index 0…1784): The array container (8-bit bit pattern per element). Can be pre-filled with start values in the FB instance editor (e.g. `[16#01, 16#00, ...]` for `BYTE`).

Functionality

On `INIT`, the block reads the `InOut` array `D1` (which may already be pre-filled with start values at design time) and immediately confirms this via `INITO`, forwarding `D1` unchanged. The block performs no computation of its own — it acts as a declarative container whose content is set via the FB instance parameterization.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_INIT_ARR'`, the same C++ base as all other `PROVIDE_ARR_*` variants; array size and data type are fixed per instantiation via the type definition.
- **InOut instead of separate input/output**: `D1` is a single `InOut` variable — pre-filling and provisioning happen through the same variable, there is no separate copy.
- **Fixed size 1785**: Other sizes of the same or different data types: `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`.

State Overview

Stateless: `INIT` reads `D1` and confirms immediately via `INITO`, without modifying the content.

Application Scenarios

- **Configuration arrays**: Providing fixed, pre-filled tables (e.g. calibration values, lookup tables, protocol header bytes) as a reusable block that can be parameterized at design time.

- **Test data generation**: Providing a predefined `BYTE` array for test setups without manually re-wiring the array in every application.

Comparison with similar function blocks

- `PROVIDE_ARR_0002_BYTE`, `PROVIDE_ARR_0007_BYTE`, `PROVIDE_ARR_0008_BYTE`, `PROVIDE_ARR_0014_BYTE`: the same generic implementation with a different array size or data type.
- **`ARRAY2ARRAY_2_BYTE`**: copies an already existing array instead of providing/pre-filling one.

Conclusion

`PROVIDE_ARR_1785_BYTE` provides a simple, design-time pre-fillable `BYTE` array container of fixed size 1785 and is suitable for configuration and test-data arrays intended for reuse across 4diac applications.

2.11.2 core

2.11.2.1 🎧 Podcast

- ESP32-S3 in Detail: Dual-Core, 32MB Power and CAN Bus for Agricultural and Construction Machinery Mechatronics
- ESP32-S3 in Detail: Dual-Core, AI Turbo and the Swiss Army Knife of IoT

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

2.11.2.2 GenericClassName

• – ◦ • – ◦ • – ◦ •

Introduction

The attribute declaration `GenericClassName` defines the class name for generic types in 4diac.

Technical Details

- **Base Data Type:** `STRING`
- **Initial Value:** `'GEN_'`
- **Usage:** Applied at the `FBType` level to identify generic implementations.

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::core

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.2.3 TypeHash

• – ◦ • – ◦ • – ◦ •

Introduction

The attribute declaration `TypeHash` defines a fixed type hash for function blocks.

Technical Details

- **Base Data Type:** `STRING`
- **Initial Value:** `''` (empty)

- **Use:** Used to identify and validate type versions.

Metadata

Attribute	Value
License	EPL-2.0
Version	1.0 (2025-06-05, Martin Erich Jobst)
4diac Package	eclipse4diac::core

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

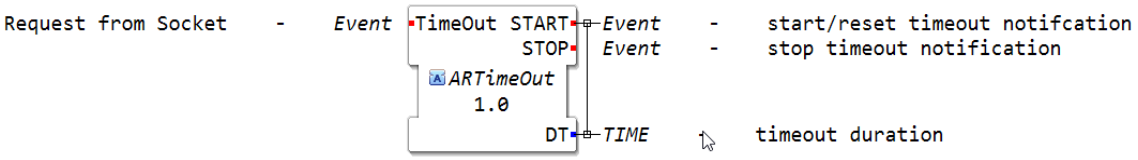
2.11.3 events

2.11.3.1 🎧 Podcast

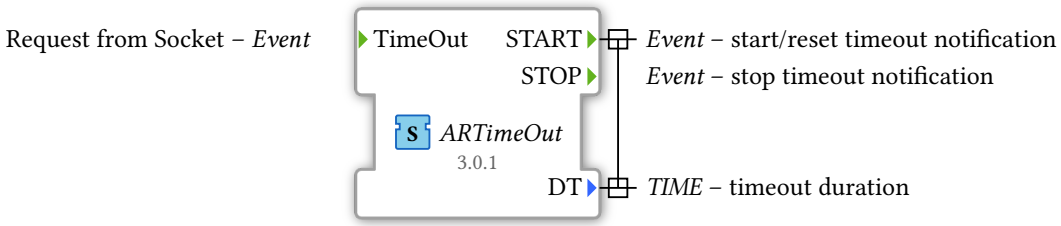
- Taster & Events: GPIO Basics for Agricultural and Construction Machinery Mechatronics Qzmsdocs000000 Qz

---

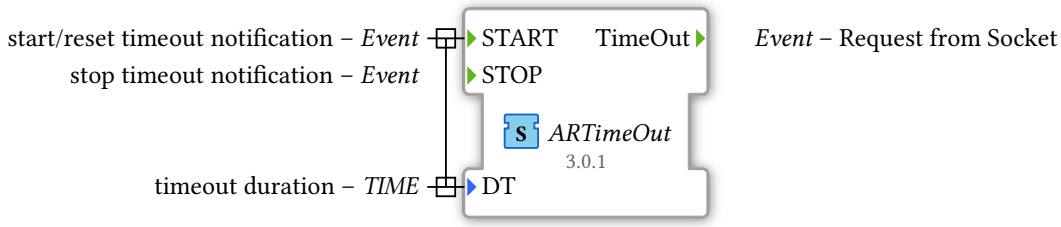
2.11.3.2 ARTimeOut



Plug (adapter output)



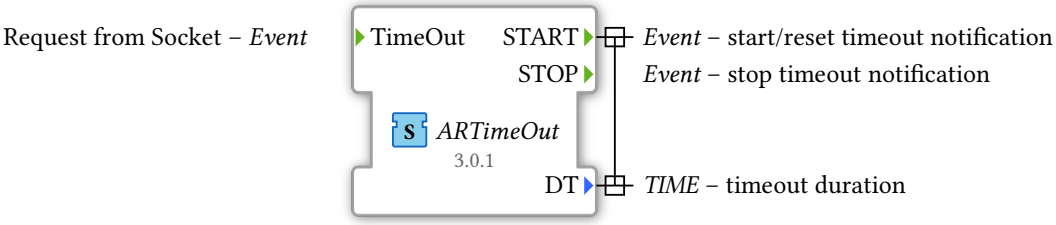
Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

IEC 61499 is an international standard for modeling distributed industrial control systems. Within this standard, adapter types are an important component for defining reusable interfaces between function blocks. The **ARTimeOut** adapter (Adapter for Resettable Timeout) is an example of such an adapter, providing an interface for a resettable timeout service.



Structure of the ARTimeOut Adapter

The ARTimeOut adapter is specified as **AdapterType** in IEC 61499 and defines an interface for communication between a socket and a plug.

Interface

The ARTimeOut adapter interface consists of:

- **Input Events**:
- **TimeOut**: This event is triggered by the SOCKET to indicate that the timeout has expired.
- **Output Events**:
- **START**: This event is triggered by the PLUG to start or reset the timeout. It is linked to the output variable **DT**, which defines the timeout duration.
- **STOP**: This event is triggered by the PLUG to stop the timeout.
- **Output Variable**:
- **DT (Duration Time)**: A time variable (TIME) that specifies the duration of the timeout.

Service Behavior

The ARTimeOut adapter offers two service sequences:

1. **Timeout**:

- The PLUG sends a **START** event with a duration of **DT** to start the timeout.
- When the timeout expires, the SOCKET sends a **TimeOut** event back to the PLUG.

1. **NormalOperation**:

- The PLUG sends a **START** event with a duration of **DT** to start the timeout.
- The PLUG can send a **STOP** event to end the timeout prematurely.

ARTimeOut Adapter Behavior

The ARTimeOut adapter enables the control of a timeout mechanism with the following functions:

1. **Start/Reset Timeout**:

- The **START** event starts or resets the timeout with the specified duration **DT**.

1. **Stop Timeout**:

- The **STOP** event terminates the timeout prematurely.

1. **Timeout Expiration**:

- When the timeout expires, the **TimeOut** event is triggered to notify the PLUG.

Application Examples

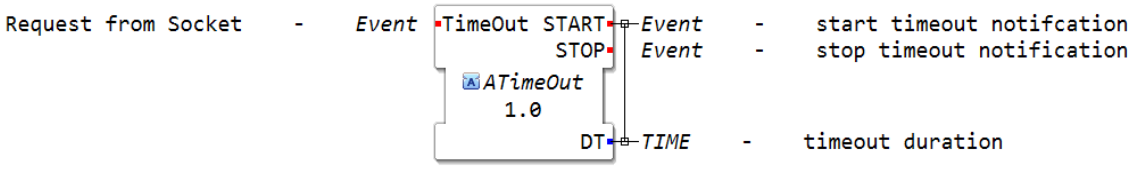
The ARTimeOut adapter can be used in various industrial control applications, particularly in scenarios requiring a reliable and resettable timeout mechanism. Examples include:

- **Network Communication**: Monitoring response timeouts in distributed systems.
- **Machine Control**: Timeouts for command execution to detect hung processes.
- **Safety Systems**: Monitoring sensors to detect malfunctions when no updates are received.

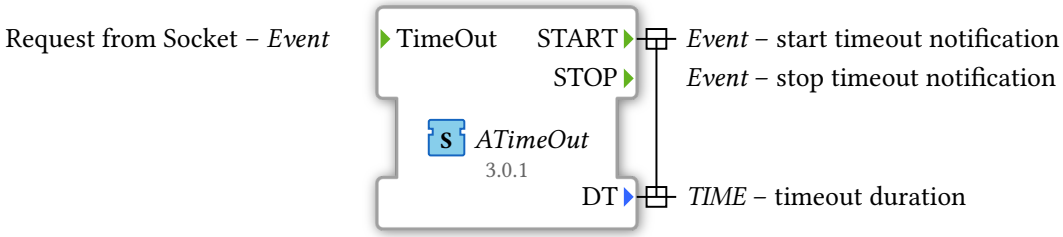
Conclusion

The ARTimeOut adapter is a useful tool within IEC 61499 to provide a standardized interface for resettable timeout services. Its clear separation between SOCKET and PLUG, along with defined service sequences, makes it ideal for integration into distributed control systems. The ability to start, stop, and notify when the timeout expires makes it a flexible solution for a wide range of industrial applications.

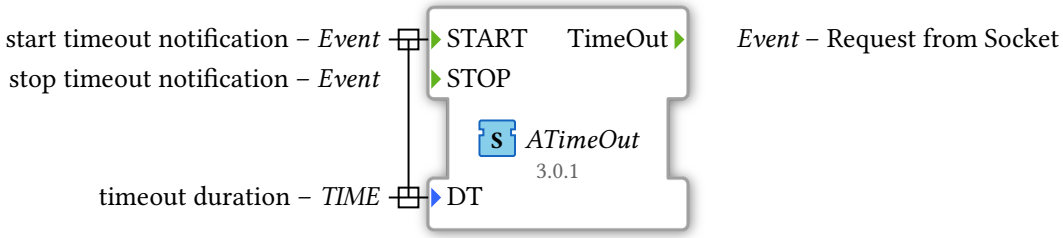
2.11.3.3 ATimeOut



Plug (adapter output)



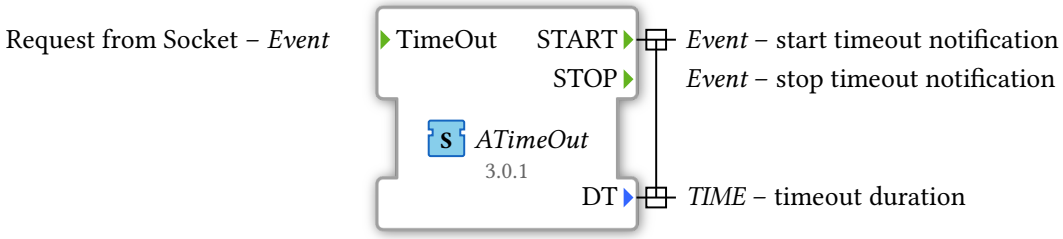
Socket (adapter input)



• – ◦ • – ◦ • – ◦ •

Introduction

The **ATimeOut** adapter is a standardized interface (AdapterType) according to IEC 61499 for implementing timeout services. It defines the communication between a service user (PLUG) and a time service provider (SOCKET). Unlike `ARTimeOut`, this adapter is designed for simple, non-retriggerable timeouts.



Structure of the ATimeOut Adapter

The ATimeOut adapter defines a clear separation of responsibilities between the control logic (Plug) and the timer (Socket).

Interface

The interface is defined from the perspective of the **plug**:

- **Input Events (received from the socket)**:
- **TimeOut**: Signals that the specified time has elapsed.
- **Output Events (sent to the socket)**:
- **START**: Initiates the timeout process. This event is associated with the variable **DT**.
- **STOP**: Terminates an ongoing timeout process.
- **Output Variables (sent to the socket)**:
- **DT (Duration Time)**: Defines the duration of the timeout (data type: TIME).

Behavior and Service Sequences

The adapter supports two basic processes:

1. **Timeout Expiration**:

- The plug sends `START` with a value for `DT`.
- The socket processes the time and sends back `TimeOut` after the time expires.

1. **Premature Termination**:

- The plug sends `START`.
- Before the time expires, the plug sends `STOP`. The socket terminates the timer; no `TimeOut` event occurs.

Technical Features

- **Non-retriggerable**: A repeated `START` during an ongoing timeout is ignored in the standard implementation (`E_TimeOut`).
- **Adapter Concept**: Enables clean encapsulation of the timing logic and simplifies the component network by reducing the number of connection lines.

Application Examples

- **Monitoring Response Times**: Waiting for an acknowledgment (e.g., from a communication partner) within a fixed time window.
- **Process Monitoring**: Ensuring that a mechanical process is completed within the expected time.

⚖️ Comparison with ARTimeOut

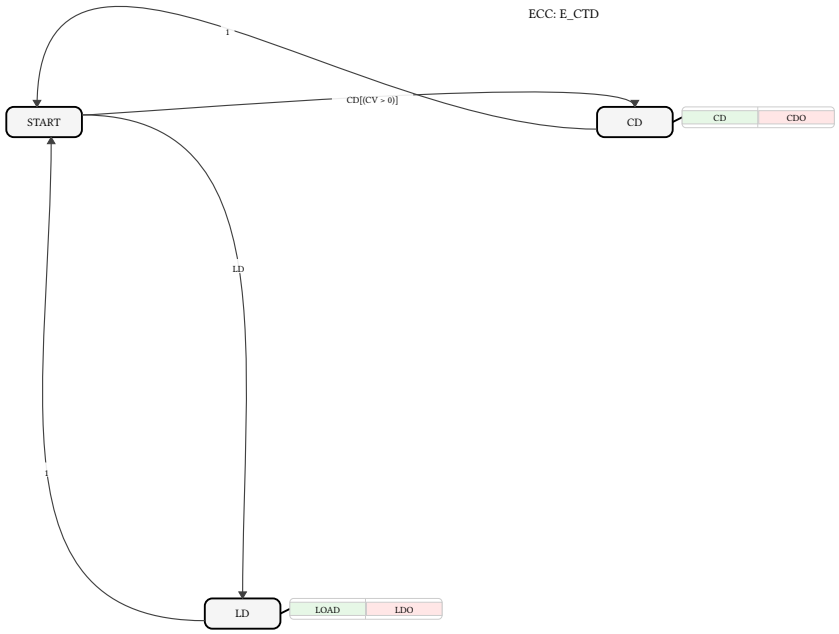
Feature	ATimeOut (this)	ARTimeOut
Type	Adapter	Adapter
Retriggerable	No	Yes
Event START	Start Timer	Start/Reset Timer

Conclusion

The ATimeOut adapter provides a robust and simple interface for time-critical monitoring tasks in distributed control systems. It forms the basis for the function block `E_TimeOut`.

2.11.3.4 E_CTD

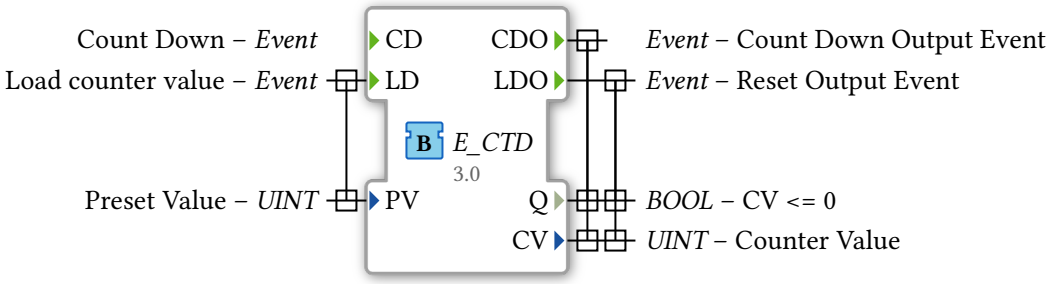
Podcast



- E_CTD: Event-Driven Down Counter according to IEC 61499

Introduction

The **E_CTD** (Event-Driven Down Counter) is an event-driven down counter according to the IEC 61499 standard. This function block is used in industrial control systems to implement counting operations that are triggered by events.



Structure of the E_CTD Function Block

Interface

Event Inputs:

- **CD (Count Down):** Triggers a counting step that decrements the counter value.
- **LD (Load):** Loads the initial value `PV` into the counter.

Event Outputs:

- **CDO (Count Down Output):** Confirms a counting step. Triggered after each `CD` event, as long as the counter value was greater than 0.
- **Related Data:** `Q`, `CV`
- **LDO (Load Output):** Confirms the successful loading of a new counter value.
- **Related Data**: `Q`, `CV`

Input Variables:

- **PV (Preset Value):** The initial value loaded during a load event (Data Type: `UINT`).

Output Variables:

- **Q (Status):** Output flag set when the counter reaches 0 (Data Type: `TRUE`).
- **CV (Counter Value):** The current counter value (Data Type: `UINT`).

Behavior of the E_CTD Block

1. **Initialization/Loading:**

- When an **LD** event occurs, the counter value `CV` is set to the value of **PV**.
- The output flag `Q` is updated based on the condition `CV = 0`.
- The **LDO** event is triggered and outputs the new counter value `CV` and the flag `Q`.

1. **Countdown:**

- With each **CD** event, the counter value **CV** is decremented by 1 if it is greater than 0.
- The output flag `Q` is then updated based on the new condition `CV = 0`.
- The **CDO** event is triggered and outputs the current counter value `CV` and the flag `Q`.

1. **Counter Reload:**

- A subsequent **LD** event resets **CV** back to **PV** at any time and triggers **LDO**.

Technical Features

- **Event-driven:** The function block operates exclusively based on events and does not require cyclic calls.
- **Flexible Initialization:** The initial value **PV** can be changed at any time by an **LD** event.

Application Examples

- **Production Lines:** Counting units produced.
- **Packaging Machines:** Control of filling processes.
- **Energy Management:** Monitoring of consumption cycles.

⚖️ Comparison with similar modules

Feature	E_CTD	E_CTU (Up Counter)	E_CTUD (Up/Down Counter)
Counting Direction	Down	Up	Both
Event-Driven	Yes	Yes	Yes
Reset Function	LD (Reload)	R (Reset)	R (Reset)

🛠️ Related Exercises

- Exercise_081

Conclusion

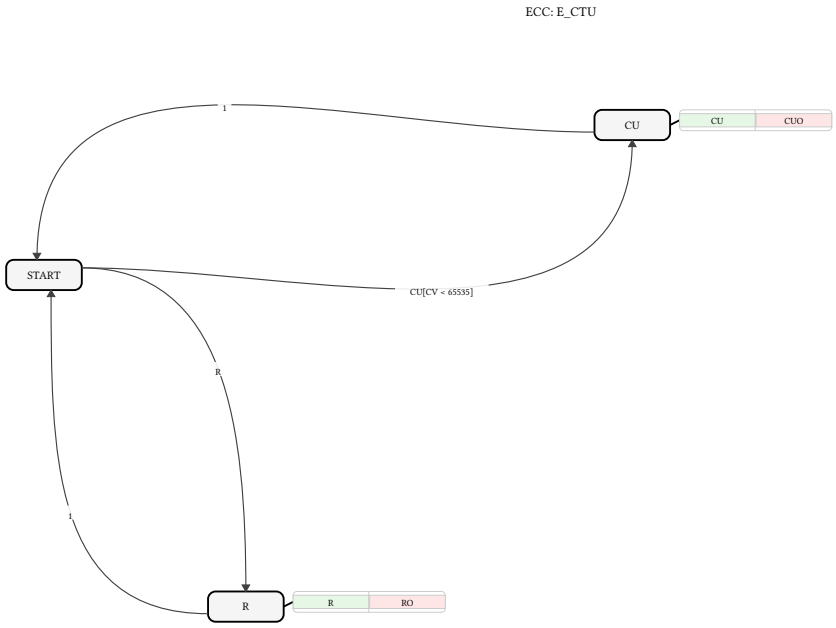
The **E_CTD** block is an essential element in IEC 61499, providing a reliable and flexible counting function for industrial controllers. Its event-driven nature makes it particularly suitable for distributed systems where cyclic polling is impractical. Its clear interface and intuitive behavior make it easy to integrate into existing control concepts.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 E_CTU Event Counter Block on ms-muc-docs.de

2.11.3.5 E_CTU

📺 Video

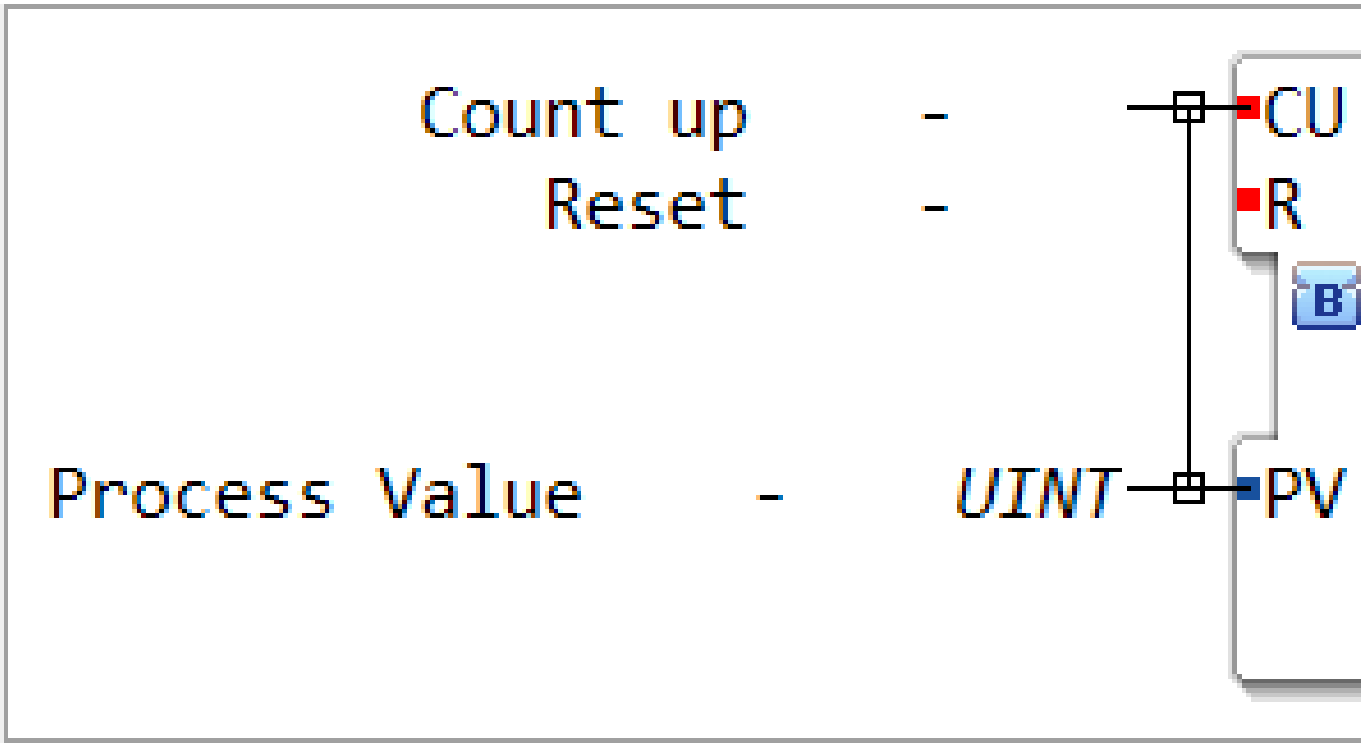


- The E_CTU upcounter

🎧 Podcast

- The E_CTU in IEC 61499: Event-driven counting and why the minimalist solution is convincing in mechanical engineering
- The E_CTU component: Event-driven incrementing in industry according to IEC 61499
- E_CTUD: Bidirectional counter in IEC 61499 systems
- Master knowledge 61499: The event-driven upcounter (E_CTU) – Robust counting in Agricultural Machinery Controls

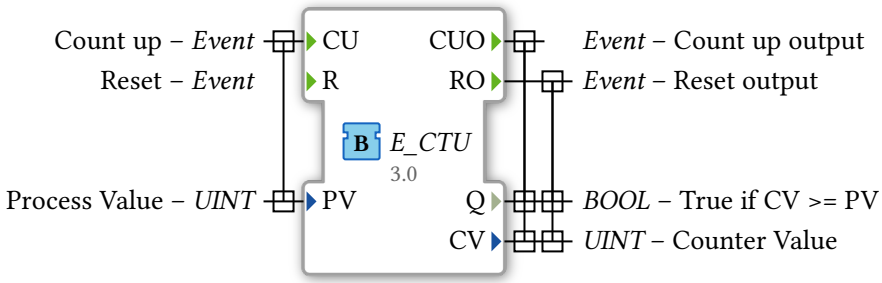
---



• – ◦ • – ◦ • – ◦ •

Introduction

The `E_CTU` (Event-Driven Up Counter) is an event-driven up counter according to the IEC 61499 standard. Its function is to increment an internal counter value with each incoming counting event and compare this value to a predefined limit. The block can be reset at any time.



Interface Structure

Event Inputs

- **CU (Count Up)**: Triggers a counting step that increments the counter value `CV` by 1.
- **Related Data**: `PV`
- **R (Reset)**: Resets the counter value `CV` to 0.

Event Outputs

- **CUO (Count Up Output)**: Confirms a count increment. Triggered after each `CU` event.
- **Related Data**: `Q`, `CV`
- **RO (Reset Output)**: Confirms the counter reset.
- **Related Data**: `Q`, `CV`

Data Inputs

- **PV (Preset Value)**: The threshold value (data type: `UINT`). This value is compared to the counter reading on every `CU` event.

Data Outputs

- **Q (Status)**: Output flag that is set when the counter reading reaches or exceeds the threshold value (data type: `PV`) (data type: `BOOL`).
- **CV (Counter Value)**: The current counter value (data type: `UINT`).

Functionality

The `E_CTU` block has two main functions: counting and resetting.

1. **Counting (CU)**: When a `CU` event occurs and the internal counter value `CV` has not yet reached the maximum value for `UINT` (65535), `CV` is incremented by 1. Then, `CV` is compared to the limit value at the `PV` input. If `CV >= PV` is the current value, the output `Q` is set to `TRUE`; otherwise, it is set to `FALSE`. After the counting process, the event `CUO` is triggered, which outputs the current counter value `CV` and the status flag `Q`.

If `CV >= PV` is the current value, the output `Q` is set to `TRUE`; otherwise, it is set to `FALSE`. 2. **Reset (R)**: When a `R` event occurs, the counter value `CV` is immediately reset to 0 and the status flag `Q` is set to `FALSE`. Subsequently, the `RO` event is triggered, which outputs the reset values `CV` and `Q`.

Technical Features

- **Event-driven**: This function block operates exclusively based on events (`CU`, `R`).
- **Overflow Protection**: The counter stops when the maximum value for `UINT` (65535) is reached to prevent an overflow.
- **PV at Each Count Step**: The limit value `PV` is linked to the `CU` event, meaning it can potentially be changed at each count step.

Application Examples

- **Piece Counter**: Counting produced parts on a conveyor belt. When a target quantity (`PV`) is reached, `Q` becomes `TRUE`.
- **Event Counting**: Recording the frequency of events, such as the activation of a switch.
- **Cycle Counter**: Counts cycles in a machine to signal maintenance intervals.

⚖️ Comparison with similar function blocks

Feature	E_CTU (Up Counter)	E_CTD (Down Counter)	E_CTUD (Up/Down Counter)
Counting Direction	Up	Down	Both
Event-Driven	Yes	Yes	Yes
Reset Function	R (Reset to 0)	LD (Set to PV)	R (Reset to 0)

🛠️ Related exercises

- Uebung_040
- Uebung_040_2
- Uebung_040_AX
- Uebung_041
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_084
- Uebung_12x_sub

Conclusion

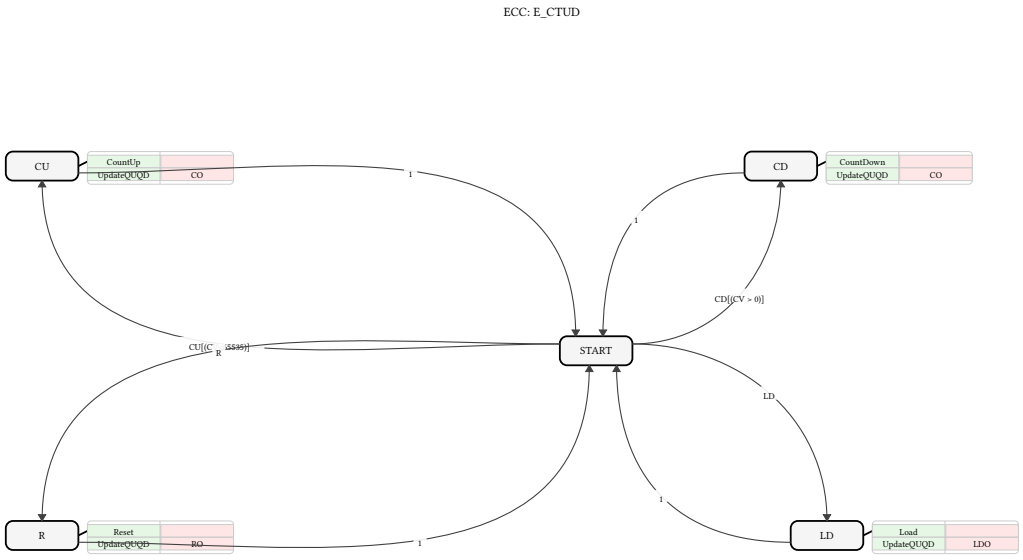
The `E_CTU` is a basic and versatile counter module for event-driven systems according to IEC 61499. Its simple interface and predictable behavior make it a robust tool for a wide range of counting and monitoring tasks in industrial automation.

🌐 Related topic subpages on ms-muc-docs.de

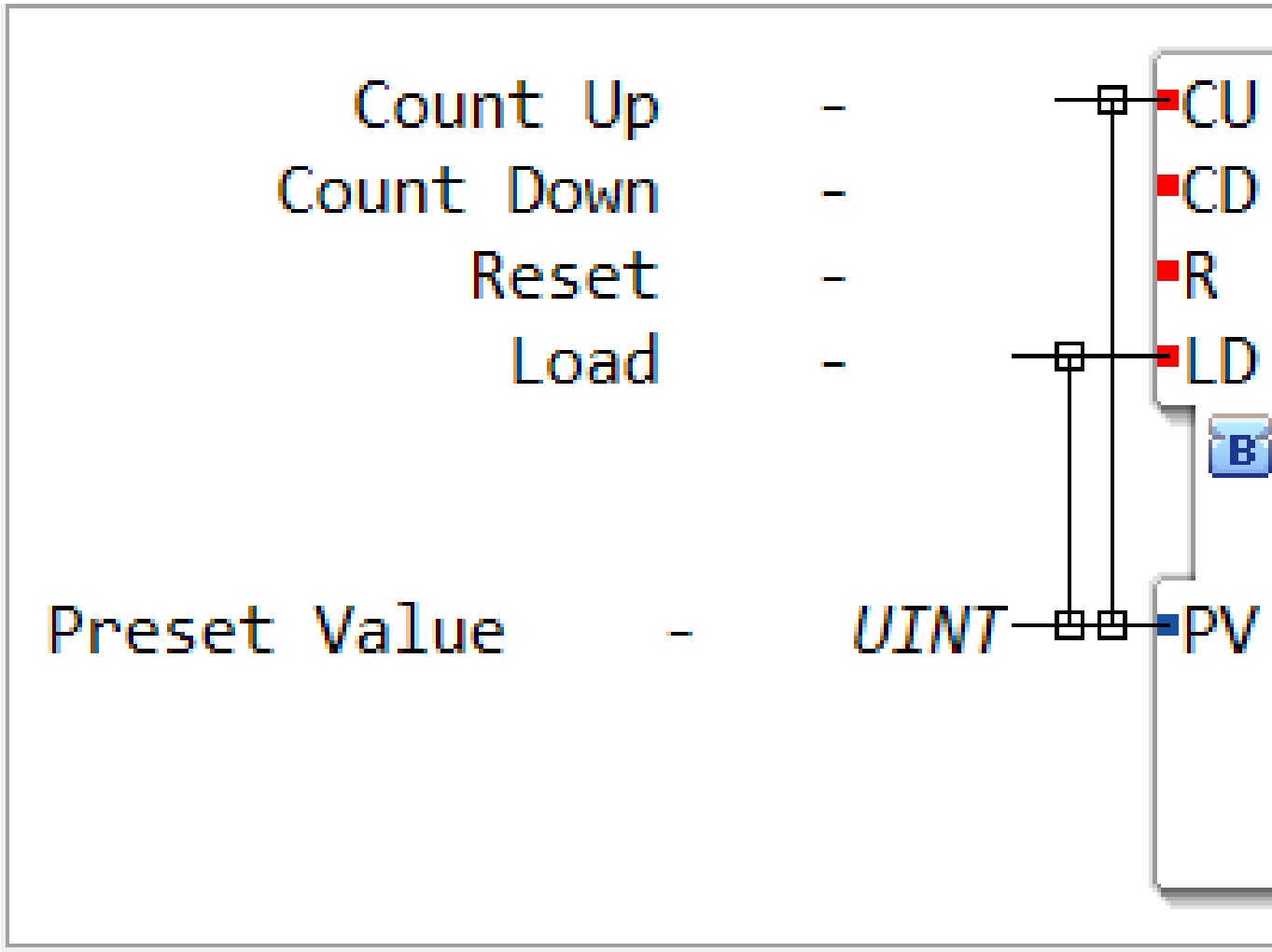
- 🌐 E_CTU Event Counter Module on ms-muc-docs.de

2.11.3.6 E_CTUD

🎧 Podcast

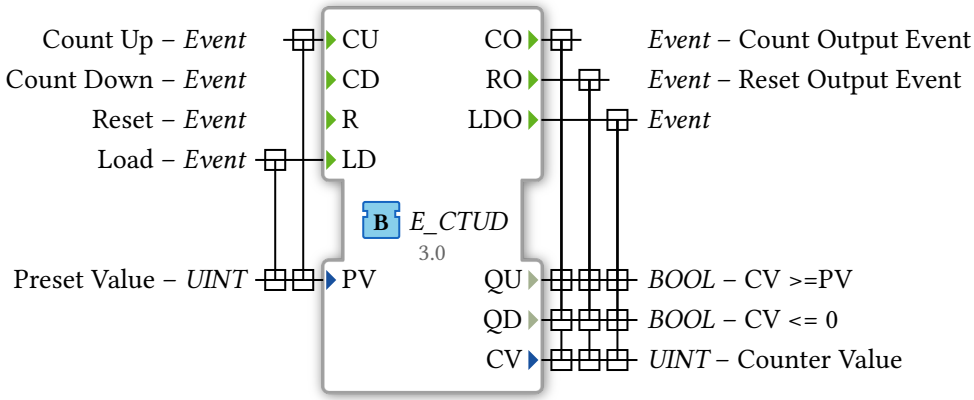


- E_CTUD: Bidirectional Counter in IEC 61499 Systems



• – ◦ • – ◦ • – ◦

The `E_CTUD` (Event-Driven Up-Down Counter) is an event-driven up- and down-counter compliant with the IEC 61499 standard. It can increment, decrement, reset, or load a counter value based on separate events. This makes it a flexible and powerful component for a wide variety of counting applications.



- **CU (Count Up)**: Triggers an up count.
- **Linked Data**: `PV`
- **CD (Count Down)**: Triggers a down count.
- **R (Reset)**: Resets the counter to 0.
- **LD (Load)**: Loads a new value into the counter.
- **Linked Data**: `PV`

- **CO (Count Output)**: Acknowledges a counting operation (`CU` or `CD`).
- **Linked Data**: `QU`, `CV`, `QD`
- **RO (Reset Output)**: Confirms that the counter has been reset.
- **Linked Data**: `QU`, `CV`, `QD`
- **LDO (Load Output)**: Confirms that a new counter value has been loaded.
- **Linked Data**: `QU`, `CV`, `QD`
- **PV (Preset Value)**: The threshold value for `QU` or the value to be loaded for `LD` (Data type: `UINT`).
- **QU (Status Up)**: Output flag that is set when `TRUE` (Data type: `CV >= PV`) is reached.
- **CV (Counter Value)**: The current counter value (data type: `UINT`).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

The `E_CTUD` counter responds to four different events:

1. **Count Up (CU)**: When a `CU` event occurs and `CV` is less than the maximum value (65535), `CV` is incremented by 1. The `CO` event is then triggered.
2. **Count Down (CD)**: If a `CD` event occurs and `CV` is greater than 0, `CV` is decremented by 1. The `CO` event is then triggered.
3. **Reset (R)**: If a `R` event occurs, `CV` is set to 0. The `RO` event is then triggered.
4. **Load (LD)**: When a `LD` event occurs, `CV` is set to the value of `PV`. Then, the `LDO` event is triggered.

After each of these actions, the status flags `QU` and `QD` are updated based on the new value of `CV` (`QU = (CV >= PV)` and `QD = (CV == 0)`). The respective output events (`CO`, `RO`, `LDO`) then output the current counter value (`CV`) and the two status flags.

- **Bidirectional Counting**: The function block supports both up and down counting in a single block.
- **Comprehensive Control**: In addition to counting, it also offers functions for explicit loading and resetting.
- **Two Status Outputs**: `QU` signals when the upper limit is reached, and `QD` signals when the lower limit (0) is reached.
- **Overflow and Underflow Protection**: Counting operations are only performed within the valid limits (0 to 65535).
- **Position Detection**: Counting incremental encoder steps in both directions.
- **Fill Level Control**: Detecting inflows and outflows in a tank.
- **Storage Location Management**: Counting incoming and outgoing pallets.

Feature	E_CTUD (Up/Down)	E_CTU (Up)	E_CTD (Down)
Counting Direction	Up & Down	Up Only	Down Only
Reset (to 0)	Yes (`R`)	Yes (`R`)	No
Load (to PV)	Yes (`LD`)	No	Yes (`LD`)
Top Status	`QU` (`CV >= PV`)	`Q` (`CV >= PV`)	No
Bottom Status	`QD` (`CV = 0`)	No	`Q` (`CV = 0`)

- Exercise_082

The `E_CTUD` is a universal counter module that combines and extends the functionality of a simple up and down counter. With its four control events (`CU`, `CD`, `R`, `LD`) and two status outputs (`QU`, `QD`), it offers maximum flexibility for complex counting and monitoring tasks in industrial automation.

- 🌐 E_CTU Event Counter module on ms-muc-docs.de

Technical Features

Application Scenarios

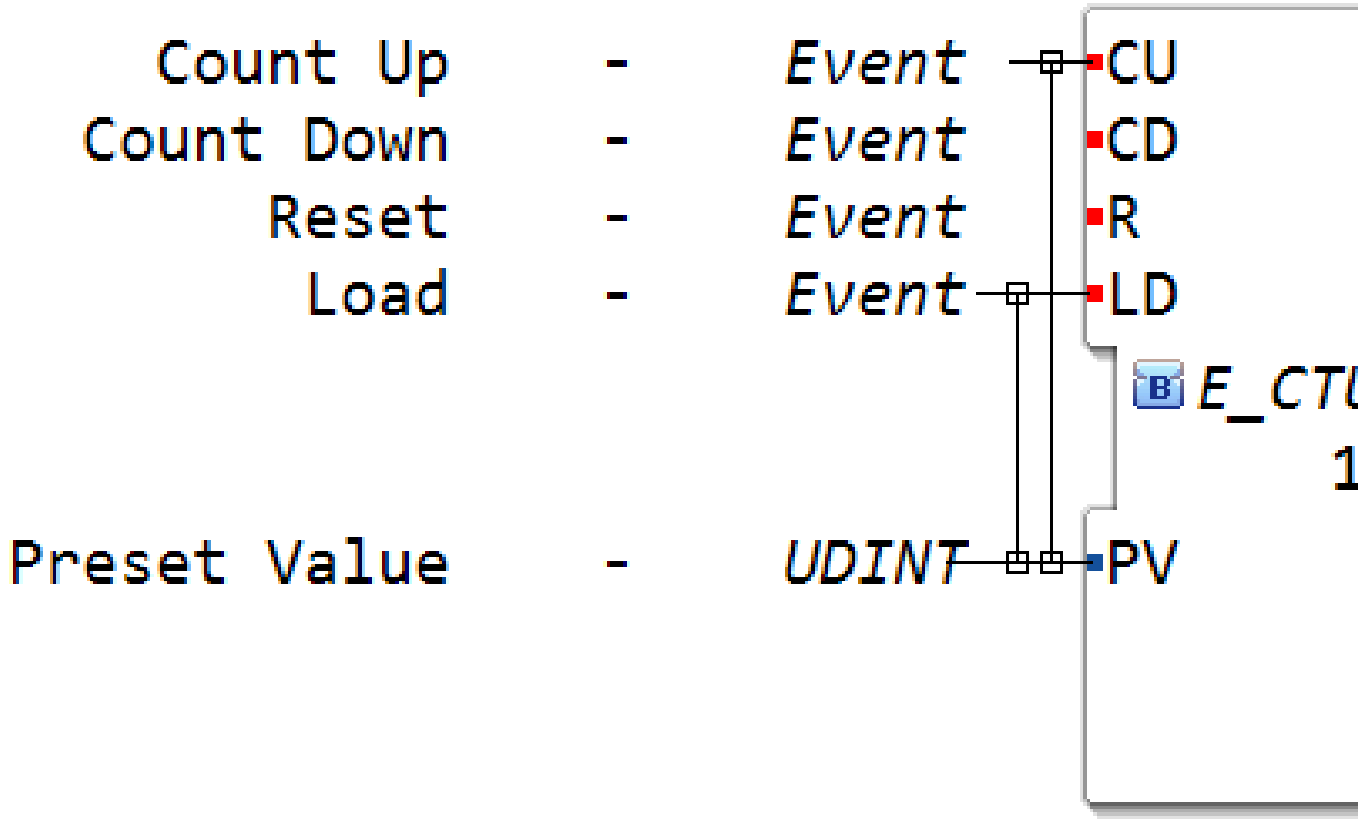
⚖️ Vergleich mit ähnlichen Bausteinen

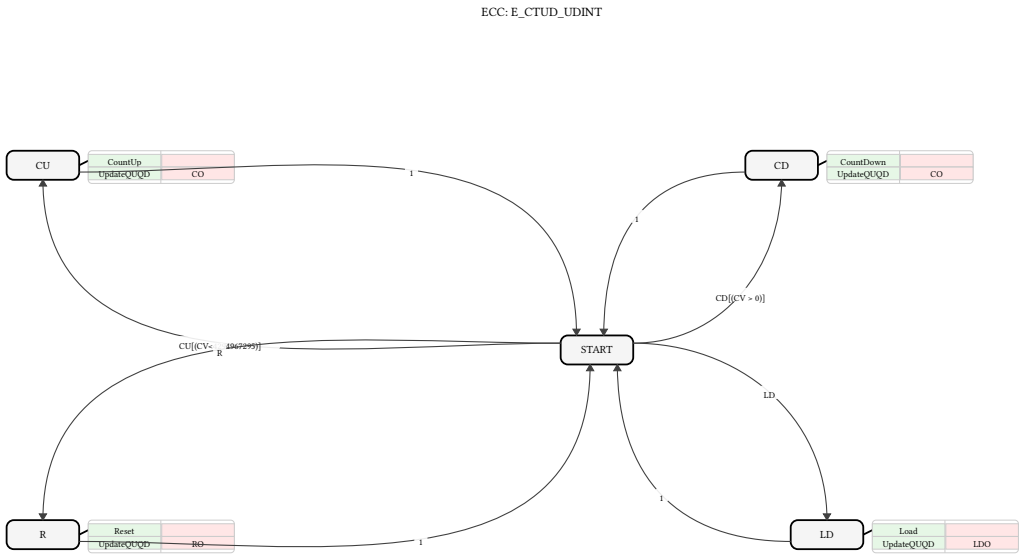
🛠️ Zugehörige Übungen

Conclusion

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

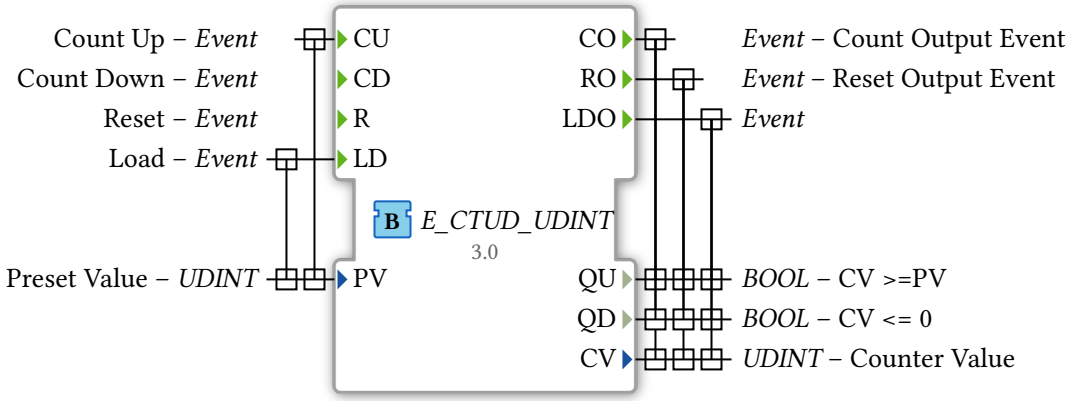
2.11.3.7 E_CTUD_UDINT





• – ○ • – ○ • – ○ •

The `E_CTUD_UDINT` is a variant of the `E_CTUD` counter that uses the `UDINT` data type (Unsigned Double Integer, 32-bit). This event-driven up and down counter can cover a very large counter range. It can increment, decrement, reset, or load a counter value with a predefined value based on separate events.



- **CU (Count Up)**: Triggers an up count.
- **Related Data**: `PV`
- **CD (Count Down)**: Triggers a countdown.
- **R (Reset)**: Resets the counter to 0.
- **LD (Load)**: Loads a new value into the counter.
- **Related Data**: `PV`
- **CO (Count Output)**: Acknowledges a counting operation (`CU` or `CD`).
- **Linked Data**: `QU`, `CV`, `QD`
- **RO (Reset Output)**: Confirms that the counter has been reset.
- **Linked Data**: `QU`, `CV`, `QD`
- **LDO (Load Output)**: Confirms that a new counter value has been loaded.
- **Linked Data**: `QU`, `CV`, `QD`
- **PV (Preset Value)**: The threshold value for `QU` or the value to be loaded for `LD` (Data type: `UDINT`).
- **QU (Status Up)**: Output flag that is set when `TRUE` (Data type: `CV >= PV`) (Data type: `BOOL`) is reached.
- **CV (Counter Value)**: The current counter value (data type: `UDINT`).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

The `E_CTUD_UDINT` responds to four different events:

1. **Count Up (CU)**: When a `CU` event occurs and `CV` is less than the maximum value (4,294,967,295), `CV` is incremented by 1. The `CO` event is then triggered.
2. **Count Down (CD)**: If a `CD` event occurs and `CV` is greater than 0, `CV` is decremented by 1. The `CO` event is then triggered.
3. **Reset (R)**: If a `R` event occurs, `CV` is set to 0. The `RO` event is then triggered.
4. **Load (LD)**: When a `LD` event occurs, `CV` is set to the value of `PV`. Then, the `LDO` event is triggered.

After each of these actions, the status flags `QU` and `QD` are updated based on the new value of `CV` (`QU = (CV >= PV)` and `QD = (CV == 0)`). The respective output events (`CO`, `RO`, `LDO`) then output the current counter value (`CV`) and the two status flags.

- **Large Counting Range**: By using `UDINT`, the counter can take values from 0 to 4,294,967,295.
- **Bidirectional Counting**: The function block supports both up and down counting in a single block.
- **Comprehensive Control**: In addition to counting, it also offers functions for explicit loading and resetting.
- **Overflow and Underflow Protection**: Counting operations are only performed within the valid `UDINT` limits (0 to 4,294,967,295).
- **Total Counter**: Recording total production quantities or operating hours over the entire lifespan of a machine, where a 16-bit counter is insufficient.
- **Energy Measurement**: Counting pulses from an energy meter (e.g., Wh or kWh) over extended periods.
- **High-Resolution Position Detection**: Counting a very large number of increments from a high-resolution encoder.
- Exercise_009
- Exercise_034b
- Exercise_083

The `E_CTUD_UDINT` is the `UDINT` variant of the universal `E_CTUD` counter. It offers the same functionality, but with a significantly larger counting range (32 bits). This makes it the ideal choice for applications where the counter value can exceed the limit of a 16-bit `UINT` counter. Its robust, event-driven nature and comprehensive control and status functions are retained.

- 🌐 E_CTU Event Counter module on ms-muc-docs.de

Technical Features

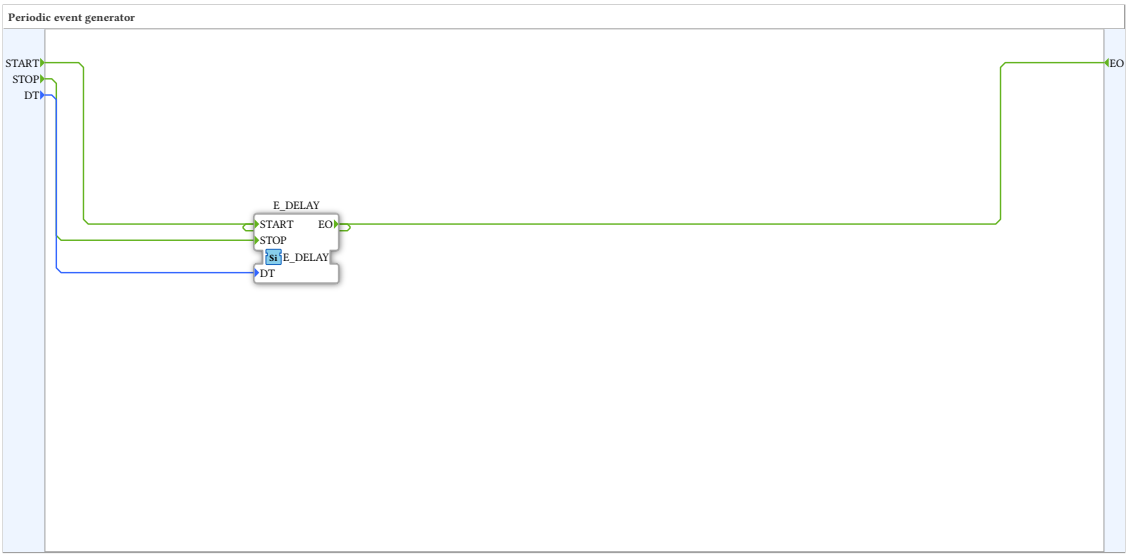
Application Scenarios

🛠️ Zugehörige Übungen

Conclusion

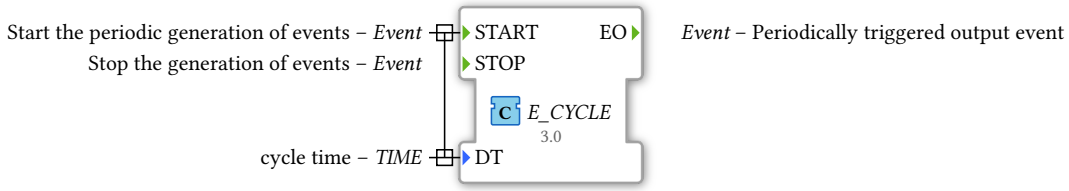
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

2.11.3.8 E_CYCLE



Introduction

The `E_CYCLE` is a function block according to IEC 61499 that serves as a periodic event generator. After starting, the block repeatedly fires an output event (`EO`) at a defined time interval (`DT`). It is the fundamental tool for creating clocks and cyclic sequences.



Interface Structure

Event Inputs

- **START**: Starts the periodic generation of `EO` events.
- **Related Data**: `DT`
- **STOP**: Stops the generation of `EO` events.

Event Outputs

- **EO (Event Output)**: The cyclically triggered output event.

Data Inputs

- **DT (Delay Time)**: The time interval between individual `EO` events (data type: `TIME`).

Functionality

The functionality of `E_CYCLE` is internally based on a feedback loop with a `E_DELAY` block.

1. **Starting the Cycle**:

- A `START` event at the input triggers the internal timer with the duration specified at the `DT` input.
- After this time `DT` has elapsed, the first `EO` event is triggered at the output.

1. **Cyclic Operation**:

- The triggered `EO` event is immediately fed back internally to the timer's start input.
- This immediately restarts the timer, and after another time interval `DT`, the next `EO` event is triggered.
- This process repeats continuously, generating a periodic sequence of `EO` events at intervals of `DT`.

1. **Stopping the Cycle**:

- A `STOP` event at the input breaks the internal feedback loop.
- The timer stops, and no further `EO` events are generated until another `START` event occurs.

Technical Features

- **Event Generator**: The module generates a chain of events, not a continuous state (such as a square wave). To implement blinking, the `EO` event would, for example, trigger a `E_T_FF` (toggle flip-flop).
- **Precise timing control**: Enables precise definition of the cycle time via the `TIME` data type.
- **Event-driven architecture**: All control (start/stop) is purely event-based.

Application scenarios

- **Clock generator**: Periodically triggers other function blocks at a fixed clock rate (e.g., for sampling or calculations).
- **Blinker/warning lights**: As a pulse generator for a downstream flip-flop to produce a visual signal.
- **Watchdog**: Periodically sends a "life sign." If the signal is absent, an error can be detected.
- **Test Automation**: Generation of periodic stimuli for systems under test.

⚖️ **Comparison with similar components**

Feature	E_CYCLE	E_DELAY	E_PULSE (hypothetical)
Function	Periodic Clock	Single Delay	Single Pulse of Fixed Duration
Repetition	Continuous	Once per `START` event	Once per `REQ` event
Control	START/STOP	START/STOP	REQ

🛠️ **Related exercises**

- Uebung_007
- Uebung_007_AX
- Uebung_007a1
- Uebung_007a1_AX
- Uebung_007a2
- Uebung_007a2_AX
- Uebung_007a3
- Uebung_008
- Uebung_009
- Uebung_020c3
- Uebung_020c3_AX
- Uebung_020e2
- Uebung_020e2_AX
- Uebung_020f2
- Uebung_020f2_AX
- Uebung_084
- Uebung_152
- Exercise_153

Conclusion

The `E_CYCLE` function block is an essential tool for all applications requiring periodic or cyclic logic. It easily implements a clock generator whose frequency can be precisely set via the `DT` parameter. Its purely event-based behavior makes it an efficient and fundamental component in the IEC 61499 system architecture.

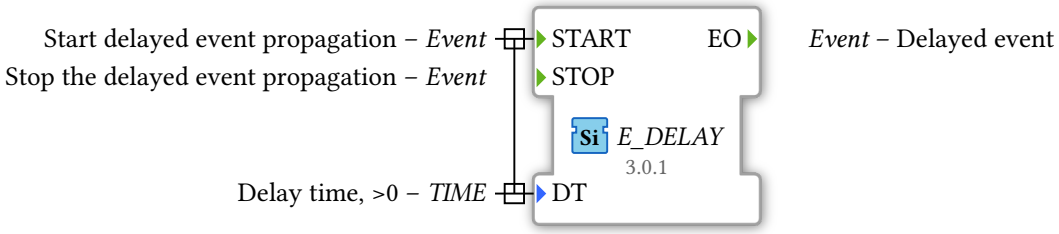
2.11.3.9 E_DELAY

🎧 Podcast

- E_DELAY in IEC 61499: Precise, Cancelable Time Delay in Control Systems

Introduction

The **E_DELAY** is a key functional block in the IEC 61499 standard for industrial control systems. As an event-driven timer, it enables the delayed triggering of events with a configurable duration and the option to cancel. This block is widely used in time-controlled automation processes.



Structure of the E_DELAY Block

Interface

Event Inputs:

- `START`: Initializes the delay
- `STOP`: Terminates the active delay

Event Outputs:

- `EO` (Event Output): Triggered after the delay has elapsed

Data Inputs:

- `DT` (Delay Time): TIME value for the delay duration

Functionality

1. **Delay Start:**

- On the `START` event, the timer starts with the set time `DT`
- During runtime, further `START` events ignored

1. **Delay Termination:**

- The `STOP` event immediately terminates the active delay
- No `EO` event is generated

1. **Delay Completion:**

- The `EO` event is triggered exactly after `DT`
- If DT ≤ T#0s, immediate triggering occurs without delay

Technical Features

✓ **Precise Time Control** (millisecond-accurate delay)
✓ **Termination Function** for flexible process control
✓ **Event-driven** (no cyclical call required)
✓ **Real-time capable** for industrial applications

Application Scenarios

- **Machine Safety**: Delayed shutdown of systems
- **Process Control**: Time-controlled process steps
- **Alarm Management**: Delayed fault messages
- **Test Automation**: Time-controlled test sequences

⚖️ Comparison with similar function blocks

Feature	E_DELAY	E_CYCLE	E_PULSE
Operating principle	Single delay	Continuous cycle	Single pulse
Abort option	Yes	Yes	No
Time parameters	DT (Delay)	DT (Cycle time)	PT (Pulse duration)

🛠️ Related exercises

- Uebung_018
- Uebung_020b
- Uebung_020b_AX
- Uebung_020d
- Uebung_020d_AX
- Uebung_024
- Uebung_025
- Uebung_026
- Uebung_160b
- Exercise_160b_AX

Conclusion

The E_DELAY function block is an indispensable tool for time-controlled automation processes. Its main advantages are:

- Precise and flexible timing control
- Immediate response to control events
- Robust integration into distributed control architectures

Thanks to its reliable operation and compliance with IEC 61499, it is ideally suited for safety applications and complex process control in industrial environments. Its cancel function makes it particularly valuable for flexible production scenarios.

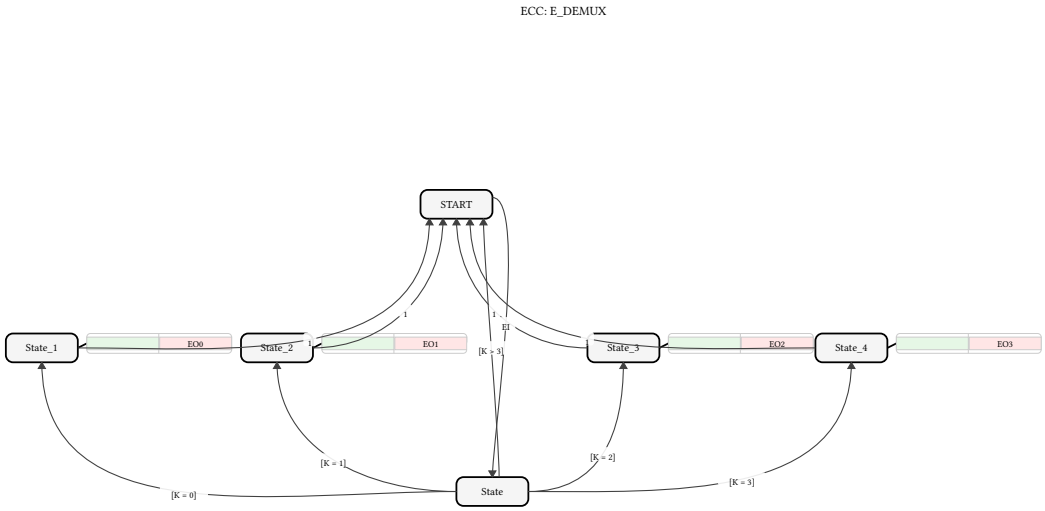
2.11.3.10 E_DEMUX

Introduction

The `E_DEMUX` (Event Demultiplexer) is a function block according to IEC 61499 that forwards a single input event (`EI`) to one of several outputs. The selection of the specific output is determined by the value of an input variable (`K`). This version of the block is a 1-to-4 demultiplexer.



Interface Structure



Event Inputs

- **EI (Event Input)**: The input event to be distributed.
- **Related Data**: `K`

Event Outputs

- **EO0**: Triggered when `EI` and `K = 0` arrive.
- **EO1**: Triggered when `EI` and `K = 1` arrive.
- **EO2**: Triggered when `EI` and `K = 2` arrive.
- **EO3**: Triggered when `EI` and `K = 3` arrive.

Data Inputs

- **K**: The selection index that determines which output is triggered (data type: `UINT`).

Functionality

1. **Event Reception**: The function block waits for an event at input `EI`.
2. **Selection**: When the `EI` event arrives, the value of the data variable `K` is evaluated.
3. **Forwarding**:

- If `K` = 0, the event is forwarded to `EO0`.
- If `K` = 1, the event is forwarded to `EO1`.
- If `K` = 2, the event is forwarded to `EO2`.
- If `K` = 3, the event is forwarded to `EO3`.

1. **Invalid Index**: If the value of `K` is outside the valid range [0, 3], no output event is triggered, and the `EI` event is discarded.

The input event is therefore always forwarded exclusively to exactly one output, provided the index `K` is valid.

Technical Features

- **1-to-4 Distribution**: This function block distributes an event to four possible outputs.
- **Index-driven**: The logic is based on a numerical index.
- **Deterministic behavior**: The routing is clearly and unambiguously defined, ensuring reliable control.

Application scenarios

- **State machines**: Selection of the next state transition based on a calculated index.
- **Mode switching**: Activation of different plant components depending on the selected operating mode (`K` = mode number).
- **Sequencers/Step chains**: Activation of the next step in a chain.
- **Error routing**: Routing of a general error event to a specific handler based on an error code (`K` = error code).

⚖️ Comparison with similar function blocks

Feature	E_DEMUX (this)	E_MUX	E_SWITCH
Operating principle	1:4 distribution	n:1 merging	1:2 distribution
Control	Index `K` [0-3]	Index `K`	`BOOL` condition `G`
Event flow	Splitting	Merging	Conditional switch

Note: Other variants exist, such as `E_DEMUX_2` and `E_DEMUX_8` for 2 and 8 outputs, respectively.

🛠️ Related Exercises

- Exercise_040
- Exercise_040_AX
- Exercise_087

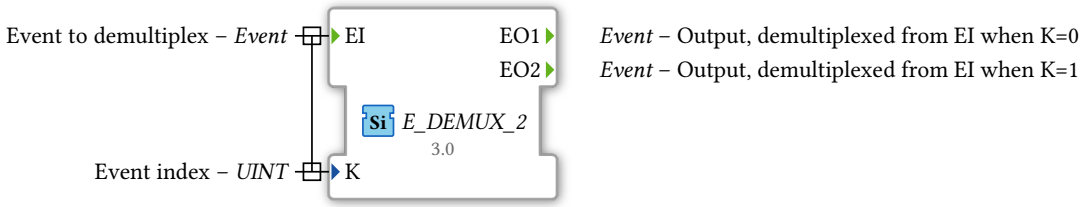
Conclusion

The `E_DEMUX` is a fundamental building block for controlling event flow in IEC 61499 applications. It enables a clear, index-based division of event streams and is therefore a key tool for implementing state logic, mode switching, and sequence control.

2.11.3.11 E_DEMUX_2

• – ◦ • – ◦ • – ◦ •

The `E_DEMUX_2` (Event Demultiplexer) is a function block according to IEC 61499 that forwards a single input event (`EI`) to one of two outputs. The output selection is determined by the value of the input variable `K`.



- **EI (Event Input)**: The input event to be distributed.
- **Associated Data**: `K`
- **EO1**: Triggered when `EI` arrives and the selection index is `K = 0`.
- **EO2**: Triggered when `EI` arrives and the selection index is `K = 1`.
- **K**: The zero-based selection index that determines which output is triggered (data type: `UINT`).

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

1. **Event Reception**: The function block waits for an event at input `EI`.

2. **Forwarding**:

If `K` = 0, the event is forwarded to `EO1`.
If `K` = 1, the event is forwarded to `EO2`.

1. **Invalid Index**: If the value of `K` is outside the valid range [0, 1], no output event is triggered.

The input event is therefore always forwarded exclusively to exactly one output, provided the index `K` is valid.

- **1-to-2 Distribution**: This function block distributes an event to two possible outputs.
- **Index-Driven**: The logic is based on a numeric index (`K`).
- **Confusing Naming Convention**: Note that the outputs are named 1-based (`EO1`, `EO2`), but the selection index `K` is 0-based (`K=0` for `EO1`, `K=1` for `EO2`).
- **Generic Building Block**: The functionality is provided by the generic class `GEN_E_DEMUX`.
- **Conditional Branching**: Splitting an event flow based on a simple condition that results in either 0 or 1.
- **State Machines**: Selection between two possible next states.
- **Activation/Deactivation**: Forwarding an event to an "on" or "off" path.

The `E_DEMUX_2` is a specialized version of the demultiplexer for cases where an event needs to be split into exactly two paths. It is useful for simple binary decisions in the event flow. The inconsistent naming of the outputs relative to the index value requires special attention during implementation.

Technical Features

Application Scenarios

Conclusion

2.11.3.12 E_DEMUX_4

• – ◦ • – ◦ • – ◦ •

Introduction

The `E_DEMUX_4` (Event Demultiplexer) is a function block according to IEC 61499 that forwards a single input event (`EI`) to one of four outputs. The output selection is determined by the value of the input variable `K`.



Interface Structure

Event Inputs

- **EI (Event Input)**: The input event to be distributed.
- **Related Data**: `K`

Event Outputs

- **EO1**: Triggered when `EI` arrives and the selection index is `K = 0`.
- **EO2**: Triggered when `EI` arrives and the selection index is `K = 1`.
- **EO3**: Triggered when `EI` arrives and the selection index is `K = 2`.
- **EO4**: Triggered when `EI` arrives and the selection index is `K = 3`.

Data Inputs

- **K**: The zero-based selection index that determines which output is triggered (data type: `UINT`).

Functionality

1. **Event Reception**: The function block waits for an event at input `EI`.
2. **Selection**: When the `EI` event arrives, the value of the data variable `K` is evaluated.
3. **Forwarding**:

- If `K` = 0, the event is forwarded to `EO1`.
- If `K` = 1, the event is forwarded to `EO2`.
- If `K` = 2, the event is forwarded to `EO3`.
- If `K` = 3, the event is forwarded to `EO4`.

1. **Invalid Index**: If the value of `K` is outside the valid range [0, 3], no output event is triggered.

Technical Features

- **1-to-4 Distribution**: This function block distributes an event across four possible outputs.
- **Index-Driven**: The logic is based on a numeric index (`K`).
- **Confusing Naming Convention**: Note that the outputs are named 1-based (`EO1` to `EO4`), but the selection index `K` is 0-based (`K=0` for `EO1`, `K=1` for `EO2`, etc.).
- **Generic Building Block**: The functionality is provided by the generic class `GEN_E_DEMUX`.

Application Scenarios

- **State Machines**: Selection of the next state transition based on a calculated index.
- **Mode Switching**: Activation of different system components depending on the selected operating mode (`K` = mode number).
- **Sequencer/Step Chains**: Activation of one of four possible next steps.

🛠️ Related Exercises

- Exercise_040_2
- Exercise_087a1
- Exercise_087a2

Conclusion

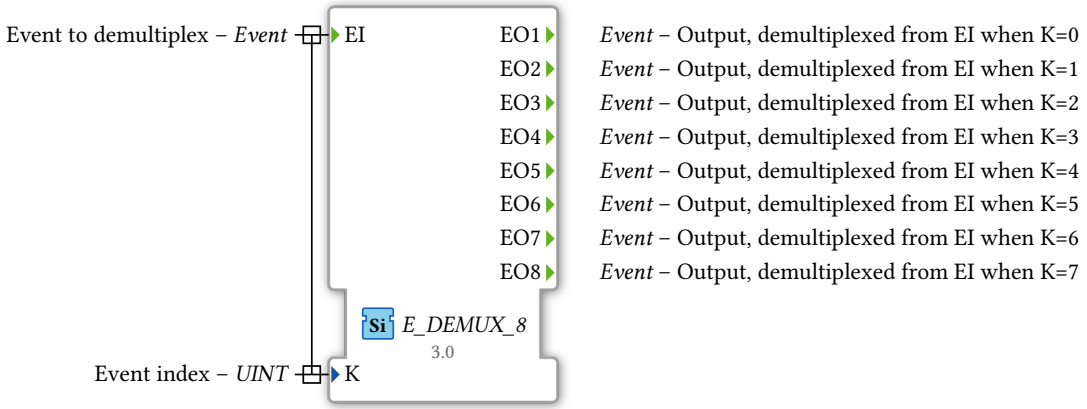
The `E_DEMUX_4` is a standard implementation of the demultiplexer principle for four outputs. It is useful for splitting an event flow into up to four paths. The inconsistent naming of the outputs in relation to the index value requires special attention during implementation.

2.11.3.13 E_DEMUX_8

• – ◦ • – ◦ • – ◦ •

Introduction

The `E_DEMUX_8` (Event Demultiplexer) is a function block according to IEC 61499 that forwards a single input event (`EI`) to one of eight outputs. The output selection is determined by the value of the input variable `K`.



Interface Structure

Event Inputs

- **EI (Event Input)**: The input event to be distributed.
- **Related Data**: `K`

Event Outputs

- **EO1**: Triggered when `EI` arrives and `K = 0` is present.
- **EO2**: Triggered when `EI` arrives and `K = 1` is present.
- **EO3**: Triggered when `EI` arrives and `K = 2` is present.
- **EO4**: Triggered when `EI` arrives and `K = 3` is present.
- **EO5**: Triggered when `EI` arrives and is `K = 4`.
- **EO6**: Triggered when `EI` arrives and is `K = 5`.
- **EO7**: Triggered when `EI` arrives and is `K = 6`.
- **EO8**: Triggered when `EI` arrives and is `K = 7`.

Data Inputs

- **K**: The zero-based selection index that determines which output is triggered (data type: `UINT`).

Functionality

1. **Event Receipt**: The function block waits for an event at input `EI`.
2. **Selection**: When the `EI` event arrives, the value of the data variable `K` is evaluated.
3. **Forwarding**: The event is forwarded to output `EO(K+1)`. For example:

- If `K` = 0, the event is forwarded to `EO1`.
- If `K` = 7, the event is forwarded to `EO8`.

1. **Invalid Index**: If the value of `K` is outside the valid range [0, 7], no output event is triggered.

Technical Features

- **1-to-8 Distribution**: This function block distributes an event across eight possible outputs.
- **Index-Driven**: The logic is based on a numeric index (`K`).
- **Confusing Naming Convention**: Note that the outputs are named 1-based (`EO1` to `EO8`), but the selection index `K` is 0-based (`K=0` for `EO1`, `K=7` for `EO8`, etc.).
- **Generic Block**: The functionality is provided by the generic class `GEN_E_DEMUX`.

Application Scenarios

- **State Machines**: Selection of the next state transition from up to eight possibilities.
- **Mode Switching**: Activation of different system components depending on the selected operating mode.
- **Communication Router**: Forwards messages or commands to one of eight channels.

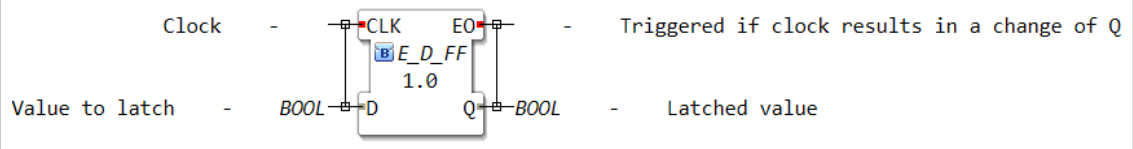
🛠️ Related Exercises

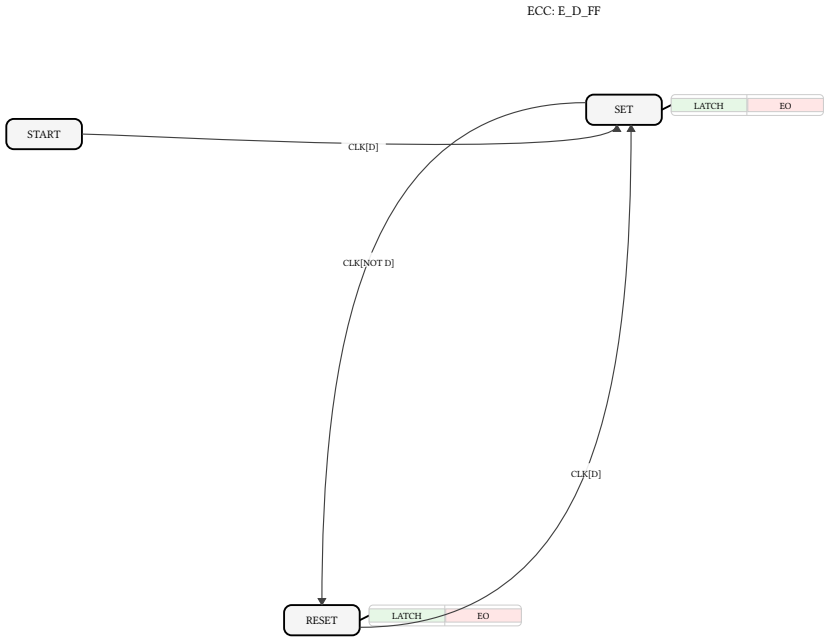
- Exercise_006c
- Exercise_041

Conclusion

The `E_DEMUX_8` is a useful implementation of the demultiplexer principle for use cases with up to eight different event paths. As with the other `E_DEMUX` blocks, the inconsistent naming of the outputs relative to the index value requires special attention during implementation.

2.11.3.14 E_D_FF

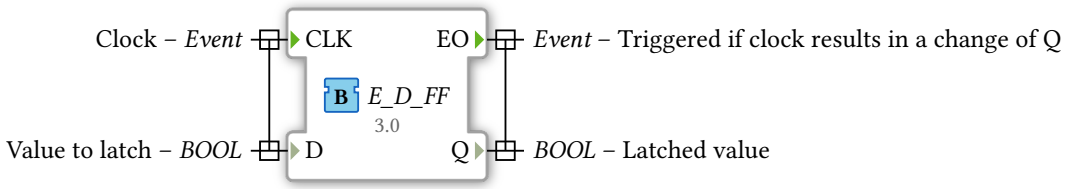




• – ○ • – ○ • – ○ •

Introduction

IEC 61499 is an international standard for modeling distributed industrial control systems. The **E_D_FF** (Data Flip-Flop) is a fundamental memory element in this standard that stores digital signals synchronously with a clock signal. This functional block plays a central role in state machines and memory applications in industrial automation solutions.



Structure of the E_D_FF Block

Interface

Event Inputs:

- `CLK` (Clock): Clock event that triggers the transfer of the data input `D`.
- **Linked Data**: `D`

Event Outputs:

- `EO` (Event Output): Triggered when the `CLK` pulse results in a change to the output `Q`.
- **Linked Data**: `Q`

Data Inputs:

- `D`: The data value to be stored (data type: `BOOL`).

Data Outputs:

- `Q`: The currently stored value (data type: `BOOL`).

Functionality

1. **Data Storage:**

- On each rising edge of the `CLK` signal, the value of `D` is transferred to `Q`.
- When the value changes, the `EO` event is triggered.

1. **Data Retention:**

- Between clock edges, the stored value `Q` remains stable.
- Changes at the `D` input have no effect without a clock edge.

Technical Features

✓ **Clock-controlled storage** for synchronous systems ✓ **Event output** for change detection ✓ **Deterministic behavior** in real-time systems ✓ **Easy integration** into IEC 61499 applications

Application Scenarios

- **State Storage** in Automation Processes
- **Input Buffering** for Operator Input
- **Edge Detection** in Signal Processing Chains
- **Synchronization** between Asynchronous System Components

⚖️ Comparison with Similar Function Blocks

Feature	E_D_FF	E_SR	E_R_TRIG
Memory Type	D Flip-Flop	SR Latch	Edge Detector
Clocking	Required	None	None
Data Retention	Yes	Yes	No
Event Output	On Change	On Set/Reset	On Edge

🛠️ Related Exercises

- Exercise_071a
- Exercise_071b
- Exercise_072b
- Exercise_085

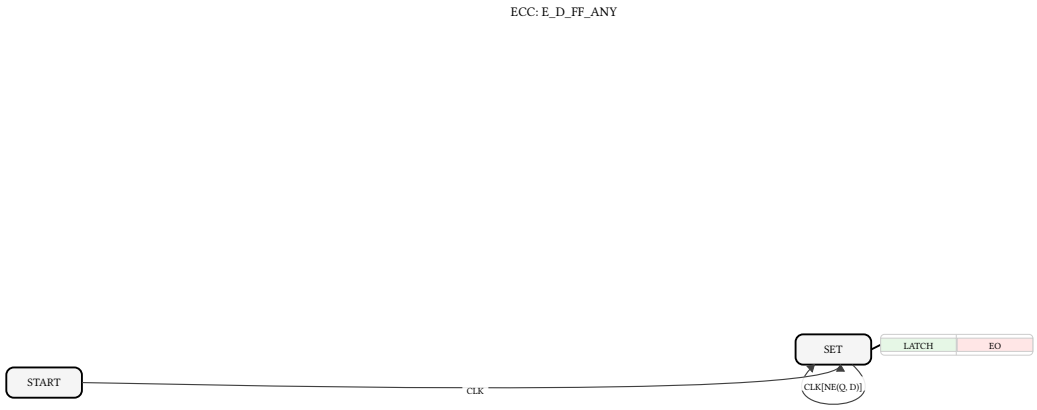
Conclusion

The E_D_FF block represents an essential memory element for IEC 61499-based control systems. Its main advantages are:

- Reliable clock-synchronous data storage
- Immediate feedback of state changes
- Robust integration into distributed control architectures

Due to its simple yet effective functionality, it forms the basis for more complex memory and state controls in industrial automation solutions. Strict adherence to the IEC 61499 standards ensures seamless interoperability with other functional modules.

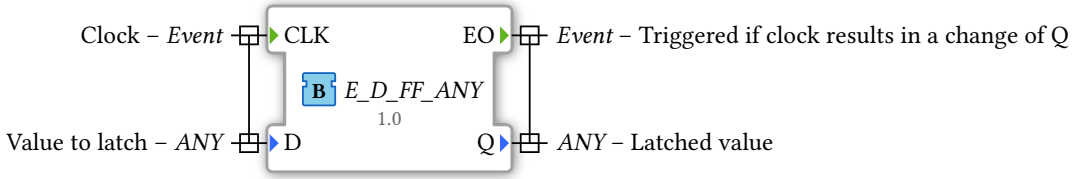
2.11.3.15 E_D_FF_ANY



• – ◦ • – ◦ • – ◦ •

Introduction

`E_D_FF_ANY` is the generically-typed variant of `E_D_FF`: while `E_D_FF` only latches `BOOL` values, `E_D_FF_ANY` accepts an input `D` of data type `ANY` and can therefore be used as a clocked latch with built-in change detection for arbitrary data types (e.g. `TIME`, `DINT`, `REAL`, `STRING`).



Interface Structure

Event Inputs

- **CLK**: Clock event, latches the current value of `D`.
  - **Connected data**: `D`

Event Outputs

- **EO**: Only triggered if `Q` actually changed as a result of the `CLK` pulse.
  - **Connected data**: `Q`

Data Inputs

- **D** (ANY): The value to be latched, of arbitrary data type.

Data Outputs

- **Q** (ANY): The last latched value, of the same data type as `D`.

Functionality

On every `CLK` event, the ECC compares the new value `D` with the currently stored `Q` (`NE(Q, D)`). Only if the value has actually changed is `Q := D` executed and `EO` triggered — if the value is unchanged, the

block remains in state SET without generating an event. The initial state START always latches the value unconditionally on the first CLK, regardless of change detection.

Technical Features

- **ANY typing**: Unlike E_D_FF (fixed to BOOL), E_D_FF_ANY can be instantiated with any IEC 61131-3 data type the target environment supports for generic ANY adaptation.
- **Built-in change detection**: Unlike a plain latch, E_D_FF_ANY only triggers EO on an actual value change — this significantly reduces event load in downstream FB networks.
- **Basis for the _TMIN variants**: E_D_FF_ANY_TMIN additionally combines this block with a minimum spacing time between two EO events.

State Overview

State	Meaning
START	Initial state, first CLK unconditionally latches D
SET	Q holds the last latched value; further CLK events only trigger EO again if NE(Q, D)

Application Scenarios

- **Change-driven downstream processing of arbitrary data types**: A TIME or REAL value should only trigger a follow-up event if it actually changed — e.g. as a building block inside composite FBs such as ATM_SUB (see the adapter library), which combines F_MOVE + E_D_FF_ANY for explicit, FBNetwork-visible change detection.
- **Generic caching**: Buffering an arbitrary value with notification only on a genuine change, independent of the concrete data type.

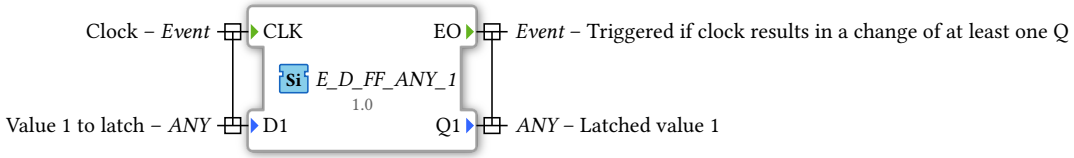
Comparison with similar function blocks

- **E_D_FF**: functionally identical, but fixed to BOOL.
- E_D_FF_ANY_TMIN: the same functionality, additionally with a minimum spacing time between EO events.
- E_D_FF_TMIN: the BOOL variant with a minimum spacing time.

Conclusion

E_D_FF_ANY provides a generically-typed, change-detecting latch and is a central building block wherever composite FBs for arbitrary data types need to trigger an event only on an actual value change.

2.11.3.16 E_D_FF_ANY_1



• – ◦ • – ◦ • – ◦ •

Introduction

E_D_FF_ANY_1 is the variant of E_D_FF_ANY for **one** latched value of any data type (ANY) and is part of the family E_D_FF_ANY_1 to E_D_FF_ANY_5 with numbered ports (D1/Q1). It is a clock-driven data latch (D flip-flop) with built-in change detection: the output event EO is only triggered if Q1 changes due to the clock.

Interface Structure

Event Inputs

- **CLK**: clock, takes over the current value of D1
  - **Associated data**: D1

Event Outputs

- **EO**: triggered if the clock causes a change of Q1

- **Associated data**: `Q1`

Data Inputs

- **D1** (ANY): value 1 to latch

Data Outputs

- **Q1** (ANY): latched value 1

Adapters

No adapters are used.

Functionality

On every `CLK` event `D1` is taken over and compared with the previously stored value `Q1`. If `D1` differs from `Q1`, the value is stored and `EO` is triggered; if the value is unchanged, no output event is produced. The block is a generic FB (`eclipse4diac::core::GenericClassName = GEN_E_D_FF_ANY`); the native implementation `GEN_E_D_FF_ANY` (FORTE) is required.

Technical Details

- **Generic block**: the types of `D1` and `Q1` are `ANY` and are fixed at instantiation
- Port names `D1`/`Q1` (instead of `D`/`Q` for `E_D_FF_ANY`) correspond to the numbered family
- Version 1.0 (2026-08-26): extension of `E_D_FF_ANY` to a numbered variable

Application Scenarios

- Adapter wrappers around functions with one result, with output only on value change (e.g. `CEIL_AR`)
- Change-driven further processing of one value of any data type

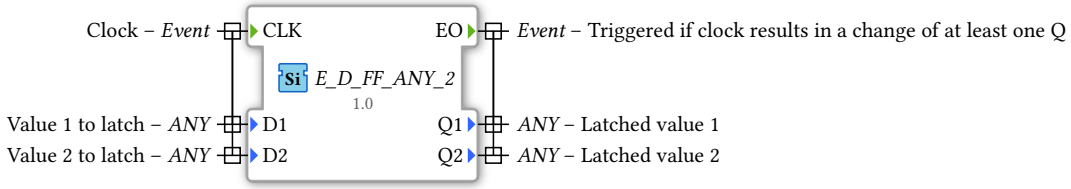
⚖️ Comparison with Similar Building Blocks

`E_D_FF_ANY` also stores one value (ports `D`/`Q`), `E_D_FF_ANY_2` two, `E_D_FF_ANY_3` three, `E_D_FF_ANY_4` four and `E_D_FF_ANY_5` five values.

Conclusion

`E_D_FF_ANY_1` is a generic, change-detecting one-value latch that only outputs `EO` if the latched value changes.

2.11.3.17 E_D_FF_ANY_2



• – ◦ • – ◦ • – ◦ •

Introduction

The `E_D_FF_ANY_2` is a data latch (D flip-flop) for **two `ANY` variables** (generic block). The output `EO` is fired only if at least one `Qi` changes due to the clock. It is the extension of `E_D_FF_ANY` to two variables.

Interface Structure

Event Inputs

- **CLK**: Clock; takes `D1` and `D2` into the latches. Associated data: `D1`, `D2`

Event Outputs

- **EO**: Triggered if the clock results in a change of at least one `Q`. Associated data: `Q1`, `Q2`

Data Inputs

- **D1** (`ANY`): Value 1 to latch

- **D2** (ANY): Value 2 to latch

Data Outputs

- **Q1** (ANY): Latched value 1
- **Q2** (ANY): Latched value 2

Adapters

This function block uses no adapters.

How it works

On every CLK event, D1 and D2 are compared with the stored values Q1 and Q2. If at least one Di differs from the stored Qi, the values are taken over and EO is fired. If both values are unchanged, no EO is fired.

Technical Details

- Service interface FB (generic, ANY types); it requires the native implementation GEN_E_D_FF_ANY (FORTE).
- Unlike the simple D flip-flop E_D_FF, EO is fired only on an actual change.
- The variant for three variables is E_D_FF_ANY_3.

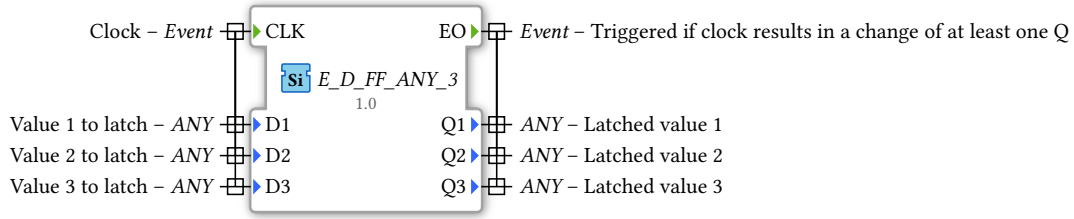
Application Scenarios

- Jointly latching related values (e.g. control output and control error) and forwarding only on change, to avoid unnecessary follow-up events

Summary

E_D_FF_ANY_2 stores two arbitrary values and signals an event only if at least one of them has changed.

2.11.3.18 E_D_FF_ANY_3



• – ◦ • – ◦ • – ◦ •

Introduction

The E_D_FF_ANY_3 is the variant of E_D_FF_ANY for **three** simultaneously latched values of arbitrary data type (ANY). It is a clock-driven data latch with built-in change detection: the output event EO is triggered only if at least one of the three outputs Q1..Q3 actually changes due to the clock.

Interface Structure

Event Inputs

- **CLK**: clock, takes over the current values of D1, D2 and D3
  - **Associated data**: D1, D2, D3

Event Outputs

- **EO**: triggered if the clock results in a change of at least one Qi
  - **Associated data**: Q1, Q2, Q3

Data Inputs

- **D1** (ANY): value 1 to latch
- **D2** (ANY): value 2 to latch
- **D3** (ANY): value 3 to latch

Data Outputs

- **Q1** (ANY): latched value 1
- **Q2** (ANY): latched value 2
- **Q3** (ANY): latched value 3

Adapters

No adapters are used.

Functionality

On every `CLK` event, `D1`, `D2` and `D3` are taken over and compared with the previously stored values `Q1`, `Q2` and `Q3`. If at least one `Di` differs from its `Qi`, the values are stored and `EO` is triggered; if all values are unchanged, no output event is produced. The block is a generic service interface FB whose native implementation is `GEN_E_D_FF_ANY` (FORTE).

Technical Details

- **Generic block**: the types of `D1`..`D3` and `Q1`..`Q3` are `ANY` and are fixed on instantiation
- **Common change detection**: a single `EO` reports the change of any of the three values; which one has changed cannot be seen from the event
- Serves to pass on several related values (e.g. control output, control error and limit flag of a controller) with just one event and to avoid follow-up events when the values are unchanged

Application Scenarios

- Adapter wrappers around functions delivering three results, with output only on value change (e.g. `CTRL_PI_AR`)
- Change-driven further processing of several values of arbitrary type

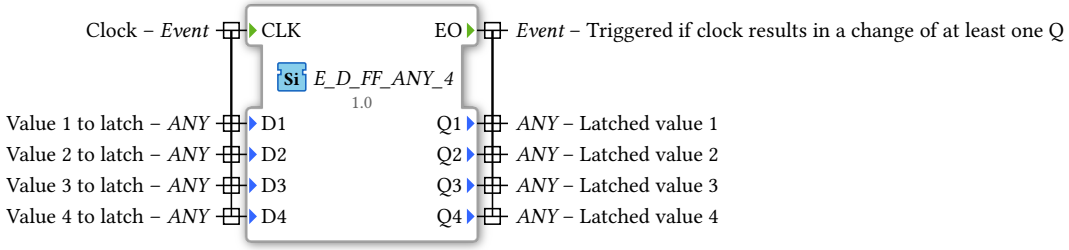
⚖️ Comparison with Similar Blocks

`E_D_FF_ANY` latches one value, `E_D_FF_ANY_3` three. Corresponding variants with another number of values exist as `E_D_FF_ANY_1`, `E_D_FF_ANY_2`, `E_D_FF_ANY_4` and `E_D_FF_ANY_5`.

Conclusion

`E_D_FF_ANY_3` is a generic, change-detecting three-value latch that outputs `EO` only if at least one of the three latched values changes.

2.11.3.19 E_D_FF_ANY_4



• – ◦ • – ◦ • – ◦ •

Introduction

`E_D_FF_ANY_4` is the variant of `E_D_FF_ANY` for **four** simultaneously latched values of any data type (`ANY`). It is a clock-driven data latch (D flip-flop) with built-in change detection: the output event `EO` is only triggered if at least one of the four outputs `Q1`..`Q4` changes due to the clock.

Interface Structure

Event Inputs

- **CLK**: clock, takes over the current values of `D1`..`D4`
  - **Associated data**: `D1`, `D2`, `D3`, `D4`

Event Outputs

- **EO**: triggered if the clock causes a change of at least one `Qi`
  - **Associated data**: `Q1`, `Q2`, `Q3`, `Q4`

Data Inputs

- **D1** (ANY): value 1 to latch
- **D2** (ANY): value 2 to latch
- **D3** (ANY): value 3 to latch
- **D4** (ANY): value 4 to latch

Data Outputs

- **Q1** (ANY): latched value 1
- **Q2** (ANY): latched value 2
- **Q3** (ANY): latched value 3
- **Q4** (ANY): latched value 4

Adapters

No adapters are used.

Functionality

On every `CLK` event `D1`..`D4` are taken over and compared with the previously stored values `Q1`..`Q4`. If at least one `Di` differs from its `Qi`, the values are stored and `EO` is triggered; if all values are unchanged, no output event is produced. The block is a generic FB (`eclipse4diac::core::GenericClassName = GEN_E_D_FF_ANY`).

Technical Details

- **Generic block**: the types of `D1`..`D4` and `Q1`..`Q4` are `ANY` and are fixed at instantiation
- **Common change detection**: a single `EO` reports the change of any of the four values; which one changed cannot be seen from the event

Application Scenarios

- Adapter wrappers around functions that deliver four results, with output only on value change
- Change-driven further processing of several values of any data type

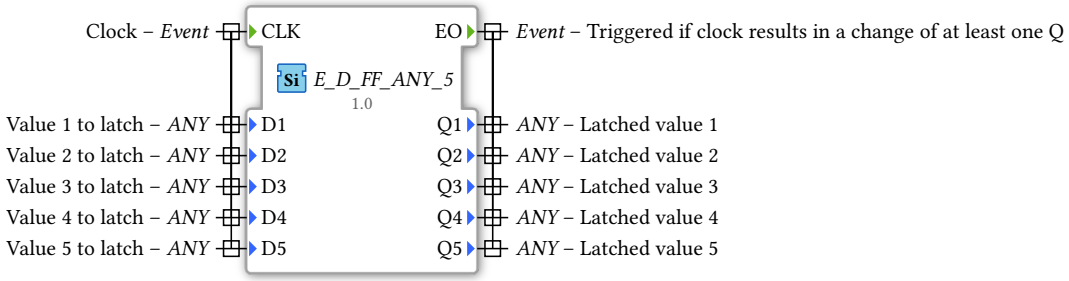
⚖️ Comparison with Similar Building Blocks

`E_D_FF_ANY` stores one value, `E_D_FF_ANY_2` two, `E_D_FF_ANY_3` three and `E_D_FF_ANY_4` four values.

Conclusion

`E_D_FF_ANY_4` is a generic, change-detecting four-value latch that only outputs `EO` if at least one of the four latched values changes.

2.11.3.20 E_D_FF_ANY_5



• – ◦ • – ◦ • – ◦ •

Introduction

`E_D_FF_ANY_5` is the variant of `E_D_FF_ANY` for **five** simultaneously latched values of any data type (`ANY`). It is a clock-driven data latch (D flip-flop) with built-in change detection: the output event `EO` is only triggered if at least one of the five outputs `Q1`..`Q5` changes due to the clock.

Interface Structure

Event Inputs

- **CLK**: clock, takes over the current values of `D1..D5`
  - **Associated data**: `D1`, `D2`, `D3`, `D4`, `D5`

Event Outputs

- **EO**: triggered if the clock causes a change of at least one `Qi`
  - **Associated data**: `Q1`, `Q2`, `Q3`, `Q4`, `Q5`

Data Inputs

- **D1** (ANY): value 1 to latch
- **D2** (ANY): value 2 to latch
- **D3** (ANY): value 3 to latch
- **D4** (ANY): value 4 to latch
- **D5** (ANY): value 5 to latch

Data Outputs

- **Q1** (ANY): latched value 1
- **Q2** (ANY): latched value 2
- **Q3** (ANY): latched value 3
- **Q4** (ANY): latched value 4
- **Q5** (ANY): latched value 5

Adapters

No adapters are used.

Functionality

On every `CLK` event `D1..D5` are taken over and compared with the previously stored values `Q1..Q5`. If at least one `Di` differs from its `Qi`, the values are stored and `EO` is triggered; if all values are unchanged, no output event is produced. The block is a generic FB (`eclipse4diac::core::GenericClassName = GEN_E_D_FF_ANY`); the native implementation `GEN_E_D_FF_ANY` (FORTE) is required.

Technical Details

- **Generic block**: the types of `D1..D5` and `Q1..Q5` are `ANY` and are fixed at instantiation
- **Common change detection**: a single `EO` reports the change of any of the five values; which one changed cannot be seen from the event
- Version 1.0 (2026-08-26): extension of `E_D_FF_ANY` to 5 variables

Application Scenarios

- Adapter wrappers around functions that deliver five results, with output only on value change
- Change-driven further processing of several values of any data type

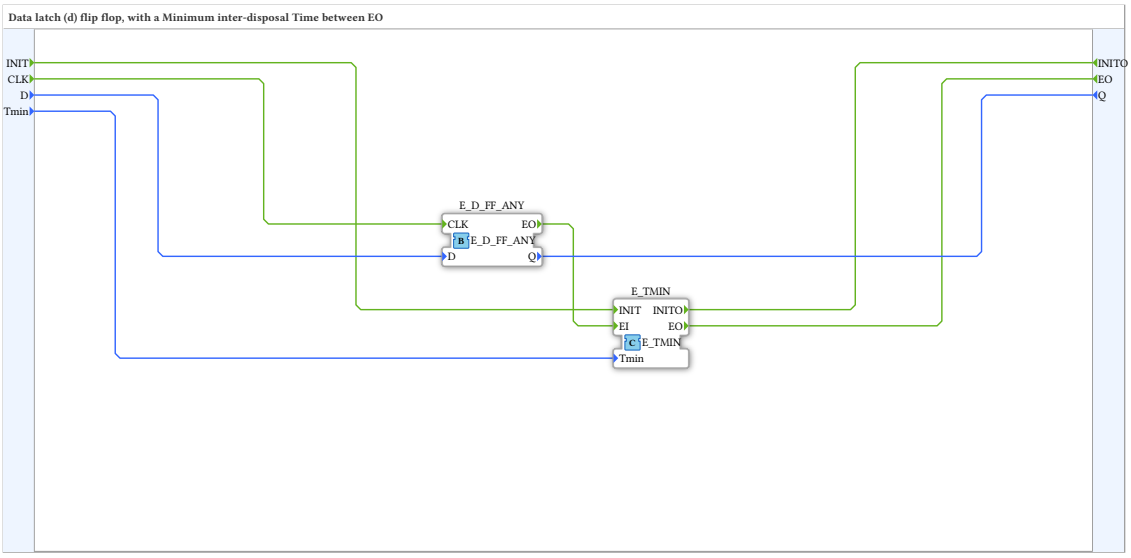
⚖️ Comparison with Similar Building Blocks

`E_D_FF_ANY` stores one value, `E_D_FF_ANY_2` two, `E_D_FF_ANY_3` three, `E_D_FF_ANY_4` four and `E_D_FF_ANY_5` five values.

Conclusion

`E_D_FF_ANY_5` is a generic, change-detecting five-value latch that only outputs `EO` if at least one of the five latched values changes.

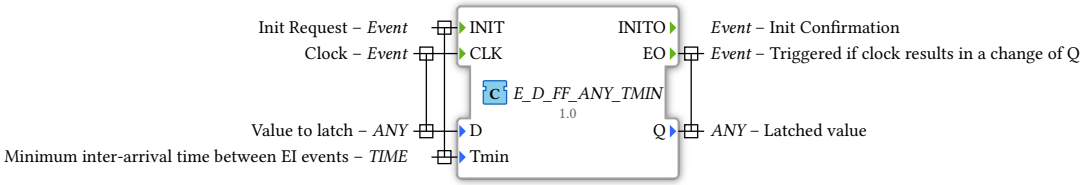
2.11.3.21 E_D_FF_ANY_TMIN



• — ○ • — ○ • — ○ •

Introduction

`E_D_FF_ANY_TMIN` is the generically-typed variant of E_D_FF_TMIN: it combines E_D_FF_ANY (change-detecting latch for arbitrary data types) with E_TMIN (minimum spacing time), making it suitable for any data type, not just `BOOL`.



Interface Structure

Event Inputs

- **INIT**: Initialization, carries `Tmin`.
- **CLK**: Clock event, latches `D` into `Q`.
  - **Connected data**: `D`

Event Outputs

- **INITO**: Confirms the initialization.
- **EO**: Change event throttled to `Tmin`.
  - **Connected data**: `Q`

Data Inputs

- **D** (ANY): The value to be latched, of arbitrary data type.
- **Tmin** (TIME): Minimum spacing between two `EO` events.

Data Outputs

- **Q** (ANY): The last latched value.

Functionality

Identical to E_D_FF_TMIN, but with E_D_FF_ANY instead of `E_D_FF`: `CLK` triggers the internal change detection (`NE(Q, D)`), whose `EO` is forwarded to the internal E_TMIN, which throttles it to at least `Tmin` between two triggers of `EO`.

Technical Features

- **ANY typing**: can be instantiated with any data type supported by the target environment (e.g. `TIME`, `REAL`, `DINT`).
- **Two-stage filtering**: content-based change detection (`E_D_FF_ANY`) plus time-based throttling (`E_TMIN`); `Q` is unaffected by this and always current.

State Overview

Stateless wiring logic; see E_D_FF_ANY (change detection) and E_TMIN (time throttling).

Application Scenarios

- **Low-noise forwarding of arbitrary measurements**: A `REAL` or `TIME` value fluctuates slightly, but `EO` should only trigger at most every `Tmin` — e.g. to limit the transmission rate to an upstream system.

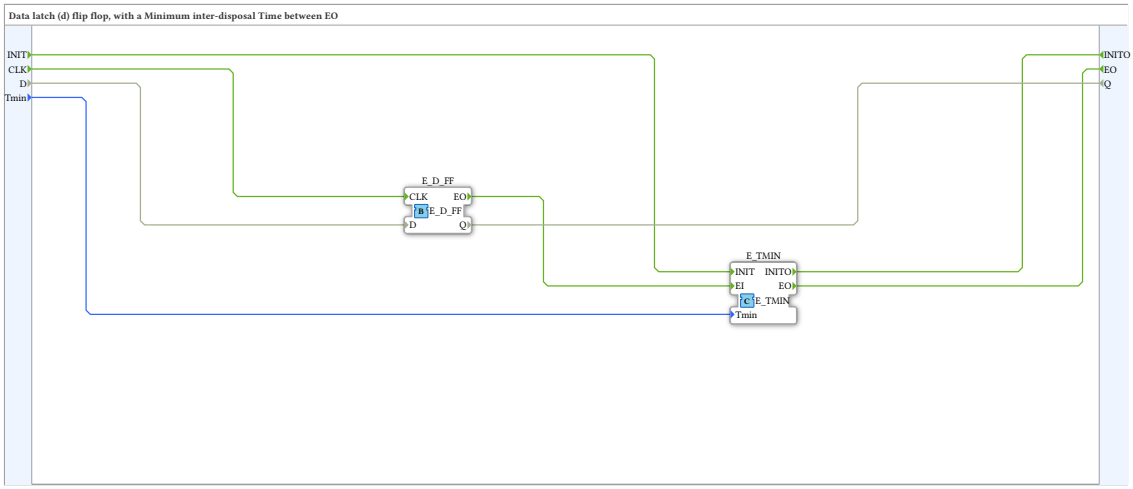
Comparison with similar function blocks

- **E_D_FF_ANY**: the same change detection, but without time throttling.
- **E_D_FF_TMIN**: the same functionality, but fixed to `BOOL`.
- **E_TMIN**: the internally used throttling logic.

Conclusion

`E_D_FF_ANY_TMIN` provides a generically-typed, change-detecting latch with a guaranteed minimum spacing between change events, for arbitrary data types.

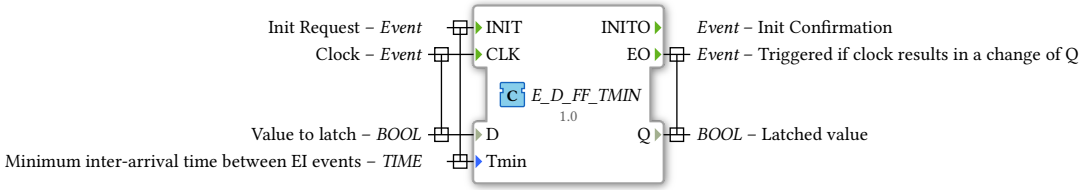
2.11.3.22 E_D_FF_TMIN



• – ○ • – ○ • – ○ •

Introduction

`E_D_FF_TMIN` combines `E_D_FF` (clocked `BOOL` latch with change detection) with E_TMIN (minimum spacing time): changes to `Q` are still latched immediately, but the change event `EO` is additionally limited to at most one trigger per `Tmin` interval.



Interface Structure

Event Inputs

- **INIT**: Initialization, carries `Tmin`.
- **CLK**: Clock event, latches `D` into `Q`.

  - **Connected data**: `D`

Event Outputs

- **INITO**: Confirms the initialization.
- **EO**: Change event throttled to `Tmin`.
  - **Connected data**: `Q`

Data Inputs

- **D** (BOOL): The value to be latched.
- **Tmin** (TIME): Minimum spacing between two `EO` events.

Data Outputs

- **Q** (BOOL): The last latched value.

Functionality

`CLK` internally triggers the encapsulated `E_D_FF`, whose `D`/`Q` data connection is forwarded directly to the outer `D`/`Q` ports. Only when `E_D_FF` detects an actual value change (`Q := D`) does it trigger its own `EO` — this is forwarded to the `EI` input of the internal E_TMIN, which only passes the event through to the outer `EO` output if at least `Tmin` has passed since the last pass-through.

Technical Features

- **Two-stage filtering**: first content-based change detection (`E_D_FF`), then time-based throttling (`E_TMIN`) — `Q` itself is not affected by this and is always current, even while `EO` is being throttled.
- **Pure wiring of two standard blocks**, no ECC of its own.

State Overview

Stateless wiring logic; see `E_D_FF` (change detection) and E_TMIN (time throttling) for the respective internal states.

Application Scenarios

- **Low-noise status reporting of a `BOOL` signal**: A digital state changes frequently over short periods (bouncing, noise); `EO` should only trigger at most every `Tmin`, while `Q` always shows the current value.

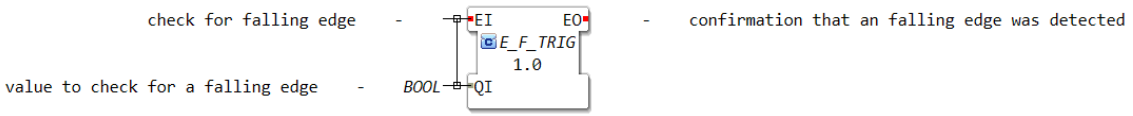
Comparison with similar function blocks

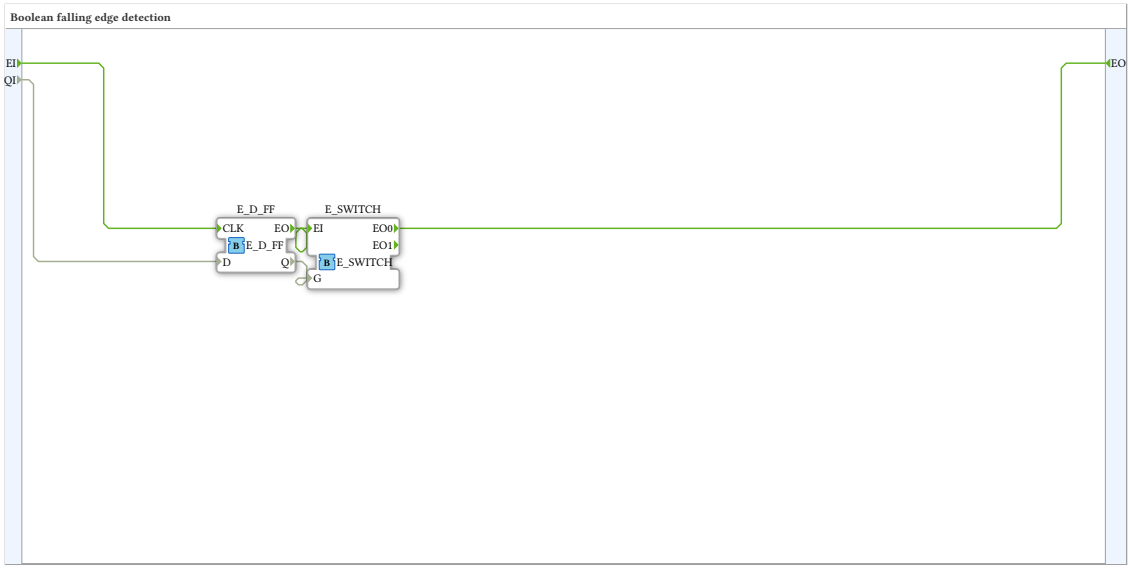
- **`E_D_FF`**: the same change detection, but without time throttling.
- **E_D_FF_ANY_TMIN**: the same functionality for arbitrary data types instead of just `BOOL`.
- **E_TMIN**: the internally used throttling logic.

Conclusion

`E_D_FF_TMIN` provides a change-detecting `BOOL` latch with a guaranteed minimum spacing between change events and is suitable for low-noise event forwarding of digital signals.

2.11.3.23 E_F_TRIG





• – ○ • – ○ • – ○ •

Introduction

The **E_F_TRIG** (Falling Edge Trigger) is a fundamental function block of the IEC 61499 standard for detecting falling signal edges. This block plays an important role in event detection and signal processing in industrial control systems.



Structure of the E_F_TRIG Block

Interface

Event Inputs:

- **EI (Event Input)**: Triggers the edge detection.
- **Associated Data**: `QI`

Event Outputs:

- **EO (Event Output)**: Triggered when an edge is detected.

Data Inputs:

- **QI (Qualifier Input)**: The Boolean signal to be monitored (data type: `BOOL`).

Functionality

The `E_F_TRIG` input is designed to detect a falling edge at the `QI` input. A falling edge occurs when `QI` was `TRUE` at the last `EI` event and is `FALSE` at the current `EI` event.

Each `EI` event compares the current state of `QI` with the state from the previous cycle. If a falling edge is detected, the `EO` event should be triggered.

Caution: Incorrect Implementation The internal logic of the function block (as of `events-3.0.0`) is incorrect and actually implements a **rising edge detector (`E_R_TRIG`)** instead of a falling one. The `EO` event is triggered when `QI` transitions from `FALSE` to `TRUE`.

Caution: Incorrect Implementation The internal logic of the function block (as of `events-3.0.0`) is incorrect and actually implements a **rising edge detector (`E_R_TRIG`)** instead of a falling one.** The `EO` event is triggered when `QI` transitions from `FALSE` to `TRUE`.**

**

Technical Features

✓ **Event-driven**: The check only occurs when a `EI` event occurs.
✓ **Faulty logic**: The function block is named `E_F_TRIG`, but its internal wiring of `E_D_FF` and `E_SWITCH` causes it to react to a **rising edge**. Users who require falling edge detection must instead use the `E_R_TRIG` function block and invert its input or wait for a correction of the function block.

Application Scenarios (for a falling edge)

- **Sensor data evaluation**: Detect when a sensor changes from "active" to "inactive".
- **Machine Safety**: Detection when a protective contact changes from "closed" (TRUE) to "open" (FALSE).
- **Process Monitoring**: Detection of the end of a signal or process.

⚖️ **Comparison with similar function blocks**

Feature	E_F_TRIG (this)	E_R_TRIG	E_SWITCH
Detected Edge	Falling (named), Rising (implemented)	Rising	-
Event Control	Yes	Yes	Yes
Principle	Edge Detector	Edge Detector	Event Switch

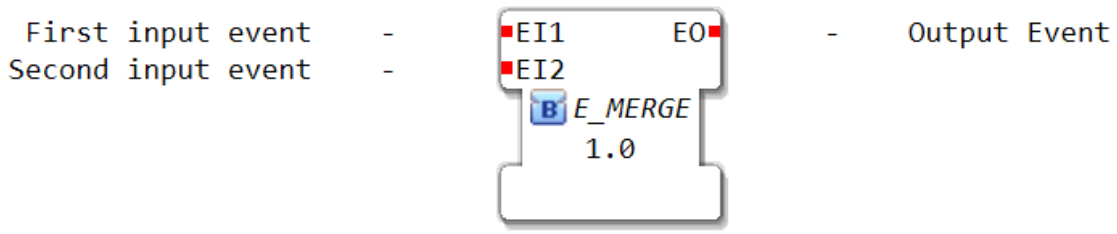
🛠️ **Related Exercises**

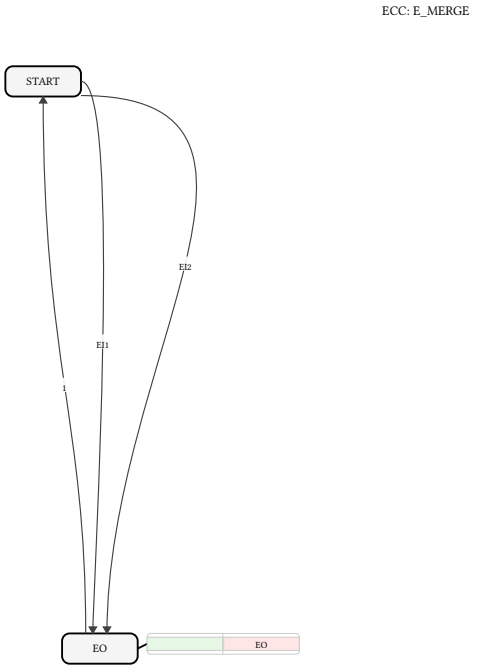
- Exercise_088

Conclusion

The E_F_TRIG function block is designed to detect falling edges, a common use case in control engineering. **However, due to an implementation error in the `events-3.0.0` library, this function block also functions as a detector for rising edges.** This discrepancy between name and function is critical and must be carefully considered during use. For reliable detection of falling edges, alternative logic should be used.

2.11.3.24 E_MERGE

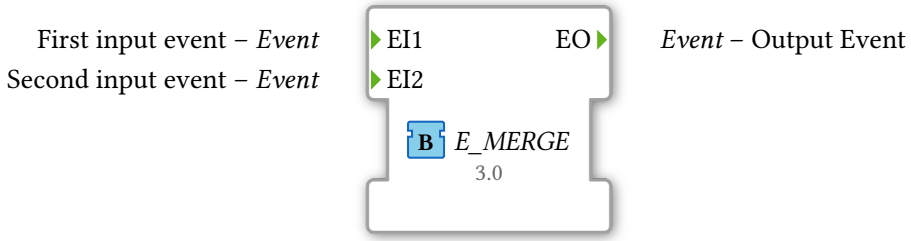




• – ◦ • – ◦ • – ◦ •

Introduction

The **E_MERGE** is a fundamental function block of the IEC 61499 standard that combines multiple event streams into a single output. This logical OR operation of events is essential for control logic in industrial automation systems.



Structure of the E_MERGE Block

Interface

Event Inputs:

- `EI1` (Event Input 1): First event input
- `EI2` (Event Input 2): Second event input

Event Outputs:

- `EO` (Event Output): Merged event output

Functionality

1. **Event Merging:**

- Each event at `EI1` or `EI2` triggers an output event at `EO`
- The inputs are logically ORed

1. **Independent Processing:**

- Events at both inputs are Equal treatment

- No prioritization of specific inputs

1. **Immediate forwarding:**

- No delay between input and output events
- No memory behavior or state maintenance

Technical Features

✓ **Simple and fast** event linking ✓ **Lossless** event forwarding ✓ **Real-time capable** for industrial applications ✓ **Expandable** to multiple inputs

Application Scenarios

- **Operating concepts**: Combining control signals from multiple buttons
- **Sensor data**: Combining events from different sensors
- **Fault management**: Central location for various fault messages
- **Process control**: Linking process events

⚖️ Comparison with similar function blocks

Feature	E_MERGE	E_DEMUX	E_SWITCH
Functional principle	OR operation	Distribution	Conditional forwarding
Direction	n:1	1:n	1:1
Event flow	Combination	Splitting	Selection

Similar Building Blocks

For use cases requiring more than two event inputs, the library provides additional variants:

- **E_MERGE**: This building block (2 inputs)
- **E_MERGE_2**: Functionally identical to `E_MERGE` (2 inputs)
- **E_MERGE_3**: A variant with 3 inputs (`EI1`, `EI2`, `EI3`)
- **E_MERGE_4**: A variant with 4 inputs

These building blocks allow for the easy merging of up to four event sources into a single output.

🛠️ Related Exercises

- Exercise_004a2
- Exercise_004a2_AX

Conclusion

The E_MERGE block is a fundamental building block for event processing in IEC 61499 systems. Its main advantages are:

- Simple and efficient event combination
- Immediate response to input events
- Flexible application possibilities in various control scenarios

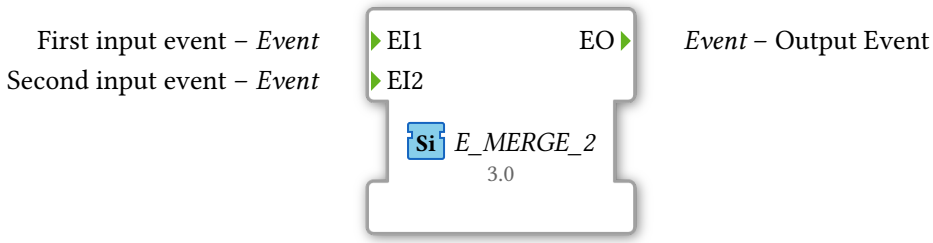
Due to its clear functionality and standard compliance, it is ideally suited for basic logic tasks in industrial automation solutions. Its deterministic operation makes it particularly valuable for safety-critical applications.

2.11.3.25 E_MERGE_2

• – ◦ • – ◦ • – ◦ •

Introduction

The `E_MERGE_2` is a fundamental function block according to the IEC 61499 standard. It combines two event streams (`EI1`, `EI2`) into a single output (`EO`). This logical OR operation of events is essential for control logic in industrial automation systems. This block is functionally identical to the `E_MERGE` block.



Interface Structure

Event Inputs:

- `EI1` (Event Input 1): First event input
- `EI2` (Event Input 2): Second event input

Event Outputs:

- `EO` (Event Output): Combined event output

Functionality

The function is simple and direct:

- An event at input `EI1` immediately triggers an event at output `EO`.
- An event at input `EI2` also immediately triggers an event at output `EO`.

The inputs are logically ORed and forwarded without prioritization or delay.

Technical Features

- **2-to-1 Linking**: Combines two event streams into one.
- **Stateless**: The function block has no internal memory.
- **Generic Function Block**: The functionality is provided by the generic class `GEN_E_MERGE`, enabling optimized execution by the 4diac runtime environment.

Application Scenarios

- **Operating Concepts**: Combining control signals from two pushbuttons into a single function.
- **Sensors**: Combining trigger events from two different sensors.
- **Fault Management**: Creating a collective fault message from two potential fault sources.

🛠️ Related Exercises

- Exercise_004a2_2

Conclusion

The `E_MERGE_2` is a fundamental and frequently used building block for the simplest form of event linking. Its clear functionality and efficient implementation as a generic building block make it a reliable component for basic control logic.

🌐 Related Topic Subpages on ms-muc-docs.de

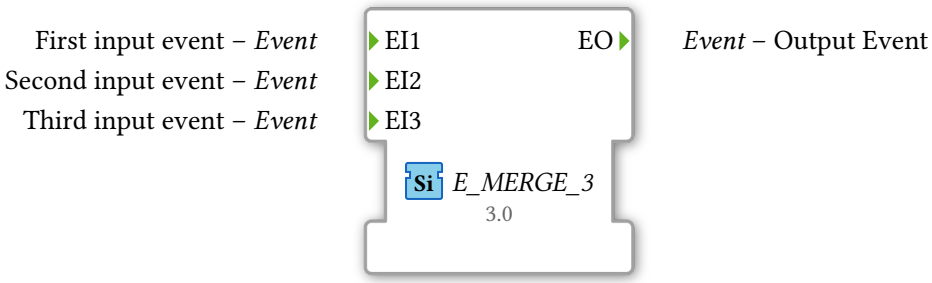
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.3.26 E_MERGE_3

• – ◦ • – ◦ • – ◦ •

Introduction

The `E_MERGE_3` is a fundamental function block of the IEC 61499 standard that combines three event streams (`EI1`, `EI2`, `EI3`) into a single output (`EO`). This logical OR operation of events is essential for control logic in industrial automation systems.



Interface Structure

Event Inputs:

- `EI1` (Event Input 1): First event input
- `EI2` (Event Input 2): Second event input
- `EI3` (Event Input 3): Third event input

Event Outputs:

- `EO` (Event Output): Combined event output

Functionality

The function is simple and direct:

- An event at input `EI1`, `EI2`, or `EI3` immediately triggers an event at output `EO` without prioritization.

The inputs are logically ORed and forwarded without delay.

Technical Features

- **3-to-1 Linking**: Combines three event streams into one.
- **Stateless**: The function block has no internal memory.
- **Generic Function Block**: The functionality is provided by the generic class `GEN_E_MERGE`, enabling optimized execution by the 4diac runtime environment.

Application Scenarios

- **Operating Concepts**: Combining control signals from three buttons into a common function (e.g., "Manual," "Automatic," and "Service" start the same process).
- **Sensors**: Combining the trigger events from three different sensors into a single event.
- **Fault Management**: Creating a collective fault message from three potential fault sources.

🛠️ Related Exercises

- Exercise_004a2_3

Conclusion

The `E_MERGE_3` is a simple yet useful building block for linking up to three event sources. Its clear functionality and efficient implementation as a generic building block make it a reliable component for basic control logic.

🌐 Related Topic Subpages on ms-muc-docs.de

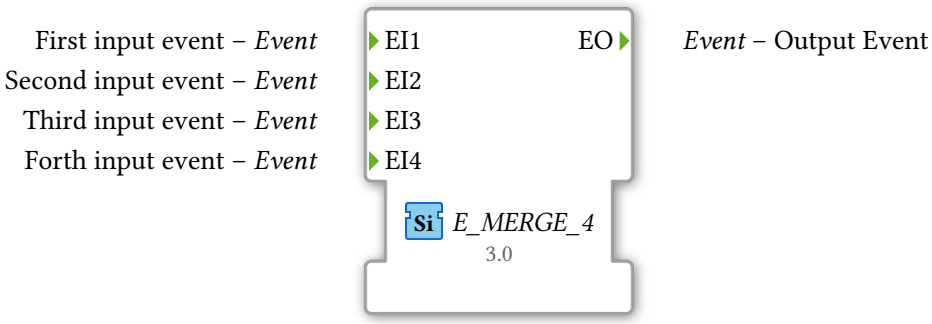
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.3.27 E_MERGE_4

• – ◦ • – ◦ • – ◦ •

Introduction

The `E_MERGE_4` is a fundamental function block of the IEC 61499 standard that combines four event streams (`EI1` to `EI4`) into a single output (`EO`). This logical OR operation of events is essential for control logic in industrial automation systems.



Interface Structure

Event Inputs:

- `EI1` (Event Input 1): First Event Input
- `EI2` (Event Input 2): Second Event Input
- `EI3` (Event Input 3): Third Event Input
- `EI4` (Event Input 4): Fourth Event Input

Event Outputs:

- `EO` (Event Output): Combined Event Output

Functionality

The function is simple and direct:

- An event at one of the four inputs (`EI1` to `EI4`) immediately triggers an event at the output without prioritization. `EO`.

The inputs are logically ORed and forwarded without delay.

Technical Features

- **4-to-1 Linking**: Combines four event streams into one.
- **Stateless**: The block has no internal memory.
- **Generic Block**: The functionality is provided by the generic class `GEN_E_MERGE`, enabling optimized execution by the 4diac runtime environment.

Application Scenarios

- **Collective Fault Message**: Combines four different fault events into a single alarm message.
- **Operation**: A process step can be triggered by four different buttons or sources.
- **Sensor Array**: A single “object detected” event is triggered when one of four sensors detects an object.

Conclusion

The `E_MERGE_4` is a simple yet useful building block for linking up to four event sources. Its clear functionality and efficient implementation as a generic building block make it a reliable component for basic control logic.

🌐 Related topic subpages on ms-muc-docs.de

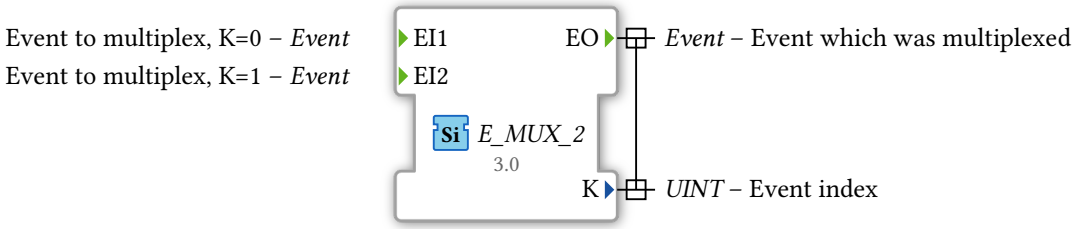
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.11.3.28 E_MUX_2

• – ○ • – ○ • – ○ •

Introduction

The `E_MUX_2` (Event Multiplexer) is a function block according to IEC 61499 that combines two event inputs (`EI1`, `EI2`) into a single event output (`EO`). In addition to combining the inputs, the block outputs a data output, `K`, indicating which input triggered the event. It is the counterpart to the `E_DEMUX_2`.



Interface Structure

Event Inputs:

- **EI1**: First input channel. - **EI2**: Second input channel.

Event Outputs:

- **EO (Event Output)**: The common event output.
- **Associated Data**: `K`

Data Outputs:

- **K**: The index of the triggering input channel (data type: `UINT`).
- `K = 0` if `EI1` was triggered.
- `K = 1` if `EI2` was triggered.

Functionality

1. **Event Receipt**: The function block waits for an event at one of its inputs (`EI1` or `EI2`).
2. **Forwarding and Identification**:

- When an event arrives at `EI1`, the data output `K` is set to `0`, and the `EO` event is immediately triggered.
- When an event arrives at `EI2`, the data output `K` is set to `1`, and the `EO` event is immediately triggered.

In this way, the event flow is merged while preserving information about the event's origin.

Technical Features

- **2-to-1 Multiplexer**: Merges two event streams into one.
- **Origin Index**: Indicates which input triggered the event.
- **Stateless**: The function block has no internal memory.
- **Generic Function Block**: The functionality is provided by the generic class `GEN_E_MUX`.

Application Scenarios

- **Command Combining**: Two different buttons (e.g., "Start Forward" and "Start Reverse") trigger the same process, but the process needs to know which button was pressed.
- **Combined Alarms**: Two different alarms are routed to a central alarm routine, which then displays the specific alarm text based on `K`.
- **Simplified Logic**: Instead of having two separate paths for similar logic, the triggering events are merged, and the index `K` is used as a parameter for subsequent logic.

🛠️ **Related Exercises**

- Exercise_172

Conclusion

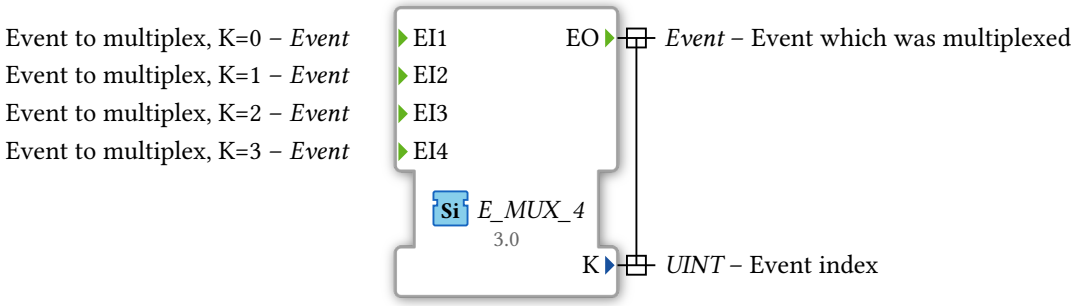
The `E_MUX_2` is a useful component for bundling event streams while simultaneously identifying the event source. It is the standard counterpart to `E_DEMUX_2` and is frequently used to reduce wiring complexity and centralize logic.

2.11.3.29 E_MUX_4

• – ◦ • – ◦ • – ◦ •

Introduction

The `E_MUX_4` (Event Multiplexer) is a function block according to IEC 61499 that combines four event inputs (`EI1` to `EI4`) into a single event output (`EO`). In addition to combining the inputs, the block outputs a data output, `K`, indicating which input triggered the event. It is the counterpart to the `E_DEMUX_4`.



Interface Structure

Event Inputs:

- **EI1**: First input channel. - **EI2**: Second input channel.
- **EI3**: Third input channel.
- **EI4**: Fourth input channel.

Event Outputs:

- **EO (Event Output)**: The common event output.
- **Associated Data**: `K`

Data Outputs:

- **K**: The index of the triggering input channel (data type: `UINT`).
- `K = 0` if `EI1` was triggered.
- `K = 1` if `EI2` was triggered.
- `K = 2` when `EI3` is triggered.
- `K = 3` when `EI4` is triggered.

Functionality

1. **Event Receipt**: The function block waits for an event at one of its four inputs.
2. **Forwarding and Identification**: When an event arrives at `EIn` (where `n` represents 1 to 4), the data output `K` is set to `n-1`, and the `EO` event is immediately triggered.

In this way, the event flow is merged while preserving information about the event's origin.

Technical Features

- **4-to-1 Multiplexer**: Merges four event streams into one.
- **Origin Index**: Indicates which input triggered the event.
- **Stateless**: The function block has no internal memory.
- **Generic Function Block**: The functionality is provided by the generic class `GEN_E_MUX`.

Application Scenarios

- **Command Grouping**: Four different buttons trigger the same process, but the process needs to know which of the four buttons was pressed.
- **Combined Alarms**: Four different alarms are routed to a central alarm routine, which then displays the specific alarm text based on `K`.
- **Mode Selection**: Combines the startup events from four different operating modes.

🛠️ Related Exercises

- Exercise_087a2

Conclusion

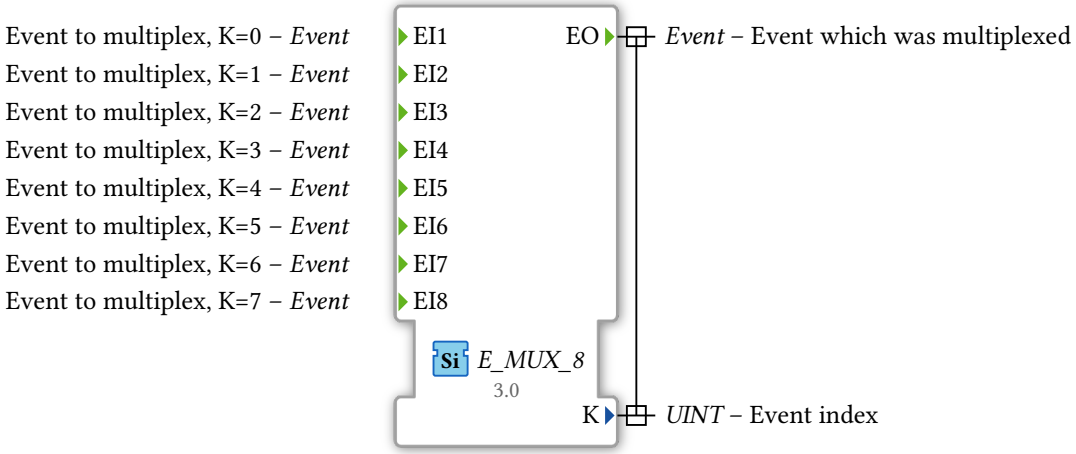
The `E_MUX_4` is a useful component for bundling event streams from up to four sources while simultaneously identifying the event's source. It is the standard counterpart to the `E_DEMUX_4` and is frequently used to reduce wiring complexity and centralize logic.

2.11.3.30 E_MUX_8

• − ◦ • − ◦ • − ◦ •

Introduction

The `E_MUX_8` (Event Multiplexer) is a function block according to IEC 61499 that combines eight event inputs (`EI1` to `EI8`) into a single event output (`EO`). In addition to combining the inputs, the block outputs a data output, `K`, indicating which input triggered the event. It is the counterpart to the `E_DEMUX_8`.



Interface Structure

Event Inputs:

- **EI1 ... EI8**: The eight separate input channels.

Event Outputs:

- **EO (Event Output)**: The common event output.
- **Associated Data**: `K`

Data Outputs:

- **K**: The index of the triggering input channel (data type: `UINT`).
- `K = 0` if `EI1` was triggered.
- `K = 1` if `EI2` was triggered.
- ...
- `K = 7` when `EI8` is triggered.

Functionality

1. **Event Reception**: The function block waits for an event at one of its eight inputs.
2. **Forwarding and Identification**: When an event arrives at `EIn` (where `n` represents 1 to 8), the data output `K` is set to `n-1`, and the `EO` event is immediately triggered.

In this way, the event streams are merged while preserving information about the event's origin.

Technical Features

- **8-to-1 Multiplexer**: Combines eight event streams into one.
- **Origin Index**: Indicates which input triggered the event.
- **Stateless**: The function block has no internal memory.
- **Generic Function Block**: The functionality is provided by the generic class `GEN_E_MUX`.

Application Scenarios

- **Keyboard Matrix**: Groups the signals from eight keys into a central evaluation logic.
- **Comprehensive Alarming**: Groups eight different alarms into a central routine, which then processes the specific alarm message based on `K`.
- **Prioritized Command Selection**: Eight command sources are combined, and downstream logic determines the priority based on the index `K`.

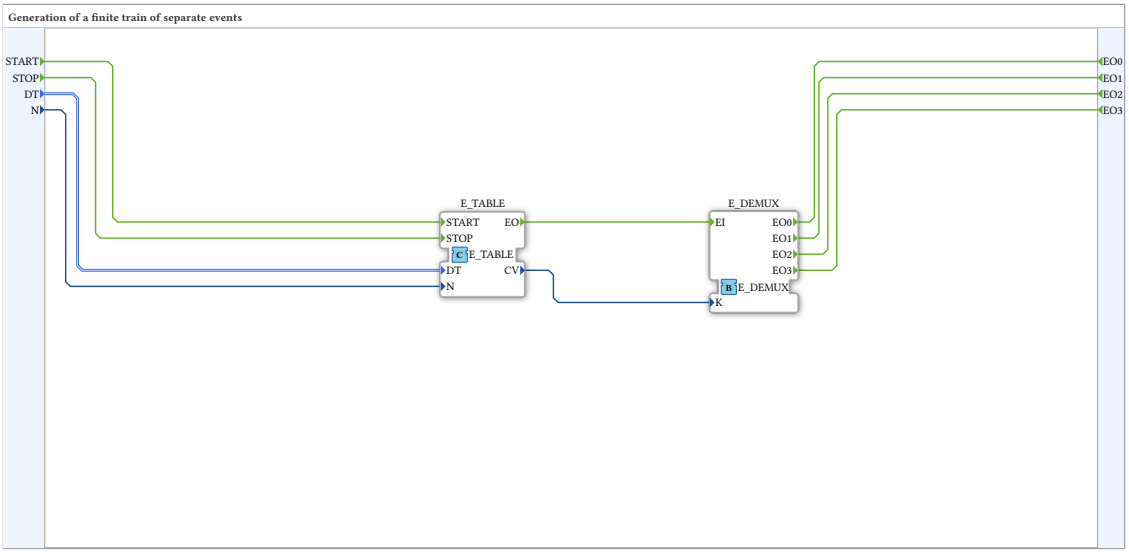
🛠️ Related Exercises

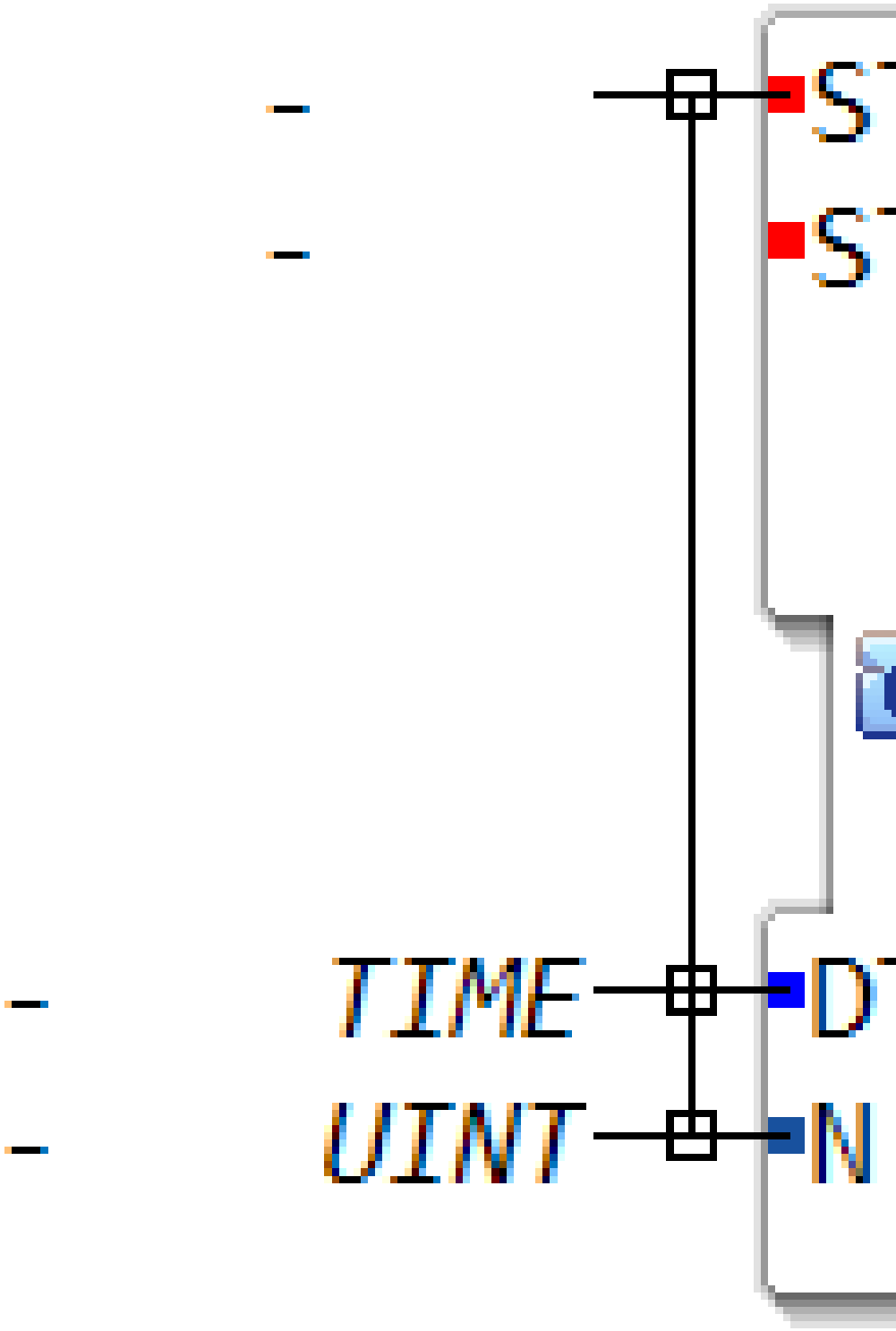
- Exercise_173

Conclusion

The `E_MUX_8` is a useful component for bundling event streams from up to eight sources while simultaneously identifying the event's source. It is the standard counterpart to the `E_DEMUX_8` and is frequently used to reduce wiring complexity and centralize logic.

2.11.3.31 E_N_TABLE

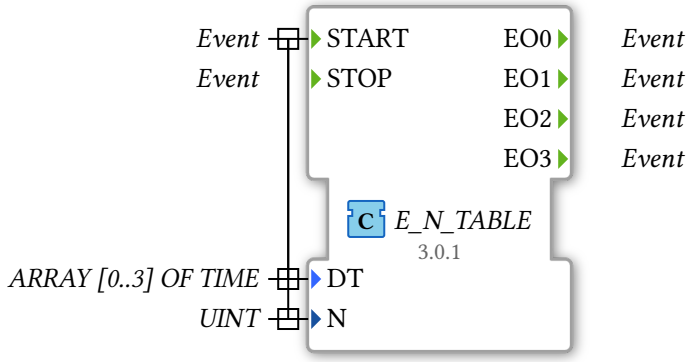




• – ◦ • – ◦ • – ◦ •

Introduction

The `E_N_TABLE` (Event N-Table) is a function block according to IEC 61499 that generates a finite sequence of time-staggered events on separate outputs. Internally, it combines a table-controlled timer (`E_TABLE`) with a demultiplexer (`E_DEMUX`) to create a chain of events, with each event output on its own channel.



Interface Structure

Event Inputs

- **START**: Starts the generation of the event sequence.
- **Related Data**: `DT`, `N`
- **STOP**: Stops the sequence prematurely.

Event Outputs

- **EO0**: Outputs the first event of the sequence (index 0).
- **EO1**: Outputs the second event of the sequence (index 1).
- **EO2**: Outputs the third event of the sequence (index 2).
- **EO3**: Outputs the fourth event of the sequence (index 3).

Data Inputs

- **DT**: An array of durations (data type: `TIME`, size: 4). `DT[i]` defines the delay time to wait *before* the `i`-th event.
- **N**: The total number of events to be generated (data type: `UINT`, max. 4 for this function block).

Functionality

1. **Start of sequence**: A `START` event triggers the internal `E_TABLE` function block. The number of events to be generated is determined by `N`.
2. **First event**: The function block waits for the time interval defined in `DT[0]`. Then, the first event is triggered at output `EO0`.
3. **Subsequent Events**: The function block waits for the time interval defined in `DT[1]` and then triggers the second event at `EO1`. This process repeats for the next events according to the durations in `DT[2]`, `DT[3]`, and so on, until `N` events have been generated.
4. **End of Sequence**: The sequence ends automatically after `N` events have been triggered.
5. **Stop**: A `STOP` event immediately terminates the sequence at any point.

Example:

- `N = 3`
- `DT = [T#2s, T#5s, T#1s]`
- After a `START` event:

1. After 2 seconds, `EO0` is triggered.
2. 5 seconds later, `EO1` is triggered.
3. 1 second later, `EO2` is triggered.
4. The sequence is complete. `EO3` is not triggered.

Technical Features

- **Sequence Generator**: Generates a chain of individual, time-staggered events.
- **Demultiplexing**: Each event in the chain is routed to its own dedicated output.
- **Table-driven**: The time intervals between events are not fixed but are read from the `DT` array.

Application Scenarios

- **Control of step sequences**: Each output (`EO0`, `EO1`, ...) can trigger a subsequent step in a process chain.

- **Complex control**: Controlling various actuators in a precisely defined temporal sequence (e.g., valves in a rinsing process).
- **Test automation**: Generating a complex, time-defined stimulus sequence for a test object.

🛠️ Related Exercises

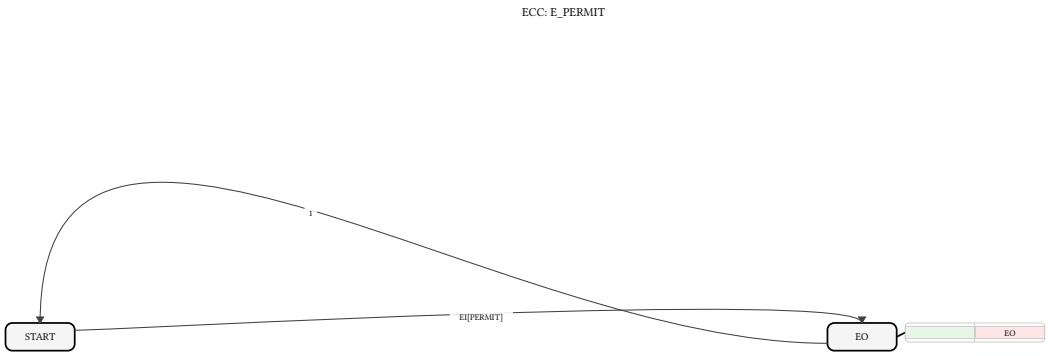
- Exercise_093b

Conclusion

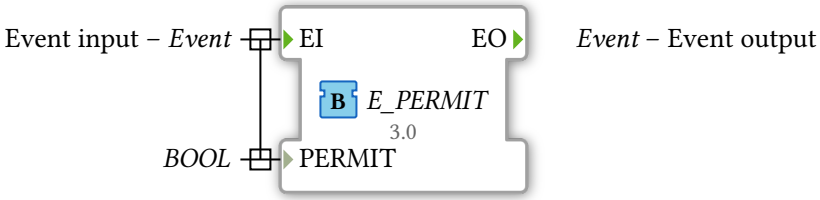
The `E_N_TABLE` is a powerful building block for generating complex, time-defined event sequences. Unlike a simple `E_CYCLE`, which only generates a periodic clock signal, `E_N_TABLE` allows the definition of variable time intervals and the distribution of individual sequence events across separate channels. This makes it ideal for controlling sequential processes.

2.11.3.32 E_PERMIT

🎧 Podcast



- The E_PERMIT block: The “gatekeeper” for events in IEC 61499 systems decoded
- Decoding E_PERMIT: The Unsung Hero of Industrial Automation’s Safety and Reliability The E_PERMIT (Event Permit) is a fundamental function block according to IEC 61499 that acts as a controllable “gate” for events. It allows an input event to pass to the output only if an explicit enable condition is met.



- **EI (Event Input)**: The input event to be checked.
- **Associated Data**: `PERMIT`
- **EO (Event Output)**: The event output, which is only triggered if permission has been granted.
- **PERMIT**: The permission condition (data type: `BOOL`).

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

1. **Event Reception**: The function block waits for an event at input `EI`.
2. **Conditional Forwarding**:

- **If `PERMIT` = `TRUE`**: The event is allowed through and output at `EO`.
- **If `PERMIT` = `FALSE`**: The event is blocked, and nothing happens. Output `EO` is not triggered.

This function block thus acts as a simple monitor for the event flow.

- **Event Gate**: Serves as a fundamental "gate" for controlling event flows.
- **Stateless**: The function block itself has no internal memory; its decision is based solely on the value of `PERMIT` at the moment of the `EI` event.
- **Allowances/Interlocks**: A process step (`EI`) may only be started if a safety allow (`PERMIT`) is present (e.g., safety door closed).
- **Operating Mode Switching**: Commands from a manual controller (`EI`) are only forwarded if the system is in "Manual" mode (`PERMIT` = true).
- **Data Validation**: An event that triggers further data processing is only triggered if a previous data validation was successful (`PERMIT` = true).
- **`E_SWITCH`**: While `E_PERMIT` either passes or blocks an event (1-to-1 or 1-to-0), `E_SWITCH` forwards an event to one of two different outputs (1-to-2). `E_PERMIT` is a gate, and `E_SWITCH` is a switch.
- Exercise_009
- Exercise_080c
- Exercise_094

The `E_PERMIT` block is a fundamental and widely used block for implementing conditions and enable statements in the event-driven logic of IEC 61499. Its simplicity and clear function make it an indispensable tool for creating safe and robust control sequences.

Technical Features

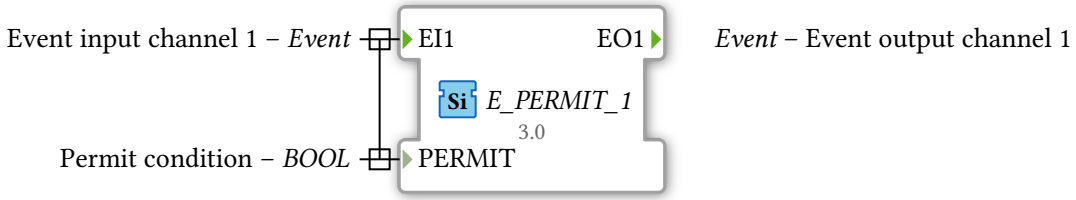
Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

🛠️ Zugehörige Übungen

Conclusion

2.11.3.33 E_PERMIT_1



• – ◦ • – ◦ • – ◦ •

Introduction

The `E_PERMIT_1` is a function block for the **permissive propagation of one event channel** (generic block). The event `EI1` is forwarded to `EO1` only if the permit condition `PERMIT` is fulfilled. It is the one-channel variant of the generic `E_PERMIT` family.

Interface Structure

Event Inputs

- **EI1**: Event input, channel 1. Associated data: `PERMIT`

Event Outputs

- **EO1**: Event output, channel 1

Data Inputs

- **PERMIT** (`BOOL`): Permit condition

Data Outputs

No data outputs

Adapters

This function block uses no adapters.

How it works

On every `EI1` event, `PERMIT` is evaluated. If `PERMIT = TRUE`, `EO1` is fired; otherwise the event is suppressed.

Technical Details

- Generic block that requires the native implementation `GEN_E_PERMIT` (FORTE).
- Pure event filter: no data is stored or output.
- `PERMIT` is associated with `EI1` and is therefore read anew on every event.
- See also `E_PERMIT` (standard variant).

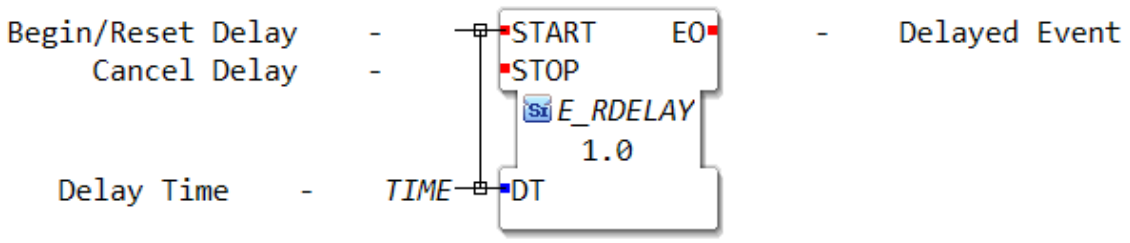
Application Scenarios

- Forwarding events only while a permit is active, e.g. key events only in automatic mode
- Blocking event paths by a status signal

Summary

`E_PERMIT_1` forwards a single event only if the `PERMIT` condition is fulfilled.

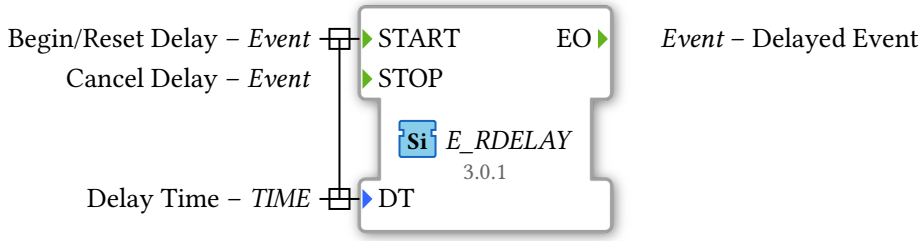
2.11.3.34 E_RDELAY



• – ◦ • – ◦ • – ◦ •

Introduction

The **E_RDELAY** (Reloadable Delay) is an extended delay function block according to IEC 61499, which, unlike the simple E_DELAY, enables a reloadable and abortable event delay. Developed under the EPL-2.0 license.



Interface Structure

Event Inputs

- `START`: Starts/Resets the delay (with DT parameter)
- `STOP`: Cancels the active delay

Event Outputs

- `EO`: Delayed output event

Data Inputs

- `DT` (Delay Time): Delay duration (TIME data type)

Operating Principle

1. **Delay Start**:

- On the `START` event, the timer starts with the configured `DT` time
- New `START` resets the timer during an active delay

1. **Delay Termination**:

- `STOP` immediately terminates the active delay
- No `EO` event is generated

1. **Delay Completion**:

- `EO` is triggered once after exactly `DT`
- Immediate triggering occurs if DT ≤ T#0s

Service Sequences (according to the XML specification)

1. **event_delay**:

- Normal delay with START ⟶ EO

1. **delay_canceled**:

- START followed by STOP (no EO)

1. **reload_delay**:

- Multiple START events reload (reset) the delay; only one EO is triggered

Technical Special Features

✓ **Reloadable Delay** (Reset Function) ✓ **Cancellable** during runtime ✓ **Deterministic Timing Behavior** ✓ **Real-Time Capable** Implementation

Application Scenarios

- **Machine Safety**: Delayed Shutdown Sequences
- **Process Control**: Time-Controlled State Transitions
- **Alarm Systems**: Fault Message Delay
- **Robot Control**: Motion Sequence Timing

⚖️ **Comparison with E_DELAY**

Feature	E_RDELAY	E_DELAY

Reset Function	✓ (via START)	❌
Multiple Triggers	Only 1 EO	Only 1 EO
Service Sequences	3 defined (`event_delay`, `delay_canceled`, `reload_delay`)	3 defined (`event_delay`, `delay_canceled`, `no_multiple_delay`)

🛠️ Related Exercises

- Exercise_018a

Conclusion

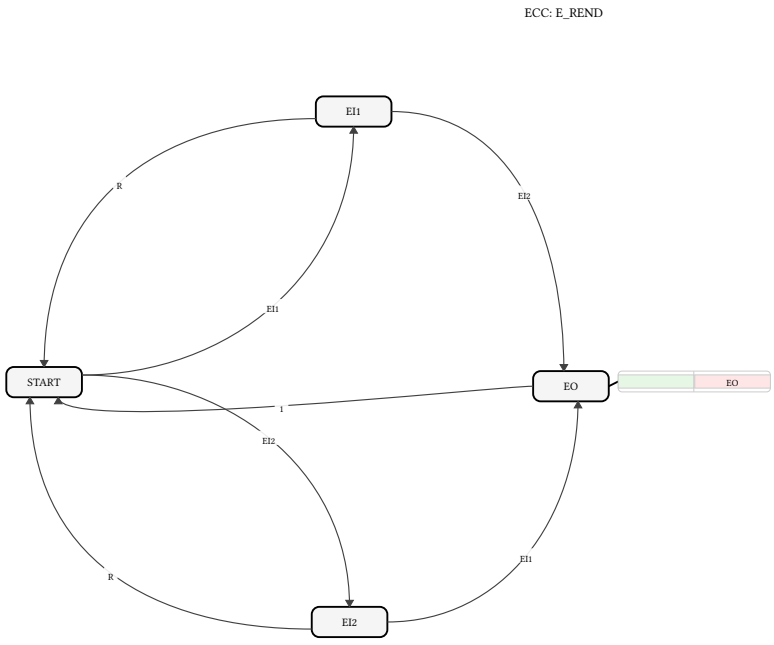
The E_RDELAY function block extends the classic delay function with essential, industry-ready features:

- Flexible reconfiguration during operation
- Safe termination of time sequences
- Standardized service interfaces

Thanks to its XML-based specification, the function block is particularly well-suited for use in distributed control systems. The implemented version offers reliable timing control for critical automation processes.

2.11.3.35 E_REND

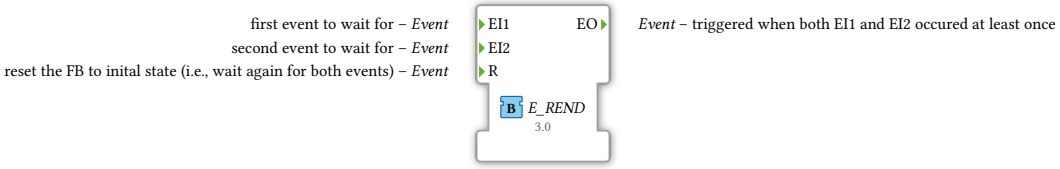
🎧 Podcast



- E_REND: Event Synchronization in IEC 61499
- E_REND: Event Synchronization in IEC 61499

Introduction

The `E_REND` (Event Rendezvous) is a function block according to IEC 61499 that serves as a synchronization point for two different event streams. It only fires an output event when it has received at least one event from each of its inputs. This corresponds to a logical AND operation over time.



Interface Structure

Event Inputs:

- **EI1**: The first event input.
- **EI2**: The second event input.
- **R (Reset)**: Resets the function block to its initial state.

Event Outputs:

- **EO (Event Output)**: Triggered after both `EI1` and `EI2` have been received at least once since the last reset.

Functionality

1. **Waiting for Events**: In its initial state, the function block waits for events at `EI1` and `EI2`.
2. **Store the First Event**: When an event arrives at the first input (e.g., `EI1`), the function block stores this internally and then waits for the event at the second input (`EI2`).
3. **Rendezvous and Triggering**: As soon as the second event (`EI2`) arrives, the "rendezvous" condition is met. The function block immediately triggers the output event `EO`.
4. **Automatic Reset**: Immediately after `EO` is triggered, the function block automatically resets to its initial state and waits for the next pair of events, `EI1` and `EI2`.
5. **External Reset**: An event at the `R` input resets the function block to its initial state at any time, discarding all previously recorded but incomplete event pairs.

Important: The input events `EI1` and `EI2` do **not** have to arrive simultaneously. The order in which they arrive is also irrelevant.

Technical Features

- **Static**: Unlike a `E_MERGE` (OR), `E_REND` has an internal state to remember the arrival of the first event.
- **Synchronization**: Used to synchronize two asynchronous event flows.

Application Scenarios

- **Process Synchronization**: A subsequent process step (`EO`) may only start when two independent preconditions are met (e.g., "Component clamped" (`EI1`) and "Safety door closed" (`EI2`)).
- **Acknowledgement**: An action is only executed when both a command (`EI1`) and manual acknowledgement by an operator (`EI2`) are present.
- **Material Flow**: A conveyor belt stops (`EO`) when both the front and rear sensors have detected a long component.

🛠️ Related Exercises

- Exercise_004a6
- Exercise_004a6_AX
- Exercise_004a7
- Exercise_004a7_AX
- Exercise_025
- Exercise_026

Conclusion

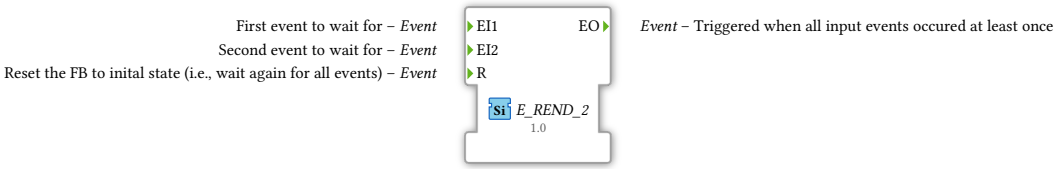
The `E_REND` block is a critical tool for process synchronization in IEC 61499. It provides a robust method to ensure that multiple conditions are met before a subsequent action is triggered. Its ability to "remember" the arrival of events makes it significantly more powerful than simple logical connections and is fundamental for controlling asynchronous and parallel processes.

2.11.3.36 E_REND_2

• – ◦ • – ◦ • – ◦ •

Introduction

`E_REND_2` is the generic 2-input variant of the rendezvous block `E_REND` (2 inputs). It only triggers `EO` once it has received at least one event since the last reset from **all** 2 inputs (`EI1`, `EI2`) — a logical AND across 2 asynchronous event streams.



Interface Structure

Event Inputs

- **EI1**: Event input 1.
- **EI2**: Event input 2.
- **R (Reset)**: Resets the block to its initial state, discarding all events recorded so far.

Event Outputs

- **EO**: Triggered once all 2 inputs have fired at least once since the last reset.

Functionality

`E_REND_2` is implemented as a generic function block (`eclipse4diac::core::GenericClassName = 'GEN_E_REND'`) — the same C++ implementation underlies all `E_REND_n` variants; only the number of inputs differs. Internally, the block remembers which of the `EI1`…`EI2` inputs have already received at least one event. Once all 2 inputs are marked, `EO` is triggered and the internal marker is reset. An `R` event resets the marker at any time without triggering `EO`.

Technical Features

- **Generic implementation**: Like `E_REND` (2 inputs) itself, `E_REND_2` is based on the same generic `GEN_E_REND` C++ class; the input count is fixed at compile time via the type definition.
- **Order-independent**: The inputs `EI1`, `EI2` do not need to arrive in any particular order.
- **Automatic reset after triggering**: After every `EO`, the block starts over from scratch.

State Overview

The block internally tracks a bitmask of inputs that have already arrived; once the mask is complete, `EO` is triggered and the mask is cleared.

Application Scenarios

- **Synchronizing 2 independent preconditions** before a follow-up step is triggered (e.g. 2 sensors must all have fired before a machine continues).
- **Extending `E_REND`** to more than two event sources without having to cascade multiple `E_REND` blocks.

Comparison with similar function blocks

- **`E_REND`**: the 2-input base variant.
- `E_REND_3`, `E_REND_4`: the same generic implementation with a different input count.
- **`E_MERGE_2`**: the logical OR of multiple event inputs (fires on each individual event instead of only after all of them).

Conclusion

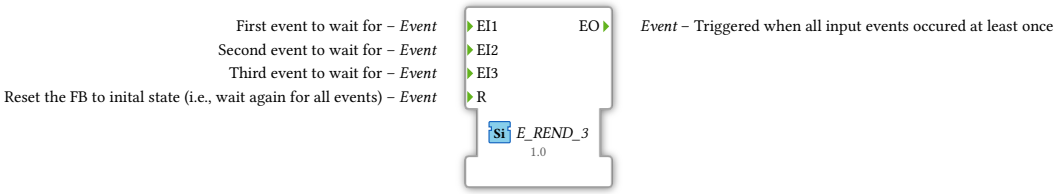
`E_REND_2` extends the rendezvous principle of `E_REND` to 2 simultaneously synchronized event streams and is the generic, 2-input-instantiated variant of the same underlying implementation.

2.11.3.37 E_REND_3

• – ◦ • – ◦ • – ◦ •

Introduction

`E_REND_3` is the generic 3-input variant of the rendezvous block `E_REND` (2 inputs). It only triggers `EO` once it has received at least one event since the last reset from **all** 3 inputs (`EI1`, `EI2`, `EI3`) — a logical AND across 3 asynchronous event streams.



Interface Structure

Event Inputs

- **EI1**: Event input 1.
- **EI2**: Event input 2.
- **EI3**: Event input 3.
- **R (Reset)**: Resets the block to its initial state, discarding all events recorded so far.

Event Outputs

- **EO**: Triggered once all 3 inputs have fired at least once since the last reset.

Functionality

`E_REND_3` is implemented as a generic function block (`eclipse4diac::core::GenericClassName = 'GEN_E_REND'`) — the same C++ implementation underlies all `E_REND_n` variants; only the number of inputs differs. Internally, the block remembers which of the `EI1`…`EI3` inputs have already received at least one event. Once all 3 inputs are marked, `EO` is triggered and the internal marker is reset. An `R` event resets the marker at any time without triggering `EO`.

Technical Features

- **Generic implementation**: Like `E_REND` (2 inputs) itself, `E_REND_3` is based on the same generic `GEN_E_REND` C++ class; the input count is fixed at compile time via the type definition.
- **Order-independent**: The inputs `EI1`, `EI2`, `EI3` do not need to arrive in any particular order.
- **Automatic reset after triggering**: After every `EO`, the block starts over from scratch.

State Overview

The block internally tracks a bitmask of inputs that have already arrived; once the mask is complete, `EO` is triggered and the mask is cleared.

Application Scenarios

- **Synchronizing 3 independent preconditions** before a follow-up step is triggered (e.g. 3 sensors must all have fired before a machine continues).
- **Extending `E_REND`** to more than two event sources without having to cascade multiple `E_REND` blocks.

Comparison with similar function blocks

- **`E_REND`**: the 2-input base variant.
- `E_REND_2`, `E_REND_4`: the same generic implementation with a different input count.
- **`E_MERGE_3`**: the logical OR of multiple event inputs (fires on each individual event instead of only after all of them).

Conclusion

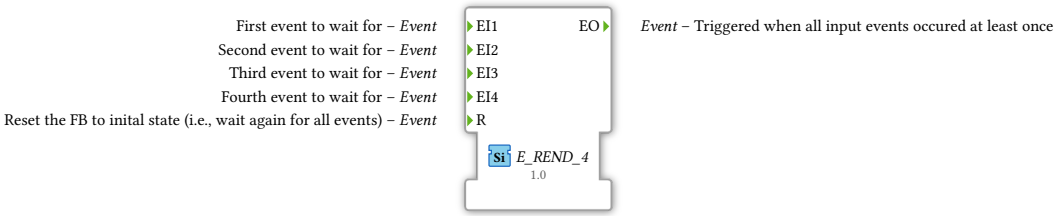
`E_REND_3` extends the rendezvous principle of `E_REND` to 3 simultaneously synchronized event streams and is the generic, 3-input-instantiated variant of the same underlying implementation.

2.11.3.38 E_REND_4

• – ◦ • – ◦ • – ◦ •

Introduction

`E_REND_4` is the generic 4-input variant of the rendezvous block `E_REND` (2 inputs). It only triggers `EO` once it has received at least one event since the last reset from **all** 4 inputs (`EI1`, `EI2`, `EI3`, `EI4`) — a logical AND across 4 asynchronous event streams.



Interface Structure

Event Inputs

- **EI1**: Event input 1.
- **EI2**: Event input 2.
- **EI3**: Event input 3.
- **EI4**: Event input 4.
- **R (Reset)**: Resets the block to its initial state, discarding all events recorded so far.

Event Outputs

- **EO**: Triggered once all 4 inputs have fired at least once since the last reset.

Functionality

`E_REND_4` is implemented as a generic function block (`eclipse4diac::core::GenericClassName = 'GEN_E_REND'`) — the same C++ implementation underlies all `E_REND_n` variants; only the number of inputs differs. Internally, the block remembers which of the `EI1`…`EI4` inputs have already received at least one event. Once all 4 inputs are marked, `EO` is triggered and the internal marker is reset. An `R` event resets the marker at any time without triggering `EO`.

Technical Features

- **Generic implementation**: Like `E_REND` (2 inputs) itself, `E_REND_4` is based on the same generic `GEN_E_REND` C++ class; the input count is fixed at compile time via the type definition.
- **Order-independent**: The inputs `EI1`, `EI2`, `EI3`, `EI4` do not need to arrive in any particular order.
- **Automatic reset after triggering**: After every `EO`, the block starts over from scratch.

State Overview

The block internally tracks a bitmask of inputs that have already arrived; once the mask is complete, `EO` is triggered and the mask is cleared.

Application Scenarios

- **Synchronizing 4 independent preconditions** before a follow-up step is triggered (e.g. 4 sensors must all have fired before a machine continues).
- **Extending `E_REND`** to more than two event sources without having to cascade multiple `E_REND` blocks.

Comparison with similar function blocks

- **`E_REND`**: the 2-input base variant.
- `E_REND_2`, `E_REND_3`: the same generic implementation with a different input count.
- **`E_MERGE_4`**: the logical OR of multiple event inputs (fires on each individual event instead of only after all of them).

Conclusion

`E_REND_4` extends the rendezvous principle of `E_REND` to 4 simultaneously synchronized event streams and is the generic, 4-input-instantiated variant of the same underlying implementation.

2.11.3.39 E_RESTART

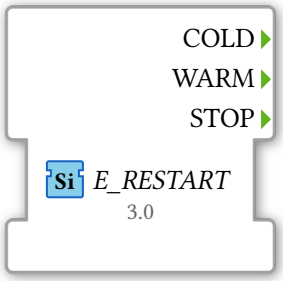



```
COLD  -  Information on cold restart
WARM  -  Information on warm restart
STOP  -  information that the resource is to be stopped
```

• – ◦ • – ◦ • – ◦ •

Introduction

The **E_RESTART** is a special service interface function block according to IEC 61499 (Annex A) for controlling restart sequences in distributed automation systems. The block serves as an interface between resources and higher-level controllers.



Event – Information on cold restart
Event – Information on warm restart
Event – information that the resource is to be stopped

Interface Structure

Event Outputs

- `COLD`: Signals a cold start (full reset)
- `WARM`: Indicates a warm start (partial reset)
- `STOP`: Indicates an upcoming stop command

Service Interfaces

- Left: E_RESTART (FB interface)
- Right: RESOURCE (Device interface)

Functionality

1. **Cold Start Sequence**:

- Upon an external `start` command to the resource
- Triggers the `COLD` event

1. **Warm Start Sequence**:

- Upon `restart` resource command
- Generation of the `WARM` event

1. **Stop Sequence**:

- Upon receiving the `stop` signal
- Output of the `STOP` event

Service Sequences (XML Specification)

1. **cold_restart**:

- `start` ⟶ `COLD` (Full reinitialization)

1. **warm_restart**:

- `restart` ⟶ `WARM` (State-preserving restart)

1. **stopping**:

- `stop` ⟶ `STOP` (Ordered Shutdown)

Technical Features

✓ **Standardized Restart Control** according to IEC 61499 Annex A ✓ **Three Operating Modes** (Cold/Warm/Stop) ✓ **Cross-Resource** Consistency ✓ **EPL 2.0 Open-Source** Implementation

Application Scenarios

- **Field Device Control**: Ordered Reinitialization
- **System Recovery**: Automatic Restart Routines
- **Energy Management**: Scheduled Shutdown
- **Safety-Critical Systems**: State-Safe Restarts

⚖️ Comparison with Similar Function Blocks

Feature	E_RESTART	E_CYCLE	E_DELAY
Purpose	System Control	Time Control	Event Delay
Event Type	Control Commands	Periodic	One-Time
Standard	Annex A	Core	Core

🛠️ Related Exercises

- Exercise_174

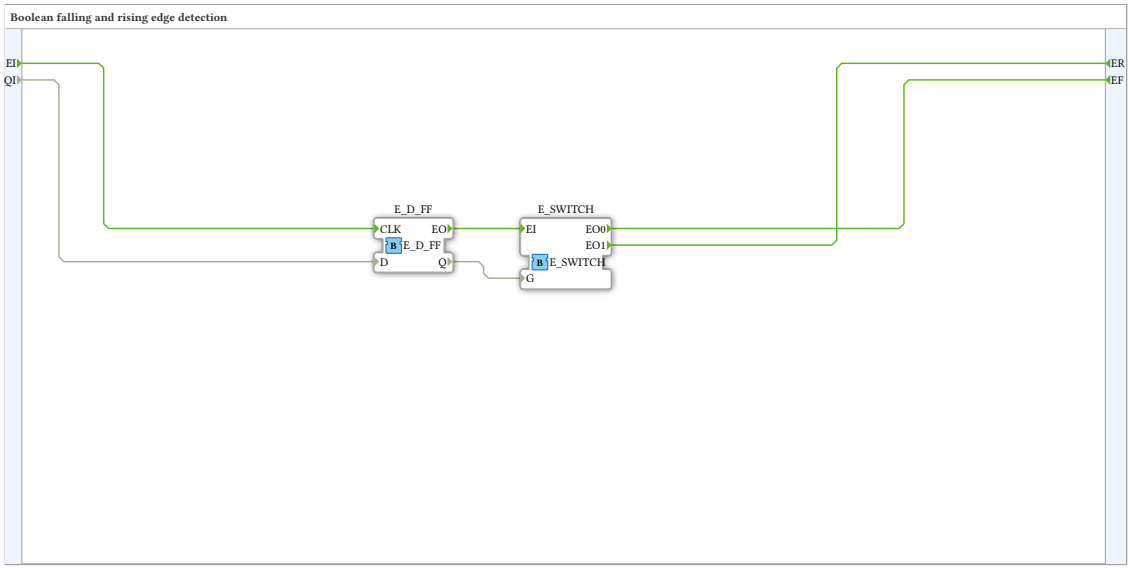
Conclusion

The E_RESTART block offers a standards-compliant solution for critical system control tasks:

- Standardized restart management
- Separation of cold/warm start logic
- Reliable signal forwarding

It particularly enables the development of robust, fault-tolerant control systems. The clear service interface according to IEC 61499 Annex A ensures interoperability across different device platforms.

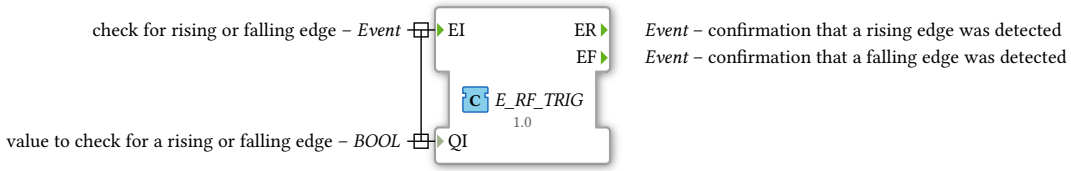
2.11.3.40 E_RF_TRIG



• – ○ • – ○ • – ○ •

Introduction

E_RF_TRIG (rising/falling trigger) detects both rising and falling edges of a BOOL value and reports them via two separate output events. It combines the functionality of E_R_TRIG (rising edge) and E_F_TRIG (falling edge) in a single block.



Interface Structure

Event Inputs

- **EI**: Checks QI for a rising or falling edge.
  - **Connected data**: QI

Event Outputs

- **ER**: Confirms a detected rising edge (QI from FALSE to TRUE).
- **EF**: Confirms a detected falling edge (QI from TRUE to FALSE).

Data Inputs

- **QI** (BOOL): The value to check for edge transitions.

Functionality

Internally, an E_D_FF stores the last known value of QI and, on every EI event, delivers the new value via its EO to an E_SWITCH. Depending on the current Q value (G input), this forwards the event either to EO1 (⟶ ER, rising edge) or EO0 (⟶ EF, falling edge). Since E_D_FF only triggers EO on an actual change of Q, neither ER nor EF is triggered for an unchanged QI.

Technical Features

- **Composed of two standard blocks**: E_D_FF (change detection) + E_SWITCH (branch by value), no ECC of its own.
- **Only on actual change**: Repeated EI events with unchanged QI produce no output event, since E_D_FF already filters internally.
- **Basis for E_TLIM**: There, E_RF_TRIG is used to detect the start (ER) and end (EF) of a TRUE state.

State Overview

Stateless wiring logic: the only stored state is the last known QI value inside the internal E_D_FF.

Application Scenarios

- **Edge detection without two separate blocks**: Wherever both the start and the end of a TRUE state need to be detected (e.g. start and end of a signal), E_RF_TRIG saves a second block compared to using E_R_TRIG/E_F_TRIG separately.
- **Duration measurement of states**: Combined with E_DELAY/E_SR (see E_TLIM) to measure how long a BOOL signal remains in a state.

Comparison with similar function blocks

- **E_R_TRIG**: detects only rising edges.
- **E_F_TRIG**: detects only falling edges.
- E_TLIM: uses E_RF_TRIG internally to time-limit a TRUE state.

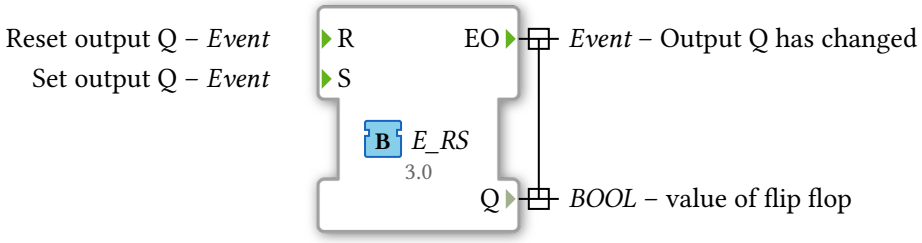
Conclusion

E_RF_TRIG provides compact, combined edge detection for rising and falling transitions of a BOOL signal and is a building block for more complex timing and state logic such as E_TLIM.

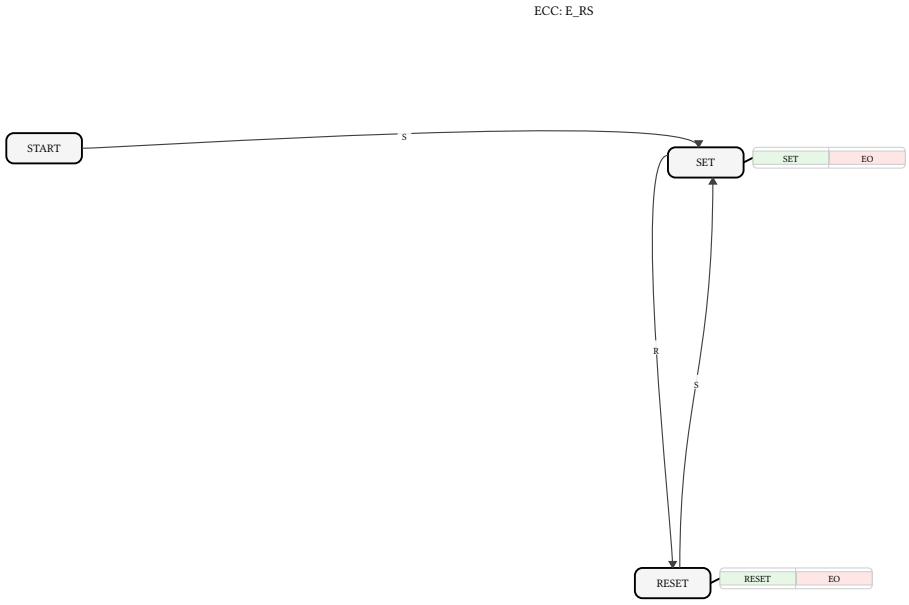
2.11.3.41 E_RS

Introduction

The `E_RS` (Event-driven RS Flip-Flop) is an event-driven, bistable function block according to IEC 61499. It serves as a basic memory element controlled by separate "Set" and "Reset" events. Its output `Q` retains its state until an opposing event occurs.



Interface Structure



Event Inputs:

- **S (Set)**: Sets the output `Q` to `TRUE`.
- **R (Reset)**: Sets the output `Q` to `FALSE`.

Event Outputs:

- **EO (Event Output)**: Triggered when the state of `Q` changes.
- **Associated Data**: `Q`

Data Outputs:

- **Q**: The current state of the flip-flop (data type: `BOOL`).

Functionality

The `E_RS` block functions as a simple latch:

1. **Set**: When an event arrives at the input `S`, the output `Q` is set to `TRUE`. If `Q` was previously `FALSE`, the `EO` event is triggered.

2. **Reset**: When an event arrives at the input R, the output Q is set to FALSE. If Q was previously TRUE, the EO event is triggered.
3. **Save**: Between events, Q retains its last set state.

Technical Features and Standards Comparison

According to **DIN EN 61499-1 (Table A.1, Note 8)**, the implementation of this function block is identical to E_SR. Both function blocks (E_RS and E_SR) exist to maintain consistency with the types in IEC 61131-3, even though IEC 61499 does not have an inherent "dominance" of events, as is the case with level-controlled inputs in classic PLC programming.

- **Comparison to IEC 61131-3**: See RS (Bistable, priority reset). While in IEC 61131-3 the RS function block has a defined "reset dominance" (if R and S are TRUE simultaneously, R wins), the behavior in IEC 61499 for closely spaced events depends on the processing order of the runtime environment (ECC). Since events are transient, there is no permanent conflict between two static signals.
- **Functional Identity**: E_RS and E_SR are technically identical. Their graphical representation and naming conventions simply follow the established naming conventions to facilitate orientation for developers.
- **Change Detection**: The EO output is only triggered by an actual state change.

Application Scenarios

- **Start/Stop Logic**: A "Start" button is connected to S, and a "Stop" button to R, to control the state of a machine.
- **Start/Stop Logic**: - **Error Storage**: An error event sets the function block (S), which stores the error state until it is explicitly acknowledged by an operator or another process (R).
- **Mode Storage**: Stores the current operating mode of a system (e.g., "Manual" vs. "Automatic").

Related Function Blocks

- **E_SR**: Functionally identical to E_RS, with the inputs in the symbol swapped.
- **E_D_FF**: Clock-based storage (Data Latch). E_D_FF takes the value at the D input on a CLK event.

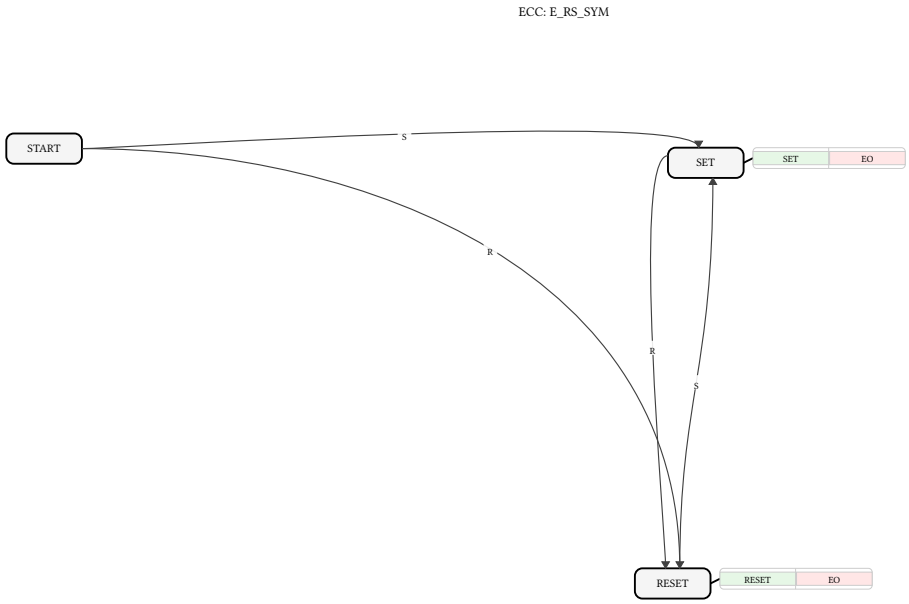
🛠️ Related Exercises

- Exercise_006b
- Exercise_020a
- Exercise_020b
- Exercise_020d

Conclusion

The E_RS block is a fundamental memory block in IEC 61499. It is ideal for simple state storage where a state is set by one event and explicitly reset by another. The lack of guaranteed set or reset dominance for simultaneous events must be considered in critical applications.

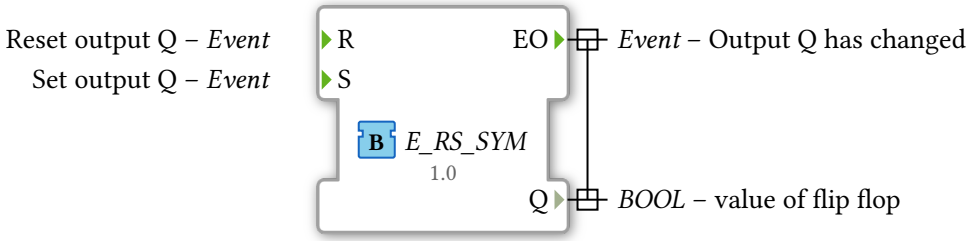
2.11.3.42 E_RS_SYM



• – ◦ • – ◦ • – ◦ •

Introduction

`E_RS_SYM` (Event-driven RS Flip-Flop, symmetric start-up behaviour) is an event-driven, bistable memory element according to IEC 61499. It behaves functionally like E_RS/E_SR, but differs in its initial state: while `E_RS`/`E_SR` only produce a defined output after the first `S` or `R` event, `E_RS_SYM` already reacts symmetrically to both events in its initial state `START` and immediately transitions into the matching `SET` or `RESET` state.



Interface Structure

Event Inputs

- **S (Set)**: Sets output `Q` to `TRUE`.
- **R (Reset)**: Sets output `Q` to `FALSE`.

Event Outputs

- **EO (Event Output)**: Triggered after every `S` or `R` event.
  - **Connected data**: `Q`

Data Outputs

- **Q**: The current state of the flip-flop (data type: `BOOL`).

Functionality

The ECC has three states: `START`, `SET`, and `RESET`. From `START`, both an `S` and an `R` event lead to a defined follow-up state (`SET` or `RESET`) — unlike `E_RS`, where `START` only reacts to one of the two events and the

other has no effect initially. From SET and RESET, the block behaves like a classic RS flip-flop: R switches from SET to RESET, S from RESET to SET. Every state change runs the SET (Q := TRUE) or RESET (Q := FALSE) algorithm and triggers EO.

Technical Features

- **Symmetric start-up behaviour**: The key difference to E_RS/E_SR lies in the START state: both input events (S and R) lead to a defined transition there, so a correct Q value is set regardless of which event arrives first after startup.
- **No init mechanism**: Unlike E_RS_SYM_INIT, this block has no INIT/INITO interface; the start state results solely from the first S or R event received.

State Overview

State	Meaning
START	Initial state, waits symmetrically for S or R
SET	Q = TRUE, reachable from START (via S) or RESET (via S)
RESET	Q = FALSE, reachable from START (via R) or SET (via R)

Application Scenarios

- **Start/stop logic with a guaranteed initial state**: Wherever it is unpredictable after an application restart whether a set or a reset signal arrives first, but a correct Q value is still required immediately.
- **Fault storage**: Like E_RS, but with the additional guarantee that even a first R event (e.g., an acknowledgment before the first fault) correctly results in Q = FALSE.

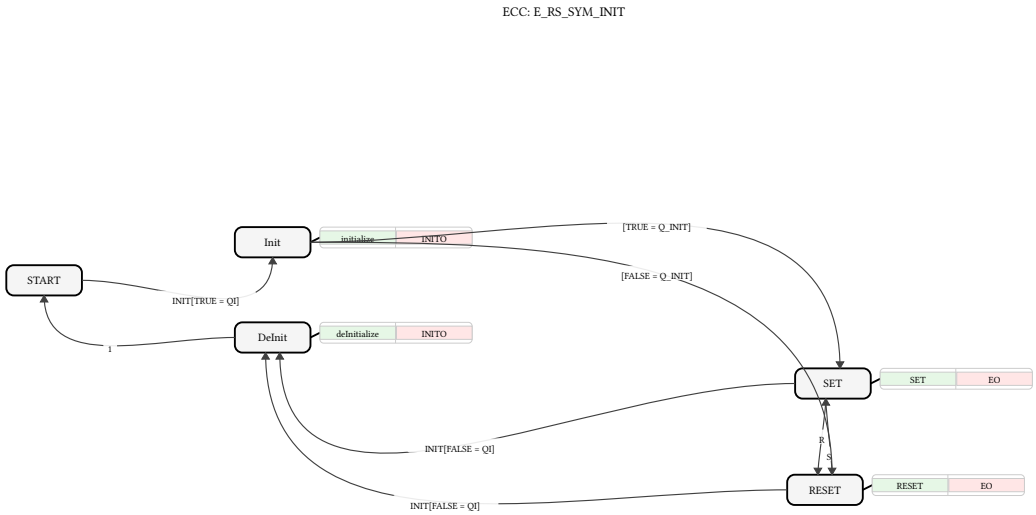
Comparison with similar function blocks

- **E_RS** / **E_SR**: functionally nearly identical, but without symmetric start-up behaviour — in the initial state only one of the two events has an effect.
- **E_RS_SYM_INIT**: the same base functionality, extended with an explicit INIT/INITO interface for setting a defined start value.
- **E_SR_SYM**: functionally identical, only the order of S/R in the interface definition is swapped (naming convention analogous to E_RS/E_SR).

Conclusion

E_RS_SYM provides a bistable memory element with guaranteed, well-defined behaviour from the very first event received, making it suitable wherever the initial-state behaviour of E_RS/E_SR is insufficient.

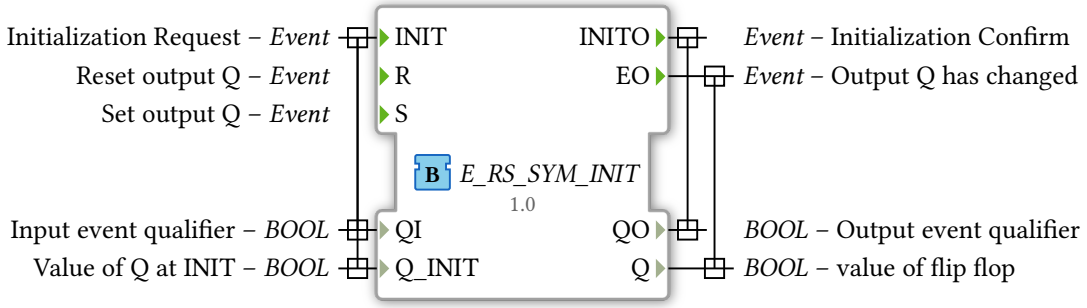
2.11.3.43 E_RS_SYM_INIT



• – ○ • – ○ • – ○ •

Introduction

`E_RS_SYM_INIT` extends E_RS_SYM with an explicit `INIT`/`INITO` interface: instead of the start value resulting purely from the first `S` or `R` event, it can be initialized (via `INIT`) with a defined value `Q_INIT`, or deinitialized via `QI = FALSE`.



Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `QI` and `Q_INIT`.
- **S (Set)**: Sets output `Q` to `TRUE`.
- **R (Reset)**: Sets output `Q` to `FALSE`.

Event Outputs

- **INITO**: Confirms (de-)initialization, carries `QO`.
- **EO**: Triggered after every `S` or `R` event, carries `Q`.

Data Inputs

- **QI** (BOOL): Input event qualifier — `TRUE` initializes, `FALSE` deinitializes.
- **Q_INIT** (BOOL): The value `Q` is set to during initialization.

Data Outputs

- **QO** (BOOL): Output event qualifier, mirrors `QI`.
- **Q** (BOOL): The current state of the flip-flop.

Functionality

From `START`, `INIT` with `QI = TRUE` leads into state `Init`, which sets `QO := QI` and triggers `INITO`; from there, depending on `Q_INIT`, it moves directly to `SET` (`Q_INIT = TRUE`) or `RESET` (`Q_INIT = FALSE`) — without triggering `EO`. From `SET`/`RESET`, the block behaves like E_RS_SYM: `S` switches to `SET`, `R` to `RESET`, each triggering `EO`. A subsequent `INIT` with `QI = FALSE` leads from `SET` or `RESET` into state `DeInit` (sets `QO := FALSE`, triggers `INITO`) and from there back to `START`.

Technical Features

- **QI as init/deinit switch**: `QI = TRUE` initializes the block with `Q_INIT`, `QI = FALSE` deinitializes it and returns it to the unconfigured `START` state.
- **Separate event channels**: Initialization (`INIT`/`INITO`) is fully separated from the running set/reset logic (`S`/`R`/`EO`) — an initialization during runtime deliberately resets `Q`, confirmed via `INITO` rather than `EO`.

State Overview

State	Meaning
START	Unconfigured initial state
Init	Initialization in progress, `QO := QI`
DeInit	Deinitialization in progress, `QO := FALSE`
SET	`Q = TRUE`
RESET	`Q = FALSE`

Application Scenarios

- **Well-defined startup with a preset value**: The start value of `Q` should come from a configuration variable (`Q_INIT`) at system start, instead of depending on which of `S`/`R` happens to arrive first.
- **Controlled reset of entire subnetworks**: `INIT`/`QI = FALSE` can be used to deliberately deinitialize the block, e.g. when disabling part of an FB network.

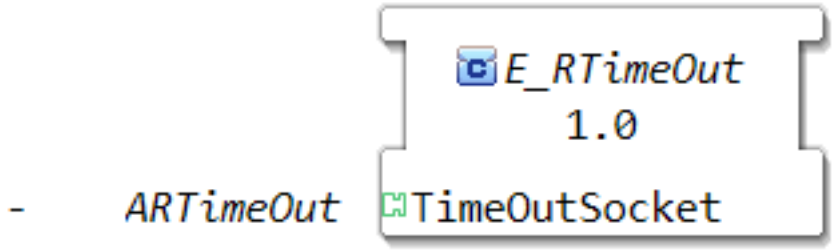
Comparison with similar function blocks

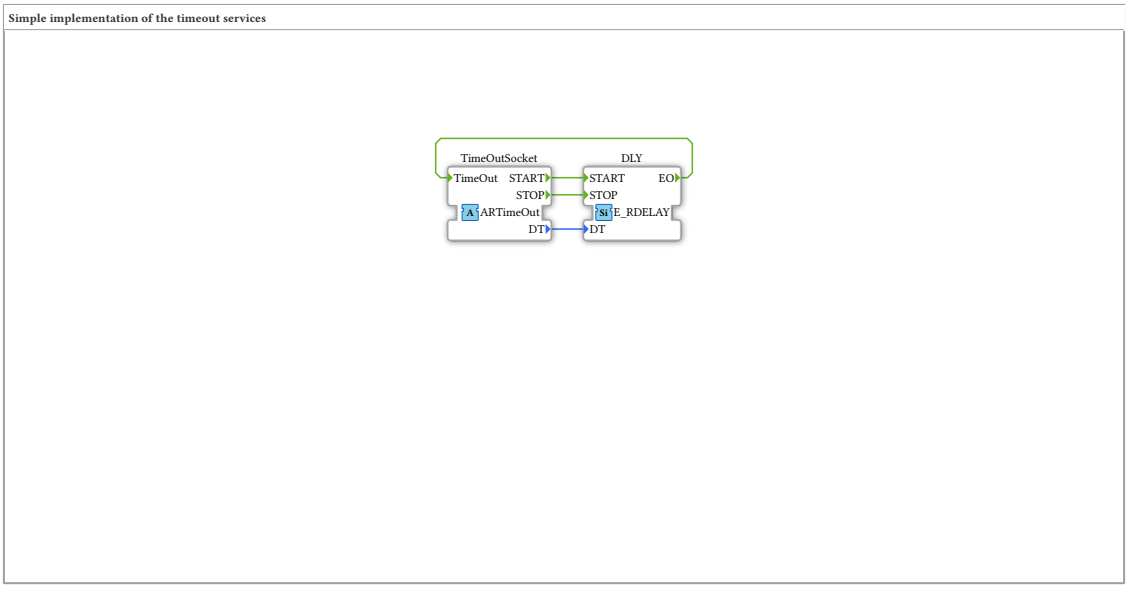
- E_RS_SYM: the same base functionality without an `INIT`/`INITO` interface.
- E_SR_SYM_INIT: functionally identical, only the order of `S`/`R` in the interface is swapped.
- E_T_FF_INIT: the same INIT/DeInit structure, but with toggle instead of set/reset behaviour during normal operation.

Conclusion

`E_RS_SYM_INIT` provides a bistable memory element with a deliberately configurable start value, making it suitable for applications where the pure chance of the first set/reset event is not sufficient.

2.11.3.44 E_RTimeOut (Resettable Timeout Service)

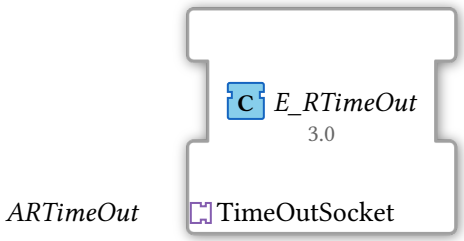




• – ○ • – ○ • – ○ •

Introduction

The **E_RTimeOut** is a special function block according to IEC 61499-2. It implements a resettable timeout service by internally using an E_RDELAY block.



Structure and Interfaces

Adapter Interface (Socket Perspective)

The block uses a **socket** named `TimeOutSocket` of type `ARTimeOut` (`iec61499::events::ARTimeOut`); the block has no other event or data connections. Since it is a socket, the signal directions are inverted compared to the adapter definition (plug):

- **Inputs (received from the socket)**:
- `START`: Starts or resets the internal timer.
- `STOP`: Stops the internal timer.
- `DT` (TIME): The delay time to use.
- **Output (sent to the socket)**:
- `TimeOut`: Signaled to the connected plug after the time has elapsed.

Internal Components

- `DLY` (E_RDELAY): Core component for time control

Functionality

1. **Timeout Initialization**:

- The timer starts when the `START` event occurs on the socket.
- Uses the configured `DT` value.

1. **Timeout Reset**:

- A new `START` event resets the running timer.
- Uses the new `DT` value.

1. **Timeout Termination**:

- The `STOP` event terminates the active timer.

1. **Timeout Triggering**:

- After `DT` expires, the `TimeOut` event is generated
- Signaled via the adapter socket

Technical Features

✓ **Resettable Timeout Function**
  ✓ **Adapter-Based Interface** (ARTimeOut)
  ✓ **Internal E_RDELAY Implementation**
  ✓ **IEC 61499-2 Compliant**

Application Scenarios

- **Network Communication**: Response Timeout Monitoring
- **Machine Safety**: Monitoring of Motion Periods
- **Process Control**: Time-Limited Operation Control
- **Device Control**: Watchdog Functionality

⚖️ **Comparison with Similar Components**

Feature	E_RTimeOut	E_DELAY	E_TABLE
Reset Function	✓	❌	❌
Interface	Adapter	Direct	Direct
Standard	61499-2	61499-1	61499-1

🛠️ **Related Exercises**

- Exercise_035b
- Exercise_170

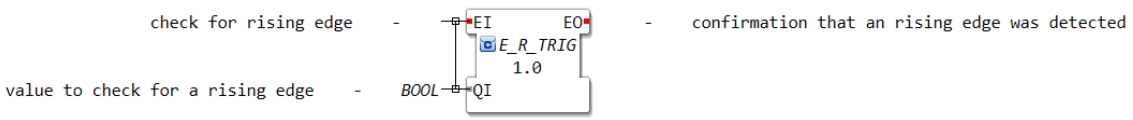
Conclusion

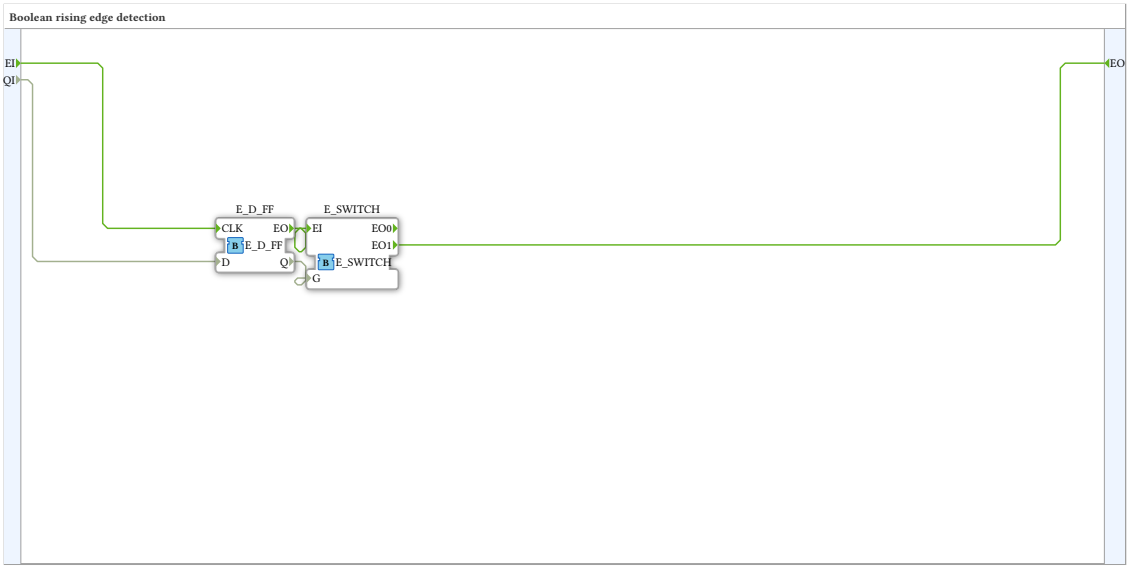
The E_RTimeOut block offers an elegant solution for timeout requirements in distributed control systems:

- Reusable adapter interface
- Flexible timeout configuration
- Reliable reset functionality

By internally utilizing the E_RDELAY block, it combines precise timing control with a robust architecture. Its standardized implementation according to IEC 61499-2 makes it particularly suitable for interoperable automation solutions.

2.11.3.45 E_R_TRIG (Rising Edge Detection)





• – ◦ • – ◦ • – ◦ •

Introduction

The **E_R_TRIG** (Rising Edge Trigger) is a fundamental function block according to IEC 61499 for detecting rising edges in digital signals. Based on the XML specification, it implements reliable edge detection for industrial control systems.



Structure of the E_R_TRIG Block

Interface

Event Inputs:

- `EI` (Event Input): Enables edge detection (linked to the `QI` value)

Event Outputs:

- `EO` (Event Output): Acknowledges a detected rising edge

Data Inputs:

- `QI` (Query Input): Input signal to be monitored (BOOL)

Internal Structure

According to the XML specification, the block consists of:

1. **E_D_FF**: D flip-flop for signal state storage
2. **E_SWITCH**: Switch for conditional Event Forwarding

Functionality

1. **Edge Detection**:

- With each `EI` event, the current `QI` value is compared with the stored state from the previous cycle.
- A rising edge occurs when `QI` transitions from `FALSE` to `TRUE`.
- Upon detection of such an edge, the `EO` event should be triggered.

1. **Internal Structure and Faulty Behavior**:

- The function block is internally composed of a `E_D_FF` and a `E_SWITCH`.

- **Caution: Faulty Implementation!** The internal logic of the function block (as of `events-3.0.0`) is faulty and actually implements a **falling edge detector (`E_F_TRIG`)** instead of a rising one.
- The `EO` event is triggered when `QI` transitions from `TRUE` to `FALSE`.

Technical Features

✓ **Event-driven** architecture (no polling).
✓ **State storage** via an internal D flip-flop.
✓ **Faulty Logic**: The function block is named `E_R_TRIG` (Rising), but its internal wiring causes it to react to a **falling edge**. Users who require rising edge detection must use the logic of the `E_F_TRIG` function block, which ironically correctly detects a rising edge.

Application Scenarios (for a rising edge)

- **Sensor Data Evaluation**: Detect when a sensor changes from "inactive" to "active".
- **Machine Safety**: Detect a start or enable signal.
- **Process Control**: Start a process when a signal becomes `TRUE`.

⚖️ Comparison with Similar Function Blocks

Feature	E_R_TRIG (this one)	E_F_TRIG	E_D_FF
Detected Edge	Rising (named), Falling (implemented)	Falling (named), Rising (implemented)	Clock Edge
Event Output	Yes	Yes	Yes
Memory Behavior	Yes	Yes	Yes

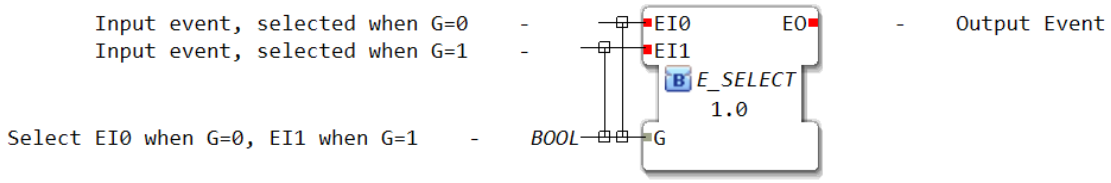
🛠️ Related Exercises

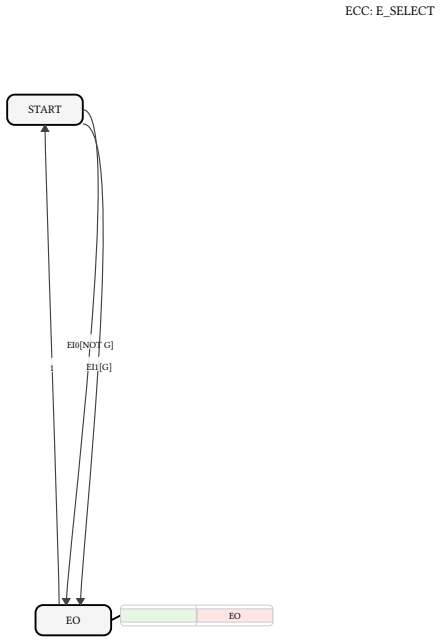
- Exercise_089

Conclusion

The E_R_TRIG block is designed to detect rising edges. **Due to an implementation error in the `events-3.0.0` library, this function block functions as a detector for falling edges.** This discrepancy between name and function is critical and must be taken into account when using it. For reliable detection of rising edges, the `E_F_TRIG` function block should be used instead, whose logic (contrary to *its* name) correctly detects a rising edge.

2.11.3.46 E_SELECT

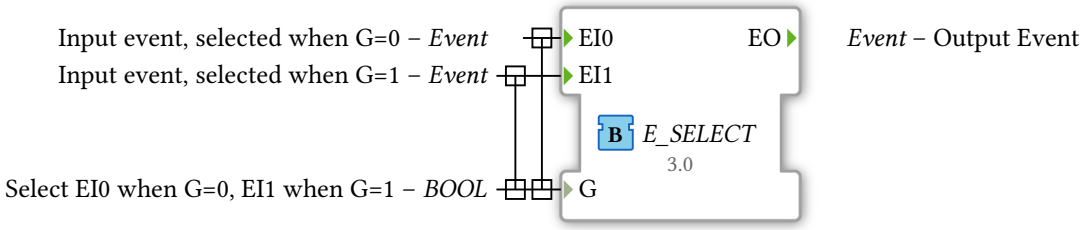




• – ○ • – ○ • – ○ •

Introduction

The **E_SELECT** is a basic function block according to IEC 61499 (Annex A) that enables the conditional forwarding of events based on a control signal. The current version 1.0 is licensed under EPL-2.0.



Interface Structure

Event Inputs

- `EI0`: Input event (passed on if G=0)
- `EI1`: Input event (passed on if G=1)

Event Outputs

- `EO`: Output event (passed on)

Data Inputs

- `G` (BOOL): Control signal for selection:
- G=0: Pass on EI0
- G=1: Pass on EI1

Functionality

1. **Event Processing**:

- Upon input of EI0 or EI1, the G-value is evaluated.
- Only the event matching the G-value is passed on.

1. **State Machine** (ECC):

- **START**: Wait state
- **EO**: Output state (with EO action)
- Transitions:
- EI0 at G=0 ⟶ EO
- EI1 at G=1 ⟶ EO
- Always returns to START

1. **Execution Logic**:

- Deterministic event selection
- No buffering of events

Technical Features

✓ **Boolean control** of event selection ✓ **Real-time processing**
✓ **State-based implementation (BasicFB) ✓ **EPL 2.0 Open-Source** implementation

Application Scenarios

- **Branched Process Control**: Alternative execution paths
- **Mode Switching**: Operating mode change
- **Error Handling**: Alternative error routines
- **Test Automation**: Switching between test and normal operation

⚖️ **Comparison with similar function blocks**

Feature	E_SELECT	E_SWITCH	E_MERGE
Selection Criterion	Boolean (G)	Boolean (G)	None
Direction	2:1 (Multiplexer)	1:2 (Demultiplexer)	n:1 (OR Gate)
State Model	BasicFB	BasicFB	BasicFB/Generic

🛠️ **Related Exercises**

- Exercise_095

Conclusion

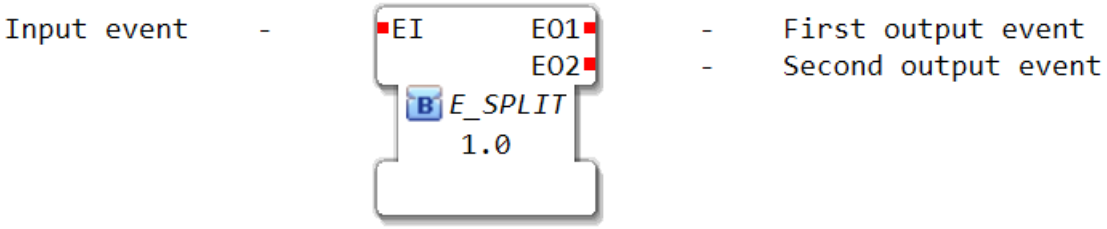
The E_SELECT function block offers a robust solution for event-based control decisions:

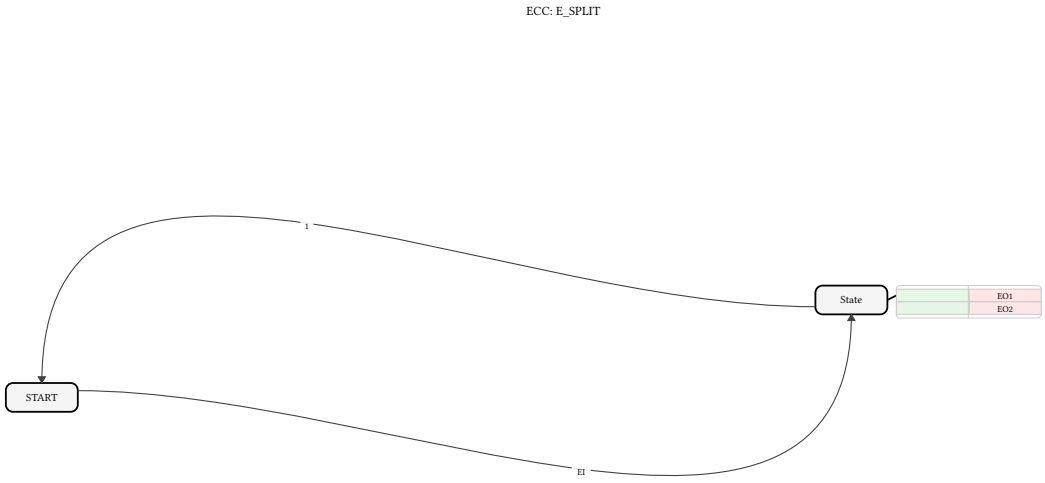
- Simple yet effective selection
- Clear state machine implementation
- Standards-compliant interface

Due to its deterministic operation, it is particularly suitable for safety-critical applications and complex control logic. Its use as a BasicFB also enables integration into all IEC 61499-compliant development environments.

See also: https://www.holobloc.com/doc/fb/rt/events/E_SELECT.htm

2.11.3.47 E_SPLIT (Event Distributor)

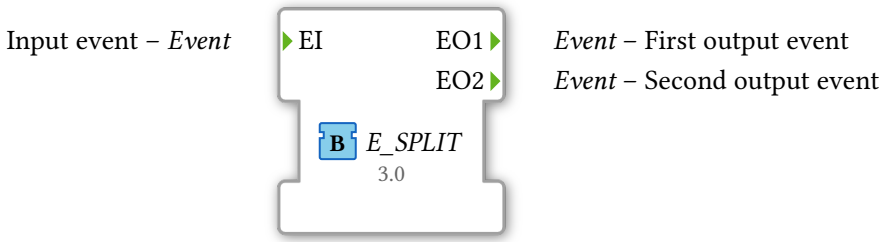




• – ◦ • – ◦ • – ◦ •

Introduction

The **E_SPLIT** is a standards-compliant function block (IEC 61499-1 Annex A) for event distribution, developed under the EPL 2.0 license. Version 1.0 splits an incoming event sequentially into two output events.



Interface Structure

Event Input

- `EI`: Input event (trigger for distribution)

Event Outputs

- `EO1`: First output event
- `EO2`: Second output event

Functionality

1. **Event Receipt**:

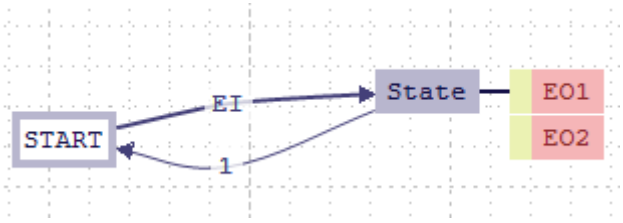
- Upon receipt of `EI`, the state machine is activated.

1. **Sequential Processing**:

- **START State**: Waits for input event.
- **STATE State**:
- Executes action `EO1` (immediately).
- Executes action `EO2`. (Immediately after)
- Automatic return to START

1. **Execution Order**:

- Guaranteed sequence: EI ⟶ EO1 ⟶ EO2
- Deterministic timing



Technical Features

✓ **Strict Sequencing** (EO1 before EO2)
✓ **State-based Implementation** (BasicFB) ✓ **Real-time event processing**
✓ **EPL 2.0 Open-Source** Implementation

Application Scenarios

- **Flow Control**: Clocked process steps
- **Device Control**: Activation sequences
- **Security Systems**: Delayed emergency routines
- **Test Automation**: Triggers for test sequences

⚖️ **Comparison with similar building blocks**

Feature	E_SPLIT	E_DEMUX	E_MERGE
Functional principle	1:2 sequence	1:n distribution	n:1 combination
Event sequence	Fixed	Address-dependent	Arbitrary
State model	BasicFB	Variable	None

🛠️ **Related Exercises**

- Exercise_004a4
- Exercise_004a4_AX
- Exercise_080b

Conclusion

The E_SPLIT block offers a reliable solution for sequential event distribution:

- Guaranteed event sequence
- Simple yet effective functionality
- Robust state machine model

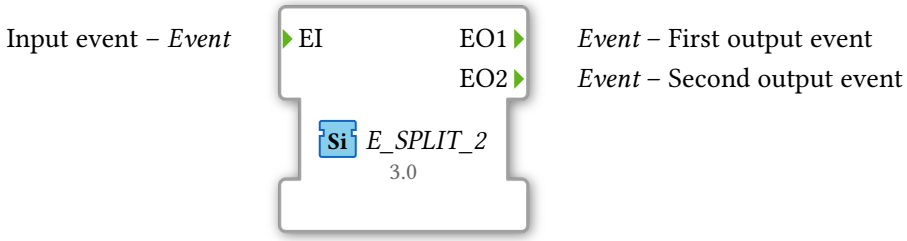
Due to its deterministic operation, it is particularly suitable for time-critical control tasks and safety-related applications. Its standards-compliant implementation enables seamless integration into IEC 61499-based systems.

2.11.3.48 E_SPLIT_2

• – ◦ • – ◦ • – ◦ •

Introduction

The `E_SPLIT_2` is a basic function block according to IEC 61499 that splits a single input event (`EI`) into a sequence of two output events (`EO1` and `EO2`). This block is functionally identical to `E_SPLIT`.



Interface Structure

Event Input

- **EI**: The input event that triggers the sequence.

Event Outputs

- **EO1**: The first output event of the sequence.
- **EO2**: The second output event of the sequence.

Functionality

When an event arrives at input `EI`, the function block immediately triggers the two output events in succession. The sequence is guaranteed:

1. First, `EO1` is triggered.
2. Immediately afterward, `EO2` is triggered.

Technical Features

- **1-to-2 Sequence**: Splits an event into a guaranteed sequence of two events.
- **Stateless**: The function block has no internal memory beyond a single transaction.
- **Generic Function Block**: The functionality is provided by the generic class `GEN_E_SPLIT`, enabling optimized execution by the 4diac runtime environment.

Application Scenarios

- **Reset and Start**: A single `RESET` event can be used to first reset a value (`EO1`) and then restart a process (`EO2`).
- **Read and Process**: A `REQ` event can first read a sensor value (`EO1`) and then trigger a calculation using the new value (`EO2`).
- **Sequence Control**: Triggering two consecutive actions with a single trigger event.

🛠️ Related Exercises

- Exercise_004a8
- Exercise_004a8_AX

Conclusion

The `E_SPLIT_2` block is a useful tool for generating a fixed, short event sequence from a single trigger. Its simplicity and guaranteed execution order make it a robust component for sequence control in IEC 61499 applications.

🌐 Related Topic Subpages on ms-muc-docs.de

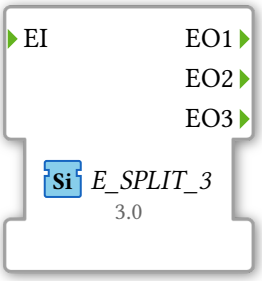
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.3.49 E_SPLIT_3

• – ◦ • – ◦ • – ◦ •

The `E_SPLIT_3` is a basic function block according to IEC 61499 that splits a single input event (`EI`) into a sequence of three output events (`EO1`, `EO2`, `EO3`).

Input event – *Event*



Event – First output event
Event – Second output event
Event

- **EI**: The input event that triggers the sequence.
- **EO1**: The first output event of the sequence.
- **EO3**: The third output event of the sequence.

Event Outputs

Ereignis-Eingang

Interface Structure

Introduction

Functionality

When an event arrives at input `EI`, the function block immediately triggers the three output events in succession. The order is guaranteed:

1. First, `EO1` is triggered.
2. Immediately afterward, `EO2` is triggered.
3. Immediately afterward, `EO3` is triggered.

- **1-to-3 Sequence**: Splits an event into a guaranteed sequence of three events.
- **Stateless**: The function block has no internal memory beyond a single transaction.
- **Generic Building Block**: The functionality is provided by the generic class `GEN_E_SPLIT`.
- **Initialization Sequence**: A `INIT` event can be used to initialize three different system components sequentially (`EO1`, `EO2`, `EO3`).
- **Sequence Control**: Triggering three consecutive actions with a single trigger event (e.g., “Close gripper,” “Raise arm,” “Rotate arm”).
- Exercise_004a9
- Exercise_004a9_AX

The `E_SPLIT_3` block is a useful tool for generating a fixed, short event sequence of three steps from a single trigger. Its simplicity and guaranteed execution order make it a robust component for sequence control in IEC 61499 applications.

Technical Features

Application Scenarios

🛠️ Zugehörige Übungen

Conclusion

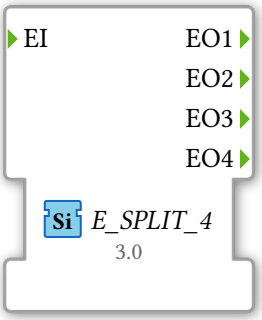
2.11.3.50 E_SPLIT_4

• – ◦ • – ◦ • – ◦ •

Introduction

The `E_SPLIT_4` is a basic function block according to IEC 61499 that splits a single input event (`EI`) into a sequence of four output events (`EO1` to `EO4`).

Input event – *Event*



Event – First output event
Event – Second output event
Event
Event

Interface Structure

Event Input

- **EI**: The input event that triggers the sequence.

Event Outputs

- **EO1**: The first output event of the sequence.
- **EO2**: The second output event of the sequence.
- **EO3**: The third output event of the sequence.
- **EO4**: The fourth output event of the sequence.

Functionality

When an event arrives at input `EI`, the function block immediately triggers the four output events in succession. The order is guaranteed:

1. First, `EO1` is triggered.
2. Immediately afterward, `EO2` is triggered.
3. Immediately afterward, `EO3` is triggered.
4. Immediately afterward, `EO4` is triggered.

Technical Features

- **1-to-4 Sequence**: Splits an event into a guaranteed sequence of four events.
- **Stateless**: The function block has no internal memory beyond a single transaction.
- **Generic Block**: The functionality is provided by the generic class `GEN_E_SPLIT`.

Application Scenarios

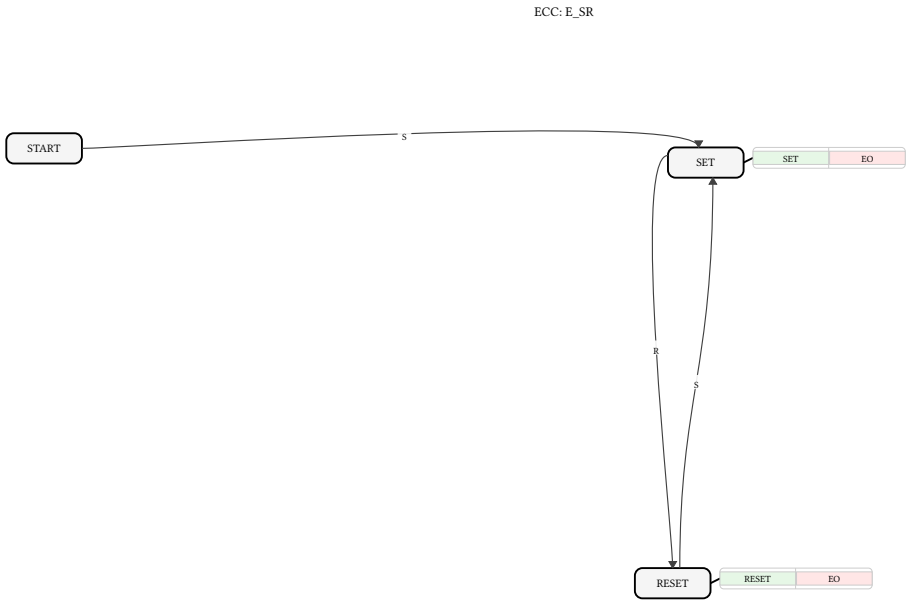
- **Complex Initialization**: A `INIT` event can be used to initialize four different system components sequentially.
- **Sequence Control**: Triggering four consecutive actions with a single trigger event.

Conclusion

The `E_SPLIT_4` block is a useful tool for generating a fixed, short event sequence of four steps from a single trigger. Its simplicity and guaranteed execution order make it a robust component for sequence control in IEC 61499 applications.

2.11.3.51 E_SR

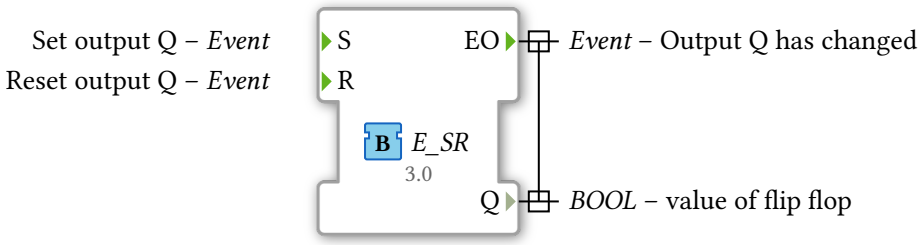
Podcast



- IEC 61499: The E_SR Function Block Decoded – Simplicity Meets Event Control
- Decoding the E_SR Function Block: The Unsung Hero of Industrial Automation

Introduction

The `E_SR` (Event-driven SR Flip-Flop) is an event-driven, bistable function block according to IEC 61499. It serves as a basic memory element controlled by separate "Set" and "Reset" events. Its output, `Q`, retains its state until an opposing event occurs.



Interface Structure

Event Inputs:

- **S (Set)**: Sets the output `Q` to `TRUE`.
- **R (Reset)**: Sets the output `Q` to `FALSE`.

Event Outputs:

- **EO (Event Output)**: Triggered when the state of `Q` changes.
- **Associated Data**: `Q`

Data Outputs:

- **Q**: The current state of the flip-flop (data type: `BOOL`).

Functionality

The `E_SR` block functions as a simple latch:

1. **Set**: When an event arrives at the input `S`, the output `Q` is set to `TRUE`. If `Q` was previously `FALSE`, the `EO` event is triggered.
2. **Reset**: When an event arrives at the input `R`, the output `Q` is set to `FALSE`. If `Q` was previously `TRUE`, the `EO` event is triggered.
3. **Save**: Between events, `Q` retains its last set state.

Technical Features and Standards Comparison

According to **DIN EN 61499-1 (Table A.1, Note 8)**, the implementation of this function block is identical to E_RS. Both function blocks (`E_SR` and `E_RS`) exist to maintain consistency with the types in IEC 61131-3, even though IEC 61499 does not have an inherent "dominance" of events, as is the case with level-controlled inputs in classic PLC programming.

- **Comparison to IEC 61131-3**: See SR (Bistable, set first). While in IEC 61131-3 the `SR` function block has a defined "set dominance" (if S and R are TRUE simultaneously, S wins), in IEC 61499 the behavior with closely spaced events depends on the processing order of the runtime environment (ECC). Since events are transient, there is no permanent conflict between two static signals.
- **Functional Identity**: `E_SR` and `E_RS` are technically identical. Their graphical representation and naming conventions simply follow established naming conventions to aid developers.
- **Change Detection**: The `EO` output is only triggered by an actual state change.

Application Scenarios

- **Start/Stop Logic**: A "Start" button is connected to `S`, and a "Stop" button to `R`, to control the state of a machine.
- **Start/Stop Logic**: - **Error Storage**: An error event sets the function block (`S`), which stores the error state until it is explicitly acknowledged by an operator or another process (`R`).
- **Mode Storage**: Stores the current operating mode of a system (e.g., "Manual" vs. "Automatic").

Related Function Blocks

- **E_RS**: Functionally identical to `E_SR`. The only difference is the graphical arrangement of the `S` and `R` connections on the symbol.
- **`E_D_FF`**: Also stores a state, but on a clock-based basis. `E_D_FF` takes the value from the `D` input when a `CLK` event occurs.

E_D_FF`

🛠️ Related exercises

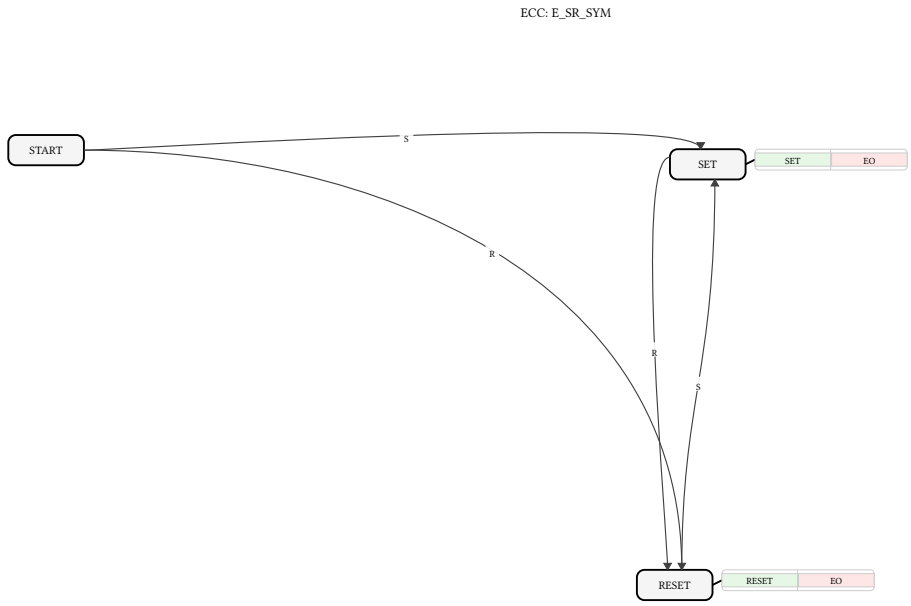
- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_006
- Uebung_006c
- Uebung_006d
- Uebung_007a3
- Uebung_008
- Uebung_009
- Uebung_013
- Uebung_014
- Uebung_015
- Uebung_016
- Uebung_019b
- Uebung_019c
- Uebung_021
- Uebung_022
- Uebung_023
- Uebung_024
- Exercise_025
- Exercise_026_sub

- Exercise_039a_sub_Outputs
- Exercise_160b

Conclusion

The E_SR block is a fundamental memory block in IEC 61499. It is ideal for simple state storage where a state is set by one event and explicitly reset by another. The lack of guaranteed set or reset dominance for simultaneous events must be considered in critical applications.

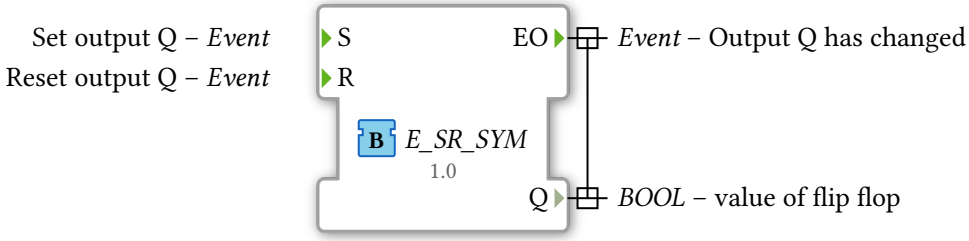
2.11.3.52 E_SR_SYM



• – ○ • – ○ • – ○ •

Introduction

E_SR_SYM (Event-driven SR Flip-Flop, symmetric start-up behaviour) is functionally identical to E_RS_SYM — both blocks exist, analogous to E_RS/E_SR, purely to preserve the IEC 61131-3 naming convention (SR = named set-first, RS = named reset-first), even though IEC 61499 has no true dominance between events.



Interface Structure

Event Inputs

- **S (Set)**: Sets output Q to TRUE.
- **R (Reset)**: Sets output Q to FALSE.

Event Outputs

- **EO (Event Output)**: Triggered after every S or R event.

- **Connected data**: `Q`

Data Outputs

- **Q**: The current state of the flip-flop (data type: `BOOL`).

Functionality

Identical to E_RS_SYM: the ECC has the states `START`, `SET`, and `RESET`. Already in state `START`, both `S` and `R` lead to a defined follow-up state. From `SET`/`RESET`, `R` switches to `RESET`, `S` to `SET`. Every transition sets `Q` accordingly and triggers `EO`.

Technical Features

- **Functionally identical to `E_RS_SYM`**: The graphical representation and naming (`S` before `R` in the interface) follows the IEC 61131-3 convention but has no effect on actual behaviour.
- **Symmetric start-up behaviour**: As with `E_RS_SYM`, the initial state already reacts in a well-defined way to both events.

State Overview

State	Meaning
START	Initial state, waits symmetrically for `S` or `R`
SET	`Q = TRUE`
RESET	`Q = FALSE`

Application Scenarios

See E_RS_SYM — identical use cases; `E_SR_SYM` is preferred wherever a project consistently uses the `SR` naming convention (set listed first).

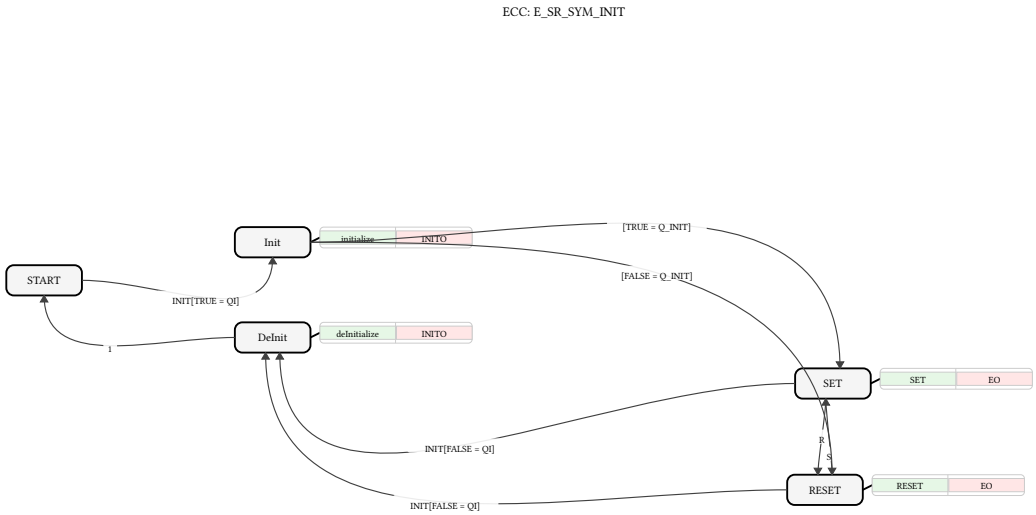
Comparison with similar function blocks

- **E_RS_SYM**: functionally identical, only the order of `S`/`R` in the symbol is swapped.
- **E_SR_SYM_INIT**: the same base functionality, extended with an `INIT`/`INITO` interface.
- **E_SR**: without symmetric start-up behaviour.

Conclusion

`E_SR_SYM` is the naming-convention counterpart of `E_RS_SYM` and functionally identical variant of the symmetric set-reset flip-flop.

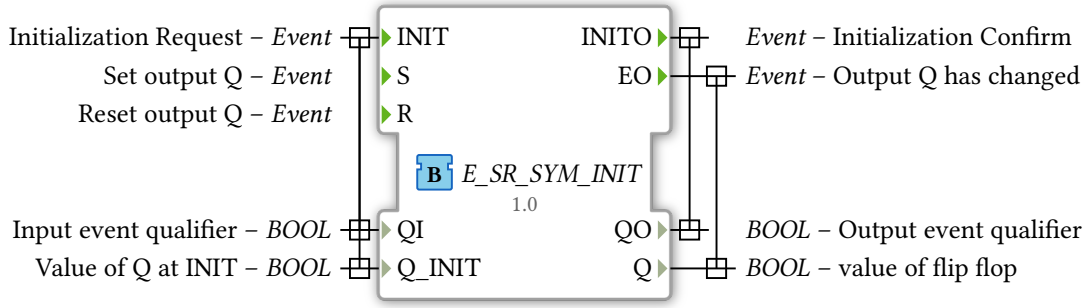
2.11.3.53 E_SR_SYM_INIT



• – ○ • – ○ • – ○ •

Introduction

`E_SR_SYM_INIT` is functionally identical to E_RS_SYM_INIT — analogous to E_SR_SYM/E_RS_SYM, it exists purely to preserve the `SR` naming convention (set listed first) and extends E_SR_SYM with an `INIT`/`INIT0` interface.



Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `QI` and `Q_INIT`.
- **S (Set)**: Sets output `Q` to `TRUE`.
- **R (Reset)**: Sets output `Q` to `FALSE`.

Event Outputs

- **INIT0**: Confirms (de-)initialization, carries `QO`.
- **EO**: Triggered after every `S` or `R` event, carries `Q`.

Data Inputs

- **QI** (BOOL): Input event qualifier — `TRUE` initializes, `FALSE` deinitializes.
- **Q_INIT** (BOOL): The value `Q` is set to during initialization.

Data Outputs

- **QO** (BOOL): Output event qualifier, mirrors `QI`.
- **Q** (BOOL): The current state of the flip-flop.

Functionality

Identical to E_RS_SYM_INIT: `INIT` with `QI = TRUE` initializes `Q` via `Q_INIT`, `INIT` with `QI = FALSE` deinitializes the block back to state `START`. During normal operation, `S`/`R` switch between `SET` and `RESET`, each confirmed via `EO`.

Technical Features

See E_RS_SYM_INIT — identical behaviour, only the `S`/`R` order in the symbol is swapped.

State Overview

State	Meaning
START	Unconfigured initial state
Init	Initialization in progress, `QO := QI`
DeInit	Deinitialization in progress, `QO := FALSE`
SET	`Q = TRUE`
RESET	`Q = FALSE`

Application Scenarios

See E_RS_SYM_INIT.

Comparison with similar function blocks

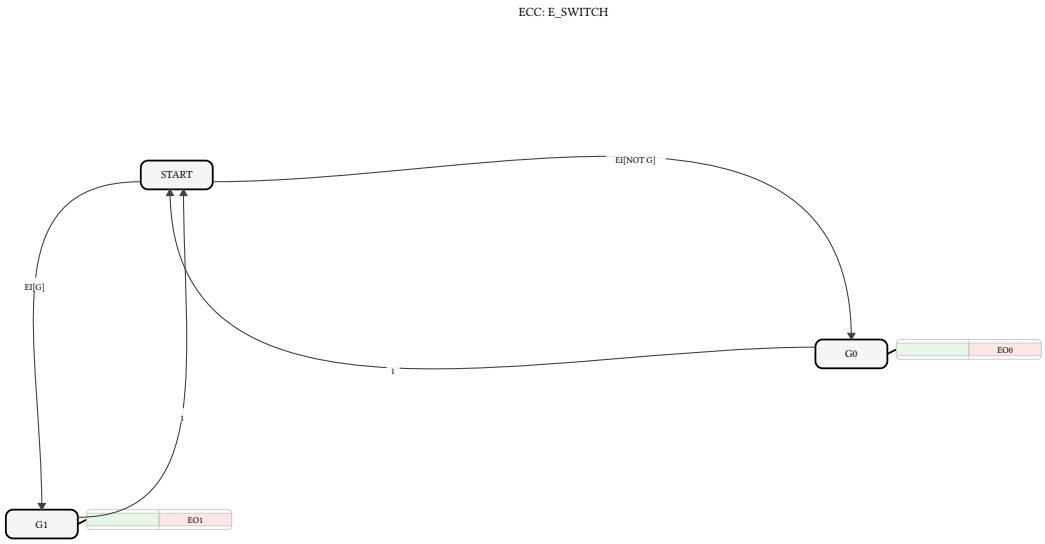
- **E_RS_SYM_INIT**: functionally identical, swapped `S`/`R` order.
- **E_SR_SYM**: the same base functionality without `INIT`/`INITO`.

Conclusion

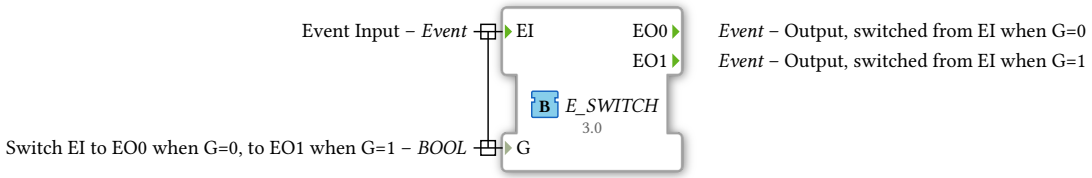
`E_SR_SYM_INIT` is the naming-convention counterpart of `E_RS_SYM_INIT` and functionally identical variant with a configurable start value.

2.11.3.54 E_SWITCH

🎧 Podcast



- E_SWITCH: The Switch of Automation – Why Simplicity Revolutionizes IEC 61499
- E_SWITCH: The Unsung Hero of Industrial Automation's Modular Design The `E_SWITCH` (Event Switch) is a basic function block according to IEC 61499 that functions as a conditional event switch or 1-to-2 demultiplexer. It forwards a single input event (`EI`) to one of two outputs based on the state of a Boolean control variable (`G`).



- **EI (Event Input)**: The input event to be forwarded.
- **Associated Data**: `G`
- **EO0**: The first event output.
- **EO1**: The second event output.
- **G (Guard)**: The Boolean condition that controls the switch (data type: `BOOL`).

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

1. **Event Reception**: The function block waits for an event at input `EI`.
2. **Conditional Forwarding**:

- **If `G` = `FALSE`**: The event is forwarded to output `EO0`.

- **If `G` = `TRUE`**: The event is forwarded to output `EO1`.

The event at input `EI` is always routed to exactly one of the two outputs, depending on the control condition `G`.

- **Event Switch**: Serves as a fundamental "switch" or "if-else" construct for the event flow.
- **Stateless**: The block itself has no internal memory; its decision is based solely on the value of `G` at the moment of the `EI` event.
- **Conditional Logic**: Execution of alternative actions based on a condition. If `G` is true, execute action A (`EO1`); otherwise, execute action B (`EO0`).
- **Mode-Dependent Processing**: Route commands to different processing paths depending on whether the system is in "Automatic" mode (`G=TRUE`) or "Manual" mode (`G=FALSE`).
- **Error Handling**: Route a sensor event either to the normal processing logic (`G=FALSE` for "no error") or to an error handling routine (`G=TRUE` for "error detected").

Feature	E_SWITCH	E_SELECT	E_DEMUX
Selection Criterion	Boolean (G)	Boolean (G)	Index (K)
Direction	1:2 (Demultiplexer)	2:1 (Multiplexer)	1:n (Demultiplexer)
State Model	BasicFB	BasicFB	BasicFB/Generic

- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_005
- LinksRechts_T_FF_Event
- Uebung_007a3
- Uebung_008
- Uebung_009
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c3
- Uebung_020d
- Uebung_020e2
- Uebung_020f2
- Uebung_020i
- Uebung_025
- SR_TO_QX_TRIG_EO
- Exercise_039a_sub_Outputs
- Exercise_086
- Exercise_088
- Exercise_089

The `E_SWITCH` block is a fundamental and widely used block for implementing conditional branching in the event-driven logic of IEC 61499. Its simplicity and clear function as an "if-else" switch for events make it an indispensable tool for creating flexible and robust control software.

Technical Features

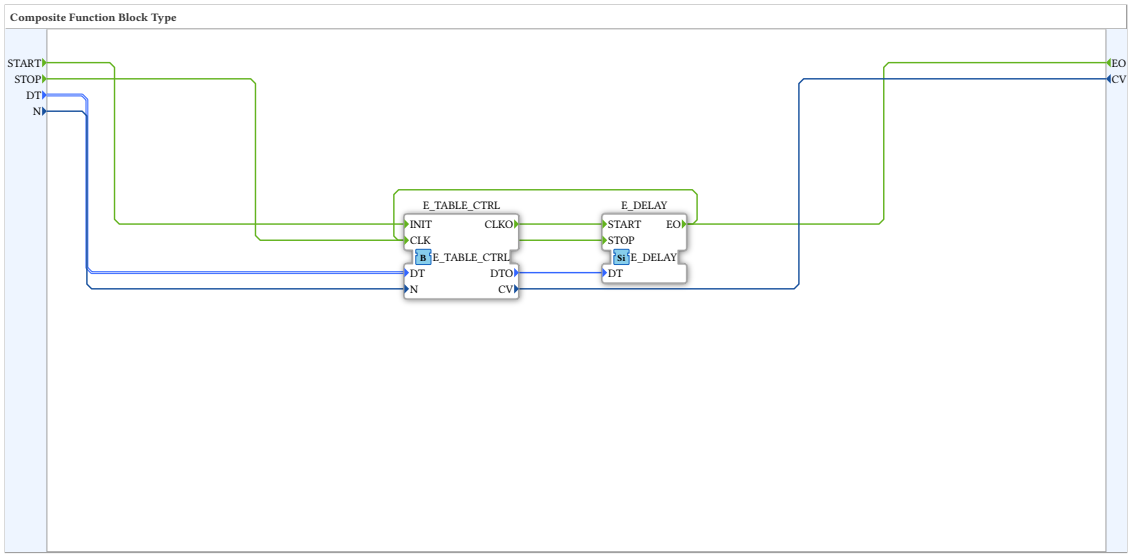
Application Scenarios

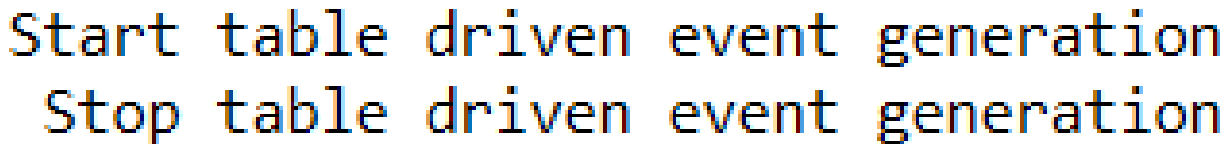
⚖️ Vergleich mit ähnlichen Bausteinen

🛠️ Zugehörige Übungen

Conclusion

2.11.3.55 E_TABLE

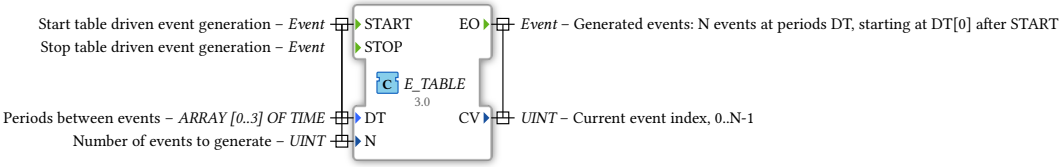




• – ◦ • – ◦ • – ◦ •

Introduction

The `E_TABLE` (Event Table) is a function block according to IEC 61499 that generates a finite sequence of events with individually definable time intervals. It reads the time intervals from a table (an array) and fires a specified number of events sequentially.



Interface Structure

Event Inputs

- **START**: Starts the generation of the event sequence.
- **Connected Data**: `DT`, `N`
- **STOP**: Stops the sequence prematurely.

Event Outputs

- **EO (Event Output)**: The output event that is triggered N times.
- **Associated Data**: DT, N

Data Inputs

- **DT**: An array of durations (data type: `TIME`, size: 4). `DT[i]` defines the delay time *before* the ith event.
- **N**: The total number of events to be generated (data type: `UINT`, max. 4 for this function block).

Data Outputs

- **CV (Current Value)**: The index of the currently triggered event (0 to N-1) (Data type: `UINT`).

Functionality

1. **Start of the sequence**: A `START` event triggers the function block. The number of events to be generated is determined by `N`.
2. **Event generation**: The function block processes the table:

- It waits for the time interval defined in `DT[0]`. Then, the first `EO` event is triggered, and `CV` is set to `0`.
- It waits for the time interval defined in `DT[1]`. Then, the second `EO` event is triggered, and `CV` is set to `1`.
- This process repeats until `N` events have been generated.

1. **End of Sequence**: The sequence ends automatically after `N` events have been triggered.
2. **Stop**: A `STOP` event immediately terminates the sequence at any point.

Example:

- `N` = 3
- `DT` = `[T#2s, T#5s, T#1s]`
- After a `START` event:

1. After 2 seconds, `EO` is triggered with `CV=0`.
2. 5 seconds later, `EO` is triggered with `CV=1`.
3. 1 second later, `EO` is triggered with `CV=2`.
4. The sequence is complete.

Technical Features

- **Table-Controlled Time Intervals**: The strength of this function block lies in its ability to define variable time intervals between events, rather than a fixed interval.
- **Sequence Counter**: The `CV` output provides valuable information about the current position in the sequence.

Application Scenarios

- **Control of Step Sequences**: Triggering steps with varying wait or processing times.
- **Complex Control**: Controlling an actuator with a series of pulses with variable pauses.
- **Test Automation**: Generating a complex, time-defined stimulus sequence for a test object, where the `CV` output can be used for synchronization with the evaluation.

⚖️ Comparison with Similar Function Blocks

Feature	E_TABLE	E_CYCLE	E_N_TABLE
Event Interval	Variable (Array `DT`)	Fixed (`DT`)	Variable (Array `DT`)
Outputs	One `EO` channel	One `EO` channel	Multiple (`EO0..EO3`)
Counter Output (`CV`)	Yes	No	No (internal)

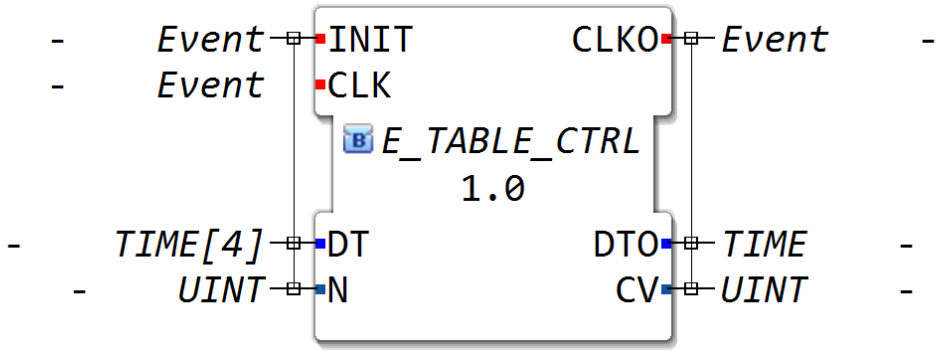
🛠️ Related Exercises

- Exercise_093

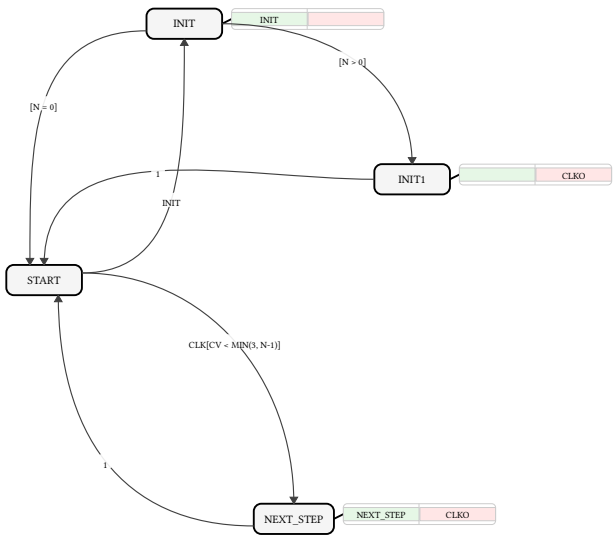
Conclusion

The `E_TABLE` function block is a highly flexible timer for generating complex, finite event sequences. Its ability to read variable time intervals from a table makes it ideal for non-periodic, sequential control tasks. It forms the basis for other function blocks such as `E_N_TABLE`, but unlike the latter, it provides the current counter value as useful output information.

2.11.3.56 E_TABLE_CTRL



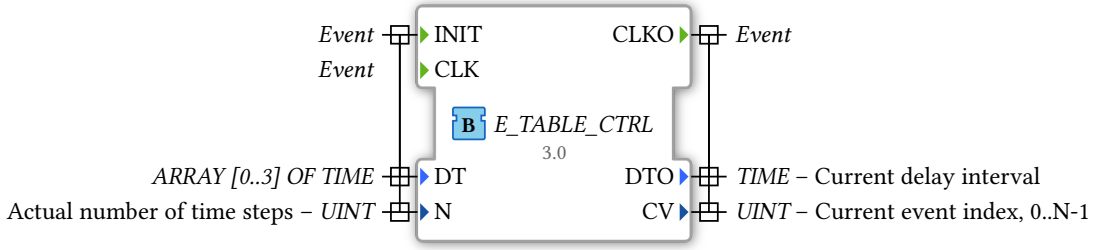
ECC: E_TABLE_CTRL



• – ○ • – ○ • – ○ •

Introduction

The **E_TABLE_CTRL** is a support function block for E_TABLE according to IEC 61499-1 (Annex A), under the EPL-2.0 license. Version 1.0 enables precise control of event sequences based on a configurable time table.



Interface Structure

Event Inputs

- `INIT`: Initializes the event table (with DT and N parameters)
- `CLK`: Clock signal for table increment

Event Outputs

- `CLKO`: Generated clock event (with DTO and CV data)

Data Inputs

- `DT` (TIME array): Time intervals for event generation
- `N` (UINT): Number of active time steps

Data Outputs

- `DTO` (TIME): Current time interval
- `CV` (UINT): Current event index (0..N-1)

Functionality

1. **Initialization**:

- On the `INIT` event, the index (CV) is set to 0
- The first time interval (DTO) is taken from the DT array

1. **Table Control**:

- Each `CLK` event increments CV by 1
- The next time interval from the DT array is loaded
- `CLKO` is generated at each step

1. **State Machine** (ECC):

- **START**: Wait state
- **INIT**: Initialization phase
- **INIT1**: First event generation
- **NEXT_STEP**: Table progress

Technical Features

✓ **Table-driven** Scheduling ✓ **Array-based** configuration (up to 4 time steps) ✓ **State-based** implementation (BasicFB) ✓ **Real-time capable** event generation

Application Scenarios

- **Process control**: Complex timing sequences
- **Test automation**: Programmable test sequences
- **Machine control**: Motion sequences
- **Production lines**: Cycle-controlled processes

Relationship to E_TABLE

The `E_TABLE_CTRL` block is not intended as a standalone block for direct use, but rather as the **internal control logic** of the composite function block `E_TABLE`.

Within `E_TABLE`, `E_TABLE_CTRL` interacts with a `E_DELAY` block:

1. `E_TABLE_CTRL` receives the `START` command and calculates the first delay time, `DT0`.
2. It sends `DT0` via `CLK0` to the `E_DELAY` block.
3. `DT0` to `CLK0`.
4. `E_DELAY` 3. After `E_DELAY` has expired, it signals this via its `EO` output back to the `CLK` input of `E_TABLE_CTRL`.
5. `E_TABLE_CTRL` then calculates the next delay time, and the cycle repeats.

This block thus encapsulates the pure state logic (which step is next, how long it takes), while `E_DELAY` executes the actual time delay.

This block therefore encapsulates the pure state logic (which step is next, how long it takes), while `E_DELAY` performs the actual time delay.

🛠️ Related Exercises

- Exercise_175

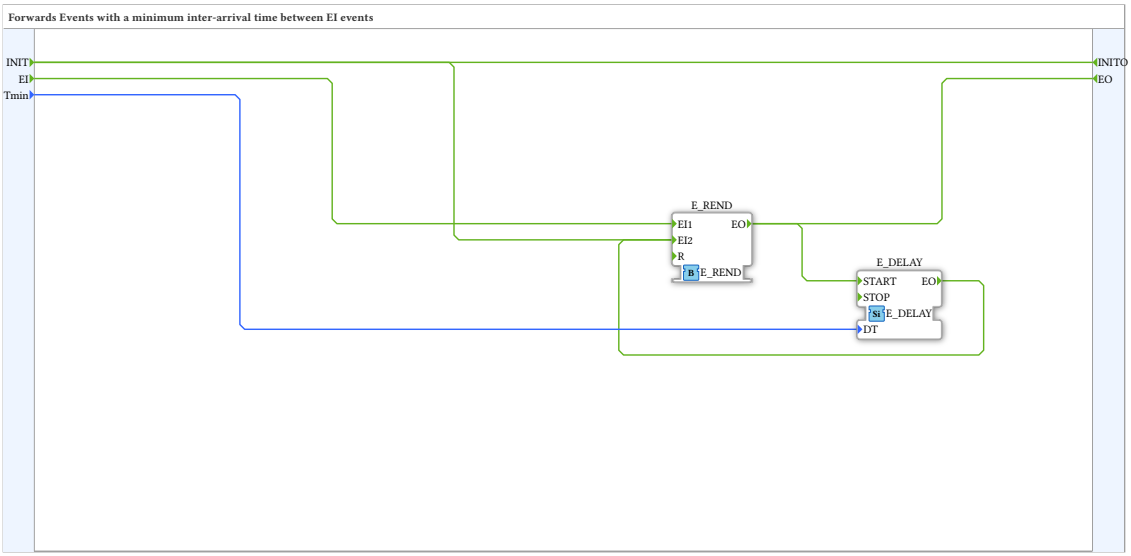
Conclusion

The E_TABLE_CTRL function block extends the possibilities of table-driven event generation:

- Flexible configuration of multiple time intervals
- Precise control of complex processes
- Robust state machine implementation

Thanks to its array-based timing, it is ideally suited for applications with variable process steps. Integration as a BasicFB ensures reliable operation in IEC 61499-based control systems.

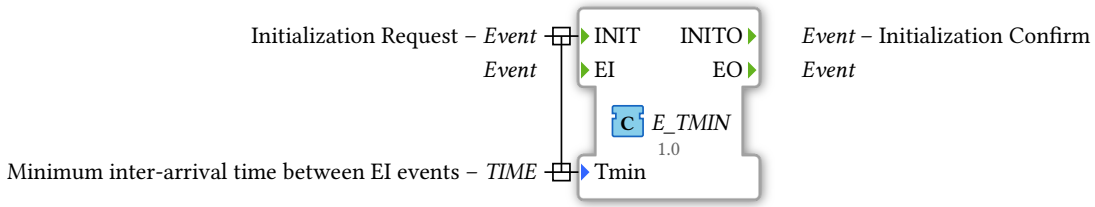
2.11.3.57 E_TMIN



• — ○ • — ○ • — ○ •

Introduction

`E_TMIN` (minimum inter-arrival time) is a composite function block that only forwards incoming events at `EI` to `EO` if at least the time span `Tmin` has passed since the last forwarded event. It acts as an event throttle (rate limiter) for arbitrary event streams.



Interface Structure

Event Inputs

- **INIT**: Initialization, carries `Tmin`.
- **EI**: Incoming event to be throttled.

Event Outputs

- **INITO**: Confirms the initialization.
- **EO**: The throttled output event.

Data Inputs

- **Tmin** (TIME): Minimum spacing between two consecutive `EI` events that actually result in `EO`.

Functionality

Internally, `E_TMIN` uses an `E_REND` (rendezvous of two events) and an `E_DELAY`: when `EI` arrives, it is forwarded to `E_REND.EI1`. `E_REND` fires `EO` once it has received at least one event on both `EI1` and `EI2` — initially `EI2` is already "pre-filled" via `INIT`, so the first `EI` is passed through immediately. Every event let through by `E_REND` simultaneously starts `E_DELAY` with time `Tmin`; only once it elapses does it deliver the next `EI2` event to `E_REND`, meaning the next `EI` can pass through at the earliest after `Tmin`.

Technical Features

- **Built from two standard elements**: `E_TMIN` is pure wiring composed of `E_REND` and `E_DELAY`, with no ECC of its own.
- **First event unthrottled**: The first `EI` after `INIT` is passed through immediately; throttling only kicks in afterwards.
- **Building block for the `_TMIN` family**: Used internally by E_D_FF_TMIN and E_D_FF_ANY_TMIN to additionally throttle their `EO` outputs.

State Overview

Stateless wiring logic: `E_REND` internally remembers whether an `EI2` (from `E_DELAY`) has already arrived, `E_DELAY` restarts after every event it lets through.

Application Scenarios

- **Throttling rapidly changing sensor events** so downstream logic (e.g. logging, network transmission) isn't overloaded by too high an event rate.
- **Building block inside E_D_FF_TMIN/E_D_FF_ANY_TMIN** to additionally time-limit change events.

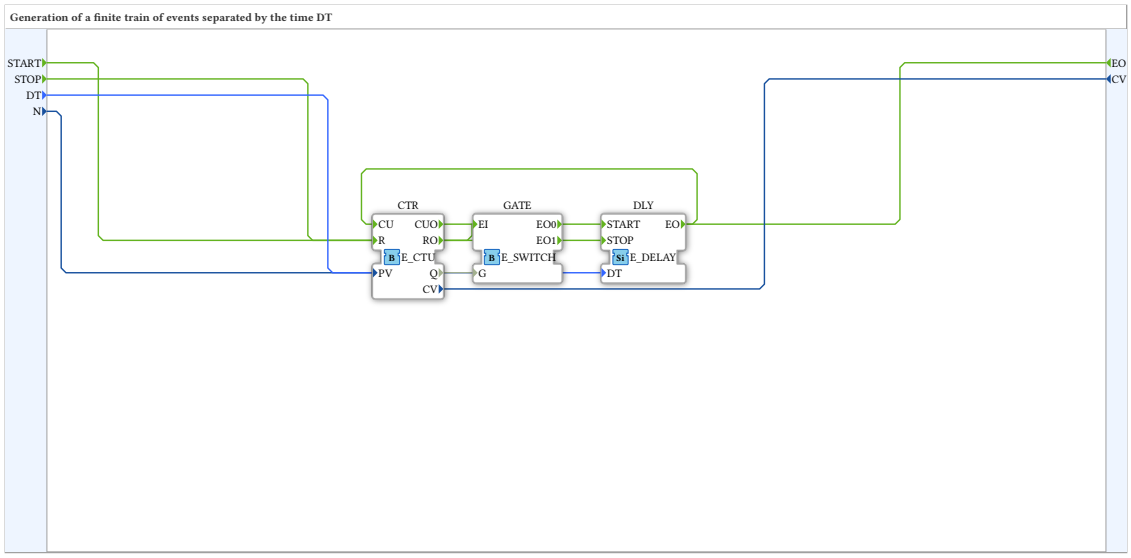
Comparison with similar function blocks

- **`E_REND`**: the internally used synchronization basis, without a time component.
- **`E_DELAY`**: the internally used timer.
- E_D_FF_TMIN / E_D_FF_ANY_TMIN: additionally combine `E_TMIN` with a data latch (`E_D_FF`/`E_D_FF_ANY`).

Conclusion

`E_TMIN` is a simple but effective event-throttling block composed of two standard elements (`E_REND`, `E_DELAY`) that guarantees a minimum spacing time between passed-through events.

2.11.3.58 E_TRAIN



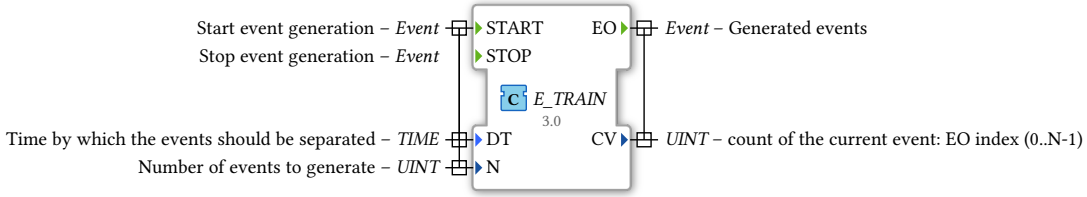
Start event generation -
Stop event generation -

Time by which the events should be seperated -
Number of events to generate -

• – ◦ • – ◦ • – ◦ •

Introduction

The `E_TRAIN` (Event Train) is a function block according to IEC 61499 that generates a finite sequence of events (“event train”) at fixed time intervals. After starting, `N` events are generated at intervals of `DT`.



Interface Structure

Event Inputs:

- **START**: Starts the generation of the event sequence.
- **Related Data**: `DT`, `N`
- **STOP**: Prematurely stops the ongoing generation of the event sequence.

Event Outputs:

- **EO (Event Output)**: The output event that is triggered periodically.
- **Related Data**: `CV`

Data Inputs:

- **DT (Delay Time)**: The time interval between each `EO` event (Data type: `TIME`).
- **N**: The total number of events to be generated (Data type: `UINT`).

Data Outputs:

- **CV (Current Value)**: The counter value indicating the index of the currently triggered event (0 to N-1) (Data type: `UINT`).

Functionality

1. **Start of the Sequence**: A `START` event triggers the function block. An internal counter is reset, and the parameters `N` (count) and `DT` (time interval) are applied. The timer for the first event is started.
2. **Event Generation**:

- After the time specified in `DT` has elapsed, the first `EO` event is triggered. The output of `CV` has the value `0`.
- Immediately afterward, the timer for the next event is restarted.
- After another time interval `DT`, the next `EO` event is triggered with `CV=1`.

1. **End of Sequence**: This cycle repeats until `N` events have been generated. After the last event (with `CV = N-1`) is triggered, the function block stops automatically.
2. **Stop**: A `STOP` event can terminate the sequence prematurely at any time.

Technical Features

- **Pulse Generator**: Generates a fixed number of pulses at constant intervals.
- **Internal Counter**: The module uses an internal counter (`E_CTU`) to track the number of generated events.
- **Sequence Counter**: The output (`CV`) provides valuable information about the current position in the sequence.

Application Scenarios

- **Stepper Motor Control**: Generates an exact number of steps (`N`) at a defined speed (determined by `DT`).
- **Dosing Processes**: Triggers dispensing processes (`N`) at intervals of `DT`.
- **Test Sequences**: Sending a defined number of test triggers to a device under test.

⚖️ Comparison with similar building blocks

Feature	E_TRAIN	E_CYCLE	E_TABLE
Event Interval	Fixed (`DT`)	Fixed (`DT`)	Variable (Array `DT`)
Event Count	Finite (`N`)	Infinite	Finite (`N`)
Counter Output (`CV`)	Yes	No	Yes

🛠️ Related Exercises

- Exercise_035a2
- Exercise_035a2_AX
- Exercise_091

Conclusion

The `E_TRAIN` module is ideal for applications requiring a fixed number of events within a constant time interval. It combines the functionality of a clock (`E_CYCLE`) with a counter to generate a controlled, finite pulse sequence. The `CV` output provides useful feedback on the sequence's progress.

🌐 Related topic subpages on ms-muc-docs.de

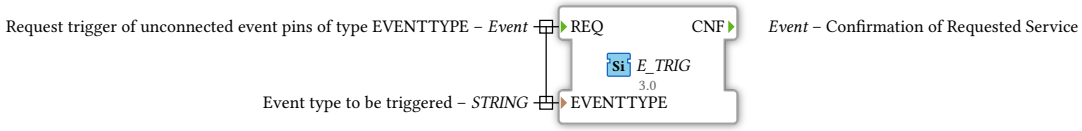
- 🌐 E_CTU Event Counter module on ms-muc-docs.de

2.11.3.59 E_TRIG

• – ◦ • – ◦ • – ◦ •

Introduction

The `E_TRIG` is a special service function block according to IEC 61499. Its function is to programmatically trigger *unconnected* event inputs of a specific type (name) within a resource. This enables advanced control and initialization scenarios.



Interface Structure

Event Inputs:

- **REQ (Request)**: Triggers the trigger mechanism.
- **Connected Data**: `EVENTTYPE`

Event Outputs:

- **CNF (Confirmation)**: Confirms that the trigger command has been issued.

Data Inputs:

- **EVENTTYPE**: The name of the event inputs to be triggered (data type: `STRING`).
- **Default Value**: 'EInit'

Functionality

1. **Trigger Request**: Upon a `REQ` event, the function block reads the string at the `EVENTTYPE` input.
2. **Internal Triggering**: The function block instructs the 4diac runtime environment (the resource) to find all *unconnected* event inputs of function blocks within the same resource whose name matches the `EVENTTYPE` string.
3. **Event Generation**: The runtime environment triggers these found event inputs.
4. **Acknowledgement**: Immediately after the command is issued, the `E_TRIG` block triggers its `CNF` event.

Example: Suppose several function blocks in a resource have an unconnected input named `INIT`. A `E_TRIG` block with `EVENTTYPE = 'INIT'` would, upon a `REQ` event, trigger all these `INIT` inputs simultaneously to force a re-initialization of these blocks.

Technical Features

- **Service Block**: This is not a simple logic block, but an interface to a special service of the 4diac runtime environment.

Example: - **Effects Unconnected Inputs**: This functionality explicitly targets inputs that are not graphically wired in the network. This is a powerful feature, but it should be used with caution.

- **Default Use Case `EInit`**: The default value 'EInit' indicates the primary use case: manually triggering initialization events that are normally executed only once when a resource starts.

Application Scenarios

- **Manual Re-Initialization**: Resetting parts of an application to a defined initial state without having to restart the entire resource.
- **Dynamic Configuration**: After loading a new configuration, relevant components can be made to adopt the new parameters via a targeted `E_TRIG` call.
- **Testing and Debugging**: Manually triggering specific events in a running application for testing purposes.

🛠️ Related Exercises

- Exercise_176

Conclusion

The `E_TRIG` building block is a tool for advanced use cases and for developers with in-depth knowledge of the 4diac execution environment. It offers a powerful method to bypass the standard, graphically defined

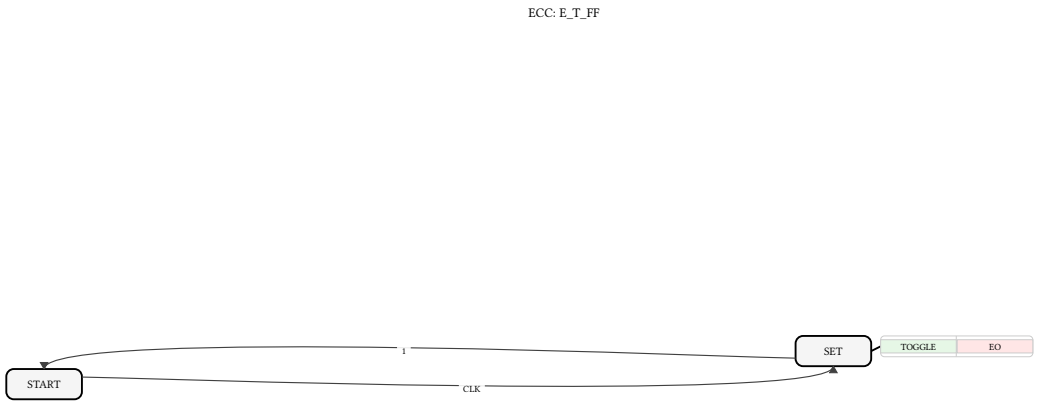
event flow logic and interact directly with the runtime environment. It should be used with caution, as its effect is not immediately apparent from the graphical network.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

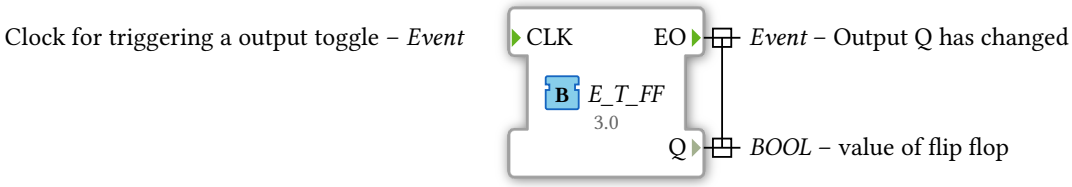
2.11.3.60 E_T_FF

🎧 Podcast



- The E_T_FF in IEC 61499: Modular Toggle for Industry 4.0
- The E_T_FF_SR Component: Heart of IEC 61499 – Storing, Switching, Responding
- Unpacking E_T_FF_SR: The Secret Toggle Switch of Industrial Control Systems

The `E_T_FF` (Event-driven Toggle Flip-Flop) is an event-driven toggle switch that changes its state (`Q`) with each incoming clock event (`CLK`). It is the digital equivalent of an impulse relay, where a short pulse permanently changes the state.



- **CLK (Clock)**: The clock event that toggles the state of `Q`.
- **EO (Event Output)**: Triggered when the state of `Q` changes.
- **Associated Data**: `Q`
- **Q**: The current state of the flip-flop (data type: `BOOL`).

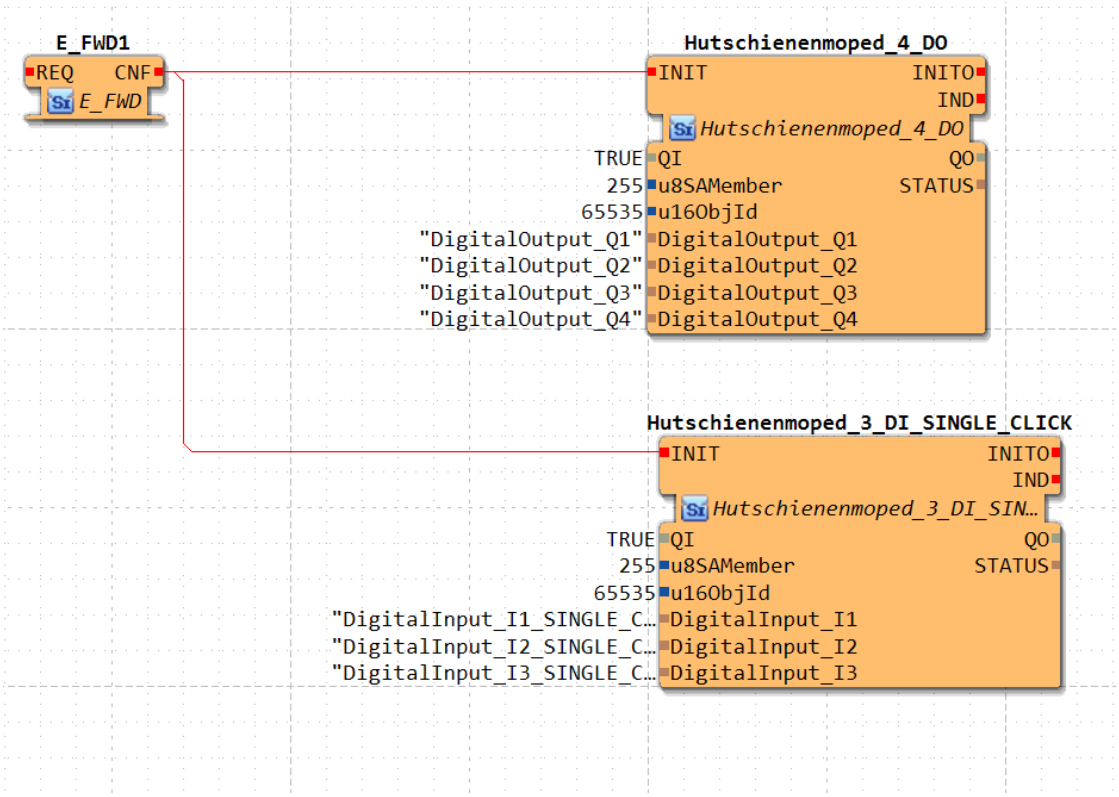
The `E_T_FF` component is a simple toggle flip-flop:

1. **State Change**: Upon each incoming `CLK` event, the output `Q` changes its state: If `Q` was `TRUE`, it becomes `FALSE`, and if `Q` was `FALSE`, it becomes `TRUE`.
2. **Event Triggering**: Each state change of `Q` triggers the `EO` event.

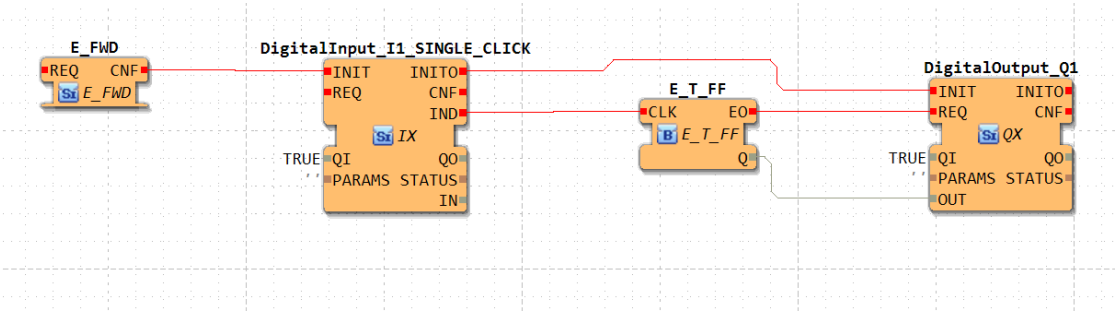
- **Impulse Switch Analogy**: The component behaves like an impulse switch: A short pulse (`CLK`) switches the light (`Q`) on, the next pulse switches it off.
- **Memory Function**: `Q` stores the last state of the flip-flop.
- **Stateless Between Clock Cycles**: Changes to the `CLK` input only affect `Q` at the time of the event.

A push button logic for a lamp can be implemented using a `E_T_FF`:

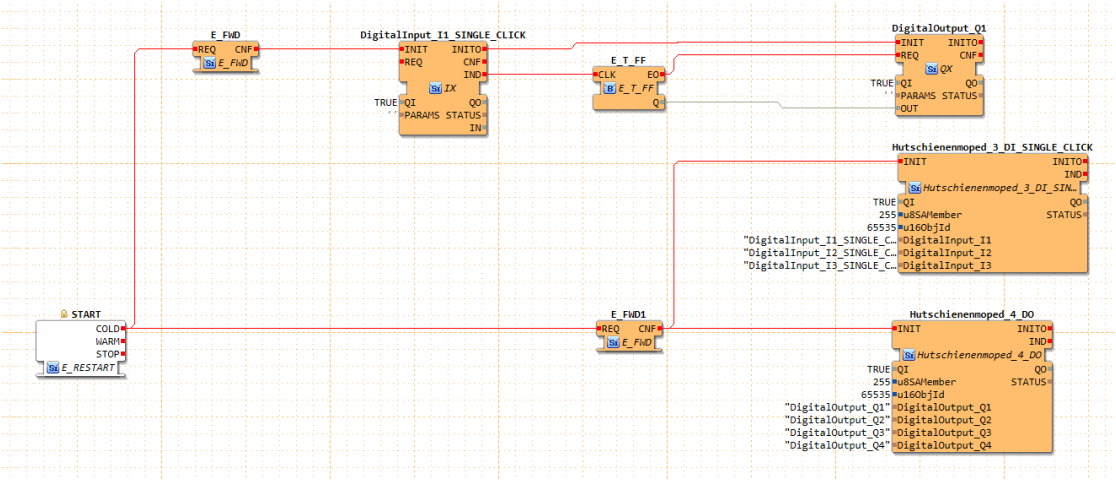
- **Concept**: A push button generates a `CLK` event. Each press turns the lamp (`Q`) on or off.
- **Graphical Representation**:
- Mapping:



- Application:



- Embedded Resource:



Taster für eine Lampe

Technical Features

Anwendungsbeispiele

Data Outputs

Event Outputs

Event Inputs

Interface Structure

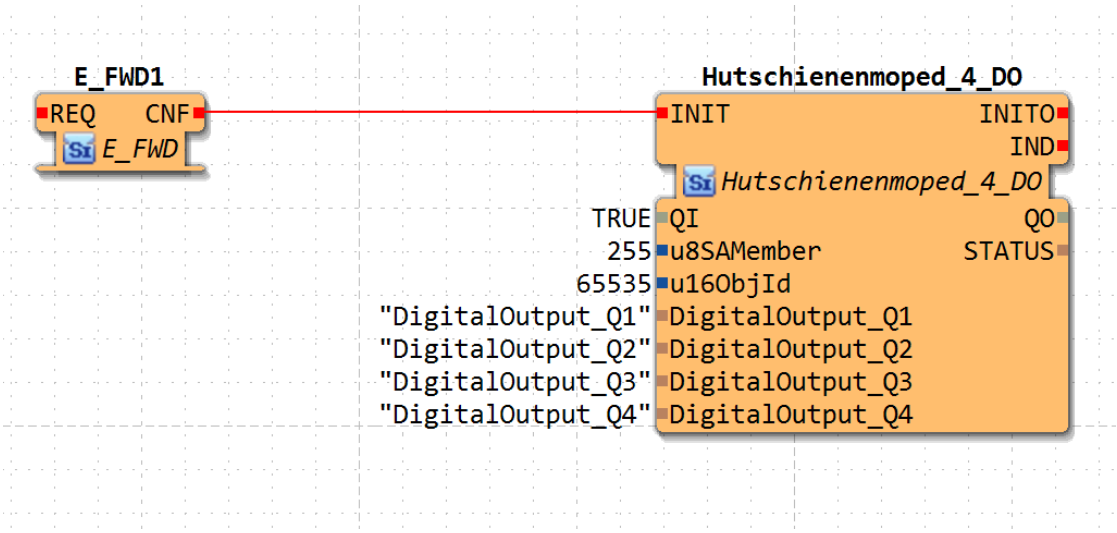
Introduction

Functionality

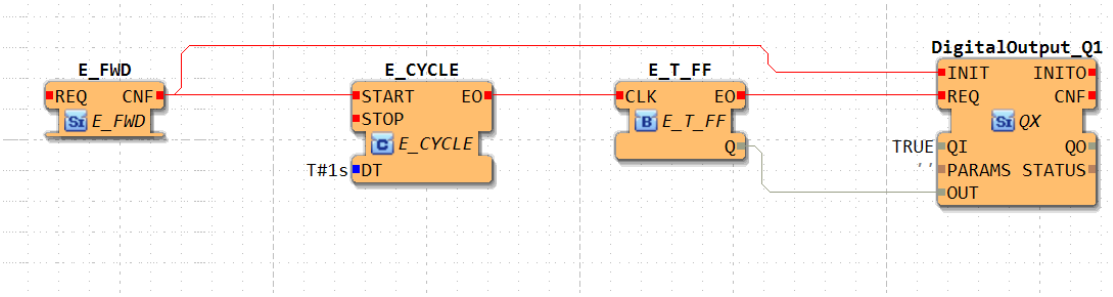
Blinker

A blinker can be implemented using feedback with a timer:

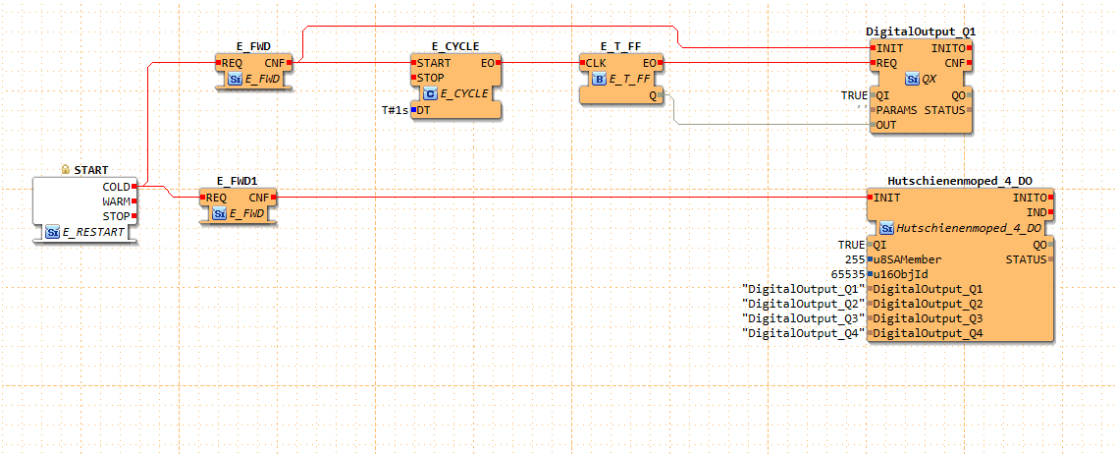
- **Concept**: The `EO` event of `E_T_FF` triggers a `E_DELAY` event, whose `EO` event, in turn, serves as the `CLK` event for `E_T_FF`. This creates a periodic state change.
- Mapping:



- Application:



- Embedded resource:



- Uebung_004a
- Uebung_004a2
- Uebung_004a2_2
- Uebung_004a2_3
- Uebung_004a3
- Uebung_004a4
- Uebung_004a5
- Uebung_004a6
- Uebung_004a8
- Uebung_004a9
- Uebung_004c1
- Uebung_004c2
- Uebung_004c3
- Uebung_004c4
- Uebung_004c5
- Uebung_004c6
- Uebung_004c7
- Uebung_005
- LinksRechts_T_FF_Event
- Uebung_007
- Uebung_007a1
- Uebung_007a2
- Uebung_010b2
- Uebung_010b3
- Uebung_010b6
- Uebung_010b7
- Uebung_010b8
- Uebung_010b9
- Uebung_010bA
- Uebung_010bA2
- Uebung_010bA3
- Uebung_010bA4
- Uebung_035a2

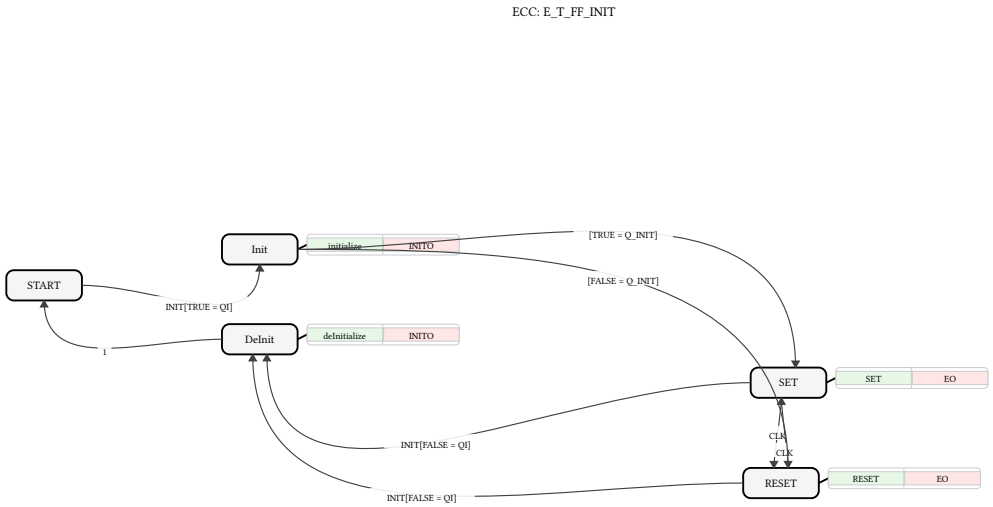
- Uebung_080c
- Uebung_087
- Uebung_087a1
- Uebung_088
- Uebung_089
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Exercise_094a
- Exercise_095
- Exercise_152
- Exercise_153

The `E_T_FF` module is a simple yet extremely versatile memory module. Its impulse relay function is fundamental for implementing push-button logic, flashers, and other bistable control circuits in IEC 61499 applications.

🛠️ Zugehörige Übungen

Conclusion

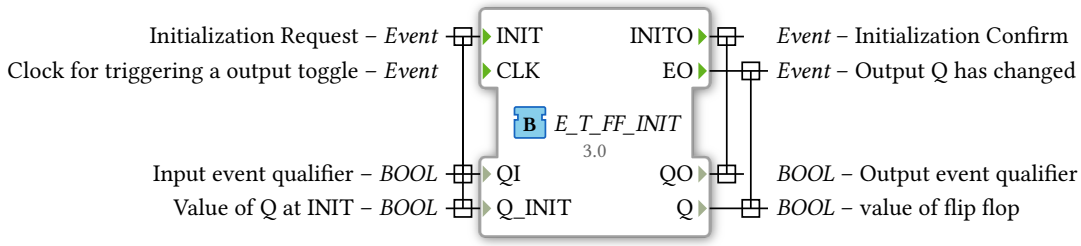
2.11.3.61 E_T_FF_INIT



• — ○ • — ○ • — ○ •

Introduction

`E_T_FF_INIT` (toggle flip-flop with initialization) combines the toggling behaviour of a toggle flip-flop (output `Q` switches state on every `CLK` event) with an explicit `INIT`/`INITO` interface for setting a defined start value.



Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `QI` and `Q_INIT`.
- **CLK**: Clock event, triggers a state change of `Q`.

Event Outputs

- **INITO**: Confirms (de-)initialization, carries `QO`.
- **EO**: Triggered on every `CLK` event, carries `Q`.

Data Inputs

- **QI** (BOOL): Input event qualifier — `TRUE` initializes, `FALSE` deinitializes.
- **Q_INIT** (BOOL): The value `Q` is set to during initialization.

Data Outputs

- **QO** (BOOL): Output event qualifier, mirrors `QI`.
- **Q** (BOOL): The current state of the flip-flop.

Functionality

`INIT` with `QI = TRUE` initializes `Q` via `Q_INIT` (states `Init` ⟶ `SET`/`RESET`, depending on `Q_INIT`). During normal operation, every `CLK` event flips the state: from `SET` to `RESET` and back, each confirmed via `EO` carrying the current `Q`. `INIT` with `QI = FALSE` deinitializes the block (`DeInit` ⟶ `START`).

Technical Features

- **Toggle instead of set/reset**: Unlike E_RS_SYM_INIT/E_SR_SYM_INIT, which have separate `S`/`R` inputs, `E_T_FF_INIT` has only a single clock input `CLK` that toggles between `SET` and `RESET` on every event.
- **Same INIT/DeInit structure** as the `_SYM_INIT` blocks: `QI` switches between initialization and deinitialization, `Q_INIT` determines the start value.

State Overview

State	Meaning
START	Unconfigured initial state
Init	Initialization in progress, `QO := QI`
DeInit	Deinitialization in progress, `QO := FALSE`
SET	`Q = TRUE`, switches to `RESET` on `CLK`
RESET	`Q = FALSE`, switches to `SET` on `CLK`

Application Scenarios

- **Blink logic with a defined start state**: A signal should toggle on every clock tick, but start with a known initial value (`Q_INIT`) after system startup instead of a random state.
- **Frequency division**: `E_T_FF_INIT` halves the event frequency from `CLK` to `EO` (returns to the same state every second `CLK`), with a controlled start value.

Comparison with similar function blocks

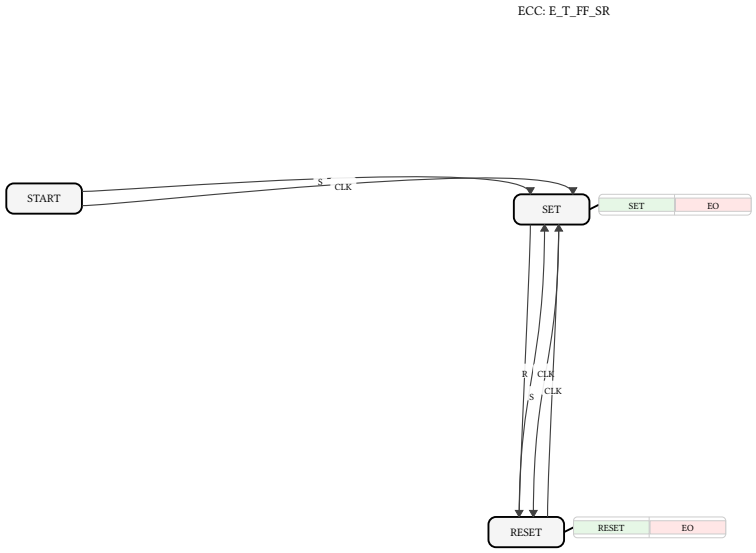
- **`E_T_FF`**: the same basic toggle functionality without an `INIT`/`INITO` interface.
- E_RS_SYM_INIT / E_SR_SYM_INIT: the same INIT/DeInit structure, but with separate set/reset inputs instead of a single clock input.
- E_T_FF_SR_SYM / E_T_FF_SR_SYM_INIT: additionally combine `S`/`R` inputs with the toggle behaviour.

Conclusion

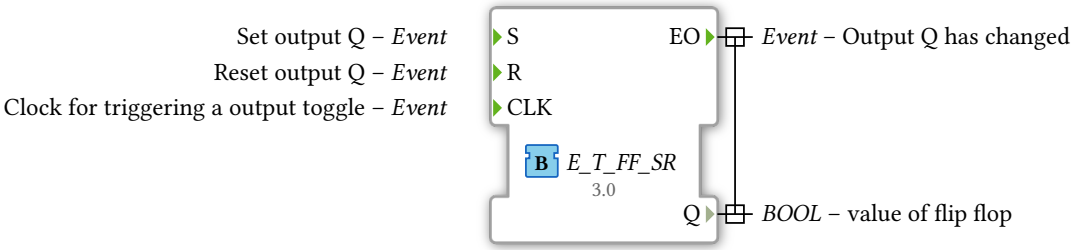
`E_T_FF_INIT` provides a toggle flip-flop with a configurable, reliably defined start value and is suitable for blink and frequency-divider logic where the state after system startup must not be left to chance.

2.11.3.62 E_T_FF_SR

🎧 Podcast



- The E_T_FF_SR Block: The Heart of IEC 61499 – Storing, Switching, Responding
- Unpacking E_T_FF_SR: The Secret Toggle Switch of Industrial Control Systems The `E_T_FF_SR` (Event-driven Toggle Flip-Flop with Set/Reset) is an extended bistable function block according to IEC 61499. It combines the functionality of a `E_T_FF` (Toggling) with additional `S` (Set) and `R` (Reset) inputs.



- **S (Set)**: Sets the output `Q` to `TRUE`.
- **R (Reset)**: Sets the output `Q` to `FALSE`.
- **CLK (Clock)**: Triggers a toggle of the output `Q`.
- **EO (Event Output)**: Triggered when the state of `Q` changes.
- **Associated Data**: `Q`
- **Q**: The current state of the flip-flop (data type: `BOOL`).

Data Outputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

The E_T_FF_SR is a stateful component whose output Q is influenced by three event inputs:

1. **Set (S)**: When a S event occurs, Q is set to TRUE. If Q was previously FALSE, EO is triggered.
2. **Reset (R)**: When a R event occurs, Q is set to FALSE. If Q was previously TRUE, EO is triggered.
3. **Toggling (CLK)**: When a CLK event occurs, Q is toggled. If Q changes its state, EO is triggered.

The function block starts in the START state (implies Q is undefined/FALSE).

- If the first event is S, Q is set to TRUE.
- **If the first event is CLK, Q is set to TRUE** (not toggled from FALSE to TRUE). Subsequent CLK events will then toggle normally.
- **Asynchronous Set/Reset**: The S and R inputs can overwrite the state of Q at any time.
- **Toggle Function**: The CLK input allows for easy state switching.
- **No Prioritization (Dominance)**: As with E_RS and E_SR, there is no fixed priority when S, R, or CLK arrive simultaneously. The processing order of the 4diac runtime environment determines the final state.
- **Initial Behavior with CLK**: Upon receiving a CLK event from the START state, the function block is initially set (Q=TRUE) instead of toggling. This should be considered during system initialization.
- **Control with Manual Override**: A toggle switch (CLK) for a lamp, which can be switched directly on (S) or off (R) if needed (e.g., for safety reasons).
- **Mode Switching**: Switch between different modes (CLK), with the option to directly access a basic mode (R) or a special mode (S).
- **Error Reset and Toggle**: An error state can be set using S, acknowledged using R, and the error handling state can be toggled using CLK.
- Exercise_004a7
- Exercise_006a
- Exercise_006a2
- Exercise_006a3
- Exercise_006a4
- Exercise_179
- Exercise_180

The E_T_FF_SR block offers maximum flexibility for memory and control tasks by combining the toggle function with direct set and reset capabilities. combined. The specific behaviors of the START state and the lack of a prioritization guarantee for simultaneous events must be carefully considered during implementation.

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Besonderheiten des Verhaltens aus dem `START`-Zustand

Technical Features

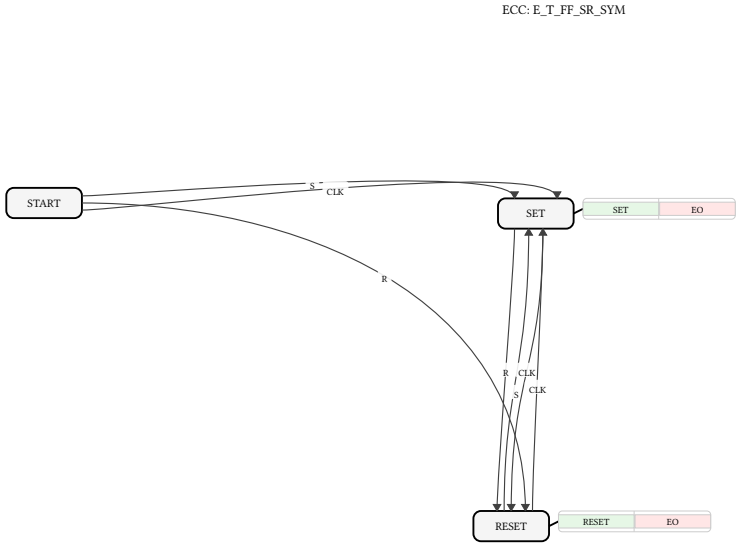
Application Scenarios

🛠️ Zugehörige Übungen

Conclusion

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

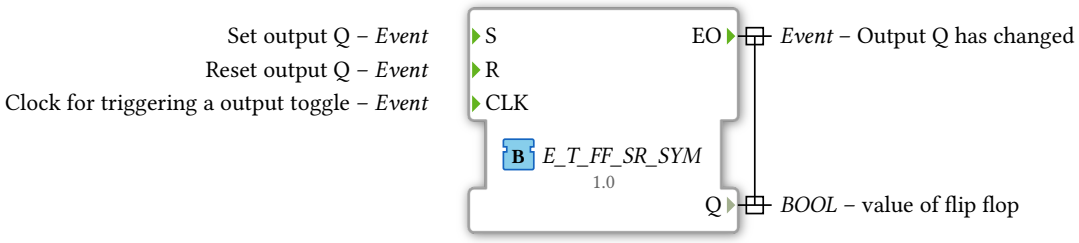
2.11.3.63 E_T_FF_SR_SYM



• – ○ • – ○ • – ○ •

Introduction

`E_T_FF_SR_SYM` combines the functionality of E_RS_SYM (bistable set/reset with symmetric start-up) and a toggle flip-flop in a single block: in addition to the `S` and `R` inputs, it has a clock input `CLK` that toggles output `Q` independently of `S`/`R`.



Interface Structure

Event Inputs

- **S (Set)**: Sets `Q` to `TRUE`.
- **R (Reset)**: Sets `Q` to `FALSE`.

- **CLK**: Inverts the current state of Q (toggle).

Event Outputs

- **EO**: Triggered after every S, R, or CLK event, carries Q.

Data Outputs

- **Q** (BOOL): The current state.

Functionality

From the initial state START, S, R, **and** CLK all lead to a defined follow-up state (CLK leads to SET) — the symmetric start-up behaviour of E_RS_SYM applies here to all three events. During normal operation (states SET/RESET), S switches to SET, R to RESET, and CLK switches to the *opposite* state each time (toggle) — independent of the preceding S/R call. Every transition sets Q accordingly and triggers EO.

Technical Features

- **Combined set/reset/toggle behaviour**: Unlike E_T_FF, which only toggles, Q can here be set deliberately via S/R as well as toggled via CLK.
- **Symmetric start-up for all three inputs**: START reacts to S, R, and CLK equally well-defined.
- **No init mechanism**: For a configurable start value, see E_T_FF_SR_SYM_INIT.

State Overview

State	Meaning
START	Initial state, reacts symmetrically to S, R, CLK
SET	Q = TRUE; R⟶RESET, CLK⟶RESET
RESET	Q = FALSE; S⟶SET, CLK⟶SET

Application Scenarios

- **Manual operation with clock override**: A state can be either deliberately set/reset (S/R, e.g. by an operator) or toggled by a clock (CLK, e.g. by a timer), without needing two separate blocks.
- **Blink logic with override**: A blink signal (CLK) can be overridden at any time by an explicit S/R.

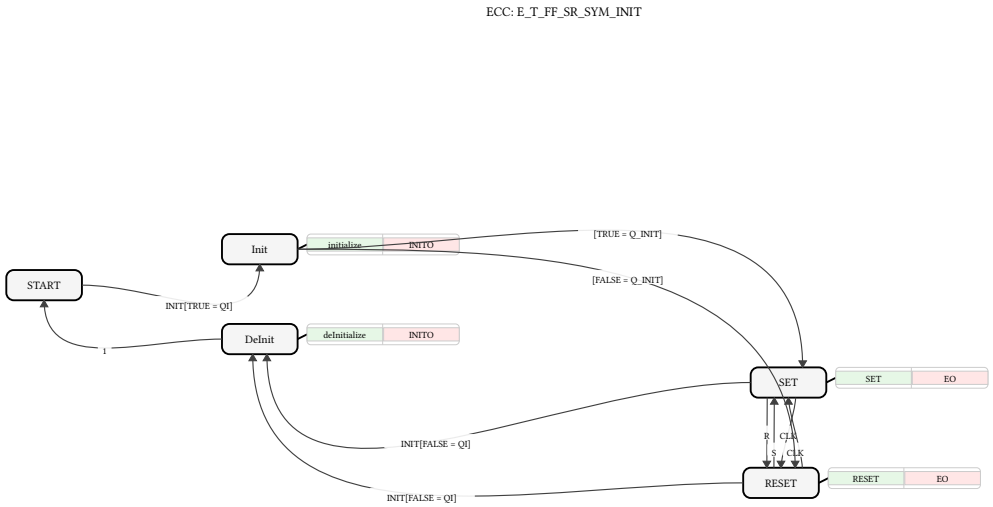
Comparison with similar function blocks

- **E_RS_SYM**: the same set/reset logic, but without the CLK toggle input.
- **E_T_FF**: pure toggle block without S/R.
- **E_T_FF_SR_SYM_INIT**: the same functionality, extended with INIT/INITO.

Conclusion

E_T_FF_SR_SYM combines deliberate set/reset with toggle functionality in a single block with guaranteed, well-defined start-up behaviour, making it suitable wherever both modes of operation are needed at once.

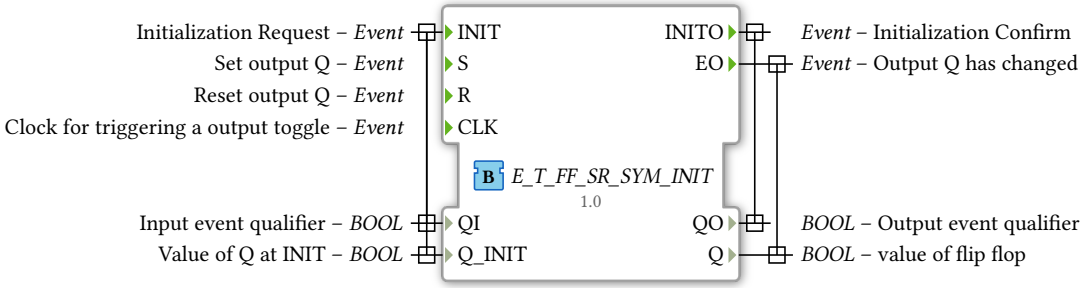
2.11.3.64 E_T_FF_SR_SYM_INIT



• – ○ • – ○ • – ○ •

Introduction

`E_T_FF_SR_SYM_INIT` extends E_T_FF_SR_SYM with an explicit `INIT`/`INITO` interface: the start value of `Q` is not determined by the first `S`, `R`, or `CLK` event, but is set deliberately via `Q_INIT`.



Interface Structure

Event Inputs

- **INIT**: Initialization request, carries `QI` and `Q_INIT`.
- **S (Set)**: Sets `Q` to `TRUE`.
- **R (Reset)**: Sets `Q` to `FALSE`.
- **CLK**: Inverts the current state of `Q`.

Event Outputs

- **INITO**: Confirms (de-)initialization, carries `QO`.
- **EO**: Triggered after every `S`, `R`, or `CLK` event, carries `Q`.

Data Inputs

- **QI** (BOOL): Input event qualifier — `TRUE` initializes, `FALSE` deinitializes.
- **Q_INIT** (BOOL): The value `Q` is set to during initialization.

Data Outputs

- **QO** (BOOL): Output event qualifier.
- **Q** (BOOL): The current state.

Functionality

`INIT` with `QI = TRUE` initializes `Q` via `Q_INIT` (`Init` ⟶ `SET`/`RESET`). During normal operation, `S`/`R` switch deliberately, `CLK` toggles between `SET` and `RESET` — identical to E_T_FF_SR_SYM, but without its symmetric start-up behaviour from `START`, since the start value instead comes from `INIT`. `INIT` with `QI = FALSE` deinitializes the block back to state `START`.

Technical Features

- **Combines all three mechanisms**: deliberate set/reset (`S`/`R`), toggle (`CLK`), and configurable initialization (`INIT`/`Q_INIT`) in a single block.
- **Same INIT/DeInit structure** as E_RS_SYM_INIT, E_SR_SYM_INIT, and E_T_FF_INIT.

State Overview

State	Meaning
START	Unconfigured initial state
Init	Initialization in progress, `QO := QI`
DeInit	Deinitialization in progress, `QO := FALSE`
SET	`Q = TRUE`; `R`/`CLK`⟶RESET
RESET	`Q = FALSE`; `S`/`CLK`⟶SET

Application Scenarios

- **Fully-featured operator/blink element** with a defined start value: combines manual set/reset, clock-driven toggling, and a start state that can be configured at startup.

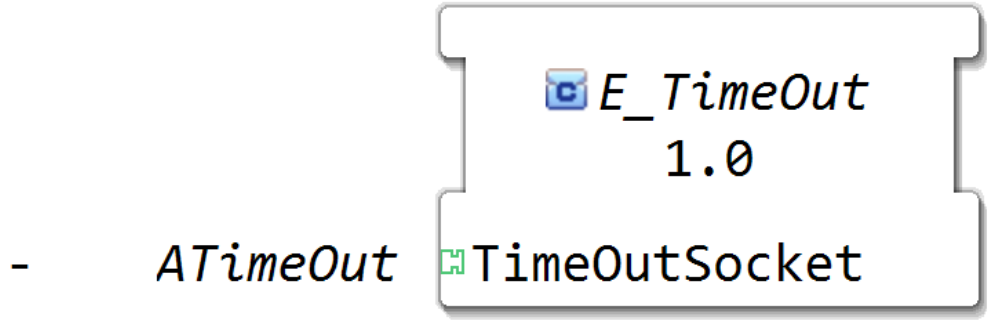
Comparison with similar function blocks

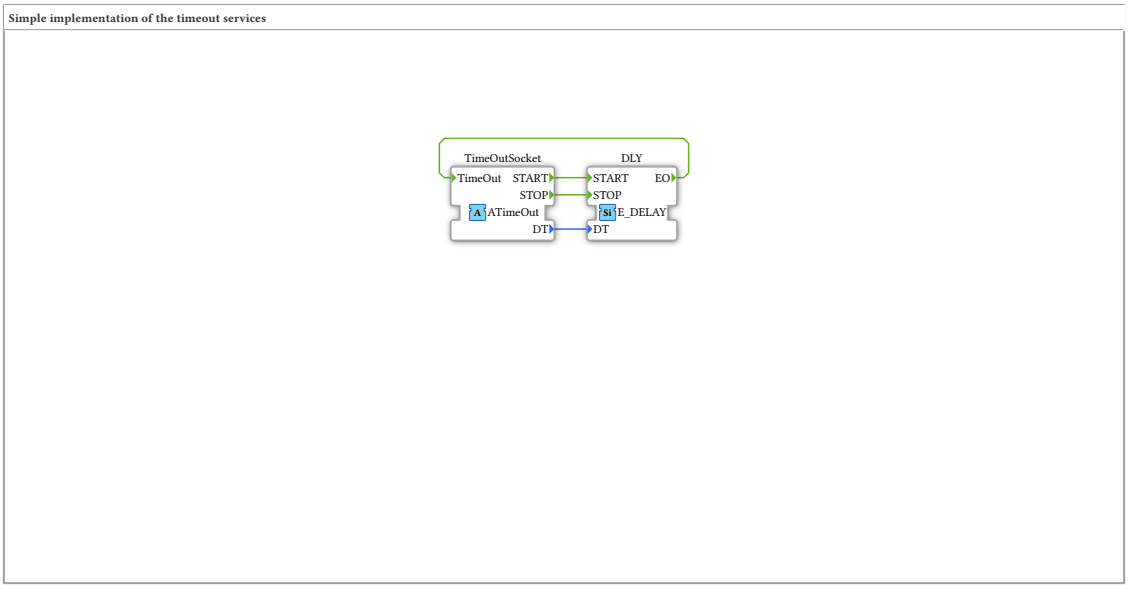
- **E_T_FF_SR_SYM**: the same functionality without `INIT`/`INITO`.
- **E_T_FF_INIT**: only toggle + init, without `S`/`R`.
- **E_RS_SYM_INIT** / **E_SR_SYM_INIT**: only set/reset + init, without `CLK`.

Conclusion

`E_T_FF_SR_SYM_INIT` is the functionally most complete block in the `E_*_SYM` family, combining set/reset, toggle, and configurable initialization in a single function block.

2.11.3.65 E_TimeOut

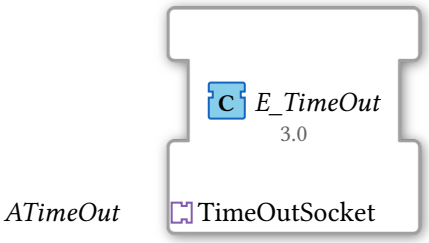




• – ○ • – ○ • – ○ •

Introduction

The **E_TimeOut** is a standards-compliant function block (IEC 61499-1) for implementing timeout services. Version 1.0 offers simple yet effective timeout functionality through internal use of an E_DELAY block. The **E_TimeOut** is a composite function block. Within the network of a composite function block, each adapter added to its interface is represented by an adapter block, which looks like a function block. The interface elements of this adapter block are connected like a function block.



Interface Structure

Adapter Interface (Socket Perspective)

The block uses a **socket** `TimeOutSocket` of type `ATimeOut`. Since this is a socket, the signal directions are inverted compared to the adapter definition (plug):

- **Inputs (received from the socket)**:
- `START`: Starts the internal timer.
- `STOP`: Stops the internal timer.
- `DT` (TIME): The delay time to be used.
- **Output (sent to the socket)**:
- `TimeOut`: Signaled to the connected plug after the specified time has elapsed.

Internal Components

- `DLY` (E_DELAY): Core component for time control

Functionality

1. **Timeout Initialization**:

- Upon a `START` event at the socket, the timer starts with the configured `DT` value.
- Any further `START` event while the timer is running is ignored.

1. **Timeout Termination**:

- A `STOP` event immediately terminates the active timer. No `TimeOut` event is generated.

1. **Timeout Trigger**:

- After `DT` expires, the `TimeOut` event is generated once.

Technical Features

✓ **Adapter-based** interface (`ATimeOut`).
✓ **Simple, non-retriggerable timeout logic**.
✓ **Deterministic** timing behavior.

Application Scenarios

- **Network Communication**: Monitoring for a response within a fixed timeframe. When the response arrives, the timer is canceled via `STOP`.
- **Device Control**: Simple watchdog functions that do not require resetting.
- **Process Monitoring**: Ensuring that a process step does not exceed a maximum duration.

⚖️ **Comparison with E_RTimeOut**

Feature	E_TimeOut (this one)	E_RTimeOut
Internal Block	E_DELAY	E_RDELAY
`START` on running timer	Ignored	Restarts timer
Adapter Type	ATimeOut	ARTimeOut

🛠️ **Related exercises**

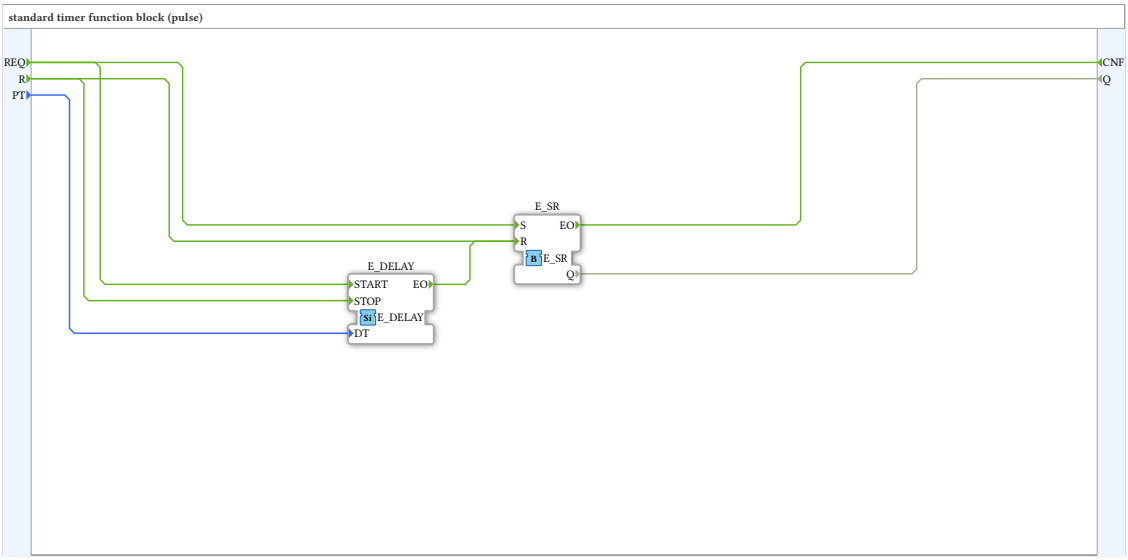
- Uebung_035
- Uebung_035a
- Uebung_035a1_AX
- Uebung_035a2
- Uebung_035a2_AX
- Uebung_035a3
- Uebung_035a3_AX
- Uebung_035c
- Exercise_036
- Exercise_037
- Exercise_038
- Exercise_038_AX
- Exercise_039
- Exercise_039a

Conclusion

The E_TimeOut block provides a robust basic implementation for non-retriggerable timeout requirements. It is ideal for cases where a timer is started and should either run to completion or be explicitly aborted. For scenarios that require “re-triggering” or resetting the timer (such as a watchdog that is periodically “kicked”), the `E_RTimeOut` block is the better choice.

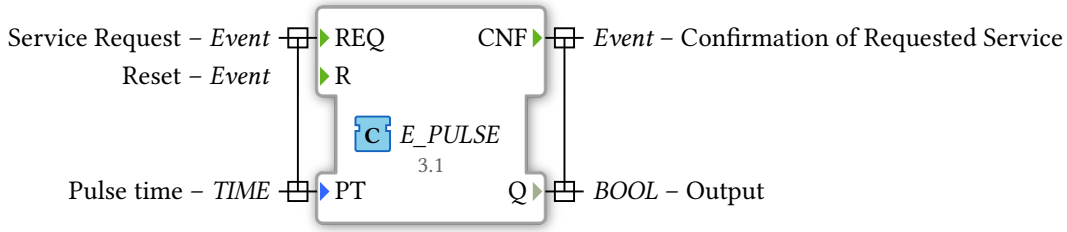
2.11.3.66 timers

E_PULSE



Important note: This function block requires only one event and no cyclic calls. It has no output ET and does not display the elapsed time.

• – ◦ • – ◦ • – ◦ •



The following text was generated by https://chat.deepseek.com/ on February 5, 2025, and has not yet been proofread (HF)

The E_PULSE function block in IEC 61499: An overview

Introduction

IEC 61499 is an international standard that enables the modeling of distributed industrial control systems. A central concept in this standard is the function block (FB), which serves as a fundamental component for the control logic. The **E_PULSE** (Event-driven Pulse) function block is an example of such a function block used in IEC 61499 to generate a pulse of a specific duration. This paper describes the structure, behavior, and application of the E_PULSE function block using the provided source code. A particularly noteworthy aspect is that this function block **also functions when called non-cyclically** because it is internally based on the **E_DELAY** function block from IEC 61499.

Structure of the E_PULSE Function Block

The E_PULSE function block is a **Composite Function Block (CFB)** specified according to IEC 61499. A CFB consists of an interface and an internal network of function blocks that work together to achieve the desired behavior.

Interface

The interface of the E_PULSE block consists of:

- **Event Inputs**:
- **REQ (Request)**: This event triggers the execution of the block. It is linked to the input variable **PT** (Pulse Time).
- **R (Reset)**: This event resets the timer.

- **Event Output**:
- **CNF (Confirmation)**: This event is triggered when the execution of the block is complete. It is linked to the output variable **Q**.
- **Input Variable**:
- **PT (Pulse Time)**: This is a time variable (TIME) that defines the duration of the pulse.
- **Output Variable**:
- **Q (Output)**: This is a Boolean variable that indicates the state of the pulse. It is **TRUE** as long as the pulse is active.

Internal Network (FBNetwork)

The E_PULSE block uses two internal function blocks to implement the desired behavior:

1. **E_DELAY**: This block implements the delay time **PT**. When the **START** event occurs, the timer starts running. When the delay time has elapsed, the **EO** event is triggered.
2. **E_SR**: This block is a bistable element that stores the pulse state. When the **S** event occurs, the output **Q** is set to **TRUE**. When the **R** event occurs, the output **Q** is set to **FALSE**.

Behavior of the E_PULSE Block

The E_PULSE block behaves like a pulse generator that produces a pulse with a specific duration **PT**. The behavior of the function block can be described as follows:

1. **Pulse Start**: When the **REQ** event occurs, the timer is started, and the output **Q** is set to **TRUE**.
2. **Pulse Duration**: The output **Q** remains **TRUE** until the delay time **PT** has elapsed.
3. **Pulse End**: Once the delay time **PT** has elapsed, the output **Q** is set to **FALSE**, and the **CNF** event is triggered to signal the end of the pulse.
4. **Reset**: When the **R** event occurs, the timer is reset, and the output **Q** is set to **FALSE**.

Independence from Cyclic Calling

A key advantage of the E_PULSE function block is that it **does not need to be called cyclically** to function correctly. This is because the function block is internally based on the **E_DELAY** function block, which is specified in IEC 61499. The **E_DELAY** function block is an event-driven timer that operates independently of the cyclic call of the parent function block. This means that the E_PULSE function block can also be used in systems that do not operate cyclically, such as event-driven or distributed control systems.

Application Examples

The E_PULSE function block can be used in various industrial control applications, especially in scenarios where a pulse of a specific duration is required. Some examples are:

- **Machine Control**: The function block can be used to generate a short pulse that activates a machine for a specific period of time, for example, to start or stop a process.
- **Lighting Control**: This function block can be used in lighting systems to switch on the light for a specific period of time, e.g., for emergency lighting.
- **Valve Control**: This function block can be used to open a valve for a specific period of time, e.g., to release a specific quantity of liquid or gas.

🛠️ Related Exercises

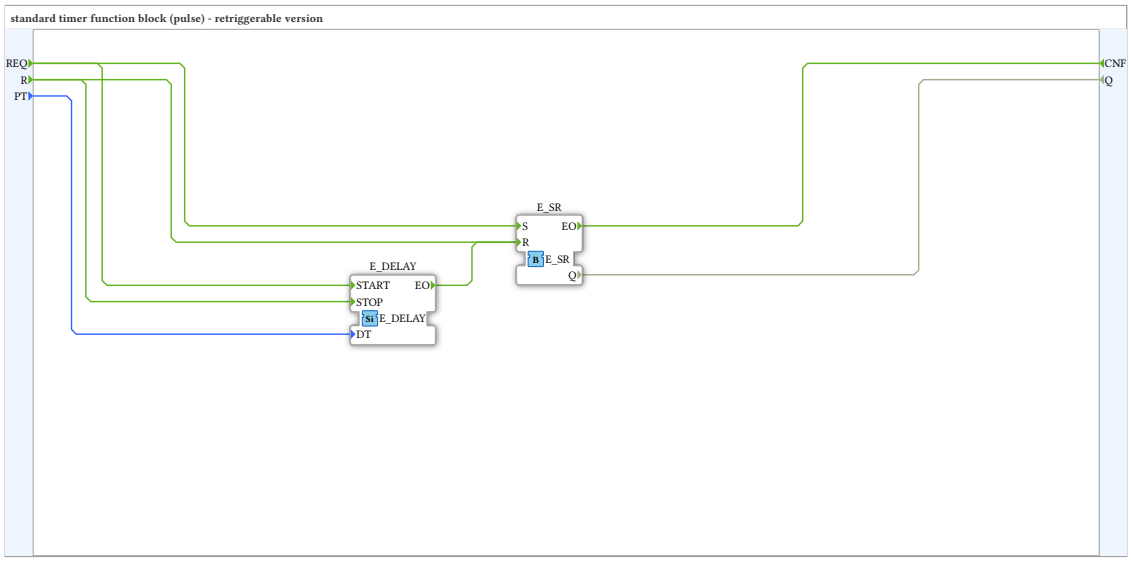
- Exercise_020h
- Exercise_020i

Conclusion

The E_PULSE function block is a versatile function block in IEC 61499 that implements a pulse with a specific duration. Due to its internal use of the **E_DELAY** function block, it can function correctly **even when not called cyclically**. This makes it particularly suitable for event-driven and distributed control systems where cyclic calls are not always guaranteed.

The provided source code demonstrates how the E_PULSE function block can be implemented in practice and offers a solid foundation for developing similar function blocks in industrial control systems. Its independence from cyclic calls is a crucial advantage that makes the function block suitable for a wide range of applications.

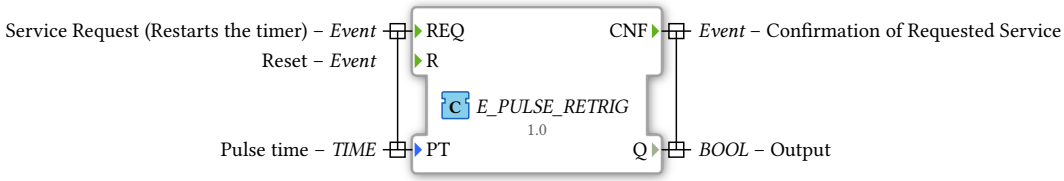
E_PULSE_RETRIG



• – ○ • – ○ • – ○ •

Introduction

`E_PULSE_RETRIG` is the retriggerable variant of `E_PULSE`: it generates a `TRUE` pulse of duration `PT` on `Q`, but restarts the pulse from the beginning on every new `REQ` event instead of ignoring a running interval.



Interface Structure

Event Inputs

- **REQ**: Starts (or retriggers) the pulse, carries `PT`.
- **R**: Immediately aborts the running pulse.

Event Outputs

- **CNF**: Confirmation, carries `Q`.

Data Inputs

- **PT** (TIME): Duration of the pulse.

Data Outputs

- **Q** (BOOL): `TRUE` during the pulse duration, otherwise `FALSE`.

Functionality

Every `REQ` event simultaneously sets `E_SR.S` (⟶ `Q := TRUE`) and restarts `E_DELAY` with time `PT` — a new `REQ` during a running pulse thus restarts the `E_DELAY` timer from scratch instead of waiting for the original pulse to elapse. When `E_DELAY` elapses, `E_SR.R` is triggered (`Q := FALSE`). An `R` event stops the timer immediately and resets `Q` to `FALSE` directly.

Technical Features

- **Retriggerable**: The key difference to `E_PULSE` (not retriggerable) is that every `REQ` restarts the `E_DELAY` timer — a pulse can thus be extended indefinitely through repeated `REQ` events.
- **Composed of `E_DELAY` + `E_SR`**: pure wiring, no ECC of its own.

State Overview

Stateless wiring logic: `E_SR` holds the current `Q` state, `E_DELAY` restarts on every `REQ`.

Application Scenarios

- **Dead-time/watchdog logic**: `Q` should remain `TRUE` as long as `REQ` events keep arriving regularly (e.g. a sensor heartbeat), and only become `FALSE` once `PT` has elapsed since the last `REQ` without a further one.
- **Extendable pulses**, e.g. a light that restarts its on-duration on every new motion detection (`REQ`) instead of timing out.

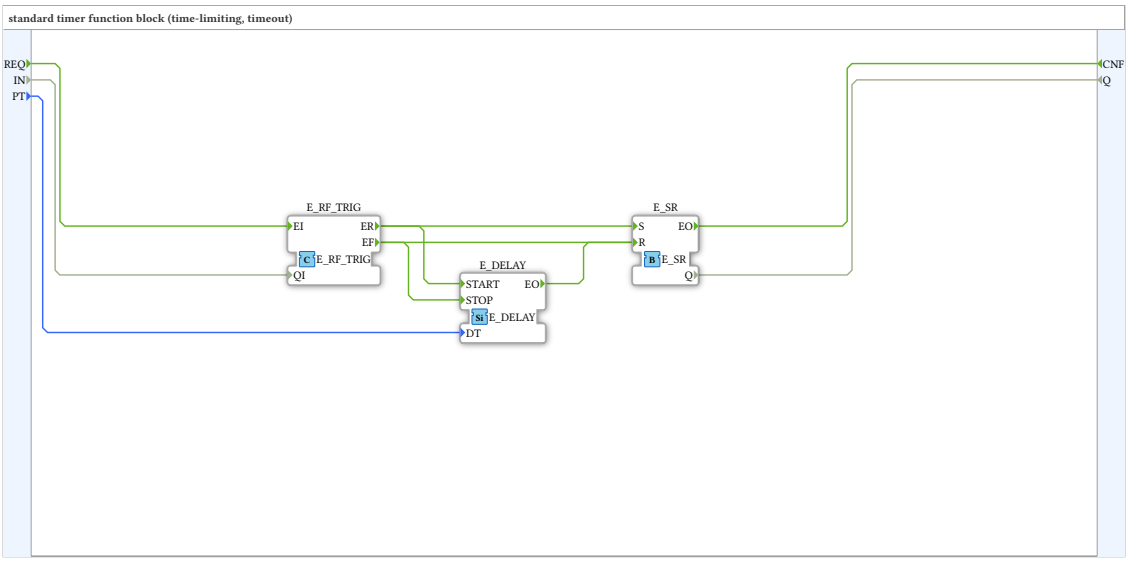
Comparison with similar function blocks

- **`E_PULSE`**: the same base functionality, but not retriggerable (`REQ` during a running pulse has no effect).
- **E_TP_RETRIG**: retriggerable pulse, triggered by a rising edge of `IN` instead of a separate `REQ` event.

Conclusion

`E_PULSE_RETRIG` provides an extendable `TRUE` pulse whose duration restarts on every `REQ` event, suiting watchdog and dead-time applications where a running pulse should be kept alive by repeated requests.

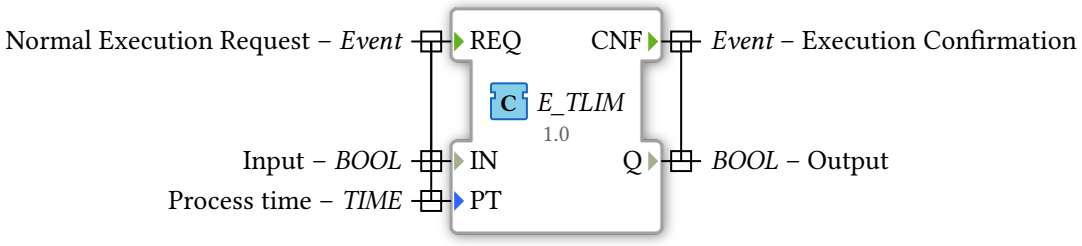
E_TLIM



• – ○ • – ○ • – ○ •

Introduction

`E_TLIM` (time-limiting) limits how long `Q` can stay `TRUE` at most: as soon as `IN` switches to `TRUE`, `Q` also becomes `TRUE`; if `IN` stays `TRUE` longer than `PT`, `Q` automatically switches to `FALSE` once `PT` elapses (timeout). If `IN` already falls back to `FALSE` before `PT` elapses, `Q` follows immediately.



Interface Structure

Event Inputs

- **REQ**: Trigger, carries `IN` and `PT`.

Event Outputs

- **CNF**: Confirmation, carries `Q`.

Data Inputs

- **IN** (BOOL): The input state being monitored.
- **PT** (TIME): Maximum time `Q` may stay `TRUE` after a rising edge of `IN`.

Data Outputs

- **Q** (BOOL): `TRUE` while `IN` is `TRUE` and `PT` has not yet elapsed.

Functionality

`E_RF_TRIG` detects edge transitions of `IN`: a rising edge (`ER`) starts `E_DELAY` (with `PT`) and sets `E_SR.S` (→ `Q := TRUE`). A falling edge (`EF`) immediately stops `E_DELAY` and sets `E_SR.R` (→ `Q := FALSE`) — so `IN` always wins over the timeout. If `E_DELAY` instead runs to completion (because `IN` stayed `TRUE` longer than `PT`), this also triggers `E_SR.R`, causing `Q` to fall back to `FALSE` even without a falling edge of `IN` (the timeout case).

Technical Features

- **Two paths to `Q := FALSE`**: either a falling edge of `IN` (regular case) or `PT` elapsing (timeout case) — both go through `E_SR.R`.
- **Composed of three standard blocks**: `E_RF_TRIG` (edge detection) + `E_DELAY` (timeout timer) + `E_SR` (state storage), no ECC of its own.

State Overview

State (implicit via `E_SR`/`E_DELAY`)	Meaning
Q = FALSE, timer inactive	Idle, `IN = FALSE`
Q = TRUE, timer running	`IN` is `TRUE`, `PT` not yet elapsed
Q = FALSE (timeout)	`IN` was `TRUE` longer than `PT`, automatically reset

Application Scenarios

- **Monitoring maximum durations**: A signal may be active for at most `PT` (e.g. a maximum on-time for an actuator as a safety limit); if `IN` stays active longer, `Q` is automatically reset.
- **Timeout monitoring of process steps**: `IN` marks the running step, `Q` indicates whether the step is still within the allowed time `PT`.

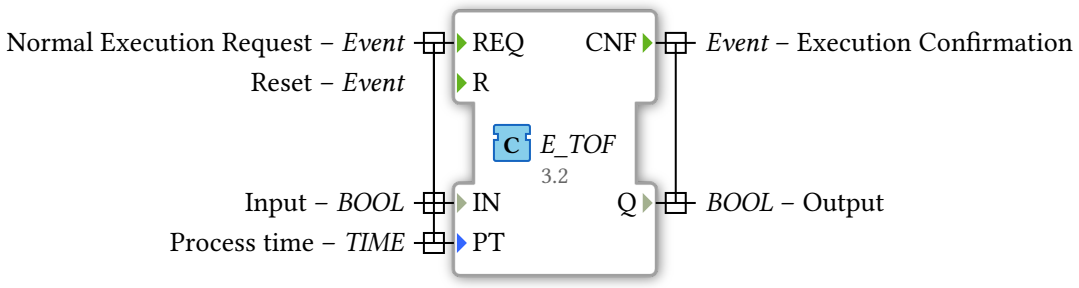
Comparison with similar function blocks

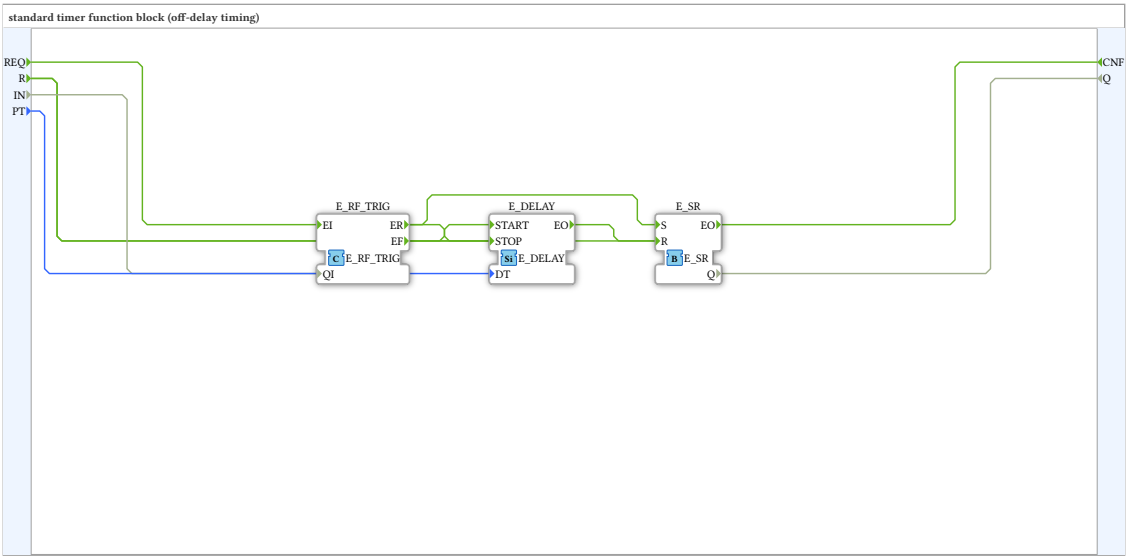
- **`E_TON`/`E_TOF`** (standard IEC 61131-3 timers): delay the *turning on* or *turning off* of `Q` relative to `IN`, but do not limit its maximum on-time.
- **E_PULSE_RETRIG** / **E_TP_RETRIG**: generate a pulse of fixed duration, regardless of how long `IN`/`REQ` actually stays active.
- **`E_RF_TRIG`**: the internally used edge detection.

Conclusion

`E_TLIM` limits the maximum on-time of a `BOOL` signal and is a simple timeout guard for actuators or process steps where staying too long in the `TRUE` state should be ended automatically.

E_TOF





🎧 Podcast

- E_TOF vs. FB_TOF: The event timer that doesn't tick cyclically – a revolution for automation systems?
- FB_TOF and E_TOF: Delay timers in IEC 61131-3 and 61499

IEC 61499 is an international standard that enables the modeling of distributed industrial control systems. A central concept in this standard is the function block (FB), which serves as a fundamental component for the control logic. The **E_TOF** (Event-driven Off-Delay Timer) function block is an example of such a function block used in IEC 61499 to model a delayed shutdown function. This paper describes the structure, behavior, and application of the E_TOF function block using the provided source code. A particularly noteworthy aspect is that this function block **also functions when called non-cyclically** because it is internally based on the **E_DELAY** function block from IEC 61499.

The E_TOF function block is a **Composite Function Block (CFB)** specified according to IEC 61499. A CFB consists of an interface and an internal network of function blocks that work together to implement the desired behavior.

Struktur des E_TOF-Bausteins

Introduction

Der Baustein E_TOF in der IEC 61499: Ein Überblick

Beschreibung

Interface

The interface of the E_TOF block consists of:

- **Event Inputs**:
- **REQ (Request)**: This event triggers the execution of the block. It is linked to the input variables **IN** and **PT**.
- **R (Reset)**: This event resets the timer.
- **Output Event**:
- **CNF (Confirmation)**: This event is triggered when the execution of the block is complete. It is linked to the output variable **Q**.
- **Input Variables**:
- **IN (Input)**: This is a Boolean variable that controls the state of the timer. If **IN** is set to **TRUE**, the timer starts. If **IN** is set to **FALSE**, the delay time begins.
- **PT (Process Time)**: This is a time variable (TIME) that defines the delay time after which the output **Q** is set to **FALSE**.
- **Output Variable**:
- **Q (Output)**: This is a Boolean variable that indicates the timer's state. It remains **TRUE** until the delay time **PT** has elapsed.

Internal Network (FBNetwork)

The E_TOF block uses three function blocks internally to implement the desired behavior:

1. **E_SWITCH**: This block controls the flow of events based on the input **IN**. If **IN** is set to **TRUE**, the event is forwarded to **E_RS.S**. If **IN** is set to **FALSE**, the event is forwarded to **E_DELAY.START**.
2. **E_DELAY**: This block implements the delay time **PT**. When the **START** event occurs, the timer starts running. When the delay time has elapsed, the **EO** event is triggered.
3. **E_RS**: This block is a bistable element that stores the timer's state. When the **S** event occurs, the output **Q** is set to **TRUE**. When the **R** event occurs, the output **Q** is set to **FALSE**.

The E_TOF block behaves like an off-delay timer, implementing a delayed shutdown function. The block's behavior can be described as follows:

1. **Timer Start**: When the **REQ** event occurs and the input variable **IN** is set to **TRUE**, the timer starts. The output **Q** is set to **TRUE**.
2. **Delay Time**: When the input variable **IN** is set to **FALSE**, the delay time **PT** begins. The output **Q** remains **TRUE** until the delay time has elapsed.
3. **Delay Time Expiration**: Once the delay time **PT** has elapsed, the output **Q** is set to **FALSE**, and the **CNF** event is triggered to signal the end of the delay.
4. **Reset**: When the **R** event occurs, the timer is reset, and the output **Q** is set to **FALSE**.

A key advantage of the E_TOF block is that it **does not need to be called cyclically** to function correctly. This is because the block is internally based on the **E_DELAY** block specified in IEC 61499. The **E_DELAY** block is an event-driven timer that operates independently of the cyclic call of the parent block. This means that the E_TOF block can also be used in systems that do not operate cyclically, such as event-driven or distributed control systems.

The E_TOF block can be used in various industrial control applications, especially in scenarios where a delayed shutdown function is required. Some examples are:

- **Machine Control**: The block can be used to ensure that a machine continues to run for a specific period of time after being switched off, for example, to complete processes or to implement safety precautions.
- **Lighting Control**: The block can be used in lighting systems to keep the light on for a specific period of time after the switch has been turned off.
- **Valve Control**: This function block can be used to keep a valve open for a specific period of time after it has closed, for example, to release pressure.
- Exercise_020e

The E_TOF function block is a versatile function block in IEC 61499 that implements a delayed shutdown function. Due to its internal use of the **E_DELAY** function block, it can function correctly **even if it is not called cyclically**. This makes it particularly suitable for event-driven and distributed control systems where cyclic calls are not always guaranteed.

The provided source code demonstrates how the E_TOF function block can be implemented in practice and provides a solid foundation for developing similar function blocks in industrial control systems. The independence from cyclical calls is a crucial advantage that makes the component suitable for a wide variety of applications.

The fact that it doesn't require cyclical calls is a key advantage.## ⚖️ Comparison of E_TOF and FB_TOF

⚖️ Vergleich E_TOF zu FB_TOF

Conclusion

🛠️ Zugehörige Übungen

Anwendungsbeispiele

Unabhängigkeit vom zyklischen Aufruf

Verhalten des E_TOF-Bausteins

Chapter 1: The FB_TOF Block

The **FB_TOF** (Off-Delay Timer) is a standardized function block according to IEC 61131-3 that implements a delayed shutdown function. A key feature of this block is the **ET** (Elapsed Time) output, which displays

the elapsed time since the start of the delay. This output is particularly useful for monitoring the progress of the delay time and for use in control logic.

However, a critical aspect of the FB_TOF block is that it **must be called cyclically** to function correctly. This means that the block must be executed in every cycle of the PLC program to correctly calculate the elapsed time **ET** and maintain the delay time **PT**. If the function block is not called regularly, the timer cannot function correctly, and the delay time may not be adhered to. This dependence on cyclic calling makes the FB_TOF function block ideal for traditional PLC systems that operate in fixed cycles.

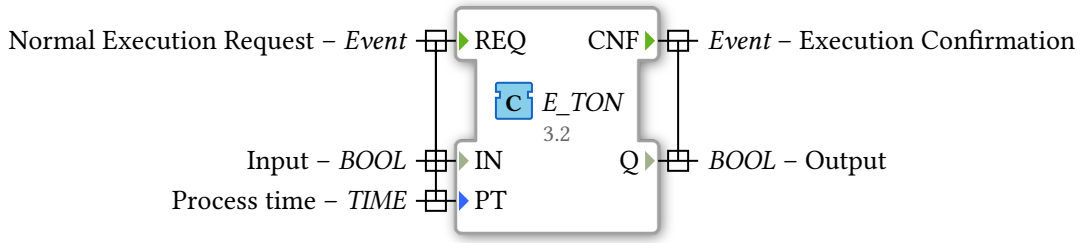
The **E_TOF** (Event-driven Off-Delay Timer) is a function block according to IEC 61499 that also implements a delayed shutdown function. Unlike the FB_TOF, the E_TOF **does not have an ET** output that displays the elapsed time. Instead, the E_TOF is internally based on the **E_DELAY** function block specified in IEC 61499, which provides an event-driven delay function.

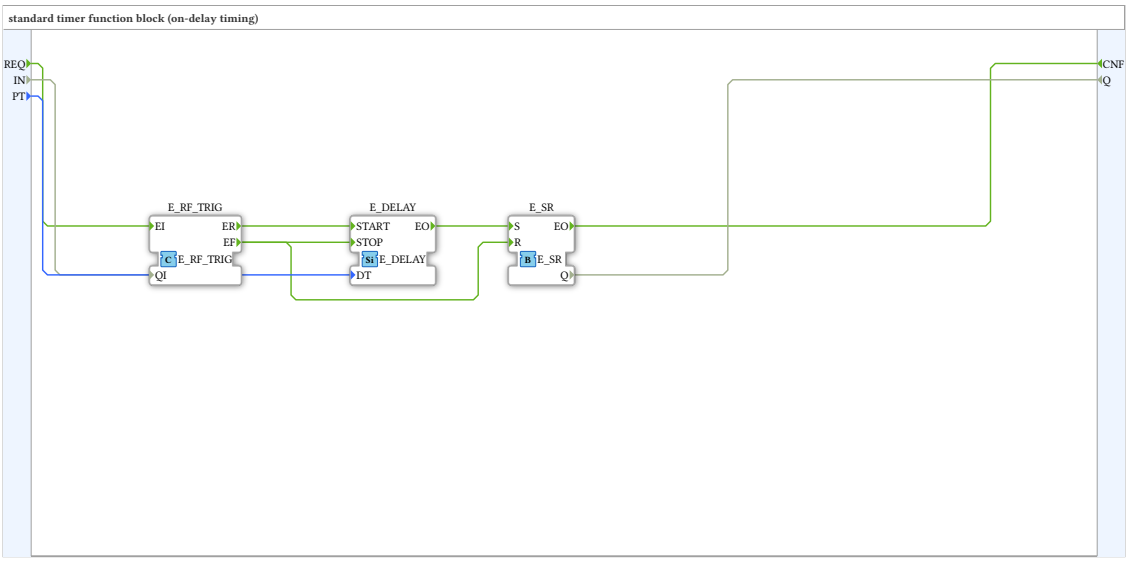
A key advantage of the E_TOF function block is that it **does not need to be called cyclically** to function correctly. Since the internal **E_DELAY** block operates independently of the cyclic call of the parent block, the E_TOF block can also be used in systems that do not operate cyclically, such as event-driven or distributed control systems. This makes the E_TOF block particularly flexible and suitable for modern, distributed control architectures where cyclic calls are not always guaranteed.

In summary, the FB_TOF block, with its **ET** output, enables detailed monitoring of elapsed time but requires a cyclic call. The E_TOF block, on the other hand, does without the **ET** output but offers the flexibility to operate independently of cyclic calls, making it ideal for event-driven systems.

Kapitel 2: Der E_TOF-Baustein

E_TON





Podcast

- E_TON in Industrial Automation: How a Simple Timer Creates Safety and Stability

Text

The **E_TON** function block is a standardized function block type (FBType) according to the **IEC 61499** standard, used for **On-Delay Timers** in industrial automation systems. This function block allows a

signal to be switched on with a time delay, which is required in many control applications, for example, to safely control machines or processes.

General Description

- **Name**: E_TON
- **Type**: Standard Timer Function Block (On-Delay Timing)
- **Purpose**: This function block is used to implement a time-delayed turn-on function. It is particularly useful in applications where a delay is required when a signal is switched on.
- **License**: This function block is licensed under the Eclipse Public License 2.0 (EPL-2.0).

Interface Description

Event Inputs:

1. **REQ** (Normal Execution Request):

- **Type**: Event
- **Description**: This event triggers the normal execution of the function block. It is used to start or update the timer.
- **Associated Variables**:
- **IN**: The input value that starts the timer.
- **PT**: The specified duration for the on-delay.

Event Outputs:

1. **CNF** (Execution Confirmation):

- **Type**: Event
- **Description**: This event confirms the execution of the function block. It is triggered when the timer expires or a state change occurs.
- **Associated Variable**:
- **Q**: The output value indicating the timer's state.

Input Variables:

1. **IN**:

- **Type**: BOOL
- **Description**: The input value that starts the timer. When **IN** is set to `TRUE`, the turn-on delay (**PT**) starts. The output **Q** is only set to `TRUE` after the **PT** time has elapsed.

1. **PT**:

- **Type**: TIME
- **Description**: The preset duration for the on-delay. This time is measured from the moment **IN** is set to `TRUE`.

Output Variables:

1. **Q**:

- **Type**: BOOL
- **Description**: The output value that indicates the timer's state. **Q** is set to `TRUE` after the on-delay (**PT**) has elapsed.

–

Functionality

1. **On-Delay**:

- When **IN** is set to `TRUE`, the on-delay timer (**PT**) starts.
- After **PT** expires, **Q** is set to `TRUE`.
- If **IN** is set to `FALSE` during the timer, the timer stops, and **Q** remains at `FALSE`.

1. **Confirmation of Execution (CNF)**:

- The **CNF** event is triggered when the timer expires and **Q** is set to `TRUE`.

–

Application Examples

- **Machine Control**: Delayed response to sensors or switching commands, e.g., to prevent motor overload.
- **Process Automation**: Time-controlled switching operations in industrial processes, e.g., in conveyor belts or heating controls.

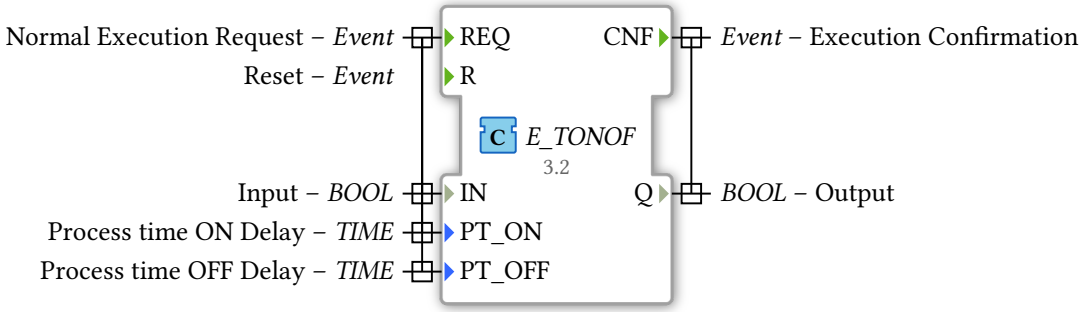
Summary

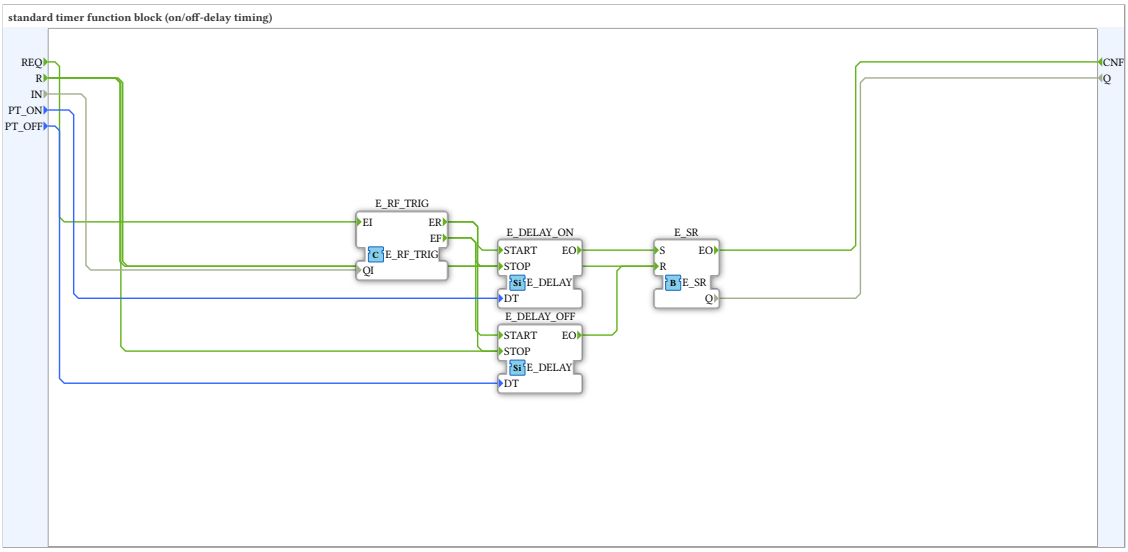
The **E_TON** function block is a simple yet powerful tool in IEC 61499 for implementing time-delayed turn-on operations in industrial automation systems. It provides a reliable way to turn on signals with a defined delay, which is essential in many control applications. The function block is easy to configure and can be used in a variety of scenarios where a time delay is required.

🛠️ Related Exercises

- Exercise_020c
- Exercise_020c2
- Exercise_039b
- Exercise_160b2

E_TONOF





Important note: This function block requires only one event and no cyclic calls. It does not have an output ET and does not display the elapsed time.

Description

The **E_TONOF** function block is a standardized function block type (FBType) according to the **IEC 61499** standard, used for time-delayed switching operations in industrial automation systems. This function block combines the functions of an **On-Delay Timer** and an **Off-Delay Timer** in a single block. It allows for the time delay of both the switching on and off of a signal.

General Description

- **Name**: E_TONOF
- **Type**: Standard Timer Function Block (On/Off Delay Timing)
- **Purpose**: This function block is used to implement time-delayed switching operations in control systems. It is particularly useful in applications where a delay is required when switching signals on or off.
- **License**: This function block is licensed under the Eclipse Public License 2.0 (EPL-2.0).

Interface Description

Event Inputs:

1. **REQ** (Normal Execution Request):

- **Type**: Event
- **Description**: This event triggers the normal execution of the function block. It is used to start or update the timer.
- **Associated Variables**:
- **IN**: The input value that starts the timer.
- **PT_ON**: The preset duration for the on-delay.
- **PT_OFF**: The preset duration for the off-delay.

1. **R** (Reset):

- **Type**: Event
- **Description**: This event resets the timer. When this event is triggered, the timer is stopped, and all internal states are reset.

Event Outputs

1. **CNF** (Execution Confirmation):

- **Type**: Event
- **Description**: This event confirms the execution of the function block. It is triggered when the timer has expired or a state change has occurred.
- **Associated Variable**:
- **Q**: The output value indicating the timer's state.

Input Variables:

1. **IN**:

- **Type**: BOOL
- **Description**: The input value that starts the timer. When **IN** is set to `TRUE`, the on-delay starts. When **IN** is set to `FALSE`, the off-delay starts.

1. **PT_ON**:

- **Type**: TIME
- **Description**: The preset duration for the on-delay. This time is measured from the moment **IN** is set to `TRUE`.

1. **PT_OFF**:

- **Type**: TIME
- **Description**: The preset duration for the off-delay. This time is measured from the moment **IN** is set to `FALSE`.

Output Variables:

1. **Q**:

- **Type**: BOOL
- **Description**: The output value that indicates the timer's state. **Q** is set to `TRUE` after the On-Delay has elapsed and remains at `TRUE` until the Off-Delay has elapsed.

_

Functionality

1. **On-Delay**:

- When **IN** is set to `TRUE`, the On-Delay timer (**PT_ON**) starts.
- After **PT_ON** expires, **Q** is set to `TRUE`.

1. **Off-Delay**:

- When **IN** is set to `FALSE`, the off-delay timer (**PT_OFF**) starts.
- **Q** remains at `TRUE` until **PT_OFF** expires, and is then reset to `FALSE`.

1. **Reset**:

- When the **R** event is triggered, the timer is reset, and **Q** is immediately set to `FALSE`.

–

Application Examples

- **Machine Control**: Delayed response to sensors or switching commands, e.g. B. to prevent motor overload.
- **Process Automation**: Time-controlled switching operations in industrial processes, e.g., in conveyor belts or heating controls.

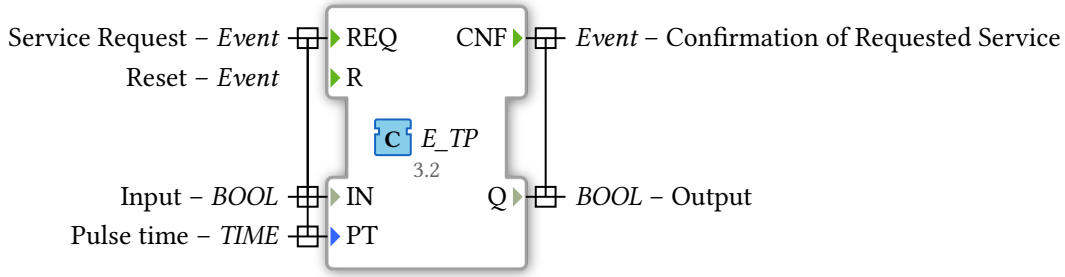
Summary

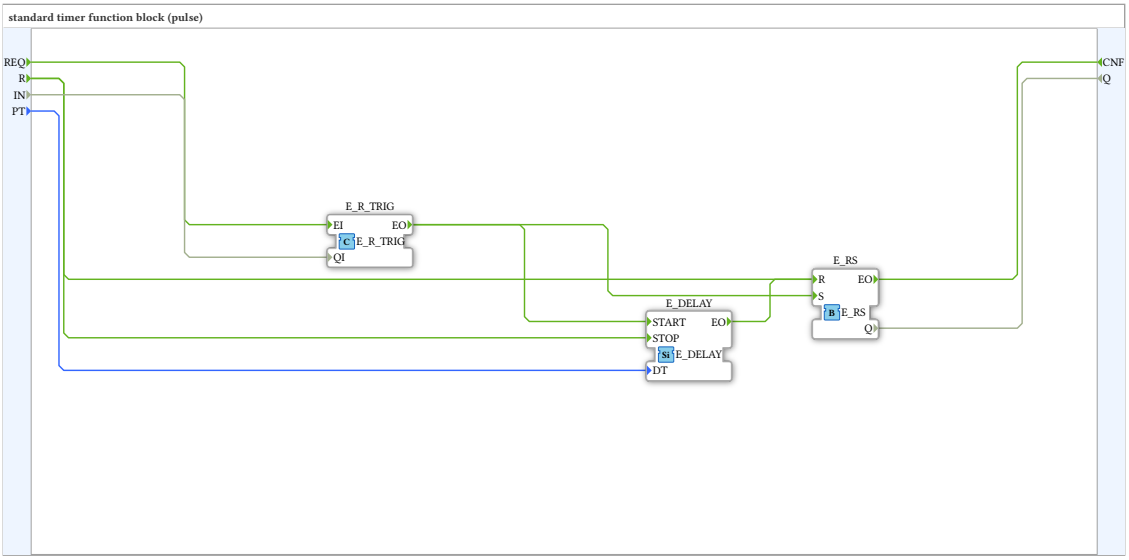
The **E_TONOF** block is a powerful tool in IEC 61499 for implementing time-delayed switching operations in industrial automation systems. It combines the functions of a switch-on and switch-off delay timer in a single block and also offers the option of resetting the timer via a reset event. This makes it particularly flexible and versatile for use in complex control tasks.

🛠️ Related Exercises

- Exercise_020g

E_TP





Important note: This function block requires only one event and no cyclic calls. It does not have an output ET and does not display the elapsed time.

Description

The **E_TP** function block (also known as **Pulse Generator** or **Timer Pulse**) is a standardized function block (FB) according to the **IEC 61499** (DIN EN 61499) standard. This block is used to generate a pulsating output that is activated for a specific duration when an input signal is received. It is particularly useful in control applications where time-controlled actions are required.

Description of the E_TP Block

General Function:

- The **E_TP** block generates a pulsating output (**Q**) that is activated for a predefined time (**PT**) as soon as an input signal (**IN**) is received.
- When the input signal **IN** is set to **TRUE**, the output **Q** is set to **TRUE** for the duration **PT**. After the time **PT** has elapsed, **Q** is reset to **FALSE**, regardless of the state of **IN**.
- The block has a reset function (**R**) that resets the timer and immediately sets the output **Q** to **FALSE**, regardless of whether the time **PT** has expired or not.

Interfaces:

1. **Event Inputs:**

- **REQ (Service Request):** This event is triggered when the timer is to be started. It is linked to the input variables **IN** and **PT**.
- **R (Reset):** This event is triggered to reset the timer. It is linked to the input variable **IN**.

1. **Output Events:**

- **CNF (Confirmation of Requested Service):** This event is triggered when the timer expires and the output **Q** is reset. It is linked to the output variable **Q**.

1. **Input Variables:**

- **IN (BOOL):** The input that starts the timer. When **IN** is set to **TRUE**, the timer starts.
- **PT (TIME):** The specified duration for which the output **Q** is activated.

1. **Output Variables:**

- **Q (BOOL):** The output that is set to **TRUE** for the duration **PT** after **IN** has been activated.

Behavior:

- When **IN** is set to **TRUE**, the output **Q** is immediately set to **TRUE** and remains active for the duration **PT**.
- After the time **PT** has elapsed, **Q** is automatically reset to **FALSE**.

- When the reset event **R** is triggered, the timer is immediately reset, and **Q** is set to **FALSE**, regardless of whether the time **PT** has elapsed or not.

Application Examples:

- **Machine Control:** This function block can be used to activate a machine for a specific time, e.g., to switch on a motor for a set duration.
- **Flashing Light Control:** This function block can be used to activate a flashing light for a specific time.
- **Delayed Actions:** In scenarios where an action should be performed after a certain delay, this function block can be used to implement the delay.

Versioning:

- **Version 1.0:** Original version of the function block, created by Franz Hoepfinger on March 4, 2024.
- **Version 1.1:** A reset function was added to allow manual resetting of the timer. This version was released on April 23, 2024.

Summary

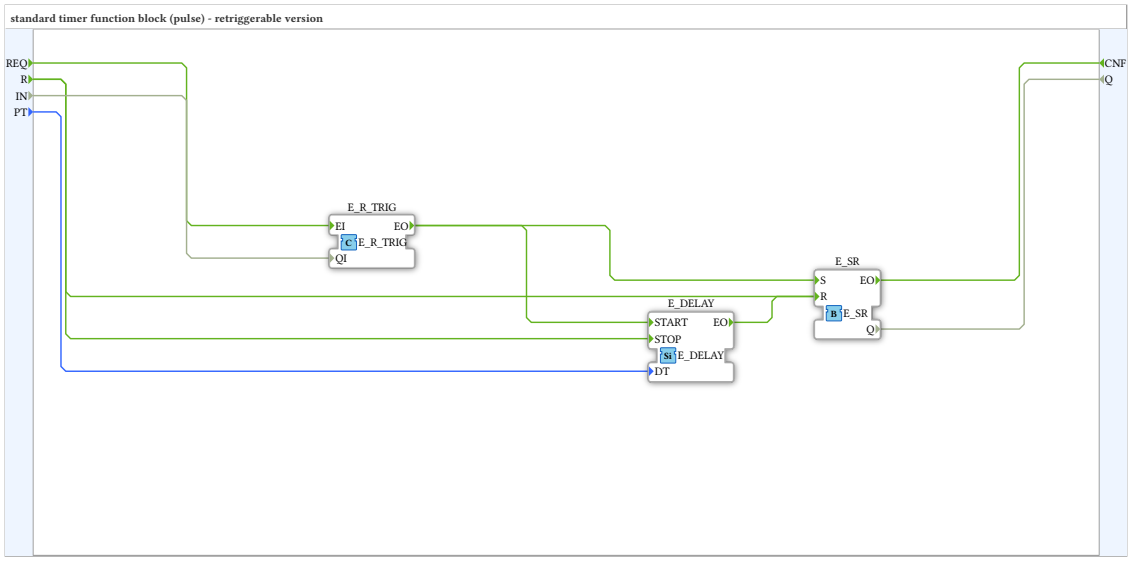
The **E_TP** function block is a useful tool in IEC 61499-based control technology for implementing time-controlled actions. Its simple interface and clear behavior make it easy to integrate into various control applications. The reset function in version 1.1 increases the flexibility of the function block, as the timer can now be reset manually.

The **E_TP** function block is a useful tool in IEC 61499-based control technology for implementing time-controlled actions.

🛠️ Related exercises

- Uebung_020f
- Uebung_039b
- Uebung_177
- Uebung_178

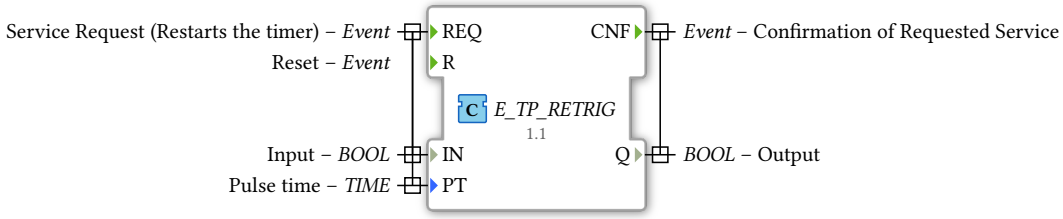
E_TP_RETRIG



• — ○ • — ○ • — ○ •

Introduction

`E_TP_RETRIG` is the retriggerable variant of the standard timer `E_TP`: on every rising edge of `IN`, it generates a `TRUE` pulse of duration `PT` on `Q` — unlike `E_TP`, however, a new rising edge during a running pulse restarts it instead of being ignored.



Interface Structure

Event Inputs

- **REQ**: Checks `IN` for a rising edge, carries `IN` and `PT`.
- **R**: Immediately aborts the running pulse.

Event Outputs

- **CNF**: Confirmation, carries `Q`.

Data Inputs

- **IN** (BOOL): The input to check for rising edges.
- **PT** (TIME): Duration of the pulse.

Data Outputs

- **Q** (BOOL): `TRUE` during the pulse duration.

Functionality

`E_R_TRIG` detects rising edges of `IN` on every `REQ`. Every detected edge sets `E_SR.S` (⟶ `Q := TRUE`) and simultaneously restarts `E_DELAY` (with `PT`) — a new rising edge during a running pulse thus extends it instead of being ignored. When `E_DELAY` elapses, it sets `E_SR.R` (⟶ `Q := FALSE`). An `R` event stops the timer immediately and resets `Q` directly.

Technical Features

- **Retriggerable via an `IN` edge rather than `REQ` alone**: Unlike E_PULSE_RETRIG, which retriggers on *every* `REQ` event, `E_TP_RETRIG` only reacts to actual rising edges of `IN` — repeated `REQ` events while `IN` is already held `TRUE` without a new edge do not trigger a new pulse.
- **Composed of three standard blocks**: `E_R_TRIG` + `E_DELAY` + `E_SR`, no ECC of its own.

State Overview

Stateless wiring logic: `E_SR` holds the `Q` state, `E_DELAY` restarts on every detected rising edge of `IN`.

Application Scenarios

- **Extendable pulses triggered by edge transitions**: e.g. motion-sensor-controlled lighting, where every new motion (rising edge of `IN`) restarts the on-duration.
- **Debounced but extendable response to a digital input**, where only genuine edge transitions — not mere `REQ` repetitions — should trigger a new pulse run.

Comparison with similar function blocks

- **`E_TP`**: the same base functionality, but not retriggerable.
- E_PULSE_RETRIG: retriggerable pulse, triggered by *every* `REQ` event instead of an `IN` edge.
- E_TLIM: limits the on-time of `IN` itself, instead of generating a fixed pulse.

Conclusion

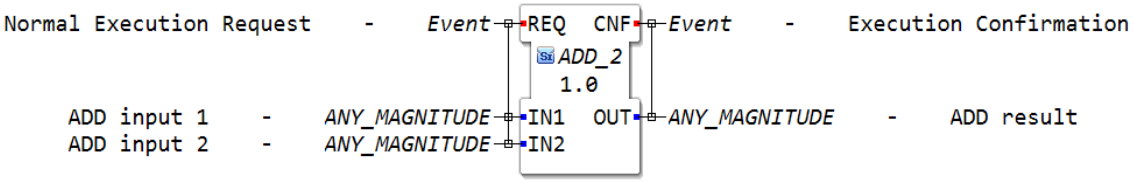
`E_TP_RETRIG` provides an edge-triggered, extendable pulse of fixed duration and is suitable for applications where repeated genuine edge transitions — not mere `REQ` repetitions — should keep the pulse alive.

2.11.4 iec61131-3

2.11.4.1 Arithmetic Functions

Here you will find standard functions for arithmetic operations (addition, subtraction, multiplication, division, etc.).

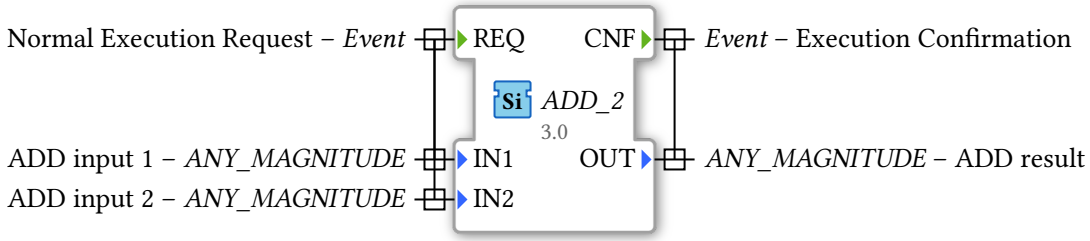
ADD_2



• – ◦ • – ◦ • – ◦ •

Introduction

The **ADD_2** is a generic function block for adding two values, developed under the EPL-2.0 license. Version 1.0 supports various numeric data types according to the IEC 61131-3 standard.



Interface Structure

Event Inputs

- `REQ`: Calculation request (with input values)

Event Outputs

- `CNF`: Calculation confirmation (with result)

Data Inputs

- `IN1` (ANY_MAGNITUDE): First summand
- `IN2` (ANY_MAGNITUDE): Second summand

Data Outputs

- `OUT` (ANY_MAGNITUDE): Sum result

Functionality

1. **Calculation Request**:

- `REQ` event with Values for IN1 and IN2

1. **Addition**:

- Type-preserving calculation (IN1 + IN2)
- Supported types: INT, DINT, REAL, LREAL, etc.

1. **Result Output**:

- `CNF` event with calculated OUT value
- Automatic type adjustment

Technical Features

✓ **Generic Implementation** (GEN_ADD) ✓ **Type-Flexible** Inputs/Outputs ✓ **IEC 61131-3 compliant** ✓ **Deterministic** calculation

Supported Data Types

Type	Example
INT	123 + 456 = 579
REAL	1.23 + 4.56 = 5.79
TIME	T#1s + T#2s = T#3s
LREAL	1.23e10 + 4.56e10

Application Scenarios

- **Process Control**: Signal Processing
- **Data Analysis**: Measurement Aggregation
- **Machine Control**: Position Calculation
- **Energy Management**: Consumption Summing

⚖️ Comparison with Similar Function Blocks

Feature	ADD_2	ADD_4	MUL_2
Operation	Addition	Addition	Multiplication
Inputs	2	4	2
Type Support	ANY_MAG	ANY_MAG	ANY_MAG

Error Handling

- Type incompatibility results in:
- OUT = 0 (for numeric types)
- Empty value (for other types)
- No exception generation

🛠️ Related Exercises

- Exercise_000
- Exercise_000_AX
- Exercise_110
- Exercise_111

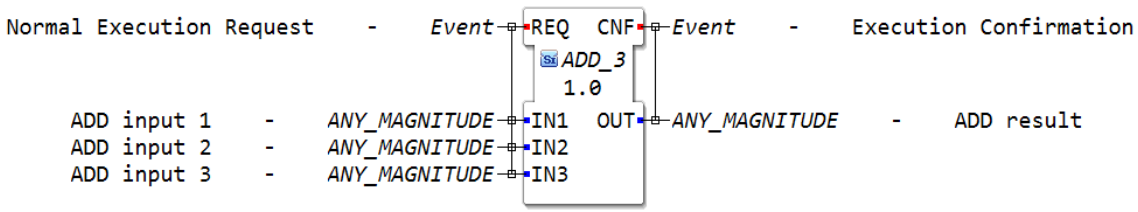
Conclusion

The ADD_2 block offers a flexible solution for arithmetic operations:

- Simple yet powerful addition
- Broad type support
- Standards-compliant implementation

Especially valuable for applications requiring type-safe calculations with variable inputs. The generic implementation allows for reuse in a wide variety of control contexts.

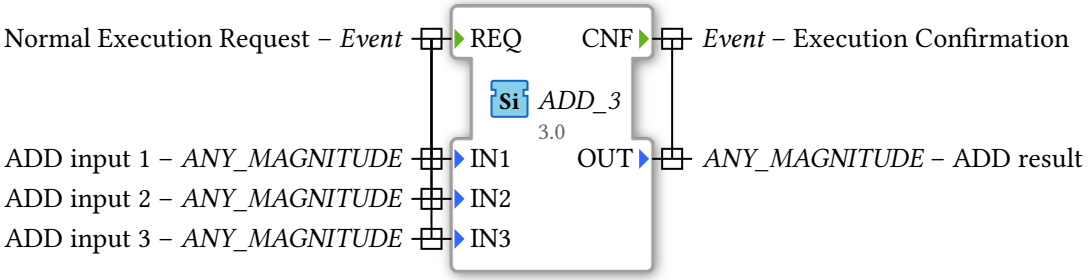
ADD_3



• – ◦ • – ◦ • – ◦ •

Introduction

The **ADD_3** is an enhanced version of the adder for calculating the sum of three values, developed under the EPL-2.0 license. Version 1.0 offers type-safe addition for various numeric data types according to the IEC 61131-3 standard.



Interface Structure

Event Inputs

- `REQ`: Calculation request (with three input values)

Event Outputs

- `CNF`: Calculation confirmation (with sum result)

Data Inputs

- `IN1` (ANY_MAGNITUDE): First addend
- `IN2` (ANY_MAGNITUDE): Second addend
- `IN3` (ANY_MAGNITUDE): Third addend

Data Outputs

- `OUT` (ANY_MAGNITUDE): Result of addition (IN1 + IN2) + IN3)

Operating Principle

1. **Requirement**:

- The `REQ` event triggers the calculation
- All three input values are evaluated simultaneously

1. **Calculation**:

- Type-preserving summation
- Supports all ANY_MAGNITUDE types:
- Integers (INT, DINT)
- Floating-point numbers (REAL, LREAL)
- Time values (TIME, LTIME)

1. **Result Output**:

- The `CNF` event with the calculated OUT value
- Automatic type conversion for mixed inputs

Technical Features

✓ **Three-input addition** in a single operation ✓ **Generic implementation** (GEN_ADD class) ✓ **Broad type support** (ANY_MAGNITUDE) ✓ **Deterministic execution**

Typical Use Cases

- **Measurement Aggregation**: Summing data from multiple sensors
- **Energy Balancing**: Calculating total energy consumption
- **Positioning Technology**: Multidimensional coordinates
- **Production Statistics**: Quantity measurement

⚖️ Comparison with ADD_2

Feature	ADD_3	ADD_2
Inputs	3	2
Calculation	IN1+IN2+IN3	IN1+IN2
Performance	Higher efficiency with 3 inputs	Lightweight

Error Handling

- In case of type incompatibility:
- OUT = 0 (numeric types)
- Empty value (non-numeric types)
- Execution is not aborted

🛠️ Related Exercises

- Exercise_087
- Exercise_087a1

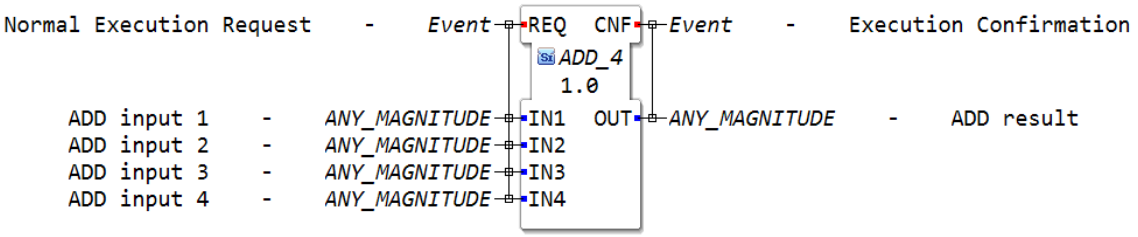
Conclusion

The ADD_3 block expands the possibilities of type-safe addition:

- Efficient triple summation
- Consistent type handling
- Optimized performance compared to serial ADD_2 usage

Particularly advantageous for applications with multiple values to be aggregated. The generic implementation allows for maximum flexibility in heterogeneous control environments.

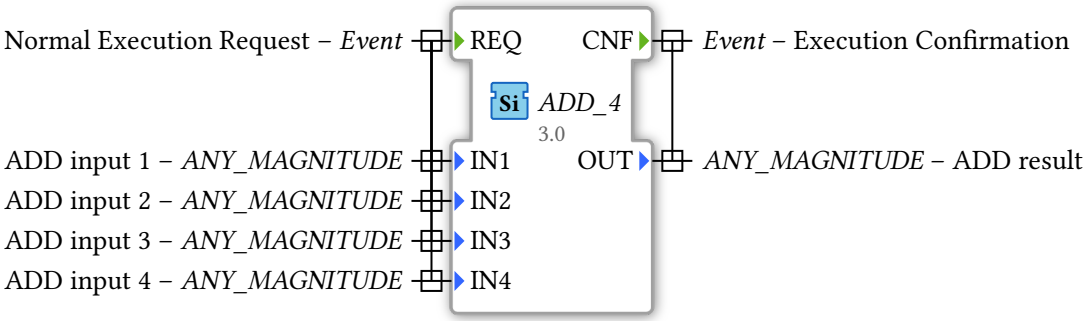
ADD_4



• – ◦ • – ◦ • – ◦ •

Introduction

The **ADD_4** is a powerful function block for summing four values, developed under the EPL-2.0 license. As part of the GEN_ADD class, version 1.0 offers a type-safe implementation according to the IEC 61131-3 standard.



Interface Structure

Event Inputs

- `REQ`: Calculation request (with four input values)

Event Outputs

- `CNF`: Calculation confirmation (with total result)

Data Inputs

- `IN1` (ANY_MAGNITUDE): First addend
- `IN2` (ANY_MAGNITUDE): Second addend
- `IN3` (ANY_MAGNITUDE): Third addend
- `IN4` (ANY_MAGNITUDE): Fourth addend

Data Outputs

- `OUT` (ANY_MAGNITUDE): Result of addition (IN1+IN2+IN3+IN4)

Functional Features

1. **Calculation Triggering**:

- A single `REQ` event processes all four inputs
- Parallel value acquisition

1. **Type Handling**:

- Automatic type conversion for mixed inputs
- Result in the "largest" input type

1. **Calculation Order**:

- Mathematically associative (IN1+(IN2+(IN3+IN4)))
- Guaranteed reproducible results

Technical Specifications

✓ **Four-input addition** in one operation ✓ **Full ANY_MAGNITUDE support** ✓ **Optimized calculation performance** ✓ **Deterministic behavior**

Supported Data Types

Category	Examples
Integers	INT, DINT, UDINT
Floating Point	REAL, LREAL
Time Values	TIME, LTIME
Special Types	DATE_AND_TIME

Application Scenarios

- **Energy Monitoring**: Summing of Consumers
- **Production Control**: Quantity Measurement of Multiple Lines
- **Measurement Data Analysis**: Aggregation of Sensor Values
- **Robot Control**: Multidimensional Position Calculation

⚖️ Comparison with Other Adders

Function Block	Inputs	Advantages
ADD_2	2	Simplest Variant
ADD_3	3	Compromise
ADD_4	4	Maximum Integration

Error Management

- **Type Conflicts**: Result 0 in case of incompatibility
- **Overflow**: Wrap-around depending on data type
- **Status Feedback**: Via accompanying logic

Conclusion

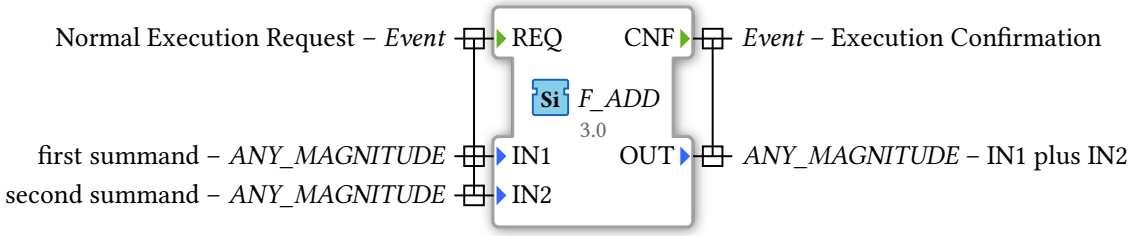
The ADD_4 block offers maximum efficiency for summation tasks:

- Compact quadruple calculation
- Broadest type support
- Optimized performance

Ideal for complex control applications with multiple values to be aggregated. The generic implementation as GEN_ADD enables maximum flexibility with type-safe execution.

F_ADD

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_ADD** is a standards-compliant arithmetic function block according to IEC 61131-3, developed by TU Wien ACIN under the EPL-2.0 license. Version 1.0 enables type-safe addition of values (ANY_MAGNITUDE) for industrial control applications.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with input values)

Event Outputs

- `CNF`: Calculation confirmation (with result)

Data Inputs

- `IN1` (ANY_MAGNITUDE): First operand
- `IN2` (ANY_MAGNITUDE): Second operand

Data Outputs

- `OUT` (ANY_MAGNITUDE): Sum of IN1 and IN2

Functionality

1. **Activation**:

- Triggered by the `REQ` event
- Processes both input values simultaneously

1. **Type Handling**:

- Supports all ANY_MAGNITUDE types:
- Numeric types (INT, DINT, REAL, LREAL)
- Time types (TIME, LTIME)
- Automatic type conversion

1. **Computation**:

- Executes IN1 + IN2

- Maintains the highest input accuracy

1. **Result Output**:

- `CNF` event with computation result
- Latency: One execution cycle

Technical Specifications

✓ Generic Type Support (ANY_MAGNITUDE) **✓ Deterministic Execution ✓ Overflow Protection ✓ IEC 61131-3 Compliant**

Supported Data Types

Category	Examples
Integers	INT, DINT, UINT, UDINT
Floating Point	REAL, LREAL
Time	TIME, LTIME
Mixed	INT + REAL ⟶ REAL

Application Examples

- **Process Control**: Signal Conditioning
- **Measurement Systems**: Sensor Value Aggregation
- **Motion Control**: Position Calculations
- **Energy Management**: Consumption Summing

⚖️ **Comparison with Similar Function Blocks**

Feature	F_ADD	F_SUB	F_MUL
Operation	Addition	Subtraction	Multiplication
Input Types	ANY_MAGNITUDE	ANY_MAGNITUDE	ANY_NUM
Output Type	Equals Input Type	Equals	Equals

Error Handling

- **Overflow**: Wrap-around for integers, ±INF for floating-point numbers
- **Type Mismatch**: Automatic type matching
- **Invalid Input**: Returns 0/default value

Special Notes

- For time calculations: TIME + TIME ⟶ TIME
- Mixed operations use the “higher” type
- No implicit scaling (e.g., 100ms + 1s = 1100ms)

Conclusion

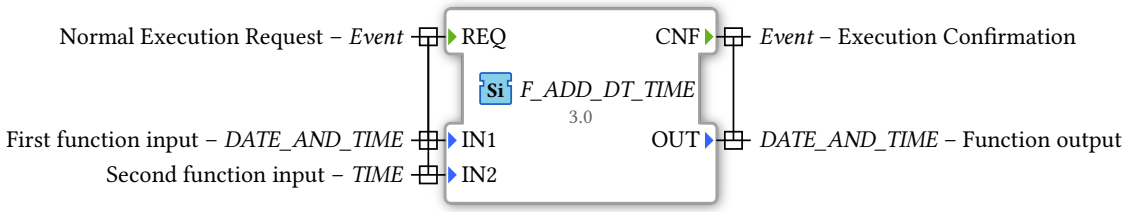
The F_ADD block offers essential arithmetic functions:

- Robust, type-safe addition
- Consistent behavior across all data types
- Seamless integration into control algorithms

Its standardized implementation makes it a reliable component for industrial automation systems that require precise arithmetic operations with various data types.

F_ADD_DT_TIME

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_ADD_DT_TIME** function block is a specialized function module for adding time values to date/time values, developed under the EPL 2.0 license. Version 1.0 enables precise time calculations using DATE_AND_TIME and TIME data types.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with input values)

Event Outputs

- `CNF`: Calculation confirmation (with result)

Data Inputs

- `IN1` (DATE_AND_TIME): Output date/time
- `IN2` (TIME): Duration for addition

Data Outputs

- `OUT` (DATE_AND_TIME): Result of the addition

Functionality

1. **Activation**:

- Triggered by the `REQ` event
- Processes DATE_AND_TIME and TIME values

1. **Calculation**:

- Adds time duration (IN2) to timestamp (IN1)
- Automatically takes into account:
- Leap years
- Month lengths
- Time zones (if configured)

1. **Result output**:

- `CNF` event with new DATE_AND_TIME
- Format: DT#2023-12-31-23:59:59

Technical Specifications

✓ Precise time/date arithmetic ✓ Automatic calendar calculation
✓ IEC 61131-3 compliant ✓ Deterministic execution

Application examples

- **Process control**: Due date calculations
- **Production planning**: Scheduling adjustments
- **Maintenance systems**: Maintenance interval calculation
- **Data logging**: Timestamp corrections

Special notes

- **Scope of validity**:
- Supports date range 1970-2099
- Time resolution: 1 millisecond

- **Error cases**:
- In case of overflow: Wrap-around within the valid range
- In case of invalid input: OUT = IN1

⚖️ Comparison with standard F_ADD

Feature	F_ADD_DT_TIME	F_ADD
Input type 1	DATE_AND_TIME	ANY_MAGNITUDE
Input type 2	TIME	ANY_MAGNITUDE
Operation	Specialized time addition	Generic addition

Conclusion

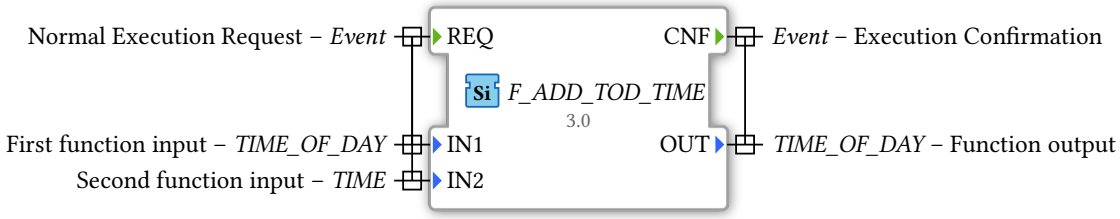
The F_ADD_DT_TIME function block offers essential functions for time calculations:

- Precise handling of date/time operations
- Automatic calendar corrections
- Reliable integration into control systems

Particularly valuable for applications requiring complex time calculations with automatic consideration of calendar rules. The specialized implementation guarantees correct results for all valid input values.

F_ADD_TOD_TIME

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_ADD_TOD_TIME** function block is a specialized function module for adding time values to times of day, developed under the EPL 2.0 license. Version 1.0 enables precise time calculations using TIME_OF_DAY and TIME data types.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with input values)

Event Outputs

- `CNF`: Calculation confirmation (with result)

Data Inputs

- `IN1` (TIME_OF_DAY): Output time (time of day)
- `IN2` (TIME): Duration for addition

Data Outputs

- `OUT` (TIME_OF_DAY): Result of the addition

Functionality

1. **Activation**:

- Triggered by the `REQ` event
- Processes TIME_OF_DAY and TIME values

1. **Calculation**:

- Adds time duration (IN2) to time of day (IN1)
- Automatically takes into account:
- Day transitions (24-hour format)
- Millisecond accuracy

1. **Result Output**:

- `CNF` event with new TIME_OF_DAY
- Format: TOD#23:59:59.999

Technical Specifications

✓ Precise Time of Day Calculation ✓ Automatic Day Transition ✓ IEC 61131-3 Compliant ✓ Millisecond Accurate Calculation

Application Examples

- **Shift Planning**: Working Time Calculations
- **Lighting Control**: Daylight Simulation
- **Production Processes**: Time-Controlled Sequences
- **Scheduling Systems**: Arrival Time Calculations

Special Notes

- **Scope of Application**:
- 00:00:00.000 to 23:59:59.999
- If the limit is exceeded: Wrap-around (e.g., 25:00:00 ⟶ 01:00:00)
- **Error Cases**:
- In case of overflow: Cyclic incrementing
- In case of invalid input: OUT = IN1

⚖️ Comparison with similar function blocks

Feature	F_ADD_TOD_TIME	F_ADD_DT_TIME
Input Type 1	TIME_OF_DAY	DATE_AND_TIME
Output Type	TIME_OF_DAY	DATE_AND_TIME
Range	Daily Cycle	Absolute Date

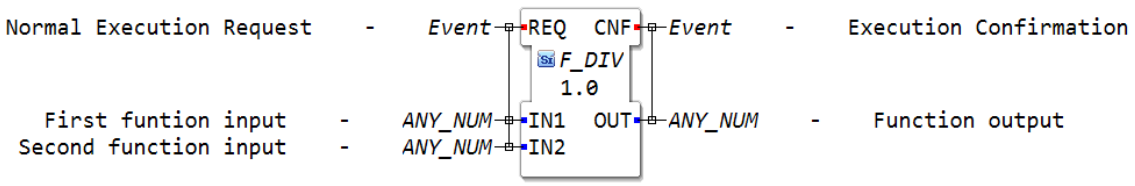
Conclusion

The F_ADD_TOD_TIME function block offers essential functions for calculating the time of day:

- Precise handling of time-of-day operations
- Automatic handling of day transitions
- Optimized performance for real-time applications

Ideal for control applications that require recurring time-of-day-based operations. The specialized implementation guarantees correct results in 24-hour format.

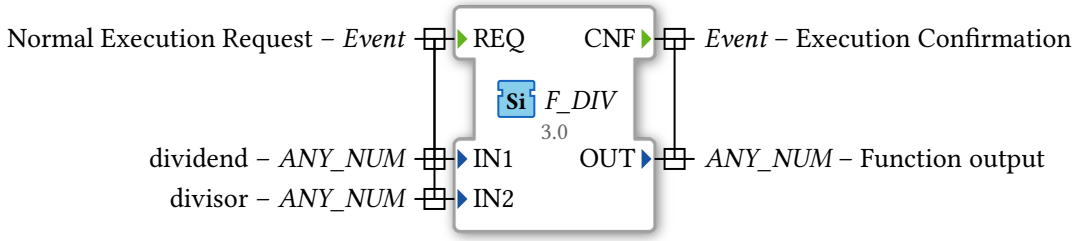
F_DIV



• – ◦ • – ◦ • – ◦ •

Introduction

The **F_DIV** is a standards-compliant function block for dividing numeric values, developed under the EPL-2.0 license. Version 1.0 enables type-safe division of various numeric data types according to the IEC 61131-3 standard.



Interface Structure

Event Inputs

- `REQ`: Calculation request (with dividend and divisor)

Event Outputs

- `CNF`: Calculation confirmation (with division result)

Data Inputs

- `IN1` (ANY_NUM): Dividend (numerator)
- `IN2` (ANY_NUM): Divisor (denominator)

Data Outputs

- `OUT` (ANY_NUM): Result of division (IN1 / IN2)

Functional Principle

1. **Calculation Trigger**:

- `REQ` event with values for IN1 and IN2
- Both inputs must be of type ANY_NUM

1. **Division**:

- Type-preserving calculation (IN1 / IN2)
- Supported types:
- Integers (INT, DINT)
- Floating-point numbers (REAL, LREAL)
- Mixed types (automatic conversion)

1. **Result Output**:

- `CNF` event with calculated OUT value
- Floating-point result for integer division

Technical Features

✓ **Type-safe division** (ANY_NUM support) ✓ **Automatic type conversion** ✓ **IEC 61131-3 compliant** ✓ **Deterministic execution**

Error Handling

- **Division by zero**:
- OUT = 0 (for Integers)
- OUT = INF/NAN (for floating-point numbers)
- No execution abort
- **Type incompatibility**:
- OUT = 0
- No exception generation

Application Scenarios

- **Process control**: Control deviation calculation
- **Data analysis**: Normalization of measured values
- **Machine control**: Speed calculations
- **Energy management**: Efficiency calculations

⚖️ Comparison with similar function blocks

Feature	F_DIV	F_MUL	F_ADD
Operation	Division	Multiplication	Addition
Inputs	2	2	2
Type support	ANY_NUM	ANY_NUM	ANY_NUM

Supported Data Types

Category	Examples
Integers	INT, DINT, UDINT
Floating Point	REAL, LREAL
Type Combinations	INT/REAL, DINT/LREAL

🛠️ Related Exercises

- Exercise_111

Conclusion

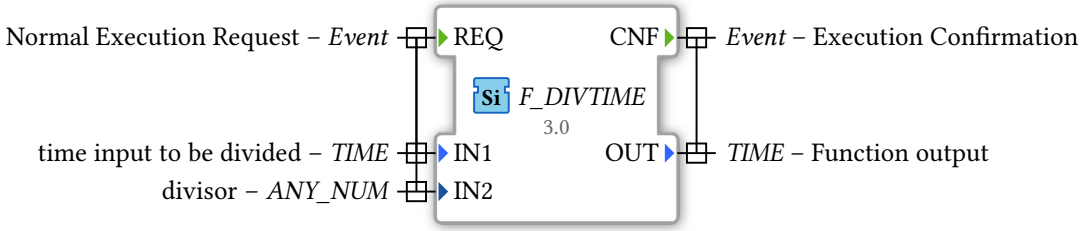
The F_DIV function block offers a robust solution for division operations:

- Type-safe calculation
- Flexible input combinations
- Standards-compliant implementation

Particularly valuable for applications requiring precise calculations with variable inputs. Automatic type conversion and error handling make it a reliable function block in industrial control systems.

F_DIVTIME

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_DIVTIME** function block is a specialized function module for dividing time values by numerical factors, developed under the EPL 2.0 license. Version 1.0 enables precise time calculations using TIME and ANY_NUM data types.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with input values)

Event Outputs

- `CNF`: Calculation confirmation (with result)

Data Inputs

- `IN1` (TIME): Time duration to be divided
- `IN2` (ANY_NUM): Divisor (numeric value)

Data Outputs

- `OUT` (TIME): Result of the division (IN1 / IN2)

Functionality

1. **Activation**:

- Triggered by the `REQ` event
- Processes TIME values and numeric divisors

1. **Calculation**:

- Divides time value (IN1) by numeric value (IN2)
- Performs automatic type conversion
- Calculation accurate to milliseconds

1. **Result Output**:

- `CNF` event with new TIME value
- Format: T#1d5h30m15s250ms

Technical Specifications

✓ Precise Time Division ✓ Support for ANY_NUM divisors ✓ Calculation accurate to milliseconds ✓ Automatic Type Conversion

Application Examples

- **Process Optimization**: Average Cycle Time Calculation
- **Production Planning**: Time Allocation to Stations
- **Robot Control**: Speed Adjustment
- **Energy Management**: Consumption Time Calculations

Error Handling

- **Division by Zero**:
- OUT = T#0s
- No Aborting Execution
- **Type Incompatibility**:
- OUT = IN1 (original value is retained)
- No exception generation

Special Notes

- **Calculation Accuracy**:
- Minimum 1 millisecond
- Rounding to the nearest millisecond
- **Validity Range**:
- Positive time values (T#0s to T#106751d23h47m16s854ms)
- Divisor must be positive

⚖️ **Comparison with Standard F_DIV**

Feature	F_DIVTIME	F_DIV
Input Type 1	TIME	ANY_NUM
Input Type 2	ANY_NUM	ANY_NUM
Output Type	TIME	ANY_NUM

Conclusion

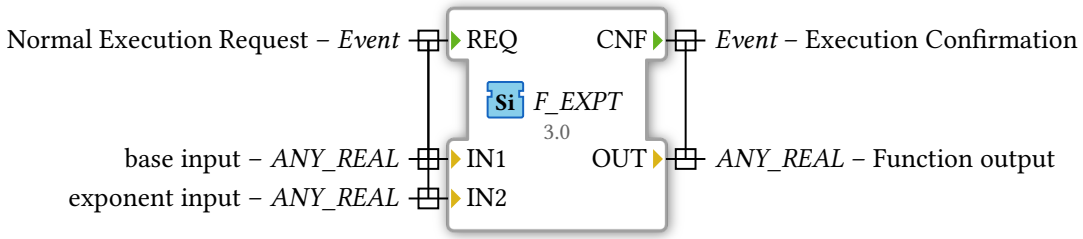
The F_DIVTIME function block offers essential functions for time-based calculations:

- Precise division of time values
- Flexible input parameters
- Reliable error handling

Especially valuable for applications requiring the division of time intervals. The specialized implementation guarantees correct results for all valid input combinations.

F_EXPT

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_EXPT** function block is a mathematical function module for calculating power values, developed under the EPL-2.0 license. Version 1.0 enables the calculation of base-exponent combinations with ANY_REAL data types.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with base and exponent)

Event Outputs

- `CNF`: Calculation confirmation (with result)

Data Inputs

- `IN1` (ANY_REAL): Base value
- `IN2` (ANY_REAL): Exponent

Data Outputs

- `OUT` (ANY_REAL): Result of exponentiation (IN1^IN2)

Functionality

1. **Activation**:

- Triggered by the `REQ` event
- Processes two ANY_REAL values

1. **Calculation**:

- Calculates IN1 raised to the power of IN2
- Supports all REAL and LREAL combinations
- Automatic type conversion (higher accuracy)

1. **Result Output**:

- `CNF` event with calculated power value
- Result in the more accurate input type

Technical Specifications

✓ Precise power calculation
✓ Support for REAL and LREAL
✓ Automatic type conversion
✓ IEC 61131-3 compliant

Application Examples

- **Physical calculations**: Acceleration formulas
- **Financial mathematics**: Compound interest calculation
- **Control engineering**: Nonlinear characteristics

- **Signal processing**: Exponential filters

Error Handling

- **Invalid inputs**:
- Negative base with non-integer exponent: OUT = 0
- Zero to a negative exponent: OUT = +INF
- Infinity calculations according to IEEE 754

Special Notes

- **Accuracy**:
- LREAL inputs produce LREAL results
- REAL inputs produce REAL results
- Mixed inputs: LREAL result
- **Performance**:
- LREAL calculations require more cycles
- For real-time applications, prefer REAL

⚖️ **Comparison with similar building blocks**

Feature	F_EXPT	F_LN	F_POWER
Function	a^b	ln(a)	a^b
Input types	ANY_REAL	ANY_REAL	ANY_NUM
Accuracy	High	High	Medium

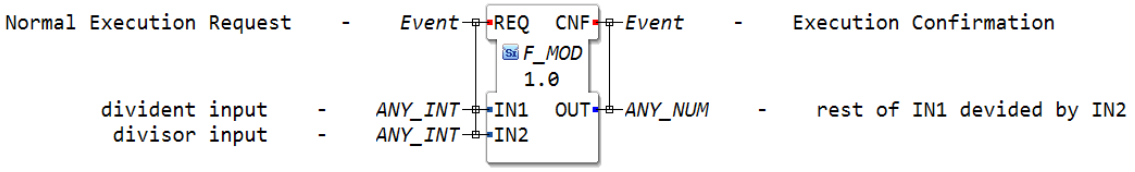
Conclusion

The F_EXPT function block offers powerful mathematical functions:

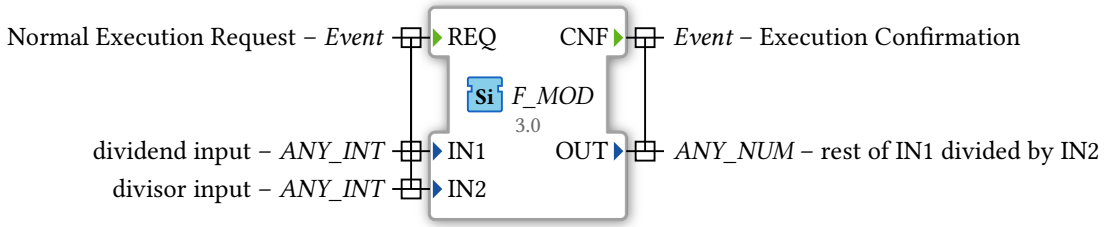
- Precise exponential calculations
- Flexible input combinations
- Reliable error handling

Indispensable for advanced control algorithms requiring nonlinear calculations. Its standards-compliant implementation enables its use in demanding industrial applications.

F_MOD



• – ◦ • – ◦ • – ◦ •



Introduction

The **F_MOD** is an arithmetic function block for calculating the remainder of a division, developed under the EPL-2.0 license. Version 1.0 enables the modulo operation for integer data types according to the IEC 61131-3 standard.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with dividend and divisor)

Event Outputs

- `CNF`: Calculation confirmation (with remainder)

Data Inputs

- `IN1` (ANY_INT): Dividend (integer)
- `IN2` (ANY_INT): Divisor (integer)

Data Outputs

- `OUT` (ANY_NUM): Remainder of division (IN1 mod IN2)

Functionality

1. **Activation**:

- Triggered by the `REQ` event
- Processes two integer values

1. **Modulo Operation**:

- Calculates IN1 modulo IN2
- Formula: OUT = IN1 - (IN1 / IN2) * IN2
- Supported types: INT, DINT, UINT, UDINT

1. **Result Output**:

- `CNF` event with residual value
- Sign corresponds to dividend (IN1)

Technical Specifications

✓ Integer Modulo Operation ✓ Support for Signed and Unsigned Types ✓ IEC 61131-3 Compliant ✓ Deterministic Execution

Application Examples

- **Cyclic Counting**: Ring Buffer Indices
- **Time Calculations**: Weekday Determination
- **Message Processing**: Checksum Calculation
- **Positioning**: Stepper Motor Control

Error Handling

- **Division by Zero**:
- OUT = IN1 (Original value is retained)
- No execution interruption
- **Type incompatibility**:
- Automatic type matching
- No exception generation

Special Notes

- **Sign handling**:
- Result has the sign of the dividend
- Example: (-7 mod 3) = −1
- **Performance**:
- Optimized calculation for all integer types
- One cycle latency

⚖️ Comparison with similar function blocks

Feature	F_MOD	F_DIV	F_REM

Operation	Modulo	Division	Remainder
Input types	ANY_INT	ANY_NUM	ANY_NUM
Output type	ANY_NUM	ANY_NUM	ANY_NUM

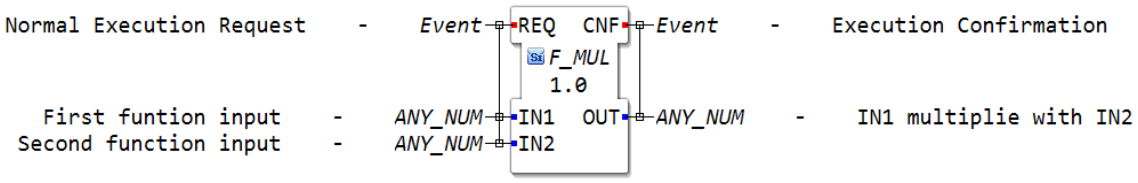
Conclusion

The F_MOD function block offers essential functions for cyclic calculations:

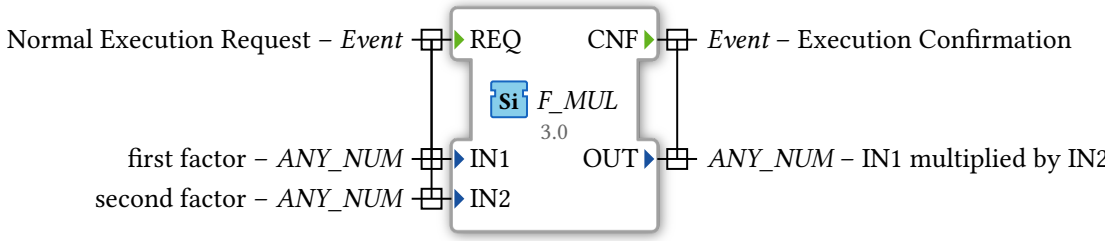
- Efficient residual value calculation
- Flexible type support
- Reliable error handling

Indispensable for control applications with periodic patterns or index calculations. Its standards-compliant implementation enables its use in real-time systems with deterministic requirements.

F_MUL



• – ◦ • – ◦ • – ◦ •



Introduction

The **F_MUL** is a standards-compliant function block for multiplying numeric values, developed under the EPL-2.0 license. Version 1.0 enables type-safe multiplication of various numeric data types according to the IEC 61131-3 standard.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with two factors)

Event Outputs

- `CNF`: Calculation confirmation (with result)

Data Inputs

- `IN1` (ANY_NUM): First factor
- `IN2` (ANY_NUM): Second factor

Data Outputs

- `OUT` (ANY_NUM): Product of the multiplication (IN1 × IN2)

Operating Principle

1. **Calculation Trigger**:

- `REQ` event with values for IN1 and IN2
- Both inputs must be of type ANY_NUM be

1. **Multiplication**:

- Type-preserving calculation (IN1 × IN2)
- Supported types:
- Integers (INT, DINT)
- Floating-point numbers (REAL, LREAL)
- Mixed types (automatic conversion)

1. **Result output**:

- `CNF` event with calculated product
- Result in the "largest" input type

Technical Features

✓ **Type-safe multiplication** (ANY_NUM support) ✓ **Automatic type conversion** ✓ **Overflow protection** ✓ **Deterministic execution**

Application Scenarios

- **Process control**: Scaling calculations
- **Data analysis**: Measured value processing
- **Machine control**: Position calculations
- **Energy management**: Power calculations

Error handling

- **Overflow handling**:
- Wrap-around in case of Integers
- ±INF (± infinity) on floating-point overflow
- **Type incompatibility**:
- OUT = 0
- No exception generation

⚖️ **Comparison with similar function blocks**

Feature	F_MUL	F_ADD	F_DIV
Operation	Multiplication	Addition	Division
Inputs	2	2	2
Type support	ANY_NUM	ANY_NUM	ANY_NUM

🛠️ **Related Exercises**

- Exercise_034b

Conclusion

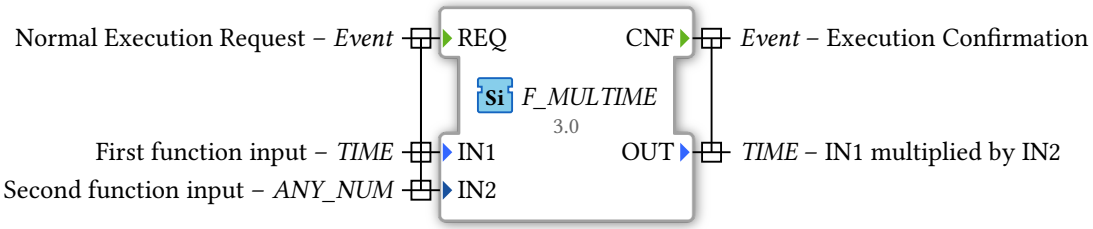
The F_MUL function block offers a robust solution for multiplication operations:

- Precise type-preserving calculations
- Flexible input combinations
- Standards-compliant implementation

Particularly valuable for applications that need to implement scaled calculations or physical formulas. Automatic type conversion makes it a versatile tool in industrial control systems.

F_MULTIME

• − ◦ • − ◦ • − ◦ •



Introduction

The **F_MULTIME** function block is a standards-compliant function block for multiplying time values by numerical factors, developed under the EPL-2.0 license. Version 1.0 enables type-safe scaling of time intervals according to the IEC 61131-3 standard.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with time value and multiplier)

Event Outputs

- `CNF`: Calculation confirmation (with scaled time value)

Data Inputs

- `IN1` (TIME): Time value (base value)
- `IN2` (ANY_NUM): Scaling factor

Data Outputs

- `OUT` (TIME): Scaled time value (IN1 × IN2)

Functional Principle

1. **Calculation Trigger**:

- `REQ` event with time value (IN1) and factor (IN2)
- IN1 must be of type TIME, IN2 must be of type ANY_NUM

1. **Multiplication**:

- Time value is converted to nanoseconds
- Multiplication by a numeric factor
- Conversion back to TIME data type

1. **Result Output**:

- `CNF` event with scaled time value
- Result always of type TIME

Technical Features

✓ **Type-safe time value processing**
  ✓ **Precise nanosecond calculation**
  ✓ **Automatic type conversion** (ANY_NUM ⟶ REAL)
  ✓ **Overflow protection** (limit to maximum TIME value)

Application Scenarios

- **Cycle time adjustment**: Scaling of time intervals
- **Process control**: Lengthening/shortening of timeouts
- **Motion control**: Speed adjustments
- **Scheduling**: Dynamic Schedule Scaling

Error Handling

- **Overflow Handling**:
- Limit to T#24d20h31m23s647ms (Maximum)
- No wrap-around for time values

- **Type Incompatibility**:
- OUT = T#0s
- No exception generation

⚖️ Comparison with Similar Function Blocks

Feature	F_MULTIME	F_MUL	F_DIVTIME
Operation	Time Multiplication	Numeric Multiplication	Time Division
Inputs	TIME × ANY_NUM	ANY_NUM × ANY_NUM	TIME ÷ ANY_NUM
Output Type	TIME	ANY_NUM	TIME

🛠️ Related Exercises

- Exercise_020c2

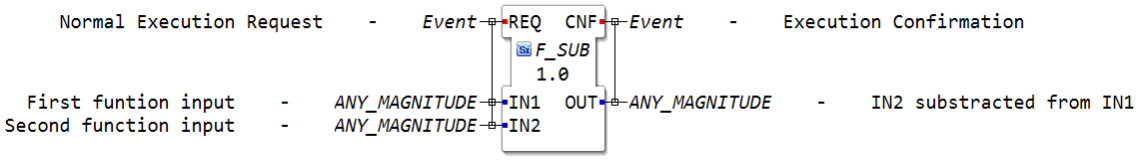
Conclusion

The F_MULTIME function block offers a specialized solution for scaling time values:

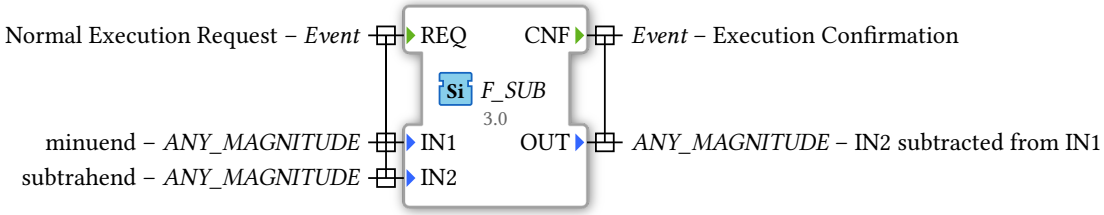
- Industrial-grade precision (nanosecond basis)
- Safe type conversion
- Robust overflow handling

Essential for applications with dynamically adjustable time parameters and cyclical control processes.

F_SUB



• – ◦ • – ◦ • – ◦ •



Introduction

The **F_SUB** is a standards-compliant function block for subtracting numeric values, developed under the EPL-2.0 license. Version 1.0 enables type-safe subtraction of various numeric data types according to the IEC 61131-3 standard.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with minuend and subtrahend)

Event Outputs

- `CNF`: Calculation confirmation (with result)

Data Inputs

- `IN1` (ANY_MAGNITUDE): Minuend (value from which to subtract)
- `IN2` (ANY_MAGNITUDE): Subtrahend (value to be subtracted)

Data Outputs

- `OUT` (ANY_MAGNITUDE): Difference of the subtraction (IN1 - IN2)

Operating Principle

1. **Calculation Trigger**:

- `REQ` event with values for IN1 and IN2
- Both inputs must be of type ANY_MAGNITUDE

1. **Subtraction**:

- Type-preserving calculation (IN1 - IN2)
- Supported types:
- Integers (INT, DINT)
- Floating-point numbers (REAL, LREAL)
- Time values (TIME, DATE)
- Mixed types (automatic conversion)

1. **Result output**:

- `CNF` event with calculated difference
- Result in the "largest" input type

Technical Features

✓ **Type-safe subtraction** (ANY_MAGNITUDE support) ✓ **Automatic type conversion** ✓ **Underrun protection** ✓ **Deterministic execution**

Application Scenarios

- **Process control**: Deviation Calculation
- **Data Analysis**: Difference Calculation of Measured Values
- **Machine Control**: Position Differences
- **Time Control**: Time Interval Calculations

Error Handling

- **Underflow Handling**:
- Wrap-around for integers
- -INF for floating-point underflow
- **Type Incompatibility**:
- OUT = 0
- No Exception Generation

⚖️ Comparison with Similar Function Blocks

Feature	F_SUB	F_ADD	F_MUL
Operation	Subtraction	Addition	Multiplication
Inputs	2	2	2
Type Support	ANY_MAGNITUDE	ANY_NUM	ANY_NUM

🛠️ Related Exercises

- Exercise_040
- Exercise_040_2
- Exercise_040_AX
- Exercise_041
- Exercise_072c

Conclusion

The F_SUB function block offers a robust solution for subtraction operations:

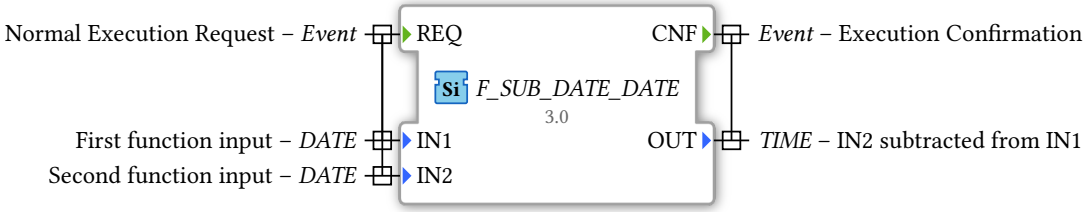
- Precise type-preserving calculations
- Flexible input combinations

- Standards-compliant implementation

Particularly valuable for applications that need to implement difference calculations or relative changes. Automatic type conversion makes it an indispensable tool in industrial control systems.

F_SUB_DATE_DATE

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_SUB_DATE_DATE** function block is a standards-compliant function block for calculating time differences between two date values, developed under the EPL-2.0 license. Version 1.0 enables the precise calculation of time intervals according to the IEC 61131-3 standard.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with start and end date)

Event Outputs

- `CNF`: Calculation confirmation (with time difference)

Data Inputs

- `IN1` (DATE): Start date (minuend)
- `IN2` (DATE): End date (subtrahend)

Data Outputs

- `OUT` (TIME): Calculated time interval (IN1 - IN2)

Operating Principle

1. **Calculation Trigger**:

- `REQ` event with two DATE values
- Both inputs must be of type DATE

1. **Difference Calculation**:

- Conversion of both date values to seconds since January 1, 1970
- Subtraction of the timestamps (IN1 - IN2)
- Conversion to the TIME data type

1. **Result Output**:

- `CNF` event with calculated time interval
- Positive values if IN1 > IN2, negative values if IN1 < IN2

Technical Features

✓ **Precise Time Difference** (Accurate to the Day) ✓ **Leap Year Consideration**
✓ **Negative Time Intervals** for Retrospective Calculations ✓ **Platform-Independent Time Calculation**

Application Scenarios

- **Process Duration Calculation**: Production time between two dates
- **Maintenance Intervals**: Remaining time until next maintenance
- **Project Planning**: Time remaining until deadline

- **Inventory Management**: Shelf Life Calculation

Error Handling

- **Date Limits**:
- Supports the range 1970-01-01 to 2106-02-07
- On exceedance: OUT = T#0s
- **Type Incompatibility**:
- OUT = T#0s
- No exception generation

⚖️ **Comparison with Similar Function Blocks**

Feature	F_SUB_DATE_DATE	F_SUB_TIME	F_SUB_DT_DT
Input Types	DATE - DATE	TIME - TIME	DT - DT
Output Type	TIME	TIME	TIME
Accuracy	Day-accurate	Nanosecond	Microsecond

Conclusion

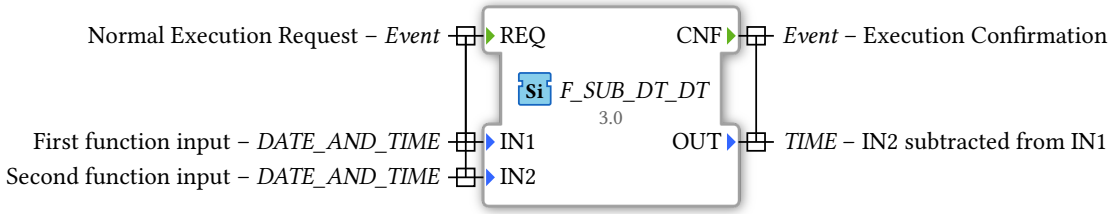
The F_SUB_DATE_DATE function block offers an optimized solution for date differences:

- Calendar-accurate calculations
- Easy handling of time intervals
- Standards-compliant implementation

Particularly valuable for applications with deadline-related calculations and long-term scheduling.

F_SUB_DT_DT

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_SUB_DT_DT** is a standards-compliant function block for the precise calculation of time differences between two timestamps (DATE_AND_TIME), developed under the EPL-2.0 license. Version 1.0 enables microsecond-accurate time interval calculations according to the IEC 61131-3 standard.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with start and end timestamps)

Event Outputs

- `CNF`: Calculation confirmation (with time difference)

Data Inputs

- `IN1` (DATE_AND_TIME): Start time (minuend)
- `IN2` (DATE_AND_TIME): End time (subtrahend)

Data Outputs

- `OUT` (TIME): Calculated time interval (IN1 - IN2)

Functional Principle

1. **Calculation Trigger**:

- `REQ` event with two DATE_AND_TIME values
- Format: DT#1970-01-01-00:00:00 to DT#2106-02-07-06:28:15

1. **Difference Calculation**:

- Conversion to epoch microseconds
- Subtraction of timestamps (IN1 - IN2)
- Conversion to TIME data type

1. **Result Output**:

- `CNF` event with calculated interval
- Positive values for IN1 > IN2, negative values for IN1 < IN2

Technical Features

✓ **Microsecond Accuracy**
✓ **Time Zone-Independent Calculation** (UTC Base)
✓ **Full Calendar Arithmetic** (Leap Seconds, Month Lengths) ✓ **Cross-Platform Consistency**

Application Scenarios

- **Process Analysis**: Precise Machine Runtime Measurement
- **Laboratory Automation**: Precise Response Time Calculation
- **Energy Monitoring**: Consumption Period Differences
- **PLC Diagnostics**: Event Timestamp Evaluation

Error Handling

- **Time Range Exceedance**:
- Supports DT#1970-01-01-00:00:00 to DT#2106-02-07-06:28:15
- On Overflow: OUT = T#0s
- **Invalid Inputs**:
- OUT = T#0s for invalid timestamps
- No Exception Generation

⚖️ Comparison with Similar Function Blocks

Feature	F_SUB_DT_DT	F_SUB_DATE_DATE	F_SUB_TIME
Accuracy	Microseconds	Days	Nanoseconds
Input Type	DATE_AND_TIME	DATE	TIME
Typical Application	Process Diagnostics	Scheduling	Cycle Time

Conclusion

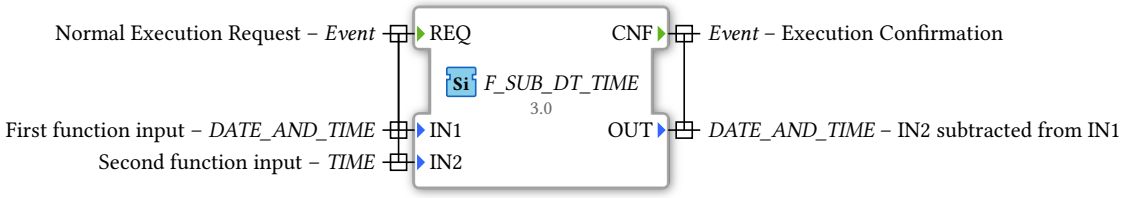
The F_SUB_DT_DT block offers the most precise solution for timestamp discrepancies:

- Industrial-grade microsecond accuracy
- Calendar-accurate UTC calculations
- Robust handling of large time intervals

Essential for high-precision process analysis and technical diagnostic systems.

F_SUB_DT_TIME

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_SUB_DT_TIME** function block is a standards-compliant function block for subtracting time intervals from timestamps, developed under the EPL-2.0 license. Version 1.0 enables precise time calculations according to the IEC 61131-3 standard.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with timestamp and time interval)

Event Outputs

- `CNF`: Calculation confirmation (with corrected timestamp)

Data Inputs

- `IN1` (DATE_AND_TIME): Base time
- `IN2` (TIME): Time interval to be subtracted

Data Outputs

- `OUT` (DATE_AND_TIME): Calculated time (IN1 - IN2)

Functional Principle

1. **Calculation Trigger**:

- `REQ` event with DATE_AND_TIME and TIME value
- Time range: DT#1970-01-01-00:00:00 to DT#2106-02-07-06:28:15

1. **Time Calculation**:

- Conversion to epoch microseconds
- Subtraction of the time interval
- Calendar-accurate back-calculation

1. **Result Output**:

- `CNF` event with new timestamp
- Automatic calendar correction (leap years, month ends)

Technical Features

✓ Microsecond accuracy ✓ Complete calendar arithmetic ✓ Negative time intervals (time shift into the future) **✓ Cross-platform consistency**

Application Scenarios

- **Scheduling**: Deadline calculation
- **Process control**: Predetermining start times
- **Maintenance systems**: Planning preventive maintenance Maintenance
- **Data Analysis**: Historical Time Correction

Error Handling

- **Time Range Exceedance**:
- In case of underflow: OUT = DT#1970-01-01-00:00:00
- In case of overflow: OUT = DT#2106-02-07-06:28:15
- **Special Cases**:
- IN2 = T#0s ⟶ OUT = IN1
- IN2 negative ⟶ Addition at time

⚖️ Comparison with similar function blocks

Feature	F_SUB_DT_TIME	F_ADD_DT_TIME	F_SUB_DT_DT
Operation	Time - Interval	Time + Interval	Time - Time
Result	DATE_AND_TIME	DATE_AND_TIME	TIME
Accuracy	Microseconds	Microseconds	Microseconds

Conclusion

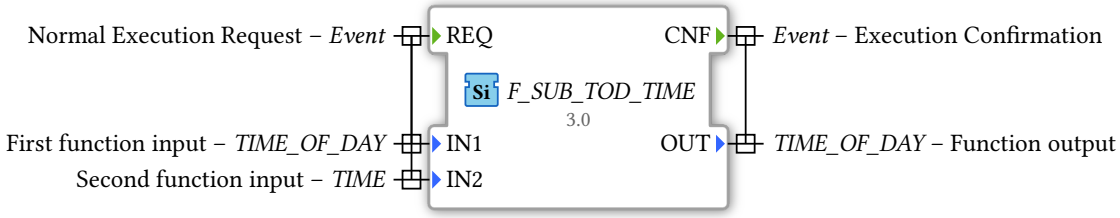
The F_SUB_DT_TIME function block offers a reliable solution for time calculations:

- Industrial-grade precision
- Full calendar support
- Intuitive handling of time intervals

Essential for all applications involving date-related calculations and historical time analysis.

F_SUB_TOD_TIME

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_SUB_TOD_TIME** function block is a standards-compliant function block for subtracting time intervals from times of day (TIME_OF_DAY), developed under the EPL 2.0 license. Version 1.0 enables time-of-day-related calculations according to the IEC 61131-3 standard.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with time of day and time interval)

Event Outputs

- `CNF`: Calculation confirmation (with corrected time of day)

Data Inputs

- `IN1` (TIME_OF_DAY): Base time of day (TOD#hh:mm:ss)
- `IN2` (TIME): Time interval to be subtracted

Data Outputs

- `OUT` (TIME_OF_DAY): Calculated time of day (IN1 - IN2)

Operating Principle

1. **Calculation Trigger**:

- `REQ` event with TIME_OF_DAY and TIME value
- Valid range: TOD#00:00:00 to TOD#23:59:59.999999

1. **Time Calculation**:

- Conversion to seconds since midnight
- Subtraction of the time interval
- Time-of-day calculation with overflow handling

1. **Result Output**:

- `CNF` event with new time of day
- Automatic 24-hour overflow (circular calculation)

Technical Features

✓ **Microsecond Accuracy** ✓ **24-hour Overflow Handling** (circular time of day) ✓ **Negative Intervals** (time addition) ✓ **Real-time Calculation**

Application Scenarios

- **Shift Planning**: Calculating the start of work
- **Lighting Control**: Correct sunrise times
- **Production Planning**: Calculate machine run times
- **Traffic Control Systems**: Schedule adjustments

Error Handling

- **Overflow Handling**:
- In case of overrun: circular calculation (modulo 24h)
- Example: TOD#01:00:00 - T#2h = TOD#23:00:00
- **Special Cases**:
- IN2 = T#0s ⟶ OUT = IN1
- IN2 negative ⟶ Add the time (TOD#10:00 + T#−2h = TOD#12:00)

⚖️ **Comparison with similar function blocks**

Feature	F_SUB_TOD_TIME	F_ADD_TOD_TIME	F_SUB_DT_TIME
Operation	Time of Day - Interval	Time of Day + Interval	Timestamp - Interval
Result	TIME_OF_DAY	TIME_OF_DAY	DATE_AND_TIME
Overflow	24-hour Circular	24-hour Circular	Calendar-based

Conclusion

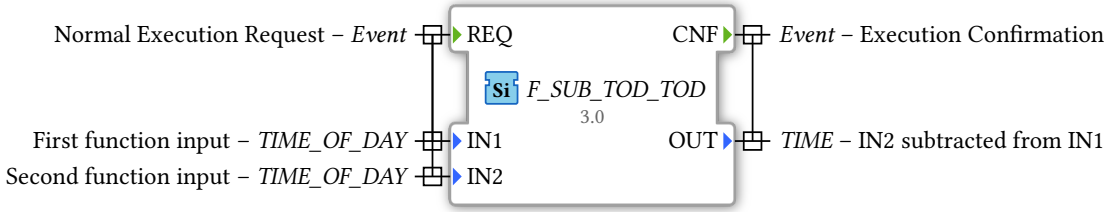
The F_SUB_TOD_TIME block offers an optimized solution for time-of-day calculations:

- Precise circular time calculation
- Easy handling of shift and operating hours
- Intuitive handling of day transitions

Ideal choice for all applications with time-of-day-dependent controls.

F_SUB_TOD_TOD

• − ◦ • − ◦ • − ◦ •



Introduction

The **F_SUB_TOD_TOD** function block is a standards-compliant function block for calculating time differences between two times of day (TIME_OF_DAY), developed under the EPL-2.0 license. Version 1.0 enables precise time-of-day calculations according to the IEC 61131-3 standard.

Interface Structure

Event Inputs

- `REQ`: Calculation request (with start and end time of day)

Event Outputs

- `CNF`: Calculation confirmation (with time difference)

Data Inputs

- `IN1` (TIME_OF_DAY): Start time of day (TOD#hh:mm:ss)
- `IN2` (TIME_OF_DAY): End time of day

Data Outputs

- `OUT` (TIME): Calculated time interval (IN1 - IN2)

Operating Principle

1. **Calculation Trigger**:

- `REQ` event with two TIME_OF_DAY values
- Valid range: TOD#00:00:00 to TOD#23:59:59.999999

1. **Difference Calculation**:

- Conversion of both values to seconds since midnight
- Difference calculation with 24-hour overflow consideration
- Result as TIME data type

1. **Result Output**:

- `CNF` event with calculated interval
- Positive values for IN1 > IN2 (within the same day)
- Negative values at the end of the day

Technical Features

✓ Microsecond Accuracy ✓ Automatic 24-hour Overflow Handling
✓ Bidirectional Difference Calculation
✓ Real-Time Capable Execution

Application Scenarios

- **Time and Attendance Recording**: Shift Duration Calculation
- **Energy Management**: Peak Load Analysis
- **Production Logistics**: Throughput Time Measurement
- **Building Automation**: Daylight Utilization Times

Error Handling

- **Special Cases**:
- IN1 = IN2 ⟶ OUT = T#0s
- Across midnight: OUT = -(24h - Difference)
- Example: TOD#01:00:00 - TOD#23:00:00 = T#−22h
- **Limit Values**:
- Maximum Difference: ±T#23:59:59.999999
- No Overflow Exception

⚖️ Comparison with Similar Function Blocks

Feature	F_SUB_TOD_TOD	F_SUB_DT_DT	F_SUB_TIME
Input Types	TOD - TOD	DT - DT	TIME - TIME
Output Type	TIME	TIME	TIME
Special Features	24-hour circular	Calendar-based	Simple difference

Conclusion

The F_SUB_TOD_TOD function block offers the optimal solution for calculating time-of-day differences:

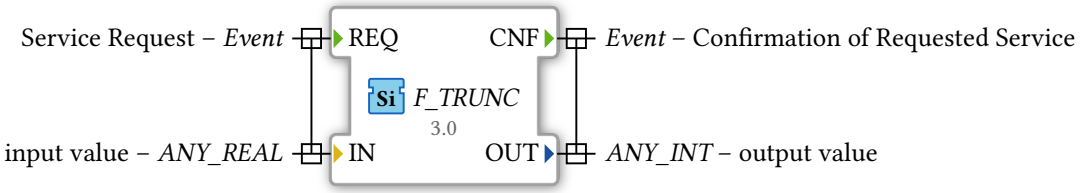
- Precise handling of day transitions
- Simple measurement of operating time intervals

- Standards-compliant implementation

Especially valuable for all applications with time-of-day-dependent process analyses.

F_TRUNC

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_TRUNC** function block is a standards-compliant function block for rounding floating-point numbers toward zero, developed under the EPL-2.0 license. Version 1.0 enables type-safe conversion according to the IEC 61131-3 standard.

•

Interface Structure

Event Inputs

- `REQ`: Calculation request (with input value)

Event Outputs

- `CNF`: Calculation confirmation (with rounded value)

Data Inputs

- `IN` (ANY_REAL): Input value (floating-point number)

Data Outputs

- `OUT` (ANY_INT): Rounded integer value

Functional Principle

1. **Calculation Trigger**:

- `REQ` event with an ANY_REAL value
- Supported types: REAL, LREAL

1. **Truncation Operation**:

- Removes decimal places without Rounding
- Example: 3.9 ⟶ 3, −2.7 ⟶ −2
- No mathematical rounding (≠ F_ROUND)

1. **Result Output**:

- `CNF` event with integer result
- Automatic type conversion (INT, DINT, etc.)

Technical Features

✓ Lossless Type Conversion
✓ Consistent Truncation Toward Zero
✓ Deterministic Execution
✓ No Overflow Exceptions

Application Scenarios

- **Data Preprocessing**: Index Calculations
- **Measurement Processing**: Discretization
- **Control Logic**: Integer Position Calculations

- **Financial Calculations**: Currency Conversions

Error Handling

- **Value Range Exceedance**:
- For values that are too large: Limit to the target type maximum
- No exception generation
- **Special Cases**:
- NaN ⟶ 0

±INF ⟶ Maximum value of the target type

⚖️ **Comparison with similar function blocks**

Feature	F_TRUNC	F_ROUND	F_FLOOR
Operation	Direction Zero	Mathematical	Round Down
Example	3.9 ⟶ 3	3.9 ⟶ 4	3.9 ⟶ 3
Example	−2.7 ⟶ −2	−2.7 ⟶ −3	−2.7 ⟶ −3

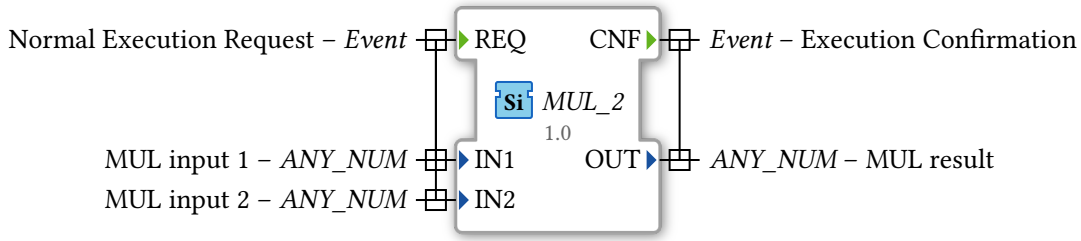
Conclusion

The F_TRUNC function block offers a precise solution for integer conversion:

- Predictable behavior (truncation toward zero)
- Type-safe floating-point-to-integer conversion
- Robust handling of extreme values

Essential for all applications with precise range boundaries and index calculations.

MUL_2



• – ◦ • – ◦ • – ◦ •

Introduction

The **MUL_2** is a generic function block that calculates the arithmetical product of two numeric inputs (`OUT = IN1 * IN2`). The data types of the ports (`ANY_NUM`) are fixed on instantiation; the native implementation is `GEN_MUL`.

Interface Structure

Event Inputs

- **REQ**: normal execution request (with `IN1`, `IN2`)

Event Outputs

- **CNF**: execution confirmation (with `OUT`)

Data Inputs

- **IN1** (ANY_NUM): first factor
- **IN2** (ANY_NUM): second factor

Data Outputs

- **OUT** (ANY_NUM): product

Adapters

No adapters are used.

Functionality

On `REQ`, `IN1` and `IN2` are read, multiplied, and the product is output at `OUT` together with the event `CNF`.

Technical Details

- Generic block (generic class `GEN_MUL`): the port types are `ANY_NUM` and are fixed per instance
- Stateless service interface block

Application Scenarios

- Scaling a value by a factor in an event network
- Area, power or quantity calculations from two quantities

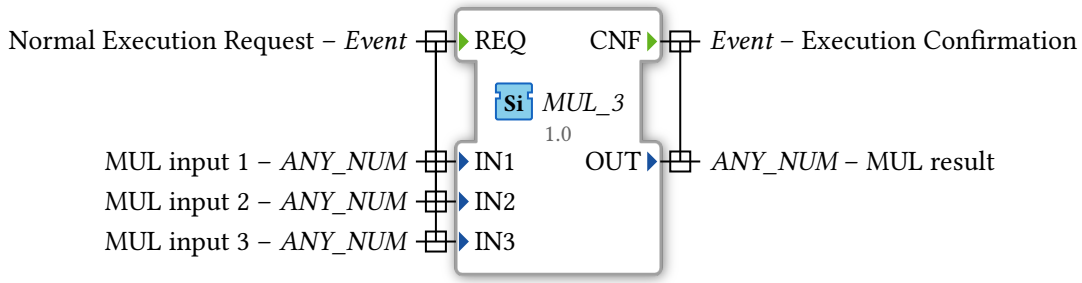
⚖️ Comparison with Similar Blocks

`ADD_2` correspondingly forms the sum of two values; for multiplication with adapters, there are the AR-based variants in the adapter library.

Conclusion

`MUL_2` multiplies two numeric values of arbitrary `ANY_NUM` type in an event-driven way.

MUL_3



• – ◦ • – ◦ • – ◦ •

Introduction

MUL_3 is a generic function block that calculates the arithmetic product of three numeric inputs (`OUT = IN1 * IN2 * IN3`). The data types of the ports (`ANY_NUM`) are fixed at instantiation; the native implementation is `GEN_MUL`.

Interface Structure

Event Inputs

- **REQ**: normal execution request (with `IN1`, `IN2`, `IN3`)

Event Outputs

- **CNF**: execution confirmation (with `OUT`)

Data Inputs

- **IN1** (ANY_NUM): first factor
- **IN2** (ANY_NUM): second factor
- **IN3** (ANY_NUM): third factor

Data Outputs

- **OUT** (ANY_NUM): product

Adapters

No adapters are used.

Functionality

On REQ, IN1, IN2 and IN3 are read, multiplied, and the product is output at OUT together with the event CNF.

Technical Details

- Generic block (generic class GEN_MUL): the port types are ANY_NUM and are fixed per instance
- Stateless service interface block

Application Scenarios

- Product of three quantities in an event network (e.g. volume calculation from length, width, height)

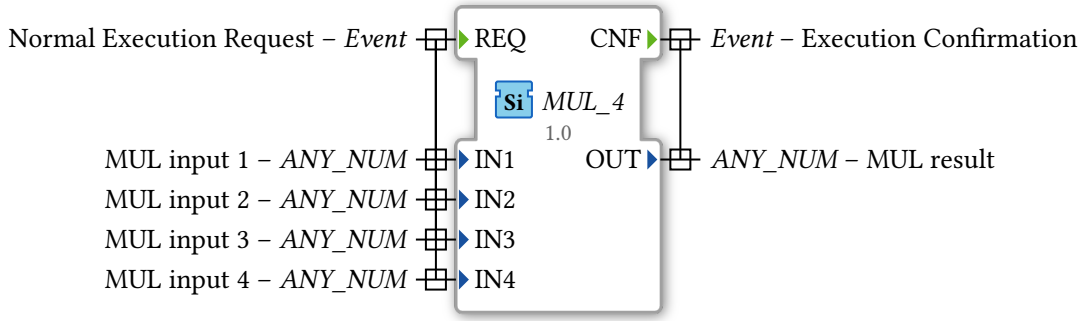
⚖️ Comparison with Similar Building Blocks

MUL_2 multiplies two values, MUL_3 three.

Conclusion

MUL_3 multiplies three numeric values of any ANY_NUM type in an event-driven manner.

MUL_4



• – ◦ • – ◦ • – ◦ •

Introduction

The **MUL_4** is a generic function block that calculates the arithmetic product of four numeric inputs (OUT = IN1 * IN2 * IN3 * IN4). The data types of the ports (ANY_NUM) are fixed at instantiation; the native implementation is GEN_MUL.

Interface Structure

Event Inputs

- **REQ**: normal execution request (with IN1, IN2, IN3, IN4)

Event Outputs

- **CNF**: execution confirmation (with OUT)

Data Inputs

- **IN1** (ANY_NUM): first factor
- **IN2** (ANY_NUM): second factor
- **IN3** (ANY_NUM): third factor
- **IN4** (ANY_NUM): fourth factor

Data Outputs

- **OUT** (ANY_NUM): product

Adapters

No adapters are used.

Functionality

On REQ, IN1, IN2, IN3 and IN4 are read, multiplied and the product is output at OUT together with the event CNF.

Technical Details

- Generic block (generic class `GEN_MUL`): the port types are `ANY_NUM` and are fixed per instance
- Stateless service interface block
- Version 1.0 (2026-08-28)

Application Scenarios

- Product of four quantities in an event network

⚖️ Comparison with Similar Building Blocks

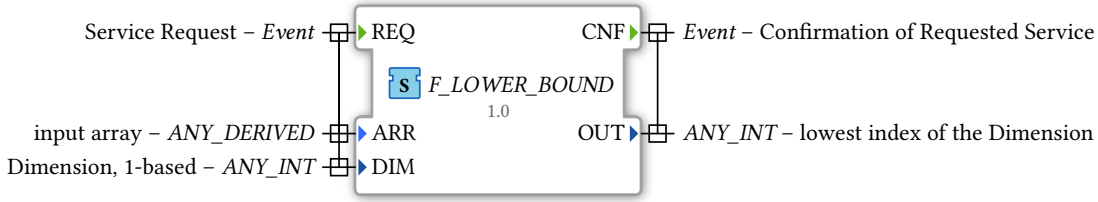
`MUL_2` multiplies two values, `MUL_3` three, `MUL_4` four.

Conclusion

`MUL_4` multiplies four numeric values of any `ANY_NUM` type in an event-driven manner.

2.11.4.2 Arrays

F_LOWER_BOUND



• – ◦ • – ◦ • – ◦ •

Introduction

`F_LOWER_BOUND` returns the lower index bound of a selected dimension of an array of arbitrary type. Together with F_UPPER_BOUND, it forms the basis for generic, size-independent array processing — e.g. `F_LEN_ARRAY` (see `utils::arrays`) is computed from both bounds.

Interface Structure

Event Inputs

- **REQ**: Triggers the query, carries `ARR` and `DIM`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **ARR** (ANY_DERIVED): The array whose index bound is to be determined.
- **DIM** (ANY_INT, default `UDINT#1`): The array dimension to consider, 1-based.

Data Outputs

- **OUT** (ANY_INT): The lowest index of the selected dimension.

Functionality

On `REQ`, the block determines the lower index bound of the dimension of `ARR` selected via `DIM` and outputs it via `OUT`. `CNF` is then triggered.

Technical Features

- **`ANY_DERIVED` input**: `ARR` accepts arrays of any element type and any dimensionality.
- **1-based dimension selection**: `DIM = 1` denotes the first dimension; for multi-dimensional arrays, this lets a specific dimension be queried.
- **Not necessarily `0`/greater than `0`**: IEC 61131-3 arrays can be declared with arbitrary index bounds (including negative, or not starting at `0`) — `F_LOWER_BOUND` returns the actually declared bound, making no assumption about its value.

State Overview

Stateless: every `REQ` immediately results in the query and `CNF`.

Application Scenarios

- **Generic array processing**: Loops that iterate over an array's elements independent of its actually declared size/index bounds.
- **Building block for length calculation**: `F_LEN_ARRAY` (`utils::arrays`) combines `F_LOWER_BOUND` with `F_UPPER_BOUND` to compute the element count of a dimension.

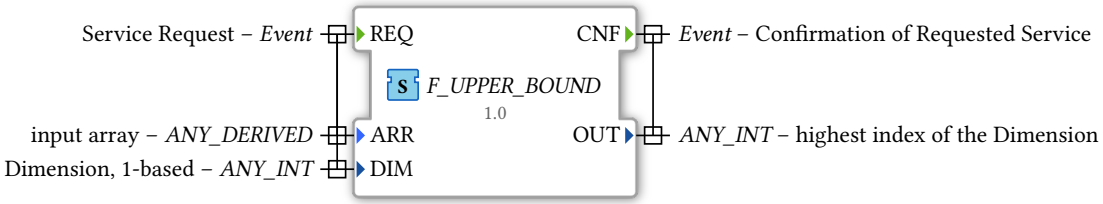
Comparison with similar function blocks

- F_UPPER_BOUND: the other index bound of the same dimension.
- **`F_LEN_ARRAY`** (`utils::arrays`): computes the element count from `F_LOWER_BOUND` and `F_UPPER_BOUND`.

Conclusion

`F_LOWER_BOUND` provides the lower index bound of an array dimension and is a fundamental building block for generic, size-independent array processing in IEC 61131-3 applications.

F_UPPER_BOUND



• – ◦ • – ◦ • – ◦ •

Introduction

`F_UPPER_BOUND` returns the upper index bound of a selected dimension of an array of arbitrary type. Together with F_LOWER_BOUND, it forms the basis for generic, size-independent array processing — e.g. `F_LEN_ARRAY` (see `utils::arrays`) is computed from both bounds.

Interface Structure

Event Inputs

- **REQ**: Triggers the query, carries `ARR` and `DIM`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **ARR** (ANY_DERIVED): The array whose index bound is to be determined.
- **DIM** (ANY_INT, default `UDINT#1`): The array dimension to consider, 1-based.

Data Outputs

- **OUT** (ANY_INT): The highest index of the selected dimension.

Functionality

On `REQ`, the block determines the upper index bound of the dimension of `ARR` selected via `DIM` and outputs it via `OUT`. `CNF` is then triggered.

Technical Features

- **`ANY_DERIVED` input**: `ARR` accepts arrays of any element type and any dimensionality.
- **1-based dimension selection**: `DIM = 1` denotes the first dimension; for multi-dimensional arrays, this lets a specific dimension be queried.
- **Not necessarily `0`/greater than `0`**: IEC 61131-3 arrays can be declared with arbitrary index bounds (including negative, or not starting at `0`) — `F_UPPER_BOUND` returns the actually declared bound, making no assumption about its value.

State Overview

Stateless: every `REQ` immediately results in the query and `CNF`.

Application Scenarios

- **Generic array processing**: Loops that iterate over an array's elements independent of its actually declared size/index bounds.
- **Building block for length calculation**: `F_LEN_ARRAY` (`utils::arrays`) combines `F_UPPER_BOUND` with `F_LOWER_BOUND` to compute the element count of a dimension.

Comparison with similar function blocks

- **F_LOWER_BOUND**: the other index bound of the same dimension.
- **`F_LEN_ARRAY`** (`utils::arrays`): computes the element count from `F_UPPER_BOUND` and `F_LOWER_BOUND`.

Conclusion

`F_UPPER_BOUND` provides the upper index bound of an array dimension and is a fundamental building block for generic, size-independent array processing in IEC 61131-3 applications.

2.11.4.3 Bistable Elements

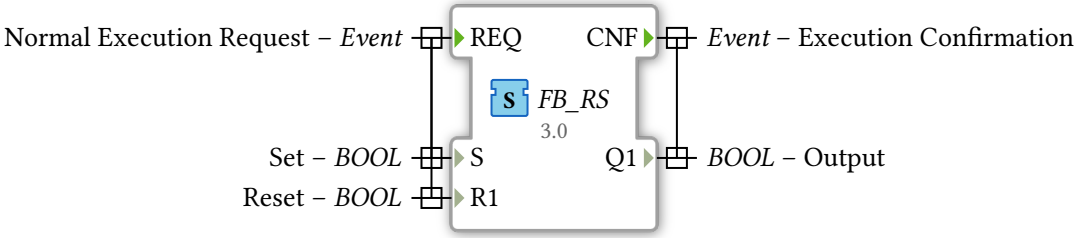
Here you will find functional building blocks for bistable elements (flip-flops).

FB_RS

• – ◦ • – ◦ • – ◦ •

Introduction

The FB_RS function block is a bistable element that functions as a set-reset flip-flop (RS flip-flop). It stores a state and changes it based on the input signals S (Set) and R1 (Reset). The block is implemented according to the IEC 61499-1 standard and is suitable for control applications that require state storage.



Interface Structure

Event Inputs

- **REQ**: Normal Execution Request. Triggers the execution of the function block. Linked to the data inputs S and R1.

Event Outputs

- **CNF**: Execution Confirmation. Outputs after the inputs have been processed. Linked to the data output Q1.

Data Inputs

- **S (BOOL)**: Set input. Sets output Q1 to TRUE if S is TRUE and R1 is FALSE.
- **R1 (BOOL)**: Reset input. Sets output Q1 to FALSE if R1 is TRUE.

Data Outputs

- **Q1 (BOOL)**: Output. Represents the stored state of the RS flip-flop.

Adapters

No adapters available.

Functionality

The FB_RS processes inputs S and R1 on every REQ event. Output Q1 is updated according to the following logic:

- If R1 is TRUE, Q1 is set to FALSE (Reset takes priority).
- If S is TRUE and R1 is FALSE, Q1 is set to TRUE.
- If both inputs are FALSE, Q1 retains its previous state.

The logic is implemented using the following algorithm:
ALGORITHM REQ Q1 := (NOT R1) AND (S OR Q1); END_ALGORITHM

Technical Features

- **IEC 61131-3 Compliance**: This component precisely replicates the behavior of the classic `RS` flip-flop. Since the inputs `S` and `R1` are data inputs that are sampled simultaneously during the `REQ` event, logical prioritization is necessary.
- **Reset Dominance (Priority)**: The reset input (`R1`) takes precedence. If `S` and `R1` are simultaneously `TRUE`, the output `Q1` is set to `FALSE`.
- **Storage Behavior**: The block retains its state as long as there is no change to `S` or `R1` (or no `REQ` event occurs).

Storage Behavior: The block retains its state as long as there is no change to `S` or `R1` (or no `REQ` event occurs).

State Overview

The `FB_RS` evaluates the inputs at `REQ`:

1. **Q1 = FALSE**: If `R1` is TRUE (regardless of `S`). -> **Reset Dominance**
2. **Q1 = TRUE**: If `S` is TRUE AND `R1` is FALSE.
3. **Q1 Unchanged**: If both are FALSE.

Application Scenarios

- Storage of switching states in control applications according to IEC 61131-3 logic.
- Implementation of interlock logic (e.g., emergency stop circuits) where switching off (reset) must always take precedence.

⚖️ Comparison with similar building blocks

- **E_RS**: The `E_RS` is purely event-driven (“Last Event Wins”). The `FB_RS`, on the other hand, evaluates static signals at time `REQ` and enforces reset dominance.
- **FB_SR**: The counterpart with **set dominance**.

🛠️ Related Exercises

- Exercise_006e2
- Exercise_006e2_AX
- Exercise_160b2
- Exercise_160b2_AX

Conclusion

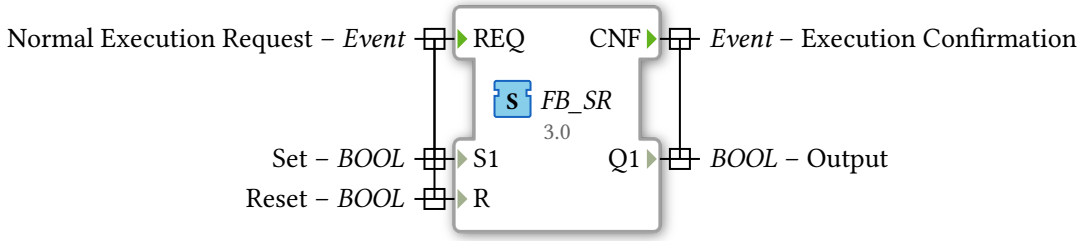
The FB_RS is a fundamental and robust function block for state storage in IEC 61499-based control systems. Its simple and clear logic makes it particularly suitable for applications requiring reliable set-reset behavior. Prioritizing the reset input ensures that safety functions can be executed reliably.

FB_SR

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_SR` is a bistable element that functions as a set-reset flip-flop (SR flip-flop). It stores a state based on the input signals and retains this state until it is changed by new input signals.



Interface Structure

Event Inputs

- **REQ**: Normal Execution Request (trigger for executing the function block)

Event Outputs

- **CNF**: Execution Acknowledgement (triggered after processing the inputs)

Data Inputs

- **S1** (BOOL): Set Input (sets output `Q1` to `TRUE` if `TRUE`)
- **R** (BOOL): Reset Input (sets output `Q1` to `FALSE` if `TRUE`)

Data Outputs

- **Q1** (BOOL): Output (stored state of the flip-flop)

Adapter

No adapters available.

Functionality

The function block `FB_SR` processes the inputs `S1` (Set) and `R` (Reset) according to the following logic:

- If `S1` is `TRUE`, `Q1` is set to `TRUE`.
- If `R` is `TRUE`, `Q1` is set to `FALSE`.
- If both inputs are `FALSE`, `Q1` retains its previous state.

The logic is implemented using the following algorithm:
ALGORITHM REQ Q1 := S1 OR ((NOT R) AND Q1); END_ALGORITHM

Technical Features

- **IEC 61131-3 Compliance**: This function block precisely replicates the behavior of the classic `SR` flip-flop. Since the inputs `S1` and `R` are data inputs that are sampled simultaneously during the `REQ` event, logical prioritization is necessary.
- **Set Dominance**: The function block implements **set priority**. If `S1` and `R` are simultaneously `TRUE`, the output `Q1` is set to `TRUE`.
- **Trigger**: Execution (logic calculation) occurs only on the event `REQ`.

State Overview

The state of the flip-flop (`Q1`) depends on the current values of `S1` and `R` (at the time of `REQ`):

- `S1 = TRUE`: `Q1` is set to `TRUE` (independently of `R`). -> **Set Dominance**
- `R = TRUE` AND `S1 = FALSE`: `Q1` is set to `FALSE`.
- `S1 = FALSE` and `R = FALSE`: `Q1` remains unchanged.

Application Scenarios

- Storage of binary states in control applications where IEC 61131-3 behavior is required.
- Use in switching networks where the dominance of the set signal is necessary for safety or process reasons.

⚖️ Comparison with similar building blocks

- **E_SR**: The `E_SR` is purely event-driven. There are no simultaneous signals and therefore no dominance in this sense (the last event wins). The `FB_SR`, on the other hand, evaluates static signals at time `REQ` and enforces set dominance.
- **FB_RS**: The counterpart with **reset dominance** (reset priority).

🛠️ Related Exercises

- Exercise_006e1
- Exercise_006e1_AX

Conclusion

The `FB_SR` is a basic and efficient function block for storing binary states in control applications. Its simple logic and clear interface make it ideal for use in various industrial automation solutions.

2.11.4.4 Bitwise Operators

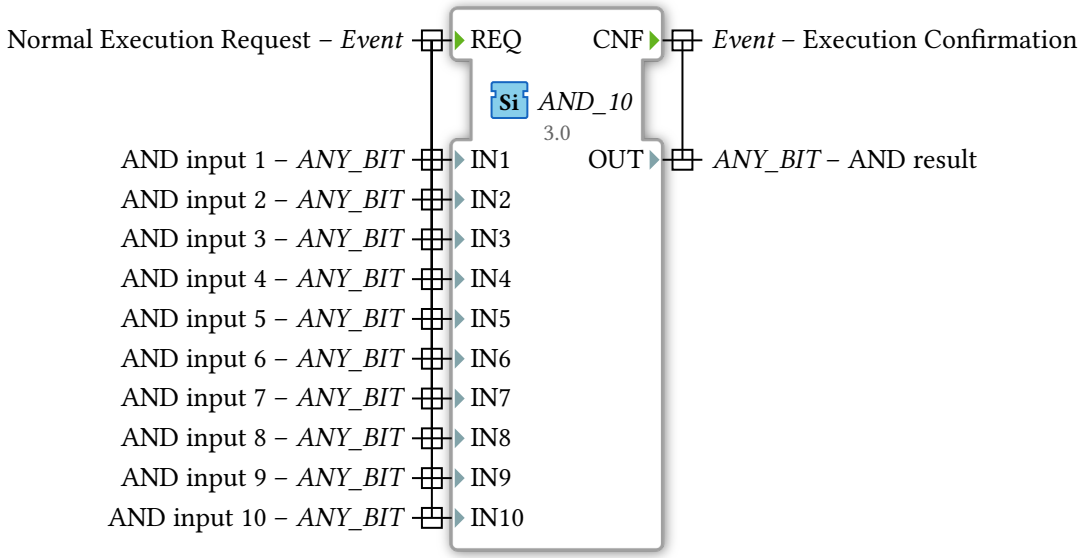
Here you will find functions for bitwise logic operations (AND, OR, XOR, shift, rotate).

AND_10

• – ◦ • – ◦ • – ◦ •

Introduction

The `AND_10` function block is a generic building block for calculating a bitwise logical AND operation. It supports up to 10 input variables and is classified according to the IEC 61131-3 standard. This function block is part of the `iec61131::bitwiseOperators` package and is suitable for applications requiring logical operations at the bit level.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. It is linked to all inputs (`IN1` to `IN10`).

Event Outputs

- **CNF**: Signals the completion of the calculation. Linked to output `OUT`.

Data Inputs

- **IN1** to **IN10** (`ANY_BIT`): Up to 10 input variables for bitwise AND operation. Each input can be any bit data type (e.g., BOOL, BYTE, WORD, DWORD).

Data Outputs

- **OUT** (`ANY_BIT`): Result of the bitwise AND operation of all inputs. The data type corresponds to that of the inputs.

Adapters

- No adapters available.

Functionality

The `AND_10` function block performs a bitwise logical AND operation on the values of all active inputs (`IN1` to `IN10`). The result is output at `OUT` as soon as the `REQ` event is received. The calculation is confirmed by the `CNF` event.

Technical Features

- **Generic Implementation**: The function block uses the generic class `GEN_AND`, which allows for flexible use with various bit data types.
- **Scalability**: Supports up to 10 inputs, enabling more complex logical operations.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Calculation State**: Executes the AND operation as soon as `REQ` is activated.
3. **Output State**: Outputs the result via `OUT` and signals completion with `CNF`.

Application Scenarios

- Logical filtering of signal groups.
- Bitwise masking of data values.
- Control logic in industrial automation systems.

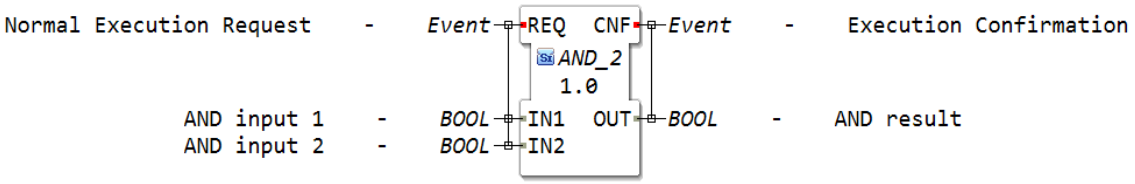
⚖️ Comparison with Similar Function Blocks

- **Standard AND Function Block**: Typically supports only 2 inputs. `AND_10` offers greater flexibility with 10 inputs.
- **Generic Logic Blocks**: Similarly generic blocks could support fewer inputs or more specific data types.

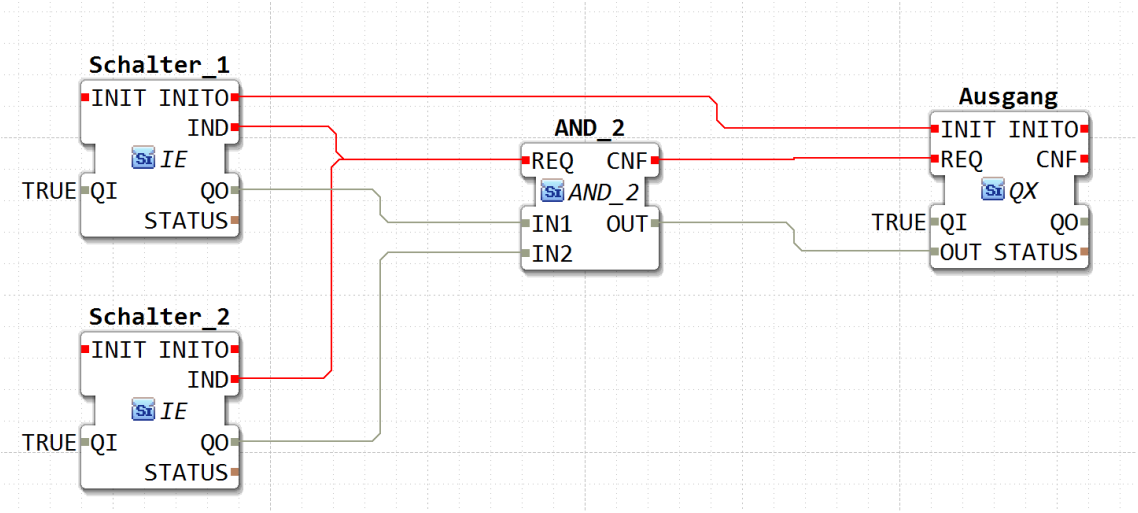
Conclusion

The `AND_10` function block is a versatile tool for bitwise logic operations in IEC 61131-3-based control systems. Its generic nature and support for up to 10 inputs make it particularly useful for complex logic tasks.

AND_2



OUT is only switched when a Boolean signal is present at IN1 and IN2, e.g., from a switch.

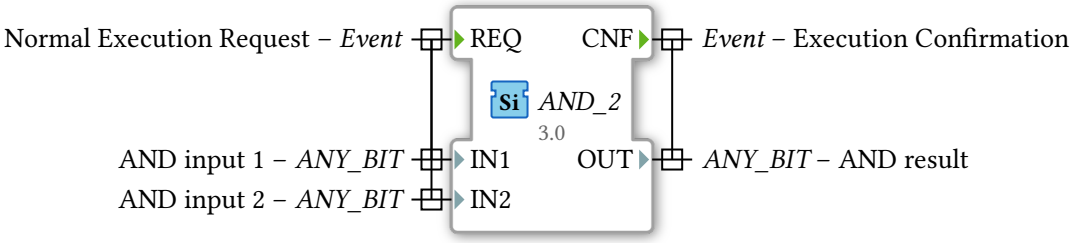


The same principle applies to all other “AND” blocks, except that all IN inputs must always be True.

• – ◦ • – ◦ • – ◦ •

Introduction

The AND_2 function block is a generic block for calculating the bitwise Boolean AND operation. It implements a standard function according to IEC 61131-3 and can be used with any bit data type.



Interface Structure

Event Inputs

- **REQ**: Normal execution request - triggers the calculation of the AND operation

Event Outputs

- **CNF**: Execution confirmation - signals the completion of the calculation

Data Inputs

- **IN1** (ANY_BIT): AND input 1 - first operand for the bitwise AND operation
- **IN2** (ANY_BIT): AND input 2 - second operand for the bitwise AND operation

Data Outputs

- **OUT** (ANY_BIT): AND result - result of the bitwise AND operation

Adapters

No adapter interfaces are available.

Functionality

Upon arrival of the REQ event, the function block performs a bitwise AND operation between the two input values IN1 and IN2. The result is output at the data output OUT, and simultaneously, the CNF event is generated to confirm execution.

The operation is performed separately for each bit of the input values:

- Bit = 1 if both corresponding bits of the inputs are 1
- Bit = 0 if at least one of the corresponding bits of the inputs is 0

Technical Features

- **Generic Data Type**: Uses ANY_BIT, which allows its use with various bit data types (BOOL, BYTE, WORD, DWORD, LWORD)
- **Bitwise Operation**: Performs the AND operation at the bit level
- **Event-Driven**: Execution is driven by events, not by cycle times

State Transitions

1. **Wait State**: Block waits for REQ event
2. **Compute State**: Upon REQ event, IN1 and IN2 are processed
3. **Output State**: Result is output to OUT and a CNF event is generated
4. Returns to Wait State

Application Scenarios

- Bitmask operations in control applications
- Filtering of data by masking unwanted bits
- Logical operations in digital circuits
- Data preprocessing in communication protocols

⚖️ Comparison with similar building blocks

Compared to other logical operation blocks:

- AND_2 is generic and supports various bit data types
- Operates bitwise, unlike purely Boolean AND blocks
- Event-driven, unlike cycle-driven blocks

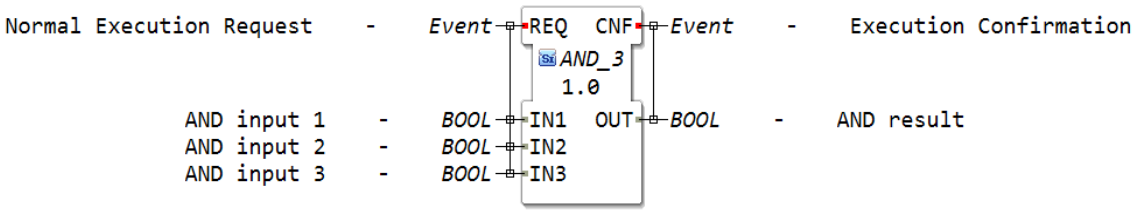
🛠️ Related exercises

- Exercise_002a
- Exercise_002a_AX
- Exercise_002b2
- Exercise_002b3_AX
- Exercise_006a3_AX
- Exercise_088

Conclusion

The AND_2 function block offers a flexible and standards-compliant solution for bitwise AND operations in IEC 61131-3 based control systems. Its generic nature allows for use in a wide range of applications, while its event-driven architecture ensures efficient resource utilization.

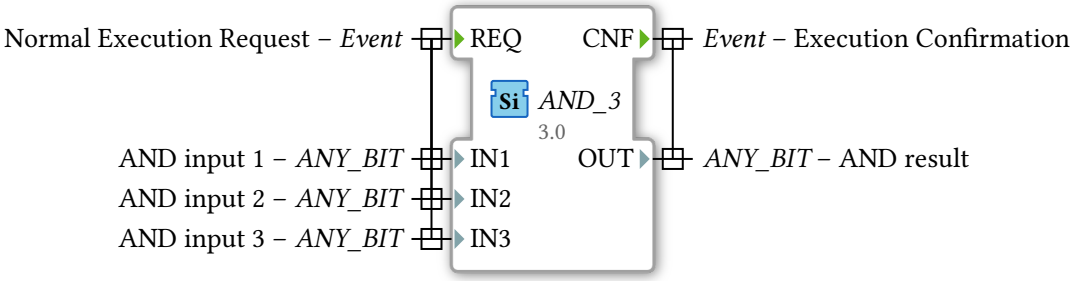
AND_3



• – ○ • – ○ • – ○ •

Introduction

The AND_3 function block is a generic function block for calculating the bitwise logical AND operation with three inputs. It performs a bitwise AND operation on three input values and outputs the result. The function block supports all ANY_BIT data types and is therefore flexible for use with different bit widths.



Interface Structure

Event Inputs

- **REQ**: Normal execution request - triggers the calculation of the AND operation

Event Outputs

- **CNF**: Execution confirmation - signals the completion of the calculation

Data Inputs

- **IN1**: AND input 1 (ANY_BIT)
- **IN2**: AND input 2 (ANY_BIT)
- **IN3**: AND input 3 (ANY_BIT)

Data Outputs

- **OUT**: AND result (ANY_BIT)

Adapters

No adapters available.

Functionality

Upon arrival of the REQ event, the AND_3 block performs a bitwise AND operation on the three input values IN1, IN2, and IN3. The result is output at the OUT data output, and simultaneously a CNF event is generated to signal successful computation.

The operation follows the formula: OUT = IN1 ∧ IN2 ∧ IN3

Technical Features

- Generic function block that works with all ANY_BIT data types (BOOL, BYTE, WORD, DWORD, LWORD)
- Supports three inputs for extended AND operations
- Type safety through consistent use of the same data type for all inputs and outputs

State Transitions

1. **Wait State**: The function block waits for a REQ event
2. **Calculation State**: Upon REQ, the inputs are processed
3. **Output State**: The result is output to OUT and a CNF event is generated
4. Return to Wait State

Application Scenarios

- Bitmask operations in control applications
- Multiple condition checks in logic circuits
- Signal processing with multiple input signals
- Filter operations in data processing systems

⚖️ Comparison with Similar Function Blocks

Compared to Compared to standard two-input AND gates, AND_3 offers the ability to process three inputs simultaneously, reducing the number of gates required in more complex circuits. AND_3 provides greater flexibility compared to fixed-bit-width gates due to its generic ANY_BIT typing.

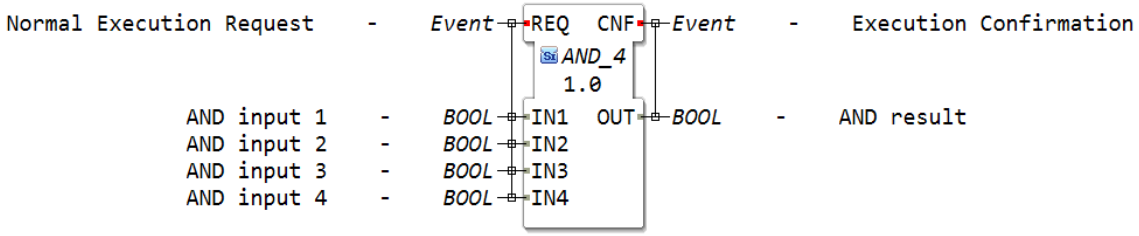
🛠️ Related Exercises

- Exercise_002a6_AX

Conclusion

The AND_3 function block is a versatile and efficient gate for bitwise AND operations with three inputs. Its generic nature makes it suitable for various applications and data types, while its clear event-driven handling enables reliable integration into control systems.

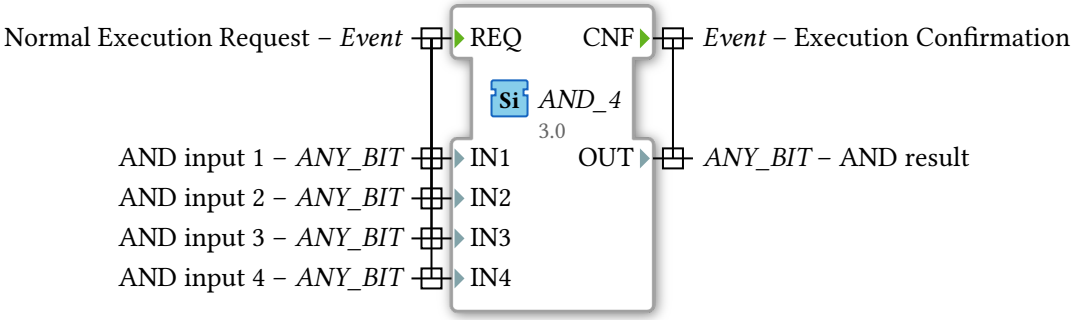
AND_4



• – ◦ • – ◦ • – ◦ •

Introduction

The AND_4 function block is a generic function block for calculating bitwise Boolean AND operations. It processes four input values and performs a logical AND operation at the bit level. The function block is classified according to the IEC 61131-3 standard and supports any bit data type.



Interface Structure

Event Inputs

- **REQ**: Normal execution request - triggers the calculation

Event Outputs

- **CNF**: Execution confirmation - signals the completion of the calculation

Data Inputs

- **IN1**: AND input 1 (ANY_BIT data type)
- **IN2**: AND input 2 (ANY_BIT data type)
- **IN3**: AND input 3 (ANY_BIT data type)
- **IN4**: AND input 4 (ANY_BIT data type)

Data Outputs

- **OUT**: AND result (ANY_BIT data type)

Adapters

No adapter interfaces are available.

Functionality

Upon receiving the REQ event, the AND_4 block performs a bitwise AND operation on all four input values. The result is output at the OUT data output, and simultaneously, the CNF event is triggered to confirm successful calculation. The operation is performed separately for each bit of the input values.

Technical Features

- Supports the generic data type ANY_BIT, enabling use with various bit data types such as BOOL, BYTE, WORD, DWORD, etc.
- Four independent inputs for flexible linking options
- Event-driven execution for deterministic behavior
- Compatible with the IEC 61131-3 standard

State Transitions

1. **Wait State**: Block waits for a REQ event
2. **Compute State**: Upon receiving a REQ, all four inputs are bitwise ANDed
3. **Output State**: Result is output to OUT, and a CNF event is triggered
4. Returns to Wait State

Application Scenarios

- Bitmask operations in control applications
- Parity checks and error detection
- Signal processing in industrial automation systems
- Logical operations of multiple digital signals
- Filter operations in data processing systems

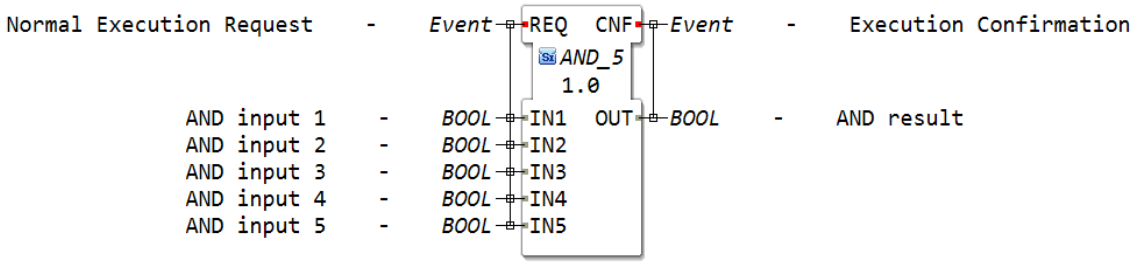
⚖️ Comparison with Similar Blocks

Compared to Compared to standard AND blocks with fewer inputs, AND_4 offers the ability to combine four signals simultaneously, enabling more complex logical operations in a single step. Compared to sequential AND operations using multiple blocks, AND_4 provides a more compact and user-friendly solution.

Conclusion

The AND_4 function block represents an efficient and flexible solution for bitwise AND operations with four inputs. Its generic nature allows for use in various applications, while its event-driven architecture ensures reliable integration into more complex control systems.

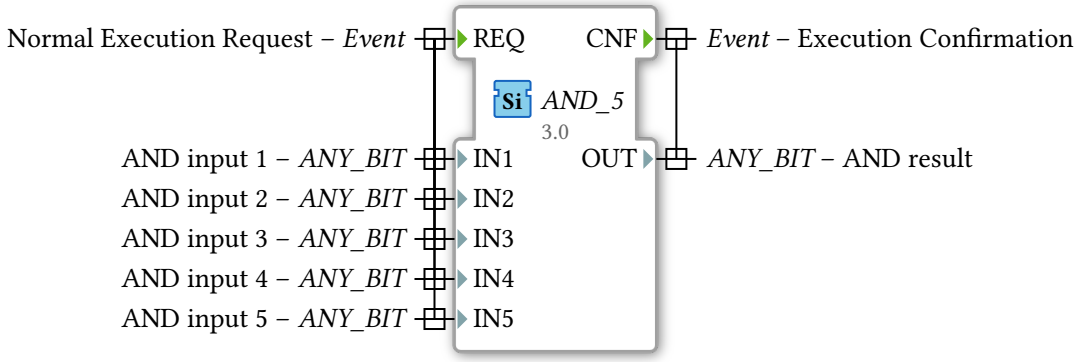
AND_5



• – ◦ • – ◦ • – ◦ •

Introduction

The AND_5 function block is a generic function block for calculating bitwise logical AND operations with five inputs. It implements a standard bitwise operation according to IEC 61131-3 and can work with any bit data type.



Interface Structure

Event Inputs

- **REQ**: Normal execution request - triggers the calculation of the AND operation

Event Outputs

- **CNF**: Execution confirmation - signals the completion of the calculation

Data Inputs

- **IN1**: AND input 1 (ANY_BIT data type)
- **IN2**: AND input 2 (ANY_BIT data type)
- **IN3**: AND input 3 (ANY_BIT data type)
- **IN4**: AND input 4 (ANY_BIT data type)
- **IN5**: AND input 5 (ANY_BIT data type)

Data Outputs

- **OUT**: AND result (ANY_BIT data type)

Adapters

No adapter interfaces are available.

Functionality

Upon arrival of the REQ event, the function block performs a bitwise AND operation on all five inputs. The result is output at the OUT data output, and simultaneously, the CNF event is sent to confirm the completion of the calculation.

The operation is performed separately for each bit position:

$OUT_i = IN1_i \wedge IN2_i \wedge IN3_i \wedge IN4_i \wedge IN5_i$

Technical Features

- **Generic Data Type**: Uses ANY_BIT, making the block compatible with various bit data types (BOOL, BYTE, WORD, DWORD, LWORD)
- **Five Inputs**: Allows the combination of up to five different bit values
- **Event-Driven**: The calculation is only performed when requested by the REQ event

State Transitions

1. **Wait State**: Block waits for REQ event
2. **Calculation State**: Upon REQ input, all inputs are combined bitwise using an AND operation
3. **Output State**: Result is output to OUT and a CNF event is sent
4. Returns to Wait State

Application Scenarios

- Bitmask operations in control applications
- Multiple condition checks in safety systems
- Signal processing with multiple input signals
- Filter operations in data processing systems

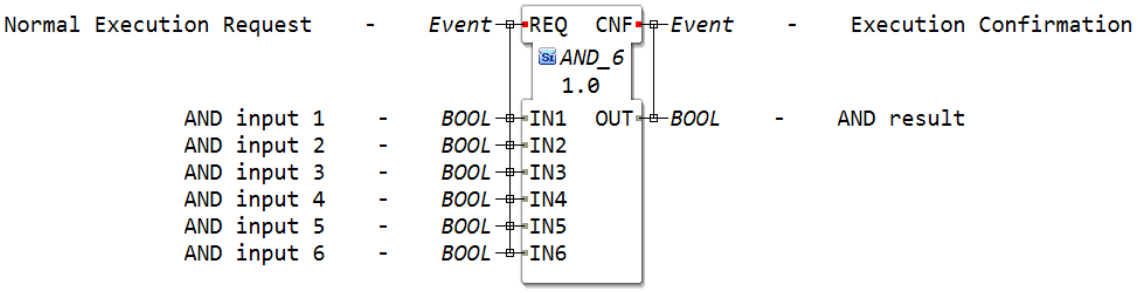
⚖️ Comparison with similar function blocks

- **AND_2**: Provides only two inputs for AND operation
- **OR_5**: Performs bitwise OR operations with five inputs
- **XOR_5**: Performs bitwise exclusive OR operations

Conclusion

The AND_5 function block offers a flexible and efficient solution for bitwise AND operations with five inputs. Its generic nature allows its use with various bit data types, making it a versatile building block for complex logic operations in automation applications.

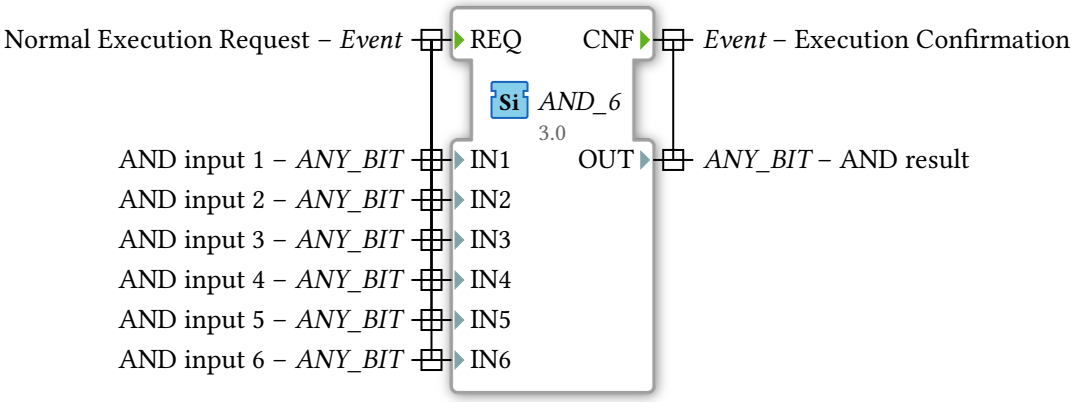
AND_6



• – ◦ • – ◦ • – ◦ •

Introduction

The AND_6 function block is a generic function block for calculating the bitwise Boolean AND operation with six inputs. It implements a standard function according to IEC 61131-3 and performs a logical AND operation at the bit level for up to six input values.



Interface Structure

Event Inputs

- **REQ** (Normal Execution Request): Starts the execution of the AND operation. Triggers the processing of the input data.

Event Outputs

- **CNF** (Execution Confirmation): Signals the completion of the calculation and the availability of the result.

Data Inputs

- **IN1** (ANY_BIT): AND Input 1
- **IN2** (ANY_BIT): AND Input 2
- **IN3** (ANY_BIT): AND Input 3
- **IN4** (ANY_BIT): AND Input 4
- **IN5** (ANY_BIT): AND Input 5

- **IN6** (ANY_BIT): AND Input 6

Data Outputs

- **OUT** (ANY_BIT): AND result of the bitwise operation of all six inputs

Adapter

No adapter interfaces available.

Functionality

When the REQ event is triggered, the AND_6 block performs a bitwise AND operation on all six input values. The operation is performed separately for each bit position: The result bit at position i is only 1 if all six input bits at position i have the value 1. Once the calculation is complete, the CNF event is triggered and the result is made available at the OUT output. ...

Technical Features

- Uses the generic data type ANY_BIT, which supports various bit data types (BOOL, BYTE, WORD, DWORD, LWORD)
- All inputs must have the same data type
- The result has the same data type as the inputs
- Implemented as a generic function block with the class 'GEN_AND'

State Transitions

1. **Wait State**: Block waits for a REQ event
2. **Compute State**: On REQ, all inputs are performed with a bitwise AND operation
3. **Output State**: Result is output to OUT and a CNF event is triggered
4. Returns to Wait State

Application Scenarios

- Bitmask operations in control applications
- Multiple condition checks in logic circuits
- Signal processing with multiple input signals
- Safety-related circuits with multiple monitoring signals

⚖️ Comparison with Similar Blocks

Compared to simpler AND blocks With fewer inputs, AND_6 allows you to combine up to six signals simultaneously, simplifying the program structure and requiring fewer logic blocks. Compared to serial AND operations, it offers improved clarity and lower latency.

Conclusion

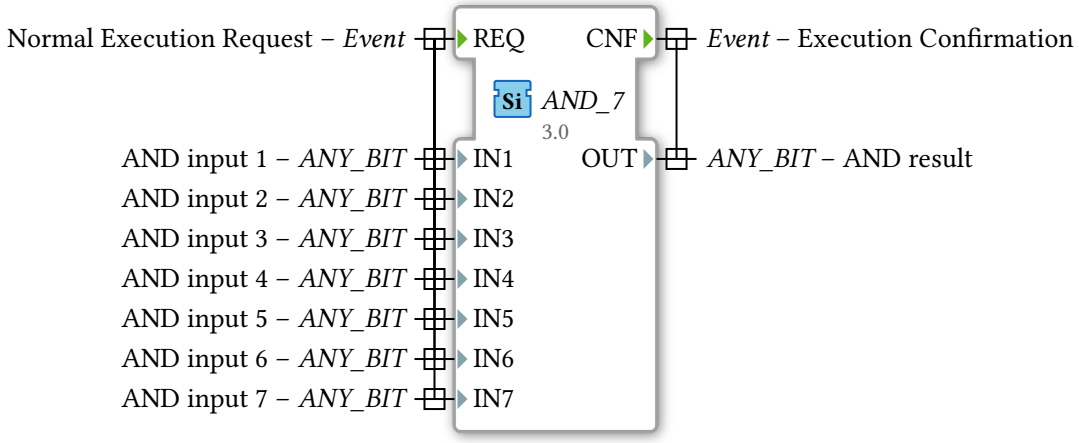
The AND_6 function block provides an efficient and clear solution for complex bitwise AND operations with up to six inputs. Its generic implementation enables its use with various bit data types, making it a versatile tool in automation technology.

AND_7

• – ◦ • – ◦ • – ◦ •

Introduction

The AND_7 function block performs a bitwise logical AND operation on up to seven input variables. It is a generic function block that can work with various bit data types (ANY_BIT). The block is classified according to the IEC 61131-3 standard and is suitable for use in control and automation systems.



Interface Structure

Event Inputs

- **REQ** (Normal Execution Request): Starts the execution of the function block. It is used with all seven input variables (IN1 to IN7).

Event Outputs

- **CNF** (Execution Confirmation): Signals successful execution and returns the result of the AND operation via the variable OUT.

Data Inputs

- **IN1** (ANY_BIT): First input for the AND operation
- **IN2** (ANY_BIT): Second input for the AND operation
- **IN3** (ANY_BIT): Third input for the AND operation
- **IN4** (ANY_BIT): Fourth input for the AND operation
- **IN5** (ANY_BIT): Fifth input for the AND operation
- **IN6** (ANY_BIT): Sixth input for the AND operation
- **IN7** (ANY_BIT): Seventh input for the AND operation

Data Outputs

- **OUT** (ANY_BIT): Result of the bitwise AND operation of all input values

Adapters

This function block does not use adapters.

Functionality

The AND_7 block performs a bitwise AND operation on all active inputs (IN1 to IN7) on every REQ event. The result is output to OUT, and the CNF event is triggered. The block is generic and can work with various bit data types (such as BOOL, BYTE, WORD, DWORD, LWORD).

Technical Features

- Supports the generic data type ANY_BIT
- Implemented as a generic function block (GEN_AND)
- Part of the iec61131::bitwiseOperators package
- EPL-2.0 license

State Overview

The block has no internal state. On every REQ event, the operation is executed immediately, and the result is returned via CNF.

Application Scenarios

- Bitwise mask operations in control applications
- Logical combination of multiple digital signals

- Signal processing in automation systems
- Generic logic operations in IEC 61131-3 compliant systems

⚖️ Comparison with similar blocks

- Standard AND blocks typically offer only 2-4 inputs
- AND_7 enables the processing of up to seven inputs in a single operation
- More flexible than type-specific blocks due to its generic implementation

Conclusion

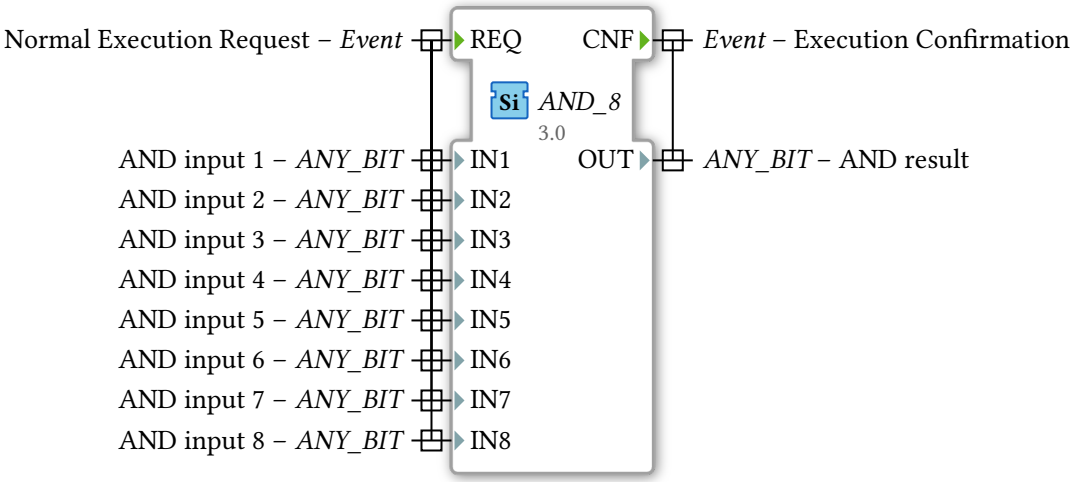
The AND_7 function block offers enhanced AND logic functionality for systems that require the processing of multiple input signals in a single operation. Its generic nature and IEC 61131-3 compatibility make it a versatile tool for automation applications.

AND_8

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_8` performs a bitwise logical AND operation on up to eight inputs. It is a generic function block that can work with various bit data types. The block is classified according to the IEC 61131-3 standard and is used for processing Boolean operations.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. The operation is executed when this event occurs. It is linked to all data inputs (`IN1` to `IN8`).

Event Outputs

- **CNF**: Confirms the execution of the function block. This event is output after the calculation and is linked to the data output `OUT`.

Data Inputs

- **IN1** to **IN8** (`ANY_BIT`): Eight inputs for bitwise AND operations. Each input can accept any bit data type (e.g., BOOL, BYTE, WORD, DWORD, LWORD).

Data Outputs

- **OUT** (`ANY_BIT`): The result of the bitwise AND operation on all inputs. The data type corresponds to that of the inputs.

Adapters

The function block does not have any adapters.

Adapters ## Functionality

The `AND_8` block performs a bitwise AND operation on the values of all active inputs (`IN1` to `IN8`). The result is output at `OUT`. The operation is initiated by the event `REQ` and acknowledged by `CNF`.

Technical Features

- **Generic Implementation**: The block can work with different bit data types because the inputs and outputs are of type `ANY_BIT`.
- **Flexible Number of Inputs**: Up to eight inputs can be used, and not all of them need to be configured.

State Overview

1. **Idle State**: The block waits for the event `REQ`.
2. **Execution State**: At `REQ`, the inputs are processed and the result is calculated.
3. **Acknowledgement State**: After the calculation, `CNF` is triggered and the result is output to `OUT`.

Application Scenarios

- Logical combination of multiple bit signals in control applications.
- Filtering of signals by bitwise masking.
- Generic processing of bit data in complex automation systems.

⚖️ Comparison with Similar Blocks

- **AND (Standard)**: Standard AND block with typically two inputs. `AND_8` offers greater flexibility through eight inputs and generic types.
- **GEN_AND**: Generic AND implementation that serves as the basis for `AND_8`. `AND_8` is a specialized version with a fixed number of inputs.

Conclusion

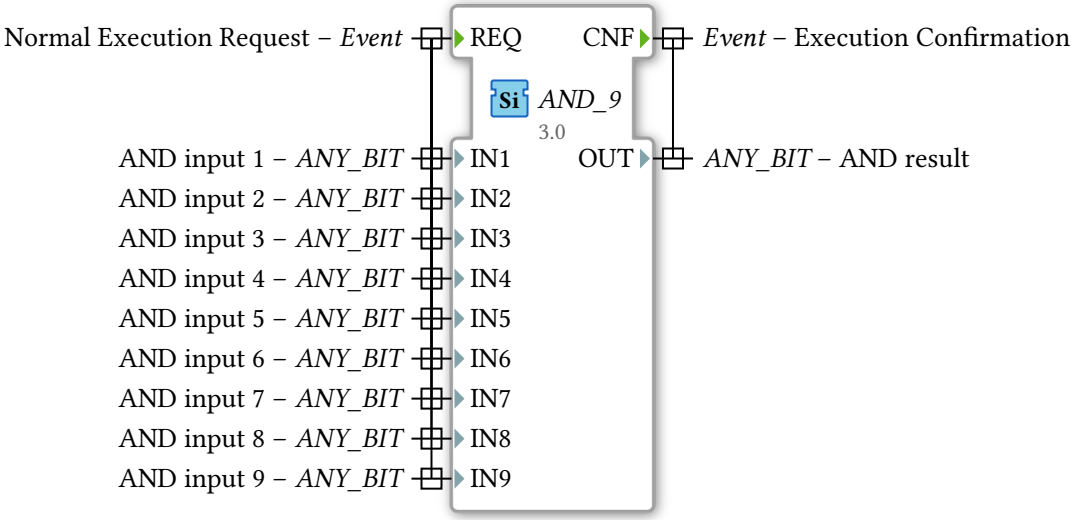
The `AND_8` function block is a powerful tool for bitwise logic operations in IEC 61131-3 compliant control systems. Its generic nature and flexible number of inputs make it particularly versatile.

AND_9

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `AND_9` performs a bitwise logical AND operation on up to nine input variables. It is a generic function block that can work with various bit data types. The block is classified according to the IEC 61131-3 standard and is suitable for automation applications.



Interface Structure

Event Inputs

- `REQ` (Normal Execution Request): Starts the execution of the function block. The operation is executed when this event occurs.

Event Outputs

- `CNF` (Execution Confirmation): Signals successful completion of the operation.

Data Inputs

- `IN1` to `IN9` (ANY_BIT): Up to nine input variables on which the bitwise AND operation is performed. Each input can have any bit data type (e.g., BOOL, BYTE, WORD, DWORD, LWORD).

Data Outputs

- `OUT` (ANY_BIT): The result of the bitwise AND operation on all input variables. The data type corresponds to that of the input variables.

Adapters

No adapters are available.

Functionality

The function block `AND_9` performs a bitwise AND operation on the values of the input variables `IN1` through `IN9`. The result is output to `OUT`. The operation is initiated by the event `REQ` and confirmed upon completion by the event `CNF`.

Technical Features

- **Generic Implementation**: The function block can work with various bit data types, enabling high flexibility.
- **Scalability**: Supports up to nine input variables, allowing for more complex logical operations.

State Overview

1. **Idle State**: Waits for the event `REQ`.
2. **Execution State**: Performs the bitwise AND operation.
3. **Acknowledge State**: Sends the event `CNF` and outputs the result to `OUT`.

Application Scenarios

- Logical combination of multiple digital signals in automation technology.
- Filtering of signals using logical masks.
- Implementation of safety functions where multiple conditions must be met simultaneously.

⚖️ Comparison with Similar Function Blocks

- **AND (Standard)**: By default, many PLC systems only support two-input AND operations. `AND_9` extends this functionality to up to nine inputs.
- **GEN_AND**: The generic AND block on which `AND_9` is based can be configured for various data types and a variable number of inputs.

Conclusion

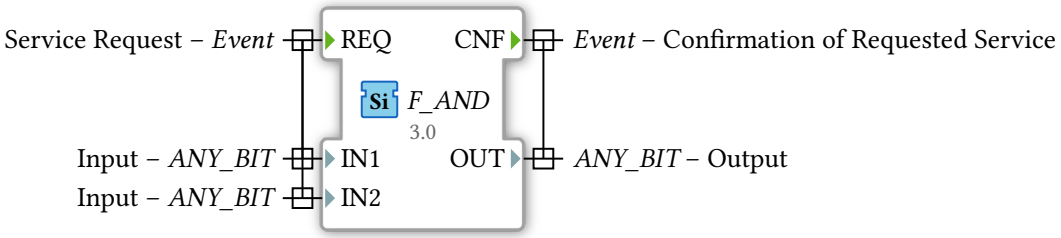
The `AND_9` function block is a powerful tool for bitwise logic operations in automation technology. Its generic nature and support for up to nine inputs make it particularly flexible and versatile. Ideal for applications requiring complex logical operations.

F_AND

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_AND` is a standardized bitwise Boolean AND operator according to IEC 61131-3. It performs a bitwise AND operation on two input variables and outputs the result. This block is particularly useful for logical operations in control applications.



Interface Structure

Event Inputs

- **REQ**: Service Request. Triggers the execution of the AND operation. Linked to the data inputs `IN1` and `IN2`.

Event Outputs

- **CNF**: Confirmation of the requested service. Triggered after successful execution of the operation. Linked to the data output `OUT`.

Data Inputs

- **IN1**: First input for the bitwise AND operation. Type: `ANY_BIT`.
- **IN2**: Second input for the bitwise AND operation. Type: `ANY_BIT`.

Data Outputs

- **OUT**: Result of the bitwise AND operation. Type: `ANY_BIT`.

Adapters

- No adapters available.

Functionality

The `F_AND` block performs a bitwise AND operation between the input variables `IN1` and `IN2` when the event `REQ` is triggered. The result is passed to the output `OUT`, and the event `CNF` is triggered to confirm successful execution.

Technical Features

- Supports the data type `ANY_BIT`, which means that various bit types such as `BOOL`, `BYTE`, `WORD`, `DWORD`, etc., can be used.
- The block conforms to the IEC 61131-3 standard and is classified as a “standard bitwise boolean function”.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Execution State**: Executes the bitwise AND operation when `REQ` is received.
3. **Acknowledgement State**: Sends the `CNF` event and outputs the result to `OUT`.

Application Scenarios

- Logical controls in industrial automation systems.
- Bitwise mask operations in data processing applications.
- Combination with other logic blocks for complex control logic.

⚖️ Comparison with Similar Blocks

- **F_OR**: Performs a bitwise OR operation.
- **F_XOR**: Performs a bitwise exclusive OR operation.
- **F_NOT**: Performs a bitwise negation.

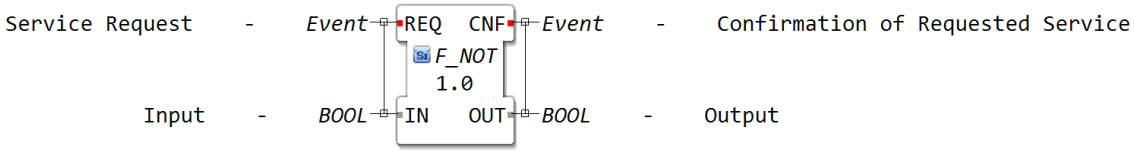
🛠️ Related Exercises

- Exercise_002a2
- Exercise_002a2_AX

Conclusion

The F_AND function block is a fundamental building block for bitwise logic operations in IEC 61131-3 compliant control systems. Its simple and standardized interface makes it reliable and versatile for use in various automation applications.

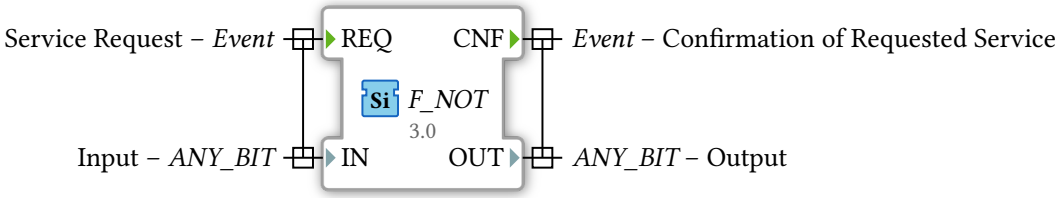
F_NOT



• – ◦ • – ◦ • – ◦ •

Introduction

The F_NOT function block implements a bitwise logical NOT operation according to the IEC 61131-3 standard. It performs a bitwise inversion of the input value and outputs the result.



Interface Structure

Event Inputs

- **REQ**: Service Request - Triggers the calculation of the NOT operation

Event Outputs

- **CNF**: Confirmation of Requested Service - Signals the completion of the calculation

Data Inputs

- **IN**: Input value of type ANY_BIT - The value to be inverted

Data Outputs

- **OUT**: Output value of type ANY_BIT - The inverted result

Adapters

No adapters available

Functionality

Upon receiving a REQ event, the F_NOT block performs a bitwise logical NOT operation on the input value IN. Each bit of the input value is inverted (0 becomes 1, 1 becomes 0). The result is output at output OUT and confirmed by the CNF event.

Technical Features

- Supports the ANY_BIT data type, enabling use with various bit data types (BOOL, BYTE, WORD, DWORD, etc.)
- Real-time capable execution
- Standard-compliant implementation according to IEC 61131-3

State Transition

1. Wait state for REQ event
2. On REQ: Execute the NOT operation
3. Output the result to OUT
4. Send the CNF event

5. Return to wait state

Application Scenarios

- Bit manipulation in control programs
- Inversion of status bits
- Logical negation of conditions
- Signal inversion in digital circuits

⚖️ Comparison with Similar Function Blocks

Compared to other logical operations such as F_AND or F_OR, F_NOT performs only a unary operation (one operand), while other logical function blocks typically implement binary operations (two operands).

🛠️ Related Exercises

- Exercise_006a3_sub

Conclusion

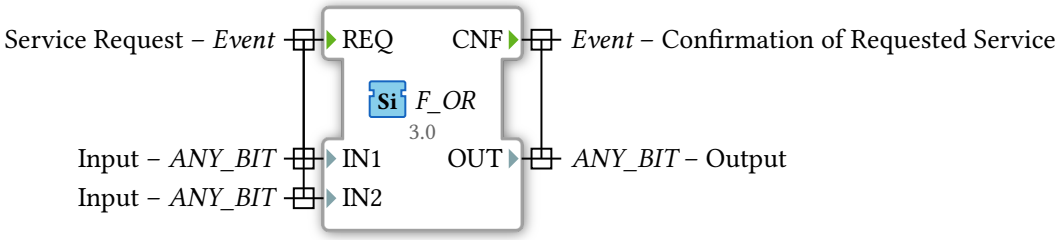
The F_NOT function block offers a standards-compliant, efficient solution for bitwise inversion operations in IEC 61131-3 compliant control systems. Its simple interface and universal ANY_BIT type support make it a versatile building block for various applications.

F_OR

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_OR` implements a bitwise OR operation according to the IEC 61131-3 standard. It performs a logical OR operation on two input variables of type `ANY_BIT` and outputs the result.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the OR operation. Upon this event, the inputs `IN1` and `IN2` are evaluated.

Event Outputs

- **CNF**: Signals the successful completion of the operation. This event is output along with the result `OUT`.

Data Inputs

- **IN1**: First input for the bitwise OR operation (Type: `ANY_BIT`).
- **IN2**: Second input for the bitwise OR operation (Type: `ANY_BIT`).

Data Outputs

- **OUT**: Result of the bitwise OR operation (Type: `ANY_BIT`).

Adapters

This function block does not use any adapters.

Functionality

When the `REQ` event is triggered, the `F_OR` block performs a bitwise OR operation on the values at `IN1` and `IN2`. The result is output to `OUT`, and the `CNF` event signals successful computation.

Technical Features

- **Type Flexibility**: Supports all `ANY_BIT` data types (e.g., BOOL, BYTE, WORD, DWORD, LWORD).
- **Simple Implementation**: No state machine is required, as this is a static operation.

State Overview

The function block has no internal state machine, as the operation is performed directly upon the occurrence of the `REQ` event.

Application Scenarios

- Bitwise operation in control algorithms
- Mask operations in signal processing
- Logical combination of digital signals

⚖️ Comparison with similar function blocks

- **F_AND**: Performs a bitwise AND operation
- **F_XOR**: Performs a bitwise exclusive OR operation
- **F_NOT**: Performs a bitwise negation

Conclusion

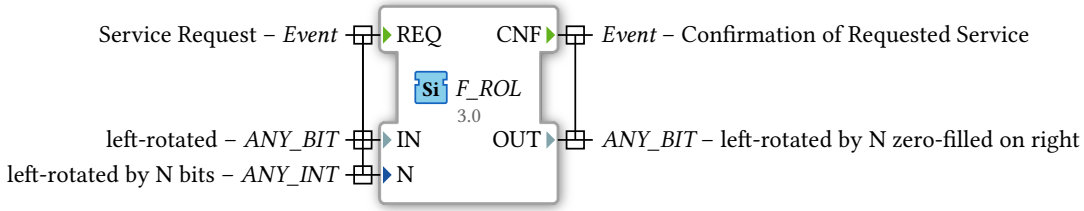
The `F_OR` function block offers a simple and efficient way to perform bitwise OR operations in IEC 61131-3 compliant control systems. Its type flexibility and direct operation make it a universally applicable function block for logical operations.

F_ROL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ROL` (Rotate Left) performs a bitwise left rotation. It rotates the input value a specified number of bits to the left. The bits rotated out on the left side are inserted again on the right side. This is a cyclic operation in which no bits are lost.



Interface Structure

Event Inputs

- **REQ**: Service request that triggers the execution of the left rotation. It is linked to the data inputs `IN` and `N`.

Event Outputs

- **CNF**: Confirmation of the completed service. Linked to the data output `OUT`.

Data Inputs

- **IN**: The input value to be rotated. The data type is `ANY_BIT` (any bit type).
- **N**: The number of bits by which the input value should be rotated. The data type is `ANY_INT` (any integer type).

Data Outputs

- **OUT**: The result of the left rotation. The data type is `ANY_BIT` (the same as the input value).

Adapters

- No adapters available.

Functionality

The function block `F_ROL` performs a bitwise left rotation of the input value `IN` by `N` bits. Bits that fall outside the range on the left side are reinserted on the right side. This is a true rotation; no bits are padded with zeros.

Technical Features

- Supports any bit type (`ANY_BIT`) for the input value `IN` and the output value `OUT`.
- The number of rotations `N` can be any integer value (`ANY_INT`).
- The operation is cyclical, meaning that if `N` is greater than the bit length of `IN`, the rotation is performed modulo the bit length.
- Unlike a shift, no bits are lost during rotation.

State Overview

1. **Initial State**: Waits for the event `REQ`.
2. **Execution State**: Upon receiving `REQ`, the left rotation is performed and the result is output via `OUT`. The event `CNF` is triggered.
3. **Return to Initial State**: After `CNF` is triggered, the block returns to its initial state.

Application Scenarios

- Bitwise manipulation of data in control applications.
- Cryptographic operations requiring bitwise rotations.
- Signal processing where data bits need to be shifted cyclically.
- Checksum calculations and data encoding.

⚖️ Comparison with Similar Components

- **F_SHL (Shift Left)**: Shifts bits to the left and pads the resulting free positions on the right with zeros. This is not a cyclic operation.
- **F_ROR (Rotate Right)**: Performs a right rotation, rotating bits from right to left.
- **F_ROL vs. F_SHL**: While F_SHL performs a shift with zero padding, F_ROL performs a cyclic rotation without zero padding.

Conclusion

The function block `F_ROL` is an efficient tool for bitwise left rotations in control applications. Its flexibility in supporting various bit and integer types makes it versatile. The cyclic nature of the rotation ensures that no data bits are lost, making it particularly suitable for applications with strict data integrity requirements. The clear distinction between rotation (cyclic) and shift (with zero padding) is crucial for correct application.

–

Changes at a Glance:

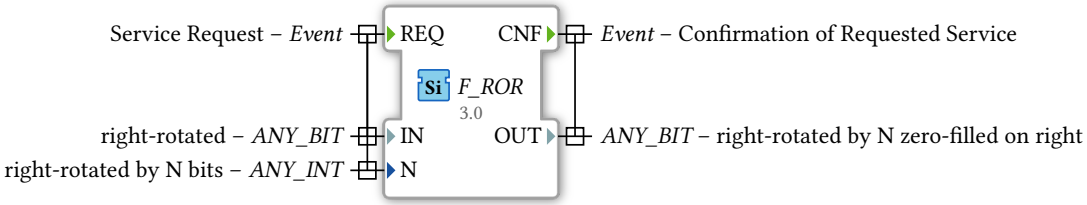
- Correction of the description: With ROL, the rotated-out bits are reinserted on the right, not padded with zeros.
- Clarification that this is a true cyclic rotation.
- Improved distinction between rotation (ROL/ROR) and shift (SHL/SHR).
- Consistent wording throughout the document.

F_ROR

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ROR` (Rotate Right) performs a bitwise right rotation. It belongs to the category of standardized bit shift functions according to IEC 61131-3. The block rotates the input data a specified number of bits to the right and fills the freed bits with the bits rotated out.



Interface Structure

Event Inputs

- `REQ`: Service request that triggers the operation. Linked to the data inputs `IN` and `N`.

Event Outputs

- `CNF`: Confirmation of the operation performed. Linked to the data output `OUT`.

Data Inputs

- `IN`: The input value to be rotated (Type: `ANY_BIT`).
- `N`: The number of bits by which the input value should be rotated right (Type: `ANY_INT`).

Data Outputs

- `OUT`: The result of the right rotation (Type: `ANY_BIT`).

Adapters

No adapters available.

Functionality

The function block performs a bitwise right rotation of the input value `IN` by `N` bits. The bits rotated out are then reinserted on the left side. This differs from a simple right shift, where the freed bits are padded with zeros.

Technical Features

- Supports all `ANY_BIT` data types for the input value `IN` and the output value `OUT`.
- The parameter `N` can accept any `ANY_INT` data type.
- The operation is independent of the word length of the input value.

State Overview

The function block has no internal state. The operation is re-executed with each `REQ` event.

Application Scenarios

- Cryptographic operations requiring bitwise rotations.
- Signal processing requiring bitwise manipulation of data.
- Low-level hardware controllers requiring bitwise operations.

⚖️ Comparison with Similar Components

- `F_SHR` (Shift Right): Performs a right shift, but pads the freed bits with zeros.
- `F_ROL` (Rotate Left): Performs a left rotation, similar to `F_ROR`, but in the opposite direction.

Conclusion

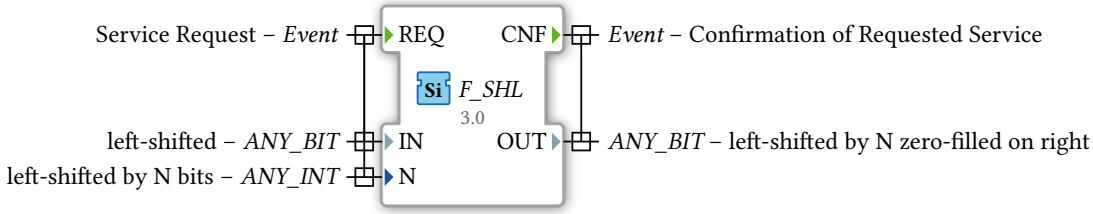
The `F_ROR` function block is a useful tool for applications requiring bitwise rotations. Its simple interface and support for various data types make it versatile. Unlike simple shift operations, it preserves all bits of the input value, making it indispensable for certain applications.

F_SHL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_SHL performs a left shift on any bit data type. The bits of the input value are shifted left by a specified number of positions, and the resulting free positions on the right are padded with zeros. This block is classified according to the IEC 61131-3 standard and is suitable for bitwise operations.



Interface Structure

Event Inputs

- **REQ**: Service request. Triggers the left shift. Linked to the data inputs IN and N.

Event Outputs

- **CNF**: Confirmation of the requested service. Signals the completion of the operation. Linked to the data output OUT.

Data Inputs

- **IN**: The input value to be left-shifted. The data type is ANY_BIT (any bit data type).
- **N**: The number of positions to shift. The data type is ANY_INT (any integer data type).

Data Outputs

- **OUT**: The result of the left shift. The data type is the same as the input value IN. The resulting free positions on the right are padded with zeros.

Adapters

No adapters are defined.

Adapters ## Functionality

The function block F_SHL performs a bitwise left shift of the input value IN by N positions. The bits are shifted to the left, and the newly created positions on the right side are padded with zeros. The operation is triggered by the event REQ and acknowledged by CNF.

Technical Features

- Supports any bit data type (ANY_BIT) for the input value IN.
- The number of shifts N can be any integer data type (ANY_INT).
- The operation is standardized according to IEC 61131-3.

State Overview

1. **Idle State**: Waits for the event REQ.
2. **Execution State**: Performs the left shift when REQ is received.
3. **Acknowledge State**: Sends CNF after the operation is complete.

Application Scenarios

- Bit manipulation in control applications.
- Shift operations in digital signal processing algorithms.
- General bitwise operations in IEC 61131-3 compliant control systems.

⚖️ Comparison with Similar Function Blocks

- **F_SHR**: Performs a right shift, unlike the left shift of F_SHL.
- **F_ROL** and **F_ROR**: Rotate bits instead of shifting them, with overflowing bits being reinserted on the other side.

🛠️ Related Exercises

- Exercise_034

Conclusion

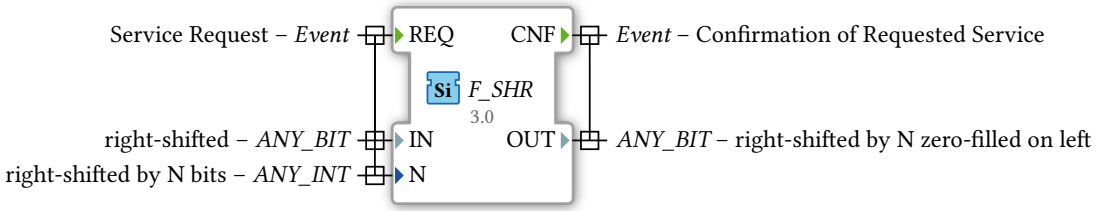
The F_SHL function block is an efficient tool for bitwise left shifts in control systems. Its flexibility in supporting various bit and integer data types makes it versatile. Clear event handling and compliance with the IEC 61131-3 standard ensure reliable integration into existing systems.

F_SHR

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_SHR performs a bitwise right shift. It is part of the standard bit operators according to IEC 61131-3 and is used for logical and arithmetic operations in control applications.



Interface Structure

Event Inputs

- REQ: Starts the right shift. It is linked to the data IN and N.

Event Outputs

- CNF: Confirms the completion of the right shift. Returns the result OUT.

Data Inputs

- IN (ANY_BIT): The input value to be shifted right.
- N (ANY_INT): The number of bits to shift.

Data Outputs

- OUT (ANY_BIT): The result of the right shift. The freed bits on the left are padded with zeros.

Adapters

No adapters available.

Functionality

1. Upon a REQ event, the value IN is shifted right by N bits.
2. The freed bits on the left side are padded with zeros (logical right shift).
3. The result is output via OUT, triggering the CNF event.

Technical Features

- Supports all bit data types (ANY_BIT), e.g. B. BOOL, BYTE, WORD, DWORD, LWORD.
- N can use any integer type (ANY_INT), e.g. B. SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT.
- The right shift is independent of the sign (logical shift).

Status Overview

- **Idle**: Waiting for REQ.
- **Processing**: Performing the right shift.
- **Completed**: Sending CNF with the result.

Application Scenarios

- Bit manipulation in control algorithms.
- Data compression or extraction.

- Fast division by powers of two (for unsigned numbers).

⚖️ Comparison with similar function blocks

- `F_SHL`: Performs a left shift.
- `F_ROR`/`F_ROL`: Rotation operations instead of shifts.
- `F_SHR` is specific for logical right shifts (zero padding).

Conclusion

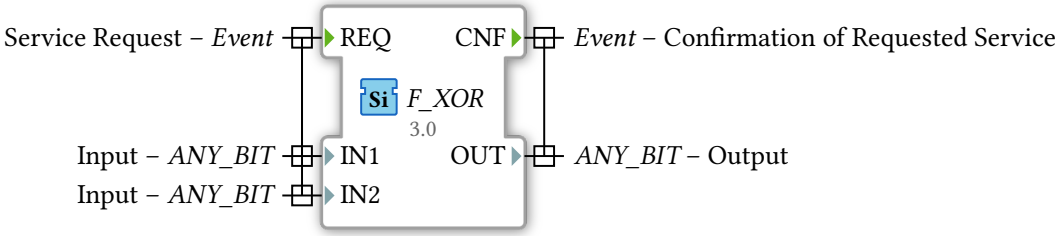
The `F_SHR` function block is an efficient tool for bitwise operations in IEC 61131-3 environments. Its simple interface and standards compliance make it ideal for control applications involving bit manipulation.

F_XOR

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_XOR` performs a bitwise XOR operation (exclusive OR) on two input values. It is part of the standard bitwise operations according to IEC 61131-3 and is suitable for logical processing in control applications.



Interface Structure

Event Inputs

- `REQ`: Starts the XOR operation. When activated, the input data `IN1` and `IN2` are processed.

Event Outputs

- `CNF`: Confirms successful execution of the XOR operation and outputs the result via `OUT`.

Data Inputs

- `IN1`: First input value for the XOR operation (Type: `ANY_BIT`).
- `IN2`: Second input value for the XOR operation (Type: `ANY_BIT`).

Data Outputs

- `OUT`: Result of the bitwise XOR operation (Type: `ANY_BIT`).

Adapters

No adapters available.

Functionality

When the event input `REQ` is activated, the function block performs a bitwise XOR operation between the values `IN1` and `IN2`. The result is output via the data output `OUT`, and the event `CNF` is triggered to signal the completion of the operation.

Technical Features

- Supports the data type `ANY_BIT`, meaning that various bit data types (e.g., BOOL, BYTE, WORD, DWORD) can be processed.
- The initial values of the inputs are empty, requiring explicit assignment of the input values.

State Overview

1. **Idle State**: Waits for the event `REQ`.
2. **Processing State**: Performs the XOR operation when `REQ` is activated.

3. **Output State**: Sends the result via `OUT` and triggers `CNF`.

Application Scenarios

- Logical processing in control systems.
- Bitwise manipulation of data in industrial automation processes.
- Implementation of security checks or checksum calculations.

⚖️ Comparison with Similar Function Blocks

- **F_AND / F_OR**: Perform bitwise AND and OR operations, respectively, while `F_XOR` offers an exclusive OR operation. - **F_NOT**: Inverts bits while `F_XOR` compares two input values.

Conclusion

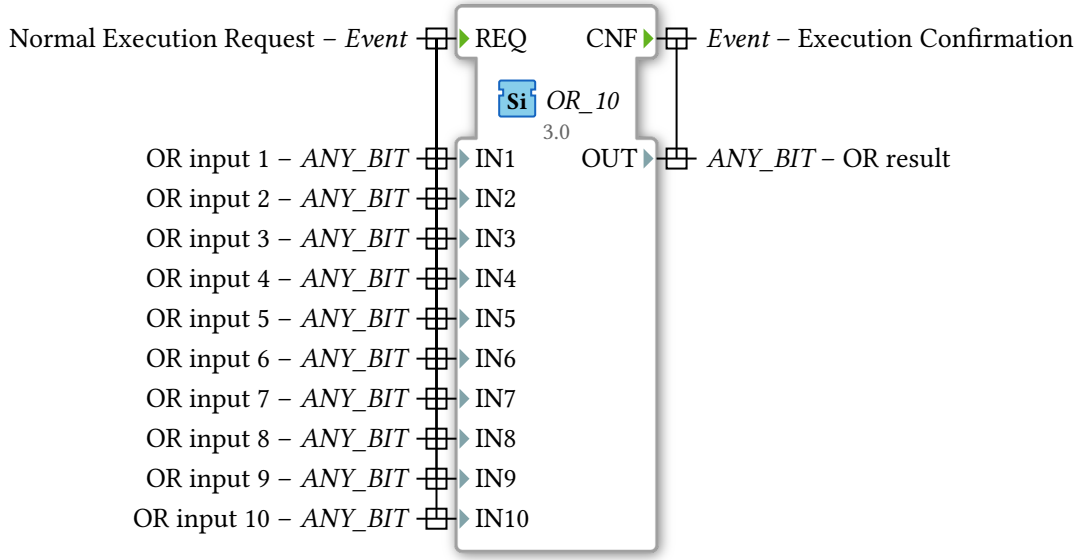
The `F_XOR` function block is an efficient tool for bitwise XOR operations in IEC 61131-3 compliant control systems. Its flexibility with the `ANY_BIT` data type makes it versatile for use in various automation applications.

OR_10

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_10` performs a bitwise OR operation on up to 10 input variables. It is a generic function block that can work with any bit data type (ANY_BIT). The block is classified according to the IEC 61131-3 standard and is used for the simple execution of logical OR operations in control applications.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. When this event is triggered, all input variables (IN1 to IN10) are read and the OR operation is performed.

Event Outputs

- **CNF**: Signals the successful completion of the OR operation. This event is output along with the result (OUT).

Data Inputs

- **IN1 to IN10**: Up to 10 input variables of type ANY_BIT, containing the values to be combined. Each variable can have any bit data type (e.g., BOOL, BYTE, WORD, DWORD, etc.).

Data Outputs

- **OUT**: The result of the bitwise OR operation on all input variables. The data type corresponds to that of the input variables.

Adapters

No adapters are defined for this function block.

Functionality

The function block performs a bitwise OR operation on all active input variables. This means that a logical OR operation is performed on each bit of the input variables. The result is output at the output variable OUT. The operation is initiated by the REQ event and confirmed by the CNF event.

Technical Features

- **Generic Implementation**: The function block can work with any bit data type, enabling high flexibility.
- **Scalability**: Up to 10 input variables can be processed, making the block suitable for more complex applications.

State Overview

1. **Idle State**: The block waits for the REQ event.
2. **Execution State**: Upon receiving REQ, the input variables are read and the OR operation is performed.
3. **Acknowledge State**: After successful operation, the CNF event is triggered, and the block returns to the idle state.

Application Scenarios

- Logical operations in control applications
- Signal processing in industrial automation systems
- Combining multiple binary signals to produce a result

⚖️ Comparison with similar function blocks

Compared to simple OR blocks with fewer inputs, OR_10 offers the ability to combine up to 10 signals, making it suitable for more complex applications. Other blocks are often limited to fewer inputs or are not implemented generically.

Conclusion

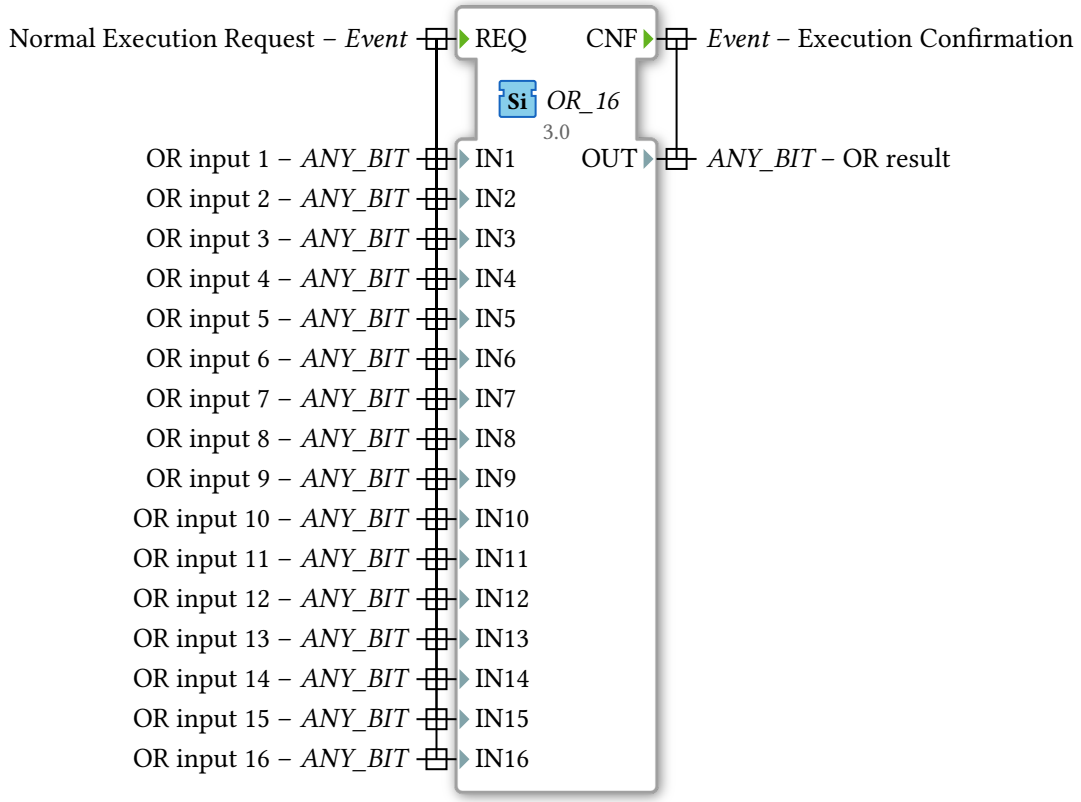
The OR_10 function block is a flexible and powerful tool for bitwise OR operations in IEC 61131-3-based control systems. Its generic nature and support for up to 10 input variables make it particularly suitable for demanding applications.

OR_16

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_16` performs a bitwise logical OR operation on up to 16 input variables. It is a generic function block that can work with various bit data types (ANY_BIT). The block is part of the package `iec61131::bitwiseOperators` and conforms to the IEC 61131-3 standard.



Interface Structure

Event Inputs

- `REQ` (Normal Execution Request): Triggers the calculation. It is processed with all 16 input variables.

Event Outputs

- `CNF` (Execution Confirmation): Confirms the execution of the operation. Linked to the result `OUT`.

Data Inputs

16 inputs of type `ANY_BIT`:

- `IN1` to `IN16`: The input values for the OR operation. Each input is annotated as "OR input X" (where X is the input number).

Data Outputs

- `OUT` (ANY_BIT): The result of the bitwise OR operation of all input values.

Adapters

The function block does not use adapters.

Functionality

Upon receiving the `REQ` event, the block performs a bitwise OR operation on all active input values (`IN1` to `IN16`). The result is output to `OUT`, and the `CNF` event is triggered.

Technical Features

- Generic Implementation: Works with all `ANY_BIT` data types (e.g., BOOL, BYTE, WORD, DWORD, LWORD)
- Supports up to 16 input variables
- Compatible with IEC 61131-3 standard

State Overview

1. Wait State: Waits for `REQ` event

2. Execute State: On REQ, inputs are processed and the result is calculated
3. Acknowledge State: Outputs the result and triggers CNF

Application Scenarios

- Bitwise logic operations in control applications
- Signal processing with multiple conditions
- Generic logic blocks in libraries

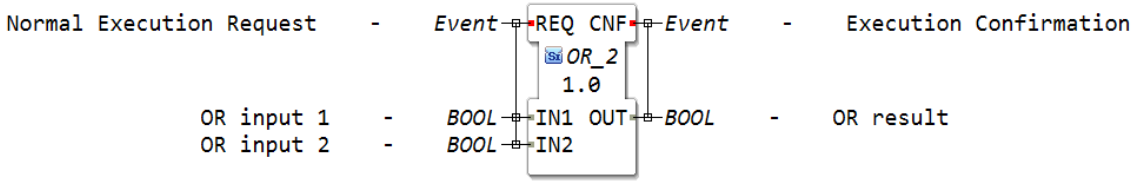
⚖️ Comparison with Similar Blocks

- Compared to simpler OR blocks (e.g., OR_2), it offers OR_16 offers significantly more inputs
- Similar to AND_16 or XOR_16, but with a different logical operation
- Generic implementation allows for more flexible use than type-specific blocks

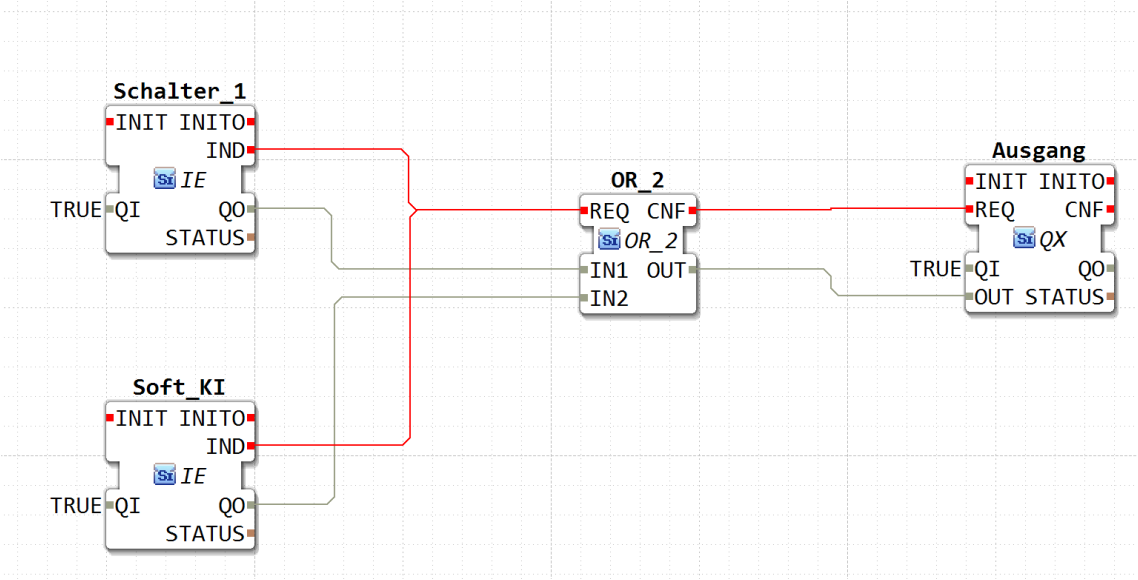
Conclusion

The OR_16 function block offers a flexible and standards-compliant solution for bitwise OR operations with up to 16 inputs. Its generic nature makes it versatile, while its clear interface structure facilitates integration into various control applications.

OR_2



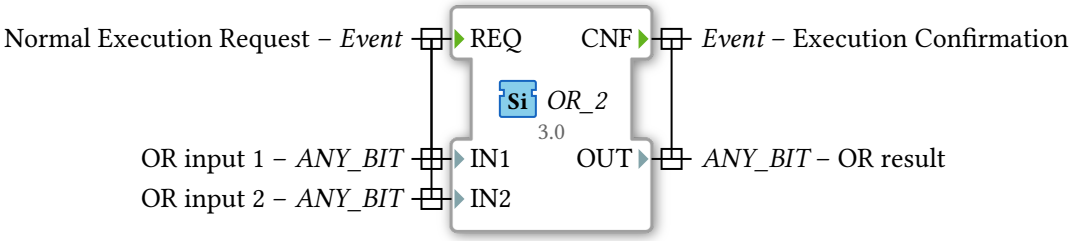
A common use case is a soft AI and an AUX button. TRUE dominates, so as soon as an input is TRUE, the output is set to TRUE.



• – ○ • – ○ • – ○ •

Introduction

The OR_2 function block is a generic function block for calculating bitwise Boolean OR operations. It performs a logical OR operation on two input values and outputs the result. The function block is classified according to the IEC 61131-3 standard and supports generic data types for maximum flexibility.



Interface Structure

Event Inputs

- **REQ**: Normal execution request - triggers the calculation of the OR operation

Event Outputs

- **CNF**: Execution confirmation - signals the completion of the calculation

Data Inputs

- **IN1**: OR input 1 (Type: ANY_BIT) - first operand for the OR operation
- **IN2**: OR input 2 (Type: ANY_BIT) - second operand for the OR operation

Data Outputs

- **OUT**: OR result (Type: ANY_BIT) - result of the bitwise OR operation

Adapters

This function block does not use adapters.

Functionality

Upon arrival of the REQ event, the OR_2 block performs a bitwise OR operation between the two input values IN1 and IN2. The result is output at the OUT data output, and simultaneously, the CNF event is generated to confirm successful calculation.

The OR operation is performed at the bit level, with each bit of the input values being processed separately. The result bit is 1 if at least one of the corresponding input bits is 1.

Technical Features

- **Generic Type Support**: Uses the ANY_BIT data type, meaning the block can work with various bit data types (BOOL, BYTE, WORD, DWORD, LWORD)
- **Standard-Compliant Implementation**: Conforms to the IEC 61131-3 standard for bitwise Boolean functions
- **Event-Driven Execution**: The calculation is only performed when a REQ event is active

State Transitions

1. **Wait State**: Block waits for a REQ event
2. **Calculation State**: Upon a REQ event, IN1 and IN2 are processed
3. **Output State**: Result is output to OUT and a CNF event is generated
4. **Return**: Block returns to the wait state

Application Scenarios

- Bitwise mask operations in control applications
- Combining status flags in process control systems
- Logical operations in digital circuits
- Data manipulation in Communication Protocols

⚖️ Comparison with Similar Building Blocks

Compared to other Boolean operations like AND or XOR, OR_2 offers the specific property that the result is true if at least one of the operands is true. Unlike simple Boolean OR blocks, OR_2 supports generic bit data types, thus enabling the processing of various data widths.

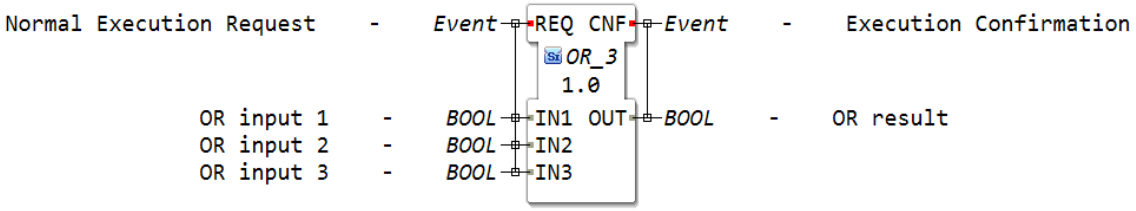
🛠️ Related exercises

- Uebung_002a3
- Uebung_002a3_AX
- Uebung_002b2
- Uebung_002b3_AX
- Uebung_035a
- Uebung_035a2
- Uebung_035a2_AX
- Uebung_035a3
- Exercise_035a3_AX
- Exercise_039_sub_Outputs
- Exercise_089
- Exercise_160_AX
- Exercise_160b2_AX
- Exercise_160b_AX

Conclusion

The OR_2 function block provides a flexible and standards-compliant solution for bitwise OR operations in automation systems. Its generic nature allows for use in various application contexts with different data types, while the event-driven architecture ensures efficient resource utilization.

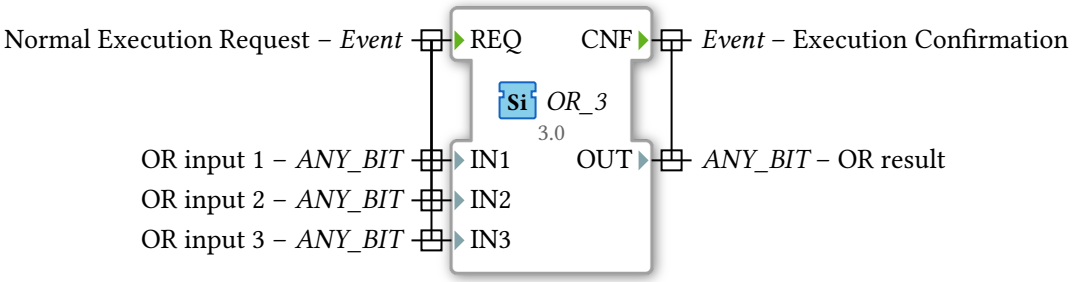
OR_3



• – ◦ • – ◦ • – ◦ •

Introduction

The OR_3 function block is a generic function block for calculating bitwise Boolean OR operations with three inputs. It performs a logical OR operation at the bit level and supports various bit data types according to the IEC 61131-3 standard.



Interface Structure

Event Inputs

- **REQ**: Normal execution request - triggers the calculation of the OR operation

Event Outputs

- **CNF**: Execution confirmation - signals the completion of the calculation

Data Inputs

- **IN1**: OR input 1 (ANY_BIT data type)
- **IN2**: OR input 2 (ANY_BIT data type)
- **IN3**: OR input 3 (ANY_BIT data type)

Data Outputs

- **OUT**: OR result (ANY_BIT data type)

Adapters

No adapters available.

Functionality

The OR_3 function block performs a bitwise OR operation on the three input values IN1, IN2, and IN3 upon each REQ event. The result is output at the OUT data output, and simultaneously a CNF event is generated to signal the completion of the operation.

The operation is performed separately for each bit position: OUTi = IN1i OR IN2i OR IN3i

Technical Features

- Supports the generic ANY_BIT data type, enabling its use with various bit data types such as BOOL, BYTE, WORD, DWORD, etc.
- Implemented as a generic function block according to the IEC 61131-3 standard
- Belongs to the “iec61131::bitwiseOperators” package

State Overview

1. **Wait State**: Block waits for a REQ event
2. **Calculation State**: Inputs are processed upon a REQ event
3. **Output State**: Result is output to OUT and a CNF event is generated

Application Scenarios

- Combining multiple digital signals in control applications
- Mask operations in data processing
- Parallel processing of multiple conditions
- Signal linking in automation systems

⚖️ Comparison with similar building blocks

Compared to simple OR blocks with only two inputs, OR_3 offers the possibility of directly connecting three inputs, simplifying the program structure and avoiding additional chaining. Unlike arithmetic blocks, OR_3 operates exclusively at the bit level.

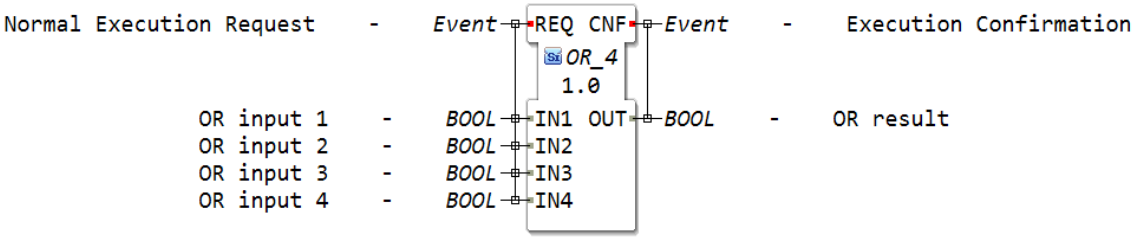
🛠️ Related Exercises

- Exercise_002a5_AX
- Exercise_002a5b_AX

Conclusion

The OR_3 function block represents an efficient and flexible solution for bitwise OR operations with three inputs. Its generic nature allows its use in various application areas and with different bit data types, making it a versatile tool in automation technology.

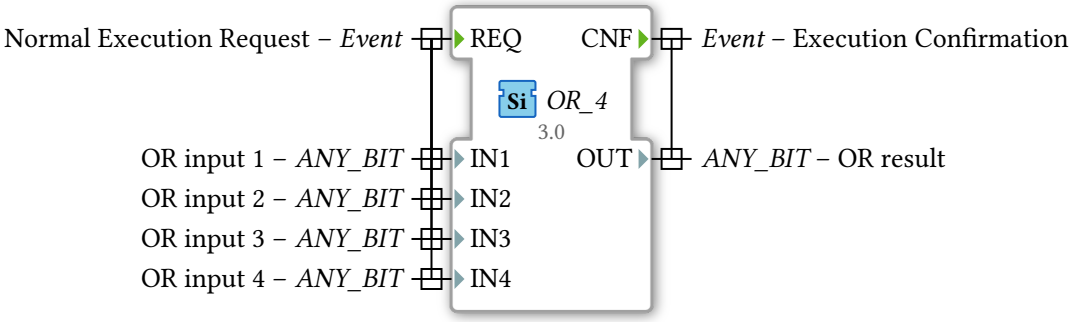
OR_4



• – ○ • – ○ • – ○ •

Introduction

The OR_4 function block is a generic function block for calculating bitwise OR operations with four inputs. It performs a bitwise OR operation on the input data and outputs the result. The function block is suitable for all ANY_BIT data types, thus enabling flexible use with various bit data types.



Interface Structure

Event Inputs

- **REQ**: Normal execution request - triggers the calculation of the OR operation

Event Outputs

- **CNF**: Execution confirmation - signals the completion of the calculation

Data Inputs

- **IN1**: OR input 1 (ANY_BIT)
- **IN2**: OR input 2 (ANY_BIT)
- **IN3**: OR input 3 (ANY_BIT)
- **IN4**: OR input 4 (ANY_BIT)

Data Outputs

- **OUT**: OR result (ANY_BIT)

Adapters

No adapters available.

Functionality

Upon arrival of the REQ event, the OR_4 block performs a bitwise OR operation on all four inputs. The operation is executed separately for each bit position: OUTi = IN1i OR IN2i OR IN3i OR IN4i. After completion of the calculation, the CNF event is output along with the result.

Technical Features

- Generic building block for ANY_BIT data types
- Supports various bit data types (BOOL, BYTE, WORD, DWORD, LWORD)
- Bitwise processing of all inputs
- Four independent inputs for maximum flexibility

State Transitions

1. **Wait State**: Block waits for REQ event
2. **Compute State**: Upon REQ event, block performs OR calculation
3. **Output State**: Sends CNF event with result and returns to wait state

Application Scenarios

- Bitmask operations in control systems
- Combining multiple status bits into an overall status
- Parallel processing of multiple conditions
- Signal processing in industrial automation systems

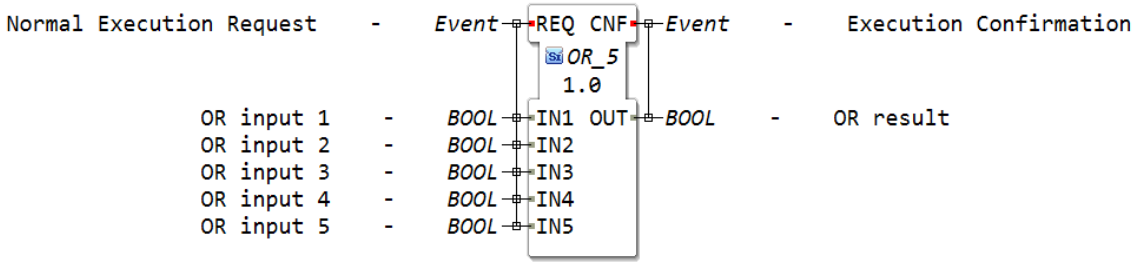
⚖️ Comparison with Similar Building Blocks

Compared to simpler OR blocks with fewer inputs, OR_4 offers the ability to process four inputs in parallel, reducing the number of blocks required in complex circuits. Compared to function blocks with fixed data types, OR_4 offers greater flexibility through the use of ANY_BIT.

Conclusion

The OR_4 function block is a versatile and efficient four-input function block for bitwise OR operations. Its generic nature allows for use in various applications, while its clear event-driven control ensures reliable integration into more complex control sequences.

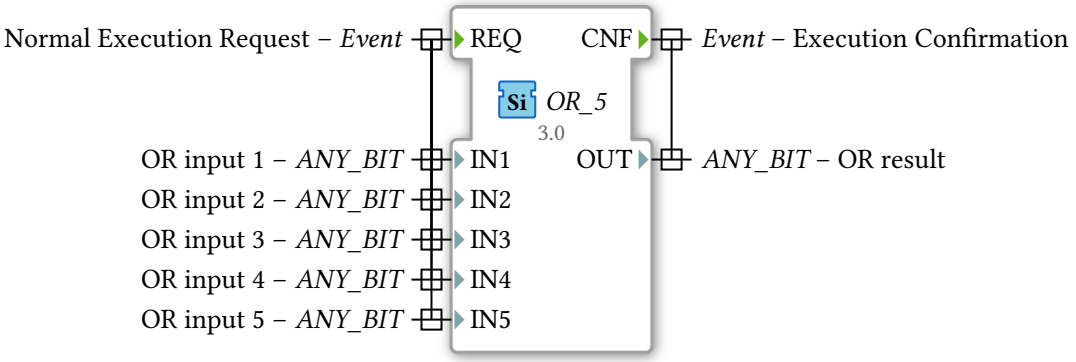
OR_5



• – ◦ • – ◦ • – ◦ •

Introduction

The OR_5 function block is a generic function block for performing a bitwise OR operation with five inputs. It implements a standard bit function according to IEC 61131-3 and can work with any bit data type.



Interface Structure

Event Inputs

- **REQ**: Normal execution request - triggers the calculation of the OR operation

Event Outputs

- **CNF**: Execution confirmation - signals the completion of the calculation

Data Inputs

- **IN1**: OR input 1 (ANY_BIT data type)
- **IN2**: OR input 2 (ANY_BIT data type)
- **IN3**: OR input 3 (ANY_BIT data type)
- **IN4**: OR input 4 (ANY_BIT data type)
- **IN5**: OR input 5 (ANY_BIT data type)

Data Outputs

- **OUT**: OR result (ANY_BIT data type)

Adapters

No adapter interfaces are available.

Functionality

The OR_5 function block performs a bitwise OR operation on all five input values. When the REQ event is triggered, the values at IN1 to IN5 are processed, and the result is output to OUT. The CNF event signals the completion of the operation.

The OR operation is performed according to the following logic:
OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5

Technical Features

- **Generic Data Type**: Uses ANY_BIT, enabling use with various bit data types (BOOL, BYTE, WORD, DWORD, LWORD)
- **Five Inputs**: Offers more flexibility than standard OR blocks with typically two inputs
- **Event-Driven**: Operates in real time with REQ/CNF signaling

State Transitions

1. **Wait State**: Block waits for a REQ event
2. **Compute State**: Upon REQ, all inputs are processed
3. **Output State**: Result is set to OUT and CNF is triggered
4. Returns to Wait State

Application Scenarios

- Combining multiple binary signals in control applications
- Monitoring systems with multiple input conditions
- Signal processing in industrial automation systems
- Use in safety circuits with multiple monitoring sensors

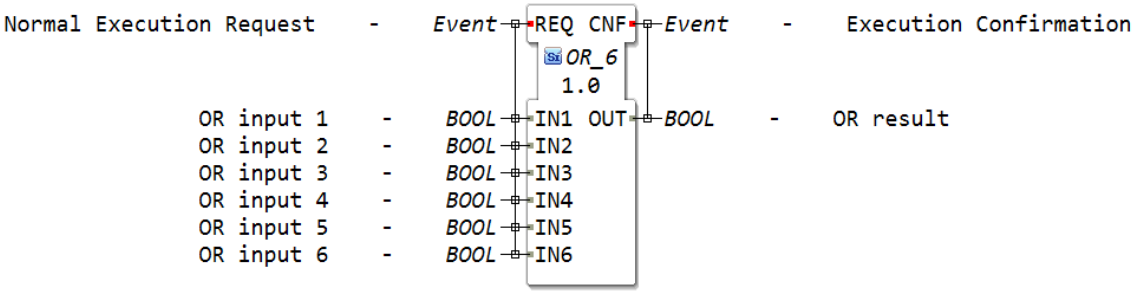
⚖️ Comparison with similar devices Function Blocks

- **Standard OR Blocks**: Typically only two inputs, OR_5 offers five inputs
- **AND Blocks**: Perform AND operations instead of OR operations
- **XOR Blocks**: Perform exclusive OR operations

Conclusion

The OR_5 function block is a versatile and powerful five-input block for bitwise OR operations. Its generic nature allows for use in various application areas, while its event-driven architecture ensures precise control over execution timing.

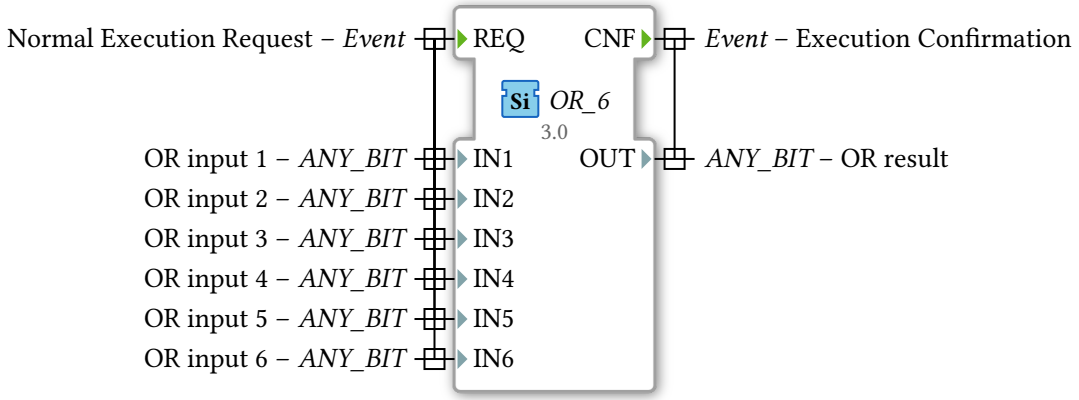
OR_6



• – ◦ • – ◦ • – ◦ •

Introduction

The OR_6 function block is a generic bitwise OR function block that processes six input values. It performs a bitwise logical OR operation on the six input values and outputs the result. The function block is classified according to the IEC 61131-3 standard and can work with any bit data type (ANY_BIT).



Interface Structure

Event Inputs

- **REQ**: Normal Execution Request - triggers the calculation of the OR operation

Event Outputs

- **CNF**: Execution Acknowledgement - signals the completion of the calculation

Data Inputs

- **IN1**: OR Input 1 (ANY_BIT)
- **IN2**: OR Input 2 (ANY_BIT)
- **IN3**: OR Input 3 (ANY_BIT)
- **IN4**: OR Input 4 (ANY_BIT)
- **IN5**: OR Input 5 (ANY_BIT)
- **IN6**: OR Input 6 (ANY_BIT)

Data Outputs

- **OUT**: OR Result (ANY_BIT)

Adapters

No adapters Available.

Functionality

Upon receiving the REQ event, the OR_6 block performs a bitwise OR operation on all six input values. The operation is executed according to the following logic:

OUT = IN1 OR IN2 OR IN3 OR IN4 OR IN5 OR IN6 Once the calculation is complete, the CNF event is output along with the result value. ...

Technical Features

- **Generic Data Type**: Uses ANY_BIT, enabling use with various bit data types (BOOL, BYTE, WORD, DWORD, LWORD)
- **Six Inputs**: Processes up to six input values in a single operation
- **Bitwise Operation**: Performs the OR operation at the bit level
- **Event-Driven**: Computation is only performed upon a REQ event

State Transitions

1. **Wait State**: Block waits for a REQ event
2. **Compute State**: All inputs are processed upon a REQ event
3. **Output State**: After computation, a CNF event with the result is output
4. **Return**: Returns to the wait state

Application Scenarios

- **Bitmask Operations**: Combining multiple bitmasks
- **Control Logic**: Summarizing multiple conditions

- **Signal Processing**: Combining multiple digital signals
- **Data Filtering**: Selective activation of bits from different sources

⚖️ Comparison with similar building blocks

- **OR_2**: Processes only two inputs
- **OR_4**: Processes four inputs
- **GEN_OR**: Generic basic implementation that extends OR_6
- **XOR building blocks**: Perform exclusive OR operations

Conclusion

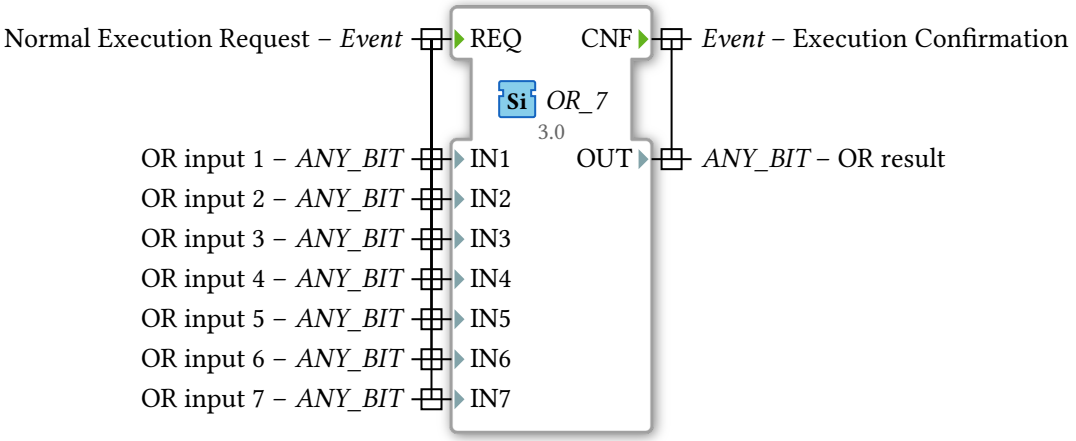
The OR_6 function block offers an efficient solution for bitwise OR operations with up to six input values. Its generic nature allows for maximum flexibility in using different bit data types, while the event-driven architecture ensures precise control of the calculation timing.

OR_7

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `OR_7` performs a bitwise logical OR operation on up to seven input variables. It is a generic function block that can work with various bit data types (`ANY_BIT`). The block is part of the `iec61131::bitwiseOperators` library and implements a standard bitwise operation according to IEC 61131-3.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This event passes all input variables (`IN1` to `IN7`).

Event Outputs

- **CNF**: Signals the successful execution of the OR operation and returns the result via the variable `OUT`.

Data Inputs

- **IN1** to **IN7** (`ANY_BIT`): Up to seven input variables on which the bitwise OR operation is performed. Each input can have any bit data type (e.g., `BOOL`, `BYTE`, `WORD`, `DWORD`, `LWORD`).

Data Outputs

- **OUT** (`ANY_BIT`): The result of the bitwise OR operation. The data type corresponds to that of the input variables.

Adapters

The function block does not use any adapters.

Functionality

When the `REQ` event is triggered, the function block performs a bitwise OR operation on the values of the input variables `IN1` through `IN7`. The result is stored in the variable `OUT`, and the `CNF` event is triggered to indicate successful computation.

Technical Features

- **Generic Implementation**: The function block can handle various bit data types (`ANY_BIT`), making it highly versatile.
- **Scalability**: Supports up to seven input variables, enabling more complex logical operations.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Execution State**: Performs the OR operation and calculates the result.
3. **Acknowledgement State**: Triggers the `CNF` event and outputs the result via `OUT`.

Application Scenarios

- **Logical Operations**: Combining multiple binary signals to produce a common result.
- **Control Engineering**: Used in control algorithms where multiple conditions must be met simultaneously (OR operation).
- **Data Processing**: Bitwise operations at the byte, word, or DWord level.

⚖️ Comparison with Similar Function Blocks

- **OR (Standard)**: Standard OR blocks in IEC 61131-3 typically support only two inputs. `OR_7` extends this functionality to up to seven inputs.
- **GEN_OR**: The generic OR block on which `OR_7` is based can theoretically process any number of inputs, but is less specific in its implementation.

Conclusion

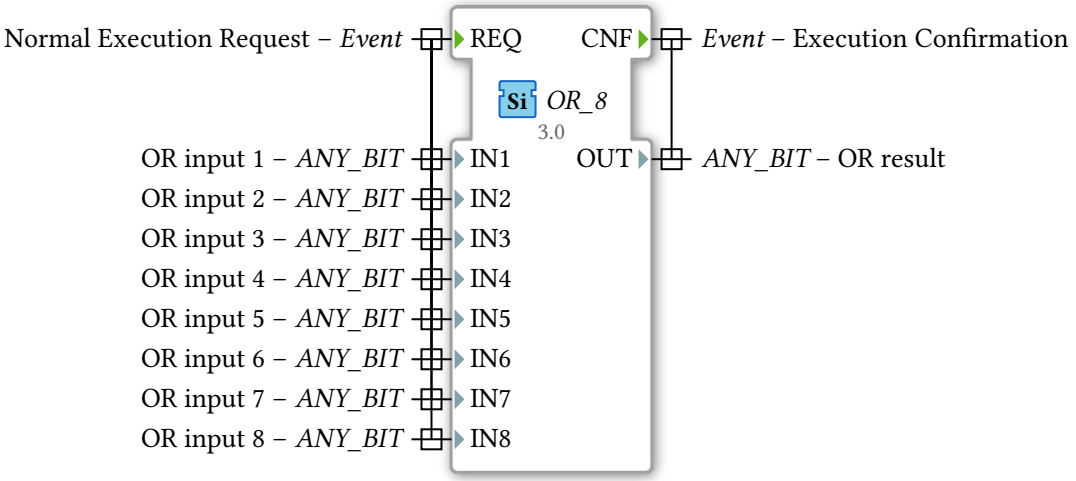
The `OR_7` function block offers a flexible and powerful solution for bitwise OR operations with up to seven input variables. Its generic nature allows its use with various bit data types, making it a versatile tool in control and automation engineering.

OR_8

• – ◦ • – ◦ • – ◦ •

Introduction

The OR_8 function block is a generic function block for calculating a bitwise OR operation. It supports up to 8 inputs and is classified according to the IEC 61131-3 standard. This function block is particularly suitable for applications where multiple binary signals need to be logically combined.



Interface Structure

Event Inputs
- **REQ**: Starts the execution of the function block. It is linked to all data inputs.

Event Outputs
- **CNF**: Signals the successful execution of the function block. It is linked to the data output `OUT`.

Data Inputs
- **IN1** to **IN8**: Each of these inputs accepts an arbitrary bit type (`ANY_BIT`). These represent the operands for the OR operation.

Data Outputs
- **OUT**: Returns the result of the bitwise OR operation of inputs `IN1` to `IN8`. The data type corresponds to that of the inputs (`ANY_BIT`).

Adapters
This function block does not have any adapters.

Functionality
The OR_8 function block performs a bitwise OR operation on the values at inputs `IN1` to `IN8`. The result is output at `OUT` as soon as the event `REQ` is received. Execution is confirmed by the event `CNF`.

Technical Features
- **Generic Implementation**: The function block is generic and can work with different bit types (`ANY_BIT`).
- **Flexible Number of Inputs**: Supports up to 8 inputs, making it suitable for complex logic operations.

State Overview
1. **Idle State**: Waits for the event `REQ`.
2. **Execution State**: Performs the OR operation and sets the output `OUT`.
3. **Acknowledgement State**: Sends the event `CNF` to confirm execution.

Application Scenarios
- Logical combination of multiple sensor data.
- Control logic in industrial automation systems.
- Combination of binary control signals.

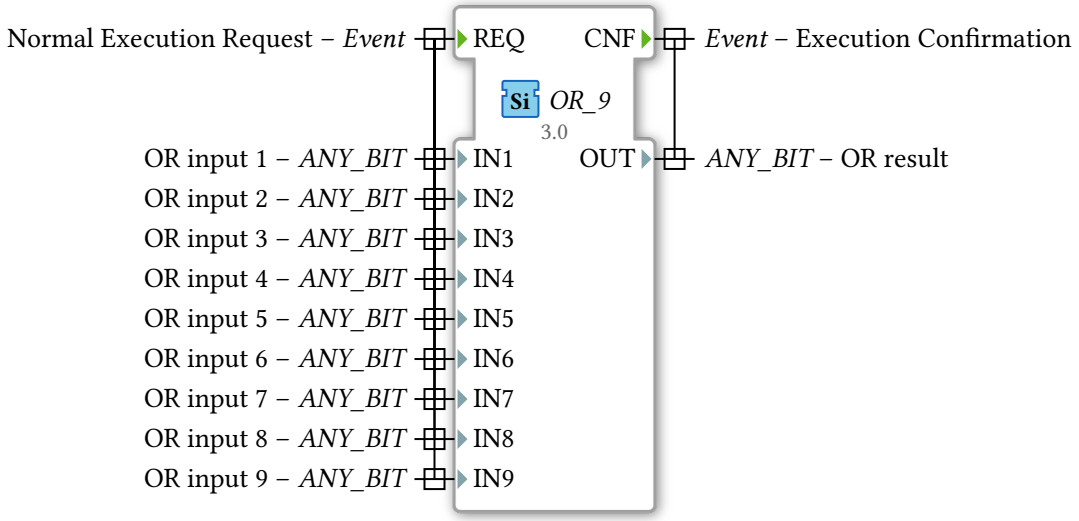
⚖️ Comparison with Similar Function Blocks
- **OR_2/OR_4**: Similar function blocks with fewer inputs (2 or 4, respectively). OR_8 offers greater flexibility for more complex applications.
- **GEN_OR**: Generic base implementation that is further specialized by OR_8.

Conclusion
The OR_8 function block is a powerful tool for bitwise OR operations in industrial automation. Its generic nature and support for up to 8 inputs make it particularly versatile.

OR_9

• − ◦ • − ◦ • − ◦ •

Introduction
The OR_9 function block is a generic function block for calculating a bitwise logical OR operation. It supports up to nine input variables and outputs the result of the OR operation. The function block is part of the IEC 61131-3 standard library for bitwise Boolean functions.



Interface Structure

Event Inputs

- **REQ**: Normal execution request input. Triggers the calculation of the OR operation. Linked to all data inputs (IN1 to IN9).

Event Outputs

- **CNF**: Execution confirmation. Triggered after the calculation of the OR operation. Linked to the data output OUT.

Data Inputs

- **IN1** (ANY_BIT): OR Input 1
- **IN2** (ANY_BIT): OR Input 2
- **IN3** (ANY_BIT): OR Input 3
- **IN4** (ANY_BIT): OR Input 4
- **IN5** (ANY_BIT): OR Input 5
- **IN6** (ANY_BIT): OR Input 6
- **IN7** (ANY_BIT): OR Input 7
- **IN8** (ANY_BIT): OR Input 8
- **IN9** (ANY_BIT): OR Input 9

Data Outputs

- **OUT** (ANY_BIT): Result of the OR operation of all inputs

Adapters

This module does not have any adapters.

Functionality

The OR_9 function block performs a bitwise logical OR operation on all active inputs (IN1 to IN9) when the REQ event is received. The result is output at the OUT output, and the CNF event is triggered simultaneously.

Technical Features

- Supports the generic data type ANY_BIT for all inputs and the output
- Can work with different bit lengths (e.g., BYTE, WORD, DWORD)
- Implemented as a generic function block (GEN_OR)

State Overview

1. Wait state: Waits for a REQ event
2. Compute state: Performs an OR operation

3. Output state: Sets OUT and triggers CNF
4. Returns to wait state

Application Scenarios

- Bitwise signal processing
- Combinational logic circuits
- Control logic with multiple input conditions
- Signal processing in industrial automation systems

⚖️ Comparison with Similar Function Blocks

- Compared to simpler OR function blocks, OR_9 offers the ability to process up to nine inputs
- Similar to other bitwise operators, but specifically optimized for OR operations
- Generic implementation allows for more flexible application than type-specific variants

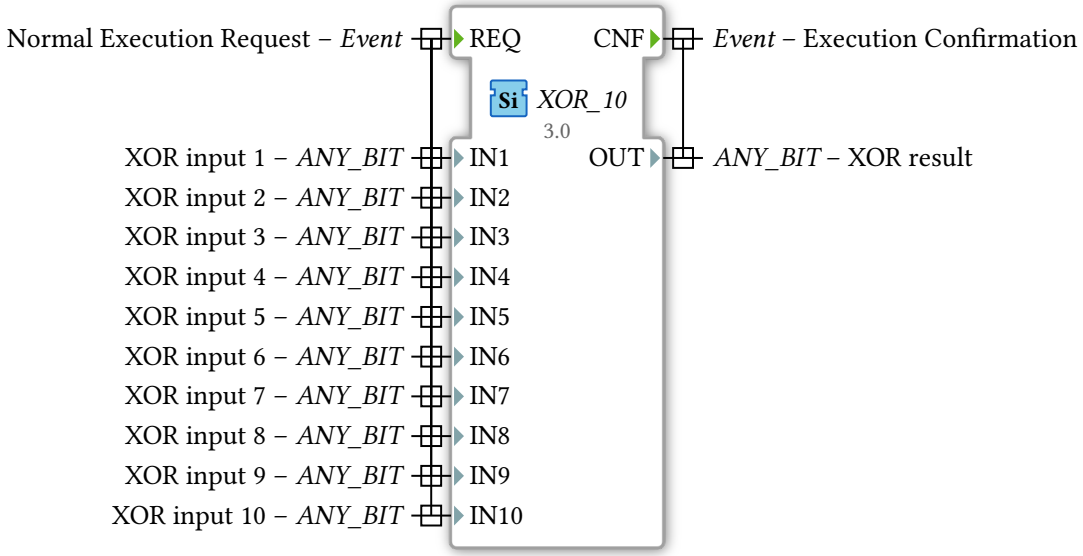
Conclusion

The The OR_9 function block is a versatile building block for bitwise OR operations in industrial control systems. Its generic implementation and support for up to nine inputs make it particularly flexible for various applications. The clear interface structure and compliance with IEC 61131-3 enable easy integration into existing automation solutions.

XOR_10

• – ◦ • – ◦ • – ◦ •

The XOR_10 function block is a generic function block for calculating a bitwise XOR operation with up to 10 inputs. It is part of the IEC 61131-3 standard library for bitwise Boolean operations and can be used with various bit data types (ANY_BIT).



- **REQ**: Starts the calculation of the XOR operation. It connects to all data inputs (IN1 to IN10).
- **CNF**: Signals the completion of the calculation and outputs the result via the OUT output.
- **IN1** to **IN10** (ANY_BIT): Up to 10 input values for XOR calculation. Each input can contain any bit data type (BOOL, BYTE, WORD, DWORD, LWORD).
- **OUT** (ANY_BIT): Result of the bitwise XOR operation of all active inputs.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use adapters.

Upon receiving the REQ event, the function block calculates a bitwise XOR operation of all active inputs (IN1 to IN10). The result is output at the OUT output, and the CNF event is triggered.

- Result bit = 1 if an odd number of input bits at that position are 1
- Result bit = 0 if an even number of input bits at that position are 1
- Supports generic bit data types (ANY_BIT)
- Processes up to 10 input values
- All inputs must have the same data type
- Unused inputs are treated as 0
- Part of the `iec61131::bitwiseOperators` package

1. **Wait State**: Waits for REQ event
2. **Compute State**: Performs XOR operation
3. **Output State**: Sends result and CNF event
4. Returns to Wait State

- Bitwise encryption operations
- Parity checks
- Error detection in digital systems
- Control logic with multiple Input Signals
- Generic Logic Operations in Automation Systems
- Compared to simple 2-input XOR blocks, XOR_10 offers greater flexibility
- Similar to OR_10/AND_10 blocks, but with XOR logic
- Generic implementation as opposed to type-specific blocks (e.g., XOR_BOOL, XOR_WORD)

The XOR_10 function block provides a flexible solution for complex bitwise XOR operations in IEC 61131-3-based control systems. Its generic nature and support for up to 10 inputs make it particularly suitable for applications requiring more than the traditional two inputs.

Functionality

Technical Features

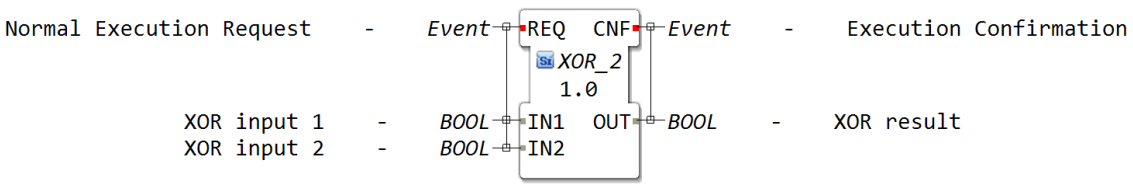
State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

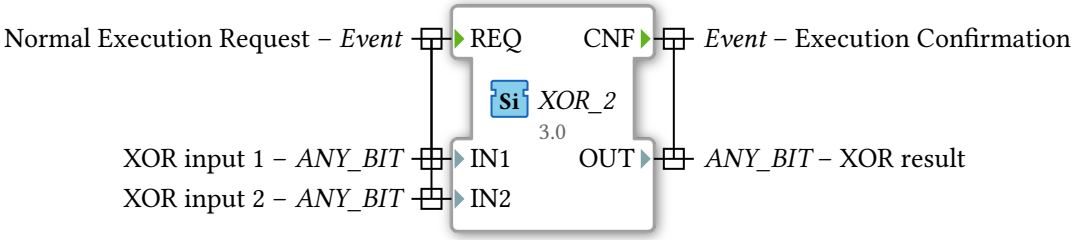
XOR_2



• – ◦ • – ◦ • – ◦ •

Introduction

The XOR_2 function block is a generic function block for calculating the bitwise Boolean XOR operation. It performs an exclusive OR operation between two input values and outputs the result. The function block is suitable for all ANY_BIT data types, thus enabling flexible use with different bit lengths.



Interface Structure

Event Inputs

- **REQ**: Normal execution request - triggers the calculation of the XOR operation

Event Outputs

- **CNF**: Execution confirmation - signals the completion of the calculation

Data Inputs

- **IN1** (ANY_BIT): XOR input 1 - first operand for the XOR operation
- **IN2** (ANY_BIT): XOR input 2 - second operand for the XOR operation

Data Outputs

- **OUT** (ANY_BIT): XOR result - result of the bitwise XOR operation

Adapters

The function block does not have any adapter interfaces.

Functionality

Upon receiving the REQ event, the XOR_2 block performs a bitwise XOR operation between the two input values IN1 and IN2. The operation is performed separately for each bit pair of input values:

- If both bits are the same (both 0 or both 1), the result bit is set to 0.
- If the bits are different, the result bit is set to 1.

After the calculation is complete, the result is output via the OUT output, and the CNF event is sent to confirm execution.

- If the bits are different, the result bit is set to 1.

Technical Features

- **Generic Implementation**: Supports all ANY_BIT data types (BOOL, BYTE, WORD, DWORD, LWORD)
- **Bitwise Operation**: Performs the XOR operation at the bit level
- **Event-Driven Execution**: Synchronization via REQ/CNF events
- **Type Safety**: Guarantees correct operation for all supported data types

State Transitions

1. **Wait State**: Block waits for REQ event
2. **Compute State**: On REQ event, block performs XOR operation
3. **Output State**: After computation, OUT is set and a CNF event is sent
4. **Return**: Block returns to wait state

Application Scenarios

- **Bit Manipulation**: Reversing specific bits in data words
- **Parity Checking**: Creating parity bits for error detection
- **Data Encryption**: Simple cryptographic operations

- **Control Engineering**: Logical operations in automation systems
- **Data Comparison**: Detection of differences between data values

⚖️ Comparison with similar blocks

Compared to other logical blocks, XOR_2 offers:

- **Flexibility**: Support for various bit lengths through the ANY_BIT type
- **Specific Logic**: Exclusive OR operation as opposed to AND/OR operations
- **Generic Implementation**: Reusability across different data types

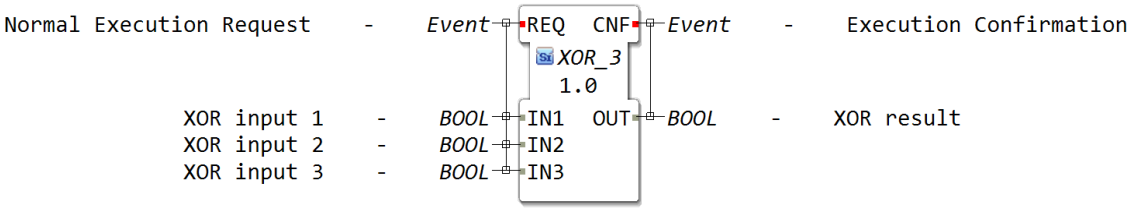
Conclusion

The XOR_2 function block provides an efficient and flexible solution for bitwise XOR operations in 4diac systems. Its generic nature allows for use in a wide range of applications, while its event-driven architecture ensures reliable integration into complex control systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

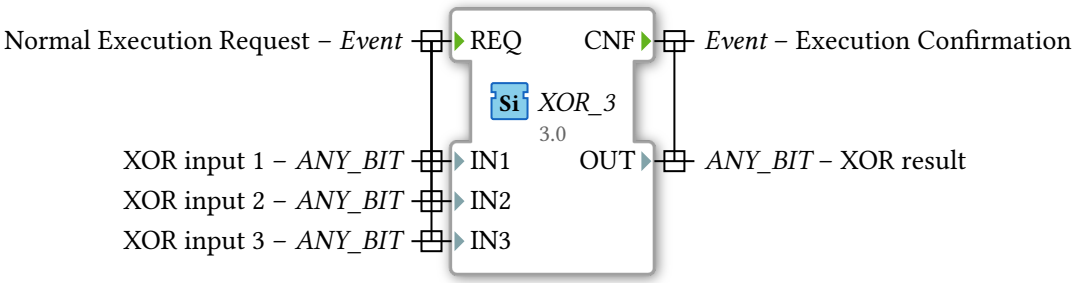
XOR_3



• – ◦ • – ◦ • – ◦ •

Introduction

The XOR_3 function block is a generic function block for calculating bitwise XOR operations with three inputs. It implements the logical exclusive OR operation at the bit level according to the IEC 61131-3 standard and can be used with various bit data types.



Interface Structure

Event Inputs

- **REQ**: Normal Execution Request - triggers the calculation of the XOR operation

Event Outputs

- **CNF**: Execution Acknowledgement - signals the completion of the calculation

Data Inputs

- **IN1**: XOR Input 1 (ANY_BIT data type)
- **IN2**: XOR Input 2 (ANY_BIT data type)
- **IN3**: XOR Input 3 (ANY_BIT data type)

Data Outputs

- **OUT**: XOR Result (ANY_BIT data type)

Adapters

The function block does not use adapters.

Functionality

Upon arrival of the REQ event, the function block performs a bitwise XOR operation on the three input values IN1, IN2, and IN3. The operation is executed separately for each bit position and corresponds to the logical function: OUT = IN1 XOR IN2 XOR IN3. After completion of the calculation, the CNF event is output along with the result value.

Technical Features

- Uses the generic data type ANY_BIT, which is compatible with various bit data types (BOOL, BYTE, WORD, DWORD, LWORD)
- Implemented as a generic function block with the class 'GEN_XOR'
- Supports exactly three inputs for the XOR operation
- Conforms to the IEC 61131-3 classification for standardized bitwise Boolean functions

State Overview

The function block operates statelessly:

1. Wait state: No active calculation
2. Calculation state: The XOR operation is executed upon a REQ event
3. Output state: A CNF event with the result is output

Application Scenarios

- Bitwise encryption operations
- Parity checks in communication protocols
- Error detection in digital systems
- Control logic with multiple conditions
- Data processing in industrial automation systems

⚖️ Comparison with Similar Function Blocks

Compared to XOR_3 offers the ability to process three inputs simultaneously, unlike standard two-input XOR blocks. Compared to AND/OR operations, XOR delivers an exclusive result where the output bit is only 1 if an odd number of the input bits are 1.

🛠️ Related Exercises

- Exercise_002a7_AX

Conclusion

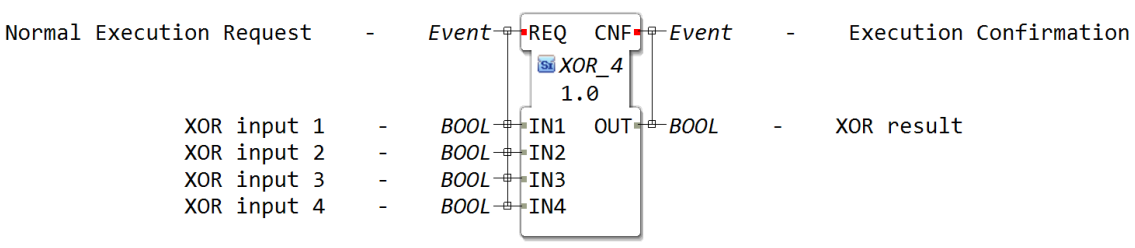
The XOR_3 function block provides an efficient and flexible solution for bitwise XOR operations with three inputs in 4diac systems. Its generic nature allows its use with various bit data types, making it versatile for industrial automation applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

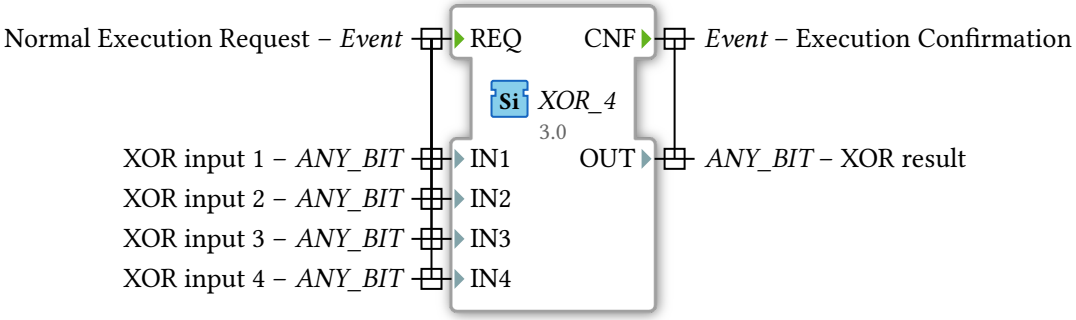
XOR_4



• – ◦ • – ◦ • – ◦ •

Introduction

The XOR_4 function block is a generic function block for calculating bitwise XOR operations with four inputs. It implements the exclusive OR operation at the bit level according to the IEC 61131-3 standard and can work with various bit data types.



Interface Structure

Event Inputs

- **REQ**: Normal execution request - triggers the calculation of the XOR operation

Event Outputs

- **CNF**: Execution confirmation - signals the completion of the calculation

Data Inputs

- **IN1**: XOR input 1 (ANY_BIT data type)
- **IN2**: XOR input 2 (ANY_BIT data type)
- **IN3**: XOR input 3 (ANY_BIT data type)
- **IN4**: XOR input 4 (ANY_BIT data type)

Data Outputs

- **OUT**: XOR result (ANY_BIT data type)

Adapters

The function block does not use adapters.

Functionality

Upon arrival of the REQ event, the XOR_4 block performs a bitwise XOR operation on all four inputs. The operation is executed according to the following logic:

OUT = IN1 XOR IN2 XOR IN3 XOR IN4 Once the calculation is complete, the CNF event, along with the result, is output to OUT. ...

Technical Features

- **Generic Data Type**: Uses ANY_BIT, enabling use with various bit data types (BOOL, BYTE, WORD, DWORD, LWORD)
- **Four-Input XOR Operation**: Processes four inputs in a single operation
- **Event-Driven**: Execution is triggered by the REQ event
- **Standard-Compliant**: Conforms to the IEC 61131-3 standard for bitwise Boolean functions

State Transitions

1. **Wait State**: Block waits for REQ event
2. **Compute State**: All inputs are processed upon REQ event
3. **Output State**: CNF event is output with the computed result
4. Returns to Wait State

Application Scenarios

- **Bit Manipulation**: Used in control applications for bitwise operations
- **Parity Checking**: Implementation of parity bits or error detection

- **Data Encryption**: Simple cryptographic operations
- **Signal Processing**: Combining multiple digital signals

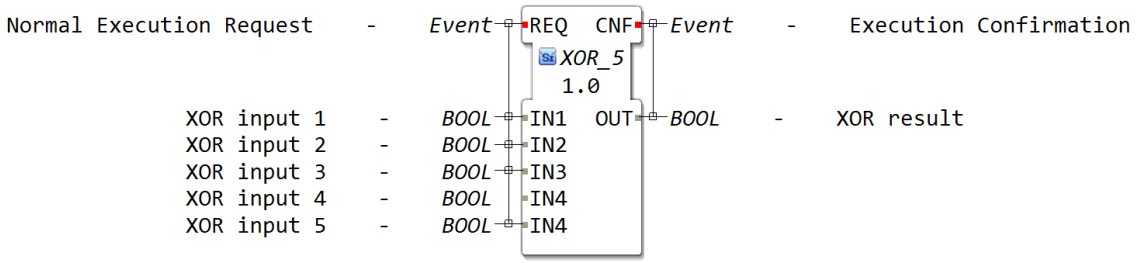
⚖️ Comparison with Similar Blocks

- **AND/OR Blocks**: Process logical AND/OR operations instead of exclusive OR
- **XOR with Fewer Inputs**: This block offers four inputs compared to standard XOR blocks, which typically have two inputs
- **Generic vs. Specific Blocks**: Can process various bit data types, while specific blocks are limited to one data type

Conclusion

The XOR_4 function block offers a flexible and standards-compliant solution for four-input bitwise XOR operations. Its generic nature allows for use in various application areas, while its event-driven architecture ensures efficient integration into control systems.

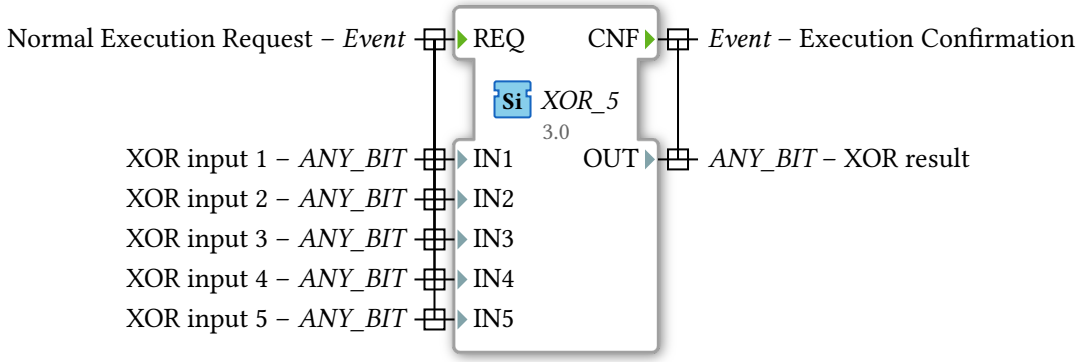
XOR_5



• – ◦ • – ◦ • – ◦ •

Introduction

The XOR_5 function block is a generic function block for calculating the bitwise Boolean XOR operation with five inputs. It implements the exclusive OR operation for arbitrary bit data types according to the IEC 61131-3 standard.



Interface Structure

Event Inputs

- **REQ**: Normal Execution Request - triggers the calculation of the XOR operation

Event Outputs

- **CNF**: Execution Confirmation - signals the completion of the calculation

Data Inputs

- **IN1**: XOR Input 1 (ANY_BIT data type)
- **IN2**: XOR Input 2 (ANY_BIT data type)
- **IN3**: XOR Input 3 (ANY_BIT data type)
- **IN4**: XOR Input 4 (ANY_BIT data type)
- **IN5**: XOR Input 5 (ANY_BIT data type)

Data Outputs

- **OUT**: XOR Result (ANY_BIT data type)

Adapters

No adapter interfaces are available.

Functionality

Upon arrival of the REQ event, the function block performs a bitwise XOR operation on all five input values. The result is output to the OUT output, and simultaneously, the CNF event is generated to confirm the completion of the calculation.

The XOR operation is performed according to the following principle:

- Each bit position is processed individually.
- The result bit is 1 if an odd number of input bits at that position are 1.
- The result bit is 0 if an even number of input bits at that position are 1.

Technical Features

- Uses the generic ANY_BIT data type, which is compatible with various bit data types (BOOL, BYTE, WORD, DWORD, LWORD).
- Supports the processing of five input values.
- Implemented as a generic function block according to the IEC 61131-3 standard.
- Package mapping: iec61131::bitwiseOperators

State Overview

The function block has a simple state machine:

1. **Wait State**: Waiting for a REQ event.
2. **Compute State**: Processing the XOR operation.
3. **Output State**: Setting the OUT value and triggering the CNF Event

Application Scenarios

- Bitwise data processing in control systems
- Parity checks and error detection
- Cryptographic operations
- Data manipulation and transformation
- Signal processing with multiple input signals

⚖️ Comparison with similar components

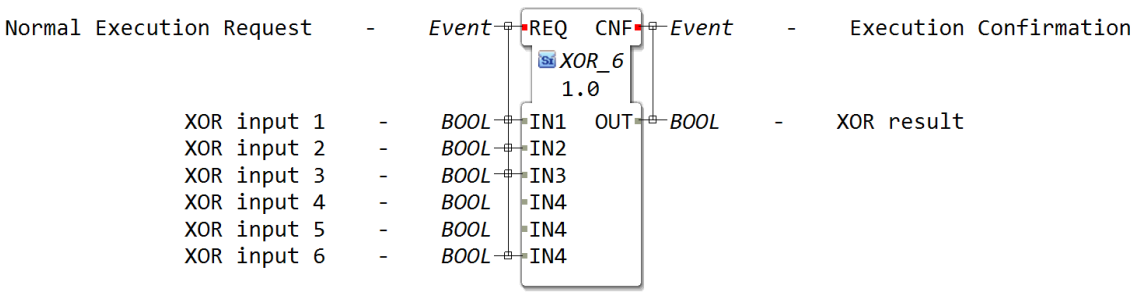
Compared to standard XOR components with fewer inputs, XOR_5 offers:

- Extended functionality with five inputs instead of two
- Greater flexibility for more complex logical operations
- Reduced chaining depth compared to serial XOR operations

Conclusion

The XOR_5 function block provides a powerful and flexible solution for bitwise XOR operations with five inputs. Its generic implementation allows its use with various bit data types, making it a versatile tool for diverse applications in automation technology.

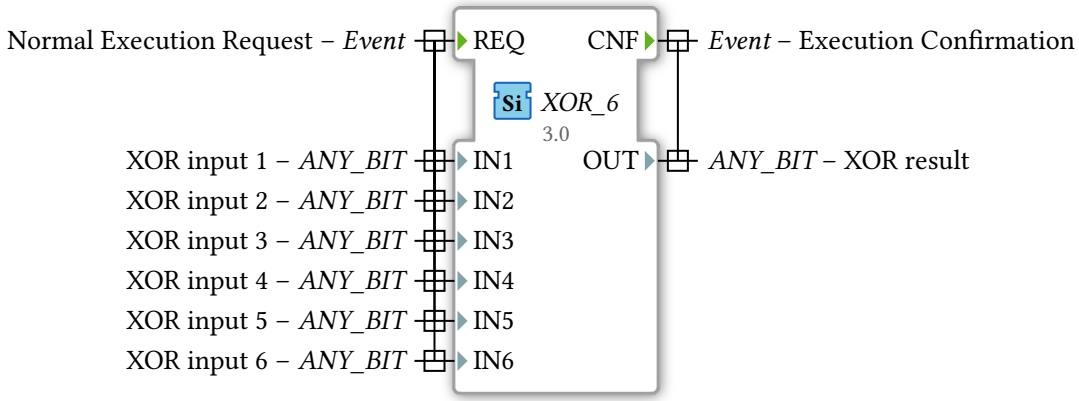
XOR_6



• – ◦ • – ◦ • – ◦ •

Introduction

The XOR_6 function block is a generic function block for calculating bitwise XOR operations with six inputs. It implements the Boolean exclusive OR operation at the bit level according to the IEC 61131-3 standard and can work with various bit data types.



Interface Structure

Event Inputs

- **REQ**: Normal Execution Request - triggers the calculation of the XOR operation

Event Outputs

- **CNF**: Execution Acknowledgement - signals the completion of the calculation

Data Inputs

- **IN1**: XOR Input 1 (ANY_BIT data type)
- **IN2**: XOR Input 2 (ANY_BIT data type)
- **IN3**: XOR Input 3 (ANY_BIT data type)
- **IN4**: XOR Input 4 (ANY_BIT data type)
- **IN5**: XOR Input 5 (ANY_BIT data type)
- **IN6**: XOR Input 6 (ANY_BIT data type)

Data Outputs

- **OUT**: XOR Result (ANY_BIT Data Type)

Adapter

No adapter interfaces are available.

Functionality

The XOR_6 block performs a bitwise XOR operation on all six inputs. When the REQ event is triggered, the values at IN1 to IN6 are processed, and the result is output to OUT. The operation follows the mathematical definition of the XOR function, where the result is true for each bit if an odd number of the corresponding input bits are true.

The calculation is performed according to the formula: OUT = IN1 ⊕ IN2 ⊕ IN3 ⊕ IN4 ⊕ IN5 ⊕ IN6

Technical Features

- **Generic Data Type**: Uses ANY_BIT, enabling compatibility with various bit data types (BOOL, BYTE, WORD, DWORD, LWORD)
- **Six Inputs**: Offers extended functionality compared to standard XOR blocks with fewer inputs
- **Event-Driven Execution**: Follows the 4diac execution model with REQ/CNF event pairs

State Overview

The function block has a simple state machine:

1. **Wait State**: Block waits for a REQ event

2. **Calculate State**: Processes the inputs when a REQ event is triggered
3. **Output State**: Sends a CNF event with the calculated result

Application Scenarios

- **Parity Check**: Checks the Data integrity through XOR operation of multiple data words
- **Data encryption**: Simple cryptographic operations
- **Error detection**: Generation of checksums in communication protocols
- **Control engineering**: Logical operations in automation applications

⚖️ Comparison with similar blocks

Compared to standard two-input XOR blocks, XOR_6 offers:

- Enhanced functionality with six direct inputs
- Elimination of the need to cascade multiple XOR blocks
- More compact implementation for complex XOR operations

Conclusion

The XOR_6 function block provides a powerful and flexible solution for bitwise XOR operations with multiple inputs. Its generic nature allows for broad applicability in various automation projects, while its six-input functionality supports efficient implementations of complex logical operations.

🌐 Related topic subpages on ms-muc-docs.de

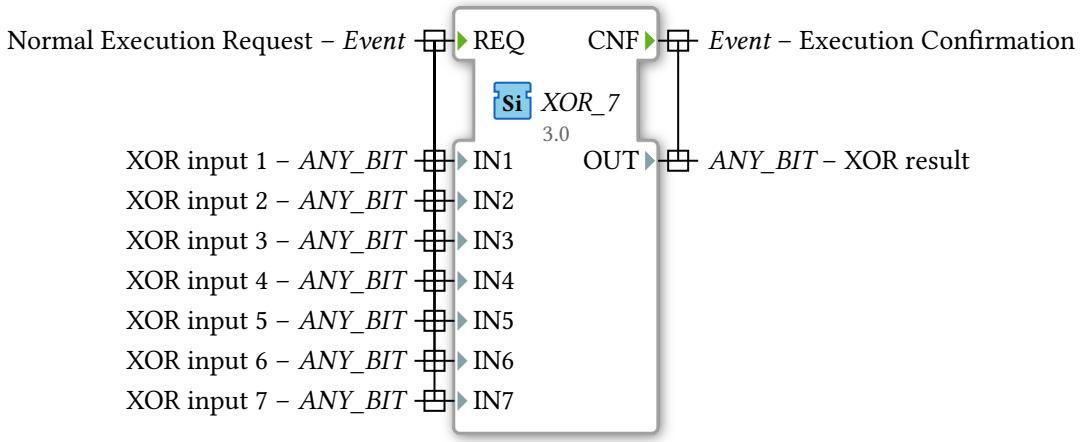
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

XOR_7

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `XOR_7` performs a bitwise XOR operation on seven input values. It is a generic function block that can be used with various bit data types (e.g., BOOL, BYTE, WORD, DWORD, LWORD). The block is classified according to the IEC 61131-3 standard and is used for processing Boolean operations.



Interface Structure

Event Inputs

- **REQ** (Normal Execution Request): Triggers the calculation of the XOR operation. It is linked to all data inputs (`IN1` to `IN7`).

Event Outputs

- **CNF** (Execution Confirmation): Signals successful calculation and outputs the result via the data output `OUT`.

Data Inputs

- **IN1** (XOR input 1): First input value (Type: `ANY_BIT`).
- **IN2** (XOR input 2): Second input value (Type: `ANY_BIT`).
- **IN3** (XOR input 3): Third input value (Type: `ANY_BIT`).
- **IN4** (XOR input 4): Fourth input value (Type: `ANY_BIT`).
- **IN5** (XOR input 5): Fifth input value (Type: `ANY_BIT`).
- **IN6** (XOR input 6): Sixth input value (Type: `ANY_BIT`).
- **IN7** (XOR input 7): Seventh input value (Type: `ANY_BIT`).

Data Outputs

- **OUT** (XOR result): Result of the bitwise XOR operation (Type: `ANY_BIT`).

Adapters

No adapters available.

Functionality

The function block performs a bitwise XOR operation on the seven input values (`IN1` to `IN7`). The operation is triggered by the event `REQ`. The result is output via `OUT` and confirmed by the event `CNF`.

Technical Features

- **Generic Implementation**: Supports various bit data types (`ANY_BIT`).
- **Flexibility**: Can operate with up to seven input values.
- **Standard Compliant**: Complies with the IEC 61131-3 standard.

State Overview

The function block has no internal states. The operation is re-executed with each `REQ` event.

Application Scenarios

- Bitwise encryption.
- Error detection in data transmissions.
- Logical controls in automation systems.

⚖️ Comparison with Similar Function Blocks

- **XOR (Standard)**: Typically offers only two inputs, while `XOR_7` supports seven inputs.
- **AND/OR Blocks**: Perform other logical operations, while `XOR_7` is specifically optimized for the XOR operation.

Conclusion

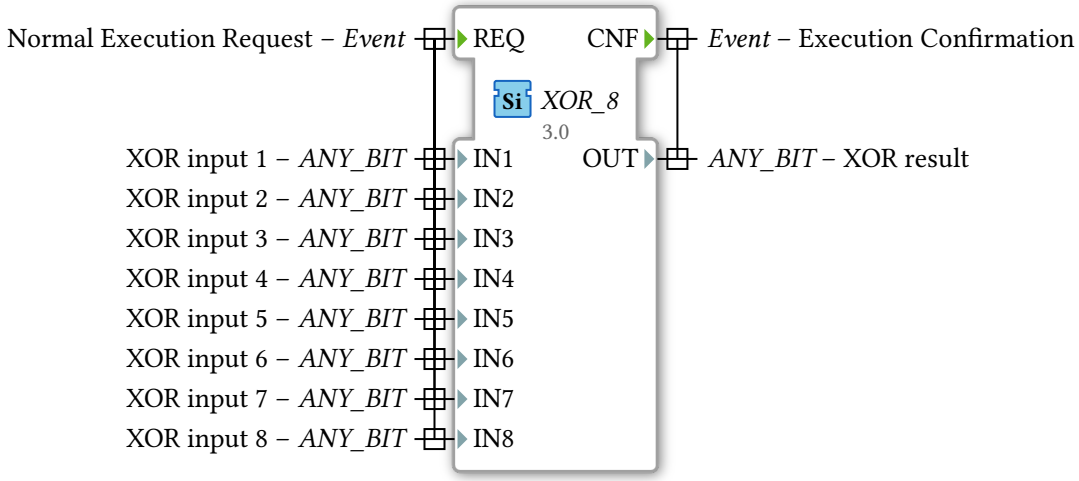
The `XOR_7` function block is a powerful tool for bitwise XOR operations with up to seven input values. Its generic nature and standards compliance make it versatile for use in various automation and control scenarios.

XOR_8

• – ◦ • – ◦ • – ◦ •

Introduction

The XOR_8 function block performs a bitwise XOR operation on up to 8 input variables. It is a generic function block that can work with various bit data types (ANY_BIT). The block is classified according to the IEC 61131-3 standard and provides a simple way to perform XOR operations in control applications.



Interface Structure

Event Inputs

- **REQ** (Normal Execution Request): Starts the calculation of the XOR operation. It is linked to all data inputs.

Event Outputs

- **CNF** (Execution Confirmation): Signals the completion of the calculation. It is linked to the data output OUT.

Data Inputs

- **IN1** (ANY_BIT): XOR Input 1
- **IN2** (ANY_BIT): XOR Input 2
- **IN3** (ANY_BIT): XOR Input 3
- **IN4** (ANY_BIT): XOR Input 4
- **IN5** (ANY_BIT): XOR Input 5
- **IN6** (ANY_BIT): XOR Input 6
- **IN7** (ANY_BIT): XOR Input 7
- **IN8** (ANY_BIT): XOR Input 8

Data Outputs

- **OUT** (ANY_BIT): Result of the XOR operation

Adapters

The function block does not have any adapter interfaces.

Functionality

Upon receiving the REQ event, the function block performs a bitwise XOR operation on all active input variables (IN1 to IN8). The result is output at OUT, and the CNF event is triggered to signal the completion of the operation.

Technical Features

- Supports the generic data type ANY_BIT, meaning that various bit data types (e.g., BOOL, BYTE, WORD, DWORD, LWORD) can be used.
- The operation is strictly bitwise.
- The function block is part of the “iec61131::bitwiseOperators” package.

State Overview

The function block has no complex states. It responds to the REQ event with an immediate calculation and output of the result.

Application Scenarios

- Bitwise Checksum Calculation
- Data Encryption Operations
- Error Detection in Data Transmissions
- Control Logic with Bitwise Operations

⚖️ Comparison with Similar Function Blocks

Compared to simpler XOR blocks (e.g., with only 2 inputs), XOR_8 offers the ability to process up to 8 inputs simultaneously. Other bitwise operators (such as AND_8 or OR_8) perform similar operations with different logical combinations.

Conclusion

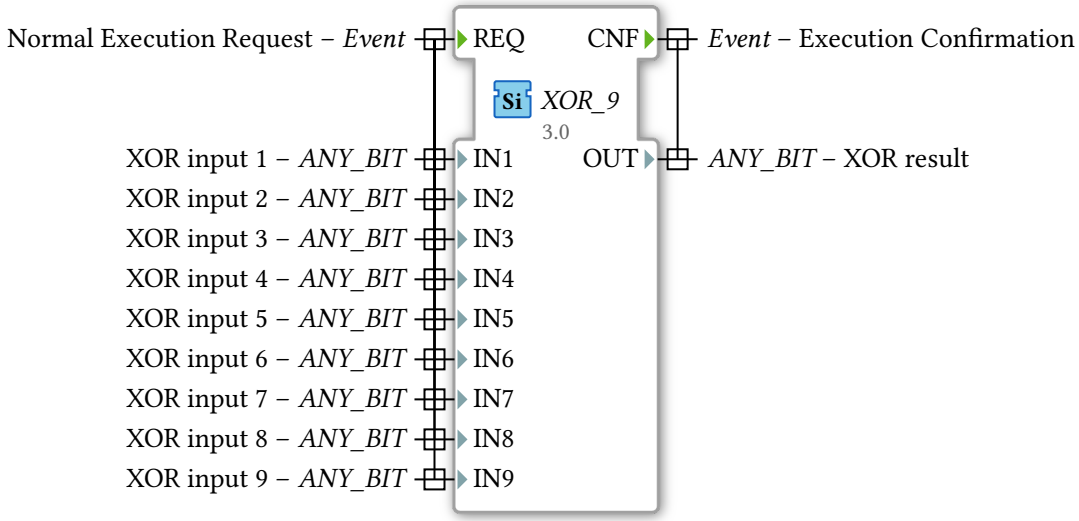
The XOR_8 function block provides a flexible and standards-compliant solution for bitwise XOR operations in control systems. Its generic nature allows its use with various bit data types, while support for up to 8 inputs enables more complex logical operations. Its simple interface and deterministic execution make it a reliable component for control applications.

XOR_9

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `XOR_9` is used for bitwise calculation of the XOR operation with up to nine inputs. It is part of the standard bit operations according to IEC 61131-3 and enables the processing of any bit data type (`ANY_BIT`).



Interface Structure

Event Inputs

- `REQ` (Normal Execution Request): Starts the calculation of the XOR operation. Triggers the evaluation of all combined data inputs.

Event Outputs

- `CNF` (Execution Confirmation): Signals the completion of the calculation and outputs the result via the data output `OUT`.

Data Inputs

- `IN1` to `IN9` (XOR input 1-9): Up to nine inputs for bitwise XOR operation. Each input supports the data type `ANY_BIT` (e.g., BOOL, BYTE, WORD, DWORD, LWORD).

Data Outputs

- `OUT` (XOR result): Result of the bitwise XOR operation of all active inputs. The data type corresponds to that of the inputs (`ANY_BIT`).

Adapters

This function block has no adapters.

Functionality

When the `REQ` event is triggered, the function block calculates the XOR operation of all passed input values (`IN1` to `IN9`). The result is output to `OUT`, and the `CNF` event signals that the result is ready.
Example (for BOOL inputs): OUT = IN1 XOR IN2 XOR ... XOR IN9`

Technical Features

- **Generic Implementation:** Supports all `ANY_BIT` data types by using the generic class `GEN_XOR`.
- **Flexible Number of Inputs:** Unused inputs are ignored (act as neutral elements).
- **Bitwise Operation:** The operation is performed separately for each bit (for multi-bit data types).

State Overview

1. **Idle:** Waits for the `REQ` event.
2. **Processing:** Performs an XOR operation and sets `OUT`.
3. **Ready:** Sends `CNF` and returns to Idle.

Application Scenarios

- Parity checking in communication protocols
- Toggles control signals (e.g., switching between states)
- Basic cryptographic operations
- Error detection in binary data

⚖️ Comparison with similar function blocks

Feature	XOR_9	Standard XOR (2 inputs)
Number of inputs	9	2
Data type	ANY_BIT	Type-dependent (e.g., BOOL)
Flexibility	High	Low

Conclusion

The `XOR_9` function block extends the classic XOR functionality by supporting multiple inputs and generic data types. Ideal for applications that require more complex bitwise operations without having to build individual function block chains. IEC 61131-3 compliance ensures broad compatibility.

2.11.4.5 String Functions

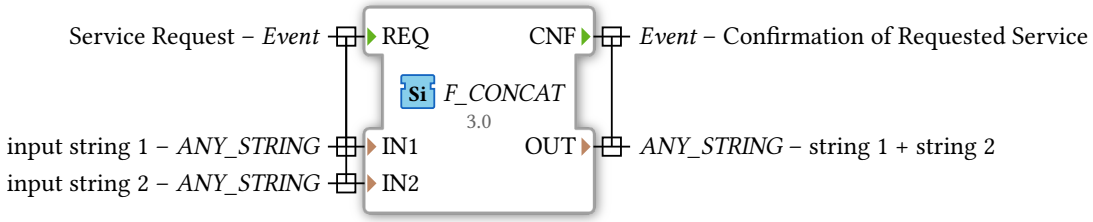
Here you will find functions for manipulating strings.

F_CONCAT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_CONCAT` is used to concatenate two strings. It is part of the standard library for character and string operations according to IEC 61131-3.



Interface Structure

Event Inputs

- `REQ` (Service Request): Starts the concatenation operation. The block requires both input strings (`IN1` and `IN2`) to execute the operation.

Event Outputs

- `CNF` (Confirmation of Requested Service): Signals the successful completion of the concatenation operation. The result is provided via the data output `OUT`.

Data Inputs

- `IN1` (input string 1): First input string of type `ANY_STRING`.
- `IN2` (input string 2): Second input string of type `ANY_STRING`.

Data Outputs

- `OUT` (string 1 + string 2): Result of concatenating `IN1` and `IN2` as `ANY_STRING`.

Adapters

This function block has no adapters.

Functionality

When the event `REQ` is triggered, the two input strings `IN1` and `IN2` are concatenated. The result is output via `OUT`, and the event `CNF` signals the successful completion of the operation.

Technical Features

- Supports the data type `ANY_STRING`, enabling flexible use with various string types.
- Simple and direct implementation without additional parameters or complex state logic.

State Overview

The function block has no internal state. The operation is re-executed with each `REQ` event.

Application Scenarios

- Combining text segments in automated messaging systems.
- Creating dynamic strings in control logic, e.g., for logging or notifications.

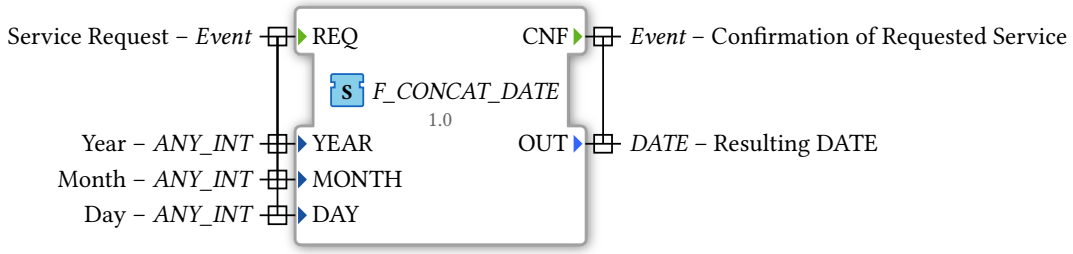
⚖️ Comparison with Similar Blocks

- Unlike `F_STRING_TO_*` blocks, which convert strings, `F_CONCAT` does not perform type conversion.
- Simpler than blocks with additional formatting options like `F_STRING_FORMAT`.

Conclusion

F_CONCAT` is a basic and efficient building block for string processing in IEC 61131-3 environments. Its simplicity and direct functionality make it ideal for basic concatenation tasks.

F_CONCAT_DATE



• – ◦ • – ◦ • – ◦ •

Introduction

`F_CONCAT_DATE` combines individual date/time components (`YEAR`, `MONTH`, `DAY`) into a composite `DATE` value. It is the reverse of F_SPLIT_DATE, which splits the same `DATE` value back into its individual components.

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries `YEAR`, `MONTH`, `DAY`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **YEAR** (ANY_INT): year.
- **MONTH** (ANY_INT): month.
- **DAY** (ANY_INT): day.

Data Outputs

- **OUT** (`DATE`): The value composed from the individual components.

Functionality

On `REQ`, the input values `YEAR`, `MONTH`, `DAY` are combined into a `DATE` value and output via `OUT`. `CNF` is then triggered.

Technical Features

- **`ANY_INT` inputs**: The date/time components accept any integer type, simplifying wiring with differently typed source values.
- **No documented range checking**: The block assumes plausible input values (e.g. `MONTH` 1–12); explicit validation is the caller's responsibility.

State Overview

Stateless: every `REQ` immediately results in the merge and `CNF`.

Application Scenarios

- **Building timestamps** from separately captured or computed individual values, e.g. from sensor data, user input, or communication protocols.
- **Configuration evaluation**: Combining date/time components held in individual variables into a usable `DATE` value.

Comparison with similar function blocks

- F_SPLIT_DATE: the reverse direction — splits a `DATE` value into its individual components.
- **`F_CONCAT_DATE_TOD`**: instead combines an already-complete `DATE` and `TIME_OF_DAY` value into `DATE_AND_TIME`, rather than from individual fields.

Conclusion

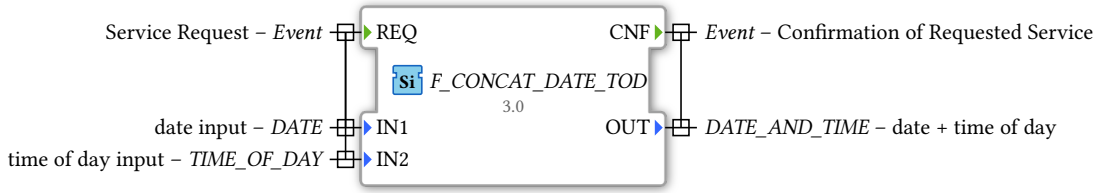
F_CONCAT_DATE provides a simple, direct merge of individual date/time values into a DATE value, complementing the corresponding split function F_SPLIT_DATE.

F_CONCAT_DATE_TOD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_CONCAT_DATE_TOD concatenates a date (DATE) and a time of day (TIME_OF_DAY) into a combined date and time stamp (DATE_AND_TIME). This block is particularly useful in scenarios where separate date and time information needs to be combined into a single timestamp.



Interface Structure

Event Inputs

- **REQ**: Starts the processing of the function block. When this event is triggered, the input data IN1 (date) and IN2 (time of day) are processed.

Event Outputs

- **CNF**: Signals successful processing and outputs the combined timestamp OUT.

Data Inputs

- **IN1** (DATE): Input for the date.
- **IN2** (TIME_OF_DAY): Input for the time of day.

Data Outputs

- **OUT** (DATE_AND_TIME): The combined date and time stamp generated from the inputs IN1 and IN2.

Adapters

This function block has no adapters.

Functionality

When the event REQ is triggered, the input data IN1 (date) and IN2 (time of day) are read and combined into a value DATE_AND_TIME. The result is output via the variable OUT, and the event CNF signals successful processing.

Technical Features

- The function block supports the standardized data types DATE, TIME_OF_DAY, and DATE_AND_TIME according to IEC 61131-3.
- Processing occurs synchronously with the triggering of the REQ event.

State Overview

1. **Idle**: Waits for the REQ event.
2. **Processing**: Processes the input data and combines it into a DATE_AND_TIME value.
3. **Output**: Outputs the result via OUT and signals CNF.

Application Scenarios

- Combining separately recorded date and time information in control systems.
- Generating timestamps for logging and monitoring purposes.

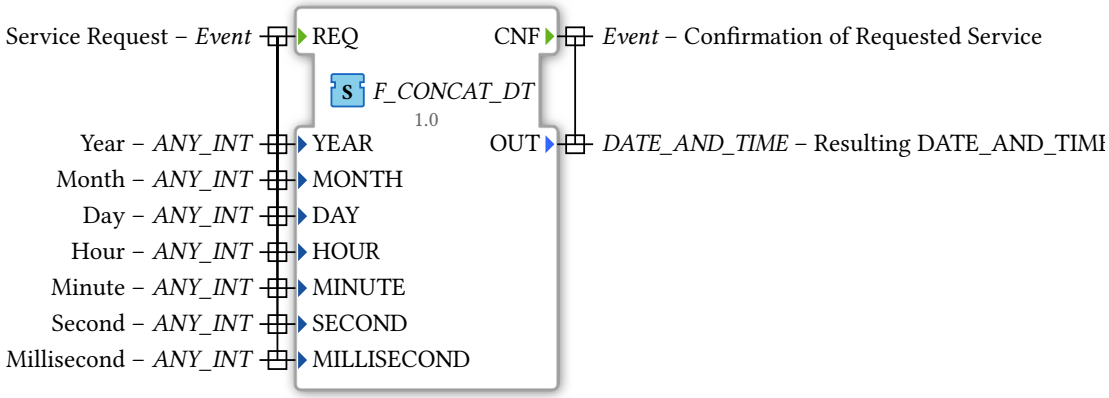
⚖️ Comparison with Similar Function Blocks

Compared to other function blocks offering similar functionality, `F_CONCAT_DATE_TOD` stands out due to its simplicity and direct processing. There are no additional parameters or complex configuration options, making it particularly suitable for simple use cases.

Conclusion

The `F_CONCAT_DATE_TOD` function block is an efficient tool for combining date and time information in control systems. Its ease of use and clear interface structure make it a reliable solution for basic timestamp operations.

F_CONCAT_DT



• – ◦ • – ◦ • – ◦ •

Introduction

`F_CONCAT_DT` combines individual date/time components (`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`) into a composite `DATE_AND_TIME` value. It is the reverse of F_SPLIT_DT, which splits the same `DATE_AND_TIME` value back into its individual components.

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **YEAR** (ANY_INT): year.
- **MONTH** (ANY_INT): month.
- **DAY** (ANY_INT): day.
- **HOUR** (ANY_INT): hour.
- **MINUTE** (ANY_INT): minute.
- **SECOND** (ANY_INT): second.
- **MILLISECOND** (ANY_INT): millisecond.

Data Outputs

- **OUT** (`DATE_AND_TIME`): The value composed from the individual components.

Functionality

On `REQ`, the input values `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` are combined into a `DATE_AND_TIME` value and output via `OUT`. `CNF` is then triggered.

Technical Features

- **`ANY_INT` inputs**: The date/time components accept any integer type, simplifying wiring with differently typed source values.
- **No documented range checking**: The block assumes plausible input values (e.g. `MONTH` 1–12); explicit validation is the caller's responsibility.

State Overview

Stateless: every `REQ` immediately results in the merge and `CNF`.

Application Scenarios

- **Building timestamps** from separately captured or computed individual values, e.g. from sensor data, user input, or communication protocols.
- **Configuration evaluation**: Combining date/time components held in individual variables into a usable `DATE_AND_TIME` value.

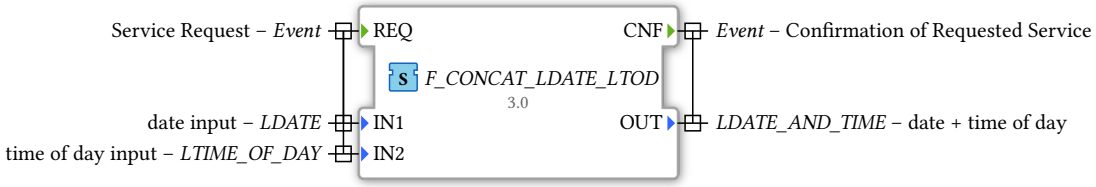
Comparison with similar function blocks

- F_SPLIT_DT: the reverse direction — splits a `DATE_AND_TIME` value into its individual components.
- **`F_CONCAT_DATE_TOD`**: instead combines an already-complete `DATE` and `TIME_OF_DAY` value into `DATE_AND_TIME`, rather than from individual fields.

Conclusion

`F_CONCAT_DT` provides a simple, direct merge of individual date/time values into a `DATE_AND_TIME` value, complementing the corresponding split function F_SPLIT_DT.

F_CONCAT_LDATE_LTOD



• – ◦ • – ◦ • – ◦ •

Introduction

`F_CONCAT_LDATE_LTOD` concatenates a date (`LDATE`) and a time of day (`LTIME_OF_DAY`) into a combined `LDATE_AND_TIME` timestamp. It is the `L`-prefixed (64-bit resolution) counterpart to `F_CONCAT_DATE_TOD`.

Interface Structure

Event Inputs

- **REQ**: Triggers the concatenation, carries `IN1` and `IN2`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **IN1** (LDATE): The date.
- **IN2** (LTIME_OF_DAY): The time of day.

Data Outputs

- **OUT** (LDATE_AND_TIME): The combined date and time timestamp.

Functionality

On `REQ`, `IN1` (date) and `IN2` (time of day) are combined into an `LDATE_AND_TIME` value and output via `OUT`. `CNF` is then triggered.

Technical Features

- **`L` data types**: Uses the high-resolution `LDATE`/`LTIME_OF_DAY`/`LDATE_AND_TIME` types with nanosecond resolution instead of the classic `DATE`/`TIME_OF_DAY`/`DATE_AND_TIME` types.

State Overview

Stateless: every `REQ` immediately results in the concatenation and `CNF`.

Application Scenarios

- **Combining separately captured, high-resolution date and time information** in control systems that work with `L` time types throughout.

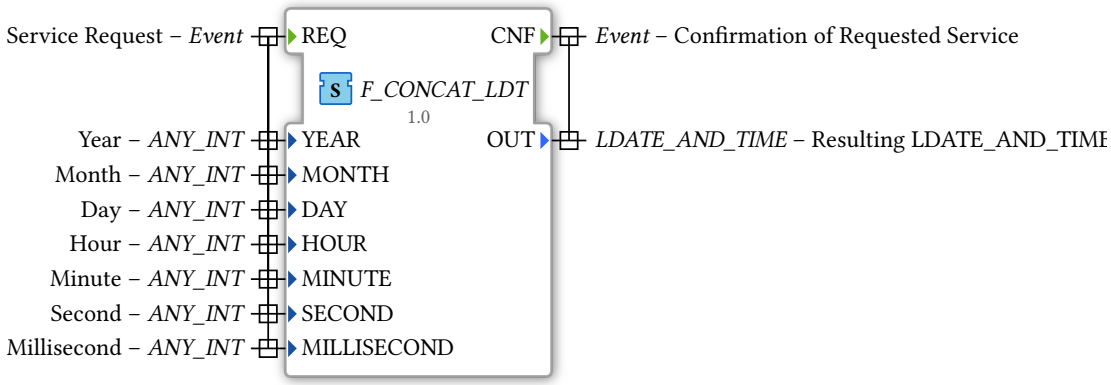
Comparison with similar function blocks

- **`F_CONCAT_DATE_TOD`**: the same functionality with the classic `DATE`/`TIME_OF_DAY`/`DATE_AND_TIME` types.
- F_CONCAT_LDT: instead combines individual fields (`YEAR`, `MONTH`, ...) rather than already-complete `LDATE`/`LTIME_OF_DAY` values.

Conclusion

`F_CONCAT_LDATE_LTOD` provides the high-resolution variant of date/time concatenation and is suitable for applications working with `L` time types throughout.

F_CONCAT_LDT



• – ◦ • – ◦ • – ◦ •

Introduction

`F_CONCAT_LDT` combines individual date/time components (`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`) into a composite `LDATE_AND_TIME` value. It is the reverse of F_SPLIT_LDT, which splits the same `LDATE_AND_TIME` value back into its individual components.

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **YEAR** (ANY_INT): year.
- **MONTH** (ANY_INT): month.
- **DAY** (ANY_INT): day.
- **HOUR** (ANY_INT): hour.
- **MINUTE** (ANY_INT): minute.
- **SECOND** (ANY_INT): second.
- **MILLISECOND** (ANY_INT): millisecond.

Data Outputs

- **OUT** (`LDATE_AND_TIME`): The value composed from the individual components.

Functionality

On `REQ`, the input values `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` are combined into a `LDATE_AND_TIME` value and output via `OUT`. `CNF` is then triggered.

Technical Features

- **`ANY_INT` inputs**: The date/time components accept any integer type, simplifying wiring with differently typed source values.
- **No documented range checking**: The block assumes plausible input values (e.g. `MONTH` 1–12); explicit validation is the caller's responsibility.

State Overview

Stateless: every `REQ` immediately results in the merge and `CNF`.

Application Scenarios

- **Building timestamps** from separately captured or computed individual values, e.g. from sensor data, user input, or communication protocols.
- **Configuration evaluation**: Combining date/time components held in individual variables into a usable `LDATE_AND_TIME` value.

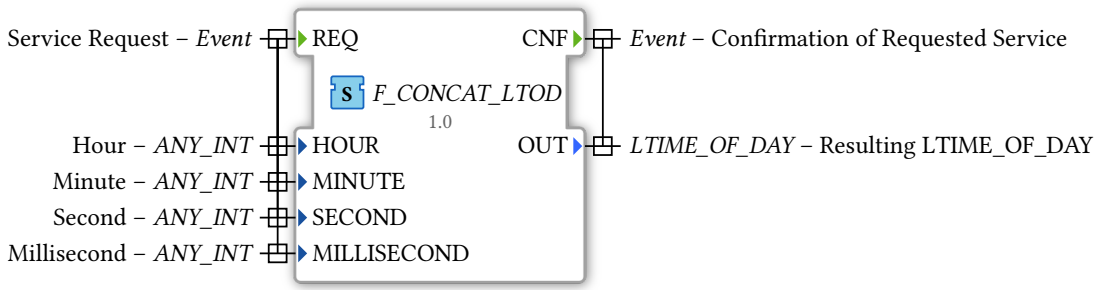
Comparison with similar function blocks

- F_SPLIT_LDT: the reverse direction — splits a `LDATE_AND_TIME` value into its individual components.
- **`F_CONCAT_DATE_TOD`**: instead combines an already-complete `DATE` and `TIME_OF_DAY` value into `DATE_AND_TIME`, rather than from individual fields.

Conclusion

`F_CONCAT_LDT` provides a simple, direct merge of individual date/time values into a `LDATE_AND_TIME` value, complementing the corresponding split function F_SPLIT_LDT.

F_CONCAT_LTOD



• – ◦ • – ◦ • – ◦ •

Introduction

`F_CONCAT_LTOD` combines individual date/time components (`HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`) into a composite `LTIME_OF_DAY` value. It is the reverse of F_SPLIT_LTOD, which splits the same `LTIME_OF_DAY` value back into its individual components.

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **HOUR** (ANY_INT): hour.
- **MINUTE** (ANY_INT): minute.
- **SECOND** (ANY_INT): second.
- **MILLISECOND** (ANY_INT): millisecond.

Data Outputs

- **OUT** (`LTIME_OF_DAY`): The value composed from the individual components.

Functionality

On `REQ`, the input values `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` are combined into a `LTIME_OF_DAY` value and output via `OUT`. `CNF` is then triggered.

Technical Features

- **`ANY_INT` inputs**: The date/time components accept any integer type, simplifying wiring with differently typed source values.
- **No documented range checking**: The block assumes plausible input values (e.g. `MONTH` 1–12); explicit validation is the caller's responsibility.

State Overview

Stateless: every `REQ` immediately results in the merge and `CNF`.

Application Scenarios

- **Building timestamps** from separately captured or computed individual values, e.g. from sensor data, user input, or communication protocols.
- **Configuration evaluation**: Combining date/time components held in individual variables into a usable `LTIME_OF_DAY` value.

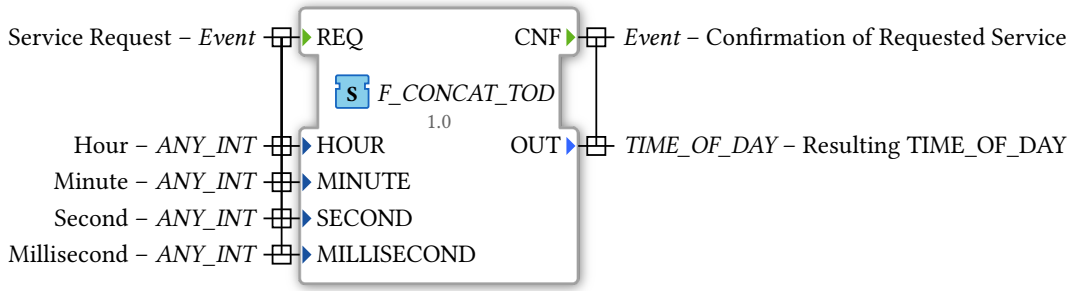
Comparison with similar function blocks

- F_SPLIT_LTOD: the reverse direction — splits a `LTIME_OF_DAY` value into its individual components.
- **`F_CONCAT_DATE_TOD`**: instead combines an already-complete `DATE` and `TIME_OF_DAY` value into `DATE_AND_TIME`, rather than from individual fields.

Conclusion

`F_CONCAT_LTOD` provides a simple, direct merge of individual date/time values into a `LTIME_OF_DAY` value, complementing the corresponding split function F_SPLIT_LTOD.

F_CONCAT_TOD



• – ◦ • – ◦ • – ◦ •

Introduction

`F_CONCAT_TOD` combines individual date/time components (`HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`) into a composite `TIME_OF_DAY` value. It is the reverse of F_SPLIT_TOD, which splits the same `TIME_OF_DAY` value back into its individual components.

Interface Structure

Event Inputs

- **REQ**: Triggers the merge, carries `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **HOUR** (ANY_INT): hour.
- **MINUTE** (ANY_INT): minute.
- **SECOND** (ANY_INT): second.
- **MILLISECOND** (ANY_INT): millisecond.

Data Outputs

- **OUT** (`TIME_OF_DAY`): The value composed from the individual components.

Functionality

On `REQ`, the input values `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND` are combined into a `TIME_OF_DAY` value and output via `OUT`. `CNF` is then triggered.

Technical Features

- **`ANY_INT` inputs**: The date/time components accept any integer type, simplifying wiring with differently typed source values.
- **No documented range checking**: The block assumes plausible input values (e.g. `MONTH` 1–12); explicit validation is the caller's responsibility.

State Overview

Stateless: every `REQ` immediately results in the merge and `CNF`.

Application Scenarios

- **Building timestamps** from separately captured or computed individual values, e.g. from sensor data, user input, or communication protocols.
- **Configuration evaluation**: Combining date/time components held in individual variables into a usable `TIME_OF_DAY` value.

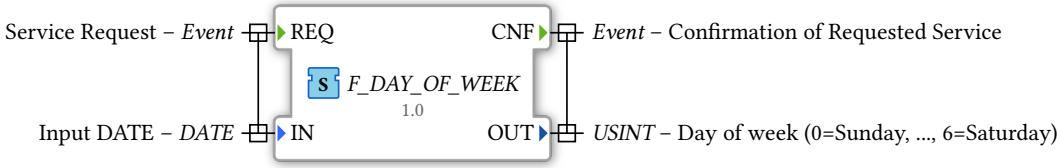
Comparison with similar function blocks

- F_SPLIT_TOD: the reverse direction — splits a `TIME_OF_DAY` value into its individual components.
- **`F_CONCAT_DATE_TOD`**: instead combines an already-complete `DATE` and `TIME_OF_DAY` value into `DATE_AND_TIME`, rather than from individual fields.

Conclusion

`F_CONCAT_TOD` provides a simple, direct merge of individual date/time values into a `TIME_OF_DAY` value, complementing the corresponding split function F_SPLIT_TOD.

F_DAY_OF_WEEK



• – ◦ • – ◦ • – ◦ •

Introduction

`F_DAY_OF_WEEK` determines the day of the week, as a number, from a date (`DATE`). It is a simple calendar-calculation function block per IEC 61131-3.

Interface Structure

Event Inputs

- **REQ**: Triggers the calculation, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **IN** (DATE): The input date.

Data Outputs

- **OUT** (USINT): The day of the week (`0` = Sunday, `1` = Monday, ..., `6` = Saturday).

Functionality

On `REQ`, the algorithm's ST statement `OUT := DAY_OF_WEEK(IN)` computes the day of the week for `IN` and outputs it via `OUT`. `CNF` is then triggered.

Technical Features

- **`0–6` encoding**: Sunday is `0`, Saturday is `6` — this encoding should be kept in mind when comparing against other, possibly Monday-based day-of-week conventions.

State Overview

Stateless: every `REQ` immediately results in the calculation and `CNF`.

Application Scenarios

- **Day-of-week-dependent control logic**: e.g. different operating programs on weekdays vs. weekends.
- **Display/logging**: Outputting the day of the week alongside a date in displays or log entries.
- **Scheduling**: Checking whether a computed date falls on a particular day of the week.

Comparison with similar function blocks

- F_SPLIT_DATE: splits a date into year/month/day, but does not compute a day of the week.

Conclusion

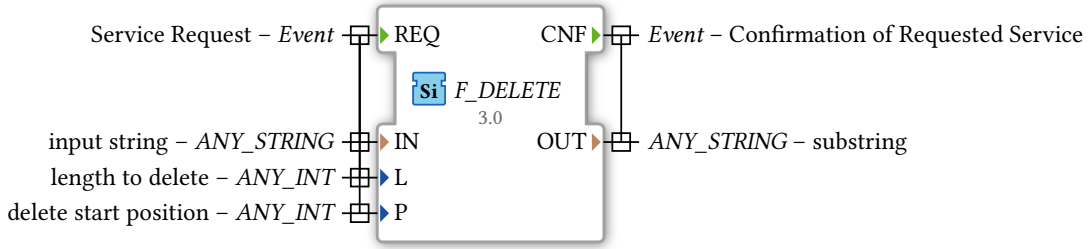
`F_DAY_OF_WEEK` provides a simple calendar calculation to determine the day of the week from a `DATE` value and is suitable for day-of-week-dependent control and display logic.

F_DELETE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DELETE` is used to manipulate strings. It removes a specific number of characters (`L`) from an input string (`IN`), starting at a defined position (`P`). The result is output as a new string (`OUT`).



Interface Structure

Event Inputs

- **REQ**: Starts the processing of the function block. The corresponding data are `IN`, `L`, and `P`.

Event Outputs

- **CNF**: Signals successful processing and returns the result `OUT`.

Data Inputs

- **IN**: The input string from which characters are to be removed (Type: `ANY_STRING`).
- **L**: The number of characters to be removed (Type: `ANY_INT`).
- **P**: The starting position from which characters are to be removed (Type: `ANY_INT`).

Data Outputs

- **OUT**: The resulting string after character removal (Type: `ANY_STRING`).

Adapters

No adapters available.

Functionality

1. The function block is activated by the event `REQ`.
2. Characters `L` are removed from the string `IN`, starting at position `P`.
3. The result is stored in `OUT`, and the event `CNF` is triggered.

Technical Features

- The function block supports various string types (`ANY_STRING`), allowing for flexible use.
- The position `P` and the length `L` must be valid values within the length of `IN` to ensure correct results.

State Overview

1. **Idle**: Waits for the event `REQ`.
2. **Processing**: Processes the input data and removes the characters.
3. **Completed**: Outputs the result and signals `CNF`.

Application Scenarios

- Cleansing user input.
- Processing protocol or log data.
- Manipulating text data in automated processes.

⚖️ Comparison with Similar Building Blocks

- **F_INSERT**: Inserts characters at a specific position.
- **F_REPLACE**: Replaces characters in a string.
- **F_SUBSTRING**: Extracts a substring from a string.

Conclusion

The `F_DELETE` function block is a useful tool for manipulating strings in the 4diac IDE. Its simple and flexible interface allows it to be used in various applications to remove characters from strings.

🌐 Related topic subpages on ms-muc-docs.de

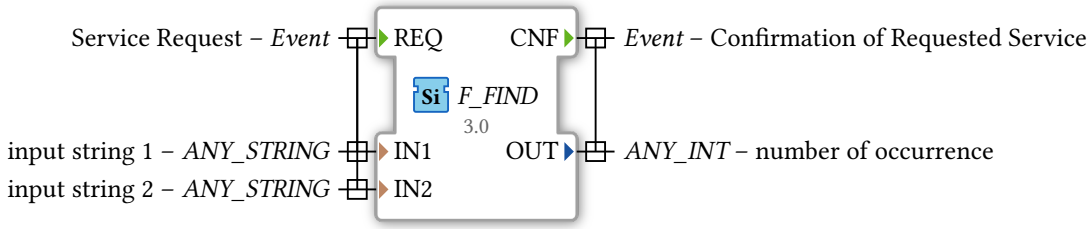
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

F_FIND

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_FIND` determines the position of the first occurrence of a string (IN2) within another string (IN1). If the searched string is not found, the block returns the value 0.



Interface Structure

Event Inputs

- **REQ**: Starts the search for the string. Inputs IN1 and IN2 are linked to this event.

Event Outputs

- **CNF**: Confirms the completion of the search operation. Output OUT is linked to this event.

Data Inputs

- **IN1** (ANY_STRING): The string to be searched.
- **IN2** (ANY_STRING): The string to search for.

Data Outputs

- **OUT** (ANY_INT): The position of the first occurrence of IN2 in IN1. If not found, 0 is returned.

Adapters

No adapters are present.

Functionality

The function block `F_FIND` searches the string IN1 for the first occurrence of the string IN2. The position of the first character of the found string is returned as the result. If IN2 is not contained in IN1, 0 is returned.

Technical Features

- Supports any string type (ANY_STRING).
- Returns any integer type (ANY_INT).
- The search is case-sensitive.

State Overview

1. **Idle**: Waiting for the REQ event.
2. **Processing**: Searches IN1 for IN2.
3. **Completed**: Sends the CNF event with the result.

Application Scenarios

- Searching for substrings in text.
- Validating input data.
- Processing strings in automation processes.

⚖️ Comparison with Similar Blocks

- Compared to `F_FIND_STRING`, `F_FIND` offers similar functionality but is specifically for searching for the first position of a substring.
- Other blocks, such as `F_MID` or `F_LEFT`, offer additional string manipulation capabilities but lack a search function.

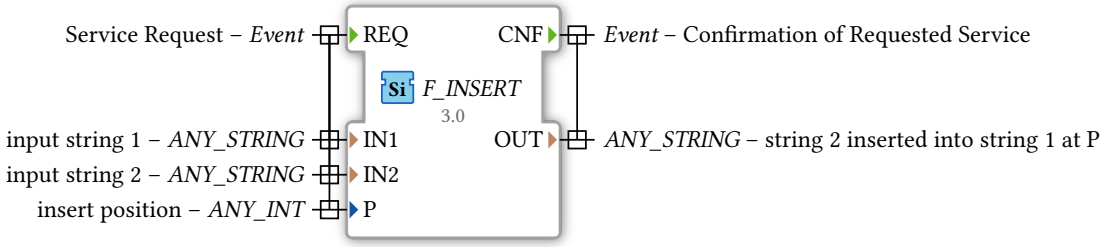
Conclusion

The `F_FIND` function block is a useful tool for searching for substrings in automation applications. Its simple and efficient implementation makes it ideal for integration into more complex control logic.

F_INSERT

• – ◦ • – ◦ • – ◦ •

The function block F_INSERT is used to manipulate character strings. It inserts one character string (IN2) into another character string (IN1) at a specific position (P). The block is part of the standard character and string functions according to IEC 61131-3.



- REQ: Service request. Triggers the execution of the function block. Linked to the data inputs IN1, IN2, and P.
- CNF: Confirmation of the requested service. Linked to the data output OUT.
- IN1 (ANY_STRING): The first input string into which IN2 is inserted.
- P (ANY_INT): The position in IN1 where IN2 is to be inserted.
- OUT (ANY_STRING): The resulting string after the insertion operation.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters available.

When the event REQ is triggered, the string IN2 is inserted into the string IN1 at the position defined by P. The result of this operation is output via the data output OUT, and the event CNF is triggered to indicate successful processing.

- Supports all string types (ANY_STRING), allowing for flexible use.
- The position P can be any integer value (ANY_INT), but must be within the length of IN1 to ensure defined behavior.

The function block has no internal state. The operation is re-executed with each REQ event.

- Text processing in automation applications.
- Dynamic creation of log messages or notifications.
- Manipulation of data strings before further processing or output.
- Similar to F_CONCAT, which, however, simply concatenates two strings without considering a specific insertion position.

Unlike F_MID, which extracts parts of a string, F_INSERT inserts one string into another.

`F_INSERT`` is a useful function block for applications that require precise string manipulation. Its simple yet effective functionality makes it a valuable tool in automation engineering.

Functionality

Application Scenarios

State Overview

Technical Features

⚖️ Vergleich mit ähnlichen Bausteinen

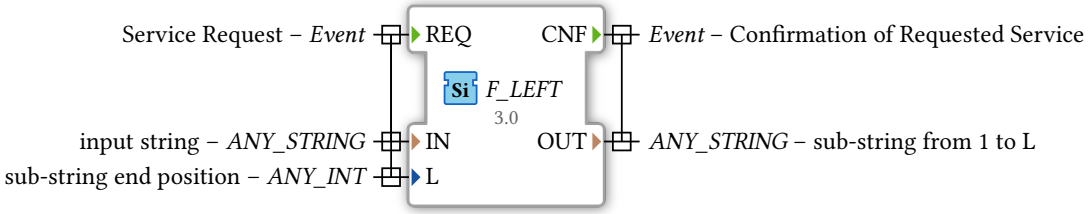
Conclusion

F_LEFT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LEFT` extracts the leftmost characters `L` from the input string `IN`. It is part of the standard character and string functions according to IEC 61131-3.



Interface Structure

Event Inputs

- **REQ**: Starts the string extraction. It is linked to the data inputs `IN` and `L`.

Event Outputs

- **CNF**: Confirms successful extraction. Linked to data output `OUT`.

Data Inputs

- **IN** (`ANY_STRING`): The input string from which the leftmost characters are to be extracted.
- **L** (`ANY_INT`): The number of characters to be extracted from the left.

Data Outputs

- **OUT** (`ANY_STRING`): The extracted substring consisting of the leftmost characters of `L` from `IN`.

Adapters

No adapters are available.

Adapters ## Functionality

When the event `REQ` is triggered, the function block extracts the first characters `L` from the input string `IN` and outputs the result as `OUT`. The event `CNF` signals successful execution.

Technical Features

- Supports all string types (`ANY_STRING`).
- `L` must be a valid integer value. If `L` is greater than the length of `IN`, the entire string is returned.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Processes the input and extracts the characters.
3. **Completed**: Sends `CNF` with the result.

Application Scenarios

- Extracting prefixes from strings.

- Shortening strings for specific requirements.

⚖️ Comparison with Similar Function Blocks

- **F_RIGHT**: Extracts characters from the right.
- **F_MID**: Extracts characters from the middle of a string.

Conclusion

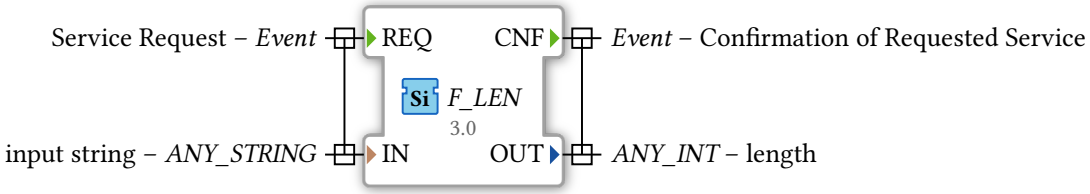
F_LEFT` is a simple and efficient function block for extracting substrings from the beginning of a string. Its use is intuitive and it meets the requirements of IEC 61131-3.

F_LEN

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LEN` is used to determine the length of an input string. It is part of the standard library for character and string functions according to IEC 61131-3.



Interface Structure

Event Inputs

- `REQ`: Service request. When this event is triggered, the string length calculation begins.

Event Outputs

- `CNF`: Acknowledgement of the service request. Triggered as soon as the calculation is complete.

Data Inputs

- `IN` (ANY_STRING): The input string whose length is to be determined.

Data Outputs

- `OUT` (ANY_INT): The length of the input string.

Adapters

No adapters available.

Functionality

The function block `F_LEN` calculates the length of the passed string (`IN`) and outputs the result as an integer (`OUT`). The calculation is started by the event `REQ` and confirmed by `CNF`.

Technical Features

- Supports all string types (`ANY_STRING`).
- Returns the length as an integer type (`ANY_INT`).

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Calculation State**: Calculates the string length.
3. **Acknowledgement State**: Sends the `CNF` event with the calculated length.

Application Scenarios

- Checking the length of user input.
- Validating string data before further processing.
- Logging and debugging string operations.

⚖️ Comparison with Similar Function Blocks

- Similar to `LEN` in other programming languages, but a standardized function block in IEC 61131-3.
- Unlike `F_MID` or `F_LEFT`, which extract parts of a string, `F_LEN` only returns the length.

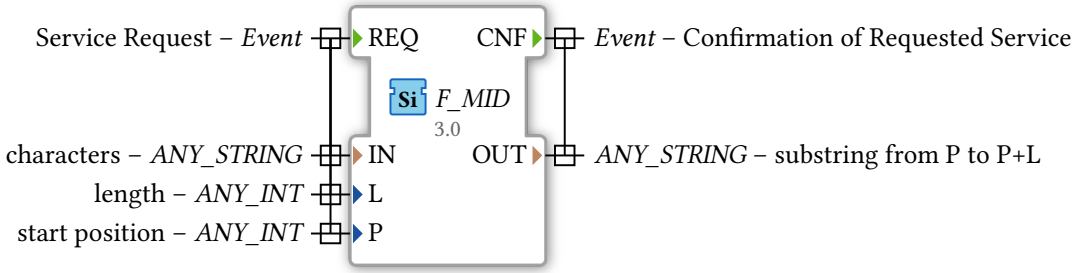
Conclusion

The `F_LEN` function block is a simple yet essential tool for string processing in IEC 61131-3 environments. Its clear interface and straightforward functionality make it a reliable building block for diverse applications.

F_MID

• – ◦ • – ◦ • – ◦ •

The function block `F_MID` is used to extract a substring from a given string. A specific number of characters are extracted starting from a defined position. This function block is part of the standard character and string functions according to IEC 61131-3.



- **REQ**: Service request to execute the extraction. This is linked to the data inputs `IN`, `L`, and `P`.
- **CNF**: Confirmation of successful request execution. Linked to the data output `OUT`.
- **IN**: The input string from which the substring is to be extracted (Type: `ANY_STRING`).
- **L**: The length of the substring to be extracted (Type: `ANY_INT`).
- **P**: The starting position from which the extraction should begin (Type: `ANY_INT`).
- **OUT**: The extracted substring (Type: `ANY_STRING`).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- No adapters available.

The function block `F_MID` extracts a substring from the input string `IN`. The extraction begins at position `P` and includes `L` characters. The result is output via output `OUT` as soon as the event output `CNF` is triggered.

- The function block supports any string type (`ANY_STRING`) and integer type (`ANY_INT`).
- The function block supports arbitrary string types (`ANY_STRING`) and integer types (`ANY_INT`).
- The start position `P` and the length `L` must be valid values within the length of the input string to ensure correct results.

1. **Initialization**: The function block waits for a request via `REQ`.
2. **Processing**: Upon receiving `REQ`, the input data is processed and the substring is extracted.
3. **Output**: The extracted substring is output via `OUT`, and `CNF` is triggered.

- Extraction of substrings from longer texts or data strings.
- Use in string processing routines, e.g., for parsing data formats.

- Similar function blocks like `F_LEFT` or `F_RIGHT` extract substrings from the beginning or end of a string, while `F_MID` allows for flexible extraction from any position.

The `F_MID` function block offers a simple and efficient way to extract substrings from a given string. Its flexibility and compatibility with various string and integer types make it a useful tool in string manipulation.

Functionality

Application Scenarios

State Overview

Technical Features

⚖️ Vergleich mit ähnlichen Bausteinen

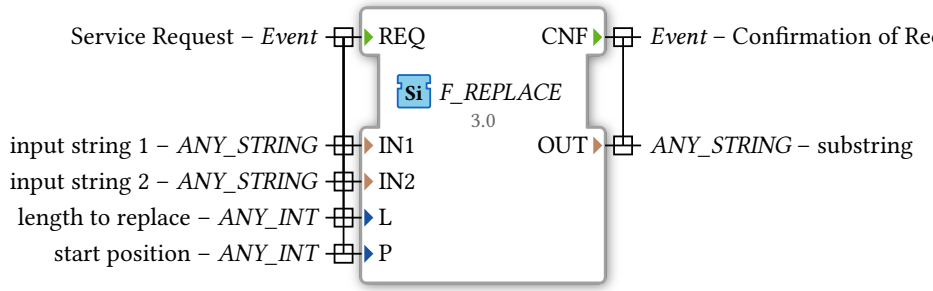
Conclusion

F_REPLACE

• – ◦ • – ◦ • – ◦ •

Introduction

The F_REPLACE function block is used to manipulate strings. It replaces a specific number of characters (L) in the first input string (IN1) with the second input string (IN2), starting at a specified position (P).



Interface Structure

Event Inputs

- **REQ**: Service request. Triggers the replacement operation. Linked to the data inputs IN1, IN2, L, and P.

Event Outputs

- **CNF**: Confirmation of the requested service. Triggered when the replacement is complete. Linked to the data output OUT.

Data Inputs

- **IN1** (ANY_STRING): The first input string in which the replacement is performed.
- **IN2** (ANY_STRING): The second input string used as a replacement.
- **L** (ANY_INT): The number of characters to be replaced.
- **P** (ANY_INT): The starting position for the replacement in IN1.

Data Outputs

- **OUT** (ANY_STRING): The resulting string after the replacement.

Adapters

No adapters available.

Operation

The function block performs the following operation:

1. Upon receiving the REQ event, the input values IN1, IN2, L, and P are read.

2. In the string IN1, exactly L characters are replaced with the string IN2, starting at position P.
3. The result is stored in OUT, and the CNF event is triggered.

Technical Features

- Supports any string type (ANY_STRING).
- The position P and length L must be valid indices within the string IN1 to ensure correct results.

State Overview

The function block has no internal states. The operation is re-executed with each REQ event.

Application Scenarios

- Text processing in automation applications.
- String manipulation in protocol messages.
- Dynamic adjustment of configuration strings.

⚖️ Comparison with Similar Function Blocks

- Similar to F_INSERT, however, F_REPLACE replaces existing characters instead of inserting new ones.
- Unlike F_DELETE, F_REPLACE replaces existing characters with new ones.

Conclusion

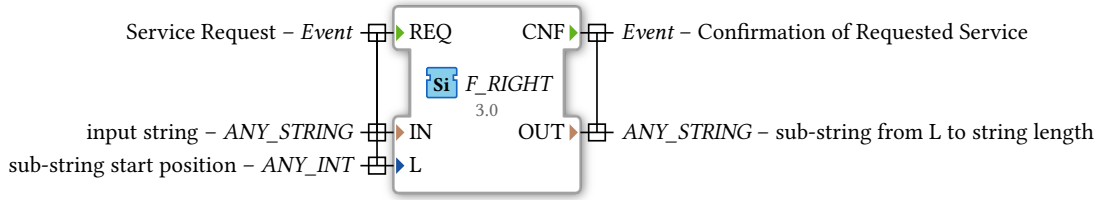
F_REPLACE is a useful function block for string manipulation in IEC 61131-3-based controllers. Its simple and clear functionality makes it ideal for various applications in automation technology.

F_RIGHT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_RIGHT` is a standard function block for processing character strings according to the IEC 61131-3 standard. It extracts the rightmost characters `L` from the input string `IN` and outputs them as the result `OUT`. This function block is particularly useful in applications where parts of character strings need to be processed or analyzed.



Interface Structure

Event Inputs

- **REQ**: Service Request. Triggers the extraction of the rightmost characters `L`. Linked to the data `IN` and `L`.

Event Outputs

- **CNF**: Confirmation of the requested service. Linked to the result `OUT`.

Data Inputs

- **IN**: Input string from which the rightmost characters are to be extracted. The data type is `ANY_STRING`.
- **L**: Number of characters to be extracted from the right of the input string. The data type is `ANY_INT`.

Data Outputs

- **OUT**: Result string containing the extracted characters. The data type is `ANY_STRING`.

Adapters

This function block has no adapters.

Functionality

When the event `REQ` is triggered, the input string `IN` is processed. The function block extracts the rightmost characters `L` from `IN` and outputs the result via the output `OUT`. The event `CNF` signals successful processing and provides the result.

Technical Features

- Supports various string types (`ANY_STRING`), allowing for flexible use.
- The position `L` can be specified using various integer types (`ANY_INT`).
- This function block is part of the `iec61131::charString` package.

State Overview

The function block has no internal states. Processing occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Extracting file extensions from filenames.
- Processing fixed-length string formats where relevant information is located at the end of the string.
- General string manipulation in control and automation applications.

⚖️ Comparison with Similar Function Blocks

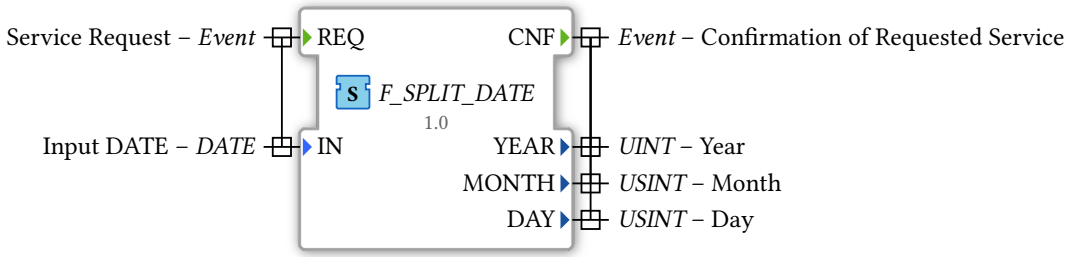
- Compared to `F_LEFT`, which extracts the leftmost characters of a string, `F_RIGHT` offers the opposite functionality.

Other string function blocks, such as `F_MID`, allow the extraction of substrings from the middle of a string, while `F_RIGHT` focuses specifically on the rightmost characters.

Conclusion

The function block `F_RIGHT` is a simple and effective building block for extracting characters from the end of a string. Its flexibility regarding supported data types makes it a versatile tool for string manipulation.

F_SPLIT_DATE



• – ◦ • – ◦ • – ◦ •

Introduction

`F_SPLIT_DATE` splits a `DATE` value into its individual date/time components (`YEAR`, `MONTH`, `DAY`). It is the reverse of F_CONCAT_DATE, which combines the same individual components back into a `DATE` value.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `YEAR`, `MONTH`, `DAY`.

Data Inputs

- **IN** (`DATE`): The date/time value to split.

Data Outputs

- **YEAR** (UINT): year.

- **MONTH** (USINT): month.
- **DAY** (USINT): day.

Functionality

On `REQ`, `IN` is split into its individual components, output via `YEAR`, `MONTH`, `DAY`. `CNF` is then triggered.

Technical Features

- **Fixed output types**: Unlike F_CONCAT_DATE (generic `ANY_INT`), the outputs here are fixed to concrete integer types chosen to fit each value's range (e.g. `USINT` for month/day, whose range is small).

State Overview

Stateless: every `REQ` immediately results in the split and `CNF`.

Application Scenarios

- **Displaying/processing individual date/time components**, e.g. for a formatted display on a panel or for calculations that only need a specific component.
- **Logging**: Breaking a timestamp down into its individual values for structured log output.

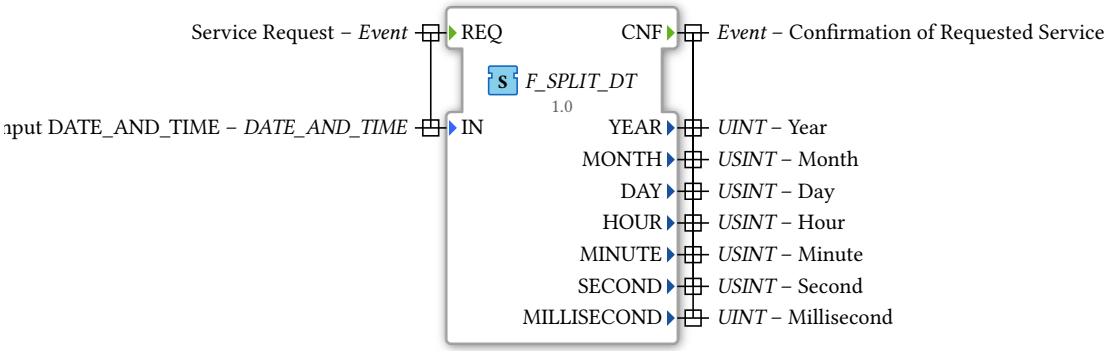
Comparison with similar function blocks

- **F_CONCAT_DATE**: the reverse direction — combines individual components into a `DATE` value.

Conclusion

`F_SPLIT_DATE` provides a simple, direct split of a `DATE` value into its individual components, complementing the corresponding merge function F_CONCAT_DATE.

F_SPLIT_DT



• – ◦ • – ◦ • – ◦ •

Introduction

`F_SPLIT_DT` splits a `DATE_AND_TIME` value into its individual date/time components (`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`). It is the reverse of F_CONCAT_DT, which combines the same individual components back into a `DATE_AND_TIME` value.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Data Inputs

- **IN** (`DATE_AND_TIME`): The date/time value to split.

Data Outputs

- **YEAR** (UINT): year.

- **MONTH** (USINT): month.
- **DAY** (USINT): day.
- **HOUR** (USINT): hour.
- **MINUTE** (USINT): minute.
- **SECOND** (USINT): second.
- **MILLISECOND** (UINT): millisecond.

Functionality

On `REQ`, `IN` is split into its individual components, output via `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`. `CNF` is then triggered.

Technical Features

- **Fixed output types**: Unlike F_CONCAT_DT (generic `ANY_INT`), the outputs here are fixed to concrete integer types chosen to fit each value's range (e.g. `USINT` for month/day, whose range is small).

State Overview

Stateless: every `REQ` immediately results in the split and `CNF`.

Application Scenarios

- **Displaying/processing individual date/time components**, e.g. for a formatted display on a panel or for calculations that only need a specific component.
- **Logging**: Breaking a timestamp down into its individual values for structured log output.

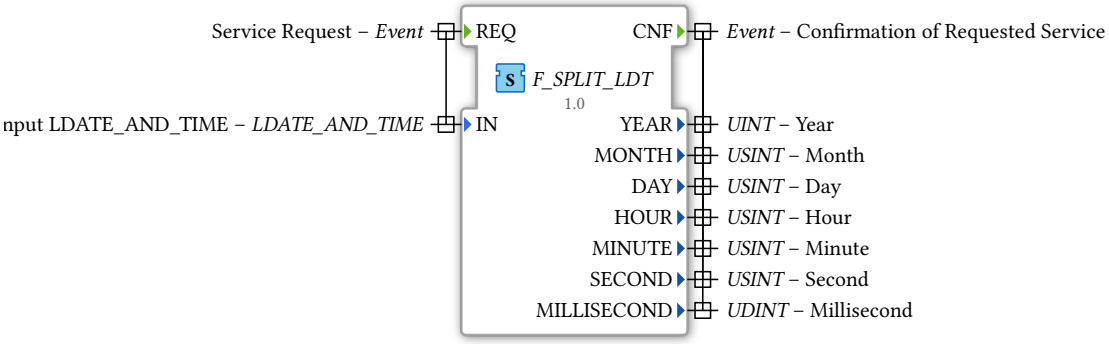
Comparison with similar function blocks

- **F_CONCAT_DT**: the reverse direction — combines individual components into a `DATE_AND_TIME` value.

Conclusion

`F_SPLIT_DT` provides a simple, direct split of a `DATE_AND_TIME` value into its individual components, complementing the corresponding merge function F_CONCAT_DT.

F_SPLIT_LDT



• – ◦ • – ◦ • – ◦ •

Introduction

`F_SPLIT_LDT` splits a `LDATE_AND_TIME` value into its individual date/time components (`YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`). It is the reverse of F_CONCAT_LDT, which combines the same individual components back into a `LDATE_AND_TIME` value.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Data Inputs

- **IN** (`LDATE_AND_TIME`): The date/time value to split.

Data Outputs

- **YEAR** (UINT): year.
- **MONTH** (USINT): month.
- **DAY** (USINT): day.
- **HOUR** (USINT): hour.
- **MINUTE** (USINT): minute.
- **SECOND** (USINT): second.
- **MILLISECOND** (UDINT): millisecond.

Functionality

On `REQ`, `IN` is split into its individual components, output via `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`. `CNF` is then triggered.

Technical Features

- **Fixed output types**: Unlike F_CONCAT_LDT (generic `ANY_INT`), the outputs here are fixed to concrete integer types chosen to fit each value's range (e.g. `USINT` for month/day, whose range is small).

State Overview

Stateless: every `REQ` immediately results in the split and `CNF`.

Application Scenarios

- **Displaying/processing individual date/time components**, e.g. for a formatted display on a panel or for calculations that only need a specific component.
- **Logging**: Breaking a timestamp down into its individual values for structured log output.

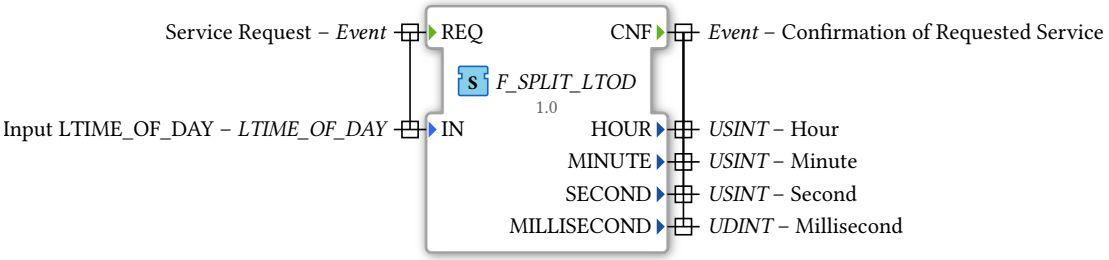
Comparison with similar function blocks

- **F_CONCAT_LDT**: the reverse direction — combines individual components into a `LDATE_AND_TIME` value.

Conclusion

`F_SPLIT_LDT` provides a simple, direct split of a `LDATE_AND_TIME` value into its individual components, complementing the corresponding merge function F_CONCAT_LDT.

F_SPLIT_LTOD



• – ◦ • – ◦ • – ◦ •

Introduction

`F_SPLIT_LTOD` splits a `LTIME_OF_DAY` value into its individual date/time components (`HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`). It is the reverse of F_CONCAT_LTOD, which combines the same individual components back into a `LTIME_OF_DAY` value.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Data Inputs

- **IN** (`LTIME_OF_DAY`): The date/time value to split.

Data Outputs

- **HOUR** (USINT): hour.
- **MINUTE** (USINT): minute.
- **SECOND** (USINT): second.
- **MILLISECOND** (UDINT): millisecond.

Functionality

On `REQ`, `IN` is split into its individual components, output via `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`. `CNF` is then triggered.

Technical Features

- **Fixed output types**: Unlike F_CONCAT_LTOD (generic `ANY_INT`), the outputs here are fixed to concrete integer types chosen to fit each value's range (e.g. `USINT` for month/day, whose range is small).

State Overview

Stateless: every `REQ` immediately results in the split and `CNF`.

Application Scenarios

- **Displaying/processing individual date/time components**, e.g. for a formatted display on a panel or for calculations that only need a specific component.
- **Logging**: Breaking a timestamp down into its individual values for structured log output.

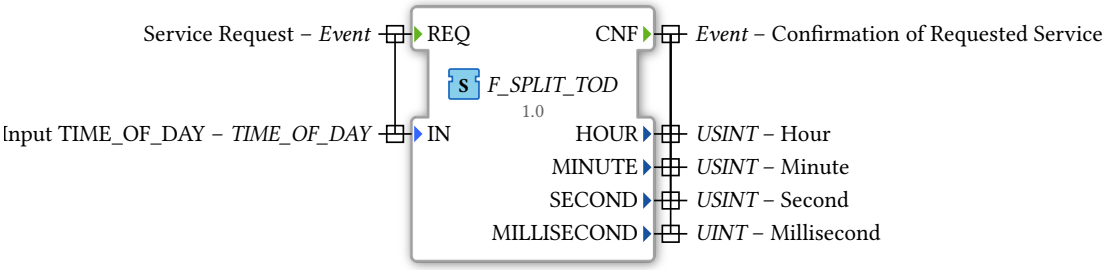
Comparison with similar function blocks

- **F_CONCAT_LTOD**: the reverse direction — combines individual components into a `LTIME_OF_DAY` value.

Conclusion

`F_SPLIT_LTOD` provides a simple, direct split of a `LTIME_OF_DAY` value into its individual components, complementing the corresponding merge function F_CONCAT_LTOD.

F_SPLIT_TOD



• – ◦ • – ◦ • – ◦ •

Introduction

`F_SPLIT_TOD` splits a `TIME_OF_DAY` value into its individual date/time components (`HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`). It is the reverse of F_CONCAT_TOD, which combines the same individual components back into a `TIME_OF_DAY` value.

Interface Structure

Event Inputs

- **REQ**: Triggers the split, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`.

Data Inputs

- **IN** (`TIME_OF_DAY`): The date/time value to split.

Data Outputs

- **HOUR** (USINT): hour.
- **MINUTE** (USINT): minute.
- **SECOND** (USINT): second.
- **MILLISECOND** (UINT): millisecond.

Functionality

On `REQ`, `IN` is split into its individual components, output via `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`. `CNF` is then triggered.

Technical Features

- **Fixed output types**: Unlike F_CONCAT_TOD (generic `ANY_INT`), the outputs here are fixed to concrete integer types chosen to fit each value's range (e.g. `USINT` for month/day, whose range is small).

State Overview

Stateless: every `REQ` immediately results in the split and `CNF`.

Application Scenarios

- **Displaying/processing individual date/time components**, e.g. for a formatted display on a panel or for calculations that only need a specific component.
- **Logging**: Breaking a timestamp down into its individual values for structured log output.

Comparison with similar function blocks

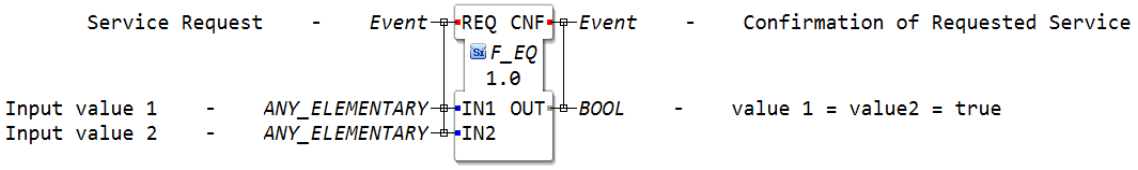
- **F_CONCAT_TOD**: the reverse direction — combines individual components into a `TIME_OF_DAY` value.

Conclusion

`F_SPLIT_TOD` provides a simple, direct split of a `TIME_OF_DAY` value into its individual components, complementing the corresponding merge function F_CONCAT_TOD.

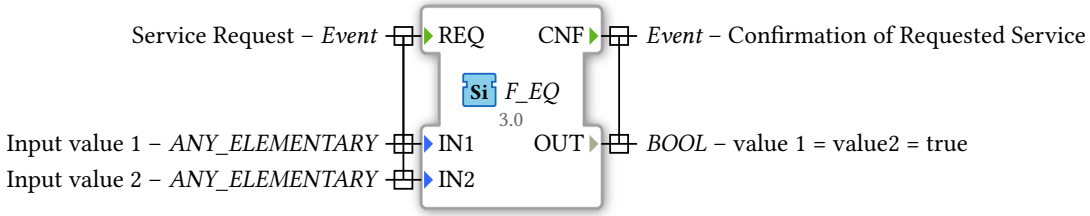
2.11.4.6 ⚖️ Comparison functions

F_EQ



https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_eq.html

• – ○ • – ○ • – ○ •



Interface

Event Inputs

- `REQ`: Service request, carries `IN1` and `IN2`.

Event Outputs

- `CNF`: Confirmation of the requested service, carries `OUT`.

Data Inputs

- `IN1` (`ANY_ELEMENTARY`): Input value 1.
- `IN2` (`ANY_ELEMENTARY`): Input value 2.

Data Outputs

- `OUT` (`BOOL`): `TRUE` if value 1 = value 2.

The following text was generated by https://chat.deepseek.com/ on February 5, 2025, and has not yet been proofread (HF)

The EQ Operator in IEC 61499: An Introduction

IEC 61499 is a standard that deals with function block modeling for industrial process measurement and control systems. It provides a framework for developing distributed control systems based on the use of function blocks. A central element of this standard is the definition of operators used in the function blocks to perform logical and arithmetic operations. One of these operators is the EQ operator, which will be examined in more detail in this paper.

What is the EQ operator?

The EQ operator, also known as the equality operator, is a logical operator used to check whether two values are equal. In IEC 61499, this operator is frequently used in function blocks to evaluate conditions and make decisions. The EQ operator compares two input values and returns a Boolean value (TRUE or FALSE) depending on whether the values are equal or not.

Syntax and Usage

In IEC 61499, the EQ operator is typically used in the expressions of function blocks. The syntax of the EQ operator is simple:

EQ(Eingang1, Eingang2) Here, `Eingang1` and `Eingang2` are the two values to be compared. The operator returns `TRUE` if `Eingang1` and `Eingang2` are equal; otherwise, the return value is `FALSE`.

Application Examples

1. **Comparison of Sensor Values:**

Suppose a function block needs to check if two sensors are returning the same value. The EQ operator can be used to compare the values of the two sensors:

IF EQ(Sensor1, Sensor2) THEN

1. **Controlling Process Steps:**

In a production process, the EQ operator can be used to check whether a specific process step has been completed. If the current step equals the target step, the next step can be initiated:

IF EQ(AktuellerSchritt, ZielSchritt) THEN

1. **Error Detection:**

The EQ operator can also be used for error detection. For example, if a target value and an actual value are compared and they do not match, an error signal can be generated:

IF NOT EQ(Sollwert, Istwert) THEN

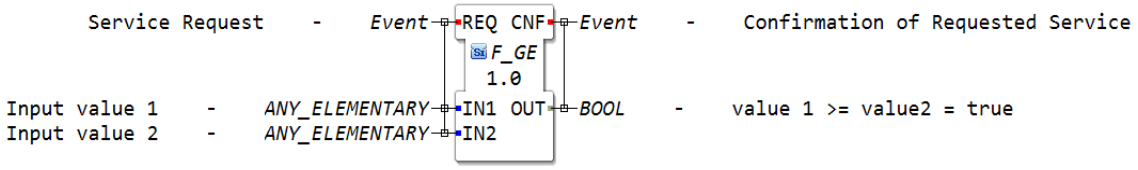
Advantages of the EQ Operator

- **Simplicity:** The EQ operator is easy to understand and use, which simplifies the development of control logic.
- **Flexibility:** The operator can be applied to various data types, including integers, floating-point numbers, and even Boolean values.
- **Efficiency:** Because the EQ operator can be used directly in function blocks, it reduces the need for additional logic and makes the code more efficient.

Conclusion

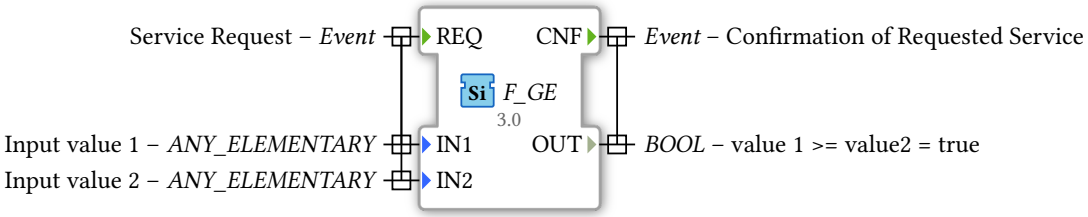
The EQ operator is a fundamental tool in IEC 61499 that simplifies the implementation of logical comparisons in industrial control systems. Its simple syntax and versatile applicability contribute to the development of robust and efficient control algorithms. Whether in sensor monitoring, process control, or fault detection, the EQ operator is an indispensable element in the world of industrial automation.

F_GE



https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_ge.html

• – ◦ • – ◦ • – ◦ •



Interface

Port	Type	Description
REQ	Event	Service request (with IN1, IN2)
CNF	Event	Confirmation of requested service (with OUT)
IN1	ANY_ELEMENTARY	Input value 1
IN2	ANY_ELEMENTARY	Input value 2
OUT	BOOL	TRUE if IN1 greater than or equal to IN2

The following text was generated by https://chat.deepseek.com/ on February 5, 2025, and has not yet been proofread (HF)

The GE Operator in IEC 61499: An Introduction

IEC 61499 is an international standard that deals with the modeling of function blocks for industrial automation systems. It provides a standardized framework for the development of distributed control systems, in which logical and arithmetic operators play a central role. One of these operators is the **GE Operator** (Greater Than or Equal To), which will be examined in more detail in this article.

What is the GE Operator?

The GE operator is a comparison operator used to determine whether one value is greater than or equal to another. In IEC 61499, this operator is frequently used in function blocks to evaluate conditions and make decisions based on numerical comparisons. The GE operator compares two input values and returns a Boolean value (`TRUE` or `FALSE`) depending on whether the first value is greater than or equal to the second.

Syntax and Usage

In IEC 61499, the GE operator is used in function block expressions. The syntax of the GE operator is as follows:

GE(Eingang1, Eingang2) Here, `Eingang1` and `Eingang2` are the two values to be compared. The operator returns `TRUE` if `Eingang1` is greater than or equal to `Eingang2`; otherwise, the return value is `FALSE`.

Application Examples

1. **Monitoring Limit Values:**

In industrial processes, it is often necessary to monitor limit values. The GE operator can be used to check whether a measured value reaches or exceeds a specific threshold:
IF GE(Messwert, Schwellenwert) THEN

1. **Controlling Production Processes:**

In a production process, the GE operator can be used to determine whether a specific quantity of material has been produced. If the produced quantity is greater than or equal to the target quantity, the process can be stopped:
IF GE(ProduzierteMenge, ZielMenge) THEN

1. **Temperature Control:**

In a heating system, the GE operator can be used to check whether the current temperature has reached or exceeded the setpoint. If so, the heating can be switched off:
IF GE(AktuelleTemperatur, SollTemperatur) THEN

1. **Energy Consumption Monitoring:**

The GE Operator can also be used to monitor energy consumption. If consumption exceeds a certain threshold, energy-saving measures can be initiated:
IF GE(EnergieVerbrauch, Grenzwert) THEN

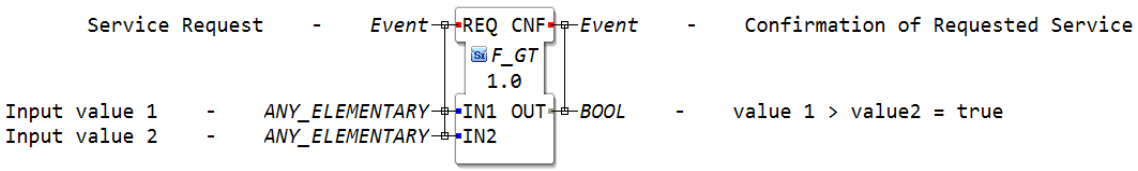
Advantages of the GE Operator

- **Simple Logic:** The GE operator allows complex conditions to be formulated in simple, easily understandable expressions.
- **Flexibility:** The operator can be applied to various numeric data types, including integers and floating-point numbers.
- **Efficiency:** By being directly integrated into function blocks, the GE operator reduces the need for additional programming and makes the code more efficient.
- **Robustness:** The GE operator helps to develop robust control algorithms based on precise numerical comparisons.

Conclusion

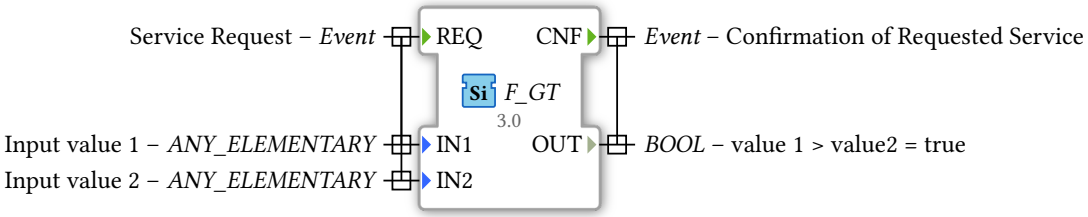
The GE operator is an important tool in IEC 61499 that simplifies the implementation of numerical comparisons in industrial control systems. Its ability to determine whether one value is greater than or equal to another makes it an indispensable component in process monitoring, control, and decision-making. Whether in production control, temperature regulation, or energy consumption monitoring – the GE Operator is a powerful and versatile element in the world of industrial automation. Its simplicity and efficiency contribute to the development of reliable and effective control systems.

F_GT



https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_gt.html

• – ○ • – ○ • – ○ •



The following text was generated by https://chat.deepseek.com/ on February 5, 2025, and has not yet been proofread (HF)

The GT Operator in IEC 61499: An Introduction

IEC 61499 is an international standard that deals with the modeling of function blocks for industrial automation systems. It provides a standardized framework for the development of distributed control systems, in which logical and arithmetic operators play a central role. One of these operators is the **GT Operator** (Greater Than), which will be examined in more detail in this article.

Interface

Event Inputs

- `REQ`: Service request (with `IN1`, `IN2`)

Event Outputs

- `CNF`: Confirmation of the requested operation (with `OUT`)

Data Inputs

- `IN1` (ANY_ELEMENTARY): Input value 1
- `IN2` (ANY_ELEMENTARY): Input value 2

Data Outputs

- `OUT` (BOOL): value 1 > value 2 = TRUE

What is the GT Operator?

The GT operator is a comparison operator used to determine whether one value is greater than another. In IEC 61499, this operator is frequently used in function blocks to evaluate conditions and make decisions based on numerical comparisons. The GT operator compares two input values and returns a Boolean value (`TRUE` or `FALSE`) depending on whether the first value is greater than the second.

Syntax and Usage

In IEC 61499, the GT operator is used in function block expressions. The syntax of the GT operator is as follows:

GT(Eingang1, Eingang2) Here, `Eingang1` and `Eingang2` are the two values to be compared. The operator returns `TRUE` if `Eingang1` is greater than `Eingang2`; otherwise, the return value is `FALSE`.

Application Examples

1. **Monitoring Limit Values:**

In industrial processes, it is often necessary to monitor limit values. The GT operator can be used to check whether a measured value exceeds a specific threshold:

IF GT(Messwert, Schwellenwert) THEN

1. **Controlling Production Processes:**

In a production process, the GT operator can be used to determine whether a specific quantity of material has been produced. If the produced quantity is greater than the target quantity, the process can be stopped:

IF GT(ProduzierteMenge, ZielMenge) THEN

1. **Temperature Control:**

In a heating system, the GT operator can be used to check whether the current temperature exceeds the setpoint. If so, the heating can be switched off:

IF GT(AktuelleTemperatur, SollTemperatur) THEN

1. **Energy Consumption Monitoring:**

The GT operator can also be used to monitor energy consumption. If consumption exceeds a certain value, energy-saving measures can be initiated:

IF GT(EnergieVerbrauch, Grenzwert) THEN

Advantages of the GT Operator

- **Simple Logic:** The GT operator allows complex conditions to be formulated in simple, easily understandable expressions.
- **Flexibility:** The operator can be applied to various numeric data types, including integers and floating-point numbers.

- **Efficiency:** By being directly integrated into function blocks, the GT operator reduces the need for additional programming and makes the code more efficient.
- **Robustness:** The GT operator helps to develop robust control algorithms based on precise numerical comparisons.

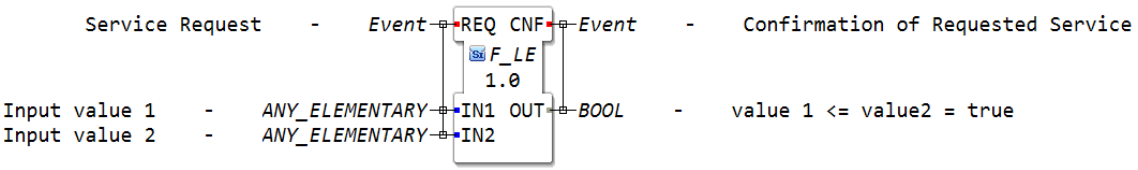
🛠️ Related Exercises

- Exercise_060
- Exercise_071
- Exercise_071a
- Exercise_110
- Exercise_111

Conclusion

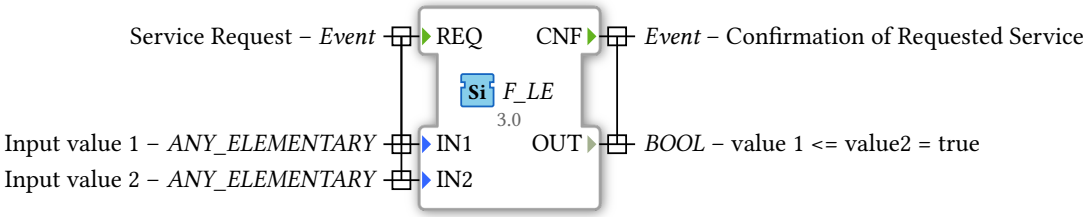
The GT operator is an important tool in IEC 61499 that simplifies the implementation of numerical comparisons in industrial control systems. Its ability to determine whether one value is greater than another makes it an indispensable component in process monitoring, control, and decision-making. Whether in production control, temperature regulation, or energy consumption monitoring – the GT Operator is a powerful and versatile element in the world of industrial automation. Its simplicity and efficiency contribute to the development of reliable and effective control systems.

F_LE



https://content.helpme-codesys.com/de/CODESYS%20Development%20System/_cds_operator_le.html
https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_le.html

• – ◦ • – ◦ • – ◦ •



Interface

Event Inputs

- `REQ`: Service request, carries `IN1` and `IN2`.

Event Outputs

- `CNF`: Confirmation of the requested service, carries `OUT`.

Data Inputs

- `IN1` (`ANY_ELEMENTARY`): Input value 1.
- `IN2` (`ANY_ELEMENTARY`): Input value 2.

Data Outputs

- `OUT` (`BOOL`): `TRUE` if value 1 <= value 2.

The following text was generated by https://chat.deepseek.com/ on February 5, 2025, and has not yet been proofread (HF)

The LE Operator in IEC 61499: An Introduction

IEC 61499 is an international standard that deals with the modeling of function blocks for industrial automation systems. It provides a standardized framework for the development of distributed control

systems, in which logical and arithmetic operators play a central role. One of these operators is the **LE operator** (Less Than or Equal To), which will be examined in more detail in this paper.

What is the LE operator?

The LE operator is a comparison operator used to determine whether one value is less than or equal to another. In IEC 61499, this operator is frequently used in function blocks to evaluate conditions and make decisions based on numerical comparisons. The LE operator compares two input values and returns a Boolean value (`TRUE` or `FALSE`) depending on whether the first value is less than or equal to the second.

Syntax and Usage

In IEC 61499, the LE operator is used in function block expressions. The syntax of the LE operator is as follows:

LE(Eingang1, Eingang2) Here, `Eingang1` and `Eingang2` are the two values to be compared. The operator returns `TRUE` if `Eingang1` is less than or equal to `Eingang2`; otherwise, the return value is `FALSE`.

Application Examples

1. **Monitoring Limit Values:**

In industrial processes, it is often necessary to monitor limit values. The LE operator can be used to check whether a measured value is below or equal to a specific threshold:

IF LE(Messwert, Schwellenwert) THEN

1. **Controlling Production Processes:**

In a production process, the LE operator can be used to determine whether a specific quantity of material has been produced. If the produced quantity is less than or equal to the target quantity, the process can continue:

IF LE(ProduzierteMenge, ZielMenge) THEN

1. **Temperature Control:**

In a cooling system, the LE operator can be used to check whether the current temperature is less than or equal to the set temperature. If so, the cooling can be switched off:

IF LE(AktuelleTemperatur, SollTemperatur) THEN

1. **Energy Consumption Monitoring:**

The LE operator can also be used to monitor energy consumption. If consumption is less than or equal to a certain value, energy-saving measures can be deactivated:

IF LE(EnergieVerbrauch, Grenzwert) THEN

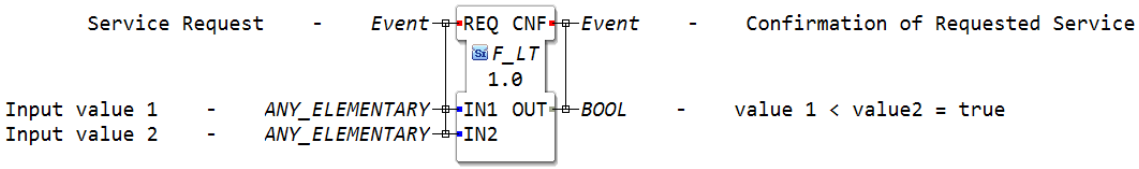
Advantages of the LE Operator

- **Simple Logic:** The LE operator allows complex conditions to be formulated in simple, easily understandable expressions.
- **Flexibility:** The operator can be applied to various numeric data types, including integers and floating-point numbers.
- **Efficiency:** By being directly integrated into function blocks, the LE operator reduces the need for additional programming and makes the code more efficient.
- **Robustness:** The LE operator helps to develop robust control algorithms based on precise numerical comparisons.

Conclusion

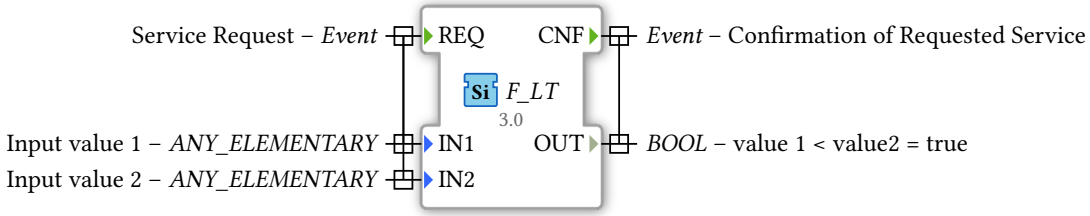
The LE operator is an important tool in IEC 61499 that simplifies the implementation of numerical comparisons in industrial control systems. Its ability to determine whether one value is less than or equal to another makes it an indispensable component in process monitoring, control, and decision-making. Whether in production control, temperature regulation, or energy consumption monitoring – the LE operator is a powerful and versatile element in the world of industrial automation. Its simplicity and efficiency contribute to the development of reliable and effective control systems.

F_LT



https://content.helpme-codesys.com/de/CODESYS%20Development%20System/_cds_operator_lt.html

• – ◦ • – ◦ • – ◦ •



Interface

Port	Type	Description
REQ	Event	Service request (with IN1, IN2)
CNF	Event	Confirmation of requested service (with OUT)
IN1	ANY_ELEMENTARY	Input value 1
IN2	ANY_ELEMENTARY	Input value 2
OUT	BOOL	TRUE if IN1 less than IN2

The following text was generated by https://chat.deepseek.com/ on February 5, 2025, and has not yet been proofread (HF)

The LT Operator in IEC 61499: An Introduction

IEC 61499 is an international standard that deals with the modeling of function blocks for industrial automation systems. It provides a standardized framework for the development of distributed control systems, in which logical and arithmetic operators play a central role. One of these operators is the **LT Operator** (Less Than), which will be examined in more detail in this article.

What is the LT Operator?

The LT operator is a comparison operator used to determine whether one value is less than another. In IEC 61499, this operator is frequently used in function blocks to evaluate conditions and make decisions based on numerical comparisons. The LT operator compares two input values and returns a Boolean value (`TRUE` or `FALSE`) depending on whether the first value is less than the second.

Syntax and Usage

In IEC 61499, the LT operator is used in function block expressions. The syntax of the LT operator is as follows:

LT(Eingang1, Eingang2) Here, `Eingang1` and `Eingang2` are the two values to be compared. The operator returns `TRUE` if `Eingang1` is less than `Eingang2`; otherwise, the return value is `FALSE`.

Application Examples

1. **Monitoring Limit Values:**

In industrial processes, it is often necessary to monitor limit values. The LT operator can be used to check whether a measured value falls below a specific threshold:

IF LT(Messwert, Schwellenwert) THEN

1. **Controlling Production Processes:**

In a production process, the LT operator can be used to determine whether a specific quantity of material has been produced. If the produced quantity is less than the target quantity, the process can continue:

IF LT(ProduzierteMenge, ZielMenge) THEN

1. **Temperature Control:**

In a cooling system, the LT operator can be used to check if the current temperature is lower than the setpoint temperature. If so, cooling can remain active:
IF LT(AktuelleTemperatur, SollTemperatur) THEN

1. **Energy Consumption Monitoring:**

The LT operator can also be used to monitor energy consumption. If consumption falls below a certain threshold, energy-saving measures can be deactivated:
IF LT(EnergieVerbrauch, Grenzwert) THEN

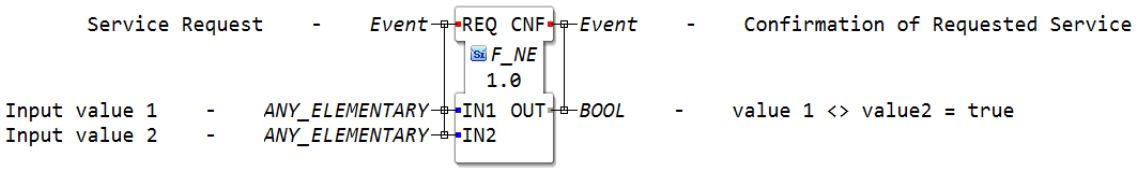
Advantages of the LT Operator

- **Simple Logic:** The LT operator allows complex conditions to be formulated in simple, easily understandable expressions.
- **Flexibility:** The operator can be applied to various numeric data types, including integers and floating-point numbers.
- **Efficiency:** By being directly integrated into function blocks, the LT operator reduces the need for additional programming and makes the code more efficient.
- **Robustness:** The LT operator helps to develop robust control algorithms based on precise numerical comparisons.

Conclusion

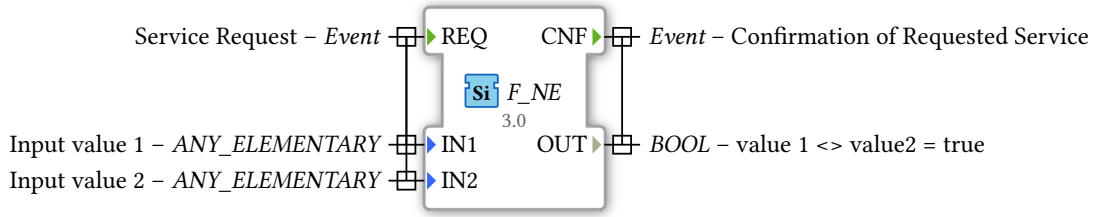
The LT operator is an important tool in IEC 61499 that simplifies the implementation of numerical comparisons in industrial control systems. Its ability to determine whether one value is less than another makes it an indispensable component in process monitoring, control, and decision-making. Whether in production control, temperature regulation, or energy consumption monitoring – the LT Operator is a powerful and versatile element in the world of industrial automation. Its simplicity and efficiency contribute to the development of reliable and effective control systems.

F_NE



https://content.helpme-codesys.com/de/CODESYS%20Development%20System/_cds_operator_ne.html
https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_ne.html

• – ◦ • – ◦ • – ◦ •



The following text was generated by https://chat.deepseek.com/ on February 5, 2025, and has not yet been proofread (HF)

The NE Operator in IEC 61499: An Introduction

IEC 61499 is an international standard that deals with the modeling of function blocks for industrial automation systems. It provides a standardized framework for the development of distributed control systems, in which logical and arithmetic operators play a central role. One of these operators is the **NE operator** (Not Equal), which will be examined in more detail in this article.

Interface

Event Inputs

- `REQ`: Service request (with `IN1`, `IN2`)

Event Outputs

- `CNF`: Confirmation of the requested operation (with `OUT`)

Data Inputs

- `IN1` (ANY_ELEMENTARY): Input value 1
- `IN2` (ANY_ELEMENTARY): Input value 2

Data Outputs

- `OUT` (BOOL): value 1 <> value 2 = TRUE

What is the NE operator?

The NE operator is a comparison operator used to determine whether two values are unequal. In IEC 61499, this operator is frequently used in function blocks to evaluate conditions and make decisions based on comparisons. The NE operator compares two input values and returns a Boolean value (`TRUE` or `FALSE`) depending on whether the values are unequal.

Syntax and Usage

In IEC 61499, the NE operator is used in function block expressions. The syntax of the NE operator is as follows:

NE(Eingang1, Eingang2) Here, `Eingang1` and `Eingang2` are the two values to be compared. The operator returns `TRUE` if `Eingang1` is not equal to `Eingang2`; otherwise, the return value is `FALSE`.

Application Examples

1. **Monitoring Sensor Values:**

In industrial processes, it is often necessary to monitor sensor values. The NE operator can be used to check whether two sensors are providing different values:

IF NE(Sensor1, Sensor2) THEN

1. **Control of Production Processes:**

In a production process, the NE operator can be used to determine whether a specific process step is not yet complete. If the current step is not the target step, the process can continue:

IF NE(AktuellerSchritt, ZielSchritt) THEN

1. **Temperature Control:**

In a heating system, the NE operator can be used to check if the current temperature is not equal to the set temperature. If this is the case, the heating can remain activated:

IF NE(AktuelleTemperatur, SollTemperatur) THEN

1. **Error Detection:**

The NE operator can also be used for error detection. For example, if a setpoint and an actual value are compared and they do not match, an error signal can be generated:

IF NE(Sollwert, Istwert) THEN

Advantages of the NE Operator

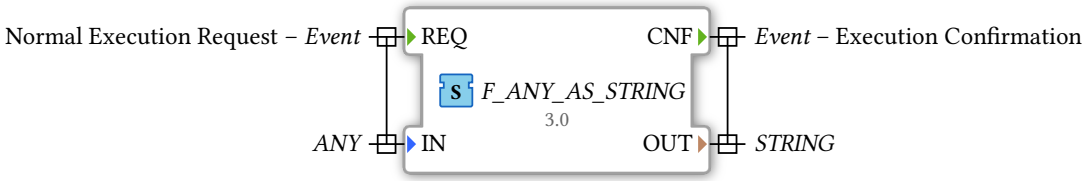
- **Simple Logic:** The NE operator allows complex conditions to be formulated in simple, easily understandable expressions.
- **Flexibility:** The operator can be applied to various data types, including integers, floating-point numbers, and even Boolean values.
- **Efficiency:** By being directly integrated into function blocks, the NE operator reduces the need for additional programming and makes the code more efficient.
- **Robustness:** The NE operator helps develop robust control algorithms based on precise comparisons.

Conclusion

The NE operator is an important tool in IEC 61499 that simplifies the implementation of comparisons in industrial control systems. Its ability to determine whether two values are unequal makes it an indispensable component in process monitoring, control, and decision-making. Whether in production control, temperature regulation, or fault detection – the NE operator is a powerful and versatile element in the world of industrial automation. Its simplicity and efficiency contribute to the development of reliable and effective control systems.

2.11.4.7 conversion

F_ANY_AS_STRING



• – ◦ • – ◦ • – ◦ •

Introduction

`F_ANY_AS_STRING` converts a value of arbitrary data type (`ANY`) into its textual representation as a `STRING`. It serves as a generic "to text" converter, e.g. for diagnostics or logging, without needing a dedicated conversion function for every data type.

Interface Structure

Event Inputs

- **REQ**: Triggers the conversion, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **IN** (ANY): The value to convert, of arbitrary data type.

Data Outputs

- **OUT** (STRING): The textual representation of `IN`.

Functionality

On `REQ`, the block generates a textual representation from the generically typed input `IN` and outputs it via `OUT`, then triggers `CNF`. The exact formatting follows the IEC 61131-3 literal notation of the concrete data type bound to `IN`.

Technical Features

- **Generic `ANY` input**: The block accepts any IEC 61131-3 data type bound to `IN` at compile time.
- **Universal debug/logging block**: Since virtually any data type can be output as `STRING`, the block is particularly suited for generic diagnostic output.

State Overview

Stateless: every `REQ` immediately results in the conversion and `CNF`.

Application Scenarios

- **Diagnostics and logging**: Output of arbitrary process values as readable text, e.g. for log files or diagnostic displays.
- **Generic display blocks** that need a textual representation independent of the actual data type of the connected value.

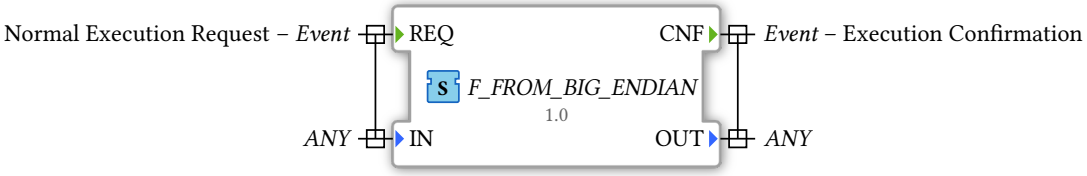
Comparison with similar function blocks

- **Type-specific `*_TO_STRING` functions** (e.g. `DINT_TO_STRING`): require a dedicated instance per data type, but may offer finer-grained formatting control.

Conclusion

`F_ANY_AS_STRING` provides a convenient, generically typed text conversion for arbitrary data types and is particularly suited for diagnostics and logging applications where the concrete data type may vary.

F_FROM_BIG_ENDIAN



• – ◦ • – ◦ • – ◦ •

Introduction

`F_FROM_BIG_ENDIAN` converts a value from big-endian (most significant byte first) byte order. It belongs to the family of endian conversion functions needed for cross-platform exchange of binary data (e.g. over network protocols or with third-party systems).

Interface Structure

Event Inputs

- **REQ**: Triggers the conversion, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **IN** (ANY): The value to convert, of arbitrary data type.

Data Outputs

- **OUT** (ANY): The converted value, same data type as `IN`.

Functionality

On `REQ`, the block interprets the byte sequence of `IN` as a big-endian (most significant byte first)-encoded value and converts it into the target platform's native byte order. The result is output via `OUT`, then `CNF` is triggered. The bit width of the data type stays unchanged — only the byte order is adjusted.

Technical Features

- **Generic `ANY` data type**: `IN`/`OUT` adapt to whichever concrete data type is connected at compile time; the byte count follows from its size.
- **Pure byte reordering, no value reinterpretation**: The block only changes the arrangement of the bytes, not their meaning or sign.

State Overview

Stateless: every `REQ` immediately results in the conversion and `CNF`.

Application Scenarios

- **Network communication**: Many network protocols (e.g. TCP/IP headers) use big-endian ("network byte order"); `F_FROM_BIG_ENDIAN` ensures the correct byte order when sending/receiving.
- **Interoperability with third-party systems**: Exchanging binary data with systems whose native byte order differs from the target platform's.

Comparison with similar function blocks

- **`F_FROM_LITTLE_ENDIAN`**: the same conversion direction, but for the other byte order.

- **F_TO_BIG_ENDIAN**: the reverse direction for the same byte order.

Conclusion

F_FROM_BIG_ENDIAN provides a simple, generically typed byte-order conversion and is an important building block for correct interoperability with network protocols and third-party systems of differing endianness.

F_FROM_LITTLE_ENDIAN



• – ◦ • – ◦ • – ◦ •

Introduction

F_FROM_LITTLE_ENDIAN converts a value from little-endian (least significant byte first) byte order. It belongs to the family of endian conversion functions needed for cross-platform exchange of binary data (e.g. over network protocols or with third-party systems).

Interface Structure

Event Inputs

- **REQ**: Triggers the conversion, carries IN.

Event Outputs

- **CNF**: Confirms completion, carries OUT.

Data Inputs

- **IN** (ANY): The value to convert, of arbitrary data type.

Data Outputs

- **OUT** (ANY): The converted value, same data type as IN.

Functionality

On REQ, the block interprets the byte sequence of IN as a little-endian (least significant byte first)-encoded value and converts it into the target platform's native byte order. The result is output via OUT, then CNF is triggered. The bit width of the data type stays unchanged — only the byte order is adjusted.

Technical Features

- **Generic ANY data type**: IN/OUT adapt to whichever concrete data type is connected at compile time; the byte count follows from its size.
- **Pure byte reordering, no value reinterpretation**: The block only changes the arrangement of the bytes, not their meaning or sign.

State Overview

Stateless: every REQ immediately results in the conversion and CNF.

Application Scenarios

- **Network communication**: Many network protocols (e.g. TCP/IP headers) use big-endian ("network byte order"); F_FROM_LITTLE_ENDIAN ensures the correct byte order when sending/receiving.
- **Interoperability with third-party systems**: Exchanging binary data with systems whose native byte order differs from the target platform's.

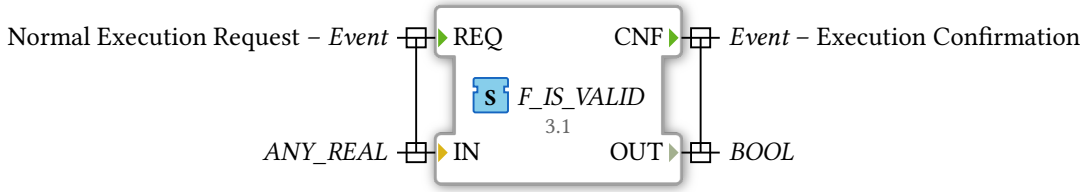
Comparison with similar function blocks

- **F_FROM_BIG_ENDIAN**: the same conversion direction, but for the other byte order.
- **F_TO_LITTLE_ENDIAN**: the reverse direction for the same byte order.

Conclusion

`F_FROM_LITTLE_ENDIAN` provides a simple, generically typed byte-order conversion and is an important building block for correct interoperability with network protocols and third-party systems of differing endianness.

F_IS_VALID



• – ◦ • – ◦ • – ◦ •

Introduction

`F_IS_VALID` checks whether a floating-point value (`REAL`/`LREAL`) is a valid, finite value — i.e. neither `NaN` (Not a Number) nor `+/-Infinity`. It guards numerical calculations against invalid intermediate results.

Interface Structure

Event Inputs

- **REQ**: Triggers the check, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **IN** (ANY_REAL): The floating-point value to check.

Data Outputs

- **OUT** (BOOL): `TRUE` if `IN` is a valid, finite value; `FALSE` for `NaN` or `Infinity`.

Functionality

On `REQ`, the algorithm's ST statement `OUT := IS_VALID(IN)` checks whether `IN` is neither `NaN` nor `+/-Infinity`, and outputs the result via `OUT`. `CNF` is then triggered.

Technical Features

- **`ANY_REAL` input**: Accepts both `REAL` and `LREAL`.
- **Detects typical error results**: `NaN` arises e.g. from `0.0/0.0`, `Infinity` from division by `0.0` with a nonzero numerator — both classic sources of error in floating-point calculations.

State Overview

Stateless: every `REQ` immediately results in the check and `CNF`.

Application Scenarios

- **Guarding calculation chains**: Checking a floating-point result for validity before further processing, to prevent follow-on errors (e.g. `NaN` propagation).
- **Sanity checking sensor data**: Detecting faulty or out-of-range measurements.

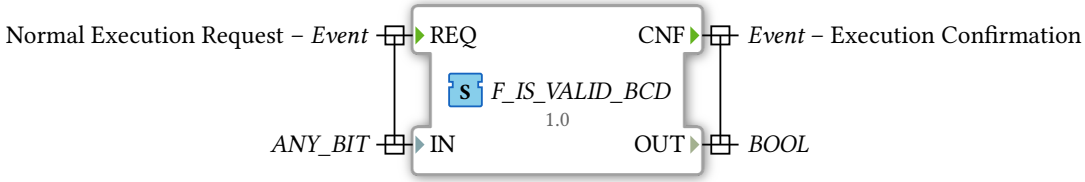
Comparison with similar function blocks

- F_IS_VALID_BCD: the same basic idea of validity checking, but for BCD-encoded bit strings instead of floating-point numbers.

Conclusion

`F_IS_VALID` provides a simple, central validity check for floating-point values and helps catch `NaN`/`Infinity`-related follow-on errors in calculation chains early.

F_IS_VALID_BCD



• – ◦ • – ◦ • – ◦ •

Introduction

`F_IS_VALID_BCD` checks whether a bit string contains valid BCD (Binary Coded Decimal) data — i.e. whether every nibble (4-bit group) represents a value between `0` and `9`. It validates BCD-encoded values before they are converted.

Interface Structure

Event Inputs

- **REQ**: Triggers the check, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **IN** (ANY_BIT): The bit string to check (`BYTE`, `WORD`, `DWORD`, or `LWORD`).

Data Outputs

- **OUT** (BOOL): `TRUE` if every nibble of `IN` holds a valid BCD value (`0`–`9`); `FALSE` if at least one nibble holds an invalid value (`10`–`15`, i.e. `A`–`F`).

Functionality

On `REQ`, the algorithm's ST statement `OUT := IS_VALID_BCD(IN)` checks nibble by nibble whether every 4-bit group of `IN` represents a valid decimal digit (`0`–`9`), and outputs the result via `OUT`. `CNF` is then triggered.

Technical Features

- **`ANY_BIT` input**: Accepts bit strings of any width (`BYTE`, `WORD`, `DWORD`, `LWORD`); the number of nibbles checked follows from their size.
- **Pre-check for BCD conversion**: A useful companion to the `*_BCD_TO_*` conversion blocks (cf. `F_BYTE_BCD_TO_USINT` etc.), catching invalid input data before the actual conversion.

State Overview

Stateless: every `REQ` immediately results in the check and `CNF`.

Application Scenarios

- **Validating fieldbus/protocol data** transmitted in BCD encoding, before converting it to a numeric value.
- **Error detection** for incorrectly transmitted or misinterpreted bit strings meant to be interpreted as BCD.

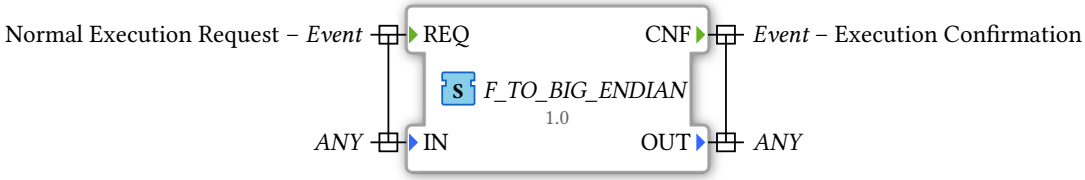
Comparison with similar function blocks

- F_IS_VALID: the same basic idea of validity checking, but for floating-point numbers instead of BCD bit strings.
- **`F_BYTE_BCD_TO_USINT`** and related conversion blocks: implicitly rely on the same BCD validity assumption; `F_IS_VALID_BCD` makes the check explicit and usable upstream.

Conclusion

`F_IS_VALID_BCD` provides a simple upstream check for BCD-encoded bit strings and protects downstream BCD conversion blocks from invalid input data.

F_TO_BIG_ENDIAN



• – ◦ • – ◦ • – ◦ •

Introduction

`F_TO_BIG_ENDIAN` converts a value to big-endian (most significant byte first) byte order. It belongs to the family of endian conversion functions needed for cross-platform exchange of binary data (e.g. over network protocols or with third-party systems).

Interface Structure

Event Inputs

- **REQ**: Triggers the conversion, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **IN** (ANY): The value to convert, of arbitrary data type.

Data Outputs

- **OUT** (ANY): The converted value, same data type as `IN`.

Functionality

On `REQ`, the block converts the natively encoded value `IN` into a big-endian (most significant byte first) byte sequence. The result is output via `OUT`, then `CNF` is triggered. The bit width of the data type stays unchanged — only the byte order is adjusted.

Technical Features

- **Generic `ANY` data type**: `IN`/`OUT` adapt to whichever concrete data type is connected at compile time; the byte count follows from its size.
- **Pure byte reordering, no value reinterpretation**: The block only changes the arrangement of the bytes, not their meaning or sign.

State Overview

Stateless: every `REQ` immediately results in the conversion and `CNF`.

Application Scenarios

- **Network communication**: Many network protocols (e.g. TCP/IP headers) use big-endian ("network byte order"); `F_TO_BIG_ENDIAN` ensures the correct byte order when sending/receiving.
- **Interoperability with third-party systems**: Exchanging binary data with systems whose native byte order differs from the target platform's.

Comparison with similar function blocks

- **`F_TO_LITTLE_ENDIAN`**: the same conversion direction, but for the other byte order.
- **`F_FROM_BIG_ENDIAN`**: the reverse direction for the same byte order.

Conclusion

`F_TO_BIG_ENDIAN` provides a simple, generically typed byte-order conversion and is an important building block for correct interoperability with network protocols and third-party systems of differing endianness.

F_TO_LITTLE_ENDIAN



• – ◦ • – ◦ • – ◦ •

Introduction

`F_TO_LITTLE_ENDIAN` converts a value to little-endian (least significant byte first) byte order. It belongs to the family of endian conversion functions needed for cross-platform exchange of binary data (e.g. over network protocols or with third-party systems).

Interface Structure

Event Inputs

- **REQ**: Triggers the conversion, carries `IN`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **IN** (ANY): The value to convert, of arbitrary data type.

Data Outputs

- **OUT** (ANY): The converted value, same data type as `IN`.

Functionality

On `REQ`, the block converts the natively encoded value `IN` into a little-endian (least significant byte first) byte sequence. The result is output via `OUT`, then `CNF` is triggered. The bit width of the data type stays unchanged — only the byte order is adjusted.

Technical Features

- **Generic `ANY` data type**: `IN`/`OUT` adapt to whichever concrete data type is connected at compile time; the byte count follows from its size.
- **Pure byte reordering, no value reinterpretation**: The block only changes the arrangement of the bytes, not their meaning or sign.

State Overview

Stateless: every `REQ` immediately results in the conversion and `CNF`.

Application Scenarios

- **Network communication**: Many network protocols (e.g. TCP/IP headers) use big-endian ("network byte order"); `F_TO_LITTLE_ENDIAN` ensures the correct byte order when sending/receiving.
- **Interoperability with third-party systems**: Exchanging binary data with systems whose native byte order differs from the target platform's.

Comparison with similar function blocks

- **`F_TO_BIG_ENDIAN`**: the same conversion direction, but for the other byte order.
- **`F_FROM_LITTLE_ENDIAN`**: the reverse direction for the same byte order.

Conclusion

`F_TO_LITTLE_ENDIAN` provides a simple, generically typed byte-order conversion and is an important building block for correct interoperability with network protocols and third-party systems of differing endianness.

BCD Conversions

🎧 Podcast

- The VBCD Valve Engineering Datasheet Decoded
- The VBCD DE-A Valve: How an Invisible Hero Safely Controls Cranes, Excavators, and More
- The VBCD DE-A Valve: Unpacking the Silent Guardian of Hydraulic Safety and Precision
- VBCD DE A: The Hydraulic Heart for Safety and Control of Extreme Cranes
- VBCD DE A [illegible]

F_BYTE_BCD_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_BCD_TO_USINT` converts a BCD-encoded BYTE value to a USINT value. BCD (Binary Coded Decimal) is an encoding in which each digit of a decimal number is represented by 4 bits. This function block is particularly useful in applications that need to process BCD-encoded data.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the algorithm for converting the BCD-encoded BYTE value to a USINT value is executed.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is activated once the conversion is complete.

Data Inputs

- **IN**: The BCD-encoded BYTE value to be converted to a USINT value.

Data Outputs

- **OUT**: The resulting USINT value after the conversion.

Adapters

- This function block has no adapters.

Functionality

The function block performs the conversion of a BCD-encoded BYTE value to a USINT value as soon as the event `REQ` is triggered. The conversion is performed using the built-in function `BYTE_BCD_TO_USINT`. After successful conversion, the event `CNF` is triggered to signal the completion of the operation.

Technical Features

- Conversion occurs directly and without any additional delay.
- The function block is optimized for processing BCD-encoded data.

State Overview

1. **Idle State**: The function block waits for the `REQ` event.
2. **Conversion State**: Conversion is performed when `REQ` is triggered.
3. **Completion State**: After successful conversion, `CNF` is triggered, and the function block returns to the idle state.

Application Scenarios

- Processing of BCD-encoded data in industrial control systems.

- Conversion of BCD-encoded sensor data into numerical values.
- Integration into control logic that needs to process BCD data.

⚖️ Comparison with Similar Function Blocks

- Compared to generic conversion function blocks, `F_BYTE_BCD_TO_USINT` specializes in converting BCD-encoded BYTE values to USINT values.
- Other function blocks might offer additional conversion options, but they may be less efficient for this specific use case.

Conclusion

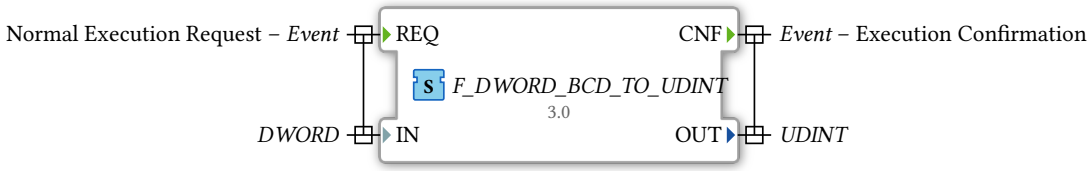
The `F_BYTE_BCD_TO_USINT` function block provides an efficient and specialized solution for converting BCD-encoded BYTE values to USINT values. Its simple and straightforward operation makes it ideal for applications requiring fast and reliable conversion.

F_DWORD_BCD_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_BCD_TO_UDINT` converts a BCD-encoded DWORD value to a UDINT value. BCD (Binary Coded Decimal) is an encoding method where each digit of a decimal number is represented by four bits. This function block is particularly useful in applications that need to process BCD-encoded data.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- `IN` (DWORD): The BCD-encoded input value to be converted to a UDINT value.

Data Outputs

- `OUT` (UDINT): The converted UDINT value.

Adapters

This function block has no adapters.

Operation

The function block performs the conversion when the `REQ` event is triggered. The conversion is performed using the function `DWORD_BCD_TO_UDINT`, which converts the BCD-encoded value `IN` into a UDINT value. After the conversion is complete, the event `CNF` is triggered, and the converted value is available at the output `OUT`.

Technical Features

- The function block has a simple structure and contains only one algorithm for the conversion.
- The conversion is performed directly without additional state management.

State Overview

The function block has no internal state. The conversion is performed anew with each `REQ` event.

Application Scenarios

- Processing of BCD-encoded data from external devices or sensors.
- Conversion of BCD-encoded values to UDINT format for further calculations.

⚖️ Comparison with similar function blocks

- Compared to generic conversion function blocks, `F_DWORD_BCD_TO_UDINT` specializes in converting BCD-encoded DWORD values to UDINT values.
- Other conversion function blocks might offer additional features or flexibility, but may be less efficient for this specific use case.

Conclusion

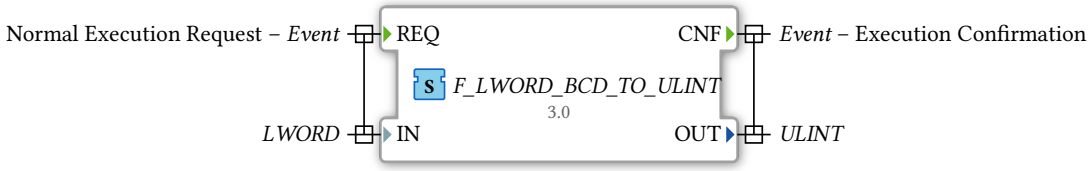
The `F_DWORD_BCD_TO_UDINT` function block is an efficient tool for the specific task of BCD-to-UDINT conversion. Its simplicity and straightforward operation make it a reliable solution for applications requiring this type of conversion.

F_LWORD_BCD_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LWORD_BCD_TO_ULINT` converts a BCD-encoded `LWORD` value to a `ULINT` value. This is particularly useful in applications where BCD-encoded data needs to be processed, such as in industrial control systems or communication protocols.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input for the BCD-encoded `LWORD` value to be converted.

Data Outputs

- **OUT**: The output for the converted `ULINT` value.

Adapters

No adapters are present.

Operation

The function block performs the conversion when the `REQ` event occurs. The BCD-encoded value `LWORD` at input `IN` is converted into a value `ULINT` and made available at output `OUT`. The `CNF` event signals successful conversion.

Technical Features

- The function block uses the BCD-to-ULINT conversion algorithm defined in the IEC 61131-3 standard library.
- The conversion occurs synchronously with the `REQ` event.

State Overview

1. **Idle State**: The function block waits for the `REQ` event.

2. **Conversion State**: Upon receipt of `REQ`, the conversion is performed and the `CNF` event is triggered.

Application Scenarios

- Processing of BCD-encoded data in industrial controllers.
- Conversion of BCD-encoded values to numeric values for further calculations.

⚖️ Comparison with Similar Function Blocks

- **F_BCD_TO_ULINT**: Converts BCD-encoded values to `ULINT`, but for different data types than `LWORD`.
- **F_LWORD_TO_ULINT**: Converts `LWORD` to `ULINT`, but without BCD encoding.

Conclusion

The `F_LWORD_BCD_TO_ULINT` function block is a specialized building block for converting BCD-encoded `LWORD` values to `ULINT`. It is easy to use and particularly suitable for applications that work with BCD-encoded data.

F_UDINT_TO_BCD_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UDINT_TO_BCD_DWORD` converts an unsigned 32-bit integer value (UDINT) into a BCD-encoded DWORD. This conversion is particularly useful in applications where numeric values need to be represented or processed in a human-readable format.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: An unsigned 32-bit integer value (UDINT) to be converted into a BCD-encoded DWORD.

Data Outputs

- **OUT**: The result of the conversion as a BCD-encoded DWORD.

Adapters

No adapters are available.

Functionality

The function block performs the conversion as soon as the event `REQ` is received. The algorithm converts the value at input `IN` from UDINT to a BCD-encoded DWORD and outputs the result to `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

Technical Details

- The conversion is performed using the function `UDINT_TO_BCD_DWORD`, which is implemented in the function block's algorithm.
- The function block is part of the package `iec61131::conversion`.

State Overview

1. **Wait State**: The function block waits for the event `REQ`.

2. **Active State**: Upon receiving `REQ`, the conversion is performed and the result is output to `OUT`.
3. **Confirmation State**: After successful conversion, `CNF` is triggered, and the function block returns to the wait state.

Application Scenarios

- **Display Control**: Converting numeric values for display on seven-segment displays or LCDs.
- **Data Communication**: Preparing data for transmission in BCD-encoded protocols.
- **Printer Control**: Converting numeric values for printing in a readable format.

⚖️ Comparison with Similar Function Blocks

- **F_UDINT_TO_STRING**: Converts a UDINT to a string, but is less efficient for BCD-specific applications.
- **F_UDINT_TO_DWORD**: Performs a direct type conversion without BCD encoding.

Conclusion

The function block `F_UDINT_TO_BCD_DWORD` provides an efficient and specialized solution for converting unsigned 32-bit integers to BCD-encoded DWORDs. Its simple interface and reliable operation make it ideal for applications that require BCD encoding.

F_UINT_TO_BCD_WORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_BCD_WORD` converts an unsigned integer value (UINT) into a BCD-encoded word (WORD). BCD (Binary Coded Decimal) is a method for representing decimal numbers where each digit is encoded by 4 bits. This function block is particularly useful in applications where numerical values need to be output in a human-readable format, such as in display systems or printers.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The unsigned integer value (UINT) to be converted.

Data Outputs

- **OUT**: The result of the conversion as a BCD-encoded word (WORD).

Adapters

This function block does not use adapters.

Operation

When the event input `REQ` is triggered, the algorithm is executed that converts the value of `IN` into a BCD-encoded `WORD` and outputs the result to `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

Technical Features

- The function block implements direct UINT to WORD conversion using BCD encoding.

- The conversion is performed in a single algorithm step, ensuring high efficiency.

State Overview

The function block has no internal states other than the initial readiness to process the `REQ` event.

Application Scenarios

- Display controllers that require numeric values in BCD format.
- Printers and other output devices that process BCD-encoded data.
- Systems that require compatibility with legacy BCD-based components.

⚖️ Comparison with Similar Building Blocks

Compared to generic conversion building blocks, `F_UINT_TO_BCD_WORD` offers a specialized solution for BCD encoding that can be more efficient and easier to use than general-purpose conversion methods.

Conclusion

The `F_UINT_TO_BCD_WORD` function block is a specialized tool for the efficient conversion of unsigned integer values to the BCD format. Its simple interface and straightforward operation make it ideal for applications requiring fast and reliable BCD encoding.

F_ULINT_TO_BCD_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_TO_BCD_LWORD` converts a `ULINT` value (64-bit unsigned integer) to a `LWORD` value type in BCD (Binary Coded Decimal) format. This block is particularly useful in applications where numeric values need to be converted into a format compatible with BCD systems.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. The value to be converted is passed via the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is output via the data output `OUT`.

Data Inputs

- **IN**: The input for the `ULINT` value to be converted.

Data Outputs

- **OUT**: The output for the converted `LWORD` value type in BCD format.

Adapters

This function block does not have any adapters.

Functionality

The function block performs the conversion from `ULINT` to `LWORD` in BCD format as soon as the `REQ` event is triggered. The conversion is performed using the integrated function `ULINT_TO_BCD_LWORD`. Once the conversion is complete, the `CNF` event is triggered, and the converted value is available at the `OUT` output.

Technical Features

- Supports the conversion of 64-bit unsigned integer (`ULINT`) to 64-bit BCD (`LWORD`).
- No additional parameters or settings are required.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Conversion**: Performs the conversion as soon as `REQ` is activated.
3. **Completion**: Sends the `CNF` event and outputs the converted value.

Application Scenarios

- Industrial control systems that process BCD-encoded data.
- Communication with peripheral devices that expect BCD data.
- Data conversion in automation applications.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_ULINT_TO_BCD_LWORD` offers a specialized solution for conversion to the BCD format.
- Other blocks might offer additional conversion options, but they may be less efficient for this specific use case.

Conclusion

The `F_ULINT_TO_BCD_LWORD` function block is an efficient and specialized solution for converting `ULINT` to `LWORD` in BCD format. Its ease of use and direct functionality make it ideal for applications requiring fast and reliable data conversion.

F_USINT_TO_BCD_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_USINT_TO_BCD_BYTE` converts an unsigned 8-bit integer value (USINT) into a BCD-encoded byte (BYTE). BCD (Binary Coded Decimal) is an encoding in which each decimal digit is represented by four bits.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: An unsigned 8-bit integer value (USINT) to be converted to BCD.

Data Outputs

- **OUT**: The result of the conversion as a BCD-encoded byte (BYTE).

Adapters

No adapters are present.

Functionality

The function block performs the conversion when the event `REQ` occurs. The algorithm converts the input value `IN` from USINT to a BCD-encoded byte and stores the result in `OUT`. The event `CNF` is then triggered to signal the completion of the conversion.

Technical Features

- The function block is deterministic and does not perform any internal state changes.
- The conversion is performed directly via the function `USINT_TO_BCD_BYTE`.

State Overview

The function block has no internal state. The conversion is performed anew with each `REQ` event.

Application Scenarios

- Display control for seven-segment displays.
- Communication with devices that expect BCD-encoded data.
- Data conversion in control systems that require the BCD format.

⚖️ Comparison with Similar Function Blocks

- Unlike generic conversion blocks, `F_USINT_TO_BCD_BYTE` is specialized for converting USINT to BCD.

Other blocks might offer additional features such as error handling or different encodings, but are often more complex.

Conclusion

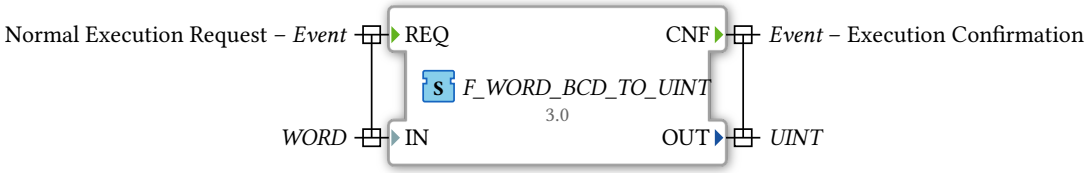
The function block `F_USINT_TO_BCD_BYTE` is an efficient tool for the specific task of converting USINT to BCD. Its simplicity and determinism make it a reliable choice in control applications that require BCD-encoded data.

F_WORD_BCD_TO_UINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_BCD_TO_UINT` converts a BCD-encoded WORD value into an unsigned integer (UINT). This is particularly useful in applications where data in BCD format needs to be processed or displayed.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs

- `IN` (WORD): The BCD-encoded input value to be converted.

Data Outputs

- `OUT` (UINT): The converted unsigned integer value.

Adapters

This function block does not use adapters.

Operation

When the `REQ` event is triggered, the BCD-encoded value at the input `IN` is converted to an unsigned integer value. The result is output at `OUT`, and the `CNF` event signals the completion of the conversion.

Technical Features

- The function block is simple and direct, without complex state logic.
- The conversion is performed using the function `WORD_BCD_TO_UINT`.

State Overview

Since it is a simple function block, there are no complex states. The block reacts directly to the `REQ` event with a conversion and the output of the result.

Application Scenarios

- Conversion of BCD-encoded data from external devices or sensors.
- Use in control systems that need to process BCD data.
- Display of BCD data in a human-readable format.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_WORD_BCD_TO_UINT` specializes in BCD to UINT conversion.
- Other blocks might support additional features such as error handling or different encoding formats, but they are more complex.

Conclusion

The `F_WORD_BCD_TO_UINT` function block is an efficient tool for the specific task of BCD to UINT conversion. Its simplicity and straightforward operation make it ideal for applications requiring fast and reliable conversions.

BOOL

BOOL conversions

🎧 Podcast

- George Boole: How an autodidact laid the foundation of our digital world with 0 and 1

🛠️ Related exercises

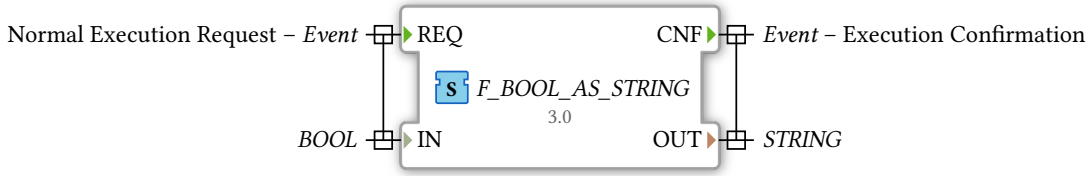
- Uebung_001_AX_b
- Uebung_002a2_AX
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b3
- Uebung_004b_AX
- Uebung_006a3
- LinksRechts_T_FF_Event
- Uebung_035a2
- Uebung_035a3
- SoftkeyOverride_TO_QX_BG
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8
- Uebung_094a_AX
- Uebung_160

F_BOOL_AS_STRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BOOL_AS_STRING` is used to convert a Boolean value (`BOOL`) into a string (`STRING`). This functionality is particularly useful when Boolean values are needed for output or further processing in a text format.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs

- **IN**: The Boolean value (`BOOL`) to be converted into a string.

Data Outputs

- **OUT**: The resulting string (`STRING`) representing the converted Boolean value.

Adapters

No adapters are defined for this function block.

Functionality

The function block uses a simple algorithm to convert the Boolean value `IN` into a string `OUT`. The conversion is performed by the function `BOOL_AS_STRING(IN)`, which internally converts the value of `IN` into a corresponding string (e.g., `TRUE` or `FALSE`).

Technical Features

- The function block is implemented simply and efficiently, without additional states or complex logic.
- The conversion occurs synchronously with the triggering of the `REQ` event.

State Overview

Since this is a simple function block, there are no explicit states. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- **Logging**: Conversion of Boolean values for logging purposes.
- **User Interfaces**: Display of Boolean values in a human-readable format.
- **Data Communication**: Preparation of Boolean values for transmission as text.

⚖️ Comparison with Similar Function Blocks

- **F_STRING_TO_BOOL**: Performs the reverse conversion from `STRING` to `BOOL`.
- **F_BOOL_TO_***: Other conversion blocks that convert Boolean values into various data types.

Conclusion

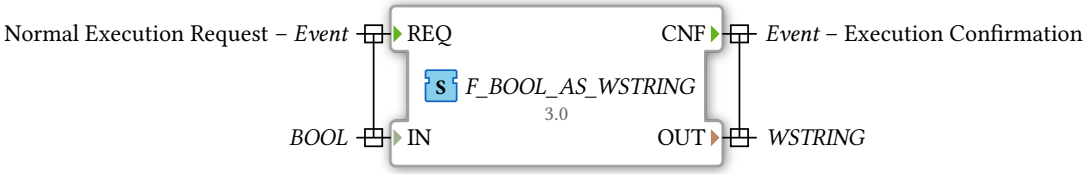
The `F_BOOL_AS_STRING` function block offers a simple and effective way to convert Boolean values to strings. Its clear interface and direct functionality make it a useful tool in many use cases.

F_BOOL_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BOOL_AS_WSTRING` converts a Boolean value (`BOOL`) into a wide string (`WSTRING`). This functionality is particularly useful when Boolean values need to be output or further processed in a human-readable format.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion of the Boolean input value `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion and returns the result `OUT`.

Data Inputs

- `IN` (`BOOL`): The Boolean value to be converted.

Data Outputs

- `OUT` (`WSTRING`): The converted value as a wide string.

Adapters

No adapters available.

Functionality

This function block converts the Boolean input value `IN` into a wide string (`WSTRING`). The conversion is performed using the function `BOOL_AS_WSTRING`, which converts the value `TRUE` to `"TRUE"` and `FALSE` to `"FALSE"`. The conversion is triggered by the event `REQ`, and the result is confirmed by the event `CNF`.

Technical Features

- The function block has a simple structure and contains only one conversion algorithm.
- The conversion is performed directly and without delay.

State Overview

The function block has no internal states. The conversion occurs immediately after the `REQ` event is triggered.

Application Scenarios

- Displaying Boolean values in user interfaces.
- Logging Boolean states to log files.
- Further processing of Boolean values in systems that expect strings.

⚖️ Comparison with Similar Function Blocks

- Unlike `F_BOOL_TO_STRING`, this function block generates a wide string (`WSTRING`) instead of a regular string (`STRING`).
- Similar function blocks like `F_INT_AS_WSTRING` or `F_REAL_AS_WSTRING` convert other data types to `WSTRING`.

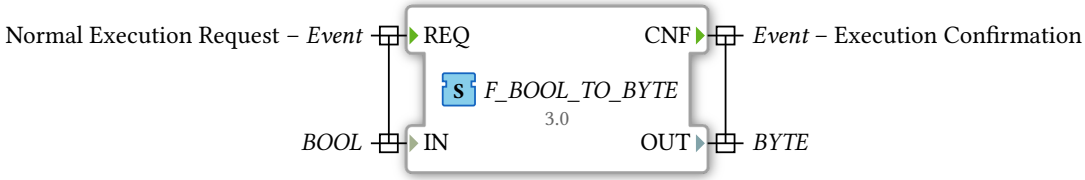
Conclusion

The function block `F_BOOL_AS_WSTRING` provides a simple and efficient way to convert Boolean values to wide strings. Its slim design makes it particularly suitable for applications where fast and uncomplicated conversion is required.

F_BOOL_TO_BYTE

• – ◦ • – ◦ • – ◦ •

The function block `F_BOOL_TO_BYTE` converts a Boolean value (`BOOL`) into a byte value (`BYTE`). This conversion is useful when Boolean signals need to be processed in systems that require byte data.



- **REQ**: This event input triggers the execution of the function block. It is linked to the data input `IN`.
- **CNF**: This event output signals the successful execution of the function block. It is linked to the data output `OUT`.
- **IN** (`BOOL`): The Boolean input value to be converted into a byte.
- **OUT** (`BYTE`): The resulting byte value after conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use any adapters.

Functionality

The function block performs the conversion from `BOOL` to `BYTE` as soon as the event `REQ` is triggered. The Boolean value `IN` is converted to a byte value, with `FALSE` becoming `0` and `TRUE` becoming `1`. The result is output via `OUT`, and the event `CNF` signals successful execution.

- The function block is a simple function block (`SimpleFB`) and uses an ST algorithm (`ST`) for the conversion.

The function block has no internal states. Execution occurs immediately upon triggering the `REQ` event.

- **Signal Processing**: Conversion of Boolean control signals into byte values for communication with devices that expect byte data.
- **Data Logging**: Conversion of Boolean status information into byte format for storage or transmission.
- **F_BOOL_TO_INT**: Converts a Boolean value to an integer value, while `F_BOOL_TO_BYTE` is specifically optimized for byte output.
- **F_BOOL_TO_WORD**: Similar to `F_BOOL_TO_BYTE`, but designed for larger data formats (Word).

The function block `F_BOOL_TO_BYTE` is a simple and efficient tool for converting Boolean values to byte data. It is particularly suitable for applications that require a compact representation of Boolean signals in byte format. Its simple interface and direct functionality make it easy to integrate into existing systems.

Technical Features

State Overview

Application Scenarios

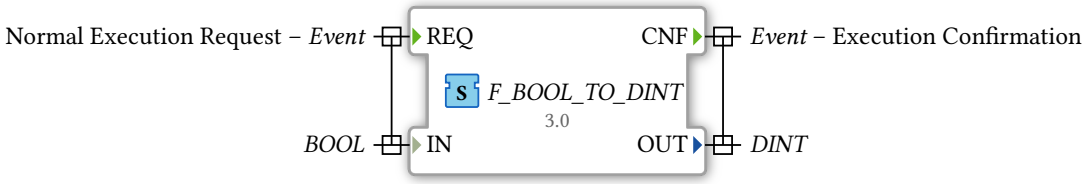
⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

F_BOOL_TO_DINT

• – ◦ • – ◦ • – ◦ •

The function block F_BOOL_TO_DINT converts a Boolean value (BOOL) into a 32-bit integer value (DINT). This conversion is useful when Boolean signals need to be integrated into numerical calculations or control systems.



- **REQ**: Starts the conversion. When this event is triggered, the Boolean input value (IN) is converted into a DINT value.
- **CNF**: Signals the completion of the conversion. This event is output along with the converted output value (OUT).
- **IN**: The Boolean input value (BOOL) to be converted to a DINT value.
- **OUT**: The converted 32-bit integer value (DINT). The value is 1 if IN is TRUE, otherwise 0.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use any adapters.

Functionality

The function block performs the conversion from BOOL to DINT by converting the Boolean value TRUE to 1 and FALSE to 0. The conversion is triggered by the event REQ, and the result is confirmed by the event CNF.

- The conversion is performed using the function BOOL_TO_DINT, which is implemented in the function block's algorithm.

The function block has no internal states. The conversion occurs immediately upon triggering the REQ event.

- Integration of Boolean signals into numerical controllers.
- Conversion of switching states (TRUE/FALSE) into numerical values for further calculations.
- Interfaces between Boolean and numerical system components.
- **F_BOOL_TO_INT**: Converts BOOL to a 16-bit integer (INT). In contrast, F_BOOL_TO_DINT returns a 32-bit integer.
- **F_BOOL_TO_REAL**: Converts BOOL to a floating-point value (REAL). F_BOOL_TO_DINT, on the other hand, returns an integer value.

The function block F_BOOL_TO_DINT is a simple and efficient tool for converting Boolean values to 32-bit integers. It is particularly suitable for applications where Boolean signals need to be integrated into numerical controllers. Its deterministic operation and clear interface structure make it a reliable component in IEC 61499-based automation.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

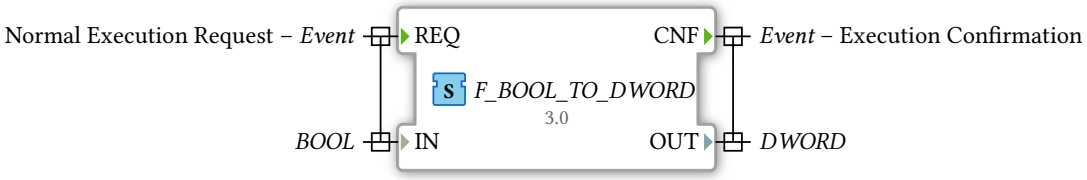
Conclusion

F_BOOL_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BOOL_TO_DWORD` converts a Boolean value (`BOOL`) into a 32-bit unsigned integer value (`DWORD`). This block is particularly useful in scenarios where Boolean signals need to be converted into numerical values, for example, for communication with systems that can only process numeric input.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The Boolean value (`BOOL`) to be converted to `DWORD`.

Data Outputs

- **OUT**: The converted 32-bit unsigned integer value (`DWORD`). The value is either `1` (TRUE) or `0` (FALSE).

Adapters

- No adapters are available.

**

Functionality

The function block performs the conversion as soon as the event `REQ` is received. The Boolean value `IN` is converted to `DWORD`:

- `TRUE` becomes `1`
- `FALSE` becomes `0`

After successful conversion, the event `CNF` is triggered, and the converted value is available at output `OUT`.

Technical Features

- The conversion occurs instantly and without delay.
- The block is deterministic and has no internal state.

State Overview

This function block has no internal states. The conversion is performed anew with each `REQ` event.

Application Scenarios

- Conversion of switching states (On/Off) into numerical control signals.

- Integration into control systems that can only process numerical inputs.
- Logging of Boolean events as numerical values.

⚖️ Comparison with Similar Blocks

- Unlike `F_BOOL_TO_INT` or `F_BOOL_TO_REAL`, this block specifically converts to a `DWORD` type, which is optimized for 32-bit systems.
- Similar blocks, such as `F_DWORD_TO_BOOL`, perform the reverse conversion.

Conclusion

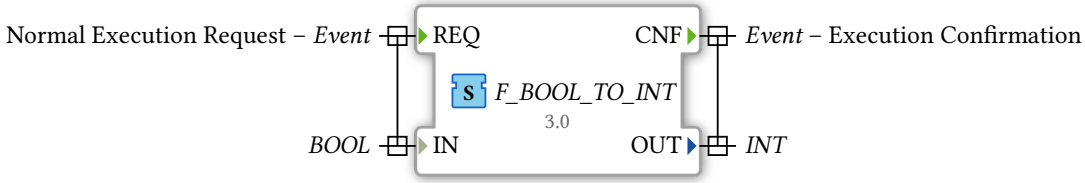
The `F_BOOL_TO_DWORD` function block is a simple and efficient building block for converting Boolean values to `DWORD`. Its deterministic functionality and clear interface make it a reliable component in control and communication applications.

F_BOOL_TO_INT

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_BOOL_TO_INT` converts a Boolean value (`BOOL`) into an integer value (`INT`). Specifically, `TRUE` is converted to `1`, and `FALSE` to `0`. This block is particularly useful in scenarios where Boolean logic needs to be integrated into numerical calculations.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. The input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The Boolean value (`BOOL`) to be converted.

Data Outputs

- **OUT**: The converted integer value (`INT`). `TRUE` becomes `1`, and `FALSE` becomes `0`.

Adapter

- No adapters available.

Functionality

When the event `REQ` is triggered, the Boolean value at input `IN` is read and converted into an integer value. The result is output at `OUT`, and the event `CNF` is triggered to signal the completion of the operation.

Technical Features

- The block is a simple function block (`SimpleFB`) without internal states.
- The conversion is performed directly and without delay.

State Overview

Because it is a simple function block, there are no state transitions or internal states. The operation is re-executed with each `REQ` event.

Application Scenarios

- Integration of Boolean logic into numerical control systems.

- Conversion of switching states (`TRUE`/`FALSE`) into numerical values for further calculations.
- Use in control engineering, where Boolean signals need to be converted into numerical control signals.

⚖️ Comparison with Similar Function Blocks

- **F_BOOL_TO_REAL**: Converts `BOOL` to `REAL` (floating-point number), similar to `F_BOOL_TO_INT`, but with floating-point output.
- **F_INT_TO_BOOL**: Performs the reverse conversion (`INT` to `BOOL`).

Conclusion

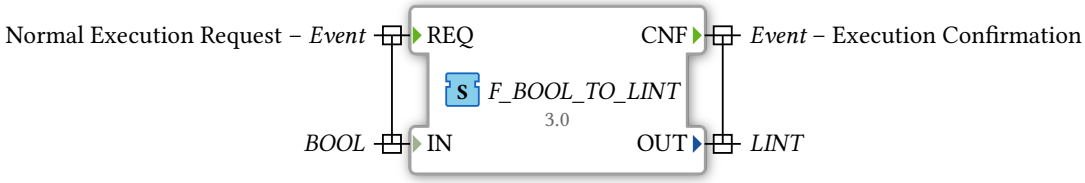
The `F_BOOL_TO_INT` function block is a simple yet effective tool for converting Boolean values to integer values. Its simplicity and direct operation make it ideal for applications requiring fast and straightforward conversion.

F_BOOL_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BOOL_TO_LINT` converts a Boolean value (`BOOL`) into a 64-bit integer value (`LINT`). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: This input triggers the execution of the function block. It is linked to the data input `IN`.

Event Outputs

- **CNF**: This output signals the successful execution of the function block. It is linked to the data output `OUT`.

Data Inputs

- **IN**: The input for the Boolean value (`BOOL`) to be converted.

Data Outputs

- **OUT**: The output for the converted 64-bit integer value (`LINT`).

Adapters

This function block has no adapters.

Functionality

The function block performs the conversion from `BOOL` to `LINT` when the event `REQ` is received. The conversion is performed using the function `BOOL_TO_LINT(IN)`, where the Boolean value `IN` is converted into a value `LINT`. The result is output at `OUT`, and the event `CNF` is triggered.

Technical Features

- The block is a simple function block (`SimpleFB`) and uses an ST algorithm for the conversion.
- The conversion is deterministic and occurs without delay.

State Overview

The function block has no internal state. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control applications where Boolean values need to be converted to numeric values.
- Integration into larger systems that require consistent data type handling.

⚖️ Comparison with Similar Blocks

- Compared to other conversion blocks such as `F_BOOL_TO_INT` or `F_BOOL_TO_DINT`, this block offers higher precision through the use of the `LINT` data type (64-bit).

Conclusion

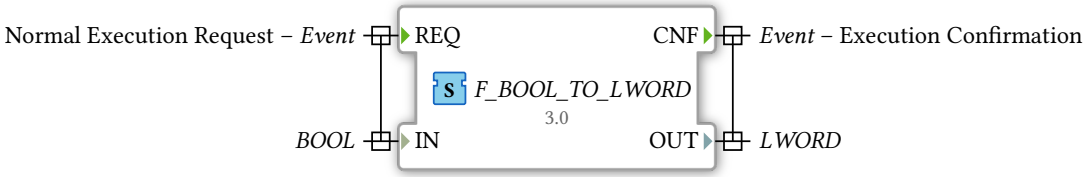
The `F_BOOL_TO_LINT` function block is an efficient tool for converting Boolean values to 64-bit integer values. Its simple and deterministic operation makes it a reliable component in control applications.

F_BOOL_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BOOL_TO_LWORD` converts a Boolean value (`BOOL`) into a 64-bit unsigned integer (`LWORD`). This conversion is particularly useful in scenarios where Boolean values need to be embedded in larger data structures or processed with other long integer values.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion. Linked to the data input `IN`.

Event Outputs

- **CNF**: Execution Confirmation. Triggered after successful conversion. Linked to the data output `OUT`.

Data Inputs

- **IN**: The Boolean value (`BOOL`) to be converted to `LWORD`.

Data Outputs

- **OUT**: The converted 64-bit unsigned integer value (`LWORD`).

Adapters

This function block does not use adapters.

Adapters ... ## Functionality

This function block converts `BOOL` to `LWORD` when the `REQ` event occurs. The Boolean value `IN` is converted to `LWORD`, while `FALSE` is converted to `0` and `TRUE` to `1`. The result is output at the data output `OUT`, and the `CNF` event is triggered to indicate successful execution.

Technical Features

- The conversion is performed using the function `BOOL_TO_LWORD`, which converts the Boolean value into a 64-bit value.
- The function block is simple and deterministic because it does not store any internal states.

State Overview

Since it is a simple function block without internal states, there are no state transitions or management.

Application Scenarios

- Embedding Boolean values in larger data structures that use `LWORD`.

- Use in systems that need to process Boolean values as long integers, e.g., when communicating with external systems or storing them in databases.

⚖️ Comparison with similar function blocks

- **F_BOOL_TO_**: Similar function blocks exist for other data types (e.g., `F_BOOL_TO_BYTE`, `F_BOOL_TO_WORD`). These convert Boolean values to other integer data types with different bit widths.
- **F__TO_LWORD**: Other conversion blocks convert various data types (e.g., `INT`, `DINT`) to `LWORD`, but `F_BOOL_TO_LWORD` is specific to Boolean values.

Conclusion

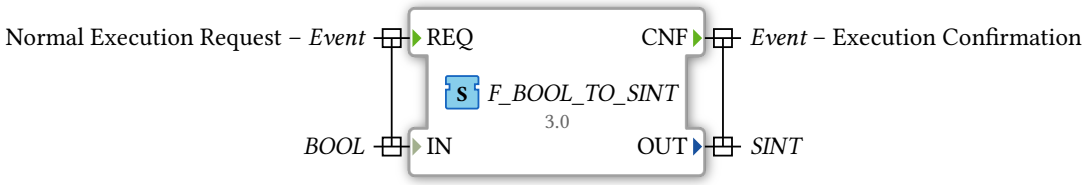
The `F_BOOL_TO_LWORD` function block offers a simple and efficient way to convert Boolean values into 64-bit integer values. Its deterministic operation and clear interface structure make it a useful component in control and automation systems, especially where Boolean values need to be integrated into larger data structures.

F_BOOL_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BOOL_TO_SINT` converts a Boolean value (`BOOL`) into a signed 8-bit integer (`SINT`). This conversion is useful when Boolean values are used in calculations or controls that require integer values.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event input is triggered, the Boolean value at data input `IN` is converted to a value `SINT`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event is output along with the converted value at data output `OUT`.

Data Inputs

- **IN**: The Boolean value (`BOOL`) to be converted to a value `SINT`.

Data Outputs

- **OUT**: The converted value (`SINT`). The output value is `1` if the input is `IN` or `TRUE`; otherwise, it is `0`.

Adapters

This function block does not use any adapters.

Functionality

The function block converts the Boolean input value `IN` to a value `SINT`. The conversion is performed according to the following rule:

- `TRUE` is converted to `1`.
- `FALSE` is converted to `0`.

The conversion is triggered by the event `REQ`. After successful conversion, the event `CNF` is triggered, and the converted value is output at `OUT`.

Technical Features

- The function block is a simple algorithm (`SimpleFB`) that performs the conversion directly.
- The conversion occurs without delay as soon as the event `REQ` arrives.

State Overview

The function block has no internal state. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of Boolean signals into numerical values for mathematical operations.
- Integration of Boolean control signals into systems that expect integer values.

⚖️ Comparison with Similar Function Blocks

- **F_BOOL_TO_INT**: Converts `BOOL` to a 16-bit integer (`INT`). This block is similar, but the output value has a larger bit width.
- **F_BOOL_TO_USINT**: Converts `BOOL` to an unsigned 8-bit integer (`USINT`). The difference lies in the interpretation of the output value (signed vs. unsigned).

Conclusion

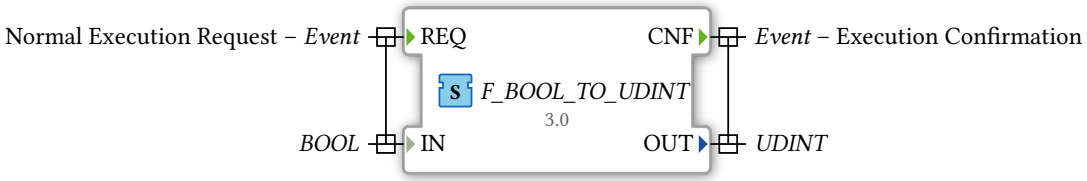
The function block `F_BOOL_TO_SINT` is a simple and efficient tool for converting Boolean values to signed 8-bit integers. It is particularly useful in control systems where Boolean signals need to be converted into numerical values.

F_BOOL_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BOOL_TO_UDINT` converts a `BOOL` value to a `UDINT` value. This conversion is useful when Boolean values are used in calculations or storage operations that require a larger numeric data type.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `BOOL` to be converted.

Data Outputs

- **OUT**: The output of type `UDINT`, containing the result of the conversion.

Adapters

No adapters are supported.

Functionality

The function block performs the conversion from `BOOL` to `UDINT` when the event `REQ` is triggered. The Boolean value `TRUE` is converted to `1`, and `FALSE` to `0`. The result is output at data output `OUT`, and the event `CNF` signals successful execution.

Technical Features

- The conversion is instantaneous and without delay.
- The function block is suitable for use in real-time systems.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of Boolean switching states for numerical calculations.
- Integration of Boolean signals into systems that can only process numeric values.

⚖️ Comparison with similar building blocks

- Compared to other conversion blocks such as `F_BOOL_TO_INT` or `F_BOOL_TO_DINT`, this block offers conversion to an unsigned 32-bit integer.

Conclusion

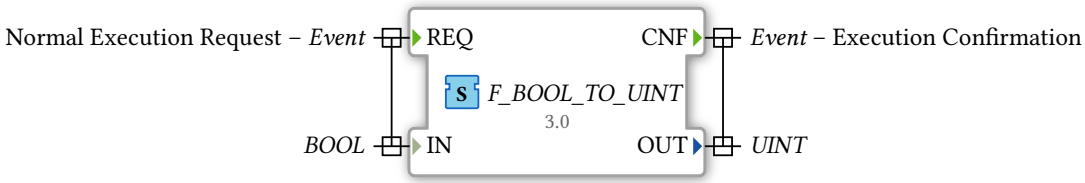
The `F_BOOL_TO_UDINT` function block is a simple and efficient building block for converting Boolean values to unsigned 32-bit integers. It is particularly useful in scenarios where Boolean signals need to be integrated into numerical processes.

F_BOOL_TO_UINT

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_BOOL_TO_UINT` converts a Boolean value (`BOOL`) into an unsigned integer value (`UINT`). This conversion is particularly useful when Boolean signals need to be converted into numerical values, for example, for further calculations or control purposes.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN**: The Boolean input value (`BOOL`) to be converted.

Data Outputs

- **OUT**: The converted unsigned integer value (`UINT`).

Adapters

- No adapters are available.

Functionality

The function block performs the conversion by transforming the Boolean value `IN` into an unsigned integer value `OUT`. The conversion is performed according to the following rule:

- `FALSE` becomes `0`
- `TRUE` becomes `1`

The conversion is triggered by the event `REQ` and confirmed by the event `CNF`.

Technical Features

- The function block is a simple function block (`SimpleFB`) and performs the conversion in a single algorithm.
- The conversion is performed directly and without delay.

State Overview

The function block has no internal states. The conversion is performed anew with each `REQ` event.

Application Scenarios

- Conversion of Boolean sensor data into numerical values for further processing.
- Integration into control systems that require numerical values instead of Boolean values.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_BOOL_TO_INT` or `F_BOOL_TO_REAL` offer conversions to other data types, but `F_BOOL_TO_UINT` is specifically optimized for unsigned integer values.

🛠️ Related Exercises

- Exercise_087
- Exercise_087a1
- Exercise_090a1
- Exercise_090a1_AX
- Exercise_090a2
- Exercise_090a2_AX

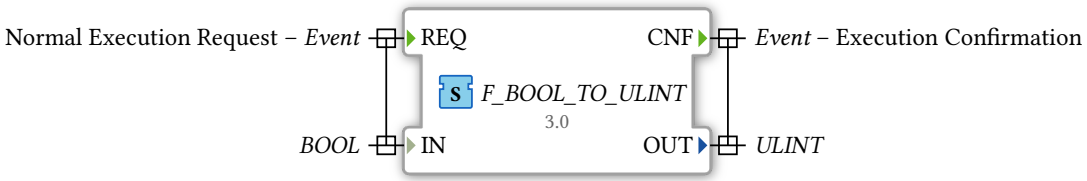
Conclusion

The `F_BOOL_TO_UINT` function block is a simple and efficient tool for converting Boolean values to unsigned integers. Its clear interface and direct functionality make it a reliable component in control and automation systems.

F_BOOL_TO_ULINT

• – ◦ • – ◦ • – ◦ •

The function block `F_BOOL_TO_ULINT` converts a Boolean value (`BOOL`) into an unsigned 64-bit integer value (`ULINT`). This conversion is particularly useful in scenarios where Boolean states need to be converted into numerical values, for example, for processing in mathematical operations or for storage in databases.



- **REQ**: Normal execution request. Triggers the conversion. Linked to the data input `IN`.
- **CNF**: Execution Confirmation. Triggered after successful conversion. Linked to the data output `OUT`.
- **IN**: Input for the Boolean value (`BOOL`) to be converted.
- **OUT**: Output for the converted unsigned 64-bit integer value (`ULINT`).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters available.

Functionality

This function block performs the conversion from `BOOL` to `ULINT` when the event `REQ` is received. The Boolean value `IN` is converted to a value `ULINT`, with `FALSE` becoming `0` and `TRUE` becoming `1`. The result is output at `OUT`, and the event `CNF` is triggered.

- The function block is a simple function block (`SimpleFB`) and uses an ST algorithm for the conversion.
- The conversion is performed using the function `BOOL_TO_ULINT`, which directly converts the Boolean value into a `ULINT` value.

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

- Conversion of Boolean switching states into numerical values for further calculations.
- Integration into control systems that need to store Boolean signals in numerical databases.
- Use in data processing chains that require type conversion.
- Similar function blocks like `F_BOOL_TO_INT` or `F_BOOL_TO_DINT` convert Boolean values to other numeric types, but not to `ULINT`.
- `F_BOOL_TO_ULINT` is specific to converting to the unsigned 64-bit type and is therefore suitable for applications requiring a large range of values.

The function block `F_BOOL_TO_ULINT` provides a simple and efficient way to convert Boolean values to unsigned 64-bit integers. Its clear interface and direct operation make it ideal for applications requiring high-precision type conversion with a large range of values.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

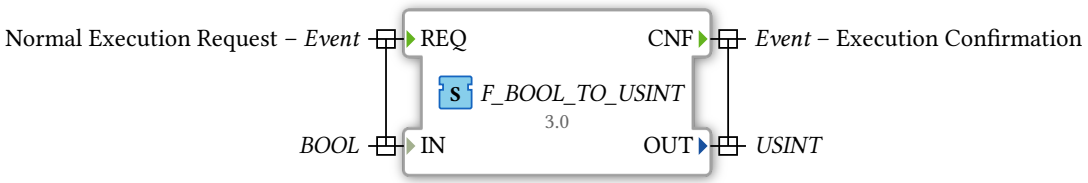
Conclusion

F_BOOL_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BOOL_TO_USINT` converts a Boolean value (`BOOL`) into an unsigned 8-bit integer (`USINT`). This conversion is useful when Boolean values need to be used in numerical operations or for communication with systems that do not directly support Boolean values.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs
- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs
- **IN**: The Boolean input value (`BOOL`) to be converted.

Data Outputs
- **OUT**: The converted unsigned 8-bit integer (`USINT`).

Adapters
This function block does not use adapters.

Functionality
The function block performs the conversion by transforming the Boolean value `IN` into a numeric value:

- `FALSE` is converted to `0`.
- `TRUE` is converted to `1`.

The conversion is triggered by the event `REQ`, and the result is output via the output `OUT`, accompanied by the confirmation event `CNF`.

Technical Features
- The function block is a simple function block (`SimpleFB`) and performs the conversion in a single algorithm.
- The conversion occurs directly and without delay as soon as the `REQ` event arrives.

State Overview
The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios
- Conversion of Boolean switching states into numerical values for processing in mathematical operations.
- Communication with systems or devices that do not directly support Boolean values but can process numerical values.

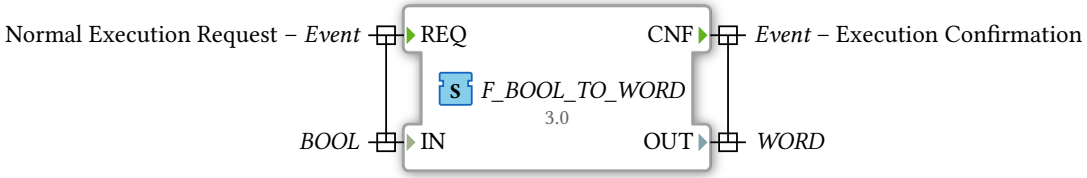
⚖️ Comparison with Similar Function Blocks
- Compared to other conversion function blocks such as `F_BOOL_TO_INT` or `F_BOOL_TO_UINT`, `F_BOOL_TO_USINT` is specifically optimized for conversion to an unsigned 8-bit integer.

Similar function blocks can use larger or signed integer types, which may be more or less suitable depending on the use case.

Conclusion
The `F_BOOL_TO_USINT` function block is a simple and efficient tool for converting Boolean values to unsigned 8-bit integers. Its direct and lag-free operation makes it ideal for real-time applications where fast and reliable conversions are required.

F_BOOL_TO_WORD
• – ◦ • – ◦ • – ◦ •

Introduction
The function block `F_BOOL_TO_WORD` converts a Boolean value (`BOOL`) into a 16-bit word value (`WORD`). This conversion is particularly useful in scenarios where Boolean signals need to be embedded in or further processed within word data structures.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion. Linked to the data input `IN`.

Event Outputs

- **CNF**: Execution Confirmation. Triggered after successful conversion. Linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `BOOL` to be converted to `WORD`.

Data Outputs

- **OUT**: The output value of type `WORD` containing the result of the conversion.

Adapters

- No adapters available.

Functionality

This function block performs the conversion of `BOOL` to `WORD` when the event `REQ` is received. The Boolean value `IN` is converted into a 16-bit word, with `FALSE` being interpreted as `0` and `TRUE` as `1`. The result is output via `OUT`, and the event `CNF` is triggered to indicate successful conversion.

Technical Features

- The function block is a simple function block (`SimpleFB`) and uses an ST algorithm for conversion.
- The conversion is performed directly by the function `BOOL_TO_WORD(IN)`.

State Overview

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Embedding Boolean signals in word data structures for communication with other systems.
- Use in control logic where Boolean values need to be processed as part of larger data words.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_BOOL_TO_BYTE` or `F_BOOL_TO_DWORD` offer conversions to other data types (`BYTE` or `DWORD`). `F_BOOL_TO_WORD` is specific to 16-bit word values.

Conclusion

The function block `F_BOOL_TO_WORD` provides a simple and efficient way to convert Boolean values to word values. Its clear interface structure and direct functionality make it ideal for applications that require such conversions.

BYTE

BYTE conversions

Here you will find functions for converting the BYTE data type.

🛠️ Related exercises

- SIN_TO_PCAN_Callback_Byte
- Uebung_128b

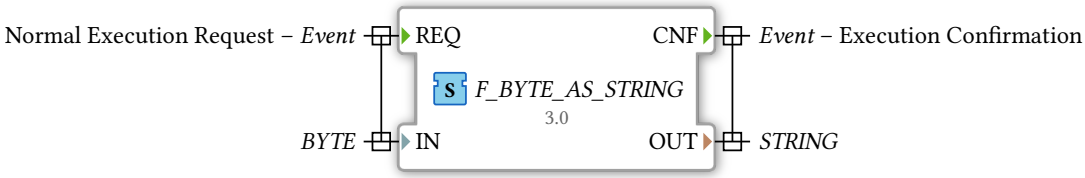
- Uebung_12x_sub

F_BYTE_AS_STRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_AS_STRING` is used to convert a BYTE value into a STRING. This functionality is particularly useful when numeric data needs to be converted into a text format suitable for display or logging.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. Triggered after successful conversion.

Data Inputs

- **IN**: BYTE value to be converted into a STRING.

Data Outputs

- **OUT**: Result of the conversion as a STRING.

Adapters

This function block does not use any adapters.

Functionality

When the REQ event is triggered, the incoming BYTE value (IN) is converted to a STRING and output via the OUT output. Simultaneously, the CNF event is triggered to signal successful execution.

Technical Features

- The conversion is performed using the function `BYTE_AS_STRING`.
- The function block is simple and has no internal states or complex logic.

State Overview

Since it is a simple function block, there are no state transitions or machines.

Application Scenarios

- Conversion of sensor values for display on an HMI.
- Preparation of data for logging or communication with other systems that expect string data.

⚖️ Comparison with Similar Building Blocks

- Compared to other conversion blocks like `F_WORD_AS_STRING` or `F_DWORD_AS_STRING`, this one is specifically optimized for BYTE values.
- Simpler and more specialized than generic conversion blocks.

Conclusion

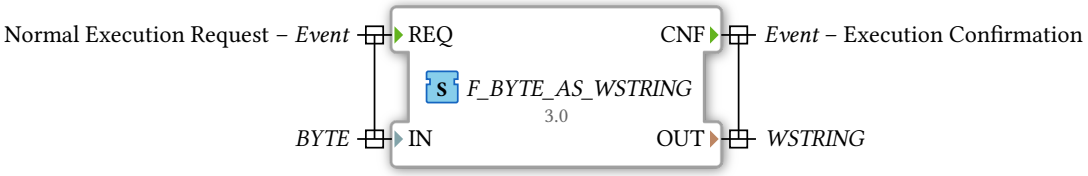
The `F_BYTE_AS_STRING` function block offers a simple and efficient way to convert BYTE values to strings. Its simplicity makes it reliable and easy to integrate into various applications.

F_BYTE_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_AS_WSTRING` converts a BYTE value into a WSTRING value. This is particularly useful in scenarios where byte data needs to be further processed or output as Unicode strings.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. It is linked to the input value `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. Returns the converted value via `OUT`.

Data Inputs

- **IN** (BYTE): The byte value to be converted to a WSTRING.

Data Outputs

- **OUT** (WSTRING): The resulting Unicode string after conversion.

Adapters

- No adapters available.

Functionality

This function block uses the built-in function `BYTE_AS_WSTRING` to convert the input byte value to a WSTRING. The conversion is triggered by the event `REQ`, and the result is returned via `CNF`.

Technical Features

- The conversion is performed directly without additional parameters.
- The block is optimized for use in IEC 61131-3 compliant environments.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Completed**: Sends the `CNF` event with the result.

Application Scenarios

- Converting byte data for display in user interfaces.
- Processing byte streams in Unicode-compatible systems.

⚖️ Comparison with Similar Blocks

- Unlike `F_BYTE_AS_STRING`, this block generates a Unicode string (WSTRING) instead of an ASCII string (STRING).
- Similar to `F_WORD_AS_WSTRING`, but optimized for 8-bit data.

Conclusion

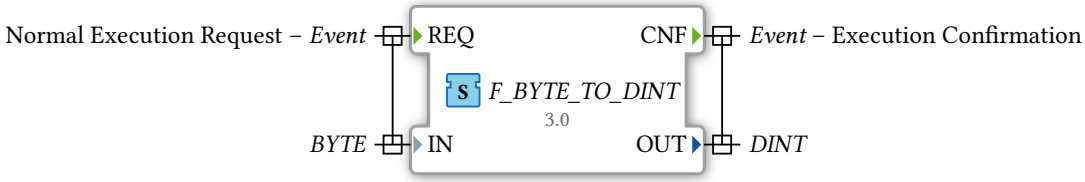
The `F_BYTE_AS_WSTRING` block is a simple and efficient way to convert byte data into Unicode strings. Its clear interface and direct functionality make it ideal for numerous applications in automation technology.

F_BYTE_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_TO_DINT` converts a `BYTE` value to a `DINT` value. It is part of the `iec61131::conversion` library and enables simple and efficient type conversion between these two data types.



Interface Structure

Event Inputs

- **REQ**: This input triggers the execution of the function block. It is linked to the data input `IN`.

Event Outputs

- **CNF**: This output signals the successful execution of the function block. It is linked to the data output `OUT`.

Data Inputs

- **IN**: The input expects a `BYTE` value, which is to be converted to a `DINT` value.

Data Outputs

- **OUT**: The output provides the converted `DINT` value.

Adapters

This function block does not have any adapters.

Functionality

The function block performs the conversion from `BYTE` to `DINT` as soon as the event `REQ` is received. The conversion is carried out using the function `BYTE_TO_DINT(IN)`, which converts the value of the input `IN` to the corresponding `DINT` value and outputs it as `OUT`. After successful conversion, the event `CNF` is triggered.

Technical Features

- The function block has a simple structure and contains only one algorithm for the conversion.
- The conversion is performed directly without any additional processing steps.

State Overview

The function block has no internal states. The execution occurs once each time the `REQ` event is received.

Application Scenarios

- Type conversion in control applications where `BYTE` data needs to be converted to `DINT` data.
- Integration into larger systems that require type compatibility between different data formats.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks such as `F_WORD_TO_DINT` or `F_INT_TO_DINT`, this function block is specifically designed for converting `BYTE` to `DINT`.
- It offers a simple and direct solution for this specific conversion.

Conclusion

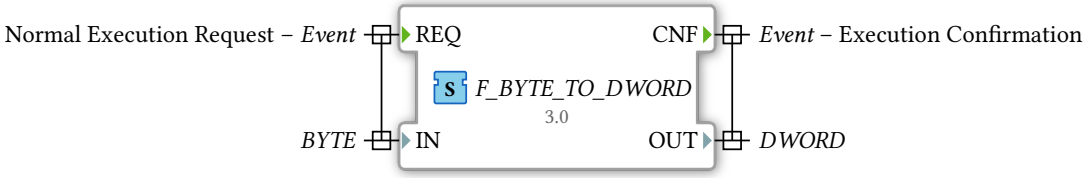
The `F_BYTE_TO_DINT` function block is a useful tool for type conversion in IEC 61131-3-based control systems. Its simple and efficient operation makes it a reliable solution for applications requiring conversion from `BYTE` to `DINT`.

F_BYTE_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_TO_DWORD` converts a `BYTE` value to a `DWORD` value. This block is particularly useful in scenarios where an extension of the data width is required, for example, when processing sensor data or communicating between systems with different data formats.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is coupled to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is coupled to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `BYTE` to be converted to a `DWORD` value.

Data Outputs

- **OUT**: The output value of type `DWORD`, containing the result of the conversion.

Adapters

This function block does not use any adapters.

Functionality

When the event input `REQ` is triggered, the algorithm is executed. This algorithm converts the `BYTE` value at input `IN` into a `DWORD` value and outputs the result at output `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

Technical Features

- The function block is simple and efficient because it directly utilizes the built-in function `BYTE_TO_DWORD`.
- No additional states or complex logic are required.

State Overview

Because it is a simple function block, there are no state transitions or internal states. The operation is performed atomically upon triggering `REQ`.

Application Scenarios

- Extending data widths in communication protocols.
- Processing sensor data that is in `BYTE` but needs to be further processed in `DWORD`.
- Compatibility adjustments between different system components.

⚖️ Comparison with similar function blocks

- **F_WORD_TO_DWORD**: Converts `WORD` to `DWORD`, similar in functionality, but with a different input data size.
- **F_BYTE_TO_WORD**: Converts `BYTE` to `WORD`, offering a smaller output data size compared to `F_BYTE_TO_DWORD`.

Conclusion

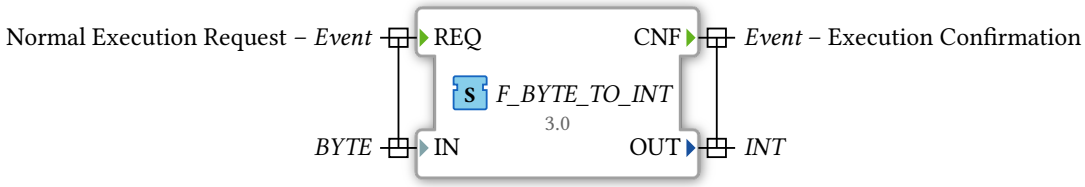
The `F_BYTE_TO_DWORD` function block is a simple and efficient tool for converting `BYTE` to `DWORD` values. Its clear interface and direct functionality make it ideal for applications requiring data width expansion.

F_BYTE_TO_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_TO_INT` converts a `BYTE` value to a `INT` value. It is part of the `iec61131::conversion` package and enables simple and efficient type conversion between these two data types.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN**: The input of type `BYTE`, containing the value to be converted.

Data Outputs

- **OUT**: The output of type `INT`, containing the result of the conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion from `BYTE` to `INT` as soon as the event `REQ` is received. The conversion is performed using the integrated function `BYTE_TO_INT`. After successful conversion, the result is output via output `OUT`, and the event `CNF` is triggered.

Technical Features

- The function block is a simple FB without internal state management.
- The conversion is performed directly and without delay.

State Overview

Since it is a simple function block, there are no state transitions or complex state logic.

Application Scenarios

- Type conversion in control applications where `BYTE` values need to be converted to `INT` values.
- Integration into larger control systems that need to process different data types.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_WORD_TO_INT` or `F_DWORD_TO_INT` offer conversions for other data types, but with the same basic principle.
- Unlike more complex conversion blocks, `F_BYTE_TO_INT` provides a simple and direct solution for the specific conversion of `BYTE` to `INT`.

Conclusion

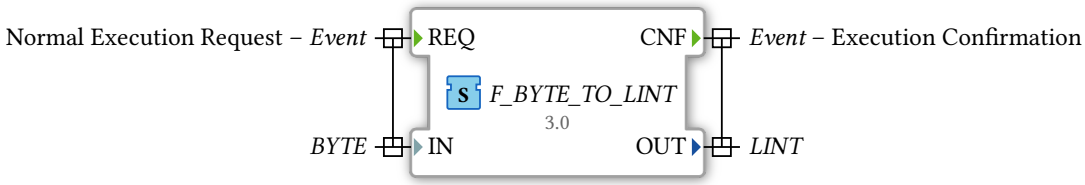
The `F_BYTE_TO_INT` function block is an efficient and easy-to-use tool for converting `BYTE` values to `INT` values. Its simplicity and direct operation make it ideal for applications that require fast and reliable type conversions.

F_BYTE_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_TO_LINT` converts a `BYTE` value to a `LINT` value. This block is particularly useful in scenarios where a type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (`BYTE`): The input value to be converted to the `LINT` type.

Data Outputs

- **OUT** (`LINT`): The converted output value.

Adapters

This function block has no adapters.

Functionality

When the `REQ` event is triggered, the value at input `IN` of type `BYTE` is converted to type `LINT` and output `OUT`. The `CNF` event signals the successful completion of the conversion.

Technical Features

- The function block uses the built-in function `BYTE_TO_LINT` for the conversion.
- It is a simple function block without state machines or complex algorithms.

State Overview

Since it is a simple function block, there are no state transitions or complex state logic.

Application Scenarios

- Type conversion in control applications where `BYTE` values need to be converted to `LINT` values.
- Use in data processing chains where different data types are processed.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_WORD_TO_LINT` or `F_DWORD_TO_LINT` offer similar functionality, but for different input types.
- Compared to generic conversion blocks, `F_BYTE_TO_LINT` is specialized and optimized for converting `BYTE` to `LINT`.

Conclusion

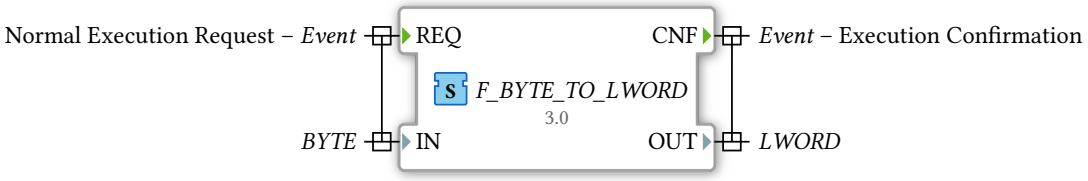
The `F_BYTE_TO_LINT` function block is an efficient and specialized tool for converting `BYTE` values to `LINT` values. Its simplicity and direct functionality make it a reliable choice for relevant use cases.

F_BYTE_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_TO_LWORD` converts a `BYTE` value to a `LWORD` value. It is part of the `iec61131::conversion` library and enables simple and efficient type conversion.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `BYTE` to be converted.

Data Outputs

- **OUT**: The output value of type `LWORD`, containing the result of the conversion.

Adapters

- No adapters are available.

Functionality

Upon receiving an incoming `REQ` event, the function block converts the `BYTE` value at input `IN` to a `LWORD` value. The result is output at `OUT` and confirmed by a `CNF` event.

Technical Features

- The function block uses the `BYTE_TO_LWORD` function for conversion.
- It is a simple function block without a state machine.

State Overview

Since it is a simple function block, there are no state transitions. The conversion occurs directly upon a `REQ` event.

Application Scenarios

- Type conversion in control applications where `BYTE` values need to be converted to `LWORD` values.
- Use in data processing chains where different data types are processed.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks are `F_WORD_TO_LWORD` or `F_DWORD_TO_LWORD`, which convert other data types.
- Unlike more complex conversion blocks, `F_BYTE_TO_LWORD` offers a simple and direct solution for the specific conversion of `BYTE` to `LWORD`.

Conclusion

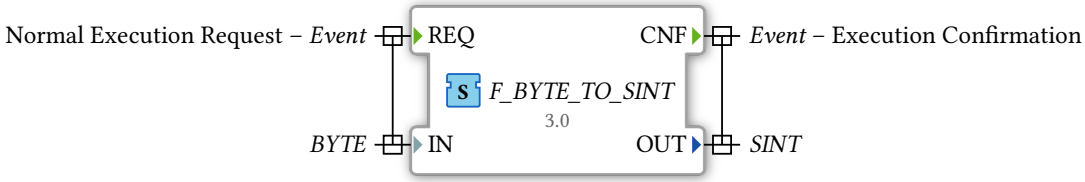
The `F_BYTE_TO_LWORD` function block is an efficient and simple solution for converting `BYTE` to `LWORD` values. Its clear interface structure and direct functionality make it ideal for use in control applications.

F_BYTE_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_TO_SINT` converts a data type `BYTE` to the data type `SINT`. This block is particularly useful in scenarios where a direct type conversion without additional calculations is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: Expects an input value of type `BYTE`, which is to be converted to `SINT`.

Data Outputs

- **OUT**: Outputs the converted value of type `SINT`.

Adapters

This function block has no adapters.

Functionality

Upon receiving an incoming `REQ` event, the function block converts the `BYTE` value at input `IN` to a `SINT` value and outputs the result via output `OUT`. Subsequently, the `CNF` event is triggered to signal the successful completion of the operation.

Technical Features

- The conversion is performed directly without any additional parameters or settings.
- The block is implemented as a simple function block (SimpleFB), which guarantees efficient execution.

State Overview

The function block has no internal states. Execution is purely event-driven and occurs with each `REQ` event.

Application Scenarios

- Type conversion in control applications, where `BYTE` data needs to be converted to `SINT`.
- Integration into larger function block networks where different data types are processed.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_BYTE_TO_SINT` is specialized and therefore offers greater efficiency for this specific conversion.
- Other blocks might offer additional features such as range checks or scaling, which is not the case here.

Conclusion

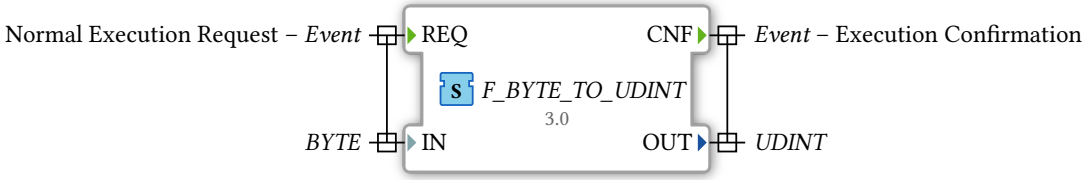
The `F_BYTE_TO_SINT` function block is an efficient tool for the direct conversion of `BYTE` to `SINT`. Its simplicity and specialization make it ideal for applications where fast and uncomplicated type conversion is required.

F_BYTE_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_TO_UDINT` is used to convert a BYTE value to a UDINT value. This function block is particularly useful in scenarios where a type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: This event input triggers the conversion. The input is linked to the data input `IN`.

Event Outputs

- **CNF**: This event output signals the successful completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `BYTE`, containing the value to be converted.

Data Outputs

- **OUT**: The output of type `UDINT`, containing the result of the conversion.

Adapters

This function block has no adapters.

Functionality

When the event input `REQ` is triggered, the algorithm is executed, converting the value at input `IN` (of type `BYTE`) into a `UDINT` value and outputting the result at output `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

Technical Features

- The function block uses the built-in function `BYTE_TO_UDINT` for the conversion.
- The conversion is performed without additional parameters or configuration.

State Overview

The function block has no internal states. Execution occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Type conversion in control applications where BYTE values need to be converted to UDINT values.
- Integration into larger control systems that require uniform data processing.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks such as `F_WORD_TO_UDINT` or `F_DWORD_TO_UDINT`, this block is specifically optimized for converting `BYTE` to `UDINT`.
- Similar blocks can process different input data types but are otherwise comparable in their functionality.

🛠️ Related Exercises

- Exercise_011a

Conclusion

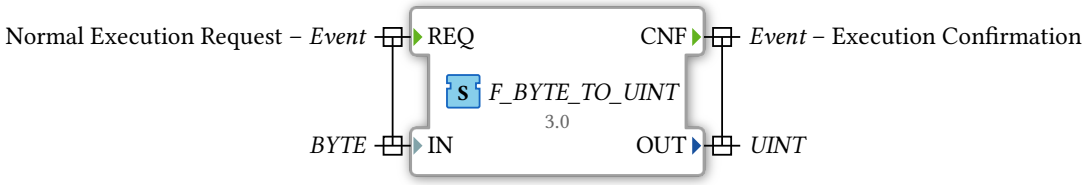
The `F_BYTE_TO_UDINT` function block offers a simple and efficient way to convert BYTE values to UDINT values. Its clear interface structure and direct functionality make it ideal for applications requiring fast and reliable type conversion.

F_BYTE_TO_UINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_TO_UINT` converts a `BYTE` value to a `UINT` value. This is particularly useful when data of different types needs to be processed in a system and type conversion is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Confirms the successful execution of the function block. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `BYTE`, which is to be converted to a `UINT` value.

Data Outputs

- **OUT**: The output of type `UINT`, which contains the result of the conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion of the `BYTE` value at the input `IN` to a `UINT` value as soon as the event `REQ` is triggered. The result is output at `OUT`, and the event `CNF` signals successful execution.

Technical Features

- The function block has a simple structure and contains only one algorithm that performs the conversion.
- The conversion is performed using the function `BYTE_TO_UINT`, which converts the `BYTE` value into a `UINT` value.

State Overview

The function block has no internal state. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control and automation systems.
- Data processing in systems that use different data types.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_WORD_TO_UINT` or `F_DWORD_TO_UINT` perform similar conversions, but with different input types.
- `F_BYTE_TO_UINT` specializes in converting `BYTE` to `UINT` and is therefore more efficient for this specific use case.

🛠️ Related Exercises

- Exercise_006c

Conclusion

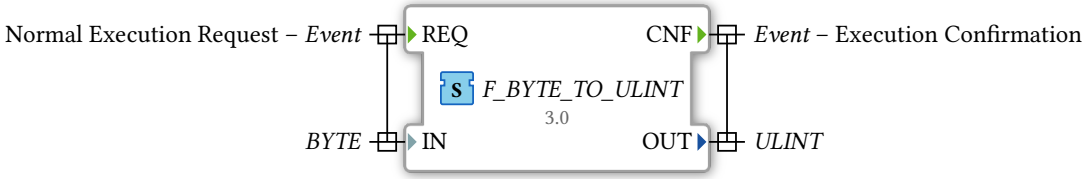
The function block `F_BYTE_TO_UINT` is a simple and efficient tool for converting `BYTE` values to `UINT` values. Its clear interface structure and direct functionality make it ideal for applications requiring such type conversions.

F_BYTE_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_BYTE_TO_ULINT converts a BYTE value to a ULINT value. This conversion is particularly useful in scenarios where data of varying sizes needs to be processed or transferred.



Interface Structure

Event Inputs

- **REQ**: This input triggers the execution of the function block. The input is linked to the data input IN.

Event Outputs

- **CNF**: This output signals the successful execution of the function block. The output is linked to the data output OUT.

Data Inputs

- **IN**: The input expects a BYTE value, which is to be converted to a ULINT value.

Data Outputs

- **OUT**: The output provides the converted ULINT value.

Adapters

No adapters are present.

Functionality

The function block performs the conversion from BYTE to ULINT as soon as the event REQ is received. The conversion is performed using the function BYTE_TO_ULINT, which converts the input value IN into the corresponding value ULINT. After successful conversion, the event CNF is triggered, and the converted value is available at the output OUT.

Technical Features

- The function block has a simple structure and contains only one conversion algorithm.
- The conversion is performed without additional parameters or complex logic.

State Overview

The function block has no internal states. Execution occurs immediately upon receiving the REQ event.

Application Scenarios

- Data conversion in control applications where BYTE values need to be converted to larger data types such as ULINT.
- Integration into larger systems that require uniform data processing in ULINT.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as F_WORD_TO_ULINT or F_DWORD_TO_ULINT perform conversions from other data types to ULINT. F_BYTE_TO_ULINT is specifically for converting BYTE.

Conclusion

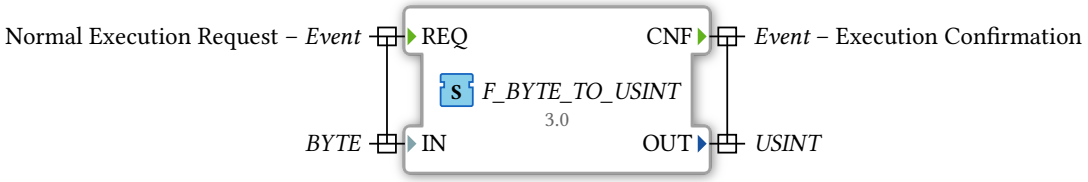
The F_BYTE_TO_ULINT function block offers a simple and efficient way to convert BYTE values into ULINT values. Its clear interface and direct functionality make it a useful component in automation technology.

F_BYTE_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_TO_USINT` converts a `BYTE` value to a `USINT` value (Unsigned Short Integer). This function block is particularly useful in scenarios where type conversion between these two data types is required, for example, in communication between different systems or in data processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `BYTE` to be converted.

Data Outputs

- **OUT**: The output value of type `USINT`, containing the result of the conversion.

Adapters

No adapters are available.

Functionality

The function block performs the conversion by transforming the `BYTE` value at input `IN` into a `USINT` value and providing the result at output `OUT`. The conversion occurs synchronously with the triggering of the `REQ` event and is confirmed by the `CNF` event.

Technical Features

- The function block uses the built-in function `BYTE_TO_USINT` for the conversion.
- It is a simple function block without a state machine or complex logic.

State Overview

Because it is a simple function block, there are no state transitions or state management. The operation is atomic and is executed on every `REQ` event.

Application Scenarios

- Type conversion in communication protocols where different systems use different data types.
- Data processing in control systems where conversion between `BYTE` and `USINT` is required.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_WORD_TO_UINT` or `F_DWORD_TO_UDINT` perform similar conversions, but with different data types.
- `F_BYTE_TO_USINT` is specifically optimized for converting `BYTE` to `USINT` and is therefore more efficient for this use case.

Conclusion

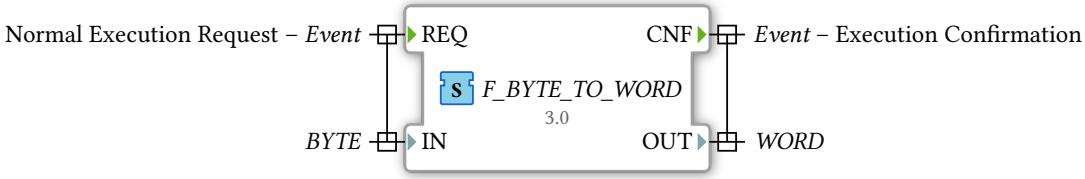
The `F_BYTE_TO_USINT` function block is a simple yet effective building block for converting between `BYTE` and `USINT`. Its simplicity and efficiency make it ideal for applications requiring fast and reliable type conversion.

F_BYTE_TO_WORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_BYTE_TO_WORD` converts a `BYTE` data type to a `WORD` data type. This conversion is useful when data of different sizes needs to be processed or transferred.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `BYTE` to be converted.

Data Outputs

- **OUT**: The output of type `WORD`, containing the result of the conversion.

Adapters

No adapters are present.

Functionality

The function block performs the conversion from `BYTE` to `WORD` as soon as the event `REQ` is received. The conversion is performed using the function `BYTE_TO_WORD(IN)`, which converts the input value `IN` into the corresponding value `WORD`. The result is output at `OUT`, and the event `CNF` signals the completion of the operation.

Technical Features

- The function block has a simple structure and contains only one conversion algorithm.
- The conversion is performed directly and without any additional delay.

State Overview

The function block has no internal states. The conversion is performed on each `REQ` event, and the result is output immediately.

Application Scenarios

- **Data Conversion**: When a `BYTE` value needs to be processed in a system that expects `WORD` values.
- **Interface Adaptation**: For communication between systems that use different data types.

⚖️ Comparison with Similar Building Blocks

- **F_WORD_TO_BYTE**: Performs the reverse conversion, from `WORD` to `BYTE`.
- **F_BYTE_TO_INT**: Converts `BYTE` to `INT`, which is another type of data type conversion.

Conclusion

The `F_BYTE_TO_WORD` function block is a simple and efficient tool for converting `BYTE` values to `WORD` values. Its clear interface structure and direct functionality make it ideal for applications requiring fast and straightforward data type conversions.

DINT

DINT Conversions

🎧 Podcast

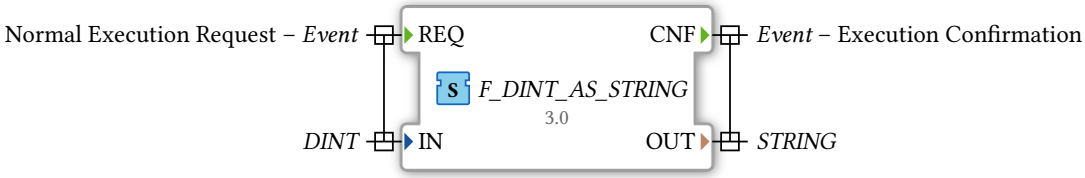
- SINT, INT, DINT: Why the choice of data type determines efficiency and errors ... SINT

F_DINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_AS_STRING` converts a 32-bit integer value (DINT) into a string (STRING). This functionality is particularly useful when numeric values are needed for display or logging in a human-readable format.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is activated along with the converted value at data output `OUT`.

Data Inputs

- **IN**: Expects a 32-bit integer value (DINT) to be converted into a string.

Data Outputs

- **OUT**: Returns the converted value as a string (STRING).

Adapters

This function block has no adapter interfaces.

Operation

When the `REQ` event is triggered, the algorithm is started that converts the value of `IN` (DINT type) into a STRING. The result is output to `OUT`, and the `CNF` event signals successful conversion.

Technical Features

- The conversion is performed directly without additional parameters.
- The function block is part of the `iec61131::conversion` package.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Displaying numeric values in user interfaces.
- Logging measured values to text files or databases.
- Preparing data for communication with systems that can only process strings.

⚖️ Comparison with Similar Blocks

Compared to other conversion blocks, `F_DINT_AS_STRING` offers a specialized and efficient solution for converting DINT to STRING. Other blocks might offer additional features such as formatting or locale settings, but are often more complex to use.

Conclusion

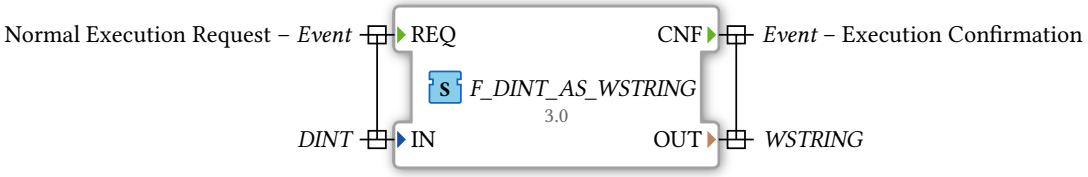
The `F_DINT_AS_STRING` function block is a simple and effective solution for converting integers to strings. Its clear interface and direct functionality make it ideal for applications requiring fast and straightforward conversion.

F_DINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_AS_WSTRING` converts a 32-bit integer value (DINT) into a Unicode string (WSTRING). This functionality is particularly useful when numeric values are needed as text for display or further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion of the input value. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (DINT): The 32-bit integer value to be converted.

Data Outputs

- **OUT** (WSTRING): The resulting Unicode string after conversion.

Adapters

No adapters available.

Functionality

When the `REQ` event is triggered, the value at input `IN` is read and converted into a Unicode string using the function `DINT_AS_WSTRING`. The result is output at `OUT`, and the `CNF` event signals successful execution.

Technical Features

- The function block is a simple algorithm without state machines.
- Conversion is performed directly without additional parameters.

State Overview

Since it is a simple function block, there are no state transitions or management.

Application Scenarios

- Displaying numeric values in user interfaces.
- Logging numeric data in text format.
- Preparing data for text-based communication protocols.

⚖️ Comparison with Similar Blocks

- Compared to `F_INT_AS_WSTRING`, this block processes 32-bit integers instead of 16-bit integers.
- Similar blocks like `F_DINT_AS_STRING` convert to ASCII instead of Unicode strings.

Conclusion

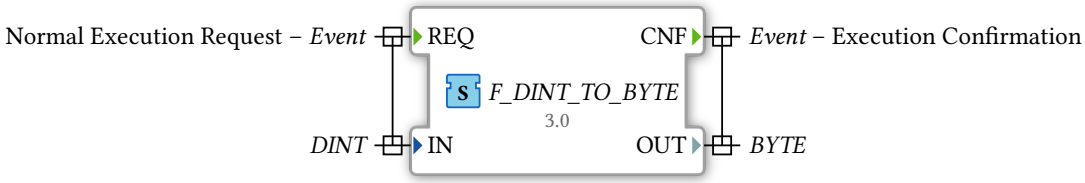
The `F_DINT_AS_WSTRING` function block offers a simple and efficient way to convert 32-bit integers to Unicode strings. Its clear interface and direct functionality make it ideal for applications that require numerical data in text format.

F_DINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_TO_BYTE` converts a 32-bit integer value (DINT) to an 8-bit integer value (BYTE). This conversion is particularly useful when data needs to be exchanged or processed between systems with different word lengths.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. The converted value is available at data output `OUT`.

Data Inputs

- **IN**: The input value of type `DINT` (32-bit integer) to be converted.

Data Outputs

- **OUT**: The output value of type `BYTE` (8-bit integer) containing the result of the conversion.

Adapters

This function block does not have any adapter interfaces.

Functionality

When the event input `REQ` is triggered, the algorithm is executed, converting the value at input `IN` (of type `DINT`) into a value of `BYTE`. The result is output at `OUT`, and the event `CNF` signals the completion of the operation.

The algorithm used is:
OUT := DINT_TO_BYTE(IN);

Technical Features

- Converting `DINT` to `BYTE` can result in data loss if the original value is outside the representable range of `BYTE` (0 to 255).
- The function block is simple and deterministic, without internal states or complex logic.

State Overview

The function block has no internal state. Execution occurs exclusively upon triggering the `REQ` event.

Application Scenarios

- Reducing data width for communication with peripheral devices that can only process BYTE data.
- Preprocessing data for systems with limited storage capacity.
- Integration into larger control systems where different data types need to be converted.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_DINT_TO_BYTE` is specialized and optimized for the specific conversion of `DINT` to `BYTE`.
- Similar blocks like `F_INT_TO_BYTE` or `F_WORD_TO_BYTE` offer conversions between other data types, but with similar functionality.

Conclusion

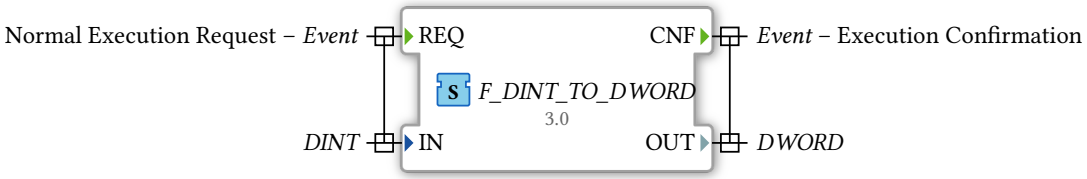
The `F_DINT_TO_BYTE` function block is a simple and efficient tool for converting 32-bit integers to 8-bit values. Its simplicity and determinism make it reliable, but users should consider potential data loss with large input values. Ideal for applications where space-saving data representation is required.

F_DINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_TO_DWORD` converts a 32-bit integer value (DINT) into a 32-bit unsigned value (DWORD). This conversion is particularly necessary when signed values need to be converted to unsigned values, for example, for further processing in certain control algorithms or for communication with other systems.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion of the input value `IN` from DINT to DWORD.

Event Outputs

- **CNF**: Signals successful completion of the conversion and provides the converted value `OUT`.

Data Inputs

- **IN** (DINT): The signed 32-bit input value to be converted.

Data Outputs

- **OUT** (DWORD): The unsigned 32-bit output value after the conversion.

Adapters

- No adapters are available.

Functionality

This function block performs the DINT to DWORD conversion when the event `REQ` is triggered. The value of input `IN` is converted using the function `DINT_TO_DWORD` and made available at output `OUT`. After the conversion is complete, the event `CNF` is triggered.

Technical Features

- The conversion is performed directly and without any additional delay.
- The function block is optimized for use in real-time control systems.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Conversion State**: Performs the conversion and sets the output `OUT`.
3. **Acknowledgement State**: Triggers the `CNF` event and returns to the idle state.

Application Scenarios

- Conversion of signed sensor data to unsigned values for further processing.
- Preparation of data for communication with systems that expect unsigned values.

⚖️ Comparison with Similar Blocks

- Unlike generic conversion blocks, `F_DINT_TO_DWORD` specializes in DINT to DWORD conversion and therefore offers optimized and error-free conversion for this specific use case.

Conclusion

The `F_DINT_TO_DWORD` function block is an efficient and reliable tool for converting signed to unsigned 32-bit values. Its ease of use and direct integration into the 4diac IDE make it an ideal solution for relevant applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

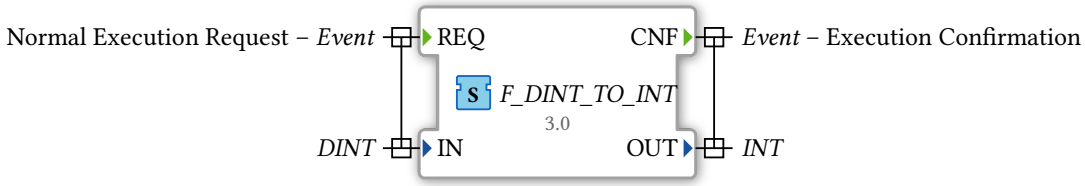
]

F_DINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_TO_INT` converts a 32-bit DINT value to a 16-bit INT value. This conversion is particularly necessary in scenarios where data needs to be exchanged between systems with different word lengths.



Interface Structure

Event Inputs

- `REQ` (Normal Execution Request): Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- `CNF` (Execution Confirmation): Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- `IN` (DINT): The 32-bit DINT value to be converted.

Data Outputs

- `OUT` (INT): The resulting 16-bit INT value after conversion.

Adapters

No adapters available.

Functionality

This function block performs the conversion of a 32-bit DINT value (`IN`) to a 16-bit INT value (`OUT`) as soon as the event `REQ` is triggered. The conversion is carried out by the function `DINT_TO_INT`. After the conversion is complete, the event `CNF` is triggered.

Technical Details

- Data loss can occur if the original DINT value is outside the representable range of an INT value (-32768 to 32767).
- This function block is part of the package `iec61131::conversion`.

State Overview

This function block has no internal state. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Data conversion in control systems with different word widths.
- Integration of systems that use different data types.
- Reduction of memory usage when the higher precision of DINT is not required.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_DINT_TO_INT` specializes in converting DINT to INT and therefore offers a simple and direct solution for this specific use case.
- Other conversion blocks, such as `F_INT_TO_DINT`, perform the opposite conversion.

Conclusion

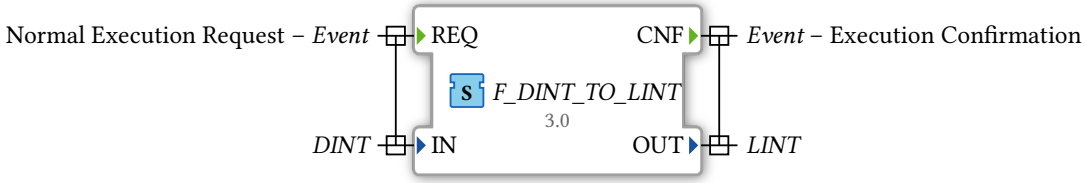
The function block `F_DINT_TO_INT` is an efficient tool for converting 32-bit DINT values to 16-bit INT values. Its simple and specialized functionality makes it ideal for applications where such conversions are required regularly. However, when using it, the potential for data loss with large values must be taken into account.

F_DINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_TO_LINT` converts a 32-bit integer value (DINT) to a 64-bit integer value (LINT). This block is particularly useful in scenarios where extending the value range or changing the type is required for subsequent processing steps.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals successful completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN** (DINT): The 32-bit integer value to be converted.

Data Outputs

- **OUT** (LINT): The resulting 64-bit integer value after conversion.

Adapters

This function block does not use any adapters.

Operation

When the `REQ` event is triggered, the value of the `IN` input is copied to the `OUT` output. An automatic type conversion from DINT to LINT occurs, preserving the numeric value.

Technical Features

- The conversion is performed without data loss, as the LINT type supports a larger range of values than DINT.
- The algorithm is implemented in Structured Text (ST) and consists of a simple assignment.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Extending the value range for subsequent calculations
- Type adaptation in heterogeneous systems
- Data preparation for interfaces with varying requirements

⚖️ Comparison with Similar Function Blocks

Compared to generic conversion blocks, `F_DINT_TO_LINT` offers a specialized and type-safe solution for the specific conversion between DINT and LINT. Other blocks might offer more general conversion options, but may be less efficient or require additional configuration.

Conclusion

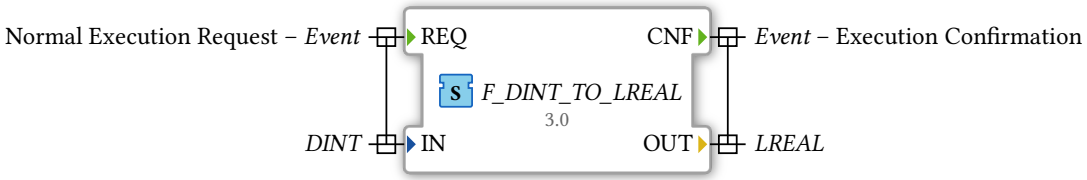
The `F_DINT_TO_LINT` function block provides a simple and efficient solution for converting 32-bit to 64-bit integers. Its clear interface and direct functionality make it a reliable component in control systems where such type conversions are required.

F_DINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_TO_LREAL` converts a 32-bit integer value (DINT) into a 64-bit floating-point value (LREAL). This conversion is particularly necessary in control systems where different data types need to be processed in various parts of the application.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at data output `OUT`.

Data Inputs

- **IN**: The input value of type `DINT`, which is to be converted to a `LREAL` value.

Data Outputs

- **OUT**: The output value of type `LREAL`, which represents the result of the conversion.

Adapters

This function block does not use any adapters.

Functionality

The function block performs a direct type conversion. When the event `REQ` is triggered, the value of `IN` is read and converted to the data type `LREAL`. The result is output to `OUT`, and the event `CNF` signals successful conversion.

Technical Features

- The conversion is performed without additional scaling or offset adjustment.
- The function block is deterministic and has no internal states.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of integers to floating-point numbers for mathematical calculations.
- Integration into control systems where different data types need to be processed.
- Use in control engineering where precise floating-point calculations are required.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_DINT_TO_LREAL` specializes in converting `DINT` to `LREAL`, thus offering greater efficiency and code readability.
- Other building blocks might offer additional features such as scaling or limiting, which is not the case here.

Conclusion

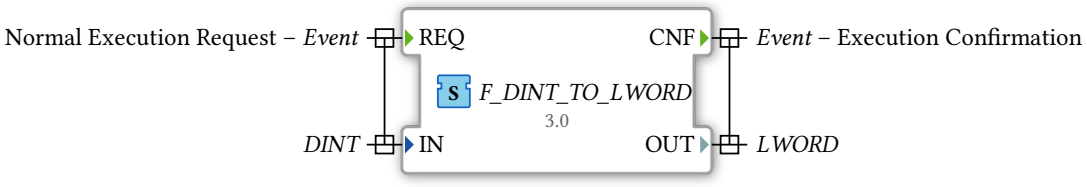
The `F_DINT_TO_LREAL` function block is a simple and efficient building block for converting integers to floating-point numbers. Its clear interface and deterministic operation make it ideal for use in control systems.

F_DINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_TO_LWORD` converts a 32-bit integer value (DINT) to a 64-bit unsigned integer value (LWORD). This block is particularly useful in scenarios where extending the value range or changing the type is required for subsequent processing steps.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the function block reads the value at the data input `IN` and performs the conversion.

Event Outputs

- **CNF**: Signals the completion of the conversion. Simultaneously, the converted value is output at data output `OUT`.

Data Inputs

- **IN** (DINT): The 32-bit integer value to be converted to a 64-bit unsigned integer value.

Data Outputs

- **OUT** (LWORD): The resulting 64-bit unsigned integer value after the conversion.

Adapters

This function block does not have any adapter interfaces.

Operation

When event input `REQ` is triggered, the function block's algorithm is activated. This function reads the value at input `IN`, converts it from DINT to LWORD, and stores the result at output `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

Technical Features

- The conversion is performed by directly assigning the DINT value to an LWORD, with the value being extended accordingly.
- The function block is deterministic and performs no additional calculations or checks.

State Overview

The function block has no internal states. The operation is re-executed with each `REQ` event.

Application Scenarios

- Extending the value range for subsequent operations that require 64-bit values.
- Type conversion in heterogeneous systems where different data types need to be processed.
- Use in control systems where higher precision or a larger value range is required.

⚖️ Comparison with similar function blocks

- Unlike generic conversion function blocks, `F_DINT_TO_LWORD` specializes in converting DINT to LWORD.
- Similar function blocks such as `F_INT_TO_DINT` or `F_LINT_TO_LWORD` offer conversions between other integer types.

Conclusion

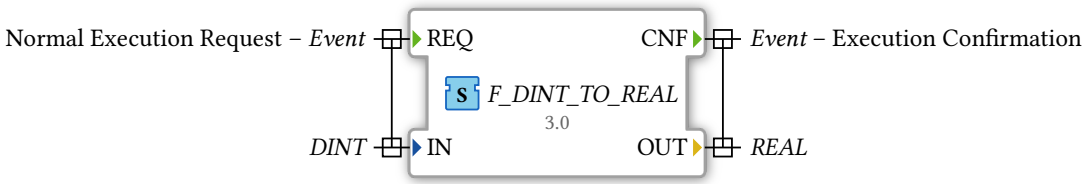
The `F_DINT_TO_LWORD` function block is a simple yet effective function block for converting 32-bit to 64-bit integer values. Its deterministic operation and clear interface make it a reliable component in control and automation solutions.

F_DINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_TO_REAL` converts an integer value of type `DINT` into a floating-point value of type `REAL`. This conversion is necessary in many control and automation applications when data needs to be exchanged between different systems or components.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `DINT`, which is to be converted to a `REAL` value.

Data Outputs

- **OUT**: The converted output value of type `REAL`.

Adapters

No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The algorithm `REQ` converts the input value `IN` of type `DINT` into a `REAL` value and stores the result in `OUT`. The event `CNF` is then triggered to signal the completion of the conversion.

Technical Features

- The function block is implemented simply and efficiently, without additional states or complex logic.
- The conversion is performed directly by the function `DINT_TO_REAL`.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of integer sensor values to floating-point values for further calculations.
- Data conversion between systems that use different data types.
- Integration into control algorithms that require floating-point values.

⚖️ Comparison with Similar Function Blocks

- **F_INT_TO_REAL**: Converts `INT` to `REAL`, but has a smaller input range compared to `DINT`.
- **F_LINT_TO_REAL**: Converts `LINT` to `REAL`, has a larger input range, but is unnecessary in most cases if `DINT` is sufficient.

Conclusion

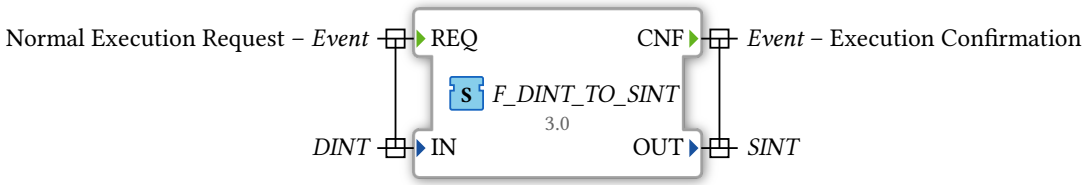
The function block `F_DINT_TO_REAL` is a simple and efficient tool for converting `DINT` values to `REAL` values. It is particularly useful in applications where data with different data types needs to be exchanged between different systems or components. Its direct and stateless implementation makes it reliable and easy to integrate.

F_DINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_TO_SINT` converts a 32-bit integer value (DINT) to an 8-bit integer value (SINT). This block is particularly useful in scenarios where type conversion between integer data types of different sizes is required.



Interface Structure

Event Inputs
- **REQ**: Triggers the conversion. It is associated with the input value `IN`.

Event Outputs
- **CNF**: Signals the completion of the conversion. Returns the converted value via `OUT`.

Data Inputs
- **IN** (DINT): The 32-bit integer value to be converted.

Data Outputs
- **OUT** (SINT): The resulting 8-bit integer value after conversion.

Adapters
- No adapters available.

Functionality
This function block performs the conversion from `DINT` to `SINT` as soon as the event `REQ` is triggered. The algorithm uses the built-in function `DINT_TO_SINT` to perform the conversion. After successful conversion, the event `CNF` is output with the result `OUT`.

Technical Features
- **Type Conversion**: Conversion can lead to data loss if the original `DINT` value is outside the representable range of `SINT` (-128 to 127).
- **Simple Implementation**: The function block is implemented as a simple function block (SimpleFB) without internal states.

State Overview
The function block has no internal states. Execution occurs immediately upon receiving the `REQ` event.

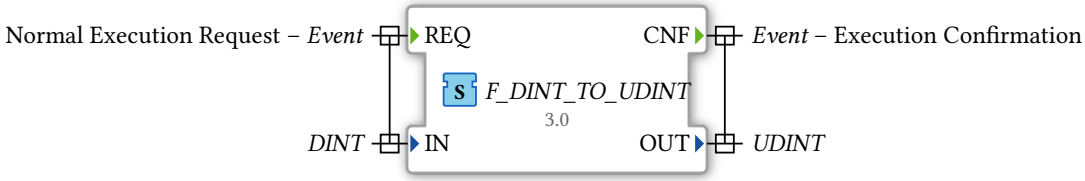
Application Scenarios
- **Data Reduction**: When only the lower 8 bits of a 32-bit value are relevant.
- **Compatibility**: Adapting values for systems or components that can only process `SINT`.
- **Logging**: Memory-optimized storage of values where the higher bits are not needed.

⚖️ Comparison with similar function blocks
- **F_INT_TO_SINT**: Similar function block, but for 16-bit to 8-bit conversion.
- **F_ANY_TO_**: Generic conversion function blocks that are more flexible but potentially less efficient.

Conclusion
The `F_DINT_TO_SINT` function block provides a simple and efficient way to convert 32-bit integers to 8-bit integers. Its use is particularly useful in resource-constrained environments or when adapting data types between different system components. However, be aware of the potential for data loss with large input values.

F_DINT_TO_UDINT
• − ◦ • − ◦ • − ◦ •

Introduction
The function block `F_DINT_TO_UDINT` converts a 32-bit signed integer value (DINT) to a 32-bit unsigned integer value (UDINT). This conversion is useful when exchanging data between systems that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at data output `OUT`.

Data Inputs

- **IN**: The DINT value to be converted (32-bit signed integer).

Data Outputs

- **OUT**: The converted UDINT value (32-bit unsigned integer).

Adapters

No adapters are supported.

Functionality

When the event input `REQ` is triggered, the function block's algorithm is executed. The value at input `IN`, of type `DINT`, is converted to a `UDINT` value and output at `OUT`. The event `CNF` is then triggered to signal the successful completion of the conversion.

Technical Features

- The conversion is performed directly by the function `DINT_TO_UDINT`.
- There is no additional error handling for overflows or invalid values.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering `REQ`.

Application Scenarios

- Conversion of sensor values that are in the form of signed integers for systems that expect unsigned values.
- Data preparation for communication protocols that use unsigned integers.

⚖️ Comparison with Similar Function Blocks

- **F_INT_TO_UINT**: Converts 16-bit signed integers to unsigned integers.
- **F_LINT_TO_ULINT**: Converts 64-bit signed integers to unsigned integers.
- **F_SINT_TO_USINT**: Converts 8-bit signed integers to unsigned integers.

🛠️ Related Exercises

- Exercise_009a

Conclusion

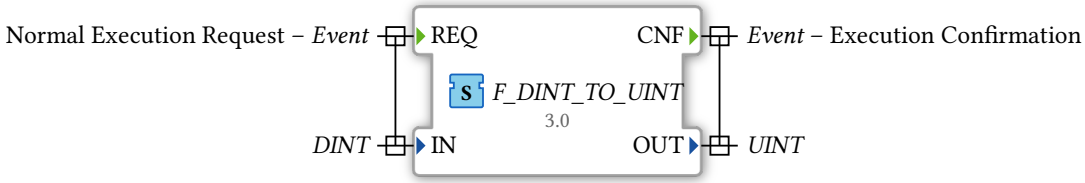
The function block `F_DINT_TO_UDINT` provides a simple and efficient way to convert signed 32-bit integers to unsigned 32-bit integers. It is particularly useful in scenarios where data needs to be exchanged between systems with different data type requirements.

F_DINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_TO_UINT` converts a 32-bit integer value (DINT) to an unsigned 32-bit integer value (UINT). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals successful completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `DINT` to be converted.

Data Outputs

- **OUT**: The output value of type `UINT`, representing the result of the conversion.

Adapters

No adapters are defined for this function block.

Functionality

This function block performs the conversion from `DINT` to `UINT` when the event `REQ` is received. The conversion is performed using the built-in function `DINT_TO_UINT`. After successful conversion, the event `CNF` is triggered, and the converted value is output via the output `OUT`.

Technical Features

- The conversion is performed directly without any additional delay.
- The block does not support error handling for invalid input values.

State Overview

1. **Idle**: Waits for the event `REQ`.
2. **Conversion**: Performs the conversion and outputs the result via `OUT`, followed by an event `CNF`.

Application Scenarios

- Type conversion in control applications where signed values need to be converted to unsigned values.
- Data processing in industrial automation systems.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_DINT_TO_UINT` specializes in converting between `DINT` and `UINT`, thus offering a simpler and more direct solution for this specific use case.

Conclusion

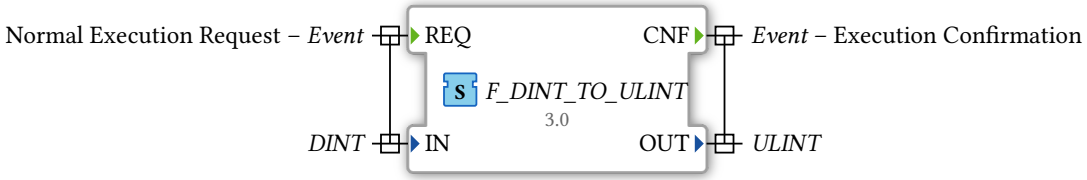
The `F_DINT_TO_UINT` function block is an efficient tool for the specific conversion between `DINT` and `UINT` data types. Its simplicity and direct functionality make it ideal for applications requiring fast and reliable type conversion.

F_DINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DINT_TO_ULINT` converts a 32-bit signed integer value (`DINT`) into a 64-bit unsigned integer value (`ULINT`). This block is particularly useful when data needs to be exchanged between systems with different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event, along with the converted value, is output at data output `OUT`.

Data Inputs

- **IN** (`DINT`): The signed 32-bit integer value to be converted.

Data Outputs

- **OUT** (`ULINT`): The unsigned 64-bit integer value containing the result of the conversion.

Adapters

- No adapters are available.

Functionality

The function block performs the conversion of `DINT` to `ULINT` as soon as the event `REQ` is received. The algorithm `REQ` is executed, converting the value of `IN` using the function `DINT_TO_ULINT` and outputting it to `OUT`. Finally, the event `CNF` is triggered to signal the successful completion of the operation.

Technical Features

- The function block supports direct conversion without additional parameters.
- The conversion is performed without data loss because the `ULINT` type covers a larger value range than `DINT`.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Conversion State**: Performs the conversion and sets the output `OUT`.
3. **Acknowledgement State**: Triggers the `CNF` event and returns to the idle state.

Application Scenarios

- Data conversion in control systems that use different data types.
- Integration of components that require unsigned 64-bit values.
- Increased compatibility between different systems.

⚖️ Comparison with similar function blocks

- **F_INT_TO_ULINT**: Converts a smaller signed integer (`INT`) to `ULINT`.
- **F_LINT_TO_ULINT**: Converts a 64-bit signed integer (`LINT`) to `ULINT`.
- **F_DINT_TO_LINT**: Converts `DINT` to a signed 64-bit integer (`LINT`).

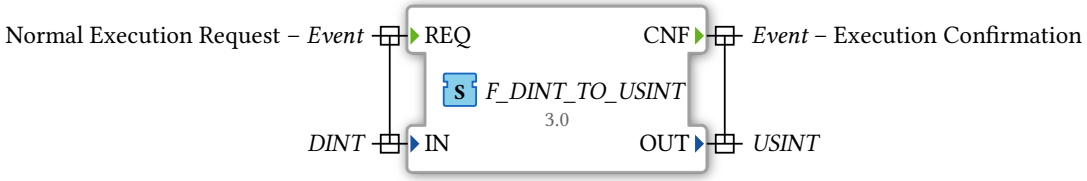
Conclusion

The F_DINT_TO_ULINT function block provides a simple and efficient way to convert signed 32-bit integer values to unsigned 64-bit integer values. Its clear interface and reliable operation make it ideal for applications requiring safe and lossless data type conversion.

F_DINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

The function block F_DINT_TO_USINT converts a 32-bit integer value (DINT) to an 8-bit unsigned integer value (USINT). This conversion is useful when data needs to be exchanged between systems or components that use different data types.



- **REQ**: Starts the conversion. This event input is linked to the data input IN.
- **CNF**: Signals the completion of the conversion. This event output is linked to the data output OUT.
- **IN**: The input value of type DINT (32-bit integer) to be converted.
- **OUT**: The output value of type USINT (8-bit unsigned integer) containing the result of the conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use any adapters.

When the event input REQ is triggered, the algorithm is executed. This algorithm converts the value at input IN (of type DINT) into a value of type USINT and outputs the result at output OUT. The event output CNF is then activated to signal the successful completion of the conversion.

- The conversion is performed by the built-in function DINT_TO_USINT.
- Please note that data loss may occur during the conversion from DINT to USINT, as the value range of USINT (0 to 255) is smaller than that of DINT (-2,147,483,648 to 2,147,483,647).

1. **Idle State**: The function block is waiting for the event REQ.
2. **Active State**: Upon receiving REQ, the conversion is performed and CNF is triggered.

- Data conversion in control systems when different data types are used.
- Reduction of memory usage when only small unsigned values are required.
- Interface adaptation between different system components.
- **F_INT_TO_USINT**: Converts a 16-bit integer value (INT) to an 8-bit unsigned integer value (USINT). In comparison, F_DINT_TO_USINT processes larger input values.
- **F_LINT_TO_USINT**: Converts a 64-bit integer value (LINT) to an 8-bit unsigned integer value (USINT). F_DINT_TO_USINT is designed for smaller input ranges.

The function block F_DINT_TO_USINT offers a simple and efficient way to convert 32-bit integers to 8-bit unsigned integers. It is particularly useful in scenarios where memory space needs to be optimized or data needs to be exchanged between systems with different data types. However, due to the potential for data loss, it should be ensured that the input values are within the valid range of USINT.

Functionality

Application Scenarios

State Overview

Technical Features

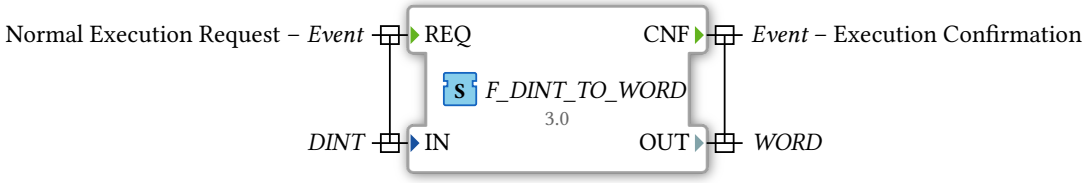
⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

F_DINT_TO_WORD

• – ◦ • – ◦ • – ◦ •

The function block `F_DINT_TO_WORD` converts a 32-bit integer value (DINT) into a 16-bit word value (WORD). This conversion is particularly useful when data needs to be exchanged between systems with different word widths.



- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is read and converted.
- **CNF**: Signals the successful completion of the conversion. The converted value is available at data output `OUT`.
- **IN**: The 32-bit integer value (DINT) to be converted.
- **OUT**: The converted 16-bit word value (WORD).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block has no adapters.

When the event input `REQ` is triggered, the algorithm is executed, converting the value of `IN` from DINT to WORD and outputting the result to `OUT`. The event `CNF` is then triggered to signal the completion of the operation.

- The conversion truncates the upper 16 bits of the DINT value, as a WORD can only store 16 bits. It is therefore important to ensure that the original DINT value is within the representable range of a WORD (0 to 65535) to avoid data loss.
- The function block is deterministic and has no internal states.

Since this is a simple function block without internal states, there are no state transitions. The block performs the conversion on each `REQ` event and signals completion with `CNF`.

- Communication between systems that use different word widths.
- Adapting data for peripheral devices that can only process WORD data.
- Use in control systems where memory optimization is required.
- Compared to generic conversion blocks like `F_ANY_TO_ANY`, `F_DINT_TO_WORD` is specialized and therefore more efficient when precisely this type of conversion is required.

- Similar blocks like `F_INT_TO_WORD` work with smaller integer types, while `F_DINT_TO_WORD` handles the larger DINT type.

The `F_DINT_TO_WORD` function block is a specialized and efficient tool for converting 32-bit integers to 16-bit words. Its simplicity and determinism make it a reliable component in control applications, especially for data adaptation between different systems.

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

DT

DT Conversions

🎧 Podcast

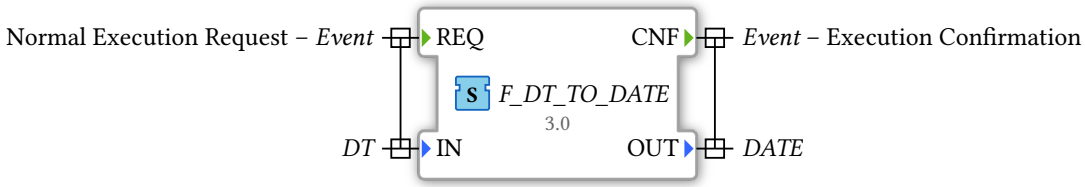
- ISOBUS Button: More Than Just a Click – The Standardization of Agricultural Technology
- Understanding Switch Logic: How a Toggle Flip-Flop Works with logiBUS® – Simple Control in Agricultural Technology
- As an Agricultural Technology Specialist Through Hell: How Lanz-Wery Survived War, Occupation, and Hyperinflation – Insights into Original Business Reports 1915-1922
- Old Bavarian for Beginners: From Gratler Colds to City Dwellers – A Loud Journey Through Lesson 3
- CAN Bus Diagnostics: Beyond the Data – The Truth About Errors in Agricultural Technology

]

F_DT_TO_DATE

• – ◦ • – ◦ • – ◦ •

The function block `F_DT_TO_DATE` converts a date-time value (DT) into a date (DATE). This functionality is particularly useful when control applications only require the date without time information.



- `REQ`: Starts the conversion process. Connected to the data input `IN`.
- `CNF`: Signals the completion of the conversion. Connected to the data output `OUT`.
- `IN` (DT): The input for the date-time value to be converted.
- `OUT` (DATE): The output for the converted date.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use any adapters.

Upon receiving a `REQ` event, the DT value at input `IN` is converted to a DATE value and output at `OUT`. Simultaneously, the `CNF` event is triggered to signal the completion of the operation.

- The function block implements the conversion using the function `DT_TO_DATE`.
- The conversion is atomic and occurs in a single execution step.

The function block has no internal states. Each `REQ` request is processed independently.

- Extracting the date from timestamps in logging systems
- Preprocessing date information for reports or visualizations
- Interfaces to systems that can only process DATE data types
- Compared to `F_DT_TO_TIME`, this function block only converts the date portion
- Similar to `F_DT_TO_TOD`, but focusing on the date instead of the time of day

The `F_DT_TO_DATE` function block offers a simple and efficient way to convert DT values into pure dates. Its lean implementation makes it particularly suitable for real-time applications where performance is critical.

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

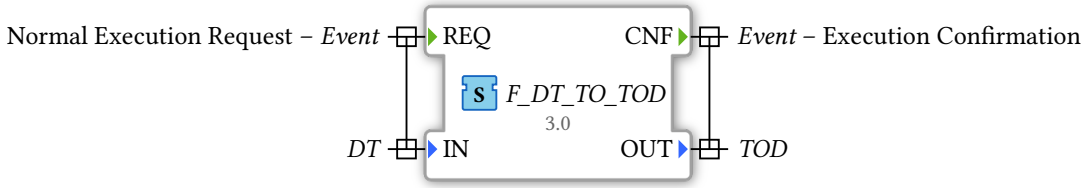
Conclusion

F_DT_TO_TOD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DT_TO_TOD` converts a date and time value (DT) into a pure time value (TOD - Time of Day). This function is particularly useful when only the time of day needs to be extracted from a combined date and time stamp.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the function block performs the conversion.

Event Outputs

- **CNF**: Signals the successful completion of the conversion.

Data Inputs

- **IN**: The input of type `DT`, containing the date and time value to be converted.

Data Outputs

- **OUT**: The output of type `TOD`, providing the converted time value (without the date).

Adapters

This function block does not use any adapters.

Operation

When the `REQ` event is triggered, the algorithm is executed, converting the input value `IN` of type `DT` into a `TOD` value. The result is output at `OUT`, and the `CNF` event signals the successful completion of the operation.

Technical Features

- The function block is implemented as a simple FB (SimpleFB), ensuring fast and efficient execution.
- The conversion is performed using the built-in function `DT_TO_TOD`.

State Overview

The function block has no internal states. The conversion is performed anew with each `REQ` event.

Application Scenarios

- Extracting the time of day from timestamps for time-based control systems.
- Preprocessing data for systems that only require the time of day.

⚖️ Comparison with Similar Function Blocks

- Unlike `F_TOD_TO_DT`, this function block converts in the opposite direction.
- Similar function blocks like `F_DT_TO_DATE` or `F_DT_TO_TIME` extract other parts of a date and time value.

Conclusion

The `F_DT_TO_TOD` function block provides a simple and efficient way to extract the time of day from a combined date and time value. Its clear interface and simple functionality make it a useful tool in control applications.

DWORD

DWORD conversions

Here you will find functions for converting the DWORD data type.

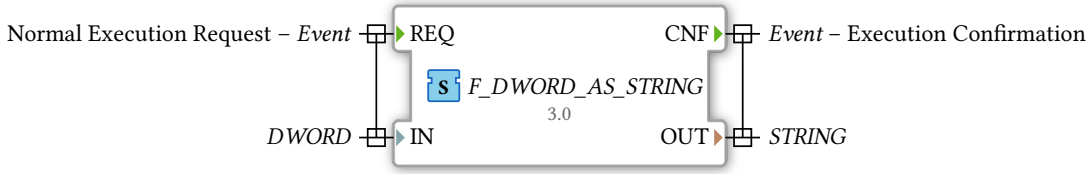
🛠️ Related exercises

- Uebung_034b
- SIN_TO_PCAN_Callback_DWord
- Uebung_152
- Uebung_153

F_DWORD_AS_STRING

• − ◦ • − ◦ • − ◦ •

The function block `F_DWORD_AS_STRING` is used to convert a DWORD value into a STRING. This function block is particularly useful when numeric values need to be converted into a human-readable format or prepared for text output.



- **REQ**: Starts the conversion of the DWORD value to a STRING. This input is connected to the data input `IN`.
- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.
- **IN**: The DWORD value to be converted to a STRING.
- **OUT**: The resulting STRING after the DWORD value has been converted.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The value at input `IN` (DWORD) is converted to a STRING and output at `OUT`. Successful conversion is indicated by the event `CNF`.

- The function block uses the function `DWORD_AS_STRING` for conversion.
- The conversion occurs synchronously with the triggering of the `REQ` event.

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Complete**: Signals completion with `CNF` and outputs the result to `OUT`.

- Conversion of numeric IDs or status values into readable text format.
- Preparation of data for logging or user display.
- Integration into systems that require string-based communication.
- Compared to generic conversion blocks, `F_DWORD_AS_STRING` specializes in converting DWORD to STRING and therefore offers a simple and direct solution for this specific use case.
- Other blocks might offer additional conversion options or parameters, but may be more complex to use.

The `F_DWORD_AS_STRING` function block is an efficient and easy-to-use tool for converting DWORD values to strings. Its clear interface structure and direct functionality make it ideal for applications requiring fast and reliable conversion.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

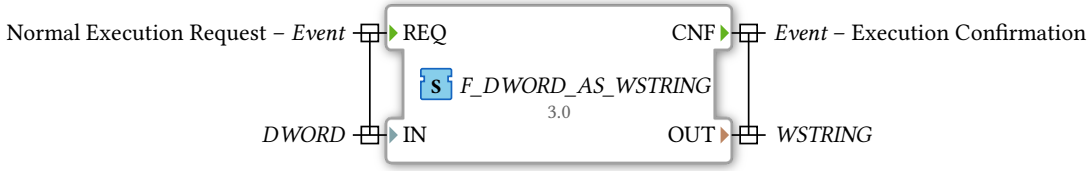
Conclusion

F_DWORD_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_AS_WSTRING` is used to convert a DWORD value to a WSTRING value. This function block is particularly useful in scenarios where numeric values need to be converted into a string, for example, for display or logging.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This input triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. This output signals the completion of the conversion.

Data Inputs

- **IN**: Input for the DWORD value to be converted.

Data Outputs

- **OUT**: Output for the resulting WSTRING value after conversion.

Adapters

- No adapters available.

Functionality

The function block performs the conversion from DWORD to WSTRING as soon as the event `REQ` is received. The conversion is carried out using the function `DWORD_AS_WSTRING`. After successful conversion, the event `CNF` is triggered, and the converted value is available at output `OUT`.

Technical Features

- The function block is implemented simply and efficiently, without additional state management.
- The conversion is performed directly and without delay.

State Overview

The function block has no internal states. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Displaying numeric values in a user interface.
- Logging numeric data in text format.
- Communicating with systems that require strings.

⚖️ Comparison with Similar Function Blocks

- Compared to generic conversion blocks, `F_DWORD_AS_WSTRING` specializes in converting DWORD to WSTRING and is therefore more efficient in this specific use case.
- Other blocks might have additional parameters or states, which is not necessary here.

Conclusion

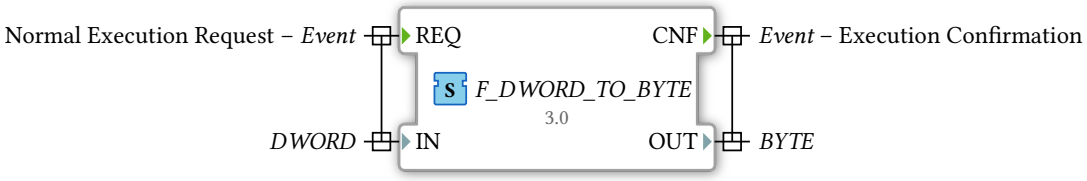
The `F_DWORD_AS_WSTRING` function block is a simple and efficient block for converting DWORD values to WSTRING. Its simplicity makes it particularly suitable for applications where fast and direct conversion is required.

F_DWORD_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_TO_BYTE` converts a 32-bit value (DWORD) to an 8-bit value (BYTE). This conversion is useful in scenarios where data reduction or type conversion is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `DWORD` (32-bit) to be converted to `BYTE`.

Data Outputs

- **OUT**: The output value of type `BYTE` (8-bit) containing the result of the conversion.

Adapters

- No adapters are available.

Functionality

This function block performs the conversion from `DWORD` to `BYTE` as soon as the event `REQ` is received. The conversion is performed using the function `DWORD_TO_BYTE`. After successful conversion, the event `CNF` is triggered.

Technical Details

- Data loss can occur if the original `DWORD` value is outside the representable range of `BYTE` (0-255).
- The function block is implemented as a simple FB (SimpleFB) and uses an ST algorithm for the conversion.

State Overview

1. **Idle**: Waits for the event `REQ`.
2. **Processing**: Performs the conversion.
3. **Completed**: Sends the event `CNF` and returns to the idle state.

Application Scenarios

- Data reduction in control systems.
- Type conversion in communication protocols.
- Adapting data for specific hardware interfaces.

⚖️ Comparison with similar function blocks

- **F_WORD_TO_BYTE**: Converts a 16-bit value (WORD) to a BYTE.
- **F_DWORD_TO_WORD**: Converts a 32-bit value (DWORD) to a 16-bit value (WORD).
- **F_BYTE_TO_DWORD**: Performs the reverse conversion (BYTE to DWORD).

Conclusion

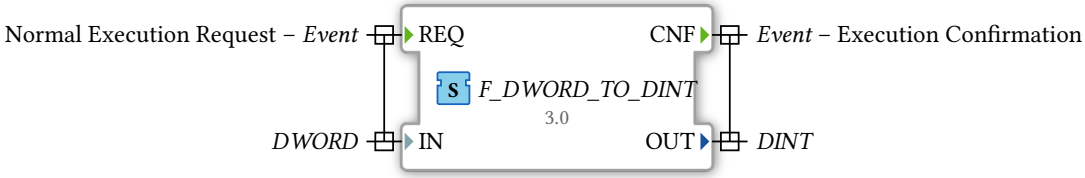
The function block `F_DWORD_TO_BYTE` provides a simple and efficient way to convert 32-bit values to 8-bit values. However, due to the potential for data loss with large values, its use should be carefully considered.

F_DWORD_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_TO_DINT` converts a `DWORD` value to a `DINT` value. This conversion is particularly necessary when exchanging data between systems that use different data types. The block is part of the `iec61131::conversion` package and conforms to standard 61499-1.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs

- **IN**: The input expects a `DWORD` value to be converted.

Data Outputs

- **OUT**: The output provides the converted `DINT` value.

Adapters

No adapters are present.

Functionality

The function block performs the conversion from `DWORD` to `DINT` as soon as the event `REQ` is received. The conversion is performed using the function `DWORD_TO_DINT(IN)`. After successful conversion, the event `CNF` is triggered, and the converted value is output via the output `OUT`.

Technical Features

- The block is implemented as a simple function block (`SimpleFB`).
- The conversion is performed directly without additional state management.

State Overview

Since it is a simple function block, there are no explicit states. The conversion is performed with each `REQ` event.

Application Scenarios

- Conversion of `DWORD` data supplied by one system into `DINT` data for another system.
- Use in control systems where data types need to be harmonized.

⚖️ Comparison with similar blocks

- Similar blocks could be `F_WORD_TO_INT` or `F_LWORD_TO_LINT`, but these convert different data types.
- Unlike more complex conversion blocks, `F_DWORD_TO_DINT` offers a direct and simple solution for this specific conversion.

Conclusion

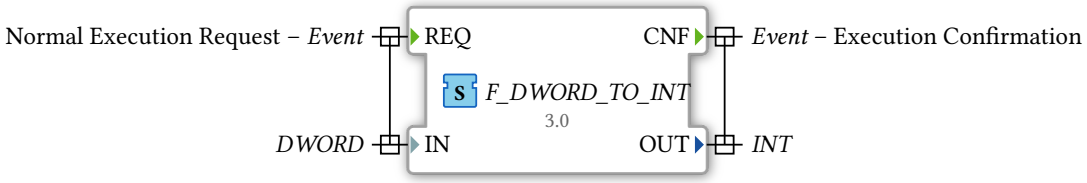
The `F_DWORD_TO_DINT` function block provides an efficient and simple way to convert `DWORD` values to `DINT` values. Its clear interface and direct operation make it ideal for applications requiring fast and reliable type conversion.

F_DWORD_TO_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_TO_INT` converts a DWORD value to an INT value. It is part of the package `iec61131::conversion` and enables easy conversion between these two data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion process. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `DWORD`, containing the value to be converted.

Data Outputs

- **OUT**: The output of type `INT`, providing the result of the conversion.

Adapters

This function block has no adapters.

Operation

When an event is received at input `REQ`, the algorithm is executed, converting the value of `IN` (of type `DWORD`) to a value of type `INT`. The result is output at `OUT` and confirmed by the event `CNF`.

Technical Features

- The function block uses the standard function `DWORD_TO_INT` for conversion.
- It is a simple function block without internal states or complex logic.

State Overview

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Conversion of DWORD values supplied by hardware or other systems into INT values for further processing.
- Integration into control systems that need to process different data types.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks such as `F_WORD_TO_INT` or `F_BYTE_TO_INT`, this function block is specifically optimized for converting DWORD to INT.
- It offers a simple and direct solution without additional functions or parameters.

Conclusion

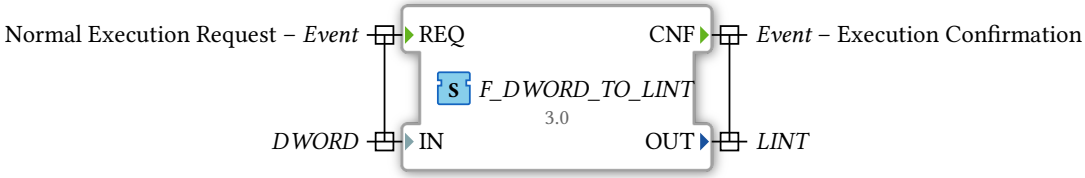
The `F_DWORD_TO_INT` function block is a useful tool for converting between DWORD and INT data types. Its simplicity and direct functionality make it a reliable solution for various use cases.

F_DWORD_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_TO_LINT` converts a `DWORD` value to a `LINT` value. This conversion is particularly useful when data needs to be exchanged between systems or components that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `DWORD`, which is to be converted to a `LINT` value.

Data Outputs

- **OUT**: The output value of type `LINT`, which contains the result of the conversion.

Adapters

This function block does not use any adapters.

Functionality

When an event is received at input `REQ`, the algorithm is executed. This algorithm converts the `DWORD` value at input `IN` into a `LINT` value and outputs the result at output `OUT`. The event `CNF` is then triggered to signal the completion of the conversion.

Technical Features

- The function block has a simple structure and contains only one algorithm that performs the conversion.
- The conversion is performed directly without any additional processing steps.

State Overview

The function block has no internal states. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Converting data between systems that use different data types.
- Use in control systems where type conversion is required.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion function blocks, `F_DWORD_TO_LINT` specializes in converting `DWORD` to `LINT`.
- Other function blocks could perform similar conversions for other data types, such as `F_WORD_TO_INT`.

Conclusion

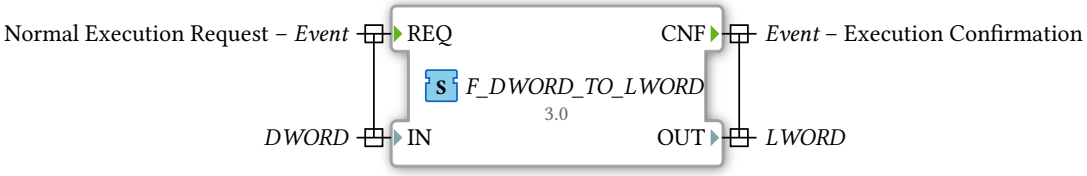
The `F_DWORD_TO_LINT` function block is a simple and efficient tool for converting `DWORD` values to `LINT` values. Its clear interface structure and direct functionality make it a reliable solution for type conversions in control systems.

F_DWORD_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_DWORD_TO_LWORD converts a 32-bit data value (DWORD) to a 64-bit data value (LWORD). This conversion is particularly useful in scenarios where an increase in data width is required to ensure compatibility with subsequent processing steps.



Interface Structure

Event Inputs

- **REQ** (Type: Event): Starts the execution of the function block. This input is linked to the data input IN.

Event Outputs

- **CNF** (Type: Event): Signals the successful completion of the conversion. The output is linked to the data output OUT.

Data Inputs

- **IN** (Type: DWORD): The 32-bit input value to be converted to a 64-bit value.

Data Outputs

- **OUT** (Type: LWORD): The resulting 64-bit output value after conversion.

Adapters

This function block does not have any adapters.

Functionality

This function block performs the DWORD to LWORD conversion by converting the 32-bit input value (IN) to a 64-bit value (OUT). The conversion occurs automatically upon receiving an event at input REQ. After successful conversion, the event CNF is triggered.

Technical Features

- The function block is implemented as a simple FB (SimpleFB) and uses an ST algorithm for the conversion.
- The conversion is performed without data loss, as the 32-bit value is embedded into the 64-bit value.

State Overview

The function block has no internal state. The conversion occurs immediately upon receiving the REQ event.

Application Scenarios

- Extending the data width for subsequent processing steps.
- Ensuring compatibility between systems that use different data widths.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks (e.g., F_WORD_TO_DWORD), this function block is specifically optimized for converting 32-bit to 64-bit values.
- Similar blocks might offer additional features such as scaling or offset adjustment, which is not the case here.

Conclusion

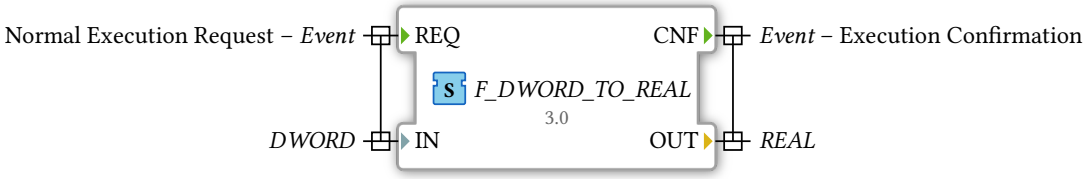
The F_DWORD_TO_LWORD function block is a simple and efficient tool for converting 32-bit to 64-bit values. Its clear interface structure and direct functionality make it ideal for applications that only require data width extension.

F_DWORD_TO_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_TO_REAL` converts a DWORD value (32-bit unsigned integer) to a REAL value (32-bit floating-point number). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is activated as soon as the conversion is complete and the result value is available at data output `OUT`.

Data Inputs

- **IN**: DWORD value to be converted to a REAL value.

Data Outputs

- **OUT**: The converted REAL value.

Adapters

No adapters available.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The value at input `IN` is converted to a REAL value using the function `DWORD_TO_REAL` and output at output `OUT`. Finally, the event `CNF` is triggered to signal the successful completion of the operation.

Technical Features

- The conversion is performed directly and without any additional delay.
- No special error handling is performed, e.g., for invalid input values.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Conversion State**: Performs the conversion and sets the output `OUT`.
3. **Confirmation State**: Enables `CNF` and returns to the idle state.

Application Scenarios

- Type conversion in control applications where DWORD data needs to be converted to floating-point numbers.
- Integration into larger function blocks that require type conversion.

⚖️ Comparison with similar function blocks

- Similar function blocks such as `F_WORD_TO_REAL` or `F_INT_TO_REAL` offer similar functionality, but for different input types.
- `F_DWORD_TO_REAL` is specifically optimized for converting DWORD to REAL.

Conclusion

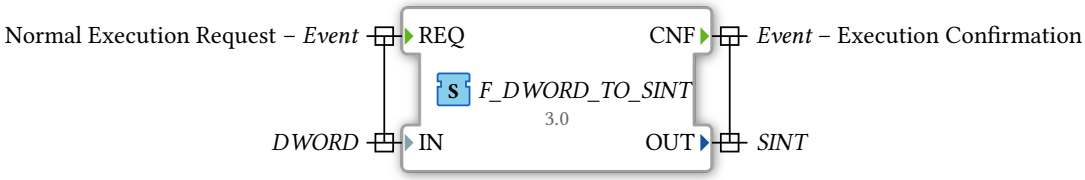
The `F_DWORD_TO_REAL` function block is a simple and efficient building block for converting DWORD to REAL values. Its clear interface and direct functionality make it a reliable component in control applications.

F_DWORD_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_TO_SINT` converts a 32-bit DWORD value to an 8-bit SINT value (signed integer). This block is part of the IEC 61131 conversion library and enables simple type conversion between these data types.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion process. This input is linked to the data input `IN`.

Event Outputs

- `CNF`: Signals successful completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- `IN` (DWORD): The 32-bit input value to be converted to a SINT value.

Data Outputs

- `OUT` (SINT): The converted 8-bit SINT value.

Adapters

This function block does not have any adapter interfaces.

Functionality

When the `REQ` event is triggered, the algorithm that converts the `IN` value of type DWORD into a SINT value is executed. The conversion is performed by the built-in function `DWORD_TO_SINT`. After the conversion is complete, the `CNF` event is triggered, and the converted value is available at the `OUT` output.

Technical Features

- Data loss can occur during the conversion if the DWORD value is outside the representable range of a SINT (-128 to 127).
- The function block is a simple block without internal state management.

State Overview

The function block has no internal states. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control applications where DWORD values need to be converted to more compact SINT values.
- Data reduction when only the least significant part of a DWORD value is required.

⚖️ Comparison with Similar Function Blocks

- Compared to `F_DWORD_TO_INT` or `F_DWORD_TO_USINT`, this function block converts to a signed 8-bit value.
- Similar function blocks for other target types are available in the IEC 61131 conversion library.

Conclusion

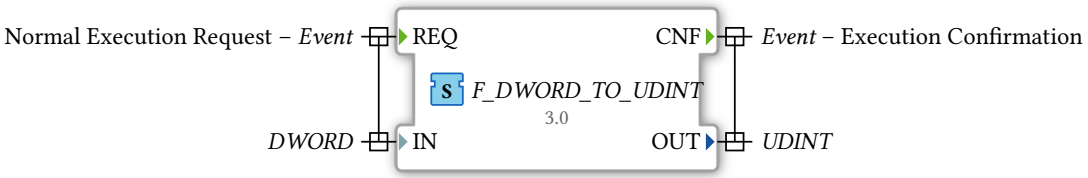
The `F_DWORD_TO_SINT` function block provides a simple and efficient way to convert DWORD values to SINT values. It is particularly useful in applications where storage space or data bandwidth needs to be optimized. When using it, the possible value range of the SINT data type must be considered to avoid unwanted data loss.

F_DWORD_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_TO_UDINT` converts a `DWORD` value to a `UDINT` value. This block is part of the `iec61131::conversion` package and enables simple and efficient type conversion.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `DWORD`, containing the value to be converted.

Data Outputs

- **OUT**: The output of type `UDINT`, providing the result of the conversion.

Adapters

- This function block does not use any adapters.

Functionality

The function block performs the conversion from `DWORD` to `UDINT` as soon as the `REQ` event is triggered. The conversion is performed using the built-in function `DWORD_TO_UDINT`. After the conversion is complete, the `CNF` event is triggered, and the converted value is available at output `OUT`.

Technical Features

- The function block has a simple structure and contains only one algorithm (`REQ`) that performs the conversion.
- The conversion is performed directly and without any additional delay.

State Overview

The function block has no internal state. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Type conversion in control applications where `DWORD` values need to be converted to `UDINT` values.
- Integration into larger control systems that require type compatibility between different data formats.

⚖️ Comparison with similar blocks

- Similar function blocks such as `F_WORD_TO_UINT` or `F_BYTE_TO_USINT` perform similar type conversions, but for different data types.
- Unlike generic conversion blocks, `F_DWORD_TO_UDINT` is specialized and optimized for converting `DWORD` to `UDINT`.

🛠️ Related exercises

- Uebung_011
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS

- Uebung_012b
- Uebung_012c
- Uebung_020c2_sub
- Uebung_028
- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_103
- Uebung_151
- Uebung_151_AX
- Uebung_152
- Uebung_153

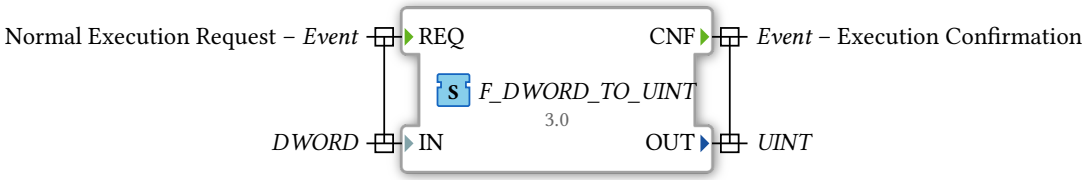
Conclusion

The `F_DWORD_TO_UDINT` function block provides a simple and efficient solution for converting `DWORD` values to `UDINT` values. Its clear interface structure and direct functionality make it a reliable component in control applications.

F_DWORD_TO_UINT

• – ◦ • – ◦ • – ◦ •

The function block `F_DWORD_TO_UINT` converts a `DWORD` value to a `UINT` value. This function block is particularly useful in scenarios where type conversion between these two data types is required, for example, in communication between different systems or when processing data from different sources.



- **REQ**: This input triggers the execution of the function block. The associated data value `IN` is processed upon this event.
- **CNF**: This output is activated after successful conversion and confirms execution. The converted value `OUT` is output with this event.
- **IN** (`DWORD`): The input value to be converted from `DWORD` to `UINT`.
- **OUT** (`UINT`): The output value after conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use adapters.

The function block performs the conversion from `DWORD` to `UINT` as soon as the event `REQ` is received. The conversion is performed using the function `DWORD_TO_UINT`, which converts the input value `IN` and outputs the result to `OUT`. After successful conversion, the event `CNF` is triggered.

- The function block is a simple algorithm (`SimpleFB`) and does not perform any complex state transitions.
- The conversion occurs directly and without delay.

Since it is a simple function block, there are no complex state transitions. The block responds exclusively to the `REQ` event, outputting the converted value, and to the `CNF` event.

- Type conversion in communication interfaces where different systems use different data types.
- Data processing in control systems where conversion between `DWORD` and `UINT` is required.
- Compared to other conversion blocks such as `F_WORD_TO_UINT` or `F_BYTE_TO_UINT`, this one is specifically optimized for converting `DWORD` to `UINT`.
- Compared to other conversion blocks like `F_WORD_TO_UINT` or `F_BYTE_TO_UINT`, this one is specifically optimized for converting `DWORD` to `UINT`.

Similar function blocks might offer additional features such as range checks or error handling, which this function block does not.

The `F_DWORD_TO_UINT` function block is a simple and efficient tool for converting `DWORD` values to `UINT` values. Its simplicity makes it particularly suitable for applications where fast and direct type conversion is required.

Functionality

Technical Features

State Overview

Application Scenarios

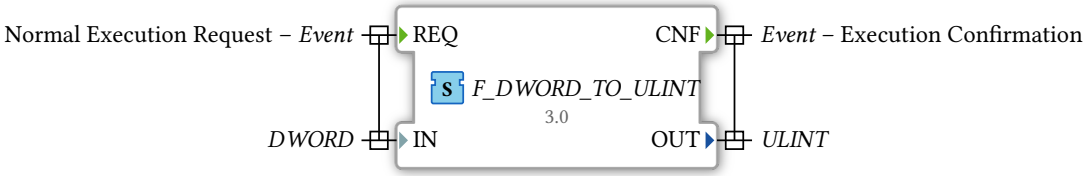
⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

F_DWORD_TO_ULINT

• – ◦ • – ◦ • – ◦ •

The function block `F_DWORD_TO_ULINT` converts a `DWORD` value to a `ULINT` value. It is part of the `iec61131::conversion` package and is typically used in automation applications where type conversion between these two data types is required.



- **REQ**: This input triggers the execution of the function block. It is linked to the data input `IN`.
- **CNF**: This output signals the successful execution of the function block. It is linked to the data output `OUT`.
- **IN**: The input expects a `DWORD` value, which is to be converted to a `ULINT` value.
- **OUT**: The output provides the converted `ULINT` value.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use any adapters.

The function block performs the conversion from `DWORD` to `ULINT` as soon as the `REQ` event arrives. The algorithm within the function block uses the function `DWORD_TO_ULINT` to carry out the conversion. After successful conversion, the `CNF` event is triggered, and the converted value is available at the `OUT` output.

- The function block is implemented as a simple function block (`SimpleFB`).

- The conversion occurs directly and without any additional delay.
- The function block is licensed under the Eclipse Public License 2.0.

This function block has no internal state. Execution occurs exclusively upon the occurrence of the `REQ` event.

- Type conversion in automation applications where `DWORD` values need to be converted to `ULINT` values.
- Integration into larger control systems that process different data types.
- Similar function blocks such as `F_WORD_TO_UINT` or `F_BYTE_TO_USINT` perform similar conversions, but for different data types.
- `F_DWORD_TO_ULINT` specializes in converting `DWORD` to `ULINT` and offers no additional functionality.

The `F_DWORD_TO_ULINT` function block is a simple and efficient tool for type conversion in IEC 61131-3-based control systems. Its clear interface and direct operation make it a reliable component in automation projects.

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

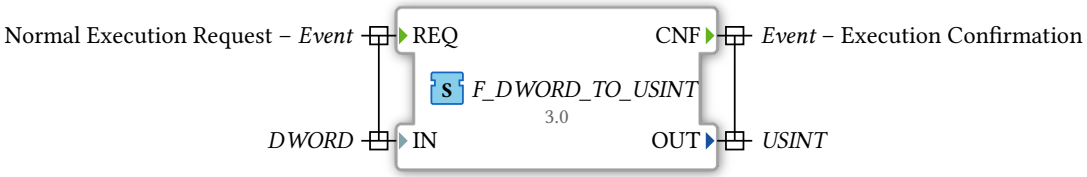
Conclusion

F_DWORD_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_TO_USINT` converts a 32-bit value (DWORD) to an 8-bit unsigned integer (USINT). This conversion is particularly useful when data needs to be exchanged between systems with different word lengths.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at data output `OUT`.

Data Inputs

- **IN** (DWORD): The 32-bit value to be converted to an 8-bit unsigned integer.

Data Outputs

- **OUT** (USINT): The resulting 8-bit unsigned integer after conversion.

Adapters

This function block does not have any adapter interfaces.

Operation

The function block performs the conversion by transforming the 32-bit value at input `IN` into an 8-bit unsigned integer. The conversion is performed by the function `DWORD_TO_USINT`, which converts the value accordingly. During the conversion, only the least significant byte of the DWORD value is considered, while the higher-order bytes are ignored.

Technical Features

- The conversion truncates the higher-order bytes, which can lead to data loss if the original DWORD value is greater than 255.
- The function block is deterministic and performs the conversion in a single execution step.

State Overview

The function block has no internal state. The conversion is performed anew each time the `REQ` event is triggered.

Application Scenarios

- Data reduction in communication protocols where only 8-bit values can be transmitted.
- Adapting data formats between different system components.
- Use in controllers where specific hardware can only process 8-bit data.

⚖️ Comparison with Similar Function Blocks

- **F_WORD_TO_USINT**: Converts a 16-bit value (WORD) to an 8-bit unsigned integer. In contrast, `F_DWORD_TO_USINT` processes 32-bit values.
- **F_DWORD_TO_BYTE**: Similar in functionality, but the output type is BYTE, which is also 8 bits, but unsigned.

Conclusion

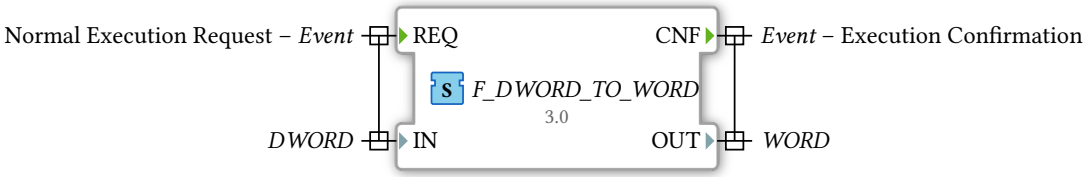
The function block `F_DWORD_TO_USINT` is a simple and efficient tool for converting 32-bit values to 8-bit unsigned integers. It is particularly suitable for applications where data needs to be reduced to a smaller word width. However, due to the potential for data loss with large input values, its use should be carefully planned.

F_DWORD_TO_WORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_DWORD_TO_WORD` converts a 32-bit DWORD value to a 16-bit WORD value. This conversion is useful when data needs to be exchanged between systems with different word lengths or when memory space needs to be optimized.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN** (DWORD): The 32-bit input value to be converted.

Data Outputs

- **OUT** (WORD): The converted 16-bit output value.

Adapters

This function block does not use adapters.

Operation

When the `REQ` event is triggered, the algorithm is executed that converts the DWORD value at the input `IN` to a WORD value and outputs the result to `OUT`. The `CNF` event is then triggered to signal the completion of the conversion.

Technical Features

- The conversion is performed by truncating the 16 most significant bits of the DWORD value.
- No check is performed to prevent potential data loss during this conversion.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Communication between systems with different word widths
- Memory optimization in embedded systems
- Data preprocessing in control systems

⚖️ Comparison with Similar Function Blocks

- Unlike `F_DWORD_TO_BYTE`, this function block converts to a WORD value instead of a BYTE.
- Similar to `F_WORD_TO_BYTE`, but with different data types.

Conclusion

The `F_DWORD_TO_WORD` function block offers a simple and efficient way to convert DWORD values to WORD values. It is particularly useful in systems that need to handle different word widths. However, be aware of the potential for data loss during the conversion.

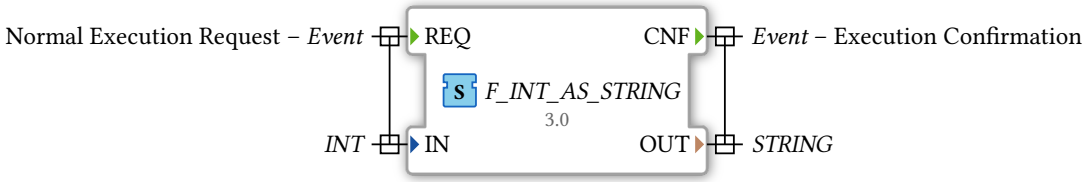
INT

F_INT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_AS_STRING` converts an integer value (`INT`) into a string (`STRING`). This functionality is particularly useful when numeric values need to be output in a user-readable format or further processed in a word processor.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion of the input value. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN** (`INT`): The integer value to be converted to a string.

Data Outputs

- **OUT** (`STRING`): The resulting string after the input value has been converted.

Adapters

- This function block has no adapters.

Functionality

When the event `REQ` is triggered, the algorithm is executed that converts the integer value `IN` to a string and outputs the result to `OUT`. The event `CNF` is then triggered to signal the successful completion of the conversion.

Technical Features

- The conversion is performed using the function `INT_AS_STRING`, which directly converts the `INT` value into a `STRING` value.
- The function block is simple and efficient because it does not require complex state machines or additional processing steps.

State Overview

Because it is a simple function block, there are no explicit states. The functionality is executed directly upon triggering the `REQ` event.

Application Scenarios

- Displaying numeric values in user interfaces or logs.
- Preparing data for output in text formats (e.g., CSV, JSON).
- Integration into systems that require strings for further processing.

⚖️ Comparison with similar building blocks

- Compared to generic conversion blocks, `F_INT_AS_STRING` specializes in converting `INT` to `STRING` and is therefore more efficient and easier to use.
- Other blocks might support additional parameters such as formatting or locale settings, which is not the case here.

Conclusion

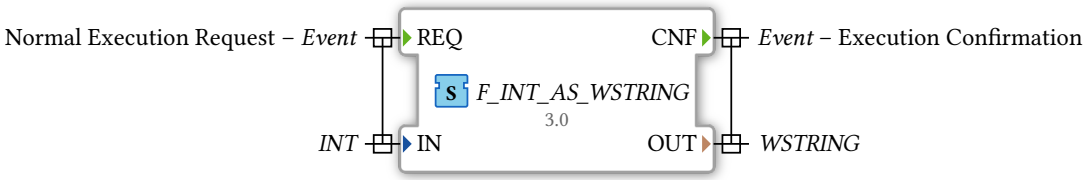
The `F_INT_AS_STRING` function block offers a simple and effective way to convert integer values to strings. Its simplicity and direct functionality make it ideal for applications that require fast and straightforward conversion.

F_INT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_AS_WSTRING` converts an integer value (`INT`) into a wide string (`WSTRING`). This functionality is particularly useful when numeric values are needed as text for display or further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at data output `OUT`.

Data Inputs

- **IN**: The integer value (`INT`) to be converted into a wide string.

Data Outputs

- **OUT**: The resulting wide string (`WSTRING`) after conversion.

Adapters

- No adapters are available.

Functionality

The function block uses the built-in function `INT_AS_WSTRING` to convert the integer input value into a wide string. The conversion is initiated by the event `REQ` and confirmed by the event `CNF`.

Technical Features

- The function block has a simple structure and performs only basic data type conversion.
- No additional parameters or complex algorithms are required.

State Overview

1. **Ready**: Waits for the event `REQ`.
2. **Active**: Performs the conversion and sets the output `OUT`.
3. **Confirmed**: Sends the event `CNF` and returns to the ready state.

Application Scenarios

- Displaying numeric values in user interfaces.
- Logging numeric data in text format.
- Preparing data for communication with systems that expect text formats.

⚖️ Comparison with Similar Function Blocks

- **F_DINT_AS_WSTRING**: Converts a long integer value (`DINT`) to a wide string.
- **F_REAL_AS_WSTRING**: Converts a floating-point value (`REAL`) to a wide string.
- **F_STRING_AS_INT**: Performs the reverse conversion (string to integer).

Conclusion

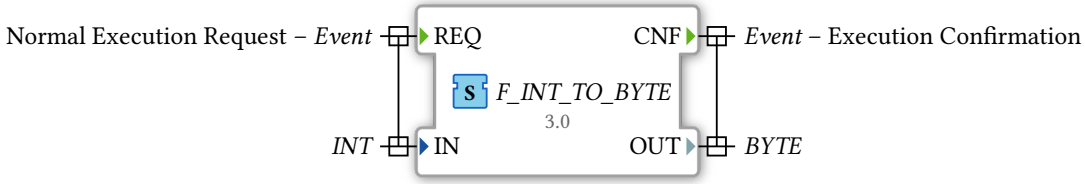
The `F_INT_AS_WSTRING` function block is a simple and efficient tool for converting integers to wide strings. Its clear interface and direct functionality make it a reliable solution for basic data type conversions.

F_INT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_BYTE` is used to convert an integer value (`INT`) to a byte value (`BYTE`). It is part of the package `iec61131::conversion` and provides a simple and efficient way to convert data types within IEC 61499 systems.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN**: The integer value (`INT`) to be converted to a byte value.

Data Outputs

- **OUT**: The resulting byte value (`BYTE`) after the conversion.

Adapters

This function block does not use any adapters.

Functionality

When an event is received at input `REQ`, the algorithm is executed, converting the value of `IN` from `INT` to `BYTE` and outputting the result to `OUT`. The event `CNF` is then triggered to signal the successful completion of the conversion.

The algorithm is defined as follows:
ALGORITHM REQ OUT := INT_TO_BYTE(IN); END_ALGORITHM

Technical Features

- The function block is implemented simply and efficiently, without additional states or complex logic.
- The conversion is performed directly and without delay.

State Overview

Since it is a simple function block, there are no explicit states. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control applications where integer values need to be converted to byte values.
- Integration into larger systems that require strict type control.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_DINT_TO_BYTE` or `F_WORD_TO_BYTE` offer conversions between other data types but are designed for different source types.
- `F_INT_TO_BYTE` is specifically optimized for converting `INT` to `BYTE`.

Conclusion

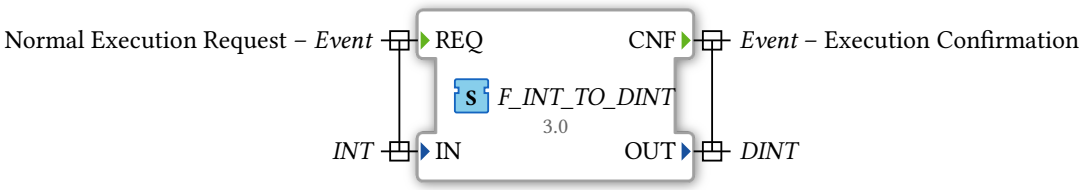
The function block `F_INT_TO_BYTE` is a useful tool for type conversion in IEC 61499-based systems. Its simple and efficient implementation makes it ideal for applications requiring fast and reliable conversions.

F_INT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_DINT` is used to convert a 16-bit integer value (`INT`) to a 32-bit integer value (`DINT`). This conversion is useful when exchanging data between systems or components that use different integer formats.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (`INT`): The 16-bit integer value to be converted.

Data Outputs

- **OUT** (`DINT`): The resulting 32-bit integer value after the conversion.

Adapters

This function block does not use adapters.

Operation

The function block performs a direct type conversion. When the `REQ` event occurs, the value of `IN` is copied to `OUT`. The 16-bit `INT` value is automatically expanded to a 32-bit `DINT` value. After the conversion is complete, the `CNF` event is triggered.

Technical Features

- The conversion is lossless, as a `INT` value can be converted to a `DINT` value without any issues.
- The function block is deterministic and has no internal states.

State Overview

The function block has no internal states. It reacts exclusively to the `REQ` event with an immediate conversion and output of the `CNF` event.

Application Scenarios

- Data communication between systems that use different integer formats.
- Extending integer values for further calculations or storage.
- Compatibility with libraries or functions that require `DINT` values.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_INT_TO_DINT` is specialized and optimized for converting `INT` to `DINT`.
- Other conversion blocks might offer additional features such as overflow checks or scaling, which are not necessary here.

Conclusion

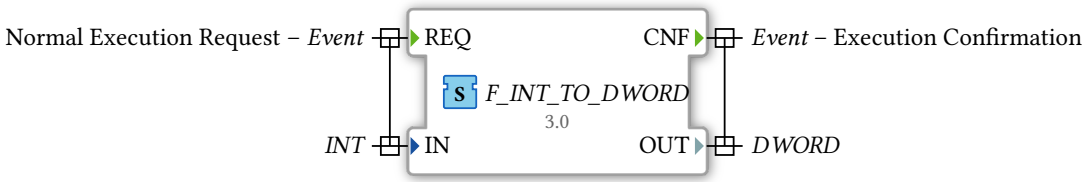
The `F_INT_TO_DINT` function block is a simple and efficient building block for lossless conversion of 16-bit to 32-bit integer values. It is particularly useful in scenarios where type conversion without additional calculations is required.

F_INT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_DWORD` converts an integer value (INT) to a double-word value (DWORD). This conversion is particularly useful when exchanging data between systems that use different data types.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs
- **CNF**: Confirms the successful execution of the conversion. This output is linked to the data output `OUT`.

Data Inputs
- **IN** (INT): The integer value to be converted to a DWORD value.

Data Outputs
- **OUT** (DWORD): The resulting DWORD value after conversion.

Adapters
This function block does not use any adapters.

Functionality
The function block performs the INT to DWORD conversion as soon as the event `REQ` is triggered. The conversion is carried out using the function `INT_TO_DWORD`, which converts the input value `IN` to the output value `OUT`. After successful conversion, the event `CNF` is triggered.

Technical Features
- This function block is a simple function block (SimpleFB) and uses an ST algorithm for conversion.
- The conversion is performed directly and without any additional delay.

State Overview
The function block has no internal states. The conversion occurs immediately after the `REQ` event is triggered.

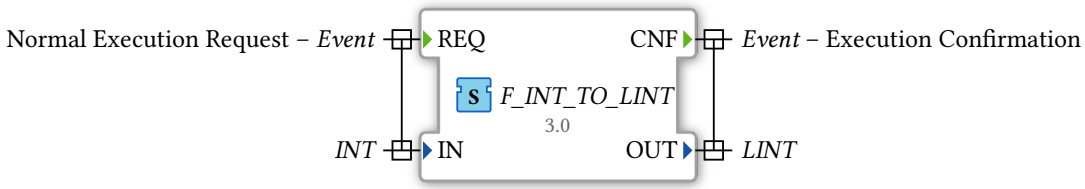
Application Scenarios
- Data conversion in control systems that use different data types.
- Integration of systems that generate INT values but require DWORD values.

⚖️ Comparison with Similar Function Blocks
- Compared to other conversion blocks such as `F_INT_TO_REAL` or `F_DWORD_TO_INT`, this function block specializes in converting INT to DWORD.
- Simpler and more direct than universal conversion blocks that support multiple data types.

Conclusion
The `F_INT_TO_DWORD` function block is an efficient and simple tool for converting integer to double-word values. Its clear interface and direct functionality make it ideal for applications requiring fast and reliable data type conversion.

F_INT_TO_LINT
• – ◦ • – ◦ • – ◦ •

Introduction
The function block `F_INT_TO_LINT` converts a 16-bit integer value (`INT`) into a 64-bit long integer value (`LINT`). This block is particularly useful in scenarios where extending the value range or changing the type is required for subsequent processing steps.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs
- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs
- **IN** (`INT`): The 16-bit integer value to be converted.

Data Outputs
- **OUT** (`LINT`): The resulting 64-bit long integer value after the conversion.

Adapters
No adapters are present.

Functionality
The function block performs a direct assignment of the input value `IN` of type `INT` to the output value `OUT` of type `LINT`. The conversion is performed implicitly through type conversion, preserving the numeric value.

Technical Features
- **Simple Implementation**: The algorithm consists of a single assignment.
- **No Data Loss**: Since `LINT` has a larger range of values than `INT`, there are no overflow problems.
- **Deterministic Behavior**: Execution is predictable and fast.

State Overview
The function block has no internal state. Every call to `REQ` results in immediate execution and output via `CNF`.

Application Scenarios
- **Data Processing**: Extending integer values for subsequent calculations with a larger value range.
- **Interface Adaptation**: Adapting data for components that expect `LINT`.
- **Logging**: Storing `INT` values in systems that use `LINT`.

⚖️ Comparison with Similar Function Blocks
- **F_INT_TO_DINT**: Converts `INT` to `DINT` (32-bit), offering a smaller value range than `LINT`.
- **F_DINT_TO_LINT**: Converts `DINT` to `LINT`, similar but with a different input type.
- **F_LINT_TO_INT**: Reverse conversion, potentially resulting in data loss.

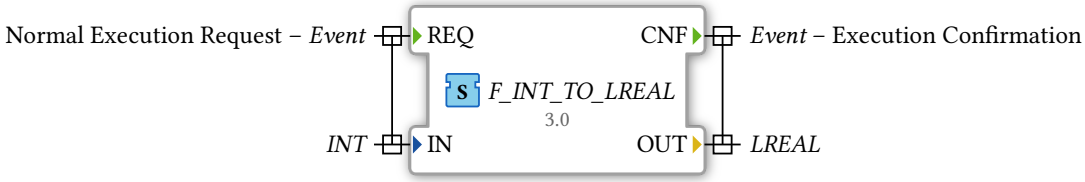
Conclusion
The `F_INT_TO_LINT` function block is a simple and efficient tool for type conversion in the 4diac IDE. Its strength lies in its direct and lossless conversion, making it ideal for applications with extended value requirements.

🌐 Related topic subpages on ms-muc-docs.de
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

F_INT_TO_LREAL
• – ◦ • – ◦ • – ◦ •

Introduction
The function block `F_INT_TO_LREAL` converts an integer value (INT) into a 64-bit floating-point number (LREAL). This block is particularly useful in applications where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `INT`, which is to be converted to a value of `LREAL`.

Data Outputs

- **OUT**: The output value of type `LREAL`, which contains the result of the conversion.

Adapters

- No adapters are available.

Functionality

The function block performs the conversion by directly assigning the integer input value `IN` to the floating-point output value `OUT`. The conversion occurs automatically when an event arrives at the `REQ` input. After the conversion is complete, the `CNF` event is triggered.

Technical Features

- The conversion is performed without additional parameters or configuration.
- The function block is optimized for use in the 4diac IDE and supports real-time processing.

State Overview

1. **Idle State**: Waits for a `REQ` event.
2. **Conversion State**: Performs the type conversion.
3. **Acknowledge State**: Sends the `CNF` event and returns to the idle state.

Application Scenarios

- Type conversion in control applications requiring floating-point operations.
- Integration into larger control systems that process mixed data types.

⚖️ Comparison with Similar Function Blocks

- Compared to generic conversion blocks, `F_INT_TO_LREAL` specializes in converting `INT` to `LREAL` and is therefore more efficient at this specific task.
- Other function blocks might offer additional features such as scaling or range checking, which is not the case here.

Conclusion

The `F_INT_TO_LREAL` function block is a simple and efficient building block for converting integer values to floating-point numbers. Its simplicity and direct operation make it a reliable choice for basic type conversions in automation technology.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

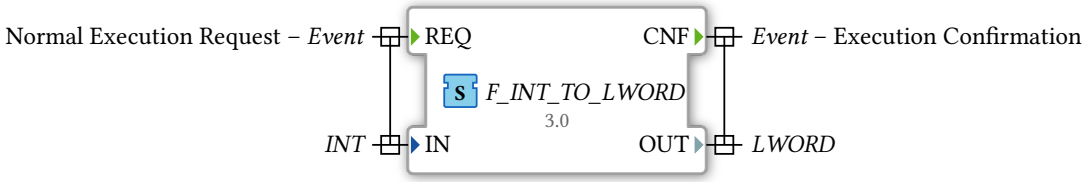
]

F_INT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_LWORD` converts an integer value (`INT`) into a 64-bit unsigned integer value (`LWORD`). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Triggers the execution of the function block. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals successful execution of the function block. This output is connected to the data output `OUT`.

Data Inputs

- **IN** (`INT`): The input value to be converted to a `LWORD` value.

Data Outputs

- **OUT** (`LWORD`): The converted output value.

Adapters

No adapters available.

Functionality

Upon receiving an incoming `REQ` event, the function block converts the `INT` value at the input `IN` to a `LWORD` value. The result is output at `OUT`, and the `CNF` event signals successful execution.

The algorithm is defined as follows:
ALGORITHM REQ OUT := INT_TO_LWORD(IN); END_ALGORITHM

Technical Features

- The function block is simple and efficient because it directly utilizes the built-in `INT_TO_LWORD` function.
- No additional states or complex logic are required.

State Overview

The function block has no internal states. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control applications where `INT` values need to be converted to `LWORD` values.
- Integration into larger systems that process different data types.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_DINT_TO_LWORD` or `F_SINT_TO_LWORD` offer conversions from other integer types to `LWORD`.
- `F_INT_TO_LWORD` is specifically optimized for converting `INT` to `LWORD`.

Conclusion

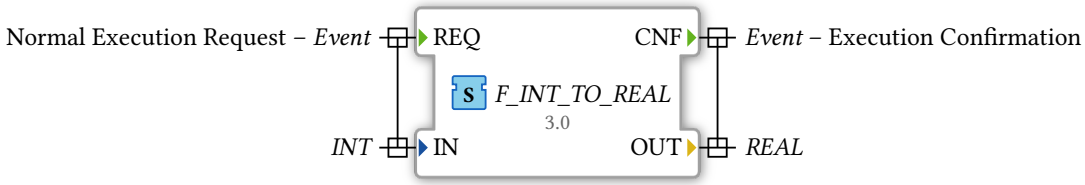
The `F_INT_TO_LWORD` function block provides a simple and efficient solution for converting `INT` values to `LWORD` values. Its clear interface and direct functionality make it ideal for use in control applications.

F_INT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_REAL` converts an integer value (`INT`) into a floating-point number (`REAL`). This functionality is particularly useful when data needs to be converted between different numerical representations in control applications.



Interface Structure

Event Inputs

- **REQ**: This input triggers the execution of the conversion. It is linked to the data input `IN`.

Event Outputs

- **CNF**: This output signals the completion of the conversion. It is linked to the data output `OUT`.

Data Inputs

- **IN** (`INT`): The integer value to be converted to a floating-point number.

Data Outputs

- **OUT** (`REAL`): The resulting floating-point value after conversion.

Adapters

This function block does not use any adapters.

Functionality

When the event `REQ` is triggered, the value of the data input `IN` is read and converted to the type `REAL`. The result is output at data output `OUT`, and the event `CNF` signals the successful completion of the operation.

Technical Features

- The conversion is performed directly without additional parameters or settings.
- The function block is implemented simply and efficiently, ensuring fast execution.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of sensor values (e.g., acquired as `INT`) into floating-point numbers for further calculations.
- Integration into control logic that needs to process mixed data types.

⚖️ Comparison with Similar Function Blocks

- Compared to generic conversion blocks, `F_INT_TO_REAL` is specialized and therefore more efficient for this specific use case.
- Other blocks might offer additional features such as range checks or scaling, which are not required here.

Conclusion

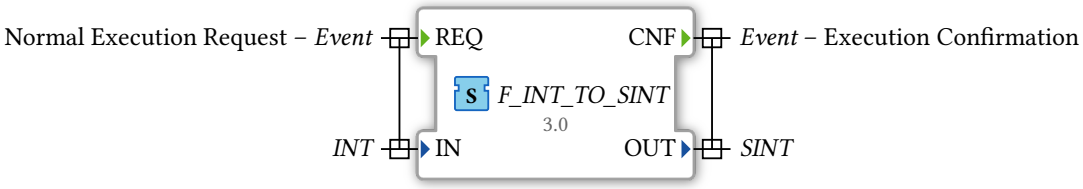
The `F_INT_TO_REAL` function block is a simple and effective tool for converting integer values to floating-point numbers. Its clear interface and direct functionality make it ideal for applications requiring fast and reliable type conversion.

F_INT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_SINT` converts an integer value of type `INT` into a signed 8-bit integer value of type `SINT`. This conversion is useful when data needs to be exchanged or processed between systems with different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `INT` to be converted.

Data Outputs

- **OUT**: The converted output value of type `SINT`.

Adapters

No adapters are present.

Functionality

This function block performs the conversion from `INT` to `SINT` as soon as the event `REQ` is triggered. The algorithm within the block uses the function `INT_TO_SINT` to perform the conversion. After the conversion is complete, the event `CNF` is triggered, and the converted value is available at the output `OUT`.

Technical Features

- The function block has a simple structure and contains only one algorithm for the conversion.
- There is no additional error handling for overflow or underflow conditions. The user must ensure that the input value is within the valid range for `SINT` (-128 to 127).

State Overview

The function block has no internal states. The conversion occurs immediately after the `REQ` event is triggered.

Application Scenarios

- Data conversion in control systems where different data types need to be processed.
- Communication between systems that use different data types.
- Reducing memory usage when a smaller data type is sufficient.

⚖️ Comparison with Similar Function Blocks

- **F_INT_TO_USINT**: Converts `INT` to `USINT` (unsigned 8-bit integer).
- **F_INT_TO_DINT**: Converts `INT` to `DINT` (32-bit integer), which does not involve data reduction.
- **F_INT_TO_REAL**: Converts `INT` to `REAL` (floating-point number).

Conclusion

The `F_INT_TO_SINT` function block is a simple and efficient tool for converting `INT` to `SINT` values. It is particularly suitable for applications that need to optimize storage space or exchange data between

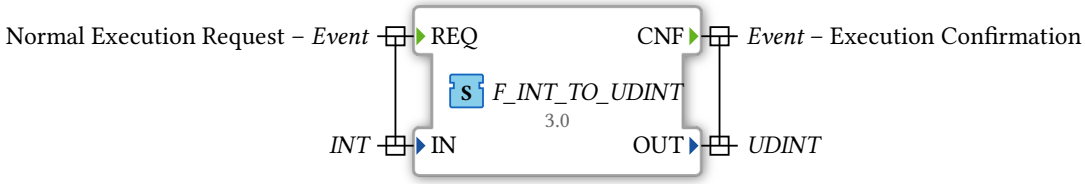
systems with different data types. However, users should ensure that the input values are within the valid range for `SINT` to avoid unexpected results.

F_INT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_UDINT` converts a signed 16-bit integer value (`INT`) into an unsigned 32-bit integer value (`UDINT`). This conversion is particularly necessary in scenarios where data needs to be exchanged between systems with different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The signed 16-bit integer value (`INT`) to be converted.

Data Outputs

- **OUT**: The resulting unsigned 32-bit integer value (`UDINT`).

Adapters

- No adapters are present.

Functionality

This function block performs the conversion from `INT` to `UDINT` as soon as the event `REQ` is received. The conversion is performed using the function `INT_TO_UDINT(IN)`, which converts the input value `IN` into the corresponding value `UDINT`. After successful conversion, the event `CNF` is triggered, and the converted value is available at the output `OUT`.

Technical Features

- The conversion is performed without additional parameters or configuration.
- The function block is suitable for use in real-time systems because it operates deterministically.

State Overview

1. **Idle State**: Waits for the event `REQ`.
2. **Conversion State**: Performs the conversion and sets the output `OUT`.
3. **Acknowledgement State**: Triggers the event `CNF` and returns to the idle state.

Application Scenarios

- Data conversion in control systems that use different data types.
- Integration of sensor data in `INT` format into systems requiring `UDINT` format.
- Compatibility layer between different hardware or software components.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_DINT_TO_UDINT` or `F_SINT_TO_UDINT` offer conversions between other data types, but not specifically for `INT` to `UDINT`.

Unlike generic conversion blocks, `F_INT_TO_UDINT` is optimized for a specific conversion.

Conclusion

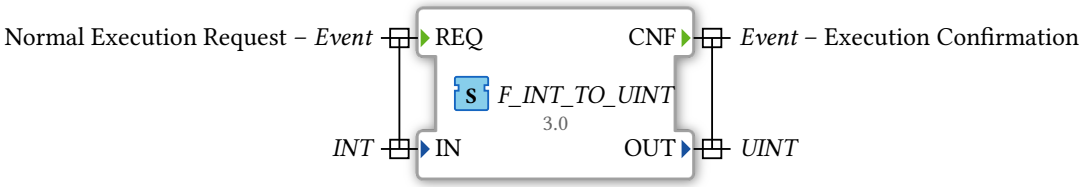
The function block `F_INT_TO_UDINT` is a specialized and efficient tool for converting signed 16-bit integer values to unsigned 32-bit integer values. Its simple interface and deterministic operation make it ideal for use in real-time applications.

F_INT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_UINT` converts a signed integer value (`INT`) to an unsigned integer value (`UINT`). This conversion is useful when exchanging data between systems that use different data types, or when unsigned processing is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN** (`INT`): The signed integer value to be converted.

Data Outputs

- **OUT** (`UINT`): The resulting unsigned integer value after the conversion.

Adapters

No adapters are used.

Functionality

The function block performs the conversion from `INT` to `UINT` when the event `REQ` is triggered. The conversion is performed using the function `INT_TO_UINT(IN)`. After successful conversion, the event `CNF` is triggered, and the converted value is available at output `OUT`.

Technical Features

- The conversion is performed without additional parameters or configuration.
- The block is implemented as a simple function block (`SimpleFB`) and uses an ST algorithm for the conversion.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Conversion State**: Performs the conversion and sets the output `OUT`.
3. **Acknowledgement State**: Triggers the `CNF` event and returns to the idle state.

Application Scenarios

- **Data Processing**: Converts sensor values or control data in `INT` format for systems that expect `UINT`.
- **Communication**: Prepares data for transmission between systems with different data type requirements.

⚖️ Comparison with Similar Blocks

- **F_UINT_TO_INT**: Performs the reverse conversion from `UINT` to `INT`.
- **Generic Conversion Blocks**: May offer more flexibility but are also more complex to use.

Conclusion

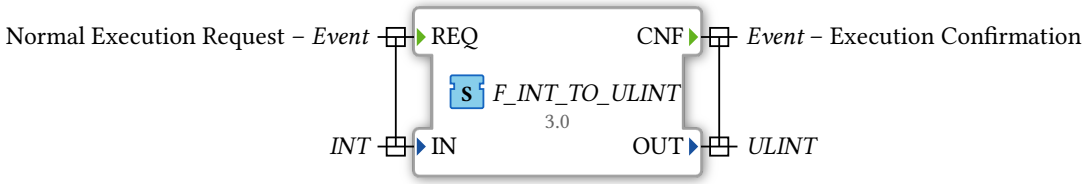
The `F_INT_TO_UINT` function block is a simple and efficient tool for converting signed to unsigned integer values. Its clear interface and simple functionality make it ideal for applications requiring fast and reliable type conversion.

F_INT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_ULINT` converts an integer value of data type `INT` into an unsigned 64-bit integer value of data type `ULINT`. This conversion is particularly useful when values need to be exchanged between different systems or components that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion of the input value. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion and outputs the converted value via the data output `OUT`.

Data Inputs

- **IN**: The input value of data type `INT` to be converted.

Data Outputs

- **OUT**: The converted output value of data type `ULINT`.

Adapters

This function block does not use any adapters.

Operation

The function block performs the conversion as soon as the event `REQ` is received. The algorithm within the function block converts the `INT` value at input `IN` into a `ULINT` value and outputs the result at `OUT`. Successful conversion is signaled by triggering the `CNF` event.

Technical Features

- The conversion is performed directly and without any additional delay.
- The function block is suitable for use in real-time systems.
- The implementation complies with the IEC 61131-3 standard.

State Overview

The function block has no internal states. The conversion is performed each time the `REQ` event is received.

Application Scenarios

- Conversion of sensor values for processing in systems that require `ULINT` values.
- Integration of components that use different data types.
- Use in control systems that require type conversion.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks such as `F_DINT_TO_ULINT` or `F_SINT_TO_ULINT`, this function block is specifically optimized for converting `INT` to `ULINT`.
- Similar function blocks might offer additional features such as range checks or error handling, which this function block does not.

Conclusion

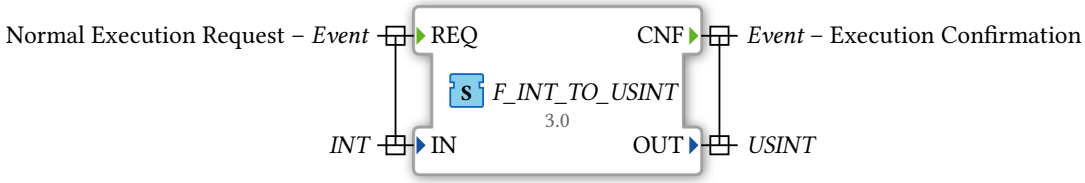
The function block `F_INT_TO_ULINT` is a simple and efficient building block for converting `INT` values to `ULINT` values. It is particularly useful in scenarios where direct and fast type conversion is required. Its clear interface structure and simple functionality make it a reliable tool in automation technology.

F_INT_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_USINT` converts an integer value of data type `INT` (16-bit signed integer) to data type `USINT` (8-bit unsigned integer). This block is particularly useful in scenarios where type conversion between numeric values of different sizes is required.



Interface Structure

Event Inputs

- **REQ** (Type: `Event`): Starts the execution of the conversion. The input is linked to the data input `IN`.

Event Outputs

- **CNF** (Type: `Event`): Signals successful completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (Type: `INT`): The input value to be converted.

Data Outputs

- **OUT** (Type: `USINT`): The result of the conversion from `INT` to `USINT`.

Adapters

This function block does not have any adapter interfaces.

Adapters ... ## Functionality

When the event input `REQ` is triggered, the algorithm of the function block is executed. This converts the value at the data input `IN` from type `INT` to type `USINT` and stores the result in the data output `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

Technical Details

- The conversion is performed by the function `INT_TO_USINT`, which converts the `INT` value into a `USINT` value. Data loss can occur if the original `INT` value is outside the representable range of `USINT` (0 to 255).
- This function block is part of the `iec61131::conversion` package, which provides various type conversions.

State Overview

The function block has no internal state. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control applications where different data types need to be processed.
- Reducing memory usage by converting to smaller data types where possible.
- Preparing data for communication with systems that can only process `USINT` values.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_DINT_TO_USINT` or `F_SINT_TO_USINT` offer conversions from other integer types to `USINT`. `F_INT_TO_USINT` is specifically optimized for converting `INT` to `USINT`.
- Unlike generic conversion blocks, this function block provides a direct and efficient solution for specific type conversion.

Conclusion

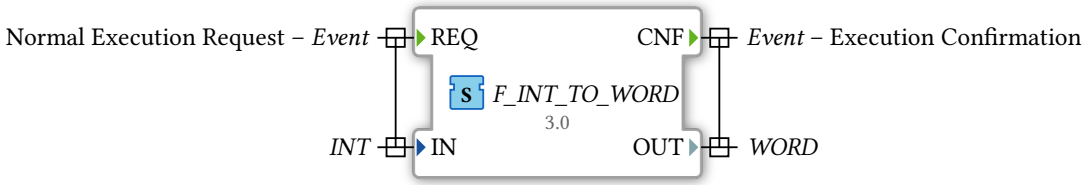
The `F_INT_TO_USINT` function block is a simple and efficient function block for converting `INT` to `USINT` values. It is ideally suited for applications that require precise and specific type conversion, especially in environments with limited resources or specific data type requirements.

F_INT_TO_WORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_INT_TO_WORD` converts an integer value (INT) into a 16-bit word value (WORD). This conversion is particularly useful in scenarios where data needs to be exchanged between systems with different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. Simultaneously, the converted value is output at data output `OUT`.

Data Inputs

- **IN**: The input for the integer value (INT) to be converted.

Data Outputs

- **OUT**: The output for the converted 16-bit word value (WORD).

Adapters

No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The value at input `IN` is read and converted into a WORD value using the function `INT_TO_WORD`. The result is output at `OUT`, and simultaneously the event `CNF` is triggered to signal the successful completion of the operation.

Technical Features

- The function block is implemented simply and efficiently, without additional states or complex logic.
- The conversion occurs directly and without delay.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Data conversion in control systems where different data types need to be processed.
- Interfaces between systems that use different data types.
- Logging and monitoring where data in uniform formats is required.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_INT_TO_WORD` specializes in converting INT to WORD and is therefore more efficient at this specific task.
- Other blocks, such as `F_DINT_TO_WORD` or `F_SINT_TO_WORD`, offer similar functionality but for different source data types.

Conclusion

The `F_INT_TO_WORD` function block is a simple and effective tool for converting integer values to 16-bit word values. Its clear interface and straightforward operation make it a reliable component in control and automation systems.

INT Conversions

🎧 Podcast

- Eclipse 4diac 3.0: ST Interpreter, FBE, and 7200 Commits – The Turbo for Distributed Automation
- From PLC to Distributed Intelligence: Understanding IEC 61499 for Industry 4.0
- Simplifying Industrial Control: Your Deep Dive into Eclipse 4diac and IEC 61499
- Events and IEC 61499: The Starting Signal for Intelligent Systems
- ERP Systems: The Digital Brain – Opportunities, Risks, and the Path to Intelligent Production

🛠️ Related Exercises

- SR_TO_QX_TRIG_EO
- Uebung_039_sub_NumbAnsicht
- Softkey_T_FF_ILOCK_TO_QX_BG
- Bool8_TO_logiBUS_QX8

🌐 Matching topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

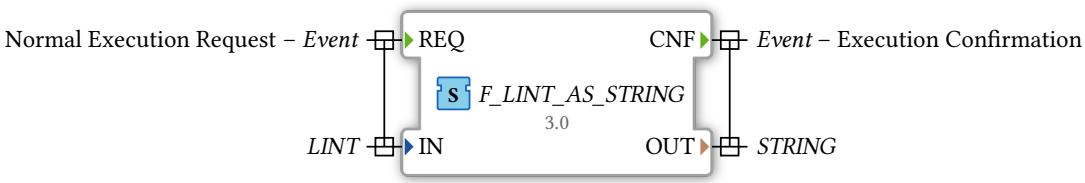
LINT

F_LINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LINT_AS_STRING` converts a LINT data type (64-bit integer) to a STRING data type (character string). This block is particularly useful when numeric values are needed as text for output or further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event is triggered, the value at the data input `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event, along with the converted value, is output at data output `OUT`.

Data Inputs

- **IN** (LINT): The input for the LINT value to be converted.

Data Outputs

- **OUT** (STRING): The output for the converted STRING value.

Adapters

- No adapters are present.

Functionality

When the `REQ` event is triggered, the function block converts the LINT value at input `IN` into a STRING value. The result is output at `OUT` and confirmed by the `CNF` event.

Technical Features

- The function block is a simple function block (SimpleFB) and uses an ST algorithm for conversion.
- The conversion is performed directly by the function `LINT_AS_STRING(IN)`.

State Overview

1. **Initialization**: The function block waits for the `REQ` event.
2. **Conversion**: The LINT value is converted upon `REQ`.
3. **Acknowledgement**: The `CNF` event is triggered, and the converted value is output.

Application Scenarios

- Displaying numeric values in user interfaces.
- Logging numeric data to text files.
- Preparing data for communication via text-based protocols.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks like `F_INT_AS_STRING` or `F_DINT_AS_STRING`, this block supports the larger LINT data type (64-bit).

Conclusion

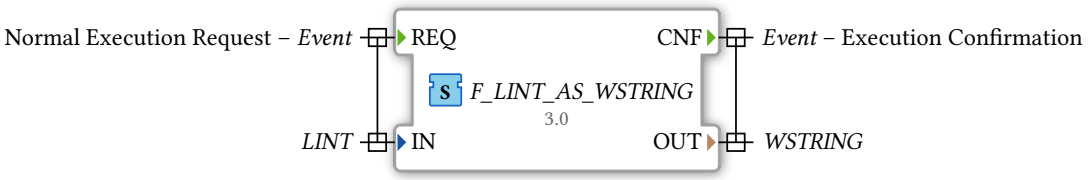
The `F_LINT_AS_STRING` function block offers a simple and efficient way to convert large integers into readable strings. Its clear interface and straightforward operation make it ideal for applications requiring conversion between these data types.

F_LINT_AS_WSTRING

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_LINT_AS_WSTRING` is used to convert a LINT value (64-bit integer) into a WSTRING (wide string). This block is particularly useful in scenarios where numeric values need to be output as strings or processed further.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion of the LINT value to a WSTRING. This input is connected to the data input `IN`.

Event Outputs
- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs
- **IN** (LINT): The LINT value to be converted to a WSTRING.

Data Outputs
- **OUT** (WSTRING): The resulting WSTRING after conversion.

Adapters
No adapters are present.

Functionality
The function block performs the conversion as soon as the `REQ` event occurs. The LINT value from `IN` is converted to a WSTRING and output to `OUT`. The conversion is confirmed by the `CNF` event.

Technical Features
- The function block uses the `LINT_AS_WSTRING` function for conversion.
- The conversion is performed directly without additional parameters.

State Overview
1. **Idle State**: Waits for the `REQ` event.
2. **Conversion State**: Performs the conversion and outputs the result.
3. **Confirmation State**: Sends the `CNF` event to confirm the conversion.

Application Scenarios
- Displaying numeric values in user interfaces.
- Logging numeric data to text files.
- Communicating with systems that expect string data.

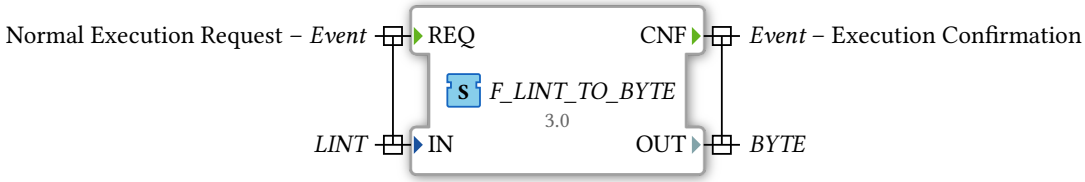
⚖️ Comparison with Similar Function Blocks
- **F_INT_AS_WSTRING**: Converts INT (16-bit) to WSTRING.
- **F_DINT_AS_WSTRING**: Converts DINT (32-bit) to WSTRING.
- **F_LINT_AS_STRING**: Converts LINT to STRING (not Wide String).

Conclusion
The `F_LINT_AS_WSTRING` function block provides a simple and efficient way to convert 64-bit integers to wide strings. Its clear interface and direct functionality make it ideal for applications that require conversion between these data types.

F_LINT_TO_BYTE

• − ◦ • − ◦ • − ◦ •

Introduction
The function block `F_LINT_TO_BYTE` converts a LINT data type (64-bit integer) to a BYTE data type (8-bit integer). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Triggers the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals successful completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: Input for the LINT value to be converted.

Data Outputs

- **OUT**: Output for the converted BYTE value.

Adapters

- No adapters available.

Functionality

This function block performs the conversion from LINT to BYTE as soon as the event `REQ` is triggered. The conversion is carried out using the function `LINT_TO_BYTE(IN)`, where the value from `IN` is converted to the BYTE type and output to `OUT`. After successful conversion, the event `CNF` is triggered.

Technical Features

- The function block is a simple algorithm (SimpleFB) and does not perform any complex state transitions.
- Data loss can occur during conversion if the LINT value is outside the representable range of a BYTE (0 to 255).

State Overview

Since this is a simple function block, there are no state transitions. Execution occurs directly upon triggering `REQ`.

Application Scenarios

- Type conversion in control applications where LINT data needs to be converted to BYTE data.
- Use in data processing chains where different data types need to be made compatible.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_INT_TO_BYTE` or `F_DINT_TO_BYTE` perform similar conversions, but with different output data types.
- `F_LINT_TO_BYTE` is specifically for converting 64-bit to 8-bit integers.

Conclusion

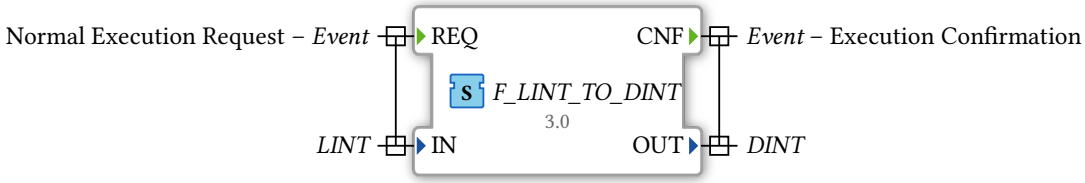
The `F_LINT_TO_BYTE` function block provides a simple and efficient way to convert LINT data to BYTE data. Its use is essential in type conversion scenarios; however, potential data loss with large values should be considered.

F_LINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LINT_TO_DINT` converts a 64-bit integer value (LINT) to a 32-bit integer value (DINT). This conversion is useful when data needs to be exchanged between systems with different word widths or when memory space needs to be optimized.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (LINT): The 64-bit integer value to be converted.

Data Outputs

- **OUT** (DINT): The resulting 32-bit integer value after conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion by calling the algorithm `REQ` as soon as the event `REQ` occurs. The algorithm uses the function `LINT_TO_DINT` to convert the value from `IN` to `OUT`. After successful conversion, the event `CNF` is triggered.

Technical Features

- The function block is part of the package `iec61131::conversion`.
- It is a simple function block without internal states or complex logic.
- The conversion can result in data loss if the original LINT value is outside the representable range of a DINT.

State Overview

Because it is a simple function block, there are no state transitions or internal states. The logic is executed directly upon the arrival of the `REQ` event.

Application Scenarios

- Data compatibility between systems with different word widths.
- Memory optimization when 64-bit values are not required.
- Integration into larger control systems that require DINT values.

⚖️ Comparison with Similar Blocks

- Unlike generic conversion blocks, `F_LINT_TO_DINT` specializes in converting LINT to DINT.
- Other blocks might offer additional features such as range checks or error handling, which is not the case here.

Conclusion

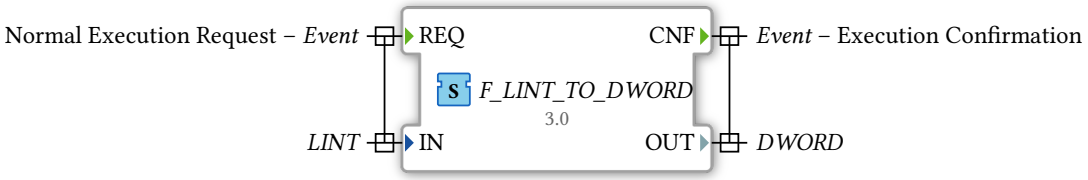
The `F_LINT_TO_DINT` function block provides a simple and efficient way to convert 64-bit integer values to 32-bit integer values. Its simplicity makes it reliable and easy to integrate; however, users should be aware of potential data loss when converting large values.

F_LINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LINT_TO_DWORD` converts a LINT value (64-bit integer) to a DWORD value (32-bit unsigned integer). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `LINT`, containing the value to be converted.

Data Outputs

- **OUT**: The output of type `DWORD`, containing the result of the conversion.

Adapters

- No adapters are available.

Functionality

The function block performs the conversion by transforming the `LINT` value at input `IN` into a `LINT_TO_DWORD` value. The result is output at `OUT`. The conversion is started by the event `REQ` and confirmed by `CNF`.

Technical Features

- The conversion truncates the 32 most significant bits of the `LINT` value, as `DWORD` can only store 32 bits.
- No overflow check is performed, which can lead to data loss with large `LINT` values.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Complete**: Sends the `CNF` event and outputs the result.

Application Scenarios

- Type conversion in control applications where different data types need to be processed.
- Data reduction when only the lower 32 bits of a `LINT` value are relevant.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_INT_TO_DINT` or `F_DINT_TO_LINT` perform similar type conversions, but between different data types.
- `F_LINT_TO_DWORD` is specifically for converting `LINT` to `DWORD`.

Conclusion

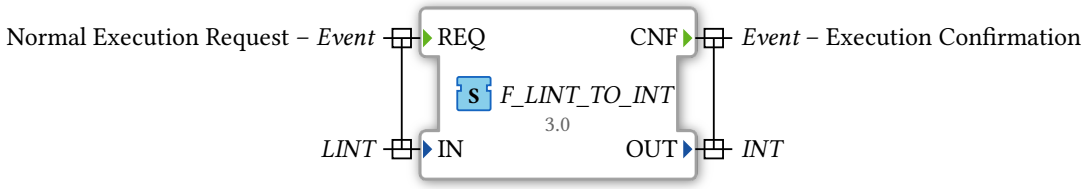
The `F_LINT_TO_DWORD` function block offers a simple and efficient way to convert `LINT` values to `DWORD` values. Its use is recommended in scenarios requiring type conversion between these specific data types, although potential data loss with large values should be considered.

F_LINT_TO_INT

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_LINT_TO_INT` is used to convert a 64-bit integer value (LINT) to a 16-bit integer value (INT). This conversion is useful when data needs to be exchanged or processed between systems with different word lengths.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (LINT): The 64-bit integer value to be converted.

Data Outputs

- **OUT** (INT): The resulting 16-bit integer value after conversion.

Adapters

- No adapters are present.

Functionality

This function block performs the conversion from a LINT to an INT value as soon as the event `REQ` is received. The conversion is carried out by the function `LINT_TO_INT(IN)`, which converts the value of `IN` to an INT value and outputs the result to `OUT`. The event `CNF` is then triggered to signal successful conversion.

Technical Details

- Data loss can occur during conversion if the original LINT value is outside the representable range of an INT value (-32,768 to 32,767).
- The function block has a simple structure and contains only one conversion algorithm.

State Overview

1. **Idle State**: Waits for the event `REQ`.
2. **Conversion State**: Performs the conversion and outputs the result.
3. **Acknowledge State**: Sends the event `CNF` and returns to the idle state.

Application Scenarios

- Data communication between systems with different word widths.
- Reduced memory usage when full 64-bit precision is not required.
- Integration into control systems that can only process 16-bit integers.

⚖️ Comparison with similar function blocks

- **F_LINT_TO_DINT**: Converts LINT to DINT (32-bit integer).
- **F_LINT_TO_UDINT**: Converts LINT to UDINT (32-bit unsigned integer).
- **F_LINT_TO_ULINT**: Converts LINT to ULINT (64-bit unsigned integer).

Conclusion

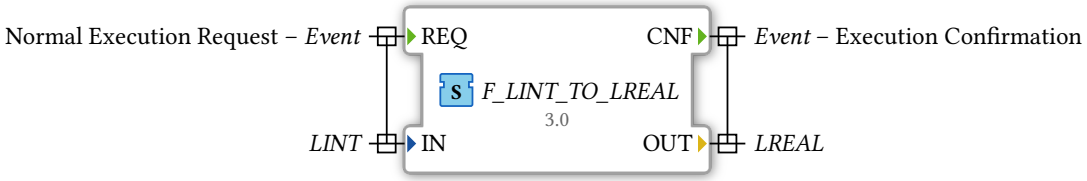
The `F_LINT_TO_INT` function block provides a simple and efficient way to convert 64-bit integer values to 16-bit integer values. It is particularly useful in scenarios where reducing data width is necessary, however, the possibility of data loss during conversion should be considered.

F_LINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LINT_TO_LREAL` converts an integer value of type `LINT` (64-bit integer) to a floating-point value of type `LREAL` (64-bit floating-point). This conversion is particularly necessary when integer values are to be used in calculations with floating-point numbers.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `LINT` to be converted to a `LREAL` value.

Data Outputs

- **OUT**: The converted output value of type `LREAL`.

Adapters

This function block does not use adapters.

Functionality

When the event `REQ` is triggered, the algorithm is executed, converting the value of `IN` of type `LINT` to type `LREAL` and outputting the result to `OUT`. The event `CNF` is then triggered to signal the completion of the conversion.

Technical Features

- The conversion is performed directly without additional parameters.
- The function block is suitable for use in real-time systems because it operates deterministically.

State Overview

The function block has no internal states. The conversion occurs immediately upon the triggering of `REQ`.

Application Scenarios

- Conversion of large integers to floating-point numbers for mathematical calculations.
- Integration into control systems that need to process mixed data types.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks such as `F_INT_TO_REAL` or `F_DINT_TO_REAL`, `F_LINT_TO_LREAL` handles larger number ranges (64-bit).

Conclusion

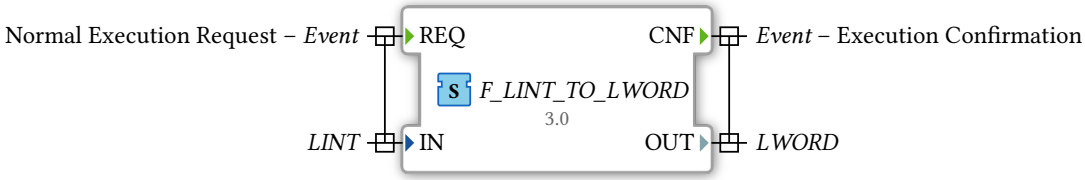
The `F_LINT_TO_LREAL` function block provides an efficient and reliable method for converting 64-bit integers to 64-bit floating-point numbers. Its simple interface and deterministic operation make it ideal for use in industrial control systems.

F_LINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LINT_TO_LWORD` converts a LINT data type (64-bit signed integer) to an LWORD data type (64-bit bit string). This block is particularly useful in scenarios where numeric values need to be converted to their binary representation.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event input is triggered, the function block reads the value at the data input `IN` and performs the conversion.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. This event, along with the converted value, is output at the data output `OUT`.

Data Inputs

- **IN**: Expects a value of type `LINT` (64-bit signed integer) to be converted.

Data Outputs

- **OUT**: Outputs the converted value of type `LWORD` (64-bit bit string).

Adapters

This function block has no adapter interfaces.

Functionality

When the `REQ` event is triggered, the value at input `IN` is read and converted into a `LWORD` value using the function `LINT_TO_LWORD`. The result is provided at output `OUT`, and simultaneously, the `CNF` event is triggered.

Technical Features

- The conversion occurs directly and without any additional delay.
- The function block is deterministic and has no internal states.

State Overview

Since this is a simple function block, there are no state transitions or state machines. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of numeric values into their binary representation for bitwise operations.
- Use in protocol implementations where numeric values must be transmitted as bit strings.
- Integration into systems that perform low-level bit manipulation.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_LINT_TO_LWORD` specializes in converting `LINT` to `LWORD` and therefore offers an optimized and type-safe solution.
- Similar blocks such as `INT_TO_WORD` or `DINT_TO_DWORD` work with smaller data types.

Conclusion

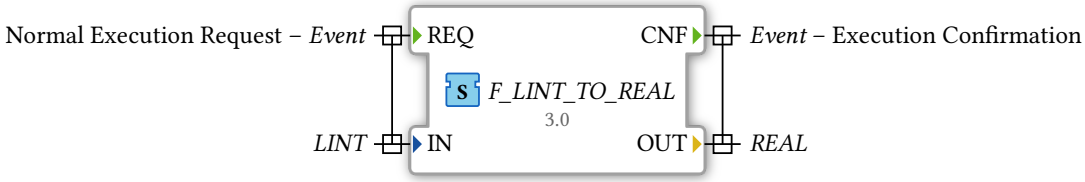
The `F_LINT_TO_LWORD` function block provides an efficient and reliable method for converting 64-bit integers to 64-bit bit strings. Its simplicity and determinism make it ideal for real-time applications where fast and precise type conversions are required.

F_LINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LINT_TO_REAL` converts an integer value of type `LINT` (64-bit integer) to a floating-point value of type `REAL`. This conversion is useful when data needs to be transformed from an integer format to a floating-point format, for example, for further calculations or display.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event, along with the converted value, is output at data output `OUT`.

Data Inputs

- **IN**: The input of type `LINT`, which provides the integer value to be converted.

Data Outputs

- **OUT**: The output of type `REAL`, which provides the result of the conversion.

Adapters

- This function block does not use any adapters.

Operation

The function block performs the conversion as soon as the event `REQ` is received. The algorithm reads the value of `IN`, converts it using the function `LINT_TO_REAL`, and writes the result to `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

Technical Features

- The conversion is performed without rounding or scaling, as it is a direct type conversion.
- The function block is suitable for use in real-time systems because it does not involve any delays or complex calculations.

State Overview

The function block has no internal state. The operation is re-executed with each `REQ` event.

Application Scenarios

- Conversion of sensor values stored as `LINT` to `REAL` for further processing.

- Integration into systems that require floating-point operations but receive input data as integers.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_INT_TO_REAL` or `F_DINT_TO_REAL` perform similar conversions, but with smaller integer types (`INT` or `DINT`).
- `F_LINT_TO_REAL` is specifically optimized for 64-bit integers (`LINT`).

Conclusion

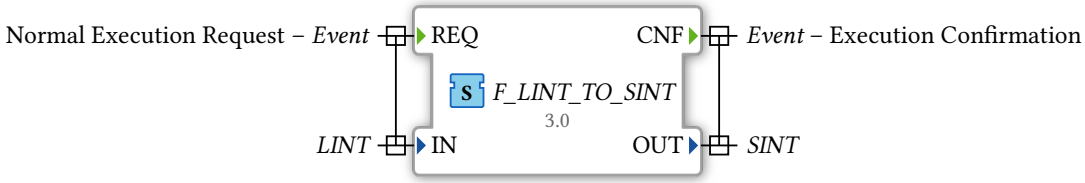
The `F_LINT_TO_REAL` function block offers a simple and efficient way to convert 64-bit integer values to floating-point numbers. Its clear interface and direct functionality make it ideal for applications requiring type conversion without additional processing.

F_LINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LINT_TO_SINT` converts a 64-bit integer value (`LINT`) into an 8-bit integer value (`SINT`). This conversion is useful when data needs to be exchanged or processed between systems with different word lengths.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is connected to data output `OUT`.

Data Inputs

- **IN** (`LINT`): The 64-bit integer value to be converted.

Data Outputs

- **OUT** (`SINT`): The resulting 8-bit integer value after conversion.

Adapters

No adapters are available.

Functionality

When the event `REQ` is triggered, the algorithm is executed, converting the value of `IN` (of type `LINT`) to a `SINT` value and outputting the result to `OUT`. The event `CNF` is then triggered to signal the completion of the conversion.

Technical Features

- Data loss can occur if the original `LINT` value is outside the representable range of `SINT` (-128 to 127).
- The function block is simple and deterministic, without internal states or complex logic.

State Overview

This function block has no internal states. Execution occurs solely in response to the `REQ` event.

Application Scenarios

- Data reduction during transmission between systems with different word widths.
- Processing of sensor data where only the lower 8 bits are relevant.

- Compatibility with older systems that can only process 8-bit data.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_LINT_TO_SINT` is specialized and optimized for converting `LINT` to `SINT`.
- Other conversion blocks might offer additional features such as overflow handling or scaling, which is not the case here.

Conclusion

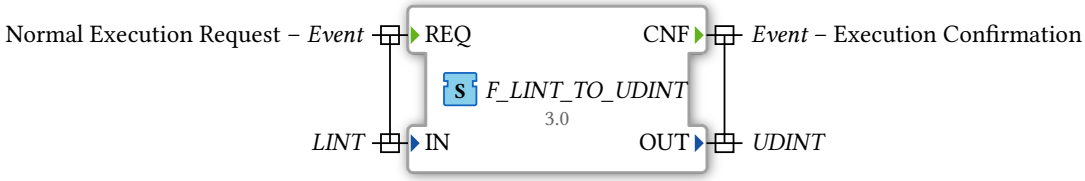
The `F_LINT_TO_SINT` function block is a simple and efficient tool for converting 64-bit to 8-bit integers. Its simplicity makes it robust and easy to understand, although it lacks advanced features such as overflow control. It is ideal for applications requiring direct and fast conversion.

F_LINT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LINT_TO_UDINT` converts a 64-bit signed integer value (`LINT`) to a 32-bit unsigned integer value (`UDINT`). This block is part of the `iec61131::conversion` library and is suitable for applications requiring type conversion.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (`LINT`): The 64-bit signed integer value to be converted.

Data Outputs

- **OUT** (`UDINT`): The resulting 32-bit unsigned integer value after the conversion.

Adapters

No adapters available.

Functionality

When the event `REQ` is triggered, the algorithm is executed, converting the value of `IN` from `LINT` to `UDINT` and outputting the result to `OUT`. The event `CNF` is then triggered to signal the completion of the operation.

Technical Features

- The block uses the built-in function `LINT_TO_UDINT` for the conversion.
- It is a simple function block without state machines or complex logic.

State Overview

The block has no internal states. The conversion occurs immediately upon triggering `REQ`.

Application Scenarios

- Type conversion in control applications where 64-bit values need to be converted to 32-bit values.
- Use in data processing chains where different data types need to be made compatible.

⚖️ Comparison with Similar Blocks

- Similar blocks like `F_DINT_TO_UDINT` or `F_INT_TO_UINT` offer conversions between other integer types, but not for `LINT` to `UDINT`.
- Unlike generic conversion blocks, `F_LINT_TO_UDINT` is specialized and optimized for this specific conversion.

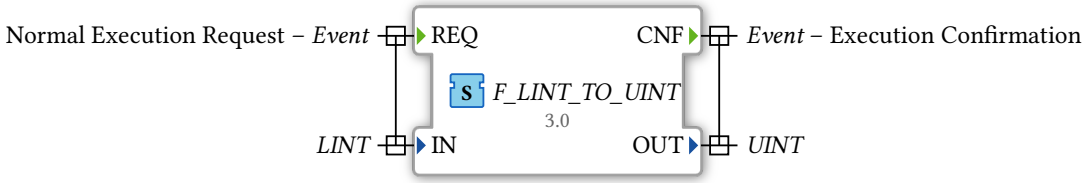
Conclusion

The `F_LINT_TO_UDINT` block is an efficient tool for converting 64-bit signed integer values to 32-bit unsigned integer values. Its simplicity and direct operation make it a reliable component in control applications.

F_LINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

The function block `F_LINT_TO_UINT` is used to convert a 64-bit integer value (`LINT`) into a 16-bit unsigned integer value (`UINT`). This block is particularly useful in scenarios where type conversion between numeric data types of different sizes is required.



- **REQ**: Starts the conversion. This input is linked to the data input `IN`.
- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.
- **IN**: This input expects a value of type `LINT` (64-bit integer) to be converted.
- **OUT**: This output returns the converted value of type `UINT` (16-bit unsigned integer).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters are defined.

The function block performs the conversion from `LINT` to `UINT` as soon as the event `REQ` is triggered. The algorithm `REQ` uses the function `LINT_TO_UINT` to convert the input value. The result is output via `OUT`, and the event `CNF` signals successful execution.

- **Type Conversion**: It is important to note that data loss can occur when converting from a larger to a smaller data type if the original value is outside the representable range of `UINT`.

The function block has no internal states. The conversion is re-executed with each `REQ` event.

- **Data Processing**: In control systems where different data types need to be harmonized.
- **Interface Compatibility**: For communication between systems that use different data types.
- **F_LINT_TO_INT**: Converts `LINT` to `INT`, while preserving the sign.
- **F_LINT_TO_ULINT**: Converts `LINT` to `ULINT` without reducing the bit width.

The `F_LINT_TO_UINT` function block is a simple yet effective tool for type conversion in IEC 61499-based systems. Its clear interface and direct operation make it a reliable solution for conversion tasks. However, be aware of potential data loss during the conversion process.

Functionality

Application Scenarios

State Overview

Technical Features

⚖️ Vergleich mit ähnlichen Bausteinen

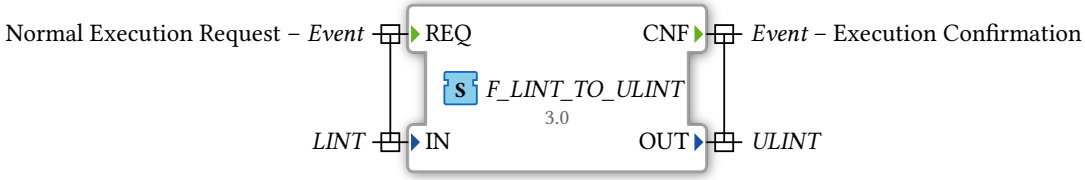
Conclusion

F_LINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LINT_TO_ULINT` converts a signed 64-bit integer value (`LINT`) to an unsigned 64-bit integer value (`ULINT`). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: This input triggers the conversion. It is linked to the data input `IN`.

Event Outputs

- **CNF**: This output signals the completion of the conversion. It is linked to the data output `OUT`.

Data Inputs

- **IN** (`LINT`): The signed 64-bit integer value to be converted.

Data Outputs

- **OUT** (`ULINT`): The unsigned 64-bit integer value output as the result of the conversion.

Adapters

This function block has no adapters.

Functionality

The function block performs the conversion from `LINT` to `ULINT` as soon as the event `REQ` is triggered. The conversion is carried out using the function `LINT_TO_ULINT(IN)`, which converts the value of `IN` into the corresponding unsigned type. After successful conversion, the event `CNF` is triggered, and the converted value is available at output `OUT`.

Technical Features

- The function block is a simple block without internal state management.
- The conversion occurs directly and without delay.

State Overview

Since it is a simple function block, there are no state transitions or complex state logic.

Application Scenarios

- Type conversion in control applications where signed and unsigned values need to be processed.

- Data processing in industrial automation systems where different data types need to be made compatible.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks could be `F_INT_TO_UINT` or `F_DINT_TO_UDINT`, but these are designed for smaller integer types.
- Unlike generic conversion function blocks, `F_LINT_TO_ULINT` is specialized for the 64-bit variants of the data types.

Conclusion

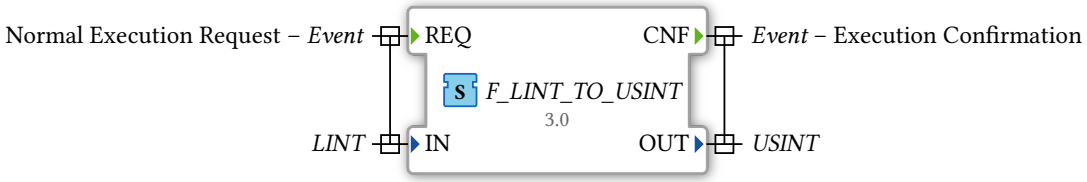
The `F_LINT_TO_ULINT` function block provides a simple and efficient way to convert signed 64-bit integer values to unsigned values. Its clear interface structure and direct functionality make it a reliable tool in industrial automation.

F_LINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LINT_TO_USINT` converts a 64-bit integer value (`LINT`) into an 8-bit unsigned integer value (`USINT`). This block is part of the IEC 61131 conversion library and is typically used in automation applications where data types of different sizes need to be processed.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (`LINT`): The 64-bit integer value to be converted.

Data Outputs

- **OUT** (`USINT`): The resulting 8-bit unsigned integer value after the conversion.

Adapters

No adapters are present.

Operation

The function block performs the conversion when the event `REQ` is triggered. The value of `IN` of type `LINT` is converted to type `USINT`, and the result is output to `OUT`. The conversion is confirmed by the event `CNF`.

Technical Features

- The conversion truncates higher-order bits if the value of `LINT` is outside the representable range of `USINT` (0 to 255).
- The block is deterministic and has no internal state.

State Overview

The function block has no internal state. Execution occurs immediately after receiving the `REQ` event.

Application Scenarios

- Conversion of sensor values or counters stored as `LINT` for processing in systems that only support `USINT`.
- Use in control systems where memory optimization is crucial.

⚖️ Comparison with Similar Blocks

- Similar conversion blocks like `F_INT_TO_USINT` or `F_DINT_TO_USINT` operate with smaller input data types (`INT` and `DINT`, respectively).
- Unlike generic conversion blocks, `F_LINT_TO_USINT` specializes in converting `LINT` to `USINT`.

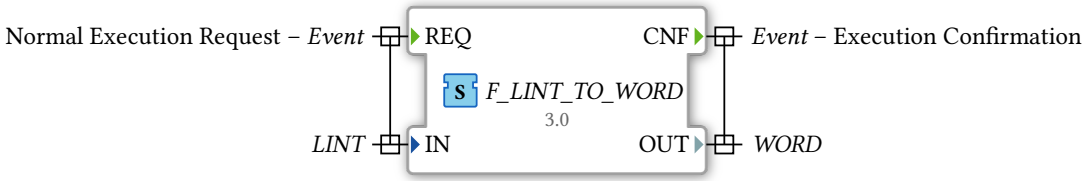
Conclusion

The `F_LINT_TO_USINT` function block is a specialized and efficient tool for converting 64-bit integers to 8-bit unsigned values. Its simple and deterministic operation makes it reliable in automation applications.

F_LINT_TO_WORD

• – ◦ • – ◦ • – ◦ •

The function block `F_LINT_TO_WORD` converts a LINT data type (64-bit integer) to a WORD data type (16-bit word count). This conversion is useful when data needs to be exchanged between systems with different word widths or when an explicit type conversion is required.



- `REQ` (Normal Execution Request): Starts the execution of the conversion. Linked to the data input `IN`.
- `CNF` (Execution Confirmation): Signals successful completion of the conversion. Linked to the data output `OUT`.
- `IN` (LINT): The input value to be converted from LINT to WORD.
- `OUT` (WORD): The converted output value in WORD format.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use adapters.

Functionality

The function block performs the conversion from LINT to WORD when the event `REQ` is triggered. The conversion is performed using the built-in function `LINT_TO_WORD`, which takes the value of the input `IN` and converts it to the WORD data type. The result is output at `OUT`, and the event `CNF` signals successful execution.

- The conversion truncates the higher-order bits of the LINT value because WORD can only store 16 bits.
- There is no overflow or underflow check during the conversion.

The function block is stateless and re-executes the conversion with each `REQ` event.

- Data communication between systems with different word lengths.
- Similar function blocks such as `F_DINT_TO_WORD` or `F_INT_TO_WORD` perform similar conversions, but with different output data types.

- `F_LINT_TO_WORD` is specific to converting 64-bit to 16-bit and does not offer additional functionality such as overflow checking.

The `F_LINT_TO_WORD` function block is a simple yet effective tool for converting LINT to WORD data. It is particularly useful in scenarios where data width reduction is required without the need for additional validation.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

LINT Conversions

Here you will find functions for converting the LINT data type.

🛠️ Related Exercises

- Exercise_072c

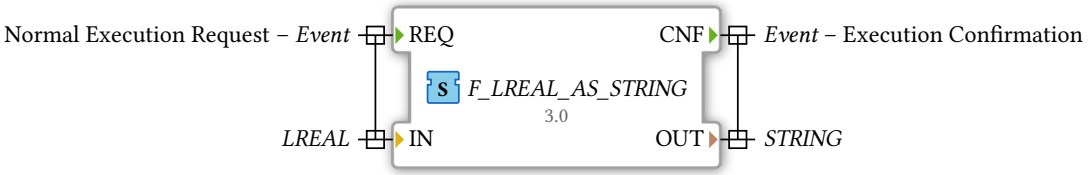
LREAL

F_LREAL_AS_STRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_AS_STRING` is used to convert an LREAL value (64-bit floating-point number) into a STRING. This functionality is particularly useful when numeric values are needed as strings for display, logging, or communication.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs

- `IN` (LREAL): The input value to be converted to a STRING.

Data Outputs

- `OUT` (STRING): The resulting STRING after conversion.

Adapters

This function block does not use any adapters.

Operation

When the `REQ` event is triggered, the algorithm is executed, which converts the LREAL value `IN` to a STRING and outputs the result via `OUT`. The `CNF` event signals the successful completion of the conversion.

Technical Features

- The conversion is performed using the function `LREAL_AS_STRING`.
- The function block is simple and has no internal states or complex logic.

State Overview

Since it is a simple function block, there are no state transitions or machines.

Application Scenarios

- Displaying floating-point numbers in user interfaces.
- Logging measured values to text files or databases.
- Preparing data for communication via text protocols.

⚖️ Comparison with Similar Function Blocks

- Unlike `F_STRING_TO_LREAL`, this function block performs the reverse conversion.
- Compared to generic conversion function blocks, this one specializes in LREAL-to-STRING conversions.

Conclusion

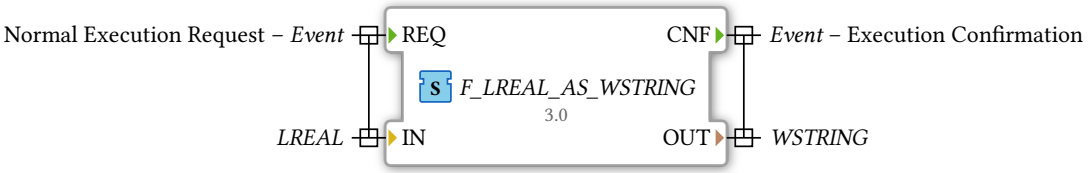
The `F_LREAL_AS_STRING` function block offers a simple and efficient way to convert floating-point numbers into strings. Its clear interface and straightforward functionality make it a useful component in many automation applications.

F_LREAL_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_AS_WSTRING` converts an LREAL value (64-bit floating-point number) into a WSTRING (wide string). This is particularly useful when numeric values are needed for display or further processing as text.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion of the input value. The input value `IN` is associated with this event.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output value `OUT` is output with this event.

Data Inputs

- **IN** (LREAL): The input value to be converted to a WSTRING.

Data Outputs

- **OUT** (WSTRING): The result string after conversion.

Adapters

This function block does not use any adapters.

Functionality

When the `REQ` event is triggered, the input value `IN` of type LREAL is converted to a WSTRING using the function `LREAL_AS_WSTRING` and made available at output `OUT`. The `CNF` event signals successful conversion.

Technical Features

- The conversion is performed directly without additional parameters.
- This function block is suitable for use in real-time systems.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Converts the input value.
3. **Done**: Sends the `CNF` event with the converted value.

Application Scenarios

- Displaying floating-point values on user interfaces.
- Logging measured values to text files.
- Preparing data for communication with systems that expect string data.

⚖️ Comparison with Similar Function Blocks

- Compared to `F_REAL_AS_WSTRING`, this function block processes 64-bit floating-point numbers instead of 32-bit.
- Similar to `F_DINT_AS_WSTRING`, but for floating-point numbers instead of integers.

Conclusion

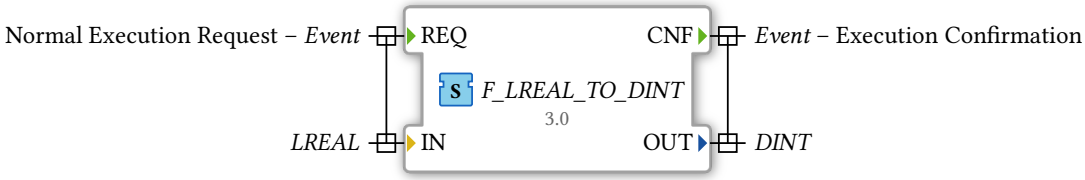
The `F_LREAL_AS_WSTRING` function block offers a simple and efficient way to convert LREAL values to WSTRINGs. Its clear interface and direct functionality make it a useful tool in various use cases.

F_LREAL_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_TO_DINT` converts an LREAL value (64-bit floating-point number) to a DINT value (32-bit integer). This block is part of the `iec61131::conversion` package and implements a simple type conversion.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion process. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (LREAL): The input value to be converted from LREAL to DINT.

Data Outputs

- **OUT** (DINT): The result of the conversion as a DINT value.

Adapters

This function block does not have any adapters.

Operation

When the `REQ` event is triggered, the algorithm is executed that converts the `IN` value from LREAL to DINT and outputs the result to `OUT`. The `CNF` event is then triggered to signal the successful completion of the operation.

Technical Features

- The conversion is performed by the built-in function `LREAL_TO_DINT`.
- This is a simple function block without state management or complex logic.

State Overview

The function block has no internal states. Execution is purely event-driven and occurs with each `REQ` event.

Application Scenarios

- Type conversion in control applications where floating-point numbers need to be converted to integers.
- Use in calculations where an integer representation is required.

⚖️ Comparison with Similar Blocks

- Compared to other conversion blocks such as `F_REAL_TO_INT` or `F_LREAL_TO_LINT`, this block offers a specific conversion from LREAL to DINT.

Similar blocks can have different source or target types, such as `F_DINT_TO_LREAL` for reverse conversion.

Conclusion

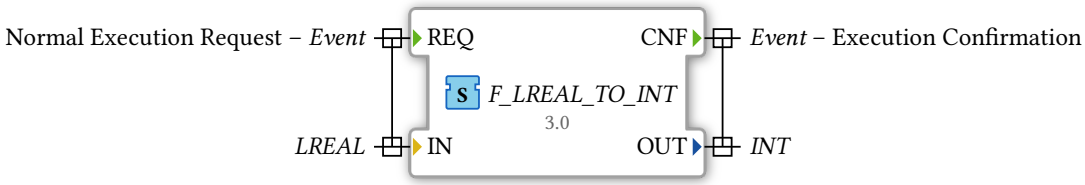
The `F_LREAL_TO_DINT` function block is a simple and efficient tool for type conversion in IEC 61499-based control systems. Its clear interface and direct operation make it ideal for applications requiring fast conversion of floating-point numbers to integers.

F_LREAL_TO_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_TO_INT` converts an LREAL value (64-bit floating-point number) into an INT value (16-bit integer). This conversion is performed by rounding the LREAL value down to the nearest integer value.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion of the input value. The input value is passed with this event.

Event Outputs

- `CNF`: Signals the completion of the conversion and outputs the converted value.

Data Inputs

- `IN` (LREAL): The input value to be converted to an INT value.

Data Outputs

- `OUT` (INT): The converted integer value.

Adapters

This function block does not use any adapters.

Functionality

Upon receiving the `REQ` event, the algorithm that converts the `IN` value from LREAL to INT is executed. The conversion is performed by the function `LREAL_TO_INT`, which rounds the LREAL value down to the nearest integer value. The result is output to `OUT`, and the `CNF` event is triggered.

Technical Features

- The conversion truncates decimal places (not rounding, but rounding down).
- Overflow problems may occur with values outside the integer range.

State Overview

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Conversion of sensor values (e.g., temperature measurements) for processing in integer-based systems.
- Preparation of floating-point values for output on integer displays or interfaces.

⚖️ Comparison with Similar Function Blocks

- `F_REAL_TO_INT`: Converts REAL (32-bit floating-point number) to INT.
- `F_LREAL_TO_DINT`: Converts LREAL to DINT (32-bit integer) for a wider range of values.

Conclusion

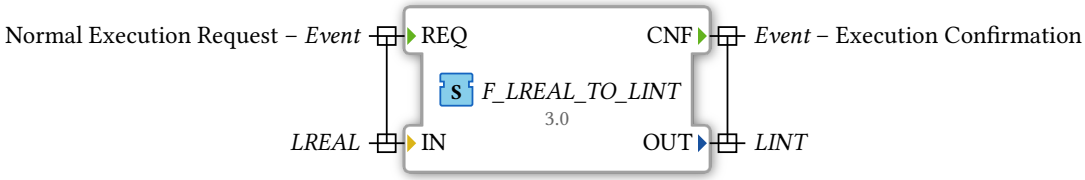
The `F_LREAL_TO_INT` function block is a simple yet effective tool for converting floating-point numbers to integers. It is particularly well-suited for applications requiring fast and direct conversion without additional logic. However, for values exceeding the INT range, alternative function blocks such as `F_LREAL_TO_DINT` should be considered.

F_LREAL_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_TO_LINT` is used to convert an LREAL value (64-bit floating-point number) to a LINT value (64-bit integer). This conversion is particularly necessary in scenarios where floating-point numbers need to be converted to integers, for example, when processing sensor values or controlling actuators.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This input triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. This output signals the completion of the conversion.

Data Inputs

- **IN**: LREAL (64-bit floating-point number). The input value to be converted.

Data Outputs

- **OUT**: LINT (64-bit integer). The converted output value.

Adapters

No adapters available.

Functionality

This function block performs the conversion from LREAL to LINT as soon as the event `REQ` is received. The conversion is carried out using the function `LREAL_TO_LINT`, which converts the input value `IN` to the output value `OUT`. After the conversion is complete, the event `CNF` is triggered.

Technical Features

- The conversion truncates the decimal part of the floating-point number.
- The value is not rounded.
- Values outside the representable range of LINT may result in overflow problems.

State Overview

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Conversion of sensor values (e.g., temperature measurements) into integers for further processing.
- Control of actuators that expect integer values.
- Data preprocessing in control systems.

⚖️ Comparison with Similar Function Blocks

- `F_REAL_TO_INT`: Converts REAL (32-bit floating-point number) to INT (16-bit integer).
- `F_LREAL_TO_DINT`: Converts LREAL (64-bit floating-point number) to DINT (32-bit integer).
- `F_LREAL_TO_LINT` offers higher accuracy through the use of 64-bit integers.

Conclusion

The function block `F_LREAL_TO_LINT` is a simple and efficient building block for converting 64-bit floating-point numbers to 64-bit integers. It is particularly suitable for applications requiring high accuracy and a large value range. Its simple interface and direct operation make it easy to integrate into existing control systems.

F_LREAL_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_TO_LWORD` converts an LREAL value (64-bit floating-point number) to an LWORD value (64-bit unsigned integer). This conversion is particularly necessary in scenarios where floating-point numbers need to be converted to a binary representation.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at data output `OUT`.

Data Inputs

- **IN** (LREAL): The input value to be converted.

Data Outputs

- **OUT** (LWORD): The converted output value.

Adapters

This function block does not use any adapters.

Functionality

The function block performs the conversion by transforming the LREAL value at input `IN` into an LWORD value. The conversion is performed directly without additional parameters. When event `REQ` is triggered, the algorithm that converts the value and outputs event `CNF` is executed.

Technical Features

- The conversion is performed using the function `LREAL_TO_LWORD`, which is implemented in the function block's algorithm.
- The function block is simple and has no internal states or complex logic.

State Overview

Since it is a simple function block, there are no state transitions or internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of floating-point numbers into a binary representation for further processing in other systems.
- Use in control systems where floating-point values need to be converted into a format understandable to the hardware.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_REAL_TO_WORD` or `F_DINT_TO_DWORD` perform similar conversions, but with different data types.
- `F_LREAL_TO_LWORD` is specifically for converting 64-bit floating-point numbers to 64-bit unsigned integers.

Conclusion

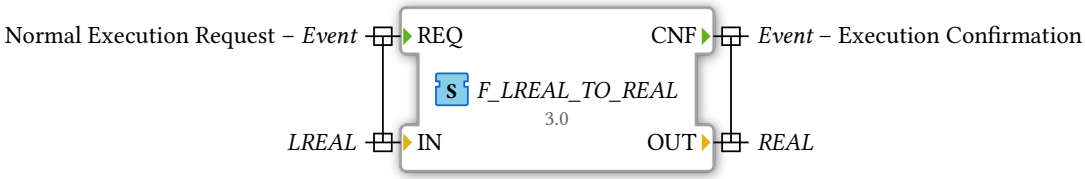
The `F_LREAL_TO_LWORD` function block provides a simple and efficient way to convert floating-point numbers to a binary representation. Its clear interface and direct functionality make it ideal for applications requiring such conversions.

F_LREAL_TO_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_TO_REAL` converts an LREAL value (64-bit floating-point number) to a REAL value (32-bit floating-point number). This block is particularly useful in scenarios where type conversion between these two numeric data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The LREAL input value to be converted.

Data Outputs

- **OUT**: The REAL output value containing the conversion result.

Adapters

This function block has no adapters.

Operation

When the event input `REQ` is triggered, the algorithm is executed, converting the LREAL value of `IN` to a REAL value and outputting the result to `OUT`. The event `CNF` is then triggered to signal the completion of the operation.

Technical Features

- The function block performs a direct type conversion, taking into account potential accuracy losses due to the different bit widths of the data types.
- The conversion is performed using the built-in function `LREAL_TO_REAL`.

State Overview

The function block is stateless and performs the conversion on every `REQ` event.

Application Scenarios

- Type conversion in control applications where LREAL values from high-precision calculations need to be converted into REAL values for further processing.
- Compatibility with systems or function blocks that can only process REAL data types.

⚖️ Comparison with Similar Function Blocks

- Compared to generic conversion blocks, `F_LREAL_TO_REAL` specializes in converting between LREAL and REAL, thus offering a clearly defined and efficient solution for this specific use case.

Conclusion

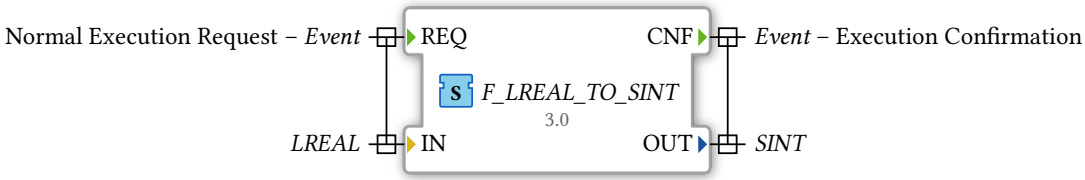
The `F_LREAL_TO_REAL` function block is a simple yet effective building block for converting between LREAL and REAL data types. Its clear interface and direct functionality make it a reliable component in control applications where such type conversions are required.

F_LREAL_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_TO_SINT` converts a `LREAL` value (64-bit floating-point number) to a `SINT` value (8-bit integer count). This conversion is useful when floating-point numbers need to be converted into a compact integer representation, for example, for storage or transmission in systems with limited bandwidth.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (`LREAL`): The input value to be converted.

Data Outputs

- **OUT** (`SINT`): The converted output value.

Adapters

No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is received. The `LREAL` value at input `IN` is converted into a `SINT` value and output as `OUT`. The `CNF` event signals the completion of the conversion.

Technical Features

- The conversion is performed by the function `LREAL_TO_SINT`, which converts the `LREAL` value into a `SINT` value.
- This is a simple function block without state management or complex logic.

State Overview

The function block has no internal states. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Conversion of floating-point numbers to integers for storage in limited memory areas.
- Use in control systems where floating-point numbers need to be converted into compact integer formats.
- Integration into data processing chains where data width reduction is required.

⚖️ Comparison with Similar Function Blocks

- **F_REAL_TO_SINT**: Converts `REAL` (32-bit floating-point number) to `SINT`. Compared to `F_LREAL_TO_SINT`, this method processes input data with lower precision.
- **F_LREAL_TO_INT**: Converts `LREAL` to `INT` (16-bit integer). Offers a wider output width compared to `F_LREAL_TO_SINT`.

Conclusion

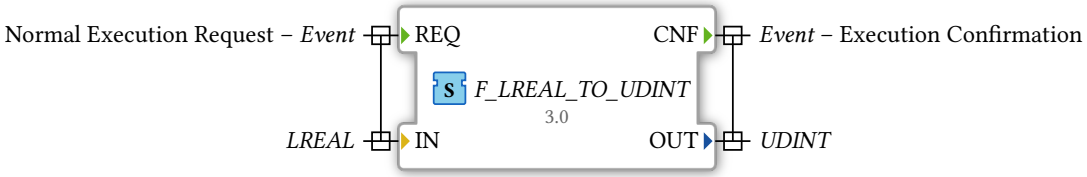
The `F_LREAL_TO_SINT` function block is a simple and efficient tool for converting 64-bit floating-point numbers to 8-bit integers. Its simple interface and direct operation make it ideal for applications where space or bandwidth limitations are a concern.

F_LREAL_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_TO_UDINT` is used to convert a 64-bit floating-point value (LREAL) into a 32-bit unsigned integer value (UDINT). This conversion is useful when floating-point numbers need to be converted into unsigned integers, for example, for specific control or communication applications.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN** (LREAL): The input value to be converted.

Data Outputs

- **OUT** (UDINT): The converted output value.

Adapters

This function block does not use any adapters.

Functionality

The function block performs the conversion from an LREAL value to a UDINT value as soon as the `REQ` event is received. The conversion is performed using the built-in function `LREAL_TO_UDINT`. After the conversion is complete, the `CNF` event is triggered, and the converted value is available at the output `OUT`.

Technical Features

- The conversion truncates the floating-point value to the nearest integer.
- There is no overflow or underflow check. Values outside the UDINT range (0 to 4,294,967,295) result in undefined behavior.

State Overview

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Conversion of sensor data (e.g., temperature measurements) into unsigned integer values for further processing.
- Preparation of floating-point values for communication with systems that only support unsigned integers.

⚖️ Comparison with similar components

- **F_REAL_TO_UINT**: Converts 32-bit floating-point values (REAL) to 16-bit unsigned integer values (UINT).
- **F_LREAL_TO_DINT**: Converts 64-bit floating-point values (LREAL) to 32-bit signed integer values (DINT).
- **F_LREAL_TO_UDINT** is specific for converting to unsigned 32-bit integers.

🛠️ Related Exercises

- Exercise_152
- Exercise_153

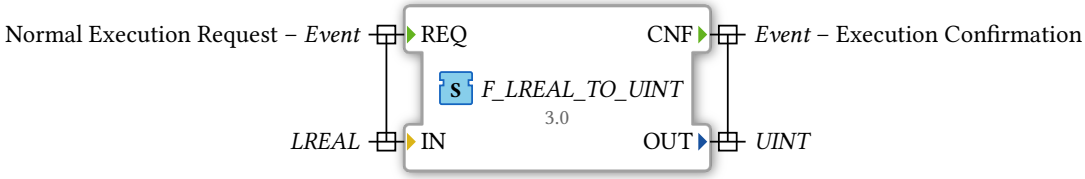
Conclusion

The function block `F_LREAL_TO_UDINT` offers a simple and efficient way to convert floating-point values into unsigned integers. However, due to the lack of overflow checking, it should only be used in scenarios where the input values are known and within the valid range.

F_LREAL_TO_UINT

• – ◦ • – ◦ • – ◦ •

The function block `F_LREAL_TO_UINT` converts an LREAL value (64-bit floating-point number) to a UINT value (unsigned 16-bit integer). This block is part of the `iec61131::conversion` library and enables simple and efficient type conversion in automation applications.



- **REQ**: Starts the conversion. When this event occurs, the algorithm for converting the input value is executed.
- **CNF**: Signals the completion of the conversion. This event is output after successful conversion of the value.

- **IN**: The input expects an LREAL number (64-bit floating-point number) to be converted into a UINT value.
- **OUT**: Outputs the result of the conversion as a UINT value (unsigned 16-bit integer).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block has no adapters.

When the `REQ` event occurs, the algorithm for converting the LREAL value at input `IN` into a UINT value is executed. The result is output at `OUT`, and the `CNF` event signals the successful completion of the operation.

- **Data Types**: The function block processes 64-bit floating-point numbers (LREAL) and outputs 16-bit unsigned integers (UINT).
- **Algorithm**: The conversion is performed directly by the built-in `LREAL_TO_UINT` function.
- **Error Handling**: The function block does not have built-in error handling for overflow or invalid input values.

1. **Wait State**: The function block waits for the `REQ` event.
2. **Execution State**: The conversion is performed at `REQ`.
3. **Confirmation State**: After successful conversion, `CNF` is output, and the function block returns to the wait state.

- Conversion of sensor values (e.g., temperature measurements) into integer values for further processing.
- Type conversion in control applications where floating-point numbers are required to integers for specific operations.
- Interface adaptation between system components with different data types.
- Compared to generic conversion blocks, `F_LREAL_TO_UINT` is specialized for converting LREAL to UINT.
- Other conversion blocks might offer additional functions such as range checking or error handling, but are more complex to use.

The `F_LREAL_TO_UINT` function block offers a simple and efficient solution for the specific conversion of 64-bit floating-point numbers to 16-bit unsigned integers. Its simple interface and direct operation make it ideal for applications where fast type conversion without additional functions is required. However, for applications with critical values, additional checks should be implemented.

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

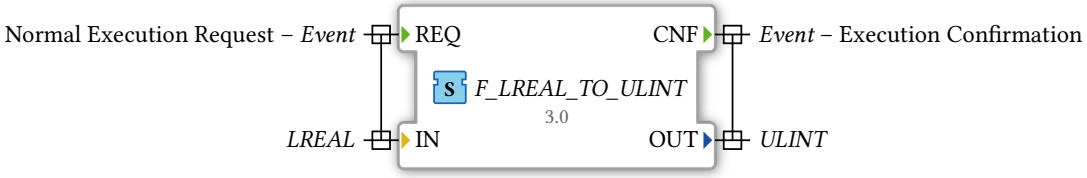
F_LREAL_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_TO_ULINT` converts an LREAL value (64-bit floating-point number) to a ULINT value (64-bit unsigned integer). This conversion is particularly necessary in scenarios where

floating-point numbers need to be converted to unsigned integers, for example, for certain calculations or for storage in specific data formats.



Interface Structure

Event Inputs

- `REQ` (Normal Execution Request): Starts the conversion of the input value. It is linked to the data input `IN`.

Event Outputs

- `CNF` (Execution Confirmation): Signals the completion of the conversion. Linked to the data output `OUT`.

Data Inputs

- `IN` (LREAL): The input value to be converted.

Data Outputs

- `OUT` (ULINT): The converted output value.

Adapters

No adapters available.

Functionality

The function block performs the conversion as soon as the event `REQ` is received. The algorithm `REQ` converts the LREAL value `IN` into a ULINT value using the function `LREAL_TO_ULINT` and stores the result in `OUT`. The event `CNF` is then triggered to signal the completion of the conversion.

Technical Features

- The conversion is performed without rounding; that is, the decimal portion is truncated.
- Values outside the representable range of ULINT (0 to 2^64-1) may result in overflow or unexpected results.

State Overview

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Conversion of sensor values (e.g., temperature measurements) into unsigned integers for further processing.
- Conversion of floating-point numbers for storage in databases or protocols that only support integers.
- Use in control algorithms that require unsigned integers.

⚖️ Comparison with Similar Components

- `F_REAL_TO_UINT`: Converts REAL (32-bit floating-point number) to UINT (16-bit unsigned integer).
- `F_LREAL_TO_LINT`: Converts LREAL to LINT (64-bit signed integer).
- `F_LREAL_TO_ULINT` is specific to 64-bit unsigned conversions and therefore offers higher accuracy and a wider range of values compared to 16- or 32-bit versions.

Conclusion

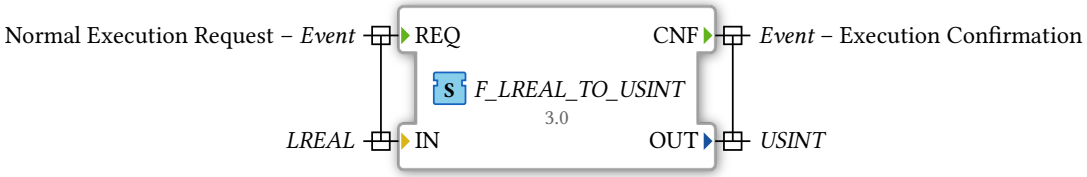
The function block `F_LREAL_TO_ULINT` offers a simple and efficient way to convert floating-point numbers to unsigned integers. Its clear interface and direct functionality make it ideal for applications requiring precise conversion without rounding.

F_LREAL_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LREAL_TO_USINT` converts an LREAL value (64-bit floating-point number) to a USINT value (8-bit unsigned integer). This conversion is particularly useful when floating-point numbers need to be converted to unsigned integers, for example, for controlling hardware registers or communicating with systems that can only process integers.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: LREAL (64-bit floating-point number) – The input value to be converted.

Data Outputs

- **OUT**: USINT (8-bit unsigned integer) – The result of the conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion by executing the algorithm `REQ` as soon as the event `REQ` occurs. The algorithm converts the LREAL value `IN` to a USINT value using the function `LREAL_TO_USINT` and stores the result in `OUT`. The event `CNF` is then triggered to signal the completion of the conversion.

Technical Details

- The conversion truncates the floating-point value to an 8-bit unsigned integer. Note that values outside the range of 0 to 255 can lead to unexpected results (overflow/underflow).
- The block is implemented as a simple function block (SimpleFB) and uses an ST algorithm for the conversion.

State Overview

The function block has no internal state. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Conversion of sensor values (e.g., temperature, voltage) into unsigned integers for controlling hardware.
- Communication with systems that can only process integers (e.g., older controllers or protocols).
- Data reduction when high precision is not required.

⚖️ Comparison with Similar Blocks

- Compared to `F_REAL_TO_USINT` (conversion of 32-bit floating-point numbers), this block offers higher input precision through the use of 64-bit floating-point numbers.
- Similar blocks such as `F_LREAL_TO_INT` or `F_LREAL_TO_UINT` offer conversions to other integer types with different value ranges.

🛠️ Related Exercises

- Exercise_126b_sub

Conclusion

The `F_LREAL_TO_USINT` function block is a simple yet effective tool for converting 64-bit floating-point numbers to 8-bit unsigned integers. It is particularly suitable for applications requiring a limited output size; however, the input signal's range should be considered to avoid overflows.

LREAL conversions

Here you will find functions for converting the LREAL data type.

🛠️ Related exercises

- Uebung_003a0
- Uebung_003a0_AX
- Uebung_010c2
- Uebung_010c2_AX
- Uebung_035a
- Uebung_035a2
- Uebung_035a3

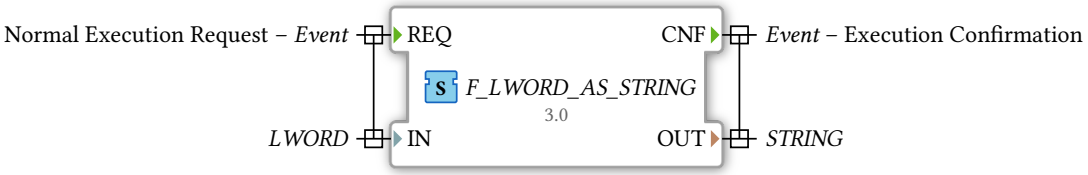
LWORD

F_LWORD_AS_STRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LWORD_AS_STRING` converts a `LWORD` data type to a `STRING` data type. This block is particularly useful when large binary data (64-bit) needs to be converted into a readable string.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- `IN` (LWORD): The input value to be converted.

Data Outputs

- `OUT` (STRING): The result of the conversion.

Adapters

No adapters are present.

Functionality

The function block performs the conversion as soon as the `REQ` event is triggered. The `LWORD` value at input `IN` is converted to `STRING` and output `OUT`. Successful conversion is signaled by the `CNF` event.

Technical Features

- The function block uses the standard algorithm `LWORD_AS_STRING` for conversion.
- The block is part of the `iec61131::conversion` package.

State Overview

1. **Idle State**: The function block waits for the `REQ` event.
2. **Conversion State**: Upon receiving `REQ`, the conversion is performed.
3. **Completion State**: After successful conversion, the `CNF` event is triggered.

Application Scenarios

- Conversion of large binary data into readable strings.
- Use in systems that need to work with binary and textual data.

⚖️ Comparison with Similar Function Blocks

- Unlike other conversion blocks that may process different data types, this one is specifically optimized for converting `LWORD` to `STRING`.

Conclusion

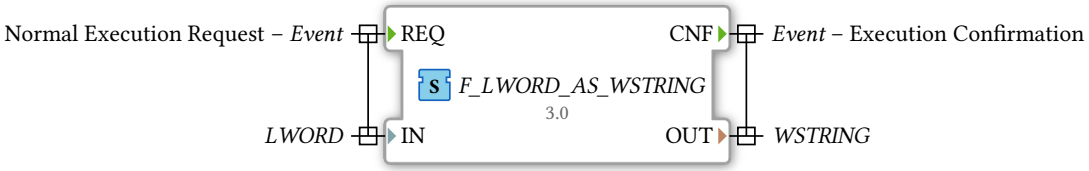
The `F_LWORD_AS_STRING` function block provides a simple and efficient way to convert large binary data into strings. Its clear interface structure and simple functionality make it a valuable component in automation technology.

F_LWORD_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LWORD_AS_WSTRING` is used to convert an LWORD data type to a WSTRING data type. This functionality is particularly useful when data needs to be exchanged between systems that use different data types.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This event input triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. This event is output once the conversion is complete.

Data Inputs

- **IN**: Input variable of type LWORD containing the value to be converted.

Data Outputs

- **OUT**: Output variable of type WSTRING containing the converted value.

Adapters

No adapters are present.

Functionality

The function block performs the conversion from LWORD to WSTRING as soon as the event `REQ` is received. The conversion is carried out using the integrated function `LWORD_AS_WSTRING`. After successful conversion, the event `CNF` is output.

Technical Features

- The function block is a simple FB (SimpleFB) and contains a single algorithm.
- The conversion occurs directly and without any additional delay.

State Overview

The function block has no internal states. Execution occurs immediately upon receipt of the `REQ` event.

Application Scenarios

- Data conversion in industrial control systems.
- Exchange of data between systems that use different data types.
- Integration into larger control applications where type conversion is required.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks, `F_LWORD_AS_WSTRING` specializes in converting LWORD to WSTRING.
- Other blocks might support more general conversions, but may be less efficient for this specific use case.

Conclusion

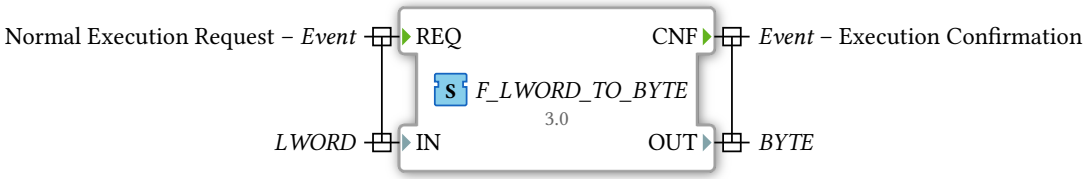
The `F_LWORD_AS_WSTRING` function block offers an efficient and direct method for converting LWORD to WSTRING. Its simple structure and clear functionality make it a reliable tool in control applications.

F_LWORD_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LWORD_TO_BYTE` converts a 64-bit value (LWORD) to an 8-bit value (BYTE). This conversion is particularly useful when data needs to be exchanged or processed between systems with different word lengths.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at data output `OUT`.

Data Inputs

- **IN**: The input for the 64-bit value (LWORD) to be converted.

Data Outputs

- **OUT**: The output for the converted 8-bit value (BYTE).

Adapters

- No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The value at input `IN` is read and converted into a BYTE value using the function `LWORD_TO_BYTE`. The result is output at `OUT`, and the event `CNF` signals the completion of the operation.

Technical Details

- The conversion can result in data loss because a 64-bit value is converted to an 8-bit value. It is important to ensure that the original value is within the representable range of a byte (0 to 255).

- The function block is implemented as a simple function block (SimpleFB) and uses an ST algorithm for the conversion.

State Overview

1. **Idle State**: Waits for the event `REQ`.
2. **Active State**: Performs the conversion and outputs the result.
3. **Completion State**: Signals completion with `CNF` and returns to the idle state.

Application Scenarios

- Data Reduction: When only the lower 8 bits of a 64-bit value are relevant.
- Communication with devices that can only process BYTE data.
- Memory optimization in systems with limited resources.

⚖️ Comparison with Similar Components

- **F_WORD_TO_BYTE**: Converts a 16-bit value (WORD) to a BYTE.
- **F_DWORD_TO_BYTE**: Converts a 32-bit value (DWORD) to a BYTE.
- **F_LWORD_TO_WORD**: Converts a 64-bit value to a 16-bit value.

Conclusion

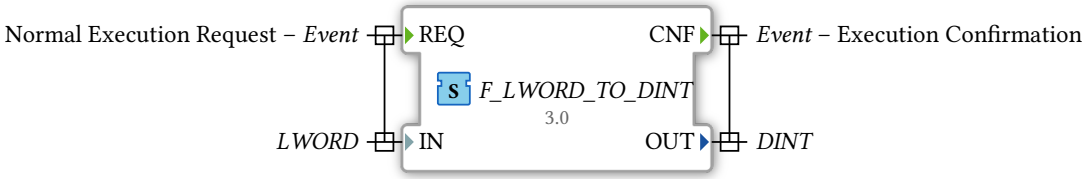
The function block `F_LWORD_TO_BYTE` offers a simple and efficient way to convert 64-bit data to 8-bit data. It is particularly useful in scenarios where only a portion of the data is needed or the target platform supports limited data types. However, be aware of the potential data loss during the conversion.

F_LWORD_TO_DINT

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_LWORD_TO_DINT` converts a `LWORD` data type to a `DINT` data type. This block is particularly useful in scenarios where type conversion between these two data types is required, for example, when processing large binary datasets or when communicating between systems with different data type requirements.



Interface Structure

Event Inputs

- **REQ**: This event input triggers the execution of the function block. It is connected to the data input `IN`.

Event Outputs

- **CNF**: This event output signals the successful execution of the function block. It is connected to the data output `OUT`.

Data Inputs

- **IN** (LWORD): The input for the `LWORD` value to be converted.

Data Outputs

- **OUT** (DINT): The output for the converted `DINT` value.

Adapters

This function block does not use any adapters.

Functionality

The function block converts the incoming `LWORD` value (`IN`) into a `DINT` value (`OUT`) as soon as the `REQ` event is triggered. The conversion is performed using the built-in function `LWORD_TO_DINT`. After successful conversion, the `CNF` event is triggered to indicate completion of the operation.

Technical Features

- The function block has a simple structure and consists of a single algorithm that performs the conversion.
- There is no state machine or complex logic, as the conversion is performed directly without intermediate steps.

State Overview

Since this is a simple function block, there are no state transitions or complex state logic. The block performs the conversion on each `REQ` event and signals completion with `CNF`.

Application Scenarios

- **Data Conversion**: In systems where `LWORD` data needs to be converted to `DINT`, e.g., in communication between different hardware components.
- **Data Processing**: When processing large binary data sets that need to be further processed in a smaller format.

⚖️ Comparison with Similar Blocks

- **F_LWORD_TO_INT**: Similar function block, but converts to the smaller `INT` data type.
- **F_LWORD_TO_UDINT**: Converts to an unsigned `DINT` data type (`UDINT`).

Conclusion

The `F_LWORD_TO_DINT` function block is a simple and efficient tool for converting `LWORD` to `DINT` data. Its simplicity and direct functionality make it a reliable solution for type conversions in 4diac IDE-based systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

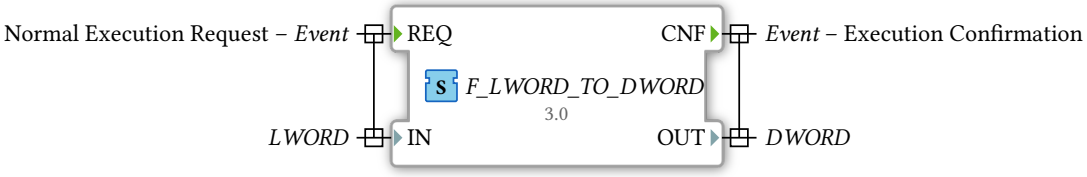
]

F_LWORD_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LWORD_TO_DWORD` is used to convert an LWORD data type to a DWORD data type. This conversion is useful when data needs to be exchanged between systems or components that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN**: Input for the LWORD value to be converted.

Data Outputs

- **OUT**: Output for the converted DWORD value.

Adapters

- No adapters available.

Functionality

The function block performs the conversion from LWORD to DWORD when the event `REQ` is triggered. The algorithm within the function block uses the function `LWORD_TO_DWORD` to perform the conversion. After successful conversion, the event `CNF` is triggered to signal the completion of the operation.

Technical Features

- The function block is a simple function block (SimpleFB) and performs a direct conversion.
- There are no additional states or complex logic, as the conversion occurs in a single step.

State Overview

Since this is a simple function block, there are no state transitions or state machines. The function block reacts directly to the `REQ` event and outputs the result via `CNF`.

Application Scenarios

- **Data Compatibility**: When one system creates LWORD data, but another system can only process DWORD data.
- **Protocol Conversion**: For communication between different protocols or interfaces that use different data types.

⚖️ Comparison with Similar Function Blocks

- **F_DWORD_TO_LWORD**: Performs the reverse conversion (DWORD to LWORD).
- **F_WORD_TO_DWORD**: Converts WORD to DWORD, but has a smaller input size compared to LWORD.

Conclusion

The `F_LWORD_TO_DWORD` function block is a simple and efficient tool for converting LWORD to DWORD data. Its simplicity and direct functionality make it a reliable solution for data conversion tasks in 4diac IDE-based systems.

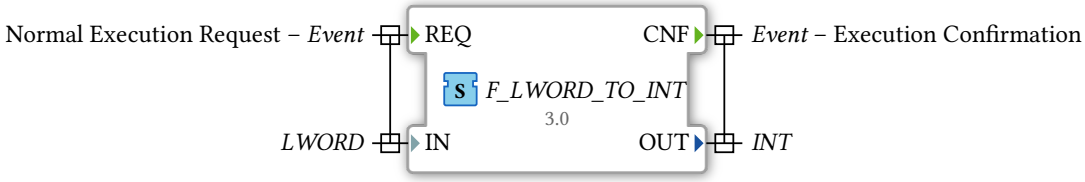
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

F_LWORD_TO_INT

• – ◦ • – ◦ • – ◦ •

The function block `F_LWORD_TO_INT` converts a `LWORD` value (64-bit unsigned integer) to a `INT` value (16-bit signed integer). This conversion is useful when data needs to be exchanged or processed between systems with different word lengths.



- **REQ**: Starts the conversion. This input is connected to the data input `IN`.
- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.
- **IN**: The input value of type `LWORD` (64-bit unsigned integer) to be converted.
- **OUT**: The output value of type `INT` (16-bit signed integer) containing the result of the conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- No adapters are available.

The function block performs the conversion by transforming the `LWORD` value at input `IN` into a `INT` value and outputting it at output `OUT`. The conversion occurs upon an event at input `REQ` and is confirmed by an event at output `CNF`.

- Converting a 64-bit value to a 16-bit value can result in data loss if the original value is outside the representable range of `INT`.
- Converting from a 64-bit value to a 16-bit value can lead to data loss if the original value is outside the representable range of `INT`.
- The function block is implemented as a simple function block (`SimpleFB`) and uses an ST algorithm for conversion.

The function block has no internal states. The conversion occurs immediately upon an event at the input `REQ`.

- Conversion of 64-bit data values for systems that can only process 16-bit data.
- Data reduction when only the lower 16-bit portion of a 64-bit value is relevant.
- `F_DWORD_TO_INT`: Converts a 32-bit value to a 16-bit value.
- `F_LWORD_TO_DINT`: Converts a 64-bit value to a 32-bit value.
- `F_LWORD_TO_SINT`: Converts a 64-bit value to an 8-bit value.

The function block `F_LWORD_TO_INT` provides a simple and efficient way to convert 64-bit data values to 16-bit data values. However, when using it, be aware of the potential data loss if the original value is outside the representable range of `INT`.

Functionality

Application Scenarios

State Overview

Technical Features

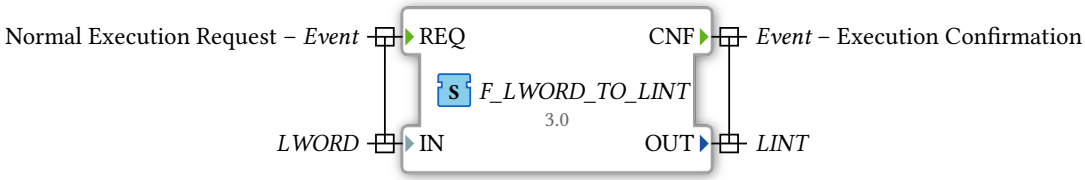
⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

F_LWORD_TO_LINT

• – ○ • – ○ • – ○ •

The function block `F_LWORD_TO_LINT` converts a `LWORD` value (64-bit unsigned) to a `LINT` value (64-bit signed). This block is particularly useful in scenarios where type conversion between these two data types is required.



- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.
- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.
- **IN**: Input for the `LWORD` value to be converted.
- **OUT**: Output for the converted `LINT` value.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- No adapters are present.

Functionality

The function block performs the conversion from `LWORD` to `LINT` as soon as the event `REQ` is triggered. The conversion is performed using the function `LWORD_TO_LINT(IN)`. Upon successful conversion, the event `CNF` is triggered.

- The function block is a simple FB (SimpleFB) and uses an ST algorithm for the conversion.
- The conversion is performed directly without additional state management.

Since it is a simple function block, there are no complex state transitions. The block responds to `REQ` by executing the algorithm and signals completion with `CNF`.

- Type conversion in control applications where `LWORD` data needs to be converted to `LINT`.
- Similar blocks could be `F_DWORD_TO_DINT` or `F_WORD_TO_INT`, but these are designed for smaller data types.
- Unlike generic conversion blocks, this one is specifically optimized for converting `LWORD` to `LINT`.

The `F_LWORD_TO_LINT` function block offers an efficient and direct method for converting `LWORD` to `LINT`. Its simplicity and specialization make it a reliable component in control applications.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

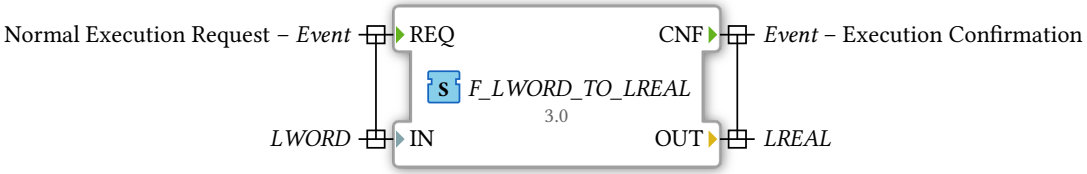
Conclusion

F_LWORD_TO_LREAL

• – ○ • – ○ • – ○ •

Introduction

The function block `F_LWORD_TO_LREAL` converts an LWORD data value (64-bit unsigned integer) to an LREAL data value (64-bit floating-point number according to IEEE 754). This block is particularly useful in applications where direct type conversion between these two data types is required.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. The input value `IN` is processed upon this event.

Event Outputs
- **CNF**: Signals the completion of the conversion and outputs the converted value `OUT`.

Data Inputs
- **IN** (LWORD): The input value to be converted.

Data Outputs
- **OUT** (LREAL): The result of the conversion.

Adapters
This function block has no adapters.

Operation
When the `REQ` event is triggered, the algorithm that converts the `IN` value of type LWORD to an LREAL value is executed. The result is output via the `OUT` output, and the `CNF` event signals successful conversion.

Technical Features
- The conversion is performed directly without additional parameters.
- The function block is optimized for use in real-time systems.

State Overview
The function block has a simple state machine:

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Complete**: Sends the `CNF` event with the result.

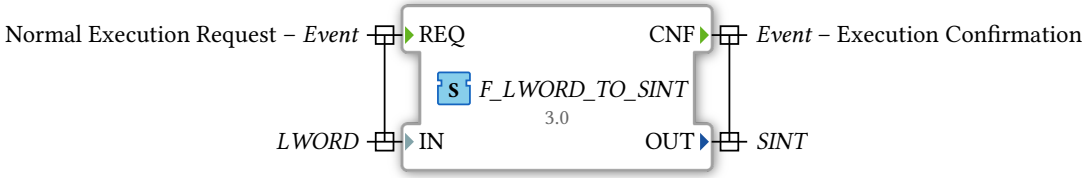
Application Scenarios
- Data processing in control systems where LWORD values need to be converted to floating-point numbers.
- Integration into larger function block networks that process mixed data types.

⚖️ Comparison with similar blocks
- Similar blocks such as `F_DWORD_TO_REAL` or `F_WORD_TO_REAL` offer conversions for smaller data types.
- `F_LWORD_TO_LREAL` is specifically optimized for 64-bit conversions.

Conclusion
The `F_LWORD_TO_LREAL` function block is an efficient tool for directly converting 64-bit integers to floating-point numbers. Its simple interface and reliable operation make it ideal for use in complex automation solutions.

F_LWORD_TO_SINT
• – ◦ • – ◦ • – ◦ •

Introduction
The function block `F_LWORD_TO_SINT` converts a `LWORD` value (64-bit unsigned integer) to a `SINT` value (8-bit signed integer). This conversion is useful when data needs to be transferred from a larger range of values to a smaller, more specific range.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. When activated, the input value `IN` is processed.

Event Outputs

- **CNF**: Signals successful completion of the conversion and provides the result value `OUT`.

Data Inputs

- **IN** (`LWORD`): The input value that is converted from `LWORD` to `SINT`.

Data Outputs

- **OUT** (`SINT`): The result of the conversion.

Adapters

This function block does not use adapters.

Functionality

The function block performs a direct type conversion of the input value `IN` from type `LWORD` to type `SINT`. The conversion is carried out by the function `LWORD_TO_SINT(IN)`, which is implemented in the algorithm `REQ`. Upon successful conversion, the event `CNF` is triggered.

Technical Features

- **Type Conversion**: Since `LWORD` covers a larger range of values than `SINT`, data loss can occur if the input value is outside the representable range of `SINT` (-128 to 127).
- **Algorithm**: The algorithm is implemented in Structured Text (ST) and performs a direct conversion.

State Overview

1. **Idle State**: Waits for the event `REQ`.
2. **Execution State**: The conversion is performed upon `REQ`.
3. **Acknowledgement State**: After successful conversion, `CNF` is triggered, and the block returns to the idle state.

Application Scenarios

- **Data Reduction**: When only the lower 8 bits of a 64-bit value are relevant.
- **Communication**: When transferring data between systems with different word sizes.
- **Memory Optimization**: When memory space is limited and smaller data types are preferred.

⚖️ Comparison with similar building blocks

- **F_LWORD_TO_INT**: Converts `LWORD` to `INT` (16-bit signed integer).
- **F_LWORD_TO_DINT**: Converts `LWORD` to `DINT` (32-bit signed integer).
- **F_LWORD_TO_USINT**: Converts `LWORD` to `USINT` (8-bit unsigned integer).

Unlike these blocks, `F_LWORD_TO_SINT` is specifically for conversion to an 8-bit signed integer.

Conclusion

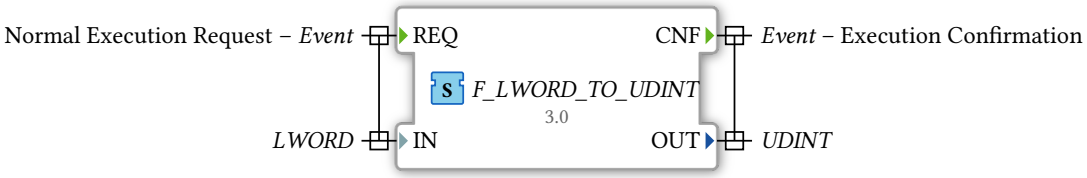
The function block `F_LWORD_TO_SINT` provides a simple and efficient way to convert 64-bit values to 8-bit signed integers. It is particularly useful in scenarios where memory efficiency or the processing of portions of a larger data type is required. However, when using it, care must be taken to avoid potential data loss with values outside the `SINT` range.

F_LWORD_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LWORD_TO_UDINT` converts a `LWORD` value (64-bit unsigned integer) to a `UDINT` value (32-bit unsigned integer). This conversion is useful when data needs to be transferred or processed between systems with different word lengths.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- `IN` (LWORD): The 64-bit unsigned integer value to be converted.

Data Outputs

- `OUT` (UDINT): The resulting 32-bit unsigned integer value after the conversion.

Adapters

This function block does not use any adapters.

Functionality

When the `REQ` event is triggered, the value of `IN` is converted from `LWORD` to `UDINT`, and the result is output to `OUT`. The `CNF` event signals the successful completion of the conversion.

The conversion is performed by the function `LWORD_TO_UDINT`, which returns the lower 32 bits of the `LWORD` value as `UDINT`.

Technical Features

- The function block is suitable for use in real-time systems.
- The conversion is deterministic and has a constant execution time.
- Values greater than the maximum `UDINT` value (2^32 - 1) are truncated.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Communication between systems with different word widths.
- Processing data from external sources that provide 64-bit values, but only 32-bit values are required.
- Reducing memory usage when dealing with known small value ranges.

⚖️ Comparison with Similar Function Blocks

- `F_LWORD_TO_DINT`: Converts to a 32-bit signed integer.
- `F_LWORD_TO_UINT`: Converts to a 16-bit unsigned integer.
- `F_LWORD_TO_WORD`: Converts to a 16-bit unsigned value.

Conclusion

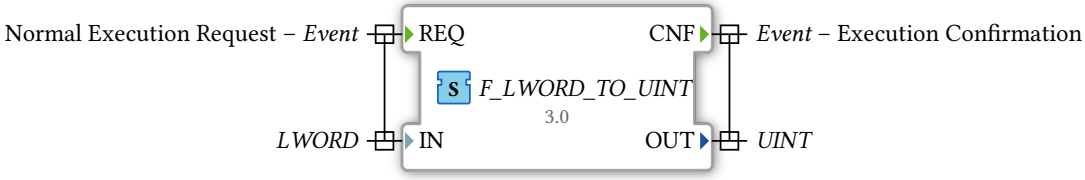
The F_LWORD_TO_UDINT function block provides a simple and efficient way to convert 64-bit values to 32-bit values. It is particularly useful in systems that need to handle different word lengths.

F_LWORD_TO_UINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_LWORD_TO_UINT converts a LWORD data type to a UINT data type. This function block is particularly useful in scenarios where type conversion between these two data types is required, for example, in communication between systems with different data formats.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This event input triggers the conversion. It is linked to the data input IN.

Event Outputs

- **CNF**: Execution Confirmation. This event is triggered after successful conversion and is associated with the data output OUT.

Data Inputs

- **IN**: Input data of type LWORD to be converted to the UINT data type.

Data Outputs

- **OUT**: Output data of type UINT containing the result of the conversion.

Adapters

No adapters available.

Functionality

The function block performs the conversion from LWORD to UINT when the event REQ is received. The conversion is performed using the function LWORD_TO_UINT(IN), which converts the value of IN and passes the result to OUT. After successful conversion, the event CNF is triggered.

Technical Features

- The function block is a simple algorithm (SimpleFB) and does not perform any complex state transitions.
- The conversion occurs directly without additional conditions or delays.

State Overview

Since it is a simple function block, there are no explicit states. The functionality is limited to directly responding to the REQ event, subsequently outputting the converted value, and triggering the CNF event.

Application Scenarios

- Type conversion in communication interfaces between systems that use different data types.
- Data processing in control systems where conversion from LWORD to UINT is required.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks could be F_DWORD_TO_UINT or F_WORD_TO_UINT, which, however, process different input data types.
- The main difference lies in the input data type (LWORD), which has a larger bit width than DWORD or WORD.

Conclusion

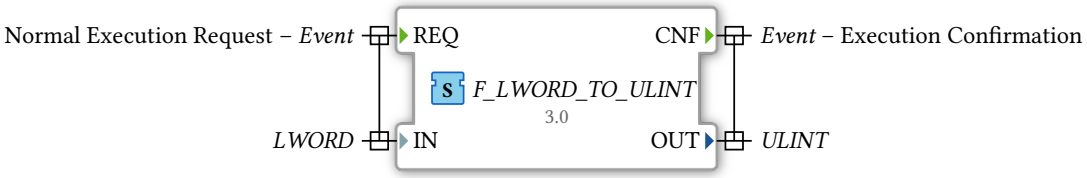
The F_LWORD_TO_UINT function block provides a simple and efficient way to convert LWORD data to the UINT data type. Its clear interface structure and direct functionality make it a useful component in various use cases, especially where type conversions between these data types are required.

F_LWORD_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_LWORD_TO_ULINT is used to convert an LWORD data type to a ULINT data type. This conversion is particularly necessary in scenarios where data of different bit lengths needs to be processed or transmitted.



Interface Structure

Event Inputs

- **REQ**: This event input triggers the conversion. It is linked to the data input IN.

Event Outputs

- **CNF**: This event output signals the successful completion of the conversion. It is linked to the data output OUT.

Data Inputs

- **IN**: The input expects a variable of type LWORD, which is to be converted.

Data Outputs

- **OUT**: The output returns the result of the conversion as ULINT.

Adapters

This function block has no adapters.

Functionality

When the event input REQ is triggered, the algorithm is started, which converts the value of IN of type LWORD to type ULINT and outputs the result as OUT. The event CNF is then triggered to signal the successful conversion.

Technical Features

- This function block is particularly efficient because it directly utilizes the built-in function LWORD_TO_ULINT.
- There are no special constraints or limitations when using this function block.

State Overview

This function block has no internal states. The conversion occurs immediately upon triggering the REQ event.

Application Scenarios

- Data conversion in control systems where different data types need to be processed.
- Integration into larger systems that require type conversion between LWORD and ULINT.

⚖️ Comparison with Similar Function Blocks

Compared to other conversion blocks, F_LWORD_TO_ULINT specializes in converting LWORD to ULINT and therefore offers an optimized solution for this specific use case.

Conclusion

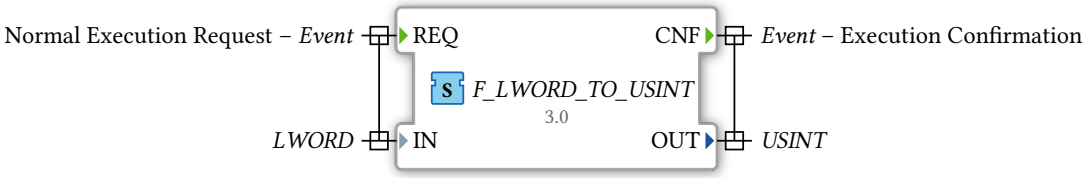
The `F_LWORD_TO_ULINT` function block is a simple and efficient building block for converting between the data types `LWORD` and `ULINT`. Its clear interface structure and direct functionality make it a reliable component in control applications.

F_LWORD_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LWORD_TO_USINT` converts a 64-bit value (LWORD) to an 8-bit unsigned integer (USINT). This conversion is particularly useful when data from a larger range of values needs to be transformed into a smaller, more specific range.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion and outputs the result via the data output `OUT`.

Data Inputs

- **IN (LWORD)**: The 64-bit value to be converted to a USINT value.

Data Outputs

- **OUT (USINT)**: The resulting 8-bit unsigned integer value after the conversion.

Adapters

No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The algorithm converts the 64-bit input value `IN` into an 8-bit unsigned integer value and stores the result in `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

Technical Details

- The conversion is performed by the built-in function `LWORD_TO_USINT`.
- Note that when converting from a larger to a smaller data type, data loss may occur if the original value is outside the representable range of the target type.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Conversion State**: Performs the conversion. 3. **Acknowledge State**: Sends the `CNF` event and returns to the idle state.

Application Scenarios

- Processing 64-bit data that needs to be reduced to 8 bits.
- Communication between systems with different data widths.
- Memory optimization in embedded systems.

⚖️ Comparison with Similar Components

- **F_LWORD_TO_BYTE**: Also converts LWORD to an 8-bit value, but without a sign (BYTE).

- **F_LWORD_TO_INT**: Converts to a 16-bit signed integer, offering a larger value range, but with a sign.

Conclusion

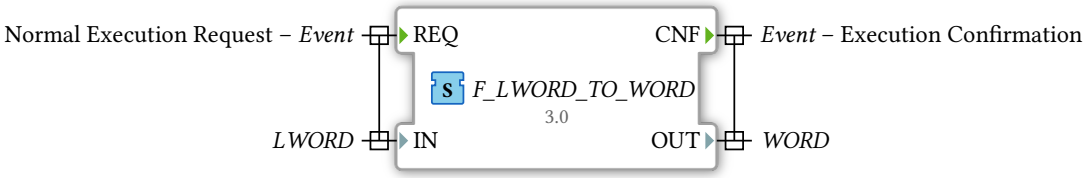
The `F_LWORD_TO_USINT` function block is a simple and efficient tool for converting 64-bit data to 8-bit unsigned integer values. It is particularly useful in scenarios where storage space or data bandwidth is limited. However, it should be ensured that the original data is within the representable range of the target type to avoid data loss.

F_LWORD_TO_WORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_LWORD_TO_WORD` is used to convert a 64-bit value (LWORD) to a 16-bit value (WORD). This conversion is particularly necessary in scenarios where data needs to be exchanged or processed between systems with different word widths.



Interface Structure

Event Inputs

- **REQ**: Signals the start of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: LWORD (64-bit) – The input value to be converted.

Data Outputs

- **OUT**: WORD (16-bit) – The converted output value.

Adapters

No adapters available.

Functionality

This function block performs the conversion from a 64-bit value (LWORD) to a 16-bit value (WORD). During the conversion, the lower 16 bits of the input value are retained, while the upper 48 bits are discarded. The conversion occurs synchronously with the `REQ` event, and the result is confirmed with the `CNF` event.

Technical Features

- The conversion is performed without rounding or scaling, as it is a direct bit conversion.
- No check is performed to prevent potential data loss due to word width reduction. The user must ensure that the input value is within the permissible range.

State Overview

The function block has no internal states. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Communication between systems with different word widths.
- Processing data originating from 64-bit sources but intended for use in 16-bit systems.
- Reducing the data width to optimize storage space when the higher-order bits are not required.

⚖️ Comparison with Similar Function Blocks

- **F_LWORD_TO_DWORD**: Converts LWORD to DWORD (32-bit), retaining more data bits.
- **F_LWORD_TO_BYTE**: Converts LWORD to BYTE (8-bit), reducing the data width even further.
- **F_LWORD_TO_**: Similar function blocks for other target data types, each with a different bit width as a result.

Conclusion

The `F_LWORD_TO_WORD` function block offers a simple and efficient way to reduce 64-bit data to 16-bit data. It is particularly useful in heterogeneous system environments where data needs to be exchanged between components with different word widths. However, due to the lack of data loss checks, users should ensure that the conversion is appropriate and safe.

LWORD Conversions

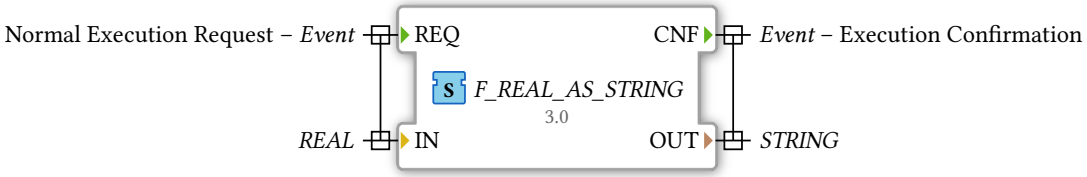
Here you will find functions for converting the LWORD data type.

REAL

F_REAL_AS_STRING

• – ◦ • – ◦ • – ◦ •

The function block `F_REAL_AS_STRING` converts a REAL value to a STRING value. It is part of the package `iec61131::conversion` and enables the easy conversion of numeric values to a string.



- **REQ**: Starts the conversion of the REAL value to a STRING. This input is linked to the data input `IN`.
- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.
- **IN**: The REAL value to be converted to a STRING.
- **OUT**: The converted STRING value of input `IN`.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block has no adapters.

Functionality

When an event is present at input `REQ`, the algorithm is executed, converting the REAL value of `IN` to a STRING and outputting the result to `OUT`. The event `CNF` is then triggered to signal the successful completion of the conversion.

- The function block uses the function `REAL_AS_STRING` for conversion.
- It is a simple function block without a state machine.

Since it is a simple function block, there are no state transitions. The conversion occurs directly upon triggering the `REQ` event.

- Conversion of sensor values for display on an HMI.
- Conversion of numerical values for logging in text files.

- Integration into systems that require string-based communication.
- Compared to other conversion blocks such as `F_INT_AS_STRING` or `F_BOOL_AS_STRING`, this one is specifically designed for real values.
- Simpler and more direct than universal conversion blocks, which may require additional parameters.

The `F_REAL_AS_STRING` function block offers an efficient and straightforward way to convert REAL values to STRINGs. Its simple interface and direct functionality make it ideal for applications where numeric values need to be processed as strings.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

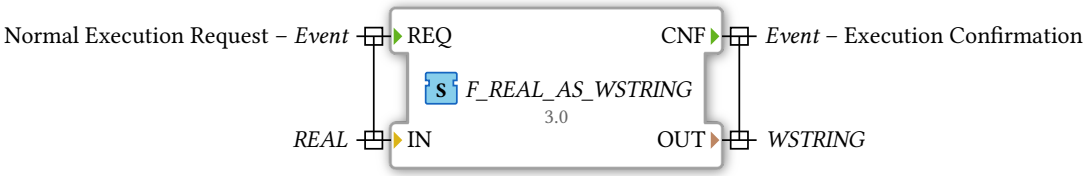
Conclusion

F_REAL_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_REAL_AS_WSTRING` converts a `REAL` value into a `WSTRING` value. This block is particularly useful when numeric values need to be converted into a string suitable for output or further processing.



Interface Structure

Event Inputs

- **REQ**: This input triggers the conversion. Upon receiving an event, the algorithm for converting the `REAL` value into a `WSTRING` value is started.

Event Outputs

- **CNF**: This event signals the successful completion of the conversion and returns the result as a `WSTRING` value.

Data Inputs

- **IN**: The input expects a `REAL` value, which is to be converted to a `WSTRING` value.

Data Outputs

- **OUT**: The output returns the converted `WSTRING` value after the conversion is complete.

Adapters

This function block does not have any adapter interfaces.

Functionality

The function block `F_REAL_AS_WSTRING` converts the `REAL` value at input `IN` to a `WSTRING` value upon receiving an incoming `REQ` event. The result is provided at output `OUT` and confirmed by a `CNF` event.

The conversion algorithm is implemented as a simple ST (Structured Text) statement:
ALGORITHM REQ OUT := REAL_AS_WSTRING(IN); END_ALGORITHM

Technical Features

- The function block is part of the `iec61131::conversion` package.

- The conversion is performed directly without additional parameters or settings.

Status Overview

1. **Idle**: Waits for a `REQ` event.
2. **Processing**: Performs the conversion.
3. **Completed**: Sends the `CNF` event with the result.

Application Scenarios

- **Data Output**: Converts numeric values for display on a control panel or in a log file.
- **Data Communication**: Prepares numeric values for transmission as strings in network protocols.

⚖️ Comparison with Similar Function Blocks

- **F_REAL_TO_STRING**: Similar function block, but it converts to `STRING` instead of `WSTRING`. `F_REAL_AS_WSTRING` is better suited for Unicode-enabled environments.
- **F_WSTRING_AS_REAL**: Performs the reverse conversion (from `WSTRING` to `REAL`).

Conclusion

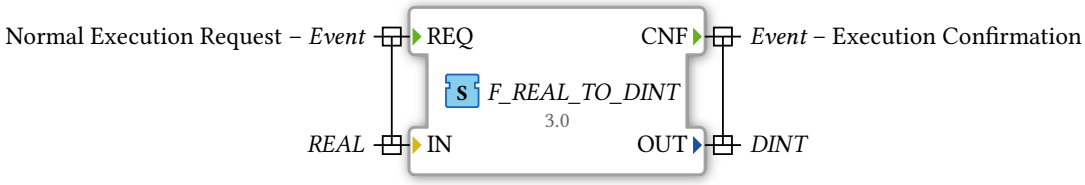
The function block `F_REAL_AS_WSTRING` provides a simple and efficient way to convert `REAL` values to `WSTRING` values. Its clear interface and direct functionality make it ideal for applications that require a Unicode-compatible string representation of floating-point numbers.

F_REAL_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_REAL_TO_DINT` converts a REAL value (floating-point number) into a DINT value (32-bit integer). This conversion is performed by rounding the REAL value down to the nearest integer.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. Triggered after successful conversion.

Data Inputs

- **IN**: Input for the REAL value to be converted.

Data Outputs

- **OUT**: Output for the converted DINT value.

Adapters

No adapters available.

Functionality

When the `REQ` event is triggered, the algorithm that converts the `IN` value from REAL to DINT is executed. The result is output at `OUT`, and the `CNF` event is triggered.

Technical Features

- The conversion is performed by rounding down the REAL value.
- The function block is simple and has no internal states.

State Overview

No states are present, as the block is stateless.

Application Scenarios

- Conversion of sensor values (floating-point) to integer values for control logic.
- Data processing in control systems where integer values are required.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks might offer other conversion methods, e.g., rounding or rounding up.
- Compared to generic conversion blocks, this one specializes in REAL-to-DINT conversion.

Conclusion

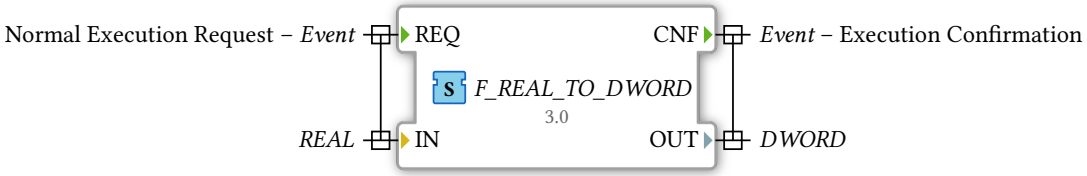
The `F_REAL_TO_DINT` function block is a simple and efficient building block for converting floating-point numbers to integers. It is particularly useful in scenarios where fast and direct conversion is required.

F_REAL_TO_DWORD

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_REAL_TO_DWORD` converts a REAL value (floating-point number) to a DWORD value (32-bit unsigned integer). This conversion is particularly useful when floating-point values need to be processed in systems that only support integer operations.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. The value to be converted is provided via the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is output via the data output `OUT`.

Data Inputs

- **IN**: REAL value to be converted to a DWORD value.

Data Outputs

- **OUT**: The resulting DWORD value after conversion.

Adapters

This function block does not use any adapters.

Operation

When an event is received at input `REQ`, the algorithm is executed, converting the REAL value of `IN` to a DWORD value and outputting the result to `OUT`. The event `CNF` is then triggered to signal the completion of the operation.

Technical Features

- The conversion is performed by the function `REAL_TO_DWORD`, which directly converts the REAL value into a DWORD value. The floating-point value is rounded to the nearest integer.
- Note that data loss can occur with very large or small values, as the DWORD type offers only limited precision.

State Overview

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Processing sensor data that is in floating-point form but needs to be further processed in integer form.
- Integration into control systems that only support integer operations.
- Data conversion for communication protocols that require DWORD values.

⚖️ Comparison with Similar Function Blocks

- **F_REAL_TO_INT**: Converts REAL to INT (signed integer). In contrast, `F_REAL_TO_DWORD` returns an unsigned value.
- **F_REAL_TO_LREAL**: Converts REAL to LREAL (higher precision), while `F_REAL_TO_DWORD` converts the value to an integer type.

🛠️ Related Exercises

- Exercise_126b2_sub

Conclusion

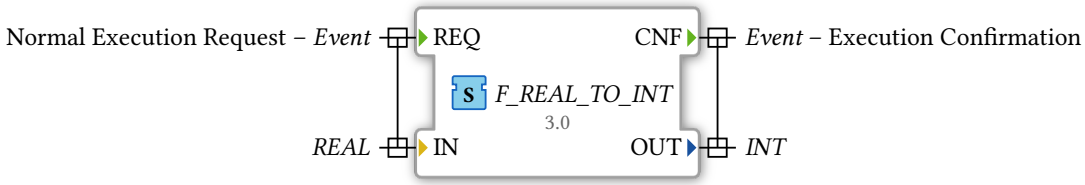
The function block `F_REAL_TO_DWORD` provides a simple and efficient way to convert floating-point values to unsigned integers. It is particularly useful in environments that only support integer operations or when data needs to be processed further in a specific format. However, when using this method, care must be taken to avoid potential data loss during the conversion process.

F_REAL_TO_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_REAL_TO_INT` converts a REAL value (floating-point number) to an INT value (integer). This conversion is particularly useful when data needs to be exchanged between systems that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN**: The input for the REAL value to be converted.

Data Outputs

- **OUT**: The output for the converted INT value.

Adapters

- No adapters are available.

Functionality

The function block performs the conversion by converting the REAL value at input `IN` into an INT value and outputting the result at output `OUT`. The conversion is triggered by the event `REQ` and confirmed by the event `CNF`.

Technical Details

- The conversion is performed using the function `REAL_TO_INT`, which converts the REAL value into an INT value.
- The function block is simple and efficient because it does not use complex algorithms or state machines.

State Overview

The function block has no internal states. The conversion occurs immediately after the `REQ` event is triggered.

Application Scenarios

- **Data Conversion**: Converting sensor values (floating-point numbers) to integers for further processing.
- **Interface Compatibility**: Adapting data between systems that expect different data types.

⚖️ Comparison with Similar Blocks

- **F_TRUNC**: Converts REAL to INT by truncating the decimal places, while `F_REAL_TO_INT` performs standard rounding.
- **F_ROUND**: Rounds REAL to INT, similar to `F_REAL_TO_INT`, but with specific rounding rules.

Conclusion

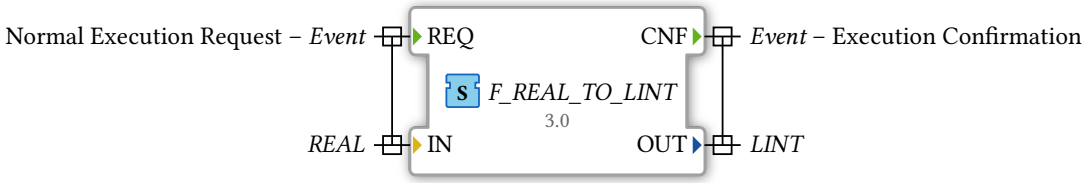
The `F_REAL_TO_INT` function block is a simple and effective tool for converting floating-point numbers to integers. Its simplicity and direct functionality make it ideal for applications requiring fast and straightforward data type conversion.

F_REAL_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_REAL_TO_LINT` converts a REAL value (floating-point number) to a LINT value (64-bit integer). This conversion is particularly useful when numerical calculations need to be performed with different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event input is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event, along with the converted value, is output at data output `OUT`.

Data Inputs

- **IN**: The input for the REAL value to be converted to a LINT value.

Data Outputs

- **OUT**: The output where the converted LINT value is displayed.

Adapters

This function block has no adapters.

Operation

When the `REQ` event is triggered, the algorithm `REQ` is executed. This function converts the REAL value at input `IN` into a LINT value using the function `REAL_TO_LINT` and writes the result to output `OUT`. The `CNF` event is then triggered to signal the successful completion of the conversion.

Technical Details

- Converting REAL to LINT can result in data loss if the REAL value has decimal places, as these are truncated during the conversion.
- This function block is part of the `iec61131::conversion` package.

State Overview

This function block has no internal states. The conversion occurs immediately after the `REQ` event is triggered.

Application Scenarios

- Conversion of sensor values (e.g., temperature measurements) for further processing into integer operations.
- Integration into control systems that need to process mixed data types.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks (e.g., `F_REAL_TO_INT`), `F_REAL_TO_LINT` offers higher precision through the use of the 64-bit LINT data type.
- Similar blocks, such as `F_LINT_TO_REAL`, perform the reverse conversion.

Conclusion

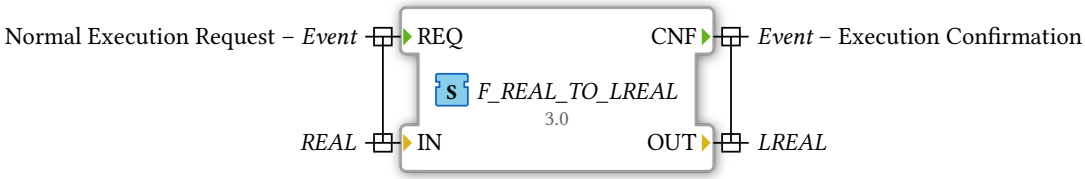
The `F_REAL_TO_LINT` function block is a simple and efficient block for converting floating-point numbers to 64-bit integers. Its use is particularly recommended in scenarios with high precision requirements, although potential data loss during conversion must be taken into account.

F_REAL_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_REAL_TO_LREAL` is used to convert a REAL value to an LREAL value. This block is part of the `iec61131::conversion` package and enables simple type conversion between the two numeric data types.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion. Linked to the data input `IN`.

Event Outputs

- **CNF**: Execution confirmation. Triggered after successful conversion. Linked to the data output `OUT`.

Data Inputs

- **IN**: Input value of type `REAL` to be converted.

Data Outputs

- **OUT**: Output value of type `LREAL` containing the result of the conversion.

Adapters

This function block does not use any adapters.

Functionality

Upon receiving the event `REQ`, the function block converts the REAL value `IN` to an LREAL value and assigns the result to the output `OUT`. The event `CNF` is then triggered to confirm successful execution.

Technical Features

- The function block is a simple building block without internal states or complex logic.
- Conversion is performed directly by assignment, as `LREAL` offers higher accuracy than `REAL` and can therefore represent all values without loss.

State Overview

Since it is a simple function block, there are no state transitions or management. Execution occurs atomically upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control applications where higher accuracy is required.
- Integration into existing systems that use different data types.
- Use in mathematical calculations that require higher precision.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_INT_TO_REAL` or `F_DINT_TO_LINT` also perform type conversions, but between different data types.
- Unlike complex conversion blocks, `F_REAL_TO_LREAL` offers a simple and direct conversion without additional parameters or logic.

Conclusion

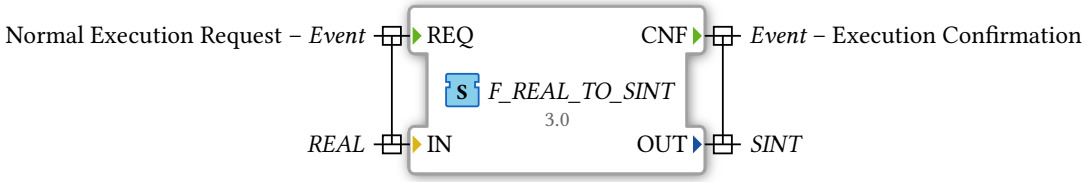
The `F_REAL_TO_LREAL` function block is a useful tool for the simple and efficient conversion of REAL to LREAL values. Its simplicity and direct functionality make it ideal for applications that require fast type conversion without added complexity.

F_REAL_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_REAL_TO_SINT` converts a `REAL` value (floating-point number) to a `SINT` value (8-bit signed integer). This block is particularly useful in control applications where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. The value at input `IN` is processed upon this event.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event, along with the converted value, is output at `OUT`.

Data Inputs

- **IN**: Input for the `REAL` value to be converted to a `SINT` value.

Data Outputs

- **OUT**: Output for the converted `SINT` value.

Adapters

- This function block does not have any adapters.

Functionality

The function block performs the conversion by transforming the `REAL` value at input `IN` into a `SINT` value using the function `REAL_TO_SINT`. The conversion occurs when the event `REQ` is triggered, and the result is output with the event `CNF`.

Technical Features

- The function block is implemented as a simple function block (`SimpleFB`) containing only one algorithm.
- The conversion truncates the `REAL` value to the value range of `SINT` (-128 to 127). Values outside this range result in an overflow.

State Overview

The function block has no internal state. The conversion occurs immediately upon the occurrence of the `REQ` event.

Application Scenarios

- Type conversion in control algorithms where floating-point values need to be converted to integers.
- Signal processing where `REAL` sensor data needs to be reduced to `SINT`.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_REAL_TO_INT` or `F_REAL_TO_USINT` offer conversions to other integer types. `F_REAL_TO_SINT` is specific to 8-bit signed integers.

Conclusion

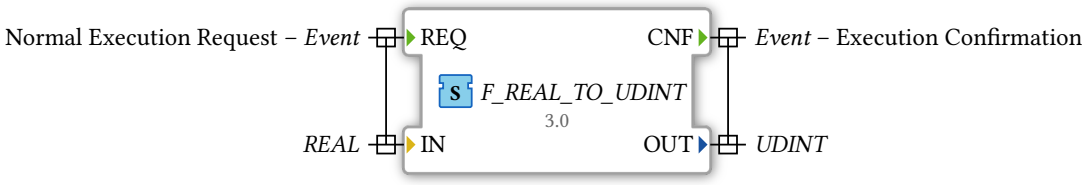
The `F_REAL_TO_SINT` function block is a simple and efficient building block for converting floating-point numbers to 8-bit signed integers. Its use is advantageous in scenarios where such type conversion is required, particularly in control and signal processing applications.

F_REAL_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_REAL_TO_UDINT` converts a REAL value to a UDINT value. This block is part of the `iec61131::conversion` package and implements a simple but important type conversion for use in control applications.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (REAL): The input value to be converted from REAL to UDINT.

Data Outputs

- **OUT** (UDINT): The result of the conversion.

Adapters

This function block does not use any adapters.

Operation

When an event is received at input `REQ`, the algorithm is executed, which converts the REAL value at input `IN` into a UDINT value and outputs the result at output `OUT`. The event `CNF` is then triggered to signal the successful completion of the conversion.

Technical Features

- The conversion is performed using the built-in function `REAL_TO_UDINT`.
- The function block is deterministic and has no internal states.

State Overview

Since this is a simple function block, there are no state transitions or machines. The block performs the conversion on each `REQ` event and signals completion with `CNF`.

Application Scenarios

- Type conversion in control applications where REAL values need to be processed as UDINT.
- Integration into larger function block networks that process different data types.

⚖️ Comparison with Similar Function Blocks

- Similar conversion blocks like `F_REAL_TO_INT` or `F_REAL_TO_LREAL` offer conversions to other data types, but not specifically to UDINT.
- Unlike more complex blocks, `F_REAL_TO_UDINT` provides a simple and direct solution for this specific conversion.

🛠️ Related Exercises

- Exercise_072c

Conclusion

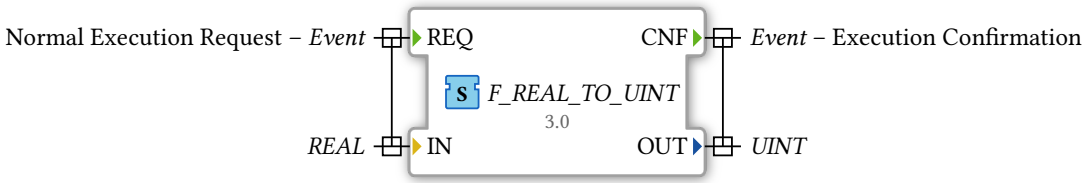
The `F_REAL_TO_UDINT` function block is a simple yet essential building block for type conversion in IEC 61499-based control applications. Its deterministic and stateless nature makes it reliable and easily integrated into various applications.

F_REAL_TO_UINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_REAL_TO_UINT` converts a REAL value (floating-point number) to a UINT value (unsigned integer). This conversion is particularly useful when exchanging data between systems that use different data types or when an explicit type conversion is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event input is triggered, the algorithm for converting the REAL value to a UINT value is executed.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event is triggered as soon as the algorithm has been successfully executed.

Data Inputs

- **IN**: The REAL input value to be converted to a UINT value.

Data Outputs

- **OUT**: The UINT output value representing the result of the conversion.

Adapters

This function block has no adapter interfaces.

Functionality

When the `F_REAL_TO_UINT` event is triggered, the function block `F_REAL_TO_UINT` converts the REAL value at input `IN` to a UINT value. The result is output at `OUT`, and the `CNF` event signals the successful completion of the operation.

The underlying algorithm is:
ALGORITHM REQ OUT := REAL_TO_UINT(IN); END_ALGORITHM

Technical Features

- The conversion is performed using the standard function `REAL_TO_UINT`, which rounds the REAL value to the nearest UINT value.
- Note that undefined behavior may occur with values outside the representable range of UINT (0 to 65535).

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Type conversion in control applications where floating-point values need to be converted to integers.
- Data preparation for systems that can only process unsigned integers.
- Interfaces between different system components with different data types.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks like `F_REAL_TO_INT` or `F_REAL_TO_UDINT`, this function block specializes in converting to unsigned 16-bit integers (UINT).
- Similar blocks might offer additional parameters such as rounding modes or range checks, which is not the case here.

Conclusion

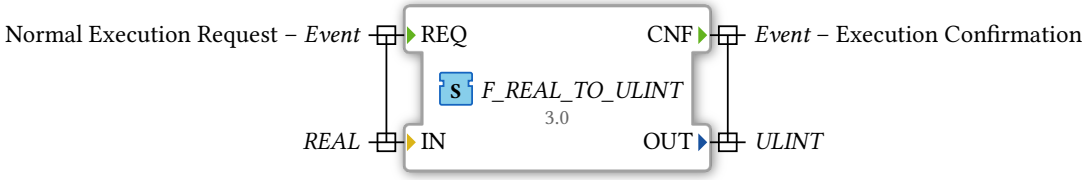
The `F_REAL_TO_UINT` function block provides a simple and efficient way to convert floating-point numbers to unsigned integers. Its clear interface structure and direct functionality make it ideal for applications requiring fast and straightforward type conversion. However, when using it, care must be taken to ensure the valid range of values is observed to avoid unexpected behavior.

F_REAL_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_REAL_TO_ULINT` converts a `REAL` value (floating-point number) to a `ULINT` value (64-bit unsigned integer). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: Input for the `REAL` value to be converted.

Data Outputs

- **OUT**: Output for the converted `ULINT` value.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion as soon as the `REQ` event is triggered. The `REAL` value is converted at input `IN` to a `ULINT` value and output as `OUT`. The conversion is confirmed by the `CNF` event.

Technical Details

- The conversion is performed using the function `REAL_TO_ULINT`.
- This function block is part of the `iec61131::conversion` package.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Conversion State**: Performs the conversion and outputs the result.
3. **Acknowledgement State**: Sends the `CNF` event to confirm the conversion.

Application Scenarios

- Type conversion in control applications where floating-point numbers need to be converted to unsigned integers.
- Data processing in industrial automation systems.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_REAL_TO_INT` or `F_REAL_TO_LINT` perform conversions to other integer types. `F_REAL_TO_ULINT` is specific to 64-bit unsigned integers.

Conclusion

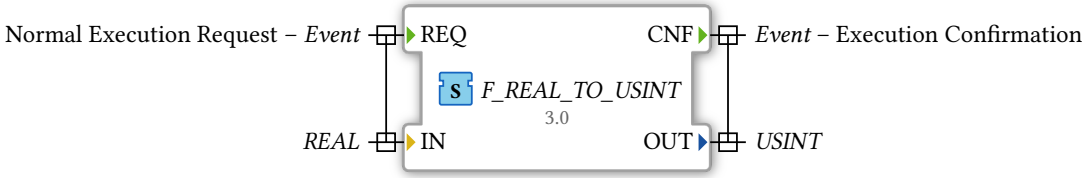
The `F_REAL_TO_ULINT` function block offers a simple and efficient way to convert `REAL` values into `ULINT` values. Its clear interface structure and simple functionality make it a valuable component in industrial automation.

F_REAL_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_REAL_TO_USINT` converts a REAL value (floating-point number) to a USINT value (unsigned 8-bit integer). This conversion is particularly useful when floating-point numbers need to be processed in a range that only accepts integer values.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. Triggered as soon as the converted value is available at data output `OUT`.

Data Inputs

- **IN**: REAL value to be converted to a USINT value.

Data Outputs

- **OUT**: The converted USINT value.

Adapters

This function block has no adapters.

Functionality

The function block performs a direct conversion of the REAL value at input `IN` to a USINT value. The conversion is performed by calling the internal function `REAL_TO_USINT`. Once the conversion is complete, the event `CNF` is triggered, and the converted value is available at output `OUT`.

Technical Features

- The conversion truncates the decimal portion (no rounding).
- The input value should be in the range of 0 to 255, as USINT can only represent values within this range. Values outside this range will result in undefined behavior or an overflow.

State Overview

1. **Idle**: Waits for the event `REQ`.
2. **Processing**: Performs the conversion.
3. **Completed**: Sends the event `CNF` and provides the converted value.

Application Scenarios

- Conversion of sensor values (e.g., temperature, pressure) into integer values for further processing.
- Preprocessing of data for systems that can only process integers.

⚖️ Comparison with Similar Function Blocks

- **F_REAL_TO_INT**: Converts REAL to INT (signed 16-bit integer count).
- **F_REAL_TO_UINT**: Converts REAL to UINT (unsigned 16-bit integer count).
- **F_REAL_TO_ULINT**: Converts REAL to ULINT (unsigned 64-bit integer count).

Conclusion

The function block `F_REAL_TO_USINT` is a simple and efficient tool for converting floating-point numbers to unsigned 8-bit integers. It is particularly suitable for applications where memory or processing resources are limited.

REAL conversions

🎧 Podcast

- IEC 61499: Factory automation in transition - from vision to industrial reality?
- From “Get Away with Electronics” to Essential: How Agricultural Tech Earned Trust and Delivered Real Relief

🛠️ Related exercises

- Uebung_072c
- Uebung_151_AX
- Uebung_152
- Uebung_153

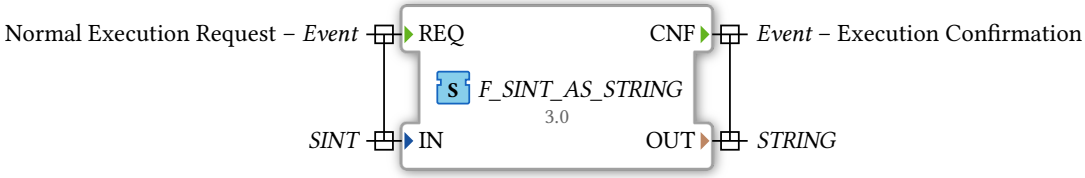
SINT

F_SINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_AS_STRING` converts a `SINT` value (8-bit integer) into a `STRING` (string). This block is particularly useful when numeric values are needed as text for display or further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The `SINT` value to be converted (8-bit integer).

Data Outputs

- **OUT**: The converted `STRING` value.

Adapters

No adapters are available.

Functionality

The function block performs the conversion as soon as the `REQ` event occurs. The `SINT` value at the input `IN` is converted to `STRING` and output at `OUT`. The conversion is performed using the function `SINT_AS_STRING`. After the conversion is complete, the `CNF` event is triggered.

Technical Features

- The conversion is performed directly without additional parameters.
- The function block is simple and efficient, as it only performs a basic data type conversion.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Conversion**: Performs the conversion and outputs the result.
3. **Confirmation**: Triggers the `CNF` event and returns to the idle state.

Application Scenarios

- Displaying numeric values in user interfaces.
- Logging numeric data as text.
- Preparing data for communication via text-based protocols.

⚖️ Comparison with similar building blocks

- **F_INT_AS_STRING**: Converts a `INT` value (16-bit) to a `STRING` value.
- **F_DINT_AS_STRING**: Converts a `DINT` value (32-bit) to a `STRING` value.
- **F_LINT_AS_STRING**: Converts a `LINT` value (64-bit) to a `STRING` value.

Compared to these building blocks, `F_SINT_AS_STRING` specializes in converting 8-bit integers.

Conclusion

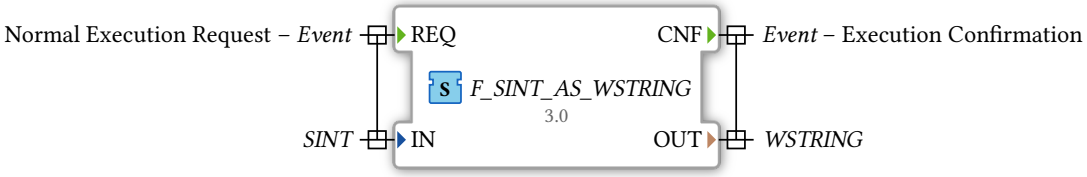
The `F_SINT_AS_STRING` function block is a simple and efficient tool for converting `SINT` values to `STRING`s. Its clear interface and direct functionality make it ideal for applications where small integers need to be represented as text.

F_SINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_AS_WSTRING` is used to convert a SINT value (8-bit signed integer) into a WSTRING (wide string). This block is particularly useful when numeric values need to be converted into a text format suitable for display or further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs

- **IN**: The SINT value (8-bit signed integer) to be converted to a WSTRING.

Data Outputs

- **OUT**: The resulting WSTRING after the SINT value conversion.

Adapters

- No adapters are available.

Functionality

This function block performs the conversion of the SINT value (`IN`) to a WSTRING (`OUT`) as soon as the event `REQ` is triggered. The conversion is performed using the integrated function `SINT_AS_WSTRING`. After successful conversion, the event `CNF` is triggered.

Technical Features

- The function block has a simple structure and contains only one conversion algorithm.
- The conversion is performed directly without additional parameters or complex logic.

State Overview

1. **Idle State**: Waits for the `REQ` event.

2. **Conversion State**: Performs the conversion and outputs the result via `OUT`.
3. **Confirmation State**: Triggers the `CNF` event and returns to the idle state.

Application Scenarios

- Displaying numeric values in user interfaces.
- Logging numeric data in text format.
- Further processing of numeric values in text-based systems.

⚖️ Comparison with similar function blocks

- **F_INT_AS_WSTRING**: Converts an INT value (16-bit) to a WSTRING.
- **F_DINT_AS_WSTRING**: Converts a DINT value (32-bit) to a WSTRING.
- **F_LINT_AS_WSTRING**: Converts a LINT value (64-bit) to a WSTRING.

Conclusion

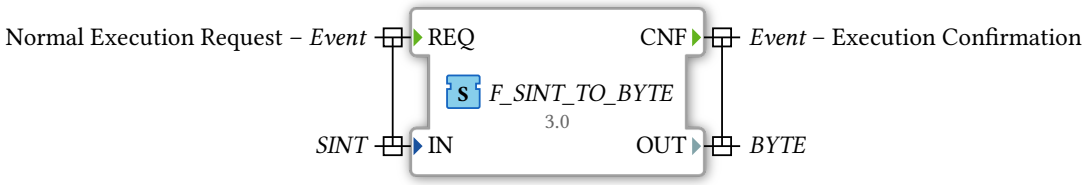
The `F_SINT_AS_WSTRING` function block offers a simple and efficient way to convert SINT values to WSTRINGs. Its clear interface and simple operation make it a useful component in various applications, especially where numeric values need to be represented or processed in text format.

F_SINT_TO_BYTE

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_SINT_TO_BYTE` converts a signed 8-bit integer value (SINT) to an unsigned 8-bit integer value (BYTE). This conversion is useful when exchanging data between systems that use different representations for 8-bit values.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at data output `OUT`.

Data Inputs

- **IN** (SINT): The signed 8-bit integer value to be converted.

Data Outputs

- **OUT** (BYTE): The unsigned 8-bit integer value representing the result of the conversion.

Adapters

- No adapters are available.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The value at input `IN`, of type SINT, is converted to a BYTE value and output at `OUT`. The event `CNF` signals the successful completion of the conversion.

Technical Features

- The conversion is performed directly by the function `SINT_TO_BYTE`.
- The block is simple and deterministic, without internal states or complex logic.

State Overview

The function block has no internal states. The conversion occurs immediately after `REQ` is triggered, followed by the output of the result and confirmation by `CNF`.

Application Scenarios

- Communication between systems that use signed and unsigned 8-bit values.
- Data processing in control systems where type conversion is required.
- Integration into larger function block networks that process different data types.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_SINT_TO_BYTE` specializes in converting SINT to BYTE and is therefore more efficient for this specific use case.
- Other blocks, such as `F_INT_TO_BYTE` or `F_DINT_TO_BYTE`, are designed for larger integer types.

Conclusion

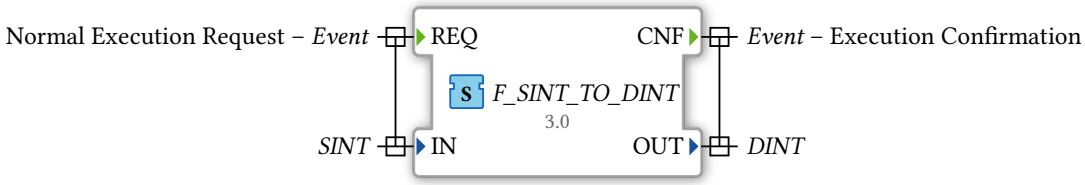
The `F_SINT_TO_BYTE` function block is a simple yet effective tool for converting between signed and unsigned 8-bit integers. Its simplicity and determinism make it reliable for applications that require such type conversions.

F_SINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_DINT` converts a `SINT` value (8-bit signed count) to a `DINT` value (32-bit signed count). This block is particularly useful in scenarios where type conversion is required to prepare data for further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `SINT` to be converted.

Data Outputs

- **OUT**: The converted output value of type `DINT`.

Adapters

This function block has no adapters.

Operation

When the event input `REQ` is triggered, the value at the data input `IN` is read and converted to type `DINT`. The result is output at data output `OUT`, and the event `CNF` signals the successful completion of the conversion.

Technical Features

- The conversion is performed directly by assignment (`OUT := IN`), as `SINT` is automatically converted to `DINT`.

- The function block is simple and efficient because it does not perform any complex calculations.

State Overview

1. **Idle State**: Waits for the event `REQ`.
2. **Conversion State**: Upon receiving `REQ`, the conversion is performed, and `CNF` is triggered.

Application Scenarios

- Type conversion in control applications where `SINT` data needs to be converted to `DINT`.
- Preparing data for further processing steps that require `DINT`.

⚖ Comparison with Similar Function Blocks

- Similar function blocks such as `F_INT_TO_DINT` or `F_USINT_TO_UDINT` perform similar conversions, but with different output and input types, respectively.
- `F_SINT_TO_DINT` is specifically for converting `SINT` to `DINT`.

Conclusion

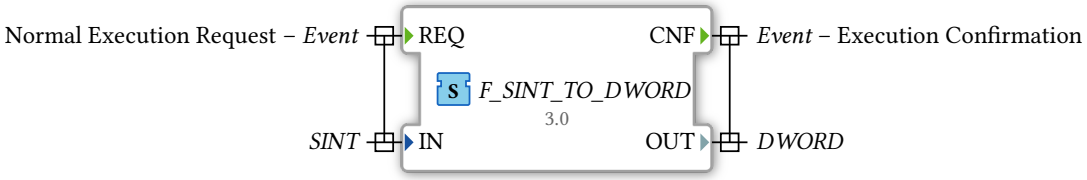
The `F_SINT_TO_DINT` function block is a simple and efficient building block for type conversion from `SINT` to `DINT`. Its clear interface structure and simple functionality make it a reliable component in control applications.

F_SINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_DWORD` converts a signed 8-bit integer value (SINT) to an unsigned 32-bit integer value (DWORD). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `SINT` to be converted.

Data Outputs

- **OUT**: The output value of type `DWORD`, containing the result of the conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The algorithm converts the `SINT` value at input `IN` into a `DWORD` value and stores the result at output `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

Technical Features

- The conversion is performed directly by the function `SINT_TO_DWORD(IN)`.

- The function block is implemented as a simple function block (SimpleFB), enabling fast and efficient execution.

State Overview

1. **Idle State**: Waiting for the `REQ` event.
2. **Conversion State**: Performs the conversion and sets the output `OUT`.
3. **Acknowledge State**: Triggers the `CNF` event and returns to the idle state.

Application Scenarios

- Type conversion in control applications where `SINT` values need to be converted to `DWORD` values.
- Integration into larger control systems that process different data types.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_INT_TO_DINT` or `F_BYTE_TO_WORD` offer conversions between other data types, but not specifically between `SINT` and `DWORD`.
- Unlike generic conversion blocks, `F_SINT_TO_DWORD` is specialized and therefore more efficient for this specific use case.

Conclusion

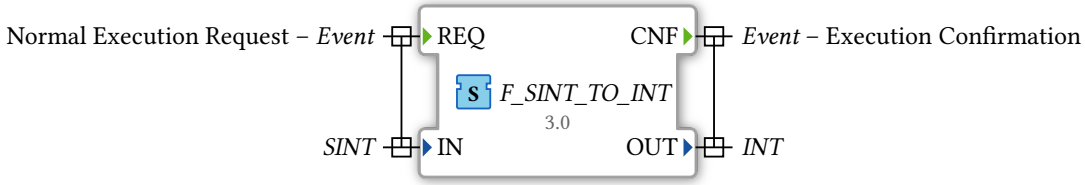
The `F_SINT_TO_DWORD` function block is a specialized and efficient tool for converting `SINT` values to `DWORD` values. Its simple interface and direct operation make it ideal for applications that require fast and reliable type conversion.

F_SINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_INT` converts a `SINT` value (8-bit signed integer) to a `INT` value (16-bit signed integer count). This block is particularly useful in scenarios where type conversion between different integer formats is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (`SINT`): The input value to be converted.

Data Outputs

- **OUT** (`INT`): The converted output value.

Adapters

- No adapters are available.

Functionality

This function block directly assigns the `SINT` input value to the `INT` output. Since `SINT` is a smaller data type than `INT`, the conversion is performed without data loss.

The algorithm is defined as follows:

ALGORITHM REQ OUT := IN; END_ALGORITHM

Technical Features

- The conversion is lossless, as the value range of `SINT` (-128 to 127) fits completely within the value range of `INT` (-32768 to 32767).
- The function block is deterministic and has no internal states.

State Overview

The function block has no internal states. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control applications where various integer formats need to be processed.
- Integration into larger function block networks where uniform data type processing is required.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_INT_TO_SINT` perform the reverse conversion, but with potential data loss.
- Unlike generic conversion blocks, `F_SINT_TO_INT` is specialized and optimized for this specific conversion.

Conclusion

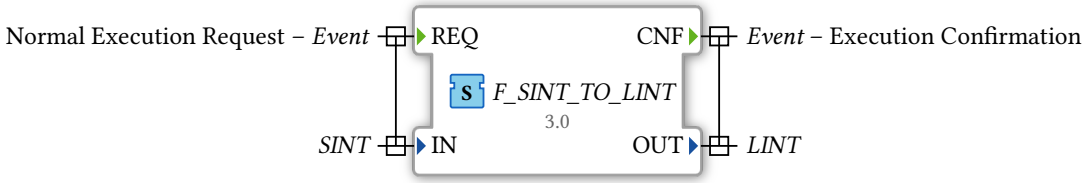
The `F_SINT_TO_INT` function block offers a simple and efficient way to convert `SINT` values to `INT` values. Its deterministic and lossless operation makes it a reliable component in control applications.

F_SINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_LINT` converts a `SINT` value (8-bit integer) to a `LINT` value (64-bit integer). This conversion is useful when processing data with different bit widths.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- `IN` (`SINT`): The 8-bit integer value to be converted.

Data Outputs

- `OUT` (`LINT`): The converted 64-bit integer value.

Adapters

No adapters are available.

Functionality

The function block performs a direct assignment of the input value `IN` to the output value `OUT`. Since `LINT` has a larger bit width than `SINT`, the conversion is lossless.

Technical Features

- The conversion is lossless because the `LINT` data type can represent all possible values of the `SINT` data type.
- The function block is deterministic and has no internal state.

State Overview

The function block has no internal state. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Processing 8-bit sensor data in 64-bit systems
- Compatibility between systems with different bit widths
- Data conversion in automation systems

⚖️ Comparison with similar modules

- Compared to other conversion modules (e.g., `F_INT_TO_LINT`), this module specifically works with 8-bit input data.
- Similar to `F_BYTE_TO_LWORD`, but for signed integers.

Conclusion

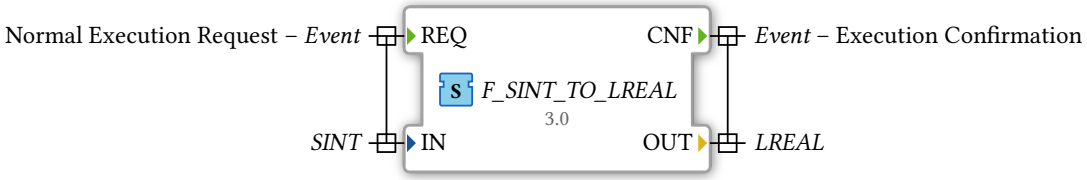
The `F_SINT_TO_LINT` function module offers a simple and efficient way to convert 8-bit integers to 64-bit integers. Its simple structure and deterministic operation make it a reliable module in automation solutions.

F_SINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_LREAL` converts an integer value of type `SINT` (8-bit integer) to a floating-point value of type `LREAL` (64-bit floating-point). This block is particularly useful in applications where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: Input for the `SINT` value to be converted.

Data Outputs

- **OUT**: Output for the converted `LREAL` value.

Adapters

No adapters are available.

Functionality

The function block performs a direct type conversion. Upon the arrival of a `REQ` event, the value of `IN` is read, converted to the type `LREAL`, and output to `OUT`. Subsequently, the `CNF` event is triggered.

Technical Features

- The conversion is performed without any additional scaling or rounding.
- The function block is implemented simply and efficiently, as it requires only one assignment.

State Overview

The function block has no internal states. It reacts exclusively to the `REQ` event with an immediate conversion and output.

Application Scenarios

- Type conversion in control applications where `SINT` values need to be used in calculations with `LREAL` values.
- Data preparation for visualization or logging where higher accuracy is required.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_INT_TO_REAL` or `F_DINT_TO_LREAL` offer conversions between other data types, but not specifically from `SINT` to `LREAL`.
- This function block specializes in converting 8-bit integers to 64-bit floating-point numbers.

Conclusion

The `F_SINT_TO_LREAL` function block is a simple and efficient tool for type conversion in the 4diac IDE. Its clear interface and direct functionality make it a reliable component in control applications.

🌐 Related topic subpages on ms-muc-docs.de

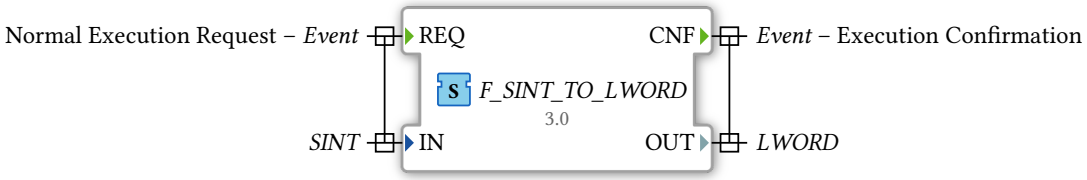
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

F_SINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_LWORD` converts a signed 8-bit integer value (`SINT`) to an unsigned 64-bit integer value (`LWORD`). This block is part of the `iec61131::conversion` package and is typically used in automation applications where data type conversions are required.



Interface Structure

Event Inputs

- **REQ** (Normal Execution Request): Starts the execution of the function block. Linked to data input `IN`.

Event Outputs

- **CNF** (Execution Confirmation): Signals completion of the conversion. Linked to data output `OUT`.

Data Inputs

- **IN** (`SINT`): The signed 8-bit integer value to be converted.

Data Outputs

- **OUT** (`LWORD`): The resulting unsigned 64-bit integer value after conversion.

Adapters

This function block does not use adapters.

Adapters ## Functionality

This function block performs the conversion from `SINT` to `LWORD` as soon as the event `REQ` is triggered. The conversion is carried out using the integrated function `SINT_TO_LWORD`. After successful conversion, the event `CNF` is triggered, and the converted value is available at output `OUT`.

Technical Features

- The conversion is performed without additional parameters or configuration.
- The function block is deterministic and has no internal states.

State Overview

Since this is a simple function block, there are no state transitions or internal states. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control applications where different data types need to be processed.
- Integration into larger function block networks where data needs to be converted from a lower to a higher precision format.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_INT_TO_LWORD` or `F_DINT_TO_LWORD` perform conversions from other integer types to `LWORD`. `F_SINT_TO_LWORD` is specifically for converting 8-bit integers.
- Unlike generic conversion blocks, this function block offers a type-safe and optimized solution for this specific conversion.

Conclusion

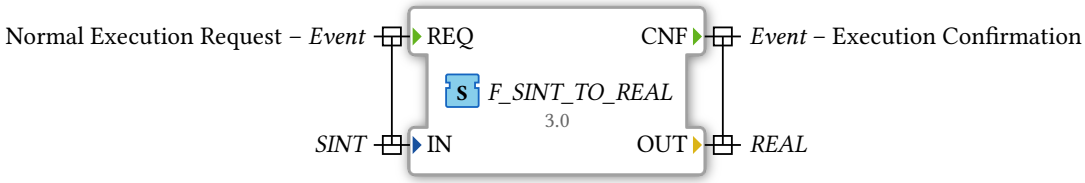
The `F_SINT_TO_LWORD` function block offers an efficient and simple way to convert `SINT` values to `LWORD` values. Its deterministic operation and clear interfaces make it ideal for use in automation projects where such type conversions are regularly required.

F_SINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_REAL` converts a signed 8-bit integer value (`SINT`) into a floating-point value (`REAL`). This conversion is particularly useful in control applications where different data types need to be processed.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (`SINT`): The signed 8-bit integer value to be converted.

Data Outputs

- **OUT** (`REAL`): The resulting floating-point value after conversion.

Adapters

No adapters available.

Functionality

The function block performs a direct type conversion. When an event arrives at input `REQ`, the value of `IN` is converted to the data type `REAL` and output to `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

Technical Features

- **Simple Conversion**: The conversion occurs without additional calculations or scaling.
- **No Error Handling**: There is no special handling for overflow or underflow conditions.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- **Measurement Processing**: Conversion of sensor data, which is available as `SINT`, to `REAL` for further calculations.
- **Control Algorithms**: Integration into control loops where floating-point values are required.

⚖️ Comparison with Similar Function Blocks

- **F_INT_TO_REAL**: Converts a 16-bit integer (`INT`) to `REAL`.
- **F_DINT_TO_REAL**: Converts a 32-bit integer (`DINT`) to `REAL`.
- **F_SINT_TO_REAL** is specific to 8-bit integers, providing a more compact solution for this data type.

Conclusion

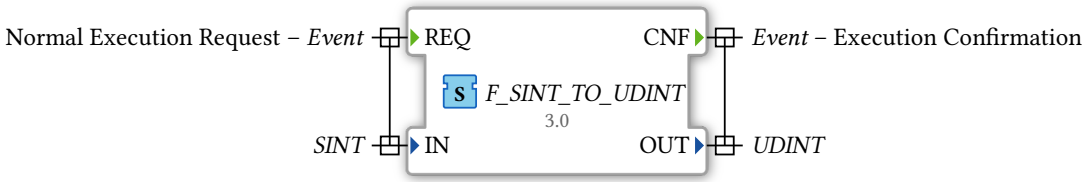
The `F_SINT_TO_REAL` function block is a simple and efficient tool for converting 8-bit integer values to floating-point numbers. Its clear interface and direct functionality make it ideal for applications that require fast and straightforward type conversion.

F_SINT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_UDINT` converts a signed 8-bit integer value (SINT) to an unsigned 32-bit integer value (UDINT). This conversion is particularly useful when exchanging data between systems that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `SINT` to be converted.

Data Outputs

- **OUT**: The output value of type `UDINT`, containing the result of the conversion.

Adapters

- No adapters are available.

Functionality

The function block performs the conversion by transforming the `SINT` value at input `IN` into a `UDINT` value and providing the result at output `OUT`. The conversion is triggered by an event at input `REQ` and confirmed by an event at output `CNF`.

Technical Details

- The conversion is performed using the function `SINT_TO_UDINT`.
- This function block is part of the package `iec61131::conversion`.

State Overview

1. **Idle State**: Waits for an event at input `REQ`.
2. **Active State**: Executes the conversion and signals its completion with an event at output `CNF`.

Application Scenarios

- Conversion of sensor values stored as `SINT` to a `UDINT` value for further processing.
- Data preparation for systems that expect unsigned integer values.

⚖️ Comparison with Similar Function Blocks

- **F_INT_TO_UDINT**: Converts a signed 16-bit integer (`INT`) to an unsigned 32-bit integer (`UDINT`).
- **F_DINT_TO_UDINT**: Converts a signed 32-bit integer (`DINT`) to an unsigned 32-bit integer (`UDINT`).

Conclusion

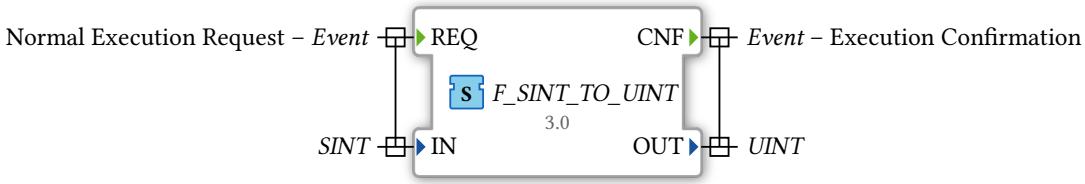
The function block `F_SINT_TO_UDINT` provides a simple and efficient way to convert `SINT` values to `UDINT` values. Its clear interface structure and simple operation make it ideal for applications requiring such conversion.

F_SINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_UINT` converts a signed 8-bit integer value (`SINT`) into an unsigned 16-bit integer value (`UINT`). This conversion is particularly necessary when exchanging data between systems that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs

- **IN**: The input expects a value of type `SINT` (signed 8-bit integer).

Data Outputs

- **OUT**: The output returns the converted value as `UINT` (unsigned 16-bit integer).

Adapters

- No adapters are present.

Operation

The function block performs the conversion as soon as the event `REQ` is received. The input value `IN`, of type `SINT`, is converted to a value of `UINT` and output `OUT`. Successful conversion is signaled by the event `CNF`.

Technical Features

- The conversion is performed using the function `SINT_TO_UINT`, which is implemented in the function block's algorithm.
- The function block is simple and requires no additional state management.

State Overview

Since this is a simple function block, there are no complex states. The block reacts directly to the `REQ` event by executing the conversion and outputting the result.

Application Scenarios

- Conversion of sensor values stored as `SINT` for systems that expect `UINT`.
- Data preparation for communication protocols that require unsigned values.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_SINT_TO_UINT` is specialized and optimized for converting `SINT` to `UINT`.
- Other blocks, such as `F_INT_TO_UINT` or `F_DINT_TO_UDINT`, offer similar functionality, but for different data types.

🛠️ Related exercises

- Uebung_035
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- Uebung_039_sub_NumbAnsicht
- Uebung_040
- Exercise_040_2
- Exercise_040_AX
- Exercise_041

Conclusion

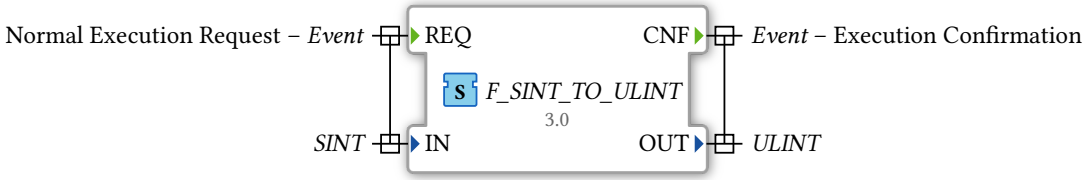
The `F_SINT_TO_UINT` function block is an efficient tool for the specific conversion of signed to unsigned integer values. Its simplicity and direct operation make it a reliable solution in control and automation systems.

F_SINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_ULINT` converts a signed 8-bit integer value (`SINT`) to an unsigned 64-bit integer value (`ULINT`). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `SINT` to be converted.

Data Outputs

- **OUT**: The converted output value of type `ULINT`.

Adapters

No adapters are present.

Operation

The function block performs the conversion when the event `REQ` is received. The input value `IN`, of type `SINT`, is converted to a value of `ULINT` and output `OUT`. Successful conversion is indicated by the event `CNF`.

Technical Features

- The conversion is performed by the built-in function `SINT_TO_ULINT`.
- The function block is simple and deterministic, as it does not store any internal states.

State Overview

The function block has no internal state. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control applications where `SINT` values need to be converted to larger unsigned integer types.
- Data processing in systems that work with different integer formats.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_INT_TO_ULINT` or `F_DINT_TO_ULINT` perform similar conversions, but with different input types.
- `F_SINT_TO_ULINT` is specifically optimized for converting `SINT` to `ULINT`.

Conclusion

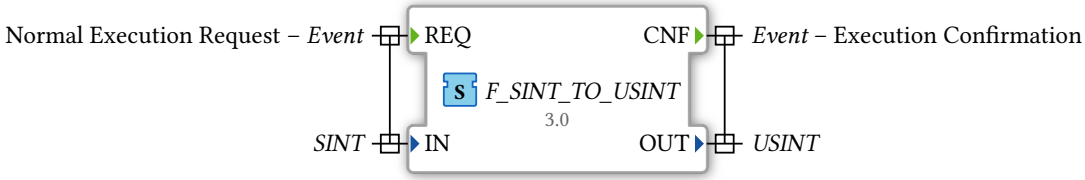
The `F_SINT_TO_ULINT` function block provides a simple and efficient way to convert `SINT` values to `ULINT`. It is particularly useful in applications where safe and fast type conversion is required.

F_SINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_USINT` converts a signed 8-bit integer value (SINT) to an unsigned 8-bit integer value (USINT). This conversion is useful when exchanging data between systems that use different representations for numeric values.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN** (SINT): The signed 8-bit integer value to be converted.

Data Outputs

- **OUT** (USINT): The resulting unsigned 8-bit integer value after the conversion.

Adapters

This function block does not use any adapters.

Operation

When the event `REQ` is triggered, the algorithm that converts the value of `IN` from SINT to USINT is executed and outputs the result to `OUT`. The event `CNF` is then triggered to signal the successful completion of the conversion.

Technical Features

- The function block performs a direct type conversion, converting the signed value to an unsigned value.
- There is no additional logic for handling overflow conditions or invalid inputs.

State Overview

The function block has a simple state machine:

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion and triggers `CNF`.
3. No other states, as it is a simple, stateless block.

Application Scenarios

- Conversion of sensor values that are delivered as signed values but are to be processed as unsigned values.
- Data preparation for communication protocols that require unsigned values.

⚖️ Comparison with Similar Function Blocks

- **F_INT_TO_UINT**: Converts signed 16-bit integer values to unsigned 16-bit integer values.
- **F_DINT_TO_UDINT**: Converts signed 32-bit integer values to unsigned 32-bit integer values.
- **F_SINT_TO_USINT** is specific to 8-bit values and is therefore more efficient for this data size.

Conclusion

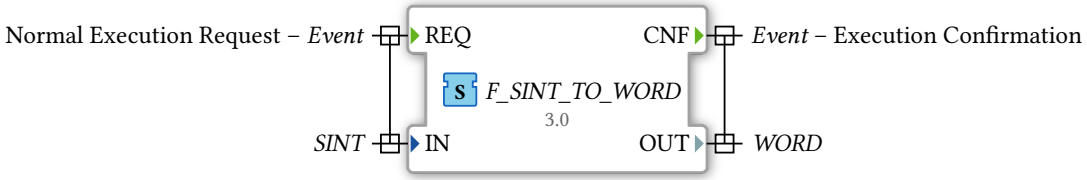
The `F_SINT_TO_USINT` function block is a simple yet effective tool for converting signed to unsigned 8-bit integer values. Its simplicity and efficiency make it ideal for applications where such conversions need to be performed regularly.

F_SINT_TO_WORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SINT_TO_WORD` converts a SINT value (8-bit signed integer count) to a WORD value (16-bit unsigned integer count). This block is particularly useful in scenarios where type conversion between different integer formats is required.



Interface Structure

Event Inputs

- `REQ` (Normal Execution Request): Starts the execution of the conversion. It is associated with the input value `IN`.

Event Outputs

- `CNF` (Execution Confirmation): Signals the completion of the conversion. Returns the converted value via `OUT`.

Data Inputs

- `IN` (SINT): The input value to be converted from SINT to WORD.

Data Outputs

- `OUT` (WORD): The converted WORD value.

Adapters

This function block does not use adapters.

Operation

When the `REQ` event is triggered, the algorithm that converts the `IN` value from SINT to WORD is executed. The result is output via `OUT`, and the `CNF` event signals successful conversion.

Technical Features

- The conversion is performed directly by the function `SINT_TO_WORD(IN)`.
- No additional error handling is required for overflow or underflow, as SINT (8-bit) is converted to WORD (16-bit), and therefore no values are lost.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering `REQ`.

Application Scenarios

- Type conversion in control applications where different integer formats need to be processed.
- Data preprocessing for communication protocols or sensor data.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_INT_TO_WORD` or `F_DINT_TO_WORD` offer conversions from other integer types to WORD.
- `F_SINT_TO_WORD` is specifically optimized for converting 8-bit signed integers to 16-bit unsigned integers.

Conclusion

The `F_SINT_TO_WORD` function block is a simple and efficient building block for type conversion from SINT to WORD. Its clear interface and direct functionality make it ideal for applications requiring fast and reliable conversion.

SINT Conversions

🎧 Podcast

- SINT, INT, DINT: Why the choice of data type determines efficiency and errors

🛠️ Related Exercises

- Exercise_039_sub_NumbDisplay

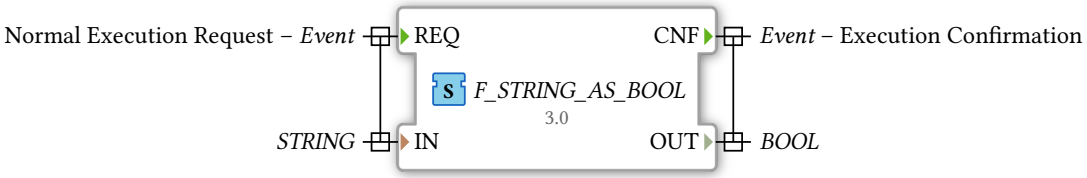
STRING

F_STRING_AS_BOOL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_BOOL` converts a `STRING` value to a `BOOL` value. This block is particularly useful when strings need to be converted into Boolean values, for example, when processing user input or interpreting text data.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This input triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. This output signals the successful completion of the conversion.

Data Inputs

- **IN** (`STRING`): The input string to be converted into a Boolean value.

Data Outputs

- **OUT** (`BOOL`): The resulting Boolean value after conversion.

Adapters

No adapters are present.

Functionality

The function block uses the function `STRING_AS_BOOL` to convert the input string `IN` into a Boolean value `OUT`. The conversion is triggered by the event `REQ`, and the event `CNF` is output upon completion of the operation.

Technical Features

- The conversion is performed according to the IEC 61131-3 specification for converting strings to Boolean values.
- The function block is part of the `iec61131::conversion` package.

State Overview

1. **Initialization**: The function block waits for the `REQ` event.
2. **Execution**: Upon receiving `REQ`, the string `IN` is converted to a Boolean value, and the result is output to `OUT`.
3. **Acknowledgement**: After successful conversion, the `CNF` event is output.

Application Scenarios

- Processing user input in string form (e.g., “true” or “false”).
- Interpreting text data from external sources (e.g., configuration files or network communication).
- Integration into systems that store or transmit Boolean values as strings.

⚖️ Comparison with Similar Function Blocks

- **F_STRING_TO_BOOL**: A similar function block that may offer additional options for string interpretation.
- **F_STRING_AS_***: Other conversion function blocks for various data types (e.g., `F_STRING_AS_INT`).

Conclusion

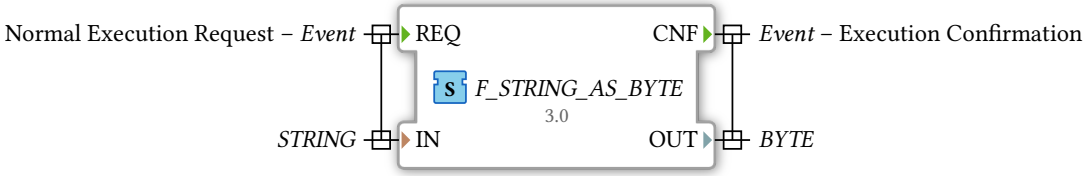
The `F_STRING_AS_BOOL` function block provides a simple and efficient way to convert string values to Boolean values. Its clear interface structure and reliability make it ideal for applications requiring such conversion.

F_STRING_AS_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_BYTE` converts a `STRING` value to a `BYTE` value. This functionality is particularly useful when strings need to be converted into their corresponding byte values, for example, for processing in protocols or when communicating with hardware.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This event input triggers the conversion. It is linked to the data input `IN`.

Event Outputs

- **CNF**: Execution confirmation. This event is triggered after successful conversion and is associated with the data output `OUT`.

Data Inputs

- **IN**: Input of type `STRING`, which is to be converted to a `BYTE` value.

Data Outputs

- **OUT**: Output of type `BYTE`, containing the result of the conversion.

Adapters

No adapters are present.

Functionality

The function block converts the `STRING` value `IN` to the `BYTE` value `OUT` as soon as the event `REQ` is received. The conversion is performed using the function `STRING_AS_BYTE`. After successful conversion, the event `CNF` is triggered.

Technical Features

- The function block is a simple function block (`SimpleFB`) and uses an ST algorithm for the conversion.
- The conversion occurs directly and without any additional delay.

State Overview

1. **Wait State**: The function block waits for the event `REQ`.
2. **Execution State**: Upon receiving `REQ`, the conversion is performed.
3. **Acknowledgement State**: After successful conversion, `CNF` is triggered, and the function block returns to the wait state.

Application Scenarios

- Conversion of strings to byte values for communication with serial interfaces.
- Processing of ASCII characters in byte form for protocols or hardware controls.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks like `F_STRING_TO_INT` or `F_STRING_TO_REAL`, `F_STRING_AS_BYTE` is specifically designed to convert to a `BYTE` value.
- A similar block could be `F_CHAR_AS_BYTE`, which converts individual characters, while `F_STRING_AS_BYTE` works with strings.

Conclusion

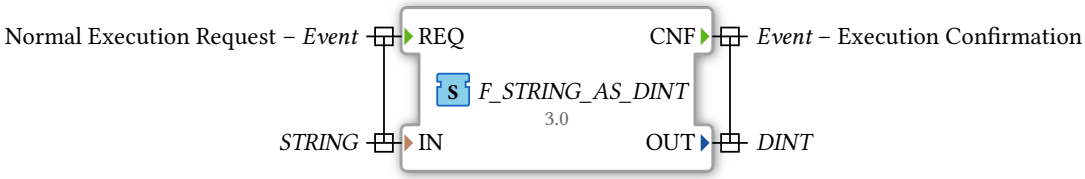
The `F_STRING_AS_BYTE` function block offers a simple and efficient way to convert strings to byte values. Its clear interface structure and direct functionality make it ideal for applications requiring such conversion.

F_STRING_AS_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_DINT` converts a `STRING` value to a `DINT` value. This function is particularly useful when strings need to be converted into numeric values, for example, when processing user input or interpreting text data.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (`STRING`): The string value to be converted.

Data Outputs

- **OUT** (`DINT`): The result of the conversion as a 32-bit integer.

Adapters

No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The `STRING` value at input `IN` is converted into a `DINT` value and output at `OUT`. The conversion is performed using the function `STRING_AS_DINT`. Upon completion of the conversion, the event `CNF` is triggered.

Technical Details

- The conversion can fail if the value of `STRING` does not contain a valid numeric representation. In such cases, the output `OUT` may be undefined.
- The function block is part of the package `iec61131::conversion`.

State Overview

The function block has no internal state. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Processing user input in the form of strings.
- Interpreting text data from files or network sources.
- Conversion of configurable parameters stored as strings.

⚖️ Comparison with similar function blocks

- Unlike `F_STRING_TO_*` function blocks, which perform explicit type conversions, `F_STRING_AS_DINT` directly interprets the `STRING` value as `DINT`.
- Similar function blocks like `F_STRING_TO_DINT` might offer additional formatting parameters, but they are not included in the standard library.

Conclusion

The `F_STRING_AS_DINT` function block provides a simple and efficient way to convert strings to 32-bit integers. Its use is particularly useful in scenarios where direct interpretation of the string as a numeric value is required.

F_STRING_AS_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_DWORD` converts a `STRING` value to a `DWORD` value. This functionality is particularly useful when strings need to be converted into numeric values, for example, for further processing in control algorithms.



Interface Structure

Event Inputs

- `REQ`: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- `IN` (`STRING`): The input value to be converted to a `DWORD` value.

Data Outputs

- `OUT` (`DWORD`): The result of the conversion.

Adapters

- No adapters available.

Functionality

The function block converts the input value `STRING`, `IN`, into a value `DWORD` as soon as the event `REQ` is triggered. The conversion is performed using the function `STRING_AS_DWORD`. After successful conversion, the event `CNF` is triggered, and the result value is available at the output `OUT`.

Technical Features

- The function block is a simple function block (`SimpleFB`) and performs the conversion in a single algorithm.
- The conversion is performed directly without additional state management.

State Overview

The function block has no internal states. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Conversion of user input or communication data into numeric values.
- Use in control algorithms that require numeric values instead of strings.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion function blocks such as `F_STRING_TO_INT` or `F_STRING_TO_REAL`, this function block is specifically designed for conversion to the `DWORD` data type.
- Direct use of the `STRING_AS_DWORD` function enables efficient and specialized conversion.

Conclusion

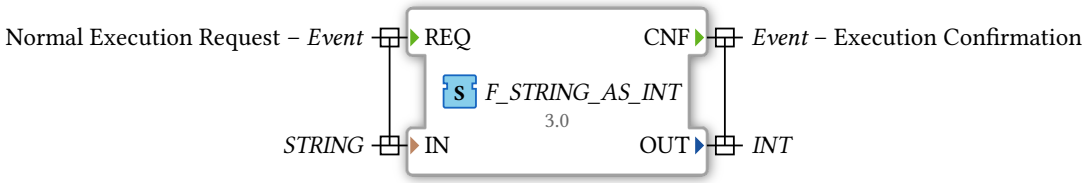
The `F_STRING_AS_DWORD` function block offers a simple and efficient way to convert strings into `DWORD` values. Its clear interface structure and direct functionality make it ideal for applications requiring fast and reliable conversion.

F_STRING_AS_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_INT` converts a `STRING` value to a `INT` value. This functionality is particularly useful when strings representing numeric values need to be converted into integer values.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event input is triggered, the `STRING` value at the data input `IN` is converted to a `INT` value.

Event Outputs

- **CNF**: Signals successful completion of the conversion. This event, along with the converted `INT` value, is output at data output `OUT`.

Data Inputs

- **IN**: The `STRING` value to be converted to a `INT` value.

Data Outputs

- **OUT**: The converted `INT` value.

Adapters

No adapters are present.

Operation

The function block performs the conversion as soon as the REQ event is received. The built-in function STRING_AS_INT is used to convert the STRING value at input IN into a INT value. The result is output at OUT and confirmed by the CNF event.

Technical Features

- The conversion is performed directly without additional parameters.
- The function block is implemented simply and efficiently, without complex state logic.

State Overview

The function block has no internal states. The conversion occurs immediately upon receiving the REQ event.

Application Scenarios

- Processing user input that is in string form but requires numeric values.
- Conversion of data from text files or communication protocols into numeric values.
- Integration into systems that use string-based data sources but require numerical processing.

⚖️ Comparison with similar building blocks

- Compared to generic conversion building blocks, F_STRING_AS_INT specializes in converting STRING to INT.
- Other building blocks might offer additional features such as error handling or support for various numeric formats, which is not the case here.

Conclusion

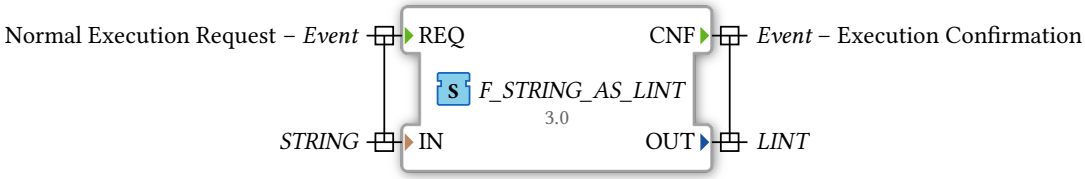
The F_STRING_AS_INT function block is a simple and effective tool for converting strings to integer values. Its strength lies in its simplicity and direct applicability, without added complexity. This building block is ideal for applications that require fast and straightforward conversion.

F_STRING_AS_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_STRING_AS_LINT converts a STRING value to a LINT value. This functionality is particularly useful when strings need to be converted into numeric values for use in further calculations or control logic.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion of the STRING value to a LINT value. This event input is linked to the data input IN.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output OUT.

Data Inputs

- **IN**: The input for the STRING value, which is to be converted to a LINT value.

Data Outputs

- **OUT**: The output for the converted LINT value.

Adapters

No adapters are available.

Functionality

The function block converts the `STRING` value to a `LINT` value as soon as the `REQ` event is triggered. The conversion is performed using the function `STRING_AS_LINT()`. After successful conversion, the `CNF` event is triggered, and the converted value is available at the `OUT` output.

Technical Features

- The function block is a simple function block (SimpleFB) and performs the conversion in a single algorithm.
- The conversion is performed directly without additional state management.

State Overview

Because it is a simple function block, there are no complex state transitions. The block performs the conversion on each `REQ` event and signals completion with the `CNF` event.

Application Scenarios

- Conversion of user input to numeric values.
- Processing of strings containing numeric data in control systems.
- Integration into larger control logics that require numeric calculations.

⚖️ Comparison with Similar Blocks

- **F_STRING_AS_INT**: Converts a `STRING` value to a `INT` value, but is limited to smaller numeric ranges.
- **F_STRING_AS_REAL**: Converts a `STRING` value to a `REAL` value; suitable for floating-point numbers.
- **F_LINT_AS_STRING**: Performs the reverse conversion, i.e., from `LINT` to `STRING`.

Conclusion

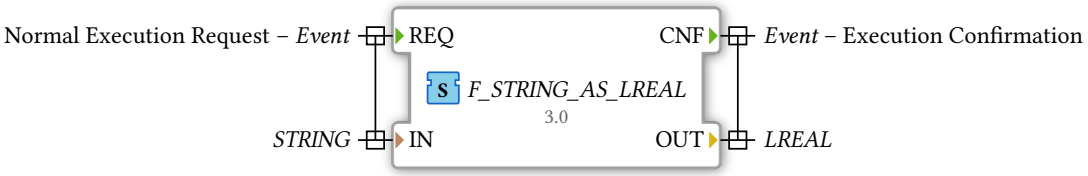
The function block `F_STRING_AS_LINT` offers a simple and efficient way to convert strings into large numeric values (`LINT`). Its direct functionality and clear interface structure make it ideal for use in control systems that require such conversion.

F_STRING_AS_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_LREAL` converts a STRING value to an LREAL value (64-bit floating-point number). This block is particularly useful when strings need to be converted into numeric values, for example, when processing user input or reading data from external sources.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- `IN` (STRING): The string value to be converted.

Data Outputs

- `OUT` (LREAL): The result of the conversion as a 64-bit floating-point number.

Adapters

This function block does not use any adapters.

Operation

When the `REQ` event is triggered, the value at the input `IN` is read as a string and converted into an LREAL value. The result is output as `OUT`, and the `CNF` event signals successful execution.

The conversion algorithm is implemented as ST code:
ALGORITHM REQ OUT := STRING_AS_LREAL(IN); END_ALGORITHM

Technical Features

- The conversion follows the standard rules for converting STRING to LREAL in IEC 61131-3.
- For invalid input values (e.g., non-numeric strings), the behavior may vary depending on the runtime environment.

State Overview

The function block has a simple state machine:

1. Waits for the `REQ` event
2. Performs the conversion
3. Sends the `CNF` event
4. Returns to wait state

Application Scenarios

- Processing user input in numeric controllers
- Converting text data from files or network sources
- Interfaces to systems that transmit numeric data as strings

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, this function block specializes in STRING-to-LREAL conversion.
- Other similar blocks might offer additional parameters for number formatting.

Conclusion

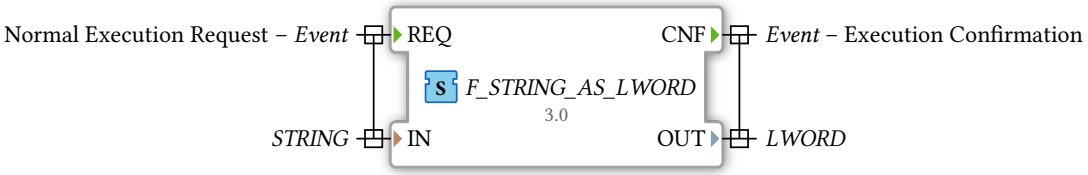
The `F_STRING_AS_LREAL` function block offers a simple and efficient way to convert strings to 64-bit floating-point numbers. Its clear interface and simple functionality make it particularly suitable for standard conversion tasks in automation projects.

F_STRING_AS_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_LWORD` converts a `STRING` value to a `LWORD` value. It is part of the `iec61131::conversion` package and enables easy conversion between these two data types.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (`STRING`): The input string to be converted to a `LWORD` value.

Data Outputs

- **OUT** (`LWORD`): The converted `LWORD` value.

Adapters

No adapters are present.

Functionality

The function block performs the conversion when the `REQ` event is received. The input string `IN` is converted into a value `LWORD` using the function `STRING_AS_LWORD` and output as `OUT`. After the conversion is complete, the event `CNF` is triggered.

Technical Features

- The conversion is performed directly without additional parameters.
- The block is implemented as a simple function block (`SimpleFB`) containing only one algorithm.

State Overview

The function block has no internal state. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Conversion of string data into `LWORD` values for further processing in control systems.
- Integration into systems that require `LWORD` values but receive string input.

⚖️ Comparison with Similar Building Blocks

- Compared to other conversion building blocks, `F_STRING_AS_LWORD` specializes in converting `STRING` to `LWORD`.
- Other building blocks, such as `F_STRING_TO_*`, may offer similar functionality for other data types.

Conclusion

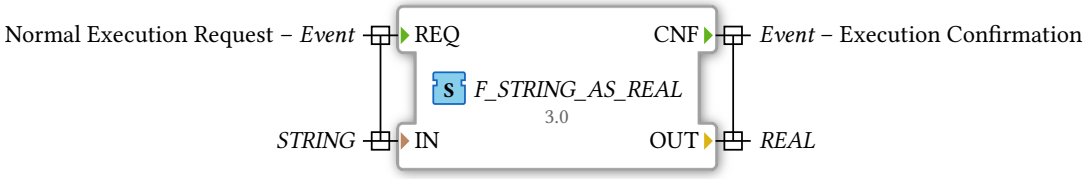
The `F_STRING_AS_LWORD` function block provides a simple and efficient way to convert string data into `LWORD` values. It is particularly useful in scenarios where such conversions are needed regularly.

F_STRING_AS_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_REAL` converts a `STRING` value to a `REAL` value. It is particularly useful when data from a string needs to be converted into a numerical representation, for example, when processing user input or interpreting text files.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event is triggered, the algorithm for converting the `STRING` value to a `REAL` value is executed.

Event Outputs

- **CNF**: Signals successful completion of the conversion. This event is output along with the converted `REAL` value.

Data Inputs

- **IN** (`STRING`): The input value to be converted to a `REAL` value.

Data Outputs

- **OUT** (`REAL`): The result of the conversion.

Adapters

This function block does not use adapters.

Functionality

When the `REQ` event is triggered, the function block converts the `STRING` value (`IN`) into a `REAL` value. The conversion is performed using the built-in function `STRING_AS_REAL`. After successful conversion, the `CNF` event is triggered, and the result is output via `OUT`.

Technical Features

- The conversion can fail if the input `STRING` is not a valid `REAL` number. In this case, a runtime error may occur.
- The function block is simple and does not include any additional error handling.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Completed**: Sends the `CNF` event with the result.

Application Scenarios

- Processing user input in string form.
- Interpreting data from text files or network communication.
- Converting configuration values stored as strings.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_STRING_AS_REAL` specializes in converting `STRING` to `REAL`.
- Other blocks might offer additional features such as error handling or formatting, but they are more complex.

Conclusion

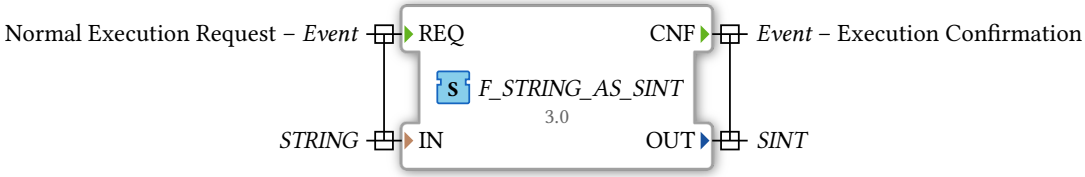
The `F_STRING_AS_REAL` function block is a simple and efficient tool for converting strings to floating-point numbers. Its strength lies in its simplicity and direct applicability, although it lacks integrated error handling. For robust applications, additional security measures should be implemented.

F_STRING_AS_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_SINT` converts a `STRING` value into a `SINT` value (8-bit signed integer). This block is particularly useful when strings need to be converted into numeric values, for example, when processing user input or interpreting text data.



Interface Structure

Event Inputs

- `REQ`: Triggers the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- `IN` (`STRING`): The string to be converted into a `SINT` number.

Data Outputs

- `OUT` (`SINT`): The result of the conversion as an 8-bit signed integer.

Adapters

This function block does not use any adapters.

Functionality

When the `REQ` event is triggered, the function `STRING_AS_SINT` is applied to the value of the `IN` input. The result is stored in the `OUT` output, and the `CNF` event signals the successful completion of the operation.

Technical Features

- The conversion is performed according to the IEC 61131-3 specification for converting strings to numeric values.
- The function block is part of the `iec61131::conversion` package.

State Overview

1. **Idle**: Waiting for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Completed**: Signals completion with `CNF` and returns to the idle state.

Application Scenarios

- Interpreting user input into numerical control commands.
- Processing text data from external sources (e.g., files or network communication).
- Converting configuration values into control parameters.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_STRING_AS_SINT` specializes in conversion to 8-bit integers.
- Similar blocks like `F_STRING_AS_INT` or `F_STRING_AS_DINT` offer conversions to larger integer types.

Conclusion

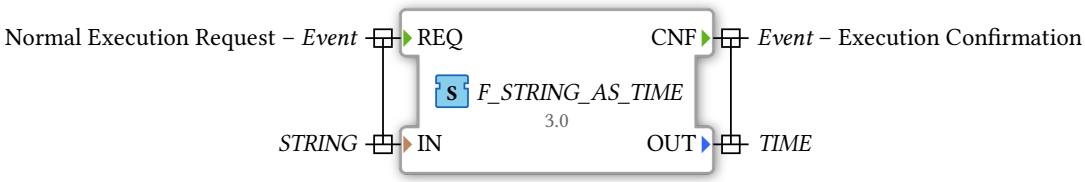
F_STRING_AS_SINT` is an efficient and specialized function block for converting strings to 8-bit integers. Its simple interface and clear functionality make it ideal for control applications that work with text-based input.

F_STRING_AS_TIME

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_STRING_AS_TIME` converts a time value in STRING format to TIME format. This is particularly useful when time values are in text format and need to be converted into a machine-readable format for further processing.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion of the time value. When this event input is triggered, the function block begins converting the string passed to `IN`.

Event Outputs

- `CNF`: Signals the successful completion of the conversion. This event, along with the converted time value, is output at data output `OUT`.

Data Inputs

- `IN` (STRING): The input for the time value in STRING format, which is to be converted to TIME format.

Data Outputs

- `OUT` (TIME): The output at which the converted time value is displayed in TIME format.

Adapters

No adapters are available.

Functionality

The function block uses the built-in function `STRING_AS_TIME` to convert the string received via input `IN` into a TIME value. The conversion is triggered by the event `REQ` and confirmed by the event `CNF` upon successful conversion.

Technical Features

- The conversion is performed directly without intermediate steps.
- The function block is part of the package `iec61131::conversion`.

State Overview

1. **Idle**: The function block waits for the event `REQ`.
2. **Processing**: Upon receiving `REQ`, the conversion is performed.
3. **Completion**: After successful conversion, the event `CNF` is triggered, and the state changes back to Idle.

Application Scenarios

- Processing of time values that are available as text in user input or external data sources.
- Integration into systems that require time values in TIME format for control or regulation tasks.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks, `F_STRING_AS_TIME` specializes in converting time values and therefore offers a direct and efficient solution for this specific use case.

Conclusion

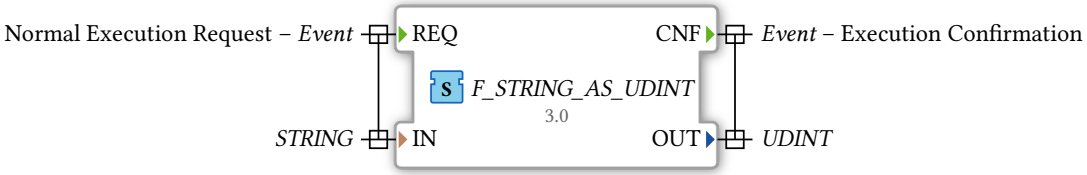
The `F_STRING_AS_TIME` function block is an efficient tool for converting time values in STRING format to TIME format. Its simple and direct operation makes it ideal for applications that require fast and reliable conversion.

F_STRING_AS_UDINT

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_STRING_AS_UDINT` converts a `STRING` value to a `UDINT` value (unsigned double integer). It is part of the `iec61131::conversion` package and enables the easy conversion of strings to numeric values.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs

- **IN**: The input expects a `STRING` value, which is to be converted to a `UDINT` value.

Data Outputs

- **OUT**: The output provides the converted `UDINT` value.

Adapters

The function block does not use any adapters.

Functionality

The function block performs the conversion using the built-in function `STRING_AS_UDINT`. When the event `REQ` is triggered, the `STRING` value at input `IN` is converted to a `UDINT` value and output `OUT`. The event `CNF` is then triggered to signal the successful completion of the conversion.

Technical Features

- The conversion is performed directly without additional parameters.
- The function block is optimized for simple conversions and does not support complex formatting or error handling.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Conversion**: Performs the conversion and outputs the result.
3. **Acknowledgement**: Triggers the event `CNF` and returns to the idle state.

Application Scenarios

- Conversion of user input or communication data into numeric values.
- Integration into systems that require numeric processing but receive strings as input.

⚖️ Comparison with Similar Function Blocks

- Compared to generic conversion function blocks, `F_STRING_AS_UDINT` offers a specialized and efficient solution for converting `STRING` to `UDINT`.
- Other function blocks might offer additional features such as error handling or formatting, but are more complex to use.

Conclusion

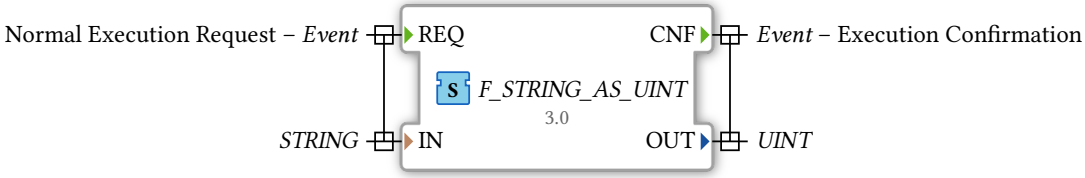
The `F_STRING_AS_UDINT` function block is a simple and efficient solution for converting strings to numeric values. It is particularly suitable for applications where a fast and direct conversion is required.

F_STRING_AS_UINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_STRING_AS_UINT converts a STRING value to a UINT value (unsigned integer). This function is particularly useful when numeric values are stored as strings and are needed in numerical form for further calculations or control tasks.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion of the STRING value to UINT. The input is linked to data input IN.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to data output OUT.

Data Inputs

- **IN** (STRING): The input for the string value to be converted.

Data Outputs

- **OUT** (UINT): The output for the converted numeric value.

Adapters

No adapters are available.

Functionality

When the event REQ is triggered, the algorithm is executed, which converts the STRING value from IN into a UINT value using the function STRING_AS_UINT. The result is output to OUT, and the event CNF signals the successful completion of the conversion.

Technical Features

- The function block has a simple structure and contains only one algorithm for the conversion.
- The conversion is performed directly without additional parameters or complex state logic.

State Overview

The function block has no internal states. Execution occurs immediately upon the triggering of REQ.

Application Scenarios

- Processing user input in string form.
- Converting data from text files or communication protocols for further numerical processing.
- Integration into control systems that require numerical values from string sources.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, F_STRING_AS_UINT specializes in converting STRING to UINT, thus providing a simple and direct solution for this specific use case.
- Other blocks might support additional features such as error handling or different numeric formats, which is not the case here.

Conclusion

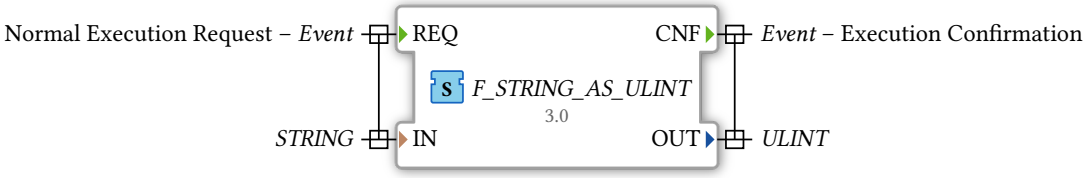
The F_STRING_AS_UINT function block is an efficient and straightforward solution for converting strings to unsigned integer values. Its simplicity makes it particularly suitable for applications where a fast and direct conversion is required.

F_STRING_AS_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_ULINT` converts a `STRING` value to a `ULINT` value (unsigned 64-bit integer). This block is particularly useful when strings need to be converted to numeric values, for example, when processing user input or parsing data from external sources.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `STRING`, containing the string to be converted.

Data Outputs

- **OUT**: The output of type `ULINT`, containing the result of the conversion.

Adapters

No adapters are present.

Operation

The function block performs the conversion as soon as the event `REQ` is triggered. The built-in function `STRING_AS_ULINT` is used to convert the input `STRING` into a value `IN`. The result is output at `OUT`, and the event `CNF` signals the successful completion of the operation.

Technical Features

- The function block is part of the package `iec61131::conversion`.
- The conversion is performed directly without any additional parameters or settings.

State Overview

The function block has no internal states. The conversion is re-executed with each `REQ` event.

Application Scenarios

- **Data Processing**: Converting strings to numeric values for calculations or storage.
- **User Input**: Processing input that is in string form but should be interpreted as numbers.
- **Logging**: Converting logical or textual data to numeric formats for further analysis.

⚖️ Comparison with Similar Building Blocks

- **F_STRING_AS_INT**: Converts a `STRING` to a `INT` (signed 32-bit integer). In contrast, `F_STRING_AS_ULINT` works with a larger range of values and without a sign.
- **F_STRING_AS_LINT**: Similar, but for `LINT` (signed 64-bit integer). `F_STRING_AS_ULINT` is optimized for unsigned values.

Conclusion

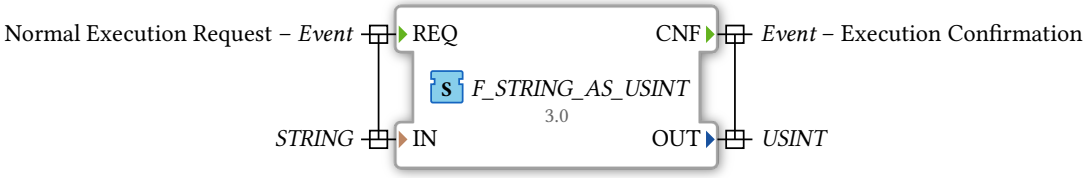
The function block `F_STRING_AS_ULINT` provides a simple and efficient way to convert strings to unsigned 64-bit integer values. Its clear interface and direct functionality make it ideal for applications requiring fast and reliable conversion.

F_STRING_AS_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_USINT` converts a `STRING` value to a `USINT` value (Unsigned Short Integer). This block is part of the `iec61131::conversion` package and enables the easy conversion of strings to numeric values.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: This input expects a `STRING` value, which is to be converted to a `USINT` value.

Data Outputs

- **OUT**: Outputs the converted `USINT` value.

Adapters

This function block does not have any adapters.

Functionality

When the `REQ` event is triggered, the algorithm is executed, converting the `STRING` value at the `IN` input into a `USINT` value. The result is output at `OUT`, and the `CNF` event signals the completion of the conversion.

Technical Features

- The block uses the `STRING_AS_USINT` function for the conversion.
- The conversion occurs synchronously with the triggering of the `REQ` event.

State Overview

The function block has no internal states. The conversion occurs immediately after the `REQ` event is triggered.

Application Scenarios

- Conversion of user input or communication data into numeric values.
- Use in control logic where strings need to be converted into numeric values.

⚖️ Comparison with Similar Blocks

- Compared to other conversion blocks such as `F_STRING_AS_INT` or `F_STRING_AS_UINT`, this block is specifically optimized for conversion to `USINT` values.

Conclusion

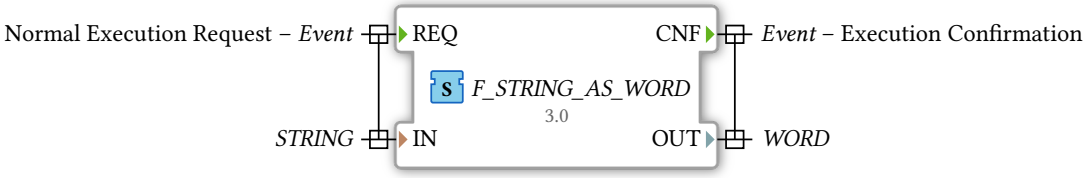
The `F_STRING_AS_USINT` function block provides a simple and efficient way to convert `STRING` values to `USINT` values. It is particularly useful in applications where strings need to be converted into numeric values.

F_STRING_AS_WORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_STRING_AS_WORD` is used to convert a `STRING` value to a `WORD` value. This functionality is particularly useful in scenarios where strings need to be converted into numeric values for use in further calculations or control logic.



Interface Structure

Event Inputs

- **REQ**: Normal execution request input. Used to start the conversion. Connected to the data input `IN`.

Event Outputs

- **CNF**: Execution Confirmation. Triggered after successful conversion. Associated with the data output `OUT`.

Data Inputs

- **IN**: Input of type `STRING`. Contains the string value to be converted.

Data Outputs

- **OUT**: Output of type `WORD`. Contains the result of the input string conversion.

Adapters

No adapters available.

Functionality

The function block performs the conversion when the event `REQ` is received. The input string `IN` is converted into a `WORD` value using the function `STRING_AS_WORD`, and the result is output to `OUT`. The confirmation event `CNF` is then triggered.

Technical Features

- The function block has a simple structure and contains only one conversion algorithm.
- The conversion is performed directly without additional state management.

State Overview

The function block has no internal states. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Conversion of user input or communication data into numerical values.
- Use in control logic that requires numerical processing.
- Integration into larger automation systems that process string and word data.

⚖️ Comparison with similar function blocks

- Compared to other conversion function blocks such as `F_STRING_TO_INT` or `F_STRING_TO_REAL`, this function block specifically converts to the `WORD` data type.
- Simpler and more direct than function blocks with additional features such as error handling or formatting.

Conclusion

The `F_STRING_AS_WORD` function block offers a simple and efficient way to convert strings to `WORD` values. Its clear interface and direct functionality make it ideal for use in various automation and control scenarios.

F_STRING_TO_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_STRING_TO_WSTRING converts a STRING data type to a WSTRING data type. This is particularly useful in environments where conversion between narrow (8-bit) and wide (16-bit) strings is required, for example, when internationalizing applications or communicating with systems that use different character encodings.



Interface Structure

Event Inputs

- **REQ**: Initiates the conversion. When this event is triggered, the input value IN is read and processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event is output along with the converted output value OUT.

Data Inputs

- **IN** (STRING): The input string to be converted to WSTRING.

Data Outputs

- **OUT** (WSTRING): The resulting wide string after the conversion.

Adapters

This function block does not use any adapters.

Operation

The function block performs the conversion as soon as the REQ event arrives. The value STRING from IN is converted to WSTRING using the function STRING_TO_WSTRING, and the result is output to OUT. Simultaneously, the CNF event is triggered to signal the successful completion of the operation.

Technical Features

- The conversion is performed without additional parameters or configuration.
- The block is implemented as a simple function block (SimpleFB), which guarantees efficient execution.

State Overview

The function block has no internal states. The conversion is performed anew with each REQ event.

Application Scenarios

- **Internationalization**: Conversion of local strings to Unicode-enabled WSTRING values.
- **System Communication**: Exchange of data between systems using different character sets.
- **Data Processing**: Processing of text data in different encodings.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, F_STRING_TO_WSTRING specializes in converting between STRING and WSTRING.
- Other blocks, such as F_WSTRING_TO_STRING, offer the reverse functionality.

Conclusion

The F_STRING_TO_WSTRING function block is a simple and efficient tool for converting strings between different encodings. Its clear interface and direct functionality make it particularly suitable for applications that require fast and reliable conversion.

STRING Conversions

Here you will find functions for converting the STRING data type. String Conversions ...

🛠️ Related exercises

- NumericValue_ID_TO_NVS
- Uebung_020c2_sub
- Uebung_051
- Uebung_052
- Uebung_120
- Uebung_121
- Uebung_122
- Uebung_122b
- Uebung_123
- Uebung_124
- Uebung_125
- Uebung_126
- SIN_TO_PCAN_Callback_DWord
- SIN_TO_PCAN_Callback_Byte
- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_12x_sub
- Uebung_130
- Uebung_131
- Uebung_132
- Uebung_133
- Uebung_134

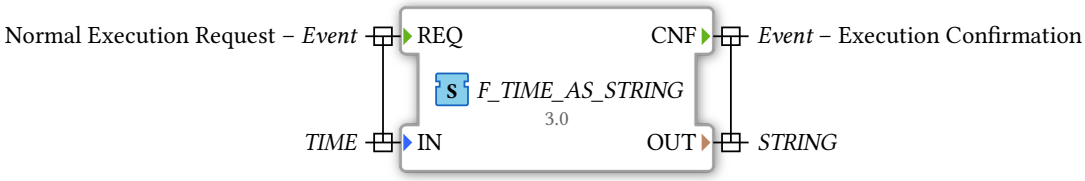
TIME

F_TIME_AS_STRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_AS_STRING` is used to convert a `TIME` value into a `STRING` value. It is particularly useful when time values are needed for display or logging in textual form.



Interface Structure

Event Inputs

- **REQ**: Triggers the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the successful execution of the function block. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (`TIME`): The time value to be converted to a string.

Data Outputs

- **OUT** (`STRING`): The resulting string representing the converted time value.

Adapters

No adapters are defined for this function block.

Functionality

The function block performs the conversion as soon as the event `REQ` is received. The input value `TIME` is converted to a value `STRING` and output `OUT`. The conversion is performed using the function `TIME_AS_STRING`. After the conversion is complete, the event `CNF` is triggered.

Technical Features

- This function block is a simple function block (SimpleFB) and does not perform any complex state transitions.
- The conversion is performed directly without additional parameters.

State Overview

Since this is a simple function block, there are no state transitions. This block performs the conversion on every `REQ` event and signals completion with `CNF`.

Application Scenarios

- Displaying timestamps in user interfaces.
- Logging timestamps in textual log files.
- Integration into systems that process timestamps as strings.

⚖️ Comparison with Similar Blocks

- Compared to other conversion blocks like `F_INT_AS_STRING` or `F_REAL_AS_STRING`, this block is specifically optimized for converting `TIME` values.
- Similar functionality could also be achieved with a custom algorithm, but this block offers a standardized and reusable solution.

Conclusion

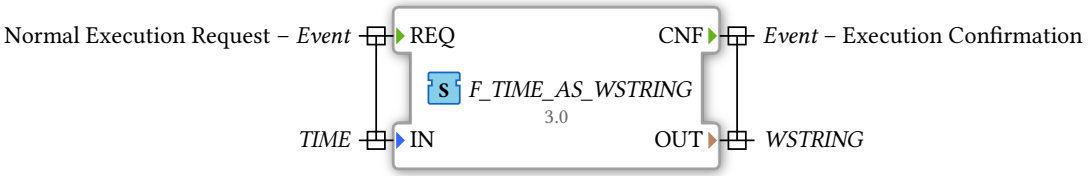
The `F_TIME_AS_STRING` function block offers a simple and efficient way to convert time values into strings. Its clear interface and direct functionality make it a useful building block for various applications where time values are required in textual form.

F_TIME_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_AS_WSTRING` converts a TIME value to a WSTRING value. This is particularly useful when time values are needed for display or logging in a human-readable format.



Interface Structure

Event Inputs

- **REQ**: Normal execution call. When this input is triggered, the function block performs the conversion.

Event Outputs

- **CNF**: Confirmation of execution. Triggered after successful conversion.

Data Inputs

- **IN**: Input for the TIME value to be converted.

Data Outputs

- **OUT**: Output for the converted WSTRING value.

Adapters

This function block has no adapters.

Functionality

The function block converts the incoming TIME value (`IN`) into a WSTRING value (`OUT`) as soon as the event `REQ` is triggered. The conversion is performed using the function `TIME_AS_WSTRING`. After successful conversion, the event `CNF` is triggered.

Technical Features

- The function block has a simple structure and contains only one algorithm for the conversion.
- The conversion is performed directly and without additional parameters.

State Overview

The function block has no complex states. It responds to the `REQ` event by executing the algorithm and confirms the execution with the `CNF` event.

Application Scenarios

- Displaying time values in user interfaces.
- Logging timestamps in a readable format.
- Integration into systems that require time values as strings.

⚖️ Comparison with Similar Blocks

- Compared to other conversion blocks, `F_TIME_AS_WSTRING` specializes in converting TIME to WSTRING.
- Other blocks might offer additional formatting options, but this block is simple and efficient.

Conclusion

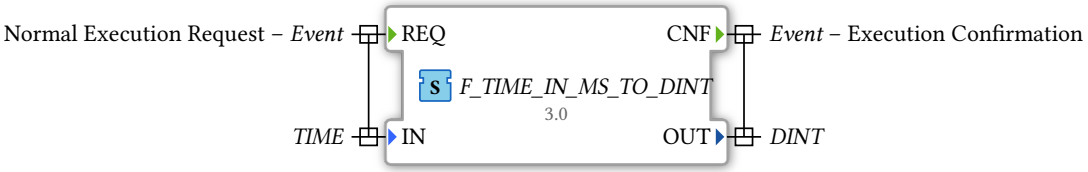
The `F_TIME_AS_WSTRING` function block is a simple and effective tool for converting TIME values to WSTRING. It is particularly suitable for applications where time values need to be displayed or processed in a readable form.

F_TIME_IN_MS_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_MS_TO_DINT` converts a TIME value in milliseconds to a DINT value. This is particularly useful when time values need to be converted into numerical values for use in calculations or for control purposes.



Interface Structure

Event Inputs

- **REQ**: Normal execution request event. Used to start the conversion. Linked to the data input `IN`.

Event Outputs

- **CNF**: Execution confirmation event. Signals the completion of the conversion. Linked to data output `OUT`.

Data Inputs

- **IN** (TIME): The input value in milliseconds to be converted.

Data Outputs

- **OUT** (DINT): The converted numeric value.

Adapters

No adapters are available.

Functionality

The function block performs the conversion by transforming the TIME value in milliseconds into a DINT value. The conversion occurs when the event `REQ` is triggered. After the conversion is complete, the event `CNF` is triggered, and the converted value is available at output `OUT`.

Technical Features

- The function block uses a simple algorithm for the direct conversion of TIME to DINT.
- The conversion is deterministic and occurs without delay.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Conversion State**: Performs the conversion and triggers the `CNF` event.

Application Scenarios

- Conversion of time values for use in numerical calculations.
- Integration into control systems that require numerical time values.
- Use in logging and monitoring systems that store timestamps in numerical form.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the block **`F_MULTIME`** (from the `arithmetic` category) for this purpose:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- **F_TIME_TO_UDINT**: Converts TIME to UDINT, while `F_TIME_IN_MS_TO_DINT` converts to DINT.
- **F_TIME_IN_US_TO_DINT**: Converts TIME in microseconds to DINT, while this function block uses milliseconds.

Conclusion

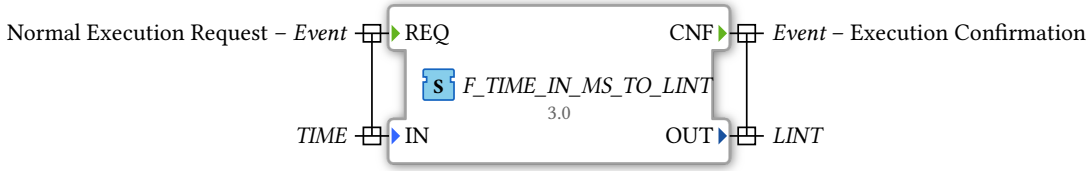
The function block `F_TIME_IN_MS_TO_DINT` is a simple and efficient tool for converting time values in milliseconds to numeric DINT values. It is particularly suitable for applications requiring fast and deterministic conversion.

F_TIME_IN_MS_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_MS_TO_LINT` converts a time value (`TIME`) in milliseconds into a 64-bit integer value (`LINT`). This functionality is particularly useful when time values are needed in numerical form for calculations or comparisons.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion of the time value. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: This input expects a time value (`TIME`) to be converted into milliseconds.

Data Outputs

- **OUT**: Outputs the converted time value as a 64-bit integer (`LINT`).

Adapters

This function block does not have any adapters.

Functionality

When the event input `REQ` is triggered, the algorithm is executed, which converts the time value `IN` into milliseconds and outputs the result as `LINT` via the output `OUT`. The event `CNF` signals the successful completion of the conversion.

Technical Features

- The function block uses the function `TIME_IN_MS_TO_LINT` to convert the time value.
- The conversion is performed with high accuracy because the `LINT` data type is 64 bits wide.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Time measurements where the results need to be processed further in numerical form.
- Control applications that require precise time intervals in milliseconds.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is sought to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion function block does not usually exist, as this is elegantly solved using multiplication.

Use the function block **`F_MULTIME`** (from the `arithmetic` category) for this:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- Compared to other conversion function blocks that may only convert to smaller integer types, this function block offers higher accuracy by using the `LINT` data type.
- Similar function blocks might convert time to other units (e.g., seconds or microseconds), while this function block is specifically designed for milliseconds.

Conclusion

The `F_TIME_IN_MS_TO_LINT` function block is an efficient tool for the precise conversion of time values in milliseconds to a 64-bit integer value. Its simple and direct operation makes it particularly suitable for applications that require accurate time measurements and calculations.

F_TIME_IN_MS_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_MS_TO_LREAL` is used to convert a TIME value in milliseconds to an LREAL value. This is particularly useful when time values are needed for further calculations or analyses with higher precision.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. Triggered after successful conversion.

Data Inputs

- **IN**: Input variable of type `TIME`. The time value to be converted in milliseconds.

Data Outputs

- **OUT**: Output variable of type `LREAL`. The converted value is a floating-point number.

Adapters

- No adapters are available.

Functionality

The function block performs the conversion by transforming the incoming TIME value in milliseconds into an LREAL value. The conversion is performed using the algorithm `REQ`, which uses the function `TIME_IN_MS_TO_LREAL`.

Technical Features

- The function block has a simple structure and contains only one algorithm for the conversion.
- The conversion is performed directly and without any additional delay.

State Overview

The function block has no internal state. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Conversion of time values for precise calculations.
- Integration into control systems that require floating-point values for time measurements.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the block **`F_MULTIME`** (from the category `arithmetic`) for this purpose:

- Multiply your numeric value by the desired time base.

- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- Compared to other conversion function blocks, this one is specifically designed for converting TIME to LREAL and therefore offers a simple and direct solution for this use case.

Conclusion

The `F_TIME_IN_MS_TO_LREAL` function block is a simple and efficient tool for converting time values in milliseconds to floating-point numbers. Its clear interface and direct functionality make it a reliable component in control systems.

F_TIME_IN_MS_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_MS_TO_UDINT` converts a `TIME` value in milliseconds to a `UDINT` value. This is particularly useful when time values are to be used in numerical calculations or for control purposes.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. Triggered after successful conversion.

Data Inputs

- **IN**: Input of type `TIME`. The time value in milliseconds to be converted.

Data Outputs

- **OUT**: Output of type `UDINT`. The converted numeric value.

Adapters

No adapters are available.

Functionality

The function block performs the conversion by transforming the `TIME` value in milliseconds into a `UDINT` value. The conversion occurs synchronously with the triggering of the `REQ` event. After the conversion is complete, the `CNF` event is triggered.

Technical Features

- The function block is simple and has no internal states.
- The conversion is immediate and instantaneous.

State Overview

Since it is a simple function block, there are no state transitions or complex state logic.

Application Scenarios

- Conversion of time values for numerical calculations.
- Integration into control logic that requires numerical time values.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block does not usually exist, as this is elegantly solved using multiplication.

Use the function block **`F_MULTIME`** (from the category `arithmetic`):

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- Compared to other conversion function blocks, this one is specifically designed for converting `TIME` to `UDINT`.
- Other function blocks might offer additional features such as scaling or filtering, which is not the case here.

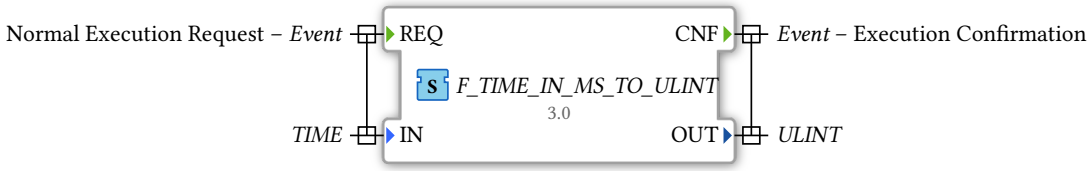
Conclusion

The `F_TIME_IN_MS_TO_UDINT` function block is a simple and efficient tool for converting time values in milliseconds into numerical values. It is particularly suitable for applications where a fast and direct conversion is required.

F_TIME_IN_MS_TO_ULINT

• – ◦ • – ◦ • – ◦ •

The function block `F_TIME_IN_MS_TO_ULINT` converts a time value in milliseconds (`TIME`) into an unsigned 64-bit integer value (`ULINT`). This block is particularly useful when time values are needed in numerical form for further calculations or comparisons.



- **REQ**: Starts the conversion. This input is linked to the data input `IN`.
- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.
- **IN**: This input expects a time value of type `TIME`, specified in milliseconds.
- **OUT**: Returns the converted value as an unsigned 64-bit integer (`ULINT`).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block has no adapters.

The function block performs the conversion by transforming the `TIME` value in milliseconds into a `ULINT` value. The conversion occurs synchronously when the `REQ` event occurs. After the conversion is complete, the `CNF` event is triggered, and the converted value is available at output `OUT`.

- The function block uses the `TIME_IN_MS_TO_ULINT` function for the conversion.
- The implementation is in Structured Text (ST).
- The function block is part of the `iec61131::conversion` package.

The function block has no internal states. The conversion occurs immediately upon the occurrence of the `REQ` event.

- Conversion of time values for mathematical operations.
- Integration into systems that store or process timestamps as numeric values.

Application Scenarios

State Overview

Technical Features

Functionality

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is sought to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

For this purpose, use the function block **`F_MULTIME`** (from the category `arithmetic`):

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).
- Unlike generic conversion blocks, this function block is specialized for converting `TIME` to `ULINT`.
- Other blocks might offer additional options, such as conversion to other data types, but are less specific.

The `F_TIME_IN_MS_TO_ULINT` function block provides an efficient and specialized solution for converting millisecond time values to unsigned 64-bit integers. Its simple interface and straightforward operation make it ideal for applications that require numeric time values.

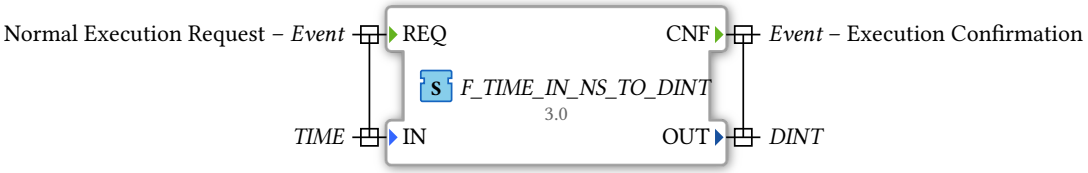
⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

F_TIME_IN_NS_TO_DINT

• – ◦ • – ◦ • – ◦ •

The function block `F_TIME_IN_NS_TO_DINT` converts a TIME value in nanoseconds to a DINT value. This conversion is useful when time values are needed in numerical calculations or for further processing.



- **REQ**: Starts the execution of the function block. This input is linked to the data input `IN`.
- **CNF**: Signals the successful completion of the conversion. This output is linked to the data output `OUT`.
- **IN** (TIME): The input for the time value in nanoseconds to be converted.
- **OUT** (DINT): The output for the converted numeric value.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters are included.

Functionality

Upon receiving an incoming `REQ` event, the function block converts the TIME value `IN` to a DINT value `OUT`. The conversion is performed using the function `TIME_IN_NS_TO_DINT`. After successful conversion, the `CNF` event is triggered.

- The function block has a simple structure and contains only one algorithm for the conversion.

1. **Waiting for REQ**: The function block waits for an incoming `REQ` event.
2. **Conversion**: Upon `REQ`, the TIME value `IN` is converted to DINT.
3. **Output**: After successful conversion, the `CNF` event is triggered.

- Conversion of time values for numerical calculations.
- Integration into control systems where time values are required as numerical values.

Application Scenarios

State Overview

Technical Features

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the block **`F_MULTIME`** (from the `arithmetic` category) for this purpose:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).
- Compared to other conversion blocks, `F_TIME_IN_NS_TO_DINT` specializes in converting TIME to DINT.
- Similar blocks might offer other data types or additional functions, such as scaling or filtering.

The function block `F_TIME_IN_NS_TO_DINT` provides a simple and efficient way to convert time values in nanoseconds into numerical DINT values. It is particularly useful in applications where time values need to be integrated into calculations or control logic.

⚖️ Vergleich mit ähnlichen Bausteinen

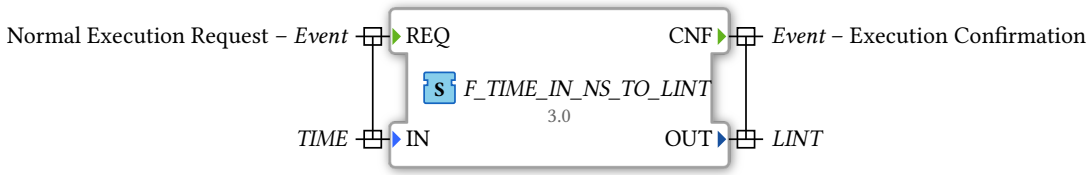
Conclusion

F_TIME_IN_NS_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_NS_TO_LINT` converts a time value in nanoseconds (`TIME`) into a 64-bit integer value (`LINT`). This conversion is useful when time values are needed in numerical form for calculations or comparisons.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: This input expects a time value of type `TIME`, specified in nanoseconds.

Data Outputs

- **OUT**: This output returns the converted value as a 64-bit integer (`LINT`).

Adapters

- No adapters are present.

Operation

This function block performs the conversion when the event `REQ` is triggered. The algorithm converts the time value `IN` in nanoseconds to a value `LINT` and stores the result in `OUT`. The event `CNF` is then triggered to signal the completion of the operation.

Technical Details

- The function block is implemented as a *SimpleFB*, meaning it has no internal state and is re-executed with each call to `REQ`.
- The conversion is performed directly by the function `TIME_IN_NS_TO_LINT`, which converts the value `TIME` to a value `LINT`.

State Overview

Since this is a simple function block, there are no state transitions or state machines.

Application Scenarios

- Time measurement and processing in control systems where numerical operations on time values are required.
- Integration into systems that need to store or transmit timestamps in a numerical format.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the function block **`F_MULTIME`** (from the category `arithmetic`):

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- Unlike generic conversion blocks, this function block specializes in converting `TIME` to `LINT`.
- Other blocks might offer additional features such as scaling or formatting, which is not the case here.

Conclusion

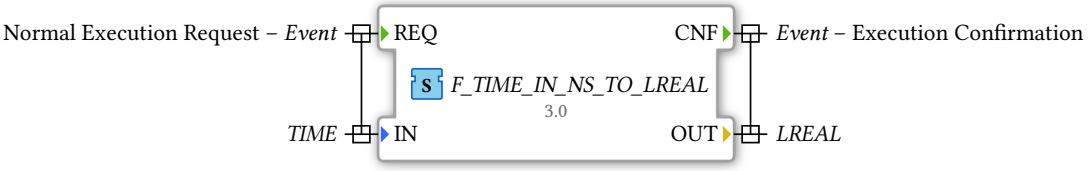
The `F_TIME_IN_NS_TO_LINT` function block provides a simple and efficient way to convert time values in nanoseconds to 64-bit integers. It is particularly useful in applications that require precise time measurements and calculations.

F_TIME_IN_NS_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_NS_TO_LREAL` is used to convert a time value in nanoseconds (`TIME`) into a floating-point value (`LREAL`). This conversion is particularly useful in control applications where time values need to be used in mathematical calculations.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input expects a time value in nanoseconds (`TIME`).

Data Outputs

- **OUT**: Returns the converted floating-point value (`LREAL`).

Adapters

- No adapters are available.

Functionality

This function block converts the nanosecond time value `IN` (`TIME`) into a floating-point value (`LREAL`). The conversion occurs every time the event `REQ` is triggered. After successful conversion, the event `CNF` is triggered, and the result value is available at output `OUT`.

Technical Features

- The function block is implemented as a simple FB (`SimpleFB`).
- The conversion is performed using the built-in function `TIME_IN_NS_TO_LREAL`.

State Overview

The function block has no internal states. Execution occurs immediately upon the occurrence of the `REQ` event.

Application Scenarios

- Time measurements in control systems that require further processing in mathematical expressions.
- Integration into control loops where time values are required as floating-point numbers.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the block **`F_MULTIME`** (from the `arithmetic` category) for this purpose:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with Similar Building Blocks

- Unlike generic conversion blocks, `F_TIME_IN_NS_TO_LREAL` is specifically optimized for converting time values in nanoseconds to floating-point numbers.
- Other blocks, such as `F_TIME_TO_LREAL`, might offer similar functionality, but without explicitly specifying the nanosecond unit.

Conclusion

The `F_TIME_IN_NS_TO_LREAL` function block provides an efficient and specialized solution for converting time values to floating-point numbers. Its ease of use and direct integration make it ideal for applications requiring precise time processing.

F_TIME_IN_NS_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_NS_TO_UDINT` converts a time value in nanoseconds (`TIME`) into an unsigned 32-bit integer value (`UDINT`). This conversion is particularly useful when time values are needed in a numeric format for further calculations or control purposes.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event input is triggered, the function block performs the conversion of the time value present at `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event is triggered as soon as the converted value is output to `OUT`.

Data Inputs

- **IN** (`TIME`): The input for the time value in nanoseconds to be converted.

Data Outputs

- **OUT** (`UDINT`): The output to which the converted unsigned 32-bit integer value is output.

Adapters

This function block does not use any adapters.

Functionality

The function block performs the conversion by converting the nanosecond time value (`TIME`) received at `IN` into an unsigned 32-bit integer value (`UDINT`) using the function `TIME_IN_NS_TO_UDINT`. The conversion occurs synchronously with the triggering of the `REQ` event. After successful conversion, the `CNF` event is triggered, and the converted value is output to `OUT`.

Technical Features

- The conversion is instantaneous and without delay.
- The function block is suitable for use in real-time applications.
- No additional parameters or configurations are required.

State Overview

1. **Idle State**: The function block waits for the `REQ` event.
2. **Conversion State**: When `REQ` is triggered, the time value is converted.
3. **Acknowledgement State**: After successful conversion, `CNF` is triggered, and the function block returns to the idle state.

Application Scenarios

- Conversion of timestamps for use in numerical calculations.
- Integration into control logic that needs to process time values in a numerical format.
- Use in diagnostic systems to display durations in a readable format.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the block **`F_MULTIME`** (from the category `arithmetic`) for this purpose:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with Similar Blocks

- Unlike generic conversion blocks, this function block is specifically optimized for converting time values to nanoseconds.
- Similar blocks might offer additional parameters or configuration options, but this function block is designed for simplicity and efficiency.

Conclusion

The `F_TIME_IN_NS_TO_UDINT` function block provides a simple and efficient way to convert nanosecond time values to unsigned 32-bit integer values. Its clear interface and direct operation make it ideal for applications requiring fast and reliable conversion.

F_TIME_IN_NS_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_NS_TO_ULINT` converts a `TIME` value in nanoseconds to a `ULINT` value. This conversion is particularly useful when time values are needed in numerical calculations or for further processing steps.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input expects a `TIME` value in nanoseconds, which is to be converted.

Data Outputs

- **OUT**: Outputs the converted `ULINT` value.

Adapters

No adapters are available.

Functionality

The function block converts the nanosecond value `TIME` (`IN`) to a nanosecond value `ULINT` (`OUT`) as soon as the event `REQ` is triggered. The conversion is performed using the function `TIME_IN_NS_TO_ULINT`. After the conversion is complete, the event `CNF` is triggered.

Technical Features

- The function block has a simple structure and contains only one algorithm for the conversion.
- The conversion is direct and without any additional delay.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of time values for use in numerical calculations.
- Integration into control systems where time values need to be further processed as numerical values.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is sought to convert a numerical value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block does not usually exist, as this is elegantly solved using multiplication.

Use the function block **`F_MULTIME`** (from the category `arithmetic`) for this:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- Compared to other conversion function blocks, this one is specifically designed to convert `TIME` to `ULINT`.
- Similar function blocks might use different data types or units, but this one is optimized for nanoseconds.

Conclusion

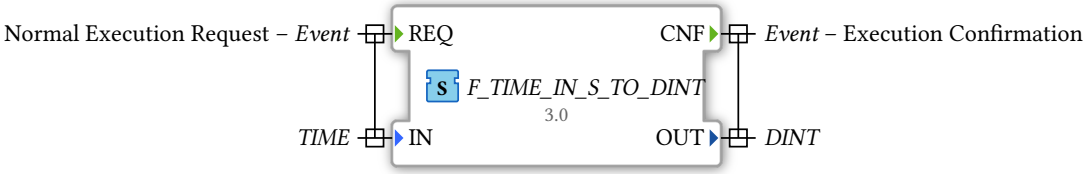
The `F_TIME_IN_NS_TO_ULINT` function block offers a simple and efficient way to convert time values in nanoseconds into `ULINT` values. Its clear interface and direct functionality make it a useful component in automation technology.

F_TIME_IN_S_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_S_TO_DINT` converts a time value in seconds (`TIME`) into a signed integer value (`DINT`). This conversion is useful when time values need to be used in numerical calculations or for control purposes.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input expects a time value of type `TIME` in seconds.

Data Outputs

- **OUT**: The output returns the converted value as a signed integer (`DINT`).

Adapters

No adapters are used.

Functionality

The function block performs the conversion when the event `REQ` is triggered. The algorithm converts the time value `IN` in seconds to a `DINT` value and stores the result in `OUT`. After the conversion is complete, the event `CNF` is triggered.

Technical Features

- The function block uses the function `TIME_IN_S_TO_DINT` for the conversion.
- The block is simple and efficient because it is directly based on the ST algorithm.

State Overview

The function block has no internal states. The conversion occurs immediately after `REQ` is triggered.

Application Scenarios

- Conversion of time values for numerical calculations.
- Integration into control logic that requires numerical time values.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the block **`F_MULTIME`** (from the `arithmetic` category) for this purpose:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with Similar Building Blocks

- Compared to other conversion blocks, this one is specifically designed to convert `TIME` to `DINT`.
- Simpler and more direct than general conversion blocks, which may require additional parameters.

Conclusion

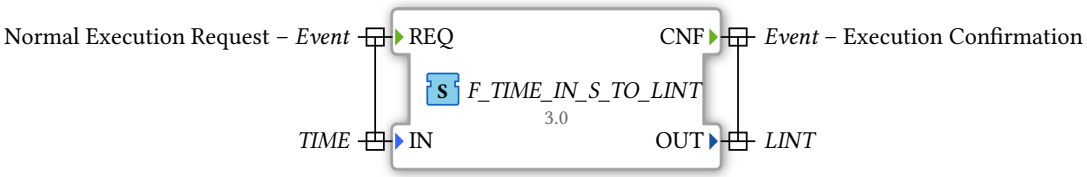
The `F_TIME_IN_S_TO_DINT` function block offers an efficient and simple solution for converting time values in seconds to signed integers. Its clear interface and direct functionality make it ideal for control applications.

F_TIME_IN_S_TO_LINT

• – ○ • – ○ • – ○ •

Introduction

The function block `F_TIME_IN_S_TO_LINT` converts a time value in seconds (`TIME`) into a 64-bit integer value (`LINT`). This conversion is particularly useful when time values are needed in numerical form for calculations or comparisons.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion of the time value. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input for the time value in seconds (`TIME`) to be converted.

Data Outputs

- **OUT**: The output for the converted 64-bit integer value (`LINT`).

Adapters

No adapters are present.

Functionality

The function block performs the conversion by transforming the time value in seconds (`TIME`) into a 64-bit integer value (`LINT`). The conversion occurs when the `REQ` event is triggered, and the result is output via the `OUT` data output, followed by a `CNF` event.

Technical Features

- The function block uses the standard `61499-1`.
- The conversion is performed using the function `TIME_IN_S_TO_LINT`, which is implemented in the function block's algorithm.
- The function block is part of the `iec61131::conversion` package.

State Overview

The function block does not have complex states. It reacts to the `REQ` event, performs the conversion, and signals completion with the `CNF` event.

Application Scenarios

- Conversion of time values for numerical calculations.
- Integration into control systems that require time values in a numerical format.
- Use in systems that require high precision in time measurement.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block is not usually available, as this is elegantly solved using multiplication.

Use the function block **`F_MULTIME`** (from the category `arithmetic`) for this:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- Compared to other conversion function blocks that may support different data types, this function block is specifically designed for converting `TIME` to `LINT`.

Other function blocks might offer conversion to other integer types (e.g., `INT` or `DINT`), but this block offers higher precision by using `LINT`.

Conclusion

The `F_TIME_IN_S_TO_LINT` function block is an efficient tool for converting time values in seconds to 64-bit integers. Its simple and straightforward operation makes it a reliable component in control systems that require precise time measurements and calculations.

F_TIME_IN_S_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_S_TO_LREAL` converts a time value in seconds (`TIME`) into a floating-point value (`LREAL`). This conversion is useful when time values are needed in mathematical calculations or for further processing steps.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input expects a time value of type `TIME` in seconds, which is to be converted.

Data Outputs

- **OUT**: Outputs the converted value as a floating-point number (`LREAL`).

Adapters

- No adapters are available.

Functionality

The function block performs the conversion from `TIME` to `LREAL` as soon as the event `REQ` is triggered. The conversion is performed using the function `TIME_IN_S_TO_LREAL`, which converts the time value in seconds into a corresponding floating-point value. Upon completion of the conversion, the event `CNF` is triggered.

Technical Features

- The function block is a simple algorithm (`SimpleFB`) and does not perform any complex state transitions.
- The conversion is instantaneous and without any additional delay.

State Overview

Since it is a simple function block, there are no state transitions. The conversion occurs immediately upon triggering `REQ`.

Application Scenarios

- Conversion of time values for mathematical operations.
- Integration into control logic where time values are required as floating-point numbers.
- Use in control engineering or data processing where precise time measurements are necessary.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the block **`F_MULTIME`** (from the `arithmetic` category) for this purpose:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with Similar Function Blocks

- Unlike other conversion blocks that may require additional parameters or states, `F_TIME_IN_S_TO_LREAL` is designed for a specific and direct conversion.
- Similar function blocks could be, for example, `TIME_TO_STRING` or `TIME_TO_INT`, which, however, provide different output types.

Conclusion

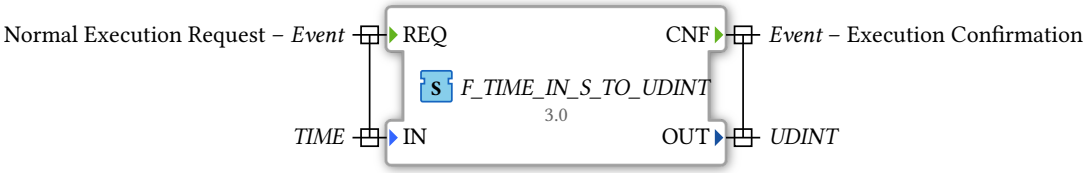
The `F_TIME_IN_S_TO_LREAL` function block offers an efficient and direct way to convert time values in seconds to floating-point numbers. Its simple structure and fast execution make it ideal for applications that require precise and straightforward conversion.

F_TIME_IN_S_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_S_TO_UDINT` converts a TIME value in seconds to a UDINT value. This conversion is useful when time values are needed in numerical operations or for further processing steps.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. Triggered after successful conversion.

Data Inputs

- **IN**: TIME value in seconds to be converted.

Data Outputs

- **OUT**: Converted UDINT value.

Adapters

No adapters available.

Functionality

This function block converts the incoming TIME value (IN) in seconds into a UDINT value (OUT). The conversion is performed by calling the function `TIME_IN_S_TO_UDINT(IN)` in the algorithm. When the REQ event is triggered, the conversion is carried out and the CNF event is triggered.

Technical Features

- The function block is simple and efficient because it directly uses the built-in function `TIME_IN_S_TO_UDINT`.

- No additional states or complex logic are required.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the REQ event.

Application Scenarios

- Conversion of time values for numerical calculations.
- Integration into control systems where time values are required as numeric values.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the function block **`F_MULTIME`** (from the category `arithmetic`) for this purpose:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- Compared to other conversion function blocks, this one is specifically optimized for converting TIME to UDINT.
- Other function blocks might offer additional features such as scaling or limit checking, which is not the case here.

Conclusion

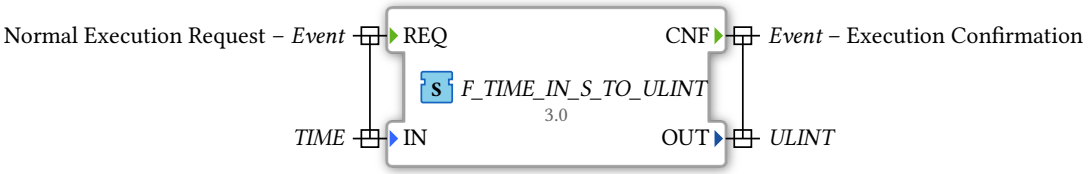
The `F_TIME_IN_S_TO_UDINT` function block is a simple and efficient function block for converting TIME values to UDINT. It is ideally suited for applications where time values are required in numerical form.

F_TIME_IN_S_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_S_TO_ULINT` converts a TIME value in seconds to a ULINT value. This is particularly useful when time values need to be converted into numerical formats for use in calculations or further processing.



Interface Structure

Event Inputs

- **REQ**: Normal execution request event. This event triggers the conversion and is linked to the data input `IN`.

Event Outputs

- **CNF**: Confirmation event after successful execution. This event is output along with the data output `OUT`.

Data Inputs

- **IN**: Input of type `TIME` containing the time value to be converted in seconds.

Data Outputs

- **OUT**: Output of type `ULINT` containing the converted numeric value.

Adapters

No adapters are present.

Functionality

The function block performs the conversion by transforming the TIME value (in seconds) into a ULINT value. The conversion occurs when the `REQ` event is triggered, and the result is output via the `OUT` output, accompanied by the `CNF` event.

Technical Features

- The function block uses the standard algorithm `TIME_IN_S_TO_ULINT` for the conversion.
- The function block is part of the `iec61131::conversion` package.

State Overview

The function block has no internal states. Execution occurs immediately upon the occurrence of the `REQ` event.

Application Scenarios

- Conversion of time values for use in numerical calculations.
- Integration into control systems that need to process time values in numerical form.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the block **`F_MULTIME`** (from the category `arithmetic`) for this purpose:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with Similar Function Blocks

- Unlike generic conversion blocks, this function block specializes in converting TIME to ULINT.
- Other blocks might offer different output types or additional functionality, such as conversion to other numeric formats.

Conclusion

The `F_TIME_IN_S_TO_ULINT` function block provides a simple and efficient way to convert time values to ULINT numeric values. Its clear interface and straightforward functionality make it ideal for applications that require such conversions.

F_TIME_IN_US_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_US_TO_DINT` converts a TIME value in microseconds (µs) to a DINT value. This conversion is particularly useful when time values are to be used in numerical calculations or for control purposes.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. Upon this event, the input value `IN` is processed, and the result is output at `OUT`.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. This event, along with the converted value, is output at `OUT`.

Data Inputs

- **IN**: The input value of type `TIME`, specified in microseconds, to be converted to a DINT value.

Data Outputs

- **OUT**: The converted value of type `DINT`, representing the result of converting the TIME value to microseconds.

Adapters

This function block does not use any adapters.

Operation

The function block performs the conversion by transforming the TIME value in microseconds into a DINT value. The conversion occurs synchronously with the arrival of the `REQ` event. The result is immediately output at `OUT` and confirmed by the `CNF` event.

Technical Features

- The conversion is performed directly and without delay.
- The function block is suitable for use in real-time applications.
- No additional parameters or configurations are required.

State Overview

1. **Idle State**: The function block waits for the `REQ` event.
2. **Active State**: Upon receipt of `REQ`, the conversion is performed and the result is output.
3. **Confirmation State**: The `CNF` event is triggered to signal successful conversion.

Application Scenarios

- Conversion of time values for use in numerical calculations.
- Integration into control algorithms that require time values in a different format.
- Use in systems that process timestamps or durations in microseconds.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is sought to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block does not usually exist, as this is elegantly solved using multiplication.

For this purpose, use the block **`F_MULTIME`** (from the category `arithmetic`):

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- Compared to generic conversion function blocks, this function block specializes in converting TIME to DINT.
- Other function blocks might offer additional features such as scaling or filtering, which is not the case here.

Conclusion

The `F_TIME_IN_US_TO_DINT` function block is a simple and efficient tool for converting time values in microseconds to a DINT value. Its simple interface and direct functionality make it ideal for applications that require fast and reliable conversion.

F_TIME_IN_US_TO_LINT

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_TIME_IN_US_TO_LINT` converts a TIME value in microseconds (µs) into a LINT value (64-bit integer). This conversion is useful when time values are needed in numerical calculations or for further processing.



Interface Structure

Event Inputs

- `REQ` (Normal Execution Request): Starts the execution of the function block. Linked to input `IN`.

Event Outputs

- `CNF` (Execution Confirmation): Signals the completion of the conversion. Linked to output `OUT`.

Data Inputs

- `IN` (TIME): The input value in microseconds (µs) to be converted into a LINT value.

Data Outputs

- `OUT` (LINT): The converted 64-bit integer value.

Adapters

This function block does not use adapters.

Functionality

The function block performs the conversion by transforming the TIME input value (`IN`) in microseconds into a LINT value (`OUT`). The conversion occurs when the `REQ` event is triggered and is confirmed by the `CNF` event.

Technical Features

- The algorithm uses the function `TIME_IN_US_TO_LINT` to perform the conversion.
- The function block is suitable for use in real-time applications.

State Overview

The function block has a simple state machine:

1. **Idle State**: Waits for the `REQ` event.
2. **Execution State**: Performs the conversion and triggers the `CNF` event.

Application Scenarios

- Time measurement and processing in control systems.
- Integration of time values into numerical calculations.
- Logging and analysis of time durations.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block does not usually exist, as this is elegantly solved using multiplication.

Use the function block **`F_MULTIME`** (from the category `arithmetic`) for this:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- Compared to other conversion function blocks, this one is specifically optimized for converting TIME to LINT.
- Similar function blocks might use different data types or units, e.g., milliseconds instead of microseconds.

Conclusion

The `F_TIME_IN_US_TO_LINT` function block provides an efficient and reliable way to convert time values in microseconds to 64-bit integers. Its simple interface and clear functionality make it ideal for a wide range of applications in automation technology.

F_TIME_IN_US_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_US_TO_LREAL` converts a time value in microseconds (`TIME`) into a double-precision floating-point number (`LREAL`). This conversion is particularly useful in control applications where time values are needed for further calculations.



Interface Structure

Event Inputs

- **REQ**: Trigger for the conversion. Upon this event, the input value `IN` is processed, and the result is output to `OUT`.

Event Outputs

- **CNF**: Confirms successful completion of the conversion. This event is output along with the converted value `OUT`.

Data Inputs

- **IN** (`TIME`): The input value in microseconds to be converted to a `LREAL` value.

Data Outputs

- **OUT** (`LREAL`): The converted value as a double-precision floating-point number.

Adapters

This function block does not use adapters.

Functionality

The function block performs the conversion by transforming the `TIME` input value `IN` into a `LREAL` value using the function `TIME_IN_US_TO_LREAL`. The result is passed to the output `OUT` and confirmed by the event `CNF`.

Technical Features

- The conversion is performed directly and without any additional delay.
- The function block is optimized for real-time applications and supports precise time measurements.

State Overview

The function block has no internal states. The conversion is performed every time the `REQ` event occurs.

Application Scenarios

- Conversion of time durations for mathematical calculations.

- Integration into control systems that require floating-point values for control algorithms.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the function block **`F_MULTIME`** (from the category `arithmetic`) for this purpose:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ **Comparison with Similar Blocks**

- Unlike generic conversion blocks, this function block specializes in converting `TIME` to `LREAL` and therefore offers optimized performance.
- Other blocks might have additional parameters or states, which is not the case here.

Conclusion

The `F_TIME_IN_US_TO_LREAL` function block is an efficient tool for the precise conversion of time values to floating-point numbers. Its simple and direct operation makes it ideal for applications that require fast and accurate conversions.

F_TIME_IN_US_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_US_TO_UDINT` converts a TIME value in microseconds (µs) into a UDINT value. This conversion is particularly useful in applications where time values need to be processed further in numerical form.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `TIME`, containing the time value to be converted in microseconds.

Data Outputs

- **OUT**: The output of type `UDINT`, containing the result of the conversion.

Adapters

No adapters are present.

Functionality

This function block converts the TIME value (in µs) passed to `IN` into a UDINT value. The conversion is performed by calling the function `TIME_IN_US_TO_UDINT(IN)`. After successful conversion, the result is output to `OUT`, and the event `CNF` is triggered.

Technical Features

- The function block has a simple structure and contains only one algorithm (`REQ`) that performs the conversion.
- There is no state machine or complex logic, as the block is solely responsible for the conversion.

State Overview

Because it is a simple function block, there are no state transitions or state machines. The block performs the conversion on each `REQ` event and signals completion with `CNF`.

Application Scenarios

- Time measurement and processing in control systems.
- Conversion of time values for storage or transmission in numerical form.
- Integration into systems that need to process time values in µs as UDINT.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block doesn't usually exist, as this is elegantly solved using multiplication.

Use the block **`F_MULTIME`** (from the `arithmetic` category) for this purpose:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with Similar Blocks

- Compared to other conversion blocks, `F_TIME_IN_US_TO_UDINT` specializes in converting TIME to UDINT.
- Similar blocks might support other time formats or other target data types, e.g., TIME to REAL or TIME to STRING.

Conclusion

The `F_TIME_IN_US_TO_UDINT` function block offers a simple and efficient way to convert microsecond time values to UDINT values. Its streamlined design makes it particularly suitable for applications where performance and simplicity are paramount.

F_TIME_IN_US_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_TIME_IN_US_TO_ULINT` converts a time value in microseconds (`TIME`) into an unsigned 64-bit integer value (`ULINT`). This is particularly useful when time values need to be converted for numerical calculations or further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN**: This input expects a time value of type `TIME` (in microseconds) to be converted.

Data Outputs

- **OUT**: Returns the converted value as an unsigned 64-bit integer (`ULINT`).

Adapters

- No adapters are available.

Functionality

The function block uses the algorithm `REQ` to convert the incoming `TIME` value into a `ULINT` value in microseconds. The conversion is performed by the function `TIME_IN_US_TO_ULINT(IN)`. Once the conversion is complete, the event `CNF` is triggered to signal successful completion.

Technical Features

- The function block has a simple structure and contains only one algorithm.
- The conversion is direct and requires no additional parameters or state management.

State Overview

Because it is a simple function block, there are no complex state transitions. The block responds to the `REQ` event, performs the conversion, and signals completion with `CNF`.

Application Scenarios

- Time measurement and processing in control systems.
- Conversion of time values for use in calculations or logging systems.
- Integration into larger control algorithms that require numeric time values.

Reverse Conversion (Numeric to TIME)

Often, a function like `DINT_TO_TIME` is needed to convert a numeric value (e.g., 500) back into a time value (e.g., 500 ms). Such an explicit conversion block is not usually available, as this is elegantly solved using multiplication.

Use the function block **`F_MULTIME`** (from the category `arithmetic`) for this:

- Multiply your numeric value by the desired time base.
- **Example:** `500 * T#1ms = T#500ms`
- This allows for flexible scaling (e.g., * T#1s for seconds, * T#100ms for tenths of a second).

⚖️ Comparison with similar function blocks

- Compared to generic conversion blocks, `F_TIME_IN_US_TO_ULINT` specializes in converting `TIME` to `ULINT`.
- Other blocks might offer additional features such as scaling or filtering, which is not the case here.

Conclusion

The function block `F_TIME_IN_US_TO_ULINT` is a simple yet effective tool for converting microsecond time values into numerical values. Its simplicity and direct operation make it a reliable component in control applications that require precise time processing.

TIME Conversions

🎧 Podcast

- The three timers of DIN EN 61131-3 decoded – TP, TON & TOF explained precisely
- E_TOF vs. FB_TOF: The event timer that doesn't tick cyclically – a revolution for automation systems?
- E_TON in industrial automation: How a simple timer creates safety and stability
- Event-driven timers: When which one makes sense for a fan – ETON, ETOF, ETP & more decoded
- FB_TOF and E_TOF: Delay timers in IEC 61131-3 and 61499

🛠️ Related exercises

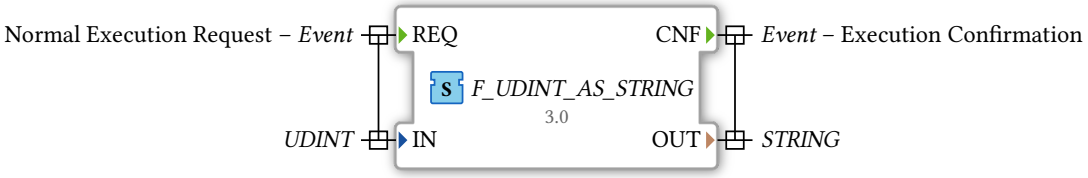
- Uebung_072c

UDINT

F_UDINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

The function block F_UDINT_AS_STRING converts an unsigned 32-bit integer value (UDINT) into a string (STRING). This block is particularly useful when numeric values need to be converted into a readable text format for display or logging.



- **REQ**: Starts the conversion. This input is linked to the data input IN.
- **CNF**: Signals the completion of the conversion. The output is linked to the data output OUT.
- **IN**: The unsigned 32-bit integer value (UDINT) to be converted to a string.
- **OUT**: The resulting string (STRING) after conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- No adapters are available.

When the event REQ is triggered, the algorithm is executed, converting the value of the input IN into a string and outputting the result to OUT. The event CNF is then triggered to signal the successful completion of the operation.

- The function block uses the function UDINT_AS_STRING for conversion.
- The implementation is in Structured Text (ST).

1. **Initialization**: The function block is ready and waiting for the REQ event.
2. **Conversion**: Upon receiving REQ, the value of IN is converted and output to OUT.
3. **Acknowledgement**: The CNF event is triggered to signal the completion of the conversion.

- Displaying numerical values in user interfaces.
- Logging measured values to text files or databases.
- Preparing data for communication via text protocols (e.g., HTTP, MQTT).
- Compared to generic conversion modules, F_UDINT_AS_STRING offers a specialized solution for converting UDINT to STRING.
- Other modules might support additional parameters such as formatting or length limits, which is not the case here.

The F_UDINT_AS_STRING function block is a simple and efficient module for converting unsigned 32-bit integers to strings. Its clear interface and simple functionality make it ideal for basic conversion tasks in automation projects.

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

F_UDINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UDINT_AS_WSTRING` converts an unsigned 32-bit integer value (`UDINT`) into a Unicode string (`WSTRING`). This functionality is particularly useful when numeric values are needed as text for display or further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The unsigned 32-bit integer value (`UDINT`) to be converted to a Unicode string.

Data Outputs

- **OUT**: The resulting Unicode string (`WSTRING`) after conversion.

Adapters

No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The algorithm converts the value of `IN` into a `WSTRING` representation and stores the result in `OUT`. The event `CNF` is then triggered to signal the completion of the operation.

Technical Features

- The conversion is performed directly without additional parameters.
- The function block is optimized for use in IEC 61131-compliant environments.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Completed**: Signals completion with `CNF` and returns to the idle state.

Application Scenarios

- Displaying numeric values in user interfaces.

- Logging numeric data to text files.
- Preparing data for communication via text-based protocols.

⚖️ Comparison with Similar Function Blocks

- Compared to `F_DINT_AS_WSTRING`, this function block processes unsigned values.
- Similar to `F_UDINT_AS_STRING`, but with Unicode support.

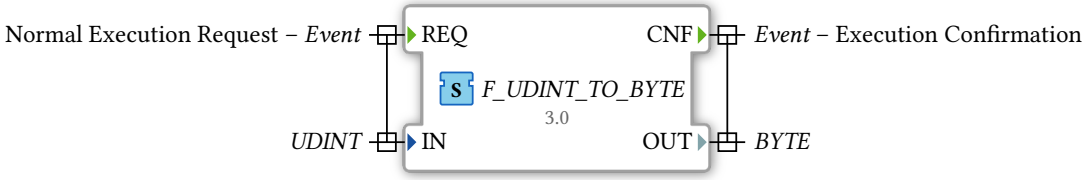
Conclusion

The `F_UDINT_AS_WSTRING` function block provides a simple and efficient way to convert unsigned 32-bit integer values to Unicode strings. Its clear interface and straightforward functionality make it a valuable tool in many automation applications.

F_UDINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

The function block `F_UDINT_TO_BYTE` converts a 32-bit unsigned integer (UDINT) value to an 8-bit unsigned integer (BYTE) value. This conversion is useful when data needs to be exchanged between systems with different word lengths or when memory space needs to be optimized.



- **REQ**: Starts the conversion. This input is linked to the data input `IN`.
- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.
- **IN**: The 32-bit unsigned integer value (UDINT) to be converted.
- **OUT**: The resulting 8-bit unsigned integer value (BYTE) after the conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters are present.

Functionality

The function block performs the conversion when the event `REQ` is received. The value at input `IN`, of type UDINT, is converted to a BYTE value and output at `OUT`. The event `CNF` signals the successful completion of the conversion.

- The conversion is performed by the function `UDINT_TO_BYTE`, which maps the UDINT value to a BYTE value. Data loss can occur if the original value is outside the representable range of a BYTE (0 to 255).
- The function block is implemented as a simple function block (SimpleFB) and uses an ST algorithm for the conversion.

1. **Idle State**: The function block waits for the event `REQ`.
2. **Conversion State**: Upon receiving `REQ`, the conversion is performed, and `CNF` is triggered.

- **Data Reduction**: When only the lower 8 bits of a 32-bit value are relevant.
- **Interface Adaptation**: For communication with systems that can only process BYTE data.
- **Memory Optimization**: In environments with limited memory.
- **F_DINT_TO_BYTE**: Converts a 32-bit signed integer (DINT) to a BYTE value.

- **F_INT_TO_BYTE**: Converts a 16-bit signed integer (INT) to a BYTE value.
- **F_UDINT_TO_WORD**: Converts a 32-bit unsigned integer (UDINT) to a 16-bit unsigned integer (WORD).

The function block `F_UDINT_TO_BYTE` provides a simple and efficient way to convert 32-bit unsigned integer values to 8-bit unsigned integer values. It is particularly useful in scenarios where memory or communication bandwidth needs to be optimized. However, be aware of the potential data loss with values outside the byte range.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

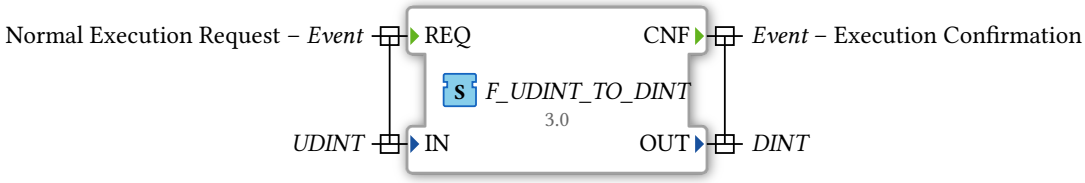
Conclusion

F_UDINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UDINT_TO_DINT` converts an unsigned 32-bit integer value (UDINT) to a signed 32-bit integer value (DINT). This conversion is useful in scenarios where data is exchanged between systems that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (UDINT): The unsigned 32-bit integer value to be converted.

Data Outputs

- **OUT** (DINT): The resulting signed 32-bit integer value after conversion.

Adapters

No adapters are present.

Functionality

The function block performs the conversion by transforming the value of input `IN` from type `UDINT` to type `DINT`. The conversion occurs synchronously when the event `REQ` is triggered. After the conversion is complete, the event `CNF` is triggered, and the converted value is available at output `OUT`.

Technical Features

- The conversion is performed directly without additional processing.
- The function block is suitable for use in real-time systems.
- The implementation is in Structured Text (ST).

State Overview

1. **Idle State**: Waits for the event `REQ`.
2. **Conversion State**: Performs the conversion when `REQ` is triggered.
3. **Acknowledgement State**: Triggers `CNF` as soon as the conversion is complete.

Application Scenarios

- Data conversion in control systems.
- Interfaces between systems that use different data types.
- Real-time data processing requiring type conversions.

⚖️ Comparison with similar function blocks

- Compared to generic conversion function blocks, `F_UDINT_TO_DINT` specializes in converting between `UDINT` and `DINT`.
- Other function blocks might offer additional features such as range checks or error handling, which is not the case here.

Conclusion

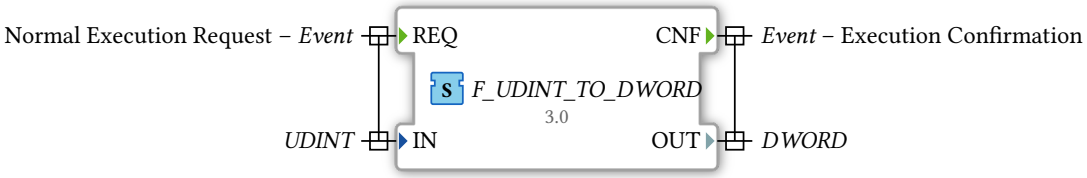
The function block `F_UDINT_TO_DINT` is a simple and efficient function block for converting between unsigned and signed 32-bit integers. It is particularly useful in scenarios where fast and direct type conversion is required.

F_UDINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UDINT_TO_DWORD` converts a `UDINT` value (32-bit unsigned integer) into a `DWORD` value (32-bit data word). This conversion is particularly useful in scenarios where numeric values need to be further processed or transmitted as bit patterns.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (`UDINT`): The 32-bit unsigned integer value to be converted.

Data Outputs

- **OUT** (`DWORD`): The result of the conversion as a 32-bit data word.

Adapters

No adapters available.

Functionality

When the `REQ` event is triggered, the input value `IN` of type `UDINT` is converted to a `DWORD` value and output `OUT`. The `CNF` event signals the successful completion of the conversion.

The underlying algorithm is implemented in Structured Text (ST):

ALGORITHM REQ OUT := UDINT_TO_DWORD(IN); END_ALGORITHM

Technical Features

- The conversion is performed directly through type conversion, without changing the bit representation.
- The function block is deterministic and has no internal states.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of numeric values for bitwise processing.
- Compatibility with systems that expect `DWORD` data.
- Data preparation for communication protocols or hardware registers.

⚖️ Comparison with Similar Function Blocks

- Unlike arithmetic function blocks, `F_UDINT_TO_DWORD` does not change the value, but only the data type.

Similar conversion blocks exist for other data types (e.g., `INT_TO_DINT`), but with different source and target types.

🛠️ Related Exercises

- Exercise_034b
- Exercise_152
- Exercise_153

Conclusion

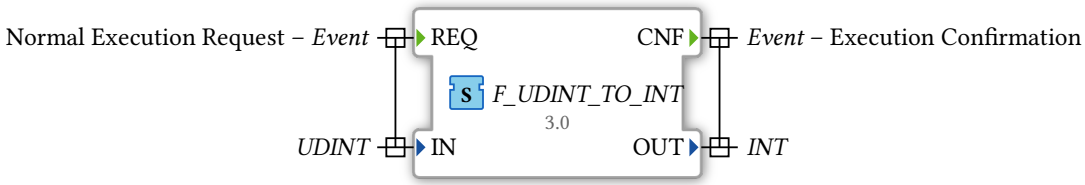
F_UDINT_TO_DWORD` is a simple yet essential function block for type conversions in IEC 61499-based control systems. Its deterministic operation and clear interface make it a reliable tool for data transformations.

F_UDINT_TO_INT

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_UDINT_TO_INT` converts a `UDINT` value (32-bit unsigned integer) to a `INT` value (16-bit signed integer). This conversion is useful when data needs to be exchanged between systems or components that use different data types.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. Triggered after successful conversion.

Data Inputs

- **IN** (`UDINT`): The input value to be converted.

Data Outputs

- **OUT** (`INT`): The converted output value.

Adapters

- No adapters available.

Functionality

This function block performs the conversion from `UDINT` to `INT` when the `REQ` event is received. The algorithm uses the built-in function `UDINT_TO_INT` to perform the conversion. After successful conversion, the `CNF` event is triggered, and the converted value is available at the `OUT` output.

Technical Features

- Data loss can occur if the `UDINT` value is outside the representable range of `INT` (i.e., greater than 32767 or less than −32768).
- The function block is simple and has no internal states or complex logic.

State Overview

The function block has no internal state. It responds exclusively to the `REQ` event and performs the conversion immediately.

Application Scenarios

- Data conversion in control systems where different data types are used.
- Integration of components that expect or deliver different data types.
- Logging or displaying data originally stored as `UDINT` but intended to be displayed as `INT`.

⚖️ Comparison with similar building blocks

- **F_DINT_TO_INT**: Converts `DINT` (32-bit signed) to `INT`.
- **F_UDINT_TO_DINT: `Konvertiert UDINTzuDINT(32-Bit vorzeichenbehaftet), ohne Datenverlust, da der Wertebereich vonDINT` is larger.
- **F_INT_TO_UDINT**: Performs the reverse conversion (`INT` to `UDINT`).

Conclusion

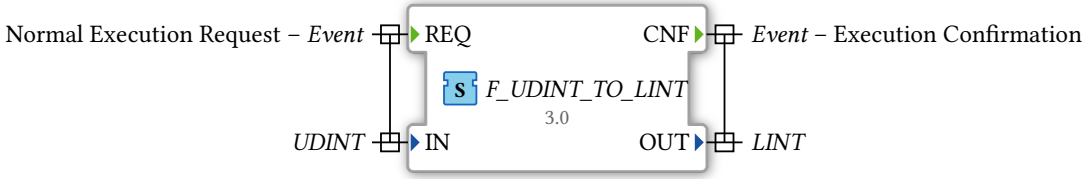
The `F_UDINT_TO_INT` function block is a simple and efficient tool for converting `UDINT` to `INT` values. It is particularly useful in scenarios where such a type conversion is required; however, potential data loss with large values should be considered.

F_UDINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UDINT_TO_LINT` converts a 32-bit unsigned integer value (UDINT) to a 64-bit signed integer value (LINT). This block is part of the `iec61131::conversion` package and enables simple and efficient type conversion between these two data types.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (UDINT): The 32-bit unsigned integer value to be converted.

Data Outputs

- **OUT** (LINT): The resulting 64-bit signed integer value after the conversion.

Adapters

This function block does not use adapters.

Operation

When the event input `REQ` is triggered, the value of the data input `IN` (UDINT) is copied to the data output `OUT` (LINT). The conversion is performed directly because LINT covers a larger range of values than UDINT and can therefore represent all possible UDINT values without data loss. After the conversion is complete, the event `CNF` is triggered.

Technical Features

- **Algorithm**: The function block uses a simple ST algorithm that directly assigns the input value to the output value.
- **Data Loss**: Since LINT has a larger range of values than UDINT, there is no data loss during this conversion.
- **Performance**: The conversion is very efficient and takes place in constant time.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- **Type Conversion**: In scenarios where an unsigned 32-bit value needs to be converted to a signed 64-bit value, e.g., when processing sensor data or in mathematical calculations.
- **Data Compatibility**: When integrating systems that use different integer types.

⚖️ Comparison with Similar Components

- **F_UDINT_TO_DINT**: Converts UDINT to DINT (32-bit signed), but can cause overflow because DINT has a smaller range of values.
- **F_UDINT_TO_ULINT**: Converts UDINT to ULINT (64-bit unsigned), but retains the unsigned nature.

🛠️ Related Exercises

- Exercise_072c

Conclusion

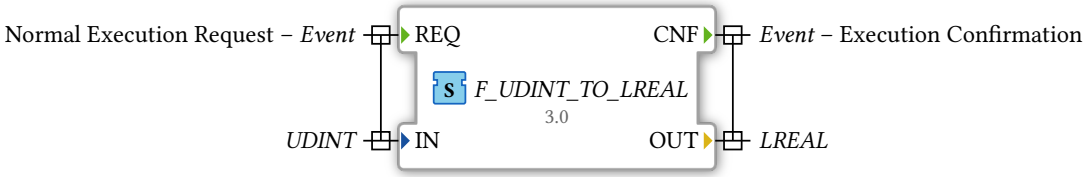
The `F_UDINT_TO_LINT` function block provides a simple and efficient way to convert unsigned 32-bit integer values to signed 64-bit integer values. Its direct and lossless conversion makes it ideal for applications that require an extended range of values.

F_UDINT_TO_LREAL

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_UDINT_TO_LREAL` converts a 32-bit unsigned integer value (UDINT) into a 64-bit floating-point value (LREAL). This block is particularly useful in applications where numerical values of varying precision need to be processed.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is processed.

Event Outputs
- **CNF**: Signals the completion of the conversion. The converted value is available at data output `OUT`.

Data Inputs
- **IN** (UDINT): The unsigned 32-bit integer value to be converted.

Data Outputs
- **OUT** (LREAL): The resulting 64-bit floating-point value after conversion.

Adapters
This function block does not use any adapters.

Operation
When event input `REQ` is triggered, the value at input `IN` (of type UDINT) is read and converted to an LREAL value. The result is output at `OUT`, and event `CNF` signals successful conversion.

Technical Features
- The algorithm is implemented directly in ST (Structured Text) and performs a simple type conversion.
- No additional error handling or limit checks are required, as UDINT values can be directly converted to LREAL.

State Overview
1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Converts `IN` to `OUT`.
3. **Completed**: Sends `CNF` and returns to the idle state.

Application Scenarios
- Data processing in control systems where different numeric formats need to be processed.
- Integration into larger systems that require both integer and floating-point operations.

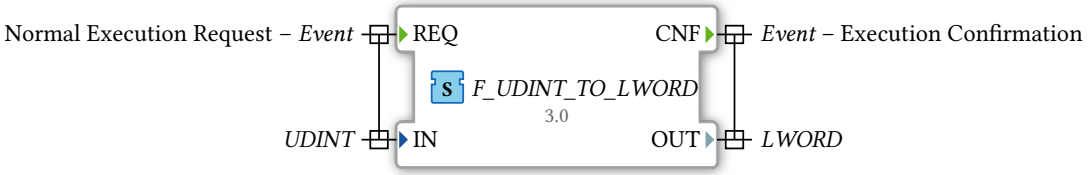
⚖️ Comparison with Similar Function Blocks
- Compared to generic conversion blocks, `F_UDINT_TO_LREAL` specializes in converting UDINT to LREAL and is therefore more efficient in this specific application.
- Other blocks might offer additional functionalities such as range checks or scaling, which are not necessary here.

Conclusion
The `F_UDINT_TO_LREAL` function block is a simple yet effective block for the specific conversion of unsigned 32-bit integers to 64-bit floating-point numbers. Its simplicity and direct implementation make it a reliable choice for such applications.

F_UDINT_TO_LWORD

• − ◦ • − ◦ • − ◦ •

Introduction
The function block `F_UDINT_TO_LWORD` converts a 32-bit unsigned integer value (UDINT) to a 64-bit unsigned integer value (LWORD). This block is particularly useful in scenarios where an extension of the data bit width is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN**: The input for the UDINT value to be converted.

Data Outputs

- **OUT**: The output for the converted LWORD value.

Adapters

No adapters are present.

Functionality

The function block performs the UDINT to LWORD conversion by expanding the incoming 32-bit value to a 64-bit value. The conversion occurs each time the `REQ` event occurs and is confirmed by the `CNF` event.

Technical Features

- The function block uses the built-in function `UDINT_TO_LWORD` for the conversion.
- This is a simple function block without a state machine or complex logic.

State Overview

Because it is a simple function block, there are no state transitions or internal states.

Application Scenarios

- Extending data widths in control systems.
- Compatibility with systems that use different bit widths.
- Data processing in industrial automation solutions.

⚖️ Comparison with Similar Blocks

Compared to other conversion blocks such as `F_DINT_TO_LWORD` or `F_UINT_TO_LWORD`, this one is specifically optimized for converting UDINT to LWORD. It offers a direct and efficient solution for this specific conversion.

Conclusion

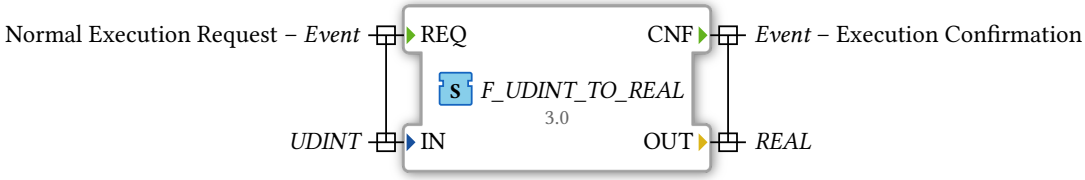
The `F_UDINT_TO_LWORD` function block is a simple and efficient building block for converting 32-bit unsigned integer values to 64-bit unsigned integer values. Its clear interface structure and simple functionality make it a reliable component in industrial automation solutions.

F_UDINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UDINT_TO_REAL` converts an unsigned 32-bit integer value (`UDINT`) into a 32-bit floating-point value (`REAL`). This block is particularly useful in scenarios where numeric values of different data types need to be processed.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at the data output `OUT`.

Data Inputs

- **IN**: Expects an input value of type `UDINT` (unsigned 32-bit integer).

Data Outputs

- **OUT**: Outputs the converted value as `REAL` (32-bit floating-point number).

Adapters

- No adapters available.

Functionality

When the event input `REQ` is triggered, the algorithm is started. This algorithm converts the `UDINT` value at input `IN` into a `REAL` value and provides the result at output `OUT`. The event `CNF` is then triggered to signal the successful completion of the conversion.

Technical Features

- The conversion is performed directly using the built-in function `UDINT_TO_REAL`.
- The function block is simple and efficient because it does not require complex state machines or additional processing steps.

State Overview

The function block has no internal states. Processing occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of raw data (e.g., from sensors or communication protocols) into floating-point values for further calculations.
- Integration into control systems where different data types need to be processed.

⚖️ Comparison with Similar Function Blocks

- Compared to generic conversion function blocks, `F_UDINT_TO_REAL` is specialized and optimized for the specific conversion of `UDINT` to `REAL`.
- Similar function blocks might offer additional features such as range checks or scaling, which is not the case here.

🛠️ Related Exercises

- Exercise_151
- Exercise_151_AX
- Exercise_152
- Exercise_153

Conclusion

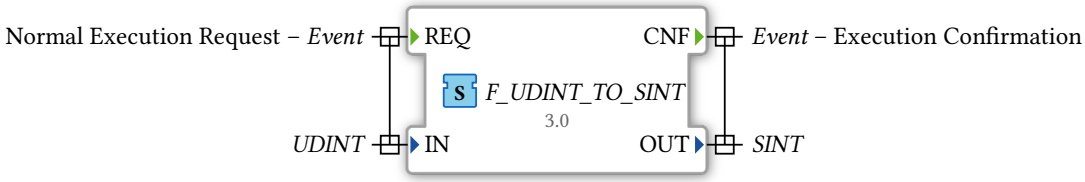
The `F_UDINT_TO_REAL` function block is a simple and efficient building block for converting unsigned 32-bit integers to floating-point numbers. Its simplicity makes it particularly suitable for applications where fast and direct conversion is required.

F_UDINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UDINT_TO_SINT` converts a 32-bit unsigned integer value (UDINT) to an 8-bit signed integer value (SINT). This conversion is useful when data needs to be exchanged between systems or components that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs

- **IN** (UDINT): The unsigned 32-bit integer value to be converted.

Data Outputs

- **OUT** (SINT): The resulting signed 8-bit integer value after conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion when the event `REQ` is triggered. The algorithm converts the value of `IN` (UDINT) to a SINT value and stores the result in `OUT`. The event `CNF` is then triggered to signal the completion of the operation.

Technical Features

- The function block is simple and deterministic, as it only performs a direct type conversion.
- There is no additional logic or error handling for values that are outside the representable range of SINT.

State Overview

The function block has no internal state. The conversion occurs immediately upon triggering `REQ`.

Application Scenarios

- Data conversion in control systems where different data types are used.
- Interfaces between systems that require different integer formats.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_DINT_TO_SINT` or `F_INT_TO_SINT` perform similar conversions, but with different output or input data types.
- `F_UDINT_TO_SINT` is specifically for converting UDINT to SINT.

Conclusion

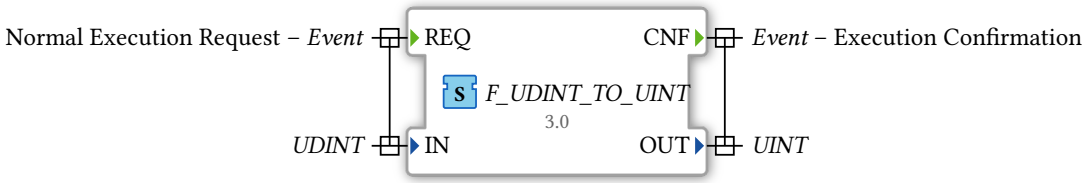
The `F_UDINT_TO_SINT` function block is a simple and efficient tool for converting unsigned 32-bit integers to signed 8-bit integers. It is particularly useful in scenarios where such type conversions are needed regularly.

F_UDINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UDINT_TO_UINT` converts a 32-bit unsigned integer value (UDINT) to a 16-bit unsigned integer value (UINT). This block is part of the `iec61131::conversion` package.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion of the input value. The input value `IN` is passed with this event.

Event Outputs

- **CNF**: Signals successful conversion and outputs the converted value `OUT`.

Data Inputs

- **IN** (UDINT): The 32-bit unsigned integer value to be converted.

Data Outputs

- **OUT** (UINT): The converted 16-bit unsigned integer value.

Adapters

This function block does not have any adapters.

Operation

Upon receiving the event `REQ`, the input value `IN` of type UDINT is converted to type UINT and output via `OUT`. Simultaneously, the event `CNF` is triggered to signal the successful completion of the conversion.

Technical Features

- The conversion is performed by direct type conversion, whereby the higher-order bits of the UDINT value are truncated if the value is outside the representable range of UINT.
- The block is suitable for use in real-time systems.

State Overview

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Conversion of 32-bit values to 16-bit values in control applications.
- Use in systems that need to work with different integer sizes.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_UDINT_TO_UINT` is specialized for converting UDINT to UINT and is therefore more efficient in execution.
- Similar blocks like `F_DINT_TO_INT` convert between other integer types.

🛠️ Related Exercises

- Exercise_103

Conclusion

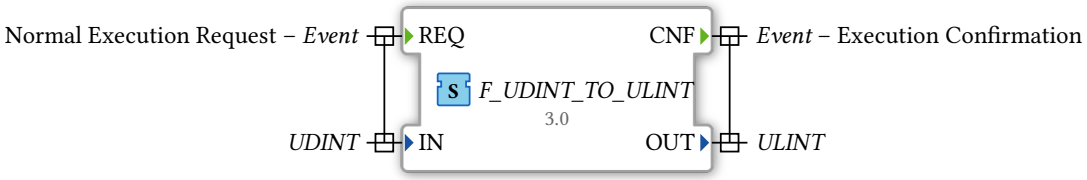
The F_UDINT_TO_UINT function block is a specialized and efficient building block for converting 32-bit to 16-bit unsigned integer values. It is particularly suitable for applications where fast and direct type conversion is required.

F_UDINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_UDINT_TO_ULINT converts a 32-bit unsigned integer (UDINT) value to a 64-bit unsigned integer (ULINT). This block is particularly useful when exchanging data between systems or components that use different integer sizes.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the function block reads the value at the data input IN and performs the conversion.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is activated along with the converted value at data output OUT.

Data Inputs

- **IN**: The input for the 32-bit unsigned integer (UDINT) value to be converted.

Data Outputs

- **OUT**: The output for the converted 64-bit unsigned integer (ULINT) value.

Adapters

This function block does not use adapters.

Operation

The function block performs a direct type conversion. When the REQ event is triggered, the value of IN is read and assigned to OUT without further processing. Since ULINT covers a larger range of values than UDINT, this conversion is always safe and lossless.

The algorithm is implemented as follows:
ALGORITHM REQ OUT := IN; END_ALGORITHM

Technical Features

- **Lossless Conversion**: Since ULINT can represent all values of UDINT, there is no data loss.
- **Simple Implementation**: The function block requires no additional logic or error handling.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the REQ event.

Application Scenarios

- **Data Compatibility**: Used when systems need to communicate with different integer sizes.
- **Extended Value Range**: Enables the use of UDINT values in contexts that require ULINT.

⚖️ Comparison with Similar Function Blocks

- **F_UDINT_TO_LINT**: Converts UDINT to a 64-bit signed integer (LINT). In contrast, `F_UDINT_TO_ULINT` retains the unsigned character.
- **F_DINT_TO_ULINT**: Converts a 32-bit signed integer (DINT) to ULINT. `F_UDINT_TO_ULINT` is specific for unsigned values.

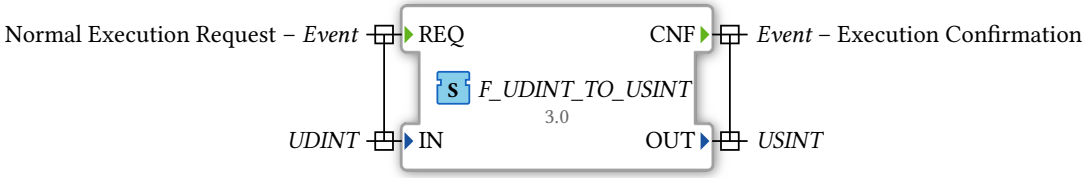
Conclusion

The `F_UDINT_TO_ULINT` function block is a simple and efficient tool for safely converting 32-bit unsigned integers to 64-bit unsigned integers. Its lossless operation and ease of use make it ideal for applications requiring an extended range of values.

F_UDINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

The function block `F_UDINT_TO_USINT` converts a 32-bit unsigned integer value (UDINT) to an 8-bit unsigned integer value (USINT). This block is particularly useful in scenarios where type conversion between numerical values of different sizes is required.



- **REQ**: Starts the conversion. This input is connected to the data input `IN`.
- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.
- **IN**: The input expects a 32-bit unsigned integer value (UDINT) to be converted.
- **OUT**: The output provides the converted 8-bit unsigned integer value (USINT).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is received. The algorithm converts the value of `IN` (UDINT) to the corresponding `OUT` (USINT) value. After successful conversion, the event `CNF` is triggered.

- The conversion is performed by the built-in function `UDINT_TO_USINT`.
- No range check is performed. Values outside the USINT range (0 to 255) may produce unexpected results.

The function block has no internal states. The conversion is performed anew with each `REQ` event.

- Type conversion in control applications where memory optimization is crucial.
- Communication between systems using different data types.
- Compared to generic conversion blocks, `F_UDINT_TO_USINT` is specialized and therefore more efficient in execution.
- Other blocks might offer additional functionalities such as range checks, which is not the case here.

The `F_UDINT_TO_USINT` function block is a simple and efficient building block for the specific conversion of UDINT to USINT values. It is particularly suitable for applications where fast and direct type conversion is required.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

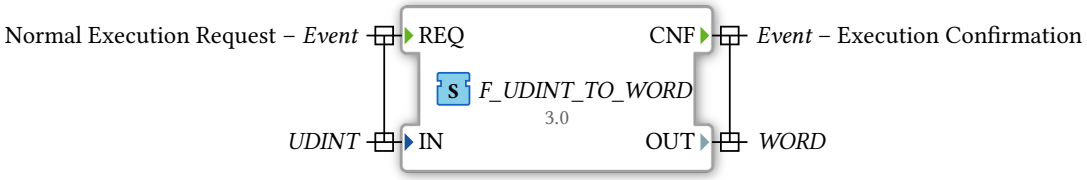
Conclusion

F_UDINT_TO_WORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UDINT_TO_WORD` converts a 32-bit unsigned integer value (UDINT) to a 16-bit unsigned integer value (WORD). This conversion is particularly necessary in scenarios where data needs to be exchanged between systems with different word lengths.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (UDINT): The 32-bit unsigned integer value to be converted.

Data Outputs

- **OUT** (WORD): The resulting 16-bit unsigned integer value after conversion.

Adapters

No adapters are available.

Functionality

This function block performs the conversion from UDINT to WORD by executing the algorithm `REQ`. The input value `IN`, of type UDINT, is converted into a WORD value and output at `OUT`. The conversion occurs synchronously with the event `REQ`, and the result is confirmed with the event `CNF`.

Technical Details

- The conversion truncates the 16 most significant bits of the UDINT value, since a WORD only contains 16 bits.
- The function block is part of the package `iec61131::conversion`.

State Overview

The function block has no internal state. The conversion is re-executed with each event `REQ`.

Application Scenarios

- Data reduction from 32-bit to 16-bit in embedded systems.
- Compatibility with legacy systems that can only process 16-bit data.
- Memory optimization in resource-constrained environments.

⚖️ Comparison with Similar Function Blocks

- Compared to `F_DINT_TO_WORD` or `F_INT_TO_WORD`, this function block processes unsigned values.
- Similar function blocks like `F_UDINT_TO_DWORD` do not perform bit width reduction.

Conclusion

The `F_UDINT_TO_WORD` function block provides a simple and efficient way to convert 32-bit unsigned integer values to 16-bit unsigned integer values. It is particularly useful in scenarios where a reduction in data width is required without altering the fundamental semantics of the data.

UDINT conversions

🛠️ Related exercises

- Uebung_009
- Uebung_009_AX
- Uebung_009a
- Uebung_011
- Uebung_011a
- Uebung_011a2
- Uebung_012
- NumericValue_ID_TO_NVS
- Uebung_012b
- Uebung_012c
- Uebung_015
- Uebung_015a
- Uebung_020c2_sub
- Uebung_028
- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_034b
- Uebung_072c
- Uebung_083
- Uebung_083_AX
- Uebung_103
- Uebung_111
- Uebung_151_AX
- Uebung_152
- Uebung_153

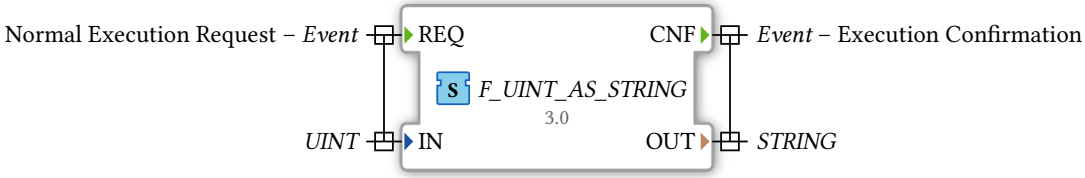
UINT

F_UINT_AS_STRING

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_UINT_AS_STRING` converts an unsigned integer value (`UINT`) into a string (`STRING`). This functionality is particularly useful when numeric values are needed as text for output, logging, or further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is activated along with the converted value at data output `OUT`.

Data Inputs

- **IN** (`UINT`): The unsigned integer value to be converted to a string.

Data Outputs

- **OUT** (`STRING`): The string generated from the input value `IN`.

Adapters

This function block has no adapters.

Functionality

When the event input `REQ` is triggered, the algorithm is started, which converts the value of `IN` to `STRING` using the function `UINT_AS_STRING`. The result is output to `OUT`, and the event `CNF` signals successful conversion.

Technical Features

- The conversion is performed directly and without delay.
- The function block is implemented simply and efficiently, without additional states or complex logic.

State Overview

The function block has no internal state. Each execution is independent of previous calls.

Application Scenarios

- Displaying numerical values in user interfaces.
- Logging measured values to text files.
- Preparing data for communication via text-based protocols.

⚖️ Comparison with similar building blocks

- Unlike generic conversion blocks, `F_UINT_AS_STRING` specializes in converting `UINT` to `STRING`.
- Other blocks might support additional parameters such as formatting or base (decimal, hexadecimal), which is not the case here.

Conclusion

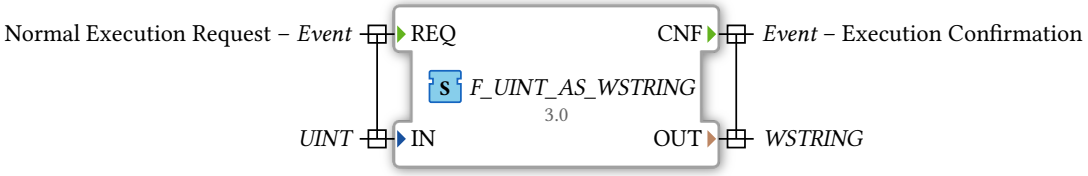
The `F_UINT_AS_STRING` function block is a simple and effective tool for converting unsigned integer values to strings. Its strength lies in its simplicity and direct applicability in scenarios where a fast and straightforward conversion is required.

F_UINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_AS_WSTRING` converts an unsigned integer value (`UINT`) into a wide-string representation (`WSTRING`). This block is particularly useful in scenarios where numeric values need to be converted into human-readable text.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event is triggered, the value at data input `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event, along with the converted value, is output at data output `OUT`.

Data Inputs

- **IN**: The input for the `UINT` value to be converted.

Data Outputs

- **OUT**: The output for the converted `WSTRING` value.

Adapters

This function block does not use adapters.

Functionality

When the `REQ` event is triggered, the algorithm is executed, converting the `UINT` value at input `IN` into a `WSTRING` value. The result is provided at output `OUT`, and the `CNF` event signals successful conversion.

Technical Features

- The function block is implemented simply and efficiently, without additional states or complex logic.
- The conversion is performed directly via the built-in function `UINT_AS_WSTRING`.

State Overview

The function block has no internal states. Processing occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Displaying numeric values in user interfaces.
- Logging and debugging, where numeric values need to be output as text.
- Communication with systems that expect string data.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_UINT_AS_WSTRING` specializes in converting `UINT` to `WSTRING` and is therefore more efficient for this specific use case.
- Other blocks might offer additional features such as formatting or locale adjustment, which is not the case here.

Conclusion

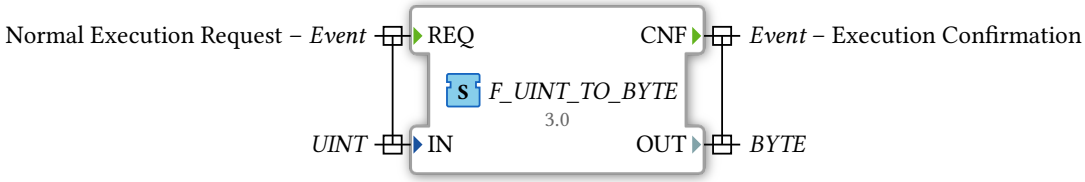
The `F_UINT_AS_WSTRING` function block is a simple and effective block for converting unsigned integer values to wide strings. Its simplicity and direct functionality make it ideal for applications that require quick and straightforward conversion.

F_UINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_BYTE` converts an unsigned 16-bit integer value (UINT) to an 8-bit byte value (BYTE). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input for the UINT value to be converted.

Data Outputs

- **OUT**: The output for the converted BYTE value.

Adapters

No adapters are present.

Functionality

The function block performs the conversion as soon as an event arrives at input `REQ`. The value at input `IN`, of type UINT, is converted into a BYTE value and output at output `OUT`. The conversion is performed using the function `UINT_TO_BYTE`. After the conversion is complete, event `CNF` is triggered.

Technical Features

- The conversion is performed without additional parameters or configuration.
- The function block is deterministic and has no internal states.

State Overview

The function block has no internal states. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Type conversion in communication protocols where BYTE data is expected.
- Data reduction when only the lower 8 bits of a UINT value are relevant.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_UINT_TO_BYTE` is specialized in converting UINT to BYTE.
- Other blocks might offer additional features such as range checks or scaling, which is not the case here.

🛠️ Related Exercises

- Exercise_12x_sub

Conclusion

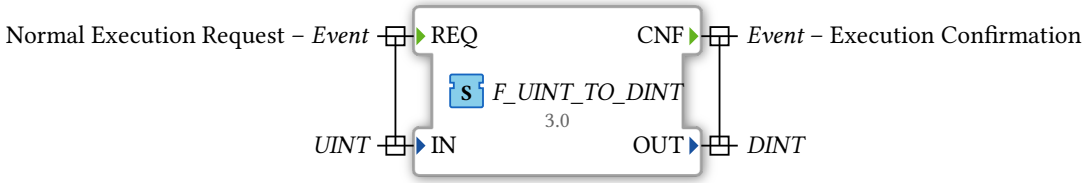
The `F_UINT_TO_BYTE` function block is a simple and efficient building block for the specific conversion of UINT to BYTE. It is particularly suitable for applications where fast and direct type conversion is required.

F_UINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_DINT` converts an unsigned 16-bit integer value (UINT) to a signed 32-bit integer value (DINT). This conversion is useful when exchanging data between systems that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `UINT` to be converted.

Data Outputs

- **OUT**: The output value of type `DINT`, containing the result of the conversion.

Adapters

This function block does not use any adapters.

Operation

When an event is received at input `REQ`, the value of `IN` is read and directly converted to the data type `DINT`. The result is output to `OUT`, and the event `CNF` is triggered to signal the completion of the conversion.

Technical Features

- The function block performs a direct type conversion without additional calculations or scaling.
- The conversion is lossless because the `DINT` type covers a larger range of values than `UINT`.

State Overview

1. **Idle**: Waits for an event at input `REQ`.
2. **Converting**: Performs the conversion and outputs the result.
3. **Completed**: Signals the completion of the conversion with the event `CNF`.

Application Scenarios

- Data conversion in control systems that use different data types.
- Integration of sensor data in the format `UINT` into systems that expect `DINT`.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_UINT_TO_DINT` is specialized and efficient for converting `UINT` to `DINT`.
- Other blocks might offer additional features such as scaling or range checking, which are not required here.

Conclusion

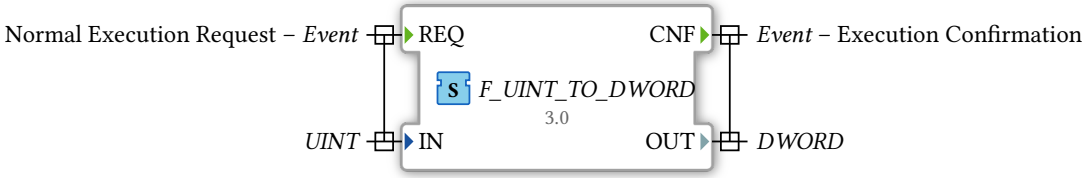
The `F_UINT_TO_DINT` function block is a simple and efficient building block for the direct conversion of `UINT` to `DINT`. It is particularly useful in scenarios where lossless type conversion is required.

F_UINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_DWORD` converts a UINT value (16-bit unsigned integer) to a DWORD value (32-bit unsigned integer). This conversion is particularly necessary in scenarios where data with different bit widths needs to be processed or transmitted.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: UINT value (16-bit) to be converted to a DWORD value.

Data Outputs

- **OUT**: DWORD value (32-bit) containing the result of the conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion as soon as an event arrives at input `REQ`. The algorithm `REQ` converts the UINT value at input `IN` to a DWORD value and stores the result in `OUT`. The event `CNF` is then triggered to signal the completion of the conversion.

Technical Features

- The conversion is performed without data loss, as the value range of UINT (0 to 65535) can be fully mapped to DWORD (0 to 4294967295).
- The function block is part of the `iec61131::conversion` package.

State Overview

The function block has no internal state. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Data conversion in control systems when different bit widths need to be harmonized.
- Integration into larger function block networks that process both UINT and DWORD data.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks (e.g., `F_INT_TO_DINT`), `F_UINT_TO_DWORD` specializes in converting 16-bit to 32-bit unsigned integers.
- Similar blocks might offer additional features such as scaling or limit checks, which is not the case here.

Conclusion

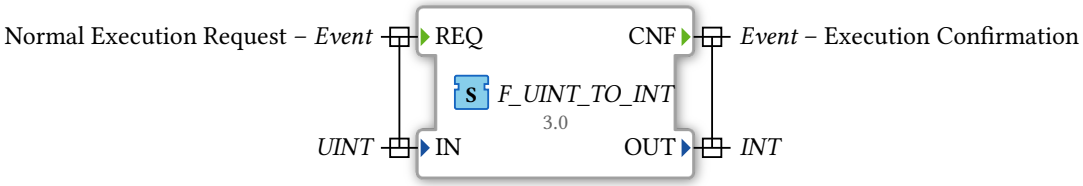
The `F_UINT_TO_DWORD` function block provides a simple and efficient way to convert UINT values to DWORD values. Its clear interface and deterministic operation make it ideal for use in control and automation solutions.

F_UINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_INT` converts an unsigned integer value (UINT) to a signed integer value (INT). This functionality is particularly useful when exchanging data between systems that use different integer formats.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the function block reads the value at the data input `IN` and performs the conversion.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. Simultaneously, the converted value is output at data output `OUT`.

Data Inputs

- **IN**: The unsigned integer value (UINT) to be converted is entered here.

Data Outputs

- **OUT**: Outputs the converted signed integer value (INT).

Adapters

This function block does not have any adapter interfaces.

Functionality

When the event input `REQ` is triggered, the function block's algorithm is activated. It reads the value at input `IN`, converts it from UINT to INT, and outputs the result at output `OUT`. The event `CNF` is then triggered to signal the successful completion of the conversion.

The conversion is performed using the built-in function `UINT_TO_INT()`.

Technical Features

- The function block has a simple structure and contains only one algorithm.
- No additional libraries or adapters are required.
- The conversion is performed directly and without delay.

State Overview

The function block has no internal states. The conversion is performed anew each time `REQ` is triggered.

Application Scenarios

- Data conversion between systems that use different integer formats.
- Preprocessing of sensor data that is delivered as UINT but is to be further processed as INT.
- Establishing compatibility between different protocols or interfaces.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_UINT_TO_INT` is specialized and therefore more efficient.
- Similar blocks like `F_INT_TO_UINT` perform the reverse conversion.
- General-purpose blocks like `F_ANY_TO_ANY` offer more flexibility but may be less performant.

Conclusion

The `F_UINT_TO_INT` function block is a simple yet effective tool for converting between unsigned and signed integer values. Its simplicity and straightforward functionality make it a reliable solution for relevant use cases.

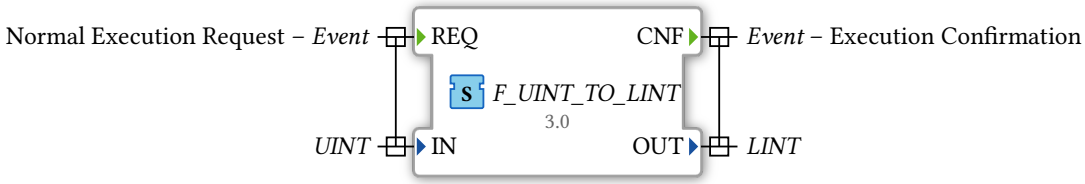
F_UINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_LINT` converts an unsigned 16-bit integer value (UINT) to a signed 64-bit integer value (LINT). This conversion is particularly useful when data needs to be exchanged or processed between systems with different word lengths.

• – ◦ • – ◦ • – ◦ •



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `UINT`, which is to be converted to a value of `LINT`.

Data Outputs

- **OUT**: The converted output value of type `LINT`.

Adapters

- No adapters are available.

• – ◦ • – ◦ • – ◦ •

Functionality

The function block performs a direct assignment of the input value `IN` to the output `OUT`. Since `LINT` has a larger value range than `UINT`, the conversion is lossless. The block is activated by the event `REQ` and signals the completion of the operation with the event `CNF`.

• – ◦ • – ◦ • – ◦ •

Technical Features

- **Lossless Conversion**: Since `LINT` (64 bits) covers a larger value range than `UINT` (16 bits), no information is lost during the conversion.
- **Simple Implementation**: The block uses a simple algorithm for direct value assignment.

• – ◦ • – ◦ • – ◦ • –

State Overview

1. **Idle State**: The block is waiting for the event `REQ`.
2. **Activation State**: Upon receiving `REQ`, the input value `IN` is read and converted.
3. **Acknowledgement State**: After successful conversion, the event `CNF` is triggered, and the block returns to the idle state.

• – ◦ • – ◦ • – ◦ •

Application Scenarios

- **Data Conversion**: In systems that need to process both 16-bit and 64-bit data.

- **Interface Adaptation**: For communication between devices with different word widths.
- **Data Processing**: In algorithms that require higher precision for calculations.

• – ◦ • – ◦ • – ◦ •

⚖️ Comparison with similar function blocks

- **F_UINT_TO_INT**: Converts `UINT` to `INT` (16 bits), but with a signed value.
- **F_UINT_TO_DINT**: Converts `UINT` to `DINT` (32 bits), offering a wider range of values than `INT`, but less than `LINT`.
- **F_UINT_TO_ULINT**: Converts `UINT` to `ULINT` (64 bits), remaining unsigned.

• – ◦ • – ◦ • – ◦ •

Conclusion

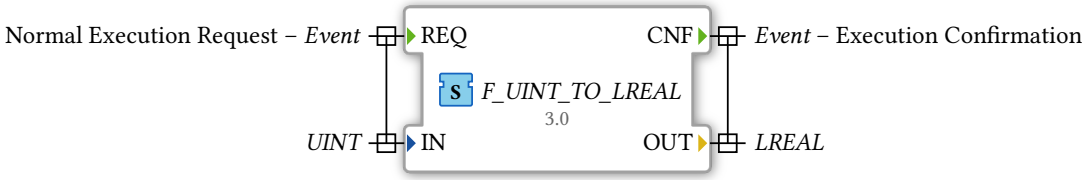
The `F_UINT_TO_LINT` function block is a simple and efficient tool for converting 16-bit unsigned integer values to 64-bit signed integer values. Its lossless conversion and ease of use make it ideal for applications requiring higher precision or compatibility with 64-bit systems.

F_UINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_LREAL` converts an unsigned 16-bit integer value (UINT) to a 64-bit floating-point value (LREAL). This conversion is particularly useful in control applications where different data types need to be processed.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. Upon this event, the value at input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at output `OUT`.

Data Inputs

- **IN** (UINT): The unsigned 16-bit integer value to be converted to an LREAL value.

Data Outputs

- **OUT** (LREAL): The resulting 64-bit floating-point value after conversion.

Adapters

No adapters are available.

Functionality

The function block performs the conversion by directly converting the UINT value from `IN` to an LREAL value and outputting it to `OUT`. The conversion occurs when the event `REQ` is triggered, and the event `CNF` signals completion.

Technical Features

- **Algorithm**: The algorithm is implemented in Structured Text (ST) and consists of a simple assignment (`OUT := IN`).
- **Data Types**: Supports conversion from UINT (0 to 65535) to LREAL (64-bit floating point).

State Overview

1. **Idle**: Waits for the event `REQ`.
2. **Conversion**: Upon `REQ`, the value from `IN` is read and converted.
3. **Completion**: The event `CNF` is triggered, and the converted value is available at `OUT`.

Application Scenarios

- **Measurement Processing**: Conversion of discrete sensor values (e.g., raw ADC data) into floating-point values for further calculations.
- **Data Communication**: Conversion of integer values for use in floating-point operations.

⚖️ Comparison with Similar Function Blocks

- **F_INT_TO_LREAL**: Converts a signed integer (INT) to LREAL.
- **F_DINT_TO_LREAL**: Converts a 32-bit integer (DINT) to LREAL.
- **F_ULINT_TO_LREAL**: Converts a 64-bit unsigned integer (ULINT) to LREAL.

Conclusion

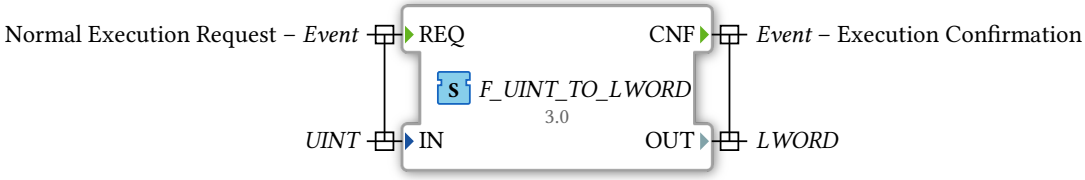
The `F_UINT_TO_LREAL` function block provides a simple and efficient way to convert unsigned 16-bit integer values to 64-bit floating-point values. Its clear interface and direct operation make it ideal for applications requiring fast and reliable type conversion.

F_UINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_LWORD` converts a `UINT` value (16-bit unsigned integer) to a `LWORD` value (64-bit unsigned integer). This conversion is particularly necessary in scenarios where data with different bit widths needs to be processed or transmitted.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `UINT`, which is to be converted to a `LWORD` value.

Data Outputs

- **OUT**: The converted output value of type `LWORD`.

Adapters

This function block has no adapters.

Operation

The function block performs the conversion using the algorithm `REQ`, which is activated when the event `REQ` is triggered. The algorithm converts the `UINT` input value `IN` into a `LWORD` value and assigns it to the output `OUT`. After the conversion is complete, the event `CNF` is triggered.

Technical Features

- The conversion is performed without data loss because the `LWORD` type covers a larger range of values than `UINT`.
- The function block is deterministic and has no internal states.

State Overview

Since this is a simple function block, there are no state transitions or internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Data conversion in control systems where different data types need to be processed.
- Integration into larger function block networks that process data with varying bit widths.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_UINT_TO_LWORD` specializes in converting `UINT` to `LWORD`, thus offering an optimized and type-safe solution.
- Other conversion blocks might offer additional features such as error handling or overflow checking, which is not the case here.

Conclusion

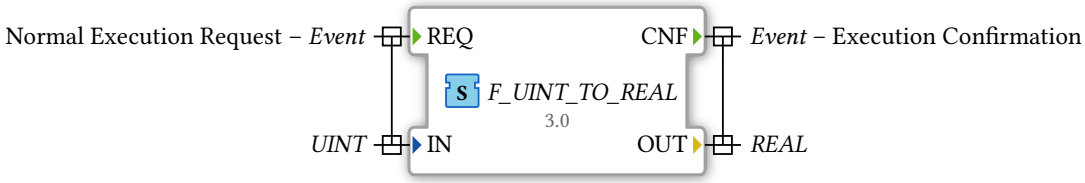
The `F_UINT_TO_LWORD` function block offers a simple and efficient way to convert `UINT` values to `LWORD` values. Its simplicity and determinism make it a reliable component in control applications. However, for more complex requirements, such as error handling, more advanced function blocks should be considered.

F_UINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_REAL` converts an unsigned integer value (`UINT`) into a floating-point number (`REAL`). This functionality is particularly useful when exchanging data between systems that use different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `UINT`, which is to be converted to a `REAL` value.

Data Outputs

- **OUT**: The output value of type `REAL`, which represents the result of the conversion.

Adapters

This function block has no adapters.

Operation

The function block performs the conversion as soon as the event `REQ` is triggered. The algorithm within the block directly assigns the value of `IN` to `OUT`, performing an implicit type conversion from `UINT` to `REAL`. Upon completion of the conversion, the event `CNF` is triggered.

Technical Features

- The conversion is performed without any additional parameters or settings.
- The block is part of the package `iec61131::conversion`.

State Overview

The function block has no internal state. The conversion occurs immediately upon the triggering of `REQ`.

Application Scenarios

- Conversion of sensor values stored as `UINT` to `REAL` for further calculations.
- Compatibility between systems using different data types.

⚖️ Comparison with Similar Blocks

- Unlike more complex conversion blocks, `F_UINT_TO_REAL` offers a simple, direct conversion without additional functionality.
- Similar blocks might offer additional features such as scaling or range checks, which is not the case here.

🛠️ Related Exercises

- Exercise_072c

Conclusion

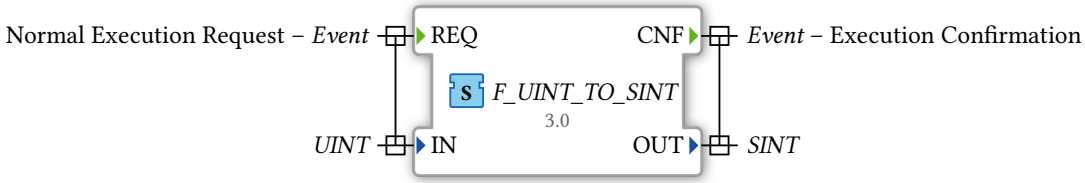
The `F_UINT_TO_REAL` function block is a simple and efficient tool for converting the type from `UINT` to `REAL`. Its strength lies in its simplicity and direct applicability, without the overhead of additional functions.

F_UINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_SINT` converts an unsigned 16-bit integer value (UINT) to a signed 8-bit integer value (SINT). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When activated, the input value `IN` is read and processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output value `OUT` is then available.

Data Inputs

- **IN** (UINT): The unsigned 16-bit integer value to be converted.

Data Outputs

- **OUT** (SINT): The resulting signed 8-bit integer value after the conversion.

Adapters

- No adapters are available.

Functionality

The function block performs the conversion by converting the `UINT` value at input `IN` to a `SINT` value and outputting it at output `OUT`. The conversion occurs when the `REQ` event is triggered, and the `CNF` event signals completion.

Technical Features

- Data loss can occur during the conversion if the `UINT` value is outside the representable range of `SINT` (0 to 127).
- The function block is implemented as a simple function block (SimpleFB) and uses an ST algorithm for the conversion.

State Overview

The function block has no internal states. The conversion occurs immediately upon the triggering of the `REQ` event.

Application Scenarios

- Type conversion in control applications, where `UINT` values need to be converted to `SINT` values.
- Data processing in industrial automation systems, where different data types are processed.

⚖️ Comparison with Similar Function Blocks

- Compared to generic conversion function blocks, `F_UINT_TO_SINT` specializes in converting `UINT` to `SINT`.
- Other conversion function blocks might offer additional features such as range checks or error handling.

Conclusion

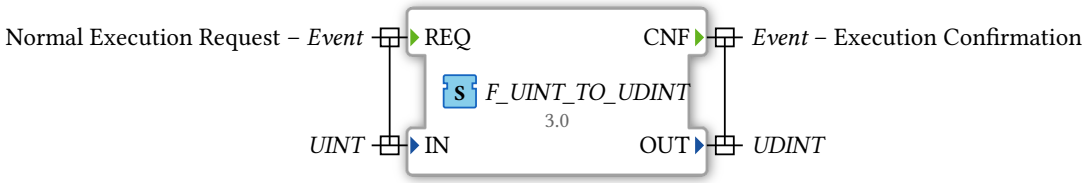
The `F_UINT_TO_SINT` function block is an efficient tool for the specific conversion of `UINT` to `SINT`. Its simplicity and direct functionality make it ideal for applications requiring fast and straightforward type conversion.

F_UINT_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_UDINT` converts a 16-bit unsigned integer value (`UINT`) to a 32-bit unsigned integer value (`UDINT`). This block is part of the `iec61131::conversion` package and is suitable for applications requiring an extension of the bit width of numeric values.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The 16-bit unsigned integer value to be converted (`UINT`).

Data Outputs

- **OUT**: The converted 32-bit unsigned integer value (`UDINT`).

Adapters

This function block does not use any adapters.

Functionality

When the event input `REQ` is triggered, the value of the data input `IN` is read and directly assigned to the data output `OUT`. Since `UDINT` has a larger bit width than `UINT`, the conversion occurs without data loss. The event `CNF` is then triggered to signal the completion of the operation.

Technical Features

- The conversion is lossless because `UDINT` completely covers the value range of `UINT`.
- The block is deterministic and has no internal states.

State Overview

The function block has no internal states. The operation is re-executed with each `REQ` event.

Application Scenarios

- Extending the bit width of sensor data for further processing.
- Ensuring compatibility between systems that use different integer formats.
- Preprocessing data for algorithms that require `UDINT`.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_UINT_TO_UDINT` is specialized and optimized for the specific conversion of `UINT` to `UDINT`.
- Other blocks, such as `F_INT_TO_DINT`, handle signed integers, while this block processes unsigned values.

🛠️ Related Exercises

- Exercise_015
- Exercise_015a

Conclusion

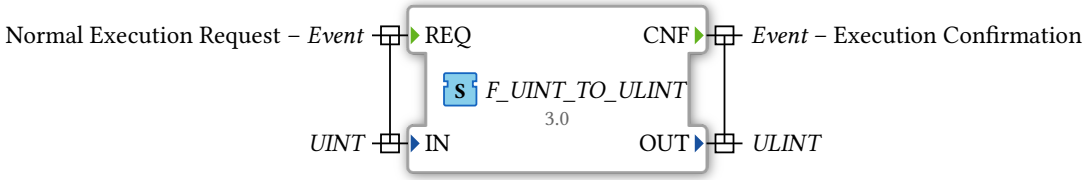
The `F_UINT_TO_UDINT` function block is an efficient and reliable tool for converting 16-bit to 32-bit unsigned integer values. Its simplicity and determinism make it ideal for real-time applications where performance and precision are critical.

F_UINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_ULINT` converts an unsigned 16-bit integer value (`UINT`) into an unsigned 64-bit integer value (`ULINT`). This conversion is particularly necessary when exchanging values between systems with different word widths.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `UINT` to be converted.

Data Outputs

- **OUT**: The converted output value of type `ULINT`.

Adapters

No adapters are available.

Functionality

This function block performs a direct conversion of the input value `IN` of type `UINT` to the output value `OUT` of type `ULINT`. The conversion occurs without changing the numerical value; only the data type is extended.

Technical Features

- The conversion is lossless because the `ULINT` type covers a larger range of values than `UINT`.
- The function block is deterministic and has no internal states.

State Overview

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Integration of systems with different word widths
- Data processing in controllers that need to handle both 16-bit and 64-bit data
- Data extension for subsequent calculations requiring higher precision

⚖️ Comparison with similar function blocks

- Unlike `F_INT_TO_LINT`, this function block processes unsigned data types
- Similar conversion function blocks exist for other data type combinations (e.g., `F_BYTE_TO_WORD`)

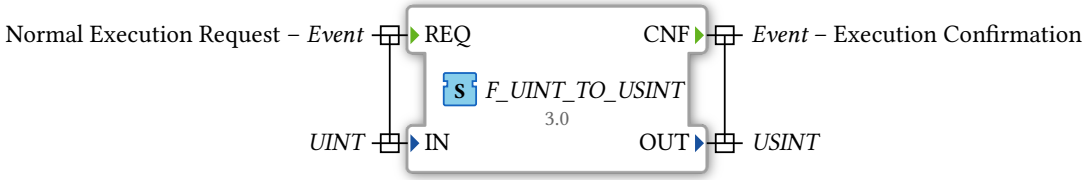
Conclusion

The `F_UINT_TO_ULINT` function block offers a simple and efficient way to convert 16-bit unsigned integers to 64-bit unsigned integers. Its simple structure and deterministic operation make it a reliable function block for systems with mixed word widths.

F_UINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

The function block `F_UINT_TO_USINT` converts an unsigned 16-bit integer value (UINT) to an unsigned 8-bit integer value (USINT). This block is part of the `iec61131::conversion` library and implements a simple type conversion.



- **REQ**: Starts the conversion process. This input is linked to the data input `IN`.
- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.
- **IN** (UINT): The unsigned 16-bit integer value to be converted.
- **OUT** (USINT): The resulting unsigned 8-bit integer value after conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use any adapters.

When the `REQ` event is triggered, the algorithm that converts the `IN` value from UINT to USINT is executed and outputs the result to `OUT`. The `CNF` event is then triggered to signal the successful completion of the operation. The conversion is performed by truncating the higher-order bits if the input value exceeds the USINT range.

- **Value Range Handling**: Since USINT can only represent values from 0 to 255, higher-order bits are discarded for input values outside this range. This can lead to data loss.
- **Deterministic Behavior**: The conversion is performed in a single execution step without state management.

The function block is stateless and re-executes the conversion with each `REQ` event.

- **Data Reduction**: Use in systems where storage space is limited and data size reduction is required.
- **F_UINT_TO_SINT**: Converts UINT to a signed 8-bit integer (SINT), handling the value range differently.
- **F_UINT_TO_INT**: Retains the 16-bit size but converts to a signed type.

F_UINT_TO_USINT` is a simple and efficient function block for converting 16-bit to unsigned 8-bit data. Its use is ideal in scenarios where a space-saving representation is required; however, the possibility of data loss with large input values should be considered.

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

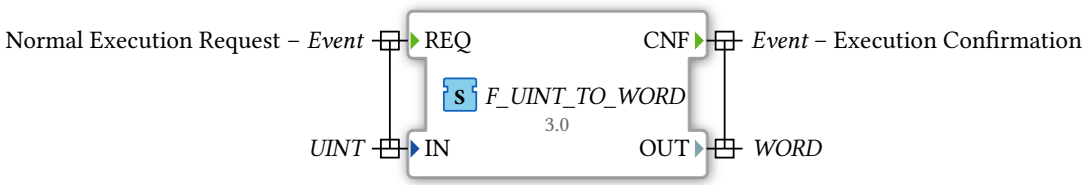
Conclusion

F_UINT_TO_WORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_UINT_TO_WORD` converts an unsigned 16-bit integer value (UINT) into a 16-bit word (WORD). This conversion is particularly necessary in scenarios where data needs to be exchanged between different representation formats or systems.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs
- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs
- **IN**: The UINT value to be converted.

Data Outputs
- **OUT**: The converted WORD value.

Adapters
- No adapters available.

Functionality
The function block performs the conversion by converting the UINT value of `IN` into a WORD using the function `UINT_TO_WORD` and outputting the result to `OUT`. The conversion is triggered by the event `REQ` and confirmed by `CNF`.

Technical Features
- The conversion is performed directly without any additional processing.
- The function block is deterministic and has no internal states.

State Overview
The function block has no internal states. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios
- Data conversion in control systems.
- Communication between systems that use different data types.
- Integration of legacy systems that require WORD data types.

⚖️ Comparison with Similar Function Blocks
- Compared to other conversion blocks such as `F_INT_TO_WORD` or `F_BYTE_TO_WORD`, `F_UINT_TO_WORD` is specifically designed for converting unsigned 16-bit integers.
- The function block has a simpler structure than generic conversion blocks that support multiple data types.

Conclusion
The `F_UINT_TO_WORD` function block offers an efficient and direct solution for converting UINT to WORD values. Its simplicity and determinism make it a reliable choice for control and communication applications.

UINT Conversions
Here you will find functions for converting the UINT data type. UINT Conversions ...

🛠️ Related exercises
- Uebung_006c
- Uebung_010b4_sub_AX
- Uebung_010b5_sub_AX
- Uebung_010c2
- Uebung_010c2_AX
- AuxIXA_TO_logiBUS_QXA_BG
- NumericValue_ID_TO_NVS
- Uebung_020c2_sub
- IX_TO_logiBUS_LED_strip_QX

- Uebung_035
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- Uebung_039_sub_NumbAnsicht
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_040
- Uebung_040_AX
- Uebung_041
- Uebung_087
- Uebung_087a1
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2
- Uebung_090a2_AX
- Uebung_103

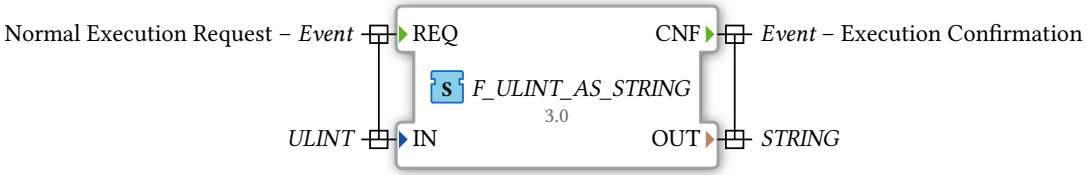
ULINT

F_ULINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_AS_STRING` converts a `ULINT` value (64-bit unsigned integer) into a `STRING` value. This functionality is particularly useful when numeric values are needed as strings for output, logging, or further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (`ULINT`): The 64-bit unsigned integer value to be converted.

Data Outputs

- **OUT** (`STRING`): The result string after the conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion when the event `REQ` is received. The `ULINT` value at input `IN` is converted to `STRING` and output at `OUT`. Successful conversion is signaled by the event `CNF`.

Technical Features

- The function block has a simple structure and contains only one algorithm that performs the conversion.
- The conversion is performed directly without additional parameters or complex logic.

State Overview

The function block has no internal states. The conversion is performed anew with each `REQ` event.

Application Scenarios

- **Logging**: Conversion of numeric values for output to log files.
- **User Interfaces**: Display of numeric values as text in user interfaces.
- **Data Communication**: Preparation of numeric values for transmission as strings.

⚖️ Comparison with similar building blocks

- **F_INT_AS_STRING**: Converts a `INT` value to a `STRING` value. In contrast, `F_ULINT_AS_STRING` supports the larger `ULINT` data type.
- **F_DINT_AS_STRING**: Converts a `DINT` value to a `STRING` value. Similar, but for 32-bit signed integers.

Conclusion

The function block `F_ULINT_AS_STRING` offers a simple and efficient way to convert `ULINT` values to `STRING` values. Its clear interface and direct functionality make it ideal for applications requiring such a conversion.

F_ULINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_AS_WSTRING` converts an unsigned 64-bit integer value (`ULINT`) into a Unicode string (`WSTRING`). This functionality is particularly useful in scenarios where numeric values are needed for display or further processing as text.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN**: The unsigned 64-bit integer value (`ULINT`) to be converted to a Unicode string.

Data Outputs

- **OUT**: The resulting Unicode string (`WSTRING`) after the conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion when the event `REQ` is triggered. The value at input `IN` is converted into a Unicode string using the function `ULINT_AS_WSTRING` and output at `OUT`. The event `CNF` signals the successful completion of the conversion.

Technical Features

- Uses the function `ULINT_AS_WSTRING` for the conversion.
- Supports the full value range of `ULINT` (0 to 18,446,744,073,709,551,615).

State Overview

- **Idle**: Waits for the `REQ` event.
- **Converting**: Performs the conversion and outputs the result via `CNF`.

Application Scenarios

- Displaying large numeric values in user interfaces.
- Logging numeric data in text format.
- Preparing data for communication with systems that expect string data.

⚖️ Comparison with Similar Function Blocks

- Compared to `F_INT_AS_WSTRING` or `F_DINT_AS_WSTRING`, this function block supports the larger value range of `ULINT`.
- Similar to `F_ULINT_TO_WSTRING`, but with direct conversion without additional formatting options.

Conclusion

The `F_ULINT_AS_WSTRING` function block offers a simple and efficient way to convert large unsigned integer values into Unicode strings. Its clear interface and reliable functionality make it a valuable component in the 4diac IDE.

🌐 Related topic subpages on ms-muc-docs.de

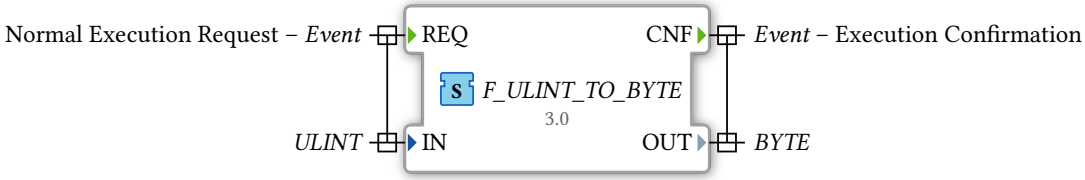
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

F_ULINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_TO_BYTE` converts a 64-bit unsigned integer value (ULINT) to an 8-bit unsigned integer value (BYTE). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. It is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. Linked to the data output `OUT`.

Data Inputs

- **IN** (ULINT): The 64-bit unsigned integer value to be converted.

Data Outputs

- **OUT** (BYTE): The resulting 8-bit unsigned integer value after conversion.

Adapters

No adapters available.

Functionality

This function block performs the ULINT to BYTE conversion when the `REQ` event is triggered. The conversion is performed using the function `ULINT_TO_BYTE(IN)`. After successful conversion, the `CNF` event is triggered, and the converted value is available at output `OUT`.

Technical Features

- The function block is a simple algorithm without state management.
- Data loss can occur during conversion if the ULINT value is outside the representable range of a BYTE (0 to 255).

State Overview

Because it is a simple function block, there are no state transitions or complex state logic.

Application Scenarios

- Type conversion in control applications where ULINT values need to be reduced to BYTEs.
- Data processing in embedded systems with limited memory capacity.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_DINT_TO_BYTE` or `F_UINT_TO_BYTE` perform similar conversions, but with different output data types.
- `F_ULINT_TO_BYTE` is specifically for converting 64-bit unsigned integers to 8-bit unsigned integers.

Conclusion

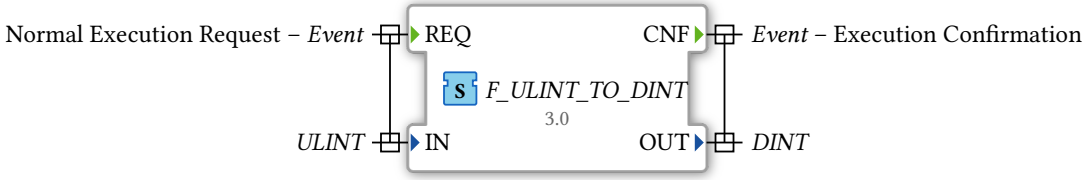
The `F_ULINT_TO_BYTE` function block provides a simple and efficient way to convert ULINT values to BYTE values. It is particularly useful in applications where data width reduction is required; however, potential data loss should be considered.

F_ULINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_TO_DINT` converts a `ULINT` value (64-bit unsigned integer) to a `DINT` value (32-bit signed integer). This conversion is particularly relevant in scenarios where data needs to be exchanged between systems with different data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN** (`ULINT`): The unsigned 64-bit value to be converted.

Data Outputs

- **OUT** (`DINT`): The result of the conversion as a signed 32-bit value.

Adapters

No adapters are present.

Functionality

When the `REQ` event is triggered, the algorithm is executed, converting the `ULINT` value at input `IN` to a `DINT` value and outputting the result at output `OUT`. Subsequently, the `CNF` event is triggered to signal the completion of the conversion.

Technical Details

- Data loss can occur if the original `ULINT` value is outside the representable range of `DINT` (i.e., greater than 2^{31}-1).
- The function block is part of the `iec61131::conversion` package.

State Overview

The function block has no internal states. The conversion occurs immediately upon receipt of the `REQ` event.

Application Scenarios

- Data conversion in control systems that use different data types.
- Integration of systems that use different bit widths for integers.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_ULINT_TO_DINT` specializes in converting `ULINT` to `DINT`.
- Other conversion blocks may support different data types or conversion directions.

Conclusion

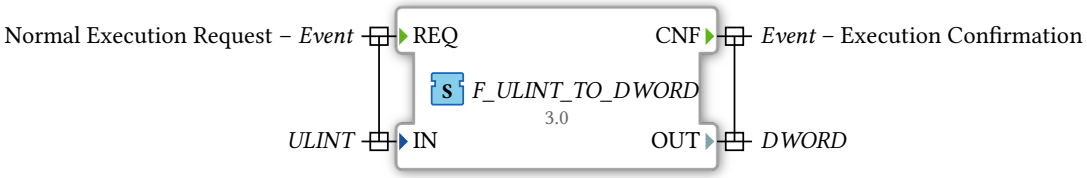
The `F_ULINT_TO_DINT` function block offers a simple and efficient way to convert unsigned 64-bit values to signed 32-bit values. Its use is beneficial in scenarios with varying data type requirements; however, potential data loss during the conversion should be considered.

F_ULINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_TO_DWORD` converts a `ULINT` value (64-bit unsigned integer) to a `DWORD` value (32-bit unsigned integer). This block is part of the `iec61131::conversion` package and enables simple and efficient type conversion.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- **IN** (`ULINT`): The 64-bit unsigned integer value to be converted to `DWORD`.

Data Outputs

- **OUT** (`DWORD`): The resulting 32-bit unsigned integer value after the conversion.

Adapters

This function block does not use any adapters.

Operation

The function block performs the conversion as soon as the `REQ` event is received. The `ULINT` value at input `IN` is converted to a `DWORD` value and output `OUT`. The conversion is confirmed by the `CNF` event.

Technical Features

- Data loss can occur if the `ULINT` value is greater than the maximum representable value of a `DWORD` (2^{32} - 1).
- The function block is implemented as a simple FB (`SimpleFB`) and uses an ST algorithm for the conversion.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Conversion**: Performs the conversion and outputs the result via `OUT`.
3. **Acknowledgement**: Sends the `CNF` event and returns to the idle state.

Application Scenarios

- Type conversion in control applications where 64-bit values need to be converted to 32-bit values.
- Data processing in systems with different word lengths.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_DINT_TO_DWORD` or `F_LINT_TO_DWORD` offer conversions between other data types, but not specifically from `ULINT` to `DWORD`.
- Unlike generic conversion blocks, this one is specialized and optimized for the `ULINT` to `DWORD` conversion.

Conclusion

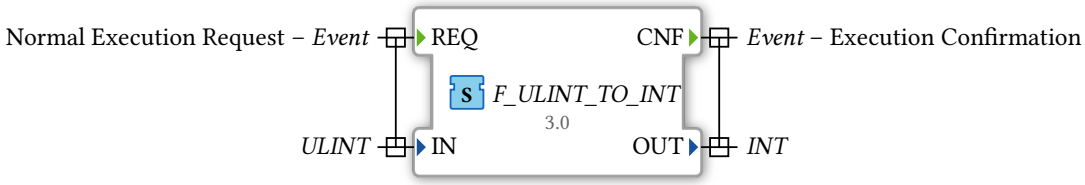
The `F_ULINT_TO_DWORD` function block provides a simple and efficient solution for converting 64-bit unsigned integers to 32-bit unsigned integers. Its clear interface structure and simple functionality make it ideal for applications that require such type conversions. However, be aware of the potential for data loss with large input values.

F_ULINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_TO_INT` converts a 64-bit unsigned integer value (ULINT) into a 16-bit signed integer value (INT). This conversion is particularly necessary when data needs to be exchanged between systems with different data types.



Interface Structure

Event Inputs

- **REQ**: This input triggers the conversion. Upon arrival of this event, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: This event signals the completion of the conversion. It is output along with the converted value at data output `OUT`.

Data Inputs

- **IN** (ULINT): This input provides the 64-bit unsigned integer value to be converted.

Data Outputs

- **OUT** (INT): This output provides the converted 16-bit signed integer value.

Adapters

This function block does not have any adapters.

Functionality

The function block performs the conversion from ULINT to INT as soon as the event `REQ` occurs. The conversion is performed using the function `ULINT_TO_INT(IN)`, which converts the value of `IN` to the target type `INT`. After successful conversion, the event `CNF` is triggered, and the converted value is output to `OUT`.

Technical Features

- **Data Type Conversion**: Converting ULINT (64-bit) to INT (16-bit) can result in data loss if the original value is outside the representable range of INT (i.e., greater than 32767 or less than −32768).
- **Simple Implementation**: The function block is implemented as a SimpleFB and uses a simple ST algorithm for the conversion.

State Overview

1. **Idle State**: The function block waits for the event `REQ`.
2. **Conversion State**: The conversion is performed upon receipt of `REQ`.
3. **Acknowledge State**: After the conversion, `CNF` is triggered, and the function block returns to the idle state.

Application Scenarios

- **Data Compatibility**: Used when systems need to communicate with different data types.
- **Data Reduction**: Can be used to save memory when the higher precision of ULINT is not required.

⚖️ Comparison with Similar Blocks

- **F_ULINT_TO_DINT**: Converts ULINT to DINT (32-bit), offering a larger value range than INT.
- **F_ULINT_TO_LINT**: Converts ULINT to LINT (64-bit), maintaining full precision.

Conclusion

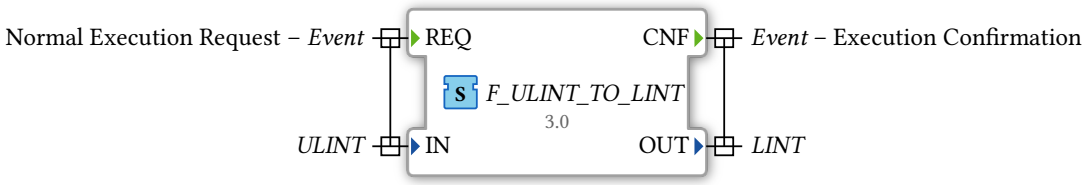
The function block `F_ULINT_TO_INT` is a simple yet effective tool for converting 64-bit unsigned integer values to 16-bit signed integer values. It is particularly useful in scenarios where memory efficiency or compatibility with older systems is required. However, be aware of potential data loss when converting large values.

F_ULINT_TO_LINT

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_ULINT_TO_LINT` converts an unsigned 64-bit integer value (ULINT) to a signed 64-bit integer value (LINT). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (ULINT): The unsigned 64-bit integer value to be converted.

Data Outputs

- **OUT** (LINT): The resulting signed 64-bit integer value after conversion.

Adapters

- No adapters are available.

Functionality

The function block performs the conversion by transforming the value of input `IN` from type `ULINT` to type `LINT`. The conversion is performed directly and without further processing. The result is output at `OUT` as soon as the event `CNF` is triggered.

Technical Features

- The conversion is performed using the function `ULINT_TO_LINT`, which is implemented in the function block's algorithm.
- The block is suitable for use in real-time systems because it does not involve any delays or complex calculations.

State Overview

The function block has no internal states. The conversion is performed and immediately acknowledged each time the `REQ` event occurs.

Application Scenarios

- Type conversion in control and automation systems.
- Data processing in industrial applications where unsigned and signed values need to be processed.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks, such as `F_DINT_TO_INT`, this one is specifically designed for 64-bit data types.
- Similar blocks might include additional features like range checks or error handling, which is not the case here.

Conclusion

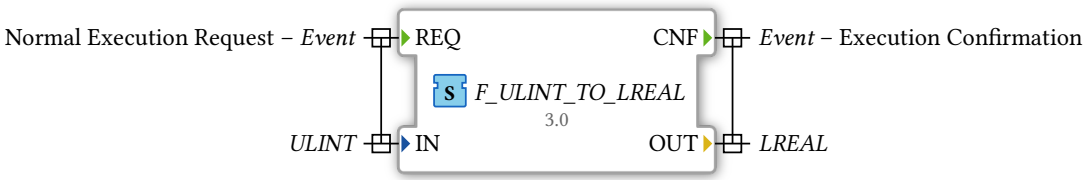
The `F_ULINT_TO_LINT` function block is a simple and efficient tool for converting between the data types `ULINT` and `LINT`. Its simplicity makes it reliable and easy to integrate into existing systems.

F_ULINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_TO_LREAL` converts a `ULINT` value (64-bit unsigned integer) to a `LREAL` value (64-bit floating point). This conversion is particularly necessary in scenarios where integer values need to be converted to floating-point numbers for further calculations.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- `IN` (`ULINT`): The input value to be converted.

Data Outputs

- `OUT` (`LREAL`): The converted output value.

Adapters

No adapters available.

Functionality

The function block performs the conversion from `ULINT` to `LREAL` when the `REQ` event is triggered. The conversion is carried out using the built-in function `ULINT_TO_LREAL`. After the conversion is complete, the `CNF` event is triggered, and the converted value is made available at output `OUT`.

Technical Features

- The function block has a simple structure and contains only one algorithm (`REQ`) that performs the conversion.
- The conversion is lossless because `LREAL` covers a larger range of values than `ULINT`.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of integer values to floating-point numbers for further calculations.
- Use in control systems where floating-point operations are required.

⚖️ Comparison with Similar Blocks

- Compared to other conversion blocks (e.g., `F_INT_TO_REAL`), this one is specifically optimized for converting 64-bit unsigned integers to 64-bit floating points.
- Similar blocks may have lower precision or a smaller range of values.

Conclusion

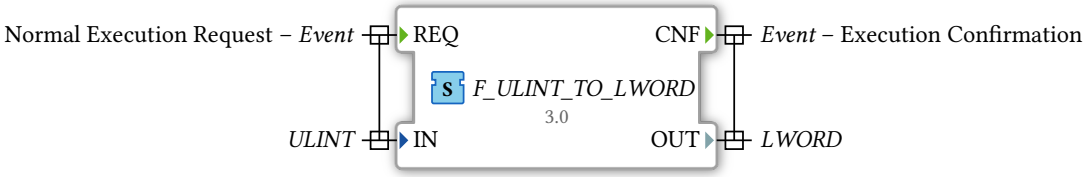
The `F_ULINT_TO_LREAL` function block offers a simple and efficient way to convert `ULINT` values to `LREAL` values. Its clear interface and lossless conversion make it a useful component in various applications.

F_ULINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_TO_LWORD` converts a ULINT data type (64-bit unsigned integer) to an LWORD data type (64-bit bit string). This block is particularly useful in applications where direct type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. When activated, the input value `IN` is read and processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output value `OUT` is available at this time.

Data Inputs

- **IN**: ULINT (64-bit unsigned integer) - The input value to be converted to an LWORD.

Data Outputs

- **OUT**: LWORD (64-bit bit string) - The result of the conversion.

Adapters

This function block does not use any adapters.

Functionality

The function block performs the conversion by transforming the ULINT input value `IN` into an LWORD output value `OUT`. The conversion is performed directly without any further processing. When the event input `REQ` is activated, the algorithm is executed and the result is signaled via the event output `CNF`.

Technical Features

- The conversion is direct and without data loss, as both data types are 64 bits wide.
- The function block is deterministic and has no internal states.

State Overview

The function block has no internal states. Execution occurs immediately upon activation of the `REQ` event.

Application Scenarios

- Type conversion in control applications where ULINT values must be processed as bit strings.
- Communication between systems that use different data types.
- Debugging and logging where bit representations of numeric values are required.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks like `F_DINT_TO_LWORD` or `F_UDINT_TO_LWORD`, this block works with the larger ULINT data type.
- Similar blocks might offer additional features like scaling or formatting, but this block specializes in pure type conversion.

Conclusion

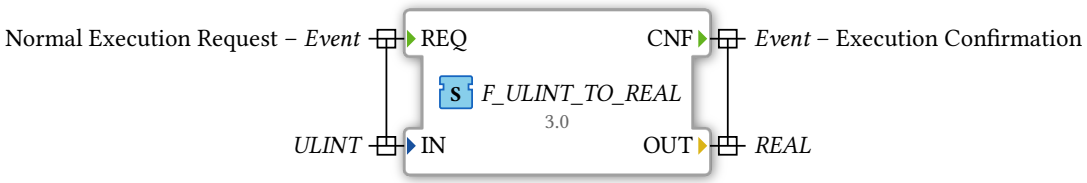
The `F_ULINT_TO_LWORD` function block is a simple and efficient block for directly converting ULINT to LWORD. Its deterministic and stateless nature makes it reliable and easy to integrate into existing applications.

F_ULINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_TO_REAL` converts a `ULINT` value (64-bit unsigned integer) to a `REAL` value (32-bit floating-point number). This block is particularly useful in scenarios where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `ULINT`, which is to be converted to a `REAL` value.

Data Outputs

- **OUT**: The output value of type `REAL`, representing the result of the conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is triggered. The algorithm within the function block uses the function `ULINT_TO_REAL` to convert the value `ULINT` into a value `REAL`. The result is output via the data output `OUT`, and the event `CNF` signals successful execution.

Technical Features

- The function block is part of the package `iec61131::conversion`.
- Supports the conversion of 64-bit unsigned integers to 32-bit floating-point numbers.
- Simple and direct implementation without additional parameters or complex logic.

State Overview

The function block has a simple state machine:

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Complete**: Sends the `CNF` event and outputs the result.

Application Scenarios

- Data processing in control systems where type conversion is required.
- Integration into larger control algorithms that perform floating-point calculations with values originally stored as integers.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_DINT_TO_REAL` or `F_LINT_TO_REAL` offer conversions from other integer types to `REAL`, but with different input sizes and precisions.
- `F_ULINT_TO_REAL` is specifically optimized for 64-bit unsigned integers.

Conclusion

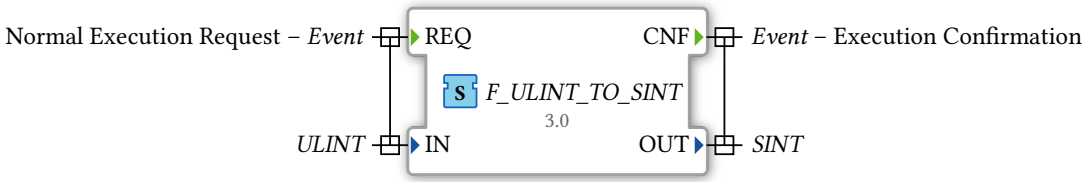
The `F_ULINT_TO_REAL` function block is a simple yet essential building block for type conversion in IEC 61499-based control systems. Its clear interface and efficient operation make it a reliable solution for conversion tasks.

F_ULINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_ULINT_TO_SINT` converts a `ULINT` value (64-bit unsigned integer) to a `SINT` value (8-bit signed integer). This block is particularly useful in applications where type conversion between these two data types is required.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs
- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.

Data Inputs
- **IN**: The input expects a `ULINT` value to be converted.

Data Outputs
- **OUT**: The output provides the converted `SINT` value.

Adapters
No adapters are present.

Functionality
The function block performs the conversion as soon as the `REQ` event is triggered. The algorithm converts the `ULINT` value to `IN` and stores the result in `OUT`. The event `CNF` is then triggered to signal the completion of the operation.

Technical Details
- The conversion is performed directly by the function `ULINT_TO_SINT`.
- Note that data loss can occur when converting from a larger to a smaller data type if the `ULINT` value is outside the representable range of `SINT`.

State Overview
The function block has no internal state. The conversion is performed on every `REQ` event.

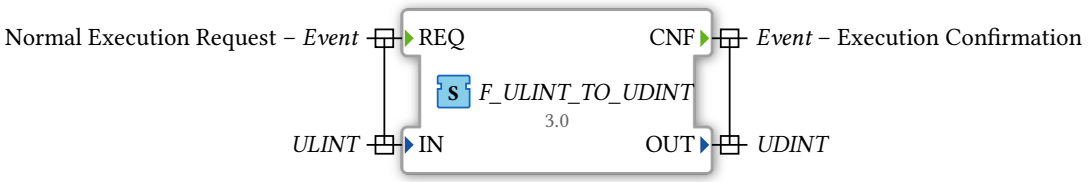
Application Scenarios
- Type conversion in control applications
- Data processing where different data types need to be handled
- Communication between systems with different data types

⚖️ Comparison with Similar Function Blocks
- Similar function blocks like `F_DINT_TO_SINT` or `F_LINT_TO_SINT` perform similar conversions, but with different output data types.
- `F_ULINT_TO_SINT` is specifically for converting `ULINT` to `SINT`.

Conclusion
The `F_ULINT_TO_SINT` function block provides a simple and efficient way to convert `ULINT` values to `SINT` values. It is particularly useful in applications where type conversion is required; however, potential data loss during the conversion should be considered.

F_ULINT_TO_UDINT
• – ◦ • – ◦ • – ◦ •

Introduction
The function block `F_ULINT_TO_UDINT` converts a 64-bit unsigned integer value (ULINT) to a 32-bit unsigned integer value (UDINT). This conversion is useful when exchanging data between systems or components that use different data types.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. It is linked to the data input `IN`.

Event Outputs
- **CNF**: Signals the completion of the conversion. Linked to the data output `OUT`.

Data Inputs
- **IN**: ULINT (64-bit unsigned integer) - The input value to be converted.

Data Outputs
- **OUT**: UDINT (32-bit unsigned integer) - The converted output value.

Adapters
No adapters are available.

Functionality
This function block performs a direct conversion of a ULINT value to a UDINT value. The conversion is carried out by calling the function `ULINT_TO_UDINT` within the algorithm `REQ`. During the conversion, the 64-bit value is truncated to 32 bits. Note that data loss can occur if the original value is greater than the maximum value of a UDINT.

Technical Features
- The function block is simple and has no internal states.
- The conversion is performed by a single algorithm (`REQ`) that is executed with each call to the `REQ` event.
- There is no overflow error handling. The user must ensure that the input value is within the valid range for UDINT.

State Overview
Since the function block is stateless, there are no state transitions or internal states.

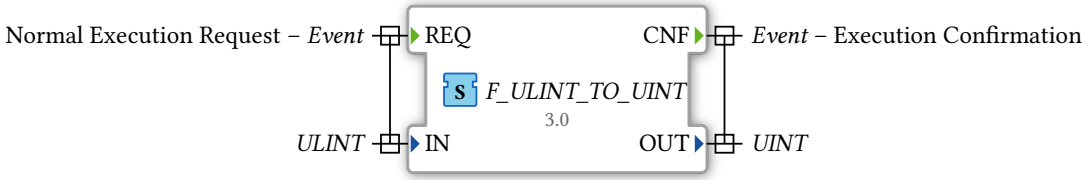
Application Scenarios
- Data conversion in control systems where different data types are used.
- Reduced memory requirements when 64-bit values are not needed.
- Compatibility with older systems that only support 32-bit values.

⚖️ Comparison with Similar Function Blocks
- **F_ULINT_TO_INT**: Converts ULINT to INT (signed integer), taking the sign into account.
- **F_ULINT_TO_LINT**: Converts ULINT to LINT (64-bit signed integer), which does not reduce the data size but does change the sign.
- **F_UDINT_TO_ULINT**: Reverse conversion from UDINT to ULINT, which does not result in any data loss.

Conclusion
The `F_ULINT_TO_UDINT` function block is a simple and efficient tool for converting 64-bit unsigned integer values to 32-bit unsigned integer values. It is particularly useful in scenarios where memory efficiency or compatibility with older systems is required. However, users should be careful to avoid overflows, as the function block does not offer automatic error handling.

F_ULINT_TO_UINT
• − ◦ • − ◦ • − ◦ •

The function block `F_ULINT_TO_UINT` converts a 64-bit unsigned integer value (ULINT) to a 16-bit unsigned integer value (UINT). This block is part of the `iec61131::conversion` package and implements a simple type conversion.



- **REQ**: Starts the conversion process. This input is connected to the data input `IN`.
- **CNF**: Signals the completion of the conversion. The output is connected to the data output `OUT`.
- **IN**: ULINT (64-bit unsigned integer) - The input value to be converted.
- **OUT**: UINT (16-bit unsigned integer) - The result of the conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters are present.

Functionality

The function block performs the conversion from ULINT to UINT when the `REQ` event occurs. The conversion is carried out using the function `ULINT_TO_UINT`. After the conversion is complete, the `CNF` event is triggered.

- The function block performs a direct type conversion, truncating the higher-order bits of the ULINT value if the value is outside the UINT range.
- The function block is deterministic and has no internal states.

The function block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

- Type conversion in control applications where ULINT values need to be reduced to UINT.
- Data processing in embedded systems with limited memory capacity.
- Compared to generic conversion function blocks, `F_ULINT_TO_UINT` is specialized and optimized for the specific conversion of ULINT to UINT.

Other function blocks might offer additional functionalities such as range checks or scaling, which is not the case here.

The `F_ULINT_TO_UINT` function block is a simple and efficient building block for converting 64-bit unsigned integer values to 16-bit unsigned integer values. It is particularly useful in applications where direct and fast type conversion is required.

Technical Features

State Overview

Application Scenarios

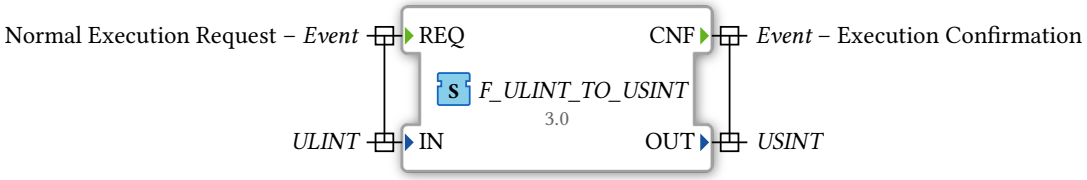
⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

F_ULINT_TO_USINT

• – ◦ • – ◦ • – ◦ •

The function block F_ULINT_TO_USINT converts a 64-bit unsigned integer value (ULINT) to an 8-bit unsigned integer value (USINT). This block is particularly useful in scenarios where type conversion between these two data types is required.



- **REQ**: Starts the execution of the function block. Connected to the data input IN.
- **CNF**: Signals successful completion of the conversion. Connected to the data output OUT.
- **IN**: The input value of type ULINT (64-bit unsigned integer) to be converted.
- **OUT**: The output value of type USINT (8-bit unsigned integer) containing the result of the conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use any adapters.

When the event REQ is triggered, the algorithm is executed, converting the value of IN of type ULINT to type USINT and outputting the result to OUT. The event CNF is then triggered to signal the successful completion of the operation.

- The conversion truncates the value of ULINT to the size of USINT. It is important to ensure that the input value is within the range of USINT (0 to 255) to avoid unexpected results.

The function block has a simple state machine:

1. **Idle**: Waits for the REQ event.
2. **Processing**: Performs the conversion and triggers CNF.

- Type conversion in control systems where data is in different formats.
- Reducing data storage size when only small value ranges are needed.
- Similar function blocks such as F_DINT_TO_INT or F_LREAL_TO_REAL perform similar type conversions, but for different data types.

Unlike generic conversion blocks, F_ULINT_TO_USINT is specialized and optimized for converting between ULINT and USINT.

The F_ULINT_TO_USINT function block provides an efficient and simple way to convert values from ULINT to USINT. Its clear interface and simple operation make it a useful building block in IEC 61131 programming.

Functionality

Application Scenarios

State Overview

Technical Features

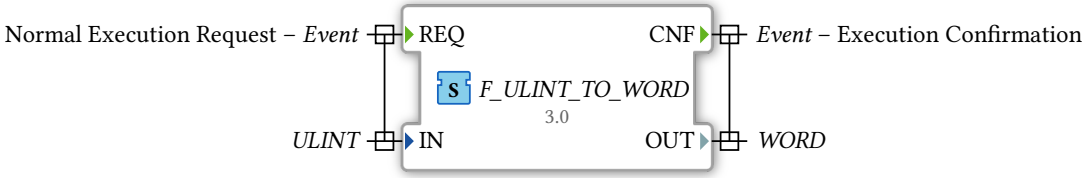
⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

F_ULINT_TO_WORD

• – ◦ • – ◦ • – ◦ •

The function block `F_ULINT_TO_WORD` converts a 64-bit unsigned integer value (ULINT) to a 16-bit unsigned integer value (WORD). This block is part of the `iec61131::conversion` package and implements a simple type conversion.



- **REQ**: Starts the conversion process. This input is connected to the data input `IN`.
- **CNF**: Signals successful completion of the conversion. The output is connected to the data output `OUT`.
- **IN**: ULINT (64-bit unsigned integer) - The input value to be converted.
- **OUT**: WORD (16-bit unsigned integer) - The result of the conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters are available.

Functionality

This function block performs the conversion from ULINT to WORD when the `REQ` event occurs. The conversion is carried out using the built-in function `ULINT_TO_WORD`. After successful conversion, the `CNF` event is triggered.

- The block does not support overflow error handling (since ULINT has a larger value range than WORD).
- The conversion is instantaneous and without any additional delay.

The block has no internal states. The conversion occurs immediately upon receiving the `REQ` event.

- Type conversion in control applications where ULINT values need to be converted to WORD values.
- Data reduction when only the lower 16 bits of a ULINT value are relevant.
- Similar blocks such as `F_DINT_TO_WORD` or `F_LINT_TO_WORD` offer conversions from other integer types to WORD, but with different value ranges and precision.

The `F_ULINT_TO_WORD` function block is a simple and efficient tool for converting ULINT to WORD values. It is particularly useful in scenarios where data width reduction is required. However, be aware of the potential data loss when converting from larger to smaller data types.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

ULINT Conversions

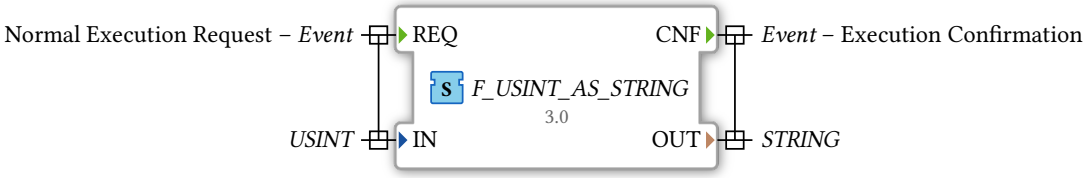
Here you will find functions for converting the ULINT data type.

USINT

F_USINT_AS_STRING

• – ◦ • – ◦ • – ◦ •

The function block `F_USINT_AS_STRING` converts an unsigned 8-bit integer value (USINT) into a string (STRING). This functionality is particularly useful when numeric values need to be converted into a readable text format for display or logging.



- **REQ**: Starts the conversion. This input is linked to the data input `IN`.
- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.
- **IN**: An unsigned 8-bit integer value (USINT) to be converted to a string.
- **OUT**: The resulting string (STRING) containing the converted value.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- No adapters are present.

Functionality

When the event input `REQ` is triggered, the algorithm is executed, converting the USINT value `IN` to a string and outputting the result to `OUT`. The event `CNF` is then triggered to signal the successful completion of the operation.

- The function block uses the function `USINT_AS_STRING` for conversion.
- It is a simple function block without a state machine or complex logic.

Since it is a simple function block, there are no state transitions or management. The operation is executed on each `REQ` event and acknowledged with `CNF`.

- Displaying numerical values on a control panel.
- Logging measured values to a text file or database.
- Preparing data for communication via text protocols (e.g., HTTP, MQTT).

- Compared to generic conversion blocks, `F_USINT_AS_STRING` specializes in converting USINT to STRING and is therefore more efficient in this specific application.
- Other blocks might offer additional options such as formatting or locale settings, which is not the case here.

The `F_USINT_AS_STRING` function block is a simple and efficient tool for converting unsigned 8-bit integers to strings. Its simplicity makes it particularly suitable for applications where performance and reliability are paramount.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

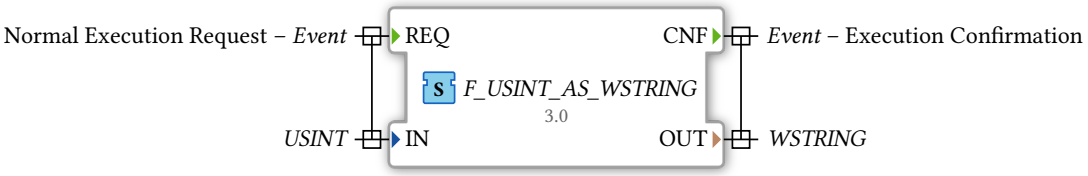
Conclusion

F_USINT_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_USINT_AS_WSTRING` converts an unsigned 8-bit integer value (`USINT`) into a Unicode string (`WSTRING`). This functionality is particularly useful in scenarios where numeric values need to be output or processed as text.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at data output `OUT`.

Data Inputs

- **IN**: An unsigned 8-bit integer value (`USINT`) to be converted to a Unicode string.

Data Outputs

- **OUT**: The resulting Unicode string (`WSTRING`) after the conversion.

Adapters

- No adapters are available.

Operation

The function block performs the conversion as soon as the event `REQ` is triggered. The input value `IN` is taken and converted into a Unicode string using the function `USINT_AS_WSTRING`. The result is output `OUT`, and the event `CNF` signals successful execution.

Technical Features

- Uses the built-in function `USINT_AS_WSTRING` for conversion.
- Supports the full value range of `USINT` (0 to 255).

State Overview

1. **Idle**: Waits for the event `REQ`.
2. **Conversion**: Processes the input value and generates the output.
3. **Acknowledgement**: Sends the event `CNF` and returns to the idle state.

Application Scenarios

- Displaying numeric values in user interfaces.
- Logging numeric data to text files.
- Communicating with systems that expect text data.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_USINT_AS_WSTRING` specializes in converting `USINT` to `WSTRING`.
- Other blocks might support different data types or additional formatting options.

Conclusion

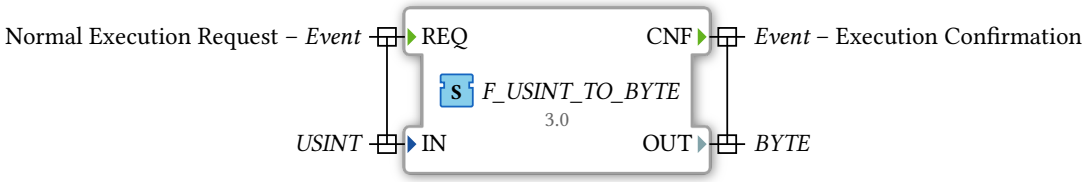
The `F_USINT_AS_WSTRING` function block provides a simple and efficient way to convert unsigned 8-bit integer values to Unicode strings. Its clear interface and direct functionality make it ideal for projects that require such conversions.

F_USINT_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_USINT_TO_BYTE` converts an unsigned 8-bit integer value (USINT) into a byte value (BYTE). This conversion is particularly necessary in scenarios where data needs to be exchanged between different formats or systems.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `USINT` (unsigned 8-bit integer) to be converted to a `BYTE` value.

Data Outputs

- **OUT**: The output value of type `BYTE`, representing the result of the conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion as soon as the event `REQ` is received. The input value `IN`, of type `USINT`, is converted to a value of `BYTE` and output `OUT`. The event `CNF` signals the successful completion of the conversion.

Technical Features

- The conversion is performed directly without any additional processing steps.

- The block is suitable for use in real-time systems because it does not perform any time-critical operations.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Conversion**: Performs the conversion and outputs the result via `OUT`.
3. **Confirmation**: Sends the `CNF` event to confirm successful conversion.

Application Scenarios

- Data conversion in control systems.
- Communication between systems with different data types.
- Preprocessing of sensor data.

⚖️ Comparison with similar function blocks

- **F_BYTE_TO_USINT**: Performs the reverse conversion (BYTE to USINT).
- **F_INT_TO_STRING**: Converts an integer value to a string, but is more complex to handle.

🛠️ Related Exercises

- Exercise_126b_sub

Conclusion

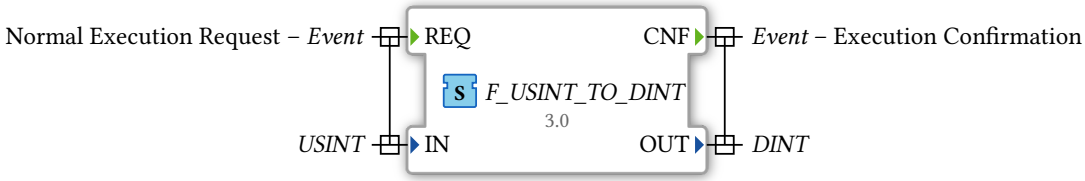
The `F_USINT_TO_BYTE` function block is a simple and efficient tool for converting `USINT` values to `BYTE` values. Its clear interface structure and direct functionality make it ideal for use in various industrial applications.

F_USINT_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_USINT_TO_DINT` converts a `USINT` value (8-bit unsigned integer) to a `DINT` value (32-bit signed integer). This conversion is useful when values need to be converted between different data types, especially when an extension of the bit width is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event is triggered, the value of `IN` is converted to `OUT`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event is output along with the updated value of `OUT`.

Data Inputs

- **IN** (`USINT`): The input value to be converted.

Data Outputs

- **OUT** (`DINT`): The converted output value.

Adapters

No adapters available.

Functionality

The function block directly assigns the input value `IN` (of type `USINT`) to the output `OUT` (of type `DINT`). Since `DINT` has a larger bit width than `USINT`, the value is extended without data loss.

The algorithm is triggered by the `REQ` event and outputs the `CNF` event after the conversion.

Technical Features

- The function block is deterministic and has no internal state.
- The conversion is lossless, as `DINT` can represent all values of `USINT`.

State Overview

The function block has no internal states. Execution occurs immediately after the `REQ` event is triggered.

Application Scenarios

- Extension of 8-bit unsigned values to 32-bit signed values in control applications.
- Compatibility between different parts of a control system that use different data types.

⚖️ Comparison with Similar Function Blocks

- Compared to `F_SINT_TO_DINT` or `F_INT_TO_DINT`, this function block handles only unsigned 8-bit values.

Similar conversion blocks exist for other data types, such as `F_BOOL_TO_INT` or `F_REAL_TO_LREAL`.

Conclusion

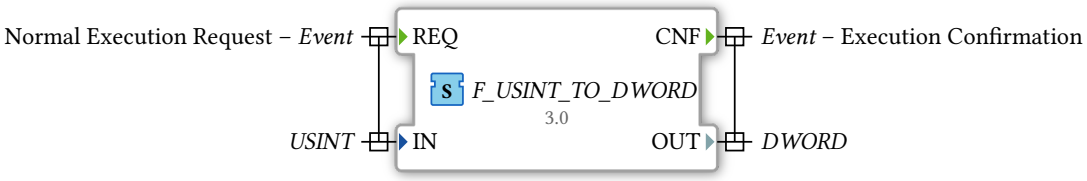
The `F_USINT_TO_DINT` function block is a simple and efficient building block for converting 8-bit unsigned integer values to 32-bit signed integer values. It is particularly useful in scenarios where type extension without data loss is required.

F_USINT_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_USINT_TO_DWORD` converts a `USINT` value (8-bit unsigned integer) to a `DWORD` value (32-bit unsigned integer). This conversion is particularly useful when data needs to be exchanged or processed between systems with different word lengths.



Interface Structure

Event Inputs

- `REQ` (Normal Execution Request): Starts the conversion. This input is connected to the data input `IN`.

Event Outputs

- `CNF` (Execution Confirmation): Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs

- `IN` (USINT): The input value to be converted. The data type is `USINT` (8-bit unsigned integer).

Data Outputs

- `OUT` (DWORD): The converted output value. The data type is `DWORD` (32-bit unsigned integer).

Adapters

No adapters available.

Functionality

The function block performs the conversion from `USINT` to `DWORD` as soon as the event `REQ` is triggered. The algorithm `REQ` uses the function `USINT_TO_DWORD` to convert the input value `IN` to the output value `OUT`. After successful conversion, the event `CNF` is triggered.

Technical Features

- The function block has a simple structure and contains only one algorithm.
- The conversion is performed without data loss because the `DWORD` data type covers a larger value range than `USINT`.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Conversion State**: Performs the conversion.
3. **Acknowledgement State**: Triggers the `CNF` event and returns to the idle state.

Application Scenarios

- Data conversion in control systems that use different word lengths.
- Integration of peripheral devices that deliver data in different formats.
- Data processing in industrial automation solutions.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_BYTE_TO_DWORD` or `F_WORD_TO_DWORD` offer conversions between other data types.
- `F_USINT_TO_DWORD` specializes in converting 8-bit unsigned integers to 32-bit unsigned integers.

Conclusion

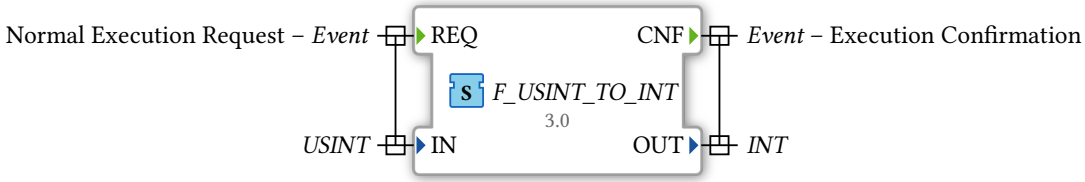
The `F_USINT_TO_DWORD` function block is a simple and efficient tool for converting `USINT` values to `DWORD` values. Its clear interface structure and simple functionality make it a reliable solution for data conversion tasks in industrial automation systems.

F_USINT_TO_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_USINT_TO_INT` converts an unsigned 8-bit integer value (USINT) to a signed 16-bit integer value (INT). This block is part of the `iec61131::conversion` package and enables simple type conversion between these two data types.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN** (USINT): The unsigned 8-bit integer value to be converted.

Data Outputs

- **OUT** (INT): The resulting signed 16-bit integer value after the conversion.

Adapters

This function block does not use any adapters.

Operation

The function block performs the conversion by directly assigning the value of input `IN` to output `OUT`. The conversion occurs when event `REQ` is triggered and is confirmed by event `CNF`.

Technical Features

- **Simple Type Conversion**: The function block performs no additional processing or validation of the data.
- **Efficient Implementation**: The conversion is performed directly in an ST algorithm without additional logic.

State Overview

The function block has no internal states. The conversion occurs immediately upon the occurrence of the `REQ` event.

Application Scenarios

- **Type Conversion in Control Applications**: When an unsigned 8-bit value needs to be converted to a signed 16-bit value.
- **Data Compatibility**: In scenarios where different components expect different data types.

⚖️ Comparison with Similar Function Blocks

- **F_USINT_TO_UINT**: Converts USINT to UINT (unsigned 16-bit integer).
- **F_USINT_TO_DINT**: Converts USINT to DINT (signed 32-bit integer).
- **F_USINT_TO_UDINT**: Converts USINT to UDINT (unsigned 32-bit integer).

Conclusion

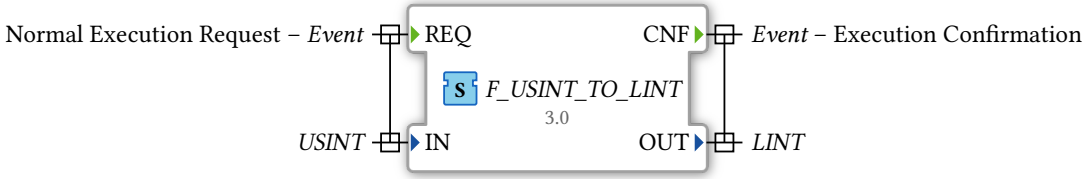
The `F_USINT_TO_INT` function block provides a simple and efficient way to convert unsigned 8-bit integer values to signed 16-bit integer values. It is particularly useful in applications where type conversion without additional processing is required.

F_USINT_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_USINT_TO_LINT` converts a `USINT` value (8-bit unsigned integer) to a `LINT` value (64-bit signed integer). This conversion is particularly useful when values from smaller data types need to be converted to larger ones without losing the original information.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `USINT` to be converted.

Data Outputs

- **OUT**: The converted output value of type `LINT`.

Adapters

No adapters are available.

Functionality

This function block directly assigns the input value `IN` to the output value `OUT`. Since `USINT` is an 8-bit unsigned integer and `LINT` is a 64-bit signed integer, the value is converted to the larger data type without any change to the numeric value.

Technical Features

- **Algorithm**: The algorithm consists of a simple assignment (`OUT := IN;`).
- **Data Type Compatibility**: Since `USINT` is an unsigned data type and `LINT` is a signed data type, the value is implicitly extended without any data loss.

State Overview

The function block has no internal state. The conversion occurs immediately upon receiving the `REQ` event.

Application Scenarios

- **Data Processing**: When values need to be converted from smaller to larger data types.
- **Communication**: When transferring data between systems with different data type requirements.

⚖️ Comparison with Similar Function Blocks

- **F_USINT_TO_INT**: Converts `USINT` to `INT` (16-bit signed integer). In contrast, `F_USINT_TO_LINT` offers greater range due to its 64-bit output.
- **F_USINT_TO_ULINT**: Converts `USINT` to `ULINT` (64-bit unsigned integer). Here, the output is also treated as unsigned.

Conclusion

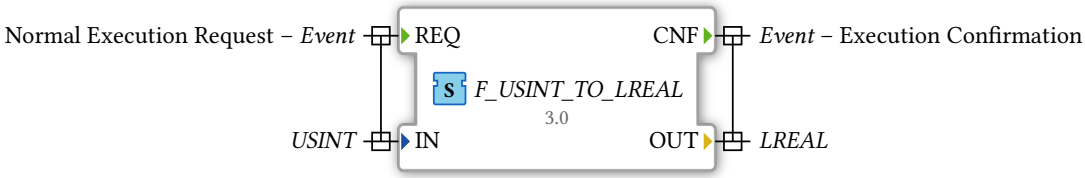
The `F_USINT_TO_LINT` function block is a simple yet effective building block for converting 8-bit unsigned integer values to 64-bit signed integer values. Its direct and lossless conversion makes it ideal for applications where preserving the numeric value in a larger data type is required.

F_USINT_TO_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_USINT_TO_LREAL` converts an unsigned 8-bit integer value (USINT) to a 64-bit floating-point value (LREAL). This conversion is particularly useful when data needs to be exchanged or processed between systems with different data types.



Interface Structure

Event Inputs

- `REQ`: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- `IN` (USINT): The input value to be converted.

Data Outputs

- `OUT` (LREAL): The converted output value.

Adapters

- No adapters are available.

Functionality

The function block performs a direct type conversion. Upon the arrival of a `REQ` event, the value of `IN` is converted to the LREAL data type and output to `OUT`. The `CNF` event is then triggered to signal the completion of the conversion.

Technical Features

- The conversion is performed without any additional scaling or adaptation.
- The block is optimized for simple type conversions and has no complex logic or state management.

State Overview

The function block has no internal states. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Conversion of sensor data in USINT format for calculations requiring LREAL.
- Interfaces between systems with different data type requirements.
- Simple type conversions in automation applications.

⚖️ Comparison with similar blocks

- Compared to generic conversion blocks, `F_USINT_TO_LREAL` is specialized and therefore more efficient for this specific conversion.
- Other blocks might offer additional features such as scaling or range checks, which are not required here.

Conclusion

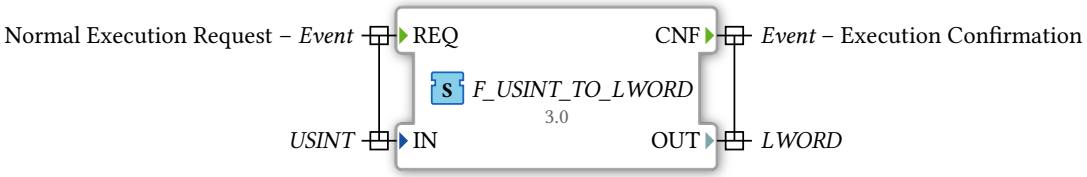
The `F_USINT_TO_LREAL` function block is a simple and efficient building block for the direct conversion of USINT to LREAL. Its strength lies in its simplicity and specialization, making it ideal for specific use cases where no additional logic is needed.

F_USINT_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_USINT_TO_LWORD` converts a `USINT` value (8-bit unsigned integer) to a `LWORD` value (64-bit unsigned integer). This conversion is useful when processing data with different bit lengths, especially in systems that require a uniform data width.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input `IN` is read and converted.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at the data output `OUT`.

Data Inputs

- **IN**: Expects a `USINT` value (8-bit unsigned integer) to be converted.

Data Outputs

- **OUT**: Returns the converted `LWORD` value (64-bit unsigned integer).

Adapters

This function block has no adapter interfaces.

Operation

The function block performs the conversion as soon as the `REQ` event is triggered. The algorithm reads the value of `IN`, converts it from `USINT` to `LWORD`, and provides the result to `OUT`. The `CNF` event is then triggered to signal the successful completion of the operation.

Technical Features

- The conversion is lossless because the `USINT` value is converted to the larger `LWORD` type.
- The function block is deterministic and has no internal state.

State Overview

The function block has no internal state. Execution is purely event-driven, triggered by the `REQ` event.

Application Scenarios

- Data conversion in control systems where different data widths need to be processed.
- Integration into larger function block networks that require a uniform data width.

⚖️ Comparison with Similar Blocks

- Compared to other conversion blocks (e.g., `F_INT_TO_LWORD`), this one is specifically optimized for converting 8-bit unsigned integers.
- Similar blocks might offer additional features such as scaling or error handling, but this block is limited to pure conversion.

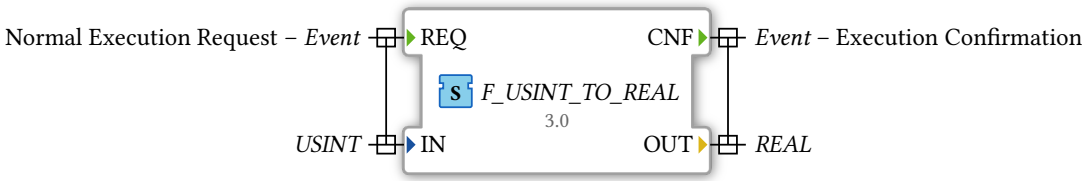
Conclusion

The `F_USINT_TO_LWORD` function block is a simple and efficient block for converting 8-bit unsigned integers to 64-bit unsigned integers. Its simplicity and determinism make it a reliable component in control systems.

F_USINT_TO_REAL

• – ◦ • – ◦ • – ◦ •

The function block `F_USINT_TO_REAL` converts an unsigned 8-bit integer value (USINT) into a 32-bit floating-point value (REAL). This conversion is particularly necessary in scenarios where numeric values of different data types need to be processed or used further.



- **REQ**: Starts the conversion. This input is linked to the data input `IN`.
- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.
- **IN**: An unsigned 8-bit integer value (USINT) to be converted to a REAL value.
- **OUT**: The resulting 32-bit floating-point value (REAL) after the conversion.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

This function block does not use any adapters.

The function block performs the conversion by directly converting the USINT value from `IN` to the REAL data type and outputting it to `OUT`. The conversion occurs synchronously upon receiving the `REQ` event and is confirmed by the `CNF` event.

- The conversion is performed without additional scaling or rounding.
- The block is optimized for simple conversions and does not provide error handling for invalid values.

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Completed**: Sends the `CNF` event and outputs the converted value.

- Conversion of sensor values (e.g., raw ADC data) into floating-point values for further calculations.
- Compared to generic conversion blocks, `F_USINT_TO_REAL` is specialized and therefore more efficient for this specific conversion.
- Other blocks, such as `F_INT_TO_REAL` or `F_DINT_TO_REAL`, are designed for other integer data types.

The `F_USINT_TO_REAL` function block is a simple and efficient function block for converting unsigned 8-bit integers to floating-point values. Its simplicity and direct operation make it ideal for applications requiring fast and reliable type conversion.

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

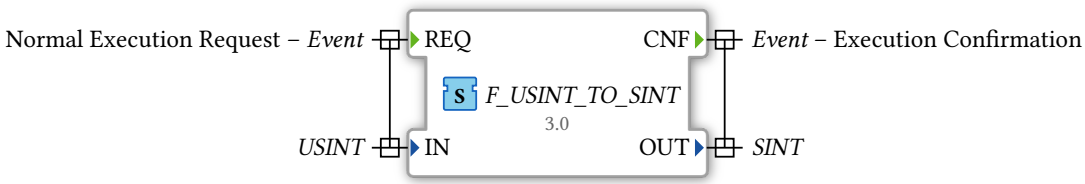
Conclusion

F_USINT_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_USINT_TO_SINT` converts an unsigned 8-bit integer value (USINT) to a signed 8-bit integer value (SINT). This block is particularly useful when exchanging data between systems that use different data types.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. When activated, the input value `IN` is processed.

Event Outputs
- **CNF**: Signals the completion of the conversion. The output value `OUT` is then available.

Data Inputs
- **IN** (USINT): The unsigned 8-bit integer value to be converted.

Data Outputs
- **OUT** (SINT): The resulting signed 8-bit integer value after the conversion.

Adapters
No adapters are available.

Functionality
The function block performs the conversion by converting the USINT value `IN` to a SINT value `OUT`. The conversion occurs directly upon activation of the `REQ` event, and the result is output via the `CNF` event.

Technical Features
- The conversion is performed without additional parameters or configuration.
- The algorithm is implemented in Structured Text (ST) and uses the function `USINT_TO_SINT`.

State Overview
1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Completed**: Sends the `CNF` event with the result.

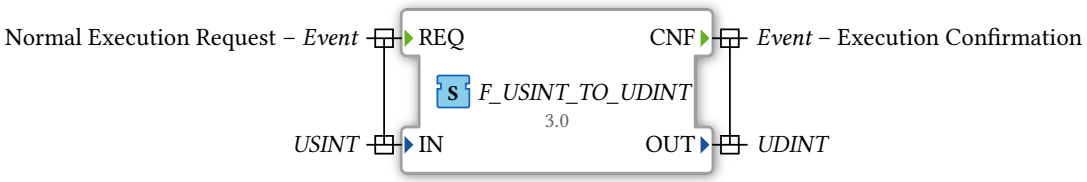
Application Scenarios
- Data conversion in control systems.
- Communication between devices with different data types.
- Preprocessing of sensor data.

⚖️ Comparison with Similar Function Blocks
- Similar function blocks such as `F_INT_TO_REAL` or `F_BOOL_TO_INT` offer conversions between other data types.
- `F_USINT_TO_SINT` specializes in conversion between 8-bit integer types.

Conclusion
The `F_USINT_TO_SINT` function block is a simple and efficient building block for converting between unsigned and signed 8-bit integer values. Its clear interface and direct functionality make it ideal for use in various control and communication applications.

F_USINT_TO_UDINT
• – ◦ • – ◦ • – ◦ •

Introduction
The function block `F_USINT_TO_UDINT` converts a `USINT` value (8-bit unsigned integer) to a `UDINT` value (32-bit unsigned integer). This block is particularly useful in scenarios where type conversion without data loss is required.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. This input is connected to the data input `IN`.

Event Outputs
- **CNF**: Signals the completion of the conversion. This output is connected to the data output `OUT`.

Data Inputs
- **IN** (`USINT`): The input value to be converted.

Data Outputs
- **OUT** (`UDINT`): The converted output value.

Adapters
No adapters are available.

Functionality
This function block directly assigns the input value `IN` of type `USINT` to the output value `OUT` of type `UDINT`. Since `UDINT` covers a larger range of values than `USINT`, the conversion occurs without data loss.

Technical Features
- **Algorithm**: The algorithm `REQ` performs the assignment `OUT := IN`.
- **Data Integrity**: The original value is preserved during conversion to a larger data type.

State Overview
The function block has no internal states. The conversion occurs immediately upon the occurrence of the `REQ` event.

Application Scenarios
- **Type Conversion**: In control systems where different data types need to be processed.
- **Data Processing**: In communication between components that use different data types.

⚖️ Comparison with similar function blocks
- **F_USINT_TO_INT**: Converts `USINT` to `INT`, but with a potential sign change.
- **F_USINT_TO_UINT**: Converts `USINT` to `UINT` (16-bit), which offers a smaller value range than `UDINT`.

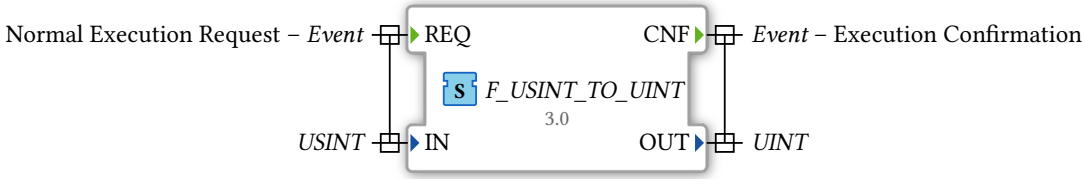
🛠️ Related Exercises
- Exercise_111

Conclusion
The `F_USINT_TO_UDINT` function block is a simple and efficient building block for the safe conversion of `USINT` to `UDINT`. Its direct assignment and the wider range of values for the output type make it ideal for applications where data integrity must be guaranteed.

F_USINT_TO_UINT

• – ◦ • – ◦ • – ◦ •

Introduction
The function block `F_USINT_TO_UINT` converts a `USINT` value (Unsigned Short Integer) to a `UINT` value (Unsigned Integer). This block is particularly useful in scenarios where a type conversion without data loss is required.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input value of type `USINT` to be converted.

Data Outputs

- **OUT**: The output value of type `UINT`, containing the result of the conversion.

Adapters

No adapters available.

Functionality

This function block directly assigns the input value `IN` of type `USINT` to the output value `OUT` of type `UINT`. Since `USINT` has a smaller value range than `UINT`, the conversion occurs without data loss.

The algorithm is triggered by the event `REQ` and outputs the result via the event `CNF`.

Technical Features

- The function block is implemented in the ST (Structured Text) programming language.
- The conversion occurs without additional calculations or checks, as `USINT` can be automatically converted to `UINT`.

State Overview

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Type conversion in control applications, where `USINT` values need to be converted to `UINT`.
- Use in data processing chains where different data types need to be made compatible.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion function blocks such as `F_INT_TO_REAL` or `F_BOOL_TO_STRING`, this function block is specifically for conversion between two unsigned integer types.
- Similar function blocks might perform additional checks or scaling, which is not necessary here.

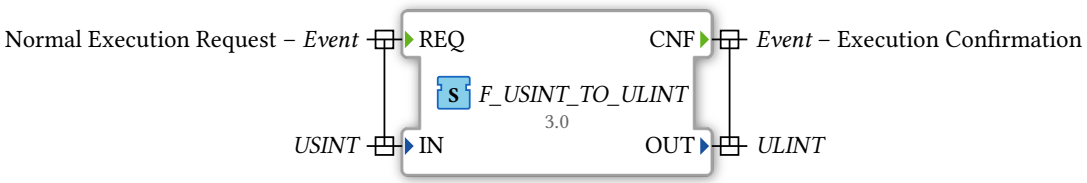
Conclusion

The `F_USINT_TO_UINT` function block is a simple and efficient building block for converting `USINT` to `UINT`. It is ideally suited for applications where fast and lossless type conversion is required.

F_USINT_TO_ULINT

• – ◦ • – ◦ • – ◦ •

The function block `F_USINT_TO_ULINT` converts a `USINT` value (8-bit unsigned integer) to a `ULINT` value (64-bit unsigned integer). This conversion is useful when data needs to be exchanged or processed between systems with different word lengths.



- `REQ`: Starts the execution of the conversion. This input is linked to the data input `IN`.
- `CNF`: Signals the completion of the conversion. This output is linked to the data output `OUT`.

- `IN` (USINT): The input value to be converted.
- `OUT` (ULINT): The converted output value.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters available.

The function block performs a direct conversion of the input value `IN` of type `USINT` to the output value `OUT` of type `ULINT`. The conversion is lossless because the value range of `USINT` (0 to 255) can be fully mapped to `ULINT` (0 to 2^64-1).

- The conversion is performed using a single algorithm (`REQ`), which is executed when the `REQ` event is triggered.
- The algorithm is implemented in ST (Structured Text) and consists of a simple assignment: `OUT := IN;`.

The function block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

- Data conversion in control systems that use different word widths.
- Integration of peripheral devices that provide data in different formats.
- General type conversion in IEC 61131-3-based programs.
- Similar function blocks such as `F_SINT_TO_LINT` or `F_UINT_TO_UDINT` perform similar conversions, but between different data types.
- `F_USINT_TO_ULINT` is specifically optimized for converting 8-bit to 64-bit unsigned integers.

The function block `F_USINT_TO_ULINT` provides a simple and efficient way to convert values of type `USINT` to `ULINT`. Its clear interface and direct functionality make it ideal for applications requiring type conversion without additional logic.

Functionality

Application Scenarios

State Overview

Technical Features

⚖️ Vergleich mit ähnlichen Bausteinen

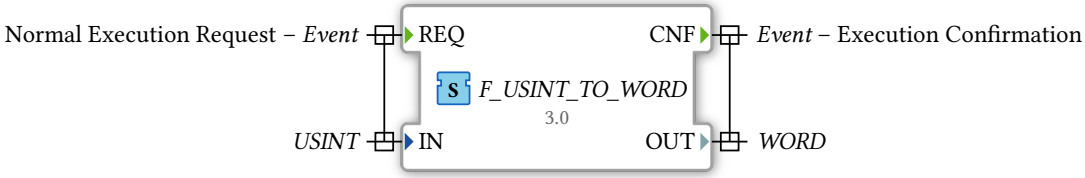
Conclusion

F_USINT_TO_WORD

• – ○ • – ○ • – ○ •

Introduction

The function block `F_USINT_TO_WORD` converts an unsigned 8-bit integer value (USINT) into a 16-bit word value (WORD). This conversion is particularly necessary in scenarios where data with different bit widths needs to be processed or transmitted.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (USINT): The unsigned 8-bit integer value to be converted.

Data Outputs

- **OUT** (WORD): The resulting 16-bit word value after conversion.

Adapters

- No adapters are present.

Functionality

The function block performs the conversion by taking the USINT value from `IN` and converting it to a WORD value. This is done by direct assignment using the function `USINT_TO_WORD(IN)`. The conversion is triggered by the event `REQ` and acknowledged by `CNF`.

Technical Features

- The conversion is lossless, as a USINT (0-255) can be easily converted into a WORD (0-65535).
- The function block is deterministic and has no internal states.

State Overview

The function block has no internal states. Execution is purely event-driven:

1. The conversion is performed upon receipt of `REQ`.
2. Upon completion, `CNF` is triggered.

Application Scenarios

- Data preparation for communication protocols that expect WORD data.
- Integration of sensor data in USINT format into systems that process WORD.
- General type conversion in automation applications.

⚖️ Comparison with Similar Building Blocks

- Compared to generic conversion blocks, `F_USINT_TO_WORD` is specialized and optimized for the specific conversion of USINT to WORD.
- Other blocks might offer additional features such as scaling or error handling, but they are more complex.

Conclusion

The `F_USINT_TO_WORD` function block is a simple and efficient building block for the specific conversion of USINT to WORD. Its simplicity and determinism make it a reliable choice in scenarios that require this conversion.

USINT conversions

Here you will find functions for converting the USINT data type.

🛠️ Related exercises

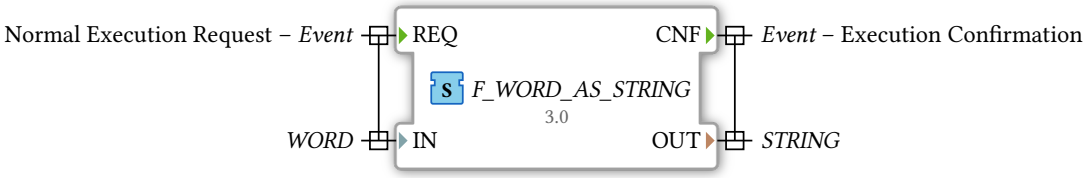
- Funk_IX_TO_DataPanel_MI_QX
- Funk_IXA_TO_DataPanel_MI_QXA
- IX_TO_Aux_QX
- IXA_TO_Aux_QXA
- IX_TO_logiBUS_LED_strip_QX
- SIN_TO_PCAN_Callback_Byte

WORD

F_WORD_AS_STRING

• – ◦ • – ◦ • – ◦ •

The function block `F_WORD_AS_STRING` converts a `WORD` data type to a `STRING`. This is particularly useful when numeric values need to be output or further processed in a human-readable format.



- **REQ**: Starts the conversion. When this event is triggered, the value of the data input `IN` is processed.
- **CNF**: Signals the completion of the conversion. This event, along with the converted value, is output at data output `OUT`.
- **IN**: Expects an input value of type `WORD`, which is to be converted to `STRING`.
- **OUT**: Outputs the converted value as `STRING`.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- No adapters are available.

Functionality

The function block uses the algorithm `REQ` to convert the `WORD` value at input `IN` to `STRING`. The conversion is performed by the function `WORD_AS_STRING(IN)`. After successful conversion, the event `CNF` is triggered, and the converted value is provided at output `OUT`.

- The function block has a simple structure and requires no additional parameters or complex logic.
- The conversion is performed directly without intermediate steps.

1. **Initialization**: The function block waits for the event `REQ`.
2. **Processing**: Upon receiving `REQ`, the value of `IN` is converted.
3. **Completion**: After the conversion, `CNF` is triggered and the value is output to `OUT`.

- Displaying numeric values in user interfaces.
- Logging `WORD` values in a readable format.
- Further processing of numeric values in systems that expect `STRING` data.
- Compared to generic conversion blocks, `F_WORD_AS_STRING` specializes in converting `WORD` to `STRING` and is therefore more efficient for this specific use case.

The F_WORD_AS_STRING function block is a simple and efficient tool for converting WORD values to STRING. Its clear interface and direct functionality make it ideal for applications requiring fast and straightforward conversion.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

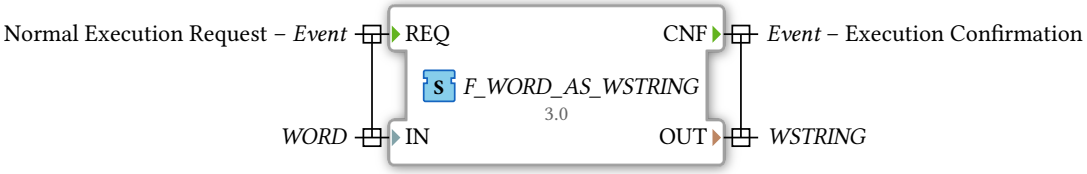
Conclusion

F_WORD_AS_WSTRING

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_WORD_AS_WSTRING converts a WORD data type to a WSTRING data type. This function block is particularly useful in scenarios where numeric values need to be converted into a string, for example, for display or logging.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the value at the data input IN is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion. The converted value is available at data output OUT.

Data Inputs

- **IN** (WORD): The input value to be converted to WSTRING.

Data Outputs

- **OUT** (WSTRING): The output value after conversion.

Adapters

This function block has no adapters.

Functionality

When the event input REQ is triggered, the algorithm is executed that converts the value at input IN of type WORD to WSTRING. The result is provided at output OUT, and the event CNF signals successful execution.

Technical Features

- The conversion is performed directly without intermediate steps.
- The function block is optimized for simple conversions and requires no additional configuration.

State Overview

1. **Idle**: Waits for the REQ event.
2. **Processing**: Performs the conversion.
3. **Completed**: Signals completion with CNF.

Application Scenarios

- Displaying numeric values in a user interface.
- Logging `WORD` values as readable strings.
- Data conversion for communication with systems that expect `WSTRING`.

⚖️ Comparison with similar function blocks

- Compared to generic conversion function blocks, `F_WORD_AS_WSTRING` is specialized and therefore more efficient for this specific task.
- Other function blocks might support additional parameters or more complex conversions, which is not necessary here.

Conclusion

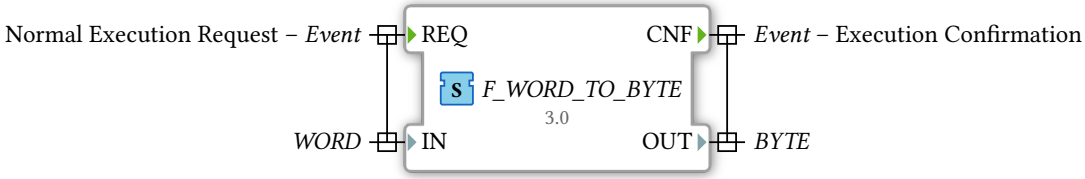
The `F_WORD_AS_WSTRING` function block is a simple and efficient function block for converting `WORD` to `WSTRING`. Its strength lies in its simplicity and direct applicability for specific conversion tasks.

F_WORD_TO_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_TO_BYTE` is used to convert a `WORD` data type to a `BYTE` data type. It is part of the `iec61131::conversion` package and is suitable for applications where type conversion between these two data types is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN** (`WORD`): The input value to be converted from `WORD` to `BYTE`.

Data Outputs

- **OUT** (`BYTE`): The converted output value of type `BYTE`.

Adapters

No adapters are defined for this function block.

Operation

The function block performs the conversion when the event `REQ` is received. The input value `IN`, of type `WORD`, is converted to a `BYTE` value using the function `WORD_TO_BYTE` and output `OUT`. After the conversion is complete, the event `CNF` is triggered.

Technical Features

- The function block is a simple function block (`SimpleFB`) and uses an ST algorithm for the conversion.
- The conversion is performed directly and without any additional delay.

State Overview

1. **Idle State**: The function block waits for the event `REQ`.
2. **Conversion State**: Upon receiving `REQ`, the conversion is performed and `CNF` is triggered.

Application Scenarios

- Type conversion in control applications where `WORD` data needs to be converted to `BYTE` data.
- Integration into larger control systems that require compatible data type conversion.

⚖️ Comparison with Similar Function Blocks

- Compared to generic conversion functions, this function block offers a specialized and optimized solution for converting `WORD` to `BYTE`.
- Other similar blocks might offer additional features such as error handling or optional parameters, which is not the case here.

Conclusion

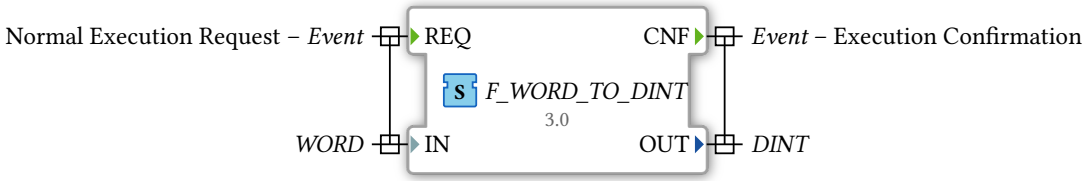
The `F_WORD_TO_BYTE` function block is an efficient and simple solution for converting `WORD` to `BYTE` data. It is particularly suitable for applications where fast and direct type conversion is required.

F_WORD_TO_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_TO_DINT` converts a `WORD` data type to a `DINT` data type. This function block is particularly useful in scenarios where type conversion between these two data types is required, for example, when processing data from different sources or adapting data for different system components.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This input triggers the conversion. It is linked to the data input `IN`.

Event Outputs

- **CNF**: Execution Confirmation. This output signals the completion of the conversion. It is linked to the data output `OUT`.

Data Inputs

- **IN**: Input variable of type `WORD` containing the value to be converted.

Data Outputs

- **OUT**: Output variable of type `DINT` containing the result of the conversion.

Adapters

This function block has no adapters.

Functionality

The function block `F_WORD_TO_DINT` performs the conversion from a `WORD` to a `DINT` data type. When the `REQ` event is triggered, the value at input `IN` is read and converted using the function `WORD_TO_DINT`. The result is output to `OUT`, and the `CNF` event signals the completion of the operation.

Technical Features

- The function block has a simple structure and contains only one algorithm for the conversion.
- The conversion occurs directly and without any additional delay as soon as the `REQ` event arrives.

State Overview

The function block has no internal states. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Type conversion in control systems where data of different types needs to be processed.
- Integration of components that use different data types.
- Data preparation for further processing steps.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion function blocks such as `F_INT_TO_REAL` or `F_BOOL_TO_BYTE`, `F_WORD_TO_DINT` is specifically designed for converting `WORD` to `DINT`.
- Similar function blocks might offer additional features such as range checks or scaling, which is not the case here.

Conclusion

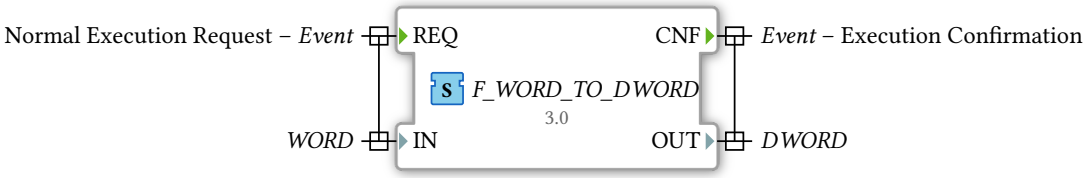
The function block `F_WORD_TO_DINT` is a simple and efficient tool for converting `WORD` values to `DINT` values. Its clear interface structure and direct functionality make it a reliable component in control applications where such type conversions are required.

F_WORD_TO_DWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_TO_DWORD` converts a `WORD` value to a `DWORD` value. It is part of the `iec61131::conversion` package and enables simple and efficient type conversion.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `WORD` to be converted.

Data Outputs

- **OUT**: The output of type `DWORD`, containing the result of the conversion.

Adapters

No adapters are present.

Functionality

The function block performs the conversion from `WORD` to `DWORD` as soon as the event `REQ` is received. The conversion is performed using the function `WORD_TO_DWORD(IN)`. After successful conversion, the event `CNF` is triggered.

Technical Features

- The function block is a simple function block (`SimpleFB`) and uses an ST algorithm for the conversion.
- The conversion is performed directly without any additional processing steps.

State Overview

1. **Idle**: Waits for the event `REQ`.
2. **Processing**: Performs the conversion.

3. **Completed**: Sends the event `CNF` and returns to the idle state.

Application Scenarios

- Type conversion in control applications where `WORD` values need to be converted to `DWORD` values.
- Integration into larger control systems that require type compatibility.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks, `F_WORD_TO_DWORD` offers specialized functionality for converting `WORD` to `DWORD` without additional features or complexity.

Conclusion

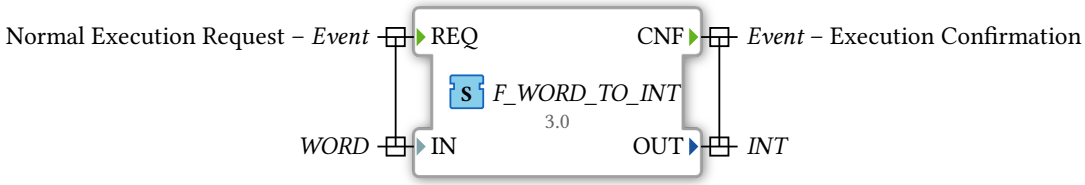
The `F_WORD_TO_DWORD` function block is a simple and efficient tool for type conversion in IEC 61131-3-based control systems. Its clear interface structure and direct functionality make it a reliable solution for conversion tasks.

F_WORD_TO_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_TO_INT` converts a `WORD` data type to a `INT` data type. It is part of the `iec61131::conversion` package and enables simple and efficient type conversion within IEC 61131-3-based control systems.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `WORD`, which is to be converted to a `INT` value.

Data Outputs

- **OUT**: The output of type `INT`, which contains the result of the conversion.

Adapters

- No adapters are present.

Operation

The function block performs the conversion as soon as the event `REQ` is triggered. The conversion is performed using the function `WORD_TO_INT`, which converts the value of input `IN` to a value of `INT` and passes the result to output `OUT`. Upon completion of the conversion, the event `CNF` is triggered.

Technical Features

- The function block is a simple function block (`SimpleFB`) and uses an ST algorithm for the conversion.
- The conversion is performed directly and without any additional delay.

State Overview

The function block has no internal states. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Type conversion in control applications where `WORD` data needs to be converted to `INT` data.
- Integration into larger control systems that require type compatibility.

⚖️ Comparison with Similar Blocks

- Similar function blocks could be `F_BYTE_TO_INT` or `F_DWORD_TO_INT`, but these convert different data types.
- Unlike more complex conversion blocks, `F_WORD_TO_INT` offers a simple and direct solution for the specific conversion of `WORD` to `INT`.

Conclusion

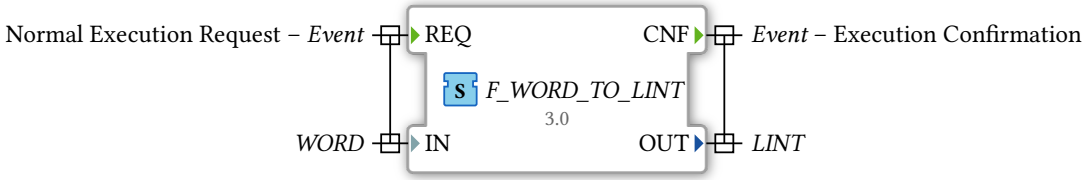
The `F_WORD_TO_INT` function block is an efficient and easy-to-use tool for type conversion in IEC 61131-3-based systems. Its clear interface and direct operation make it ideal for applications requiring fast and reliable conversion.

F_WORD_TO_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_TO_LINT` converts a `WORD` data type to a `LINT` data type. It is part of the `iec61131::conversion` package and enables simple and efficient type conversion within IEC 61499-based control systems.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals successful completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `WORD` to be converted.

Data Outputs

- **OUT**: The output of type `LINT`, containing the result of the conversion.

Adapters

- No adapters are present.

Functionality

This function block performs the conversion from `WORD` to `LINT` as soon as the event `REQ` is triggered. The conversion is performed using the function `WORD_TO_LINT`, which converts the input value `IN` and outputs the result to `OUT`. Upon successful conversion, the event `CNF` is triggered.

Technical Features

- The function block is a simple algorithm without a state machine.
- The conversion occurs directly and without any additional delay.
- The block is optimized for use in real-time control systems.

State Overview

Since it is a simple function block, there are no state transitions or complex state logic. Execution occurs immediately upon receiving the `REQ` event.

Application Scenarios

- Type conversion in control applications where `WORD` data needs to be converted to `LINT`.
- Integration into larger control systems that process different data types.
- Use in data processing chains where bit width extension is required.

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_WORD_TO_LINT` is specialized and therefore more efficient.
- Other blocks, such as `F_INT_TO_LINT` or `F_DWORD_TO_LINT`, offer similar functionality, but for different data types.

Conclusion

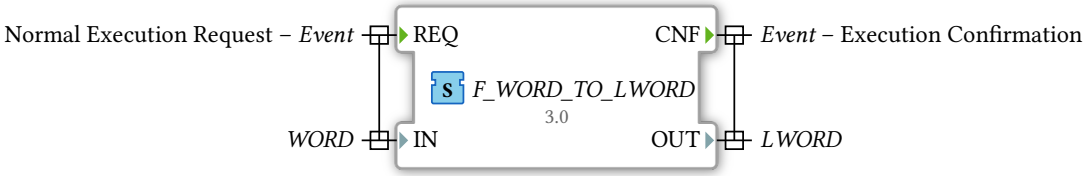
The `F_WORD_TO_LINT` function block is a simple and efficient building block for type conversion from `WORD` to `LINT`. It is ideally suited for applications requiring fast and direct conversion and can be easily integrated into existing control systems.

F_WORD_TO_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_TO_LWORD` is used to convert a `WORD` data type to a `LWORD` data type. This function block is particularly useful in scenarios where an extension of the data width is required, for example, in communication between systems with different data word sizes.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN**: Expects an input of type `WORD`, which is to be converted to `LWORD`.

Data Outputs

- **OUT**: Outputs the converted result of type `LWORD`.

Adapters

This function block has no adapters.

Functionality

The function block performs the conversion as soon as the event `REQ` is received. The conversion is performed using the function `WORD_TO_LWORD`, which converts the input value `IN` of type `WORD` into the output value `OUT` of type `LWORD`. Upon successful conversion, the event `CNF` is triggered.

Technical Features

- The conversion is direct and requires no additional parameters or state management.

- The function block is suitable for use in real-time systems because it does not involve any delays or complex calculations.

State Overview

The function block has no internal states. The operation is stateless and is re-executed with each `REQ` event.

Application Scenarios

- Extending the data width for subsequent processing steps.
- Ensuring compatibility between systems with different word sizes.
- Simple data type conversion in control and automation systems.

⚖️ Comparison with Similar Function Blocks

Compared to generic conversion blocks, `F_WORD_TO_LWORD` offers a specialized and efficient solution for converting `WORD` to `LWORD`. Other blocks might be more general, but less optimized for this specific task.

Conclusion

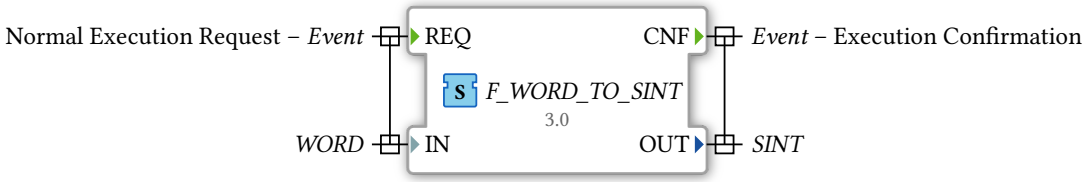
The `F_WORD_TO_LWORD` function block is a simple yet effective block for the specific conversion of `WORD` to `LWORD`. Its simplicity and direct functionality make it a reliable choice in many automation and control scenarios.

F_WORD_TO_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_TO_SINT` converts a `WORD` data type to a `SINT` data type. It is part of the `iec61131::conversion` package and enables simple and efficient type conversion within 4diac IDE applications.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN** (`WORD`): The input value to be converted from `WORD` to `SINT`.

Data Outputs

- **OUT** (`SINT`): The converted output value of type `SINT`.

Adapters

This function block has no adapters.

Functionality

When the `REQ` event is triggered, the algorithm is executed. This algorithm converts the `WORD` value at input `IN` into a `SINT` value and outputs the result at output `OUT`. Subsequently, the `CNF` event is triggered to signal the successful completion of the conversion.

Technical Features

- The function block is a simple function block (`SimpleFB`) and performs the conversion in a single algorithm step.
- The conversion is performed using the built-in function `WORD_TO_SINT`.

State Overview

The function block has no internal states. Execution occurs directly upon triggering the `REQ` event.

Application Scenarios

- Type conversion in control applications where `WORD` data needs to be converted to `SINT` data.
- Integration into larger control systems that process different data types.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_INT_TO_SINT` or `F_DWORD_TO_SINT` offer conversions between other data types but are comparable in their functionality.

Conclusion

The `F_WORD_TO_SINT` function block offers a simple and efficient way to convert `WORD` data to `SINT` data. Its clear interface structure and direct functionality make it ideal for applications requiring fast and reliable type conversion.

🌐 Related topic subpages on ms-muc-docs.de

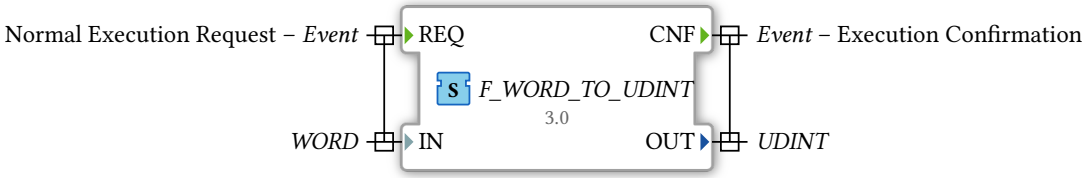
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

F_WORD_TO_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_TO_UDINT` converts a `WORD` data type to a `UDINT` data type. This function block is particularly useful in scenarios where type conversion is required to prepare data for further processing.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event is triggered, the input value `IN` is processed.

Event Outputs

- **CNF**: Signals the completion of the conversion and outputs the result via output `OUT`.

Data Inputs

- **IN**: The input value of type `WORD`, which is to be converted to `UDINT`.

Data Outputs

- **OUT**: The result of the conversion as `UDINT`.

Adapters

This function block has no adapters.

Functionality

When the `REQ` event is triggered, the algorithm is started, which converts the `WORD` value at the `IN` input into a `UDINT` value. The result is provided at the `OUT` output, and the `CNF` event signals successful execution.

Technical Features

- The function block uses the built-in function `WORD_TO_UDINT` for the conversion.
- The conversion is performed without additional parameters or complex logic, enabling fast and efficient processing.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Processing State**: Performs the conversion.
3. **Completion State**: Sends the `CNF` event and outputs the result.

Application Scenarios

- Type conversion in control applications.
- Preparing data for further calculations or storage.
- Integration into larger automation solutions where different data types need to be processed.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks such as `F_INT_TO_REAL` or `F_BOOL_TO_BYTE` offer similar functionality for other data types.
- `F_WORD_TO_UDINT` specializes in converting `WORD` to `UDINT` and therefore offers an optimized solution for this specific use case.

Conclusion

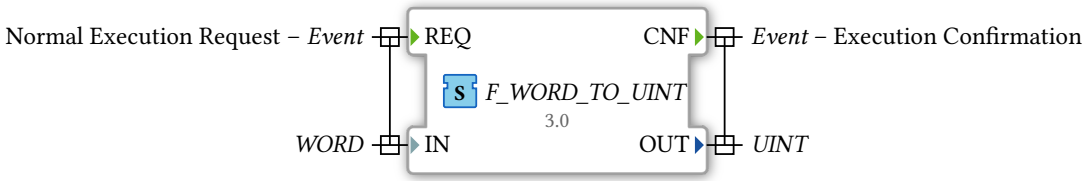
The `F_WORD_TO_UDINT` function block is a simple and efficient building block for converting `WORD` values to `UDINT` values. Its clear interface and simple functionality make it a reliable component in automation projects.

F_WORD_TO_UINT

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_WORD_TO_UINT` converts a `WORD` data type to a `UINT` data type. This conversion is particularly useful in scenarios where type conversion between these two data types is required, for example, in communication between different systems or when processing data from different sources.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: The input of type `WORD` to be converted.

Data Outputs

- **OUT**: The output of type `UINT`, containing the result of the conversion.

Adapters

This function block has no adapters.

Operation

The function block performs the conversion from `WORD` to `UINT` as soon as the event `REQ` is received. The conversion is performed using the function `WORD_TO_UINT(IN)`. After successful conversion, the result is output via output `OUT`, and the event `CNF` is triggered.

Technical Features

- The function block has a simple structure and contains only one algorithm for the conversion.
- The conversion is performed directly without any additional processing steps.

State Overview

The function block has no internal states. The conversion is re-executed with each `REQ` event.

Application Scenarios

- Type conversion in control systems.
- Data processing in industrial automation solutions.
- Communication between systems with different data types.

⚖️ Comparison with Similar Building Blocks

Compared to other conversion building blocks, `F_WORD_TO_UINT` is specifically designed to convert `WORD` to `UINT`. Other building blocks might offer additional functions or a wider range of conversions, but they may be more complex in their structure.

Conclusion

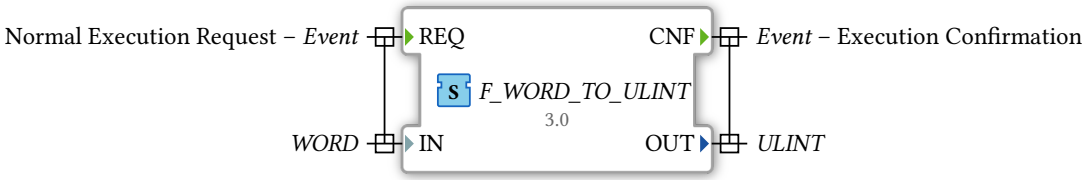
The `F_WORD_TO_UINT` function block is a simple and efficient tool for the specific conversion of `WORD` to `UINT`. Its simplicity makes it particularly suitable for applications where fast and direct type conversion is required.

F_WORD_TO_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_TO_ULINT` is used to convert a `WORD` data type to a `ULINT` data type. This function block is particularly useful in scenarios where data formats need to be extended or converted.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- **IN**: Expects a `WORD` value to be converted.

Data Outputs

- **OUT**: Outputs the converted `ULINT` value.

Adapters

No adapters are available.

Functionality

This function block performs the conversion from `WORD` to `ULINT` as soon as the `REQ` event occurs. The conversion is performed using the built-in function `WORD_TO_ULINT`. After successful conversion, the `CNF` event is triggered.

Technical Features

- This function block is part of the `iec61131::conversion` package.
- The conversion is performed without additional parameters or settings.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Done**: Sends the `CNF` event and outputs the result.

Application Scenarios

- Data processing in control systems where different data types need to be made compatible.
- Data extension for further calculations or storage.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks might offer direct conversions between other data types, e.g., `F_WORD_TO_INT`.
- This function block specializes in converting to `ULINT`, which is particularly relevant for large, unsigned numbers.

Conclusion

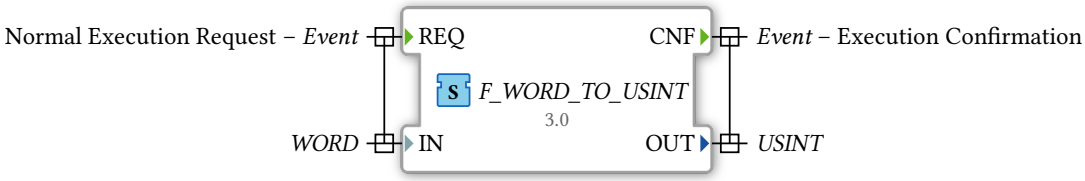
The `F_WORD_TO_ULINT` function block is a simple yet effective function block for converting `WORD` to `ULINT`. Its clear interface and simple functionality make it a reliable solution in many automation scenarios.

F_WORD_TO_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WORD_TO_USINT` converts a `WORD` data type to a `USINT` data type. It is part of the `iec61131::conversion` package and is suitable for applications where type conversion between these two data types is required.



Interface Structure

Event Inputs

- `REQ` (Event): Starts the execution of the function block. This input is linked to the data input `IN`.

Event Outputs

- `CNF` (Event): Signals the completion of the conversion. This output is linked to the data output `OUT`.

Data Inputs

- `IN` (WORD): The input value to be converted.

Data Outputs

- `OUT` (USINT): The converted output value.

Adapters

No adapters are present.

Functionality

The function block performs the conversion from `WORD` to `USINT` as soon as the event `REQ` is received. The conversion is performed using the function `WORD_TO_USINT`, which converts the input value `IN` into the output value `OUT`. Upon completion of the conversion, the event `CNF` is triggered.

Technical Features

- The function block is a simple function block (SimpleFB) and uses an ST algorithm for the conversion.
- The conversion occurs directly and without any additional delay.

State Overview

1. **Idle State**: The function block waits for the event `REQ`.
2. **Execution State**: Upon receiving `REQ`, the conversion is performed.
3. **Confirmation State**: After successful conversion, `CNF` is triggered, and the function block returns to its idle state.

Application Scenarios

- Type conversion in control applications where a `WORD` value needs to be converted to a `USINT` value.
- Integration into larger control systems that need to process different data types.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion function blocks such as `F_INT_TO_REAL` or `F_BOOL_TO_STRING`, `F_WORD_TO_USINT` specializes in converting `WORD` to `USINT`.
- Similar building blocks could be `F_USINT_TO_WORD`, which performs the reverse conversion.

Conclusion

The function block `F_WORD_TO_USINT` offers a simple and efficient way to convert `WORD` values to `USINT` values. Its clear interface structure and simple functionality make it ideal for applications requiring such type conversion.

Word Conversions

Here you will find functions for converting Word data types. Word Conversions ...

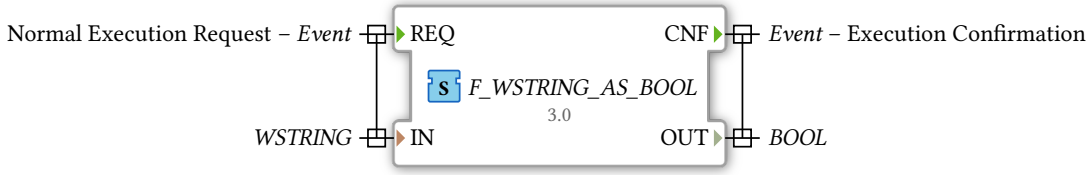
WSTRING

F_WSTRING_AS_BOOL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WSTRING_AS_BOOL` is used to convert a WSTRING value to a BOOL value. This is particularly useful in scenarios where strings need to be converted to Boolean values, for example, when processing user input or interpreting configuration data.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. Triggered after successful conversion.

Data Inputs

- **IN**: WSTRING. The input value to be converted to a BOOL value.

Data Outputs

- **OUT**: BOOL. The result of the conversion.

Adapters

No adapters available.

Functionality

The function block performs the conversion when the event `REQ` is received. The conversion is carried out using the function `WSTRING_AS_BOOL`, which converts the WSTRING input value `IN` into a BOOL value `OUT`. After successful conversion, the event `CNF` is triggered.

Technical Features

- The conversion is performed directly and without any additional delay.
- The function block is implemented as a simple function block (SimpleFB) and uses an ST algorithm for the conversion.

State Overview

1. **Idle**: Waits for the event `REQ`.
2. **Processing**: Performs the conversion.
3. **Complete**: Triggers the event `CNF` and returns to the idle state.

Application Scenarios

- Processing user input in string form.
- Interpreting configuration files that store Boolean values as strings.
- Interfaces between systems that use different data types.

⚖️ Comparison with Similar Function Blocks

- **F_STRING_AS_BOOL**: Converts STRING to BOOL instead of WSTRING.
- **F_BOOL_AS_WSTRING**: Performs the reverse conversion (BOOL to WSTRING).

Conclusion

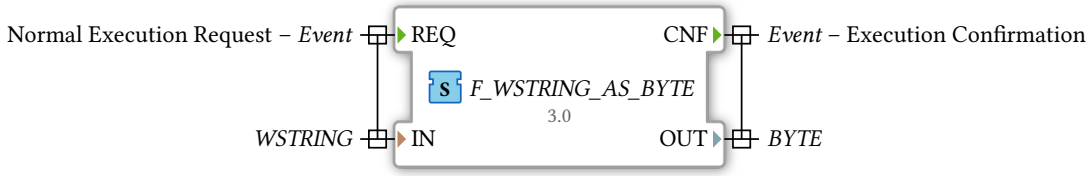
The `F_WSTRING_AS_BOOL` function block offers a simple and efficient way to convert WSTRING values to BOOL values. Its clear interface structure and direct functionality make it a useful tool in various application scenarios.

F_WSTRING_AS_BYTE

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WSTRING_AS_BYTE` is used to convert a WSTRING data type to a BYTE data type. This block is particularly useful in scenarios where converting wide strings into individual bytes is required.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. The input value `IN` is processed upon this event.

Event Outputs

- **CNF**: Signals the completion of processing and outputs the result via the output `OUT`.

Data Inputs

- **IN** (WSTRING): The input value to be converted into a BYTE value.

Data Outputs

- **OUT** (BYTE): The result of the conversion.

Adapters

- No adapters available.

Functionality

When the `REQ` event occurs, the function block converts the WSTRING input `IN` into a BYTE value. The result is output via `OUT`, and the `CNF` event signals the successful completion of the operation.

Technical Features

- The function block uses the `WSTRING_AS_BYTE` function for the conversion.
- This is a simple function block without internal state management.

State Overview

The function block has no internal states. Processing occurs directly upon the occurrence of the `REQ` event.

Application Scenarios

- Conversion of WSTRING data to BYTE values for further processing in systems that only support BYTE data.
- Use in communication protocols where strings need to be converted to bytes.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks such as `F_STRING_AS_BYTE`, this one is specifically designed for WSTRING data types.
- Similar blocks might have additional parameters or states, but this function block is limited to simple conversion.

Conclusion

The `F_WSTRING_AS_BYTE` function block offers a simple and efficient way to convert WSTRING data into BYTE values. Its clear interface structure and direct functionality make it a reliable solution for corresponding use cases.

F_WSTRING_AS_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WSTRING_AS_DINT` converts a WSTRING value to a DINT value. It is part of the package `iec61131::conversion` and enables the easy conversion of wide strings to integer values.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. Triggered after successful conversion.

Data Inputs

- **IN**: Input variable of type `WSTRING`. Contains the string to be converted.

Data Outputs

- **OUT**: Output variable of type `DINT`. Contains the result of the conversion.

Adapters

This function block does not use any adapters.

Functionality

The function block converts the value of the input variable `IN` of type `WSTRING` to a value of type `DINT` and stores the result in the output variable `OUT`. The conversion is performed by calling the function `WSTRING_AS_DINT` within the algorithm `REQ`.

Technical Features

- The conversion is performed directly via the built-in function `WSTRING_AS_DINT`.
- The function block is simple and has no additional states or complex logic.

State Overview

The function block has no internal states. Execution occurs directly upon receiving the `REQ` event.

Application Scenarios

- Conversion of user input into numeric values.
- Processing of strings containing numeric data in control systems.

⚖️ Comparison with Similar Function Blocks

- Similar function blocks like `F_STRING_AS_DINT` convert simple strings (`STRING`) to `DINT`, while this block processes wide strings (`WSTRING`).
- Compared to generic conversion blocks, this one is specifically designed to convert `WSTRING` to `DINT`.

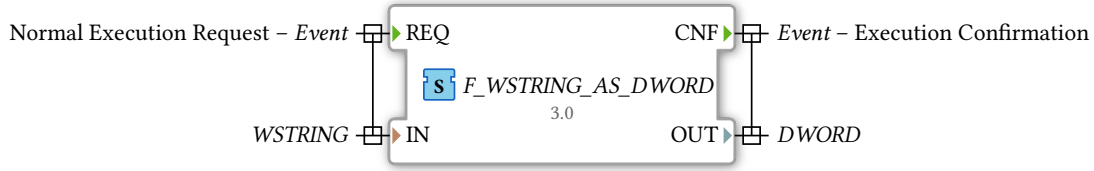
Conclusion

The `F_WSTRING_AS_DINT` function block provides a simple and efficient way to convert wide strings to integer values. It is particularly useful in scenarios where wide strings need to be processed.

F_WSTRING_AS_DWORD

• – ◦ • – ◦ • – ◦ •

The function block `F_WSTRING_AS_DWORD` is used to convert a WSTRING value to a DWORD value. This functionality is particularly useful when strings need to be converted into numeric values, for example, for further processing in control algorithms.



- **REQ**: Normal execution request. Triggers the conversion.
- **CNF**: Execution confirmation. Triggered after successful conversion.
- **IN**: Input of type WSTRING. The string to be converted to a DWORD value.
- **OUT**: Output of type DWORD. The converted numeric value.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters available.

Functionality

The function block performs the conversion as soon as the event `REQ` is received. The conversion is carried out using the function `WSTRING_AS_DWORD`, which converts the WSTRING input value into a DWORD value. After successful conversion, the event `CNF` is triggered, and the converted value is available at output `OUT`.

- The function block is a simple function block (SimpleFB) and has a single algorithm that is executed when `REQ` is triggered.

The function block has no internal state. The conversion is performed anew with each `REQ` event.

- Conversion of user input (e.g., via HMI) into numeric values for control.
- Processing of strings containing numeric information in control algorithms.
- Compared to other conversion blocks such as `F_STRING_AS_DWORD`, this one is specifically designed for WSTRING input.
- Similar function blocks might offer additional parameters or error handling, but this function block is limited to basic conversion.

The `F_WSTRING_AS_DWORD` function block offers a simple and efficient way to convert WSTRING values to DWORD values. Its simplicity makes it particularly suitable for applications where direct conversion without additional complexity is required.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

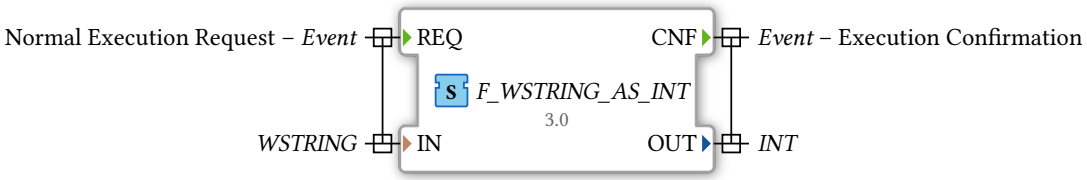
Conclusion

F_WSTRING_AS_INT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WSTRING_AS_INT` is used to convert a WSTRING value to an INT value. This block is particularly useful when strings need to be converted into numeric values, for example, when processing user input or interpreting text data.



Interface Structure

Event Inputs
- **REQ**: This event input triggers the execution of the function block. It is linked to the data input `IN`.

Event Outputs
- **CNF**: This event output signals the successful execution of the function block. It is linked to the data output `OUT`.

Data Inputs
- **IN**: This input expects a WSTRING value, which is to be converted to an INT value.

Data Outputs
- **OUT**: This output provides the converted INT value.

Adapters
This function block has no adapters.

Functionality
The function block performs the conversion as soon as the event `REQ` is received. The conversion is carried out using the function `WSTRING_AS_INT`, which converts the WSTRING value from `IN` into an INT value and outputs the result to `OUT`. The event `CNF` is then triggered to signal successful execution.

Technical Features
- The function block is a simple block without internal states or complex logic.
- The conversion is performed directly and without delay.

State Overview
Because it is a simple function block, there are no state transitions or internal states.

Application Scenarios
- Processing user input that is in string form but must be interpreted as numeric values.
- Interpreting text data from external sources that needs to be converted into numeric values.
- Integration into systems that need to process strings and numeric values.

⚖️ Comparison with Similar Blocks
- Compared to other conversion blocks such as `F_STRING_AS_INT` or `F_WSTRING_AS_REAL`, this block is specifically designed for converting WSTRING to INT.

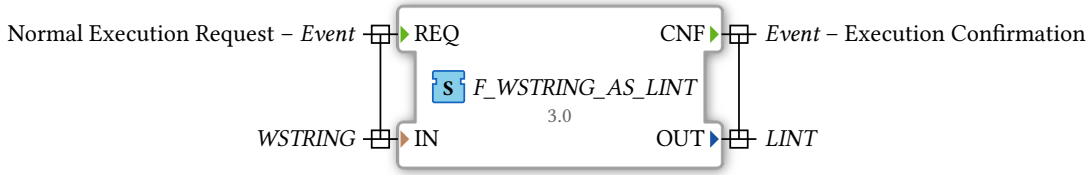
Other blocks might offer additional features like error handling or formatting, but this block is limited to basic conversion.

Conclusion
The `F_WSTRING_AS_INT` function block is a simple and efficient tool for converting WSTRING values to INT values. Its clear interface and straightforward functionality make it ideal for applications requiring fast and uncomplicated conversion.

F_WSTRING_AS_LINT

• − ◦ • − ◦ • − ◦ •

Introduction
The function block `F_WSTRING_AS_LINT` is used to convert a WSTRING value to a LINT value. This block is particularly useful in scenarios where strings need to be converted into numeric values, for example, when processing user input or interpreting text data.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This event input triggers the conversion. It is linked to the data input `IN`.

Event Outputs

- **CNF**: Execution confirmation. This event is output after successful conversion and is linked to the data output `OUT`.

Data Inputs

- **IN** (WSTRING): The input for the string to be converted to a LINT value.

Data Outputs

- **OUT** (LINT): The output for the converted LINT value.

Adapters

This function block does not use any adapters.

Functionality

The function block converts the WSTRING value at input `IN` to a LINT value at output `OUT` as soon as the event `REQ` is received. The conversion is performed using the function `WSTRING_AS_LINT`. After successful conversion, the event `CNF` is output.

Technical Features

- The function block has a simple structure and contains only one conversion algorithm.
- The conversion can fail if the string does not contain a valid numeric representation. In this case, the event `CNF` is not output.

State Overview

1. **Idle**: The function block waits for the event `REQ`.
2. **Processing**: Upon receiving `REQ`, the conversion is performed.
3. **Confirmation**: After successful conversion, `CNF` is output, and the function block returns to the idle state.

Application Scenarios

- Processing user input that is in string form but should be interpreted as numeric values.
- Interpreting text data from external sources that contains numeric information.

⚖️ Comparison with Similar Function Blocks

- **F_STRING_AS_LINT**: Similar function block, but for the STRING data type instead of WSTRING.
- **F_WSTRING_AS_INT**: Converts WSTRING to INT, but offers a smaller value range than LINT.

Conclusion

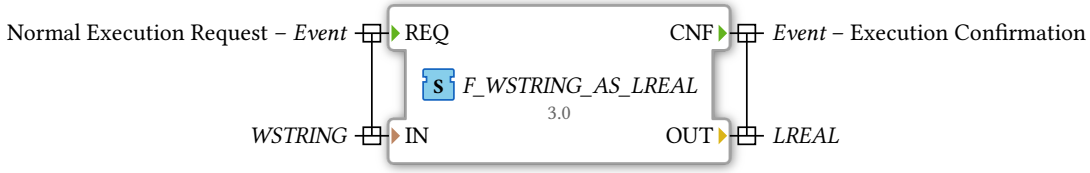
The `F_WSTRING_AS_LINT` function block is a simple and effective tool for converting strings to large numeric values. Its clear interface and simple functionality make it a reliable solution for corresponding use cases.

F_WSTRING_AS_LREAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WSTRING_AS_LREAL` is used to convert a WSTRING value into an LREAL value. This block is particularly useful when strings need to be converted into numerical values, for example, for mathematical calculations or data processing.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. When this input is activated, the conversion of the WSTRING value at input `IN` into an LREAL value begins.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. The converted LREAL value is available at output `OUT`.

Data Inputs

- **IN** (WSTRING): The input for the string to be converted to an LREAL value.

Data Outputs

- **OUT** (LREAL): The output for the converted LREAL value.

Adapters

This function block has no adapters.

Functionality

When the `REQ` event is triggered, the function block converts the WSTRING value at input `IN` into an LREAL value. The conversion is performed using the function `WSTRING_AS_LREAL`. After successful conversion, the `CNF` event is triggered, and the converted value is provided at output `OUT`.

Technical Features

- The conversion is performed directly without additional parameters.
- The function block is optimized for use in real-time systems.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Complete**: Signals completion with the `CNF` event.

Application Scenarios

- Conversion of user input into numeric values.
- Processing of text data from external sources for calculations.
- Integration into systems that need to process both strings and numeric values.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion blocks, `F_WSTRING_AS_LREAL` is specifically designed for converting WSTRING to LREAL.
- Other blocks might offer additional parameters or error handling, but this block is optimized for simplicity and efficiency.

Conclusion

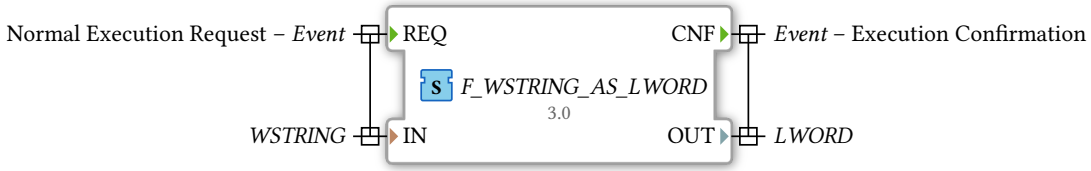
The F_WSTRING_AS_LREAL function block provides a simple and efficient way to convert WSTRING values to LREAL values. Its clear interface structure and direct functionality make it a valuable component in many applications.

F_WSTRING_AS_LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_WSTRING_AS_LWORD is used to convert a WSTRING data type to an LWORD data type. This functionality is particularly useful in scenarios where strings need to be converted into a binary form suitable for further processing.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This event input triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. This event is output after successful conversion.

Data Inputs

- **IN**: Input of type WSTRING. This is the string to be converted.

Data Outputs

- **OUT**: Output of type LWORD. This is the result of the conversion.

Adapters

No adapters available.

Functionality

The function block performs the conversion from WSTRING to LWORD as soon as the event REQ is received. The conversion is carried out using the integrated function WSTRING_AS_LWORD. After successful conversion, the event CNF is output, along with the converted value.

Technical Features

- The function block is a simple FB (SimpleFB) and uses an ST algorithm for the conversion.
- The conversion is performed directly and without any further delay.

State Overview

1. **Idle**: Waits for the event REQ.
2. **Processing**: Performs the conversion.
3. **Completed**: Outputs the event CNF and returns to the idle state.

Application Scenarios

- Conversion of strings to binary data for further processing in control systems.
- Use in protocol conversions where strings need to be converted to binary form.

⚖️ Comparison with Similar Function Blocks

- Compared to other conversion function blocks such as F_STRING_AS_LWORD, this function block specifically works with the WSTRING data type, which supports Unicode characters.

Conclusion

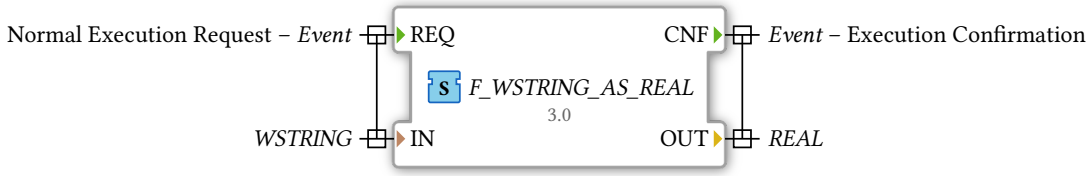
The F_WSTRING_AS_LWORD function block provides a simple and efficient way to convert WSTRING data to LWORD data. Its clear interface structure and direct functionality make it a useful tool in various applications.

F_WSTRING_AS_REAL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block F_WSTRING_AS_REAL is used to convert a WSTRING value into a REAL value. This block is particularly useful when strings representing numeric values need to be converted into numeric data types for further calculations or control logic.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this input is triggered, the WSTRING value contained in the data input IN is converted into a REAL value.

Event Outputs

- **CNF**: Signals the completion of the conversion. This output is activated as soon as the conversion is complete and the result value is available in the data output OUT.

Data Inputs

- **IN**: Enter the WSTRING value to be converted into a REAL value here.

Data Outputs

- **OUT**: Contains the converted REAL value after successful execution of the function block.

Adapters

This function block does not use any adapters.

Functionality

The function block performs the conversion as soon as the event REQ is triggered. The conversion is carried out using the internal function WSTRING_AS_REAL, which converts the WSTRING input IN into a REAL value and stores the result in the output OUT. The event CNF is then triggered to signal the successful completion of the operation.

Technical Features

- The function block supports WSTRING to REAL conversion, which is particularly useful in international applications because WSTRING supports Unicode characters.
- The conversion is performed in a single algorithm step, ensuring efficient execution.

State Overview

1. **Idle State**: The function block waits for the REQ event to be triggered.
2. **Conversion State**: The conversion is performed when REQ is triggered.
3. **Confirmation State**: After successful conversion, CNF is triggered, and the function block returns to the idle state.

Application Scenarios

- **Data Processing**: Conversion of user input or read strings into numeric values for calculations.

- **Interface Communication**: Conversion of strings received via networks or interfaces into numeric values for control logic.
- **International Applications**: Processing of Unicode-encoded strings containing numeric data.

⚖️ Comparison with Similar Function Blocks

- **F_STRING_AS_REAL**: Similar function block, but for the STRING data type instead of WSTRING. `F_WSTRING_AS_REAL` is preferable when Unicode support is required.
- **F_WSTRING_TO_***: Other conversion blocks for WSTRING, but they convert to other data types such as INT or BOOL.

Conclusion

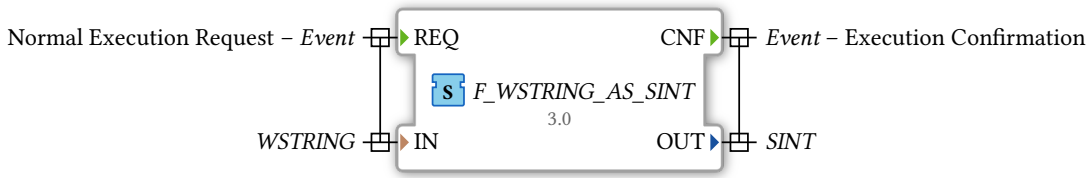
The `F_WSTRING_AS_REAL` function block is an efficient tool for converting Unicode strings into numeric REAL values. Its simple interface and reliable operation make it ideal for applications requiring international character sets or complex data processing.

F_WSTRING_AS_SINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WSTRING_AS_SINT` converts a WSTRING value to a SINT value (8-bit integer). This block is particularly useful in scenarios where strings need to be converted to numeric values.



Interface Structure

Event Inputs

- **REQ**: Triggering event for the conversion. Linked to the data input `IN`.

Event Outputs

- **CNF**: Confirms successful completion of the conversion. Linked to the data output `OUT`.

Data Inputs

- **IN** (WSTRING): The input string to be converted to a SINT value.

Data Outputs

- **OUT** (SINT): The resulting 8-bit integer value after conversion.

Adapters

No adapters available.

Functionality

The function block performs the conversion as soon as the `REQ` event occurs. The function `WSTRING_AS_SINT` is applied to the input value `IN`, and the result is output to `OUT`. Successful execution is indicated by the `CNF` event.

Technical Features

- The conversion is performed using the built-in function `WSTRING_AS_SINT`.
- The block is optimized for use in IEC 61131-3 compliant environments.

State Overview

The block has no internal states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- Conversion of user input to numeric values.
- Processing of strings from external sources (e.g., sensors or network communication).

⚖️ Comparison with Similar Blocks

- Compared to generic conversion blocks, `F_WSTRING_AS_SINT` specializes in converting WSTRING to SINT.
- Other blocks might offer additional features such as error handling or type checking.

Conclusion

The `F_WSTRING_AS_SINT` block offers an efficient and direct method for converting WSTRING values to SINT values. Its simplicity and specialization make it ideal for specific use cases.

F_WSTRING_AS_TIME

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WSTRING_AS_TIME` is used to convert a string in WSTRING format into a TIME value. This block is particularly useful when time values are in string format and need to be converted to TIME format for use in further calculations or control logic.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. When this event input is triggered, the WSTRING value contained in the data input `IN` is converted into a TIME value.

Event Outputs

- **CNF**: Signals the successful completion of the conversion. This event, along with the converted TIME value, is output at data output `OUT`.

Data Inputs

- **IN** (WSTRING): The string to be converted into a TIME value.

Data Outputs

- **OUT** (TIME): The converted TIME value.

Adapters

This function block has no adapters.

Functionality

When event input `REQ` is triggered, the WSTRING value contained in data input `IN` is converted into a TIME value using the function `WSTRING_AS_TIME`. The result is output at the data output `OUT`, and the event `CNF` signals the successful completion of the conversion.

Technical Features

- The function block has a simple structure and contains only one algorithm that performs the conversion.
- There are no additional states or complex logic, as the conversion is performed directly without intermediate steps.

State Overview

Since it is a simple function block, there are no explicit states. The conversion occurs immediately upon triggering the `REQ` event.

Application Scenarios

- **Data Conversion**: When time values are available as strings (e.g., from user input or an external data source) and need to be converted into the TIME format.
- **Interface Communication**: When communicating with systems that transmit time values as strings.

⚖️ Comparison with Similar Function Blocks

- **F_STRING_AS_TIME**: Similar function block, but it uses STRING instead of WSTRING. `F_WSTRING_AS_TIME` is suitable for Unicode strings.
- **F_TIME_AS_WSTRING**: Performs the reverse conversion (TIME to WSTRING).

Conclusion

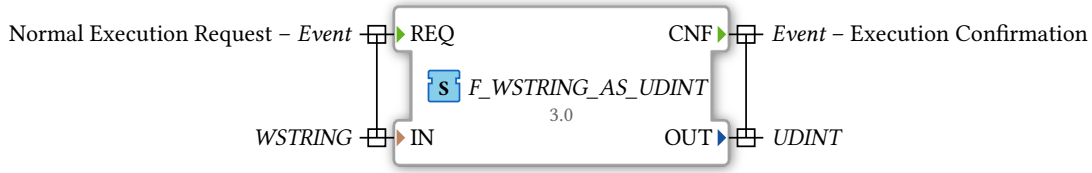
The function block `F_WSTRING_AS_TIME` is a simple and effective tool for converting WSTRING strings to TIME values. Its clear interface and direct functionality make it a reliable solution for relevant use cases.

F_WSTRING_AS_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WSTRING_AS_UDINT` is used to convert a WSTRING value to a UDINT value. This block is particularly useful in scenarios where strings need to be converted into numeric values, for example, when processing user input or interpreting data from external sources.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This input triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. This output signals the completion of the conversion.

Data Inputs

- **IN**: WSTRING. The input string to be converted to a UDINT value.

Data Outputs

- **OUT**: UDINT. The converted numeric value.

Adapters

This function block does not use any adapters.

Operation

The function block performs the conversion when the event `REQ` is received. The conversion is carried out using the function `WSTRING_AS_UDINT`, which converts the WSTRING input `IN` into a UDINT value and outputs the result to `OUT`. Upon completion of the conversion, the event `CNF` is triggered.

Technical Features

- The function block is simple and efficient because it directly utilizes the built-in conversion function.
- There is no additional error handling for invalid input; the conversion is performed according to the rules of IEC 61131-3.

State Overview

The function block has a simple state machine:

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Complete**: Sends the `CNF` event and returns to the idle state.

Application Scenarios

- **Data Processing**: Converts user input or external data into numeric values.
- **Logging**: Converts strings into numeric IDs for storage or further processing.
- **Interface Communication**: Prepares data for transmission via protocols that require numeric values.

⚖️ Comparison with Similar Function Blocks

- **F_STRING_AS_UDINT**: Similar function block, but for STRING instead of WSTRING.
- **F_UDINT_AS_WSTRING**: Performs the reverse conversion (UDINT to WSTRING).

Conclusion

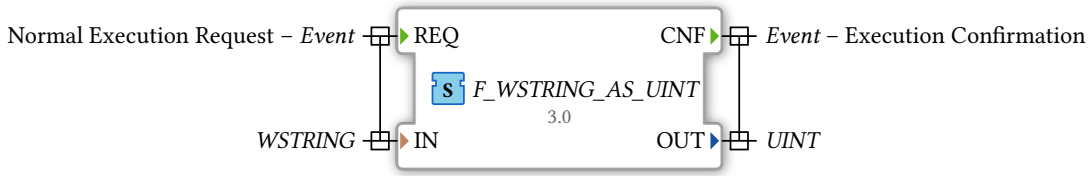
The `F_WSTRING_AS_UDINT` function block is a simple and effective tool for converting WSTRING values to UDINT. Its clear interface and direct functionality make it ideal for applications that require such conversions.

F_WSTRING_AS_UINT

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `F_WSTRING_AS_UINT` is used to convert a WSTRING value to a UINT value. This function is particularly useful when strings need to be converted into numeric values, for example, for further calculations or control tasks.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. This input triggers the conversion.

Event Outputs

- **CNF**: Execution confirmation. This output signals the completion of the conversion.

Data Inputs

- **IN**: WSTRING value to be converted to a UINT value.

Data Outputs

- **OUT**: UINT value resulting from the conversion of the WSTRING input.

Adapters

- No adapters available.

Functionality

The function block converts the WSTRING input `IN` to a UINT value `OUT` as soon as the event `REQ` is triggered. The conversion is performed using the built-in function `WSTRING_AS_UINT`. After successful conversion, the event `CNF` is triggered to signal the completion of the operation.

Technical Features

- The function block is a simple function block (SimpleFB) and performs the conversion in a single algorithm.
- The conversion is direct and occurs without additional state management.

State Overview

Since this is a simple function block, there are no complex state transitions. The block responds to the `REQ` event, performs the conversion, and signals completion with `CNF`.

Application Scenarios

- Converting user input to numeric values.
- Processing strings containing numeric data for control purposes.
- Integration into larger control systems that need to process both strings and numeric values.

⚖️ Comparison with Similar Blocks

- Compared to other conversion blocks such as `F_STRING_AS_INT` or `F_WSTRING_AS_INT`, this block specifically converts to a UINT value, which is suitable for non-negative numbers.
- Using WSTRING allows for the processing of Unicode strings, which is more flexible compared to simple STRING conversions.

Conclusion

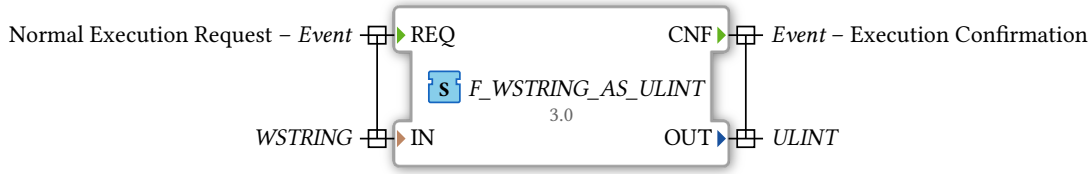
The `F_WSTRING_AS_UINT` function block provides a simple and efficient way to convert WSTRING values to UINT values. Its straightforward operation and clear interface make it a useful component in control applications that work with strings and numeric values.

F_WSTRING_AS_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WSTRING_AS_ULINT` is used to convert a WSTRING data type to a ULINT data type. This function block is particularly useful in scenarios where strings need to be converted into numeric values, for example, when processing user input or parsing data from external sources.



Interface Structure

Event Inputs

- **REQ**: Starts the conversion. This event input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. This event output is linked to the data output `OUT`.

Data Inputs

- **IN** (WSTRING): The input string to be converted to a ULINT value.

Data Outputs

- **OUT** (ULINT): The converted numeric value.

Adapters

No adapters are defined for this function block.

Functionality

The function block performs the conversion when the `REQ` event is received. The conversion is performed using the function `WSTRING_AS_ULINT`, which converts the string `IN` to an unsigned 64-bit integer (`ULINT`). After successful conversion, the `CNF` event is triggered, and the converted value is available at output `OUT`.

Technical Features

- The conversion is strictly typed, meaning the input must contain a valid numeric representation in the string to produce correct results.
- The function block does not support error handling for invalid input. It is the user's responsibility to ensure that the input is correctly formatted.

State Overview

1. **Idle**: The function block waits for the `REQ` event.
2. **Processing**: The conversion is performed upon receipt of `REQ`.
3. **Completed**: After the conversion, the `CNF` event is triggered, and the function block returns to its idle state.

Application Scenarios

- Converting user input to numeric values.
- Parsing data from external sources (e.g., files, network) into numeric formats.
- Use in control applications where strings need to be converted to control values.

⚖️ Comparison with Similar Function Blocks

- Compared to generic conversion function blocks, `F_WSTRING_AS_ULINT` specializes in converting WSTRING to ULINT and therefore offers an optimized and type-safe solution for this specific use case.
- Other function blocks, such as `F_STRING_AS_INT` or `F_WSTRING_AS_LINT`, offer similar functionality, but for different data types.

Conclusion

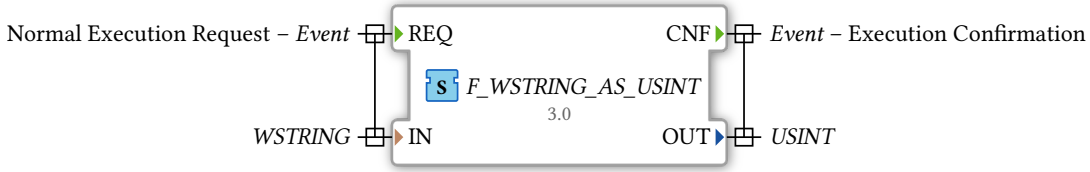
The `F_WSTRING_AS_ULINT` function block is an efficient tool for converting strings to unsigned 64-bit integer values. Its simplicity and specialization make it ideal for applications requiring fast and type-safe conversion. However, note that it lacks built-in error handling for invalid input.

F_WSTRING_AS_USINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_WSTRING_AS_USINT` converts a WSTRING value to a USINT value. It is part of the package `iec61131::conversion` and enables the easy conversion of wide strings to unsigned 8-bit integers.



Interface Structure

Event Inputs

- **REQ**: Starts the execution of the conversion. This input is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the completion of the conversion. The output is linked to the data output `OUT`.

Data Inputs

- **IN** (WSTRING): The input string to be converted to a USINT value.

Data Outputs

- **OUT** (USINT): The result of the conversion as an unsigned 8-bit integer.

Adapters

This function block does not use any adapters.

Functionality

The function block performs the conversion when the event `REQ` is triggered. The conversion is carried out using the function `WSTRING_AS_USINT`, which converts the WSTRING input value to a USINT value. After the conversion is complete, the event `CNF` is triggered, and the result value is available at output `OUT`.

Technical Features

- The conversion is performed directly without additional parameters.
- The function block is optimized for use in IEC 61499-compliant environments.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Done**: Sends the `CNF` event and provides the result.

Application Scenarios

- Conversion of character strings to numeric values for control applications.
- Use in systems that use character strings as input but require numeric processing.

⚖️ Comparison with Similar Function Blocks

- Unlike generic conversion blocks, `F_WSTRING_AS_USINT` specializes in converting WSTRING to USINT.
- Other blocks might offer additional parameters or error handling, but this block is designed for simplicity and efficiency.

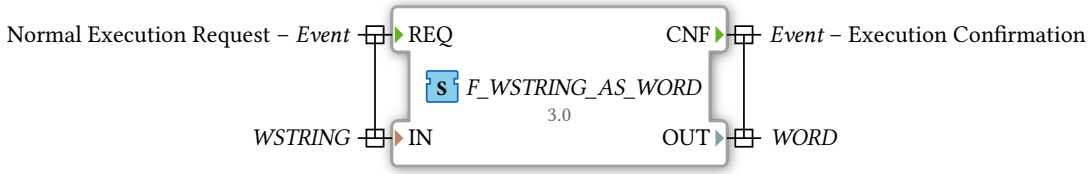
Conclusion

The `F_WSTRING_AS_USINT` function block provides a simple and efficient way to convert WSTRING values to USINT values. Its clear interface and direct functionality make it ideal for applications that require fast and straightforward conversions.

F_WSTRING_AS_WORD

• – ◦ • – ◦ • – ◦ •

The function block `F_WSTRING_AS_WORD` converts a WSTRING data type to a WORD data type. This functionality is particularly useful when strings need to be converted into numeric values, for example, for further processing in control algorithms.



- **REQ**: Starts the execution of the function block. This event input is linked to the data input `IN`.
- **CNF**: Signals the successful completion of the conversion. This event output is linked to the data output `OUT`.
- **IN**: Expects an input of type `WSTRING`, which is to be converted into a `WORD` value.
- **OUT**: Returns the result of the conversion as `WORD`.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters
This function block has no adapters.

Functionality
The function block performs the conversion as soon as the event `REQ` is received. The conversion is performed using the function `WSTRING_AS_WORD`, which converts the input `WSTRING` into a value `IN` and outputs the result via the output `OUT`. Successful execution is signaled by the event `CNF`.

- The conversion is performed directly without additional parameters.
- The function block is optimized for use in IEC 61131-compliant environments.

The function block has no internal states. Execution occurs immediately upon receiving the event `REQ`.

- Conversion of strings to numeric values for control commands.
- Integration into systems that require conversion between different data types.
- Compared to generic conversion function blocks, `F_WSTRING_AS_WORD` specializes in converting `WSTRING` to `WORD`.
- Other function blocks might offer additional parameters or error handling, but this function block is designed for simplicity and efficiency.

The `F_WSTRING_AS_WORD` function block provides a simple and efficient way to convert `WSTRING` data to `WORD` values. Its clear interface and direct operation make it ideal for applications that require fast and reliable conversions.

Technical Features

State Overview

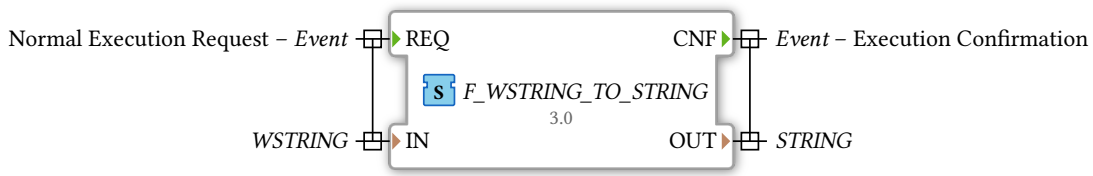
Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

F_WSTRING_TO_STRING
• – ◦ • – ◦ • – ◦ •

Introduction
The function block `F_WSTRING_TO_STRING` is used to convert a WSTRING data type to a STRING data type. This function block is particularly useful in scenarios where converting wide strings (Unicode) to simple strings (ASCII or similar) is required.



Interface Structure

Event Inputs
- **REQ**: Starts the conversion. When this event is triggered, the WSTRING input `IN` is processed.

Event Outputs
- **CNF**: Signals the completion of the conversion and outputs the result via the STRING output `OUT`.

Data Inputs
- **IN** (WSTRING): The input string in WSTRING format to be converted.

Data Outputs
- **OUT** (STRING): The converted string in STRING format.

Adapters
- No adapters are available.

Functionality
The function block performs the conversion as soon as the event `REQ` is triggered. The conversion is performed using the integrated function `WSTRING_TO_STRING`, which converts the WSTRING input `IN` into a STRING output `OUT`. After successful conversion, the event `CNF` is triggered.

Technical Features
- The function block does not support any additional parameters or configurations for the conversion.
- The conversion is performed directly without intermediate storage.

State Overview
1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Performs the conversion.
3. **Complete**: Triggers `CNF` and outputs the result.

Application Scenarios
- Conversion of Unicode text to ASCII text.
- Use in systems that do not support wide strings.
- Integration into existing systems that only work with STRING data types.

⚖️ Comparison with Similar Function Blocks
- Unlike generic conversion blocks, `F_WSTRING_TO_STRING` specializes in converting WSTRING to STRING.
- Similar function blocks might offer additional options such as character set conversion, but this block is limited to basic functionality.

Conclusion
The `F_WSTRING_TO_STRING` function block provides a simple and efficient way to convert WSTRING data to STRING data. Its clear interface and direct functionality make it ideal for projects that require fast and straightforward conversion.

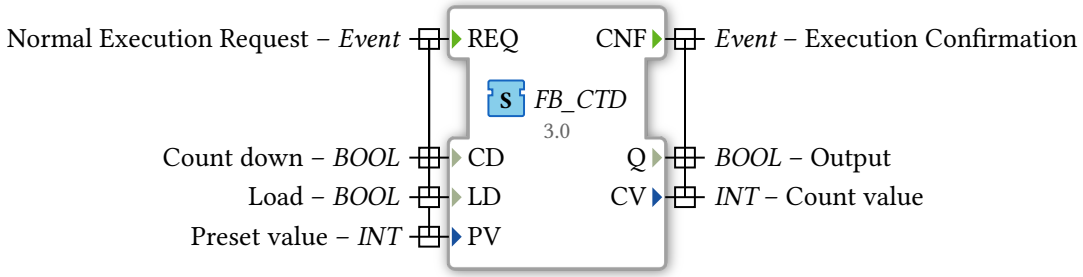
WSTRING Conversions
Here you will find functions for converting the WSTRING data type to other data types.

…

2.11.4.8 Counters
Here you will find function blocks for various counter types (forward, reverse, etc.).

FB_CTD
• – ◦ • – ◦ • – ◦ •

The FB_CTD (Down Counter) is a function block that acts as a down counter. It counts down the internal counter value on each trigger event and outputs a signal when the counter reaches or falls below a specific value.



- **REQ**: Normal execution request that starts the counting process.
- **CNF**: Acknowledgement of execution, triggered after the counting process.
- **CD (BOOL)**: Count Down - Signal to count down the counter.
- **LD (BOOL)**: Load - Signal to load the preset value into the counter.
- **PV (INT)**: Preset Value - Preset value loaded into the counter.
- **Q (BOOL)**: Output - Signal activated when the counter value is ≤ 0.
- **CV (INT)**: Count Value - Current counter value.

Adapters

No adapters available.

Adapter

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

The FB_CTD performs the following operations:

1. When the **LD** signal is active, the **PV** value is loaded into the counter (**CV**).
2. When the **CD** signal is active and the counter value (**CV**) is greater than −32768, the counter is decremented by 1.
3. The output signal **Q** is set to `TRUE` when the counter value (**CV**) is ≤ 0.

- The counter value (**CV**) is a 16-bit integer with a range of −32768 to 32767.
- The FB_CTD is a simple function block without an internal state machine.

The FB_CTD has no explicit states but reacts directly to the input signals.

- Controlling processes that require a specific number of steps.
- Monitoring time intervals or event counts.
- Use in automation systems where a counting counter is needed.
- **FB_CTU (Up Counter)**: Counts upwards instead of downwards.
- **FB_CTUD (Up-Down Counter)**: Combines up and down counting in one function block.

The FB_CTD is a simple and efficient function block for downcounting operations in automation applications. Its clear interface and simple functionality make it ideal for basic counting tasks.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

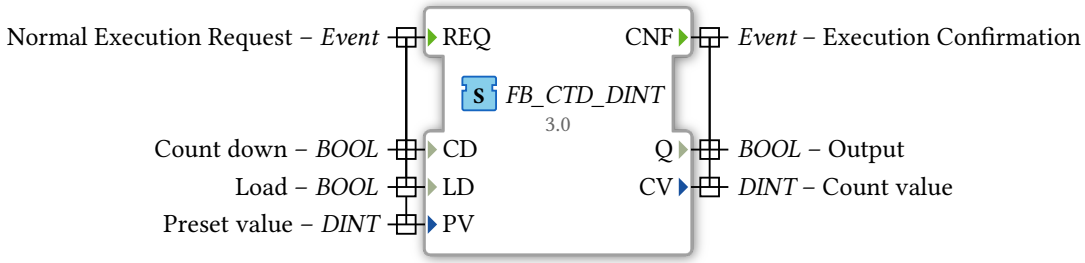
Conclusion

FB_CTD_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_CTD_DINT` is a down counter for 32-bit integers (DINT). It is used to decrease a value incrementally and output a signal when a specific threshold is reached.



Interface Structure

Event Inputs

- `REQ`: Normal execution request (trigger for the counting process)

Event Outputs

- `CNF`: Execution confirmation (triggered after completion of the counting process)

Data Inputs

- `CD` (BOOL): Count Down command
- `LD` (BOOL): Load command
- `PV` (DINT): Preset Value

Data Outputs

- `Q` (BOOL): Output signal (is TRUE if CV ≤ 0)
- `CV` (DINT): Current Count Value

Functionality

1. When the `REQ` event is triggered, the following algorithm is executed:

- If `LD` is TRUE, the counter is set to the value of `PV`
- If `CD` is TRUE and the current value (`CV`) is greater than −2147483648, `CV` is decremented by 1

1. The output signal `Q` is set to TRUE if `CV` is less than or equal to 0 is
2. Upon completion, the `CNF` event is triggered

Technical Features

- Uses 32-bit integers (DINT) with a value range of −2147483648 to 2147483647
- The counter stops when the minimum DINT value (-2147483648) is reached
- The block is included in the IEC 61131-3 standard library under `iec61131::counters`

State Overview

The function block has no explicit states but operates event-driven:

1. Waits for the `REQ` event

2. Performs the counting operation
3. Sends the `CNF` event

Application Scenarios

- Down counting in production processes
- Time controls with countdown function
- Monitoring of remaining quantities or steps

⚖️ Comparison with similar blocks

- Unlike up counters (CTU), this block decrements the counter value
- Compared to 16-bit counters (INT), this block offers a larger value range
- Similar to timers, but for general counting purposes without a time reference

Conclusion

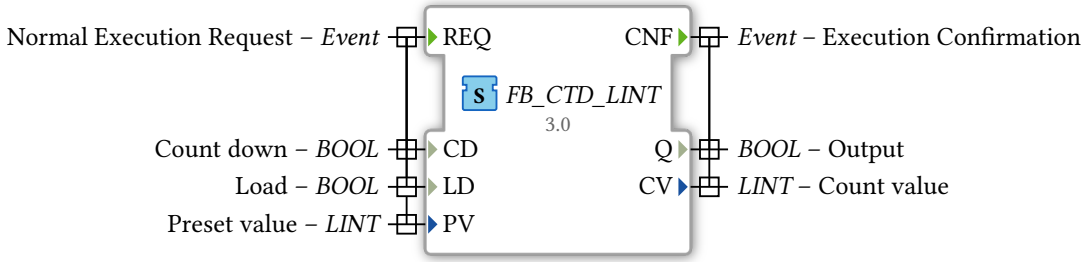
The `FB_CTD_DINT` is a robust and easy-to-use down counter for 32-bit values. Its clear interface and deterministic operation make it ideal for control tasks where reliable down counting is required.

FB_CTD_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_CTD_LINT` is a countdown counter for 64-bit integers (LINT). It allows counting down an internal counter value and provides functions for loading a predefined value. The block is particularly suitable for applications that need to work with large ranges of numbers.



Interface Structure

Event Inputs

- `REQ`: Starts the normal execution of the counting process. Linked to the data `CD`, `PV`, and `LD`.

Event Outputs

- `CNF`: Confirms execution and returns the current values of `Q` and `CV`.

Data Inputs

- `CD` (BOOL): Signals whether the counter should be decremented.
- `LD` (BOOL): Loads the predefined value (`PV`) into the counter.
- `PV` (LINT): The predefined value that is loaded when `LD` is active.

Data Outputs

- `Q` (BOOL): Signals whether the counter value (`CV`) is less than or equal to zero.
- `CV` (LINT): The current counter value.

Adapters

No adapters available.

Functionality

The function block operates according to the following logic:

1. When `LD` is active, the predefined value (`PV`) is loaded into the counter (`CV`).

2. If `CD` is active and the counter value (`CV`) has not reached the minimum 64-bit value, `CV` is decremented by 1.
3. The output `Q` is set to `TRUE` if `CV` is less than or equal to zero.

Technical Features

- Uses the data type `LINT` (64-bit integer), which allows for a very large number range.
- The minimum value for `LINT` is taken into account to prevent overflows.

Status Overview

- **Initialization**: No specific initialization; `CV` starts with an undefined value.
- **Loading**: When `LD` is active, `CV` is overwritten with `PV`.
- **Counting**: When `CD` is active, `CV` is decremented unless the minimum value has been reached.
- **Output**: `Q` is set based on the current value of `CV`.

Application Scenarios

- Controlling processes with large counting ranges, e.g., in production automation.
- Monitoring of time intervals or event counts in industrial control systems.

⚖️ Comparison with similar components

- Similar to `FB_CTD_INT`, but with an extended number range (`LINT` instead of `INT`).
- Offers greater flexibility compared to simpler counters due to its large value range and loading function.

Conclusion

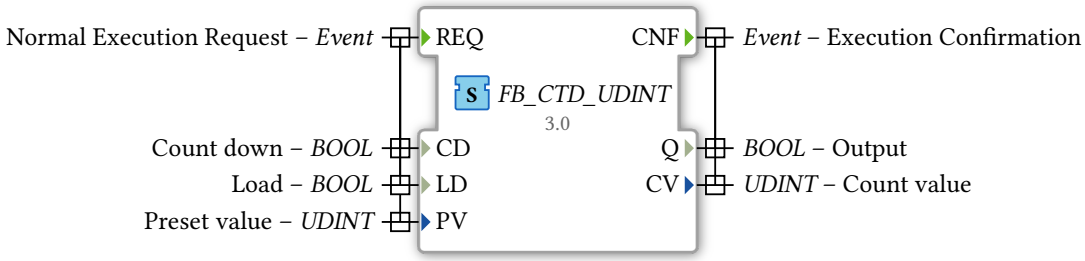
The `FB_CTD_LINT` is a powerful down counter for applications requiring large number ranges. Its ease of use and robust implementation make it a reliable choice for complex control tasks.

FB_CTD_UDINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_CTD_UDINT` is a down counter for unsigned 32-bit integers (UDINT). It is used to decrement a counter value on each event until it reaches zero.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Used to trigger the counting process or the loading of a new value.

Event Outputs

- **CNF**: Execution confirmation. Triggered after the request has been processed and returns the current counter value and the output state.

Data Inputs

- **CD** (BOOL): Counts down when `TRUE` is active and the counter value is greater than zero.
- **LD** (BOOL): Loads the specified value (`PV`) into the counter when `TRUE` is active.
- **PV** (UDINT): Specified value loaded into the counter when `LD` is active.

Data Outputs

- **Q** (BOOL): Outputs `TRUE` when the counter value is less than or equal to zero.

- **CV** (UDINT): Current counter value.

Adapters

No adapters available.

Functionality

The function block processes the input events and data as follows:

1. If `LD` is enabled (`TRUE`), the specified value (`PV`) is loaded into the counter (`CV`).
2. If `CD` is enabled (`TRUE`) and the counter value (`CV`) is greater than zero, the counter value is decremented by one.
3. The output `Q` is set to `TRUE` if the counter value (`CV`) is less than or equal to zero.
4. After processing, the output event `CNF` is triggered to confirm the new state.

Technical Features

- The counter uses unsigned 32-bit integers (UDINT).
- The counter value is only decremented if `CD` is enabled and the current value is greater than zero.
- The output `Q` signals when the counter value reaches zero or below.

State Overview

- **Initialization**: The counter value (`CV`) is undefined until a `LD` signal is received.
- **Loading**: When `LD` is enabled, `PV` is loaded into `CV`.
- **Countdown**: When `CD` and `CV > 0` are enabled, `CV` is decremented by one.
- **Initial State**: `Q` becomes `TRUE` when `CV <= 0` is enabled.

Application Scenarios

- Controlling processes that require a specific number of steps or events.
- Time-controlled processes that require a countdown.
- Monitoring cycles or repetitions in automated systems.

⚖️ Comparison with Similar Components

- Unlike an up counter (CTU), this component decrements the counter value.
- Similar to other counter components, but specialized for unsigned 32-bit integers.

Conclusion

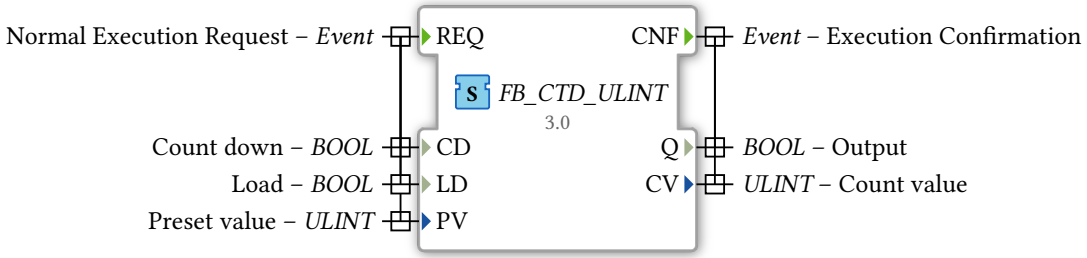
The `FB_CTD_UDINT` is an efficient and simple down counter for applications requiring a counting function with unsigned 32-bit integers. Its clear interface and simple operation make it a useful component in automation technology.

FB_CTD_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The FB_CTD_ULINT is a down counter for the ULINT (Unsigned Long Integer) data type. It is used to count down from a predefined starting value and signals when the counter reaches or falls below 0.



Interface Structure

Event Inputs

- **REQ**: Normal Execution Request. Used to update the counter. Linked to the data inputs CD, PV, and LD.

Event Outputs

- **CNF**: Execution Confirmation. Triggered after the counter has been updated. Linked to the data outputs Q and CV.

Data Inputs

- **CD (BOOL)**: Signals that the counter should be decremented by 1 (Count Down).
- **LD (BOOL)**: Signals that the counter should be loaded with the value of PV (Load).
- **PV (ULINT)**: The preset value with which the counter is loaded when LD is enabled.

Data Outputs

- **Q (BOOL)**: Signals whether the counter reading is less than or equal to 0.
- **CV (ULINT)**: The current counter reading (Count Value).

Adapters

No adapters are defined for this function block.

Functionality

The function block operates according to the following rules:

1. When LD (Load) is enabled, the counter reading (CV) is set to the preset value (PV).
2. If CD (Count Down) is enabled and the counter value (CV) is greater than 0, the counter value is decremented by 1.
3. Output Q is set to TRUE if the counter value (CV) is less than or equal to 0.

Technical Features

- The counter uses the ULINT data type, which allows for very large positive numbers (0 to 18,446,744,073,709,551,615).
- The counter is only updated when the REQ event is triggered.

State Overview

- **Initialization**: The counter value (CV) is undefined until the first REQ event.
- **Loading (LD)**: CV is set to PV.
- **Count (CD)**: CV is decremented by 1 as long as CV > 0.
- **Output Q**: TRUE if CV ≤ 0.

Application Scenarios

- Controlling production processes with downward counting (e.g., remaining parts in a warehouse).
- Time control with large time intervals.
- Monitoring of events with a limited number of occurrences.

⚖️ Comparison with Similar Function Blocks

- Compared to other counters (e.g., FB_CTD_INT), this function block offers a larger number range through ULINT.
- Similar to FB_CTD, but specifically optimized for ULINT values.

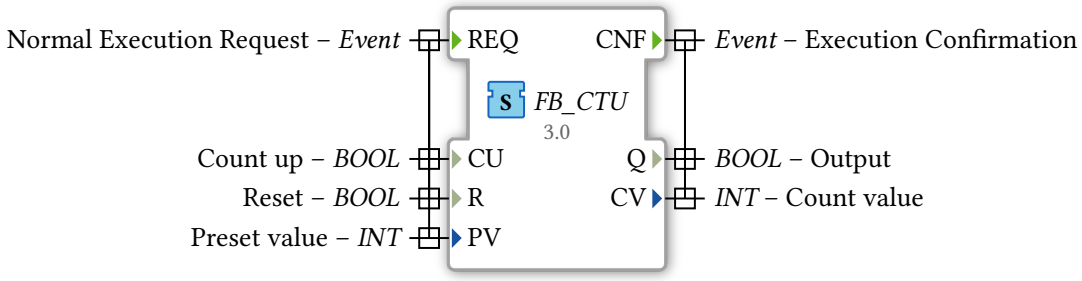
Conclusion

The FB_CTD_ULINT is a powerful downward counter for applications requiring very large counting ranges. Its simple and clear interface makes it easy to integrate into various control systems.

FB_CTU

• − ◦ • − ◦ • − ◦ •

The FB_CTU (Up Counter) is a function block that acts as an up counter. It increments the number of events and can be reset. The counter value is compared to a predefined value (PV), and an output signal (Q) is triggered as soon as the counter value reaches or exceeds the predefined value.



- **REQ**: Normal Execution Request. Used to trigger the counting process. Linked to the data inputs `CU` and `R`.
- **CNF**: Execution Acknowledgement. Triggered after the counting process is complete. Linked to the data outputs `Q` and `CV`.
- **CU** (BOOL): Increments when `TRUE` is reached. Only evaluated when `R` is reached or `FALSE` is reached.
- **R** (BOOL): Resets the counter value (`CV`) to `0` when `TRUE` is reached.
- **PV** (INT): Predefined value against which the counter value (`CV`) is compared.
- **Q** (BOOL): Outputs `TRUE` when the counter value (`CV`) reaches or exceeds the predefined value (`PV`).
- **CV** (INT): Current counter value.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters available.

Functionality

The FB_CTU performs the following steps when the event `REQ` occurs:

1. If `R` is `TRUE`, the counter value (`CV`) is reset to `0`. ...
2. The output `Q` is set to `TRUE` if `CV` is greater than or equal to `PV`.

- The counter value (`CV`) is limited to a maximum value of 32767.
- The FB_CTU is a simple function block without internal state management between calls.

The FB_CTU has no internal states. The logic is re-executed with each `REQ` event.

- Counting events, e.g., production steps in a manufacturing line.
- Monitoring processes where a specific number of steps must be completed.
- Controlling sequences that should trigger an action after a certain number of repetitions.
- **FB_CTD**: A down counter that, unlike FB_CTU, counts down.
- **FB_CTUD**: A combined up and down counter that can count both up and down.

The FB_CTU is a simple and effective function block for up counting operations. It is particularly suitable for applications where events need to be counted and an action triggered when a specific value is reached. Its clear interface structure and simple functionality make it a useful component in control engineering.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

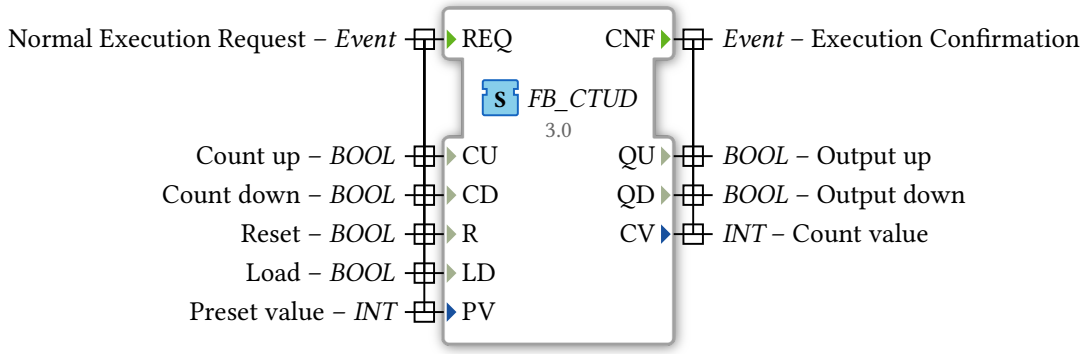
Conclusion

FB_CTUD

• – ◦ • – ◦ • – ◦ •

Introduction

The FB_CTUD (Up Down Counter) is a function block that acts as a counter with up and down counting capabilities. It allows events to be counted in both directions and offers additional functions such as reset and loading a predefined value.



Interface Structure

Event Inputs

- **REQ**: Normal Execution Request. This event input triggers the execution of the counting process and is linked to the data inputs CU, CD, R, LD, and PV.

Event Outputs

- **CNF**: Execution Confirmation. This event is output after the counting process is complete and is linked to the data outputs QU, QD, and CV.

Data Inputs

- **CU** (BOOL): Counts up.
- **CD** (BOOL): Counts down.
- **R** (BOOL): Resets the counter value to 0.
- **LD** (BOOL): Loads the predefined value PV into the counter.
- **PV** (INT): Predefined value that is loaded into the counter when LD is enabled.

Data Outputs

- **QU** (BOOL): Signals whether the counter value is greater than or equal to the predefined value PV.
- **QD** (BOOL): Signals whether the counter value is less than or equal to 0.
- **CV** (INT): Current counter value.

Adapters

The FB_CTUD does not have any adapters.

Functionality

The function block FB_CTUD performs the following operations:

1. **Reset (R)**: When R is enabled, the counter value (CV) is set to 0.
2. **Load (LD)**: When LD is enabled, the predefined value (PV) is loaded into the counter (CV).

3. **Count (CU/CD)**:

- When CU is enabled and the counter value (CV) is less than 32767, CV is incremented by 1.
- When CD is enabled and the counter value (CV) is greater than −32768, CV is decremented by 1.
- CU and CD cannot be enabled simultaneously.

1. **Outputs (QU/QD)**:

- QU is enabled when CV is greater than or equal to PV.
- QD is enabled when CV is less than or equal to 0.

Technical Features

- The counter value (CV) is of type INT and has a range of −32768 to 32767.
- The FB_CTUD is a simple function block (SimpleFB) and uses a Structured Text (ST) algorithm to implement its logic.

State Overview

The FB_CTUD has no explicit states, but the counter value (CV) can be considered a state, which is changed by the inputs CU, CD, R, and LD.

Application Scenarios

- **Production Lines**: Counting units produced (count up) or remaining units (count down).
- **Warehouse Management**: Monitoring inventory levels.
- **Control Engineering**: Time-controlled processes or event counting.

⚖️ Comparison with Similar Function Blocks

- **FB_CTU**: Simple up counter without a down counter function.
- **FB_CTD**: Simple down counter without up counting functionality.
- **FB_CTUD**: Combines both functions in one block and also offers reset and load functions.

Conclusion

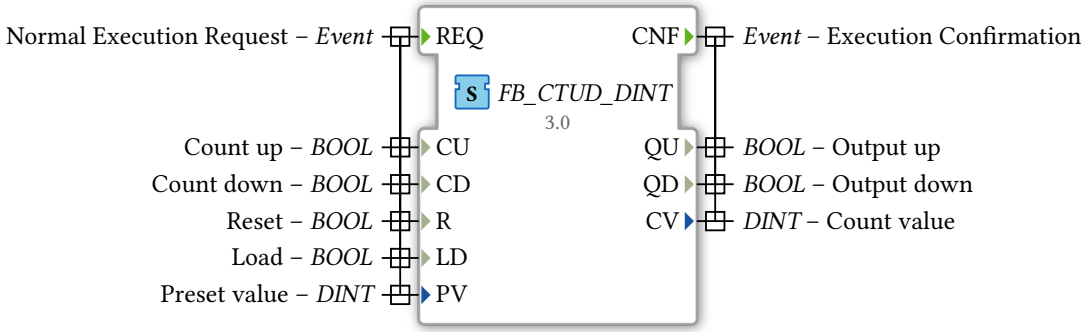
The FB_CTUD is a versatile function block that combines both up and down counting functions in a single block. With its additional functions, such as reset and loading a predefined value, it is suitable for a wide range of control engineering applications. Its ease of use and clear interface structure make it a practical solution for counting tasks.

FB_CTUD_DINT

• − ◦ • − ◦ • − ◦ •

Introduction

The FB_CTUD_DINT is a function block that acts as an up/down counter with a DINT counter value. It allows counting in both directions as well as resetting and loading a predefined value. The counter is particularly suitable for control applications where a flexible counting function is required.



Interface Structure

Event Inputs

- **REQ**: Normal execution request (triggers the counting operation)

Event Outputs

- **CNF**: Execution confirmation (triggered after completion of the counting operation)

Data Inputs

- **CU** (BOOL): Counts up
- **CD** (BOOL): Counts down
- **R** (BOOL): Resets the counter
- **LD** (BOOL): Loads the preset value
- **PV** (DINT): Preset value

Data Outputs

- **QU** (BOOL): Signals whether the count has reached or exceeded the preset value
- **QD** (BOOL): Signals whether the Output down (CV)
- **CV** (DINT): Current count value

Adapter

No adapters available.

Functionality

The function block performs the following operations when the REQ event occurs:

1. **Reset (R)**: Sets the count value (CV) to 0 if R is active.
2. **Load (LD)**: Loads the preset value (PV) into the count value (CV) if LD is active.
3. **Count (CU/CD)**:

- If CU is active and the count value has not reached the maximum value (2147483647), the count value is incremented by 1.
- If CD is active and the count value has not reached the minimum value (-2147483648), the count value is decremented by 1.

1. **Setting Outputs**:

- QU is set to TRUE when the counter value reaches or exceeds the preset value.
- QD is set to TRUE when the counter value reaches or falls below 0.

Technical Features

- The counter value (CV) is of type DINT (32-bit signed integer), which allows for a large counting range.
- The function block prevents the counter value from overflowing by checking the limit values.
- Simultaneously activating CU and CD does not change the counter value.

State Overview

The function block does not have an internal state in the sense of a state machine. The logic is re-executed with each REQ event.

Application Scenarios

- Production counting (e.g., piece counting on a conveyor belt)
- Position detection (e.g., in linear axes)
- Event counting with the option to reverse direction

⚖️ Comparison with similar function blocks

- Compared to simple counters (e.g., CTU or CTD), this function block offers both counting directions in a single block.
- Compared to counters with a smaller data type (e.g., INT), the DINT type offers a larger counting range.

Conclusion

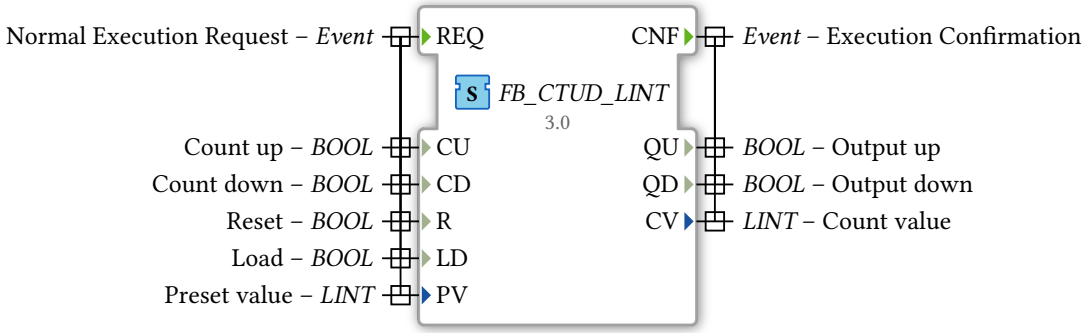
The FB_CTUD_DINT is a versatile counter function block for applications requiring bidirectional counting functions with a large counting range. Its integrated reset and load functions, as well as the status outputs QU and QD, make it particularly suitable for complex control tasks.

FB_CTUD_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The FB_CTUD_LINT is a function block for an incrementing and decrementing counter with 64-bit integer values (LINT). It provides functions for counting up and down, resetting the counter, and loading a predefined value. The counter is particularly suitable for applications requiring large number ranges.



Interface Structure

Event Inputs

- **REQ**: Normal Execution Request. Used to start the counting process. Linked to the data inputs CU, CD, R, LD, and PV.

Event Outputs

- **CNF**: Execution Confirmation. Outputted after completion of the counting operation. Linked to the data outputs QU, QD, and CV.

Data Inputs

- **CU (BOOL)**: Count Up signal.
- **CD (BOOL)**: Count Down signal.
- **R (BOOL)**: Reset counter signal.
- **LD (BOOL)**: Load predefined value signal.
- **PV (LINT)**: Predefined value loaded when LD is active.

Data Outputs

- **QU (BOOL)**: Signal indicating whether the counter value is greater than or equal to the predefined value.
- **QD (BOOL)**: Signal indicating whether the counter value is less than or equal to 0.
- **CV (LINT)**: Current counter value.

Adapter

No adapters available.

Functionality

The function block performs the following operations when the REQ event occurs:

1. **Reset (R)**: Resets the counter value (CV) to 0.
2. **Load (LD)**: Loads the predefined value (PV) into the counter value (CV).
3. **Count Up (CU)**: Increments the counter value by 1, provided the maximum value (9223372036854775807) is not exceeded.
4. **Count Down (CD)**: Decrements the counter value by 1, provided the minimum value (-9223372036854775808) is not undercut.
5. **Outputs QU and QD**: Updated based on the current counter value.

Technical Features

- Uses 64-bit integers (LINT) for the counter value and the predefined value.
- Supports both counting up and down with overflow and underflow protection.
- Simultaneous activation of CU and CD is not possible (prevents conflicts).

State Overview

The function block has no internal state other than the current counter value (CV). The logic is re-executed with each REQ event.

Application Scenarios

- Industrial controllers with large counting ranges.
- Monitoring of production processes.
- Time measurement or event counting over long periods.

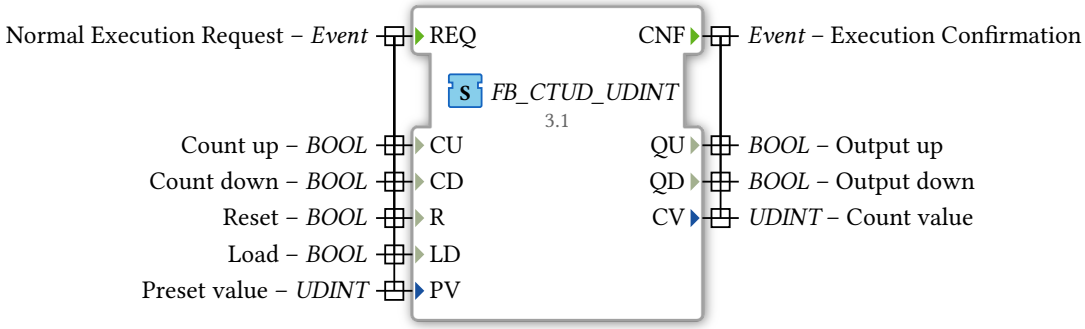
⚖️ Comparison with Similar Function Blocks

- Compared to simple counters, FB_CTUD_LINT offers a larger number range (64-bit).
- Similar to FB_CTUD, but with a LINT data type for more advanced applications.

Conclusion

The FB_CTUD_LINT is a high-performance counter for applications requiring large number ranges. Its flexibility and robustness make it ideal for industrial controls and complex counting tasks.

FB_CTUD_UDINT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_CTUD_UDINT` is a combined up/down counter for data type `UDINT` (32-bit unsigned integer). It counts up on `CU` and down on `CD`, can be loaded with a preset value `PV` via `LD`, and reset to `0` via `R`.

Interface Structure

Event Inputs

- **REQ**: Triggers execution, linked with `CU`, `CD`, `R`, `LD`, `PV`.

Event Outputs

- **CNF**: Confirms execution, linked with `QU`, `QD`, `CV`.

Data Inputs

- **CU** (BOOL): Count up when `TRUE`.
- **CD** (BOOL): Count down when `TRUE`.
- **R** (BOOL): Reset — sets `CV` to `0`, takes priority over `LD`/`CU`/`CD`.
- **LD** (BOOL): Load — loads `PV` into `CV`, provided `R` is not active.
- **PV** (UDINT): Preset value compared against `CV`.

Data Outputs

- **QU** (BOOL): `TRUE` when `CV >= PV`.
- **QD** (BOOL): `TRUE` when `CV <= 0`.
- **CV** (UDINT): Current count value.

Functionality

On every `REQ` event, the block evaluates the inputs in the following priority: if `R` is active, `CV := 0`. Otherwise, if `LD` is active, `CV := PV`. If neither `R` nor `LD` is active and `CU`/`CD` are not active at the same time, `CV` is incremented by 1 on `CU` (provided `CV < 4294967295`) or decremented by 1 on `CD` (provided `CV > 0`). Finally, `QU := CV >= PV` and `QD := CV <= 0` are computed.

Technical Features

- **Prioritization**: `R` (reset) takes priority over `LD` (load), which in turn takes priority over the actual counting operation.
- **Simultaneous `CU+CD`**: If both are `TRUE` at the same time, no count change occurs (`NOT(CU AND CD)` condition in the algorithm's ST).
- **Range limiting**: The count value is clamped at `4294967295` (the maximum value of `UDINT`) on the upper end and at `0` on the lower end.

State Overview

1. **Reset**: `R = TRUE` ⟶ `CV := 0`.
2. **Load**: `R = FALSE`, `LD = TRUE` ⟶ `CV := PV`.
3. **Counting**: `R = FALSE`, `LD = FALSE` ⟶ `CV` is adjusted based on `CU`/`CD`.
4. **Output**: `QU`/`QD` are recomputed from the current `CV`, `CNF` is triggered.

Application Scenarios

- **Bidirectional counting tasks with a large value range**, e.g. piece counts or position values that need both incrementing and decrementing and can exceed the range of `UINT`/`DINT`.
- **Fill-level or buffer counting**, where `QU`/`QD` act as upper/lower-bound signals for downstream logic.

Comparison with similar function blocks

- **`FB_CTUD`**: the same functionality for data type `INT`.
- **`FB_CTUD_DINT`/`FB_CTUD_LINT`/`FB_CTUD_ULINT`**: the same functionality for other integer types.
- **`FB_CTU_UDINT`/`FB_CTD_UDINT`**: the pure up- or down-counter variants for `UDINT`, without the combined `LD` function.

Conclusion

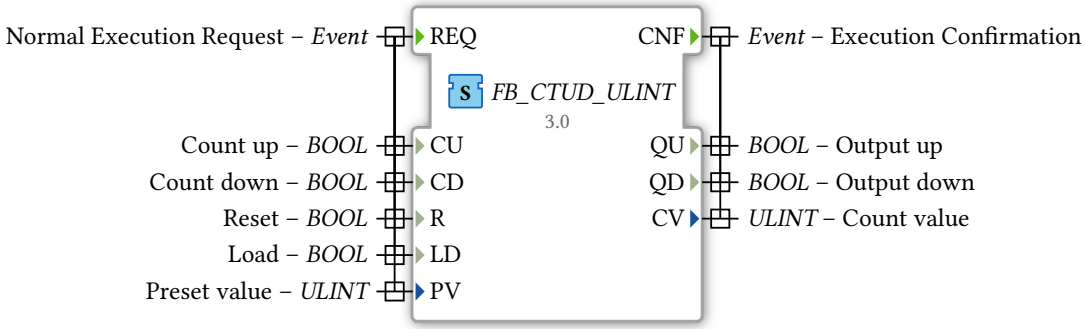
`FB_CTUD_UDINT` provides a complete up/down counter with load and reset functionality for the `UDINT` value range, closing the gap between the already available `FB_CTUD_*` variants for other data types.

FB_CTUD_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_CTUD_ULINT` is a counter with up and down counting functionality for the data type `ULINT` (unsigned 64-bit integer). It offers additional functions such as reset, loading a default value, and status outputs for the counting direction.



Interface Structure

Event Inputs

- `REQ`: Starts the normal execution of the counting process. Linked to the data inputs `CU`, `CD`, `R`, `LD`, and `PV`.

Event Outputs

- `CNF`: Confirms execution and provides the current counter values via the data outputs `QU`, `QD`, and `CV`.

Data Inputs

- `CU` (BOOL): Count Up Signal
- `CD` (BOOL): Count Down Signal
- `R` (BOOL): Reset Signal (sets the counter to 0)
- `LD` (BOOL): Load Signal (loads the preset value `PV` into the counter)
- `PV` (ULINT): Preset Value (default value for the counter)

Data Outputs

- `QU` (BOOL): Signals whether the counter value is greater than or equal to the preset value
- `QD` (BOOL): Signals whether the counter value is less than or equal to 0
- `CV` (ULINT): Current counter value

Functionality

The counter performs the following operations on the `REQ` event:

1. When `R` (Reset) is active, the counter is reset to 0.
2. When `LD` (Load) is active, the default value `PV` is loaded into the counter.
3. If neither Reset nor Load is active:

- When `CU` is active (and not simultaneously `CD`), the counter is incremented (up to the maximum value). 18446744073709551615)
- When `CD` is active (and not simultaneously `CU`), the counter counts down (to the minimum value of 0).

1. The outputs `QU` and `QD` are updated according to the current counter value.

Technical Features

- Uses the 64-bit unsigned integer data type (ULINT)
- Maximum counter value: 18446744073709551615
- Special handling for the simultaneous activation of `CU` and `CD` (no counting operation)
- Status outputs for exceeding/falling below the limit

Application Scenarios

- Industrial counting tasks with large Number ranges
- Position detection in large systems
- Material flow monitoring with high production volumes
- General control tasks with up/down counting function

⚖️ Comparison with similar function blocks

- Compared to simple counters, this function block offers both counting directions
- Compared to smaller data types (e.g., INT), ULINT enables extremely large counting ranges
- Integrated status outputs (`QU`, `QD`) simplify monitoring

Conclusion

The `FB_CTUD_ULINT` is a powerful counter function block for applications requiring large number ranges and flexible counting functions. Its combined up/down counting function and additional control and status options make it suitable for complex counting tasks in industrial control systems.

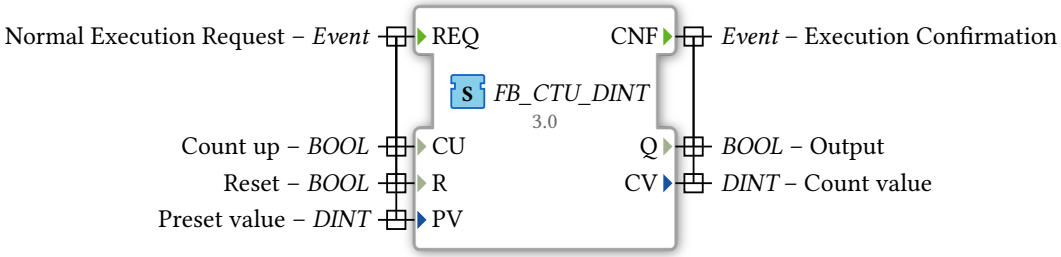
FB_CTU_DINT

• – ◦ • – ◦ • – ◦ •

Introduction

The FB_CTU_DINT is an up counter for the DINT data type (32-bit count). It increments with each count pulse and can be reset to a predefined value. This function block is particularly suitable for applications requiring large count ranges.

•



Interface Structure

Event Inputs

- **REQ**: Normal execution request (triggers the counting process)

Event Outputs

- **CNF**: Execution confirmation (triggered after completion of the counting process)

Data Inputs

- **CU** (BOOL): Count input (increments the counter when TRUE)
- **R** (BOOL): Reset input (sets the counter value to 0 when TRUE)
- **PV** (DINT): Preset value (defines the comparison value for the Q output)

Data Outputs

- **Q** (BOOL): Status output (becomes TRUE if CV ≥ PV)
- **CV** (DINT): Current counter value (Count Value)

Adapters

No adapters available.

Functionality

The FB_CTU_DINT operates according to the following rules:

1. When reset is active (R=TRUE), the counter value (CV) is reset to 0.
2. When the counter input is active (CU=TRUE) and reset is inactive, the counter value is incremented by 1 (up to the maximum value of 2147483647).
3. The output Q is set to TRUE as soon as the counter value (CV) reaches or exceeds the preset value (PV).

Technical Features

- Uses 32-bit integers (DINT) for counter values
- Maximum counter value: 2147483647
- Implemented as a simple function block (SimpleFB)
- Supports the IEC 61131-3 standard

State Overview

The function block does not have an internal state machine. The logic is executed on each REQ event.

Application Scenarios

- Production counting in manufacturing plants
- Event counting in process control systems
- Cycle time monitoring
- General counting tasks in automation technology

⚖️ Comparison with similar function blocks

- Compared to FB_CTU_INT, FB_CTU_DINT offers a larger counting range (32-bit vs. 16-bit)
- Compared to FB_CTU_LINT, FB_CTU_DINT has a smaller value range but lower memory requirements
- Similar to IEC 61131-3 CTU function blocks, but implemented in the 4diac IDE

Conclusion

The FB_CTU_DINT is a robust and easy-to-use increment counter for applications with medium to large counting ranges. Its clear interface and simple functionality make it particularly suitable for standard counting tasks in automation technology.

🌐 Related topic subpages on ms-muc-docs.de

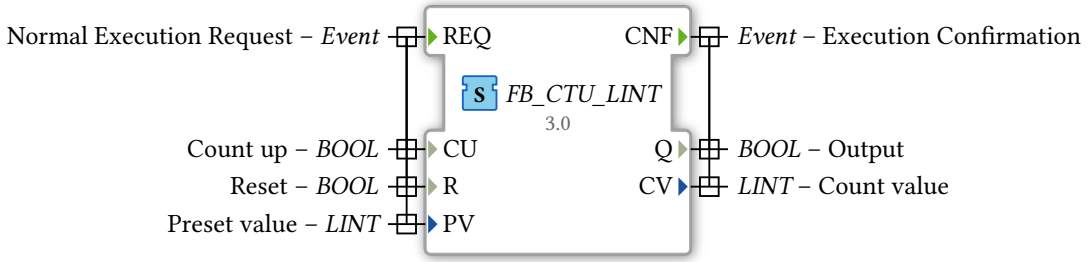
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

FB_CTU_LINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_CTU_LINT` is a counter-up function for large integers. It increments with each incoming counting event and can be reset to a predefined value. The block uses the data type `LINT` (64-bit integer) for the counter value and the setpoint.



Interface Structure

Event Inputs

- **REQ**: Normal execution request, triggers the counting operation. It is associated with the data `CU` and `R`.

Event Outputs

- **CNF**: Execution confirmation, output after the counting operation. Linked to the data `Q` and `CV`.

Data Inputs

- **CU** (BOOL): Count Up command. If TRUE, the counter is incremented.
- **R** (BOOL): Reset. If TRUE, the counter is reset to 0.
- **PV** (LINT): Preset Value. The setpoint at which output `Q` is set to TRUE.

Data Outputs

- **Q** (BOOL): Output. Becomes TRUE when the counter value (`CV`) reaches or exceeds the setpoint value (`PV`).
- **CV** (LINT): Current Value. Current counter reading.

Adapters

This function block does not use adapters.

Functionality

The function block performs the following operations on each `REQ` event:

1. If `R` is TRUE, the counter (`CV`) is reset to 0.
2. If `CU` is TRUE and the counter has not yet reached its maximum value (9223372036854775807), the counter is incremented by 1.
3. The output `Q` is set to TRUE when the counter value (`CV`) reaches or exceeds the setpoint (`PV`).

Technical Features

- Uses 64-bit integers (`LINT`) for counter value and setpoint
- Maximum counter value: 9223372036854775807 (2^63-1)
- Implemented in Structured Text (ST)

State Overview

The function block does not have an internal state machine. The logic is re-executed with each `REQ` event.

Application Scenarios

- Counting events in large number ranges
- Monitoring production quantities
- Controlling processes with high count values

⚖️ Comparison with similar blocks

- Compared to `FB_CTU` (standard counter), this block supports larger number ranges (`LINT` instead of `INT`)
- Similar to `FB_CTUD`, but without a down counting function

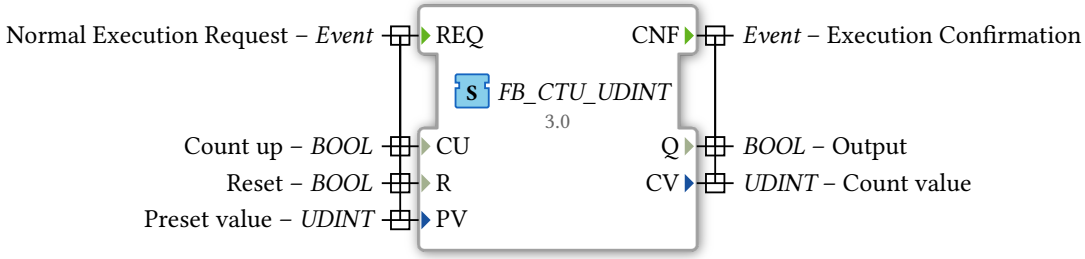
Conclusion

The `FB_CTU_LINT` is a powerful up counting block for applications requiring large counting ranges. Its simple interface and robust implementation make it a useful building block for industrial control applications.

FB_CTU_UDINT

• – ◦ • – ◦ • – ◦ •

The function block `FB_CTU_UDINT` is an up counter for the data type `UDINT` (unsigned double integer). It increments with each event `REQ` as long as the input `CU` is active, and can be reset via the input `R`. The counter compares the current counter value with a predefined setpoint (`PV`) and outputs a signal via the output `Q` when this setpoint is reached or exceeded.



- **REQ**: Starts the execution of the function block. It is linked to the data inputs `CU` and `R`.
- **CNF**: Confirms the execution of the function block. It is linked to the data outputs `Q` and `CV`.
- **CU** (`BOOL`): Increment signal. The counter value is incremented on `TRUE`.
- **PV** (`UDINT`): Preset value against which the current counter reading is compared.
- **Q** (`BOOL`): Signal generated when the counter reading (`CV`) reaches or exceeds the preset value (`PV`).
- **CV** (`UDINT`): Current counter reading.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

The function block performs the following steps on each `REQ` event:

1. If `R` is active (`TRUE`), the counter value (`CV`) is reset to `0`.
2. If `CU` is active (`TRUE`) and the counter value (`CV`) is less than the maximum value (`4294967295`), the counter value is incremented by `1`.
3. The output `Q` is set to `TRUE` when the counter reading (`CV`) reaches or exceeds the target value (`PV`).

- The counter reading (`CV`) is of type `UDINT` and can take values between `0` and `4294967295`.

- The function block is deterministic and performs the counting operation on every `REQ` event.

The function block has no internal states other than the current counter reading (`CV`). The logic is re-evaluated with each `REQ` event.

- Counting events in control applications (e.g., production counters).
- Monitoring processes where a specific number of steps must be completed.
- As part of a larger control logic that reacts to counter readings.
- Compared to other counters (e.g., `FB_CTU_INT`), this function block uses the `UDINT` data type, which allows for a larger range of values.
- Similar function blocks might offer additional features such as counting down or combined counting up/down.

The `FB_CTU_UDINT` is a simple and efficient up counter for applications that require a large range of values. Its clear interface and deterministic functionality make it a reliable component in control applications.

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

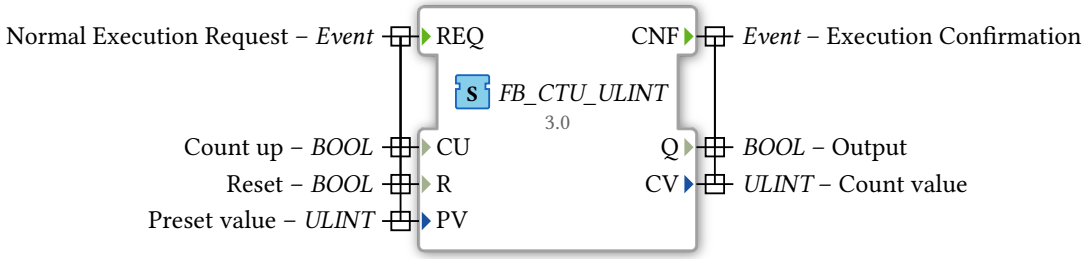
Conclusion

FB_CTU_ULINT

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_CTU_ULINT` is an up counter for the data type `ULINT` (Unsigned Long Integer). It increments with each counting event and can be reset to a predefined value. This block is particularly suitable for applications requiring large counting ranges.



Interface Structure

Event Inputs

- **REQ**: Normal execution request. Used to trigger the counting process. Linked to the data `CU` and `R`.

Event Outputs

- **CNF**: Execution Confirmation. Triggered after processing the counting operation. Linked to the data `Q` and `CV`.

Data Inputs

- **CU** (BOOL): Count Request. Increments the counter when `TRUE` is active, unless `R` is active.
- **R** (BOOL): Reset. Resets the counter value (`CV`) to `0` when `TRUE` is active.
- **PV** (ULINT): Preset Value. Defines the threshold at which the output `Q` is activated.

Data Outputs

- **Q** (BOOL): Output. Outputs `TRUE` when the counter value (`CV`) reaches or exceeds the preset value (`PV`).
- **CV** (ULINT): Current Counter Value. Returns the current counter value.

Functionality

This function block increments the counter value (`CV`) by 1 when the input `CU` is activated (`TRUE`) and the counter value has not reached the maximum value (`18446744073709551615`). When the input `R` is enabled (`TRUE`), the counter value is reset to `0`. The output `Q` is set to `TRUE` as soon as the counter value (`CV`) reaches or exceeds the preset value (`PV`).

Technical Features

- The counter value (`CV`) is of type `ULINT`, which allows for a very large counting range (0 to 18446744073709551615).
- The block is deterministic and suitable for real-time applications.

Status Overview

- **Reset State**: `CV = 0` (if `R = TRUE`).
- **Count State**: `CV` is incremented by 1 (if `CU = TRUE` and `R = FALSE`).
- **Threshold Reached**: `Q = TRUE` (if `CV >= PV`).

Application Scenarios

- Counting production units in high-speed systems.
- Monitoring events in long-running systems.
- Controlling processes with large counting ranges.

⚖️ Comparison with Similar Function Blocks

- Compared to `FB_CTU_INT` or `FB_CTU_DINT`, `FB_CTU_ULINT` offers a significantly larger counting range.
- Similar to `FB_CTU`, but specialized for the `ULINT` data type.

Conclusion

The `FB_CTU_ULINT` is a powerful increment counter for applications requiring large counting ranges. Its ease of use and deterministic operation make it a reliable choice for industrial control applications.

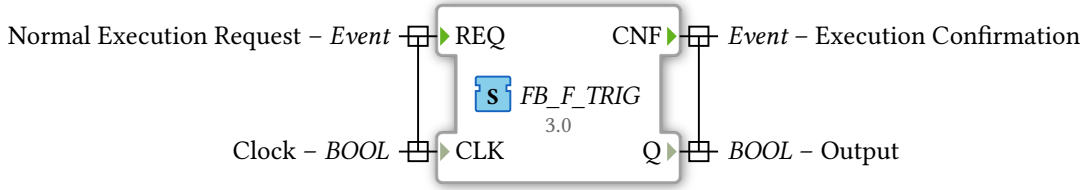
2.11.4.9 Edge Detection

Here you will find function blocks for detecting signal edges (rising, falling).

FB_F_TRIG

• – ◦ • – ◦ • – ◦ •

The function block `FB_F_TRIG` is used to detect the falling edge of a Boolean signal. It is frequently used in control applications to detect state changes from `TRUE` to `FALSE` and trigger corresponding actions.



- `REQ` (Normal Execution Request): Starts the execution of the function block. It is linked to the data input `CLK`.
- `CNF` (Execution Confirmation): Confirms the execution of the function block. Linked to the data output `Q`.
- `CLK` (BOOL): The signal to be monitored, whose falling edge is to be detected.
- `Q` (BOOL): Outputs `TRUE` if a falling edge is detected, otherwise `FALSE`.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- No adapters available.

The function block stores the previous state of the `CLK` signal in an internal variable `MEM`. On each call via the `REQ` event, it checks whether the current `CLK` signal is `FALSE` and whether the previous state (`MEM`) was `TRUE`. If so, `Q` is set to `TRUE`; otherwise, it is set to `FALSE`. The current state of `CLK` is then stored in `MEM`.

- The function block uses an internal memory variable (`MEM`) to store the previous state of the `CLK` signal.
- The logic for detecting the falling edge is implemented in an ST algorithm.

1. **Initialization**: `MEM` is initialized with `TRUE`.
2. **Falling Edge Detection**:

- When `CLK` transitions from `TRUE` to `FALSE`, `Q` is set to `TRUE`.
- In all other cases, `Q` is set to `FALSE`.

1. **Saving the Current State**: The current state of `CLK` is saved to `MEM`.

- Detection of button or switch actuations in control systems.
- Synchronization of processes upon signal changes.
- Triggering of actions on falling edges in time-critical applications.
- **FB_R_TRIG**: Detection of a rising edge (from `FALSE` to `TRUE`).
- **FB_R_F_TRIG**: Combined detection of rising and falling edges. - **FB_F_TRIG** is specialized for detecting falling edges and is therefore more efficient in this specific application.
- Exercise_178
- Exercise_178_AX

The `FB_F_TRIG` function block is a simple and efficient function block for detecting falling edges in Boolean signals. Its clear interface and simple operation make it ideal for control applications where state changes need to be detected.

Functionality

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

🛠️ Zugehörige Übungen

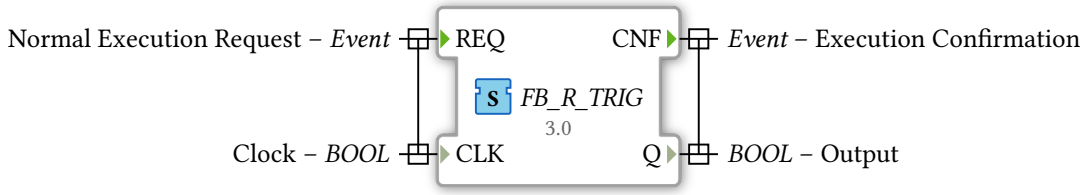
Conclusion

FB_R_TRIG

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_R_TRIG` is used to detect the rising edge of a Boolean signal. It is typically used to detect and react to state changes from `FALSE` to `TRUE`.



Interface Structure

Event Inputs

- **REQ**: Normal Execution Request. Used to trigger edge detection. Linked to the data input `CLK`.

Event Outputs

- **CNF**: Execution Acknowledgement. Triggered after edge detection is complete. Linked to the data output `Q`.

Data Inputs

- **CLK** (BOOL): The input signal whose rising edge is to be detected.

Data Outputs

- **Q** (BOOL): The output signal `TRUE` is generated when a rising edge is detected at the input `CLK`.

Adapters

This function block does not use any adapters.

Functionality

`FB_R_TRIGstores the previous state of the input signalCLKin an internal variableMEM. Each time the algorithm is called via the event REQ, it checks whether the current state was CLKorTRUEand the previous state wasFALSE. If so, the output Qis set toTRUE; otherwise, it is set to FALSE. The current state of CLKis then stored inMEM` for the next iteration.

Technical Features

- The function block is implemented according to the IEC 61499-1 standard.
- It uses a simple edge detection algorithm that can be executed efficiently and quickly.

State Overview

This function block does not have a complex state machine. The state is represented solely by the value of the internal variable `MEM`, which stores the previous state of `CLK`.

Application Scenarios

- Detection of key presses or switch activations.
- Synchronization of processes to rising edges of control signals.
- Triggering of actions upon specific events.

⚖️ Comparison with Similar Function Blocks

- **FB_F_TRIG**: Detection of a falling edge (from `TRUE` to `FALSE`).
- **FB_R_TRIG** is specific for rising edges and therefore complementary to `FB_F_TRIG`.

🛠️ Related Exercises

- Exercise_177
- Exercise_177_AX

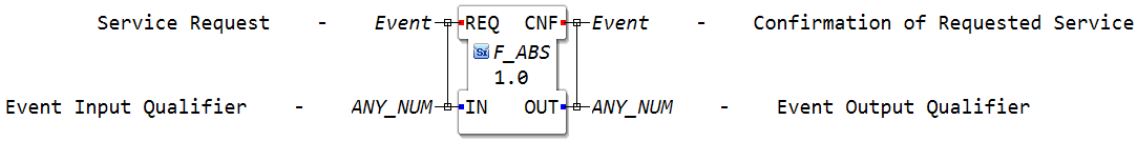
Conclusion

The `FB_R_TRIG` function block is a simple and effective function block for detecting rising edges. Its clear interface structure and simple functionality make it a useful component in many control applications.

2.11.4.10 Numerical Functions

Here you will find mathematical functions such as trigonometric functions, logarithms, roots, etc. ...

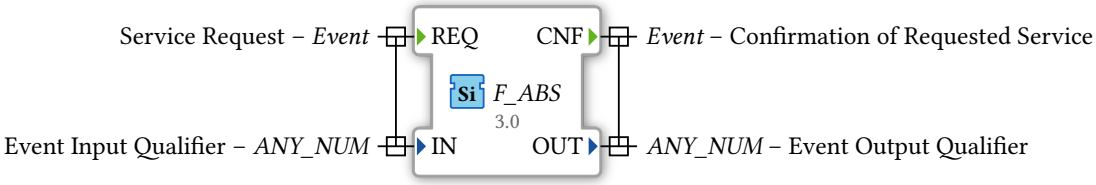
F_ABS



• – ◦ • – ◦ • – ◦ •

Introduction

The F_ABS function block calculates the absolute value of a numeric input. It belongs to the category of standard functions for numeric variables according to IEC 61131-3 and is part of the iec61131::numerical package.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the calculation of the absolute value

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the calculation

Data Inputs

- **IN**: Numeric Input (ANY_NUM) - the value whose absolute value is to be calculated

Data Outputs

- **OUT**: Numeric Output (ANY_NUM) - the calculated absolute value of the input

Adapters

No adapter interfaces are available.

Functionality

Upon receiving a REQ event, the F_ABS block calculates the absolute value of the numeric value passed to IN. The result is output to OUT and confirmed by a CNF event. The block supports all numeric data types (ANY_NUM).

Technical Features

- Supports all numeric data types according to IEC 61131-3
- Simple, deterministic operation without internal states
- Immediate calculation upon REQ event

State Overview

The F_ABS block has no internal states. It behaves purely functionally and reacts immediately to incoming REQ events.

Application Scenarios

- Signal processing for sign removal
- Magnitude calculations in mathematical operations
- Use in control engineering for distance calculations

- Preprocessing of measured values

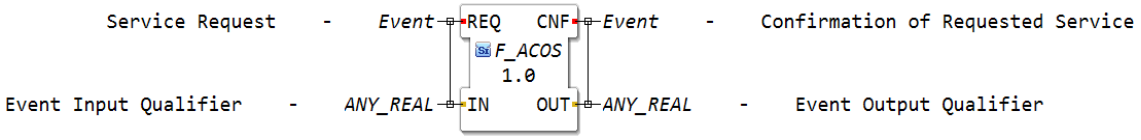
⚖️ Comparison with Similar Function Blocks

Compared to other mathematical function blocks, F_ABS stands out due to its simplicity and specific functionality. While blocks like F_ADD or F_MUL perform more complex operations, F_ABS focuses exclusively on magnitude calculation.

Conclusion

The F_ABS function block offers a reliable and standards-compliant solution for calculating absolute values in IEC 61131-3-based control systems. Its simple interface and deterministic functionality make it a robust building block for numerical operations.

F_ACOS

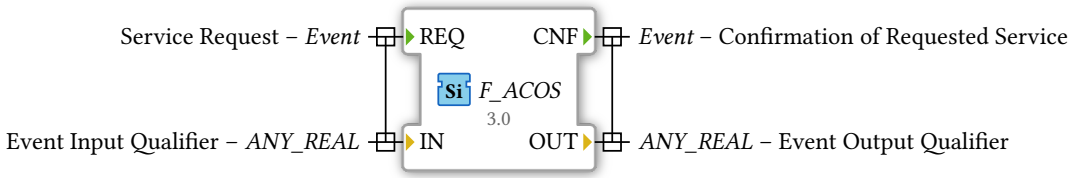


• – ◦ • – ◦ • – ◦ •

Introduction

The F_ACOS function block calculates the principal value of the arccosine of a numerical input. It implements the mathematical function for determining the angle whose cosine corresponds to the input value. The block belongs to the category of standard functions for numeric variables according to IEC 61131-3.

•



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the calculation of the arccosine

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the calculation

Data Inputs

- **IN**: Input value of type ANY_REAL - the value for which the arccosine is to be calculated

Data Outputs

- **OUT**: Output value of type ANY_REAL - the result of the arccosine calculation

Adapters

No adapter interfaces are available.

Functionality

Upon receipt of a REQ event, the F_ACOS block calculates the arccosine of the current IN value. The result is output to the OUT port, and a CNF event is simultaneously generated to signal the successful completion of the operation.

The function calculates the prime value of the arccosine, which lies in the range from 0 to π (0° to 180°).

Technical Features

- Supports the ANY_REAL data type for maximum flexibility with numeric inputs
- Compliant with the IEC 61131-3 standard for programmable logic controllers (PLCs)

- Returns the mathematically correct principal value of the arccosine
- Guarantees defined behavior for valid input values in the range [−1, 1]

State Transitions

1. **Idle State**: Waits for a REQ event
2. **Calculating State**: Processes the arccosine function upon REQ input
3. **Output State**: Sends a CNF event with the calculated OUT value

Application Scenarios

- Angle measurement and calculation in technical systems
- Robotics and motion control
- Signal processing and control engineering
- Scientific calculations and simulations
- Geometric transformations in CAD/CAM systems

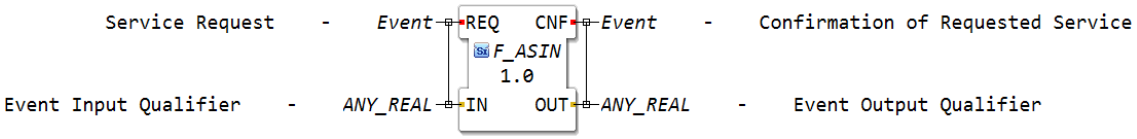
⚖️ Comparison with Similar Function Blocks

Compared to other trigonometric functions such as F_SIN F_ACOS (sine) or F_COS (cosine) provides the inverse operation. While F_COS maps an angle to a value between −1 and 1, F_ACOS maps a value between −1 and 1 to an angle between 0 and π.

Conclusion

The F_ACOS function block provides a reliable and standards-compliant implementation of the arccosine function for industrial control systems. Its simple interface and deterministic operation make it ideal for real-time applications in automation solutions.

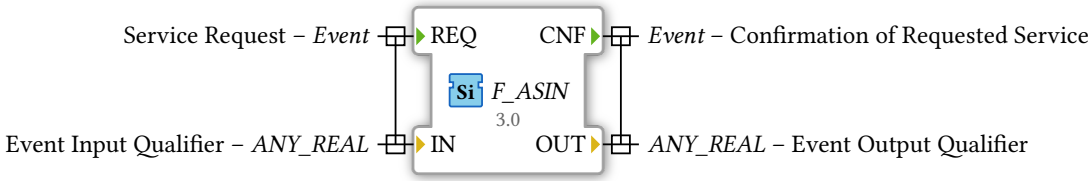
F_ASIN



• – ◦ • – ◦ • – ◦ •

Introduction

The F_ASIN function block is a standard function block according to IEC 61131-3 that calculates the arcsine of a numeric input. It belongs to the category of standard mathematical functions for a numeric variable and returns the principal value of the arcsine function.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the arcsine calculation

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the calculation

Data Inputs

- **IN**: Input value of type ANY_REAL - the value for which the arcsine is to be calculated

Data Outputs

- **OUT**: Output value of type ANY_REAL - the result of the arcsine calculation

Adapters

No adapter interfaces are available.

Functionality

The F_ASIN function block calculates the arcsine (inverse sine function) of the input value IN. The calculation is performed when the REQ event is triggered. After the calculation is complete, the CNF event is output along with the result value OUT.

The function operates according to the mathematical definition: OUT = arcsin(IN), where the result lies in the range [-π/2, π/2] radians.

Technical Features

- Supports the ANY_REAL data type for maximum flexibility with various real number types
- Compliant with the IEC 61131-3 standard for numeric functions
- Simple event-driven processing
- Returns the principal value of the arcsine function

State Overview

The function block has a simple state machine:

1. **Wait State**: Waits for a REQ event
2. **Calculate State**: Performs an arcsine calculation
3. **Output State**: Sends a CNF event with the result

Application Scenarios

- Calculating angles from sine values in trigonometric applications
- Signal processing in control engineering and automation
- Geometric calculations in technical systems
- Scientific and engineering calculations

⚖️ Comparison with Similar Function Blocks

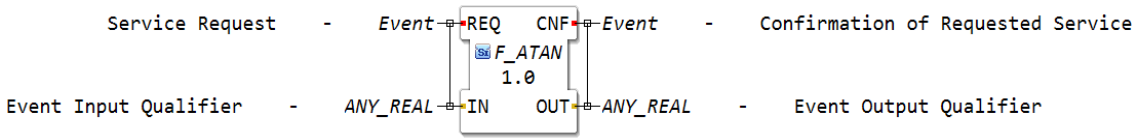
Compared to other trigonometric functions:

- F_ASIN is the inverse function of F_SIN
- Similar to F_ACOS (Arc cosine) and F_ATAN (Arc tangent)
- Returns results within the specific range of values for the arcsine function

Conclusion

The F_ASIN function block provides a standards-compliant, reliable implementation of the arcsine function for automation applications. Its simple interface and event-driven processing make it ideal for use in more complex control algorithms that require trigonometric calculations.

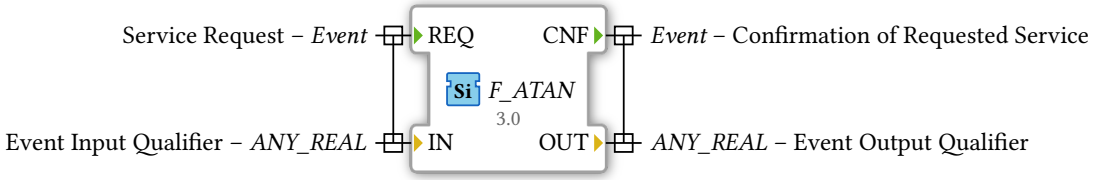
F_ATAN



• — ○ • — ○ • — ○ •

Introduction

The F_ATAN function block calculates the arctangent of a real number. It is a standard function for a numeric variable according to IEC 61131-3. The function block converts a tangent value into the corresponding angle in radians.



Interface Structure

Event Inputs
- **REQ**: Service Request - triggers the calculation of the arctangent

Event Outputs
- **CNF**: Confirmation of the requested service - signals the completion of the calculation

Data Inputs
- **IN**: Input value of type ANY_REAL - the tangent value for which the arctangent is to be calculated

Data Outputs
- **OUT**: Output value of type ANY_REAL - the calculated arctangent value in radians

Adapters
No adapter interfaces are available.

Functionality
Upon receiving a REQ event, the F_ATAN function block calculates the arctangent of the value passed to IN. The result is output to OUT, and a CNF event is simultaneously generated to signal the completion of the calculation.

The function calculates the principal value of the arctangent, which lies in the range of $-\pi/2$ to $\pi/2$.

Technical Features
- Supports the ANY_REAL data type, enabling various real data types (REAL, LREAL)
- Compliant with the IEC 61131-3 standard
- Deterministic computational behavior
- Suitable for real-time applications

State Transitions
1. **Initial State**: Waits for a REQ event
2. **Computation State**: ATAN(IN) is calculated upon a REQ event
3. **Output State**: The result is sent to OUT and a CNF event is triggered
4. Returns to the initial state

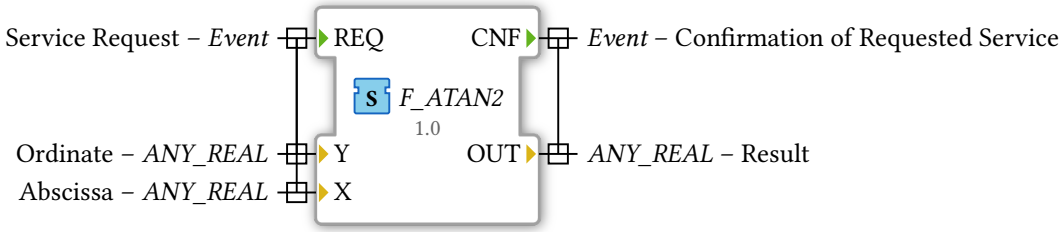
Application Scenarios
- Angle measurement and calculation in control systems
- Coordinate transformations in robotics applications
- Control engineering for position controllers
- Signal processing in mathematical calculations
- Navigation and orientation systems

⚖️ Comparison with Similar Function Blocks
- **F_ATAN2**: Calculates the arctangent of two coordinates (x,y) and returns an angle in the range from $-\pi$ to π
- **F_SIN/F_COS**: Basic trigonometric functions for sine and cosine
- **F_TAN**: Calculates the tangent of an angle (inverse function of F_ATAN)

Conclusion

The F_ATAN function block offers a reliable and standards-compliant implementation of the arctangent function for automation applications. Its simple interface and deterministic operation make it ideal for real-time applications in industrial automation where trigonometric calculations are required.

F_ATAN2



• – ◦ • – ◦ • – ◦ •

Introduction

`F_ATAN2` computes the principal value of the arc tangent of `Y/X`, taking into account the signs of both arguments (two-parameter arc tangent). Unlike the plain `ATAN` function (which takes only one argument), `F_ATAN2` returns the angle in the correct quadrant across the full angular range from `-π` to `+π`.

Interface Structure

Event Inputs

- **REQ**: Triggers the calculation, carries `Y` and `X`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **Y** (ANY_REAL): Ordinate (Y coordinate).
- **X** (ANY_REAL): Abscissa (X coordinate).

Data Outputs

- **OUT** (ANY_REAL): The computed angle in radians, in the range `(-π, π]`.

Functionality

On `REQ`, the block computes the angle between the positive X axis and the point `(X, Y)` — equivalent to `ATAN(Y/X)`, but evaluating the signs of `X` and `Y` individually to determine the correct quadrant (including the special case `X = 0`). The result is output via `OUT`, then `CNF` is triggered.

Technical Features

- **Quadrant correctness**: Unlike `ATAN(Y/X)`, `F_ATAN2` returns the correct angle even for negative `X` (2nd and 3rd quadrant), without the caller having to manually correct for the quadrant.
- **`ANY_REAL` typing**: Accepts both `REAL` and `LREAL` for `Y`, `X`, and `OUT`.
- **Special case `X = 0`**: Returns `+π/2` or `-π/2` depending on the sign of `Y`, instead of a division by zero as with directly using `ATAN(Y/X)`.

State Overview

Stateless: every `REQ` immediately results in the calculation and `CNF`.

Application Scenarios

- **Angle calculation from Cartesian coordinates**: e.g. determining the heading or orientation of an object from its X/Y position relative to a reference point.
- **Control engineering**: Computing phase angles or steering angles from two component signals.
- **Navigation**: Course calculation from velocity components.

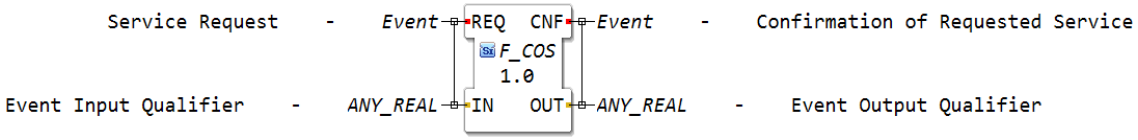
Comparison with similar function blocks

- **F_ATAN**: computes the arc tangent of a single value, only returns angles in the range (-π/2, π/2), and requires manual quadrant correction.

Conclusion

F_ATAN2 provides quadrant-correct angle calculation from two Cartesian components and is the standard block for angle/direction calculations where ATAN alone is insufficient.

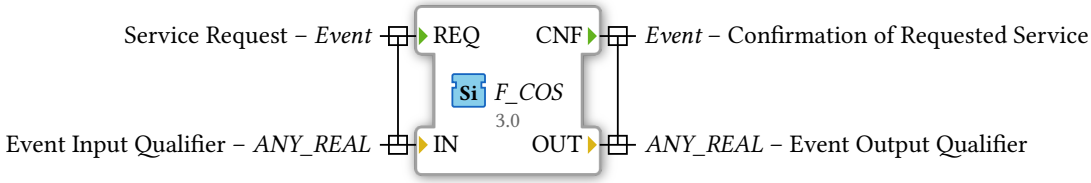
F_COS



• — ◦ • — ◦ • — ◦ •

Introduction

The F_COS function block calculates the cosine of an input value in radians. It is a standard function for numerical calculations according to IEC 61131-3.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the cosine calculation

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the calculation

Data Inputs

- **IN**: Input value in radians (ANY_REAL data type)

Data Outputs

- **OUT**: Calculated cosine value of the input value (ANY_REAL data type)

Adapters

No adapters available.

Functionality

Upon receiving a REQ event, the F_COS block calculates the cosine of the value passed to IN. The input value is interpreted as an angle in radians. After the calculation is complete, the CNF event is output to OUT along with the result.

Technical Features

- Supports the ANY_REAL data type for maximum flexibility
- Calculations are performed in radians
- Compliant with the IEC 61131-3 standard
- Part of the iec61131::numerical package

State Transitions

1. **Wait State**: Block waits for a REQ event
2. **Calculate State**: Cosine is calculated upon a REQ event
3. **Output State**: CNF event with result is output

4. **Return**: Returns to the wait state

Application Scenarios

- Signal processing in control engineering
- Mathematical calculations in control systems
- Trigonometric functions in technical applications
- Calculations in robotics and motion control

⚖️ Comparison with Similar Function Blocks

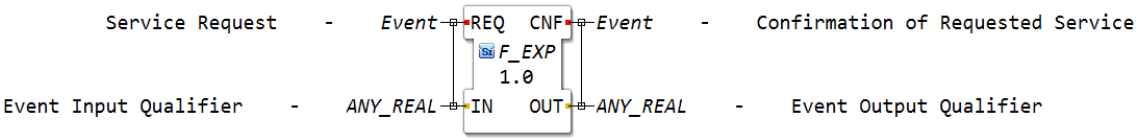
Compared to other trigonometric functions:

- F_SIN: Calculates sine instead of cosine
- F_TAN: Calculates tangent
- F_ACOS: Calculates Arccosine (Inverse Function)

Conclusion

The F_COS function block offers a reliable and standards-compliant implementation of the cosine function for industrial control systems. Its simple interface and flexible data type support make it ideal for various automation applications.

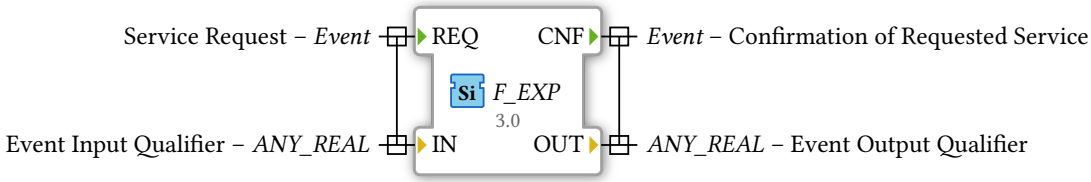
F_EXP



• – ◦ • – ◦ • – ◦ •

Introduction

The F_EXP function block calculates the natural exponential function (e^x) for numerical input values. It is one of the standard functions for numerical variables according to IEC 61131-3 and is used for mathematical calculations in control applications.



Interface Structure

Event Inputs

- **REQ**: Service Request - Triggers the calculation of the exponential function

Event Outputs

- **CNF**: Confirmation of Requested Service - Confirms successful calculation

Data Inputs

- **IN**: ANY_REAL - Numeric input value for which the exponential function is to be calculated

Data Outputs

- **OUT**: ANY_REAL - Result of the exponential function (e^IN)

Adapters

No adapter interfaces are available.

Functionality

Upon receipt of a REQ event, the F_EXP block calculates the natural exponential function of the value passed to IN. The result is output to OUT and confirmed by a CNF event. The function is mathematically equivalent to calculating e^x, where x is the input value.

Technical Features

- Supports the ANY_REAL data type for maximum flexibility with numerical inputs
- Real-time calculation of the exponential function
- Simple and direct interface without additional parameters
- Implemented in accordance with the IEC 61131-3 standard

State Transitions

1. **Wait State**: Block waits for REQ event
2. **Calculate State**: Upon REQ event, e^IN is calculated
3. **Output State**: Result is set to OUT and a CNF event is triggered
4. Return to Wait State

Application Scenarios

- Mathematical modeling and simulation
- Gain calculations in control engineering
- Growth processes and exponential developments
- Signal processing and filtering
- Scientific calculations in industrial control systems

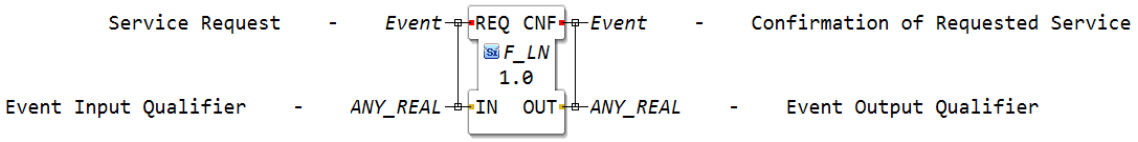
⚖️ Comparison with similar function blocks

Compared to other mathematical function blocks such as F_LN (natural logarithm) or F_POWER (power function), F_EXP offers a A specialized implementation for the exponential function with optimized performance. The use of ANY_REAL enables broader compatibility than type-specific implementations.

Conclusion

The F_EXP function block provides a reliable and standards-compliant implementation of the natural exponential function for industrial control systems. Its simple interface and flexible data type support make it ideal for various mathematical applications in automation projects.

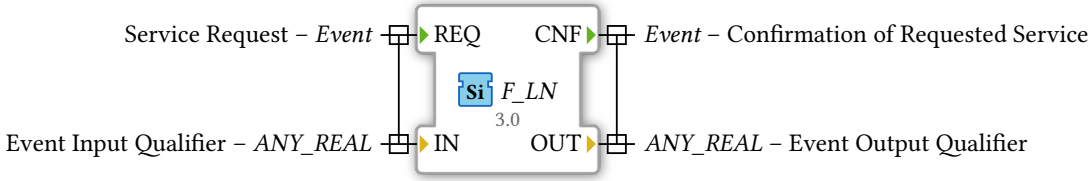
F_LN



• — ○ • — ○ • — ○ •

Introduction

The F_LN function block calculates the natural logarithm (logarithm to base e) of a numeric input value. It belongs to the category of standard functions for a numeric variable according to the IEC 61131-3 standard.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the calculation of the natural logarithm

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the calculation

Data Inputs

- **IN**: Numeric input value of type ANY_REAL for which the natural logarithm is to be calculated

Data Outputs

- **OUT**: Calculated natural logarithm of the input value of type ANY_REAL

Adapters

No adapter interfaces are available.

Functionality

Upon receiving a REQ event, the F_LN block calculates the natural logarithm (ln) of the value passed to IN. The result is output to OUT, and simultaneously, a CNF event is generated to signal the successful completion of the operation.

Technical Features

- Supports the ANY_REAL data type, which includes various numeric data types such as REAL, LREAL, etc.
- Implemented according to the IEC 61131-3 standard for numeric functions
- Real-time execution with event-driven control

State Overview

The function block features a simple state machine:

1. **Wait State**: Block waits for a REQ event
2. **Calculate State**: Processes the logarithm calculation
3. **Output State**: Sends a CNF event with the calculated result

Application Scenarios

- Scientific calculations and mathematical modeling
- Signal processing and data analysis
- Control engineering for exponential processes
- Scaling of measured values in logarithmic representations

⚖️ Comparison with Similar Blocks

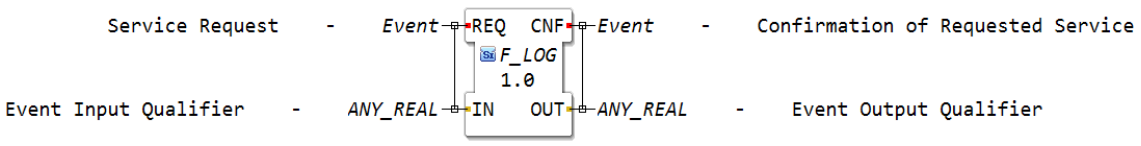
Compared to other mathematical function blocks:

- F_LN specializes in natural logarithms (base e)
- F_LOG calculates logarithms to any value Basis
- F_EXP calculates the exponential function (inverse function of LN)
- Simpler structure than general logarithm functions

Conclusion

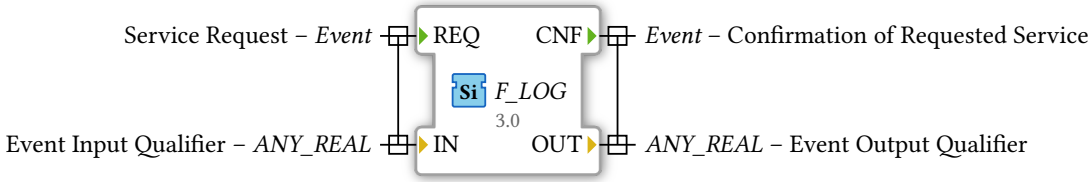
The F_LN function block offers an efficient and standards-compliant implementation for calculating the natural logarithm. Its event-driven architecture makes it particularly suitable for real-time applications where mathematical operations must be performed precisely and reliably.

F_LOG



Introduction

The F_LOG function block calculates the base-10 logarithm (decimal logarithm) of a numeric input value. It belongs to the category of standard functions for a numeric variable according to IEC 61131-3 and is part of the package `iec61131::numerical`.



Interface Structure

Event Inputs

- **REQ**: Service request - triggers the calculation of the logarithm

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the calculation

Data Inputs

- **IN**: Numeric input value of type ANY_REAL for which the logarithm is to be calculated

Data Outputs

- **OUT**: Calculated logarithm value of type ANY_REAL

Adapters

No adapter interfaces are available.

Functionality

Upon arrival of a REQ event, the F_LOG block calculates the base-10 logarithm ($\log_{10}$) of the current IN value. The result is output at the OUT output, and simultaneously a CNF event is generated to signal successful computation.

Technical Features

- Supports the ANY_REAL data type, enabling various numeric data types such as REAL, LREAL, etc.
- Complies with the IEC 61131-3 standard specification
- Simple and deterministic operation without internal states

State Overview

The function block has no complex internal states. It behaves purely functionally: REQ event ⟶ computation ⟶ CNF event.

Application Scenarios

- Scientific calculations and measurement data processing
- Signal processing in automation technology
- Scaling of measured values in logarithmic representations
- Audio and acoustics applications (e.g., dB calculations)

⚖️ Comparison with similar blocks

Compared to other logarithmic functions:

- F_LN: Calculates the natural logarithm (base e)
- F_LOG: Specifically calculates the base-10 logarithm
- Both blocks follow the same event/data flow paradigm

Conclusion

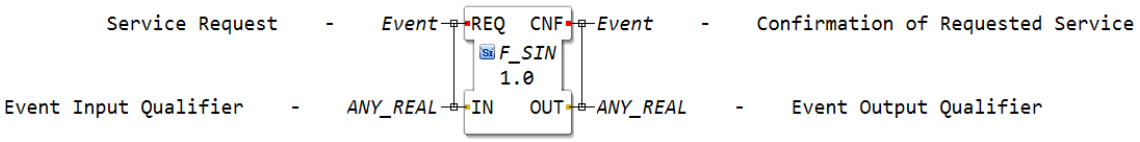
The F_LOG function block offers a standardized, reliable implementation for calculating the base-10 logarithm in 4diac control systems. Its simple interface and deterministic operation make it ideal for use in real-time automation applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

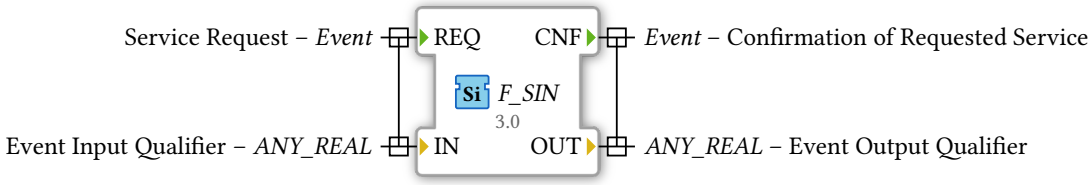
F_SIN



• – ◦ • – ◦ • – ◦ •

Introduction

The F_SIN function block calculates the sine of an input value in radians. It is a standard function for numerical calculations according to IEC 61131-3, enabling mathematical sine operations in control applications.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the sine calculation

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the calculation

Data Inputs

- **IN**: Input value in radians (Type: ANY_REAL) - the angle whose sine is to be calculated

Data Outputs

- **OUT**: Output value (Type: ANY_REAL) - the result of the sine calculation

Adapters

No adapter interfaces are available.

Functionality

Upon receiving a REQ event, the function block calculates the sine of the value passed to IN. The result is output to OUT, and simultaneously a CNF event is generated to signal successful calculation. The calculation is performed according to the standard mathematical definition of the sine function.

Technical Features

- Supports the ANY_REAL data type for maximum flexibility with numerical inputs
- Compliant with the IEC 61131-3 standard for programmable logic controllers (PLCs)
- Simple event-driven processing
- Direct mapping of the mathematical sine function

State Transitions

The function block has two main states:

1. **Wait State**: Waits for a REQ event
2. **Calculate State**: Processes the sine calculation and generates a CNF event

Application Scenarios

- Signal processing in control engineering
- Calculation of oscillations and periodic processes
- Mathematical transformations in technical applications
- Position calculations in robotics and motion control

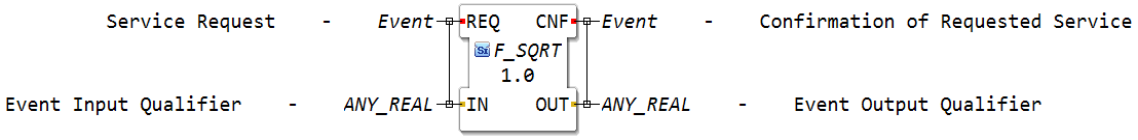
⚖️ Comparison with Similar Function Blocks

Compared to other trigonometric functions such as F_COS (cosine) or F_TAN (tangent), F_SIN offers a specific sine calculation. Compared to general-purpose mathematical blocks, F_SIN stands out due to its specialized and optimized implementation.

Conclusion

The F_SIN function block provides a reliable and standards-compliant solution for sine calculations in IEC 61131-3-based control systems. Its simple interface and direct functionality make it ideal for numerous technical applications requiring trigonometric calculations.

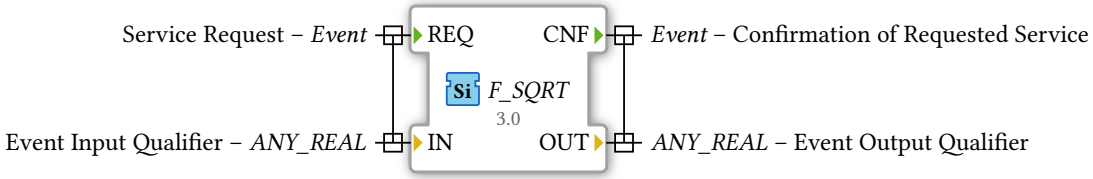
F_SQRT



• — ◦ • — ◦ • — ◦ •

Introduction

The F_SQRT function block is a standard function block according to IEC 61131-3 that calculates the square root of a numeric input. It belongs to the category of numeric functions and is used for the mathematical calculation of the square root of any real input value.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the square root calculation

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the calculation

Data Inputs

- **IN** (ANY_REAL): Input value from which the square root is to be calculated

Data Outputs

- **OUT** (ANY_REAL): Result of the square root calculation

Adapters

No adapter interfaces are available.

Functionality

The F_SQRT function block calculates the square root of the input value IN. The calculation is performed when activated by the REQ event. After the calculation is complete, the CNF event is triggered, and the result is output via the OUT variable.

Technical Features

- Uses the ANY_REAL data type for maximum flexibility with various real number types
- Complies with the IEC 61131-3 standard specification
- Simple and direct functionality without additional parameters

State Transitions

1. **Wait State**: Block waits for a REQ event
2. **Calculate State**: Upon a REQ event, the square root is calculated
3. **Output State**: After calculation, a CNF event is triggered and the result is output

Application Scenarios

- Mathematical calculations in control systems
- Signal processing and filtering
- Scientific and technical applications
- Control engineering and process control

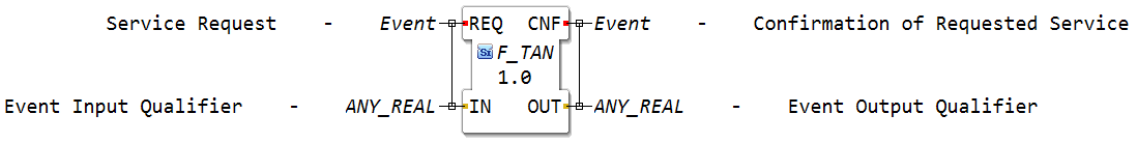
⚖️ Comparison with Similar Blocks

Compared to other mathematical function blocks, F_SQRT offers a specialized function for calculating square roots. Other blocks such as F_ADD or F_MUL offer basic arithmetic operations, while F_SQRT provides a specific mathematical function.

Conclusion

The F_SQRT function block is an essential tool for numerical calculations in IEC 61131-3 based control systems. Its simple interface and reliable functionality make it a valuable component for mathematical operations in industrial automation solutions.

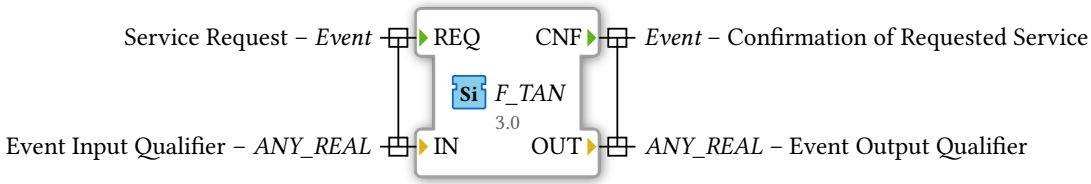
F_TAN



• – ◦ • – ◦ • – ◦ •

Introduction

The F_TAN function block calculates the tangent of an input value in radians. It is a standard function for numerical calculations according to IEC 61131-3.



Interface Structure

Event Inputs

- **REQ**: Service request - triggers the tangent calculation

Event Outputs

- **CNF**: Confirmation of requested service - signals the completion of the calculation

Data Inputs

- **IN**: Input value in radians (ANY_REAL data type)

Data Outputs

- **OUT**: Calculated tangent value (ANY_REAL data type)

Adapters

No adapter interfaces are available.

Functionality

Upon receiving a REQ event, the F_TAN block calculates the tangent of the angle passed to IN in radians. The result is output to OUT, and simultaneously, a CNF event is generated to signal the completion of the calculation.

Technical Features

- Supports the ANY_REAL data type for maximum flexibility with numeric inputs
- Calculations are performed in radians (not degrees)
- Implemented according to the IEC 61131-3 standard for numeric functions
- Mathematical peculiarity: Very large values or overflows can occur for angles close to $\pi/2 + k\pi$ ($k \in \mathbb{Z}$)

State Transitions

1. **Wait State**: Block waits for a REQ event
2. **Calculate State**: Upon a REQ event, the tangent is calculated
3. **Output State**: Result is sent to OUT and a CNF event is generated
4. Return to Wait State

Application Scenarios

- Trigonometric calculations in control systems
- Signal processing and filtering
- Robotics and motion control
- Mathematical modeling and simulation
- Control engineering and position determination

⚖️ Comparison with similar function blocks

- **F_SIN**: Calculates sine instead of tangent
- **F_COS**: Calculates cosine instead of tangent
- **F_ATAN**: Inverse function (arcutangent)
- **F_TAN** specifically provides the tangent function for trigonometric applications

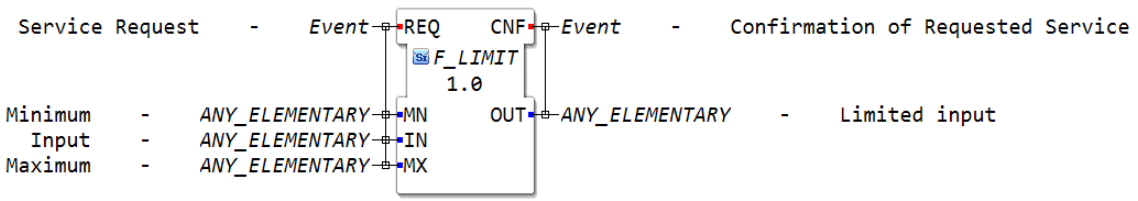
Conclusion

The F_TAN function block is an essential element for trigonometric calculations in IEC 61131-3 based control systems. Its simple and standards-compliant interface enables reliable integration into more complex control algorithms, especially in applications requiring trigonometric functions.

2.11.4.11 Selection Functions

Here you will find functions for selecting values (Max, Min, Limit, Mux, etc.).

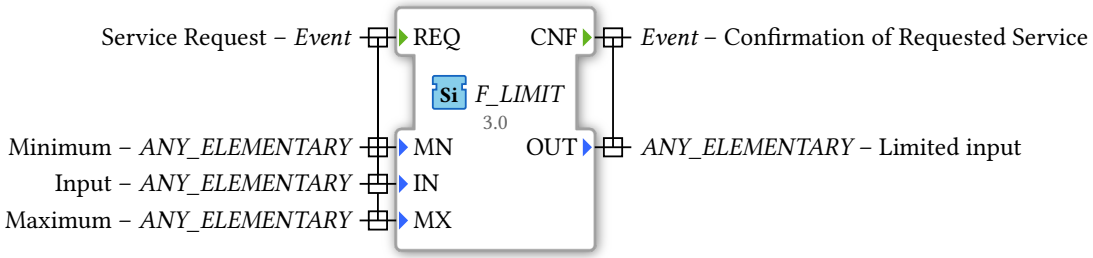
F_LIMIT



• – ◦ • – ◦ • – ◦ •

Introduction

The F_LIMIT function block is a standard selection function block according to IEC 61131-3 that acts as a limiter. It limits an input value to a defined range between a lower and upper limit.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the limiting operation

Event Outputs

- **CNF**: Acknowledgement of the requested service operation

Data Inputs

- **MN**: Minimum - lower limit for the limit
- **IN**: Input - input value to be limited
- **MX**: Maximum - upper limit for the limit

Data Outputs

- **OUT**: Limited Input - the limited output value

Adapters

No adapter interfaces are available.

Functionality

The F_LIMIT function block takes an input value (IN) and limits it to the range between the minimum (MN) and maximum (MX). When activated by the REQ event, the following operation is performed:

- If IN < MN, dann OUT = MN
- Wenn IN > MX, then OUT = MX
- If MN ≤ IN ≤ MX, then OUT = IN

After the operation is complete, the CNF event is output with the calculated OUT value.

Technical Features

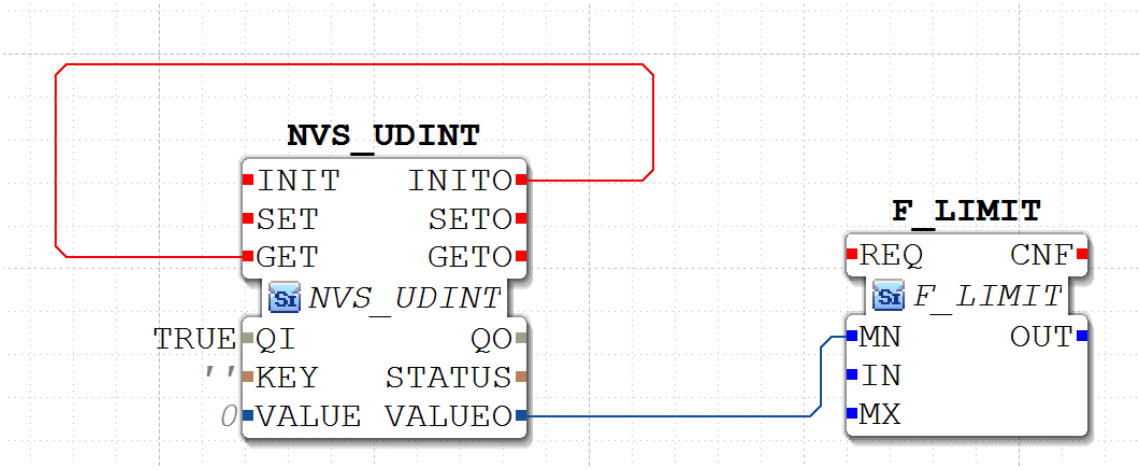
- Uses the generic data type ANY_ELEMENTARY, meaning the block can be used with various elementary data types (INT, REAL, etc.)
- All data inputs and the data output use the same data type for consistent operation
- EPL-2.0 License (Eclipse Public License)

State Transitions

1. **Wait State**: Block waits for a REQ event
2. **Active**: The limiting operation is performed upon a REQ event
3. **Output**: After calculation, a CNF event with an OUT value is output
4. **Return**: Returns to the wait state

Application Scenarios

- Signal limiting in control engineering
- Protection against exceeding and falling below limit values
- Normalization of measured values to a permissible range
- Safety limiting in process control



⚖️ Comparison with similar blocks

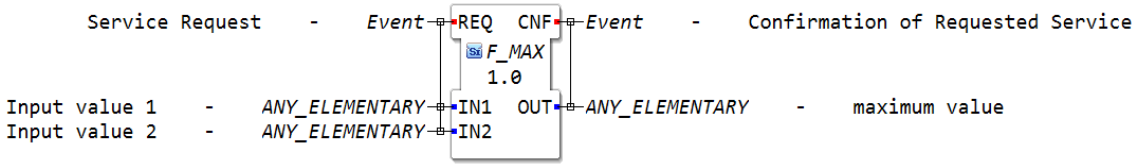
Compared to Other selection function blocks:

- F_LIMIT specifically optimized for value limiting
- Simpler than complex comparison blocks with multiple outputs
- More universally applicable due to the ANY_ELEMENTARY data type

Conclusion

The F_LIMIT function block offers a simple and efficient solution for value limiting in IEC 61131-3 based control systems. Its generic type support makes it flexible for various applications where input values need to be restricted to a defined range.

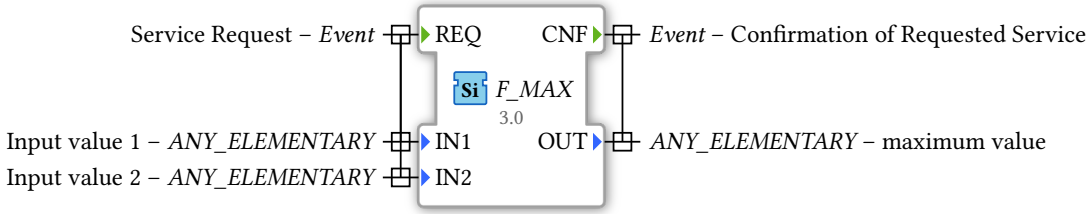
F_MAX



• – ○ • – ○ • – ○ •

Introduction

The F_MAX function block is a standard selection function block according to IEC 61131-3 that determines the maximum value from two input values. The block compares any two elementary data types and returns the larger of the two values as the result.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the calculation of the maximum

Event Outputs

- **CNF**: Confirmation of the requested service - signals completion of the calculation

Data Inputs

- **IN1**: Input value 1 (any elementary data type)
- **IN2**: Input value 2 (any elementary data type)

Data Outputs

- **OUT**: Maximum value (same data type as the inputs)

Adapters

No adapter interfaces are available.

Functionality

When a REQ event arrives, the F_MAX block compares the two input values IN1 and IN2. The larger of the two values is output at OUT. Simultaneously, a CNF event is generated to signal the completion of the operation. The block supports all elementary data types (ANY_ELEMENTARY), making it universally applicable for various numeric and non-numeric comparisons.

Technical Features

- Use of the generic data type ANY_ELEMENTARY for maximum flexibility
- Automatic type matching between inputs and output
- Real-time capable execution
- Standard-compliant implementation according to IEC 61131-3

State Transitions

1. **Wait State**: Block waits for REQ event
2. **Calculation State**: Upon REQ event, IN1 and IN2 are compared
3. **Output State**: Maximum is output to OUT and a CNF event is generated
4. Return to Wait State

Application Scenarios

- Determining the maximum in control algorithms
- Limit monitoring in process control systems
- Signal processing and filtering
- Quality assurance of measured values
- Prioritization logic in controllers

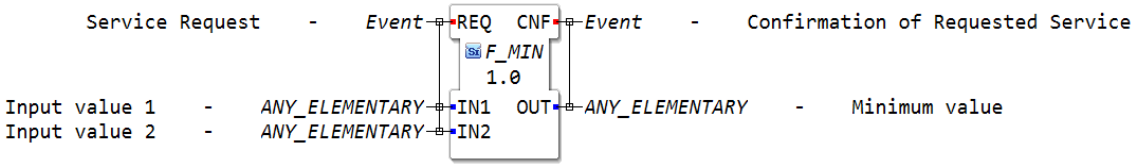
⚖️ Comparison with similar blocks

Compared to specific maximum blocks (such as MAX_INT, MAX_REAL, etc.), F_MAX offers the advantage of type independence. While specific blocks can only process one data type, F_MAX is universally applicable to all elementary data types.

Conclusion

The F_MAX function block represents a flexible and standards-compliant solution for determining maximum values in automation systems. Its type independence makes it particularly versatile, while its simple interface structure enables intuitive integration into various applications.

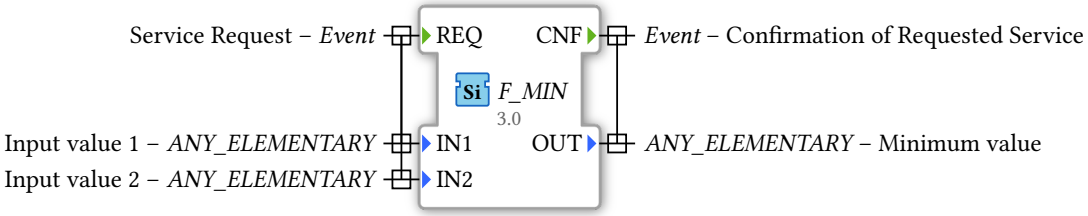
F_MIN



• — ○ • — ○ • — ○ •

Introduction

The F_MIN function block is a standard selection function block according to IEC 61131-3 that determines the minimum of two input values. The block compares any two elementary data types and returns the smaller of the two values as the result.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the calculation of the minimum

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the calculation

Data Inputs

- **IN1**: Input value 1 (any elementary data type)
- **IN2**: Input value 2 (any elementary data type)

Data Outputs

- **OUT**: Minimum value (same data type as the inputs)

Adapters

No adapter interfaces are available.

Functionality

When the REQ event is activated, the F_MIN block compares the two input values IN1 and IN2. The smaller of the two values is output at OUT. Simultaneously, the CNF event is generated to signal the completion of the operation.

Technical Features

- Supports any elementary data type (ANY_ELEMENTARY)
- Type safety: Output has the same data type as the inputs
- Real-time capable execution
- Simple and deterministic operation

State Transitions

1. **Wait State**: Block waits for REQ event
2. **Compute State**: Upon REQ event, IN1 and IN2 are compared
3. **Output State**: Minimum value is output to OUT and a CNF event is generated
4. Return to Wait State

Application Scenarios

- Limit monitoring in control engineering
- Minimum selection in data processing chains
- Safety-relevant limits
- Signal processing with dynamic input values

⚖️ Comparison with similar function blocks

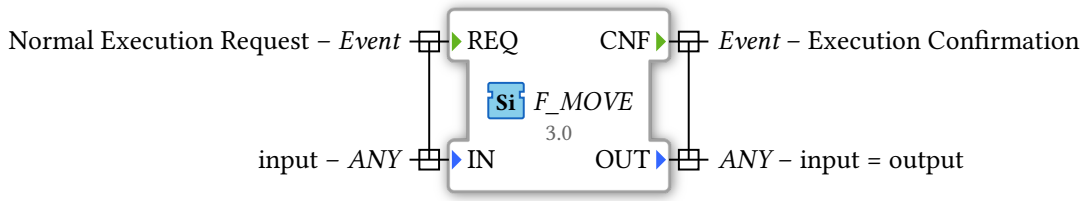
Compared to F_MAX (maximum function), F_MIN determines the smaller value instead of the larger one. Both blocks share the same interface structure and work with any elementary data type.

Conclusion

The F_MIN block represents a simple, reliable, and standards-compliant solution for minimum determination in IEC 61131-3-based control systems. Its flexibility regarding supported data types makes it a universally applicable building block in various automation applications.

F_MOVE

• – ◦ • – ◦ • – ◦ •



Introduction

The **F_MOVE** is a universal function block for type-safe data transfer, developed under the EPL-2.0 license. Version 1.0 enables the transfer of any data type (ANY) between variables in control applications.

Interface Structure

Event Inputs

- `REQ`: Transfer request (with input value)

Event Outputs

- `CNF`: Transfer acknowledgment (with output value)

Data Inputs

- `IN` (ANY): Source data (any type)

Data Outputs

- `OUT` (ANY): Destination data (identical type to input)

Functionality

1. **Activation**:

- Triggered by the `REQ` event
- Processes any data type (ANY)

1. **Data Transfer**:

- 1:1 copy of the input value
- Type-preserving transfer
- Supports all IEC 61131-3 Data Types

1. **Confirmation**:

- `CNF` event after successful transfer
- OUT receives an exact copy of IN

Technical Specifications

✓ Generic Type Support (ANY) **✓ Bit-perfect data copy ✓ One-cycle latency ✓ IEC 61131-3 compliant**

Application Examples

- **Data Conversion**: Type conversions
- **Signal Forwarding**: Process data distribution
- **Buffer Operations**: Intermediate storage
- **Interfaces**: Data encapsulation

Special Notes

- **Type Compatibility**:
- Input and output types must be identical
- No automatic type conversion
- **Performance**:
- Consistently fast execution

- Independent of data type

⚖️ Comparison with similar function blocks

Feature	F_MOVE	F_BLOCK_MOVE	F_CAST
Functionality	Single Value	Memory Block	Type Conversion
Type Support	ANY	BYTE Array	ANY
Latency	1 Cycle	Variable	1 Cycle

🛠️ Related Exercises

- Exercise_002b2
- Exercise_103c
- Exercise_110
- Exercise_111
- Exercise_171

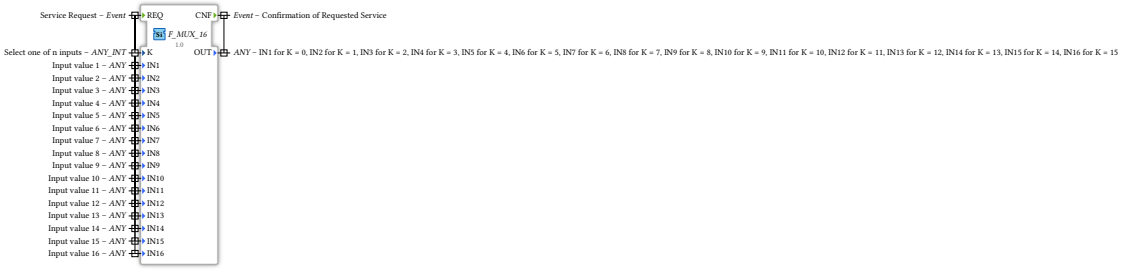
Conclusion

The F_MOVE block offers essential data transfer functions:

- Universal type support
- Reliable 1:1 data copying
- Minimal resource consumption

Indispensable for control applications with complex data structures or type-sensitive operations. The generic implementation as GEN_FORTE_F_MOVE enables maximum flexibility in heterogeneous automation environments.

F_MUX_16



• — ○ • — ○ • — ○ •

Introduction

The `F_MUX_16` is a multiplexer according to IEC 61131-3 (standard selection function, generic block) that, based on the selector `K`, switches one of **16 inputs** to the output `OUT`. It is the 16-channel variant of `F_MUX_8` and `F_MUX_4`.

Interface Structure

Event Inputs

- **REQ**: Service request. Associated data: `IN1` to `IN16`, `K`

Event Outputs

- **CNF**: Confirmation of the requested service. Associated data: `OUT`

Data Inputs

- **K** (`ANY_INT`): Selects one of the inputs
- **IN1** to **IN16** (`ANY`): Input values 1 to 16

Data Outputs

- **OUT** (ANY): Selected input value (see table)

Adapters

This function block uses no adapters.

How it works

On REQ, the input selected by K is output at OUT, then CNF is fired. K counts from 0:

K	OUT	K	OUT
0	IN1	8	IN9
1	IN2	9	IN10
2	IN3	10	IN11
3	IN4	11	IN12
4	IN5	12	IN13
5	IN6	13	IN14
6	IN7	14	IN15
7	IN8	15	IN16

Technical Details

- Generic block (GEN_F_MUX, FORTE); the input data types are ANY and must match OUT.
- Belongs to the package iec61131::selection.

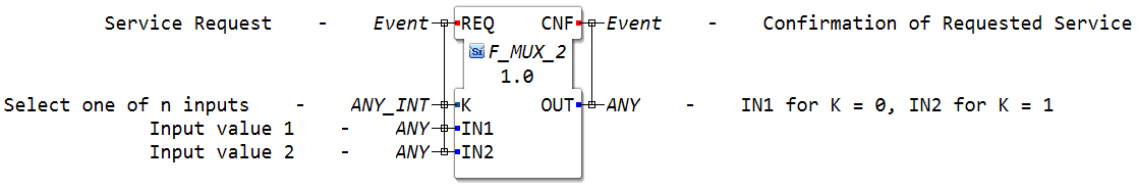
Application Scenarios

- Selecting one of up to 16 values, e.g. constants, setpoints or object IDs, via an index

Summary

F_MUX_16 switches one of 16 input values to the output based on the index K (0 to 15).

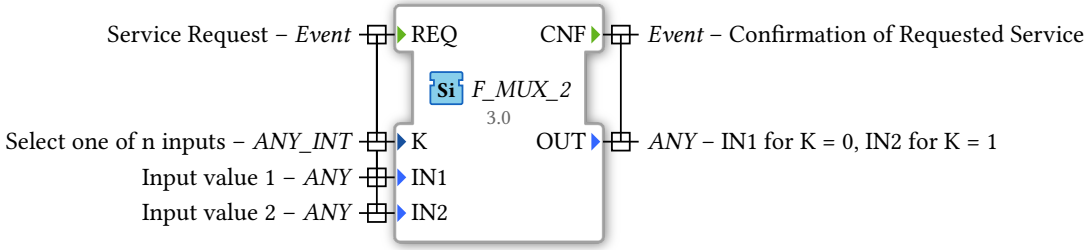
F_MUX_2



• – ◦ • – ◦ • – ◦ •

Introduction

The F_MUX_2 is a multiplexer function block according to IEC 61131-3, classified as a standard selection function. This block allows selection between two input signals based on a control signal.



Interface Structure

Event Inputs
- **REQ**: Service Request - triggers multiplexer processing

Event Outputs
- **CNF**: Confirmation of requested service - signals completion of processing

Data Inputs
- **K**: Control input for selecting one of n inputs (ANY_INT type)
- **IN1**: Input value 1 (ANY type)
- **IN2**: Input value 2 (ANY type)

Data Outputs
- **OUT**: Output value - returns IN1 for K = 0, IN2 for K = 1 (ANY type)

Adapters
No adapter interfaces are available.

Functionality
The F_MUX_2 operates as a 2:1 multiplexer. Upon receiving a REQ event, the function block evaluates the control input K:

- If K = 0, the value of IN1 is passed to OUT.
- If K = 1, the value of IN2 is passed to OUT.

After successful processing, the CNF event is triggered to signal the completion of the operation. ...

Technical Features
- Use of generic data types (ANY and ANY_INT) for maximum flexibility
- Support for various data types for input values
- Simple and deterministic processing
- EPL 2.0 license

State Transitions
1. **Wait Time**: No active processing
2. **Processing**: Upon REQ event - evaluation of K and selection of the corresponding input
3. **Completion**: Triggering of CNF after successful data transmission

Application Scenarios
- Signal diverter in control applications
- Selection between different operating modes
- Switching between redundant signal sources
- Processing path selection in complex control systems

⚖️ Comparison with Similar Components
Compared to other multiplexers, F_MUX_2 offers:

- Specifically optimized for 2 inputs
- Simpler design than multiplexers with more inputs
- Direct and clear control logic
- Lower resource consumption with 2-way selection

🛠️ Related Exercises
- Exercise_090a1

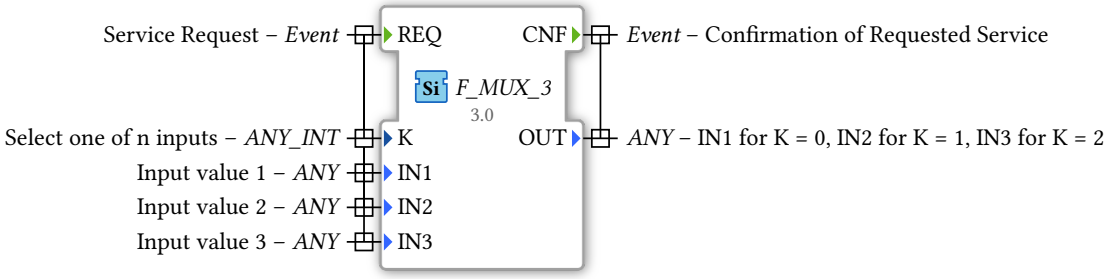
Conclusion
The F_MUX_2 is an efficient and reliable multiplexer for applications requiring selection between two input signals. Its simple structure and use of generic data types make it a versatile solution for various control tasks.

F_MUX_3

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_MUX_3` is a multiplexer that selects three input signals based on a control signal and forwards them to the output. It is part of the IEC 61131-3 standard library and is used for signal selection in automation applications.



Interface Structure

Event Inputs

- **REQ**: Service Request – triggers the selection and forwarding of the corresponding input signal. Linked to the data inputs `IN1`, `IN2`, `IN3`, and `K`.

Event Outputs

- **CNF**: Confirmation of Requested Service – confirms the successful selection and forwarding of the signal. Linked to the data output `OUT`.

Data Inputs

- **K** (ANY_INT): Control signal indicating which input is selected (0 for `IN1`, 1 for `IN2`, 2 for `IN3`).
- **IN1** (ANY): Input value 1.
- **IN2** (ANY): Input value 2.
- **IN3** (ANY): Input value 3.

Data Outputs

- **OUT** (ANY): Output value corresponding to the selected input (`IN1` for `K = 0`, `IN2` for `K = 1`, `IN3` for `K = 2`).

Adapters

No adapters included.

Functionality

Upon a `REQ` event, the function block evaluates the value of `K` and forwards the corresponding input (`IN1`, `IN2`, or `IN3`) to the output `OUT`. The `CNF` event is then triggered to confirm the successful operation.

Technical Specifications

- Supports any data type (`ANY`) for the inputs and output.
- The control input `K` must be an integer type (`ANY_INT`).
- Initial values for the inputs are not predefined.

State Overview

The function block has no internal states. Selection and routing occur immediately upon a `REQ` event.

Application Scenarios

- Signal routing in control applications.
- Selection between different sensor data.
- Dynamic configuration of signal paths.

⚖️ Comparison with Similar Function Blocks

- `F_MUX_2`: A simpler multiplexer with only two inputs.
- `F_SEL`: A more general selector that may support more complex selection criteria.

🛠️ Related Exercises

- Exercise_090a2

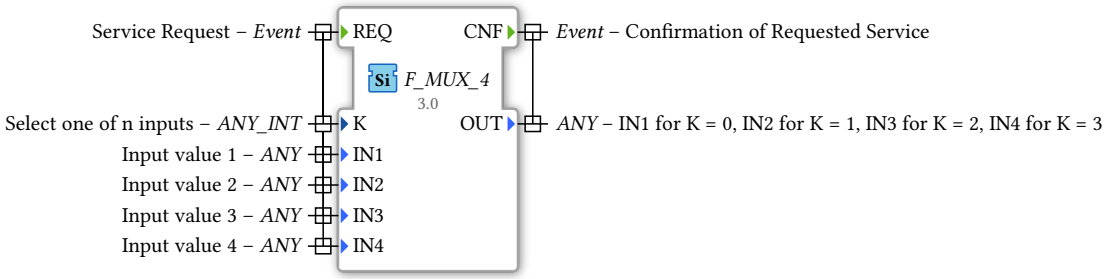
Conclusion

The `F_MUX_3` is a simple and efficient multiplexer for applications where selection between three signals is required. Its flexibility regarding data types makes it versatile.

F_MUX_4

• – ◦ • – ◦ • – ◦ •

The function block `F_MUX_4` is a four-input multiplexer that selects one of four input values based on a control signal and outputs it. It is part of the IEC 61131-3 standard library and is used for selection operations in control applications.



- `REQ`: Service request. Triggers the selection and output of the corresponding input value. Linked to the data inputs `IN1`, `IN2`, `IN3`, `IN4`, and `K`.
- `CNF`: Confirmation of the requested service. Triggered after the output value is set. Linked to the data output `OUT`.
- `K` (`ANY_INT`): Control signal that determines the input to be selected.
- `CNF`: Confirmation of the requested service. Triggered after the output value is set.
- `K` (`ANY_INT`): Control signal that determines the input to be selected ...### **Event Outputs**
- `K = 1`: Selects `IN2`
- `K = 2`: Selects `IN3`
- `K = 3`: Selects `IN4`
- `IN1` (`ANY`): Input value 1
- `IN2` (`ANY`): Input value 2
- `IN3` (`ANY`): Input value 3.
- `IN4` (`ANY`): Input value 4.
- `OUT` (`ANY`): Output value corresponding to the input selected by `K`.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters available.

Functionality

Upon receiving the event REQ, the function block evaluates the value of K and outputs the corresponding input value (IN1 to IN4) at output OUT. The event CNF is then triggered to confirm successful selection and output.

- Supports any data type (ANY) for the inputs and output.
- The control input K must be an integer value (ANY_INT).
- The initial values of the inputs are empty; no default values are provided.

1. **Idle State**: Waits for the event REQ.
2. **Selection State**: Evaluates K and selects the corresponding input.
3. **Output State**: Sets OUT to the selected value and triggers CNF.

- Selection between different sensor data based on control logic.
- Switching between different operating modes in a controller.
- Dynamic selection of data sources depending on external conditions.
- F_MUX_2: Simple multiplexer with only two inputs.
- F_MUX_3: Three-input multiplexer.
- F_MUX_4 offers an extension to four inputs, providing greater flexibility in selection.

The F_MUX_4 is a versatile and easy-to-use multiplexer, ideally suited for applications requiring selection between four different input values. Its support for arbitrary data types and clear event-driven control make it a valuable component in the IEC 61131-3 standard library.

Technical Features

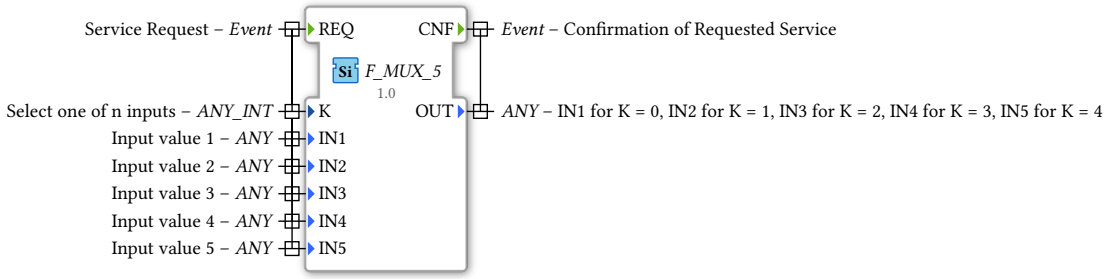
State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

F_MUX_5



• – ◦ • – ◦ • – ◦ •

Introduction

F_MUX_5 is the generic 5-input variant of the standard multiplexer block F_MUX (cf. F_MUX_2/F_MUX_3/F_MUX_4). Based on the selector K, it picks one of 5 input values (IN1, IN2, IN3, IN4, IN5) and outputs it via OUT.

Interface Structure

Event Inputs

- **REQ**: Triggers the selection, carries K and IN1, IN2, IN3, IN4, IN5.

Event Outputs

- **CNF**: Confirms completion, carries OUT.

Data Inputs

- **K** (ANY_INT): Selects one of the 5 inputs (0-based).

- IN1, IN2, IN3, IN4, IN5 (ANY): The 5 candidate input values.

Data Outputs

- **OUT** (ANY): The selected input value: IN1 when K = 0, IN2 when K = 1, IN3 when K = 2, IN4 when K = 3, IN5 when K = 4.

Functionality

On REQ, depending on the value of K, the corresponding input (IN1, IN2, IN3, IN4, IN5) is passed through to OUT: IN1 when K = 0, IN2 when K = 1, IN3 when K = 2, IN4 when K = 3, IN5 when K = 4. CNF is then triggered.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_F_MUX', the same C++ base as F_MUX_2/F_MUX_3/F_MUX_4; only the number of inputs differs per instantiation.
- **ANY data type**: Both inputs IN1…IN5 and OUT are generically typed and adapt to the connected data type when wired.
- **0-based selector**: K = 0 selects IN1, K = 4 selects IN5.

State Overview

Stateless: every REQ immediately results in the selection and CNF.

Application Scenarios

- **Multi-way selection with more than 4 sources**: Wherever a selection is needed among more than the 4 inputs supported by F_MUX_4, without cascading multiple F_MUX blocks.
- **Signal switching**: Selecting one of several sensor or source signals based on a control value K.

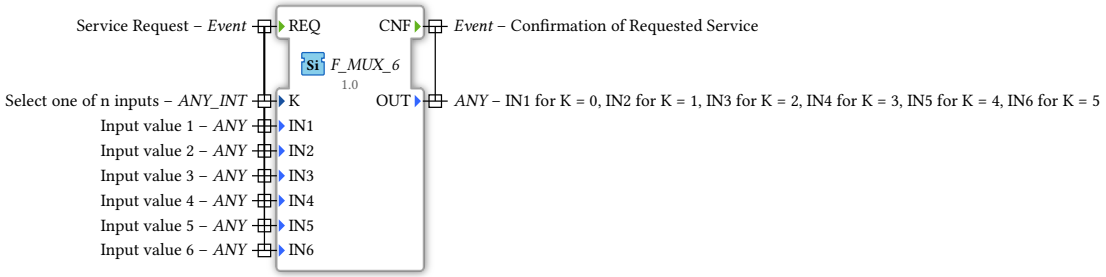
Comparison with similar function blocks

- **F_MUX_4**: the same generic implementation with 4 instead of 5 inputs.
- **F_SEL**: binary selection between exactly two values based on a BOOL selector, functionally a special case of F_MUX_2.

Conclusion

F_MUX_5 extends the multiplexer principle of F_MUX_2/F_MUX_3/F_MUX_4 to 5 selectable inputs and is the generic, 5-input-instantiated variant of the same underlying implementation.

F_MUX_6



• – ○ • – ○ • – ○ •

Introduction

F_MUX_6 is the generic 6-input variant of the standard multiplexer block F_MUX (cf. F_MUX_2/F_MUX_3/F_MUX_4). Based on the selector K, it picks one of 6 input values (IN1, IN2, IN3, IN4, IN5, IN6) and outputs it via OUT.

Interface Structure

Event Inputs

- **REQ**: Triggers the selection, carries K and IN1, IN2, IN3, IN4, IN5, IN6.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **K** (ANY_INT): Selects one of the 6 inputs (`0`-based).
- `IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6` (ANY): The 6 candidate input values.

Data Outputs

- **OUT** (ANY): The selected input value: `IN1` when `K = 0`, `IN2` when `K = 1`, `IN3` when `K = 2`, `IN4` when `K = 3`, `IN5` when `K = 4`, `IN6` when `K = 5`.

Functionality

On `REQ`, depending on the value of `K`, the corresponding input (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`) is passed through to `OUT`: `IN1` when `K = 0`, `IN2` when `K = 1`, `IN3` when `K = 2`, `IN4` when `K = 3`, `IN5` when `K = 4`, `IN6` when `K = 5`. `CNF` is then triggered.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_F_MUX'`, the same C++ base as `F_MUX_2`/`F_MUX_3`/`F_MUX_4`; only the number of inputs differs per instantiation.
- **`ANY` data type**: Both inputs `IN1`…`IN6` and `OUT` are generically typed and adapt to the connected data type when wired.
- **`0`-based selector**: `K = 0` selects `IN1`, `K = 5` selects `IN6`.

State Overview

Stateless: every `REQ` immediately results in the selection and `CNF`.

Application Scenarios

- **Multi-way selection with more than 4 sources**: Wherever a selection is needed among more than the 4 inputs supported by `F_MUX_4`, without cascading multiple `F_MUX` blocks.
- **Signal switching**: Selecting one of several sensor or source signals based on a control value `K`.

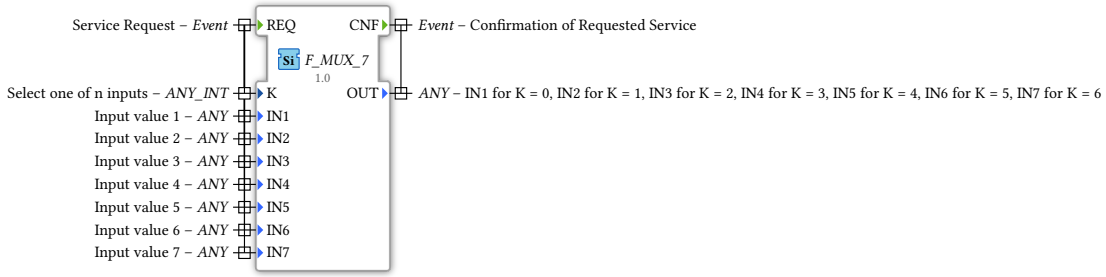
Comparison with similar function blocks

- **`F_MUX_5`**: the same generic implementation with 5 instead of 6 inputs.
- **`F_SEL`**: binary selection between exactly two values based on a `BOOL` selector, functionally a special case of `F_MUX_2`.

Conclusion

`F_MUX_6` extends the multiplexer principle of `F_MUX_2`/`F_MUX_3`/`F_MUX_4` to 6 selectable inputs and is the generic, 6-input-instantiated variant of the same underlying implementation.

F_MUX_7



• – ○ • – ○ • – ○ •

Introduction

`F_MUX_7` is the generic 7-input variant of the standard multiplexer block `F_MUX` (cf. `F_MUX_2`/`F_MUX_3`/`F_MUX_4`). Based on the selector `K`, it picks one of 7 input values (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`) and outputs it via `OUT`.

Interface Structure

Event Inputs

- **REQ**: Triggers the selection, carries `K` and `IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **K** (ANY_INT): Selects one of the 7 inputs (`0`-based).
- `IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7` (ANY): The 7 candidate input values.

Data Outputs

- **OUT** (ANY): The selected input value: `IN1` when `K = 0`, `IN2` when `K = 1`, `IN3` when `K = 2`, `IN4` when `K = 3`, `IN5` when `K = 4`, `IN6` when `K = 5`, `IN7` when `K = 6`.

Functionality

On `REQ`, depending on the value of `K`, the corresponding input (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`) is passed through to `OUT`: `IN1` when `K = 0`, `IN2` when `K = 1`, `IN3` when `K = 2`, `IN4` when `K = 3`, `IN5` when `K = 4`, `IN6` when `K = 5`, `IN7` when `K = 6`. `CNF` is then triggered.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_F_MUX'`, the same C++ base as `F_MUX_2`/`F_MUX_3`/`F_MUX_4`; only the number of inputs differs per instantiation.
- **`ANY` data type**: Both inputs `IN1`…`IN7` and `OUT` are generically typed and adapt to the connected data type when wired.
- **`0`-based selector**: `K = 0` selects `IN1`, `K = 6` selects `IN7`.

State Overview

Stateless: every `REQ` immediately results in the selection and `CNF`.

Application Scenarios

- **Multi-way selection with more than 4 sources**: Wherever a selection is needed among more than the 4 inputs supported by `F_MUX_4`, without cascading multiple `F_MUX` blocks.
- **Signal switching**: Selecting one of several sensor or source signals based on a control value `K`.

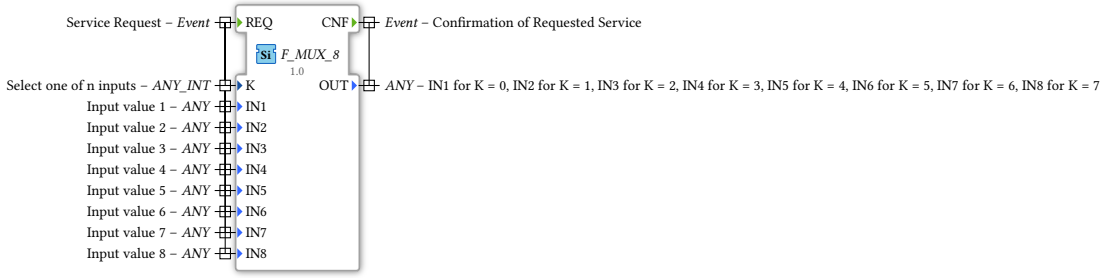
Comparison with similar function blocks

- **`F_MUX_6`**: the same generic implementation with 6 instead of 7 inputs.
- **`F_SEL`**: binary selection between exactly two values based on a `BOOL` selector, functionally a special case of `F_MUX_2`.

Conclusion

`F_MUX_7` extends the multiplexer principle of `F_MUX_2`/`F_MUX_3`/`F_MUX_4` to 7 selectable inputs and is the generic, 7-input-instantiated variant of the same underlying implementation.

F_MUX_8



• – ○ • – ○ • – ○ •

Introduction

F_MUX_8 is the generic 8-input variant of the standard multiplexer block F_MUX (cf. F_MUX_2/F_MUX_3/F_MUX_4). Based on the selector K, it picks one of 8 input values (IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8) and outputs it via OUT.

Interface Structure

Event Inputs

- **REQ**: Triggers the selection, carries K and IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8.

Event Outputs

- **CNF**: Confirms completion, carries OUT.

Data Inputs

- **K** (ANY_INT): Selects one of the 8 inputs (0-based).
- IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8 (ANY): The 8 candidate input values.

Data Outputs

- **OUT** (ANY): The selected input value: IN1 when K = 0, IN2 when K = 1, IN3 when K = 2, IN4 when K = 3, IN5 when K = 4, IN6 when K = 5, IN7 when K = 6, IN8 when K = 7.

Functionality

On REQ, depending on the value of K, the corresponding input (IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8) is passed through to OUT: IN1 when K = 0, IN2 when K = 1, IN3 when K = 2, IN4 when K = 3, IN5 when K = 4, IN6 when K = 5, IN7 when K = 6, IN8 when K = 7. CNF is then triggered.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_F_MUX', the same C++ base as F_MUX_2/F_MUX_3/F_MUX_4; only the number of inputs differs per instantiation.
- **ANY data type**: Both inputs IN1…IN8 and OUT are generically typed and adapt to the connected data type when wired.
- **0-based selector**: K = 0 selects IN1, K = 7 selects IN8.

State Overview

Stateless: every REQ immediately results in the selection and CNF.

Application Scenarios

- **Multi-way selection with more than 4 sources**: Wherever a selection is needed among more than the 4 inputs supported by F_MUX_4, without cascading multiple F_MUX blocks.
- **Signal switching**: Selecting one of several sensor or source signals based on a control value K.

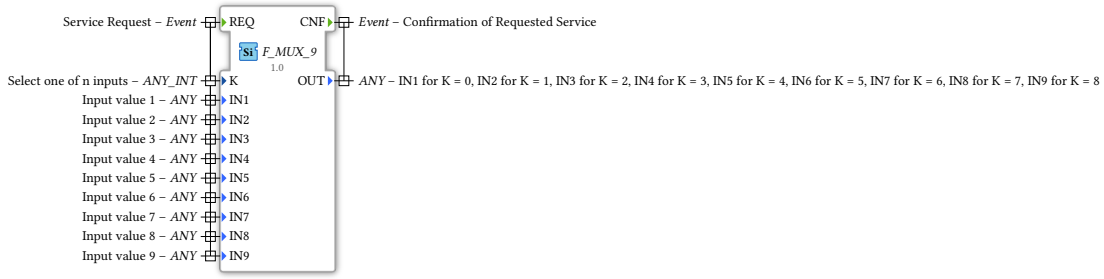
Comparison with similar function blocks

- **F_MUX_7**: the same generic implementation with 7 instead of 8 inputs.
- **F_SEL**: binary selection between exactly two values based on a BOOL selector, functionally a special case of F_MUX_2.

Conclusion

F_MUX_8 extends the multiplexer principle of F_MUX_2/F_MUX_3/F_MUX_4 to 8 selectable inputs and is the generic, 8-input-instantiated variant of the same underlying implementation.

F_MUX_9



• – ○ • – ○ • – ○ •

Introduction

`F_MUX_9` is the generic 9-input variant of the standard multiplexer block `F_MUX` (cf. `F_MUX_2`/`F_MUX_3`/`F_MUX_4`). Based on the selector `K`, it picks one of 9 input values (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`, `IN8`, `IN9`) and outputs it via `OUT`.

Interface Structure

Event Inputs

- **REQ**: Triggers the selection, carries `K` and `IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`, `IN8`, `IN9`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **K** (ANY_INT): Selects one of the 9 inputs (`0`-based).
- `IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`, `IN8`, `IN9` (ANY): The 9 candidate input values.

Data Outputs

- **OUT** (ANY): The selected input value: `IN1` when `K = 0`, `IN2` when `K = 1`, `IN3` when `K = 2`, `IN4` when `K = 3`, `IN5` when `K = 4`, `IN6` when `K = 5`, `IN7` when `K = 6`, `IN8` when `K = 7`, `IN9` when `K = 8`.

Functionality

On `REQ`, depending on the value of `K`, the corresponding input (`IN1`, `IN2`, `IN3`, `IN4`, `IN5`, `IN6`, `IN7`, `IN8`, `IN9`) is passed through to `OUT`: `IN1` when `K = 0`, `IN2` when `K = 1`, `IN3` when `K = 2`, `IN4` when `K = 3`, `IN5` when `K = 4`, `IN6` when `K = 5`, `IN7` when `K = 6`, `IN8` when `K = 7`, `IN9` when `K = 8`. `CNF` is then triggered.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_F_MUX'`, the same C++ base as `F_MUX_2`/`F_MUX_3`/`F_MUX_4`; only the number of inputs differs per instantiation.
- **`ANY` data type**: Both inputs `IN1`…`IN9` and `OUT` are generically typed and adapt to the connected data type when wired.
- **`0`-based selector**: `K = 0` selects `IN1`, `K = 8` selects `IN9`.

State Overview

Stateless: every `REQ` immediately results in the selection and `CNF`.

Application Scenarios

- **Multi-way selection with more than 4 sources**: Wherever a selection is needed among more than the 4 inputs supported by `F_MUX_4`, without cascading multiple `F_MUX` blocks.
- **Signal switching**: Selecting one of several sensor or source signals based on a control value `K`.

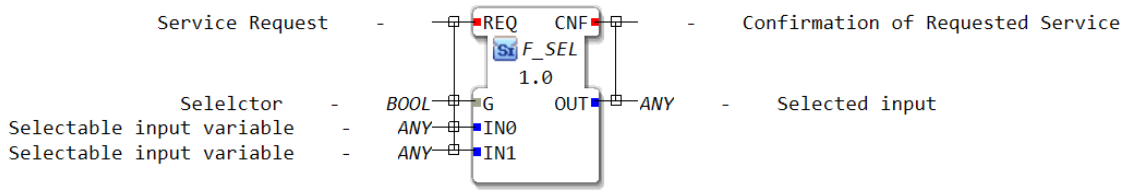
Comparison with similar function blocks

- **`F_MUX_8`**: the same generic implementation with 8 instead of 9 inputs.
- **`F_SEL`**: binary selection between exactly two values based on a `BOOL` selector, functionally a special case of `F_MUX_2`.

Conclusion

`F_MUX_9` extends the multiplexer principle of `F_MUX_2`/`F_MUX_3`/`F_MUX_4` to 9 selectable inputs and is the generic, 9-input-instantiated variant of the same underlying implementation.

F_SEL

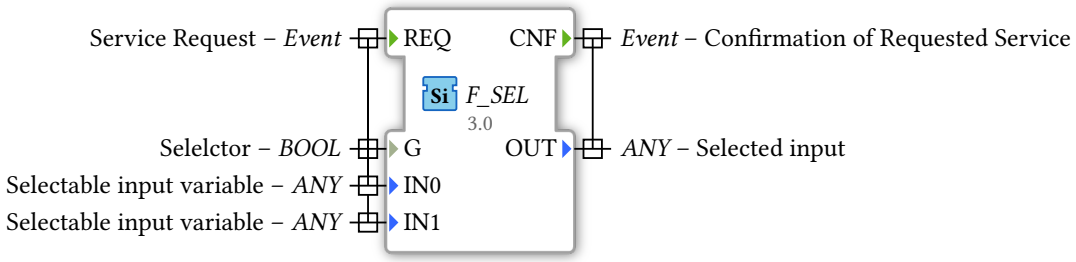


https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_operator_sel.html

• – ◦ • – ◦ • – ◦ •

Introduction

The F_SEL function block is a binary selector according to the IEC 61131-3 standard. It enables selection between two input values based on a control signal. The block is used for conditional data selection in control applications.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the selection operation

Event Outputs

- **CNF**: Confirmation of the requested service operation

Data Inputs

- **G**: Selector (BOOL) - Control signal for selection
- **IN0**: Selectable Input Variable (ANY) - First Input
- **IN1**: Selectable Input Variable (ANY) - Second Input

Data Outputs

- **OUT**: Selected Input (ANY) - Result of the selection

Adapters

No adapter interfaces available.

Functionality

When a REQ event arrives, the F_SEL block evaluates the selector input G:

- If G = FALSE (0): OUT is set to the value of IN0
- If G = TRUE (1): OUT is set to the value of IN1

After the selection operation is complete, the CNF event is output with the calculated output value.

Technical Features

- Uses generic data types (ANY) for the inputs and outputs IN0, IN1, and OUT
- Enables the processing of various data types

- Real-time execution upon event triggering
- Simple and deterministic operation

State Transitions

1. **Wait State**: Block waits for REQ event
2. **Execute State**: Upon REQ event, G is evaluated and data is selected
3. **Output State**: CNF event with result is output, return to wait state

Application Scenarios

- Switching between two sensors or measured values
- Selection between different operating modes
- Conditional data forwarding in process chains
- Implementation of multiplexer functionality

⚖️ Comparison with Similar Function Blocks

Compared to other selection blocks, F_SEL offers:

- Simple binary selection (only two inputs)
- Flexible type support through ANY data type
- Standardized Event Control according to IEC 61131-3

🛠️ Related Exercises

- Exercise_015
- Exercise_016
- Exercise_060

Conclusion

The F_SEL block provides a fundamental and versatile selection function for control applications. Its simple structure and flexible type support make it a valuable component for various selection and switching tasks in industrial automation systems.

2.11.4.12 Timer

FB_TOF

🎧 Podcast

- E_TOF vs. FB_TOF: The event timer that doesn't tick cyclically – a revolution for automation systems?
- FB_TOF and E_TOF: Delay timers in IEC 61131-3 and 61499

• – ◦ • – ◦ • – ◦ •

The following text was generated by https://chat.deepseek.com/ on February 5, 2025, and has not yet been proofread (HF)

Introduction

Important note: This function block only functions correctly if it is called cyclically.

The FB_TOF function block in IEC 61131-3: An Overview

Overview

IEC 61131-3 is an international standard that governs the programming of programmable logic controllers (PLCs). A central concept in this standard is the function block (FB), which serves as a reusable component for control logic. The **FB_TOF** (Off-Delay Timer) block is an example of such a function block used in IEC 61131-3 to model a delayed shutdown function. This paper describes the structure, behavior, and application of the FB_TOF block using the provided source code. An important aspect to emphasize is that this block only functions correctly if it is called **cyclically**.

Structure of the FB_TOF Block

The FB_TOF block is a standardized function block according to IEC 61131-3. It consists of an interface that defines input and output variables as well as events.

Interface

The interface of the FB_TOF block consists of:

- **Input Event**:
- **REQ (Request)**: This event triggers the execution of the block. It is linked to the input variables **IN** and **PT**.
- **Output Event**:
- **CNF (Confirmation)**: This event is triggered when the execution of the block is complete. It is linked to the output variables **Q** and **ET**.
- **Input Variables**:
- **IN (Input)**: This is a Boolean variable that controls the state of the timer. If **IN** is set to **TRUE**, the timer starts. If **IN** is set to **FALSE**, the delay time begins.
- **PT (Process Time)**: This is a time variable (TIME) that defines the delay time after which the output **Q** is set to **FALSE**.
- **Output Variables**:
- **Q (Output)**: This is a Boolean variable that indicates the timer's state. It remains **TRUE** until the delay time **PT** has elapsed.
- **ET (Elapsed Time)**: This is a time variable (TIME) that indicates the elapsed time since the delay started.

Behavior of the FB_TOF Block

The FB_TOF block behaves like an off-delay timer, implementing a delayed shutdown function. The block's behavior can be described as follows:

1. **Timer Start**: When the **REQ** event occurs and the input variable **IN** is set to **TRUE**, the timer starts. The output **Q** is set to **TRUE**, and the elapsed time **ET** begins to count.
2. **Delay Time**: When the input variable **IN** is set to **FALSE**, the delay time **PT** begins. The output **Q** remains **TRUE** until the delay time has elapsed.
3. **Delay Time Expiration**: Once the delay time **PT** has elapsed, the output **Q** is set to **FALSE**, and the **CNF** event is triggered to signal the end of the delay.

Cyclic Calling of the FB_TOF Block

A critical aspect of the FB_TOF block is that it must be called **cyclically** to function correctly. This means that the block must be executed in every cycle of the PLC program to update the timer correctly. If the function block is not called cyclically, the elapsed time **ET** cannot be calculated correctly, and the output **Q** might not be set to **FALSE** in time.

Cyclical calling is necessary because the function block must update the timer's state (i.e., the elapsed time **ET**) in each cycle. If the function block is not called regularly, the timer cannot function correctly, and the delay time **PT** might not be adhered to.

Application Examples

The FB_TOF function block can be used in various industrial control applications, especially in scenarios where a delayed shutdown function is required. Some examples are:

- **Machine Control**: The function block can be used to ensure that a machine continues to run for a specific period of time after being switched off, for example, to complete processes or to implement safety precautions.
- **Lighting Control**: This function block can be used in lighting systems to keep the light on for a specific period of time after the switch is turned off.
- **Valve Control**: This function block can be used to keep a valve open for a specific period of time after it has closed, for example, to release pressure.

🛠️ Related Exercises

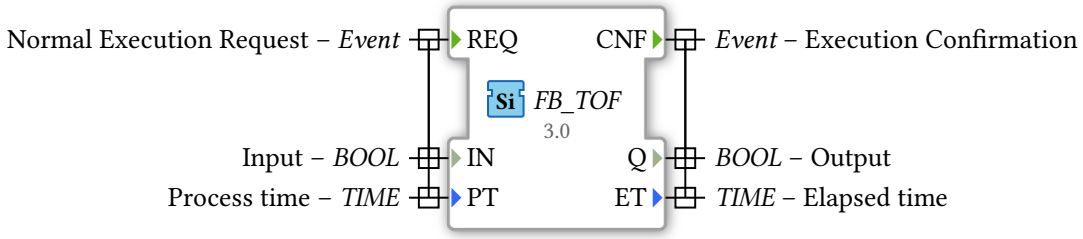
- Exercise_020e2
- Exercise_020e2_AX

Conclusion

The FB_TOF function block is a simple yet powerful function block in IEC 61131-3 that implements a delayed shutdown function. Its clear structure and ease of use make it ideal for applications requiring

time-delayed control. An important aspect to consider is that the function block must be called **cyclically** to function correctly. Only through regular calls in each PLC cycle can the timer accurately calculate the elapsed time and maintain the delay time.

The provided source code demonstrates how the FB_TOF function block can be implemented in practice and offers a solid foundation for developing similar function blocks in industrial control systems. The cyclical calling is a crucial factor for the correct operation of the function block.



⚖️ Comparison of E_TOF and FB_TOF

Chapter 1: The FB_TOF Function Block

The **FB_TOF** (Off-Delay Timer) is a standardized function block according to IEC 61131-3 that implements a delayed shutdown function. A key feature of this function block is the **ET** (Elapsed Time) output, which displays the elapsed time since the delay started. This output is particularly useful for monitoring the delay time progress and for use in control logic.

``A critical aspect of the FB_TOF block, however, is that it **must be called cyclically** to function correctly. This means that the block must be executed in every cycle of the PLC program to correctly calculate the elapsed time **ET** and adhere to the delay time **PT**. If the block is not called regularly, the timer cannot operate correctly, and the delay time may not be adhered to. This dependency on cyclic calling makes the FB_TOF block ideal for traditional PLC systems that operate in fixed cycles.

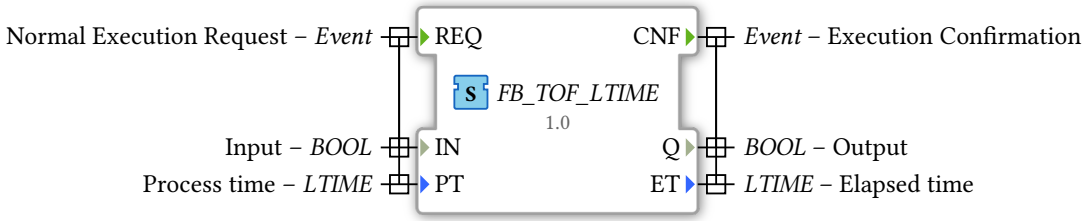
Chapter 2: The E_TOF Block

The **E_TOF** (Event-driven Off-Delay Timer) is a function block according to IEC 61499 that also implements a delayed shutdown function. Unlike the FB_TOF, the E_TOF **does not have an output ET** that displays the elapsed time. Instead, the E_TOF function block is internally based on the **E_DELAY** block, which is specified in IEC 61499 and provides an event-driven delay function.

A key advantage of the E_TOF function block is that it **does not need to be called cyclically** to function correctly. Because the internal **E_DELAY** block operates independently of the cyclic call of the parent block, the E_TOF can also be used in systems that do not operate cyclically, such as event-driven or distributed control systems. This makes the E_TOF function block particularly flexible and suitable for modern, distributed control architectures where cyclic calls are not always guaranteed.

In summary, the FB_TOF function block enables detailed monitoring of elapsed time through its **ET** output, but requires a cyclic call. The E_TOF, on the other hand, does without the output **ET**, but offers the flexibility to work independently of cyclic calls, making it ideal for event-driven systems.

FB_TOF_LTIME



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_TOF_LTIME` is the `LTIME` variant (64-bit duration, nanosecond resolution) of the standard timer block `FB_TOF` (off-delay). It behaves functionally identically to `FB_TOF`, but uses data type `LTIME` instead of `TIME` for `PT/ET`.

Interface Structure

Event Inputs

- **REQ**: Triggers execution, linked with `IN` and `PT`.

Event Outputs

- **CNF**: Confirms execution, linked with `Q` and `ET`.

Data Inputs

- **IN** (BOOL): Input signal.
- **PT** (LTIME): Preset time.

Data Outputs

- **Q** (BOOL): Output signal, see Functionality.
- **ET** (LTIME): Elapsed time since the timing started.

Functionality

Q becomes TRUE immediately when IN switches to TRUE. When IN switches to FALSE, Q stays TRUE for the duration PT before falling back to FALSE. Internally, the block (analogous to `FB_TOF`) holds the internal helper variables `MEM` and `StartTime` to track edge transitions of `IN` and the start time of the measurement — at `LTIME` resolution instead of `TIME`.

Technical Features

- **`LTIME` instead of `TIME`**: Provides 64-bit nanosecond resolution instead of the 32-bit millisecond resolution of `TIME`, relevant for very long or very precise timing.
- **Functionally identical to `FB_TOF`**: Same timing behaviour, only the data type of `PT`/`ET` differs.

State Overview

See `FB_TOF` — identical state behaviour, just with `LTIME` values instead of `TIME`.

Application Scenarios

- **Timing with very high precision or very long duration**, where the millisecond resolution or value range of `TIME` is insufficient.
- **Consistent `LTIME` usage** in applications that work with `LTIME` throughout, to avoid data type conversions.

Comparison with similar function blocks

- **`FB_TOF`**: the same functionality with `TIME` instead of `LTIME`.

Conclusion

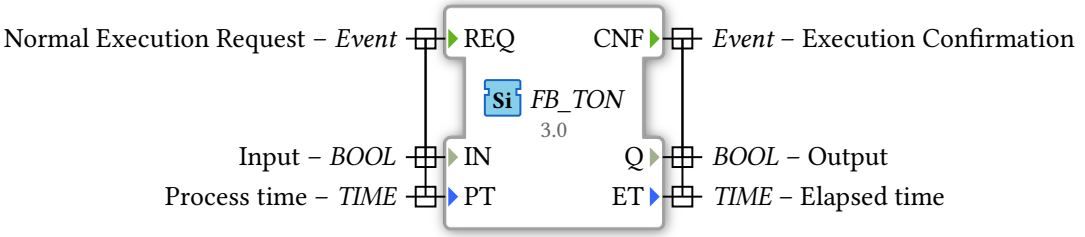
`FB_TOF_LTIME` provides the same off-delay functionality as `FB_TOF`, but with `LTIME` resolution, and is suitable for applications working with high-resolution time values throughout.

FB_TON

• – ◦ • – ◦ • – ◦ •

Introduction

Important note: This function block only functions correctly if it is called cyclically. The FB_TON is a standardized timer function block according to IEC 61131-3 that implements an on-delay. This function block is used to implement time-controlled delays in control programs.



Interface Structure

Event Inputs
- **REQ**: Normal Execution Request - Starts time monitoring when an IN signal is active

Event Outputs
- **CNF**: Execution Confirmation - Confirms completion of time monitoring

Data Inputs
- **IN**: Input - Boolean input that starts the timer (TRUE = timer is running)
- **PT**: Process Time - Time value that defines the delay duration

Data Outputs
- **Q**: Output - Boolean output that becomes TRUE after the delay time has elapsed
- **ET**: Elapsed Time - Displays the elapsed time

Adapters
No adapter interfaces are available.

Functionality
The FB_TON starts the time measurement when the IN input changes from FALSE to TRUE. Once the specified time PT has elapsed, output Q is set to TRUE. Output ET continuously displays the elapsed time. If IN is set to FALSE during the time measurement, the timer is reset and Q remains FALSE.

Technical Features
- Time values are processed in the TIME data type
- The timer is immediately reset when IN=FALSE
- ET always displays the current elapsed time, even if the timer is still running
- Implementation compliant with the IEC 61131-3 standard

State Transitions
1. **Inactive**: IN = FALSE, Q = FALSE, ET = 0
2. **Active (Timer running)**: IN = TRUE, Q = FALSE, ET counts up
3. **Time expired**: IN = TRUE, Q = TRUE, ET = PT

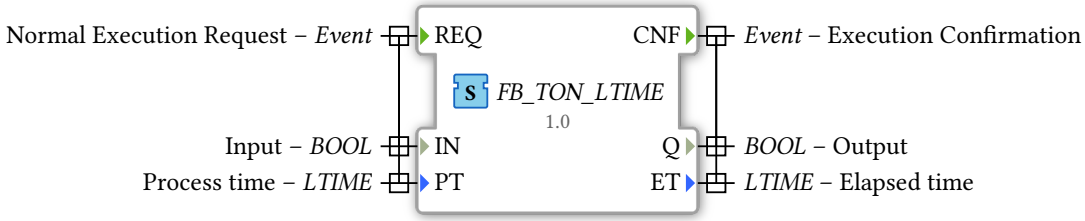
Application Scenarios
- Delayed switching operations in machine controls
- Safety time monitoring
- Process controls with defined waiting times
- Delayed alarm triggers

⚖️ Comparison with similar function blocks
- **FB_TOF**: Off-delay instead of on-delay
- **FB_TP**: Pulse timer with fixed pulse duration
- **FB_TON_EXT**: Extended timer variants with additional functions

🛠️ Related Exercises
- Exercise_020c3
- Exercise_020c3_AX

Conclusion
The FB_TON is a basic and robust timer module for standard turn-on delays in industrial control systems. Its simple and intuitive operation, as well as its compliance with the IEC 61131-3 standard, make it a reliable choice for time-controlled applications.

FB_TON_LTIME



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_TON_LTIME` is the `LTIME` variant (64-bit duration, nanosecond resolution) of the standard timer block `FB_TON` (on-delay). It behaves functionally identically to `FB_TON`, but uses data type `LTIME` instead of `TIME` for `PT`/`ET`.

Interface Structure

Event Inputs

- **REQ**: Triggers execution, linked with `IN` and `PT`.

Event Outputs

- **CNF**: Confirms execution, linked with `Q` and `ET`.

Data Inputs

- **IN** (BOOL): Input signal.
- **PT** (LTIME): Preset time.

Data Outputs

- **Q** (BOOL): Output signal, see Functionality.
- **ET** (LTIME): Elapsed time since the timing started.

Functionality

Q only becomes TRUE once IN has been continuously TRUE for PT. If IN falls back to FALSE before PT elapses, Q stays FALSE. Internally, the block (analogous to `FB_TON`) holds the internal helper variables `MEM` and `StartTime` to track edge transitions of `IN` and the start time of the measurement — at `LTIME` resolution instead of `TIME`.

Technical Features

- **`LTIME` instead of `TIME`**: Provides 64-bit nanosecond resolution instead of the 32-bit millisecond resolution of `TIME`, relevant for very long or very precise timing.
- **Functionally identical to `FB_TON`**: Same timing behaviour, only the data type of `PT`/`ET` differs.

State Overview

See `FB_TON` — identical state behaviour, just with `LTIME` values instead of `TIME`.

Application Scenarios

- **Timing with very high precision or very long duration**, where the millisecond resolution or value range of `TIME` is insufficient.
- **Consistent `LTIME` usage** in applications that work with `LTIME` throughout, to avoid data type conversions.

Comparison with similar function blocks

- **`FB_TON`**: the same functionality with `TIME` instead of `LTIME`.

Conclusion

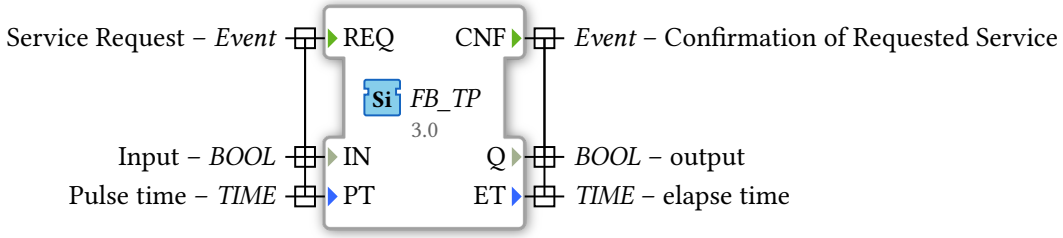
`FB_TON_LTIME` provides the same on-delay functionality as `FB_TON`, but with `LTIME` resolution, and is suitable for applications working with high-resolution time values throughout.

FB_TP

• – ◦ • – ◦ • – ◦ •

Introduction

Important note: This function block only functions correctly if it is called cyclically. The FB_TP (Timer Pulse) is a standardized timer function block according to IEC 61131-3 that provides a pulsed output function. The function block generates a pulse with a defined duration, independent of the duration of the input signal.



Interface Structure

Event Inputs

- **REQ**: Service Request - Starts the timer

Event Outputs

- **CNF**: Confirmation of requested service - Signals the end of the time pulse

Data Inputs

- **IN**: Input signal (BOOL) - Controls the start of the timer
- **PT**: Pulse Time (TIME) - Defines the duration of the output pulse

Data Outputs

- **Q**: Output signal (BOOL) - Outputs the current state of the timer
- **ET**: Elapsed Time (TIME) - Displays the elapsed time of the current pulse

Adapters

No adapter interfaces are available.

Functionality

The FB_TP functions as a monostable multivibrator (one-shot). On a rising edge at the IN input, the timer starts and sets the Q output to TRUE for the duration defined in PT. The elapsed time is continuously output at the ET output. After the set time PT has expired, Q is automatically reset to FALSE, regardless of the state of the IN input.

When a rising edge is detected at the IN input, the timer starts and sets the Q output to TRUE for the duration defined in PT.

Technical Features

- Time measurement is independent of the input signal duration
- ET output displays the current pulse propagation time
- Immediate response to a rising edge at the IN input
- Automatic reset after time elapses

State Transitions

1. **Idle State**: IN = FALSE, Q = FALSE, ET = 0
2. **Start**: Rising edge at IN ⟶ Q = TRUE, ET counting begins
3. **Progression Time**: Q remains TRUE while ET < PT
4. **End**: When ET ≥ PT ⟶ Q = FALSE, ET counting stops

Application Scenarios

- Generation of precisely defined pulses

- Time-controlled sequences
- Delay circuits
- Clock generation for process control
- Safety time monitoring

⚖️ Comparison with similar function blocks

- **FB_TON**: Switch-on delay - Q only becomes active after PT has elapsed
- **FB_TOF**: Switch-off delay - Q remains active for PT after IN reset
- **FB_TP**: Pulse generator - Generates a fixed pulse independent of IN

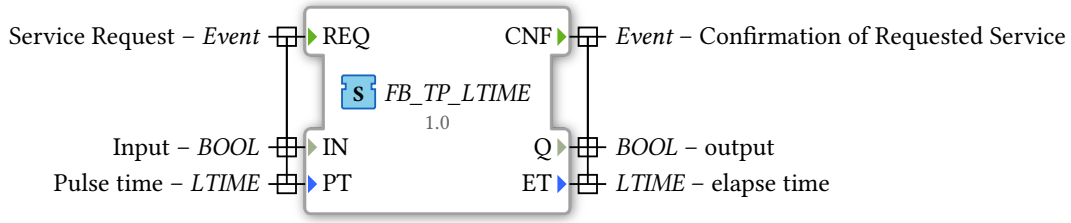
🛠️ Related exercises

- Exercise_020f2
- Exercise_020f2_AX

Conclusion

The FB_TP is an essential component for time-controlled applications in automation technology. Its ease of use and reliable operation make it ideal for applications requiring precise time pulses. The clear separation of the start pulse and pulse duration enables flexible control concepts.

FB_TP_LTIME



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FB_TP_LTIME` is the `LTIME` variant (64-bit duration, nanosecond resolution) of the standard timer block `FB_TP` (pulse). It behaves functionally identically to `FB_TP`, but uses data type `LTIME` instead of `TIME` for `PT`/`ET`.

Interface Structure

Event Inputs

- **REQ**: Triggers execution, linked with `IN` and `PT`.

Event Outputs

- **CNF**: Confirms execution, linked with `Q` and `ET`.

Data Inputs

- **IN** (BOOL): Input signal.
- **PT** (LTIME): Preset time.

Data Outputs

- **Q** (BOOL): Output signal, see Functionality.
- **ET** (LTIME): Elapsed time since the timing started.

Functionality

A rising edge of IN triggers a TRUE pulse of fixed duration PT on Q, regardless of how long IN stays TRUE. Internally, the block (analogous to `FB_TP`) holds the internal helper variables `MEM` and `StartTime` to track edge transitions of `IN` and the start time of the measurement — at `LTIME` resolution instead of `TIME`.

Technical Features

- **`LTIME` instead of `TIME`**: Provides 64-bit nanosecond resolution instead of the 32-bit millisecond resolution of `TIME`, relevant for very long or very precise timing.
- **Functionally identical to `FB_TP`**: Same timing behaviour, only the data type of `PT`/`ET` differs.

State Overview

See `FB_TP` — identical state behaviour, just with `LTIME` values instead of `TIME`.

Application Scenarios

- **Timing with very high precision or very long duration**, where the millisecond resolution or value range of `TIME` is insufficient.
- **Consistent `LTIME` usage** in applications that work with `LTIME` throughout, to avoid data type conversions.

Comparison with similar function blocks

- **`FB_TP`**: the same functionality with `TIME` instead of `LTIME`.

Conclusion

`FB_TP_LTIME` provides the same pulse functionality as `FB_TP`, but with `LTIME` resolution, and is suitable for applications working with high-resolution time values throughout.

2.11.5 🔌 io

2.11.5.1 📺 Video

- Automation of the Future
- The Great Migration
- Digital Decisions
- Ideal Diodes
- The Dual Wi-Fi Solution

2.11.5.2 🎧 Podcast

- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Target Groups, Structure, and Unique Selling Proposition
- Taming the Software Dragon: Industrial Automation and the Future of Production
- Eclipse 4diac: Innovation Engine Research vs. User Needs – What Really Drives Development?
- Taming the Software Dragon: Why IEC 61499 Could Revolutionize Industry

---

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Diode & Semiconductor Basics on ms-muc-docs.de
- 🌐 Smart Switch vs. Relay on ms-muc-docs.de

2.11.5.3 IB

🎧 Podcast

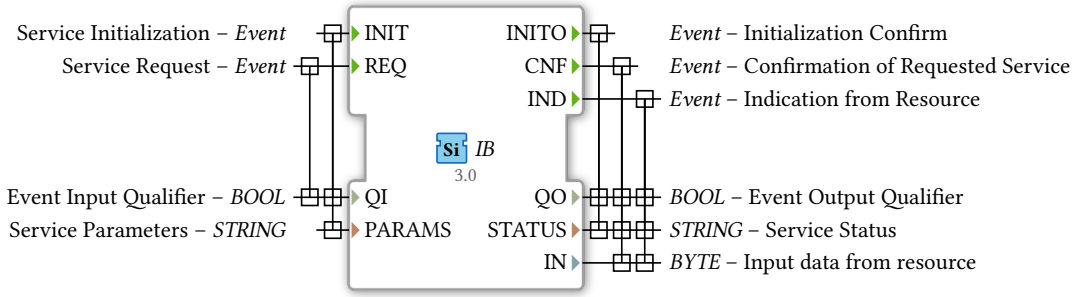
- Eclipse 4diac: Innovation Engine Research vs. User Needs – What Really Drives Development?
- DIN EN 61499-1: The Digital Lego Kit for Flexible Automation and a Smart Future
- DIN EN 61499-1: The Lego Bricks for Flexible and Event-Driven Industrial Controls
- IEC 61499: The DNA of Flexible Automation – Powering Industry 4.0 from Factories to the High Seas
- IEC 61499 vs. 61131: Unpacking the Architectural Revolution of Distributed Control and Industry 4.0

📺 Video

- LogiBUS Project Update

Introduction

The IB (Input Byte) function block is a service interface function block for processing byte input data. It serves as an interface between an IEC 61499 control system and external input devices or resources that supply byte data.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the function block and configures the interface parameters
- **REQ**: Service Request - Requests current data input

Event Outputs

- **INITO**: Initialization Acknowledgement - Confirms successful initialization
- **CNF**: Service Acknowledgement - Confirms the requested service operations
- **IND**: Indication from Resource - Signals incoming data from the external resource

Data Inputs

- **QI**: Event Input Qualifier (BOOL) - Controls the execution of events
- **PARAMS**: Service Parameters (STRING) - Configuration parameters for the interface

Data Outputs

- **QO**: Event Output Qualifier (BOOL) - Status indicator for the output events
- **STATUS**: Service Status (STRING) - Status information about the current operation
- **IN**: Input data from the resource (BYTE) - The received byte data

Adapter

No adapter interfaces are available.

Functionality

The IB function block operates as a bidirectional interface for byte input data. During initialization (INIT), the interface parameters are configured. Data can be explicitly requested via REQ events, while IND events signal asynchronous data inputs. The received byte data is provided via the IN output.

Technical Features

- Supports both request-based (REQ/CNF) and event-based (IND) data acquisition
- BYTE data type for compact 8-bit data processing
- String-based parameter configuration for flexible customization
- Status feedback via SERVICE status output

State Overview

The function block goes through the following main states:

1. **Not Initialized**: Waiting for an INIT event
2. **Initialized**: Ready for data acquisition
3. **Data Request**: Processing REQ events
4. **Data Receive**: Processing incoming IND events

Application Scenarios

- Reading 8-bit sensor data

- Interface to simple digital input devices
- Communication with peripheral devices that provide byte data
- Protocol implementations with byte-oriented data formats

⚖️ Comparison with Similar Function Blocks

Compared to other input blocks:

- IB specialized for the BYTE data type (8-bit)
- Offers both polling (REQ) and interrupt-type (IND) operating modes
- Simpler than more complex protocol blocks, but more specific for byte data

🛠️ Related Exercises

- Exercise_006c
- Exercise_011a

Conclusion

The IB function block offers an efficient and flexible solution for integrating byte-based input data into IEC 61499 control systems. Its dual operating mode (request-based and event-driven) makes it versatile for various applications in industrial automation.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.5.4 ID

🎧 Podcast

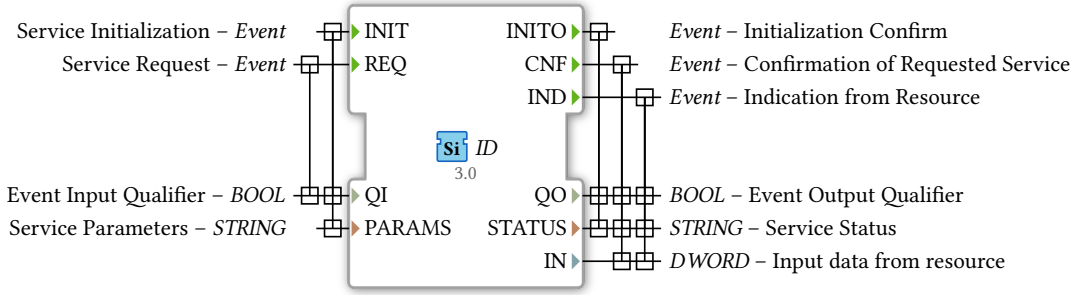
- 4diac IDE: Your “Hello World” of Automation – The Blinking Tutorial Local
- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- From Pyramid to Plug-and-Play: The Rise of Self-Configurable Industrial Automation
- Building Tomorrow’s Factories: Bridging OT and IT with IEC 61499

📺 Video

- Ideal Diodes

Introduction

The ID function block is an input service interface module for double-word (DWORD) input data. It serves as an interface between the control logic and external input devices or resources and enables the querying of 32-bit input data.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the function block and configures the parameters
- **REQ**: Service Request - Triggers a query of the input data

Event Outputs

- **INITO**: Initialization Acknowledgement - Confirms successful initialization
- **CNF**: Confirmation of Requested Service - Confirms a successful service request
- **IND**: Indication from Resource - Signals incoming data from the resource

Data Inputs

- **QI**: Event Input Qualifier (BOOL) - Enables/disables the service
- **PARAMS**: Service Parameters (STRING) - Configuration parameters for the service

Data Outputs

- **QO**: Event Output Qualifier (BOOL) - Status of the Service Output
- **STATUS**: Service status (STRING) - Status information about the service
- **IN**: Input data from the resource (DWORD) - The read 32-bit input data

Adapter

No adapter interfaces available.

Functionality

The ID block functions as a service interface for double-word input. During initialization (INIT), the service parameters are configured. Subsequently, input data can be selectively queried from the connected resource via REQ events. The block returns the read data via the IN outputs along with status information.

Technical Features

- Processes 32-bit data (DWORD)
- Supports both requested (REQ/CNF) and spontaneous (IND) data transmissions
- Flexible parameter configuration via STRING parameters
- Qualifier-based control (QI/QO) for service activation

State Overview

The function block goes through the following main states:

1. **Not Initialized**: Before INIT processing
2. **Initialized**: After successful INIT processing, ready for data queries
3. **Data Query**: During the processing of REQ events
4. **Data Reception**: During spontaneously received data (IND)

Application Scenarios

- Querying 32-bit sensor data
- Reading digital input signals in groups
- Communication with peripheral devices that provide double-word data
- Integration of external measurement systems into 4diac controllers

⚖️ Comparison with Similar Function Blocks

Compared to ID offers simpler input blocks:

- Extended status feedback
- Configurable service parameters
- Support for both operating modes (demand-driven and ad hoc)
- 32-bit data width instead of simple BOOL or BYTE values

🛠️ Related Exercises

- Exercise_011
- Exercise_011a2
- Exercise_012
- Exercise_012a_sub
- Exercise_012b
- Exercise_012c
- Uebung_020c2_sub
- Uebung_028

- Uebung_034
- Uebung_034a1_Q1
- Uebung_034a1_Q2
- Uebung_034a1_Q4
- Uebung_060
- Uebung_103
- Uebung_150
- Exercise_150_AX
- Exercise_151
- Exercise_151_AX
- Exercise_152
- Exercise_153

Conclusion

The ID function block provides a powerful and flexible interface for double-word input data. Its comprehensive status feedback and configurable parameters make it particularly suitable for demanding applications requiring reliable and informative input data processing.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 Diode & Semiconductor Basics on ms-muc-docs.de
- 🌐 Smart Switch vs. Relay on ms-muc-docs.de

2.11.5.5 IE

🎧 Podcast

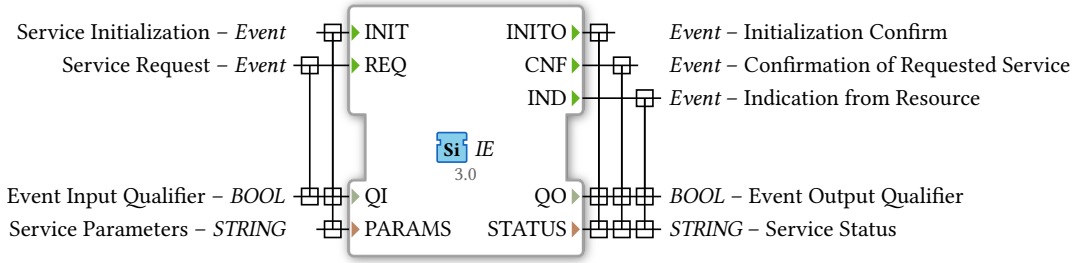
- 4diac IDE: Your “Hello World” of Automation – The Blinking Tutorial Local
- 4diac IDE: Your Open-Source Toolkit for Distributed Industrial Automation according to IEC 61499
- 4diac IDE: How the IEC 61499 Standard is Revolutionizing Industrial Automation
- 4diac Presentation: Target Groups, Structure, and Unique Selling Proposition Sharpening
- Taming the Software Dragon: Industrial Automation and the Future of Production

📺 Video

- The Great Migration
- The Art of Soldering ()
- The World of Standardization (](https://www.youtube.com/watch?v=9phDmkJVaGM))
- Soldering Like a Pro (](https://www.youtube.com/watch?v=8ulMWcxaB-c))
- The Secret of the Field (](https://www.youtube.com/watch?v=MmMrEXum4w4))

Introduction

The IE (Input Event) function block is a service interface function block for processing event input data. It serves as an interface between the control logic and external input devices or signals and enables the initialization, querying, and indication of input events.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the function block with parameters
- **REQ**: Service Request - Triggers a query of the input state

Event Outputs

- **INITO**: Initialization Acknowledgement - Confirms successful initialization
- **CNF**: Service Acknowledgement - Confirms the processed service request
- **IND**: Resource Indication - Signals incoming events from the hardware

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Enables/disables event processing
- **PARAMS** (STRING): Service Parameters - Configuration parameters for the service

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Status of the Event Processing
- **STATUS** (STRING): Service status - Feedback on the current operating state

Adapter

No adapter interfaces are available.

Functionality

The IE function block operates as a bidirectional interface for input events. During initialization (INIT), the service parameters are configured. Targeted queries of the input state can be performed via REQ events. Simultaneously, the block can asynchronously generate IND events when input events are spontaneously detected by the hardware.

Technical Features

- Supports both poll-based (REQ/CNF) and interrupt-based (IND) operating modes
- String-based parameter and status transmission for flexible configuration
- Separate qualifiers for input and output events (QI/QO)
- Robust error handling through STATUS feedback

State Overview

The function block goes through the following main states:

1. **Not Initialized**: Block waits for an INIT event
2. **Initialized**: Block is ready for operation and can process REQ and IND events
3. **Query Active**: Processing a REQ request
4. **Indication Active**: Processing a spontaneous input event

Application Scenarios

- Querying digital inputs (e.g., pushbuttons, switches)
- Monitoring sensor signals
- Interface to external input devices
- Event-driven control applications
- Hardware-level I/O management in automation systems

⚖️ Comparison with Similar building blocks

Compared to simpler input blocks, IE offers extended functionality:

- Compared to pure I/O blocks: Supports both poll-based and event-based queries
- Compared to static input blocks: Dynamic parameterization at runtime
- Enhanced status feedback for improved error diagnosis

🛠️ Related Exercises

- Exercise_004a
- Exercise_004a2
- Exercise_004a2_2
- Exercise_004a2_3
- Exercise_004a2_AX
- Uebung_004a3
- Uebung_004a3_AX
- Uebung_004a4

- Uebung_004a4_AX
- Uebung_004a5
- Uebung_004a5_AX
- Uebung_004a6
- Uebung_004a6_AX
- Uebung_004a7
- Uebung_004a7_AX
- Uebung_004a8
- Uebung_004a8_AX
- Uebung_004a9
- Uebung_004a9_AX
- Uebung_004a_AX
- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_004b_AX
- Uebung_004b_AX_ASR
- Uebung_004b_AX_ASR_X
- Uebung_004c1
- Uebung_004c1_AX
- Uebung_004c2
- Uebung_004c2_AX
- Uebung_004c3
- Uebung_004c3_AX
- Uebung_004c4
- Uebung_004c4_AX
- Uebung_004c5
- Uebung_004c5_AX
- Uebung_006
- Uebung_006_AX
- Uebung_006a
- Uebung_006a2
- Uebung_006a2_AX
- Uebung_006a3
- Uebung_006a3_AX
- Uebung_006a4
- Uebung_006a4_AX
- Uebung_006a_AX
- Uebung_006b
- Uebung_006b_AX
- Uebung_006d
- Uebung_006d_AX
- Uebung_007a1
- Uebung_007a1_AX
- Uebung_007a2
- Uebung_007a2_AX
- Uebung_007a3
- Uebung_007a3_AX
- Uebung_009a
- Uebung_010b2
- Uebung_010b2_AX
- Uebung_010b3
- Uebung_010b3_AX
- Uebung_010b6
- Uebung_010b6_AX
- Uebung_010b7
- Uebung_010b7_AX
- Uebung_010b8
- Uebung_010b8_AX

- Uebung_010b9
- Uebung_010b9_AX
- Uebung_010bA
- Uebung_010bA2
- Uebung_010bA2_AX
- Uebung_010bA3
- Uebung_010bA3_AX
- Uebung_010bA4
- Uebung_010bA4_AX
- Uebung_010bA_AX
- Uebung_013
- Uebung_013_AX
- Uebung_014
- Uebung_015
- Uebung_015a
- Uebung_016
- Uebung_016a
- Uebung_017
- Uebung_018
- Uebung_018a
- Uebung_019
- Uebung_019a
- Uebung_019b
- Uebung_019c
- Uebung_020f3
- Uebung_020f3_AX
- Uebung_020h
- Uebung_020h_AX
- Uebung_020i
- Uebung_020i_AX
- Uebung_021
- Uebung_022
- Uebung_023
- Uebung_024
- Uebung_025
- Uebung_026
- Uebung_031
- Uebung_034b
- Uebung_035
- Uebung_035a
- Uebung_035a1_AX
- Uebung_035a1b_AX
- Uebung_035a2
- Uebung_035a2_AX
- Uebung_035a3
- Uebung_035a3_AX
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- Uebung_038_AX
- Uebung_039
- Uebung_039a
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_040
- Uebung_040_2
- Uebung_040_AX
- Uebung_041

- Uebung_042
- Uebung_043
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_081
- Uebung_082
- Uebung_083
- Uebung_083_AX
- Uebung_084
- Uebung_085
- Uebung_087
- Uebung_087a1
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Uebung_094a
- Uebung_094a_AX
- Uebung_095
- Uebung_095_AX
- Uebung_110
- Uebung_111
- Uebung_124
- Uebung_127
- Uebung_128
- Uebung_128b
- Uebung_132
- Uebung_150_AX
- Uebung_151_AX
- Uebung_152
- Exercise_153
- Exercise_160b
- Exercise_160b_AX
- Exercise_171_AX
- Exercise_179
- Exercise_180

Conclusion

The IE function block provides a flexible and robust solution for handling input events in 4diac-based control systems. Its ability to process both synchronous queries and asynchronous indications makes it particularly suitable for applications requiring both reactive and proactive input processing.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.5.6 IL

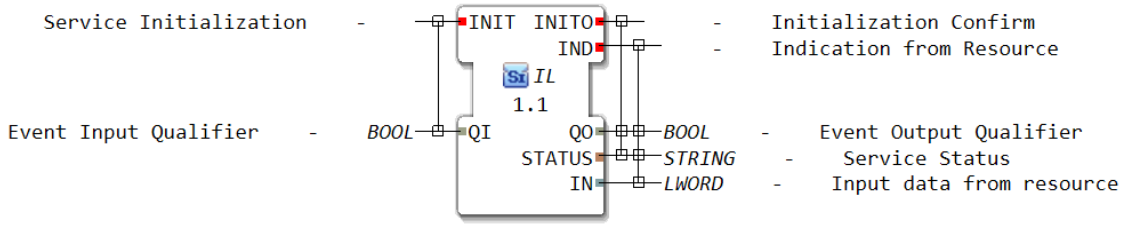
📺 Video

- The silent failure

🎧 Podcast

- 4diac IDE: Your open-source toolkit for distributed industrial automation according to IEC 61499
- Eclipse 4diac 3.0: ST interpreter, FBE, and 7200 commits – The turbocharger for distributed automation
- Eclipse 4diac: Your bicycle for Industry 4.0 – Mastering software complexity and controlling distributed systems
- IEC 61499: The future of industrial automation and distributed systems

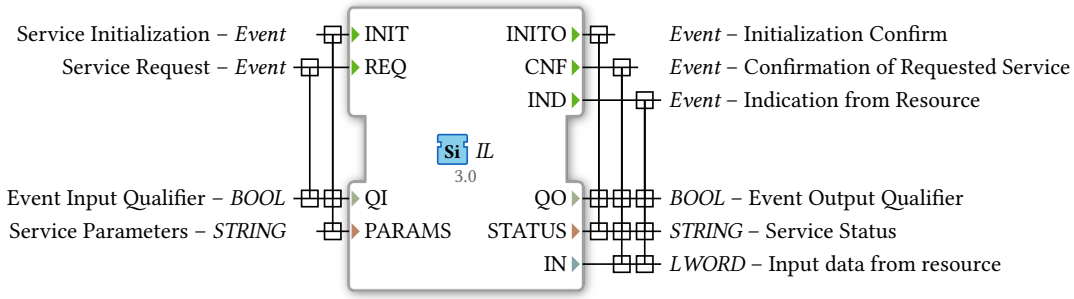
- [Step 2 - 4diac applications] Distribute](https://podcasters.spotify.com/pod/show/eclipse-4diac-de/episodes/Schritt-2—4diac-Anwendungen-verteilen-e36974b)



• – ◦ • – ◦ • – ◦ •

Introduction

The IL (Input Long Word) function block is a Service Interface Function Block for processing long word input data. It serves as an interface between the control logic and external input devices or resources that provide 64-bit data values.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation
- **IND**: Indication from Resource

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Controls the execution of the service
- **PARAMS** (STRING): Service Parameters for Configuration

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Displays the status of the operation
- **STATUS** (STRING): Service Status Information
- **IN** (LWORD): Input Data from the Resource (64-bit value)

Adapters

No adapter interfaces are available.

Functionality

The IL block functions as an input service interface for long word data. Upon receiving the INIT event, the service is initialized and the configuration parameters are passed via PARAMS. The REQ event triggers a read operation, in which data is read from the connected resource. The block supports both requested reads (CNF) and spontaneous data updates (IND) from the resource.

Technical Features

- Processes 64-bit Long Word (LWORD) data types
- Supports both poll-based and event-based data queries
- Flexible parameter configuration via STRING parameters
- Status feedback via STATUS output

State Overview

The block goes through the following states:

1. **Not Initialized**: Before the INIT operation
2. **Initialized**: After successful INIT operation, ready for data queries
3. **Active**: During data querying or when receiving indications

Application Scenarios

- Reading 64-bit sensor data
- Communication with external devices that provide Long Word data
- High-precision data acquisition systems
- Control systems that need to process large numerical values

⚖️ Comparison with Similar Blocks

Compared to other input blocks such as IB (Byte), IW (Word), or ID (Double Word), IL processes the largest data size (64 bits). It thus offers the highest precision for numerical input values in the 4diac I/O library.

Conclusion

The IL function block provides a robust and flexible interface for processing 64-bit input data in IEC 61499 control systems. Its support for various operating modes (polling and event-driven) makes it versatile for use in a wide range of industrial applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

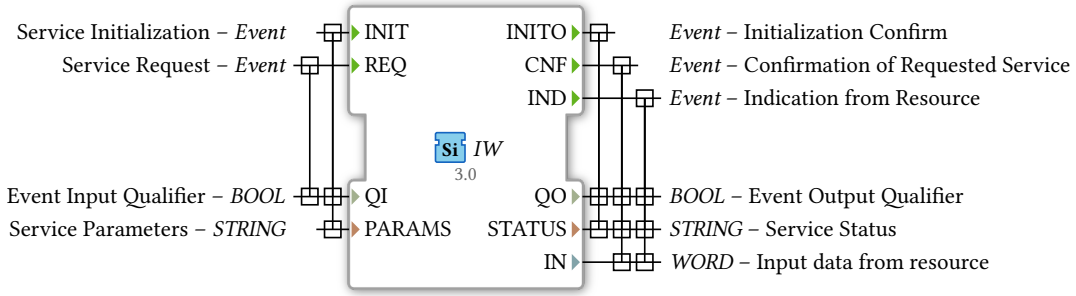
2.11.5.7 IW

🎧 Podcast

- Miniware TS101: The Mobile Soldering All-Rounder – Strengths, Weaknesses, and the USB-C Revolution

Introduction

The IW (Input Word) function block is a service interface function block for processing word input data in 4diac systems. It serves as an interface between the control logic and external input devices or resources that provide word data (16-bit).



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the function block and configures the hardware parameters

- **REQ**: Service Request - Requests a current input from the connected device

Event Outputs

- **INITO**: Initialization Acknowledgement - Confirms completion of initialization
- **CNF**: Requested Service Acknowledgement - Acknowledges a successful REQ request
- **IND**: Resource Indication - Signals incoming data without a prior request

Data Inputs

- **QI**: Event Input Qualifier (BOOL) - Enables/disables processing
- **PARAMS**: Service Parameters (STRING) - Configuration parameters for the hardware interface

Data Outputs

- **QO**: Event Output Qualifier (BOOL) - Processing status
- **STATUS**: Service status (STRING) - Error messages or status information
- **IN**: Input data from the resource (WORD) - The read 16-bit data

Functionality

The IW block operates as a bidirectional interface for Word input data. During initialization (INIT), the hardware parameters are configured. The block can respond to explicit requests (REQ) as well as process asynchronous data indications (IND) from the connected resource. The read Word data is output to the IN output.

Technical Features

- Processes 16-bit WORD data types
- Supports both request-based (REQ/CNF) and event-based (IND) communication
- String-based parameter configuration for flexible hardware connectivity
- Comprehensive status feedback via STATUS output

State Overview

The block goes through the following main states:

- **Not Initialized**: Block waits for an INIT event
- **Initialized**: Block is ready and can process data
- **Data Request**: Processing a REQ request
- **Data Reception**: Processing incoming IND data

Application Scenarios

- Connecting 16-bit input devices (sensors, encoders, switch groups)
- Reading digital input words from process peripherals
- Interface to I/O modules with word-based inputs
- Monitoring status words from distributed systems

⚖️ Comparison with Similar Blocks

Compared to IB (Input Byte) IW processes 16-bit data instead of 8-bit data. Compared to ID (Input Double Word), IW offers a narrower data width. The block's structure is similar to other I/O interface blocks, but it is specifically optimized for WORD data types.

Conclusion

The IW function block provides a robust and flexible interface for processing 16-bit input data in 4diac systems. Its clear event structure and comprehensive status feedback make it a reliable solution for connecting word-based input devices in industrial automation applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

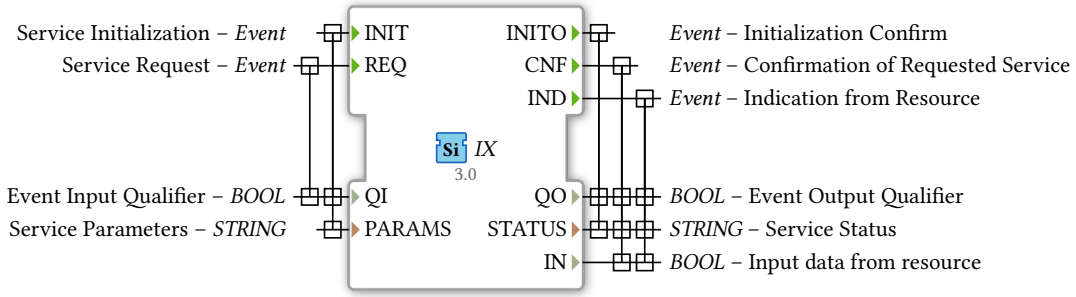
2.11.5.8 IX

🎧 Podcast

- Infineon MOTIX BTM9020/9021EP: Datasheet Analysis for Automotive – Robust Motor Driver with Intelligent Diagnostics (HW vs. SPI)
- Integrated Full Bridge ICs MOTIX™ BTM9020EP

Introduction

The IX function block is a service interface function block for Boolean input data. It serves as an interface for communication with input devices and enables the querying and processing of digital input signals in 4diac systems.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the function block and configures the hardware parameters
- **REQ**: Service Request - Requests a query of the current input value

Event Outputs

- **INITO**: Initialization Acknowledgement - Confirms successful initialization
- **CNF**: Confirmation of Requested Service - Confirms a successful query
- **IND**: Indication from Resource - Signals a change in the input signal's state

Data Inputs

- **QI**: Event Input Qualifier (BOOL) - Controls the activation of the service functionality
- **PARAMS**: Service Parameters (STRING) - Contains configuration parameters for the hardware interface

Data Outputs

- **QO**: Event Output Qualifier (BOOL) - Displays the service execution status
- **STATUS**: Service status (STRING) - Provides status information about the service execution
- **IN**: Input data from the resource (BOOL) - Contains the current value of the digital input

Adapter

No adapter interfaces are available.

Functionality

The IX function block acts as an intermediary between the control logic and physical input devices. During initialization (INIT), the hardware parameters are configured. Subsequently, the current state of the input can be queried via REQ events. The block can process both poll-based queries (REQ/CNF) and event-based notifications (IND) upon state changes.

Technical Features

- Supports both request-based and event-driven operating modes
- Boolean data type for simple digital inputs
- Flexible parameter configuration via STRING parameters
- Comprehensive status feedback for fault diagnosis

State Overview

The function block goes through the following main states:

1. **Not Initialized**: Block is inactive
2. **Initialized**: Block is ready for operation after successful INIT processing
3. **Query Active**: Processing a REQ request
4. **Indication Ready**: Ready for event-driven notifications

Application Scenarios

- Querying digital sensors (limit switches, proximity switches)
- Monitoring push-button inputs
- Reading switch positions
- Digital signal processing in automation systems

⚖️ Comparison with Similar Blocks

Compared to other input blocks, IX offers an enhanced service interface with both poll-based and event-driven operating modes. While simpler input blocks often only provide direct values, IX offers additional status information and error handling.

🛠️ Related exercises

- Uebung_001
- Uebung_001c
- Uebung_002
- Uebung_002a
- Uebung_002a2
- Uebung_002a3
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b2
- Uebung_002b3
- Uebung_003
- Uebung_003a0
- Uebung_003a_sub
- Funk_IX_TO_DataPanel_MI_QX
- IX_TO_Aux_QX
- Uebung_003d
- Uebung_005
- Uebung_006e1
- Uebung_006e2
- Uebung_010
- Uebung_010a
- Uebung_010a2
- Uebung_010a3
- Uebung_010a4
- Uebung_010b1
- Uebung_010c
- Uebung_010c2
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c
- Uebung_020c2
- Uebung_020c3
- Uebung_020d
- Uebung_020e
- Uebung_020e2
- Uebung_020f
- Uebung_020f2
- Uebung_020g

- Uebung_020i
- Uebung_028
- Uebung_029
- Uebung_030
- Uebung_032
- IX_TO_logiBUS_LED_strip_QX
- SoftkeyOverride_TO_QX_BG
- Uebung_039b
- Uebung_049
- Uebung_051
- Uebung_052
- Uebung_053
- Uebung_054
- Uebung_055
- Uebung_056
- Uebung_085
- Uebung_086
- Uebung_087
- Uebung_087a1
- Uebung_087a2
- Uebung_088
- Uebung_089
- Uebung_090a1
- Uebung_090a1_AX
- Uebung_090a2
- Uebung_090a2_AX
- Uebung_094
- Uebung_094a
- Uebung_095
- Uebung_160
- Uebung_160b2
- Uebung_177
- Exercise_178

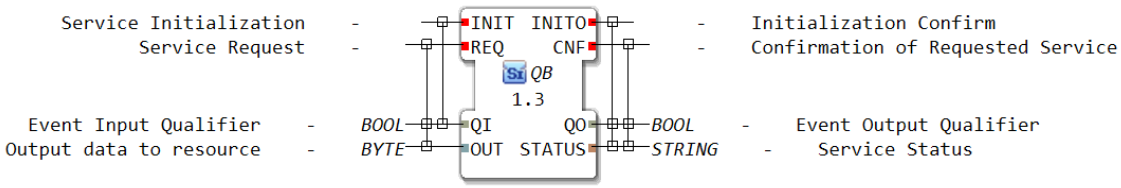
Conclusion

The IX function block provides a robust and flexible solution for integrating Boolean input data into 4diac systems. Its comprehensive error handling and flexible operating modes make it particularly suitable for reliable automation applications with digital input signals.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

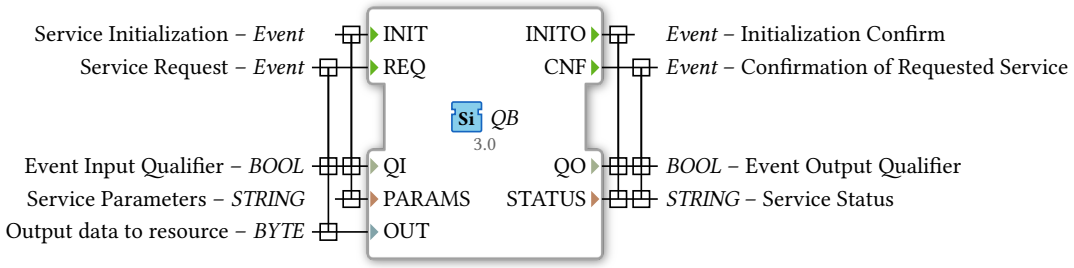
2.11.5.9 QB



• – ◦ • – ◦ • – ◦ •

Introduction

The QB function block is an output service interface for byte output data. It serves as an interface between the control logic and physical output devices or resources, enabling the output of byte data to external hardware.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters for Configuration
- **OUT** (BYTE): Output Data for the Resource

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status Information

Adapters

No adapter interfaces are available.

Functionality

The QB block initializes itself with the corresponding PARAMS via the INIT event. After successful initialization, it confirms this with INITO. Byte data (OUT) can be output to the connected resource via the REQ event. Processing is acknowledged with the CNF event and status information.

Technical Features

- Specialized for byte data output
- Supports configurable initialization
- Provides detailed status information via STRING output
- Uses qualifiers (QI/QO) for event control

State Overview

1. **Not Initialized**: Block awaits INIT event
2. **Initialized**: Block ready for REQ requests
3. **Active**: Processing output requests
4. **Error**: Status message in case of problems

Application Scenarios

- Controlling digital output modules
- Communication with byte-based peripherals
- Integration into I/O subsystems
- Controlling hardware with byte commands

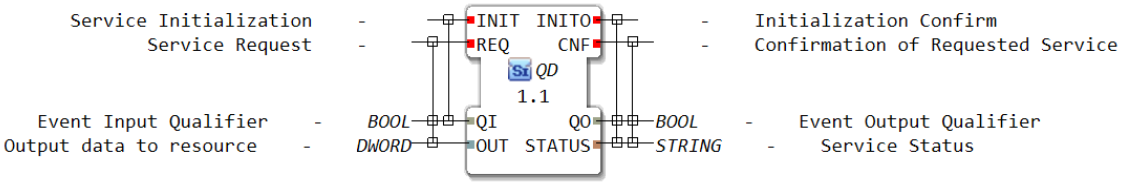
⚖️ Comparison with Similar Blocks

Compared to more general output blocks, QB is specifically optimized for byte data. Other blocks, such as WORD or DWORD output blocks, work with larger data types, while QB enables efficient processing of 8-bit data.

Conclusion

The QB function block offers a robust and standardized interface for byte output data in industrial control systems. Its clear event control and status feedback make it particularly suitable for reliable I/O operations in automated processes.

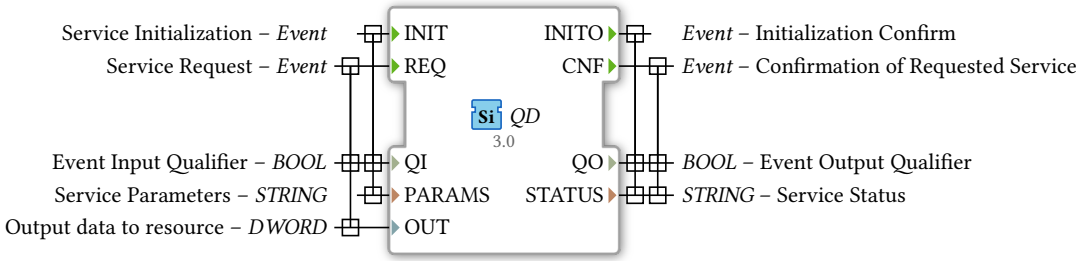
2.11.5.10 QD



• – ◦ • – ◦ • – ◦ •

Introduction

The QD function block is an output service interface component for outputting double-word (DWORD) data. It serves as an interface between the application logic and the physical output devices of a resource manager.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - initializes the function block with the required parameters
- **REQ**: Service Request - triggers data output to the resource

Event Outputs

- **INITO**: Initialization Acknowledgement - confirms completion of initialization
- **CNF**: Acknowledgement of Requested Service Execution - signals completion of data output

Data Inputs

- **QI**: Event Input Qualifier (BOOL) - controls the execution of service requests
- **PARAMS**: Service Parameters (STRING) - configuration parameters for the service
- **OUT**: Output Data (DWORD) - the double-word data to be transmitted to the resource

Data Outputs

- **QO**: Event Output Qualifier (BOOL) - status of the Service Execution
- **STATUS**: Service status (STRING) - detailed status information about service execution

Adapter

No adapter interfaces are available.

Functionality

The QD block operates as a cyclic service interface component. Upon receiving the INIT event, the block is initialized with the configuration data passed via PARAMS. The actual data output is triggered by the REQ event, whereby the DWORD data provided in the OUT input is transferred to the assigned resource. Each service request is acknowledged by corresponding confirmation events (INITO or CNF).

Technical Features

- Processes 32-bit data (DWORD)
- Supports configurable service configuration
- Provides detailed status feedback
- Works with qualifier flags for reliable service control

State Overview

The block goes through the following states:

1. **Not Initialized**: Block waits for an INIT event
2. **Initialized**: Block is ready for operation and waits for REQ events
3. **Service Execution**: Processing a REQ request
4. **Acknowledgement**: Feedback of the service result

Application Scenarios

- Control of digital output modules with 32-bit data width
- Control of complex actuators with multiple parameter passing
- Integration into distributed control systems with resource management
- Applications with high output data integrity requirements

⚖️ Comparison with Similar Blocks

Compared to simpler output blocks, QD offers:

- Enhanced parameterization through STRING parameters
- DWORD data width instead of simple BOOL or BYTE values
- Detailed status feedback for fault diagnosis
- Complete service interface characteristics with initialization procedure

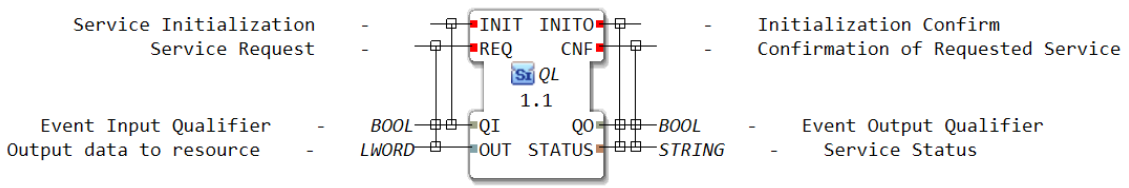
🛠️ Related Exercises

- Exercise_060

Conclusion

The QD function block represents a powerful and flexible solution for outputting double-word data in IEC 61499 systems. Its comprehensive parameterization and detailed status feedback make it particularly suitable for demanding automation applications that require reliable and configurable output services.

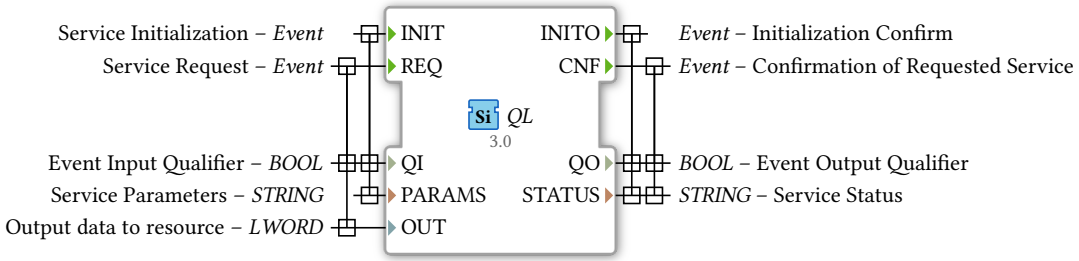
2.11.5.11 QL



• – ○ • – ○ • – ○ •

Introduction

The QL function block is an output service interface block for processing long-word output data. It serves as an interface between the control logic and external resources and enables the output of 64-bit data values.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters
- **OUT** (LWORD): Output Data for the Resource (64-bit)

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status Information

Adapters

No adapter interfaces are available.

Functionality

The QL block processes two main operations: initialization and data output. During the INIT operation, the service parameters are configured and the connection to the resource is initialized. The REQ operation performs the actual data output of the LWORD value to the connected resource. Each operation is acknowledged by corresponding confirmation events.

Technical Features

- Processing of 64-bit data values (LWORD)
- String-based parameter configuration
- Status feedback via STRING variable
- Qualifier for event control (QI/QO)

State Transitions

The block has two main states:

1. **Initialization State**: Processing the INIT request with PARAMS
2. **Operating State**: Processing REQ requests with OUT data

Application Scenarios

- Output of 64-bit control values to hardware interfaces
- Communication with external peripherals
- Integration into distributed control systems
- Connection to dedicated I/O modules

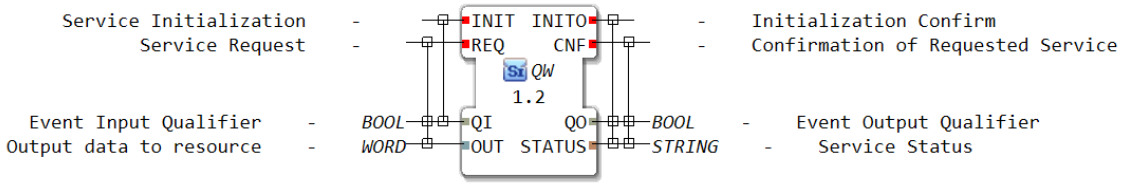
⚖️ Comparison with Similar Blocks

Compared to simpler output blocks, QL offers 64-bit data processing and extended parameter configuration. Compared to more general-purpose communication blocks, QL is specifically optimized for output functionality.

Conclusion

The QL function block provides a powerful interface for the output of long-word data and, through its structured event control and status feedback, offers a reliable basis for connecting external resources in automation systems.

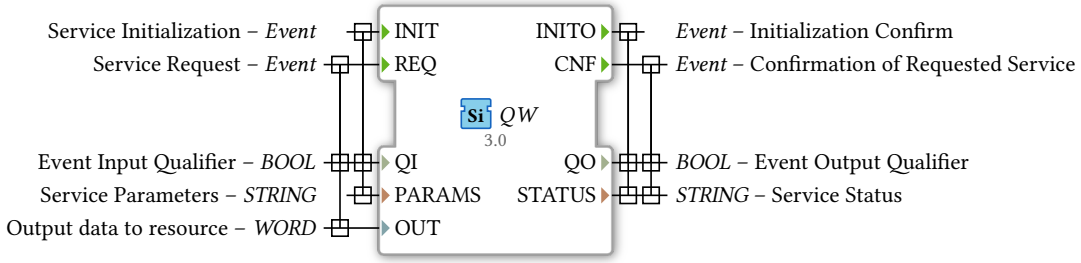
2.11.5.12 QW



• – ◦ • – ◦ • – ◦ •

Introduction

The QW function block is an output service interface module for outputting Word data. It serves as an interface between the control logic and physical output devices or resources, enabling the secure transfer of 16-bit data to external hardware.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs

- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters for Configuration
- **OUT** (WORD): Output Data for the Resource

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status Information

Adapters

No adapter interfaces are present.

Functionality

Upon receiving the INIT event, the QW block initializes the hardware interface with the passed PARAMS. After successful initialization, it confirms this with INITO. A REQ request outputs the WORD data received by OUT to the configured resource. The operation is confirmed with CNF, where STATUS provides information about the success or failure of the operation.

Technical Features

- Processes 16-bit WORD data
- String-based parameter configuration via PARAMS
- Status feedback via STRING variable
- Qualifier bits (QI/QO) for event control

State Overview

1. **Not Initialized**: Block awaits INIT event
2. **Initialized**: Block is ready and awaits REQ events
3. **Active**: Processes output requests and generates acknowledgments

Application Scenarios

- Control of digital output modules
- Communication with I/O systems
- Control of hardware with 16-bit data words
- Integration into distributed control systems

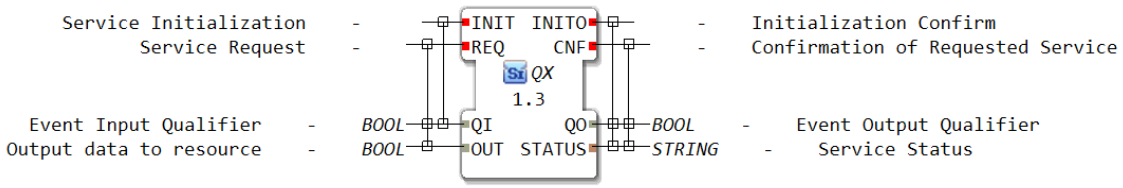
⚖️ Comparison with Similar Components

Compared to QB (byte output), QW processes 16-bit instead of 8-bit data. Compared to QD (double word), it offers a narrower data width with reduced resource requirements. The structure is similar to other I/O service blocks, but is specifically designed for WORD data.

Conclusion

The QW function block provides a robust and standardized interface for outputting 16-bit data to external resources. Its clear event handling and status feedback make it ideal for reliable real-time I/O operations in industrial control systems.

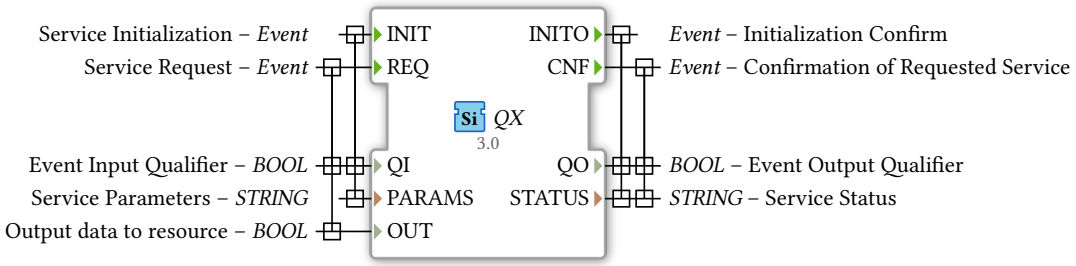
2.11.5.13 QX



• – ◦ • – ◦ • – ◦ •

Introduction

The QX function block is an output service interface function block for Boolean output data. It serves as an interface between the control logic and physical output devices and enables the output of digital signals to external resources.



Interface Structure

Event Inputs
- **INIT**: Service Initialization Event
- **REQ**: Service Request Event

Event Outputs
- **INITO**: Initialization Acknowledgement
- **CNF**: Acknowledgement of Requested Service Operation

Data Inputs
- **QI** (BOOL): Event Input Qualifier
- **PARAMS** (STRING): Service Parameters for Configuration
- **OUT** (BOOL): Output Data for the Resource

Data Outputs
- **QO** (BOOL): Event Output Qualifier
- **STATUS** (STRING): Service Status Information

Adapters
No adapter interfaces are available.

Functionality
The QX block processes two main operations: initialization and service requests. The INIT event initializes the service with the specified parameters. The REQ event outputs the Boolean OUT value to the connected resource. Each operation generates a corresponding acknowledgment (INITO or CNF) containing status information.

The INIT event initializes the service with the specified parameters.

Technical Features
- Specialized for Boolean output data
- Supports configurable service configuration
- Provides detailed status information via the STRING output STATUS
- Uses qualifiers (QI/QO) for event control

State Overview
The block cycles through the following states:

1. **Not Initialized**: Before the first INIT operation
2. **Initialized**: After successful INIT operation, ready for REQ operations
3. **Active**: During the processing of REQ operations

Application Scenarios
- Control of digital outputs (relays, LEDs, valves)
- Interface to actuators in automation systems
- Integration into I/O subsystems for distributed controllers
- Test and simulation environments for output signals

⚖️ Comparison with Similar Blocks
Compared to generic output blocks, QX offers specific service interface functionality with configurable settings and detailed status reporting. Other blocks, such as simple BOOL outputs, typically have fewer configuration options and less status information.

🛠️ Related exercises
- Uebung_001
- Uebung_001c
- Uebung_002
- Uebung_002a
- Uebung_002a2

- Uebung_002a3
- Uebung_002a4
- Uebung_002a5b
- Uebung_002b2
- Uebung_002b3
- Uebung_003
- Uebung_003a0
- Uebung_003a_sub
- Funk_IX_TO_DataPanel_MI_QX
- IX_TO_Aux_QX
- Uebung_003d
- Uebung_004a
- Uebung_004a2
- Uebung_004a2_2
- Uebung_004a2_3
- Uebung_004a3
- Uebung_004a4
- Uebung_004a5
- Uebung_004a6
- Uebung_004a7
- Uebung_004a8
- Uebung_004a9
- Uebung_004b
- Uebung_004b2
- Uebung_004b3
- Uebung_004c1
- Uebung_004c2
- Uebung_004c3
- Uebung_004c4
- Uebung_004c5
- Uebung_004c6
- Uebung_004c7
- Uebung_005
- Uebung_006
- Uebung_006a
- Uebung_006a2
- Uebung_006a3
- Uebung_006a4
- Uebung_006b
- Uebung_006c
- Uebung_006d
- Uebung_006e1
- Uebung_006e2
- Uebung_007
- Uebung_007a1
- Uebung_007a2
- Uebung_007a3
- Uebung_008
- Uebung_009
- Uebung_010
- Uebung_010a
- Uebung_010a2
- Uebung_010a3
- Uebung_010a4
- Uebung_010b1
- Uebung_010b2
- Uebung_010b3
- Uebung_010b6
- Uebung_010b7

- Uebung_010b8
- Uebung_010b9
- Uebung_010bA
- Uebung_010bA2
- Uebung_010bA3
- Uebung_010bA4
- Uebung_010c
- Uebung_010c2
- Uebung_013
- Uebung_019b
- Uebung_019c
- Uebung_020a
- Uebung_020b
- Uebung_020c
- Uebung_020c2
- Uebung_020c3
- Uebung_020d
- Uebung_020e
- Uebung_020e2
- Uebung_020f
- Uebung_020f2
- Uebung_020f3
- Uebung_020g
- Uebung_020h
- Uebung_020i
- Uebung_021
- Uebung_022
- Uebung_023
- Uebung_024
- Uebung_025
- SR_TO_QX_TRIG_EO
- Uebung_028
- Uebung_029
- Uebung_030
- Uebung_032
- IX_TO_logiBUS_LED_strip_QX
- Uebung_035
- Uebung_035a
- Uebung_035a2
- Uebung_035a3
- Uebung_035b
- Uebung_035c
- Uebung_036
- Uebung_037
- Uebung_038
- SoftkeyOverride_TO_QX_BG
- Softkey_T_FF_ILOCK_TO_QX_BG
- Uebung_039b
- Uebung_040
- Uebung_040_2
- Uebung_041
- Uebung_049
- Uebung_051
- Uebung_052
- Uebung_053
- Uebung_054
- Uebung_055
- Uebung_056
- Bool8_TO_logiBUS_QX8

- Uebung_071
- Uebung_071a
- Uebung_071b
- Uebung_072b
- Uebung_080
- Uebung_080b
- Uebung_080c
- Uebung_081
- Uebung_082
- Uebung_083
- Uebung_084
- Uebung_085
- Uebung_087
- Uebung_087a1
- Uebung_087a2
- Uebung_088
- Uebung_089
- Uebung_090a1
- Uebung_090a2
- Uebung_091
- Uebung_093
- Uebung_093b
- Uebung_094
- Uebung_094a
- Uebung_095
- Uebung_110
- Uebung_111
- Uebung_160
- Uebung_160b
- Uebung_160b2
- Uebung_177
- Uebung_178
- Uebung_179
- Exercise_180

Conclusion

The QX function block provides a robust and configurable solution for Boolean output services. Its structured event handling and detailed status feedback make it particularly suitable for reliable automation applications where transparency regarding the I/O status is required.

2.11.5.14 ADS

🎧 Podcast

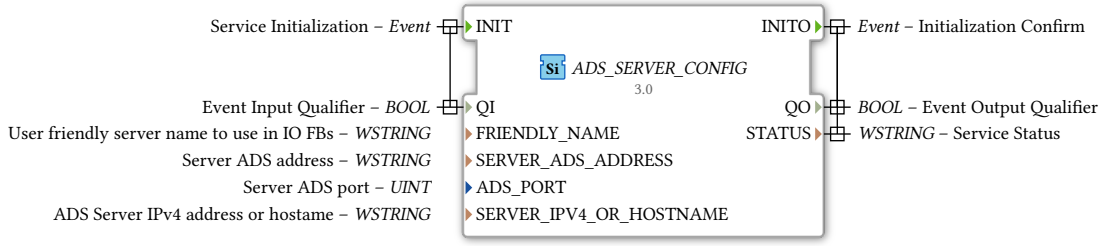
- Germany's Green Energy Crossroads: Navigating the End of Renewable Subsidies with Innovative Solutions

ADS_SERVER_CONFIG

• – ◦ • – ◦ • – ◦ •

Introduction

The ADS_SERVER_CONFIG function block is used to configure ADS server connections in 4diac systems. It enables the initialization and parameterization of connection settings for ADS communication, which can be used in downstream I/O function blocks.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event - Starts the configuration of the ADS server

Event Outputs

- **INITO**: Initialization Acknowledgement - Signals successful completion of the configuration

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Enables/disables initialization
- **FRIENDLY_NAME** (WSTRING): User-friendly server name for use in I/O function blocks
- **SERVER_ADS_ADDRESS** (WSTRING): ADS address of the server
- **ADS_PORT** (UINT): ADS port of the server
- **SERVER_IPV4_OR_HOSTNAME** (WSTRING): IPv4 address or hostname of the ADS server

Data Outputs

- **QO** (BOOL): Event output qualifier - Indicates the initialization status
- **STATUS** (WSTRING): Service status information

Adapter

No adapter interfaces are available.

Functionality

The function block is activated by the INIT event and processes the provided configuration parameters. Upon successful processing, it outputs the INITO event and makes the configured parameters available to downstream ADS-IO function blocks. The STATUS output provides detailed information about the configuration process.

Technical Features

- Uses WSTRING data types for international character support
- Supports both IPv4 addresses and hostnames for maximum flexibility
- Separate parameters for ADS address and network address enable complex network configurations
- Port configuration as UINT for numeric port specification

State Overview

1. **Inactive**: Waiting for INIT event
2. **Configuration**: Processing input parameters
3. **Active**: Configuration completed successfully, parameters are available
4. **Error**: Configuration failed, STATUS contains error information

Application Scenarios

- Initial system configuration in ADS-based automation systems
- Central management of ADS server connection parameters
- Dynamic reconfiguration of network settings
- Integration into larger I/O management systems

⚖️ Comparison with Similar Blocks

Compared to simple ADS I/O blocks, ADS_SERVER_CONFIG provides a central configuration option for multiple downstream blocks. It separates the configuration logic from the actual data communication, thus enabling reusable server settings.

Conclusion

ADS_SERVER_CONFIG is an essential component for structured ADS communication in 4diac systems. Through centralized configuration management, it simplifies the maintenance and modification of network settings and promotes consistent connection parameters across various I/O function blocks.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

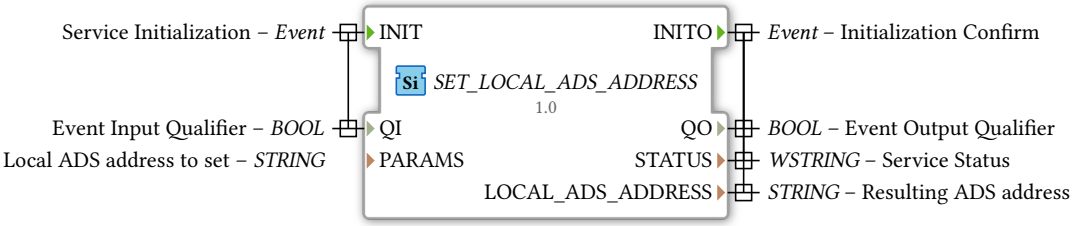
]

SET_LOCAL_ADS_ADDRESS

• – ◦ • – ◦ • – ◦ •

Introduction

The SET_LOCAL_ADS_ADDRESS function block is a Service Interface Function Block for configuring the local ADS address. ADS (Automation Device Specification) is a communication protocol commonly used in automation technology. This block allows the dynamic setting of the local ADS address during runtime.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Starts the configuration process for the local ADS address

Event Outputs

- **INITO**: Initialization Confirmation - Signals the completion of the configuration process

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Enables/disables the initialization process
- **PARAMS** (STRING): Local ADS address to be set

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Confirms the successful completion of the operation
- **STATUS** (WSTRING): Service Status - Contains status information about the performed operation
- **LOCAL_ADS_ADDRESS** (STRING): Resulting ADS Address - The actual ADS address set

Functionality

The function block is accessed by The INIT event is activated. Upon activation, it reads the PARAMS input, which contains the desired local ADS address in string format. The function block then configures the local ADS interface with the specified address. After completion, the INITO event is triggered, and the resulting ADS address and status information are made available at the outputs.

Technical Features

- Uses STRING data type for ADS addresses
- Provides comprehensive status information in WSTRING format

- Supports dynamic address configuration at runtime
- Implements a standardized initialization protocol

State Transitions

1. **Initial State**: Waits for INIT event
2. **Configuration State**: Processes the ADS address parameters
3. **Completion State**: Sends INITO event with result data

Application Scenarios

- Dynamic configuration of ADS devices in distributed automation systems
- Addressing of ADS components in flexible production facilities
- Integration into systems with variable network configurations
- Maintenance and commissioning of ADS-based controllers

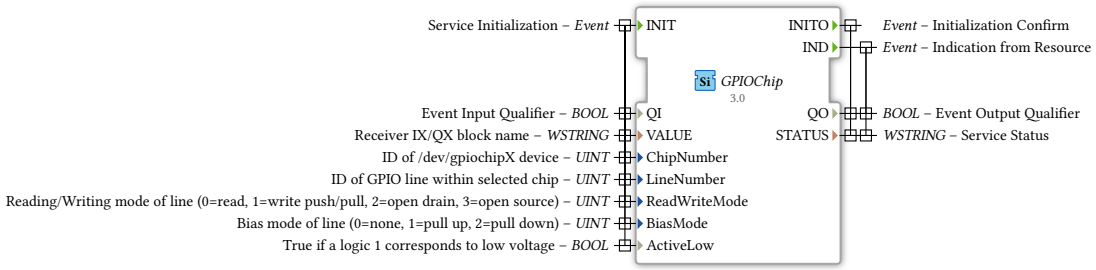
⚖️ Comparison with Similar Blocks

Compared to static ADS configuration blocks, SET_LOCAL_ADS_ADDRESS enables dynamic address management at runtime. The function block also provides detailed status feedback, which simplifies troubleshooting.

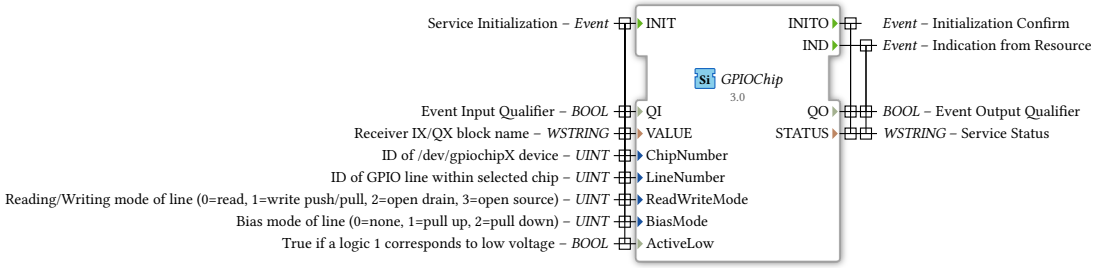
Conclusion

The SET_LOCAL_ADS_ADDRESS function block offers a flexible solution for the dynamic configuration of ADS addresses in IEC 61499-based control systems. Its clear interface definition and comprehensive status feedback make it particularly suitable for applications requiring flexible network configuration.

2.11.5.15 GPIOChip



GPIOChip



• – ○ • – ○ • – ○ •

Introduction

The GPIOChip function block is a service interface function block for controlling GPIO chips under Linux systems. It enables access to GPIO lines via the Linux GPIO Character Device Interface (/dev/gpiochipX) and supports various operating modes for reading and writing GPIO signals.

Interface Structure

Event Inputs

- **INIT**: Service initialization event with the associated data QI, VALUE, ChipNumber, LineNumber, ReadWriteMode, BiasMode, and ActiveLow

Event Outputs

- **INITO**: Initialization confirmation with QO and STATUS
- **IND**: Resource indication with QO and STATUS

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **VALUE** (WSTRING): Receiver IX/QX block name
- **ChipNumber** (UINT): ID of the /dev/gpiochipX device (initial value: 0)
- **LineNumber** (UINT): ID of the GPIO line within the selected chip (initial value: 0)
- **ReadWriteMode** (UINT): Read/write mode of the line (0=read, 1=write) push/pull, 2=open drain, 3=open source) (Initial value: 0)
- **BiasMode** (UINT): Bias mode of the line (0=none, 1=pull up, 2=pull down) (Initial value: 0)
- **ActiveLow** (BOOL): True when logic 1 corresponds to low voltage (Initial value: 0)

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (WSTRING): Service Status

Adapter

No adapter interfaces available.

Functionality

Upon receiving the INIT event, the GPIOChip FB initializes a connection to a specific GPIO chip and a defined GPIO line. The configuration includes the operating mode (read/write), the bias mode, and the polarity setting (ActiveLow). After successful initialization, the block confirms with INITO+; in case of errors, it confirms with INITO- and a corresponding STATUS. ...

Technical Features

- Supports various GPIO operating modes: Read, Write (Push/Pull), Open Drain, and Open Source
- Configurable bias settings: None, Pull-Up, Pull-Down
- Active-Low mode for inverted logic
- Linux GPIO Character Device Interface based
- Automatic error handling with status feedback

State Overview

The function block goes through the following states:

1. **Not Initialized**: Initial state before INIT
2. **Initialization**: During GPIO chip configuration
3. **Ready**: After successful initialization (INITO+)
4. **Error**: In case of initialization errors (INITO-)

Application Scenarios

- Controlling GPIO pins in embedded Linux systems
- Low-level I/O operations in automation systems
- Interface to external sensors and actuators
- Prototyping and development of IoT devices
- Industrial control applications

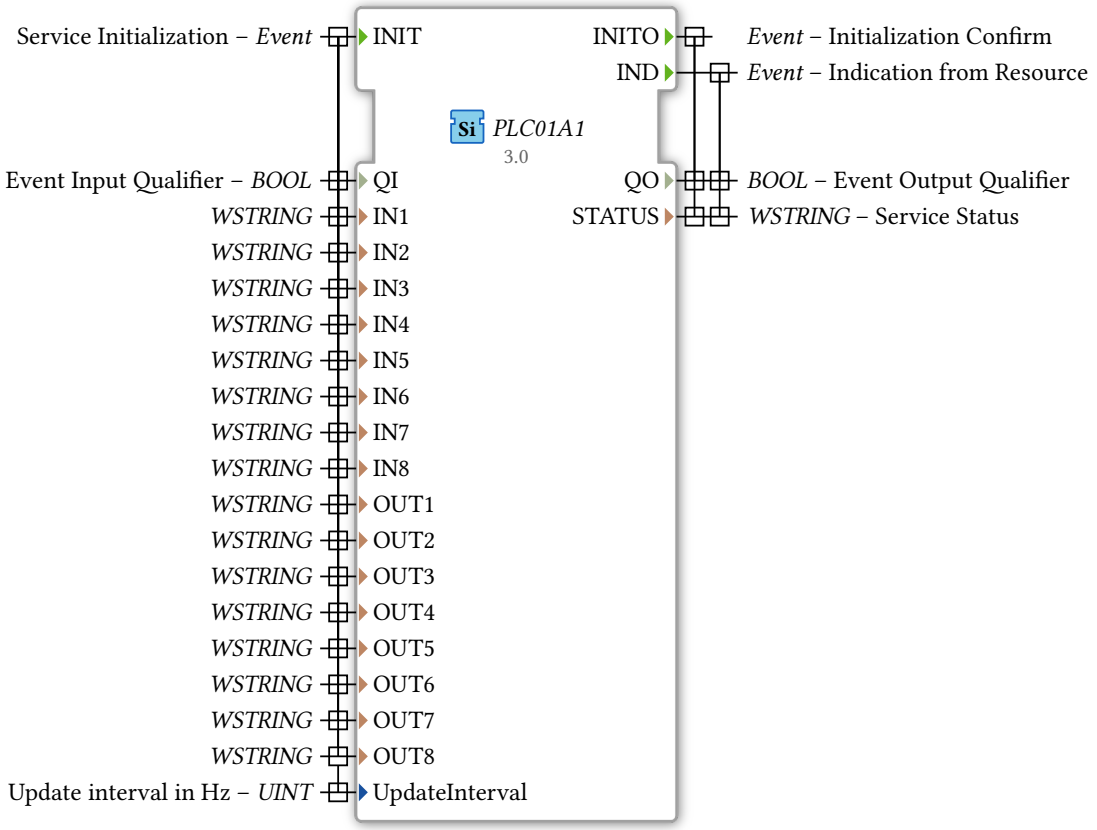
⚖️ Comparison with similar Building Blocks

Compared to simple digital I/O blocks, GPIOChip offers advanced features such as configurable bias modes, various write modes, and direct access to the Linux GPIO subsystem. It is specifically designed for Linux systems with GPIOChip support.

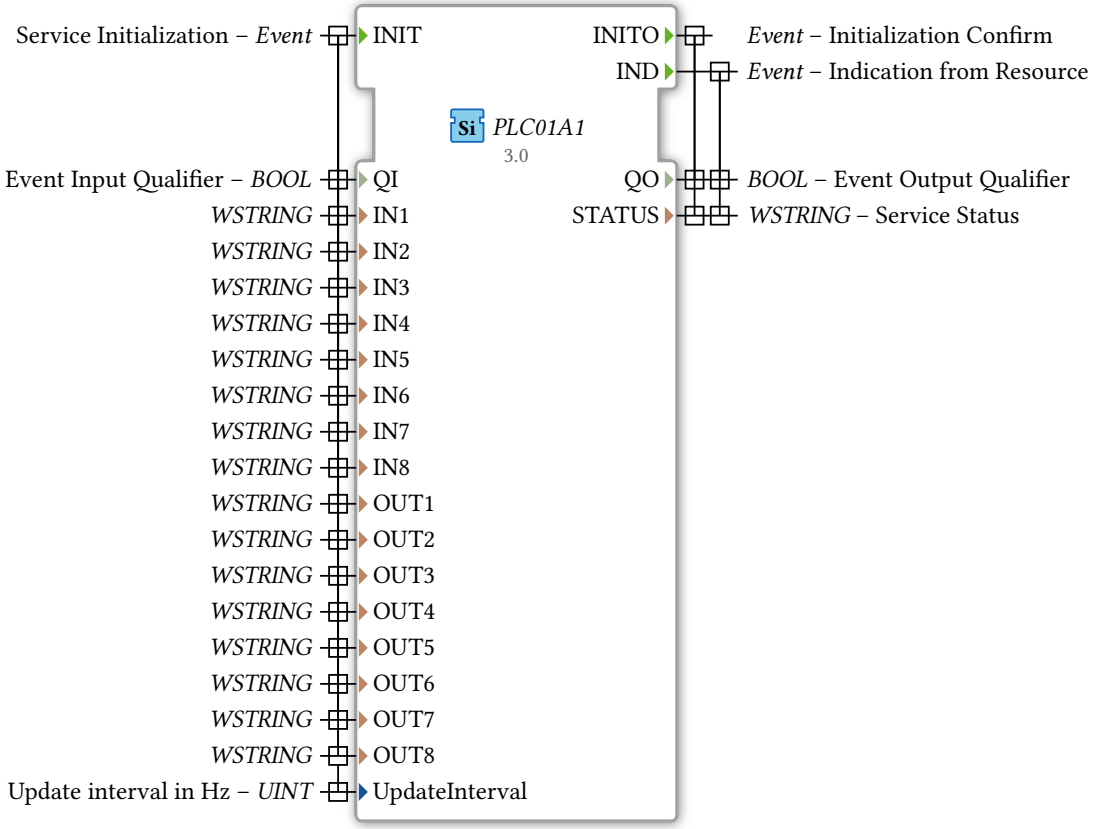
Conclusion

The GPIOChip functional block provides a powerful interface for GPIO hardware control under Linux. Its extensive configurability and standardized service interface make it ideal for demanding embedded applications that require direct hardware access.

2.11.5.16 PLC01A1



PLC01A1



• – ◦ • – ◦ • – ◦ •

Introduction

The PLC01A1 function block serves as an interface for accessing the inputs and outputs of the PLC01A1 module. This block enables the control of 8 digital outputs and the reading of 8 digital inputs via a standardized 4diac interface.

Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- Linked to: QI, UpdateInterval, IN1-IN8, OUT1-OUT8

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS
- **IND**: Resource Indication
- Linked to: QO, STATUS

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **IN1-IN8** (WSTRING): 8 Digital Inputs
- **OUT1-OUT8** (WSTRING): 8 Digital Outputs
- **UpdateInterval** (UINT): Update Interval in Hz (Initial Value: 25)

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (WSTRING): Service Status Information

Adapter

No adapter interfaces available.

Functionality

The PLC01A1 function block initializes the hardware interface upon receiving the INIT event. After successful initialization, the INITO event is output. The block continuously monitors inputs IN1-IN8 and updates outputs OUT1-OUT8 according to the configuration. The update interval can be adjusted using the UpdateInterval parameter.

Technical Features

- Uses WSTRING data type for all inputs/outputs
- Configurable update interval (default: 25 Hz)
- Separate initialization and operating states
- Status feedback via STATUS output

State Overview

1. **Not Initialized**: Waits for INIT event
2. **Initialization**: Processes INIT with configuration data
3. **Operation**: Monitors inputs and controls outputs
4. **Error**: Outputs error status via STATUS

Application Scenarios

- Connecting PLC01A1 hardware to 4diac controllers
- Digital signal processing in automation systems
- I/O management in distributed control systems
- Prototype development with PLC01A1 modules

⚖️ Comparison with Similar Components

Compared to generic I/O components, PLC01A1 offers specific customization for the PLC01A1 module with Eight predefined inputs and eight outputs. The use of WSTRING enables flexible data representation, but differs from typical BOOL-based I/O blocks.

Conclusion

The PLC01A1 function block provides a reliable and configurable interface for PLC01A1 hardware in 4diac-based control systems. Thanks to the clear separation of the initialization and operating phases, as well as the configurable update rate, the block is suitable for both real-time applications and general automation tasks.

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

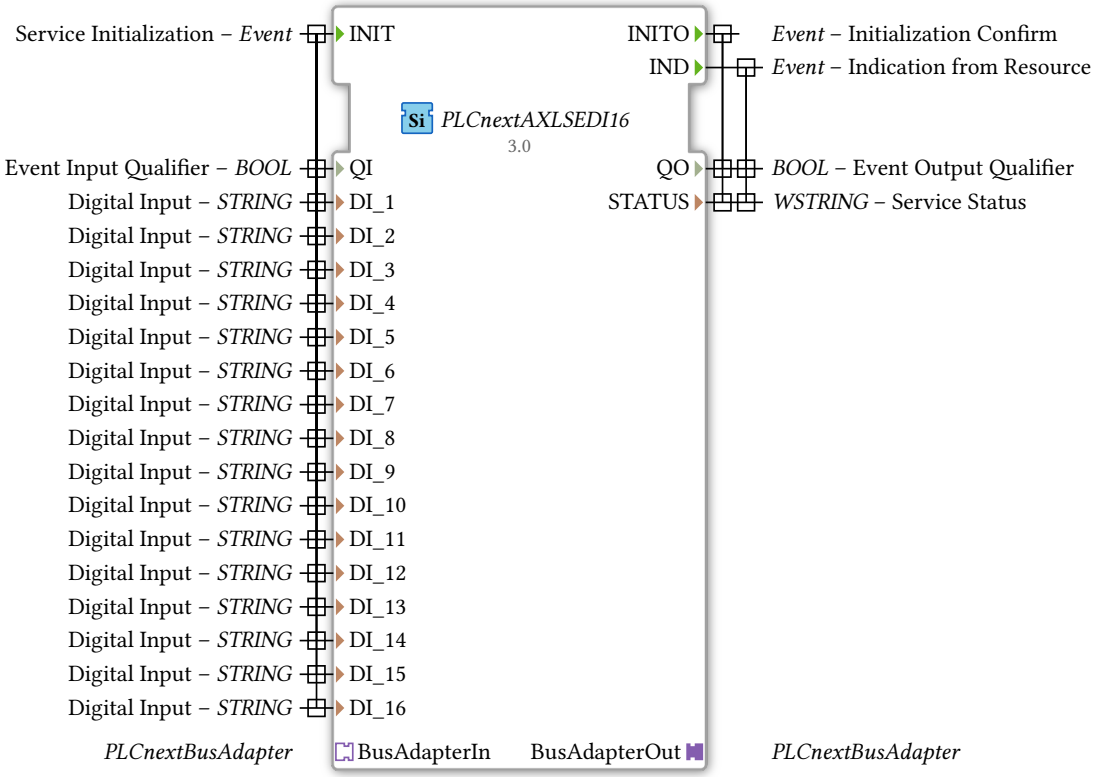
2.11.5.17 PLCnext

PLCnextAXLSEDI16

• – ◦ • – ◦ • – ◦ •

Introduction

The PLCnextAXLSEDI16 is a Service Interface Function Block for connecting to PLCnext systems. This function block serves as an interface for digital inputs and enables communication with the PLCnext bus architecture. It supports 16 digital input channels and offers standardized initialization and status feedback.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the function block with the configured parameters

Event Outputs

- **INITO**: Initialization Confirm - Confirms successful initialization
- **IND**: Indication from Resource - Signals status changes or events from the resource

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Controls the initialization
- **DI_1 to DI_16** (STRING): Digital Input - 16 digital input channels for configuration

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Event output status
- **STATUS** (WSTRING): Service Status - Detailed status information

Adapters

- **BusAdapterOut** (Plug): Outgoing bus adapter for PLCnext Communication
- **BusAdapterIn** (Socket): Incoming bus adapter for PLCnext communication

Functionality

The function block initializes itself via the INIT event and configures the 16 digital input channels based on the DI_Parameters. After successful initialization, it confirms this via INITO. During operation, it continuously monitors the inputs and signals changes via the IND event. Communication with the PLCnext bus takes place via the integrated adapter interfaces.

Technical Features

- Supports 16 independent digital input channels
- String-based input configuration
- Unicode-enabled status feedback (WSTRING)
- Integrated PLCnext bus adapter communication
- Standard-compliant IEC 61499-2 implementation

State Overview

1. **Not Initialized**: Function block awaits INIT event
2. **Initialization**: Processing configuration parameters
3. **Ready for Operation**: Digital input monitoring active
4. **Fault State**: Status feedback issues

Application Scenarios

- Connecting digital sensors to PLCnext systems
- Industrial automation controllers
- Monitoring systems with multiple input signals
- PLCnext-based control architectures

⚖️ Comparison with Similar Blocks

Compared to simple digital input blocks, PLCnextAXLSEDI16 offers enhanced configurability through string parameters and integrated bus communication for PLCnext systems. The 16 channels enable a higher input density than standard modules.

Conclusion

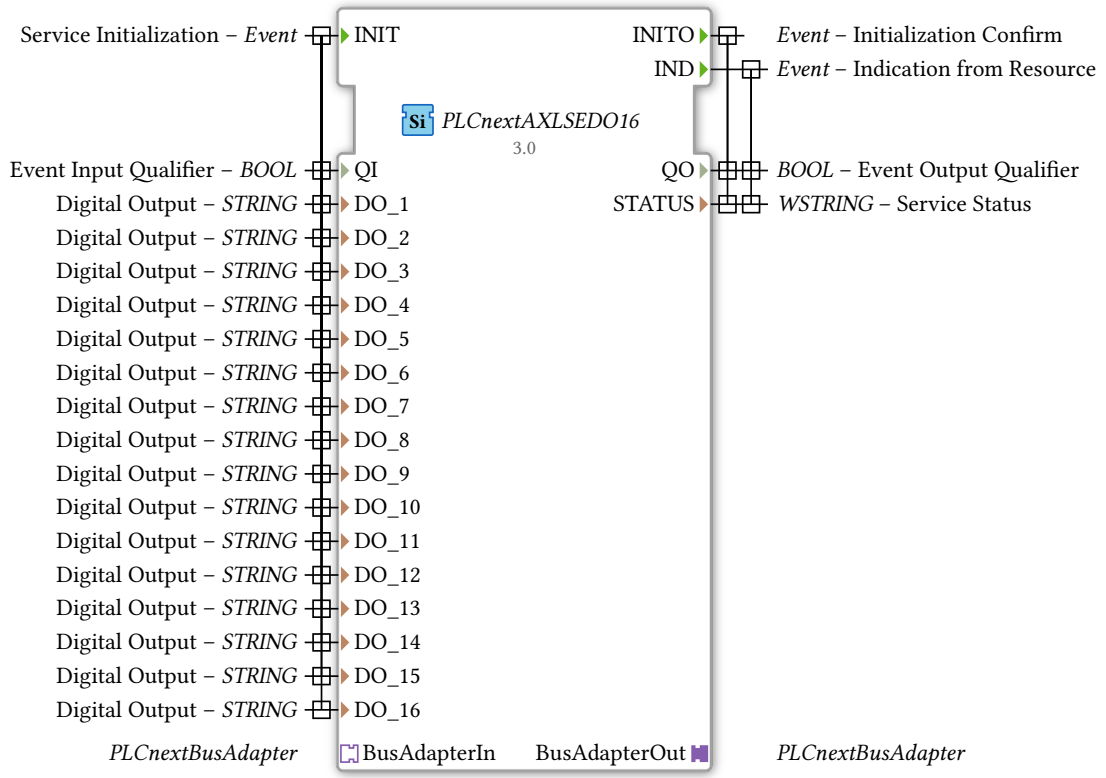
The PLCnextAXLSEDI16 is a powerful function block for integrating digital inputs into PLCnext-based automation systems. Its flexible configuration and robust bus communication make it ideal for industrial applications with multiple digital signals.

PLCnextAXLSEDO16

• – ◦ • – ◦ • – ◦ •

Introduction

The PLCnextAXLSEDO16 is a Service Interface Function Block for controlling 16 digital outputs in PLCnext systems. This block serves as an interface between the IEC 61499-based control logic and the physical outputs of the PLCnext AXLSEDO16 module.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the function block and configures the digital outputs

Event Outputs

- **INITO**: Initialization Confirm - Confirms successful initialization
- **IND**: Indication from Resource - Signals status changes or errors from the resource adapter

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Controls the initialization
- **DO_1** to **DO_16** (STRING): Digital Outputs - Configure the respective output channels

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Status of the event output
- **STATUS** (WSTRING): Service Status - Detailed status information

Adapters

- **BusAdapterOut** (Plug): Outgoing bus adapter for the Communication with the Hardware
- **BusAdapterIn** (Socket): Incoming bus adapter for feedback from the hardware

Functionality

Upon receiving the INIT event, the module initializes the 16 digital outputs based on the configuration values in DO_1 to DO_16. Initialization is confirmed by INITO. During operation, status changes and errors can be reported via the IND event. Communication with the physical hardware occurs via the bus adapter interfaces.

Technical Features

- Supports 16 independent digital outputs
- Uses STRING type for output configuration, enabling flexible parameterization
- Integrates seamlessly into PLCnext systems via specific bus adapters
- Provides comprehensive status feedback via the WSTRING variable

State Overview

1. **Not Initialized**: Block is waiting for the INIT event
2. **Initialization in Progress**: Processing configuration parameters
3. **Ready for Operation**: Outputs are configured and ready for operation
4. **Fault State**: In case of problems, STATUS is populated with error information

Application Scenarios

- Control of relays and actuators in automation systems
- Control of lighting systems
- Control of valves and motors
- Signal output in process control systems

⚖️ Comparison with Similar Blocks

Compared to simpler digital output blocks, PLCnextAXLSEDO16 offers:

- A higher number of channels (16 instead of the typical 16) 8)
- Specific integration for PLCnext hardware
- Extended status feedback
- Flexible string-based configuration

Conclusion

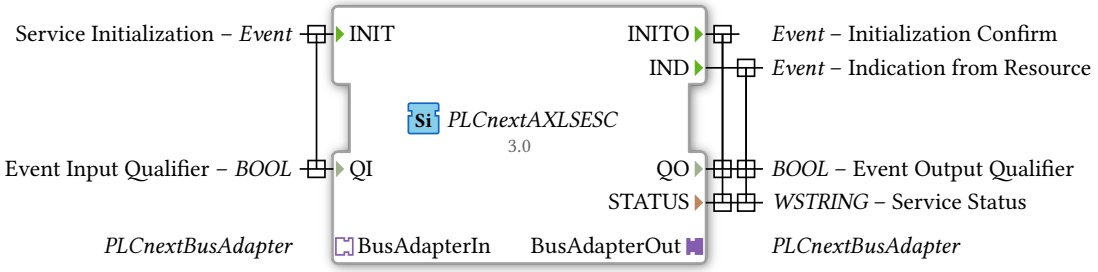
The PLCnextAXLSEDO16 is a powerful function block for controlling digital outputs in PLCnext environments. Its 16 channels, flexible configurability, and comprehensive status feedback make it ideal for complex automation tasks where multiple actuators need to be controlled simultaneously.

PLCnextAXLSESC

• – ◦ • – ◦ • – ◦ •

Introduction

The PLCnextAXLSESC is a Service Interface Function Block that serves as an interface for PLCnext bus systems. This block enables communication between the 4diac environment and PLCnext hardware components via special bus adapters.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the service with the qualifier QI

Event Outputs

- **INITO**: Initialization Acknowledgement - Confirms successful initialization with QO and STATUS
- **IND**: Resource Indication - Signals status changes with QO and STATUS

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Controls the service initialization

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Displays the status of the service output
- **STATUS** (WSTRING): Service Status - Contains detailed status information as a Unicode string

Adapters

- **BusAdapterOut** (Plug): Outgoing bus adapter for PLCnext communication
- **BusAdapterIn** (Socket): Incoming bus adapter for PLCnext communication

Functionality

The PLCnextAXLSESC function block acts as an intermediary between the 4diac controller and PLCnext hardware. Upon receiving an INIT event with QI enabled, the function block initializes communication via the bus adapters. Successful initialization is confirmed by the INITO event. During operation, status changes and indications from the hardware can be reported via the IND event.

Technical Features

- Uses WSTRING for STATUS output, supporting international character sets
- Implements a bidirectional adapter system for flexible bus communication
- Separate events for initialization and ongoing indications
- Qualifier bits (QI/QO) for reliable event control

State Overview

1. **Not Initialized**: Waits for an INIT event with QI=TRUE
2. **Initialization Phase**: Processes bus adapter communication
3. **Operational**: Sends INITO acknowledgment and is ready for IND events
4. **Active Operation**: Processes incoming indications and outputs STATUS

Application Scenarios

- Integration of PLCnext hardware into 4diac-based control systems
- Communication with PLCnext I/O modules and peripherals
- Monitoring and control of industrial automation processes
- Bridge functionality between 4diac and PLCnext Ecosystem

⚖️ Comparison with Similar Function Blocks

Compared to generic I/O blocks, PLCnextAXLSESC offers specific adaptations for the PLCnext platform, particularly through its integrated bus adapters. While standard I/O blocks provide general interfaces, this block is specifically optimized for the requirements of the PLCnext architecture.

Conclusion

The PLCnextAXLSESC function block provides a reliable and specialized interface for integrating PLCnext hardware into 4diac control systems. Its clear event structure and flexible adapter architecture enable robust communication in industrial automation applications.

🌐 Related Topic Subpages on ms-muc-docs.de

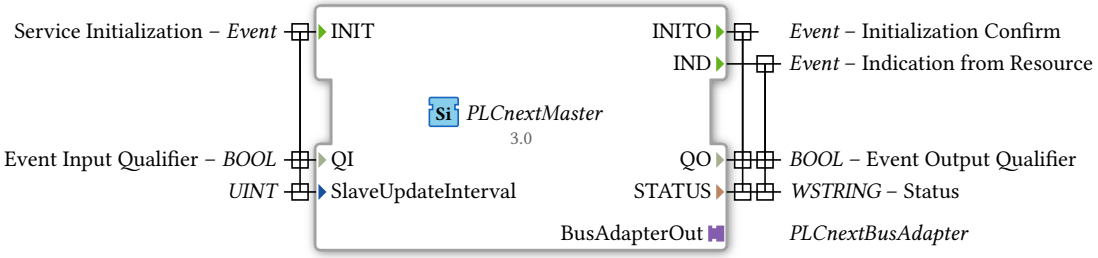
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

PLCnextMaster

• – ◦ • – ◦ • – ◦ •

Introduction

The PLCnextMaster is a Service Interface Function Block that serves as the master component for communication with PLCnext systems. This block enables the initialization and control of bus communication in PLCnext-based automation solutions.



Interface Structure

Event Inputs
- **INIT**: Service Initialization
- Linked to: QI, SlaveUpdateInterval

Event Outputs
- **INITO**: Initialization Confirmation
- Linked to: QO, STATUS
- **IND**: Resource Indication
- Linked to: QO, STATUS

Data Inputs
- **QI** (BOOL): Event Input Qualifier - Controls service activation
- **SlaveUpdateInterval** (UINT): Update interval for slaves with an initial value of 25

Data Outputs
- **QO** (BOOL): Event Output Qualifier - Confirms the operation status
- **STATUS** (WSTRING): Status information as a Unicode string

Adapters
- **BusAdapterOut**: PLCnextBusAdapter for Bus Communication
- Type: eclipse4diac::io::plcnext::PLCnextBusAdapter

Functionality
The PLCnextMaster initializes and manages communication with PLCnext systems. The service starts when the INIT event is enabled with QI=TRUE. The SlaveUpdateInterval defines the update interval for connected slave components. After successful initialization, INITO confirms the operating state, while IND events indicate asynchronous status changes.

Technical Features
- Supports Unicode status messages (WSTRING)
- Configurable slave update interval
- Adapter-based bus communication
- Initial value for SlaveUpdateInterval set to 25

State Overview
1. **Inactive**: Before INIT activation
2. **Initialization**: During INIT processing
3. **Active**: After successful INITO confirmation
4. **Ready**: For IND events during active communication

Application Scenarios
- PLCnext-based automation systems
- Industry 4.0 applications
- Networked control systems
- IoT edge computing solutions

⚖️ Comparison with similar components
Compared to generic bus master components, the PLCnextMaster offers specific optimizations for PLCnext hardware and supports the specific bus architecture of PLCnext systems.

Conclusion
The PLCnextMaster is a specialized function block for integrating PLCnext systems into 4diac-based automation solutions. Its configurable parameters and adapter interface enable flexible and reliable communication in industrial control environments.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

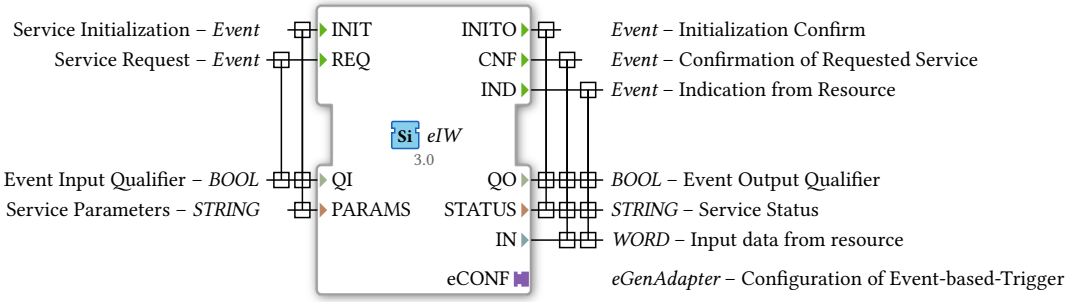
2.11.5.18 eIO

eIW

• – ◦ • – ◦ • – ◦ •

Introduction

The eIW function block is an input service interface for word input data (WORD) with event-based triggering. It serves as an interface between the control application and physical input devices and enables the querying of 16-bit data words.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - initializes the function block and configures the hardware parameters
- **REQ**: Service Request - triggers a query for the input data

Event Outputs

- **INITO**: Initialization Acknowledgement - confirms successful initialization
- **CNF**: Acknowledge of Requested Service Request - provides the requested input data
- **IND**: Indication from Resource - signals asynchronous state changes of the input data

Data Inputs

- **QI** (BOOL): Event Input Qualifier - controls the execution of service requests
- **PARAMS** (STRING): Service Parameters - configuration parameters for the hardware interface

Data Outputs

- **QO** (BOOL): Event Output Qualifier - indicates success or failure Operation at
- **STATUS** (STRING): Service status - detailed status information about the executed operation
- **IN** (WORD): Input data from the resource - the read 16-bit data

Adapter

- **eCONF** (eclipse4diac::io::eio::eGenAdapter): Configuration of event-based triggering - allows customization of the trigger behavior

Functionality

The eIW function block acts as an intermediary between the control application and physical input devices. During initialization (INIT), the hardware parameters are configured. Subsequently, input data can be selectively queried via REQ events. The block supports both synchronous queries (CNF) and asynchronous notifications for state changes (IND).

Technical Features

- Processes 16-bit word data (WORD data type)
- Supports event-based and polling-based query modes
- Provides detailed status feedback via the STATUS output

- Enables flexible hardware configuration via the PARAMS input
- Integrates an adapter interface for advanced trigger configuration

State Overview

The function block goes through the following states:

1. **Not Initialized**: Block is operational but not configured
2. **Initialized**: Hardware parameters are configured, block is operational
3. **Query Active**: Service request is being processed
4. **Data Available**: Input data is available and is being output

Application Scenarios

- Querying digital input groups (16-bit)
- Reading encoder values
- Monitoring status registers
- Acquiring sensor data words
- Interface to I/O modules with word access

⚖️ Comparison with Similar Devices Function Blocks

Compared to simple digital input blocks, eIW offers:

- Advanced 16-bit data processing
- Detailed status feedback
- Flexible configuration options
- Support for event-based and polling-based operating modes
- Adapter interface for extended functionality

Conclusion

The eIW function block provides a powerful and flexible interface for processing word input data in IEC 61499 control systems. Its combined support for synchronous queries and asynchronous notifications makes it particularly suitable for applications requiring both deterministic queries and time-critical state changes.

–

🌐 Related topic subpages on ms-muc-docs.de

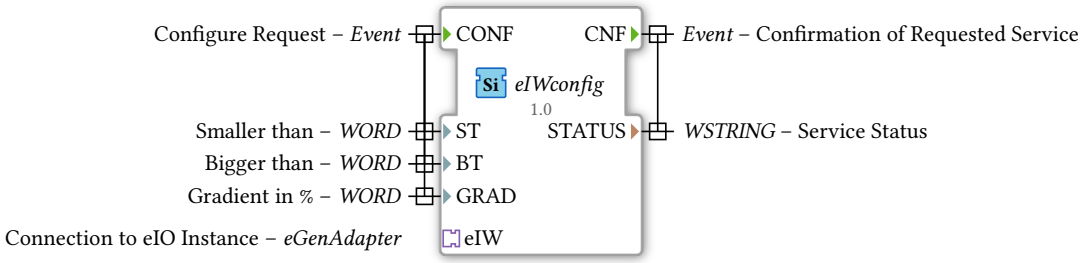
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

eIWconfig

• – ◦ • – ◦ • – ◦ •

Introduction

The eIWconfig function block serves as a service interface for configuring eIO instances. It allows you to set thresholds and gradients for industrial I/O operations.



Interface Structure

Event Inputs

- **CONF**: Configuration request - triggers the configuration of the eIO instance

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the configuration

Data Inputs

- **ST** (WORD): “Smaller than” - Lower threshold
- **BT** (WORD): “Bigger than” - Upper threshold
- **GRAD** (WORD): Gradient in percent - Steepness of the transitions

Data Outputs

- **STATUS** (WSTRING): Service status - feedback on the configuration process

Adapter

- **eIW**: Connection to the eIO instance (Type: eclipse4diac::io::eio::eGenAdapter)

Functionality

The function block receives via The CONF event is a configuration request along with the parameters ST, BT, and GRAD. These parameters are transmitted to the eIO instance via the eIW adapter. After successful configuration, the block confirms the process with the CNF event and provides status feedback via STATUS.

Conf event.

Technical Features

- Uses WORD data types for all numeric inputs
- STATUS output as WSTRING for detailed status information
- Adapter-based architecture for loose coupling to the eIO instance

State Overview

1. **Ready**: Waiting for CONF event
2. **Configuring**: Processing configuration parameters
3. **Acknowledging**: Sending CNF event with status information

Application Scenarios

- Configuration of I/O thresholds in automation systems
- Setting filter parameters for sensor signals
- Dynamic adjustment of limit values in control systems

⚖️ Comparison with Similar Blocks

Compared to simple I/O configuration blocks, eIWconfig offers extended functionality with gradient control and an adapter-based architecture for greater flexibility.

Conclusion

The eIWconfig function block provides a flexible and extensible solution for configuring eIO instances, distinguished by its adapter-based architecture and extensive parameterization options.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

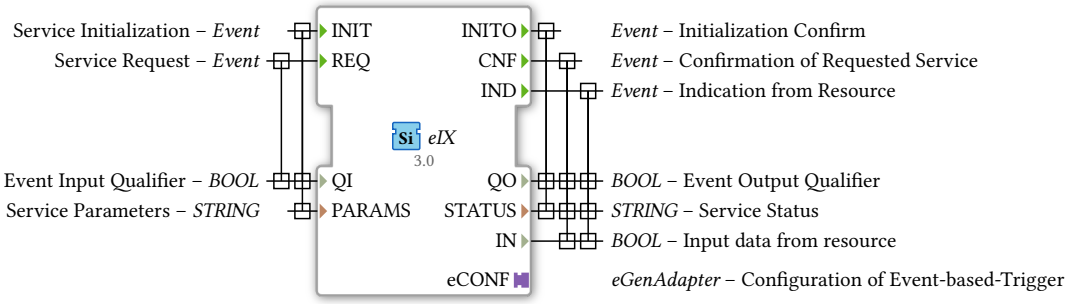
]

eIX

• – ◦ • – ◦ • – ◦ •

Introduction

The eIX function block is an input service interface for Boolean input data with event-based triggering. It serves as an interface between the control logic and physical input devices and enables the querying of digital input signals.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the function block with parameters
- **REQ**: Service Request - Triggers a query of the input signal

Event Outputs

- **INITO**: Initialization Acknowledgement - Confirms successful initialization
- **CNF**: Service Acknowledgement - Confirms the processed service request
- **IND**: Resource Indication - Signals a status change of the input

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Enables/disables processing
- **PARAMS** (STRING): Service Parameters - Configuration parameters for the service

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Processing status
- **STATUS** (STRING): Service status - Error or status information
- **IN** (BOOL): Input data from the resource - Current value of the digital input

Adapter

- **eCONF**: Configuration of event-based triggering - Adapter for trigger configuration

Functionality

The eIX block acts as a service interface for digital inputs. During initialization (INIT), the service parameters are configured. Targeted queries of the input signal can be triggered via REQ events. The block returns the current input values via CNF and can report asynchronous status changes via IND.

Technical Features

- Event-based query mechanisms
- Separate initialization and operating modes
- String-based status feedback
- Adapter-based configuration of trigger logic
- Support for both requested and indexed operations

State Overview

1. **Not Initialized**: Block waits for INIT event
2. **Initialized**: Block is ready for operation and can process REQ events
3. **Active**: Processing service requests and providing feedback of input values
4. **Error**: In case of problems, STATUS is populated with error information

Application Scenarios

- Querying digital sensors (limit switches, proximity switches)
- Monitoring switching states
- Integration of pushbuttons and control elements
- Signal processing in automation systems
- Condition monitoring of machines and systems

⚖️ Comparison with Similar Blocks

Compared to simple digital input blocks, eIX offers extended functionality:

- Event-based instead of cyclic querying
- Detailed Status feedback
- Configurable trigger mechanisms
- Separate initialization and operating phases

Conclusion

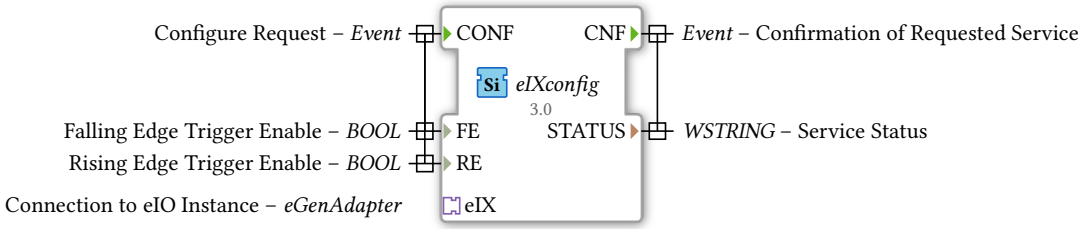
The eIX function block represents a flexible and robust solution for integrating digital inputs into IEC 61499-based control systems. Its event-based architecture and comprehensive status feedback make it particularly suitable for applications requiring reliable and configurable input processing.

eIXconfig

• – ◦ • – ◦ • – ◦ •

Introduction

The eIXconfig function block serves as a Service Interface Function Block for configuring eIO instances. It allows you to set trigger properties for inputs and outputs and provides a connection to eIO instances via an adapter.



Interface Structure

Event Inputs

- **CONF**: Configuration Request - triggers the configuration of the eIO instance

Event Outputs

- **CNF**: Confirmation of the requested service - signals the completion of the configuration

Data Inputs

- **FE** (BOOL): Falling Edge Trigger Enable - enables edge triggering on a falling edge
- **RE** (BOOL): Rising Edge Trigger Enable - enables edge triggering on a rising edge

Data Outputs

- **STATUS** (WSTRING): Service Status - provides status information about the configuration process

Adapter

- **eIX**: Connection to the eIO instance - establishes communication with the eIO hardware

Functionality

The eIXconfig block receives a Configuration request. Along with the data inputs FE and RE, the trigger settings for the eIO instance are configured. After successful configuration, the CNF event is output with the corresponding status. Actual communication with the hardware takes place via the eIX adapter.

Technical Features

- Uses WSTRING for status outputs, enabling comprehensive status information
- Supports separate configuration of rising and falling edge detection
- Implemented as a Service Interface Function Block according to the IEC 61499-2 standard

State Transitions

1. **Initialization State**: Block waits for a CONF event

2. **Configuration State**: Processes trigger settings and transmits them to the eIO instance
3. **Acknowledgement State**: Outputs CNF with status information

Application Scenarios

- Configuration of eIO hardware components in industrial control systems
- Setting trigger properties for digital inputs
- Integration into larger automation systems with eIO components

⚖️ Comparison with Similar Blocks

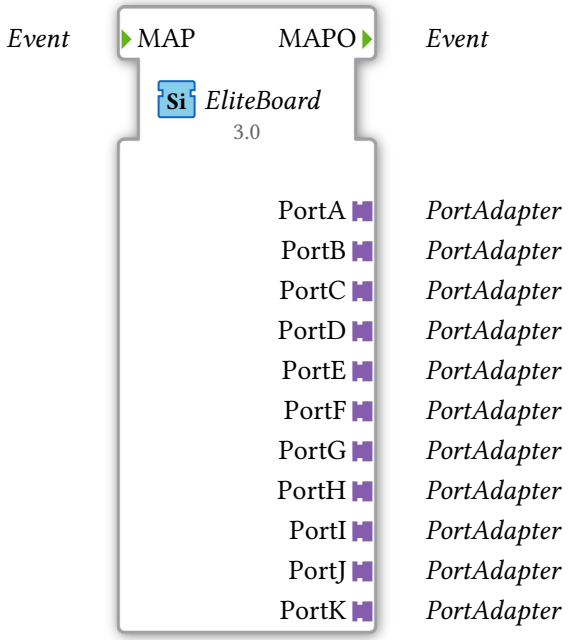
Compared to simple IO configuration blocks, eIXconfig offers specific eIO integration and advanced trigger configuration options. Using an adapter allows for a clean separation between functionality and hardware communication.

Conclusion

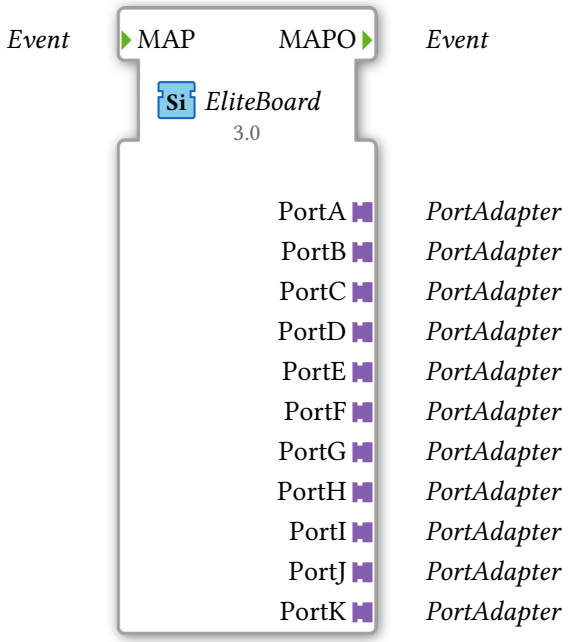
The eIXconfig function block represents a specialized solution for configuring eIO instances. Its clear separation of configuration logic and hardware communication, along with flexible trigger settings, makes it ideally suited for demanding industrial applications.

2.11.5.19 Eliteboard

eliteboard



EliteBoard



• – ◦ • – ◦ • – ◦ •

Introduction

The EliteBoard function block is a Service Interface Function Block Type that serves as an interface for the EliteBoard hardware. This block enables the connection and control of up to 11 different ports via standardized adapter interfaces.

Interface Structure

Event Inputs

- **MAP**: Initializes the mapping process for the configured ports

Event Outputs

- **MAPO**: Signals the successful completion of the mapping process

Data Inputs

No data inputs available

Data Outputs

No data outputs available

Adapters

The function block has 11 adapter interfaces (plugs):

- **PortA** to **PortK**: Each of type `eclipse4diac::io::eliteboard::PortAdapter`

These adapters enable connection to various port interfaces of the EliteBoard hardware.

Functionality

Upon receiving the MAP event, the EliteBoard function block initializes the configured port adapters. After successful initialization of all connected ports, the MAPO event is triggered. Each port adapter (Port A to Port K) can be individually configured for specific input/output functions.

Technical Features

- Supports up to 11 independent port interfaces
- Uses standardized port adapter interfaces
- Simple initialization via a single event

- Robust error handling through centralized mapping

State Overview

1. **Initial State**: Waiting for MAP event
2. **Mapping State**: Initializing all configured port adapters
3. **Ready State**: All ports are initialized, MAPO event is triggered

Application Scenarios

- Control of EliteBoard hardware in industrial automation systems
- Connecting multiple input/output interfaces in embedded systems
- Integration into distributed control systems according to IEC 61499
- Hardware-level control tasks with modular port management

⚖️ Comparison with similar devices

Compared to simple I/O devices, EliteBoard offers structured management of multiple ports via standardized adapter interfaces. While simple I/O modules typically use direct data inputs/outputs, EliteBoard relies on an adapter-based architecture for improved modularity and reusability.

Conclusion

The EliteBoard function block provides a flexible and scalable solution for connecting EliteBoard hardware to 4diac-based control systems. By using standardized adapter interfaces and supporting multiple ports, it enables a modular and maintainable system architecture for industrial automation applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

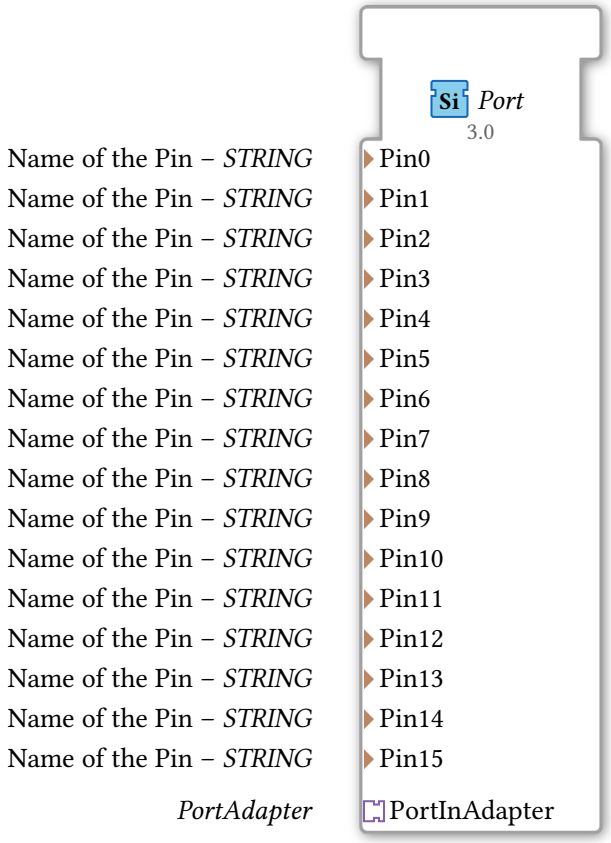
Port

🎧 Podcast

- The Technology Panorama of 1863: Lanz & Comp. and the Revolution of German Agriculture through Import, Innovation, and Guano
- ESP32-S3-DevKitC-1 Document Analysis: The Memory Monster (32MB Flash/16MB PSRAM) and the Power of Dual USB Ports

Introduction

The Port Function Block serves as a Service Interface Function Block for configuring and managing digital inputs/outputs on an EliteBoard system. It enables the mapping of pin names to physical ports via an adapter mechanism.



Interface Structure

Event Inputs

No event inputs available

Event Outputs

No event outputs available

Data Inputs

- **Pin0** (STRING): Name of Pin 0
- **Pin1** (STRING): Name of Pin 1
- **Pin2** (STRING): Name of Pin 2
- **Pin3** (STRING): Name of Pin 3
- **Pin4** (STRING): Name of Pin 4
- **Pin5** (STRING): Name of Pin 5
- **Pin6** (STRING): Name of Pin 6
- **Pin7** (STRING): Name of Pin 7
- **Pin8** (STRING): Name of Pin 8
- **Pin9** (STRING): Name of Pin 9
- **Pin10** (STRING): Name Pin 10
- **Pin 11** (STRING): Name of Pin 11
- **Pin 12** (STRING): Name of Pin 12
- **Pin 13** (STRING): Name of Pin 13
- **Pin 14** (STRING): Name of Pin 14
- **Pin 15** (STRING): Name of Pin 15

Data Outputs

No data outputs available

Adapters

- **PortInAdapter** (eclipse4diac::io::eliteboard::PortAdapter): Socket adapter for communication with the port subsystem

Functionality

The Port FB serves as a configuration interface for up to 16 digital pins. Individual pin names can be assigned via the STRING inputs, which are then passed on to the underlying system via the PortAdapter. The function block itself does not perform any direct input/output operations, but merely provides the configuration parameters.

Technical Features

- Supports configuration of 16 digital pins
- Uses the STRING data type for pin names
- Implemented as a Service Interface Function Block
- Uses the adapter pattern for hardware abstraction
- Part of the eclipse4diac::io::eliteboard package

State Overview

Since this is a pure configuration block without event handling, the Port FB has no explicit states. The configuration is set statically via the data inputs.

Application Scenarios

- Configuration of digital inputs/outputs on EliteBoard systems
- Abstraction of hardware pins using logical names
- Integration into larger control systems with unified pin management
- Reusable pin configurations across various applications

⚖️ Comparison with similar components

Compared to direct GPIO components, the Port FB offers a higher level of abstraction by using logical pin names instead of direct hardware addressing. It primarily serves configuration purposes, while other components typically provide the actual input/output functionality.

Conclusion

The Port Function Block provides an elegant solution for configuration-based pin management in Eclipse 4diac systems. By using adapters, it enables a clear separation between configuration and hardware access, improving the maintainability and portability of control applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 ESP32 & ESP32-S3 DevKit on ms-muc-docs.de

]

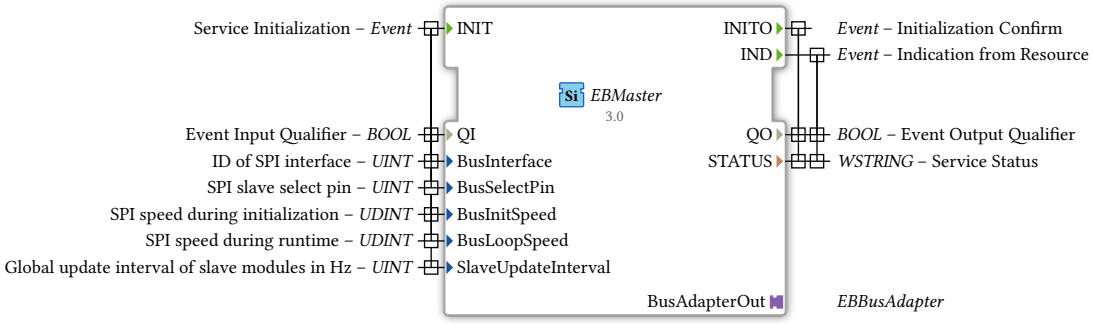
2.11.5.20 embrick

EBMaster

• – ◦ • – ◦ • – ◦ •

Introduction

The EBMaster function block is a Service Interface Function Block for communication with EtherBrick modules via SPI (Serial Peripheral Interface). It serves as the master component for controlling and configuring SPI bus communication with connected slave modules.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - Initializes the SPI bus and configures the communication parameters

Event Outputs

- **INITO**: Initialization Acknowledgement - Confirms successful completion of initialization
- **IND**: Resource Indication - Signals status changes or events from the connected modules

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Enables/Disables the function block
- **BusInterface** (UINT): ID of the SPI interface (initial value: 1)
- **BusSelectPin** (UINT): SPI Slave Select Pin (initial value: 49)
- **BusInitSpeed** (UDINT): SPI speed during initialization in Hz (initial value: 300000)
- **BusLoopSpeed** (UDINT): SPI speed during operation in Hz (initial value: 700000)
- **SlaveUpdateInterval** (UINT): Global update interval of the slave modules in Hz (initial value: 25)

Data Outputs

- **QO** (BOOL): Event Output Qualifier - status of the event output
- **STATUS** (WSTRING): Service Status - detailed status information

Adapters

- **BusAdapterOut**: Adapter for communication with the EtherBrick bus system

Functionality

The EBMaster manages SPI communication as the master component. During initialization (INIT event), it configures the SPI interface with the specified parameters. During operation, it controls the data transmission to the connected slave modules at the configured update interval. The block allows for different speeds during the initialization and operation phases.

Technical Features

- Support for various SPI speeds for initialization and operation
- Configurable slave select pins
- Adjustable global update interval for slave modules
- WSTRING-based status feedback for detailed fault diagnosis

State Overview

The function block goes through the following states:

1. **Not Initialized**: Waits for INIT event
2. **Initialization**: Configures SPI parameters upon INIT event
3. **Ready**: Sends INITO upon successful initialization
4. **Active**: Communicates with slaves according to the configured interval
5. **Fault**: Sends an IND event with STATUS information in case of problems

Application Scenarios

- Control of EtherBrick I/O modules in automation systems
- SPI-based communication in embedded systems

- Industrial controllers with modular I/O expansions
- Real-time data acquisition with configurable Update Rates

⚖️ Comparison with Similar Blocks

Compared to generic SPI master blocks, EBMaster offers specific optimizations for EtherBrick modules, including pre-configured parameter values and specialized bus adapter interfaces.

Conclusion

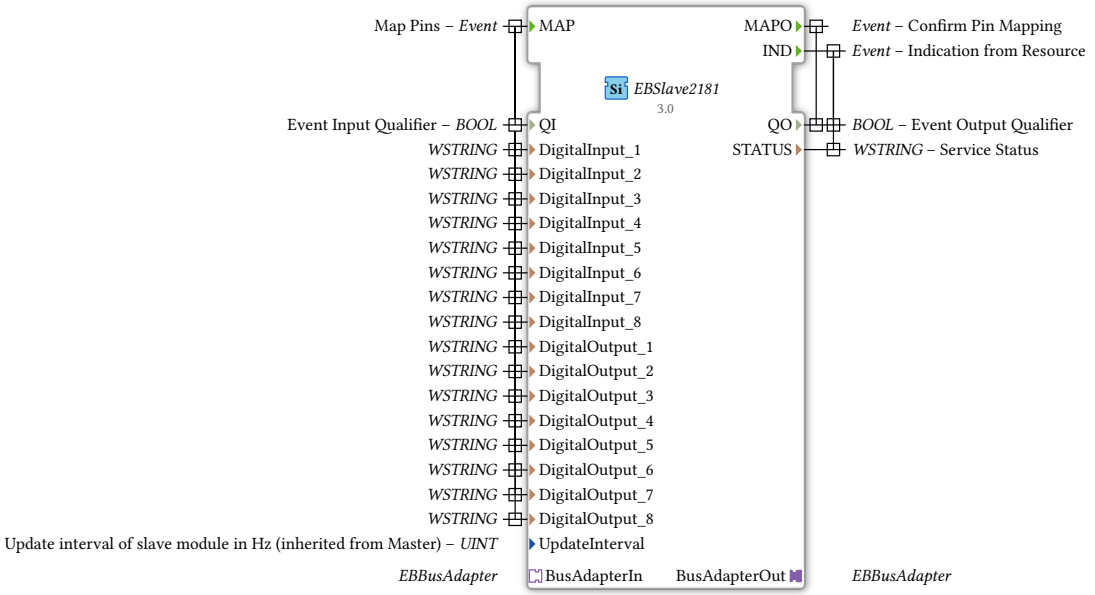
The EBMaster function block provides a robust and configurable solution for SPI master communication with EtherBrick modules. By separating initialization and operating speeds and offering a configurable update interval, it enables optimal adaptation to various application requirements.

EBSlave2181

• – ◦ • – ◦ • – ◦ •

Introduction

The EBSlave2181 is a Service Interface Function Block for communication with EtherBrick slave modules. This function block enables the configuration and monitoring of the digital inputs and outputs of an EtherBrick slave module and serves as an interface between the control logic and the physical fieldbus communication.



Interface Structure

Event Inputs

- **MAP**: Used to configure pin assignments. Triggers the assignment of digital inputs and outputs.

Event Outputs

- **MAPO**: Confirms successful pin assignment.
- **IND**: Indicates status changes or errors from the slave module.

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Enables/disables the function block
- **DigitalInput_1** to **DigitalInput_8** (WSTRING): Configuration of digital inputs 1-8
- **DigitalOutput_1** to **DigitalOutput_8** (WSTRING): Configuration of digital outputs 1-8
- **UpdateInterval** (UINT): Update interval of the slave module in Hz (inherited from the master)

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Status of the function block
- **STATUS** (WSTRING): Service Status - Contains status information or error messages

Adapters

- **BusAdapterOut** (Plug): Outgoing bus adapter for EtherBrick communication
- **BusAdapterIn** (Socket): Incoming bus adapter for EtherBrick communication

Functionality

The The EBSlave2181 acts as an intermediary between the IEC 61499 control logic and an EtherBrick slave module. Upon receiving the MAP event, the configured digital inputs and outputs are assigned to the slave module. The MAPO event confirms the successful completion of this assignment. The IND event signals status changes or error states of the slave module.

Technical Features

- Supports 8 digital inputs and 8 digital outputs
- Uses WSTRING data type for pin configuration
- Enables configurable update intervals
- Integrates into the EtherBrick architecture via standardized bus adapters
- Implements qualifier patterns (QI/QO) for reliable status management

State Overview

The function block has the following operating states:

- **Inactive**: QI = FALSE, function block does not respond to events
- **Configuring**: Processing the MAP event and assigning pins
- **Active**: Successfully configured, waiting for IND events from the slave
- **Error**: STATUS contains error information, QO can be FALSE

Application Scenarios

- Connecting EtherBrick slave modules in distributed automation systems
- Configuring digital I/O modules in industrial plants
- Integration into control systems with EtherBrick Fieldbus Communication

Real-time Monitoring and Diagnostics of Slave Modules

⚖️ Comparison with Similar Components

Compared to generic I/O function blocks, the EBSlave2181 offers specific integration for EtherBrick systems with predefined bus adapter interfaces. It specializes in the configuration and monitoring of slave modules and provides direct support for EtherBrick communication protocols.

Conclusion

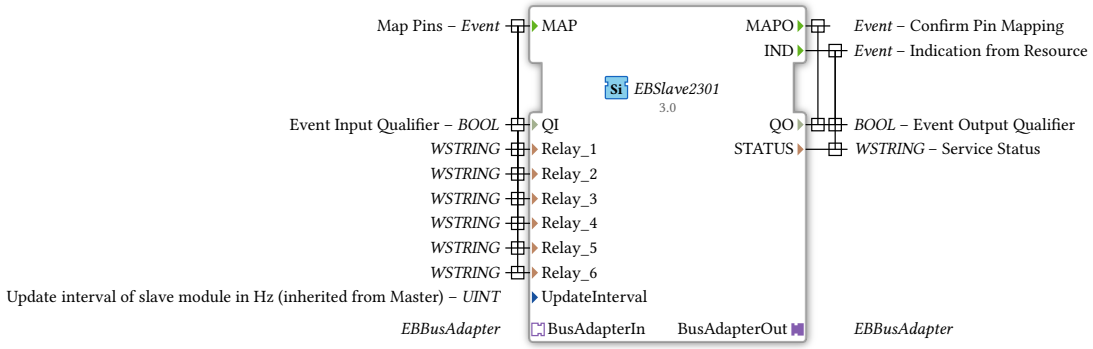
The EBSlave2181 is a specialized service interface function block that provides a reliable and standardized interface for integrating EtherBrick slave modules into IEC 61499-based control systems. Its structured interface and clear state management make it a robust solution for industrial fieldbus communication.

EBSlave2301

• – ◦ • – ◦ • – ◦ •

Introduction

The EBSlave2301 is a Service Interface Function Block for communication with an EtherBrick slave module. This function block serves as an interface for configuring and controlling up to six relay outputs via an EtherBrick bus system.



Interface Structure

Event Inputs

- **MAP**: Triggered to configure the pin assignments. Accompanied by the data variables Relay_1 to Relay_6 and QI.

Event Outputs

- **MAPO**: Confirms successful pin assignment. Outputs with QO.
- **IND**: Message from the resource interface. Outputs with QO and STATUS.

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Enables/disables the function block
- **Relay_1** to **Relay_6** (WSTRING): Relay pin configuration
- **UpdateInterval** (UINT): Slave module update interval in Hz (inherited from the master)

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Operation status
- **STATUS** (WSTRING): Service status information

Adapters

- **BusAdapterOut** (Plug): Outgoing bus adapter of type EBBusAdapter
- **BusAdapterIn** (Socket): Incoming bus adapter of type EBBusAdapter

Functionality

The EBSlave2301 allows the configuration of up to six relay outputs via the MAP event. When activated by QI, the relay parameters are transmitted to the EtherBrick system via the bus adapters. The MAPO event confirms successful configuration, while IND events provide status information from the resource interface.

Technical Features

- Supports WSTRING data types for relay configuration
- Uses an adapter-based bus system for communication
- UpdateInterval is inherited from the master system
- Provides qualified event control via QI and QO

State Overview

The function block toggles between the following states:

- Inactive: QI = FALSE
- Configuration mode: On MAP event with QI = TRUE
- Ready for operation: After successful MAPO acknowledgment
- Fault state: On IND event with status information

Application Scenarios

- Control of relay-based actuators in automation systems
- Integration into EtherBrick-based I/O systems
- Distributed control system with master-slave architecture

- Applications with configurable relay assignments

⚖️ Comparison with Similar Devices

Compared to simple digital output devices, the EBSlave2301 offers:

- Configurable pin assignments for Runtime
- Adapter-based bus communication
- Qualified event control
- Status feedback from the resource interface

Conclusion

The EBSlave2301 is a specialized function block for the flexible control of relay outputs in EtherBrick systems. Its configurable interfaces and adapter-based architecture make it particularly suitable for demanding automation applications with dynamic I/O configurations.

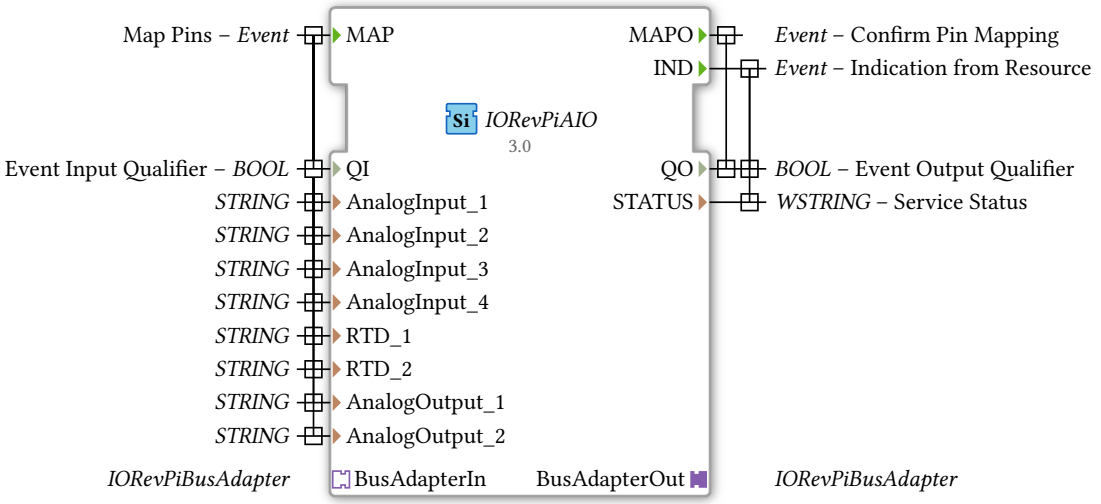
2.11.5.21 revpi

🔌 IORevPiAIO

• – ◦ • – ◦ • – ◦ •

Introduction

The IORevPiAIO function block provides an interface for the analog I/O module of the Revolution Pi from KUNBUS GmbH. This module enables the control and reading of analog inputs and outputs, as well as RTD (Resistance Temperature Detector) sensors, via the Revolution Pi system.



Interface Structure

Event Inputs

- **MAP**: Starts the pin mapping for all configured analog inputs and outputs

Event Outputs

- **MAPO**: Confirms successful pin mapping
- **IND**: Displays status information from the resource manager

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Enables/Disables the function block
- **AnalogInput_1** (STRING): Configuration for analog input 1
- **AnalogInput_2** (STRING): Configuration for analog input 2
- **AnalogInput_3** (STRING): Configuration for analog input 3
- **AnalogInput_4** (STRING): Configuration for analog input 4
- **RTD_1** (STRING): Configuration for RTD sensor 1
- **RTD_2** (STRING): Configuration for RTD Sensor 2

- **AnalogOutput_1** (STRING): Configuration for Analog Output 1
- **AnalogOutput_2** (STRING): Configuration for Analog Output 2

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Operation Status
- **STATUS** (WSTRING): Detailed Service Status Information

Adapters

- **BusAdapterIn** (Socket): Input adapter for Revolution Pi bus communication
- **BusAdapterOut** (Plug): Output adapter for Revolution Pi bus communication

Functionality

The IORevPiAIO function block manages communication with the Revolution Pi Analog I/O module. Upon receiving the MAP event, all configured analog inputs and outputs, as well as RTD sensors, are initialized and assigned according to the string parameters. The block uses dedicated bus adapters for communication with the Revolution Pi hardware.

Technical Features

- Supports up to 4 analog inputs
- Supports up to 2 analog outputs
- Integrated RTD sensor support (2 channels)
- String-based pin mapping configuration
- Bus adapter architecture for low-level communication

State Overview

The function block has the following operating states:

- **Inactive**: QI = FALSE, no operations
- **Ready**: QI = TRUE, waiting for MAP event
- **Mapping**: Processes pin mapping after MAP event
- **Active**: Successfully configured, ready for data operations

Application Scenarios

- Industrial process automation with analog sensors
- Temperature measurement with RTD sensors
- Analog signal processing in control systems
- Revolution Pi-based automation solutions

⚖️ Comparison with Similar Blocks

Compared to generic analog I/O blocks, IORevPiAIO offers specific integration for Revolution Pi hardware and additionally supports RTD temperature sensors. The bus adapter architecture enables efficient communication with the Revolution Pi system.

Conclusion

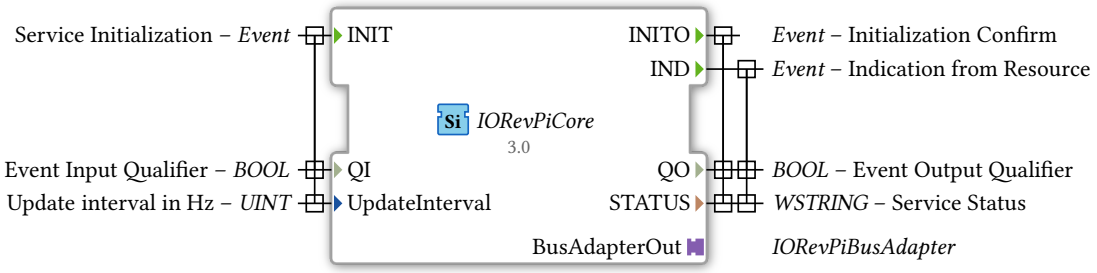
The IORevPiAIO function block offers a reliable and specialized interface for analog I/O operations on Revolution Pi systems. With its integrated support for RTD sensors and flexible configuration via string parameters, it is particularly well-suited for industrial automation applications with analog measurement and control tasks.

🔌 IORevPiCore

• – ◦ • – ◦ • – ◦ •

Introduction

The IORevPiCore function block serves as the core module for the Revolution Pi hardware from KUNBUS GmbH. It enables basic communication and configuration of the Revolution Pi I/O modules within a 4diac-based control application.



Interface Structure

Event Inputs

- **INIT**: Initialization event to activate the function block

Event Outputs

- **INITO**: Confirmation of successful initialization
- **IND**: Indication event for status changes of resources

Data Inputs

- **QI** (BOOL): Event input qualifier
- **UpdateInterval** (UINT): Update interval in Hz (default: 25)

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (WSTRING): Service status as a Unicode string

Adapters

- **BusAdapterOut**: Output adapter for Revolution Pi bus communication (Type: eclipse4diac::io::revpi::IORevPiBusAdapter)

Functionality

The The IORevPiCore block initiates communication with the Revolution Pi hardware upon receiving the INIT event. After successful initialization, INITO is output with the corresponding status. The block continuously monitors the hardware state and outputs IND events when changes occur. The UpdateInterval determines the sampling rate for hardware monitoring.

Technical Features

- Standard update interval of 25 Hz
- Unicode string output for status information
- Integration via a specific bus adapter for Revolution Pi hardware
- Automatic hardware detection and initialization

State Overview

1. **Inactive**: Block awaits INIT event
2. **Initialization**: Processing configuration parameters
3. **Active**: Monitoring Revolution Pi hardware and outputting IND events upon status changes

Application Scenarios

- Controlling Revolution Pi I/O modules in industrial automation solutions
- Integrating Revolution Pi hardware into 4diac-based control systems
- Monitoring and diagnostics of Revolution Pi components

⚖️ Comparison with Similar Blocks

Compared to generic I/O blocks, IORevPiCore offers specific optimizations for the Revolution Pi hardware platform, including specialized bus communication and hardware-level configuration parameters.

Conclusion

The IORevPiCore function block provides a reliable and easy-to-configure interface for integrating Revolution Pi hardware into 4diac-based automation solutions. Its standardized interface and flexible configuration options enable efficient integration of industrial I/O components.

🌐 Related topic subpages on ms-muc-docs.de

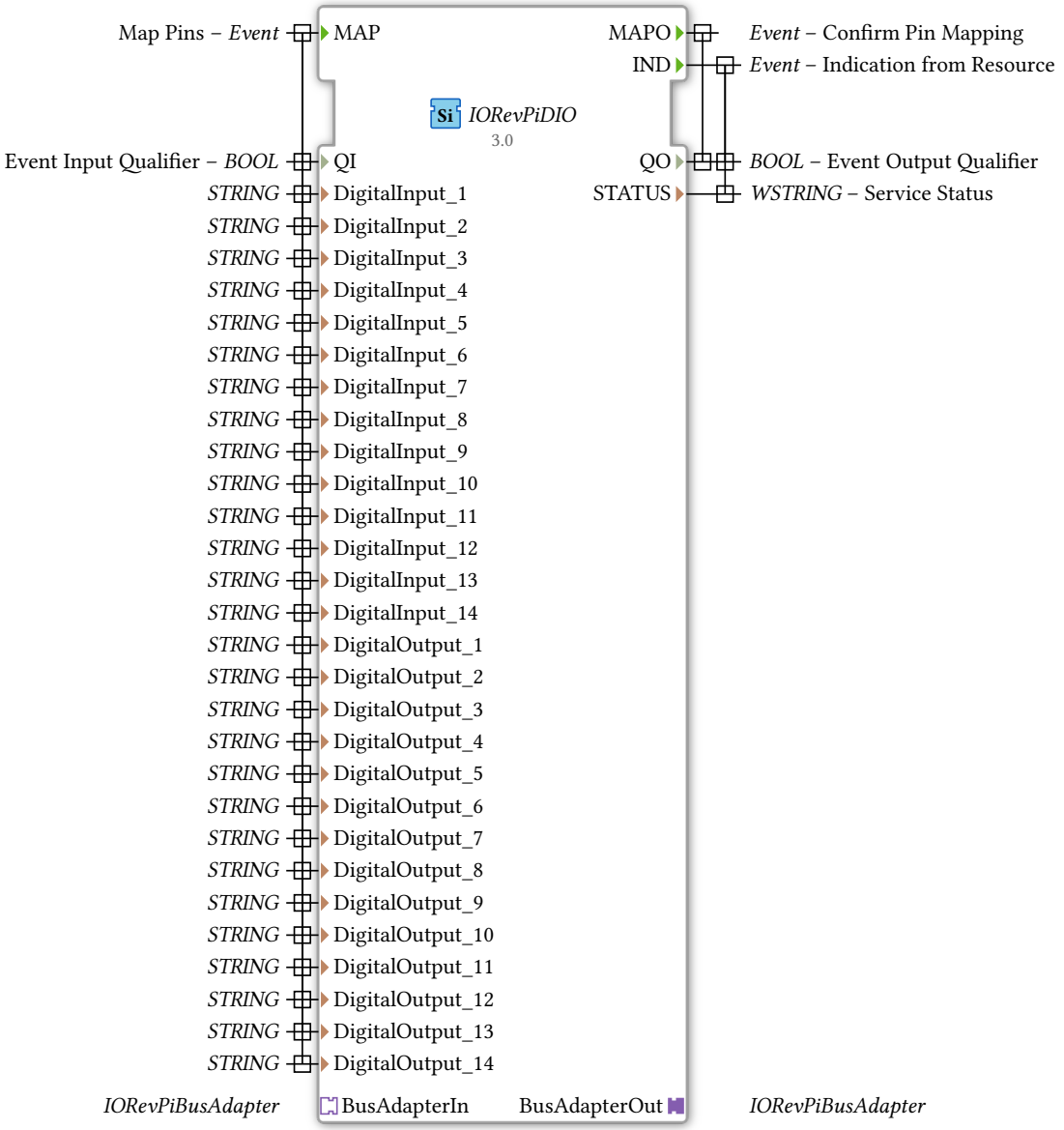
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

🔌 IORevPiDIO

• – ◦ • – ◦ • – ◦ •

Introduction

The IORevPiDIO function block is a digital input/output module for Revolution Pi systems from KUNBUS GmbH. It enables the control and monitoring of digital inputs and outputs via the Revolution Pi system and serves as an interface between the 4diac IDE and the Revolution Pi hardware.



Interface Structure

Event Inputs

- **MAP**: Event for mapping the pins with all Digital Input and Digital Output data variables, as well as QI

Event Outputs

- **MAPO**: Confirmation event for successful pin mapping with QO
- **IND**: Indication event from the resource with QO and STATUS

Data Inputs

- **QI** (BOOL): Event Input Qualifier
- **DigitalInput_1** to **DigitalInput_14** (STRING): Configuration of digital inputs 1-14
- **DigitalOutput_1** to **DigitalOutput_14** (STRING): Configuration of digital outputs 1-14

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (WSTRING): Service Status

Adapter

- **BusAdapterOut** (Plug): Outgoing bus adapter of type IORevPiBusAdapter
- **BusAdapterIn** (Socket): Incoming bus adapter of type IORevPiBusAdapter

Functionality

This function block enables the configuration and control of up to 14 digital inputs and 14 digital outputs of a Revolution Pi system. The MAP event transmits the pin configurations, which are confirmed with MAPO. The IND output signals status changes and error states. Communication with the hardware occurs via the IORevPiBusAdapter interface.

Technical Features

- Supports up to 14 digital inputs and 14 digital outputs
- Uses STRING type for pin configuration
- Integrated bus adapter for Revolution Pi communication
- Provides comprehensive status feedback via WSTRING

State Transitions

1. **Initialization**: Waits for a MAP event with configuration data
2. **Configuration**: Processes pin assignments and confirms via MAPO
3. **Operation**: Monitors digital inputs/outputs and signals via IND
4. **Fault Handling**: Provides status messages in case of communication problems with the hardware

Application Scenarios

- Industrial automation with Revolution Pi
- Digital signal processing in control systems
- Connecting sensors and actuators to 4diac-based controllers
- Prototyping and development of IoT solutions

⚖️ Comparison with Similar Devices

Compared to other I/O devices, IORevPiDIO offers specific support for Revolution Pi hardware with a large number of configurable Inputs/Outputs. The integrated bus adapter enables direct communication with the Revolution Pi platform.

Conclusion

The IORevPiDIO function block provides a powerful interface for digital inputs/outputs in Revolution Pi systems and allows for easy integration into 4diac-based automation solutions. Its extensive configuration options and reliable status feedback make it a robust solution for industrial applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

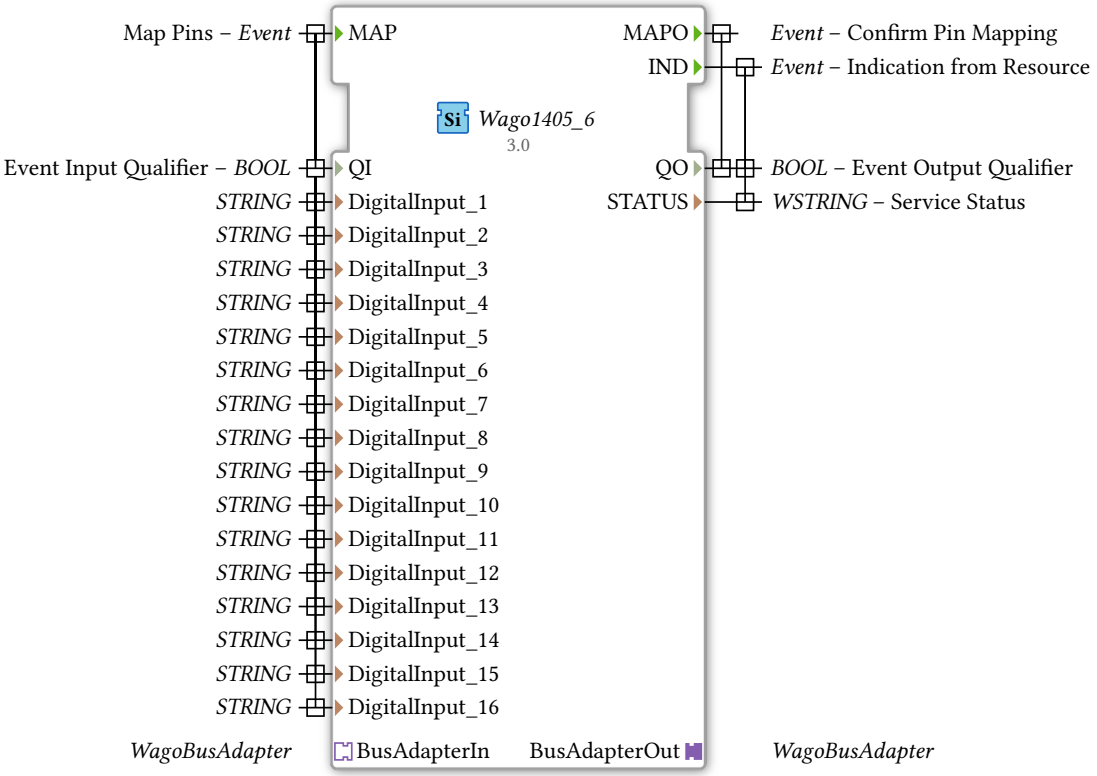
2.11.5.22 wago

Wago1405_6

• – ◦ • – ◦ • – ◦ •

Introduction

The Wago1405_6 is a Service Interface Function Block for connecting to Wago bus systems. This function block serves as an interface for configuring and controlling the digital inputs of a Wago I/O system and enables communication via special bus adapters.



Interface Structure

Event Inputs

- **MAP**: Map Pins - Used to configure pin assignments for digital inputs

Event Outputs

- **MAPO**: Confirm Pin Mapping - Confirms successful pin mapping
- **IND**: Indication from Resource - Message from the resource interface

Data Inputs

- **QI**: Event Input Qualifier (BOOL) - Qualifies the event inputs
- **DigitalInput_1** to **DigitalInput_16**: Pin Configuration (STRING) - Assigns digital inputs 1-16

Data Outputs

- **QO**: Event Output Qualifier (BOOL) - Qualifies the event outputs
- **STATUS**: Service Status (WSTRING) - Status information of the service interface

Adapters

- **BusAdapterOut**: Outgoing bus adapter (plug) of type eclipse4diac::io::wago::WagoBusAdapter
- **BusAdapterIn**: Incoming bus adapter (socket) of type eclipse4diac::io::wago::WagoBusAdapter

Functionality

The Wago1405_6 function block allows the configuration of 16 digital inputs via the MAP event. When the MAP event is activated, the pin assignments are configured using the DigitalInput_x parameters. Successful configuration is confirmed via the MAPO event. The IND event is used for asynchronous communication with the underlying resource interface and provides status information.

Technical Features

- Supports 16 digital inputs
- Uses STRING type for pin configuration
- Implements bidirectional bus adapter communication
- Provides comprehensive status feedback via WSTRING

State Overview

The function block operates with the following main states:

1. **Inactive**: Waiting for MAP event
2. **Configuration**: Processing pin assignments
3. **Acknowledge**: Sending MAPO acknowledgement
4. **Operation**: Asynchronous status messages via IND

Application Scenarios

- Connecting Wago 750-1405/750-1406 I/O systems
- Configuring digital inputs in automation systems
- Integration into distributed control systems with 4diac
- Industrial I/O configuration and monitoring

⚖️ Comparison with Similar Blocks

Compared to generic I/O function blocks, the Wago1405_6 offers specific Adaptations for Wago bus systems and more extensive configuration options for digital inputs. The integration of the bus adapters enables seamless integration into Wago-specific communication architectures.

Conclusion

The Wago1405_6 function block represents a specialized solution for connecting and configuring Wago I/O systems. With its extensive configuration options for 16 digital inputs and the integrated bus adapter interface, it offers an efficient solution for industrial automation applications using Wago components.

🌐 Related topic subpages on ms-muc-docs.de

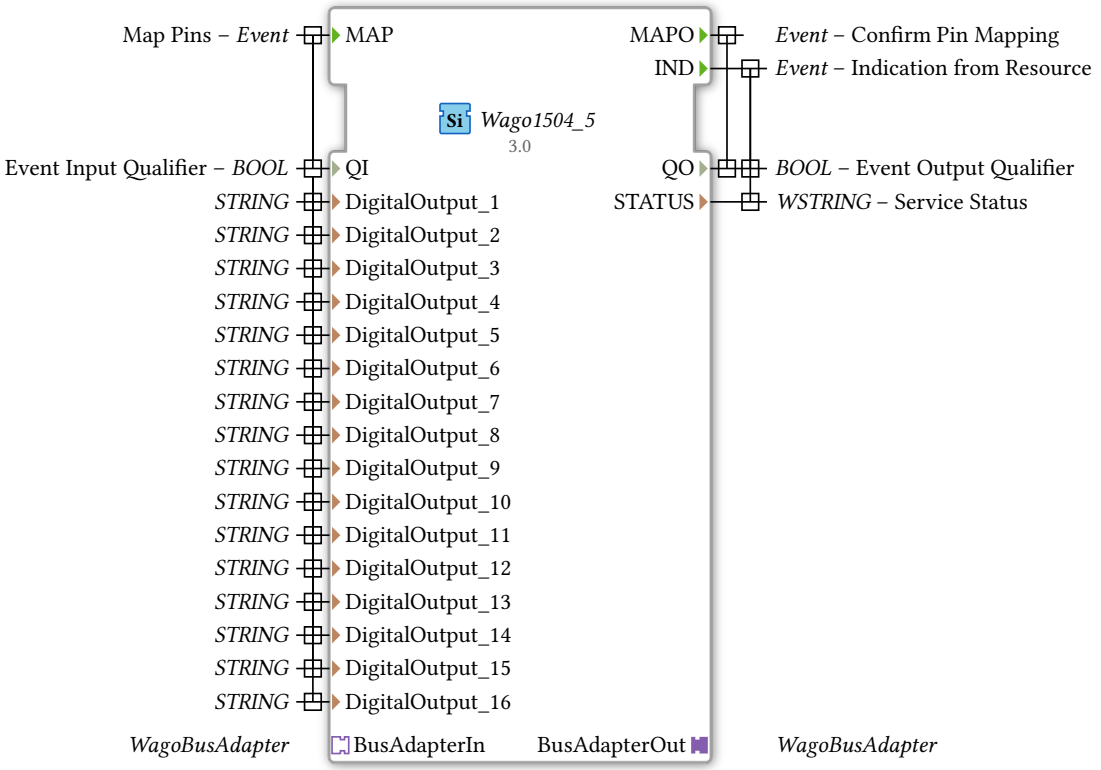
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

Wago1504_5

• – ◦ • – ◦ • – ◦ •

Introduction

The Wago1504_5 is a Service Interface Function Block for controlling the digital outputs of a Wago 750-1504/005-000 module. This function block enables the configuration and control of 16 digital outputs via a bus adapter interface.



Interface Structure

Event Inputs

- **MAP**: Map Pins - Initializes the pin mapping of the digital outputs

Event Outputs

- **MAPO**: Confirm Pin Mapping - Confirms successful pin mapping
- **IND**: Indication from Resource - Message from the resource adapter

Data Inputs

- **QI**: Event Input Qualifier (BOOL) - Qualifies the event inputs
- **DigitalOutput_1** to **DigitalOutput_16** (STRING) - Configuration strings for the respective digital outputs

Data Outputs

- **QO**: Event Output Qualifier (BOOL) - Qualifies the event outputs
- **STATUS**: Service Status (WSTRING) - Service status information

Adapters

- **BusAdapterOut** (Plug): Outgoing WagoBusAdapter
- **BusAdapterIn** (Socket): Incoming WagoBusAdapter

Functionality

The Wago1504_5 function block serves as an interface between the 4diac controller and the Wago 750-1504/005-000 I/O module. Upon receiving the MAP event, the 16 digital outputs are assigned according to the transmitted string configuration. Successful assignment is confirmed by the MAPO event. The IND output is used to provide feedback of status information from the resource adapter.

Technical Features

- Supports 16 independent digital outputs
- Uses string-based configuration for pin assignment
- Implements bus adapter communication for connection to Wago hardware

- Provides status feedback via WSTRING data type

State Overview

The function block cycles through the following states:

1. **Inactive**: Waiting for MAP event
2. **Mapping**: Processing pin assignment
3. **Acknowledge**: Sends MAPO upon successful assignment
4. **Ready**: Ready for IND messages

Application Scenarios

- Control of relays and actuators in automation systems
- Digital signal output in process controllers
- Integration of Wago 750-1504/005-000 modules into 4diac-based controllers
- Industrial automation with distributed I/O systems

⚖️ **Comparison with Similar Function Blocks**

Compared to generic digital The Wago1504_5 output block offers:

- Specific adaptation for Wago hardware
- Extended configuration options via string parameters
- Integrated bus adapter communication
- Status feedback with detailed information

Conclusion

The Wago1504_5 function block provides a reliable and specialized interface for controlling Wago 750-1504/005-000 digital output modules. Its structured event control and comprehensive status feedback make it particularly suitable for robust industrial automation solutions.

–

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

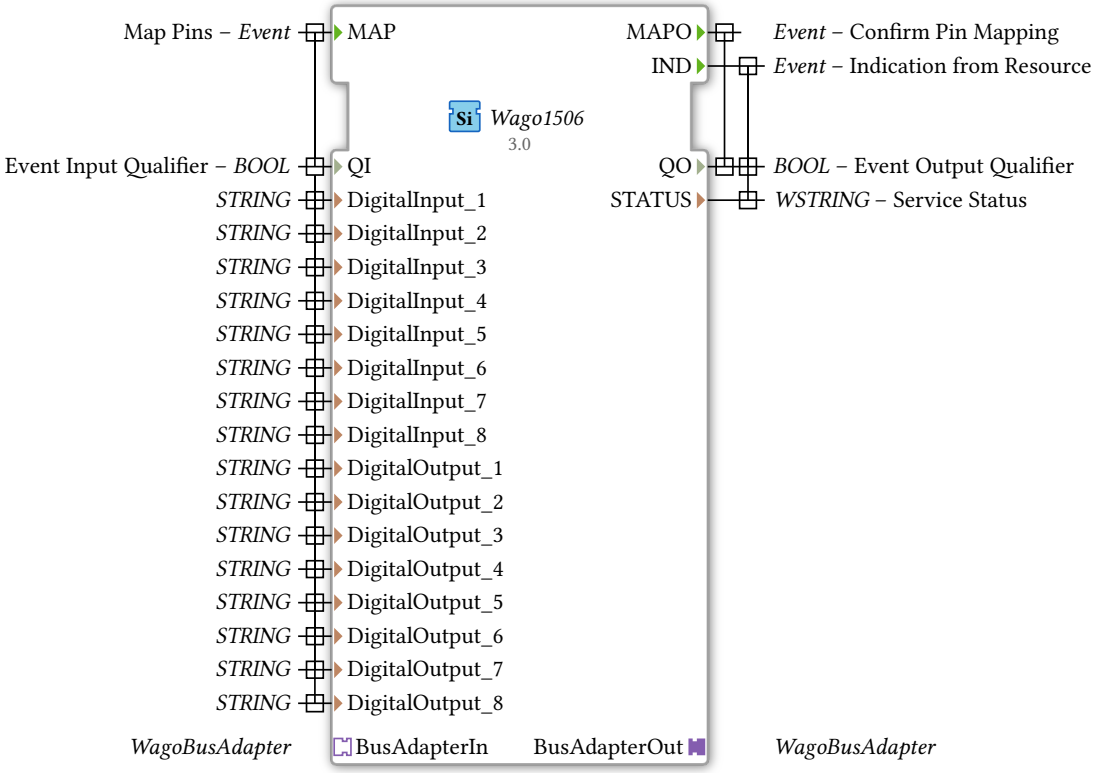
]

Wago1506

• – ◦ • – ◦ • – ◦ •

Introduction

The Wago1506 is a Service Interface Function Block for controlling Wago 750-1506 I/O modules. This function block enables the configuration and control of digital inputs and outputs via a bus adapter and serves as an interface between the control logic and the physical hardware.



Interface Structure

Event Inputs

- **MAP**: Used to configure the pin assignment. Triggers the mapping of the digital inputs and outputs.

Event Outputs

- **MAPO**: Confirms successful pin mapping.
- **IND**: Displays status information from the resource.

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Enables/disables the function block
- **DigitalInput_1** to **DigitalInput_8** (STRING): Configuration of digital inputs 1-8
- **DigitalOutput_1** to **DigitalOutput_8** (STRING): Configuration of digital outputs 1-8

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Operation status
- **STATUS** (WSTRING): Detailed service status information

Adapters

- **BusAdapterOut** (Plug): Outgoing bus adapter for Wago communication
- **BusAdapterIn** (Socket): Incoming bus adapter for Wago communication

Functionality

The Wago1506 FB manages communication with a Wago 750-1506 I/O module. Upon receiving the MAP event, the configured pin assignments are passed to the bus adapter interface. The function block monitors the hardware status and outputs IND events with corresponding status information when changes occur.

Technical Features

- Supports 8 digital inputs and 8 digital outputs
- Uses STRING types for pin configuration, enabling flexible addressing
- Integrates specific Wago bus adapters for reliable hardware communication
- Provides comprehensive status feedback via WSTRING

Status Overview

1. **Inactive**: QI = FALSE, FB awaits activation
2. **Configured**: After a MAP event with successful pin assignment
3. **Active**: QI = TRUE, FB communicates with hardware
4. **Error**: In case of communication problems or invalid configurations

Application Scenarios

- Control of Wago 750-1506 digital I/O modules
- Industrial automation applications
- Machine control with digital signals
- Process automation with distributed I/Os

⚖️ Comparison with Similar Function Blocks

Compared to generic I/O function blocks, the Wago1506 offers specific optimizations for Wago hardware, specifically for the 750-1506 module. The integration of dedicated bus adapters ensures more reliable communication than universal solutions.

Conclusion

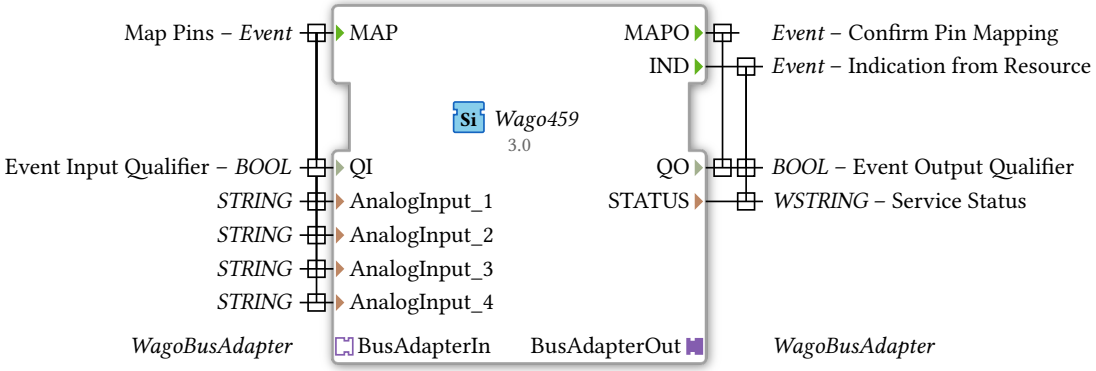
The Wago1506 function block represents a robust and specialized solution for controlling Wago 750-1506 I/O modules. Its clear separation of configuration (MAP) and operation (IND), along with comprehensive status feedback, makes it ideally suited for industrial automation projects with high reliability requirements.

Wago459

• – ◦ • – ◦ • – ◦ •

Introduction

The Wago459 is a Service Interface Function Block for connecting Wago-750-459 analog input modules in 4diac systems. This function block enables the configuration and monitoring of up to four analog input channels and establishes communication with the Wago bus system.



Interface Structure

Event Inputs

- **MAP**: Initiates the pin assignment for the analog inputs. Linked to the data variables AnalogInput_1 to AnalogInput_4 and QI.

Event Outputs

- **MAPO**: Confirms successful pin assignment. Linked to QO.
- **IND**: Provides status information from the resource manager. Linked to QO and STATUS.

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Controls the activation of the function block
- **AnalogInput_1** (STRING): Configuration for analog input 1
- **AnalogInput_2** (STRING): Configuration for analog input 2
- **AnalogInput_3** (STRING): Configuration for analog input 3

- **AnalogInput_4** (STRING): Configuration for analog input 4

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Indicates the operating status
- **STATUS** (WSTRING): Detailed status information of the service

Adapters

- **BusAdapterOut** (Plug): Outgoing bus adapter connection of type WagoBusAdapter
- **BusAdapterIn** (Socket): Incoming bus adapter connection of type WagoBusAdapter

Functionality

The Wago459 function block serves as Interface between the 4diac controller and the Wago 750-459 analog input module. Upon receiving the MAP event, the configured pin assignments (AnalogInput_1 to AnalogInput_4) are transmitted to the bus system. The block confirms successful assignments via MAPO and continuously provides status information via IND events.

Technical Features

- Supports up to four analog input channels
- Uses STRING data types for flexible pin configuration
- Provides comprehensive status feedback via WSTRING
- Implements a bidirectional adapter system for bus communication
- Includes qualifier variables (QI/QO) for reliable state management

State Overview

The function block has the following operating states:

- Inactive: Waits for a MAP event with QI enabled
- Configuration: Processes pin assignments after a MAP event
- Operational: Continuously monitors the module status and sends IND events
- Error: In case of problems, STATUS is populated with error information

Application Scenarios

- Connecting Wago 750-459 analog input modules in automation systems
- Acquiring data from analog sensors (0-10V, 0-20mA, 4-20mA)
- Process monitoring in industrial control systems
- Integration into distributed systems Peripheral Systems with Wago Bus Architecture

⚖️ Comparison with Similar Modules

Compared to generic I/O function blocks, Wago459 offers:

- Specific optimization for Wago-750-459 hardware
- Pre-configured adapters for Wago bus systems
- Direct support for all four channels of the module
- Integrated status monitoring and error handling

Conclusion

The Wago459 function block represents a reliable and specialized solution for integrating Wago-750-459 analog input modules into 4diac-based control systems. Its clear interface definition and comprehensive status feedback enable efficient implementation of analog measurement processing in industrial automation projects.

🌐 Related topic subpages on ms-muc-docs.de

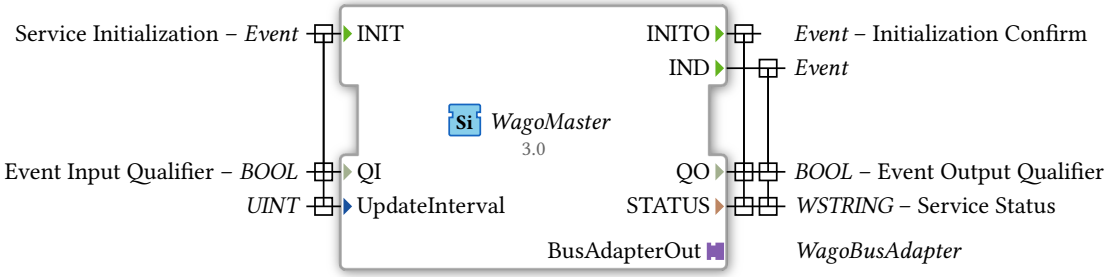
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

WagoMaster

- – ◦ • – ◦ • – ◦ •

Introduction

The WagoMaster is a Service Interface Function Block for controlling and monitoring Wago bus systems. This block serves as the master component for communication with Wago I/O systems and enables the initialization and management of the bus connection.



Interface Structure

Event Inputs

- **INIT**: Service Initialization
- Linked to: QI, UpdateInterval

Event Outputs

- **INITO**: Initialization Confirmation
- Linked to: QO, STATUS
- **IND**: Indication Event
- Linked to: STATUS, QO

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Controls service activation
- **UpdateInterval** (UINT): Defines the update interval for bus communication

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Displays the service execution status
- **STATUS** (WSTRING): Service Status - Contains status information and error messages

Adapters

- **BusAdapterOut**: WagoBusAdapter - Establishes the connection to the Wago bus system

Functionality

The WagoMaster initializes and manages communication with Wago bus systems. Upon activation via the INIT event, the service is started, and bus communication is performed periodically according to the defined UpdateInterval. The function block reports successful initialization via INITO and continuously provides status information via IND events.

Technical Features

- Supports WSTRING for detailed status messages
- Configurable update interval for flexible adaptation to various application requirements
- Integrated qualifier variables (QI/QO) for reliable service control
- Specialized WagoBusAdapter for optimized bus communication

State Overview

The function block goes through the following states:

1. **Inactive**: Before initialization
2. **Initialization**: Upon INIT event with QI=TRUE
3. **Active**: After successful initialization (QO=TRUE)
4. **Error**: In case of communication problems (STATUS with error message)

Application Scenarios

- Control of Wago 750 series I/O modules

- Integration into automation systems with Wago controllers
- Monitoring and diagnostics of Wago bus networks
- Industry 4.0 applications with distributed Wago I/O systems

⚖️ Comparison with Similar Function Blocks

Compared to generic The WagoMaster bus master module offers specific optimizations for Wago bus systems and integrated status diagnostics via WSTRING. Its specialized adapter interface enables more efficient communication than universal bus adapters.

Conclusion

The WagoMaster is a reliable and specialized module for integrating Wago bus systems into 4diac-based control systems. Its configurable parameters and detailed status feedback make it particularly suitable for demanding industrial automation applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

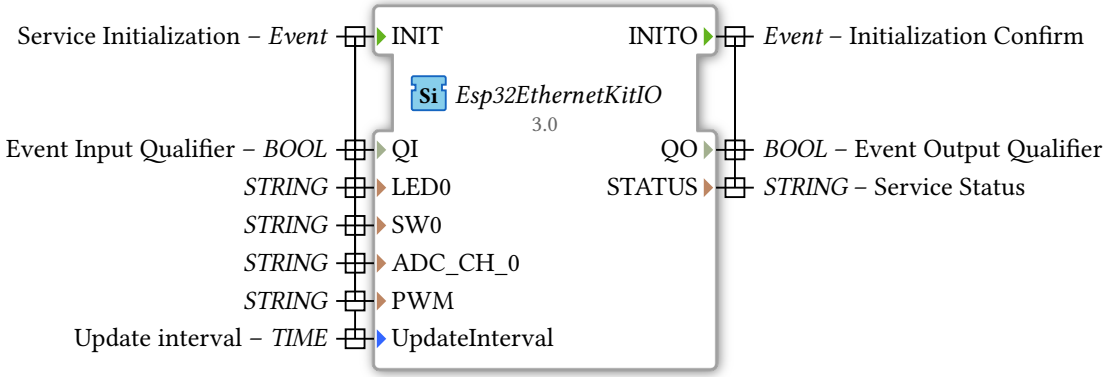
2.11.5.23 zephyr

Esp32EthernetKitIO

• – ◦ • – ◦ • – ◦ •

Introduction

The Esp32EthernetKitIO function block serves as a template for modular input/output operations with the Esp32EthernetKit board. This block enables the control and monitoring of various hardware components of the board via a standardized interface.



Interface Structure

Event Inputs

- **INIT**: Service Initialization - triggers the initialization process of the function block

Event Outputs

- **INITO**: Initialization Confirmation - signals successful completion of initialization

Data Inputs

- **QI** (BOOL): Event Input Qualifier - controls the execution of the function block
- **LED0** (STRING): Configuration for LED0
- **SW0** (STRING): Configuration for Switch SW0
- **ADC_CH_0** (STRING): Configuration for Analog-to-Digital Converter Channel 0
- **PWM** (STRING): Configuration for Pulse Width Modulation
- **UpdateInterval** (TIME): Update interval with a default value of 40 ms

Data Outputs

- **QO** (BOOL): Event Output Qualifier - status of the function block execution
- **STATUS** (STRING): Service status - provides detailed status information

Adapter

No adapter interfaces available.

Functionality

The Esp32EthernetKitIO chip initializes and manages the hardware components of the Esp32EthernetKit board. Upon receiving the INIT event, the configured input/output components (LEDs, switches, ADC, PWM) are set up according to the passed string parameters. The update interval determines how frequently the input/output states are updated.

Technical Features

- Uses string parameters for flexible configuration of hardware components
- Supports mixed signal types (digital, analog, PWM)
- Provides a configurable update interval for real-time applications
- Implemented in the Zephyr RTOS framework

State Transitions

1. **Initialization State**: Waits for an INIT event
2. **Activation State**: Processes INIT with configuration parameters
3. **Operating State**: Performs continuous I/O operations
4. **Status Message**: Sends INITO with status information

Application Scenarios

- Control of lighting systems with LEDs
- Capture of user input via pushbuttons/switches
- Measurement of analog sensor data via ADC
- Control of motors or actuators using PWM
- IoT devices with the ESP32 Ethernet Kit board

⚖️ Comparison with Similar Building Blocks

Compared to simple I/O blocks, it offers Esp32EthernetKitIO is an integrated solution for the specific hardware board, supporting various signal types and configurable update intervals.

Conclusion

The Esp32EthernetKitIO function block provides a flexible and powerful solution for controlling Esp32EthernetKit boards in 4diac-based control systems. Its modular configuration via string parameters enables versatile use in various industrial and IoT applications.

🌐 Related topic subpages on ms-muc-docs.de

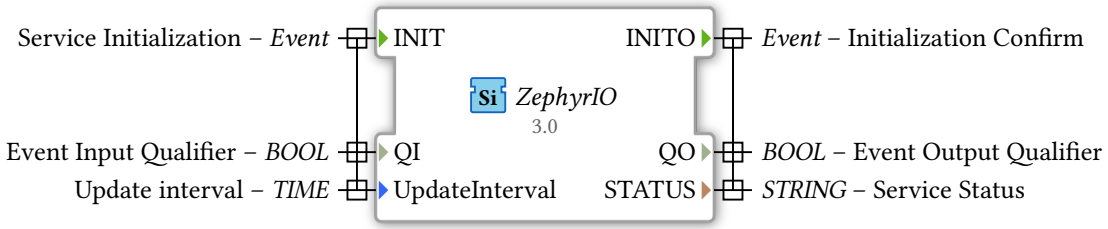
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de
- 🌐 The PWM Signal & Infographic on ms-muc-docs.de

ZephyrIO

• – ◦ • – ◦ • – ◦ •

Introduction

The ZephyrIO function block serves as a template for modular input/output functionality with boards that use the Zephyr operating system. This block enables the initialization and configuration of I/O services in distributed automation systems.



Interface Structure

Event Inputs

- **INIT**: Service Initialization Event
- Linked to: QI, UpdateInterval

Event Outputs

- **INITO**: Initialization Acknowledgement
- Linked to: QO, STATUS

Data Inputs

- **QI** (BOOL): Event Input Qualifier - Controls the execution of the initialization process
- **UpdateInterval** (TIME): Update interval for the service (default: T#40ms)

Data Outputs

- **QO** (BOOL): Event Output Qualifier - Displays the initialization status
- **STATUS** (STRING): Service status information

Adapters

No adapter interfaces available.

Functionality

The ZephyrIO function block initializes I/O services for Zephyr-based hardware. Upon receiving the INIT event, the service is started with the configured update interval. After successful initialization, the INITO event is output with corresponding status information.

Technical Features

- Specifically optimized for Zephyr RTOS
- Configurable update interval for periodic operations
- Standard update interval of 40 milliseconds
- Status feedback via STRING variable

Status Overview

1. **Not Initialized**: Waiting for INIT event
2. **Initialization in progress**: Processing INIT with QI=true
3. **Initialized**: Sending INITO with status information

Application Scenarios

- Integration of Zephyr-based I/O boards into 4diac systems
- Modular I/O extensions for embedded systems
- Hardware-level real-time control with Zephyr RTOS

⚖️ Comparison with similar components

Compared to generic I/O components, ZephyrIO offers specific optimizations for the Zephyr operating system and enables seamless integration of Zephyr-based hardware into 4diac environments.

Conclusion

The ZephyrIO function block represents a specialized solution for integrating Zephyr-based I/O components into IEC 61499 systems and offers a standardized interface for modular hardware extensions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

2.11.6 net

2.11.6.1 🎧 Podcast

- Industrial Networks: The Nervous System of the Modern Factory – OT, IT & the Future of Automation
- ISOBUS: The Secret Language of Agricultural Machinery – From Plug-and-Play to High-Speed Networking
- Protecting Solenoid Valves: Your Guide Against Voltage Spikes in Agricultural and Construction Machinery
- Open Source in Industry: More Than Code – A Network for Innovation and Collaboration
- Smart Power: How Decentralized Control Reduces Energy Costs in Industrial Plants and Stabilizes the Grid

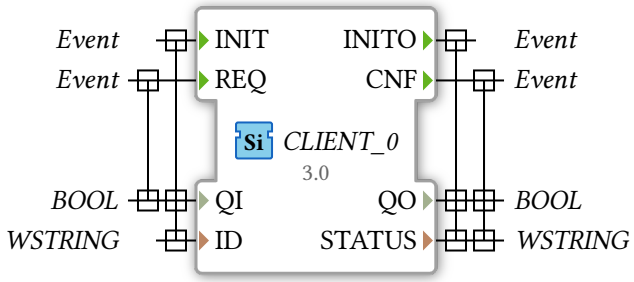
2.11.6.2 CLIENT_0

• – ◦ • – ◦ • – ◦ •

Introduction

The CLIENT_0 function block is used to communicate with a SERVER_0 block via a network connection. It enables the establishment and closure of connections. Unlike CLIENT_1, there is no explicit data exchange via data inputs/outputs; synchronization occurs primarily through events.

•



Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE)
- **REQ**: Requests interaction with the server

Event Outputs

- **INITO**: Confirms that a new connection has been established (QI = TRUE) or that a connection has been closed (QI = FALSE)
- **CNF**: Confirms that the request has been executed

Data Inputs

- **QI** (BOOL): Controls the connection status (TRUE = open connection, FALSE = close connection)
- **ID** (WSTRING): Identifier for the connection

Data Outputs

- **QO** (BOOL): Connection status (TRUE = connected, FALSE = disconnected)
- **STATUS** (WSTRING): Status information about The Connection

Adapter

No adapter interfaces are available.

Functionality

The CLIENT_0 block initializes a network connection to the SERVER_0 block via the INIT event. Upon successful connection, the INITO event is triggered with QO=TRUE. Interaction with the server can be initiated via the REQ event, the completion of which is confirmed by CNF.

Technical Features

- No data inputs/outputs (SD_x, RD_x) available
- WSTRING data type for ID and STATUS
- Generic implementation via GEN_CLIENT class name

State Overview

1. **Not connected**: Initial state, QO = FALSE
2. **Connection Established**: INIT processed with QI = TRUE
3. **Connected**: INITO confirmed with QO = TRUE
4. **Interaction**: REQ/CNF cycle
5. **Connection Disconnected**: INIT processed with QI = FALSE

Application Scenarios

- Pure event synchronization between systems
- "Heartbeat" monitoring without user data
- Trigger signals over network

Metadata

Attribute	Value
Copyright	(c) 2025 Monika Wenger
License	EPL-2.0
Version	3.0 (December 5, 2025, Monika Wenger)
4diac Package	iec61499::net

–

🌐 Related subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

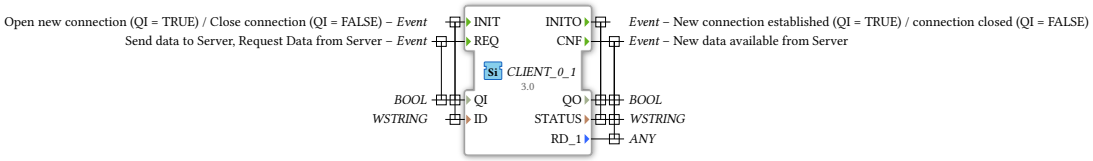
]

2.11.6.3 CLIENT_0_1

• – ◦ • – ◦ • – ◦ •

Introduction

The CLIENT_0_1 function block is used to communicate with a corresponding server block (e.g., SERVER_1_0) via a network connection. It is specifically designed to **receive** data from the server without sending any data itself (Receive Only Client).



Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE)
- **REQ**: Requests data from the server

Event Outputs

- **INITO**: Confirms that a new connection has been established (QI = TRUE) or that a connection has been closed (QI = FALSE)
- **CNF**: Signals that new data has been received from the server and is available at RD_1

Data Inputs

- **QI** (BOOL): Controls the connection status (TRUE = open connection, FALSE = close connection)
- **ID** (WSTRING): Identifier for the connection

Data Outputs

- **QO** (BOOL): Connection status (TRUE = connected, FALSE = disconnected)
- **STATUS** (WSTRING): Connection status information
- **RD_1** (ANY): Data received from the server

Adapter

No adapter interfaces are available.

Functionality

The CLIENT_0_1 block establishes a connection via INIT. It requests data from the server using the REQ event. When the server responds, the data is provided at output RD_1, and the CNF event is triggered. There is no data input for sending user data.

Technical Features

- Unidirectional data transmission (receive)
- Use of the ANY data type for RD_1 enables flexible data types
- WSTRING data type for ID and STATUS

Application Scenarios

- Querying sensor values or status information from a remote system
- Monitoring systems that only read data

Metadata

Attribute	Value
Copyright	© 2017, 2025 fortiss GmbH, JKU Linz
License	EPL-2.0
Version	3.0 (2025-19-26, Alois Zoitl)
4diac Package	iec61499::net

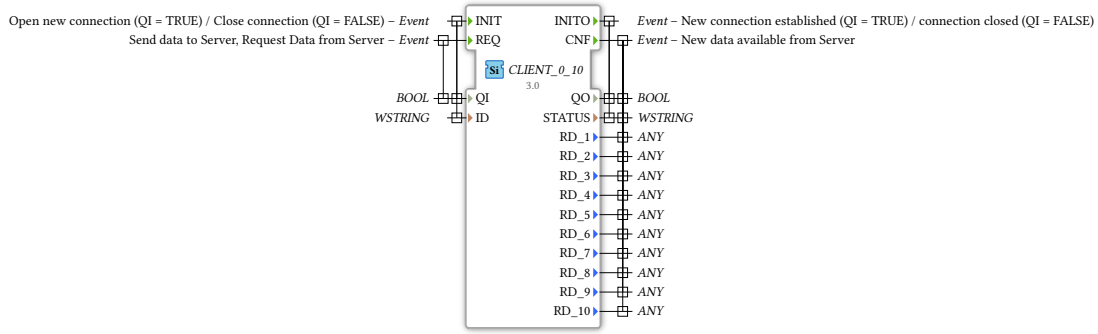
_

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.6.4 CLIENT_0_10



• – ◦ • – ◦ • – ◦ •

Introduction

The `CLIENT_0_10` function block is the generic client variant with 0 send and 10 receive data fields for communicating with a matching SERVER_10_0 block. It receives 10 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`) from the server without sending any payload itself. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`CLIENT_0_10` transmits no payload data with `REQ`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.
- **RD_4** (ANY): Receive value 4, delivered by the server with `CNF`.
- **RD_5** (ANY): Receive value 5, delivered by the server with `CNF`.
- **RD_6** (ANY): Receive value 6, delivered by the server with `CNF`.
- **RD_7** (ANY): Receive value 7, delivered by the server with `CNF`.
- **RD_8** (ANY): Receive value 8, delivered by the server with `CNF`.
- **RD_9** (ANY): Receive value 9, delivered by the server with `CNF`.
- **RD_10** (ANY): Receive value 10, delivered by the server with `CNF`.

Functionality

`CLIENT_0_10` initializes a connection to the matching `SERVER_10_0` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, a request without payload is sent. Once response data (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`) is available, the block triggers `CNF` and provides `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_10_0`**: `CLIENT_0_10` is only functionally compatible with a `SERVER_10_0` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 10 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_10_0` with the matching field count.

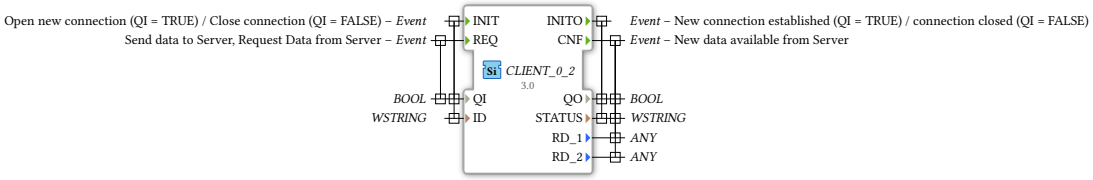
Comparison with similar function blocks

- **SERVER_10_0**: the direct counterpart on the server side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_0_10` provides the generic client variant of the `GEN_CLIENT` family tailored to 0 send and 10 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.5 CLIENT_0_2



• – ○ • – ○ • – ○ •

Introduction

The `CLIENT_0_2` function block is the generic client variant with 0 send and 2 receive data fields for communicating with a matching SERVER_2_0 block. It receives 2 data values (`RD_1` `RD_2`) from the server without sending any payload itself. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1` `RD_2`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`CLIENT_0_2` transmits no payload data with `REQ`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.

Functionality

`CLIENT_0_2` initializes a connection to the matching `SERVER_2_0` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, a request without payload is sent. Once response data (`RD_1` `RD_2`) is available, the block triggers `CNF` and provides `RD_1` `RD_2`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_2_0`**: `CLIENT_0_2` is only functionally compatible with a `SERVER_2_0` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 2 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_2_0` with the matching field count.

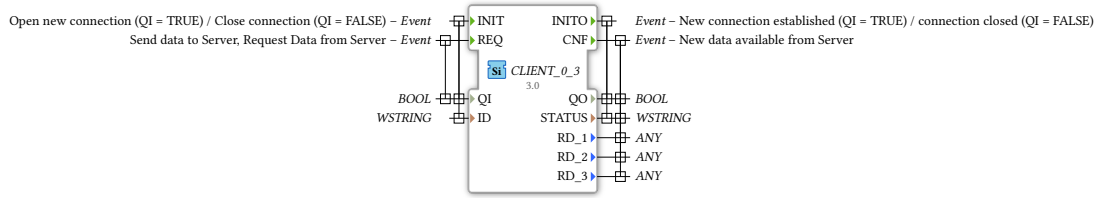
Comparison with similar function blocks

- SERVER_2_0: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_0_2` provides the generic client variant of the `GEN_CLIENT` family tailored to 0 send and 2 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.6 CLIENT_0_3



• – ○ • – ○ • – ○ •

Introduction

The CLIENT_0_3 function block is the generic client variant with 0 send and 3 receive data fields for communicating with a matching SERVER_3_0 block. It receives 3 data values (RD_1 RD_2 RD_3) from the server without sending any payload itself. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.
- **REQ**: Sends data to the server and requests data from the server, carries QI.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **CNF**: Signals that new data is available from the server, carries QO, STATUS and RD_1 RD_2 RD_3.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (CLIENT_0_3 transmits no payload data with REQ, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with CNF.
- **RD_2** (ANY): Receive value 2, delivered by the server with CNF.
- **RD_3** (ANY): Receive value 3, delivered by the server with CNF.

Functionality

CLIENT_0_3 initializes a connection to the matching SERVER_3_0 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, a request without payload is sent. Once response data (RD_1 RD_2 RD_3) is available, the block triggers CNF and provides RD_1 RD_2 RD_3.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_3_0**: CLIENT_0_3 is only functionally compatible with a SERVER_3_0 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 3 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_3_0 with the matching field count.

Comparison with similar function blocks

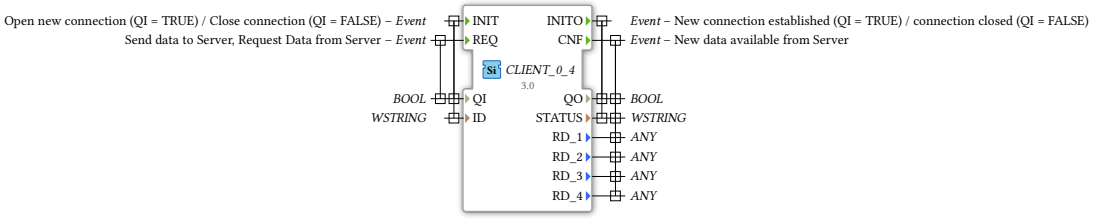
- **SERVER_3_0**: the direct counterpart on the server side, with send/receive field counts swapped.

- **CLIENT_1 / SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_0_3` provides the generic client variant of the `GEN_CLIENT` family tailored to 0 send and 3 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.7 CLIENT_0_4



• – ○ • – ○ • – ○ •

Introduction

The `CLIENT_0_4` function block is the generic client variant with 0 send and 4 receive data fields for communicating with a matching SERVER_4_0 block. It receives 4 data values (`RD_1 RD_2 RD_3 RD_4`) from the server without sending any payload itself. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1 RD_2 RD_3 RD_4`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`CLIENT_0_4` transmits no payload data with `REQ`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.
- **RD_4** (ANY): Receive value 4, delivered by the server with `CNF`.

Functionality

`CLIENT_0_4` initializes a connection to the matching `SERVER_4_0` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, a request without payload is sent. Once response data (`RD_1 RD_2 RD_3 RD_4`) is available, the block triggers `CNF` and provides `RD_1 RD_2 RD_3 RD_4`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.

- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_4_0**: CLIENT_0_4 is only functionally compatible with a SERVER_4_0 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 4 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_4_0 with the matching field count.

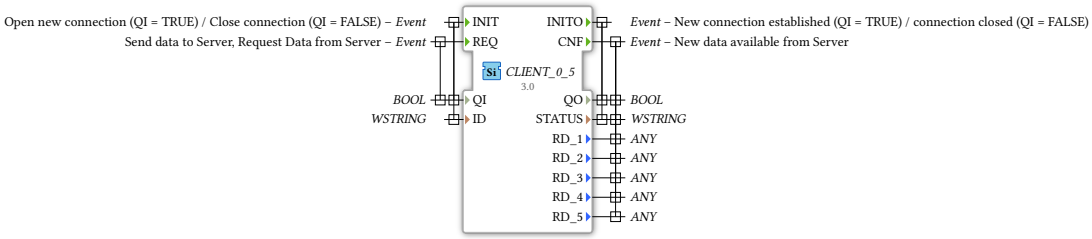
Comparison with similar function blocks

- SERVER_4_0: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

CLIENT_0_4 provides the generic client variant of the GEN_CLIENT family tailored to 0 send and 4 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.8 CLIENT_0_5



• — ○ • — ○ • — ○ •

Introduction

The CLIENT_0_5 function block is the generic client variant with 0 send and 5 receive data fields for communicating with a matching SERVER_5_0 block. It receives 5 data values (RD_1 RD_2 RD_3 RD_4 RD_5) from the server without sending any payload itself. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.
- **REQ**: Sends data to the server and requests data from the server, carries QI.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **CNF**: Signals that new data is available from the server, carries QO, STATUS and RD_1 RD_2 RD_3 RD_4 RD_5.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).

- No send fields (`CLIENT_0_5` transmits no payload data with `REQ`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.
- **RD_4** (ANY): Receive value 4, delivered by the server with `CNF`.
- **RD_5** (ANY): Receive value 5, delivered by the server with `CNF`.

Functionality

`CLIENT_0_5` initializes a connection to the matching `SERVER_5_0` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, a request without payload is sent. Once response data (`RD_1 RD_2 RD_3 RD_4 RD_5`) is available, the block triggers `CNF` and provides `RD_1 RD_2 RD_3 RD_4 RD_5`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_5_0`**: `CLIENT_0_5` is only functionally compatible with a `SERVER_5_0` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 5 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_5_0` with the matching field count.

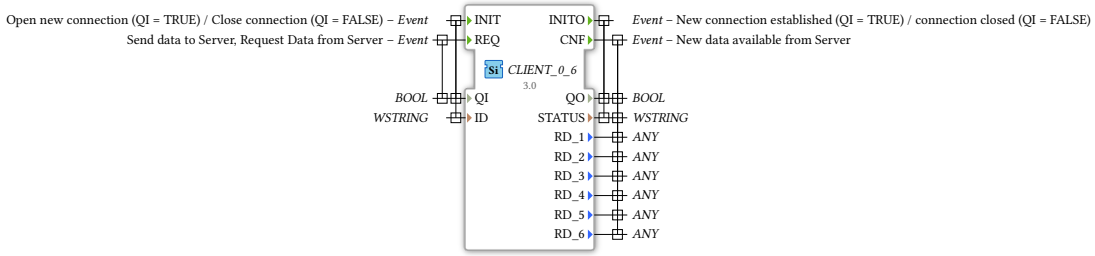
Comparison with similar function blocks

- SERVER_5_0: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_0_5` provides the generic client variant of the `GEN_CLIENT` family tailored to 0 send and 5 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.9 CLIENT_0_6



• – ○ • – ○ • – ○ •

Introduction

The CLIENT_0_6 function block is the generic client variant with 0 send and 6 receive data fields for communicating with a matching SERVER_6_0 block. It receives 6 data values (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6) from the server without sending any payload itself. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.
- **REQ**: Sends data to the server and requests data from the server, carries QI.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **CNF**: Signals that new data is available from the server, carries QO, STATUS and RD_1 RD_2 RD_3 RD_4 RD_5 RD_6.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (CLIENT_0_6 transmits no payload data with REQ, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with CNF.
- **RD_2** (ANY): Receive value 2, delivered by the server with CNF.
- **RD_3** (ANY): Receive value 3, delivered by the server with CNF.
- **RD_4** (ANY): Receive value 4, delivered by the server with CNF.
- **RD_5** (ANY): Receive value 5, delivered by the server with CNF.
- **RD_6** (ANY): Receive value 6, delivered by the server with CNF.

Functionality

CLIENT_0_6 initializes a connection to the matching SERVER_6_0 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, a request without payload is sent. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6) is available, the block triggers CNF and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_6_0**: CLIENT_0_6 is only functionally compatible with a SERVER_6_0 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 6 values, without carrying unused surplus data fields.

- **Client-side connection** to a communication partner already laid out as a `SERVER_6_0` with the matching field count.

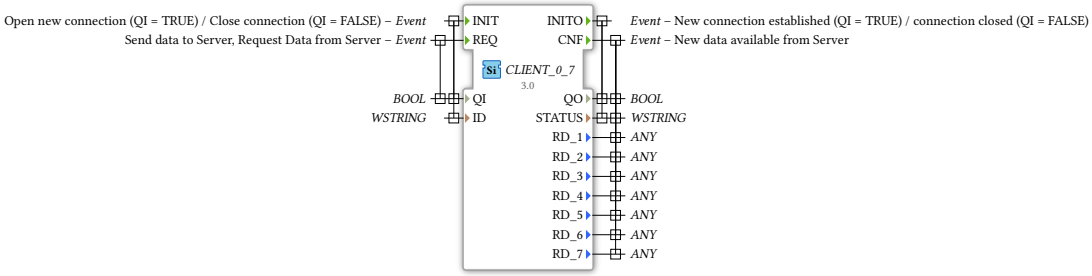
Comparison with similar function blocks

- **SERVER_6_0**: the direct counterpart on the server side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_0_6` provides the generic client variant of the `GEN_CLIENT` family tailored to 0 send and 6 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.10 CLIENT_0_7



• – ◦ • – ◦ • – ◦ •

Introduction

The `CLIENT_0_7` function block is the generic client variant with 0 send and 7 receive data fields for communicating with a matching SERVER_7_0 block. It receives 7 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`) from the server without sending any payload itself. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`CLIENT_0_7` transmits no payload data with `REQ`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.
- **RD_4** (ANY): Receive value 4, delivered by the server with `CNF`.
- **RD_5** (ANY): Receive value 5, delivered by the server with `CNF`.
- **RD_6** (ANY): Receive value 6, delivered by the server with `CNF`.
- **RD_7** (ANY): Receive value 7, delivered by the server with `CNF`.

Functionality

CLIENT_0_7 initializes a connection to the matching SERVER_7_0 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, a request without payload is sent. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7) is available, the block triggers CNF and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_7_0**: CLIENT_0_7 is only functionally compatible with a SERVER_7_0 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 7 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_7_0 with the matching field count.

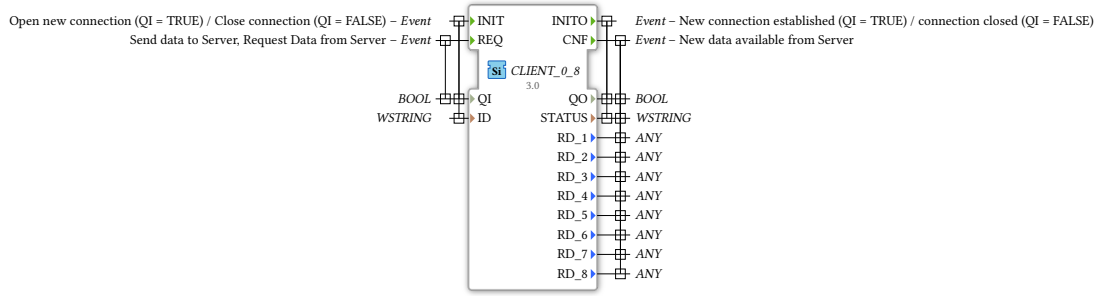
Comparison with similar function blocks

- SERVER_7_0: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

CLIENT_0_7 provides the generic client variant of the GEN_CLIENT family tailored to 0 send and 7 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.11 CLIENT_0_8



• – ○ • – ○ • – ○ •

Introduction

The CLIENT_0_8 function block is the generic client variant with 0 send and 8 receive data fields for communicating with a matching SERVER_8_0 block. It receives 8 data values (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8) from the server without sending any payload itself. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`CLIENT_0_8` transmits no payload data with `REQ`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.
- **RD_4** (ANY): Receive value 4, delivered by the server with `CNF`.
- **RD_5** (ANY): Receive value 5, delivered by the server with `CNF`.
- **RD_6** (ANY): Receive value 6, delivered by the server with `CNF`.
- **RD_7** (ANY): Receive value 7, delivered by the server with `CNF`.
- **RD_8** (ANY): Receive value 8, delivered by the server with `CNF`.

Functionality

`CLIENT_0_8` initializes a connection to the matching `SERVER_8_0` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, a request without payload is sent. Once response data (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`) is available, the block triggers `CNF` and provides `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_8_0`**: `CLIENT_0_8` is only functionally compatible with a `SERVER_8_0` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 8 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_8_0` with the matching field count.

Comparison with similar function blocks

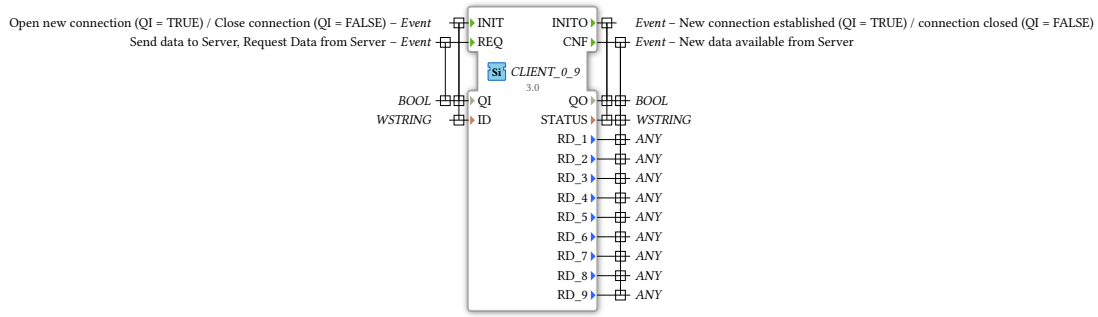
- **SERVER_8_0**: the direct counterpart on the server side, with send/receive field counts swapped.

- **CLIENT_1 / SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_0_8` provides the generic client variant of the `GEN_CLIENT` family tailored to 0 send and 8 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.12 CLIENT_0_9



• – ◦ • – ◦ • – ◦ •

Introduction

The `CLIENT_0_9` function block is the generic client variant with 0 send and 9 receive data fields for communicating with a matching SERVER_9_0 block. It receives 9 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9`) from the server without sending any payload itself. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`CLIENT_0_9` transmits no payload data with `REQ`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.
- **RD_4** (ANY): Receive value 4, delivered by the server with `CNF`.
- **RD_5** (ANY): Receive value 5, delivered by the server with `CNF`.
- **RD_6** (ANY): Receive value 6, delivered by the server with `CNF`.
- **RD_7** (ANY): Receive value 7, delivered by the server with `CNF`.
- **RD_8** (ANY): Receive value 8, delivered by the server with `CNF`.
- **RD_9** (ANY): Receive value 9, delivered by the server with `CNF`.

Functionality

CLIENT_0_9 initializes a connection to the matching SERVER_9_0 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, a request without payload is sent. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9) is available, the block triggers CNF and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_9_0**: CLIENT_0_9 is only functionally compatible with a SERVER_9_0 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 9 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_9_0 with the matching field count.

Comparison with similar function blocks

- SERVER_9_0: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

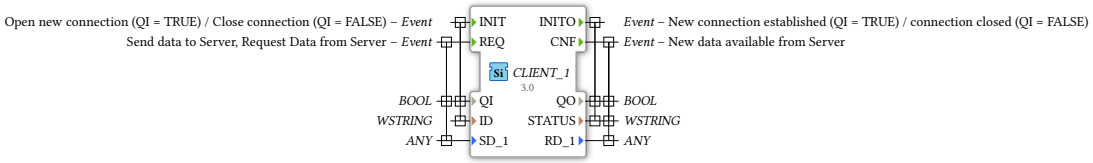
CLIENT_0_9 provides the generic client variant of the GEN_CLIENT family tailored to 0 send and 9 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.13 CLIENT_1

• – ◦ • – ◦ • – ◦ •

Introduction

The CLIENT_1 function block is used to communicate with a SERVER_1 block via a network connection. It enables the establishment and closure of connections as well as data exchange between client and server.



Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE)
- **REQ**: Sends data to the server and requests data from the server

Event Outputs

- **INITO**: Confirms the establishment of a new connection (QI = TRUE) or the closing of a connection (QI = FALSE)
- **CNF**: Signals that new data is available from the server

Data Inputs

- **QI** (BOOL): Controls the connection status (TRUE = open connection, FALSE = close connection)
- **ID** (WSTRING): Identifier for the connection
- **SD_1** (ANY): Data to be sent to the server

Data Outputs

- **QO** (BOOL): Connection status (TRUE = connected, FALSE = disconnected)
- **STATUS** (WSTRING): Connection status information
- **RD_1** (ANY): Data received from the server

Adapter

No adapter interfaces are present.

Functionality

The CLIENT_1 block initializes a network connection to the SERVER_1 block via the INIT event. Upon successful connection, the INITO event is triggered with QO=TRUE. Data can be sent to and requested from the server via the REQ event. Received data is made available via the CNF event using the corresponding data outputs.

Technical Features

- Use of the ANY data type for SD_1 and RD_1 enables flexible data types
- WSTRING data type for ID and STATUS supports Unicode characters
- Generic implementation through GEN_CLIENT class names

State Overview

1. **Not connected**: Initial state, QO = FALSE
2. **Connection establishing**: INIT processed with QI = TRUE
3. **Connected**: INITO confirmed with QO = TRUE
4. **Data transmission**: REQ/CNF cycle for data exchange
5. **Connection disconnection**: INIT processed with QI = FALSE

Application Scenarios

- Communication between distributed control systems
- Data exchange in Industry 4.0 applications
- Client-server architectures in automation systems
- Networked control components via TCP/IP or other protocols

⚖️ Comparison with similar modules

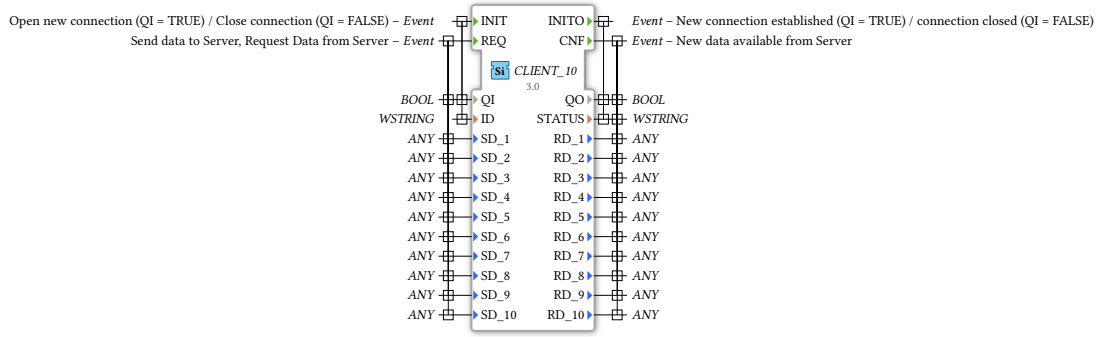
Compared to simpler communication modules, this offers CLIENT_1:

- Bidirectional data exchange in a single transaction
- Flexible data types through ANY type
- Explicit connection control
- Detailed status feedback

Conclusion

The CLIENT_1 function block provides a robust solution for client-server communication in distributed automation systems. Its flexible interface and explicit connection control make it particularly suitable for demanding network applications in industrial environments.

2.11.6.14 CLIENT_10



• – ◦ • – ◦ • – ◦ •

Introduction

The `CLIENT_10` function block is the generic client variant with 10 send and 10 receive data fields for communicating with a matching SERVER_10 block. It transmits 10 data values (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`) to the server and receives 10 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`) back. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.
- **SD_2** (ANY): Send value 2, transmitted to the server with `REQ`.
- **SD_3** (ANY): Send value 3, transmitted to the server with `REQ`.
- **SD_4** (ANY): Send value 4, transmitted to the server with `REQ`.
- **SD_5** (ANY): Send value 5, transmitted to the server with `REQ`.
- **SD_6** (ANY): Send value 6, transmitted to the server with `REQ`.
- **SD_7** (ANY): Send value 7, transmitted to the server with `REQ`.
- **SD_8** (ANY): Send value 8, transmitted to the server with `REQ`.
- **SD_9** (ANY): Send value 9, transmitted to the server with `REQ`.
- **SD_10** (ANY): Send value 10, transmitted to the server with `REQ`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.
- **RD_4** (ANY): Receive value 4, delivered by the server with `CNF`.
- **RD_5** (ANY): Receive value 5, delivered by the server with `CNF`.
- **RD_6** (ANY): Receive value 6, delivered by the server with `CNF`.

- **RD_7** (ANY): Receive value 7, delivered by the server with CNF.
- **RD_8** (ANY): Receive value 8, delivered by the server with CNF.
- **RD_9** (ANY): Receive value 9, delivered by the server with CNF.
- **RD_10** (ANY): Receive value 10, delivered by the server with CNF.

Functionality

CLIENT_10 initializes a connection to the matching SERVER_10 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10 are transmitted to the server. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10) is available, the block triggers CNF and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_10**: CLIENT_10 is only functionally compatible with a SERVER_10 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 10 values and receive exactly 10 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_10 with the matching field count.

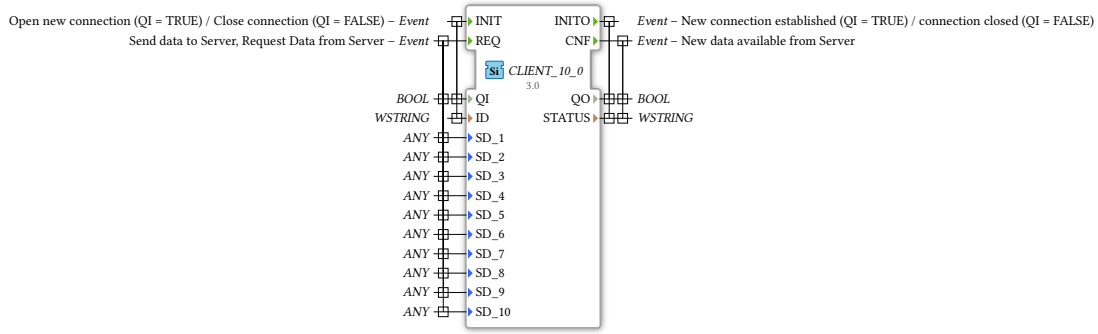
Comparison with similar function blocks

- **SERVER_10**: the direct counterpart on the server side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

CLIENT_10 provides the generic client variant of the GEN_CLIENT family tailored to 10 send and 10 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.15 CLIENT_10_0



• – ○ • – ○ • – ○ •

Introduction

The CLIENT_10_0 function block is the generic client variant with 10 send and 0 receive data fields for communicating with a matching SERVER_0_10 block. It transmits 10 data values (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10) to the server without receiving any data back. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.
- **REQ**: Sends data to the server and requests data from the server, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **CNF**: Signals that new data is available from the server, carries QO, STATUS.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with REQ.
- **SD_2** (ANY): Send value 2, transmitted to the server with REQ.
- **SD_3** (ANY): Send value 3, transmitted to the server with REQ.
- **SD_4** (ANY): Send value 4, transmitted to the server with REQ.
- **SD_5** (ANY): Send value 5, transmitted to the server with REQ.
- **SD_6** (ANY): Send value 6, transmitted to the server with REQ.
- **SD_7** (ANY): Send value 7, transmitted to the server with REQ.
- **SD_8** (ANY): Send value 8, transmitted to the server with REQ.
- **SD_9** (ANY): Send value 9, transmitted to the server with REQ.
- **SD_10** (ANY): Send value 10, transmitted to the server with REQ.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (CLIENT_10_0 receives no payload data via CNF, only connection/status information).

Functionality

CLIENT_10_0 initializes a connection to the matching SERVER_0_10 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10 are transmitted to the server. Once a response is available, the block triggers CNF.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_0_10**: CLIENT_10_0 is only functionally compatible with a SERVER_0_10 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.

5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 10 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_0_10` with the matching field count.

Comparison with similar function blocks

- **SERVER_0_10**: the direct counterpart on the server side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

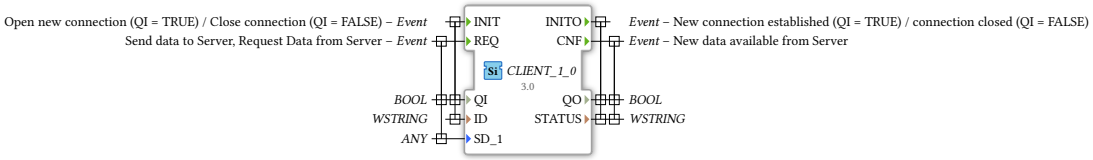
`CLIENT_10_0` provides the generic client variant of the `GEN_CLIENT` family tailored to 10 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.16 CLIENT_1_0

• – ◦ • – ◦ • – ◦ •

Introduction

The CLIENT_1_0 function block is used to communicate with a corresponding server block (e.g., SERVER_0_1) via a network connection. It is specifically designed for **sending** data to the server without receiving any data (Send Only Client).



Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE)
- **REQ**: Sends the data present at SD_1 to the server

Event Outputs

- **INITO**: Confirms that a new connection has been established (QI = TRUE) or that a connection has been closed (QI = FALSE)
- **CNF**: Confirms that the data was sent successfully

Data Inputs

- **QI** (BOOL): Controls the connection status (TRUE = open connection, FALSE = close connection)
- **ID** (WSTRING): Identifier for the connection
- **SD_1** (ANY): Data to be sent to the server

Data Outputs

- **QO** (BOOL): Connection status (TRUE = connected, FALSE = disconnected)
- **STATUS** (WSTRING): Status information about the connection

Adapter

No adapter interfaces are available.

Functionality

The CLIENT_1_0 block establishes a connection via INIT. When the REQ event is triggered, the block sends the value of SD_1 to the server. Successful transmission is confirmed by the CNF event. There is no data output for received data.

Technical Features

- Unidirectional data transmission (sending)
- Use of the ANY data type for SD_1 enables flexible data types
- WSTRING data type for ID and STATUS

Application Scenarios

- Sending control commands or setpoints to a remote system (without a direct response as data)
- Logging clients that push data to a central server

Metadata

Attribute	Value
Copyright	(c) 2017, 2025 fortiss GmbH, JKU Linz
License	EPL-2.0
Version	3.0 (2025-19-26, Alois Zoitl)
4diac Package	iec61499::net

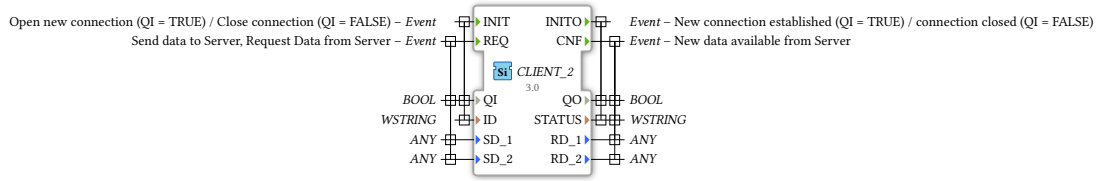
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.6.17 CLIENT_2



• – ○ • – ○ • – ○ •

Introduction

The `CLIENT_2` function block is the generic client variant with 2 send and 2 receive data fields for communicating with a matching SERVER_2 block. It transmits 2 data values (`SD_1` `SD_2`) to the server and receives 2 data values (`RD_1` `RD_2`) back. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1` `SD_2`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1` `RD_2`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.
- **SD_2** (ANY): Send value 2, transmitted to the server with `REQ`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.

Functionality

`CLIENT_2` initializes a connection to the matching `SERVER_2` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, the values `SD_1` `SD_2` are transmitted to the server. Once response data (`RD_1` `RD_2`) is available, the block triggers `CNF` and provides `RD_1` `RD_2`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_2`**: `CLIENT_2` is only functionally compatible with a `SERVER_2` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 2 values and receive exactly 2 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_2` with the matching field count.

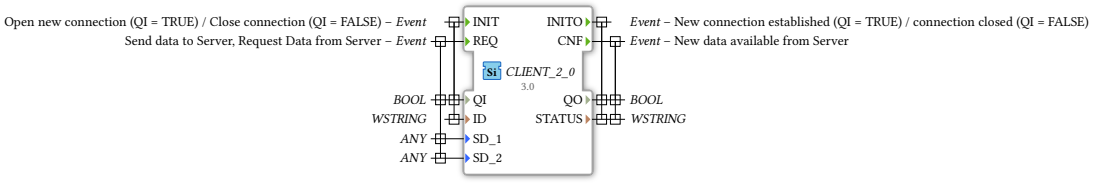
Comparison with similar function blocks

- SERVER_2: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_2` provides the generic client variant of the `GEN_CLIENT` family tailored to 2 send and 2 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.18 CLIENT_2_0



• – ○ • – ○ • – ○ •

Introduction

The `CLIENT_2_0` function block is the generic client variant with 2 send and 0 receive data fields for communicating with a matching SERVER_0_2 block. It transmits 2 data values (`SD_1` `SD_2`) to the server without receiving any data back. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1` `SD_2`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.
- **SD_2** (ANY): Send value 2, transmitted to the server with `REQ`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`CLIENT_2_0` receives no payload data via `CNF`, only connection/status information).

Functionality

`CLIENT_2_0` initializes a connection to the matching `SERVER_0_2` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, the values `SD_1` `SD_2` are transmitted to the server. Once a response is available, the block triggers `CNF`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_0_2`**: `CLIENT_2_0` is only functionally compatible with a `SERVER_0_2` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 2 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_0_2` with the matching field count.

Comparison with similar function blocks

- SERVER_0_2: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

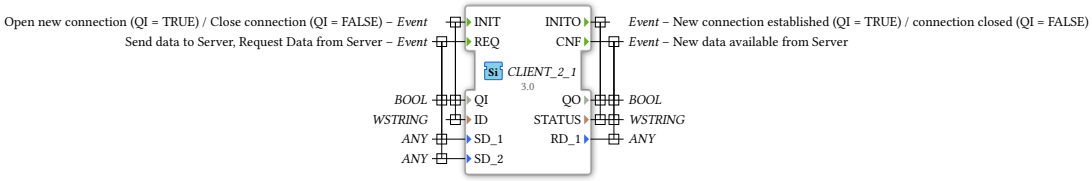
`CLIENT_2_0` provides the generic client variant of the `GEN_CLIENT` family tailored to 2 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.19 CLIENT_2_1

• – ◦ • – ◦ • – ◦ •

Introduction

The CLIENT_2_1 function block is used to communicate with a SERVER_1_2 block via a network connection. It enables the establishment and termination of connections as well as data exchange between client and server.



Interface Structure

Event Inputs

- **INIT**: Initialization event to open (QI = TRUE) or close (QI = FALSE) a connection
- **REQ**: Request event to send data to the server and request data from the server

Event Outputs

- **INITO**: Acknowledge event for connection establishment (QI = TRUE) or connection termination (QI = FALSE)
- **CNF**: Acknowledge event when new data is available from the server

Data Inputs

- **QI** (BOOL): Quality indicator for the INIT event (TRUE = open connection, FALSE = close connection)
- **ID** (WSTRING): Identifier for the connection
- **SD_1** (ANY): First data transmission block to the server
- **SD_2** (ANY): Second data transmission block to the Server

Data Outputs

- **QO** (BOOL): Quality indicator for output events
- **STATUS** (WSTRING): Status information about the connection or operation
- **RD_1** (ANY): Data block received from the server

Adapter

No adapter interfaces available.

Functionality

The CLIENT_2_1 block manages the entire client side of client-server communication. With INIT and QI=TRUE, a new connection to the specified server is established. With INIT and QI=FALSE, an existing connection is closed. The REQ event simultaneously triggers the sending of data SD_1 and SD_2 to the server and the request for data from the server. Upon successful operation, the received data is output via RD_1.

Technical Features

- Supports the ANY data type for flexible data transfer
- Uses WSTRING for status messages and connection IDs
- Provides bidirectional data exchange in a single operation
- Implements complete connection management (establishment/disconnection)

State Overview

The block can assume the following states:

- Not connected (initial state)
- Establishing connection
- Connected and ready for operation
- Disconnecting connection in progress

Application Scenarios

- Industrial automation systems with distributed control

- IoT applications with client-server architecture
- Data exchange between different control levels
- Remote monitoring and control of devices

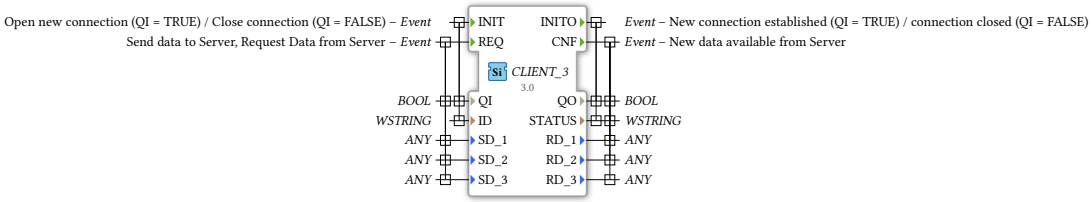
⚖️ Comparison with Similar Blocks

Compared to simpler client blocks, CLIENT_2_1 offers the ability to send two different data blocks simultaneously and receive one data block at a time. The integration of connection management and data exchange into a single block simplifies the application logic.

Conclusion

The CLIENT_2_1 function block represents a powerful solution for client-server communication in distributed automation systems. By combining connection management and bidirectional data exchange in a single block, it enables efficient and transparent network applications.

2.11.6.20 CLIENT_3



• – ○ • – ○ • – ○ •

Introduction

The `CLIENT_3` function block is the generic client variant with 3 send and 3 receive data fields for communicating with a matching SERVER_3 block. It transmits 3 data values (`SD_1` `SD_2` `SD_3`) to the server and receives 3 data values (`RD_1` `RD_2` `RD_3`) back. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1` `SD_2` `SD_3`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.
- **SD_2** (ANY): Send value 2, transmitted to the server with `REQ`.
- **SD_3** (ANY): Send value 3, transmitted to the server with `REQ`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.

Functionality

CLIENT_3 initializes a connection to the matching SERVER_3 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, the values SD_1 SD_2 SD_3 are transmitted to the server. Once response data (RD_1 RD_2 RD_3) is available, the block triggers CNF and provides RD_1 RD_2 RD_3.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_3**: CLIENT_3 is only functionally compatible with a SERVER_3 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 3 values and receive exactly 3 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_3 with the matching field count.

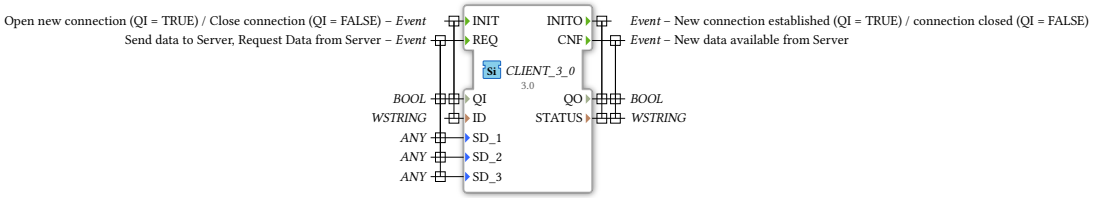
Comparison with similar function blocks

- SERVER_3: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

CLIENT_3 provides the generic client variant of the GEN_CLIENT family tailored to 3 send and 3 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.21 CLIENT_3_0



• – ○ • – ○ • – ○ •

Introduction

The CLIENT_3_0 function block is the generic client variant with 3 send and 0 receive data fields for communicating with a matching SERVER_0_3 block. It transmits 3 data values (SD_1 SD_2 SD_3) to the server without receiving any data back. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.

- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1` `SD_2` `SD_3`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.
- **SD_2** (ANY): Send value 2, transmitted to the server with `REQ`.
- **SD_3** (ANY): Send value 3, transmitted to the server with `REQ`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`CLIENT_3_0` receives no payload data via `CNF`, only connection/status information).

Functionality

`CLIENT_3_0` initializes a connection to the matching `SERVER_0_3` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, the values `SD_1` `SD_2` `SD_3` are transmitted to the server. Once a response is available, the block triggers `CNF`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_0_3`**: `CLIENT_3_0` is only functionally compatible with a `SERVER_0_3` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 3 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_0_3` with the matching field count.

Comparison with similar function blocks

- SERVER_0_3: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

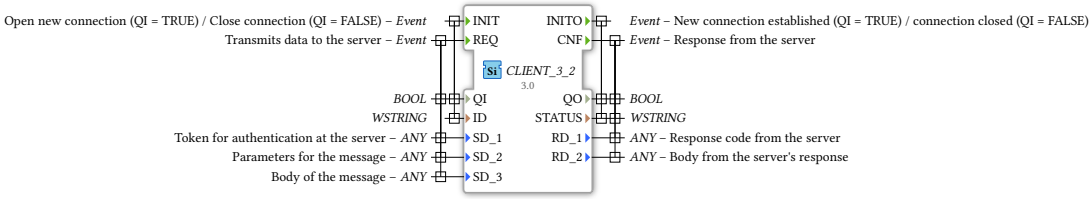
`CLIENT_3_0` provides the generic client variant of the `GEN_CLIENT` family tailored to 3 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.22 CLIENT_3_2

• – ◦ • – ◦ • – ◦ •

Introduction

The CLIENT_3_2 is a function block for HTTP communication, designed as a client with three inputs and two outputs. It enables communication with a server via the HTTP protocol and supports both opening and closing connections as well as sending requests and receiving responses.



Interface Structure

Event Inputs

- **INIT** (Type: EInit) - Opens a new connection (QI = TRUE) or closes a connection (QI = FALSE)
- **REQ** (Type: Event) - Sends data to the server

Event Outputs

- **INITO** (Type: EInit) - Signals that a new connection has been established (QI = TRUE) or a connection has been closed (QI = FALSE)
- **CNF** (Type: Event) - Response from the server

Data Inputs

- **QI** (BOOL) - Quality indicator for INIT and REQ events
- **ID** (WSTRING) - Connection identifier
- **SD_1** (ANY) - Token for authentication with the server
- **SD_2** (ANY) - Parameter for the Message
- **SD_3** (ANY) - Message Body

Data Outputs

- **QO** (BOOL) - Quality indicator for INITO and CNF events
- **STATUS** (WSTRING) - Status information
- **RD_1** (ANY) - Server response code
- **RD_2** (ANY) - Server response body

Adapter

No adapter interfaces available.

Functionality

The CLIENT_3_2 function block operates in two main phases: connection management and data transmission. The INIT event establishes or closes a connection to the server. After a successful connection, data can be sent to the server via the REQ event. The server response is reported back via the CNF event.

Technical Features

- Use of WSTRING for ID and STATUS for international character set support
- ANY type for SD_1, SD_2, SD_3, RD_1, and RD_2 enables flexible data types
- Separate quality indicators for inputs (QI) and outputs (QO)
- Support for authentication tokens (SD_1)

State Overview

1. **Initialization State**: Waiting for INIT event
2. **Connection Establishment**: Processing INIT with QI=TRUE
3. **Ready**: Connection established, waiting for REQ events
4. **Data Transfer**: Sending REQ and waiting for CNF
5. **Connection Termination**: Processing INIT with QI=FALSE

Application Scenarios

- HTTP REST API communication in automation systems

- Web service integration in industrial controllers
- Data exchange with cloud services
- Authenticated communication with Backend Servers

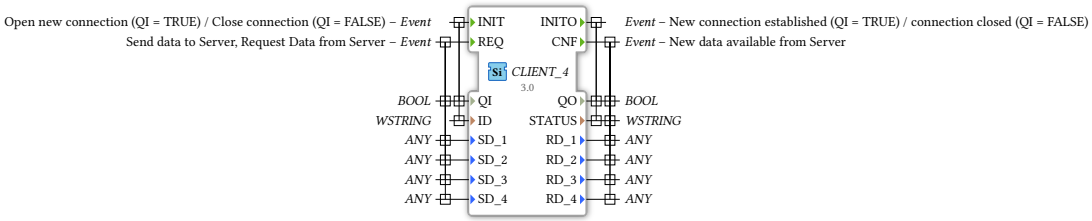
⚖️ Comparison with Similar Modules

Compared to simpler client modules, CLIENT_3_2 offers enhanced functionality through three separate data inputs (SD_1-3) and two data outputs (RD_1-2), enabling structured transmission of authentication data, parameters, and message content.

Conclusion

CLIENT_3_2 is a versatile HTTP client module that, thanks to its flexible data interface and comprehensive status feedback, enables robust communication with HTTP servers in industrial automation environments.

2.11.6.23 CLIENT_4



• – ○ • – ○ • – ○ •

Introduction

The `CLIENT_4` function block is the generic client variant with 4 send and 4 receive data fields for communicating with a matching SERVER_4 block. It transmits 4 data values (`SD_1` `SD_2` `SD_3` `SD_4`) to the server and receives 4 data values (`RD_1` `RD_2` `RD_3` `RD_4`) back. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.
- **SD_2** (ANY): Send value 2, transmitted to the server with `REQ`.
- **SD_3** (ANY): Send value 3, transmitted to the server with `REQ`.
- **SD_4** (ANY): Send value 4, transmitted to the server with `REQ`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.
- **RD_4** (ANY): Receive value 4, delivered by the server with `CNF`.

Functionality

CLIENT_4 initializes a connection to the matching SERVER_4 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, the values SD_1 SD_2 SD_3 SD_4 are transmitted to the server. Once response data (RD_1 RD_2 RD_3 RD_4) is available, the block triggers CNF and provides RD_1 RD_2 RD_3 RD_4.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_4**: CLIENT_4 is only functionally compatible with a SERVER_4 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 4 values and receive exactly 4 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_4 with the matching field count.

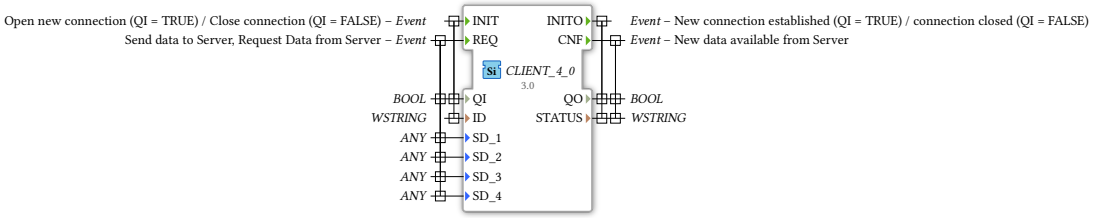
Comparison with similar function blocks

- SERVER_4: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

CLIENT_4 provides the generic client variant of the GEN_CLIENT family tailored to 4 send and 4 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.24 CLIENT_4_0



• – ○ • – ○ • – ○ •

Introduction

The CLIENT_4_0 function block is the generic client variant with 4 send and 0 receive data fields for communicating with a matching SERVER_0_4 block. It transmits 4 data values (SD_1 SD_2 SD_3 SD_4) to the server without receiving any data back. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.
- **SD_2** (ANY): Send value 2, transmitted to the server with `REQ`.
- **SD_3** (ANY): Send value 3, transmitted to the server with `REQ`.
- **SD_4** (ANY): Send value 4, transmitted to the server with `REQ`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`CLIENT_4_0` receives no payload data via `CNF`, only connection/status information).

Functionality

`CLIENT_4_0` initializes a connection to the matching `SERVER_0_4` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, the values `SD_1` `SD_2` `SD_3` `SD_4` are transmitted to the server. Once a response is available, the block triggers `CNF`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_0_4`**: `CLIENT_4_0` is only functionally compatible with a `SERVER_0_4` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 4 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_0_4` with the matching field count.

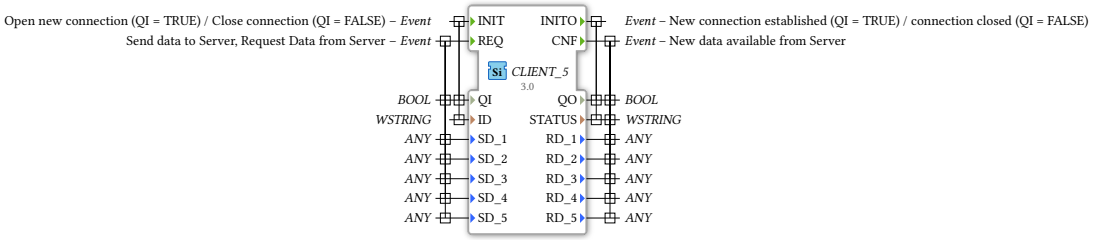
Comparison with similar function blocks

- **SERVER_0_4**: the direct counterpart on the server side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_4_0` provides the generic client variant of the `GEN_CLIENT` family tailored to 4 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.25 CLIENT_5



• – ◦ • – ◦ • – ◦ •

Introduction

The `CLIENT_5` function block is the generic client variant with 5 send and 5 receive data fields for communicating with a matching SERVER_5 block. It transmits 5 data values (`SD_1 SD_2 SD_3 SD_4 SD_5`) to the server and receives 5 data values (`RD_1 RD_2 RD_3 RD_4 RD_5`) back. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1 SD_2 SD_3 SD_4 SD_5`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1 RD_2 RD_3 RD_4 RD_5`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.
- **SD_2** (ANY): Send value 2, transmitted to the server with `REQ`.
- **SD_3** (ANY): Send value 3, transmitted to the server with `REQ`.
- **SD_4** (ANY): Send value 4, transmitted to the server with `REQ`.
- **SD_5** (ANY): Send value 5, transmitted to the server with `REQ`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.
- **RD_4** (ANY): Receive value 4, delivered by the server with `CNF`.
- **RD_5** (ANY): Receive value 5, delivered by the server with `CNF`.

Functionality

`CLIENT_5` initializes a connection to the matching `SERVER_5` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, the values `SD_1 SD_2 SD_3 SD_4 SD_5` are transmitted to the server. Once response data (`RD_1 RD_2 RD_3 RD_4 RD_5`) is available, the block triggers `CNF` and provides `RD_1 RD_2 RD_3 RD_4 RD_5`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.

- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_5**: CLIENT_5 is only functionally compatible with a SERVER_5 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 5 values and receive exactly 5 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_5 with the matching field count.

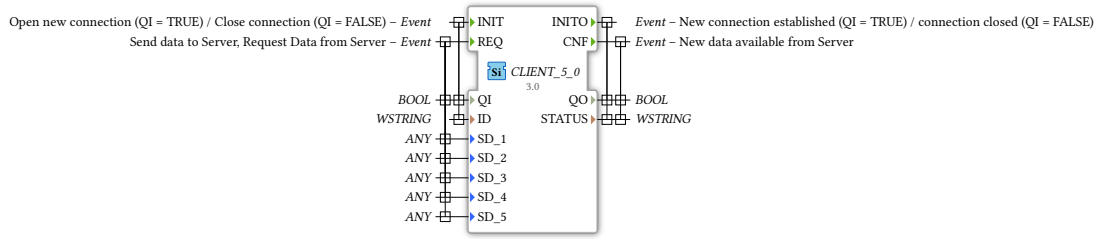
Comparison with similar function blocks

- SERVER_5: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

CLIENT_5 provides the generic client variant of the GEN_CLIENT family tailored to 5 send and 5 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.26 CLIENT_5_0



• – ○ • – ○ • – ○ •

Introduction

The CLIENT_5_0 function block is the generic client variant with 5 send and 0 receive data fields for communicating with a matching SERVER_0_5 block. It transmits 5 data values (SD_1 SD_2 SD_3 SD_4 SD_5) to the server without receiving any data back. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.
- **REQ**: Sends data to the server and requests data from the server, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **CNF**: Signals that new data is available from the server, carries QO, STATUS.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).

- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with REQ.
- **SD_2** (ANY): Send value 2, transmitted to the server with REQ.
- **SD_3** (ANY): Send value 3, transmitted to the server with REQ.
- **SD_4** (ANY): Send value 4, transmitted to the server with REQ.
- **SD_5** (ANY): Send value 5, transmitted to the server with REQ.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (CLIENT_5_0 receives no payload data via CNF, only connection/status information).

Functionality

CLIENT_5_0 initializes a connection to the matching SERVER_0_5 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, the values SD_1 SD_2 SD_3 SD_4 SD_5 are transmitted to the server. Once a response is available, the block triggers CNF.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_0_5**: CLIENT_5_0 is only functionally compatible with a SERVER_0_5 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 5 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_0_5 with the matching field count.

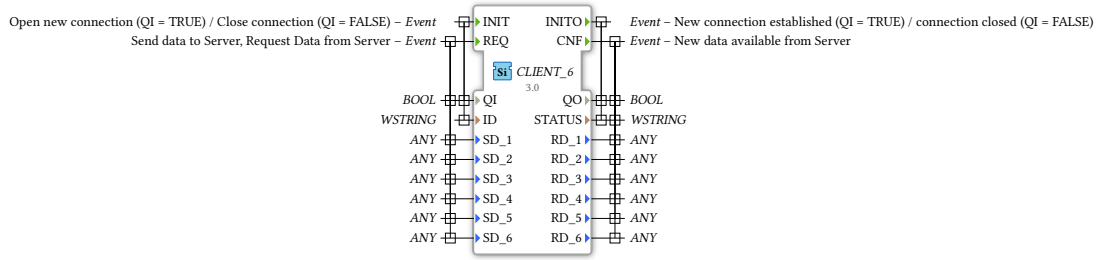
Comparison with similar function blocks

- SERVER_0_5: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

CLIENT_5_0 provides the generic client variant of the GEN_CLIENT family tailored to 5 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.27 CLIENT_6



• – ○ • – ○ • – ○ •

Introduction

The CLIENT_6 function block is the generic client variant with 6 send and 6 receive data fields for communicating with a matching SERVER_6 block. It transmits 6 data values (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6) to the server and receives 6 data values (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6) back. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.
- **REQ**: Sends data to the server and requests data from the server, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5 SD_6.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **CNF**: Signals that new data is available from the server, carries QO, STATUS and RD_1 RD_2 RD_3 RD_4 RD_5 RD_6.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with REQ.
- **SD_2** (ANY): Send value 2, transmitted to the server with REQ.
- **SD_3** (ANY): Send value 3, transmitted to the server with REQ.
- **SD_4** (ANY): Send value 4, transmitted to the server with REQ.
- **SD_5** (ANY): Send value 5, transmitted to the server with REQ.
- **SD_6** (ANY): Send value 6, transmitted to the server with REQ.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with CNF.
- **RD_2** (ANY): Receive value 2, delivered by the server with CNF.
- **RD_3** (ANY): Receive value 3, delivered by the server with CNF.
- **RD_4** (ANY): Receive value 4, delivered by the server with CNF.
- **RD_5** (ANY): Receive value 5, delivered by the server with CNF.
- **RD_6** (ANY): Receive value 6, delivered by the server with CNF.

Functionality

CLIENT_6 initializes a connection to the matching SERVER_6 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 are transmitted to the server. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6) is available, the block triggers CNF and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_6**: CLIENT_6 is only functionally compatible with a SERVER_6 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.

4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 6 values and receive exactly 6 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_6 with the matching field count.

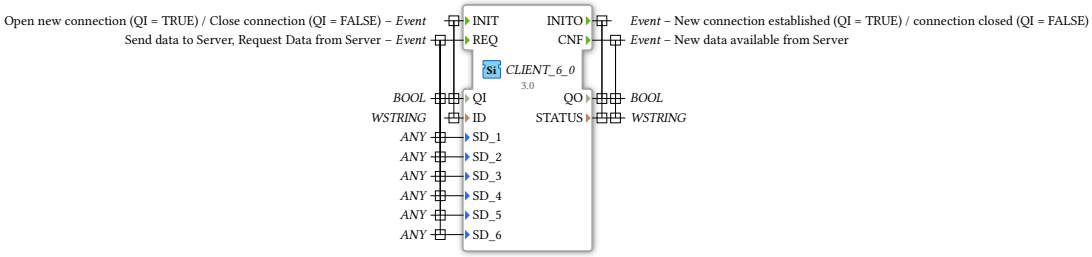
Comparison with similar function blocks

- SERVER_6: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

CLIENT_6 provides the generic client variant of the GEN_CLIENT family tailored to 6 send and 6 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.28 CLIENT_6_0



• – ○ • – ○ • – ○ •

Introduction

The CLIENT_6_0 function block is the generic client variant with 6 send and 0 receive data fields for communicating with a matching SERVER_0_6 block. It transmits 6 data values (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6) to the server without receiving any data back. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.
- **REQ**: Sends data to the server and requests data from the server, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5 SD_6.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **CNF**: Signals that new data is available from the server, carries QO, STATUS.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with REQ.
- **SD_2** (ANY): Send value 2, transmitted to the server with REQ.
- **SD_3** (ANY): Send value 3, transmitted to the server with REQ.
- **SD_4** (ANY): Send value 4, transmitted to the server with REQ.
- **SD_5** (ANY): Send value 5, transmitted to the server with REQ.
- **SD_6** (ANY): Send value 6, transmitted to the server with REQ.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`CLIENT_6_0` receives no payload data via `CNF`, only connection/status information).

Functionality

`CLIENT_6_0` initializes a connection to the matching `SERVER_0_6` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, the values `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` are transmitted to the server. Once a response is available, the block triggers `CNF`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_0_6`**: `CLIENT_6_0` is only functionally compatible with a `SERVER_0_6` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 6 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_0_6` with the matching field count.

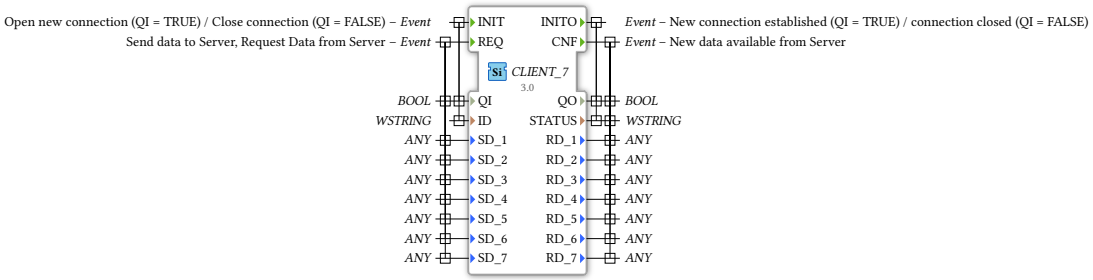
Comparison with similar function blocks

- SERVER_0_6: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_6_0` provides the generic client variant of the `GEN_CLIENT` family tailored to 6 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.29 CLIENT_7



• – ○ • – ○ • – ○ •

Introduction

The `CLIENT_7` function block is the generic client variant with 7 send and 7 receive data fields for communicating with a matching SERVER_7 block. It transmits 7 data values (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7`) to the server and receives 7 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7`) back. Like

all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.
- **REQ**: Sends data to the server and requests data from the server, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **CNF**: Signals that new data is available from the server, carries QO, STATUS and RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with REQ.
- **SD_2** (ANY): Send value 2, transmitted to the server with REQ.
- **SD_3** (ANY): Send value 3, transmitted to the server with REQ.
- **SD_4** (ANY): Send value 4, transmitted to the server with REQ.
- **SD_5** (ANY): Send value 5, transmitted to the server with REQ.
- **SD_6** (ANY): Send value 6, transmitted to the server with REQ.
- **SD_7** (ANY): Send value 7, transmitted to the server with REQ.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with CNF.
- **RD_2** (ANY): Receive value 2, delivered by the server with CNF.
- **RD_3** (ANY): Receive value 3, delivered by the server with CNF.
- **RD_4** (ANY): Receive value 4, delivered by the server with CNF.
- **RD_5** (ANY): Receive value 5, delivered by the server with CNF.
- **RD_6** (ANY): Receive value 6, delivered by the server with CNF.
- **RD_7** (ANY): Receive value 7, delivered by the server with CNF.

Functionality

CLIENT_7 initializes a connection to the matching SERVER_7 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 are transmitted to the server. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7) is available, the block triggers CNF and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_7**: CLIENT_7 is only functionally compatible with a SERVER_7 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 7 values and receive exactly 7 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_7` with the matching field count.

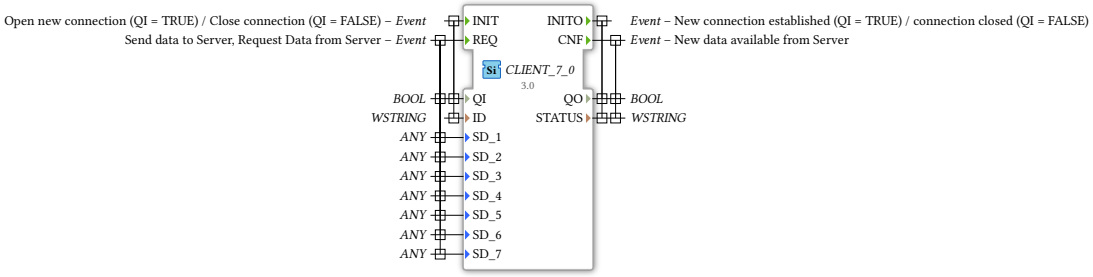
Comparison with similar function blocks

- SERVER_7: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_7` provides the generic client variant of the `GEN_CLIENT` family tailored to 7 send and 7 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.30 CLIENT_7_0



• – ○ • – ○ • – ○ •

Introduction

The `CLIENT_7_0` function block is the generic client variant with 7 send and 0 receive data fields for communicating with a matching SERVER_0_7 block. It transmits 7 data values (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7`) to the server without receiving any data back. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.
- **SD_2** (ANY): Send value 2, transmitted to the server with `REQ`.
- **SD_3** (ANY): Send value 3, transmitted to the server with `REQ`.
- **SD_4** (ANY): Send value 4, transmitted to the server with `REQ`.
- **SD_5** (ANY): Send value 5, transmitted to the server with `REQ`.
- **SD_6** (ANY): Send value 6, transmitted to the server with `REQ`.
- **SD_7** (ANY): Send value 7, transmitted to the server with `REQ`.

Data Outputs

- **QO** (BOOL): Current connection status.

- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`CLIENT_7_0` receives no payload data via `CNF`, only connection/status information).

Functionality

`CLIENT_7_0` initializes a connection to the matching `SERVER_0_7` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, the values `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` are transmitted to the server. Once a response is available, the block triggers `CNF`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_0_7`**: `CLIENT_7_0` is only functionally compatible with a `SERVER_0_7` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 7 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_0_7` with the matching field count.

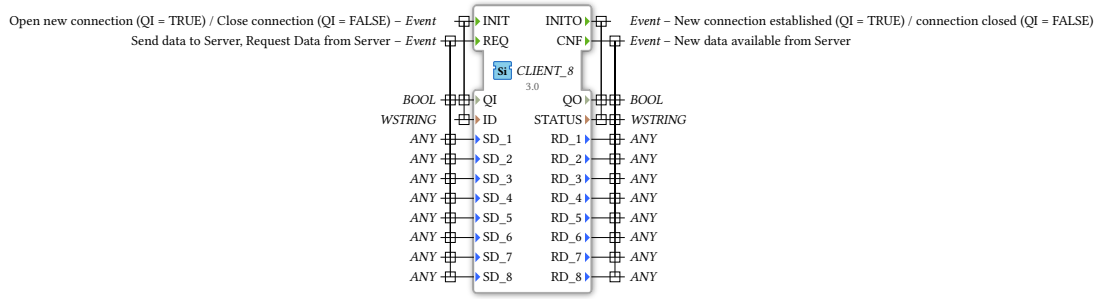
Comparison with similar function blocks

- SERVER_0_7: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_7_0` provides the generic client variant of the `GEN_CLIENT` family tailored to 7 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.31 CLIENT_8



• — ○ • — ○ • — ○ •

Introduction

The `CLIENT_8` function block is the generic client variant with 8 send and 8 receive data fields for communicating with a matching SERVER_8 block. It transmits 8 data values (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8`) to the server and receives 8 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8`) back. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS` and `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.
- **SD_2** (ANY): Send value 2, transmitted to the server with `REQ`.
- **SD_3** (ANY): Send value 3, transmitted to the server with `REQ`.
- **SD_4** (ANY): Send value 4, transmitted to the server with `REQ`.
- **SD_5** (ANY): Send value 5, transmitted to the server with `REQ`.
- **SD_6** (ANY): Send value 6, transmitted to the server with `REQ`.
- **SD_7** (ANY): Send value 7, transmitted to the server with `REQ`.
- **SD_8** (ANY): Send value 8, transmitted to the server with `REQ`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with `CNF`.
- **RD_2** (ANY): Receive value 2, delivered by the server with `CNF`.
- **RD_3** (ANY): Receive value 3, delivered by the server with `CNF`.
- **RD_4** (ANY): Receive value 4, delivered by the server with `CNF`.
- **RD_5** (ANY): Receive value 5, delivered by the server with `CNF`.
- **RD_6** (ANY): Receive value 6, delivered by the server with `CNF`.
- **RD_7** (ANY): Receive value 7, delivered by the server with `CNF`.
- **RD_8** (ANY): Receive value 8, delivered by the server with `CNF`.

Functionality

`CLIENT_8` initializes a connection to the matching `SERVER_8` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, the values `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8` are transmitted to the server. Once response data (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`) is available, the block triggers `CNF` and provides `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_8`**: `CLIENT_8` is only functionally compatible with a `SERVER_8` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 8 values and receive exactly 8 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_8 with the matching field count.

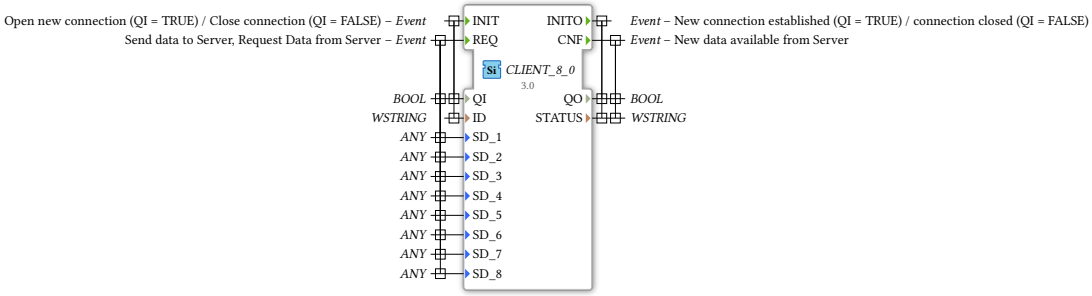
Comparison with similar function blocks

- SERVER_8: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

CLIENT_8 provides the generic client variant of the GEN_CLIENT family tailored to 8 send and 8 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.32 CLIENT_8_0



• – ○ • – ○ • – ○ •

Introduction

The CLIENT_8_0 function block is the generic client variant with 8 send and 0 receive data fields for communicating with a matching SERVER_0_8 block. It transmits 8 data values (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8) to the server without receiving any data back. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.
- **REQ**: Sends data to the server and requests data from the server, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **CNF**: Signals that new data is available from the server, carries QO, STATUS.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with REQ.
- **SD_2** (ANY): Send value 2, transmitted to the server with REQ.
- **SD_3** (ANY): Send value 3, transmitted to the server with REQ.
- **SD_4** (ANY): Send value 4, transmitted to the server with REQ.
- **SD_5** (ANY): Send value 5, transmitted to the server with REQ.
- **SD_6** (ANY): Send value 6, transmitted to the server with REQ.
- **SD_7** (ANY): Send value 7, transmitted to the server with REQ.
- **SD_8** (ANY): Send value 8, transmitted to the server with REQ.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`CLIENT_8_0` receives no payload data via `CNF`, only connection/status information).

Functionality

`CLIENT_8_0` initializes a connection to the matching `SERVER_0_8` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `REQ`, the values `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` are transmitted to the server. Once a response is available, the block triggers `CNF`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_CLIENT'`, the same C++ base as all other `CLIENT_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `SERVER_0_8`**: `CLIENT_8_0` is only functionally compatible with a `SERVER_0_8` block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 8 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_0_8` with the matching field count.

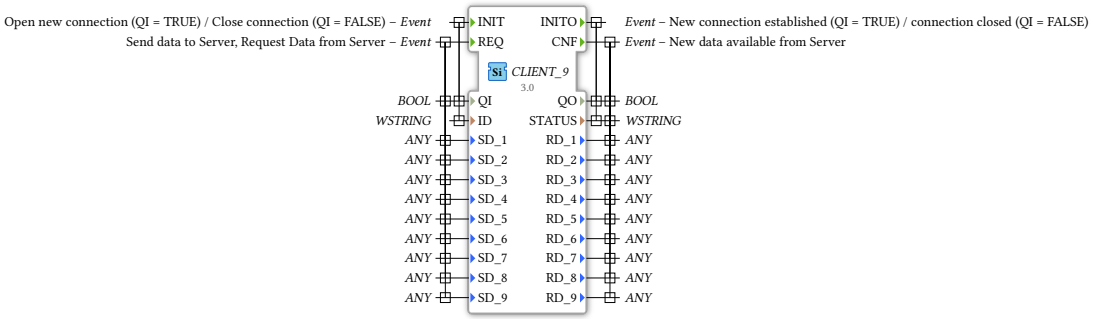
Comparison with similar function blocks

- SERVER_0_8: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_8_0` provides the generic client variant of the `GEN_CLIENT` family tailored to 8 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.33 CLIENT_9



• – ○ • – ○ • – ○ •

Introduction

The `CLIENT_9` function block is the generic client variant with 9 send and 9 receive data fields for communicating with a matching SERVER_9 block. It transmits 9 data values (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9`) to the server and receives 9 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8`

RD_9) back. Like all CLIENT_* blocks, it is based on the generic GEN_CLIENT implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries QI and ID.
- **REQ**: Sends data to the server and requests data from the server, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **CNF**: Signals that new data is available from the server, carries QO, STATUS and RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with REQ.
- **SD_2** (ANY): Send value 2, transmitted to the server with REQ.
- **SD_3** (ANY): Send value 3, transmitted to the server with REQ.
- **SD_4** (ANY): Send value 4, transmitted to the server with REQ.
- **SD_5** (ANY): Send value 5, transmitted to the server with REQ.
- **SD_6** (ANY): Send value 6, transmitted to the server with REQ.
- **SD_7** (ANY): Send value 7, transmitted to the server with REQ.
- **SD_8** (ANY): Send value 8, transmitted to the server with REQ.
- **SD_9** (ANY): Send value 9, transmitted to the server with REQ.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the server with CNF.
- **RD_2** (ANY): Receive value 2, delivered by the server with CNF.
- **RD_3** (ANY): Receive value 3, delivered by the server with CNF.
- **RD_4** (ANY): Receive value 4, delivered by the server with CNF.
- **RD_5** (ANY): Receive value 5, delivered by the server with CNF.
- **RD_6** (ANY): Receive value 6, delivered by the server with CNF.
- **RD_7** (ANY): Receive value 7, delivered by the server with CNF.
- **RD_8** (ANY): Receive value 8, delivered by the server with CNF.
- **RD_9** (ANY): Receive value 9, delivered by the server with CNF.

Functionality

CLIENT_9 initializes a connection to the matching SERVER_9 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 are transmitted to the server. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9) is available, the block triggers CNF and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_9**: CLIENT_9 is only functionally compatible with a SERVER_9 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `REQ`/`CNF` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 9 values and receive exactly 9 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a `SERVER_9` with the matching field count.

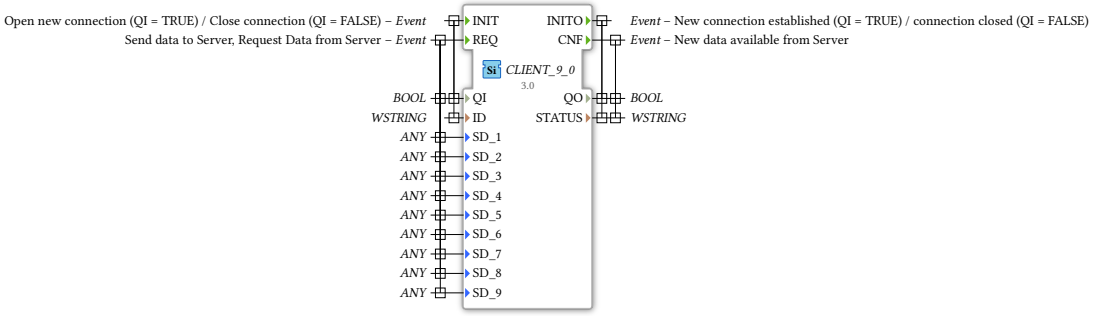
Comparison with similar function blocks

- SERVER_9: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`CLIENT_9` provides the generic client variant of the `GEN_CLIENT` family tailored to 9 send and 9 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.34 CLIENT_9_0



• – ○ • – ○ • – ○ •

Introduction

The `CLIENT_9_0` function block is the generic client variant with 9 send and 0 receive data fields for communicating with a matching SERVER_0_9 block. It transmits 9 data values (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9`) to the server without receiving any data back. Like all `CLIENT_*` blocks, it is based on the generic `GEN_CLIENT` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Opens a new connection (QI = TRUE) or closes an existing connection (QI = FALSE), carries `QI` and `ID`.
- **REQ**: Sends data to the server and requests data from the server, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **CNF**: Signals that new data is available from the server, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the server with `REQ`.

- **SD_2** (ANY): Send value 2, transmitted to the server with REQ.
- **SD_3** (ANY): Send value 3, transmitted to the server with REQ.
- **SD_4** (ANY): Send value 4, transmitted to the server with REQ.
- **SD_5** (ANY): Send value 5, transmitted to the server with REQ.
- **SD_6** (ANY): Send value 6, transmitted to the server with REQ.
- **SD_7** (ANY): Send value 7, transmitted to the server with REQ.
- **SD_8** (ANY): Send value 8, transmitted to the server with REQ.
- **SD_9** (ANY): Send value 9, transmitted to the server with REQ.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (CLIENT_9_0 receives no payload data via CNF, only connection/status information).

Functionality

CLIENT_9_0 initializes a connection to the matching SERVER_0_9 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via REQ, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 are transmitted to the server. Once a response is available, the block triggers CNF.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_CLIENT', the same C++ base as all other CLIENT_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart SERVER_0_9**: CLIENT_9_0 is only functionally compatible with a SERVER_0_9 block on the server side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: REQ/CNF cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 9 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Client-side connection** to a communication partner already laid out as a SERVER_0_9 with the matching field count.

Comparison with similar function blocks

- SERVER_0_9: the direct counterpart on the server side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

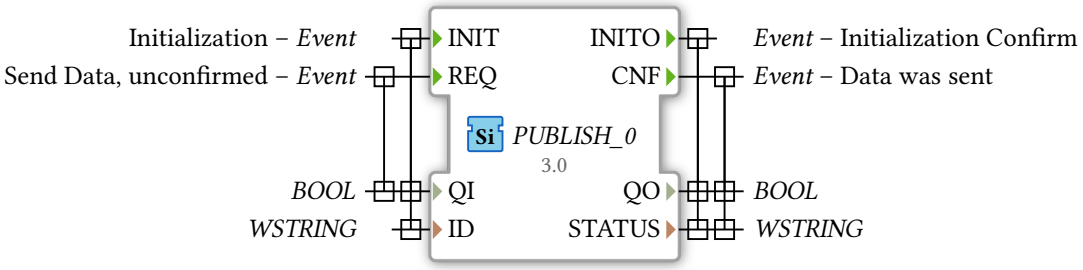
CLIENT_9_0 provides the generic client variant of the GEN_CLIENT family tailored to 9 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.35 PUBLISH_0

• − ◦ • − ◦ • − ◦ •

Introduction

The PUBLISH_0 function block is used to publish data to one or more SUBSCRIBE_0 blocks. It enables unacknowledged communication between different components in a 4diac system.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **REQ**: Data transmission request (unacknowledged) with associated data QI

Event Outputs

- **INITO**: Initialization acknowledgment with associated data QO and STATUS
- **CNF**: Data transmission acknowledgment with associated data QO and STATUS

Data Inputs

- **QI** (BOOL): Block activation quality indicator
- **ID** (WSTRING): Communication identification string

Data Outputs

- **QO** (BOOL): Block output quality indicator
- **STATUS** (WSTRING): Status information about the executed operation

Adapters

No adapter interfaces available.

How it Works

The PUBLISH_0 block enables the publication of data via a publish-subscribe pattern. Upon initialization (INIT), the block is configured with a specific ID. Upon a send request (REQ), the data is distributed to all registered subscribers. Communication is unacknowledged, meaning no response is expected from the recipients.

Technical Features

- Uses WSTRING data types for ID and STATUS for international character support
- Implements unconfirmed communication
- Supports multiple subscribers via a single Publish instance
- Generic implementation as GEN_PUBLISH

State Overview

The block goes through the following main states:

1. **Uninitialized**: Block is operational but not configured
2. **Initialized**: Block is configured and ready to send
3. **Ready to Send**: Block can distribute data to subscribers

Application Scenarios

- Distribution of sensor data to multiple processing nodes
- Broadcast of system status information
- Distribution of control commands to multiple actuators
- Data exchange in distributed control systems

⚖️ Comparison with Similar Blocks

Compared to confirmed communication blocks, PUBLISH_0 offers:

- Higher performance through unconfirmed transmission
- Lower latency
- Simpler implementation
- No guarantee of Delivery

Conclusion

The PUBLISH_0 block is an efficient tool for unacknowledged communication in distributed control systems. Its simple interface and support for multiple receivers make it ideal for scenarios where performance is more important than guaranteed delivery.

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

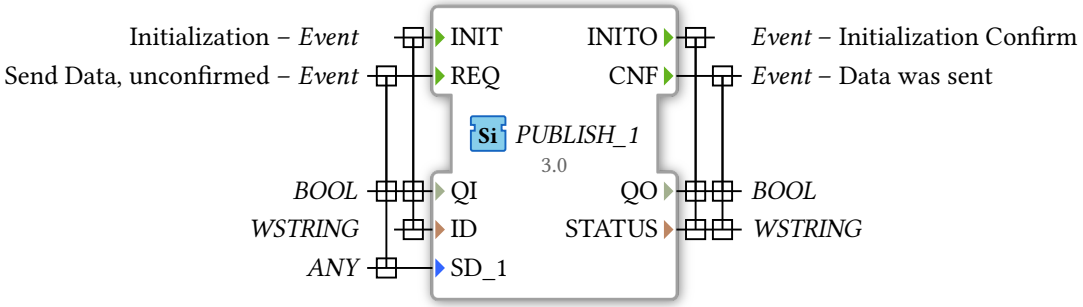
]

2.11.6.36 PUBLISH_1

• – ◦ • – ◦ • – ◦ •

Introduction

The PUBLISH_1 function block is used to send data to one or more SUBSCRIBE_1 blocks. It implements a publish-subscribe communication pattern, in which data is distributed to all subscribed recipients without acknowledgment.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **REQ**: Data send request (unacknowledged) with associated data QI and SD_1

Event Outputs

- **INITO**: Initialization acknowledgement with associated data QO and STATUS
- **CNF**: Data sent acknowledgement with associated data QO and STATUS

Data Inputs

- **QI** (BOOL): Qualifier for initialization and operation
- **ID** (WSTRING): Publish channel identifier
- **SD_1** (ANY): Data to be sent (any data type)

Data Outputs

- **QO** (BOOL): Output status qualifier
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

No adapter interfaces are available.

Functionality

The PUBLISH_1 block enables the distribution of data to multiple recipients via a publish-subscribe mechanism. Upon initialization (INIT event), the block is configured with a specific ID. Upon receiving a REQ event, the data passed to SD_1 is sent to all SUBSCRIBE_1 blocks with the same ID. Communication is unacknowledged, meaning no response is expected from the recipients.

Technical Features

- Using the ANY data type for SD_1 allows sending arbitrary data structures
- The WSTRING data type for ID and STATUS supports international character sets
- Unacknowledged communication for reduced latency
- Generic implementation via the GenericClassName attribute

State Overview

1. **Not Initialized**: Block awaits INIT event
2. **Initialized**: Block ready to receive REQ events
3. **Send Active**: Processing REQ events and distributing data

Application Scenarios

- Distributing sensor data to multiple processing nodes
- Broadcasting system status information
- Distributing configuration data to multiple controllers
- Multi-cast communication in distributed automation systems

⚖️ Comparison with Similar Blocks

Compared to acknowledged communication blocks, PUBLISH_1 offers reduced latency through unacknowledged transmission. Compared to point-to-point connections, it enables efficient one-to-many communication without explicitly establishing connections to each recipient.

Conclusion

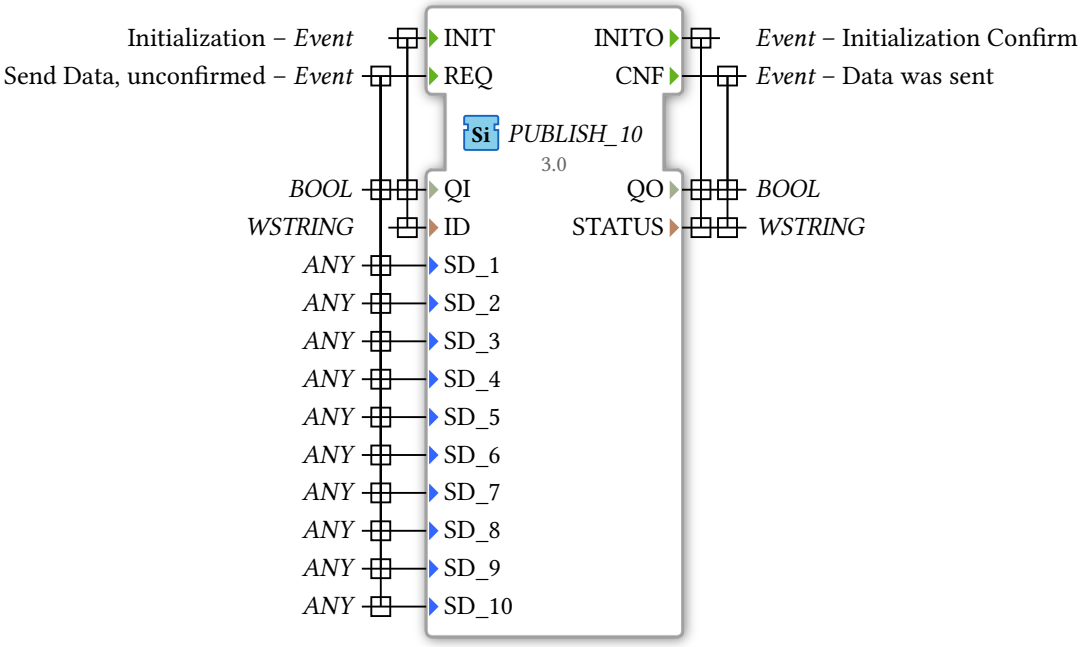
The PUBLISH_1 block is an efficient tool for unacknowledged one-to-many communication in distributed automation systems. Its flexible data type support and simple configuration make it ideal for scenarios where data needs to be distributed to multiple recipients without requiring acknowledgments.

2.11.6.37 PUBLISH_10

• – ◦ • – ◦ • – ◦ •

Introduction

The PUBLISH_10 function block is used to distribute data to one or more SUBSCRIBE_10 blocks. It enables the unacknowledged transmission of up to 10 different data values using a publish-subscribe communication pattern.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **REQ**: Send request for unacknowledged data transmission with 10 data variables

Event Outputs

- **INITO**: Acknowledgement of initialization with status information
- **CNF**: Acknowledgement that data has been sent

Data Inputs

- **QI** (BOOL): Qualifier for initialization (TRUE = enable, FALSE = disable)
- **ID** (WSTRING): Identification string for the communication channel
- **SD_1** to **SD_10** (ANY): 10 different data values of any type to be sent

Data Outputs

- **QO** (BOOL): Qualifier output (TRUE = successful, FALSE = failed)
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

No adapter interfaces are available.

Functionality

The PUBLISH_10 block operates according to the publish-subscribe principle. After successful initialization with the INIT event, up to 10 different data values can be distributed simultaneously to all registered subscribers via the REQ event. Data transmission is unacknowledged, meaning that the block does not expect any feedback from the recipients.

Technical Features

- Supports up to 10 different data sources simultaneously
- Uses the generic data type ANY for maximum flexibility
- Unicode string support for ID and STATUS
- Unacknowledged communication for reduced latency
- Generic implementation via the GEN_PUBLISH base class

State Overview

1. **Not Initialized**: Block is inactive
2. **Initialized**: Block is ready for data distribution
3. **Ready to Send**: Processes REQ events and distributes data

Application Scenarios

- Distribution of sensor data to multiple consumers
- Broadcast communication in distributed systems
- Data distribution in real-time control systems
- Multi-consumer data pipeline architectures

⚖️ Comparison with Similar Blocks

Compared to acknowledged communication blocks, PUBLISH_10 offers reduced latency through unacknowledged transmission. Compared to blocks with fewer data channels, it enables the simultaneous distribution of multiple data streams.

Conclusion

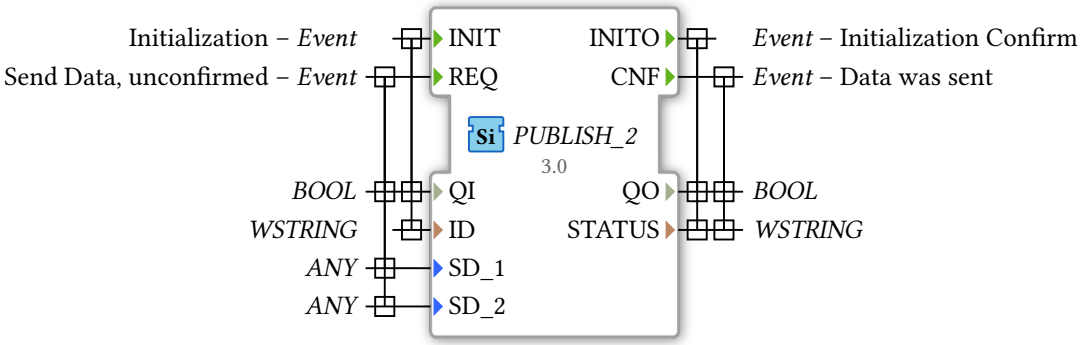
The PUBLISH_10 function block is a powerful solution for unacknowledged multi-data publish-subscribe communication in IEC 61499 systems. Its flexibility in data acquisition and support for up to 10 parallel data channels make it ideal for complex data distribution tasks in industrial automation systems.

2.11.6.38 PUBLISH_2

• – ◦ • – ◦ • – ◦ •

Introduction

The PUBLISH_2 function block is used to send data to one or more SUBSCRIBE_2 blocks. It enables unacknowledged communication between different components in a distributed system.



Interface Structure

Event Inputs

- **INIT**: Initialization event with the associated data QI and ID
- **REQ**: Request event to send data with the associated data QI, SD_1, and SD_2

Event Outputs

- **INITO**: Confirmation of initialization with the associated data QO and STATUS
- **CNF**: Confirmation that data has been sent, with the associated data QO and STATUS

Data Inputs

- **QI** (BOOL): Qualifier for initialization and operation
- **ID** (WSTRING): Identifier for communication
- **SD_1** (ANY): First data value to be sent (any data type)
- **SD_2** (ANY): Second data value to be sent (any data type)

Data Outputs

- **QO** (BOOL): Qualifier for the output state

- **STATUS** (WSTRING): Status information about the operating state

Adapters

No adapters available.

Functionality

The PUBLISH_2 block initializes itself via the INIT event and then sends data via the REQ event. The data SD_1 and SD_2 are distributed to all connected SUBSCRIBE_2 blocks. Communication is unacknowledged, meaning that no response is expected from the receivers.

Technical Features

- Supports any data type for SD_1 and SD_2
- Uses WSTRING for ID and STATUS for international character support
- Unacknowledged communication mode (Publish/Subscribe pattern)
- Flexible data transmission with two separate data channels

State Overview

1. **Not Initialized**: Block waits for INIT event
2. **Initialized**: Block is ready and can process REQ events
3. **Ready to Send**: Processes REQ events and sends data to the subscriber

Application Scenarios

- Distributed systems with multiple data receivers
- Real-time data distribution without an acknowledgment mechanism
- Systems with variable data types
- Communication between different control components

⚖️ Comparison with Similar Blocks

Compared to acknowledged communication blocks, PUBLISH_2 offers higher performance but lower reliability. Its support for any data type makes it more flexible than type-specific communication blocks.

Conclusion

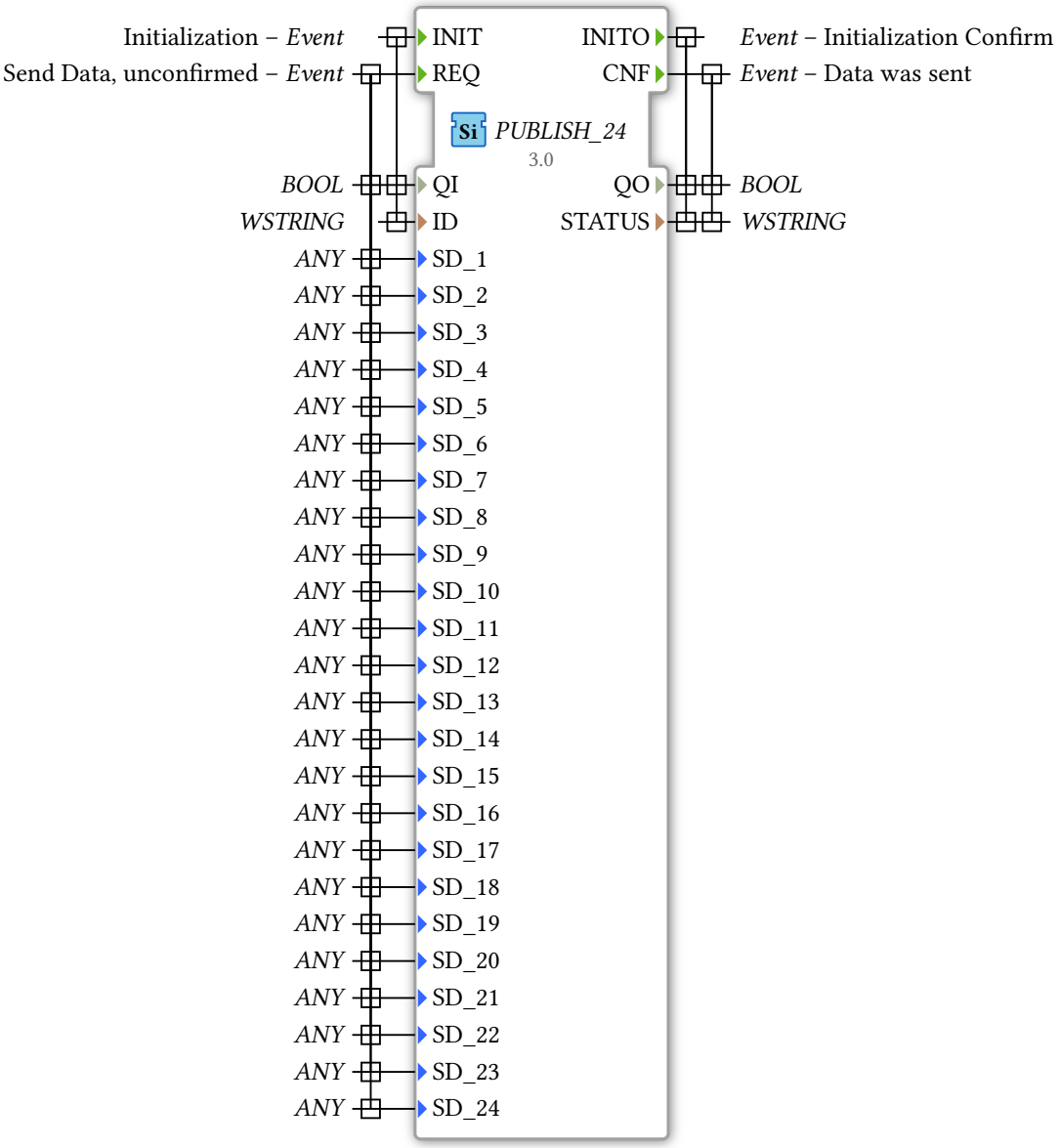
The PUBLISH_2 block is an efficient tool for unacknowledged data distribution in distributed systems. Its flexibility in data type support and ease of use make it ideal for scenarios where performance is more important than guaranteed delivery.

2.11.6.39 PUBLISH_24

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `PUBLISH_24` distributes data to one or more `SUBSCRIBE_24` blocks. It allows the unconfirmed transmission of up to 24 data values using a publish-subscribe communication pattern.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation. Associated with `QI`, `ID`.
- **REQ**: send data, unconfirmed. Associated with `QI`, `SD_1` to `SD_24`.

Event Outputs

- **INITO** (`EInit`): initialisation confirm. Associated with `QO`, `STATUS`.
- **CNF**: data was sent. Associated with `QO`, `STATUS`.

Data Inputs

- **QI** (`BOOL`): event input qualifier
- **ID** (`WSTRING`): identification string of the communication channel
- **SD_1** to **SD_24** (`ANY`): data values of any type to send

Data Outputs

- **QO** (`BOOL`): event output qualifier
- **STATUS** (`WSTRING`): status information

Adapters

No adapter interfaces available.

Functionality

After initialisation with `INIT`, `REQ` distributes up to 24 data values to all registered subscribers. The transmission is unconfirmed; `CNF` only reports that the data was sent.

Technical Details

- Generic block (generic class `GEN_PUBLISH`): the number of data ports is 24 here, their types (`ANY`) are fixed at instantiation
- Unicode strings (`WSTRING`) for `ID` and `STATUS`
- Counterpart: `SUBSCRIBE_24` with `RD_1` to `RD_24`

Application Scenarios

- Distributing many related values (e.g. measured values or states) with a single event to other applications or devices

⚖ Comparison with Similar Building Blocks

`PUBLISH_10` transmits up to 10 data values, `PUBLISH_24` up to 24.

Conclusion

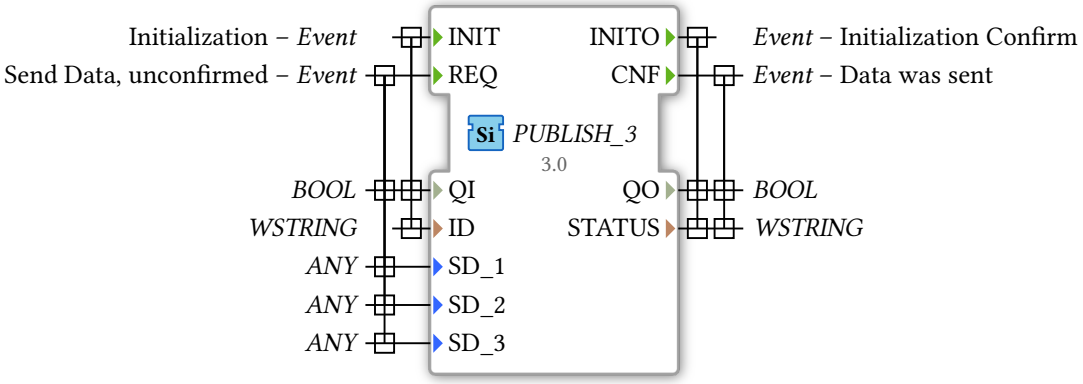
`PUBLISH_24` is the unconfirmed publish variant for up to 24 data values.

2.11.6.40 PUBLISH_3

• – ◦ • – ◦ • – ◦ •

Introduction

The PUBLISH_3 function block is used to distribute data to one or more SUBSCRIBE_3 blocks. It enables the unacknowledged transmission of up to three different data values to subscribers in a distributed system.



Interface Structure

Event Inputs

- **INIT**: Initialization event with the associated data QI and ID
- **REQ**: Send request for data (unacknowledged) with the associated data QI, SD_1, SD_2, and SD_3

Event Outputs

- **INITO**: Acknowledgement of initialization with the associated data QO and STATUS
- **CNF**: Acknowledgement that data has been sent, with the associated data QO and STATUS

Data Inputs

- **QI** (BOOL): Qualifier for initialization and send operations
- **ID** (WSTRING): Publisher identifier
- **SD_1** (ANY): First data value to be sent (any data type)

- **SD_2** (ANY): Second data value to be sent (Any data type)
- **SD_3** (ANY): Third data value to be sent (any data type)

Data Outputs

- **QO** (BOOL): Qualifier for output events
- **STATUS** (WSTRING): Status information about the executed operation

Adapters

No adapter interfaces are available.

Functionality

The PUBLISH_3 block is first initialized via the INIT event, where the ID parameter defines the unique identifier for this publisher. After successful initialization, the block acknowledges with INITO. Data can then be sent via the REQ event, with up to three different data values (SD_1, SD_2, SD_3) being transmitted simultaneously. The transmission is unacknowledged, but the block issues a CNF event after sending.

Technical Features

- Supports the ANY data type for all three data channels, offering maximum flexibility in the types of data to be transmitted
- Uses WSTRING for ID and STATUS, thus enabling international character sets
- Unacknowledged communication for reduced latency
- Can distribute data to multiple subscribers simultaneously

State Overview

1. **Not Initialized**: Block is operational but not configured for data exchange
2. **Initialized**: Block is configured and ready to send data
3. **Sending**: Block is processing a send request
4. **Ready**: Block has successfully sent data and is ready for further requests

Application Scenarios

- Distribution of sensor data to multiple processing nodes
- Broadcast of control information in distributed systems
- Data distribution in production facilities with multiple consumers
- Measurement distribution in monitoring systems

⚖️ Comparison with Similar Blocks

Compared to simpler PUBLISH blocks, PUBLISH_3 offers the ability to send up to three different data values simultaneously This increases efficiency in systems with multiple related data points. Using ANY data types offers greater flexibility than type-specific implementations.

Conclusion

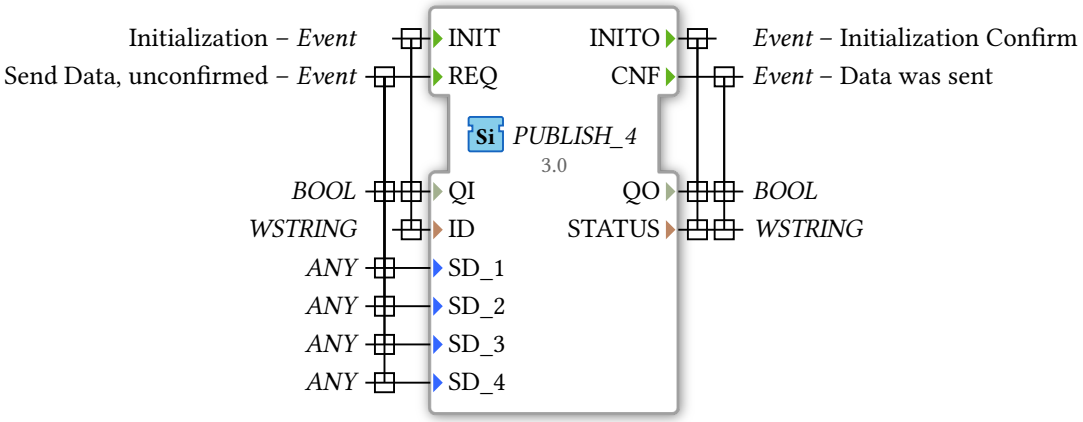
The PUBLISH_3 function block is a powerful solution for distributing data in distributed automation systems. Its ability to transfer up to three different data values simultaneously, combined with the flexibility of ANY data types, makes it particularly suitable for complex applications where multiple related pieces of information need to be distributed efficiently.

2.11.6.41 PUBLISH_4

• – ◦ • – ◦ • – ◦ •

Introduction

The PUBLISH_4 function block is used to publish data to one or more SUBSCRIBE_4 blocks. It enables the unacknowledged transmission of up to four different data values using a publish-subscribe communication pattern.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **REQ**: Data send request (unacknowledged) with associated data QI, SD_1, SD_2, SD_3, and SD_4

Event Outputs

- **INITO**: Initialization acknowledgement with associated data QO and STATUS
- **CNF**: Data sent acknowledgement with associated data QO and STATUS

Data Inputs

- **QI** (BOOL): Qualifier for initialization and send operations
- **ID** (WSTRING): Identification string for communication
- **SD_1** (ANY): First data value to be sent (any data type)
- **SD_2** (ANY): Second data value to be sent (Any data type)
- **SD_3** (ANY): Third data value to be sent (any data type)
- **SD_4** (ANY): Fourth data value to be sent (any data type)

Data Outputs

- **QO** (BOOL): Qualifier for output events
- **STATUS** (WSTRING): Status information as a wide string

Adapters

No adapter interfaces are available.

Functionality

The PUBLISH_4 block initializes itself via the INIT event and confirms this with INITO. After successful initialization, up to four different data values (SD_1 to SD_4) can be sent simultaneously to all connected SUBSCRIBE_4 blocks via the REQ event. Data transmission is unacknowledged; the block only confirms that the data was sent (CNF), not that it was received.

Technical Features

- Supports any data type (ANY) for all four data channels
- Uses wide strings (WSTRING) for ID and STATUS
- Unacknowledged send mode (fire-and-forget)
- Maximum capacity of four simultaneous data values
- Generic implementation as GEN_PUBLISH

State Overview

1. **Not Initialized**: Block awaits INIT event
2. **Initialized**: Block ready to receive REQ events
3. **Send Active**: Processing REQ events and triggering CNF

Application Scenarios

- Distributed systems with a publish-subscribe architecture
- Real-time data distribution to multiple receivers
- Systems with unidirectional data communication
- Applications requiring simultaneous transmission of multiple data values

⚖️ Comparison with Similar Blocks

Compared to acknowledged communication blocks, PUBLISH_4 offers lower latency due to its unacknowledged send mode. Compared to blocks with fewer data channels, it enables the simultaneous transmission of up to four different data values.

Conclusion

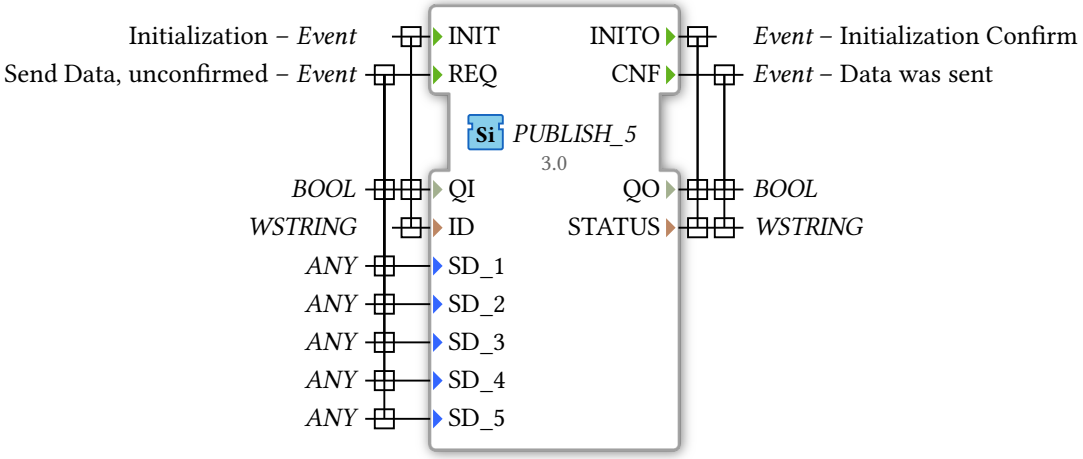
The PUBLISH_4 function block is an efficient solution for unacknowledged data distribution in distributed systems. Its flexibility in supporting any data type and the ability to send four data values in parallel make it particularly suitable for complex communication scenarios in industrial automation systems.

2.11.6.42 PUBLISH_5

• − ◦ • − ◦ • − ◦ •

Introduction

The PUBLISH_5 function block is used to publish data to one or more SUBSCRIBE_5 blocks. It allows the transmission of up to five different data values using a publish-subscribe communication pattern.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **REQ**: Request event to send data (unacknowledged) with associated data QI, SD_1 to SD_5

Event Outputs

- **INITO**: Acknowledgement of initialization with associated data QO and STATUS
- **CNF**: Acknowledgement that data has been sent with associated data QO and STATUS

Data Inputs

- **QI** (BOOL): Qualifier for initialization/operation
- **ID** (WSTRING): Identifier for the publish channel
- **SD_1** (ANY): First data value to be sent
- **SD_2** (ANY): Second data value to be sent
- **SD_3** (ANY): Third data value to be sent Data Value
- **SD_4** (ANY): Fourth data value to be sent
- **SD_5** (ANY): Fifth data value to be sent

Data Outputs

- **QO** (BOOL): Qualifier for output status
- **STATUS** (WSTRING): Status information about the operation

Adapters

No adapter interfaces are available.

Functionality

The PUBLISH_5 block initializes itself via the INIT event and configures the publish channel with the specified ID. After successful initialization, data can be sent to all connected SUBSCRIBE_5 blocks via the REQ event. Data SD_1 to SD_5 is transmitted simultaneously. Each transmission is acknowledged by a CNF event.

Technical Features

- Supports the ANY data type for maximum flexibility in the data to be sent
- Uses WSTRING for status messages and channel IDs
- Implements an unconfirmed sending procedure
- Provides space for up to five different data values per send

State Overview

1. **Not Initialized**: Block waits for INIT event
2. **Initialized**: Block is ready for REQ events
3. **Send**: Processes REQ and sends data to subscriber
4. **Confirm**: Sends CNF after successful data transmission

Application Scenarios

- Distributed systems with publisher-subscriber architecture
- Data distribution to multiple receivers in real-time systems
- Flexible messaging between different control components
- Systems with variable data structures (through ANY type support)

⚖️ Comparison with Similar Blocks

Compared to simpler PUBLISH blocks, PUBLISH_5 offers the Ability to send up to five different data values simultaneously. The use of the ANY data type allows for greater flexibility than type-specific publish blocks.

Conclusion

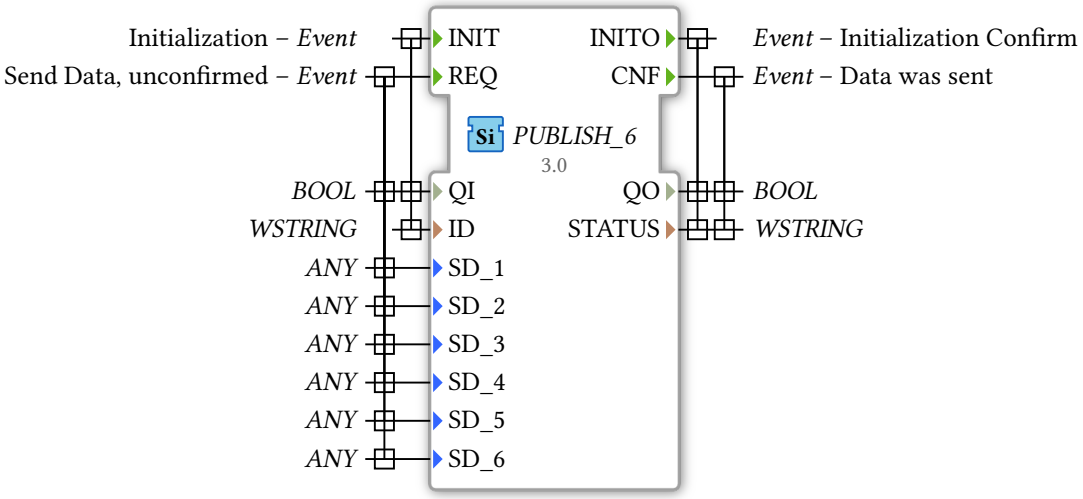
The PUBLISH_5 function block is a powerful solution for publish-subscribe communication in distributed automation systems. Its flexibility, thanks to the ANY data type and the ability to send multiple data values simultaneously, makes it particularly suitable for complex data distribution tasks in industrial control systems.

2.11.6.43 PUBLISH_6

• – ○ • – ○ • – ○ •

Introduction

The PUBLISH_6 function block is used to publish data to one or more SUBSCRIBE_6 blocks. It allows the transmission of up to six different data values via a publish-subscribe communication pattern and offers both acknowledged and unacknowledged send operations.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **REQ**: Data transmission request (unacknowledged) with associated data QI and SD_1 to SD_6

Event Outputs

- **INITO**: Initialization acknowledgment with associated data QO and STATUS
- **CNF**: Acknowledgement that data has been sent, with associated data QO and STATUS

Data Inputs

- **QI** (BOOL): Quality indicator for initialization and transmission operations
- **ID** (WSTRING): Identifier for the communication channel
- **SD_1** to **SD_6** (ANY): Six different data values of any type to be sent

Data Outputs

- **QO** (BOOL): Quality indicator for Output Operations
- **STATUS** (WSTRING): Status information about the executed operation

Adapter

No adapter interfaces are available.

Functionality

The PUBLISH_6 block initializes itself via the INIT event and confirms the initialization with INITO. Data can be sent via the REQ event, with up to six different data values (SD_1 to SD_6) being transmitted simultaneously. A CNF event is issued after successful data transmission. The block supports the publish-subscribe pattern, allowing multiple subscribers to receive the sent data.

Technical Features

- Support for up to six different data values of any type (ANY)
- Use of WSTRING for status messages and channel identification
- Unacknowledged send operations (REQ) with subsequent acknowledgment (CNF)
- Generic implementation for flexible reuse

State Overview

1. **Not Initialized**: Block waits for INIT event
2. **Initialized**: Block ready to receive REQ events
3. **Send Operation**: Processing the send request and data transmission
4. **Acknowledgement**: Output of CNF after successful transmission

Application Scenarios

- Distribution of sensor data to multiple processing nodes
- Broadcast communication in distributed control systems
- Data distribution in production facilities with multiple consumers
- Flexible messaging in IoT applications

⚖️ Comparison with Similar Blocks

Compared to simpler Publish blocks, PUBLISH_6 offers the ability to simultaneously publish up to six different data values This increases efficiency when transmitting related data. The use of the ANY data type allows for maximum flexibility in the data formats to be transmitted.

Conclusion

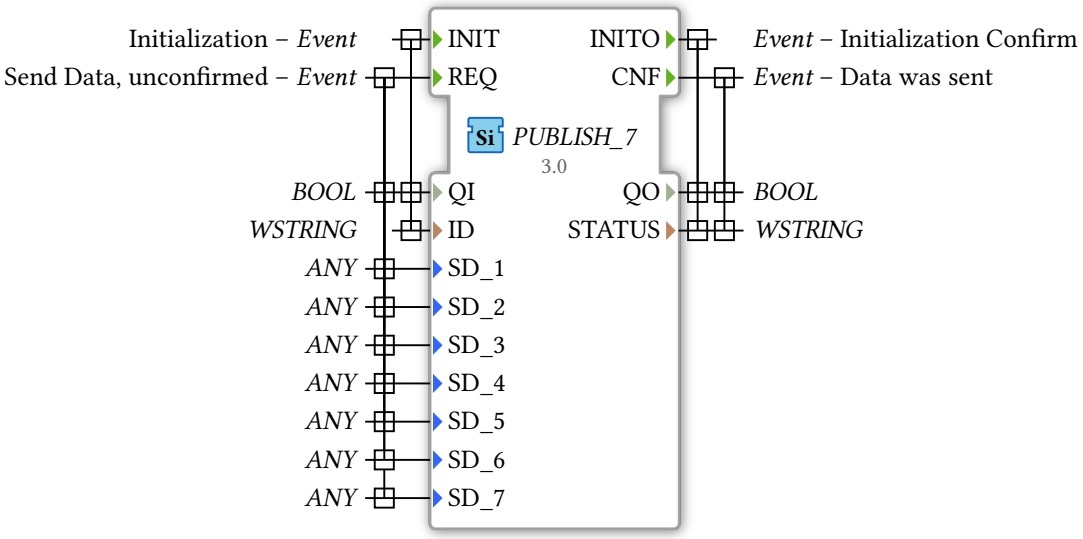
The PUBLISH_6 function block is a powerful tool for publish-subscribe communication in distributed automation systems. Its ability to transmit multiple data values simultaneously makes it particularly efficient for applications where related datasets need to be distributed. Flexible type support and robust error handling through status messages make it a reliable solution for industrial communication requirements.

2.11.6.44 PUBLISH_7

• – ◦ • – ◦ • – ◦ •

Introduction

The PUBLISH_7 function block is used to distribute data to one or more SUBSCRIBE_7 blocks. It enables the unacknowledged transmission of up to seven different data values via a publish-subscribe architecture.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **REQ**: Send request for data (unacknowledged) with seven data variables

Event Outputs

- **INITO**: Initialization acknowledgement with QO and STATUS
- **CNF**: Acknowledgement that data has been sent with QO and STATUS

Data Inputs

- **QI** (BOOL): Quality indicator for initialization
- **ID** (WSTRING): Identifier for the publish channel
- **SD_1** to **SD_7** (ANY): Seven different data values of any type

Data Outputs

- **QO** (BOOL): Quality indicator for output state
- **STATUS** (WSTRING): Status information as a Unicode string

Adapter

No adapter interfaces available.

Functionality

The PUBLISH_7 block initializes itself via the INIT event with a specific channel ID. After successful initialization, up to seven data values (SD_1 to SD_7) can be distributed simultaneously to all connected SUBSCRIBE_7 blocks via the REQ event. Data transmission is unacknowledged; the CNF output only confirms that the data was sent, not that it was received.

Technical Features

- Supports the ANY data type for maximum flexibility
- Uses WSTRING for ID and STATUS for international character set support
- Provides seven independent data channels
- Implemented according to the IEC 61499-2 standard
- Uses GenericClassName 'GEN_PUBLISH' for generic implementation

State Overview

1. **Not Initialized**: Block awaits INIT event
2. **Initialized**: Block ready for REQ events
3. **Ready to Send**: Processes REQ events and distributes data

Application Scenarios

- Distribution of sensor data to multiple consumers
- Broadcast communication in distributed systems
- Data distribution in production facilities
- Measurement distribution in monitoring systems

⚖️ Comparison with Similar Blocks

Compared to simpler PUBLISH blocks, PUBLISH_7 offers the ability to distribute seven different data values in parallel, while simpler variants typically support only one or a few data channels.

Conclusion

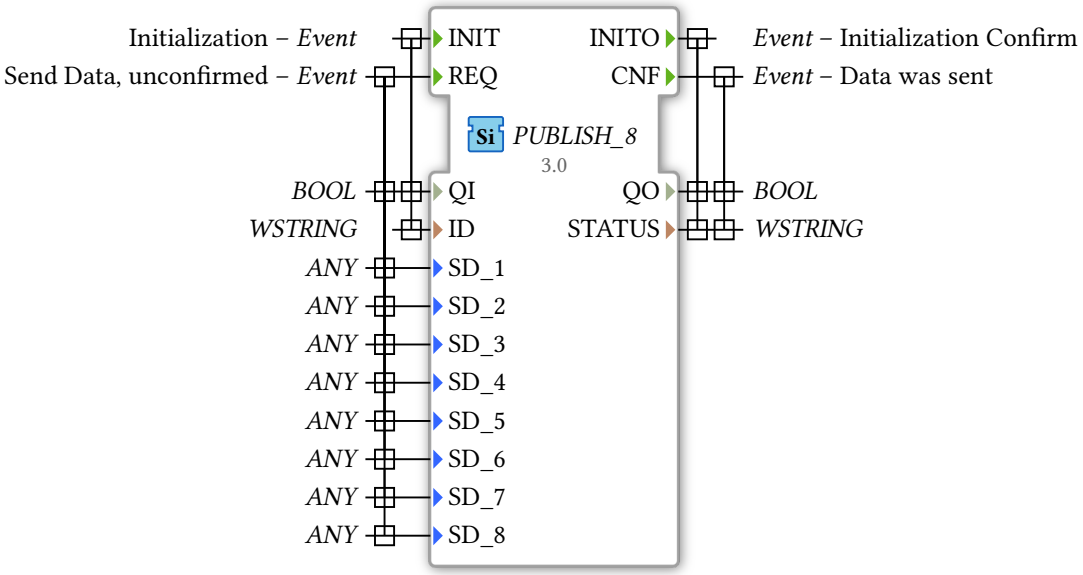
The PUBLISH_7 function block is a powerful solution for publish-subscribe communication in IEC 61499 systems, which, thanks to its seven parallel data channels and flexibility in data types, is particularly suitable for complex data distribution tasks.

2.11.6.45 PUBLISH_8

• − ◦ • − ◦ • − ◦ •

Introduction

The PUBLISH_8 function block is used to send data to one or more SUBSCRIBE_8 blocks. It implements a publisher-subscriber communication pattern and allows the distribution of up to eight different data values to subscribed recipients.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **REQ**: Send request for data (unacknowledged) with all eight SD_x data values

Event Outputs

- **INITO**: Initialization acknowledgment with QO and STATUS
- **CNF**: Acknowledgement that data has been sent, with QO and STATUS

Data Inputs

- **QI** (BOOL): Qualifier for initialization
- **ID** (WSTRING): Identifier for communication
- **SD_1** to **SD_8** (ANY): Eight different send data of any type

Data Outputs

- **QO** (BOOL): Qualifier for output state
- **STATUS** (WSTRING): Status information

Adapter

No adapter interfaces are available.

Functionality

The PUBLISH_8 block enables data distribution via a publisher-subscriber model. Upon receiving a REQ event, all eight SD_x data values are sent to all subscribed SUBSCRIBE_8 blocks. The INIT initialization configures the block for communication, with the ID parameter serving to identify the publisher.

Technical Features

- Supports up to eight different data values of any type (ANY)
- Uses Wide-String (WSTRING) for ID and STATUS for international character support
- Implements unacknowledged communication (REQ without direct acknowledgment)
- Generic implementation based on GEN_PUBLISH

State Overview

The block has two main states:

1. **Not Initialized**: Block is operational but not configured for communication
2. **Initialized**: Block is configured and can distribute data to subscribers

Application Scenarios

- Distribution of process data to multiple display or control systems
- Implementation of data broadcasting in distributed systems
- Use in PLC systems for machine-to-machine communication
- Use in Industry 4.0 applications for flexible data distribution

⚖️ Comparison with Similar Blocks

Compared to simpler PUBLISH blocks, PUBLISH_8 offers the ability to process up to eight different data values in parallel. Distributing data allows for a higher data density per communication cycle. The use of ANY data types offers maximum flexibility in the types of data to be transmitted.

Conclusion

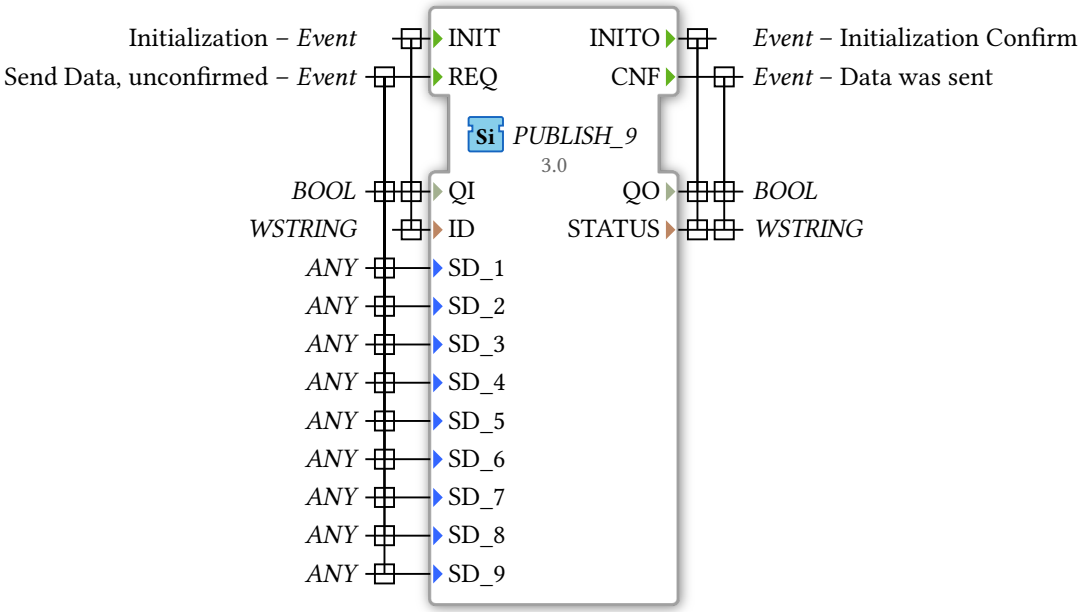
The PUBLISH_8 function block is a powerful solution for distributing multiple data values in distributed automation systems. Its flexibility in data types and support for eight parallel data channels make it ideal for complex communication scenarios in industrial applications.

2.11.6.46 PUBLISH_9

• – ◦ • – ◦ • – ◦ •

Introduction

The PUBLISH_9 function block is used to publish data to one or more SUBSCRIBE_9 blocks. It enables the unacknowledged transmission of up to nine different data values over a communication network.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **REQ**: Send request for unacknowledged data transmission with nine data values

Event Outputs

- **INITO**: Initialization acknowledgment with QO and STATUS
- **CNF**: Acknowledgement that data has been sent, with QO and STATUS

Data Inputs

- **QI** (BOOL): Qualifier for initialization (TRUE = enable, FALSE = disable)
- **ID** (WSTRING): Identifier for the communication channel
- **SD_1** to **SD_9** (ANY): Nine different data values of any type to be sent

Data Outputs

- **QO** (BOOL): Qualifier for output state
- **STATUS** (WSTRING): Status information about the operating state

Adapter

No adapter interfaces are available.

Functionality

The PUBLISH_9 block is first initialized via the INIT event, where the ID parameter defines the communication channel. With the QI input enabled (TRUE), data can be sent via the REQ event. The block transmits all nine SD_x data values simultaneously to all connected SUBSCRIBE_9 blocks. The transmission is unacknowledged, but the CNF event signals successful processing of the send request.

Technical Features

- Supports up to nine different data values of any type (ANY)
- Unacknowledged communication (fire-and-forget principle)
- WSTRING-based channel identification for flexible addressing
- Generic implementation via the GEN_PUBLISH base class

State Overview

The block has two main states: initialized and uninitialized. After successful initialization (INIT with QI=TRUE), the block switches to the active state and can send data. Upon deactivation (INIT with QI=FALSE), the block is deinitialized.

Application Scenarios

- Distributed systems with publisher-subscriber architecture
- Data exchange between different control components
- Broadcast communication in automation networks
- Systems with one-to-many communication relationships

⚖️ Comparison with similar blocks

Compared to acknowledged communication blocks, PUBLISH_9 offers reduced latency by eliminating the need for acknowledgments. Compared to blocks with fewer data channels, it enables the simultaneous transmission of multiple data values, thus increasing system efficiency.

Conclusion

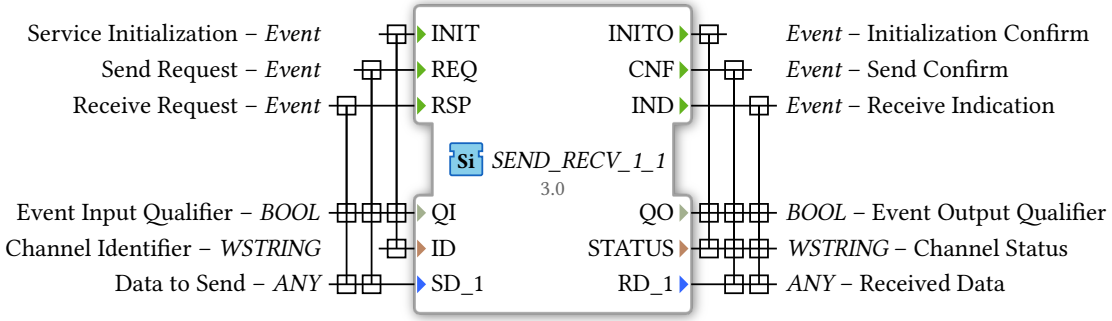
The PUBLISH_9 function block is a powerful solution for unacknowledged multiple data transmissions in distributed automation systems. Its data transmission flexibility and support for nine different data values make it ideal for complex communication scenarios in industrial control systems.

2.11.6.47 SEND_RECV_1_1

• – ◦ • – ◦ • – ◦ •

Introduction

The SEND_RECV_1_1 function block implements a duplex communication interface for bidirectional data exchange over communication channels. The block enables both sending and receiving data over a shared channel and supports generic data types for maximum flexibility.



Interface Structure

Event Inputs
- **INIT**: Service initialization with associated data QI and ID
- **REQ**: Send request with associated data QI and SD_1
- **RSP**: Receive request with associated data QI and SD_1

Event Outputs
- **INITO**: Initialization acknowledgment with QO and STATUS
- **CNF**: Send acknowledgment with QO, STATUS, and RD_1
- **IND**: Receive indication with QO, STATUS, and RD_1

Data Inputs
- **QI** (BOOL): Event input qualifier
- **ID** (WSTRING): Channel identifier
- **SD_1** (ANY): Data to be sent (generic) Data Type)

Data Outputs
- **QO** (BOOL): Event Output Qualifier
- **STATUS** (WSTRING): Channel Status Information
- **RD_1** (ANY): Received Data (generic data type)

Adapter
No adapter interfaces are available.

Functionality
The function block operates as a bidirectional communication interface, initialized via the INIT event input. After successful initialization, data can be sent via REQ and received via RSP. Communication occurs via a channel identified by its ID. The generic ANY data type allows the transmission of any data structure.

Technical Features
- Support for generic data types (ANY) for maximum flexibility
- Bidirectional communication over a shared channel
- Unicode string support for channel IDs and status messages
- Separate qualifiers for input and output events
- Status feedback via WSTRING for detailed error information

State Overview
The function block goes through the following states:

1. **Not Initialized**: Before INIT call
2. **Initialized**: After successful INIT/INITO sequence
3. **Send Operation**: During REQ/CNF operation
4. **Receive Operation**: During RSP/IND operation

Application Scenarios
- Client-server communication in distributed systems
- Bidirectional data transmission between control components
- Protocol implementations with request-response patterns
- Network communication in industrial automation systems

⚖️ Comparison with Similar Blocks
Compared to simple SEND or SEND_RECV_1_1 offers RECV blocks the combined functionality of both operations in a single block. Its duplex capability distinguishes it from simplex communication blocks and enables more efficient communication patterns.

Conclusion

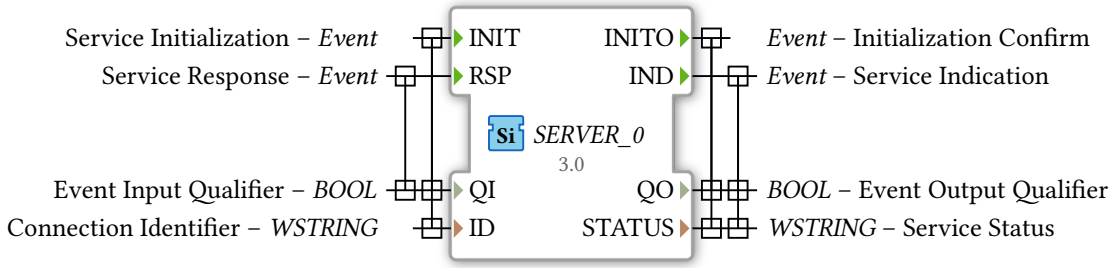
The SEND_RECV_1_1 function block provides a versatile solution for bidirectional communication requirements in distributed automation systems. By supporting generic data types and offering combined send/receive functionality, it provides an efficient foundation for various communication scenarios.

2.11.6.48 SERVER_0

• – ◦ • – ◦ • – ◦ •

Introduction

The SERVER_0 function block serves as the counterpart to a CLIENT_0 block. It enables the establishment of a server-client connection without exchanging user data via data inputs/outputs.



Interface Structure

Event Inputs

- **INIT** (Type: Event) - Enables the server for a client connection (QI = TRUE) or shuts down the server (QI = FALSE)
- **RSP** (Type: Event) - Sends a response/acknowledgment to the client

Event Outputs

- **INITO** (Type: Event) - Signals readiness for a client connection (QI = TRUE) or a closed state (QI = FALSE)
- **IND** (Type: Event) - Indicates that a request has been received from the client

Data Inputs

- **QI** (BOOL) - Qualifier for the INIT operation
- **ID** (WSTRING) - Identifier for the server instance

Data Outputs

- **QO** (BOOL) - Qualifier for status output
- **STATUS** (WSTRING) - Status Information

Functionality

The SERVER_0 block manages the connection.

1. **Initialization**: Activation via INIT.
2. **Synchronization**:

- When the client sends a request (REQ), the IND event is triggered on the server.
- The server can then respond with RSP, which triggers the CNF event on the client.

No user data (SD_x, RD_x) is transmitted during this process.

Technical Features

- No data inputs/outputs available
- WSTRING type for ID and STATUS
- Generic implementation via the 'GEN_SERVER' class attribute

State Overview

- **Not Initialized**: Server inactive

- **Initialized**: Server ready
- **Indication (IND)**: Client has requested
- **Response (RSP)**: Server responding

Application Scenarios

- Event synchronization
- Signaling of states without data content

Metadata

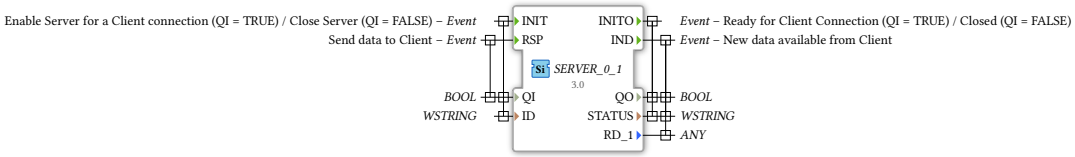
Attribute	Value
Copyright	(c) 2025 Monika Wenger
License	EPL-2.0
Version	3.0 (2025-12-05, Monika Wenger)
4diac package	iec61499::net

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.6.49 SERVER_0_1



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_0_1` function block is the generic server variant with 0 send and 1 receive data fields for communicating with a matching CLIENT_1_0 block. It receives 1 data values (`RD_1`) from the client without sending any payload itself. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`SERVER_0_1` transmits no payload data with `RSP`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.

Functionality

SERVER_0_1 initializes a connection to the matching CLIENT_1_0 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, a request without payload is sent. Once response data (RD_1) is available, the block triggers IND and provides RD_1.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_1_0**: SERVER_0_1 is only functionally compatible with a CLIENT_1_0 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 1 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_1_0 with the matching field count.

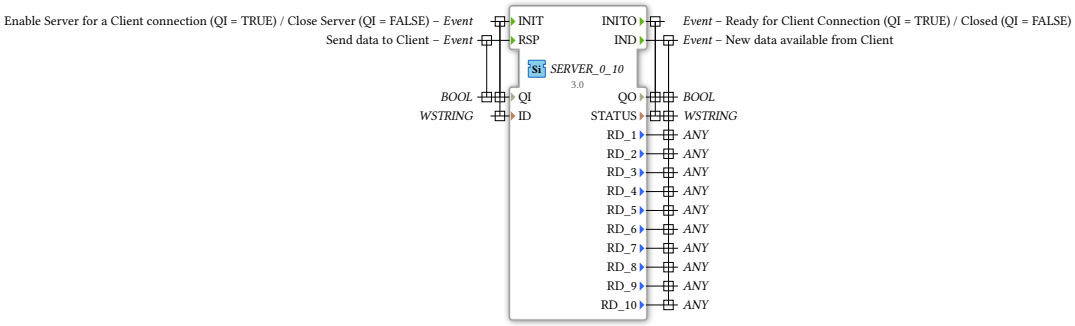
Comparison with similar function blocks

- CLIENT_1_0: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_0_1 provides the generic server variant of the GEN_SERVER family tailored to 0 send and 1 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.50 SERVER_0_10



• – ○ • – ○ • – ○ •

Introduction

The SERVER_0_10 function block is the generic server variant with 0 send and 10 receive data fields for communicating with a matching CLIENT_10_0 block. It receives 10 data values (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10) from the client without sending any payload itself. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`SERVER_0_10` transmits no payload data with `RSP`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.
- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.
- **RD_4** (ANY): Receive value 4, delivered by the client with `IND`.
- **RD_5** (ANY): Receive value 5, delivered by the client with `IND`.
- **RD_6** (ANY): Receive value 6, delivered by the client with `IND`.
- **RD_7** (ANY): Receive value 7, delivered by the client with `IND`.
- **RD_8** (ANY): Receive value 8, delivered by the client with `IND`.
- **RD_9** (ANY): Receive value 9, delivered by the client with `IND`.
- **RD_10** (ANY): Receive value 10, delivered by the client with `IND`.

Functionality

`SERVER_0_10` initializes a connection to the matching `CLIENT_10_0` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, a request without payload is sent. Once response data (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`) is available, the block triggers `IND` and provides `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_10_0`**: `SERVER_0_10` is only functionally compatible with a `CLIENT_10_0` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 10 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_10_0` with the matching field count.

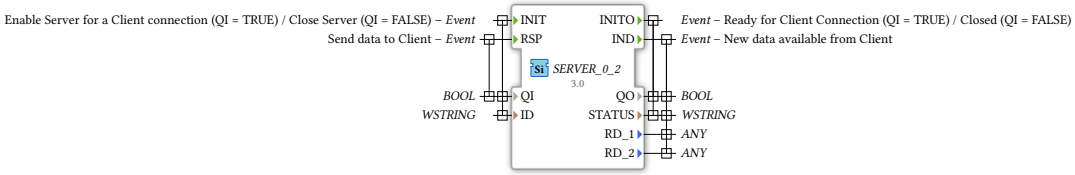
Comparison with similar function blocks

- **CLIENT_10_0**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_0_10 provides the generic server variant of the GEN_SERVER family tailored to 0 send and 10 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.51 SERVER_0_2



• – ○ • – ○ • – ○ •

Introduction

The SERVER_0_2 function block is the generic server variant with 0 send and 2 receive data fields for communicating with a matching CLIENT_2_0 block. It receives 2 data values (RD_1 RD_2) from the client without sending any payload itself. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries QI and ID.
- **RSP**: Sends data to the client, carries QI.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **IND**: Signals that new data is available from the client, carries QO, STATUS and RD_1 RD_2.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (SERVER_0_2 transmits no payload data with RSP, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with IND.
- **RD_2** (ANY): Receive value 2, delivered by the client with IND.

Functionality

SERVER_0_2 initializes a connection to the matching CLIENT_2_0 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, a request without payload is sent. Once response data (RD_1 RD_2) is available, the block triggers IND and provides RD_1 RD_2.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_2_0**: SERVER_0_2 is only functionally compatible with a CLIENT_2_0 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 2 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_2_0` with the matching field count.

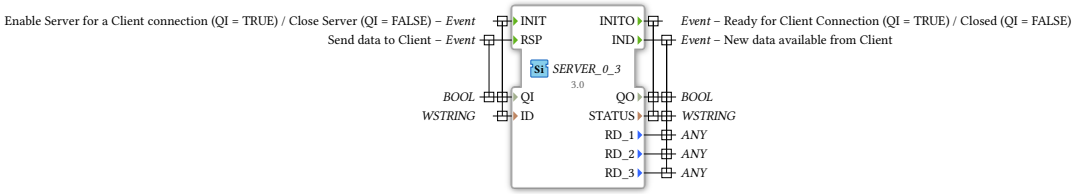
Comparison with similar function blocks

- **CLIENT_2_0**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_0_2` provides the generic server variant of the `GEN_SERVER` family tailored to 0 send and 2 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.52 SERVER_0_3



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_0_3` function block is the generic server variant with 0 send and 3 receive data fields for communicating with a matching CLIENT_3_0 block. It receives 3 data values (`RD_1 RD_2 RD_3`) from the client without sending any payload itself. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1 RD_2 RD_3`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`SERVER_0_3` transmits no payload data with `RSP`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.

- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.

Functionality

`SERVER_0_3` initializes a connection to the matching `CLIENT_3_0` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, a request without payload is sent. Once response data (`RD_1 RD_2 RD_3`) is available, the block triggers `IND` and provides `RD_1 RD_2 RD_3`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_3_0`**: `SERVER_0_3` is only functionally compatible with a `CLIENT_3_0` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 3 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_3_0` with the matching field count.

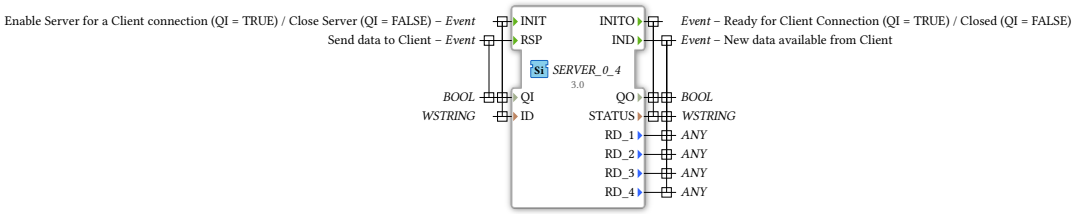
Comparison with similar function blocks

- CLIENT_3_0: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_0_3` provides the generic server variant of the `GEN_SERVER` family tailored to 0 send and 3 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.53 SERVER_0_4



• – ◦ • – ◦ • – ◦ •

Introduction

The `SERVER_0_4` function block is the generic server variant with 0 send and 4 receive data fields for communicating with a matching CLIENT_4_0 block. It receives 4 data values (`RD_1 RD_2 RD_3 RD_4`) from the client without sending any payload itself. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.

- **RSP**: Sends data to the client, carries QI.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **IND**: Signals that new data is available from the client, carries QO, STATUS and RD_1 RD_2 RD_3 RD_4.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (SERVER_0_4 transmits no payload data with RSP, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with IND.
- **RD_2** (ANY): Receive value 2, delivered by the client with IND.
- **RD_3** (ANY): Receive value 3, delivered by the client with IND.
- **RD_4** (ANY): Receive value 4, delivered by the client with IND.

Functionality

SERVER_0_4 initializes a connection to the matching CLIENT_4_0 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, a request without payload is sent. Once response data (RD_1 RD_2 RD_3 RD_4) is available, the block triggers IND and provides RD_1 RD_2 RD_3 RD_4.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_4_0**: SERVER_0_4 is only functionally compatible with a CLIENT_4_0 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 4 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_4_0 with the matching field count.

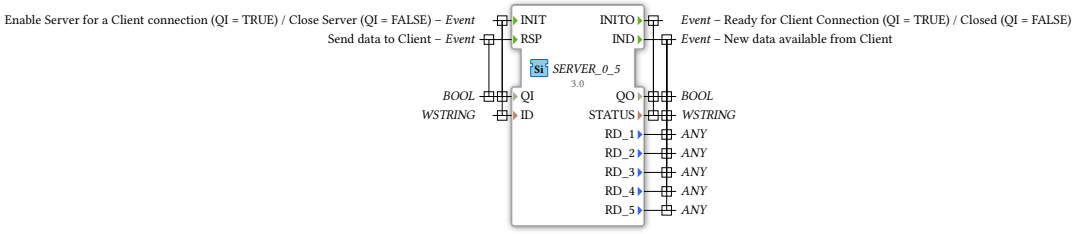
Comparison with similar function blocks

- CLIENT_4_0: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_0_4 provides the generic server variant of the GEN_SERVER family tailored to 0 send and 4 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.54 SERVER_0_5



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_0_5` function block is the generic server variant with 0 send and 5 receive data fields for communicating with a matching CLIENT_5_0 block. It receives 5 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5`) from the client without sending any payload itself. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4` `RD_5`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`SERVER_0_5` transmits no payload data with `RSP`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.
- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.
- **RD_4** (ANY): Receive value 4, delivered by the client with `IND`.
- **RD_5** (ANY): Receive value 5, delivered by the client with `IND`.

Functionality

`SERVER_0_5` initializes a connection to the matching `CLIENT_5_0` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, a request without payload is sent. Once response data (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5`) is available, the block triggers `IND` and provides `RD_1` `RD_2` `RD_3` `RD_4` `RD_5`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_5_0`**: `SERVER_0_5` is only functionally compatible with a `CLIENT_5_0` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 5 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_5_0` with the matching field count.

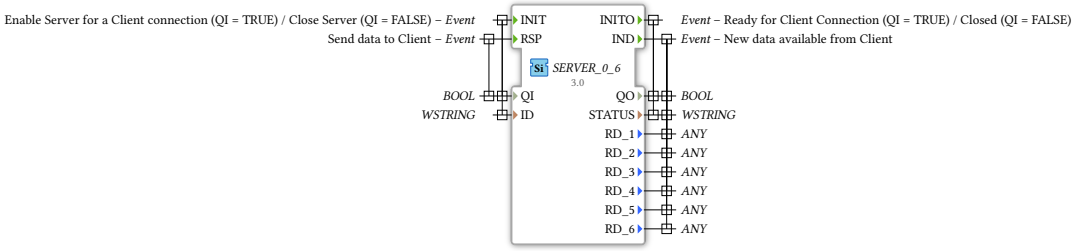
Comparison with similar function blocks

- **CLIENT_5_0**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_0_5` provides the generic server variant of the `GEN_SERVER` family tailored to 0 send and 5 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.55 SERVER_0_6



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_0_6` function block is the generic server variant with 0 send and 6 receive data fields for communicating with a matching CLIENT_6_0 block. It receives 6 data values (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6`) from the client without sending any payload itself. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`SERVER_0_6` transmits no payload data with `RSP`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.

- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.
- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.
- **RD_4** (ANY): Receive value 4, delivered by the client with `IND`.
- **RD_5** (ANY): Receive value 5, delivered by the client with `IND`.
- **RD_6** (ANY): Receive value 6, delivered by the client with `IND`.

Functionality

`SERVER_0_6` initializes a connection to the matching `CLIENT_6_0` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, a request without payload is sent. Once response data (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6`) is available, the block triggers `IND` and provides `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_6_0`**: `SERVER_0_6` is only functionally compatible with a `CLIENT_6_0` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 6 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_6_0` with the matching field count.

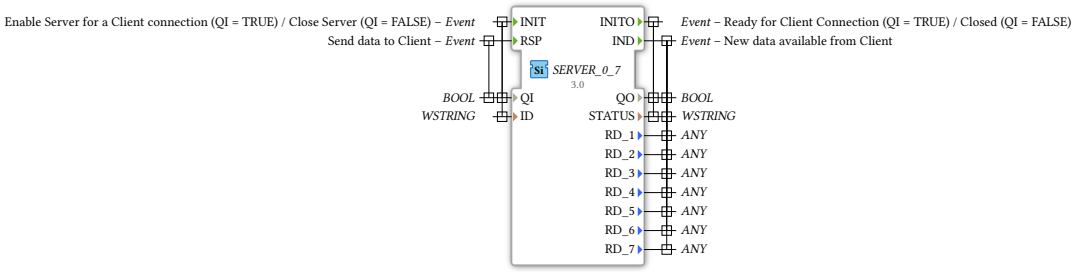
Comparison with similar function blocks

- **CLIENT_6_0**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_0_6` provides the generic server variant of the `GEN_SERVER` family tailored to 0 send and 6 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.56 SERVER_0_7



• – ○ • – ○ • – ○ •

Introduction

The SERVER_0_7 function block is the generic server variant with 0 send and 7 receive data fields for communicating with a matching CLIENT_7_0 block. It receives 7 data values (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7) from the client without sending any payload itself. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries QI and ID.
- **RSP**: Sends data to the client, carries QI.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **IND**: Signals that new data is available from the client, carries QO, STATUS and RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (SERVER_0_7 transmits no payload data with RSP, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with IND.
- **RD_2** (ANY): Receive value 2, delivered by the client with IND.
- **RD_3** (ANY): Receive value 3, delivered by the client with IND.
- **RD_4** (ANY): Receive value 4, delivered by the client with IND.
- **RD_5** (ANY): Receive value 5, delivered by the client with IND.
- **RD_6** (ANY): Receive value 6, delivered by the client with IND.
- **RD_7** (ANY): Receive value 7, delivered by the client with IND.

Functionality

SERVER_0_7 initializes a connection to the matching CLIENT_7_0 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, a request without payload is sent. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7) is available, the block triggers IND and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_7_0**: SERVER_0_7 is only functionally compatible with a CLIENT_7_0 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 7 values, without carrying unused surplus data fields.

- **Server-side connection** to a communication partner already laid out as a `CLIENT_7_0` with the matching field count.

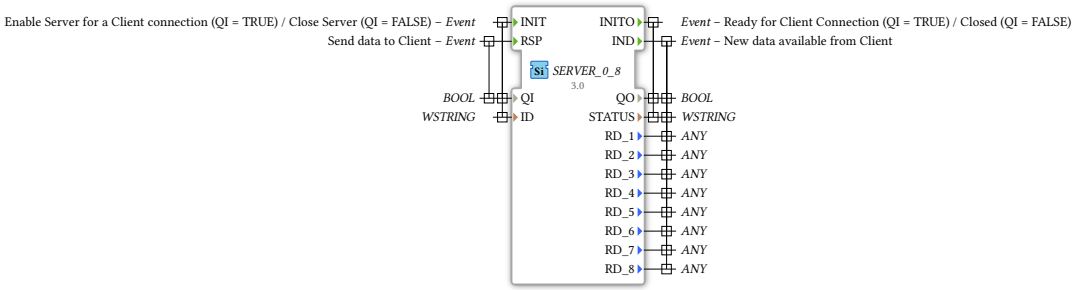
Comparison with similar function blocks

- **CLIENT_7_0**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_0_7` provides the generic server variant of the `GEN_SERVER` family tailored to 0 send and 7 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.57 SERVER_0_8



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_0_8` function block is the generic server variant with 0 send and 8 receive data fields for communicating with a matching CLIENT_8_0 block. It receives 8 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8`) from the client without sending any payload itself. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`SERVER_0_8` transmits no payload data with `RSP`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.
- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.
- **RD_4** (ANY): Receive value 4, delivered by the client with `IND`.
- **RD_5** (ANY): Receive value 5, delivered by the client with `IND`.
- **RD_6** (ANY): Receive value 6, delivered by the client with `IND`.
- **RD_7** (ANY): Receive value 7, delivered by the client with `IND`.
- **RD_8** (ANY): Receive value 8, delivered by the client with `IND`.

Functionality

SERVER_0_8 initializes a connection to the matching CLIENT_8_0 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, a request without payload is sent. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8) is available, the block triggers IND and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_8_0**: SERVER_0_8 is only functionally compatible with a CLIENT_8_0 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 8 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_8_0 with the matching field count.

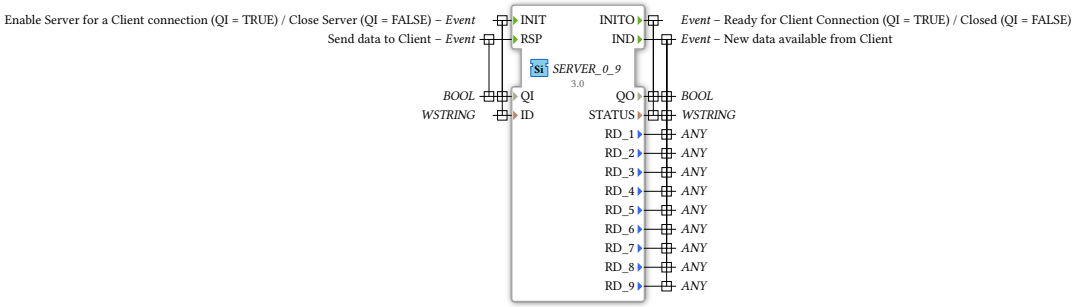
Comparison with similar function blocks

- CLIENT_8_0: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_0_8 provides the generic server variant of the GEN_SERVER family tailored to 0 send and 8 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.58 SERVER_0_9



• – ◦ • – ◦ • – ◦ •

Introduction

The SERVER_0_9 function block is the generic server variant with 0 send and 9 receive data fields for communicating with a matching CLIENT_9_0 block. It receives 9 data values (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9) from the client without sending any payload itself. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- No send fields (`SERVER_0_9` transmits no payload data with `RSP`, only connection control).

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.
- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.
- **RD_4** (ANY): Receive value 4, delivered by the client with `IND`.
- **RD_5** (ANY): Receive value 5, delivered by the client with `IND`.
- **RD_6** (ANY): Receive value 6, delivered by the client with `IND`.
- **RD_7** (ANY): Receive value 7, delivered by the client with `IND`.
- **RD_8** (ANY): Receive value 8, delivered by the client with `IND`.
- **RD_9** (ANY): Receive value 9, delivered by the client with `IND`.

Functionality

`SERVER_0_9` initializes a connection to the matching `CLIENT_9_0` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, a request without payload is sent. Once response data (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9`) is available, the block triggers `IND` and provides `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_9_0`**: `SERVER_0_9` is only functionally compatible with a `CLIENT_9_0` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 0 values and receive exactly 9 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_9_0` with the matching field count.

Comparison with similar function blocks

- **CLIENT_9_0**: the direct counterpart on the client side, with send/receive field counts swapped.

- **CLIENT_1 / SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

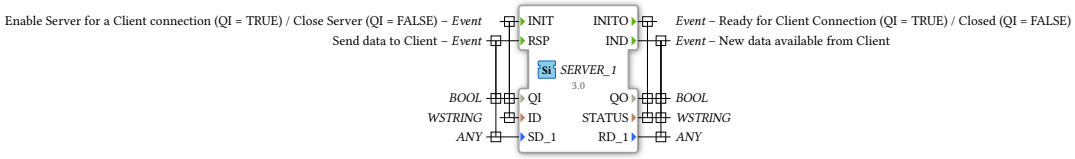
`SERVER_0_9` provides the generic server variant of the `GEN_SERVER` family tailored to 0 send and 9 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.59 SERVER_1

• – ○ • – ○ • – ○ •

Introduction

The SERVER_1 function block communicates with a CLIENT_1 block and enables the establishment of a server-client connection. The block can receive data from and send data to clients, providing a flexible communication interface for various use cases.



Interface Structure

Event Inputs

- **INIT** (Type: EInit) - Enables the server for a client connection (QI = TRUE) or shuts down the server (QI = FALSE)
- **RSP** (Type: Event) - Sends data to the client

Event Outputs

- **INITO** (Type: EInit) - Signals readiness for a client connection (QI = TRUE) or a closed state (QI = FALSE)
- **IND** (Type: Event) - Displays newly available data from the client

Data Inputs

- **QI** (BOOL) - Qualifier for the INIT operation
- **ID** (WSTRING) - Identifier for the server instance
- **SD_1** (ANY) - Sends data to the client

Data Outputs

- **QO** (BOOL) - Qualifier for status output
- **STATUS** (WSTRING) - Status information
- **RD_1** (ANY) - Data received from the client

Functionality

The SERVER_1 block manages server-client connections through two main operations:

1. **Initialization/Deactivation**: The INIT event with the corresponding QI value activates or deactivates the server. Upon successful operation, INITO is triggered with status information.
2. **Data Communication**:

- Receiving: Incoming data from the client triggers the IND event and is available via RD_1.
- Sending: The RSP event enables sending data via SD_1 to the connected client.

Technical Features

- Using the ANY data type for SD_1 and RD_1 allows for flexible data formats.
- The WSTRING type for ID and STATUS supports Unicode strings.
- Generic implementation via the 'GEN_SERVER' class attribute.
- Package membership: iec61499::net

State Overview

The block can assume the following states:

- **Not Initialized**: Server not operational.

- **Initialized**: Server ready for client connections.
- **Data Receiving**: Processing incoming data.
- **Data Sending**: Sending data to the client.

Application Scenarios

- Industrial Automation Systems
- Networked Control Systems
- IoT Communication Applications
- Distributed Control Systems (DCS)
- Machine-to-Machine Communication

⚖️ Comparison with Similar Blocks

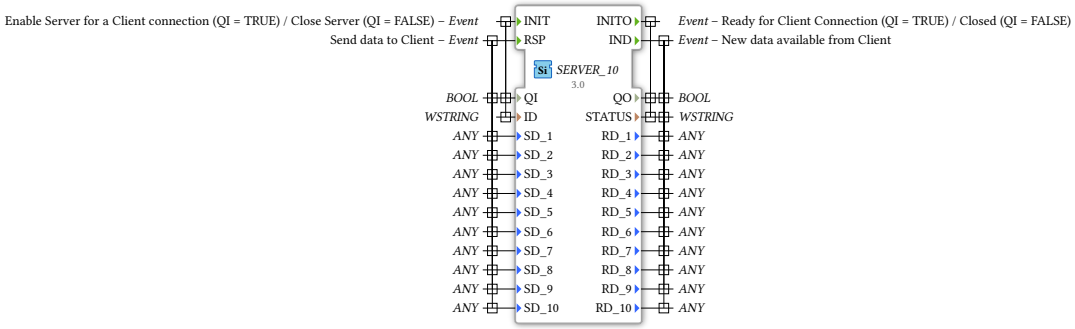
Compared to other communication blocks, SERVER_1 offers:

- Specific server functionality for CLIENT_1 blocks
- Flexible data transmission via ANY type
- Explicit initialization and deactivation control
- Status feedback for improved error detection

Conclusion

The SERVER_1 function block represents a robust and flexible solution for server-client communication in distributed automation systems. Its clear interface definition and flexible data type support make it particularly suitable for a wide variety of industrial communication requirements.

2.11.6.60 SERVER_10



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_10` function block is the generic server variant with 10 send and 10 receive data fields for communicating with a matching CLIENT_10 block. It transmits 10 data values (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`) to the client and receives 10 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`) back. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` `SD_10`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9` `RD_10`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.
- **SD_3** (ANY): Send value 3, transmitted to the client with `RSP`.
- **SD_4** (ANY): Send value 4, transmitted to the client with `RSP`.
- **SD_5** (ANY): Send value 5, transmitted to the client with `RSP`.
- **SD_6** (ANY): Send value 6, transmitted to the client with `RSP`.
- **SD_7** (ANY): Send value 7, transmitted to the client with `RSP`.
- **SD_8** (ANY): Send value 8, transmitted to the client with `RSP`.
- **SD_9** (ANY): Send value 9, transmitted to the client with `RSP`.
- **SD_10** (ANY): Send value 10, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.
- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.
- **RD_4** (ANY): Receive value 4, delivered by the client with `IND`.
- **RD_5** (ANY): Receive value 5, delivered by the client with `IND`.
- **RD_6** (ANY): Receive value 6, delivered by the client with `IND`.
- **RD_7** (ANY): Receive value 7, delivered by the client with `IND`.
- **RD_8** (ANY): Receive value 8, delivered by the client with `IND`.
- **RD_9** (ANY): Receive value 9, delivered by the client with `IND`.
- **RD_10** (ANY): Receive value 10, delivered by the client with `IND`.

Functionality

`SERVER_10` initializes a connection to the matching `CLIENT_10` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, the values `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10` are transmitted to the client. Once response data (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10`) is available, the block triggers `IND` and provides `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9 RD_10`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_10`**: `SERVER_10` is only functionally compatible with a `CLIENT_10` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 10 values and receive exactly 10 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_10` with the matching field count.

Comparison with similar function blocks

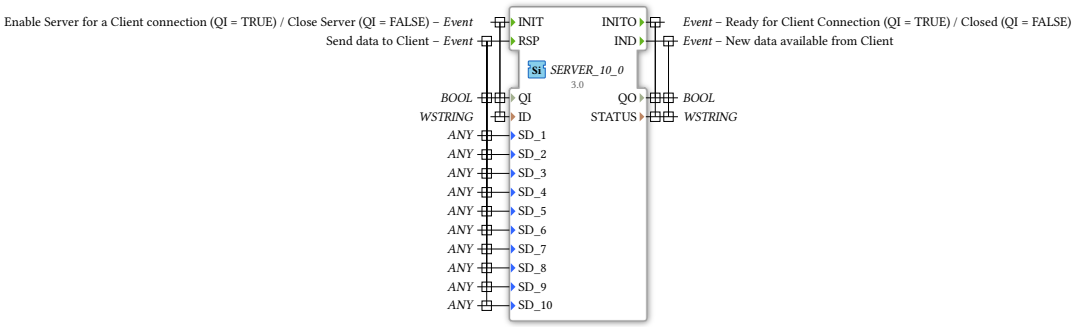
- **CLIENT_10**: the direct counterpart on the client side, with send/receive field counts swapped.

- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_10 provides the generic server variant of the GEN_SERVER family tailored to 10 send and 10 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.61 SERVER_10_0



• – ○ • – ○ • – ○ •

Introduction

The SERVER_10_0 function block is the generic server variant with 10 send and 0 receive data fields for communicating with a matching CLIENT_0_10 block. It transmits 10 data values (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10) to the client without receiving any data back. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries QI and ID.
- **RSP**: Sends data to the client, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **IND**: Signals that new data is available from the client, carries QO, STATUS.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with RSP.
- **SD_2** (ANY): Send value 2, transmitted to the client with RSP.
- **SD_3** (ANY): Send value 3, transmitted to the client with RSP.
- **SD_4** (ANY): Send value 4, transmitted to the client with RSP.
- **SD_5** (ANY): Send value 5, transmitted to the client with RSP.
- **SD_6** (ANY): Send value 6, transmitted to the client with RSP.
- **SD_7** (ANY): Send value 7, transmitted to the client with RSP.
- **SD_8** (ANY): Send value 8, transmitted to the client with RSP.
- **SD_9** (ANY): Send value 9, transmitted to the client with RSP.
- **SD_10** (ANY): Send value 10, transmitted to the client with RSP.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (SERVER_10_0 receives no payload data via IND, only connection/status information).

Functionality

SERVER_10_0 initializes a connection to the matching CLIENT_0_10 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 SD_10 are transmitted to the client. Once a response is available, the block triggers IND.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_0_10**: SERVER_10_0 is only functionally compatible with a CLIENT_0_10 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 10 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_0_10 with the matching field count.

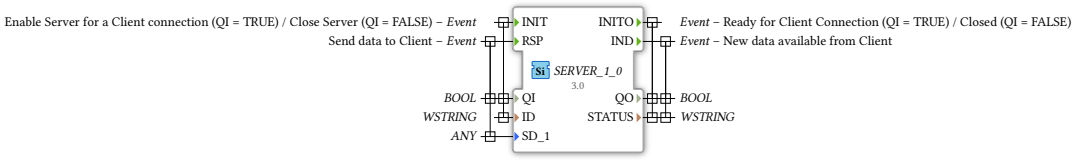
Comparison with similar function blocks

- CLIENT_0_10: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_10_0 provides the generic server variant of the GEN_SERVER family tailored to 10 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.62 SERVER_1_0



• – ○ • – ○ • – ○ •

Introduction

The SERVER_1_0 function block is the generic server variant with 1 send and 0 receive data fields for communicating with a matching CLIENT_0_1 block. It transmits 1 data values (SD_1) to the client without receiving any data back. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries QI and ID.
- **RSP**: Sends data to the client, carries QI and SD_1.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`SERVER_1_0` receives no payload data via `IND`, only connection/status information).

Functionality

`SERVER_1_0` initializes a connection to the matching `CLIENT_0_1` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, the values `SD_1` are transmitted to the client. Once a response is available, the block triggers `IND`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_0_1`**: `SERVER_1_0` is only functionally compatible with a `CLIENT_0_1` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 1 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_0_1` with the matching field count.

Comparison with similar function blocks

- **CLIENT_0_1**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

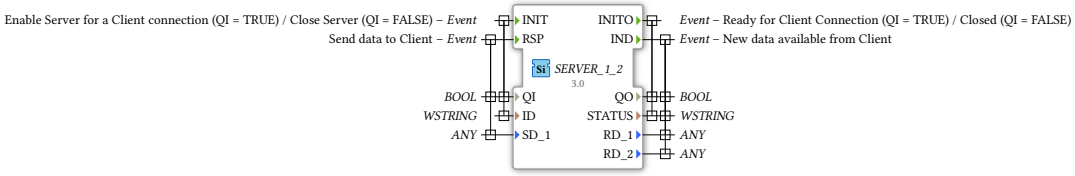
`SERVER_1_0` provides the generic server variant of the `GEN_SERVER` family tailored to 1 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.63 SERVER_1_2

• – ◦ • – ◦ • – ◦ •

Introduction

The SERVER_1_2 function block communicates with a CLIENT_2_1 block and enables the creation of a server-client connection. The block can receive and send data, as well as initialize and terminate the connection.



Interface Structure

Event Inputs

- **INIT**: Initializes the server for a client connection (QI = TRUE) or shuts down the server (QI = FALSE)
- **RSP**: Sends data to the client

Event Outputs

- **INITO**: Signals readiness for a client connection (QI = TRUE) or a closed state (QI = FALSE)
- **IND**: Displays newly available data from the client

Data Inputs

- **QI** (BOOL): Qualifier for the INIT event (TRUE = enable, FALSE = disable)
- **ID** (WSTRING): Identification string for the server instance
- **SD_1** (ANY): Data to be sent to the client

Data Outputs

- **QO** (BOOL): Status output (TRUE = active, FALSE = inactive) (inactive)
- **STATUS** (WSTRING): Status information as a string
- **RD_1** (ANY): Data received from the client (Channel 1)
- **RD_2** (ANY): Data received from the client (Channel 2)

Functionality

The SERVER_1_2 block operates as a server in a client-server architecture. Upon initialization (INIT with QI=TRUE), the server is made available for incoming connections. When data is received from the client, the IND event is triggered, and the received data is made available via RD_1 and RD_2. Data can be sent back to the client via the RSP event.

Technical Features

- Supports the ANY data type for flexible data transmission
- Uses WSTRING for status messages and identification
- Provides two separate data channels for receiving (RD_1 and RD_2)
- Implemented as a generic server block

Status Overview

- **Inactive**: Server not initialized (QO = FALSE)
- **Active**: Server ready for client connections (QO = TRUE)
- **Data Receive**: New data available from the client (IND event)
- **Data Send**: Data is being sent to the client (RSP event)

Application Scenarios

- Network communication in distributed automation systems
- Data exchange between different control components
- Implementation of server services in IEC 61499-based systems
- Bidirectional communication in industrial networks

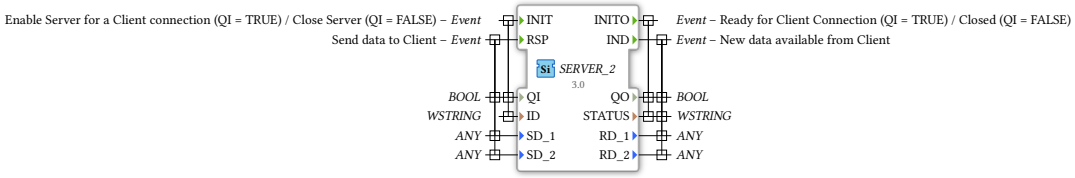
⚖️ Comparison with Similar Blocks

Compared to simpler communication blocks, SERVER_1_2 offers a complete server implementation with Initialization and status management. The two separate receive channels enable structured data processing.

Conclusion

The SERVER_1_2 function block represents a robust solution for server-based communication in IEC 61499 systems. Its flexible data type support and clear state management make it ideally suited for various industrial communication scenarios.

2.11.6.64 SERVER_2



• – ◦ • – ◦ • – ◦ •

Introduction

The `SERVER_2` function block is the generic server variant with 2 send and 2 receive data fields for communicating with a matching CLIENT_2 block. It transmits 2 data values (`SD_1` `SD_2`) to the client and receives 2 data values (`RD_1` `RD_2`) back. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1` `SD_2`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1` `RD_2`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.

Functionality

`SERVER_2` initializes a connection to the matching `CLIENT_2` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, the values `SD_1` `SD_2` are transmitted to the client. Once response data (`RD_1` `RD_2`) is available, the block triggers `IND` and provides `RD_1` `RD_2`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_2`**: `SERVER_2` is only functionally compatible with a `CLIENT_2` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 2 values and receive exactly 2 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_2` with the matching field count.

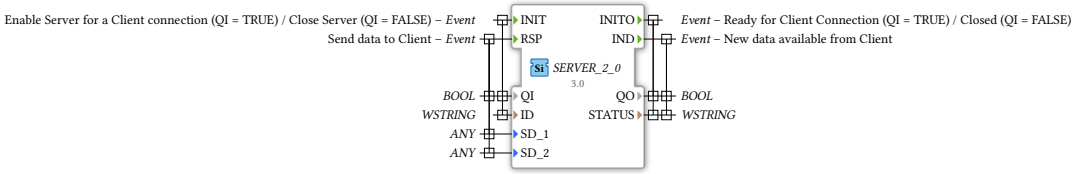
Comparison with similar function blocks

- CLIENT_2: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_2` provides the generic server variant of the `GEN_SERVER` family tailored to 2 send and 2 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.65 SERVER_2_0



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_2_0` function block is the generic server variant with 2 send and 0 receive data fields for communicating with a matching CLIENT_0_2 block. It transmits 2 data values (`SD_1` `SD_2`) to the client without receiving any data back. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1` `SD_2`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`SERVER_2_0` receives no payload data via `IND`, only connection/status information).

Functionality

SERVER_2_0 initializes a connection to the matching CLIENT_0_2 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, the values SD_1 SD_2 are transmitted to the client. Once a response is available, the block triggers IND.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_0_2**: SERVER_2_0 is only functionally compatible with a CLIENT_0_2 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 2 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_0_2 with the matching field count.

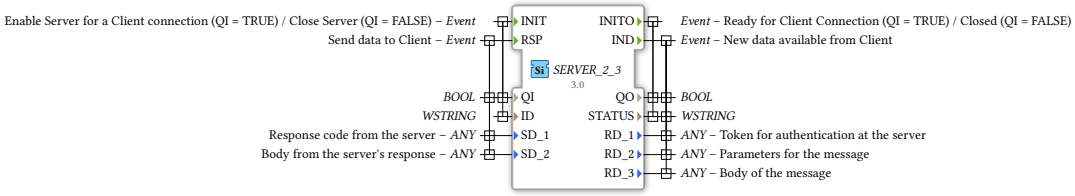
Comparison with similar function blocks

- CLIENT_0_2: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_2_0 provides the generic server variant of the GEN_SERVER family tailored to 2 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.66 SERVER_2_3



• – ○ • – ○ • – ○ •

Introduction

The SERVER_2_3 function block is the generic server variant with 2 send and 3 receive data fields for communicating with a matching CLIENT_3_2 block. It transmits 2 data values (SD_1 SD_2) to the client and receives 3 data values (RD_1 RD_2 RD_3) back. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries QI and ID.
- **RSP**: Sends data to the client, carries QI and SD_1 SD_2.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.
- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.

Functionality

`SERVER_2_3` initializes a connection to the matching `CLIENT_3_2` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, the values `SD_1` `SD_2` are transmitted to the client. Once response data (`RD_1` `RD_2` `RD_3`) is available, the block triggers `IND` and provides `RD_1` `RD_2` `RD_3`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_3_2`**: `SERVER_2_3` is only functionally compatible with a `CLIENT_3_2` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 2 values and receive exactly 3 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_3_2` with the matching field count.

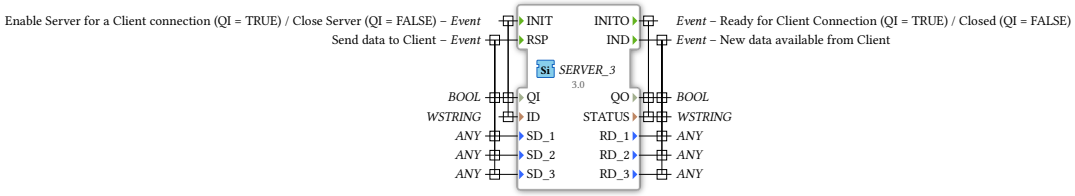
Comparison with similar function blocks

- CLIENT_3_2: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_2_3` provides the generic server variant of the `GEN_SERVER` family tailored to 2 send and 3 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.67 SERVER_3



• – ◦ • – ◦ • – ◦ •

Introduction

The SERVER_3 function block is the generic server variant with 3 send and 3 receive data fields for communicating with a matching CLIENT_3 block. It transmits 3 data values (SD_1 SD_2 SD_3) to the client and receives 3 data values (RD_1 RD_2 RD_3) back. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries QI and ID.
- **RSP**: Sends data to the client, carries QI and SD_1 SD_2 SD_3.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **IND**: Signals that new data is available from the client, carries QO, STATUS and RD_1 RD_2 RD_3.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with RSP.
- **SD_2** (ANY): Send value 2, transmitted to the client with RSP.
- **SD_3** (ANY): Send value 3, transmitted to the client with RSP.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with IND.
- **RD_2** (ANY): Receive value 2, delivered by the client with IND.
- **RD_3** (ANY): Receive value 3, delivered by the client with IND.

Functionality

SERVER_3 initializes a connection to the matching CLIENT_3 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, the values SD_1 SD_2 SD_3 are transmitted to the client. Once response data (RD_1 RD_2 RD_3) is available, the block triggers IND and provides RD_1 RD_2 RD_3.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_3**: SERVER_3 is only functionally compatible with a CLIENT_3 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.

2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 3 values and receive exactly 3 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_3` with the matching field count.

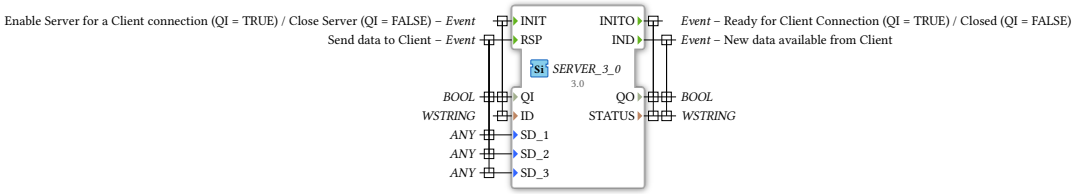
Comparison with similar function blocks

- CLIENT_3: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_3` provides the generic server variant of the `GEN_SERVER` family tailored to 3 send and 3 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.68 SERVER_3_0



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_3_0` function block is the generic server variant with 3 send and 0 receive data fields for communicating with a matching CLIENT_0_3 block. It transmits 3 data values (`SD_1 SD_2 SD_3`) to the client without receiving any data back. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1 SD_2 SD_3`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.
- **SD_3** (ANY): Send value 3, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`SERVER_3_0` receives no payload data via `IND`, only connection/status information).

Functionality

SERVER_3_0 initializes a connection to the matching CLIENT_0_3 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, the values SD_1 SD_2 SD_3 are transmitted to the client. Once a response is available, the block triggers IND.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_0_3**: SERVER_3_0 is only functionally compatible with a CLIENT_0_3 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 3 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_0_3 with the matching field count.

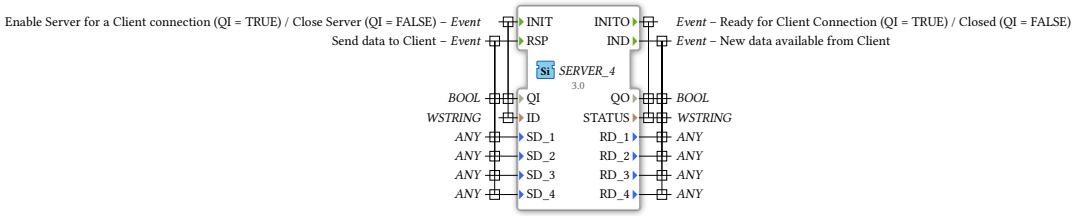
Comparison with similar function blocks

- CLIENT_0_3: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_3_0 provides the generic server variant of the GEN_SERVER family tailored to 3 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.69 SERVER_4



• – ○ • – ○ • – ○ •

Introduction

The SERVER_4 function block is the generic server variant with 4 send and 4 receive data fields for communicating with a matching CLIENT_4 block. It transmits 4 data values (SD_1 SD_2 SD_3 SD_4) to the client and receives 4 data values (RD_1 RD_2 RD_3 RD_4) back. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries QI and ID.
- **RSP**: Sends data to the client, carries QI and SD_1 SD_2 SD_3 SD_4.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.
- **SD_3** (ANY): Send value 3, transmitted to the client with `RSP`.
- **SD_4** (ANY): Send value 4, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.
- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.
- **RD_4** (ANY): Receive value 4, delivered by the client with `IND`.

Functionality

`SERVER_4` initializes a connection to the matching `CLIENT_4` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, the values `SD_1` `SD_2` `SD_3` `SD_4` are transmitted to the client. Once response data (`RD_1` `RD_2` `RD_3` `RD_4`) is available, the block triggers `IND` and provides `RD_1` `RD_2` `RD_3` `RD_4`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_4`**: `SERVER_4` is only functionally compatible with a `CLIENT_4` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 4 values and receive exactly 4 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_4` with the matching field count.

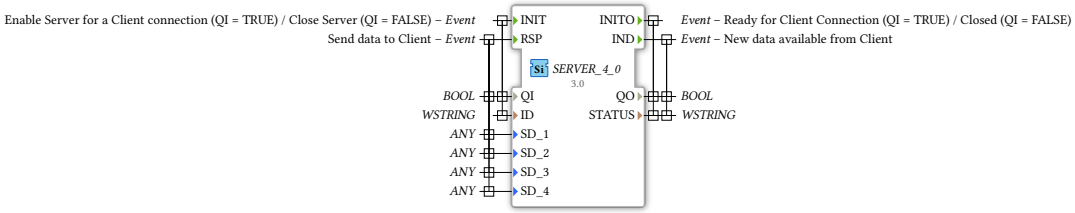
Comparison with similar function blocks

- `CLIENT_4`: the direct counterpart on the client side, with send/receive field counts swapped.
- `CLIENT_1` / `SERVER_1`: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_4` provides the generic server variant of the `GEN_SERVER` family tailored to 4 send and 4 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.70 SERVER_4_0



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_4_0` function block is the generic server variant with 4 send and 0 receive data fields for communicating with a matching CLIENT_0_4 block. It transmits 4 data values (`SD_1 SD_2 SD_3 SD_4`) to the client without receiving any data back. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1 SD_2 SD_3 SD_4`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.
- **SD_3** (ANY): Send value 3, transmitted to the client with `RSP`.
- **SD_4** (ANY): Send value 4, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`SERVER_4_0` receives no payload data via `IND`, only connection/status information).

Functionality

`SERVER_4_0` initializes a connection to the matching `CLIENT_0_4` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, the values `SD_1 SD_2 SD_3 SD_4` are transmitted to the client. Once a response is available, the block triggers `IND`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_0_4`**: `SERVER_4_0` is only functionally compatible with a `CLIENT_0_4` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.

4. **Data exchange**: `RSP/IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 4 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_0_4` with the matching field count.

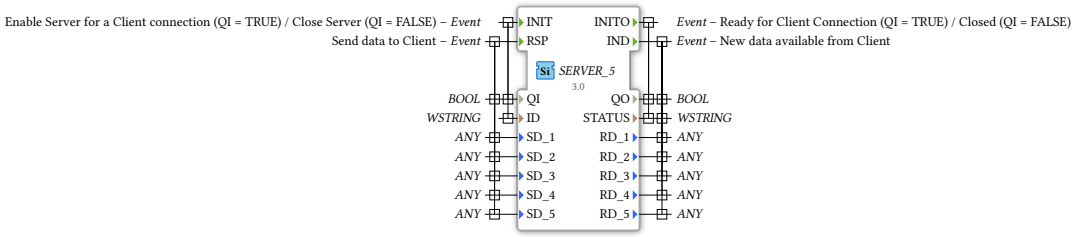
Comparison with similar function blocks

- **CLIENT_0_4**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_4_0` provides the generic server variant of the `GEN_SERVER` family tailored to 4 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.71 SERVER_5



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_5` function block is the generic server variant with 5 send and 5 receive data fields for communicating with a matching CLIENT_5 block. It transmits 5 data values (`SD_1 SD_2 SD_3 SD_4 SD_5`) to the client and receives 5 data values (`RD_1 RD_2 RD_3 RD_4 RD_5`) back. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1 SD_2 SD_3 SD_4 SD_5`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1 RD_2 RD_3 RD_4 RD_5`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.
- **SD_3** (ANY): Send value 3, transmitted to the client with `RSP`.
- **SD_4** (ANY): Send value 4, transmitted to the client with `RSP`.
- **SD_5** (ANY): Send value 5, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.

- **RD_2** (ANY): Receive value 2, delivered by the client with IND.
- **RD_3** (ANY): Receive value 3, delivered by the client with IND.
- **RD_4** (ANY): Receive value 4, delivered by the client with IND.
- **RD_5** (ANY): Receive value 5, delivered by the client with IND.

Functionality

SERVER_5 initializes a connection to the matching CLIENT_5 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, the values SD_1 SD_2 SD_3 SD_4 SD_5 are transmitted to the client. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5) is available, the block triggers IND and provides RD_1 RD_2 RD_3 RD_4 RD_5.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_5**: SERVER_5 is only functionally compatible with a CLIENT_5 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 5 values and receive exactly 5 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_5 with the matching field count.

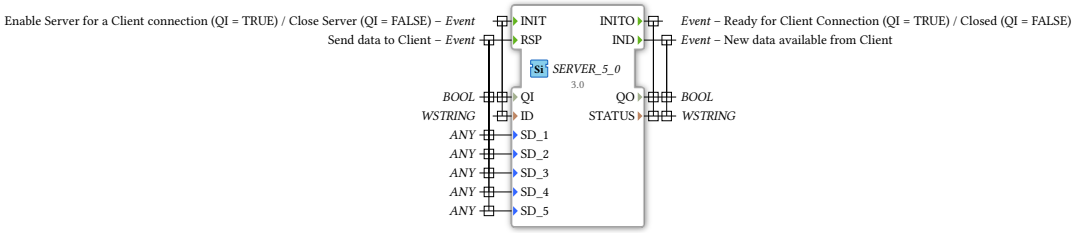
Comparison with similar function blocks

- CLIENT_5: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_5 provides the generic server variant of the GEN_SERVER family tailored to 5 send and 5 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.72 SERVER_5_0



• – ○ • – ○ • – ○ •

Introduction

The SERVER_5_0 function block is the generic server variant with 5 send and 0 receive data fields for communicating with a matching CLIENT_0_5 block. It transmits 5 data values (SD_1 SD_2 SD_3 SD_4 SD_5) to the client without receiving any data back. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4` `SD_5`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.
- **SD_3** (ANY): Send value 3, transmitted to the client with `RSP`.
- **SD_4** (ANY): Send value 4, transmitted to the client with `RSP`.
- **SD_5** (ANY): Send value 5, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`SERVER_5_0` receives no payload data via `IND`, only connection/status information).

Functionality

`SERVER_5_0` initializes a connection to the matching `CLIENT_0_5` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, the values `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` are transmitted to the client. Once a response is available, the block triggers `IND`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_0_5`**: `SERVER_5_0` is only functionally compatible with a `CLIENT_0_5` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 5 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_0_5` with the matching field count.

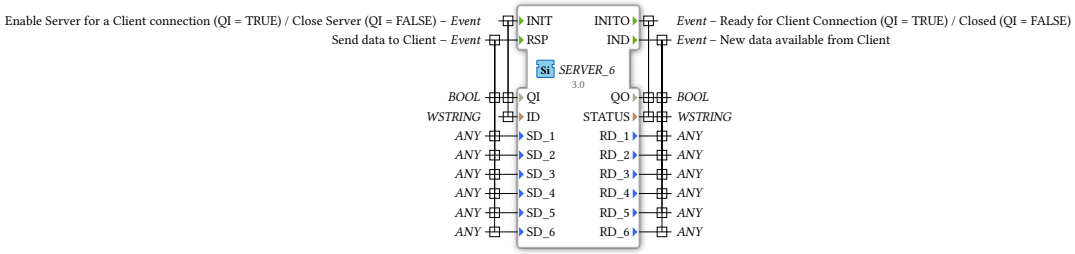
Comparison with similar function blocks

- **CLIENT_0_5**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_5_0` provides the generic server variant of the `GEN_SERVER` family tailored to 5 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.73 SERVER_6



• – ○ • – ○ • – ○ •

Introduction

The SERVER_6 function block is the generic server variant with 6 send and 6 receive data fields for communicating with a matching CLIENT_6 block. It transmits 6 data values (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6) to the client and receives 6 data values (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6) back. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries QI and ID.
- **RSP**: Sends data to the client, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5 SD_6.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **IND**: Signals that new data is available from the client, carries QO, STATUS and RD_1 RD_2 RD_3 RD_4 RD_5 RD_6.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with RSP.
- **SD_2** (ANY): Send value 2, transmitted to the client with RSP.
- **SD_3** (ANY): Send value 3, transmitted to the client with RSP.
- **SD_4** (ANY): Send value 4, transmitted to the client with RSP.
- **SD_5** (ANY): Send value 5, transmitted to the client with RSP.
- **SD_6** (ANY): Send value 6, transmitted to the client with RSP.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with IND.
- **RD_2** (ANY): Receive value 2, delivered by the client with IND.
- **RD_3** (ANY): Receive value 3, delivered by the client with IND.
- **RD_4** (ANY): Receive value 4, delivered by the client with IND.
- **RD_5** (ANY): Receive value 5, delivered by the client with IND.
- **RD_6** (ANY): Receive value 6, delivered by the client with IND.

Functionality

SERVER_6 initializes a connection to the matching CLIENT_6 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 are transmitted to the client. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6) is available, the block triggers IND and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_6`**: `SERVER_6` is only functionally compatible with a `CLIENT_6` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 6 values and receive exactly 6 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_6` with the matching field count.

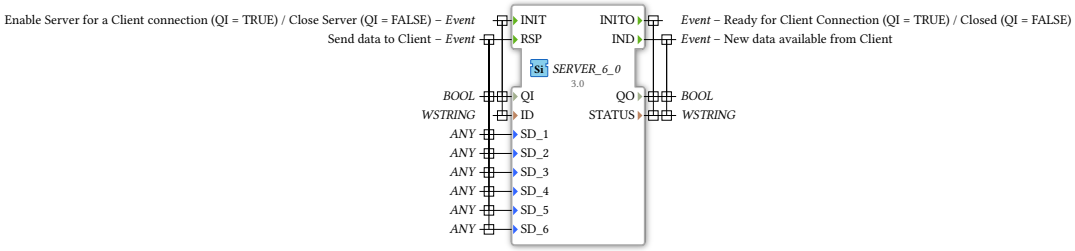
Comparison with similar function blocks

- CLIENT_6: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_6` provides the generic server variant of the `GEN_SERVER` family tailored to 6 send and 6 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.74 SERVER_6_0



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_6_0` function block is the generic server variant with 6 send and 0 receive data fields for communicating with a matching CLIENT_0_6 block. It transmits 6 data values (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6`) to the client without receiving any data back. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with RSP.
- **SD_2** (ANY): Send value 2, transmitted to the client with RSP.
- **SD_3** (ANY): Send value 3, transmitted to the client with RSP.
- **SD_4** (ANY): Send value 4, transmitted to the client with RSP.
- **SD_5** (ANY): Send value 5, transmitted to the client with RSP.
- **SD_6** (ANY): Send value 6, transmitted to the client with RSP.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (SERVER_6_0 receives no payload data via IND, only connection/status information).

Functionality

SERVER_6_0 initializes a connection to the matching CLIENT_0_6 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 are transmitted to the client. Once a response is available, the block triggers IND.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_0_6**: SERVER_6_0 is only functionally compatible with a CLIENT_0_6 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 6 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_0_6 with the matching field count.

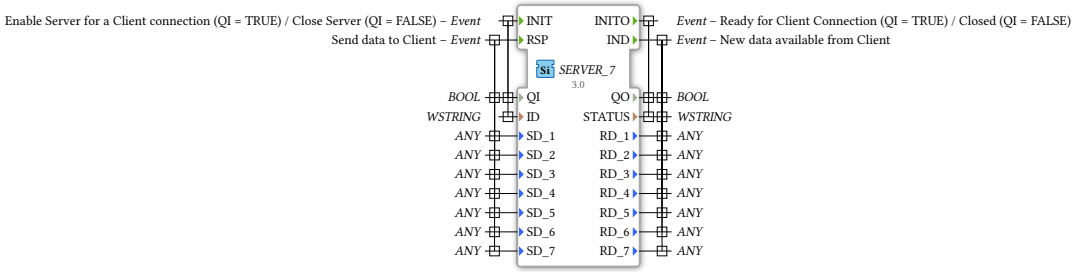
Comparison with similar function blocks

- **CLIENT_0_6**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_6_0 provides the generic server variant of the GEN_SERVER family tailored to 6 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.75 SERVER_7



• – ○ • – ○ • – ○ •

Introduction

The `SERVER_7` function block is the generic server variant with 7 send and 7 receive data fields for communicating with a matching CLIENT_7 block. It transmits 7 data values (`SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7`) to the client and receives 7 data values (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7`) back. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.
- **SD_3** (ANY): Send value 3, transmitted to the client with `RSP`.
- **SD_4** (ANY): Send value 4, transmitted to the client with `RSP`.
- **SD_5** (ANY): Send value 5, transmitted to the client with `RSP`.
- **SD_6** (ANY): Send value 6, transmitted to the client with `RSP`.
- **SD_7** (ANY): Send value 7, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.
- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.
- **RD_4** (ANY): Receive value 4, delivered by the client with `IND`.
- **RD_5** (ANY): Receive value 5, delivered by the client with `IND`.
- **RD_6** (ANY): Receive value 6, delivered by the client with `IND`.
- **RD_7** (ANY): Receive value 7, delivered by the client with `IND`.

Functionality

`SERVER_7` initializes a connection to the matching `CLIENT_7` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, the values `SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7` are transmitted to the client. Once response data (`RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7`) is available, the block triggers `IND` and provides `RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_7`**: `SERVER_7` is only functionally compatible with a `CLIENT_7` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 7 values and receive exactly 7 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_7` with the matching field count.

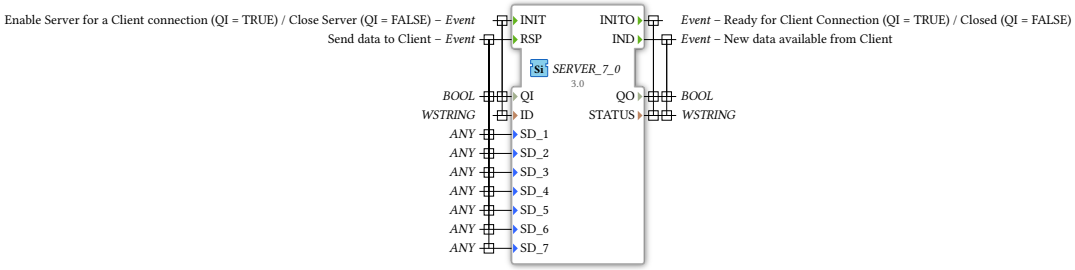
Comparison with similar function blocks

- CLIENT_7: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_7` provides the generic server variant of the `GEN_SERVER` family tailored to 7 send and 7 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.76 SERVER_7_0



• – ◦ • – ◦ • – ◦ •

Introduction

The `SERVER_7_0` function block is the generic server variant with 7 send and 0 receive data fields for communicating with a matching CLIENT_0_7 block. It transmits 7 data values (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7`) to the client without receiving any data back. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with RSP.
- **SD_2** (ANY): Send value 2, transmitted to the client with RSP.
- **SD_3** (ANY): Send value 3, transmitted to the client with RSP.
- **SD_4** (ANY): Send value 4, transmitted to the client with RSP.
- **SD_5** (ANY): Send value 5, transmitted to the client with RSP.
- **SD_6** (ANY): Send value 6, transmitted to the client with RSP.
- **SD_7** (ANY): Send value 7, transmitted to the client with RSP.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (SERVER_7_0 receives no payload data via IND, only connection/status information).

Functionality

SERVER_7_0 initializes a connection to the matching CLIENT_0_7 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 are transmitted to the client. Once a response is available, the block triggers IND.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_0_7**: SERVER_7_0 is only functionally compatible with a CLIENT_0_7 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 7 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_0_7 with the matching field count.

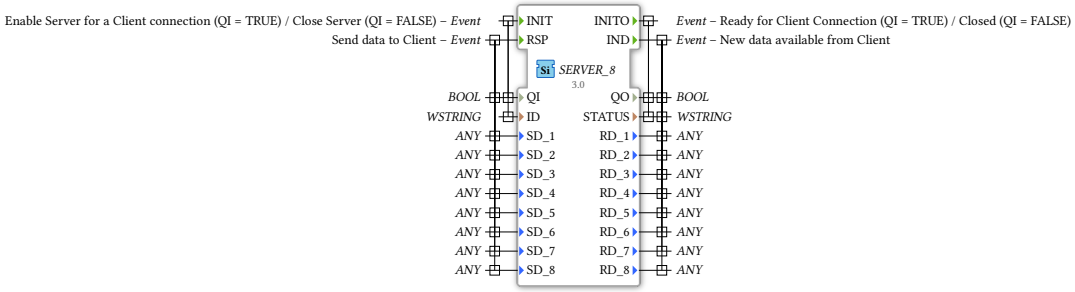
Comparison with similar function blocks

- CLIENT_0_7: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_7_0 provides the generic server variant of the GEN_SERVER family tailored to 7 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.77 SERVER_8



• – ○ • – ○ • – ○ •

Introduction

The SERVER_8 function block is the generic server variant with 8 send and 8 receive data fields for communicating with a matching CLIENT_8 block. It transmits 8 data values (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8) to the client and receives 8 data values (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8) back. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries QI and ID.
- **RSP**: Sends data to the client, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries QO and STATUS.
- **IND**: Signals that new data is available from the client, carries QO, STATUS and RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8.

Data Inputs

- **QI** (BOOL): Controls the connection state (TRUE = open connection, FALSE = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with RSP.
- **SD_2** (ANY): Send value 2, transmitted to the client with RSP.
- **SD_3** (ANY): Send value 3, transmitted to the client with RSP.
- **SD_4** (ANY): Send value 4, transmitted to the client with RSP.
- **SD_5** (ANY): Send value 5, transmitted to the client with RSP.
- **SD_6** (ANY): Send value 6, transmitted to the client with RSP.
- **SD_7** (ANY): Send value 7, transmitted to the client with RSP.
- **SD_8** (ANY): Send value 8, transmitted to the client with RSP.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with IND.
- **RD_2** (ANY): Receive value 2, delivered by the client with IND.
- **RD_3** (ANY): Receive value 3, delivered by the client with IND.
- **RD_4** (ANY): Receive value 4, delivered by the client with IND.
- **RD_5** (ANY): Receive value 5, delivered by the client with IND.
- **RD_6** (ANY): Receive value 6, delivered by the client with IND.
- **RD_7** (ANY): Receive value 7, delivered by the client with IND.
- **RD_8** (ANY): Receive value 8, delivered by the client with IND.

Functionality

SERVER_8 initializes a connection to the matching CLIENT_8 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, the values SD_1 SD_2 SD_3 SD_4 SD_5

SD_6 SD_7 SD_8 are transmitted to the client. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8) is available, the block triggers IND and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_8**: SERVER_8 is only functionally compatible with a CLIENT_8 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 8 values and receive exactly 8 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_8 with the matching field count.

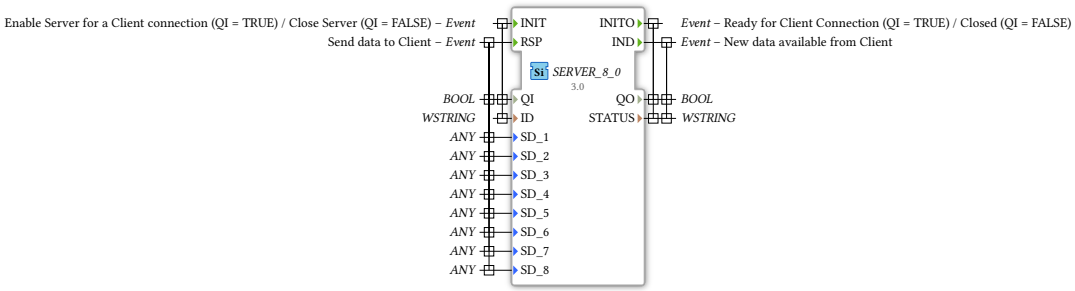
Comparison with similar function blocks

- CLIENT_8: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_8 provides the generic server variant of the GEN_SERVER family tailored to 8 send and 8 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.78 SERVER_8_0



• – ○ • – ○ • – ○ •

Introduction

The SERVER_8_0 function block is the generic server variant with 8 send and 0 receive data fields for communicating with a matching CLIENT_0_8 block. It transmits 8 data values (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8) to the client without receiving any data back. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries QI and ID.
- **RSP**: Sends data to the client, carries QI and SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.
- **SD_3** (ANY): Send value 3, transmitted to the client with `RSP`.
- **SD_4** (ANY): Send value 4, transmitted to the client with `RSP`.
- **SD_5** (ANY): Send value 5, transmitted to the client with `RSP`.
- **SD_6** (ANY): Send value 6, transmitted to the client with `RSP`.
- **SD_7** (ANY): Send value 7, transmitted to the client with `RSP`.
- **SD_8** (ANY): Send value 8, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`SERVER_8_0` receives no payload data via `IND`, only connection/status information).

Functionality

`SERVER_8_0` initializes a connection to the matching `CLIENT_0_8` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, the values `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` are transmitted to the client. Once a response is available, the block triggers `IND`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_0_8`**: `SERVER_8_0` is only functionally compatible with a `CLIENT_0_8` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 8 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_0_8` with the matching field count.

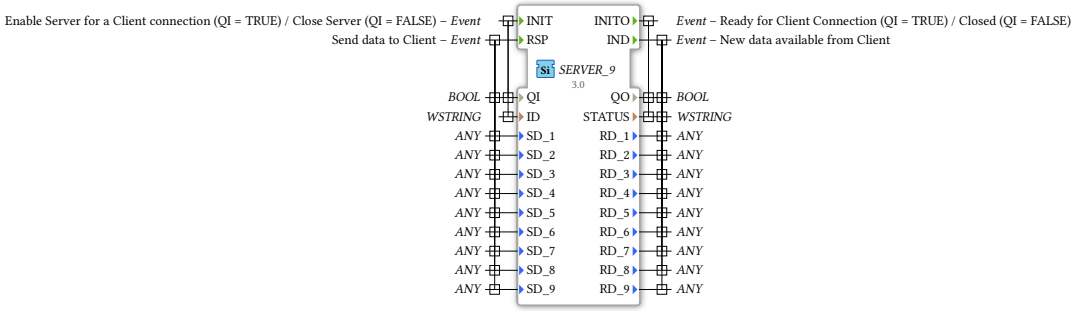
Comparison with similar function blocks

- **CLIENT_0_8**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

`SERVER_8_0` provides the generic server variant of the `GEN_SERVER` family tailored to 8 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.79 SERVER_9



• – ◦ • – ◦ • – ◦ •

Introduction

The `SERVER_9` function block is the generic server variant with 9 send and 9 receive data fields for communicating with a matching CLIENT_9 block. It transmits 9 data values (`SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9`) to the client and receives 9 data values (`RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9`) back. Like all `SERVER_*` blocks, it is based on the generic `GEN_SERVER` implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS` and `RD_1` `RD_2` `RD_3` `RD_4` `RD_5` `RD_6` `RD_7` `RD_8` `RD_9`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.
- **SD_3** (ANY): Send value 3, transmitted to the client with `RSP`.
- **SD_4** (ANY): Send value 4, transmitted to the client with `RSP`.
- **SD_5** (ANY): Send value 5, transmitted to the client with `RSP`.
- **SD_6** (ANY): Send value 6, transmitted to the client with `RSP`.
- **SD_7** (ANY): Send value 7, transmitted to the client with `RSP`.
- **SD_8** (ANY): Send value 8, transmitted to the client with `RSP`.
- **SD_9** (ANY): Send value 9, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- **RD_1** (ANY): Receive value 1, delivered by the client with `IND`.
- **RD_2** (ANY): Receive value 2, delivered by the client with `IND`.
- **RD_3** (ANY): Receive value 3, delivered by the client with `IND`.
- **RD_4** (ANY): Receive value 4, delivered by the client with `IND`.
- **RD_5** (ANY): Receive value 5, delivered by the client with `IND`.
- **RD_6** (ANY): Receive value 6, delivered by the client with `IND`.
- **RD_7** (ANY): Receive value 7, delivered by the client with `IND`.
- **RD_8** (ANY): Receive value 8, delivered by the client with `IND`.
- **RD_9** (ANY): Receive value 9, delivered by the client with `IND`.

Functionality

SERVER_9 initializes a connection to the matching CLIENT_9 block via INIT (when QI = TRUE) or closes it (when QI = FALSE); completion is confirmed via INITO. Via RSP, the values SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9 are transmitted to the client. Once response data (RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9) is available, the block triggers IND and provides RD_1 RD_2 RD_3 RD_4 RD_5 RD_6 RD_7 RD_8 RD_9.

Technical Features

- **Generic implementation**: eclipse4diac::core::GenericClassName = 'GEN_SERVER', the same C++ base as all other SERVER_* variants; the number and type (ANY) of send/receive fields are fixed per instantiation via the type definition.
- **ANY data fields**: All SD_i/RD_i are generically (ANY) typed and adapt to whichever data type is wired to them.
- **Counterpart CLIENT_9**: SERVER_9 is only functionally compatible with a CLIENT_9 block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, QO = FALSE.
2. **Connecting**: INIT with QI = TRUE is processed.
3. **Connected**: INITO with QO = TRUE confirms the connection.
4. **Data exchange**: RSP/IND cycle for send/receive data.
5. **Disconnecting**: INIT with QI = FALSE is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 9 values and receive exactly 9 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a CLIENT_9 with the matching field count.

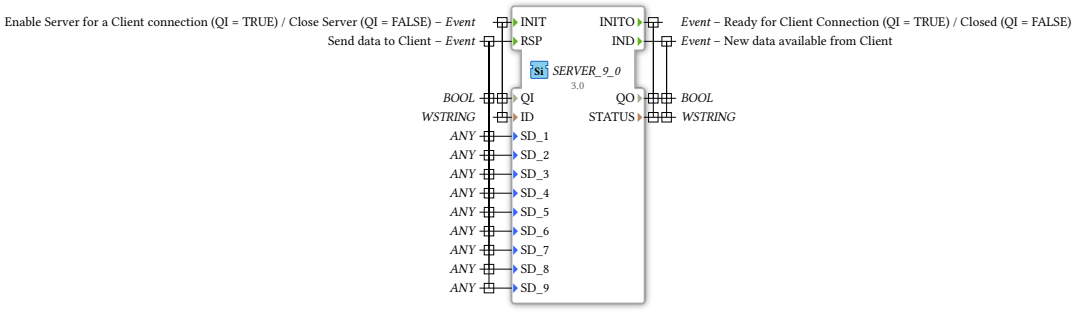
Comparison with similar function blocks

- CLIENT_9: the direct counterpart on the client side, with send/receive field counts swapped.
- CLIENT_1 / SERVER_1: the same generic implementation with one send and one receive field each.

Conclusion

SERVER_9 provides the generic server variant of the GEN_SERVER family tailored to 9 send and 9 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.80 SERVER_9_0



• – ○ • – ○ • – ○ •

Introduction

The SERVER_9_0 function block is the generic server variant with 9 send and 0 receive data fields for communicating with a matching CLIENT_0_9 block. It transmits 9 data values (SD_1 SD_2 SD_3 SD_4 SD_5 SD_6 SD_7 SD_8 SD_9) to the client without receiving any data back. Like all SERVER_* blocks, it is based on the generic GEN_SERVER implementation — the same C++ base as CLIENT_1/SERVER_1 only the number of send/receive fields differs per instantiation.

Interface Structure

Event Inputs

- **INIT**: Enables the server for a client connection (QI = TRUE) or closes it (QI = FALSE), carries `QI` and `ID`.
- **RSP**: Sends data to the client, carries `QI` and `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9`.

Event Outputs

- **INITO**: Confirms connection setup/teardown, carries `QO` and `STATUS`.
- **IND**: Signals that new data is available from the client, carries `QO`, `STATUS`.

Data Inputs

- **QI** (BOOL): Controls the connection state (`TRUE` = open connection, `FALSE` = close connection).
- **ID** (WSTRING): Connection identifier (e.g. target address/port).
- **SD_1** (ANY): Send value 1, transmitted to the client with `RSP`.
- **SD_2** (ANY): Send value 2, transmitted to the client with `RSP`.
- **SD_3** (ANY): Send value 3, transmitted to the client with `RSP`.
- **SD_4** (ANY): Send value 4, transmitted to the client with `RSP`.
- **SD_5** (ANY): Send value 5, transmitted to the client with `RSP`.
- **SD_6** (ANY): Send value 6, transmitted to the client with `RSP`.
- **SD_7** (ANY): Send value 7, transmitted to the client with `RSP`.
- **SD_8** (ANY): Send value 8, transmitted to the client with `RSP`.
- **SD_9** (ANY): Send value 9, transmitted to the client with `RSP`.

Data Outputs

- **QO** (BOOL): Current connection status.
- **STATUS** (WSTRING): Status information about the connection.
- No receive fields (`SERVER_9_0` receives no payload data via `IND`, only connection/status information).

Functionality

`SERVER_9_0` initializes a connection to the matching `CLIENT_0_9` block via `INIT` (when `QI = TRUE`) or closes it (when `QI = FALSE`); completion is confirmed via `INITO`. Via `RSP`, the values `SD_1` `SD_2` `SD_3` `SD_4` `SD_5` `SD_6` `SD_7` `SD_8` `SD_9` are transmitted to the client. Once a response is available, the block triggers `IND`.

Technical Features

- **Generic implementation**: `eclipse4diac::core::GenericClassName = 'GEN_SERVER'`, the same C++ base as all other `SERVER_*` variants; the number and type (`ANY`) of send/receive fields are fixed per instantiation via the type definition.
- **`ANY` data fields**: All `SD_i`/`RD_i` are generically (`ANY`) typed and adapt to whichever data type is wired to them.
- **Counterpart `CLIENT_0_9`**: `SERVER_9_0` is only functionally compatible with a `CLIENT_0_9` block on the client side — the number of send fields on one side must match the number of receive fields on the other.

State Overview

1. **Disconnected**: Initial state, `QO = FALSE`.
2. **Connecting**: `INIT` with `QI = TRUE` is processed.
3. **Connected**: `INITO` with `QO = TRUE` confirms the connection.
4. **Data exchange**: `RSP`/`IND` cycle for send/receive data.
5. **Disconnecting**: `INIT` with `QI = FALSE` is processed.

Application Scenarios

- **Communication between distributed control systems** that need to send exactly 9 values and receive exactly 0 values, without carrying unused surplus data fields.
- **Server-side connection** to a communication partner already laid out as a `CLIENT_0_9` with the matching field count.

Comparison with similar function blocks

- **CLIENT_0_9**: the direct counterpart on the client side, with send/receive field counts swapped.
- **CLIENT_1** / **SERVER_1**: the same generic implementation with one send and one receive field each.

Conclusion

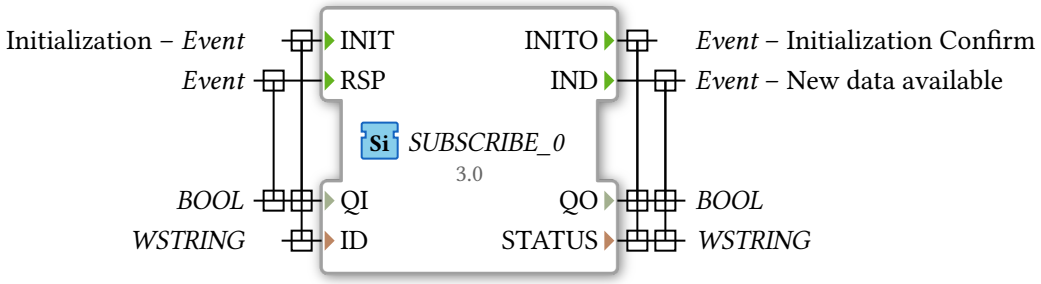
`SERVER_9_0` provides the generic server variant of the `GEN_SERVER` family tailored to 9 send and 0 receive fields, suiting network connections whose payload count differs from the standard single-field variant.

2.11.6.81 SUBSCRIBE_0

• – ◦ • – ◦ • – ◦ •

Introduction

The SUBSCRIBE_0 function block is used to subscribe to data from a PUBLISH_0 block. It enables communication between different components in a distributed automation system by receiving data from a publisher and triggering events when new data is available.



Interface Structure

Event Inputs

- **INIT** (Type: EInit) - Initialization Event
- Linked to: QI, ID
- **RSP** (Type: Event) - Response Event
- Linked to: QI

Event Outputs

- **INITO** (Type: EInit) - Initialization Confirmation
- Linked to: QO, STATUS
- **IND** (Type: Event) - Indication of New Available Data
- Linked to: QO, STATUS

Data Inputs

- **QI** (Type: BOOL) - Qualifier for Initialization
- **ID** (Type: WSTRING) - Subscription Connection Identifier

Data Outputs

- **QO** (Type: BOOL) - Qualifier for Initial State
- **STATUS** (Type: WSTRING) - Status Information

Adapter

No adapter interfaces are present.

Functionality

The SUBSCRIBE_0 block initializes itself via the INIT event and establishes a connection to a PUBLISH_0 block, which is identified by the ID parameter. After successful initialization, the block confirms this via the INITO event. When new data is received from the publisher, the IND event is triggered, informing the application about available data.

Technical Features

- Uses WSTRING data types for ID and STATUS for international character support
- Implements a qualification model with QI and QO for state management
- Provides status feedback via the STATUS output

State Overview

1. **Not Initialized**: Initial state before INIT
2. **Initialization Phase**: During INIT processing
3. **Ready**: Successfully initialized and waiting for data
4. **Data Receiving**: Processes incoming data and triggers IND

Application Scenarios

- Distributed automation systems with data distribution
- IoT applications with a publisher-subscriber pattern
- Monitoring systems that collect status data from various sources
- Control systems with decoupled communication between components

⚖️ Comparison with Similar Blocks

Compared to other communication blocks, SUBSCRIBE_0 focuses specifically on the publisher-subscriber pattern and provides a simple interface for subscribing to data streams. Unlike client-server modules, it operates asynchronously and is push-based.

Conclusion

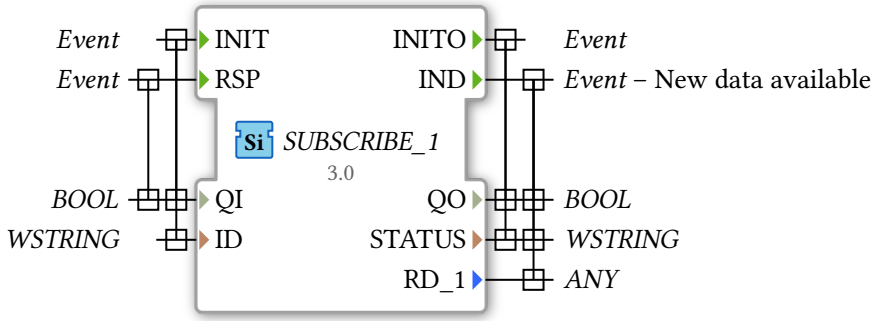
The SUBSCRIBE_0 function block offers a robust solution for subscribing to data in distributed automation systems. Its clear interface and established publisher-subscriber pattern make it a reliable choice for loosely coupled communication scenarios.

2.11.6.82 SUBSCRIBE_1

• – ◦ • – ◦ • – ◦ •

Introduction

The SUBSCRIBE_1 function block is used to subscribe to data from a PUBLISH_1 block. It enables communication between different components in a distributed system by receiving data from a publisher and making it available when new data is available.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **RSP**: Response event with associated data QI

Event Outputs

- **INITO**: Initialization output with associated data QO and STATUS
- **IND**: Indication event when new data is available with associated data QO, STATUS, and RD_1

Data Inputs

- **QI** (BOOL): Qualifier Input - Controls block activation
- **ID** (WSTRING): Identifier - Unique ID for the subscription connection

Data Outputs

- **QO** (BOOL): Qualifier Output - Block execution status
- **STATUS** (WSTRING): Status information as Unicode String

- **RD_1** (ANY): Received Data - Can contain any data type

Adapter

No adapter interfaces are available.

Functionality

The SUBSCRIBE_1 block initializes itself via the INIT event and subscribes to data from a PUBLISH_1 block with the specified ID. Upon successful initialization, it confirms this via INITO. When new data is available from the publisher, the IND event is triggered, and the received data is made available via RD_1. The STATUS output provides status information about the subscription process.

Technical Features

- Uses WSTRING for ID and STATUS for Unicode compatibility
- RD_1 supports the ANY data type for maximum flexibility in received data
- Two separate event paths for initialization and data reception
- Qualifier inputs (QI) and outputs (QO) for reliable state control

State Overview

1. **Not Initialized**: Block waits for INIT event
2. **Initialized**: Subscription active, waits for data from the publisher
3. **Data Reception**: Processes incoming data and triggers IND event

Application Scenarios

- Distributed control systems with data distribution
- IoT applications with publisher-subscriber architecture
- Industry 4.0 communication structures
- Real-time data exchange between different control components

⚖️ Comparison with Similar Blocks

Compared to other communication blocks, SUBSCRIBE_1 offers a specialized solution for the publisher-subscriber pattern with a focus on Flexibility is enhanced by the ANY data type and Unicode support.

Conclusion

The SUBSCRIBE_1 function block is an essential building block for distributed systems in 4diac, enabling robust and flexible communication between different components. Its simple interface and support for any data type make it versatile for use in various automation applications.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

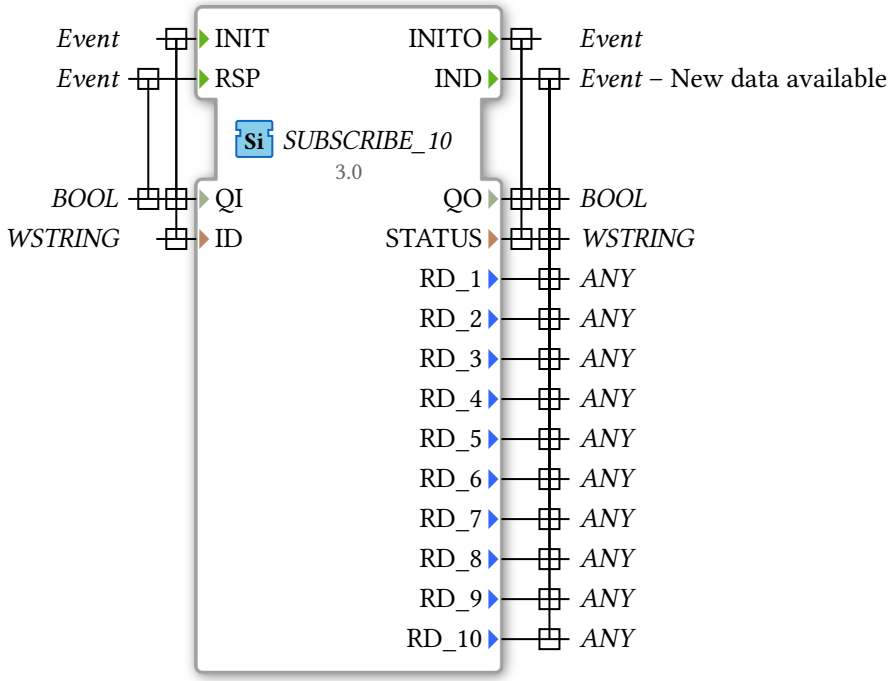
]

2.11.6.83 SUBSCRIBE_10

• – ◦ • – ◦ • – ◦ •

Introduction

The SUBSCRIBE_10 function block acts as a subscriber in a publish-subscribe communication pattern and allows data to be received from a PUBLISH_10 block. The block can receive and process up to 10 different data values simultaneously.



Interface Structure

Event Inputs

- **INIT** (Type: EInit) - Initialization Event
- Linked to: QI, ID
- **RSP** (Type: Event) - Response Event
- Linked to: QI

Event Outputs

- **INITO** (Type: EInit) - Initialization Output
- Linked to: QO, STATUS
- **IND** (Type: Event) - Indication Event (New Data Available)
- Linked to: QO, STATUS, RD_1 to RD_10

Data Inputs

- **QI** (BOOL) - Qualified Input (Activation)
- **ID** (WSTRING) - Identification String for the Connection

Data Outputs

- **QO** (BOOL) - Qualified Output (Status)
- **STATUS** (WSTRING) - Status Information
- **RD_1** to **RD_10** (ANY) - Received Data Values 1-10

Adapter

No adapter interfaces available.

Functionality

The SUBSCRIBE_10 block initializes via the INIT event and connects to a corresponding PUBLISH_10 block based on the specified ID. As soon as new data is available from the publisher, the IND event is triggered, and the received data is made available via the RD_1 to RD_10 outputs.

Technical Features

- Supports the ANY data type for all received data, offering maximum flexibility in data types
- Can receive up to 10 different data values in parallel
- Uses WSTRING for status and identification information

- Implements a reliable publish-subscribe communication pattern

State Overview

1. **Not Initialized**: Block is inactive
2. **Initialized**: Connection to the publisher established, waiting for data
3. **Data Receiving**: Receives and processes incoming data
4. **Error State**: In case of connection problems or errors

Application Scenarios

- Distributed control systems
- Data distribution in automation networks
- Communication between different control components
- Monitoring systems with multiple data sources

⚖️ Comparison with Similar Blocks

Compared to simpler subscribe blocks, SUBSCRIBE_10 offers the ability to receive up to 10 different data values simultaneously, making it suitable for more complex applications with multiple data points.

Conclusion

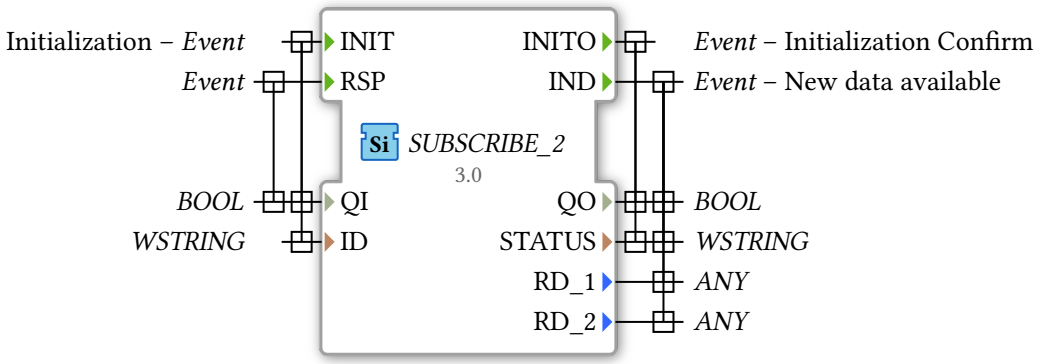
The SUBSCRIBE_10 function block is a powerful tool for distributed communication in IEC 61499 systems. Its ability to process up to 10 different data values and its flexible ANY data type support make it particularly suitable for complex automation applications with extensive data exchange requirements.

2.11.6.84 SUBSCRIBE_2

• – ◦ • – ◦ • – ◦ •

Introduction

The SUBSCRIBE_2 function block acts as a subscriber for a PUBLISH_2 block and enables the receipt of data via a publish-subscribe communication pattern. The block can receive and process two different data values from a publisher.



Interface Structure

Event Inputs

- **INIT**: Initialization event with the associated data QI and ID
- **RSP**: Response event with the associated data value QI

Event Outputs

- **INITO**: Initialization confirmation with the data QO and STATUS
- **IND**: Indication event when new data is available with QO, STATUS, RD_1, and RD_2

Data Inputs

- **QI** (BOOL): Qualifier Input - Controls block activation
- **ID** (WSTRING): Identification string for assignment to the corresponding publisher

Data Outputs

- **QO** (BOOL): Qualifier Output - Block operation status
- **STATUS** (WSTRING): Status information as a Unicode string
- **RD_1** (ANY): Received Data 1 (any data type)
- **RD_2** (ANY): Received Data 2 (any data type)

Adapter

No adapter interfaces are available.

Functionality

The SUBSCRIBE_2 block initializes itself via the INIT event and subscribes to a specific publisher using the ID identifier. After successful initialization, the block confirms this via INITO. When data is received from the publisher, the block triggers the IND event and makes the received data available via RD_1 and RD_2.

Technical Features

- Supports receiving two different data values simultaneously
- Uses WSTRING for status and identification information
- Implements generic data types (ANY) for maximum flexibility
- Based on the GEN_SUBSCRIBE generic template

State Overview

The block goes through the following main states:

1. **Not Initialized**: Block waits for the INIT event
2. **Initialized**: Block is ready to receive data
3. **Data Receipt**: Processing incoming data and triggering IND

Application Scenarios

- Distributed control systems with data distribution
- IoT applications with publisher-subscriber architecture
- Industrial automation with decentralized data communication
- Systems with flexible data type requirements

⚖️ Comparison with Similar Blocks

Compared to simpler SUBSCRIBE blocks, SUBSCRIBE_2 offers the ability to receive two different data values in parallel. The use of ANY data types allows for greater flexibility compared to type-specific implementations.

Conclusion

The SUBSCRIBE_2 block represents an enhanced solution for publish-subscribe communication in 4diac systems, offering exceptional flexibility through its support for two data channels and generic data types.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

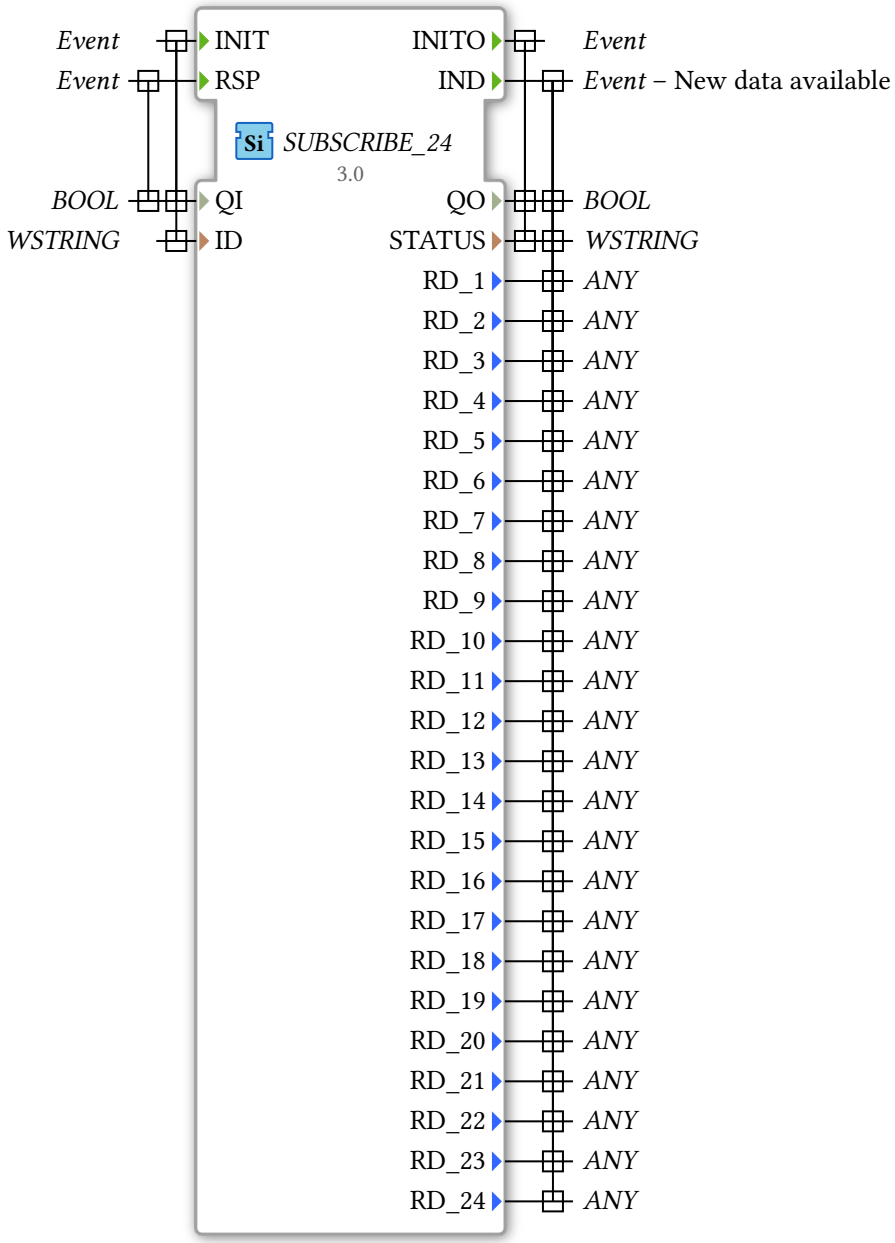
]

2.11.6.85 SUBSCRIBE_24

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SUBSCRIBE_24` subscribes to a `PUBLISH_24` block and provides the received data (up to 24 values) at its outputs.



Interface Structure

Event Inputs

- **INIT** (`EInit`): initialisation. Associated with `QI`, `ID`.
- **RSP**: response to an indication. Associated with `QI`.

Event Outputs

- **INITO** (`EInit`): initialisation confirm. Associated with `QO`, `STATUS`.
- **IND**: new data available. Associated with `QO`, `STATUS`, `RD_1` to `RD_24`.

Data Inputs

- **QI** (`BOOL`): event input qualifier
- **ID** (`WSTRING`): identification string of the communication channel (must match that of the `PUBLISH_24`)

Data Outputs

- **QO** (`BOOL`): event output qualifier
- **STATUS** (`WSTRING`): status information

- **RD_1** to **RD_24** (ANY): received data values

Adapters

No adapter interfaces available.

Functionality

After initialisation with INIT, the block signals IND whenever new data is received and provides the values RD_1 to RD_24. The service is implemented as a service interface (APPLICATION ↔ RESOURCE); RSP answers the indication.

Technical Details

- Generic block (generic class GEN_SUBSCRIBE): the number of data ports is 24 here, their types (ANY) are fixed at instantiation
- Unicode strings (WSTRING) for ID and STATUS
- Counterpart: PUBLISH_24 with SD_1 to SD_24

Application Scenarios

- Receiving many related values that a PUBLISH_24 sends elsewhere

⚖️ Comparison with Similar Building Blocks

SUBSCRIBE_10 receives up to 10 data values, SUBSCRIBE_24 up to 24.

Conclusion

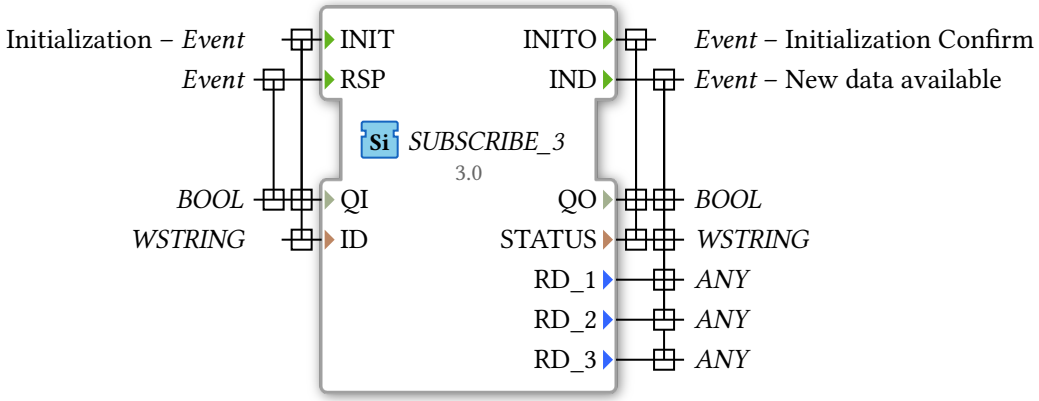
SUBSCRIBE_24 is the receiving counterpart of PUBLISH_24 for up to 24 data values.

2.11.6.86 SUBSCRIBE_3

• – ◦ • – ◦ • – ◦ •

Introduction

The SUBSCRIBE_3 function block is used to subscribe to data from a PUBLISH_3 block. It enables the reception of three different data streams over a network connection and provides a standardized mechanism for communication between distributed system components.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **RSP**: Response event with associated data QI

Event Outputs

- **INITO**: Initialization confirmation with associated data QO and STATUS
- **IND**: Data availability event with associated data QO, STATUS, RD_1, RD_3, and RD_2

Data Inputs

- **QI** (BOOL): Qualifier for initialization and operation
- **ID** (WSTRING): Identification string for the connection

Data Outputs

- **QO** (BOOL): Qualifier for output state
- **STATUS** (WSTRING): Connection status information
- **RD_1** (ANY): Received data type 1
- **RD_2** (ANY): Received data type 2
- **RD_3** (ANY): Received data type 3

Adapter

No adapter interfaces available.

Functionality

The SUBSCRIBE_3 block initializes itself via the INIT event and establishes a connection to a corresponding PUBLISH_3 block. After successful initialization, it confirms this via INITO. When data is received from the publisher, the IND event is triggered, and the received data is output via RD_1, RD_2, and RD_3. The STATUS parameter provides information about the connection status.

Technical Features

- Supports three independent data channels (RD_1, RD_2, RD_3) with ANY data type
- Uses WSTRING for ID and STATUS for international character support
- Implements a reliable initialization protocol with QI/QO handshake
- Generic implementation via the GEN_SUBSCRIBE base class

State Overview

1. **Not Initialized**: Block waits for INIT event
2. **Initialization**: Processing INIT with ID parameter
3. **Ready**: Successful connection to the publisher
4. **Data Receiving**: Processing incoming data with IND trigger

Application Scenarios

- Distributed control systems in automation technology
- Data distribution in IoT applications
- Communication between different control levels
- Machine-to-machine communication in Industry 4.0 environments

⚖️ Comparison with Similar Blocks

Compared to simpler SUBSCRIBE blocks, this offers SUBSCRIBE_3 offers the ability to receive three different data streams in parallel. The use of ANY data types allows for greater flexibility in the transmitted data formats.

Conclusion

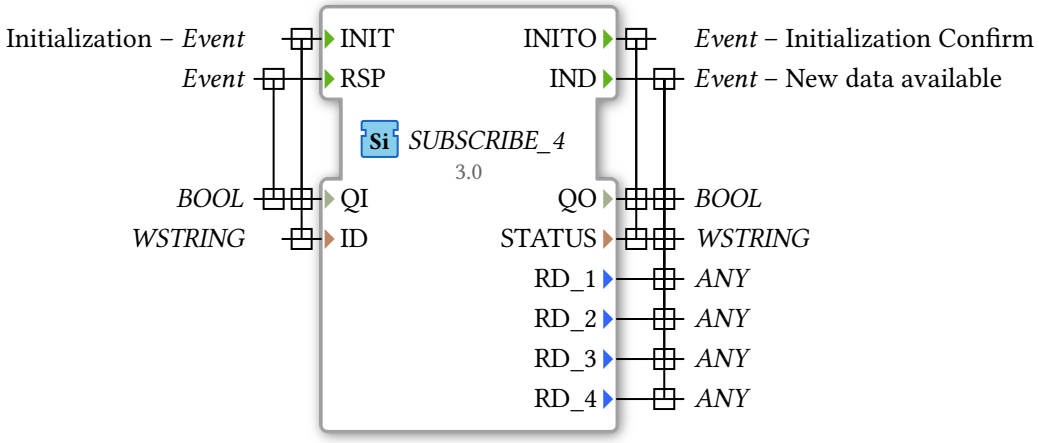
The SUBSCRIBE_3 function block represents a powerful solution for communication in distributed automation systems. Its support for multiple data channels and flexible data types makes it particularly suitable for complex applications where multiple data sources need to be monitored simultaneously.

2.11.6.87 SUBSCRIBE_4

• − ○ • − ○ • − ○ •

Introduction

The SUBSCRIBE_4 function block is used to subscribe to data from a PUBLISH_4 block. It allows the receipt of up to four different data points via a communication link and makes them available when they become available.



Interface Structure

Event Inputs

- **INIT** (Type: EInit) - Initialization Event
- Linked to: QI, ID
- **RSP** (Type: Event) - Response Event
- Linked to: QI

Event Outputs

- **INITO** (Type: EInit) - Initialization Acknowledgement
- Linked to: QO, STATUS
- **IND** (Type: Event) - Data Availability Message
- Linked to: QO, STATUS, RD_1, RD_3, RD_2, RD_4

Data Inputs

- **QI** (BOOL) - Qualified Input (Activation/Deactivation)
- **ID** (WSTRING) - Identifier for the Communication Connection

Data Outputs

- **QO** (BOOL) - Qualified Output (Status Feedback)
- **STATUS** (WSTRING) - Status Information
- **RD_1** (ANY) - Received Data 1
- **RD_2** (ANY) - Received Data 2
- **RD_3** (ANY) - Received Data 3
- **RD_4** (ANY) - Received Data 4

Adapters

No adapter interfaces are available.

Functionality

Upon receiving the INIT event, the SUBSCRIBE_4 block initializes a subscription connection to a PUBLISH_4 block using the specified ID. After successful initialization, it confirms this with INITO. When new data is available from the publisher, the IND event is triggered, and the received data is made available via the RD_1 to RD_4 outputs.

Technical Features

- Supports receiving up to four different data types (ANY type)
- Uses WSTRING for status and identification information
- Generic implementation via the GEN_SUBSCRIBE base class
- Robust error handling via STATUS output

State Overview

1. **Inactive**: Block is not initialized

2. **Initialized**: Subscription active, waiting for data
3. **Data Receiving**: Processes incoming data and outputs an IND event

Application Scenarios

- Distributed automation systems
- Data distribution in decentralized controllers
- Communication between different control components
- Monitoring systems with data subscription

⚖ Comparison with Similar Blocks

Compared to simpler SUBSCRIBE blocks, SUBSCRIBE_4 offers the ability to receive up to four different data points simultaneously. This enhanced functionality makes it particularly suitable for more complex communication scenarios where multiple data sources need to be consolidated.

Conclusion

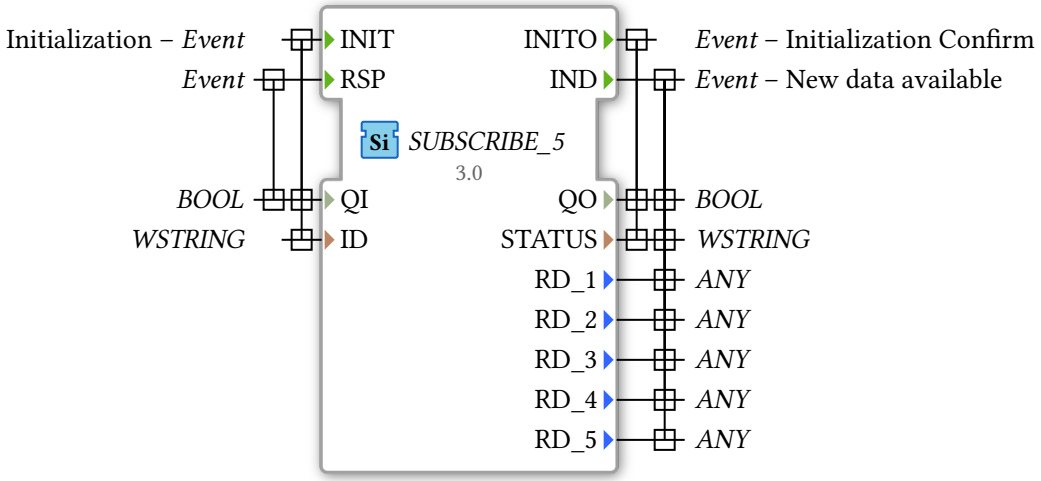
The SUBSCRIBE_4 function block represents a powerful solution for subscription-based communication in IEC 61499 systems. Its ability to manage multiple data points in parallel makes it a flexible choice for demanding automation applications with distributed data communication.

2.11.6.88 SUBSCRIBE_5

• − ◦ • − ◦ • − ◦ •

Introduction

The SUBSCRIBE_5 function block is used to subscribe to data from a PUBLISH_5 block. It allows the reception of up to five different data points over a network connection and provides a standardized interface for communication between distributed automation components.



Interface Structure

Event Inputs

- **INIT**: Initialization event with the associated data QI and ID
- **RSP**: Response event with the associated data point QI

Event Outputs

- **INITO**: Initialization confirmation with the data QO and STATUS
- **IND**: Indication event when new data is available with QO, STATUS, and the five data points RD_1 to RD_5

Data Inputs

- **QI** (BOOL): Qualifier for initialization - activates/deactivates the block
- **ID** (WSTRING): Identification string for assignment to the corresponding PUBLISH_5 block

Data Outputs

- **QO** (BOOL): Qualifier output - indicates the operating status
- **STATUS** (WSTRING): Status Information and Error Messages
- **RD_1** to **RD_5** (ANY): Received data points 1 to 5 with any data type

Adapter

No adapter interfaces are available.

Functionality

The SUBSCRIBE_5 block initializes itself via the INIT event and then establishes a connection to the corresponding PUBLISH_5 block. Upon successful initialization, it confirms this via INITO. As soon as new data is available from the publisher, it is output via the IND event using the corresponding data outputs RD_1 to RD_5. The STATUS output provides information about the connection status and any errors.

Technical Features

- Supports up to five different data points simultaneously
- Uses WSTRING for ID and STATUS for international character set support
- ANY type for data outputs enables flexible data types
- Generic implementation via the GEN_SUBSCRIBE base class

State Overview

1. **Not Initialized**: Block waits for INIT event
2. **Initialization Phase**: Processing INIT with ID parameter
3. **Connected**: Successful connection to the publisher, ready to receive data
4. **Data Reception**: Processing incoming data and output via IND

Application Scenarios

- Distributed automation systems
- Data distribution in production plants
- Machine-to-machine communication
- Monitoring systems with multiple sensor data
- SCADA systems with decentralized data sources

⚖️ Comparison with Similar Blocks

Compared to simpler SUBSCRIBE blocks, SUBSCRIBE_5 offers the ability to process up to five different data points in parallel. received. The use of ANY types makes it more flexible than type-specific subscribe blocks, but requires a correct type mapping to the corresponding publisher.

Conclusion

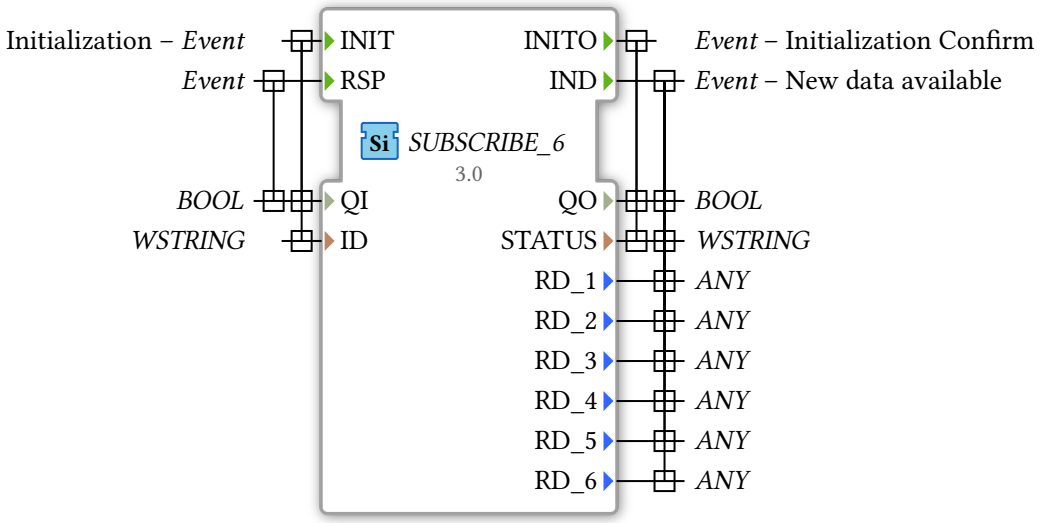
The SUBSCRIBE_5 function block represents a powerful and flexible solution for receiving multiple data streams in distributed automation systems. Its generic implementation and support for various data types make it particularly suitable for complex applications with variable data structures.

2.11.6.89 SUBSCRIBE_6

• − ◦ • − ◦ • − ◦ •

Introduction

The SUBSCRIBE_6 function block is used to subscribe to data from a PUBLISH_6 block. It allows the receipt of up to six different data values via a network connection and makes them available for further processing in an IEC 61499 system.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **RSP**: Response event with associated data QI

Event Outputs

- **INITO**: Initialization confirmation with associated data QO and STATUS
- **IND**: Indication event for newly available data with associated data QO, STATUS, and RD_1 to RD_6

Data Inputs

- **QI** (BOOL): Qualifier Input - Controls block activation
- **ID** (WSTRING): Identification string for assignment to the corresponding PUBLISH_6 block

Data Outputs

- **QO** (BOOL): Qualifier Output - Block operation status
- **STATUS** (WSTRING): Status information as a Unicode string
- **RD_1** to **RD_6** (ANY): Received data values (up to 6 different data types)

Functionality

The SUBSCRIBE_6 block initializes itself via the INIT event and establishes a connection to a PUBLISH_6 block with the specified ID. Upon successful initialization, it confirms this with INITO. As soon as new data is available from the PUBLISH_6 block, the SUBSCRIBE_6 block triggers the IND event and makes the received data available via the RD_1 to RD_6 outputs.

Technical Features

- Supports up to six different data values simultaneously
- Uses WSTRING for status messages and identification
- Flexible data types through ANY type for received data
- Network-based communication between distributed systems

State Overview

The block goes through the following main states:

1. **Not Initialized**: Block is inactive
2. **Initialization**: Processing the INIT event
3. **Connected**: Successful connection to the publisher
4. **Data Reception**: Receives and processes incoming data

Application Scenarios

- Distributed control systems with data distribution

- Monitoring systems with centralized data acquisition
- Industry 4.0 applications with machine data
- SCADA systems with decentralized data sources

⚖️ Comparison with Similar Blocks

Compared to simpler Subscribe blocks, SUBSCRIBE_6 offers the ability to receive up to six different data values in parallel, enabling higher data throughput capacity. Using WSTRING for status messages allows for more detailed error information.

Conclusion

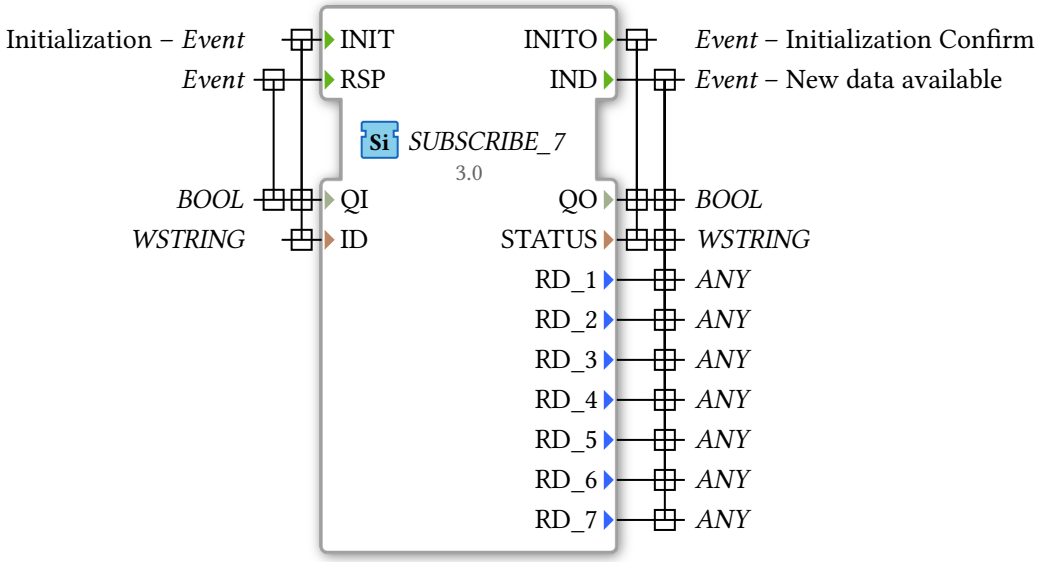
The SUBSCRIBE_6 function block is a powerful solution for distributed systems that require the simultaneous reception of multiple data streams. Its flexible architecture and support for various data types make it ideal for complex industrial automation applications.

2.11.6.90 SUBSCRIBE_7

• – ◦ • – ◦ • – ◦ •

Introduction

The SUBSCRIBE_7 function block acts as a subscriber for data from a PUBLISH_7 block. It enables the reception of up to seven different data points over a single communication link. This block is part of the IEC 61499 network communication functionality and is used in distributed automation systems.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **RSP**: Response event with associated data QI

Event Outputs

- **INITO**: Initialization confirmation with associated data QO and STATUS
- **IND**: Indication event when new data is available with associated data QO, STATUS, and RD_1 to RD_7

Data Inputs

- **QI** (BOOL): Qualifier for initialization
- **ID** (WSTRING): Identifier for the connection

Data Outputs

- **QO** (BOOL): Qualifier for output state
- **STATUS** (WSTRING): Connection status information

- **RD_1** to **RD_7** (ANY): Received data points 1 to 7

Adapter

No adapter interfaces available.

Functionality

The SUBSCRIBE_7 block initializes itself via the INIT event and establishes a connection to a corresponding PUBLISH_7 block. After successful initialization, it confirms this via INITO. When data is received from the publisher, the IND event is triggered, and the received data is output via RD_1 to RD_7. The RSP input allows for the handling of reply messages.

Technical Features

- Supports up to seven different data points simultaneously
- Uses WSTRING for status and identification information
- ANY data type for received data enables flexible data types
- Generic implementation via the GEN_SUBSCRIBE base class

State Overview

1. **Not Initialized**: Block waits for INIT event
2. **Initialization**: Processing the INIT request
3. **Ready**: Successfully connected, waiting for data
4. **Data Receiving**: Processing incoming data with IND trigger

Application Scenarios

- Distributed automation system with data distribution
- Machine networking in Industry 4.0 environments
- Data exchange between different control components
- Monitoring systems with centralized data collection

⚖️ Comparison with Similar Blocks

Compared to simpler SUBSCRIBE blocks, SUBSCRIBE_7 offers the ability to receive up to seven data points in parallel. Compared to individual communication solutions, it offers standardized IEC 61499 interfaces and simplified configuration.

Conclusion

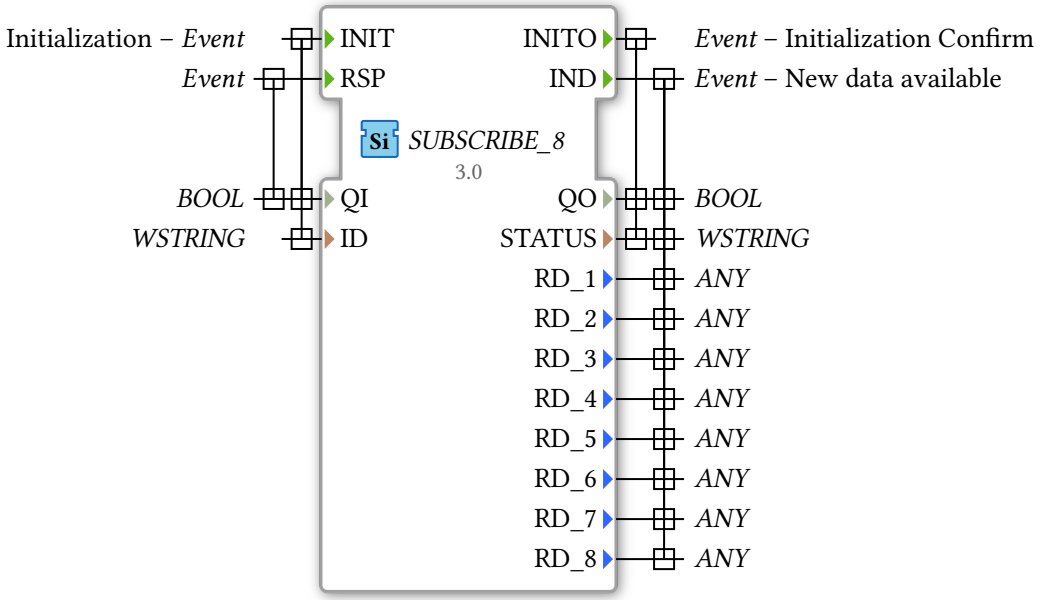
The SUBSCRIBE_7 block is a powerful communication module for distributed automation systems. Its flexible data transmission and standardized interfaces provide a reliable foundation for industrial communication applications.

2.11.6.91 SUBSCRIBE_8

• – ◦ • – ◦ • – ◦ •

Introduction

The SUBSCRIBE_8 function block acts as a subscriber for a PUBLISH_8 block and allows the receipt of up to 8 different data values over a single communication link. The block implements a publish-subscribe communication pattern and is part of the iec61499::net package.



Interface Structure

Event Inputs

- **INIT** - Initialization Event
- Linked to: QI, ID
- **RSP** - Response Event
- Linked to: QI

Event Outputs

- **INITO** - Initialization Confirmation
- Linked to: QO, STATUS
- **IND** - Indication Event for New Data
- Linked to: QO, STATUS, RD_1 to RD_8

Data Inputs

- **QI** (BOOL) - Qualified Input for Initialization
- **ID** (WSTRING) - Identification String for the Connection

Data Outputs

- **QO** (BOOL) - Qualified Output
- **STATUS** (WSTRING) - Status Information
- **RD_1** to **RD_8** (ANY) - Received Data Values 1 to 8

Adapter

No adapter interfaces available.

Functionality

The SUBSCRIBE_8 block initializes itself via the INIT event and establishes a connection to a corresponding PUBLISH_8 block. After successful initialization, it confirms this with the INITO event. When data is received from the publisher, the IND event is triggered, and the received data is made available via the RD_1 to RD_8 outputs.

Technical Features

- Supports the ANY data type for all data outputs, enabling maximum flexibility in the transmitted data types
- Uses WSTRING for status messages and identification
- Implements a reliable initialization protocol with qualification bits
- Can receive up to 8 different data values in parallel

State Overview

1. **Not Initialized** - Block waiting for INIT event
2. **Initialization** - Processing the INIT event
3. **Ready** - Successfully initialized, waiting for data
4. **Data Receiving** - Processing incoming data with IND triggering

Application Scenarios

- Distributed control systems with data distribution
- Monitoring systems with centralized data acquisition
- Communication between different control components
- Systems with a publish-subscribe architecture

⚖️ Comparison with Similar Blocks

Compared to simpler subscribe blocks, SUBSCRIBE_8 offers the ability to receive up to 8 different data values in parallel. Using the ANY data type offers greater flexibility than type-specific subscribe blocks.

Conclusion

The SUBSCRIBE_8 function block provides a powerful and flexible solution for publish-subscribe communication in distributed automation systems. Its ability to receive multiple data values of different types makes it particularly suitable for complex communication scenarios in industrial control systems.

2.11.6.92 SUBSCRIBE_9

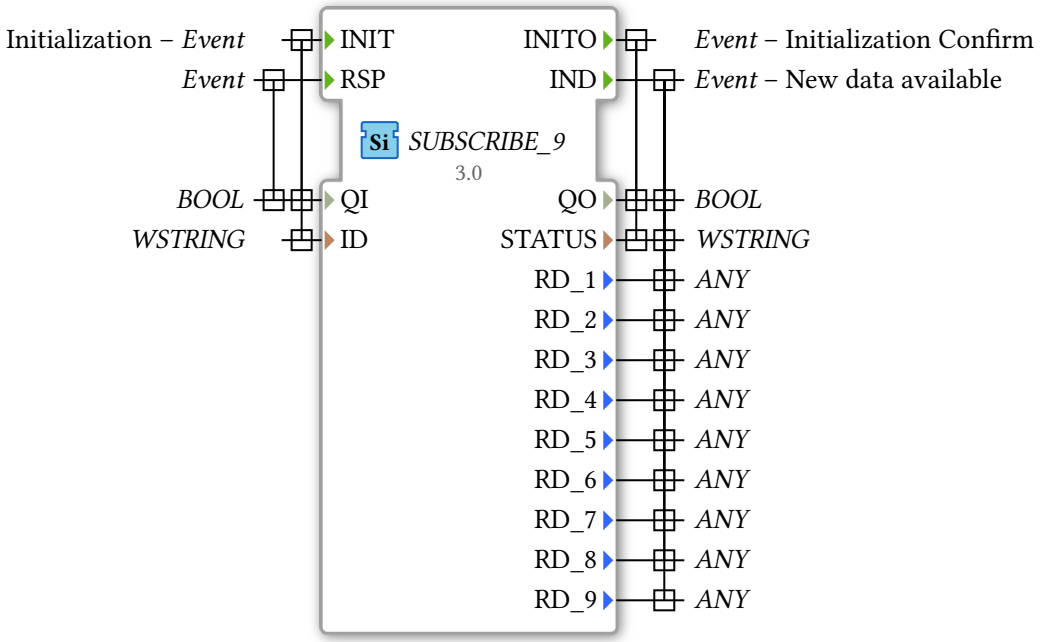
• – ◦ • – ◦ • – ◦ •

Introduction

The SUBSCRIBE_9 function block is used to subscribe to data from a PUBLISH_9 block. It allows the receipt of up to 9 different data points via a network connection and makes them available for further processing in the control system.

• – ◦ • – ◦ • – ◦

The SUBSCRIBE_9 function block is used to subscribe to data from a PUBLISH_9 block. It enables the reception of up to 9 different data points via a network connection and makes them available for further processing in the control system.



Interface Structure

Event Inputs

- **INIT**: Initialization event with associated data QI and ID
- **RSP**: Response event with associated data QI

Event Outputs

- **INITO**: Initialization confirmation with QO and STATUS
- **IND**: Indication event when new data is available with QO, STATUS, and all 9 RD_x data outputs

Data Inputs

- **QI** (BOOL): Qualifier for initialization (TRUE = enable, FALSE = disable)
- **ID** (WSTRING): Identification string for the connection to the publisher

Data Outputs

- **QO** (BOOL): Qualifier output for operating status
- **STATUS** (WSTRING): Status information as Unicode String
- **RD_1** to **RD_9** (ANY): Received data points 1-9 with any data type

Adapter

No adapter interfaces available.

Functionality

Upon receiving the INIT event, the SUBSCRIBE_9 block initializes a connection to a PUBLISH_9 block based on the specified ID. After successful initialization, it confirms this with the INITO event. When data is received from the publisher, the IND event is triggered, and the data is made available via the RD_1 to RD_9 outputs.

Technical Features

- Supports up to 9 different data points in parallel
- Uses WSTRING for status and identification information
- ANY data types for the received data enable flexible data types
- Generic implementation as GEN_SUBSCRIBE

State Overview

1. **Not Initialized**: Block waits for INIT event
2. **Initialized**: Connection to the publisher established, ready to receive data
3. **Data Receiving**: Processes incoming data and triggers IND event

Application Scenarios

- Distributed control systems with data distribution
- Machine networking in Industry 4.0 environments
- Monitoring systems with centralized data collection
- Systems with multiple decentralized sensor nodes

⚖️ Comparison with Similar Blocks

Compared to simpler SUBSCRIBE blocks, SUBSCRIBE_9 offers the ability to receive up to 9 different data points simultaneously, increasing efficiency in more complex network structures.

Conclusion

The SUBSCRIBE_9 function block is a powerful solution for receiving multiple data streams in distributed automation systems and is particularly suitable for applications with high data throughput.

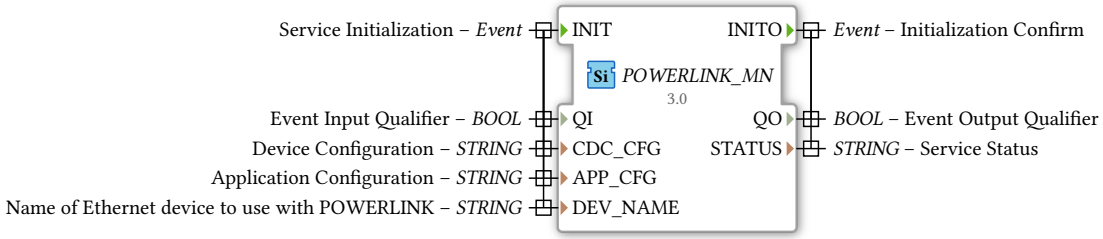
2.11.7 powerlink

2.11.7.1 POWERLINK_MN

• – ◦ • – ◦ • – ◦ •

Introduction

The POWERLINK_MN function block is used to configure the openPOWERLINK fieldbus. It acts as a Managing Node (MN) and initializes the network.



Interface Structure

Event Inputs

- **INIT**: Initializes the service. Starts the configuration process.

Event Outputs

- **INITO**: Confirms initialization. Indicates whether the start was successful.

Data Inputs

- **QI** (BOOL): Qualifier for the INIT input (TRUE = Start, FALSE = Stop).
- **CDC_CFG** (STRING): Device Configuration.
- **APP_CFG** (STRING): Application Configuration.
- **DEV_NAME** (STRING): Name of the Ethernet device to be used for POWERLINK (e.g., “eth0”).

Data Outputs

- **QO** (BOOL): Qualifier for the INITO output.
- **STATUS** (STRING): Service status information.

Functionality

This block initializes the openPOWERLINK stack on the specified Ethernet interface (`DEV_NAME`). The configuration files for the device (`CDC_CFG`) and application (`APP_CFG`) are loaded.

Application Scenarios

- Central control of a POWERLINK network.
- Master component in a real-time Ethernet system.

Metadata

Attribute	Value
Copyright	(c) 2011 AIT
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	powerlink

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

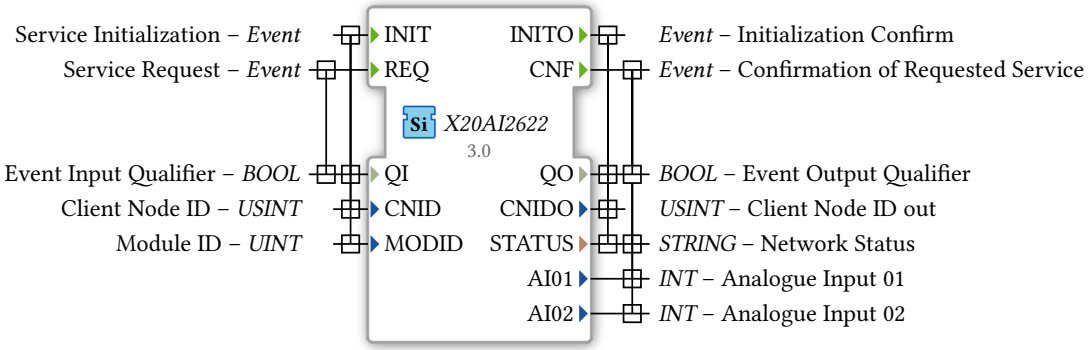
]

2.11.7.2 X20AI2622

• – ◦ • – ◦ • – ◦ •

Introduction

The X20AI2622 function block provides an interface for analog inputs in the openPOWERLINK fieldbus. It corresponds to the B&R X20AI2622 module.



Interface Structure

Event Inputs

- **INIT**: Service initialization.
- **REQ**: Request to update input data.

Event Outputs

- **INITO**: Initialization confirmation.
- **CNF**: Request to update the input data.

Data Inputs

- **QI** (BOOL): Qualifier for INIT and REQ.
- **CNID** (USINT): Client Node ID (address of the node in the POWERLINK network).
- **MODID** (UINT): Module ID (position of the module in the node).

Data Outputs

- **QO** (BOOL): Qualifier for outputs.
- **CNIDO** (USINT): Confirmed Client Node ID.
- **STATUS** (STRING): Network status.
- **AI01** (INT): Analog input 01.
- **AI02** (INT): Analog input 02.

Functionality

After initialization with `CNID` and `MODID`, the function block outputs the current analog values of the physical module at outputs `AIxx` every `REQ`.

Metadata

Attribute	Value
Copyright	(c) 2014 AIT
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	powerlink

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

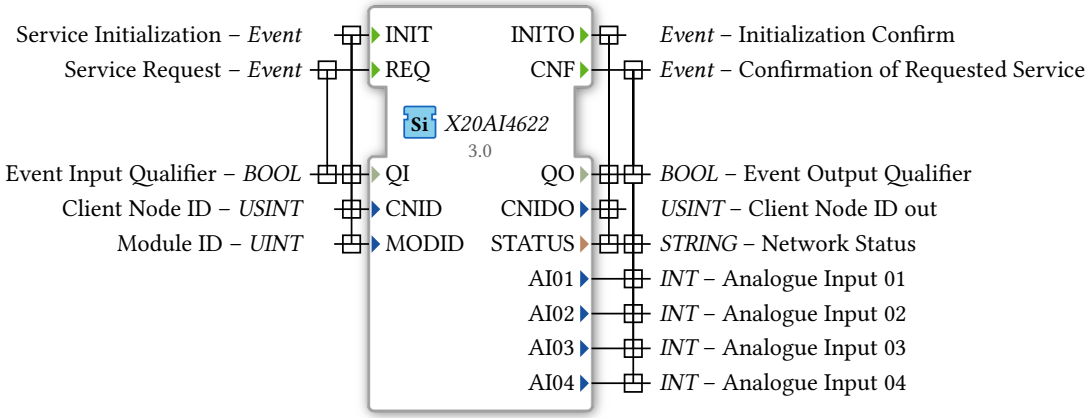
]

2.11.7.3 X20AI4622

• – ◦ • – ◦ • – ◦ •

Introduction

The X20AI4622 function block provides an interface for analog inputs in the openPOWERLINK fieldbus. It corresponds to the B&R X20AI4622 module with 4 analog inputs.



Interface Structure

Event Inputs

- **INIT**: Service initialization.
- **REQ**: Request to update input data.

Event Outputs

- **INITO**: Initialization confirmation.
- **CNF**: Request to update the input data.

Data Inputs

- **QI** (BOOL): Qualifier for INIT and REQ.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.

Data Outputs

- **QO** (BOOL): Qualifier for outputs.
- **CNIDO** (USINT): Confirmed Client Node ID.
- **STATUS** (STRING): Network status.
- **AI01** (INT): Analog input 01.
- **AI02** (INT): Analog input 02.
- **AI03** (INT): Analog input 03.
- **AI04** (INT): Analog input 04.

Functionality

After initialization, the module provides the current analog values of the four channels.

Metadata

Attribute	Value
Copyright	(c) 2011 AIT
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	Powerlink

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

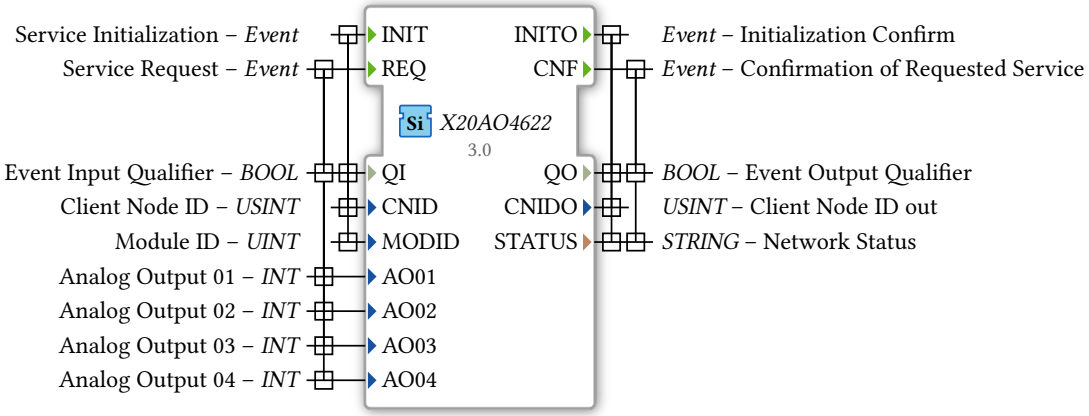
]

2.11.7.4 X20AO4622

• – ◦ • – ◦ • – ◦ •

Introduction

The X20AO4622 function block is used to control analog outputs via the openPOWERLINK fieldbus. It corresponds to the B&R X20AO4622 module (4 outputs).



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **REQ**: Writes the values of the inputs `AOxx` to the module.

Event Outputs

- **INITO**: Initialization confirmed.
- **CNF**: Write operation confirmed.

Data Inputs

- **QI** (BOOL): Qualifier.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.
- **AO01** (INT): Analog value for output 01.
- **AO02** (INT): Analog value for output 02.
- **AO03** (INT): Analog value for output 03.
- **AO04** (INT): Analog value for output 04.

Data Outputs

- **QO** (BOOL): Qualifier.
- **CNIDO** (USINT): Confirmed Node ID.
- **STATUS** (STRING): Status.

Functionality

The integer values present at inputs `AO01-AO04` are sent to the output module when the `REQ` event is triggered.

Metadata

Attribute	Value
Copyright	(c) 2011 AIT

License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	powerlink

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

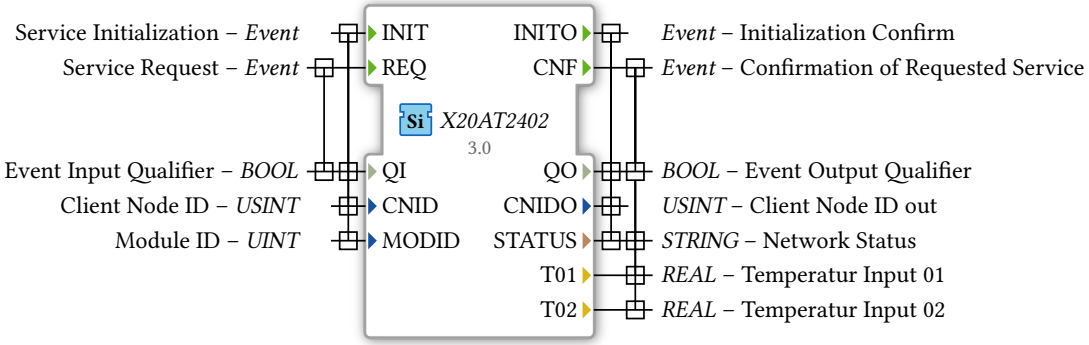
]

2.11.7.5 X20AT2402

• – ◦ • – ◦ • – ◦ •

Introduction

The X20AT2402 function block is used to read temperature values via the openPOWERLINK fieldbus. It corresponds to the B&R X20AT2402 temperature input module (2 channels).



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **REQ**: Request data update.

Event Outputs

- **INITO**: Initialization confirmed.
- **CNF**: New temperature data available.

Data Inputs

- **QI** (BOOL): Qualifier.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.

Data Outputs

- **QO** (BOOL): Qualifier.
- **CNIDO** (USINT): Confirmed Node ID.
- **STATUS** (STRING): Status.
- **T01** (REAL): Temperature input 01 (typically in °C, depending on module configuration).
- **T02** (REAL): Temperature input 02.

Functionality

Reads temperature values from the physical module and provides them as REAL values.

Metadata

Attribute	Value

Copyright	(c) 2012 AIT
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac package	powerlink

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

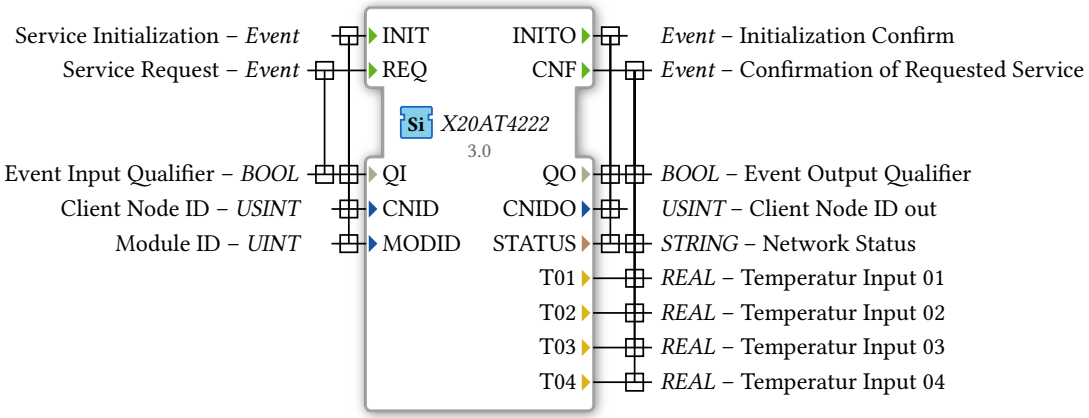
]

2.11.7.6 X20AT4222

• – ◦ • – ◦ • – ◦ •

Introduction

The X20AT4222 function block is used to read temperature values via the openPOWERLINK fieldbus. It corresponds to the B&R X20AT4222 temperature input module (4 channels).



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **REQ**: Request data update.

Event Outputs

- **INITO**: Initialization confirmed.
- **CNF**: New temperature data available.

Data Inputs

- **QI** (BOOL): Qualifier.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.

Data Outputs

- **QO** (BOOL): Qualifier.
- **CNIDO** (USINT): Confirmed Node ID.
- **STATUS** (STRING): Status.
- **T01** (REAL): Temperature Input 01.
- **T02** (REAL): Temperature Input 02.
- **T03** (REAL): Temperature Input 03.
- **T04** (REAL): Temperature Input 04.

Functionality

Reads temperature values from the physical module and provides them as real values.

Metadata

Attribute	Value
Copyright	(c) 2012 AIT
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	Powerlink

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

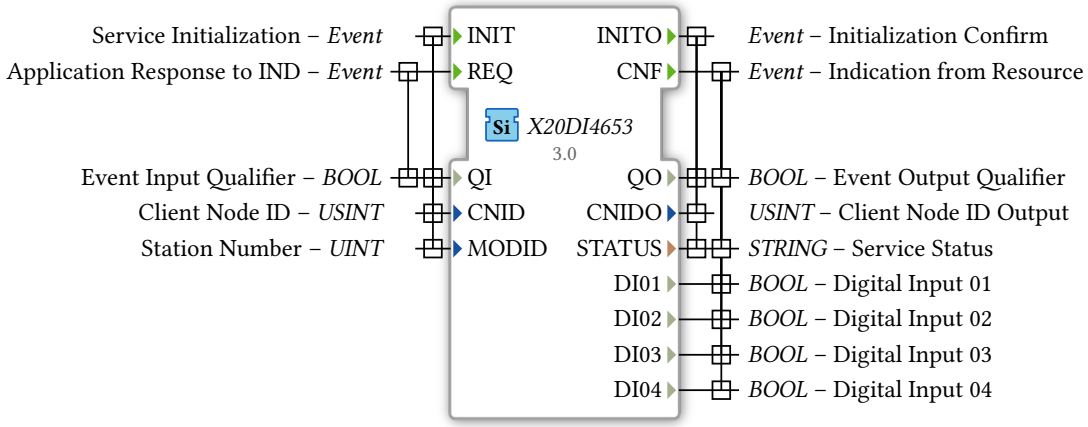
]

2.11.7.7 X20DI4653

• – ◦ • – ◦ • – ◦ •

Introduction

The X20DI4653 function block is used to read digital signals via the openPOWERLINK fieldbus. It corresponds to the B&R X20DI4653 digital input module (4 channels).



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **REQ**: Update input data.

Event Outputs

- **INITO**: Initialization confirmed.
- **CNF**: New data available.

Data Inputs

- **QI** (BOOL): Qualifier.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.

Data Outputs

- **QO** (BOOL): Qualifier.

- **CNIDO** (USINT): Confirmed Node ID.
- **STATUS** (STRING): Status.
- **DI01** (BOOL): Digital Input 01.
- **DI02** (BOOL): Digital Input 02.
- **DI03** (BOOL): Digital Input 03.
- **DI04** (BOOL): Digital Input 04.

Functionality

Reads the state of the digital inputs from the module and provides them as Boolean values.

Metadata

Attribute	Value
Copyright	(c) 2011 AIT
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	Powerlink

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

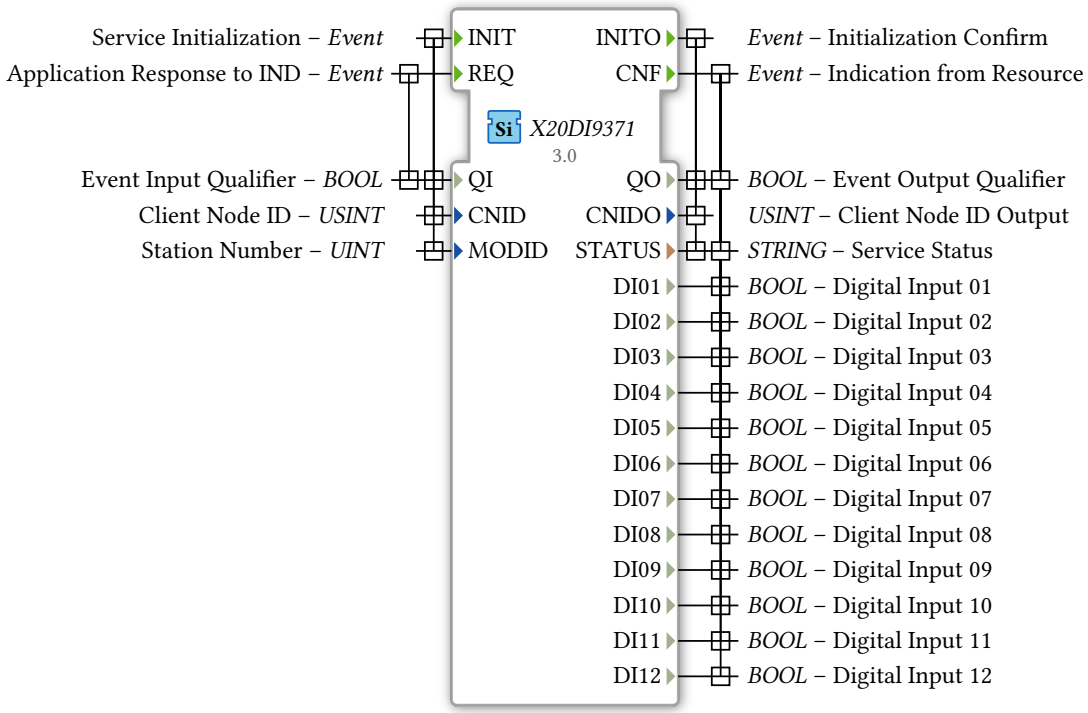
]

2.11.7.8 X20DI9371

• – ◦ • – ◦ • – ◦ •

Introduction

The X20DI9371 function block is used to read digital signals via the openPOWERLINK fieldbus. It corresponds to the B&R X20DI9371 digital input module (12 channels).



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **REQ**: Update input data.

Event Outputs

- **INITO**: Initialization confirmed.
- **CNF**: New data available.

Data Inputs

- **QI** (BOOL): Qualifier.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.

Data Outputs

- **QO** (BOOL): Qualifier.
- **CNIDO** (USINT): Confirmed Node ID.
- **STATUS** (STRING): Status.
- **DI01** - **DI12** (BOOL): Digital Inputs 01 to 12.

Functionality

Reads the status of the 12 digital inputs from the module.

Metadata

Attribute	Value
Copyright	(c) 2011 AIT
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	powerlink

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

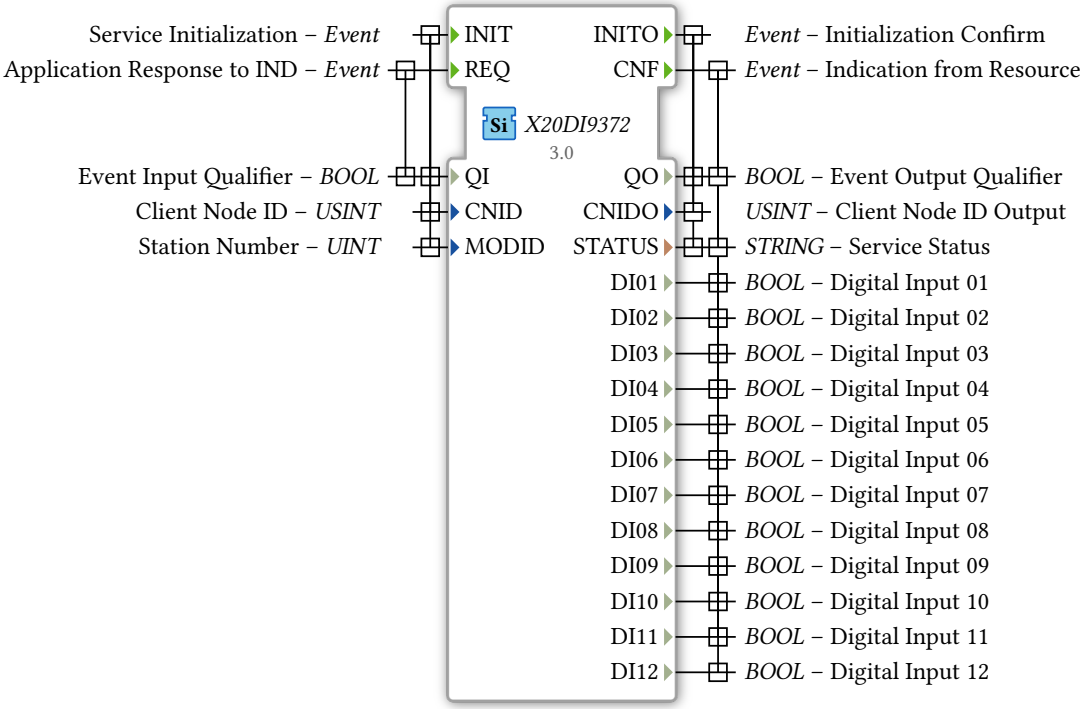
]

2.11.7.9 X20DI9372

• – ◦ • – ◦ • – ◦ •

Introduction

The X20DI9372 function block is used to read digital signals via the openPOWERLINK fieldbus. It corresponds to the B&R X20DI9372 digital input module (12 channels).



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **REQ**: Update input data.

Event Outputs

- **INITO**: Initialization confirmed.
- **CNF**: New data available.

Data Inputs

- **QI** (BOOL): Qualifier.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.

Data Outputs

- **QO** (BOOL): Qualifier.
- **CNIDO** (USINT): Confirmed Node ID.
- **STATUS** (STRING): Status.
- **DI01** - **DI12** (BOOL): Digital Inputs 01 to 12.

Functionality

Reads the status of the 12 digital inputs from the module.

Metadata

Attribute	Value
Copyright	(c) 2012 AIT
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	powerlink

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

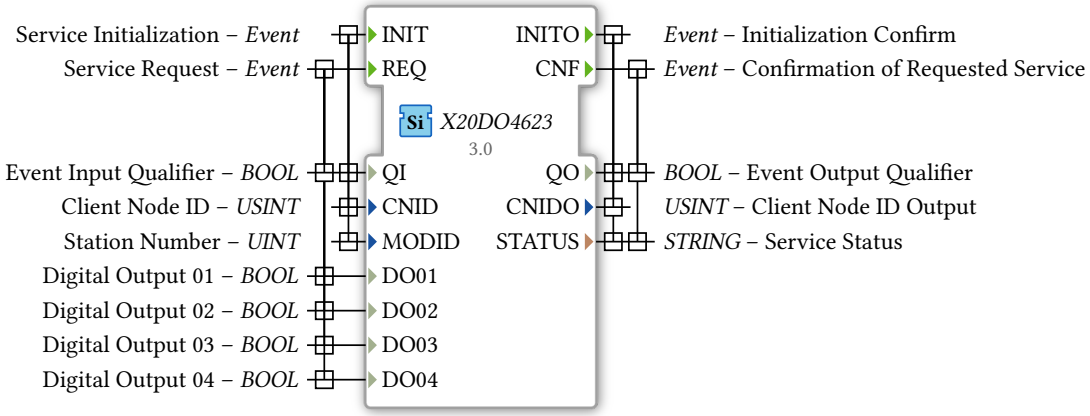
]

2.11.7.10 X20DO4623

• – ◦ • – ◦ • – ◦ •

Introduction

The X20DO4623 function block is used to control digital outputs via the openPOWERLINK fieldbus. It corresponds to the B&R X20DO4623 digital output module (4 channels).



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **REQ**: Write outputs.

Event Outputs

- **INITO**: Initialization confirmed.
- **CNF**: Write operation confirmed.

Data Inputs

- **QI** (BOOL): Qualifier.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.
- **DO01** (BOOL): Digital Output 01.
- **DO02** (BOOL): Digital Output 02.
- **DO03** (BOOL): Digital Output 03.
- **DO04** (BOOL): Digital Output 04.

Data Outputs

- **QO** (BOOL): Qualifier.
- **CNIDO** (USINT): Confirmed Node ID.
- **STATUS** (STRING): Status.

Functionality

Sets the digital outputs on the module according to the input values `DO01-DO04`.

Metadata

Attribute	Value
Copyright	(c) 2011 AIT

License	EPL-2.0
Version	3.0 (April 14, 2025, Patrick Aigner)
4diac package	powerlink

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

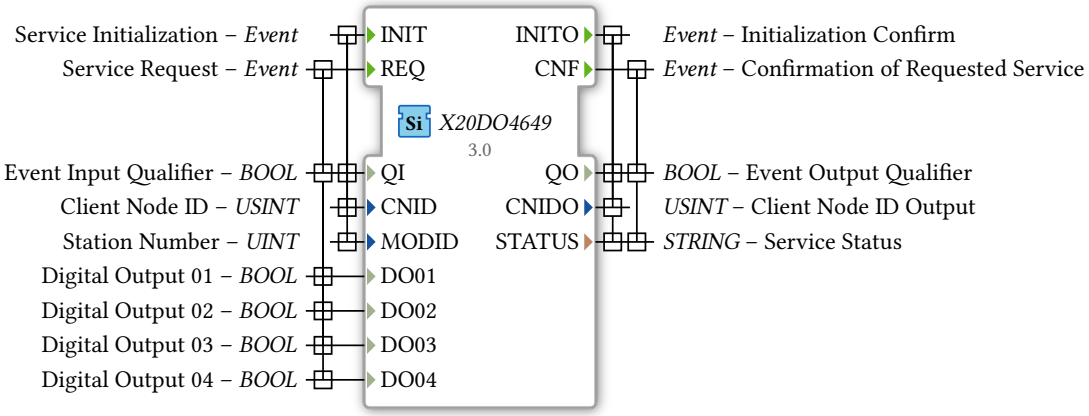
]

2.11.7.11 X20DO4649

• – ◦ • – ◦ • – ◦ •

Introduction

The X20DO4649 function block is used to control digital outputs via the openPOWERLINK fieldbus. It corresponds to the B&R X20DO4649 digital output module (4 channels).



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **REQ**: Write outputs.

Event Outputs

- **INITO**: Initialization confirmed.
- **CNF**: Write operation confirmed.

Data Inputs

- **QI** (BOOL): Qualifier.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.
- **DO01** (BOOL): Digital Output 01.
- **DO02** (BOOL): Digital Output 02.
- **DO03** (BOOL): Digital Output 03.
- **DO04** (BOOL): Digital Output 04.

Data Outputs

- **QO** (BOOL): Qualifier.
- **CNIDO** (USINT): Confirmed Node ID.
- **STATUS** (STRING): Status.

Functionality

Sets the digital outputs on the module according to the input values `DO01-DO04`.

Metadata

Attribute	Value
Copyright	(c) 2011 AIT
License	EPL-2.0
Version	3.0 (April 14, 2025, Patrick Aigner)
4diac package	powerlink

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

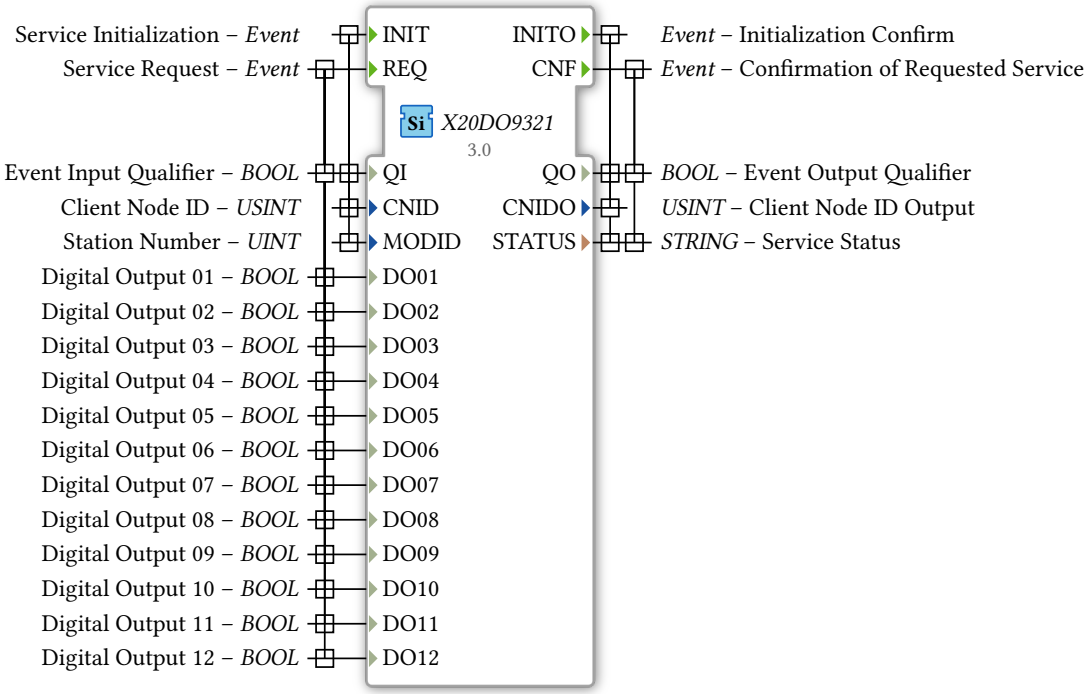
]

2.11.7.12 X20DO9321

• – ◦ • – ◦ • – ◦ •

Introduction

The X20DO9321 function block is used to control digital outputs via the openPOWERLINK fieldbus. It corresponds to the B&R X20DO9321 digital output module (12 channels).



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **REQ**: Write outputs.

Event Outputs

- **INITO**: Initialization confirmed.
- **CNF**: Write operation confirmed.

Data Inputs

- **QI** (BOOL): Qualifier.

- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.
- **DO01** - **DO12** (BOOL): Digital outputs 01 to 12.

Data Outputs

- **QO** (BOOL): Qualifier.
- **CNIDO** (USINT): Confirmed Node ID.
- **STATUS** (STRING): Status.

Functionality

Sets the 12 digital outputs on the module.

Metadata

Attribute	Value
Copyright	(c) 2012 AIT
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac package	powerlink

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

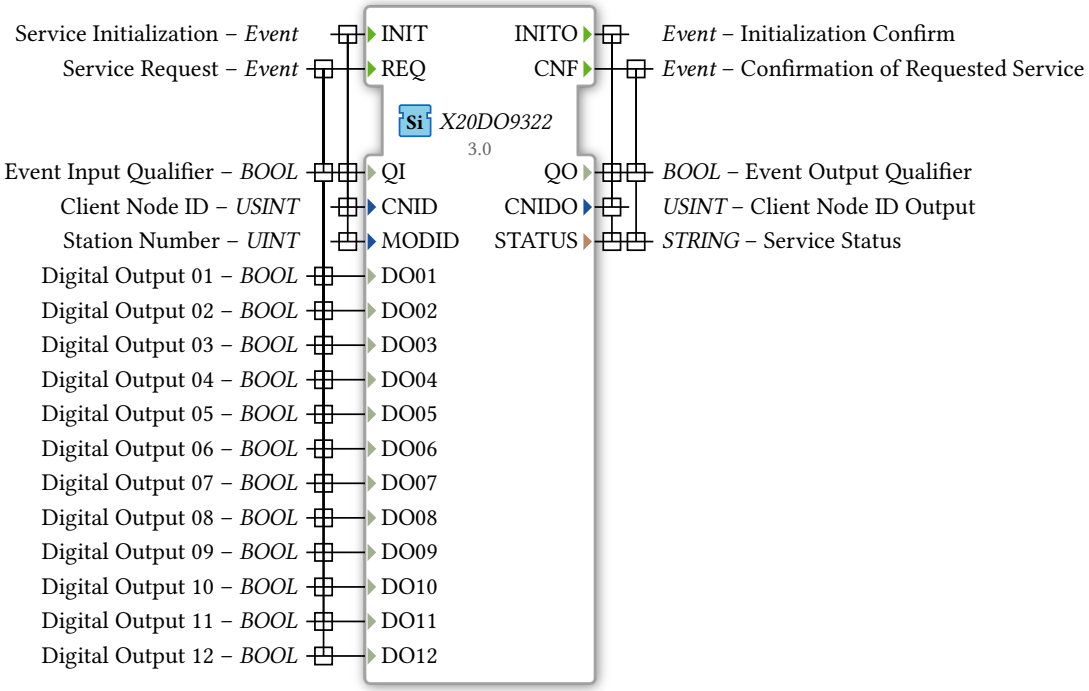
]

2.11.7.13 X20DO9322

• – ◦ • – ◦ • – ◦ •

Introduction

The X20DO9322 function block is used to control digital outputs via the openPOWERLINK fieldbus. It corresponds to the B&R X20DO9322 digital output module (12 channels).



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **REQ**: Write outputs.

Event Outputs

- **INITO**: Initialization confirmed.
- **CNF**: Write operation confirmed.

Data Inputs

- **QI** (BOOL): Qualifier.
- **CNID** (USINT): Client Node ID.
- **MODID** (UINT): Module ID.
- **DO01** - **DO12** (BOOL): Digital outputs 01 to 12.

Data Outputs

- **QO** (BOOL): Qualifier.
- **CNIDO** (USINT): Confirmed Node ID.
- **STATUS** (STRING): Status.

Functionality

Sets the 12 digital outputs on the module.

Metadata

Attribute	Value
Copyright	(c) 2011 AIT
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac package	powerlink

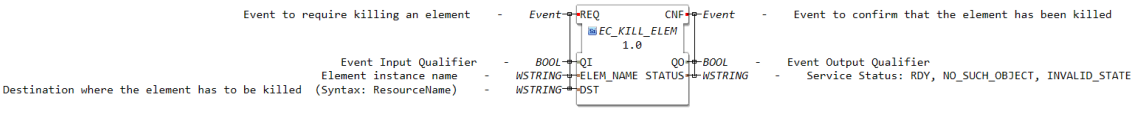
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.8 reconfiguration

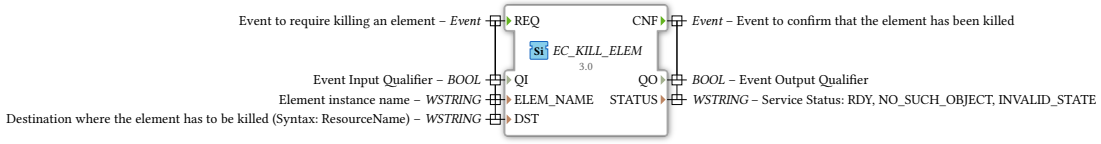
2.11.8.1 EC_KILL_ELEM



• — ○ • — ○ • — ○ •

Introduction

The EC_KILL_ELEM function block is used to terminate instances according to the state machine of IEC 61499 function blocks. It enables the targeted termination of function block instances, connections (event/data), resources, or devices within a 4diac system.



Interface Structure

Event Inputs
- **REQ**: Event to request the termination of an element

Event Outputs
- **CNF**: Event to confirm that the element has been terminated

Data Inputs
- **QI** (BOOL): Event input qualifier
- **ELEM_NAME** (WSTRING): Name of the element instance
- **DST** (WSTRING): Destination where the element must be terminated (Syntax: ResourceName)

Data Outputs
- **QO** (BOOL): Event output qualifier
- **STATUS** (WSTRING): Service status (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapters
No adapter interfaces are available.

Functionality
The function block responds to the REQ event and attempts to terminate the specified element (function block, connection, resource, or device) at the specified target resource. Upon execution, the CNF event is output with the corresponding status.

Technical Features
- Supports terminating various element types (FBs, connections, resources, devices)
- Uses WSTRING data types for element names and targets
- Provides detailed status feedback on the termination process
- Implemented according to IEC 61499 Execution Control Services

Status Overview
The function block has several service sequences:

- **normal_establishment**: Successful initialization
- **unsuccessful_establishment**: Failed initialization
- **request_confirm**: Successful termination request
- **request_inhibited**: Suppressed termination request
- **request_error**: Failed termination request
- **application_initiated_termination**: Application-initiated termination
- **resource_initiated_termination**: Resource-initiated termination

Application Scenarios
- Dynamic reconfiguration of automation systems
- Targeted termination of faulty components
- Resource management in distributed systems
- System maintenance and -Updates
- Error Handling and System Recovery

⚖️ Comparison with Similar Blocks
Compared to other execution control blocks, EC_KILL_ELEM offers specific functions for terminating element instances, while similar blocks often provide creation or management functions. Its ability to handle various element types (FBs, connections, resources, devices) makes it particularly versatile.

Conclusion
EC_KILL_ELEM is an essential building block for reconfiguration tasks in IEC 61499-based systems. Its ability to selectively terminate elements enables dynamic system adjustments and robust error handling strategies in industrial automation solutions.

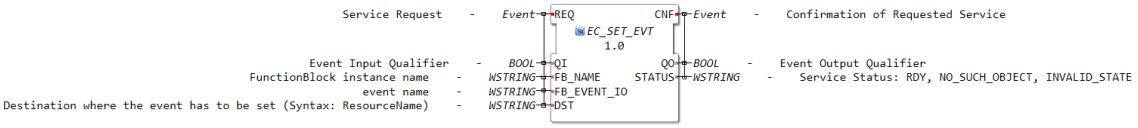
–

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

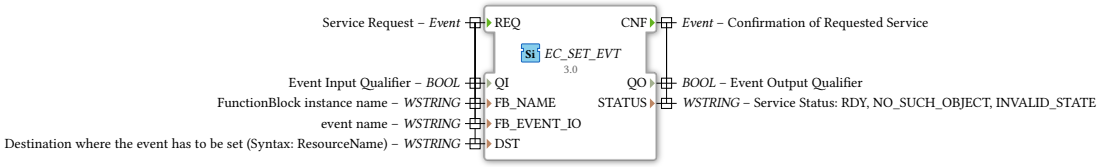
2.11.8.2 EC_SET_EVT



• — ○ • — ○ • — ○ •

Introduction

The EC_SET_EVT function block is used to set event inputs or outputs of a function block instance. It enables the targeted triggering of events in other function blocks and can be used, for example, as an event sniffer. This function block belongs to the category of reconfiguration services according to the IEC 61499-1 standard.



Interface Structure

Event Inputs

- **REQ**: Service Request - triggers the execution of the setting operation

Event Outputs

- **CNF**: Acknowledgement of the requested service operation

Data Inputs

- **QI** (BOOL): Event Input Qualifier - enables/disables the service
- **FB_NAME** (WSTRING): Name of the function block instance
- **FB_EVENT_IO** (WSTRING): Name of the event to be set
- **DST** (WSTRING): Destination where the event is to be set (Syntax: ResourceName)

Data Outputs

- **QO** (BOOL): Event Output Qualifier - indicates the status of the service execution
- **STATUS** (WSTRING): Service status (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapter

No adapter interfaces are available.

Functionality

The EC_SET_EVT function block receives a request via the REQ event to set a specific event in another function block instance. The input data specifies the target function block instance (FB_NAME), the event to be set (FB_EVENT_IO), and the destination (DST). After successful execution, the CNF event is returned with the corresponding status.

…

Technical Features

- Implementation in C++ for GCC compiler version 3.0 and later
- Supports wide-string data types for international character sets
- Provides detailed status feedback for troubleshooting
- Operates as a reconfiguration service between the application and the resource

State Overview

The function block supports several service sequences:

- **normal_establishment**: Successful initialization
- **unsuccessful_establishment**: Failed initialization
- **request_confirm**: Successful service execution
- **request_inhibited**: Suppressed service request
- **request_error**: Failed service execution
- **application_initiated_termination**: Application-side termination
- **resource_initiated_termination**: Resource-side termination

Application Scenarios

- Debugging and monitoring of event flows
- Automated test environments
- Dynamic reconfiguration of control systems
- Event-based diagnostic functions
- Simulation of external Events

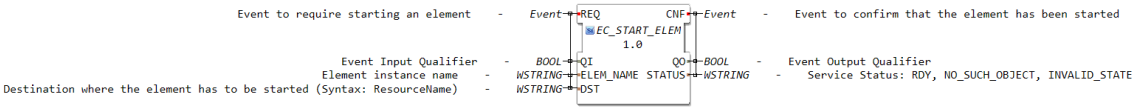
⚖️ Comparison with Similar Function Blocks

Compared to standard function blocks, EC_SET_EVT offers special reconfiguration capabilities that go beyond normal event processing. It allows external access to events in other function blocks, which is particularly valuable for diagnostic and testing purposes.

Conclusion

The EC_SET_EVT function block is a powerful tool for advanced control applications, especially for debugging, testing, and dynamic reconfiguration. Its ability to selectively set events in other function block instances makes it a valuable component for complex automation systems.

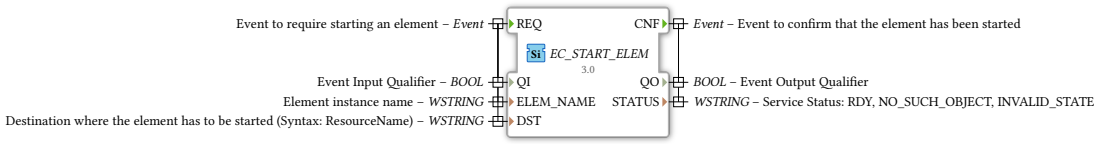
2.11.8.3 EC_START_ELEM



• — ○ • — ○ • — ○ •

Introduction

The EC_START_ELEM function block is used to start instances according to the state machine definition of IEC 61499 function blocks. It enables the targeted starting of function block instances, connections (event/data), resources, or devices within a distributed automation system.



Interface Structure

Event Inputs

- **REQ**: Event to request the start of an element. Associated with the data ELEM_NAME, DST, and QI.

Event Outputs

- **CNF**: Event to confirm that the element has been started. Associated with the data STATUS and QO.

Data Inputs

- **QI** (BOOL): Event input qualifier for controlling the request
- **ELEM_NAME** (WSTRING): Name of the element instance to be started
- **DST** (WSTRING): Destination where the element must be started (Syntax: ResourceName)

Data Outputs

- **QO** (BOOL): Event output qualifier for status feedback
- **STATUS** (WSTRING): Service status with possible values: RDY, NO_SUCH_OBJECT, INVALID_STATE

Adapters

No adapter interfaces are available.

Operation

The function block responds to the REQ event and attempts to start the specified element (function block, connection, resource, or device) at the specified destination. The start process follows the IEC 61499 state machine. Upon successful startup, the CNF event is returned with the status “RDY”. In case of errors, corresponding status messages such as “NO_SUCH_OBJECT” or “INVALID_STATE” are returned.

Technical Features

- Supports wide-string data types for element names and destinations
- Implements a qualifier system to control execution logic
- Provides detailed status feedback for various error states
- Complies with the IEC 61499 standard for execution control services

State Overview

The function block has several service sequences:

- **normal_establishment**: Successful initialization
- **unsuccessful_establishment**: Failed initialization
- **request_confirm**: Successful request processing
- **request_inhibited**: Suppressed request
- **request_error**: Failed request processing
- **application_initiated_termination**: Application-side termination
- **resource_initiated_termination**: Resource-side termination

Application Scenarios

- Dynamically starting function blocks in running systems
- Reconfiguring automation systems at runtime
- Distributed management of resources and devices
- Fault-tolerant systems with restart functionality

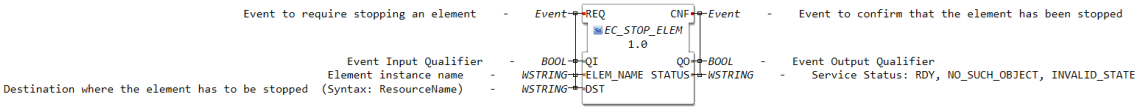
⚖️ Comparison with similar function blocks

Compared to simple start function blocks, EC_START_ELEM offers extended functionality for distributed systems with specific destination specification and detailed status feedback. It is part of the reconfiguration services and supports various element types beyond those of simple function blocks.

Conclusion

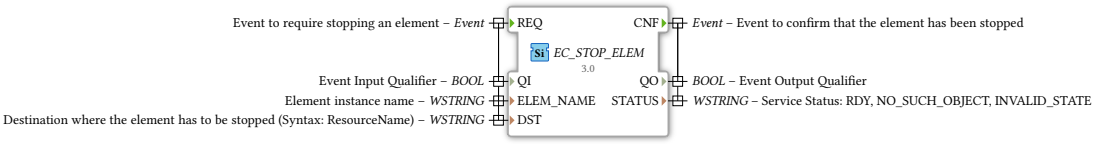
EC_START_ELEM is a powerful function block for the dynamic control of automation system components. Its ability to start various element types at specific destinations makes it particularly valuable for reconfigurable and adaptive control systems according to IEC 61499.

2.11.8.4 EC_STOP_ELEM



Introduction

The EC_STOP_ELEM function block is used to stop instances according to the state machine of IEC 61499 function blocks. It can stop function block instances, connections (event/data), resources, or devices. This block belongs to the category of reconfiguration services and implements an execution control service according to IEC 61499-1.



Interface Structure

Event Inputs

- **REQ**: Event to request that an element be stopped

Event Outputs

- **CNF**: Event to confirm that the element has been stopped

Data Inputs

- **QI** (BOOL): Event input qualifier
- **ELEM_NAME** (WSTRING): Element instance name
- **DST** (WSTRING): Destination where the element must be stopped (Syntax: ResourceName)

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (WSTRING): Service status: RDY, NO_SUCH_OBJECT, INVALID_STATE

Adapters

No adapter interfaces are available.

Functionality

The EC_STOP_ELEM function block processes stop requests for various elements within an IEC 61499 system. Upon receipt of a REQ event with an enabled QI input, the stop process for the specified element is initiated. After completion of the stop process, a CNF event is output with the corresponding status.

Technical Features

- Supports stopping various element types: FBs, connections, resources, and devices
- Uses WSTRING data types for element names and targets
- Provides detailed status feedback on possible error states
- Implemented according to the IEC 61499-1 standard for execution control services

State Overview

The function block has several service sequences:

- **normal_establishment**: Successful initialization
- **unsuccessful_establishment**: Failed initialization
- **request_confirm**: Successful stop request
- **request_inhibited**: Suppressed stop request
- **request_error**: Failed stop request
- **application_initiated_termination**: Application-initiated termination
- **resource_initiated_termination**: Resource-initiated termination

Application Scenarios

- Dynamic reconfiguration of control systems
- Planned shutdown of system components
- Error handling Targeted stopping of faulty components
- Maintenance work on running systems

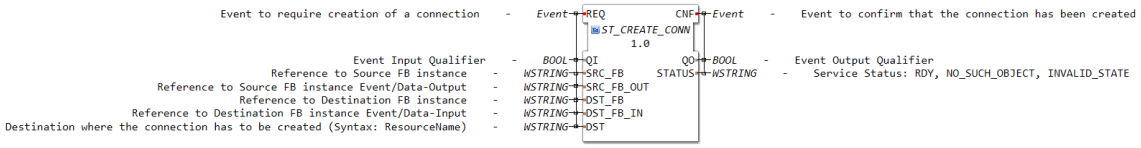
⚖️ Comparison with similar function blocks

Compared to other reconfiguration function blocks, EC_STOP_ELEM offers a specialized function for the targeted stopping of individual system components. While general reconfiguration function blocks often support more complex operations, this function block focuses on the specific task of controlled stopping.

Conclusion

EC_STOP_ELEM is an essential function block for dynamic control systems that require runtime reconfiguration. Its clear interface and reliable status feedback make it a valuable tool for developing maintainable and flexible automation solutions according to IEC 61499.

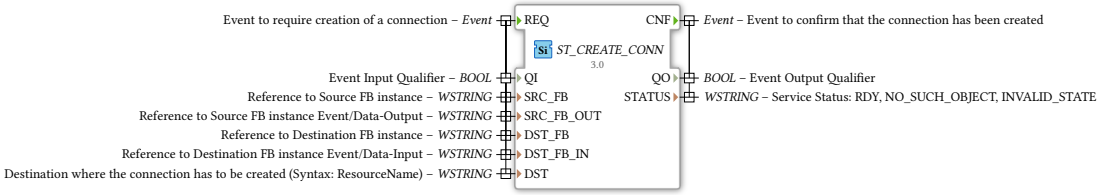
2.11.8.5 ST_CREATE_CONN



• — ○ • — ○ • — ○ •

Introduction

The ST_CREATE_CONN function block is used to create event or data connections between two function block instances at runtime. This component enables the dynamic reconfiguration of connections in a 4diac system and is classified as a structural service.



Interface Structure

Event Inputs

- **REQ**: Event requesting the creation of a connection

Event Outputs

- **CNF**: Event confirming that the connection has been created

Data Inputs

- **QI** (BOOL): Event input qualifier
- **SRC_FB** (WSTRING): Reference to the source FB instance
- **SRC_FB_OUT** (WSTRING): Reference to the event/data output of the source FB instance
- **DST_FB** (WSTRING): Reference to the target FB instance
- **DST_FB_IN** (WSTRING): Reference to the event/data input of the target FB instance
- **DST** (WSTRING): Destination where the connection must be created (Syntax: ResourceName)

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (WSTRING): Service status (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapter

No adapter interfaces are available.

Functionality

The function block establishes a connection between a source and a target function block instance upon request (REQ event). The connection can be used for both events and data streams. After successful processing, an acknowledgment (CNF event) is issued with the corresponding status.

Technical Features

- Supports dynamic reconfiguration at runtime
- Uses wide-string data types for all references
- Provides detailed status feedback
- Implemented as a structured service according to IEC 61499-1

State Overview

The function block supports several service sequences:

- Normal connection creation
- Failed connection creation
- Request acknowledgment cycle
- Request prevention
- Error handling
- Application- and resource-initiated termination

Application Scenarios

- Dynamic system reconfiguration
- Runtime adaptation of control logic
- Flexible connection management in distributed systems
- Adaptive automation systems

⚖️ Comparison with Similar Function Blocks

Unlike statically configured connections, ST_CREATE_CONN enables the dynamic creation of connections at runtime. Other reconfiguration function blocks typically focus on creating or deleting FB instances, while this function block is specifically optimized for connection management.

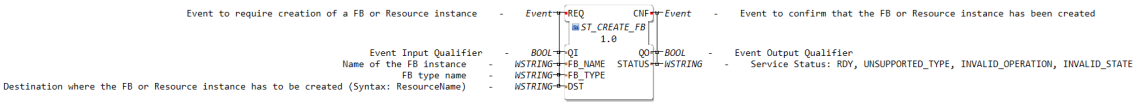
Conclusion

ST_CREATE_CONN is an essential building block for systems requiring dynamic reconfiguration. Its ability to create connections between function blocks at runtime makes it particularly valuable for adaptive and flexible automation solutions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

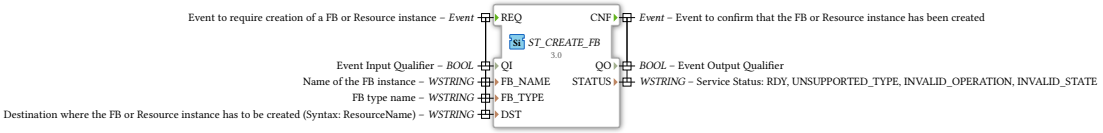
2.11.8.6 ST_CREATE_FB



• — ○ • — ○ • — ○ •

Introduction

The ST_CREATE_FB function block is used to dynamically create function block or resource instances during the runtime of an IEC 61499 system. This block enables the reconfiguration of automation systems by generating new function blocks based on specific FB types.



Interface Structure

Event Inputs

- **REQ**: Event requesting the creation of a function block (FB) or resource instance

Event Outputs

- **CNF**: Event confirming that the FB or resource instance has been created

Data Inputs

- **QI** (BOOL): Event input qualifier
- **FB_NAME** (WSTRING): Name of the FB instance to be created
- **FB_TYPE** (WSTRING): FB type name

- **DST** (WSTRING): Destination where the FB or resource instance should be created (Syntax: Resource-Name)

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (WSTRING): Service status (RDY, UNSUPPORTED_TYPE, INVALID_OPERATION, INVALID_STATE)

Adapter

No adapter interfaces are available.

Functionality

The ST_CREATE_FB function block receives a request to create a new function block instance via the REQ event. The required parameters (FB_NAME, FB_TYPE, DST) are included in the request. After successful processing of the request, the function block returns a confirmation via the CNF event, which contains the status of the operation in the STATUS output.

Technical Features

- Supports dynamic reconfiguration of IEC 61499 systems
- Uses WSTRING data types for international character set support
- Provides detailed status feedback for fault diagnosis
- Implemented as a structural service according to the IEC 61499-1 standard

Status Overview

The function block has several service sequences:

- **normal_establishment**: Successful initialization
- **unsuccessful_establishment**: Failed initialization
- **request_confirm**: Successful request processing
- **request_inhibited**: Suppressed request
- **request_error**: Failed request processing
- **application_initiated_termination**: Application-initiated termination
- **resource_initiated_termination**: Resource-initiated termination

Application Scenarios

- Dynamic expansion of automation systems
- Adaptive control systems that adjust to changing production requirements
- Modular systems Architectures where components are added at runtime
- Reconfigurable manufacturing systems

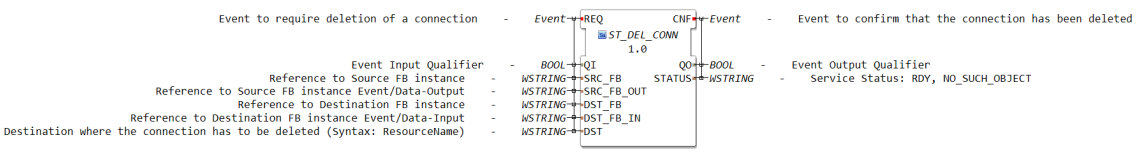
⚖️ Comparison with similar function blocks

Compared to static function blocks, ST_CREATE_FB offers the unique ability to create instances at runtime. While standard function blocks are defined at compile time, this block enables a flexible system architecture.

Conclusion

The ST_CREATE_FB function block is an essential tool for developing dynamic and reconfigurable automation systems according to IEC 61499. Its ability to create function block instances at runtime makes it particularly valuable for applications requiring adaptability and flexibility.

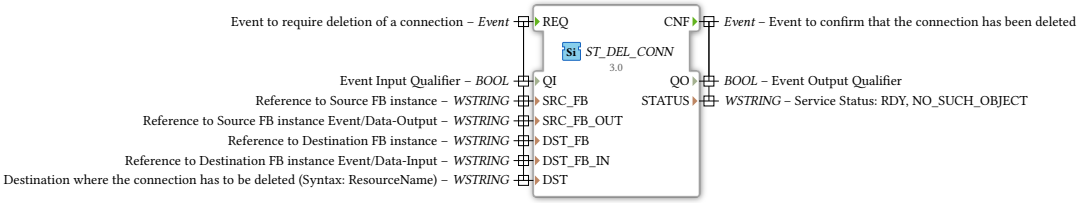
2.11.8.7 ST_DEL_CONN



• — ○ • — ○ • — ○ •

Introduction

The ST_DEL_CONN function block is used to delete event or data connections between two function block instances within a 4diac system. It enables the dynamic reconfiguration of connections at runtime and belongs to the category of structural services according to IEC 61499-1.



Interface Structure

Event Inputs

- **REQ**: Event to request the deletion of a connection

Event Outputs

- **CNF**: Event confirming that the connection has been deleted

Data Inputs

- **QI** (BOOL): Event input qualifier
- **SRC_FB** (WSTRING): Reference to the source FB instance
- **SRC_FB_OUT** (WSTRING): Reference to the event/data output of the source FB instance
- **DST_FB** (WSTRING): Reference to the target FB instance
- **DST_FB_IN** (WSTRING): Reference to the event/data input of the target FB instance
- **DST** (WSTRING): Destination where the connection must be deleted (Syntax: ResourceName)

Data Outputs

- **QO** (BOOL): Event Output Qualifier
- **STATUS** (WSTRING): Service Status: RDY, NO_SUCH_OBJECT

Adapters

No adapter interfaces are available.

Functionality

When the REQ event is triggered, the specified connection between the indicated function block instances is deleted. The block expects complete references for both the source and target function block instances, including their inputs and outputs. After successful processing, the CNF event is output with the corresponding status.

Technical Features

- Supports the deletion of both event and data connections
- Works with wide-string data types for all reference parameters
- Provides status feedback on successful or failed operations
- Implemented according to the IEC 61499-1 standard for structured services

State Overview

The function block supports various service sequences:

- Normal initialization (INIT+ ⟶ INITO+)
- Failed initialization (INIT+ ⟶ INITO-)
- Successful request (REQ+ ⟶ CNF+)
- Suppressed request (REQ- ⟶ CNF-)
- Failed request (REQ+ ⟶ CNF-)
- Application- and resource-side termination

Application Scenarios

- Dynamic reconfiguration of control systems

- Removal of connections at runtime
- Error handling for connection problems
- Adaptive systems that need to adjust to changing requirements

⚖️ Comparison with Similar Blocks

Specialized compared to other reconfiguration blocks ST_DEL_CONN focuses exclusively on deleting connections, while other function blocks may support creating or modifying connections.

Conclusion

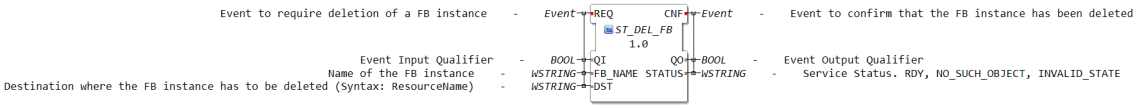
ST_DEL_CONN is an essential function block for systems requiring dynamic reconfiguration. Its clear interface definition and reliable status feedback make it a dependable tool for managing connections in IEC 61499-based control systems.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

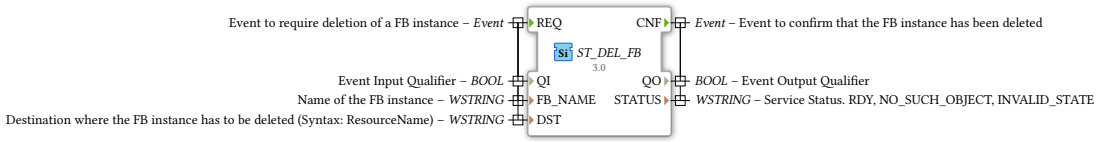
2.11.8.8 ST_DEL_FB



• — ○ • — ○ • — ○ •

Introduction

The ST_DEL_FB function block is used to delete a function block instance of a specific function block type. This function block belongs to the category of reconfiguration services and enables the dynamic management of function block instances at runtime.



Interface Structure

Event Inputs

- **REQ**: Event to request the deletion of a function block instance

Event Outputs

- **CNF**: Event confirming that the function block instance has been deleted

Data Inputs

- **QI** (BOOL): Event input qualifier
- **FB_NAME** (WSTRING): Name of the function block instance to be deleted
- **DST** (WSTRING): Destination where the function block instance must be deleted (Syntax: Resource-Name)

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (WSTRING): Service status (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapter

No adapter interfaces are available.

Functionality

The ST_DEL_FB function block responds to the REQ event and attempts to delete the specified function block instance at the indicated destination. Upon successful execution, a CNF event with the corresponding status is output. The block supports various operating states and error handling.

Technical Features

- Supports dynamic reconfiguration of function blocks
- Uses WSTRING data types for international character sets
- Provides detailed status feedback on the deletion process
- Implemented according to the IEC 61499-1 standard for structured services

State Overview

The function block supports several service sequences:

- Normal Establishment
- Unsuccessful Establishment
- Request Confirmation
- Suppressed Request Inhibited
- Request Error
- Application-Initiated Termination
- Resource-Initiated Termination

Application Scenarios

- Dynamic reconfiguration of control systems
- Resource management in distributed automation systems
- Runtime adaptation of function block architectures
- Maintenance and updates of automation systems

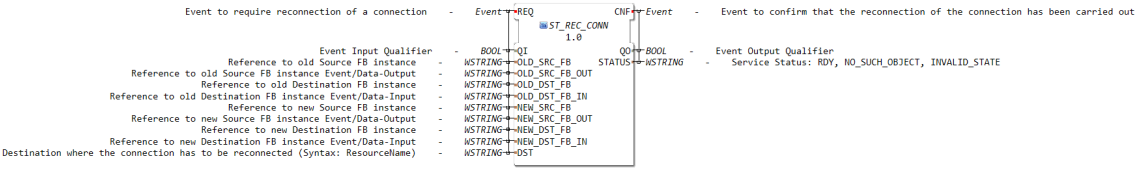
⚖️ Comparison with similar Function Blocks

Compared to static function blocks, ST_DEL_FB enables the dynamic management of function block instances at runtime. It differs from creation blocks through its specific deletion functionality, thus offering complementary functionality for complete lifecycle management.

Conclusion

The ST_DEL_FB function block is an essential tool for systems requiring dynamic reconfiguration. Its robust error handling and detailed status feedback make it a reliable solution for managing function block instances in complex automation environments.

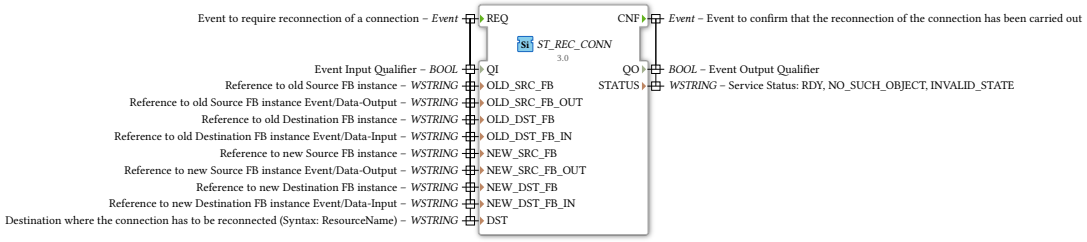
2.11.8.9 ST_REC_CONN



• — ○ • — ○ • — ○ •

Introduction

The ST_REC_CONN function block is used for the dynamic reconfiguration of connections between function blocks in 4diac systems. It enables the redirection of event or data connections from existing source and target function blocks to new instances during runtime.



Interface Structure

Event Inputs

- **REQ**: Starts the reconfiguration process for a connection

Event Outputs

- **CNF**: Confirms whether the connection reconfiguration was successful or failed

Data Inputs

- **QI** (BOOL): Qualifies the event input
- **OLD_SRC_FB** (WSTRING): Reference to the old source FB instance
- **OLD_SRC_FB_OUT** (WSTRING): Reference to the old source FB output (event/data)
- **OLD_DST_FB** (WSTRING): Reference to the old target FB instance
- **OLD_DST_FB_IN** (WSTRING): Reference to the old target FB input (event/data)
- **NEW_SRC_FB** (WSTRING): Reference to the new Source FB Instance
- **NEW_SRC_FB_OUT** (WSTRING): Reference to the new source FB output (event/data)
- **NEW_DST_FB** (WSTRING): Reference to the new target FB instance
- **NEW_DST_FB_IN** (WSTRING): Reference to the new target FB input (event/data)
- **DST** (WSTRING): Destination for reconfiguration (Syntax: ResourceName)

Data Outputs

- **QO** (BOOL): Qualifies the event output
- **STATUS** (WSTRING): Service status (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapters

No adapter interfaces are available.

Functionality

Upon arrival of the REQ event, the function block analyzes the passed parameters and performs the following steps:

1. Identifies the existing connection between OLD_SRC_FB and OLD_DST_FB
2. Disconnects the existing connection
3. Establishes a new connection between NEW_SRC_FB and NEW_DST_FB
4. Provides feedback on the operation status via the CNF event

The QI input controls the execution: If FALSE, the operation is not performed.

Technical Features

- Supports both event and data connections
- Uses WSTRING data types for all reference parameters
- Works with resource-specific target specifications
- Provides detailed status feedback
- Implemented according to the IEC 61499-1 standard for reconfiguration

State Overview

The function block operates according to a simple request-acknowledgment principle:

- Waiting state: No active operation
- Execution state: Processing the REQ request
- Completion state: Sending the CNF acknowledgment

Application Scenarios

- Dynamic system reconfiguration during operation
- Hot-swapping of function blocks
- Load balancing through connection redirection
- Fault-tolerant systems with redundant components
- Adaptive control systems with changing operating modes

⚖️ Comparison with similar blocks

Unlike static connections in 4diac, ST_REC_CONN enables:

- Dynamic adjustment of connections at runtime
- Greater flexibility in Adaptive Systems
- More complex reconfiguration scenarios compared to simple connection blocks

Conclusion

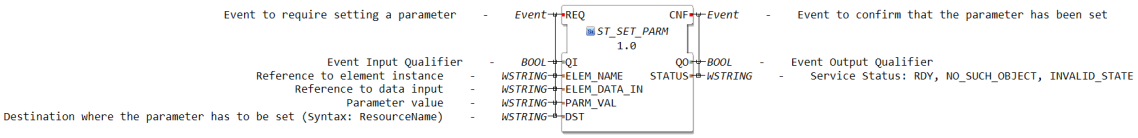
ST_REC_CONN is an essential component for flexible and adaptive automation systems that require dynamic reconfiguration during operation. Its comprehensive parameterization enables precise control of connection changes, while status feedback ensures reliable monitoring of the reconfiguration processes.
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

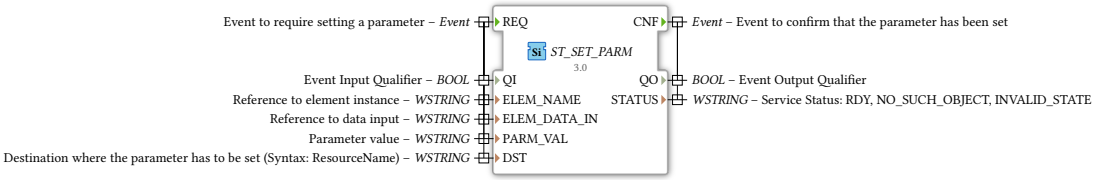
2.11.8.10 ST_SET_PARM



• – ○ • – ○ • – ○ •

Introduction

The ST_SET_PARM function block is used to set parameters for data inputs of function blocks, resources, or device instances within a 4diac system. This function block enables the dynamic configuration of system parameters at runtime.



Interface Structure

Event Inputs

- **REQ**: Starts the process of setting a parameter. Associated with the data ELEM_NAME, ELEM_DATA_IN, PARM_VAL, DST, and QI.

Event Outputs

- **CNF**: Confirms that the parameter was set successfully. Associated with the data STATUS and QO.

Data Inputs

- **QI** (BOOL): Event input qualifier
- **ELEM_NAME** (WSTRING): Reference to element instance
- **ELEM_DATA_IN** (WSTRING): Reference to data input
- **PARM_VAL** (WSTRING): Parameter value

- **DST** (WSTRING): Destination where the parameter must be set (Syntax: ResourceName)

Data Outputs

- **QO** (BOOL): Event output qualifier
- **STATUS** (WSTRING): Service status (RDY, NO_SUCH_OBJECT, INVALID_STATE)

Adapters

No adapters available.
Data Inputs

Functionality

Upon arrival of the REQ event, the ST_SET_PARM function block attempts to set the specified parameter value (PARM_VAL) for the specified data input (ELEM_DATA_IN) of the target instance (ELEM_NAME) at the specified destination (DST). Upon completion of the process, the CNF event is output with the corresponding status.

Technical Features

- Supports wide-string data types for all text-based parameters
- Provides detailed status feedback on the success or failure of the operation
- Enables configuration across resource boundaries
- Implemented as a reconfiguration service according to the IEC 61499-1 standard

State Overview

1. **Idle State**: Waits for a REQ event
2. **Processing State**: Sets the parameter based on the input data
3. **Acknowledgement State**: Sends a CNF event with status information

Application Scenarios

- Dynamic adjustment of system parameters during operation
- Configuration of function block parameters in distributed systems
- Automated parameterization in production plants
- Maintenance and optimization of control systems

⚖️ Comparison with Similar Function Blocks

Compared to static parameterization methods, ST_SET_PARM offers the ability to change parameters at runtime, enabling greater flexibility in adaptive control systems. Other configuration blocks typically only operate during the initialization phase.

Conclusion

The ST_SET_PARM function block is an essential tool for the dynamic reconfiguration of 4diac systems. Its ability to set parameters at runtime makes it particularly valuable for applications requiring adaptability and flexibility. The clear interface definition and detailed status feedback ensure reliable integration into complex automation systems.

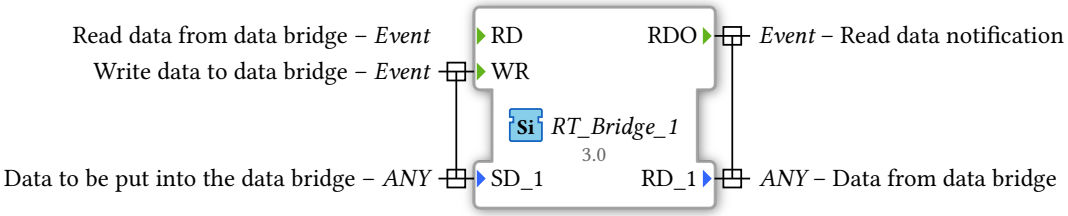
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.9 rtevents

2.11.9.1 RT_Bridge_1



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_1 function block serves as a decoupling bridge for a data connection between real-time event chains. It enables secure data exchange between different execution contexts.

Interface Structure

Event Inputs

- **RD**: Read data from the bridge.
- **WR**: Write data to the bridge.

Event Outputs

- **RDO**: Notification of read data.

Data Inputs

- **SD_1** (ANY): Data to be written to the bridge.

Data Outputs

- **RD_1** (ANY): Data read from the bridge.

Functionality

The module temporarily stores the data received via `WR` and makes it available at the output at `RD`. This decouples the write and read processes in terms of timing.

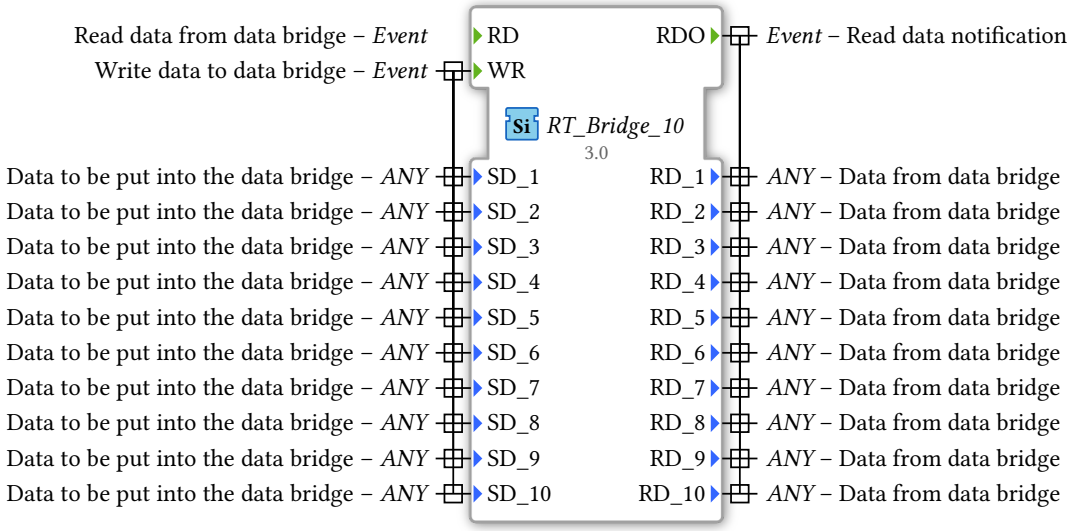
Metadata

Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.2 RT_Bridge_10



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_10 function block serves as a decoupling bridge for 10 data connections between real-time event chains.

Interface Structure

Event Inputs

- **RD**: Read data.
- **WR**: Write data.

Event Outputs

- **RDO**: Read acknowledgment.

Data Inputs

- **SD_1** - **SD_10** (ANY): Input data.

Data Outputs

- **RD_1** - **RD_10** (ANY): Output data.

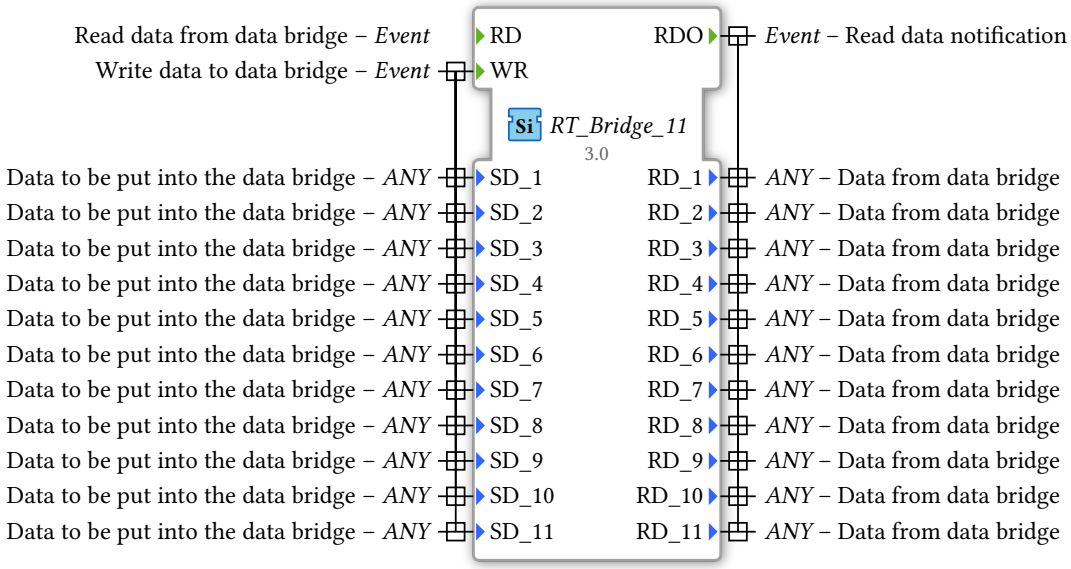
Metadata

Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.11.9.3 RT_Bridge_11



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_11 function block serves as a decoupling bridge for 11 data connections between real-time event chains.

Interface Structure

Event Inputs

- **RD**: Read data.
- **WR**: Write data.

Event Outputs

- **RDO**: Read acknowledgment.

Data Inputs

- **SD_1** - **SD_11** (ANY): Input data.

Data Outputs

- **RD_1** - **RD_11** (ANY): Output data.

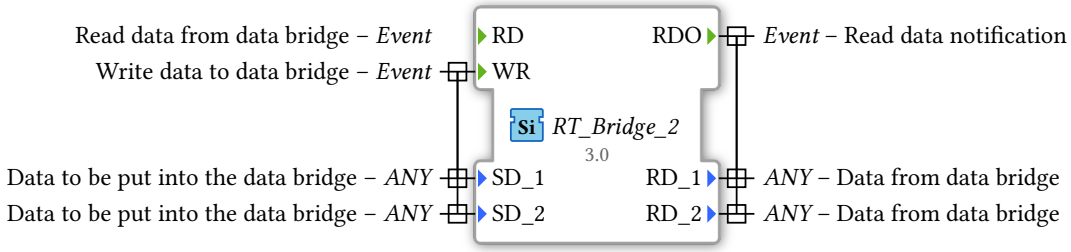
Metadata

Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.11.9.4 RT_Bridge_2



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_2 function block serves as a decoupling bridge for two data connections between real-time event chains.

Interface Structure

Event Inputs

- **RD**: Read data.
- **WR**: Write data.

Event Outputs

- **RDO**: Read acknowledgment.

Data Inputs

- **SD_1** - **SD_2** (ANY): Input data.

Data Outputs

- **RD_1** - **RD_2** (ANY): Output data.

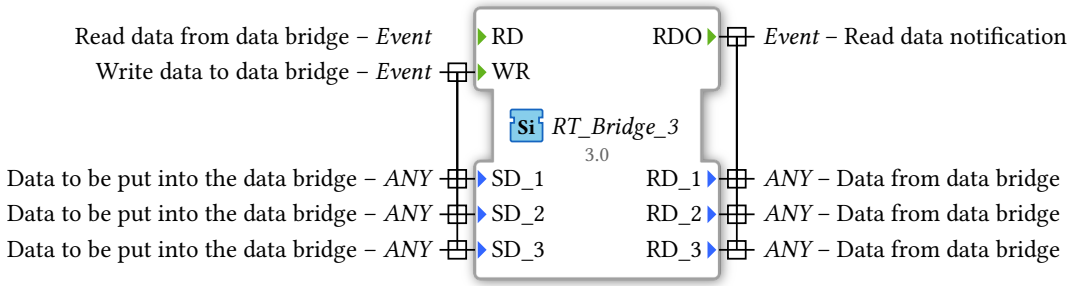
Metadata

Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (April 14, 2025, Patrick Aigner)
4diac package	eclipse4diac::rtevents

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.5 RT_Bridge_3



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_3 function block serves as a decoupling bridge for three data connections between real-time event chains.

Interface Structure

Event Inputs

- **RD**: Read data.
- **WR**: Write data.

Event Outputs

- **RDO**: Read acknowledgment.

Data Inputs

- **SD_1** - **SD_3** (ANY): Input data.

Data Outputs

- **RD_1** - **RD_3** (ANY): Output data.

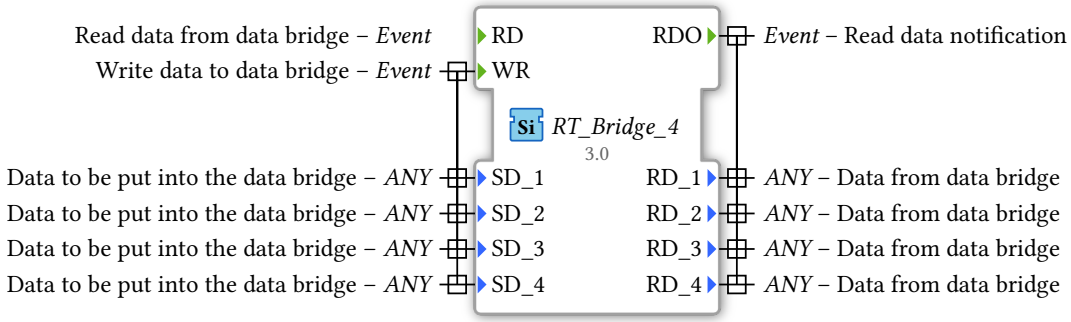
Metadata

Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (April 14, 2025, Patrick Aigner)
4diac package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.6 RT_Bridge_4



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_4 function block serves as a decoupling bridge for four data connections between real-time event chains.

Interface Structure

Event Inputs

- **RD**: Read data.
- **WR**: Write data.

Event Outputs

- **RDO**: Read acknowledgment.

Data Inputs

- **SD_1** - **SD_4** (ANY): Input data.

Data Outputs

- **RD_1** - **RD_4** (ANY): Output data.

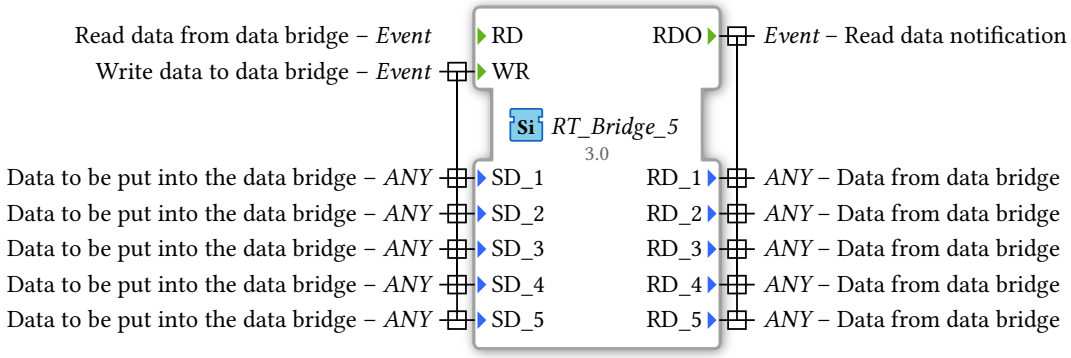
Metadata

Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (April 14, 2025, Patrick Aigner)
4diac package	eclipse4diac::rtevents

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.7 RT_Bridge_5



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_5 function block serves as a decoupling bridge for 5 data connections between real-time event chains.

Interface Structure

Event Inputs

- **RD**: Read data.
- **WR**: Write data.

Event Outputs

- **RDO**: Read acknowledgment.

Data Inputs

- **SD_1** - **SD_5** (ANY): Input data.

Data Outputs

- **RD_1** - **RD_5** (ANY): Output data.

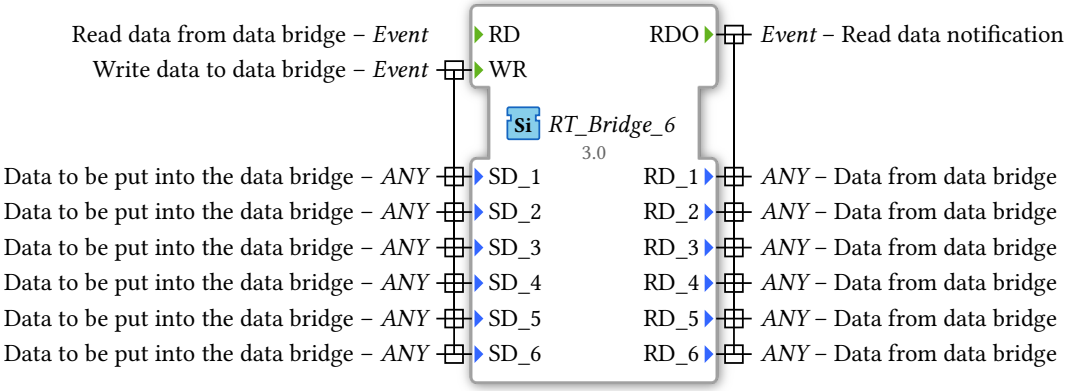
Metadata

Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (April 14, 2025, Patrick Aigner)
4diac package	eclipse4diac::rtevents

🌐 **Related topic subpages on ms-muc-docs.de**

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.8 RT_Bridge_6



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_6 function block serves as a decoupling bridge for six data connections between real-time event chains.

Interface Structure

Event Inputs

- **RD**: Read data.
- **WR**: Write data.

Event Outputs

- **RDO**: Read acknowledgment.

Data Inputs

- **SD_1** - **SD_6** (ANY): Input data.

Data Outputs

- **RD_1** - **RD_6** (ANY): Output data.

Metadata

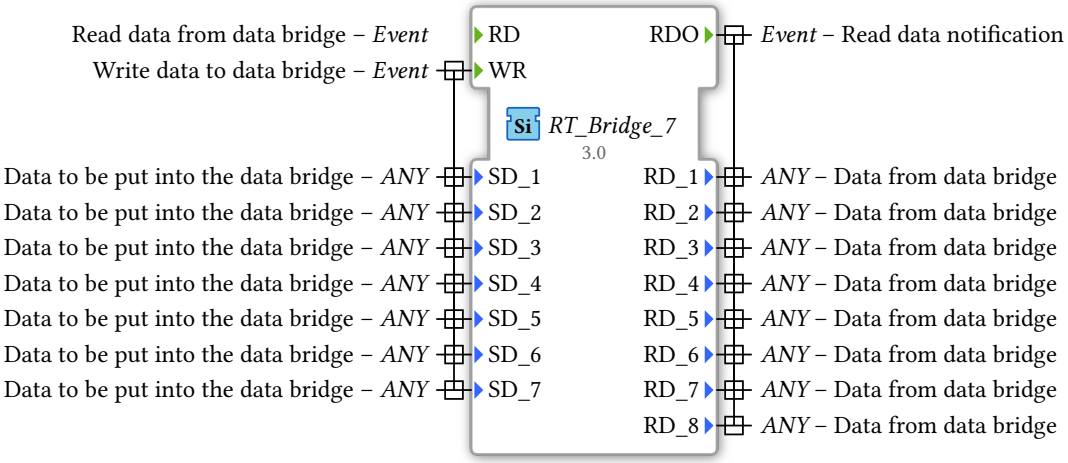
Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (April 14, 2025, Patrick Aigner)

4diac package	eclipse4diac::rtevents

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.9 RT_Bridge_7



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_7 function block serves as a decoupling bridge for 7 data connections between real-time event chains.

Interface Structure

Event Inputs

- **RD**: Read data.
- **WR**: Write data.

Event Outputs

- **RDO**: Read acknowledgment.

Data Inputs

- **SD_1** - **SD_7** (ANY): Input data.

Data Outputs

- **RD_1** - **RD_7** (ANY): Output data.

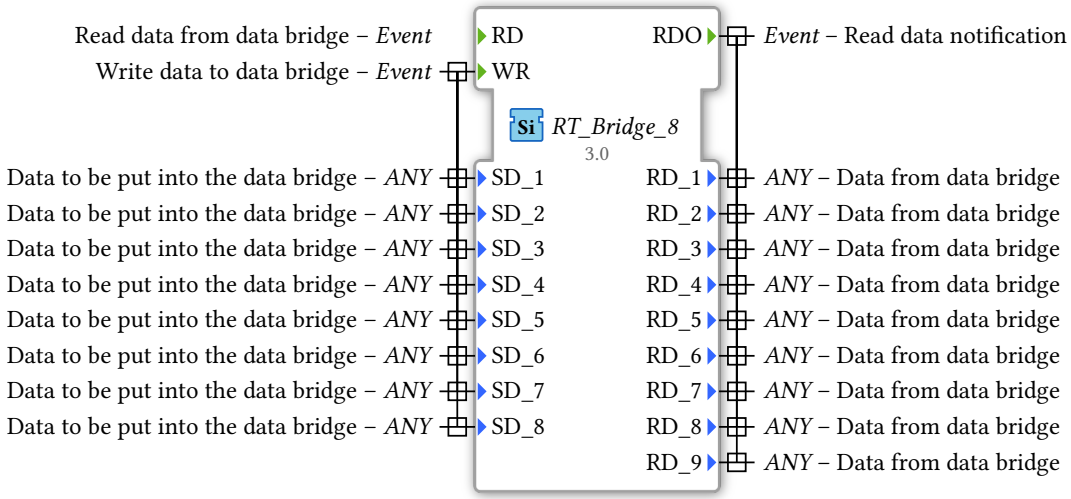
Metadata

Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (April 14, 2025, Patrick Aigner)
4diac package	eclipse4diac::rtevents

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.10 RT_Bridge_8



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_8 function block serves as a decoupling bridge for 8 data connections between real-time event chains.

Interface Structure

Event Inputs

- **RD**: Read data.
- **WR**: Write data.

Event Outputs

- **RDO**: Read acknowledgment.

Data Inputs

- **SD_1** - **SD_8** (ANY): Input data.

Data Outputs

- **RD_1** - **RD_8** (ANY): Output data.

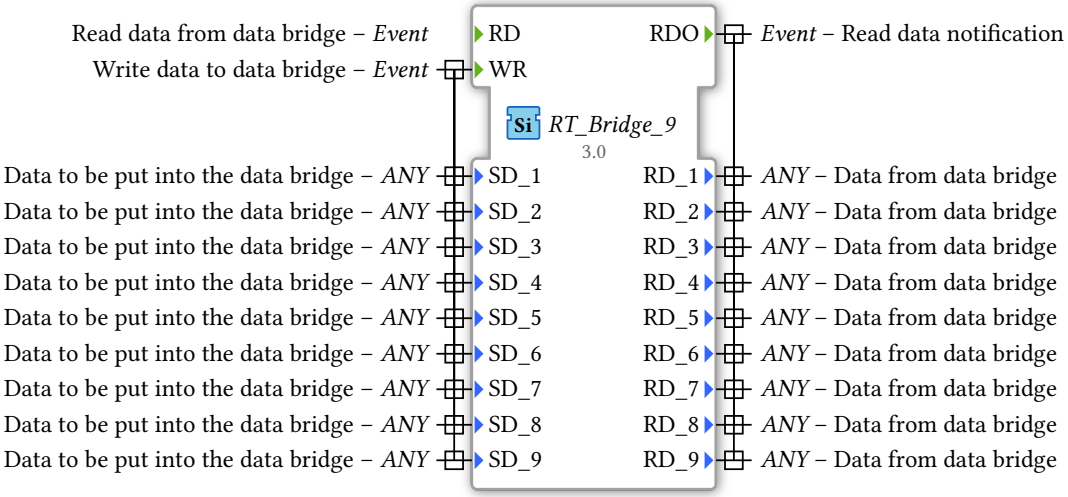
Metadata

Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (April 14, 2025, Patrick Aigner)
4diac package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.11 RT_Bridge_9



• – ◦ • – ◦ • – ◦ •

Introduction

The RT_Bridge_9 function block serves as a decoupling bridge for 9 data connections between real-time event chains.

Interface Structure

Event Inputs

- **RD**: Read data.
- **WR**: Write data.

Event Outputs

- **RDO**: Read acknowledgment.

Data Inputs

- **SD_1** - **SD_9** (ANY): Input data.

Data Outputs

- **RD_1** - **RD_9** (ANY): Output data.

Metadata

Attribute	Value
Copyright	(c) 2023 Johannes Kepler University Linz
License	EPL-2.0
Version	3.0 (April 14, 2025, Patrick Aigner)
4diac package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

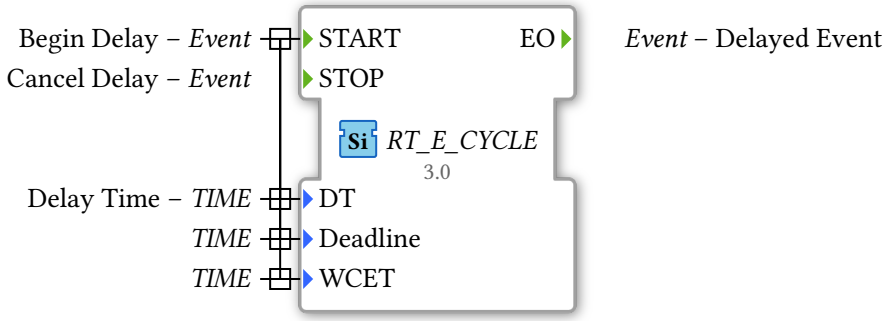
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.12 RT_E_CYCLE

• – ◦ • – ◦ • – ◦ •

Introduction

The RT_E_CYCLE function block implements periodic event generation (cycle) with real-time capabilities. It allows you to define deadlines and execution times (WCET).



Interface Structure

Event Inputs

- **START**: Starts the cycle.
- **STOP**: Stops the cycle.

Event Outputs

- **EO**: The cyclically generated event.

Data Inputs

- **DT** (TIME): Cycle time (delay time).
- **Deadline** (TIME): Deadline for event processing.
- **WCET** (TIME): Worst-case execution time.

Data Outputs

- **QO** (BOOL): Status indicator.

Functionality

After the `START` event, the `EO` event is triggered periodically at time `DT`. The parameters `Deadline` and `WCET` are used for real-time scheduling and monitoring.

Metadata

Attribute	Value
Copyright	(c) 2008 ACIN
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

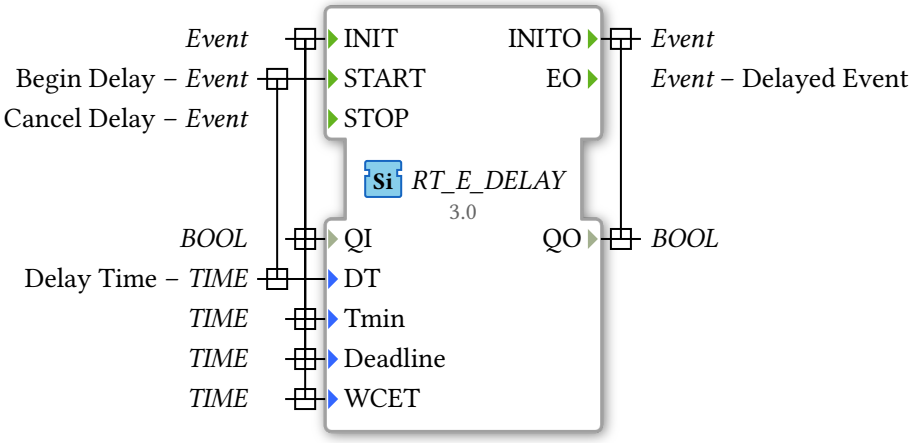
]

2.11.9.13 RT_E_DELAY

• – ◦ • – ◦ • – ◦ •

Introduction

The RT_E_DELAY function block delays an event by a defined time (DT) while adhering to real-time requirements.



Interface Structure

Event Inputs

- **INIT**: Initialization.
- **START**: Starts the delay.
- **STOP**: Stops the delay.

Event Outputs

- **INITO**: Initialization acknowledged.
- **EO**: The delayed event.

Data Inputs

- **QI** (BOOL): Qualifier.
- **DT** (TIME): Delay time.
- **Tmin** (TIME): Minimum time.
- **Deadline** (TIME): Deadline.
- **WCET** (TIME): Worst-case execution time.

Data Outputs

- **QO** (BOOL): Status.

Metadata

Attribute	Value
Copyright	(c) 2008, 2014 ACIN, Profactor GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

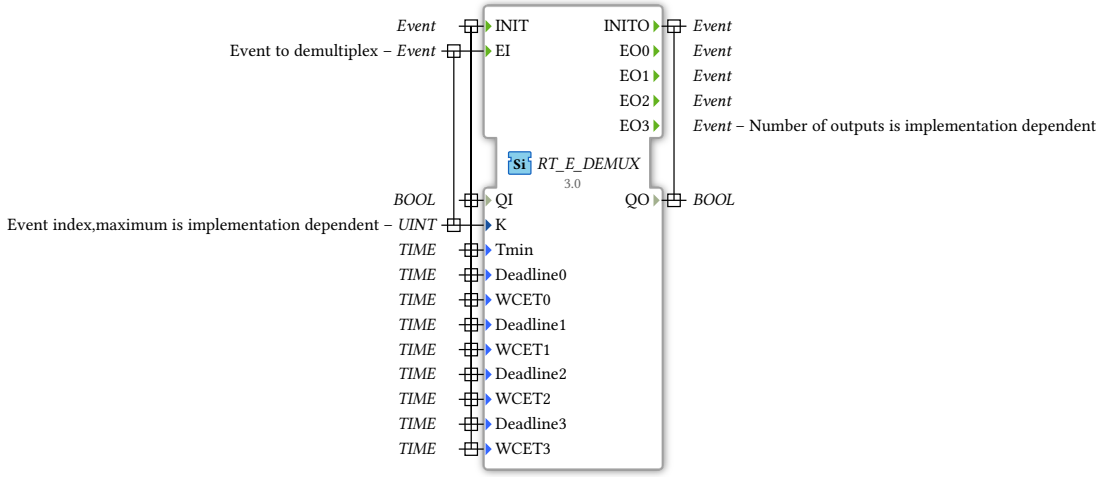
]

2.11.9.14 RT_E_DEMUX

• – ◦ • – ◦ • – ◦ •

Introduction

Real-time version of the E_DEMUX block. Forwards an input event to one of several outputs.



Interface

Event inputs

Event	Associated variables	Description
INIT	QI, Tmin, Deadline0..Deadline3, WCET0..WCET3	Initialization
EI	K	Event to demultiplex

Event outputs

Event	Associated variables	Description
INITO	QO	Confirmation of the initialization
EO0..EO3	-	Output events (the number of outputs is implementation dependent)

Input variables

Name	Type	Description
QI	BOOL	Initialization input
K	UINT	Event index, the maximum is implementation dependent
Tmin	TIME	Timing parameter of the real-time variant (not further commented in the source)
Deadline0..Deadline3	TIME	Timing parameter of the real-time variant (not further commented in the source), per output EO0..EO3
WCET0..WCET3	TIME	Timing parameter of the real-time variant (not further commented in the source), per output EO0..EO3

Output variables

Name	Type	Description
QO	BOOL	Initialization output (with INITO)

Metadata

Attribute	Value

License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

–

🌐 Related topic subpages on ms-muc-docs.de

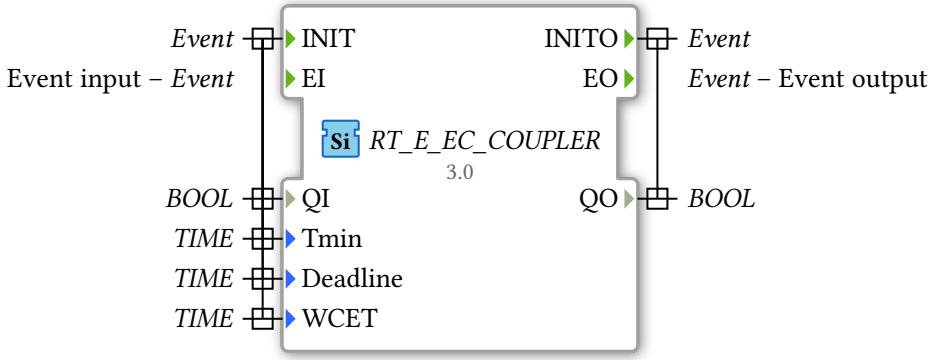
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.15 RT_E_EC_COUPLER

• – ◦ • – ◦ • – ◦ •

Introduction

Couples event chains with real-time parameters.



Interface Structure

Event Inputs

- **INIT**: Initialization, carries `QI`, `Tmin`, `Deadline` and `WCET`.
- **EI**: Event input (start of the coupling); carries no data.

Event Outputs

- **INITO**: Confirms the initialization, carries `QO`.
- **EO**: Event output (continuation of the event chain); carries no data.

Data Inputs

- **QI** (`BOOL`): Qualifier for the initialization.
- **Tmin** (`TIME`): Real-time parameter (minimum distance).
- **Deadline** (`TIME`): Real-time parameter (deadline).
- **WCET** (`TIME`): Real-time parameter (worst-case execution time).

Data Outputs

- **QO** (`BOOL`): Output qualifier of the initialization.

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

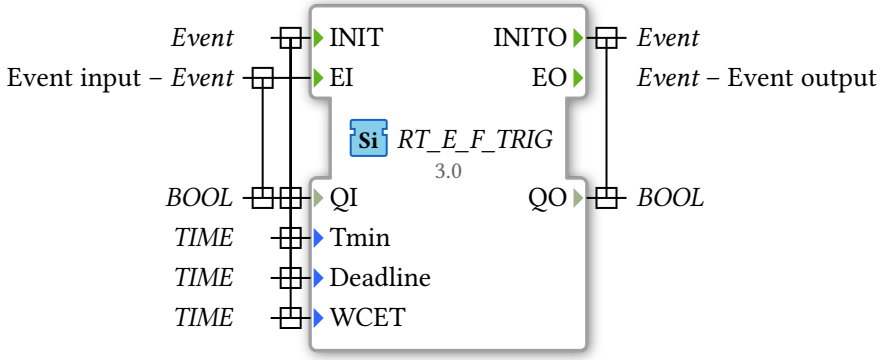
]

2.11.9.16 RT_E_F_TRIG

• – ◦ • – ◦ • – ◦ •

Introduction

Real-time version of the E_F_TRIG block (Falling Edge).



Interface

Port	Type	Description
INIT	Event	Initialization (with QI, Tmin, Deadline, WCET)
EI	Event	Event input (with QI)
INITO	Event	Initialization done (with QO)
EO	Event	Event output
QI	BOOL	Input signal
Tmin	TIME	Real-time parameter (with INIT)
Deadline	TIME	Real-time parameter (with INIT)
WCET	TIME	Real-time parameter (with INIT)
QO	BOOL	Output signal (with INITO)

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

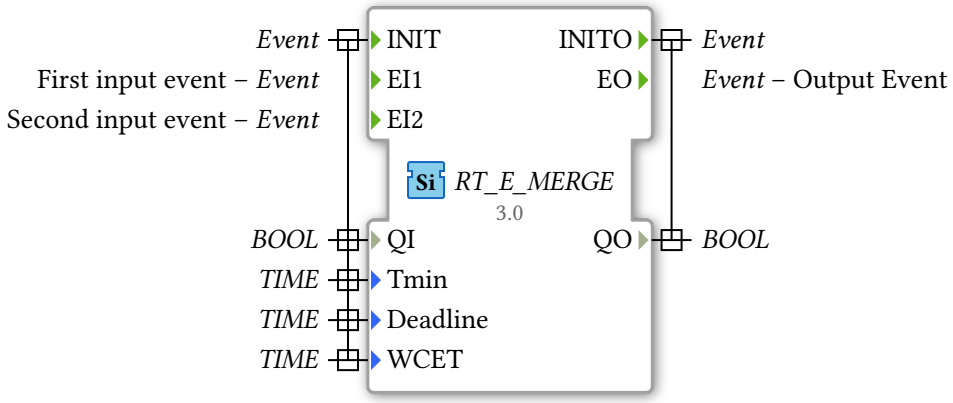
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.17 RT_E_MERGE

• – ◦ • – ◦ • – ◦ •

Introduction

Real-time version of the E_MERGE block. Merges multiple events.



Interface

Event inputs

Event	Associated variables	Description
INIT	QI, Tmin, Deadline, WCET	Initialization
EI1	-	First input event
EI2	-	Second input event

Event outputs

Event	Associated variables	Description
INITO	QO	Confirmation of the initialization
EO	-	Output event

Input variables

Name	Type	Description
QI	BOOL	Initialization input
Tmin	TIME	Timing parameter of the real-time variant (not further commented in the source)
Deadline	TIME	Timing parameter of the real-time variant (not further commented in the source)
WCET	TIME	Timing parameter of the real-time variant (not further commented in the source)

Output variables

Name	Type	Description
QO	BOOL	Initialization output (with INITO)

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

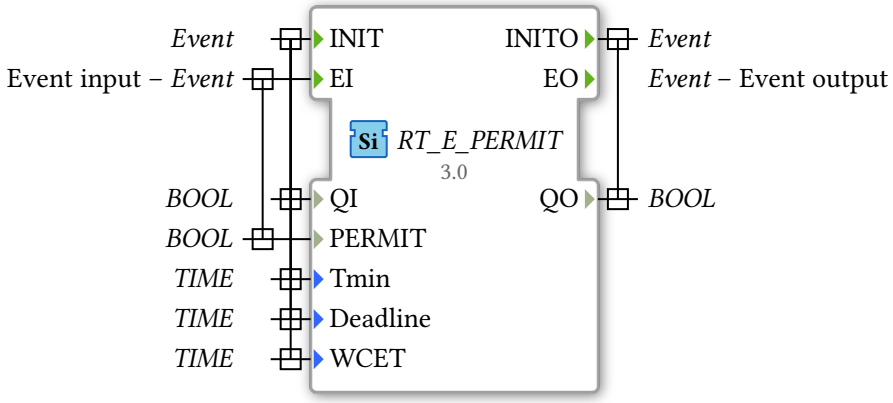
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.18 RT_E_PERMIT

• – ◦ • – ◦ • – ◦ •

Introduction

Real-time version of the E_PERMIT block. Allows an event only if the condition (PERMIT) is true.



Interface Structure

Event Inputs

- **INIT**: Initialization, carries `QI`, `Tmin`, `Deadline` and `WCET`.
- **EI**: Event input, carries `PERMIT`.

Event Outputs

- **INITO**: Confirms the initialization, carries `QO`.
- **EO**: Event output; carries no data.

Data Inputs

- **QI** (`BOOL`): Qualifier for the initialization.
- **PERMIT** (`BOOL`): Permission condition for propagating the event.
- **Tmin** (`TIME`): Real-time parameter (minimum distance).
- **Deadline** (`TIME`): Real-time parameter (deadline).
- **WCET** (`TIME`): Real-time parameter (worst-case execution time).

Data Outputs

- **QO** (`BOOL`): Output qualifier of the initialization.

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

–

🌐 Related topic subpages on ms-muc-docs.de

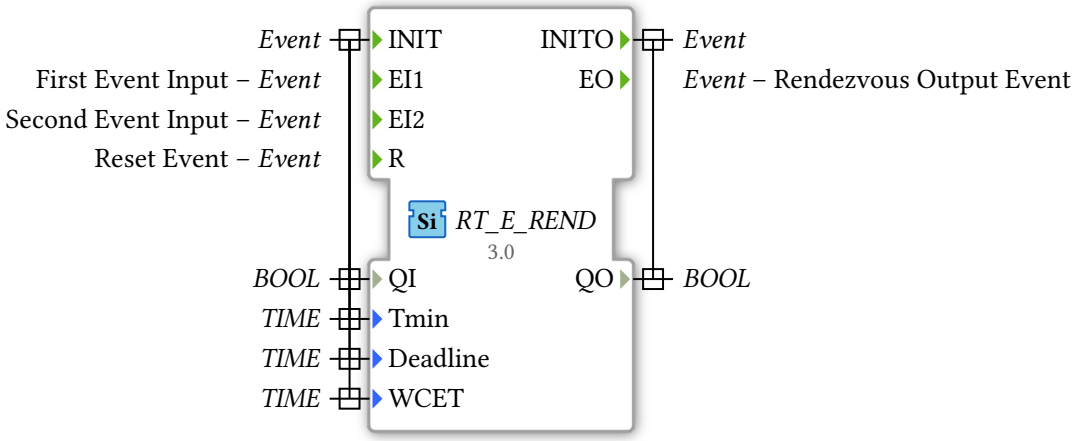
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.19 RT_E_REND

• – ◦ • – ◦ • – ◦ •

Introduction

Real-time version of the E_REND block (Rendezvous). Waits for two events.



Interface

Port	Type	Description
INIT	Event	Initialization (with QI, Tmin, Deadline, WCET)
EI1	Event	First event input
EI2	Event	Second event input
R	Event	Reset event
INITO	Event	Initialization done (with QO)
EO	Event	Rendezvous output event
QI	BOOL	Input (with INIT)
Tmin	TIME	Real-time parameter (with INIT)
Deadline	TIME	Real-time parameter (with INIT)
WCET	TIME	Real-time parameter (with INIT)
QO	BOOL	Output (with INITO)

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

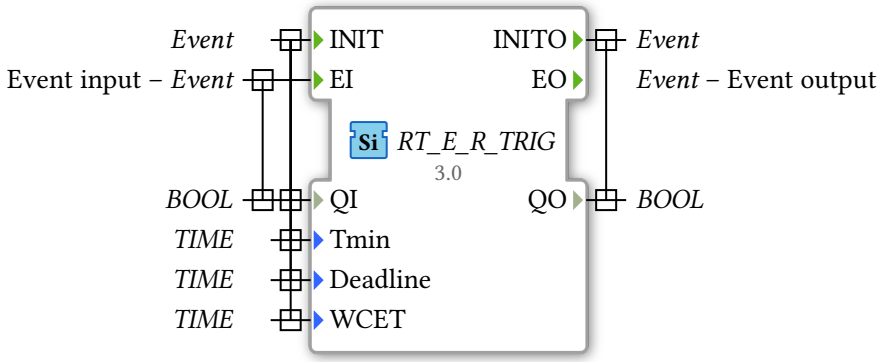
]

2.11.9.20 RT_E_R_TRIG

• – ◦ • – ◦ • – ◦ •

Introduction

Real-time variant of the E_R_TRIG function block (rising edge).



Interface

Event inputs

Event	Associated variables	Description
INIT	QI, Tmin, Deadline, WCET	Initialization
EI	QI	Event input

Event outputs

Event	Associated variables	Description
INITO	QO	Confirmation of the initialization
EO	-	Event output

Input variables

Name	Type	Description
QI	BOOL	Input (with INIT: initialization; with EI: value to be evaluated)
Tmin	TIME	Timing parameter of the real-time variant (not further commented in the source)
Deadline	TIME	Timing parameter of the real-time variant (not further commented in the source)
WCET	TIME	Timing parameter of the real-time variant (not further commented in the source)

Output variables

Name	Type	Description
QO	BOOL	Initialization output (with INITO)

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac package	eclipse4diac::rtevents

Related topic subpages on ms-muc-docs.de

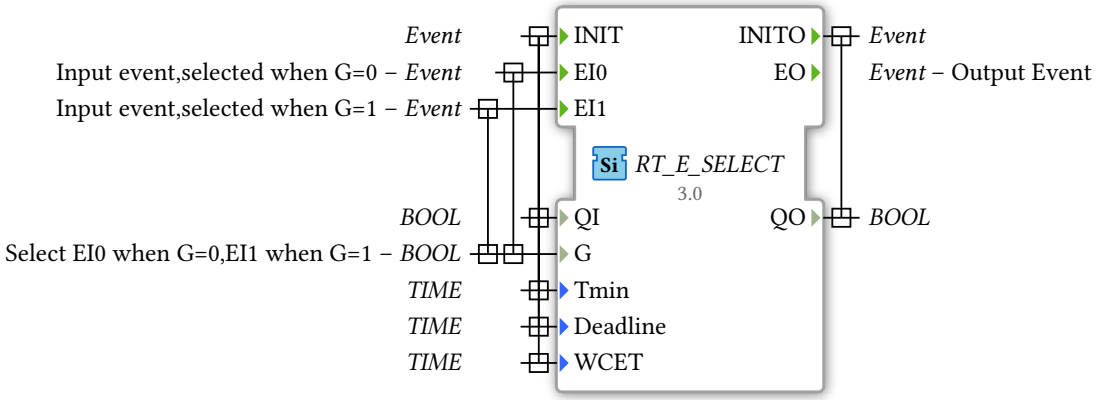
- Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.11.9.21 RT_E_SELECT

• – ○ • – ○ • – ○ •

Introduction

Real-time version of the E_SELECT block. Selects one of two input events based on a selector.



Interface Structure

Event Inputs

- **INIT**: Initialization, carries `QI`, `Tmin`, `Deadline` and `WCET`.
- **EI0**: Input event, selected when `G` = 0; carries `G`.
- **EI1**: Input event, selected when `G` = 1; carries `G`.

Event Outputs

- **INITO**: Confirms the initialization, carries `QO`.
- **EO**: Output event; carries no data.

Data Inputs

- **QI** (`BOOL`): Qualifier for the initialization.
- **G** (`BOOL`): Selector: `EI0` when `G` = 0, `EI1` when `G` = 1.
- **Tmin** (`TIME`): Real-time parameter (minimum distance).
- **Deadline** (`TIME`): Real-time parameter (deadline).
- **WCET** (`TIME`): Real-time parameter (worst-case execution time).

Data Outputs

- **QO** (`BOOL`): Output qualifier of the initialization.

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

–

Related topic subpages on ms-muc-docs.de

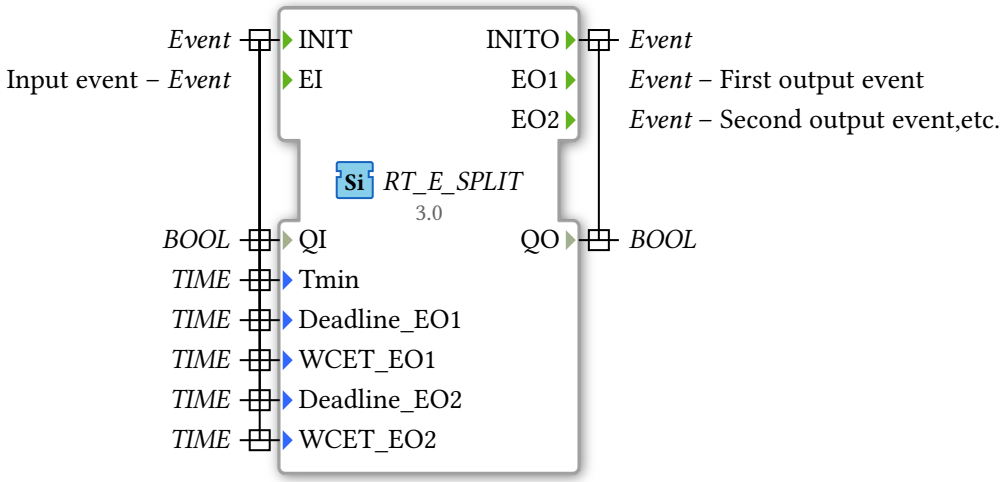
- Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.9.22 RT_E_SPLIT

• – ◦ • – ◦ • – ◦ •

Introduction

Real-time version of the E_SPLIT block. Splits one event into multiple events.



Interface

Port	Type	Description
INIT	Event	Initialization (with QI, Tmin, Deadline_EO1, WCET_EO1, Deadline_EO2, WCET_EO2)
EI	Event	Input event
INITO	Event	Initialization done (with QO)
EO1	Event	First output event
EO2	Event	Second output event
QI	BOOL	Input (with INIT)
Tmin	TIME	Real-time parameter (with INIT)
Deadline_EO1	TIME	Real-time parameter for EO1 (with INIT)
WCET_EO1	TIME	Real-time parameter for EO1 (with INIT)
Deadline_EO2	TIME	Real-time parameter for EO2 (with INIT)
WCET_EO2	TIME	Real-time parameter for EO2 (with INIT)
QO	BOOL	Output (with INITO)

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

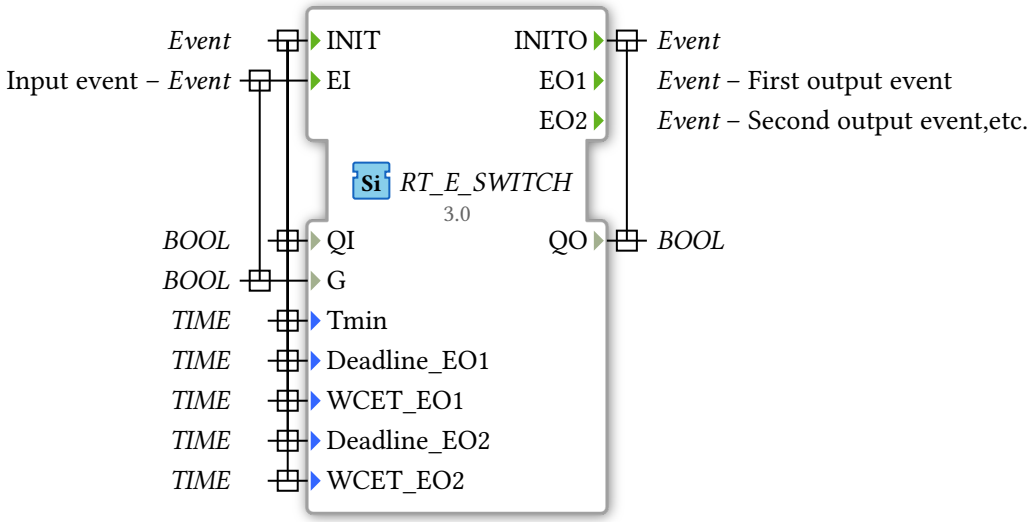
]

2.11.9.23 RT_E_SWITCH

• – ◦ • – ◦ • – ◦ •

Introduction

Real-time version of the E_SWITCH block. Routes the input event EI to one of the two outputs EO1/EO2 depending on G.



Interface

Event inputs

Event	Associated variables	Description
INIT	QI, Tmin, Deadline_EO1, WCET_EO1, Deadline_EO2, WCET_EO2	Initialization
EI	G	Input event

Event outputs

Event	Associated variables	Description
INITO	QO	Confirmation of the initialization
EO1	-	First output event
EO2	-	Second output event

Input variables

Name	Type	Description
QI	BOOL	Initialization input
G	BOOL	Switching condition, associated with EI
Tmin	TIME	Timing parameter of the real-time variant (not further commented in the source)
Deadline_EO1, Deadline_EO2	TIME	Timing parameter of the real-time variant (not further commented in the source), per output
WCET_EO1, WCET_EO2	TIME	Timing parameter of the real-time variant (not further commented in the source), per output

Output variables

Name	Type	Description
QO	BOOL	Initialization output (with INITO)

Metadata

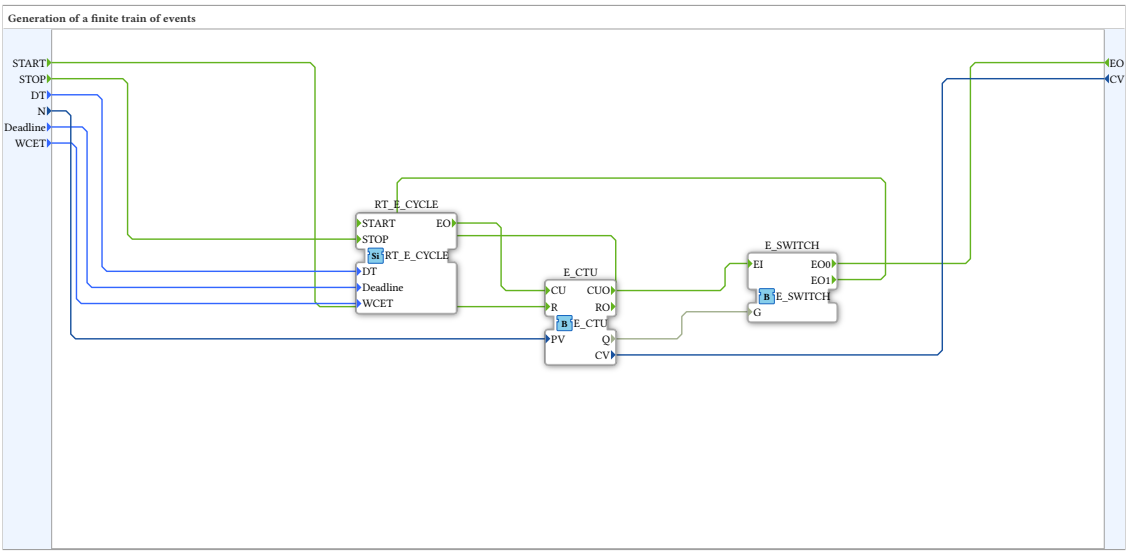
Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

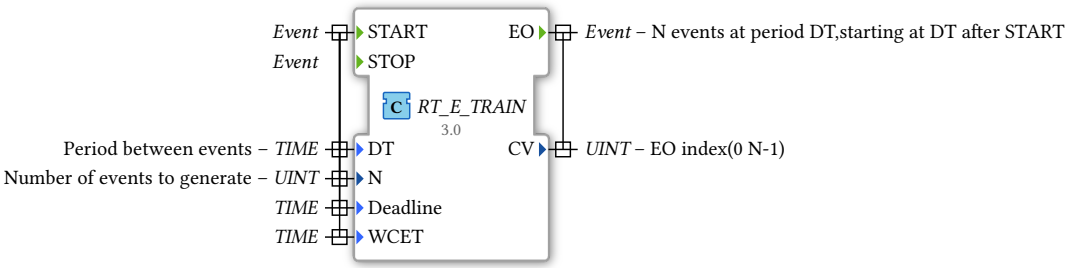
2.11.9.24 RT_E_TRAIN



• – ○ • – ○ • – ○ •

Introduction

Real-time version of the E_TRAIN block. Generates a sequence of events.



Interface Structure

Event Inputs

- **START**: Starts the event train, carries `DT`, `N`, `Deadline` and `WCET`.
- **STOP**: Stops the event train; carries no data.

Event Outputs

- **EO**: `N` events at period `DT`, starting `DT` after `START`; carries `CV`.

Data Inputs

- **DT** (`TIME`): Period between events.
- **N** (`UINT`): Number of events to generate.
- **Deadline** (`TIME`): Real-time parameter (deadline).
- **WCET** (`TIME`): Real-time parameter (worst-case execution time).

Data Outputs

- **CV** (`UINT`): Index of the current `EO` (0 to N-1).

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::rtevents

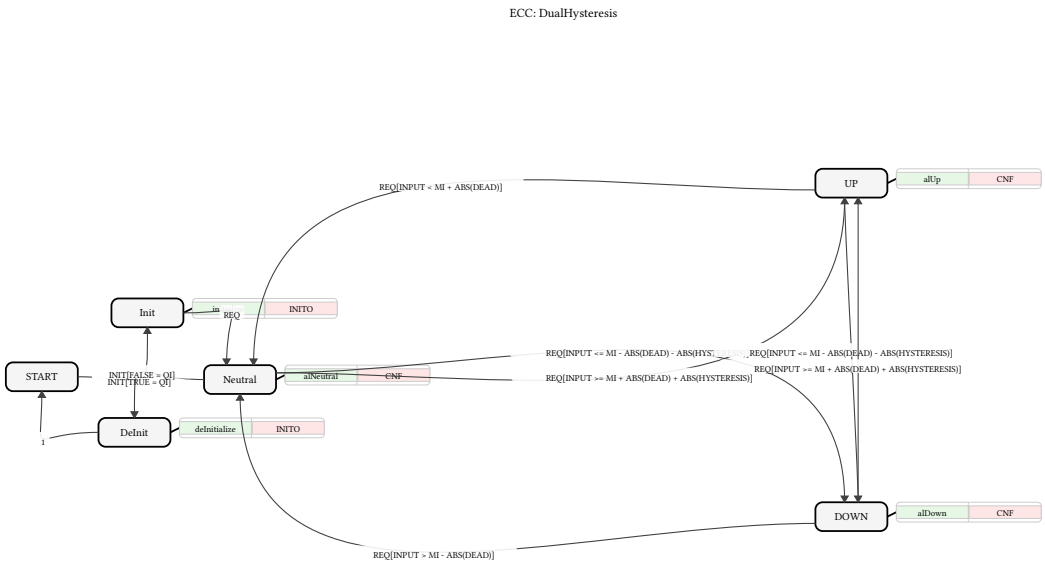
🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.10 signal processing

...

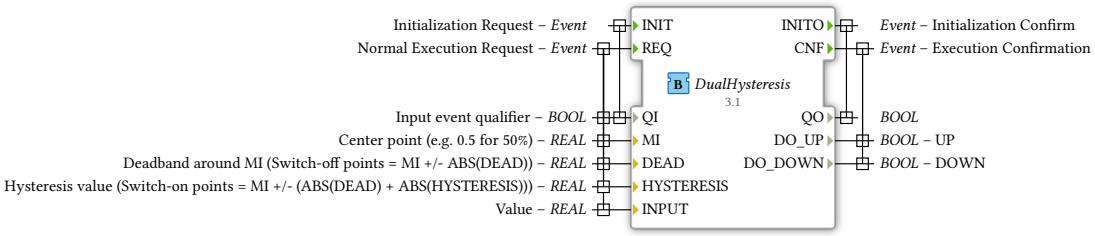
2.11.10.1 DualHysteresis



• – ○ • – ○ • – ○ •

Introduction

The DualHysteresis function block enables bidirectional conversion of analog to digital signals using hysteresis. It is particularly useful in control applications where reliable switching point conversion with deadband and hysteresis is required.



Interface Structure

Event Inputs

- `INIT`: Initialization Request (with `QI`)
- `REQ`: Normal Execution Request (with `QI`, `MI`, `DEAD`, `INPUT`, `HYSTERESIS`)

Event Outputs

- `INITO`: Initialization Acknowledgement (with `QO`)
- `CNF`: Execution Confirmation (with `DO_UP`, `DO_DOWN`)

Data Inputs

- `QI` (BOOL): Input Event Qualifier
- `MI` (REAL, Initial Value: 0.5): Average Setting (e.g., 0.5 for 50%)
- `DEAD` (REAL, Initial Value: 0.1): Dead Band around MI (e.g., 0.1 for 10%)
- `HYSTERESIS` (REAL, Initial Value: 0.1): Hysteresis (e.g., 0.1 for 10%)
- `INPUT` (REAL): Input value

Data Outputs

- `QO` (BOOL): Output event qualifier
- `DO_UP` (BOOL): Upward signal
- `DO_DOWN` (BOOL): Downward signal

Adapters

No adapters available.

Functionality

The function block goes through various states:

1. **Initialization**: Sets the outputs to safe values
2. **Neutral**: Waits for valid input values
3. **UP/DOWN**: Activates the corresponding outputs when the threshold values are exceeded
4. **Deinitialization**: Returns to the safe state

Hysteresis prevents excessive switching for small signal changes around the threshold value.

Technical Features

- Uses a deadband (DEAD) around the mean (MI)
- Implements symmetrical hysteresis (HYSTERESIS)
- All parameters are configurable
- Safe state upon deactivation (QI=FALSE)

State Overview

1. **START → Init** (when INIT with QI=TRUE)
2. **Init → Neutral** (when REQ)
3. **Neutral → UP** (when INPUT ≥ MI + DEAD + HYSTERESIS)
4. **Neutral → DOWN** (when INPUT ≤ MI - DEAD - HYSTERESIS)
5. **UP/DOWN → Neutral** (when returning to the deadband range)

6. **Neutral → DeInit** (when INIT with QI=FALSE)
7. **DeInit → START** (automatic)

Application Scenarios

- Position control with deadband
- Temperature monitoring with hysteresis
- Level detection with different switching points for filling/emptying
- General signal conditioning with noise suppression

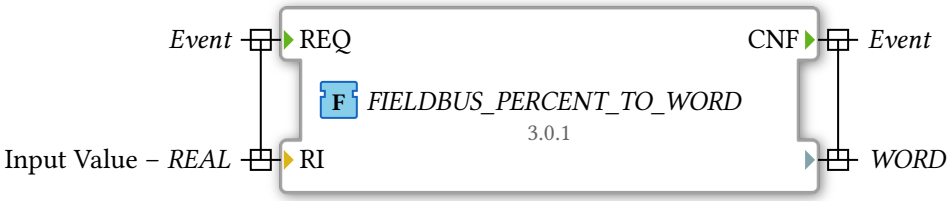
⚖️ **Comparison with similar components**

- Simple hysteresis blocks typically only have one switching output
- Standard comparators do not offer deadband or hysteresis functionality
- This block combines both directions in a single component with configurable parameters

Conclusion

The DualHysteresis block offers a flexible solution for applications requiring reliable signal conversion with deadband and hysteresis in both directions. Its clear state machine and configurable parameters make it particularly versatile.

2.11.10.2 FIELDBUS_PERCENT_TO_WORD



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FIELDBUS_PERCENT_TO_WORD` converts a REAL value in the range of 0.0 to 100.0 into a WORD value in the range of 0 to FAFF (hexadecimal). This function block is particularly suitable for signal processing in fieldbus systems according to the SAE J1939 and ISO 11783 standards.

Interface Structure

Event Inputs

- `REQ`: Trigger event for the conversion. Linked to the input value `RI`.

Event Outputs

- `CNF`: Confirmation event triggered after successful conversion.

Data Inputs

- `RI` (REAL): Input value in the range 0.0 to 100.0 to be converted.

Data Outputs

- (WORD): The converted output value in the range 0 to FAFF (hexadecimal).

Adapters

No adapters available.

Functionality

The function block performs the following steps:

1. Multiplication of the input REAL value (`RI`) by the converted maximum valid signal value (`FIELDBUS_SIGNAL::VALID_SIGNAL_W`).
2. Conversion of the result to a UDINT value.
3. Converting the UDINT value to the final WORD output value.

The conversion is performed using the following formula:
FIELDBUS_PERCENT_TO_WORD = UDINT_TO_WORD(REAL_TO_UDINT(RI * UDINT_TO_REAL(WORD_TO_UDINT(FIELDBUS_SIGNAL::VALID_SIGNAL_W))))

Technical Features

- Uses special type conversion functions (REAL_TO_UDINT, UDINT_TO_WORD, etc.)
- Works with the constant value `FIELDBUS_SIGNAL::VALID_SIGNAL_W` as the maximum limit
- Embedded in the package `eclipse4diac::signalprocessing`

State Overview

1. Wait state: Waits for the `REQ` event
2. Process state: Conversion is performed upon `REQ` input
3. Output state: `CNF` is triggered after successful conversion

Application Scenarios

- Signal processing in agricultural machinery
- Vehicle control systems
- Industrial automation with fieldbus communication
- Process data conversion between different systems

⚖️ Comparison with similar modules

- Specialized to the specific requirements of SAE J1939 and ISO 11783
- Offers more precise control over the value range than generic conversion blocks
- Optimized for use in signal processing

Conclusion

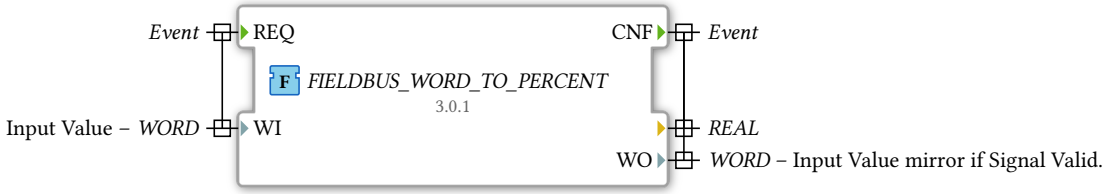
The `FIELDBUS_PERCENT_TO_WORD` function block is a specialized tool for the precise conversion of percentage values into fieldbus-specific WORD values. Its clear interface and efficient implementation make it particularly suitable for applications in mobile machine control and industrial automation.
–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

2.11.10.3 FIELDBUS_WORD_TO_PERCENT



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `FIELDBUS_WORD_TO_PERCENT` converts a REAL value in the range of 0.0 to 100.0 into a WORD value in the range of 0 to FAFF (hexadecimal). This is particularly useful for signal processing in fieldbus systems that operate according to the SAE J1939 and ISO 11783 standards.

Interface Structure

Event Inputs

- **REQ**: Starts the conversion. Linked to the data input `WI`.

Event Outputs

- **CNF**: Confirms the completion of the conversion. Outputs the converted value via the data output `WO`.

Data Inputs

- **WI** (WORD): The input value to be converted.

Data Outputs

- **WO** (WORD): A mirror image of the input value `WI`, if the signal is valid.
- **(REAL)**: The converted REAL value in the range of 0.0 to 100.0.

Functionality

The function block first checks whether the input value `WI` is less than or equal to the defined `VALID_SIGNAL_W`. If so, the value is mirrored (`WO := WI`) and then converted into a REAL value between 0.0 and 100.0. The conversion is performed by dividing the converted WORD value by the maximum valid signal value (`VALID_SIGNAL_W`).

Technical Features

- The block supports the SAE J1939 and ISO 11783 standards.
- Conversion only occurs if the input signal is valid (`WI <= VALID_SIGNAL_W`).
- The output `WO` is used to validate the input signal.

State Overview

1. **Initialization**: The block waits for a `REQ` event.
2. **Processing**: Upon `REQ`, `WI` is checked and converted if necessary.
3. **Output**: Upon successful conversion, `CNF` is triggered and the results are output.

Application Scenarios

- Signal processing in agricultural machinery (ISO 11783).
- Control systems in commercial vehicles (SAE J1939).
- General fieldbus communication where WORD-to-REAL conversions are required.

⚖️ Comparison with Similar Blocks

- Unlike generic conversion blocks, `FIELDBUS_WORD_TO_PERCENT` is specifically optimized for the requirements of SAE J1939 and ISO 11783.
- Other blocks may not offer direct mirroring of the input signal for validation.

Conclusion

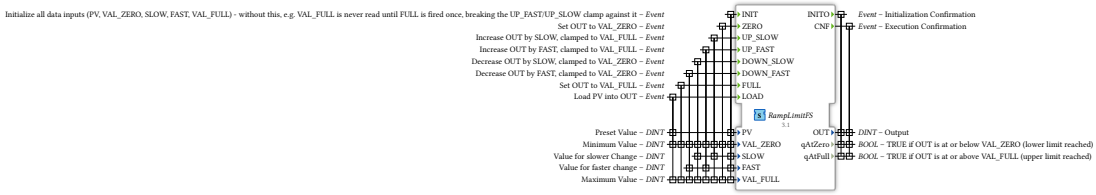
The `FIELDBUS_WORD_TO_PERCENT` block is a specialized converter for signal processing in fieldbus systems. Its simple interface and robust functionality make it ideal for applications that require reliable conversion from WORD to REAL values.

2.11.10.4 RampLimitFS

• — ○ • — ○ • — ○ •

Introduction

The `RampLimitFS` function block incrementally increases or decreases an output value at different speeds (fast/slow), similar to a cruise control function in vehicles. It also allows you to set minimum and maximum values and load a predefined value.



Interface Structure

Event Inputs

- `INIT`: Initializes all data inputs (`PV`, `VAL_ZERO`, `SLOW`, `FAST`, `VAL_FULL`) together and sets `OUT` to `VAL_ZERO`. **Must be fired once before any other event** — without `INIT`, e.g. `VAL_FULL` stays at its default value of 0 until `FULL` itself has fired once, which breaks the clamp in `UP_SLOW`/`UP_FAST` (see Technical Details).

- ZERO: Sets the output OUT to the value of VAL_ZERO.
- UP_SLOW: Increments OUT by the value of SLOW.
- UP_FAST: Increments OUT by the value of FAST.
- DOWN_SLOW: Decrements OUT by the value of SLOW.
- DOWN_FAST: Decrements OUT by the value of FAST.
- FULL: Sets OUT to the value of VAL_FULL.
- LOAD: Loads the value of PV into OUT.

Event Outputs

- INITO: Confirms initialization (response to INIT) and outputs the current value of OUT.
- CNF: Confirms execution and outputs the current value of OUT.

Data Inputs

- PV (DINT): Preset value, loaded with LOAD.
- VAL_ZERO (DINT): Minimum value for OUT.
- SLOW (DINT): Step size for slow changes.
- FAST (DINT): Step size for fast changes.
- VAL_FULL (DINT): Maximum value for OUT.

Data Outputs

- OUT (DINT): Current output value.
- qAtZero (BOOL): TRUE if OUT is at or below VAL_ZERO (lower limit reached).
- qAtFull (BOOL): TRUE if OUT is at or above VAL_FULL (upper limit reached).

Adapters

No adapters available.

Functionality

This function block allows for incremental adjustment of the output value OUT through various events:

- **Incremental Increase/Decrease**:
  - UP_SLOW/DOWN_SLOW: Changes OUT by the SLOW value.
  - UP_FAST/DOWN_FAST: Changes OUT by the FAST value.
- **Direct Value Assignment**:
  - ZERO sets OUT to VAL_ZERO.
  - FULL sets OUT to VAL_FULL.
  - LOAD loads the value of PV into OUT.

The output value is always clamped to the boundaries VAL_ZERO (minimum) and VAL_FULL (maximum).

Technical Details

- **Output Value Limitation**: The value of OUT is automatically clamped to the defined limits (VAL_ZERO, VAL_FULL) if the step sizes (SLOW, FAST) would exceed them.
- **Flexible Step Sizes**: Different rates of change via SLOW and FAST.
- **INIT is mandatory before the first event**: Each event only reads the data inputs it needs for its own calculation, as declared via its WITH list. Since every event recomputes qAtZero/qAtFull at the end, every WITH list carries both VAL_ZERO and VAL_FULL — e.g. UP_SLOW reads SLOW, VAL_FULL (for the actual clamp) **and** VAL_ZERO (only for qAtZero), while DOWN_SLOW reads SLOW, VAL_ZERO (clamp) **and** VAL_FULL (only for qAtFull). Without a preceding INIT call, these values have never been assigned a valid value and sit at their default of 0 — usually unnoticed for VAL_ZERO, but for VAL_FULL every UP_SLOW/UP_FAST step then immediately clamps to 0.
- **Limit detection via qAtZero/qAtFull**: Every event (including INIT) recomputes qAtZero := OUT <= VAL_ZERO and qAtFull := OUT >= VAL_FULL after the actual value change and outputs both alongside INITO/CNF. A calling block therefore doesn't need to check the limits itself - useful e.g. to hide a control when a limit is reached (see ScrollFS/ScrollFS_PHYS_Softkey in isobus::UT::Q, which use exactly this for the scroll arrow buttons).

State Overview

The function block has no internal state. Each event immediately triggers its corresponding calculation and output. The one exception: `INIT` must run once before all other events, so that `VAL_ZERO`/`VAL_FULL`/`SLOW`/`FAST`/`PV` have a defined value in the block at all.

Application Scenarios

- **Cruise Control**: Stepwise speed adjustment with slow/fast key presses.
- **Process Control**: Limited setpoint specification with manual fine-tuning.
- **Parameter Loading**: Loading predefined values (e.g., from non-volatile memory).

⚖️ Comparison with Similar Blocks

- **Comparison with Standard Ramp Functions**: `RampLimitFS` additionally offers the option to switch between fast and slow step sizes and assign direct values (min/max).
- **Flexibility**: Unlike simple counters, this function block allows for dynamic adjustment of step sizes and limits.

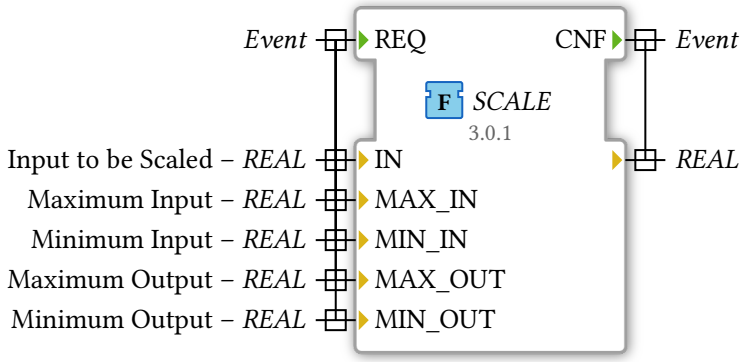
🛠️ Related Exercises

- Exercise_009a

Conclusion

The `RampLimitFS` function block is ideal for applications where a value needs to be adjusted stepwise within defined limits. By differentiating between fast and slow changes and offering the ability to load direct values, it provides a high degree of flexibility for various control tasks.

2.11.10.5 SCALE



• – ◦ • – ◦ • – ◦ •

Introduction

The SCALE function block is used to scale an input value from an input range to an output range. This is particularly useful in signal processing when values from one measurement range need to be transformed into another.

Interface Structure

Event Inputs

- **REQ**: Starts the scaling operation. It is linked to the data inputs IN, MAX_IN, MIN_IN, MAX_OUT, and MIN_OUT.

Event Outputs

- **CNF**: Signals the completion of the scaling operation.

Data Inputs

- **IN**: The input value to be scaled (REAL).
- **MAX_IN**: The maximum value of the input range (REAL).

- **MIN_IN**: The minimum value of the input range (REAL).
- **MAX_OUT**: The maximum value of the output range (REAL).
- **MIN_OUT**: The minimum value of the output range (REAL).

Data Outputs

- **(unnamed)**: The scaled output value (REAL).

Adapters

No adapters available.

Functionality

The SCALE function block calculates the scaled value using the following formula:
SCALE := (IN - MIN_IN) * (MAX_OUT - MIN_OUT) / (MAX_IN - MIN_IN) + MIN_OUT; This formula linearly transforms the input value from the input range [MIN_IN, MAX_IN] to the output range [MIN_OUT, MAX_OUT].

Technical Features

- The function block is implemented in ST (Structured Text).
- The calculation is performed in a single step, ensuring high efficiency.
- No state management is required, as this is a static transformation.

State Overview

The function block has no internal states. The operation is re-executed with each REQ event.

Application Scenarios

- Scaling of sensor values (e.g., from 4-20 mA to 0-100%).
- Transformation of measured values for display or further processing.
- General signal processing in industrial control systems.

⚖️ Comparison with Similar Function Blocks

- Compared to other scaling function blocks, SCALE is particularly easy and straightforward to use.
- No additional parameters or complex settings are required.

Ideal for simple linear scaling without additional logic.

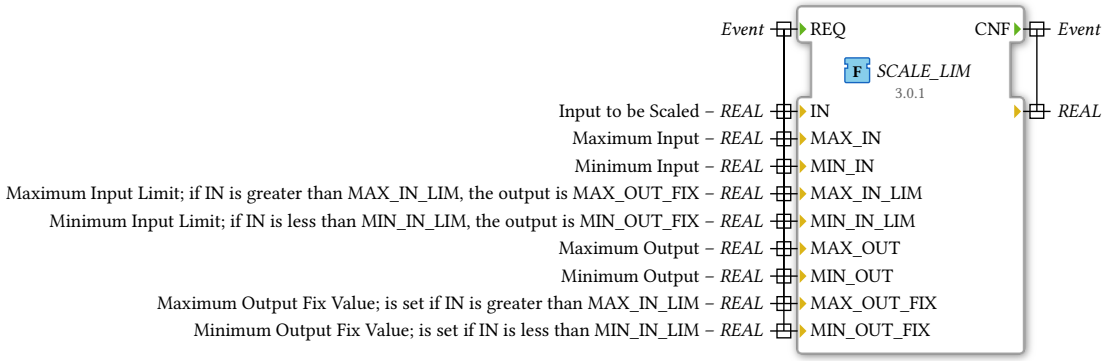
🛠️ Related Exercises

- Exercise_042

Conclusion

The SCALE function block offers an efficient and simple solution for linear scaling tasks. Its clear interface and direct implementation make it a valuable component in signal processing.

2.11.10.6 SCALE_LIM



• – ◦ • – ◦ • – ◦ •

Introduction

The SCALE_LIM function block is a scaling module with additional limiting functions. It enables the linear scaling of an input value within a defined range and also offers the option of fixing output values when certain input limits are exceeded.

Interface Structure

Event Inputs

- **REQ**: Starts the scaling operation. It is linked to all data inputs.

Event Outputs

- **CNF**: Signals the completion of the scaling operation.

Data Inputs

- **IN** (REAL): Input value to be scaled
- **MAX_IN** (REAL): Maximum input value for scaling
- **MIN_IN** (REAL): Minimum input value for scaling
- **MAX_IN_LIM** (REAL): Upper input limit, above which MAX_OUT_FIX is output
- **MIN_IN_LIM** (REAL): Lower input limit, below which MIN_OUT_FIX is output
- **MAX_OUT** (REAL): Maximum output value for scaling
- **MIN_OUT** (REAL): Minimum output value for scaling
- **MAX_OUT_FIX** (REAL): Fixed output value if MAX_IN_LIM is exceeded
- **MIN_OUT_FIX** (REAL): Fixed output value if MIN_IN_LIM is not reached

Data Outputs

- (REAL): Scaled output value

Adapters

No adapters available.

Functionality

The function block performs the following operations:

1. Checks whether the input value IN exceeds the defined limits MIN_IN_LIM or MAX_IN_LIM
2. If IN exceeds MAX_IN_LIM: Outputs MAX_OUT_FIX
3. In normal operating range: Linear scaling of the input value according to the formula:

(IN - MIN_IN) * (MAX_OUT - MIN_OUT) / (MAX_IN - MIN_IN) + MIN_OUT

Technical Features

- Implemented in Structured Text (ST)
- Supports both scaling and limiting
- Part of the `eclipse4diac::signalprocessing` package

State Overview

The function block (FB) has a simple state machine:

1. Waits for a REQ event
2. Performs scaling/limiting
3. Sends a CNF event

Application Scenarios

- Signal processing in control systems
- Scaling of sensor values with limit monitoring
- Process control with defined operating ranges

⚖️ Comparison with Similar Function Blocks

Compared to simple scaling function blocks, SCALE_LIM additionally offers:

- Limit monitoring
- Option to output fixed values when a limit is exceeded

- More flexible definition of scaling ranges

🛠️ Related Exercises

- Exercise_043

Conclusion

SCALE_LIM is a versatile function block for applications that require both scaling and limiting values. Its additional limit functions make it particularly suitable for safety-critical applications where defined responses to limit violations are necessary.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

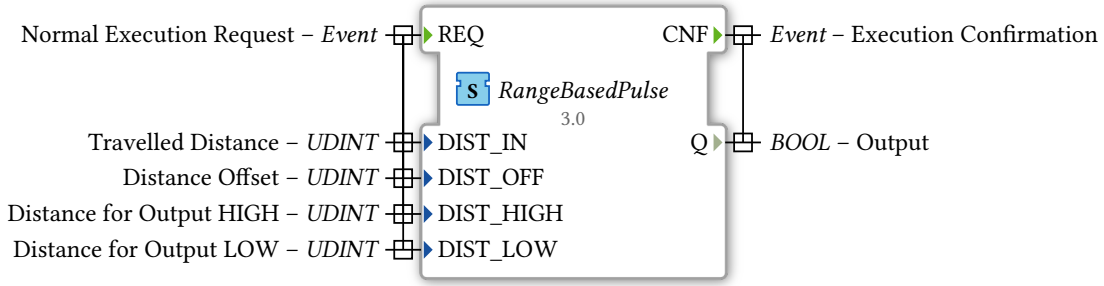
2.11.10.7 distance

RangeBasedPulse

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `RangeBasedPulse` is a distance-based pulse generator that controls a Boolean output (Q) based on the distance traveled (DIST_IN). The function block is started with a HIGH pulse and periodically generates output pulses according to the configured distance values for HIGH and LOW phases.



Interface Structure

Event Inputs

- `REQ`: Normal execution request. Linked to the input data DIST_IN, DIST_OFF, DIST_HIGH, and DIST_LOW.

Event Outputs

- `CNF`: Execution confirmation. Linked to output Q.

Data Inputs

- `DIST_IN` (UDINT): Distance traveled
- `DIST_OFF` (UDINT): Distance offset
- `DIST_HIGH` (UDINT): Distance for HIGH output
- `DIST_LOW` (UDINT): Distance for LOW output

Data Outputs

- `Q` (BOOL): Output signal

Adapters

No adapters included.

Functionality

The function block calculates the remainder (DIST_REMAINDER) of the sum of the distance traveled (DIST_IN) and the offset (DIST_OFF) modulo the sum of DIST_HIGH and DIST_LOW. The output Q is set to HIGH if DIST_REMAINDER is greater than or equal to DIST_HIGH; otherwise, it remains LOW.

Technical Features

- Uses UDINT (unsigned double integer) for distance calculations
- Implements a simple modulo operation for periodic pulse generation
- Supports an offset value for flexible start positions

State Overview

- `REQ`: The only state that executes the algorithm and sends the CNF acknowledgment

Application Scenarios

- Control of machines with distance-based switching operations
- Generation of periodic signals based on distances
- Position-dependent control in vehicle or conveyor systems

⚖️ Comparison with Similar Function Blocks

Unlike simple timers, this function block works with distance values instead of time intervals. It offers greater flexibility through the offset parameter and allows the definition of individual HIGH and LOW phases.

🛠️ Related Exercises

- Exercise_071b
- Exercise_072b

Conclusion

The RangeBasedPulse FB is a specialized function block for applications requiring distance-based control signals. Its simple configuration and flexible offset setting make it well-suited for various industrial control tasks.

2.11.10.8 measurement

2.11.11 system

2.11.11.1 🎧 Podcast

- Eclipse 4diac: Your Bike for Industry 4.0 – Mastering Software Overhead and Controlling Distributed Systems
- IEC 61499: The Future of Industrial Automation and Distributed Systems
- Industry 4.0: The Magic of Self-Configuring Systems with fortiss, IEC 61499, and Eclipse 4diac
- Eclipse 4diac: Revolutionizing Industrial Control with Open-Source Cyber-Physical Systems
- The E_PERMIT Block: The “Gatekeeper” for Events in IEC 61499 Systems Decoded

---

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.11.2 devices

BeagleBoneBlack

• – ◦ • – ◦ • – ◦ •

Introduction

This device represents a FORTE runtime environment on a BeagleBone Black board.

Metadata

Attribute	Value
License	EPL-2.0

Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	iec61499::system

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

FBRT_WINDOW

• – ◦ • – ◦ • – ◦ •

Introduction

This device represents an FBRT (Function Block RunTime) window application.

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	iec61499::system

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

FORTE_PC

• – ◦ • – ◦ • – ◦ •

Introduction

This device represents a FORTE runtime environment on a standard PC (Windows or Linux).

Metadata

Attribute	Value
Copyright	(c) 2015 fortiss GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	iec61499::system

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

PFC200

• – ◦ • – ◦ • – ◦ •

Introduction

This device represents a FORTE runtime environment on a WAGO PFC200 controller.

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	iec61499::system

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

Raspberry Pi

• – ◦ • – ◦ • – ◦ •

Introduction

This device represents a FORTE runtime environment on a Raspberry Pi.

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	iec61499::system

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.11.3 resources

Config_EMB_RES

• – ◦ • – ◦ • – ◦ •

Introduction

A configurable embedded resource for running IEC 61499 applications.

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	iec61499::system

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

EMB_RES

• – ◦ • – ◦ • – ◦ •

Introduction

A basic embedded resource for executing function block networks. By default, it includes an E_RESTART function block for start/stop events.

Metadata

Attribute	Value
Copyright	(c) 2017 fortiss GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac package	iec61499::system

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.11.4 segments

Ethernet

• – ◦ • – ◦ • – ◦ •

Introduction

An Ethernet network segment for connecting devices.

Metadata

Attribute	Value
Copyright	(c) 2008 Profactor GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	iec61499::system

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

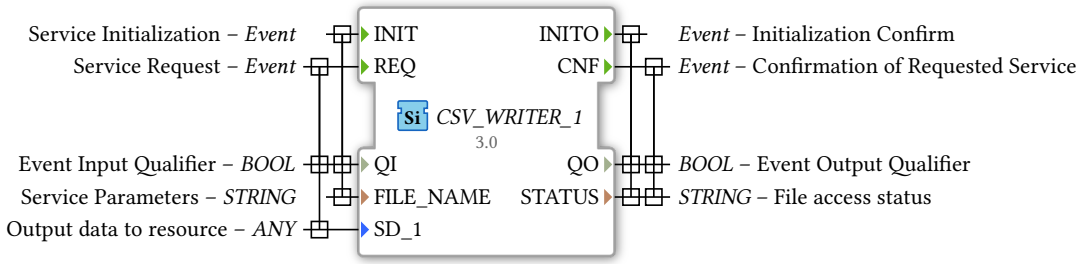
2.11.12 utils

2.11.12.1 CSV_WRITER_1

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `CSV_WRITER_1` serves as a service interface for writing data to a CSV file. It enables the initialization of the write operation and the controlled writing of data to a file with status feedback.



Interface Structure

Event Inputs

- **INIT**: Initializes the write service. Triggered by the variables `QI` (qualifier) and `FILE_NAME` (filename).
- **REQ**: Requests the writing of data. Triggered by the variables `QI` and `SD_1` (data to be written).

Event Outputs

- **INITO**: Confirms initialization. Returns `QO` (qualifier) and `STATUS` (initialization state).
- **CNF**: Confirms completion of the write operation. Returns `QO` and `STATUS` (write operation state).

Data Inputs

- **QI**: Boolean qualifier for events.
- **FILE_NAME**: String specifying the name of the file to be written.
- **SD_1**: Data of type `ANY` to be written to the CSV file.

Data Outputs

- **QO**: Boolean qualifier for output events.
- **STATUS**: String returning the file access status.

Adapters

No adapters available.

Functionality

1. **Initialization**: The `INIT` event initializes the write service with the specified filename (`FILE_NAME`). The status is returned via `INITO`.
2. **Write Data**: The `REQ` event triggers the data write operation (`SD_1`). Success or failure is reported via `CNF` and `STATUS`.

Technical Features

- Supports generic data types (`ANY`) for the data to be written.
- Status feedback as a string for detailed error analysis.

Status Overview

- **Initialization State**: Set by `INIT` and confirmed by `INITO`.
- **Write State**: Activated by `REQ` and completed by `CNF`.

Application Scenarios

- Logging of process data to CSV files.
- Export of measurement data for further analysis.
- Automated data storage in industrial control systems.

⚖️ Comparison with Similar Blocks

- Compared to simple file write blocks, `CSV_WRITER_1` offers detailed status feedback and support for generic data types.
- Other blocks might be limited to specific data types or offer less feedback.

Conclusion

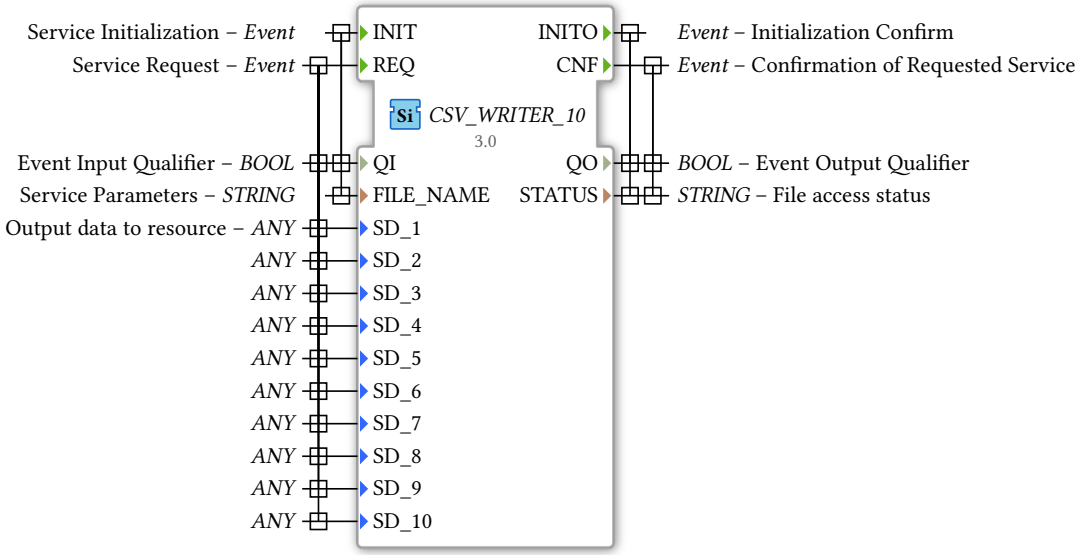
The `CSV_WRITER_1` is a flexible and reliable function block for writing data to CSV files, ideal for applications that require robust logging and data export functionality.

2.11.12.2 CSV_WRITER_10

• – ◦ • – ◦ • – ◦ •

Introduction

The CSV_WRITER_10 function block is used to write data to a CSV file. It allows you to save up to 10 different data values in a structured text file. This block is particularly useful for data collection, logging, and exchanging data with other systems.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block and specifies the file name (using QI and FILE_NAME)
- **REQ**: Triggers the write operation (using QI and SD_1 to SD_10)

Event Outputs

- **INITO**: Confirms initialization (using QO and STATUS)
- **CNF**: Confirms the write operation (using QO and STATUS)

Data Inputs

- **QI**: BOOL - Qualifies the event inputs
- **FILE_NAME**: STRING - Name of the CSV file
- **SD_1 to SD_10**: ANY - Data values to be written (any type)

Data Outputs

- **QO**: BOOL - Qualifies the Event Outputs
- **STATUS**: STRING - File access status message

Adapter

No adapters available.

Functionality

1. **Initialization**: INIT with a valid FILE_NAME prepares the CSV file.
2. **Write Data**: REQ writes the current values from SD_1 to SD_10 as a new line to the file.
3. **Status Feedback**: Each action generates a status message (successful/failed).

Technical Features

- Supports any data type (ANY) for the values to be written.
- Automatic conversion of data to text format.
- Thread-safe file access.
- Maximum capacity of 10 data values per line.

Status Overview

1. **Not Initialized**: Before INIT.
2. **Ready**: After successful initialization.
3. **Error**: In case of file access problems.

Application Scenarios

- Data acquisition of process values.
- Creation of log files.
- Export of measurement data for external analysis.
- Interface to spreadsheet programs.

⚖️ Comparison with Similar Modules

- Compared to simpler This block offers CSV writers greater flexibility through 10 data channels.
- Unlike hard-typed writers, it supports any data type.
- Provides more detailed status feedback than basic versions.

Conclusion

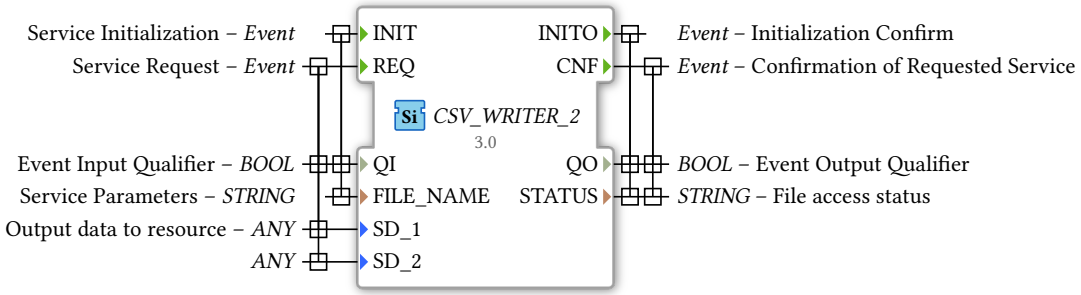
The CSV_WRITER_10 is a powerful function block for structured data storage, particularly well-suited for applications with multiple values to be logged simultaneously. Its flexible type support and detailed status feedback make it a robust solution for industrial data acquisition tasks.

2.11.12.3 CSV_WRITER_2

• – ◦ • – ◦ • – ◦ •

Introduction

The CSV_WRITER_2 is a function block used to write data to a CSV file. It provides a simple interface for initialization and the writing process, supporting two different data fields. This function block is particularly useful for applications where data needs to be stored persistently or exported for further analysis.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block and configures the filename for the CSV file.
- Input data: `QI` (BOOL), `FILE_NAME` (STRING)
- **REQ**: Starts the data writing process to the CSV file.
- Input data: `QI` (BOOL), `SD_1` (ANY), `SD_2` (ANY)

Event Outputs

- **INITO**: Confirms the initialization of the function block.
- Output data: `QO` (BOOL), `STATUS` (STRING)
- **CNF**: Confirms the successful completion of the write operation.

- Output data: `QO` (BOOL), `STATUS` (STRING)

Data Inputs

- **QI**: Qualifies the event (TRUE for valid, FALSE for invalid).
- **FILE_NAME**: The name of the CSV file to be written to.
- **SD_1**: The first data field to be written to the CSV file (any data type).
- **SD_2**: The second data field to be written to the CSV file (any data type).

Data Outputs

- **QO**: Qualifies the output event (TRUE for success, FALSE for failure).
- **STATUS**: Returns the status of the file access (e.g., success message or error description).

Adapters

This function block has no adapters.

Operation

1. **Initialization**: Sending a `INIT` event with the desired file name (`FILE_NAME`) initializes the function block. The initialization status is reported via `INITO`.
2. **Write Data**: The `REQ` event triggers the write operation. The data `SD_1` and `SD_2` are written to the CSV file. The success or failure of the write operation is reported via `CNF`.

Technical Features

- Supports any data type for the data to be written (`SD_1` and `SD_2`).
- The status (`STATUS`) provides detailed information about the success or failure of the file access.
- The function block is part of the `eclipse4diac::utils` package.

Status Overview

1. **Inactive**: The function block is waiting for initialization.
2. **Initialized**: The function block is ready to write data.
3. **Write Operation**: The function block is writing data to the CSV file.

Application Scenarios

- Logging process data to a CSV file for later analysis.
- Exporting measurement data for external evaluation.
- Storing configuration data in a readable format.

⚖️ Comparison with Similar Function Blocks

- Compared to simpler write function blocks, `CSV_WRITER_2` supports two data fields and provides detailed status feedback.
- Other CSV write function blocks may be less flexible in their data type support or may not provide status feedback.

Conclusion

The `CSV_WRITER_2` function block is a flexible and reliable function block for writing data to CSV files. Its support for arbitrary data types and detailed status feedback make it ideal for applications that require persistent data storage.

🌐 Related topic subpages on ms-muc-docs.de

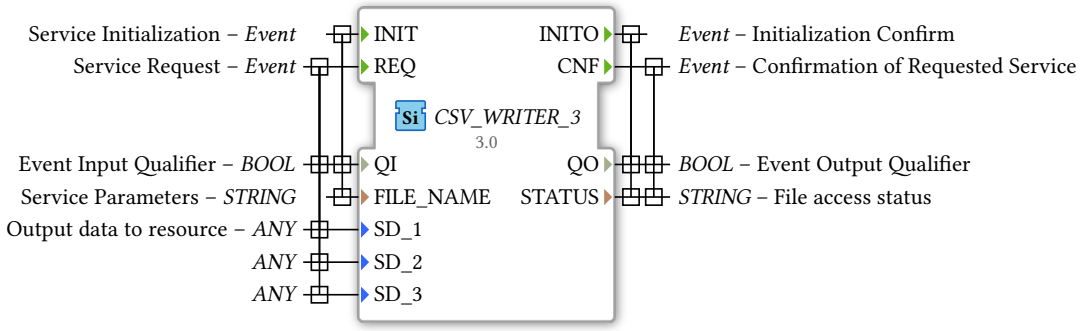
- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

2.11.12.4 CSV_WRITER_3

• – ◦ • – ◦ • – ◦ •

Introduction

The CSV_WRITER_3 is a function block used to write data to a CSV file. It supports writing up to three different data fields to a file, specified via the parameter `FILE_NAME`. This function block is particularly useful for logging and data storage in automation systems.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block. Triggered with the data `QI` (Qualifier Input) and `FILE_NAME` (Filename).
- **REQ**: Starts the write operation. Triggered with the data `QI`, `SD_1`, `SD_2`, and `SD_3` (the data to be written).

Event Outputs

- **INITO**: Confirms initialization. Returns `QO` (Qualifier Output) and `STATUS` (Initialization Status).
- **CNF**: Confirms completion of the write operation. Returns `QO` and `STATUS`.

Data Inputs

- **QI**: Boolean value indicating the quality of the event input.
- **FILE_NAME**: String specifying the name of the CSV file.
- **SD_1**, **SD_2**, **SD_3**: Data fields of type `ANY` that are written to the CSV file.

Data Outputs

- **QO**: Boolean value indicating the quality of the event output.
- **STATUS**: String indicating the status of the file access.

Adapters

No adapters available.

Functionality

The function block initializes upon receiving the `INIT` event and acknowledges this with `INITO`. Upon receiving the `REQ` event, the data `SD_1`, `SD_2`, and `SD_3` are written to the specified CSV file. The write status is reported via `CNF`.

Technical Features

- Supports writing to up to three different data fields.
- The data type of the data to be written is `ANY`, allowing for high flexibility.
- Detailed file access status is reported.

State Overview

1. **Initialization State**: Waiting for the `INIT` event.
2. **Ready State**: After successful initialization, waits for the `REQ` event.
3. **Write State**: Writes data to the CSV file and reports the status back.

Application Scenarios

- Logging sensor data to a CSV file.
- Storing process data for later analysis.
- Exchanging data with other systems via CSV files.

⚖️ Comparison with Similar Function Blocks

- Compared to simpler CSV writer function blocks, `CSV_WRITER_3` supports writing to multiple data fields (`SD_1`, `SD_2`, `SD_3`).

Other function blocks might be restricted to specific data types, while `CSV_WRITER_3` uses the `ANY` type.

Conclusion

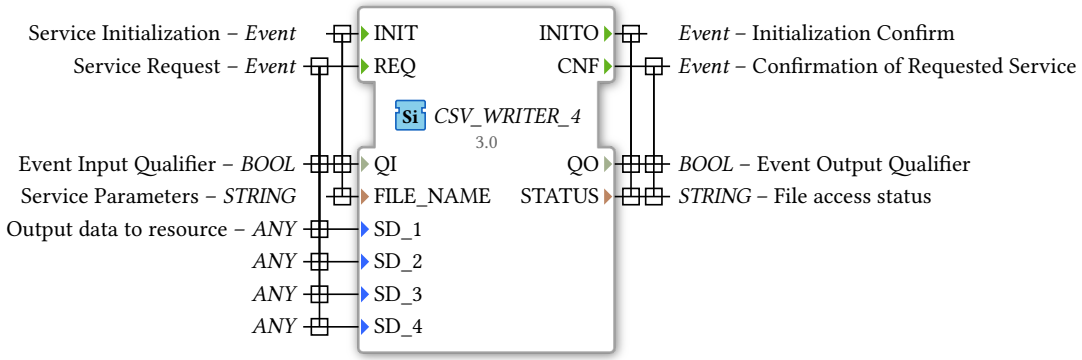
The `CSV_WRITER_3` is a flexible and powerful function block for storing data in CSV files. Its support for multiple data fields and the `ANY` data type makes it particularly versatile.

2.11.12.5 CSV_WRITER_4

• – ◦ • – ◦ • – ◦ •

Introduction

The CSV_WRITER_4 function block is used to write data to a CSV file. It supports saving up to four different data values in a structured format. The function block is particularly useful for data acquisition and logging in automation applications.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block and configures the output file. It links to data in `QI` and `FILE_NAME`.
- **REQ**: Triggers the write operation. Linked to `QI` and the data values `SD_1` through `SD_4`.

Event Outputs

- **INITO**: Confirms initialization. Returns `QO` and `STATUS`.
- **CNF**: Confirms the write operation is complete. Returns `QO` and `STATUS`.

Data Inputs

- **QI** (BOOL): Qualifies the INIT and REQ events.
- **FILE_NAME** (STRING): Name of the CSV file.
- **SD_1** to **SD_4** (ANY): The data values to be written (up to 4 different ones).

Data Outputs

- **QO** (BOOL): Qualifies the output events.
- **STATUS** (STRING): Status message about file access.

Adapters

No adapters available.

Functionality

1. **Initialization**: The INIT event with `QI` active prepares the CSV file with the specified `FILE_NAME`.
2. **Write Data**: The REQ event with `QI` active triggers the write operation. The values from `SD_1` to `SD_4` are written to the file.
3. **Confirmation**: After each operation (INIT or REQ), the status is reported via `INITO` or `CNF`.

Technical Features

- Supports generic data types (ANY) for the data to be written.
- The function block is part of the `eclipse4diac::utils` package.
- The generic class is defined as `GEN_CSV_WRITER`.

Status Overview

1. **Inactive**: Waiting for initialization.
2. **Initialized**: File is ready, waiting for write requests.
3. **Writing**: Processing a write request.

Application Scenarios

- Logging of process data.
- Data acquisition for analysis purposes.
- Export of measured values to a standardized format.

⚖️ Comparison with Similar Function Blocks

- Compared to simpler write function blocks, CSV_WRITER_4 offers the ability to store multiple data values simultaneously.
- The use of ANY types makes it more flexible than function blocks with fixed data types.

Conclusion

CSV_WRITER_4 is a versatile function block for data acquisition, particularly well-suited for applications that require logging structured data. Its flexibility and ease of integration make it a practical solution for many automation tasks.

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

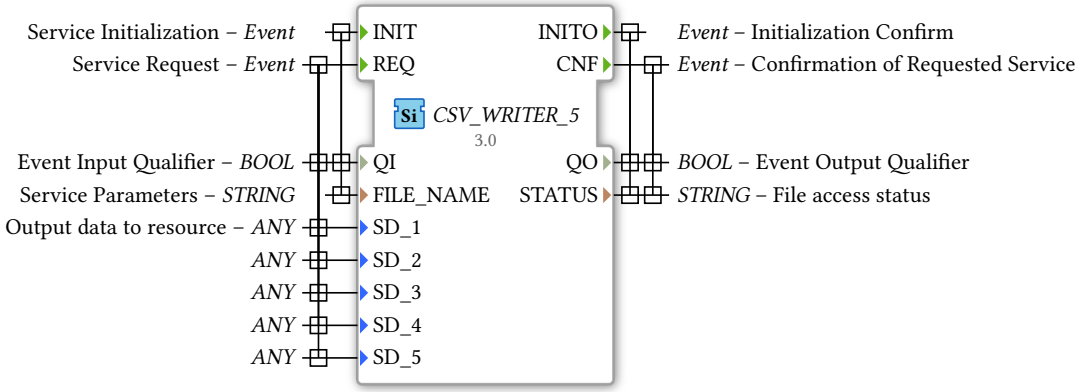
]

2.11.12.6 CSV_WRITER_5

• – ◦ • – ◦ • – ◦ •

Introduction

The CSV_WRITER_5 is a function block for outputting data to a CSV file. It allows you to write up to five different data points to a user-defined file. This function block is particularly useful for data acquisition and logging applications.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block and opens the specified CSV file.
- Linked Data: QI (BOOL), FILE_NAME (STRING)
- **REQ**: Triggers the write operation.

- Linked Data: QI (BOOL), SD_1 to SD_5 (ANY)

Event Outputs

- **INITO**: Confirms initialization.
- Linked Data: QO (BOOL), STATUS (STRING)
- **CNF**: Confirms successful write operation.
- Linked Data: QO (BOOL), STATUS (STRING)

Data Inputs

- **QI**: Qualifies the event (TRUE for active, FALSE for inactive).
- **FILE_NAME**: Name of the CSV file to be written to.
- **SD_1 to SD_5**: The data to be written (any type).

Data Outputs

- **QO**: Returns the status of the function block (TRUE for successful, FALSE for error).
- **STATUS**: Contains a status message about the last operation.

Adapters

No adapters available.

Functionality

1. **Initialization**: The INIT event opens the CSV file with the specified name. If the file does not exist, it will be created.
2. **Write Data**: The REQ event triggers the write operation. Data SD_1 to SD_5 is written as a single line to the CSV file.
3. **Acknowledge**: After each write operation, the CNF event is triggered, which returns the status of the operation.

Technical Features

- Supports any data type for the output data (ANY).
- The status output (STATUS) returns detailed error messages.
- Initialization can be repeated if necessary to open a new file.

Status Overview

- **Inactive**: Waiting for initialization.
- **Ready**: File is open and ready for write operations.
- **Error**: An error has occurred (e.g., file access denied).

Application Scenarios

- Logging sensor data.
- Storing process data for later analysis.
- Export of data to a table-based format.

⚖️ Comparison with similar building blocks

- Compared to simpler CSV writer blocks, CSV_WRITER_5 offers greater flexibility by supporting five data points and any data type.
- Other blocks might be limited to specific data types or support fewer data points.

Conclusion

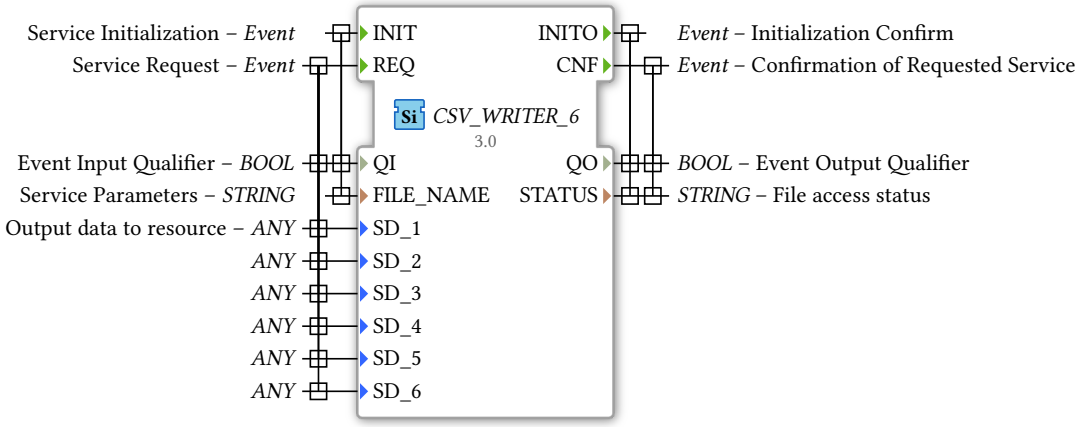
CSV_WRITER_5 is a powerful and flexible function block for data collection and logging. Its support for any data type and multiple data points makes it ideal for complex applications.

2.11.12.7 CSV_WRITER_6

• − ◦ • − ◦ • − ◦ •

Introduction

The function block `CSV_WRITER_6` is used to write data to a CSV file. It allows the storage of up to six different data values in a comma-separated format. The block is particularly useful for data acquisition and logging in automation systems.



Interface Structure

Event Inputs

- `INIT`: Initializes the function block. Triggered by the variables `QI` and `FILE_NAME`.
- `REQ`: Starts the write operation. Triggered by the variables `QI`, `SD_1` through `SD_6`.

Event Outputs

- `INITO`: Confirms initialization. Returns `QO` and `STATUS`.
- `CNF`: Confirms successful write operation. Returns `QO` and `STATUS`.

Data Inputs

- `QI` (BOOL): Qualifies the event (e.g., whether the write operation should be activated).
- `FILE_NAME` (STRING): Name of the CSV file to be written to.
- `SD_1` to `SD_6` (ANY): The data values to be written.

Data Outputs

- `QO` (BOOL): Qualifies the output event (e.g., whether the operation was successful).
- `STATUS` (STRING): File access status message.

Adapters

No adapters available.

Functionality

1. **Initialization**: The `INIT` event initializes the function block with the file name (`FILE_NAME`). The status is reported via `INITO`.
2. **Write Operation**: The `REQ` event triggers the write operation. The data `SD_1` to `SD_6` are written to the CSV file. Success is confirmed via `CNF`.

Technical Features

- Supports any data type (`ANY`) for the data to be written.
- The status (`STATUS`) provides detailed information about whether the file access was successful or failed.

State Overview

1. **Initialization State**: Waiting for the `INIT` event.
2. **Ready State**: After successful initialization, waits for the `REQ` event.
3. **Write State**: Processes the data and writes it to the file.

Application Scenarios

- Real-time logging of sensor data.

- Storage of process data for later analysis.
- Data acquisition in test and validation environments.

⚖️ Comparison with Similar Blocks

- Compared to simpler CSV writer blocks, `CSV_WRITER_6` offers the ability to store up to six different data values simultaneously.
- Other blocks may be limited to specific data types, while this block supports `ANY` types.

Conclusion

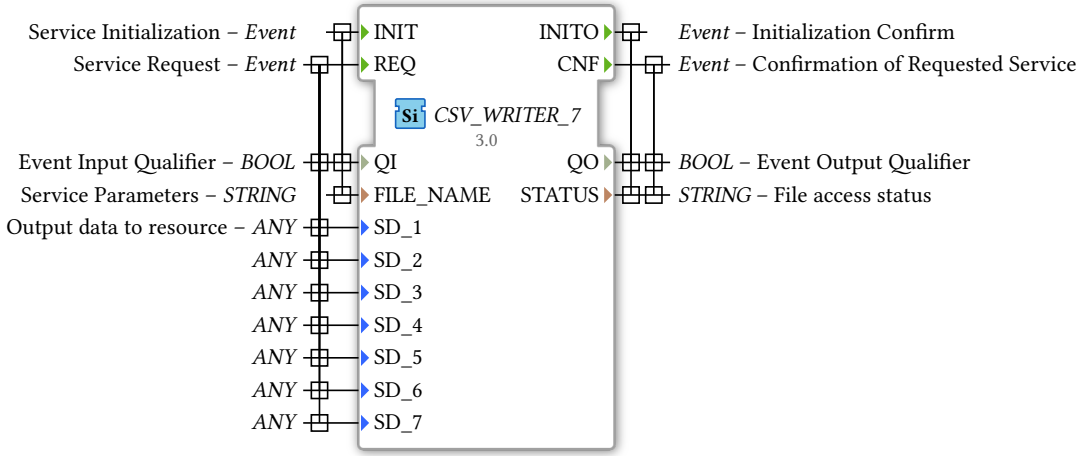
The `CSV_WRITER_6` is a flexible and powerful function block for storing data in CSV files. Its support for various data types and the ability to store multiple values simultaneously make it ideal for complex data collection tasks.

2.11.12.8 CSV_WRITER_7

• – ◦ • – ◦ • – ◦ •

Introduction

The CSV_WRITER_7 is a function block that writes data to a CSV file. It supports writing up to seven different data fields to a specified file. This function block is particularly useful for logging and data storage in automation applications.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block. Used with the variables `QI` (Qualifier Input) and `FILE_NAME` (CSV file name).
- **REQ**: Starts the write operation. Used with the variables `QI` and the seven data fields `SD_1` to `SD_7`.

Event Outputs

- **INITO**: Confirms initialization. Returns `QO` (qualifier output) and `STATUS` (initialization status).
- **CNF**: Confirms the write operation is complete. Returns `QO` and `STATUS`.

Data Inputs

- **QI**: Boolean qualifier indicating whether the operation should be performed.
- **FILE_NAME**: Name of the CSV file to which data should be written.
- **SD_1** to **SD_7**: Data fields that are written to the CSV file. Each field can have any data type (`ANY`).

Data Outputs

- **QO**: Boolean qualifier indicating the success or failure of the operation.
- **STATUS**: String describing the status of the file access.

Adapters

No adapters are present.

Operation

1. **Initialization**: The function block is initialized by the `INIT` event. The filename (`FILE_NAME`) is passed to this event.
2. **Write Operation**: The `REQ` event triggers the write operation. The data `SD_1` to `SD_7` will be written to the specified CSV file.
3. **Confirmation**: Upon completion of the operation, the `CNF` event is triggered, which returns the status of the operation.

Technical Features

- Supports writing to up to seven different data fields.
- The data fields can be of any data type (`ANY`).
- The file access status is returned in detail.

State Overview

1. **Initialization State**: Waiting for the `INIT` event.
2. **Ready**: After successful initialization, the function block waits for the `REQ` event.
3. **Write Operation**: The function block is active during the write operation.
4. **Completed**: The `CNF` event is triggered after the write operation is complete.

Application Scenarios

- Logging of process data in industrial automation.
- Storage of measurement data for later analysis.
- Data acquisition in real-time systems.

⚖️ Comparison with Similar Function Blocks

- **CSV_WRITER_5**: Supports only five data fields.
- **FILE_WRITER**: More general function block that is not specifically optimized for CSV files.

Conclusion

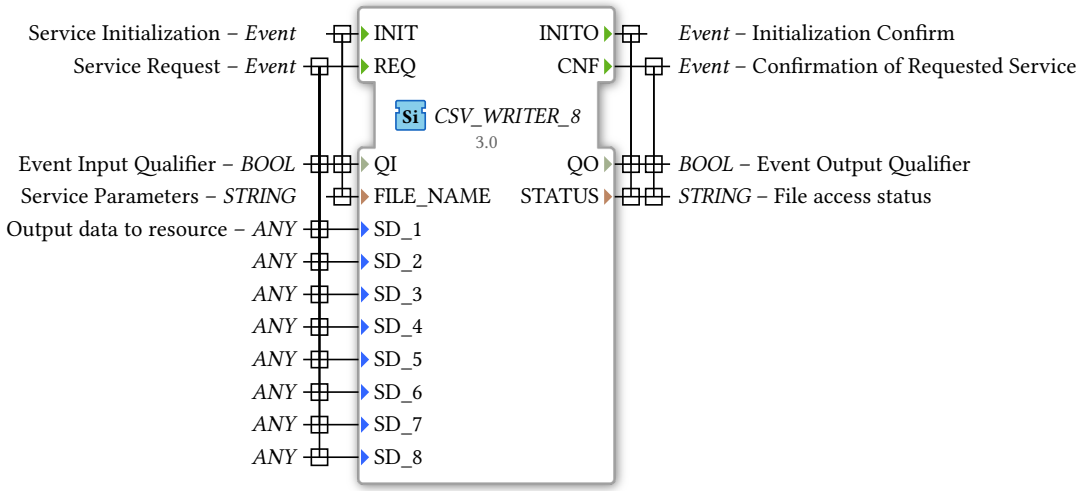
The CSV_WRITER_7 is a powerful function block for storing data in CSV files. Its flexibility in supporting various data types and its detailed status feedback make it an ideal choice for many automation applications.

2.11.12.9 CSV_WRITER_8

• – ◦ • – ◦ • – ◦ •

Introduction

The CSV_WRITER_8 is a function block for outputting data to a CSV file. It allows you to write up to 8 different data values to a specified file. This function block is part of the eclipse4diac::utils package family and is particularly suitable for data acquisition and logging tasks.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block. It uses the variables QI (Qualifier) and FILE_NAME (Filename).
- **REQ**: Triggers a write operation. It uses QI and the data variables SD_1 to SD_8.

Event Outputs

- **INITO**: Confirms initialization. Returns QO (qualifier) and STATUS (status message).
- **CNF**: Confirms the completion of the write operation. Also returns QO and STATUS.

Data Inputs

- **QI**: Boolean qualifier for events.
- **FILE_NAME**: String specifying the path and name of the CSV file.
- **SD_1 to SD_8**: Data variables of type ANY to be written to the CSV file.

Data Outputs

- **QO**: Boolean qualifier for output events.
- **STATUS**: String describing the status of the file access.

Functionality

The function block initializes itself upon receiving the INIT event and confirms this with INITO. The REQ event writes the current values of SD_1 to SD_8 to the specified CSV file. After the write operation is complete, the CNF event is triggered.

Technical Features

- Supports generic data types (ANY) for output data.
- Provides status messages for successful and failed operations.
- Part of the eclipse4diac::utils package.

State Overview

1. **Initialization State**: Waiting for the INIT event.
2. **Ready**: After successful initialization.
3. **Write Operation**: Active during the REQ event.
4. **Acknowledgement**: Sends CNF upon completion.

Application Scenarios

- Logging of process data.
- Data acquisition for analysis purposes.
- Export of control data in CSV format.

⚖️ Comparison with Similar Blocks

Compared to simpler CSV writer blocks, CSV_WRITER_8 offers the ability to write up to 8 different data values simultaneously and process generic data types.

Conclusion

CSV_WRITER_8 is a powerful function block for data acquisition and logging in CSV files. Its data processing flexibility and detailed status messages make it a good choice for complex applications.

🌐 Related Topic Subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

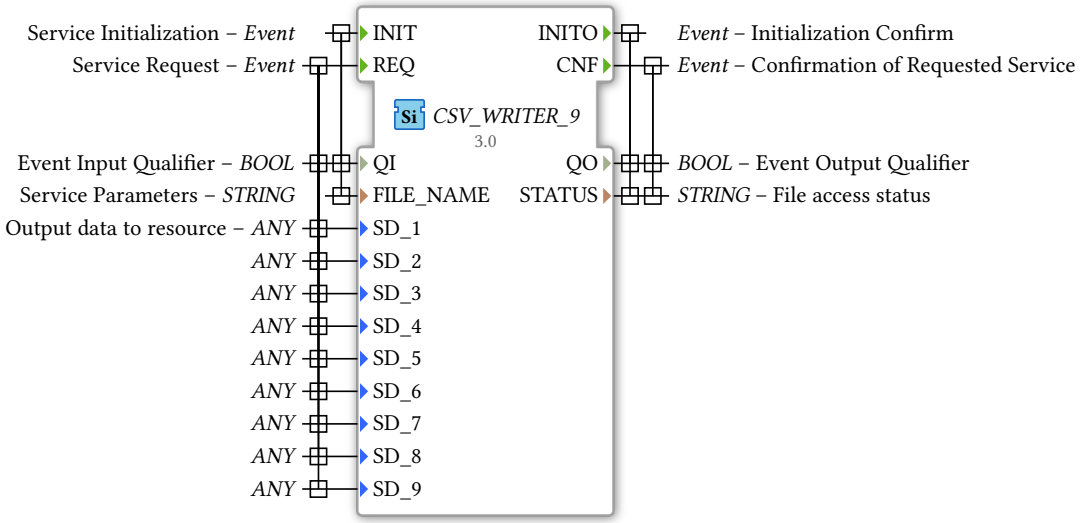
]

2.11.12.10 CSV_WRITER_9

• – ◦ • – ◦ • – ◦ •

Introduction

The CSV_WRITER_9 is a function block for creating CSV files. It allows you to write up to nine different data points to a CSV file. This function block is part of the eclipse4diac::utils package and is suitable for applications that need to store data in a standardized format.



Interface Structure

Event Inputs

- **INIT**: Initializes the function block and sets the file name. Used with `QI` and `FILE_NAME`.
- **REQ**: Starts the writing process to the CSV file. Used with `QI` and data points `SD_1` through `SD_9`.

Event Outputs

- **INITO**: Confirms initialization and returns the status. Used with `QO` and `STATUS`.
- **CNF**: Confirms successful or failed write operation. Used with `QO` and `STATUS`.

Data Inputs

- **QI**: Boolean qualifier for event inputs.
- **FILE_NAME**: String specifying the name of the CSV file.
- **SD_1** to **SD_9**: Data points of type `ANY`, which are written to the CSV file.

Data Outputs

- **QO**: Boolean qualifier for event outputs.
- **STATUS**: String indicating the status of file access.

Adapters

No adapters are defined.

Functionality

1. **Initialization**: The function block is initialized by the `INIT` event. The filename is set via `FILE_NAME`.
2. **Write Operation**: The `REQ` event triggers the write operation. Data points `SD_1` to `SD_9` are written to the CSV file.
3. **Confirmation**: The status of the operation is returned via `INITO` or `CNF`.

Technical Features

- Supports up to nine data points (`SD_1` to `SD_9`) of type `ANY`.
- The file access status is returned via the `STATUS` output.
- This function block is part of the eclipse4diac::utils package and is subject to the Eclipse Public License 2.0.

Status Overview

- **Initialization**: Successful (`INITO+`) or failed (`INITO-`).

- **Write Operation**: Successful (`CNF+`) or failed (`CNF-`).

Application Scenarios

- Data acquisition and storage in industrial control systems.
- Logging of process data to a standardized CSV file.
- Integration into larger automation solutions for data processing.

⚖️ Comparison with Similar Function Blocks

- Compared to simpler CSV writer function blocks, CSV_WRITER_9 offers the ability to write up to nine data points simultaneously.
- Other function blocks may offer less flexibility in data types due to the use of `ANY`.

Conclusion

The CSV_WRITER_9 is a powerful function block for creating CSV files in the 4diac IDE. Its flexibility in data ingestion and clear status feedback make it a good choice for applications that require structured data storage.

🌐 Related topic subpages on ms-muc-docs.de

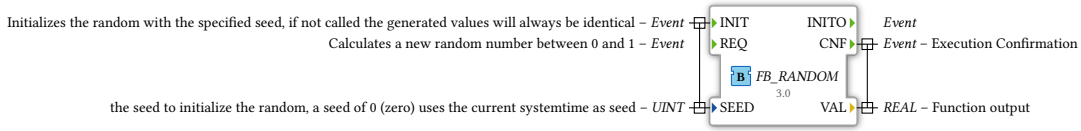
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

2.11.12.11 FB_RANDOM

• – ○ • – ○ • – ○ •

Introduction

The FB_RANDOM function block generates a random REAL number between 0 and 1.



Interface Structure

Event Inputs

- **INIT**: Initializes the random number generator with a seed.
- **REQ**: Calculates a new random number.

Event Outputs

- **INITO**: Initialization complete.
- **CNF**: Calculation complete, new value available.

Data Inputs

- **SEED** (UINT): The seed value for the random number generator. A value of 0 uses the current system time.

Data Outputs

- **VAL** (REAL): The generated random number.

Metadata

Attribute	Value
Copyright	(c) 2012 Profactor GmbH
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac Package	eclipse4diac::utils

🌐 Related topic subpages on ms-muc-docs.de

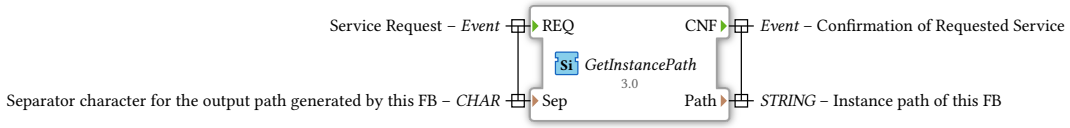
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

2.11.12.12 GetInstancePath

• – ◦ • – ◦ • – ◦ •

The function block `GetInstancePath` is used to determine the instance path of the function block within the 4diac IDE. This path can be used, for example, for diagnostic purposes or for dynamic configuration. The function block is particularly useful in complex applications where precise identification of the instance location is required.



- **REQ**: Service request. Triggers the determination of the instance path.
- **CNF**: Acknowledgement of the service request. Signals successful determination of the instance path.
- **Sep**: Separator character (CHAR) for the generated path. The default value is `'/'`.
- **Path**: The determined instance path as a STRING.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

No adapters available.

Functionality

1. Upon receiving the `REQ` event, the function block begins determining its own instance path.
2. The path is formatted with the specified separator character (`Sep`).
3. The `CNF` event is triggered, and the determined path is output via the `Path` output.

- The function block uses a service interface to communicate with the resource level.
- **Idle**: Waiting for a `REQ` request.
- **Processing**: Determines the instance path and outputs it via `CNF`.
- Diagnostics and Logging: Identifying the exact location of a function block instance in a complex application.
- Dynamic Configuration: Using the path to dynamically adjust parameters or behaviors.
- Unlike static path specifications, `GetInstancePath` allows for dynamic path determination, which is particularly advantageous in reusable or nested components.

The `GetInstancePath`-FB is a useful tool for dynamically determining the instance path in 4diac applications. Its simple interface and flexible configuration make it ideal for diagnostic and configuration purposes.

–

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

Technical Features

State Overview

Application Scenarios

⚖️ Vergleich mit ähnlichen Bausteinen

Conclusion

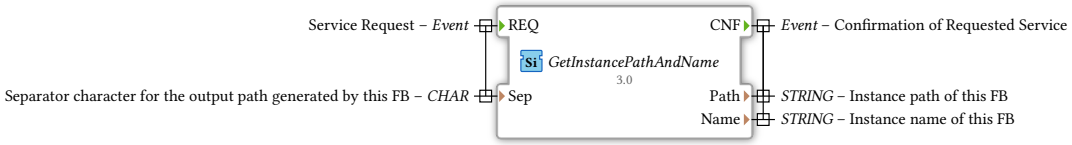
🌐 Passende Themen-Unterseiten auf ms-muc-docs.de

2.11.12.13 GetInstancePathAndName

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `GetInstancePathAndName` is used to retrieve the instance path and instance name from the runtime environment. This is particularly useful for debugging purposes or for dynamic configurations where the exact path of an instance is required.



Interface Structure

Event Inputs

- **REQ**: Service request that triggers the retrieval of the instance path and name. Concatenated with the separator character `Sep`.

Event Outputs

- **CNF**: Acknowledgement of the requested service request. The output variables `Path` (instance path) and `Name` (instance name) are linked.

Data Inputs

- **Sep** (CHAR): Separator character for the generated output path. The default value is `'/'`.

Data Outputs

- **Path** (STRING): The instance path of the function block.
- **Name** (STRING): The instance name of the function block.

Adapters

No adapters are available.

Functionality

1. The function block is activated upon receiving a `REQ` event.
2. The function block determines the current instance path and instance name from the runtime environment.
3. The instance path is formatted with the specified separator character `Sep`.
4. The `CNF` event is triggered, and the determined values for `Path` and `Name` are output.

Technical Features

- The default separator for the path is `'/'`, but it can be changed via the input `Sep`.
- The output variables `Path` and `Name` are of type `STRING` and contain the complete path and the instance name, respectively.

State Overview

The function block has no internal state. It reacts exclusively to the `REQ` event and immediately outputs the corresponding data.

Application Scenarios

- **Debugging**: Determining the exact path of an instance for troubleshooting.
- **Dynamic Configuration**: Using the instance path for dynamic configurations or logging.
- **System Monitoring**: Monitoring and logging instance paths in complex systems.

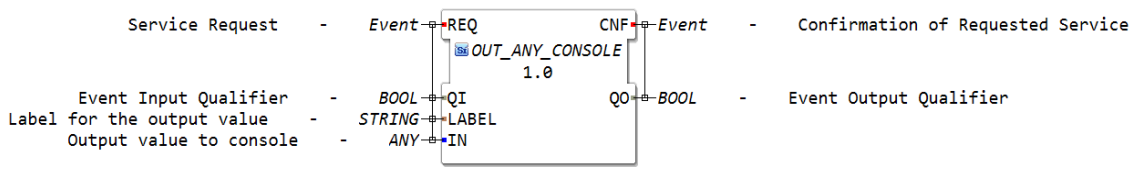
⚖️ Comparison with Similar Blocks

- Compared to other function blocks that provide similar information, `GetInstancePathAndName` offers the flexibility to customize the path separator.
- Other blocks might only provide the path or only the name, while this function block combines both.

Conclusion

The `GetInstancePathAndName` function block is a simple yet powerful tool for retrieving instance information from the runtime environment. Its flexibility and simplicity make it ideal for debugging and dynamic configurations.

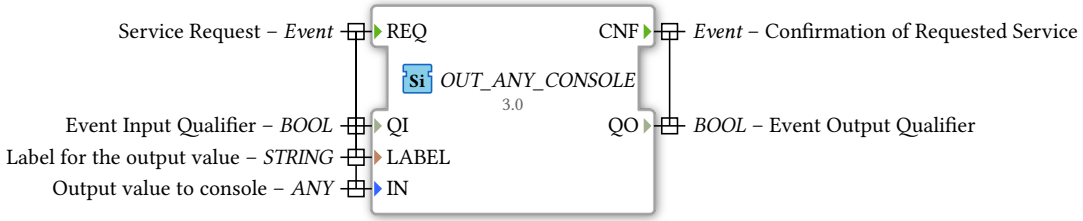
2.11.12.14 OUT_ANY_CONSOLE



• — ○ • — ○ • — ○ •

Introduction

The **OUT_ANY_CONSOLE** function block is a universal debugging tool for outputting variable values to the system console, developed under the EPL 2.0 license (version 3.0). The block supports outputting any data type for development and diagnostic purposes.



Interface Structure

Event Inputs

- `REQ`: Trigger for console output (with qualifier `QI`, value `IN`, and optional `LABEL`)

Event Outputs

- `CNF`: Acknowledgement of the output operation

Data Inputs

Parameter	Type	Description
`QI`	BOOL	Enables output when TRUE
`LABEL`	STRING	Optional description text
`IN`	ANY	Any value to be output

Data Outputs

Parameter	Type	Description
QO	BOOL	Confirms successful output

Functionality

1. **Triggering**:

- The REQ event with QI=TRUE starts output
- The combination of LABEL and IN is formatted

1. **Execution**:

- The value is written to standard output
- Automatic type conversion for ANY

1. **Confirmation**:

- CNF with QO=TRUE on success
- CNF with QO=FALSE on errors

Technical Features

✓ **Universal Type Support** (ANY data type) ✓ **Flexible labeling** with LABEL parameters ✓ **Eclipse 4diac integration** (Package: eclipse4diac::utils) ✓ **Platform-independent** for various target systems

Application Scenarios

- **Debugging**: Runtime variable inspection
- **Logging**: Event tracking
- **System diagnostics**: Condition monitoring
- **Development**: Rapid algorithm testing

Example outputs

LABEL	IN (value)	Console output
“Temperature”	23.5	“Temperature: 23.5”
“Status”	TRUE	“Status: TRUE”
““	42	“42”

⚖️ **Comparison with similar building blocks**

Feature	OUT_ANY_CONSOLE	Standard debugging	Extended logging
Data types	Any	Limited	Limited
Labeling	Optional	No	Yes
Integration	4diac-specific	System-dependent	Framework-dependent
Real-time capable	Yes	Partially	No

Conclusion

The OUT_ANY_CONSOLE block is an indispensable tool for development:

- **Flexible**: Supports all data types
- **Easy**: Quick integration into existing systems
- **Powerful**: Suitable for real-time debugging

Ideal for:

- Prototype development
- System testing
- Error analysis

- Live diagnostics

The Swiss Army knife for developers in Eclipse 4diac environments

---

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & color reference on ms-muc-docs.de

]

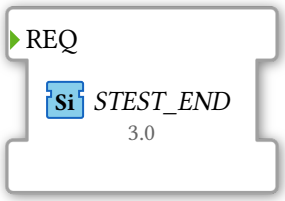
2.11.12.15 STEST_END

• – ◦ • – ◦ • – ◦ •

Introduction

The STEST_END function block is a Service Interface Function Block that serves as an endpoint for service requests. It is typically used in conjunction with other service function blocks to implement a service-oriented architecture in the 4diac IDE.

Service Request – *Event*



Interface Structure

Event Inputs

- **REQ**: Service Request - Triggered to initiate a service request.

Event Outputs

No event outputs available.

Data Inputs

No data inputs available.

Data Outputs

No data outputs available.

Adapters

No adapters available.

Functionality

The STEST_END function block serves as an endpoint for service requests. When the REQ input is triggered, the service request is processed. Since it is an endpoint, there are no other event or data outputs.

Technical Features

- Implemented as a Service Interface Function Block.
- Supports communication between RESOURCE and APPLICATION interfaces.

State Overview

The function block has no complex states. It simply responds to and processes the REQ request.

Application Scenarios

- As an endpoint in a service-oriented architecture.
- In test environments to simulate or terminate service requests.

⚖️ Comparison with Similar Function Blocks

Compared to other service function blocks, STEST_END has no outputs or additional interfaces, making it ideal as an endpoint.

Conclusion

The STEST_END function block is a simple yet essential building block for service-oriented architectures in the 4diac IDE. Its simplicity makes it a reliable endpoint for service requests.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

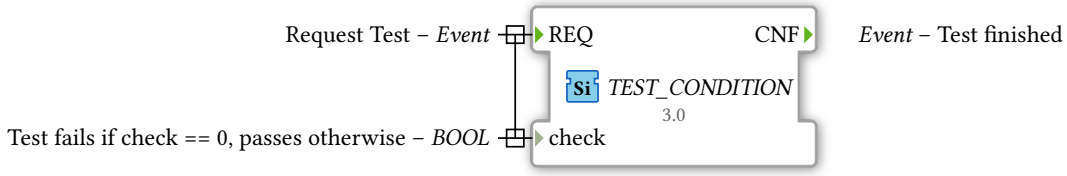
]

2.11.12.16 TEST_CONDITION

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `TEST_CONDITION` is used to check a condition and signals whether it is met or not. It is typically used in test environments to validate specific conditions.



Interface Structure

Event Inputs

- **REQ**: Starts the condition check. When this input is triggered, the function block begins evaluating the data input `check`.

Event Outputs

- **CNF**: Signals the completion of the check. This event is triggered after the condition has been evaluated.

Data Inputs

- **Check**: A Boolean value (`BOOL`) representing the condition to be checked.
- If `check == TRUE`, the condition is considered fulfilled (test passed).
- If `check == FALSE`, the condition is considered not fulfilled (test failed).

Data Outputs

- None present.

Adapters

- None present.

Functionality

1. The function block is activated by the event `REQ`.
2. The value of the data input `check` is evaluated.
3. The result of the evaluation is processed internally (no direct feedback via data outputs).
4. The event `CNF` is triggered to signal the completion of the check.

Technical Features

- The function block has a simple structure and does not contain any complex state machines.
- No data outputs are provided, as the result of the check is processed internally only.

State Overview

The function block has a simple sequence:

1. **Idle State**: Waits for the `REQ` event.
2. **Evaluation State**: Evaluates `check` and triggers `CNF`.
3. Returns to the **Idle State**.

Application Scenarios

- **Test Automation**: Checking conditions in automated test sequences.
- **Fault Detection**: Validating system states or input data.
- **Prototypical Implementation**: Rapid integration into test environments for logic validation.

⚖️ Comparison with Similar Function Blocks

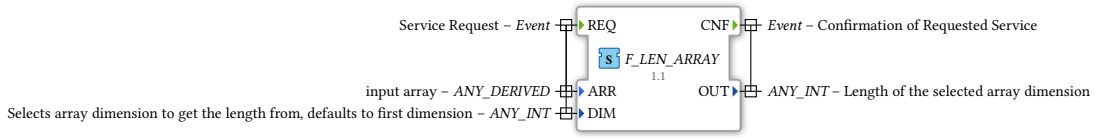
- Compared to other test blocks, `TEST_CONDITION` offers a minimalist interface and is specialized for evaluating a single condition.
- More complex test blocks might support additional features such as time monitoring or multiple conditions.

Conclusion

The `TEST_CONDITION` function block is a simple yet effective tool for checking conditions in test environments. Its streamlined interface makes it easy to integrate, while its functionality is sufficient for basic validations. However, for more complex requirements, more advanced test blocks should be considered.

2.11.12.17 Arrays

F_LEN_ARRAY



• – ◦ • – ◦ • – ◦ •

Introduction

`F_LEN_ARRAY` is a simple function block that returns the length (element count) of a selected dimension of an array of arbitrary type. It complements the standard functions `UPPER_BOUND`/`LOWER_BOUND` with a directly usable length calculation.

Interface Structure

Event Inputs

- **REQ**: Triggers the calculation, carries `ARR` and `DIM`.

Event Outputs

- **CNF**: Confirms completion, carries `OUT`.

Data Inputs

- **ARR** (ANY_DERIVED): The array whose length is to be determined.
- **DIM** (ANY_INT, default `DINT#1`): The array dimension to consider (for multi-dimensional arrays).

Data Outputs

- **OUT** (ANY_INT): The length of the selected dimension, i.e. the element count.

Functionality

The block is implemented as a `SimpleFB` with a single ECC state. On `REQ`, the algorithm's ST statement computes `OUT := ADD(SUB(UPPER_BOUND(ARR, DIM), LOWER_BOUND(ARR, DIM)), 1)` — i.e. the upper minus the lower index bound of the selected dimension, plus 1 (since both bounds are inclusive). `CNF` is then triggered.

Technical Features

- **`ANY_DERIVED` input**: `ARR` accepts arrays of any element type and any dimensionality.
- **Dimension selection via `DIM`**: For multi-dimensional arrays, `DIM` selects which dimension's length to query; the default is the first dimension (`DINT#1`).
- **Based on `UPPER_BOUND`/`LOWER_BOUND`**: The block is a direct combination of these two IEC 61131-3 standard functions, saving the manual arithmetic.

State Overview

Stateless: every `REQ` immediately results in the calculation and `CNF`.

Application Scenarios

- **Dynamic array processing**: Looping over an array whose actual size is only known at runtime (e.g. depending on the caller).
- **Sanity checking**: Comparing the actual array length against an expected size before accessing individual elements.

Comparison with similar function blocks

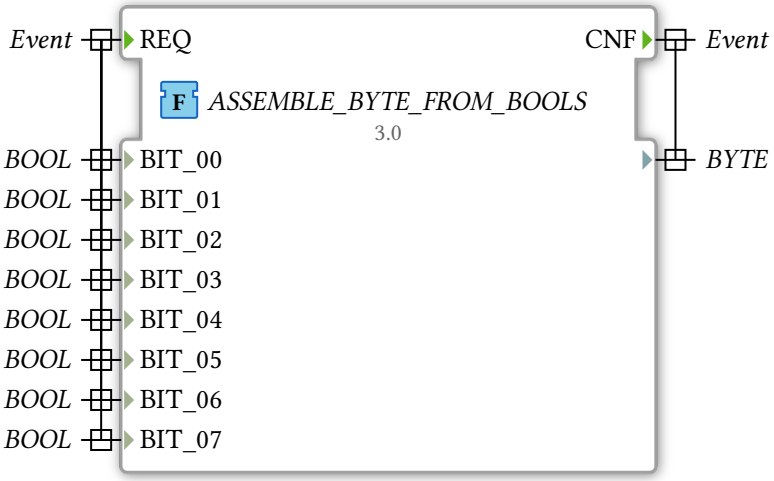
- **`UPPER_BOUND`/`LOWER_BOUND`**: the two standard functions `F_LEN_ARRAY` combines to perform its calculation.

Conclusion

`F_LEN_ARRAY` provides a simple, directly usable length calculation for arrays of arbitrary type and a selectable dimension, without requiring `UPPER_BOUND`/`LOWER_BOUND` to be combined manually.

2.11.12.18 assembling

ASSEMBLE_BYTE_FROM_BOOLS



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_BYTE_FROM_BOOLS` combines eight Boolean input values into a single byte. Each Boolean input represents one bit of the resulting byte. This block is particularly useful when individual Boolean signals need to be combined into a compact byte value.

Interface Structure

Event Inputs

- `REQ`: Starts processing the Boolean inputs and generates the resulting byte.

Event Outputs

- `CNF`: Signals successful aggregation of the Boolean values into a single byte.

Data Inputs

- `BIT_00` to `BIT_07`: Eight Boolean inputs, each corresponding to one bit of the resulting byte.

Data Outputs

- `BYTE`: The composite byte generated from the eight Boolean inputs.

Adapters

No adapters are included.

Functionality

This function block combines the eight Boolean inputs (`BIT_00` to `BIT_07`) into a single byte. Each Boolean value is mapped to the corresponding bit of the byte:

- `BIT_00` corresponds to bit 0 (LSB)
- `BIT_01` corresponds to bit 1
- ...
- `BIT_07` corresponds to bit 7 (MSB)

When the event `REQ` is triggered, the current values of the Boolean inputs are read and combined into a byte. The result is output at the data output, and the event `CNF` signals successful processing.

Technical Features

- The function block is implemented in ST (Structured Text).
- Bit mapping is performed directly by assigning the Boolean values to the corresponding bits of the byte.

State Overview

1. **Idle State**: Waits for the `REQ` event.
2. **Processing State**: Reads the Boolean inputs and assembles the byte.
3. **Acknowledge State**: Outputs the resulting byte and signals `CNF`.

Application Scenarios

- Summarizing 8 individual switch states into one byte for communication with other systems.
- Compressing Boolean status information for more efficient data transmission.
- Use in control systems where multiple Boolean signals need to be processed as a single byte.

⚖️ Comparison with Similar Function Blocks

- Compared to manual bit operations in ST or other languages, this function block offers a predefined and maintainable solution.

Similar function blocks might have fewer inputs or offer additional features like inverted logic, which is not the case here.

🛠️ Related Exercises

- Exercise_035a1b_AX
- Exercise_053

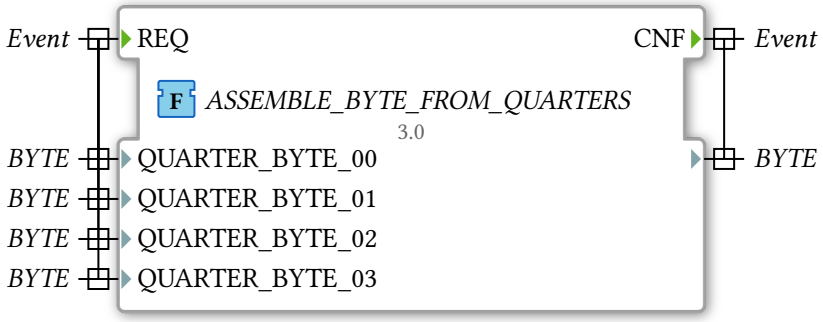
Conclusion

The `ASSEMBLE_BYTE_FROM_BOOLS` function block is a simple and efficient solution for combining eight Boolean values into one byte. Its clear interface and direct implementation make it ideal for applications that require compact processing of Boolean signals.

ASSEMBLE_BYTE_FROM_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_BYTE_FROM_QUARTERS` combines four individual quarter bytes into a complete byte. This is particularly useful in scenarios where data needs to be transmitted or processed in parts and then reassembled.

Interface Structure

Event Inputs

- `REQ`: Starts combining the quarter bytes into one byte. It is linked to the four quarter bytes (`QUARTER_BYTE_00` to `QUARTER_BYTE_03`).

Event Outputs

- `CNF`: Signals successful concatenation of the quarter bytes into a single byte.

Data Inputs

- `QUARTER_BYTE_00`: First quarter byte (least significant bits).
- `QUARTER_BYTE_01`: Second quarter byte.
- `QUARTER_BYTE_02`: Third quarter byte.
- `QUARTER_BYTE_03`: Fourth quarter byte (most significant bits).

Data Outputs

- `BYTE`: The concatenated byte resulting from the combination of the four quarter bytes.

Adapters

No adapters available.

Functionality

The function block accepts four quarter bytes and combines them into a complete byte. The individual quarter bytes are shifted to the correct position in the resulting byte using bit shift operations (`SHL`) and joined together using logical OR operations (`OR`). The precise shifts are defined by constants (`quarterconst::SHIFT_QUARTER_00` to `quarterconst::SHIFT_QUARTER_03`).

Technical Features

- Uses bit shift operations (`SHL`) and logical OR operations (`OR`) to combine the quarter bytes.
- The shift positions are defined as constants, allowing for flexible adjustment.

State Overview

1. **Initialization**: The function block is initialized and waits for the `REQ` event.
2. **Processing**: Upon receiving `REQ`, the four quarter bytes are processed and combined into the resulting byte.
3. **Acknowledgement**: After successful combination, the `CNF` event is triggered, and the resulting byte is output.

Application Scenarios

- **Data Communication**: Combining partially transmitted data into a complete byte.

- **Data Processing**: Processing data that exists in smaller units but is required as a whole.

⚖️ Comparison with Similar Blocks

- Compared to generic bit manipulation function blocks, `ASSEMBLE_BYTE_FROM_QUARTERS` offers a specialized and optimized solution for combining quarter bytes.
- Other blocks may offer similar functionality, but are often less specific and require more configuration.

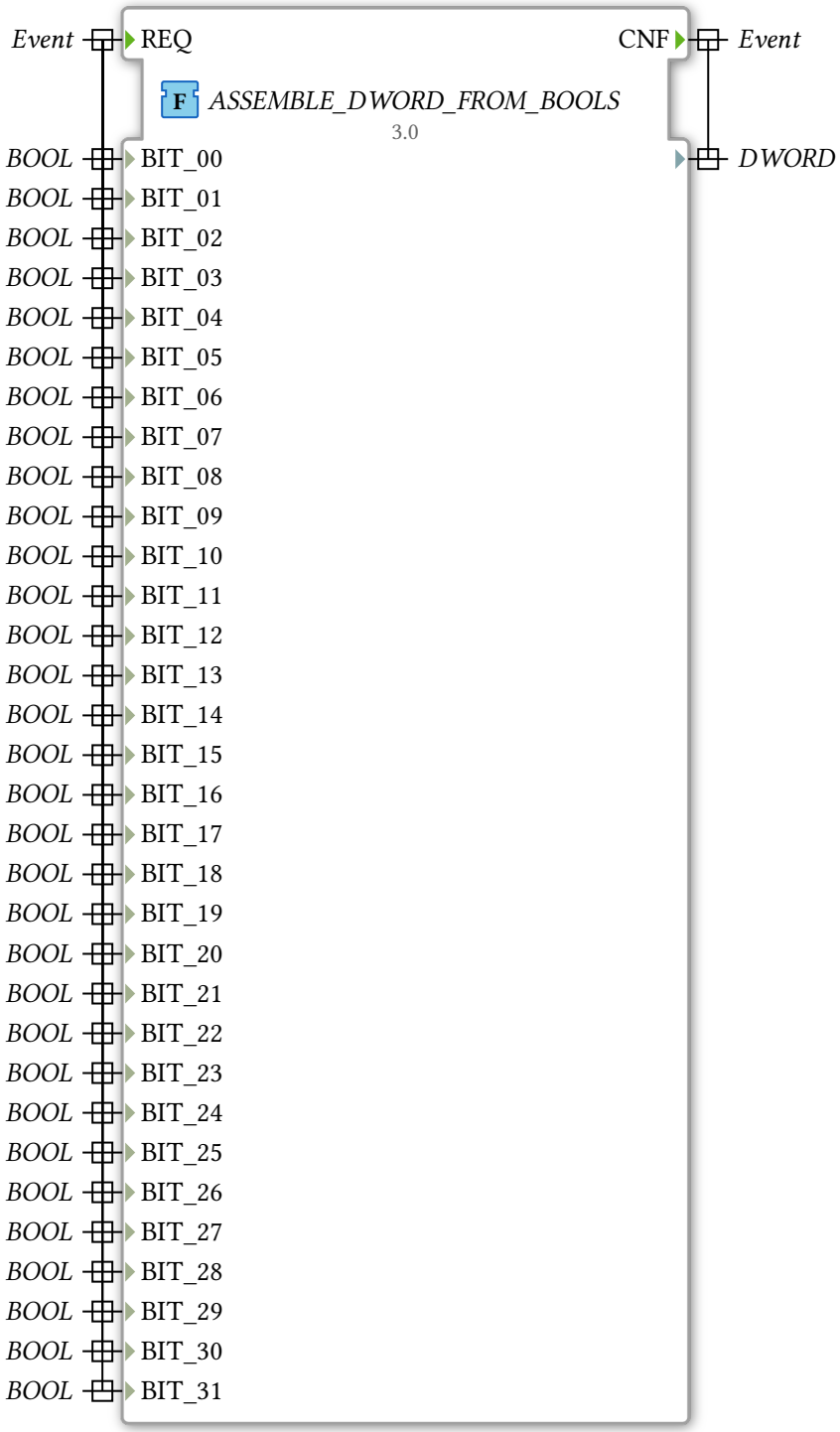
🛠️ Related Exercises

- Exercise_056

Conclusion

The `ASSEMBLE_BYTE_FROM_QUARTERS` function block is an efficient and specialized solution for combining four quarter bytes into a complete byte. By using bit-shift and logical operations, it offers a high-performance and flexible way to process data in relevant application scenarios.

ASSEMBLE_DWORD_FROM_BOOLS



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_DWORD_FROM_BOOLS` combines 32 individual BOOL values into a single DWORD value. This is particularly useful when multiple binary states need to be processed or transmitted in a compact form.

Interface Structure

Event Inputs

- `REQ`: Trigger event that starts processing the input data. It is linked to all 32 bit inputs.

Event Outputs

- `CNF`: Confirmation event that is output after successful combination of the BOOL values into a DWORD.

Data Inputs

32 BOOL inputs (`BIT_00` to `BIT_31`), each representing one bit in the resulting DWORD:

- `BIT_00`: Bit 0 (least significant bit)
- `BIT_01`: Bit 1
- ...
- `BIT_31`: Bit 31 (most significant bit)

Data Outputs

- `OUT`: DWORD composed of the 32 input BOOLs.

Adapters

No adapters included.

Functionality

When the `REQ` event is triggered, the function block combines the 32 BOOL inputs into a DWORD. Each BOOL input corresponds to one bit in the DWORD:

- `BIT_00` becomes bit 0 (LSB)
- `BIT_31` becomes bit 31 (MSB)

After successful combination, the `CNF` event is output.

Technical Features

- The function block is implemented in ST (Structured Text).
- Bit mapping is performed directly by assigning the BOOL values to the corresponding bit positions of the DWORD.
- No buffering or state storage: Processing occurs anew with each `REQ` event.

State Overview

The function block has no internal state. Processing is purely event-driven:

1. Receive `REQ` ⟶ Start processing.
2. Read BOOL values and combine them into a DWORD.
3. Output `CNF`.

Application Scenarios

- Compression of multiple binary signals for transmission via communication protocols.
- Storage of multiple switching states in a single register.
- Efficient processing of bitmasks in control algorithms.

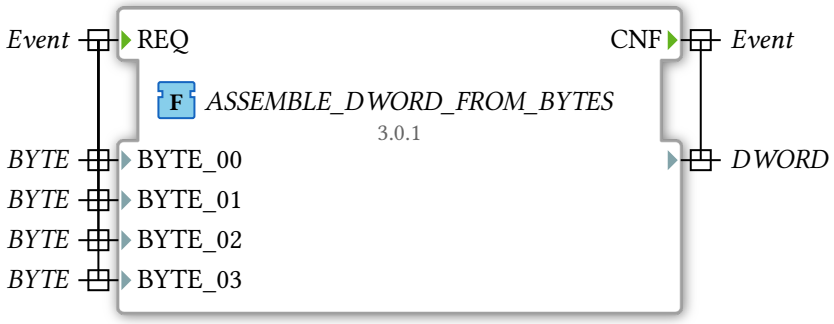
⚖️ Comparison with Similar Function Blocks

- Compared to manual bit operations in ST or other FB types, this function block offers a predefined, error-free solution for combining BOOLs into a DWORD.
- Similar function blocks might process fewer bits or offer additional functions such as bit shifts.

Conclusion

The `ASSEMBLE_DWORD_FROM_BOOLS` function block is an efficient and reliable solution for combining 32 binary inputs into a single DWORD. Its simple and clear interface makes it ideal for applications requiring compact representations of multiple binary states.

ASSEMBLE_DWORD_FROM_BYTES



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_DWORD_FROM_BYTES` combines four BYTE values into a single DWORD value. This is particularly useful when data from multiple sources needs to be merged for processing or storage as a single unit.

Interface Structure

Event Inputs

- **REQ**: Initiates the combination of the BYTE values into a DWORD. This is linked to the data inputs `BYTE_00`, `BYTE_01`, `BYTE_02`, and `BYTE_03`.

Event Outputs

- **CNF**: Signals the successful combination of the BYTE values into a DWORD. The combined DWORD value is available at the data output.

Data Inputs

- **BYTE_00**: First byte of the DWORD to be combined.
- **BYTE_01**: Second byte of the DWORD to be combined.
- **BYTE_02**: Third byte of the DWORD to be combined.
- **BYTE_03**: Fourth byte of the DWORD to be combined.

Data Outputs

- **(unnamed)**: The combined DWORD value created from the four input bytes.

Adapters

No adapters available.

Functionality

The function block accepts four BYTE values and combines them into a single DWORD value. The combination is achieved by directly assigning the individual bytes to the corresponding positions in the DWORD:

- `BYTE_00` is assigned to the least significant byte (B0) of the DWORD.
- `BYTE_01` is assigned to the next byte (B1).
- `BYTE_02` is assigned to the next byte (B2).
- `BYTE_03` is assigned to the most significant byte (B3).

Technical Features

- The function block is implemented in the ST (Structured Text) language.
- The bytes are assigned directly using the `%B0` to `%B3` syntax, which accesses the individual bytes of the DWORD.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Upon receiving `REQ`, the input bytes are combined and the DWORD value is calculated.

3. **Completed**: Sends the `CNF` event with the combined DWORD value.

Application Scenarios

- Combining data from multiple sensors or sources into a single value.
- Preprocessing data for transmission or storage.
- Use in protocol implementations where data needs to be segmented into bytes and later reassembled.

⚖️ Comparison with Similar Building Blocks

- Similar function blocks might combine individual bytes into other data types (e.g., WORD).

Unlike manual combinations in ST code, this block offers a predefined and reusable solution.

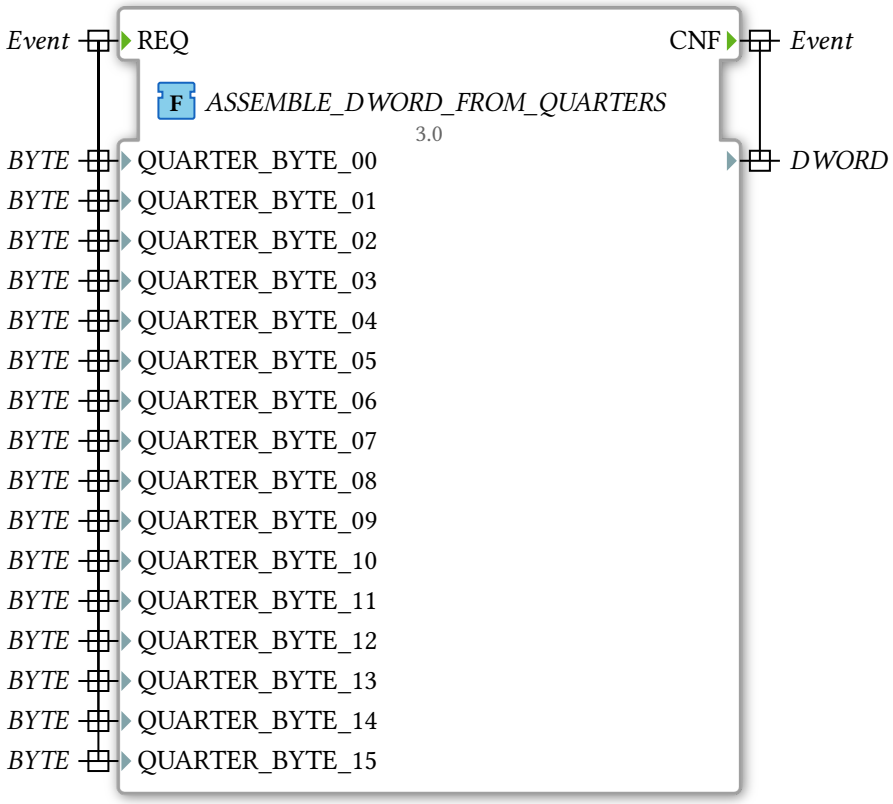
Conclusion

The `ASSEMBLE_DWORD_FROM_BYTES` function block provides a simple and efficient way to combine four byte values into a DWORD. Its clear interface and direct implementation make it a useful tool in various use cases.

ASSEMBLE_DWORD_FROM_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_DWORD_FROM_QUARTERS` combines 16 quarter bytes (4 bits each) into a DWORD (32 bits). This is particularly useful when data exists in smaller units and needs to be combined into a larger unit.

Interface Structure

Event Inputs
- `REQ`: Starts combining the quarter bytes into a DWORD. All 16 quarter bytes are combined.

Event Outputs
- `CNF`: Signals the successful combination of the quarter bytes into a DWORD.

Data Inputs
- `QUARTER_BYTE_00` to `QUARTER_BYTE_15`: 16 inputs of type `BYTE`, each containing one quarter byte (4 bits).

Data Outputs
- (Unnamed): One output of type `DWORD` containing the combined result.

Adapters
No adapters are present.

Operation
The function block accepts 16 quarter bytes and combines them into a 32-bit DWORD. Each quarter byte is shifted according to its position in the DWORD and then concatenated with the others. The shifting is performed using constants (`quarterconst::SHIFT_QUARTER_XX`) that define the position of each quarter byte in the DWORD.

Technical Features
- Uses bit shifts (`SHL`) and logical OR operations (`OR`) to correctly combine the quarter bytes.
- The conversion from `BYTE` to `DWORD` is performed using `BYTE_TO_DWORD`.

State Overview
1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Processes the quarter bytes and combines them into a DWORD.
3. **Complete**: Sends the `CNF` event and provides the combined DWORD.

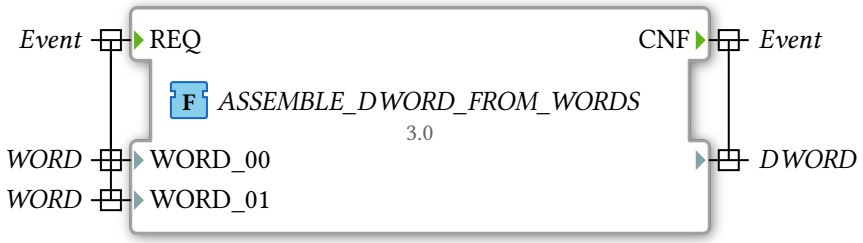
Application Scenarios
- Data compression: Combining multiple small data packets into one larger one.
- Communication protocols: Processing data transmitted in smaller units.
- Signal processing: Summarizing parts of a signal into a complete value.

⚖️ Comparison with similar building blocks
- Similar building blocks could be `BYTE_TO_WORD` or `WORD_TO_DWORD`, but these offer less flexibility as they only combine fixed values.
- `ASSEMBLE_DWORD_FROM_QUARTERS` offers more control over the position of each individual bit in the resulting DWORD.

🛠️ Related Exercises
- Exercise_060

Conclusion
The `ASSEMBLE_DWORD_FROM_QUARTERS` function block is a powerful tool for combining 16 quarter-bytes into a DWORD. Its flexible and precise operation makes it ideal for applications where data exists in smaller units and needs to be combined into a larger unit.

ASSEMBLE_DWORD_FROM_WORDS



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_DWORD_FROM_WORDS` combines two 16-bit words (WORD) into a 32-bit double word (DWORD). This functionality is particularly useful when data from two separate sources needs to be merged.

Interface Structure

Event Inputs

- `REQ`: Starts the processing of the function block. Must be linked to the data inputs `WORD_00` and `WORD_01`.

Event Outputs

- `CNF`: Signals successful processing and output of the combined DWORD.

Data Inputs

- `WORD_00`: First 16-bit word (WORD) inserted into the resulting DWORD.
- `WORD_01`: Second 16-bit word (WORD) inserted into the resulting DWORD.

Data Outputs

- (no name): The combined 32-bit double word (DWORD) consisting of the two input words.

Adapters

This function block does not use adapters.

Functionality

The function block receives the two 16-bit input words (`WORD_00` and `WORD_01`) and combines them into a single 32-bit double word (DWORD). `WORD_00` is placed in the lower 16 bits and `WORD_01` in the upper 16 bits of the resulting DWORD.

Technical Details

- The implementation is in Structured Text (ST).
- The function block is part of the `eclipse4diac::utils::assembling` package.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Processes the input words and creates the DWORD.
3. **Complete**: Sends the `CNF` event with the combined DWORD.

Application Scenarios

- Combining two 16-bit sensor data points into a 32-bit value.
- Processing data from two separate communication channels.
- Extending 16-bit data to 32 bits for further processing.

⚖️ Comparison with Similar Blocks

- Unlike simple bit operations, this function block offers a clearly defined interface and is integrated into the 4diac IDE.
- Similar functionality could be achieved with basic operations, but this block simplifies handling.

Conclusion

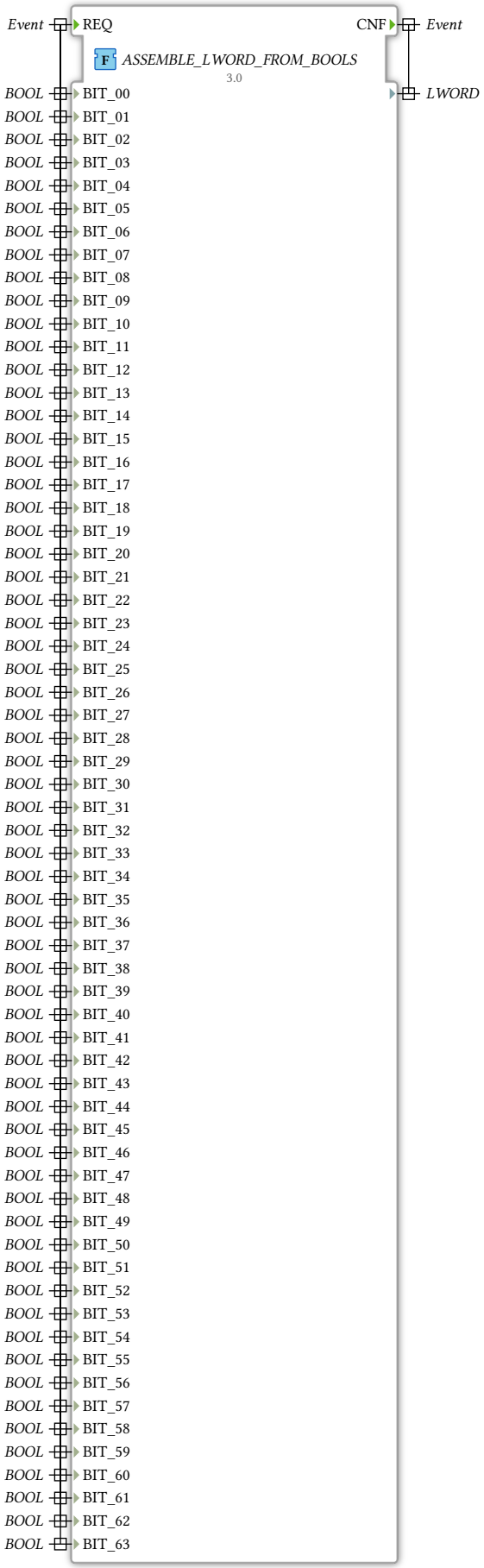
The `ASSEMBLE_DWORD_FROM_WORDS` function block provides a simple and efficient way to combine two 16-bit words into a 32-bit double word. Its clear interface and direct integration with 4diac make it particularly useful for applications that work with combined word data.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

ASSEMBLE_LWORD_FROM_BOOLS


Event
REQ
CNF
Event
F
ASSEMBLE_LWORD_FROM_BOOLS
3.0
BOOL BIT_00
BOOL BIT_01
BOOL BIT_02
BOOL BIT_03
BOOL BIT_04
BOOL BIT_05
BOOL BIT_06
BOOL BIT_07
BOOL BIT_08
BOOL BIT_09
BOOL BIT_10
BOOL BIT_11
BOOL BIT_12
BOOL BIT_13
BOOL BIT_14
BOOL BIT_15
BOOL BIT_16
BOOL BIT_17
BOOL BIT_18
BOOL BIT_19
BOOL BIT_20
BOOL BIT_21
BOOL BIT_22
BOOL BIT_23
BOOL BIT_24
BOOL BIT_25
BOOL BIT_26
BOOL BIT_27
BOOL BIT_28
BOOL BIT_29
BOOL BIT_30
BOOL BIT_31
BOOL BIT_32
BOOL BIT_33
BOOL BIT_34
BOOL BIT_35
BOOL BIT_36
BOOL BIT_37
BOOL BIT_38
BOOL BIT_39
BOOL BIT_40
BOOL BIT_41
BOOL BIT_42
BOOL BIT_43
BOOL BIT_44
BOOL BIT_45
BOOL BIT_46
BOOL BIT_47
BOOL BIT_48
BOOL BIT_49
BOOL BIT_50
BOOL BIT_51
BOOL BIT_52
BOOL BIT_53
BOOL BIT_54
BOOL BIT_55
BOOL BIT_56
BOOL BIT_57
BOOL BIT_58
BOOL BIT_59
BOOL BIT_60
BOOL BIT_61
BOOL BIT_62
BOOL BIT_63
LWORD

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_LWORD_FROM_BOOLS` combines 64 individual BOOL values into a single LWORD data type. This is particularly useful when multiple binary states need to be processed or transmitted in a compact form.

Interface Structure

Event Inputs

- `REQ`: Starts combining the 64 BOOL inputs into an LWORD.

Event Outputs

- `CNF`: Signals the successful combination of the BOOL values into an LWORD.

Data Inputs

64 BOOL inputs (`BIT_00` to `BIT_63`), each representing one bit in the resulting LWORD.

Data Outputs

- One LWORD composed of the 64 BOOL inputs.

Adapters

No adapters included.

Operation

The function block assigns each of the 64 bits of the LWORD output to a corresponding BOOL input. Upon triggering the `REQ` event, all BOOL inputs are read and inserted into the LWORD. The `CNF` event signals the completion of this process.

Technical Features

- Processes 64 BOOL inputs in parallel.
- Direct bit mapping without additional calculations.
- Efficient implementation through direct bit assignment.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Combines the BOOL values into an LWORD.
3. **Complete**: Sends the `CNF` event and outputs the combined LWORD.

Application Scenarios

- Compressing multiple binary signals for transmission.
- Processing bitmasks in control applications.
- Efficient storage of binary states.

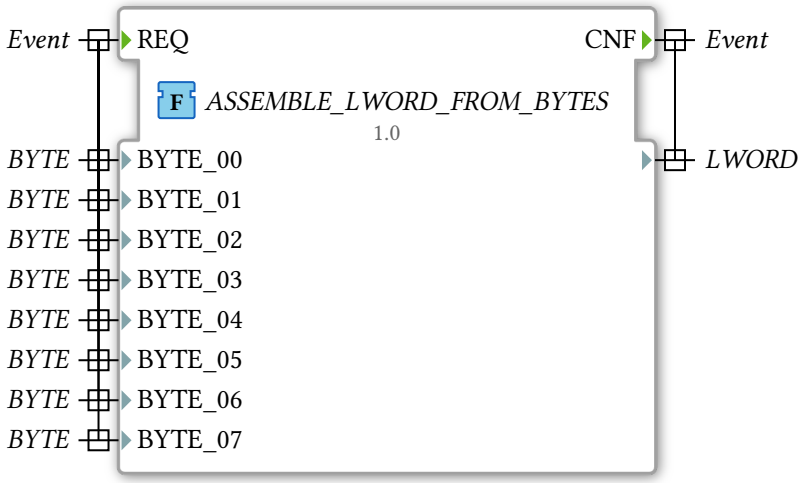
⚖️ Comparison with Similar Function Blocks

- Similar to `BOOL_TO_*` function blocks, but optimized for 64 bits.
- Offers more flexibility than hard-coded bit operations.

Conclusion

The `ASSEMBLE_LWORD_FROM_BOOLS` function block is an efficient tool for combining multiple BOOL values into a single LWORD data type, ideal for applications with high bit density.

ASSEMBLE_LWORD_FROM_BYTES



• – ◦ • – ◦ • – ◦ •

Introduction

The function `ASSEMBLE_LWORD_FROM_BYTES` combines eight BYTE values into a single LWORD value. It is useful when data from several byte sources (e.g. protocol fields) has to be assembled into a 64-bit unit.

Interface Structure

Event Inputs

- **REQ**: Initiates the combination of the BYTE values into an LWORD. Linked to `BYTE_00` to `BYTE_07`.

Event Outputs

- **CNF**: Signals the completed combination. The LWORD value is available at the data output.

Data Inputs

- **BYTE_00**: Least significant byte (B0) of the LWORD.
- **BYTE_01**: Byte B1.
- **BYTE_02**: Byte B2.
- **BYTE_03**: Byte B3.
- **BYTE_04**: Byte B4.
- **BYTE_05**: Byte B5.
- **BYTE_06**: Byte B6.
- **BYTE_07**: Most significant byte (B7) of the LWORD.

Data Outputs

- **(unnamed)**: The assembled LWORD value (return value of the function).

Adapters

No adapters available.

Functionality

The function assigns each input byte directly to the corresponding byte of the return value:

- `BYTE_00` ⟶ `%B0` (least significant)
- `BYTE_01` ⟶ `%B1`
- `BYTE_02` ⟶ `%B2`
- `BYTE_03` ⟶ `%B3`
- `BYTE_04` ⟶ `%B4`
- `BYTE_05` ⟶ `%B5`
- `BYTE_06` ⟶ `%B6`
- `BYTE_07` ⟶ `%B7` (most significant)

Technical Details

- Implemented as a function (`FUNCTION ... : LWORD`) in ST (Structured Text), package `eclipse4diac::utils::assembling`.
- Individual bytes are accessed with the partial access syntax `%B0` to `%B7`.
- No state of its own.

Application Scenarios

- Assembling 64-bit values from individual bytes, e.g. in protocol implementations or CAN data.

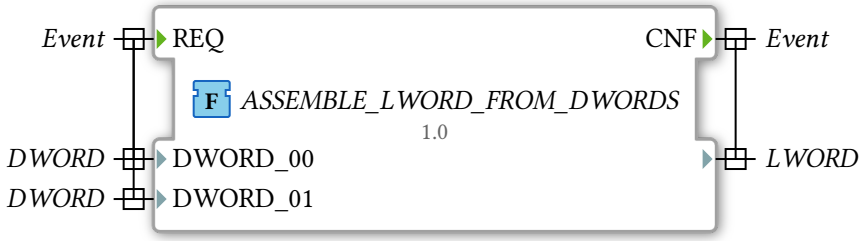
⚖ Comparison with Similar Building Blocks

- ASSEMBLE_DWORD_FROM_BYTES assembles four bytes into a DWORD, ASSEMBLE_LWORD_FROM_WORDS four words into an LWORD.

Conclusion

`ASSEMBLE_LWORD_FROM_BYTES` assembles eight BYTE values byte by byte into an LWORD.

ASSEMBLE_LWORD_FROM_DWORDS



• – ◦ • – ◦ • – ◦ •

Introduction

The function `ASSEMBLE_LWORD_FROM_DWORDS` combines two 32-bit double words (`DWORD`) into one 64-bit long word (`LWORD`).

Interface Structure

Event Inputs

- `REQ`: Starts the processing. Associated data: `DWORD_00`, `DWORD_01`

Event Outputs

- `CNF`: Signals the processing and the output of the combined `LWORD`.

Data Inputs

- `DWORD_00`: Double word 0 (`DWORD`), lowest 32 bits of the result
- `DWORD_01`: Double word 1 (`DWORD`), highest 32 bits (bits 32 to 63)

Data Outputs

- (no name): The combined 64-bit long word (`LWORD`), also the return value of the function

Adapters

This function does not use any adapters.

How it works

The function assigns the two input values to the return value via the partial access `.%D0` and `.%D1`:

```
ASSEMBLE_LWORD_FROM_DWORDS.%D0 := DWORD_00;
ASSEMBLE_LWORD_FROM_DWORDS.%D1 := DWORD_01;
```

`DWORD_00` thus forms the least significant and `DWORD_01` the most significant double word of the `LWORD`.

Technical Details

- Implemented in Structured Text (ST) as a `FUNCTION` without internal state

- Part of the package `eclipse4diac::utils::assembling`
- The inverse function is `SPLIT_LWORD_INTO_DWORDS`
- Version 1.0 (2026-05-16)

Application Scenarios

- Assembling a 64-bit value from two 32-bit registers (e.g. fieldbus data)
- Generating 64-bit bit patterns from two double words

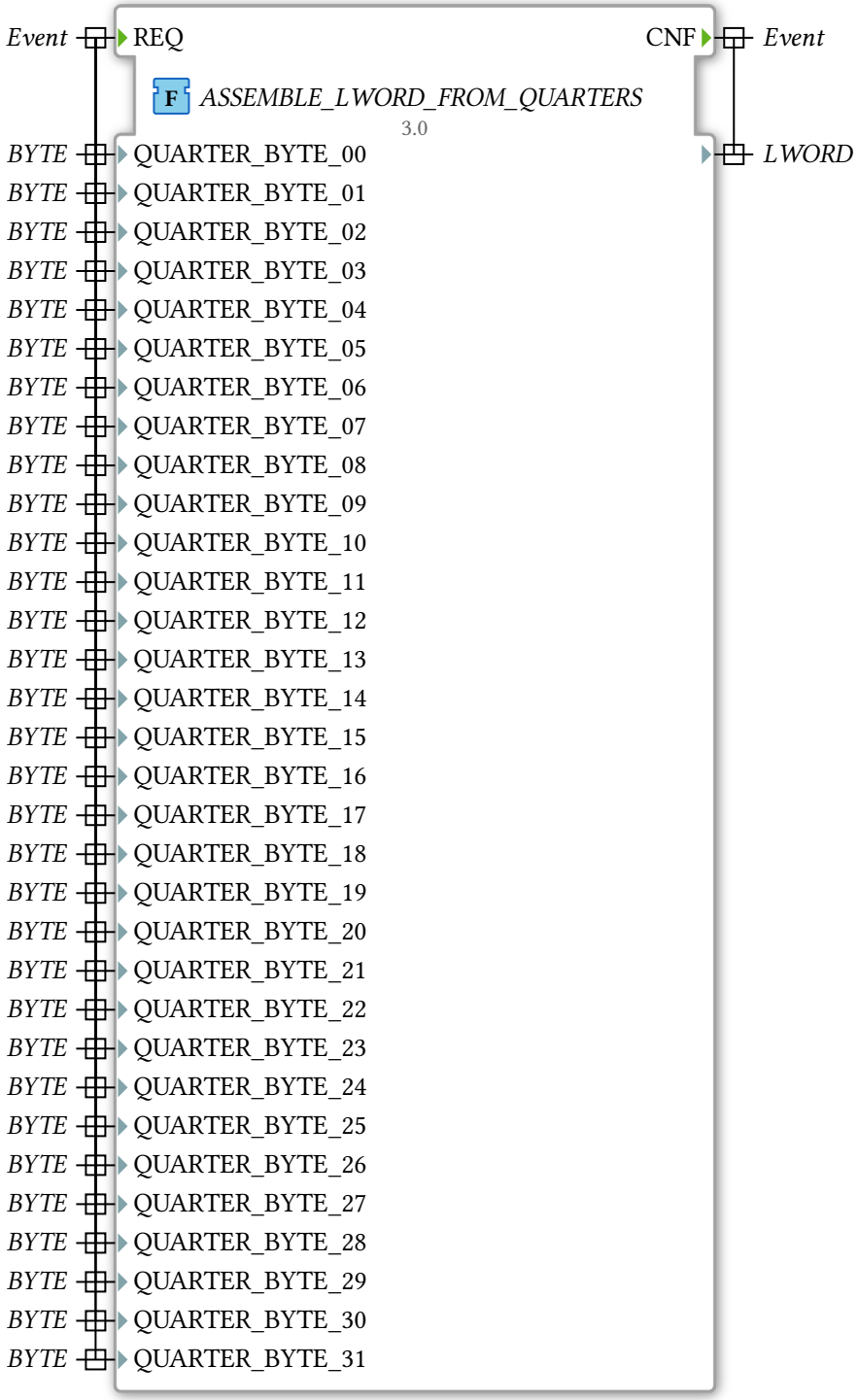
Summary

`ASSEMBLE_LWORD_FROM_DWORDS` assembles two `DWORD` values, starting with the least significant one, into an `LWORD`.

ASSEMBLE_LWORD_FROM_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦

Introduction

The function block `ASSEMBLE_LWORD_FROM_QUARTERS` combines 32 byte values (QUARTER_BYTE_00 to QUARTER_BYTE_31) into a 64-bit LWORD value. This is particularly useful when data is in smaller units and needs to be combined into a larger data word.

Interface Structure

Event Inputs

- `REQ`: Starts the processing and combines the 32 byte inputs into an LWORD.

Event Outputs

- `CNF`: Signals the successful merging of the bytes into an LWORD.

Data Inputs

- `QUARTER_BYTE_00` to `QUARTER_BYTE_31` (32 inputs of type `BYTE`): The individual byte values that are combined into an LWORD.

Data Outputs

- (Unnamed, type `LWORD`): The combined 64-bit value that is created from the 32 byte inputs.

Adapters

No adapters are present.

Functionality

The function block receives the 32 byte inputs and combines them into a 64-bit LWORD. Each byte value is shifted to a specific position in the LWORD (using the `SHL` operation) and then logically combined with the current LWORD value (`OR` operation). The shifting is performed according to predefined constants (`quarterconst::SHIFT_QUARTER_XX`) that define the exact position of each byte in the LWORD.

Technical Features

- Uses bitwise operations (`SHL`, `OR`) for efficient byte combination.
- The combination order ranges from `QUARTER_BYTE_31` (most significant byte) to `QUARTER_BYTE_00` (least significant byte).
- The function block is implemented in ST (Structured Text) and uses a library (`quarterconst`) for the shift constants.

State Overview

The function block has no internal states. Processing occurs directly upon receiving the `REQ` event, and the result is immediately output via `CNF`.

Application Scenarios

- Reassembling data that has been transmitted or stored in smaller units (bytes).
- Processing protocols or data formats that transmit large values in segmented form.
- Use in systems that work with 64-bit data but only have 8-bit interfaces available.

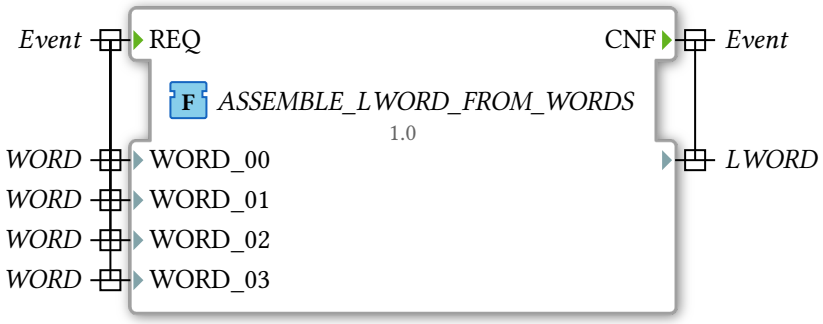
⚖️ Comparison with Similar Blocks

- Similar function blocks could be... For example, `ASSEMBLE_DWORD_FROM_BYTES`, which only combines 32-bit values.
- Unlike manual implementations, this function block offers a standardized and optimized solution for combining 32 bytes into a 64-bit LWORD.

Conclusion

The `ASSEMBLE_LWORD_FROM_QUARTERS` function block provides an efficient and reliable method for combining 32-byte values into a 64-bit LWORD. By using bitwise operations and predefined shift constants, the solution is performant and easily integrated into existing systems. Ideal for applications that work with large, segmented datasets.

ASSEMBLE_LWORD_FROM_WORDS



• – ◦ • – ◦ • – ◦ •

Introduction

The function `ASSEMBLE_LWORD_FROM_WORDS` combines four 16-bit words (`WORD`) into one 64-bit long word (`LWORD`).

Interface Structure

Event Inputs

- `REQ`: Starts the processing. Associated data: `WORD_00`, `WORD_01`, `WORD_02`, `WORD_03`

Event Outputs

- `CNF`: Signals the processing and the output of the combined `LWORD`.

Data Inputs

- `WORD_00`: Word 0 (`WORD`), lowest 16 bits of the result
- `WORD_01`: Word 1 (`WORD`), bits 16 to 31
- `WORD_02`: Word 2 (`WORD`), bits 32 to 47
- `WORD_03`: Word 3 (`WORD`), highest 16 bits (bits 48 to 63)

Data Outputs

- (no name): The combined 64-bit long word (`LWORD`), also the return value of the function

Adapters

This function uses no adapters.

How it works

The function assigns the four input words to the return value via the partial access `.%W0` to `.%W3`:

```
ASSEMBLE_LWORD_FROM_WORDS.%W0 := WORD_00;
ASSEMBLE_LWORD_FROM_WORDS.%W1 := WORD_01;
ASSEMBLE_LWORD_FROM_WORDS.%W2 := WORD_02;
ASSEMBLE_LWORD_FROM_WORDS.%W3 := WORD_03;
```

`WORD_00` thus forms the least significant and `WORD_03` the most significant word of the `LWORD`.

Technical Details

- Implemented in Structured Text (ST) as a `FUNCTION` without internal state
- Part of the package `eclipse4diac::utils::assembling`
- The inverse is `SPLIT_LWORD_INTO_DWORDS` or the respective `SPLIT_*` functions in the package `splitting`

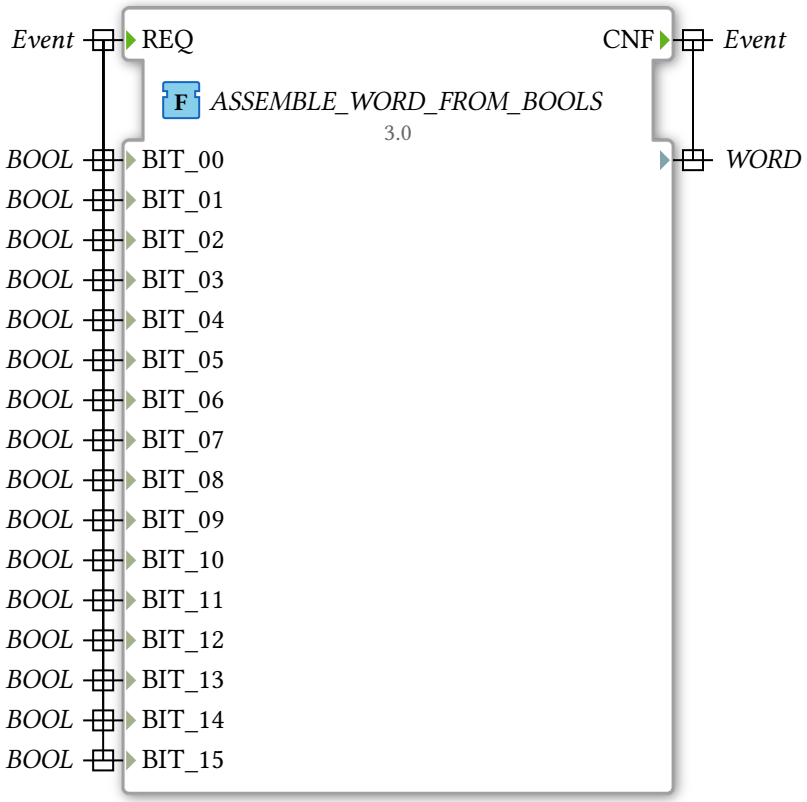
Application Scenarios

- Assembling a 64-bit value from four 16-bit registers (e.g. fieldbus/Modbus data)
- Creating 64-bit bit patterns from individual words

Summary

`ASSEMBLE_LWORD_FROM_WORDS` assembles four `WORD` values, starting with the least significant, into an `LWORD`.

ASSEMBLE_WORD_FROM_BOOLS



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_WORD_FROM_BOOLS` combines 16 Boolean input values into a single WORD output value. Each Boolean input represents a single bit in the resulting WORD.

Interface Structure

Event Inputs

- `REQ`: Starts processing and takes all current values from the Boolean inputs.

Event Outputs

- `CNF`: Signals the successful combination of the Boolean values into a WORD.

Data Inputs

16 Boolean inputs (BIT_00 to BIT_15), each corresponding to one bit in the resulting WORD:

- `BIT_00` to `BIT_15` (BOOL): Individual bits for the WORD to be assembled

Data Outputs

- `(Unbenannt)` (WORD): The resulting WORD, assembled from the 16 Boolean inputs.

Adapters

No adapters are available.

Operation

When the `REQ` event is triggered, all 16 Boolean input values are read and combined into a 16-bit WORD. Here, the following correspond to:

- `BIT_00` corresponds to bit 0 (LSB)
- `BIT_15` corresponds to bit 15 (MSB)

The assignment is performed directly via bitwise operations, where each Boolean input value is mapped to the corresponding bit in the WORD.

Technical Features

- The bit order is fixed (BIT_00 = LSB, BIT_15 = MSB)
- The block does not perform any additional calculations or transformations
- The operation is atomic and occurs in one step

State Overview

1. **Idle**: Waits for the `REQ` event
2. **Processing**: Processes the inputs and assembles the WORD
3. **Output**: Sends `CNF` with the resulting WORD

Application Scenarios

- Combining individual switching states into a compact data word
- Preparing data for communication with devices that expect WORD data
- Bitwise control of hardware registers

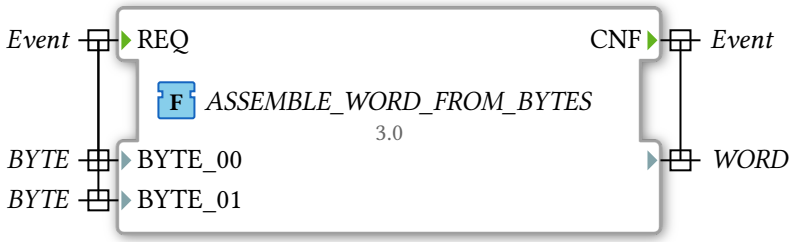
⚖️ Comparison with Similar Function Blocks

- Unlike general combination blocks, this function block is specifically optimized for converting BOOL to WORD
- Offers a more direct and efficient Solution than using multiple individual blocks
- Similar to BIT_TO_* blocks, but with a fixed 16-bit WORD output

Conclusion

The `ASSEMBLE_WORD_FROM_BOOLS` function block provides an efficient and direct method to convert 16 Boolean signals into a single WORD. Its simple and deterministic operation makes it particularly suitable for applications where individual bits need to be combined into compact data words.

ASSEMBLE_WORD_FROM_BYTES



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `ASSEMBLE_WORD_FROM_BYTES` combines two byte values into a single word (WORD). This is particularly useful when data from two separate byte sources needs to be concatenated into a single 16-bit value.

Interface Structure

Event Inputs

- **REQ**: Starts processing. The input data `BYTE_00` and `BYTE_01` are passed with this event.

Event Outputs

- **CNF**: Signals successful merging of the bytes into a WORD.

Data Inputs

- **BYTE_00**: The least significant byte (LSB) of the resulting WORD.
- **BYTE_01**: The most significant byte (MSB) of the resulting WORD.

Data Outputs

- **(no name)**: The resulting WORD, composed of `BYTE_00` (LSB) and `BYTE_01` (MSB).

Adapters

No adapters are used.

Functionality

The function block combines the two input bytes `BYTE_00` and `BYTE_01` into a 16-bit WORD. The least significant byte (LSB) is `BYTE_00` and the most significant byte (MSB) is `BYTE_01`. The operation occurs immediately after the `REQ` event arrives, and the result is output with the `CNF` event.

Technical Features

- The concatenation is performed directly by assigning the bytes to the corresponding positions in the WORD.
- The block is optimized for use in real-time systems and has a deterministic runtime.

State Overview

1. **Idle**: Waits for the `REQ` event.
2. **Processing**: Processes the input data and creates the WORD.
3. **Output**: Sends the `CNF` event along with the resulting WORD.

Application Scenarios

- Combining sensor values stored as separate bytes.
- Processing communication protocols where data is transmitted in byte pairs.
- General data manipulation in control systems.

⚖️ Comparison with Similar Blocks

- Compared to generic computing blocks, `ASSEMBLE_WORD_FROM_BYTES` is specialized and more efficient for this specific task.
- Similar blocks might offer additional features such as byte swapping or error handling, but they are more complex.

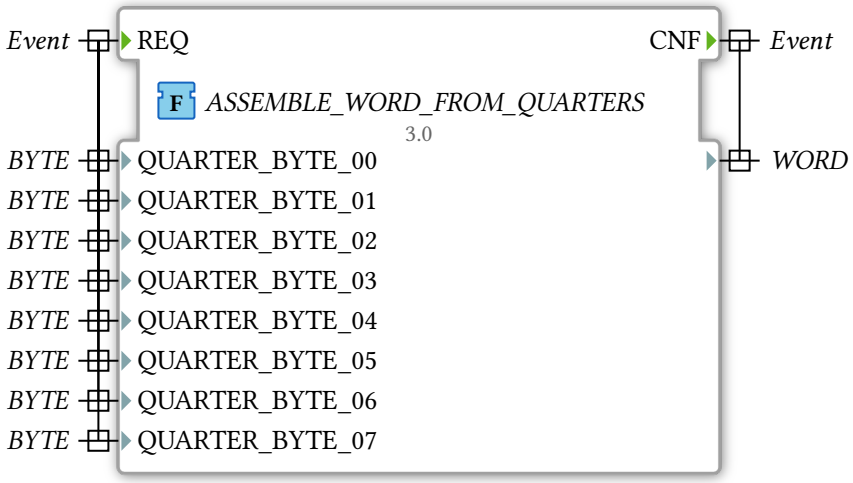
Conclusion

The `ASSEMBLE_WORD_FROM_BYTES` function block is a simple and efficient block for combining two bytes into a WORD. It is particularly suitable for applications where performance and simplicity are paramount.

ASSEMBLE_WORD_FROM_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Introduction

The function block ASSEMBLE_WORD_FROM_QUARTERS combines 8 quarter bytes into a WORD. This is particularly useful when data is in smaller units and needs to be combined into a larger unit.

Interface Structure

Event Inputs

- REQ: Starts the processing and combines the 8 quarter bytes into a WORD.

Event Outputs

- CNF: Signals the successful combination of the quarter bytes into a WORD.

Data Inputs

- QUARTER_BYTE_00 to QUARTER_BYTE_07 (Type: BYTE): The 8 quarter bytes that are combined to form a WORD.

Data Outputs

- OUT (Type: WORD): The resulting WORD composed of the 8 quarter bytes.

Adapters

No adapters are available.

Functionality

The function block receives 8 quarter bytes and combines them into a WORD. Each quarter byte is moved to a specific position within the WORD and concatenated using logical OR operations. The exact shifts are defined by constants (quarterconst::SHIFT_QUARTER_00 to quarterconst::SHIFT_QUARTER_07).

Technical Features

- Uses bitwise operations (SHL and OR) to combine the bytes.
- The shift positions are defined by constants, allowing for flexible adjustment.

State Overview

1. **Idle**: Waits for the REQ event.
2. **Processing**: Combines the quarter bytes into a WORD.
3. **Complete**: Sends the CNF event along with the resulting WORD.

Application Scenarios

- Data compression: Combining smaller data units into larger ones.
- Communication protocols: Composing data packets from individual bytes.
- Signal processing: Combining signal segments into a complete signal.

⚖️ Comparison with similar function blocks

- Similar function blocks could be BYTE_TO_WORD or SHL; however, ASSEMBLE_WORD_FROM_QUARTERS offers a specialized function for combining eight quarter bytes into a WORD.
- Unlike generic combination function blocks, this function block is optimized for the specific requirement of combining quarter bytes.

Conclusion

The ASSEMBLE_WORD_FROM_QUARTERS function block is an efficient tool for combining eight quarter bytes into a WORD. By using bitwise operations and defined shift positions, it provides a reliable and flexible solution for this specific task.

2.11.12.19 const

quarterconst

🎧 Podcast

- QUARTER

• – ◦ • – ◦ • – ◦ •

Introduction

Global constants for quarter values.

Metadata

Attribute	Value
License	EPL-2.0
Version	3.0 (2025-04-14, Patrick Aigner)
4diac package	eclipse4diac::utils

–

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

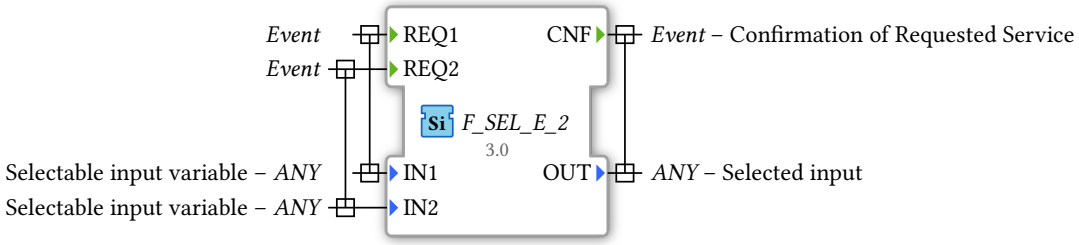
2.11.12.20 selection

F_SEL_E_2

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SEL_E_2` is an event-based selector that selects between two input variables. It belongs to the category of standard selection functions according to IEC 61131-3 and is typically used in control applications where conditional selection between two input signals is required.



Interface Structure

Event Inputs

- `REQ0`: Event input that triggers the selection of the first input variable (`IN0`).
- `REQ1`: Event input that triggers the selection of the second input variable (`IN1`).

Event Outputs

- `CNF`: Confirmation event that is output after an input variable is selected. This event is linked to the output variable `OUT`.

Data Inputs

- `IN0`: First selectable input variable (Type: `ANY`).
- `IN1`: Second selectable input variable (Type: `ANY`).

Data Outputs

- `OUT`: Selected input variable (type: `ANY`). The value corresponds to either `IN0` or `IN1`, depending on the triggering event (`REQ0` or `REQ1`).

Adapters

- This function block has no adapters.

Functionality

The function block `F_SEL_E_2` selects between two input variables based on the incoming events:

- If the event `REQ0` is triggered, the value of `IN0` is passed to `OUT`.
- When the event `REQ1` is triggered, the value of `IN1` is passed to `OUT`.
- After selection, the confirmation event `CNF` is issued to signal successful selection.

Technical Features

- The function block supports the data type `ANY`, meaning it is compatible with various data types.
- Selection is exclusively event-driven; that is, the logical condition for selection is controlled by external events.

State Overview

1. **Initial State**: The function block waits for an event (`REQ0` or `REQ1`).
2. **Selection State**:

- `REQ0` selects `IN0`.
- `REQ1` selects `IN1`.

1. **Confirmation State**: After selection, `CNF` is output, and the block returns to its initial state.

Application Scenarios

- Selection between two sensor data points based on external conditions.
- Switching between two operating modes in a controller.
- Dynamic selection of input signals depending on process events.

⚖️ Comparison with Similar Components

- Compared to other selection blocks (e.g., `SEL` in IEC 61131-3), `F_SEL_E_2` is purely event-driven and does not require any additional logical condition for selection.
- Similar blocks, such as `MUX` (multiplexer), allow selection from multiple inputs but are often data-flow driven and not event-based.

🛠️ Related Exercises

- Exercise_019
- Exercise_074

Conclusion

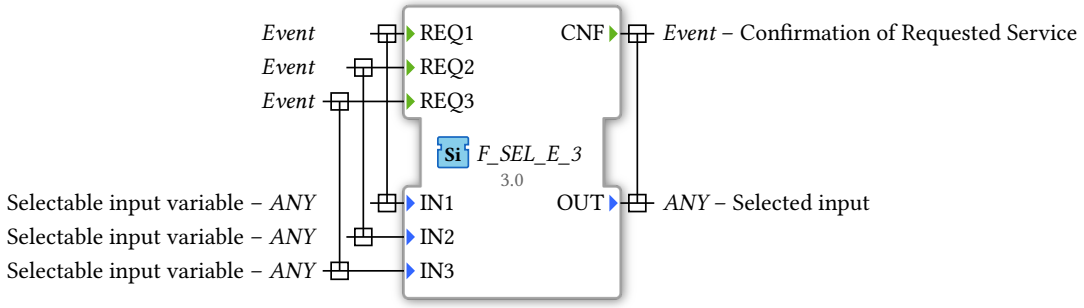
The `F_SEL_E_2` is a flexible and easy-to-use function block for applications requiring event-based selection between two input variables. Its compatibility with the `ANY` data type makes it particularly versatile.

F_SEL_E_3

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `F_SEL_E_3` is an event-based selector that selects one of three input variables based on the triggering event and passes it to the output. It is part of the `eclipse4diac::utils::selection` package and complies with the IEC 61131-3 standard for standard selection functions.



Interface Structure

Event Inputs

- `REQ0`: Selects the input variable `IN0`
- `REQ1`: Selects the input variable `IN1`
- `REQ2`: Selects the input variable `IN2`

Event Outputs

- `CNF`: Confirms the execution of the selection and returns the selected value

Data Inputs

- `IN0`: First selectable input variable (Type: ANY)
- `IN1`: Second selectable input variable (Type: ANY)
- `IN2`: Third selectable input variable (Type: ANY)

Data Outputs

- `OUT`: Selected input variable (Type: ANY)

Adapters

No adapters available.

Functionality

The function block waits for an incoming event (`REQ0`, `REQ1`, or `REQ2`). Upon the occurrence of such an event, the corresponding input variable (`IN0`, `IN1`, or `IN2`) is selected and forwarded to the output `OUT`. Simultaneously, the acknowledgment event `CNF` is triggered.

When such an event occurs, the corresponding input variable (`IN0`, `IN1`, or `IN2`) is selected and forwarded to the output `OUT`.

Technical Features

- Supports the ANY data type for maximum flexibility
- Event-based triggering for deterministic behavior
- Simple and direct selection without additional logic

State Overview

1. Waiting state: No active event
2. Selection state: Event received, corresponding input selected
3. Acknowledgement state: Output updated and CNF event triggered

Application Scenarios

- Selection between different sensor data streams
- Switching between operating modes
- Multiplexing of data sources in control systems
- Implementation of priority logic

⚖️ Comparison with Similar Function Blocks

- Unlike `F_SEL_E_2`, this function block offers three inputs instead of two
- Similar to `F_MUX`, but with event-based instead of data-driven selection
- More complex than simple gates, but more specialized for selection tasks

🛠️ Related Exercises

- Exercise_015a
- Exercise_016a

Conclusion

The `F_SEL_E_3` is an efficient and flexible component for applications that require event-driven switching between three different data sources. Its simple interface and support for any data type make it a versatile tool in automation technology.

🌐 Related topic subpages on ms-muc-docs.de

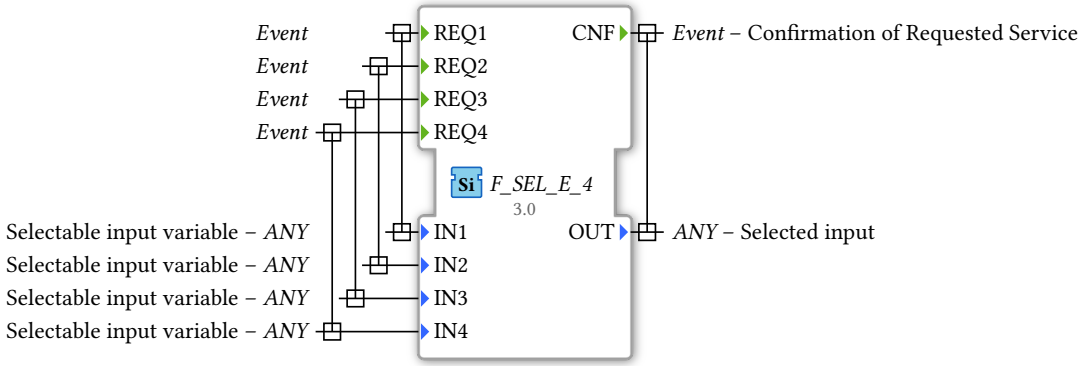
- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

F_SEL_E_4

• – ◦ • – ◦ • – ◦ •

Introduction

The F_SEL_E_4 function block is an event-based selector that selects one input from up to four inputs and forwards it to the output. It is classified according to the IEC 61131-3 standard and serves as a standard selection function in automation systems.



Interface Structure

Event Inputs

- **REQ0**: Event that selects the value of IN0.
- **REQ1**: Event that selects the value of IN1.
- **REQ2**: Event that selects the value of IN2.
- **REQ3**: Event that selects the value of IN3.

Event Outputs

- **CNF**: Confirmation event triggered as soon as an input is selected and the value is passed to OUT.

Data Inputs

- **IN0**: First selectable input (Type: ANY).
- **IN1**: Second selectable input (Type: ANY).
- **IN2**: Third selectable input (Type: ANY).
- **IN3**: Fourth selectable input (Type: ANY).

Data Outputs

- **OUT**: Selected input (Type: ANY).

Adapters

No adapters available.

Functionality

Based on the triggering event (REQ0 to REQ3), F_SEL_E_4 selects the corresponding input (IN0 to IN3) and passes the value to OUT. The CNF event confirms the successful selection and transmission.

Technical Features

- Supports the ANY data type, enabling flexible use with various data types.
- Event-based selection, allowing for precise control.

State Overview

The function block has no internal states. Selection and transmission occur immediately upon the arrival of the corresponding REQ event.

Application Scenarios

- Selection between different sensor data in a controller.
- Multiplexing of control signals in automation systems.
- Flexible data transmission depending on external events.

⚖️ Comparison with Similar Function Blocks

- Compared to other selectors, F_SEL_E_4 offers event-based control, enabling more precise selection than time- or state-based alternatives.
- Support for the ANY data type makes it more flexible than type-specific selectors.

🛠️ Related Exercises

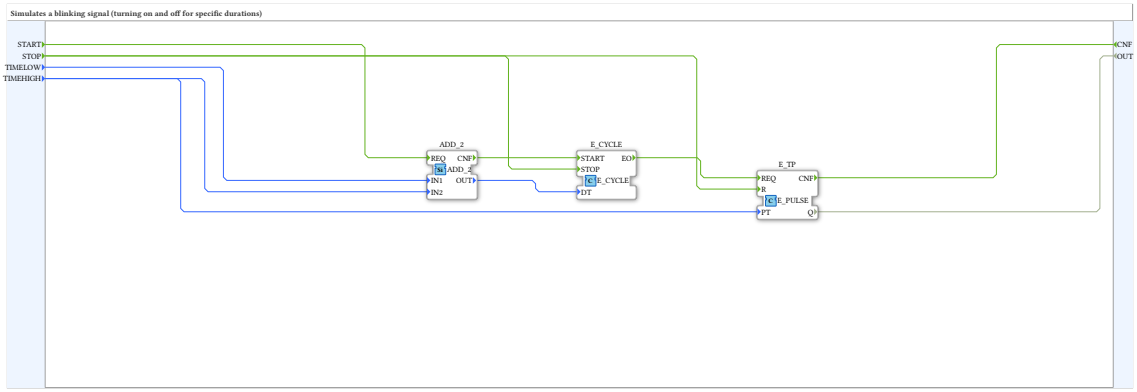
- Exercise_019a
- Exercise_019b
- Exercise_019c

Conclusion

The F_SEL_E_4 is a versatile and flexible function block for event-based data selection in automation systems. Its simple interface and support for various data types make it a practical solution for many use cases.

2.11.12.21 signals

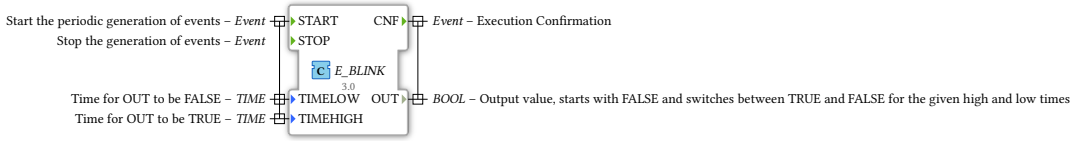
E_BLINK



• – ○ • – ○ • – ○ •

Introduction

The E_BLINK function block simulates a flashing signal that alternates between the states TRUE and FALSE. The duration of each state can be configured individually. Unlike similar implementations, this function block always returns to FALSE upon the STOP event.



Interface Structure

Event Inputs

- **START**: Starts the periodic generation of events. Uses the data `TIMELOW` and `TIMEHIGH`.
- **STOP**: Stops the generation of events and sets the output to FALSE.

Event Outputs

- **CNF**: Acknowledges execution and returns the current output value `OUT`.

Data Inputs

- **TIMELOW**: Time in milliseconds during which the output is FALSE.
- **TIMEHIGH**: Time in milliseconds during which the output is TRUE.

Data Outputs

- **OUT**: The output value that toggles between TRUE and FALSE.

Adapters

No adapters available.

Functionality

The E_BLINK function block internally uses an E_PULSE timer and an E_CYCLE generator to implement the periodic state changes. During the START event, the times for TRUE and FALSE are added and used as the cycle time for E_CYCLE. The E_PULSE timer then controls the duration of the TRUE state, while the time difference determines the FALSE state.

Technical Features

- The function block always resets the output to FALSE upon the STOP event.
- The implementation differs from similar solutions like CODESYS BLINK, which retain the last state.

State Overview

1. **Inactive**: Output is FALSE.
2. **Active**: Output toggles periodically between TRUE and FALSE.
3. **Stopped**: Output is set to FALSE.

Application Scenarios

- Controlling flashing warning lights
- Generating clock signals for testing purposes
- Visualizing state changes in user interfaces

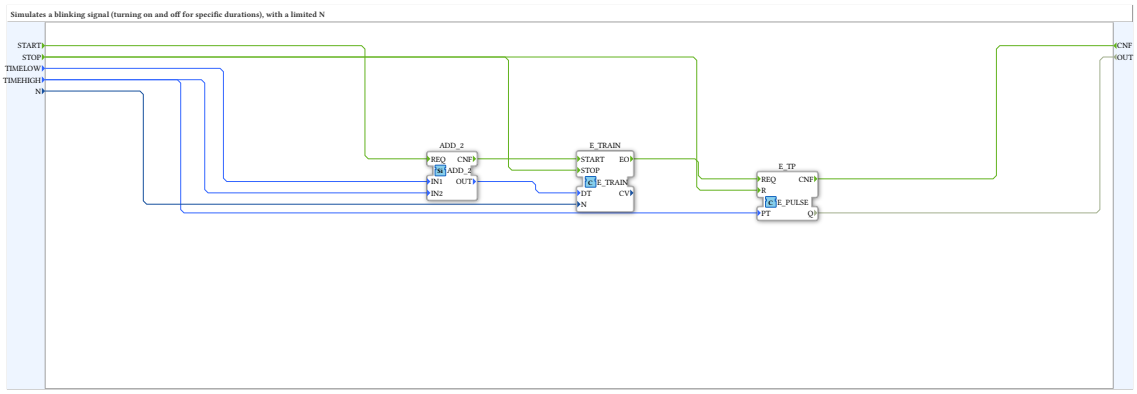
⚖️ Comparison with Similar Function Blocks

Compared to the CODESYS BLINK function block, E_BLINK offers simplified handling due to the automatic reset to FALSE upon stopping. The functionality is similar, but the behavior during a STOP event is more consistent.

Conclusion

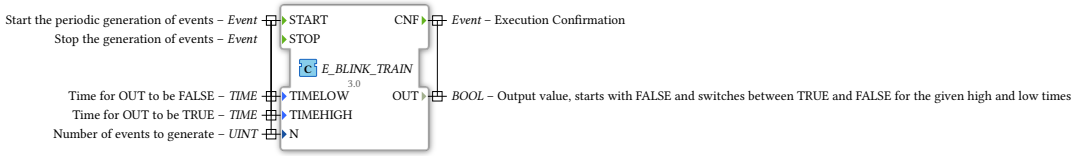
The E_BLINK function block offers a simple and reliable way to generate periodic signals with configurable on and off times. The clear definition of the STOP behavior makes it particularly suitable for safety-critical applications.

E_BLINK_TRAIN



• – ○ • – ○ • – ○ •

The function block `E_BLINK_TRAIN` simulates a flashing signal that is switched on and off for specific time intervals. The number of flash cycles can be limited. Unlike similar flashing function blocks, this FB always returns to the state `FALSE` when stopped.



- `START`: Starts the periodic generation of events. Linked to the data inputs `TIMELOW`, `TIMEHIGH`, and `N`.
- `STOP`: Stops event generation.
- `CNF`: Acknowledges execution and returns the current output value `OUT`.
- `TIMELOW` (TIME): Time during which `OUT` has the value `FALSE`.
- `N` (UINT): Number of blink cycles to generate.
- `OUT` (BOOL): Output value that alternates between `TRUE` and `FALSE`. Starts with `FALSE`.

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Functionality

1. Upon a `START` event, the sum of `TIMELOW` and `TIMEHIGH` is calculated and used as the period for the blink cycle.
2. The function block generates `N` blink cycles, where `OUT` is set to `TRUE` for `TIMEHIGH` and to `FALSE` for `TIMELOW`.
3. Upon a `STOP` event, generation is terminated and `OUT` is reset to `FALSE`.
4. Each state change is confirmed via the `CNF` event.

- Uses internal function blocks `E_PULSE` (for timing) and `E_TRAIN` (for cycle limiting).
- The period duration is determined by adding `TIMELOW` and `TIMEHIGH`.
- Unlike some other blink function blocks, `FALSE` is always output upon stopping.

1. **Inactive**: `OUT` = `FALSE` (initial state or after `STOP`).
2. **Active**: Generates blink cycles according to the set times and number of cycles.

- `OUT` toggles between `TRUE` (`TIMEHIGH`) and `FALSE` (`TIMELOW`).
- Display of warning or status lights with a defined blink frequency.
- Control of periodic signals in industrial automation processes.
- Simulation of blink patterns for testing purposes.
- **CODESYS BLINK FB**: Retains the last state when stopped, while `E_BLINK_TRAIN` always reverts to `FALSE`.
- **Simple Timer FBs**: Do not offer an integrated limit on the number of cycles (`N`).
- Exercise_035a3

The `E_BLINK_TRAIN` is a versatile function block for generating blinking signals with configurable times and a limited number of cycles. Its unique feature is that it reverts to `FALSE` when stopped, making it ideal for safety-critical applications. The clear interface structure and the internal use of proven timer FBs make it reliable and easy to use.

Technical Features

State Overview

Application Scenarios

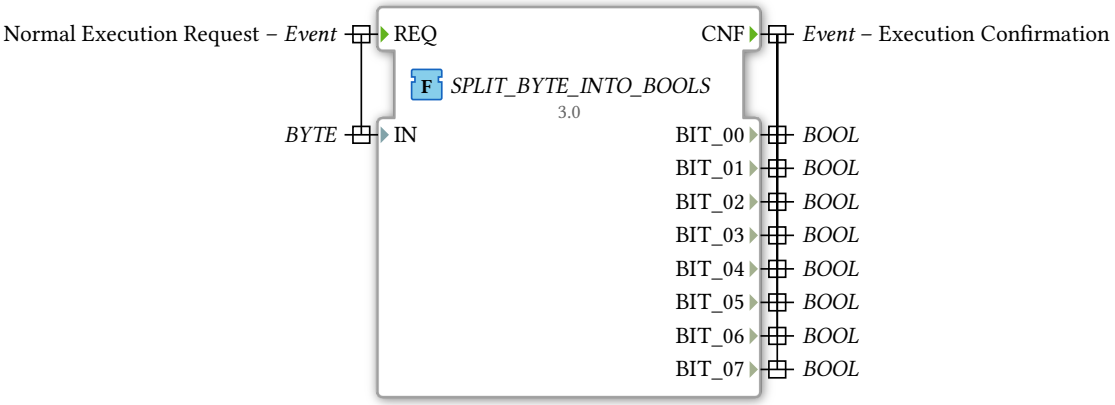
⚖️ Vergleich mit ähnlichen Bausteinen

🛠️ Zugehörige Übungen

Conclusion

2.11.12.22 splitting

SPLIT_BYTE_INTO_BOOLS



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_BYTE_INTO_BOOLS` is used to split a byte into its individual bits and output them as separate BOOL values. This is particularly useful when working with hardware that evaluates or sets individual bits of a byte.

Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. When this event is triggered, the bits of the current byte value are evaluated at `IN`.

Event Outputs

- **CNF**: Confirms the execution of the function block. This event is triggered after all bits have been evaluated and forwarded to the corresponding outputs.

Data Inputs

- **IN**: The byte to be split into its individual bits. The data type is `BYTE`.

Data Outputs

- **BIT_00** to **BIT_07**: The individual bits of the byte, where `BIT_00` represents the least significant bit (LSB) and `BIT_07` represents the most significant bit (MSB). The data type of all outputs is `BOOL`.

Adapters

This function block does not use adapters.

Operation

This function block splits the incoming byte `IN` into its 8 bits and assigns them to the corresponding BOOL outputs. Each bit is extracted directly from the byte and output as a separate BOOL value. Evaluation occurs with each `REQ` event.

Technical Features

- Bit extraction is performed directly via the access notation `IN.%X0` to `IN.%X7`, enabling efficient and fast processing.
- The function block is implemented in the ST (Structured Text) programming language.

State Overview

The function block has no internal states. Execution occurs immediately upon receiving the `REQ` event, and the results are immediately available via the `CNF` event and the data outputs.

Application Scenarios

- **Hardware Control**: When controlling digital inputs and outputs organized as bytes, individual bits can be easily evaluated.
- **Data Processing**: When processing byte data where individual bits have a specific meaning (e.g., status flags).

⚖️ Comparison with Similar Blocks

- Compared to generic bit manipulation functions, this block offers a specialized and simple solution for decomposing a byte into bits.
- Other blocks might require more complex operations such as bit shifts or masks, while this block enables direct bit extraction.

🛠️ Related Exercises

- Exercise_053

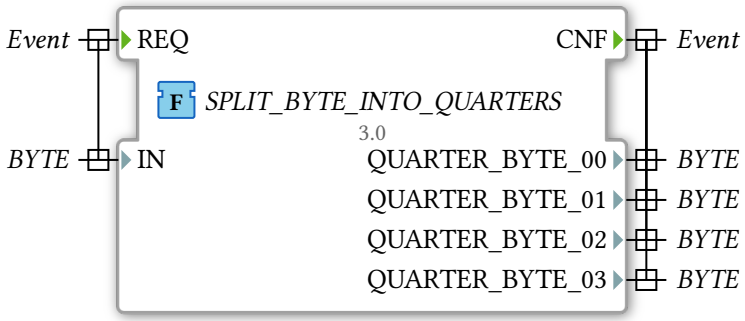
Conclusion

The `SPLIT_BYTE_INTO_BOOLS` function block is a simple and efficient tool for decomposing a byte into its individual bits. It is particularly suitable for applications requiring fast and direct bit extraction. Its clear interface and simple operation make it easy to integrate into existing systems.

SPLIT_BYTE_INTO_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_BYTE_INTO_QUARTERS` is used to split a byte into four quarter bytes. This function is particularly useful when individual parts of a byte need to be processed separately, for example, when decoding data or manipulating bits.

Interface Structure

Event Inputs

- `REQ`: Starts the processing of the input byte. It is linked to the data input `IN`.

Event Outputs

- `CNF`: Signals the completion of processing and provides the four quarter bytes. It is linked to the data outputs `QUARTER_BYTE_00`, `QUARTER_BYTE_01`, `QUARTER_BYTE_02`, and `QUARTER_BYTE_03`.

Data Inputs

- `IN` (BYTE): The input byte to be split into four quarter bytes.

Data Outputs

- `QUARTER_BYTE_00` (BYTE): The first quarter byte (bits 0-1).
- `QUARTER_BYTE_01` (BYTE): The second quarter byte (bits 2-3).
- `QUARTER_BYTE_02` (BYTE): The third quarter byte (bits 4-5).
- `QUARTER_BYTE_03` (BYTE): The fourth quarter byte (bits 6-7).

Adapters

No adapters are used.

Functionality

The function block extracts the four quarter bytes from the input byte `IN` using bitwise operations:

1. Each quarter byte is isolated using a bitwise AND operation with a corresponding mask.
2. The result is then shifted into the correct position using a right shift (`SHR`).
3. The masks and shift values used are defined in the constant file `quarterconst`.

Technical Features

- The bitwise operations are efficient and fast because they are performed directly at the processor level.
- The use of constants (`quarterconst`) allows for flexible adjustment of masks and shift values.

State Overview

The function block has no internal state. Processing occurs exclusively when the `REQ` event occurs.

Application Scenarios

- Decoding data packed into a byte.
- Extracting individual control bits from a status byte.
- Processing data in communication protocols where information is encoded in quarter bytes.

⚖️ Comparison with Similar Function Blocks

- Compared to generic bit manipulation function blocks, `SPLIT_BYTE_INTO_QUARTERS` offers a specialized and optimized solution for splitting a byte into four parts.

Other blocks, such as `EXTRACT_BITS` or `BIT_SLICE`, are more general but require more configuration and may be less efficient.

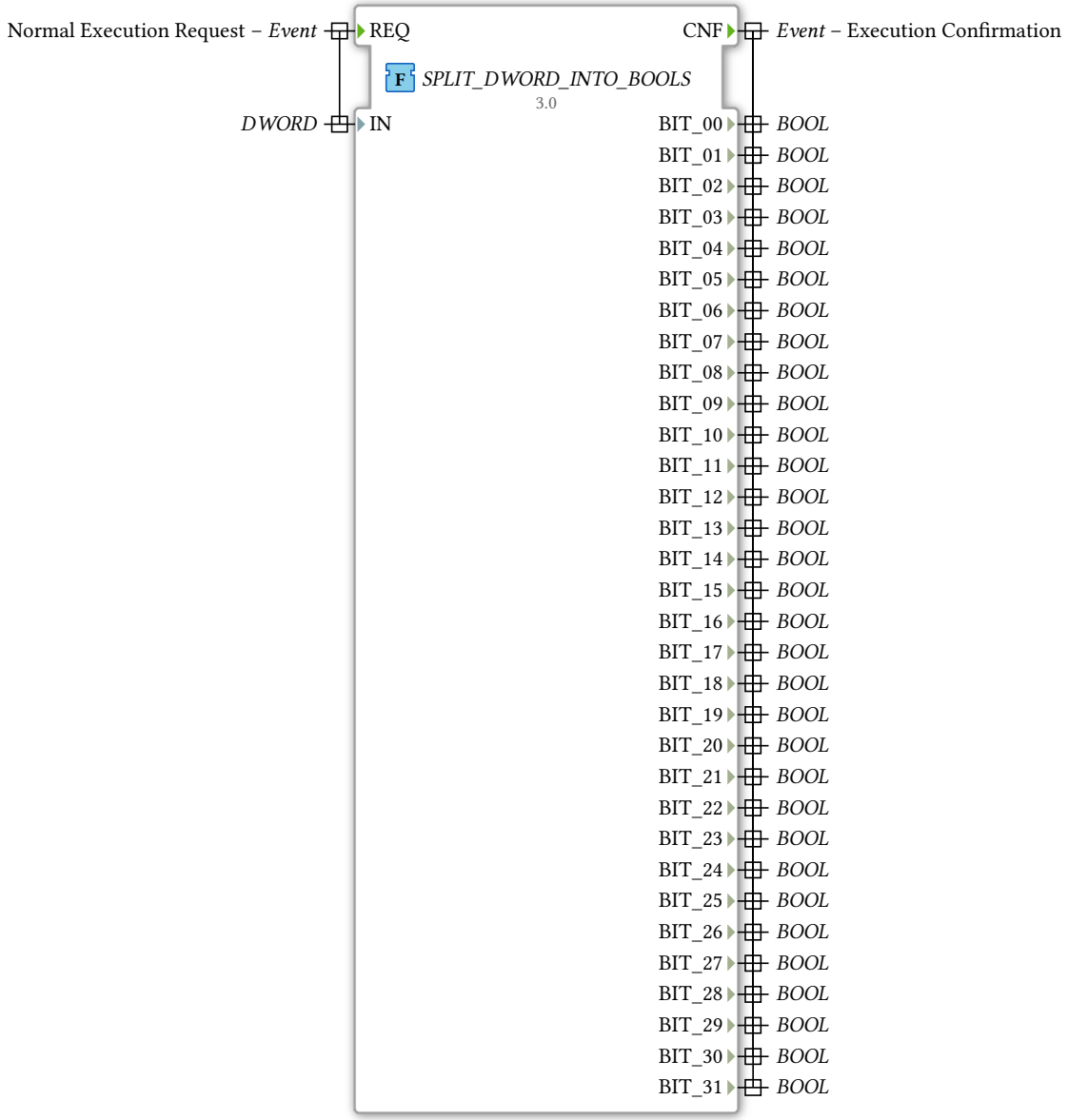
🛠️ Related Exercises

- Exercise_056

Conclusion

The `SPLIT_BYTE_INTO_QUARTERS` function block is an efficient and specialized solution for splitting a byte into four quarter bytes. Its use of bitwise operations and constants makes it fast and flexible. It is ideal for applications where data is in a compact form and needs to be processed separately.

SPLIT_DWORD_INTO_BOOLS



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_DWORD_INTO_BOOLS` is used to split a 32-bit DWORD into its individual bits and output them as separate BOOL values. This is particularly useful when compressed data needs to be processed in a controller and the individual bits need to be addressed separately.

Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. When this input is activated, the bits of the input DWORD are evaluated and made available at the outputs.

Event Outputs

- **CNF**: Confirms the successful execution of the function block. This event is triggered along with the updated BOOL outputs.

Data Inputs

- **IN**: The input DWORD to be split into 32 individual bits.

Data Outputs

- **BIT_00** to **BIT_31**: The 32 individual bits of the input DWORD, each output as a BOOL value. Each output represents one bit of the DWORD, where `BIT_00` corresponds to the least significant bit (LSB) and `BIT_31` to the most significant bit (MSB).

Adapters

This function block does not use adapters.

Operation

The function block extracts each bit of the input DWORD `IN` and assigns it to the corresponding BOOL outputs. The mapping is direct:

- `BIT_00` = Bit 0 (LSB)
- `BIT_01` = Bit 1
- ...
- `BIT_31` = Bit 31 (MSB)

When the `REQ` event is triggered, all bits are evaluated, and the `CNF` event is triggered as soon as the outputs are updated.

Technical Features

- The function block is implemented in ST (Structured Text) and uses direct bit access to the DWORD (`IN.%X0` to `IN.%X31`).
- The outputs are updated synchronously with the `CNF` event, enabling deterministic processing.

State Overview

The function block has no internal state. The outputs are recalculated with each `REQ` event.

Application Scenarios

- Processing compressed status information, e.g., from a fieldbus or interface.
- Decoding bitmasks in control applications.
- Splitting DWORD values for individual bit manipulation.

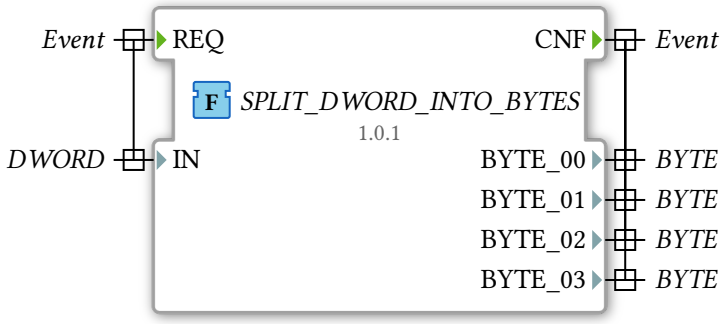
⚖ Comparison with Similar Function Blocks

- Compared to generic bit manipulation functions, this function block offers a direct and efficient method for extracting all bits of a DWORD.
- Similar function blocks might only process parts of a DWORD or contain additional logic, which is not the case here.

Conclusion

The `SPLIT_DWORD_INTO_BOOLS` function block is a specialized tool for efficiently splitting a DWORD into its individual bits. Its simple and deterministic operation makes it ideal for applications requiring bit manipulation.

SPLIT_DWORD_INTO_BYTES



• – ◦ • – ◦ • – ◦ •

Introduction

The function SPLIT_DWORD_INTO_BYTES splits a DWORD value into its four BYTE parts. It is the counterpart of ASSEMBLE_DWORD_FROM_BYTES.

Interface Structure

Event Inputs

- **REQ**: starts the split. Associated data: IN.

Event Outputs

- **CNF**: signals the completed split. Associated data: BYTE_00 to BYTE_03.

Data Inputs

- **IN** (DWORD): value to split

Data Outputs

- **BYTE_00** (BYTE): least significant byte (%B0)
- **BYTE_01** (BYTE): byte %B1
- **BYTE_02** (BYTE): byte %B2
- **BYTE_03** (BYTE): most significant byte (%B3)

Adapters

No adapters available.

Functionality

The function assigns each output the corresponding byte of the input:

- BYTE_00 := IN.%B0
- BYTE_01 := IN.%B1
- BYTE_02 := IN.%B2
- BYTE_03 := IN.%B3

Technical Details

- Implemented as an ST function (FUNCTION) with VAR_OUTPUT, package eclipse4diac::utils::splitting
- Individual bytes are accessed with the partial access syntax %B0 to %B3
- No state of its own

Application Scenarios

- Splitting 32-bit values into individual bytes, e.g. for protocols or CAN data

⚖️ Comparison with Similar Building Blocks

- SPLIT_WORD_INTO_BYTES splits a WORD into two bytes, SPLIT_DWORD_INTO_WORDS a DWORD into two words.

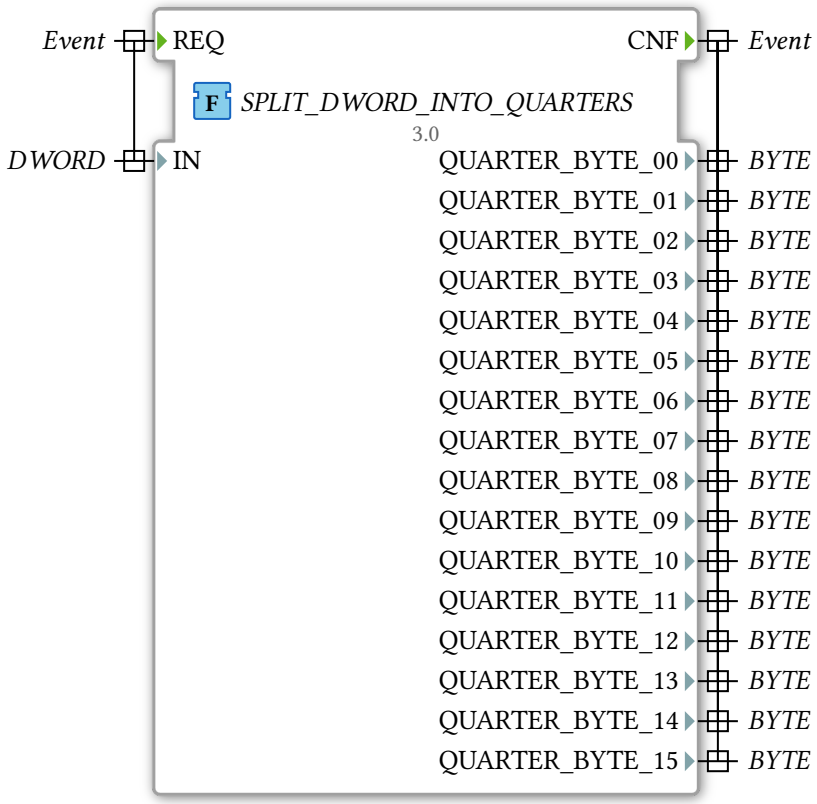
Conclusion

SPLIT_DWORD_INTO_BYTES splits a DWORD byte by byte into four BYTE outputs.

SPLIT_DWORD_INTO_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_DWORD_INTO_QUARTERS` is used to split a 32-bit word (DWORD) into its 16 individual 4-bit blocks (QUARTER_BYTE). Each of these 4-bit blocks is output as a separate BYTE variable. This block is particularly useful when individual parts of a DWORD need to be processed or analyzed separately.

Interface Structure

Event Inputs

- `REQ`: Starts processing the input DWORD.

Event Outputs

- `CNF`: Signals successful processing and provision of all 16 QUARTER_BYTE outputs.

Data Inputs

- `IN`: The input DWORD, which is split into 16 QUARTER_BYTE.

Data Outputs

- `QUARTER_BYTE_00` to `QUARTER_BYTE_15`: The 16 individual 4-bit blocks of the input DWORD, each output as a BYTE.

Adapters

No adapters are available.

Functionality

The function block extracts the 16 QUARTER_BYTE from the input DWORD using bitwise operations. Each QUARTER_BYTE is isolated using a combination of bitwise AND operation and right shift, and then converted into a BYTE value. The constants used (`DWORD_QUARTER_XX` and `SHIFT_QUARTER_XX`) are defined in a separate constant file (`quarterconst`).

Technical Features

- The block uses bitwise operations to isolate the individual 4-bit blocks.
- The output values are defined as BYTES, even though they only contain 4 bits. Therefore, the upper 4 bits of each BYTE are always 0.
- The use of predefined constants (`quarterconst`) allows for flexible adjustment of the bit positions.

State Overview

The function block has no internal state. Processing is repeated with each `REQ` event.

Application Scenarios

- Processing data packets where individual 4-bit blocks of a DWORD need to be evaluated.
- Decoding status or control information encoded in a DWORD.
- Splitting data for further processing in downstream systems.

⚖️ Comparison with Similar Blocks

- Compared to generic bit manipulation blocks, `SPLIT_DWORD_INTO_QUARTERS` offers specialized functionality for splitting a DWORD into 4-bit blocks.
- Other blocks, such as `SPLIT_DWORD_INTO_BYTES`, split a DWORD into 8-bit blocks, while this block offers finer granularity.

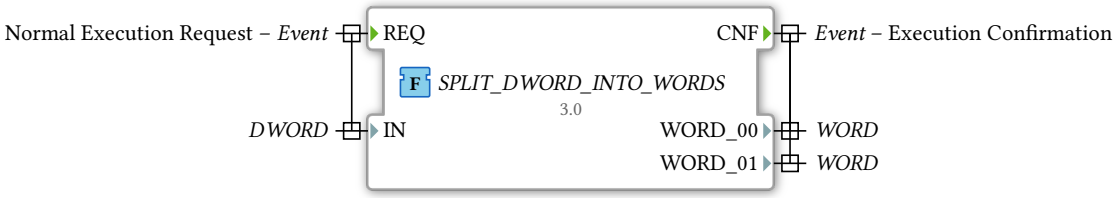
🛠️ Related Exercises

- Exercise_060

Conclusion

The `SPLIT_DWORD_INTO_QUARTERS` function block is a specialized tool for splitting a 32-bit word into 16 individual 4-bit blocks. Its simple and efficient implementation makes it ideal for applications requiring detailed analysis or processing of DWORD data.

SPLIT_DWORD_INTO_WORDS



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_DWORD_INTO_WORDS` is used to split a 32-bit DWORD into two 16-bit WORDs. This functionality is particularly useful when working with data that is stored as a DWORD but needs to be further processed in separate WORD parts.

Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. It is linked to the data input `IN`.

Event Outputs

- **CNF**: Signals the successful execution of the function block. Linked to the data outputs `WORD_00` and `WORD_01`.

Data Inputs

- **IN**: DWORD (32-bit) to be split into two WORDs.

Data Outputs

- **WORD_00**: First WORD (16-bit) of the DWORD.
- **WORD_01**: Second WORD (16-bit) of the DWORD.

Adapters

This function block does not use adapters.

Functionality

The function block takes a DWORD as input and splits it into two WORDs:

- `WORD_00` contains the lower 16 bits of the DWORD (`IN.%W0`).
- `WORD_01` contains the upper 16 bits of the DWORD (`IN.%X1`).

The split occurs with each `REQ` event, and the result is confirmed with the `CNF` event.

Technical Details

- The implementation is in Structured Text (ST).
- The function block is part of the `eclipse4diac::utils::splitting` package.

State Overview

The function block has no internal state. Processing occurs immediately with each `REQ` event.

Application Scenarios

- Processing DWORD data originating from hardware registers or communication protocols.
- Splitting 32-bit values for further processing in 16-bit systems.

⚖️ Comparison with similar blocks

- Unlike generic bit manipulation blocks, `SPLIT_DWORD_INTO_WORDS` specializes in splitting DWORDs into WORDs.
- Similar blocks might offer additional flexibility, such as selecting which bits to extract, but are often more complex to use.

Conclusion

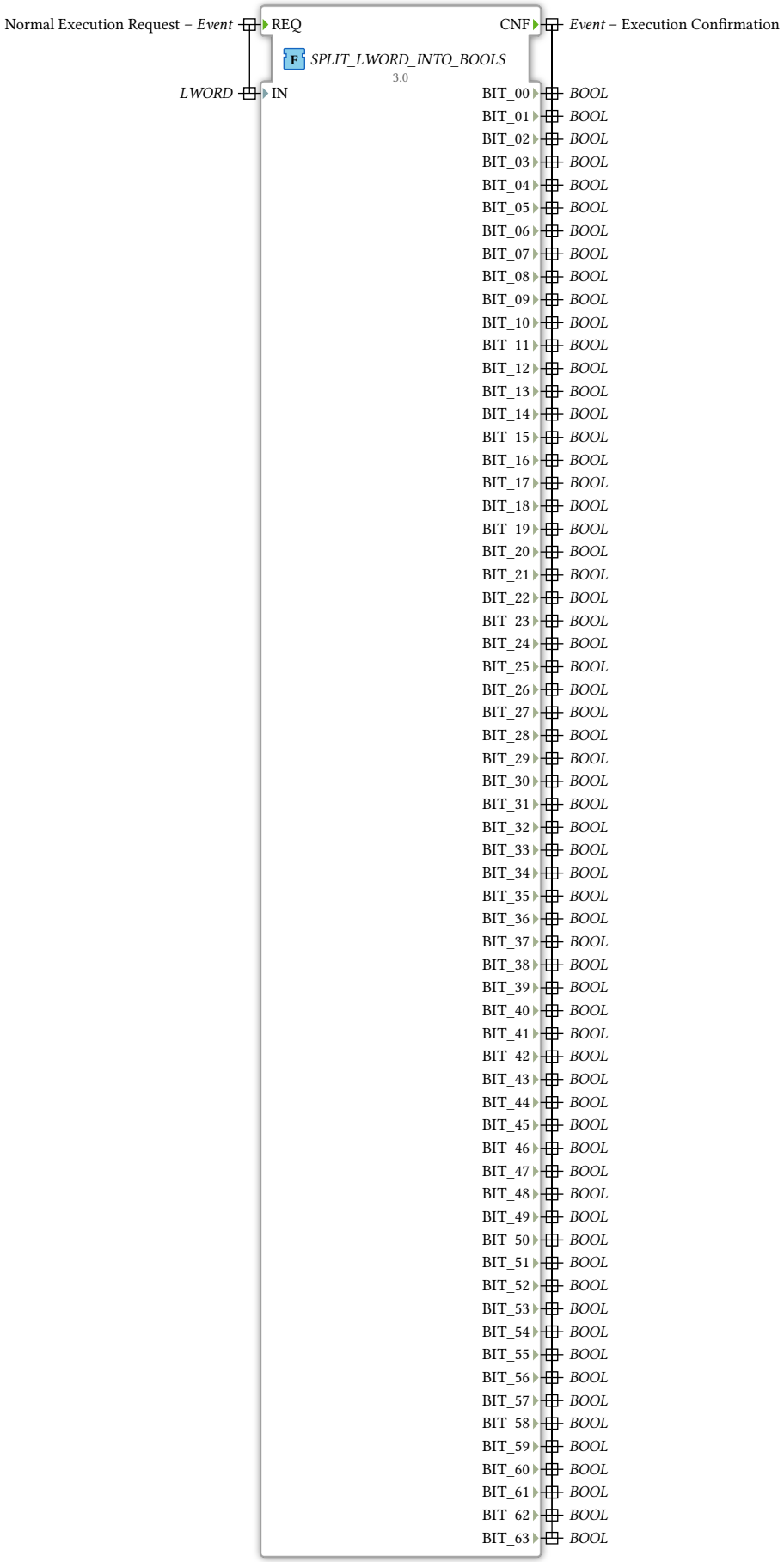
The `SPLIT_DWORD_INTO_WORDS` function block provides a simple and efficient way to split DWORD data into its WORD components. Its specialized functionality makes it particularly useful for applications that regularly perform such data conversions.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

SPLIT_LWORD_INTO_BOOLS


Normal Execution Request – Event
REQ
CNF
Event – Execution Confirmation
F
SPLIT_LWORD_INTO_BOOLS
3.0
LWORD
IN
BIT_00 BOOL
BIT_01 BOOL
BIT_02 BOOL
BIT_03 BOOL
BIT_04 BOOL
BIT_05 BOOL
BIT_06 BOOL
BIT_07 BOOL
BIT_08 BOOL
BIT_09 BOOL
BIT_10 BOOL
BIT_11 BOOL
BIT_12 BOOL
BIT_13 BOOL
BIT_14 BOOL
BIT_15 BOOL
BIT_16 BOOL
BIT_17 BOOL
BIT_18 BOOL
BIT_19 BOOL
BIT_20 BOOL
BIT_21 BOOL
BIT_22 BOOL
BIT_23 BOOL
BIT_24 BOOL
BIT_25 BOOL
BIT_26 BOOL
BIT_27 BOOL
BIT_28 BOOL
BIT_29 BOOL
BIT_30 BOOL
BIT_31 BOOL
BIT_32 BOOL
BIT_33 BOOL
BIT_34 BOOL
BIT_35 BOOL
BIT_36 BOOL
BIT_37 BOOL
BIT_38 BOOL
BIT_39 BOOL
BIT_40 BOOL
BIT_41 BOOL
BIT_42 BOOL
BIT_43 BOOL
BIT_44 BOOL
BIT_45 BOOL
BIT_46 BOOL
BIT_47 BOOL
BIT_48 BOOL
BIT_49 BOOL
BIT_50 BOOL
BIT_51 BOOL
BIT_52 BOOL
BIT_53 BOOL
BIT_54 BOOL
BIT_55 BOOL
BIT_56 BOOL
BIT_57 BOOL
BIT_58 BOOL
BIT_59 BOOL
BIT_60 BOOL
BIT_61 BOOL
BIT_62 BOOL
BIT_63 BOOL

• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_LWORD_INTO_BOOLS` is used to decompose a 64-bit LWORD data type into its individual bits. Each bit is provided as a separate BOOL output. This block is particularly useful when individual bits of a long word need to be evaluated or processed.

Interface Structure

Event Inputs

- `REQ` (Event): Starts the decomposition of the LWORD into individual bits. It is linked to the input `IN`.

Event Outputs

- `CNF` (Event): Confirms the successful decomposition of the LWORD. Linked to all 64 BOOL outputs.

Data Inputs

- `IN` (LWORD): The 64-bit word to be split into individual bits.

Data Outputs

- `BIT_00` to `BIT_63` (BOOL): The individual bits of the input LWORD, where `BIT_00` is the least significant bit (LSB) and `BIT_63` is the most significant bit (MSB).

Adapters

No adapters are included.

Functionality

The function block extracts each bit of the input LWORD (`IN`) and assigns it to the corresponding BOOL outputs. Extraction is performed via direct bit accesses (`IN.%X0` to `IN.%X63`). Upon triggering the `REQ` event, all bits are updated, and the `CNF` event confirms successful processing.

Technical Features

- Processes 64-bit data (LWORD).
- Each bit is output individually as a BOOL.
- No additional logic or filtering – direct bit mapping.

State Overview

The block has no internal state. The decomposition is performed anew with each `REQ` event.

Application Scenarios

- Evaluation of bitmasks or status registers.
- Processing of digital signals encoded as LWORDs.
- Debugging or monitoring individual bits in long data words.

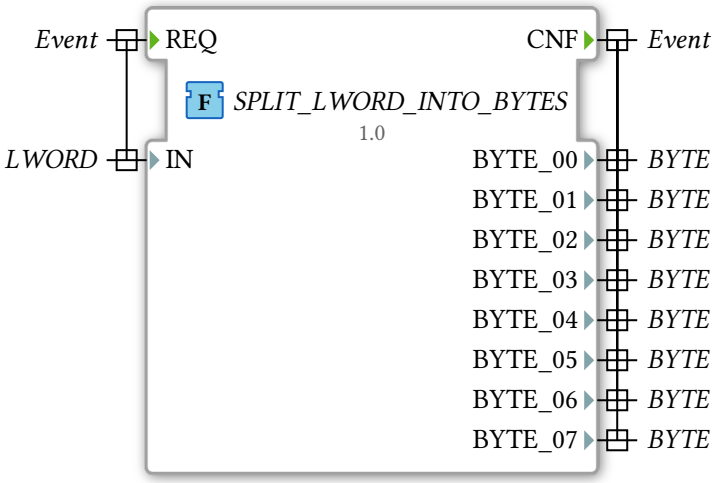
⚖️ Comparison with similar blocks

- Similar blocks like `SPLIT_WORD_INTO_BOOLS` or `SPLIT_DWORD_INTO_BOOLS` work with shorter data types (WORD and DWORD, respectively).
- This block is specifically optimized for 64-bit data.

Conclusion

The `SPLIT_LWORD_INTO_BOOLS` block is an efficient tool for decomposing 64-bit data into individual bits. Its simple and direct operation makes it ideal for applications requiring detailed bit analysis.

SPLIT_LWORD_INTO_BYTES



• – ◦ • – ◦ • – ◦ •

Introduction

The function `SPLIT_LWORD_INTO_BYTES` splits a 64-bit long word (`LWORD`) into its eight bytes (`BYTE`).

Interface Structure

Event Inputs

- `REQ`: Starts the processing. Associated data: `IN`

Event Outputs

- `CNF`: Signals the processing. Associated data: `BYTE_00` to `BYTE_07`

Data Inputs

- `IN`: The long word to be split (`LWORD`)

Data Outputs

- `BYTE_00`: Byte 0 (`BYTE`), least significant byte (bits 0 to 7)
- `BYTE_01`: Byte 1 (`BYTE`), bits 8 to 15
- `BYTE_02`: Byte 2 (`BYTE`), bits 16 to 23
- `BYTE_03`: Byte 3 (`BYTE`), bits 24 to 31
- `BYTE_04`: Byte 4 (`BYTE`), bits 32 to 39
- `BYTE_05`: Byte 5 (`BYTE`), bits 40 to 47
- `BYTE_06`: Byte 6 (`BYTE`), bits 48 to 55
- `BYTE_07`: Byte 7 (`BYTE`), most significant byte (bits 56 to 63)

Adapters

This function does not use any adapters.

How it works

The function reads the eight bytes from the input `IN` via the partial access `.%B0` to `.%B7`:

```
BYTE_00 := IN.%B0;
BYTE_01 := IN.%B1;
BYTE_02 := IN.%B2;
BYTE_03 := IN.%B3;
BYTE_04 := IN.%B4;
BYTE_05 := IN.%B5;
BYTE_06 := IN.%B6;
BYTE_07 := IN.%B7;
```

`BYTE_00` is thus the least significant and `BYTE_07` the most significant byte of the `LWORD`.

Technical Details

- Implemented in Structured Text (ST) as a `FUNCTION` without internal state
- Part of the package `eclipse4diac::utils::splitting`
- Version 1.0 (2026-05-16)

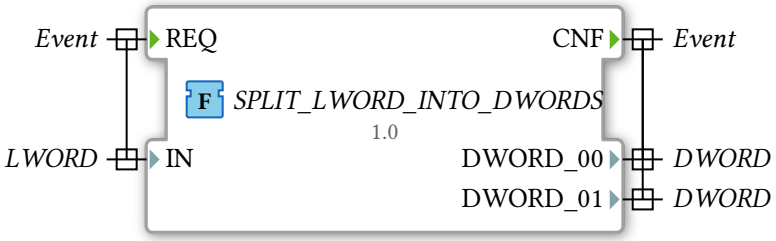
Application Scenarios

- Splitting a 64-bit value into individual bytes, e.g. for telegrams or bus messages (CAN: 8 data bytes)
- Byte-wise analysis of 64-bit bit patterns

Summary

`SPLIT_LWORD_INTO_BYTES` extracts the eight bytes of an `LWORD`, starting with the least significant one.

SPLIT_LWORD_INTO_DWORDS



• – ◦ • – ◦ • – ◦ •

Introduction

The function `SPLIT_LWORD_INTO_DWORDS` extracts the two 32-bit double words (`DWORD`) from a 64-bit long word (`LWORD`).

Interface Structure

Event Inputs

- `REQ`: Starts the processing. Associated data: `IN`

Event Outputs

- `CNF`: Signals the processing. Associated data: `DWORD_00`, `DWORD_01`

Data Inputs

- `IN` (`LWORD`): Long word to be split

Data Outputs

- `DWORD_00` (`DWORD`): Least significant double word (bits 0 to 31)
- `DWORD_01` (`DWORD`): Most significant double word (bits 32 to 63)

Adapters

This function uses no adapters.

How it works

The function assigns the parts via partial access:

```
DWORD_00 := IN.%D0;
DWORD_01 := IN.%D1;
```

Technical Details

- Implemented in Structured Text (ST) as a `FUNCTION` without internal state
- Part of the package `eclipse4diac::utils::splitting`
- The inverse is formed by the `ASSEMBLE_*` functions in the package `assembling` (e.g. `ASSEMBLE_LWORD_FROM_WORDS` for `WORD` parts).

Application Scenarios

- Splitting a 64-bit value into two 32-bit values, e.g. for fieldbus or register access

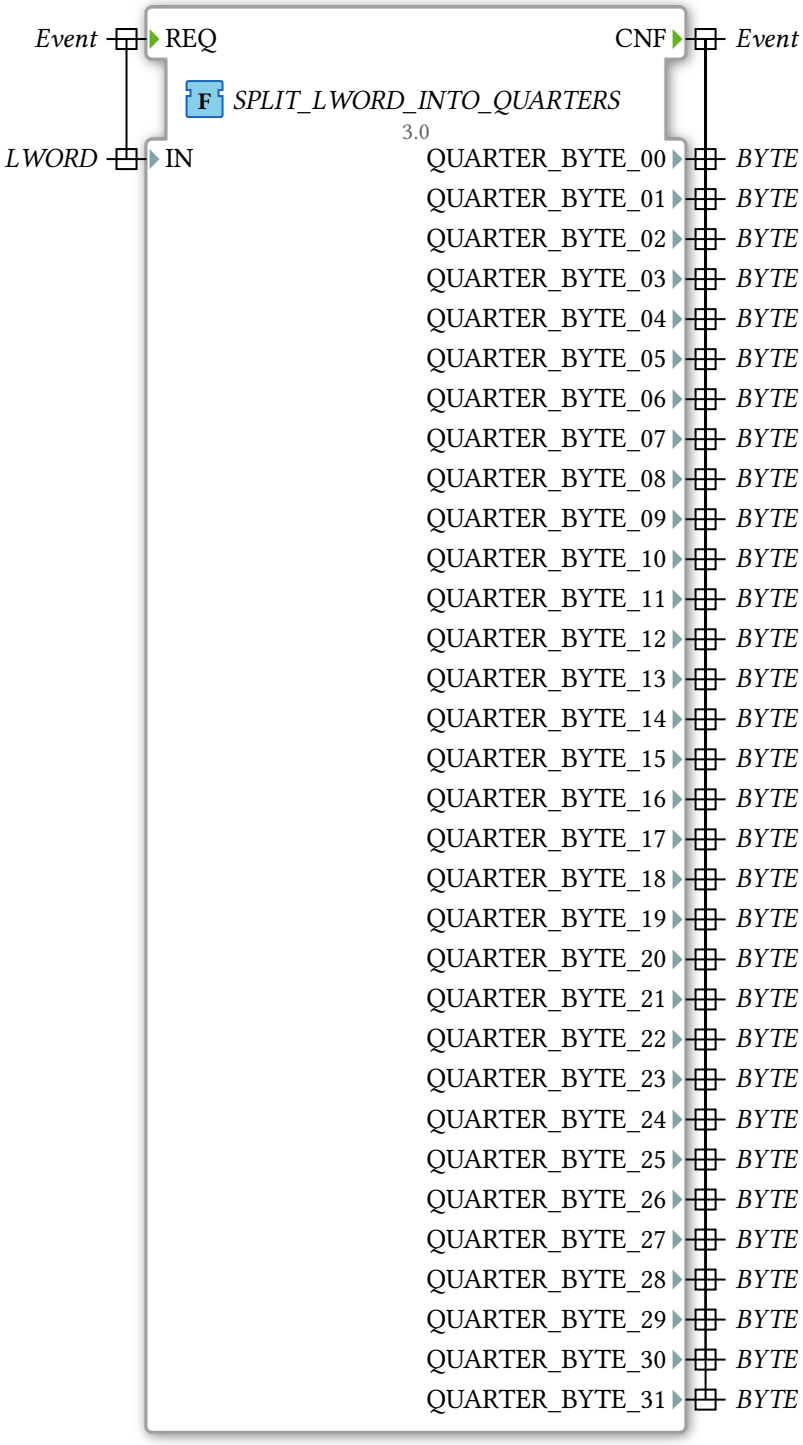
Summary

`SPLIT_LWORD_INTO_DWORDS` splits an `LWORD` into its least significant and its most significant `DWORD`.

SPLIT_LWORD_INTO_QUARTERS

Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_LWORD_INTO_QUARTERS` extracts the 32 quarter bytes (8 bits each) from a 64-bit LWORD. This allows for the targeted processing of individual byte segments within a long word.

Interface Structure

Event Inputs

- `REQ`: Starts processing the input value `IN`.

Event Outputs

- `CNF`: Confirms successful processing and returns all 32 quarter bytes.

Data Inputs

- `IN` (LWORD): The 64-bit input word to be split into quarter bytes.

Data Outputs

32 outputs of type BYTE:

- `QUARTER_BYTE_00` to `QUARTER_BYTE_31`: The extracted 8-bit segments of the input word.

Adapters

No adapters available.

Adapters ## Functionality

The function block uses bitwise operations (AND and right shift) in combination with constants (`quarterconst::LWORD_QUARTER_xx` and `quarterconst::SHIFT_QUARTER_xx`) to isolate each of the 32 quarter bytes from the input LWORD. Each quarter byte is then written to a separate BYTE output.

Technical Features

- Uses special bitmasks and shift constants for precise extraction
- Works with the largest IEC data type, LWORD (64-bit)
- Delivers all 32 quarter bytes in parallel in a single pass

State Overview

1. **Idle**: Waits for REQ event
2. **Processing**: Extracts all quarter bytes
3. **Output**: Triggers CNF event with all output values

Application Scenarios

- Processing long data words in communication protocols
- Extracting individual parameters from packed data structures
- Bitwise analysis of sensor or control data

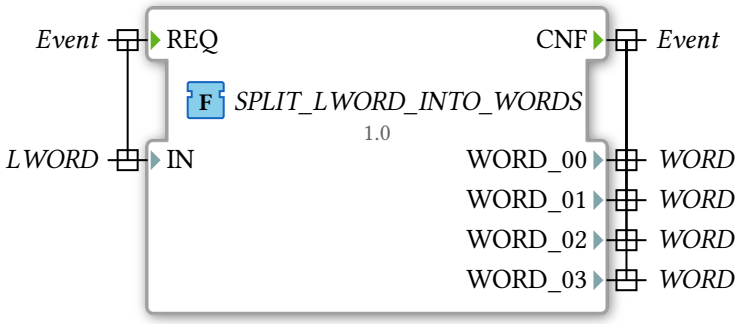
⚖️ Comparison with Similar Function Blocks

- Unlike simple BYTE extraction blocks, this function block can process all 32 segments simultaneously
- More efficient than multiple cascaded extraction blocks
- Specialized in processing LWORD data types

Conclusion

The The `SPLIT_LWORD_INTO_QUARTERS` function block offers a highly optimized solution for splitting 64-bit words into their individual byte components. Its parallel processing of all segments makes it particularly suitable for applications requiring efficient processing of large data words.

SPLIT_LWORD_INTO_WORDS



• – ◦ • – ◦ • – ◦ •

Introduction

The function `SPLIT_LWORD_INTO_WORDS` extracts the four `WORD` values from a `LWORD`. It is useful when a 64-bit value (e.g. from a communication protocol) has to be split into 16-bit parts.

Interface Structure

Event Inputs

- **REQ**: starts the processing (associated with `IN`)

Event Outputs

- **CNF**: signals completion (associated with `WORD_00`..`WORD_03`)

Data Inputs

- **IN** (LWORD): 64-bit value to be split

Data Outputs

- **WORD_00** (WORD): first word, bits 0-15 (`IN.%W0`)
- **WORD_01** (WORD): second word, bits 16-31 (`IN.%W1`)
- **WORD_02** (WORD): third word, bits 32-47 (`IN.%W2`)
- **WORD_03** (WORD): fourth word, bits 48-63 (`IN.%W3`)

Adapters

No adapters are used.

Functionality

The function assigns the four words by partial access: `WORD_00 := IN.%W0;`, `WORD_01 := IN.%W1;`, `WORD_02 := IN.%W2;`, `WORD_03 := IN.%W3;`.

Technical Details

- Pure bit accesses without calculation effort; no states
- Package `eclipse4diac::utils::splitting`, version 1.0 (2026-05-16)

Application Scenarios

- Splitting 64-bit values (e.g. from OPC-UA or CAN data) into 16-bit registers

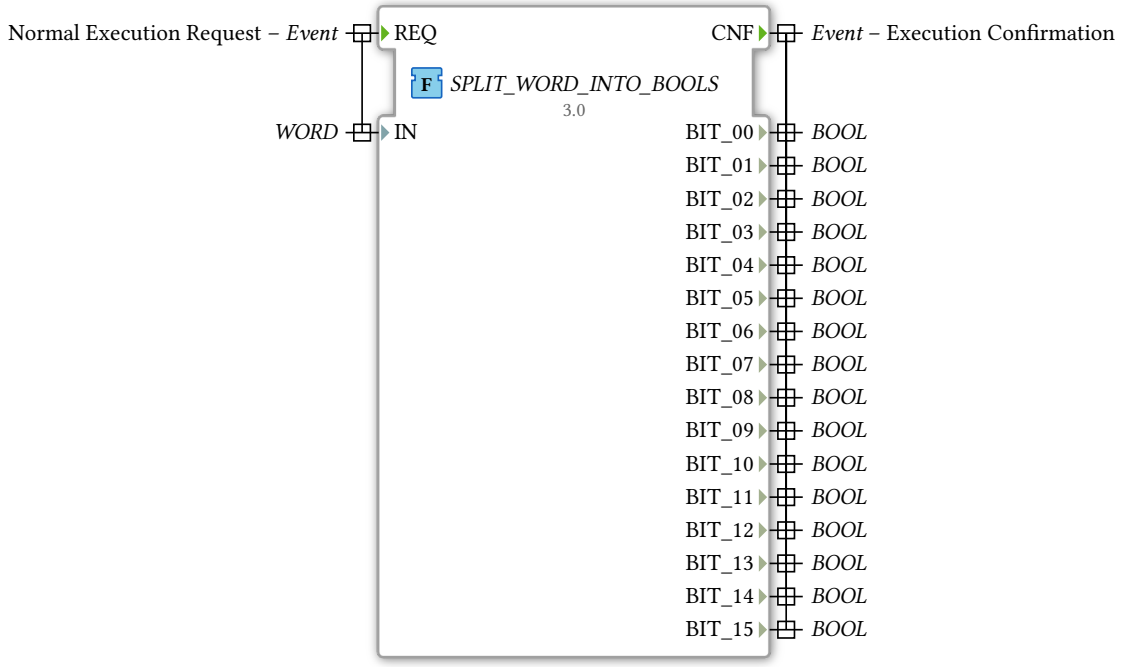
⚖️ Comparison with Similar Blocks

Analogous blocks split smaller values, such as `SPLIT_BYTE_INTO_QUARTERS` (byte into four quarter bytes) or `SPLIT_DWORD_INTO_BOOLS`.

Conclusion

`SPLIT_LWORD_INTO_WORDS` splits a `LWORD` into four `WORD` values and delivers them with a single `CNF` event.

SPLIT_WORD_INTO_BOOLS



• – ◦ • – ◦ • – ◦ •

The function block `SPLIT_WORD_INTO_BOOLS` is used to decompose a 16-bit word into its individual bits. Each bit of the input word is mapped to a separate BOOL output. This is particularly useful when individual bits of a word need to be further processed or monitored.

- `REQ` (Normal Execution Request): Starts the decomposition of the input word. It is linked to the data input `IN`.
- `CNF` (Execution Confirmation): Confirms successful word decomposition. Linked to all 16 BOOL outputs.
- `IN` (WORD): The 16-bit word to be decomposed into individual bits.
- `BIT_00` to `BIT_15` (BOOL): The individual bits of the input word, where `BIT_00` represents the least significant bit (LSB) and `BIT_15` represents the most significant bit (MSB).

Data Outputs

Data Inputs

Event Outputs

Event Inputs

Interface Structure

Introduction

Adapters

- No adapters available.

The function block parses the input word `IN` into its 16 individual bits and assigns them to the corresponding BOOL outputs. The assignment is done directly via the bit addressing of the WORD data type (e.g., `IN.%X0` for the first bit). When the `REQ` event is triggered, all bits are extracted, and the `CNF` event confirms successful execution.

- Bit extraction is performed directly via low-level hardware bit addressing, enabling efficient processing.
- The function block is deterministic and has no internal states.
- **Idle**: Waiting for the `REQ` event.
- **Processing**: Parses the word and updates the outputs.
- **Complete: `Löst das` CNF` event and returns to the idle state.

- Processing of digital input signals transmitted as WORDs.
- Decoding of status or control words in control systems.
- Interface between word-based and bit-based system components.
- Compared to generic bit manipulation functions, this block offers a specialized and simple interface for decomposing words.
- Similar blocks might support a flexible number of bits, which is not the case here.

The `SPLIT_WORD_INTO_BOOLS` function block is a simple and efficient block for decomposing 16-bit words into individual bits. It is particularly suitable for applications where clear and direct bit extraction is required.

Functionality

Technical Features

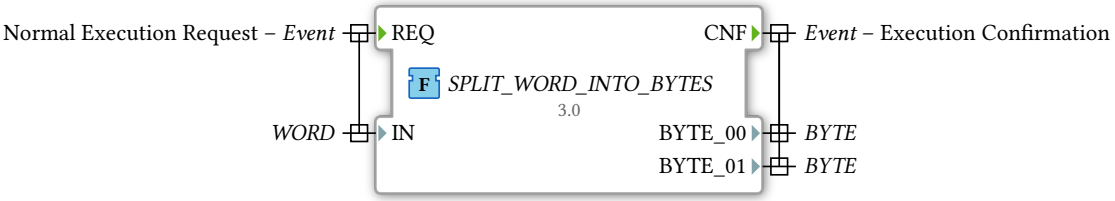
State Overview

Application Scenarios

⚖ Vergleich mit ähnlichen Bausteinen

Conclusion

SPLIT_WORD_INTO_BYTES



• – ◦ • – ◦ • – ◦ •

Introduction

The function block `SPLIT_WORD_INTO_BYTES` is used to split a 16-bit word (WORD) into its two 8-bit bytes. This is particularly useful when working with data that is transmitted as a WORD but needs to be processed byte by byte.

Interface Structure

Event Inputs

- **REQ**: Starts the execution of the function block. When this event input is triggered, processing of the input value `IN` begins.

Event Outputs

- **CNF**: Signals the successful completion of processing. This event is triggered along with the two output bytes `BYTE_00` and `BYTE_01`.

Data Inputs

- **IN**: The 16-bit word (WORD) to be split into two bytes.

Data Outputs

- **BYTE_00**: The least significant byte (bits 0 to 7) of the input word `IN`.
- **BYTE_01**: The most significant byte (bits 8 to 15) of the input word `IN`.

Adapters

No adapters available.

Adapters ## Functionality

The function block extracts the two bytes from the input word `IN` and outputs them via the outputs `BYTE_00` (lower-order byte) and `BYTE_01` (higher-order byte). Extraction is performed by direct access to the bytes of the WORD data type (`IN.%B0` for the lower-order byte and `IN.%B1` for the higher-order byte).

Technical Features

- The function block is implemented in the ST (Structured Text) programming language.
- Processing occurs synchronously with the triggering of the `REQ` event.
- The bytes are output along with the `CNF` event.

State Overview

1. **Idle State**: Waiting for the `REQ` event.
2. **Processing State**: When `REQ` is triggered, the bytes are extracted.
3. **Output State**: The extracted bytes are output with the `CNF` event, and the block returns to the idle state.

Application Scenarios

- **Data Communication**: When processing network data transmitted as WORDs but requiring byte-by-byte interpretation.
- **Low-Hardware Programming**: When working with register contents read as WORDs but containing individual bytes.
- **Data Conversion**: When converting WORD data into two separate BYTE data for further processing.

⚖️ Comparison with Similar Building Blocks

- Compared to generic extraction functions, `SPLIT_WORD_INTO_BYTES` offers a specialized and optimized solution for splitting WORD data into bytes.
- Other building blocks might offer additional features such as sign handling or bit extraction, but are less efficient for this specific task.

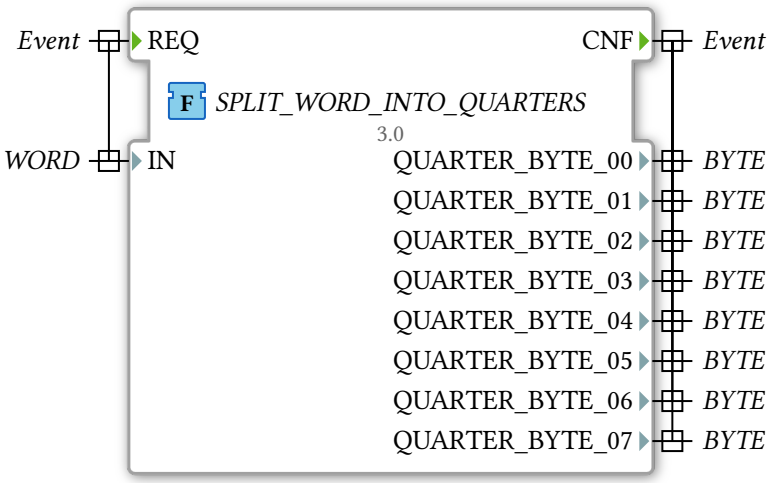
Conclusion

The `SPLIT_WORD_INTO_BYTES` function block is a simple yet effective building block for splitting 16-bit words into their individual bytes. Its clear interface and efficient implementation make it ideal for applications requiring this functionality.

SPLIT_WORD_INTO_QUARTERS

🎧 Podcast

- QUARTER



• – ◦ • – ◦ • – ◦ •

Introduction

The function block SPLIT_WORD_INTO_QUARTERS is used to split a 16-bit word into eight 2-bit quarter bytes. This is particularly useful when processing compressed data or extracting specific bit patterns.

Interface Structure

Event Inputs

- REQ: Starts processing the input word. Linked to the data input IN.

Event Outputs

- CNF: Signals the completion of processing. It is linked to all eight quarter-byte outputs.

Data Inputs

- IN (WORD): The 16-bit input word, which is split into quarter bytes.

Data Outputs

- QUARTER_BYTE_00 to QUARTER_BYTE_07 (BYTE): The eight extracted 2-bit quarter bytes. Each output contains one of the quarter bytes of the input word.

Adapters

No adapters are available.

Operation

The function block uses bitmasks and shift operations to extract the individual 2-bit segments from the input word. Each quarter byte is isolated using a bitwise AND operation with a specific mask and then shifted to the corresponding position to be output as a byte.

Technical Features

- Uses constants (quarterconst::WORD_QUARTER_XX and quarterconst::SHIFT_QUARTER_XX) for the masks and shift positions.
- Extraction is performed by combining AND operations and right shifts.

State Overview

1. **Idle**: Waits for the REQ event.
2. **Processing**: Extracts the quarter bytes from the input word.
3. **Completion**: Sends the CNF event and outputs the quarter bytes.

Application Scenarios

- Decoding compressed data.
- Extraction of control bits from a status word.
- Processing of coded sensor data.

⚖️ Comparison with similar function blocks

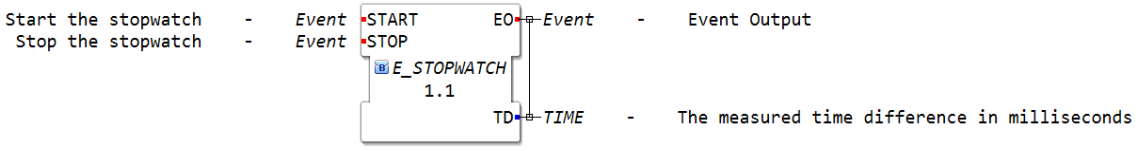
- **SPLIT_WORD_INTO_BYTES**: Splits a word into two bytes, not eight quarter-bytes.
- **BIT_EXTRACT**: Extracts individual bits or groups of bits, but is less specialized for quarter-bytes.

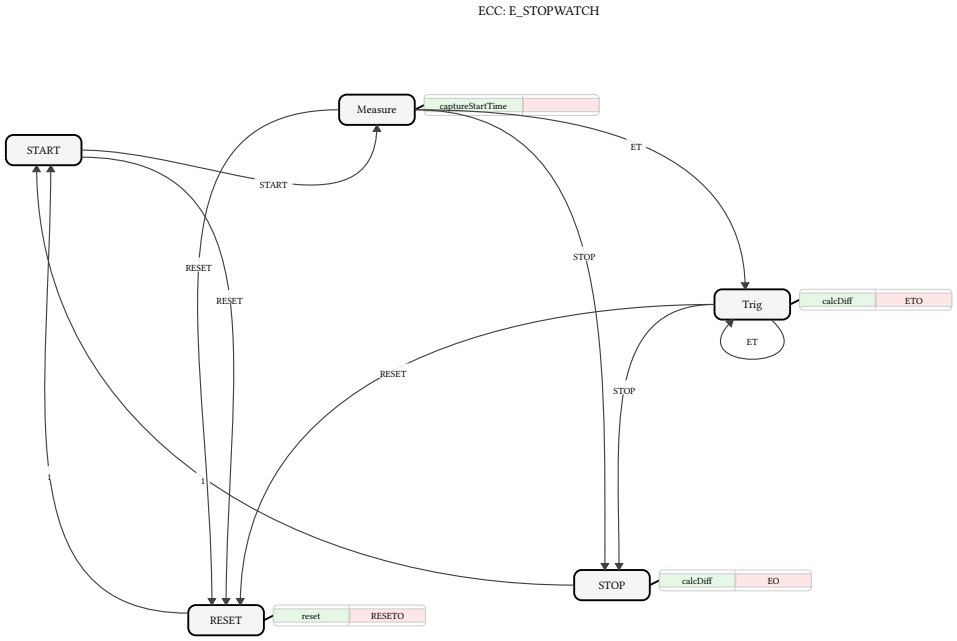
Conclusion

The SPLIT_WORD_INTO_QUARTERS function block is a specialized tool for efficiently extracting 2-bit data from a 16-bit word. Its clear interface and efficient implementation make it ideal for applications working with compressed or bit-coded data.

2.11.12.23 timing

E_STOPWATCH

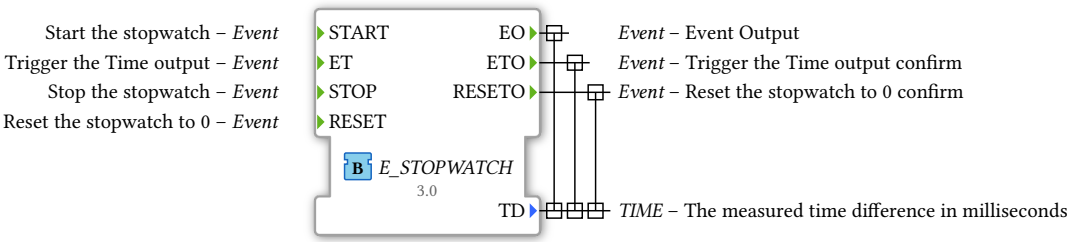




• – ○ • – ○ • – ○ •

Introduction

The **E_STOPWATCH** function block is a high-precision timing module compliant with the IEC 61499-2 standard, developed under the EPL 2.0 license. The module enables millisecond-accurate timing measurements between events in real-time systems.



Interface Structure

Event Inputs

- `START`: Starts timing
- `ET`: Triggers intermediate output of the measured time
- `STOP`: Ends timing and outputs the result
- `RESET`: Resets the stopwatch

Event Outputs

- `EO`: Outputs time difference after STOP
- `ETO`: Outputs time difference after ET trigger
- `RESETO`: Confirms the reset operation

Data Outputs

Parameter	Type	Description	Accuracy
`TD`	TIME	Measured time difference	1 ms

Functionality

1. **Start Phase**:

- `START` initializes the time measurement
- Monotone system time is recorded (`NOW_MONOTONIC`)

1. **Measurement Phase**:

- `ET` delivers intermediate results without stopping the measurement
- Multiple triggers are possible

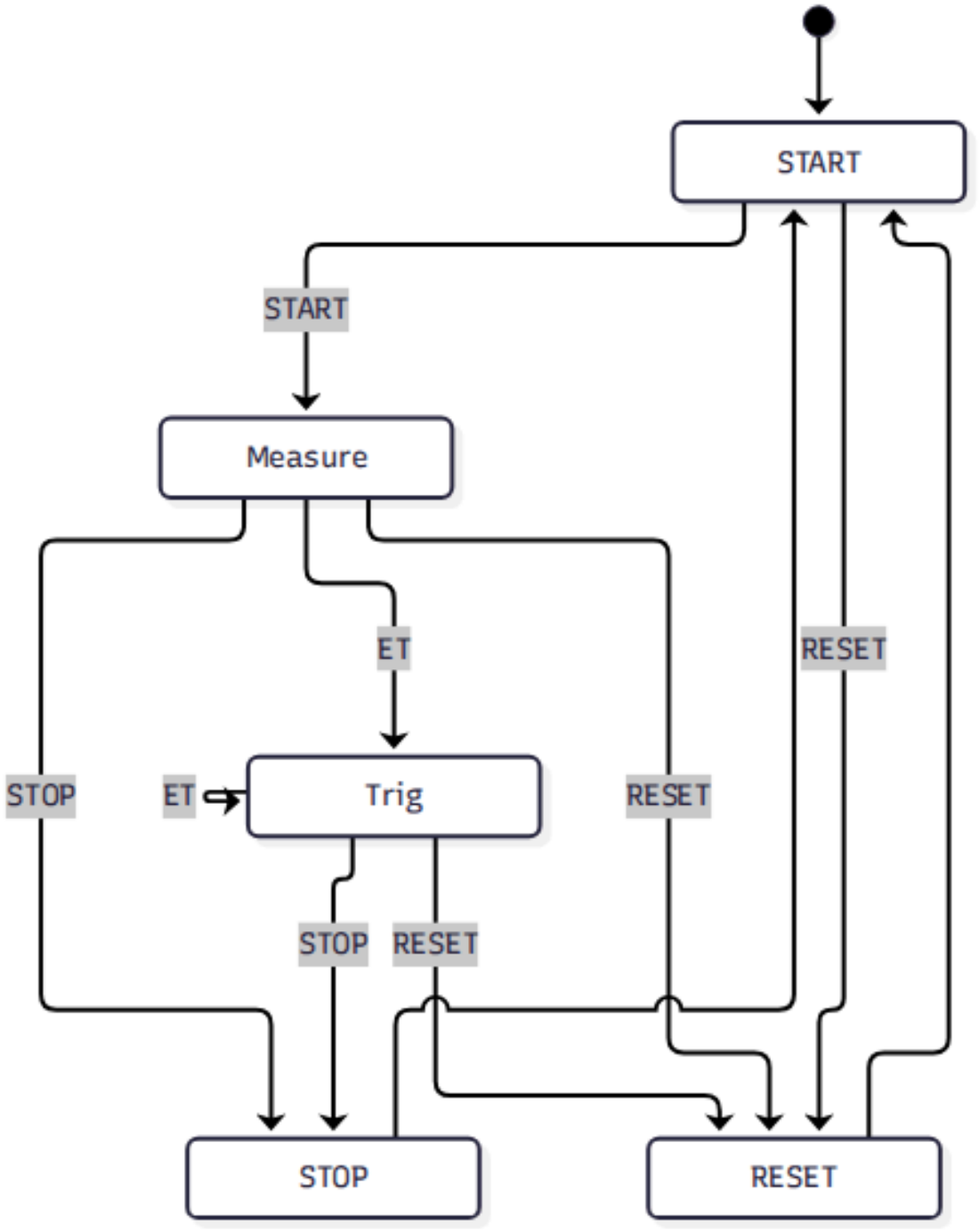
1. **Closing Phase**:

- `STOP` ends the measurement and outputs the final result
- `RESET` resets all values

Technical Features

✓ IEC 61499-2 compliant ✓ Monotone time measurement (no system time adjustments) **✓ Millisecond accuracy ✓ Multiple triggers possible** during the measurement **✓ Eclipse 4diac integration** (Package: eclipse4diac::utils::timing)

State Machine (ECC)



Application Scenarios

- **Performance Measurement**: Algorithm Runtimes
- **Process Control**: Time-Critical Processes
- **System Diagnostics**: Response Times
- **Production**: Cycle Time Measurement

Example Values

Operation	Result (TD)
START -> STOP after 1.5s	T#1s500ms

START -> ET after 750ms -> STOP after 1.5s	T#750ms (ETO), T#1s500ms (EO)
RESET during measurement	T#0s

⚖️ Comparison with Similar Components

Feature	E_STOPWATCH	Standard Timer	High-Res Timer
Accuracy	1 ms	10 ms	1 µs
Trigger during run	Yes	No	Yes
Monotonous timer	Yes	No	Yes
Reset function	Yes	Partially	Yes

🛠️ Related Exercises

- Exercise_020i
- Exercise_020i_AX

Conclusion

The E_STOPWATCH module is the precise instrument for time measurements:

- **Reliable**: Monotonous time base prevents measurement errors
- **Flexible**: Intermediate outputs without interrupting the measurement
- **Simple**: Intuitive operation with Start/Stop/Reset

Ideal for:

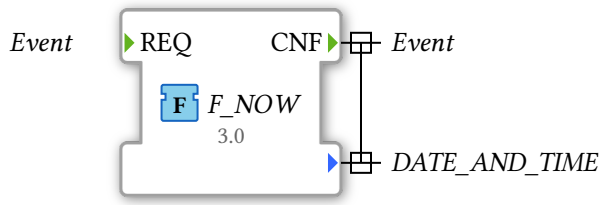
- Performance optimizations
- Real-time systems
- Process analyses
- Benchmark tests

The precise stopwatch for industrial control systems

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

F_NOW



• – ◦ • – ◦ • – ◦ •

Introduction

The **F_NOW** function is an IEC 61499-1 compliant helper function for querying the current local date and time, developed under the EPL-2.0 license (version 3.0). The function serves as a wrapper for the ST function NOW().

Interface Structure

Event Inputs

- `REQ`: Triggers the time query

Event Outputs

- `CNF`: Acknowledges the time query

Data Outputs

Parameter	Type	Description	Format
(implicit)	DATE_AND_TIME	Current timestamp	DT#2025-04-14-15:30:45

Functionality

1. **Query**:

- `REQ` triggers the time query
- System time is captured in the local time zone

1. **Output**:

- `CNF` returns the result in DATE_AND_TIME format
- Contains date and time with second accuracy

Technical Features

✓ **IEC 61499-1 compliant** ✓ **Local time zone support** (including daylight saving time) ✓ **Easy integration** as a NOW() wrapper ✓ **Eclipse 4diac integration**

Application Scenarios

- **Logging**: Timestamps for events
- **Reports**: Generation with creation date
- **Time-controlled processes**: Current time check
- **Diagnostics**: Time-based error analysis

Example Values

Call time	Return value
14.04.2025 15:30:45	DT#2025-04-14-15:30:45
31.12.2024 23:59:59	DT#2024-12-31-23:59:59

⚖️ **Comparison with similar functions**

Feature	F_NOW	NOW	TIME()
Return type	DATE_AND_TIME	DATE_AND_TIME	TIME
Time zone	Local	Local	System
Accuracy	Seconds	Seconds	Milliseconds
Daylight Saving Time	Yes	Yes	No

Limitations

⚠️ **Time jumps possible** with:

- Daylight Saving Time
- Manual time adjustments
- NTP synchronization

Conclusion

The F_NOW function offers easy access to the system time:

- **Practical**: Quick integration of existing NOW functions
- **Localized**: Automatic time zone adjustment
- **Standards compliant**: IEC-compatible implementation

Ideal for:

- Event logging
- Time-controlled processes
- Diagnostic functions
- Report generation

The simple solution for time-critical applications in 4diac

Related topic subpages on ms-muc-docs.de

- Eclipse 4diac IDE & color reference on ms-muc-docs.de

F_NOW_MONOTONIC



• – ◦ • – ◦ • – ◦ •

Introduction

F_NOW_MONOTONIC is an IEC 61499-compliant function that returns a monotonic timestamp. Unlike the system time (NOW`), this time is independent of external changes (e.g., time zones or NTP updates) and is therefore ideally suited for time-critical control applications or performance measurements.

The function serves as a wrapper for the ST function `NOW_MONOTONIC()`.

Interface Structure

Event Inputs

- **REQ** (Trigger):

Starts the execution of the function. Upon receipt of this event, the current monotonic timestamp is calculated and output via `CNF`.

Event Outputs

- **CNF** (Acknowledgement):

Signals the completion of the operation. The event is output along with the calculated time value (`TIME`).

Data Inputs

- *None* – The function requires no additional input parameters.

Data Outputs

- **TIME** (Output Variable):

Return value of type `TIME`, representing the current value of the monotonic clock.

Example: `T#123456ms` (depending on the hardware/OS reference point).

Operation

1. **Initialization**:

The function is activated by the event `REQ`. Resources are not persistently stored, as this is a stateless computation.

1. **Macro Execution**:

- Calls the system-level function `NOW_MONOTONIC()`.
- Assigns the return value to the output variable `TIME`.

- Triggers the event `CNF`.

1. **Error Handling**:

- No specific error handling, as the function has no external dependencies.
- In case of hardware/OS errors, the return value is undefined.

–

Technical Features

- **Monotonic Time**:

Guaranteed continuously increasing values, even with system time changes.
Note: The reference point is hardware-dependent (e.g., system startup).

- **License**: Eclipse Public License 2.0 (EPL-2.0).

Return Codes

- **Success**: Valid `TIME` value (e.g., `T#42s`).
- **Error**: No explicit error codes, but undefined values in case of system errors.

–

Application Scenarios

1. **Real-time Measurements**: Runtime analysis of control cycles unaffected by system time changes.
2. **Synchronization**: Timestamps for logging in distributed systems.
3. **Timeout Control**:

Reliable timers for critical processes.
–

⚖️ Comparison with Similar Function Blocks

Function	`F_NOW_MONOTONIC`	`NOW()`	`NOW_SYSTEM()`
Time Source	Monotonous (Hardware)	System Time	System Time (UTC)
Influenceable	No	Yes (NTP, Time Zone)	Yes (NTP)
Use Case	Real-time Measurements	General Timestamps	Global Timestamps

---

Conclusion

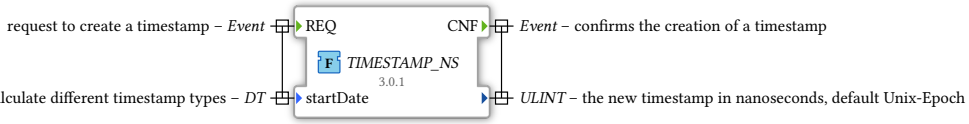
F_NOW_MONOTONIC` is a lean yet essential component for applications requiring a reliable and monotonous time source.
Its close connection to the hardware and its independence from system time changes make it particularly suitable for industrial real-time applications.
Integration with Eclipse 4diac ensures compatibility with modern automation solutions.
Note: For absolute time references, `NOW_SYSTEM()` should be used.

🌐 Related topic subpages on ms-muc-docs.de

- 🌐 Eclipse 4diac IDE & Color Reference on ms-muc-docs.de

]

TIMESTAMP_NS



• – ◦ • – ◦ • – ◦ •

Introduction

The **TIMESTAMP_NS** is an IEC 61499-compliant function block for generating high-resolution timestamps in nanoseconds. By default, it generates Unix epoch timestamps (from January 1, 1970), but supports alternative time references by adjusting the start date (`startDate`).

Developed under the EPL 2.0 license.

–

Interface Structure

Event Inputs

- **REQ** (Trigger):

Starts the timestamp calculation. Must be linked to the input parameter `startDate`.
Example: `REQ(startDate := DT#2000-01-01-00:00:00)`

Event Outputs

- **CNF** (Acknowledgement):

Signals the completion of the calculation. The generated timestamp is then available as the value `ULINT`.

Data Inputs

- **startDate** (DT):

Start date for the timestamp calculation.
Default: `DT#1970-01-01-00:00:00` (Unix epoch).
Note: Other values allow user-defined references (e.g., system startup).

Data Outputs

- **ULINT** (Output Variable):

Timestamp in nanoseconds since the defined `startDate`.
Example: `1680000000000000000` (1.68e18 ns ≈ 2023-03-28).

Functionality

1. **Initialization**:

- Activated by the `REQ` event.
- Uses the current system time (`NOW()`).

1. **Calculation**:

- Subtracts `startDate` from the current system time.
- Converts the difference to nanoseconds via `TIME_IN_NS_TO_ULINT()`.
- Outputs the result as `ULINT` and triggers `CNF`.

1. **Error Handling**:

- No explicit error message, but overflow is possible in the case of:
- Extremely large time differences (>584 years).
- Invalid `startDate` (e.g., future date).

–

Technical Features

- **Nanosecond Precision**: 64-bit counter (`ULINT`) prevents overflow up to 584 years.
- **Flexible References**: Any starting date (e.g., `DT#2000-01-01` for Y2K reference).
- **License**: Eclipse Public License 2.0 (EPL-2.0).

–

Return Codes

- **Success**: Valid `ULINT` value (nanoseconds).
- **Errors**: No explicit codes, but:
- `0` for invalid calculations (e.g., `startDate > NOW()`).
- Overflow for differences >26 ns.

–

Application Scenarios

1. **High-Performance Logging**:

Synchronization of events in distributed systems.

1. **Real-Time Measurements**:

Runtime analysis with nanosecond precision.

1. **User-Defined Time Periods**:

Calculation of intervals from arbitrary reference points.

–

⚖️ Comparison with Similar Building Blocks

Function	`TIMESTAMP_NS`	`NOW()`	`F_NOW_MONOTONIC`
Output Format	Nanoseconds (ULINT)	Time Interval (TIME)	Time Interval (TIME)
Reference Point	Configurable	System Startup	Hardware Dependent
Precision	1 ns	1 ms	1 ns (typical)
Use Case	Absolute Timestamps	Relative Time Measurement	Monotonical Measurement

---

Conclusion

TIMESTAMP_NS` is an essential component for applications requiring highly precise and flexible timestamps. Its support for user-defined start dates and nanosecond precision makes it ideal for:

- **Industrial Data Recording**
- **Real-Time Controls**
- **Scientific Measurement Systems**

Note: For monotonous time measurements, `F_NOW_MONOTONIC` should be preferred.

2.12 ⚖️ Comparison with other standards

This section presents concepts and building blocks from other standards, particularly **IEC 61131-3**, to highlight differences and similarities with IEC 61499.

2.12.1 IEC 61131-3 (Old Standard)

This section describes function blocks according to the "classic" PLC standard **IEC 61131-3**. These blocks serve as a reference and basis for comparison with the IEC 61499 concepts. !!! Warning "Important Note" These function blocks comply with **IEC 61131-3**. Behaviors, particularly regarding execution and events, may differ in IEC 61499.

2.12.1.1 Timers

2.12.1.2 Bistable Elements (Flip-Flops)

2.12.1.3 RS (Bistable, Priority Reset) - IEC 61131-3

The function block **RS** is a bistable element (flip-flop) where resetting takes precedence over setting.

Interface

Type	Name	Data Type	Description
Input	S	BOOL	Set Input (Set)
Input	R1	BOOL	Reset Input (Reset)

Output	Q1	BOOL	Output State

Function

The behavior corresponds to **Table 43** (Standard Bistable Function Blocks) of the DIN EN 61131-3 standard.

The logic is described by the following truth table:

S	R1	Q1 (current)	Description
0	0	Q1 (old)	Hold state
0	1	0	Reset
1	0	1	Set
1	1	0	**Priority Reset**

Note: The initial state of the output variable `Q1` is `0` (FALSE) by default.

Logical Equation

According to the function block body in the standard (Table 43, 2a):

Q1 := NOT R1 AND (S OR Q1); If `R1` is TRUE, the expression `NOT R1` becomes FALSE, and thus `Q1` inevitably becomes `FALSE` (0), regardless of the state of `S`. This implements reset dominance.

See also

- E_RS (IEC 61499): The event-driven equivalent in IEC 61499. Note that there is no guaranteed "dominance" for concurrent events.

2.12.1.4 SR (Bistable, Set Priority) - IEC 61131-3

The function block **SR** is a bistable element (flip-flop) where setting takes precedence over resetting.

Interface

Type	Name	Data Type	Description
Input	S1	BOOL	Set Input
Input	R	BOOL	Reset Input
Output	Q1	BOOL	Output State

Function

The behavior corresponds to **Table 43** (Standard Bistable Function Blocks) of the DIN EN 61131-3 standard.

The logic is described by the following truth table:

S1	R	Q1 (current)	Description
0	0	Q1 (old)	Hold state
0	1	0	Reset
1	0	1	Set
1	1	1	**Priority Set**

Note: The initial state of the output variable `Q1` is `0` (FALSE) by default.

Logical Equation

In graphical representations (FBS), this is often depicted as an OR gate at the set input and a memory element, but logically, the state corresponds to:

Q1 := S1 OR (Q1 AND NOT R); *(Taking into account the dominance of S1 during simultaneous activation)*

More precisely according to the normative framework (Table 43, 1a):

(Table 43, 1a):

*(Table 43, 1a):```text Q1 := S1 OR (NOT R AND Q1); If `S1` is TRUE, then `Q1` is TRUE, regardless of whether `R` is true.

See also

- E_SR (IEC 61499): The event-driven equivalent in IEC 61499. Note that there is no guaranteed "dominance" for concurrent events.

]

2.12.1.5 TOF (Timer Off-Delay) - IEC 61131-3

The function block **TOF** (Timer Off-Delay) implements a timer off. The output remains active for a defined period after the input signal ceases.

Interface

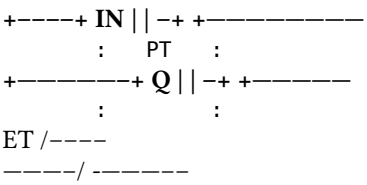
Type	Name	Data Type	Description
Input	IN	BOOL	Start Signal
Input	PT	TIME	Preset Time (Delay Time)
Output	Q	BOOL	Output Signal
Output	ET	TIME	Elapsed Time (Time Elapsed Since Falling Edge)

Function

The behavior corresponds to **Table 46** (Standard Function Blocks Timers) and **Figure 15 c)** (Turn-off Delay (TOF) Time Behavior) of the standard DIN EN 61131-3.

- **Activation:** When `IN` is `TRUE`, the output `Q` is immediately set to `TRUE`, and `ET` is held at 0.
- **Start Delay:** When `IN` transitions from `TRUE` to `FALSE` (falling edge), the time measurement `ET` starts. `Q` remains `TRUE`.
- **Process:** When `ET` reaches the value of `PT`, the output `Q` is set to `FALSE`.
- **Reset:** If `IN` becomes `TRUE` again before the time expires, `ET` is reset to 0, and `Q` remains `TRUE`.

Time Diagram (Principle)



2.12.1.6 TON (On-Delay) - IEC 61131-3

The **TON** (Timer On-Delay) function block implements an on-delay. The output is only activated when the input signal is present for a defined period.

Interface

Type	Name	Data Type	Description
Input	IN	BOOL	Start Signal
Input	PT	TIME	Preset Time (Delay Time)
Output	Q	BOOL	Output Signal
Output	ET	TIME	Elapsed Time

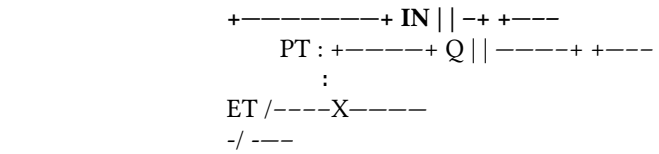
Output

Function

The behavior corresponds to **Table 46** (Standard Function Blocks Timers) and **Figure 15 b)** (Turn-on Delay (TON) Time Behavior) of the standard DIN EN 61131-3.

- **Start:** When `IN` equals `TRUE`, the time measurement begins at `ET`.
- **Sequence:** When `ET` reaches the value of `PT` and `IN` is still `TRUE`, the output `Q` is set to `TRUE`.
- **Hold:** `Q` remains `TRUE` as long as `IN` is `TRUE`.
- **Abort:** When `IN` changes to `FALSE` (before or after `PT` expires), `ET` is reset to 0 and `Q` becomes `FALSE`.

Time Diagram (Principle)



2.12.1.7 TP (Pulse) - IEC 61131-3

The **TP** (Timer Pulse) function block implements a pulse generator (monoflop). It generates an output pulse of defined duration.

Interface

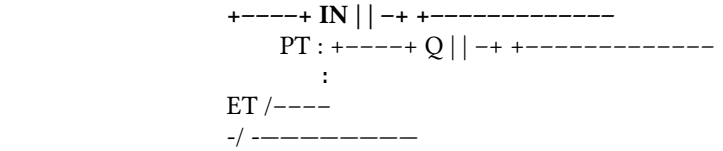
Type	Name	Data Type	Description
Input	IN	BOOL	Start signal (rising edge triggers pulse)
Input	PT	TIME	Preset Time (pulse duration)
Output	Q	BOOL	Output signal
Output	ET	TIME	Elapsed Time (time elapsed since pulse start)

Function

The behavior corresponds to **Table 46** (Standard Function Blocks Timers) and **Figure 15 a)** (Pulse (TP) Timing Behavior) of the standard DIN EN 61131-3.

- **Start:** When `IN` changes from `FALSE` to `TRUE` (rising edge), the output `Q` is set to `TRUE` and the timing measurement `ET` starts.
- **Duration:** The output `Q` remains at `TRUE` for the duration `PT`, regardless of whether `IN` changes back to `FALSE` during that time.
- **Reset:** After the time `PT` has elapsed (i.e., `ET >= PT`), `Q` reverts to `FALSE`.
- **Re-Trigger:** The pulse is **not** retriggerable. Edges at the input `IN` during pulse output (as long as `ET < PT` and `Q = TRUE`) are ignored.
- **Initialization:** `ET` is reset when `IN` `FALSE` and the timer has expired or has not yet started.

Timing Diagram (Principle)



3. static

3.1 Fonts

3.1.1 TGL Normschrift – _std Font Variants

3.1.1.1 Overview

The `_std` variants of the TGL Normschrift fonts fix the **U+0023 (#)** glyph, which in the original fonts incorrectly contains a "m/m" (millimeters) symbol instead of the standard number sign / hash character.

This makes the fonts usable in contexts where a proper # is required (e.g. programming, technical documentation, IEC 61499 function block diagrams).

3.1.1.2 Fonts

File	Style	Based on
`TGL 0-16_std.ttf`	Italic (75° slant)	TGL 0-16
`TGL 0-17_std.ttf`	Upright	TGL 0-17
`TGL 0-17 alt_std.ttf`	Upright (alternate)	TGL 0-17 Alt

3.1.1.3 Changes

New glyph at U+0023 (#)

A number sign was designed from scratch to match each font's style:

- **Stroke width** matches the font (207–208 units)
- **Round stroke caps** using the same quadratic Bézier curve pattern as the original glyphs (I, l, hyphen)
- **Height** is approximately 2/3 of the capital letter height, consistent with typical number sign proportions
- **TGL 0-16_std** (italic): vertical strokes are slanted at 75° from the baseline, matching the font's standard lean angle per DIN/TGL specification for slanted Normschrift
- **TGL 0-17_std / TGL 0-17 alt_std** (upright): all strokes are perpendicular to each other
- The glyph is constructed as a single non-overlapping outer contour with one inner counter (hole), avoiding rendering artifacts from overlapping paths

Original m/m glyph preserved at U+E000

The original "m/m" symbol that previously occupied the # position has been moved to **U+E000** (Unicode Private Use Area) so it remains accessible if needed.

Internal font names

All internal name table entries (family name, full name, PostScript name, unique ID) have `_std` appended to avoid conflicts when both original and modified fonts are installed on the same system.

3.1.1.4 Technical Details

- Format: TrueType (.ttf)
- Units per em: 2048
- Modified using Python `fonttools` library
- No other glyphs were changed
- All original font metrics, hinting, and licensing information are preserved

3.1.1.5 License

The original TGL Normschrift fonts are Copyright (c) 2010 by Peter Wiegel, released under GPL with font exception and OFL (Open Font License). The `_std` modifications are distributed under the same terms.